



Architettura

▼ info

ricevimento studenti tramite appuntamento via email.

eventuali libri di testi: 'reti logiche di person editore' e 'struttura e progetto dei calcolatori- progettare con risc-v'

▼ link utili

dolly →

<https://dolly.fim.unimore.it/2020/course/view.php?id=55>

aula →

[https://eu.bbcollab.com/collab/ui/session/join/0fe0ee254b694a858077417de1688991 \(Instruction set architecture\)](https://eu.bbcollab.com/collab/ui/session/join/0fe0ee254b694a858077417de1688991)

dropbox →

<https://github.com/riccardo98c/unimore-informatica/tree/main/architettura-dei-calcolatori%2Fslide>

▼ esame

domande di teoria ed esercizi sulla teoria, scrivere il ragionamento sugli esercizi
prima parte: risposta a crocetta + ragionamento per dire che non abbiamo tirato a sorte

seconda parte: scelta multipla dove possono essere vere più di 1.

Indice Capitoli

1. [Computer: astrazioni e tecnologia](#)
2. [Prestazione dei computer](#)
3. [Rappresentazione dell'informazione](#)
4. [Introduzione alle Reti Logiche](#)
5. [Reti logiche combinatorie](#)
6. [Componenti notevoli combinatori](#)
7. [reti logiche sequenziali](#)
8. [Formati di istruzione RISC-V \(Instruction set architecture\)](#)
9. [procedure calling](#)
10. [instruction set architeture](#)
11. [Processore - datapath e controllo](#)
12. [processore - pipelining](#)
13. [processore - instruction level parallelism](#)
14. [Memoria - cache e gerarchia](#)
15. [Memoria - memoria virtuale](#)

terza parte: esercizi più completi
non ci sono le cose su logisim; può lo stesso chiede della teoria tipo sui registri...

Computer: astrazioni e tecnologia

I computer sono sistemi pervasivi, ovvero li troviamo sotto molti aspetti della vita; sono in continua evoluzione, servono sistemi sempre più piccoli e potenti.

- dispositivi personali.
- human brain project(super calcolatori).
- auto, droni.
- web, internet, cloud, IoT(internet of things).

Legge di Moore:

il numero di transistor in un circuito integrato si raddoppia ogni 2 anni.

dato un pezzetto di silicio, io posso manipolarlo per metterci dentro dei transistor; quindi a parità di area io riesco a mettere il doppio di transistor, oppure a parità di numeri di transistor riesco a dimezzare l'area(cellulari, ecc...).

⚠ integrando più transistor su un area sempre più piccola, genera un problema di calore che va dissipato.

Classi di computer

Personal computer	Server computer	Supercomputers	Embedded computer
General purpose. varietà di software. trade-off costi/prestazioni: in base alla necessità di prestazioni, varia il costo(devo farci girare dei programmi pesanti? allora aumenta il costo del pc, se no uno base costa poco).	Alta capacità di storage(tanta memoria) e performance. server è fatto per offrire un servizio e quindi più è alta l'affidabilità, più il servizio è buono e comodo).	Capaci di eseguire le simulazioni di complessi sistemi fisici e matematici(previsioni del tempo). assoluto, attraverso il calcolo in parallelo: tanti processori che si suddividono il carico	Sistemi all'interno di altri sistemi, nascono come computer posti all'interno di sistemi che non sono computer(dentro a prestazioni/performante, treni, aerei). parallelismo anche loro. sono soggetti a vincoli di potenza/performanc e costo molto stringenti:

funzionano normalmente a batteria, quindi devono consumare il meno possibile. sistemi di tipo real time: vincolo sui tempi di esecuzione

Cloud computing: evoluzione server del personal computer.

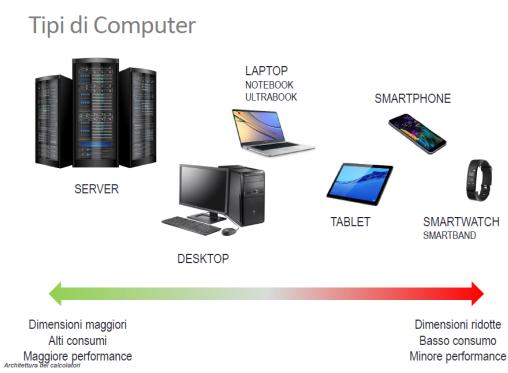
io usufruisco di un servizio che mi offre una parte della potenza dei suoi server.

Internet of things: evoluzione del cloud computing, e' l'interconnessione di tutti i device embedded tramite internet(i dispositivi smart).

tutti i sistemi che comunicano con altri sistemi tramite interconnessioni.

attualmente la tipologia di computer piu' ricercata.

▼ immagine



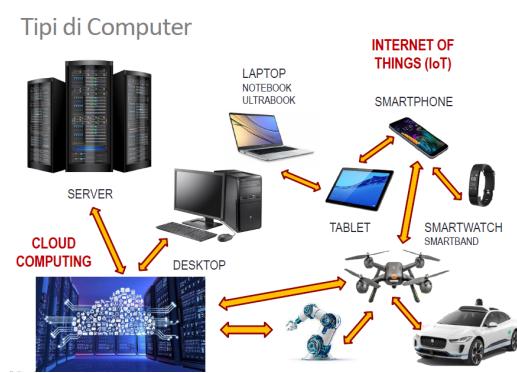
Personal mobile device: evoluzione embedded del personal computer.

bassi consumi(dispositivi a batteria). alta connettività.

basso costo(non oltre i 100 euro di solito).

→ smartphone, tablet, ...

▼ immagine



Componenti di un computer:

circa gli stessi per ogni tipo di computer:

- **Processore:**

(CPU) central processing unit. si guarda la sua frequenza per sapere la sua performance; una cpu esegue un'operazione ogni ciclo, quindi se gira a 1gigahertz, fa 1 giro al secondo.

il cuore di un computer: processa i dati in input e produce l'output.

2 blocchi:

Datapath: esegue le operazioni sui dati.

Control: il blocco logico che controlla il funzionamento del datapath, memoria, ecc...

- **Memoria:**

ospita i dati in qualunque loro stadio(originale, grezzo, qualunque formazione intermedia e finale).

organizzata come una gerarchia, basta sul tradeoff tra performance e il suo costo di realizzazione:

- **RAM**(random acces memory):

velocità più compatibili a quella del processore.

memoria volatile: quando spengo il computer, perdo tutte le informazioni precedentemente salvate su queste memorie.

- **DRAM**(dinamic RAM): è la memoria più capiente che abbiamo, economica, ma è lenta.

è la main memory ed è la memoria principale di tipo utilizzabile di un programma.

finché il processore fa calcoli semplici(addizioni, moltiplicazioni, ...) va bene, quando però bisogna entrare/leggere/scrivere dati, li ci vogliono molti cicli di cpu se si usano queste memorie.

- **SRAM**(static RAM):

- una cpu ha dentro dei **registri**(SRAM): hanno la stessa velocità del processore(i più veloci), ma sono poco capienti e più costosi.
quando devo fare delle operazioni e vado a prendere i dati, se sono dentro a questi registri, allora eseguire operazioni o prendere dati avrà la stessa velocità; se invece vado a prendere i dati in una DRAM, ci metterò molto di più(più cicli di cpu).

- **cache** memory(SRAM): a metà via tra la DRAM e i registri.

- **Hard Disk:** Memoria secondaria non volatile. molto capiente, estremamente lenta.

Magnetic disk (**HDD**) (dischino magnetico) → performance molto lenta, rapporto costo/capienza basso.

Flash memory (SD, **SDD**) (memorie flash) → performance piu' veloce, rapporto costo/capienza alto.

Optical disk (**CDROM, DVD**) → sono rom(read only memory).

di solito l'hard disk e' ibrido: una parte SSD dove ci si carica il sistema operativo(per farlo caricare piu' velocemente); e una parte HDD dove ci si mette il resto.

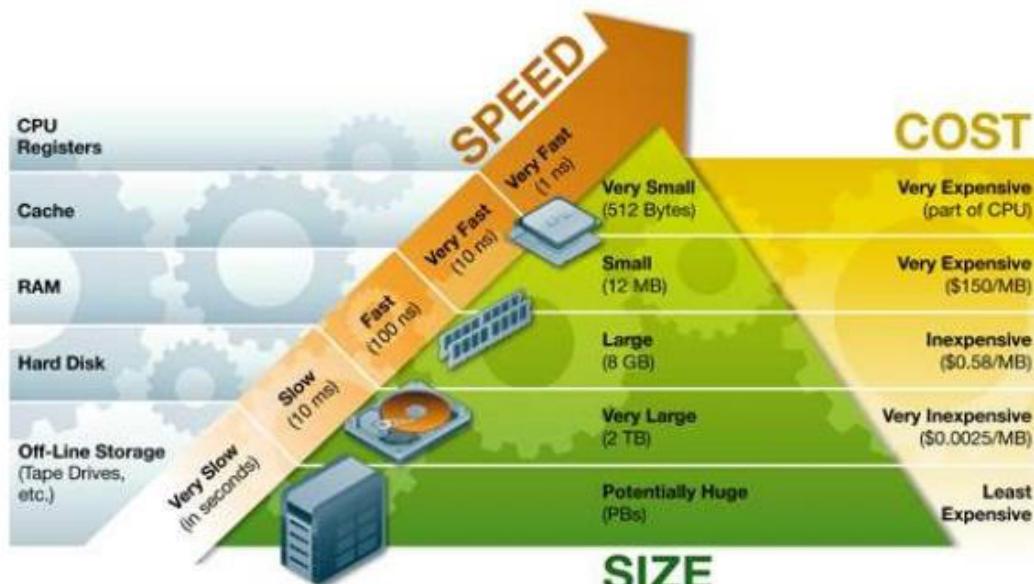
Le velocità si basano a **clock**(o cicli) delle cpu, esempio per accedere ai registri della cpu mi ci vuole un clock(un ciclo di cpu), infatti sono le memorie piu' veloci. sarà poi la cpu che in base a com'è fatta ci metterà un certo tempo a fare un ciclo; e quindi in base alla sua ampiezza e frequenza, cambiano i tempi effettivi(millisecondi, ...).

Esempio di funzionamento memoria:

ho su un hard disk il mio programma e tutti i suoi dati che è memoria non volatile, però non eseguo/carico i miei programmi da questi dispositivi, perchè sono troppo lenti; quindi dall'hard disk io carico tutte le istruzioni del programma sulla RAM che ha una velocità più compatibile con quella del processore, ma l'esecuzione avviene tramire la lettura dentro a gerarchie di memoria.

la prima volta che leggo un dato, pago tutto il costo che c'è per leggerlo dalla DRAM, dopo però l'indirizzo di questo dato(e quelli successivi) li metto dentro la cache, così la prossima volta che uso quei dati, costa molto meno(come tempo).

▼ immagine



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

- **Input/Output:**

dispositivi per gestire l'input e l'output del computer.

da memorie: Hard disk, CD/DVD, flash...

da sistemi controllati da utenti: input(tastiera, mouse, display per gli smartphone);

output(display, audio, ...)

network adapters per le comunicazione con gli altri computer: LAN, WAN, ecc...

Anatomia PC

Scheda madre = base sulla quale interconnetto i vari blocchi(le varie parti del pc);

composta da:

- ▼ immagine

immagine scheda madre generale modificata

- Main memory(DRAM) → sono memory slots dove vado a metterci la RAM
- Hard disk → connettore per l'hard disk, c'è l'interfaccia e il disco basato su memoria flash che interconnetto con la scheda madre
- **PCI slots** → ci si mettono le schede grafiche.
 - **GPU:** normalmente le gpu nascono per la grafica → rappresentare tramite triangoli e componendoli e mapping e altre cose; la gpu nasce con un sistema dedicato ad accellerare la rappresentazione di questo calcolo.
la **GPGPU**(general purpose gpu) è progettata in termini di un sistema che sa fare calcolo parallelo e general purpose → non è detto che ci devo fare solo grafica, ma posso farci anche altro, quindi esiste tipo **CUDA**(un linguaggio per le nvidia) che ci permette di programmare programmi che vanno ad utilizzare il parallelismo sulla gpu. ovvero normalmente ci sono sopra solo i processi grafici, però puoi programmarci sopra dei programmi.

- Sockets per la cpu → ospita il circuito per la cpu(il processore) e ci metto dentro il **system-on-chip**

System-on-chip → è un circuito integrato: ha più processori, ovvero ha più **core**.
(processo a 14nm significa che la dimensione del transistor è 14nm circa)

- **Ring / interconnect** per connettere i core e più in generale passaggio di dati;
L3 è una cache di livello 3, quindi dentro al ring ho gli altri 2 livelli di cache.
- **Memory interface** è l'interfaccia verso l'esterno, quindi verso DRAM o altre cose; ha anche una piccola GPU al suo interno, con un po'di core specializzati solo per il calcolo che sta dietro la grafica.

- ▼ immagine

- immagine system on chip

Anatomia di un tablet:

▼ immagine

- immagine tablet

- **display** che è l'interfaccia. utilizza la modalità **touchscreen**.
rimpiazza keyboard e mouse.
è di tipo capacitivo: consente multipli punti di "tocco" simultaneamente
- batterie; molto grandi perchè deve stare acceso il più possibili giustamente.
- il corpo principale che ospita vari dispositivi(camere, autoparlanti, sensori, ecc...).
- **scheda di calcolo** → è l'equivalente di una motherboard di un PC. è relativamente piccola rispetto al resto:
 - **controlli di I/O - circuito di alimentazione - Hard disk, che è una memoria flash - DRAM**
- **system on chip** → processo a 7nm significa che la dimensione del transistor è 7nm circa(sarebbe il gate).
 - 6 **core**, ma organizzati in un blocco con 2 cpu a 2,5GHZ e un blocco con 4 cpu a 1,5GHZ.
avere due cpu diverse è diventata la norma, perchè di base questi sistemi devono consumare poco; loro normalmente consumano le 4 cpu che hanno un GHZ minore, quindi consumano di meno e in generale il SO cerca di usare solo quello; quando però lo richiede, si usano le altre due(per il minor tempo possibile)
 - **GPU**
 - **NPU** → neural processing unit(processori dedicati per il calcolo di tipo neurale: riconoscimenti facciali, ecc...).
 - **system cache** che è sempre una **L3**
 - **DDR logic** sono i punti di accesso alla DRAM.

Astrazione calcolatori

divisione tra parte hardware e parte software di un computer:

▼ immagine

- immagine astrazione calcolatori

1. livello più basso: progettazione su silicio con tecnologia **CMOS**
[\(https://it.wikipedia.org/wiki/CMOS\)](https://it.wikipedia.org/wiki/CMOS)
2. sopra: primo livello di astrazione tale da poter modellare il comportamento dei circuiti senza doverci preoccupare della propagazione dei segnali elettrici:
i circuiti logici, sono modelli di come si deve comportare la macchina lavorando con progettazione a blocchi.
3. sopra: **CPU(RISC-V)** macro blocco composto da blocchi modellati utilizzando i circuiti logici.
4. sopra: **ISA interfaccia software**, definisce l'interfaccia di programmazione per quella macchina.
 - tra questi 2 livelli c'è il **compilatore**(e il SO perchè serve a dare una vista astratta delle risorse hardware sottostanti) che serve a tradurre le istruzioni da linguaggio ad alto livello, a linguaggio macchina.
5. livello più alto: **software**; la parte degli utilizzatori del computer. SO sopra al quale i programmi che sono creati al partire dalla scrittura di un programma, attraverso il linguaggio di programmazione.
applicazioni scritte ad alto livello.
software di sistema: **SO**, gestisce I/O da periferiche vari, gestisce memoria, schedula l'esecuzione dei vari task sulle cpu, ...
ipervisor = componenti software che permettono le virtual machine: astraggono l'hardware rendendo possibile avere più SO.

anche a livello di singolo programma cambia molto; linguaggi:

- **linguaggi ad alto livello**: mettono a disposizione del programmatore un'interfaccia più vicina al concepire il programma come linguaggio umano.
- **linguaggio binario**: file che facciamo leggere al nostro computer; la macchina interpreta sequenze di 0 e 1 come una serie di istruzioni; chi definisce come vengono interpretate? **ISA**
i primi programmi venivano programmati in binario e si faceva tramite le **schede forate**: presenza o assenza di foro veniva interpretato come presenza o assenza di 0 o 1; che significa presenza o assenza di segnale elettrico che avviene tra i transistor.
- **linguaggio assembly** → linguaggio simil macchina, ma ancora comprendibile dagli esseri umani. hanno un livello di astrazione molto basso, quindi lunghe istruzioni da scrivere e conoscenza approfondita dei registri.
si programma in **assembly** solo in alcuni casi particolari es: nei sistemi embedded quando voglio programmare una funzionalità del sistema avendo completo controllo

di come la macchina eseguirà quel pezzo di codice.
quando lancio il compilatore del mio programma scritto in alto livello, posso anche configurarlo, perchè non sono solo dei traduttori, ma ottimizzano anche il codice mentre traducono in assembly, per questo a volte si scrive direttamente in assembly.

l'assembler produce binario a partire dall'assembly.

ISA (instruction set architecture) → è un modello astratto di un computer; dici a chi si interfaccia col computer quali sono le istruzioni che questo computer comprende.
(instruction set = set di istruzioni)

quindi definisce i tipi di dato su cui la macchina lavora(interi, short, caratteri, aritmetica a virgola mobile, ...), quanti sono e come lavorano i registri(fondamentale saperlo per chi scrive in assembly perchè sono pochi e sono gli unici che vanno alla stessa velocità del processore), definisce il supporto hardware per lavorare con la DRAM(load e store → spostamento di un dato dalla DRAM e verso la DRAM con salvataggio in memorie più alte) e con I/O.

realizzazione(o implementazione) di una ISA: **CPU**.

essa realizza quella interfaccia e quella funzionalità; quando uso una CPU devo decidere quale ISA userà → noi utilizzeremo quella del RISC-V; adesso per realizzarla fisicamente devo progettare.

quando la realizzo devo decidere come interpretare le sequenze di bit; es architettura a 32 bit, prendo una sequenza di bit e la suddivido in 32 bit alla volta.

▼ immagine

la CPU è una lastra di silicio su cui ci si "stampa" dei transistor interconnessi tra di loro → questa è una "**fotografia**" di come sono disposti i transistor; anche i livelli di metallo che sono quelli che eseguono le connessioni; anche le porte logiche OR, AND, ...

immagine

Le **SoC**(system on chip) avevano tanti blocchetti di **CPU cores**, ognuno di essi contiene:

- **datacache**: cache di primo livello.
- **I cache**: cache di primo livello per le istruzioni.
- **L2 cache**: cache di secondo livello.

floating point: tipo di dato che supporta questa CPU, ovvero l'aritmetica a virgola mobile.

load/store sono le istruzioni che permettono di spostare e prendere dati sulla/dalla

DRAM; è importante perchè la DRAM è fuori dalla CPU, quindi i dati passano per quel blocco, poi la memory interface e poi raggiungono la DRAM.

Ogni CPU può essere configurata in modo diverso, ad esempio avere la L2 e L3 dentro o fuori.

La ISA definisce il comportamento di load/store; tramite la pipeline. Definizione di ISA: "Un instruction set, o Instruction Set Architecture (ISA), (in lingua italiana insieme d'istruzioni), in informatica ed elettronica, descrive quegli aspetti dell'architettura di un calcolatore che sono visibili a basso livello al programmatore. L'espressione è a volte usata anche per distinguere l'insieme suddetto di caratteristiche dalla microarchitettura, che è l'insieme di tecniche di progettazione utilizzate per implementare l'instruction set (tra cui microcodice, pipeline, sistemi di cache e così via)." citazione da

Instruction set - Wikipedia

Un instruction set, o Instruction Set Architecture (ISA), (in lingua italiana insieme d'istruzioni), in informatica ed elettronica, descrive quegli aspetti dell' architettura di un calcolatore che sono visibili a basso livello al programmatore.

W https://it.wikipedia.org/wiki/Instruction_set

- **CPU pipeline** = catena di montaggio → serie di operazione/fasi che sono fatte una dopo l'altra e si chiamano **fasi di decode**, perchè prende un'istruzione dalla memoria e la decodifica(prima dicevamo che la ISA prende i 32bit alla volta e li interpreta), questo avviene tramite il blocco **DECODE** dentro alla CPU. dentro al blocco decode stesso, abbiamo dei registri, tutto il resto è la logica che legge i bit, li interpreta e capisce dove/come scriverli e dove prendere i dati. la CPU pipeline è costruita da blocchi progettati come **reti logiche**.
 - Le **reti logiche** fungono da astrazione della tecnologia sottostante(transistor, ...) e sono reti composte: interconnettendo tra di loro le **porte logiche**(mattoncini elementari), che realizzano funzioni booleane(OR, AND, ...). (il funzionamento di un **circuito digitale** è rappresentato/modellabile da funzioni booleane)
 - Una **porta logica** è un dispositivo che realizza una **funzione booleana**(OR, AND, più generalmente un'operazione logica su uno o più input binari, che produce un output binario singolo). esse sono implementate tramite transistor; il transistor funge da **interruttore elettronico**: passa corrente, non passa corrente → stato logico vero, stato logico falso oppure 1, 0.

La maggior parte dei chip utilizza la **logica CMOS**, ma in realtà posso realizzarli come voglio; l'importante è che mi consentano di realizzare questa **rappresentazione con 2 stati 1 e 0**.

Circuito integrato: circuito di elettronni è creato prendendo un pezzo di semiconduttore(silicio) e stampando su questo pezzo i vari transistor. il silicio è usato perchè è un semiconduttore, quindi ha una condutività elettrica a metà tra un conduttore e un isolante, quindi si può alterare la carica elettrica in modo da implementare questa sottospecie di switch.

Come vengono creati questi circuiti integrati(SoC)?

si parte da un lingotto di silicio puro e stampo su una fetta di silicio tanti circuiti integrati(ovvero tanti processori i7):

1. Si parte affettando con una lama di diamante, tanti **silicon wafer**(fette di silicio).
2. Poi si ripulisce la superficie del wafer silicio, mantenendo le proprietà fisiche di cui ha bisogno
3. Poi si applica uno strato di ossido sulla superficie il **photoresist**
4. Poi una volta esposto alla luce ultravioletta, diventa un materiale che posso trattare con degli acidi per creare delle piste sulla superficie del silicio.
si utilizza una maschera che ci dice dove andare a creare queste piste che saranno le piste di **interconnessione**.
5. Il photore → esposizione a carica ionica per qualcosa. → tutto questo viene effettuato varie volte con varie maschere, così facendo alla fine avrò il mio wafer pieno di circuiti integrati per le interconnessioni. (*maxbubblegum: "anche io non ho trovato granché su google a riguardo del photore, mi trova solo robe di photoshop e photored"*)
6. Una volta che ottengo il mio wafer stampato, devo fare una verifica per provare se funzionano i circuiti stampati, quelli che non funzionano vengono eliminati.
poi dopo la fase di packeging si ritestano.

indice

Prestazioni/performance di un computer

Response time o latenza o execution time: quanto tempo ci vuole per eseguire un programma/task.

throughput: quantità di lavoro fatta per unità di tempo, quanti compiti(task, ...) riesco a fare in un'ora , ...

Cosa determina la performance di un programma?

- **Algoritmo:** numero di operazioni che esegue il costo di ogni operazione.
- **Processore e memory system:** quanto saranno veloci le operazioni(task). quanto impiega una CPU a eseguire un'operazione: **response time**
- **I/O system(anche l'OS):** quanto sono veloci le operazioni di I/O
- **ISA e compiler:** prendiamo l'algoritmo lo trasformiamo in linguaggio di programmazione, viene poi compilato in linguaggio macchina che poi viene dato all'architettura da eseguire:
l'algoritmo determina il numero di operazioni eseguite, ma l' ISA determina il numero di **istruzioni macchina** eseguite.
come progetto l'architettura definisce la velocità con cui eseguo le operazioni.
per comparare due computer, comparo le loro performance e quindi posso usare anche il tempo di esecuzione.
Il tempo di esecuzione si articola in più di una variante, perchè io lo eseguo su un programma complesso → è opportuno discriminare tra
 - Elapsed time → tempo totale: tempo CPU in cui non fa niente e in cui fa lavoro,I/O, ... intero sistema

- Define **Performance = 1/Execution Time**
- “X is n time faster than Y”

$$\text{Performance}_X / \text{Performance}_Y = \frac{1}{\text{Execution time}_Y} / \frac{1}{\text{Execution time}_X} = n$$

Execution Time viene spesso misurato in cicli di clock ed è composto da:

- **Elapsed Time:** Tempo totale utilizzato per eseguire un programma, include tutti gli aspetti: il processo, I/O, overhead del OS e tempo di attesa(**idle time**).
- **CPU Time:** tempo speso per l'esecuzione di un particolare task/job; quindi ignora le parti di I/O o quelle in cui la CPU fa altri task non inerenti, ma guarda solo quelle del mio programma e quelle del SO che servono al funzionamento del mio programma.

- **User CPU time** = solo il tempo utilizzato dalla CPU per eseguire il mio programma.
- **System CPU time** = tempo utilizzato dalla CPU per altri programmi che sono necessari per eseguire il mio programma/task.

CPU clocking(il clock della CPU): la maggior parte dei sistemi digitali sono *sistemi sincroni* che operano tramite un **clock**, un orologio del sistema che scandisce come le operazioni avvengono; bisogna sapere quanto vale il **periodo di clock**(clock period), per sapere la frequenza(**clock frequency**). Il segnale di clock è un segnale regolare che è un'alternanza di 1 e 0; e scandiscono questo andamento regolare. periodo di clock è un finestra di tempo in cui ci completa il mio segnale, la frequenza è data dalla quantità di volte in cui questa finestra avviene in un tempo(di solito in un secondo).

Il **periodo di clock** viene definito in secondi o sottomultipli; la **clockrate** o **clockfrequency** viene definita in **cicli al secondo** → 4GHz significa a 4 miliardi di cicli al secondo.

Il CPU Time può essere misurato come il numero di cicli di clock che questo programma ha impiegato, moltiplicato per al durata del singolo ciclo di clock(che si può definire come l'inverso della frequenza di clock):

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

La performance può essere migliorata:

- Riducendo i cicli di clock per eseguire quel programma, tipo agire sugli algoritmi e sui programmi.
- Incrementando la frequenza(**clock rate**).

Un progettista hardware, spesso deve considerare il compromesso tra numero di cicli di clock e clock rate; perchè quando cerco di migliorare la frequenza del processore, spesso l'unico modo è rendere la pipeline più lunga e questo porterebbe ad avere più cicli per istruzione.

Il **pipelining** aiuta a rendere più veloce la frequenza delle CPU, però ad un certo punto per farlo devo spezzare la pipeline in più stadi; vuol dire che la mia istruzione viene spezzettata in più stadi → globalmente la mia CPU è migliore, ma in realtà ho bisogno di più cicli/stadi della pipeline per il mio lavoro; quindi diventa un compromesso tra avere una frequenza più alta, ma avere più cicli per completare un'istruzione.

▼ esempio computer più veloce

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
 - Aim for 6s CPU time
 - Can do faster clock, but causes $1.2 \times$ clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Instruction count: il numero di istruzioni che richiede l'algoritmo/programma.

Il numero di istruzioni per un programma è determinato:

- Dall'algoritmo e dal programma perchè quelle operazioni vanno tradotte in istruzioni dal linguaggio.
- Dalla ISA per tradurre le istruzioni del linguaggio.
- Dal compilatore perchè non fa una banale traduzione 1 a 1, ma può ottimizzare/cambiare il numero di istruzioni eseguite.

CPI(clocks per instruction): significa il numero di cicli di clock che quella CPU impiega per eseguire un'istruzione. Però non tutte le istruzioni sono uguali, infatti una CPU impiega tempi diversi in base alla *classe dell'istruzione*.

La performance di una CPU si misura come **CPI medio**. Il CPI è determinato dall'hardware della CPU(come la progetto); dipendendo dalla classe dell'istruzione, il CPI medio a suo volta è influenzato dal **mix di istruzioni**(la percentuale del tipo di istruzione sulle istruzioni totali del task) che esegue la macchina.

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

ISA è lo stesso, quindi uso esattamente le stesse istruzioni per eseguire il programma su computer A e B. quindi l'**instruction count** è lo stesso per entrambi; quindi posso semplificarlo nel rapporto

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \quad \text{A is faster...} \\ \text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps} \\ \frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2 \quad \dots \text{by this much}\end{aligned}$$

I cicli di clock possono essere definiti come il prodotto tra il numero di istruzioni di un programma per il **CPI medio**; che si ricava a partire dalla definizione dei singoli CPI * il numero di istruzione del loro tipo di istruzione. io ho il numero di clock cicles che servono per eseguire un certo programma che è composto da un certo instruction count è dato dalla sommatoria tra i vari CPI medi per le varie classi di istruzioni con il numero di istruzioni medio per quelle determinate classi di istruzioni. Se il mio programma ha metà istruzioni matematiche con CPI = 1 e metà aritmetiche con CPI = 10 e il mio instruction count generale è 100 :

diventano `instruction count matematico * CPI matematico = 50`; stessa cosa per aritmetico, quindi = 500

dunque i clock cycles totali eseguiti per quel programma è uguale alla somma dei 2 → 550

dalla formula per clock cycles → il CPI medio per quel programma = 550/100 = 5.50 =

clock cycles / instruction counts

questo perchè faccio instruction count della mia classe/instruction count generale.

$$\text{Clock Cycles} = \sum_{i=1}^n (\text{CPI}_i \times \text{Instruction Count}_i)$$

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^n \left(\underbrace{\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

▼ CPI esempio:

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

■ Sequence 1: IC = 5

- Clock Cycles
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$
 $= 10$

- Avg. CPI = $10/5 = 2.0$

■ Sequence 2: IC = 6

- Clock Cycles
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$
 $= 9$

- Avg. CPI = $9/6 = 1.5$

qui i computer hanno ISA diverso, quindi le istruzioni di un programma sono di un numero diverso in base all'ISA.

i clock cycles li calcolo come al solito con numero istruzione di quella classe * il CPI di quella classe

→ in tutto ci vogliono 10 cicli per eseguire la sequenza 1 → quindi il CPI medio è $10/5 = 2.0$

stessa cosa per la sequenza 2; quindi avrò cicli di clock = 9 → cpi medio = $9/6 = 1.5$



se in un esercizio ci danno un instruction count generico, allora usiamo la formula del CPI generale(quella a sinistra).

se invece ci danno un instruction mix, si usa la formula a destra(quella con la frequenza di occorrenza di quel tipo di istruzione nel instruction mix).

CPU time tenendo fuori altri contributi di sistema come i/o; si può calcolare come:
instruction count * CPI medio(o il numero di istruzioni per ogni classe di istruzione * CPI di quella classe) * frequenza clock.

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

In base alla ISA cambia il CPI medio.

Quindi la frequenza dipende da:

- Algoritmo:
 - impatto sull'Instruction count → determina il numero di istruzioni elementari da eseguire
 - può avere impatto anche sul CPI, perchè in base all'algoritmo, cambia l'Instruction mix(quindi la percentuale del tipo delle istruzioni).
- Programming language:
 - effetto sul numero di istruzioni e CPI; quante operazioni aritmetiche il mio programma fa per ogni accesso alla memoria.
- Compilatore:
 - per effetto della ISA può cambiare il numero di istruzioni eseguite e anche il tipo. → instruction count e CPI.
- ISA:
 - instruction count e CPI
 - inoltre ha effetto sul tempo di clock, perchè stabilisce numero di stadi della pipeline e quindi ha impatto sulla durata del clock.

Se io non avessi una pipeline, dovrebbe realizzare tutta la CPU come un unico blocco logico che esegue in un unico ciclo di clock; ma questo blocco ha un segnale elettrico che gli passa in mezzo → più il mio blocco è complesso e grande, più il segnale elettrico ci metterà. Una pipeline ha più parti, ognuna di queste parti ha una funzionalità. Ci metterò un certo tempo per percorrerla tutta, che è correlato col tempo di propagazione del mio segnale elettrico → se io non spezzo l'esecuzione, il mio ciclo di clock è determinato da questo tempo.

Se io utilizzo un design pipeline cosa faccio? → spezzo a metà la pipeline e ci metto un registro, serve per salvare lo stato del segnale → io faccio la prima metà, salvo e poi faccio la seconda metà → quindi adesso la frequenza è il doppio; perchè faccio il doppio di task nel tempo di uno.

pensa a risorse → ogni passaggio della pipeline è una risorsa; invece di tenere

occupata ogni risorsa/stadio per una sola operazione finché non finisce; → appena la prima operazione finisce il primo stadio/risorsa, allora sblocca subito la risorsa/stadio per la prossima operazione. → così facendo incremento il **throughput e quindi la frequenza**.

con pipeline io eseguo più istruzioni rispetto a non pipeline → quindi il primo ciclo esegue metà della prima istruzione; al secondo ciclo, la prima istruzione finisce di eseguire e nel mentre prendo una nuova istruzione e la eseguo fino a metà e così via → quindi "**raddoppio il throughput**" oppure raddoppio la frequenza.

Il problema dell'uso intensivo del pipelining sono:

- i limiti fisici → consumo di potenza e soprattutto generazione di calore che non si riesce più a dissipare.
- limiti su istruzioni → quanto massimo parallelismo posso sfruttare dalla mie istruzioni; una pipeline funziona bene se non ci sono dipendenze tra le istruzioni.
- un programma non avrà mai indipendenza completa tra le sue istruzioni; semplicemente se faccio un'operazione su una variabile e poi ci calcolo altro dal suo risultato, ho creato una dipendenza tra due operazioni. Quindi dopo un po' non si cerca più il parallelismo dentro al singolo programma, ma si cerca tra i vari programmi tra di loro; per questo ci sono più core.

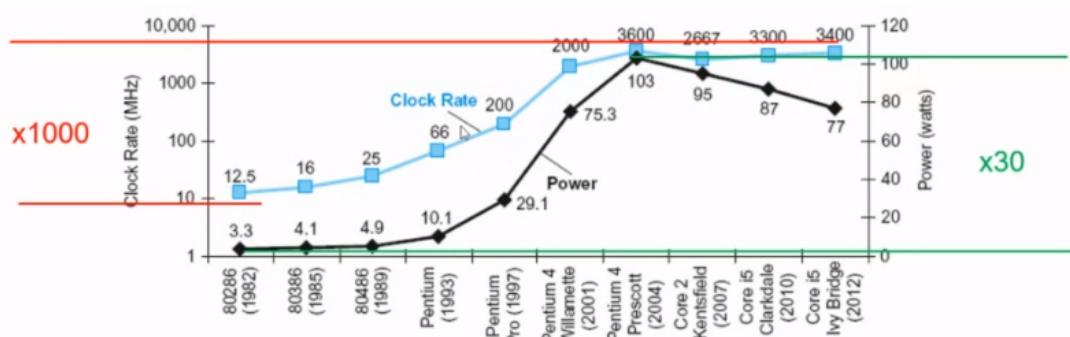
▼ CPU al giorno d'oggi e parallelismo(**legge Amdahl**)

le CPU ultimamente vengono progettate come sistemi multicore; la tecnologia CMOS è governata da una formula:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

$\times 30$ $5V \rightarrow 1V$ $\times 1000$

▼ immagine power trends



CPU sempre più veloci sono CPU che consumano sempre di più.

La variazione nella frequenza nella velocità dei processori è cresciuta di un fattore

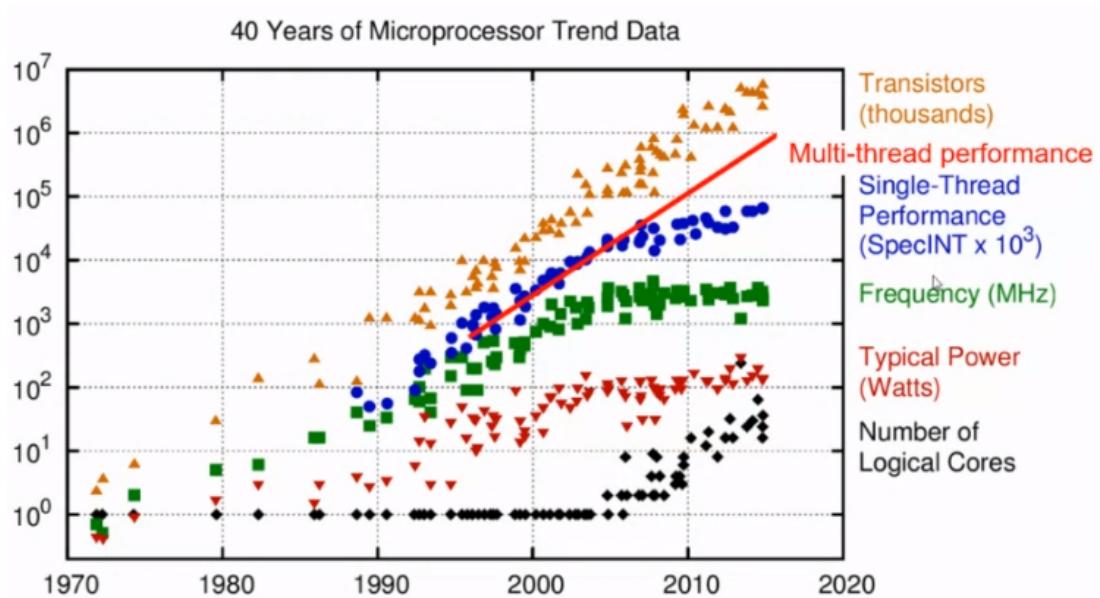
migliaia; la power consumption (il TDP praticamente) è cresciuto ancora più velocemente(fattore 30); il voltaggio è cresciuto di 5v. → (guardare formula sopra). Intorno al 2004 il power era salito troppo; da qui la frequenza dei processori si livella, mentre il consumo di potenza cala. *Maxbubblegum*: "pensa sirot che questa cosa cambiò moltissimo lo sviluppo dei processori per Intel: loro avevano pianificato di incrementare sempre di più la frequenza e tenere i core bassi. Pensavamo che l'aumento della frequenza fosse il trend da seguire. Gli ultimi processori davvero potenti di Intel in relazione al periodo storico in cui sono usciti sono quelli con architettura Maxwell. In quel periodo erano riusciti a fare dei quad core con frequenze di tutto rispetto ancora per i nostri giorni e anche lato portatile la serie di processori m e mq, ancora oggi è una pietra miliare. Successivamente a quella generazione di processori l'azienda si concentrò molto di più sull'efficienza delle cpu e, soprattutto sui portatili, abbiamo avuto, fino all'8th gen della serie U, dei processori davvero molto più deboli, al netto ovviamente di consumi inferiori, di quelli partoriti dalla quarta generazione (con l'architettura sopra citata). Un processore che ancora è leggendario, soprattutto nel panorama dei portatili Thinkpad è l'i7 4810mq; tale è la sua fama che ancora oggi la si usa per confrontare una cpu laptop nuova. Lato amd invece sin dalla generazione degli FX avevano capito che la cosa importante era aumentare il numero di core, ma la loro prima implementazione (FX-Series) non fu particolarmente brillante. A seguire investirono praticamente tutto sullo sviluppo dei Ryzen e questo ha ripagato moltissimo. Ancora a distanza di svariati anni il Ryzen 5 3600 è un processore davvero molto più capace di molti i5 e i7 di 11th e 10th gen (soprattutto per operazioni di compressione e decompressione di dati). Attualmente, anche in ambito lavorativo e per i server, i Ryzen di AMD sono la scelta migliore e lo saranno probabilmente fino al passaggio ad architettura ARM (siamo alla x86_64). Altre aziende come Apple sta già migrando verso tale piattaforma, e nel primo trimestre del 2021 ha triplicato le vendite rispetto alle vendite dell'anno precedente. L'architettura ARM, non solo è più potente, ma estremamente più efficiente. L'unico limite al momento è dato dalla compatibilità delle applicazioni scritte per x86_64 che devono essere tradotte da Rosetta 2 per poter essere utilizzate correttamente."

Questo è chiamato il **muro di power**: si creano problemi di **densità di potenza**(troppi transistor), ovvero si crea troppo calore che non si può più dissipare. L'attività dei transistor su chip, crea tante problematiche: calore, consumo di energia e problemi di performance principalmente. Quindi si crea questo limite dato da più fattori: il power(visto prima), il parallelismo a livello di istruzione(non si può creare un programma senza dipendenze) ed infine la **latenza di memoria**; i processori sono sempre più veloci, mentre la memoria principale(DRAM) migliora, ma molto

più lentamente e quindi aumenta sempre più il divario tra un'operazione sui registri della CPU ed una sulla DRAM.

La soluzione a tutto ciò sono i **multiprocessori**: invece di avere una CPU sempre più potente, si usano più processori su un chip. Sono più semplici, consumano meno e vanno più lente → ma lo stesso lavoro che potevi fare solo su un CPU, lo dividi in più CPU.

▼ grafico multicore



Numero di core: dal 2004 si sono iniziati a mettere più core; ad oggi non ci sono solo quad core octa core, ... , quelli che si sentono nelle pubblicità. La CPU di un computer normale, non richiede centinaia di processori, perchè i compiti di questo computer non sono difficili; mentre in altri sistemi, ci sono dei compiti che richiedono molti core e quindi le CPU hanno centinaia di core. Ci sono state CPU con centinaia di core usate per i PC, ma si è capito che non era la maniera giusta di progettare quel processore; quindi la CPU è progettata avendo 4/6/8 core ed è affiancata da un processore dedicato al parallelismo che ha centinaia di core(**GPU**).

Consumi di potenza(semplificati): dal 2004 siamo riusciti a far calare molto l'aumento di consumi di potenza.

Frequenza stabile.

Transistor: sempre di più.

Performance: ma di un singolo processo; singolo thread. col multiprocesso dovrebbe aumentare.

Col multicore la programmazione diventa parallela.

Speedup: è il metodo per sapere di quanto una CPU di nuova generazione è migliore della vecchia.

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}}$$

Execution time of the "old" system

Execution time of the "new" (improved) system

Quando si parla di parallelismo viene un problema:

Legge di Amdahl:

Il migliorare un particolare aspetto di un sistema di calcolo, ci rende possibile migliorare la performance complessiva, soltanto per il tempo che è interessato nella miglioria. Il tempo complessivo migliorato è dato da tempo migliorato(che è interessato dalla miglioria)/il fattore di miglioria + tempo non affetto

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

▼ esempio legge

Example: multiply accounts for 80s/100s

- How much improvement in multiply performance to get Speedup=5?

$$T_{\text{old}} = 100 = T_{\text{affected}} + T_{\text{unaffected}} = 80 + 20$$

$$T_{\text{new}} = T_{\text{improved}} = \frac{80}{n} + 20$$

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = 5 \rightarrow \frac{T_{\text{old}}}{\frac{80}{n} + 20} = 5 \rightarrow \frac{100}{\frac{80}{n} + 20} = 5 \rightarrow \frac{100}{5} = \frac{80}{n} + 20 \rightarrow 20 = \frac{80}{n} + 20 \rightarrow 0 = \frac{80}{n}$$

sappiamo che di 100 secondi, 80 sono spesi a fare moltiplicazioni, mentre 20 altre operazioni; voglio migliorare lo speedup di 5 → voglio trovare n tale per cui lo speedup sia = 5

questa equazione non può essere verificata; non posso ottenere uno speedup generale pari a 5.

non è che solo perché ho 10 processori, avrò uno speedup pari a 10; perché tutti i miei programmi hanno una porzione di **tempo affected**(può essere parallelizzata), ma ha anche una porzione non **affected** e quindi non può essere cambiata dall'avere più processori.

IPC è l'inverso del CPI: numero di istruzioni diviso numero di cicli di clock → ci dice in media quante istruzioni un processore riesce a ritirare per ciclo.

$$\text{IPC} = \frac{\text{Instruction Count}}{\text{Clock Cycles}} = \frac{1}{\text{CPI}}$$

MIPS : Millions of Instructions Per Second

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \\ &= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}\end{aligned}$$

ha dei limiti questa metrica: i MIPS non tengono conto delle differenze tra diversi ISA nei computer.

esercizi capitolo 2

▼ Prima prova parziale 29 aprile 2020

▼ quesito

1. (2, -.5) Si consideri un certo programma con 70% di istruzioni di tipo aritmetico, 10% load/store e 20% branch. Si assuma che le istruzioni aritmetiche eseguano in due cicli, le load/store in sei e i branch in tre. Qual è il CPI medio?
 - a) 1,2
 - b) 2,6
 - c) 3,7
 - d) Nessuna delle precedenti

▼ soluzione

Coi dati forniti si ottiene

$$\begin{aligned} CPI_{medio} &= \frac{70 \text{ istr}}{100 \text{ istr}} * 2 \frac{\text{cicli}}{\text{istr}} + \frac{10 \text{ istr}}{100 \text{ istr}} * 6 \frac{\text{cicli}}{\text{istr}} + \frac{20 \text{ istr}}{100 \text{ istr}} * 3 \frac{\text{cicli}}{\text{istr}} \\ &= \frac{140 + 60 + 60 \text{ cicli}}{100 \text{ istr}} = 2,6 \frac{\text{cicli}}{\text{istr}} \end{aligned}$$

load/store di solito pesano molto, infatti CPI = 6

branch = istruzioni di salto condizionale → if, ciclo for, cicli while.

avendo una pipeline, io aggiungo ad ogni ciclo, un'istruzione; il problema avviene quando ho un if → io ho il risultato dell'if solo quando l'istruzione dell'if ha attraversato tutta la pipeline; quindi in questi casi si crea della logica di speculazione → io ipotizzo che succederà una cosa e la predizione può andar bene o male; se è andata bene ottimo, se no devo svuotare la pipeline e ricaricare una nuova istruzione.

per questo in media i branch pesano più di un'istruzione → se va bene pesano 1, se no di più.

si guarda sempre la MEDIA.

▼ Prima prova parziale 26 giugno 2020

▼ quesito

1. (3, -, 5) Qual è il CPI medio di un programma con 80% di istruzioni di tipo aritmetico, 10% load/store e 10% branch, per cui le istruzioni aritmetiche eseguono in un ciclo, i branch in tre, mentre le load/store hanno un tempo di accesso medio di K cicli? K va calcolato sapendo che:
 - di un totale di 110 load/store effettuate dal programma 80 sono hit nella L1 cache, 20 sono hit in L2 (miss in L1) e 10 sono accessi in memoria principale (miss in L1 e in L2);
 - il tempo di hit in L1 è 1,5 cicli, il tempo di hit in L2 è 5 cicli (include il tempo di miss in L1) e il tempo di accesso in DRAM è 55 cicli (include il tempo di miss in L1 e in L2)
 - a) 1,2
 - b) 1,8
 - c) 3,7
 - d) Nessuna delle precedenti

▼ soluzione

Coi dati forniti si ottiene

$$K = \frac{80 \text{ ld/stL1} * 1,5 \frac{\text{cicli}}{\text{ld/stL1}} + 20 \text{ ld/stL2} * 5 \frac{\text{cicli}}{\text{ld/stL2}} + 10 \text{ ld/stDRAM} * 55 \frac{\text{cicli}}{\text{ld/stDRAM}}}{110 \text{ ld/st}}$$

$$= \frac{120 + 100 + 550 \text{ cicli}}{110 \text{ ld/st}} = \frac{770}{110} = 7$$

da cui

$$CPI = \frac{80 * 1 + 10 * 7 + 10 * 3}{100} = \frac{80 + 70 + 30}{100} = \frac{180}{100} = 1,8$$

il programma ha un totale di 110 load/store = 110 operazioni che accedono alla memoria → le load store sono dirette in DRAM; se noi non avessimo una gerarchia di memoria, allora le operazione avrebbero lo stesso costo; però noi le abbiamo e ci dice che 80 di queste sono hit nel primo livello(cache L1), 20 sono hit nel secondo livello(miss L1 e cache L2) e 10 sono accessi in memoria principale DRAM(miss L1 e miss L2).

come funziona una gerarchia di cache? se sono fortunato trovo il dato(**hit**) nel livello più vicino(L1), se non c'è, faccio **miss** e provo nel secondo livello e poi terzo.

il tempo di hit in L1 è 1,5 cicli; in L2 è 5 cicli(include tempo di miss) e in DRAM è 50 cicli...

per risolverlo calcolo il numero di cicli delle load/store che sono la percentuale degli accessi sugli accessi totale * il costo(in cicli) degli accessi e poi sommo i 3 risultati(ho 3 accessi diversi L1 L2 e DRAM) e dopo ho il mio numero di cicli per load store.

▼ Prima prova parziale 7 luglio 2020

▼ quesito

1. (3, -5) Si calcoli il tempo di accesso medio A delle load/store di una CPU sapendo che:
 - di un totale di 70 load/store effettuate da un programma 55 sono hit nella L1 cache, 7 sono hit in L2 (miss in L1) e 8 sono accessi in memoria principale (miss in L1 e in L2);
 - il tempo di hit in L1 è 1 ciclo, il tempo di hit in L2 è 5 cicli (include il tempo di miss in L1) e il tempo di accesso in DRAM è 50 cicli (include il tempo di miss in L1 e in L2)

Qual è l'**IPC** medio di un programma con 70% di istruzioni di tipo aritmetico, 20% load/store e 10% **branch**, considerando che le istruzioni aritmetiche eseguono in un ciclo, i **branch** in 4 cicli, le load/store in A cicli.

- a) 0,4
- b) 0,5
- c) 0,7
- d) Nessuna delle precedenti

▼ soluzione

Coi dati forniti si ottiene

$$K = \frac{(55 * 1 + 7 * 5 + 8 * 50) \text{ cicli}}{70 \text{ ld/st}} = \frac{55 + 35 + 400}{70} = \frac{490}{70} = 7$$

da cui

$$IPC = \frac{100}{70 * 1 + 20 * 7 + 10 * 4} = \frac{100}{70 + 140 + 40} = \frac{100}{250} = 0,4$$

sempre stesso metodo per trovare i cicli. però noi cerchiamo l'IPC medio che è l'inverso di CPI → quindi invece di fare i vari CPI * istruzioni / 100 → faccio 100/CPI*istruzioni

▼ Prima prova parziale 24 luglio 2020

▼ quesito

1. (2, -5) Qual è il CPI medio di un programma con 80% di istruzioni di tipo aritmetico, 20% load/store e 10% branch, considerando che le istruzioni aritmetiche eseguono in un ciclo, i branch in 4 cicli, le load/store in A cicli. Si può determinare sapendo che il programma ha un miss rate del 5%, che il costo di una hit è di 2 cicli e il costo medio di una miss 82 cicli.
 - a) 2,4
 - b) 1,5
 - c) 0,7
 - d) Nessuna delle precedenti

▼ soluzione

Coi dati forniti si ottiene

$$CPI = \frac{80 * 1 + 20 \left(\frac{5 * 82 + 95 * 2}{100} \right) + 10 * 4}{100} = \frac{80 + 120 + 40}{100} = \frac{240}{100} = 2,4$$

▼ Prima prova parziale 19 febbraio 2021

▼ quesito

2. (3, -.5) Si vuole eseguire un programma con 40 istruzioni di tipo **aritmetico**, 45 **load/store** e 10 **branch** su un processore con una frequenza di 2 GHz. Considerando che il processore ha un CPI per le istruzioni **aritmetiche** e i **branch** pari a 1 e un CPI per le **load/store** pari a 10 si dica quanto impiega il programma a eseguire, sia in cicli di clock che in secondi.
- 500 cicli = 0,004 s = 4 ms
 - $1/500 \text{ cicli} = 2*10^9/500 \text{ s}$
 - 500 cicli = 250 ns
 - Nessuna delle precedenti

▼ soluzione

Coi dati forniti si ottiene

$$\begin{aligned} DURATA_{CICLI} &= 40 ISTR_{ARITM} * 1 \frac{CICLI}{ISTR_{ARITM}} + 45 ISTR_{LD/ST} * 10 \frac{CICLI}{ISTR_{LD/ST}} + 10 ISTR_{BRANCH} * 1 \frac{CICLI}{ISTR_{BRANCH}} \\ &= 500 CICLI \\ DURATA_{SECONDI} &= \frac{CICLI}{FREQ} = \frac{500 CICLI}{2 * 10^9 \frac{CICLI}{SEC}} = 250 * 10^{-9} SECONDI = 250 ns \end{aligned}$$

durata cicli

durata totale(CPU time)

▼ Prima prova parziale 29 aprile 2020

▼ quesito

1. (2, -.5) Si consideri il seguente programma C.

```
int main ()
{
    work ();      // sequenziale, 1000 cicli
    for (int i=0; i<20; i++)
        work (); // 1000 cicli
}
```

Sapendo che la funzione `work ()` è composta da una sequenza di 1000 istruzioni, che complessivamente impiegano 1000 cicli a eseguire su un processore, e assumendo che diverse iterazioni del loop siano invece eseguibili in parallelo, qual è lo speedup che si ottiene eseguendo questo programma su 10 processori, rispetto ad un solo processore?

- 21
- 10
- 7
- 2

▼ soluzione

$$\begin{aligned}
 T_{old} &= T_{sequenziale} = 1000 + 20 * 1000 = 21000 \\
 T_{new} &= T_{parallelo} = 1000 + 20 / 10 * 1000 = 3000 \\
 \text{Speedup} &= T_{old} / T_{new} = 21000 / 3000 = 7
 \end{aligned}$$

speedup con legge di Amdahl

1000 cicli per eseguire → **CPI** di work = 1; noi però possiamo parallelizzare solo il loop di 20 ripetizioni.

speedup = tempo vecchio/t new → che in questo caso sono:

Tvecchio = su un solo processore → $1000 + 20000(\text{dentro loop}) = 21000$ cicli

Tnew = versione parallela con 10 processori → $1000 + 20000/10(\text{dentro loop}) = 3000$

$$\rightarrow \text{speedup} = 21000 / 3000 = 7$$

se io avessi 20 processori: $T_{new} = 1000 + 1000 = 2000 \rightarrow 21000 / 2000 = 10,5$

se io avessi 40 processori: $T_{new} = 1000 + 500 \dots 21000 / 1500 = 14$

non sto più guadagnando così tanto al raddoppiare dei miei core.

LEGGE DI ANDALH!!

sono le applicazioni che non hanno abbastanza parallelismo.

indice

RAPPRESENTAZIONE DELL'INFORMAZIONE

Rappresentazioni diverse hanno proprietà diverse.

Sistema decimale posizionale:

- La rappresentazione di un numero intero in base 10 è una sequenza di cifre scelte fra l'insieme {0 1 2 3 4 5 6 7 8 9}

- Il valore di una rappresentazione

$$\begin{aligned}
 &a_N a_{N-1} \dots a_0 , \quad a_{-1} a_{-2} a_{-3} a_{-4} \dots \\
 &\text{è dato da} \\
 &a_N \cdot 10^N + a_{N-1} \cdot 10^{N-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0 \\
 &\quad + \\
 &a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} + a_{-3} \cdot 10^{-3} + a_{-4} \cdot 10^{-4} + \dots
 \end{aligned}$$

(parte intera)

(parte frazionaria)

- $b = 10$ è la **base**

- 10^i è il **peso** della cifra a_i nel valore del numero

$$\begin{aligned}
 253 &= 2 \times 100 + 5 \times 10 + 3 \times 1 = \\
 &= 2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0
 \end{aligned}$$

$$\begin{aligned}
 23,47 &= 2 \times 10 + 3 \times 1 + 4 \times 0,1 + 7 \times 0,01 = \\
 &= 2 \times 10 + 3 \times 1 + 4 \times (1/10) + 7 \times (1/100) = \\
 &= 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}
 \end{aligned}$$

Aritmeticamente si moltiplica la **base** per il **peso** → il peso equivale a 10^i , dove i è la posizione della cifra.

dopo la virgola si moltiplicano per frazioni di 10.

Il numero massimo rappresentabile con n cifre è dato dalla cifra più grande ripetuta per n volte → quindi 99999... ovvero $((10^N) - 1)$

Questa ci serve per sapere quanti bit posso rappresentare con la mia rappresentazione.

Rappresentazione binaria:

- La rappresentazione di un numero intero in **base 2** è una sequenza di cifre scelte fra **{0,1}** :

• es: 10, 110, 1

- Il valore di una rappresentazione

$a_N \dots a_0, a_{-1}a_{-2} \dots a_{-3}a_{-4}$

è dato da

(parte intera)

$$a_N \cdot 2^N + a_{N-1} \cdot 2^{N-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + \\ a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + a_{-3} \cdot 2^{-3} + a_{-4} \cdot 2^{-4} + \dots$$

(parte frazionaria)

- b = 2** è la **base**, 2^i è il **peso** della cifra a_i nel valore del numero

• 10	= $1 \cdot 2^1 + 0 \cdot 2^0 = 2$
• 110	= $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 4 + 2 + 0 = 6$
• 1	= $1 \cdot 2^0 = 1$

L'informazione interna ad un computer è codificata con sequenza di due simboli: **0** e **1** → sono le cifre della **base**.

si è scelta la logica binaria perchè è più facile realizzare dispositivi che si interfaccino su due stati.

Parliamo di **bit** come **unità elementare di informazione** (binary digit).

Raggruppato in sequenza → **byte** e **word** (possono essere sequenze di vari bits in base al calcolatore)

la **word** è la grandezza che si utilizza per designare la dimensione della **parola** con cui un calcolatore sa lavorare.

l'unità di lavoro con cui un calcolatore processa le informazioni

▼ esempi di processori con word diverse

- Il massimo numero rappresentabile con N cifre è **99...9** (N volte 9, la cifra che vale di più), pari a $10^N - 1$
 - es: su tre cifre il massimo numero rappresentabile è **999** pari a $10^3 - 1 = 1000 - 1$

- Quindi se voglio rappresentare K diversi numeri (cioè 0, 1, 2, ..., K-1) mi servono almeno almeno x cifre dove 10^x è la più piccola potenza di 10 che supera K
 - es: se voglio 25 configurazioni diverse mi servono almeno 2 cifre perché $10^2 = 100$ è la più piccola potenza di 10 maggiore di 25

- Il massimo numero rappresentabile con N cifre è **11...1** (N volte 1, la cifra che vale di più), pari a $2^N - 1$
 - es: su tre cifre il massimo numero rappresentabile è **111** pari a $2^3 - 1 = 8 - 1 = 7$

- Quindi se voglio rappresentare K diversi numeri (cioè 0, 1, 2, ..., K-1) mi servono almeno almeno x cifre dove 2^x è la più piccola potenza di 2 che supera K
 - es: se voglio 25 configurazioni diverse mi servono almeno 5 cifre perché $2^5 = 32$ è la più piccola potenza di 2 maggiore di 25

- **word**: sequenza di 32, 64, ... bits (4, 8, ... Bytes)

- È la *parola* con cui un calcolatore sa lavorare
 - Processori a 8bit (parole da 1Byte): Intel8080, Zilog Z80



- **word**: sequenza di 32, 64, ... bits (4, 8, ... Bytes)

- È la *parola* con cui un calcolatore sa lavorare
 - Processori a 32bit (parole da 4Byte): IA32, ARMv3-ARMv7, RISC-V



Il numero massimo rappresentabile con n cifre è una sequenza di n volte 1, che è la cifra più grande → ovvero $(2^N) - 1$

se avessi progettato il mio automa a stati finiti e mi servissero 25 stati da rappresentare
→ allora mi servono per forza 5 bit in rappresentazione binari

Conversioni di interi: **da BASE 10 → a BASE 2**

da BASE 2 a BASE 10 basta fare una somma pesata del numero

- Somma pesata delle cifre binarie:

$$\begin{aligned} \text{es.: } 1101_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13_{10} \end{aligned}$$

- Successione di divisioni per 2:
 - termina quando il resto è 0
- La conversione in binario si ottiene leggendo i resti determinati in ordine inverso

es.:

	/ 2	/ 2	/ 2	/ 2
13	6	3	1	0
	1	0	1	1

Quozienti

Resti

 $13_{10} = (\underline{\underline{1101}})_2$

▼ esempi di numeri e le loro conversioni

0	0	8	1000	16	10000
1	1	9	1001	17	10001
2	10	10	1010	18	10010
3	11	11	1011		...
4	100	12	1100		
5	101	13	1101		
6	110	14	1110		
7	111	15	1111		

2^0	=	1	2^8	=	256
2^1	=	2	2^9	=	512
2^2	=	4	2^{10}	=	1024
2^3	=	8	2^{11}	=	2048
2^4	=	16	2^{12}	=	4096
2^5	=	32		...	
2^6	=	64	2^{16}	=	65536
2^7	=	128		...	
			2^{24}	\cong	16 milioni
				...	

ARITMETICA BINARIA:

Addizione è analoga in **base 2** → però $1 + 1$ non fa 11; ma fa 10, perchè io raggiungo il riporto e aggiungo 1 e dove ho fatto il riporto metto 0

• addizione:

$$\begin{array}{l} 0+0=0 \\ 0+1=1 \\ 1+0=1 \\ 1+1=0 \text{ col riporto di 1} \end{array}$$

$$\begin{array}{r} \overset{1}{\underset{1}{\underset{1}{\text{0}}}} \text{0101} + \text{5}_{10} + \\ \text{0011} = \text{3}_{10} = \\ \hline \text{1000} \quad \text{8}_{10} \end{array}$$

• moltiplicazione:

$$\begin{array}{l} \cdot \text{ es.: per } 2, 2^2, 2^3, \dots \leftrightarrow \text{shift (traslazione) verso sinistra di 1, 2, 3 bit} \\ \text{1101} \times \text{100} = \text{110100} \\ (13 \times 4 = 52) \end{array}$$

• sottrazione

$$\begin{array}{l} 0-0=0 \\ 0-1=1 \text{ col prestito di 1 dalla cifra precedente} \\ 1-0=1 \\ 1-1=0 \end{array}$$

Moltiplicazione: ci limitiamo a vedere il caso notevole per potenze di 2; ovvero uno shift verso sinistra, dell'esponente del numero per cui sto modificando.

shift → la mia stringa spostata di x posizioni e a destra mi troverò $x0$ in più rispetto a prima.

la moltiplicazione realizzata su una CPU costa molto; mentre lo **shift** costa poco come la add → quindi si cerca di utilizzare lo shift

rappresentazione numero dentro al computer:

utilizza la notazione binaria. Ogni numero viene rappresentato con un numero finito di cifre binarie.

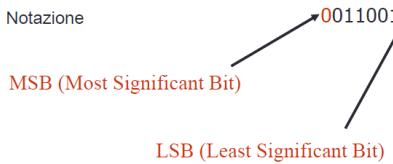
numeri di "tipo" diverso hanno rappresentazioni diverse → numeri interi, razionali, positivi ,negativi ,

→ **interi positivi** si rappresentano con **4 o 8 byte(long int)**; tipo **carattere 1 solo byte**

Notazione binaria: la cifra più significativa è quella più a sinistra e quella meno significativa quella più a destra

- Abbiamo già incontrato alcuni termini utili:
 - byte**: una sequenza di 8 bit
 - word** (parola): 2, 4, 8 byte (dipende dalla macchina) unità minima che può essere fisicamente letta o scritta nella memoria (ed elaborata)
- Tipicamente gli interi positivi si rappresentano usando 4 o 8 byte
 - Esistono varianti a 2 byte (es. il tipo short int in C)

Notazione



Alcuni punti importanti:

- se uso 4 byte (32 bit) posso rappresentare solo i numeri positivi da 0 a $2^{32}-1$, che sono molti ma non tutti!
- se moltiplico o sommo due numeri molto elevati posso ottenere un numero che non è rappresentabile
 - es: vediamo cosa succede in base 10 con solo 3 cifre:
 $500 + 636 = 1136$ risultato 136

se uso solo 3 cifre non ho lo spazio fisico per scrivere la prima cifra (1) che viene 'persa', è un fenomeno chiamato **overflow**

$$\begin{array}{r} 101 + \\ 110 = \\ \hline 1011 \end{array}$$

il primo 1 non trova spazio

1136 in realtà non sarei in grado di rappresentarlo perchè non ho una quarta cifra, e sarebbe un grosso errore.

se non ho lo spazio fisico; quindi viene persa una cifra; ottengo un errore chiamato **overflow**

per rappresentare numeri sia positivi che negativi ci sono diverse convenzioni di rappresentazione:

Modulo e segno:

il bit più a sinistra assume il significato di **segno** e gli altri numeri il **modulo**.

modulo e segno con 3 bit → codifica semplice (non devo cambiare niente) la prima cifra è un + o un -

però non funziona più l'operazione di somma e sottrazione.

bisogna cambiare la logica di addizione e sottrazioni.

- Modulo e segno** (es con 3 bit)

0 segno +
1 segno -
- codifica semplice
- operazioni aritmetiche più complesse

es.: $+2 \leftrightarrow 010$ e $-2 \leftrightarrow 110$

$$\begin{array}{r} 001 + \\ 110 = \\ \hline 111 \end{array} \quad \begin{array}{r} 1 + \\ -2 = \\ \hline -3 \end{array}$$

Errato!

- Occorre differenziare tra i bit del numero e quelli di segno
- Bisogna codificare in modo diverso le operazioni aritmetiche.

Complemento a due (es con 4 bit)

es: $+5 = 0101$ *Come si rappresenta -5 ??*

Partendo da $+5 = 0101$:

- si invertono gli 1 con gli 0: 1010
- si aggiunge 1:

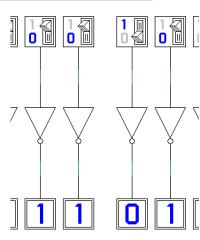
$$\begin{array}{r} 1010 + 1 = 1011 = -5 \\ -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ = -8 + 0 + 2 + 1 = -5 \end{array}$$

- Il primo bit non rappresenta solo il segno!
- Non occorre più pertanto differenziare i bit.

Complemento a uno

Complemento a uno - Wikipedia

Il complemento a uno (in inglese ones' complement), o complemento alla base diminuita, è un metodo di
W https://it.wikipedia.org/wiki/Complemento_a_1



Complemento a due:

bisogna **complementare la cifra** →

invertire tutte le cifre(come un

complemento a 1)

poi per fare il complemento a due, ci

aggiungo un 1 alla fine.

il primo bit rappresenta il segno e si porta dietro il significato della rappresentazione posizionale.

l'aritmetica binaria rimane invariata

$$\begin{array}{r} 0001 + \\ 1011 = \\ \hline 1100 \end{array} \quad \begin{array}{r} 1 + \\ -5 = \\ \hline -4 \end{array}$$

OK!

è la più utilizzata.

- **Ottale** (base 8): { 0, 1, 2, 3, 4, 5, 6, 7 } (10↔8)
- **Esadecimale** (base 16): { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F } (10↔16)

- Usate perché semplici conversioni da base 2 a base 8 o 16:
 - Partendo dal numero in base 2:
 - Se ne raggruppano le cifre a blocchi di 3 (ottale) o 4 (esadecimale)
 - Si convertono i singoli gruppi nella base di destinazione

Quando si convertono, se restasse fuori una posizione vuota, si aggiungerebbe uno 0 in quella posizione.
si scelgono perchè hanno una conversione semplice tra loro e la base 2

Sistema di codifica BCD:

E' una codifica di cifre decimali in binario.

