

1. Introduzione

Bisogna implementare una versione semplificata dell'algoritmo di equalizzazione dell'istogramma.

Il modulo deve equalizzare un'immagine in scala di grigi da 0 a 255 di grandezza massima 128*128.

Esempio (images from [Wikipedia](#))

Immagine di partenza:



Risultato con algoritmo completo:



Risultato con algoritmo semplificato:



Protocollo di comunicazione

L'interazione è divisa in tre fasi:

- 1) Reset e comunicazione di inizio elaborazione
- 2) Elaborazione.
- 3) Comunicazione di fine elaborazione

tb_done rappresenta il segnale di DONE ed è settato dal modulo implementato.

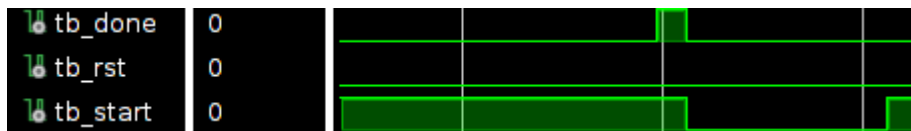
tb_rst e tb_start rappresentano il segnale di START e di RESET e sono settati dall'esterno.

1) Reset e comunicazione di inizio elaborazione

Nel caso di elaborazione della prima immagine il segnale di RESET viene portato ad 1 e poi a 0. Successivamente il segnale di START è portato ad 1 ed inizia l'elaborazione.



Nel caso di elaborazione di immagini successive il segnale di START è portato ad 1 ed inizia l'elaborazione.



2) Elaborazione

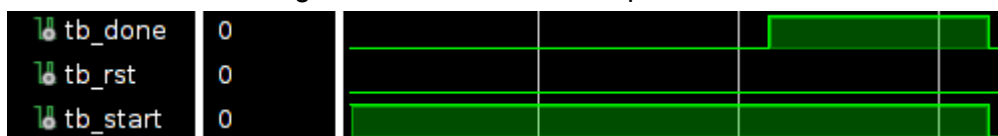
Durante l'elaborazione il segnale di DONE rimane basso.

L'elaborazione è sua volta divisa in:

- 1) Lettura dimensione immagine
Si leggono i primi due byte della memoria e si salva la dimensione dell'immagine.
- 2) Lettura immagine
Si legge l'immagine dalla memoria, si salvano massimo e minimo e si calcola lo shift level.
- 3) Scrittura
Si legge ogni pixel e lo si riscrive shiftato a sinistra di una quantità pari allo shift level.

3) Comunicazione di fine elaborazione

Il modulo comunica la fine dell'elaborazione mettendo il segnale DONE ad 1. DONE rimane alto finché il segnale START non viene riportato a 0.



Algoritmo da implementare in dettaglio

1. $\Delta_VALUE = \text{MAX_PIXEL_VALUE} - \text{MIN_PIXEL_VALUE}$

Viene determinato il Δ_VALUE , che ci dirà di quanto modificare l'immagine.

- Per valori grandi l'immagine cambierà poco.
- Per 255 resterà identica.
- Per valori piccoli l'immagine cambierà molto.

2. $\text{SHIFT_LEVEL} = (8 - \text{FLOOR}(\text{LOG}_2(\Delta_VALUE + 1)))$

Viene calcolato di quanto dobbiamo shiftare l'immagine.

Si può ricavare lo shift level dai seguenti controlli a soglia:

Δ_VALUE	SHIFT_LEVEL
0	8
Da 1 a 2	7
Da 3 a 6	6
Da 7 a 14	5
Da 15 a 30	4
Da 31 a 62	3
Da 63 a 126	2
Da 127 a 254	1
255	0

Per ogni pixel:

3) $\text{TEMP_PIXEL} =$
 $(\text{CURRENT_PIXEL_VALUE} - \text{MIN_PIXEL_VALUE})$
 $\ll \text{SHIFT_LEVEL}$

Viene effettuato lo shift a sinistra del valore binario associato al pixel.

4) $\text{NEW_PIXEL_VALUE} = \text{MIN}(255, \text{TEMP_PIXEL})$

Se il valore shiftato è maggiore di 255 verrà scritto in memoria 255.

