Contents

Chapter 1

Introduzione

La tecnica implementativa basata sull'uso di pipeline prevede la sovrapposizione temporale dell'esecuzione di diverse istruzioni; al giorno d'oggi le pipeline sono la chiave su cui si basa la velocità dei calcolatori. Il concetto è analogo a quello di una catena di montaggio in cui un prodotto incontra diversi stadi di semilavorazione, in ognuno dei quali viene eseguita un'operazione ben determinata, per uscire poi dalla catena completamente lavorato; mentre nel frattempo, altri pezzi sono in fase di lavorazione.

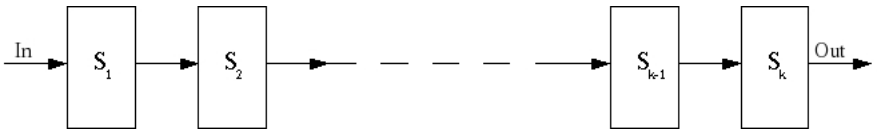

fig. 1.1

Come si evince dalla fig. 1.1, l'esecuzione di una istruzione da parte del processore è divisa in k stadi ordinati temporalmente, al generico istante t, uno stadio preleva l'istruzione da quello precedente, la elabora e la invia in ingresso allo stadio successivo.

Sebbene una pipeline lineare possa essere composta da una successione di stadi completamente asincroni, non è possibile utilizzarla all'interno dei

calcolatori. Ciò comporta una modifica alla struttura generale di fig. 1.1, considerando l'aggiunta fra due stadi successivi di latch register regolati da un clock comune come mostrato in fig. 1.2.

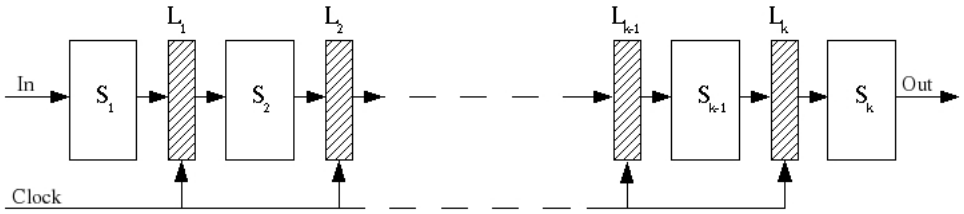

fig. 1.2

Durante il periodo di clock, ogni stadio elabora il contenuto del latch register precedente, l'output prodotto viene memorizzato sul latch register successivo sul fronte in salita del clock.

Considerando quindi una pipeline a k stadi, a regime, ad ogni clock questa contiene k istruzioni, ognuna in una diversa fase di elaborazione.

Chapter 2

Prestazioni

Per poter effettivamente considerare il notevole vantaggio introdotto dall'architettura a pipeline, è possibile fare un confronto di questa soluzione inplementativa con quelle monociclo e multiciclo.

Definiamo pertanto:

Monociclo: architettura nella quale il tempo di esecuzione di una istruzione è dato dalla somma dei tempi dei singoli stadi.

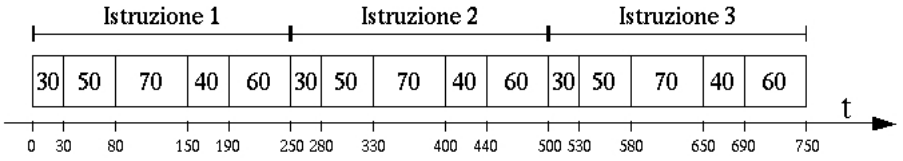

fig. 2.1

Multiciclo: architettura nella quale il tempo di esecuzione di una istruzione è dato sempre dalla somma dei tempi dei singoli stadi, con la particolarità che questi sono tutti uniformati al tempo di esecuzione dello stadio più lento, quindi in realtà il tempo totale di esecuzione è dato dal prodotto del tempo dello stadio più lento

per il numero degli stadi.

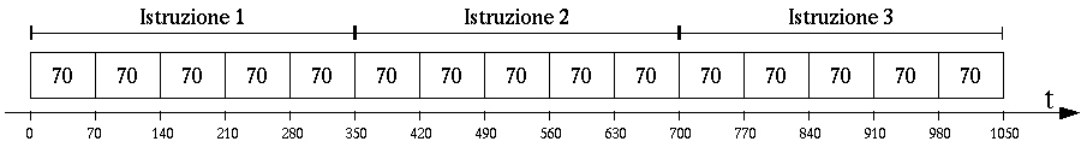

fig. 2.2

Considerando, invece, l'architettura a pipeline notiamo come il tempo totale di elaborazione di un'istruzione è dato sempre dalla somma dei tempi di esecuzione dei singoli stadi uniformati a quello più lento (multiciclo) con in aggiunta il tempo relativo ai latch register inseriti fra uno stadio e l'altro. Questo permette a ciascuna fase di poter funzionare in parallelo con le altre.

Se pertanto definiamo:

t_i = tempo di risposta del generico stadio

t_{pd} = propagation delay dei latch register

T_{mono} = $\sum_{i=1}^{k} t_i$ - tempo di esecuzione di una istruzione soluzione monociclo

t_{multi} = $\max \{t_i\}_{i=1,2,...,k}$ - tempo di esecuzione singolo stadio soluzione multiciclo

T_{multi} = kt_{multi} - tempo di esecuzione di una istruzione soluzione multiciclo

t_p = $\max \{t_i\}_{i=1,2,...,k} + t_{pd}$ - tempo di esecuzione singolo stadio soluzione pipeline

T_p = $kt_p + kt_{pd}$ - tempo di esecuzione di una istruzione soluzione pipeline

k = numero di stadi della pipeline

Si nota, quindi, come il tempo totale di elaborazione di un'istruzione nella struttura in pipeline è pari a $kt_p + kt_{pd}$ contro i Kt_{multi} del multiciclo; malgrado il ritardo introdotto dai latch register l'architettura a pipeline introduce un miglioramento complessivo delle prestazioni dato che a regime elabora un'istruzione ogni T_{multi}.

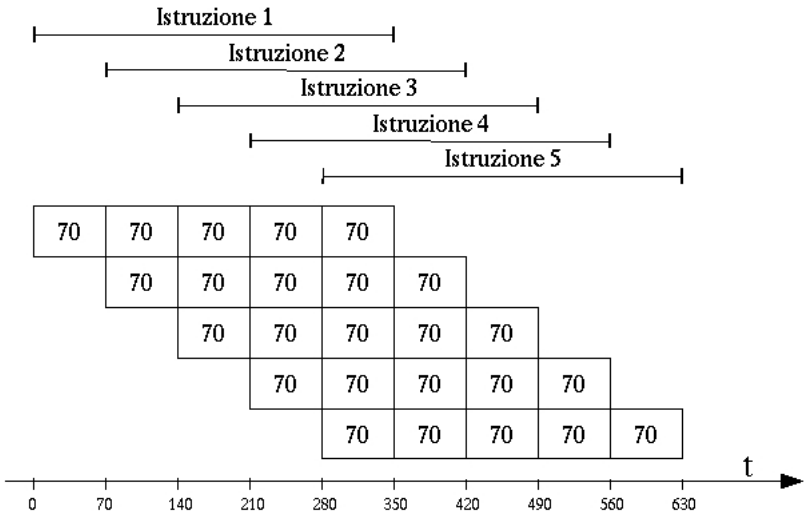

fig. 2.3

Per quanto riguarda il throughput avremo $T_k = T_p/k$

Chapter 3

Formato Istruzioni

Caratteristiche principali:

- set di istruzioni limitato
- codificate su un numero di bit stabilito e uguale per ogni istruzione in modo tale da eliminare la fase di decodifica
- le istruzioni devono lavorare su registri così da evitare accessi in memoria

Possiamo inoltre considerare istruzioni di lettura/scrittura in memoria non solo di parole ma anche di semiparole e byte nonchè istruzioni di salto.

Avremo pertanto un set di istruzioni così diviso:

Istruzioni aritmetiche fra registri: istruzioni per operare sui dati

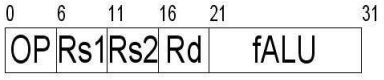

fig. 3.1

OP: contiene il codice del tipo di istruzione

Rs1,Rs2: identificano i registri sorgente

Rsd: identifica il registro destinatario del risultato

fALU: identifica la specifica operazione aritmetica (ADD,SUB,OR,MUL,...)

Esempio:

ADD R1,R2,R3; R1<=R2+R3;

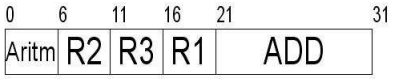

fig. 3.2

Istruzioni di lettura e scrittura di dati in memoria

LD LOAD
lettura dalla memoria di un operando e memorizzazione in un registro

ST STORE
scrittura in memoria di un operando contenuto in un registro

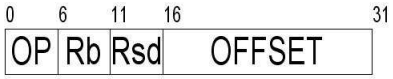

fig. 3.3

Rb: numero d'ordine del registro base

OFFSET: scostamento diviso per 4

Esempio:

LD R13,100(R6) R13<=M[R6+100*4]

ST 100(R6),R13 R13=>M[R6+100*4]

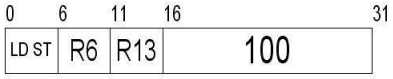

fig. 3.4

Notiamo come possiamo trovare le seguenti varianti relative alla lunghezza dell'operando da trasferire:

LB SB lettura/scrittura di un byte

LH SH lettura/scrittura di una semiparola

LW SW lettura/scrittura di una parola

Il campo offset deve contenere lo scostamento effettivo e non il suo rapporto a quattro dato che non si trasferiscono sempre parole, pertanto le semiparole devono essere allineate su indirizzi pari mentre le parole su indirizzi multipli di quattro. E' ovvio che l'architettura presenterà un po' di logica addizionale basata sul codice dell'istruzione e sui 2 bit meno significativi dell'indirizzo per poter trasferire i tre tipi di dati. In seguito per l'analisi in pipeline di questo tipo di istruzioni ci riferiremo al loro formato più generale LD/ST.

