SPP - Simple Pipelined Processor

Stefano Mercogliano - M63000983

Contents

La presente documentazione è volta alla spiegazione del semplice processore da me implementato al fine di approfondire e consolidare le conoscenze relative al pipelining ottenute durante il corso di calcolatori elettronici 2. si cercherà di esporre da prima il processo di progettazione di una architettura single cycle, per poi trasformarla in una architettura pipelined in grado di evitare Hazard strutturali, e gestire in maniera approfondita quelli relativi ai Dati, e in maniera superficiale quelli per i salti.

1 Progettazione

Il processore di cui discuteremo di qui in avanti prenderà il nome di **SPP**, o Simple pipelined processor, che andremo a progettare dalle fondamenta fino ad arrivare ad una struttura completa e funzionante in grado di realizzare semplici programmi attraverso un certo ISA.
Il progetto utilizzerà come falsariga i modelli architetturali MIPS e DLX, partendo da una progettazione single cycle, passando poi per una versione pipelined.

Facendo riferimento alle architetture prima citate, ipotizziamo di introdurre i primi elementi architetturali della nostra architettura:

- **Program Counter** : classico registro che punti alla successiva istruzione da eseguire
- **Instruction Memory** : Memoria per le istruzioni
- **Registers** : insieme dei registri ad uso generale del processore
- **ALU** : unità logico aritmetica
- **DM** : Memoria per i dati

Possiamo ora fare alcune assunzioni relative alla memoria : questa sarà accedibile contemporaneamente e in due sezioni logicamente distinte, ovvero quella relativa alle istruzioni e quella relativa ai dati, secondo un modello *Harvard*. Stiamo inoltre ipotizzando che tali memorie coincidano con delle cache di primo livello, e con un assenza completa di cache miss. Poste queste ipotesi, possiamo ragionare sullo spazio di indirizzamento del processore e sul suo parallelismo.

Per quanto ne concerne lo *spazio di indirizzamento*, la scelta è stata quella di indirizzi su 8 bit, per un totale di 256 locazioni di memoria sia per il porto istruzioni, che per quello dati. Sebbene possano sembrare pochi, e di fatti lo sono, la scelta del dimensionamento del sistema è legata principalmente ad una semplice gestione dei segnali di controllo, e ad una immediata decodifica visiva delle istruzioni, al fine di poter testare rapidamente il funzionamento

dell'architettura.

Il parallelismo previsto invece sarà di 16 bit per parola, ovvero, ad ogni indirizzo corrisponderà una parola di 16 bit. Ne consegue allora che il PC presenterà una dimensione di 8 bit, mentre i registri generali saranno di 16 bit, e nel dettaglio, ne implementeremo 16, $R_0, R_1, \cdots R_F$.

1.1 Instruction set architecture

A valle di quanto detto in precedenza, possiamo assumere che la generica istruzione prelevata dall'IM avrà una dimensione totale di 16 bit. Tale **istruzione**, dovrà essere strutturata in maniera appropriata al fine di realizzare correttamente le operazioni che ci aspettiamo di poter effettuare.

Nel mio caso specific ho deciso di prevedere, in accordo a quando accadeva nel DLX, due istruzioni per l'interfacciamento memoria dati, ovvero una Load Word ed una Store Word. Inoltre era necessario introdurre operazioni di ADD e SUB, non chè di addizioni con immediati, per poter testare completamente conflitti sui dati.
Infine le altre istruzioni fondamentali per testare i rimanenti conflitti, ovvero quelli sul controllo, dovevano essere quelle di Jump e di Branch. Per semplificare l'architettura, l'unico tipo di salto condizionato implementato è quello di Branch on equal.

Per semplicità di testing, si è deciso di codificare tutti i possibili codici operativi su 4 bit, ovvero una cifra esadecimale. Ciò significa allora che i 4 bit più significativi della istruzione saranno relativi all' **OpCode**; possiamo dunque riassumere in tabella l'ISA del processore SPP:

OPCODE	mnemonic	descrizione
0x0	NOP	Nessuna istruzione
0x1	ADD	Addizione tra due operandi
0x2	SUB	Sottrazione tra due operandi
0x3	ADDi	Addizione con immediato
0x4	OR	Or logica tra due operandi
0x5	ORi	Or logica con immediato
0x6	AND	And logica tra due operandi
0x7	ANDi	And logica con immediato
0x8	MOVE	spostamento dati da un registro ad un altro
0x9	SETR	inizializzazione di un registro
0xA	SW	caricamento in memoria
0xB	LW	caricamento dalla memoria
0xC	JAM	salto incondizionato
0xD	BEQ	salto condizionato su uguaglianza
0xE	SHR	shift a destra
0xF	SHL	shift a sinistra

Per arricchire tale tabella si rende necessario stabilire come gli operandi vengano estratti e gestiti per ogni codice operativo. l'SPP non prevede modalità di indirizzamento particolari in quanto gli operandi sono interamente codificati all'interno di una singola istruzione. Vale la pena però distinguere come però i 12 bit successivi all'opcode siano codificati. Tale codifica prende largamente spunto dalle architetture MIPS, adattando però il formato delle parole R,I,J alle esigenze del nostro sistema.

- **Operandi registri** :Avendo a disposizione 12 bit, è immediato codificare su ogni nibble uno dei possibili 16 registri
- **Operandi immediati** : i successivi 4 bit all'OpCode codificano per la sorgente, i successivi 4 invece vengono codificati per l'immediato (da 0 a 15 dunque), e gli ultimi 4 saranno dedicati al registro sorgente. Fanno eccezione a tal proposito le istruzioni di SETR, LW e SW che avranno codificati i loro immediati su 8 bit, invece che su 4.
- **Operandi di salto** Similmente al caso precedente, 8 bit saranno dedicati all'indirizzo per il JAM, mentre gli ultimi 4 rimarranno inutilizzati.

Nel caso del BEQ invece saranno previsti 2 operandi registri per il confronto, e gli ultimi 4 bit saranno usati per codificare l'offset (signed) di salto rispetto all'attuale PC.

possiamo quindi presentare una versione arricchita dell'ISA presentato in precedenza con degli operandi d'esempio

OPCODE	mnemonic	Operandi	Codice
0x0	NOP		0x0—
0x1	ADD	R1,R2,R3	0x1123
0x2	SUB	R1,R2,R3	0x2123
0x3	ADDi	R1,A,R3	0x31A3
0x4	OR	R1,R2,R3	0x4123
0x5	ORi	R1,1,R3	0x5113
0x6	AND	R1,R2,R3	0x6123
0x7	ANDi	R1,1,R3	0x7113
0x8	MOVE	R2,R3	0x82-3
0x9	SETR	R2,FF	0x9FF2
0xA	SW	R1,FF	0xA1FF
0xB	LW	R1,FF	0xBFF1
0xC	JAM	R2,AB	0xCAB-
0xD	BEQ	R2,R1,6	0xD216
0xE	SHR		
0xF	SHL		

1.2 Datapath

Possiamo ora introdurre il datapath e come esso sia stato sviluppato a partire dalle istruzioni precedentemente presentate.
Partendo dal presupposto di avere come elementi fondamentali quelli presentati nella sezione relativa alla progettazione, diventa vitale capire come interconnettere dali componenti, e cosa applicare al *contorno*.

come prima cosa introduciamo la differenziazione doverosa tra **Unità operativa** che realizza le funzionalità della macchina sequenziale che stiamo realizzando, e tra l'**Unità di controllo**, atta invece alla generazione dei seg-

nali per *controllare* tale sistema.

è chiaro che per poter descrivere correttamente una unità di controllo è anzitutto necessario progettare il datapath per l'unità operativa; secondo questo approccio quindi cerchiamo di descrivere come questo datapath sia stato realizzato.

