

Introduction to computer Design

L'evoluzione verificatasi nelle architetture dei sistemi di elaborazione a partire dagli anni '80 è da motivare principalmente grazie a 2 eventi:

- Evoluzione nei dispositivi a semiconduttore
- Modifiche al progetto delle architetture del sistema (soprattutto del processore)

Quanto a prestazioni, inizialmente le curve di rendimento si è mantenuta relativamente bassa a come delle mancanze di evoluzione nelle creazioni di dispositivi a semiconduttore. Con l'avvento dei processori RISC l'incremento delle prestazioni diventa significativo: un contributo è dovuto anche all'avvento dei compilatori, grazie anche ad una più stretta collaborazione tra chi progetta l'hardware e chi realizza i compilatori.

Diverse varianti:

- Super Computer: vengono rimpiazzati da collezioni di microprocessori
- Main frame: sono stati sostituiti dai sistemi multiprocessori per il costo. Destinati a grossi centri di calcolo.
- Mini computer: sono stati sostituiti da server basati su microprocessore
- PC e Workstation: Negli anni 70 dominano il mercato

I super computer hanno parti significative fatte ad hoc per realizzare un determinato compito. Ciò accade per i PC e Workstation che invece di quel tipo di comportamento hanno ben poche, poiché si punta ad un assemblaggio intelligente di moduli esistenti sul mercato. Differenze di costo!!!

Ad oggi la classificazione che viene fatta è la seguente:

1. Desktop computing
2. Server
3. Embedded computer: sistemi di elaborazione general purpose con hardware e software specifico.

1. I Desktop Computing sono macchine con prezzo compreso tra i 1000\$ e i 10000\$. In queste macchine il problema è trovare un compromesso tra costo e prestazioni (PC & Workstation)
2. Nel caso di Server, vengono presi in considerazione alcuni parametri fondamentali:
 - a. Disponibilità: è la probabilità che in un generico istante la macchina funzioni (n° di volte che la macchina va in crash nell'arco di un certo intervallo di tempo)
 - b. Scalabilità: è la possibilità di aumentare le prestazioni aggiungendo moduli nuovi.
 - c. Throughput: è la quantità di lavoro che viene fatto (è diverso dalle prestazioni che sono indicazioni più generiche). Da notare è che i desktop sono macchine mono utente mentre i server devono gestire molti eventi in "Parallelo".
3. Gli Embedded: sono macchine che assolvono un ben determinato scopo. Stiamo parlando di sistemi che sia per caratteristiche hardware che per i problemi che risolvono si caratterizzano come sistemi diversi e con tipologie di progetto variabile a seconda anche dei vincoli imposti dal committente. La serie di esempi di vincoli da poter soddisfare è:
 - a. Real-Time performance: fondamentale la velocità di risposta da parte del sistema.
 - b. Memory minimization: minimizzazione dei costi basata su uno dei pochi banchi di memoria.
 - c. Consumo di potenza

Esistono una gamma di soluzioni abbastanza varia per implementarli: ciò che interviene può essere: le prestazioni, il costo gli aspetti real-time.

A seconda dei casi ci sono diversi mix che realizzano determinati vincoli:

- Standard DSP + SW (ad hoc): basso costo con prestazioni soddisfacenti
- Standard PROC + Custom SW: funzionalità realizzata da software specifico.
- Processore + SW ad hoc + Logica ad hoc : ad oggi si aggiunge la logica in FPGA (logiche programmabili)

Quando si parla di prestazione i due parametri che vengono presi in considerazione sono:

1. Tempo di risposta: tempo che intercorre dalla richiesta di una operazione al momento in cui l'operazione viene svolta. Questo parametro è determinante per l'utente.
2. Throughput: total o ammontare della mole di lavoro svolto in una certa unità di tempo. Questo parametro è determinante per il sistemista.

Si vuole che il throughput sia massimo e sia minimo il tempo di risposta. Come è facile capire non è semplice ottimizzare entrambe i parametri.

Come già detto all'utente interessa il tempo di risposta, mentre dal punto di vista di chi progetta è interessante cosa ha contribuito a determinare quel tempo (cioè il tempo di CPU, tempo di risposta di un periferica, tempo di IDLE ecc...)

Il tempo dunque ha diverse componenti che contribuiscono a ciascuno dei due parametri. Il calcolo e l'analisi delle prestazioni non è dunque un lavoro semplice.

All'atto della progettazione di un processore, vanno ovviamente date delle linee indicative sulla fascia che il processore dovrà occupare. Il progettista dovrà farsi un'idea delle prestazioni che una certa architettura potrà avere, e vedremo che ci sono una serie di scelte architetture volte a migliorare le prestazioni. Ad esempio prendiamo il caso della cache e delle quantità di memoria da dedicare alla cache: ma dove sta il punto di max di aumento delle prestazioni in relazione all'aumento della cache? Ovviamente vanno presi in considerazione anche gli applicativi: se il processore è embedded sarà in una fascia stretta di applicazioni, oppure se è general purpose si dovranno selezionare dei programmi campione (benchmark) rispetto ai quali valutare le prestazioni.

I benchmark rappresentano la tipica applicazione che dovrà girare su quella architettura (ad esempio nel caso del sistema bancario ci servono degli applicativi, mentre nel settore dei processori general-purpose le cose cambiano).

Ecco che quindi saltano fuori diversi benchmarks:

- Programmi reali: (compilatori, editor di testo...) opportunamente modificati.
- Kernel: ad esempio LiverCore. Linpeck, ovvero parti critiche di programmi frequentemente utilizzati).
- Toy benchmark: setaccio di Eratostene -> numeri primi, quick sort-> ordinamento e quindi valutare il costo di un certo elemento.
- Synthetic BM: righe di varie tipologie di programmi

Un set standard di benchmark è lo SPEC92, ed è un insieme di benchmarks con cui si deve testare la propria architettura al fine di certificarne le prestazioni. L'esistenza del set standard garantisce che i risultati siano reali e confrontabili tra tutti i produttori di architetture. E' fondamentale quindi scegliere i benchmark. Supponendo di scegliere gli SPEC92, avrò un certo numero che caratterizza le prestazioni del mio processore rispetto ad ogni SPEC92. Come faccio a rapportarmi a gli altri produttori per poter dire a che livello è la mia architettura? E' Necessario combinare in maniera opportuna i risultati ottenuti dai vari benchmark utilizzati!

Una volta ottenuti i risultati e valutate le prestazioni, supponiamo che venga individuato un punto debole, ad esempio l'unità floating-point è lenta. Dobbiamo quindi rendere più veloce quel componente.

Subentra la legge di Amdahl: se vuoi sapere quanto più veloce andrà il tuo sistema dopo aver migliorato la componente lenta devi anche considerare non solo la nuova velocità della componente (Fraction enhancement) ma anche la frequenza di utilizzo di quella componente (Speedup Enhancement).

Tale legge permette di determinare quanto può essere il guadagno dell'incremento delle prestazioni di una certa componente. In genere in fase di progetto quello che si deve fare è creare un modello che simuli l'architettura che sto progettando e che mi dica come va il mio processore (il modello va provato su una rete di Workstation e spesso richiede anni per un processore dei giorni nostri e di categoria general purpose).

L'ultima spiaggia è la costruzione di un prototipo per ottenere dei risultati.

Un modello semplice è l'equazione delle prestazioni di una CPU banale (ad es 80x86):

$$CPU_{time} = \left(\sum_{i=1}^n CPI_i * IC_i \right) * Clock\ cycle\ time$$

Laddove:

- **CPI_i**: n° di colpi di clock richiesti da una parte dell'istruzione *i* (dipende dall'istruzione set e HW organizzato)
- **IC_i**: n° di volte che l'istruzione *i* è eseguita all'interno del programma (dip. dall'istr. Set e dal compilatore)
- **Clock cycle time**: è l'inverso della frequenza di clock e dipende dall'organizzazione e dalle tecniche HW.

Questo è un meccanismo banale che però ci permette di fare delle prime stime. Con questo modellino mescoliamo insieme 3 parametri che hanno un legame stretto con 3 parti estremamente eterogenei. Vediamo perché:

- **CPI** -> dipende dalla componente della CPU interessata dall'operazione
- **IC** -> dipende dal compilatore oltre che dal programma in se.
- **Clock cycle time** -> dipende strettamente dall'HW

Questa equazione ha però dei limiti poiché se applicata ai sistemi odierni deve subire delle modifiche al fine di rispecchiare la situazione reale. L'equazione diventa praticamente inutilizzabile nel caso dei processori superscalari.

Instruction Set Principles

Vedremo quali sono le **istruzioni** che il processore offre, come anche la **modalità operative** e i modi di **indirizzamento**. Per chi crea un compilatore tutto ciò ha un impatto determinante, ma si mostrano anche dei riflessi verso l'esterno per quanto riguarda la minimizzazione delle quantità di memoria da dedicare alle righe di codice, nel caso di applicazioni embedded la memoria è un elemento critico da rendere minimo e ci+ ha un impatto anche sul modo in cui il processore viene presentato a chi scriverà codice per quel processore. Non si guarda come il processore è fatto dentro, quanto poi ai modi di indirizzamento, a quanti registri ha e poche altre nozioni.

L'Instruction Set architecture viene definita da chi progetta il processore, ed in una fase molto avanzata del periodo di progetto. I processori possono essere suddivisi sulla base di come suddividono gli operandi e le istruzioni tipiche, ovvero:

- Register-Memory: è quell' dell'80x86 in cui gli operandi vengono messi nei registri o in celle di memoria. Ci sono però casi in cui una istruzione non supporti tutte le condizioni di quelle due. Il risultato va a finire in un registro o in una cella di memoria.
- Accumulato: caso in cui tutti i registri collasano in un unico registro che è l'accumulatore. Come soluzione non è efficiente. Ci dovranno essere delle soluzioni che mi permettono di spostare il contenuto dell'accumulatore in memoria: ciò richiede frequenti accessi in memoria. Tale istruzione è eseguita tra un accumulatore e la cella di memoria.
- Register-Register/Load-Store: all'interno del processore c'è un buon numero di registri e le operazioni Aritmetico/Logiche lavorano su registri (vantaggio perché non ci sono indirizzamenti in memoria, sfruttato molto nelle architetture RISC). Load/Store sono le istruzioni che swappano da memoria a registro e viceversa.
- Stack: i registri sono visti come uno stack cioè in pile e l'indirizzamento non lo posso fare in modo diretto ma solo attraverso il top dello stack. Inoltre nel codice avrò moltissime istruzioni frammentate di operazioni sullo stack. E' molto semplice poiché non ci sono modi di indirizzamento dato che operandi e risultato stanno all'interno dello stack.

Ovviamente guardando il codice questo varierà sensibilmente in relazione anche a quale di queste 4 soluzioni si sfrutta.

Attualmente, tutti i processori sono General-Purpose Register Machine.

Le ragioni per cui ciò avviene sono le seguenti:

- I registri sono di gran lunga più veloci della memoria
- I registri sono più semplici da gestire per un compilatore

Un altro modo per classificare la CPU è quello di andare a guardare:

- Il n° di operandi su cui opera una istruzione, dove per operandi non intendiamo quelli logici ma quelli su cui opera l'istruzione in se. Ad esempio l'80x86 è un processore a due operandi di cui uno può essere una cella di memoria. Tipicamente più aumenta il numero di operandi più aumenta la lunghezza dell'istruzione. Altro impatto è quello sulle prestazioni qualora si operi su celle di memoria (più lenta) o su registri (di sicuro la più veloce)
- Tipico n° di operandi di memoria per ogni istruzione ALU

Ci sono anche differenze sui modi di indirizzamento in memoria. Vediamole.

Le alternative sono tra due:

- Little-endian: indirizzo punta al least significant byte.
- Big-endian: indirizzo punta al most significant byte

Altro problema nasce nel caso di accessi allineati o non allineati. Cosa vuol dire: significa che in base a come io organizzo gli indirizzi (ad es. 8 bit) allora dovrò avere gli indirizzi. Il vantaggio sta in termini di prestazioni perché gli accessi in memoria sono semplificati, ma lo svantaggio sta nello spreco di memoria nel caso di memorizzazione di dati molto eterogenei tra loro. Nel caso di applicazioni embedded non si propende per l'allineamento poiché è determinante lo spreco di memoria.

Altro elemento è la scelta del numero dei modi di indirizzamento.

Più ce ne sono e peggio è perché in fase di decodifica si spenderà + tempo oltre che molti bit per la decodifica stessa.

Nei processori RISC è in quelli superscalari si propende per i pochi modi di indirizzamento, ma in fase di progetto occorre trovare un buon trade-off.

Vediamo una rapida carrellata dei modi di indirizzamento

- REGISTER MODE

| Addressing mode | Example instruction | Meaning | When used |
|-----------------|---------------------|-----------------------------------|--------------------------------|
| Register | Add R4, R3 | Regs [R4] ← Regs [R4] + Regs [R3] | When a value is in a register. |

| | | | |
|-------|---|---|--|
| CODOP | I | X | |
|-------|---|---|--|

CODOP-> combinazione di bit, ad esempio se vorrò 8 istruzioni mi basteranno 3 bit

I-> modi di indirizzamento se (immediate X=val, register X=reg, direct X= indirizzo cella di memoria)

- IMMEDIATE MODE

| | | | |
|-----------|------------|---------------------------|----------------|
| Immediate | Add R4, #3 | Regs [R4] ← Regs [R4] + 3 | For constants. |
|-----------|------------|---------------------------|----------------|

Molto usato per le costanti

- DISPLACEMENT MODE

| | | | |
|--------------|-----------------|--|----------------------------|
| Displacement | Add R4, 100(R1) | Regs [R4] ← Regs [R4] + Mem[100+Regs [R1]] | Accessing local variables. |
|--------------|-----------------|--|----------------------------|

È il modo di indirizzamento per gli accessi in memoria (si compone di costante più registro ADD R4, 100 (R1))

- INDIRECT MODE

| | | | |
|-------------------------------|--------------|--|--|
| Register deferred or indirect | Add R4, (R1) | Regs [R4] ← Regs [R4] + Mem[Regs [R1]] | Accessing using a pointer or a computed address. |
|-------------------------------|--------------|--|--|

- INDEXED MODE

| | | | |
|---------|-------------------|--|--|
| Indexed | Add R3, (R1 + R2) | Regs [R3] ← Regs [R3] + Mem[Regs [R1] + Regs [R2]] | Sometimes useful in array addressing; R1 = base of array; R2 = index amount. |
|---------|-------------------|--|--|

Utile nel caso di calcoli su matrici

- DIRECT MODE

| | | | |
|--------------------|----------------|-----------------------------------|--|
| Direct or absolute | Add R1, (1001) | Regs [R1] ← Regs [R1] + Mem[1001] | Sometimes useful for accessing static data; address constant may need to be large. |
|--------------------|----------------|-----------------------------------|--|

- MEMORY INDIRECT MODE

| | | | |
|------------------------------------|---------------|---|---|
| Memory indirect or memory deferred | Add R1, @(R3) | Regs [R1] ← Regs [R1] + Mem[Mem[Regs [R3]]] | If R3 is the address of a pointer p, then mode yields *p. |
|------------------------------------|---------------|---|---|

ADD R1, @(R3) R3 è l'indirizzo di un puntatore P, carico in R1 *p

- AUTO INCREMENT MODE

| | | | |
|---------------|---------------|--|---|
| Autoincrement | Add R1, (R2)+ | Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + d | Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d. |
|---------------|---------------|--|---|

- AUTO DECREMENT MODE

| | | | |
|----------------|---------------|--|---|
| Auto-decrement | Add R1, -(R2) | Regs[R2] ← Regs[R2] - d Regs[R1] ← Regs[R1] + Mem[Regs[R2]] | Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack. |
|----------------|---------------|--|---|

- SCALED MODE

| | | | |
|--------|----------------------|--|---|
| Scaled | Add R1, 100(R2) [R3] | Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d] | Used to index arrays. May be applied to any indexed addressing mode in some machines. |
|--------|----------------------|--|---|

Nello scegliere il modo quanto a numero e a tipologie bisogna porre attenzione poichè sono differenti I riflessi che si possono ottenere infatti (all'aumentare dei modi di indirizzamento):

- Si riduce il numero di istruzioni
- Cresce il livello di complessità dell'architettura della CPU
- Cresce il carico medio sulle CPI, come aumento del n° di colpi di clock di CPU

Inoltre, qualora nei modi di indirizzamento venga utilizzato un displacement, quanti bit dovrò utilizzare per poter specificare quel displacement? Anche in questo caso ci si è basati sulle statistiche ottenute dagli SPEC92. Ad esempio è scarso (5%) il caso in cui per il displacement occorrono più di 8 bit, ma il tutto come sempre vari in relazione alla destinazione d'uso del processore. Vediamo le conclusioni che si possono trarre dalle analisi quantitative, in termini prestazionali.

- Displacement, immediate e register indirect: 75% -> 99% dei modi di indirizzamento
- N° di bit per il displacement: da 12 a 16 bit
- La dimensione dell'immediate field dovrebbe essere almeno 8 o 16 bit (50% e 80% rispettivamente dei casi).

Altra scelta critica è quella delle operazioni da includere nello instruction set di un processore. Posson esserci a seconda del mercato a cui è dedicato un certo processore alcune appendici per le manipolazioni di particolari tipi di dato (es. 80x86 per manipolazione delle stringhe).

NB: Più le istruzioni sono numerose più sarà grande il codice operativo. I RISC hanno poche istruzioni e questo mi permette di avere un codice macchina più compatto. Quando devo decidere come utilizzare le risorse che ho per migliorare una certa architettura devo come sempre fare riferimento alla legge di Amgahl. Ovviamente non tutte le istruzioni del processore vengono usate con la stessa frequenza, e l'obiettivo comune è quello di velocizzare quelle che sono le istruzioni più comunemente utilizzate.

Istruzioni di controllo del flusso

Sono abbastanza critiche perchè anche queste hanno una frequenza abbastanza elevata. Si possono distinguere 4 categorie:

- Condicional brancher
- Jump
- Procedure Call
- Return

Nell'istruzione di salto non si memorizza l'indirizzo a cui saltare bensì il displacement da sommare al program counter al fine di andare a quell'indirizzo.

Sono rari i casi di salti ad indirizzi molto lontani, ed in genere degli indirizzi ad 8 bit sono più che sufficienti a coprire la maggior parte dei casi circa 90%.

Nel caso di chiamata/ritorno da procedure si propende per passare ad alcune varianti tra cui una che rispetta la gestione delle procedure. Si memorizza l'indirizzo della procedura in un registro e all'istruzione di chiamata si passa il registro, e questa istruzione è molto più compatta rispetto a tutte le altre (5 bit dell'indice del registro vs 32 bit dell'indirizzo).

Per quanto riguarda i processori RISC nel caso di salto condizionato spesso non si guarda al valore di un bit di flag quanto più al valore di un registro (Condition Register). Altra alternativa è quella di effettuare il salto condizionato valutando l'eguaglianza di due registri, ma essendo spezzate le operazioni di COMPARE e di BRANCH ciò potrebbe creare dei problemi con gli interrupt. Queste alternative però mi permettono di avere le due operazioni all'interno della stessa istruzione (ed è il COMPARE ed il BRANCH). L'ipotesi più semplice è quella basata sui bit di flag, che è il CONDITION CODE.

Un elemento da non trascurare è quello della memorizzazione all'indirizzo di ritorno:

- Memorizzato dal chiamante
- Memorizzato dal chiamato

L'architettura deve offrire una scelta per una di queste due alternative.

Altro fattore è considerare quali tipi di operandi trattare e quali bit dedicargli.

Instruction Encoding

Ci si combatte tra due alternative:

- Codifica variabile: si dedica un numero di bit variabile a seconda dell'istruzione che mi trovo davanti. Ad esempio, caso dell'80x86:
 - NOP (1byte)
 - STI (1byte)
 - MOV AX, var (4 byte)
 - MOV var, 5392 (almeno 5 byte)

Complice non poco l'unità di decodifica ma anche l'unità di fetch. L'aver una set di istruzioni variabile mi complica il processore, oltre che renderlo meno efficiente. Di contro la rappresentazione del codice sarà più compatta, a svantaggio dell'efficienza.

- Codifica fissa: si guarda al numero di byte più grande utilizzato e tutte le istruzioni avranno quella dimensione. Ovviamente avrò spreco di memoria, ma avrò un vantaggio in termini prestazionali. Altro vantaggio è avere un posizionamento fisso dei componenti contenuti in un'istruzione. Il grosso

vantaggio di avere una codifica fissa sta quindi non solo nella dimensione fissa delle istruzioni ma anche nel posizionamento fisso, il che permette di avere pochi segnali di decodifica. Ergo l'unità di decodifica si semplifica di gran lunga!!!

La tendenza è quella ad andare verso una codifica fissa.

Register Allocation

Abbiamo detto che più registri ci sono e meglio è perché ciò mi permette di minimizzare gli accessi in memoria. Ma per il compilatore la situazione si complica non poco perché dovrà essere lui a decidere in quali registri andare ad allocare le variabili che vengono dichiarate ed usate. I registri devono essere giocati e devono essere usati in modo tale da avere nei registri le sole variabili che mi servono in un dato momento.

Se il compilatore azzecca le scelte in termini di allocazione delle variabili nei registri, ecco che da un colpo eccezionale alle prestazioni. La soluzione ottima sicuramente richiederebbe troppo tempo (secoli di CPU), per cui si opta per una soluzione di media efficienza.

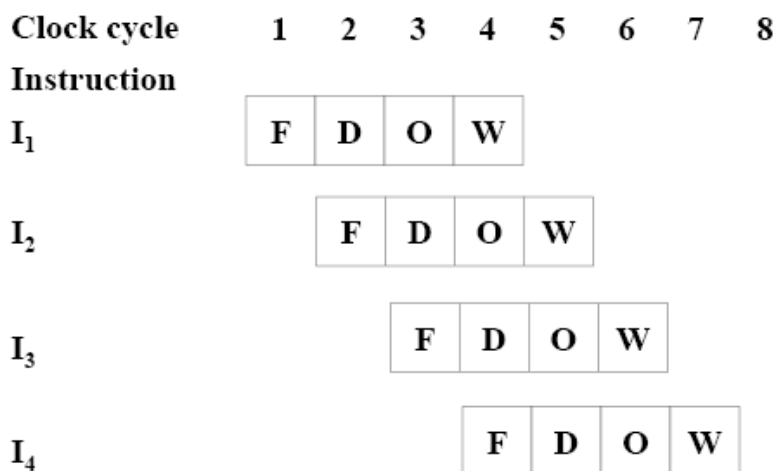
PIPELINING

L'idea è quella di suddividere il ciclo di esecuzione delle istruzioni in diverse fasi (in questo caso 4) facendo in modo che ogni stadio possa essere eseguito in un certo n° di colpi di clock

- Fetch
- Decode
- Execute
- Write Back

Ogni operazione competerà ad un modulo HW che la eseguirà in un colpo di clock.

Ad ogni colpo di clock viene caricata una nuova istruzione in modo tale che ad ogni istante siano tutti e 4 i moduli a lavorare a tempo pieno.



Come si può notare, a parte la fase iniziale di riempimento degli stadi di pipeline con I₁ che richiede 4 colpi di clock, una volta a regime avrà un'istruzione completa ogni colpo di clock.

Questo comporta anche che si passa da un modello sequenziale ad un modello in cui le istruzioni si sovrappongono (il che crea dei problemi che vedremo come risolvere).

Nel caso ideale in cui riesco a suddividere le istruzioni in 4 fasi ogniuna assegnate ad un modulo a ogni modulo lavora ad una frequenza ben definita e senza intoppi, il throughput sarà:
 $Throughput_{pipeline} = Throughput_{senza} * n^{\circ} \text{ stadi};$

NB: la tipologia del processore è Register-Register/Load-Store.

Ma questo è un caso ideale ed è l'estremo superiore del vantaggio che ci potrà dare l'uso delle pipeline.

L'idea che viene qui seguita è partire da un processore senza pipeline fino ad evolvere verso quello con pipeline. Il primo caso che vedremo è proprio quello di un processore in cui non c'è pipeline ma l'istruzione viene comunque sottoposta ad alcune fasi in questo caso 5, che sono:

- Fetch: caricamento dell'istruzione (cache/memoria)
- Decodifica: Decodifica l'istruzione e reperisce gli operandi
- Esecuzione: il processore nonostante non abbia pipeline è orientato ai RISC ed è di tipo Load/Store ovvero solo 2 operazioni lavorano in memoria mentre tutto il resto lavora sui registri. Ecco quindi perché l'istruzione rispecchia quello di un processore RISC. In questa fase viene richiesta alle ALU il calcolo dell'indirizzo effettivo, per le operazioni di Load and Store.
- Accesso alla memoria e salti: in questa fase le operazioni di Load/Store accedono alla memoria, mentre le istruzioni di salto (che modificano il valore del PC) terminano qui.
- Write-Back: scrittura dei risultati sui registri.

Adesso andiamo a vedere sui dettagli cosa fanno le varie istruzioni nei vari stadi.

Nella fase di Fetch si preleva il PC, accediamo in memoria e copiamo il contenuto nell'Instruction Register, Dopo di che sommiamo 4 al PC e salviamo il valore nell'NPC (Next Program Counter).

