Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

# Progetto di Reti Logiche
# 2019-2020

Andrea Riva
10560217
Alessandro Sanvito
10578314

Marzo 2020

POLITECNICO
MILANO 1863

# Indice

**1 Introduzione** **1**

**2 Architettura** **2**
2.1 Control Unit (CU) . . . . . 2
2.2 RAM driver . . . . . 3
2.3 Loader . . . . . 3
2.4 Memory . . . . . 4
2.5 Encoder . . . . . 4

**3 Risultati sperimentali** **6**

**4 Test benches** **7**
4.1 Primo indirizzo di ogni working zone . . . . . 7
4.2 Offset della prima working zone . . . . . 7
4.3 Indirizzo non appartenente a nessuna working zone . . . . . 7
4.4 Esecuzioni successive senza cambio di working zones . . . . . 7
4.5 Esecuzioni successive con cambio di working zones . . . . . 7
4.6 Indirizzi vicini ai margini delle working zones . . . . . 8
4.7 Indirizzi vicini ai margini dello spazio di indirizzamento . . . . . 8
4.8 Fuzzing . . . . . 8

**5 Conclusioni** **9**

# 1. Introduzione

Come da specifica funzionale del progetto, l'obiettivo principale dell'architettura qui trattata è la codifica di un indirizzo ad 8 bit secondo il metodo delle working zones, che, durante l'esecuzione, rimangono immutate nei primi otto indirizzi di una RAM esterna al circuito, mentre l'indirizzo da codificare risiede nella nona posizione e la codifica deve essere scritta dall'hardware qui descritto nella decima. La codifica vera e propria confronta l'indirizzo con le working zones e, nel caso in cui l'offset con una di esse sia compreso tra 0 e 3, il risultato viene composto secondo la formula '1' & WZ_NUMBER & ONE_HOT_OFFSET, in cui WZ_NUMBER è codificato in binario naturale, altrimenti l'indirizzo rimane immutato. Da specifica viene inoltre garantita l'assenza di sovrapposizioni tra working zones e la loro corretta formattazione, con il bit più significativo sempre posto a '0'.
Un approccio di questo tipo si basa su un'idea espressa già in letteratura[1], che consiste nel codificare un indirizzo in termini di offset rispetto a pochi indirizzi di base più frequentemente usati, i quali vengono inferiti a runtime. Questa tecnica permette un considerevole risparmio energetico nella comunicazione con il bus, pur mantenendo una codifica più compatta rispetto al semplice one-hot grazie a considerazioni sulla località degli indirizzi.
A partire da queste premesse, si è scelto di progettare l'architettura trattata avendo come scopo la codifica di un gran numero di indirizzi consecutivamente rispetto a working zones immutate, ottimizzando la velocità della codifica rispetto al numero di cicli di clock e cercando di limitare il più possibile continue comunicazioni con la RAM in quanto dispendiose da un punto di vista energetico e temporale.

[1] E. Musoll, T. Lang and J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6, no. 4, pp. 568-572, Dec. 1998.