In primo luogo è necessario poter incrementare il program counter tramite un addizionatore ad 8 bit, che altro non farà che incrementare il valore precedentemente salvato nel PC.
Tale nuovo indirizzo però non è sufficiente, in quanto potrebbe esser necessario aggiungere a tale indirizzo o un offset dovuto a dei **Salti condizionati**, oppure uno dovuto a dei **salti non condizionati**.
Per completare la struttura di **fetch** sarà quindi necessario introddurre due multiplexer che stabiliscano se inoltrare in retroazione al Program Counter, l'indirizzo successivo, l'indirizzo spiazzato, oppure l'indirizzo specificato dal salto non condizionato.

Una volta stabilito l'indirizzo della prossima istruzione, tale indirizzo sarà inoltrato alla **Instruction Memory**, che, una volta letto il dato, effettuerà il **Parsing** dell'istruzione, ovvero la sua decodifica, inoltrando parti di questa istruzione verso i differenti componenti dell'architettura.

Secondo il modello di programmazione presentato precedentemente, siamo sicuri di dover poter effettuare operazioni di tipo *registro*.
Posta la codifica prima presentata, è allora chiaro che i bit 11...8 saranno usati per selezionare il primo registro sorgente, i bit 7...4 per il secondo registro sorgente, e infine i bit 3...0 per il registro destinazione.

sappiamo inoltre però che l'istruzione di salto condizionato prevede un offset su 4 bit contenuto negli ultimi 4 bit dell'istruzione. Tali 4 bit, opportunamente estesi tramite un **sign extender**, saranno sommati all'indirizzo successivo su 8 bit, così da poter generare l'eventuale segnale di salto da inoltrare al multiplexer relativo al salto condizionato citato in precedenza.

Nel caso di istruzioni immediate, esse presentano 3 tipi di formati ben distinti, che possiamo riassumere di seguito:

- Un primo caso è relativo alle operazioni di ADDi, ORi,ANDi, che prevedono di inoltrare un immediato nei bit 7...4.
- Nel secondo caso stiamo considerando un immediato su 8 bit trasportato nei bit 7...0, come accade ad esempio nell'operazione di STORE
- Terzo caso è relativo ad un immediato su 8 bit registrato sui bit 11...4 come accade nelle istruzioni di SETR e di LOAD.

per poter discriminare quali dei 3 immediati dovrà esser prelevato faremo riferimento ad un multiplexer a 3 ingressi ed una uscita, ad un solo bit.
Per terminare tale discorso vale la pena ricordare che l'istruzione di JAM conterrà l'indirizzo al quale saltare nei bit 11...4, che verranno quindi inoltrati al multiplexer relativo al salto non condizionato citato nella struttura di fetch, la quale ora sarà operativamente completa.

la struttura così descritta possiamo assimilarla ad una struttura di **Decodifica**, che può coincidere con l'omonimo stato del ciclo del processore.

Tocca adesso coprire la struttura di **esecuzione**, costituita dalla ALU; questa prevederà una serie di operazioni interne per realizzare tutte le funzionalità descritte dai codici operativi. In aggiunta a ciò saranno fondamentali i due operandi in ingresso, entrambi da 16 bit, e un risultato in uscita, anch'esso su 16 bit, con l'ulteriore presenza di un bit flag Z che sarà poi usato per la gestione dei salti.

L'alu riceverà come primo operando quello direttamente prelevato dalla struttura register, e come secondo operando o il secondo registro selezionato dai registers, oppure uno degli operandi opportunamente estesi a 16 bit. Tale decisione vedremo, ricadrà tra i compiti della unità di controllo.

Le operazioni interne dell'ALU implementate sono 8, e possiamo riassumerle in tabella:

OP	descrizione
000	Restituisci operando A
001	Restituisci operando B
010	Somma A e B
011	Sottrai A a B
100	AND tra A e B
101	OR tra A e B
110	Shift a destra di A (futuri aggiornamenti)
111	Shift a sinistra di A (futuri aggiornamenti)

è ora necessario descrivere la il comportamento relativo alla struttura di **memorizzazione e scrittura**; la memoria riceverà in ingresso sia il dato prodotto dall'ALU che un indirizzo da 8 bit. Data la struttura delle due istruzioni di memorizzazione LW e SW, siamo certi che l'indirizzo sarà contenuto nella parte dell'istruzione atta a contenere l'immediato. Ciò significa allora che l'uscita del multiplexer relativo alla selezione dell'immediato verrà anche inoltrata al porto di memorizzazione.

Infine l'eventuale dato tornato dalla memoria, oppure il risultato dell'ALU, verranno inoltrate in retroazione verso la struttura dei registri generali, al fine di salvare il dato nel registro specificato dal segnale di selezione per il registro destinazione. a tal proposito faremo allora uso di un ultimo multiplexer.

Il datapath, e quindi la struttura funzionale del processore, è ormai realizzata e teoricamente funzionante. Prima ancora di parlare di tempificazione, cerchiamo di capire come i segnali di controllo per l'abilitazione di tale datapath vengano gestiti.

L'idea alla base per la realizzazione di tale unità è quella di utilizzare una tecnica *microprogrammata*, facendo quindi riferimento ad una LUT e ad un automa d'appoggio. Tale schematizzazione era solo una bozza iniziale del progetto, che in fase finale funzionerà unicamente con una Look Up Table che assurgerà al ruolo di MicroRom.

Essendo questo un processore single cycle, ci aspettiamo che ogni istruzione riesca a completare in un singolo colpo di clock, a meno di interazioni con

la memoria, ragion per cui ad ogni istruzione corrisponderà una sola entry nella LUT, che genererà i segnali abilitanti per quella specifica istruzione.

Essendo il codice operativo chiave d'accesso per questa microRom, sarà necessario inoltrarvi solo i 4 bit più significativi della istruzione attualmente prelevata ; verrà così selezionata la *control word* relativa, che abiliterà tutti i componenti richiesti.

Una singola control word è composta da 12 bit, ovvero il numero di segnali necessari a controllare il datapath. Tali segnali saranno in ordine:

- *ControlWord[11] - Write register signal* : segnale per l'abilitazione alla scrittura nei registers
- *ControlWord[10] - Alu Source signal* : segnale per la selezione dell'operando sorgente 2 per l'alu (registro o immediato)
- *ControlWord[9..7] - Alu Operation signal* : segnale per la selezione dell'operazione da far effettuare all'ALU
- *ControlWord[6] - Branch signal* : segnale per l'abilitazione del multiplexer per il branch (messo in and con il flag Z)
- *ControlWord[5] - Jump signal* : segnale per l'abilitazionedel multiplexer per il salto non condizionato
- *ControlWord[4] - Write Memory signal* : segnale per l'abilitazione alla scrittura sulla memoria
- *ControlWord[3] - Read Memory signal* : segnale per l'abilitazione alla lettura dalla memoria
- *ControlWord[2] - Memory or Result signal* : segnale per l'abilitazione del multiplexer di selezione tra risultato ottenuto dalla ALU o dalla memoria
- *ControlWord[1..0] - Immediate signal* : segnali per la selezione della tipologia di immediato

possiamo così presentare per ogni codice operativo, quali segnali vengono specificamente attivati nella LUT, e dunque l'organizzazione della stessa:

segnali	opcode
000000000000	NOP
110100000000	ADD
110110000000	SUB
100100000000	ADDi
111010000000	OR
101010000000	ORi
111000000000	AND
101000000000	ANDi
100000000000	MOVE
100010000010	SETR
000000010001	SW
100000001110	LW
000000100000	JAM
010111000000	BEQ

Per concludere la trattazione relativa al datapath, vale la pena discutere relativamente alla *tempificazione* dell'intera architettura.
Partiremo dall'ipotesi forte che la memoria astragga una cache tanto grande quanto la memoria centrale e che non sia possibile avere cache miss. Ciò si traduce nell'avere sia memoria IM che memoria DM in grado di rispondere nell'arco di un colpo di clock; nella fattispecie le due memorie leggeranno un dato sul fronte di salita e lo presenteranno sul fronte di discesa.