Memoria:

Lo spazio che verrà utilizzato per lettura e scrittura dipende dalla dimensione dell'immagine.

Indirizzo di memoria		Memoria prima dell'elaborazione		Memoria dopo l'elaborazione
0	Dimensioni immagine= $\text{mem}(0) * \text{mem}(1)$	2		2
1		2		2
2	Spazio di Lettura	76	Elaborazione →	76
3		131		131
4		109		109
5		89		89
6	Spazio di Scrittura			0
7				255
8				255
9				104
...	

2. Architettura

Ho optato per l'architettura datapath + state machine invece di architettura ad-hoc perchè ha una divisione dei compiti che rende più facile comprendere il funzionamento di ogni processo.

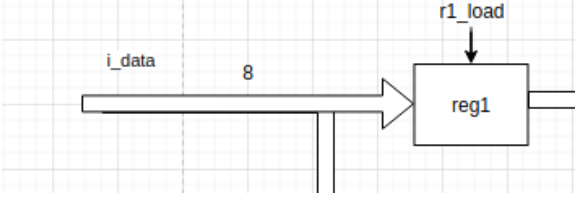
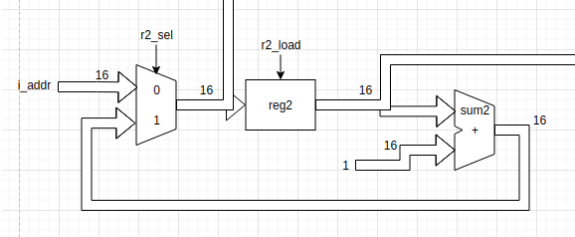
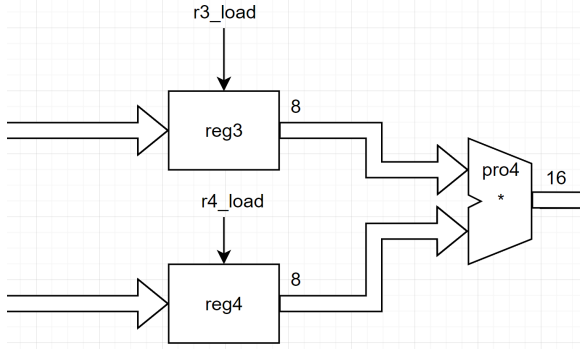
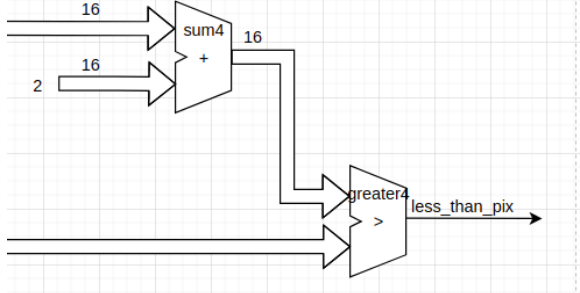
Schema dei segnali:

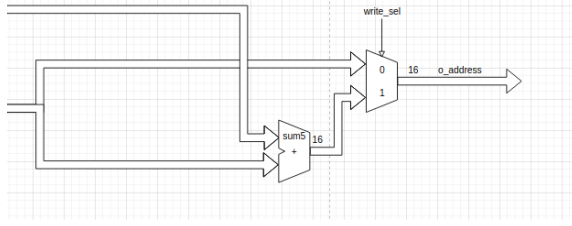
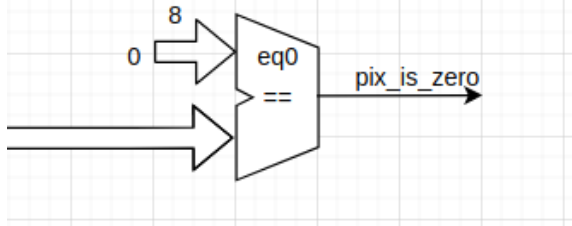
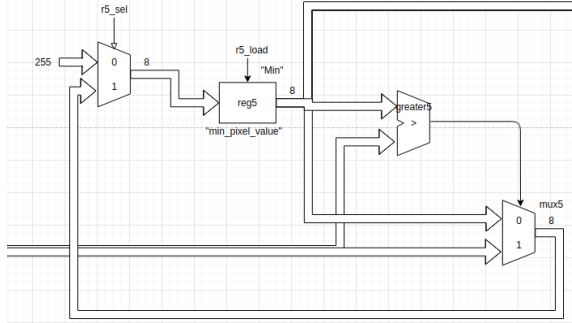
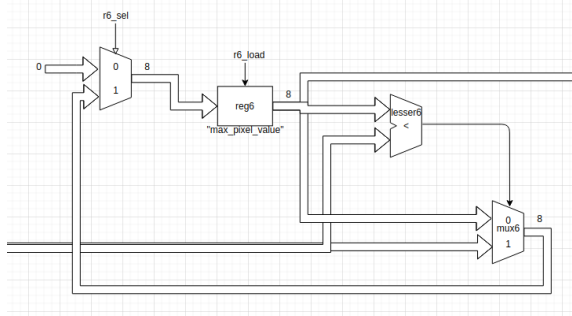
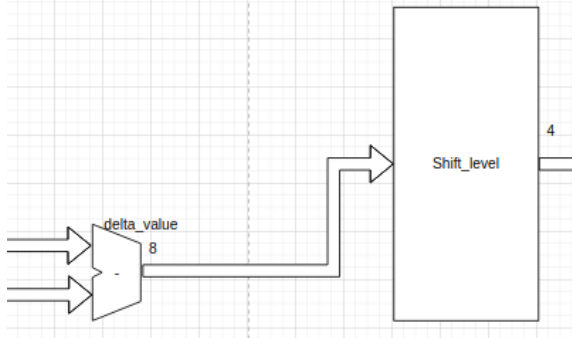
<p>To state machine -></p> <p>i_clk i_rst i_start i_data</p>	State machine	<p>To datapath -></p> <p>i_clk i_rst i_data i_address</p> <p>r1_load r2_load r2_sel r3_load r4_load r5_load r5_sel r6_load r6_sel write_sel</p>	Datapath
<p><- From state machine:</p> <p>o_address o_done o_en o_we o_data</p>		<p><- From datapath</p> <p>pix_is_zero less_than_pix o_address o_data</p>	

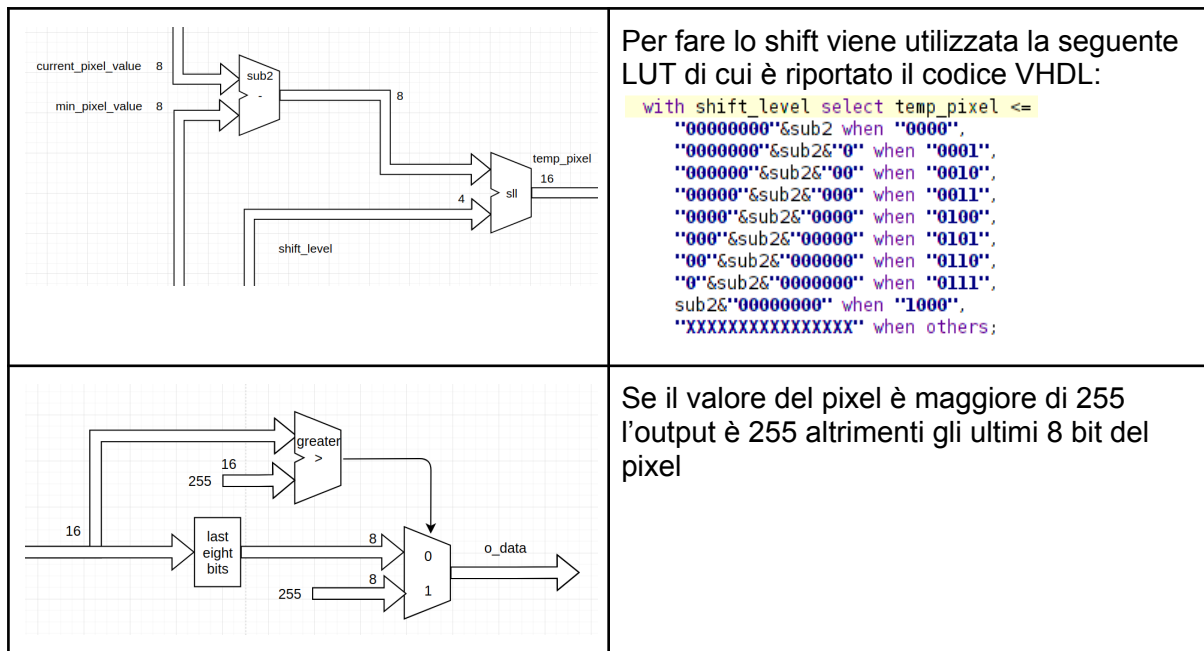
a) Datapath

Nel modulo datapath i dati in input seguono il percorso scelto dalla macchina a stati e si ha un processo per ogni registro.

