Prova Finale (Progetto di Reti Logiche)

Prof. Gianluca Palermo – Anno 2019/2020

Marco Riva (Codice Persona 10605051 – Matricola 889593)
Tommaso Pozzi (Codice Persona 10572283 – Matricola 891456)

Indice

1. Introduzione

1.1 Scopo del progetto

Si definiscono otto zone di lavoro costituite da intervalli di indirizzi di dimensione fissa che partono tutti da un indirizzo base. Lo scopo del progetto è di implementare un componente hardware, descritto in VHDL, che, presi in ingresso gli indirizzi base delle otto zone di lavoro e un indirizzo da codificare, trasformi quest'ultimo, nel caso in cui appartenga all'intervallo di una delle zone di lavoro, utilizzando il metodo di codifica a bassa dissipazione di potenza denominato "Working Zone".

1.2 Specifiche generali

Gli indirizzi in input sono 9: i primi 8 sono le Working Zone (WZ) e il 9° è l'indirizzo (ADDR) da codificare. Tutti gli ingressi sono a 8 bit. Ogni WZ corrisponde ad un intervallo fisso di 4 indirizzi. Il componente hardware deve codificare un indirizzo in uscita a 8 bit secondo l'algoritmo spiegato di seguito:

- se ADDR non appartiene a nessuna WZ, l'output sarà composto da un primo bit (WZ_BIT) uguale a 0, concatenato ad ADDR, trasmesso così com'è.

Esempio: VALORE NON PRESENTE IN NESSUNA WORKING-ZONE.

Indirizzo Memoria	Valore	Commento
0	4	// Indirizzo Base WZ 0
1	13	// Indirizzo Base WZ 1
2	22	// Indirizzo Base WZ 2
3	31	// Indirizzo Base WZ 3
4	37	// Indirizzo Base WZ 4
5	45	// Indirizzo Base WZ 5
6	77	// Indirizzo Base WZ 6
7	91	// Indirizzo Base WZ 7
8	42	// ADDR da codificare

L'output sarà:

8	7	6	5	4	3	2	1
0	0	1	0	1	0	1	0

Figura 1: l'8° bit (MSB) rappresenta WZ_BIT; ADDR viene codificato a partire dal 7° bit.

- se ADDR appartiene ad una WZ, l'output sarà composto da una concatenazione di 3 numeri binari per un totale di 8 bit. In ordine:
    - WZ_BIT uguale a 1 che indica che ADDR è stato trovato all'interno di una WZ;
    - WZ_NUM (3 bit) che indica il numero della WZ a cui ADDR appartiene;
    - WZ_OFFSET (4 bit) che indica, in codifica one-hot, a quale dei 4 indirizzi della WZ selezionata corrisponde ADDR.

Esempio: VALORE PRESENTE IN UNA WORKING-ZONE.

Indirizzo Memoria	Valore	Commento
0	4	// Indirizzo Base WZ 0
1	13	// Indirizzo Base WZ 1
2	22	// Indirizzo Base WZ 2
3	31	// Indirizzo Base WZ 3
4	37	// Indirizzo Base WZ 4
5	45	// Indirizzo Base WZ 5
6	77	// Indirizzo Base WZ 6
7	91	// Indirizzo Base WZ 7
8	33	// ADDR da codificare

L'output sarà:

8	7	6	5	4	3	2	1
1	0	1	1	0	1	0	0

Figura 2: l'8° bit (MSB) rappresenta WZ_BIT; dal 7° al 5° bit viene specificato WZ_NUM; dal 4° al 1° bit viene indicato WZ_OFFSET.