# 2. Architettura

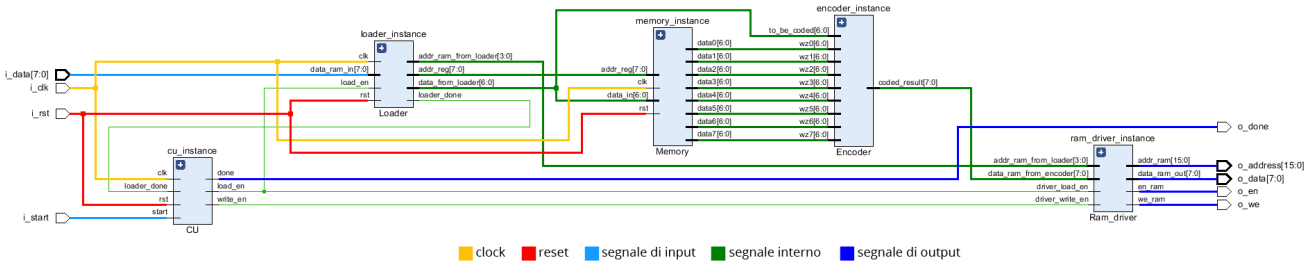



Figura 2.1: Schema funzionale dell'architettura, ingrandimento a fine documento

A partire dalle specifiche funzionali e non, abbiamo voluto individuare blocchi logici con compiti ben definiti, ispirandoci ad architetture note, in modo da garantire una migliore riusabilità e facilitare la fase di progettazione. In particolare le soluzioni più interessanti implementate nel design per rispondere ai requisiti non funzionali consistono nell'adozione di otto registri che permettono di fare caching degli indirizzi base delle working zones e di parallelizzare la vera e propria operazione di codifica.

## 2.1 Control Unit (CU)

Questo componente sequenziale è realizzato da una collezione di process e implementa la macchina a stati che controlla il protocollo di comunicazione del circuito verso l'esterno e coordina le varie parti che lo costituiscono per permetterne il corretto funzionamento. La macchina a stati è composta da 4 stati:

- wait_start: stato in cui si porta il componente dopo aver ricevuto il comando di reset (rst alto); mette il circuito in attesa del segnale di start;
- load: stato in cui si porta il componente dopo che il circuito ha ricevuto il segnale di start (start alto); attiva la fase di caricamento di dati (prima gli indirizzi base delle working zones, poi il dato da codificare) dalla memoria ai registri interni del circuito;
- write_result: stato in cui si porta il componente dopo che il circuito ha codificato il dato; attiva la fase di scrittura del risultato nella memoria;
- done_up: stato in cui si porta il componente dopo che il circuito ha scritto il dato codificato secondo la codifica working zones nella memoria; alza il segnale di done e attende che, come da protocollo, il segnale di start venga abbassato, per poi ritornare allo stato wait_start.

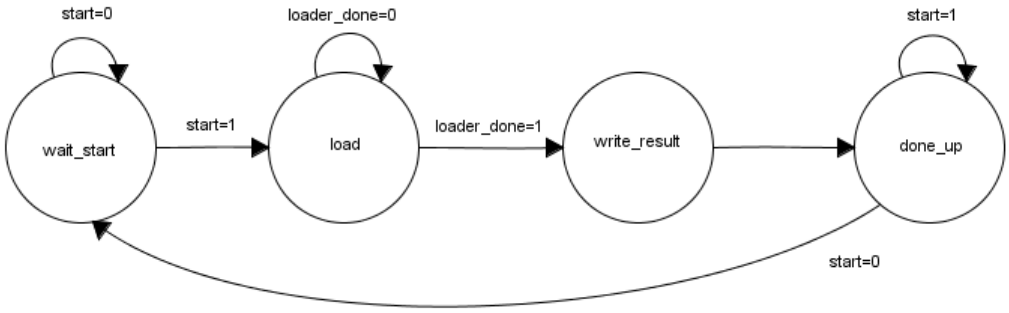



Figura 2.2: Diagramma degli stati della CU

## 2.2 RAM driver

È un componente combinatorio; è realizzato tramite una semplice architettura dataflow e fa da intermediario tra la RAM e i componenti che necessitano di accedervi (loader e encoder). Questa importante funzione si esplicita nell'elaborazione dei segnali di controllo provenienti dalla CU, che svolge invece un compito di coordinamento.

## 2.3 Loader

È un componente sequenziale; è realizzato da una collezione di process e si occupa di caricare dati dalla memoria esterna e renderli disponibili agli altri componenti interni del circuito. In particolare, in primo luogo il Loader carica uno dopo l'altro gli indirizzi base delle working zones e li rende disponibili al componente Memory affinchè li memorizzi nei suoi registri interni, utilizzando internamente una pipeline per ridurre i tempi di trasferimento dei dati dalla memoria esterna alla Memory interna. Successivamente, il Loader carica il dato da codificare con la codifica Working Zone e lo rende disponibile all'Encoder. La macchina a stati è composta da 11 stati:

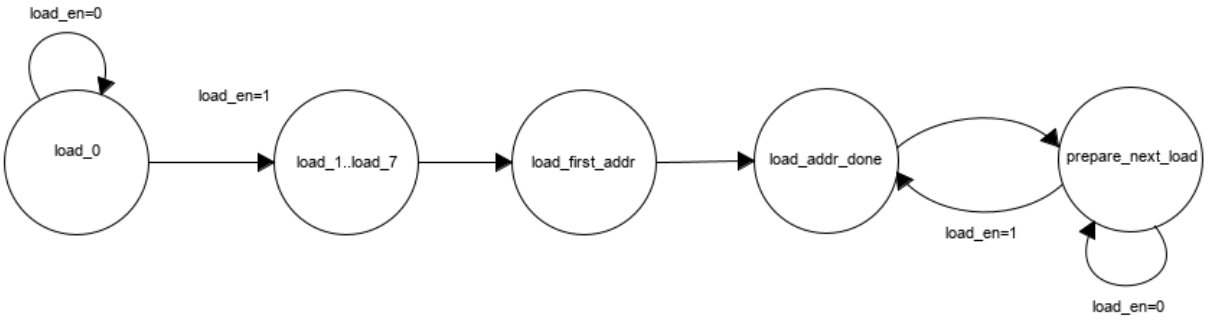



Figura 2.3: Diagramma degli stati del loader

- load_0: stato in cui si porta il componente dopo aver ricevuto il segnale di reset; il componente prepara il Ram Driver al caricamento dalla memoria esterna dell'indirizzo base della working zone 0 e attende il segnale di load_en dalla CU prima di procedere allo stato successivo;

- load_1: stato in cui si porta il componente dopo aver caricato dalla memoria esterna dell'indirizzo base della working zone 0; il componente attiva il caricamento dalla memoria esterna dell'indirizzo base della working zone 1 e inoltra alla Memory l'indirizzo base della working zone 0;
- load_X con $X \in [2, 7]$: stato in cui si porta il componente dopo aver caricato dalla memoria esterna dell'indirizzo base della working zone X-1; il componente attiva il caricamento dalla memoria esterna dell'indirizzo base della working zone X e inoltra alla Memory l'indirizzo base della working zone X-1;
- load_first_addr: stato in cui si porta il componente dopo aver caricato dalla memoria esterna dell'indirizzo base della working zone 7; attiva il caricamento dalla memoria esterna del dato da codificare secondo la codifica Working Zones e inoltra alla Memory l'indirizzo base della working zone 7;
- load_addr_done: stato in cui si porta il componente dopo aver caricato dalla memoria esterna del dato da codificare; rende disponibile il dato caricato all'Encoder e informa la CU dell'avvenuto caricamento alzando il segnale loader_done, dopodichè passa allo stato load_addr;
- prepare_next_load: stato in cui si porta il componente dopo aver terminato la sequenza di operazioni di lettura dalla memoria esterna; il componente prepara il Ram Driver al caricamento dalla memoria esterna del successivo dato da codificare e attende il segnale di load_en dalla CU prima di passare allo stato load_addr_done.

## 2.4 Memory

Come suggerisce il nome, questo componente sequenziale, realizzato con approccio dataflow e strutturale, memorizza in 8 registri da 7 bit, implementati con latch di tipo D, gli indirizzi base delle working zones caricati in sequenza dal Loader, per poi renderli disponibili all'elaborazione parallela dell'Encoder.
La memoria inverte gli indirizzi base delle working zones bit a bit prima di memorizzarli nei registri, in modo da fornire una codifica in complemento-1 all'Encoder, che solo successivamente trasformerà la stessa in una codifica a complemento a 2; negare bit a bit i dati prima di memorizzarli consente di effettuare tale operazione occupando l'area di un solo negatore a 7 bit senza inficiare le prestazioni (in quanto i dati provenienti dal Loader giungono già uno dopo l'altro).
La memoria opera con registri da 7 bit secondo l'assunzione, come da specifica, che il bit più significativo di ogni working zone è fisso ed è dunque futile la memorizzazione dello stesso.