Stabilito ciò, supponiamo che PC e i registri ad uso generale Registers, siano tutti sensibili al fronte di *discesa* del clock; tutti gli altri componenti in gioco sono componenti combinatori, e quindi non necessiteranno di segnali di Clock o reset.
Poichè tale architettura sarà solo simulata non sarà necessario porsi il problema del dimensionamento dei due fronti di clock; ci è infatti sufficiente sapere che tali fronti dovranno essere ampi a sufficienza per far stabilizzare tutti i segnali e commutare tutti i componenti.

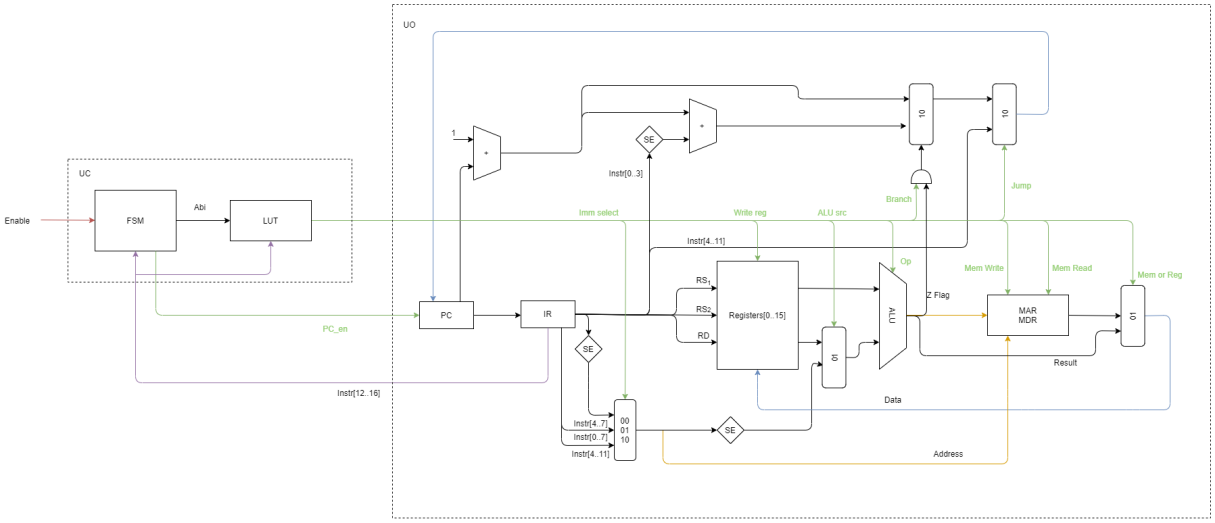

Figure 1: Datapath del processore SPP single cycle

2 Pipelining

Il processore come presentato in precedenza rappresenta una buona base per poter implementare del pipelining, e quindi incrementare il throughput dell'intero sistema.
Partiremo presentando implementazione delle Pipes e la suddivisione del processore in stages, passando poi per l'identificazione dei possibili conflitti e la loro risoluzione.

2.1 Pipes e Stages

Il sistema è stato realizzato sulla falsariga di MIPS e DLX con l'obiettivo poterlo suddividere nettamente in 5 fasi di elaborazioni tra loro distinte e indipendenti. Possiamo infatti scorgere e definire tali 5 fasi come di seguito

1. **Fase di Fetch** : Fase comprendente il PC e l'interazione con la memoria IM;
2. **Fase di Decode** : Fase relativa al parsing della istruzione e del direzionamento dei segnali, nonchè della selezione dei registri
3. **Fase di Execute** : Fase relativa al calcolo tramite ALU
4. **Fase di Memorizzazione** : Fase relativa all'interfacciamento con la memoria, contenente
5. **Fase di writeBack** : Fase di scrittura sui registri interni al processore

L'architettura, così come realizzata, si presta bene alla realizzazione del passamano, tuttavia bisogna fare alcuni accorgimenti; in primo luogo è necessario introdurre le 4 Pipes, ovvero dei registri funzionalmente identici, ma che presentino ovviamente interfacce diverse. Tali registri dovranno prendere in consegna tutti i segnali prodotti da una fase, così da inoltrarli allo stage successivo.

Per esempio la pipe posta tra la fase di Instruction Fetch e quella di Instruction Decode, della Pipe_IF_ID, dovrà ricevere in ingresso i 16 bit

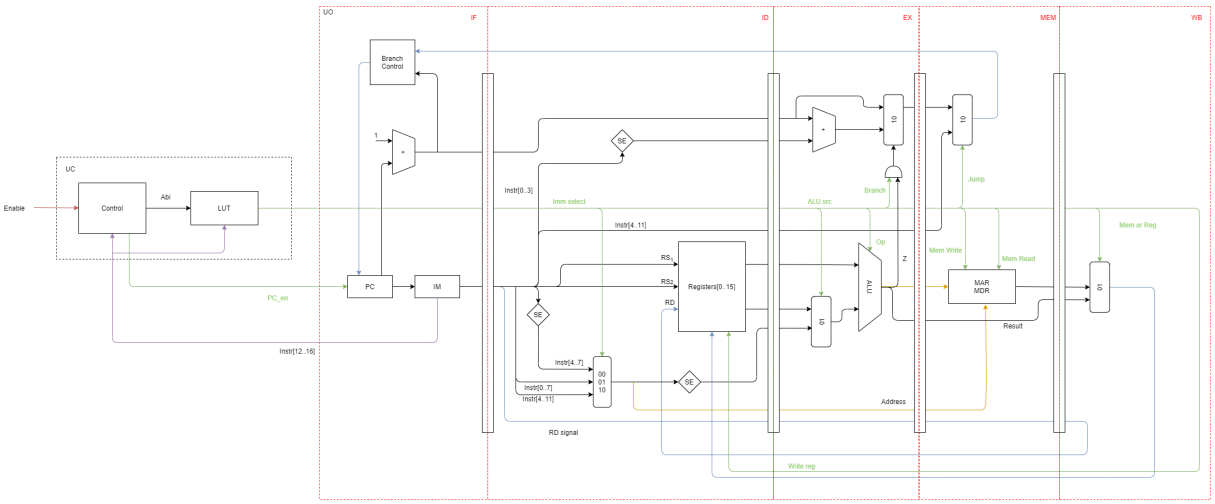

Figure 2: Datapath del processore SPP con pipeline a 5 stadi

dell'istruzione, gli 8 bit dell'indirizzo successivo da inoltrare, e ovviamente i 12 segnali di abilitazione prodotti dalla LUT.

Inoltre, al fine di poter garantire un corretto funzionamento del sistema sarà necessario rendere le pipes sensibili sul fronte di *salita*.
Nonostante l'architettura si presti ad una immediata trasformazione in una architettura pipeline, per evitare conflitti strutturali bisogna effettuare un piccolo accorgimento relativo alla selezione del registro destinazione interno; questo dovrà essere inoltrato negli stage successivi e mandato in retroazione solo quando l'istruzione richiedente la scrittura in quella specifica destinazione non avrà raggiunto la fase di Write Back, una fase nella quale sarà possibile scrivere nei registri.

A valle di quanto appena esposto possiamo quindi presentare una prima versione dell'architetture pipelined dell'SPP, ancora scevra da qualsivoglia considerazione relativa a conflitti.

