Relazione “Prova Finale (Progetto di Reti Logiche)”

Informazioni studente:

Nome: Andrea

Cognome: Bellani

Codice persona: 10733192

Matricola: 956505

Sommario

1 Introduzione

1.1 Descrizione del componente

Il componente ha il compito di "correggere" una sequenza di dati. La "correzione" consiste in:

- sostituire i dati "non specificati" con gli ultimi dati "utili" letti;
- calcolare un valore di "credibilità" sulla sostituzione fatta.

I dati in memoria sono memorizzati "uno per byte", con un byte "libero" tra un dato e l'altro atto a contenere il valore di credibilità che il componente dovrà calcolare.

Potremmo vedere i dati in memoria come dati raccolti da un sensore, i quali potrebbero essere "non specificati" (a causa, ad esempio, di malfunzionamenti). Il compoenente li riconosce poiché ad essi viene assegnato il valore "speciale" "0".

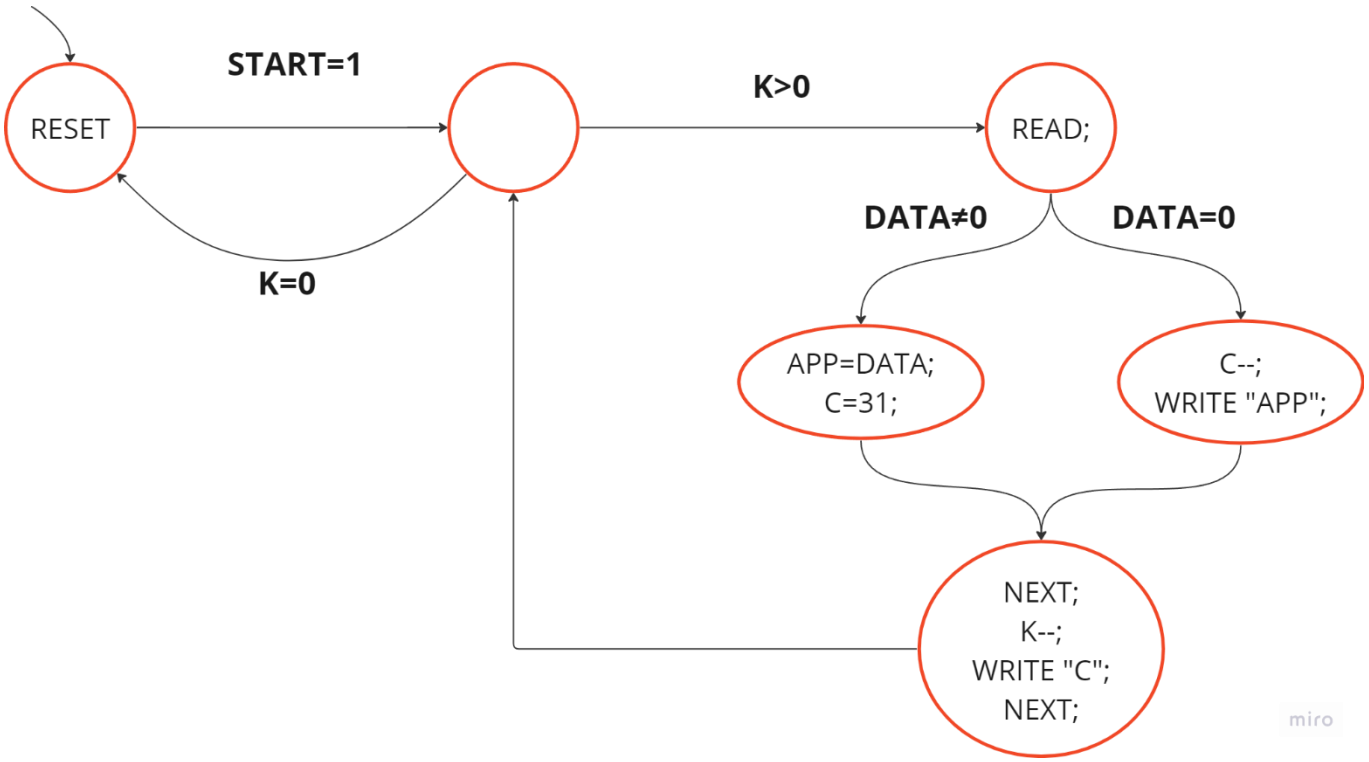
La credibilità va posta a "31" (massimo valore) per i dati "utili" e a "$31 - d$" per i dati "non specificati" letti, dove "d" è la distanza (sul numero di dati) tra il valore "non specificato" e l'ultimo dato "utile" letto.

Alcuni esempi:

- $[128; 0; 0; 0] \Rightarrow [128; 31; 128; 30]$
- $[128; 0; 2; 0] \Rightarrow [128; 31; 2; 31]$
- $[0; 0; 7; 0] \Rightarrow [0; 0; 7; 31]$

1.2 Logica dell'implementazione

Il componente legge, calcola ed aggiorna "una cella alla volta" della porzione di memoria da analizzare seguendo il seguente schema (il diagramma degli stati effettivo della macchina a stati è specificato al paragrafo "2.2.1", questo ne rappresenta semplicemente una sintesi "di alto livello"):



- "READ" : lettura dalla memoria;
- "APP" : registro atto a salvare l'ultimo dato "non nullo" letto dalla memoria;
- "C" : contatore della credibilità;
- "NEXT" : calcolo indirizzo di memoria successivo;
- "K" : contatore delle coppie "valore – credibilità" ancora da analizzare;
- "WRITE" : scrittura in memoria.