$IR \leftarrow MEM[PC]$

$NPC \leftarrow PC + 4$ dove 4 sono byte ovvero 32 bit.

Tutto questo viene fatto in un solo colpo di clock per entrambe.

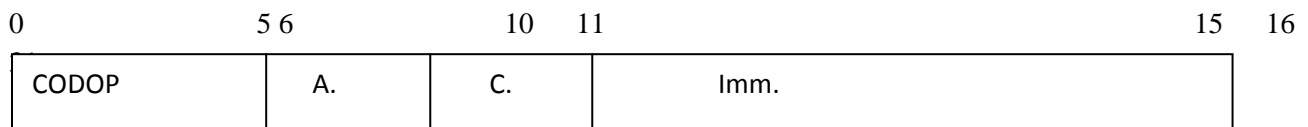
Sommiamo 4 perché abbiamo un processore a parallelismo fisso e del resto è un processore RISC. Nella 1° riga avviene un miracolo: in un solo colpo di clock si viene fornito l'accesso in memoria. Non accade spesso, ma in questo caso abbiamo una cache che è compatibile con i tempi di clock, e che mi fornisca l'istruzione in un solo colpo di clock.

Nota: ci siamo dimenticati di dire che ognuna delle 5 fasi dura 1 colpo di clock, e che il datapath è diviso in 5 pezzi, uno per ogni fase.

ATTENZIONE IL DATAPATH ED IL SUO SCHEMA LOGICO.

Nella seconda fase avviene la Decodifica. Le tre operazioni che vengono svolte non hanno nulla a che fare con la decodifica ma sono delle operazioni di accesso agli operandi. La decodifica delle istruzioni avviene su 32 bit ed è estremamente rigida per semplificare la decodifica stessa. In particolare si suppone che nella generica istruzione aritmetico-logica avremo una struttura in cui nei primi 6 bit c'è il codice operativo, poi ci sono 5 bit per il 1° operando e 5 per il 2° operando (sempre perché ci guardiamo bene nelle istruzioni aritmetico-logiche del fare accesso alla memoria e sfruttiamo i registri, ma uno dei due può anche essere un valore immediato):

Formato dell'istruzione aritmetico-logica



A e C sono indici di registro, quindi con 5 bit avrò 32 registri a disposizione

Ecco quindi le operazioni che vengono effettuare:

$$A \leftarrow \text{Regs}[\text{IR}_{6..10}]$$

$$C \leftarrow \text{Regs}[\text{IR}_{11..15}]$$

$$\text{Imm} \leftarrow ((\text{IR}_{16})^{16} \#\#\text{IR}_{16..31})$$

Ma dovremo chiederci come fa il processore a capire che quelle di cui stiamo parlando è un'istruzione aritmetico-logica?

Qui viene fatta una assunzione, ovvero, il processore assume che quella sia una istruzione AL. Ma se non era un'istruzione AL ed era un'istruzione di salto? Non succede nulla perché avrò caricato nei registri A e B delle info che non hanno senso ma che non uso, quindi non recano danno a nessuno ma se si fosse trattato di un'istruzione AL (e lo capirò solo dopo) mi troverò gli operandi già nei registri. Lo stesso discorso vale per l'immediato: c'è un piccolo particolare, ovvero, l'immediato è su 16 bit mentre il registro è a 32 bit quindi per far quadrare le cose dovrò praticare un'estensione del segno in modo che la prima parte del registro si riempia del bit di segno e nella parte alta del registro ci vada a finire l'immediato intatto. Ovviamente anche in questo caso l'immediato non mi serve, ovvero in un immediato ho informazioni insensate, non lo userò. Come già detto prima.

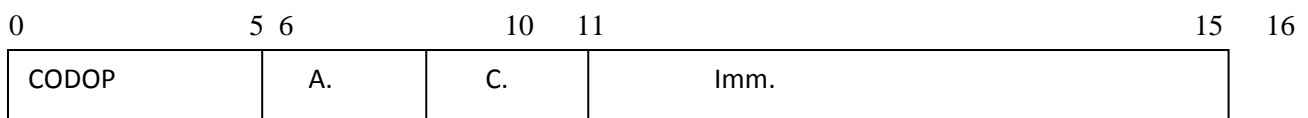
Questa mentalità è molto diffusa e si basa sul presupposto che se quello che faccio è dannoso, lo faccio comunque ed in questo modo semplifico di gran lunga l'hardware e i suoi compiti.

La terza fase è la fase di esecuzione oltre che quella in cui avviene il calcolo dell'indirizzo.

È la più complicata a causa del fatto che le operazioni che devono essere eseguite dipendono fortemente dalla tipologia di istruzione, ovvero, l'unità di controllo a questo punto capisce quale tipo di istruzione ha davanti e produce operazioni differenti a seconda del fatto che l'istruzione sia:

1. Memory Reference: è il tipico caso dell'operazione di Load and Store. Il formato è differente da quello precedentemente visto. Il modo di indirizzamento è quello basato sul displacement.

Load/Store



Viene svolta l'operazione: $\text{ALUOutput} \leftarrow A + \text{Imm}$

Gli indirizzi sono $\text{Imm}(\text{Rx})$ dove Imm è un immediato e Rx è un registro

L'ALUOutput è un registro temporaneo in cui viene memorizzato il risultato dell'operazione eseguita dall'ALU

In B ci sarà la destinazione o la sorgente a seconda che rispettivamente venga eseguita una Store o una Load.

Nota: per le istruzioni Aritmetico-Logiche bisogna fare una attenta distinzione tra il caso in cui sia una operazione tra registro e immediato oppure tra Registro e Registro. Se ne accorge dal CODOP.

2. Istruzioni Aritmetico Logiche

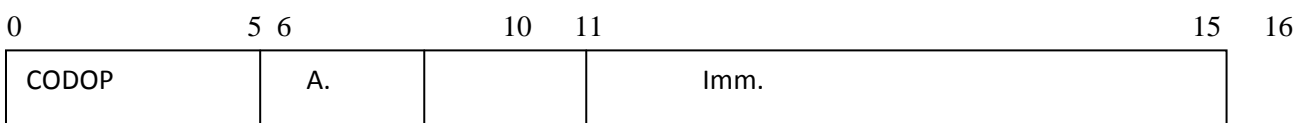
- a. Register-Register $ALUOutput \leftarrow A \text{ op } B$
- b. Register-Immediate $ALUOutput \leftarrow A \text{ op } Imm$

Pilotando il MUX in base a che succede nella ALU farò arrivare ad uno il valore di Imm o di B a seconda dell'istruzione.

3. Branch

Sono di due tipi come già detto, ovvero condizionati oppure incondizionati. Se sono incondizionati si assume che al posto di Imm ci sia un offset da sommare al PC (dove si prende il valore di NPC ovvero quello incrementato).

Quindi il formato è:



(Jump Format)

Il risultato andrà a finire dentro al PC e questo vuol dire che l'indirizzo a cui saltare viene calcolato in questa fase. L'ultima istruzione fa riferimento ai salti condizionati (si assume che le condizioni siano contenuta in A e ne nostro caso ci può essere un'unica condizione che è il valore di A

$$ALUOutput \leftarrow NPC \text{ (Pilotato MUX alto)} + Imm \text{ (Pilotato Mux alto)}$$

Nota: Cond è un flip-flop $Cond \leftarrow (A \text{ op } 0)$

Il salto si verifica soltanto se l'operando contenuto in A vale 0.

Il risultato della condizione viene memorizzato in Cond (che è un flip flop) che va in ingresso al MUX in alto a destra (che lavora al colpo di clock successivo e, pilotato il MUX, il PC viene o meno settato all'istruzione a cui saltiamo).

NB: Il 4° Stadio vede lavorare solo le istruzioni Load/Store e i salti.

Vediamo i salti: si va a vedere il valore contenuto in COND. (PEZZO MANCANTE)

Nella Load/Store avviene l'accesso in memoria, in ALUOutput c'è l'indirizzo di memoria (cache dati) a cui voglio accedere e il valore che mi ritorna la cache dati lo carico (load) nel LMD mentre se devo scrivere (Store) scriverò in memoria il valore contenuto in B. Se c'è un cache miss non avrò più un solo colpo di clock ma ne avrò qualcuno in più. Ecco quindi che le operazioni saranno:

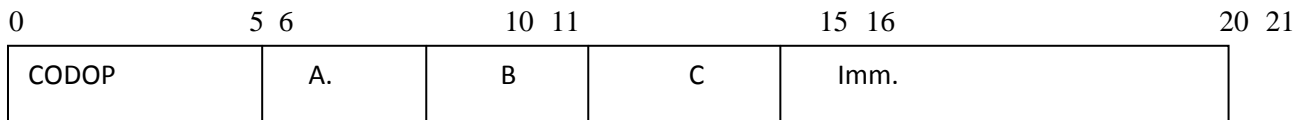
- Memory Reference:
 - Load $LMD \leftarrow Mem[ALUOutput]$
 - Store $MEM[ALUOutput] \leftarrow B$
- Branch
 - If(cond) $PC \leftarrow ALUOutput$ else $PC \leftarrow NPC$;

La quinta fase è quella di Write-back. Nel caso della Load viene fatto un accesso al registro

$$Regs[IR_{11..15}] \leftarrow LMD \quad LMD \text{ è un reg tmp in cui abbiamo memorizzato il valore della fase 4}$$

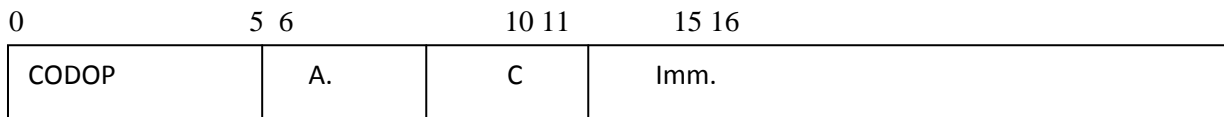
Qualora l'operazione sia register-register il formato sarà il seguente

$$Regs[IR_{16..20}] \leftarrow ALUOutput$$



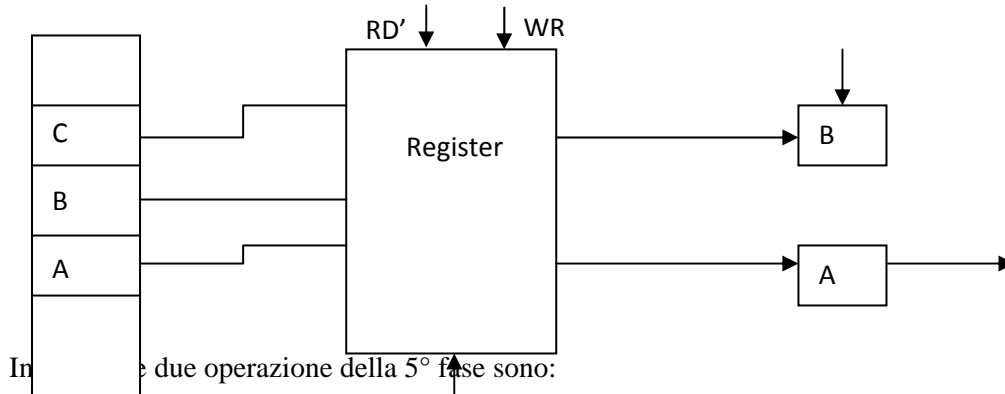
Operazione Aritmetica Reg-Reg

$\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput}$



Reg-Imm

Ecco quindi perchè bisogna distinguere I casi. E' comunque in C che vien scritto il risultato dell'operazione eventualmente eseguita nei passi precedenti.



$\text{Regs}[\text{IR}_{16..20}] \leftarrow \text{ALUOutput}$

$\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput}$

Nota: per il processore appena descritto fin qui alcune operazioni non sfruttano il 4° o il 5° stadio ma le consideriamo costituite da 5 colpi di clock.

A questo punto vediamo come trasformare il processore che abbiamo presentato in un processore con PIPELINE

Bisogna che:

- Ad ogni colpo di clock cominci il fetch di una nuova istruzione. E' chiaro che se facciamo questo bisogna che ci siano delle barriere che separino i vari stadi per permettere che ogni stadio lavori su ciò che gli compete senza mescolare tutto. Questo si realizza facendo in modo che ci siano dei registri a separare i vari stadi, in cui lo stadio a monte scrive il valore ad ogni colpo di clock affinché ciò che serve allo stadio a valle quest'ultimo se lo vada a prendere dal registro. I registri di pipeline sono tutti cloccati e sincronizzati con lo stesso clock.
- Ci sono alcune risorse che nella versione senza pipeline funzionano in fasi diverse. Un esempio tipico è il Register File, che adesso nel generico colpo di clock lavorerà in due stadi differenti (due accessi in lettura ed uno in scrittura). Altra componente di questo tipo è la cache; la soluzione la si trova avendo una cache di tipo dual-port, o addirittura due cache separate.

- Altro problema è legato ai salti

Fatte queste considerazioni lo schema finale si complica un po se tutto funziona bene, la pipeline mi permette di:

- Moltiplicare per N (nel caso =5) le prestazioni
- Nel passare delle versioni senza alla versione con pipeline il clock rallenta
- Il numero degli stati di pipeline è limitato da
 1. Necessità di bilanciare gli stadi (stessa durata)
 2. Pipelining overhead

Pipeline Hazard

Sono situazioni in cui la pipeline non sputa nulla fuori, e si possono distinguere in:

- Structural Hazard: sono causati dal conflitto di risorse
- Data Hazard: un'istruzione necessita di un risultato elaborato dall'istruzione che la precede
- Control Hazard: dipendono dai salti condizionati nella pipeline

I più semplici da capire sono gli Hazard strutturali che sono dovuti al fatto che uno stadio non riesce a completare in un tempo standard l'operazione e richiede più tempo. Oppute può accadere che al Register File venga richiesto un numero elevato di operazioni da svolgere. Altro caso in cui questo accade è nel caso di cache miss.

Cosa succede? La pipeline va in STALLO, e quando un'istruzione stalla:

- L'istruzione che parte dopo quella stallata andrà anch'essa in stallo
- L'istruzione che parte prima di quella stallata continuerà a lavorare.

Uno stallo implica che nei colpi di clock successivi a quello in cui si verifica lo stallo nella pipeline si formerà una bolla che non mi fa uscire nulla. Ad esempio uno stallo si può verificare quando il fetch richiede molto più tempo del dovuto, le istruzioni che vengono rallentate sono quelle che seguono all'istruzione che ha richiesto più colpi di clock.

Come risolviamo il problema degli Hazard Strutturali? Semplice dovremo investire in termini di hardware affinché si possano coprire i soliti problemi, ovvero aggiungendo porte al register file etc.

Gli Hazard di Dato sono dovuti al fatto che le istruzioni sono parzialmente sovrapposte in fase di esecuzione.

Vediamo un esempio:

ASSUNZIONE: il 1° operando è quello destinazione, siano richieste le seguenti operazioni:

DADD R1,R2,R3

DSUB R4,R5,R1

AND R6,R1,R7

OR R8,R1,R9

XOR R10,R1,R11

Queste istruzioni apparentemente innocenti se messe nella pipeline provocano l'emissione di un risultato errato!!!

Caso ancora più critico è quello del comportamento non deterministico, determinato dal meccanismo degli interrupt (anche un processore RISC deve servire gli interrupt!). A seconda di dove arriva l'interrupt il processore può fornire un risultato errato. Inoltre almeno per come è fatta la pipeline che abbiamo visto solo nel 4° stadio può sovraccaricare la memoria e nel 5° modificare i registri: nasce quindi un problema di interruzione della pipeline.

Ci sono due modi per gestire gli Hazard di Dato:

- A bordo del processore c'è una logica che si accorge quando si verifica un hazard di dato e manda in stallo la pipeline: questo è un metodo brutale ma che garantisce la correttezza delle operazioni, facendo degradare le prestazioni
- E' solo una soluzione parziale ed è nota come FORWARDING. Il concetto è molto semplice: il problema del data Hazard nasce dal fatto che la ALU non ha ancora il valore corretto nel registro. Allora faccio in modo che l'istruzione che ha bisogno di quel valore non lo prenda dal registro ma lo ricava dal registro a valle dell'ALU, ecco perché si chiama forwarding. Ciò di cui abbiamo bisogno è della logica che riconosce l'Hazard. Soprattutto a seconda di dove mi trovo dovrò andare a prendere il valore al posto giusto e bisognerà fare attenzione a questo. Il data forwarding non risolve tutti i problemi di data hazard. Occorre che sia disponibile in tutti gli stadi e a volte risulta necessario mandare comunque in stallo la pipeline.

Nella soluzione ibrida, la logica di controllo deve potersi accorgere dei Data Hazard facendo laddove sensato e possibile con il forwarding, mentre in altri casi con lo stallo della pipeline.

Il problema degli data hazard che abbiamo finora visto verificarsi soltanto con registri, possono in realtà verificarsi anche con celle di memoria, nei seguenti casi:

- Gli accessi in memoria fra load e la store non vengono fatti nello stesso stadio
- Se una load fa un Miss e la store fa un cache miss, è possibile che la politica di gestione dei cache miss porti a non avere un dato corretto letto dalla load (data hazard). Ricorda: la LOAD e la STORE operano sulla cache.
- Da notare è che questo problema o caso si verifica solo nel caso in cui la cache funziona in modalità write back anziché in modalità write through. Nella prima il dato viene trasferito in RAM quando possibile, mentre nella seconda viene trasferito in corso d'opera.

L'ultima categoria di Hazard è quella dei control Hazard, che sono dovuti ai salti. Condizionati: mentre valuto la condizione la pipeline continuerà a caricare istruzioni, ma se alla fine scopre che devo fare il salto dovrò svuotare la pipeline. Inoltre, è anche probabile che siano state effettuate operazioni di scrittura, il che mi darebbe molto fastidio.

I control hazard creano un degradamento delle prestazioni molto più forte rispetto ai data hazard, perché causano lo svuotamento della pipeline.

Prima soluzione: quando trovo un salto, blocco la pipeline andando ad inibire l'eventuale caricamento di altre istruzioni. Nel frattempo l'istruzione di salto arriva al punto in cui scopro dove devo saltare e quindi la pipeline riparte. Soluzione logica, ma INEFFICIENTE. Ci sono altre 3 soluzioni:

1. Predict Untaken: si lascia che la pipeline prosegua e nel caso in cui il salto deve essere fatto si svuota la pipeline e si effettua il salto, altrimenti si continua. L'operazione di svuotamento può essere realizzata in due modi diversi:

- a. Si impedisce che lo stadio successivo lavori, ovvero si manda la decodifica in stallo al colpo di clock successivo al fetch dell'istruzione sbagliata
- b. Modifichiamo un'istruzione trasformandola in una NOP (o reset)

Bisogna anticipare il prima possibile lo stadio in cui si capisce se il salto deve essere effettuato o meno. Nessuno ci vieta di anticipare il calcolo della condizione di salto allo stadio di decodifica.

Con la previsione *Untaken* si assume che il salto non sia preso, per cui se sarà preso svuoterò la pipeline.

2. Il *Predict Taken*: è un po più problematico ed in conseguenza di ciò non largamente usato, poiché si assume che la condizione di salto sia verificata.

Nota: il compilatore se furbo può fare la differenza!

3. *Delayed branch*: l'implementazione di questa tecnica consiste nell'accettare che dopo il salto esiste sempre un'istruzione che venga eseguita comunque; nel momento in cui scopre che il salto non doveva essere fatto allora va bene (ma se il salto andava fatto non fa nulla sulla pipeline ma si anticipa l'esecuzione dell'operazione) e sarà il compilatore a farsi carico di andare a cercare un'istruzione che vada eseguita comunque (se non le trova, 30% dei casi mette una NOP).

Implementing a pipelined processor

Il processore di cui ci occupiamo ha le seguenti caratteristiche:

- È un processore Load/Store
- Le istruzioni di salto condizionato fanno il confronto di un registro con zero
- Ci sono istruzioni ALU intere
- Il famoso CPI è uguale a 5

Nel passare dalla versione senza pipeline a quella con pipeline abbiamo inserito dei registri di pipeline tra ogni stadio a monte e il suo stadio a valle. Questi registri hanno inglobato alcuni registri precedentemente esistenti ed inoltre sono fondamentali per disaccoppiare gli stadi e renderli indipendenti congelando il risultato nel registro di pipeline per lo stadio a valle cosicché al successivo colpo di clock lo stadio a monte possa cominciare a lavorare su una nuova istruzione.

Approfondiamo la questione su i controlli degli hazard, cominciando su quelli di dato e quelli strutturali. In particolare quello di dato si possono riscontrare in fase di decodifica: ma è da dire che le istruzioni hanno un comportamento prevedibile (ovvero se produce un risultato o richiede un dato), ed inoltre alla fine dello stadio ID possiamo dire se c'è bisogno del data forwarding oppure se c'è bisogno di mandare in stallo la pipeline. Per semplificarci la vita, quando ci accorgiamo di non poter attuare il data forwarding si manda in stallo la pipeline alla fine dello stadio ID ovvero prima che essa possa entrare nello stadio di EX (tale comportamento è chiamato *ISSUE*).

Vediamo la dipendenza che richiede lo stallo

LD R1, 45(R2)

DADD R5,R1,R7

DSUB R8,R6,R7

OR R9,R6,R7 ben 2 Stalli!!!

Sono operazioni ALU invece che producono un risultato a valle dello stadio di EXECUTE, saltano il MEM e scrivono nello stadio di WB.

Deve essere possibile confrontare il contenuto del registro che ha l'indice dell'input dell'istruzione appena codificata con l'indice del registro contenente l'output. Questo richiede l'utilizzo di operatori.

Lo stesso discorso lo possiamo fare con il forwarding, ad esempio:

```
LD      R1, 45(R2)      F D E M W
DADD   R5,R6,R7         F D E M W
DSUB   R8,R1,R7         F st E M W
OR     R9,R6,R7         F D E M W
```

R1 viene prodotto in LMD solo a valle dello stadio di MEM.

Bisogna confrontare l'indice del registro di input che sta per andare in esecuzione con l'indice dei registri di output dei due stadi che stanno a valle (ok, va bene per le operazioni intere).

Ho due soluzioni per implementare lo stallo:

- Si forza nel registro tra la D e la E tutti zeri forzando la NOP, dato che nel registro di pipeline è contenuto il registro di cod. op. e la NOP ha tutti zeri.
- Si forza il registro di F/D affinché mantengano il valore, il che si traduce nel far rifare ad ogni stadio l'operazione eseguita al colpo di clock precedente.

Per quanto riguarda invece il data forwarding:

può essere implementato collegando ALUOutput o MemoryOutput con ALUInput, MemoryInput e la zero detection unit

La logica da dover poter confrontare:

Il registro di destinazione contenuto in EX/MEM e MEM/WB (che sono i registri di pipeline di cui abbiamo precedentemente parlato) con i registri sorgenti contenuti in ID/EXm ed EX/MEM.

Già la scorsa volta avevamo visto come comportarci nei confronti delle istruzioni di salto condizionato, che implicavano che magari le istruzioni successive non dovevano essere eseguita. Il n° di queste istruzioni da scartare dipende da quando capiamo che quella è un'istruzione di salto e se vada eseguita o meno, e dal valore del program counter calcolato qualora si decide che il salto vada fatto:

```
BEQ  RX, ind
```

I1

I2

I3

Il n° delle istruzioni dipendono da come è strutturata la BEQ. Se il branch delayed slot è pari a 2, allora dovrò buttare via due istruzioni!

Per far sì che il problema sia minimo, si implementa la seguente soluzione:

- Anticipiamo nel 2° stadio il modulo di comparazione con 0, e sfruttando le regolarità della codifica delle istruzioni prendo comunque gli operandi ovvero RX lo mando in input al register file cosicché la condizione venga calcolata. Inoltre prima nello stadio 3 consideravo la condizione di stallo e poi caricavo o meno nel program counter il mio indirizzo (si preleva il valore del nuovo pc, ovvero oldval+4, sommandogli un immediato esteso su 32 bit): adesso anticipo tutto al 2° stadio. Entra il modulo ADD nel 2° stadio: il suo output è l'indirizzo in cui saltare che :
 - Nel caso di salto condizionato viene preso in considerazione solo nel caso in cui il salto vada fatto
 - Nel caso di salto incondizionato verrà preso in considerazione comunque.

Anticipando tutto al 2° stadio ho ottenuto di aver ridotto il branch delayed slot ad 1, quindi le istruzioni che dovrò buttar via sul caso in cui il salto non vada fatte è 1.

EXCEPTIONS

Le eccezioni sono tutti quegli eventi che interrompono il flusso di esecuzione del programma, annoverabili come interrupt, divisione per zero, page fault, o anche il fatto che sul BUS si sia verificato un errore.

