lunedì 25 febbraio 2019

Architettura degli Elaboratori

Rappresentazione dell'informazione

- Rappresentazione numeri
- Rappresentazione testo
- Rappresentazione di altri dati costruiti da come rappresento testo e numeri

I numeri

I. Utilizza un **numero finito** di **cifre**

II. Cioè lo spazio allocato per una **variabile** ha un **numero finito di bit**, è possibile decidere anche quanto occupano

III. Il fatto che sia finito lo spazio allocato, genera la nascita di **errori**

IV. Ho n cifre, invece di rappresentare da 0 a 2^n -1 ma voglio i negativi

A. ***Mod1***: Prendo **un bit per rappresentare il segno** quindi posso rappresentare
[-2^(n-1)+1 ,-0 ,+0 2^(n-1)-1]

1. I **due zeri** sono un po' un casino
2. Questo è il metodo del modulo e segno, cioè un bit per segno e il resto è un valore modulo

B. ***Mod2***: Complemento a 1 -> **Complemento a 2**

1. Dato k, se voglio -k prima complemento a 1 tutte le cifre, le nego individualmente [-(2^(n-1), ….. ,+ 2(n-1)-1]
2. Poi sommo 1
3. In pratica il complemento ad **1 flippa** i bit e basta, quello a **2 aggiunge 1**

C. ***Mod3***: Eccesso 128

1. Ce ne freghiamo di interpretare il codice con naturalezza
2. Usata nei numeri in **virgola mobile** insomma
3. Si chiama **eccesso perchè effettuo uno scroll di tutta la sequenza di codici** e la forzo ad iniziare dove voglio

a) Rappresentazioni **polarizzate**

4. Sia comple a 2 che eccesso **hanno un solo 0**, non due

V. Le operazioni risultano esattamente come in base 10, sia somma che sottrazione ecc.

Numeri in virgola mobile

I. La differenza tra numeri in virgola mobile e numeri in virgola fissa sta nel fatto che la distanza tra due codici **consecutivi** è la stessa in tutto **l'intervallo**

   A. Questo aspetto è colui che distingue i numeri in **virgola mobile**

   B. Nei numeri in virgola fissa posso rappresentare anche valori interi ma con un **preciso intervallo** tra un valore e l'altro

   C. Il **floating point** non è la soluzione ma è la nascita di un bel po' di problemi

   D. Con la virgola mobile è possibile variare la dimensione dell'intervallo tra un valore e l'altro

II. Chiaramente **la spaziatura è costante**, stessa cosa per l'arrotondamento

   A. L'errore di arrotondamento è concettualmente come nei numeri naturali

   B. In pratica non è come nei reali che tra un valore e l'altro ho una densità infinita

   C. Ci si accontenta di rappresentare numeri con un margine di errore

III. Come si risolve?

   A. Se devo rappresentare un valore che di norma starebbe in mezzo ad altri due, molto semplicemente io **scelgo quello più vicino**

   B. Se la spaziatura è costante, allora è costante anche l'errore

   C. In virgola **fissa** si intende che la spaziatura è **costante** ovviamente

IV. In virgola mobile invece **si mantiene il rapporto tra l'errore ed il valore** rappresentato **costante**

   A. Nel senso che più è piccino, più aumento la precisione, più è grosso e meno serve precisione

   B. Chiaramente la spaziatura cambia progressivamente, quindi il rapporto tra valore e numero rimane costante

V. Quando uso Virgola fissa e quando Mobile?

A. **Se conosco l'intervallo** da rappresentare allora applico la **virgola fissa**

B. **Se no**n è specificato o chiaro l'intervallo allora si ricorre alla **virgola mobile**

Lo **shift**

I. ***Left***: equivale a **moltiplicare** il numero **per la base** (2 se binario)

II. ***Right***: è la stessa identica cosa ma con la **divisione** (divisione per due)

A. ATTENZIONE A REPLICARE BENE IL SEGNO

B. Se shifto a **destra**, si inserisce uno zero alla **fine**

Principi della virgola mobile

I. Voglio separare quanto il numero è grande dalla precisione del modo in cui lo voglio rappresentare

A. **Mantissa** ed **Esponente** insomma tipo: π = 3,14 = 0,314 x 10^1 = 3,14 x 10^0

B. La **mantissa** si **moltiplica** per una base elevata ad un **esponente**

C. Supponendo di avere 3 valori dedicati alla mantissa e due valori all'esponente, praticamente dividiamo l'asse dei reali in un tot in intervalli che dipendono da quante cifre utilizzo

1. Sia per mantissa che per esponente

2. **Overflow**: trabocco rispetto al numero di bit che ho a disposizione nell'hw

a) Può avvenire anche nella virgola mobile

b) Dato un certo punto, oltre un tot non posso andare

II. Tutto quindi va a dividersi in

A. **Overflow** negativo

B. Numeri negativi espressibili

C. **Underflow** negativo

1. Oltre tot non posso rappresentare valori più piccoli

D. Zero

E. **Underflow** positivo

F. Numeri positivi espressibili

G. **Overflow** positivo

I. I numeri in virgola mobile non formano un continuo a differenza dei numeri reali, pertanto saremo sempre soggetti a errori di approssimazione

A. Chiaro che se stiamo nella matematica discreta dei naturali non si pone il problema

B. Il rounding non è fatto esplicitamente

C. Date quindi n cifre è necessario fare il rounding Siccome n è definito

D. Contiamo poi che nella virgola fissa non esiste poi l'esponente

II. **IEEE** Floating point standard **754**

A. Ci sono 3 **formati**

1. ***Singola precisione***, 32 bit

a) 1 bit di segno

b) 23 di frazione

(1) Contiamo che son bit che o valgono 0 o 1

(2) Normalizzata sarebbe 0,qualcosa si **diminuisce l'esponente e si shifta il numero finchè** la prima cifra a sinistra del punto è diversa da 0