Esempio: 111000110101₂

Conversione in base 8

$$\underbrace{111}_{3} \underbrace{000}_{3} \underbrace{110}_{3} \underbrace{101}_{3} = 7065_8$$

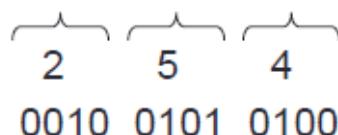
Conversione in base 16

$$\underbrace{1110}_{4} \underbrace{0011}_{4} \underbrace{0101}_{4} = E35_{16}$$

però le cifre legali, sono solo quelle dell'alfabeto decimale. Infatti nel **BCD** non vado mai oltre la cifra numero **9(1001)**; quindi tutte le cifre che si possono ottenere ad arrivare ad **1111** non si possono rappresentare. Non è la più efficiente, ma è comoda perchè è più veloce.

BCD (Binary-Coded Decimal)

- Si codificano in binario (4 bit) le singole cifre decimali.
- es.: 254



- nessun errore di conversione
- precisione dei calcoli decimali
- spreco di cifre
- usato nelle calcolatrici tascabili

Rappresentazione numeri razionali:

hanno numero finito di cifre dopo la virgola; rappresentati solitamente su **4/8 byte(precisione singola/doppia)**.

(i floating point hanno 32 bit; i double hanno 64 bit).

Rappresentazione a virgola fissa: io partiziono in maniera statica i bit che ho dentro alla cifra, dividendoli in parte intera e parte frazionaria.
inoltre è inefficiente! perchè spreco molte posizioni di bit(utilizzo di 0)

- **Razionali**
 - numero finito di cifre periodiche dopo la virgola
 - ad esempio [3.12](#) oppure [3.453](#)
 - rappresentazione solitamente su 4/8 byte
- **Rappresentazione in virgola fissa :**
 - riservo X bit per la parte frazionaria
 - es : con 3 bit per la parte intera e 2 per quella frazionaria [011.11](#), [101.01](#)

- Come si converte in base 10 una rappresentazione in virgola fissa?

- es:

$$\begin{aligned} 101.01 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = \\ &= 4 + 0 + 1 + 0 + 0.25 = 5.25 \end{aligned}$$

dove $2^{-1} = 1/2 = 0.5$, $2^{-2} = 1/2^2 = 0.25$
e in generale $2^{-n} = 1/2^n$

Parte intera	Parte frazionaria
--------------	-------------------

- Problemi della rappresentazione in virgola fissa
 - overflow**
 - underflow**
 - quando si scende al di sotto del minimo numero rappresentabile
 - es. vediamo in base 10, con 2 cifre riservate alla parte frazionaria $0.01 / 2 = 0.005$ **non rappresentabile usando solo due cifre!!!**
- Problemi della rappresentazione in virgola fissa (cont.)
 - spreco di bit per memorizzare molti '0' quando lavoro con numeri molto piccoli o molto grandi
 - es. in base 10, con 5 cifre per la parte intera e 2 cifre per la parte frazionaria **10000.00 oppure 00000.02**
 - i bit vengono usati più efficientemente con la notazione **esponenziale o floating point** (virgola mobile)

Rappresentazione a virgola mobile:

- Rappresentazione in **virgola mobile**
 - idea:**
 - quando lavoro con numeri molto piccoli uso tutti i bit disponibili per rappresentare le cifre dopo la virgola
 - quando lavoro con numeri molto grandi le uso tutte per rappresentare le cifre in posizioni elevate
 - questo permette di rappresentare numeri piccoli con intervalli minori fra loro rispetto ai numeri grandi
 - questo riduce gli errori nel calcolo a parità di bit utilizzati
- ogni numero N è rappresentato da una coppia (**mantissa M, esponente E**) con il seguente significato
 $N = M \cdot 2^E$
- esempi:**
 - in base 10, con 3 cifre per la mantissa e 2 cifre per l'esponente riesco a rappresentare
 $349\ 000\ 000\ 000 = 3,49 \cdot 10^{11}$
 con la coppia (3.49,11) perché $M = 3.49$ ed $E = 11$
 - in base 10, con 3 cifre per la mantissa e 2 per l'esponente riesco a rappresentare
 $0.000\ 000\ 002 = 2.0 \cdot 10^{-9}$
 con la coppia (2.0,-9) perché $M = 2.0$ ed $E = -9$
 - sia **0.000 000 002** che **349 000 000 000** non sono rappresentabili in virgola fissa usando solo 5 (3M+2E) cifre decimali

Utilizza una coppia **M,E** → mantissa e esponente(**2^E**).

Se decidessi di usare 3 cifre per la mantissa e 2 per l'esponente riesco a rappresentare molto; anche rappresentazioni piccole. In base 10 l'esponente è 10^E .

Standard IEEE 754:

E' uno standard che è stato definito per rendere univoca la maniere in cui fra diversi sistemi di calcolo, si rappresentano i numeri con la virgola mobile. Posso usare librerie se la mia macchina non li rappresenta così. Diversi tipi di precisione **float** e **double**. Il numero è rappresentato da **esponente, mantissa** e un **bit per il segno**.

Precisione singola = 32 bit. → dei 32 bit, **23 li uso per la mantissa; 1 per il segno e 8 per l'esponente.**

E max = 127; Emin = -126

- Si specificano 3 parametri:
 - P:** precisione o numero di bit che compongono la mantissa
 - E_{max}:** esponente massimo
 - E_{min}:** esponente minimo
- Ad esempio per la **precisione singola** (32 bit)
 - P=23, E_{max}=127 e E_{min}=-126**
 - 1 bit **segno**; 8 bit **esponente**; 23 bit **mantissa**
 - La mantissa viene normalizzata scegliendo l'esponente in modo che sia sempre nella forma 1,xxxx...
 - L'esponente è polarizzato, ovvero ci si somma **E_{max}**
 - costante di polarizzazione o **bias**
 - $0.15625_{(10)} = \frac{1}{8} + \frac{1}{32} = 2^{-3} + 2^{-5} = 0.00101_{(2)}$
 - $0.00101_{(2)} = 1.01_{(2)} \times 2^{-3}$ **Normalizzazione** della mantissa
 - Parte intera della mantissa (prima cifra) sempre diversa da zero
 - In base 2 l'unica cifra diversa da 0 è 1 → **posso non memorizzarla**
 - Parte frazionaria della mantissa: $.01_{(2)}$
 - Esponente: -3
 - Esponente polarizzato (precisione singola): $-3 + 127 = 124$

La mantissa viene **normalizzata**: devo sempre far scorrere il mio numero, assumendo che **la prima cifra dopo la virgola sia un 1** → 1,xxxx così facendo posso non memorizzare questo 1 e posso usare un bit in più!! (in un sistema binario) per l'esponente si utilizza una rappresentazione per **bias(polarizzato)**: si somma sempre l'esponente massimo all'esponente attuale. Parola a 32 bit → primo bit = segno; successivi 8 = esponente; successivi 23 = mantissa. Per la precisione doppia non cambia niente concettualmente. Cambiano il numero di bit rappresentabili.

- Per la **precisione doppia (64 bit)**
 - **P=52, E_{max}=1023 e E_{min}=-1022** (1 bit segno; 11 bit esponente)
 - **Parte frazionaria della mantissa:** .01₍₂₎
 - **Esponente:** -3
 - **Esponente polarizzato** (precisione singola): $-3 + 1023 = 1020$

Relazioni per lo standard IEEE 754:

	esp	M	numero
Numero normalizzato	$0 < \text{esp} < 255$	qualsiasi	$(-1)^s (1, M) 2^{\text{esp}-127}$
Numero denormalizzato	$\text{esp}=0$	$M \neq 0$	$(-1)^s (0, M) 2^{-126}$ Riduce la perdita di precisione se underflow
Zero	$\text{esp}=0$	$M = 0$	$(-1)^s 0$
Infinito	$\text{esp}=255$	$M = 0$	$(-1)^s \infty$
NaN (Not a Number)	$\text{esp}=255$	$M \neq 0$	NaN

- **Numero normalizzato:** il numero ha implicito un 1 prima della virgola; esponente tra 0 e 255 esclusi(esclusi perchè il fatto che l'esponente valga 0 o 255; assumono un significato particolare).
- **NaN:** è un'eccezione da gestire come per la divisione per 0.
- Quando ho **esp 0 e mantissa ≠ 0** allora mi trovo in **underflow**, ovvero non ho abbastanza cifre per rappresentare il mio numero.
- **Numero denormalizzato:** c'è implicito uno 0 dopo la virgola.
Riduco la perdita di precisione se c'è underflow, così facendo posso aggiungere un bit di precisione alla mantissa, perchè do per scontato che ci sia uno 0 dove prima avrei

dovuto avere un 1 → **adesso ho 0,M invece di 1,M !!**

guadagno molti shift; perchè prima per forza avevo un uno a sinistra della virgola, adesso invece posso avere uno 0. Quindi posso avere come **mantissa 0,000000...1 e guadagnare dei shift con l'esponente**; prima avrei dovuto fare 1,... **andando a mettere tutti quelli zero, come shift nell'esponente.**

riduco la perdita di informazione nel caso di underflow.

Numero più piccolo con

normalizzazione: la mantissa è 1,... già di base(definizione) → quindi è per forza 1,... * 10 alla - 126

numero più piccolo senza

normalizzazione: la mantissa è 0,00...1 → quindi guadagno shift, infatti l'esponente arriva a -149

• I numeri piu' piccoli (vicini allo zero) rappresentabili

- Esp=1, M=0 → $\pm 2^{-126} \approx \pm 1.17549 \times 10^{-38}$ (norm. singola)
 $\pm 2^{-1022} \approx \pm 2.22507 \times 10^{-308}$ (norm. doppia)
- Esp=0, M=00...1 → $\pm 2^{-149} \approx \pm 1.40130 \times 10^{-45}$ (denorm. singola)
 $\pm 2^{-1074} \approx \pm 4.94066 \times 10^{-324}$ (denorm. doppia)

• I numeri finiti piu' grandi (lontani dallo zero) rappresentabili sono

- Esp=254, M=11...1 → $\pm (1-2^{-24}) \times 2^{128} \approx \pm 3.40282 \times 10^{38}$ (prec. singola)
 $\pm 1.79769 \times 10^{308}$ (prec. doppia)

Mantissa rappresentata con **modulo e segno**, mentre esponente rappresentato con **complemento a 2**.

Conversione da decimale a standard IEEE 754:

- Parte intera: uguale a quella da base 2 a base 10.
- Parte frazionaria: moltiplico la cifra per 2 e annoto se ho del riporto e vado avanti così finchè o la parte frazionaria è arrivata a 0 o ho finito le cifre per rappresentare.

- Voglio rappresentare il numero -36,47 usando la convenzione IEEE754, ovvero vedere come viene realmente memorizzata la variabile

`float f=-36.47`

- Prima di tutto calcolo la rappresentazione binaria. A tal fine calcolo parte intera e frazionaria mediante iterazione di divisioni/moltiplicazioni per 2.

$$0.15625_{(10)} = \frac{1}{8} + \frac{1}{32} = 2^{-3} + 2^{-5} = 0.00101_{(2)}$$

Esempio (2)

Si continua finchè la parte frazionaria è diversa da zero (o finchè ci sono cifre nella rappresentazione...)

(parte intera)		(parte frazionaria)	
36	div 2	(resto)	
18		0	0 , 94
9		0	0 , 94 x 2
4		1	1 , 88
2		0	0 , 88 x 2
1		0	1 , 76
0			0 , 76 x 2
			1 , 52
			0 , 52 x 2
			1 , 04
			0 , 04 x 2
			0 , 08
...	

$$-36.47_{10} = -100100,011110..._2 = -1,001001111 \times 2^5$$

Ora ho tutti gli elementi da collocare nella rappresentazione.

(parte frazionaria)	
0,15625	x 2
0,3125	x 2
0,625	x 2
0,25	x 2
0,5	x 2
0,0	x 2

▼ Strumento per fare delle prove su questa rappresentazione:

<https://www.hschmidt.net/FloatConverter/IEEE754.html>

la mantissa subito è **denormalizzata** perchè l'esponente è zero.
 quando il valore rappresentato dai 8 bit dell'esponente è pari a 1 → vuol dire che ho 2^{-126} e con mantissa la più piccola possibile; allora **ho il numero più piccolo rappresentabile**

! In generale fixed point si usa quando si hanno dei vincoli, consumi energia, o costi, ecc..., ma nella maggior parte delle volte, si usa il floating point perchè molto più preciso.

- Vogliamo rappresentare i giorni della settimana :
 - $\{Lu, Ma, Me, Gio, Ve, Sa, Do\}$
 - usando sequenze 0 e 1
- Questo significa costruire un **codice**, cioè una tabella di corrispondenza che ad ogni giorno associa una opportuna sequenza
- In principio possiamo scegliere in modo del tutto arbitrario....

Noi dentro al progetto non troveremo scritti lunedì martedì, ... ; ma troveremo dei circuiti logici che traducono delle sequenze di bit in questi giorni.

Una possibile codifica binaria per i giorni della settimana

Lunedì	0100010001
Martedì	001
Mercoledì	1100000
Giovedì	1
Venerdì	101010
Sabato	111111
Domenica	000001

• **Problema:** la tabella di corrispondenza fra codifiche tutte di lunghezza diversa

- spreco di memoria
- devo capire come interpretare una sequenza di codifiche
- **11000011** = Me Gio Gio
- **110000011** = Gio Gio Do Gio

• **Soluzione:** si usa un **numero di bit uguale per tutti**, il minimo indispensabile

Problemi:

difficoltà della gestione, spreco di bit → **complessità della realizzazione**.

solo con lunedì sono vincolato a dire che la mia macchina lavorerà con una **sequenza di 10 bit**; poi come faccio a sapere che a volte devo leggere 10, a volte 3, ... tutto molto complicato e ci vogliono regole per gestire tutti questi casi → **inefficienza risorse**.

ambiguità: sequenza che posso rappresentare più valori. difficoltà ad interpretare i

risultati; non ho una regola; **per questo un calcolatore lavora con una word** → io leggo sempre 32(o 64,...) bit non più non meno.

Soluzione: numero di bit uguale per tutti i giorni; che numero? → il minimo indispensabile e sappiamo come farlo: la minima potenza di 2 che è maggiore di 7(giorni settimana): **2^3 che è 8 → 3 cifre**

bisogna specificare che la combinazione 111 non rappresenta informazione utile nel mio sistema.

- Per rappresentare 7 oggetti diversi servono almeno 3 bit (minima potenza di due che supera 7 è $8 = 2^3$) quindi :
 - 000 Lunedì
 - 001 Martedì
 - 010 Mercoledì
 - 011 Giovedì
 - 100 Venerdì
 - 101 Sabato
 - 110 Domenica
 - 111 non ammesso

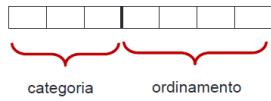
Rappresentazione di caratteri e stringhe:

Stringhe == sequenza di caratteri terminante in modo particolare(**\n**).

Ci sono più rappresentazioni di caratteri e stringhe:

- Tipologia di caratteri:
 - alfabeto e interpunzioni: A, B, ..., Z, a, b, ..., z, ;, ", ..
 - cifre e simboli matematici: 0, 1, ..., 9, +, -, >, ..
 - caratteri speciali: £, \$, %, ..
 - caratteri di controllo: CR, DEL,
- Le *stringhe* sono sequenze di caratteri terminate in modo particolare.
- I *caratteri* sono un insieme finito di oggetti e seguono la strategia vista per i giorni della settimana
- **ASCII (American Standard Code for Information Interchange):**
 - Codice a 7 bit (standard)
- **ASCII esteso a 8 bit (non standard)**
 - es.: A 01000001
 - (00101000
- **UNICODE**: su 16 bit (65536 diverse configurazioni): più recente, permette di rappresentare anche alfabeti diversi e simboli per la scrittura di lingua orientali.

- **ASCII a 7 bit**
I 7 bit sono suddivisi logicamente in due campi di 3 e 4 bit.



I primi tre bit rappresentano categorie di caratteri, mentre gli ultimi quattro servono a rispettare l'ordinamento dei caratteri all'interno di ogni categoria.

Categorie

1°bit	2°bit	3°bit	Caratteri rappresentati
0	1	0	simboli di punteggiatura, simboli speciali e di operazione
0	1	1	numerali
1	0	0	maiuscole (A - O)
1	0	1	maiuscole (P - Z)
1	1	0	minuscole (a - o)
1	1	1	minuscole (p - z)

3 bit per la categoria: non tutte le sequenze hanno un significato. se il primo bit è a 1 sono lettere... guardare immagine.

i bit restanti(4) sono i bit che codificano in binario le varie cifre che possono essere rappresentate:

Nel caso di numeri(che sono in base 10) allora sappiamo che le cifre sono 9; quindi risulta conveniente la codifica **BCD**(raggruppo cifre... già fatta). quindi **001 0010** → è il carattere che rappresenta il numero 2 il codice 2(0010) rappresenta sempre **b**, **sono i primi 3 bit a dirmi se è maiuscola o minuscola.**

Immagine digitale: è una griglia con una sequenza di pixel i quali sono rappresentati in memoria(ce ne sono varie in realtà); rappresenta l'intensità o del livello di grigio(se bianca e nera) o l'intensità di ogni canale(un canale rappresenta un colore; ci sono più canali) quindi sto rappresentando 255 livelli(8 bit) di intensità su ogni canale. sono formate da **pixels**.

- 4°, 5°, 6°, 7° bit
- Nei numerali sono costituiti dalla codifica in binario su 4 bit delle cifre decimali (codice BCD).
- Per i caratteri dell'alfabeto la codifica è tale da rispettare l'ordinamento alfabetico

• Esempi

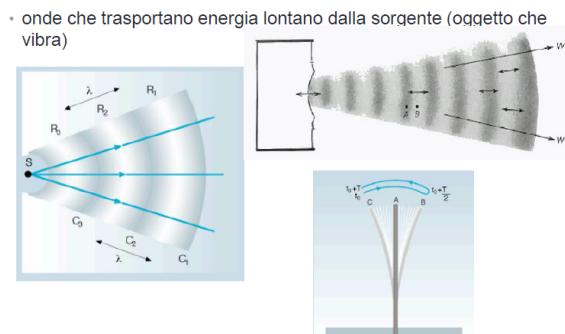
b (2ª lettera) : 110 0010
 2
B: 100 0010

Un'**immagine digitale** può essere vista come una funzione bidimensionale $f(x,y)$, dove f rappresenta l'**intensità o livello di grigio** dell'immagine in quel punto: 0 rappresenta il **nero**, 255 il **bianco**

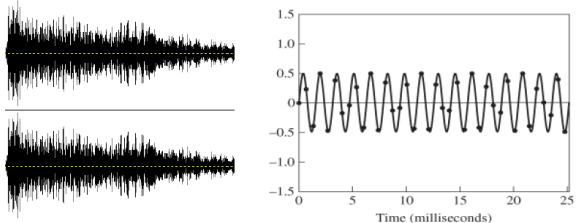


Un'**immagine** è quindi una **matrice** di elementi chiamati **pixels** (picture elements).

File audio: ci sono più possibili codifiche, tipo **MP3**, che è un **formato compresso** → memorizzo parte della sorgente sonora. campionamento dell'ampiezza delle onde sonore con una certa frequenza. una forma d'onda audio non è altro che una rappresentazione della **variazione della frequenza del mio segnale audio**. questo è rappresentabile in binario, quantizzando i valori che ho sull'asse sinistro e io memorizzo il livello discretizzato(non tutti i numeri) dei valori della frequenza



- onde che trasportano energia lontano dalla sorgente (oggetto che vibra)
- E' possibile rappresentare il suono con la sua forma d'onda
- Al calcolatore, basta rappresentare una opportuna sequenza di campioni della forma d'onda



Sui file **raw** (gli audiofili usano solitamente i **FLAC** per quel che riguarda la musica) noi memorizziamo più informazioni di quelli che l'orecchio(anche occhio per immagini/video)

riesce a percepire, per questo l'MP3 è compresso, ovvero toglie le frequenze che non sono percepibili da un umano; ovviamente però è sempre una compressione, quindi si perdono anche informazioni utili, soprattutto in base a cosa fai la compressione: vuoi che occupi poco spazio? allora perderai qualità dell'audio. se no peserà molto di più. Poi dipende anche dal sistema su cui andrai a sentire gli audio(le casse del computer,...). *Maxbubblegum*: "Ci sono servizi di straming che fanno della bandiera della qualità un loro cavallo di battaglia → Tidal".

Indice

RETI LOGICHE

Reti combinatorie → circuiti che dato un input producono un output → implementazione di una funzione logica

però da sola una rete combinatoria non mi permette di creare un automa a stati, perchè la rete combinatoria non ha memoria.

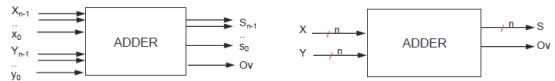
reti logiche sequenziali → abbiamo bisogno di introdurre memoria nel nostro sistema. macchine a stati finiti.

- Livello di astrazione che studia i sistemi digitali a livello di componenti LOGICI elementari indipendentemente dalla tecnologia con cui il sistema viene realizzato.
- **Rete logica:** sistema digitale avente n segnali binari di ingresso ed m segnali binari di uscita.
- I segnali sono rigorosamente binari (0/1).
- I segnali sono grandezze funzioni del tempo

$$X = \{x_{n-1}(t), \dots, x_0(t)\}$$

$$Z = \{z_{m-1}(t), \dots, z_0(t)\}$$

$$z_i(t) = f_i(x_{n-1}(t), \dots, x_0(t))$$
- I segnali di ingresso ed uscita delle reti logiche possono essere singoli segnali binari (es. RESET) o segnali digitali composti in parole codificate come un insieme di segnali binari

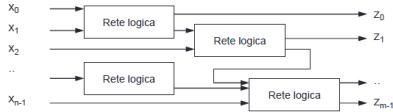


I segnali di ingresso ed uscita possono essere singoli(1 o 0) oppure segnali composti in parole codificate.

l'adder prende in ingresso dei input e out ha la somma(un composto) e l'overhead singole frecce = singolo segnale

Frecce con taglietto e lettera n = **segnale composto da n bit**

- **Proprietà di interconnessione:** l'interconnessione di più reti logiche, aventi per ingresso segnali esterni o uscite di altre reti logiche e per uscite segnali di uscita esterne o ingressi di altre reti logiche, è ancora una rete logica



- Reti COMBINATORIE $z_i(t) = f(x_0(t), \dots, x_{n-1}(t))$
- Reti SEQUENZIALI $z_i(t) = f((x_0(t), \dots, x_{n-1}(t), t)$
- Rete **combinatoria**: ogni segnale di uscita dipende solo dai valori degli ingressi in quell'istante
- Rete **sequenziale**: ogni segnale di uscita dipende dai valori degli ingressi in quell'istante e dai valori che gli ingressi hanno assunto negli istanti precedenti

Una transizione di STATO riassume tutto quello che è successo in passato(nel nostro esempio quando riconoscevamo un bit). Memorizzo solo gli stati importanti = **le transizioni rilevanti**

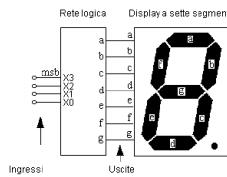
Transistori = effetti di propagazione del segnale fisico(effetti di ritardo).

Esempi:

Conversione di valori BCD su display a sette segmenti

- Descrizione comportamentale (a parole): progettare una rete logica che permette la visualizzazione su un display a sette segmenti di un valore in codice BCD.
- **Codifica BCD:** impiego di 4 cifre binarie per la rappresentazione di un numero decimale da 0 a 9.

• **Ese:** 15 decimal
1111 binario
0001 0101 BCD

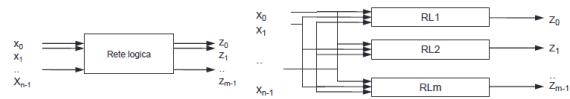


- L'uscita $Z=\{a,b,\dots,g\}$ dipende in ogni istante dalla configurazione degli ingressi $\{x_3, x_2, x_1, x_0\}$

2^4 è il più piccolo numero intero più grande di 9.
per questo uso 4 cifre binarie ed inoltre uso la codifica BCD.

- **Proprietà di decomposizione:** una rete logica complessa può essere decomposta in reti logiche più semplici (fino all'impiego di soli blocchi o gate elementari)

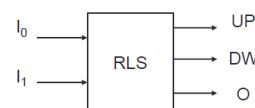
- **Proprietà di decomposizione in parallelo:** una rete logica a m uscite può essere decomposta in m reti logiche ad 1 uscita, aventi ingressi condivisi



- **Rete combinatoria:** rete senza memoria (l'uscita cambia instantaneamente dopo che l'ingresso è cambiato)
- **Rete sequenziale:** rete con memoria; è una rete in cui l'uscita cambia in funzione del cambiamento dell'ingresso e della specifica configurazione interna in quell'istante (STATO). Lo stato *riassume* la sequenza degli ingressi precedenti
 - Una rete combinatoria, quindi non ha STATO. Non ricorda gli ingressi precedenti.
 - Transitori a parte, basta conoscere gli ingressi in un istante per sapere esattamente quali saranno tutte le uscite nel medesimo istante.
 - Le reti sequenziali, invece, hanno memoria. Per sapere l'uscita in un certo istante ho due possibilità:
 - Mi ricordo TUTTI gli ingressi che si sono presentati alla rete dalla sua accensione
 - Memorizzo uno STATO del sistema, che riassume in qualche modo tutti gli ingressi precedenti al fine di valutare il valore delle uscite.

Progettare la rete logica di gestione di un ascensore.

- La rete ha tre uscite UP, DW e O. UP, DW indicano le direzioni su e giù mentre O vale 1 se la porta deve essere aperta e 0 altrimenti. La rete ha come ingresso due segnali che indicano il piano {0,1,2,3} corrispondente al tasto premuto. Per calcolare l'uscita è necessario conoscere il piano corrente che indica lo stato interno.



il mio edificio ha 4 piani → 2 bit per codificare(2^2)
in base alla codifica di questi 2 bit, cambiano gli output.
non posso determinare il valore delle uscite senza sapere qual'è lo stato

interno(voglio andare al piano 2, seno al terzo? allora DW; sono al primo? allora UP...)

già il riconoscitore di sequenza è una rete sequenziale

Descrizione delle reti combinatorie:

Descrizione a parole di una rete logica non è giusta, può essere poco univoca; infatti si fa seguire una **tabella di verità**:

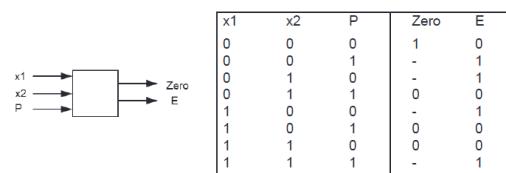
- enumerazione di tutti i possibili ingressi ai quali faccio vedere un'unica configurazione di uscita(o più nel caso ci siano più uscite).
- schemi logici: circuiti/ simboli grafici /...
- forme d'onda: non li vedremo;
- strumento di debug per la progettazione dell'HW
- linguaggi di descrizione dell'HW:
descrive come sono interconnesse le porte logiche. non li vedremo

1. **Descrizione comportamentale a parole:** descrizione a parole del comportamento della rete logica (poco formale e precisa)
2. **Tabelle di verità:** descrizione esaustiva di tutte le configurazioni di uscita per ogni possibile configurazione di ingresso
3. **Mappe:** altra rappresentazione delle tabelle della verità
4. **Espressioni dell'algebra Booleana**
5. **Schema logico:** descrizione strutturale
6. **Forme d'onda:** descrizione comportamentale in funzione del tempo
7. **Linguaggi di descrizione dell'hardware**

- **Tabella di verità:** tabella che associa tutte le possibili combinazioni degli ingressi alle corrispondenti configurazioni delle uscite e indica esaustivamente il comportamento della rete logica
- Se la rete combinatoria ha n ingressi e m uscite, allora la tabella di verità ha $(n+m)$ colonne e 2^n righe
- Oppure per la proprietà di decomposizione si possono definire tante tabelle quante sono le uscite

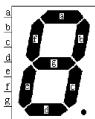
C.2) se le uscite sono indifferenti per alcune configurazioni di ingresso

Esempio: progettare una rete che indichi se due ingressi binari sono entrambi uguali a zero, se il segnale di parità pari è corretto, altrimenti indichi errore



- Si dicono **COMPLETAMENTE SPECIFICATE** se ogni valore della tabella assume il valore logico di vero o falso (1, 0)
- Si dicono **NON COMPLETAMENTE SPECIFICATE** se contengono condizioni di indifferenza. Si verifica in due casi:
C.1) *se alcune configurazioni di ingressi sono vietate*

Esempio conversione BCD 7 segmenti



x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	0	1	0	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	0	1	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	0	1	1	1	0	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	-	-	-	-	-	-	-
1	0	1	1	-	-	-	-	-	-	-
1	1	0	0	-	-	-	-	-	-	-
1	1	0	1	-	-	-	-	-	-	-
1	1	1	0	-	-	-	-	-	-	-
1	1	1	1	-	-	-	-	-	-	-

Le ultime 6 righe sono le configurazione dell'ingresso che in BCD rappresentano cifre che non sono nell'alfabeto decimale(10, 11,..., 15). Lo 0(base10) in BCD è 0000 → allora io ragiono e dico che per rappresentarla visibilmente devo accendere tutti i led tranne quello centrale: le uscite vanno tutte alte(a valore 1) tranne quella centrale(il segmento g).

I trattini sono **condizioni di indifferenza** → si verificano nel caso in cui alcune delle configurazioni di ingresso sono vietate(non voglio che escano); oppure nel caso in cui le uscite sono indifferenti(sono ininfluenti, quindi non mi interessa di come escono, se sono 1 o 0).

Funzioni combinatorie e gate elementari:

Funzioni di una sola variabile in ingresso che è un bit 0 o 1 e una uscita che cambia a seconda della funzione:
una massa che all'uscita è sempre 0 → è un **GROUND**.

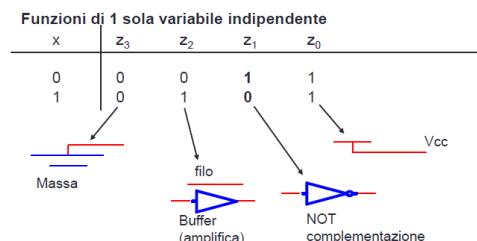
Alimentatore che è sempre 1, è sempre alto → è un **VCC** → **generatore di segnale/generatore di flusso**

se l'uscita assume lo stesso valore dell'ingresso allora si dice **filo** → **amplificatore del segnale o buffer**.

Se l'uscita è l'opposto → **negazione** o **complementazione**; nega l'ingresso → **NOT**

segnale di parità che è un unico bit; le uscite sono 2: zero e errore.
nel secondo caso diventa insignificante il valore di zero, perchè il segnale di errore è 1; quindi c'è stato un errore
→ zero è una condizione di indifferenza.
(so già che c'è un errore, quindi non mi interessa sapere com'è il valore zero)

- Le reti logiche combinatorie sintetizzano funzioni combinatorie.
- Per ogni n , è finito il numero di funzioni combinatorie di n variabili di ingresso. Alcune funzioni combinatorie elementari hanno una rappresentazione logica e grafica elementare (gate)



Funzioni a 2 variabili in ingresso (anche più ingressi vedremo più avanti):

$x_1 \ x_0$	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
0 0	0	0	0	0	0	0	0	0
0 1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1

$z_1 \rightarrow \text{AND}$



vale 1 se e solo se tutti gli ingressi valgono 1 (equivale al prodotto logico in logica positiva)

$z_7 \rightarrow \text{OR}$



vale 1 se e solo se almeno uno degli ingressi vale 1 (equivale alla somma logica in logica positiva)

$z_6 \rightarrow \text{EXOR}$



vale 1 se e solo se x_1 o x_0 valgono 1 ma non entrambi (disegualanza)

$x_1 \ x_0$	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
0 0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1

$z_8 \rightarrow \text{NOR}$



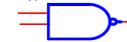
vale 1 se e solo se nè x_1 nè x_0 valgono 1 (l'uscita è il complemento di z_7)

$z_9 \rightarrow \text{EXNOR}$



EQUIVALENCE: vale 1 se e solo se x_1 e x_0 sono uguali (l'uscita è il complemento di z_6)

$z_{14} \rightarrow \text{NAND}$



vale 0 se e solo se nè x_1 nè x_0 valgono 0 (l'uscita è il complemento di z_1)

AND è un **prodotto logico**; se un solo dei due segnali è 0 allora l'uscita è 0

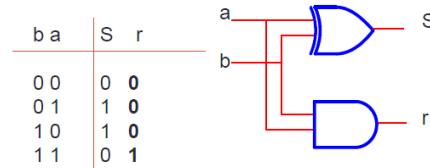
Funzioni combinatorie:

Se c'è un overflow, è un segnale in uscita, ma non viene riportato (half adder).

- Quante sono le possibili funzioni binarie di n variabili ?
- Tutte le combinazioni delle uscite per ogni configurazione di ingresso, ossia 2 elevato al numero delle possibili configurazioni di ingresso

$$N. \text{ conf} = 2^{(2^n)}$$

- Esempio di rete logica con gate elementari:** Progettare un HALF ADDER, ossia un sommatore senza riporto in ingresso



Algebra di Boole:

Sintesi = processo nel grafico:

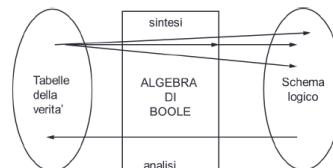
a sinistra abbiamo le tabelle dell'unità che esprimono in modo univoco il comportamento degli input; dalla tabella di verità sintetizzo uno schema/circuito logico

Analisi = processo contrario:

analisi di circuito logico per sapere la tabella

- Uno strumento potente di rappresentazione delle reti logiche combinatorie è data dalle espressioni dell'ALGEBRA DI BOOLE o ALGEBRA DI COMMUTAZIONE

- E' il sistema matematico usato per la sintesi e per l'analisi, per passare dalle tabelle della verità allo schema logico e viceversa



- L'algebra di Boole è un sistema matematico che descrive funzioni di variabili binarie: è composto da
 - un insieme di simboli $B=\{0,1\}$
 - un insieme di operazioni $O=\{+, \cdot, '\}$
 - $+$ somma logica (OR)
 - \cdot prodotto logico (AND)
 - $'$ complementazione (NOT)
 - un insieme di postulati (assiomi) P :

P1) $0 + 0 = 0$	P5) $0 \cdot 0 = 0$	P9) $0' = 1$
P2) $0 + 1 = 1$	P6) $0 \cdot 1 = 0$	P10) $1' = 0$
P3) $1 + 0 = 1$	P7) $1 \cdot 0 = 0$	
P4) $1 + 1 = 1$	P8) $1 \cdot 1 = 1$	

Proprietà di chiusura:

per ogni $a, b \in B$

$$a + b \in B$$

$$a \cdot b \in B$$

- COSTANTI** dell'algebra: le costanti 0 ed 1
- VARIABILE**: un qualsiasi simbolo che può essere sostituito da una delle due costanti

a partire dagli assiomi guardo le proprietà:

per ogni coppia di simboli binari a e b ; la loro somma(o prodotto) logica è ancora parte di $B \rightarrow 0$ o 1

postulati: definisco come funzionano AND OR e NOT; espressi in forma algebrica.

Funzioni Booleane:

- Una **funzione completamente specificata** di n variabili $f(x_{n-1}, \dots, x_1, x_0)$ è l'insieme di tutte le possibili coppie formate da un elemento di B^n (*dominio*) e da un elemento di B (*codominio*).
- La tabella della verità è un tipico modo per descrivere una funzione dell'algebra di Boole.

- Esiste corrispondenza 1:1 tra una tabella della verità e funzione Booleana.

$$f(x_2, x_1, x_0): B \times B \times B \rightarrow B$$

x2	x1	x0	f(x2, x1, x0)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Complementazione**: A complementato si indica come A' oppure \bar{A} .
- Il simbolo \cdot del prodotto logico viene spesso omesso.

funzione di 3 variabili x_2, x_1, x_0 che si possono indicare come le 3 variabili appartenente al dominio e la risultante variabile appartenente al codominio.

Un'**espressione** secondo l'algebra di Boole è una stringa di elementi di B che soddisfa una delle seguenti regole:

- una costante è un'espressione;
- una variabile è un'espressione;
- se X è un'espressione allora il complemento di X è un'espressione;
- se X, Y sono espressioni allora la somma logica di X e Y è un'espressione;
- se X, Y sono espressioni allora il prodotto logico di X e Y è un'espressione.

TEOR: ogni espressione di n variabili descrive una funzione completamente specificata che può essere **valutata** attribuendo ad ogni variabile un valore assegnato.

x2	x1	x0	f(x2, x1, x0)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- Se ogni espressione definisce univocamente una funzione non è vero il contrario: per ogni funzione esistono più espressioni che la descrivono e si dicono logicamente **equivalenti**.

TEOR: una espressione di n variabili descrive in maniera univoca uno schema logico di AND, OR e NOT

La tabella di verità enumera in maniera univoca tutte le possibili configurazioni in cui gli ingressi si presentano(le variabili), ma io posso fare la stessa cosa come **espressione booleana**.

forma **Somma di Prodotti**: vado a guardare tutte le righe della tabella in cui l'uscita vale 1 (ce ne sono 3 in questo caso);

In pratica l'espressione booleana è un **OR logico** (simbolo +) tra 3 termini (in questo caso) → basta che uno di quei 3 termini valga, per avere come uscita 1. Quindi ci saranno 3 termini di una somma logica; i singoli termini sono il prodotto logico (**AND logico**) delle 3 variabili, perché le 3 variabili devono avere esattamente quella configurazione per produrre 1 in output. Basta che uno di questi termini sia vero per avere uscita 1. per tutti gli altri input invece ho uscita 0.

Il segnale negato corrisponde a una configurazione in cui ho il valore = 0 su quella variabile!!!

→ il primo termine è il prodotto tra x_1 negato, x_2 negato e x_3 vero; e gli altri 2 termini...

Se ho il segnale 0 applicato su x_1 e x_2 ; e il segnale 1 su x_3 → allora ho in uscita 1.

Io ho 3 porte AND il cui risultato va dentro ad una porta OR e da il risultato finale (immagino sia 1 il risultato finale); sono riuscito a passare da tabella a funzione e posso fare viceversa.

Forma Prodotto di Somme (PS):

Il contrario di quella appena vista. quindi prendo gli 0 come output desiderato ed una variabile è normale se ha 0; è negata se ha output 1. Sono 2 forme logicamente equivalenti.

→ si usa l'una o l'altra in base a quanta prevalenza di 1 o 0 ho nel output finale; anche in base al costo, una potrebbe richiedere più porte logiche o fan in... di solito andiamo a scegliere sempre quella più conveniente, quella che richiede meno fan in e meno porte logiche.

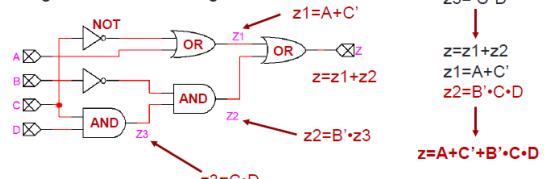
Esempio di analisi:

Per risolvere: do un nome a tutte le uscite di ogni porta logica, non delle negazioni (NOT) perché sono **porte logiche unarie**. All'uscita ottengo una espressione booleana con i nomi degli ingressi (che sono le uscite delle porte logiche precedenti).

Analisi:

1. nominando tutte le uscite dei gate logici
2. per sostituzione a partire dalle uscite si ottiene una funzione Booleana delle sole variabili di ingresso

Esercizio: Eseguire l'analisi del seguente schema



la rappresentazione prodotto di

somme è duale della rappresentazione somme di prodotti.

Teor. di Identità

- (T1) $X + 0 = X$

- (T1') $X \cdot 1 = X$

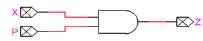
Teor. di Elementi nulli

- (T2) $X + 1 = 1$

- (T2') $X \cdot 0 = 0$

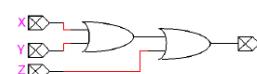
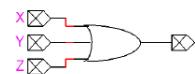
- sono molto utili nella sintesi di reti logiche: gli elementi nulli permettono di "lasciar passare" un segnale di ingresso in determinate condizioni

- es: progettare una rete logica che fornisca in uscita il valore di X se un pulsante P viene premuto altrimenti l'uscita valga sempre 0



Proprietà associativa

- (T7) $(X + Y) + Z = X + (Y + Z) = X + Y + Z$
- (T7') $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) = X \cdot Y \cdot Z$



Proprietà della combinazione

- (T10) $(X + Y) \cdot (X' + Y) = Y$
- (T10') $X \cdot Y + X' \cdot Y = Y$

Proprietà del consenso

- (T11) $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$
- (T11') $X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$

Teorema di De Morgan

- (T12) $(X + Y)' = (X' \cdot Y')$
- (T12') $(X \cdot Y)' = (X' + Y')$
- generalizzabile per n variabili



Dai teoremi dell'assorbimento o dalla proprietà distributiva:

$$XY' + Y = XY' + XY + Y = X + Y$$

$$XY' + Y = (X + Y)(Y' + Y) = X + Y$$

Principio di Dualità:

- ogni espressione algebrica presenta una forma duale ottenuta scambiando l'operatore OR con AND, la costante 0 con la costante 1 e mantenendo i letterali invariati.

- ogni proprietà vera per un'espressione è vera anche per la sua duale.

- il principio di dualità è indispensabile per trattare segnali attivi alti e segnali attivi bassi.



Idempotenza

- (T3) $X + X = X$

- (T3') $X \cdot X = X$

si usa per l'amplificazione dei segnali ed eliminazione disturbi

Involuzione

- (T4) $(X')' = X$



Complementarietà

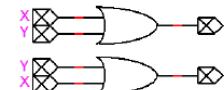
- (T5) $X + X' = 1$

- (T5') $X \cdot X' = 0$

Proprietà commutativa

- (T6) $X + Y = Y + X$

- (T6') $X \cdot Y = Y \cdot X$



Proprietà di assorbimento

- (T8) $X + X \cdot Y = X$

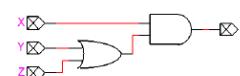
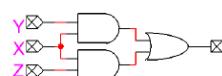
- (T8') $X \cdot (X + Y) = X$

permette di minimizzare il n. di gate

Proprietà distributiva

- (T9) $X \cdot Y + X \cdot Z = X \cdot (Y + Z)$

- (T9') $(X + Y) \cdot (X + Z) = X + Y \cdot Z$



diventa utile nel momento in cui sto minimizzando dei circuiti.

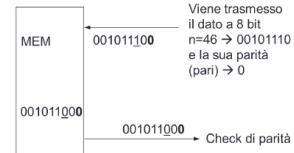
Parità

- I codici **rilevatori d'errori** sono codici in cui è possibile rilevare se sono stati commessi errori nella trasmissione
- Codici ridondanti:* in cui l'insieme dei simboli dell'alfabeto è minore dell'insieme di configurazioni rappresentabili col codice
- Codici con **bit di parità**: alla codifica binaria si aggiunge un bit di parità (codice ridondante in quanto usa 1 bit in più del necessario)

Simboli alfabeto	cod. Binaria	cod. Binaria con parità pari
0	000	000 0
1	001	001 1
2	010	010 1
3	011	011 0
4	100	100 1
5	101	101 0
6	110	110 0
7	111	111 1

- Ad ogni simbolo dell'alfabeto corrisponde una configurazione a parità pari.
- Le configurazioni a parità dispari non codificano alcun simbolo dell'alfabeto.
- Se viene rilevata una configurazione a parità dispari significa che si è verificato un errore che ha alterato un numero dispari di bit (1, 3, 5, ...).

- parità pari** rende pari il numero di 1 presenti nella parola (vale 1 se ci sono un n. dispari di 1)
- parità dispari:** il contrario
- I codici di parità rilevano la presenza di un numero dispari di errori (e quindi di errori singoli)
- es.** valore definito con 8 bit 11001011
con 9 bit con parità (pari) 110010111



- Supponiamo un errore di trasmissione durante la scrittura in memoria così che il numero memorizzato sia 001011000.
- Quando il dato viene riletto ed utilizzato viene fatto il check di parità e si verifica che quel numero non è ammissibile per la codifica binaria con parità pari perché la somma dei bit a 1 è dispari.
- Quindi viene rilevato un errore.

Esercizi fine capitolo:

piccolo archetto quando si deve attraversale un filo (per dire che un filo "scavalca" l'altro filo; non c'è connessione)
per il NOT, puoi metterlo in mezzo al filo dell'input; o un pallino prima dell'ingresso nell'operatore

Esercizio 1:

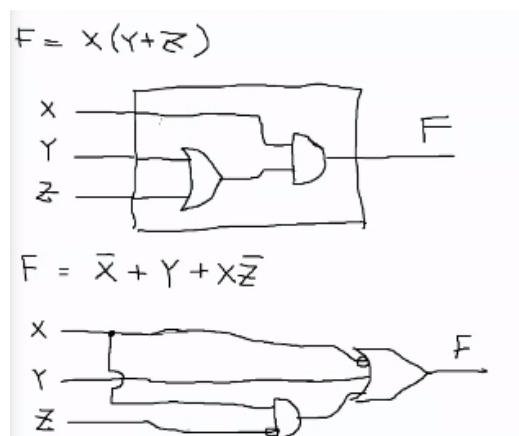
▼ traccia

Date le seguenti funzioni logiche ricavare le corrispondenti reti logiche realizzate utilizzando solo gate elementari AND, OR e NOT

$$F = X(Y + Z)$$

$$F = \bar{X} + Y + X\bar{Z}$$

▼ soluzione prof



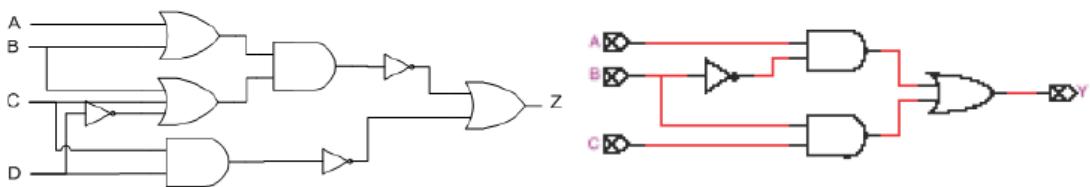
meno porte logiche utilizziamo meglio
è

anche la matrice di connessione è da minimizzare → il fan in e il fan out

Esercizio 2:

▼ traccia

Date le seguenti reti logiche determinare le tabella di verità e le funzioni logiche corrispondenti

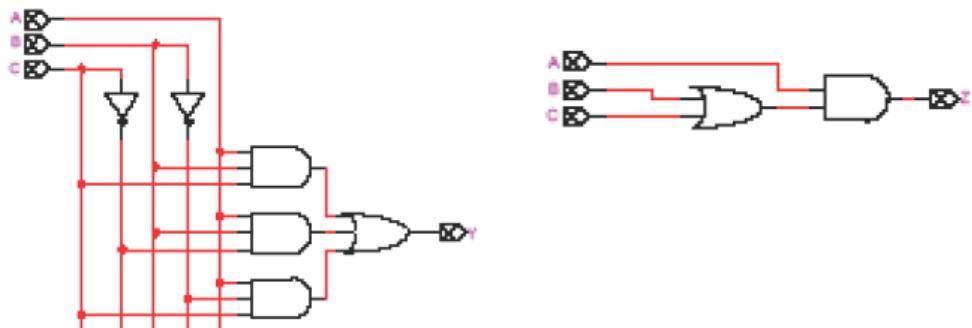


▼ soluzione mia

Esercizio 3:

▼ traccia

Date le reti di figura ricavare le tabelle di verità, le funzioni logiche in forma algebrica e dimostrare, facendo uso dei teoremi dell'algebra di Boole, che risultano logicamente equivalenti.



▼ soluzione mia

se nella mia F avessi un termine che è sempre 1 o sempre 0 (tipo $D \cdot D'$ oppure $D + D'$) e in base al loro valore e all'espressione che ho, posso semplificare il teorema.
esempio: $F = (D \cdot D') + (A \cdot B) = A \cdot B$ perché $D \cdot D' = 0$ e $0 + (A \cdot B)$ è sempre $= A \cdot B$ quindi D si può omettere dalla tabella della verità.

viceversa con $D+D'$ è sempre = 1 ; quindi $A*B + (D+D')$ è sempre = a 1 perchè $(A*B) + 1 = 1$

2 tabelle di verità uguali, i due circuiti implementano la stessa logica; ma uno dei due costa molto di più per l'implementazione.

l'ultimo passo di progettazione di una rete logica è sempre la minimizzazione!!!!
i due circuiti sono logicamente equivalenti

Esercizio 4:

▼ traccia

Ricavare le tabelle di verità delle seguenti espressioni

- $Z = W'X + Y'Z' + X'Z + Y$
- $Z = W + X'(Y' + Z)$
- $Z = WX + Y(Z' + X) + Z(X' + Y')$
- $Z = ABC + (A' + B' + C)C'$

▼ soluzione mia

Esercizio 5:

▼ traccia

Ricavare le tabelle di verità e semplificare le seguenti funzioni. Indicare anche il teorema utilizzato per ciascun passaggio della semplificazione:

- $Y = (A+B)(A+BC) + A'B' + A'C'$
- $Y = ABC + ABC' + A'BD + ABD + A'D$
- $F = (X+Y+W')(X+Y+W)(X+Y'+W)(X'+Y'+W)$
- $Y = A'C(A'BD)' + A'BC'D' + AB'C$
- $Y = (A'+B)(A+B+D)D'$
- $Y = A'B'C'D + A'B'CD + A'BC'D + AB'C'D$
- $W = X'Y + X'Y'Z$

Esercizio 6:

▼ traccia

Una assicurazione è disposta a fornire una assicurazione nei seguenti casi: il contraente è maschio e ha meno di 30 anni oppure ha più di 30 anni ed ha figli; il contraente ha più di 30 anni, non ha figli e, o è maschio o è sposato; il contraente ha più di 30 anni, non ha figli e non è sposato.

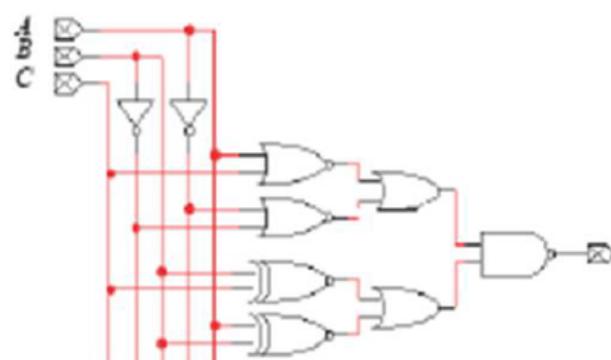
Valutazione: una donna con figlio non sposata e con meno di 30 anni può essere assicurata?

non si capisce dal testo, aspetto la soluzione del prof

Esercizio 7:

▼ traccia

Ricavare la funzione logica in forma algebrica e semplificare applicando i teoremi dell'algebra booleana. Disegnare il diagramma della rete semplificata.

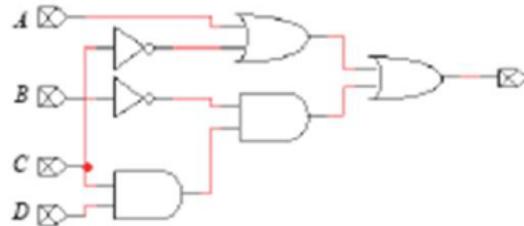


▼ soluzione mia

Esercizio 8:

▼ traccia

Ricavare la funzione logica in forma algebrica e semplificare applicando i teoremi dell'algebra booleana. Disegnare il diagramma della rete semplificata.



▼ soluzione mia

Indice

Sintesi reti logiche

Forma canonica:

- La più immediata forma di rappresentazione delle funzioni Booleane (e delle tabelle di verità) è la rappresentazione con una espressione in FORMA CANONICA

FORMA CANONICA SP (SOMMA DI PRODOTTI)

Teorema: una funzione di n variabili può essere espressa in un solo modo come somme di prodotti di n variabili (chiamati MINTERMINI)

- MINTERMINE è il prodotto logico di n letterali ognuno dei quali compare in forma vera o complementata, ma mai in entrambe.

- Da ogni tabella si deriva deterministicamente la forma SP, prendendo in OR tutti i mintermini corrispondenti alle righe in cui l'uscita vale 1, in cui ogni variabile è in forma diretta se nella colonna appare il valore 1 ed in forma complementata se appare il valore 0.

$$R = r'ab + r'a'b + rab' + rab$$

- La forma canonica può essere ottenuta per qualsiasi rete logica combinatoria
- Indipendentemente dalla complessità della rete logica da realizzare, la rete logica ottenuta dalla forma canonica è una rete molto veloce, in quanto composta da soli due livelli e mezzo (livello dei NOT)

2 livelli e mezzo perchè: livello degli AND, livello degli OR e il livello dei NOT che viene considerato mezzo.

Usando il teorema di De Morgan si può provare l'esistenza di un'altra forma canonica

FORMA CANONICA PS (PRODOTTO DI SOMME)

Teor: una funzione di n variabili può essere espressa in un solo modo come prodotto di somme di n variabili (chiamate MAXTERMINI)

- MAXTERMINE è la somma logica di n letterali ognuno dei quali compare in forma vera o complementata, ma mai in entrambe.
- Da ogni tabella si deriva deterministicamente la forma PS, prendendo in AND tutti i maxtermini corrispondenti alle righe in cui l'uscita vale 0, in cui ogni variabile è in forma diretta se nella colonna appare il valore 0 ed in forma complementata se appare il valore 1.

- Dalla tabella precedente:

$$R = (r+a+b)(r+a+b')(r+a'+b)(r'+a+b)$$

r	a	b	S	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Si dimostra essere equivalente alle precedenti

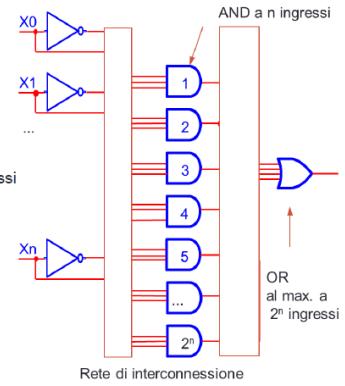
Sintesi forme canoniche

tanti NOT quanti sono i segnali di ingresso: massimo **n NOT**

al più 2^n porte AND;
il fan in peggiore è di ordine n.

- SOMME DI PRODOTTI

- una funzione combinatoria di n ingressi sintetizzata in forma canonica contiene al più n gate NOT, 2^n AND al più a n ingressi e 1 OR al più a 2^n ingressi
- Similmente per la forma canonica PS (scambiando OR e AND)



Codice gray:

codice con **Hamming 1**(unitario) → tra due configurazioni adiacenti, varia soltanto un bit; nel binario non funziona questa regola

il nostro sistema ha 3 uscite; devo creare tante reti quanti sono i segnali di uscita.(una rete con x uscite può diventare tante reti con 1 uscita)

lavoriamo con $g_0 \rightarrow$ guardo dove ha 1.

la funzione che produce in uscita il segnale di g_0 è quella in immagine.

quindi quando ho più segnali di uscita, **si fa la sintesi focalizzandosi ad una colonna alla volta.**

Sintesi in forma **canonica PS** di $g_0 \rightarrow (b_2+b_1+b_0)(b_2+b_1'+b_0')(b_2'+b_1+b_0)(b_2'+b_1'+b_0')$
se dovessi creare la rete in forma **canonica SP** → sarebbe la stessa ma con gli OR e gli AND scambiati.

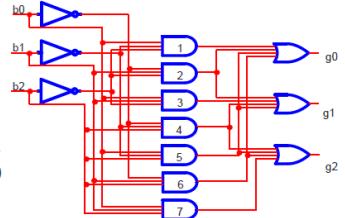
al contrario di come abbiamo considerato fino ad adesso, le espressioni canoniche non sono sempre uniche:

Esercizio : Progettare la rete logica di conversione di codice binario in codice GRAY a 3 ingressi

Il codice Gray è un codice a distanza di Hamming unitaria

BINARIO	GRAY
b ₂ ,b ₁ ,b ₀	g ₂ ,g ₁ ,g ₀
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

$$\begin{aligned} g_0 &= b_2'b_1'b_0 + b_2'b_1b_0' + b_2b_1'b_0 + b_2b_1b_0' \\ g_1 &= b_2'b_1b_0' + b_2'b_1b_0 + b_2b_1'b_0' + b_2b_1'b_0 \\ g_2 &= b_2b_1'b_0' + b_2b_1'b_0 + b_2b_1b_0' + b_2b_1b_0 \end{aligned}$$



in immagine → livello NOT, livello AND e livello OR

- Funzioni non completamente specificate se le uscite hanno condizioni di INDIFFERENZA
- Esistono alcune delle 2^n configurazioni non definite: il dominio è un sottoinsieme del dominio delle 2^n configurazioni
- Una espressione definisce una funzione non completamente specificata solo limitatamente al suo dominio.
- Le espressioni canoniche SP o PS di una funzione non completamente specificata NON SONO UNICHE
- Gli schemi logici che rappresentano la struttura delle reti logiche devono essere completamente specificate dal progettista.

Esempio:

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	-

Forme canoniche SP equivalenti
 $Z_1 = A'B + AB'$ $Z_2 = A'B + AB' + AB$

Z2 si semplifica:
 per idempotenza
 per distributiva
 per complementarietà
 per elementi nulli e commutativa

$$\begin{aligned} Z_2 &= A'B + AB + AB + AB' \\ Z_2 &= (A'+A)B + A(B+B') \\ Z_2 &= 1B + 1A \\ Z_2 &= A + B \end{aligned}$$

Ci sono 2 espressioni differenti perché abbiamo un'indifferenza; io posso comunque rendermi conto di questa uguaglianza con delle semplificazioni

Sintesi e minimizzazione:

Sintesi di reti logiche combinatorie:

- descrizione mediante tabella della verità
- sintesi della espressione canonica SP o PS
- corrispondenza 1 a 1 con uno schema logico

Tale sintesi non è minimizzata: può esserlo in termini di:

- area minima, costo minimo
- minimo n. di gate
- minimo n. di livelli
- minimo n. di interconnessioni
- fan in e fan out limitato
- necessità di magazzino...

Normalmente una rete logica si dice in forma minima per indicare il minor numero di livelli e, a parità di livelli, il minor numero di gate e di ingressi dei gate

Tecniche di minimizzazione:

- minimizzazione con manipolazione algebrica
- minimizzazione con algoritmi CAD o software appositi (es. Logisim)
- minimizzazione manuale (k-mappe)

questa sintesi non è sempre minimizzata soprattutto se ci sono indifferenze.
 vari trade off → non posso più aggiungere fan in quindi cambio altro...

MAPPE:

corrispondenza diretta tra la tabella di verità e mappa.

sottoinsieme che corrisponde a righe e uno che corrisponde a colonne; ogni cella è la combinazione dei due sottoinsiemi
 → unica configurazione dell'input.

la tabella utilizza una rappresentazione binaria; elenca le rappresentazioni in formato binario, una riga dopo l'altra.

le mappe di **Karnaugh** invece usano rappresentazione **GRAY**. perchè la mappa è un modo intuitivo di raggruppare gli 1; quindi posso dare rappresentazioni più

11110000

- Mappa: Rappresentazione più compatta della tabella di verità

E' una rappresentazione matriciale della tabella in cui le righe indicano tutte le possibili configurazioni di un sottoinsieme delle variabili di ingresso e le colonne tutte le configurazioni delle variabili rimanenti, il valore nelle celle indica il valore dell'uscita nella configurazione corrispondente

		x_3x_2	00	01	10	11
		x_1x_0	00	01	10	11
			1	0	1	-
		01	0	1	1	-
		10	1	1	1	-
		11	1	1	-	-

- OGNI CELLA CORRISPONDE AD UNA CONFIGURAZIONE DELLE VARIABILI

Mappe di Karnaugh: Mappe in cui le configurazioni successive in ogni lato sono ADIACENTI

- due configurazioni sono adiacenti (logicamente) se differiscono di un solo bit
- due celle sono adiacenti (geometricamente) se corrispondono a configurazioni adiacenti

Nelle Mappe di Karnaugh adiacenza geometrica e logica COINCIDONO

compatte, raggruppando i termini in adiacenti.

nell'immagine a destra è in binario ma dopo è in GRAY.

Mappe di Karnaugh:

K-mappa a 2 variabili

x_1	0	1
0	0	0
1	1	1

K-mappa a 3 variabili

x_2	00	01	11	10
0	0	0	-	1
1	1	1	1	1

K-mappa a 4 variabili

x_3	x_2	00	01	11	10
00	1	0	-	1	
01	0	1	-	1	
11	1	1	-	1	
10	1	1	-	-	

Criteri geometrici di adiacenza:

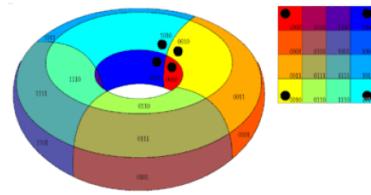
- 1 lato in comune
- un'estremità di colonna
- un'estremità di riga
- stessa posizione in sottomatrici adiacenti

K-mappa a 5 variabili

$X_4=0$ $X_4=1$

Le mappe vanno viste come «arrotolate» su se stesse.

- La prima riga risulta «adiacente» all'ultima riga. Stessa cosa per le colonne.
- Una visualizzazione 3d delle mappe che mette in risalto tale adiacenza è quella rappresentata in figura



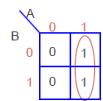
L'**adiacenza logica** si ha quando varia 1 bit da un elemento al successivo. Quella di **Karnaugh** è **adiacenza geometrica**: è quello che vedo con i miei occhi adiacente; che però corrisponde a quella logica.

le celle che sono adiacenti tra di loro a nord sud ovest est. quella logica ci dice che per passare da una cella all'altra **ho variato solo 1 bit**. Le celle che stanno nei bordi sono adiacenti a quelle sopra, destra e sinistra; ma è da considerare un'**adiacenza continua** → bypassa il bordo (tipo snake, o il famoso effetto pacman), o in egual modo le colonne. L'adiacenza geometrica(e quindi quella logica) è in tutte le direzioni. Nella prima immagine infatti non la rispetto perchè passo da 0100 a 1000 → **variato 2 bit**, in quella di **Karnaugh** invece si, ho righe adiacenti che corrispondono a codici adiacenti. I Bit più significativi (sinistra) stanno nelle righe; mentre gli altri nelle colonne.

- ogni casella della mappa è adiacente a caselle corrispondenti a mintermini (maxtermini) aventi distanza di Hamming unitaria dal mintermine (maxtermine) corrispondente alla casella considerata.

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

$$F = AB' + AB = A(B+B') = A \text{ proprietà distributiva}$$



la funzione vale 1 quando A vale 1 indipendentemente dal valore di B

- Nelle mappe due mintermini (o maxtermini) di distanza 1 sono adiacenti

x_1	x_2	00	01	11	10
0	1	1	1	0	
1	0	1	0	0	

$$Z = A'B'C' + A'B'C + A'BC + AB'C$$

idempotenza

$$Z = A'B'C' + A'B'C +$$

$$A'BC + A'B'C +$$

$$AB'C + A'B'C$$

$$Z = A'B'(C'+C) +$$

$$A'C(B'+B) +$$

$$B'C(A'+A)$$

$$Z = A'B' + A'C + B'C$$

Ogni cella rappresenta una configurazione di input: se il valore li dentro è 1, è un **mintermine**; ogni cella con 0 è un **maxtermine**. Io posso fare un raggruppamento di 1 adiacenti(se considero forma canonica normale) ed

Somma di 4 mintermini i quali utilizzano tutte e 3 le variabili → forma canonizzata completa; ma non è la minima quindi applico i teoremi...

equivale ad una minimizzazione!!

B non è necessario → solo quando A è a 1 la funzione è 1

Tabella di verità la quale ha 4 righe pari a 1 → perchè ho 4 termini con 3 lettere ognuno → per forza equivalgono a 4 righe che valgono 1.

Raggruppamenti triangolari:

- Si dice raggruppamento rettangolare di ordine p una parte di una mappa a n variabili costituita da 2^p elementi (con $p \leq n$) tali da avere $n-p$ coordinate uguali fra loro, e di far assumere alle restanti p coordinate tutte le possibili configurazioni.
- Ogni cella ha all'interno p celle adiacenti

ordine 0 1 cella
ordine 1 2 celle
ordine 2 4 celle

	X ₁ X ₀	no			
	00	01	11	10	
X ₂ X ₁	00	1	0	-	1
	01	0	1	-	1
	11	1	1	-	1
	10	1	1	-	-

- Un Raggruppamento Rettangolare (RR) nel quale la funzione assume sempre valore 1 si dice implice della funzione. In modo duale, un RR nel quale la funzione assume sempre valore 0 si dice implice della funzione.

- Un implice (implicato) corrisponde a un prodotto logico (somma logica) dei letterali delle sole variabili di ingresso che non cambiano valore, presi negati se la corrispondente variabile di ingresso vale 0 (1), non negati se tale variabile vale 1 (0).

Karnaugh ci dice che i raggruppamenti devono considerare l'adiacenza geometrica e devono essere una potenza di 2 → raggruppamenti di 2,4,8,...

- Un implice non contenuto in nessun implice di dimensioni maggiori prende il nome di implice primo.

- Si dice copertura degli 1 un insieme di implicati che contengono tutti gli 1 della funzione ed eventualmente indifferenze (copertura di 0 un insieme di implicati che contenga tutti gli 0 e al più indifferenze)

- implicante essenziale: un implice primo contenente almeno un mintermine non contenuto in nessun altro implice primo (cioè un implice primo che "copia" almeno un mintermine non coperto da altri).

