Prova Finale di Reti Logiche

AA. 2019/2020

Stefano Dalla Longa, pID 10535602
Nicolò Brandolese, pID 10531144

Politecnico di Milano, scaglione prof. Fornaciari

Indice

1 Introduzione **2**

2 Descrizione della macchina e dei processi **2**

3 Descrizione degli stati **3**

4 Test eseguiti **5**

5 Limitazioni e possibili ottimizzazioni **8**
5.1 Scalabilità del componente 8
5.2 Segnale di reset 8
5.3 Ottimizzazione dei dati nella RAM 8

6 Conclusioni **9**

1 Introduzione

Abbiamo deciso di risolvere questo problema in maniera sequenziale: provando ogni WZ data una alla volta e riportando un risultato positivo al primo successo oppure un risultato negativo dopo l'ottavo fallimento. In questo modo l'implementazione è scalabile ad un (teorico) infinito numero di WZ diverse con minimale aumento di volume; a pagarne sono le prestazioni nel caso pessimo (indirizzo appartenente all'ultima WZ o non appartenente a nessuna WZ), ma, dato l'utilizzo di WZ, si assume che il caso pessimo sia poco frequente. Per ulteriori speculazioni sulle performance della soluzione implementata, si guardi la sezione *Ottimizzazione dei dati nella RAM*.

2 Descrizione della macchina e dei processi

L'implementazione si basa sulla scomposizione in quattro sottoproblemi:

1. comunicazione con la RAM per la lettura dei dati
2. computazione per determinare l'appartenenza dell'indirizzo base ad una working zone
3. gestione del flusso d'esecuzione in base al risultato della computazione
4. comunicazione con la RAM per la scrittura del risultato

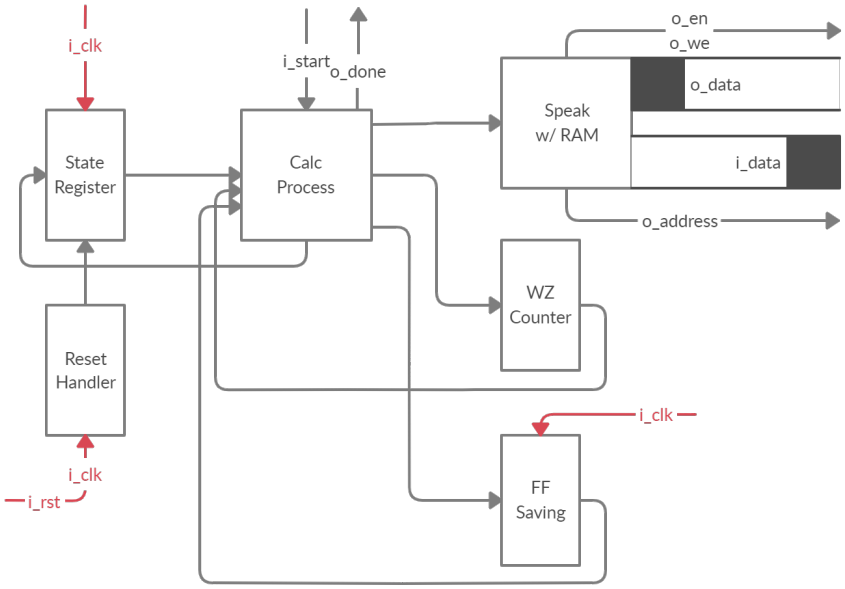
La scelta di risoluzione è stata la creazione di una macchina a stati finiti, in particolare una macchina di Mealy, basata su tre macroprocessi:

- *state_register*: ad ogni fronte di salita del clock propaga il valore del prossimo stato della macchina *next_state* al registro dello stato corrente *current_state*. Reagisce al segnale di reset eseguendo un reset sincrono, ovvero assegnando invece il valore di *START_IDLE* a *current_state* sul fronte di salita del clock.
- *speak_with_RAM*: gestisce i valori dei segnali in input e output, ad eccezione di *o_done* il quale è gestito da *calc_process*.
- *calc_process*: esegue i calcoli per determinare se l'indirizzo base appartiene alla working zone considerata

Al quale si aggiungono tre ulteriori processi dedicati alla gestione degli elementi di memoria interni:

- *reset_handler*: tiene traccia della richiesta di reset da parte della RAM tramite il segnale in input *i_rst* e permette di eseguire un reset sincrono. Si veda la sezione *Limitazioni e possibili ottimizzazioni - segnale di reset* per ulteriori chiarimenti sul funzionamento della procedura di reset.
- *wz_counter_process*: aggiorna sul fronte di discesa del clock il contatore *wz_counter* delle working zone già controllate. Il contenuto di tale contatore è aggiornato ad ogni nuova esecuzione della macchina, ed è dotato di segnale di reset. L'aggiornamento sul fronte di discesa anziché di salita del clock risolve diversi problemi di sincronia tra il contatore e i successivi stati, dal momento che il suo valore deve disponibile allo stato immediatamente successivo all'aggiornamento del suo contenuto.
- *FF_saving*: gestisce i flip-flop relativi a quattro segnali da memorizzare. Non è stato necessario fornirlo di segnale di reset.

Una rappresentazione schematica della macchina è fornita dalla figura seguente:

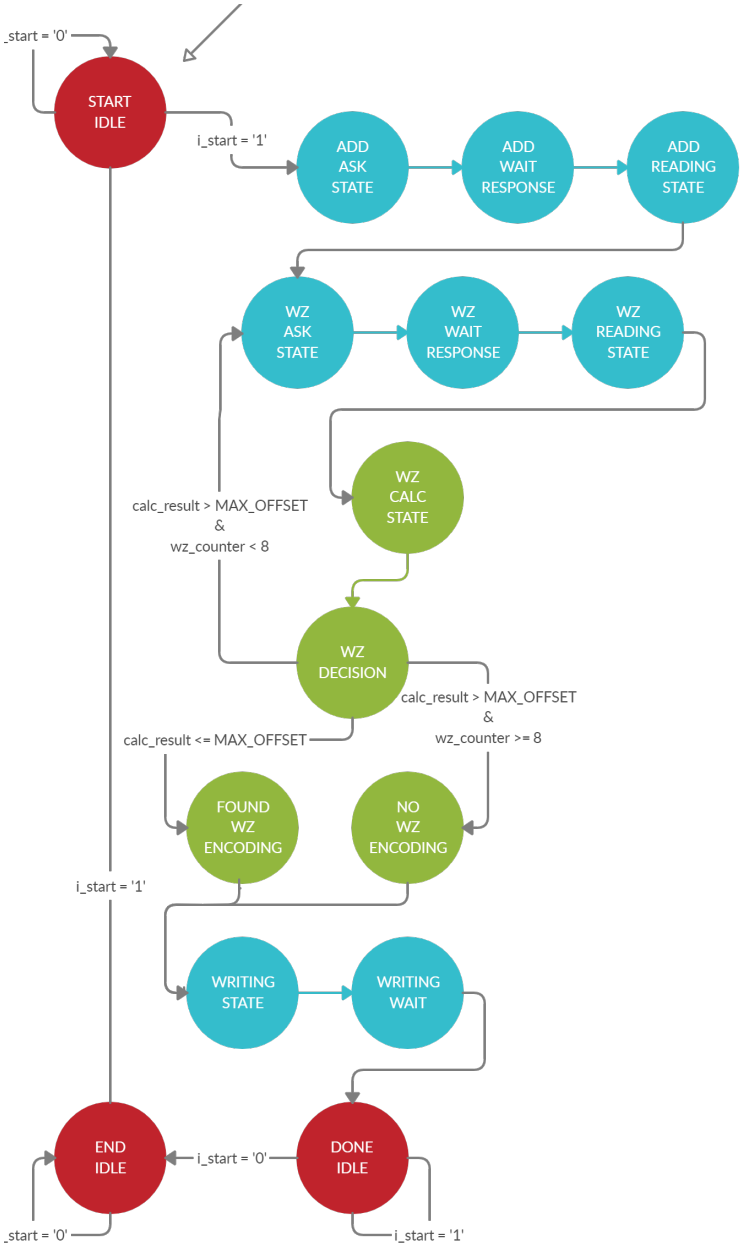


Abbiamo scelto di separare *calc_process* e *speak_with_RAM* per ragioni di leggibilità, dal momento che da un punto di vista teorico entrambi costituiscono una componente combinatoria della stessa macchina a stati finiti, e perciò sarebbe stato possibile accorparli nel medesimo processo. Dal momento però che VHDL proibisce di modificare lo stesso segnale da processi diversi, la gestione del flusso modificando il valore di *next_state* è affidata esclusivamente a *calc_process*.