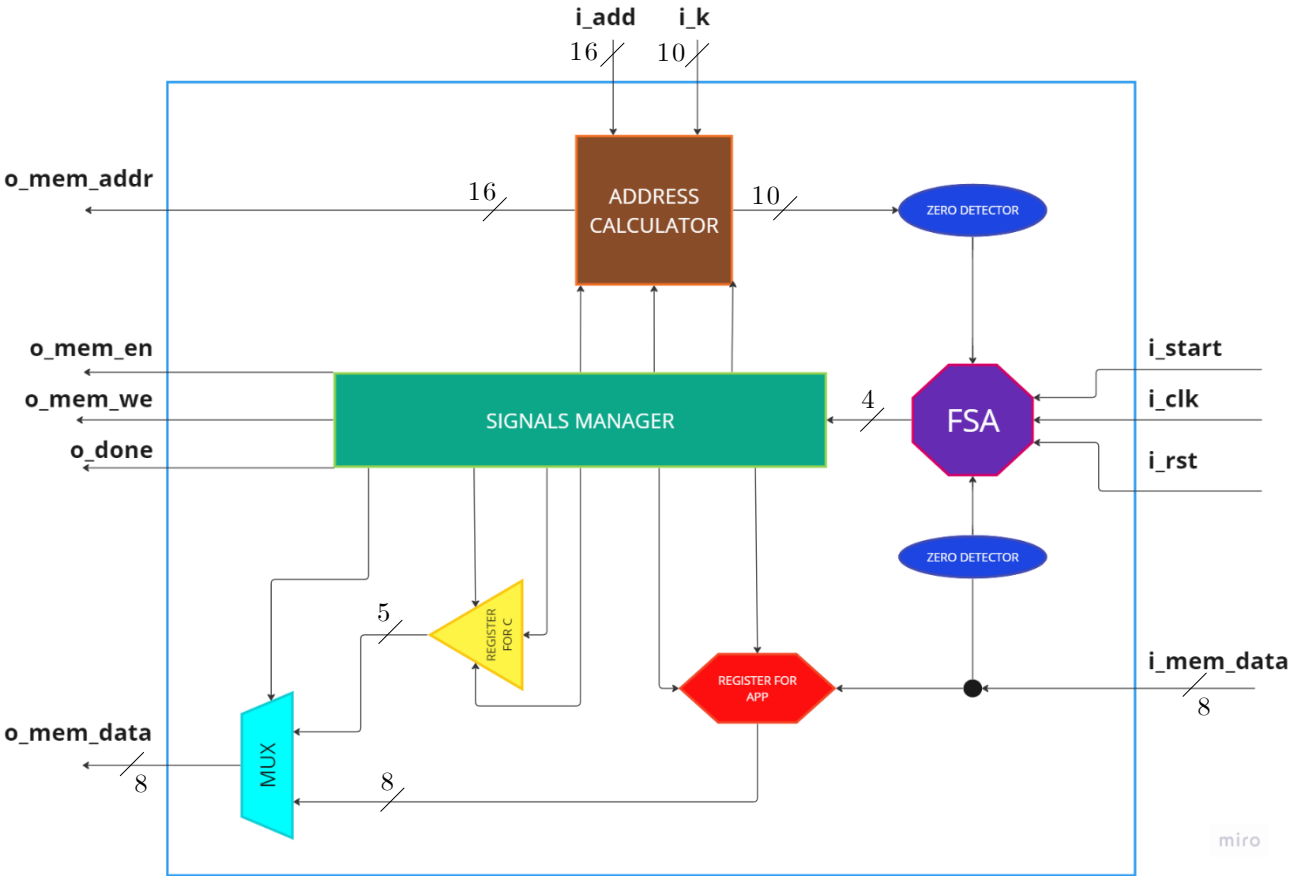
1.3 Cenni a versioni precedenti

Qui viene riportato un sintetico "storico" delle "versioni" del componente (**evidenziate** le migliorie "salienti"):

Versione	Migliorie
1.0	• definiti (solo approssimativamente) "ZERO DETECTOR", "REGISTER FOR APP" e "REGISTER FOR C" e cenno alla mancanza di un modulo per il calcolo dell'indirizzo di memoria;
1.1	• **primo adattamento (incompleto) all'interfaccia richiesta per il componente;** • aggiunta registro per il decremento di "i_k";
1.2	• **prima definizione degli stati ("4") implementati "a pipeline" ("barriere" di flip-flop che propagano i segnali tra gli stati ad ogni ciclo di clock);**
1.3	• piccoli dettagli alla "pipeline";
1.4	• ulteriori dettagli alla "pipeline";
1.5	• **sostituzione della "pipeline" con una macchina a stati ("8" stati);**
1.6	• aggiunta di un decoder per attivare i componenti "di stato in stato";
1.7	• piccole migliorie alla macchina a stati;
1.8	• ulteriori migliorie alla macchina a stati;
1.9	• ulteriori migliorie alla macchina a stati;
2.0	• conclusione adattamento all'interfaccia richiesta per il componente; • **prima implementazione VHDL;**
2.1	• riprogettazione componenti con latch ("ADDRESS CALCULATOR" e "REGISTER FOR C"); • adattamento macchina a stati al workflow richiesto per il componente;
2.2	• **sistemazione macchina a stati dopo testing del componente;**
2.3	• sostituzione decoder con "SIGNALS MANAGER";
2.4	• aggiunto stato nell'FSA per scrittura in memoria e altre minime modifiche nel "SIGNALS MANAGER";
2.5	• modifica parte iniziale diagramma degli stati dell'FSA: ora "k" viene controllato a inizio "ciclo di elaborazione" e non alla fine. Permette così di soddisfare il caso particolare (che comunque si può assumere non capiti mai, per questo non testato) di elaborazioni con "k=0". Conseguente aggiunta stato "12";
2.6	• nomi mnemonici per gli stati dell'FSA.

