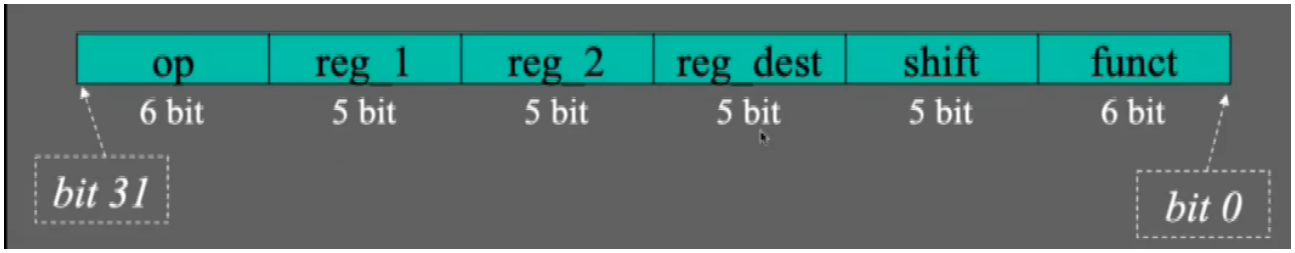
ARCH II

Struttura RISC

Formato delle istruzioni

Più è lungo il datapath (sequenza di porte logiche “attraversate” dal segnale elettrico durante l’istruzione) più lungo sarà il tempo di svolgimento dell’istruzione.
Analizziamo una versione semplificata della MISP, una macchina RISC.
È composta da 32 registri interi di dimensione 32 bit (ogni registro può contenere 32 bit, quindi è x32).



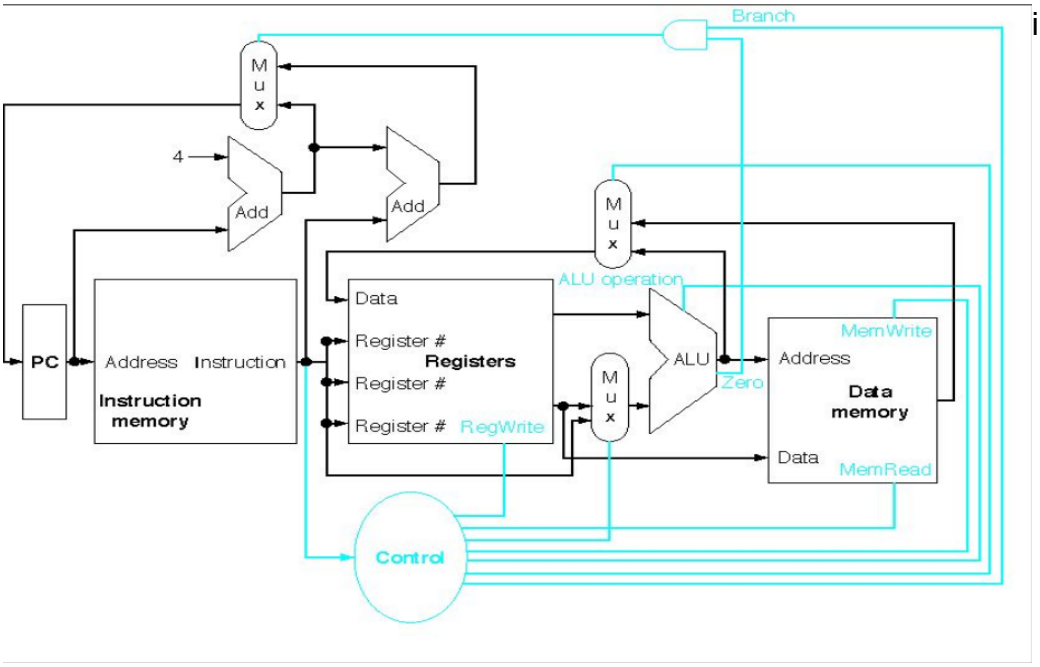
Le istruzioni sono a di lunghezza fissa little endian
op: tipo di operazioni (add, load, store,)
reg: due registri di input ed uno di destinazione
shift: operazioni di shift…..eh beh.
Funct: specifica una delle possibili varianti dell’operazione op richiesta (ad esempio se op è una istruzione “R”, cioè le operazioni che usano 2 registri e restituiscono un risultato nel registro dest, funct specifica l’operazione (somma, sottrazione, etc...))

infatti se l’operazione non fosse di tipo-R la sua struttura cambierebbe (ma non la sua lunghezza).
Ad esempio le istruzioni di tipo-I (immediate) sono così composte:

Questo formato è uguale anche per le istruzioni di tipo load e store. In tal caso “immediate value” non è un valore, bensì l’offsett da cui partire per calcolare la posizione del dato in memoria. (utile nel caso di array)
Naturalmente abbiamo istruzioni di salto condizionate e incondizionate, ma non le riporto. La cosa importante da sapere è che essendo istruzioni di salto non necessitano di due registri, bensì di un indirizzo ad una istruzione.

Ciclo di Esecuzione

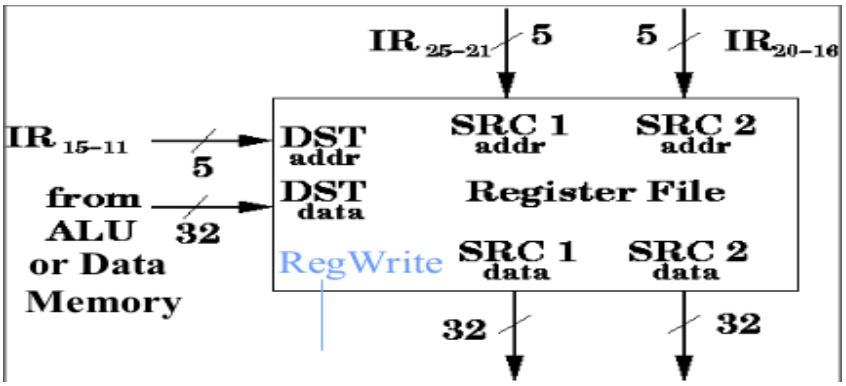
Si segue sempre il ciclo load-fetch-execute, ma durante la fase di load non carica solo il codice, bensì vengono contemporaneamente letti uno o due registri, quelli specificati dall’istruzione.
Ciò è ovviamente più rapido ed è reso possibile dal fatto che le istruzioni sono tutte a lunghezza fissa e sopratutto I “puntatori” ai registri si trovano sempre nella stessa posizione dell’istruzione . Male che vada se l’istruzione non ha bisogno dei valori dei 2 registri (ad esempio è un salto incondizionato) li butto.
In nero il datapath, in blu la Control Unit. I due Adder in alto si occupano di gestire il valore del Program Counter ad esempio (notare che il primo adder ha un ingresso fisso a 4, cioè ogni volta aumenta il valore del PC di 4 e questi andrà a leggere l’istruzione successiva (salta di 32 bit, 2^4)). Invece il multiplexer in basso a destra permette di decidere cosa fornire alla ALU i bit relativi ad un indirizzo nel codice, nel caso l’istruzione fosse di Jump l’indirizzo di un registro in caso contrario.



L’esecuzione di ciascuna istruzione in questo datapath può avvenire in un unico ciclo di clock (anche se nella realtà non è così, andrebbe contro l’idea di base dei RISC).
Comunque il discorso è sempre lo stesso di ArchI, i vari elementi di stato (flip-flop, memorie, circuiti combinatori) leggono dati in input solamente sul fronte di salita del clock.

Register file

Un grosso componente “non ancora incontrato” è il banco dei registri, composto da 32 registri da 32 bit. Si possono effettuare 2 operazioni sul Register File: lettura e scrittura



Il segnale di controllo “RegWrite” identifica l’istruzione (viene messo a 1 se si vuole scriver),
DST addr identifica il registro su cui si vuole lavorare, DST data è l’input del banco (usato solo in fase di scrittura).

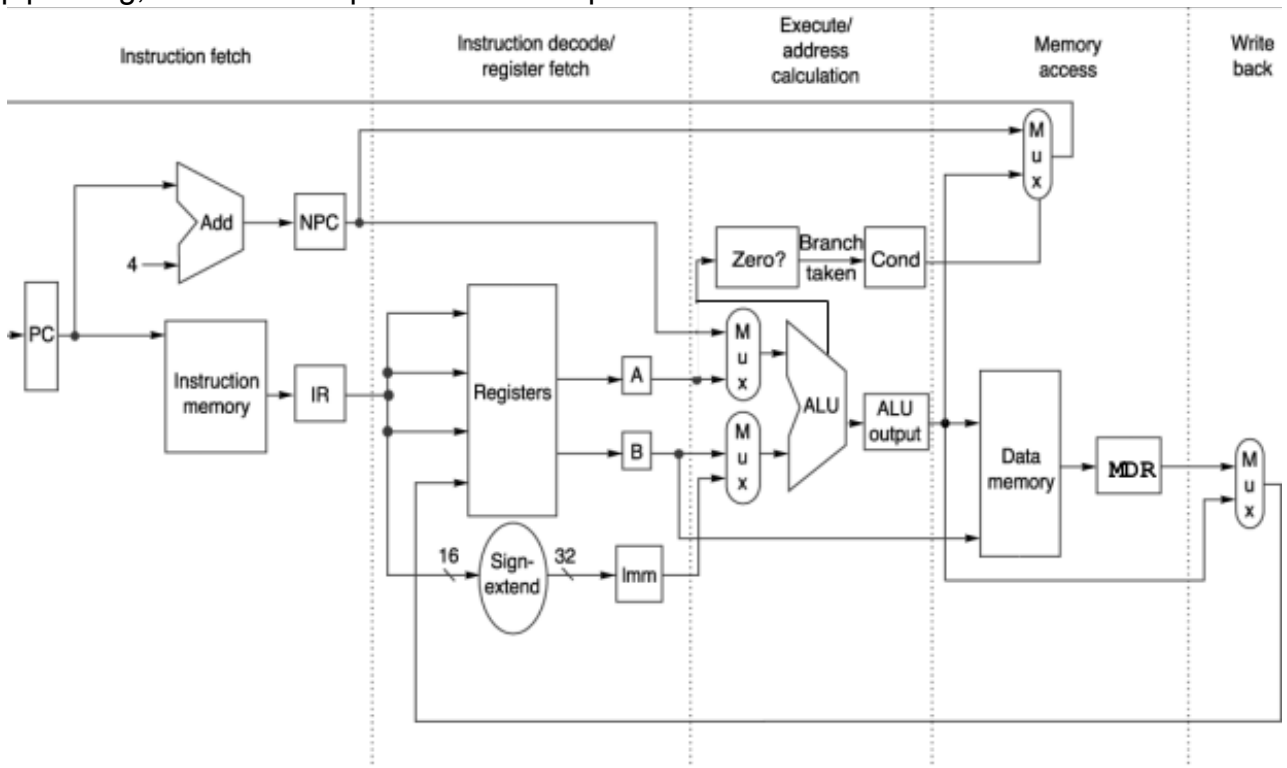
Control Unit

Si occupa di decidere il tipo di operazioni richieste alla ALU, di sincronizzare i segnali e di fornire l’accesso al banco dei registri ed alla Memoria dati (istruzioni store e load)
Analizziamo il primo: AluOP (il tipo di operazione richiesto dall’ istruzione in esecuzione, campo op e campo function).
La ALU è pilotata da un segnale a 4 bit (16 operazioni), la control unit può quindi essere vista come una semplice tabella di verità che dato un input di 6 bit (campo OP) fornisce in output tutti i segnali di controllo del datapath. Avendo la tabella di verità è quindi banale costruire un circuito che la implementi.
Vediamo, come esempio, come la Control Unit pilota l’esecuzione di una istruzione di tipo-R
1. il contenuto del PC viene usato per indirizzare la Instruction Memory e produrre in output l’istruzione da eseguire.
2. il campo op dell’istruzione viene mandato in input alla Control Unit, mentre i campi reg_1 e reg_2 vengono usati per indirizzare il register file. (questo succederebbe anche nel caso reg_1 e reg_2 fossero inutili, ad esempio nel caso di un salto non condizionale).
3. La CU riceve in input i segnali che indicano una operazione di tipo-R e fornisce in output su ALUOp i 4 bit che indicano l’istruzione di tipo-R. Alla Alu viene anche passato il campo “funct” dell’istruzione
4. nel frattempo il register file ha letto i due registri e li ha prodotti in output, il segnale ALUSrc viene impostato a zero per indicare alla ALU che sta per ricevere in input il valore di un registro (il discorso registro/indirizzo di codice fatto nella pag prima)
5. La ALU esegue l’operazione e fornisce un output. La CU imposta il segnale RegWrite del register file in modo da salvare l’output sul registro di destizinazione.

6. avviene il clock, il register file legge quello che si trova in input, il PC viene incrementato, etc… Si riparte dal passo 1.

Macchina Multiciclo

Come detto prima avere un ciclo di clock così lungo è contro la filosofia RISC (il clock deve avere una durata sufficiente a permettere l'esecuzione dell'istruzione con la durata maggiore, quindi tutte le istruzioni più "rapide" sprecano un sacco di tempo), quindi si cerca di avere un ciclo di clock il più corto possibile, scomponendo l'esecuzione di una singola istruzione in diversi *passi*.
A questo punto il clock può essere impostato alla durata di esecuzione del *passo* più lungo, il tempo sprecato è decisamente minore dato che i passi sono di durata infinitamente più breve delle istruzioni. (oltretutto la macchina multiciclo permette l'introduzione della pipeline).
Ci sono però dei vincoli: Il più importante è che ogni passo dovrà contenere al più una operazione
su ciascuna unità funzionale, non posso eseguire due operazioni differenti sulla ALU nello stesso passo.
In figura il datapath di una macchina MISP multiciclo. La differenza più notevole è la divisione in sezioni (che poi sono i passi), e la presenza di registri "tra le sezioni". Questi registri "nascosti" fanno da interfaccia fra l'output di una fase e l'input della fase successiva (IR,NPC, A, B, Imm, etc)
Si chiamano "nascosti" (in futuro li chiameremo registri di pipeline) perché non sono indirizzabili, non visibili all' Istruction Set.
Notiamo anche la presenza di 2 memorie: Data Memory e Istruction Memory. Anche lì pipelining, una memoria per i dati ed una per il codice.



Le fasi della macchina multiciclo

Ad ogni fase corrisponde un ciclo di clock.

Instruction Fetch (IF)

Viene prelevata l'istruzione con indirizzo PC dall'instruction memory e salvata nel registro IR
Contemporaneamente un adder incrementa di 4 il valore del PC e lo salva nel registro NPC (New Program Counter). Perchè un adder separato e non la ALU? Pipelining.

Instruction Decode (ID)

Si leggono i due registri, nel caso l'istruzione richiesta necessiti dei due registri scritti nell'istruzione. (al solito, se non servono si buttano)
Allo stesso modo sul registro Imm viene trasformato in un intero a 32 bit il valore del campo "immediate value" di una eventuale istruzione di tipo-I. Se l'istruzione non sarà poi di tipo-I il contenuto di Imm non sarà usato.
Viene anche calcolato l'eventuale valore del PC nel caso l'istruzione sia un JMP condizionata o meno.
Tutte queste operazioni vengono svolte in parallelo

Execute (EX)

La CU riceve l'op-code dall'instruction register, di conseguenza predispone la ALU a svolgere l'operazione richiesta.
La ALU ha già tutti gli input che gli servono dalla fase di ID, quindi fa il suo calcolo e scrive il risultato nel registro ALU Output, nel caso l'istruzione sia di salto si attiva il segnale Zero (qualora il salto sia condizionato prima si confronta il valore dei registri) che viene scritto nel registro Cond in modo che al passo successivo il valore di ALU output venga scritto nel Program Counter.

Memory access (MEM/WriteReg):

Se si sta eseguendo una load o una store il registro ALU output conterrà un indirizzo. Nel caso di una store il valore che viene salvato e quello contenuto nel registro B.
Attenzione, questo passo può essere "saltato", nel caso di in cui l'istruzione non richieda l'uso della memoria (ad esempio una istruzione di tipo-I o di tipo-R). In tal caso i dati da ALU output vengono direttamente scritti nel registro indicato, "risparmiando" un ciclo di clock che sarebbe sostanzialmente inutile.