2.2 Gestione dei conflitti

Arrivati a questo punto , in linea teorica, una pipeline a 5 stadi potrebbe garantire uno speedup netto di 5 volte rispetto alla sua controparte non pipelined presentata a inizio documento. Tuttavia questo risulta esser vero solo in condizioni specifiche in cui non si verificano istruzioni di salto, di interfacciamento memoria e di dipendenza d'uso di operandi tra più istruzioni.

ci rendiamo allora conto che situazioni eccezionali, che costituiscono in realtà la normalità dell'esecuzione, manderebbero in crisi il sistema così come presentato. è allora necessario trovare delle soluzioni per quelli che vengono chiamati **Conflitti**, o **Hazards**.

tali conflitti possono solitamente essere di 3 tipologie distinte, ovvero *strutturali*, sui *dati* e infine di *controllo*. Fortunatamente la prima tipologia di hazard non è riscontrabile in quanto l'architettura è progettata per evitare tali tipi di conflitto.

Infatti la divisione della memoria in una parte per i dati e una per le istruzioni impedisce conflitti tra la fase di IF e quella di MEM; ciò avviene sostanzialmente per ogni componente hardware, che riesce a mantenersi ben disaccoppiato dagli altri, al fine di non generare conflitti di questo tipo.

discorso ben diverso vi è da fare negli altri due casi, che approfondiremo di qui in avanti e che ci porterrano ad implementare sostanziali cambiamenti nell'architettura fino ad ora introdotta.

2.2.1 Control Hazard

La gestione dei conflitti di controllo, e quindi relativi a salti condizionati e non, risulta essere una delle parti più delicate nella progettazione di un processore pipelined. Le tecniche previste possono infatti essere sia statiche che dinamiche, e tra quelle più interessanti troviamo sicuramente le tecniche di **Branch prediction**, che fanno uso di memorie associativa ausiliarie chiamate *branch prediciton table*, accoppiate a degli automi dei *predittori*.

Nonostante l'importanza di tale tecnica, l'implementazione finale di questo processore prevederà solo una copertura estensiva dei conflitti sui dati, scegliendo un approccio risolutivo molto più semplice ma inefficiente per la gestione di conflitti di controllo.

La tecnica in questione è quella del **Flushing** della pipe, realizzato mediante una apposita unità di controllo chiamata **Branch control**.

tale unità prevederà di ricevere in ingresso i segnali di *Jump* e *Branch Taken*, inoltrati dalla fase di EXE, insieme ovviamente all'indirizzo eventualmente calcolato da queste due istruzioni. Se almeno uno dei due segnali di controllo risulta essere alto, la branch control unit inoltrerà al program counter l'indirizzo retroazionato piuttosto che il nuovo PC incrementato.

Tuttavia in caso di salto , il processore avrebbe già riempito le pipe delle fasi di IF e ID con due istruzioni "Errate", ragion per cui si rende necessario lo svuotamento delle pipes, anche detto **Flush**.

Nonostante questa sia stata la tecnica finale implementata, per futuri aggiornamenti propongo qui in avanti una possibile implementazione per la tecnica di branch prediction, con un predittore a 2 bit.

Tale tecnica prevede infatti di costruire dinamicamente la Branch Prediction table, con ogni entry caratterizzata dall'indirizzo al quale è presente l'istruzione di salto, che fungerà da chiave d'accesso, l'indirizzo al quale saltare a seguito di un branch, e 2 bit per la descrizione dell'automa.

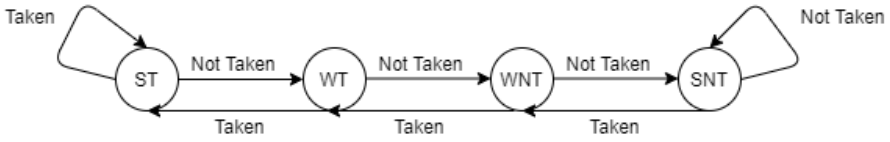

Figure 3: Predittore a 2 bit

Dunque all'atto della retroazione dei segnali di branch taken, jump e dell'indirizzo *jump_addr*, si valuta lo stato del predittore.
Tale predittore è un automa a stati finiti, composto da 4 stati:

- **Stato Weak Taken** : si prevede di effettuare un salto, e quindi la BCU inoltra al PC l'indirizzo letto nella BPT usando come chiave d'accesso l'indirizzo relativo alla attuale istruzione di salto. In caso di errore si paga penalità, ovvero si svuotano le pipes, e si transita nello stato *Weak Not Taken*, altrimenti si transita nello stato *Strong Taken*

- **Stato Strong Taken** : si prevede di effettuare un salto, e quindi la BCU inoltra al PC l'indirizzo letto nella BPT usando come chiave d'accesso l'indirizzo relativo alla attuale istruzione di salto. In caso di errore si paga penalità, ovvero si svuotano le pipes, e si transita nello stato *Weak Taken*, altrimenti si rimane nello stato *Strong Taken*

- **Stato Weak Not Taken** : si prevede di NON effettuare un salto, e quindi la BCU inoltra al PC l'indirizzo successivo. In caso di errore si paga penalità, ovvero si svuotano le pipes, e si transita nello stato *Weak Taken*, altrimenti si transita nello stato *Strong Not Taken*

- **Stato Strong Not Taken** : si prevede di NON effettuare un salto, e quindi la BCU inoltra al PC l'indirizzo successivo. In caso di errore si paga penalità, ovvero si svuotano le pipes, e si transita nello stato *Weak Not Taken*, altrimenti si rimane nello stato *Strong Not Taken*

Per popolare la branch prediction table si vuole fare riferimento alla seguente idea: Prelevata una istruzione, all'atto della sua decodifica, identificare se essa sia una istruzione di salto, oppure no. Nel primo caso la BCU dovrà stabilire se tale istruzione sia presente o meno nella BPT; in caso affermativo, la BCU inoltrerà nel PC l'indirizzo corretto in base allo stato del

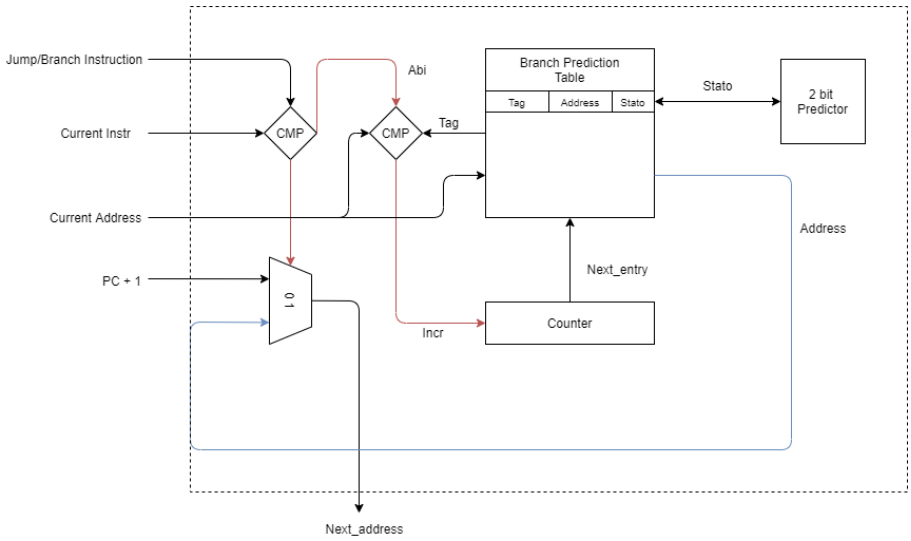

Figure 4: Schema di principo della BCU

predittore, mentre in caso contrario, la BCU dovrà inserire una nuova entry nella BPT, che consterà di 1 byte per l'indirizzo dell'istruzione corrente, 1 byte per l'indirizzo a cui saltare, e 2 bit per lo stato dell'automa, inizialmente pari a 10 (stato Weak Taken).
Allora in accoppiamento alla BPT vi sarà allora un contatore che terrà conto del numero di entry in tabella; ciò significa che superata la dimensione della tabella, le nuove entry sovrascriveranno le prime, cercando di rispettare una sorta di *principio di località temporale*.