## 2.5 Encoder

È un componente combinatorio; è realizzato da con approccio dataflow e strutturale e si occupa di codificare il dato che riceve dal Loader secondo la codifica Working Zones, utilizzando come indirizzi base quelli provenienti dalla Memory.
È costituito da 8 "Execution Lanes", alle quali è assegnata una working zone specifica, che viene confrontata con l'indirizzo da codificare tramite sottrazione per verificare l'appartenenza dello stesso, fornendo un offset a 2 bit, qualunque esso sia, in output. L'operazione di sottrazione sfrutta un semplice sommatore, il quale trae vantaggio dalla codifica in complemento-1 dei dati salvati in Memory aggiungendo un bit nella posizione meno significativa per avere una codifica in complemento-2.
Inoltre, nel caso in cui l'appartenenza sia verificata, l'Execution Lane interessata alza un segnale valid messo a disposizione del resto dell'Encoder. L'Encoder raccoglie i risultati dalle varie Execution Lanes. Se una di queste segnala di aver trovato un match con la relativa working zone, l'Encoder costruisce l'indirizzo codificato concatenando '1' all'identificativo binario della Execution Lane che ha trovato un match positivo, ottenuto

tramite una codifica prioritaria dei segnali di valid provenienti dalle lanes, e alla codifica one hot dell'offset rispetto all'indirizzo base della working zone riportato dalla Execution Lane.
In caso nessuna Execution Lane riporti un match con la rispettiva working zone, l'Encoder restituisce semplicemente l'indirizzo da codificare, come da specifica.
Avendo realizzato l'encoder come componente combinatorio e avendo utilizzato 8 Execution Lanes in parallelo, l'operazione di codifica del dato secondo la codifica Working Zones avviene molto velocemente, in particolare in meno di un ciclo di clock.

# 3. Risultati sperimentali

La sintesi effettuata dal software Vivado 2019.2 su board xc7a200tfbg484-1 vede l'impiego di 103 LUTs come logica e di 71 registri come flip flops. La maggior parte delle risorse in uso viene impiegata per realizzare le 8 execution lanes di codifica e i rispettivi moduli accessori (ad esempio, i registri interni che memorizzano gli indirizzi base delle working zones). Dal report di sintesi di Vivado si nota inoltre la scelta del software di sintesi di realizzare la codifica degli stati delle due macchine a stati presenti (control unit e loader) tramite una codifica one-hot.
L'utilizzo di una quantità maggiore di risorse per la realizzazione delle execution lanes viene ampiamente ripagato dalle ottime performance che caratterizzano l'architettura proposta. In particolare, per ogni esecuzione, la prima codifica richiede 12 cicli di clock (dallo start alzato al done alzato), mentre le codifiche successive richiedono solamente 4 cicli di clock, consentendo quindi su lunghe esecuzioni di ridurre sensibilmente il tempo necessario alla codifica rispetto a caricare ad ogni codifica gli indirizzi base delle working zones.