Write Back (WB/REG)

Usato solo nelle istruzioni che accedono alla memoria (tipo la load). Il valore salvato nella MDR viene scritto nella Data Memory all'indirizzo specificato dall'istruzione in esecuzione (che nel frattempo è stato decodificato e poi passato alla ALU).
In questa fase la CU si occupa di portare ad 1 il segnale RegWrite.

La Control Unit della macchina multiciclo

L'unità di controllo non deve solo più specificare quali segnali devono essere asseriti (ALUOp, RegWrite, etc..) bensì deve anche indicare quale sarà la fase successiva da compiere a seconda dell'istruzione; come abbiamo visto diverse istruzioni fanno uso di diverse fasi
Ogni fase sarà quindi descritta da una tabella di verità avente in input il tipo di istruzione da eseguire (il campo op come nella macchina monociclo) e la fase in cui si trova la sequenza di esecuzione dell'istruzione.
In output fornirà sì l'insieme dei segnali da asserire ma anche la fase successiva che deve venire svolta.
Sta roba ricorda tantissimo un DFA (Macchina di Moore) ed in quanto tale può essere realizzata combinando la CU monociclo con un registro che memorizzi lo stato corrente della macchina (in particolare un registro a 4 bit, disegnando il diagramma del DFA escono fuori 10 stati possibili, quindi servono almeno 4 bit).
Essendo un Automa a stati finiti possiamo rappresentarne gli stati tramite un insieme di microistruzioni, quindi possiamo descrivere l'esecuzione di un'istruzione tramite una sequenza di microistruzioni, un microprogramma.

RISC vs CISC

Macchine CISC

Proprio questa visione è stato ciò che ha portato alla tendenza ad avere tante istruzioni più complesse, la cosiddetta architettura CISC (Complex Instruction Set Computer).
Era infatti più semplice creare istruzioni complesse ragionando su un microprogramma.
Non fu ovviamente l'unico motivo, il motivo principale fu la memoria. la ram era molto lenta e la CPU cache non esisteva. aveva senso fare pochi accessi ed avere la CPU più occupata possibile durante la lettura/scrittura in RAM.
I compilatori erano ancora primitivi, i programmatori scrivevano spesso in linguaggio macchina quindi istruzioni complesse semplificavano il loro lavoro. In ultimo era conveniente descrivere la funzione di controllo tramite microistruzioni, infatti se si voleva cambiare ISA ad un processore bastava andare a sostituire la ROM, non tutta la CPU.
Una conseguenza di queste istruzioni molto complesse fu l' impossibilità di avere istruzioni di lunghezza fissa, a parte che le istruzioni erano super lunghe, la loro lunghezza era molto variabile e forzarle tutte alla massima lunghezza era un vero spreco.
Quindi:

- avendo lunghezza variabile era impossibile capire se il dato letto dalla memoria fosse l'inizio, il mezzo o la fine di una istruzione.
- avendo lunghezza variabile era impossibile "precaricare" gli operandi dato che non erano specificati semore negli stessi bit, l'istruzione andava decodificata prima di leggere gli operandi
- essendo istruzioni complesse richiedevano datapath complessi e di conseguenza un clock molto lungo

Inoltre vi era un enorme problema: non vi era differenza fra la lettura da un registro e la lettura da RAM. Una istruzione poteva richiedere un operando da registro ed uno da memoria causando un collo di bottiglia enorme. Perchè facevano sta cagata? così il programmatore non doveva preoccuparsi di copiare nei registri i dati dalla memoria, ci pensava il microcodice

Macchine RISC

Ad inizio anni ‘80 Patterson e Sequin smettono di descrivere la CU tramite microprogramma e coniano il termine RISC (Restricted Istruction Set Computer).
Contemporaneamente Hennessy genera l’architettura MIPS (Microprocessor *without* Interlocked Pipeline Stages)
Entrambi gli studi si basavano sul fatto che alcuni ricercatori (tipo Tanenbaum) avevano dimostrato che la maggior parte dei programmi spendeva la maggior parte del tempo di esecuzione eseguendo istruzioni semplici.
Quindi lo sviluppo delle nuove architetture si basò su 3 punti cardine:

1. poche istruzioni semplici per avere un clock molto rapido (dato che il datapath è più semplice)
2. l’accesso in memoria va limitato il più possibile, in particolare solo tramite load e store.
3. le istruzioni devono principalmente usare argomenti contenuti in registri, il salvataggio in memoria è una istruzione a parte (store)

Questi principi imponevano uno stile di progettazione diverso, nel bene e nel male.
In primis bisognava rinunciare ad un sofisticato livello di microcodice, poi bisognava avere delle istruzioni di lunghezza fissa e struttura simile, si poteva accedere alla ram solo tramite LOAD e STORE, i processori dovevano avere molti registri.
Il principale vantaggio di questo stile di progettazione era la possibilità di sfruttare il pipelining.

Storia

La prima svolta la diede nel 1986 la SUN Microsystem incominciando a produrre un computer con architettura RISC.
Da li anche altre altre aziende iniziarono a produrre processori RISC, fino a quando a fine anni 80 venne confrontata la macchina CISC più potente sul mercato con la macchina RISC più potente. La macchina RISC era 3 volte più potente. A quel punto tutte le aziende si misero a produrre processori RISC.
Attualmente non esistono quasi più processori CISC, persino nell’embedded, a parte i processori Intel che si porta dietro una forte eredità CISC.
Questo perché Intel si è impegnato nel fornire una backward compatibility per le CPU della sua famiglia. I programmi scritti per l’8086 possono essere fatti girare sulle CPU attuali.
Ma la verità è che neanche loro sono più CISC, all’interno dei processori è presente un “cuore” RISC ed un traduttore da Istruzioni a lunghezza variabile ad istruzione a lunghezza fissa.
D'altra parte la differenza tra RISC e CISC è diventata molto flebile, da una parte con processori CISC che permetto pipeline come quello Intel, d’altra parte con processori RISC che utilizzano ISA sempre più complessi ad esempio le GPU che hanno istruzioni molto complicate per la gestione di contenuti multimediali.

Pipelining

L’idea di base è questa: possiamo sovrapporre (almeno parzialmente) l’esecuzione di istruzioni consecutive. Ad ogni ciclo di clock, mentre una porzione del datapath è impegnata nell’esecuzione di una istruzione, un’altra porzione del datapath può essere usata per eseguire un’altra istruzione. Possiamo lanciare una nuova istruzione ogni clock.

Attenzione però a non compiere contemporaneamente due operazioni differenti che usano le stesse risorse del datapath (non posso far fare allo stesso componente più cose).
C'è una eccezione a questa regola. Presupponiamo che due istruzioni stiano facendo una ID e l'altra WB, entrambe usano il Register File, ma una in lettura ed una in scrittura quindi basta imporre che avvengano nella prima e nella seconda parte del ciclo di clock (che vuol dire? scrittura in fase di salita del clock e lettura in fase di discesa? boh, la cosa da ricordarsi è che si può fare lettura/scrittura su un registro nello stesso clock)
Per evitare questo tipo di conflitti conviene quindi dividere la memoria dei dati e la memoria delle istruzioni, inoltre gestire il PC separatamente, senza usare la ALU (a parte istruzioni di JMP/BNE ovviamente).
Come anticipato i registri che collegano le varie fasi cambiano nome, ora li chiamiamo registri di pipeline e li identifichiamo usando il nome dei due stadi che collegano, quindi: IF/ID, ID/EX, EX/MEM e MEM/WB. (WB/IF non esiste, l'output è eventualmente scritto sul Register file)

Prestazioni

Il pipelining migliora decisamente le prestazioni, per quanto l'hardware più complicato renda l'esecuzione di una singola istruzione più lenta.
un po' di calcoli per dimostrarlo:
supponiamo una CPU senza pipeline con i seguenti dati:
Operazioni con la ALU e branch richiedono 4 cicli di clock. Operazioni di accesso alla memoria richiedono 5 cicli di clock.
op. ALU = 40%; branch = 20%; accessi alla memoria = 40%.
clock = 1 ns.
tempo medio di esecuzione di una istruzione = (0,4 + 0,2) x 4ns + 0,4 x 5 ns = 4,4 ns.
Supponiamo ora che la CPU sia dotata di pipeline, e ipotiziamo che l'overhead introdotto ad ogni ciclo dall'hardware necessario ad implementare la pipeline sia di 0,2 ns (è un valore ragionevole).Abbiamo:
speedup = avg. exec time senza pipeline / avg. exec time con pipeline = 4,4 ns / 1,2 ns = 3,7.
Ossia un incremento (teorico) delle prestazioni di 3,7 volte

Problemi del pipelining

Ci sono tre classi di problemi che possono limitare la produttività della pipeline:
1. Problemi strutturali 2. Problemi sui dati 3. Problemi di controllo
Quando si verifica uno di questi problemi è necessario fermare (to stall) la pipeline ed aspettare che una istruzione precedente finisca.

Problemi Strutturali

Si verificano quando alcune risorse hardware all'interno del datapath non sono duplicate, quindi non possono essere usate contemporaneamente (ad esempio usando la stessa memoria per dati ed istruzioni).
Si possono risolvere duplicando le unità funzionali, in particolare quelle combinatorie (le ALU), al costo di avere circuiti più complessi quindi più costosi e che consumano di più. Bisogna trovare un compromesso

Problemi sui dati

Si verificano quando una istruzione vuole usare un dato che sta venendo calcolato da una istruzione precedente, quindi non è ancora utilizzabile.

se ad esempio avessimo:
DADD R1, R2, R3 //R1 = R2 + R3
DSUB R4, R1, R5
OR R8, R2, R9

DSUB avrebbe bisogno del valore R1 nella sua fase di ID, ma l'istruzione 1 DADD è solo nella fase di EX, verrà scritto il risultato in R1 solo tra 2 clock.
Notiamo però che la 3a istruzione, la OR potrebbe già essere lanciata, non gli serve attendere che DADD si concluda.
Prima soluzione, il forwarding
il risultato della ADD è in realtà già disponibile durante la fase di EX, si trova sull'output della ALU, le prossime 2 fasi servono solo a scriverlo nei registri.
L'idea del forwarding è "dirottare" il dato, collegando il registro EX/MEM (o anche il registro MEM/WB) all'input della ALU e permettendo alla CU di usarlo nel calcolo successivo (quindi bisogna creare una CU che in primis si occupi di dirottare, in secondo luogo deve anche capire quando può dirottare e quando non può).
Il forwarding non risolve però tutti i casi, risolve solo i casi in cui l'output è generato dalla ALU. se avessimo bisogno di un valore ottenuto tramite una load non potremmo fare nulla, dovremmo attendere il 5° clock per ottenerlo.

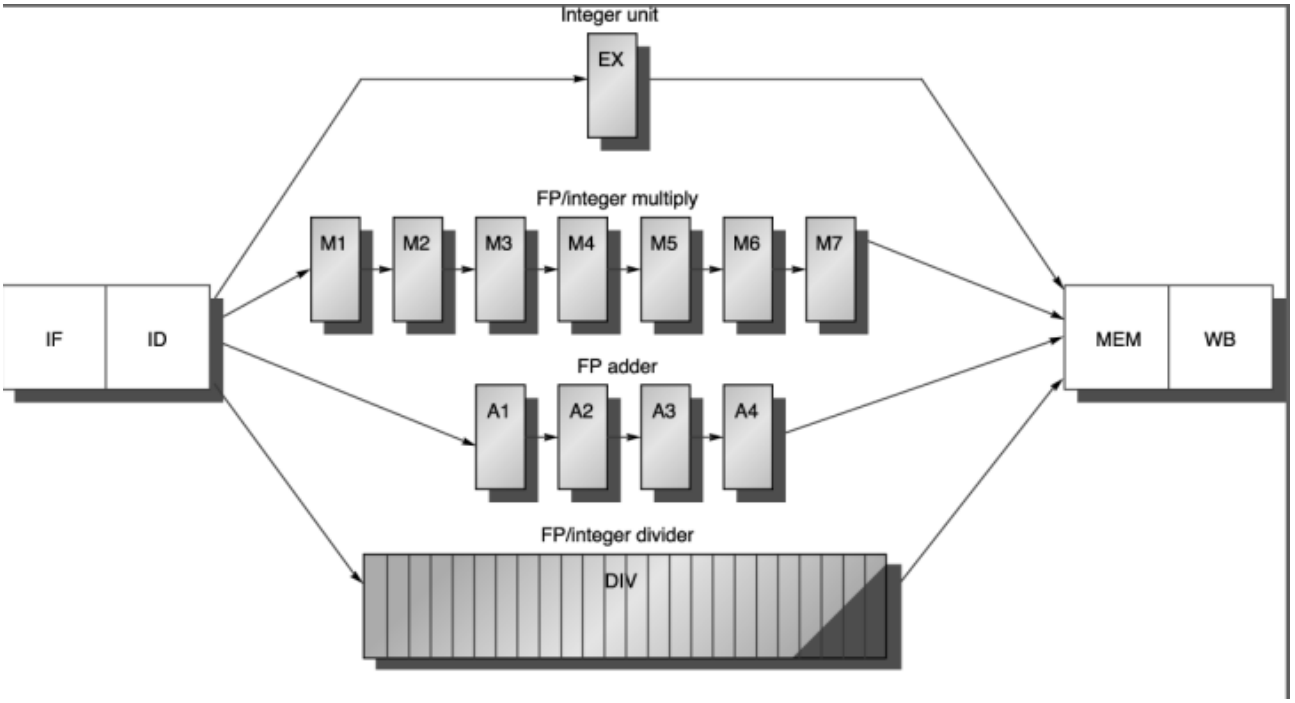
Problemi di controllo

Sto effettuando una istruzione di BRANCH……finchè non valuto la condizione non so quale sarà il prossimo valore di PC e non so quale istruzione devo lanciare.
Una soluzione abbastanza semplice è ignorare il problema, eseguiamo l'istruzione successiva se poi il salto va eseguito annulliamo l'esecuzione dell'istruzione inutile.
Abbiamo comunque uno spreco di cicli di clock, sopratutto perchè nei casi reali si eseguono spesso branch in base ad operandi anche complessi, che magari richiedono parecchi cicli di clock per essere computati.
Quindi si potrebbe implementare una cosiddetta *predizione statica (static branch prediction)*:
Facciamo due assunzioni a priori (derivano da osservazioni statistiche):
Eseguiamo tutti i salti all'indietro (quasi sempre i salti indietro sono cicli), non eseguiamo i salti in avanti (è difficile che si skippino parti di codice, di solito sono i casi di errore).
Se poi la predizione è errata abbiamo sprecato cicli di clock, però statisticamente ci azzecca.
Un' ulteriore tecnica è il *delayed branch*, Consiste nello spostare dopo un branch una istruzione I che è comunque necessario eseguire, indipendentemente dall'esito del branch. Questo richiede supporto da parte del compilatore, che deve inserire dopo l'istruzione di salto delle istruzioni che devono essere eseguite in ogni caso.
ATTENZIONE: Interrupts, trap, eccezioni di varia natura che interrompono il normale flusso di istruzioni sono molto difficili da gestire in un sistema con pipeline. Ci sono modi di gestirle, ma non li vediamo.

Pipelines più sofisticate

si può usare più di una pipeline (di solito 2), prelevando 2 istruzioni dalla memoria del codice.
Ovviamente bisogna gestire bene i conflitti, evitando di avere conflitti e dipendenze. Anche qui serve un po' di supporto da parte del compilatore.
Perchè fermarsi a 2 pipelines, le CPU moderne ne usano addirittura 3 o 4.

In particolare si può fare una roba molto furba, avere diverse pipelines che dopo la fase di ID smistano le istruzioni in unità funzionali specifiche (Integer unit, floating point unit), questo permette si di parallelizzare ma anche di differenziare il ciclo di clock. la durata di una operazione di somma è molto minore di quella di una divisione.
Perciò si fissa il clock alla durata della fase più breve e le fasi con operazioni più complesse (divisione, moltiplicazione, etc..) passano attraverso delle unità funzionali "spacchettate" in modo da essere compatibili con il clock più corto. (il clock deve comunque permettere di svolgere le fasi di IF,IF,MEM,WB in un solo ciclo).
Le architetture che sfruttano più unità funzionali nella fase EX, vengono comunemente definite superscalari, cioè architetture che avviano all'esecuzione in parallelo più istruzioni per ciclo di clock.