- Ogni implice essenziale deve essere contenuto nella somma minima. Vale il duale per gli implicati



0	0	0	0
0	1	1	0
0	0	1	1
0	0	0	0

IMPL.
ESSENZIALE

IMPL. NON PRIMO

IMPL. PRIMO,
NON ESSENZIALE

devo coprire tutti gli 1!!

anche prendendone 1 da solo, ma più ne raggruppo, più minimizzo → infatti qui ho 3 raggruppamenti con 1 elemento in comune.

mi dice già che la mia funzione ha 3 mintermini perchè 3 raggruppamenti

- Una copertura di 1 individua una forma SP

$$Z = A'B'C'D + A'BCD +$$

$$ABC'D + ABCD$$

$$Z = A'BD(C+C') + ABD(C+C')$$

$$Z = (A'+A)BD$$

$$Z = BD$$

Z=BD	AB	CD	00	01	11	10
	00		0	0	0	0
	01		0	1	1	0
	11		0	1	-	0
	10		0	0	0	0

- Una copertura di 0 individua una forma PS

	AB	CD	00	01	11	10
	00		0	0	0	0
	01		0	1	1	0
	11		0	1	-	0
	10		0	0	0	0

RR DI ORDINE 3
corrispondente a D

RR DI ORDINE 3
corrispondente a B

Forme normali e minime:

Una espressione

- si dice **normale SP** se è data dalla somma di prodotti non necessariamente di n variabili
- si dice **normale PS** se è data dal prodotto di somme non necessariamente di n variabili

Una espressione normale è equivalente alla forma canonica ma minimizzata

SINTESI MINIMA (di costo minimo)

- con il minor numero di livelli
- minimo numero di gate (ad es. di prodotti da sommare) --- forma normale
- minimo numero di connessioni
- l'espressione minima normale e non ridondante si ottiene con una copertura usando il numero minimo di RR di ordine massimo (implicanti primi)
 - **ordine massimo** : minor numero di ingressi
 - **minimo numero di RR**: minimo numero di gate
 - forma normale **irridondante**: solo implicanti essenziali

• **Forma minima PS ed SP**

- sono diverse
- Potrebbe valer la pena valutarle entrambe specialmente con indifferenze

Reti a più uscite

- a volte è conveniente scegliere degli implicanti comuni anche se non primi o essenziali

		X ₃ X ₂	00	01	11	10
		X ₁ X ₀	00	01	11	10
X ₃ X ₂	X ₁ X ₀	00	0	0	1	1
		01	0	1	1	0
X ₃ X ₂	X ₁ X ₀	11	1	1	0	0
		10	1	1	0	0

$$Z = x_3x_1'x_0' + x_3x_2x_1' + x_2x_1'x_0 + x_3'x_2x_0 + x_3'x_1$$

		X ₃ X ₂	00	01	11	10
		X ₁ X ₀	00	01	11	10
X ₃ X ₂	X ₁ X ₀	00	0	0	1	1
		01	0	1	1	0
X ₃ X ₂	X ₁ X ₀	11	1	1	0	0
		10	1	1	0	0

$$Z = x_3x_1'x_0' + x_2x_1'x_0 + x_3'x_1$$

La mappa si usa per configurazioni con 5 o 6 variabili; oltre è difficile per un umano.

Meno raggruppamenti ho → meno fan in ho.

Può essere che ci sia un forma migliore tra PS e SP.

Si minimizza perchè la forma canonica richiede un fan in troppo alto e anche gate. Le intersezioni vanno fatte! è meglio di prendere l'elemento da solo! è più minimizzato! → **non riduce il numero di porte, ma riduce il numero di fan in al primo livello.** La **forma canonica** di una mappa di karneau vuol dire **prendere gli elementi senza raggruppamenti!!** con raggruppamenti è minimizzata. Le condizioni di indifferenza ci interessano solo quando dobbiamo fare le minimizzazioni; quindi i raggruppamenti; per la forma canonica e basta possiamo non prenderli.

controllare

Il fan in è il numero di ingressi per la porta o il livello logico; se consideriamo le forme ps o sp abbiamo sempre 2 livelli e mezzo → OR AND e NOT

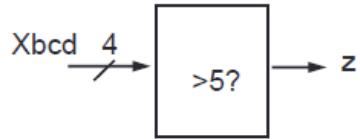
Tra 1 livello e l'altro c'è la rete di interconnessione, dove passano tutte le connessioni; quindi fan in è "l'input" in entrata del livello; che equivale al fan out del livello precedente importanti da semplificare, perchè semplificano la matrice di interconnessioni. Non c'è una fan più importante; si guarda il fan in generale → a parità di condizioni, meno fili ho meglio è. Per questo serve molto karneau, perchè **più raggruppamenti ho, meno fan in ho.**

Esercizi fine capitolo

Esercizio1:

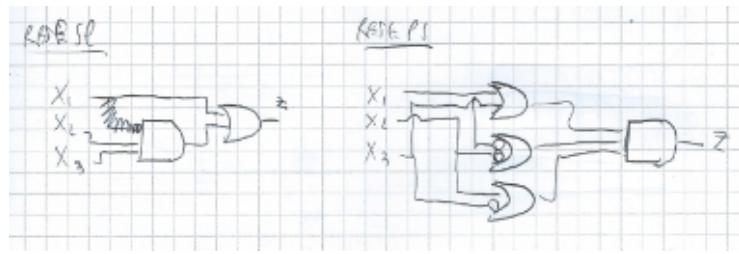
▼ traccia

Esercizio 1: Dato un numero in BCD progettare la rete logica che indichi se è maggiore di 5; quante sono le forme canoniche equivalenti? Perchè sono più di 1? Quale è la forma minima PS quale la SP?



▼ soluzione mia

<p>$\Rightarrow \text{ESR}(x_1, x_2) = \frac{x_1 + x_2}{2} \rightarrow \boxed{z} = \frac{x_1 + x_2}{2}$</p> <p>$\text{Zerowerte: } 4 \text{ bit} \rightarrow 2^4 \text{ (conf)} \quad x_1, x_2, x_3, x_4 \quad S_{(10)} = 0101_{(2)}$</p> <p>$P(z) = \text{effektiv 0.1} \Rightarrow S_{(10)} = 1001 \text{ , 0 effektiv von 0.1.}$</p>
<p>$\Rightarrow \begin{array}{ c c c c c } \hline & x_1 & x_2 & x_3 & x_4 & z \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 1 & 1 & 2 & 2 \\ \hline \end{array}$</p> <p>$\Rightarrow \text{Formel SP: } z = \bar{x}_1 x_2 x_3 \bar{x}_4 + \bar{x}_1 x_2 x_4 + x_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 \bar{x}_2 x_3 x_4$</p> <p>$\Rightarrow \text{Karnaugh Map: }$</p> <p>$\Rightarrow \text{Faktorisierung: } z = \bar{x}_1 x_2 x_3 + \bar{x}_1 x_2 x_4$</p>
<p>$\Rightarrow \begin{array}{ c c c c c } \hline & x_1 & x_2 & x_3 & x_4 & z \\ \hline 00 & 0 & 0 & 0 & 0 & 0 \\ 01 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 1 & 0 & 0 & 0 \\ 11 & 0 & 1 & 1 & 0 & 0 \\ \hline 00 & 0 & 0 & 0 & 1 & 1 \\ 01 & 0 & 0 & 1 & 1 & 1 \\ 10 & 0 & 1 & 0 & 0 & 1 \\ 11 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array} \Rightarrow z = x_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3$</p> <p>$\Rightarrow \text{Formel minima SP: } z = x_1 x_2 x_3 + (x_1 + x_2) x_1 x_2 x_3 = x_1 + x_2 x_3 (x_1 + x_2)$</p> <p>$\Rightarrow \text{Formel minima FS: } z = x_1 + x_2 x_3$</p>
<p>$\Rightarrow \begin{array}{ c c c c c } \hline & x_1 & x_2 & x_3 & x_4 & z \\ \hline 00 & 0 & 0 & 0 & 0 & 0 \\ 01 & 0 & 0 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 11 & 0 & 0 & 1 & 1 & 0 \\ \hline 00 & 0 & 0 & 0 & 0 & 0 \\ 01 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 1 & 0 & 0 & 0 \\ 11 & 0 & 1 & 1 & 0 & 0 \\ \hline \end{array} \Rightarrow z = \bar{x}_1 \bar{x}_2 \bar{x}_3 (\bar{x}_4 + x_2) (\bar{x}_4 + x_3) (\bar{x}_4 + x_2 + x_3)$</p> <p>$\Rightarrow \text{Formel minima FS: } z = \bar{x}_1 \bar{x}_2 \bar{x}_3 (\bar{x}_4 + x_2) (\bar{x}_4 + x_3) (\bar{x}_4 + x_2 + x_3)$</p>



▼ soluzione prof

x_3	x_2	x_1	x_0	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

$Z_{SP} = \bar{x}_3 x_2 x_1 \bar{x}_0 + \bar{x}_3 x_2 x_1 x_0 + x_3 \bar{x}_2 \bar{x}_1 \bar{x}_0 + x_3 \bar{x}_2 x_1 x_0$
 $Z_{SP_m} = x_3 + x_2 x_1$

x_1	x_0
$x_3 x_2$	00 01 11 10
00	0 0 0 0
01	0 0 1 1
11	1 1 1 1
10	1 1 0 0

▼ appunti

rete logica(scatolotto) in cui definiamo una funzionalità di alto livello → il numero deve essere maggiore di 5; utilizziamo un segnale di ingresso a 4 bit e un segnale di uscita.

la rappresentazione BCD richiede 4 bit. l'uscita z

fino a 5 sono 0; da 6 a 9 sono 1 e le altre non hanno valore e mettiamo condizione di indifferenza.

forme canoniche? perchè più di 1? perchè ci sono le condizioni di indifferenza, che possono essere trattati come 0 o 1 → in base a come le considero, posso avere forme che sono tutte valide, ma non determinano in maniera univoca la mia forma

Esercizio2:

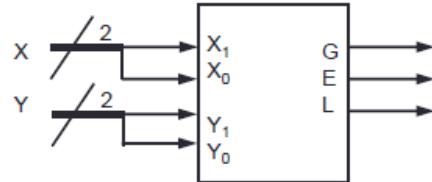
▼ traccia

Esercizio 2: Progettare un comparatore a 2 bit, ossia una rete logica che indichi quale dei due operandi a 2 bit è maggiore uguale o minore dell'altro.

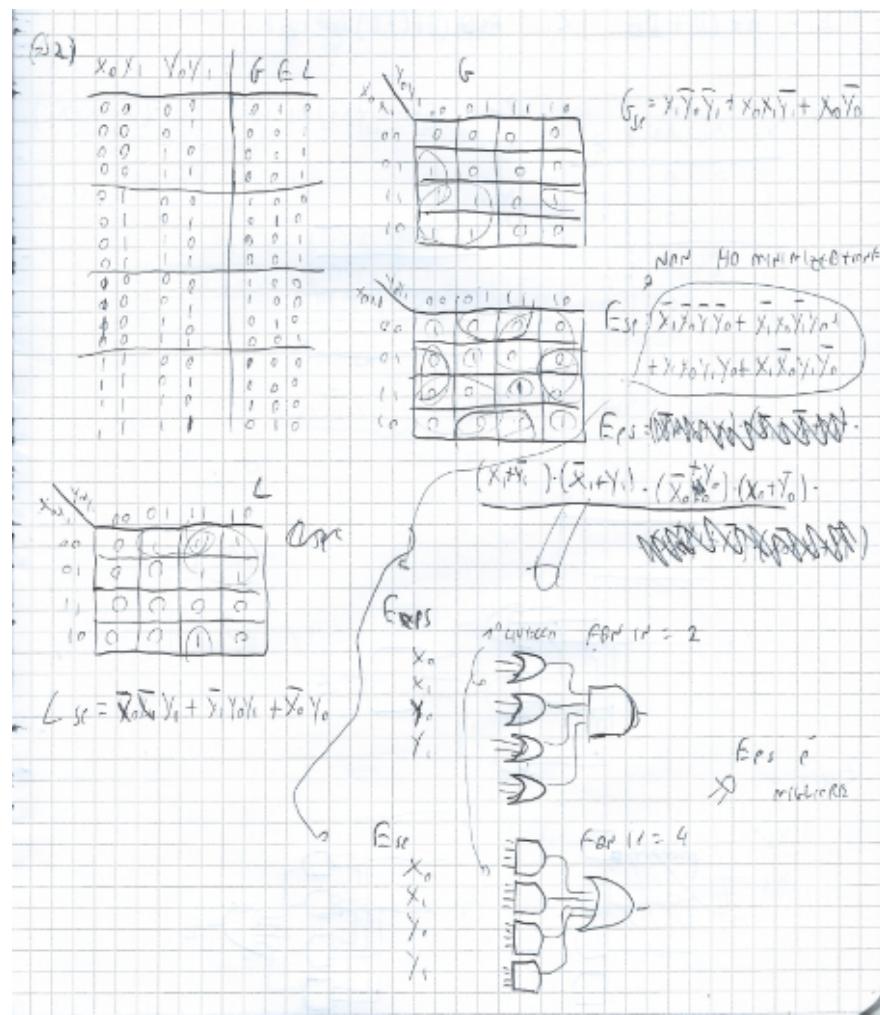
Descrizione a parole:

```
if (X>Y) {G=1,E=0,L=0;}
else if (X<Y){G=0,E=0,L=1;}
else {G=0,E=1,L=0;}
```

- 1) tabella della verità
- 2) mappe di Karnaugh
- 3) minimizzazione



▼ soluzione mia



▼ soluzione prof

▼ appunti

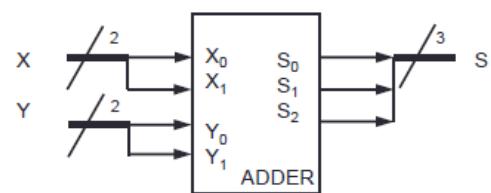
3 uscite → 3 mappe di karneau. una per uscita → 3 reti di uscita.
 nella **E** conviene raggruppare gli 0; perchè gli 1 non sono raggruppabili in gruppi , quindi verrebbe una forma lunga.
 quindi con **PS** ho 4 **OR** con 2 **fan in** raggruppati in 1 **AND**.
 con la **SP** ho 4 **AND** con 4 fan in raggruppati in 1 **OR**
 → ho semplificato usando la PS → prendendo gli 0

Esercizio3/4/5: non ho capito come fare

▼ traccia

Esercizio 3: progettare il FULL ADDER ossia il sommatore ad 1 bit con riporto di ingresso e di uscita (nella forma SP).

Esercizio 4: Progettare un sommatore a 2 bit di ingresso e 3 di uscita



Esercizio 5: Modificare l'esercizio 4 con anche il riporto di ingresso. Che differenza c'è in termini di gate e di ritardo rispetto a quello dell'esercizio 3? Come progettare un sommatore modulare usando un FULL ADDER?

ADB

▼ soluzione mia

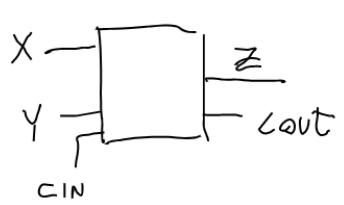
▼ soluzione prof

▼ appunti 3

sommatore ad 1 bit con riporto di ingresso e di uscita → vuol dire che abbiamo 2 input **x,y** e un segnale di riporto in ingresso **cin** e uno in uscita **cout**

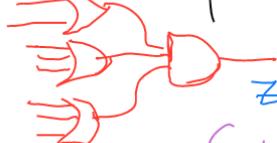
la mappa di karneau di Z non può essere minimizzata; che è uguale alla forma canonica.

le intersezioni vanno fatte! è meglio di prendere l'elemento da solo! è più minimizzato!



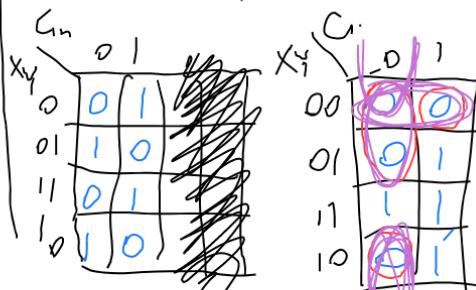
X	Y	C _{in}	Z	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

X	Y	C _{in}	Z	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$Z_{sp} = \bar{X}\bar{Y}C_{in} + \bar{X}Y\bar{C}_{in} + X\bar{Y}\bar{C}_{in} + XYC_{in}$$

$$Cout = (x+y+C_{in})(x+v+\bar{C}_{in})(x+\bar{y}+C_{in}) \cdot PS \cdot (\bar{x}+y+C_{in})$$



$$Cout = (x+c)(x+y+\bar{c})(x+\bar{y}+c)$$

$$C_{out} = (x+c)(x+v)(y+c) \cdot Cout$$

▼ appunti 4

l'immagine è quella nella traccia.

$X_1 X_0$	$Y_1 Y_0$	$S_2 S_1 S_0$	$X_1 Y_0$	$Y_1 Y_0$
0 0	0 0	0 0 0	0 0 0 0	0 0 0 0
0 0	0 1	0 0 1	0 0 0 0	0 1 1 0
0 0	1 0	0 1 0	0 0 1 0	0 1 1 0
0 0	1 1	0 1 1	0 1 1 1	0 1 1 0
0 1	0 0	0 0 1	0 1 1 1	0 0 0 0
0 1	0 1	0 1 0	0 1 1 1	0 1 1 0
0 1	1 0	0 1 1	0 1 1 1	0 1 1 0
0 1	1 1	1 0 0	1 0 0 0	0 0 0 0
1 0	0 0	0 1 0	0 1 0 0	0 0 0 0
1 0	0 1	0 1 1	0 1 0 0	0 1 1 0
1 0	1 0	1 0 0	1 0 0 0	0 1 1 0
1 0	1 1	1 0 1	1 0 1 1	0 1 1 0
1 1	0 0	0 1 1	0 1 1 1	0 0 0 0
1 1	0 1	1 0 0	1 0 0 0	0 1 1 0
1 1	1 0	1 0 1	1 0 1 1	0 1 1 0
1 1	1 1	1 1 0	1 1 1 0	0 1 1 0

$S_2 = \bar{Y}_1 X_1 + X_1 X_0 Y_0 + Y_1 Y_0 X_0$

$S_1 = \bar{X}_1 \bar{X}_0 Y_1 + \bar{X}_1 Y_1 \bar{Y}_0 + \bar{X}_1 \bar{X}_0 \bar{Y}_1 Y_0 + X_0 Y_0 Y_1 + X_1 \bar{Y}_1 \bar{Y}_0 + X_1 \bar{X}_0 \bar{Y}_1$

esercizio6:

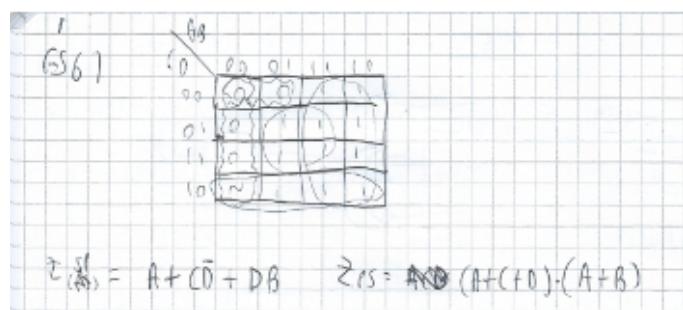
▼ traccia

Esercizio 6: trovare la forma canonica e minima
SP e PS

utile dei calcolatori

	AB	00	01	11	10
CD	00	0	0	1	1
	01	0	1	1	1
	11	0	1	1	1
	10	-	1	1	1

▼ soluzione mia



▼ soluzione prof

	AB	CD	00	01	11	10
CD	00	00	0	0	1	1
00	01	0	1	1	1	1
01	11	0	1	1	1	1
11	10	-	1	1	1	1

$Z_{SP_C} = AB\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + AB\bar{C}D + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + A\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D}$
 $Z_{PS} = (\bar{A} + B + C + D)(A + \bar{B} + C + D)(\bar{A} + B + C + \bar{D})(A + B + \bar{C} + D)$
 $Z_{SP_K} = A + BD + C\bar{D}$
 $Z_{PS_K} = (C + D + A)(A + B)$

▼ appunti

forma canonica = senza raggruppamenti

quindi si vede subito che è molto complessa la forma sp canonica; troppi AND ed ingressi...

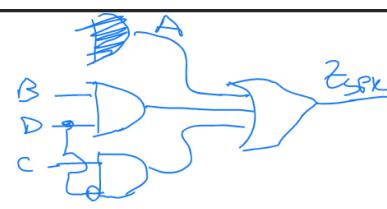
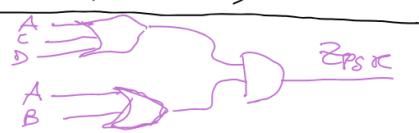
le condizioni di indifferenza ci interessano solo quando dobbiamo fare le minimizzazioni; quindi i raggruppamenti; per la forma canonica e basta possiamo non prenderli.

quale delle due forme preferiamo? no perchè hanno circa gli stessi numeri di fan in e operatori.

se proprio potresti preferire la forma ps; perchè nella sp usi un NOT; ma è poca differenza; quindi si, sarebbe meglio la ps, ma posso considerarle quasi equivalenti

	AB	CD	00	01	11	10
CD	00	00	0	0	1	1
00	01	0	1	1	1	1
01	11	0	1	1	1	1
11	10	-	1	1	1	1

$Z_{SP_C} = AB\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + AB\bar{C}D + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + A\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D}$
 $Z_{PS} = (\bar{A} + B + C + D)(A + \bar{B} + C + D)(\bar{A} + B + C + \bar{D})(A + B + \bar{C} + D)$
 $Z_{SP_K} = A + BD + C\bar{D}$
 $Z_{PS_K} = (C + D + A)(A + B)$

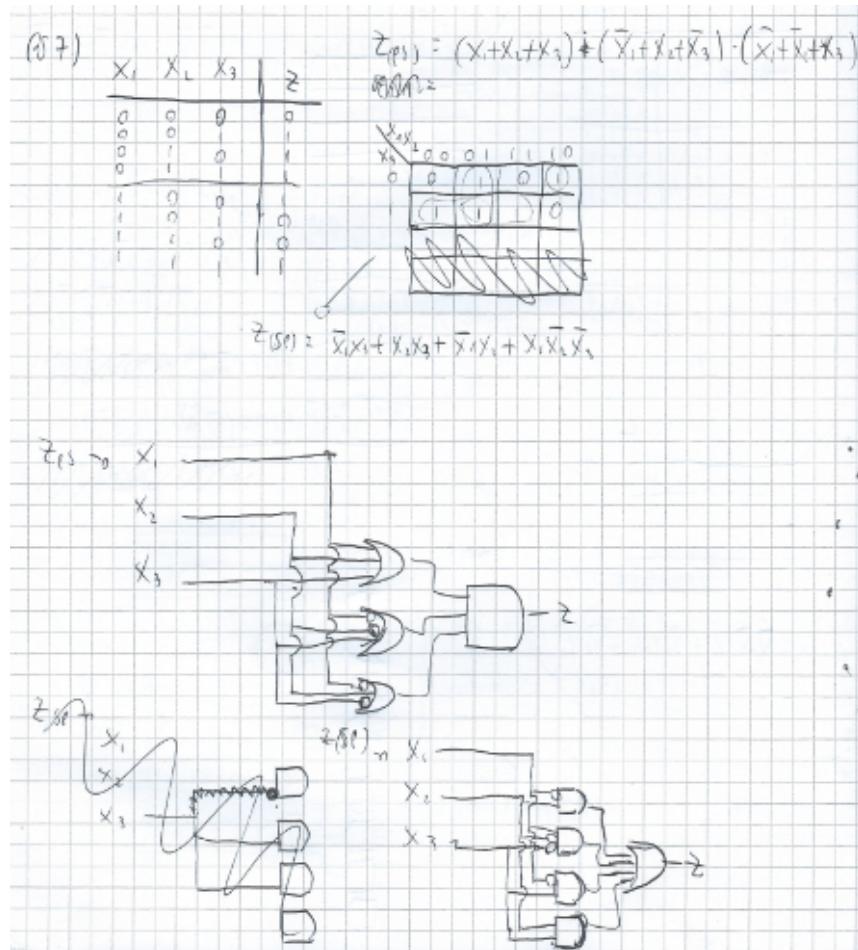



esercizio7:

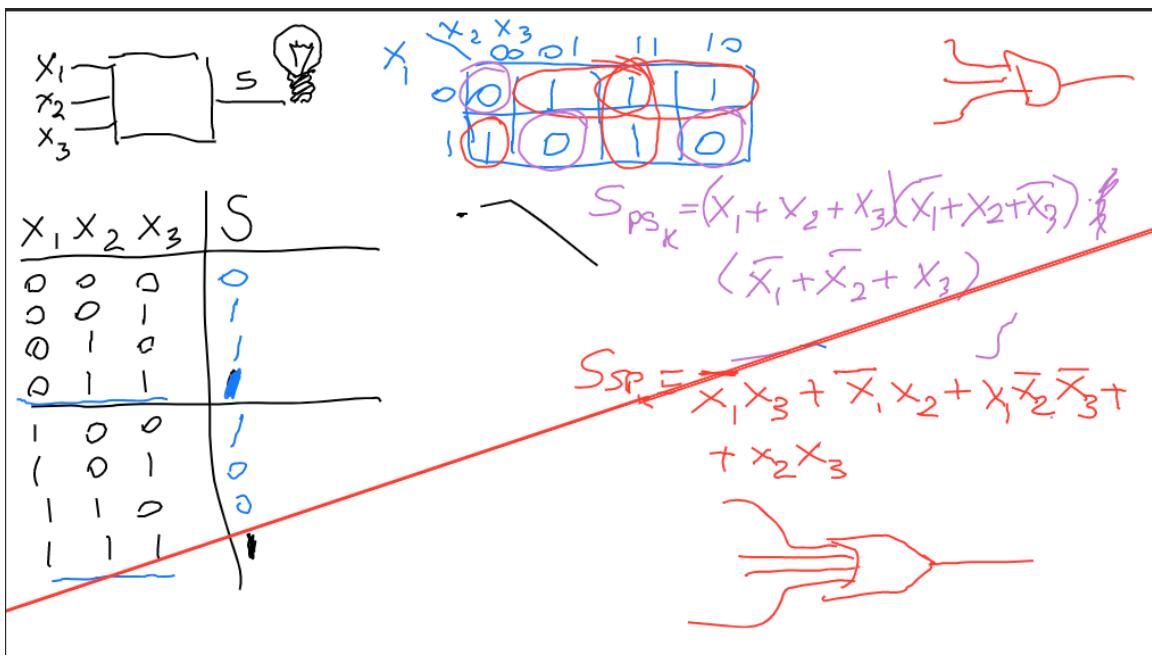
▼ traccia

Esercizio 7: Una lampadina può essere accesa o spenta da 3 interruttori X_1 , X_2 e X_3 : però viene accesa solo se sono ON un numero dispari di interruttori, oppure se X_2 e X_3 sono contemporaneamente ON. Progettare la rete logica corrispondente.

▼ soluzione mia



▼ soluzione prof



▼ appunti

il fan in è il numero di ingressi per la porta o il livello logico;

se consideriamo le forme ps o sp abbiamo sempre 2 livelli e mezzo → OR AND e NOT

tra 1 livello e l'altro c'è la rete di interconnessione, dove passano tutte le connessioni; quindi fan in è "l'input" in entrata del livello; che equivale al fan out del livello precedente

importanti da semplificare, perchè semplificano la matrice di interconnessioni.

non c'è una fan più importante; si guarda il fan in generale → a parità di condizioni, meno fili ho meglio è.

per questo serve molto karneau, perchè più raggruppamenti ho, meno fan in ho.

esercizio8:

▼ traccia

Esercizio 8: Tre interruzioni possono arrivare alla CPU anche contemporaneamente ma devono essere servite con una priorità: IR_1, IR_2 e IR_3 in ordine di priorità decrescente, nel senso che IR_1 è la più prioritaria. Progettare la rete che dà la richiesta di interruzione INT alla CPU e abilita le tre interruzioni IS_1, IS_2, IS_3 .

▼ soluzione mia

$I_{K_1} I_{K_2} I_{K_3}$	I_S_1	I_S_2	I_S_3
0 0 0	0	0	0
0 0 1	0	0	1
0 1 0	0	1	0
0 1 1	0	1	1
1 0 0	1	0	0
1 0 1	1	0	0
1 1 0	1	0	0
1 1 1	1	0	0

$I_{K_1} I_{K_2} I_{K_3}$	I_S_1	I_S_2	I_S_3
0 0 0	0	0	0
0 0 1	0	0	1
0 1 0	0	1	0
0 1 1	0	1	1
1 0 0	1	0	0
1 0 1	1	0	0
1 1 0	1	0	0
1 1 1	1	0	0

$I_{K_1} I_{K_2} I_{K_3}$	I_S_1	I_S_2	I_S_3
0 0 0	0	0	0
0 0 1	0	0	1
0 1 0	0	1	0
0 1 1	0	1	1
1 0 0	1	0	0
1 0 1	1	0	0
1 1 0	1	0	0
1 1 1	1	0	0

▼ soluzione prof

▼ appunti

criterio della priorità → non ragioniamo con le funzioni algebriche; ma con una funzione di ordine:

il più prioritario è quello a sinistra.

non è una descrizione esaustiva come la tabella della verità, dobbiamo capirlo noi.

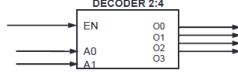
Indice

Componenti notevoli combinatori

Blocchi logici che si possono usare come componenti discreti: la mia rete può includere anche questi blocchi.

Demultiplexer/Decoder

- Il demultiplexer (decoder) realizza la funzione di smistare un singolo input in una delle n possibili uscite
- Formalmente il demultiplexer è una rete logica con 1 ingresso di dato, n segnali di controllo e 2^n uscite: l'uscita contrassegnata dall'indice pari alla configurazione dei segnali di controllo riceve l'ingresso, mentre le altre non sono abilitate (normalmente poste a livello logico 0).

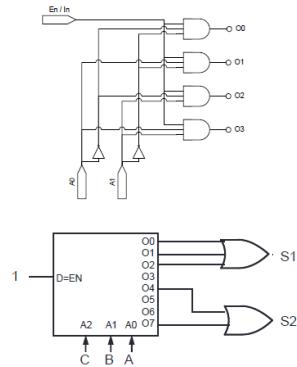


- Si dice anche decoder in quanto viene usato per decodificare un segnale binario (se si mantiene l'ingresso EN a 1).

EN = enable

decoder 2:4 → dati 2 segnali di controllo a 0 e a 1 riesce a produrre $2^2(4)$ segnali di uscita.

Può essere usato come generatore di mintermini:



Esempio:
realizzare la rete logica
 $S1 = A'B'C' + A'B'C + A'B'C'$
 $S2 = AB'C' + ABC$

questo circuito realizza la parte sinistra di una tabella di verità

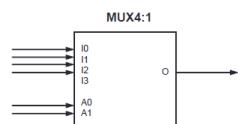
4 porte AND che corrispondono ad una configurazione per quei 2 segnali; poi ci aggiungo il segnale enable, ed essendo AND; se enable è 0 allora non funziona
→ **enable spegne o accende il circuito di fatto**

- prendo in ingresso le 3 uscite dal multiplexer e le metto dentro ad una porta OR → equivale al primo livello della forma SP??
- in generale manda alta l'uscita che corrisponde alla configurazione dei segnali di ingresso che soddisfa il criterio per quell'uscita

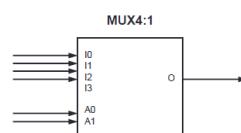
Multiplexer

Componente duale del demultiplexer

- Multiplexer:** è quel blocco logico che permette di deviare su un'unica uscita un segnale proveniente da uno tra n possibili ingressi.
- Formalmente:** è una rete logica avente 2^n ingressi di tipo *dato* e n ingressi di tipo *segnali di controllo* (o *indirizzo*) ed 1 uscita: in ogni istante il dato presente all'ingresso selezionato (mediante la configurazione dei segnali di controllo) viene riportato in uscita



- Sintesi attraverso la tabella della verità
 $O = I3A1A0 + I2A1A0' + I1A1'A0 + I0A1'A0'$
esegue la somma di tanti prodotti quanti gli ingressi (di dato), in cui ogni prodotto è un mintermine degli ingressi di controllo



A_1	A_0	I_3	I_2	I_1	I_0	O
0	0	x	x	x	0	0
0	0	x	x	x	1	1
0	1	x	x	0	x	0
0	1	x	x	1	x	1
1	0	x	0	x	x	0
1	0	x	1	x	x	1
1	1	0	x	x	x	0
1	1	1	x	x	x	1

Il segnale di ingresso rappresentato dai segnali di controllo è quello che passa.
vengono usati per progettare i BUS → le matrici di interconnessione tra CPU e altri componenti

Se io so che la mia funzione è data da una certa somma di mintermini; questi mintermini sono il risultato di un livello di porte AND.

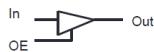
ne ho 4 m₀,m₂,m₆,m₇ → se le metto dentro ad un multiplexer, posso simulare una rete sp mettendo i bit di controllo;??? infatti ho messo 1 sui segnali I₀,I₂,... e non sugli altri.???

Il multiplexer è un **selettore**.

amplificatore tri-state:

L'uscita è uguale all'ingresso quando l'enable è a 1 è un **selettore o switch**.

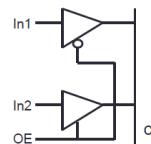
Amplificatore tri-state (1/3)



- Amplificatore tri-state: Generatore di segnale in terzo stato (Z)

In	OE	Out
X	0	Z
0	1	0
1	1	1

- L'uscita è uguale all'ingresso quando l'output enable (OE) è asserito;
- Si può pensare ad uno switch (interruttore)
- Si usa spesso per realizzare multiplexer distribuiti
- **Attenzione:** per evitare corto circuiti bisogna che in ogni istante solo un tri-state sia abilitato.



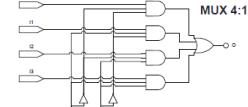
Half adder:

Sommatore di 2 bit che restituisce come uscite la somma di quei 2 bit ed eventualmente un riporto. S è un XOR; il bit di carry è un AND. Si usa al posto della funzione somma.

Full adder:

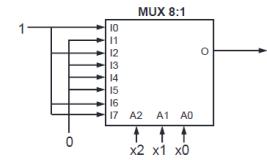
Da la possibilità di sommare il riporto che arriva da sinistra. E' un half adder con un riporto in ingresso; che viene attaccato al riporto in uscita.

- E' possibile costruire un multiplexer N:1 mettendo in cascata vari livelli di multiplexer più piccoli

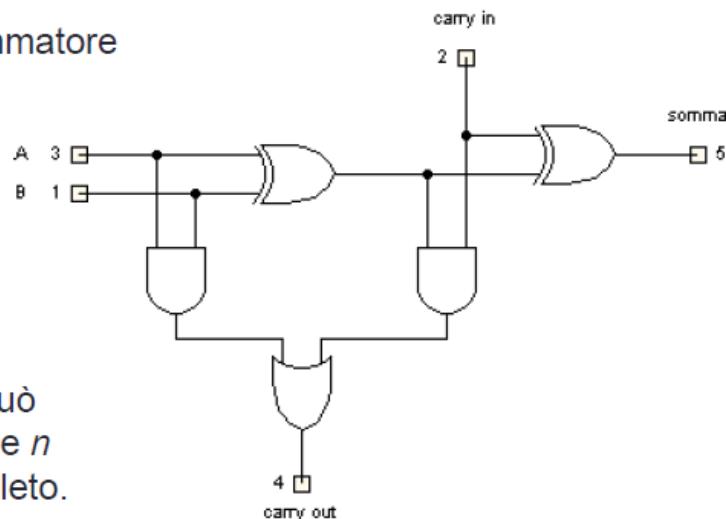


Il multiplexer non solo può essere usato come selettore, ma anche come «generatore di tabelle della verità»

- **Esercizio:**
realizzare
 $F(x_0, x_1, x_2) = m_0 + m_2 + m_6 + m_7$
(somma di 4 mintermini)



- La versione finale del sommatore completo a 1 bit è:



- Un sommatore a n bit si può ottenere replicando in serie n volte un sommatore completo.
- Il riporto (*carry out*) di un bit si usa come *carry in* del sommatore completo alla sua sinistra (cifra più significativa).

Replicando in serie un full adder, può creare un **sommatore ad n bit**:

ALU cuore della CPU:

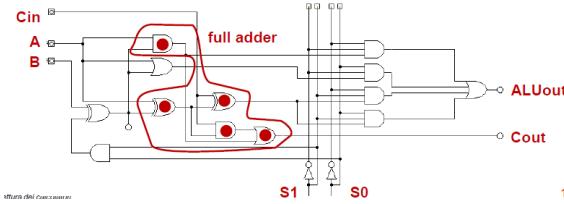
- La soluzione modulare ottenuta collegando in cascata N *full-adder* ha il vantaggio di essere chiara, facilmente replicabile, probabilmente ottimizzata dal punto di vista del numero di porte complessivo.
- In alternativa si può realizzare un sommatore generando direttamente le N uscite mediante forme minime SP (o PS). In tal caso si otterebbe una soluzione con molte più porte logiche, meno modulare ma più veloce, in quanto ogni cifra risulterebbe calcolabile tramite 2 livelli e mezzo.
- La soluzione mediante *full-adder* ha infatti un difetto, legato alla necessità di propagare il *carry* dal bit meno significativo a quello più significativo
- L'ALU (*arithmetic logic unit* o *unità aritmetico logica*) è un circuito combinatorio in grado di svolgere opportune operazioni aritmetiche e/o logiche su due operandi.
- L'operazione non è fissa ma è selezionabile mediante opportuni segnali.
- Normalmente una ALU fornisce in uscita il risultato della operazione e alcuni FLAG, che descrivono il risultato (negativo / zero) o identificano eventuali errori (es overflow) o situazioni di interesse (es. carry).
- L'elenco delle operazioni eseguite viene definita in fase di progetto e non è fissata a priori.

Full adder funziona già per la propagazione del segnale di carry, ma fino ad n . però se il bit di carry arriva fino all'ultimo full adder; dopo rimane e mi crea overflow. Mettere in parallelo = nello stesso istante tutte operano.

Add e Sub sono fatti con un circuito sommatore; nel caso della sottrazione devo complementare a 2 il secondo operando; per farlo uso un full adder. Per calcolare $a-b$ posso fare $a+b'+1$

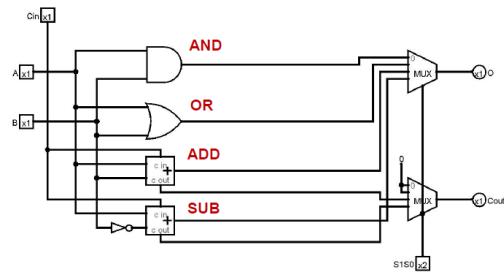
ALU a 1 bit

- Una versione ottimizzata dell'ALU può essere generata sfruttando le seguenti considerazioni:
 - E' sufficiente un unico *full-adder* per somma e sottrazione



effetti dei componenti 17

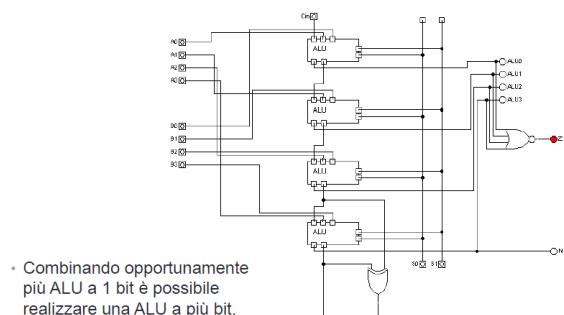
Schema logico ALU a 1 bit



- Sottrazione: $A - B = A + \text{compl.2 di } B = A + B' + 1$
→ nego B e aggiungo 1 dal carry-in (solo quello iniziale del primo bit)

Ho 4 circuiti in parallelo e 1 circuito selettori. La somma e la sottrazione posso generarmi carry o overflow. Alla fine ci basta 1 circuito sommatore → invece di tante porte logiche, ne uso molto meno perchè applico solo un full adder. Il full adder, usa la stessa porta AND che uso per fare l'AND → riuso di una porta → risparmio. Inoltre ho un solo multiplexer che seleziona le uscite, non ho più quello che selezionava il cout.

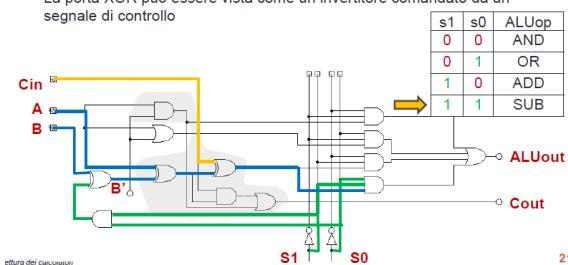
ALU a 4 bit



- Combinando opportunamente più ALU a 1 bit è possibile realizzare una ALU a più bit.

- Una versione ottimizzata dell'ALU può essere generata sfruttando le seguenti considerazioni:

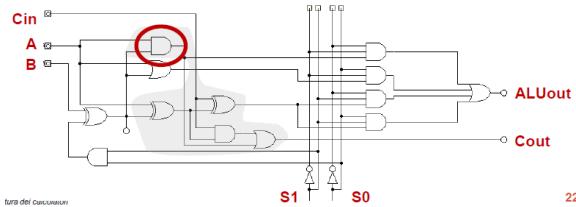
- E' sufficiente un unico *full-adder* per somma e sottrazione
- La porta XOR può essere vista come un invertitore comandato da un segnale di controllo



effetti dei componenti 21

i bit s1 e s0 servono anche per fare addizione o sottrazione; se sono entrambi 1, mi faranno il complemento di B; se no B sarà normale.

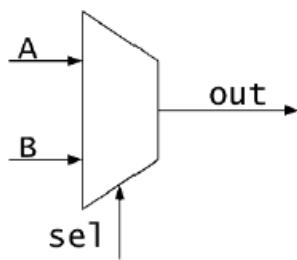
- L'AND tra A e B è necessario sia come operazione che come generazione della prima parte del carry



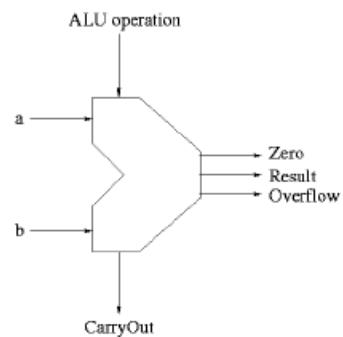
effetti dei componenti 24

Simboli:

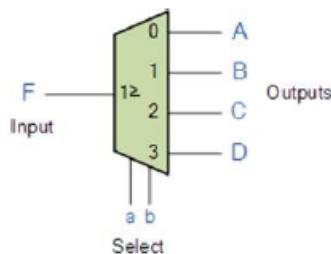
- Multiplexer:



- ALU:



- Demux:



Indice

Reti sequenziali

Reti sequenziali: reti logiche in cui in ogni istante le uscite (e il comportamento interno) dipendono non solo dalla configurazione degli ingressi in quell'istante, ma anche dalle configurazioni degli ingressi negli istanti precedenti.

- Nelle reti sequenziali il comportamento dipende dalla storia passata; devono conservare **memoria** degli eventi passati nel proprio **stato** interno.
- Variazioni delle configurazioni di ingresso modificano, oltre che le uscite, anche lo stato interno. Lo stato interno attuale si dice **stato presente**. In seguito alla variazione degli ingressi il sistema può calcolare in ogni istante quello che sarà lo **stato futuro**.

Quando avviene l'aggiornamento dello stato presente allo stato futuro appena calcolato?

- Le Reti sequenziali possono essere asincrone o sincrone:
 - **asincrone**, se le variazioni delle configurazioni di ingresso vengono sentite e modificano lo stato e le uscite in qualsiasi istante
 - **sincrone**, se le variazioni delle configurazioni di ingresso vengono sentite e modificano lo stato e le uscite solo in presenza di un opportuno evento di sincronizzazione
- L'evento di sincronizzazione è normalmente associato ad un segnale attivo (**il clock**) o al cambiamento dello stato del segnale di sincronizzazione (fronte del clock)

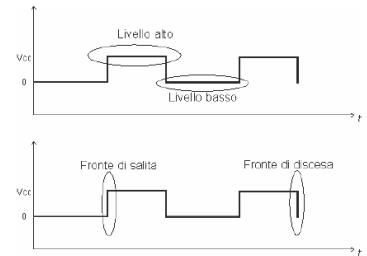
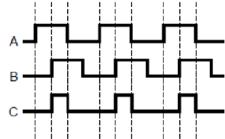
Clock

Segnale di **clock** è un segnale che oscilla con un periodo che è definito dal cristallo

- Il segnale di **clock** è generato da un circuito (realizzato con un opportuno cristallo) che emette un segnale impulsivo periodico con una precisa durata (*pulse width*) e con un preciso intervallo tra due impulsi consecutivi.
- Il **clock** è un segnale free-running ossia che continua indefinitamente (almeno finché il sistema è alimentato), di tipo periodico, con un periodo detto **tempo di clock** T_{ck} (clock cycle time); il suo reciproco è la **frequenza di clock** f_{ck} o f . Una rete che ha la frequenza di 100MHz ha un ciclo di clock di 10ns.

- Si definiscono livelli (alto e basso) e fronti o edge (di salita e di discesa) le quattro parti della forma d'onda riportata in figura

• Nei calcolatori il segnale di clock sequenzializza tutti gli eventi. Spesso nel calcolatore si usa oltre al clock primario dei clock secondari che sono sincroni ma che sono di dimensione minore (la metà) per eseguire più azioni nello stesso clock o maggiori (il doppio, il quadruplo) se alcune reti non sono sufficientemente veloci. Per questo si parla del clock della CPU, del clock di sistema, o di clock multipli (di frequenza).



Nelle reti logiche ogni evento elementare si verifica in un ciclo di clock.

- Se l'evento si verifica mentre il clock è attivo (di solito alto) si dice che la logica lavora “**a livello**”
- Se ogni evento, ogni transizione di stato e di uscite si verifica al cambiamento del clock si dice che l'evento è “**edge-triggered**” o “**a fronte**”
- Di solito si usa il fronte di salita, ma in alcuni casi si usano entrambi
- Le reti logiche che studieremo sono di tipo sincrone e normalmente di tipo **edge-triggered**.

Memoria bistabile:

- Gli elementi di base delle reti sequenziali sono gli elementi di memoria chiamati **bistabili**, capaci di mantenere al loro interno il valore 0 o il valore 1.
- Questi elementi bistabili sono gli elementi di base capaci di mantenere 1 bit di memoria. E sono gli elementi di stato

bistabile SR (Set-Reset)

Come nelle reti combinatorie si definiscono i gate elementari, così per le reti sequenziali esistono blocchi elementari per memorizzare lo stato attuale

Memoria binaria (**bistabile** asincrono): elemento capace di memorizzare il valore di una variabile di stato binaria e di commutare alla presenza di un opportuna configurazione di ingresso.

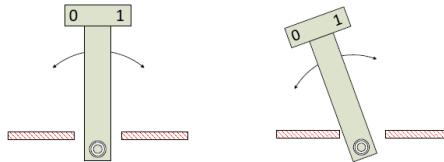
Per capire il funzionamento di un bistabile usiamo un esempio reale:

Nel disegno è stilizzato un bistabile meccanico.

La parte fatta a T è libera di ruotare attorno al perno in basso.

I due blocchi tratteggiati in rosso funzionano da fermi.

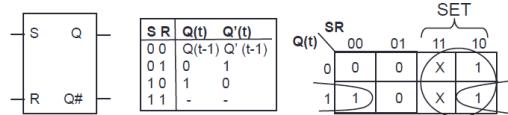
Dopo un eventuale fase di equilibrio instabile, la parte a T «cadrà» a destra o a sinistra.



restare fermo è il concetto di memoria/memorizzare

→ se non gli faccio niente, l'oggetto si ricorda il valore della condizione nel tempo precedente.

SET-RESET: è una rete con due ingressi S e R e una uscita Q (ed una uscita complementata $Q^\#$). L'uscita Q assume il valore 1 quando $S=1$ e $R=0$ o il valore 0 quando $S=0$ e $R=1$. L'uscita rimane inalterata quando $S=R=0$. La combinazione di ingresso $S=R=1$ non si deve mai verificare (configurazione proibita).



• Funzione di eccitazione del Set-Reset:

$$Q(t+1) = S + R'Q(t)$$

Sono utilizzate più di frequente le sintesi a **NOR** o **NAND**.

Latch S-R:

Rendere questi circuiti sincroni: per farlo dobbiamo attaccare un pezzo di circuito al nostro set-reset.

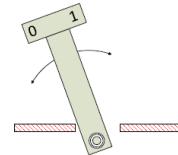
→ il **latch** lavora a **livello** → il mio latch lavora in tutto il periodo in cui il segnale è attivo.

Il segnale di **enable** è pilotato tramite il segnale del clock → lavora sulla logica del clock.

Abbiamo reso **sincrono** un circuito asincrono.(enable è il segnale di sincronizzazione) questo è di tipo

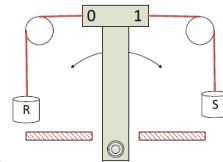
- Collocare l'oggetto a destra o a sinistra può servire per ricordarci qualcosa. Visto che la scelta è binaria, può servire per ricordarsi uno 0 o un 1. Una volta posizionato l'oggetto, starà fermo fino ad un prossimo comando.

- E' un banale esempio di memoria meccanica binaria.



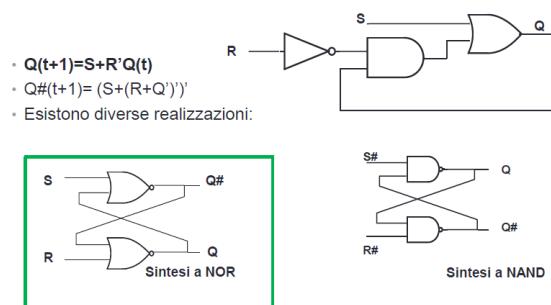
- Supponiamo di collegare, mediante appositi supporti, due corde. Chiameremo la corda a sinistra **R** e quella a destra **S** (vedi figura)

- Tirando la corda **R** si porta l'oggetto nella posizione 0, tirando la corda **S** nella posizione 1. Senza toccare le corde l'oggetto starà fermo.

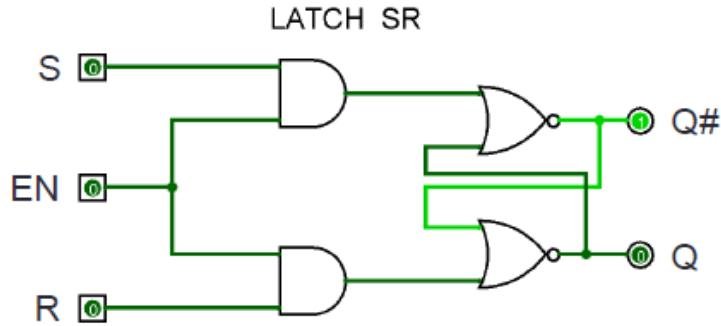


- Attenzione: tirando entrambe le corde, l'oggetto si potrebbe portare nella situazione al centro, di equilibrio instabile. Rilasciando le corde l'oggetto cadrà a sinistra o a destra in modo del tutto casuale e impredicibile. Meglio evitare...

- Chiameremo la corda di sinistra **R** per indicare l'operazione di **RESET** (porto a 0) e quella di destra **S** per indicare **SET** (porto a 1).



latch(su livello) che è diverso dal flip flop(che lavora a fronte). Non funziona più come una rete combinatoria normale, dove applicando ingressi ad un certo t, genera un output; perchè adesso esegue solo quando enable è alto.



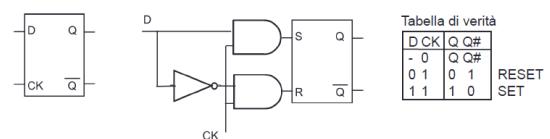
EN (enable):

- se non è attivo (0) il secondo stadio ha la configurazione di ingresso (0,0) e rimane nello stato corrente (hold)
- se è attivo (1) la rete è sensibile al cambiamento degli ingressi:
se sono entrambi a 0 la rete rimane in hold,
altrimenti la rete cambia di stato (set o reset)

D latch

SR ha gli configurazioni di ingresso che non ci interessano, ne ho 3: memoria set e reset, ma in memoria io scrivo 0 o 1 → con D latch ho solo **set** e **reset** → 1 solo bit. Chiediamo all'esterno di gestire solo un segnale. Se il segnale di clock è 0 → è un **don't care** → agisce da memoria → le uscite restano Q e Q#

- Memoria capace di mantenere l'uscita costante se il segnale di clock (o enable) non è attivo e di cambiare l'uscita campionando l'ingresso quando il segnale di clock/enable è attivo (74LS76)



- Il **flip-flop D** (derivato da questo latch) è il flip flop più usato per memorizzare dei segnali il cui valore è significativo – e quindi deve essere campionato – solo in un dato istante (**sul fronte¹ del clock** es., per memorizzare dati/indirizzi su bus multiplexati)

- una memoria è una batteria di tanti flip flop; se immaginiamo una batteria di latch → in pratica quando andiamo a memorizzare una word in binaria, semplicemente funziona

usando i valori stessi della parola; 1 sola variabile in entrata che è la word, se è 1 set, se è 0 reset???

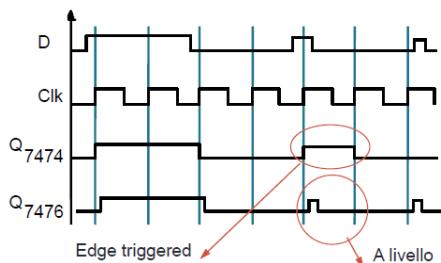
- Si definisce **latch** un bistabile sincrono trasparente, capace di memorizzare o meno segnali di ingresso in funzione di un segnale di abilitazione (*clock* o *enable*).
- La transizione di stato avviene per tutto il tempo in cui il *clock* è attivo (alto) e si hanno tante transizioni di stato quante quanti cambiamenti di ingressi avvengono in tale periodo. Il latch è *trasparente* agli ingressi quando l'*enable* è attivo

- **Flip Flop** è un dispositivo bistabile privo della proprietà di trasparenza
- Nel flip flop il cambiamento della uscita non è conseguenza del cambiamento dell'ingresso di dato ma è conseguenza del cambiamento (**edge-triggered**) di un ingresso di controllo sincrono (il *clock*) o asincrono (*preset* o *clear*)
- I flip flop si definiscono bistabili sincroni a *commutazione sul fronte* perché la transizione di stato avviene nell'istante in cui si ha l'evento significativo del *clock* (fronte di salita o di discesa) in base agli ingressi in quel momento

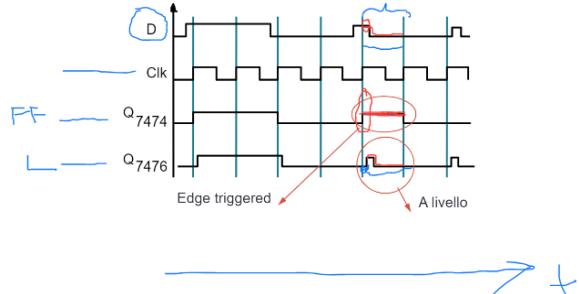
Il "problema" o caratteristica di un latch è essere **trasparente** rispetto agli ingressi: per tutto il periodo in cui il segnale è alto, se cambia il segnale di ingresso, cambia anche la configurazione dentro al latch, quindi può risultare in molte transizioni di stato. Se io però voglio solo campionare nei segnali di ingresso, quando il *clock* è alto in fronte di salita → il flop flop evita che i segnali d ingressi cambino. Il flip flop ascolta solo durante il **fronte di salita** del *clock*(o di discesa); infatti nell'immagine l'ultima variazione di D non è vista dal flip flop. Entrambi sono circuiti sincroni ma uno lavora sul fronte → campiona istantaneamente, l'altro lavora su tutto il periodo → è trasparente.

campionamento: comportamento "io scatto un'istantanea del segnale di interesse solo nel momento in cui c'è un cambiamento" ???

• Esempi di comportamenti di un Latch D (7476) e di un FF-D (7474)



• Esempi di comportamenti di un Latch D (7476) e di un FF-D (7474)



Il latch replica esattamente il comportamento di D nel periodo in cui il *clock* è alto; il flip flop invece campiona il valore del segnale di ingresso, solo nel momento in cui il *clock* diventa alto. Il flip flop si usa di solito per implementare la memoria. Perchè garantisce che mentre sto facendo la word il mio sistema campiona il segnale solo in quel momento li. Il ritardo nei segnali dell'immagine esiste davvero perchè esiste sempre il ritardo nei tempi di propagazione dei segnali. La logica flip flop è leggermente più reattiva.

il latch potrebbe essere più vincolante usarlo; perchè dobbiamo costruire una logica che catturi TUTTE le variazioni in ingresso; mentre il flip flop vuole solo il momento esatto in

cui cambia il segnale enable:



Maxbubblegum: "Si è così; il pacco del latch è che per altro, funzionando quando siamo a livello, SE mentre siamo in quello stato la configurazione di input cambia, cambia anche quella di output, mentre col flip flop noi "ci mettiamo in ascolto" solamente per un istante. Questo è molto più sicuro e stabile e ci permette di regolare il tutto con solo 1 bit. Non devo fare un bit che mi fa da "Enabled Level"."

Flip Flop JK

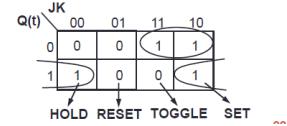
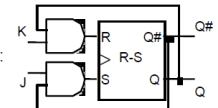
I segnali sono sempre uguali tranne l'ultimo → 11 che normalmente sarebbe proibito;
qui invece è **toggle = complemento**, praticamente mi va a scambiare il valore delle uscite col loro opposto: se una era settata a 1 diventa 0 e viceversa.

Gestisce la problematica dell'ingresso vietato; però anche lui è un po'
ridondante(come i latch)
→ ha po'tante configurazione

- Flip flop JK progettato come estensione del FF-SR
- Il problema dell'ingresso proibito (11) del FF-SR non c'è più → **toggle**
- $Q(t+1) = Q(t)K' + Q(t)'J$
- Vengono usati per il campionamento dei dati:
 - per memorizzare dato=0 JK=01
 - per memorizzare dato=1 JK=10
- basta collegare il dato a J e collegare K a J'

Tabella di verità

J	K	Q _n	Q _{n+1}	Descrizione
0	0	Q	Qn	Memoria (nessun cambiamento)
0	1	0	0	Reset
1	0	1	1	Set
1	1	Qn	Q	Toggle (complemento)



per ottenerlo c'è bisogno di utilizzare un flip flop SR!!! perchè il JK ha bisogno di lavorare a fronte del clock, se no riscontra problemi quando si inserisce 1,1 (come tutti i latch) (per creare un flip flop SR basta mettere in serie 2 latch SR)

flip flop T

funziona come componente di **toggle**.

Ha un solo ingresso e due uscite. Il segnale t lo passa sia a j, sia a k → quindi **avviene o 11 o 00**.

che nel flip flop JK equivalgono a memoria o toggle.

Li possiamo usare per i contatori, mettendo tanti Flip Flop a cascata, per cui ogni uscita si ottiene un clock dimezzato rispetto al precedente.

(Abbiamo una frequenza del segnale che raddoppia?????) Se io faccio toggle ogni volta che c'è un fronte di salita di clock → segnale di clock di ampiezza doppia.

@maxbubblegum47 dimmi se ti ho risposto

- Molto usati per fare commutare lo stato di uscita

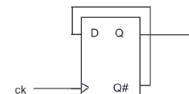
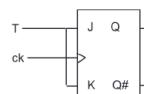
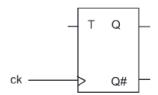


Tabella di verità:

T/Q+	Descrizione
0 Q	Memoria (nessun cambiamento)
1 Q#	Toggle (complemento)

• $Q(t+1) = Q(t)T' + Q(t)'T$



Attributo dei simboli

Proprietà: Se T=1 l'uscita Q ha frequenza dimezzata rispetto al clock.

Applicazioni: È il componente base dei [contatori](#), infatti collegando a cascata vari flip-flop T ad ogni uscita si ottiene un clock dimezzato rispetto al clock precedente.



un flip flop T lavora su fronte del clock.

quando si mettono in serie, il flip flop T successivo dipenderà dall'output di quello precedente; quindi il secondo flip flop eseguirà il toggle solo quando il primo flip flop fa uscire 1...

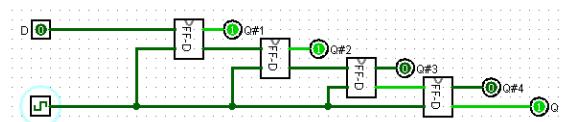
il primo flip flop fa uscire 1 ogni 2 cicli di clock, perchè lavora sul fronte.

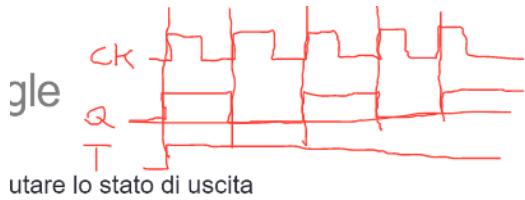
(immagine sotto).

il secondo però "vede" un 1 in ingresso che dura più di 2 cicli, perchè si basa sul 1 del primo; quindi mentre il primo flip flop riceve 0, continua a mandare in output 1; questo risulta in una frequenza raddoppiata per il secondo flip flop, perchè per cambiare di stato ha bisogno di 2 cicli del primo flip flop che a sua volta ha bisogno di 2 cicli di clock → $2 \times 2 = 4$ il doppio.

mettendone tanti in serie si ottiene un contatore, perchè il primo flip flop avrà una frequenza pari a quella della prima cifra di un numero binario → 0101010101...

il secondo flip flop(avendo bisogno del doppio dei cicli) avrà sequenza pari alle seconda cifra di un numero binario → 00110011... e così via





questi sono flip flop D che non fanno un contatore, però danno lo stesso un'idea circa

Quando usare i vari tipi di componenti?

I latch SR sono poco usati in sè; vengono usati come componenti elementare di flip flop JK e di VLSI very large scale integration chip sono flip flop D.

- Flip Flop e latch sono alla base dei circuiti sequenziali: quando usarli?
- **S-R latch** sono poco usati come blocchi funzionali (e comunque all'interno dei JK e D).
- **Flip Flop T** sono molto usati (realizzati con JK o D) all'interno dei contatori o per ricordarsi l'evoluzione di un contesto interno al sistema di elaborazione in due stati possibili
- **Flip Flop JK e D** sono entrambi i più usati: con JK si realizzano funzioni più complesse con meno logica esterna, ma richiedono più pin. In VLSI si usano più i D (componenti base della memoria)

SR:	$Q(t+1)=S+R'Q(t)$
D:	$Q(t+1)=D$
J-K:	$Q(t+1)=JQ'(t)+K'Q(t)$
T:	$Q(t+1)=TQ'(t)+T'Q(t)$

Q(t)	Q(t+1)	S	R	D	J	K	T
0	0	0	-	0	0	-	0
0	1	1	0	1	1	-	1
1	0	0	1	0	-	1	1
1	1	-	0	1	-	0	0

effettua dei calcolatori

registri

I registri e S-ram si realizzano con reti logiche

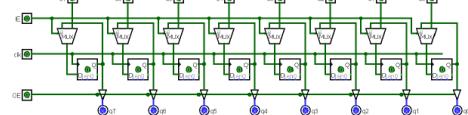
→ si mettono in cascata n flip-flop (non i T flip flop, perchè dimezzano la frequenza), se devo fare una scrittura utilizzo l'input enable; mentre se leggo utilizzo output-enable.

i MUX servono per decidere tra lettura e scrittura???(ipotesi mia)

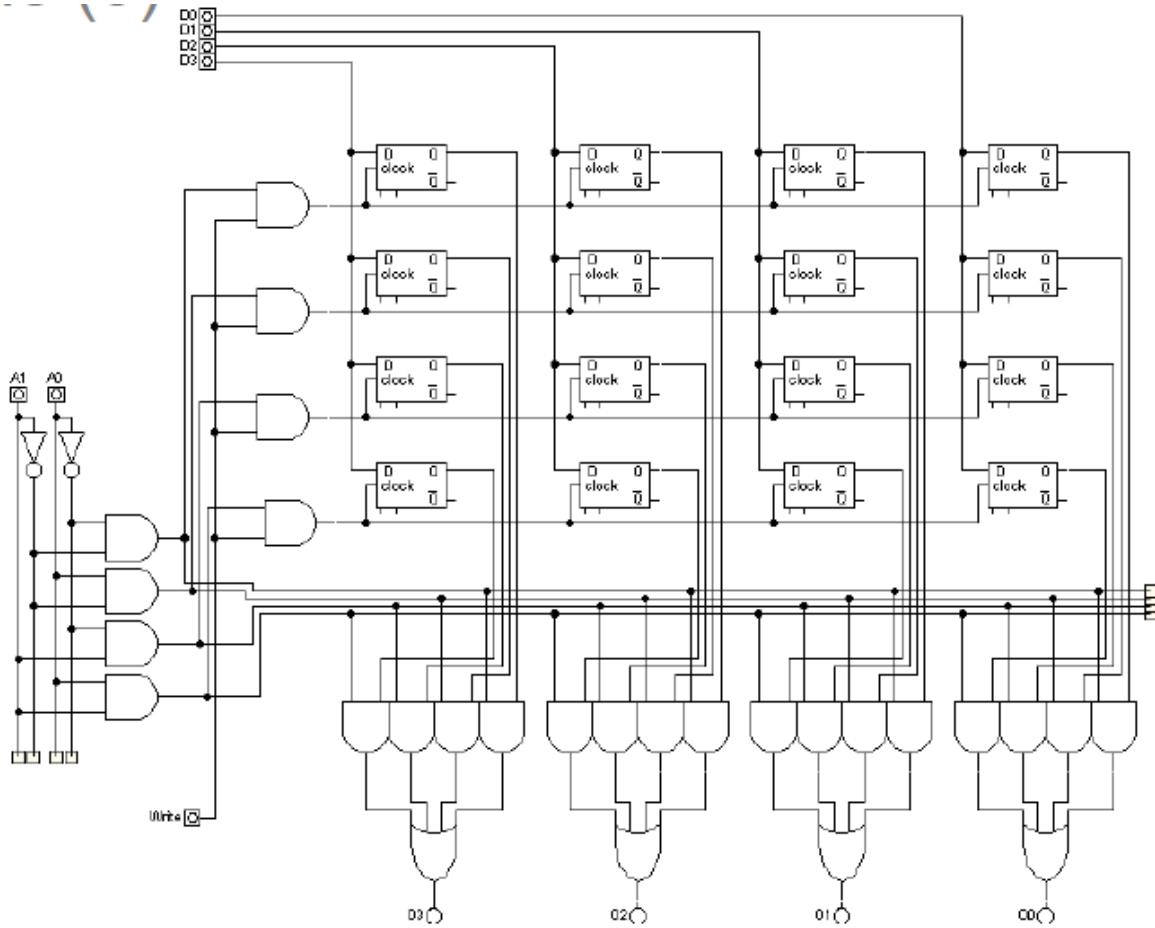
Memoria

- Le memorie sono dispositivi di memorizzazione logicamente assimilabili a banchi di registri, anche se dal punto di vista architettonico se ne discostano profondamente.
- Ogni unità di memorizzazione viene detta **cella** di memoria.
- La presenza di più di un registro introduce la ovvia necessità di **selezionare** a quale registro vogliamo accedere.

- Un registro è un elemento di memoria in cui n flip-flop vengono controllati dallo stesso clock, formando sostanzialmente una unità in grado di memorizzare parole composte da n bit.
- Tipicamente sono presenti un segnale di *Input Enable* (o *Chip Select CS*), cioè una linea che consente di attivare la fase di memorizzazione e un segnale di *Output Enable* che rende visibile in uscita la parola memorizzata.



- Dal momento che stiamo lavorando con circuiti binari, la scelta più ovvia è quella di codificare in n bit il numero e utilizzare un decoder per produrre i segnali di abilitazione della cella in questione.
- Il numero così codificato viene detto **indirizzo** della cella e il numero di bit per l'indirizzamento verrà indicato con n_a dove la a indica la parola *address* (indirizzo). Il numero di bit contenuti in ogni cella viene indicato con n_d dove d indica la parola *data* (dati).