1.3 Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
entity project_reti_logiche is
Port (
     i_clk               : in std_logic;
     i_start             : in std_logic;
     i_rst               : in std_logic;
     i_data              : in std_logic_vector(7 downto 0);
     o_address           : out std_logic_vector(15 downto 0);
     o_done              : out std_logic;
     o_en                : out std_logic;
     o_we                : out std_logic;
     o_data              : out std_logic_vector(7 downto 0)
);
end project_reti_logiche;
```

In particolare:

- `i_clk` è il segnale di `CLOCK` in ingresso generato dal test bench;
- `i_start` è il segnale di `START` generato dal test bench;
- `i_rst` è il segnale di `RESET` che inizializza la macchina pronta per ricevere il primo segnale di `START;`
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- `o_address` è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- `o_done` è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- `o_en` è il segnale di `ENABLE` da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `o_we` è il segnale di `WRITE ENABLE` da dover mandare alla memoria (=1) per potervi scrivere. Per leggere da memoria esso deve essere 0;
- `o _data` è il segnale (vettore) di uscita dal componente verso la memoria.

1.4 Dati e descrizione memoria

I dati, ciascuno di dimensione 8 bit, sono memorizzati in una memoria con indirizzamento al byte:

- gli indirizzi dallo 0 al 7 sono usati per memorizzare gli indirizzi delle WZ;
- l'indirizzo 8 è usato per memorizzare l'ADDR da codificare;
- l'indirizzo 9 è usato per scrivere l'indirizzo codificato secondo la specifica.

Working Zone 0	Indirizzo 0
Working Zone 1	Indirizzo 1
Working Zone 2	Indirizzo 2
Working Zone 3	Indirizzo 3
Working Zone 4	Indirizzo 4
Working Zone 5	Indirizzo 5
Working Zone 6	Indirizzo 6
Working Zone 7	Indirizzo 7
ADDR	Indirizzo 8
Indirizzo codificato in uscita	Indirizzo 9

Figura 3: Rappresentazione degli indirizzi significativi della memoria

2. Design