3 Descrizione degli stati

Il flusso d'esecuzione e la transizione dello stato corrente della macchina sono schematizzati nella figura seguente e usano questa convenzione:

- gli stati in blu sono tutti e soli gli stati nei quali opera *speak_with_RAM* e sono progettati per durare un ciclo di clock;
- gli stati in verde sono gli stati di calcolo e codifica da parte di *calc_process*, anch'essi progettati per cambiare ad ogni ciclo di clock;
- gli stati in rosso sono stati di idle gestiti da *calc_process*, progettati per arrestare il flusso di esecuzione ed eventualmente ricominciare la computazione in base al valore di *i_start*.



Un breve riassunto del funzionamento di ogni stato è il seguente:

START_IDLE: stato iniziale. La macchina resta in questo stato finché `i_start` vale 0 e vi ritorna quando riceve un segnale di reset;

ADD_ASK_STATE: richiede alla RAM l'indirizzo base;

ADD_WAIT_RESPONSE: stato di attesa per permettere alla RAM di processare la richiesta;

ADD_READING_STATE: salva il valore dell'indirizzo base ricevuto in input dalla RAM nel registro `base_address`;

WZ_ASK_STATE: richiede alla RAM l'n-esima working zone, dove n è il valore del registro contatore `wz_counter`;

WZ_WAIT_RESPONSE: stato di attesa per permettere alla RAM di processare la richiesta;

WZ_READING_STATE: salva il valore dell'n-esima working zone ricevuta in input dalla RAM nel registro `wz_address`;

WZ_CALC_STATE: calcola se l'indirizzo base appartiene alla working zone corrente, con l'operazione tra registri `calc_result` = `base_address` − `wz_address`;

WZ_DECISION: in base al valore di `calc_result` in grado di capire se l'indirizzo base appartiene alla working zone considerata. In caso affermativo, imposta come prossimo stato `FOUND_WZ_ENCODING`, mentre in caso contrario imposta come prossimo stato `NO_WZ_ENCODING` se la working zone considerata era l'ultima contenuta nella RAM, oppure ritorna nello stato `WZ_ASK_STATE` se la RAM contiene altre working zone da controllare[1];

FOUND_WZ_ENCODING: codifica la parola da scrivere nella ram nel registro `encoded_res` secondo la convenzione usata in caso di fallimento;

NO_WZ_ENCODING: codifica la parola da scrivere nella ram nel registro `encoded_res` secondo la convenzione usata in caso di successo;

WRITING_STATE: scrive nella RAM il contenuto di `encoded_res`;

WRITING_WAIT: stato di attesa per permettere alla RAM di processare la richiesta;

DONE_IDLE: notifica alla RAM la fine della computazione mantenendo al valore 1 il segnale `o_done`. La macchina resta in questo stato finché il segnale `i_start` vale 1, dopodiché va nello stato di `END_IDLE`;

END_IDLE: imposta a 0 il segnale `o_done`. La macchina aspetta l'innalzamento del segnale `i_start`, in seguito al quale torna in `START_IDLE`.