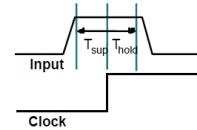
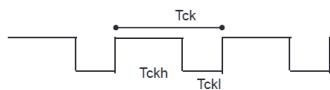


• Si dice **periodo di clock** la lunghezza del ciclo di clock e **frequenza di clock** il suo inverso

• Il **duty cycle** è la percentuale del tempo in cui il clock rimane alto

• **T_{sup}** (**tempo di setup**) è il periodo in cui gli ingressi devono rimanere stabili prima del fronte del clock per poter essere campionati correttamente

• **T_h** (**tempo di hold**) è il periodo in cui gli ingressi devono rimanere stabili dopo l'evento del clock

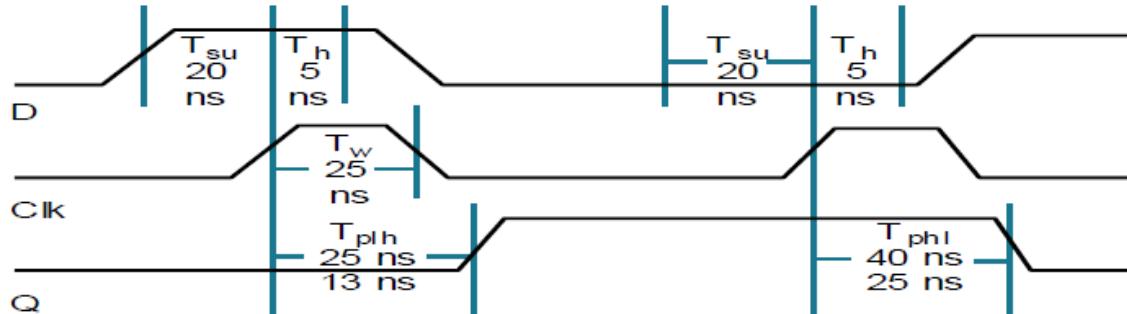


Il duty cycle può essere diversa, di solito facciamo vedere che è metà del tempo di clock, ma non sempre ci sono ritardi reali nel clock.

Tempo setup: se c'è un segnale di clock, Q sarà alta perché ho campionato D, ma perchè funzioni, va aggiunto un tempo di setup che precede il tempo di salita del clock e poi un Thold per abbracciare il tempo di salita del clock; tutto questo per avere una

"rampa" ed assicurare che il fronte di salita del clock non sia immediato, ma che sia accompagnato per assicurare il funzionamento dei componenti derivanti da esso.

- Tutte le misure sono fatte rispetto al fronte positivo del clock



- requisiti temporali del 74LS74:
- I tempi che si riportano sono
 - Setup time
 - Hold time
 - Minimum clock width
 - Propagation delays (low to high T_{plh} , high to low T_{phl} , max e typical)

Reti asincrone e sincrone

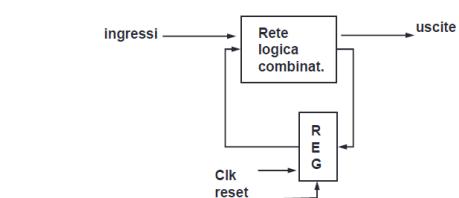
- Nel progetto di reti logiche si predilige l'impiego di reti sincrone
- Reti asincrone sono alla base delle reti sincrone, ma con un segnale di riferimento, il clock.
- Anche nelle reti sincrone (come i FF-D) esistono segnali asincroni (*clear* e *preset*)
- è meglio evitare reti asincrone (soprattutto ad alta frequenza) perché sono sensibili ad alei (corse critiche).

In alcuni casi, le reti asincrone sono inevitabili:

- circuiti di *reset*
- segnali esterni
- segnali di *handshake*
- segnali di *wait* nelle memorie

-
- Una rete sequenziale memorizza le informazioni sulle configurazioni di ingresso che si verificano nel tempo; la memorizzazione avviene in *stati interni*
 - Le variabili di stato che definiscono lo stato interno in cui si trova la rete sono memorizzate in elementi di retroazione
 - Tra le reti sequenziali, importanza fondamentale hanno le macchine a stati finiti (**FSM, Finite state machine**) in cui gli elementi di retroazione sono Flip Flop con un unico segnale di clock
 - L'insieme dei FF è detto **registro di stato** e memorizza lo stato futuro presentando a valle lo stato presente

→ Riportare indietro un uscita verso gli ingressi



Percorso di retroazione è il percorso dove inserisco la memoria(flip flop); se ho più bit inserisco più flip flop(**REG**).

Una qualunque rete sequenziale è composta da **3 blocchi combinatori**; la retroazione tramite **elementi di memoria**(flip flop) rende questa rete sequenziale; retroazione sia in uscita sia nello stato.

Differenza tra i 2 modelli di **automi a stati finiti(FSM)**:

In Mealy gli ingressi vengono

mandati ad entrambe le funzioni;

la rete di uscita dipende sia dallo stato presente sia dagli ingressi in quel istante.

In Moore invece vanno solo alla rete di stato, non a quella di uscita

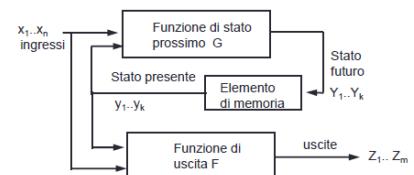
→ **la rete di uscita dipende solo dallo stato presente!!**

In Moore c'è stato e uscita; in melay c'è ingresso e uscita.

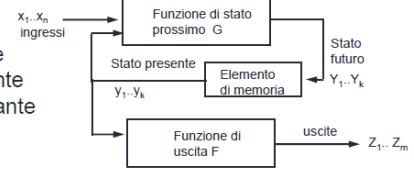
in melay spesso avrò meno stati rispetto a moore, perchè non ho bisogno dello stato esatto, ma uso gli archi, ovvero stato+ingresso questo si vede spesso negli stati finali, che in moore ci sono sempre, mentre in melay si vedono meno, perchè non ne ha bisogno.

guarda qui

- Modello generale (**automa di Mealy**): il valore delle uscite dipende dallo stato presente e dagli ingressi in quell'istante

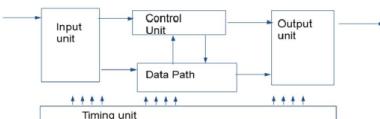


- Modello equivalente (**automa di Moore**): il valore delle uscite dipende solo dallo stato presente e non dagli ingressi in quell'istante



E' sempre possibile passare da un modello all'altro
Il modello di Moore ha più stati ma funzioni di uscita più semplici

- La maggior parte delle reti sequenziali sincrone sono descrivibili come FSM, più o meno complesse.
- Anche la **CPU** è descrivibile come una FSM avente la parte sequenziale composta dalla **Control Unit** che passa attraverso diversi stati interni (lettura delle istruzioni, decodifica, esecuzione) in base ai segnali esterni (istruzioni e dati) allo stato interno (flag, stato di esecuzione attuale, ...), per fornire le uscite, i dati elaborati e i segnali esterni al calcolatore).



- Una generica rete sequenziale è pertanto definita dalla *tupla* $\langle X, Z, S, F, G \rangle$ e richiede in pratica la realizzazione di due funzioni combinatorie (F, G) che dipendono dai due insiemi di valori (X e S). Inoltre, sono necessari dispositivi in grado di memorizzare lo stato prossimo e presentarlo come stato presente nell'intervallo di lavoro successivo della rete sequenziale.

- A seconda del progetto e della descrizione a parole si può decidere di realizzare un automa di **Moore** o di **Mealy**. Di solito l'automa di **Mealy** ha meno stati (quindi meno elementi di memoria) ma ha le reti combinatorie ed in particolare la rete combinatoria delle uscite più complessa e quindi potenzialmente più lenta. Spesso si realizza l'automa di **Moore** perché è concettualmente più semplice.

X = insieme degli ingressi.

S = insieme dei bit di stato → lo stato presente e futuro.

Z → insieme delle uscite.

F e **G** 2 reti combinatorie che fanno evolvere lo stato e le uscite.

mealy ha le uscite più complesse, perchè in ingresso alle funzioni di uscita **F**, ha anche gli ingressi della rete sequenziale **G**(moore no).

Moore dipende solo dai bit di stato → ha più stati; ciò che rende il circuito sequenziale, è che c'è memoria e retroazione.

Sintesi reti sequenziali:

1. Si prepara una **descrizione comportamentale a parole** o con un linguaggio di descrizione dell'hardware. (*specifiche di progetto*)
2. Si definisce il **diagramma degli stati** per definire le transizioni che si traduce nella tabella di flusso. Questa è la fase più importante che corrisponde in software alla creazione dell'algoritmo perché si definiscono gli stati interni e le transizioni
3. Si impiegano metodi manuali o automatici per la **minimizzazione degli stati**. Spesso il diagramma degli stati può essere minimizzato con un numero minore di stati (esistono algoritmi appositi).
4. Dal diagramma minimizzato e tabella di flusso corrispondente si crea la **tabella delle transizioni e delle uscite** con l'assegnamento degli stati (indicando quale numero binario corrisponde ad ogni stato, date le variabili di stato presente e futuro).
5. Infine si ottiene la **implementazione** (avendo scelto i componenti bistabili elementari e i gate elementari per le reti combinatorie).

- Una rete sequenziale può essere rappresentata da un **diagramma degli stati**:
 - il **diagramma degli stati** è un grafo con tanti nodi quanti gli stati e tanti archi quante le transizioni da uno stato all'altro dovute a cambiamenti degli ingressi
- nel **diagramma degli stati** vengono rappresentati inoltre i valori delle uscite per ogni stato:
 - negli archi se il modello è di Mealy
 - nei nodi se il modello è di Moore

- Esercizio:** Progettare una rete sequenziale in grado di riconoscere, in presenza di una sequenza di cifre binarie, quando si sono presentati successivamente due 0 e un 1.

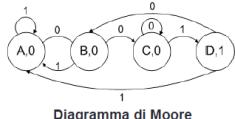


Diagramma di Moore

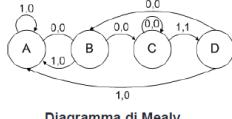


Diagramma di Mealy

- Codifica degli stati e sintesi delle uscite (Moore)

s_{1s0} - stati presenti
 S_{1S0} - stati futuri

s	s ₁	s ₀	z
A	0	0	0
B	0	1	0
C	1	1	0
D	1	0	1

$$z = s_{1s0'}$$

Tabella di Transizioni
degli stati (Moore)

Stato presente			Ingresso	Stato Futuro		
s	s ₁	s ₀	x	S	s ₁	s ₀
A	0	0	0	B	0	1
	0	0	1	A	0	0
B	0	1	0	C	1	1
	0	1	1	A	0	0
C	1	1	0	C	1	1
	1	1	1	D	1	0
D	1	0	0	B	0	1
	1	0	1	A	0	0

x	0	1
00	0	0
01	1	0
11	1	1
10	0	0

$$s_1 = s_0 x' + s_{1s0}$$

x	0	1
00	1	0
01	1	0
11	1	0
10	1	0

$$s_0 = x'$$

Io cerco 001. Finché trovo 1, rimango
nello stato limbo.



in moore ho il valore dell'uscita di fianco allo stato(A,0) e negli archi il valore dell'ingresso.

in mealy ho gli archi con **il valore dell'ingresso E il valore dell'uscita** di tale configurazione stato-ingresso; questo mi permette a volte di risparmiare sul numero degli stati

Quali sono gli stati? A B C D. quali transizioni mi fanno passare da uno stato all'altro?
leggo uno 0 o un 1.

se leggo uno 0 dallo stato A → vuol dire che ho riconosciuto la prima cifra che cerco.
quando sono in B ho già fatto il primo passo! perchè avrò già trovato uno 0 → se leggo un 1 allora si torna indietro; se invece ricevo un altro 0 passo a C.

quando sono in C → se ricevo uno 0 resto su questo stato; perchè essendo in C, ho già letto 2 zeri, quindi se al ciclo dopo leggo uno zero, vale ancora la condizione.

Se invece leggo 1 allora passa a D.

Lo stato C ha solo memoria degli ultimi 2 istanti; quello che è successo prima non lo sa
in questo riconoscitore di sequenze mi serve un minimo di 3 cicli di clock! (il primo 0, il
secondo 0 e il terzo 1).

La rete legge 1 bit alla volta; 1 bit a ciclo.

Lo stato ABCD deve essere tradotto in un segnale che ha i bit necessari per
rappresentare il numero di stati → in questo caso 4 stati → 2 bit

Per ogni stato mi escono 2 archi → devo scrivere tutte le transizioni(configurazioni)
possibili, quindi 2 per ogni ingresso. Se avessi avuto 2 ingressi allora avrei avuto 4 archi
per ogni stato.

stessa cosa vale per la tavella di transizioni → se ho 2 ingressi, ho 4 righe per stato.

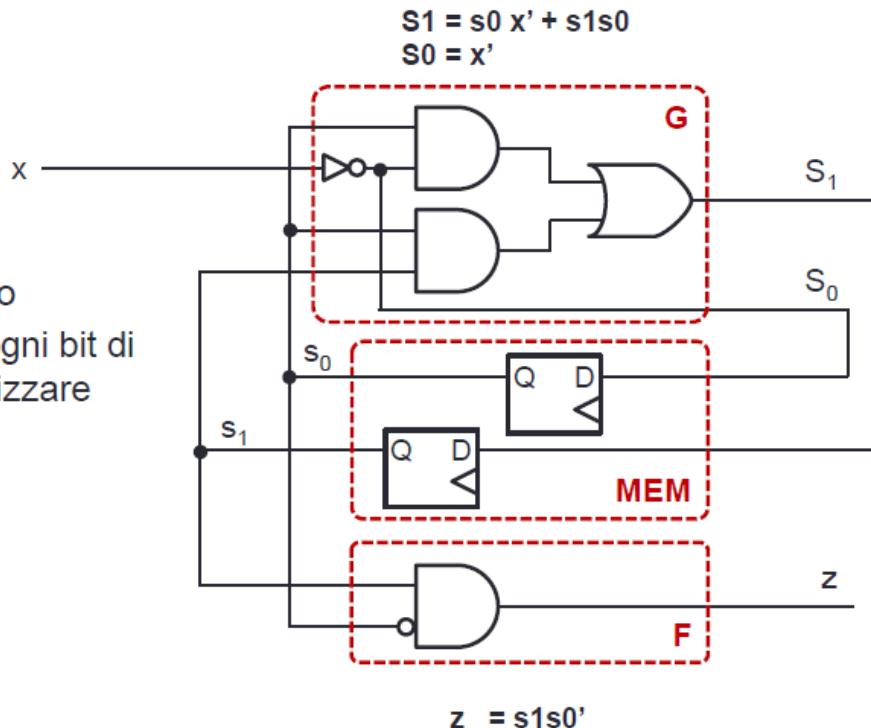
2 uscite → stato futuro e stato presente → 2 tabelle di karneau.

- Quindi ho finito perchè ho progettato sia la rete G e la rete F → devo ricomporre il circuito mi serve un bit di memoria per ogni bit di stato????

- Flip Flop D:

$$Q(t+1) = D(t)$$

- uscita = ingresso
- un flip flop per ogni bit di stato da memorizzare
- $D_1 = S_1$
- $D_0 = S_0$



• **Testo:** Effettuare la sintesi di un automa a stati finiti sincrono che controlla un ventilatore "digitale". Il ventilatore, oltre allo stato spento, può funzionare a 3 velocità differenti (V_0, V_1, V_2). Per controllare la velocità, esistono due ingressi, rispettivamente Più (P) e Meno (M) che incrementano e decrementano di una unità la velocità del ventilatore. Inizialmente il ventilatore risulta essere spento. Premendo il tasto P si porta alla velocità V_0 , quindi alle altre velocità. Dalla velocità V_0 premendo il tasto M è possibile spegnere il ventilatore. La politica da usare nel caso della pressione contemporanea dei tasti più e meno è a discrezione dello studente e deve essere riportata e commentata. Le uscite della rete devono essere la velocità del ventilatore e il suo stato (acceso/spento).

Ci sono 4 stati → spento, velocità 1, velocità 2, velocità 3 → per rappresentarli servono 2 bit($S_0 S_1$)

• Realizzare l'automa a stati e sintetizzare le funzioni di transizione di stato e di uscita. Indicare se nella sintesi si è utilizzato un automa di **Mealy** o di **Moore**. Disegnare il circuito logico corrispondente utilizzando Flip Flop D.

2 ingressi → 4 configurazioni di ingressi.
 $p=1$ e $m=1$ decido io cosa fare e qui abbiamo deciso di decrementare la velocità.

- Come prima cosa è necessario identificare bene le componenti dell'automa, ovvero ingressi, stato, uscite.
- Il numero di ingressi è pari a due (P, M). Comportamento del sistema:
 - $P=0$ e $M=0$ → rimango nello stato attuale
 - $P=1$ e $M=0$ → incremento la velocità
 - $P=0$ e $M=1$ → calo la velocità
 - $P=1$ e $M=1$ → calo la velocità
- Nel caso di pressione contemporanea di due tasti, imponiamo al sistema di considerare solo il tasto M che ha la precedenza sul P.
- Il numero totale di stati è 4: Sp, V_0, V_1, V_2 .
 - Quindi servono due bit per la sua codifica ($S_0 S_1$).

Introduciamo una codifica degli stati:

Stato	$S_0 S_1$
S_p	00
V_0	01
V_1	10
V_2	11

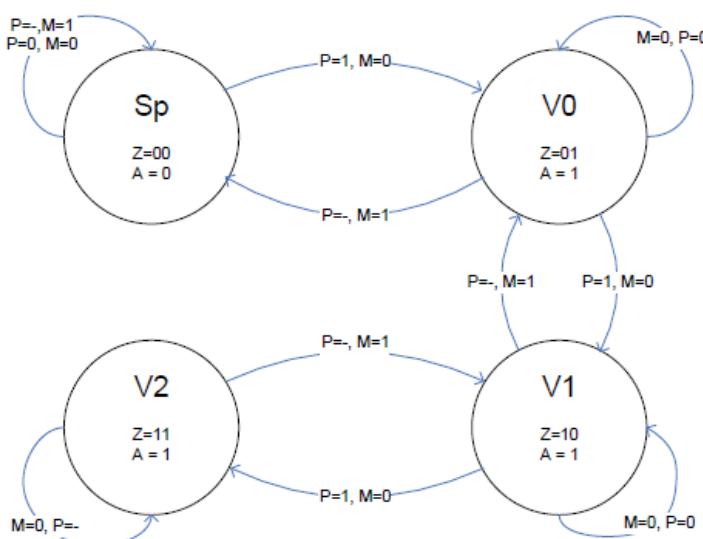
Le uscite infine sono due: la velocità, che chiamiamo Z , la quale dovendo variare da 1 a 3 deve essere composta da due bit: $Z_1 Z_0$. L'altra uscita, invece, può essere chiamata "acceso" (A), che varrà 1 se e solo se il ventilatore è acceso

2 uscite: velocità che sarà a 2 bit perchè varia da 1 a 3 → $Z_1 Z_0$ e poi l'uscita per dire se è acceso o spento → 1 bit

- Conviene moore perchè per definizione, l'uscita A, sono acceso o spento dipende solo dallo stato in cui mi trovo; stessa cosa la velocità.?????

Transizioni:

- Visto che entrambe le uscite dipendono solo dallo stato in cui si trova il ventilatore e non dall'ingresso che ha portato a quello stato, effettuiamo la sintesi dell'automa di **Moore** corrispondente.



Nota: è necessario inserire anche gli auto-anelli corrispondenti alla combinazione di ingresso $M=0$ e $P=0$!!

Da ogni nodo escono tanti archi quanti sono le possibili configurazioni degli ingressi → 4 configurazioni degli ingressi perchè 2 bit

→ però escono 3 archi da ogni nodo, perchè 2 configurazioni sono uguali [$P=0, M=1$] e [$P=1, M=1$]; entrambi riducono la velocità, quindi si usa un singolo arco per entrambi

In **Sp** ho un arco configurato da 00, 10 e 11, tutte nello stesso arco perchè vogliono dire la stessa cosa **in questo stato**(calare la velocità mentre il ventilatore è spento, risulta nel restare spento);
poi 01 è un arco che ci fa transizionare allo stato **V0**.

Da **V0** se ho 10 o 11(vogliono dire la stessa cosa) torno a Sp(spento);
00 ci fa restare in **HOLD**;
01 come prima ci fa transizionare al prossimo stato: V1

Così via...

Ultimo stato: **V3** se m=1 torno indietro(indipendentemente da quando vale p), ovvero sia con 10, sia con 11 torno indietro...

Tabella con 16 righe perchè 4 stati
ognuno con 4 configurazioni di
ingresso(2 ingressi)

Le colonne dello stato presente si
compilano enumerando
normalmente, utilizzando le
configurazioni della codifica degli
stati fatta prima.

Lo stato futuro deriva dal
diagramma delle transizioni:
**se sono in SP e gli ingressi
valgono 00 o 10 o 11 resto fermo,
se valgono 01 vado avanti; nella
tabella ho scritto la stessa cosa!**

A destra non uso una codifica!
dipende dalle transizioni;
a sinistra invece scelgo io se gray o
binario....

- La corrispondente tabella di transizione degli stati che mi permette di ricavare lo stato futuro a partire da ogni combinazione di ingresso/stato corrente è la seguente
- usando s minuscolo per lo stato presente, S maiuscolo per quello futuro

Stato presente			Ingresso		Stato Futuro			
s	s ₁	s ₀	M	P	S	S ₁	S ₀	
Sp	0	0	0	0	Sp	0	0	
Sp	0	0	0	1	V0	0	1	
Sp	0	0	1	0	Sp	0	0	
Sp	0	0	1	1	Sp	0	0	
V ₀	0	1	0	0	V0	0	1	
V ₀	0	1	0	1	V1	1	0	
V ₀	0	1	1	0	Sp	0	0	
V ₀	0	1	1	1	Sp	0	0	
V ₁	1	0	0	0	V1	1	0	
V ₁	1	0	0	1	V2	1	1	
V ₁	1	0	1	0	V0	0	1	
V ₁	1	0	1	1	V0	0	1	
V ₂	1	1	0	0	V2	1	1	
V ₂	1	1	0	1	V2	1	1	
V ₂	1	1	1	0	V1	1	0	
V ₂	1	1	1	1	V1	1	0	

Da qui tiro fuori le sintesi di
karneauh; tante quanti sono i bit di
stato → 2 (S1 e S0).

- Effettuiamo la sintesi di S₁ e S₀ mediante mappe di Karnaugh. S₁ e S₀ sono due funzioni che dipendono stato attuale e dagli ingressi.

S ₁	M.P			
	00	01	11	10
S ₁ ,S ₀	00	0	0	0
	01	0	1	0
	11	1	1	1
	10	1	0	0

$$S_1 = s_1 s_0 + s_1 \overline{M} + \overline{M} P s_0$$

S ₀	M.P			
	00	01	11	10
S ₁ ,S ₀	00	0	1	0
	01	1	0	0
	11	1	1	0
	10	0	1	1

$$S_0 = s_0 \overline{M} \overline{P} + \overline{s}_0 \overline{M} P + s_1 \overline{M} P + s_1 \overline{s}_0 M$$

Per le reti combinatorie ci servono
sempre gli ingressi e lo stato
presente quindi in karneauh
verranno usati come lati
→ 2 bit di ingresso sopra, 2 bit di
stato presente a sinistra.

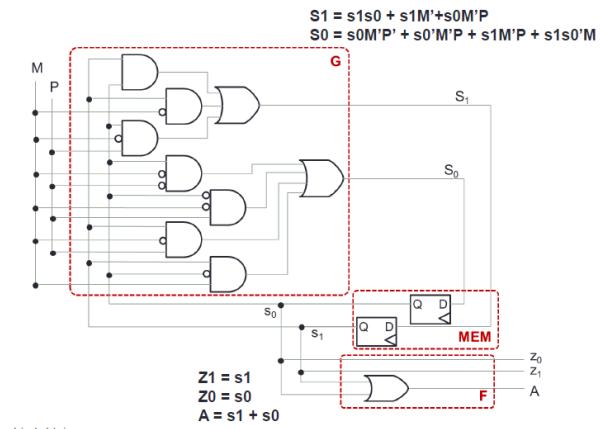
Le uscite sono facili perchè in moore dipende solo dallo stato
→ metto solo i bit dello stato.

- La sintesi delle uscite invece è molto semplice e si può evitare di passare tramite le mappe. Infatti la velocità corrisponde allo stato in cui ci si trova, e la seconda uscita vale 0 solo nello stato Spento Sp.

		Stato		Uscite			Da cui:
s	s1	s0	Z1	Z0	A		
Sp	0	0	0	0	0		
V0	0	1	0	1	1		
V1	1	0	1	0	1		
V2	1	1	1	1	1		

sintesi con flip flop D

Il numero dei flip flop varia in base al numero di stati! (numero di bit necessario per rappresentare tutte le configurazioni degli stati)



Esercizio 1

[Indice](#)

Formati di istuzione RISC-V(instruction set architecture)

Il codice binario è in pratica la word dell'architettura → il linguaggio binario sono stringhe di x bit binari che fungono da parole, però si lavora ad un linguaggio intermedio.

Lavorare unicamente con gli zeri e gli uno è davvero troppo complicato per un semplice essere umano.

Assembly: linguaggio binario tradotto in un formato capibile da un umano. Siamo al livello più basso di linguaggio che un umano può gestire; sotto troviamo direttamente il codice binario.

- Il compilatore ha il ruolo di tradurre il linguaggio ad alto livello in assembly

- L'assembler prende in ingresso un programma scritto in assembly e lo traduce in binario

Instruction set: repertorio di istruzioni che la macchina mette a disposizione del compilatore. Ci sono instruction set semplici e complicati; i vecchi computer ne avevano di semplici. Molti computer quotidiani hanno IS semplici.

RISC-V (reduced instruction set computer)

il RISC-V ha un **IS** ridotto, piccolo e semplice; è diventato ormai una ISA che viene integrata in altre cose, perchè è "gratis" (è una iniziativa di tipo Free e Open). RISC-V è una ISA stabile, quindi ci sono parti stabili e funzionanti e parti concepite come estensioni che sono disponibili ai progettisti nel caso. Ha la garanzia che il codice scritto in risc-v è compatibili in tutte le macchine che si basano su questa ISA.

RISC-V da a disposizione una licenza di microprocessore; da qua si può partire per creare un vero circuito integrato. Si può usare un FPGA per provarlo, o anche progettare software → FPGA è un circuito integrato che può essere programmato → offre i circuiti logici, ma sta al compratore programmarlo → [link](#)

RISC-V

I processori **RISC** hanno una serie di circuiti dedicati per le **poche** istruzioni che possono eseguire; circuiti ben definiti

- RISC ha un HW più semplice da implementare e anche di minor costo
- il RISC ha istruzioni load e store esplicite

Processori intel ancora oggi usano il **CISC**(complex instruction set computer)

I processori CISC hanno tante istruzioni anche molto complicate(più astratte):

- la circuiteria è più complessa e a sua volta micro programmata:
 - do la possibilità di fare programmi molto più complessi, infatti scrivere un programma in CISC è più semplice
- un design in CISC mette enfasi sul HW:
 - HW più complicato per una scrittura programmi più semplice.
 - nel CISC spendo più transistor per implementare l'HW
- nel CISC le istruzioni load e store sono integrate in altre operazioni

- RISC-V is a *Reduced Instruction-Set computer* (RISC)
 - To execute each instruction there is separate electronic circuitry in the control unit, which produces all the necessary signals. It is also called hard-wired approach
- Intel x86 is a *Complex Instruction Set Computer* (CISC)
 - The control unit contains several microelectronic circuits to generate a set of control signals and each micro-circuit is activated by a microcode.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

CPU:

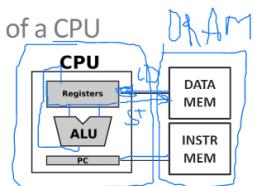
Ha una ALU, un blocco di registri attaccato alla data memory e un blocco PC

(https://it.wikipedia.org/wiki/Program_counter) attaccato alla instruction memory che ci dice che istruzione va fatta (praticamente un puntatore all'istruzione successiva da eseguire). Normalmente le 2 memorie sono la DRAM.

Nel RISC se devo lavorare sui dati:

PRIMA carico sui registri, poi faccio l'operazione su ALU, poi rимetto su registri e faccio load.

Instructions: The language of a CPU



- A CPU executes instructions via its core computation logic block: the *arithmetic-logic unit*, or *ALU*
- The *ALU* processes data that has been previously copied into the *register file* from the *data memory*¹
- *Instructions* of a program are stored in the *instruction memory*, and the special *program counter* (*PC*) register holds the *address* of the next instruction

[1] This approach is typical of reduced instruction set computers (RISC), as opposed to complex instruction set computers (CISC).

lb t0, 8(sp)	Loads (dereferences) from memory address (sp + 8) into register t0. lb = load byte, lh = load halfword, lw = load word, ld = load doubleword.
sb t0, 8(sp)	Stores (dereferences) from register t0 into memory address (sp + 8). sb = store byte, sh = store halfword, sw = store word, sd = store doubleword.

<https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly/>

Program counter (**PC**) decide la nuova istruzione

Come sono fatte le istruzioni? RISC-V si basa sulla semplicità:

1 operando di destinazione e massimo 2 operandi su cui si fa un'operazione

Introduco più variabili nelle operazioni ed è un problema perché i registri sono una risorsa costosa → ne ho pochi, di solito 32, quindi devo essere in grado di fare le operazioni utilizzando solo questi 32; spesso si fa lo spill???

- Add and subtract, three operands
- Two sources and one destination

```
add a, b, c // a gets b + c
```

- All arithmetic operations have this form

- *Design Principle 1: Simplicity favours regularity*
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

- C code (*allowed instructions are not too simple*):

$$f = (g + h) - (i + j);$$

- Compiled RISC-V code (*split in simple instructions*):

```
add t0, g, h // temp t0 = g + h
```

```
add t1, i, j // temp t1 = i + j
```

```
sub f, t0, t1 // f = t0 - t1
```

I registri sono a 32 o 64 bit

Ci sono general purpose register = normali; e ci sono registri particolari tipo quelli che supportano il floating point.

- › Arithmetic instructions use register operands

- › RISC-V has a $32 \times 64\text{-bit}$ register file

- Use for frequently accessed data
- 64-bit data is called a “doubleword”
 - › $32 \times 64\text{-bit}$ general purpose registers $x0$ to $x31$
 - 32-bit data is called a “word”

- › *Design Principle 2: Smaller is faster*

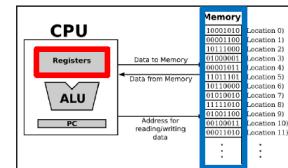
- c.f. main memory: millions of locations

- **Registers** are local to the CPU (the ALU)

- implemented with very fast memory (SRAM)
 - as fast as the ALU
 - but **VERY small**

- **Data memory** lives far away from the CPU

- implemented with slow (DRAM) technology
 - two orders of magnitude slower than the CPU
 - but **large**



Panoramica registri più importanti:

x0 è vincolato a mantenere sempre il **valore costante di 0 (hard wired)** → quasi tutte le istruzioni che hanno bisogno di un termine di paragone(confronto) vengono riscritte per avere un confronto con 0.

x1,x2,x3,x4, per CONVENZIONE (quindi non è necessario, mentre x0 è proprio costruito così), sono utilizzate per supporto al linguaggio C.

x1 ci serve per implementare la chiamata di ritorno (= return address)

x2 implementare l'astrazione della memoria automatica (= stack pointer)

x3 global pointer

x5.... sono registri che possiamo utilizzare per ospitare un risultato temporaneo
anche x9, x18.... fanno la stessa cosa. anche i save sono temporanei?????

x10-11 ospitano gli argomenti delle funzioni e del ritorno di una funzione
di solito **x10 ha il ritorno e x11 gli argomenti**

- **x0:** the constant value 0
- **x1:** return address
- **x2:** stack pointer
- **x3:** global pointer
- **x4:** thread pointer
- **x5 – x7, x28 – x31:** temporaries
- **x8:** frame pointer
- **x9, x18 – x27:** saved registers
- **x10 – x11:** function arguments/results
- **x12 – x17:** function arguments

- C code:

$$f = (g + h) - (i + j);$$

- f, ..., j in x19, x20, ..., x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Questo è un summaries che ho caricato dal pdf che parla di calling convention che fa vedere come per altro ci siano anche registri che si utilizzano per i floating point: <https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>

Un altro pdf che ti consiglio assolutamente è questo manuale sulle istruzioni di risc v: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

Un ultimo consiglio che mi sento di darti è: vedi qualche anime o leggi qualche manga per capire meglio questi argomenti.

Register allocation punto molto importante di un compilatore → il compilatore decide qual'è la maniera migliore di utilizzare i registri, essendo una risorsa piccola e molto costosa.

Come faccio a lavorare con strutture dati

più complesse come per array?

Si utilizza la main memory e si terrà sui registri solo i pezzi della struttura dati su cui lavora

→ quindi faccio precedere la mia operazione da una load dei dati dalla main memory, poi dopo l'operazione faccio una store su main memory dei dati sui registri

Risc-v è un'architettura detta **little endian**

→ il byte più significativo si trova all'indirizzo più piccolo della parola

- Main memory used for composite data
 - Arrays, structures, dynamic data

- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory

- Memory is byte addressed
 - Each address identifies an 8-bit byte

RISC-V does not require words to be aligned in memory

- RISC-V is LittleEndian
 - Least-significant byte at least address of a word
 - c.f. BigEndian: most-significant byte at least address

- Registers are faster to access than memory

- Operating on memory data requires loads and stores
 - More instructions to be executed

- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

- C code:

```
A[12] = h + A[8];
```

- h in x21, base address of A in x22

- Compiled RISC-V code:

- Index 8 requires offset of 64
 - 8 bytes per doubleword

```
1d      x9, 64(x22)
add    x9, x21, x9
sd      x9, 96(x22)
```

Anche se l'architettura è 32 bit, si utilizza una parola da 1 byte → se devo leggere porzioni più piccole di un byte è da usare uno shift sul byte

Faccio una load → metti dentro x9 il risultato dell'operazione di lettura; cose leggo?

L'indirizzo di memoria che si trova dentro ad x22(indirizzo di memoria di A) con un offset di 64; cosa sono i 64?

è l'offset in byte dell'ottavo elemento dell'array A.

Ogni elemento di un array è grande 8 byte, quindi per arrivare all'ottavo

elemento dovrò passare 7 elementi

$\rightarrow x22 * 8$;

ogni elemento pesa 8 bit(2 byte), quindi è un double

$\rightarrow x22 * (8 + 8) \rightarrow$ il byte è 4 bit

La store è duale della load \rightarrow il primo operando è il registro che contiene il registro da cui prendere l'operando. il secondo è dove storarlo.

Operando immediato = operando con un valore costante

posso usarli dentro alle mie operazioni; cambia che aggiungo una **i** alla fine del nome dell'istruzione (*add* \rightarrow *addi*)

Sono operazioni molto frequenti quelle con valori numerici costanti \rightarrow invece di fare ogni volta load e store; si è creato un circuito dedicato nel caso in cui bisogna usare un operando costante.

- Constant data specified in an instruction

addi x22, x22, 4

- Make the common case fast

- Small constants are common
- Immediate operand avoids a load instruction

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	$x5 = x6 + x7$	Three register operands; add
	Subtract	sub x5, x6, x7	$x5 = x6 - x7$	Three register operands; subtract
	Add Immediate	addi x5, x6, 20	$x5 = x6 + 20$	Used to add constants
	Load doubleword	ld x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Doubleword from memory to register
Data transfer	Store doubleword	sd x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Doubleword from register to memory
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
Logical	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345000	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits
	And	and x5, x6, x7	$x5 = x6 \& x7$	Three reg. operands; bit-by-bit AND
Conditional branch	Inclusive or	or x5, x6, x8	$x5 = x6 x8$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	$x5 = x6 \oplus x9$	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	$x5 = x6 \& 20$	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	$x5 = x6 20$	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	$x5 = x6 \oplus 20$	Bit-by-bit XOR reg. with constant

Category	Instruction	Example	Meaning	Comments
Shift	Shift left logical	sll x5, x6, x7	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	sll1 x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srl1 x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	sra1 x5, x6, 3	$x5 = x6 \gg 3$	Arithmetic shift right by immediate
	Branch if equal	beq x5, x6, 100	If $(x5 == x6)$ go to PC+100	PC-relative branch if registers equal
Unconditional branch	Branch if not equal	bne x5, x6, 100	If $(x5 != x6)$ go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	If $(x5 < x6)$ go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	If $(x5 >= x6)$ go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	If $(x5 < x6)$ go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	If $(x5 >= x6)$ go to PC+100	PC-relative branch if registers greater or equal, unsigned
Procedure call	Jump and link	jal x1, 100	$x1 = \text{PC}+4; \text{go to } \text{PC}+100$	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	$x1 = \text{PC}+4; \text{go to } x5+100$	Procedure return; indirect call

I branch sono i costrutti condizionali e vanno a sommare un numero specifico al **PC** \rightarrow **program counter**

Istruzioni

- Instructions are encoded in binary
 - Called machine code
- RISC-V instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!

Istruzioni = sequenza di 0 e 1 che vivono in questa **instruction memory** che però è sempre dentro alla DRAM; quindi anche in questo caso abbiamo il bisogno di fare load per prendere istruzioni. Per questo di solito si ragione come instruction cache e data cache.

Progettare una cache istruzioni è più facile di progettare una cache data, perchè?

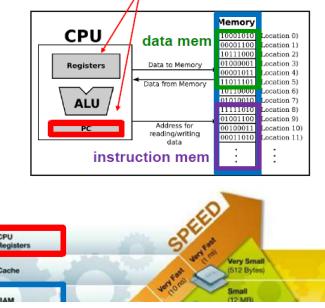
Ci sono casi in cui le due cache sono fisicamente diverse e normalmente una "instruction cache" ha **località migliore** di una data cache → località vuol dire che è più vicina a dove parte/arriva il dato → meno ritardo (ho tutti i dati più disposti in maniera più conveniente possibile). Però in realtà il motivo per cui il circuito che controlla la cache è più semplice di quello che controlla i dati?

Con la memoria istruzioni io devo **solo LEGGERE**. L'instruction memory è uno storage dal quale prendo sequenze di bit che devo interpretare in istruzioni.

A ogni ciclo il mio processore fa una **fetch** di una nuova istruzione (che equivale ad incrementare di 4 il PC); si passa però da una cache, non avviene direttamente con una load su DRAM, perchè rallenteresti tanto il processore → quindi è fondamentale avere una gerarchia di cache anche in mezzo al percorso dal PC al instruction cache?

Where do instructions live?
the program counter is actually part of the register file

- Instruction memory constitutes a logical part of the system memory
- Just like data memory, it lives far away from the CPU
 - implemented with slow (DRAM) technology
 - two orders of magnitude slower than the CPU
 - but large
- The special register program counter (PC) contains the address of the next instruction



il PC normalmente è incrementato ogni 4 perchè?

Perchè 4 sono byte e noi lavoriamo su un architettura a 32 bit → le istruzioni sono a 32 bit → **saltare da un istruzione alla prossima significa saltare di 32 bit** → di solito c'è un contatore a cui viene sommato 4

Finchè non ho branch, io continuo a leggere dall'alto verso il basso.???? Da quel che ho capito si, l'unico modo che ho per non leggere in maniera sequenziale dall'alto verso il basso è fare un salto con qualche branch.

La base della fetch consiste semplicemente nel prendere 32 bit in memoria che verranno poi processati.

Sul risc-v troviamo poi vari tipi di istruzioni:

R → aritmetiche con operandi di solo registri.

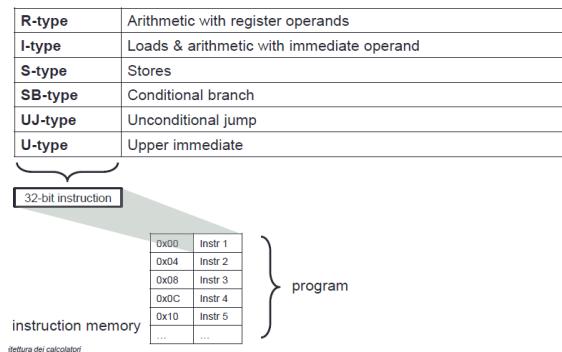
I → aritmetiche dove uno degli operandi è immediato; oppure una load:

l'offset è un immediato a tutti gli effetti; quindi ho registro, immediato * registro

S → store

SB e UJ → 2 branch uno di tipo condizionale e uno non condizionale

U → per operazioni più grandi...



RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
 - opcode**: operation code
 - rd**: destination register number
 - funct3**: 3-bit function code (additional opcode)
 - rs1**: the first source register number
 - rs2**: the second source register number
 - funct7**: 7-bit function code (additional opcode)

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
add x9, x20, x21					
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

these values come from encoding tables (see slides 51 - 53)

0000 0001 0101 1010 0000 0100 1011 0011_{two} = 015A04B3₁₆

32 bit che rappresentano l'istruzione.

La fetch quando la prende in memoria (vado a prendere in memoria la prima istruzione che devo svolgere in questo momento), la prima cosa che devo fare è capire di che **classe di istruzione** stiamo parlando, per farlo guardo i campi speciali:

- gli ultimi 7 bit sono **l'opcode** e codificano la famiglia dell'istruzione;
- dopo aver trovato la sottofamiglia, so dove andare a trovare le restanti informazioni(guardo com e è codificata la stringa di 32 bit di quella famiglia di istruzione);

In questo caso(**R**) guardo i campi **func3** e **func7**; questi campi servono per **specificare** l'istruzione; perchè anche se so che è aritmetiche, ci sono vari tipi... di solito quello che ti fa beccare subito il tipo di istruzione è sempre func3 (una volta che individui l'opcode).

rd → registro di destinazione

gli **r** sono grandi 5 bit, perchè? → **perchè ho 32 registri** → da 0 a 31 e per rappresentare questi numeri ho bisogno di 5 bit($(2^5)-1$)

Alcuni tipi di istruzioni condividono lo stesso **opcode** e anche **funct3** → qui entra in gioco funct6/7 (ad esempio si guardi la differenza tra le istruzioni **add** e **sub**) → tabella finale

L'istruzione viene codificata in binario e dopo in esadecimale. L'esadecimale, a detta del prof, è una versione comoda e compatta del binario se ti ricordi a memoria i numeri in binario da 0 a 15.

RISC-V I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- **Immediate arithmetic and load instructions**
 - *rs1*: source or base address register number
 - *immediate*: constant operand, or offset added to base address
 - 2s-complement, sign extended (see next slide)
- **Design Principle 3: Good design demands good compromises**
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: **0000 0010** => **0000 0000 0000 0010**
 - -2: **1111 1110** => **1111 1111 1111 1110**

Se riconosco un **I** dall'**opcode**, i 5 bit successivi non sono più func3, ma sono rd; quindi cambia la codifica e la lettura!

Il numero nel campo immediato è un numero in **complemento a 2 sign-extended**, però ho 8 bit ????? perchè ho 8 bit??? non ne ho 12??? @maxbubblegum47
→ se si parte con un numero rappresentabile con 8 bit, non ci stanno → si usa la sign extension(fino a 12) → non ho capito????

Maxbubblegum: "io mi sono segnato che sto sempre usando un numero di 12 bits immediato, in complemento a 2, con segno esteso (i bit + a sinistra indicano quello che riguarda il segno in maniera più precisa). Non ho mai letto questa cosa degli 8 bit per l'immediato, forse l'esempio che fa lui nelle slide che hai riportato qua sopra mi sembra, ma potrei benissimo cannare, che sia una cosa del tipo: stiamo passando da 8 bit a 16 bit ed essenzialmente per farlo "allungo" il numero usando l'ultimo bit che trovo sulla sinistra (che è il bit del segno) fino a che non arrivo a 16 bit. Ho visto ora che lo scrivi

anche tu dopo per quel che riguarda il padding. Comunque ora vedo se nel manuale dice qualcosa per quel che riguarda gli immediati a 8 bit. Sto vedendo, ma per ora non trovo nulla; se trovo qualcosa ti aggiorno". @pablo remirez



con la sign extension faccio **padding** di soli 0 se ho un numero positivo, mentre se è negativo di soli 1 → aggiungendo tanti 0 a sinistra in un numero positivo, non cambia nulla
mentre in un numero negativo devo aggiungere tanti 1 a sinistra; in questo modo sto aggiornando il numero negativo che verrà sottratto(siamo in complemento a 2, il bit più significativo è negativo)

IMPORTANTE

il campo immediato della beq(non so se vale per gli immediati in generale) viene shiftato a sinistra di 1 → perchè?

Ad esempio 128 ha bisogno di 8 bit → quindi non ci sta con il complemento a 2 → ho bisogno di altri bit per cambiare il segno → metto i restanti 4 bit tutti a 0 per dire che il mio 128 è positivo e non negativo ?????

Si cerca di mantenere le posizioni dei bit vincolati alle stesse funzionalità per semplificare il tutto → **opcode è rappresentato sempre dagli ultimi 7 bit!** Questo perché voglio andare a riutilizzare la logica che ho creato il più possibile.

I-format Example 1: Immediate arithmetic

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
addi x10, x10, 128				
				<i>// second operand is an immediate (see slides 51 - 53)</i>
128	10	0	10	19
0000 1000 0000	01010	000	01010	0010011
0000 1000 0000 0101 0000 0101 0001 0011 _{two} =				
08050513 ₁₆				

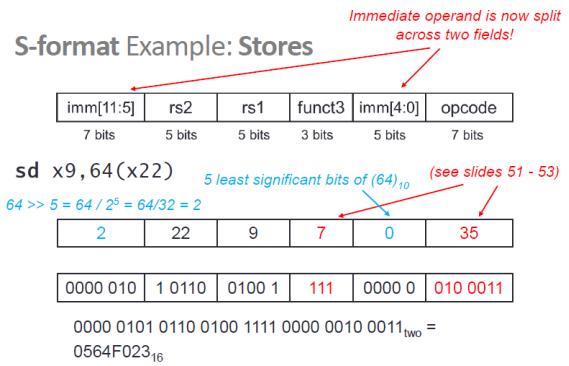
I-format Example 2: Loads

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
ld x9, 64(x22)				
				<i>(see slides 51 - 53)</i>
64	22	3	9	3
0000 0100 0000	10110	011	01001	0000011
0000 0100 0000 1011 0011 0100 1000 0011 _{two} =				
040B3483 ₁₆				

RISC-V S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for **store instructions**
 - rs1**: base address register number
 - rs2**: source operand register number
 - immediate**: offset added to base address
 - Split so that **rs1** and **rs2** fields always in the same place



il primo operando viene dal **RF** (register file) ed è rs1 che in questo caso è un **9**
 l'altro operatore è 64 che è l'offset
 il secondo registro vado a prendere la base per l'indirizzo che ottengo,
 moltiplicato per 64 → che è **22** → 64(x22)

L'offset è molto comodo per gli array (*dobbiamo stare sempre attenti però quando ce lo facciamo in assembly perché è davvero una piaga*): se io metto dentro a x22 l'indirizzo dell'array x22 = &A[0]; dopo posso accedere agli elementi dell'array, tramite un offset → **Id x10, 0(x22)**

se no dovrei fare una add prima: **add, x22, x22, 8** → questa mi servirebbe per calcolare esattamente dove andare a fare la load; l'offset lo faccio con la somma; con l'offset invece non devo farlo e posso utilizzare direttamente x22

metto 8 perchè un offset di 0 vuol dire il primo elemento → 8 byte??? Sisi sono 8 bytes, ti ho messo la parte del manuale in cui ne parla. In pratica il byte address dell'array è sempre doble word, quindi sempre 8 bit; se mi sposto di un elemento all'altro su un array passo sempre di 8 in 8. @maxbubblegum47

Code for the Body of the Procedure swap

The remaining lines of C code in swap are

```
temp    = v[k];
v[k]    = v[k+1];
v[k+1] = temp;
```

Recall that the memory address for RISC-V refers to the *byte* address, and so doublewords are really 8 bytes apart. Hence, we need to multiply the index k by 8 before adding it to the address. *Forgetting that sequential doubleword addresses differ by 8 instead of by 1 is a common mistake in assembly language programming.*

x22 è un registro che contiene l'indirizzo di A[0] → moltiplicato per 64 ottengo A[8] e ci salvo dentro il numero contenuto in x9

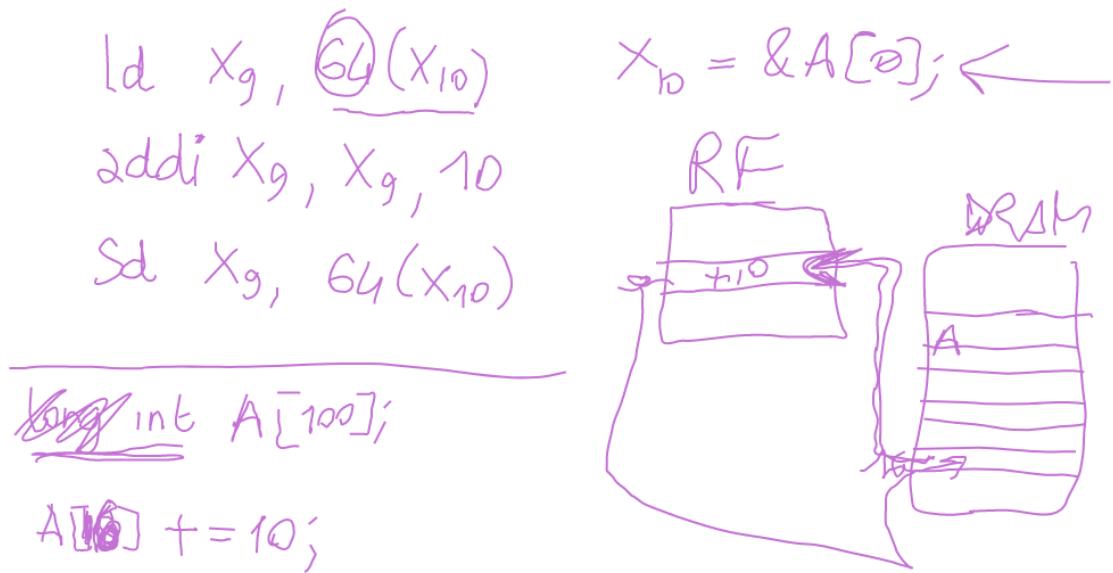
Nella solita scrittura ld x9, 64(x10) e sd x9, 64(x10): X9 nella load è un registro su cui va scritto quello che leggo da DRAM (è un saved register); qui anche se nella codifica si trova scritto **rs**(register source); **rs1** è il register source che contiene l'**indirizzo di DRAM dove andare a storare**(salvare) il dato contenuto dentro a **rs2**(che si comporta come un normale rs).

→ nella store non esiste un campo **registro di destinazione**, perchè la destinazione è dentro la DRAM, non nei registri.

→ nella store non scrivo sul registro, quindi x9 è il registro di partenza, non quello di destinazione(nel nostro esempio)

Quindi l'operazione scritta sopra scrive quello contenuto nel registro x9, dentro alla posizione in DRAM (x10[8])

▼ immagine



RF(register file) è il registro di destinazione. Nel primo caso ho una freccia che entra e una che esce; nel secondo caso ne ho solo una che esce???? Maxbubblegum:
"Si tipo vedi che prima faccio la load e mi vado a prendere dalla DRAM quello che mi serve e poi lo vado a risalvare quando vado a fare la store. Poi l'elemento in cui va a fare l'operazione dipende dal tipo di array, perché se siamo sul long int allora siamo sull'elemento numero 8 altrimenti 16. Lui prima aveva fatto l'esempio con l'array tipo long int e poi solamente int. Questa cosa come vedi non cambia il codice assembly, però cambia MOLTISSIMO il codice in C e quindi il risultato che ci aspettiamo alla fine". @pablo remirez

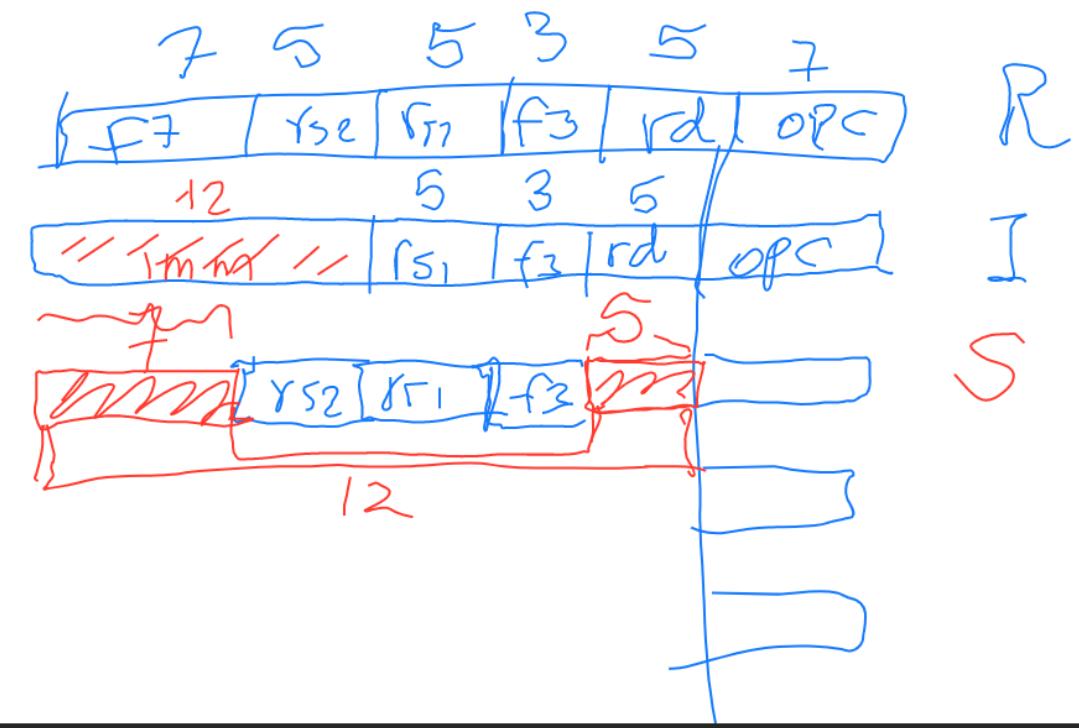
Nel tipo R ho tutto bello organizzato in cui costruisco la mia circuiteria...nel tipo I cosa faccio? Cerco di tenere il più possibile in comune → l'opcode è sempre li; rd, f3 e rs1 è

sempre li; cambiano **f7** e **rs2** che diventano **immediate**. L'idea è sempre quella di andare a risparmiare quanta più circuteria possibile.

Nel tipo S invece cambia tutto → opcode c'è sempre; f3, rs1 e rs2 ci sono ancora → però adesso **ho ancora 12 bit immediate, ma sono sparsi, divisi in 7 al lato e 5 in mezzo**, quindi vanno messi insieme e questo complica.

I primi 5 bit sono in mezzo e i restanti 7 sono i più a sinistra e si usa questo metodo perchè si cerca di usare il più possibile di riuso, pur di non creare una nuova logica, si fa così → insieme di blocchi in cui cerco di ottenere il massimo riuso. più codifica diversa

▼ immagine



More Load/Store Operations: Byte/Halfword

- RISC-V byte/halfword **load/store**
 - Load byte/halfword: Sign extend to 64 bits in rd
 - lbu rd, offset(rs1)
 - lh rd, offset(rs1)
 - Load byte/halfword unsigned: Zero extend to 64 bits in rd
 - lbu rd, offset(rs1)
 - lhu rd, offset(rs1)
 - Store byte/halfword: Store rightmost 8/16 bits
 - sb rs2, offset(rs1)
 - sh rs2, offset(rs1)

More Load/Store Operations: Word/Doubleword

- RISC-V word/doubleword **load/store**
 - Load word/doubleword: Sign extend to 64 bits in rd
 - lw rd, offset(rs1)
 - ld rd, offset(rs1)
 - Load word/doubleword unsigned: Zero extend to 64 bits in rd
 - lwu rd, offset(rs1)
 - ldu rd, offset(rs1)
 - Store word/doubleword: Store rightmost 32/64 bits
 - sw rs2, offset(rs1)
 - sd rs2, offset(rs1)

lui(load upper immediate):

More Load/Store Operations: 32-bit Constants

- Most constants are small
 - 12-bit immediate is sufficient
- For the occasional 32-bit constant **we need two instructions:**
 - Load upper immediate (lui)**
lui rd, constant
 - Copies 20-bit constant to bits [31:12] of rd
 - Extends bit 31 to bits [63:32]
 - Clears bits [11:0] of rd to 0
 - Any other instruction that populates lower bits [11:0]
 - e.g., **addi**

More Load/Store Operations: 32-bit Constants

Example: we want to copy to x19 this 64-bit constant

00000000 00000000 00000000 00000000 00111101 00000101 00000000

bits [31:12] bits [11:0]

lui rd, constant

- Copies 20-bit constant to bits [31:12] of rd
- Extends bit 31 to bits [63:32]
- Clears bits [11:0] of rd to 0

1) lui x19, 976 // 0x003D0

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000 0000
---------------------	---------------------	--------------------------	---------------------

2) addi x19,x19,1280 // 0x500

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000
---------------------	---------------------	--------------------------	----------------

E' un meccanismo che ci serve a volte quando abbiamo bisogno di lavorare con lavori numerici **più grandi di 12 bit** → nel caso di load e store ho 12 bit per la immediate, quindi **posso rappresentare massimo ($2^{12}-1$)** rappresentazioni. Come faccio per i casi con costanti più grandi? → meccanismo per **impacchettare una costante a 32 bit** → 2 istruzioni per farlo:

- I 12 bit più bassi sappiamo già come lavorarci: operazione per popolarli, ad esempio **addi** o le istruzioni immediate.
- Per i restanti come facciamo? **lui: prende una costante a 20 bit e la sposta nei 20 bit più significativi del registro destinazione; poi pulisce i 12 bit a destra.** Però io non sto semplicemente copiando questi 20 bit dentro al registro, ma li copio nei **20 bit più significativi**, quelli a sinistra **dal bit 31 al bit 12 e mette a 0 quelli a destra.**

Questa cosa serve quando vuoi usare una costante più grande di 12 bit(**ma minore di 32**) → perchè gli immediate arrivano fino a 12 bit → **se usi una variabile allora no problem perchè i registri hanno 64 bit (abbiamo 32 registri da 64 bit ciascuno).**



sono **32** registri a **64bit**

✓ perchè uso istruzioni da 32bit allora? convenzione? Noi praticamente stiamo studiando al versione base a 32 bit perché è decisamente più comodo lavorarci su, se no dovremmo avere a che fare con tipi di operazioni che gestiscono più bit. Immaginati questo primo screenshot x 2. @pablo remirez

✓ non ho capito il secondo screenshot, me lo spieghi domani magari per il resto si, è più comodo o almeno noi stiamo studiando quello; indatti se guardi nell'esempio delle lui qui → si vede che carica solo i primi 32bit, perchè evidentemente è una codifica migliore per le istruzioni; ma ha altri 32bit di registro. quando usi una variabile occupi tutti i 64bit??

@maxbubblegum47

Format		Bit																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register		funct7					rs2			rs1			funct3			rd			opcode														
Immediate		imm[11:0]					rs1			funct3			rd			opcode																	
Upper immediate		imm[31:12]					rs1			funct3			rd			opcode																	
Store		imm[11:5]					rs2			rs1			funct3			imm[4:0]			opcode														
Branch	[12]	imm[10:5]					rs2			rs1			funct3			imm[4:1]		[11]	opcode														
Jump	[20]	imm[10:1]					[11]		imm[19:12]					rd			opcode																

ISA base and extensions

Name	Description	Version	Status ^[a]	Instruction Count
Base				
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified	49
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open	49
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified	14
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	14

L'operazione con immediato non potrà mai scrivere più di 12 bit → tu prima fai la lui dei 20 bit poi a quel registro ci fai addi con la costante(quella minore di 13 bit)

I 12 bit di immediato per le store sono sparpagliati → si usa la **IMM immediate generation unit** per sistemarli → è un blocco logico creato apposta per generare gli immediati → perchè lo si fa? perchè vogliamo mantenere il riuso...

Operazioni logiche bit a bit

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xorl
Bit-by-bit NOT	~	~	xori FF..F

- Useful for extracting and inserting groups of bits in a word

Uno shift a sinistra equivale ad una moltiplicazione a sinistra.

slli = shift left logical immediate → l'immediato è il numero che dice di quante posizioni vogliamo shiftare la parola. Anche loro hanno una codifica simile alle altre, perchè **anche loro sono istruzioni immediate**. Fino ai 6 bit del

Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- I-format** with just 6 bits for immediate
- immed**: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - slli** by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srl** by i bits divides by 2^i (unsigned only)

campo immediato è uguale alla **addi**. Cambia la larghezza del campo immediato perchè su uno shift non si usano dei numeri enormi di solito. Qui ho 6 bit di immediato e 6 bit di campo funct6
→ **funct6 è molto grande perchè ho tante istanze diverse e quel campo serve per specificare che istanza ho.**

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00001101 00000000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or x9,x10,x11

x10	00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation
 - Set some bits to 1, leave others unchanged

xor x9,x10,x12 // NOT operation

x10	00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Istruzioni di tipo branch:

2 tipi: condizionali e non condizionali

Condizionali:

Consistono nel fare un salto ed andare ad una istruzione che si identifica con una **label** (è un po' come gli if e gli else in programmazione ad alto livello); quindi **fa**

saltare il program counter

all'istruzione con quella label → il concetto è lo stesso della funzione GO TO.

beq → è se s1 è uguale a s2; allora salta a branch L1 → L1 è la label di un'istruzione

bne → è se i 2 registri sono diversi

Nell'esempio la prima cosa che faccio è verificare l'uguaglianza tra i e j:

- se sono diversi, allora salta all'istruzione con label: Else → che è la sub
- se sono uguali allora continua e fa la add

Conditional Operations

- › Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially

- › **beq rs1, rs2, L1**
 - if ($rs1 == rs2$) branch to instruction labeled L1
- › **bne rs1, rs2, L1**
 - if ($rs1 != rs2$) branch to instruction labeled L1

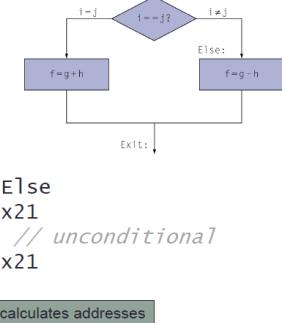
Compiling If Statements

- C code:

```
if (i==j) f = g+h;  
else f = g-h;  
... f, g, ... in x19, x20, ...
```

- Compiled RISC-V code:

```
bne x22, x23, Else  
add x19, x20, x21  
beq x0,x0,Exit // unconditional  
Else: sub x19, x20, x21  
Exit: ...
```



Come si attacca una label ad una istruzione?? Se ne parlerà dopo, ma in generale l'assembly è un linguaggio human readable e usa opzioni label: sono tradotte come un offset rispetto ad un program counter → in pratica la label è un numero che è un immediato che è un offset positivo o negativo rispetto al program counter → salto in avanti o in indietro di x posti

La branch ha un numero di bit limitato per il campo immediate → non è possibile fare salti infiniti → se ci vuole un salto troppo grande si usa altro...

La prima istruzione mi serve per **spostare a sinistra di 3 posizioni** → **moltiplicare di 8** (3 posizioni equivale a moltiplicare per 2^3); questo mi serve per **calcolare l'offset dell'elemento i** → non posso aggiungere ad x25 semplicemente un 1 o 2. **Devo aggiungerci i byte esatti** per passare all'elemento successivo e noi siamo in long int quindi 8 byte.

8 sono i byte di 1 elemento dell'array save; il tipo di dato dell'array save occupa 8 byte; per questo shift di 3 (long int save[]) → quindi mi serve un

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;  
    i in x22, k in x24, address of save in x25
```

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3  
      add x10, x10, x25  
      ld x9, 0(x10)  
      bne x9, x24, Exit  
      addi x22, x22, 1  
      beq x0, x0, Loop  
Exit: ...
```

**offset da aggiungere all'indirizzo base dell'array
save per accedere alla cella giusta. ovvero
avanzare di i posizioni.**

Se io avessi avuto un int save[] → allora avrei dovuto shiftare di 2 → perchè int= 4 byte. La seconda istruzione equivale all'indirizzo di save[i] → **sarebbe $i*8 + \&save$** → ($i*8$ l'ho calcolato prima) la terza istruzione è una load double → **mette dentro ad x9 il risultato della lettura in DRAM di 0(x10)** che è **x10 per un offset di 0** → è esattamente x10 → save[i].

La quarta istruzione fa il controllo con una branch → se x9 è diverso da x24 → allora salta all'istruzione con label exit; se no continua. La quinta istruzione è una add che somma 1 al registro x22 che è i

La sesta istruzione è un **branch non condizionale**, perchè controlla che x0 sia = a x0 → quindi sempre; e fa saltare all'istruzione con label loop.

Diamo per scontato che in x22 ci sia 0!!!

Nelle load dobbiamo sempre mettere l'offsett perchè ce lo chiede la sintassi → quindi **se non abbiamo array o roba simile ci mettiamo semplicemente 0(registro);** ovviamente non si può fare l'operazione di shift al post del'offset perchè risulterebbe in una locazione di memoria diversa, non uno shift.

blt (branche lower than) e bge (branch greater or equal)

More Conditional Operations

- › **blt rs1, rs2, L1**
 - if ($rs1 < rs2$) branch to instruction labeled L1
- › **bge rs1, rs2, L1**
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- › **Example**
 - if ($a > b$) $a += 1;$
 - a in x22, b in x23
 - bge x23, x22, Exit // branch if $b \geq a$**
 - addi x22, x22, 1**
 - Exit:**

Signed vs. Unsigned

- Signed comparison: **blt, bge**
- Unsigned comparison: **bltu, bgeu**
- Example
 - $x22 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x23 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x22 < x23 // signed$
 - $-1 < +1$
 - $x22 > x23 // unsigned$
 - $+4,294,967,295 > +1$

il maggiore stretto > e il minore uguale ≤ non esistono! → per implementarli si

cambia la logica

Codifica istruzione della famiglia SB → branch

Opcode come al solito; 2 registri; una funct3 e 1 immediato → formato abbastanza simile; ma le cose si complicano ancora di più.
opcode rs2,rs1 e funct3 sono immutate.

L'immediato è molto diverso → metto prima il bit 11 del campo immediato; poi da 1 a 4; poi da 5 a 10 infine il bit 12 perchè?
→ questo campo offset viene sommato al program counter PC; quindi **sono sommati al program counter attuale**.

L'immediato ha 12 bit che rappresentano un numero positivo o negativo che rappresenta un salto che bisogna fare nel PC:
positivo in avanti, negativo indietro.