Sicuramente esistono le seguenti tipologie di exception:

- I/O device request: scatenata da un dispositivo esterno
- System call: ed es. INT21H
- Debug: attivazione del trace mode
- Operazioni di overflow o under flow
- Eccezioni scatenate dell'unità FLOATING-POINT
- Page fault
- Se il processore può effettuare solo accessi allineati e tenuto a fare accesso non allineato
- Se il programma tenta di fare accesso ad una locazione di memoria riservata
- Istruzione illegale
- Malfunzionamento hardware
- Alimentazione mancante

Queste eccezione possono essere classificate in diversi modi:

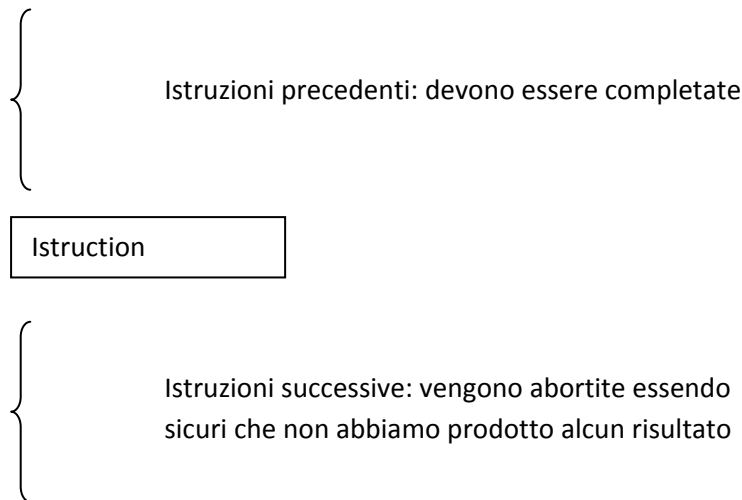
- Asincrone (tipicamente da dispositivi esterni, ma anche i guasti) oppure sincrone (ad es. INT21H da parte dell'utente che sappiamo verificarsi in corrispondenza della istruzione)
- Quelle richieste dall'utente contro quelle non richieste ad es. page fault
- Mascherabili e non mascherabili
- Alcune sono allineate alle istruzioni, altre invece si possono verificare in qualunque degli stadi di un'istruzione
- Alcune che ritornano il controllo al programma chiamante, oltre che invece ritornano al sistema operativo.

Cosa succede quando arriva un eccezione all'interno di un processore che sia dotato di pipeline? Vediamo: saremmo tentati di dire di completare le istruzioni prima di quella che ha generato l'eccezione, si abortiscono quelle successive a quella che ha causato l'eccezione e si salta alla routine di servizio per l'eccezione sollevata.

In tutto questo ragionamento è fondamentale tenere presente 2 punti focali:

- Bisogna salvare il contesto per poi poter ritornare correttamente.
- Bisogna che le istruzioni che vengono abortite non devono avere prodotto alcun risultato.

Vediamo di capire meglio:



Questo lo ottengo se concentro negli stadi finali tutte le operazioni di scrittura (ovviamente non sui registri ma in memoria). Se il meccanismo di gestione delle eccezioni è PRECISO, il tutto è garantito come anche i due punti messi precedentemente in evidenza.

Proviamo a capire i problemi legati alla gestione delle eccezioni capendo quali sono le possibili fonti di eccezione:

- IF
 - page fault sul fetch di una istruzione
 - accesso non allineato
 - violazione della memoria legato a privilegi di accesso
- ID: Codice operativo di un istruzione indefinito o illegale
- EX: eccezioni aritmetiche
- MEM: stesse di IF
- WB: non ci possono essere eccezioni, il che rende possibile il meccanismo di gestione delle eccezioni precise.

Il caso più bastardo in assoluto è quello in cui nello stesso colpo di clock si verificano due eccezioni: sarebbe opportuno gestire le eccezioni implementando un meccanismo di priorità.

LD F D E M W

DADD F D E M W

Un data page fault potrebbe verificarsi nello stadio di mem della LD, ed un'eccezione di tipo aritmetico potrebbe scatenarsi nello stadio di EX della ADD: sarebbe opportuno che l'exception

handler eseguisse prima la procedura corrispondente al page fault sulla LD, e solo dopo il ritorno di questa gestisca l'eccezione di tipo aritmetico sulla DADD nello stadio di EX.

Poi però ci sono dei casi più perversi in cui le eccezioni si verificano in ordine inverso rispetto alle istruzioni cui competono.

Una possibile soluzione è la seguente:

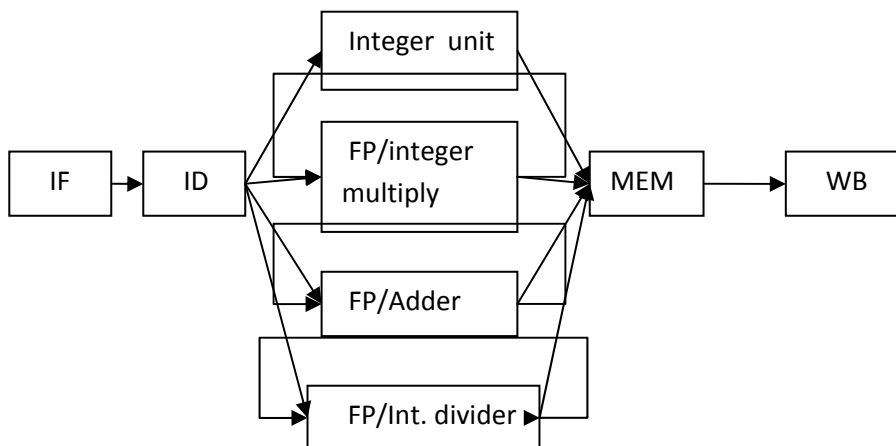
- Ad ogni istruzione associo un flag che mi dice se l'istruzione ha scatenato un'eccezione o meno
- Se l'istruzione ha scatenato un'eccezione, non deve arrivare al fondo, inibendo ogni possibile operazione di scrittura, il che mi inibisce tutte le istruzioni successive a quelle che ha generato l'eccezione
- Aspetto che termini l'esecuzione delle istruzioni precedenti a quella che ha generato l'eccezione, dopodiché servo l'eccezione

Per poter abortire le istruzioni, come richiesto, senza che effettuiamo modifiche in memoria, devo essere sicuro che tutte le operazioni di scrittura avvengano negli ultimi stadi (nel nostro caso quindi 4° e 5°). Quindi per poter gestire le eccezioni abbiamo bisogno di processori con un instruction set opportuno: ad esempio non potrei mai permettermi di fare questo ragionamento di gestione delle eccezioni con un processore di tipo CISC perché quelli effettuano operazioni di scrittura anche nello stadio di Execute, il che fa cadere tutte le garanzie che avevamo dato.

Floating point operations

L'aver dentro un'unità floating-point impone il vincolo che una istruzione non viene più processata in un colpo di clock. Una soluzione sarebbe abbassare il clock, ma questo mi farebbe decadere le prestazioni perché abbasserei la velocità di tutta la pipeline.

Un esempio pratico è il seguente:



Si può pensare che due o più istruzioni che usano l'unità floating-point possano essere eseguite parallelamente ed indipendentemente. Ciò mi evita di mandare in stallo tutta la pipeline, ma mi crea un altro problema: le istruzioni non mi arrivano nello stadio di MEME in ordine di arrivo ma indipendentemente dal tempo che ci mettono ad essere eseguite. Introduciamo un po di terminologia, rimandando il problema a dopo:

- Intervallo di inizio: quanti colpi di clock devono passare tra quello in cui spedisco un'istruzione e quello in cui spedisco un'istruzione dello stesso tipo alla stessa unità.
- Latenza: è l'intervallo richiesto per il completamento di una certa operazione. In realtà, è il numero di colpi di clock tra la richiesta di un'operazione e il momento in cui i risultati vengono resi disponibili per le prossime operazioni.

Dobbiamo pensare che per un'istruzione FP, all'interno dello stadio di EX ci sia un'altra pipeline in cui eventuali operazioni che possano essere eseguite in cascata e parallelamente vengano eseguite come in una pipe. Questo è possibile ad esempio nel caso di una somma, ma non della divisione!!! Stesso discorso delle somme lo si può fare nel caso della moltiplicazione.

La latenza e l'intervallo di inizio permettono di caratterizzare il singolo stadio rispetto all'operazione da eseguire.

Eravamo rimasti al fatto che i moduli FP incaricati della somma e della moltiplicazione potevano parallelizzare le esecuzioni, sfruttando un opportuno tipo di pipeline. Per quanto riguarda invece la divisione FP, ha tutte le proprietà negative che possiamo immaginare. L'unità di EX della nostra pipeline quindi diventa l'unione di ben 4 moduli

- Integer unit
- FP/Integer Multiply
- FP Adder
- FP/Integer Divider

Grazie a questa nuova situazione, ci sono buone probabilità di non mandare in stallo la pipeline. Diciamo che un caso in cui si può verificare lo stallo è se mi arrivano due divisioni di file, perché sappiamo che quell'operazione non può essere parallelizzata. Nello stato di EX ci potranno essere fino a 4 istruzioni, il che è un vantaggio.

Vengono comunque introdotti alcuni fattori negativi:

- L'ordine di completamento delle istruzioni non dipende dall'ordine in cui vengono scritte le operazioni
- Risulta più complicato mantenere un meccanismo di gestione precisa delle eccezioni.

Gli hazard strutturali aumentano per 2 ragioni:

1. Dovuti al fatto che la divisione FP richiede 25 colpi di clock; se arriva subito dopo un'altra istruzione di divisione FP ecco che questa si fermerà allo stadio ID bloccando la pipeline.
2. Dovuti al fatto che non è più determinato a-priori l'ordine con cui le istruzioni arrivano nello stadio di MEM/WB. Fin quando si è nello stadio di MEME, solo la Load fa qualcosa lì, ma quando vanno tutte nello stadio di WB e tentano tutte e 3 di scrivere ecco che risulta meccanico mandare in stallo la pipe permettendo una scrittura per volta.

SOLUZIONI AL PROBLEMA

1. Nel caso in cui ci sia da realizzare uno stallo dobbiamo decidere se:
 - Bloccare anticipatamente le istruzioni che sappiamo generino uno stallo
 - Bloccare l'istruzione un attimo prima che lo stallo si verifichi

Da notare è che a seconda di dove si sceglie di bloccare l'istruzione gli effetti ottenuti saranno diversi. Ecco perché la prima soluzione è quella preferibile.

2. In alternativa, per risolvere il problema si può modificare l'hardware parallelizzandolo per permettere più scritture contemporaneamente. Diciamo che questa opzione è troppo costosa.

Le dipendenze di dato per cui un'istruzione attende il risultato di un'operazione precedente può richiedere uno stallo superiore al solo colpo di clock che abbiamo sempre preso in considerazione. Il fatto che il 3° stadio gode adesso di un parallelismo, vede l'introduzione di nuovi tipi di Hazard. Una volta visti i RAW, particolare attenzione vogliono i WAW (write after write): in questo caso la soluzione sta nel controllare prima di far accedere allo stadio di EX un'istruzione dobbiamo controllare che questa non tenti di scrivere sullo stesso registro su cui sta già scrivendo una istruzione si trova già sullo stadio di EX. In questo caso dovrei mandare in stallo nello stadio di MEM l'istruzione che è terminata in anticipo rispetto al flusso di codice: l'obiettivo di questa tecnica è far sì che le operazioni di struttura mantengano l'ordine che hanno nel codice!!! Cio detto, diventa un po complicato garantire il meccanismo della precise exceptions, Vediamo questa porzione di codice:

DIV.D F0,F2,F4

ADD.D F10,F10,F8

SUB.D F12,F12,F14

Nota: sia la ADD che la SUB possono finire prima della DIV. Qualora la SUB generi un'eccezione prima che la DIV termini avrà un'eccezione imprecisa

Se è invece la DIV a generare un'eccezione questa potrebbe verificarsi dopo che le istruzioni ADD e SUB hanno già scritto il loro risultato.

La soluzione è che io accetto che l'istruzione si completi solo dopo che tutte le altre si sono completate. Mi va bene che le operazioni vadano avanti un po come vogliono, ma in fase di scrittura dovrò bufferizzare permettendo una scrittura ordinata dei dati nelle apposite locazioni e nell'ordine in cui devono essere fatte. Le istruzioni vengono congelate un attimo prima di scrivere, permettendo che queste scrivano in un ordine corretto i dati. Devo fare attenzione comunque a non bloccare tutte le altre istruzioni congelando quelle in esame per ottenere una scrittura ordinata.

MIPS R4000 PIPELINE

E' una pipeline particolare perché ha 8 stadi, quindi è un po più lunga di quelle che abbiamo finora visto.

I due stadi che si sono allungati per l'accesso in memoria possono comunque essere messi in pipeline e parallelizzate, Avendo due cache ma per le istruzioni ed un'altra per i dati. Come conseguenza della struttura della pipeline cambia anche il branch delay slot. Prima eravamo riusciti, anticipando il calcolo dell'istruzione di salto e la condizione di salto al secondo stadio, a ridurre il branch delay slot a 1.

Qui invece:

- Sono richiesti più forwarding
- Si incrementa il load delay a 2
- Si incrementa il branch delay a 3

Questo significa che nel caso di salto preso butterà via 3 istruzioni. Il load delay slot è il n° di istruzioni che dovrò attendere prima che un'istruzione possa usare un certo dato che ho calcolato. Ad esempio:

ADD R2,R0,R4

1. Non posso leggere o usare R2 (stallo)
2. Non posso leggere o usare R2 (Forwarding)
3. Ok qui si...

Load delay slot = 2.

Istrucion Level Parallelism

Abbiamo visto il ruolo determinante che il compilatore ha rivestito nel caso dei processori RISC per il riempimento del branch delay slot. Il compilatore può rivestire un ruolo importante non solo in quel caso, ma anche in altri casi. Abbiamo detto che il load delay slot è il n° di colpi di clock che deve attendere un'istruzione prima che il risultato possa essere usato. Ad esempio nel caso dell'istruzione seguente

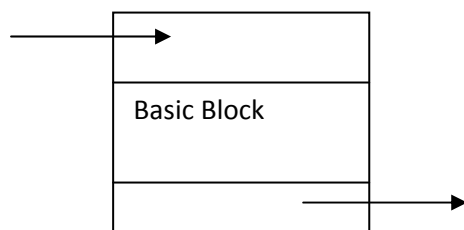
ADD R2,R1,R3 I

Il valore di R2 non sarà disponibile se non alla fine di EX quando il risultato sarà prodotto dalla ALU

Posziamo essere più o meno ottimisti sul fatto che il compilatore possa fare tutto questo, ma bisogna comunque essere realisti. Ciò che un compilatore può fare è operare nel basic block, che è un blocco di istruzioni che viene eseguito sempre tutte insieme. Le istruzioni all'interno del basic block sono quelle su cui possiamo giocare liberamente.

Il basic block è una sequenza di istruzioni che:

- Non può avere branch al suo interno, ad eccezione dell'ingresso
- Non può avere branch al suo interno, ad eccezione dell'uscita



Quindi il principale limite sul fatto che il compilatore possa ottimizzare il codice riordinando le istruzioni per andare bene su una pipeline sta nella dimensione dei basic block che è di 4 o 5 istruzioni al massimo.

Ci sono 2 processi che tentano di far esplodere l'istruzione level parallelism:

- Dinamiche: il meccanismo con cui viene sfruttato il parallelismo all'interno del codice è inglobato nell'hw stesso che riordina il codice

- Statiche: è il processore che deve ottimizzare il codice andando a riordinare le operazioni per come meglio ritiene.

L'approccio statico è ancora utilizzato per architetture di tipo embedded.

Infatti, l'ottimizzazione a carico del compilatore richiede anche tempo. Nelle applicazioni embedde sono ancora utilizzati processori RISC su architetture non superscalari che sfruttano approcci di tipo statico. Il parallelismo a livello di basic block non mi fa stare così sereno, quindi mi trovo davanti a un bivio:

- Si può abbandonare l'approccio statico sui basic-block
- Si può cercare di estendere quella che è la tecnica basata sui basic block sfruttando cioè che c'è di parallelo da eseguire.

Vediamo quindi le diverse alternative. Prendiamo in esame il seguente codice:

```
for(i=0; i<1000;i++)
    x[i]=x[i]+y[i];
```

Non è che si possa giocare molto sulle istruzioni ASM che codificano le operazioni ma guardando il loop posso modificare un po' le cose al fine di parallelizzare le operazioni: è questo il caso del loop unrolling (srotolamento del ciclo). Posso ad esempio modificare il ciclo facendo sì che il ciclo venga ripetuto n/4 volte:

```
for(i=0; i<n/4K i+=4){
    x[i]=x[i]+y[i];
    x[i+1]=x[i+1]+y[i+1];
    x[i+2]=x[i+2]+y[i+2];
    x[i+3]=x[i+3]+y[i+3];
}
```

Potremmo pensare: ben, ma che c'ho guadagnato? C'ho guadagnato che il corpo del basic block è 4 volte più grande di prima, quindi ho molte più istruzioni (dal minimo di 16 fino a più) su cui potrei giocare per eliminare gli stalli, ma soprattutto le istruzioni sono indipendenti le une dalle altre il che mi permette di parallelizzare quadagno in termini di parallelismo.

Secondo vantaggio: il nostro scopo era effettuare la somma di due vettori di 1000 elementi ciascuno quindi in termini di colpi di clock mi interessa l'operazione ADDm perché tutto il resto ovvero le load e le altre sono delle operazioni accessorie. Con il loop unrolling riesco a sommare nel minor numero di colpi di clock!! Il limite di questo approccio l'unico di fatto è la memoria occupata del codice.

Ci sono altri approcci che sfruttano il parallelismo. Ad un certo punto dell'evoluzione si parlò del vector processor, che erano processori aventi blocchi consistenti di registri e delle ALU (ce n'era una schiera infatti e non una sola) che potevano effettuare le operazioni direttamente sui vettori. Ovviamente questi processori caricavano (nel limite possibile) interi vettori nei registri, ed avevano delle ALU che come operandi riuscivano ad operare anche direttamente sui vettori! L'idea è quella di poter processare in parallelo non singoli operandi, ma vettori! L'unico difetto di questo approccio era il costo, a causa dell'enorme quantità di HW richiesto (ecco perché questo approccio fu praticamente sfruttato nell'ambito militare). Da notare è che la tecnologia MMX non si discosta molto da questo approccio!

Abbiamo detto che il compilatore finché possa ottimizzare il codice dovrà opportunamente spostare le istruzioni per evitare che si verifichino hazard. Utile a questo punto è capire le differenze tra DIPENDENZE ed HAZARD.

Una dipendenza tra due istruzioni è un legame tra due istruzioni è differente dal concetto di Hazard. L'ideale sarebbe avere tutte le istruzioni tra loro indipendenti, ma poiché è difficile che ciò si verifichi dobbiamo tener conto delle dipendenze al fine di ottimizzare. L'Hazard è la situazione in cui rischiamo di mandare in stallo la pipeline: ovviamente l'hazard dipende o deriva dalle dipendenze ma tengono conto di come è fatto il processore. Mi spiego meglio: le dipendenze possono essere cause di Hazard, e gli hazard dipendono non solo dal codice ma anche dal processore (e quindi dal fatto che il codice venga eseguito da un certo processore o meno).

- Dipendenze di dato: può essere diretta o indiretta (per proprietà transitiva A dipende da B, da cui a sua volta dipende da C). Si verifica quando un'istruzione produce un risultato che un'altra usa. Per il compilatore è estremamente semplice rilevare le dipendenze di dato che passano attraverso i registri, mentre è praticamente impossibile rilevare le dipendenze di dato che riguardano due celle di memoria: questo è un limite forte per le tecniche statiche. La tecnica dinamica invece lavorano runtime (a tempo di esecuzione) e quindi possono evitare questo fastidioso limite.
- Dipendenza di nome: si verifica quando due istruzioni fanno riferimento allo stesso registro p alla stessa locazione di memoria ma non c'è un flusso di dati associato al nome. Si distinguono in :
 - Dipendenza di output
 - Auto dipendenza

In pratica sono due istruzioni che si danno fastidio tra loro se invertite senza un criterio che mantenga il risultato.

Le dipendenze di nome possono essere eliminate e quindi non creare hazardo o risultati sbagliati andando a cambiare i registri, e se questa operazione è fatta con un po di criterio il tutto va a buon fine.

Questa operazione può essere fatta :

- Dal compilatore che sa bene quali sono i registri a loro disposizione e può gestirli come meglio crede
- Dinamicamente a tempo di esecuzione (runtime)

Attenzione che queste operazioni, sia nel caso di dipendenza di output che di auto dipendenza le operazioni vanno fatte in modo molto accurato. Così come è successo per le dipendenze di dato sul fatto che il problema si verifica sia sui registri che su celle di memoria, anche qui si pone il problema. Il fatto è che il memory renaming un po più complicato infatti:

- Devo trovarmi una cella di memoria libera su cui andare a memorizzare
- Ma devo tener conto del fatto che mentre l'accesso ai registri è controllato, l'accesso alla memoria no (ad es. il DMA può fare accesso modificando una cella che io ho scelto).

A questo punto abbiamo già detto che gli hazard di dato si possono distinguere in ben 3 categorie:

1. RAW: dovuta alla sovrapposizione delle istruzioni all'interno della pipeline, tipicamente per l'esistenza di una dipendenza di dato
2. WAW: si possono verificare quando un'istruzione termina prima o in contemporanea ad un'istruzione che la precorre: dipendenze di output
3. WAR: derivano dalle auto dipendenze

Vediamo più in dettaglio gli hazard di tipo Write after read .Come già detto si verificano in corrispondenza di auto dipendenze. In particolare sono molto ricorrenti

nel caso di processori con pipeline in cui tutte le letture vengono fatte relativamente presto e tutte le scritture relativamente tardi.

- Dipendenze di controllo: si verificano quando una istruzione dipende da un salto condizionato (branch)

```
If(p1)
```

```
{
```

```
s1
```

```
}
```

```
if(p2)
```

```
{
```

```
s2
```

```
}
```

Nota: l'eventuale eccezione scatenante da s1 si sarebbe verificata solo a valle del fatto che p1 fosse verificata.

Si può effettuare lo spostamento delle istruzioni fuori dalle clausole condizionali solo se sono sicuro di mantenere la correttezza del risultato. Facendo ciò comunque non mi è più garantito quanto affermato nella NOTA.

Se io non sposto le istruzioni fuori dall'if da cui dipendo non perdo la correttezza del codice. Se le sposto devo garantire la correttezza dei dati (flusso e risultato) e la correttezza della gestione delle eccezioni che eventualmente si potevano verificare.

Ci sono casi in cui è possibile violare la dipendenza di controllo senza inalterare il comportamento a fronte delle eccezioni o il flusso dati.

Dynamic Scheduling

Come già detto, rispetto allo scheduling statico, reco il vantaggio di poter rilevare non solo le dipendenze al livello di registri ma anche al livello di celle di memoria.

Inoltre essendo fatto in hardware, evita la ricompilazione, perché quando il processore si becca il codice non ottimizzato lui stesso a runtime ottimizzerà il codice.

Questo esempio potrebbe chiarire l'importanza di un approccio come quello del dynamic scheduling:

```
DIV.D F0,F2,F4
```

```
ADD.D F10,F0,F8
```

```
SUB.D F12,F8
```

Nelle prime due c'è una dipendenza di dato su F1. Questa dipendenza di dato unita al fatto che le DIV ci sta una vita a calcolare, ci permette di dire che la ADD andrà in stallo. Anche le SUB, che di per sé non ha dipendenze con le altre istruzioni e anche quando potrebbe tranquillamente andare avanti, verrà messa in stallo.

Possiamo migliorare le prestazioni andando ad eliminare il vincolo della esecuzione in-order delle istruzioni. Ecco quindi come introduciamo la out-of-order execution. In particolare nel momento in cui permetto al processore di eseguire le istruzioni in ordine diverso da quello originale dovrò stare attento al fatto che possono verificarsi degli hazard di tipo WAR e WAW. Infine potrò avere delle eccezioni imprecise: data un'esecuzione out-of-order, avrò istruzioni che vengono eseguite prima rispetto ad altre istruzioni che le precedono, e se queste istruzioni mi scatenano un'eccezione, ecco che non potrò più garantire il meccanismo preciso delle eccezioni.

1

2

4

5 ← con un'istruzione sulla 5, posso si bloccare la 6 e la 7, ma la 3 che prima rientrava tra le istruzioni successive alla 5 sarà stata già eseguita, dunque, IMPRECISE EXCEPTION!!!

6

3

7

Splitting ID Stage

Lo stadio di decodifica aveva l'incarico di accedere agli operandi e di decodificare l'istruzione. Volendo splittarlo in due parti, otterrò:

- **ISSUE:** decodifica l'istruzione, accertandosi sull'eventuale presenza di hazard strutturali perché se mi arriva un'istruzione di divisione FP e so che EX è già occupata con una divisione FP posso prevenire un'hazard strutturale.
- **READ OPERANDS:** una volta che sappiamo a quale unità funzionale è assegnata l'istruzione, possiamo pensare agli operandi vedendo se sono disponibili e nei casi in cui gli operandi non sono disponibili dobbiamo aspettare mentre tutte quelle pronte le facciamo andare avanti.

Dalle separazioni effettuate ottengo un disaccoppiamento tra gli hazard strutturali e gli hazard di dato.

Da notare è che Instruction fetch è in-order, come anche l'ISSUE.

E' dall'EX in poi che si va out-of-order.

Abbiamo detto che riceviamo gli hazard strutturali, ma bloccano le singole istruzioni. Gli hazard di dato vengono rilevate, e anche queste bloccano le singole istruzioni il che significa che se le altre istruzioni sono pronte queste vanno avanti tranquillamente.