ILP Dinamico

Ci sono due modi generici per aumentare la potenza computazionale di una CPU:

1. aumentare la frequenza di clock
2. aumentare il numero di istruzioni eseguite in parallelo (multiple issue)

La soluzione 1 è applicabile fino ad un certo punto, perchè costringe il progettista a riprogettare l'architettura in modo che il lavoro sia svolto in un numero maggiore di fasi. Ma avere più fasi implica avere più istruzioni in esecuzione contemporaneamente aumentando sì la potenza di calcolo ma richiedendo anche una CU più complessa e producendo più calore/richiedendo più energia.

Multiple Issue

La soluzione 2 naturalmente richiede la presenza di istruzioni indipendenti e la presenza di un datapath più complesso:
Deve essere disponibile un numero sufficiente di unità funzionali per l'esecuzione di più istruzioni in parallelo, deve essere possibile prelevare più istruzioni dalla IR e dati dalla Data Memory contemporaneamente e deve essere possibile leggere/scrivere su più registri in parallelo nello stesso ciclo di clock.

Il multiple issue permette ad una architettura pipelined di eseguire più di una istruzione per clock (CPI < 1, clock per instruction), ma questo solo idealmente, perchè usare multiple issue incrementa anche il numero di stall.

Implementazione

Per prima cosa bisogna confrontare le istruzioni tra di loro per stabilire quali sono eseguibili in parallelo.
Una volta trovate tot istruzioni avviabili in parallelo queste vengono “impacchettate” in un issue package e avviate all’esecuzione.
Ci sono due modi di identificare le istruzioni: issue statico ed issue dinamico.
Nel primo caso è il compilatore a cercare le istruzioni indipendenti, nel secondo caso è il processore che lo fa a runtime.
Nella realtà i processori usano un po’ un mix delle due
per implementarlo abbiamo quindi bisogno:

1. di più unità funzionali (più ALU, più adder, etc..)
2. deve essere possibile prelevare dalla Instruction Memory più istruzioni, e dalla Data Memory più operandi, per ciclo di clock
3. Deve essere possibile indirizzare in parallelo più registri della CPU, e deve essere possibile leggere e/o scrivere i registri usati da diverse istruzioni in esecuzione nello stesso ciclo di clock

Un processore che rispetta queste 3 caratteristiche può implementare il multiple issue e viene detto processore con **architettura superscalare**

Instruction Level Parallelism Dinamico

Le 3 tecniche usate sono: scheduling dinamico della pipeline, branch prediction e speculazione hardware.
Lo scheduling dinamico della pipeline permette di eseguire le istruzioni “out of order”.
Prendiamo ad esempio:
LD R4, 100(R2)
DADD R10, R4, R8
DSUB R12, R8, R1
L’istruzione DADD deve attendere il registro R4 ma DSUB non ha bisogno di aspettare, il processore se ne accorge e la esegue.
Attenzione che però ciò non deve modificare il risultato finale dell’esecuzione.
Il branch prediction dinamico serve a ridurre i problemi di controllo, si mantiene un “log” della storia passata di un salto in modo da predire il suo comportamento.
per ogni salto condizionato si memorizza l’esito della sua esecuzione, ossia se il salto è stato eseguito o no. Se lo stesso branch viene di nuovo eseguito, la CPU usa la storia passata di quel branch per predire se il salto verrà eseguito o no.
Se il salto non ha una “storia” non apporta guadagno.
La speculazione hardware è una estensione della branch prediction, dopo aver effettuato una prediction il processore può svolgere tutte quelle operazioni che svolgerebbe se la predizione è corretta.
Se la predizione si rivela tale bene, ho già il risultato. Altrimenti il processore che implementa la speculazione hardware deve annullare l’effetto delle istruzioni eseguite speculativamente.
La differenza fra branch prediction e speculazione hardware è sottile: nella branch prediction le istruzioni vengono avviate e messe in pipeline ma mai concluse.
Così se la predizione è sbagliata basta “azzerare” i valori salvati nelle pipeline e si può riprendere l’esecuzione.

Invece la Speculazione hardware presuppone che le istruzioni vengano concluse ed i risultati salvati. se la predizione è sbagliata è più complicato fare un “revert”.

Dipendenze sui dati e sui nomi

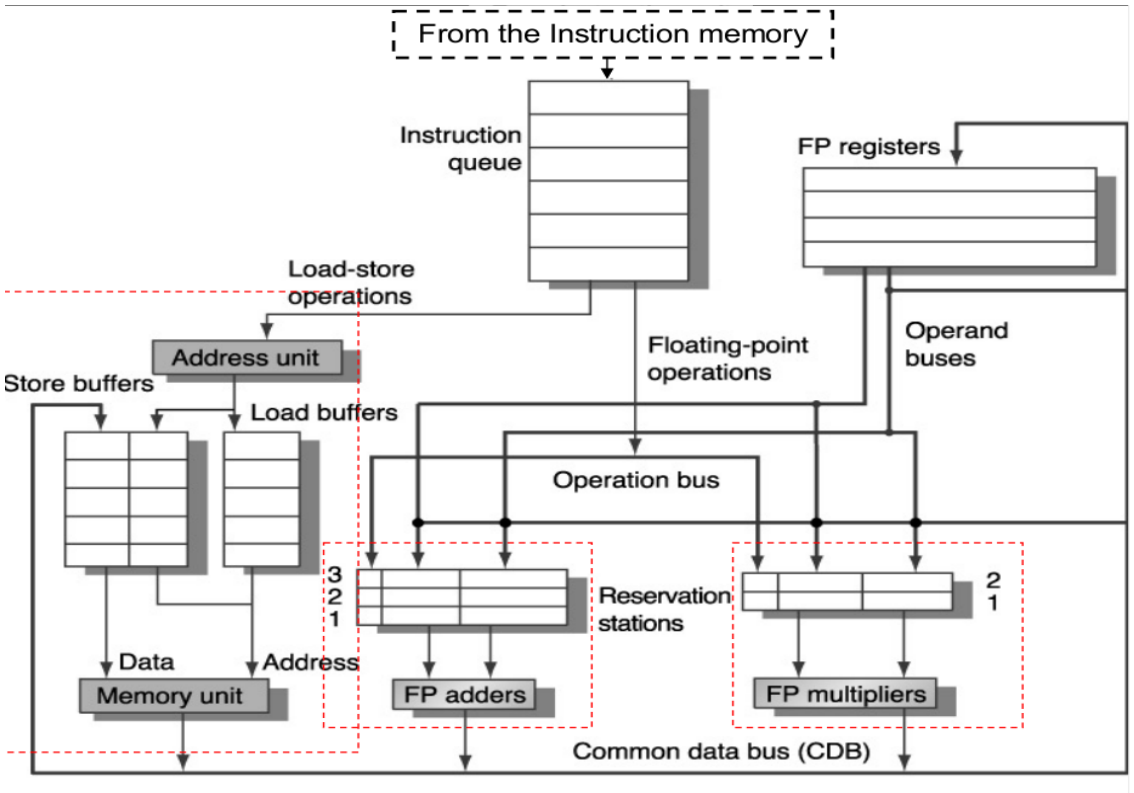
SI verifica dipendenza sui dati (true dependency/ data dependency) quando una istruzione per poter essere eseguita necessita di dati prodotti da istruzioni mandate in esecuzione in precedenza.
per sfruttare al meglio il parallelismo è fondamentale determinare le dipendenze fra istruzioni, in modo da avviare contemporaneamente le istruzioni indipendenti.
La dipendenza sui nomi è più sottile: se due istruzioni usano lo stesso registro (il programmatore è scemo) c’è il rischio che il riuso del registro vada a stallare la pipeline.
Esempio:
MUL R7, R3, R6
PRINT R7
LOAD R7, #100(R3)
Per quanto la print e la load usano lo stesso registro (R7) non vi è dipendenza sui dati, se la load usasse R6 ad esempio non avremmo problemi.
Vi sono poi 2 categorie di dipendenze:

1. antidipendenza: una istruzione legge da un registro su cui una istruzione successiva scriverà. devo fare attenzione che quest’ultima non sia eseguita prima che la prima sia riuscita a leggere.
2. dipendenza in output: due istruzioni scrivono sullo stesso registro, l’ordine di scrittura deve essere mantenuto in modo che nel registro ci sià il risultato della seconda.

Le dipendenze sui nomi si possono risolvere, basta usare un altro registro (se ce ne sono di disponibili), ciò può essere fatto staticamente dal compilatore o dinamicamente dalla CPU.
Una soluzione alternativa è di usare qualche registro aggiuntivo ma “nascosto” (nel senso di non visibile al programmatore) usato solo per gestire situazioni come questa.

Schema di Tomasulo

Tramite lo scheduling dinamico abbiamo anticipato che si possono mitigare queste dipendenze (riordinare le istruzioni e rinominare i registri); la tecnica viene chiamata “approccio di Tomasulo” dal creatore.
La tecnica si basa su due idee: tenere traccia di quando gli operandi delle istruzioni sono disponibili (indipendentemente dall’ordine con cui vengono eseguite) e la tecnica dei “registri nascosti” vista prima.
Lo schema di Tomasulo prevede l’implementazione di nuove zone di memoria, le **stazioni di prenotazione** (**reservation station**), dove le istruzioni “sostano” in attesa che gli operandi siano disponibili.
Una reservation station è composta da una o più entry, ogni entry contiene un’istruzione e un altro valore riferito all’operando.
Se l’operando è disponibile contiene il valore stesso, se l’operando non è ancora stato calcolato contiene l’identificativo della entry che contiene l’istruzione che calcolerà questo operando.
Tutte le entry sono collegate ad un Bus, il CDB (Common Data Bus) che permette di trasferire in parallelo il risultato prodotto in output da una unità funzionale a tutte le stazioni di prenotazione che lo stanno aspettando e al register file.



Notare che ci sono due buffer un po' particolari, lo store buffer ed il load buffer, infatti l'indirizzo in cui operare viene calcolato ed inserito nel buffer, nel caso della store nel buffer è presente anche il valore da scrivere.
Lo schema di Tomasullo si compone quindi di 3 operazioni:

1. ISSUE: prelievo di una istruzione da IR, decodifica ed inserimento di questa nella stazione di prenotazione relativa all'unità funzionale corretta, se l'operando è disponibile l'istruzione prosegue, altrimenti viene scritto l'identificativo dell'istruzione che lo produrrà
2. EXECUTE: l'istruzione viene prelevata da una reservation station ed inoltrata alla corretta unità funzionale
3. WRITE RESULT il suo risultato viene distribuito tramite il CDB a tutte le reservation stations.

Notare che senza componenti aggiuntivi sono stati anche implementati i registri "nascosti", sono le stesse reservation stations.
Schema di funzionamento molto esplicativo nel pdf 03-ILP-dinamico-parte1 pag 48

Branch Prediction dinamico

Abbiamo già visto cos'è la branch prediction, ma come si implementa? il modo più semplice è usare un branch prediction buffer (predittore) di dimensione 2^n in cui ogni bit è indirizzabile dagli ultimi n bit meno significativi dell'indirizzo di una istruzione di branch. (così non devo starmi a salvare la coppia istruzione-valore).
ogni bit indica se la volta precedente in cui è stata eseguito quel branch il salto è stato preso o no.
Sta cosa fa schifo, ad esempio consideriamo un ciclo iterato 10 volte. dopo di che il programma esce dal ciclo, per rientrarvi in una fase successiva. (tipo 2 cicli annidati)
Il processore valuta erroneamente il branch
Per evitare questo problema si usa di solito uno schema di predizione a due bit (local 2-bits predictor): una predizione deve essere sbagliata due volte consecutive prima di venire modificata.

Ci sono poi tecniche più sofisticate:

- Schema a predittori correlati: i predittori a 2 bit non si riferiscono ad un salto, bensì ad una coppia di salti consecutivi
- Schema a torneo: si usa sia un predittore a due bit che uno ad uno, ogni volta si usa quello che ha avuto più successo nelle predizioni passate

Un'altra cosa che si può fare per ottimizzare le predizioni è calcolarsi il valore a cui salta il programma (se salta) e salvarlo in un buffer per le prossime volte (branch target buffer)

Speculazione hardware

Quando capita? quando per analizzare un branch ci vogliono molti cicli di clock (ad esempio bisogna andare a recuperare un valore in memoria). In tal caso le istruzioni successive hanno abbastanza tempo per essere "concluse".
Come detto in precedenza la speculazione hardware salva i risultati come se la predizione fosse corretta, ma se non è così? deve cancellare tutto.
Si modifica lo schema di Tomasulo, inserendo nel suo schema una **unità di commit** (ROB, ReOrder Buffer).
se non si è certi della predizione si mettono temporaneamente i risultati delle istruzioni eseguite speculativamente all'interno del ROB.
Quando la CPU "sa" che una istruzione nel ROB doveva effettivamente essere eseguita, ne esegue il commit, la rimuove dal ROB e ne inoltra il risultato sui bus CDB e nel registro di destinazione (o una cella della memoria dati nel caso di una STORE).
ATTENZIONE: il commit deve avvenire nell'ordine in cui le istruzioni sono entrate in CPU, infatti il ROB è gestito come una coda. Infatti, pensate, che succede se una istruzione speculativa genera una eccezione, e poi si scopre che quella istruzione non doveva essere neanche eseguita?
Ogni entry del ROB è composta di 4 campi:

1. il tipo di istruzione (branch, store o ALU/Load
2. il campo di destinazione (che sia registro o memoria)
3. il valore
4. un campo ready che indica se l'istruzione è già stata eseguita o no.

Usando la speculazione hw dobbiamo anche aggiungere un passo allo schema di Tomasulo, il passo COMMIT.
Oltretutto in fase di ISSUE l'istruzione prelevata viene inserita in fondo alla ROB, in fase di WRITE RESULT il risultato non viene inviato in memoria bensì nel ROB.
Il passo di COMMIT preleva una istruzione dalla ROB (ricordati che il commit delle istruzioni viene fatto in-order, qualsiasi sia poi la loro esecuzione) e ne studia il tipo.
Se l'istruzione non è un branch il contenuto del campo valore viene scritto nella destinazione e l'istruzione rimossa dalla ROB.
Se l'istruzione è un branch con predizione corretta si prosegue con l'operazione di commit.
Se l'istruzione è un branch con predizione sbagliata tutte le istruzione presenti nella coda vengono cancellate. (tutte le istruzioni eseguite da li in poi non andavano svolte).
(FORSE DOBBIAMO ANCHE SVUOTARE LE RESERVATION STATION, ma non ne sono sicuro)
Se l'istruzione è un branch di cui non sappiamo nulla ciccia, si verifica uno stall finchè non viene valutata. ma usare un ROB molto grande permette di incontrare raramente questa situazione

ILP + Multiple issue = multiple issue dinamico.

Un processore superscalare si comporta così :

1. preleva dall IR “n” istruzioni, n è il n° di istruzioni che la memoria IR (la cache di solito) riesce a fornire in parallelo.
2. Le istruzioni vengono messe in una Instruction Queue (IQ), dove vengono analizzate dall CU per individuare eventuali dipendenze su dati e su nomi. (per il multiple issue quindi dipendenza strutturale, occhio, non per ILP. ILP se li risolve da solo i conflitti, tramite Tomasulo)
3. Le sole istruzioni indipendenti vengono eliminate dalla IQ ed inviate alle stazioni di prenotazione, le istruzioni dipendenti rimangono nella IQ
4. al ciclo successivo vengono prelevate altre N istruzioni e rianalizzate tutte le istruzione nella IQ, alcune dipendenze potrebbero essersi risolte.