2.2.2 Data Hazard

Ipotizziamo di dover far eseguire al processore così come presentato fino ad ora, il seguente programma:

```
I1 : ADD R1,R2,R3
I2 : ADDi R3,02,R4
```

L'architettura non è in grado di realizzare correttamente tali operazioni; il perchè è presto detto, in quanto il risultato fornito dalla prima istruzione nel registro R3, sarà disponibile solo a valle della fase di scrittura, e quindi dopo 3 colpi di clock rispetto alla fase di esecuzione della prima istruzione. Ne consegue che l'istruzione ADDi starà utilizzando un dato incoerente.

Le soluzioni a questo problema sono molteplici. Una prima idea potrebbe essere quella di bloccare la pipe, tecnica detta di **pipe interlocking**; tale scelta è però troppo poco efficiente, e ne esistono di differenti molto più efficaci. Una di queste è ad esempio il **forwarding con ibernazione**, che tramite l'ausilio di hardware aggiuntivo è in grado di gestire tali conflitti dinamicamente, permettendo una esecuzione delle istruzioni non necessariamente sequenziali.

In realtà la tecnica proposta per la risoluzione di tali conflitti, e quindi implementata successivamente, è quella di **Alu forwarding**.
Questa tecnica prevede di riportare in retroazione il risultato dell'ALU ottenuto in fase di esecuzione, dalla fase di memorizzazione.
Tale retroazione infatti consente alla seconda istruzione di avere disponibile il dato calcolato dalla precedente istruzione, in quanto tale risultato sarà stato calcolato già durante la fase di EXE della prima istruzione, e quindi finalizzato nella fase di MEM della prima. Ma allora durante la fase di EXE della seconda istruzione, usare il dato retroazionato permetterà di coprire la dipendenza prima esposta.

Tale dipendenza potrebbe però verificarsi anche tra lo stage di MEM di una prima istruzione, e lo stage di EXE della seconda istruzione. vediamo a tal proposito un esempio:

```
I1 : LW R1,10
```

```
I2 : ADD R1,R2,R0
```

Come possiamo notare, la prima istruzione prevede una LOAD dall'indirizzo 10 nel registro R1, per poi usare successivamente R1 per ottenere calcolare una somma da salvare in R0. Anche in questo caso però il dato a disposizione della seconda istruzione non sarà coerente, in quanto calcolato solo nella fase successiva.
Analogamente a prima l'idea è quella di retroazionare l'uscita dati dalla Data Memory, e retroazionarla verso gli operandi dell'ALU della fase di esecuzione.

Naturalmente ciò presupporrà l'introduzione di 2 multiplexer distinti che selezioneranno i due operandi dell'ALU tra le uscite dei registri, il risultato dell'ALU retroazionato e il dato memoria retroazionato.

Altro conflitto che si può verificare, è quello legato a due istruzioni di memorizzazione di fila, come ad esempio:

```
I1 : LW R2,10
I2 : SW R2,20
I3 : SUB R2,R3,R4
```

Tale istruzione infatti non fa altro che caricare un dato dalla memoria nel registro R2, per poi ricaricarlo all'indirizzo 20.
Anche in questo caso si verifica un conflitto sui dati che viene risolto retroazionando il risultato in uscita dalla fase di WB in ingresso alla Data Memory, usando un multiplexer apposito.

Tale segnale di abilitazione viene gestito da una *unità di controllo* ausiliaria, chiamata **Data HazardControl**, puramente combinatoria.
Tale unità riceverà in ingresso i segnali di lettura e scrittura da memoria, nonchè quelli di scrittura nei registri, sia dalla fase di MEM, che da quella di WB. Sarà inoltre necessario inoltrare a tale unità il registro destinazione nel quale si andrà a scrivere, e i registri sorgente con i quali effettuare un confronto. Possiamo quindi riassumere i segnali di ingresso-uscita:

- **Rs_1** : Registro sorgente 1 prelevato dalla pipe ID_EX, utilizzato per il confronto con la destinazione dell'istruzione successiva, per risolvere conflitti tra lo stage di ID e quello di EXE

- **Rs2** : Registro sorgente 2 prelevato dalla pipe ID_EX, utilizzato per il confronto con la destinazione dell'istruzione successiva, per risolvere conflitti tra lo stage di ID e quello di EXE
- **Rd_1** : Registro destinazione prelevato dalla pipe EX_MEM, utilizzato per il confronto con le sorgenti della fase precedente, al fine di risolvere i conflitti tra le due fasi.
- **Write reg 1** : Segnale che indica la scrittura nei registri da parte dell'istruzione presente attualmente nella piep EX_MEM; se tale segnale è basso, non verranno confrontate sorgenti e destinazioni
- **MR** : Segnale di memory read, che indica la presenza di una istruzione di lettura dalla memoria
- **MR** : Segnale di memory write, che indica la presenza di una istruzione di scrittura sulla memoria.
- **Write reg 2** : Segnale che indica la scrittura nei registri da parte dell'istruzione presente attualmente nella piep MEM_WB; se tale segnale è basso, non verranno confrontate sorgente e destinazione
- **RS1_2** : Registro sorgente 1 prelevato dalla pipe EX_MEM, utilizzato per il confronto con la destinazione dell'istruzione successiva, per risolvere conflitti tra lo stage di EXE e quello di ID
- **Rd_2** : Segnale registro destinazione prelevato dalla fase di WB

I segnali di uscita prodotti da tale unità saranno quelli di abilitazione dei multiplexer relativi alla selezione del dato retroazionato, piuttosto che di quello previsto dai registri/memoria.

I conflitti appena presentati sono caratterizzati tutti dall'essere conflitti di tipo **RAW**, ovvero lettura dopo scrittura. Tale architettura infatti prevede la scrittura sui registri solo nella fase di Write Back, rendendo impossibile il verificarsi di altri tipi di conflitto come quelli del tipo WAW oppure WAR.

Bisogna a questo punto precisare che eventuali operazioni di interfacciamento con la memoria nella fase di MEM, non risulterebbero comunque

Figure 5: Schema temporale dello Stall tramite flip flop T

funzionanti. Il motivo sta nel fatto che la memoria ha bisogno di un colpo di clock per poter restituire il dato;
Sul fronte alto del clock infatti, la pipe EXE_MEM leggerà i dati provenienti dalla fase di esecuzione, e li inoltrerà nella fase di memorizzazione. Tuttavia sullo stesso fronte la memoria dovrebbe effettuare una operazione di scrittura/lettura, che non potrà ancora effettuare poichè i dati non le saranno stati ancora presentati.

Ne consegue che al colpo di clock successivo la memoria effettuerà la sua operazione, ma i dati corrispondenti a questa operazione verranno già caricati nella pipe MEM_WB, rendendo l'operazione inconsistente.

La tecnica adottata è stata quella dello **stall** della pipe, almeno per un colpo di clock, al fine di mantenere coerenza con i dati.Da un punto di vista pratico ciò è stato realizzato tramite l'ausilio di un flip flop a commutazione, il cui segnale di Toggle è un or tra i segnali di mem write e mem read.

In figura è presentato uno schema temporale che esplicita il funzionamento del meccanismo di stall.

possiamo quindi apprezzare in figura una versione revisionata del processore in grado di gestire correttamente un livello di dipendenza.