(3) Cioè shifto **FINCHE**' non è 0 la cifra a sinistra e aumento di 1 l'esponente

(4) ***ATTENZIONE PERCHE***' è come la notazione scientifica tipo 3,9 x 10 ^ 5

c) 8 di esponente

(1) Esponente in eccesso fino a 127 da 0

2. ***Doppia precisione***, 64 bit

a) 1 di segno

b) 11 di esponente

c) 52 di frazione

3. Precisione estesa, 80 bit

B. Dati due pc di cui uno è in grado di fare calcoli ad 80 bit, mentre l'altro a 64

1. Data una serie di passaggi che effettuo ad 80 bit, la riconversione che effettuo alla fine NON restituisce qualcosa di diverso
2. Data una percentuale di errori ridotta nell'80 bit, chiaramente portandola dietro cambia la situazione se confrontiamo i due risultati

---

Numeri denormalizzati

I. Numeri che per essere rappresentati in forma normale si va in underflow ma pur di non perdere la possibilità di rappresentare un'approssimazione del tipo 0

A. Con questo sistema si può ottenere un degrado di prestazioni non traumatico come il crash

B. Si shifta la fraction verso destra, si mettono degli 0 a sinistra

1. Praticamente uccido la precisione pur di mantenere qualcosa

C. Dopo una certa se shifto comunque troppo avrò oggettivamente 0

D. In pratica shifto come al solito, butto via praticamente una cifra significativa

E. Si fa per evitare chiaramente la divisione per 0, chiaramente si perde però il concept di fissa

+-	0<exp<max	Qualsiasi valore	Normalized
+-	0	Qualsiasi valore NON 0	Denormalized
+-	0	0	0
+-	1 1 1 … 1	0	Infinity
+-	1 1 1 … 1	Qualsiasi valore NON 0	Not a number

OSS:

Se ho un'operazione tipo A - B, la posso eseguire con A + (-B)

Quindi diventa A + (B negato) + 1

Rappresentazione del testo

I. Il testo è rappresentato tramite tabella ascii (O unicode in tempi recenti)

II. Ad ogni valore di questi 16 bit corrisponde un simbolo in pratica

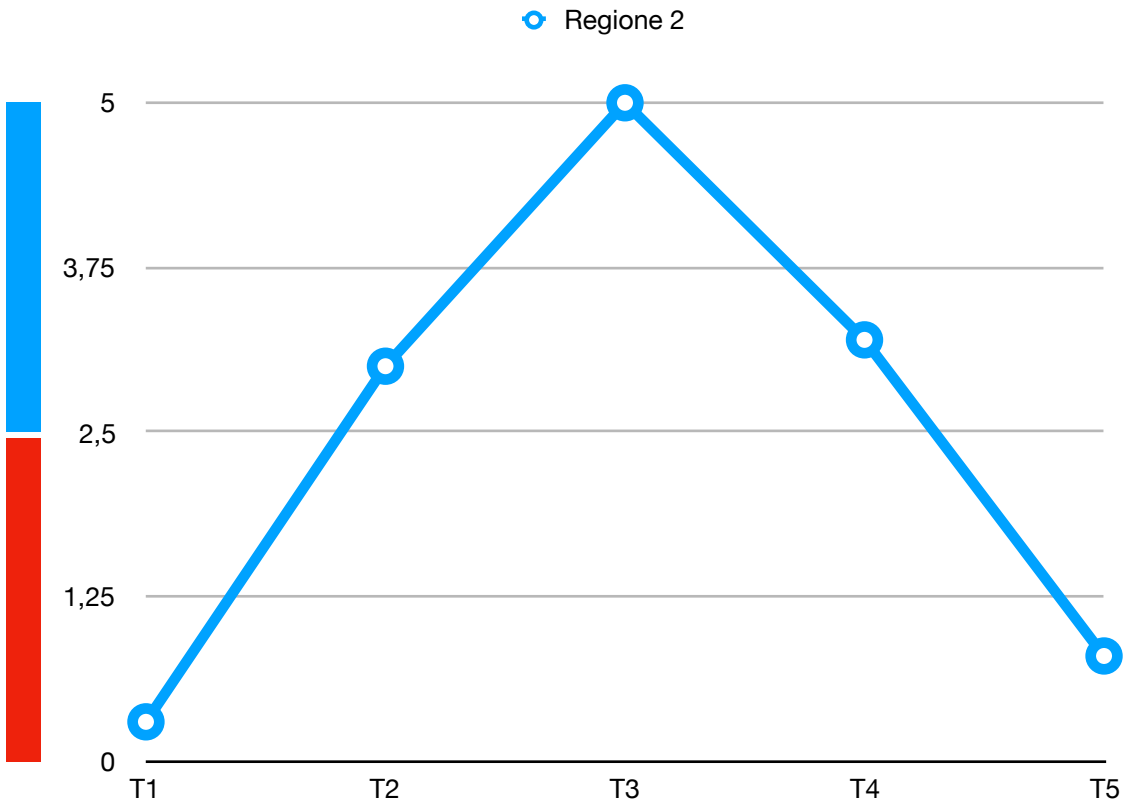
III. Chiaramente l'unicode ha possibilità di rappresentare un largo numero di simboli MA non tutti, poichè tutti è praticamente impossibile siccome sono tendenti ad infinito

Il magico Hardware

Lo 0 e l'1 Logico

I. Dato un grafico su cui Y è la tensione ed X è il tempo

II. Si impostano dei valori soglia tali che se la tensione va lì, allora si assume l'1 o lo 0 logico



I. La sezione che va da 0 a 2,5 V assume 0 logico

   A. Da 3 a 5 invece 1 logico

II. Si costruiscono circuiti in grado di eseguire operazioni su quelle determinate fasce logiche, che danno alla fine quel risultato logico