Ma questo vuol dire che in casi particolarmente sfortunati la IQ si saturi, nessuna istruzione nella IQ può essere avviata e si verifica uno “stall”.
Oltretutto notiamo che se usiamo il multiple issue dobbiamo avere un ROB che permette di effettuare commit di tante istruzioni quante possono uscire dalla IQ, altrimenti sticazzi del multiple issue, tanto il ROB fa commit di una istruzione alla volta e diventa un bottleneck
Esempio di Multiple issue su schema di Tomasulo a pag 108 nel pdf
03-ILP-dinamico-parte1 (lezione del 19-10 minuto 14:00 circa)

Vantaggi del’ ILP dinamico

Perchè si usa ILP dinamico? è molto più analizzare il codice staticamente in fase di compilazione. ci sono 4 ragioni:
Non possiamo staticamente sapere se un dato cercato è in cache di primo, secondo, terzo livello o addirittura in RAM, ILP dinamico lo sa invece e permette di fare altro mentre il dato viene caricato.
Non si possono prevedere con accuratezza i branch in modo statico
ILP statico richiede una architettura apposita (preciso datapath, cicli di clock fissi, registri fissi) ed il compilatore deve essere scritto per quella specifica architettura dato che ne deve sfruttare le caratteristiche intrinseche.
In ogni caso è una tecnica economica e facile da implementare, tanto che spesso i processori con ILP statico usando anche ILP dinamico
Ma quanto è conveniente sfruttare il parallelismo tra istruzioni?
Teoricamente (branch prediction perfetto, infiniti registri, multiple issue illimitato) tantissimo, ipoteticamente tra le 50 e le 150 istruzioni per clock.
Realisticamente tra le 2 e le 4 istruzioni per clock (ignorando comunque lo stalling).
I processori moderni hanno già raggiunto il massimo sfruttamento possibile dell’ILP, bisogna sfruttare avanzamenti tecnologici (miglior caching, uso di GPU per operazioni non grafiche) o direttamente il parallelismo esplicito (multithread)

ILP Statico

Utile nel caso embedded (ecco perchè compilare per un Arduino richiede un sacco di tempo!), ove per consumare poca energia e mantenere il costo basso i processori non sono dotati delle molte porte logiche necessarie ad implementare ILP dinamico.
L’idea di fondo è semplicemente riordinare le istruzioni macchina in fase di compilazione in modo da sfruttare il parallelismo.
In particolare se una istruzione B dipende da una istruzione A, allora B deve essere separata da A di una “distanza” in cicli di clock pari almeno al numero di cicli di clock necessari ad A per produrre il risultato che dovrà essere usato da B.
Nei cicli di clock fra A e B si avviano altre istruzioni sempre seguendo questo criterio.

Come anticipato, per sapere quanto distanziare A e B il compilatore deve sapere quanti cicli di clock sono necessari ad eseguire una istruzione, quindi il compilatore genera codice specifico per una unica architettura.

Pipeline scheduling

E' esattamente quello spiegato prima: modificando l'ordine con cui verrano eseguite le istruzioni si possono minimizzare gli stall. prendiamo ad esempio il codice a sinistra e riformuliamolo a destra, supponendo che un branch richieda 2 clock, una store/load 2 clock ed una ADD 3 cicli.

```
LOOP: LD F0, 0 (R1)
stall
FADD F4, F0, F2
stall
stall
SD F4, 0 (R1)
DADD R1, R1, #-8
stall
BNE R1, R2, LOOP
stall
```

```
LOOP: LD F0, 0 (R1)
DADD R1, R1, #-8
FADD F4, F0, F2
stall
BNE R1, R2, LOOP
SD F4, 8 (R1)
```

Ripeto, questa forma di scheduling è possibile solo se il compilatore conosce la durata di ogni operazione.

Loop unrolling statico

Studiando lo stesso ciclo visto sopra possiamo introdurre anche il "loop unrolling", infatti il codice visto sopra possiede 5 istruzioni ma 2 di queste (DADD e BNE) sono necessarie alla sola gestione delle iterazioni del ciclo.
L'idea del loop unrolling statico è quella di raggruppare in un unico macrociclo più iterazioni dello stesso ciclo, sostanzialmente cancellando 2 istruzioni ogni "unroll"
Se ad esempio eseguiamo loop unrolling del codice visto prima 4 volte otteniamo un unico macrociclo di 14 istruzioni (4*3 = 12 istruzioni aritmetiche +2 istruzioni per la gestione del ciclo).
Ci sono due downside: 1 il compilatore ha occupato più registri, infatti quello che prima veniva gestito da un registro solo (F4) ora deve essere diffuso su 4 registri, uno per ogni iterazione srotolata. possiamo riusare sempre lo stesso registro? si ma rendiamo le operazioni dipendenti tra di loro.
Infatti il secondo downside è che preso così il codice è comunque meno ottimizzato di quello generato dal pipeline scheduling, infatti svolgendo i calcoli richiede 28 cicli per eseguire 4 iterazioni, 7 clock per iterazione contro i 6 della procedura di prima.
Il vantaggio del loop unrolling sta proprio nel fatto che le nuove istruzioni generate sono indipendenti tra loro ed è quindi possibile sfruttare il multiple issue. (è anche possibile usare Pipeline scheduling e parallelizzare ancora il codice)

Difetti del loop unrolling

Il numero di registri è limitato, usare cache di 2°,3° livello o addirittura la RAM è super inefficiente.
Alcuni cicli non possono essere unrolled, in pratica solo i for, in cicli do-while no.
Inoltre il codice generato è più lungo, ciò aumenta il rischio di cache miss nella IR, di fatto rallentando il programma

Multiple issue statico

A differenza della versione dinamica gli issue packet di istruzioni indipendenti non sono forniti dalla CU, bensì è il compilatore stesso a presentare il codice in pacchetti alla CPU.
All'interno di un pacchetto gli spazi "vuoti" generati da istruzioni non indipendenti devono essere sostituiti da no-op.
Questo approccio viene chiamato VLIW (Very Long Instruction Word).
Il VLIW oltre ad implementare il multiple issue statico usa tecniche di compilazione più sofisticate.

Istruzioni predicative (o condizionali)

Una tecnica così figa che viene usata pure da processori che usano ILP dinamico.
L'idea è quella di eliminare dal codice i salti non facilmente predicibili (tipo gli if) e di sostituirle con delle specifiche istruzioni macchina, istruzioni predicative o condizionali.
Una istruzione predicativa contiene una condizione che viene valutata durante l'esecuzione della stessa.
Se la condizione è vera il resto dell'istruzione viene eseguito normalmente, altrimenti l'istruzione si comporta come una no-op
L'esempio più semplice è la move condizionale: copia il contenuto di un registro in un'altro se la condizione è vera.

```
if (R1 == 0) {
    R2 = R3;
    R4 = R5;
} else {
    R6 = R7;
    R8 = R9;
}
```

(a)

```
    CMP R1,0
    BNE L1
    MOV R2,R3
    MOV R4.R5
    BR L2
L1: MOV R6,R7
    MOV R8,R9
L2:
```

(b)

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

(c)

Questo ovviamente c'è per diverse istruzione, in alcuni casi per tutte (full predication).
Per implementare queste istruzioni basta creare degli opportuni registri predicativi (formati da un bit) che contengono il risultato di test effettuati da istruzioni precedenti.
In partica stiamo trasferendo a livello ISA tutti quei piccoli if-then-else che sono molto facili da eseguire ma molto difficili da predirre.
Quasi sempre viene implementata solo la move condizionale (anche in processori con ILP dinamico) perchè le istruzioni predicative sono più complesse da eseguire ed in genere poco usate

Architettura IA-64. Processore Itanium

Nasce alla fine degli anni '90 in sostituzione della architettura x86 (fallendo).
Unica architettura Intel completamente RISC, basata sull'approcio VLIW (rinominato EPIC perchè ad intel sono dei cazzari).
Vi è un uso intensivo dell' ILP statico, il commpilatore organizza le istruzioni in blocchi chiamati "instruction groups"; la CPU deve eseguire gli instruction groups in ordine, mentre le istruzioni al loro interno sono indipendenti, quindi si posso eseguire out-of-order.
Questa architettura possiede una istruzione predicativa per ogni corrispondente istruzione (non solo la move) ed implementa una nuova tecnica, le load speculative.

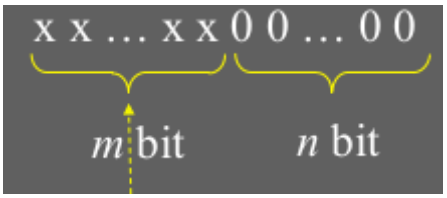
In parole povere si cerca di anticipare il più possibile una load, ma potrebbe capitare che vi sia una dipendenza sui dati ad esempio si potrebbe star cercando di effettuare la load da una locazione di memoria prima della store che va settare il valore.
Ma se è così tanto vale salvarsi da qualche parte il valore della store al posto che andare a scrivere in memoria per poi andarci a leggere subito dopo. Dove salvarlo? nella ALAT una table speciale dedicata a questo.
Quando si fa una store si salva anche il valore passato alla memoria alla ALAT, quando si fa una load si va a vedere se è presente quel dato nella ALAT, se si la entry viene azzerata, se no si esegue la normale LOAD

CACHING

La memoria principale è sempre molto più lenta della CPU, sia a causa della memoria in se che dal fatto che deve passare attraverso un bus.
Negli anni i processori sono diventati sempre più potenti, anche le memorie ma queste devono sottostare a diversi limiti fisici ed architettonici; quindi la differenza di prestazioni tra CPU e memorie è molto aumentata con il tempo facendo nascere la necessità di memorie intermedie.
in particolare una gerarchie di memorie sempre più veloci (quindi più voluminose, costose e meno capienti) fisicamente vicine alla CPU. le cache L1(di primo livello) è praticamente attaccata al processore. poi vengono L2 ed L3. (secondo e terzo livello)
Non esiste un solo tipo di memoria: ci sono i dischi magnetici, le memorie flash, etc, etc.
La più grande distinzione che possiamo individuare è DRAM vs SRAM.
SRAM (static RAM) usano flip-flop per memorizzare i bit e hanno un tempo di accesso di qualche nanosecondo, quindi le cache L1,L2,L3 sono generalmente costruite così. la DRAM (dynamic RAM) che invece usa un condensatore ed un transistor ha invece tempi di accesso nell'ordine di 10 nanosecondi. (il grosso del delay proviene dal bus però, devo creare gli indirizzi, inviarli al controller del bus e mettere in conto la distanza fisica).
ATTENZIONE: il caching non si riferisce solo alle cache L1, L2, L3. i registri fanno da cache alla cache, che fa da cache alla RAM, che fa da cache all'hardisk, etc….
La cache è molto vantaggiosa per due "ipotesi": Aree di memoria con indirizzi simili a quelli appena usati, saranno a loro volta usate nell'immediato futuro (array, istruzioni, variabili), aree di memoria accedute di recente verranno di nuovo accedute nell'immediato futuro (variabili di un ciclo, istruzioni)
Ma non è l'unica distinzione fra tipi di memoria, ricordiamo anche:
SDRAM (DRAM sincrone, con un clock), => DDR (, duoble data ram ,SDRAM che possono trasferire dati sia su clock ascendente che discendente) => DDR2, DDR3, DDR4 (il numero indica la versione, una versione più alta indica minor consumo e maggior clock)
GDRAM (memorie con un'alta leggibilità parallela adatte ai processori grafici)
FLASH Memory (Memorie molto piccole la cui riscrittura avviene a livello di interi blocchi di dati, anche dati non modificati vengono riscritti. Lente come gli Hard-disk in scrittura ma super veloci in lettura)

Funzionamento di una cache (da RAM a cache)

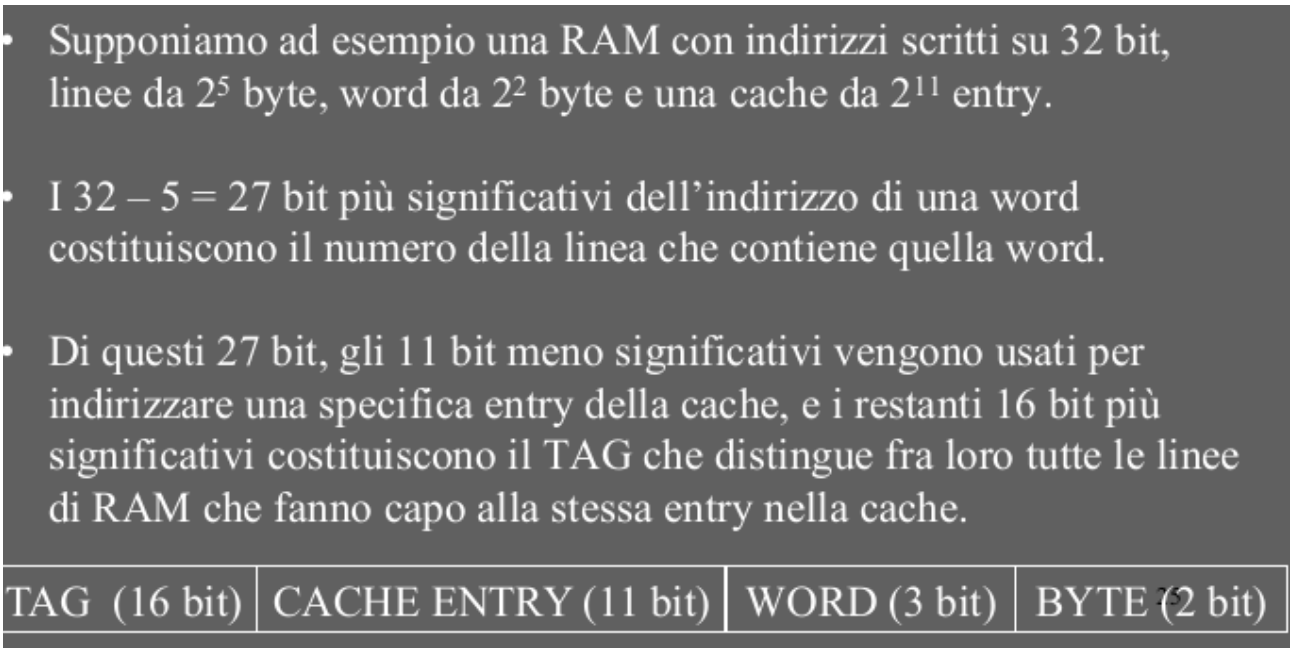
SIa RAM che cache sono divise in blocchi di dimensioni fissa (di solito da 4 a 256 byte) chiamati rispettivamente linee di cache e linee di RAM.
Ogni linea è identificata da un indirizzo, che indica il primo byte appartenente a quella linea. se le linee sono di dimensione 2^*n* byte ogni indirizzo ha la seguente forma:



gli *m* bit più significativi identificano il numero di una linea (quindi possiamo indirizzare fino a 2^*m* linee), gli *n* bit meno significativi identificano il dato preciso (ne parleremo più avanti, per non ci servono dato che in cache viene caricata sempre una linea intera).
Quando viene richiesto un dato (nel caso di dato non in cache, *cache miss*) la linea contenente il dato mancante viene prelevata dalla ram e salvata nella cache.
Dipendentemente dall'architettura il dato viene salvato in due modi diversi.

