Progetto Reti Logiche (Anno 2019-2020)

Componente elettronico "Working Zone"

Etion Pinari – Codice Persona 10619348; Matricola 900193

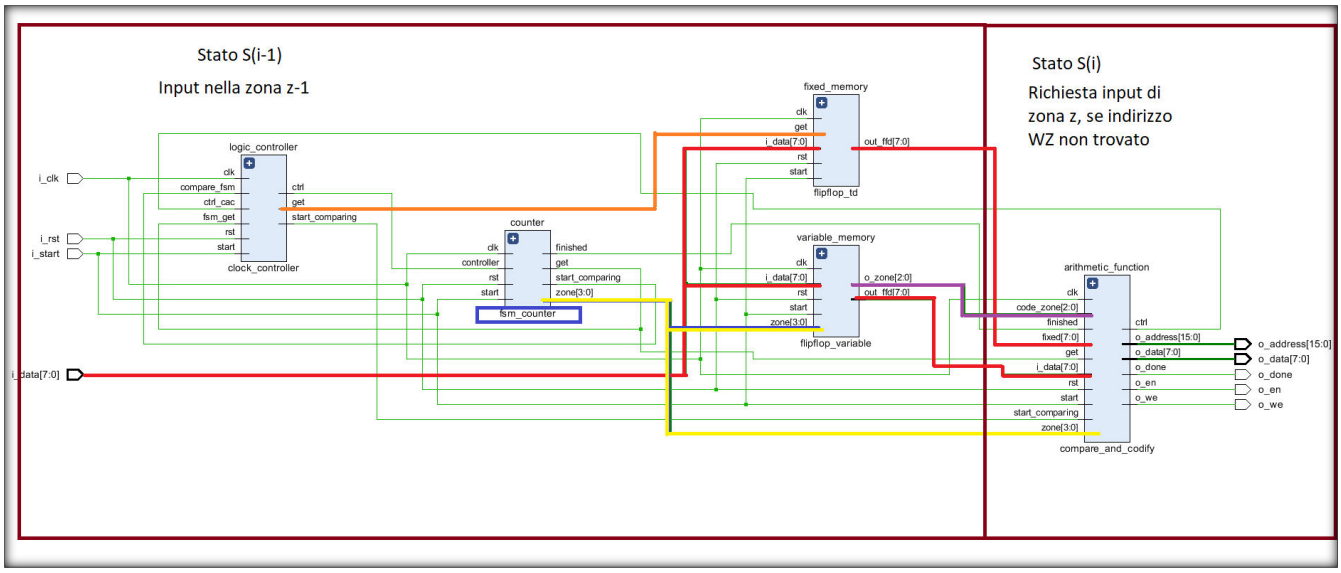
• Introduzione

Il componente elettronico ispirato al metodo di codifica "Working Zone" è stato ideato e progettato in 5 sottocomponenti con ciascuno una parte della logica implementativa comunicanti tra di loro.

Concettualmente diviso in due cicli di clock, il primo ciclo di clock assicura la carica dell'indirizzo richiesto, e il secondo ciclo di clock paragona i due indirizzi salvati nei flip-flop, codificando oppure richiedendo il prossimo indirizzo in memoria. Perciò la scelta più intuitiva era quella di utilizzare due flip-flop[1-2], una macchina FSM[3] per richiedere la zona in memoria, un'unità di tipo ALU[4] ed un'ulteriore macchina[5] con un segnale interno che gestisce i segnali di controllo prodotti dagli altri sottocomponenti.

Il componente è robusto ma richiede per ogni indirizzo due cicli di clock, però si ferma prima se trova l'indirizzo nella WZ. Inoltre per occupare meno spazio nel FPGA si è deciso di non utilizzare 9 flip flop, ed inoltre per facilitare la logica si gestisce il segnale di start basso in modo equivalente a quello di reset, alto.