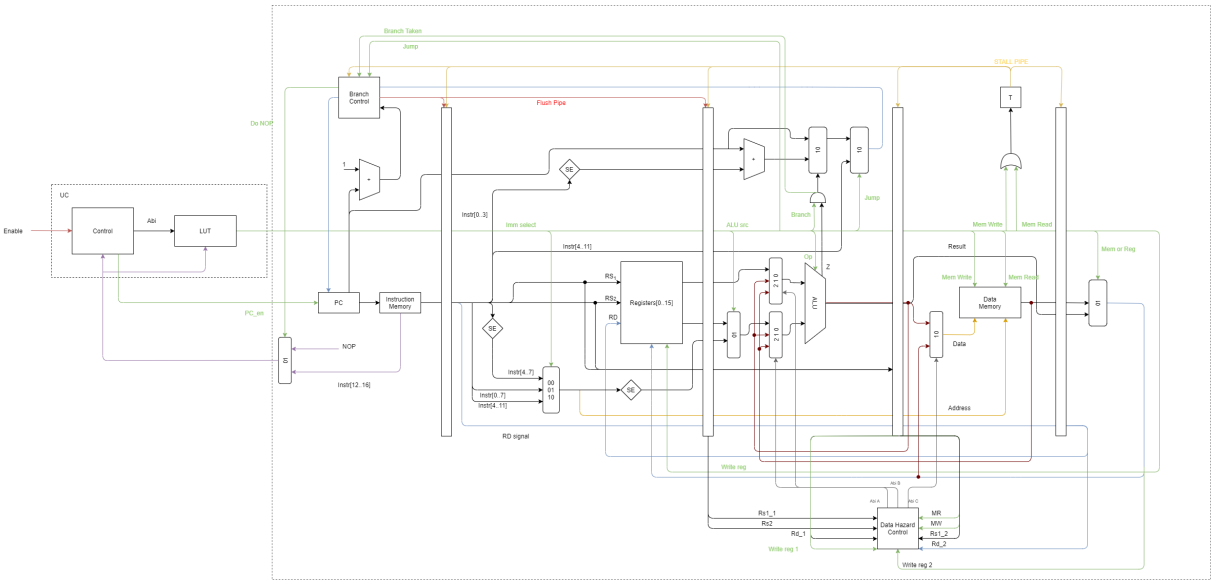

Figure 6: Datapath del processore SPP con pipeline a 5 stadi e gestione di un livello di conflitti

Nonostante le migliorie applicate, il processore non è ancora in grado di poter gestire tutti i tipi di conflitti. analizziamo il seguente codice:

```
I1 : ADD R0,R1,R2
I2 : ADD R2,R0,R3
I3 : SUB R2,R3,R4
```

Tramite l'architettura fino ad ora proposta il conflitto tra l'istruzione I2 e quella I1 relativa al registro R2, e quello tra l'istruzione I3 e l'istruzione I2 relativo al registro R3, è correttamente gestito.
Tuttavia possiamo notare come anche l'istruzione I3 dipenda da I1 a causa del registro R2, che ancora non sarà stato correttamente riscritto nella fase di WB. è allora necessario estendere la tecnica di forwarding così come la abbiamo presentata al momento.

Come possiamo immaginare sarà necessario implementare ulteriori multiplexer, e segnali di retroazione dalla fase di WB verso la fase di esecuzione. Qualora infatti la destinazione della istruzione in fase di WB (I1) coincidesse con una delle sorgenti dell'istruzione in fase di Execute (I3), sarebbe necessario retroazionare il risultato in uscita dalla fase di WB verso la fase di EXE.

ciò significa allora che i segnali che venivano retroazionati dalla fase di MEM verso quelli di EXE, dovranno essere ulteriormente filtrati da altri 4 multiplexer che permettano, operando per operando, di poter selezionare il dato retroazionato dalla fase di MEM (sia esso il dato di tipo "ALU" o quello di tipo "memoria") oppure quello retroazionato dalla fase di WB.

il Data Hazard Control sarà quindi responsabile della produzione di 4 nuovi segnali di abilitazione, che permettano la selezione corretta tra i possibili segnali in feedback.

Con questa soluzione giungiamo quindi ad un architettura in grado di gestire **2 livelli di dipendenza**.

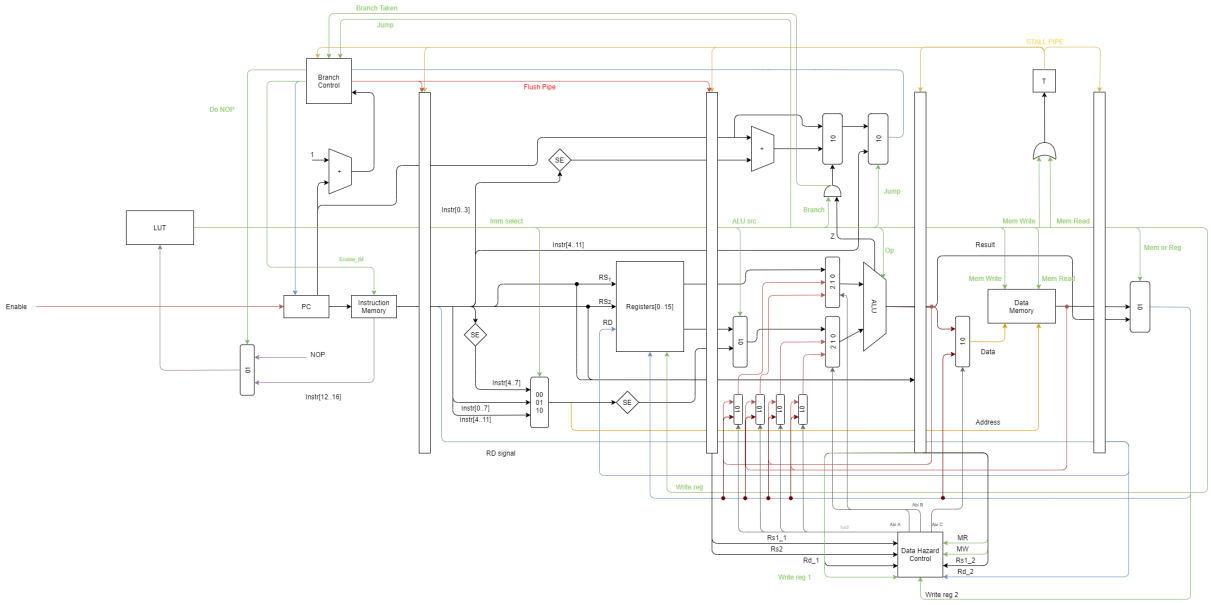

Figure 7: Datapath del processore SPP con pipeline a 5 stadi e gestione di due livelli di conflitti

A seguito di una analisi più approfondita si può però notare come in realtà anche due livelli di dipendenza non siano sufficienti a coprire tutti i possibili conflitti sui dati.
Consideriamo il seguente codice:

```
I1 : SETR R0,20
I2 : ADD R0,R1,R3
I3 : ADDi R3,02,R4
I4 : SUB R0,R4,R2
```

Le istruzioni I1,I2 ed I3, per quanto detto prima, nonostante abbiano dipendenza tra di loro, riescono a funzionare correttamente. Tuttavia l'istruzione 4 fa uso di un registro che viene scritto dall'istruzione I1, che tuttavia dovrebbe aver già completato la sua esecuzione. Nonostante ciò sia vero, bisogna considerare il fatto che la struttura Registers legge e scrive sul fronte di discesa; Ne consegue che mentre l'istruzione I4 sarà nella fase di ID e leggerà il registro R0, esso verrà contemporaneamente scritto dalla istruzione I1, ottenendo un comportamento tecnicamente impredicibile.

Questo ci costringe dunque ad inoltrare i registri sorgente della fase di ID alla Data Hazard Control unit, in maniera tale da poterli confrontare con il registro destinazione specificato dalla istruzione in fase di WB. Se si ottiene una coincidenza, l'unità piloterà due nuovi multiplexer, apposti alla struttura register nella fase di ID, che permettano la selezione del dato retroazionato in ingresso alla struttura *prima* che questo venga scritto nei registri. In questa maniera è garantito che il valore presentato alla pipe ID_EXE sarà coerentemente quanto scritto da I1 in R0, e contemporaneamente nel registro R0 verrà scritto il suo nuovo valore corretto, disponibile tranquillamente dalla prossima istruzione.