Cache direct-mapped

il tipo di cache più semplice. ogni entry memorizza una linea di RAM più due informazioni: Un bit di validità che dice se quella entry è veramente in uso o è un dato residuo ed un campo TAG che identifica univocamente la linea di RAM.
Ogni linea di RAM viene memorizzata in una entry ben precisa, tramite la più semplice operazione di hashing: n° linea ram % numero di entry nella cache.
A più linee RAM corrisponde la stessa entry (relazione RAM -> cache non iniettiva).
Proprio a questo serve il campo TAG, gli *x* bit più significativi del numero di ciascuna linea sono il TAG che permette di distinguere fra loro tutte le linee di RAM che fanno capo alla stessa entry della cache.
Quanti sono questi *x* bit? se la ram ha 2^*n* linee e la cache 2^*m* entry si leggono gli *n*-*m* bit più significativi.
m = 5, n = 3, l'indirizzo è composto da 5 bit xxxxx, se faccio indirizzo % numero di entry ottengo sicuramente un numero di 3 bit, i due bit più significativi vanno nel campo TAG.
Nel caso reale avviene la stessa cosa, ma stiamo lavorando con indirizzi più grandi.
Oltretutto il processore indirizza ai byte, ma la cache si salva le linee intere perciò gli indirizzi sono un po' più complessi. L' indirizzo può essere logicamente scomposto in TAG, CACHE ENTRY, WORD e BYTE
WORD indica l'indirizzo della word ed è un campo di *m* bit poichè una linea di RAM contiene 2^*m* word e BYTE è un campo di *n* bit, presupponendo che una word sia di 2^*n* byte.
così è confuso, quindi esempio numerico, a Gunetti piacciono questi calcoli



In questo caso quando un indirizzo viene generato vengono presi gli 11 bit di "CACHE ENTRY", se la entry corrispondente è valida ed il campo TAG della entry viene comparato con i 16 bit più significativi dell'indirizzo (nell'immagine identificati anche loro come TAG). Se tutto combacia abbiamo una cache hit, grazie al campo WORD dell'indirizzo possiamo estrarre il dato che ci interessa (si, ma BYTE a che cazzo serve? tanto persino nei registri della CPU si lavora con le word, mai con i singoli byte).
In caso uno dei requisiti non sia soddisfatto si ha una cache miss e si va a cercare in RAM il dato.
In caso di scrittura ci sono 2 modalità: write-back e write-through. in caso di modifica di dati in cache il write-through propaga la modifica alla RAM, in write-back no, vi è modifica del dato sono quando una linea viene sostituita da un'altra con differente TAG

Cache Set-Associative

Si chiamano cache set-associative ad n vie, con n di solito 2,4,6,8.
Una cache set-associativa ad n vie è suddivisa in più insiemi, ciascuno dei quali è formato da n entry e può quindi contenere n linee.
Quando caricata in cache l'insieme che contiene una linea è stabilito da: (numero linea in ram)%(n° di insiemi).
Ma una volta identificato l'insieme bisogna decidere in quale delle *n* entry dell'insieme bisogna salvare la linea di ram.(si tenta di scegliere una entry libera, se si deve sovrascrivere si sovrascrive la Least Recently Used)
Svantaggio: quando ci riferiamo ad una linea in cache non basta calcolare l'insieme, bisogna verificare se una delle entry di quell'insieme contiene la linea indirizzata.
Vantaggio: molti meno miss-cached.
L'estremo sarebbe creare una cache con un insieme solo, la cosiddetta cache completamente associativa (fully cached), ma è un assurdo perchè il sistema di controllo delle entry all'interno dell'insieme sarebbe complicatissimo da implementare.
Questo sistema infatti deve controllare tutte le linee di un insieme in parallelo per fornire un vantaggio.
Quindi, come ridurre il numero di cache miss? 1 usare cache set-associative, 2 usare una cache il più grande possibile, 3 avere delle linee grandi in modo da copiare più dati possibile dalla RAM

MIglioramenti

Si possono fare ancora alcune cose, in primis migliorare la velocità di accesso in RAM.
In genere, una linea della cache è più grande della quantità di dati che un banco di RAM è in grado di fornire ad ogni accesso, quindi ci vanno diversi accessi per prelevare un'intera linea. Questa cosa si può ottimizzare, aumentare la "banda", cioè quantità di dati prelevati da una lettura in RAM? si, funziona ma costa e fornisce poco vantaggio.
SI può usare le cosiddetta "memoria interlacciata" (interleaved memory).
Se una linea è formata da *n* word queste possono essere distribuite su *n* banchi di ram (allo stesso indirizzo) e lette in parallelo.
L'altra cosa che si può fare è la cache gerarchica.
nella cache L1 abbiamo un tempo di accesso simile a quello dei registri della CPU, nella L2 abbiamo una dimensione più elevata in modo da ridurre il numero di cache miss e stessa cosa per cache L3. così se abbiamo cache miss non dobbiamo andare fino in RAM, ma probabilmente basta salire di un livello di cache
Si può poi anche agire sul software.
Normalmente gli algoritmi sono valutati solo a livello teorico, ma spesso queste valutazioni non corrispondono alla realtà. Un algoritmo più efficente ma che causa molti cache miss è più inefficiente nella realtà

Sistemi paralleli espliciti

perchè espliciti? perchè a differenza delle tecniche viste prima in questo caso il programmatore deve scrivere l'applicazione tenendo conto del parallelismo.
Il più palese esempio è il multithreading.
Il parallelismo non può comunque mai essere assoluto, i programmi che girano in parallelo dovranno prima o poi sincronizzarsi per mettere in comune i dati elaborati da ciascuno.
Quindi un programma che gira in T tempo su un processore solo non girerà in T/4 su 4 processori, ma un po' di meno; quanto meno dipende dall'algoritmo. L'incremento di prestazioni è formalmente detto speed-up.
Formalizzando possiamo dire che:
Sia P un programma che gira in tempo T, ove f è la frazione di tempo T impiegata ad eseguire codice sequenziale (quindi 1-f è la frazione di T parallelizzabile).
Se ci sono n processori possiamo affermare che il tempo di esecuzione è $\frac{(1-f)T}{n}$
Il tempo di speed-up è definito dalla legge di Amdhal:
Tempo prima della parallelizzazione/ tempo dopo questa $\frac{fT+(1-f)T}{fT+(1-f)T/n}$ => $\frac{n}{1+(n-1)f}$
Risulta chiaro che l'unico modo di avere uno speed-up perfetto è avere f=0 ma un algoritmo ha SEMPRE delle operazioni non parallelizzabili.
Si può applicare la legge di Amdhal anche all' ILP ed il multiple issue.
Il Parallelismo presenta due problemi:

1. la quantità di codice parallelizzabile dipende dal problema che si vuole risolvere (e dal programmatore)
2. I vari processori possono dover comunicare, quindi usare spesso la memoria, quindi cache miss. (ed ecco perchè L3 è condivisa tra i Core)

Dividiamo i sistemi paralleli in 3: multithreading (stessa CPU, più thread contemporaneamente), sistemi a memoria condivisa (diverse unità elaborative, stessa memoria. un qualsiasi CPU multicore ne è un esempio), sistemi a memoria distribuita (ogni nodo computazionale ha la sua memoria privata, ci si scambia informazioni tramite un canale dedicato)

Multithreading

I thread condividono lo stesso spazio di indirizzamento.
L'idea di fondo è non sprecare inutilmente cicli di clock mentre si attende che un dato venga recuperato in RAM, quindi nel caso di cache miss. Infatti nonostante l'ILP è comunque questa l'operazione più ottimizzabile
Per implementare il multithreading la CPU deve poter gestire lo stato di computazione di ogni thread, quindi un PC ed un Register File per ogni thread.
Oltretutto bisogna tenere traccia del thread a cui appartengono le istruzioni in pipeline ed avere una cache un po' più grande della situazione "single thread" perchè bisogna assolutamente tenere in cache le istruzioni di tutti i thread, altrimenti si perde troppo tempo in cache-miss.

TIpi di multithread

Fine-grained multithreading

"a grana fine"
lo switch tra i vari thread avviene ad ogni istruzione, indipendemente dal fatto che il thread abbia generato un cache miss, secondo una politica round robin. Fra tutti i thread in attesa siamo statisticamente speranzosi del fatto che almeno uno di questi non sia in stall.
Teoricamente con questa tecnica la Pipeline non va mai in stall, infatti se la pipeline ha *k* stage e ci sono *k* thread c'è sempre una sola istruzione per thread nella pipeline.(questo naturalmente presupponendo che i cache miss richiedano meno di k cicli di clock).
Naturalmente però questo approccio è ingenuo: in primis un thread viene bloccato anche quando potrebbe proseguire, in secondo luogo imporre un numero di thread >= a *k* è una forte imposizione per il programmatore

Coarse-grained multithreading

"a grana grossa"
Lo switch avviene quando il thread va in stall, sprecando più di un ciclo di clock (il clock necessario ad accorgersi dello stall).
Nel caso ci siano pochi thread e/o questi thread generino stall raramente è più efficiente del fine grained in quanto il grosso del rallentamento avviene in fase di switch

Medium-grained multithreading

si esegue lo switch tra thread solo quando quello in esecuzione sta per eseguire una istruzione che potrebbe generare uno stall di lunga durata,come ad esempio una load (che potrebbe richiedere un dato non in cache), o un branch.

Multithreading (TLP) + multiple issue = SMT (Simultaneous Multi-threading)

In parole pover riusciamo a ficcare istruzioni appartenenti a thread diversi nella stessa pipeline.
Questo aumenta di molto il numero di istruzioni indipendenti avviabili nello stesso ciclo di clock. Il downside è che bisogna tenere conto di quali istruzioni sono di un thread e l'altro dell'altra, quindi servono più registri ed un ROB distinto per ogni thread.
Nonostante questi vantaggi vale davvero la pena di implementarlo, anche perchè come detto in precedenza l'ILP ha già raggiunto il suo massimo, non si può ottimizzare più di quanto non sia già stato ottimizzato.
Uno dei primi SMT è stato il cosiddetto "hyperthreading" nello Xeon del 2002.

Aumentando di solo il 5% la dimensione della CPU si può ottenere un incremento di prestazioni del 25-30%.
Un altro esempio notevole sono i processori UltraSPARC della SUN, che implementano fine-grained multithreading

Multiprocessori a memoria condivisa

Una architettura con più CPU che condividono lo stessa memoria primaria viene detto multiprocessore.
Tutti i processi vedono lo stesso spazio di indirizzamento (logico) ed ogni processo può accedere alla memoria tramite LOAD e STORE. quindi lo stesso sistema operativo può gestire tutte le differenti CPU.
Perciò vi è una separazione tra il SO e lo scheduling, dato che serve uno scheduler per ogni processore (ci può essere una coda di ready per ogni scheduler come no, di solito si tende a fare una coda di ready per ogni processore in modo da non dover implementare la mutua esclusione alla coda)
Le diverse code di ready portano però un nuovo problema: bisogna bilanciare il carico in modo che tutti i processori abbiano più o meno le stesse cose da fare.
Si distinguono sostanzialmente due classi di architetture multiprocessore, a seconda del modo in cui ogni CPU vede la memoria principale: i sistemi UMA e i sistemi NUMA.

Uniform Memory Access (UMA)

tutti i processori condividono la memoria primaria ed ogni CPU ha lo stesso tempo di accesso alla memoria.
Ci sono due o più CPU che si affacciano alla memoria tramite lo stesso bus, quindi questo diventa velocemente un collo di bottiglia. Difatti i processori UMA non scalano bene, finchè il numero di CPU è limitato va ancora bene.
L'altro problema è il problema di "coerenza della cache", che poi è il problema della mutua esclusione. un processore si copia in cache un valore ed un'altro modifica una variabile contenuta in quella linea di RAM, a questo punto il valore copiato dalla prima CPU è errata.
Una della soluzione, definita Snooping, consiste nel sorvegliare i bus di collegamento alla memoria.
Un semplice protocollo di coerenza è il write-through, ogni volta che avviene una write la modifica viene propagata alla memoria condivisa (di solito la RAM).
Quindi che il dato sia presente o no in cache in caso di scrittura viene sempre e comunque inviato alla memoria tramite il BUS condiviso.
Proprio su questo BUS può rimanere in ascolto l'altra CPU (quella che non ha fatto la scrittura) che se vede passare un dato presente nella sua cache segna quella entry della cache come invalida
A sto punto perchè non aggiornare direttamente la linea? infatti alcune architetture fanno così.
Il problema di questo protocollo è che fa una scrittura ad ogni write, generando sicuramente un collo di bottiglia.

Protocollo MESI (Modified, Exclusive, Shared, Invalid)

protocollo di coerenza basato su snooping che funziona anche nel caso di write-back.
L'acronimo mesi indica lo stato di una entry nella cache. Invalid, Shared (quella linea è stata copiata anche in altre cache), Exclusive (quella linea è contenuta solo in questa cache), Modified (questa linea non è aggiornata in RAM, il valore in cache è più recente del valore in RAM).

Quando una CPU C2 legge una linea già salvata in una cache di una CPU C1 lo snooper di C1 comunica a C2 che la linea ce l'ha lui, tutte e due le cache impostano la entry a Shared.
Se una entry viene modificata ma non ancora scritta in RAM lo snooper invia un segnale di invalidazione sul bus. Tutti gli altri processori impostano quella linea ad Invalid.
E se una CPU C3 tenta di leggere un dato contenuto in una linea segnata come Modified in C1? lo snooper di C1 vede che c'è una richiesta sul BUS e interviene dicendo a C3 di aspettare, a quel punto scrive la copia in RAM (inviandola contemporaneamente a C3).
Inversamente quando viene eseguita un write sulla RAM (comunque write-back ogni tanto scrive in RAM) il processore che la esegue avverte tutti gli altri snooper del fatto che il valore sta venendo aggiornato, gli altri processori o invalidano il loro valore in cache oppure lo aggiornano.

UMA crossbar switch

Per adesso abbiamo supposto che gli snooper siano in ascolto sul BUS in memoria, ma nella realtà si una un sistema a sé, in quanto l'uso di un BUS singolo crea un limite al numero di CPU che possiamo connettere.
Questo sistema segue la classica configurazione crossbar per mettere in comunicazione n CPU con n memorie, in modo che ciascuna CPU possa connettersi a ciascun banco di memoria.
Il circuito a "matrice" risultante genera però n^2 incroci, quindi il sistema scala molto male, per fare un sistema a 100 CPU c'è bisogno di 10000 switch.
Una soluzione è usare degli switch bidirezionali, che hanno 2 entrate e 2 uscite e permettono la comunicazione fra qualsiasi input e qualsiasi output (sia in un verso che nell'altro).
In questo modo possiamo mettere gli n processori in comunicazione con le n memorie con $(n/2)\log_2 n$ switch.
Il problema di questo circuito è che non tutte le sequenze di richieste possono essere servite contemporaneamente, se due comunicazioni passano dallo stesso switch.
Esempio pdf 07-multiprocessors pag 32.

Non Uniform Memory Access (NUMA)

I processori hanno uno spazio di indirizzamento (logico) condiviso ma la memoria fisica è distribuita tra le varie CPU (anche se accessibile a tutte le CPU)
I tempi di accesso ai dati variano a seconda che siano nella RAM locale o in una remota

Non Caching NUMA (NC-NUMA)

Non c'è cache, i processi richiedono dati direttamente alla memoria.
Vi è una MMU che controlla le richieste della CPU, indirizzandola alla memoria locale o ad una delle varie memorie remote.
Quindi è importante che i dati usati da una CPU siano nella memoria locale, esistono perciò dei software che si occupano di spostare le pagine ed i dati tra le varie memorie cercando di mantenere i dati "vicini"

Cache-Coherent NUMA (CC-NUMA)

ogni processore ha la sua cache, ma sta volta non possiamo usare lo snooping perchè non c'è un BUS solo che fornisce accesso alla memoria e fare un controllo crossbar switch annullerebbe tutti i vantaggi di scalabilità delle NUMA
Si usa quindi un protocollo directory-based: ad ogni nodo del sistema viene associata una directory, un database che registra in quale cache di quale nodo si trova ogni linea e qual'è il suo stato.