III. In pratica sappiamo di avere due segnali di ingresso A e B, ed una circuiteria che come output erogherà un valore C

IV. Cioè concettualmente si utilizza la porta and, ma quel che accade è che passa del tempo prima che un valore venga assunto, e quest'ultimo è dovuto al circuito di mezzo

V. Esistono quindi dei ritardi ma noi consideriamo il componente proprio ideale senza ritardo

---

Rete Combinatoria

I. Combinano gli ingressi, chiaramente di quel momento

II. Al momento T, quanto vale l'uscita con determinati ingressi?

Reti Sequenziali

I. Sono delle uscite che dipendono ANCHE da ciò che è successo prima dell'ingresso

II. In pratica ogni cosa dipende dalle precedenti robe

Reti combinatorie

I. Una rete combinatoria è possibile da realizzare semplicemente utilizzando la tabella di verità

II. In una rete combinatoria esistono più livelli, esattamente come per dire in un'equazione insomma

   A. Sullo stesso livello ci sono le operazioni che praticamente sono commutative

   B. Cioè operazioni che tu sai che non importa l'ordine in cui vengono eseguite perchè il risultato non cambia

---

Il Decoder (DeMultiplexer/DeMux)

I. Il decoder riceve n ingressi (Da filo), ed in uscita ha potenza(2, n Fili di ingresso) segnali

II. Attiva una sola uscita con la corrispondente ai bit in ingresso, tipo prendere il codice in ingresso e riconvertirlo in una sola uscita

III. La combinazione degli ingressi attiva una uscita sola

I. Per rappresentare la tavola di verità di un DeMux, si hanno 2^n righe per nIngressi+nUscite come colonne

---

Multiplexer

I. Ha 2^n ingressi su cui ci aspettiamo che arrivi un codice, che identifica di fare qualcosa tra altri ingressi, n
   A. Infine si ha una sola uscita
   B. Successivamente si hanno appunto una serie di ingressi di selezione del Multiplexer, diciamo di controllo
   C. In una sola uscita c'è l'ingresso selezionato dagli ingressi di selezione applicati sugli ingressi

II. In pratica l'uscita non è altro che uno dei suoi ingressi codificato dal valore n

III. Dato un DeMux è possibile costruire un Mux
   A. Prendo i tre ingressi, li metto in un DeMux, che avrà n uscite che metteremo dentro una serie di porte and, in cui dovrà arrivare 1, e poi andranno in una parta or

---

Reti combinatorie: Rom

I. Non sempre usata come rete combinatoria ma anche come semplice memoria

II. E' di sola lettura obv

III. Dati n ingressi per m uscite, si ottiene una tabella di verità del tipo n+m * 2^n

IV. Dato uno zero in questa tabella, si suppone che quello 0 in realtà sia un valore fisico di tensione al di sotto di una certa soglia

V. Se c'è uno 0 vuol dire che va a massa, in uscita per avere uno 0 od un 1 semplicemente io vado a dare tensione o mettere a massa (?)

VI. In pratica la rom è costruita apparentemente come dispositivo di memorizzazione, nei circuiti combinatori

VII. In pratica all'interno ci sono i connettori saldati all'interno in cui gli ingressi sono degli indirizzi

VIII. Dal main concept nascono la
   A. Prom: programmable read
   B. Eprom: erasable programmable

   C. EEprom electrically erasable programmable

IX. Oggi ci sono le memorie flash, che consentono di memorizzare in modo più semplice

X. Dati n ingressi visibili con 32 fili diversi è possibile oggettivamente fare un solo filo in cui ciascun bit vien messo in and o or

XI. ———————/———
32

XII. Dati n ingressi a 32 bit e due ingressi di selezione, in out a 32 bit avremo

---

ALU: Arithmetic logic unit

I. Effettua i calcoli aritmetici. e tutto il resto della cpu lavora per fare funzionare bene l'Alu

II. Mips 32: dove 32 sta per la "parola" del calcolatore
I dati gestiti in pratica son grandi 32 bit

III. La parola di memoria invece vale 8 bit, son due cose diverse,

IV. Strutturalmente è equivalente ad una ALU ad un bit, ma in questo caso semplicemente sono 32 alu ad un bit in cui primo e ultimo sono speciali

---

ALU ad un bit

I. Dati due ingressi A e B ad un bit combinati in and e or le cui uscite vanno in un multiplexer con un solo ingresso di selezione (operation) se va 0, allora in result va 0

   A. Altrimenti va 1

   B. Questa struttura funziona ma se al posto di and e or mettiamo qualche altra combinazione degli ingressi

   C. Il concetto è che si deve ridurre il consumo, se al posto di quelle ci fosse un casino di porte, questa struttura risulta uno spreco per via di questo consumo

---

Sommatore

I. [+] avrà in ingresso due ingressi A e B, spara fuori la somma

A. Più eventuale riporto (riporto in uscita) carry out

B. Siccome sparo fuori un carry out mi devo assicurare di prendere un carry in che si somma a tutto il resto

C. La tabella di verità sarà del tipo
A | B | C | S U M | C O

A	B	C	SUM	CARRY OUT
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

Qualsiasi 1 della colonna delle sum è visibile come un and in cui per ingressi ci sono a b e c e ci sono le negazioni dove c'è lo zero
Tutte quante poi convogliate in una or che poi spara fuori la somma

Quindi ok, il sommatore calcola sempre la somma, se mettiamo l'uscita del sommatore in un multiplexer a 4 ingressi composto da and, or, e sommatore

E' tipo un trenino tra alu che entrano dentro altre alu in cascata
TUTTE sono collegate dal bit operations che è un bus replicato su tutte le singole alu