Ciò realizza, di fatto, una gestione dei **3 livelli** di dipendenza possibili in questa architettura, rendendola robusta ai conflitti sui dati.

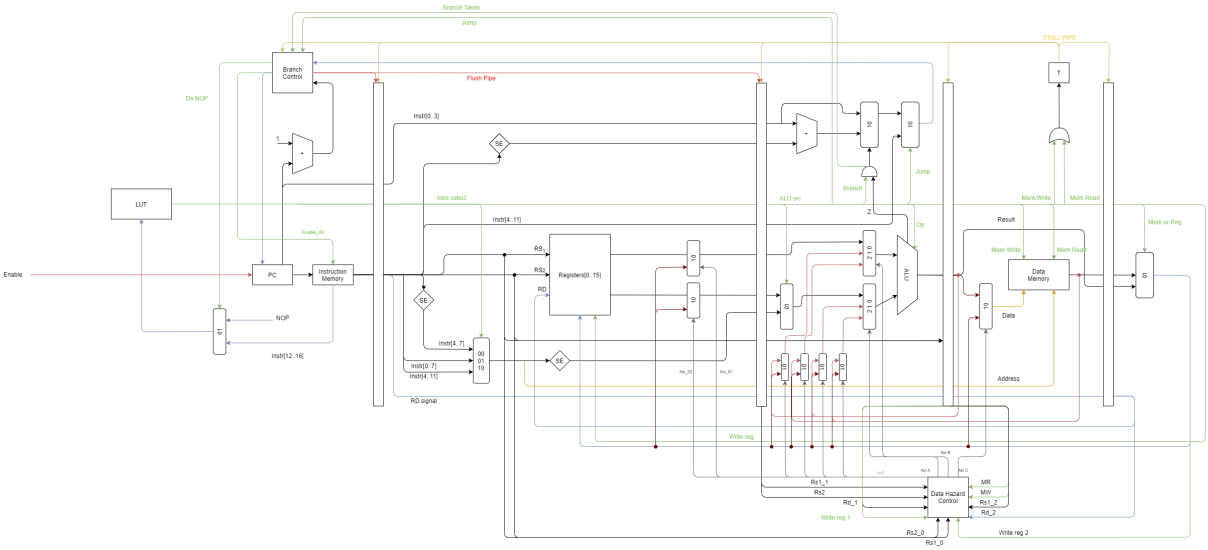

Figure 8: Datapath del processore SPP con pipeline a 5 stadi e gestione di tre livelli di conflitti

3 Implementazione

Per quanto ne concerne l'implementazione del processore SPP, esso è stato interamente realizzato in VHDL secondo una combinazione di approccio *strutturale* e *comportamentale*. Successivamente la simulazione è avvenuta per mezzo del compilatore GHDL e del generatore di onde GTKwave;
è inoltre stato realizzato un semplice assemblatore in C++, che assemblasse il codice ad "alto livello" in un codice esadecimale interpretabile dal processore, e registrato nella IM.

Citiamo brevemente i componenti utilizzati per questa implementazione, senza entrare nel merito del codice

- **Multiplexers** : I multiplexer sono stati realizzati in 5 versioni differenti; uno con 2 ingressi da 4 bit e una uscita, uno con due ingressi da 8 bit e una uscita, uno con due ingressi da 16 bit e una uscita, e altri due con 3 ingressi rispettivamente da 8 e 16 bit, con una uscita ciascuno.
- **SE** : Sign extenders da 4 ad 8 bit e da 8 a 16 bit
- **ALU** : componente combinatoria che prevede segnali di abilitazione, il flag Z in uscita, un risultato in uscita su 16 bit e due operandi in ingresso da 16 bit
- **LUT** : Control Store dell'architettura, con 16 parole da 12 bit ciascuna.
- **PC** : Program counter, registro abilitato sul fronte di discesa
- **Registers** : struttura di 16 registri abilitati sul fronte di discesa e selezionabili con 8 bit in ingresso; un nibble per il primo registro sorgente, ed un nibble per il secondo. Altri 4 bit in ingresso specificano il registro destinazione, e 16 bit di ingresso sono dedicati invece al dato da scrivere. La struttura viene abilitata alla scrittura dall'apposito segnale di controllo; infine in uscita vengono inoltrati 16 bit per il dato della prima sorgente, e 16 per il dato della seconda sorgente.
- **Instruction Memory** : Memoria per le istruzioni, di sola lettura. Abilitata sul fronte di salita, prevede uno spazio di indirizzamento da 0x00 a 0xFF, con ogni parola di dimensione 16 bit, comprensiva di codice operativo ed operandi.

- **Data Memory** Memoria per i dati, sia di lettura che scrittura. Ha un comportamento molto simile all'instruction memory, ma prevede invece due segnali di abilitazione per segnalare scrittura o lettura da tale memoria.

- **Pipes** : registri abilitati sul fronte di salita, e differenziati stage per stage. Hanno una struttura piuttosto regolare in quanto presentano ingressi che vengono direttamente presentati in uscita sul fronte alto del clock.

- **Flip Flop T** : flip flop utilizzato per gestire lo stall dovuto dall'attesa dalla memoria.

- **Stages** : Raggruppamento strutturale di tutti i componenti precedentemente definiti, al fine di separare ogni singola fase esecutiva per testarla separatamente, e successivamente integrarla.

- **Data Hazard Control** : unità esterna ai vari stages per la gestione dei conflitti sui dati, già ampiamente descritti in precedenza. essendo puramente combinatoria non necessità di clock

- **Branch Control Unit** : Unità di controllo per i salti, incapsulata all'interno della fase di Fetch; è responsabile del flush della pipe per la gestione dei salti, e dell'inoltro del prossimo indirizzo al PC. Essendo predisposta a fare solo queste operazioni, è stata implementata anch'essa in maniera puramente combinatoria

- **SPP System** : Integrazione completa delle varie fasi di elaborazione Fetch, Decode, Execute, Memory, Write Back e della Data Hazard Control unit.

L'approccio utilizzato nello sviluppo è stato del tipo *divide et impera* : per ogni componente se ne è realizzato un testbench e lo si è poi integrato con gli altri componenti, per poi rieffettuare un testbench successivo. Arrivati al

completamento di un'intero Stage, esso veniva testato opportunamente, per poi alla fine interconnettere tutti gli stages e realizzarei l sistema complessivo.

A titolo d'esempio, mostriamo un programma in esecuzione di questo tipo:

```
I1 : SETR R1,20
I2 : SETR R2,30
I3 : ADD R1,R2,R3
I4 : SW R3,20
I5 : LW R4,20
I6 : ADD R4,R3,R5
```

come possiamo notare abbiamo in questo codice tutti e tre i livelli di dipendenza, sia tra istruzioni di tipo registro, che tra istruzioni di interfacciamento della memoria.

Analizzando la figura relativa alle forme d'onda, iniziamo stabilendo l'associazioni tra fasi e colori:

- Colore verde - Clock e Fase di Fetch
- Colore arancione - Fase di Decode
- Colore blu - Fase di Esecuzione
- Colore giallo - Fase di Memorizzazione
- Colore viola - Fase di Write Back

Le forme d'onda quindi mostrano, ad ogni colpo di clock, i valori di determinati segnali per la data istruzione; possiamo notare ad esempio i diversi segnali di controllo, il valore del program counter, dell'immediato, dei registri sorgente e quelli destinazione e via dicendo.
Ci interessa però dissezionare il codice appena mostrato, e dunque iniziamo andando a tradurre ogni singola istruzione nel formato esadecimale corrispondente:

```
I1 : SETR R1,20 := 9201
I2 : SETR R2,30 := 9302
```

```
I3 : ADD R1,R2,R3 := 1123
I4 : SW R3,20 := A320
I5 : LW R4,20 := B204
I6 : ADD R4,R3,R5 := 1435
```

ad ogni colpo di clock una istruzione transita da una fase ad un altra, ragion per cui dato il ciclo di clock k, l'istruzione presente nella fase di Fetch (associata al colore verde) si troverà nella fase di Decode (colore arancione) solo al ciclo di clock k+1, e così via per ogni altra fase/istruzione.
Ciò significa che l'istruzione 9201, SETR R1,20, si troverà al colpo di clock k nella fase di fetch, k+1 in quella di decode, k+2 in quella di execute, k+3 in quella di mem e k+4 in quella di wb.
Seguendo questo ragionamento tutte le altre istruzioni risulteranno così sfasata.