Quando si vuole accedere ad un dato si chiede prima alla directory del processore a cui quella linea appartiene per sapere se il dato in RAM è aggiornato oppure se c'è una copia più recente nella cache.
Questo processo deve essere rapido, quindi si usa spesso una memoria associativa per permettere una ricerca in parallelo su tutte le entry.
Quindi un indirizzo logico sarà così suddiviso | nodo | linea | offset |
Ad esempio prendiamo un sistema con indirizzi a 32 bit, 256 nodi (2^8), 16 MB di ram locale per ogni processore e linee da 64 byte.
I primi 8 bit dell'indirizzo indicheranno il nodo, 2^{24} byte di RAM divisi du 2^6 byte per ogni linea danno 2^{18} linee indirizzabili quindi con 18 bit, i restanti 32-17-8-6 = 6 bit indicano l'offset.
Quando viene richiesto l'indirizzo l'MMU legge solo i primi 8 bit ed indirizza verso il nodo corretto, la richiesta viene instradata dal nodo alla directory dove vengono letti i 18 bit relativi alla linea. se la linea non è nella cache di qualche nodo remoto la linea viene prelevata dal banco di RAM e nella directory viene scritto che adesso la linea si trova nella cache del nodo che ha fatto la richiesta.
Se la richiesta riguarda invece una linea che si trova nella cache di un'altro processore questi si deve occupare di inviare la entry al nodo che l'ha richiesta e di invalidarla nella sua cache.
Sembra esserci molto traffico, ma in realtà l'overhead generato è relativamente basso

Sincronizzazione tra processi

Immaginiamo il classico caso della mutua esclusione, più processi usano una variabile condivisa. Girando su CPU diverse, anche se si usano operazioni atomiche si potrebbe avere due differenti processi che accedono contemporaneamente alla stessa variabile.
Si usano 2 istruzioni, una di seguito all'altra : LL (load linked): carica un valore da un indirizzo di memoria ad un registro, SC (store conditional): tenta di scrivere il contenuto di un registro in memoria.
La SC ha una particolarità, se il valore all'indirizzo di memoria specificato è != dal valore che si sta scrivendo fallisce. Allo stesso modo se si è verificato un context switch tra la LL e la SC fallisce.
In questa maniera se il valore all'indirizzo specificato viene modificato oppure si verifica un context switch la SC ritorna 0 e questo valore può essere usato per creare una operazione atomica (ad esempio una exchange atomica):

```
retry:  OR   R3, R4, R0   // copy value of R4 in R3
        LL   R2, 0(R1)    // load linked: copy [0(R1)] in R2
        SC   R3, 0(R1)    // try to store value of R3 in 0(R1)
        BEQZ R3, retry    // try again if SC failed
        MOV  R4, R2       // now put loaded value in R4
```

Se SC ha successo scrive il valore di R4 (che è il valore da scambiare ed è stato copiato in R3) dentro la cella di memoria che rappresenta la variabile. se la SC fallisce in R3 viene scritto 0 e si esegue il salto condizionato
Tramite la exchange atomica (EXCH) in pratica si mantiene il sistema in *lock spinning* finché in 0(R1) non c'è il valore 0, in pratica abbiamo creato un lock, quindi si può sviluppare qualsiasi tipo di sincronizzazione.

ATTENZIONE: comunque si sta facendo un sistema di sincronizzazione basato su busy waiting.
Questa parte è un po' complicata, pdf 07-multiprocessors pag 54 per rivederla.

Consistenza (Memory Consistency)

in che ordine una CPU deve vedere le modifiche sulle celle di RAM effettuate dalle altre CPU?
La trasmissione del segnale infatti non è istantanea, presupponiamo 3 CPU che eseguono:
CPU A: write #1, X
CPU B: write #2, X
CPU C: read X
Non è detto che C legga il valore #2 poichè il segnale ci mette un attimo a propagarsi, si può forzare questa condizione, inserendo un interfaccia fra le varia CPU e la memoria garantendo la cosiddetta "consistenza stretta". Ma questo ovviamente genera un collo di bottiglia dato che l'accesso alla memoria diventerebbe sequenziale.
(a che mi serve avere 16 CPU se poi accedono una alla volta in memoria?)
Quindi si usa la cosiddetta consistenza del processore (processor consistency), che garantisce due proprietà:

1. Le scritture da parte di una qualsiasi CPU sono viste dalle altre CPU nell'ordine in cui sono state avviate. Se CPU 1 scrive A, B e C in una locazione var, una CPU 2 che legga var in sequenza più volte leggerà prima A, poi B e poi C.
2. Per ogni locazione di memoria, qualsiasi CPU vede tutte le scritture effettuate da ogni singola CPU in quella locazione nello stesso ordine.

Esempio: C1 scrive in *var* A,B e C. contemporaneamente C2 scrive in *var* X, Y, e Z.
CPU 3 potrà leggere: A, B, C, X, Y, Z. Mentre CPU 4 potrà leggere X, Y, Z, A, B, C
Ma nessuna CPU potrà leggere una sequenza come B, A, C, X, Y, Z

Sistemi a memoria distribuita (Multicomputers)

C'è un limite alla scalabilità dei sistemi multi-processore, la memoria sempre quella è e le variabili a cui si vuole accedere sempre quelle sono.
Quindi su distribuiscono i sistemi, si crea la cosiddetta architettura multicomputer ove i vari nodi comunicano solo attraverso lo scambio di messaggi e non esiste uno spazio di indirizzamento comune.
Queste architetture sono chiamate NORMA (NO Remote Memory Access)
I più comuni multicomputer sono gli MPP (Massively Parallel Processors), comunemente conosciuti come supercomputer.

Progetto BlueGene (MPP)

nasce nel 1999, con il focus di creare non tanto il computer più potente, ma di creare il computer più "economicamente vantaggioso". Viene usato un processore economico e lento, sopperendo alla qualità con la quantità.
Vi erano 2 core in ogni processore, uno usato per i calcoli, uno per le comunicazioni I/O con gli altri processori. Infatti in un MPP non solo deve calcolare rapidamente, ma anche condividere i suoi calcoli rapidamente.
su ogni processore girava una versione leggera di linux, monoutente e con 2 soli thread, appunto uno di calcolo ed uno di comunicazione.
Ogni core aveva accesso alla propria RAM, creando un sistema molto scalabile e modulare.

Vi erano poi delle periferiche di I/O ed una connessione ethernet per la manutenzione e l'uso "quotidiano" (anche solo per inserire input ed ottenere output).
BlueGene non verrà mai costruito, verranno poi progettate ed installate 2 versioni (BlueGene/P e BlueGene/Q) più potenti (nel senso che seguono la stessa architettura ma hanno processori più potenti)

COW: Clusters Of Workstations

Cosa li distingue dagli MPP? non sono "identici" e modulari, oltretutto le singole unità funzionali non possono funzionare "da soli"
I COW sono molto più eterogenei, le macchine possono essere distinte e comunicano tra loro tramite Ethernet, non tramite una rete creata appositamente.
Google è il cluster di workstation per antonomasia, il loro COW ha diffusione globale

Processori vettoriali

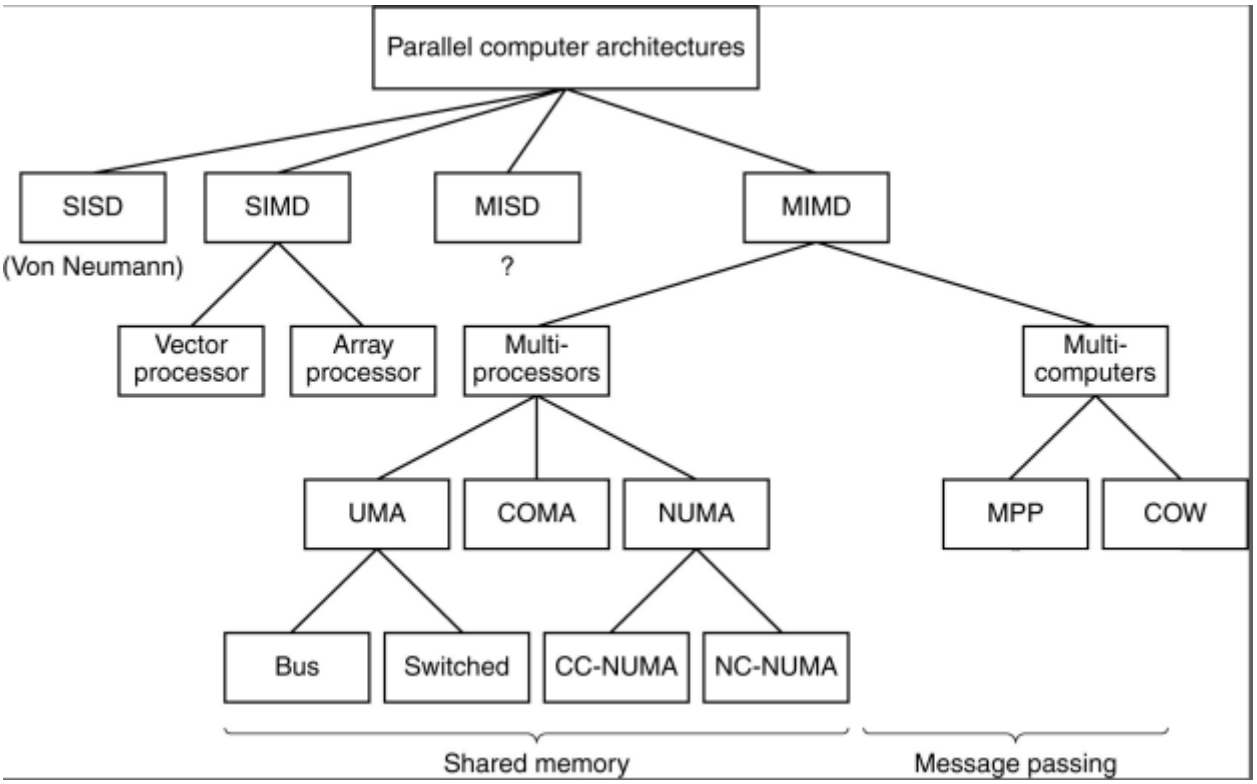
I processori vettoriali lavorano su vettori di dati, ossia, in sostanza, su array di numeri.
La loro architettura è esplicitamente indirizzata a gestire array, quindi memorie apposite e operazioni apposite.
Le istruzioni sono molto interessanti:

1. con una sola istruzione si specifica molto lavoro, con un solo comando fai praticamente quello che faresti in un loop in architetture normali
2. non vi può essere dipendenza sui dati o sui nomi all'interno del vettore (fra due vettori però sì), quindi quel controllo si può saltare
3. gli elementi dei vettori vengono salvati in locazioni adiacenti, probabilmente quindi tutto il vettore si trova solo su una linea di RAM

Quindi una architettura vettoriale avrà componenti apposite: registri vettoriali, unità funzionali vettoriali, unita di load-store vettoriali ma anche dei registri scalari (ogni tanto può servire voler salvare solo un numero)
I processori vettoriali non hanno mai veramente preso piede. La complessità del datapath fa si che la frequenza di clock sia molto bassa, oltretutto il loro costo è maggiore in quanto le unità funzionali sono più complesse e la connessione con la memoria più sofisticata.
Infine sono troppo specifici, sono efficienti solo sulle operazioni vettoriali che ,per quanto siano comuni, sono la minoranza dei casi.
Tuttavia l'esperienza progettuale dei processori vettoriali è stata almeno parzialmente trasferita nei processori superscalari moderni, in alcuni casi sono state pure aggiunte istruzioni vettoriali alle ISA.

Tassonomia

Presentiamo la tassonomia di Flynn, che si basa su due concetti: instruction stream (parallelismo sulle istruzioni) Data stream (parallelismo sui dati)
questi due concetti possono essere combinati in 4 possibili modi.
Uno stream di istruzioni è semplicemente un programma, un sistema *n* core può far procedere in parallelo *n* flussi di istruzioni
in uno stream di dati ogni dato è computato a partire dal dato precedente.



SISD Single instruction single data

la classica architettura monoprocessore, le istruzioni vengono eseguite una alla volta su un unico flusso di dati.
Non è presente una pipeline.
Nella realtà vengono usati solo per i dispositivi embedded

MISD: Multiple Instruction (stream) Single Data (stream)

Più flussi di istruzioni che operano sullo stesso flusso di dati.
Non ha senso nella realtà… può succedere che più flussi di istruzioni lavorino sullo stesso flusso di dati, ma perchè obbligarli a farlo? se hai multiple instruction tanto vale fare anche multiple data, all'occorrenza usi un flusso di dati solo.

SIMD: Single Instruction (stream) Multiple Data (stream)

I processori vettoriali, con un singolo stream di istruzioni riesco a gestire multipli flussi di dati. Anche le GPU.

MIMD: Multiple Instruction (stream) Multiple Data (stream):

Le architetture comunemente usate: multicore, multiprocessore o multicomputer

Quantum Computing

Non so se qualcuno leggera queste slides a parte me…ma non ho intenzione di riascoltare una lezione sul principio di indeterminazione di Heisenberg o l'eq di Schrodingher….quindi saltiamo al qbit. (nel caso https://www.youtube.com/watch?v=JhHMJCUmq28)
Un sistema quantistico in superposizione è contemporaneamente in più stati, un qbit può contemporaneamente valere sia 0 che 1 finchè non lo misuriamo.

in questo risiede il potenziale della computazione quantistica: corrisponde ad avere un computer capace di eseguire un algoritmo processando in parallelo tutti i possibili input ammessi per quell'algoritmo.
Il trucco quantistico sta nel misurare il valore dei qbit solo quando l'elaborazione è completa, sfruttando la superposizione per svolgerla.
Come nel caso dei computer normali si possono unire più bit per creare dati più complessi, usiamo la notazione matriciale da veri fisici quantistici (qui nel caso dei bit comuni)

$$|00\rangle = \begin{matrix}00\\01\\10\\11\end{matrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad |01\rangle = \begin{matrix}00\\01\\10\\11\end{matrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad |10\rangle = \begin{matrix}00\\01\\10\\11\end{matrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad |11\rangle = \begin{matrix}00\\01\\10\\11\end{matrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

la matrice 1x*n* indica la probabilità che il bit assuma un certo valore.
Stessa cosa vale per i dati quantistici, solo che si usano i numeri complessi

Mentre una generica coppia di qubit sarebbe rappresentata così:

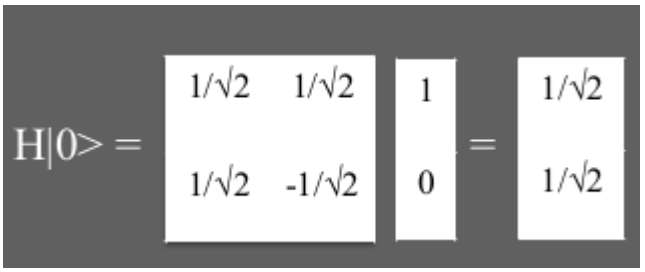
$$\begin{matrix}00\\01\\10\\11\end{matrix}\begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix}$$

con la condizione che:
$|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2 = 1$

ognuna delle entry di questa matrice corrisponde alla probabilità che questa combinazione si manifesti. per questo c0^2 + … + cn^2 + … + c3^2 deve fare uno.
questi sono i bit, e le porte logiche?
Iniziamo a rappresentare le porte logiche comuni tramite matrice:
I bit di informazione in input possono essere rappresentati come una matrice di 2 n righe x 1 colonna, e gli m bit di output saranno rappresentati con una matrice di m righe x 1 colonna.
Quindi volendo ottenere una matrice m x 1 (output) da una matrice n x 1 (input) la porta logica deve essere una matrice n x m date le proprietà di moltiplicazione delle matrici.
ed è proprio così, la porta logica NOT è questa

$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$$

Quindi possiamo costruire la NAND, e si sa che una volta costruita la NAND si può implementare qualsiasi circuito logico
Nel caso di porte logiche complesse non abbiamo uni e zeri bensì numeri complessi in input
nell'esempio si seguito la matrice di Hadamard che prendi il qbit di input e lo mette in una sovrapposizione di stati per cui il qbit di output è per metà nello stato |0> e metà nello stato |1>



Le porte logiche devono avere ancora una caratteristica per essere usate per computazioni, devono essere reversibili, sapendo un output bisogna poter dedurre l'input. Stesso discorso vale per le porte logiche quantistiche.
il vero problema progettuale è che non possiamo misurare i qbit, pena il crollo della superposizione, dobbiamo attendere la conclusione dell'elaborazione per misurare.
Oltretutto c'è il problema della decoerenza quantistica, il fenomeno per cui i qbit tendono a "decadere" in uno stato stabile a causa delle influenze ambientali, i bit classici.
Infatti l' entanglement quantistico fa sì che per quanto si tenti di isolare il sistema ci sarà influenza ambientale. Ma il problema è ancora più subdolo, dato che non misurando i qbits fino al completamento non c'è modo di verificare se è avvenuta decoerenza o no.
Ci sono solo 3 modi attualmente di ridurre il rischio di decoerenza: mantenere la computazione il più breve possibile dato che la decoerenza si verifica più o meno dopo 100 microsecondi, usare una forma di correzione degli errori implementata tramite qbit oppure ripete più volte lo stesso calcolo e prendendo il risultato che capita più di frequente

Algoritmi quantistici

Non esistono criteri generali degli algoritmi quantistici, ogni algoritmo è studiato ad hoc come combinazione di porte quantistiche.
Un esempio di algoritmo quantistico semplice è l'algoritmo di Deutsch-Joza fatto giusto per dimostrare la potenza dei computer quantici.
La soluzione classica ha complessità esponenziale rispetto ad n mentre quella quantistica ha soluzione lineare.
Data una funzione f{0,1} -> {0,1] l'algoritmo ci dice se la funzione è bilanciata o costante (slide 29 di 09-quantum-computing per la definizione), per farlo esegue dei confronti sul risultato della computazione di quella funzione, quindi bisogna fare 2^n confronti ove n è il numero di possibili funzioni
Perchè il quantum computing lo rende lineare è lineare? perchè possiamo valutare 2 volte la funzione (input 0 ed input 1) contemporaneamente.
Ci sono anche altri algoritmi che non vediamo, il più interessante è l'algoritmo di shor che permette di fattorizzare un numero in $(\log N)^3$

Supremazia quantistica

Nell'autunno del 2019 Google annunciava di aver raggiunto la Quantum Supremacy: il loro computer quantistico, il Sycamore, aveva effettuato in 200 secondi un calcolo che avrebbe richiesto 10mila anni al supercomputer più potente del mondo.
Ma cos'è la supremazia quantistica? è lil momento in cui un computer quantistico può risolvere un problema che nessun computer classico potrebbe risolvere in tempi ragionevoli.
Si ma che problema? questa definizione è appunto troppo generica, non dice nulla sull'utilità del problema. Bello ma inutile risolvere un problema insignificante molto in fretta.
Il punto è che di tantissimi problemi non si conoscono ancora algoritmi risolutivi quantistici.