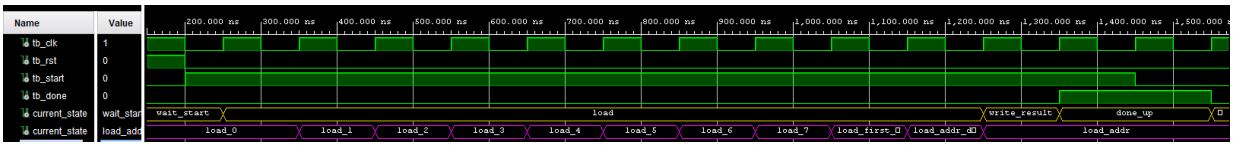



Figura 3.1: Forme d'onda della prima esecuzione

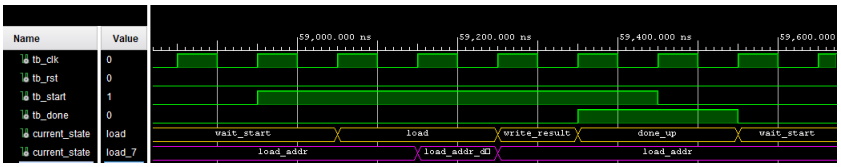



Figura 3.2: Forme d'onda di esecuzioni successive alla prima, senza rst

# 4. Test benches

## 4.1 Primo indirizzo di ogni working zone

Il test bench verifica che il circuito codifichi correttamente 8 indirizzi, coincidenti con gli indirizzi base delle 8 working zone, inviando un segnale di reset prima di codificare ciascun indirizzo. Lo scopo del test bench è la verifica del corretto calcolo del numero della working zone a cui appartiene l'indirizzo da codificare.

## 4.2 Offset della prima working zone

Il test bench verifica che il circuito codifichi correttamente 4 indirizzi, coincidenti con i 4 offset possibili all'interno della prima working zone, inviando un segnale di reset prima di codificare ciascun indirizzo. Lo scopo del test bench è la verifica del corretto calcolo dell'offset con codifica one-hot.

## 4.3 Indirizzo non appartenente a nessuna working zone

Il test bench verifica che un indirizzo non appartenente ad alcuna working zone e lontano dai confini delle stesse venga codificato correttamente. Lo scopo del test bench è la verifica della corretta identificazione della non appartenenza di un indirizzo ad alcuna working zone.

## 4.4 Esecuzioni successive senza cambio di working zones

Il test bench verifica che i 12 indirizzi testati nei test benches precedenti ($8 + 4 + 1 - 1$, dove - 1 è dovuto al fatto che l'indirizzo con offset 0 della prima working zone è presente in entrambi i primi due test benches) vengano codificati correttamente senza che venga inviato un segnale di reset tra le richieste di codifica di ciascun indirizzo. Lo scopo del test bench è la verifica del corretto funzionamento del circuito quando vengono richieste più conversioni per esecuzione a partire da indirizzi che, singolarmente, sono stati codificati correttamente.

## 4.5 Esecuzioni successive con cambio di working zones

Il test bench verifica che 2 indirizzi vengano codificati correttamente cambiando gli indirizzi base delle working zones e inviando il segnale di reset tra una codifica e l'altra. Lo scopo del test bench è la verifica della capacità del circuito di resettarsi in modo corretto.

## 4.6 Indirizzi vicini ai margini delle working zones

Il test bench verifica che alcuni indirizzi vicini ai margini delle working zones vengano codificati correttamente. In particolare, vengono testati:

- un indirizzo con offset 3 rispetto ad una working zone e coincidente con l'indirizzo precedente a quello di un'altra working zone;
- un indirizzo con offset 4 rispetto ad una working zone e coincidente con l'indirizzo precedente a quello di un'altra working zone;

Lo scopo del test bench è la verifica del corretto funzionamento del circuito in presenza di indirizzi che potrebbero essere associati ad una working zone errata in quanto vicina.