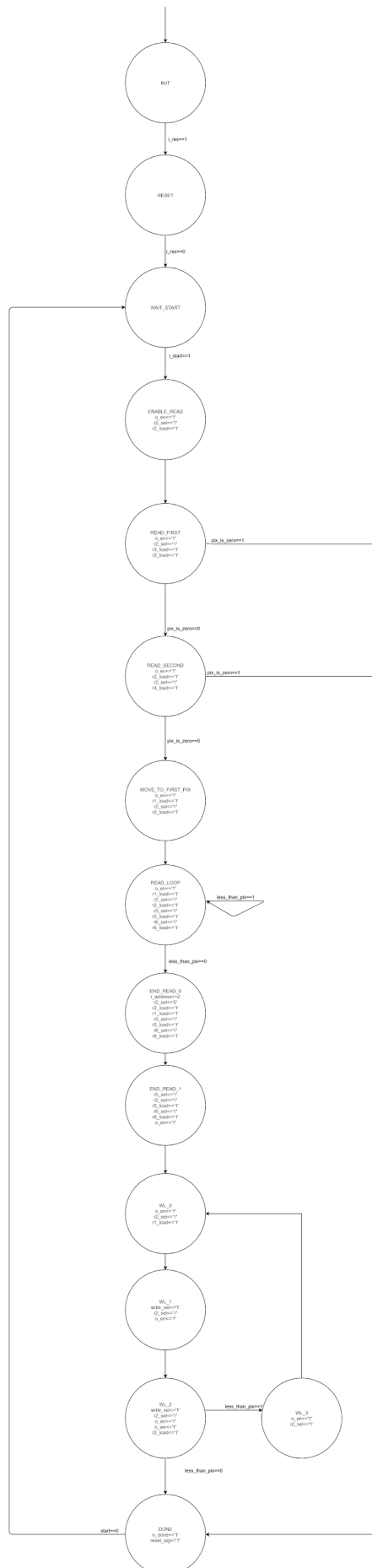
Utilizzo dei registri

	<p>Il registro r1 è usato come buffer per il byte letto dalla memoria.</p>
	<p>Questa struttura è un contatore che può essere incrementato di un'unità ad ogni ciclo di clock. i_addr è un segnale che arriva dalla state machine e serve per settare la posizione iniziale del contatore.</p>
	<p>I registri r3 ed r4 sono collegati ad input_data, che è il dato letto dalla memoria. L'output di pro4 è la dimensione dell'immagine</p>
	<p>Prende in input la dimensione dell'immagine ed il segnale in output indica se bisogna terminare la lettura o la scrittura.</p>

	<p>Gli input sono la dimensione dell'immagine e l'indirizzo di memoria in cui leggere o scrivere. write_sel seleziona se l'output è l'indirizzo di lettura o scrittura.</p>
	<p>Prende in input il dato letto dalla memoria e controlla se è uguale 0.</p>
	<p>L'input è il pixel value letto da memoria e salvato nel registro 1. In r5 è salvato il minimo valore tra i pixel in memoria.</p>
	<p>L'input è il pixel value letto da memoria e salvato nel registro 1. In r6 è salvato il massimo valore tra i pixel in memoria.</p>
	<p>Viene calcolato il delta_value da massimo e minimo e viene selezionato il corretto valore del delta_value da una lookup table.</p>



b) State machine



La state machine ha tre processi

- state_reg serve a gestire il reset: se i_rst='1' porta lo stato a RESET e resetta il datapath.
- lambda cambia lo stato in base allo stato corrente ed i segnali: i_rst, i_start, less_than_pix, pix_is_zero.
- lambda2 setta i segnali in base allo stato corrente. Di default sono impostati tutti a zero tranne reset_sig che è settato ad i_rst e serve a resettare il datapath.