4 Test eseguiti

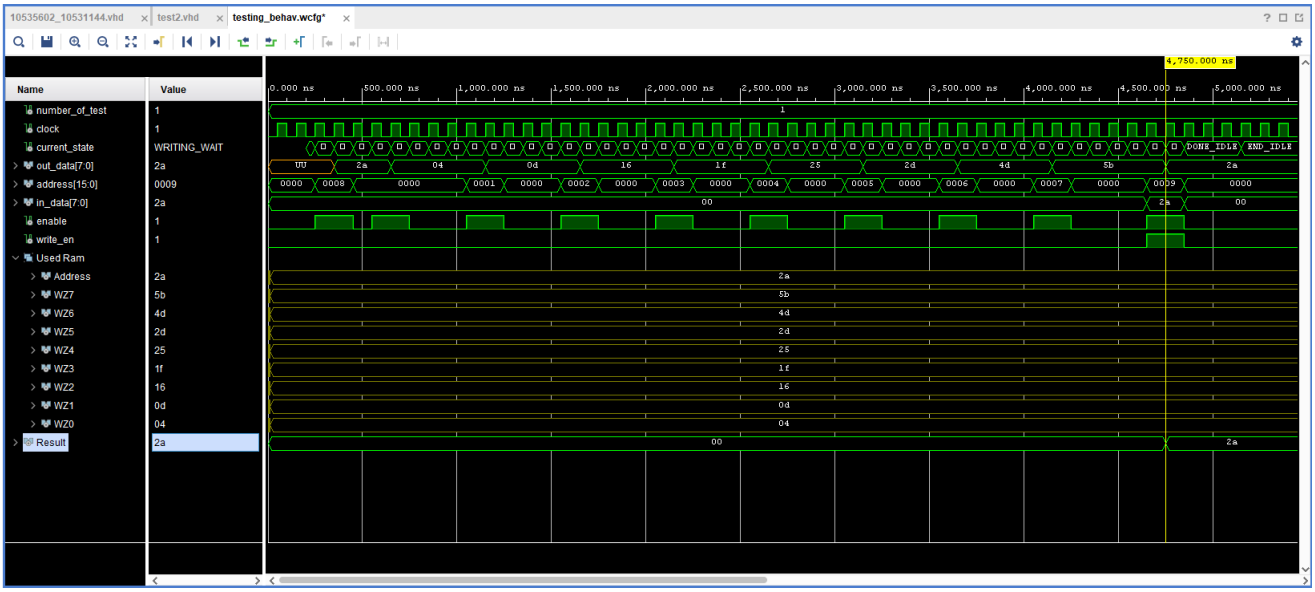
Per controllare il corretto funzionamento del circuito sono state svolte un centinaio di simulazioni, generate automaticamente (tranne pochi casi limite) e controllate manualmente. È stato usato uno script di python 3 per generare randomicamente il contenuto della RAM, scriverlo in un file di testo e scrivere il risultato da visualizzare in memoria, in codifica binaria, esadecimale e decimale, sotto forma di commento VHDL. È riportato il codice sorgente del programma, il quale accetta da riga di comando due elementi: il numero iniziale e il numero finale del test (riadattato per compatibilità con la nostra batteria di test).