Possiamo infatti notare come il valore 0x0020 in uscita dalla istruzione 9201 in WB_data_out sia visibile solo dopo 4 colpi di clock.
Dunque il risultato è tracciabile proprio da questo segnale: al colpo di clock successivo al k+4 ci sarà il risultato dell'istruzione I2, 0x0030, e successivamente il risultato 0x0050 dato dalla istruzione I3, che quindi ha correttamente gestito il primo e il secondo livello di dipendenza.

Possiamo però notare che l'istruzione I3 sembra durare molto più del dovuto. ciò è dovuto infatti allo stall delle istruzioni I4 e I5, che aumentano la "permanenza" dello stesso dato per altri 2 colpi di clock.

Alla fine, al colpo di clock k+9 verrà completata l'istruzione I6 che leggerà correttamente 0x00A0 = 100_{dec}, così come ci aspettavamo.

Per concludere possiamo presentare anche un banale esempio di gestione di conflitti sui salti, esplicabile tramite il seguente codice:

```
I1 : SETR R1,A0 := 9A01
I2 : SETR R2,05 := 9052
I3 : JAM 05 := C050
I4 : SETR R2,02 := 9022
I5 : ADD R1,R2,R3 := 1123
```

Possiamo notare in questo caso come quando l'istruzione di JAM entri

```
********** SPP Assembler ***********

Scrivi programma:

01 : SETR R1,20
02 : SETR R2,30
03 : ADD R1,R2,R3
04 : SW R3,20
05 : LW R4,20
06 : ADD R4,R3,R5
07 : HALT
Esecuzione programma......
```

Figure 9: Assemblatore dell'SPP

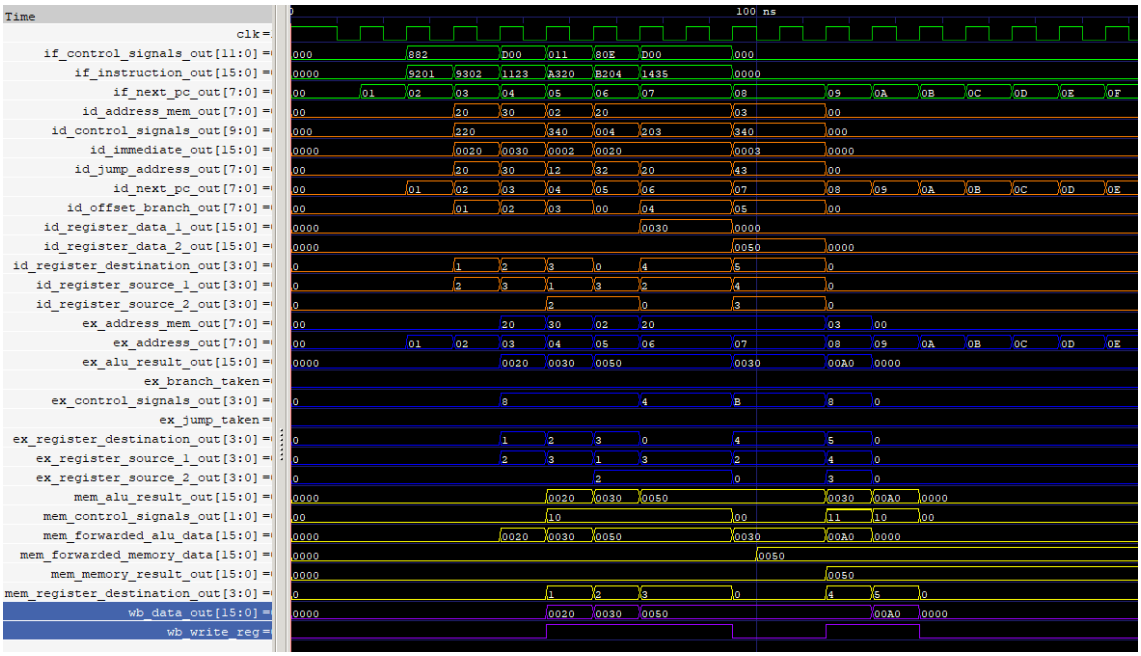

Figure 10: Esecuzione di un programma di somma con conflitti

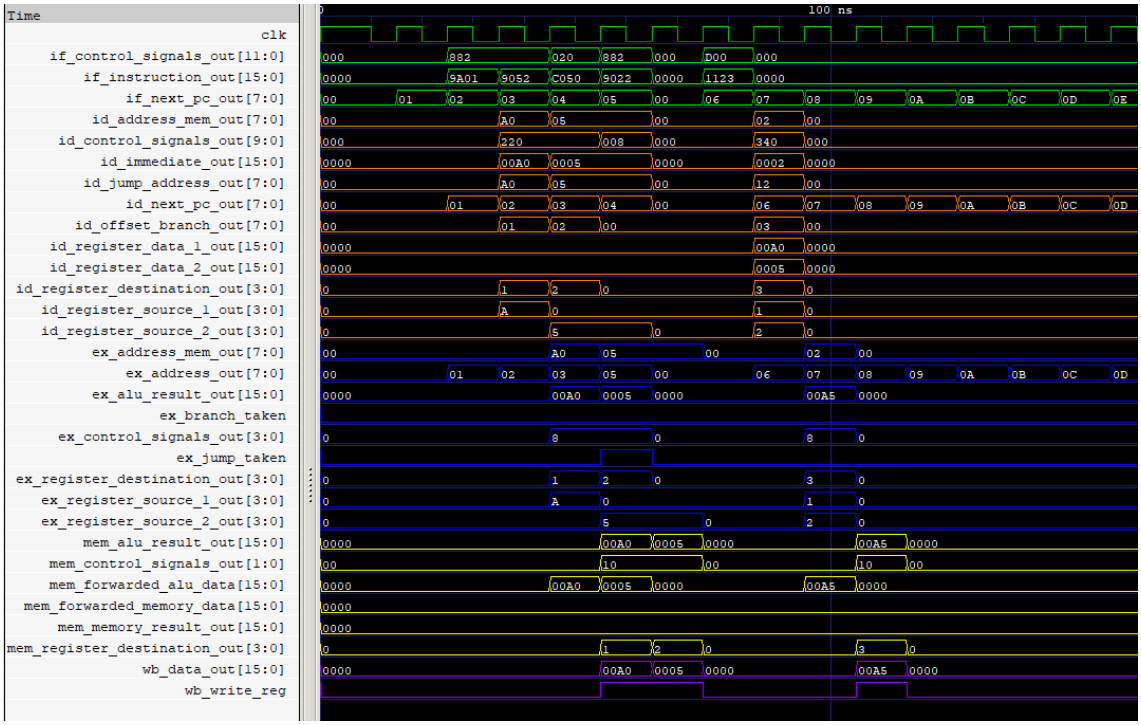

Figure 11: Esecuzione di un programma con salto

nella fase di esecuzione venga effettuato il flush delle pipes. se così non avvenisse, il risultato di questa istruzione dovrebbe essere 0x00A2, mentre invece otteniamo correttamente il risultato 0x00A5.
è tuttavia chiaro che questa tecnica risulta essere piuttosto inefficiente in quanto l'intero programma impiegherà 2 colpi di clock in più per poter terminare, rispetto ad una tecnica con previsione corretta.