Istruzioni di salto

Consideriamo le normali istruzioni, con l'aggiunta di altre più particolari:

JE-JS salto condizionato

JE (Salta se uguale)
JS (Salta in base al segno)

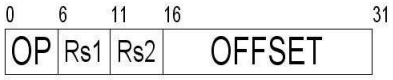

fig. 3.5

Rs1,Rs2: registri il cui contenuto deve essere confrontato

OFFSET: scostamento diviso per 4

JE effettua il salto se il contenuto dei due registri è uguale
JS effettua il salto se il contenuto del secondo registro è maggiore del primo

Esempio:

JE R1,R2,100 if(R1==R2) PC<=PC+100*4;

JS R1,R2,100 if(R1 < R2) PC<=PC+100*4;

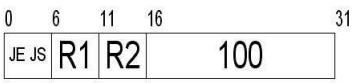

fig. 3.6

JMP salto incondizionato

fig. 3.7

IND: indirizzo della posizione di destinazione del salto (sempre positivo) diviso per 4

Esempio:

JMP etichetta; PC<=IND*4[1]

fig. 3.8

JAL (Jump And Link)

Istruzione che viene utilizzata per la chiamata a sottoprogrammi. Consiste in un salto incondizionato all'indirizzo specificato come parametro (Jump), ma a differenza della JMP salva in R31[2] l'indirizzo dell'istruzione successiva (Link).

[1]"etichetta" è un nome simbolico che identifica la posizione in memoria della destinazione del salto, IND ne rappresenta l'indirizzo reale in parole di 4 bytes

[2]R31 (Link Register) è un registro utilizzato solo per questa funzione, evita di utilizzare lo stack memory e quindi di fare accessi in memoria

fig. 3.9

IND: indirizzo della posizione di destinazione del salto diviso per 4

Esempio:

JAL etichetta; R31<=PC+4; PC<=IND*4;

fig. 3.10

JR (Jump Return)

provoca un salto incondizionato all'indirizzo contenuto nel registro specificato, viene utilizzato come ritorno da un sottoprogramma se utilizziamo R31.

fig. 3.11

Rs1: registro nel quale è memorizzato l'indirizzo di ritorno

Esempio:

JR R31; PC<=R31;

fig. 3.12

NOP (Istruzione di non operazione)

non modifica nulla, normalmente codificata con tutti zeri, l'unico effetto è quello di far aumentare di quattro il contenuto di PC.

NOP; PC<=PC+4;

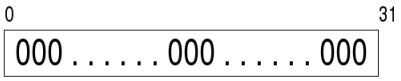

fig. 3.13

Chapter 4

Struttura di una Pipeline Logica

Consideriamo una pipeline a 5 stadi come modello di riferimento simile a quella riportata in fig. 4.1:

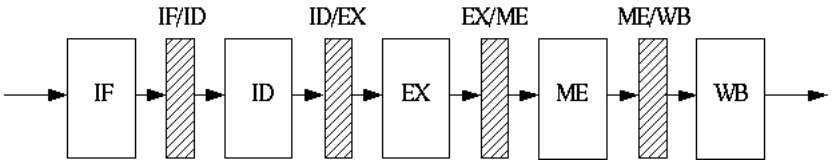

fig. 4.1

IF (Instruction Fetch) stadio nel quale viene letta l'istruzione il cui indirizzo è contenuto nel PC. Questa poi viene memorizzata nel registro IF/ID.

ID (Instruction Decode) in questo stadio viene codificato il codice di operazione prelevato da IF/ID e viene memorizzato in ID/EX. Questo è uno stadio importante perchè qui vengono determinate le azioni delle istruzioni negli stadi successivi ed in particolare nello stadio EX.

EX (Execute) qui vengono compiute le azioni specifiche di ogni istruzione. Lo stadio è costituito da una ALU che prende operandi e comandi dal registro ID/EX e porta in uscita il risultato sul registro EX/ME. In questo stadio vengono anche propagati eventuali comandi per le fasi successive.

ME (Memorize) viene utilizzato solo per istruzioni che accedono in memoria (LD e ST). Nel caso di lettura dalla memoria il dato viene prelevato e memorizzato in ME/WB. Per altri tipi di istruzioni questo è da considerarsi solo uno stadio di transito, pertanto si ha solo una propagazione dei contenuti del registro EX/ME in ME/WB.

WB (Write Back) in quest'ultimo stadio vengono completate le istruzioni di caricamento e aritmetiche.

Analizziamo più in dettaglio i moduli presenti nei vari stadi:

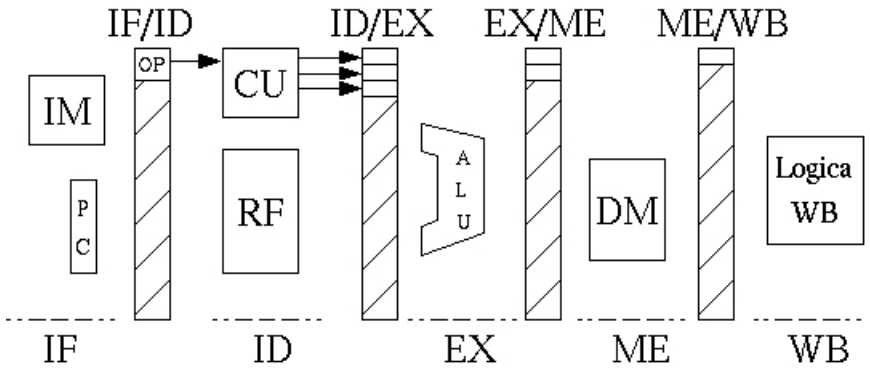

fig. 4.2

IM (Instruction Memory) è la memoria dalla quale viene prelevata l'istruzione il cui indirizzo e in PC. Questa viene memorizzata in IF/ID.

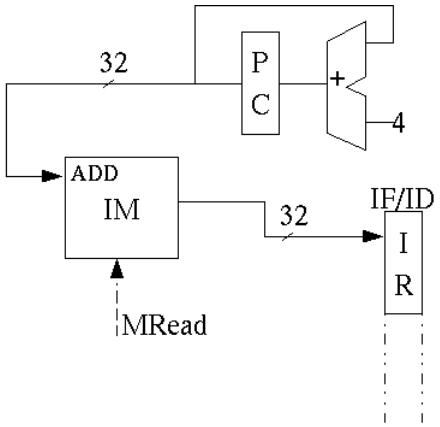

fig. 4.3

CU (Control Unit) è l'unità di controllo, comprende tutta una logica per la decodifica del codice operativo dell'istruzione, l'output viene memorizzato nei campi WB,ME,EX di ID/EX[1]. Questi non sono altro che i segnali di abilitazione e comando dei corrispondenti stadi.

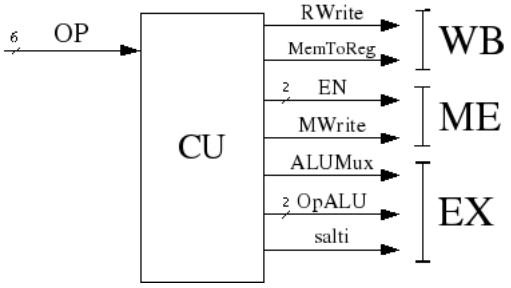

fig. 4.4

[1]In base al tipo di istruzione alcuni di questi campi possono rimanere inutilizzati

RF (Register File) sono una serie di registri del processore che vengono utilizzati come sorgenti e destinatari di operazioni e salti, vengono utilizzati per velocizzare il tempo di esecuzione della fase evitando accessi in memoria.

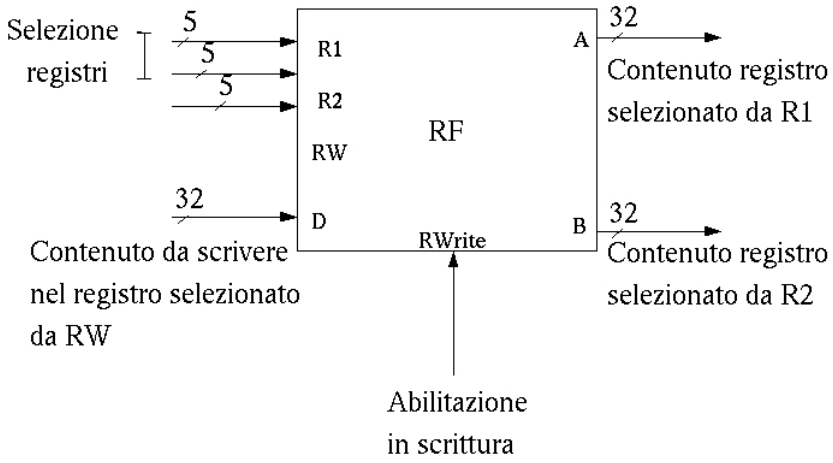

fig. 4.5

ALU notiamo dalla fig. 4.6a il suo controllo da parte di EX (OpALU1,OpALU0). Questi segnali specificano alla ALU il tipo di operazione da svolgere secondo la tabella in fig. 4.6b

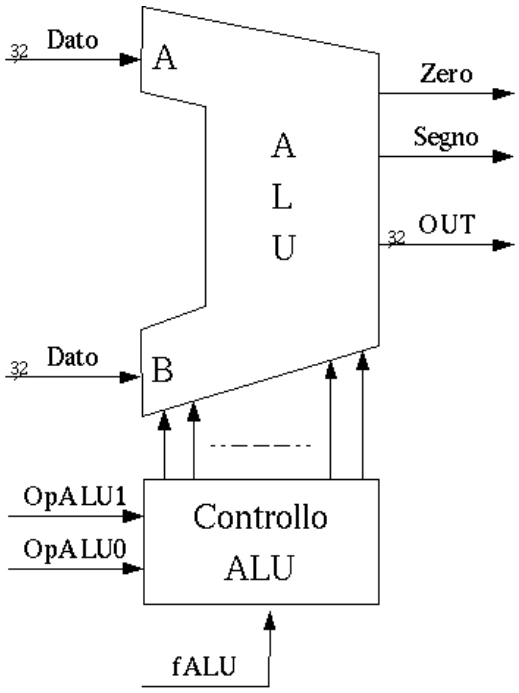

fig. 4.6a

AluOUT	OpALU1	OpALU0
B	0	0
A + B	0	1
A – B	1	0
A fALU B	1	1

fig. 4.6b

DM (Data Memory) è la memoria vera e propria. Questa viene utilizzata dalla LD e ST per il prelievo o la memorizzazione degli operandi.