Per quanto riguarda le dipendenze di nome vengono risolte con il register naming e la tecnica che vedremo fa pesante uso del register naming.

Hardware schemes for dynamic scheduling

Ci sono ben due tipi:

- Scoreboarding
- Tomasulo's unit

L'architettura di Tomasulo è alla base di procesori superscalari. A monte dell'istruzione unit c'è l'unità di fetch. Questa cosa delle istruzioni è alla base su cui lavorano gli altri moduli. A valle di questa Instruction queue c'è un'unità chiamata ISSUE che decodifica le istruzioni prelevate dall'Instruction queue e le assegna alle unità funzionali. La novità sono le *reservation station* che compaiono a monte delle unità funzionali FP:

Tomasuolo, pensando al problema degli hazard di dato e strutturali, pensò di accodare alle reservation station (buffer in cui sono accodate le istruzioni che competono ad una unità funzionale) le istruzioni che vanno eseguite da una particolare unità funzionale. Ogni unità funzionale ha una reservation station. L'ISSUE verifica se c'è spazio in una reservation station di destinazione e se non c'è spazio si verifica un hazard strutturale e quell'istruzione viene memorizzata in un buffer interno della ISSUE stessa. Per tutte le istruzioni usuali (ADD, SUB,...) la ISSUE assegna una istruzione alla relativa reservation station, dopodiché va a vedere di quali operandi ha bisogno. Dentro il register file, associato ad ogni registro c'è un flag che mi dice se quel registro ha un valore stabile o sta per cambiare: se il valore sta per cambiare dovrò attendere, fino a quando l'istruzione non avrà modificato il dato e resetto il flag. Se gli operandi sono stabili li guardo e li metto nella reservation station. Se uno o entrambi gli operandi non sono disponibili nella reservation station setterò dei flag per dire che quell'istruzione non può partire finché i registri che richiede come operandi non conterranno un valore stabile, il che richiede il completamento delle istruzioni che modificano i registri richiesti.

Il Common Data Bus CDB permette la notifica su quali operandi sono disponibili: qui reservation station controllerà se gli operandi competono ad una istruzione che li aspettava, e mentre questi passano sul CDB per essere scritti sul register file lei li preleva per fornirli all'istruzione. L'istruzione, che fino ad allora era bloccata, riceverà gli operandi e potrà proseguire. Sono quindi le reservation station a tenere memoria degli operandi che attendono e a sapere da quale unità funzionale l'attendono. Quando sul common data bus le reservation station si preleva il dato o i dati e li passa all'istruzione che li richiedeva.

Ovviamente ci sarà un'unità funzionale che scrive il risultato nel CDB, e poi anche se sono 2 o 3 le reservation station ad attendere ognuna di loro prenderà il dato e lo fornirà all'istruzione che lo attende.

Nota: I dati sono etichettati sul CDB: e questo che garantisce che la reservation station possa sapere se il dato gli interessa o meno. C'è da dire anche che alle reservation station viene comunicato qual'è il dato che gli interessa (uno o più di uno) e anche qual'è eventualmente l'unità funzionale che lo scriverà sul CDB.

Reservation Station

Le unità funzionali ricevono dall'unità di decodifica le istruzioni che competono loro, immagazzinandolo nelle reservation station. Non si tratta semplicemente di bufferizzare le istruzioni, ma svolgono operazioni più intelligenti riguardo agli operandi:

viene memorizzato per ogni istruzione che tipo di istruzione è, dopodiché se anche uno solo degli operandi non è disponibile si vede qual è l'ID della reservation station in cui risiede l'istruzione che produrrà il risultato. Qualora invece il valore degli operandi sia disponibile viene direttamente copiato nelle reservation station.

Il meccanismo delle reservation station, per quanto finora detto, ci evita i RAW (Read After Write) hazard: un'istruzione è bloccata nelle reservation station finché non viene prodotto il dato o i dati che servono loro.

Restano ancora le dipendenze di nome (WAR, WAW) e abbiamo detto che queste possono essere evitate facendo register naming. Il register naming viene fatto implicitamente nell'architettura di Tomasuolo, per via dei campi delle reservation station in cui vengono copiati degli operandi. Gli hazard non si possono verificare, e il punto chiave è che le istruzioni vengono passate nelle reservation station in modalità FIFO.

L'address unit assolve il compito di calcolare l'indirizzo effettivo.

Common Data Bus

L'unità funzionale una volta prodotto il risultato lo pone sul CDB etichettato con l'id delle reservation station in cui era contenente l'istruzione che ha prodotto il risultato. Il CDB oltre ad essere collegata alle reservation station è collegato al register file su cui vengono memorizzati i risultati.

A dire il vero è anche collegato agli store buffer. Infatti operazioni di Load e Store vengono trattate diversamente essendo inviate dall'istruzione queue all'address unit per andare a finire in due buffer distinti. Le Store attendono negli Store Buffer che venga prodotto il risultato per memorizzarlo in memoria, ovviamente per le load non vale in quanto calcolato l'indirizzo effettivo vengono avviate ai buffer da cui in modalità FIFO vengono prelevate per svolgere il loro compito.

Esecuzione delle istruzioni

L'esecuzione di una istruzione avviene in ben 3 passi (visione molto alta). Il primo stadio è quello di ISSUE, che si occupa di inviare alla reservation station giusta una certa istruzione ed effettua una decodifica, con caricamento degli operandi. La fase di EXECUTE: appena una istruzione è pronta (ovvero il valore di tutti i suoi operandi sono stati scritti nei campi delle reservation station) viene eseguita. Degno di nota è il comportamento di questa architettura nei confronti delle istruzioni di Load/Store: sono un po' diverse perché fanno accesso in memoria.

Il meccanismo è effettuato in due piani:

1. Il processore calcola l'indirizzo cui l'istruzione di Load/Store afferisce. L'indirizzo, una volta calcolato, viene memorizzato in un Load/Store Buffer. L'operazione di calcolo degli indirizzi deve essere in ordine strettamente fifo
2. L'istruzione (Load/Store) procede ad accedere in memoria, attenzione: mentre la load accede direttamente in memoria la store dovrà attendere che sul CDB compaia l'operando la andare a scrivere in memoria.

Nella fase di write si effettuano le scritture.

Le Load/Store vengono trattate con un po' più di cautela perché bisogna fare attenzione alle dipendenze di dato quando si fa riferimento a celle di memoria. Il problema è che l'indirizzo delle celle di memoria è usato solo a tempo di esecuzione. L'architettura di Tomasulo riesce ad identificare perfettamente quei pochi casi in cui una Load legga subito dopo una Store. Il vincolo come detto sta nell'ordine FIFO. Una store può quindi essere bloccata in attesa dell'operando e possono accadere:

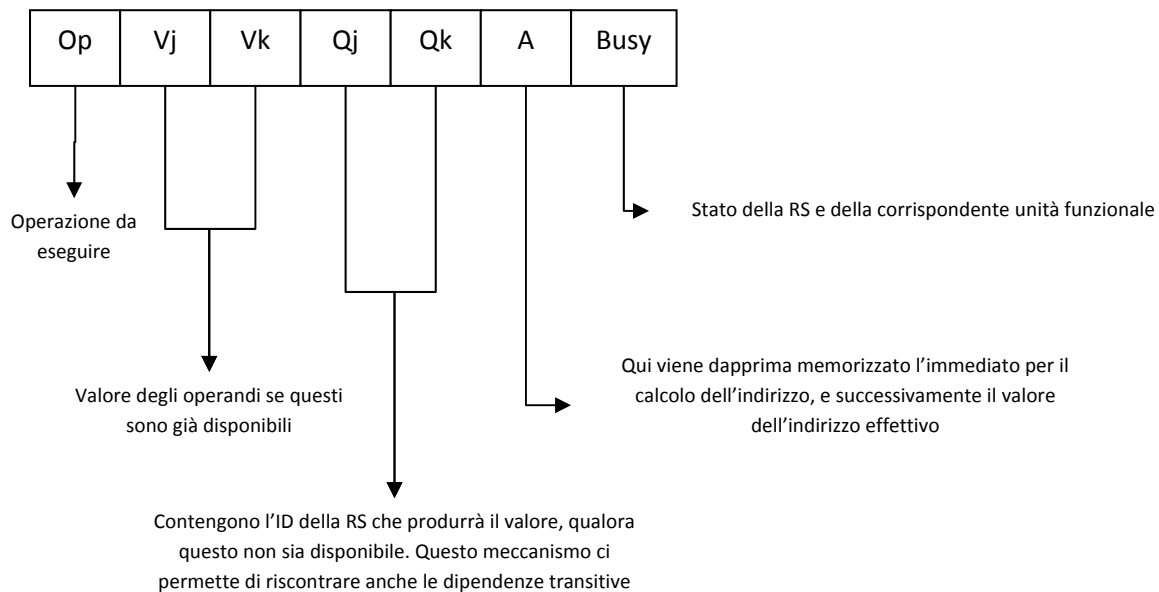
- La store viene superata da una load che non opera sulle stesse celle di memoria: quindi è TUTTO OK.
- La store occupa sulla stessa cella di memoria delle load a cui è stata superata. In questo caso ecco che la load potrebbe leggere dati obsoleti! Questo deve essere evitato

Ecco perché il calcolo degli indirizzi deve essere effettuato rispettando l'ordine con cui le istruzioni compaiono nel codice.

TAGS

Anche il register file associa ad ogni registro il tag in cui mette l'id della reservation station che sta utilizzando quel registro come output (contenuto instabile). In tal caso, l'istruzione non preleva il valore ma l'id della reservation station.

Reservation Station Field



Vantaggi:

1. Ciascuna RS si gestisce i suoi hazard, quindi la logica di controllo distribuita.
2. Si Risolvono tutti gli hazard

Svantaggi:

1. Le RS prese singolarmente sono semplici ma messe tutte insieme sono complicate. Queste RS sono una serie di CAM che beccano il dato taggato e confrontano i tag andando a prelevare solo quello che realmente gli interessa.
2. Il CDB rischia di diventare il collo di bottiglia del sistema. La soluzione adottata in alcuni sistemi è la duplicazione del CDB. Il problema è che a questo punto le RS dovranno stare in ascolto su 2 bud, il che richiede di avere una CAM dual port, comunque molto costosa.

Nota: l'istruzione queue è fondamentale perché lo stadio di fetch sia disaccoppiato dalla decodifica. Inoltre questa cosa è più piena è meglio è.

1. Di solito il fetch carica 2 istruzioni per colpo di clock. L'unità di ISSUE, molto spesso ha un parallelismo per volta: la coda si riempie a 2 istruzioni per volta e viene inoltrata un'istruzione per volta. Quando si riempie, lo stadio di decodifica può stare buono per un po' e anche se in fase di fetch si verifica un cache miss può continuare a decodificare per un po'.
2. Questa architettura è debole rispetto alle istruzioni di salto. Appena si becca un'istruzione di salto il meccanismo si blocca processando prima il salto (svuotando magari la coda delle istruzioni) solo al termine di questo riprende. Una tecnica alternativa rispetto alla precedente è quella basata su previsioni che il salto possa essere preso o non preso, e quindi continua ad eseguire operazioni opportune.

3. Ultima considerazione riguarda il loop unrolling: con l'architettura di Tomasulo il loop unrolling può essere effettuato dinamicamente da parte del processore. Le istruzioni di salto sono eseguite naturalmente in parallelo.

Branch prediction, due famiglie di tecniche: statiche, dinamiche

L'architettura di Tomasulo mi permette di andare molto più veloce rispetto all'architettura MIPS con pipeline. Le prestazioni della prima sono migliori. Un dato di fatto è però che i salti introducono aspetti un po' più negativi, perché come già detto un salto introduce una dipendenza di controllo. L'impatto negativo dei salti è ancora più accentuato dal fatto che nell'architettura di Tomasulo abbiamo molte più unità funzionali in parallelo.

L'obiettivo è quindi quello di trovare tecniche con cui gestire in maniera efficiente i salti.

Sui salti incondizionati c'è poco da fare: si tenta di calcolare l'indirizzo target il prima possibile, e possibilmente nella fase di fetch.

Molto più complicata è la situazione dei salti condizionati: in questo caso si tenta di fare delle previsioni, e le possibili scelte in questo caso sono 2:

- Il salto viene preso
- Il salto non viene preso

Ovviamente queste due situazioni non sono equiprobabili (o almeno non è detto che lo siano). Le previsioni possono essere fatte dal compilatore in prima analisi, basandosi su alcuni topcis:

- Ad esempio i salti in avanti hanno molta più probabilità di non essere presi rispetto al salto in indietro (che hanno una elevata probabilità di essere presi dato che vengono utilizzati nei cicli, dunque sicuramente i salti verranno presi più di una volta e alla fine non presi. Il compilatore, guardando quindi il tipo di salto contrassegna l'istruzione che poi il processore si troverà a trattare in modo opportuno, queste sono le **TECNICHE STATICHE**.
- Diverso è il discorso delle tecniche dinamiche in cui il processore è incaricato di attendere la scelta.

Da notare è che le tecniche dinamiche sono le seguenti

1. Branch history table (ad uno o due bit, a 2 livelli)
2. Branch prediction buffer

Nel primo, io vorrei memorizzare per ogni salto quello che è successo e quando viene preso o non preso, quindi per ogni salto vado a memorizzare se quel salto è stato preso o non preso.

Quindi:

- Ogni volta che eseguo un salto accedo alla tabella per aggiornare i dati
- Ogni volta che devo eseguire un salto consulto la tabella per verificare l'esito

Il mio obiettivo è avere una struttura quanto più compatta possibile e di facile gestione d'accesso. Vediamo le limitazioni:

- Dovendo implementare questa tabella a bordo del processore avrò un limite al livello di memoria: sia $n=2^K$ il numero di salti che posso memorizzare.

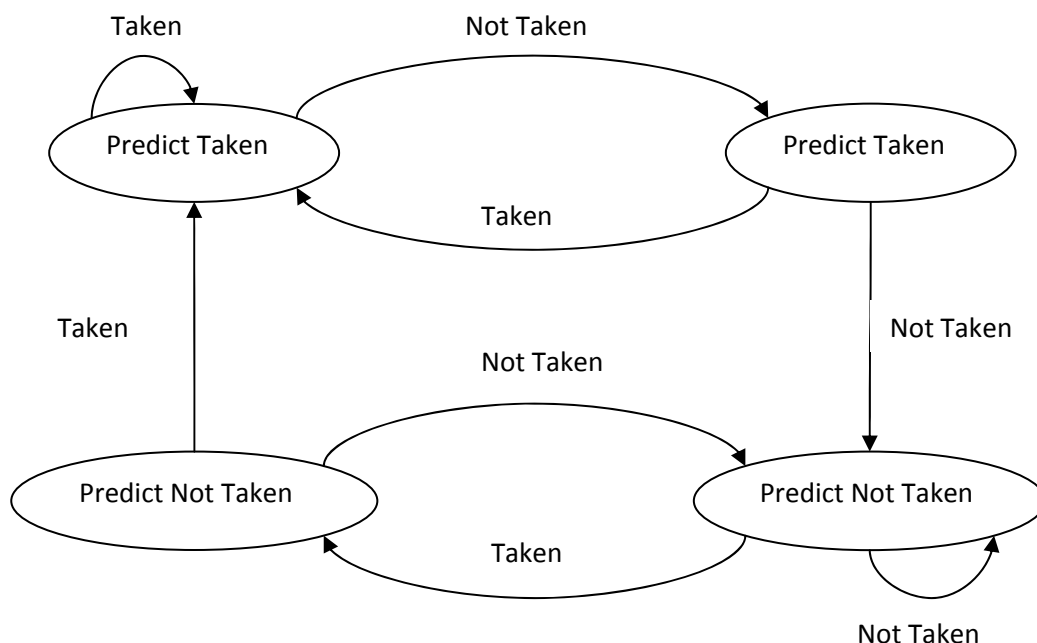
- Dopodiché, a partire dai 32 bit dell'indirizzo di salto, che è il dato con cui identifico il salto, prendo i K bit più bassi e a questi associo un bit che mi dica se il salto è stato preso o non preso e questa sarà l'organizzazione della mia entry in quella tabella. Poiché memorizzo nei k bit più bassi per identificare un salto avrò, 2^k entry che con 10→12 bit mi vanno da 1024 a 4096.

Già a partire dalle informazioni che abbiamo ci possiamo rendere conto di come ci possono essere più entry che magari hanno i k bit più bassi uguali. Il discorso ovviamente varia in relazione anche al valore di K. Qualora io accedo alla tabella in lettura per prendermi la previsione e accedo ad un'informazione riguardante un altro salto con i k bit più bassi uguali a quelli della mia istruzione, leggerò dati che non mi riguardano. E' il k (tipicamente da 10 a 12 bit) a crearmi questo problema, perché se K fosse a 32 bit non ci sarebbe questo problema.

Branch Predict Techniques

Sulla branch table avevamo già detto cos'era; in sostanza altro non è che una tabella che mi tiene traccia dei salti effettuati in passato, cui si accede con la parte bassa dell'istruzione di salto. Avevamo già detto che il problema stava nel fatto che prendendo n bit dalla parte bassa non ho sicurezza sul fatto che l'informazione sia relativa proprio al salto in esame, il che intacca la efficacia della previsione.

Con lo scopo di aumentare l'efficacia della previsione, si passa agli schemi a due bit: non mi basta sapere che il salto non venga preso al salto precedente, ma guardo un po' più a fondo nel passato, dicendo che è preso o non preso solo se accade un certo numero di volte. Quindi ci riconduce ad una macchina a stati:



Se sono nella posizione di salto preso, dovrò aspettare che il salto venga preso per 2 volte di fila prima di poter dire che il salto non viene preso, e viceversa. Si ragiona con un contatore in modulo 2. Quello che mi serve è un hardware minimo, ma mi permette di ottenere una maggiore efficacia nel meccanismo di previsione. Come già detto, la tabella è in grado di fare una previsione sul fatto che il salto venga preso o

meno. Il fatto che il salto venga preso o meno lo vedo nello stadio di decode. Ma a questo punto, la branch history table non mi dirà nulla di nuovo rispetto a quello che già saprò dalla fase di decodifica. Quindi la branch history table mi servirà nel caso in cui la valutazione della condizione di salto non venga effettuata nella fase di ID.

Altra particolarità da notare è che non è detto che se con 2 bit ho ottenuto dei miglioramenti, aumentando ancora il n° di bit otterrò miglioramenti a dismisura. Anzi, le statistiche dimostrano che con più di 2 bit non si notano miglioramenti sostanziali.

L'impatto negativo dei salti sulle prestazioni dipende:

- Dall'accuratezza con cui effettuo le previsioni
- Dal costo nell'errore delle previsioni
- Dal numero di salti che si verificano

Per i soliti programmi campione, oltre 4096 entry nella tabella non conviene andare, e questo avviene su tutti i programmi. Questa analisi è importante perché dice al progettista come dimensionare la tabella, il che ha un impatto non trascurabile anche sulla costruzione del chip.

Correlating predictors (two-level predictors)

Vediamo il seguente codice:

```
If(aa==2)
    aa=0;
if(bb==2)
    bb=0;
if(aa!=bb){ }
```

Il risultato di questo salto è influenzato dalle due istruzioni (di salto) precedenti

A questo punto sono possibili varie situazioni. Nel caso dei predittori di tipo (1,1), ad ogni salto associo l'info su 1 bit nel fatto che il salto sia stato preso, e la previsione su un altro bit sul fatto che il salto non sia stato preso.

Il caso più generico è ovviamente quello dei predittori (m,n). Il primo è il n° di salti precedenti di cui tengo conto. Il secondo è il n° di bit di cui dispongo per fare la previsione. Il 2° parametro o vale 1 o vale 2. Il 1° da 1 in poi, ma in genere vale 2,3,4 al max.

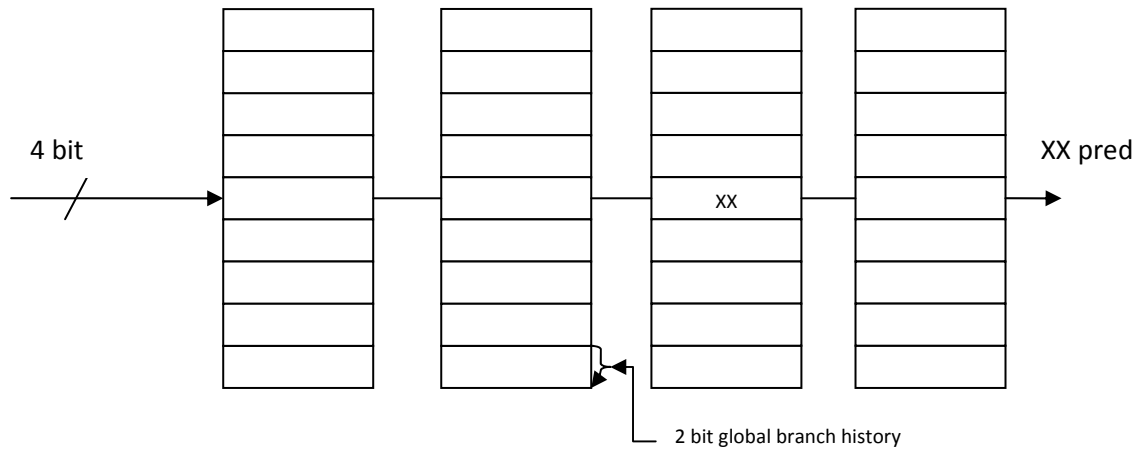
| | | | | |
|--|-------|------|------|-----|
| | NT NT | NT T | T NT | T T |
| | T | T | NT | NT |

Es. Caso 2,1

I valori che trovo li dentro possono cambiare sulla base delle FSM (Finite State Machine) che abbiamo visto prima. Attenzione alla struttura dati che salta fuori nel caso in cui ho un predittore (2,2).

Per ogni salto, considero i due salti precedenti (circa 4 previsioni, ognuna delle quali è organizzata in 2 bit!). Poi c'è ancora un parametro, che è quanti salti voglio memorizzare nella mia tabella, e che determina il

n° di entry che avrò nella tabella (abbiamo visto prima che l'ottimo è 4096). Come accedo ad una entry? Con i 4 bit bassi. Dopodiché come accedo alla previsione?

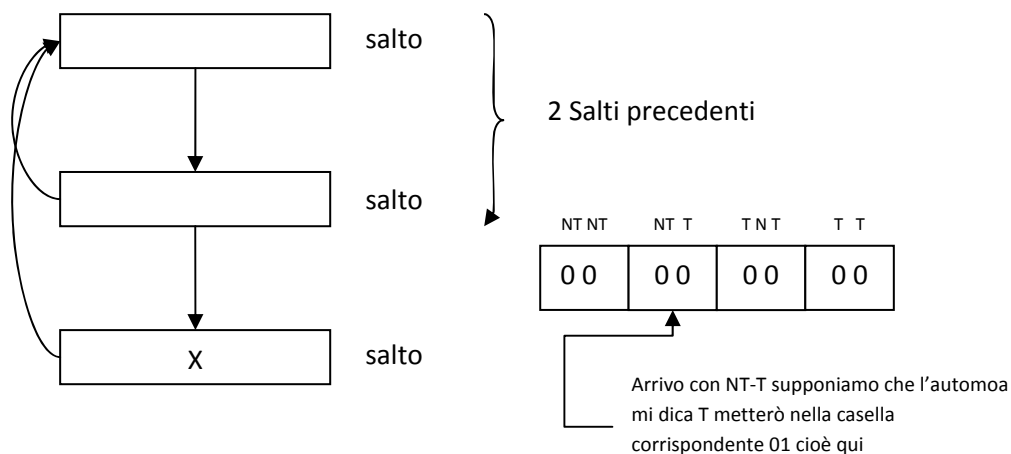


Ho una riga per ogni salto. Per ogni riga ho 4 previsioni diverse a seconda di cosa è successo nei 2 salti precedenti. Dentro ogni elemento, dei 4 che ci sono in una riga, 2 bit mi derivano dall'automa e da cosa ha sputato fuori sulla base dei due salti precedenti.

$(m,n) \rightarrow 2^m \times n \times n^* \text{ entry} = n^\circ \text{ bit richiesti}$

Non parlerò di una matrice, bensì di un vettore di n^*4 . Come potrò accedere? Prendo i 4 bit bassi dell'indirizzo e gli concateno dietro i 2 bit che derivano dall'esito degli ultimi 2 salti. Ottengo il bit, che è proprio quello che mi occorre per gestire questa situazione. Posso accedere ad una memoria da 64 entry (128 bit)

Questo discorso funziona che è una bomba, e lo dimostrano gli SPEC89 a cui sono stati sottoposti.



Altra strada, alternativa alla Branch History Table è quella Target Buffer.