2 Architettura

2.1 Struttura "di alto livello" del componente



Il componente è costituito da "8" sotto-moduli:

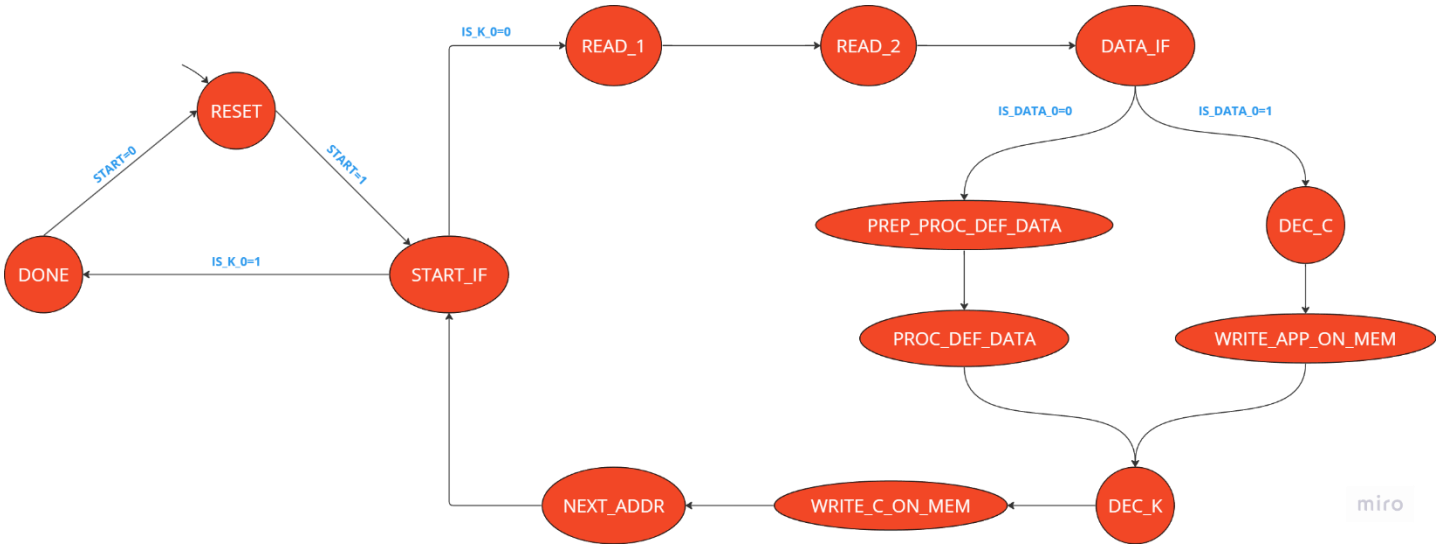
- "**FSA**" : è la macchina a stati, la quale piloterà il "signals manager";
- "**ZERO DETECTOR**" : restituisce "1" se l'ingresso è "0" (alcune transizioni della macchina a stati richiedono che determinati segnali, di più bit, siano "0");
- "**SIGNALS MANAGER**" : contiene l'intera logica combinatoria per attivare tutti gli altri componenti e i segnali di controllo ("o_mem_en", "o_mem_we" e "o_done");
- "**REGISTER FOR APP**" : registro atto a contenere l'ultimo dato non nullo letto dalla memoria (che poi andrà scritto in memoria se il dato letto è nullo);
- "**REGISTER FOR C**" : registro atto a contenere il valore calcolato per la credibilità;
- "**ADDRESS CALCULATOR**" : componente che calcola l'indirizzo di memoria a/in cui scrivere/leggere e tiene il conteggio di quanti dati vanno ancora letti dalla memoria;
- "**MUX**" : per selezionare quale dato scrivere in memoria tra "APP" e "C".

2.2 Architetture sotto-moduli

2.2.1 FSA

La macchina a stati del componente pilota il "SIGNALS MANAGER" il quale, a sua volta, pilota tutti gli altri componenti e la memoria. Il "SIGNALS MANAGER" riceve in input direttamente lo stato corrente della macchina a stati, ingressi come "i_clk" e "i_rst" non vengono mai dati in ingresso ad altri moduli, in questo modo si è sicuri che tutto "proceda" sincronizzato con la macchina a stati e che l'operazione di reset del componente sia centralizzata ("i_rst=0" ⇒ "FSA" và in "0" ⇒ "SIGNALS MANAGER" manda il reset asincrono a tutti gli altri moduli).

La macchina è costituita da "13" stati:



Ad alto livello, ogni "ciclo di elaborazione" (dallo stato "READ_1" allo stato "NEXT_ADDR") effettua l'analisi di una coppia "$(dato; credibilità)$" letta dalla memoria. Si comincia leggendo "$dato$":

- se è "0" : si va negli stati "DEC_K" e "WRITE_APP_ON_MEM" per scrivere in memoria l'ultimo valore letto di "$dato$" non nullo e decrementare il valore di credibilità salvato;
- se è "1" : si va negli stati "*_PROC_DEF_DATA" per salvare "$dato$" in "REGISTER FOR APP" e resettare a "31" il valore salvato per la credibilità.

Giunti allo stato "DEC_K" l'elaborazione procede identica per ambedue i casi: si decrementa il contatore "k" (indica il numero di cicli di elaborazione ancora da eseguire) e si scrive in memoria il valore di credibilità calcolato. In "START_IF", se ci sono ancora cicli di elaborazione da effettuare si torna in "READ_1", altrimenti si va in "DONE" per settare "o_done" e resettare il componente.