Il terzo parametro della branch che è la label, **in linguaggio macchina è il calcolo di quante posizioni ci sono da quell'istruzione branch a l'istruzione con quella label**

nell'es del while nel loop abbiamo in verità scritto -20
→ **ogni istruzione è 4 byte quindi $0x14 = 16 + 4 = 20$ e io devo tornare a 0x00**

quindi le istruzioni sono di 32bit → 4 byte e lo devi tenere a mente quando lavori sul PC
mentre i registri sono 64bit → 8 byte e lo devi tenere a mente quando lavori su memoria

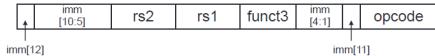
Jump and link

Istruzione di branch NON condizionale → utilizzo 20 bit di immediato → **serve quando ho bisogno di salti molto grandi**(branch condizionali fanno salti più piccoli)

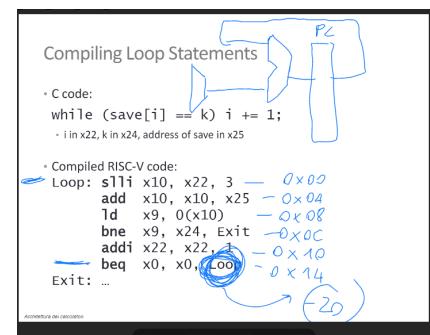
RISC-V SB-format Instructions (branch addressing)

- Branch instructions specify
 - Opcode, two registers, target address

- Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = PC + immediate × 2
(Addressing instructions down to halfword)



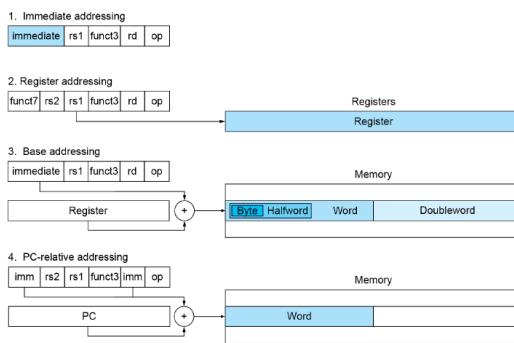
A destra ho i soliti 7 bit di opcode e seguono 5 bit di registro di destinazione; **quest'ultimo serve per implementare la chiamata di funzione: quando chiamo una funzione non posso dimenticarmi da dove l'ho chiamata**
→ devo memorizzare il PC successivo a quello di dove avviene la chiamata.
jump = salto all'offset di immediato.
link = tengo in rd la prossima istruzione che non è altro che il PC attuale + 4

si salta di 4 byte perchè per rappresentare un'istruzione ci vogliono 32 bit → 4 byte

Sia qui, sia nelle branch l'immediato è indicizzato a partire dall'1

Esistono i long jump che ci permette di saltare in qualunque parte del nostro PC → come si fa? Ho bisogno di una costante a 32 bit da dare → usiamo **lui** che abbiamo già visto.

RISC-V Addressing Summary



RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	l.r.d	0110011	011	0001000
I-type	sc.d	0110011	011	0001100
	opcode is same for a family of instructions		which get differentiated by the funct3 bits	and the funct6/7 bits

RISC-V Jump/Link Instructions (Jump addressing)

- Jump and link (**jal**) uses 20-bit immediate for larger range



- For long jumps, eg, to 32-bit absolute address
 - lui**: load address[31:12] to temp register
 - jalr**: add address[11:0] and jump to target

*PC → ...
PC + 4 → foo()..*

RISC-V Encoding Summary

Name (Field Size)	7 bits	Field 5 bits	Field 5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immed[11:0]	rs2	rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
Sb-type	immed[12:10:5]	rs2	rs1	funct3	immed[4:1:11]	opcode	Conditional branch format
UJ-type	immed[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	1b	0000011	000	n.a.
	1h	0000011	001	n.a.
	1w	0000011	010	n.a.
	1d	0000011	011	n.a.
	1bu	0000011	100	n.a.
	1hu	0000011	101	n.a.
	1wu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srlti	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UI-type	jal	1101111	n.a.	n.a.

indice

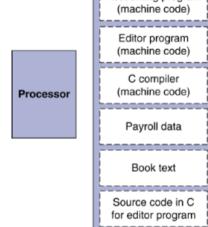
Procedure calling

Normalmente la memoria contiene diverse porzioni di codice:

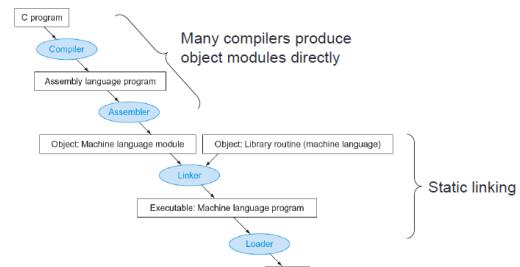
il sorgente del nostro programma C, che in realtà è interpretato in caratteri, ci vuole quindi un compiler sempre contenuto nella memoria e altre cose...

Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs



Translation and Startup



Per quanto riguarda la scrittura di un programma in C:

- utilizzo un compilatore per produrre assembly.
 - assembler per produrre codice macchiana.
 - linker attacca all'eseguibile del mio programma, i codici oggetto delle librerie.
- Il linking può essere statico o dinamico → quello statico è quello di cui parliamo.

Quando includo una libreria staticamente, il linker fa una cosa simile al jump → il linker ha un offset che dice dove andare a prendere la funzione root dentro al file oggetto math.txt

Linking dinamico = funziona a runtime → il linker include dei riferimenti dentro al mio programma che a runtime si risolveranno

L'assembler non produce solo il codice macchina; ma produce anche delle informazioni che servono per mettere insieme il tutto:

I'header = maxbubblegum: "header tipo come in c++?" @pablo remirez → esattamente @maxbubblegum47

Static = contiene dati che sono l'equivalente di variabili globali o dichiarate statiche nel programma c

→ però deve essere una parte del Programma che contiene questi valori.

Relocation info e symbol table = per tutte le funzioni che sono esterne;

quando creiamo il file oggetto, ci sono dei simboli che sono esterni al file → quindi creo una tabella dove codifico questi simboli dove ci sono le informazioni con dei riferimenti.

Però non avvengono subito, per questo si scrive la tabella, perchè si farà dopo piano piano → man mano che li incontro????

Una variabile normale → nasce e muore con la funzione; quindi non sono da tenere conto, sono nello stack

→ lo stack si popola con le funzione e al termine della funzione ripulisce tutto(variabili comprese)

Mentre quelle globali o particolari ci sono sempre, quindi ci vuole **dynamic data(heap)**.

Lo stack è la memoria automatica → cresce e decresce continuamente.

Debug info = inserisce all'interno del nostro codice, tanti simboli che servono per interrompere l'esecuzione con break point o che dicono a quale linea di codice appartiene un dato ecc... → serve per il debugger.

Nel momento in cui faccio il linking, se è statico: tutta l'informazione la risolvo prima della risoluzione del programma; prendendo i

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - **Header:** described contents of object module
 - **Text segment:** translated instructions
 - **Static data segment:** data allocated for the life of the program
 - **Relocation info:** for contents that depend on absolute location of loaded program
 - **Symbol table:** global definitions and external refs
 - **Debug info:** for associating with source code



vari ELF fatti e gli si mette assieme.

→ gli ELF sono i vari file oggetto che vengono linkati per creare l'eseguibile finale

Fase di loading

- ho risolto tutti i riferimenti e link
- carico il programma in DRAM e popolo il **PC** e lo **stack pointer**
- alloco la memoria in giro
- inizializzo le strutture dati globali.
- carica nello stack eventuali argomenti passati con la chiamata del programma.
- altre cose...
→ **in pratica il loader predisponde le risorse per far funzionare il programma.**

prima del main, il load chiama una funzione startup che copia argomenti??? e infine chiama il main e quando il main finisce, chiama **syscall exit**

Procedure calling

- popolare i registri x10 - x17 con i parametri
- allocare memoria per la procedure
→ in realtà è più un gioco di simmetria dei puntatori per gestire in maniera trasparente le stack, più che vera e propria allocazione

RISC-V procedure Calling

- Steps required
 - 1. Place parameters in registers x10 to x17
 - 2. Transfer control to procedure
 - 3. Acquire storage for procedure
 - 4. Perform procedure's operations
 - 5. Place result in register for caller
 - 6. Return to place of call (address in x1)

come si effettuano queste cose?

Jump and link

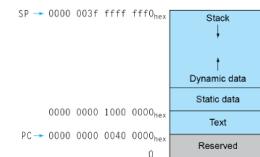
operazioni che ci servono per implementare la chiamata e il ritorno da una funzione:

x1 ospita il valore del PC dove bisogna ritornare → quindi PC + 4 perchè torno all'istruzione successiva

alla jal di solito si passa x1 e la label che

Linking Object Modules

- Produces an executable image
 - 1. Merges segments
 - 2. Resolve labels (determine their addresses)
 - 3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this
 - Program can be loaded into absolute location in virtual memory space



identifica la procedura a cui andare. questo lo faccio perchè mi serve un indirizzo per tornare indietro al normale flow del PC e si usa x1 per convenzione Quando ritorno dalla procedura, chiamo la **jalr** → utilizza un indirizzo base + offset.

Nella jal è procedure label l'offset per il PC → la label è in pratica un offset → la macchina non legge una parola, ma legge un codice binario, in questo caso la label si traduce in un PC+qualcosa. mentre nella jalr io utilizzo proprio un offset + indirizzo base.

→ nella jal mette da solo PC+4 quando gli passi x1

Si usano rispettivamente per implementare **call** e **return**.

alla jalr si passa x0 di solito perchè è costante a 0 → voglio dire che non voglio salvare un indirizzo di ritorno, perchè la jalr la uso per tornare indietro, non mi interessa tenere l'indirizzo di memoria dell'istruzione successiva; sono alla fine di una funzione.

potrei anche mettere in un altro registro, ma per convenzione si passa x0 per implementare la return.

La "l" di una jump and link → sta a significare che questa funzione fa il linking

linking = questa funzione mette PC + 4 dentro ad x1 → così posso tornare all'indirizzo successivo; se io non devo tornare indietro → PC+4 non è utile.

ovviamente **non è il valore di ritorno**

Procedure Call Instructions

- Procedure call: jump and link


```
jal x1, ProcedureLabel
```

 - Address of following instruction put in x1
 - Jumps to target address (ProcedureLabel)
- Procedure return: jump and link register


```
jalr x0, 0(x1)
```

 - Like jal, but jumps to 0 + address in x1
 - Use x0 as rd (x0 cannot be changed)
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Unconditional branch	Jump and link	<code>jal x1, 100</code>	<code>x1 = PC+4; go to PC+100</code>	PC-relative procedure call
	Jump and link register	<code>jalr x1, 100(x5)</code>	<code>x1 = PC+4; go to x5+100</code>	Procedure return; indirect call

**della funzione; è l'indirizzo di ritorno
dello stack**

Jal si usa per chiamare una funzione ; jalr si usa per tornare al punto chiamante

- Nella jal io ho un PC + offset; offset me lo da quello che metto dentro a procedureLabel → (perchè label sono degli offset in realtà...) nella jalr io metto un indirizzo + offset → devo specificare offset; di solito ci metto x1 che ha dentro il PC + 4 messo li dentro tramite la jal
- Nella jalr devo per forza specificare un offset.

Si usa x1 perchè così sono sicuro che in x1 tutti diano per scontato che metto l'indirizzo di ritorno (è una convenzione).

Procedura foglia

Leaf Procedure Example

A procedures that doesn't call other procedures

- C code:

```
typedef long long int lli;
lli leaf_example (lli g, lli h, lli i, lli j) {
    lli f;
    f = (g + h) - (i + j);
    return f;
}
```

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack (spill to mem)

RISC-V Registers	
>	x0: the constant value 0
>	x1: return address
>	x2: stack pointer
>	x3: global pointer
>	x4: thread pointer
>	x5 – x7, x28 – x31: temporaries
>	x8: frame pointer
>	x9, x18 – x27: saved registers
>	x10 – x11: function arguments/results
>	x12 – x17: function arguments

Tuttura dei calcolatori

Leaf Procedure Example

RISC-V code:

```
addi sp,sp,-24           Save x5, x6, x20 on stack
sd x5,16(sp)
sd x6,8(sp)
sd x20,0(sp)
add x5,x10,x11          x5 = g + h
add x6,x12,x13          x6 = i + j
sub x20,x5,x6            f = x5 - x6
addi x10,x20,0
ld x20,0(sp)              copy f to return register
ld x6,8(sp)               Restore x5,x6,x20 from stack
ld x5,16(sp)
addi sp,sp,24
jalr x0,0(x1)
```

Return to caller

Se guardo il grafo delle chiamate delle funzioni → questa funzione si troverà nei nodi foglia(quelli alla fine) → non chiamerà altre funzioni.

Freindly Reminder: *long long int = 64 bit* → se fossero stati int avrei dovuto dividere per 2 gli offsett delle sd e ld???

quindi anche se i registri sono da 64bit io cambio l'offsett in base al tipo di dato che ho x5 e x6 vanno salvati perchè sono registri temporanei, non so se chi chiama la funzione ha messo roba utile in x5 e x6; quindi salvo per sicurezza.
salvare = metto in DRAM, faccio una sd essenzialmente. → poi farò una load a fine funzione.

Se fossi io nel main a decidere, allora potrei decidere di non salvare x5 e x6, perchè gli userei solo per la somma; ma non sapendo chi la usa, io sto implementando una funzione

→ salvo i 2 registri perchè non so se ci sono dati utili o no.

Questo lo si fa spesso quando si crea una libreria...

Dentro alla DRAM c'è uno spazio riservato ad ogni processo, quell'area della DRAM è riservata per lo stack di ogni funzione.

sp = **stack pointer** = x2 = il puntatore alla cima dello stack. → se voglio gestire esplicitamente sullo stack, allora uso sp.

Faccio -24 perchè? lavoriamo su long long int → 8 byte. → io devo salvare 3 registri → 8*3 byte = 24 byte → riservo 24byte sullo stack pointer; facendolo crescere verso il basso. Poi parto dalla posizione 0 a scendere fino alla 24 quando carico/scarico.

La seconda istruzione scrive x5 su stack pointer con offset 16; quindi il più alto possibile...

Lo stack pointer può essere ovunque, di solito è nella DRAM; ma tu puoi anche dire che lo sp è dentro ad una memoria più veloce.

Da questo punto in poi sono intitolato a scrivere su x5, x6 e x20 → quindi faccio l'operazione usando i registri e mettiamo il risultato su x10.

Per confermare che quello che mi ha passato il chiamante resti → devo usare le load in modo duale.

ripristino lo stack pointer → dicendo che il top è tornato al valore di prima della funzione → lo faccio aggiungendo 24; questo meccanismo è il famoso "tempo di vita" della funzione.

Ultima è la jalr in x0 perchè, non ci interessa di salvare l'indirizzo e torniamo su x1 → (PC+4)

→ in pratica prima di eseguire operazioni, io salvo i valori che c'erano dentro alle variabili sullo stack pointer; poi faccio tutto e infine ricarico i valori in quelle variabili. questo è il meccanismo per non fare influenzare i dati che diamo dentro alla funzione; perchè prima salvo i valori poi prima di uscire gli ricarico.

Per variabili si intende i registri; questo perchè in x5 e x6, magari ci sono dei valori importanti che non vanno modificati.

Registri temporanei e registri saved.

Anche i saved sono temporanei, ma i saved sono gestiti diversamente:

- i registri temporanei non sono preservati dal chiamante(caller), sono volatili tra le chiamate; il chiamante se vuole essere sicuro di non avere valori modificati dentro ai registri temporanei, deve salvare i registri.

Register Usage – Calling convention

- x5 – x7, x28 – x31: temporary registers
 - Not preserved by the callee (volatile across calls, must be saved by the caller if later used)

- x8 – x9, x18 – x27: saved registers
 - Preserved across calls. If used, the callee saves and restores them

- In previous example, the stores/loads on x5 and x6 can be dropped

caller: who calls a function
callee: the function itself

RISC-V Registers
> x0: the constant value 0
> x1: return address
> x2: stack pointer
> x3: global pointer
> x4: thread pointer
> x5 – x7, x28 – x31: temporaries
> x8: frame pointer
> x9, x18 – x27: saved registers
> x10 – x11: function arguments/results
> x12 – x17: function arguments

- i saved register invece sono preservati tra chiamate; perchè per convenzione chi scrive la funzione, li salva assicurandoli.

Quindi apprendiamo adesso che noi non avremmo dovuto salvare i nostri, perchè era responsabilità del chiamante assicurarli.

In breve:

- se ho bisogno di pochi registri, uso quelli temporanei e non gestisco il salvataggio
- se ho bisogno di tanti registri e mi tocca usare quelli saved → **dentro la funzione devo gestire i salvataggi vari.**

Non c'è assolutamente bisogno di salvare sullo stack l'indirizzo di ritorno della funzione, soprattutto perchè non chiamava nessun'altra funzione.

Nel caso in cui la funzione non fosse leaf → chiama un'altra funzione → allora è necessario salvare il valore di x1 → perchè poi servirà alla nuova funzione per tornare a quel punto del processo.

Funzione ricorsiva = esempio più difficile di funzione non leaf:

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
 - Not the saved registers
 - Those are handled by the *callee* (if used)
- Restore from the stack after the call

• C code:

```
long long int fact (long long int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}

• Argument n in x10
• Result in x10
```

Preambolo: come prima allochiamo spazio per due variabili long long → x1 che è il valore di ritorno di questa funzione (senza non posso tornare indietro) e x10 che è il valore di n.

A sto punto funzione vera e propria → uso x5 un temporaneo, non lo salvo.

Uso un branch che mi fa saltare alla parte del loop che fondamentalmente è la ricorsione.

Metto in x10 il valore 1.

Ripristino lo stack pointer, senza nemmeno fare le load → perchè tanto non ci ho ancora scritto dentro.

jalr torno ad x1.

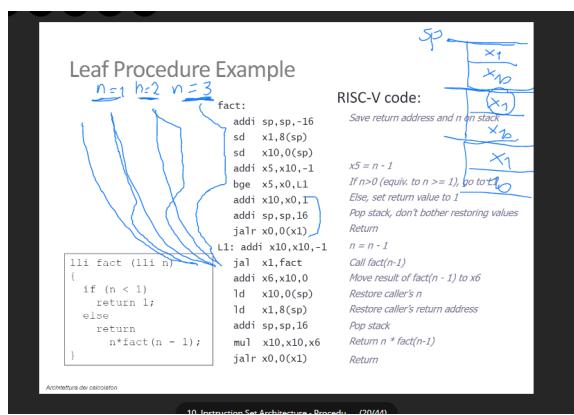
Ricorso → metto in x1 il valore -1 → perchè devo chiamare fact di n-1 → lo devo mettere io.

jal a fattoriale e mi salvo il valore di ritorno in x1.

Ad ogni salto io mi salvo x1 e x10.

Una volta arrivato alla fine → viene ritornato 1 dentro ad x10 e poi ritorna ad x1 → muovo ad x6 il valore di x10 che è 1 adesso → recupero il valore del penultimo salto su x10 e recupero l'x1

Faccio la moltiplicazione tra il return vecchio e il return adesso.



Leaf Procedure Example

main: addi x10,x0,3 addi sp,sp,-16 sd x1,8(sp) sd x10,0(sp) addi x5,x10,-1 bge x5,x0,L1 addi x10,x0,1 addi sp,sp,16 jalr x0,0(x1)	fact: addi sp,sp,-8 sw x1,4(sp) sw x10,0(sp) addi x5,x10,-1 bge x5,x0,L1 addi x10,x0,1 addi sp,sp,8 jalr x0,x1,0	RISC-V code: L1: addi x10,x10,-1 jal x1,fact addi x6,x10,0 lw x10,0(sp) addi x1,4(sp) addi sp,sp,8 addi x7,x0,1 loop: blt x6,x7,exitloop add x10,x10,x10 addi x6,x6,-1 exitloop: jalr x0,x1,0
<pre> lli fact (lli n) { if (n < 1) return 1; else return n*fact(n - 1); } </pre>		exit:

link emulatore risc-v

Il main inizializza x10 con un valore arbitrario e inizializzo anche lo stack pointer; normalmente lo fa una routine automatica.

Io ci metto dentro 1024 → bisogna essere sicuro che questo numero sia abbastanza per contenere le varie call ricorsive. → 1024 è pochino.

Anche il codice è un po' diverso → ho un 8 invece di 16 perchè l'emulatore non supporta il double → dobbiamo usare store word e load word → sw e lw invece dei soliti sd e ld

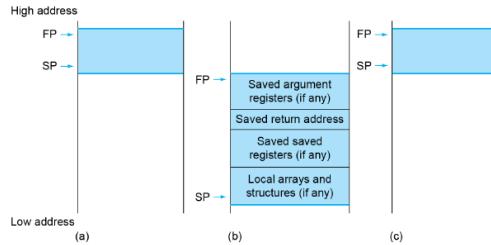
Quindi cambiamo anche gli offset perchè le word sono di 4 invece di 8 byte

Non c'è la mul perchè non è un'istruzione base; perchè ha bisogno di un'architettura fatta apposta → l'ultimo pezzo di codice esegue tante somme... che equivale alla moltiplicazione.

Across procedure call

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x11(ra)	
Stack above the stack pointer	Stack below the stack pointer

Local Data on the Stack

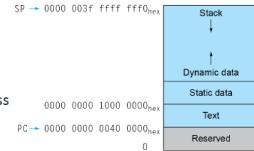


- Local data allocated by *callee*
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Memory Layout

- Text: program code

- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
- x3 (global pointer) initialized to address allowing to offsets into this segment



- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage

Nello stack i dati sono salvati di solito così: argomenti ; valori di ritorno ;

Procedure frame = identificato dal frame pointer(FP) e stack pointer(SP) → è il record di attivazione

Tutte le operazioni di load e store si possono fare per byte; halfword; word; doubleword....

Example 1: String Copy

- C code:
 - Null-terminated string

```
void strcpy (char x[], char y[])
{
    size_t i;
    i = 0;
    while ((x[i]==y[i]) != '\0')
        i += 1;
}
```

```
x10 = x; x11 = y; x19 = i;

• RISC-V code:
strcpy:
    addi sp,sp,-8           // adjust stack for 1 dw
    sd x19,0(sp)            // push x19
    add x19,x0,x0            // i=0
L1:   add x5,x19,x11          // x5 = addr of y[i]
    lbu x6,0(x5)             // x6 = y[i]
    add x7,x19,x10          // x7 = addr of x[i]
    sb x6,0(x7)              // x[i] = y[i]
    beq x6,x0,L2             // if y[i] == 0 then exit
    addi x19,x19, 1           // i = i + 1
    jal x0,L1                // next iteration of loop
L2:   ld x19,0(sp)            // restore saved x19
    addi sp,sp,8               // pop 1 dw from stack
    jalr x0,0(x1)              // and return
```

Could have used a temporary register (e.g., x28-x31) instead of x19 to avoid needing to save it

Parto allocando memoria; poi mi salvo la variabile i. poi la inizializzo a 0.

Qui metto in x5 l'indirizzo dell'elemento iesimo dell'array y → y[i] per avere il valore devo leggere dalla memoria → load byte unsigned e lo metto dentro ad x6

Usiamo unsigned perchè cerchiamo dei char → codifica senza segno...

Dentro ad x7 metto elemento iesimo di x

Storebyte (sb) → copio quello che c'era dentro ad Y[i] in X[i].

La branch confronta $Y[i]$ con 0; se è uguale a 0 → vado alla fine; se no continua per poi arrivare al jal → ciclo.

Algoritmo di sort

Example 2: Sort

- Illustrates use of assembly instructions for a C sort function
- Swap procedure (leaf)

```
void swap (long long int v[],
           long long int k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
- v in x10, k in x11, temp in x5

```
swap:
slli x6,x11,3      // reg x6 = k * 8
add x6,x10,x6     // reg x6 = v + (k * 8)
ld   x5,0(x6)      // reg x5 (temp) = v[k]
ld   x7,8(x6)      // reg x7 = v[k + 1]
sd   x7,0(x6)      // v[k] = reg x7
sd   x5,8(x6)      // v[k+1] = reg x5 (temp)
jalr x0,0(x1)      // return to calling routine
```

Serve una funzione di swap → swappa il valore di $v[k]$ con il valore dell'elemento successivo

Shiftiamo di 3 perchè lavoriamo con long long int → per moltiplicare *8 devo spostare di 3 → in x6 ho l'offset dell'elemento kappesimo di V.

Essendo $k+1$ l'elemento successivo a k → non ho bisogno di calcolare l'indirizzo di $k+1$.

ma mi basta usare offset = 8 perchè è l'elemento esattamente dopo

li, j = pseudoistruzioni → non sono davvero supportate dall'instruction set, ma nelle API si.

sono pseudoistruzioni che velocizzano come scrittura alcune istruzioni

load immediate = li = addi con un immediate + $x0\dots$ → addi $xn, x0, 2$

j = jump = jal $x0$, label....

Rendono un po' più facile la scrittura di un programma assembly.

Maxbubblegum: "quante menzogne in una sola riga". ahahah

Example 2: Sort

- Non-leaf (calls swap)

```
void sort (long long int v[], size_t n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1)
        {
            swap(v, j);
        }
    }
}
```

- v in x10, n in x11, i in x19, j in x20

```

> Skeleton of outer loop:
- for (i = 0; i <n; i += 1) {

    li x19,0      // i = 0
for1st:
    bge x19,x11,exit1 // goto exit1 if x19≥x11 (i≥n)

    addi x19,x19,1      // i += 1
    j for1st           // branch to test outer loop
exit1:

```

```

> Skeleton of inner loop:
- for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {

    addi x20,x19,-1 // j = i - 1
for2st:
    blt x20,x0,exit2 // go to exit2 if x20 < 0 (j < 0)
    slli x5,x20,3    // reg x5 = j * 8
    add x5,x10,x5    // reg x5 = v + (j * 8)
    ld x6,0(x5)      // reg x6 = v[j]
    ld x7,8(x5)      // reg x7 = v[j + 1]
    ble x6,x7,exit2 // go to exit2 if x6 ≤ x7
    mv x21,x10       // copy parameter x10 into x21
    mv x22,x11       // copy parameter x11 into x22
    mv x10,x21       // first swap parameter is v
    mv x11,x20       // second swap parameter is j
    jal x1,swap       // call swap
    addi x20,x20,-1   // j -= 1
    j for2st          // branch to test of inner loop
exit2:

```

> Preserve saved registers:

```

addi sp,sp,-40 // make room on stack for 5 regs
sd x1,32(sp) // save x1 on stack
sd x22,24(sp) // save x22 on stack
sd x21,16(sp) // save x21 on stack
sd x20,8(sp) // save x20 on stack
sd x19,0(sp) // save x19 on stack

```

> Restore saved registers:

```

exit1:
    ld x19,0(sp) // restore x19 from stack
    ld x20,8(sp) // restore x20 from stack
    ld x21,16(sp) // restore x21 from stack
    ld x22,24(sp) // restore x22 from stack
    ld x1,32(sp) // restore x1 from stack
    addi sp,sp,40 // restore stack pointer
    jalr x0,0(x1)

```

Register Usage – Calling convention	
> x5–x7,x28–x31: temporary registers	- Not preserved by the callee (visible across calls, must be saved by the caller if later used)
> x8–x9,x18–x27: saved registers	- Preserved across calls. If used, the callee saves and restores them
	RISC-V Registers all the registers used x1: return address x2: function argument x3: global pointer x4: local pointer x5–x7,x28–x31: temporary stores/loads on x5 and x6 can be dropped
calling convention	

Saving registers	
sort:	addi sp,sp,-40 # make room on stack for 5 registers sd x1,32(sp) # save return address on stack sd x22,24(sp) # save x22 on stack sd x21,16(sp) # save x21 on stack sd x20,8(sp) # save x20 on stack sd x19,0(sp) # save x19 on stack
Procedure body	
Move parameters	mv x21,x10 # copy parameter x10 into x21 mv x22,x11 # copy parameter x11 into x22 li x19,0 # i = 0 for1st:bge x19,x11,exit1 # go to exit1 if i >= n
Outer loop	addi x20,x19,-1 # j = i - 1 for2st:blt x20,x0,exit2 # go to exit2 if j < 0 slli x5,x20,3 # x5 = j * 8 add x5,x10,x5 # reg x5 = v + (j * 8) ld x6,0(x5) # reg x6 = v[j] ld x7,8(x5) # reg x7 = v[j + 1] ble x6,x7,exit2 # go to exit2 if x6 ≤ x7 mv x21,x10 # copy parameter x10 into x21 mv x22,x11 # copy parameter x11 into x22 mv x10,x21 # first swap parameter is v mv x11,x20 # second swap parameter is j jal x1,swap # call swap
Inner loop	addi x20,x20,-1 # j -= 1 j for2st # go to for2st
Pass parameters and call	addi x20,x20,-1 # j -= 1 j for2st # go to for2st
Inner loop	addi x20,x20,-1 # j -= 1 j for2st # go to for2st
Outer loop	exit2: addi x19,x19,1 # i += 1 j for1st # go to for1st
Restoring registers	
exit1:	ld x19,0(sp) # restore x19 from stack ld x20,8(sp) # restore x20 from stack ld x21,16(sp) # restore x21 from stack ld x22,24(sp) # restore x22 from stack ld x1,32(sp) # restore return address from stack addi sp,sp,40 # restore stack pointer
Procedure return	
	jalr x0,0(x1) # return to calling routine

Example 3: Arrays vs. Pointers

- Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- Pointers correspond directly to memory addresses
 - Can avoid indexing complexity
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
 - Induction variable elimination
 - Better to make program clearer and safer

• Clearing an Array

<pre> clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; } </pre>	<pre> clear2(int *array, int size) { int *p; for (p = &array[0]; p < &array[size]; p = p + 1) *p = 0; } li x5,0 // i = 0 loop1: slli x6,x5,3 // x6 = i * 8 add x7,x10,x6 // x7 = address // of array[i] sd x0,0(x7) // array[i] = 0 addi x5,x5,1 // i = i + 1 blt x5,x11,loop1 // if (i<size) // go to loop1 </pre>	<pre> mv x5,x10 // p = address // of array[0] slli x6,x11,3 // x6 = size * 8 add x7,x10,x6 // x7 = address // of array[size] loop2: sd x0,0(x5) // Memory[p] = 0 addi x5,x5,8 // p = p + 8 bitu x5,x7,loop2 // if (p<size) // go to loop2 </pre>
--	---	--

Quando lavoro con array devo sempre avere un base address + qualcosa dentro il quale ho calcolato l'offset
→ ragiono in termini di $a[i]$ → non posso usare le load senza immediate.

I puntatori invece possono essere più semplici perché se $p = \&A[0]$

→ dopo per passare al prossimo
elemento mi basta fare $p += 8$

Shift molto usato perchè la word,
halfowrd, double, ... sono tutti multipli di 2

Other RISC-V Instructions

› Base integer instructions (RV64I)

Those previously described, plus:

- auipc rd, immed // $rd = (imm << 12) + pc$
 - › follow by jalr (adds 12-bit immed) for long jump
- slt, sltu, slti, slti: set less than
- addw, subw, addiw: 32-bit add/sub
- sllw, srlw, slliw, srliw, sraiw: 32-bit shift

› 32-bit variant: RV32I

– registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

- **M**: integer multiply, divide, remainder
- **A**: atomic memory operations
- **F**: single-precision floating point
- **D**: double-precision floating point
- **C**: compressed instructions
 - 16-bit encoding for frequently used instructions

Le istruzioni che iniziano per **M** sono estensioni per moltiplicazione/divisione e resto → l'hardware è dedicato con un'unità fatta apposta che riesce a fare queste operazioni → se ho questo hardware → avrò delle istruzioni nell'instructionset che mi permette di avere il risultato della moltiplicazione.

Memory operation di tipo atomico → lettura update e scrittura di variabili.

Estensioni per i floating point → abbiamo visto che è molto complessa come rappresentazione → il processore necessita della **floating point unit**; se non ce l'ha, ci sono delle librerie software che permettono questo, rendendo il programma però più lento.



Reference card = sommario dove si trovano rapidamente le informazioni fondamentali per la isa risc-V, ordinate in ordine alfabetico
FMT = tipo dell'istruzione (logiche, aritmetiche, imemdiate, branch,)
Sulla destra ci sono le principali estensioni: le moltiplicazioni e le principali floating point e quelle atomiche, mentre in basso a destra la suddivisione dei bit nella word.

Seconda pagina:

Pseudosostituzioni: alias più corti per istruzioni molto usate.

j usato per un salto incondizionato.

bnez = branch not equal zero,...

Questa api ci da anche la possibilità di nominare in modo diverso i registri!!

zero per x0; return addres per x1; ...

i temporaneti come tn...

Quindi nel mondo reale a volte si usano questi nomi più astratti → **quando dobbiamo scrivere da 0 un programma, possiamo scegliere quale notazione usare. vale sia per i nomi sia per le pseudoistruzioni → anche per l'esame!**

Porzione riservata non accessibile; porzione di testo che contiene il programma; porzione con i dati statici e globali; heap(dati dinamici) e stack(funzioni).

indice

Instruction Set Architeture

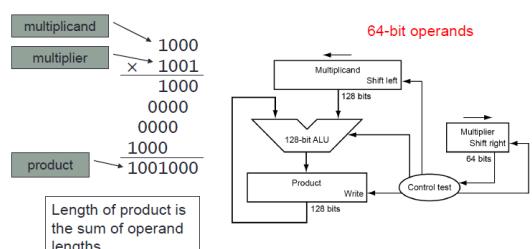
Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - RISC-V instructions

- Floating-point real numbers
 - Representation and operations
 - RISC-V instructions (extensions)

Multiplication

- Start with long-multiplication approach



Approccio base delle moltiplicazioni → **long-multiplication**.

Se la mia logica utilizza "n bit" per moltiplicando e moltiplicatore; il prodotto avrà n+n bit.

In logica binaria funziona simile a quella decimale: operazione di shift ad ogni numero nuovo e se ho 0 metto tutti 0; se ho 1 ricopio il numero...

Per implementarla:

- una ALU grande abbastanza quanto il doppio della precisione degli operandi → abbiamo operandi di 64 bit → **ALU 128 bit**

- Al primo step inizializzo a 0 il prodotto; poi a seconda di quando vale la cifra del moltiplicatore, questo shifta a destra di volta a volta.

Ogni step moltiplico il moltiplicando per una cifra del moltiplicatore poi passo alla cifra dopo → shift verso destra il moltiplicatore; poi la somma parziale la shift a sinistra di 1.

→ per lo shift a destra del moltiplicatore uso il **multiplier shift** → lo faccio perchè prendo l'ultimo elemento ogni volta e poi passo a quello subito dopo(a sinistra) → guardo solo la cifra meno significativa del moltiplicatore

→ per lo shift del moltiplicando uso il **contro test** → implementazione del mi abbasso di 1 riga e parto con già uno 0(su foglio)



in pratica faccio tante somme:

se la cifra meno significativa del moltiplicatore è 1 allora sommo il moltiplicando(pari pari) alla somma parziale precedente.

se la cifra meno significativa del moltiplicatore è 0 allora non faccio somme e mantengo la somma parziale precedente. → implemento lo scrivere 0000... (su foglio, sarebbe come sommare un numero a 0 → rimane quel numero) in ogni caso dopo faccio le seguenti operazioni:

→ **shift a sinistra IL MOLTIPLICANDO**(implemento riga sotto e parto con 1 zero in più), non la somma parziale

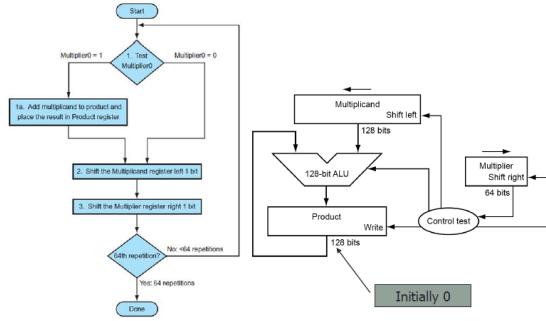
→ shift a destra il moltiplicatore → prossima cifra meno significativa → scorro il moltiplicatore

il prodotto inizializzato a 0, serve perchè funge da prima somma parziale → essendo 0 non ha peso sull'operazione di addizione successiva
nell'esempio il primo passaggio sarebbe $1000 * 1 + 0$

Allo start testo il moltiplicatore:

se ho 1 faccio lo step intermedio → sommo il moltiplicando al prodotto.
quindi inizialmente il registro prodotto ha 0

→ in pratica se la cifra del moltiplicatore è 1 faccio la somma; se no mi sposto alla prossima operazione

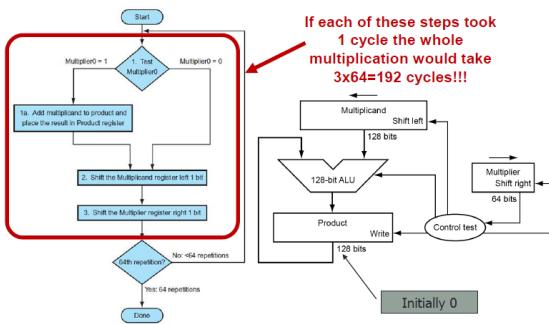


Signed multiplication works as long as sign is extended when shifting right

La moltiplicazione è un'operazione multiciclo

Tutte le volte che è possibile si sostituisce la moltiplicazione con uno shift, perchè? perchè fa tanti cicli la moltiplicazione; se il compilatore si accorge che una moltiplicazione è convertibile in uno shift, la convertono → **strength reduction** (molto conveniente!).

Can we optimize the multiplier?



Signed multiplication works as long as sign is extended when shifting right

C'è abbastanza parallelismo in questo algoritmo.
si possono unire i due shift insieme e anche fare l'addizione nello stesso ciclo
→ il registro prodotto è sempre 128 bit;

- **Example:** the bit examined to determine the next step is circled in color.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 \Rightarrow Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0000	0000 0100	0000 0010
2	1a: 1 \Rightarrow Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 \Rightarrow No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 \Rightarrow No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

- Multiplication is a multi-cycle operation
- Replace with left-shift operation whenever possible

Replacing arithmetic by shifts can occur when multiplying by constants.

Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2.

Almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

- This algorithm and hardware are easily refined to take one clock cycle per step.

- The speed up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1.
 - The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand.
 - The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders.

La mia logica così spende troppi cicli → 3 operazioni ogni bit → 3 cicli ogni bit → 64*3 cicli

Si può migliorare?

mentre la alu 64 bit (in realtà sono 129 bit perchè c'è il carry)

Il registro è diviso in 2:

- la parte a sinistra è 0 subito;
- mentre quello a destra è il moltiplicatore: ad ogni ciclo perdo 1 bit nel moltiplicatore(shift destra); così facendo man mano che perdo informazione nel moltiplicatore, ne guadagno nel moltiplicando.

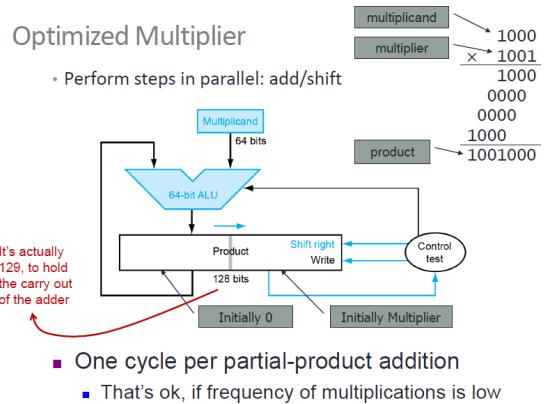
le singole operazioni di bit sono fattibili in 64 bit

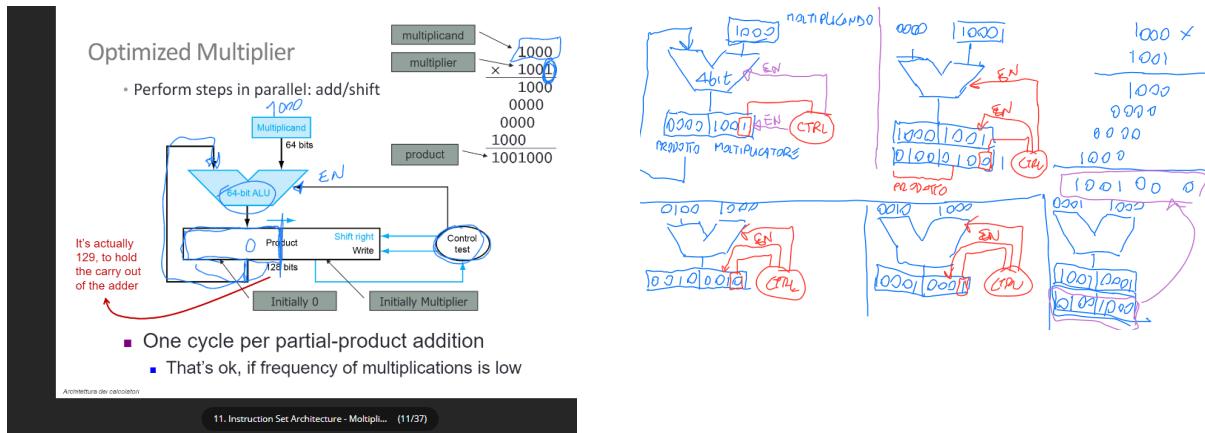
→ metà registro(sinistra) ha il risultato della somma locale + eventuale carry; quello a destra = moltiplicatore; ogni volta shifta a destra → mi si libera spazio per la parte di sinistra



facendo lo shift così non ho più bisogno di spazio aggiuntivo nella ALU per tenere in mente tutti i bit di shift.
la ALU somma solo la parte della somma parziale di sinistra(massimo 64bit)

il bit di enable è alto se l'ultima cifra del moltiplicatore è 1 → se è alto allora da l'ok all'adder e aggiunge la somma al prodotto
nel frattempo io continuo a fare shift destra.





Questo ci ha permesso di avere un algoritmo che fa fare tutto quello di primo in un ciclo per cifra → 64 cicli per operandi di 64 bit

questo funziona se ho tutti 0 → perchè con 0 faccio solo lo shift; nel momento in cui ho 1, io faccio 2 operazioni!

si può ancora fare meglio:
volendo si potrebbero collegare tante ALU a piramide per fare tutta l'operazione di moltiplicazione in 1 solo ciclo,
se la lunghezza di chiamata/tempo di propagazione del segnale di questa chiamata sta nel periodo di clock allora perfetto.

→ se no dovrei aumentare io il clock.
perchè? → link

Più adder in parallelo nello stesso istante;
posso ridurre il numero di cicli che mi servono.

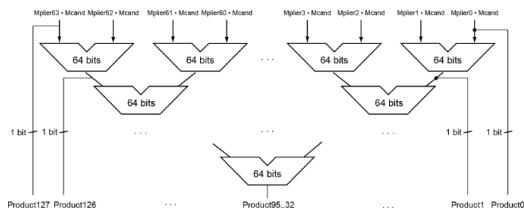
Io riuso normalmente il risultato come nuovo possibile sommatore; ma se io avessi 4 sommatori nell'esempio; allora farei tutto in un ciclo!
dovrebbero essere a cascata ovviamente.

Other optimization options?

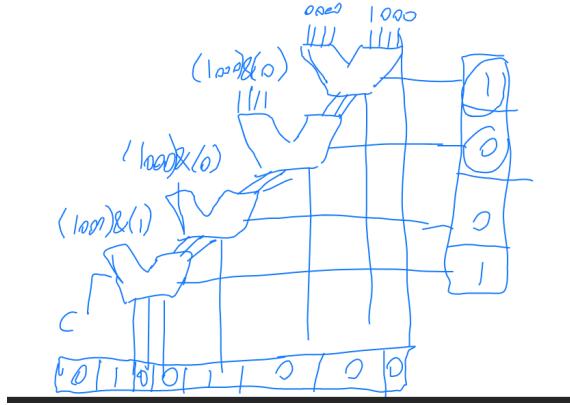
- Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 64 multiplier bits.
 - Faster multiplications are possible by essentially providing one 64-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 64 high.
 - An alternative way to organize these 64 additions is in a parallel tree:
 - Instead of waiting for 64 add times, we wait just the log₂ (64) or six 64-bit add times.

Faster Multiplier

- Uses multiple adders
 - Cost/performance tradeoff



- Can be pipelined
 - Several multiplication performed in parallel



cammino critico = percorso più lungo non interrotto dentro al circuito. può essere interrotto da un registro o arrivando alla fine del circuito

se il cammino critico sta dentro al tempo di clock, allora il circuito esegue in 1 ciclo di clock.

però non è sempre possibile farlo, ad esempio se abbiamo circuiti molto lunghi, tipo quello della moltiplicazione con tante ALU.



a questo punto si possono fare due cose:

si aumenta il tempo di clock → il circuito esegue in 1 ciclo di clock, ma è un falso speedup perchè il clock ha una frequenza minore(tempo di clock più lungo)

si inseriscono registri → se si inserisco tanti registri ad ogni ALU, il cammino critico diventa molto minore, quindi si esegue in 1 ciclo di clock, ma non tutto il circuito completo; alla fine si eseguirebbe il circuito completo nello stesso tempo di prima(un po' meno forse perchè perdo tempo nel salvare nei registri) Infatti bisogna implementare la **pipeline** → se è possibile implementare una pipeline(deve essere parallelizzabile), allora il tempo per trasformare un dato in output è sempre lo stesso, ma il throughput aumenta!

ad ogni ciclo di clock, prendo un dato in ingresso e pian piano scorrono il circuito. arriverò ad un momento in cui ad ogni ciclo esce un output

→ stesso tempo di prima, ma molto migliore

istruzioni Risc-V per la moltiplicazione:

essendo la ALU a 64 bit; io devo

spezzare l'operazione di mul

mul = da i 64 bit più bassi;

mulh = dai i 64 bit più alti; mulh si usa

spesso per controllare se c'è stato overflow.

RISC-V Multiplication

- Four multiply instructions:

- mul:** multiply
 - Gives the lower 64 bits of the product
- mulh:** multiply high
 - Gives the upper 64 bits of the product, assuming the operands are signed
- mulhu:** multiply high unsigned
 - Gives the upper 64 bits of the product, assuming the operands are unsigned
- mulhsu:** multiply high signed/unsigned
 - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned

- Use **mulh** result to check for 64-bit overflow

Altra operazione che si usa attraverso un'estensione: divisione.

metodo lungo:

Se il divisore sta nel dividendo allora metto 1 e sottraggo, poi continuo, finchè il divisore non ci sta:

→ metto 0 e tiro giù la prossima cifra, continuo finchè il divisore non stà di nuovo → metto 1 e sottraggo; vado avanti così finchè non finisco le cifre possibili.

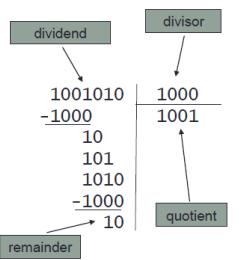
metodo più veloce:

Restoring division → io faccio sempre la sottrazione poi controllo se il risultato è positivo o negativo

Il quoziente parte a 0:

→ se il resto è ≥ 0 allora è un'operazione lecita e quindi metto nel quoziente 1 e shift il divisore
→ se invece il resto è < 0 ; ripristino e metto nel quoziente 0.

Division



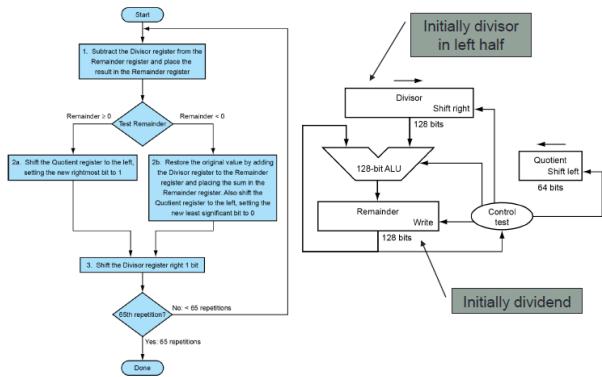
n -bit operands yield n -bit quotient and remainder

- Check for 0 divisor
- Long division approach
 - If divisor \leq dividend bits
 - 1 bit in quotient, subtract
 - Otherwise
 - 0 bit in quotient, bring down next dividend bit
- Restoring division
 - Do the subtract, and if remainder goes < 0 , add divisor back
- Signed division
 - Divide using absolute values
 - Adjust sign of quotient and remainder as required

Using a 4-bit version, divide 7_{10} by 2_{10} : 0111_{two} by 0010_{two}

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	00100000	00000111
	1: Rem = Rem - Div	0000	00100000	(0)100111
	2b: Rem < 0 $\Rightarrow +$ Div, SLL Q, Q0 = 0	0000	00100000	00000111
1	3: Shift Div right	0000	00010000	00000111
	1: Rem = Rem - Div	0000	00010000	(0)110111
	2b: Rem < 0 $\Rightarrow +$ Div, SLL Q, Q0 = 0	0000	00010000	00000111
2	3: Shift Div right	0000	00001000	00000111
	1: Rem = Rem - Div	0000	00001000	(0)111111
	2b: Rem < 0 $\Rightarrow +$ Div, SLL Q, Q0 = 0	0000	00001000	00000111
3	3: Shift Div right	0000	00000100	00000111
	1: Rem = Rem - Div	0000	00000100	(0)000011
	2b: Rem < 0 $\Rightarrow +$ Div, SLL Q, Q0 = 0	0000	00000100	00000111
4	3: Shift Div right	0000	00000010	00000011
	1: Rem = Rem - Div	0001	00000010	(0)000011
	2a: Rem $\geq 0 \Rightarrow$ SLL Q, Q0 = 1	0001	00000010	00000011
5	3: Shift Div right	0011	00000010	00000001
	2a: Rem $\geq 0 \Rightarrow$ SLL Q, Q0 = 1	0011	00000010	00000001

Division Hardware



i 64 bit di divisore all'inizio sono nella parte sinistra dell'array di 128 bit

il remainder è inizializzato con il valore del dividendo.

il control test decide quando shiftare il divisore e il quoziente e quando scrivere un nuovo valore dentro al remainder

- in pratica sottrae remainder - divisore e si mette il risultato dentro a remainder → subito il divisore sarà probabilmente molto più grande, perchè lo si mette nella parte sinistra dell'array divisore.
 - se remainder è negativo → si ripristina il remainder di prima sommando il divisore all'attuale remainder.
→ poi si shifta a sinistra il quoziente e setta a 0 la nuova cifra ottenuta(cifra meno significativa)
 - se remainder è positivo → si shifta a sinistra il quoziente e si setta il nuovo valore a 1 e basta così
- infine si shifta a destra il divisore facendolo calare di 1 cifra

il passo 2b per controllare; semplicemente guarda il bit più significativo del remainder, che non è altro che il bit del segno

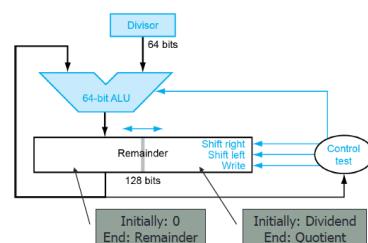
Signed division

rispetto al hardware precedente qui abbiamo
un ALU più piccola e anche il registro del
divisore.

il remainder è shiftato a sinistra, non più a
destra.

questa versione combina il quoziente con la
parte destra del remainder.

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
- Same hardware can be used for both

anche qui il registro del quoziente/reaminder in verità è di 129 bit perchè ha il carry out

questa architettura implementa il segno negli operatori → per farlo si accerta che il dividendo e il resto abbiano lo stesso segno per funzionare.

→ setta alla fine di tutto il resto allo stesso segno del dividendo.

mentre il quoziente segue le regole del segno

→ se dividendo e divisore sono negativi = quoziente positivo;

se solo uno di loro due è negativo = quoziente negativo

il resto è sempre uguale come funzionamento

Faster Division

- Can't use parallel hardware as in multiplier
 - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
 - Still require multiple steps

Come nella moltiplicazione, anche nella divisione riesco a compattare gli step mettendoli nello stesso blocco di 128 bit diviso in 2.

però nella divisione non si può ottimizzare con il parallelismo

→ perchè non posso sapere il risultato a priori dell'operazione!

→ quindi la divisione rimane sempre un'operazione multiciclo.

Per semplificare il progetto dell'hardware, lui di suo non gestisce la divisione per 0 o l'overflow; ma è chi usa l'operazione che dovrà gestirli.

Come si implementano le operazioni aritmetiche sugli operandi floating point??

RISC-V Division

- Four instructions:
 - `div, rem`: signed divide, remainder
 - `divu, remu`: unsigned divide, remainder
- Overflow and division-by-zero don't produce errors
 - Just return defined results
 - Faster for the common case of no error

div = quella che mi da il risulatto dell'operazione con segno e con resto (= quoziente)

divu = quella che mi da il resto senza segno (= unsigned)

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} (0.5 + -0.4375)$
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

1 → allineo allo stesso esponente → shift i numeri

2 → a sto punto posso sommare tra di loro le mantisse

a seguito di una somma c'è sempre il rischio che il numero non sia più normalizzato

3 → rinormalizzo il risultato!

4 → **rounding** → siccome il numero di bit per la mantissa è limitato; sommandone 2 rischio di perdere informazione → approssimo!(5-9 = 10; 1-4 = 0)

Un adder floating point è molto più complicato:

di solito per fare tutte queste operazioni è necessaria una pipeline → il floating point adder è una unità hardware a pipeline
→ mentre un'istruzione occupa uno spazio, le altre istruzioni ne occupano altri; senza aspettare...
→ nella singola istruzione non guadagno niente; ma nel singolo ciclo faccio più cose!

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

La moltiplicazione floating point è ancora più complessa:
1 → sommo gli esponenti → per quelli con bias, sottraggo il bias alla somma finale.(somma - 127)
2 → moltiplico le mantisse e come prima perderò la normalizzazione
3 → normalizzo il numero
4 → rounding
5 → controllo il segno → in base al segno degli operandi, cambia il segno del prodotto.

Floating-Point Multiplication

- Consider a 4-digit decimal example
 - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} (0.5 \times -0.4375)$
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × -ve ⇒ -ve
 - $-1.110_2 \times 2^{-3} = -0.21875$

Anche questa è un'operazione multiciclo.

In base 2: c'è da ricordare che gli esponenti hanno il bias a 127 → non basta fare la somma degli esponenti brutale

→ ma bisogna prima sottrarre 127 ad entrambi per riportarli normali.

Poi alla fine raggiungere 127 durante la normalizzazione.

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - FP ↔ integer conversion
- Operations usually takes several cycles
 - Can be pipelined



I registri per le operazioni floating point sono registri dedicati. perchè?

La questione è che: le operazioni in virgola mobile o le gestiamo con hardware dedicato (FPU):

https://it.wikipedia.org/wiki/Unità_di_calcolo_in_virgola_mobile

<https://ieeexplore.ieee.org/document/7208116>) o con software dedicato ed addirittura dei registri dedicati unicamente a queste operazioni!

Quindi i floating point hanno un register file dedicato a loro sempre di 64bit; ospito la parte significativa del risultato nei 32bit più bassi.

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6-7	t1-2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function argument / return value	Caller
x12-17	a2-7	Function argument	Caller
x18-27	s2-11	Saved register	Callee
x28-31	t3-6	Temporary	Caller
32 floating-point extension registers			
f0-7	ft0-7	Floating-point temporaries	Caller
f8-9	fs0-1	Floating-point saved registers	Callee
f10-11	fa0-1	Floating-point arguments/return values	Caller
f12-17	fa2-7	Floating-point arguments	Caller
f18-27	fs2-11	Floating-point saved registers	Callee
f28-31	ft8-11	Floating-point temporaries	Caller

Istruzioni floating point in risc-V (*molto simili alle altre solo che hanno f scritto prima...*)

FP Instructions in RISC-V

- Separate FP registers: **f0**, ..., **f31**
 - double-precision
 - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - f lw**, **f ld**
 - f sw**, **f sd**

FP Instructions in RISC-V

- Single-precision arithmetic
 - fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s
 - e.g., fadds.s f2, f4, f6
- Double-precision arithmetic
 - fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d
 - e.g., fadd.d f2, f4, f6
- Single- and double-precision comparison
 - feq.s, flt.s, fle.s
 - feq.d, flt.d, fle.d
 - Result is 0 or 1 in integer destination register
 - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
 - B.cond

nelle istruzioni floating point bisogna specificare la precisione

I confronti floating point → è l'unica istruzione che ha 2 registri sorgenti floating point e un registro destinazione intero → perchè torna 0 o 1

Da floating point ad intero

Un 1 in floating point è rappresentato molto diversamente da 1 intero → l'architettura farebbe fatica a confrontarli → per questo esiste il casting (è anche implicito a volte): il compilatore chiama l'istruzione della ISA per convertire il formato; è indispensabile a livello di architettura che ci sia qualcosa che permetta questa trasformazione.

Il registro di destinazione nell'esempio inizia per f: f0 → sono i registri speciale dedicati solo ai floating point.

I registri floating point non hanno l'equivalente di stack pont register pointer, questi registri servono solo per il calcolo.

FP Example: °F to °C

```
> C code:
float f2c (float fahr) {
    return ((5.0/9.0)*(fahr - 32.0));
}
- fahr in f10, result in f10, literals in global memory space

> Compiled RISC-V code:
f2c:
    flw    f0,const5(x3)      // f0 = 5.0f
    flw    f1,const9(x3)      // f1 = 9.0f
    fdiv.s f0, f0, f1         // f0 = 5.0f / 9.0f
    flw    f1,const32(x3)     // f1 = 32.0f
    fsub.s f10,f10,f1         // f10 = fahr - 32.0
    fmul.s f10,f0,f10        // f10 = (5.0f/9.0f) * (fahr-32.0f)
    jalr   x0,0(x1)           // return
```

FP Example: Array Multiplication

- $C = C + A \times B$
 - All 32×32 matrices, 64-bit double-precision elements
 - DGEMM (Double precision GEneral Matrix Multiply)
- C code:

```
void mm (double c[][],
          double a[][], double b[][])
{
    size_t i, j, k;
    for (i = 0; i < 32; i = i + 1)
        for (j = 0; j < 32; j = j + 1)
            for (k = 0; k < 32; k = k + 1)
                c[i][j] = c[i][j]
                           + a[i][k] * b[k][j];
}
```

 - Addresses of c, a, b in x10, x11, x12, and i, j, k in x5, x6, x7

- RISC-V code:

```

mm:...
    li    x28,32      // x28 = 32 (row size/loop end)
    li    x5,0        // i = 0; initialize 1st for loop
L1:   li    x6,0        // j = 0; initialize 2nd for loop
L2:   li    x7,0        // k = 0; initialize 3rd for loop
    slli  x30,x5,5    // x30 = i * 2**5 (size of row of c)
    add   x30,x30,x6  // x30 = i * size(row) + j
    slli  x30,x30,3    // x30 = byte offset of [i][j]
    add   x30,x10,x30 // x30 = byte address of c[i][j]
    fld   f0,0(x30)   // f0 = c[i][j]
L3:   slli  x29,x7,5    // x29 = k * 2**5 (size of row of b)
    add   x29,x29,x6  // x29 = k * size(row) + j
    slli  x29,x29,3    // x29 = byte offset of [k][j]
    add   x29,x12,x29 // x29 = byte address of b[k][j]
    fld   f1,0(x29)   // f1 = b[k][j]

```

```

slli   x29,x5,5      // x29 = i * 2**5 (size of row of a)
add    x29,x29,x7    // x29 = i * size(row) + k
slli   x29,x29,3      // x29 = byte offset of [i][k]
add    x29,x11,x29   // x29 = byte address of a[i][k]
fld    f2,0(x29)    // f2 = a[i][k]
fmul.d f1, f2, f1   // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1   // f0 = c[i][j] + a[i][k] * b[k][j]
addi   x7,x7,1        // k = k + 1
bltu  x7,x28,L3     // if (k < 32) go to L3
fsd   f0,0(x30)    // c[i][j] = f0
addi   x6,x6,1        // j = j + 1
bltu  x6,x28,L2     // if (j < 32) go to L2
addi   x5,x5,1        // i = i + 1
bltu  x5,x28,L1     // if (i < 32) go to L1

```

Pitfall: Right Shift and Division

- Left shift by i places multiplies an integer by 2^i
- Right shift divides by 2^i ?
 - Only for unsigned integers
- For signed integers
 - Arithmetic right shift: replicate the sign bit
 - e.g., $-5 / 4$
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - Rounds toward $-\infty$
 - c.f. $11111011_2 \gg 2 = 00111110_2 = +62$

Pitfall: Floating point associativity

$$\begin{aligned}
 c + (a + b) &= (c + a) + b ? \\
 c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\
 &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\
 &= 0.0 \\
 c + (a + b) &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\
 &= (0.0_{\text{ten}}) + 1.0 \\
 &= 1.0
 \end{aligned}$$

- FP numbers have limited precision and result in approximations of real results
 - $1.5_{\text{ten}} \times 10^{38} \ggg 1.0_{\text{ten}} \rightarrow 1.5_{\text{ten}} \times 10^{38} + 1.0 \approx 1.5_{\text{ten}} \times 10^{38}$.
 - That is why the sum of c , a , and b is 0.0 or 1.0, depending on the order of the floating-point additions
 - $c + (a + b) \neq (c + a) + b$
 - floating-point addition is not associative.**

risposte domande al prof:

Io in pratica uso 12 bit che sono i 13 bit di una parola → perchè l'ultima cifra è zero sempre → uso 12 bit e poi ci faccio shift a sinistra!!

Come minimo al PC devo aggiungere 2 se voglio fare un salto → quando uso istruzioni nel formato compatto → a destra avrò sempre 0 per forza.(abbiamo solo shift → *2/*4...) → il salto più grande è: "Il campo immediato della beq è a 12 bit, quindi (0111 1111 1111). Ricordando che il valore 2 nel campo immediato viene moltiplicato per due tramite la shift unit, l'offset reale diventa (1111 1111 1110) $2 = 0xFFE$. Rispetto all'indirizzo 0x2000 0000 il massimo salto in avanti è all'indirizzo 0x2000 0000 + 0x00000FFE = 0x2000 OFFE." ~soluzione slide

quindi io nelle branch shifto a sinistra l'immediato di 1 perchè tanto non posso avere un valore dispari, perchè in PC ho multipli di 2 → devo sempre avere 0 a destra e io guadagno 1 bit facendo questo shift(almeno quando cerco un valore positivo) giusto?
@maxbubblegum47

vale anche per il valori negativi?? c'è se io ho 12 bit a 1, cosa fa lo shift???
@maxbubblegum47

✓ in PC non dovrei avere multipli di 4??? non è 32 bit??? forse il minimo salto di 2 si riferisce se usiamo un architettura con istruzioni più corte? quindi la 16 bit ?????
@maxbubblegum47

il padding funziona quando vai a caricare da memoria!!! → quando prendi un numero minore di 64bit ed è un numero signed → allora viene riempito il registro copiando il numero, poi i restanti bit a sinistra vengono messi a 0 se il numero è positivo, oppure a 1 se il numero è negativo!

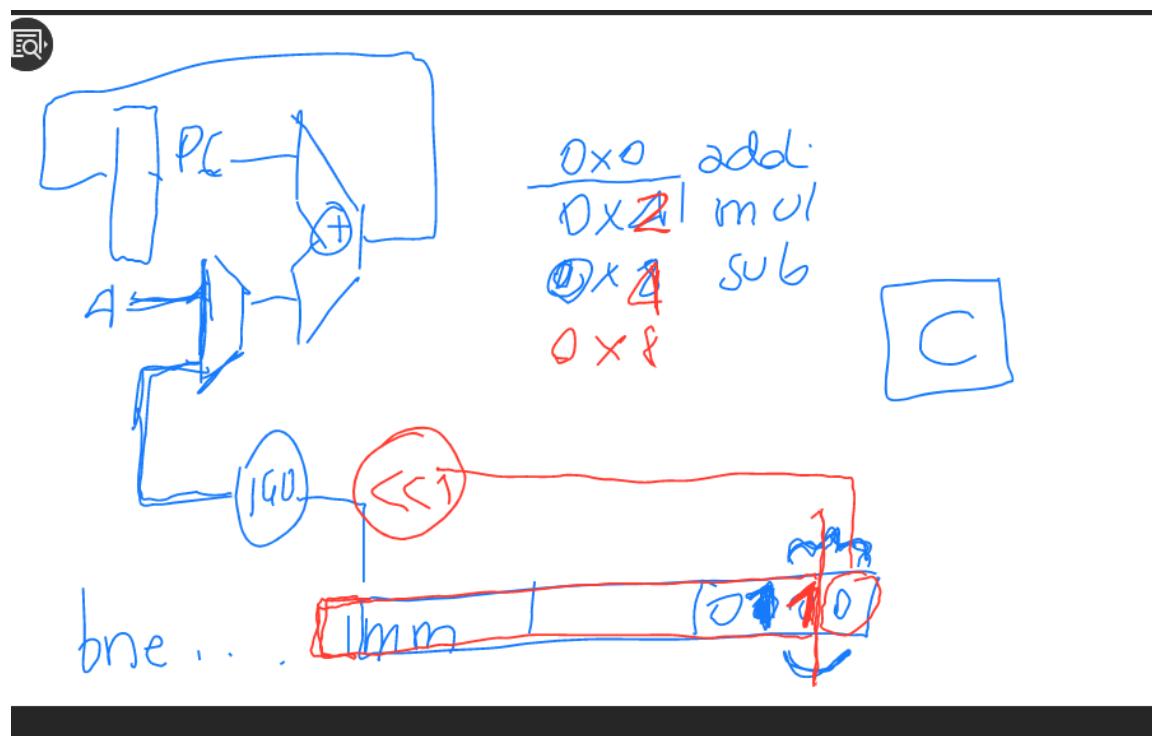
se invece è unsigned metto tutti a 0

nelle S → rs1 è il regisstro di destinazione (x9) e x22 quello in cui è contenuto il dato l'immediato sarebbe shiftato a sinistra di 2 posizioni (se siamo in PC 4byte)

però in realtà → il nostro programma usa 32bit → istruzioni = 0x0, 0x4, 0x8, ...
quindi nei 12 bit della branch io dovrei mettere dei multipli di 4 → il salto più piccolo che posso fare è di 4 → però noi sappiamo che c'è una rappresentazione del RISC-V che si chiama C

C ha un instruction set compressed → quindi ha istruzioni a 16bit → salto minimo = 2
→ quindi si è deciso di inserire l'immediato shiftato a destra di 1; e c'è un unità che prende questo immediato e lo shifta a sinistra di 1 per ripristinarlo e poi lo si userà come al solito

▼ immagine



Quali operazioni possono isolare un campo (un sottoinsieme di bit contigui) all'interno di una parola doppia? 1) AND 2) XOR 3) SHIFT a DX/SX 4) OR
isolare un campo di bit contiguous → se lo mettiamo in AND siamo sicuri che vengono selezionati i bit a 1; lo shift serve per creare le varie maschere quindi serve!

l'or no

esercizi:

▼ primo es

guardiamo l'opcode, se non basta a trovare il tipo dell'istruzione → guardiamo f3, se non basta → guardiamo f7

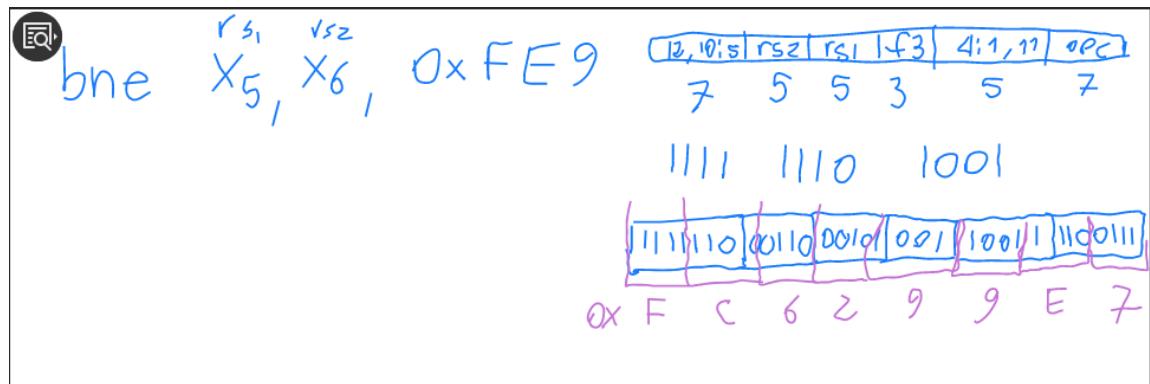
trovata l'istruzione, si guardano gli operandi

rd è 8 → x8

rs1 è 7 → x7

rs2 è 7 → x7

▼ secondo es



▼ terzo es



addi $x_5, x_0, 1$
addi $x_6, x_0, 5$

loop:

blt x_6, x_0, EXIT
addi $x_6, x_6, -1$
slli $x_5, x_5, 1$
jal x_0, loop

EXIT:

risposta = 64

▼ quarto es

CICLO:

blt x_6, x_0, EXIT
addi $x_6, x_6, -1$
addi $x_5, x_5, 3$
jal x_0, CICO

$$x_5 = 0$$

$$x_6 = 10$$

$$x_5 = ?$$

EXIT

risposta 33

▼ quinto es

cosa fa?

add t_0, x_0, x_0
loop; slti $t_1, t_0, 6$
beq t_1, x_0, end
slli $t_2, t_0, 2$
add t_3, s_0, t_2
lw $t_4, 0(t_3)$
Sub t_4, X_0, t_4
sw $t_4, 0(t_3)$
addi $t_0, t_0, 1$
jal x_0, loop
end:

slti R_d, R_s1, imm
if($R_s1 < \text{imm}$)
 $R_d = 1$

int arr[6] = {3, 1, 4, 1, 5, 9};
 \downarrow
 $s_0 = 0x\text{BFFFFF}00$

sovrascrive l'array con il negato di ogni suo elemento

slti = set less than immediate = rd, rs1, imm \rightarrow mette rd=1 se rs1 è < imm altrimenti mette 0

boh (probabilmente sarà la parte che ci chiederà di più di tutte 😂)

Lo stack pointer parte con un valore \rightarrow il sistema di runtime lo fa partire automaticamente ad un indirizzo molto alto \rightarrow perchè si sviluppa verso il basso.
Stessa cosa per il program counter (PC), ma ha un valore basso che si sviluppa in alto.

La memoria è segmentata \rightarrow segmento testo è il disassemblato \rightarrow codice tradotto in macchina e poi ritradotto in assembly??

Ci sono le istruzioni riconosciute dalla ISA:

- c'è un registro solo per i floating point(come avevamo già visto);
- segmenti di global data, una inizializzata e uno no \rightarrow in assembly dobbiamo fare a mano??

- per le variabili static → se noi non le inizializziamo loro vengono inizializzate a 0 automaticamente. C'è una parte di programma che rivela queste variabili;
 - le variabili di solito vengono trasformate in registri; queste no perché sono globali;
 - la vera e propria memoria → heap e stack;
 - i registri floating point non li utilizziamo di solito
 - istruzioni più importanti da ricordare
- ▼ immagine

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12]; addi rd, rd, symbol[11:0]	Load address
nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	subi rd, x0, rs	Two's complement

Le più usate in esame li → load immediate

Standard chiamate = tutte le chiamate hanno

Funzioni di ambiente → funzioni riconosciute dalla ISA che fanno operazioni complicate da assembly, fatte a posto per rendere il programma usabile da umano → link

Principalmente le 3 istruzioni di sequenza a slide 10!!!

Programma assembly fatto per eseguire su un sistema vero:
ci sono una serie di keyword:

- quelle che iniziano con ":" sono direttive che servono per specificare i segmenti della memoria
.globl → identifica l'inizio del segmento global che è quello dove mettiamo tutti i simboli globali → tra questi simboli c'è il nome della funzione main che deve essere raggiungibile dall'esterno

.data ci mettiamo il nome di una variabile globale:

msg1: .string "hello word" → .string è il tipo della variabile; msg1 è il nome della variabile

.text → segmento testo è il segmento della memoria dove è scritto il programma

Quando lavoro con delle stringhe → sto lavorando con variabili inizializzate → metto dentro a .data

- ▼ esempio fatto in classe