## 4.7 Indirizzi vicini ai margini dello spazio di indirizzamento

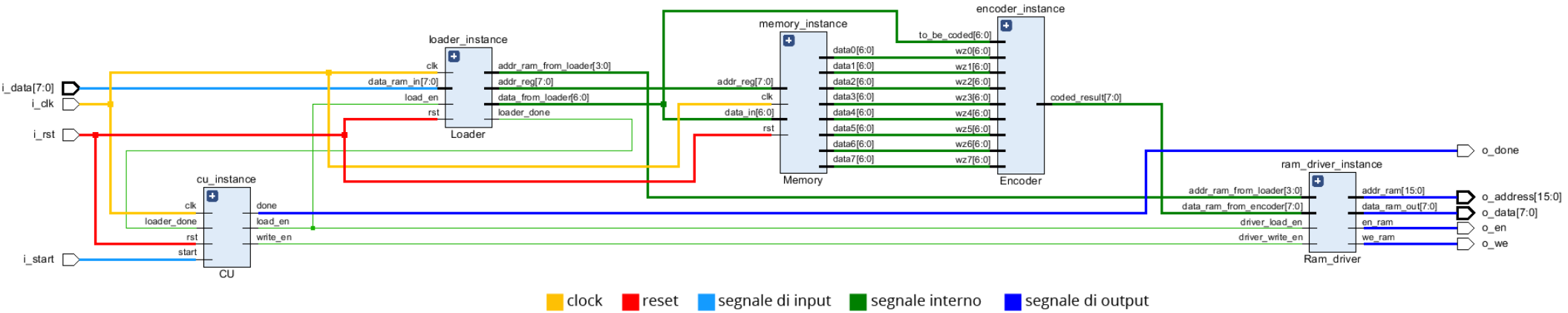
Il test bench verifica che il primo (0) e l'ultimo indirizzo (127) dello spazio di indirizzamento consentito (7 bit) vengano codificati correttamente, sia che appartengano o che non appartengano a una working zone. Lo scopo del test bench è la verifica del corretto funzionamento del circuito per l'intero spazio di indirizzamento consentito.

## 4.8 Fuzzing

Il test bench verifica che una lunga sequenza casuale di indirizzi venga codificata correttamente, generando casualmente gli indirizzi base delle working zones e inserendo casualmente operazioni di modifica delle working zones e reset. Lo scopo del test bench è la verifica del corretto funzionamento del circuito con una quantità elevata di conversioni.

# 5. Conclusioni

Come ben dimostrato dai risultati sperimentali e dal report di sintesi, l'architettura qui descritta permette prestazioni considerevoli negli scenari d'uso più tipici, grazie alla già trattata adozione di registri interni e alla parallelizzazione della codifica vera e propria. Si è inoltre sperimentalmente verificato che la velocità di esecuzione dimostrata dall'architettura permetta un considerevole aumento della frequenza di clock (almeno di un ordine di grandezza), permettendo quindi ulteriori guadagni nel tempo di esecuzione. In conclusione si considera l'obiettivo di un'ottimizzazione per gli scenari d'uso più tipici pienamente raggiunto.
Una possibile ottimizzazione potrebbe riguardare la fase di caricamento degli indirizzi base delle working zones: attualmente si caricano tutti gli 8 indirizzi base per poi confrontarli con l'indirizzo da codificare, mentre sarebbe possibile interrompere il caricamento non appena una execution lane rileva un match; sarebbe necessario aggiungere un bit per ciascun registro per tenere conto se il registro contiene un dato valido o non è ancora stato caricato, per poi modificare la macchina a stati del loader per consentire di riprendere il caricamento degli indirizzi base da una posizione successiva alla prima, in caso di successive richieste di codifica.


i_data[7:0]
i_clk
i_rst
i_start
loader_instance
clk
data_ram_in[7:0]
load_en
rst
addr_ram_from_loader[3:0]
addr_reg[7:0]
data_from_loader[6:0]
loader_done
Loader
memory_instance
addr_reg[7:0]
clk
data_in[6:0]
rst
Memory
encoder_instance
to_be_coded[6:0]
data0[6:0]
data1[6:0]
data2[6:0]
data3[6:0]
data4[6:0]
data5[6:0]
data6[6:0]
data7[6:0]
wz0[6:0]
wz1[6:0]
wz2[6:0]
wz3[6:0]
wz4[6:0]
wz5[6:0]
wz6[6:0]
wz7[6:0]
coded_result[7:0]
Encoder
cu_instance
clk
loader_done
rst
start
done
load_en
write_en
CU
ram_driver_instance
addr_ram_from_loader[3:0]
data_ram_from_encoder[7:0]
driver_load_en
driver_write_en
addr_ram[15:0]
data_ram_out[7:0]
en_ram
we_ram
Ram_driver
o_done
o_address[15:0]
o_data[7:0]
o_en
o_we
clock
reset
segnale di input
segnale interno
segnale di output