Nello specifico:

***Stato* (nome mnemonico)**	***Funzione***
RESET	• RESET di tutti i moduli;
START_IF	• se "k" non è "0" si va in "READ_1" per un nuovo ciclo di elaborazione, altrimenti si va in "DONE";
READ_1	• si "chiede" alla memoria il dato da leggere;
READ_2	• il dato richiesto viene fornito dalla memoria un ciclo di clock dopo più un ulteriore ritardo;
DATA_IF	• il dato richiesto è pronto, se è "0" si andrà nello stato "DEC_C", altrimenti nello stato "PREP_PROC_DEF_DATA";
PREP_PROC_DEF_DATA	• si setta a "31" il valore di credibilità da scrivere in memoria (in realtà si alza semplicemente il segnale "set_to_31" di "REGISTER FOR C", al prossimo ciclo di clock avverà il set a "31");
PROC_DEF_DATA	• si setta il contenuto di "REGISTER FOR C" a "31"; • si salva in "REGISTER_FOR_APP" il valore di "i_mem_data"; • si alza il segnale di "dec_k" per decrementare "k" (al prossimo ciclo di clock verrà decrementato);
DEC_C	• si decrementa il valore di "REGISTER FOR C";
WRITE_APP_ON_MEM	• si scrive il contenuto di "REGISTER FOR APP" in memoria; • si alza il segnale di "dec_k" per decrementare "k" (al prossimo ciclo di clock verrà decrementato);
DEC_K	• si decrementa "k"; • si calcola il nuovo indirizzo di memoria (quello della cella per la credibilità);
WRITE_C_ON_MEM	• si scrive in memoria il contenuto di "REGISTER FOR C";
NEXT_ADDR	• si calcola il nuovo indirizzo di memoria (quello di inizio della prossima "coppia");
DONE	• si pone "o_done=1" e si aspetta "start=0" per tornare a "RESET";

2.2.1.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
start	1	in	i_start	segnale di start del componente
is_k_0	1	in	ZERO DETECTOR(output)	indicare quando "k=0" (fine cicli di elaborazione)
is_data_0	1	in	ZERO DETECTOR(output)	indicare quando "i_mem_data=0" (si è letto un valore nullo dalla memoria)
clock	1	in	i_clk	clock del componente
rst	1	in	i_rst	reset asincrono attivo alto
output	4	out	SIGNALS MANAGER(input)	pilotare le uscite di "SIGNALS MANAGER"

2.2.1.2 Implementazione VHDL

La macchina a stati è implementata in VHDL behavioral in maniera del tutto analoga agli esempi visti col Professor Reghenzani nelle esercitazioni del corso "RETI LOGICHE" tuttavia, si è optato per un design leggermente diverso. Anziché implementare una "funzione di traduzione" all'interno dell'FSA, l'FSA pone in output una rappresentazione in "std_logic_vector(3 downto 0)" dello stato (internamente rappresentato come un segnale di tipo personalizzato "S", costruito come un "enum"):

Stato **(nome mnemonico)**	***Stato*** **(rappresentazione in "std_logic_vector")**
RESET	"0000"
START_IF	"0001"
READ_1	"0010"
READ_2	"0011"
DATA_IF	"0100"
PREP_PROC_DEF_DATA	"0101"
PROC_DEF_DATA	"0110"
DEC_C	"0111"
WRITE_APP_ON_MEM	"1000"
DEC_K	"1001"
WRITE_C_ON_MEM	"1010"
NEXT_ADDR	"1011"
DONE	"1100"

La "funzione di traduzione" è implementata dal "SIGNALS_MANAGER". Il motivo per il quale "S" non è il tipo di dato di "output" sta nel fatto che, provando diversi approcci, l'utilizzo di un ingresso "std_logic_vector" in "SIGNALS_MANAGER" era quello che permetteva di calcolare il valore logico delle uscite di "SIGNALS_MANAGER" nella maniera che personalmente ho trovato più elegante.

2.2.2 SIGNALS MANAGER

Questo è il componente che implementa la logica combinatoria, che in base allo stato dell'FSA "attiva" le funzionalità dei registri, dell'ADDRESS CALCULATOR, di "MUX" e dei segnali di controllo "o_mem_en", "o_mem_we" e "o_done". Non è dunque un componente sequenziale.

Si noti che questo componente fa anche da punto "centralizzato" per il reset di tutti i componenti sequenziali (FSA escluso). Nel momento in cui "i_rst=1" l'FSA si porta nello stato "RESET", questo viene propagato a "SIGNALS MANAGER" il quale manderà i reset asincroni a tutti i componenti. "i_rst" è dunque collegato solo all'FSA.

2.2.2.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
input	1	in	FSA(output)	indicherà lo stato attuale dell'FSA
rst_for_app	1	out	REGISTER FOR APP(rst)	pilotare "rst" di "REGISTER FOR APP"
clock_for_app	1	out	REGISTER FOR APP(clock)	pilotare "clock" di "REGISTER FOR APP"
rst_for_c	1	out	REGISTER FOR C(rst)	pilotare "rst" di "REGISTER FOR C"
clock_for_c	1	out	REGISTER FOR C(clock)	pilotare "clock" di "REGISTER FOR C"
set_to_31_for_c	1	out	REGISTER FOR C(set_to_31)	pilotare "set_to_31" di "REGISTER FOR C"
rst_for_addr_calc	1	out	ADDRESS CALCULATOR(rst)	pilotare "rst" di "ADDRESS CALCULATOR"
clock_for_addr_calc	1	out	ADDRESS CALCULATOR(clock)	pilotare "clock" di "ADDRESS CALCULATOR"
dec_k_for_addr_calc	1	out	ADDRESS CALCULATOR(dec_k)	pilotare "dec_k" di "ADDRESS CALCULATOR"
output_selector	1	out	MUX(sel)	pilotare "sel" di "MUX"
o_mem_en	1	out	o_mem_en	fornire il valore di "o_mem_en"
o_mem_we	1	out	o_mem_we	fornire il valore di "o_mem_we"
o_done	1	out	o_done	fornire il valore di "o_done"

