Progettazione e simulazione di un processore con pipeline basato su RISC-V

Fabio Pozzi - Matricola: 2143863

1 Introduzione

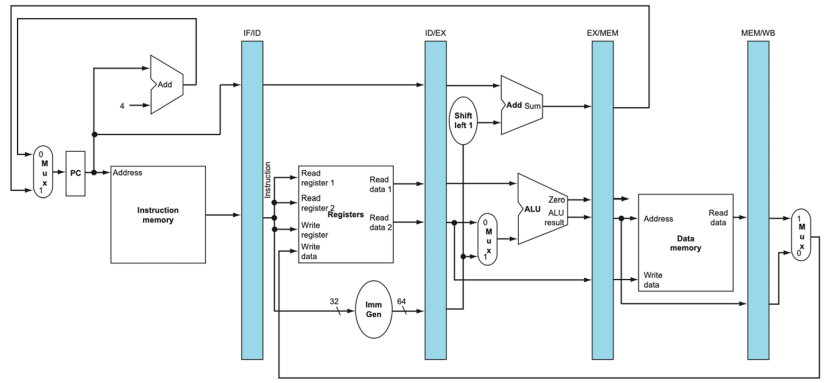

Figure 1: Implementazione del processore proposta da Patterson e Hennessy

Lo scopo del progetto è di progettare un processore basato sull'architettura RV64I, in grado di eseguire un algoritmo di bubble sort. Il processore non implementa l'intero instruction set, mentre è presente una pipeline funzionante. Il codice utilizzato per il bubble sort, che sarà commentato a pagina 7 è il seguente:

```
1  main:
2      #Imposto registro di partenza
3      addi x4 x0 0
4      addi x6 x0 1
5  loop:
6      beq x4 x5 check
7      ld x1 0(x4)
8      ld x2 8(x4)
9      addi x4 x4 8
10     blt x1 x2 loop
```

```
11      sd x2 -8(x4)
12      sd x1 0(x4)
13      add x6 x0 x0
14      beq x0 x0 loop
15  check:
16      beq x6 x0 main
17      addi x10 x0 1
18      beq x0 x0 0
```

Da qui è possibile dedurre le 6 istruzioni che ho deciso di implementare nel mio processore: `add addi ld sd beq blt`. La mia implementazione è basata sullo schema in figura 1, tratto da [2].

In un primo momento ho suddiviso i vari elementi in 4 componenti come in figura 2, ma successivamente ho preferito riorganizzare questi 4 componenti in 2 componenti come indicato nelle figure 3 e 4.

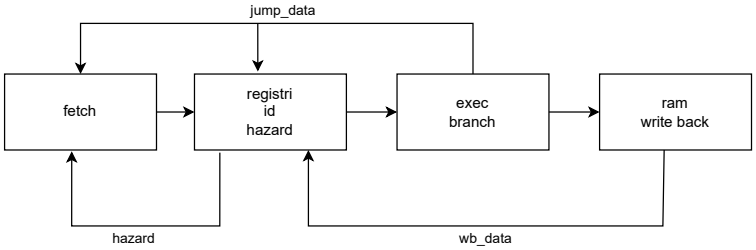

Figure 2: Implementazione iniziale del processore

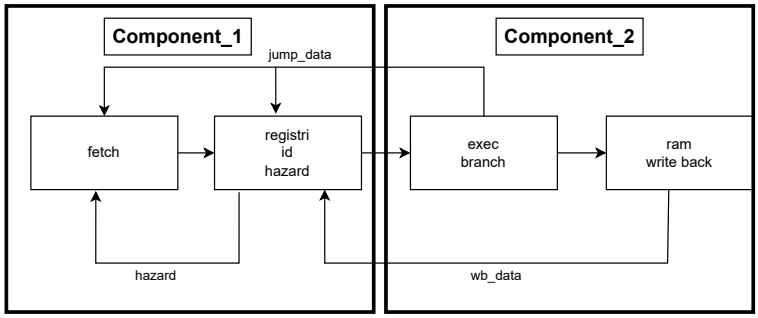

Figure 3: Riorganizzazione dei componenti

Al fine di una migliore comprensione, i nomi delle sezioni di questo documento sono gli stessi delle cartelle del mio codice. Ciascuna cartella contiene una sottocartella `src` che descrive i relativi componenti e una sottocartella `tb` che contiene i relativi testbench.