• Elaborazione



Una simulazione sintetica del componente è la seguente:

In rosso è il vettore input appartenente all'ultima zona richiesta nella memoria RAM. Esso viene salvato in uno dei due registri:

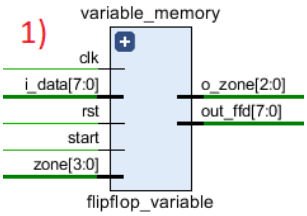
-in fixed_memory[2], in base al flag arancione *get*, che è controllato dagli stati del counter[3] (macchina FSM) e poi sincronizzato e tenuto alto nel ciclo di clock giusto, ossia quando l'i_data è effettivamente aggiornato, tramite logic_controller[5].

-in variable_memory[3] se il primo bit del vettore giallo *zone* è pari a 0 (ossia abbiamo i valori $[0,dc,dc,dc]_{bin} = [0;7]_{dec}$.

In fine arithmetic_function[4] trova la differenza fra i due indirizzi salvati una volta che il flag *start_comparing* diventa alto. Se non trova l'indirizzo nella WZ, utilizza il segnale del counter[3] chiamato *zone* per richiedere il prossimo indirizzo in memoria. Se invece trova l'indirizzo nella WZ, allora utilizza *code_zone*, ossia la zona salvata nel registro variable_memory[1], per poi codificare e salvare in memoria l'indirizzo giusto, avendo alzato il segnale *o_we* e *o_done*.

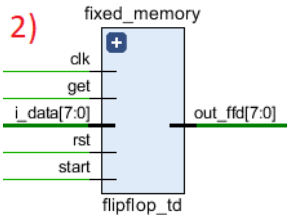
- # Architettura

1) Variable_memory (Flip Flop tipo D)



Questo flip flop ha una logica semplice per cui se la zona in *zone*[3:0] ha il primo bit pari a 1 (cioè non si trova negli indirizzi da 0 a 7), non aggiorna i dati che si trovano dentro di esso.
Inoltre in uscita riporta anche la zona in cui si trovava l'indirizzo salvato, utilizzato poi per la codifica nel componente ALU.

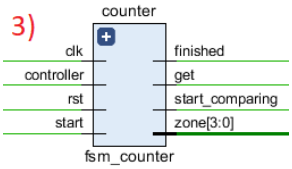
2) Fixed_memory (Flip Flop Tipo D)



Un semplice flip flop di tipo D, che come quello precedente, cancella tutti i dati salvati se *rst* è alto, oppure *start* è basso.

Il segnale *get*, gli indica quando salvare l'indirizzo.

3) FSM_Counter (FSM con segnali di controllo)



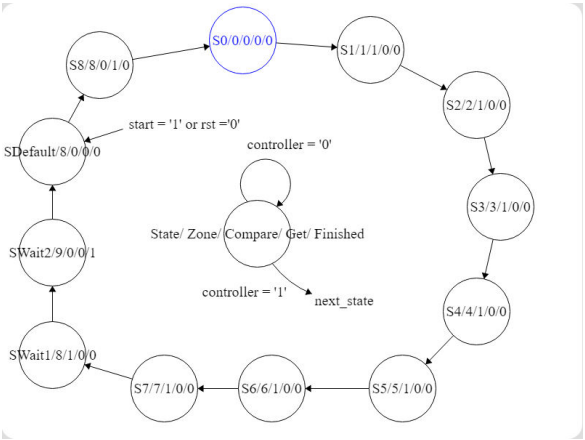
Il cervello del componente, counter ha nel suo interno tutti segnali di controllo che comandano le altre parti del componente.

Get è il segnale visto sopra in fixed_memory[2].

Start_comparing segnala all'ALU in quale ciclo di clock entrambi gli indirizzi sono aggiornati alla zona giusta e quando può iniziare a fare la differenza ed eventualmente codificare il risultato.

Finished indica se tutti gli 8 indirizzi di memoria sono stati controllati, così l'ALU può scrivere in memoria l'indirizzo non codificato giusto.

Zone ha i valori da 0 a 8 ed è collegato all'unità ALU (e nel flipflop variable[1]) per richiedere il prossimo indirizzo in memoria.