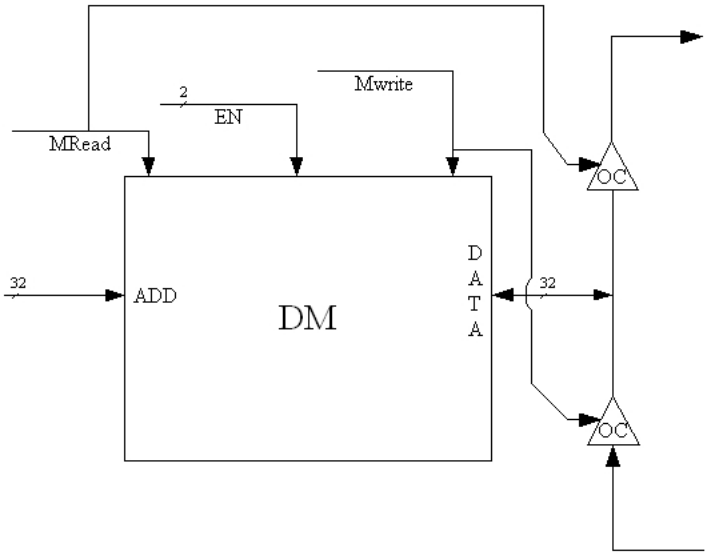

fig. 4.7

Chapter 5

Esempi di Pipeline Logica

5.1 Istruzioni aritmetiche e logiche

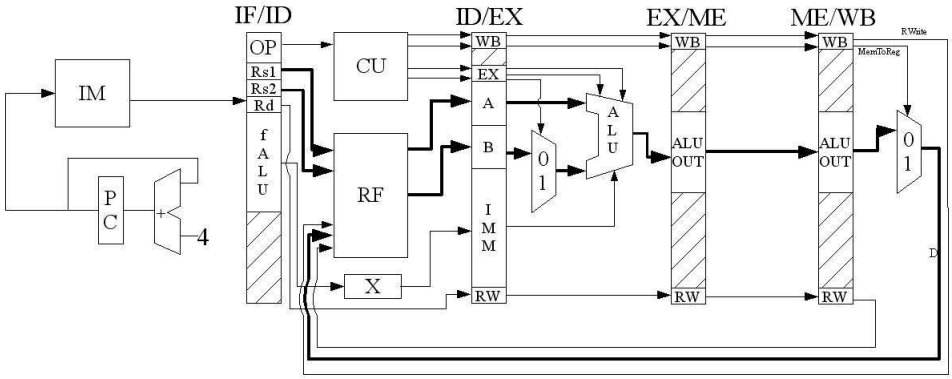

fig. 5.1

Fase IF

Viene letta l'istruzione il cui contenuto è in PC e memorizzata nel registro IF/ID:

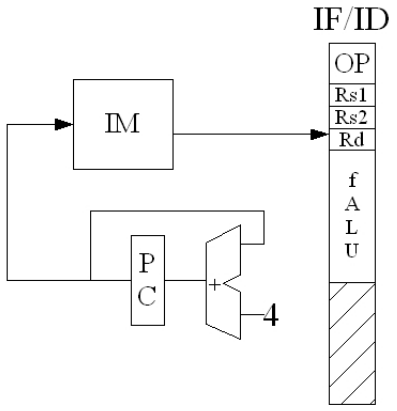

fig. 5.1.1

Fase ID

In questo stadio l'unità di controllo, sulla base del contenuto di OP genera i segnali per gli stadi successivi, in particolare per EX e per WB. Non vengono generati per lo stadio ME perchè non richiesto dato che le istruzioni aritmetiche non necessitano di accessi in memoria perchè lavorano esclusivamente sui registri, i cui contenuti sono prelevati da RF.

I segnali relativi a WB sono:

RWrite=1 abilita la scrittura nel RF del risultato dell'operazione

MemToReg=0 seleziona come output dello stadio WB il risultato della ALU

I segnali relativi a EX sono:

AluMUX=0 seleziona il campo B del registro ID/EX come ingresso B della ALU

OpALU=11 specifica alla ALU il tipo di operazione da svolgere secondo la tabella in fig. 4.6b

Considerando il RF, questo prende in ingresso gli indirizzi dei registri sorgenti che contengono gli operandi e manda in uscita il loro contenuto sui campi A e B di ID/EX.
Il valore di Rd viene propagato sul campo RW di ID/EX mentre fALU viene esteso su 32bit dal blocco X, potrebbe benissimo mantenersi su 11bit, ma per compatibilità con gli altri tipi di istruzioni viene esteso, ciò però è del tutto irrilevante ai fini della corretta esecuzione dell'istruzione.

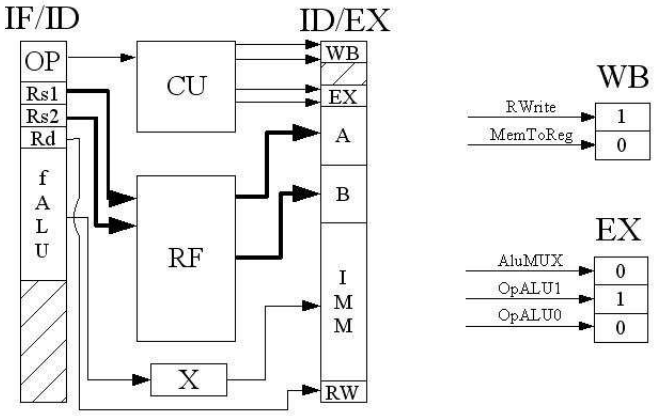

fig. 5.1.2

Fase EX

In questo stadio la ALU sulla base dei valori di OpALU e IMM esegue una particolare operazione sugli operandi A e B del registro ID/EX.
Sul registro AluOUT viene memorizzato il risultato dell'operazione, mentre vengono propagati i campi WB e RW.

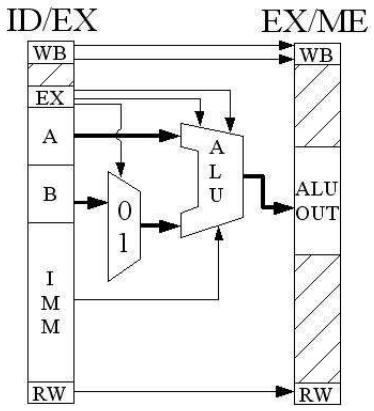

fig. 5.1.3

Fase ME

Lavorando esclusivamente sui registri, le istruzioni aritmetiche non prevedono accessi in memoria, ragion per cui i campi relativi a EX/ME vengono propagati su ME/WB.

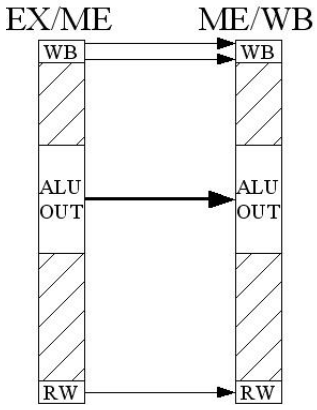

fig. 5.1.4

Fase WB

Qui si conclude l'esecuzione dell'istruzione. MemToReg abilita l'uscita del registro AluOUT. RWrite,WB e lo stesso AluOUT vengono propagati verso

RF per memorizzare il risultato dell'operazione. Questi identificano rispettivamente l'abilitazione alla scrittura in RF, l'indirizzo del registro di destinazione e il suo contenuto.