```

.globl main
.data
    str1: .string "i: "
    str2: .string "\n"
.text
main:
    li t0, 0 #i
    li t1, 10 #n
    li a7, 4
loop:
    la a0, str1
    ecall
    mv a0, t0
    li a7, 1
    ecall
    la a0, str2
    li a7, 4
    ecall
    addi t0, t0, 1
    blt t0, t1, loop
exit:
    li a0, 0
    li a7, 93
    ecall

```

▼ es fattoi n classe 2

```

.globl main
.data
    str1: .string "sum: "
    str2: .string "\n"
.text
main:
    li a7, 4
    la a0, str1
    ecall
    li a0, 10 #metto in a0 il valore di sum! -> così lo passo alla funzione sum
    jal ra, sum
    li a7, 1 #in a0 ho già il risultato della funzione; quindi cambio solo a7 per aver
    e la funzione di stampa intero
    ecall
    la a0, str2
    li a7, 4
    ecall
exit:
    li a0, 0
    li a7, 93
    ecall

sum:
    li t0, 0 #i
    li t1, 0 #sum
loop:
    beq t0, a0, sum_exit
    add t1, t1, t0
    addi t0, t0, 1

```

```

j loop
sum_exit:
    mv a0, t1 #metto il risultato in A0
    jr ra    #qui c'è PC+4

```

beq va bene usarla, ma è più robusto usare una bge!

- half-word = shortint = 2 byte
- word = int = 4byte
- doubleword = long int = 8byte

Lo stackpointer mi serve per implementare la pila → quindi implementare la vita e morte delle funzione, perchè normalmente io potrei benissimo fare una funzione senza preoccuparmi di niente.

Per specificare un vettore uso il tipo di dato → .word → intero
e scrivo in successione il numero degli elementi... 1, 2, 3, 4

(c'è da usare l'offset ovviamente)

nel RISC la cpu sa fare calcolo solo tramite il register file
in CISC ha una CPU lavora con register file, ma anche bda
sola???

[indice](#)

Processore - datapath e controllo

performance della CPU dipende dalla ISA e il compilatore
anche CPI, e cycle time, ma queste dipendono solo da com'è fatto l'hardware

una CPU risc-V in versione semplificata, ha la proprietà di preservare la posizione di campi fondamentali → opcode, funct3, funct7 e rs1 → sono sempre negli stessi posti anche in diverse istruzioni(o quasi)

come funziona l'esecuzione di istruzioni?

fetch: il PC è un registro che contiene le posizioni delle istruzioni da fare; dopo averla trovata si decodifica:

→ fase in cui determino l'indirizzo della prossima istruzione e la recupero

decodifica: si leggono i registri poi in base al tipo delle istruzioni cambia il

funzionamento della ALU:

execute: più tipi di istruzione

aritmetiche → la alu è usata per calcolare memoria → viene usata per calcolare l'indirizzo a cui andare

branch → usata per calcolare la differenza tra i due operandi e il risultato?????

memory: accedo alla datamemory per scrivere/caricare il dato se è necessario(istruzioni che usano la memoria)

writeback: scrive sul PC se serve farlo!

→ aggiorno il register file!



tutte queste fasi eseguono **SEMPRE**, solo che alcune istruzioni non fanno niente dentro a certe fasi!

io nella fetch, metto dentro un address e tiro fuori un'istruzione!

la decode determina che registri devo tirar fuori dal register file per passarli ad execute(fa quello che facciamo noi a mano)

gli operandi dell'ALU possono arrivare entrambi dal register file, oppure possono essere immediati che si prendono direttamente dall'istruzione

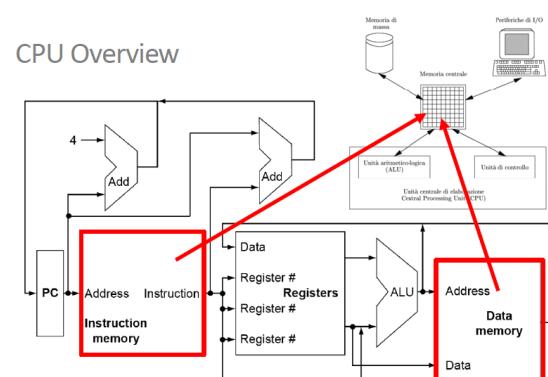
la exec passa indirizzo e dato(so lo se è store) per andare poi a fare load o store.

se fosse una load: dopo la load dalla memoria, porterei il risultato indietro alla decode;

mentre se è una store: porto indietro al register file???? → perchè devo salvare il dato giusto???

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch comparison
 - Access data memory for load/store
 - PC ← target address or PC + 4



l'adder iniziale, prende il valore del program counter e ci somma 4, poi ritorna nel PC

l'altro adder prendere l'address dall'istruzione e ci aggiunge il PC poi torna al PC → è il famoso jump

la memoria è 1 → DRAM, ma qua ho 2 memorie data e instruction a sua volta la DRAM è caricata da periferiche o memoria di massa. come funziona?

datapath = percorso dei dati

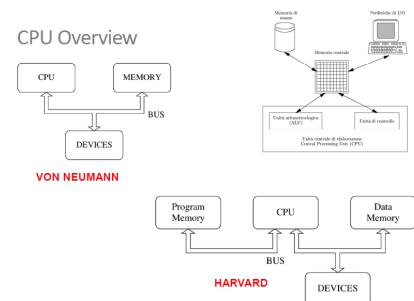
la nostra istruzione scorre attraverso tutti questi stadi/catena di montaggi per controllare il datapath serve una logica di controllo → multiplexer e segnali di controllo.

modello von neumann e modello harvard:

nel modello von neuman la CPU è collegata alla memoria e ai dispositivi I/O.

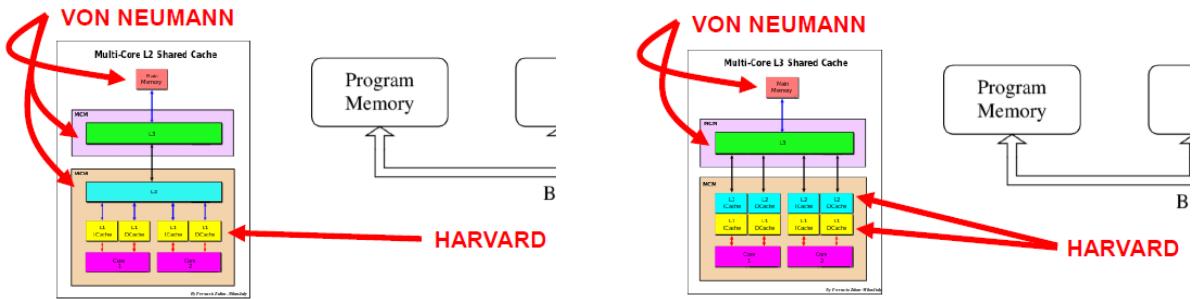
la harvard sfrutta 2 BUS, quindi ha 2 percorsi che usa per trovare in una memoria i programmi(istruzioni), nell'altra i dati
in pratica è più parallela → nel tempo in cui faccio una fetch nella memoria istruzioni, posso fare anche una load/store in quella dei dati.

in realtà una CPU ha un mix → vicino al processo, cache di primo livello: abbiamo un modello harvard perchè per performance vogliamo le 2 istruzioni in un ciclo → fetch e load/store poi si segue il modello von neumann per il resto.

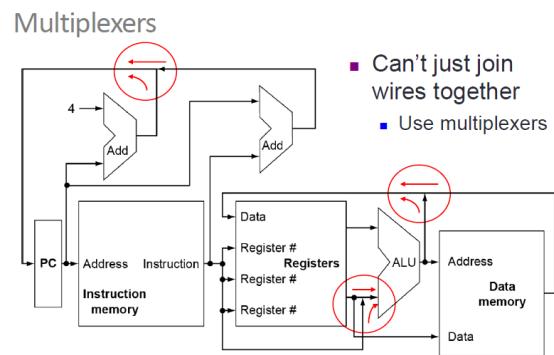


nel design della CPU tengo conto di data memory e instruction memory.(l'ISA ragiona così.)

però nella realtà cosa c'è tra la memoria e la CPU cambia da implementazione ad implementazione



non posso unire fili che arrivano da più direzioni → multiplexer!



mux alto = decide se il PC della prossima istruzione sarà incrementato di 4 o di più(ovvero se è un jump)
faccio una sottrazione tra ra1 e rs????? e guardo il bit???????

se il bit zero della ALU è = 1 → allora significa che ho preso il branch e quindi devo dire al MUX alto di usare il nuovo indirizzo calcolato come prossima istruzione, invece del solito PC+4 non ho ben capito dove calcola il nuovo indirizzo @maxbubblegum47 → forse lo dice dopo "si anche io qua ho segnato che praticamente nelle branch mi fa un controllo sugli operandi e mi ritorna il valore 0/1. Non ha spiegato forse molto bene come o comunque con che criteri, quindi nel caso se vuoi provare a chiedere lunedì poi dimmi qualcosa. In rete ho trovato qualcosa che ho riportato qua sotto cercando di tradurre: <https://electronics.stackexchange.com/questions/455007/cant-understand-what-multiplexers-do-in-cpu-datapath> @pablo remirez "

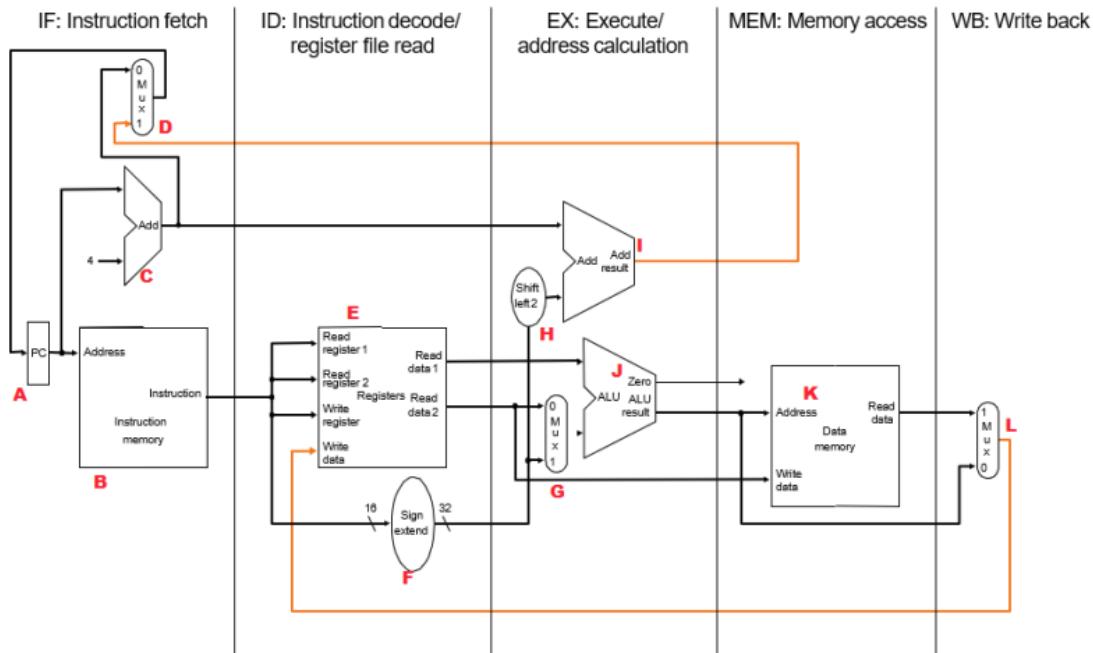


[link](#) a dove calcola il nuovo indirizzo

G multiplexer: seleziona il secondo argomento della alu e il possibili altri valori possono essere quelli dei registri o gli immediati dell'istruzione.

L multiplexer: seleziona il valore da scrivere nel registro destinazione, i possibili valori sono quelli dell'indirizzo del registro che guarda la ram o il risultato della ALU.

H shifter: moltiplica l'immediato dell'istruzione per 4. → qui c'è da stare attenti però perchè nella nostra ISA noi shiftiamo di 1, perchè in un formato possiamo avere istruzioni di 16bit ! porcoddue che casino
ahahah è infatti da internet potrebbe essere un casino prendere roba



la ALU sa fare tante operazioni e con il segnale che arriva dall'unità di controllo, la settiamo.

la data memory ha 2 segnali di controllo → **memory write** e **memory read** → in base a se è load o store il campo data è meno o più utilizzato.(ovviamente in store deve anche passare un dato)

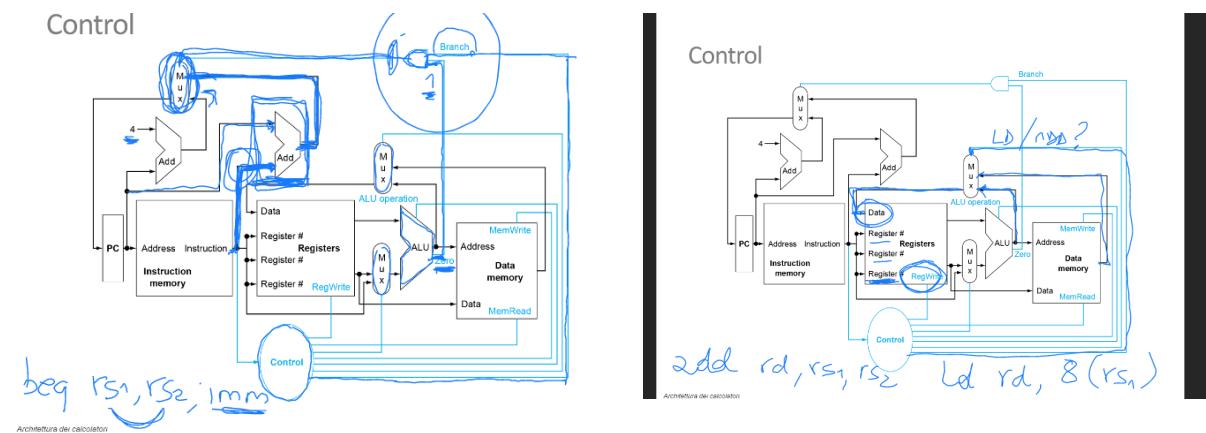
RegWrite dice se devo scrivere o no nei register file;
se ho un'istruzione R allora regwrite sarà settato ad 1; se regwrite è alto → scrivo dentro ad un registro il risultato(dentro a rd giusto???) questo lo fa subito dopo la ex o deve andare in wb non ricordo???, oppure se è basso, devo scriverci il risultato di una load(non ho capito perchè load????? @maxbubblegum47) "sinceramente non so esattamente quando andiamo a scrivere nel registro, ma immagino che sia nella fase di execute, quindi tipo prima noi fetchiamo, decodiamo, eseguiamo e poi dopo a ver messo nello rd quel che ci serve facciamo +4 al PC @pablo remirez "

→ temo invece che avvenga in WB la scrittura nel registro di destinazione, infatti più avanti scopriremo che questa cosa crea casini perchè il registro di destinazione viene sovrascritto dalle nuove istruzioni quindi dobbiamo fare forwarding o passare tra

registri....

la storia del se è basso faccio load continuo a non capirla, nel senso si è basso, ma non solo per le load → DOMANDA PROF?

control = dati gli insiemi di blocchi logici ci permette di configurare le operazioni da fare in base alla semantica.



i registri sono **edge-triggered** (solo i registri???)= lavorano sul fronte di salita → c'è anche un segnale di enable di solito(write), se è alto allora modifco il mio dato, altrimenti no(sempre in fornte di salita)

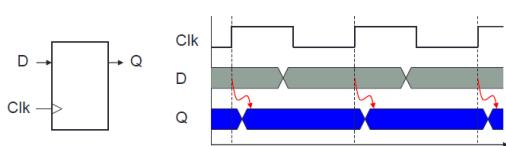
gli altri elementi, essendo combinatori → trasformano i dati durante il ciclo di clock???????? altri elementi intendi tipo le altre parti del datapath (data memory, instruction memory, adder, alu etc)? @pablo remirez

→ eh non ne ho idea, così c'è scritto ma non so cosa intenda → sicuramente tutti gli adder e mux ecc.. non sono sicuro per instruction memory e data memory

@maxbubblegum47

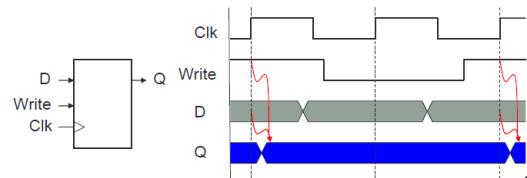
Sequential Elements

- Register: stores data in a circuit
 - Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1



Sequential Elements

- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



metodologia di clock.

un ciclo di clock più ampio → fa più cose all'interno del singolo ciclo di clock → però riduce la frequenza.

percorso critico = logica combinatoria che si trova tra 2 elementi di stato

se voglio una logica veloce, ho bisogno di percorsi critici più corti del tempo di clock →

se voglio una logica che fa più cose in un ciclo avrò un percorso critico maggiore

instruction fetch:

è comune a tutte le istruzioni! da cosa è formata?

- instruction memory pilotata dall'indirizzo dentro al PC → tira fuori 32bit di istruzioni ogni ciclo

- logica add che aggiunge 4 al PC → si trova nella sequenza di fetch.

- c'è un pezzo di logica combinatoria e un solo **elemento di stato**(il registro che memorizza il risultato) → dove ho l'istruzione di 32 bit ci attacco la logica di decodifica.

i read sono in pratica il valore del registro preso fuori e mandato in input alla ALU???

quindi sarebbe l'istruzione da 32bit??? @maxbubblegum47 "Si esatto, anche negli schemi si vede ben proprio come nelle istruzioni R Format dal blocco dei registri escano fuori come Read Data 1 e Read Data 2 proprio dei dati a 32 bit che finiscono dentro la ALU come suo input"

decode:

essa è diversa in base alla famiglia di istruzione!

dobbiamo configurare la nostra ALU in base al tipo.

c'è un percorso di writeback(non sempre è utilizzato) → scrive il risultato nel register file(rd).(è dopo la alu il writeback)

2 uscite che vanno alla alu!

nelle R format:

nella alu ci interessa il risultato non il bit

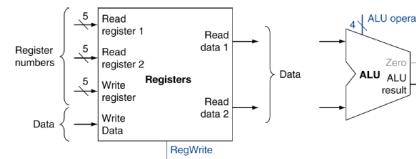
zero!

.non ci interessa la memory unit

R-Format Instructions

add x9,x20,x21

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

b. ALU

Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode

tettura dei calcolatori

per le load:

oltre alle cose prima devo utilizzare

anche la memoria e la **immediate generation unit**

imm gen = prende il mio dato immediate, lo riordina nella maniera opportuna e lo trasforma in un registro a 64bit → perchè la ALU lavora su operandi a 64bit?????

@maxbubblegum47

la imm gen prende i 32bit di istruzione, estrae i 12 bit di immediato e li manda in ingresso alla alu sotto(dopo averli trasformati???????) @maxbubblegum47)

"non ho idea, ho scritto la stessa domanda nei miei appunti" @pablo
remirez ← **DOMANDA AL PROF**

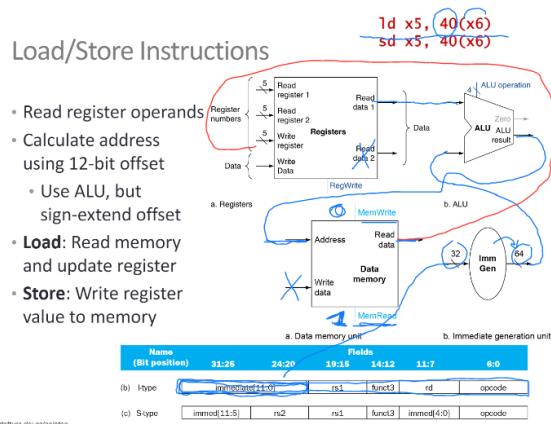
dipende dall'istruzione → lei estrarrai i 12 o 20 bit di immediato e poi trasforma in 64 biy

nella ALU ci va solo un read data, non tutti e due come nell'esempio delle R format di prima, perchè ho un immediate.

x6 è il base address → va nella ALU sopra.

il risultato della alu va in ingresso alla porta address della data memory
il segnale memRead sarà alto perchè è una load, il segnale memWrite sarà basso e in write data?? immagino niente
@maxbubblegum47 "si penso che se vado a fare una ld sicuramente leggo in memoria quindi memread è alto, mentre invece se faccio una store sicuramente devo abilitare memwrite"

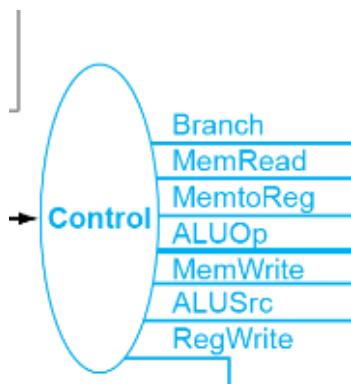
e ci sarà il writeback al register "di destinazione" ?????? "Si penso che se andiamo a fare una store e quindi scriviamo in memoria, avremo per forza il valore memwrite alto, per la questione del



writedata, non so cosa intendi perché io come valori ho segnato:

- Branch
- MemRead
- MemReg
- AluOP
- MemWrite
- AluSRC
- RegWrite

quindi di writedata non ho niente a dire il vero se non in input al blocco di Data Memory, ma quello non penso abbia un valore che vada alto o basso perché penso siano proprio dati che arrivano dal blocco dei registri" @pablo remirez → sì si intendevo proprio in entrata, quindi ha o no il valore in base al tipo di operazione



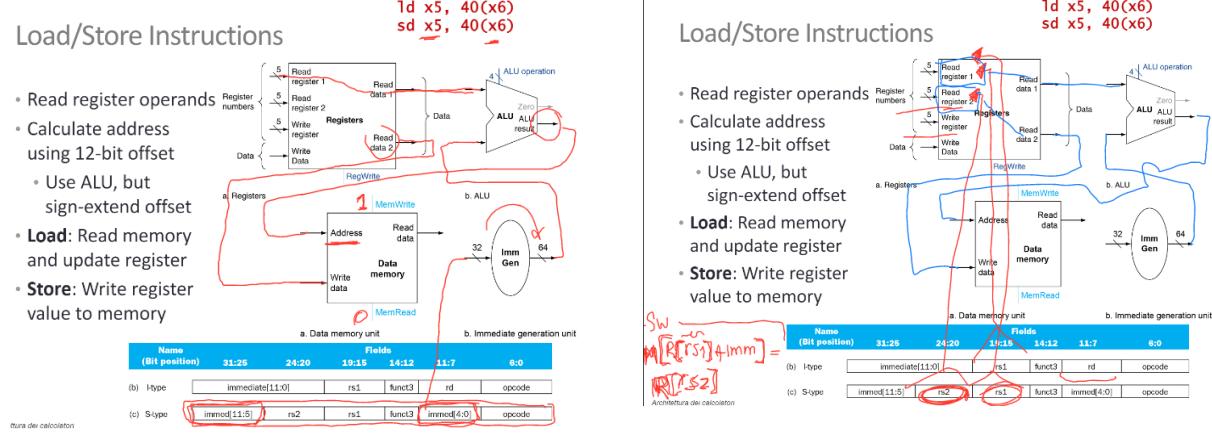
nella store:

abbiamo sempre il calcolo dell'indirizzo e anche qua la imm che calcola e da in ingresso ad ALU; l'output della ALU va sempre come adderss alla data memory
il memread a 0, il memwrite a 1 e la write data dentro alla data memory sarà pilotata dal valore dentro al registro

rs1 è il base adder register e rs2 è il valore che voglio scrivere

rs2 → register 2 → che andrà in scrittura

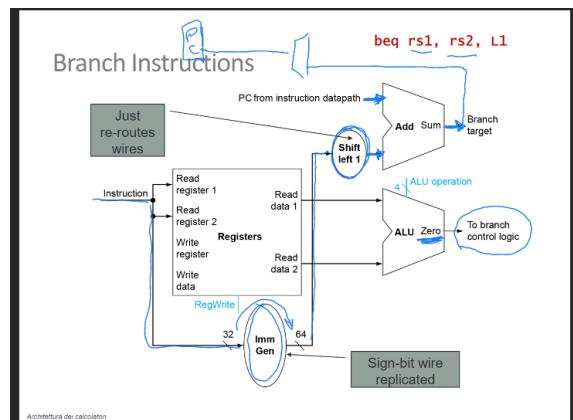
$r1 \rightarrow$ base address \rightarrow andrà a pilotare l'ingresso della ALU



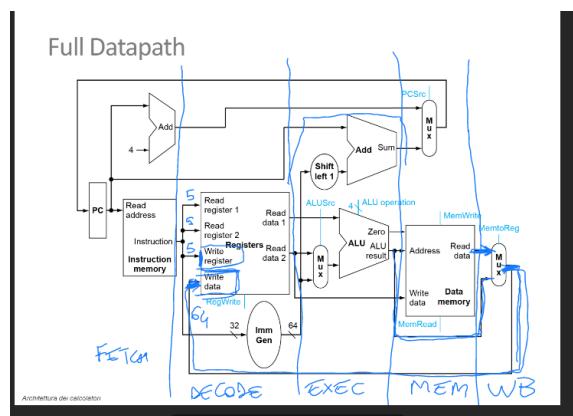
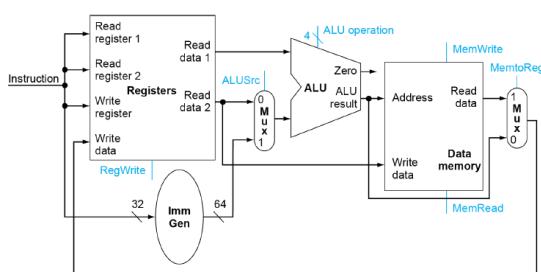
Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 1 place (halfword displacement)
 - Add to PC value

Name (Bit position)	31:25	24:20	Fields	19:15	14:12	11:7	6:0
(d) SB-type	Immed[12:10:5]	rs2		rs1	func3	Immed[4:1,11]	opcode



R-Type/Load/Store Datapath



la porta write è larga 64bit → la porta write data in ingresso al register; quelle dei registri sono 5 bit

segnali di controllo:

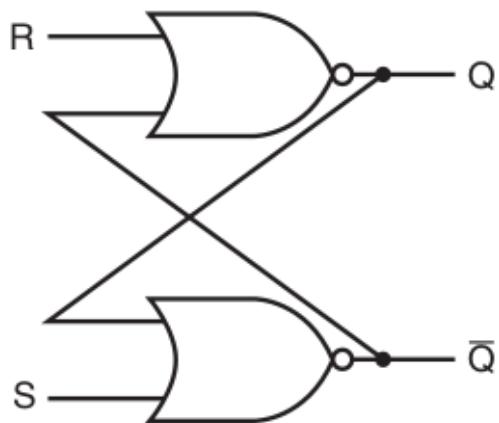
regWrite = se è alto(1), dentro al registro in write register viene scritto il valore in write data; se è basso(0) non succede niente

ALUSrc = segnale di controllo che controlla la sorgente della ALU → controlla se il segnale è un valore dentro ad un registro(0) o un immediato(1) → nelle parentesi il valore a cui è settato il segnale di controllo

PCSsrc = determina la sorgente per il program counter → determina se il

ALU operation = segnale a 4bit → fino a 16 configurazioni della alu???? serve per settare la ALU in base alla operazione da fare???? quindi sottrazione, addizione.... ???

@maxbubblegum47 dal manuale "*The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed next.*"



MemRead/Write = se sono bassi(0) non fanno la loro operazione, quindi lettura o scrittura da memoria; se sono alti(1) invece la fanno → non possono mai essere altri entrambi gisuto????? "se secondo me se stai operando su un pezzo di memoria non puoi averli entrambi enalbe, sia operativamente penso ci sia un toggle, sia perché comunque non puoi fare due cose diverse sullo stesso indirizzo di memoria. Cioè fisicamente potresti anche con una architettura multi pipe, però rischi di fare dei casini; potresti ritrovarti con dei dati alterati e quindi so che di solito quando stai lavorando su un dato indirizzo, finché non hai finito non lo cedi ad altri" @pablo remirez

MemtoReg = segnale per decidere se il valore che andrà in write data, è il risultato della ALU(0) o arriva dalla data memory(1)

da dove arrivano i segnali di controllo? → dalla codifica delle istruzioni

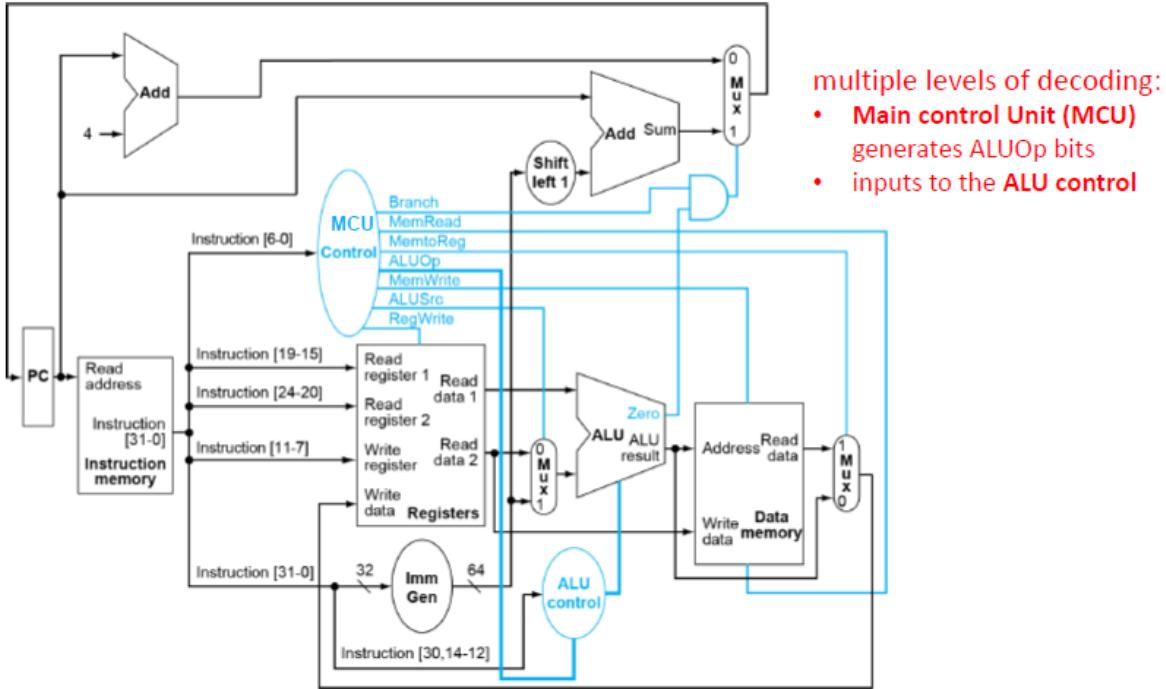


ci sono livelli multipli di codifica:

MCU = main control unit → esce memory read, memory register, →
MCU si occupa della generazione di tutti i segnali di controllo.

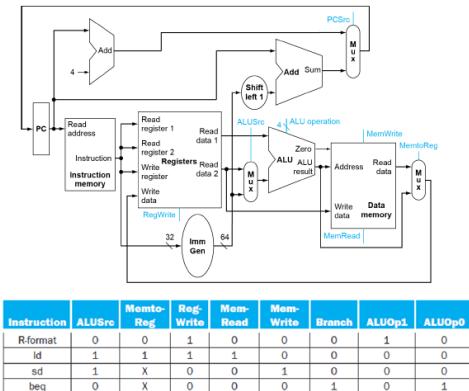
ALU control = pezzo di logica che decide come settare la ALU in base all'operazione che deve fare; la ALU control è pilotata dall'**ALUOp** che è generato sempre dal **MCU**

Datapath With Control



MCU:

The Main Control Unit



The Main Control Unit

- Control signals derived from instruction

Name (Bit position)	31:25	24:20	19:16	14:12	11:7	6:0
ALUOp	Funct7	rs2	rs1	funct3	rd	opcode
(a) R-type						
(b) I-type		immediate[11:0]		rs1	funct3	rd
(c) S-type	immed[11:5]	rs2		rs1	funct3	immed[4:0]
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp	Funct7 field	Funct3 field	Operation
0 0 0 X	X X X X X X	X X X X X X	add 0010
X 1 X X	X X X X X X	X X X X X X	subtract 0110
1 X 0 0	0 0 0 0 0 0	0 0 0 0 0 0	add 0010
1 X 0 1	0 0 0 0 0 0	0 0 0 0 0 0	subtract 0110
1 X 0 0	0 0 0 0 0 0	0 0 0 0 0 0	AND 0000
1 X 0 0	0 0 0 0 0 0	1 1 1 1 0 0	OR 0001

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and funct fields. Only the entries for the R-format ALUOp field are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries IX and XI, rather than 10 and 01. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 10, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.

qui si vede che in ALU control alla fine ci vanno dentro solo 4 bit dai funct field,
perché sono gli unici che cambiano
→ in input ha questi 4 bit e i 2 bit di ALUOp

ALU Control

Our simple RISC-V implementation covers *load doubleword* (*ld*), *store doubleword* (*sd*), *branch if equal* (*beq*), and the arithmetic-logical instructions *add*, *sub*, *and*, and *or*.

- ALU used for
 - Load/Store: F = add
 - Branch: F = subtract (and check if result is zero)
 - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

- ALU control input generated by a small control unit
 - has as inputs *funct7* and *funct3* fields, plus 2-bit control field (*ALUOp*)
 - ALUOp generated from opcode (main control unit)
 - add (00) for *loads* and *stores*, subtract and test if zero (01) for *beq*, determined by *funct7* and *funct3* for *R-type* (10).

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Required ALU action	ALU control input
0000011	00	load doubleword	XXXXXX	XXX	add	0010
0100011	00	store doubleword	XXXXXX	XXX	add	0010
1100111	01	branch if equal	XXXXXX	XXX	subtract	0110
0110011	10	add	0000000	000	add	0010
0110011	10	sub	0100000	000	subtract	0110
0110011	10	and	0000000	111	AND	0000
0110011	10	or	0000000	110	OR	0001

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See Appendix A.

efferta dei calcolatori

instruction memory pilotata dall'indirizzo in ingresso dal PC ed esce una istruzione a 32bit.

la decodifica identifica i campi registro 1, ecc...

in base all'istruzione che andiamo ad eseguire ogni campo avrà il suo significato

la imm generation unit è in grado di estrarre i bit di immediato, riordinarli e darli in ingresso al MUX che andrà alla ALU; oppure verrà shiftato a sinistra di 1 per essere sommato al PC e quindi generare un jump.

la ALU ha in uscita un risultato che può essere utilizzato per il writeback su register file; oppure un indirizzo per le load

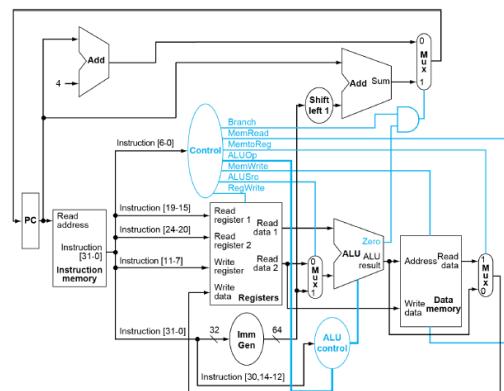
MCU → genera i segnali di controllo e i bit per ALUcontrol → gli altri bit li prende dai due campi funct!!

esempi:

R-type instruction

il numero 20 e 21 saranno mandati al register file ed il contenuto dei rispettivi registri sarà inviato alla ALU.

Datapath With Control



il segnale di controllo al MUX sarà 0 → il ALUSrc → perchè non utilizziamo un immediato

il blocco memoria è oscurato, ma non vuol dire che non lavora, è che in base agli ingressi fa qualcosa o no.

infatti in questo caso il mux dopo è a 0 → non calcola nessun indirizzo, ma è da saltare la memory.

si fa la writeback saltando la imm gen perchè non ho un salto ma solo il PC+4

load instruction

write register codificato a 5

read register 1 codificato a 6

e immediate a 40

il register 2 è oscurato perchè non c'è → in load non ho un secondo registro sorgente

alla ALU va in ingresso il valore del registro x6 e sotto ci va 40!

li somma e li passa avanti

essendo questa una load

→ memWrite a 0 e memLoad a 1

→ non ho write data e mux messo a 1

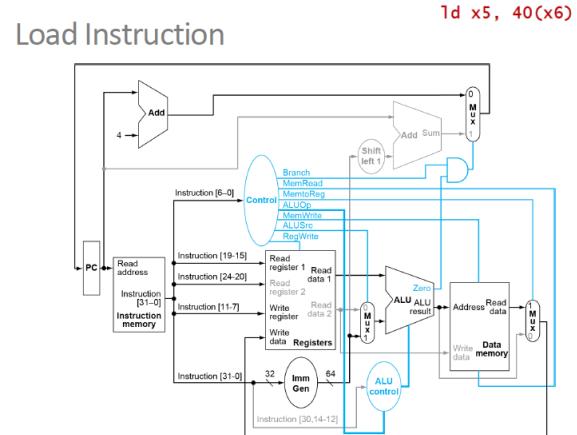
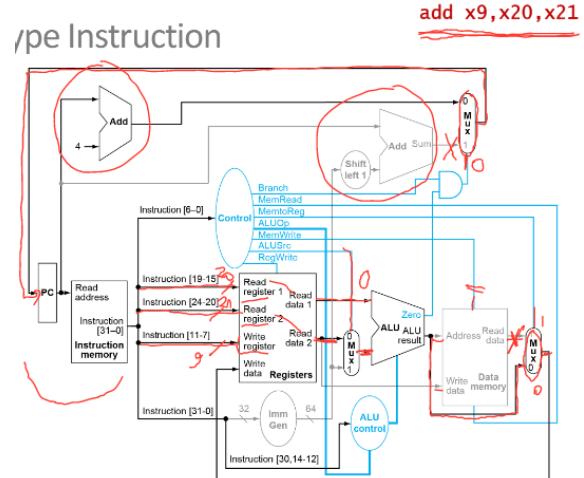
ho writeback: mi mette quello che leggo in write data dei registers

anche qua PC+4

branch

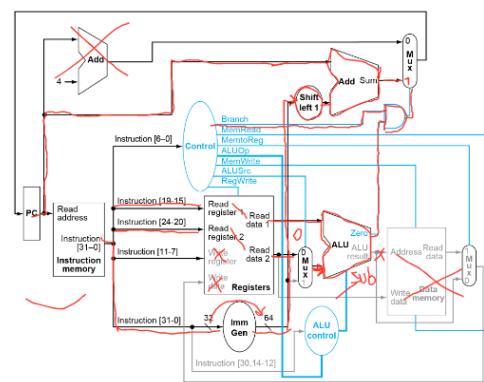
la porta di scrittura e di data non viene utilizzata

ALU control configurerà in modalità sub
→ alu sottrae i 2 numeri e setta l; uscita in base al risultato



se il branch funziona, allora ho l'uscita zero settata a 1 → mi setta il MUX in alto a 1 → il nuovo PC non sarà PC+4, ma sarà PC+immediato shiftato a sinistra di 1

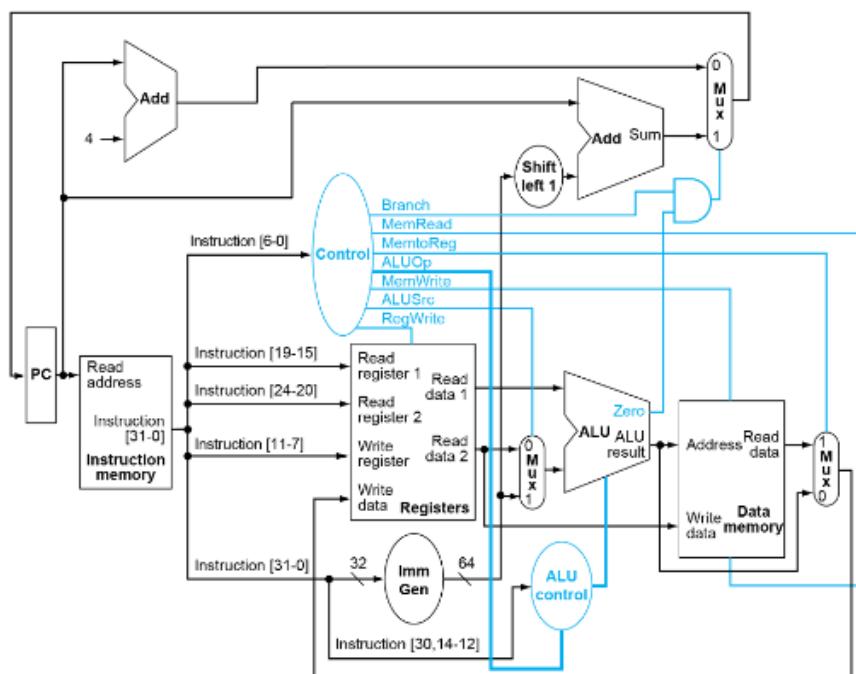
BEQ Instruction beq rs1, rs2, L1



esercizio:

Datapath With Control

beq 20, 50, L1



Architettura dei calcolatori

usiamo alu configurata come una sub → bit controllo con 0110
non uso la data memory

uso entrambi gli adder perchè? → dipende dall'esito della branch → quindi in base al risultato mi serve un addero o l'altro

se la branch fallisce, allora uso il PC+4; se no uso il PC+immediate

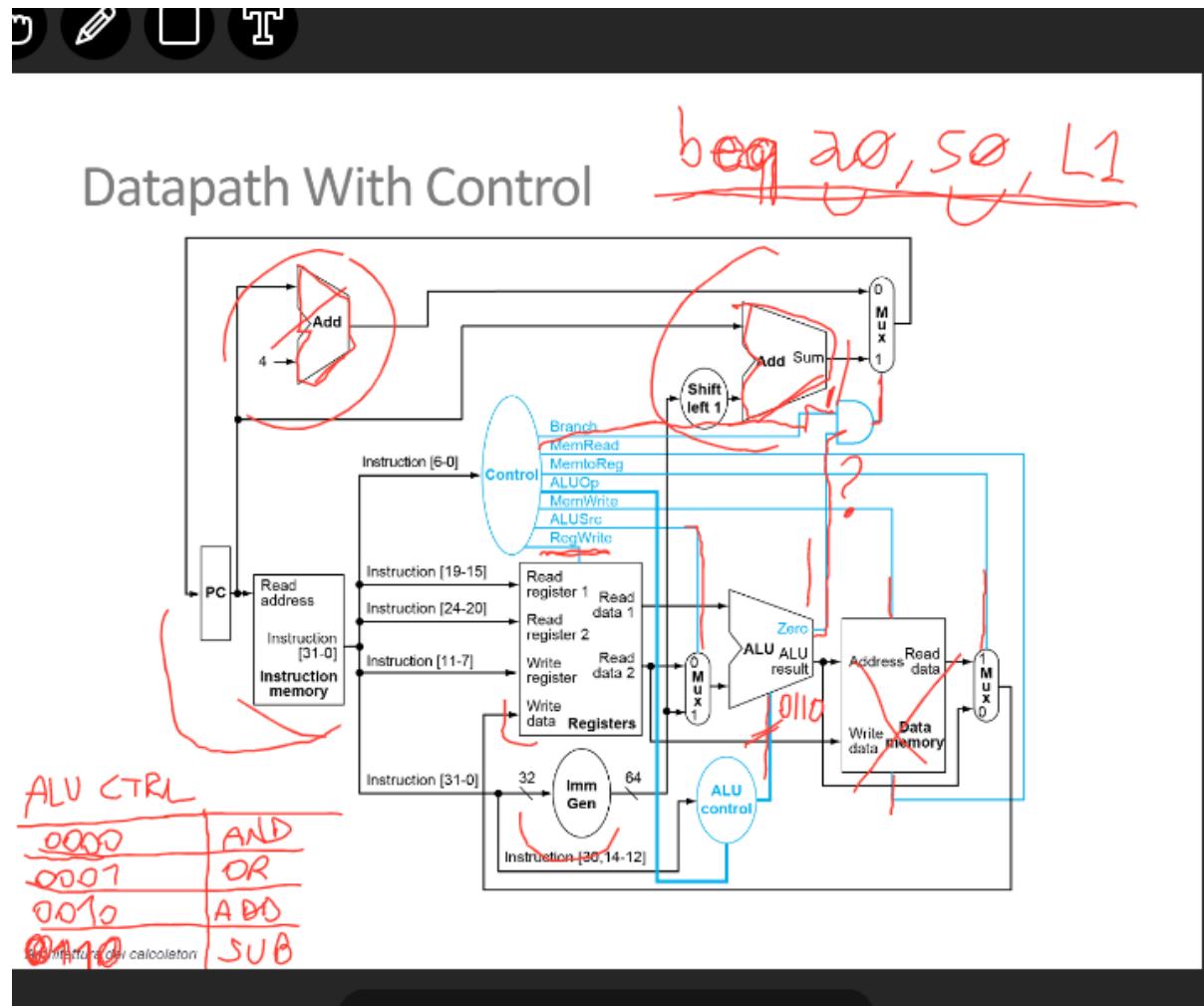
rewrite = 0

ALUsrc = 1 → perchè devo prendere l'immediato?????

memwrite = 0

memread = 0

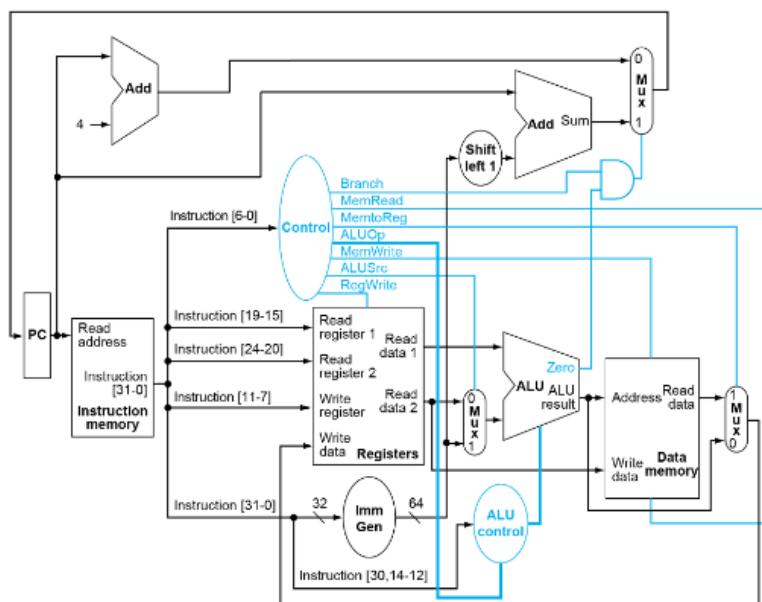
memtoreg = 0 → in verità è don't care → dal momento che metto rewrite a 0 → gli altri non funzionano più



esercizio

Datapath With Control

Sd x5, 8(x7)



Architettura dei calcolatori

gli adder solo quello a sinistra funziona → perchè il MUX di destra in alto ha valore 0, perchè non simao in una branch

registro 1 c'è x7; in register 2 no nc'è niente; in write registe c'è x5

→ read data 1 c'è il contenuto di x7 e in read data 2 niente

il valore del mux a destra è 1 perchè devo prendere l'immediato → sommare offset immediate calcola il valore 8 in sign extended

ALU control 0010 perchè faccio una add

il risultato viene passato alal data memory.

mem to reg = 1 → deve fare writeback

memwrite = 1

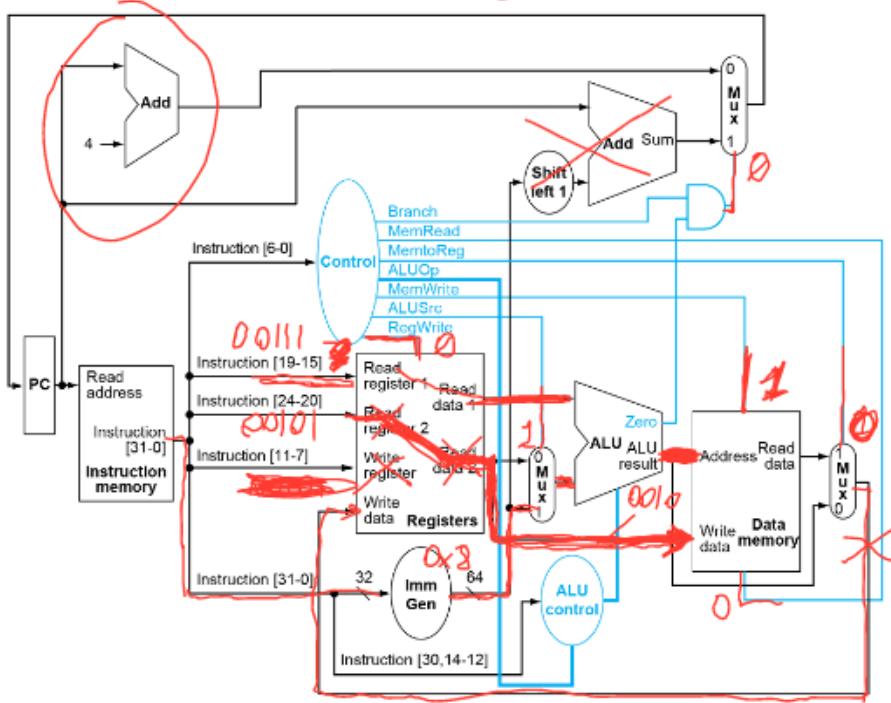
memread = 0

in write data di memory ci scrivo il valore in x5 → x5 non è iun write data, ma è in register 2?????? perchè non è un rd, ma un rs?????

regwrite = 0

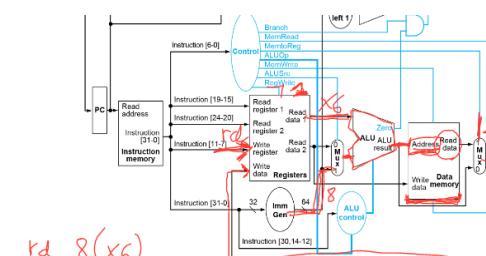
Datapath With Control

Sd ×5, 1d ×7

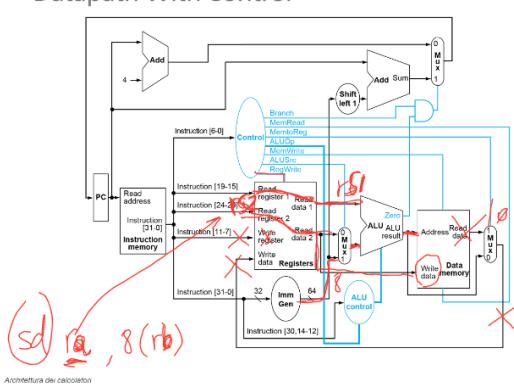


le istruzioni di ISA che lavora con la memoria quali sono?
store, load e basta.
la data memory prende un indirizzo in ingresso che è register1 + immediato.
se si tratta di una load → io leggo quello che c'è lì → alzo il segnale di write back = 1(il mux a destra) e scrivo su write data

nel caso di una store il registro sorgente ra è il registro che contiene il dato che voglio scrivere
→ mux è a 0 perchè non carico niente e nel write data mettiamo il valore nel registro 2(ra)



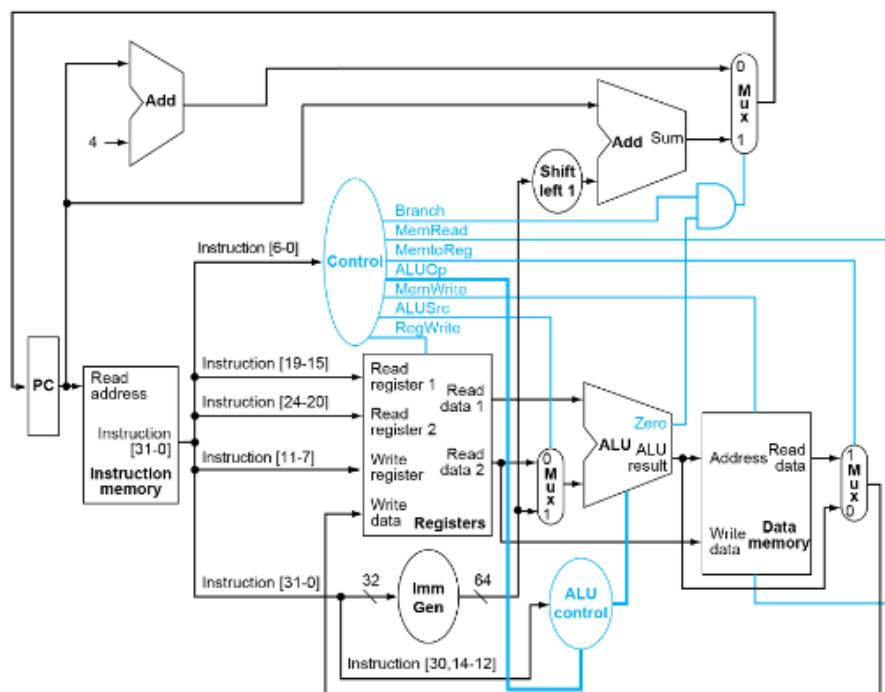
Datapath With Control



esercizio

Datapath With Control

ori X5, X6, 0xFF



Architettura dei calcolatori

la or è aritmetica → registri sicuro; l'immediato perchè lo usa...

x6 in register 1

register 2 nullo

write registe = x5 = la destinazione

regwrite = 1 → perchè devo scrivere il risultato → se regwrite è a 1, allora ho il write register non nullo

il mux a destra è a 1 perchè usiamo un immediato

l'immediato 0xFF va nella imm gen e poi al mux

la data memory non è usata → meme write e mem to reg = 0

anche il mux più a destra = 0 perchè bypassa la memory

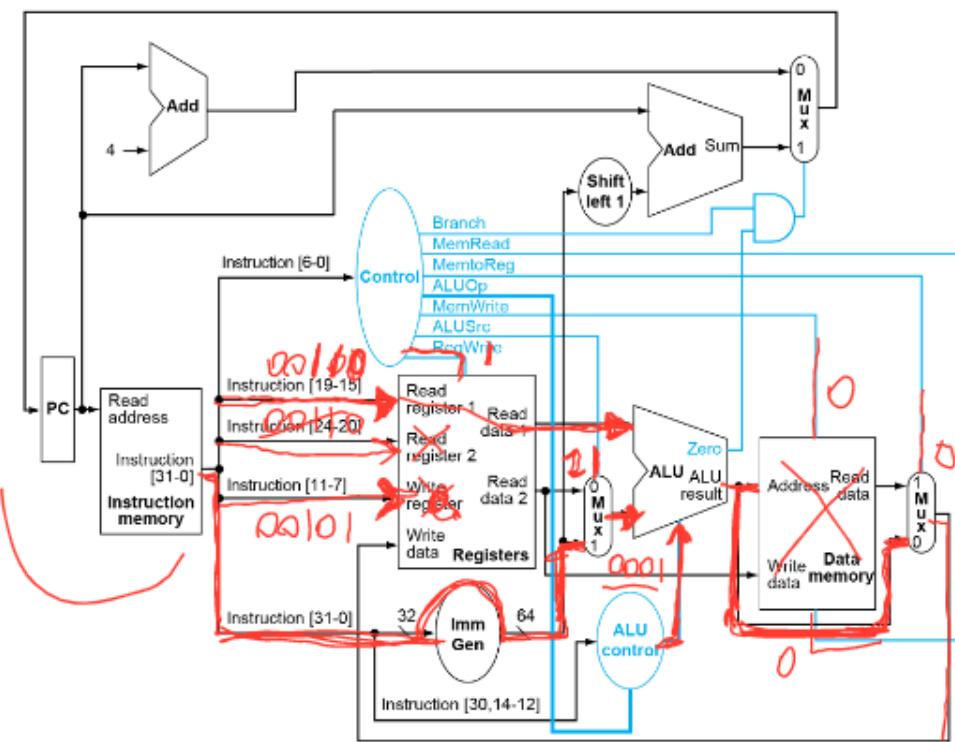
ALU cointrol = 0001

il dato torna indietro e va in write data

il mux in alto è a 0 perchè non ho un salto → calcola PC+4

Datapath With Control

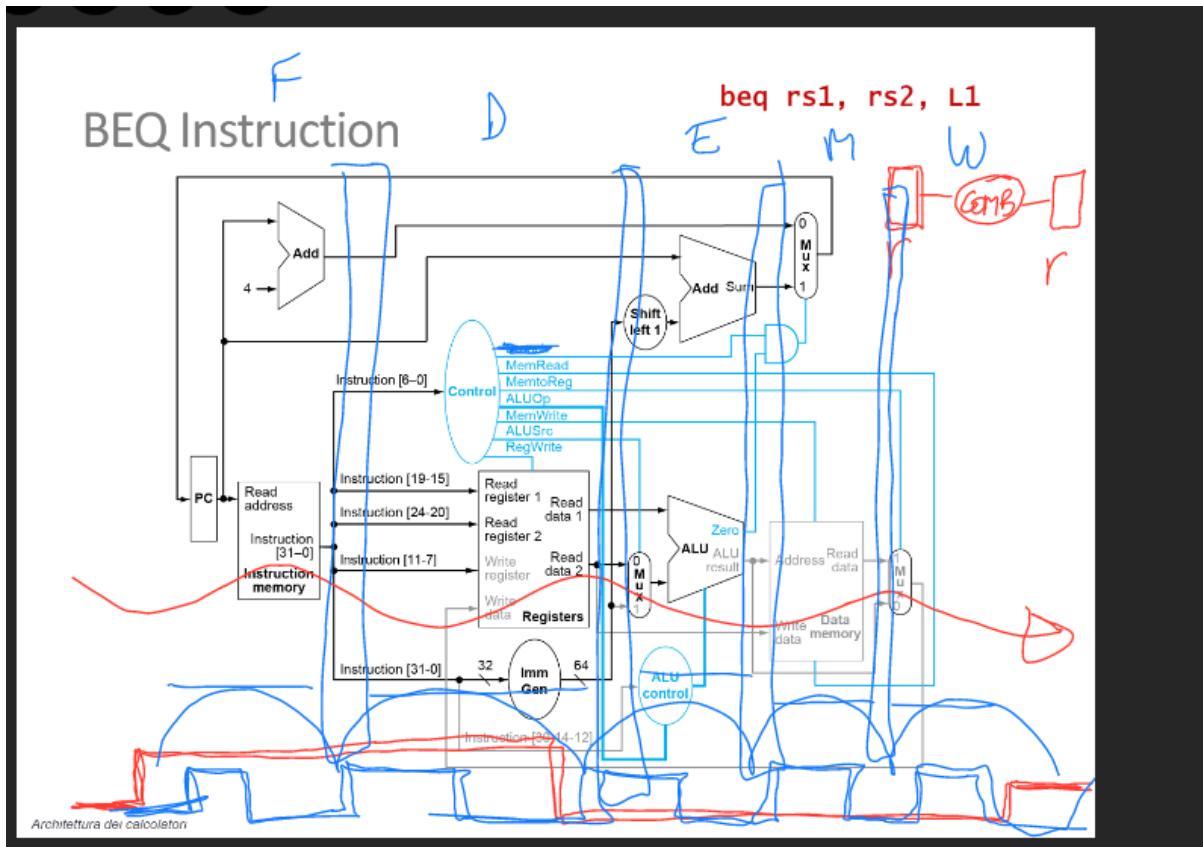
ori X₅, X₆, 0xFF



il nostro segnale di clock deve essere abbastanza largo da contenere tutta questa logica in 1 ciclo.

per consentire al segnale di clock di diventare più piccolo io posso mettere dei registri tra una fase e l'altra.

ora la mia istruzione eseguirà in più cicli, ma i segnale di clock può essere più stretto → pipeline



DOMANDE PER IL PROF

come mai il MUX in fondo viene messo a 0 per le store?? non dovrebbe essere indifferente visto che abbiamo regWrite a 0

indice

processore - pipelining

per motivi di performance quasi tutti i design nella pratica sfruttano il paradigma pipeline
 → principalmente perchè rendono la CPU più "veloce" (sappiamo che non è esattamente più veloce l'istruzione)

tutta la logica che si è vista prima è stato definita come un grosso percorso logico, la logica combinatoria è sempre stata definita come istantanea; in realtà non è così, perchè i circuiti fisici hanno un segnale elettrico che ha un tempo per propagarsi.

il ritardo che ci vuole perchè il segnale passi attraverso il percorso critico ci da una velocità → perchè il clock di CPU deve comprenderlo per eseguirlo in 1 ciclo → più il percorso critico è lungo, più sarà lungo il clock → più tempo ci vuole per 1 ciclo.

di solito si cerca di ridurre questo percorso critico e si può fare mettendo dei registri → possiamo spezzare un circuito logico complesso, mettendo dei circuiti.

→ così facendo il percorso critico non è più relativo all'intero datapath, ma al più lungo dei pezzetti → il ciclo di clock riesce ad essere più piccolo

però l'istruzione rimane la combinazione di queste fasi → la velocità della SINGOLA istruzione non è maggiore; ma rende la frequenza di clock più alta e soprattutto rende parallelizzabile la logica → posso avere più istruzioni che eseguono contemporaneamente dei pezzi(DIVERSI) della pipeline quindi avrò un transitorio(?) = cicli per RIEMPIRE la pipeline; poi ad ogni ciclo avrò un output!!

sovrappongo le esecuzioni delle istruzioni e nel complesso produco più cose
throughput = quantità di istruzioni "ritirate" in un tempo

Io speedup ideale che posso avere nella pipeline è pari al numero di stadi che ha pipeline risc-V

fetch dove prendiamo i 32bit di istruzione
la WB e la MEM non sono utilizzati da tutte le istruzioni!!! ciò però non significa che completano le istruzioni subito

→ ogni istruzione completa l'esecuzione in 5

step(quindi 5 cicli) SEMPRE, anche se non usa WB e MEM

però noi non abbiamo la durata dei vari stadi uguale per ogni istruzioni!!!

la load usa tutti gli stadi della pipeline → infatti è quella che dura di più

200ps perchè deve assorbire la pipeline????? @maxbubblegum47 "di questo discorso che poi riprende un pochino quando finisce per parlare di Amdahl e di Speedup ho capito che ci sono operazioni che richiedono più tempo di altre all'interno della pipe e questo comporta anche ad avere un miglioramento in fase di parallelismo che non è sempre esattamente proporzionato allo speedup che vorremmo ottenere; non so esattamente dove volesse arrivare, se fosse solamente un esempio il fatto che ci abbiamo parlato di pico seconodi e

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

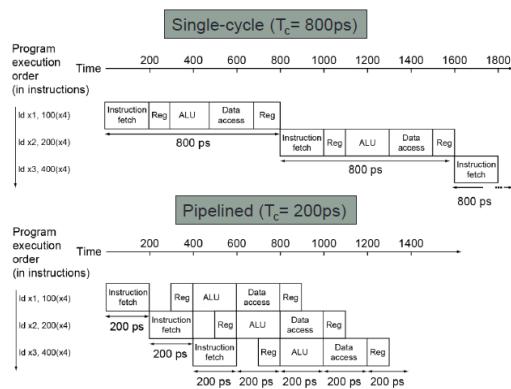
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

quant'altro, ma penso volesse solo darci un'idea delle tempistiche reali di una operazione su datapath e di come organizzare bene la pipe sia fondamental." @pablo remirez

l'unica cosa che mi è venuta in mente è che per assorbire intende che il ciclo di clock non può durare meno di 200ps perchè 200ps è il nostro cammino critico in questo esempio, quindi il clock deve "assorbire" il cammino critico per far funzionare la pipeline

ha senso? @maxbubblegum47

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
 - i.e., all take the same time
 - $\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
 - Latency (time for each instruction) does not decrease

ovviamente più gli stadi sono bilanciati(durano lo stesso tempo), più lo speedup si noterà; spesso invece gli stadi hanno durate diverse .

→ lo speedup dipende dallo stadio più lungo

noi dobbiamo dimensionare il nostro clock sullo stadio più lento!!!!

se abbiamo 3 stadi: uno da 100 uno 200 e i 300

→ io avrò cicli di clock da 300, perchè il mio clock deve assorbire il percorso critico

il design del risc-V è molto semplice da pipelinare

→ tutte le istruzioni hanno tutte 32bit e questo aiuta(una logica con istruzioni tutte a bit

diverse, complica il circuito logico
complessivo).

è fatto così per rendere il caso comune il più
veloce possibile

i formati di istruzione sono pochi e tutti molto
regolari → scrivere nella memoria è molto
semplice e veloce?????? @maxbubblegum47
non ho idea del perché "*mmmmmh non sono
sicuro, io avevo capito che era meglio sempre
solo leggere dalla memoria e che quindi avere
una buona Instruction Memory fosse comodo;
avevo capito che scrivere fosse meno stonks,
ma forse voleva solo flexare che il riscv è
velocissimo a fare tutto perché ha un caso
base forissimo quindis sti caazzi.*" @pablo
remirez

uhm però non capisco cosa significa scrivere?
noi stiamo facendo fetch, forse mi sono perso
dei pezzi

@maxbubblegum47

non so perchè può calcolare in terzo
stadio(?)... l'ultima riga dell'immagine
@maxbubblegum47

DOMANDA PROF? sisi questa cosa non
hocapito nemmeno io, non sto capendo
l'esempio che sta facendo, magari mi rivedo
direttamente la videolezione @pablo remirez

DOMANDA PROF: perchè risc-V è buono per il
pipelining?

in cisc decodifico prima perchè ho vari bit
il cischa istruzioni che calcolano direttametne
su memoria

in cisco so che in memoria hjo I; indirizzo

- RISC-V ISA designed for pipelining
- All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
 - Can decode and read registers in one step
- Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage

ha però molti problemi la pipeline: **hazards**

quando non abbiamo parallelismo? → immagino voglia dire quando non funziona
 3 tipi di hazard: structure, data e control

structure hazar

se una risorsa è occupata o non c'è, sto diminuendo il parallelismo

es: istruzione memory e data memory col design harvard: in un ciclo riesco a fare sia fetch, sia load/read;

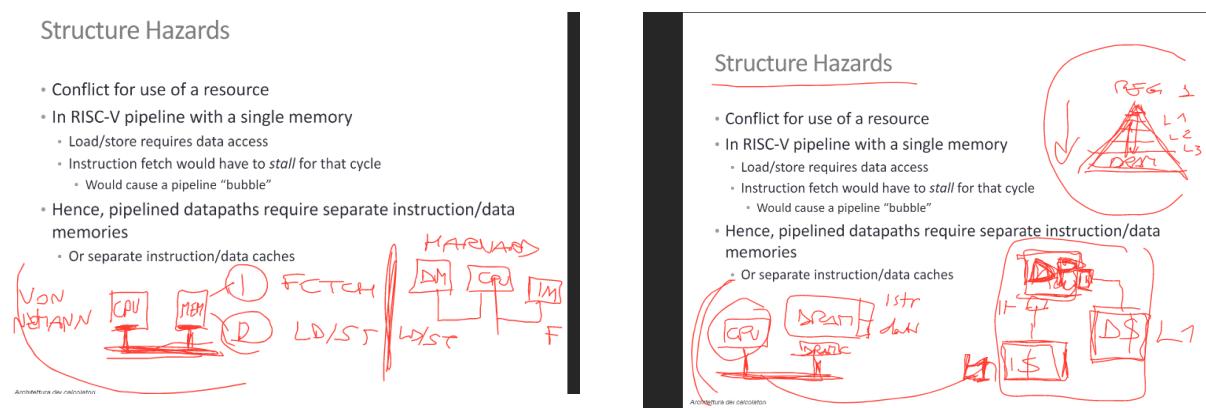
senza harvard non posso, perchè non ho un hardware che me lo permette → ho meno parallelismo perchè dovrei aspettare ogni volta.

→ manca la componente architetturale???? @maxbubblegum47 "si diciamo che ho un solo bus e quindi mi attacco al tram: è come se devo andare a prendere i bambini a scuola e ho due figli che vanno in scuole diverse, se mi aiuta mia moglie bene, se no ci metto il doppio. Una cosa proprio di limitazione architettonica come dici tu" @pablo remirez **OK**

i nostri pc hanno un CPU con un livello generale come von neuman → sarebbe un problema;

ma in verità dentro alla CPU sono come harvard → ho 2 cache di primo livello che separano data da instruction

→ al livello uno sono separate proprio per questo motivo: **voglio poter fare una load di instruction e una load/save di dato!**



data hazard

se una add scrive sul registro 1 e quella successiva legge → non riesco a farlo in un ciclo, perchè normalmente il registro viene aggiornato al WB della prima add

se per accedere ad una risorsa devo aspettare che legga/scriva la precedente istruzione

→ in pratica solo le dipendenze di dato

da 0 a 200 faccio la instruction fetch(primo ciclo di clock), poi faccio la decode...

in rosso è evidenziata una dipendenza di dato!!

uno schema pipeline ideale dovrebbe mettere la sub subito in coda dopo la add non posso farlo però!

perchè la scrittura dentro ad x19 avviene nel quinto ciclo(WB), ma la sub chiede il valore di x19 al terzo ciclo → rompe la semantica del programma!

certe fasi sono disegnate in 2 colori → quella colorata significa quando avviene l'istruzione → se è a sinistra fronte di salita, se è a destra a fronte di uscita
→ questo significa che io in fronte di salita posso scrivere in X19(WB) e sul fronte basso(di uscita) del clock esegue la decode della sub → in un ciclo di clock scrivo in x19 e prendo il dato da x19

però per farlo io devo spostare la sub in avanti:

bisogna inserire delle bubbles, stalli o nop(no operation)

→ non faccio fare istruzioni per 2 cicli; tra la add e la sub inserisco due **nop**.

questo funziona, ma avrò perso 2 cicli nel farlo

come capisco quanto nop mettere?

capisco dove sta la dipendenza, allineo le istruzioni per eliminare il problema e dopo a mano(li metto io) aggiungo tanti nop quanti sono gli spostamenti che ho fatto.

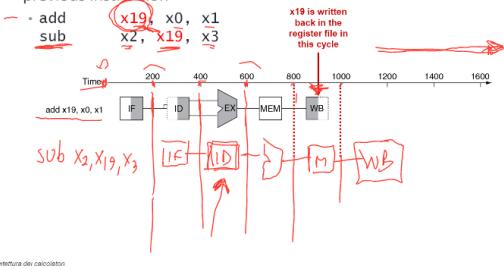
si usano questi stalli per allineare le istruzioni con dipendenza.

per come è fatto il nostro HW noi siamo vincolati ad attendere 2 cicli perché il dato venga scritto nel register file, così la sub può prenderlo.

ma se io potessi mandare il dato prodotto in uscita alla ALU, direttamente alla ALU della sub(bypassando la MEM e WB) risolverei il problema...

Data Hazards

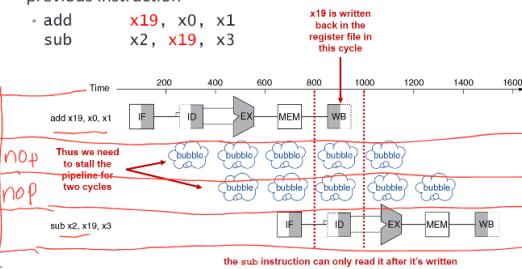
- An instruction depends on completion of data access by a previous instruction



Architettura dei calcolatori

Data Hazards

- An instruction depends on completion of data access by a previous instruction



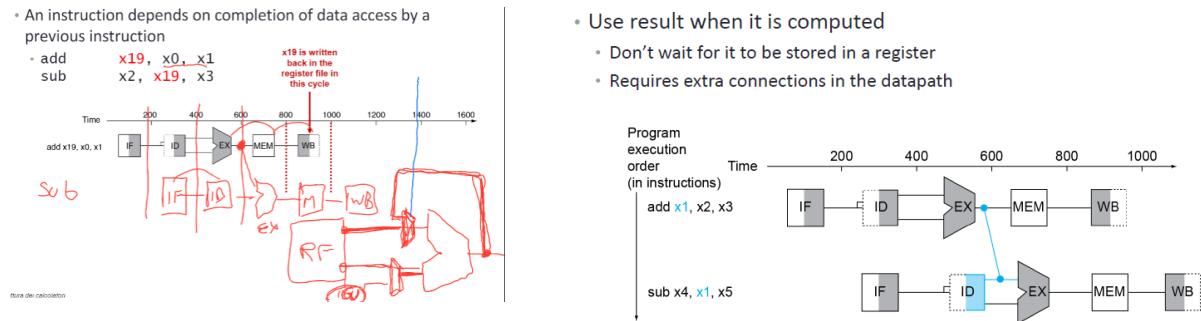
Architettura dei calcolatori

dovrei dire alla ALU di considerare come ingresso, non più quello che arriva dal register file, ma quello prodotto da lei stessa il ciclo prima!

forwarding:

complico la logica con dei controlli per creare questa cosa, però seguo sempre il credo di rendere il caso comune più veloce

→ estendiamo il nostro hw perchè decida in maniera efficiente queste cose.



ho un nuovo hazard: quando si usa un dato che deriva da una load → anche qui c'è solo 1 ciclo di differenza, ma in questo caso non posso risolvere come prima.

qua sono costretto a mettere una **nop** per forza, perchè se li allineassi normalmente, io dovrei dare il dato indietro nel tempo e non posso, mentre prima lo potevo fare esattamente nello stesso ciclo.

questo perchè una load prende un dato dalla memoria, quindi deve arrivare allo step MEM → in pratica una load può fare forwarding solo in fronte di uscita di MEM??

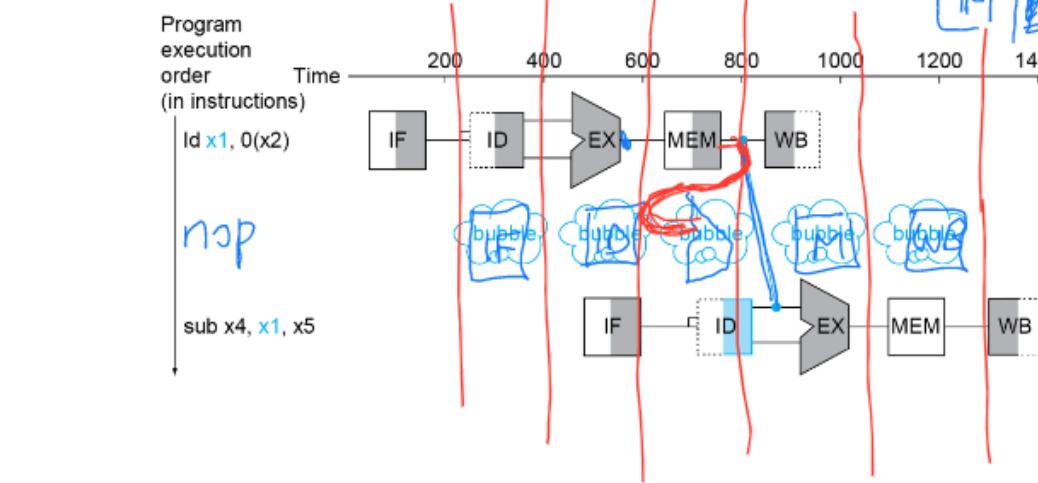
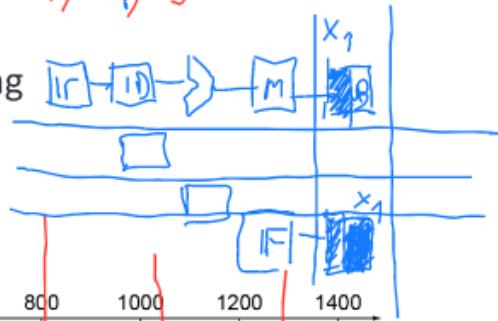
@maxbubblegum47 mia interpretazione "Si praticamente dobbiamo immaginarclo schema che ha fatto lui sotto e possiamo direi che una load può dare il suo output come input della ALU della sub, senza stare ad aspettare che ci sia anche il writeback, tanto questa fase operativamente non sta facendo nulla che possa intralciare la sub; l'idea è che appena posso subito incastro l'altra istruzione, in questo caso non appena leggo il dato dalla memoria subito faccio l'operazione di sub. L'unico packo è che come hai detto tu dopo, per allinearmi perfettamente nel ciclo devo aspettare 1 ciclo a vuoto con le bubble." @pablo remirez **OK**

la load utilizza la mem per forza, perchè il dato lo prende dalla memoria → io ho bisogno che la ex della sub sia nel ciclo successivo a quando ottengo il dato; nell'esempio della add era allineato bene (perchè add può dare già il risultato in fronte di uscita della EX), mentre nella load arriva da MEM quindi più avanti di prima → devo per forza stallare 1 ciclo, poi faccio forwarding

Load-Use Data Hazard

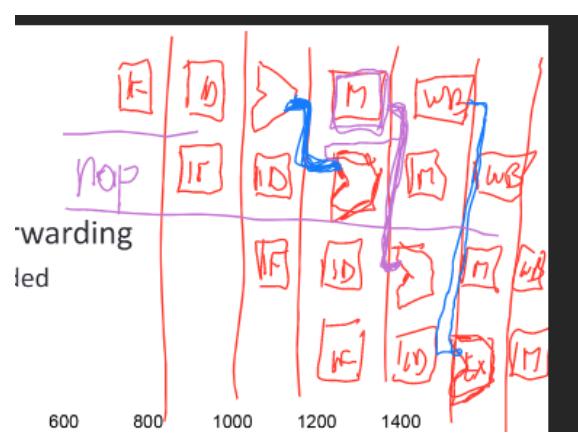
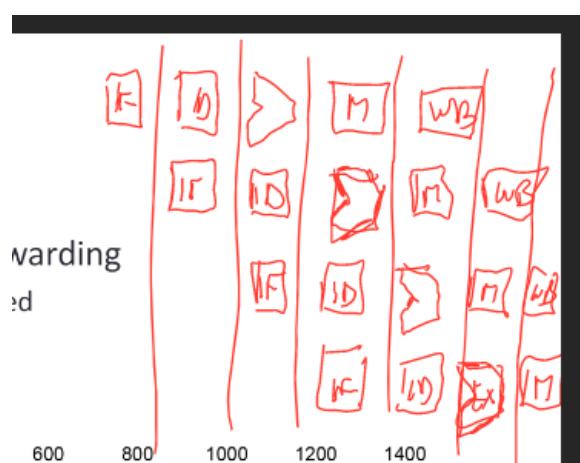
ld $x_1, 0(x_2)$
sub x_4, x_1, x_5

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!



ovviamente se non avesi logica di forwarding, la load sarebbe uguale alla add, quindi due nop per stallare 2 cicli;

mentre con la logica forwarding, la add lo fa subito al ciclo successivo; mentre la load ha bisogno di 1 ciclo stallato → perchè il dato non lo posso usare finchè non sono arrivato alla fase di memory





ovviamente quando è possibile è preferibile sostituire la nop con un'istruzione del programma → guadagno tempo → **code scheduling**
il compilatore riordina un po' il programma per guadagnare tempo, sostituendo alle nop, istruzioni utili.

siccome la pipeline ha 5 stadi → 5 cicli per la prima istruzione, poi 1 ciclo per tutte le altre istruzioni.

(nell'immagine considera la pipeline già piena, se fosse vuota avremmo 13 cicli , contro 11 senza le stall).

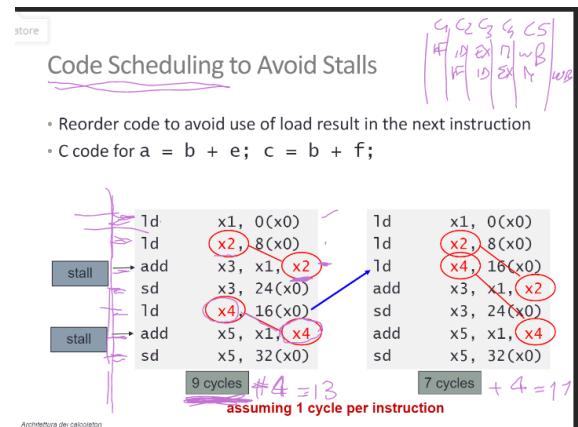
il compilatore intelligente ha osservato la load x4, ha visto che non dipende dal codice scritto prima, quindi l'ha messa subito all'inizio, insieme alle altre, evitando così i 2 nop

control hazard

il risultato di una branch dipende dall'istruzione prima → ho una decisione che dipende dai dati generati dall'istruzione prima.

le branch sono molto problematiche perchè non si sa mai se avviene PC+4 o PC+x

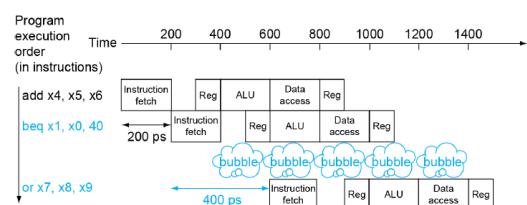
non ho capito se questo è l'esempio con branch prediction(ma non penso perchè devrebbe già fetchare altre istruzioni dopo) oppure fa vedere normalmente come funziona, però pensavo stallasse di più non solo 1 ciclo @maxbubblegum47
"dici quello che hai scritto sopra col PC + 4? beh hai ragione, perché alla fine la questione è che se mi sbaglio devo fare flush di tutta la pipe, perché lo scopro alla fine se faccio branch predicton e poi non



- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In RISC-V pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



ci prendo con la predizione; allora poi marongiu parla di modelli di predizione statica e dinamica mi pare che hanno le loro peculiarità e si basano su paradigmi di predizione diversi, uno mi pare che prenda il caso comunie nel dcodice e quindi prima fa due miss e poi in base a come sono andate fa la predizione, altri che lo fanno solo a compilation time.

Come esempio ci può stare secondo, me ti stavi effettivamente riferendo a questo discorso." @pablo remirez

no io dicevo proprio questo esempio → dice che aspetta a fare la prossima istruzione, ok, ma perchè aspetta solo un ciclo? non ho capito come funziona dentro mi sa, nel senso è come la load che deve arrivare a memory? o arriva a EX come le add?

perchè se è come le add allora ok deve fare una nop se stalla, ma solo perchè la dipendenza sta nella instruction fetch della prossima istruzione, non più nella decode.

si penso sia così alla fine e dopo mi devo essere sbagliato perchè ho scritto che stalla per 3 cicli dopo la branch e l'immagine prima dice che diventa molto problematico perchè tu devi proprio stallare, non puoi irmpiazzare con altre istruzioni, da qui il discorso sulle prediction...

ha senso? @maxbubblegum47

questo dovrebbe essere la predizione base spiegata → ovvero semplicemente non aspetto, ma continuo a fetchare @maxbubblegum47 "sì sì è questa, io faccio quello che devo fare e alla fine di tutto scopro se ho fatto una cazzata

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In RISC-V pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

oppure no, come quando trombie senza protezione e la tipa dopo 4 settimane ti dice che è incinta" @pablo remirez

okahahah

adesso spiega come applicare la predizione base → ovvero le 2 tipologie: static e dynamic.

static branch prediction:

tecnica che permette di ipotizzare se l'istruzione sarà vera o falsa e quindi il salto da fare.

→ la predizione avviene su analisi di dati: dati di profile; dati ottenuti tramite l'eseguire il programma tante volte, quindi ho le percentuali.

devo modificare il datapath in modo che possa recuperare l'istruzione nel caso la predizione fosse sbagliata.

→ analizzo il programma (prima dell'esecuzione) e mi faccio un'idea; ad esempio se ho loop di cui conosco già tutto, so il lower bound, l'upper bounds, (e a che me serve???)

da qui posso fare il calcolo(un loop con 10 iterazioni 90 volte ne fa 7 e 10 volte ne fa 4....).

dati di profiling, lancio il programma e dopo ho dati???????

→ però non misura i dati ??????????

dopo una branch devo stallare 3 cicli!! perchè il punto problematico della prossima istruzione non è più la fase di decode, ma è la fase di fetch!! perchè 3 cicli?

@maxbubblegum47 credo sia falsa questa e che in verità sia 1 ciclo

dinamic branch prediction

è un pezzo che ci permette di osservare...?????????

dal momento che devo avere una logica che risolva il problema nel caso la branch prediction abbia sbagliato → devo fare un flush della pipeline.

lui osserva il programma e da percentuali di come avviene la branch, per avere meno rallentamenti → fare la flush rallenta molto.

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

ha HW che osserva da solo (non so cosa significhi @maxbubblegum47):
 all'inizio prova e basta, poi man mano l'hw ha una tendina in cui si annota se nelle ultime volte il branch ha fallito o è avvenuto.
 però predice, quindi sbaglierebbe sicuramente, possiamo solo diminuire il numero di volte in cui sbaglia → il numero di volte in cui fa flush della pipeline.
 → è migliore dello static branch predict.

l'idea principale è quella di inserire registri in mezzo a queste 5 fasi delle pipeline di riscrivere

i registri prendono come nominativo l'acronimo della fase precedente e quello della fase successiva.
 ovviamente non basta questo, vanno fatte modifiche:

esempio con una load.

il calcolo dell'indirizzo di una branch si trova nella fase EXMEM e poi il nuovo PC lo calcola subito dopo.

@maxbubblegum47 ma che c'entra con la load ahah

il valore del PC viene propagato in avanti ad ogni ciclo fino alla EXMEM, così nel caso fosse una branch, allora lo avremmo pronto.

anche l'immediato viene salvato nel registro!

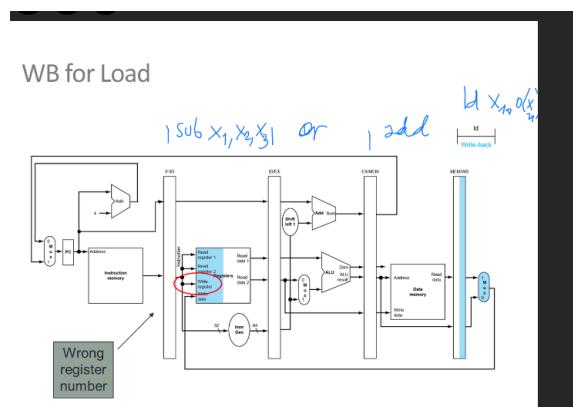
nel registro IDEX avrò il registro data 1, il registro data 2 e l'immediato (adesso a 64bit)

fase di memory → il dato letto dalla memoria lo salvo nel registro MEMWB.

nel Wb c'è un problema → la pipeline può essere riempita da altre istruzioni.

→ il risultato della load va in x1 invece di x10, perché il write register è stato settato a x1 dalla sub, che in quel momento si trova in EX e quindi ha settato i vari registri in base ai suoi criteri

il 2 non lo posso più scrivere dentro a x10, perché la sub che in quel momento

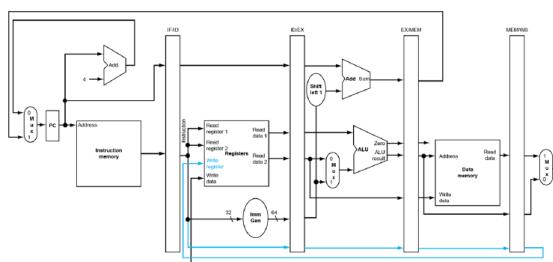


è nella fase di decode, sta settando x1 come write register!
non ci siamo salvati x10

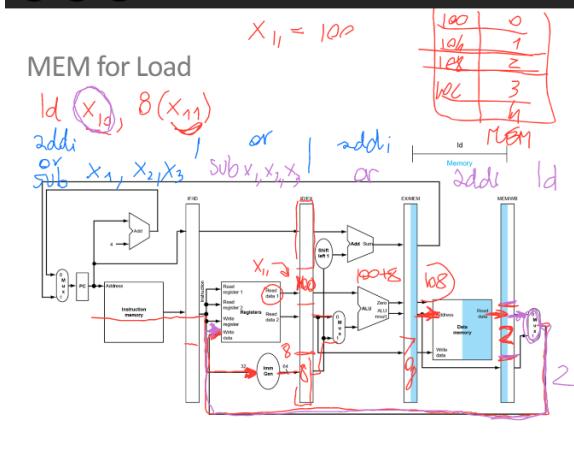
tutte le istruzioni con WB hanno questo problema.

qui avviene la prima modifica:

io x10 l'ho decodifica in ID, quindi invece di perderlo, lo passo in avanti ad ogni ciclo → così alla fine farò tornare indietro il registro dentro alla writeback

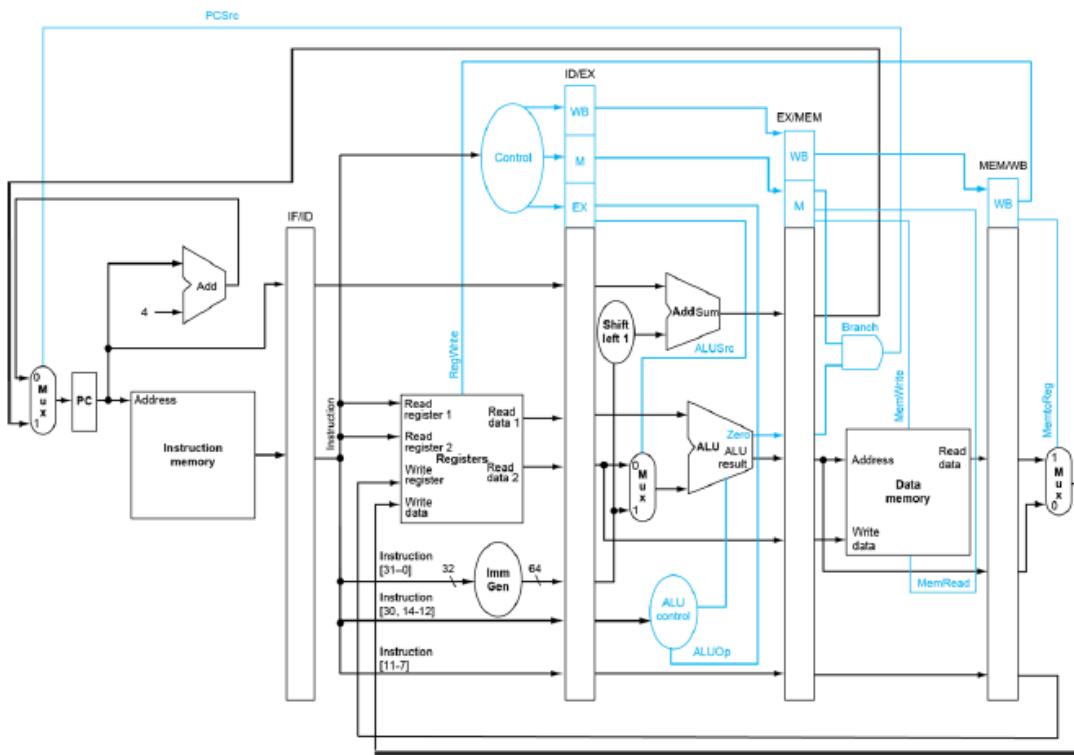


piccola nota personale, da qui si vede che il branch ritorna il risultato nella fase di MEM e quindi si deve aspettare fino a MEM



i segnali sono quasi sempre i soliti
il tag zero della ALU viene salvato nel registro e poi servirà per il salto o no del branch.
tutti i segnali sono salvati dentro ai registri.
→ anche il segnale di controllo va propagato tra gli stati
i segnali vengono settati nella fase di decode, però non vengono consumati subito →
vengono portati avanti fino a che non vengono usati

Pipelined Control



interessante che il regWrite viene settato dal WB e non direttamente dal decode, giustamente se lo facessi subito, dopo sarebbe sovrascritto

logica di controllo gestisce gli hazard???

sappiamo che il forwarding gestisce gli hazard, ma come avviene la hazard detection?

abbiamo il tempo espresso in cicli di clock.

tra la prima e la seconda c'è un forward normale senza stall; tra la prima e terza ha uno stall (che è riempito dalla seconda istruzione) ecc...???

come mai c'è un forward dalla sub alla or, però parte dalla fase di memoria della sub???

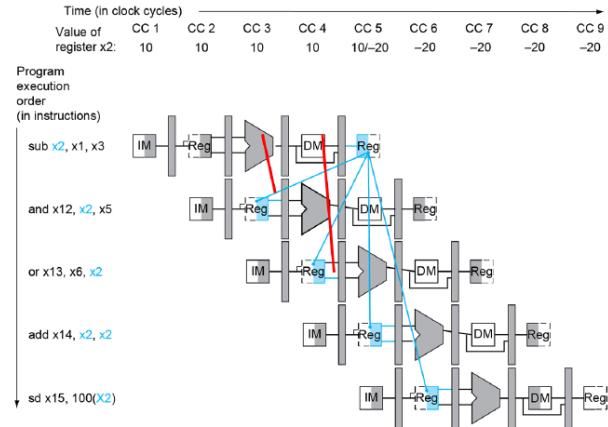
pensavo che essendo un'istruzione di tipi R, potesse fare forward solo in uscita dalla EX??? @maxbubblegum47 forse tutt'ora che ha fatto la EX, allora il dato ce l'ha e può passarlo prima del WB? mia ipotesi

Data Hazards in ALU Instructions

- Consider this sequence:

```
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(x2)
```

- We can resolve hazards with forwarding
 - How do we detect when to forward?



si fanno 4 operazioni di forwarding possibili:

noi facciamo forwarding o dalla fase di EX/MEM o dalla fase di MEM/WB.

guardo il registro rd nella fase EX/MEM e lo confronto con il registro sorgente 1 e 2 in ID/EX(quindi l'istruzione successiva):

- se il registro in cui sto salvando della attuale istruzione in esecuzione, è uguale o al registro 1 o al registro 2 (sorgenti) della istruzione attualmente in decode ID/EX
 → allora vuol dire che ho una dipendenza di dato e quindi devo fare forwarding →
 quindi devo pilotare il multiplex che sta davanti alla ALU per fargli prendere quello che sta qua(in EX/MEM), invece di prendere come al solito dal register file(quello in ID/EX).

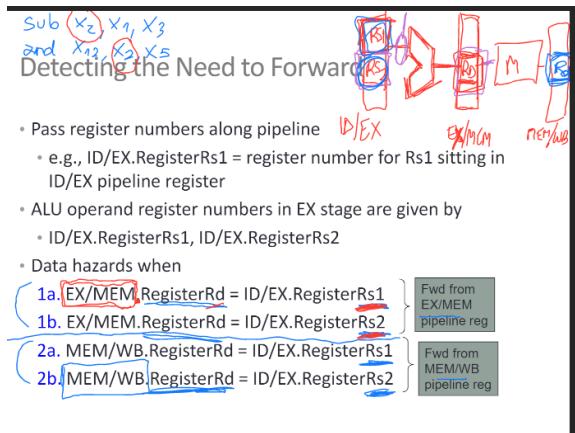
stessa cosa per i dati delle load, che li ho quando sono in MEM/WB → li controllerò il registro destinazione in MEM/WB invece di quello in EX/MEM.

si usa la notazione punto come se fosse una classe del C

ci sono altri controlli:

controllo che ho writeback, se no non passo il dato, perchè???? forse perchè se no vuol dire che io non scrivo in nessu registro, quindi non posso influenzare la operazione successiva, anche se il nome dei registri è lo stesso? mia interpretazione
 → se certi segnali sono alti: ... ????? @maxbubblegum47
controllo che il registro di destinazione non è x0!

- But only if forwarding instruction will write to a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not x0
 - EX/MEM.RegisterRd ≠ 0,
 MEM/WB.RegisterRd ≠ 0

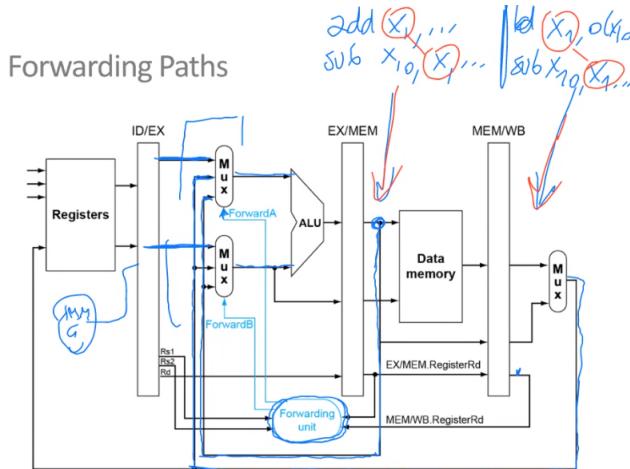


questi comandi sono dentro alla **forwarding unit**:

aggiungo 2 MUX che portano dentro ovviamente il percorso originale dal register file e anche l'imm generation unit perché può essere il secondo operando;

la cosa nuova è il percorso forwardato dalla fase di EX/MEM e quello forwardato dalla fase di MEM/WB

questi segnali li controllo tramite la forwarding unit che controlla come spiegato sopra → verifica i nomi dei registri... e se c'è da fare un forward modifica l'entrata nel MUX.



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

codifica delle MUX che determinano le entrate e c'è scritto il significato per gli umani

double data hazard:

al contrario dei precedenti esempi dove la prima istruzione scrive e le altre leggono; qui la prima istruzione scrive su x1 e le altre leggono da x1 e ci scrivono sopra.

la dipendenza è propagata per le istruzioni → abbiamo un hazard, però le nostre condizioni per il forwarding non sono applicabili qua, perchè?

succede che in uscita alla fase di EX della prima add ho il dato che mi serve e posso fare il forwarding alla fase di EX della seconda add nel ciclo successivo, poi c'è una terza add;

il dato della prima add che si è propagato alla fase di MEM, può essere mandato in avanti all'ingresso della terza ex e fin qua tutto nella norma.

→ ora però il dato che voglio nella terza add non è l'X1 in uscita dalla fase MEM della prima add, ma l'X1 in uscita dalla fase di EX della seconda add nel secondo ciclo(quello in viola)

double data hazard: ho 2 istanze dello stesso problema, la prima istanza è quella tra le prime 2 add e la seconda istanza è quella tra le ultime 2 add.

manca un controllo per verificare che non ci sia stato un double data hazard; dobbiamo rivedere le condizioni di forwarding che avvengono nella fase di memory hazard condition:

dobbiamo attivare la logica di forwarding di MEM/WB solo se non è vera la condizione di hazard per la fase di EX(dell'istruzione successiva o sempre della stessa istruzione???)

→ allora penso voglia dire che nell'istruzione che potrebbe avere un data hazard in MEM(quindi la prima), bisogno controllare che non ci sia un data hazard in EX(che per forza di cose è la seconda istruzione); se c'è allora non faccio la MEM forwarding, perchè ho un hazard più recente(in EX) giusto???

tra l'altro funziona perchè se ho un hazard in EX e poi il ciclo successivo ce l'ho in MEM vuol dire che il registro di destinazione della prima istruzione è usato sia dalla seconda, sia dalla terza, questo lo dice la definizione di hazard → ergo abbiamo un double data hazard.

come si risolve? con questo controllo, perchè se avviene, allora noi vogliamo usare il forwarding dell'hazard più RECENTE.

dimmi se ti tornano queste ultime 7 righe @maxbubblegum47 si allora vedendo il codice mi è più chiaro che spiegato, alla fine sono due hazard uno dietro l'altro e me li posso rioslvere connun forward a cascata(?)

si una sorta di cascata, nel senso che il primo per ovvie ragioni è quello che abbiamo sempre usato, quindi da EX/MEM della prima istruzione mando il dato a ID/EX della prossima istruzione;

ma al secondo hazard sempre della stessa istruzione(la prima), io devo ignorarlo e usare il secondo forwarding

dall'immagine si capisce molto bene.

MI RIMANE UNA DOMANDA → questo succede perchè anche il secondo add scrive in x1 giusto??? se avesse scritto tipo in x5, noi avremmo usato i forwarding della prima istruzione e non sarebbe capitato nulla??? @maxbubblegum47

infatti se guardi il codice sotto c'è scritto:

se è vero che il regWrite è 1(quindi è un'istruzione che scrive su un registro)

se è vero che il registro di destinazione di MEM/WB è $\neq 0$ (ipotesi da prima)

se NON è vero che [regWrite di EX/MEM(istruzione successiva) è 1 e il registro di destinazione di EX/MEM è $\neq 0$ e il registro di destinazione di EX/MEM è \neq dal registro sorgente 1 di ID/EX]

→ che in pratica significa se è vero che non c'è una data hazard in fase EX CON LO STESSO registro sorgente di ID/EX; se ho un data hazard con l'altro registro allora non ho problemi e continuo con i miei forwarding.

se è vero che il registro di destinazione di MEM/WB è = dal registro sorgente 1 di ID/EX ALLORA fai forward del registro di destinazione di MEM/WB(quindi della prima istruzione) e lo mandi al registro sorgente 1 o 2 in fase ID/EX(terza istruzione) in base a quale codice ha preso.

Double Data Hazard

- Consider the sequence:

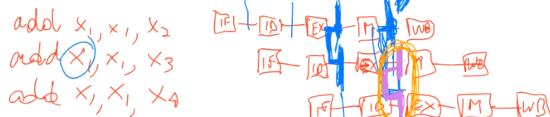
```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
```

- Both hazards occur

- Want to use the most recent

- Revise MEM hazard condition

- Only fwd if EX hazard condition isn't true



Architettura dei calcolatori

Revised Forwarding Condition

- MEM hazard