La freccia indica lo stato successivo. -Compare- indica il segnale start_comparing

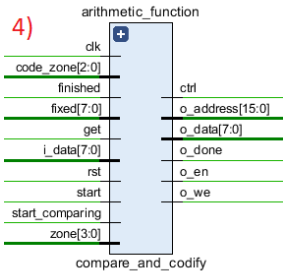
I valori interessanti sono quelli che cambiano nello stato:

-S1- una volta letto l'indirizzo in posizione 8, e nell'indirizzo 0, *start_comparing* diventa '1' e rimane invariato fino a SWait1, che dà il tempo necessario all'ALU per paragonare i due valori e infine scriverli in memoria.

-SWait2- porta il valore di *start_comparing* a '0', e *finished* a '1'.

Get è pari a ‘1’ solo nello stato S8. La macchina segue la direzione indicata dalle frecce ogni volta che in input il segnale *controller* viene portato a ‘1’ altrimenti si rimane nello stesso stato.

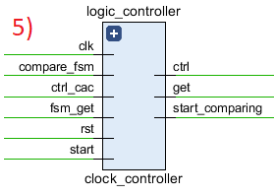
4) Compare_and_Codify (ALU e richiesta indirizzi)



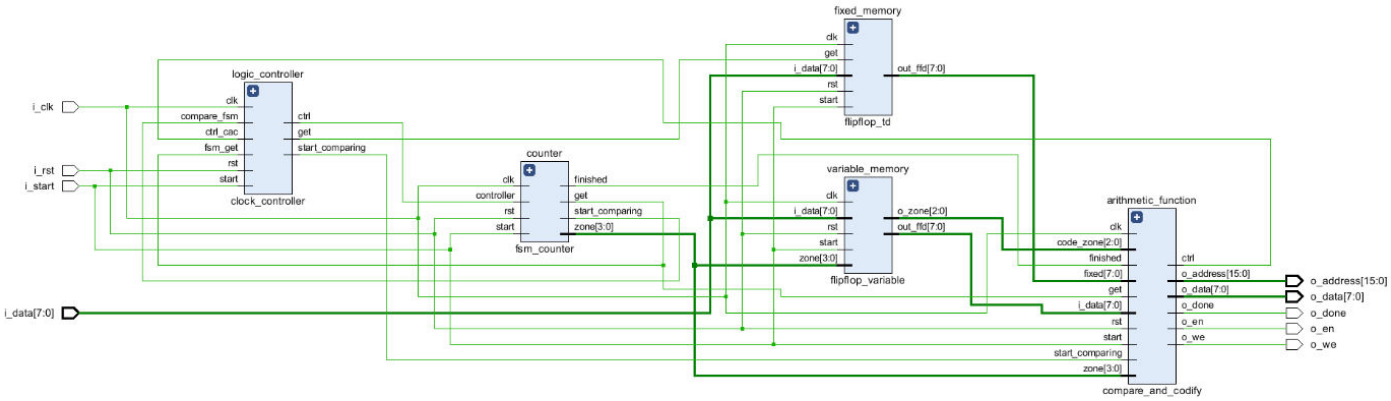
Come precedentemente accennato, l’unità ALU ha il compito di trovare la differenza tra due indirizzi, e nel caso essa non sia tra (0-3), richiede il prossimo indirizzo in memoria. *Code_zone* è il valore della zona del valore salvato nel flipflop variabile, mentre *zone* è l’indirizzo successivo da richiedere (ram(zone)).

Una volta finito il paragono degli indirizzi, manda un segnale di controllo (ctrl) al FSM per indicare se deve procedere al prossimo stato oppure rimanere in attesa.

5) Logic_controller (introduzione di ritardi)



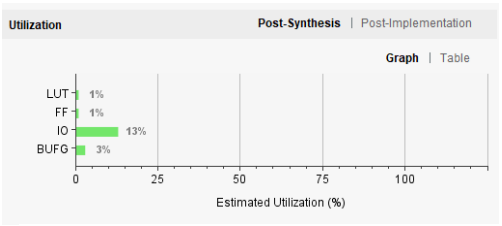
Questo componente si trova a metà strada tra l’ALU e il FSM, e non fa altro che ritardare di un ciclo di clock i segnali di controllo tramite un segnale interno messo in AND-logico con i segnali di controllo e in seguito, questo segnale interno viene aggiornato, diventando il suo contrario, da alto in basso e viceversa. I segnali di controllo quindi non si aggiornano ogni ciclo di clock, ma ogni due cicli di clock.