Quando il segnale `i_start` in ingresso viene portato a 1, il componente sviluppato inizia l'elaborazione, spostandosi dallo stato `IDLE` al primo stato della computazione. Una volta terminata la computazione, dopo avere scritto il risultato in memoria, il componente alza il segnale `o_done`. Il test bench risponde abbassando `i_start` e, successivamente, il componente riporta a 0 `o_done`; il componente ritorna nello stato `IDLE`, in attesa che il segnale `i_start` torni alto.
Il componente dispone inoltre di un segnale `i_rst` che, insieme agli altri appena elencati, ci hanno portato a definire una FSM(D), macchina a stati finiti con *data path*, che combina una normale FSM con tipici circuiti sequenziali.
Nelle seguenti sezioni troviamo, infatti, sia la descrizione della FSM, sia la descrizione della parte sequenziale della macchina che permette la gestione dei registri utilizzati.

2.1 Stati della macchina

La macchina costruita è composta da 8 stati. Qui di seguito è fornita una breve descrizione per ciascuno di questi.

2.1.1 `IDLE` state

Stato iniziale in cui si attende il segnale di `i_start`. In caso venga alzato il segnale `i_rst` si torna in questo stato.

2.1.2 `FETCH_CONST` state

Stato in cui viene richiesto l'ADDR in ingresso e la prima Working Zone. Il tutto viene fatto tornando 2 volte in questo stato, in quanto non è possibile effettuare queste operazioni in simultanea.

2.1.3 `WAIT_RAM` state

Stato in cui si attende la risposta dalla memoria in seguito alla richiesta di un dato.

2.1.4 `GET_CONST` state

Stato in cui vengono memorizzati, in due rispettivi registri, l'ADDR e le Working Zone in ingresso.

2.1.5 `COMPARE` state

Stato in cui viene confrontato l'ADDR con la Working Zone corrente.

2.1.6 `CALC` state

Stato in cui viene calcolato l'indirizzo codificato in binario, pronto per essere scritto in memoria.

2.1.7 `WRITE_OUT` state

Stato in cui viene scritta la codifica all'indirizzo della memoria 9 e viene posto a 1 `o_done`.

2.1.8 `DONE` state

Stato in cui si attende che la memoria abbassi `i_start` per poter abbassare `o_done` e tornare nello stato `IDLE`.

2.2 Scelte progettuali

La principale scelta progettuale effettuata è stata quella di descrivere il componente con due processi:

1. Il primo rappresenta la parte sequenziale della macchina e serve per gestire l'RT (register transfer) e quindi come vengono manipolati i registri.
2. Il secondo rappresenta invece la FSM che analizza i segnali in ingresso e lo stato corrente per determinare il prossimo stato in cui si evolverà il sistema.