```
if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd  $\neq 0$ )
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq 0$ )
            and (EX/MEM.RegisterRd  $\neq$  ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

```
if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd  $\neq 0$ )
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq 0$ )
            and (EX/MEM.RegisterRd  $\neq$  ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```



se il segnale di regWrite nel registro EX/MEM e se il valore del registro di destinazione è diverso da 0 → cioè è scrivibile; allora devo verificare che l'indice del registro che l'ultima istruzione vuole scrivere, sia diverso dal registro sorgente allo stadio prima.



se l'istruzione che segue(la seconda add) ha l'Rs1 o Rs2 uguale al registro destinazione che ho dentro MEM/WB(della prima istruzione) e il regWrite è settato a 1(voglio scrivere);

allora vuol dire che ho un double data hazard → quindi non è più da MEM/WB che devo fare il forward, ma lo devo fare da EX/MEM(quindi dalla seconda istruzione)

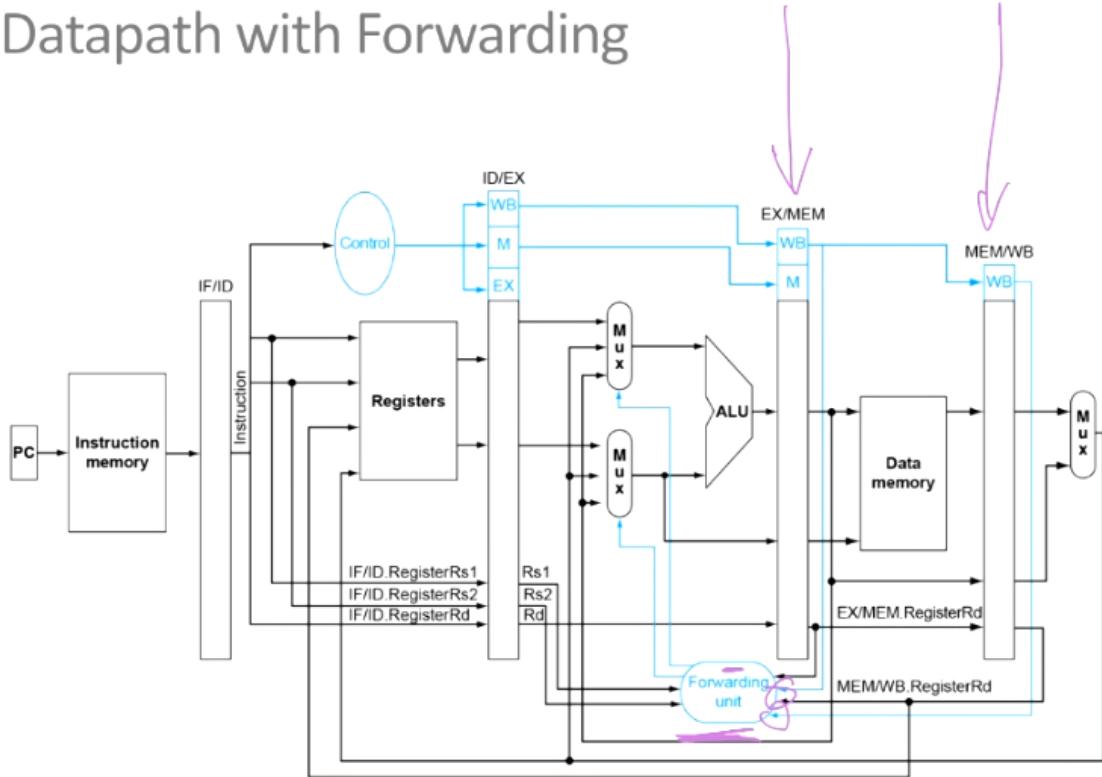
però così non basta, devo anche dire che ho un hazard tra la seconda e la terza add, sullo stesso dato dell'hazard tra la prima e la seconda. giusto???

@maxbubblegum47

ci sono degli input in più che sono quelli che permettono di modificare la mia logica di forwarding dicendogli da quale dei registri deve arrivare il segnale.

dentro alla forwarding unit ci sarà della logica che decide quale segnale tenere e quale scartare

Datapath with Forwarding



Load-use hazard detection:

detection dell'hazard vero e proprio: la condizione per cui io non sono più in grado di mandare avanti dentro alla mia pipeline una nuova istruzione e quindi sono costretto a fare uno stall.

esiste un caso che nonostante il forwarding non si riesce a mandare avanti il dato e bisogna quindi stallare:

una load che carica su x1 un valore e un'altra operazione che legge quel valore (tipo una add).

→ essendo che in una load il dato è pronto solo dopo la fase di MEM, quindi non ce l'ho subito pronto dopo l'EX; nel momento in cui la load è in fase di memoria, la add è in fase di EX e quindi il forward dalla MEM dovrebbe andare indietro nel tempo; perchè il forward avviene alla fine della fase di MEM.

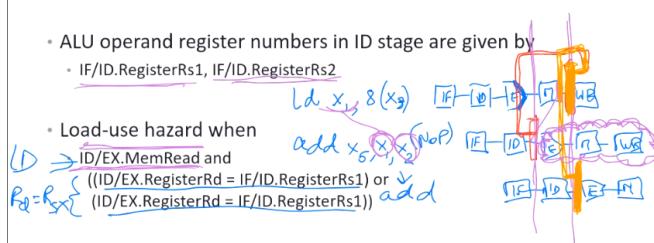
ho bisogno di aggiungere per forza una nuova istruzione → uno stallo, una nop, ... e quindi la mia add sarà spostata di 1 ciclo e adesso si riesce a fare forwarding perchè la fase di EX dell'add avviene dopo la fase di MEM della load

come faccio a rilevare questa condizione di hazard?

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage

- ALU operand register numbers in ID stage are given by
 - IF/ID.RegisterRs1, IF/ID.RegisterRs2



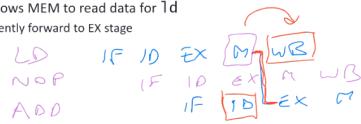
- Load-use hazard when
 - $ID \Rightarrow ID/EX.MemRead$ and
 - $((ID/EX.RegisterRd = IF/ID.RegisterRs1) \text{ or } (ID/EX.RegisterRd = IF/ID.RegisterRs2))$

- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
 - EX, MEM and WB do nop (no-operation)

- Prevent update of PC and IF/ID register
 - Using instruction is decoded again
 - Following instruction is fetched again
 - 1-cycle stall allows MEM to read data for 1d
 - Can subsequently forward to EX stage



- devo verificare che il dato che ha la dipendenza, non sia lo stesso che l'istruzione corrente sta cercando di scrivere
- devo controllare il registro in ID/EX, se MemRead è settato (io sono nella seconda istruzione, quindi non ho ancora toccato ID/EX)
 - allora so che l'istruzione che mi precede è una load e visto che è una load, controllo i registri:
 - se il registro destinazione di quella istruzione (quella precedente: la load) è uguale al mio operando sorgente 1 o 2 → allora io ho un hazard logico (???) quindi confronto il rd della load (ID/EX) con rs1 e rs2 dell'istruzione attuale (IF/ID, quella dopo la load).

quindi a questo punto ho trovato l'hazard, però l'istruzione di add è in decode, quindi devo fermarla prima di andare avanti a fare EX, MEM e WB

→ devo fare un controllo dei registri in ID/EX e devo azzerarli tutti (sono quelli della add adesso, perchè è passato 1 ciclo)

→ facendo questo il datapack interpreta l'istruzione che ho decodificato non più come una add (o quella che avevo), ma in una nop e quindi le fasi di EX, MEM e WB non faranno niente.

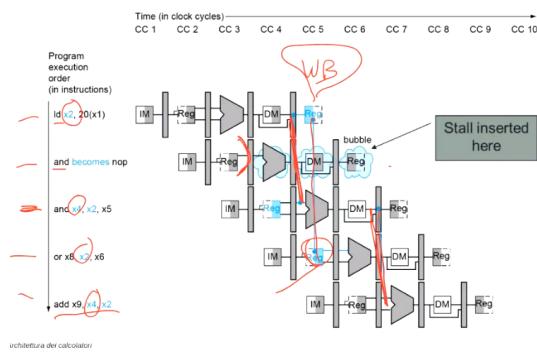
→ inoltre devo evitare che ci siano aggiornamenti del PC, perchè così al ciclo successivo, farò di nuovo la fetch e la decode della stessa istruzione → così facendo ho fatto ritardare di 1 ciclo e quindi dopo sarà possibile risolvere il problema con il forwarding.

tra la prima e la quarta istruzione c'è
dipendenza di dato, di x2

→ però quando la x2 è in fase di decode
(ID); la load è in fase di WB → quindi

sono allineati e quindi non c'è bisogno di fare alcun forward

Load-Use Data Hazard



per controllare questo abbiamo aggiunto una **hazard detection unit** che serve per rilevare questo problema, inoltre ci sono vari segnali che servono per questa logica(??)

nella hazard detection unit entrano il register destination di ID/EX e i due registri sorgenti di IF/ID.

entra anche l'ID/EX memRead
→ è quello che ci dice che l'istruzione precedente è una load

poi dalla HDU escono 3 segnali:

uno per il MUX che decide i segnali di controllo(per il forwarding)

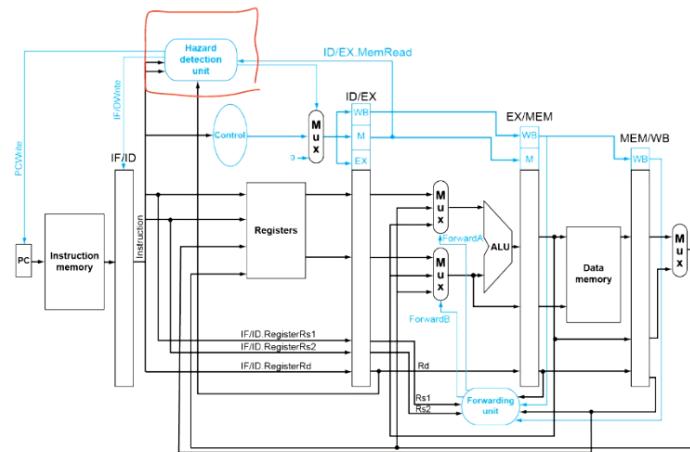
uno per il PC che decide se è da rifetchare la stessa

istruzione di prima(quindi non manda avanti il PC)

uno per IF-ID write che credo sia quello che setta tutti i controlli dell'istruzione da fare stallare a 0; quella che poi riverrà fetchata

@maxbubblegum47

Datapath with Hazard Detection





gli stalli ovviamente riducono la performance, però sono indispensabili per avere il corretto funzionamento del sistema.

sappiamo che il compilatore fa il code scheduling che l'addove dovrei inserire una nop, il compilatore prova ad inserire una nuova istruzione che non dipende da nessuna delle due adiacenti

anche nella branch si possono verificare degli hazard:
il control flow dell'applicazione raggiunge un punto in cui si altera come si sistema? con la predizione statica o dinamica.

nella branch io dovrò aspettare la fase di EX per fare la sottrazione tra x_1 e x_0 e vedere se il segnale 0 è stato sollevato, se il risultato della differenza è effettivamente 0 e quindi devo aspettare la fase de MEM per trovare il nuovo PC. quando arrivo a questo punto, mi accorgo che le istruzioni sotto hanno eseguito in maniera scorretta e quindi devo effettuare un flush di queste istruzioni.

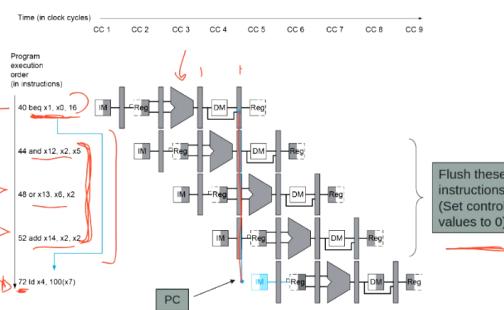
→ ovvero devo mettere tutti i valori di controllo degli stadi successivi della pipeline a 0 e quindi annullerà gli effetti delle esecuzioni di queste istruzioni

per ridurre il ritardo devo effettuare delle predizioni → devo spostare indietro nella pipeline i pezzi di hw che mi servono per determinare se il branch sarà preso o no quali sono i pezzi? l'adder per calcolare il nuovo PC e il comparatore che mi dice se i 2 operandi sono uguali tra di loro

→ io li voglio spostare fino alla fase di ID(prima non si riesce) → io sono in grado di sapere sia quali sono i 2 registri da confrontare e calcolare l'indirizzo target della nuova istruzione(ho già l'immediato nella ID).

Branch Hazards

- If branch outcome determined in MEM



Reducing Branch Delay

- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken
 - 36: sub x10, x4, x8
 - 40: beq x1, x3, 16 // PC-relative branch // to 40+16*2=72
 - 44: and x12, x2, x5
 - 48: orr x13, x2, x6
 - 52: add x14, x4, x2
 - 56: sub x15, x6, x7
 - 72: id x4, 50(x7)

(16 è moltiplicato per 2 perchè siamo in una branch → immediato shiftato a sinistra)

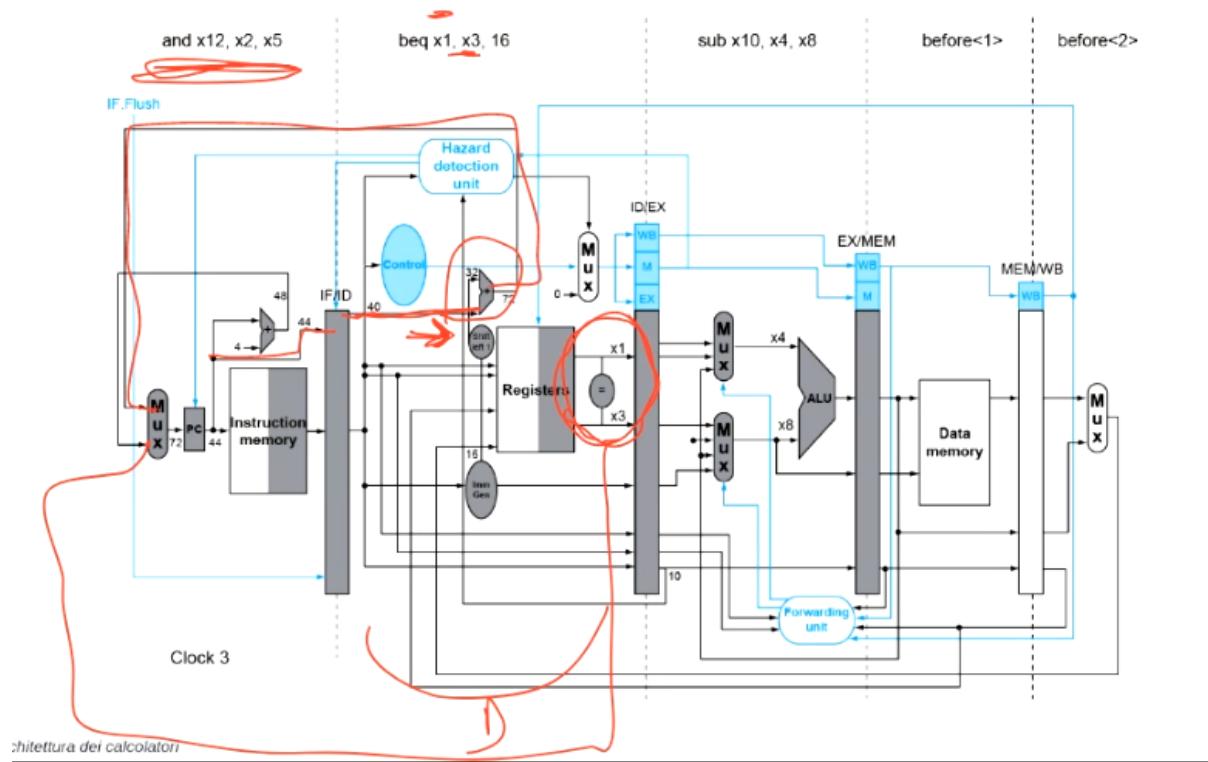
la branch è nella fase di ID e quindi nella fetch(IF) c'è già quella successiva(la and) → perchè? perchè ho usato la logica di predizione statica e quindi ha predetto che la branch non sarebbe saltata → inizia già a prendere le istruzioni dopo

- però abbiamo aggiunto un comparatore nella fase ID che confronta i registri x1 e x3(beq) → quello cerchiato di rosso
- un adder con la logica di shift che prende in ingresso il PC dal ciclo prima → sempre cerchiato di rosso
 - quindi già nella fase di ID io sono in grado di determinare se devo pilotare il MUX che c'è nella IF e così facendo se la beq avviene e devo quindi saltare ad un certo PC, verrà subito pilotato il MUX nella fase IF, così da prendere dentro l'istruzione a quel determinato PC(ce l'ho grazie al nuovo adder aggiunto).

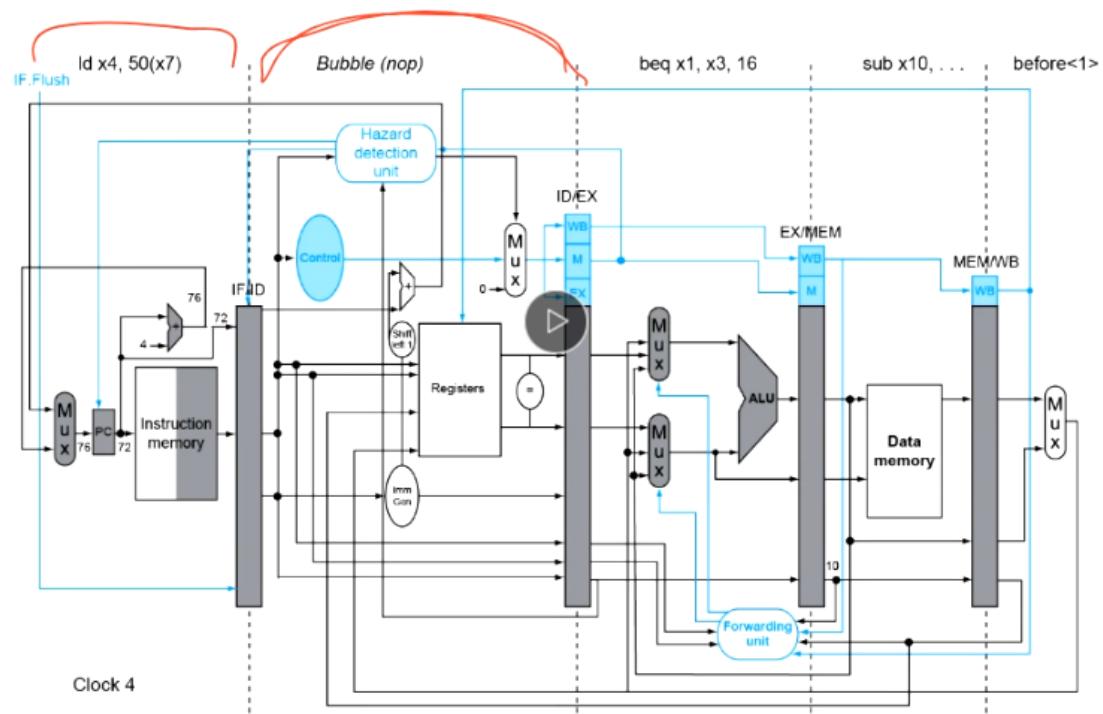
così facendo io non dovrò più flushare 3 istruzioni, ma solo 1, in questo caso solo la and è stata fetchata in maniera incorretta, non più tutte e 3 le successive operazioni

- al ciclo successivo scriverò una bubble nella ID; credo perchè ho cambiato tutti i valori di controllo a 0 @maxbubblegum47
- e la nuova istruzione fetchata è la load! come dovrebbe essere

Example: Branch Taken



Example: Branch Taken



in alcuni casi avviene la predizione dinamica:

ci vuole un buffer che è fondamentalmente una tabella della storia del sistema

→ si utilizza l'indirizzo delle istruzioni che sono state recentemente indirizzate e lui si memorizza quale è stato l'esito dell'esecuzione: branch preso o branch non preso

quindi per eseguire un branch viene prima verificata questa tabella, dopodichè si fa il fetch dell'istruzione sequenziale o con salto in base a quello che abbiamo trovato dentro la tabella

→ infine nel caso avessi predetto male, avviene una flush della pipeline e si cambia il comportamento dentro al predittore; perchè gli usi successivi si dovranno ricordare di questo evento.

come si implementa?

2 approcci:

- naive: si utilizza soltanto 1 bit per il predittore → ricorda solo se l'ultima volta che è stato usato il branch, è stato preso o no → 1 preso; 0 non preso

ha un problema, se avviene la stessa beq in due loop innestati:

si può ricorrere in un loop di predizioni sempre sbagliate, perchè se nella condizione con cui sto terminando il ciclo interno il predittore crederà che sta continuando a ciclare (perchè il bit sarà quello dell'uso precedente dove ha ciclato), quindi li per forza sbaglierà la predizione;

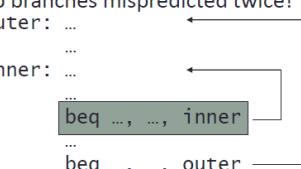
però utilizzerà la stessa predizione anche per il loop esterno e quindi avrà il bit a 0 perchè è appena uscito dal loop interno,

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Start fetching from fall-through or target
 - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

quindi il salto non è stato preso e quindi quando va per fare un nuovo ciclo del loop esterno, la predizione dirà di non prenderlo; mentre dovrebbe prenderlo(tranne per l'ultimo ciclo ovviamente).

quindi ci sono 2 misprediction una dopo l'altra ogni volta che rifaccio il ciclo esterno(tranne la prima e l'ultima) e tipo 2 misprediction su un codice da 10 istruzioni, sono già un miss rate piuttosto alto.

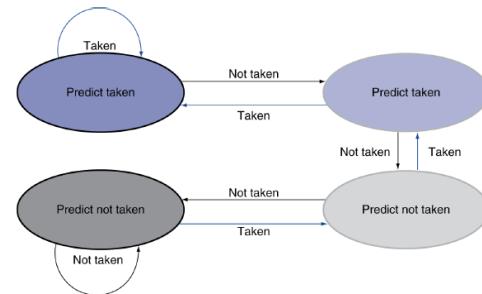
- si usano predittore da più bit, normalmente si usano 2 bit: mi assicuro che ci siano state almeno 2 predizioni successive, prima di cambiare il mio history record

tipo il caso dei loop innestati verrebbe gestito perfettamente:
quando arrivo alla fine del loop interno, ho un errore perchè la predizione lo prenderebbe, invece lui esce quindi non lo prende; però non viene cambiato l'history record, perchè è solo 1 stato non preso;
lui invece ne ha bisogno di 2 di fila per cambiare l'history record e quindi eseguirà un nuovo ciclo del loop esterno.
quindi ho cambiato stato, perchè non ho preso il branch, però sono nel secondo stato di branch taken → la predizione è ancora quella di prendere il branch.
(nell'immagine il predict taken di destra).

all'ultimo ciclo del loop esterno si sbaglierà ed avrà un non preso → essendo appena uscito dal loop interno, si troverà nello stato predict taken, ma quello verso metà(nella storia, l'ultimo branch non è stato preso), quindi adesso che il branch non viene preso di nuovo, lui passerà al nuovo stato di predict not

2-Bit Predictor

- Only change prediction on two successive mispredictions



taken(logica degli stati o più semplicemente nella storia ha gli ultimi 2 casi successivi di branch che non sono stati presi).

se incontro di nuovo lo stesso branch e di nuovo non viene preso, allora passo all'ultimo stato di predict not taken....



anche con il predittore io ho lo stesso bisogno di calcolare il nuovo PC, quindi ho sempre una penalità di 1 ciclo SOLO QUANDO PRENDO LA BRANCH; forse perchè fetcha lo stesso 1 nuova istruzione e quindi se poi il branch salta allora metto una bubble????

l'unica cosa che mi viene in mente è che se prevede che il branch non viene preso allora inizia già a fetchare le istruzioni successive (PC+4); mentre anche se prevede che avverrà un salto, lui deve calcolare il prossimo PC e lo fa nella DECODE, quindi c'è 1 ciclo in cui la branch effettua la decode e non si può prendere nessuna istruzione perchè tanto avverrà il salto → si mette una bubble.

passato questo ciclo, la decode della branch in uscita ha il nuovo PC che da in ingresso alla nuova FETCH e quindi riprende la normale funzionalità.

@maxbubblegum47 dimmi cosa ne pensi di sto ragionamento che non ne sono per niente sicuro.

in ogni caso non capisco il guadagno, guadagno che non faccio dei flush nel caso si predice un salto?? però non ho capito quando avviene la previsione???

DOMANDA PROF??

posso avere una logica di buffer, nella quale ho una piccola cache di indirizzi target che ho visto nella mia storia recente.

la mia cache può essere indirizzata dal PC(usato come input; ad esempio in un loop vedrò spesso lo stesso PC più volte:

→ se ho una hit in cache, vuol dire che ho già osservato questo PC prima → allora piuttosto che spendere tempo per calcolare l'indirizzo target del salto, lo tiro fuori da questa cache e quindi posso fetchare immediatamente al ciclo dopo l'istruzione del target PC????? → vuol dire semplicemente che se trovo un PC già visto, allora non sto a calcolare il nuovo PC, ma metto direttamente quello solito al posto di PC+4?

@maxbubblegum47

anche tu hai un vuoto dalla slide 62 alla fine del capitolo il processore pipelining?????
@maxbubblegum47

DIOCANE SALTA L'AUDIO

correzione parziale:

indice

processore - instruction level parallelism

paradigmi di design avanzato per migliorare le performance delle cpu:

- parallelismo a grana più grossa → parallelismo tra thread → cpu multicore → più processori che lavorano in parallelo

instruzion level parallelism

parallelismo a livello di singola istruzione → come sfruttare il parallelismo TRA le istruzioni del nostro programma → mandare in esecuzione più istruzioni alla volta.

pipelining è alla base di questo parallelismo; le operazioni che devo svolgere per eseguire delle istruzioni possono essere classificate in degli step.

→ gli step possono essere altrettanti stadi di una pipeline → stadi realizzati tramite blocchi hw che possono eseguire in parallelo

spezzettare le istruzioni mi permette di avere a regime (quando la mia pipeline è piena) tante istruzioni che eseguono in contemporanea (parallelo)

si può aumentare il parallelismo?

lo speedup ideale è pari alla profondità della pipeline; ma allora perchè non si fanno più stadi nella pipeline?

aumentare la profondità della pipeline → più stadi → partizionare ulteriormente in blocchi più piccoli il lavoro che viene fatto in un certo stadio.

questo riduce anche il cammino critico in caso peggiore (spezziamo la logica camminatoria in qualcosa di più piccolo) → anche il ritardo del segnale elettrico è più piccolo → ciclo di clock più corto → frequenza più alta.

ci sono dei limiti:

a un certo punto si raggiunge un limite per il quale non riusciamo a sfruttare più parallelismo a livello di istruzione(quello che c'è dentro ad un programma); e rendere le pipeline profonde, significa aggiungere della logica → costo.

allora si raggiunge un tradeoff senza benefici.

ci sono anche tanti meccanismi speculativi(le branch ecc...) e fanno buttare via del lavoro nel caso sbaglio la predizione → performa di più la CPU → più lavoro → problematiche...



miniaturizzazione dei transistor → nuove problematiche sui consumi di potenza.????? @maxbubblegum47 sono le solite problematiche che dopo un po' generi troppo calore ecc... ???? "si allora la questione è sempre che al tempo si sono scontrati con il power wall: cercavano di aggiungere sempre più logica e stati alla pipe per aumentare la frequenza ma il consumo energetico == il calore erano spropositati e non sapevano nemmeno come fare; ad oggi siamo in una situazione inversa in un certo senso, perché riusciamo a contenere di più i consumi energetici, il calore e tutto, ma mettere tanti core non è comunque una soluzione efficiente perché tutti i nostri programmi non sono pensati per sfruttare questo tipo di parallelismo hardware e a dire il vero tutti i nostri paradigmi di programmazione sono diciamo orientati verso il single core; ovviamente avere più core ci porta ad ottenere uno speedup, ma entro un certo limite; la sfida di domani è la programmazione in grado di sfruttare questo parallelismo di core, ma richiede uno sforzo enorme per chi programma. Ecco perché avere tanti core se non stai virtualizzando molte macchine virtuali e quindi non sei un server, non ha senso. Per dirti nel mondo dei videogiochi il 90% dei giochi sono studiati per 4 cores, ovvio che se ne hai 6 e vanno forte ci guadagni qualcosa, ma non prendi tanti più fps di un buon 4 core alla giusta frequenza. Proprio una questione di architettura del codice dei programmi, e dovremmo invece andare di più verso la legge di Amdahl." @pablo remirez molto figo grazie

pipeline più lunga diventa sconveniente dopo un po' per questo si smettono di fare singole pipeline e si fanno CPU multicore.

→ più semplici e con pipeline più semplice, però sono tanti! "Esatto esatto, l'unico grande limite adesso è la questione dei programmi che devono sfruttare tutto questo ben di dio; poi un'altra cosa fica ultimamente è che si sta passando da X86_64 ad ARM (vedi che cosa sta facendo Apple). ARM, da quel che so, ma non sono esperto, è un po' come il riscv" whaaaaat figo

altro modo per migliorare il parallelismo è di replicare le instruction parallelism rimanendo sul singolo core:

replicare le pipeline o alcuni stadi → **multiple issue processor** → in un singolo ciclo di clock, questi processori possono completare l'esecuzione di più istruzioni.

devo replicare i blocchi logici delle istruzioni → CPI < 1 o a volte si usa IPC

esempio: con un processore a 4 GHz con una multiple issues a 4 vie → in grado di ritirare 4 istruzioni per ogni ciclo; ho un CPI di 0,25 → IPC di 4

in realtà la problematica delle dipendenze tra i dati riduce questi valori di CPI e IPC

la dipendenza tra i dati riduce questo CPI o IPC

multiple issue può essere realizzato in modo statico o dinamico.

- **statico:** quando compiliamo il programma, in quel momento andiamo a studiare le istruzioni e vedere che dipendenze ci sono tra l'uso dei registri(data hazard, ecc...). è il compilatore che capisce dove NON ci sono dipendenze tra le istruzioni e le impacchetta in maniera che stiano a questi slot(bundle) di istruzioni che possono essere mandate sulle varie pipeline(abbiamo detto che i vari datapath o parti di esso sono replicati).
 - la detection e l'evitare i data hazard è fatto in maniera statica, cioè a tempo di compilazione dentro al compilatore.
- **dinamico:** questo controllo avviene mentre il programma esegue
 - la CPU ha della logica aggiuntiva che esamina lo stream di istruzioni e sceglie quali istruzioni possono essere eseguite e quali hanno una dipendenza quindi stanno in stall.funziona a runtime, quindi quello che non si sa a compile time;
 - tipo un puntatore che i suoi indirizzi non sono risolvibili staticamente; oppure quando lavoro con loop con un numero di iterazioni non noto, che viene determinato a runtime → allora in sto caso l'approccio dinamico mi aiuterebbe

ovviamente posso già compilare un programma che ordini le istruzioni per multiple issues, però ci saranno sempre casi in cui questo non sarà possibile.

in quei casi, l'approccio statico deve usare un approccio più conservativo: invece di usare tante vie, ne userà solo una → si comporta come se fosse una normale CPU la single issue. ??? @maxbubblegum47

→ sta cosa vuol dire che se non riesco a fare multiple issue a causa di dipendenze di dati o altre cose, mi comporto come una normale CPU single issue perchè tanto non riuscire a guadagnare niente??? mia ipotesi

la speculazione:

è un tentativo di indovinare cosa avverrà del programma durante la sua esecuzione

→ noi vogliamo schedulare questa istruzione il prima possibile, perchè se io dovesse

aspettare che la dipendenza è risolta e quindi sono sicuro di quello che avverrà, perdo più tempo rispetto a indovinare subito come andrà(stesso ragionamento dei branch: se indovino subito se farà o no il salto guadagno cicli rispetto ad aspettare che finisca l'istruzione!)



la speculazione inizia ad eseguire l'istruzione che pensa sarà giusta; poi quando sarò in grado di stabilire se la mia ipotesi era corretta:

- se era corretta allora posso completare correttamente l'operazione, ovvero pubblicare permanentemente i suoi effetti sullo stato del sistema → register file e memoria.
- se non era corretta devo effettuare il **roll-back** → cancellare gli effetti parziali/temporanei di questa istruzione e ricominciare.
questo avviene sia per l'approccio statico, sia per quello dinamico.

esempi: se speculo(?) su una istruzione branch e ho sbagliato ad ipotizzare quale branch del control flow viene eseguito → roll-back
in una load il cui indirizzo di memoria non è ancora stato determinato per dipendenze(le istruzioni che lo trovano non hanno ancora terminato l'esecuzione);

- io posso speculare su quale sarà l'indirizzo, ci si basa sempre sulla storia del sistema(la cache dalla quale osservo gli indirizzi del programma e in base al PC provo a speculare: se io ho già visto questo PC, provo a riusare l'indirizzo che l'ha caricato l'altra volta → nel senso se ho già visto questo PC → allora ho già visto sta load allora spero che l'indirizzo da cui caricare il dato sia sempre quello gisuto? @maxbubblegum47
- se non ci becco allora devo fare roll-back → magari avevo anche iniziato a computare sul dato dentro all'indirizzo

in ogni caso so che non posso completare l'esecuzione di queste istruzioni prima di essermi accertato se la speculazione è giusta o sbagliata → prima di aver terminato le istruzioni che mi servono per validare la speculazione

speculazione del compilatore:

il compilatore può riordinare le istruzioni

- tipo spostare le load prima dei branch
- se l'approccio è interamente statico(quindi non c'è nessuna logica/hw che mi permette di verificare se sto speculando bene o male) → **il compilatore stesso può aggiungere istruzioni che verificano se la speculazione era corretta; quindi sposta in anticipo le istruzioni che vuole eseguire speculativamente e poi aggiunge nei punti opportuni, codice che controlla l'esito della speculazione**

speculazione del hw:

l'hw guarda più avanti nel flusso di istruzioni del programma:

ha dei buffer dentro ai quali pubblica temporaneamente il risultato delle operazione che esegue in maniera speculativa.

- se la speculazione non era corretta: deve solo cestinare il contenuto del buffer
- se le ipotesi erano corrette: i risultati nel buffer sono trasferiti nello stato del sistema(register file e memoria)

speculazione e exceptions:

se una eccezione avviene durante un'istruzione eseguita speculativamente → ad esempio una load speculativa che viene eseguita prima di aver verificato se il puntatore è null...

approccio statico → aggiungo un'istruzione dedicata che permette di spostare più in là l'esecuzione delle eccezioni, perchè devo attendere che l'esito dell'istruzione problematica sia noto.

approccio dinamico → bufferizzo le mie eccezioni fino al momento in cui l'istruzione è completata

multiple issue statica:

il compilatore è responsabile per il raggruppamento in bundle di istruzioni → gruppi di istruzioni messe di fianco l'una all'altra.

nel caso di multiple issue, tipo dual issue: io ho replicato il datapath o parte di esso 2 volte → posso eseguire 2 istruzioni in un ciclo solo

→ l'assembly è fatto in maniera tale che ogni riga possiede 2 istruzioni(normalmente abbiamo una istruzione per riga: una istruzione alla volta), tipicamente separate dal ; o altri delimitatori

→ quando compiliamo un programma scritto così con un compilatore in grado di fare static scheduling per un dual issue → il programma che viene fuori ha due istruzioni per riga **NEL CASO MIGLIORE**

→ perchè vorrebbe dire che ho un IPC reale di 2 e quindi il compilatore è riuscito a schedulare tutto il programma mandando in esecuzione 2 istruzioni alla volta; ma sappiamo che non è sempre possibile

come funziona?

dobbiamo trovare tipi di istruzioni che utilizzano parti specifiche della pipeline.

le dipendenze di dato sono insite nell'uso di coppie di istruzioni; ad esempio la dipendenza tra load e use:

io quando carico un dato faccio la load e qualche altra istruzione lo usa

→ ha senso usare lo scheduling delle istruzioni in 2 slot in cui in uno c'è tipo di istruzione load e nell'altro tipo di istruzione aritmetica/logica/... → gli slot sono i famosi pacchetti giusto? @maxbubblegum47 o i 2 slot insieme sono 1 pacchetto?

→ se li mettessi nello stesso slot, avrei una dipendenza → non riuscirei ad eseguirli entrambi nello stesso ciclo.

dopo questa prima separazione; un pacchetto di istruzioni fatto nella maniera che volevo(tipo 2 istruzioni una dopo l'altra, o 4, o ...) va a costituire una **very long instruction word(VLIW)**:

→ i processori in grado di fare multiple issue si chiamano **very long instruction word processors**.

cosa succede quando il compilatore non è in grado di riorganizzare il programma perchè scheduli tutte le istruzioni in modo da utilizzare sempre tutti gli slot a disposizione

- devo riempire gli slot che non riesco ad usare con delle nop
- il mio programma mi mostrerà che spesso avrà schedulato 1 sola istruzione e ci ha messo di fianco una nop; più raramente sarà riuscito a schedulare 2 istruzioni

esempio:

una classica suddivisione delle tipologie di istruzioni per un dual issue sono:

1 che raggruppa tutte le istruzioni di tipo ALU/branch — 1 che raggruppa tutte le istruzioni di load/store

perchè ci sono quasi sempre dipendenze tra i tipi di istruzioni dentro ai gruppi(load con store,)

2 istruzioni → 32bit l'una(di solito si usa 32bit istruzioni) → 64 bit; se fosse stato quadruple issue, 128bit.

quando faccio il fetch dalla memoria, faccio il fetch di 64bit, non più 32

l'immagine mostra come nel tempo funziona la maniera IDEALE di schedulare le istruzioni: perchè la pipeline rimanga sempre piena il mio programma dovrebbe utilizzare sempre una istruzione di tipo ALU/branch e poi una load/store, poi di nuovo ALU e poi load e così via...

dovrei scrivere il programma in modo tale che all'indirizzo n ho una ALU/branch e a tutti gli indirizzi con n + multiplo di 8(2 istruzioni, 64bit), ho una ALU.

mentre agli indirizzi n+4 + multiplo di 8 → ho load/store.

se riuscissi a schedulare il mio codice in questa maniera, allora otterrei la mia pipeline senza mai un buco.

dal punto di vista dell'hw mi serve

RISC-V with Static Dual Issue

- Two-issue packets
- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages				
		IF	ID	EX	MEM	WB
n	ALU/branch					
n + 4	Load/store	IF	ID	EX	MEM	WB
n + 8	ALU/branch					
n + 12	Load/store		IF	ID	EX	MEM
n + 16	ALU/branch			IF	ID	EX
n + 20	Load/store				IF	ID

replicare alcune risorse!

non è necessario replicare tutto; ad esempio la parte di PC, si possono determinare gli offsetti a cui dobbiamo spostarci nella solita maniera.

@maxbubblegum47 ma io adesso prendo sempre 2 istruzioni dal instruction memory, quindi non devo effettivamente calcolare 2 PC, però come funziona con le branch????

se salto ad un determinato PC, dopo io prendo sempre 2 istruzioni da quel punto? oppure ne prendo solo una da quel PC e l'altra la prendo dal vecchio PC??? penso sia la prima che ho detto ma non sono sicuro

DOMANDA PROF: quando ho una dual issue come funziona il PC?

tipicamente si replica:

l'ALU

le porte di accesso sulla Data memory.

la immediate generation unit.

le porte di accesso al register file e quindi anche quelle di uscita.

devo rendere più grande la porta delal instruction memory.

tutti i path di write back.

la problematica degli **hazard** di fatto ci limita nel nostro utilizzo del multiple issue:

abbiamo visto che il forwarding è una tecnica che ci evita completamente lo stallo, nel caso di single issue.

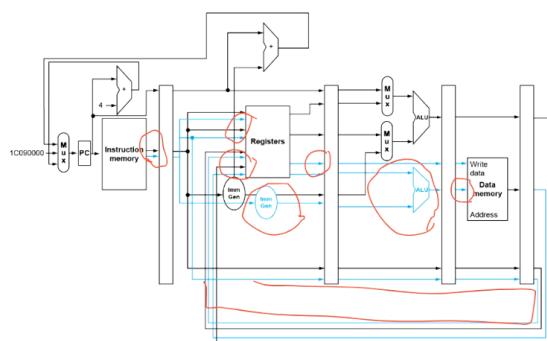
nel caso del dual issue cambia:

il problema rimane, perchè non posso mettere nello stesso slot 2 istruzioni che dipendono l'una dall'altra → devo per forza spezzettarle in 2 istruzioni.

→ quelle che prima in dual issue erano 2 istruzioni una dopo l'altra; adesso diventano 2 istruzioni una dopo l'altra, che però usano solo metà della mia very long instruction word
→ spreco di risorse.

@maxbubblegum47 questo perchè con il forwarding io riesco a passare il dato all'istruzione esattamente dopo, quindi il ciclo dopo?? mentre adesso l'istruzione

RISC-V with Static Dual Issue



esattamente dopo può essere eseguita durante lo stesso ciclo, eliminando la possibilità di fare subito forwarding?????

→ l'unica cosa che mi è venuta in mente è che non puoi farlo nello stesso ciclo perchè è la solita storia del non poter passare i dati indietro nel tempo, quindi devi farlo per forza nel ciclo dopo, ergo se hai solo quelle 2 istruzioni, dovrà mettere 2 nop → 1 da impacchettare con la prima istruzione e 1 con la seconda @maxbubblegum47 → non puoi fare forwarding dentro allo stesso pacchetto?

nel caso di una hazard di tipo load/use → avrà un ciclo di latenza(??), però devo di fatto utilizzare dual issue. non ho capito @maxbubblegum47

DOMANDA PROF: in che senso un ciclo di latenza quando ho un hazard load/use in un multiple issue?

normalmente mi servono politiche di scheduling del codice che sono più aggressive rispetto al single issue process.

esempio:

nell'immagine abbiamo un loop e lo dobbiamo schedulare per un risc-V dual issue:
il dual issue ha uno slot per istruzioni ALU/branch e un per quelle load/store.

la prima istruzione è una load che salva su x31.

la seconda istruzione ha già una dipendenza o data hazard → questa istruzione usa x31 per fare una somma, poi aggiorna il risultato rimettendolo su x31.

se è una store ceh deve mettere il risultato dentro all'indirizzo x20.

l'istruzione dopo usa x20, lo decrementa e lo rimette detto x20.

infine ho una branch che dipende da x20

→ lo scheduling è particolarmente complicato se voglio sfruttare l'instruction level parallelism:

se ho un single issue processor allora va bene come l'ho scritto. → mi basta la logica di forwarding per minimizzare le problematiche di performance

se invece devo schedulare 2 istruzioni nello stesso ciclo, posso provare a spostare le istruzioni, ma non riesco lo stesso a riempire le pipeline:

la prima load non ho modo di schedularla insieme a nessun'altra istruzione → il primo slot avrà la parte ALU/branch con una nop e la parte load/store con la load.

ciclo dopo in load/store ci potrei mettere la store di x31, ma questa store non posso schedularla insieme alla add perché sarebbe una dipendenza → non si può utilizzare.

→ cerci di anticipare l'esecuzione di altre operazioni:

anticipo la posizione della addi che da penultima posizione la metto al secondo ciclo di clock, in esecuzione; non posso schedulare la store, perché dipende da x31 dentro ad add, quindi devo aspettare(intuizione mia?????????).

stessa cosa per la add x31. → non è vero, secondo me non la schedulo perchè ho già messo la seconda addi...

che vantaggio ho avuto a schedulare prima la addi? → il fatto che poi alla fine posso schedulare la branch insieme alla store! → la store deve essere modificata, perchè ho già eseguito l'istruzione che decrementa x20, allora per far preservare l'indirizzo giusto di x20 per la store → si modifica l'istruzione mettendo un offset di 8 (io ho decrementato di 8 e siamo in doubleword); perchè adesso non è più prima della addi, quindi si ritrova decrementata di 8 rispetto al solito.

qual'è l'IPC? → si eseguono 5 istruzioni in totale in 4 cicli di clock → 5/4 → IPC = 1,25
→ più basso del 2 che ci aspettavamo di avere per via del double issue
quindi come facciamo a migliorare la performance del nostro codice in presenza di uno scheduling per dual issue risc-V?

tecniche di scheduling più aggressive:

questo loop ha 3 istruzioni che implementano una scrittura dentro ad un array → $a[i] += k$

noi abbiamo dentro x20 l'indirizzo base dell'array e dentro x21 lo scalare k

→ un registro temporaneo x31 per caricare $a[i]$

→ sommo ad $a[i]$ il contenuto di k e infine rimetto dentro alla locazione di memoria $a[i]$, il risultato aggiornato

→ con addi mi sposto all'elemento successivo(+8)

→ infine una branch espressa in funzione degli indirizzi (dell'elemento corrente e dell'array)

- tecnica loop unrolling

usata spesso per multiple issue processor, ma anche usata per multi core, ogni volta che ho bisogno di rendere esplicito il parallelismo che ho nel mio programma (tipo un loop → ho tanto lavoro da fare in maniera compatta e di solito il lavoro dentro ad una iterazione è indipendente dalle altre iterazioni).

→ srotolamento del loop, per rendere visibile più parallelismo a livello di istruzione

→ ovviamente bisogna prendere qualche accorgimento:

register renaming → rinominare i registri per evitare di creare artificialmente più indipendenze di quelle che ci sono.

il codice di controllo non si replica, perchè è il codice che controlla il loop; viene solo leggermente rivisitato: si controlla lo scalare da togliere a x20, lo moltiplico di 4 perchè io adesso faccio 4 iterazioni. (8×4)

→ il loop adesso mi sposta il puntatore di 32 byte più in là → 4 elementi più in là.

questo funziona perchè localmente le load e store sono aggiornate in modo da puntare agli elementi giusti → seconda offset -8, terza con offset -16, ...

e quindi alla fine faccio un offset di -32 e ricomincio la sequenza...

tutto il resto è stato srotolato:

→ essere srotolato di un fattore 4 consiste nell'avere un nuovo loop che avrà 4 delle

vecchie istanze del corpo del loop, raggruppate all'interno di una singola iterazione del nuovo loop

→ in una sola iterazione del nuovo loop, faccio il lavoro che facevo in 4 iterazioni successive del vecchio.

non posso tenermi sempre il registro x31, perchè sono dipendenze fittizie, non esistono nella realtà devo fare il register renaming;

→ il primo blocco usa x28, il secondo x29, ...

perchè fare tutto questo? adesso ho molte istruzioni in più con cui cercare di migliorare lo scheduling su uno slot doppio:

adesso i nostri dual issue slot sono molto meglio riempiti; ho solo 2 casi in cui non riesco a schedulare 2 istruzioni alla volta; quindi più il fattore di srotolamento cresce e più potrebbe migliorare lo scheduling.

in questo caso ho una IPC di 1,75 → molto più vicino al 2 rispetto al 1,25 di prima!

questo miglioramento però mi è costato più uso della risorsa registro(register renaming) e anche un incremento della dimensione del mio eseguibile(loop meno compatto rispetto a prima)

→ il loop unrolling ha un impatto sulla dimensione del mio codice compilato binario → solito discorso di trade off

dynamic multiple issue

processori superscalari = unici processori in grado di attuare questa tecnica.

la CPU decide a seconda di quello che avviene, se eseguire 0 o più issues in ogni ciclo

→ le issues sono gli slot? o solo le fetch di istruzioni? mi sa la seconda

@maxbubblegum47

→ verifica in tempo reale se ci sono hazard di tipo strutturale o di tipo dato e le manda in esecuzione!

→ non richiede più necessità di avere support nel compilatore per lo scheduling; però in realtà si fa lo stesso perchè il compilatore ci può già dare un ordine delle istruzioni che è ottimale per evidenziare il parallelismo.

→ dove il compilatore non riesce fare e mette le nop; l'hw che controlla le istruzioni a tempo reale può riuscire a fare un po'meglio → ulteriori miglioramenti.

@maxbubblegum47 non ho capito perchè non ne necessita, e chi ne necessita allora????

come funziona lo scheduling della pipeline dinamico? si basa sul consentire le esecuzioni delle istruzioni **out of order**(CPU out of order diverse dalle CPU in order a cui siamo abituati).

→ **out of order** significa che devo modificare la logica del cuore della mia pipeline, perchè consenta di bufferizzare le informazioni che sto mettendo dentro alla mia pipeline e io le posso eseguire nell'ordine che voglio

→ poi però PRIMA di fare WB, io devo riordinare le istruzioni in maniera che rispettino il comportamento del programma originale.

la fase di commit(quella dove scrivo lo stato permanente modificato da una certa istruzione) deve fare in modo che i risultati siano pubblicati in ordine, però dentro ho libertà di eseguire le istruzioni in out of order → in un ordine diverso da quello precisato dal mio programma.

esempio:

c'è una dipendenza tra la add e la load → potrei eseguire la sub, mentre la add attende per la load(perchè le load anche se fanno forwarding hanno bisogno di 2 cicli per passare il dato???? penso sia nel senso che devo arrivare fino alla fase di MEM, quindi un ciclo di nop e il ciclo dopo posso fare forwarding @maxbubblegum47):

uno scheduling out of order permetterebbe alla add di entrare nella pipeline, poi ad un certo punto farebbe stallare la add in attesa del risultato su x31 e nel frattempo la CPU consente alla sub di andare avanti e così via, finchè non si rileva che c'è una dipendenza → se anche la sub dipendesse da x31, allora ad un certo punto stallerebbe anche lei e quindi si proverebbe a fare andare avanti la addi, ecc...

→ questo è un meccanismo dinamico che cerca di riempire i buchi, mandando in esecuzione quello che ha e che capisce in tempo reale, che non ha dipendenze dal resto

una CPU schedulata in maniera dinamica:

fase di IF e ID che assomiglia a quella normale in order, ovviamente è fondamentale averle in order?????? NON SI SENTEEEEEEEEE DIOOOOOOOOOO

@maxbubblegum47

→ dopodiché replichiamo tante volte delle **reservation station** e poi delle **functional units**, distinte in tipologie, si sfruttano pezzi di hw diversi per eseguire logica aritmetica di tipo intero o quella floating o le load store...

→ perchè si hanno delle parti di datapath distinte; delle vere e proprie pipeline distinte, nelle quali posso eseguire istruzioni in parallelo!

→ io appena scopro che non ho dipendenze(reservation station), mando subito in esecuzione

infine a valle c'è la **commit unit** che in qualche maniera comunica con le reservation station; la reservation station mantiene lo stato degli operandi che sono stallati dal risultato di una operazione precedente.

il loop che torna indietro è esattamente come viene comunicata la reservation station → viene comunicato che una operazione che stava attendendo è stata liberata(è stata risolta la dipendenza) e quindi la reservation station può mandare in esecuzione un nuova istruzione.

DOMANDA PROF: le reservation unit sono dei regisrti? e tengono dentro il valore o l'indirizzo contenente il valore, in attesa che si sblocchi la dipendenza?

perchè dopo nel register renaming si vede che gli operandi vengono copiati dentro alla reservation unit, dunque si possono sovrascrivere nel register file

tiene dentro sia ID sia valori → perchè devo capire se ho dipendenza

** quindi in pratica le reservation unit sono dei registri che tengono in memoria i dati delle istruzioni che hanno delle dipendenze, appena si risolve una dipendenza, mandano quei dati alla functional unit @maxbubblegum47

poi io in pratica ho tante pipeline perchè ho tanti percorsi che sono formati da reservation unit e functional unit(che sono anche uguali a volte per miglior parallelismo) @maxbubblegum47 queste sono mie interpretazioni

l'aver disaccoppiato la fase di IF e ID, dalla fase di commit; ci permette di rendere più fluida la maniera in cui le operazioni eseguono nel cuore operativo della nostra CPU.

→ questo complica la logica che devo progettare per schedulare dinamicamente queste istruzioni.

→ anche questa è multiple issue ovviamente → posso ritirare più istruzioni in un singolo ciclo → potrebbero essere 0 se tutto è stallato, potrebbe essere 1 oppure tante quanti sono i blocchi hw(functional units) che sono fisicamente in grado di eseguire un'istruzione.

anche qui va implementato il **register renaming** → se no sarei stallato molto più di frequente sull'uso dei registri che io vedo scritto nelle mie istruzioni; ma dove si capisce che non esiste davvero una dipendenza o è stata creata artificialmente dall'uso di un nome di registro comune?

→ allora la reservation station e il reorder buffer possono fornire questa funzionalità di **register renaming:**

quando una istruzione viene fetchata, decodificata e munita alla reservation station:

- se l'operando è disponibile nel register file o dentro al reorder buffer:

viene copiato nella reservation station, quindi non è più necessario nel registro e può essere sovrascritto

- se l'operando non è disponibile:

si attende che una delle unità funzionali lo renda disponibile alla reservation unit; non è neanche necessario(o può non esserlo) fornire l'update del registro, come nei casi di forwarding che abbiamo già visto, si bypassa la scrittura e si fornisce direttamente l'operando alla reservation station che lo propagherà alla unità funzionale che deve eseguire l'istruzione.

speculazione:

per il branch:

posso fare una predizione del target address e continuare a mandare in esecuzione le istruzioni; posso farlo perchè posso ritardare il momento in cui farò il commit → assicurarmi che prima di fare il commit, quindi pubblicare in maniera visibile lo stato

modificato da questa istruzione, devo verificare che la mia speculation non fosse sbagliata.

in questo caso per il branch abbiamo del hw, le **commit unit**, che ci permettono di attendere l'ok per pubblicare i risultato su un register file o memoria; però l'esecuzione è già avvenuta attenzione! noi aspettiamo e basta per pubblicare.

se la speculazione era giusta allora pubblico e basta il risultatoò

se invece la mia speculation era scorretta, allora il risultato che ho nella commit unit viene scartato e debbo istruire le reservation station del fatto che devo ripetere l'istruzione.

per il load speculation:

io posso evitare il delay associato con le load, e con una cache

miss(@maxbubblegum47 che delay????) → con le load immagino sia la nop obbligatoria da fare, ma con la cache miss non ho idea, forse intende che quando sbagli la speculazione su una load, vuol dire che io ho dato un indirizzo sbagliato, ergo sono andato in memoria a cercare quell'indirizzo, ma non è quello giusto alla fine, quindi cache miss??

anche se cache miss vuol dire altro, quindi boh @maxbubblegum47

DOMANDA PROF: nella dynamic multiple issue, la speculazione delle load dice che può evitare il delay dato dalle load, che credo sia la nop obbligatoria da mettere dopo; e il delay dato dalle cache miss, non ho capito qual'è il delay della cache miss

una load è un'operazione non predicibile i ormini di latenza

è un modo di portarsi avanti mentre queste cose succedono

- facendo la predizione del effettivo indirizzo.
- facendo la predizione del valore caricato (la tecnica forse è quella della cache dove tiene salvati gli indirizzi di memoria visitati @maxbubblegum47 è una mia ipotesi);
- così posso fare avanzare la load, anche prima che le load effettive siano completate (what??? @maxbubblegum47) → nel senso che io faccio andare avanti l'istruzione, poi se sbaglia ciccia e torno indietro? @maxbubblegum47
- posso anche bypassare i valori che sto mandando in store e li posso mandare direttamente su una load unit
 - se c'è una dipendenza tra load e store sono in grado di bypassare questo tipo di dipendenza; ricordandosi sempre che il completamento di una store deve raggiungere la memoria, ma se quel risultato serve in ingresso ad una load successiva, allora posso giocare con le reservation stations, che mantengono il valore degli operandi, per disaccoppiare l'esito delle due istruzioni (oh si tutto assolutamente chiaro @maxbubblegum47) → immagino sia una specie di forwarding, nel senso che se ho una load di una store precedente, allora dopo aver fatto la store, io avrò il valore ancora

dentro alle reservation stations perchè non l'ho ancora sovrascritto(credo? non so come funzionano esattamente); quindi a sto punto invece di spendere tempo per andare a cercare in memoria il valore richiesto dalla load, lo prendo direttamente dalla reservation stations → mia interpretazione @maxbubblegum47

DOMANDA PROF: il bypass della dipendenza tra load e store significa che il valore/indirizzo del valore della store è ancora dentro alla reservation unit e quindi con la load invece di cercare in memoria, prendo il valore direttamente da lì?

non posso fare il commit della load, fin quando non sono in grado di stabilire l'esito della speculation.

- guardare e scrivere appunti sulla terza parte della lezione su instruction-level...

riprresa prof:

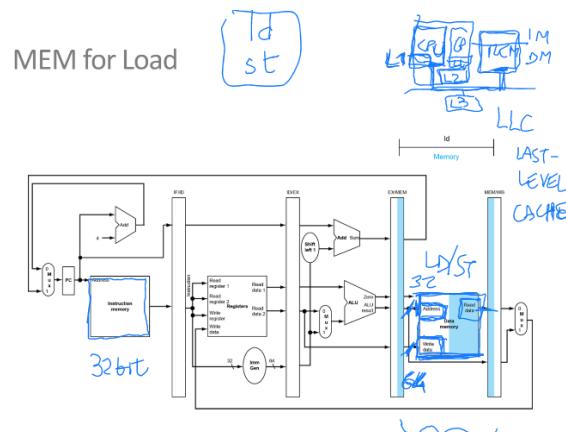
- ho perso la prima parte

la memoria cache di livello più basso si chiama LLC → last level cache

load e store sono le uniche che usano il pezzo memory del datapath → load store unit → unità con 1 port ingresso caratterizzata da un indirizzo e 1 dato in ingresso se vogliamo scrivere; in uscita 1 dato?

spazio di indirizzamento 32bit, ma dati 64bit

→ una cache lavora esattamente con questa interfaccia → cache = interfaccia hw con cui la CPU comunica con la memoria; alla CPU non frega come è fatta la memoria, lei deve lavorare tramite la load/store unit!



indice

Memoria - cache e gerarchia

normalmente ho una DRAM molto grande; la cache di primo livello è molto più piccola(32kb).

come faccio a fare in modo che questi 32kb rappresentino la DRAM?

→ località parziale e temporanea:

località temporale → esiste una buona probabilità che se un programma usa un dato, lo riutilizzerà in futuro → dato un certo indirizzo c'è una buona probabilità che si ripeta nel tempo

località spaziale → se faccio accesso ad un indirizzo, è probabile che io faccia accesso nel futuro ai suoi elementi vicini

→ combinando i due io ho dei segmenti che mostrano il riutilizzo locale e spaziale?????

→ posso progettare????

come si sfrutta?

la prima volta che accedo a DRAM copio gli indirizzi recentemente accessi e i loro indirizzi vicini e li metto su cache, subito pago molto, ma dopo il tempo di accesso sarà minore.

ovviamente avviene la stessa cosa da disco su DRAM → gerarchia memoria...

noi possiamo riscrivere il codice perchè lavori con una footprint della memoria che stai dentro alla cache:



ottimizzare il codice in modo che sfrutti la località e quindi non acceda ad indirizzi random, ma cerchi di accedere ad indirizzi vicini o di accedere spesso agli stessi indirizzi. @maxbubblegum47 mia interpretazione; concordo assolutamente, è l'unico modo che sinceramente mi verrebbe in mente per aiutare la cache ed evitare che vada in miss conflict spesso facendo poi tante replace.

hit e miss:

quanto carichiamo un programma in memoria, i dati vanno nella data memory(dm) e le istruzioni nella instruction memory(im).

alla prima richiesta di dati o istruzioni, io devo attraversare la gerarchia per portare i dati dalla DRAM al processore.

miss = richiesta di lettura o scrittura che non si trova in questo livello di gerarchia.

la prima volta che si prende un dato si fanno tante miss finchè non si raggiunge la DRAM → perchè subito la cache è vuota e pian piano si riempie.

hit = accessi ad un livello di gerarchia in cui trovo il mio dato.

una miss comporta la gestione della miss → se non trovo il dato in questa gerarchia, allora sto il processore, cerco il dato nelle gerarchie inferiori e lo porta in su(verso la cache più alta)

esempio: se ho una load che carica dall'indirizzo 0x4008.

guardo se 4008 è presente nella level 1

cache:

se è presente allora hit e continuo
se è una miss → cerco nelle altre cache,
fino alla DRAM e appena lo trovo, lo
porto alla prima cache

costi tra le tecnologie di memory: più è
verso il basso meno costa; più è alto più
costa ed è migliore
→ la cache del processore è velocissima
e costa tanto, quindi non puoi farci una
DRAM intera

perchè la DRAM è lenta?

la SRAM usa la stessa logica dei flip flop → è molto veloce

come funziona la DRAM?

i dati sono storati come cariche dentro a condensatori; ogni transistor preserva 1 bit di memoria.

la DRAM va rinfrescata: per mantenere il dato, esso va periodicamente letto e riscritto;
questo viene effettuato sulle "righe" di DRAM → ha dei costi

DRAM funziona a banchi di DRAM, ognuno di essi sono delle matrici → una DRAM ha un costo per aprirla:

funziona come righe → il costo per aprire una riga e quello per accedere ad una singola colonna che è molto minore di quello per aprire la riga.

normalmente si cerca di sfruttare la modalità di accesso a **burst**: cerco di leggere quante più colonne possibili perchè costa meno rispetto a leggere la riga.

→ quindi si progetta tutto il sottosistema perchè lavori in burst → pago la latenza iniziale(aprire la riga) e poi mi porto su??? mi porto su tutta la riga così posso accedere alle colonne più facilmente? mia interpretazione @maxbubblegum47

in burst mode quando voi chiedete una word, lei se ne porta dentro di più, tipicamente una riga intera

la DRAM è una matrice di righe e colonne; aprire una riga, dopo che è aperta posso indirizzare le singole colonne → le singole word

→ si cerca di fare in modo che ogni volta che chiedo un indirizzo di una riga; mi porto su più dati di quella riga così se sono fortunato non dovrò riaprire quella riga.(principio di località)

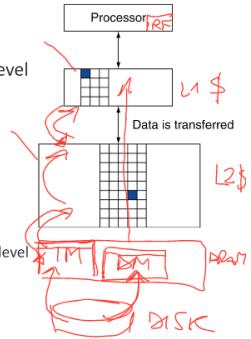
si usano i **row buffer**: pezzo di memoria non DRAM che può tenere dentro una riga → ci vengono caricate sopra le righe, così uso quelli per entrare nelle colonne.

→ sono come delle piccole cache, se sono fortunato il programma chiederà dati che ho

Memory Hierarchy Levels

- Block (aka line): unit of copying
 - May be multiple words
- If accessed data is present in upper level
 - Hit: access satisfied by upper level
 - Hit ratio: hits/Accesses
 - Then accessed data supplied from upper level
- If accessed data is absent
 - Miss: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/Accesses
 - = 1 - hit ratio
 - Then accessed data supplied from upper level

architettura dei calcolatori



portato su e salvato nel row buffer; quindi anzichè ritornare sulla riga della matrice DRAM, uso direttamente il row buffer, non torno ad accedere alla riga.
se invece mi serve un'altra riga allora non è cambiato niente.
io ottimizzo per il caso tipico! → i programmi hanno una certa regolarità.

normalmente una DRAM opera sui fronti di clock??

double data rate DRAM → la DDR raddoppia la performance operative perchè può operare sia su fronte di uscita sia su fronte di entrata.

quad data rate(QDR) DRAM → anche nei input e output → quasi 4 volte meglio della versione base.

banking = concetto di parallelismo → metto tante risorse in memoria a cui posso accedere → nel caso richiedo dati che stanno nello stesso banco, allora guadagno parallelismo

banda = quantità di memoria che posso passare dalla DRAM ai processori?????????

@maxbubblegum47 secondo me ho scritto qualcosa male

in realtà ci possono essere più cose che hanno bisogno della DRAM: multicore, periferiche I/O → in generale in ogni istante io posso avere più richieste di accesso alla DRAM

→ se la DRAM è multi banchi e ogni risorsa può accederci attraverso porte diverse e ognuno chiede dati dentro a banchi diversi → allora posso farli tutti contemporaneamente → nel caso siano tutti nello stesso banco allora non mi cambia nulla e dovranno aspettare

ogni porta è grande 64bit e sono attaccate ognuna ad un banco.

il row buffer è l'interfaccia col la quale gestiamo il banco???? → meccanismo di caching interno al banco??

→ se io ho il dato sul row buffer allora te lo do subito, se no lo cerchi dentro al banco???? → quindi il row buffer non c'entra con la cache, è proprio un meccanismo della DRAM per velocizzare?

DRAM Performance Factors

- Row buffer
 - Allows several words to be read and refreshed in parallel
- Synchronous DRAM
 - Allows for consecutive accesses in bursts without needing to send each address
 - Improves bandwidth
- DRAM banking
 - Allows simultaneous access to multiple DRAMs
 - Improves bandwidth



il row buffer può essere più profondo di una sola riga, può avere politiche diverse, può essere modificato....

l'importante è che siano più semplici, am le problematiche sono sempre le stesse...
di solito si usa 1 row buffer per banco di dati → ma può essere implementato diversamente

memoria cache:

come facciamo a sapere se il dato che ci interessa è dentro la cache?

2 soluzioni:

- **direct mapped cache**: → cache a mapping diretto

c'è solo 1 posto dentro alla cache dove io posso andare a guardare per ogni indirizzo dello spazio di memoria del mio sistema

la mia cache ha di base 8 blocchi e la DRAM ha 8 sottoinsiemi

→ io posso associare un colore ad ogni indirizzo ed avrò tanti colori quante sono i blocchi della mia cache → questi colori vengono replicati nei vari blocchi che posso avere in DRAM

→ isolo i 3 bit meno significativi e li uso per indirizzare → index

se vado ad isolare i 3 bit terminali sono sempre lo stesso pattern!

vantaggio: da punto di vista di logica è facile , tiro 3 fili e con il MUX dico quale alternativa voglio → le ricerche sono semplici perchè prendo sempre gli ultimi 3 bit e guardo dove cadono

→ però una cache funziona bene quando fa molte hit.

un programma che fa il massimo numero di hit.

la nostra cache ha 8 locazioni → il programma legge queste 8 locazioni sempre → la prima volta fa miss e dopo solo hit

svantaggio: pensiamo ad un programma che fa solo load e store, ma il peggiore possibile:

un programma che legge sempre e solo dati che hanno lo stesso colore → si mappano sulla stessa linea di cache!

→ metto in crisi la mia cache

la cache guarda gli ultimi 3 bit per capire in che blocco posso mettere quel dato → la logica per progettare il ritrovamento del dato mi basta tenere gli ultimi 3 bit e so già dove scrivere nella cache

però se ogni indirizzo ha solo 1 possibile mapping → diventa anche il mio limite.

io do un colore ad ogni linea di cache.

da qui sappiamo che il programma che usa meglio la cache, legge i primi 8 dati e poi cicla

→ la prima volta costa molto → poi costano pochissimo perchè sono già su

se il programma legge in sequenza sono caZZI → perchè faccio molte miss, ma almeno ho riempito tutto

se il programma legge sol quelle verdi → io riempio solo una linea(blocco) della mia cache!

in questa modo noi abbiamo solo un unico modo per mettere il dato in cache, che è leggerlo → se non ci arrivo allora non andrà mai in cache e rischio di sprecare spazio

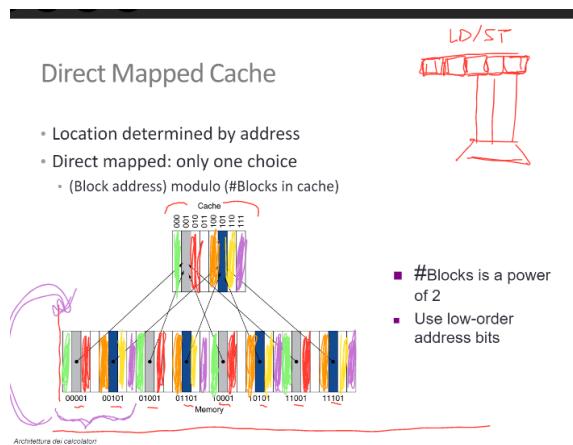
ci sono situazioni nelle quali io uso solo 1 ottavo della memoria del cache → inefficiente

c'è un modo per migliorare? togliere il problema dell'inefficienze del caso brutto:

cache completamente associativa:

8 accessi diversi? → allora ti faccio entrare lo stesso nella cache, ti metto in blocco random o uso certi riferimenti

→ non sapendo a priori dove va l'indirizzo ho bisogno di tanti comparatori → perchè devo confrontare con tutti i blocchi della cache



noi stiamo ragionando sul nostro datapath → che è la nostra CPU; dentro abbiamo sempre parlato di IM e DM; più precisamente abbiamo una fetch unit e load/store unit → ma non sono davvero lì dentro; sono astrazioni che ci dicono che in quello stadio della pipeline noi accediamo alla IC e DM

→ noi sappiamo che quelle 2 memorie logiche sono mappate sulla DRAM.

→ la fetch unit si immagina come una porta che va verso la L1 instruction cache e la load store unit

architettura di Harvard che ci permette di avere parallelismo → nello stesso ciclo porcessiamo il fetch che avviene tramite la L1 instruction cache e quella su L1 data fetch

queste 2 cache contengono le istruzioni e i dati del programma

la data memory è pilotata da un indirizzo e un dato se devo scrivere

la fetch unit è pilotata da un indirizzo

questo perchè il nostro codice si trova in un certo indirizzo in DRAM

→ la nostra CPU ad ogni ciclo produrrà un PC che diamo come indirizzo alla L1 instruction cache

→ la instruction fetch conterrà esattamente quell'indirizzo; però la prima volta che prendo questa istruzione, ne prendo anche altre

→ quando vado al PC+4 non ricordo in DRAM, ma lo troverò in L1 cache, perchè lo avrò preso su, ...

un array si trova nella data ram ad indirizzi che sono spaziati da 2 bit in base al tipo dell'array

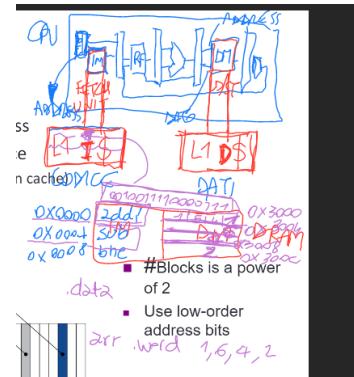
→ questi indirizzi sono quelli che mappo sulle linee della mia cache e che porto alla cache L1 D [per avvicinare alla CPU]

questo perchè la load/stor unit e la fetch unit non sanno niente di com'è la memoria, loro sanno solo che danno un indirizzo e gli restituiscono un dato

la cache ha solo 1 input: l'indirizzo (il dato se scrivo)

→ ho una rete logica che a partire da un indirizzo a 32 bit, sa come gestire lo spazio che ha a disposizione

la nostra logica deve fare 2 cose:
è presente il dato? → logica hit e miss



dove cerco il dato? → direct mapped cache oppure fully associated cache

la DMC, per ogni indirizzo della DRAM; ha solo una locazione sulla cache → vantaggio per la scrittura, ma vincola perchè in maniera poco efficiente rieschio di sprecare spazio nella FAC tutti gli indirizzi di DRAM possono andare dove vogliono nella cache → ho se mi viene dato un indirizzo → nella DMC guardo gli ultimi bit e so dove controllare se c'è; nella FAC devo compararlo con tutti gli indirizzi in cache e se non lo trovo allora vado in DRAM!!

→ utilizza tanti comparatori → costa molto in termini di area

DMC:

indirizzo - modulo - numero di blocchi nella cache

→ se divido l'indirizzo per 8, so già dove metterlo???

→ però anche se trovo dove deve andare, poi nella DRAM abba?????????????????????

3 campi: tag, index, offset

→ interpretazione dei 32 bit di indirizzo → lo divido in questi 3 campi

index = campo che mi dice in quale linea della mia cache il dato va a finire → più linee ho nella cache più sarà grande l'index → $\log(n)$

offset = data quella linea di cache, l'offset di byte a cui si trova il dato → quale byte leggere dentro alla linea

→ una linea di cache ha di solito almeno 4 word!

supponiamo che la mia cache abbia 4 word e ogni word è composta da 4 byte → se io vogliessi indirizzare ognuno di questi io ho bisogno di un offset di

più word ho su cache più ho probabilità di hit → più lunga è la mia linea di cache, più guadagno ho

con una miss (con el miss vado in DRAM) io mi porto dentro più parole contigue → con solo 2 word faccio una hit e miss e hit e miss e

per identificare la precisa word → ho bisogno di un numero di bit che sia sufficiente ad identificare tutti i byte della word nel mio campo offset

→ se ho 4 word in una linea → ho $4 * 4$ byte → 16 → offset avrà 4 bit per rappresentare le 16 cellette

tag = mi dice l'id della replica (del colore) → 2 bit sono il tag di prima perchè ci dice quali sono le varianti dello stesso colore

→ l'index ci dice che è il colore grigio, il tag ci dice quale dei colori grigi è.

in 32bit ho un bit aggiuntivo → **valid bit** → mi dice se il dato che si trova dentro a questa linea è valido o no → hit o miss

(cold cache = le prime call di cache che servono per popolarla all'inizio e quindi saranno miss per forza)

il campo data è il campo dove andiamo a mettere il dato → quello che si trova dentro alla DRAM all'indirizzo x

esempio:

64 blocchi → $2^6 = 64$ → 6 bit index.....

false sharing = condivisione fasulla dei dati → se io ho 2 processori che lavorano sugli stessi dati → quando dovrò leggere e scrivere gli stessi dati ci sono dei problemi

→ hanno dei thread che si sincronizzano → la cache non è in grado di discriminare tra i byte e le word./

dr i 2 processori hanno istruzioni che accedono alla stessa risorsa → succede qualcosa?????????????????????

succede coi sistemi multicore

il processore 1 legge e il processore 2 legge

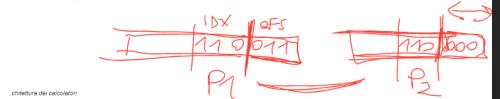
→ il processore 1 scrive, ma il processore 2 ormai ha già letto, non il risultato aggiornato

→ bisogna sincronizzare i 2 thread; devo comunicare al secondo processore che l'ho modificato

il protocollo di coerenza lavora a granularità di 1 linea di cache

Block Size Considerations

- Larger blocks should reduce miss rate
 - Due to spatial locality
- But in a fixed-sized cache
 - Larger blocks ⇒ fewer of them
 - More competition ⇒ increased miss rate
 - Larger blocks ⇒ pollution
- Larger miss penalty
 - Can override benefit of reduced miss rate
 - Early restart and critical-word-first can help



nel caso di una miss devo stallare la pipeline → ci metterò più di 1 ciclo

se è una instruction cache miss → io devo fare ripartire il fetch di istruzione!

se è una data cache miss → io stallo l'esecuzione della pipeline e sblocco l'esecuzione
->??????

per le store si complicano le cose:

c'è un problema di consistenza → se scrivo un dato e lo lascio nella L1 cache tutti gli altri pezzi del sistema possono andare a leggere???

in un sistema ci sono più componenti che possono leggere e scrivere su DRAM

→ durante una store da parte della CPU1, se io scrivo il dato sulal cache e basta: il programma non stalla, molto veloce

→ PERÒ le altre componenti, se vanno poi a fare una load del dato, vanno in DRAM → non vedono il cambiamento

quindi bisogna andare a scrivere in DRAM:

- **wrtie-through** → nel momento in cui faccio la store io propago i risultato nella memoria → funziona e risolve il problema, ma più lenta → IPC basso

write buffer → il processore mette la store su un write buffer e dopo ci pensa il write buffer a portare i risultati in DRAM

Imit: il buffer ha un limite, quando si riempie, non può più prendere dati dopo

→ si riempie facilmente perchè a scrivere in DRAM ci si mette molto di più di scrivere su cache(o sul buffer)

se il programma ha molte istruzioni load e store, il buffer si riempirà di sicuro → non c'è guadagno con questo meccanismo

se invece il programma ha un mix bilanciatro delle istruzioni → questo meccanismo è molto buono e permette di non perdere molto IPC → IPC più alto

in ogni caso se il buffer si riempie → devo stavalre la pipeline e svuotare.

- **write back** → mi interessa la reattività del sistema o l'effetto diretto che ha sulla performance → è una politica lazy

→ io il dato lo tengo sulla L1 cache, k poi se il blocco della cache deve essere cambiato, allora io mando il dato alla DRAM!; se no lo lascio lì. → nel frattempo magari vado a sovrascrivere di nuovo quel dato?????

→ ho un bit dirty che mi dice se devo ancora mandarlo in DRAM

→ la cache a volte deve fare [ulizia?????]

quando si è in multicore , si guarda chi possiede la copia più recente di quel dato.

Acnhe lei può usare il write buffer

cache associative:

fully: ogni indirizzo può andare in ogni blocco di cache....

si utilizzano cache intermedie → non completamente associative, ma nemmeno una direct map

→ nella DMC ho 8 blocchi con 1 sola configurazione per ogni indirizzo

→ nella FAC vede tu....

replacement: se io ho uan cache associativa sorge il problema di decidere quyando ho uyna **eviction**????? dove mettere il dato → che minchia è la eviciton?????

2 politiche:

list-recently used → quello utilizzato meno recentemente → sempre sfruttando la località temporale:

se questo dato non è stato usato da abbastanza cicli → allora probabilmente non lo userò

→ normalmente quando si va su specificità più alta non funziona bene → quindi su un

umero di way alto

random → soprattutto per dimensioni standard tipiche di bho??? si comporta bene; ma ceh micha dice?????????????????

→ per dimensioni di cache con grande associatività è molto simile alla LRU come funzionalità

indice

Memoria - memoria virtuale

noi abbiamo 2 unità fuinzoinali che sono la fetch unit e load/store unit che ci permettono di raggiungere la memoria.

la memoria cetrnrale è raggiunta tramitie gerarchia di cache; "*la cache praticamente funge da interfaccia per gestire la memoria*"

cache in realtà vengono chiamati anche blocchi funzionali che non sono propriamente cache

- il sistema di virutal memory è un meccaniscmo di cache per raggiungere il disco
- il disco è costosissimo → c'è bisogno di tenere in RAM il codice e i dati

più programmi ho più richieste ho da disco per portare i programmi in su

se una architettura ha 32bit; ogni processo ha???

lo spazio virtuale è molto più grande dello spazio fisico che la RAM ha a disposizione

virtual memory termine generale → gli indirizzi che la CPU genera(i famosi indirizzi dentro al PC)

inazione dei 2^{32} = 4 GB??????? "fratm non ho capito"

io però non mappo tutti quei indirizzi su un indirizzo che esiste, perchè la RAM(un tempo) è più piccola di 4GB → ho della memorua virtuale fatta a cache

io gestisco la memoria virtuale a pagine → simile al discorso per la cache; tutti gli indirizzi che cadono in un range di 4kb e sono allineati, allori sono in uan pagina in comune

io copio una pagina da memoria fisica a RAM???? "penso fosse una cosa del tipo che mi porto in cache una pagina di ram"

però quando la DRAM è piena → io ho un page fault → ???? "Praticamente faccio memory swapping, cioè scrivo su disco la roba in ram per liberarmi la ram; è una operazione che rallenta molto il computer ma è necessaria; ecco perché quando mettiamo più ram il computer ci sembra che vad più veloce."

il virtuale è più grande della fisica ed ha

indirizzi contigui (famo 4GB, bravo me piace sto romanesco) e le pagine sono 4kb, man mano che devo utilizzare porzioni del virtuale, posso creare dei mapping dentro alla cache (la fisica)
 → l'indirizzo virtuale che va da 0 a 4kb non passa esattamente l'indirizzo fisico che va da 0 a 4kb, ma mappa un indirizzo a caso

→ tipo una cache completamente associativa!

però man mano che lo uso le pagine si riempiono e svuotano... quindi si creare un mapping non bello → indirizzi prima contigui, adesso sono a random
 → addirittura ci sono pezzi della virtuale mappati sul disco! e non in DRAM
 → quando un programma chiede l'indirizzo di una zona grigia, allora dovrà gestire la eviction in cui tiro via da fisica e mappo quella appena chiesta su fisica togliendo la mappatura da disco; si dice page fault; ovviamente è l'operazione più costosa → perchè sposto da disco a RAM

il processore genera indirizzi virtuali???

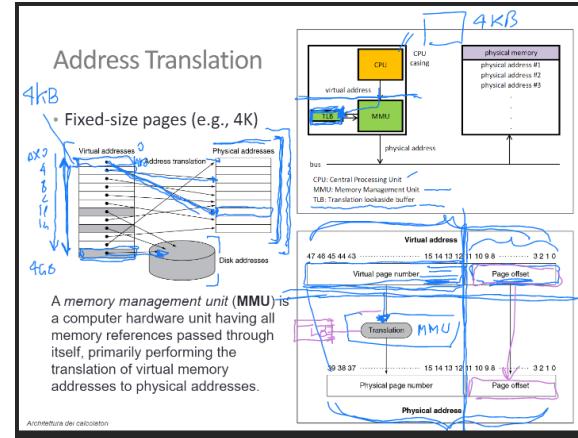
la CPU utilizza dell'hw dedicato (**MMU**) che realizza per noi un livello di indirizzamento
 → la CPU crede di chiedere un indirizzo; la MMU su base dell'indirizzo che gli è stato dato, ne dà uno fisico

TLB = translation lookaside buffer → è un buffer o cache che si tiene vicina le ultime traduzioni → quelle usate più di frequenti; questo perchè la tabella completa sta in DRAM → tengo quella lì per andare più veloce, sempre solito principio di località
 → prima volta c'è un miss su TLB, poi carica 4kb su TLB

la virtual memory gestisce la memoria con granularità a pagina → di solito è grande 4kb

la memoria virtuale può essere utilizzata per dare l'impressione al sistema di avere più memoria a disposizione di quella che fisicamente c'è (in origine); adesso invece c'è più RAM di quanti indirizzi può chiedere la CPU.

così facendo mi proteggo → il processore non crea mai l'indirizzo essatto a cui



accedere, ma una virtualizzazione → non andrò mai ad indirizzare le porzioni di memoria che contrinegono il sistema

quando si parla di indirizzi virtuali si riferisce agli indirizzi che il programma più generale → tutte le load e store passano per MMU.

virtualizzazione memoria = non far vedere ai livelli alti la memoria fisica esatta

come si traducono gli indirizzi?

in alto abbiamo virtual address(viene fuori dalla CPU) → usa un numero di bit più alto rispetto all'indirizzo fisico

→ nel fisco c'è uno spazio per il page offset → è l'offset per indirizzare tutti i byte che cadono dentro una pagina

→ il resto dei bit viene "tradotto", è la traduzione da virtuale e fisica → una volta che faccio la traduzione la tengo dentro alla TLB

→ a sinistra o i bit per trovare la pagina e l'offset trova il byte da prendere dentro la pagina

se il mio programma C gira nello spazio utente del SO, allora tutti gli indirizzi che genera il programma sono indirizzi virtuali; se invece gira su kernel, tutti i puntatori sono fisici.

se ho virtualizzazione innestate ho più traduzioni!

Page tables:

sono delle tabelle che contengono tanti oggetti quanti sono le possibili traduzioni che devo fare

→ se il mio sistema ha 4GB di spazio virtuale → 4GB/numero entry per sapere quanto pesano l'una

la tabella contiene la traduzione tra la pagina virtuale e la pagina fisica

→ non tutte le pagine virtuali hanno un mapping su memoria fisica? anche su disco

la page table è una cache associativa in pratica → non so dove cadono i miei indirizzi virtuali → devo cercarla:

look-up = vado a guardare dove sta e poi mi tengo in memoria(e poi su TLB)

page fault → operazione più costosa → sposto da disco a RAM l'indirizzo!

se la mia pagina è dirty → sto facendo eviction di una delle entry; allora prima scrivo su disco il contenuto e poi porto dentro al pagina nuova → molto costoso

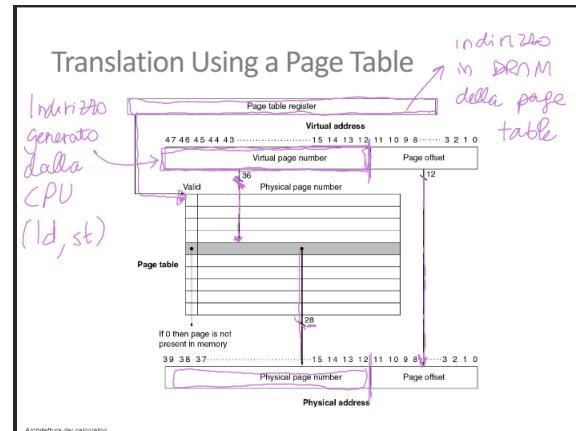
nella cache normale una linea di cache dirty viene scritta in DRAM; nella virtual memory invece se la pagina è dirty → prima trovo la pagina di cui fare eviction, poi vedo se è dirty e se lo è scrivo su disco e poi la sostituisco con la nuova pagina

→ una parte per trovare quale togliere e una parte per scrivere → solo che sta roba pesa molto di più di una linea di cache

la virtual page sta in DRAM dentro ad un array,k per forza li perchè pesa tanto

la MMU trova la page table tramite una????

se la pagina è già in memoria allora la page table registra il mapping → associa l'indirizzo virtuale al fisico???????



36 bit di virtual page number tradotti in 28 bit

valid ci dice se il mapping esiste è valido oppure no

→ se la pagina è su disco allora è a 0; se la pagina è su DRAM allora è a 1

eviction si fa solo se non c'è più spazio nella DRAM.

il mapping ci dice solo dov'è la pagina

la parte bianca equivale alla memoria fisica; la parte grigia al resto della memoria virtuale che quindi va su disco

quando non c'è spazio sulla ram bisogna fare swap tra i la pagina che è richiesta e si trova su disco e la pagina su cui fare eviction

come fa il programma ad accedere ai singoli dati? array ecc..?

il programma accede alla memoria con indirizzi, senza sapere cos'ha dentro(anche su cache)

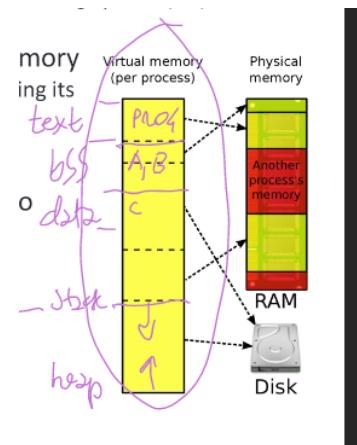
una variabile in programma cos'è?

se è statico allora ???

se è dinamica, quindi malloc → è una chiamata disistema che mi da un puntatore specifico del mio processo??????

stack heap ecc.. I sono tutti indirizzi virtuali; il processore non sa niente di loro, lui chiede solo indirizzi ed è la memoria virtuale che decide cosa fare

→ se quel indirizzo ha mapping in DRAM allora succede quello che è stato spiegato con le cache
→ se invece non c'è in DRAM



LRU algoritmo che decide che pagine va evictata

coem evitare di fare tanto costo?????????

scegliere un design associativo è di per sè una scelta migliore perchè minimizaa le volte che si fanno eviction

inoltre si usa un algoritmo di replcemente → LRU → nella memoria virtuale ci minimizza la probaiblità che un page fault avvenga

come si implementa la LRU?

→ reference bit è un bit ceh signifa se la pagina è stata acecessa di recente

→ c'è un timer nel SO ceh epriodicamente va a mettere 0 al bit

per la gestione della eviciton vera e propria → anche qua la write costa di più e la soluzione write through non è più molto pratica → perchè le pagine pesano 4kb → costa troppo

→ si usa per forza la write back → si usa il dirty bit....

nel momenot della eviction devo fare prima write back dell'informazione fino a disco e poi posso continuare....

TLB:

ogni indirizzo della CPU deve essere tradotto → anche una load/store nella migliore delle ipotesi costa più di un ciclo

→ la MMU per evitare di appesantire ulteriormente questa operazione, utilizza la TLB → su 4kb è molto probabile che un programam acceda ad indirizzi dentro alla stessa pagina...

→ motiv per cui il TLB riesce a ridutte le miss rate a percentuali molto basse.

anche la TLB è completamente associativa e anche lei viene rimpiazzata con LRU.

prima controllo sulla TLB, se non c'è l'indirizzo cercato, allora cerco in page table

la MMU ha hw dedicato per fare queste operazioni

noi abbiamo detto che la CPU genera indirizzi virtuali; e la cache???

dipende dal sistema → può vedere indirizzi virtuali o fisici

normalmente a seconda del livell odella gerarchia, si cambia il valore che legge!

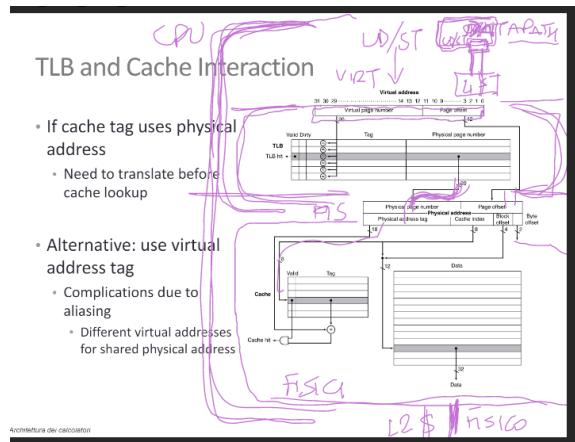
una acceeso in meoria non costa mai solo 1 ciclo perch`1e sono collegati questi 2 blocchi!

se la cache è indirizzata fisicamaente →

prima uso la MMU per tradurre l'indirizzo
e poi questo indirizzo lo do alla cache.

ci sono casi in cui la cache di primo livello
opera con indirizzo a virtuale (per più

velocità e allora la MMU sarà dopo quella cache, ma è molto raro questo



se la CPU ha al proprio interno il primo livello di cache allora ha anche MMU

la page table è una tabella dove ho indirizzo virtuale che corrisponde a indirizzo fisico siccome ce ne sono troppe, non si può portare questa logica vicino alla CPU

→ la page table vive in DRAM

→ quando la TLB non ha il mapping, allora devo fare la page walk → la porto in TLB costa molto

→ se sulla page table io ho il bit valid a 0 → ho page fault → vuol dirti che il dato è su disco → porto su DRAM, costa tantissimo

esercizi:

ci viene dato un design pipeline che è privo di estensioni alla logica di controllo → funziona esattamente come è disegnato → vuole anche dire che ci possono essere problemi (evidenziati in rosso)

sopra si vede che la pipeline è piena e le istruzioni fanno parte del programma scritto sotto.

la prima ad essere eseguita (la add) è in write back e così via...

write back → durante il fronte di salita scrivo e durante il fronte di discesa leggo

→ per questo dice a metà di un ciclo di WB → vuol dire che add sta scrivendo

abbiamo una fotografia di rf e memoria

add fa uso di write back? sì → vogliamo scrivere in x10 la somma di x2 e x1

il write data è il filo che porta la somma tra x1 e x2; chi ci dice se devo scrivere? il segnale write register → lui ci da il registro su cui scrivere.

in assenza di logic di controllo questo segnale quanto vale?

nella fase di decode abbiamo l'istruzione di load al momento → quindi il write register è decodificato per quell'istruzione → quindi vale 13 → x13

quindi quando facciamo WB noi scriviamo su x13 invece di x10 → in x13 ho 105 e su x10 nessuno ha scritto → x10 varrà ancora 10

→ risposta corretta = b

in realtà essendo a metà della WB noi nel write register avremo il valore dell'istruzione prima perchè la decode non ha ancora codificato i vari così???????

esercizio 5

campo offset → identificano il byte all'interno della linea

campo index → identifica la linea di cache (il colore)

campo tag → discrimina quale tra tutti gli indirizzi in DRAM checassono nella stessa linea

blocchi grandi di 8byte → 3bit per rappresentarli tutti → campo offset

cache grande 64byte → $64/8 = 8$ linee → campo index = 3 bit per rappresentarle tutte

infine istruzione = 32bit - 3 bit offset - 3 bit index = $32 - 6 = 26$ bit → campo tag →
risposta A corretta

compulsory miss = miss obbligata → quando la cache è fredda

io devo controllare che il campo index sia uguale → stessa linea

e che il campo tag sia uguale pure lui → stesso colore?

il campo offset non mi interessa perchè non vado dentro a cercare il dato → la cache lavora a granularità di 1 linea

0x4 = 100 → 3 bit del campo offset → non ci interessano

il campo tag = 0000... e il campo index = 000 → la linea di cache è la 0

0x16 = 10100 → campo offset = 100 uguale → ma non ci interessa (0x16 in binario non sarebbe tipo 10110? @pablo remirez) ehm in effetti sì, forse mi sono sbagliato a scrivere @maxbubblegum47 anzi forse non esiste nessun 0x16 → credo sia 0x14 come in immagine e infatti tornano i conti così

o; campo tag = 0000... → ugualae

il campo index = 10 → diverso → la linea di cache è la 2

→ non è una hit → risposta B falsa

C)

0x5 = 101 → cambio solo il campo offset, per il resto è uguale al primo accesso → la linea di cache è sempre la 0, il valid bit è a 1
→ infatti dista di 1 byte dal indirizzo del primo accesso
→ fa una hit → perchè non variano i campi index e tag → perchè dentro alla linea ho più dati, esattamente 8 dati che sono codificati dal campo offset??????



→ se faccio un accesso a qualunque byte dentro alla linea, dopo gli accessi a tutti i byte dentro a quelal linea, sono veloci

0x54 = 01010100

→ offsett = 100

→ index = 010 → la linea è sempre 2 e ho valid = 1

→ tag = 01 → il tag è diverso da quella di 0x14 → ho una conflict miss → si fa replace cosa succede se accedo di nuovo a 0x14?

→ è di nuovo una miss, ma è una conflict perchè abbiamo appena cambiato → compulsory sono solo quelle che si fanno a freddo, questa non è a freddo!

le miss obbligatorie si hanno quando il valid bit è a 0 → nessuno ha mai toccato quella linea.

c'è anche la miss di capacità???? quando ho esaurito spazio dove????

il campo offset non ci fa mai capire se è hit o miss.

quando lavora ocn un dato, si taglia l'indirizzo, portandosi con sè solo gli ultimi x bit, dove x è la dimensione del campo offset.

questo perchè offset ci serve per prendere il dato; mentre il resto ci serve per capire dove prenderlo → quindi capire se ho miss o hit

esercizio 7

pipeline a 5 stadi privo di logica di forwarding, hazard, ...

se non facciamo qualcosa di speciale,ci perderemo per strada informazioni???

nella seconda istruzione (add), x3 VALE ANCORA 5 → perchè la load non ha ancora fatto WB

nelal terza istruzione x2 vale 100 perchè non è ancora stato cambiato e x3 sempre 5 perchè load non ha ancora fatto WB

qaunbtci cicli inpiego? numero istruzioni + latenmza pipielne → 6+4 = 10

si mettono 2 nop tra Id e add → così la fase di decode della add coincide con la fase di WB della load

tra adde sub sempre 2 nop; stesso discorso

quanti cicli impiego? → 6 + 4 + 4 nop → 14 cicli

DOMANDE PROF:

[link](#)

[link](#)

[link](#)

[link](#)

[link](#)

[link](#)

[link](#)

~~prof le 16 ways set associative cache cosa sarebbero? sono il numero di linee che ho associative?~~

~~prof non ho capito se c'è solo una page table nel mio sistema, che ha tutti gli indirizzi virtuali possibili?~~

memoria virtuale ogni processo ha la sua.