2.2.2.2 Implementazione VHDL

Come si può vedere dallo storico delle versioni precedenti (paragrafo "1.3"), solo in definitiva è stato realizzato un "vero e proprio" componente per la gestione della logica combinatoria, prima era tutto gestito con un semplice decoder il cui input era lo stato attuale dell'FSA. Di fatto, questo componente non fa altro che racchiudere ciò che prima era fatto da un decoder e dalle porte logiche connesse alle sue uscite. Se, ad esempio, ci si trova nello stato "READ_2" ("FSA(output)=0011"), il decoder interno a "SIGNALS MANAGER" riceve "0011" e attiva l'uscita "0011" (tutte le altre sono poste a "0"). Le uscite del decoder sono poi opportunamente collegate alle uscite di "SIGNALS MANAGER" in modo che in ciascuno stato dell'FSA siano attivati tutti e soli i segnali di controllo di quello stato.

2.2.3 Registri

2.2.3.1 REGISTER FOR APP

Il componente non è altro che un semplice registro di "16" bit atto a contenere "i_mem_data" quando questo rappresenta un valore, non di credibilità, diverso da "0". È necessario l'utilizzo di un registro per poter mantenere l'ultimo dato letto dalla memoria diverso da "0".

2.2.3.1.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
input	8	in	i_mem_data	input del registro
clock	1	in	SIGNALS MANAGER (clock_for_app)	clock del registro (interpretato sul fronte di salita)
rst	1	in	SIGNALS MANAGER(rst_for_app)	reset asincrono del registro (attivo alto). Setta a "0" il contenuto del registro
output	8	out	MUX(in0)	verrà messo su "o_mem_data" quando viene letto un dato (non credibilità) uguale a "0"

Altre note:

- se "input=0" il registro mantiene il valore che aveva in precedenza.

2.2.3.1.2 Implementazione VHDL

Il componente è stato implementato come un unico processo la cui sensitivity list è "(input, clock, rst)".

2.2.3.2 REGISTER FOR C

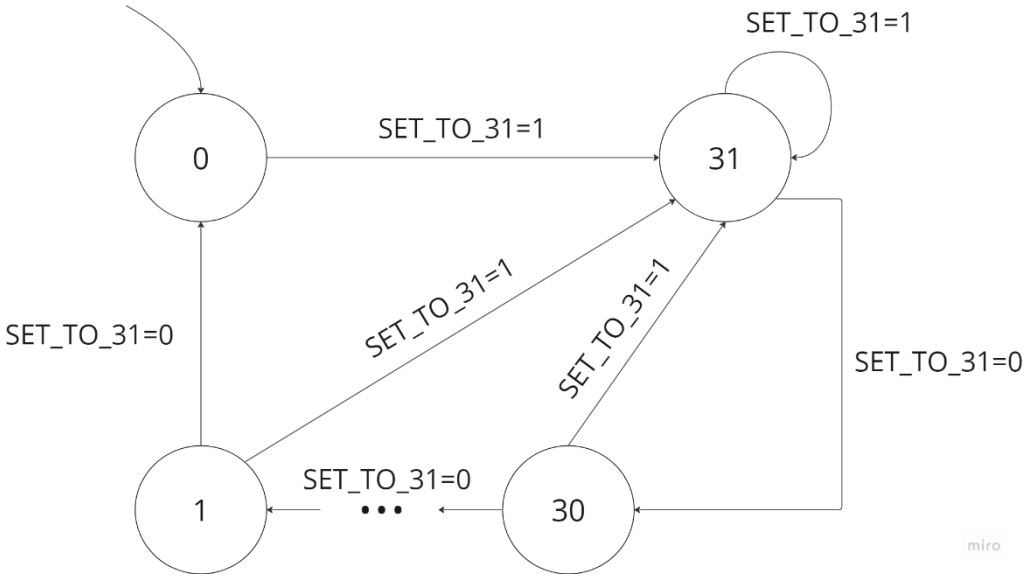
Questo componente è il registro atto a calcolare e mantere il valore di credibilità da scrivere in memoria. Si tratta di un registro da "5" bit che ad ogni ciclo di clock sottrae "1" al proprio contenuto. Oltre a questo, è presente anche un ingresso "*set_to*_31" che setta, al primo fronte di salita, il contenuto del registro a "31" (unico modo per "forzare" il contenuto del registro a un valore diverso da "0").

2.2.3.2.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
clock	1	in	SIGNALS MANAGER(clock_for_c)	clock del registro (interpretato sul fronte di salita)
set_to_31	1	in	SIGNALS MANAGER (set_to_31_for_c)	forza a "31" il contenuto del registro
rst	1	in	SIGNALS MANAGER(rst_for_c)	reset asincrono del registro (attivo alto). Setta a "0" il contenuto del registro
output	5	out	MUX(in1)	verrà messo su "o_mem_data" quando il componente si trova nello stato in cui deve scrivere in memoria la credibilità

2.2.3.2.2 Implementazione VHDL

L'implementazione iniziale in VHDL di questo componente era, a livello di struttura interna, piuttosto differente da quella attuale. Inizialmente anche "set_to_31" era un segnale asincrono (come "rst") e questo portava alla creazione di latch. Per risolvere ho pensato di re-implementare il componente come una piccola macchina a stati il cui schema è:



2.2.4 ADDRESS CALCULATOR

Questo componente è l'unico a svolgere due attività concettualmente "indipendenti":

- calcolo dell'indirizzo di memoria : il componente calcola l'indirizzo di memoria da/in cui leggere/scrivere (si incrementa "mano a mano" partendo da "i_add");
- condizione di fine elaborazione : il componente calcola la condizione ("is_k_0") che decreterà la fine dell'elaborazione (si decrementa "mano a mano" "i_k" fino a quando non è nullo).