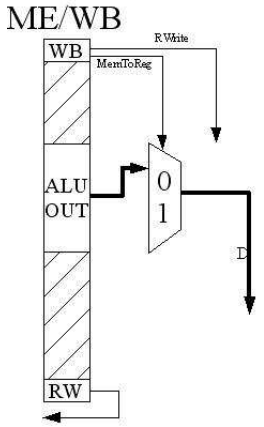

fig. 5.1.5

5.2 Load/Store

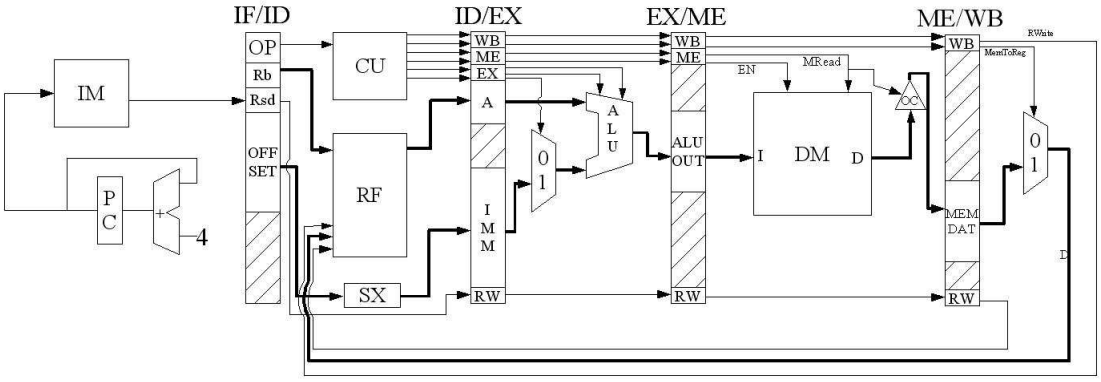

fig. 5.2a

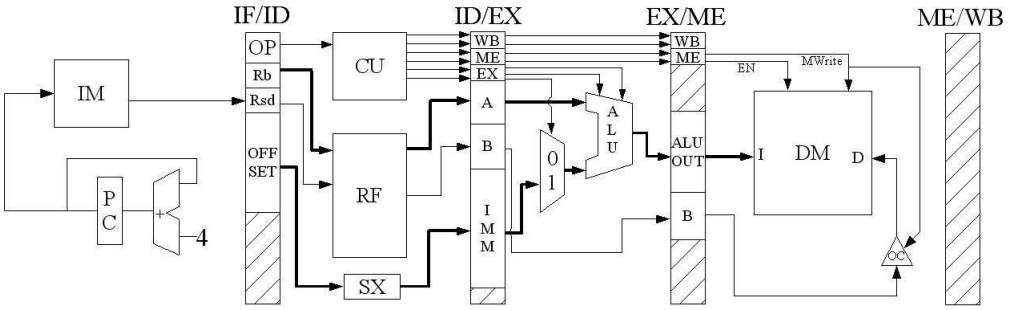

fig. 5.2b

Fase IF

Del tutto simile alle istruzioni aritmetiche, in questo caso i campi interessati, oltre OP, sono Rb che contiene il numero d'ordine del registro base, Rsd che per la LD contiene il numero d'ordine del registro destinatario dell'operazione mentre per la ST contiene il numero d'ordine del registro il cui contenuto dovrà essere scritto in memoria. Il campo OFFSET contiene lo scostamento da applicare al contenuto del registro base per ottenere l'indirizzo effettivo.

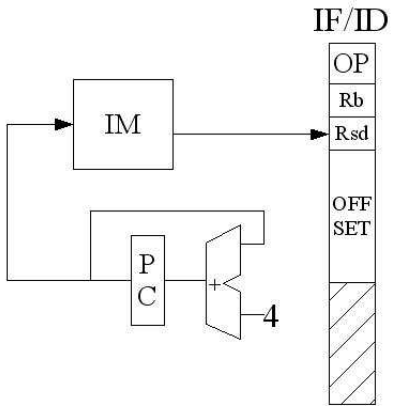

fig. 5.2.1

Fase ID

Per quanto riguarda la CU, questa genera i segnali di comando per tutti e tre gli stadi successivi ,EX,ME,WB:

OpALU=01 Operazione di somma

AluMUX=1 Seleziona IMM come ingresso B della ALU

EN=11 [ENH,ENW] rispettivamente per la LH,SH, e per la LW,SW; negli altri casi vale 00. Serve per abilitare il trasferimento di dati di lunghezza differente.

Per la LD:

RWrite=1 Scrittura in RF

MemToReg=1 Abilito come uscita dello stadio WB il campo MemDat (fig. 5.2.5)

MRead=1 Abilita DM in lettura

Per la ST:

MWrite=1 Abilita DM in scrittura

Rb determina la selezione nel RF del registro che contiene il dato da trasferire. OFFSET viene esteso tramite SX in modulo e segno su 32bit e viene memorizzato su IMM.

Nel caso di una LD, Rds viene propagato nel campo RW di ID/EX mentre per la ST costituisce un'entrata del RF per la selezione del registro che contiene il dato da memorizzare.
Le uscite del RF vengono memorizzate nei campi A(Rb) e B(Rsd) di ID/EX.

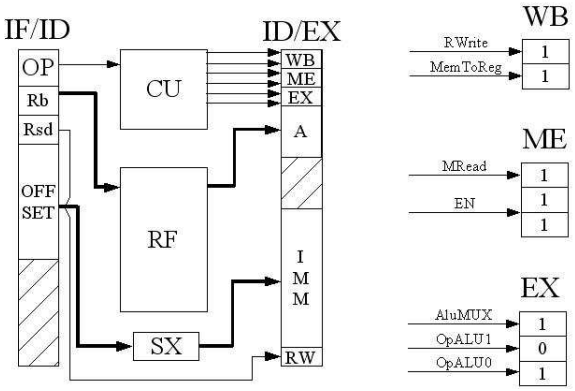

fig. 5.2.2a

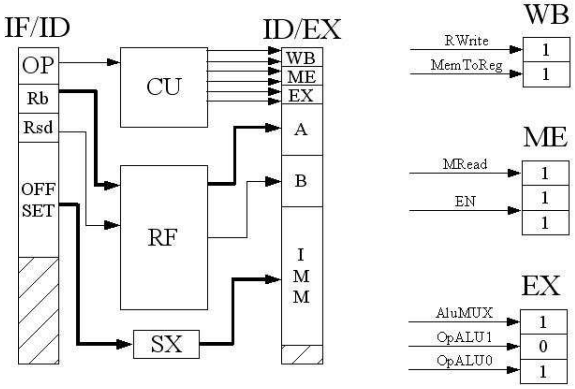

fig. 5.2.2b

Fase EX

Per entrambe le istruzioni WB e ME vengono propagati nella fase successiva, per la LD anche RW, mentre per la ST, B.

AluMUX seleziona come ingresso B della ALU, IMM; la ALU sulla base di OpALU effettua l'operazione richiesta; il risultato viene memorizzato su AluOUT di EX/ME.

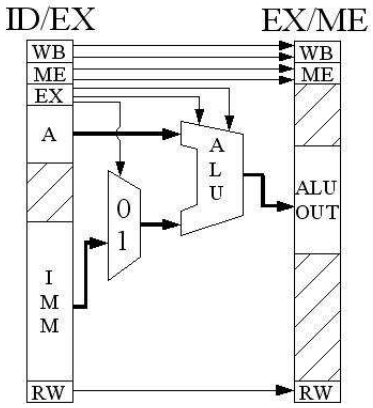

fig. 5.2.3a

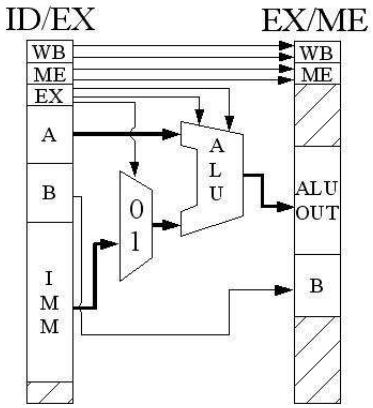

fig. 5.2.3b

Fase ME

Per la ST, qui termina la sua esecuzione: AluOUT indirizza una particolare locazione di memoria all'interno di DM, B costituisce il dato da memorizzare, MWrite abilita la memoria in scrittura, EN definisce la lunghezza del dato da trasferire.

Per la LD, WB e RW vengono propagati su ME/WB, AluOUT e EN hanno la stessa funzione della ST, MRead abilita la memoria in lettura, il contenuto della locazione indirizzata viene memorizzato nel campo MemDat di EX/WB.

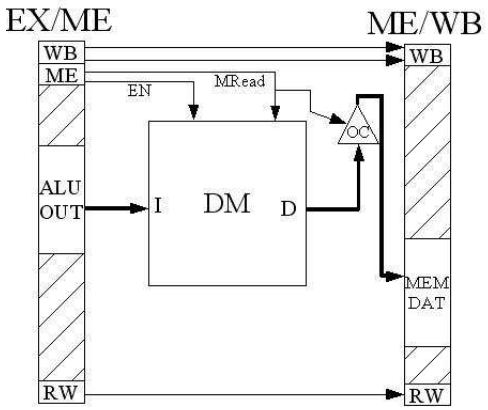

fig. 5.2.4a

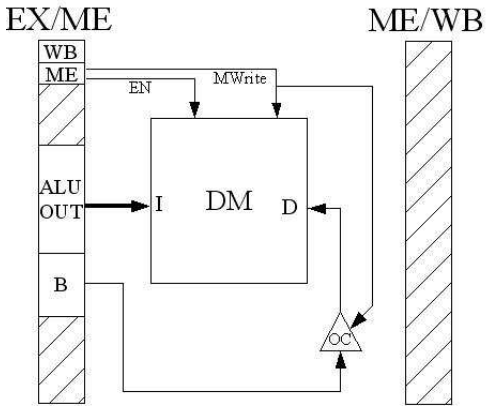

fig. 5.2.4b

Fase WB

Relativa solo alla LD, del tutto simile alle istruzioni aritmetiche, l'unica differenza è che in questo caso MemToReg abilita come uscita MemDat. Tutti

i segnali vengono propagati fino al RF dove il contenuto di MemDat sarà memorizzato nel registro indirizzato da RW.

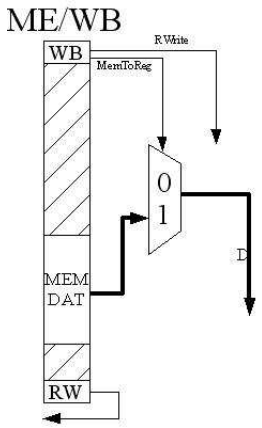

fig. 5.2.5

5.3 JMP

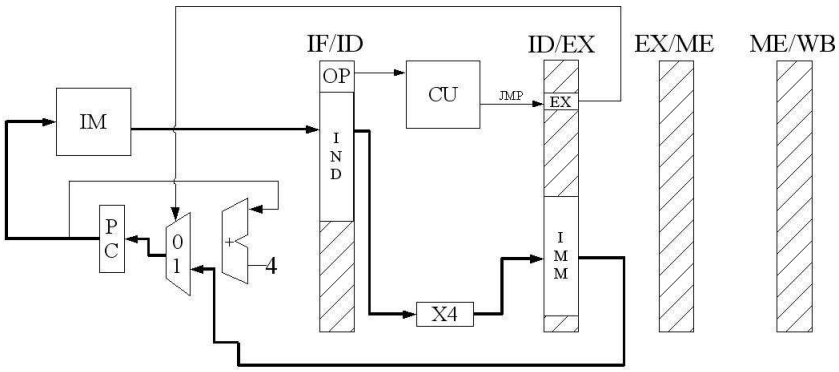

fig. 5.3

Fase IF

Comune a tutte le istruzioni di salto, in paricolare notiamo il multiplexer che seleziona l'ingresso del PC. I campi interessati di IF/ID sono solo due : OP e

IND rispettivamente per il codice operativo dell'istruzione e l'indirizzo a cui saltare.