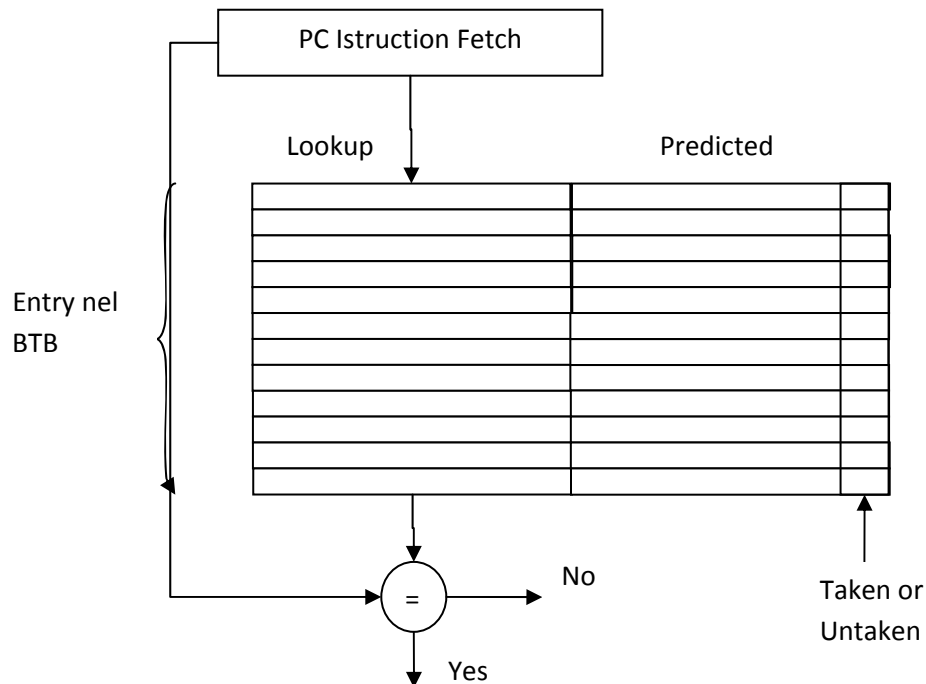
Il Branch Target Buffer contiene per ogni riga:

- L'indirizzo di un salto eseguito in precedenza

- Il valore da caricare nel PC

Ancora una volta, accediamo al Branch Target Buffer tramite gli n bit bassi: confronto il campo lookup con l'indirizzo completo del salto, vado sul predicted PC per sapere il valore da caricare nel PC. Alla fine del campo c'è un bit che mi indica se il salto debba essere preso o non preso: non è particolarmente utile, perché caricherò nel PC comunque il campo predicted PC perché:

- Caso Taken → ci sarà l'indirizzo a cui saltare
- Caso UnTaken → ci sarà l'indirizzo successivo a quello di salto



Utilizzo sempre i 4 bit più bassi (dipende dal n° di entry) dall'indirizzo del salto. Il target del salto così ce lo abbiamo alla fine della fase di fetch con un aumento considerevole delle prestazioni. Dopodichè, se l'indirizzo riguarda un'istruzione non di salto, allora so che non troverò nulla all'interno del BTB. Altrimenti, non è detto che trovi qualcosa, perché è anche possibile che il salto non sia stato ancora eseguito. Quello che però ho ottenuto, accedendo al BTB nella fase di fetch parallelamente al mi accesso in memoria per caricarmi l'istruzione, un aumento stratosferico delle prestazioni. Se accedo al BTB ottengo qualcosa, carico nel PC l'indirizzo di salto. Dopodichè, nella decodifica mi devo chiedere se il salto è stato preso o non preso. Se non viene preso, devo svuotare la pipe ed aggiornare le tighe oltre che caricare l'istruzione corretta.

Qualora non ottenga nulla, mi devo ancora rendere conto se:

- Non ho ottenuto nulla perché non è un'istruzione di salto
- Il salto non è stato ancora eseguito, e aggiornare l'entry
- L'informazione riguardante quel salto è stata sovrascritta con l'esito di un altro salto avente gli stessi n bit bassi, il che può accadere quando in tabella esiste un salto diverso con gli stessi bit bassi.

Altra strada: quando durante il fetch accedo al BTB non tiro su l'indirizzo dell'istruzione successiva ma direttamente l'istruzione successiva prevista, risparmiandomi il fetch e guadagnando ancora un colpo di clock. Questa tecnica è nota come Branch Folding. Questo lo si può fare andando a memorizzare nel campo

della predizione direttamente l'istruzione successiva prevista, anziché l'indirizzo, e ciò lo posso fare perché sia l'indirizzo che l'istruzione vanno su 32 bit.

Nota: (m,n) predictors uso gli ultimi m branch per effettuare le previsioni, Dovrò scegliere tra 2^m predittori ciascuno dei quali sarà ad n bit.

Multiple ISSUE Processor

Lo scopo è quello di fare il fetch di più istruzioni contemporaneamente al fine di veder terminare molte più istruzioni rispetto a prima. Vedremo come sfruttare le unità funzionali a bordo del processore al fine di fare il fetch di più istruzioni contemporaneamente spedendole in modo simultaneo alle unità funzionali (Multiple ISSUE).

Ci sono 3 modi di ottenere ciò:

- Processori superscalari: si possono distinguere in altre due sottocategorie, ovvero:
 - Scheduling statico delle istruzioni (compilatore)
 - Scheduling dinamico delle istruzioni (processore)
- I Very Long Instruction Word (VLIW) Processors: sono una nicchia.

Vediamo la prima categoria. E' il compilatore a decidere come distribuire nel tempo le istruzioni alle varie unità funzionali. Non deve solo trovare l'ordine che minimizzi i buchi in termini di operazioni di ogni unità funzionale, ma deve soprattutto farle arrivare in parallelo. Il processore fa il fetch: supponiamo che guarda caso siano 2 contemporaneamente, e fattala decodifica ci troviamo ad avere l'issue di due istruzioni adatte ad essere mandate in parallelo a due unità funzionali: basta che il compilatore strutturi bene la cosa. L'importante è per le istruzioni all'interno di ogni copia che vadano a finire in 2 unità funzionali diverse, altrimenti non potranno essere eseguite in parallelo. Notiamo che l'unità di ISSUE non fa nulla di particolarmente intelligente! L'approccio di Tomasulo è perfettamente compatibile con tutto questo, dovremo solo spendere un po di più in hardware per ottenere il parallelismo veloce.

Adesso vediamo come un processore MIPS possa sfruttare l'approccio statico: facciamo le seguenti assunzioni: per ogni coppia di istruzioni

- Una è sicuramente Load, Store, Branch o Integer ALU operation
- L'altra è Floating Point (attenzione la stessa Load/Store FP rientrano nella prima categoria)

Inoltre i salti sono allineati a indirizzi multipli di 8, e inoltre abbiamo istruzioni su 54 bit. Il Fetch fa due operazioni in parallelo, la decodifica va a 2 a 2, e l'Issue riesce a mandare 2 istruzioni alle rispettive unità funzionali. In questo modo, in uno stesso colpo di clock posso ottenere 2 istruzioni contemporaneamente. Potenzialmente abbiamo 2 istruzioni per colpo di clock che terminiamo. E' chiaro che ci saranno dei problemi da risolvere: ad esempio

- Ci sono delle istruzioni che si seguono tra loro provocando degli hazard

Mentre nei vecchi processori superscalari c'è il requisito di avere un formato fisso per i pacchetti di istruzioni che vanno all'issue, questo vincolo è stato rimosso negli attuali processori superscalari.

Inoltre, attenzione al fatto che il canale IN/OUT della cache deve essere raddoppiato e portato a 64 bit: ad ogni colpo di clock dobbiamo essere in grado di ottenere un pacchetto di 64 bit (2 istruzioni).

Poniamoci adesso nel caso in cui al generico colpo di clock, la cache mi restituisce un pacchetto contenente una sola istruzione perché una delle 2 fa cache miss, cosa succede? Don't worry, il meccanismo è pensato per funzionare con due istruzioni, ma se ce n'è una sola non fa nulla, tutto continua a funzionare

perfettamente. Il problema del pacchetto incompleto può essere dovuto al fatto che il compilatore non riesca bene a comporre le istruzioni tra loro: nel caso ci potrà mettere in mezzo una NOP. Certo le prestazioni degradano un po, ma il meccanismo continua a funzionare. Altro problema riguardante il caso della FP unit che non accettano 2 istruzioni per colpo di clock: caso eclatante è quello della divisione FP.

FP Register Contention: Questo è un altro problema che può venire a creare, nel caso in cui la prima istruzione è una LOAD, una STORE o una FP ci può essere una contesa sulla porta del registro FP. Possibili soluzioni sono:

- Forzare la prima istruzione ad essere eseguita
- Dotare il registro FP di una seconda porta

Altro problema è quello degli hazard di tipo Read After Write (RAW). Quando la prima istruzione è una load, store o move FP e la seconda legge il suo risultato, ecco che si verifica un RAW hazard. La seconda istruzione dovrà attendere. In realtà nell'architettura di Tomasulo ciò non ci crea problemi perché il meccanismo delle reservation station ci garantisce che l'istruzione che non ha tutti gli operandi disponibili venga messo in attesa nelle RS, finché tutti gli operandi sono disponibili.

Caso più critico riguarda i salti: le istruzioni di salto, ad esempio la fine del pacchetto, la seconda può essere l'istruzione che ipotizza che il salto sia stato preso o no. In tal caso, anche se vengono eseguite in parallelo occorre non terminare l'istruzione se l'ipotesi è errata.

Il Multiple-ISSUE statico è utilizzato per processori embedded in cui il consumo di potenza è un requisito fondamentale. Per i processori general purpose (dedicati ai desktop) si sfrutta il Multiple-Issue Dinamico combinato con schedulino dinamico, che mi garantisce due cose:

- L'issue può essere eseguito out-of-order e in esecuzione in order
- Nel caso di branch, l'unico approccio che si può immaginare è attendere che questo sia finito e vedere a quale istruzione saltare

Partiamo da un esempio basato sul ciclo (dato che abbiamo detto e che il loop unrolling viene fatto in modo automatico da questo tipo di soluzione). Il codice che viene fatto fuori dal compilatore, non sarà stato ottimizzato. Il processore, nel momento in cui fa il fetch e lo decodifica vedrà se ci sono istruzioni che possono essere eseguite per ottimizzare: siccome faccio l'issue di 2 istruzioni per colpo di clock, dovrò avere a monte due unità (rispettivamente Fetch e Decodifica) che lavorino a 2 istruzioni per volta. Però mentre prima le istruzioni venivano tirate su due per volta a

Ed ero sicuro che potessero essere eseguite in parallelo, qui si continua a tirarle su due per volta ma l'ISSUE ha la possibilità di mandarle in parallelo dato che a monte da parte del compilatore non è stata fatta alcuna speculazione.

Adesso passiamo al blocco di codice da esaminare al fine di capire il modo in cui funziona questa architettura:

| | |
|-----------|------------|
| Loop: L.D | F0, 0(R1) |
| ADD.D | F4,F0,F2 |
| S.D | F4, 0(R1) |
| DADDUI | R1,R1,#-8 |
| BNE | R1,R2,LOOP |

Nota: l'esito del salto è sempre previsto correttamente, ma per prevenzione si attende che la previsione (si prevede sempre TAKEN) sia effettuata prima di fare l'issue dell'istruzione successiva al Branch. Nota inoltre che di solito quando viene fatto il Fetch di un'istruzione di salto condizionato si mette da parte l'istruzione successiva al fine di ottenere la previsione di salto.

Un prima famiglia di miglioramenti è quello di introdurre una nuova unità funzionale intera. In genere si può individuare a monte quali sono le unità funzionali che possono costituire dei colli di bottiglia e sdoppiarle. Altra unità che può essere sdoppiata è il Common Data Bus: l'esecuzione di un'istruzione può in tal modo proseguire in virtù del fatto che il CDB è sdoppiato. Ciò introduce alcuni costi aggiuntivi:

- Quello del CDB in se
- Quello dell'unità di interfaccia (Arbiter) ai CDB
- Quello delle Reservation Station e del Register File. Le prime ad ogni colpo di clock devono stare in ascolto su entrambi i data bus, dopodiché devono essere in grado di gestire la situazione in cui sull'uno e sull'altro ci siano stati deati che interessano ad una stessa RS.

Avendo un processore semplice, c'è il lavoro del progettista che durerà relativamente poco e che permetterà di evitare bug. Inoltre, a valle del progetto terminato, ci sarà chi dovrà testarlo prima di mandarlo sul silicio. Più il progetto è complesso e più la gente che deve testarlo trbiolerà a rischio di lasciar passare alcuni bug introdotti in fase di progetto. Una componente fondamentale del costo del progetto di un processore è in particolar modo il Test. Apportando le modifiche di cui abbiamo parlato, siamo passati da prestazioni di: $15/16 = 0.94$ (15 istr. 16 colpi di clock) a prestazioni di: $15/11 = 1.36$

Quindi abbiamo migliorato le prestazioni, come possiamo intuire abbiamo aumentato i costi.

Hardware-Based Speculation

La novità sta in questo: nelle unità precedenti avevamo detto che la branch prediction lavora bene. Quando incontravamo un branch tiravamo su una previsione e eseguivamo comunque l'istruzione successiva al salto. Nel caso di multiple issue, quando si incontravano i branch si bloccavano le istruzioni a valle del Branch fino a che non si sapeva se il salto poteva essere preso o meno. Le istruzioni venivano caricate e decodificate, ma non ne veniva fatto l'issue.

Qui si dice invece: perché non le devo eseguire? Se eseguo comunque, ma l'unica cosa che aspetto a fare è la scrittura dei risultati, e attendo fino a sapere se quelle istruzioni andavano eseguite o no (ovvero se il salto andava preso o meno). Facci cioè una speculazione anche sull'esecuzione. Quindi riempio meglio il mio procesore. Questa tecnica del rinviare la scrittura dei risultati mi permette di gestire la situazione delle eccezioni: ancora una volta, quando si verifica un'eccezione in una istruzione eseguite in maniera speculativa, non le servirò subito ma attenderò di sapere se quella istruzione andava eseguita o meno: questo mi garantirà di gestire in modo corretto le eccezioni. Ovviamente, qualora l'istruzione non andava eseguita, la butterò via, e non servirò nemmeno l'eccezione.

Nell'HW-Based speculation combino 3 tecniche:

- Predizione dinamica del branch
- Speculazione
- Scheduling

Introduciamo una fase finale che comprende:

- La scrittura dei risultati (nel register file, o nella memoria)

- La gestione delle eccezioni (solo nel caso in cui una istruzione diventa COMMITTED si gestirà l'eccezione scatenata da quest'ultima)

Questa fase finale prenderà il nome di instruction COMMIT.

Questo comporta l'introduzione di una ulteriore struttura, che viene chiamata Reorder Buffer (ROB). Contiene tutta una serie di campi per ogni istruzione che permetterà la scrittura dei risultati: "devo sapere cosa devo andare a scrivere e soprattutto DOVE"

Viene chiamato Reorder Buffer perché:

- Deve garantire che la scrittura dei risultati avvenga solo in caso di conferma del fatto che quell'istruzione deve essere eseguita.
- Deve garantire che l'ordine in cui vengono scritti i risultati rispetti quello presente nel codice sorgente, ma in ordine di FETCH.

In definitiva, il ROB oltre a fungere da buffer per le istruzioni speculative è come se mi riordinasse le istruzioni in modo che la fase di scrittura dei risultati rispetti quello con cui le istruzioni sono caricate. Dunque, questo Buffer non è FIFO! Le istruzioni entrano nel Buffer out-of-order ma ne escono fuori rispettando l'ordine con cui ne vengono fatte il Fetch. Nel ROB non ci stanno solo le istruzioni speculative, ma anche quelle terminate troppo in fretta.

A questo punto: si va a vedere se sono disponibili allora li prelevo direttamente da la sbloccando l'istruzione in attesa degli operandi, mentre se non sono presenti nemmeno li la Reservation Station si metterò in ascolto sul CDB.

Quando l'istruzione è pronta ad essere spedita ad una reservation station, guardo se c'è una RS pronta ad accogliere l'istruzione, e vado a vedere se c'è un entry nel Reorder Buffer che possa accogliere quell'istruzione: solo se queste due considerazioni sono verificate potrò lasciar proseguire l'istruzione, altrimenti le block. Ciascuna Entry nel ROB ha 4 campi:

- Instruction Type
- Destination
- Value
- Ready

In realtà vedremo che c'è anche un altro flag, che è quello riguardante le eccezioni, in modo che queste possano essere gestite in modo corretto e per come avevamo precedentemente detto. A monte dell' instruction queue assumeremo che ci siano delle unità di fetch e di decodifica capaci di sparare due istruzioni per colpo di clock. Dentro l' instruction queue ci sono delle strutture dati che recano delle informazioni riguardanti la istruzione in se.

Adesso vediamo l'ISSUE: si preleva un istruzione dalla instruction queue, e vado a vedere se c'è una RS capace di accoglierla. Come anche un'entry libera ne ROB. Se le assegno un'entry nel ROB questa proseguirà. Se una di queste due condizioni non è verificata l'istruzione rimane nell' Instruction Queue. Se invece le due condizioni sono verificate si blocca l'entry nel ROB, l'istruzione passa nella RS e si passa alla ricerca degli operandi:

- Vengono cercati nel register file e se non vengono trovati si cerca nel ROB, e a questo punto:
 - Se vengono trovati si prendono gli operandi direttamente del ROB
 - Altrimenti si mette l'istruzione in attesa e la RS si metterò in ascolto sul CDB

Le istruzioni questa volta verranno etichettate con il n° della entry associata nel reorder buffer, come anche i dati quando vengono posti nel CDB.

NOTA: se un'istruzione è bloccata perché non c'è posto nella RS allora si potrà proseguire con l'issue di quelle successive, ma se non ci sono entry libere nel ROB.

Nella fase di ESECUZIONE : la si effettua quando sono pronti gli operandi. Questo meccanismo mi evita degli RAW hazard. Nella fase di Write si scrivono i valori prodotti sul CDB e contemporaneamente il n° dell'entry del ROB dove dovrà essere scritto.

L'ultima fase è quella di COMMIT: in questa fase, i risultati presenti nel ROB vengono scritti nel Register File, ed è da ricordare che il ROB riordina le istruzioni al fine di garantire la scrittura dei risultati in ordine, come anche la gestione delle eccezioni in modo corretto. Il ROB è implementato come un Buffer circolare. Quando al Top del ROB mi arriva un'istruzione di salto:

- Verifico: se avevo azzeccato la previsione tutto ok, mentre se avevo sbagliato dovrò svuotare il ROB e cominciare a caricare nel ROB dell'istruzione successiva a quella di salto. Nel caso in cui avevo azzeccato la previsione scrivo i risultati in memoria o sui registri.

In questo modo non potrò più avere conflitti di tipo WAW e WAR. Anche gli hazard di tipo RAW attraverso la memoria vengono evitati, e sono evitati in questo modo:

- Bisogna che gli indirizzi delle Load vengano sempre calcolati dopo gli indirizzi di tutte le Store precedenti. Quindi la Load potrebbe andare in stallo perché una store precedente ad esse non è ancora riuscita a calcolare l'indirizzo.
- Una volta che sono sicuro che il calcolo degli indirizzi di tutte le store precedenti siano stati calcolati, devo chiedermi: → c'è qualche Store con lo stesso indirizzo che non ha ancora scritto il risultato in memoria? Se sì devo star fermo altrimenti proseguo.

In questa struttura, le store assumono una struttura particolare, articolare in due parti:

- Una prima parte in cui si calcola l'indirizzo
- Una seconda parte in cui si scrivono i risultati in memoria

L'esecuzione ingloberà la prima parte, dopodiché questa store entrerà nel ROB anche se magari non ha gli operandi da scrivere. Quando però incontro una Load, dovrò andare a vedere se in tutti i posti in cui ci può essere una store in stallo se ce n'è ma che afferisce allo stesso mio indirizzo che compare nella Load. L'unità funzionale Load/Store deve quindi per più Load andare a scandire il ROB per accertarsi di quanto detto finora. Questo deve essere fatto in un solo colpo di clock il che richiederà di avere una CAM che come sappiamo costa!

Quanto alle eccezioni, abbiamo detto che:

- Vengono eseguite quando un'istruzione viene completata
- Rispettano l'ordine con cui le istruzioni figurano nel codice sorgente

Questo mi garantisce di avere un meccanismo di eccezioni precise. Quindi ecco che nel ROB dovremo avere per ogni entry un ulteriore flag che mi dice se quella istruzione ha scatenato o meno una eccezione in modo da gestirla nel momento in cui viene completata l'istruzione.

Nota: Il buffer circolare è ordinato sulla base del n° assegnato dal fetch, mentre il n° dell'entry è quello che mi serve quando nel momento in cui della Instruction queue devo prelevare un'istruzione e vado a vedere se c'è una RS pronta ad accoglierla e se c'è un entry libera nel ROB, per riservarmela.

Esempio:

```
Loop: LD      R2,0(R1)
        DADDIU R2,R2,#1
        SD     R2,0(R1)
        DADDIU R1,R1,#4
        BNE   R2,R3,LOOP
```

Con la struttura che abbiamo finora presentato l'unico tipo di problema che non siamo riusciti ad eliminare è la dipendenza di dato.

In questo esempio vedremo due casi, con e senza speculazione.

Il primo caso è quello senza speculazione: assumeremo che il processore sia in grado di fare il fetch, decodifica ed issue a 2 istruzioni per volta. Non essendoci speculazione non c'è reorder buffer: c'è multiple issue, Branch Prediction ma non c'è speculazione.

La SD effettua l'accesso in memoria al 7° colpo di clock perché attende l'operando della DADDIU (le store arrivano al calcolo dell'indirizzo ma se l'operando non è disponibile devono attendere). La BNE viene caricata prima del 3° colpo di clock: non appena si fa la decodifica si capisce che è un branch e l'istruzione successiva al branch che c'era sul pacchetto viene buttata via (questo può avvenire perché quello è un salto condizionato e perché si fa l'ipotesi che sia preso). Si fa il calcolo dell'indirizzo nella fase di fetch dunque uno perché già al colpo di clock 4 abbiamo il pacchetto successivo di 2 istruzioni, ma successivo a partire dal target ottenuto dal Branch Target Buffer in fase di Decodifica. La BNE viene eseguita al colpo di clock n°7, mentre la LD e la DADDIU devono attendere rispettivamente l'8 e il 9 colpo di clock.

Il secondo caso migliora la speculazione, e quindi ci sarà il commit oltre che il ROB. I numeri nella colonna del commit devono essere strettamente crescenti, anche se a volte presi a blocchi di 2.

Mentre prima la LD aspettava che il salto venisse eseguito, e quindi veniva eseguita all'8 colpo di clock, qui va al 5° perché avremo la speculazione. Come già detto in passato: se quando il branch arriva al top del ROB vediamo che la previsione è stata fatta in modo corretto allora è tutto ok. In caso contrario dovremo svuotare il ROB annullando tutti gli effetti prodotti dalla speculazione.

Dai risultati ottenuti:

- 1° caso: 19 colpi di clock
- 2° caso: 14 colpi di clock

Abbiamo visto che molti processori, anziché usare il ROB, usano la tecnica del Register Renaming. Alcuni processori, o in aggiunta al ROB o in sostituzione usano un banco di registri che estendono il register file che sono fisici ma non architetturali, nel senso che esistono ma il programmatore non li vede. Quei Registri vengono sfruttati dinamicamente dal processore al fine di evitare hazard di tipo WAR e WAW. Quindi il risultato anziché scriverlo nel ROB viene messo in quei registri ombra: quando l'istruzione viene committed non si rifà la copiatura, ma esiste una tabella di corrispondenza chiamata renaming map: quindi quando si fa

l'issue, anziché riservare un posto nel ROB per scrivere il risultato, me lo alloco nella renaming map, che mi dice istante per istante da quale registro fisico è implementato un certo registro architetturale. Questa seconda soluzione mi garantisce maggiore flessibilità per cui spesso si propende per la scelta di quest'ultima. In questa versione non ci sono registri fisici e registri architetturali, ma saranno tutti fisici, e sarà la renaming map a dirmi quale registro fisico di volta in volta fa le veci di un registro architetturale.

Per quanto riguarda il meccanismo di gestione delle eccezioni, in alcuni processori si fa un ragionamento diverso rispetto a quanto detto: molti processori distinguono le varie tipologie di eventi scatenabili da un'esecuzione e a seconda che l'istruzione sia speculativa o meno li servono subito oppure no. Quindi quando un'istruzione scatena un'eccezione e non mi costa molto in termini di tempo servire quell'eccezione, la servo, altrimenti se vedo che mi richiede troppo tempo la lascio stare, e questo anche in dipendenza del fatto che l'istruzione sia speculativa o meno.

Intel P& and NetBurst microarchitecture

Il P6 è l'architettura che sta alla base delle 3 generazioni di processori che sono

- Pentium PRO
- Pentium II
- Pentium III

Nel tempo sono state introdotte delle estensioni, cambia il clock rate, cambia l'architettura della cache e l'architettura di memoria. Le prestazioni della cache non dipendono dalle dimensioni: è solo uno dei modi per apportare miglioramenti, ma non è l'unico e tra l'altro nemmeno quello garantito.

Di solito, la L1 cache è sempre on-board sul processore mentre per la cache di secondo livello sta sempre fuori.

Dal Pentium II in poi sia le L1 che le L2 cache sono passate da 8 K a 16 K.

Quando il Pentium Pro uscì aveva la peculiarità di avere due die impacchettati in modo particolare chiamato *cartiridge*. Nelle versioni successive, dal Pentium II in poi il problema non si è più posto. Con il Pentium II si introduce l'MMX utile per l'ottimizzazione di calcoli vettoriali, mentre con il Pentium III si introdusse l'SSE per la grafica.