---

Reti sequenziali

I. Reti combinatorie più circuiteria che memorizza variabili di stato

II. Il clock

A. E' un segnale elettrico, variazione di tensione su un conduttore

B. E' un segnale che si ripete

C. Ogni ciclo ha il suo tempo, in x tempo ci sono n cicli che si misurano in herts

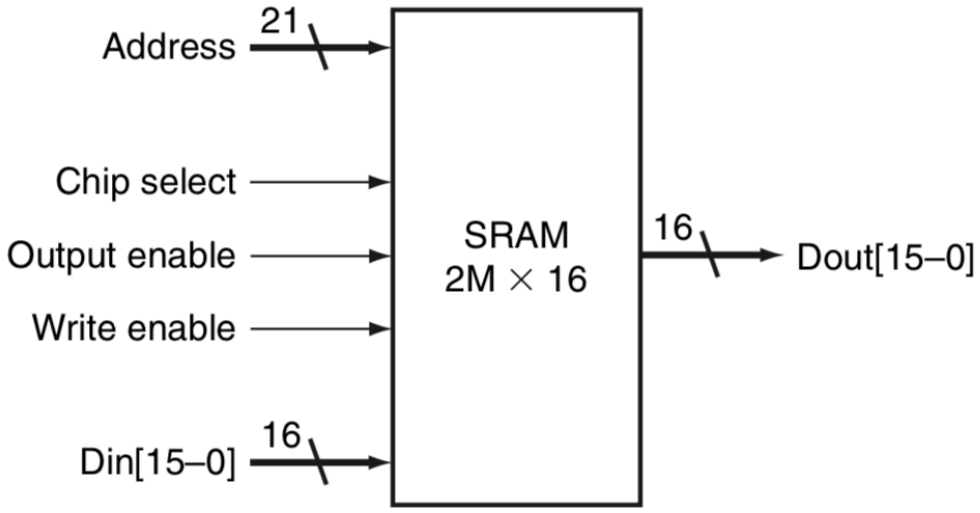
D. Ha due fronti

1. Fronte di salita

2. Fronte di discesa
3. Sono quando c'è la variazione da uno stato all'altro, è comunque un tempo trascurabile se confrontato al clock in sè

Le memorie

I. Statiche SRams (Static Random Access Memory)



A. Random access significa che io posso chiedere al dispositivo il contenuto della memoria in una certa posizione e alla volta dopo in qualsiasi altra posizione, senza vincoli

B. In figura: E' un integrato avente 2 Milioni di righe x 16 colonne

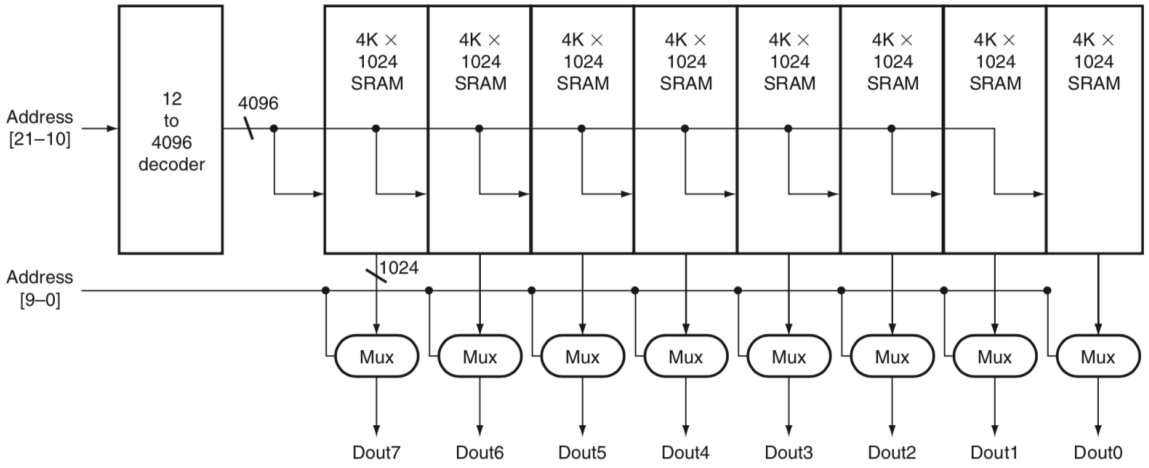
   1. La posizione della parola in questa tabella si chiama indirizzo
   2. i 21 bit di address specificano quale riga della tabella analizzare, stessa roba dei 5 bit del register files, stessa funziona, identificano un elemento tra 2^n elementi
   3. I 16 bit Din sono appunto 16 bit che servono per andare a scrivere un valore in una locazione
   4. I 16 bit Dout invece sono un valore d'uscita
   5. Il chip select serve per capire quale dei banchi di memoria attivare

C. La tabella vera e propria è composta da una serie di bit, identica praticamente ai registri, l'unica cosa che cambia (dimensioni a parte) è che è stretta e lunga

1. Siccome è lunga servono dei multiplexer alla fine, cosa che nel registro era più sempliche perchè con un multiplexer prendeva tutti i bit del register
2. Oltretutto nel register c'era un decoder che aveva 2^n uscite
3. In una Ram è impensabile (a livello di tempi) andare a muxare o demuxare tutte le uscite

D. Esiste un twist che praticamente ha per ogni uscita una porta and che prende un bit select (oltre il dato), che se diventerà l'output di conseguenza
   1. Il select è anche chiamato enable, specifica la posizione in pratica, la decodifica dell'indirizzo

In ingresso:



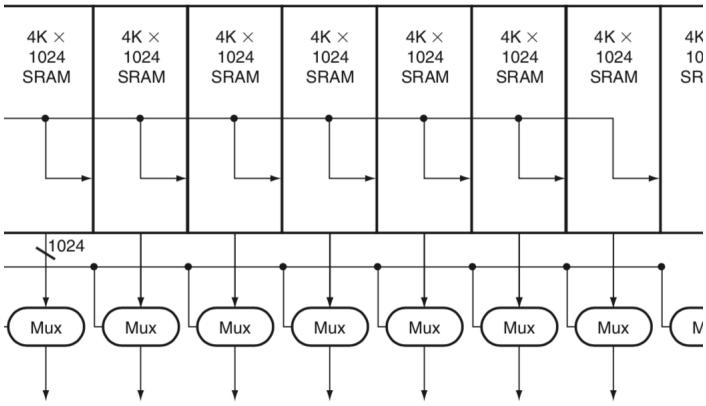
I. Partendo dagli address, si decodificano i più significativi, e ci sono 4096 uscite, alto ma non 2^21, ogni bit spostato dagli address sotto a sopra, raddoppia le uscite del decoder

II. Da lì escono 4096 fili, di cui solo uno è vero e gli altri tutti falsi, successivamente arrivano i banchi di memoria

III. Gli address sotto coincidono con il 1024 (parola) mentre 4096 sono i vari indirizzi di ram

IV. Da ogni banco escono 1024 bit, 2^10

V. Il 1024 entrano in un multiplexer assieme ai vari bit

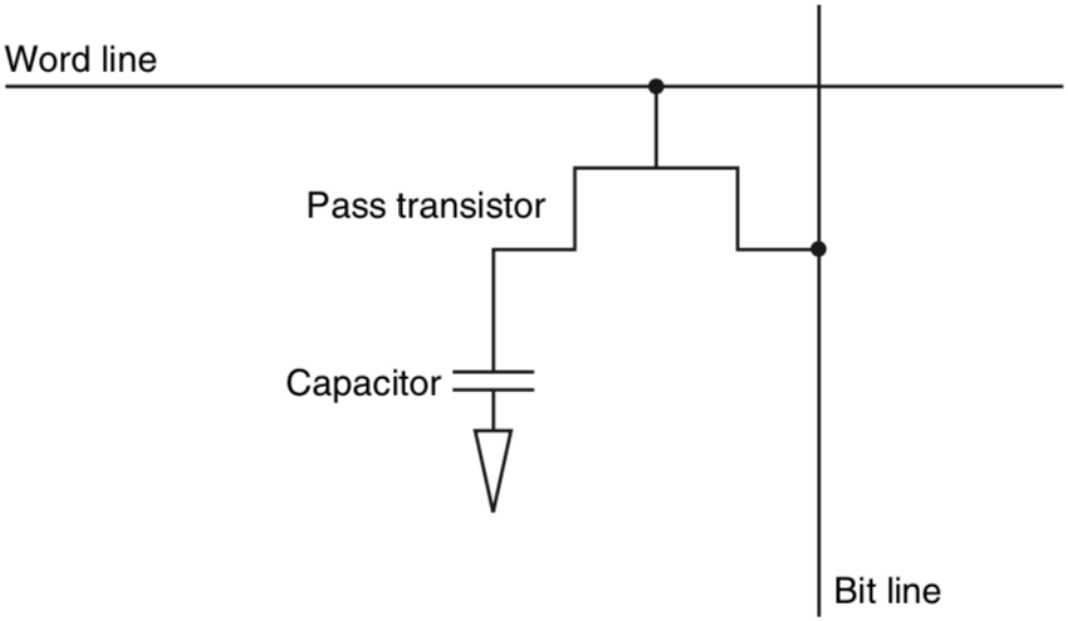


di address

VI. Oggi sram in realtà sta per Syncronous ram, oggi, perchè tutte le uscite sono temporizzate da un clock., la sincrona può essere sia statica che dinamica

---

Ram dinamica



I. A differenza della ram statica in questo caso c'è un condensatore ed un transistor, costa di meno in termini di spazio

II. Il singolo elemento memorizzato è memorizzato come carica residua su un condensatore

   A. C'è da considerare che siccome il condensatore è di quantità infinitesime del farad, tendente allo 0, basta davvero poco per fare sì che sia 0, si scarica in pochissimo tempo

   B. Prima che si scarichi va letta quando è ancora buona, e riletta quando è zero

   C. Questa operazione si chiama refresh e varia a seconda della configurazione della ram

   D. L'accedere alla ram e trovarla occupata perchè sta refreshando è molto probabile

   E. Siccome le cpu hanno ghz di clock non si ha questo problema, il tempo di refresh è irrisorio se confrontato al khz o hz

Macchine a stati finiti

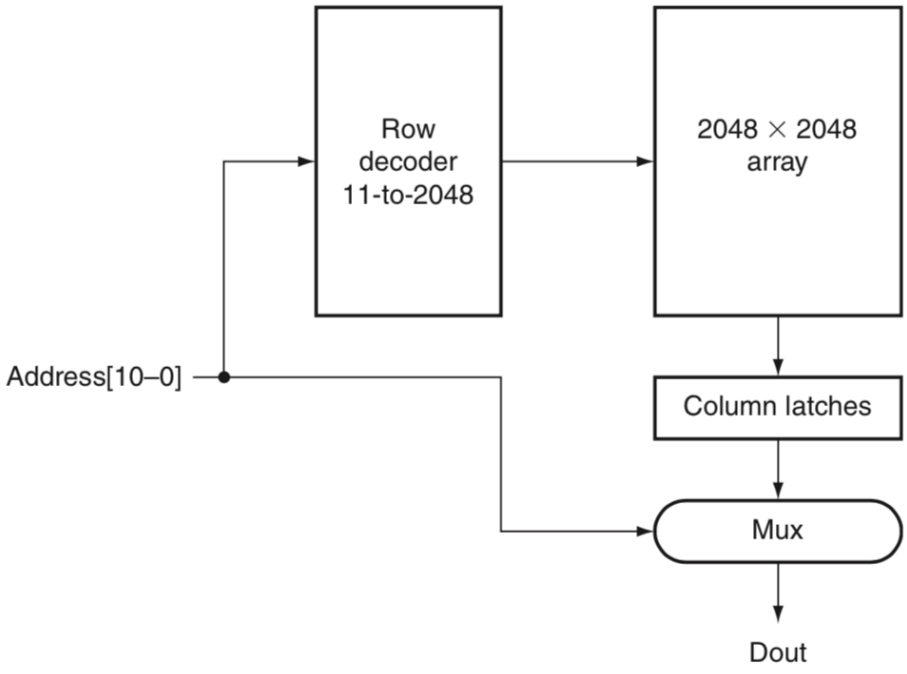
I. C'è una macchina sequenziale con una serie di ingressi