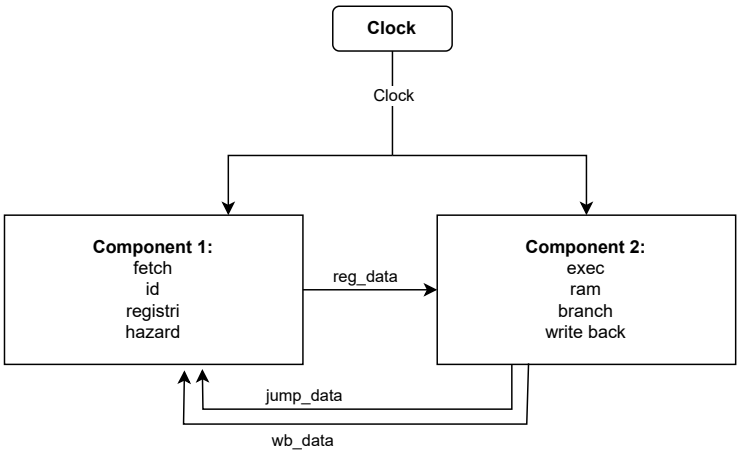

Figure 4: Implementazione finale del processore

Non sono presenti istruzioni per i salti incondizionati, ma è possibile utilizzare il comando `beq x0 x0` al loro posto.

2 ALU

L'implementazione dell'ALU è piuttosto naturale, ed è fondamentale per le istruzioni di `add addi blt beq`. Per prima cosa ho implementato un full adder e un 64 bit adder che funziona collegando 64 full adder in cascata.
In un secondo momento ho implementato un circuito sottrattore, che utilizza il complemento a 2 per cambiare il segno del numero da sottrarre e riutilizza il 64 bit adder per sommare i numeri così ottenuti. Il circuito sottrattore serve per poter confrontare due numeri per le istruzioni di `blt beq`.

Il componente di alu vero e proprio presenta un segnale di ingresso per scegliere quale istruzione eseguire tra addizione e le due istruzioni di comparazione. Se viene selezionata l'addizione il risultato in output sarà la somma tra i due valori di input, mentre nel caso della comparazione l'output sarà `1` se il risultato è positivo e `0` altrimenti.

3 Memory

Per l'implementazione delle memorie ho utilizzato il codice suggerito dal professore, implementando le modifiche utili a far funzionare il mio processore. I circuiti implementati per la memoria sono 4 e possono essere suddivisi in 2 categorie: da una parte ci sono i registri e il program counter, che sono molto simili

tra loro, e dall'altra ci sono la ram e la program memory.

3.1 Registri e program counter

L'implementazione dei registri si discosta di poco da quella che ha suggerito il professore come esempio [3].
Ad ogni indirizzo di memoria corrisponde un `std_logic_vector` da 64 bit. I registri consistono in un array di array e il contenuto dei registri è inizializzato a `0` all'inizio dell'esecuzione. È possibile leggere due registri per volta, ed è possibile scrivere un solo registro per volta. Il registro `x0` non può essere sovrascritto, ed il suo contenuto è sempre `0`.

Il program counter è invece implementato come un registro contenente un solo elemento. Il circuito del program counter è quindi una versione più semplice di quella descritta sopra: sono assenti le porte di input che servirebbero a selezionare quale indirizzo leggere o scrivere e la scrittura è sempre abilitata, in quanto questo è compatibile con l'implementazione che ho scelto per il mio processore.