L'architettura P6 trasla ciascuna istruzione a 32 bit in una serie di micro-operazioni m-ops, che sono simili alle tipiche istruzioni RISC. Ecco cos'è che garantisce ad un codice scritto per 80x86 di girare tranquillamente sul P6. L'idea è quella di dire: quando il processore fa il fetch di un'istruzione con codifica 80x86, uno stadio a valle andrà a traslare le istruzioni in una serie (ogni istruzione anche in dipendenza della sua complessità avrà una serie o una sola (m-ops, ad esempio le NOP). Ciò complica non poco le cose, ma ciò risolve il problema della compatibilità.

Il P6 ha un parallelismo di 3, il che significa che il P6 fa il dispatch (o issue) di 3 m-ops per colpo di clock. Il commit può al più avere 3 m-ops per colpo di clock.

L'architettura a valle dello stadio di traduzione in m-ops è simile a quella che abbiamo visto precedentemente. Nella sua ultima versione. Ciò che cambia sensibilmente è soprattutto il n° di stadi che sale a 14 di questi:

- 8 stadi: sono destinati a fetch, codifica e dispatch
- 3 stadi: per l'esecuzione out-of-order
- 3 stadi: per la gestione del CDB e altro

L'unità di fetch si interfaccia con la cache istruzioni e per colpo di clock tira su 2 byte che possono contenere almeno 5 istruzioni. Inoltre decodifica le istruzioni al fine di capire come impacchettarle e lo fa identificando l'inizio di ogni istruzione. Dopodichè fa la branch predict con li Branch Target Buffer. Il BTB ha 512 righe e quindi memorizza al più le informazioni di 512 salti. Qualora il BTB non mi ritorni nulla quando io gli passo un indirizzo, assume che i salti all'indietro siano presi e quelli in avanti non siano presi. Il BTB non gli dice nulla in 2 casi:

- Il salto viene eseguito per la prima volta
- La entry cui faccio riferimento ha gli stessi n bit bassi unici ma non è quella che mi interessa

Se sbaglio la previsione di un salto butto via tra i 10 e i 15 cicli: più la pipeline è profonda (aumento la frequenza di clock facendo in questo modo) e più tempo ci metterò a ricaricarla.

Poi viene l'unità di decodifica. Poiché questa è l'unità cui è demandata il compito della traduzione delle m-ops qui dentro c'è una paccata d'hardware. Ci sono 3 unità funzionali che lavorano in parallelo:

- Una si dedica alle istruzioni più complesse che richiedono una serie di m-ops per essere tradotte
- Due si dedicano alle istruzioni più semplici

Nota: in virtù delle 3 unità al max dla Fetch alla decodifica ci vanno 3 istruzioni, di cui 1 a 32 bit e 2 a 16 bit.

Questa unità è complessa perché deve produrre un buon numero di m-ops per le unità a valle. Le m-ops prodotte vengono messe in una coda che prende il nome di instruction pool.

In virtù del n° di unità, all'interno dell'instruction pool ci vanno messe al più 6m-ops per colpo di clock. La profondità dell'instruction pool è di circa 20 m-ops, ma ci sonocasi in cui raggiunge anche 30 m-ops.

La soluzione adottata per far fronte al register renaming è quello di avere 40 registri interni su cui vengono mappati i registri utenti. Il mapping è fatto tramite una tabella di conversione la RENAMING MAP. La Dispatch Unit invia l'istruzione (m-ops prelevata dall'instruction pool) ad una delle 20 reservation station, e ad una delle 40 entry nel ROB: deve essere verificata la condizione che ci sia spazio libero nelle RS di destinazione e nell'entry libera del ROB, altrimenti sappiamo cosa si verifica.

Nell'Execution Unit, ci sono 5 unità funzionali:

- 2 unità per le op FP
- 2 unità per le op Intere
- 1 per la partizione della memoria.

La Retire unit è quella che implementa il ROB, e il commit avviene con una strategia che è strettamente in ordine; quando becchiamo un salto, andiamo a controllare che la previsione sia corretta, e se è corretta si controlla l'eventuale presenza di dipendenze di dato al fine di risultate compatibili con quella tecnica di register renaming che abbiamo accennato prima.

Performances

Adesso vediamo qualcosa sulle performance: nel caso ideale, il CPI del P6 è di 0.33. Ma per arrivare a quel numero dobbiamo ipotizzare che anche l'unità di decodifica per le istruzioni complesse sappia anche decodificare una istruzione semplice. In realtà il CPI reale tende a 2. Vediamo quali sono le ragioni: ci sono una serie di casi in cui non tutto va per il verso giusto:

- Instruction cache miss

- Le istruzioni non sono immediatamente corrispondenti a 3 m-ops, il che significa che non mi trovo sempre 2 IA-16 e 1 IA-32.
- L'unità Dispatcher può bloccarmi o se finiscono le RS oppure se non ci sono entry nel ROB.
- Dipendenze di dato
- Miss sulla cache dati
- Previsioni sballate sui salti.

Ciò che vogliamo evitare è che l'istruzione pool sia vuoto. Mediamente, l'intera unità di decodifica lavora ad un'istruzione per colpo di clock. I casi in cui l'unità di decodifica non produce nulla sono i seguenti:

- Dallo stadio di fetch non arriva niente
- Lo stadio a valle non è in grado di lavorare al meglio: questo non significa che l'istruzione pool è piena, ma solo che esiste uno sbilanciamento notevole tra le 3 unità di decodifica e del carico di lavoro che riescono a smaltire.

La cosa curiosa, riguardante il miss della cache, è che siamo intorno allo 0.9 come hit ratio, il che significa che non mi sta lavorando male. Diciamo che è buona la distribuzione di un miss tra le varie cache, perché vario tra un worst case di 0.84 ad un caso ottimale di 0.99.

Un miss tra le cache mi costa 5 volte un miss sulla L1 cache.

Altra cosa da tenere in considerazione è che in 8 casi su 9 la previsione sul salto risulta essere azzeccata: diciamo che quando l'indirizzo è presente nel Branch Target Buffer. Allora azzecco la previsione.

The NetBurst Architecture

NetBurst è il nome della microarchitettura del Pentium 4. Diciamo che non è molto diversa dalla P6, ma c'è un salto, che consiste in una serie di modifiche alcune significative, altre no.

L'obiettivo è andare più veloce aumentando quel CPI che come abbiamo visto non è entusiasmante. Non basta cambiare tecnologia per andare più veloce, ma bisogna anche cambiare l'architettura in modo tale da rendere gli stadi più semplici che mi permettono di aumentare il clock rate ovvero la frequenza.

La novità è che da P6 a NetBurst il numero degli stadi aumenta, ma questa tendenza ad un certo punto si blocca e si inverte. Sebbene questo possa sembrare un controsenso, in realtà dobbiamo anche considerare che aumentando il n° di stadi aumenta anche la probabilità di avere degli stalli.

Comunque, sappiamo che con il NetBurst il numero di stadi nella pipeline aumenta. Anziché dire quali sono diciamo le differenze che sussistono tra P6 e NetBurst:

- Come già detto, pipeline più profonda
- Il NetBurst usa register renaming con 138 entry anziché usare ROB con 40 entry usato nel P6
- Nel NetBurst aumentano le unità intere, passando dalle 5 unità funzionali del P6 alle 7 unità, di cui una è una unità ALU intera, e un'unità per il calcolo degli indirizzi.
- Agisco sulla ALU, che riesce a smaltire 2 istruzioni per colpo di clock, e migliorano anche la cache dati.
- Al fine di migliorare le prestazioni del fetch uso una cache particolare che diamo trace cache.
- Estendo la dimensione del Branch Target Buffer, facendo 8 volte più grande e passando quindi da 512 a 40696 entry, al fine di migliorare la possibilità di ottenere previsioni
- Aggiungo altre istruzioni floatin-point al fine di ottimizzare il settore grafico.

Nel confronto tra PIII e PIV:

- Nel settore delle aritmetiche intere ho un miglioramento di 1.2, che è un miglioramento per modo di dire perché ho un processore a 1.7 GHz che va a 1.2 volte più volte più veloce di un processore a 1 GHz, per cui non recupero.
- Nel settore delle aritmetiche floating-point invece vado 2.9 volte più veloce, quindi questo mi consola dalle prestazioni non straordinariamente migliori del settore intero.

A questo punto la domanda che bisogna porsi è: ma il mio processore, dove e come verrà usato? Ad ogni tipologia di mercato corrisponderà una risposta.

VLIW (Very Long Instruction Word) Processors (orientati al settore special purpose)

Nascono da un'osservazione obbiettiva: quanto sono incasinati i processori superscalari. Attenzione che l'impresa non è solo nel progettare il processore superscalare, ma anche andando a testare prima di mandarlo sul silicio. E' un'impresa titanica, che fa sì che prima che Interl possa arrivare al chip fa un respin, ovvero si occorre che c'è un bug, torna indietro, lo corregge e riparte col test. Questo avviene mediamente 6 o 10 volte, e costa molto.

A questo punto si chiede: ma perché non mettiamo tante unità funzionali in parallelo, cercando di sgravare la parte di controllo? L'idea dei VLIW è quella di aumentare il numero di unità funzionali, e si può arrivare tranquillamente anche a 10 unità funzionali molte delle quali tutte uguali tra loro. Dopodiché queste unità lavoreranno tutte in parallelo. Il nome deriva dal fatto che il compilatore deve trovare a questo punto dei pacchetti di istruzioni che mi permettono di sfruttare al max tutte le unità funzionali. Ma voglio di più: creo una mega istruzione, che ingloberà tutte quelle istruzioni che dovevano rientrare nel pacchetto, e le mando alle unità funzionali che le lavoreranno tutte in parallelo. Il tipico VLIW ha un certo numero di unità funzionali, più o meno uguali fra loro.

Possiamo vedere l'istruzione di un VLIW come la giusta opposizione di istruzioni di un RISC. Ovviamente questo complica il lavoro del compilatore. E' scontato che ad ogni colpo di clock non è detto che trovi un numero di istruzioni compatibile per far lavorare tutte le unità funzionali, ma in quel caso potrà accedere solo che alcune unità non lavorino; devo comunque far sì che tutto questo accada il meno possibile. Cosa ho ottenuto? Ho complicato il compito del compilatore, ma ho semplificato di gran lunga l'hardware! Nell'HW di un VLIW non viene fatto alcun controllo sulle possibili dipendenze tra le istruzioni. La vera sfida è far sì che vengano minimizzati gli stalli. E' il compilatore che deve trovare la giusta combinazione di istruzioni (all'interno della mega-istruzione) che minimizzino gli stalli: tipico esempio, il miss sulla cache. Se una delle 10 operazioni nella fase di scrittura dei risultati fa un miss nella cache dati mi si stalla tutto fino a che quel miss non è gestito: si ferma tutto tranne quelle avanti alla stallata. Vediamo un esempio: VLIW con parallelismo 5, ovvero ho 5 unità funzionali organizzate nel modo che vedremo qui di seguito:

- 2 memory reference
- 2 FP operations
- 1 unità intera o branch

Tra tutti i tipi di hazard, l'unico che viene gestito è l'hazard di controllo, non perché si fa speculazione ma perché si controlla se la predizione era giusta o sbagliata e nel caso in cui era sbagliata si svuota la pipeline.

Nota: Qui fino a quando i dati non sono disponibili nel Register File io aspetto, perché non voglio rotture di scatole con il CDB etc. etc.

Nei VLIW ho bisogno di una enorme quantità di registri per gestire:

- Loop unrolling
- Renaming
- Gestione degli hazard

Nota di confronto: per talune applicazioni i processori VLIW riescono ad avere prestazioni migliori rispetto ai superscalari visti prima, pur avendo una minore complessità dell'hardware ed un minore consumo di potenza! Ecco dunque dove sta il motivo della loro esistenza. Li si usa come detto in applicazioni special purpose, dove si sfrutta molto il calcolo parallelo, deve esserci un basso consumo di potenza a fronte anche di una minore quantità di silicio richiesta.

Facciamo 2 considerazioni:

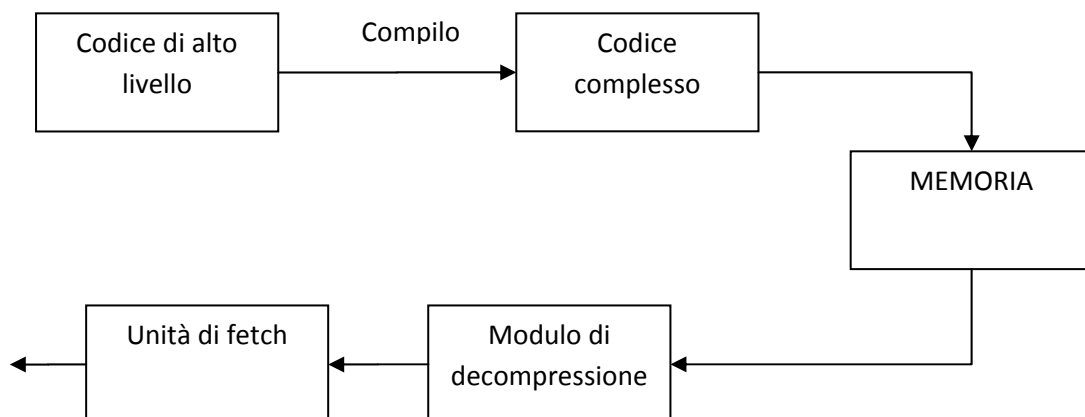
- Supponiamo di avere 5 unità funzionali. Non tutte le unità funzionali hanno bisogno di un indirizzo perché magari operano su un registro o un immediato quindi potrò risparmiare qualcosa.
- Inoltre, dato che ho differenti unità funzionali, a seconda della posizione all'interno delle micro-istruzione avrò un diverso codice operativo, e quindi risparmierò anche sui codici operativi.

Con 5 unità funzionali al fine di controllare ciascuna di esse avrò tra i 16 e i 24 bit. Questo mi farà avere delle istruzioni di lunghezza media di 150 bit. Il problema è che di questi 150 bit la metà non mi serve (abbiamo visto che 2.5 unità lavorano un colpo di clock sì è un colpo di clock no), quindi ci saranno in quella metà di bit valori che non mi servono. Quindi spreco bit inutilmente, e questa è una delle prime limitazioni.

Altra limitazione sta nell'accesso in memoria: una delle soluzioni a questo problema sta nell'usare l'interleaving, cioè aumento il parallelismo in accesso alla memoria. La dimensione del codice nei VLIW è di gran lunga più grande, e questo accade principalmente per due motivi:

- I loops subiscono un unrolling statico
- ci sono buchi vuoti nelle istruzioni

A questo punto però abbiamo una domanda: ma se abbiamo detto che i VLIW trovano spazio in applicazioni embedder come è possibile che la dimensione del codice sia grande? La memoria è infatti un punto fondamentale nelle applicazioni embedde, e non mi posso permettere di sprecarla! Ecco perché abbiamo il seguente schema:



In sostanza a monte dell'unità di fetch ottengo un modulo di decompressione che serve a scompattare il codice compresso che era stato generato all'atto della compilazione. I VLIW non sono compatibili con nulla se non con se stessi, quindi sono stati inventati dei traduttori per ottenere compatibilità con i VLIW.

Abbiamo diverse categorie di VLIW

- Disponibili come core
- Disponibili come stand alone, ad esempio TRIMEDIA TM32 che viene usato in ambiti specifici

Altro campo di utilizzabilità è il Transmeta Crusoe, volto a chi vuole usare il PC in ambito Low-power (es. i PDA). Viene fornito con un traduttore per traslare il codice eseguibile Intel in codice adatto a questo VLIW. Non è un VLIW puro: “ ha due pipeline separate, ma per le istruzioni intere ed una per le istruzioni FP”

Questo la dice lunga sul fatto che la logica di controllo deve esserci al fine di evitare problemi ed ecco perché non è un VLIW puro. Inoltre, evita il consumo di memoria avendo due formati di istruzioni

- Uno a 128 bit
- Uno a 164 bit

Questo mi permette di non sprecare, laddove possibile, i bit per l'istruzione ma complice il fetch è la decodifica perché mi devo rendere conto della lunghezza dell'istruzione che mi trovo davanti.

ARM

È un processore di tipo RISC per applicazioni embedde. La ragione per cui parliamo adesso di ARM è che come CORE è uno dei più diffusi per i così detti SoC (System on Chip). Il suo successo è dovuto alla semplicità, alla robustezza ma soprattutto ai bassi consumi energetici. ARM non vende processori ma i core, ovvero moduli (descritti in linguaggi come VHDL o Verilog) che poi vengono acquistati dai produttori di processori. Di core se ne distinguono due tipologie:

- Hard-core: sono ottimizzati da ARM, ma è vincolato ad una certa tecnologia, il che significa che se li usiamo e cambiamo tecnologia lo dovremmo buttare via (questi moduli sono pre-sintetizzati ed ottimizzati), si parla di mogulismo HW. In questo caso l'ARM fornisce un Layout fisico, ottimizzato per una data tecnologia.
- Soft-Core: non sono ottimizzati, ma godono del vantaggio che non sono vincolati ad una determinata tecnologia, per cui risultano facilmente riutilizzabili. In questo caso le ottimizzazioni le dovremmo fare da soli. In quest'altro caso l'ARM fornisce una descrizione di alto livello che può essere sintetizzata dal progettista in qualunque tecnologia.

Vediamo adesso le caratteristiche:

- È un progetto molto semplice: addirittura nel primo caso si avevano 3 stadi e solo successivamente si è passati a 5 stadi di pipeline. Il fatto di avere un progetto semplice rende meno probabili gli errori e i bug, è portabile e facilmente applicabili.
- È di tipo RISC, con architettura LOAD/STORE
- Ha un formato fisso per le istruzioni a 32 bit
- 3 formati degli indirizzi (modi di indirizzamento)

I registri sono 32 bit e si distinguono in registri utente e registri di sistema. I registri che l'utente vede dipendono dal modo di funzionamento (che sono 7 in tutto). I registri utente sono 16.

Adesso vediamo il registro di stato CPSR (Current Program Status Register):

| | | | | | | | | |
|---|---|---|---|--------|---|---|---|------|
| N | Z | C | V | UNUSED | I | F | T | MODE |
|---|---|---|---|--------|---|---|---|------|

Flag di condizione

Flag di stato

Modo di
funzionamento
corrente

Anche questo è a 32-bit. Degno di nota è il bit "T" che è modificabile via software e permette di codificare le istruzioni nel formato Thumb.

Ci sono due modi di funzionamento:

- Utente: è quello usuale
- Privilegiati: usati per gestire le eccezioni e le superviso call

Quando avviene il salto di contesto (es. interrupt), si salva il contesto andando a salvare il CPSR nel registro SPSR (il che velocizza la chiamata alla routine di servizio evitando l'accesso in memoria). Per semplicità, il meccanismo di gestione dell'I/O è di tipo MEMORY MAPPED, ovvero si mappano la parte dei registri in memoria. Le eccezioni si dividono in 3 categorie:

- Eccezioni diretto effetto di un istruzione
 - Interrupt software
 - Undefined instruction
 - Pre-fetch abort: fallisce il fetch

Nota: con il termine eccezioni si intendono anche gli interrupt, le trap e le supervisor call

- Eccezioni come effetto collaterale di un istruzione
 - Data abort: ad esempio il fallimento di una LOAD/STORE durante l'accesso in memoria
- Eccezioni generate dall'esterno
 - Reset
 - FIQ più urgenti degli IRQ, simili agli interrupt non mascherabili
 - IRQ interrupt classico

Tra le eccezioni esistono delle priorità al fine di gestire anche le richieste simultanee da parte di più eccezioni.

- Reset + alta priorità
- Data Abort
- FIQ
- IRQ
- Prefetch
- SWI and undefined instruction

Vediamo cosa succede quando arriva un'eccezione:

- Si decide quale gestire in base alle priorità (viste in precedenza)
- Si salva il PC e il CPSR (sui registri per avere max velocità): PC in R15, mentre CPSR in SPSR
- Si cambia il modo operativo a seconda del tipo di eccezione
- Si forza il valore (da 00 a 1C del vector address) del PC a seconda del tipo di eccezione

Il punto in cui accedo all'interno del vettore delle eccezioni non mi è comunicato dall'interrupt ma dal tipo di eccezione, senza bisogno di ricevere altro dall'esterno. All'interno del vettore non ci sono degli indirizzi bensì delle istruzioni (ad esempio un'istruzione di salto alla routine di gestione).

NB: Si può ottimizzare l'uso del vettore delle eccezioni facendo sì che per le eccezioni con più alta priorità ci sia all'interno della locazione a cui accedo un certo numero di istruzioni che si sovrappongono ad altrettanti elementi: ovviamente gli elementi in questione non saranno accessibili dall'esterno ma solo tramite l'istruzione di partenza.

I core ARM hanno avuto successo perché forniscono, per il core anche un ambiente di supporto e molti tool per simulare gestire i Core. Tutti gli strumenti che vengono forniti sono fondamentali e sono anche il successo dell'ARM.

In particolare, i due strumenti più importanti sono il simulatore le schede di sviluppo che permettono di simulare e debuggare il progetto ancora prima di mandare in produzione il chip.

Adesso vediamo il set di istruzioni ARM, che risulta composto dalla seguente tipologia di istruzioni:

- Trasferimento dati
- Processamento dati
- Controllo

Per le istruzioni di processamento, lavorano su operandi su 32 bit, e forniscono un risultato su 32 bit. I 2 operandi e il risultato sono specificati in modo lineare e indipendente all'interno dell'istruzione, raggruppando un forte criterio di ortogonalità.

È un RISC come architettura è formato dalle istruzioni, ma non come instruction set: non esiste a bordo di queste macchine, un motivo per limitare il numero delle istruzioni che qui risulta altro. Ciascun operando può essere shiftato prima di essere adoperato: il meccanismo, dal punto di vista del codice macchina provoca alcune cose che vedremo- Un esempio può essere:

```
ADD r3,r2,r1,LSL#3      r3:=r2+8*r1
```

Otengo un'accelerazione perché faccio più operazioni all'interno di una sola istruzione. L'operazione di shift si distingue in vari modi tutti differenti tra loro: attenzione, quando scegliamo.

Ciascun istruzione può essere trasformata in una operazione condizionale e questo può essere utile perché se svolgendo determinate operazioni ottengo una certa configurazione dei flag posso decidere di non toccarli più. Sempre con riferimento alle istruzioni di processamento dati, posso vederne il formato:

figura...

Adesso andiamo a vedere le istruzioni di trasferimento dei dati. Posso andare a distinguerne 3

- Operazioni su singoli registri
- Operazioni su registri multipli

- Swap

Tutti i modi di indirizzamento sono basati sul register indirect, ma da notare che l'autoincrement è supportato.

Figura.....

Ancora una volta ci sono due possibili valori per questo campo:

- 0:
- 1:

Adesso vediamo le istruzioni di salto, che si distinguono in:

- Istruzioni di salto vere e proprie, ovvero i branch condizionali e non
- Istruzioni di branch and link

Come già fatto, tutte le istruzioni possono trasformarsi in istruzioni condizionali. I primi bit sulla sinistra di tutte le tipologie di istruzioni vengono usati per capire se l'istruzione vada eseguita comunque oppure se l'istruzione è vincolata alla verifica di una certa condizione, e in tal caso quale condizione deve essere controllata. Grazie a questa possibilità, posso eliminare i costrutti condizionali (ovviamente solo nel caso di valutazioni di condizioni semplici non annidati).

L'esecuzione di un'istruzione vincolata come nel modo appena detto non introduce overhead.

Nei branch and link ci colleghiamo a quelle che sono le CALL.

L'indirizzo dell'istruzione successiva al ritorno è memorizzato in R14. Ovviamente nel caso in cui ho delle call annidate dovrò salvarmi opportunamente gli indirizzi al fine di poter effettuare i ritorni correttamente senza perdere nulla. Il formato delle istruzioni di controllo è il seguente:

FIGURA

Adesso vediamo l'architettura di ARM. Se utilizziamo una nomenclatura diversa quando si parla di CORE e quando si parla di architettura. In particolare nella nomenclatura dei CORE è inserita una serie di sigle inerenti alcune caratteristiche aggiuntive che il core fornisce.

Nell'ARM abbiamo 3 stadi:

- Fetch
- Decodifica
- Esecuzione

Quando si incontra un'istruzione di salto, si svuota la pipeline fino alla verifica della condizione di salto. Non bisogna illudersi delle prestazioni elevate! Soddisfacenti.

Il nucleo dell'architettura è il register Bank. La ALU ha 2 input: uno diretto dal register bank e un altro del register bank che però passa attraverso uno shift.

A partire dall'ARM 9 si hanno 5 stadi:

- Fetch
- Decodifica
- Execute
- Buffer/Data
- Write/Back

Note: l'execute della pipeline a 3 stadi è formata dagli ultimi 3 stadi della pipeline a 5 stadi.

Questa struttura è molto simile all'architettura del MIPS su cui abbiamo lavorato. Ovviamente, in questa nuova architettura vengono introdotte le due cache, una dati e una istruzioni che prima non c'erano.