   A. Un registro di stato, il cui numero di stati è obv finito

   B. L'insieme degl iingressi più l'uscita del registro sono due reti combinatorie

   C. Come in fondamenti, ogni cerchio è uno stato avente una uscita che è un arco, rappresentante un ingresso che può entrare in un altro stato, che può cambiare solo se arriva un determinato ingresso

FIGURE C.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.



   D. Se c'è un tempo, posso uscire da uno stato ed entrare in un altro senza che gli altri ingressi siano attivi

      1. Se nessuno degli stati ha un valore attivo, quando scade il tempo (clock) io passo o allo stato successivo, o cambio stato in genere

   E. Le funzioni combinatorie son due, una per le uscite una invece per il passaggio di stato

Istruction set architecture

I. Componenti di un computer (“Big picture”)

II. Tre aspetti

    A. Come fa (ISA)

    B. Come si programma (Assembly language)

    C. Come è fatto (Circuiti e datapath)

III. Esecuzione di una istruzione

    A. Schema di principio semplificato

IV. Formato istruzioni

V. Rappresentazione simbolica

VI. Simulatori

Big picture

I. • Ci sono delle periferiche con cui è possibile dare valori in input, e verrà memorizzato tutto in memoria, successivamente la CPU ha una sua memoria ed ha la ALU che fa le commutazioni

II. Tramite il datapath il procio preleva i dati da elaborare e poi li getta in memoria o comunque sulle periferiche

III. CPU

    A. Central processing unit

    B. Ha un registro che indica qual’è l’esecuzione da eseguire e la ALU oltre i suoi 32 registri

        1. Nel registro dell’istruzione da eseguire può essere contenuta un’istruzione del tipo R3 <- R1+R2

    C. La control unit decodifica l’istruzione ed attiva dei segnali di controllo che consentono di leggere i dati dai registri

    D. La alu esegue l’elaborazione effettiva dei dati e poi salva i dati nel determinato registro che ci interessa

    E. La CU praticamente fa da direttore d’orchestra

IV. Ovviamente questo è a grandi linee, ma è alla fine in tutto e per tutto il modello di Von Neumann

V. La sequenza di istruzione è prettamente logica

VI. Registri CPU:

   A. dentro il processore

   B. Pochi

   C. Veloci (coincide con il clock)

   D. Costosi

VII. Memoria

   A. Vasta, nell'ordine dei giga

   B. Relativamenrte lenta

   C. Non tanto costosa ma costa

   D. E' comunque un'insieme di registri

VIII. Memoria di massa

   A. è periferica io, molto grande

   B. Lenta ovviamente

   C. Poco costosa

   D. Persistente

Le architetture

I. CISC

   A. Istruzioni complesse

   B. Strutture complicate circuitali

   C. Occorrono meno istruzioni, ok, ma son richiesti meno passaggi (esempio intel ecc)

II. RISC

   A. Reduced istruction set computing

   B. Poche semplici istruzioni

   C. Struttura circuitale semplice, esecuzione veloce di singole istruzioni

   D. Occorrono più ustruzioni per qualcosa di semplice (MIPS, ARM, PowerPC)

III. MIPS

A. MARS simula sw del MIPS, e SPIM lo simula non su Java, mentre mars lo fa su java

---

La Risc

I. Semplice, uniforme, facile da capire

II. 32 registri di 32 bit

III. istruzioni da 32 bit

IV. Manipolazione dati solo su registri

V. Trasferimento dati tra memoria e registri

VI. Alterazioni del flusso di controllo

VII. Problema architetturale

A. Comprimere tutti i tipi di operazione in 32 bit

B. Soprattutto mantenendo più omogeneo possibile la struttura istruzioni

VIII. Le istruzioni hanno 3 formati (non coincidenti con il tipo di istruzione)

A. R type (Register type)

B. I type (immediate type)

1. Valore codificato nell'istruzione senza dovere andarlo a prendere dai registri
2. I = i+1; e simili

C. J type (jump type)

1. Fa un salto non condizionato ad un'istruzione scritta precedente

---

Formato RType

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

IX. Perchè 5 bit? Perchè i registri sono 32, k? Quindi appunto con 5 bit posso referenziare i 32 registri

X. Esempio "Somma il contenuto del registro 8 e 9 e mettili nel registro 10

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

I primi 6 bit indicano l'operation code (op code) tipo di operazione, che nel caso

della somma è 000.000 (somma algebrica perchè fa anche la differenza con il complemento a due

XI. I 6 bit finali si chiamano funct, variante della operazione

XII. Infine, nel primo registro da 5 bit scriviamo nel primo da sinistra rs (sorgente), rt (secondo sorgente) e rd (destinazione)

XIII. Gli ultimi 5 bit rimasti si usano per effettuare le operazioni di shift (shamt)

XIV. I registri valgono 32 bit, quindi sarà dentro al registro che si trovano i valori da elaborare

XV. Tutta questa sequenza di bit (32) è l'istruzione in pratica, è possibile dividerla in quartetti, in modo da poterla riprodurre in esadecimale