In questi componenti la scrittura è sempre syncata con un segnale di clock, mentre la lettura è, per quanto possibile, istantanea.

3.2 RAM e instruction memory

Anche la RAM è basata sul codice consigliato dal professore [1]. La ram è implementata come un array di byte, ma i suoi input e output sono lunghi 64 bit. Quando si richiede di leggere o scrivere un valore, si va in realtà a leggere/scrivere 8 elementi contigui dell'array.
Anche gli indirizzi sono lunghi 64 bit, ma non tutti gli indirizzi sono realmente supportati. Ho implementato un numero minore di indirizzi sia per velocizzare le simulazioni, sia perché ho riscontrato numerosi crash quando ho provato ad avviare le simulazioni in ghdl con una ram troppo grande.
Ho inoltre implementato un array denominato `ram_debug`, che serve solo a semplificare la lettura della simulazione da parte di un umano. Gli elementi di questo array sono lunghi 64 bit e raccolgono 8 byte contigui della ram, in modo da non dover leggere i byte individuali.

Il funzionamento dell'instruction memory è concettualmente simile a quello di una RAM in sola lettura, ma l'implementazione presenta delle grosse differenze. La differenza più piccola sta nel fatto che l'output è di 32 bit anziché 64 bit, dato che questa è la dimensione delle istruzioni in questo ISA.
Il mio obiettivo era di poter leggere il codice da eseguire direttamente da un file binario, e ho trovato che leggere i dati all'interno di un `signal` fosse un processo lento e problematico. Ho quindi preferito definire l'array della instruction memory come una `variable`.
All'inizio dell'esecuzione `instruction memory` legge i dati da un file binario e, una volta che i dati sono stati letti, invia un segnale `ready` come output. Solo

quando instruction memory segnala di essere pronto il segnale di clock viene inviato anche agli altri componenti, e l'esecuzione vera e propria ha inizio.

Questa implementazione si è rivelata molto utile nella fase di test: mi ha permesso in modo molto rapido di scrivere del codice assembly per testare il corretto funzionamento di ciascun elemento del mio processore.

4 Tools

La cartella tools contiene altri due elementi molto importanti: la decodifica delle istruzioni e il riconoscimento degli hazard. Il componente di decode non fa altro che riconoscere e decodificare le varie istruzioni, e comunica come output le informazioni necessarie all'esecuzione.

Il componente di hazard implementa una FIFO e serve a garantire il corretto funzionamento in lettura e scrittura dei registri. La descrizione di questo componente si trova a pagina 6, dato che trovo più sensato descrivere la sua interazione con il resto del processore dopo aver descritto la struttura del processore stesso.

5 Stages

Il processore vero e proprio utilizza i componenti già descritti, organizzandoli in 4 *stages* che ho chiamato `fetch register_decode exec mem_wb`.

Il compito di `fetch` è quello di leggere una nuova istruzione ad ogni ciclo di clock. Qui si possono trovare il program counter e instruction memory, e un adder che incrementa il program counter. L'output di questo componente è del tipo `fetch_record` e contiene una istruzione in binario e il relativo program counter. Questo componente può ricevere anche degli input dall'esterno: si tratta dei jump dovuti a `blt` e `beq` e di un segnale che, in caso sia stato identificato un hazard, richiede che il program counter non venga incrementato.

I compiti di `register_decode` sono di gran lunga più complessi: deve decodificare le istruzioni, leggere e scrivere i dati dai registri e gestire gli hazard. Se non ci sono conflitti di hazard né jump il comportamento di `register_decode` è piuttosto semplice: deve decodificare l'istruzione e mandare in output il tipo di istruzione e i relativi dati, che possono includere dati letti dai registri e immediate. In caso di jump o se si riscontra un hazard l'output è semplicemente un `nop`.
In caso sia rilevato un hazard, un apposito segnale è inviato a `fetch`, in modo da interrompere temporaneamente l'incrementazione del `program counter`.