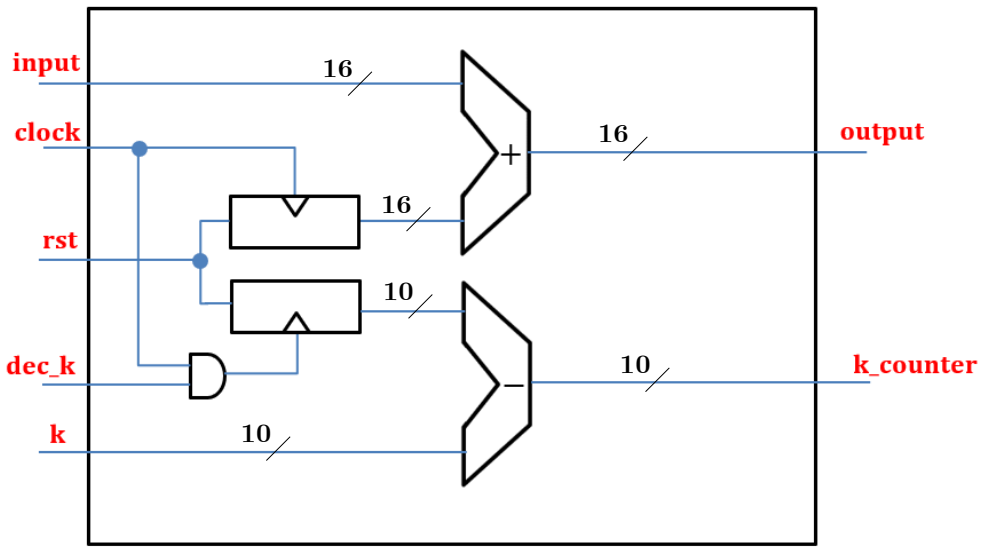
Sono state raggruppate in un unico componente poiché all'inizio ritenevo che il calcolo di una avrebbe modificato quello dell'altra, andando avanti nella progettazione ho capito che uno era del tutto indipendente dall'altro.

2.2.4.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
input	16	in	i_add	indirizzo "base" da cui accedere alla memoria
k	10	in	i_k	"k" dell'elaborazione
dec_k	1	in	SIGNALS MANAGER(dec_k_for_addr_calc)	"1" se va decrementato "k_counter"
clock	1	in	SIGNALS MANAGER(clock_for_addr_calc)	clock del componente (interpretato sul fronte di salita)
rst	1	in	SIGNALS MANAGER(rst_for_addr_calc)	reset asincrono del componente (attivo alto)
output	16	out	o_mem_addr	indirizzo di memoria a/in cui leggere/scrivere
k_counter	10	out	ZERO DETECTOR(a)	contatore di quanti dati vanno ancora letti dalla memoria

2.2.4.2 Implementazione VHDL

L'implementazione iniziale era in VHDL behavioral ma causava latch (inevitabile: quando "rst=1" le uscite devono assumere i valori dei rispettivi ingressi, i quali non sono arbitrari), ho risolto allora la cosa implementando in VHDL structural un "vero e proprio" componente che realizzasse ciò che mi serviva:



- "output" è ottenuto sommando "input" al contenuto di un registro che viene incrementato di "1" ad ogni ciclo di clock;
- "k_counter" è ottenuto sottraendo a "k" il contenuto di un registro che viene incrementato di "1" ad ogni ciclo di clock in cui "$dec\_k = 1$" sul fronte di salita del clock.

2.2.5 Altri moduli

2.2.5.1 ZERO DETECTOR

All'interno del componente sono presenti due "ZERO DETECTOR", uno con un input da "8" bit e l'altro con un input da "10", ma la loro funzione ed implementazione è del tutto analoga.

2.2.5.1.1 Interfaccia

- ZERO DETECTOR "a 10 bit" :

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
a	10	in	ADDRESS CALCULATOR(output_k)	ingresso
output	1	out	FSA(is_k_0)	"1" se "a=0"; "0" altrimenti

- ZERO DETECTOR "a 8 bit :

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
a	8	in	i_mem_data	ingresso
output	1	out	FSA(is_data_0)	"1" se "a=0"; "0" altrimenti

2.2.5.1.2 Implementazione VHDL

Il componente è stato implementato in logica combinatoria in VHDL behavioral.

2.2.5.2 MUX

Il componente non è altro che un semplice multiplexer che, pilotato dal "SIGNALS MANAGER", selezionerà se porre su "o_mem_data" il contenuto di "REGISTER FOR APP" o quello di "REGISTER FOR C" (ovvero l'ultimo dato diverso da zero letto oppure il valore calcolato per la credibilità).

2.2.5.2.1 Interfaccia

	Numero di bit	Tipo	Segnale pilotante/pilotato	Funzione
in0	8	in	REGISTER FOR APP(output)	ingresso "0"
in1	8	in	"000"&REGISTER FOR C(output)	ingresso "1"
sel	1	in	SIGNALS MANAGER(output_selector)	selettore
output	8	out	o_mem_data	dato da scrivere in memoria

Altre note:

- ho optato per implementare un mux "classico" a "8" bit, facendo questo però è stato necessario nell'architettura di "project_reti_logiche" concatenare a "000" l'output di "REGISTER FOR C", dal momento che quest'ultimo è da "5" bit.