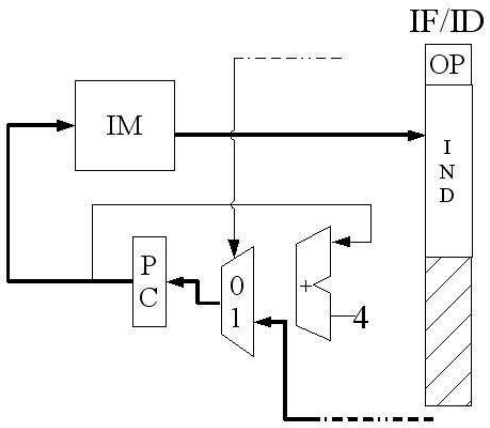

fig. 5.3.1

Fase ID

E' l'ultima fase di esecuzione della JMP. La CU, sulla base di OP genera solo il segnale di salto (JMP=1), il campo IMM di ID/EX corrisponde a IND moltiplicato per 4 ed esteso su 32bit dal blocco X4.

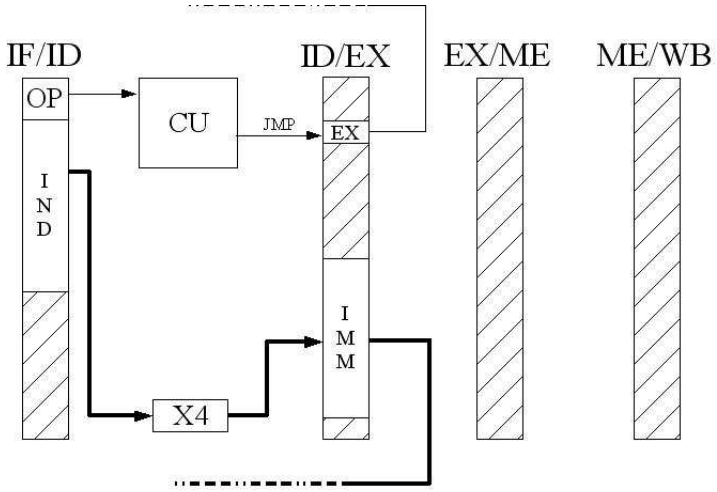

fig. 5.3.2

Fase EX,ME,WB

Nessuna operazione

5.4 JR

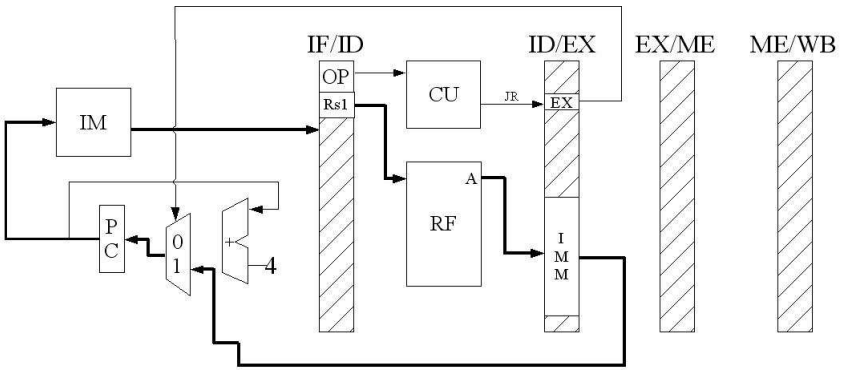

fig. 5.4

Fase IF

Del tutto analoga alla JMP; in IF/ID questa volta abbiamo sempre OP per il codice operazione e Rs1, relativo al registro in cui è memorizzato l'indirizzo a cui saltare.

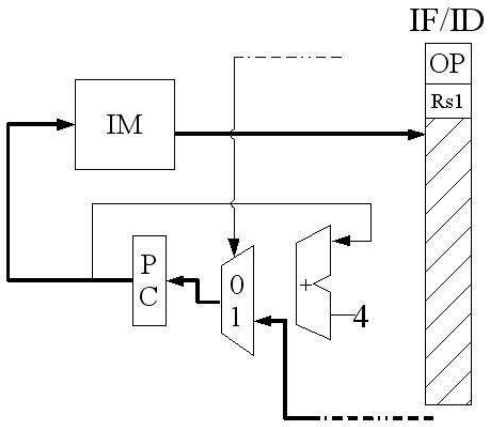

fig. 5.4.1

Fase ID

Come per la JMP, la CU genera il segnale di salto (JR=1), Rs1 è in ingresso al RF per la selezione del registro, l'output (già su 32bit) viene memorizzato in IMM.

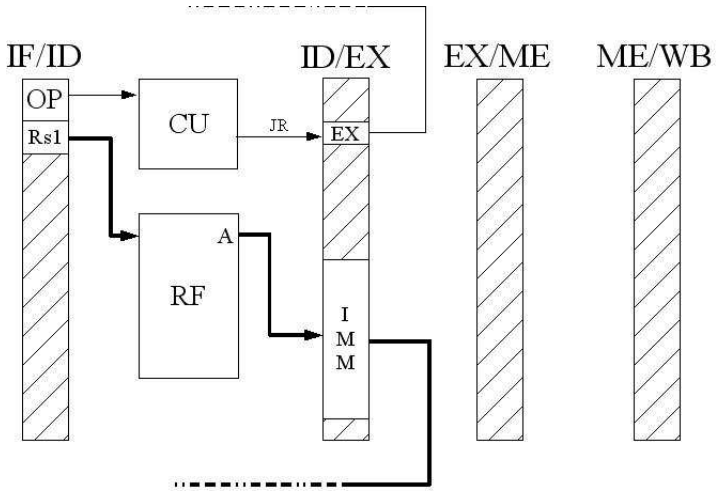

fig. 5.4.2

Fase EX,ME,WB

Nessuna operazione.

5.5 JAL

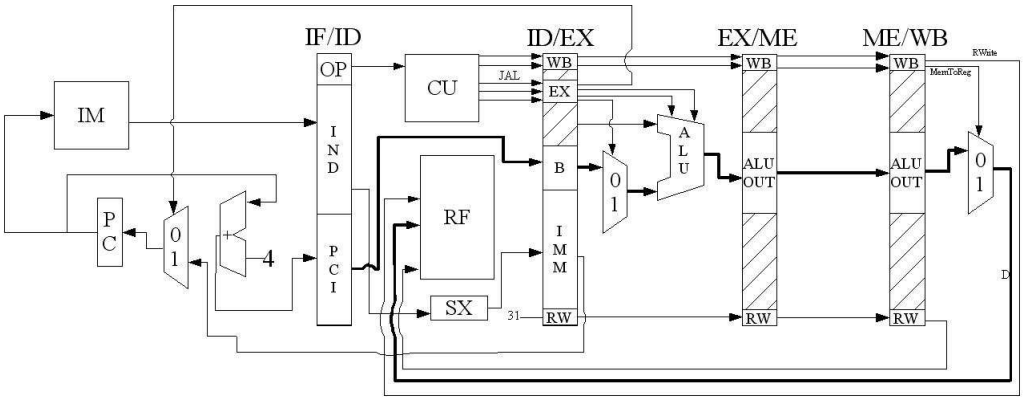

fig. 5.5

Fase IF

Analoga alla JMP per quanto riguarda il PC, IF/ID è così diviso: OP per il codice operazione, IND per l'indirizzo a cui saltare, PCI per l'indirizzo dell'istruzione successiva da memorizzare in R31.

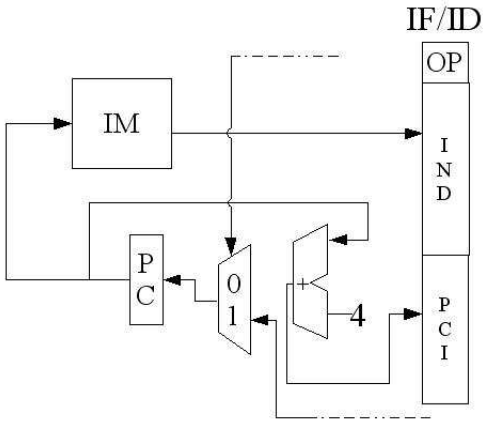

fig. 5.5.1

Fase ID

La CU genera i seguenti segnali relativi alle fasi EX e WB:

RWrite=1 per la scrittura in RF di PCI

MemToReg=0 selezione di AluOUT come uscita dallo stadio WB

AluMUX=0 selezione di B di ID/EX come ingresso della ALU

OpALU=00 nessuna operazione, la ALU propaga in uscita il segnale in ingresso

JAL=1 segnale di salto

Il campo IND attraverso il modulo X4 viene moltiplicato per 4, esteso su 32bit e memorizzato su IMM; PCI viene memorizzato in in B di ID/EX mentre in RW ci viene scritto 31 per la selezione in RF di R31.[1]

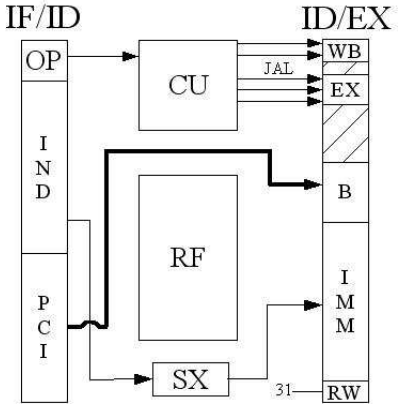

fig. 5.5.2

[1]La scrittura di PCI in R31 deve avvenire nella fase di WB, non prima per evitare conflitti con le istruzioni presenti negli stadi successivi della pipeline.

Fase EX

JAL e IMM vengono propagati verso il PC mentre OpALU e AluMUX fanno si che il campo B di ID/EX possa essere memorizzato in AluOUT senza modifiche. WB e RW vengono propagati nei corrispettivi campi di EX/ME.