1) Vista completa del componente

- # Risultati sperimentali

Possiamo notare dal report di sintesi, usando la FPGA xc7a200tfbg484-1, che il componente FSM[3] è stato sintetizzato come una macchina a stati codificati in ONE-HOT. Inoltre è interessante anche la quantità di registri, sommatori e multiplexxers utilizzati e l'utilizzo del FPGA, riportati nella prossima immagine:



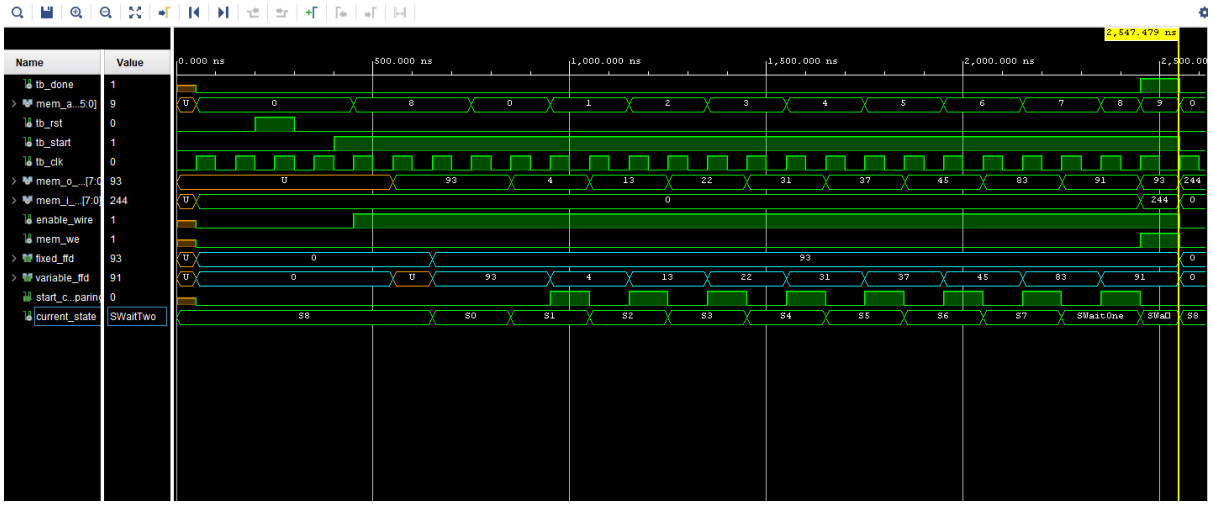
1) Utilizzo FPGA

```
+---Adders :
      3 Input      8 Bit       Adders := 1
      2 Input      3 Bit       Adders := 1
+---Registers :
                  16 Bit    Registers := 1
                   8 Bit    Registers := 2
                   7 Bit    Registers := 1
                   3 Bit    Registers := 1
                   1 Bit    Registers := 8
+---Muxes :
      4 Input     16 Bit        Muxes := 1
     13 Input     13 Bit        Muxes := 1
      2 Input     13 Bit        Muxes := 13
      3 Input      8 Bit        Muxes := 1
      2 Input      8 Bit        Muxes := 2
      3 Input      4 Bit        Muxes := 1
      2 Input      4 Bit        Muxes := 2
      4 Input      4 Bit        Muxes := 1
     13 Input      4 Bit        Muxes := 1
      2 Input      1 Bit        Muxes := 1
      3 Input      1 Bit        Muxes := 1
     13 Input      1 Bit        Muxes := 1
```

2) Utilizzo dei diversi componenti

- # Test benches

Per vedere la validità del componente si sono eseguiti dei test casuali generati in Python con più di 1.5 milioni di casi, che testavano sia i diversi *reset* alti, sia l'esecuzione continua del componente disegnato, secondo la specifica. Tutti i testbench sono stati passati sia in Behavioural Simulation, che in Post-Synthesis Functional Simulation. Due test interessanti generati manualmente sono quelli per cui l'indirizzo in Working Zone si trova in RAM(7), che produce la complessità temporale peggiore del componente, sia il caso in cui l'indirizzo in Working Zone si trova in RAM(0), producente la complessità temporale ottima.