[1] Quest'operazione costituisce il critical path della rete combinatoria

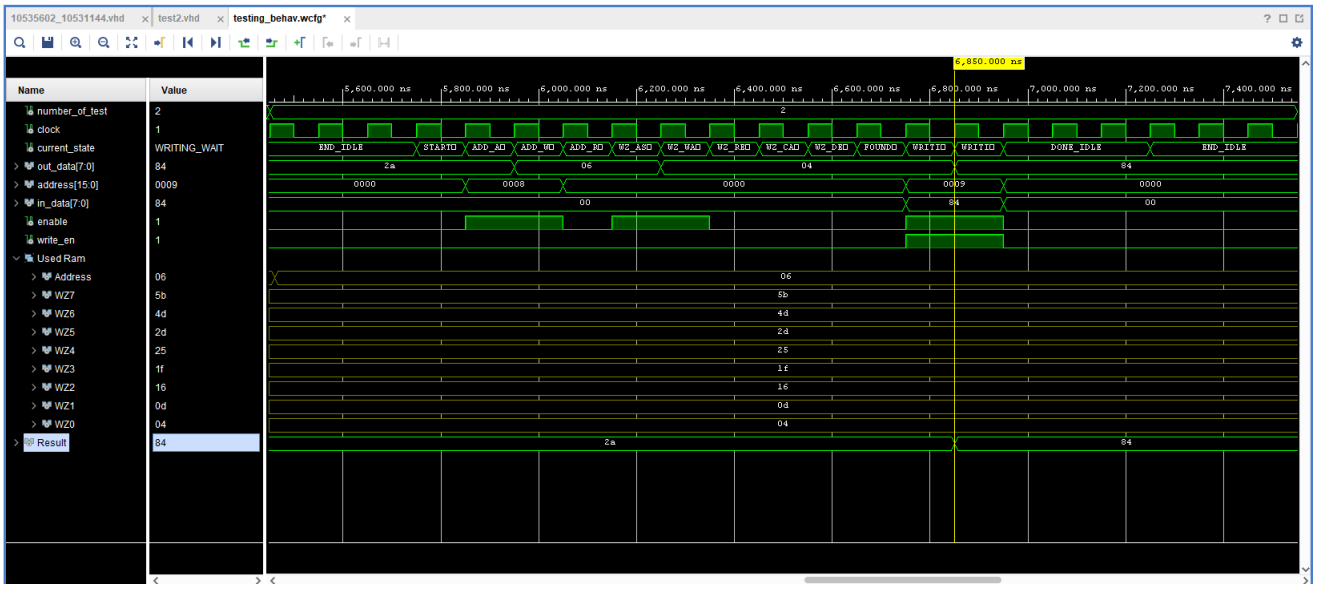
```
import sys
import random

starting = int(sys.argv[1]) #starting element
ending = int(sys.argv[2]) #ending element

f = open("random_test.txt", "w")

for i in range(ending - starting + 1):
  addr_list = []
  f.write("\t\t\t\twhen %d =>\n\n" % (starting + i))

  for j in range(9):
    random_num = random.getrandbits(7)
    f.write("\t\t\t\tRAM(%d) " %(j))
    f.write("<= assign(%d);\n" %(random_num))
    addr_list.append(random_num)

  #expected value calculation
  f.write("\t\t\t\t--expected ")

  expected = addr_list[8] #default value, in case the address doesn't belong to any
      working zone
  result = 0
  for l in range(len(addr_list)-1):
    found = 0
    result = expected - addr_list[l]
    if(result <= 3):  #found a suitable wz
      found = l
      break

  if(found != 0):   #you found a wz, now its bin encoding is written on file
    expected_bin = '1' + format(found, '03b') + (("0001" if result == 0 else "0010") if
        result <= 1 else ("0100" if result == 2 else "1000"))
    f.write(expected_bin[:4] + '.' + expected_bin[4:])
    f.write(' ')
    f.write(str(hex(int(expected_bin, 2)))[2:])
    f.write(' ')
    f.write(str(int(expected_bin, 2)))
    f.write('\n\n')
  else:
    expected_bin = format(expected, '08b')
    f.write(expected_bin[:4] + '.' + expected_bin[4:])  #expected is still the default
       value, which is now encoded into 8 digit binary number
    f.write(' ')
    f.write(str(hex(expected)[2:]))
    f.write(' ')
    f.write(str(expected))
    f.write('\n\n')

f.close() #close file
```

Qui riportiamo i risultati dei quattro test per noi più importanti:

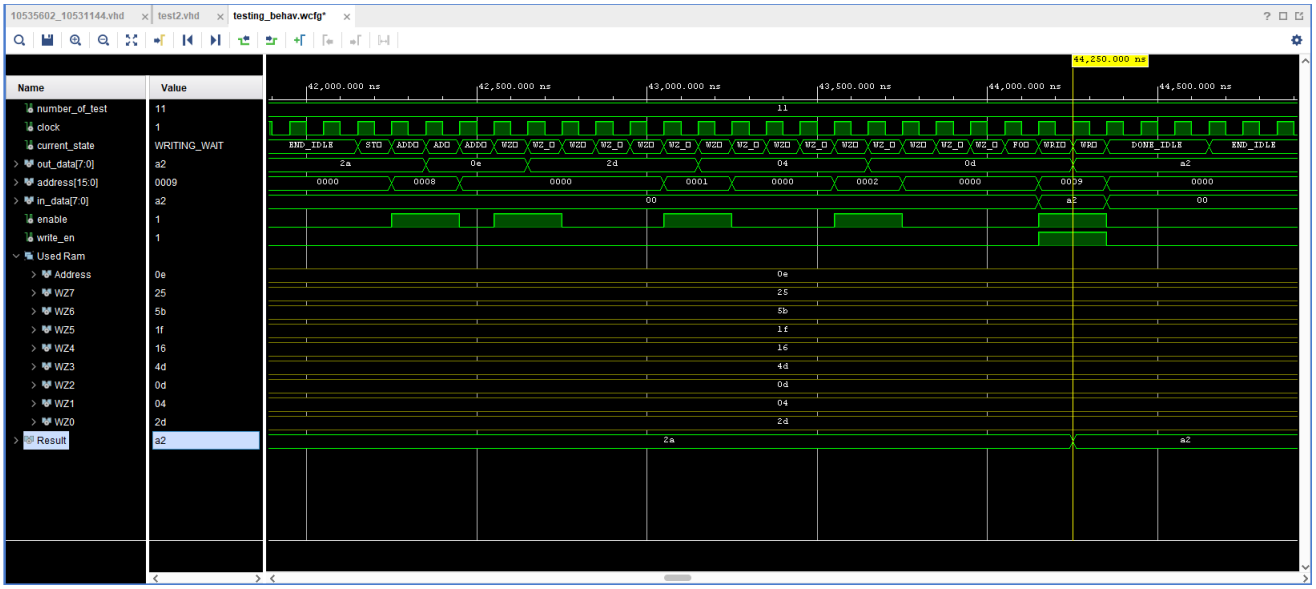
1. *WZ in ordine crescente, risultato atteso negativo*: questo test si trova come esempio nella specifica