I compiti di `exec` sono nettamente più semplici. Se l'istruzione da eseguire è di tipo `add` o `addi`, esso deve calcolare il risultato dell'addizione e mandarlo come output a `mem_wb` insieme all'indirizzo del registro di destinazione. Se invece l'istruzione è del tipo `ld` o `sd`, deve calcolare il valore dell'indirizzo di ram

interessato e inviarlo come output a `mem_wb`. Se l'istruzione è di tipo `sd`, anche il dato da scrivere nella ram è inviato a `mem_wb`.
Se l'istruzione è invece di tipo `beq` o `blt`, `exec` deve usare `alu` per confrontare i due valori. In questo caso l'output inviato a `mem_wb` sarà di tipo `nop` e, se il confronto ha esito positivo, un segnale di jump viene inviato a `register_decode` e a `fetch`.

L'ultimo elemento è `mem_wb`. Il compito principale di questo elemento è di leggere e scrivere i dati nella ram. Nel caso delle istruzioni di tipo `add addi ld` è necessario scrivere dei dati nei registri. In questo caso un segnale di output è inviato a `register_decode`, indicando l'indirizzo del registro interessato e i dati che devono essere scritti.

5.1 Hazard

La gestione degli hazard implementa una FIFO.
Quando una istruzione che richiede la scrittura di un registro raggiunge `register_decode`, essa viene comunicata al componente `hazard` che aggiunge l'indirizzo del registro alla lista dei registri in attesa di scrittura. Quando poi `register_decode` riceve una richiesta di scrittura da `ram_wb`, questa richiesta viene anche comunicata al componente `hazard`, che dovrà solo eliminare l'elemento più vecchio della lista.

Se invece `register_decode` decide che è necessaria la lettura di un registro, andrà ad interrogare il componente `hazard` per verificare se esista qualche conflitto. Se il registro da leggere è in attesa di scrittura, l'esecuzione dell'istruzione dovrà essere ritardata e un segnale di `stall` sarà inviato a `fetch`. In questo caso l'output di `register_decode` è del tipo `nop` e il program counter non viene incrementato.

Quando finalmente `register_decode` riceverà la richiesta di scrittura del registro in conflitto da `ram_wb`, l'indirizzo del registro sarà rimosto dalla FIFO, il segnale di stall diventerà `false` e l'esecuzione del codice riprenderà regolarmente.

6 Components

Ho ritenuto opportuno riorganizzare gli elementi descritti in `stages` in due componenti, che ho denominato `component_1` e `component_2`.

Questa suddivisione non era affatto necessaria, ma mi ha aiutato ad organizzare il codice in una struttura che trovo sia più facile da comprendere, soprattutto in fase di debug. Trovo più semplice gestire e comprendere la comunicazione tra questi due componenti, rispetto a quella tra i 4 `stages`.
`component_1` contiene al suo interno `fetch` e `register_decode`, mentre `component_2` contiene `exec` e `ram_wb`.

Dal mio punto di vista il comportamento di `component_2` è più *lineare*, nel senso che questo componente deve limitarsi a eseguire le istruzioni che riceve ad ogni ciclo di clock. Questo componente riceve come input dei dati da elaborare e manda come output, se necessario, delle richieste di jump e di scrittura dei registri.

Il comportamento di `component_1` è invece più intricato. In assenza di jump e hazard questo componente si limita a recuperare una istruzione per ogni ciclo di clock, decodificarla e richiederne l'esecuzione a `component_2`. Quando invece questo componente rileva un hazard o riceve una richiesta di jump, il suo comportamento cambia e il suo output diventa di tipo `nop`.

7 Processor

Questa cartella consiste nell'ultimo step e descrive il processore vero e proprio. Il file `template.vhd` implementa il processore. Qui sono presenti il clock, e vengono messi in comunicazione i due componenti `component_1` e `component_2`. Gli altri file presenti nella cartella `processor/tb` non apportano alcuna modifica al processore. Ciascuno di questi file fa eseguire al processore un binario differente.