Quando vediamo dei core che hanno nelle loro sigle la lettera T significa che quel core sarà capace di codificare le istruzioni in due modi differenti: nel caso ARM la decodifica viene fatta, su 32 bit, ma per alcune istruzioni 32 bit sono troppi, quindi nasce il set di istruzioni thumb, che prevede solo 16 bit. Ovviamente non è detto che tutto ciò che codificato su 32 bit lo potrà codificare su 16, ma avrà istruzioni più semplici. Questo instruction set è molto meno potente, sia per i 16 bit invece di 32, sia per il n° di bit da dedicare al codice operativo dato che ho molte meno istruzioni. Processando in thumb il processore diventa un po' più lento, ma richiede un minore consumo di potenza (questo a fronte di un minore numero di accessi all'A-BUS). Inoltre, codificando in thumb, ottengo molte più istruzioni, ma un codice che occupa meno spazio in memoria perché sono istruzioni più corte.

I core che supportano il thumb mi permettono di sfruttare quell' instruction set. Di solito quello che si fa è, se si vuole processare con l' instruction set thumb, e bootstrappare in ARM con 32 bit sfruttando pochissime istruzioni, e poi switchare alla modalità del thumb. Dipende da cosa vogliamo:

- Esecuzione veloce
- Risparmio di memoria (a fronte di un'esecuzione più lenta)
- Risparmio di consumo (a fronte di un'esecuzione più lenta)

Nota: tutto dipende dal valore del T bit nel CPSR (0: ARM instructions, 1: Thumb)

I core che supportano il thumb hanno anche un decompressore che viene fornito incorporato e non rallenta assolutamente le prestazioni.

Adesso caliamoci nel ruolo del progettista. Al fine di sviluppare un progetto di SoC, deve andare ad assemblare un certo numero di pezzi:

- Memory interface
- Bus architecture
- Reference peripheral specification
- Debussing mechanism

Uno degli step che fa diventare più cretini è andare a interfacciare e collegare tutti quei pezzi tra loro, perché se i pezzi sono stati acquistati da fornitori diversi, non è detto che siano pensati per parlare tra loro in modo compatibile. Una delle cose che hanno fatto i progettisti di ARM è stato definire uno standard comune di BUS, definendo:

- Come è fatto il bus dati
- Come è fatto il bus indirizzi
- Come è fatto il bus sistema di interfaccia con la memoria

La definizione di questo standard comune di Bus è stata resa pubblica in modo tale che chi progetta questi moduli da andare ad assemblare tra loro sappia a quale standard deve uniformarsi tale standard va sotto il nome di AMBA. In particolare, l'architettura del BUS seguita da AMBA è un'architettura che si articola su più livelli:

FIGURA

Descrizione: l'AHB è usato per collegare tra loro moduli ad alte prestazioni, e supporta il Burst Mode, per il trasferimento dei dati. Le periferiche più lente invece vengono collegate all'APB, che è più lento rispetto

all'AHB. L'APB viene successivamente connesso all'AHB attraverso un modulo di controllo chiamato bridge che garantisce l'interfacciamento dei due (AHB e APB). Il meccanismo di arbitraggio del BUS è centralizzato da un controller che si basa su diversi livelli di priorità. Inoltre, i progettisti di ARM hanno definito alcune specifiche per tutti coloro che hanno intenzione di andarci a mettere un Sistema Operativo su quel chip. Le specifiche sono state rese standard, e quindi riguardano le seguenti parti:

- Memory map
- Interrupt controller
- Counter time
- Reset controller

Ultimo step da analizzare è andare a capire il meccanismo di debug. Uno dei passi fondamentali è quello di debuggare il chip. Al fine di fare ciò si deve prevedere l'esistenza sul chip di comportamenti che permettono, pilotando con opportuni segnali, di vedere lo stato dei registri e quant'altro, fornendoci una visione completa dello stato interno del chip. Vorremo poter accedere al processore per poter vedere e/o modificare il contenuto dei registri. Per fare ciò si utilizza un'interfaccia, chiamata JTAG. Bisogna che si passi ad una modalità in cui il sistema figura come una catena di flip-flop pilotata da un segnale, in modo che grazie al programm counter mi farà capire cosa c'è all'interno dei registri (ovviamente quelli di ARM mi dicono in che ordine sono i registri e mi fornisce gli strumenti software per poterci lavorare). L'interfaccia che mi permette di fare ciò è la porta JTAG. Altro strumento fondamentale è l'embedded ICE, che mi permette di implementare i breakpoint. È un modulino presente sul BUS che controlla l'indirizzo che transita sul bus per accedere ad un'istruzione e lo confronta con l'indirizzo dell'istruzione passata. Quando il match viene verificato, ecco che scatena un'eccezione per bloccare tutto (ovviamente blocca tutto prima del fetch dell'istruzione su cui abbiamo collocato il breakpoint). L'ultimo componente che andiamo a guardare è il EMBEDDED TRACE MACROCELL che permette di monitorare per un certo intervallo di tempo il bus al fine di sapere anche come cambiano i valori di determinati registri.

Molto spesso i progettisti non necessitano del Core, quanto più di una intera CPU. Per far fronte alle richieste di tutti questi tipi di progettisti, l'ARM fornisce dei moduli già pronti dotati di Memory Management Unit, cache e quant'altro.

Introduction to x86 systems

Il primo Computer Personale (PC) è stato venduto da IBM nel 1981.

Le sue caratteristiche principali erano:

- 16-bit microprocessor (8088)
- 4.7 MHz frequency
- 64 Kb RAM
- Drivers for diskette (360 Kb)
- Cassette recorder (optional)
- Black and white monitor.

L'8086 viene introdotto nel 1978 come successore del 8080 8-bit; composto da 29000 transistors, un'architettura da 16-bit, 123 istruzioni (incluse quelle per la manipolazione delle stringhe), e 2 modi operativi (massimo e minimo).

Minimum and maximum mode

Il modo operativo è selezionato dal valore dato al pin (MN/MX):

- In minimum mode, l'8086 genera direttamente i segnali di controllo necessari per il bus
- In maximum mode: il processore genera solo segnali di uscita per il bus controller, che genera i bus control signal; questa configurazione supporta sistemi più complessi, dove i segnali del bus devono essere guidati con sufficiente potenza; usando il MULTIBUS molti processori possono essere combinati in un singolo sistema

•

Real Mode

Each time memory has to be accessed, two 16-bit values are combined, and a 20-bit address is generated.

Memory Access

8086

L'8086 multiplexa bus dati e indirizzi; i cicli di lettura e scrittura sul bus sono lunghi 4 clock cycle; un altro ciclo di clock può essere dinamicamente aggiunto se richiesto, in base al valore del segnale di READY.

L'8086 può accedere alla memoria in diversi modi:

- Accedere alla singola word o byte se l'indirizzo è pari
- Accedere a bytes se l'indirizzo è dispari

Il segnale BHE è usato per gestire la grandezza dei dati da trasferire.

8088

È molto simile all'8086 ma il suo data bus è di 8 bit. Quando accede a indirizzi pari di words richiede 2 cicli di bus invece che uno, per questo è più lento dell'8086.

80186/80188

Hanno i microcontrollori derivati da 8086/8088; l'istruzione set è più grande e più ottimizzato, così la velocità è del 25% più alta. Sono stati usati raramente nei PC, ma sono relativamente popolari per i sistemi special purpose.

80286

Prima realizzazione nel 1982, composto da 132000 transistor e microprocessore a 16 bit; supporta il protected virtual address mode; 24 bit per il bus di indirizzi e 16 per quello dati; alta frequenza di lavoro (25MHz), accesso a memoria ottimizzato, minor numero di cicli per eseguire un'istruzione.

Grande successo commerciale.

80386

Prima uscita nel 1985, composto da 275000 transistor; microprocessore e registri da 32 bit mentre la cosa di pre-fetch è da 16 byte. La Memory Management Unit (MMU) supporta paginazione e segmentazione; i bus dati e indirizzi sono da 32 bit e può lavorare in real, protected e virtual mode.

Tutti i segnali di controllo del bus sono generati dal **bus controller**, che legge lo stato dei segnali generati dal processore. Il bus controller ha anche un numero di buffers.

Ogni ciclo di bus richiede almeno 2 PCLK cicli (corrispondente a 4 cicli di clock) formato da:

- C1(status cycle): sono emessi i segnali di controllo e di indirizzo
- C2(command cycle): il trasferimento dati è eseguito

A 40 MHz l'80386 compie 20 milioni di cicli di bus al secondo. Dal data bus a 32 bits, la massima ampiezza di banda è 80Mb/s.

Stato d'Attesa

Se la memoria o il chip periferico non conclude una lettura o una scrittura entro i 2 PCLK, il memory controller mantiene il segnale di READY alto e l'80386 implementa un altro ciclo di C2. Questo meccanismo è ripetuto fino a quando il memory controller non abbassa il segnale di READY.

Address Pipelining

Per dare più tempo a memoria e porte, l'80386 può emettere l'indirizzo per un accesso durante un ciclo C2 al ciclo di bus precedente. In questo caso l'indirizzo è trasferito alla logica di codifica degli indirizzi quando la memoria compie il trasferimento dati. Per questo ogni ciclo di bus dura per 3 PCLK cicli, ma l'effetto è che quello che occupa due cicli di PCLK. L'address pipelining è attivato dal segnale di controllo NA. Questa tecnica permette di anticipare la generazione degli indirizzi, permettendo alle memorie di avere un ciclo più lungo a parità di ciclo di bus.

Nell'80386 la RAM è normalmente organizzata come una memoria a 32 bit, e può leggere/scrivere parole (word) da/per il data bus in un ciclo di bus. Comunque, in alcuni casi, è necessario leggere/scrivere dati con minore lunghezza. I segnali da BE0 a BE3 informano quali sono i bytes significativi. Se bisogna accedere a un double word non allineata, il processore automaticamente divide l'operazione oltre i due cicli di bus, generando i corretti segnali di indirizzo e di controllo.

Il 386 è abilitato a supportare contemporaneamente il trasferimento dati a 16 e 32 bit. (ad esempio il processore può comunicare a 32 bit con la RAM e a 16 bit con i periferici).

Usando il pin BS16, il singolo ciclo di bus può implementare sia un singolo trasferimento dati a 32 bit che una coppia di 16 bit.

80386 supporta sia l'**isolated I/O** che il **memory mapped I/O**; nel primo caso lo spazio di I/O occupa gli indirizzi da 0 a 65536. Il ciclo di I/O è identico al ciclo di memoria. I **registri** mantengono la compatibilità con le versioni precedenti di processore e per questo, oltre a funzionare a 32 bit, possono essere usati anche a 16 bit.

The Protected Mode

A partire dall'80286 i processori Intel supportano 2 modi:

- Real Mode
- Protected Mode

Il Virtual Mode è stato introdotto a partire dal 80386.

Real Mode

Permette l'accesso alla memoria come nel 8086; l'indirizzo effettivo è calcolato attraverso la formula $offset + segment_register \times 16$

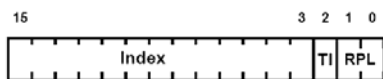
Protected Mode

E' stato introdotto per due scopi:

- Per prevenire i diversi task in un sistema operativo multitasking dal compiere accessi invalidi o non corretti
- L'effective address è calcolato in un modo differente

Un **Task** è la combinazione di un programma, i dati e le necessarie funzioni del sistema.

Nel modo protetto i valori nel registro di segmento rappresentano i **selettori** (e non l'indirizzo base).



Le informazioni fornite dai selettori sul segmento riguardano l'accesso alle tabelle dei descrittori locali e globali, e l'identificazione del livello minimo di privilegio richiesto per l'accesso al segmento Request Privilege Level (RPL).

Il valore del campo RPL del **segment selector** del code segment ad un dato tempo è il Current Privilege Level (CPL). Il programma attivo può accedere solo ai segmenti di dato che hanno un livello di privilegio uguale o maggiore del CPL. Il maggiore tra CPL e RPL è l'**Effective Privilege Level (EPL)**. (se il CPL=2 il task può accedere a dati e fare salti a istruzioni con RPL di 2 o 3).

Ci sono **4** livelli di privilegio:

- Livello 0 (massimo) è per il kernel del S.O.
- Livello 1 per le funzioni del S.O.
- Livello 2 per le funzioni minime critiche del S.O.
- Livello 3 per i programmi applicativi

A partire dall'80386, c'è uno stack separato per ogni livello di privilegio del task.

Descriptor Table

La memoria riserva:

- **Global Descriptor Table (GDT)**: che contiene i descrittori dei segmenti disponibili per tutti i task
- **Local Descriptor Table (LDT)**: contiene i descrittori dei segmenti disponibili solo dai task attivi attualmente

Ogni segment selector si riferisce o a GDT o a LDT, a seconda del valore del bit TI (0 → GDT, 1 → LDT).

Il **segment descriptor** è sui 64 bit.

Ogni descrittore contiene:

- L'indirizzo base per il segmento (32 bit)
- Il limite (20 bit) il cui significato dipende dal valore del bit **G**: se G=0 (granularità di byte) il limite è la grandezza del segmento in byte (la massima grandezza del segmento è 2^{20} bytes); invece se G=1 (granularità di pagina) il limite è la grandezza del segmento in pagine (la grandezza massima è $2^{20} \times 4Kb = 4Gb$)
- **DT**: definisce il tipo di segmento (se DT=0 è un segmento di sistema (gate, interrupt e trap), altrimenti è di applicazione)

- **DPL**: indica il livello di privilegio del segmento
- **P**: indica se il segmento è in memoria o no, se si accede a un segmento con P=0 il processore genera un'eccezione di segmento non disponibile.
- **vf**: è un campo disponibile
- **DB**: forza il processore a lavorare con operandi a 16 o 32 bit; con il 80386 si possono cambiare anche gli operandi da 32 a 16 bit.

Quando si carica in memoria un task, si assegna un valore di DPL ai suoi segmenti, e il RPL nei selettori usati dal task.

Registri

- **IDTR**: indirizzo base e limite della IDT
- **GDTR**: indirizzo base e limite della GDT

Da quando ci sono tante LDT, i loro segment descriptor sono memorizzati nella GDT

- **LDTR**: offset del descrittore di LDT all'interno della GDT
- **TR**: TSS selector per il task corrente
- **CR0**: immagazzina le informazioni di base del comportamento del processore; dal cambiamento del contenuto si può cambiare il comportamento del processore

Per ogni task il S.O. assegna la GDT e la LDT; GDTR e LDTR sono aggiornate dal kernel del S.O. usando le istruzioni LGDT e LLDT.

Il bit meno significativo di CR0 (PE) determina se il processore lavora in real (PE=0) o in protected (PE=1) mode. Per passare in protected mode si può operare in diversi modi:

- Attraverso le funzioni INT 15h 89h
- Usando le istruzioni LMSW (Load Machine Status Word)
- Usando le istruzioni MOV CR0,imm

Memory Addressing in Protected Mode

Il processore:

- Accede al segment descriptor e controlla quale deve usare tra GDT e LDT
- Recupera da GDTR o LDTR l'indirizzo base della corrispondente tabella dei descrittori
- Moltiplica l'indice del selettore per 8, e somma questo valore all'indirizzo base (se il risultato è più grande del limite ci sarà un'eccezione di *general fault exception*; altrimenti il segment descriptor da la base e il limite del segmento)
- Somma la base all'offset (controlla che sia più basso del limite)
- Accede alla memoria

Grazie alla segment descriptor cache register questa procedura è eseguita solo per l'accesso a istruzioni di nuovi segmenti; per molti accessi in memoria il calcolo dell'indirizzo è veloce come nel real mode.

Quando succede una chiamata, un salto o un interrupt, non cambia solo il valore di EIP ma anche il code segment. In real mode questo significa che cambia il valore di CS. In protected mode questo comporta: se il target segment ha lo stesso, o più alto, PL di quello corrente, la CPU carica il target segment selector nel registro CS; invece se ha PL minore la chiamata può essere eseguita solo attraverso le *call gate*.

Gates

Permettono di saltare in altri segmenti con differenti livelli di privilegio; in questo caso il segment selector non punta direttamente all'istruzione puntata dal segment selector, ma punterà alla *call gate*.

Il tipo e il campo DT nel target segment descriptor lasciano al processore la conoscenza se è un gate descriptor o un segment code.

Le Gates sono definite da DT=0 nel segment descriptor e valori da 4 a 7 e da 12 a 15 nel campo type.

Call Gates Descriptor

- Target Segment Selector: identifica il segment descriptor nella GDT o LDT; è il segmento contenente l'istruzione di target jump
- Offset: offset nel target segment
- Dword-count: forza il processore a trasferire double word allo stack della chiamata a procedura che ha fatto richiesta
- Type: deve essere 12 per le call gate

I gates sono importanti per limitare gli effetti di salti incorretti; se il salto ad un punto sbagliato viene fatto il sistema crolla; mentre se viene fatto un salto ad una gate non corretta abortisce solo il task interessato.

Sia le call che i salti possono ricorrere alle call gates. Comunque le istruzioni call possono usare le call gate per accedere a procedure appartenenti a segmenti di codice con PL più alto, come le istruzioni di salto per accedere a istruzioni con PL più basso o uguale.

Ogni livello di privilegio ha il suo stack. Quando una chiamata attraverso una gate è eseguita, il processore automaticamente trasferisce dallo stack del chiamato all'altro molte double word quante indicate dal campo DWord-count.

Nel real mode, le chiamate per interrupt sono gestite attraverso l'**Interrupt Vector Table (IVR)**; per ogni tipo di interrupt l'IVR contiene 4 bytes (2 per EIP e 2 per CS). In protected mode i salti a interrupt succedono attraverso la **Interrupt Description Table (IDT)** che può essere acceduta attraverso IDTR. La IDT contiene solo i descrittori delle interrupt gates; ogni descrittore è lungo 8 byte. La lunghezza della IDT può essere modificata cambiando il valore del campo limite (la massima grandezza è 256x8 bytes); la IDT può essere localizzata ovunque in memoria.

Gli **Interrupt Gates** sono simili alle call gates ma il campo type contiene il numero 14 e il campo DWord-count non ha significato.

Multitasking

E' basato sull'aver diversi task che vengono svolti in parallelo. Periodicamente, ogni task attivo è attivato e svolto per un breve periodo di tempo, e poi viene interrotto. Dopo un altro piccolo periodo di tempo viene ripreso dallo stesso punto in cui è stato interrotto; è richiesto un meccanismo per la gestione e il salvataggio degli stati dei tasks. Tutte le informazioni riguardanti i task sono memorizzate nel **Task State Segment (TSS)**, che è grande 104 bytes.

Quando accade che viene cambiato task, il processore deve automaticamente salvare tutte le informazioni del task corrente nel TSS identificato del **Task Register (TR)** e caricare i valori del task da far partire in segmento, offset e registri di controllo.

Quando un processore incontra un task gate, durante l'esecuzione di una chiamata, un salto o un interrupt deve:

- Memorizzare le condizioni correnti del task attivo nel TSS identificato da TR
- Scrivere il valore 1 (80286) o 9 (80386) nel campo type nel vecchio TSS descriptor; in questo caso il TSS è identificato come TSS disponibile
- Caricare in TR il nuovo TSS
- Leggere l'indirizzo base, limite e privilegi di accesso da LDT o GDT
- Indicare che il nuovo TSS è occupato scrivendo nel campo type 3 o 11
- Caricare il valore per CS e EIP per il nuovo TSS

Quando il S.O. crea un nuovo task deve assegnargli task gate, TSS e TSS descriptor.

Nei veri sistemi multitasking è il S.O. a decidere quando eseguire i task; le applicazioni non influenzano questo processo (**preemptive multitasking**). Al contrario in Windows 3.x era l'applicazione a decidere quando trasferire il controllo su un altro task (**non-preemptive multitasking**). Da Windows 95 entrambi i modi sono implementabili, dipende da come è scritta la specifica applicazione.

Virtual 8086 Mode

Sarebbe bello permettere a vecchi programmi sviluppati in real mode di essere eseguiti in ambiente multitasking in parallelo ad altri programmi eseguiti in protected mode.

L'hardware della CPU e il monitor lasciano vadere al programma un ambiente sviluppato per:

- Real mode per il calcolo degli indirizzi
- 1 Mb di memoria
- Serie di registri virtuali
- Accesso alle funzioni del sistema come se fosse il solo programma ad essere eseguito

Ogni macchina virtuale subisce lo switch dei task, così si implementa il multitasking.

In virtual mode il calcolo degli indirizzi è come in real mode. Con l'offset a 32 bit, per avere lo stesso comportamento, bisogna fare un controllo e scatta un'eccezione se l'offset è maggiore di ffffh.

Il processore passa al virtual mode quando il bit VM nel registro EFLAG è settato. Il bit VM può essere settato solo da:

- Codice con livello di privilegio uguale a 0
- Un task commuta attraverso un TSS
- Un'istruzione IRET che carica il nuovo valore di EFLAG dallo stack

Il processore deve lavorare già in modo protetto.

Il programma che lavora in virtual mode può necessitare di chiamare funzioni del S.O., ma il S.O. sulle macchine è diverso ed è condiviso da altri task; quindi 8086 è eseguito come parte di un programma 8086, oppure il S.O. emula il S.O. 8086.

Paging

Meccanismo usato dall'80386 per generare indirizzi di memoria lavorando in protected mode con **indirizzi lineari** a 32 bit. L'indirizzo lineare deve essere convertito in **indirizzo fisico** tenendo in considerazione la grandezza della memoria RAM disponibile e la posizione in memoria del segmento; questa conversione è effettuata attraverso il meccanismo di paginazione.

Il mapping tra indirizzo lineare e fisico è fatto attraverso un'unità chiamata **pagina**. Dall'80386 le pagine hanno dimensione di 4kb. Lo spazio di indirizzamento totale è diviso in 2^{20} pagine; solo alcune di queste sono in memoria principale ad un dato tempo, le altre sono in memoria secondaria. Il bit PG nel registro CR0 determina se il processore può implementare il meccanismo della paginazione o no.

La conversione degli indirizzi avviene utilizzando un meccanismo a due livelli:

- I 10 bit più significativi dell'indirizzo sono usati come offset per accedere alla *page directory* (puntato dal registro CR3)
- I seguenti 10 bit sono usati per accedere ad una entry della *page table* identificata da una entry nella *page directory*; con questo si ottiene l'indirizzo di pagina fisica
- L'indirizzo di pagina fisica è combinato con l'offset corrispondente ai 12 bit meno significativi dell'indirizzo

Se il bit P nella pagina acceduta dalla table entry è 0 la pagina non è in memoria principale e viene scatenata un'eccezione di **page fault** e il S.O. è chiamato a portare la pagina richiesta in memoria principale. Bisogna vedere quale pagina è migliore per toglierla dalla memoria, si può usare il bit A per indicare quali pagine sono state accedute nell'ultimo periodo.

TLB

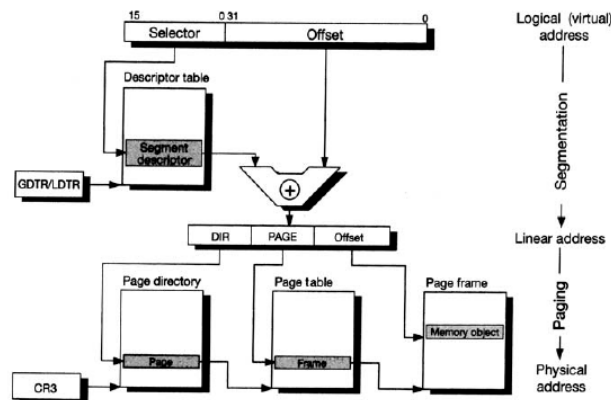
Per sostenere la traduzione degli indirizzi viene implementata una cache dove memorizzare le ultime page table entries usate; questa prende il nome di **Translation Lookside Buffer (TLB)**. Ad ogni traduzione dell'indirizzo si accede prima alla TLB e solo se accade un miss inizia la procedura per la traduzione dell'indirizzo. La TLB nel 80386 è a 4 vie set-associativa, e usa una strategia random per la sostituzione delle pagine.

La TLB contiene 4 set ognuno include 8 linee. Ogni linea è composta da 24 bit di campo informazione e 20 di campo dati. I campi etichetta includono i 18 bit più significativi (2 sono impliciti), il bit di validità e 3 bit attributo. Con la TLB la traduzione può avvenire molto in fretta senza fare accesso alla memoria principale. Le pagine possono essere protette specificando il livello di privilegio richiesto ai task per accedervi; possono essere:

- User: possono essere acceduti da tutti i task
- Supervisor: possono accedere solo i task con $CPL < 3$

Ogni page table entry ha un bit (**D Dirty Bit**) che viene settato quando la pagina è acceduta per un'operazione di scrittura. Se $D=1$ quando la pagina viene rimossa dalla memoria, il suo contenuto deve essere copiato alla memoria secondaria, per copiare i cambiamenti avvenuti durante la sua presenza in memoria principale.

Le pagine possono essere protette contro operazioni di scrittura forzando il bit R/W a 0 nella corrispondente page table entry.



Cache

La cache (corrispondente alla fast DRAM) è spesso inserita tra la CPU e la più lenta (ma grande) memoria principale (corrispondente a SRAM). La cache è normalmente controllata da un **cache controller**. A seconda della CPU il cache controller, o cache controller e cache, sono on-chip.

Cache Controller

Gestisce tutte le operazioni della cache, incluso in caso di miss le operazioni corrispondenti a:

- Possibilità di scrivere le linee della cache per sostituire la memoria
- Aggiornare la nuova linea di cache dalla memoria
- Generare i segnali READY richiesti dalla CPU

Il riempimento delle linee di cache trae vantaggio dall'efficienza del modo di trasferimento a burst.