L'algoritmo determina il risultato finale utilizzando, come operazioni logiche, solamente la concatenazione di indirizzi (`SHIFT` + `OR`) e l'`AND` tra due registri. Oltre a queste, utilizza la `ADD` per incrementare l'indirizzo di memoria da richiedere e il `CAST` di un registro al corrispondente intero.

Abbiamo inoltre deciso di utilizzare un approccio che preveda di mantenere memorizzate meno informazioni possibili. Le WZ vengono salvate solamente per il confronto con l'ADDR, poi vengono sovrascritte con l'analisi della WZ successiva. Questo approccio offre, inoltre, maggiore scalabilità, in quanto, se si dovesse aumentare il numero delle working zone da analizzare, non sarebbero necessarie modifiche al codice se non quella di cambiare le dimensioni dei vettori e l'indirizzo in ingresso dell'ADDR.
Un limite di questa scelta potrebbe verificarsi nel caso in cui ci siano più segnali di start antecedenti un segnale di reset (quindi stesse WZ, ma più ADDR diversi da codificare), perché in questo caso si dovrebbe ripetere da zero l'intero algoritmo per ogni segnale di start.

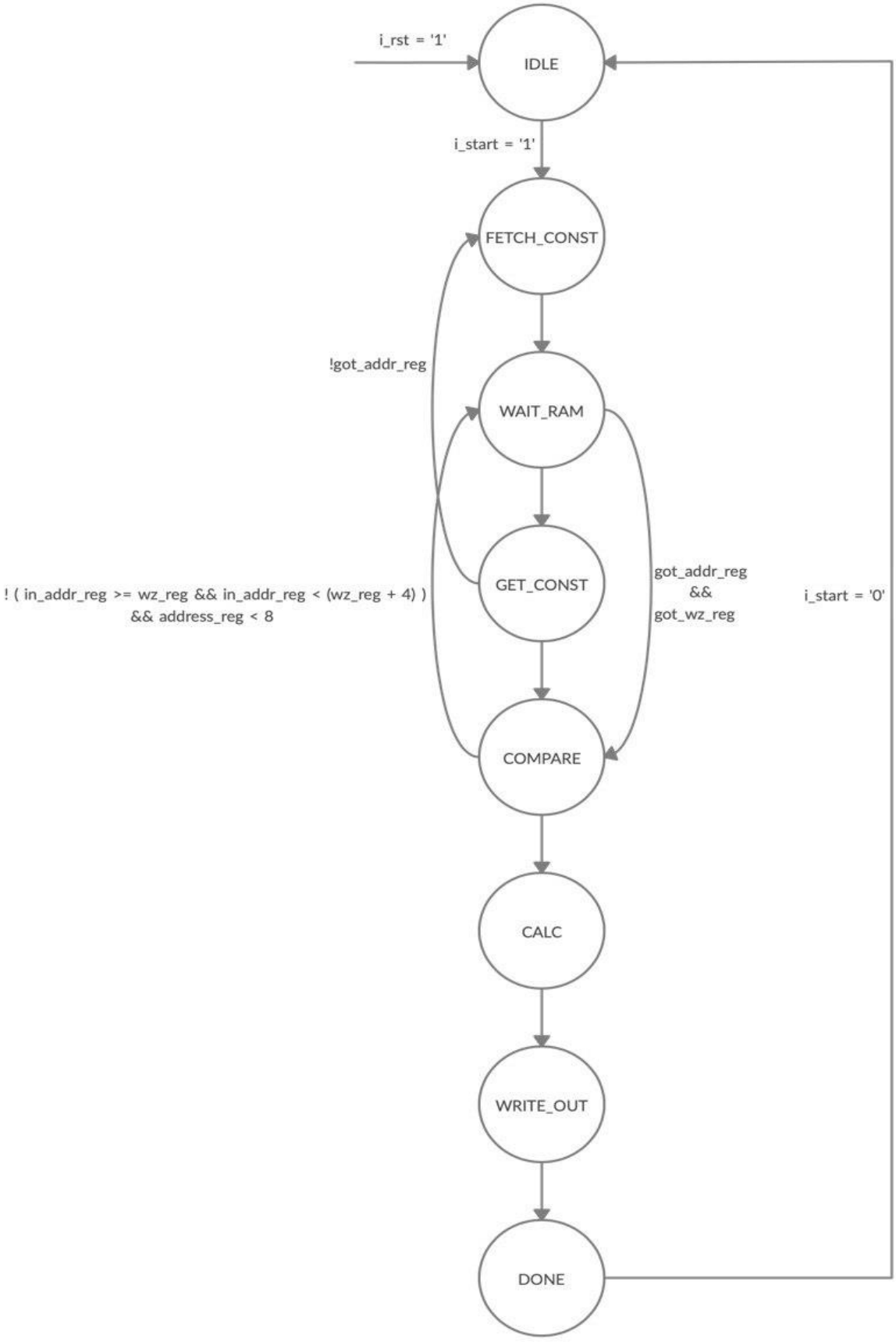

Figura 4: Macchina a Stati con le principali condizioni di transizione

3. Risultati dei test

Per verificare il corretto funzionamento del componente sintetizzato, dopo averlo testato con i test bench di esempio, abbiamo definito altri 6 test (tra i quali anche quelli che spingono la simulazione verso i corner case) in modo da cercare di massimizzare la copertura di tutti i possibili cammini che la macchina può effettuare durante la computazione.
Di seguito è fornita una breve descrizione dei test utilizzati e, per quelli più significativi, viene anche mostrato l'effettivo funzionamento corretto, grazie allo *screenshot* dell'andamento dei segnali durante la simulazione.

I test bench per la verifica dei corner case sono 4:

1. **ADDR in ingresso minimo e non presente in nessuna WZ:**

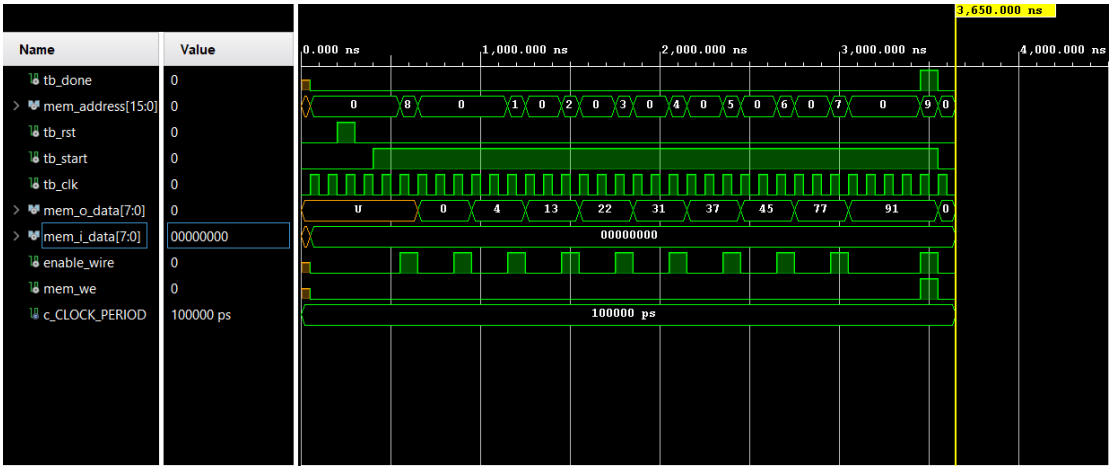

Figura 5: Simulazione con ADDR = "00000000"

2. **ADDR in ingresso minimo e presente nella prima WZ:**

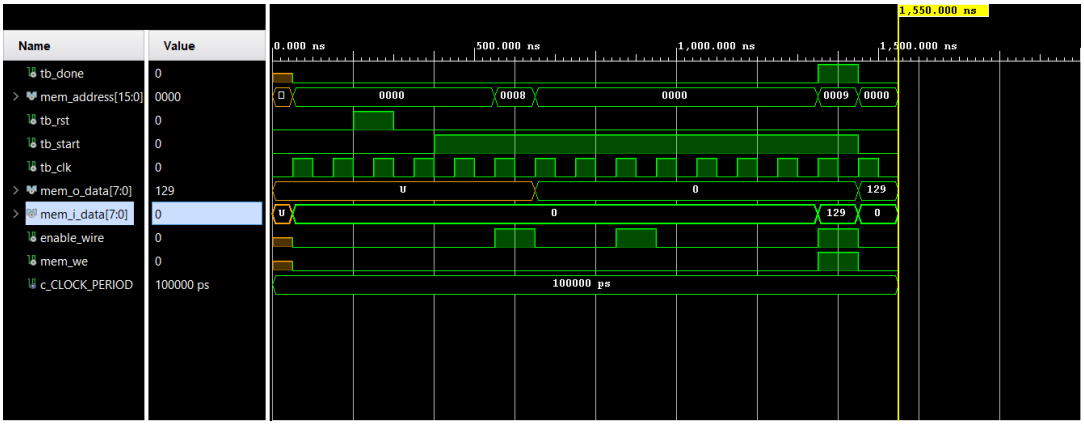

Figura 6: Simulazione con ADDR = "00000000"