GPU

Attenzione:

Mentre la parte di Gunetti è perfettamente possibile prepararla seguendo solo le slides (magari con l'aiuto di questi riassunti che sono abbastanza completi), la parte di Aldinucci NO. Fatevi un piacere e seguite le sue lezioni: consiglio le videolezioni del 2020/2021, non quelle del 2021/2022.
(le frasi sottolineate sono dirette citazioni del prof o del libro)

Nascono come coprocessori grafici, sarà poi NVIDIA a proporre il loro uso in campi diversi, sviluppando un linguaggio di programmazione apposito per questo genere di calcoli, CUDA.
Immaginiamo lo schermo come una sezione di memoria logica, colorare un pixel significa inserire un 1 in una precisa locazione. La GPU si occupa di fare proprio questo, ma non solo, anche di calcolare "dove inserire 1". In particolare le GPU devono poter calcolare trasformazioni e rotazioni di oggetti 3d su uno schermo 2d quindi operazioni matriciali.
Le GPU sono architetturalmente molto adatte a svolgere queste operazioni, sfruttando un elevato parallelismo. D'altra parte non sono in grado di lavorare "da sole", necessitano della supervisione e del controllo della CPU (la CPU fa offloading).
In generale, le GPU sono particolarmente adatte a problemi caratterizzati da un elevato livello di parallelismo nei dati e a problemi di dimensioni assai rilevanti, mentre sono meno indicate per problemi più piccoli e meno regolari.

Effetto dark silicon

La vera potenza delle GPU sta nel fatto che devono svolgere operazioni molto specifiche, quindi la loro architettura hardware può permettersi di essere più compatta, più semplice e meno "intrecciata".
Non è che sono più potenti e basta, più che altro riescono a svolgere certi calcoli specifici consumando molta meno corrente e dissipando molto meno calore.
Qui entra in gioco l'effetto dark silicon, anche presupponendo di poter consumare infinita energia e avendo una dissipazione perfetta una CPU non arriverà mai alla potenza di una GPU in quanto ad un certo punto il silicio stesso non è in grado di dissipare il calore abbastanza in fretta,
In pratica non si può usare il 100% di una scheda sempre, altrimenti "si brucia".
Questo prende il nome di effetto dark silicon, ad ogni momento solo il 15% dei transistor può essere usato contemporaneamente, il resto è "silicio spento/buio" a causa di limiti termici (ma anche elettrici)

Memory wall

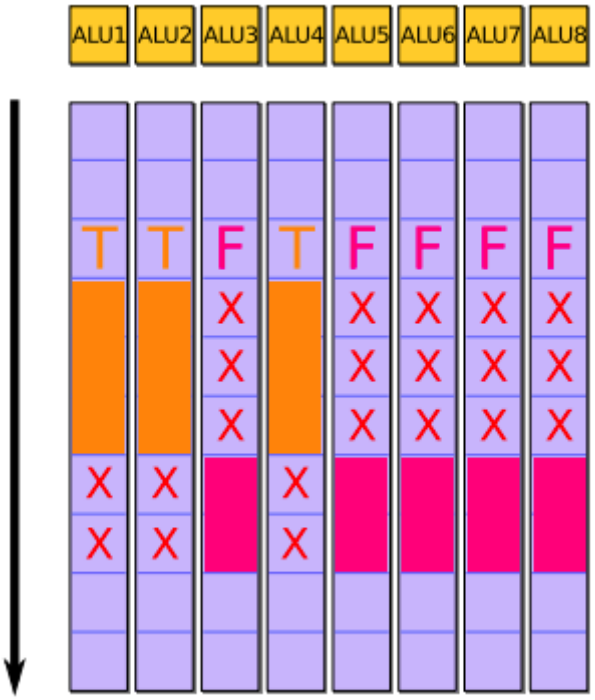
The "memory wall" is the growing disparity of speed between CPU and memory outside the CPU chip.

Differenze architetturali tra CPU e GPU

1.core semplici ma tanti.
Quindi, rimuovere tutto quello che non fa calcoli: cache, schema di Tomasulo, ILP.
Al posto di un solo core complesso si usano tanti core semplici, che fanno solo calcoli.
quindi i dati e i dati e le istruzioni inviate alla GPU devono essere pre-parallelizzate ed organizzate.

Una GPU alla fine è un grosso insieme di processori, i processori pipeline, solo che appunto non ce n'è uno, ma un gran numero, per quello le operazioni devono essere pre-parallelizzate

2. da SIMD (single instruction multiple data) a SIMT (single instruction multiple thread)
Oltretutto tutti questi core non possono effettuare operazioni troppo diverse fra loro, altrimenti dovrebbero tutti accedere alla memoria (soprattutto per leggere le istruzioni) creando bottleneck.
(qui entra anche in gioco il problema del Memory Wall, le CPU sono sempre più potenti e le memorie no, quindi il distacco fra le prestazioni causa molto bottleneck)
Quindi si usa l'idea 2, faccio la stessa operazione su diversi dati contemporaneamente (ad esempio somma elemento-elemento di un array).
Questo approccio è chiamato data parallelism, partiziono i dati e li distribuisco su diversi nodi computazionali contemporaneamente
Il programma deve essere riscritto di conseguenza, anche per evitare dipendenza tra dati/nomi e tra istruzioni.
SIMD ha un enorme problema, non puoi effettuare branching, quindi è stato abbandonato molto in fretta; L'idea è stata però riusata ed implementata in nuovi sistemi (intel SSE o intel AVX) come "parte" di un sistema più complesso
c'è una soluzione, la Branch serialization.



I core effettuano il controllo della condizione, ma indipendentemente dal risultato vengono "eseguiti" entrambi i branch. durante il branch corretto si eseguono i calcoli, durante il branch errato si attende.
Questo ovviamente ha senso solo per branch corti e la complessità aumenta molto con i cicli annidati.
Addirittura potrebbe essere più conveniente calcolare 2 volte una variabile che usare un if e calcolarla solo una volta.
Quindi nelle GPU non si utilizza un approcio SIMD, bensì si tende ad usare SIMT (single instruction multiple thread). MA SIMT È SIMD + BRANCH SERIALIZATION?? Si, alla fine sì.
3. Interleaving (latency hiding)
Mentre aspetto dei dati dalla memoria eseguo altre operazioni.

È diverso dall' ILP, non vado ad eseguire altre parti del programma, vado ad eseguire un altro "blocco" di istruzioni indipendenti mentre attendo (in realtà eseguo un altro warp)
Naturalmente devo avere un altro blocco di istruzioni indipendenti in coda.
in pratica avvio la richiesta dei dati e poi passo ad un altro warp.
Oltretutto nelle GPU non abbiamo dei circuiti "multiple purpose", quindi in ogni scheda sono presenti alcuni (la quantità è variabile) core specifici per effettuare load/store.
Ci sono proprio dei circuiti apposta che fanno solo LD/STORE.
Questo non è raro nelle GPU dato che sono molto specifiche, per poter eseguire operazioni diverse serve uno specifico circuito. Quindi in una GPU è comune trovare 4 diversi tipi di core: LD/ST , Compute (unità processuali, chiamati core o SP streaming processors, in quanto ricevono uno stream di dati e vi operano sopra), SFU (Special Function Units, tipo seno, coseno, radici), Tensor core (core adatti a svolgere operazioni matriciali, in pratica fa molto velocemente multiply-accumulate `a = a + (b * c)` andando a rendere super rapide le moltiplicazioni tra matrici)
Tornando al discorso di interleaving: il core LD/ST rimane in attesa finchè non riceve il dato, ma nel mentre gli altri core vengono usati.
4 cache esplicita. Abbiamo cancellato tutti gli strumenti necessari alla traduzione degli indirizzi, ma la cache è troppo utile per rinunciarci. Anche qui, la soluzione è mettere il programmatore al corrente dell'esistenza della cache (la cache non è più nascosta)
cache deriva dal francese "cache" nascosto.(non serve a nulla saperlo ma il prof continua a ripeterlo)

Condizioni di Bernstein

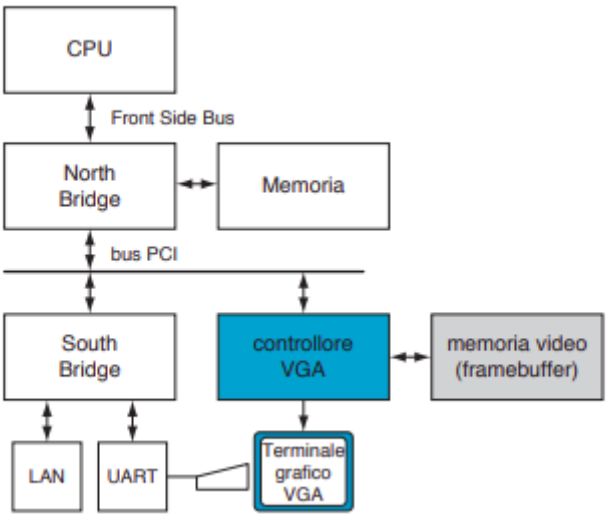
Siano A e B due istruzioni: definiamo come domain(A) le aree di memoria da cui A dipende, da dove va a leggere i dati di input. definiamo come range(A) le aree di memoria che dipendono da A, le locazioni di memoria modificate durante l'esecuzione di A.
è la stessa cosa di una funzione matematica, il dominio è il valore da cui "parte" il range è l'insieme di valori a cui può "arrivare"

- range(istruzione A) ∩ range(istruzione B)= ∅
- range(istruzione A) ∩ domain(istruzione B)= ∅
- domain(istruzione A) ∩ range(istruzione B)= ∅

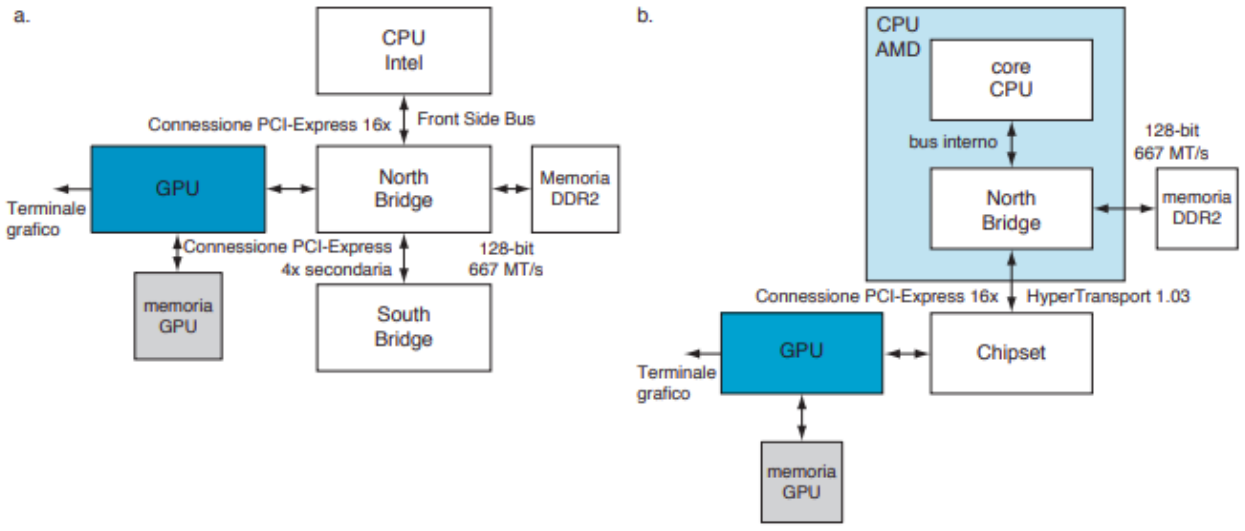
Se due (o più) istruzioni soddisfano le condizioni di Bernstein il risultato è indipendente dalla particolare sequenza di esecuzione eseguita dai processori (interleaving) e sarà quindi identico alla loro esecuzione seriale.

Comunicazione GPU e sistema

L'immagine mostra un sistema "storico", la GPU è addirittura chiamata Controllore VGA ed è collegata tramite un bus secondario, più lento e condiviso con altri sistemi I/O.



Attualmente la situazione è molto cambiata, nelle foto successive (a intel, b amd) vediamo che la CPU è praticamente collegata direttamente al processore (configurazione co-processore) tramite il north bridge.
Di solito la connessione è effettuata tramite PCI-E per garantire una rapidità di trasmissione.
La GPU è un acceleratore, per quanto complessa non ha le strutture necessarie a lanciare un programma, è sempre “slave” della CPU. ripeto, la CPU fa offloading.
Il vantaggio di questa architettura (a parte la semplicità) è che la GPU può essere usata in contemporanea da processi differenti, esempio più classico: parte della GPU renderizza la gui ed il resto è usato per effettuare calcoli (tipo ML) contemporaneamente.
Questa affermazione sottintende il fatto che i vari core della GPU possono essere usati indipendentemente.