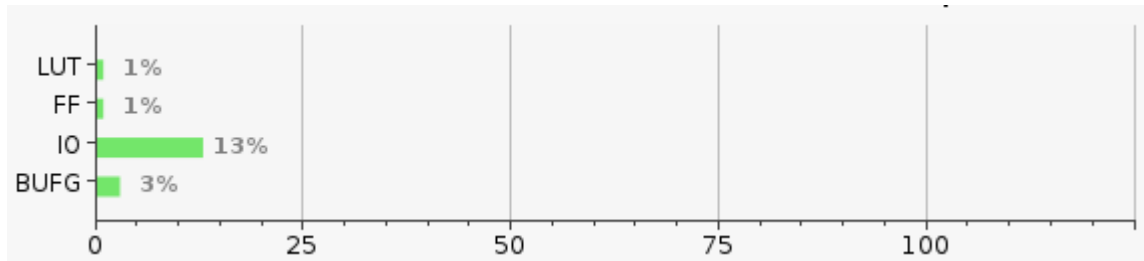
Descrizione stati

INIT	Stato iniziale in cui si trova la macchina a stati prima dell'elaborazione della prima immagine.
RESET	Viene fatto il reset del datapath. Ci si finisce solo nella prima elaborazione.
WAIT_START	Si aspetta che il segnale di start sia alto per iniziare l'elaborazione.
ENABLE_READ	Si abilita la lettura da memoria per leggere la dimensione dell'immagine
READ_FIRST	Si legge il primo byte della memoria che indica la dimensione dell'immagine lungo la prima direzione. Se la dimensione è 0 l'elaborazione termina e si va nello stato DONE. Altrimenti si continua l'elaborazione e si avanza all'indirizzo di memoria dove è memorizzata la dimensione dell'immagine lungo la seconda direzione.
READ_SECOND	Si legge il secondo byte della memoria che indica la dimensione dell'immagine lungo la seconda direzione. Se la dimensione è 0 l'elaborazione termina e si va nello stato DONE. Altrimenti si continua l'elaborazione.
MOVE_TO_FIRST_PIX	Ci si prepara al loop di lettura impostando r1_load ad 1 in modo che allo stato successivo si avrà in r1 il valore del primo pixel.
READ_LOOP	Loop di lettura sugli indirizzi dell'immagine dove si confronta e aggiorna massimo o minimo. Si esce dal loop quando less_than_pix==0, ovvero quando è stato letto il valore di tutti i pixel.
END_READ_0	Si resetta il contatore degli indirizzi per prepararsi ad un ciclo di lettura e scrittura.
END_READ_1	Si aggiorna massimo e minimo considerando l'ultimo pixel e si setta o_en ad 1 per iniziare il ciclo di lettura e scrittura.
WL_0	Primo stato del loop. Si legge il valore del pixel.
WL_1	Si calcola l'indirizzo di output
WL_2	Si abilita la scrittura in memoria
WL_3	Si scrive il valore del pixel equalizzato nella memoria ed allo stato successivo, WL_0 si ricomincia il loop con l'indirizzo successivo.
DONE	Si scrive l'ultimo pixel in memoria e si resetta il datapath.

3) Risultati sperimentali

E' stata usata la FPGA consigliata, la xc7a200tfbg484-1

Utilizzo percentuale:



I valori di utilizzo della fpga sono largamente rispettati.

Spartizione tra LUT e FF:

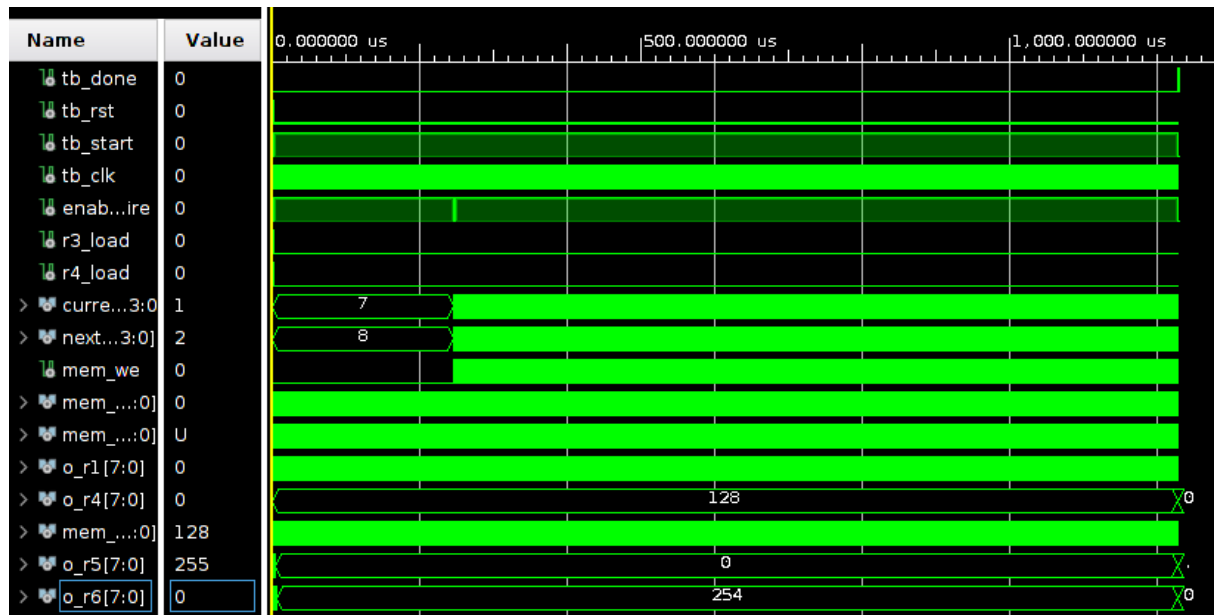
Resource	Utilization	Available	Utilization %
LUT	237	133800	0.18
FF	60	267600	0.02
IO	38	285	13.33
BUFG	1	32	3.13

Nel progetto sono stati utilizzati molte look-up table e relativamente pochi Flip-Flop.

Il datapath utilizza più FF e LUT rispetto alla state machine.

4) Simulazioni

- Immagine singola 2*2:
Il primo test bench che ho utilizzato controlla se una piccola immagine viene elaborata correttamente.
- Test bench con dimensione immagine 0*0, 0*1, 1*0:
Controllano se l'elaborazione termina con immagini di grandezza nulla.
- Test bench 128*128: Controlla la condizione limite con un'immagine di grandezza massima.



Post-Synthesis Functional Simulation

- Immagini successive:
Controlla se il datapath e la state machine vengono resettate correttamente dopo l'elaborazione di ogni immagine e che le immagini successive vengono elaborate correttamente.
- Altri test bench:
Ho utilizzato alcuni test bench disponibili su beep e costruito altri test bench con un generatore che ho scritto in python.

5) conclusioni

- L'approccio datapath+state machine rende più facile l'estensione.
- Il generatore di test bench da me scritto rende più facile il testing e mi permette di convertire la memoria in un'immagine che poi mi viene mostrata. Questo mi permette di comprendere meglio cosa fa l'algoritmo e come differisce dalla versione completa.
- Ho scritto prima e seconda direzione invece di colonne e righe nella sezione datapath poiché l'elaborazione avrebbe lo stesso risultato anche con i byte che indicano il numero di righe e di colonne invertiti.
- Nel processo lambda2 della macchina a stati la riga 204 può essere rimossa poiché per caricare i_address devo settare r2_sel a zero e questo viene fatto alla riga 206.
Tuttavia non ho più a disposizione un computer con Vivado e non posso testare il codice quindi preferisco non modificare senza testare.
In realtà anche la riga 206 può essere rimossa perché zero è il valore di default ma preferisco lasciarla perché rende più chiara la comprensione del funzionamento del processo.

```
202      when END_READ_0 =>  
203          --o_en<='1';  
204          r2_sel<='1';  
205          i_address<="000000000000010";  
206          r2_sel<='0';  
207          r2_load<='1';  
208          r1_load<='1';  
209          r5_sel<='1';  
210          r5_load<='1';  
211          r6_sel<='1';  
212          r6_load<='1';
```