---

Formato simbolico (Assembly)

I. E' un formato leggermente più comodo (assembly)

II. "Linguaggio sorgente"

III. add $10, $8, $9

   A. Codici simbolici operativi

   B. $registro = registro

IV. OCCHIO ALL'ORDINE DEGLI OPERANDI

V. Usato dai programmatori

VI. Un assemblatore traduce da sorgente ad eseguibile (assembler)

VII. Quello che viene caricato in memoria ed eseguito è SEMPRE l'eseguibile (codice macchina)

VIII. Il simulatore si occupa di questo ma non confondiamo linguaggio lowlevel e linguaggio macchina

---

Modello di Von Neumann

I. Si ha una CPU, una RAM ed una serie di periferiche p1, p2, pn collegate dai bus

   A. Bus Indirizzi

   B. Bus dati

   C. Bus controllo

      1. E' per il comando "leggi* oppure *scrivi*

II. Le dimensioni dipendono dal calcolatore, in alcuni lo stesso insieme di fili svolge in un determinato momento una funzione, mentre in un altro usa un'altra funzione (di solito con i dati succede)

III. Nella realtà non c'è un solo bus ma c'è una gerarchia di bus, la periferica dal culo fa partire poi altri bus tipo schema ad albero che vanno in altre periferiche

IV. Nel nostro caso abbiamo un solo filo per semplificare il concetto

V. LA CPU

   A. Contiene un register file, un IR (istruction register) e obv la ALU

   B. Ha il suo ciclo di clock

Il Ciclo di Clock (Fetch Decode Execute)

I. Dall'istruction pointer, che è un indirizzo, una referenza, contiene l'indirizzo del dato

II. Per ora non sappiamo come cazzo ci è finita lì quella configurazione di bit, per ora prendiamo per buono che c'è

III. In realtà sta roba si chiama PC PROGRAM COUNTER

IV. Cioè alla fine nell'ip (istruction pointer c'è un indirizzo), ha una serie di bit usati come indirizzo

V. In una singola lettura vengono spostati ben 4 byte consecutive SOLO se questi sono allineati alla parola del calcolatore da 4 byte

VI. Passaggio 1

   A. Nel PC ci sono 32 bit, sul bus indirizzi si propaga su tutto il bus elettricamente

   B. Arriva in memoria e si deposita nel campo addr

   C. Il bus controllo deciderà se è read o write, nel nostro caso è read

   D. Dopo un po' di tempo in uscita sul bus dati passano i dati richiesti, e si propagano sul bus ma devono esser letti dalla cpu

   E. Dal bus dati si va poi nell'istruction register

La fase Decode

I. I bit dell'istruction register devono andare nella control unit

II. Successivamente c'è l'execute, esecuzione, infine c'è l'update PC che aggiorna il program counter, tendenzialmente questo sta assieme alla fetch, ma in realtà sta alla fine

III. In RAM ci sono magicamente tutte le istruzioni, non sappiamo come ci son finite, ma son lì, basta questo

IV. Quindi l'update program counter semplicemente fa in modo che il fetch lo si faccia sull'istruzione dopo e non prima

---

L'istruction register

OP CODE 6 BIT	5 bit	5 bit	5 bit	Shift Amount 5 bit	Function 6 bit

l'OP code in pratica dice come interpretare i bit dell'istruzione

V. In base a quel che c'è lì dentro si decide

VI. Tipi di istruzione R

A. Operazioni aritmetico logiche tendenzialmente

B. Gli operandi però stanno non in memoria, ma nei register file

VII. Poi c'è tipo istruzione I

OP CODE	5 bit	5 bit	16 bit

Ci son tanti op code di tipo I, e quel campo da 16 si chiama campo immediato, praticamente è la situazione in cui si vuole inserire un operando leggero direttamente senza sprecare memoria per un dato in più

In generale un valore tra lo 0 e i 15 bit può stare lì dentro, tutto sto puttanaio serve solo per semplificare ed evitare di fare giri inutili con la memoria

A. Alcuni esempi di tipo I

1. La più nota è l'accesso alla memoria in lettura o scrittura

   a) Quest'accesso avviene sia per il fetch, ma anche per quando in un'istruzione è richiesto un dato in memoria

   b) Sono due cose diverse l'accedere per l'istruzione e accedere perchè un'istruzione richiede un dato in una variabile

2. Branch equal

   a) Istruzione di altetazione del flusso condizionata

b) “Salta se i due operandi sono uguali”

3. Loadward e storeward

Il salto di istruzioni

I. Tutte queste istruzioni hanno il salto condizionato

II. Esiste la possibilità di verificare con un if se è possibile effettuare il salto

III. Tendenzialmente per verificare se saltare si può fare una sottrazione che può dare 0 od 1

A. Se salto, ndo vado? Dov’è l’indirizzo a cui io arriverò? Do sta sto Target Address?

B. Nel mips32 è definito in modo relativo al program counter, e si ottiene sommando il program counter più l’immediato (quel valore dei 16 bit)

C. Siccome l’immediato è in complemento a 2 allora posso andare sia indietro che avanti, ma quanto? 16 bit in complemento a 2, quindi da -2^15 fino a 2^15

D. (C’è da fare differenza tra condizionato, incondizionato (goto) poi relativo e assoluto)

Istruzioni di tipo J

6 Bit OP CODE	Addr

26 BIT Usati come target address assoluto, j sta per JUMP

E’ un salto incondizionato ed assoluto, in pratica il goto di C#

La differenza tra assoluto e relativo, nel caso del relativo è: Il target è n - k

Nel caso del jump non mi riferisco a niente, è un valore assoluto, un indirizzo di memoria, non faccio riferimento al program counter