Addirittura le evoluzioni future Nvidia prevedono di integrare una CPU all’interno del “socket” GPU (non ho dati a riguardo) (in parole povere nella stessa scheda, è lo stesso discorso della memoria “SiP” che viene installata nel socket delle CPU Apple M1)

Blocchi

Come abbiamo visto prima è fondamentale gestire per bene il prelevare dati dalla RAM (che sia RAM o GraphicalRAM), anche perché spesso capita di dover “suddividere” il

problema in parti più piccole. Per quanto i core siano tanti (5000 e più) capita spesso di avere vettori con più elementi di quanti core ci sono disponibili.
è quindi insperabile riuscire a caricare nella cache della GPU tutto in un colpo solo, ed è qui che entrano in gioco i blocchi.
I blocchi sono sottoinsiemi di elementi, ogni volta che andiamo a leggere in ram carichiamo un “blocco”

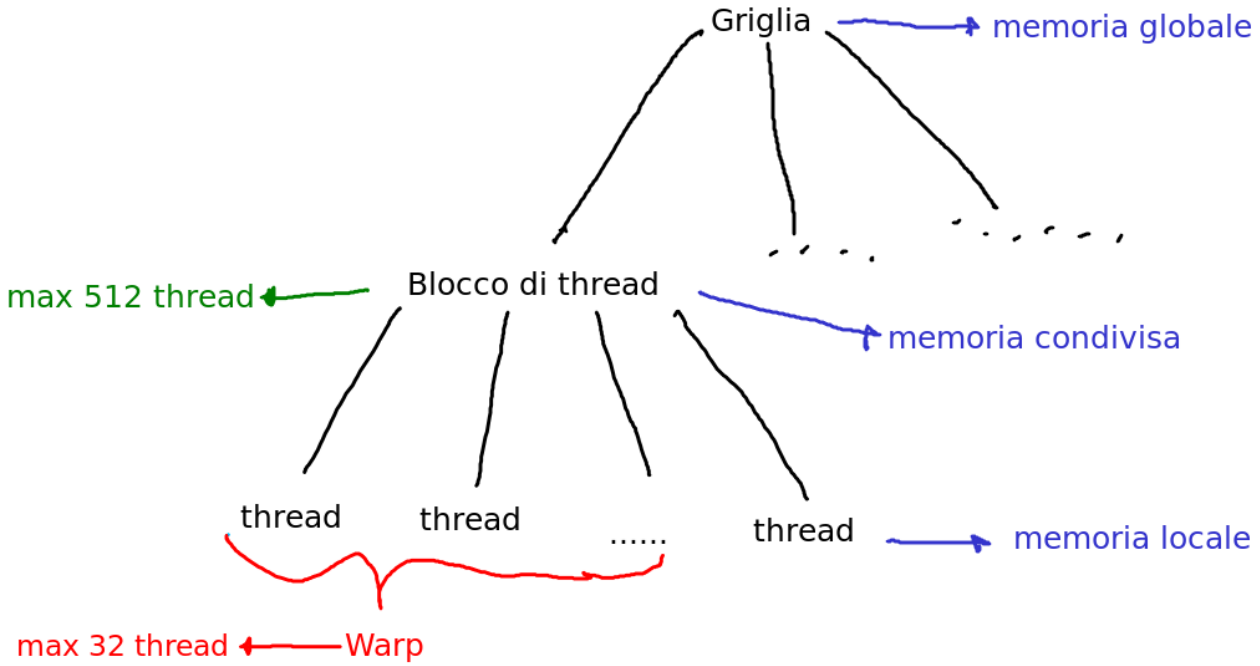
Threading

Le GPU si basano pesantemente sul multithreading, Il programmatore scrive un programma sequenziale che richiama kernel paralleli, i quali possono contenere semplici funzioni o programmi completi. Un kernel viene eseguito in parallelo su un insieme di thread paralleli. Il programmatore organizza questi thread in una gerarchia di blocchi di thread e di griglie di blocchi di thread. Un blocco di thread è un insieme di thread concorrenti che possono collaborare tra loro mediante sincronizzazione a barriera e accesso condiviso allo spazio di memoria privato del blocco. Una griglia (grid) è un insieme di blocchi di thread che possono essere eseguiti ciascuno in modo indipendente, quindi in parallelo.

Il numero di thread per blocco ed il numero di blocchi per griglia è specificato dal programmatore e viene dinamicamente impostato quando viene lanciato in esecuzione il kernel.

```
kernel<<<dimGriglia, dimBlocco>>>(... elenco parametri ...);
```

Ad ogni thread viene assegnato un ID *threadidx* relativo al suo blocco e ad ogno blocco un ID relativo alla sua griglia, *blockIdx.*
Durante la loro esecuzione, i thread possono accedere a dati appartenenti a diversi spazi di memoria, ogni thread dispone di una memoria locale privata (usata soprattutto in caso di “spilling”, cioè quando i dati privati di un thread superano la dimensione dei registri. Identificata da nessuna keyword nel programma CUDA) e può accedere alla “memoria condivisa” (una per ogni blocco, specificata nel programma CUDA dalla keyword _shared_) per comunicare con gli altri thread. per permettere alle griglie di comunicare, si usa la Memoria globale (specificata in programma CUDA dalla keyword _device_).



Ogni thread della GPU dispone dei propri registri privati, di memoria privata dedicata al thread, di un registro program counter e di un registro dello stato di esecuzione del thread, ed è quindi in grado di eseguire un frammento di codice in maniera indipendente
Ma lavorando su memorie comuni c'è bisogno di inserire meccanismi di sincronizzazione (mutua esclusione) per evitare conflitti.

La mutua esclusione richiede interleaving, è troppo “demanding”, il meccanismo più semplice e che viene implementato sulle GPU è il **sync**: un meccanismo che permette di garantire che tutti i processi abbiano eseguito una istruzione. (una specie di attesa fra thread, se io ho finito prima aspetto che gli altri arrivino dove sono io).
La global memory delle GPU usa un modello di memoria Weak (sottogruppo del modello Relaxed) (abbiamo visto cos’è la consistency nella prima parte di corso Consistenza (Memory Consistency). questo tipo di consistency è “io non garantisco proprio nulla”) quindi non è garantito l’ordine ne di scrittura in memoria ne di lettura. se si vuole garantire questi bisogna usare la sync, ma la sync è un costrutto che rallenta molto l’esecuzione, quindi va usata solo quando davvero necessario.
Il prof porta in esempio anche il modello di consistenza sequenziale di Lanport:
la scrittura e la lettura della memoria sono sequenziali, solo un processore alla volta accede alla memoria.
Abbiamo poi il total store order (tutte le scritture avvengono una alla volta, le letture no) e weak ordering: abbiamo dei costrutti chiamati fence o barrier, ma tra una fence e l’altra l’ordine può essere qualsiasi
le fence sono operazioni di sincronizzazione , chiedono al processore di essere sicuro di aver completato tutte le letture/scritture e solo a quel punto permettono al programma di ripartire.

Farm

Eseguire la stessa funzione su elementi diversi, su unità di esecuzione diverse.
Sembra quello che abbiamo visto fin’ora, dove sta la differenza?
In una farm l’operazione sugli elementi viene fatta in maniera sequenziale.
I dati vengono spediti ad unità funzionali dette Worker in ordine (vettore[0], vettore[1], vettore[2].....vettore[n]), i Worker effettuano l’operazione desiderata sull’elemento ricevuto e trasmettono il risultato. I risultati vanno naturalmente riordinati correttamente.
è interessante perchè posso scegliere quanti worker usare e quanti dati inviare ad ogni worker. (Un farm l’abbiamo già visto durante il capitolo sulla pipeline, al sottocapitolo Pipelines più sofisticate).

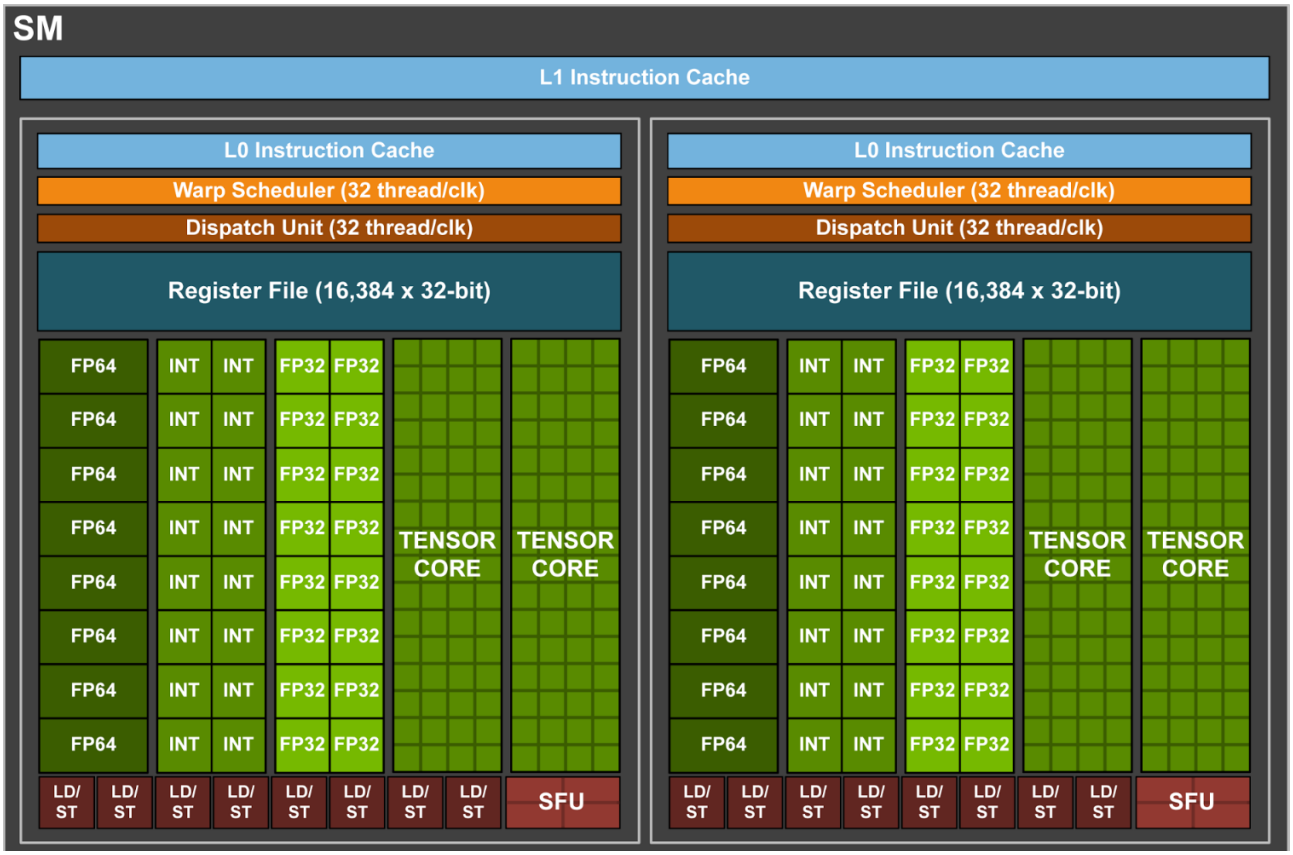
Warp scheduling

A warp is a set of 32 threads within a thread block such that all the threads in a warp execute the same instruction
Un warp consiste in massimo 32 thread (tutti dello stesso blocco), l’unità garantisce che tutti i thread all’interno di un warp sono veramente eseguiti contemporaneamente. Sono sicuro che ognuno di quei 32 thread si trova alla stessa istruzione, per sincronizzare l'esecuzione tra diversi warp bisogna ricorrere alla _syncthread.
Quando un thread esegue un’istruzione bar.sync, incrementa il contatore del numero di thread arrivati alla barriera e lo scheduler registra che il thread è in attesa, in modo da non avviarlo.
Quando un warp arriva allo SM il warp scheduler si occupa di prelevare una istruzione da un warp e mandarla in esecuzione, poi esegue una istruzione di un’altro warp (che sarà sicuramente indipendente.), e così via. in modo da interlacciare operazioni che si inchiodano sulla memoria con operazioni che hanno già i dati a disposizione.
Mentre ci sono delle load è possibile eseguire delle istruzioni indipendenti. quando la load sarà completata tornerà in coda di ready per essere eseguita. però attenzione, l’esecuzione è in order, non abbiamo speculazione hardware o pipeline. questo discorso è relativo solo alla memoria, è latency hiding.

È l'hardware a rendersi conto di quando i dati sono disponibili, i processori di load/store rimangono in "attesa attiva" finchè non hanno ricevuto il dato, il warp Scheduler si occupa solo di individuare le istruzioni che sono indipendenti a livello di dati (cosa molto semplice dato che un warp è sempre indipendente dagli altri) ed eseguirle.

Hardware vs Software

una GPU contiene molti processori core, tipicamente organizzati in multiprocessori multithread.
In cima alla gerarchia abbiamo lo streaming multiprocessor (SM) a cui vengono affidati da eseguire warp.
I Warp sono set di max 32 threads, invece i blocchi di thread possono essere anche grandi fino a 512 thread,
Allo SM viene affidato un blocco di threads, è quindi probabile che riceva più di un warp; è qui che entra in gioco lo scheduler, che si occupa di lanciare i warp i cui dati sono disponibili.
A livello software abbiamo le 3 memorie: locale, condivisa, globale, a livello hardware dipende molto dall'implementazione. alla memoria locale e condivisa corrisponde una stessa memoria (nell'architettura Fermi questa è una memoria rapida all'interno dello SM), alla memoria globale corrisponde una memoria più lenta, (nel caso dell'architettura Fermi la DRAM della GPU).
Come si può notare dalla figura sottostante abbiamo anche il componente che si occupa di fare "farming", la Dispatch Unit (occhio che il prof non ne ha mai parlato di dispatch unit) che si occupa di inviare le istruzioni ai diversi worker (FP64, INT, FP32, Tensor, ld/st, SFU) presenti nello SM. .

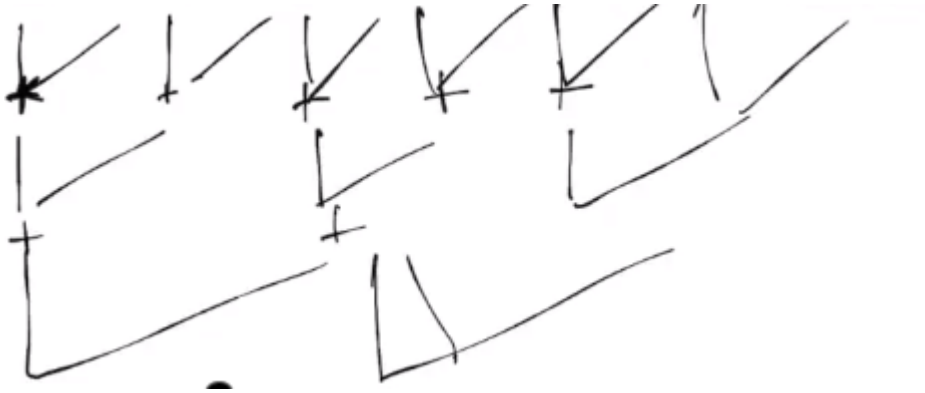


Nell'immagine sono presenti 2 cache L0, due warp scheduler e sembrano esserci 2 "unità" senza nome.
Ecco, perchè queste unità non esistono veramente, uno SM può avere anche un solo warp scheduler. le NVIDIA Fermi ne hanno due, le VOLTA 4 (l'immagine sopra è presa e

ritagliata proprio dalla documentazione VOLTA) ma credo questa sia solo una scelta progettuale.
citando la documentazione ufficiale VOLTA: “The GV100 SM is partitioned into four processing blocks”....quindi non hanno un nome.

Map Reduce

Map reduce è un paradigma di calcolo parallelo.
Map (oppure apply to all) è una operazione che esegue una operazione su tutti gli elementi di un array.
Reduce è un operazione che applica una funzione (che deve essere una operazione associativa (somma, moltiplicazione, etc))su una lista, ad esempio sommare tutti i valori di un array (come Stream.reduce())
Google ha creato la sua versione nel 2007, ma in realtà esisteva già, infatti (slegato da google) negli stessi anni viene implementato un map reduce per le GPU.
Quindi attenzione, sono simili ma non sono la stessa cosa (quello di google è un map sort reduce by key)
Vi è poi il Farm, un sistema che “processa” sequenzialmente gli elementi dell’array distribuendoli ad esecutori diversi.
Il modo più semplice di effettuare reduce parallelamente è quello di effettuare l’operazione su coppie di valori: ad esempio fare una reduce somma di un vettore di 12 elementi si può fare così: (i numeri non sono numeri, ma la posizione nel vettore)



e poi sommare gli ultimi 2 risultati
Questa cosa è super parallelizzabile, basta che ad ogni thread assegno una coppia di elemenenti, e vado a scrivere il risultato come input della prossima “somma”
in pratica

```
data[tid] += data [tid +s]
```

dove tid è “thread id” ed s è un numero che viene incrementato di potenza di due ogni ciclo (2, 4, 8)
Si può ancora ottimizzare: nel nostro vettore di 10 elementi al terzo passo avremo due thread che lavorano e tutti gli altri che aspettano.
Al posto di sommare il vicino divido l’array in due e vado a sommare il primo elemento con il primo elemento della seconda metà, poi il secondo elemento con il secondo elemento della seconda metà etc…., ad ogni round “ridivido” e faccio questa cosa.
Qual’è il vantaggio? che i thread inattivi sono tutti raggruppati a sinistra

E questo perchè è utile? perchè non posso avviare il singolo thread in una GPU, devo schedulare un intero warp, quindi averli tutti belli raggruppati assieme mi permette di non avviare warp che effettivamente poi hanno tanti thread che non fanno nulla.
Quei warp liberi li posso usare per fare altre operazioni (anche di altri programmi).