2.2.5.2.2 Implementazione VHDL

Il componente è stato implementato come un unico processo la cui sensitivity list è, in quanto blocco combinatorio, costituita da tutti gli ingressi (implementazione suggerita dal Professor Reghenzani a esercitazione).

3 Risultati sperimentali

3.1 Report sintesi

"Vivado 2018.3.1" ha sintetizzato e implementato su "Artix-7 FPGA xc7a200tfbg484-1" i singoli componenti facendo uso di (non sono stati utilizzati componenti hardware diversi da "logic LUT" o "flip-flop"):

	logic LUT	flip-flop
FSA	9	13
REGISTER FOR APP	2	8
REGISTER FOR C	2	5
ADDRESS CALCULATOR	5	26
ZERO DETECTOR A 10 BIT	2	0
ZERO DETECTOR A 8 BIT	4	0
SIGNALS MANAGER	45	0
MUX	4	0
TOTALE	**73**	**52**

tuttavia, il componente finale è stato sintetizzato e implementato con "2" "logic LUT" in meno (probabilmente a causa di condivisioni di parti combinatorie comuni a più uscite).

Non ci sono state ottimizzazioni sul numero di componenti nella fase di implementazione, né nel componente finale, né nei singoli moduli.

Per quanto riguarda le tempistiche:

	SLACK [ns]	RITARDO DATA PATH [ns]
post-sintesi	18.109	1.740 ("56.839 %" route)
post-implementazione	18.416	1.553 ("53.962 %" route)

si è dunque soddisfatto, in ambedue i casi, il requisito di tempo richiesto ("20 ns").

3.2 Simulazioni effettuate

Tutte le simulazioni sono state passate con successo in pre-sintesi funzionale, in post-sintesi funzionale e in post-implementazione funzionale.

3.2.1 Simulazioni sui moduli sequenziali

Al fine di velocizzare la procedura di debugging (oltre che la ricerca di latch), i moduli sequenziali del componente sono stati testati tutti singolarmente prima di interconnetterli e testare il componente finale. Si riportano nei paragrafi successivi i test fatti su di essi.

3.2.1.1 FSA

I test sull'FSA hanno lo scopo di verificare che il componente segua alla perfezione il diagramma degli stati al paragrafo "2.2.1" (in **rosso** le transizioni causate da reset asincroni):

- "FIRST ROUND":
  - "0▶1▶2▶3▶4▶5▶6▶9▶10▶11▶1▶2▶3▶4▶5";
- "SECOND ROUND":
  - "0▶1▶2▶3▶4▶7▶8▶9▶10▶11▶1▶12▶0▶1▶2▶3▶4▶7";
- "THIRD ROUND":
  - "0▶1▶2▶3▶4▶5▶0▶1▶2▶3▶4▶7▶8▶0▶1▶2▶3▶4▶5▶6▶9▶10▶0▶1".

3.2.1.2 REGISTER FOR APP

Si sono testate le funzionalità del modulo quali:

- reset asincrono;
- il valore in input viene salvato sul fronte di salita del clock;
- il registro deve mantenere l'ultimo valore non nullo salvato.

3.2.1.3 REGISTER FOR C

Essendo, di fatto, una macchina a stati, esso viene testato in maniera analoga a quanto visto per "FSA" (il diagramma degli stati è di gran lunga più semplice).

3.2.1.4 ADDRESS CALCULATOR

Si sono testate le funzionalità del modulo quali:

- reset asincrono;
- decremento di "k";
- calcolo indirizzo di memoria.

3.2.2 Simulazioni sul componente

Testato il corretto funzionamento dei singoli moduli si è testato il corretto funzionamento dell'intero componente. Distinguiamo le simulazioni effettuate in:

- simulazioni "base" : testano il funzionamento "base" del componente (senza corner-case/molteplici esecuzioni);
- simulazioni "corner-case" : testano il funzionamento del componente con la memoria inizializzata a valori "corner-case" su singole esecuzioni (ad esempio, tutti "0", "k=1023", "k=1", primi valori in memoria nulli ecc.);
- simulazioni "su più esecuzioni" : testano il funzionamento del componente con più esecuzioni consecutive.

3.2.2.1 Simulazioni "base"

	Test simulazione (stato iniziale memoria)
Simulazione 1	Test bench d'esempio fornito dai docenti
Simulazione 2	"Esempio 1" fornito dai docenti
Simulazione 3	"Esempio 2" fornito dai docenti
Simulazione 5	"Esempio 4" fornito dai docenti
Simulazione 6	"Esempio 5" fornito dai docenti
Simulazione 10	("k=521": omessa per brevità)
Simulazione 12	[255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0 255 0 0 0]
Simulazione 15	[255 0 1 0 0 0 3 0 4 0]

Abbiamo così testato il funzionamento "base" del componente, inteso come:

- calcolo della credibilità;
- calcolo dell'ultimo valore non nullo letto;

in particolare, molteplici di queste operazioni in successione.