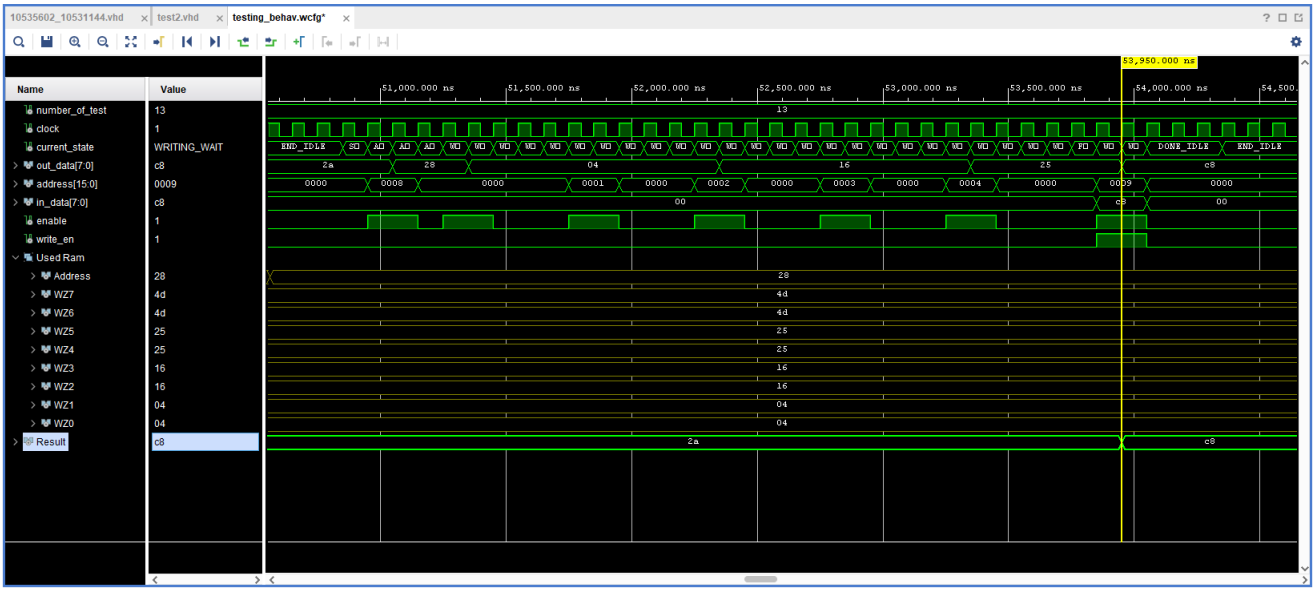
2. *WZ in ordine crescente, risultato atteso positivo*: questo test si trova come esempio nella specifica



3. *WZ in ordine sparso, risultato atteso positivo*



4. *WZ duplicate, risultato atteso positivo*: in questo caso viene considerato valido l'indirizzo della prima working zone che contiene l'indirizzo base, a prescindere dall'offset