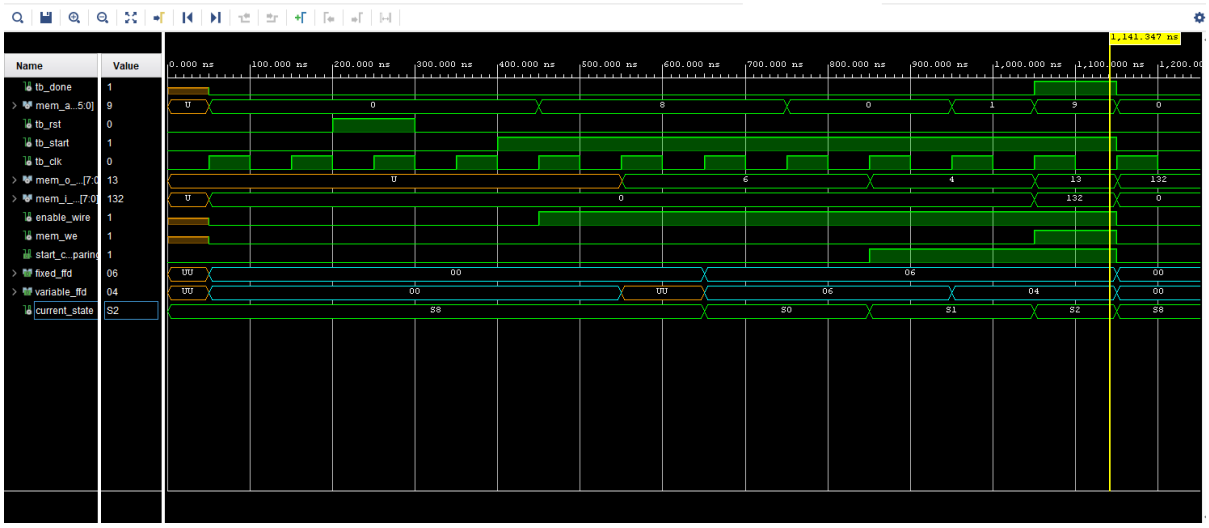


2)Caso Pessimo

Nel caso 1) l'indirizzo codificato è "1 && Zona_7 &&OneHot(93-91)" , ossia

"1 & 111 & 0100"$_{bin}$ = 244$_{dec}$. Qui il tempo d'inizio, quando *start* diventa '1' è 450ns e quello finale è 2550 ns. Perciò possiamo dire che servono all'incirca 20 cicli di clock per un'esecuzione completa.

Questo è il prezzo da pagare per avere maggiore robustezza, aspettando un intero ciclo di clock, prima di aggiornare i dati.



3) Caso Ottimo

2) Invece nel caso ottimo possiamo notare che la esecuzione inizia a 350ns e finisce a 1150ns quindi servono 7 cicli di clock, come minimo per avere il risultato. Da notare anche quando viene riportato alto il segnale start_comparing, all'inizio dello stato S1. Questa scelta implementativa è stata fatta per assicurarsi di non avere nessun bug anche nel caso di ritardo o desincronizzazione di segnali in entrata.

In uscita vediamo 132_{dec} ossia 1 & 000 & 0100.

- # Conclusione

Come accennato diverse volte, si è deciso di avere una macchina più lenta con una logica implementativa semplice, seguendo sempre la specifica ma che ci può garantire una sicurezza maggiore anche su ritardi significativi.

Il componente è sintetizzabile e supera tutti i test bench di specifica sia in Behavioural sia in Post-Sintesi.