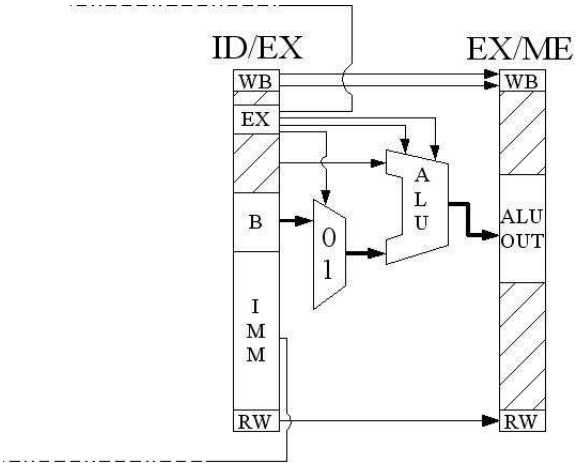

fig. 5.5.3

Fase ME

Non essendo richeista nessuna operazione sulla memoria, tutti i segnali di EX/ME vengono propagati verso ME/WB.

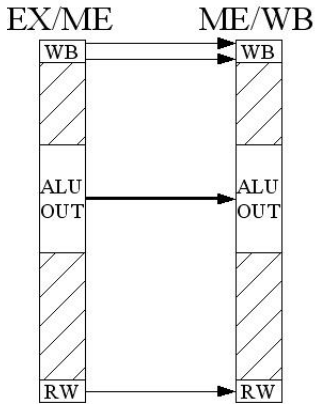

fig. 5.5.4

Fase WB

Del tutto simile alle operazioni aritmetiche, MemToReg selelziona AluOUT come uscita dello stadio che con RWrite e RW vengono propagati verso il RF dove avverrà la memorizzazione di AluOUT (PCI) in R31.

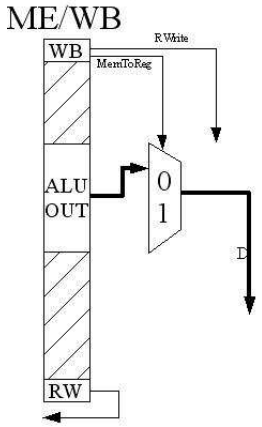

fig. 5.5.5

5.6 JE/JS

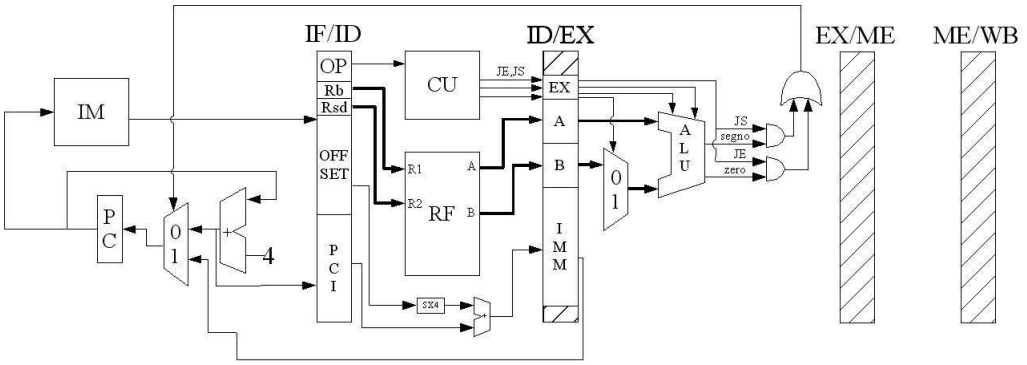

fig. 5.6

Fase IF

Analoga alla JMP, IF/ID è costituito da OP, codice operazione; Rb,Rsd registri di confronto; OFFSET spiazzamento; PCI indirizzo istruzione successiva.

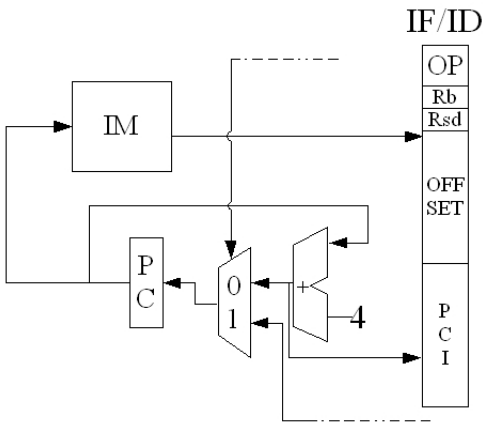

fig. 5.6.1

Fase ID

Essendo prevista solo la fase di esecuzione la CU genera:

JE=1 o JS=1 segnale di salto

OpALU=10 operazione di sottrazione

AluMUX=0 B di ID/EX come ingresso B della ALU

In IMM viene memorizzata la somma di PCI e di OFFSET moltiplicato per 4 ed esteso in segno su 32bit dal blocco SX4.
Il RF sulla base del contenuto di Rs e Rsd manda in uscita su A e B i contenuti dei registri corrispondenti.

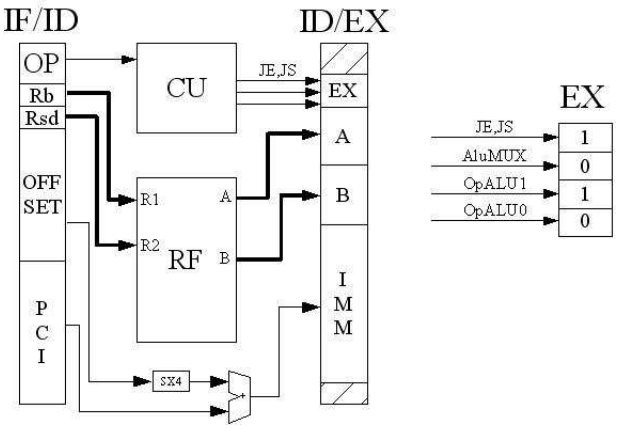

fig. 5.6.2

Fase EX

IMM viene propagato verso PC, la ALU prende in ingresso A e B di ID/EX e ne fa il confronto. In uscita abbiamo Zero e Segno che vanno in AND rispettivamente con JE e JS. I risultati costituiscono i due ingressi dell'OR successivo il cui output costituisce l'abilitazione del multiplexer di PC. Ciò perchè sulla base del confronto si decide se saltare o meno.

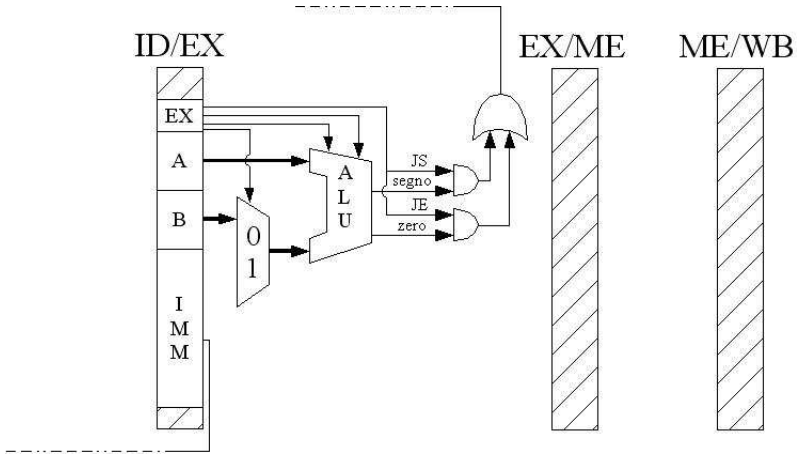

fig. 5.6.3

Fase ME,WB

Nessuna operazione

Chapter 6

Conflitti di Controllo

In un'architettura basata sull'uso di pipeline possono sorgere delle situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente seguente, tali eventi sono detti conflitti; fra le varie tipologie possiamo considerare i Conflitti di Controllo che derivano dalla presenza nel codice di diramazioni e di altre istruzioni che modifcano il valore di PC.
Per quanto riguarda le istruzioni useremo il termine di "salto" per i salti incondizionati e "diramazione" per i salti soggetti a condizione, in particolare si parlerà di diramazione "effettiva" e "non effettiva" a seconda che la condizione sia verificata o meno.

6.1 Salti

Analizziamo la situazione riportata in fig. 6.1.1 (ipotizzando che il PC venga aggiornato nella fase di EX):

LD R2,100(R4)
ADD R3,R3,R4
SUB R5,R6,R7
JMP indirizzo

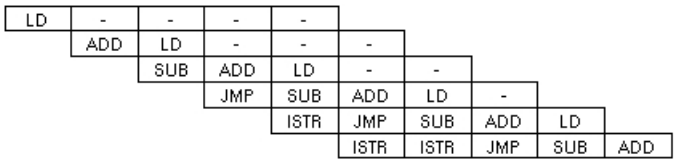

fig. 6.1.1

Bisogna introdurre due bolle per far terminare correttamente l'esecuzione delle instruzioni ADD e SUB e per non farne caricare altre due, per poi rieseguire le istruzioni dalla destinazione del salto. A questo punto la pipeline può riprendere il suo corretto funzionamento, ma si determina una bolla di due cicli di clock.

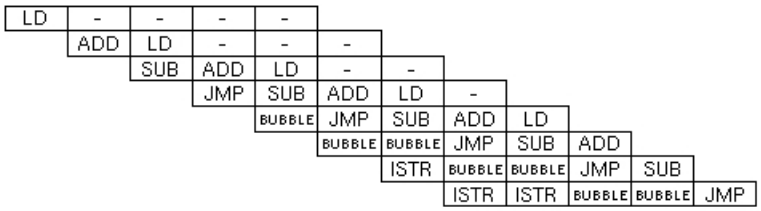

fig. 6.1.2

La cosa è del tutto irrilevante se ciò capitasse di rado ma è stato statisticamente provato che a seconda del tipo di macchina e di programma la frequenza dei salti varia dal 2% all'8%.
Una possibile soluzione a questo tipo di problema è ritardare il salto di un numero di cicli di clock pari a quelli che servono per individuarlo e aggiornare il PC. In questo caso dato che PC viene aggiornato in EX basta anteporre l'istruzione di salto di due, tuttavia c'è la possibilità di anticipare alla fase di ID l'aggiornamento di PC, in questo caso basta scambiare l'istruzione di salto con quella precedente.