Quando un'operazione di scrittura è fatta sulla cache ci sono due meccanismi possibili:

- **Write-Through**: l'operazione di scrittura è fatta sia sulla cache che sulla memoria principale. Per evitare di bloccare la CPU durante questa operazione si può usare una scrittura su buffer. Assicura coerenza nel sistema multiprocessore.
- **Write-Back**: l'operazione è fatta solo sulla cache dati. La cache dati è scritta in memoria quando avviene un cache miss e la linea di cache deve essere sostituita oppure all'esecuzione di un'istruzione WBINVD (write back and invalidate cache) oppure è stato attivato il segnale di FLUSH. Si hanno operazioni di cache miss più lente ma accessi a cache più veloci.

Quando un'operazione causa un cache miss:

- La cache scrive in memoria e la CPU riparte, mentre il cache controller riempie la linea di cache (**write-allocate**)

- Oppure la cache scrive in memoria ma il contenuto della cache non è aggiornato. Questo ha come conseguenza un basso hit-ratio ma semplifica il cache controller.

Se qualcuno (un altro processore o il DMA controller) può scrivere in memoria oltre il processore c'è bisogno di un meccanismo che dice al processore che la linea di cache può non essere coerente.

Questo è fatto associando un **bit di validità** ad ogni linea di cache che può essere resettato dal cache controller.

A volte il processore chiede al cache controller di scrivere immediatamente con una operazione di write-back una linea in memoria. Questo può succedere ad esempio quando si adotta il write-back e un blocco di dati in cache deve essere trasferito a una periferica attraverso il DMA. Questa operazione è supportata da un set di istruzioni (*cache flush*).

Se si adotta una politica set-associativa, una strategia è richiesta per selezionare la linea nel set da sostituire quando occorre un miss. Le soluzioni possibili sono LRU, Pseudo LRU (rispetto a LRU bastano 3 bit) e Random.

Il primo processore con cache on-board è il 80486; la grandezza delle cache è cresciuta nei processori successivi (8Kb nell'80486; 16Kb+16Kb nel Pentium III e 128K in Athlon).

Molto spesso una seconda cache è messa sulla scheda madre. Originariamente la cache on-line era chiamata L1 e la cache esterna chiamata L2; nei microprocessori più recenti anche la L2 cache è integrata nella CPU. Quando abbiamo due cache disponibili un miss in L1 causa un accesso alla L2, e un miss su L2 causa un accesso in memoria.

Normalmente la cache L1 lavora alla velocità della CPU (un ciclo di clock per l'accesso alla cache dati), mentre L2 è più lenta (ad esempio due cicli di clock per l'accesso).

Il problema della consistenza della cache si ha quando in un sistema si ha più di una cache; può succedere quando abbiamo cache L1 e L2 oppure quando è presente più di un processore con cache.

Per prevenire il problema della coerenza della cache viene adottato il protocollo MESI che è stato per primo implementato dal Pentium.

MESI

Si basa sull'associare ad ogni linea di cache uno stato che può assumere **4** diversi stati:

- **Modified (M)**: il dato nella linea di cache è disponibile in una sola cache dell'intero sistema; il suo valore è diverso da quello che c'è in memoria. La linea può essere letta/scritta senza aver bisogno di accessi ad altre cache o alla memoria.
- **Exclusive (E)**: il dato è presente in solo una cache dell'intero sistema. Il dato non è stato modificato fin ad ora, ed il valore è uguale a quello che c'è in memoria. Se occorre un'operazione di scrittura lo stato della linea di cache cambia in *Modified*.
- **Shared (S)**: il dato nella linea di cache può essere contenuto in più cache del sistema. Ogni scrittura del dato (indipendentemente se si usa il write-through o il write-back) deve essere fatta immediatamente in memoria, quindi le altre cache devono invalidare la corrispondente linea.
- **Invalid (I)**: la linea non è logicamente disponibile in cache. La causa può essere che la linea è vuota o che contiene un dato non valido. Ogni accesso ad una linea invalida causa un miss. Una scrittura deve essere fatta in write-through, e la write-allocate non è supportata.

Il protocollo MESI fu sviluppato con il write-back e senza la write-allocate; si può estendere al caso di write-through ma in questo caso non esistono gli stati M e E.

L'operazione di **Snooping** è fatta da ogni cache controller, che controlla il bus principale e può comunicare con gli altri cache controller. Un cache controller può chiedere ad altri cache controller lo stato di una data linea (*inquiring*). A conseguenza di questa interrogazione si possono fare molte azioni e lo stato della linea può essere cambiato.

Il protocollo specifica che azioni devono fare i cache controller quando un'operazione è fatta sulle variabili locali, un'operazione è fatta attraverso il bus o una variabile condivisa localmente e quando si riceve un'interrogazione attraverso il bus.

Nel caso in cui il sistema ha cache L1 e L2, il protocollo MESI deve tenere in considerazione che le variabili possono essere memorizzate in L1 o in L2. La consistenza tra le due cache, in principio era fatta assumendo che tutti i dati in L1 sono anche presenti in L2 (**Cache Inclusion**).

L'estensione del MESI permette di mantenere la consistenza in due modi:

- Forzare L1 ad usare write-through verso L2, e adottare il protocollo MESI sulla cache L2
- Permettere a L1 di adottare write-back, in questo caso MESI è più complesso

Bus System

Il bus system è l'elemento chiave in un sistema da quando esso influenza fortemente:

- L'architettura del sistema
- La comunicazione fra i componenti del sistema
- La comunicazione con gli elementi esterni

Architettura PC/XT

Il primo PC IBM fu il PC/XT che era equipaggiato con 8088 con data bus da 8 bit; nella versione 8086 il data bus era da 16 bit. La massima frequenza era 10MHz. Molti componenti sono ancora usati nei PC di oggi.

Abbiamo Local Bus, System Bus, X Bus e Memory Bus.

Architettura AT

E' stata introdotta con CPU 80286. Il range di frequenza va da 6 a 8 MHz; anche la scheda madre è equipaggiata con CPU da 25MHz.

In XT abbiamo un **Bus Slot** da 8bit con 62 contatti, mentre in AT ci sono 100 contatti con larghezza di 16 bit; le vecchie schede possono essere inserite nei nuovi slot e la logica bus riconosce automaticamente il tipo di scheda.

ISA (Industrial Standard Architecture) è il risultato della standardizzazione mirata a definire compatibilità per i sistemi AT. (è al 99% compatibile con l'originario bus AT). La massima frequenza operativa è 8.33 MHz.

L'introduzione di 386 e 486 a 32bit ha richiesto un'estensione del bus ISA. Furono proposte due soluzioni: la prima, **Microchannel**, era completamente una nuova architettura non compatibile con le schede precedenti; **EISA**, invece, fu l'evoluzione di ISA, più complesso, dovuto al bisogno di mantenere compatibilità con HW esistente.

EISA è un bus sincrono e ha una frequenza massima di 8.33MHz per accedere a tutte le unità esterne. La massima ampiezza di banda è 33Mbyte/s. La CPU può accedere alla memoria principale alla massima frequenza di clock attraverso il local bus. C'è un buffer che permette il trasferimento dati tra il local bus e il bus EISA. La CPU ospita la scheda EISA per controllare il bus; questo è cruciale per implementare sistemi multiprocessore.

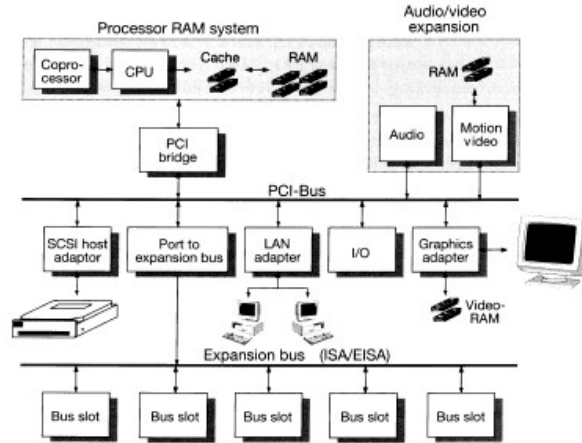
Al contrario di ISA, il bus EISA permette ad un microprocessore esterno o a una scheda EISA il controllo del bus. Si usa un modello di arbitraggio su tre livelli:

- DMA/refresh (livello più alto)
- CPU/Master
- Other masters (livello più basso)

Ad ogni livello il bus è commutato in ordine rotazionale

La principale limitazione di EISA era la frequenza operativa che iniziava a diventare critica per la gestione certi dispositivi che richiedevano banda elevata. Per questa ragione nel 1993 Intel introduce il bus **PCI** (Peripheral Component Interconnect). Le principali caratteristiche del bus PCI sono:

- Disaccoppiamento del processore ed estensione del bus attraverso un bridge
- Ampiezza 32bit con possibilità di estensione a 64
- 133Mbyte/s massimo rate di trasferimento
- Frequenza operativa maggiore di 33MHz
- Temporaneo multiplexing di bus dati e indirizzo
- Specificazione di processori indipendenti



Sul bus PCI si basa l'architettura ma sia le periferiche più lente che le più veloci hanno un loro bus; il LAN adapter è un adattatore con il bus EISA dove vengono messe le periferiche più lente.

L'unità PCI può essere integrato sulla scheda madre o implementato da un adattatore. L'Expansion Bus può essere un ISA, un EISA o anche Microchannel e può essere visto dal PCI come un adattatore. Fino a 10 unità PCI può essere supportato da un singolo bus.

Per risparmiare sul numero di linee il bus PCI multiplexa le linee di dato e di indirizzo; questo significa che un singolo trasferimento richiede da 2 a 3 cicli di clock: trasferimento di indirizzo, scrittura e lettura di dati.

Quando si lavora a 32bit a 33MHz significa che il massimo rate di trasferimento è 66Mbyte/s per le operazioni di scrittura e 44Mbyte/s per quelle di lettura.

Il **Burst Mode** è simile al modo sequenziale dell'ARM; riduce i colpi di clock per un trasferimento in caso che l'indirizzo sia sequenziale rispetto al precedente. Il massimo rate in questo mode cresce fino a 133Mbyte/s.

Il PCI bridge indipendentemente unisce l'inizio di un trasferimento di lettura e un'operazione di scrittura per accessi burst se gli indirizzi sono sequenziali; per questo i buffer di lettura e di scrittura sono inclusi nel PCI bridge.

Il PCI bus ha 3 spazi di indirizzamento: Memory, I/O e Configuration (include la configurazione dei registri di ogni unità).

Il PCI bridge disaccoppia completamente il bus primario dal secondario; per questo due unità PCI possono comunicare una con tutte le altre quando la CPU accede alla propria RAM.

L'unità PCI che richiede un ciclo è chiamata **initiator** (master) mentre l'unità indirizzata è chiamata **target** (slave).

Segnali

- FRAME: attivato dall'initiator per iniziare il trasferimento dati. Viene disattivato alla fine o quando viene interrotto il trasferimento
- STOP: attivato dal target per fermare il trasferimento
- IRDY: attivato dall'initiator per informare il target che l'indirizzo o il dato (in una fase di scrittura) è disponibile
- TRDY: attivato dal target per informare che il target è pronto per iniziare il trasferimento, o che il dato in un'operazione di lettura è disponibile

Ogni ciclo di bus è definito dal valore dei segnali da C/BE3 a C/BE0.

Il target per ogni ciclo è deciso in parallelo con il normale ciclo di bus (*hidden arbitration*); l'arbitraggio (che avviene ad ogni ciclo di bus) non richiede tempo.

Ogni unità PCI ha i segnali REQ e GNT che permette di interagire con la logica di arbitraggio; lo standard PCI non definisce quale strategia usare per l'arbitraggio.

Chipsets

Le schede madri nei primi PC (fino ai sistemi 286) erano composti da molti componenti. Grazie alla crescita della tecnologia dei semiconduttori era stato possibile integrare tutti questi componenti in pochi chips. La serie di chip creati per migliorare il disegno di schede madri con i processori è chiamato *chipset*.

I chipset non sono compatibili uno con l'altro (in termini di funzioni supportate, architettura e segnali) ma le schede madri risultanti devono poter eseguire software sviluppati per schede precedenti. Questo è possibile perché per ogni chipset viene sviluppato un proprio livello basso di sistema operativo (BIOS).

Normalmente il chipset è disegnato per un specifico processore; comunque molti processori non Intel possono essere integrati con chipset Intel, da quando questi sostituiscono esattamente i processori Intel.

Funzioni

Normalmente i chipset includono tutta la logica per la gestione della cache esterna; definiscono la quantità di memoria cachable che dipende dalla dimensione della RAM.

I chipset si differenziano in base alla quantità e al tipo di memoria supportata.

E' compito del chipset:

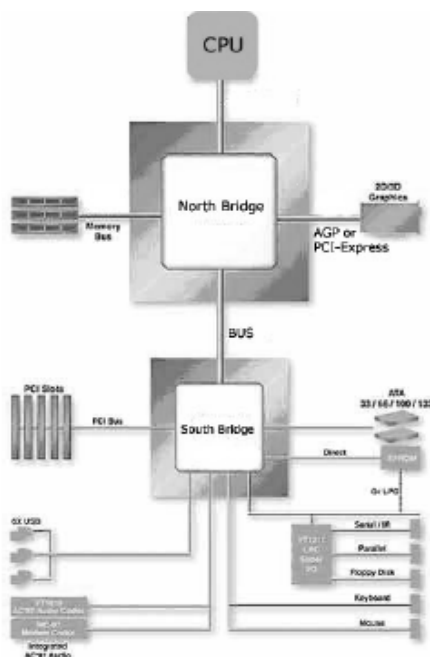
- generare i corretti segnali di controllo per accedere alla memoria
- la possibilità di bufferizzare i dati per o provenienti dal processore
- scoprire la quantità e il tipo di memoria esistente e generare gli appropriati segnali

Il chipset normalmente include il DMA controller; bridge per la connessione con altri bus; una coppia di interrupt controller e le interfacce per l'AGP e USB. I chipset più moderni includono una serie di funzioni per ridurre il consumo di potenza durante la fase inattiva del PC; la gestione della potenza avviene disattivando diverse parti del computer (video, hard disk) quando diventa idle.

Saturn Chipset

Costruito dall'Intel dal 486; si compone di tre chip:

- 82424TX: Cache e DRAM controller (**CDC**)
- 82423TX: Data Path Unit (**DPU**)
- 82378IB: System I/O (**SIO**)



Triton Chpset

E' stato il secondo chipset realizzato (1995) da Intel per i sistemi Pentium, dopo l'insuccesso dei chipset Mercuri. Anche lui composto da tre chip:

- S82437FX: Triton System Controller (**TSC**)
- S82438FX: Triton Data Paths (**TDP**)
- S82371FB: PCI ISA IDE Xcelerator (**PIIX**)

Gli attuali chipset di Intel si basano su due dispositivi: il North Bridge ed il South Bridge.

North Bridge

E' chiamato anche MCH (Memory Control Hub). E' connesso direttamente alla CPU e di base ha le funzioni di controllore della memoria, se disponibili in controllore del bus AGP e del PCI express, e l'interfaccia per il trasferimento dati con il south bridge.

In questo sistema la CPU non accede direttamente alla RAM o alla scheda video, è il north bridge che accede a questi dispositivi. Da quando il controller della memoria è sul north bridge, è questo chip che limita il tipo e la quantità massima della memoria nel sistema.

South Bridge

E' chiamato anche ICH (I/O Controller Hub). E' connesso con il north bridge ed è fondamentale incaricato al controllo dei dispositivi di I/O e i dispositivi a bordo (porte di hard disk, porte USB, bus PCI, CMOS memory). Il south bridge è connesso ad altri due chip sulla scheda madre: la ROM immagazzina parte del BIOS e il SuperI/O chip che è incaricato al controllo dei dispositivi tipo porta seriale, parallela e floppy disk.

Il south bridge non è così critico nelle prestazioni come il north bridge; può avere molta influenza sulle performance dell'hard disk driver, è il south bridge che setta il numero e la velocità delle porte USB e il numero e il tipo delle porte dell'hard disk driver che ha la scheda madre.

Quando l'idea dei bridge è iniziata ad essere usata, la comunicazione tra north e south bridge avvenivano via PCI. L'ampiezza di banda per il bus PCI è divisa tra tutti i dispositivi PCI nel sistema e i dispositivi agganciati al south bridge (specialmente l'hard disk driver). A quel tempo questo non era un problema dato che il massimo rate di trasferimento degli hard disk driver era 8MB/s e 16MB/s; al momento della crescita delle schede video e l'aumento delle performance degli hard disk, questo causa un collo di bottiglia (ad ex l'hard disk ATA/133 potrebbe usare da solo l'intera banda avendo un teorico rate di trasferimento come quella del bus PCI).

Per le schede video la soluzione è quella di creare un nuovo bus connesso direttamente al north bridge chiamato **AGP** (Accelerate Graphics Port).

La soluzione è quella di usare un bus ad alta velocità tra north e south bridge e connettere i dispositivi PCI al south bridge. La velocità di questo bus dedicato dipende dal modello del chipset.

AMD decide di implementare le funzionalità del north bridge direttamente nel processore, perciò il chipset è composto solo dal south bridge.

Memory

I sistemi PC normalmente includono DRAM che implementano la memoria principale e la SRAM che implementano la cache esterna. La gestione di queste memorie richiede una adeguata implementazione del controllore della memoria.

DRAM

Internamente è organizzata normalmente come una matrice; di conseguenza l'indirizzo è diviso in indirizzo di riga e di colonna. I segnali che controllano il trasferimento sugli indirizzi di riga e colonna sono chiamati rispettivamente RAS e CAS.

Prima di accedere alle celle, il circuito di **precharge** carica tutte le linee di bit fino al valore $V_{cc}/2$. Questo richiede un certo tempo, chiamato **RAS precharging time**. Una volta che questa operazione è completata, la propria word line è attivata, e gli amplificatori sentono e amplificano alcune differenze nelle bit line.

Un ciclo di **refresh** è normalmente richiesto ogni da 1 a 16 ms, dipende dal tipo di RAM; sono usate normalmente 3 strategie per il refresh:

- **RAS-only refresh**: si basa sul mandare alla DRAM tutti i possibili indirizzi di riga senza inviare alcun indirizzo di colonna. In questo caso la DRAM legge e amplifica i valori di ogni riga, senza produrre alcun dato. Nei PC questo è portato dal canale 0 del DMA chip, che è periodicamente attivato dal contatore 1 del timer chip e fatto da un falso ciclo di lettura.

- **CAS-before-RAS refresh**: per le normali operazioni si attiva prima il segnale RAS (prendendo l'indirizzo di riga) e successivamente si attiva il segnale CAS. Per attivare questa logica, il segnale di CAS deve essere attivato per un certo tempo prima del RAS. Si fa quindi il refresh di tutte le righe e si incrementa un contatore interno.
- **Hidden Refresh**: il ciclo di refresh è attivato dopo un ciclo di lettura tenendo il segnale CAS basso per un tempo lungo, e alternando su e giù solo il segnale di RAS. Questa tecnica permette di risparmiare tempo, il tempo richiesto per un ciclo di refresh è solitamente minore di un ciclo di lettura.

Il **Page Mode** può essere sfruttato quando due o più accessi sono fatti, con riferimento a locazioni aventi lo stesso indirizzo di riga. In questo caso l'indirizzo di riga, e il corrispondente segnale RAS, è dato una sola volta, e solo l'indirizzo di colonna è fornito (assieme al segnale di CAS). Operazioni di lettura e scrittura possono essere unite finché si accede alla stessa pagina in page mode. Spesso si specifica il massimo numero di operazioni consecutive che si possono fare in page mode. La memoria che supporta il page mode è normalmente chiamata **Fast Page Mode (FPD) DRAM**.

I vantaggi che si ottengono usando questo modo sono che riduce del 50% i tempi di accesso e del 70% i tempi di ciclo.

In **EDO mode** l'indirizzo di colonna può essere stabilito con alta frequenza.

Static-Column Mode è la versione modificata del page mode, dove il segnale CAS non ha bisogno di essere alternato su e giù durante la ricerca dell'indirizzo di colonna. Il controllo logico della DRAM trova l'indirizzo di colonna e accede al nuovo dato. Ha come conseguenza un abbassamento dei tempi di accesso e dei tempi di ciclo. Questo metodo è usato solo nei modelli IBM PS72.

Nel **Serial Mode** dopo il primo accesso (quando sono dati gli indirizzi di riga e di colonna) il segnale di CAS è portato su e giù rapidamente, a ciascun tempo un bit è shiftato fuori. Questo può essere utile per accessi fatti da CRT.

L'**Interliving** può velocizzare la sequenza degli accessi in memoria; si basa nel dividere la memoria in diversi banchi e di accedervi alternativamente. In questo caso gli accessi ai diversi banchi può essere parzialmente sovrapposto.

E' compito del memory controller capire:

- Se una sequenza di accessi in memoria si riferisce alla stessa pagina, e così entrare in page mode
- Se la sequenza alterna tra indirizzi pari e dispari, e così si può adottare l'interleaving.

Se nessuno delle due tecniche può essere usata è compito del memory controller chiedere per lo stato d'attesa.

SRAM

E' connesso al clock di sistema e lavora senza stati di attesa; internamente spesso sfrutta l'interleaving. A parte essere sincro, lavora come le altre memorie, supporta il page mode.

Le DDR SDRAM hanno una ampiezza di banda per il trasferimento dati il doppio per ciclo; il tempo di accesso delle SRAM è da 8 a 20 ns contro i 60-80 delle DRAM.

Pentium Pro

Il pentium pro (P6) è stato introdotto da Intel nel 1995. Intel integra la CPU (5.5 milioni di transistor) e la cache L2 (2 milioni di transistor) in un singolo package (*cartridge*) composto da due parti che sono connesse da un bus dedicato che va alla massima velocità della CPU (200 MHz).

Significanti problemi di consumo sono stati riscontrati dai nuovi dispositivi. Sono stati introdotti importanti vantaggi da questa architettura.

La **Pipeline** è composta da 12 stadi. I singoli stage sono più semplici, permettendo un aumento della frequenza di clock. Dovuto alla sua lunghezza è richiesto un meccanismo di branch prediction molto efficace

e un efficiente meccanismo di scheduling delle istruzioni.(Intel sostiene che si possono raggiungere 3 istruzioni per ciclo di clock).

L1 cache: sia la cache dati che la cache istruzioni hanno alcune caratteristiche comuni (2 vie set-associative; grandezza 8 Kb). La cache dati ha 2 porte di accesso che riduce i conflitti. Supporta pienamente il protocollo MESI.

Decoding Unit

Recupera una linea di cache (32 bytes) per colpo di clock dalla cache L1. identifica i limiti dell'istruzione e compie branch prediction. E' composto da 3 unità che lavorano in parallelo (2 per le istruzioni semplici e una per quelle complesse). Le istruzioni semplici sono mappate in una singola m-ops; mentre le istruzioni complesse sono mappate in più m-ops (fino a 4). L'unità di decodifica può produrre 6 m-ops per colpo di clock, che sono scritte nell'*instruction pool* (che può memorizzare al massimo 20-30 m-ops; in modo continuo controlla quale m-ops può essere inviata all'unità funzionale, in qualsiasi ordine).

La **Branch Prediction** si basa sul BTB che memorizza informazioni riguardanti 512 salti.

Converte le referenze dei registri x86 nelle referenze interne dei 40 registri del Pentium Pro, questo per fare register renaming.

Execution Unit

Ha due unità per le istruzioni FP; due unità per le istruzioni intere e un'unità di interfaccia alla memoria. Le 5 unità possono operare in parallelo o indipendentemente.

Retire Unit

Controlla continuamente nell'*instruction pool* per completare le istruzioni; questo avviene strettamente in ordine.

Appena diviene possibile, compie la risoluzione delle dipendenze di dato, la verifica delle branch prediction e aggiornamento del registro reale.

L2 cache: è connesso alla CPU attraverso un bus a 64 bit che lavora alla CPU velocità, e memorizza dati e codice. E' set associativa con una grandezza totale di 256 Kb o 512 Kb.

Per ridurre il numero degli stalli della pipeline, sono supportate due istruzioni MOV (CMOV conditional move, FCMOV conditional FPU move).

In Pentium Pro gli indirizzi possono essere estesi da 32 a 36 bit; la grandezza dipende dal valore del bit PAE nel registro CR4.

Il Pentium Pro è particolarmente adatto per il **multiprocessore**. Fino a 4 Pentium Pro può essere connesso e lavorare insieme senza aver bisogno di logica aggiuntiva. Per questo scopo è concepito un appropriato arbitraggio del bus.

Pentium II: introdotto nel 1997, supporta la tecnologia MMX. Include 7.7 milioni di transistor e è ancora inviato in un cartdridge includendo L2 cache. L1 cache ha una grandezza di 16 Kb

Pentium III: introdotto nel 1999 e include 9.5 milioni di transistor. Supporta 72 nuove istruzioni SSE che estendono la tecnologia MMX per le operazioni FP

Pentium IV: introdotto nel 2000 con 42 milioni di transistor; ha un iniziale frequenza di 1.5 GHz.