3.2.2.2 Simulazioni "corner-case"

	Corner-case analizzato	Test simulazione (stato iniziale memoria)
Simulazione 4	il primo dato letto dalla memoria è "0"	"Esempio 3" fornito dai docenti
Simulazione 7	"k=1" con coppia "valida"	[255 0]
Simulazione 8	"k=1" con coppia "nulla"	[0 0]
Simulazione 9	lunga sequenza iniziale di "0" seguita da due coppie "valide"	[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 253 0 0 0]
Simulazione 11	credibilità sempre a "31" in tutte le coppie	[255 0 255 0]
Simulazione 13	lunga sequenza di soli "0"	sequenza con "k=30" coppie "(0, 0)".
Simulazione 14	sequenza massima di soli "0"	sequenza con "k=1023" coppie "(0, 0)".
Simulazione 16	il primo valore letto è l'unico diverso da "0"	(omesso per brevità)

Abbiamo così ragionevolmente testato i corner-case possibilmente problematici, in particolare:

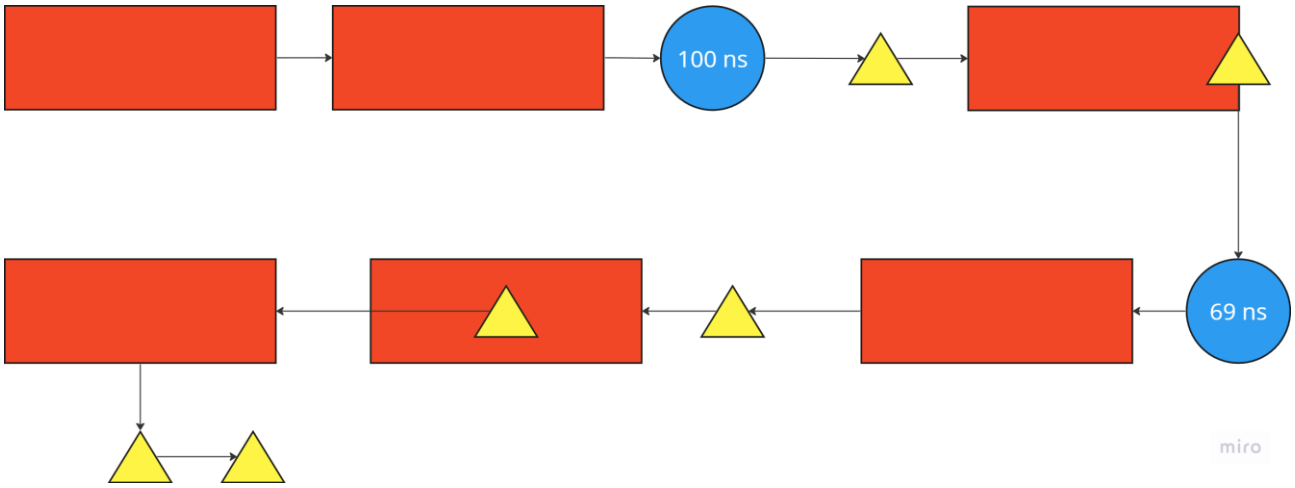
- sequenze che iniziano con "0" o con soli "0";
- "k=1", "k=1023";
- sequenze la cui credibilità non viene mai decrementata.

3.2.2.3 Simulazioni "su più esecuzioni"

Avendo analizzato la correttezza su molteplici esecuzioni "singole", sia "standard" che "con corner-case", possiamo ora testare il corretto funzionamento del componente su più esecuzioni consecutive (concatenate o meno da "reset" asincroni) con un'unica simulazione. I "round" di questa simulazione (ovvero gli intervalli in cui "start=1") analizzano valori in memoria "standard" (i corner-case sulle singole esecuzioni sono già stati verificati nelle simulazioni precedenti).

La simulazione ha il seguente sviluppo:

- coi **rettangoli** identifichiamo un "round";
- coi **triangoli** il verificarsi di un reset (se il triangolo è interno al rettangolo significa che il reset viene inviato durante l'esecuzione del round mentre se è il sul bordo viene inviato non appena "done" si alza);
- coi **cerchi** i "wait";



In questa simulazione facciamo anche un test accurato sui valori dei segnali di controllo "o_mem_en", "o_mem_we" e "o_done":

- "o_mem_en" e "o_mem_we" sempre bassi quando "start" non è alto;
- "o_mem_en" e "o_mem_we" sempre bassi quando "reset" è alto;
- "o_mem_en" e "o_mem_we" sempre bassi quando "o_done" è alto;
- "o_done" sempre basso quando "reset" è alto;
- "o_done" sempre basso sul fronte di salita di "start";
- "o_done" sempre alto sul fronte di discesa di "start" (a meno che non sia avvenuto un "reset", in quel caso ignoriamo il controllo fino a quando "start" non torna alto).

Dunque, in questa simulazione sono stati verificati:

- esecuzioni multiple di round;
- reset asincroni:
    - un certo tempo dopo la conclusione di un round;
    - appena prima di concludere un round;
    - subito dopo la conclusione di un round;
    - nel mezzo dell'esecuzione di un round;
    - appena dopo un altro reset;
- i vincoli sui segnali di controllo.

4 Conclusioni

Abbiamo realizzato un componente sequenziale utilizzando diverse delle competenze acquisite nel corso "reti logiche", quali:

- design in blocchi del componente, con particolare attenzione alla separazione tra logica combinatoria e logica sequenziale;
- progetto di una macchina a stati conforme alle proprie necessità;
- progetto di componenti sequenziali secondo diverse "logiche di funzionamento" (ad esempio, "REGISTER_FOR_C" è implementato con una macchina a stati mentre "ADDRESS_CALCULATOR" fa uso di blocchi combinatori);
- utilizzo e design (di alto livello, in VHDL behavioral) di sommatori, sottrattori e registri.

Inoltre, si è svolta un'analisi degli input interessanti da testare per verificare il corretto funzionamento del circuito e una riprogettazione (accennata nello storico delle versioni e nelle descrizioni dei componenti) al fine di rimuovere tutti i registri latch sintetizzati da Vivado.

Ultima revisione: 15/06/2024

--

Andrea Bellani