Lo stesso discorso vale per il file `processor/bubble_sort.vhd`: anche qui non viene modificata la logica descritta in `processor/template.vhd`, ma viene solo eseguito il programma di bubble sort vero e proprio.

8 ASM

Questa cartella contiene i vari codici che sono stati usati per il debug del processore e il programma di bubble sort. Tutti i programmi sono stati scritti in assembly e sono stati compilati con successo dal programma `ripes`.
La maggior parte dei programmi implementati in questa cartella servono solo a testare il comportamento di una singola funzione, mentre `fibonacci.s` implementa il calcolo dei primi numeri della sequenza di fibonacci e permette di testare nella stessa esecuzione il corretto funzionamento di tutte le istruzioni implementate.

Il programma più importante per il progetto è senza dubbio `bubble_sort.s`. Qui è implementato l'algoritmo di bubble sort secondo il seguente codice:

```
1     addi x1 x0 777
2     #Questo e' il mio registro di partenza
3     addi x2 x0 0
4     sd x1 0(x2)
5     addi x2 x2 8
6     addi x1 x0 929
7     sd x1 0(x2)
```

```
8       addi x2 x2 8
9       addi x1 x0 3
10      sd x1 0(x2)
11      addi x2 x2 8
12      addi x1 x0 515
13      sd x1 0(x2)
14      addi x2 x2 8
15      addi x1 x0 222
16      sd x1 0(x2)
17      #Questo e' il mio registro di arrivo
18      add x5 x2 x0
19  main:
20      #Imposto registro di partenza
21      addi x4 x0 0
22      addi x6 x0 1
23  loop:
24      beq x4 x5 check
25      ld x1 0(x4)
26      ld x2 8(x4)
27      addi x4 x4 8
28      blt x1 x2 loop
29      sd x2 -8(x4)
30      sd x1 0(x4)
31      add x6 x0 x0
32      beq x0 x0 loop
33  check:
34      beq x6 x0 main
35      addi x10 x0 1
36      beq x0 x0 0
```

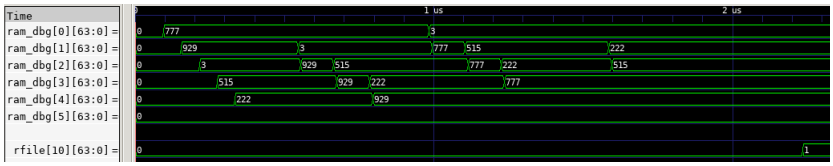

Figure 5: Esecuzione del bubble sort

Le prime 18 righe servono solo a caricare nella ram dei numeri casuali, mentre il programma vero e proprio inizia dall'etichetta `main`. La logica del programma è quella del bubble sort. Il programma si aspetta di trovare nei registri `x4` e `x5` l'indirizzo minimo e massimo della ram da considerare, e riordina tutti i numeri che trova in questo range di indirizzi. Quando il programma verifica che tutti i numeri presenti sono in ordine, segnala il termine dell'esecuzione scrivendo il numero `1` nel registro `x10` ed entra in un loop infinito con l'istruzione `beq x0 x0 0`.

Come si può vedere dalla figura 5, l'esecuzione del bubble sort procede correttamente.

References

[1] Doulos. *Simple RAM Model*. URL: `https://www.doulos.com/knowhow/vhdl/simple-ram-model/`.

[2] Patterson D.A. Hennessy J.L. *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)*. Morgan Kaufmann Publishers, 2018.

[3] Aleksandar Milenkovic. *Advanced VLSI Design Lecture 6: VHDL Synthesis*. URL: `http://www.ece.uah.edu/~milenka/cpe626-04F/lectures/l06_vhdl_syn_3p1.pdf`.