Salto Ritardato - Delay Branch (Soluzione Software)

Questo tipo di soluzione si basa fondamentalmente sul compilatore il quale analizzando il codice prevede le situazioni di salto e riempie i cicli di clock che andrebbero perduti con altre istruzioni (Delay Slot). Tipicamente l'istruzione di salto viene anticipata. Consideramo il codice di cui sopra, se ordiniamo le istruzioni in modo che la sequenza diventi la seguente:

LD R2,100(R4)
JMP indirizzo
ADD R3,R3,R4
SUB R5,R6,R7

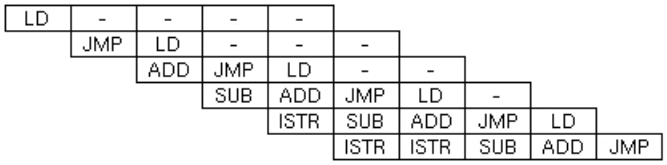

fig. 6.1.3

ci accorgiamo che non occorre inserire le due bolle dato che la ADD e la SUB verranno tranquillamente portate a termine e contemporaneamente saranno caricate nella pipeline le istruzioni relative alla destinazione del salto. Naturalmente se si intende evitare gli stalli[1] relativi ai salti, occorre che la logica della CPU non ne introduca dei suoi. Il problema fondamentale legato a questa tecnica è dovuto al fatto che un programmatore che lavori in assembler con un compilatore che applica il Delay Branch, deve prevedere come si comporta la macchina e riempire con della logica di programma eventuali intervalli di ritardo.
Notiamo il particolare come sia sempre possibile in teoria applicare la tecnica

[1]Situazione nella quale la pipeline viene arrestata per eliminare la presenza di un conflitto

del salto ritardato, ciò nonostante si potrebbero verificare situazioni intricate nelle quali il compilatore in alternativa può riempire gli intervalli di ritardo con delle NOP.

6.2 Diramazioni

Il problema legato alle diramazioni (salti condizionati) è più complesso perchè la condizione che determina il salto potrebbe verificarsi o meno, quindi il problema che si pone è legato all'istruzione successiva che si deve caricare in pipeline: istruzione di destinazione (diramazione effettiva) o istruzione successiva (diramazione non effettiva).
Pertanto nel caso di diramazione vengono eseguite le seguenti operazioni:

- se si preleva un'istruzione di diramazione, la pipeline continua a caricare le istruzioni successive;
- nello stadio EX viene effettuato il confronto, se la verifica di questo ha esito positivo la pipeline viene svuotata delle istruzioni successive alla diramazione (Pipeline Flushing) e viene caricata l'istruzione di destinazione del salto, questo comporta l'introduzione di un numero di bolle pari ai cicli di clock persi; se la verifica ha avuto esito negativo, tutto procede regolarmente come se la diramazione fosse una normale istruzione.

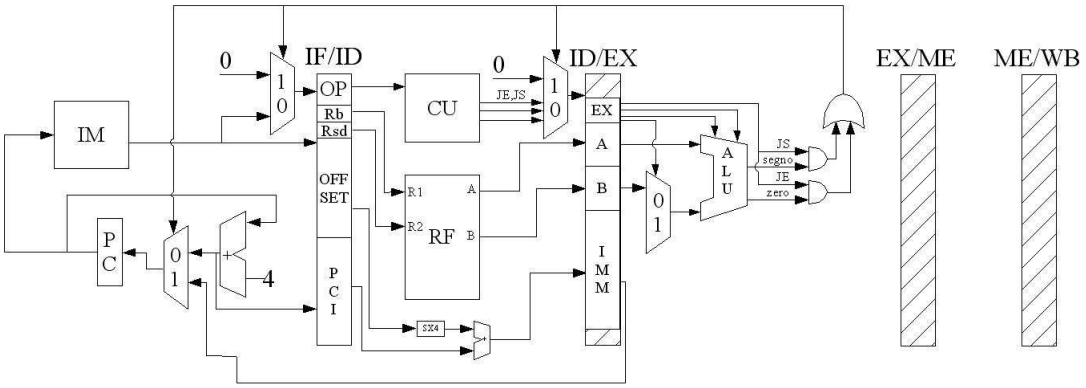

fig. 6.2

In fig 6.2 viene mostrata la logica relativa alla valutazione delle condizioni di diramazione e all'inserimento di bolle nella pipeline. Come evidenziato, dato che la valutazione avviene nello stadio EX bisogna introdurre due bolle rispettivamente in IF/ID e in ID/EX.

E' possibile evitare ciò facendo ricorso, come per i salti, ad una soluzione software: le diramazioni ritardate.

E' ovvio che il problema in questo caso sarà più complesso da gestire dovuto al fatto che eventuali istruzioni precedenti possono lavorare sui registri di confronto evitado così l'anticipazione dell'istruzione di diramazione; in ogni caso per tentare di riempire le eventuali bolle di diramazione il compilatore può prendere le istruzioni di riempimento da tre differenti posti: dalle posizioni che precedono la diramazione, dalla destinazione del salto, dalle istruzioni successive.

Chapter 7

Accenni su Pipeline Relative a Processori Commerciali

7.1 Intel Pentium

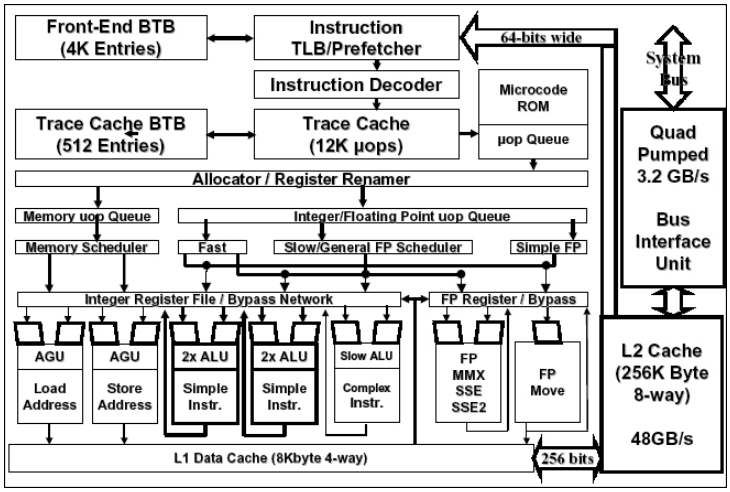

fig. 7.1.1

Il nuovo pentium 4 è dotato di una pipeline a 20 stadi, Intel definisce questa tecnologia "Hyper Pipelined Technology". Similmente al processore x86, il ciclo di esecuzione di un'istruzione che non coinvolge l'unità di float-

ing point è diviso in cinque stadi:

PF	lettura della codifica dell'istruzione successiva
D1	decodifica istruzione
D2	vengono calcolati gli indirizzi effettivi degli operandi
EX	accesso agli operandi della Data Cache nonchè esecuzione dell'istruzione[1]
WB	scrittura del risultato nei registri o in memoria[2]

La particolarità del processore pentium sta nell'utilizzo una architettura superscalare, basata sull'esecuzione in parallelo di fasi omologhe di istruzioni successive. Quindi se nel processore x86 solo un'istruzione poteva essere in esecuzione in ogni stadio, nel pentium, grazie a questa architettura, è possibile eseguire due istruzioni in parallelo per ogni stadio della pipeline come mostrato in fig. 7.1.2:

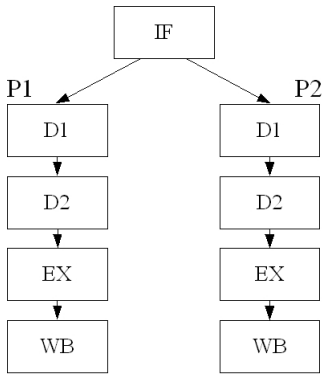

fig. 7.1.2

[1] nel caso l'esecuzione dell'istruzione prevedesse accessi in memoria questo stadio potrà richiedere più di un ciclo di clock

[2] la fase ME non è presente perchè viene utilizzata la Data Cache

Le due pipeline operano sincronamente, gli stadi D1 e D2, di P1 e P2 sono iniziati e terminati sincronamente. Se P1 è bloccata in EX, P2 può avanzare. Le istruzioni sulle due pipeline, in generale, non possono avere dati in comune, la presenza di una coppia di istruzioni incompatibile viene rilevata via hardware durante la fase D1, la stessa logica provvede poi a disattivare P2 durante la fase di EX di P1; in questo caso l'istruzione in P2 non verrà eseguita in parallelo con quella in P1 ma seguirà la prima con un ciclo di ritardo. Nessuna istruzione può avanzare nello stadio EX finche P1 e P2 non sono in WB.

L'unità di floating point è integrata nello stesso chip del processore e consente l'esecuzione di una istruzione floating point ad ogni ciclo di clock; è organizzata in una pipeline a 8 stadi i cui primi 5 coincidono con quelli eseguiti dalle istruzioni sugli interi.

Istruzioni floating point non possono essere eseguite contemporaneamente (nello stesso stadio) ad istruzioni intere poichè per i primi 5 stadi utilizzano le stesse unità funzionali.