5 Limitazioni e possibili ottimizzazioni

5.1 Scalabilità del componente

La specifica fornisce una RAM contenente 8 indirizzi relativi ad altrettante working zone. Il contatore *wz_counter* è però un contatore a 16 bit, per un migliore collegamento con il segnale d'uscita *o_address* senza avere valori di tale uscita forzati al valore 0. La macchina potrebbe essere quindi riadattata per un numero di working zone ben superiore a 8, ma sarebbe comunque necessaria una modifica allo standard di codifica dei risultati ottenuti, ad opera degli stati *FOUND_WZ_ENCODING* e *NO_WZ_ENCODING*, di fatto legati alla specifica di 8 sole working zone.

5.2 Segnale di reset

Dal momento che la specifica non riporta alcun vincolo né assunzione sul segnale di reset *i_rst* e sulla sua durata, abbiamo deciso di implementare un resetting sincrono. In particolare, la macchina è in grado di riconoscere una richiesta di reset a prescindere da quanto a lungo il segnale *i_rst* è rimasto al valore logico 1, ma deve passare un intero ciclo di clock perché tale reset avvenga. Garantiamo comunque un comportamento coerente con la richiesta di reset: in particolare, non è possibile per la macchina notificare alla RAM la fine dell'elaborazione tramite il segnale *o_done* nei cicli immediatamente successivi all'innalzamento di *i_rst*.

5.3 Ottimizzazione dei dati nella RAM

La macchina è progettata per fornire il risultato corretto anche se gli indirizzi relativi alle working zone contenuti nella RAM non sono in ordine (vedasi la sezione *Test eseguiti*). La macchina infatti esegue una scansione lineare di tutte le working zone, decidendo per ogni working zone letta se richiedere la working zone successiva o fermarsi. Perciò, tra tutte le possibili configurazioni, il caso pessimo è quello in cui l'indirizzo base non appartiene a nessuna working zone, oppure quello in cui il base address appartiene alla working zone codificata in *RAM(7)*. Tuttavia, se venisse garantito che gli indirizzi delle working zone nella RAM siano in ordine crescente (o decrescente), sarebbero possibili due modifiche:

- *Riduzione dei casi pessimi*: se l'indirizzo della working zone appena esaminata ha un valore più alto dell'indirizzo base, si può concludere che l'indirizzo base non appartiene a nessuna working zone, riducendo così il numero dei casi pessimi. Tale modifica al progetto è piuttosto modesta, dal momento che basterebbe modificare lo stato di *WZ_DECISION*.

- *Ricerca binaria*: anziché eseguire una ricerca lineare della working zone risulterebbe più efficiente eseguire una ricerca binaria. Tale strategia ridurrebbe l'intera complessità temporale asintotica della macchina: se n fosse il numero di working zone, si passerebbe da $T(n) \in \mathcal{O}(n)$ a $T(n) \in \mathcal{O}(\log n)$. Tuttavia, questa modifica è più onerosa di quella del punto precedente, dal momento che richiederebbe una modifica all'attuale metodo di scansione (affidato al semplice contatore di working zone *wz_counter*) e almeno un nuovo stato per il calcolo del prossimo indirizzo di memoria da richiedere.

6 Conclusioni

Questo componente non ha grosse pretese in termini di prestazioni. Gli stessi ragionamenti del capitolo precedente porterebbero miglioramenti significativi solo con qualche ipotesi in più, ad esempio supponendo che la memorizzazione delle working zone sia effettivamente ordinabile, cosa addirittura non molto probabile. Si tratta quindi di speculazioni dipendenti dal contesto in cui opera il componente in questione, ma esse esulano molto dallo scopo di questa prova finale.
È importante osservare che, oltre ad essere sintetizzabile, il componente descritto è implementabile e supera correttamente entrambe le simulazioni post-implementazione, sia funzionale che temporale. I punti di forza del componente sono legati alla stesura del codice: i nostri sforzi si sono concentrati sull'evitare bad-practise, fornire un programma leggibile, eliminare ogni warning e fornire una documentazione adeguata, con la forma mentis che contraddistingue l'ingegneria del software.