3. **ADDR in ingresso massimo e non presente in nessuna WZ:**

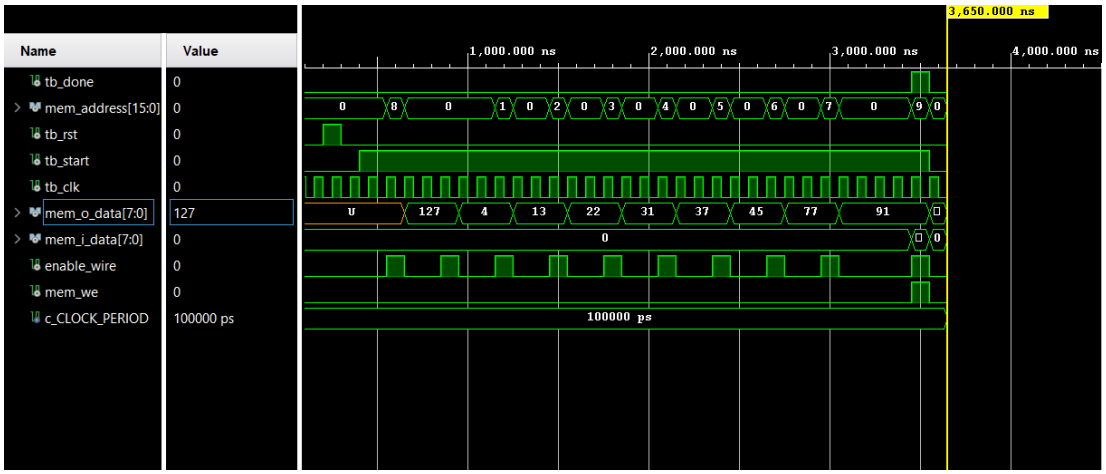

Figura 7: Simulazione con ADDR = "01111111"

4. **ADDR in ingresso massimo e presente nell'ultima WZ:**

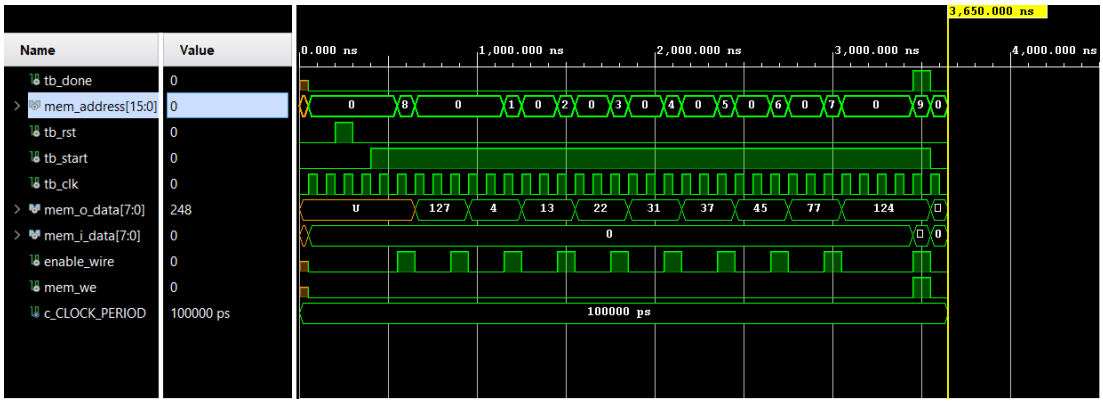

Figura 8: Simulazione con ADDR = "01111111"

I test bench, che verificano il corretto funzionamento dei segnali, sono 2:

1. **Reset Asincrono:** il test verifica che il trigger asincrono del segnale di reset non comprometta la computazione e che questa riinizi facendo ritornare la macchina nello stato iniziale `IDLE`.
2. **Doppia computazione:** il test verifica la corretta sincronizzazione dei segnali `i_start`, `i_rst` e `o_done` andando a simulare due volte di fila la stessa memoria.

Oltre ai test mirati precedentemente descritti, abbiamo deciso di simulare anche numerosi test randomici per testare ulteriormente il nostro componente. Attraverso un software in C, abbiamo generato un test bench con un gran numero di casi di test e li abbiamo simulati, in modo automatizzato, utilizzando uno script TCL, fornito a Vivado in modalità batch.

L'output dello script viene salvato in un file di log e successivamente si controlla che i messaggi delle assertions di ogni simulazione siano di superamento per ciascun test.

4. Conclusioni

4.1 Risultati della sintesi

Il componente sintetizzato supera correttamente tutti i test specificati nelle 2 simulazioni: *Behavioral* e *Post-Synthesis Functional*.
Qui di seguito è possibile vedere un confronto tra i tempi di simulazione dei due corner case che portano la macchina verso la più breve e la più lunga simulazione:

- **1550 ns:** tempo di simulazione (*Behavioral*) con ADDR presente nella prima WZ;
- **3650 ns:** tempo di simulazione (*Behavioral*) con ADDR non presente in nessuna WZ.

4.2 Ottimizzazioni

Le ottimizzazioni che abbiamo attuato sono state principalmente nella riduzione del numero di stati. In prima battuta, abbiamo steso una macchina che richiedeva tutte le Working Zone in stati differenti. Poi abbiamo deciso di utilizzare un approccio diverso che prevede di mantenere memorizzate meno informazioni possibili (come spiegato al punto 2.2). In questo modo abbiamo raggruppato le richieste di tutti i dati risparmiando così molti stati.