7.2 Amd Athlon

Gli elementi principali che compongono la CPU Athlon sono illustrati dallo schema a blocchi in fig. 7.2.1:

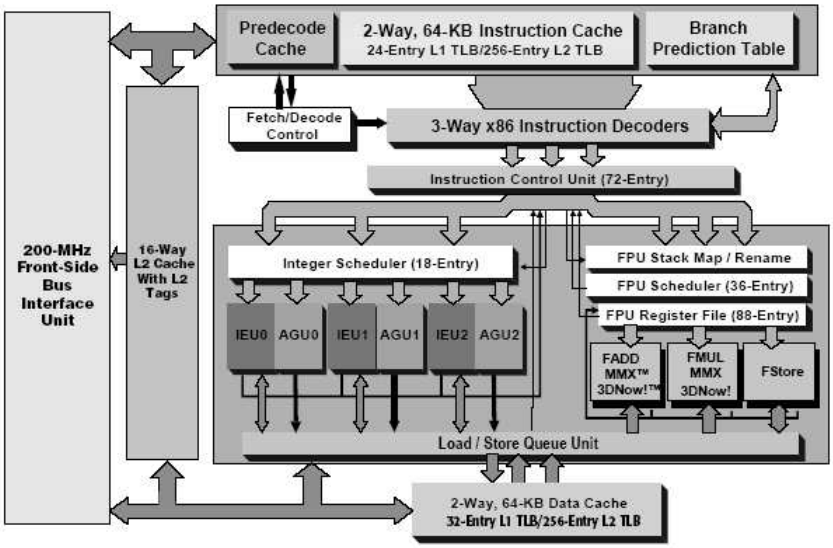

fig. 7.2.1

Possiamo notare:

- la cache L1 da 128 KB separata per Istruzioni e Dati
- il blocco di fetch e decode
- la Instruction Control Unit (ICU) a 72 ingressi
- i due scheduler indipendenti per numeri interi e floating point
- le 6 unità per interi (3 IEU (Integer Executive Unit) e 3 AGU (Addres Generation Unit))
- le 3 unità per floating point (FADD, FMUL e FSTORE)
- la LSU (Load/Store Unit) e la BIU (Bus Interface Unit)
- il controller per la cache L2 esterna

La pipeline ha una lunghezza di 10 stadi per le istruzioni intere mentre per le floating point varia da 12 a 15.

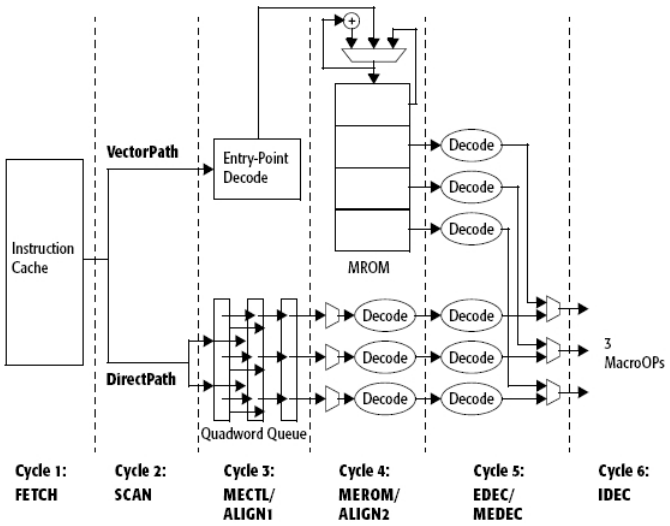

fig. 7.2.2

FETCH vengono prelevati 16 bytes dalla Instruction Cache

SCAN fase in cui si decide quali istruzioni sono abbastanza semplici da poter essere decodificate via hardware mediante opportuna logica oppure se necessitano di una decodifica mediante sequenze di microcodice. Pertanto successivamente alla fase di Scan le istruzioni possono prendere rispettivamente due vie, Direct Path Decoder nel caso di decodifica hardware e Vector Path Decoder, nel caso di decodifica tramite microcodice.

Direct Path Decoder

ALIGN1 formata da 3 buffer di 8 byte in coda capaci di poter immagazzinare fino a 24 bytes di istruzioni, il contenuto di ogni buffer

viene inviato come una sequenza di 3 istruzioni x86 (2.5 bytes per istruzione) allo stadio successivo

ALIGN2 completa la fase di allineamento e invia le istruzioni allo stadio successivo

ALIGN3 ricava dall'output della fase precedente 3 MacroOPs (MOPs)[3]

Vector Path Decoder

MECTL (Microcode Engine Control) stadio in cui le istruzioni vengono predisposte all'ingresso di una Microcode ROM

MEROM ad ogni istruzione x86 viene associata una corretta sequenza di microcodice

MEDEC le linee di microcodice vengono tradotte in un massimo di tre MOPs

IDEC le MOPs del Vector Path Decoder vengono riordinate in ordine di programma con quelle del Direct Path Decorder e passate alla ICU (Instruction Control Unit), 3 per ciclo

ICU segue le varie istruzioni nella fase di esecuzione, è formata da scheduler, da un reoder buffer composto da 24 linee ognuna della dimensione di 3 MOPs per un totale di 72 MOPs gestibili. Ad ogni ciclo preleva dalla fase successiva fino a 3 MOPs e le memorizza in un'entrata del reoder buffer

Le MOPs vengono trattate in modo differente a seconda se riguardano operazioni fra interi o in floating point. Nel caso di floating point vengono inviate

[3]MOPs sono istruzioni di lunghezza fissata contenenti un gruppo di OPs prefissato, in genere due

direttamente alla floating point pipeline mentre se sono relative a numeri interi sono inviate al proprio scheduler.

Integer Instruction Pipeline

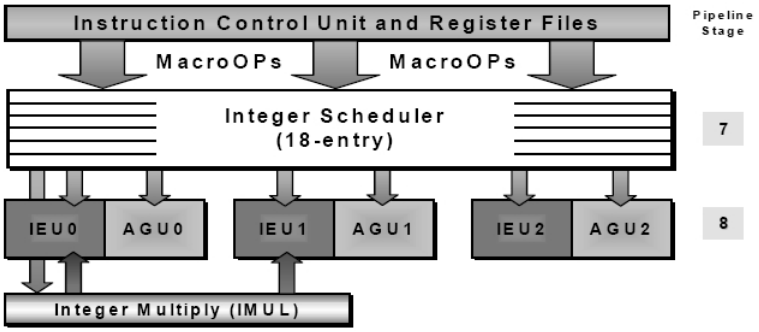

fig. 7.2.3

IS (Integer Scheduler) è dotato di una coda profonda 3 posizioni per memorizzare le tre MOPs che arrivano dallo stadio precedente. Da ogni MOPs ricava 2 OPs (una per la IEU e l'altra per la AGU). Ad ogni ciclo può ricavare un massimo di 6 OPs che redirige verso le corrispettive unità dello stadio successivo (3 IEU e 3 AGU).

IEU (Integer Execution Unit) si occupano delle usuali operazioni aritmetico-logiche e della risoluzione dei branch[4]

AGU (Addres Generation Unit) eseguono le addizioni necessarie al calcolo degli spiazzamenti degli indirizzi per le Load/Store

ADDR stadio in cui si passa se è necessario accedere ai dati neccessari al completamento delle operazioni dove viene calcolato l'indirizzo della OP di Load/Store e inviato alla Data Cache

[4]tutte le istruzioni vengono eseguite in un ciclo di clock tranne le moltiplicazioni e le divisioni che ne richiedono più di uno, queste vengono eseguite da due delle IEU (IEU0 e IEU1) che condividono un moltiplicatore e un divisore

DC avviene l'accesso alla Data Cache e le OP che attendevano di essere eseguite nello scheduler possono essere completate se tutti gli altri operandi sono disponibi

Floating Point Instruction Pipeline

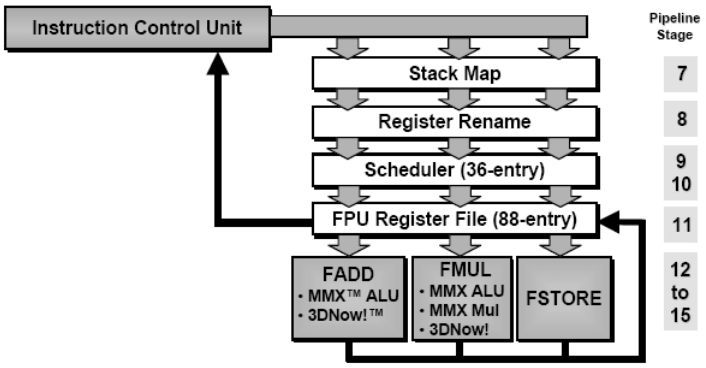

fig. 7.2.4

SR (Stack Renaming) in questo stadio vengono assorbite le 3 MOPs provenienti dalla ICU e il floating point stack viene riallocato su uno spazio di indirizzi più ampio. Ciò dovuto al fatto che la FPU esegue anche istruzioni x87 e 3DNow!/MMX condividendo gli 8 registri a 64 bit dello stack classico. In questo modo gli stack relativi al 3DNow!/MMX vengono resi visibili a seconda del contesto e quindi indipendenti

RR (Register Rename) stadio di ridenomiazione dei registri per evitare dipendenze

Scheduler ricezione e scheduling delle 3 MOPs grazie ad uno scheduler a 36 entrate

FPU RF (Floating Point Unit Register File) determina gli operandi e le 3 OPs che sono inviate alle 3 unità esecutive

FMUL	caricamento dei numeri in floating point; moltiplicazione tra numeri in foating point; esecuzione di complesse operazioni in floating point; calcoli MMX e 3DNow!
FADD	caricamento dei numeri in floating point; somme tra numeri in floating point; esecuzione di complesse operazioni in floating point; calcoli MMX e 3DNow!
FSTORE	memorizza numeri in floating point e MMX: esegue altre operazioni complesse

Bibliography

[1] Giacomo Bucci - *Architetture dei calcolatori elettronici* - McGraw-Hill, 2001

[2] http://www.lithium.it

[3] http://www.intel.com

[4] http://www.amd.com

[5] http://www.gest.unipd.it/materiale-didattico/ece2/Pentium.html (G. Manduchi - M. Moro)

[6] http://www.ing.unife.it/elett/SistemiElaborazione (Materiale Didattico Prof. Massimo Piccardi)