I 26 bit diventano 28, quindi non coincidono con i 32 del program counter, DIO BESTIA NON PENSARE MANCO PER IL CAZZO CHE SIA UN RELATIVO, E’ ASSOLUTO, diceva Mandela

I 4 bit mancanti vengono PRELEVATI dal program counter, praticamente i bit 31, 30, 29, 28

Dal punto di vista logico è un salto assoluto

IF THEN ELSE

I. Questo costrutto consente di alterare il flusso di esecuzione

II. Se a == b allora fai robe altrimenti fanne altre, si dovranno avere a e b nel register file, quindi ci saranno delle operazioni di loadword per portare la parola in registro

III. Ci sarà a in $i e poi ci sarà b in $j

Linguaggio Assembly

Tipi di linguaggio

I. Linguaggio macchina
   A. Linguaggio direttamente comprensibile dal calcolatore
   B. Alto tassi d'errore e sintassi rigida

II. Assembler
   A. Rappresentazione simbolica del linguaggio macchina
   B. Più comprensibile
   C. Tradotto da un assembler
   D. Ottimizza le prestazioni ed ha maggiore efficienza, programmi più compatti
   E. Sfrutto meglio l'hardware che c'è sotto
   F. Usato per controller di processi e macchinari, e apparati limitati piccini

III. Alto livello
   A. Java, C#
   B. Non è che è l'unico che ti consente di usare funzionalità già scritte

La compilazione

I. Scrivo un sorgente, che se è HL va nel compilatore, altrimenti diretto all'assemblatore, a prescindere da tutto arriva il linker che ci attacca le librerie e poi baammm eseguibile bitches

II. L'assembler converte un programma assembly in obj file

   A. Gestisce le etichette

      1. Cosa che NON esiste nel linguaggio macchina
      2. Esiste solo nei sorgenti assembly
      3. Tenere traccia di una richiesta che ha un suo determinato valore, praticamente l'indirizzo 0x343231 si chiama paperino? Ok quando lo trova sa che è quello
      4. Serve per non cagare il cazzo con gli indirizzi e usare nomi che decidiamo noi

   B. Gestisce le pseudo istruzioni

   C. Gestisce numeri in basi diverse

Il Loader

I. Effettua la magia della sequenza di istruzioni del programma in memoria

II. Si occupa di portare le istruzioni da eseguire in memoria ram (le prende dalla memoria di massa)
La lentezza quindi è dovuta prevalentemente alla memoria stessa che ha dei limiti imposti dal tipo di componenti

III. Non lo fa solo per le istruzioni ma anche per i dati

IV. Parte il programma dal main facendo proprio un salto,

Convenzione uso registri

Nome Simbolico	Numero	Uso
$Zero	0	costante 0
$at	1	assembler temporary
$v0 - $v1	2-3	fucntions and expressions evaluation

$a0 $a3	4-7	arguments
$t0 $ t7	8-15	temporaries
$s0 $s7	16-23	saved temporaries
$t8 $t9	24-25	temporaries
$k0 $k1	26-27	reserved for os kernel
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Sp ed fk

Sp serve per indicare nello stack dove sono arrivato, mentre stessa roba per

---

Le magiche procedure

I. Siccome contano, allora nell'istruction set c'è l'istruzione jal (jump and link)

II. Salta a una procedura indicata nell'istruzione e crea un collegamento a dove deve tornare per continuare l'esecuzione del chiamante

III. Salva nel registro 31, ra, return address (indirizzo successivo a quello dell'istruzione jal, cioè indirizzo in cui si trova la jal+4

IV. Questo registro sta nel program counter

V. Istruzione JR jump register, -> non ho fatto in tempo

VI. $a0 - $a3 registri argomento per il passaggio di parametri

VII. $v0 - $v1 registri valore per restituzione risultati

   A. Sono tutti registri come gli altri ma il loro utilizzo per il passaggio di parametri e risultati è una convenzione programmativa per scrivere procedure che possono essere scritte senza bisogno di sapere come è fatto il chiamante ed il chiamato

   B. Un parametro può pure essere un indirizzo oltre che un dato

VIII. Una procedura è un meccanismo per organizzare in modo comprensibile e riutilizzabile il codice

   A. In pratica concedono di programmare con un minimo d'ordine

B. C'è una differenza tra spia e procedura, la spia semplicemente acquisisce risorse, svolge il suo compitino e poi torna con i risultati

C. 6 passi di una procedura

1. Setting parametri in un luogo accessibile alla procedura
2. Trasferire controllo alla procedura
3. Acquisisco risorse per l'esecuzione della procedura
4. Eseguo il compito
5. Metto il risultato in un luogo in cui il chiamante può trovarlo
6. Restituisco il controllo al punto di partenza

D. Perchè è meglio dividere i registri in tot e tot invece che usarne 16 per saved temporary oppure tutti temporary

1. Ipotesi 1: Perchè seguendo uno solo dei casi, avremmo da salvare in memoria maggiormente, succede anche coi saved eh

E. Debugger e profiler

1. Il debugger serve per fare eseguire il programma in modo controllato tipo
2. Aiuta nella analisi e ricerca di errori a runtime
    a) Il profiler compila l'albero di chiamata procedure e la frequenza, e la durata di esecuzione procedure
        (1) Quanto occupa ciascuna parte del programma in pratica
3. Chi chiama chi per intenderci

F. I registri non è che non vengon toccati perchè è hardware l'impedimento, ma perchè è proprio convenzione sw

G. Non cambierebbe niente eh, però ci sono convenzioni, servono, si usano, fine

---

Com'è fatto un file obj? Che formato ha?

I. E' un file, quindi su memoria di massa

A. I campi

1. Campo Header
2. Text Segment

    a) Istruzioni tradotte da assembly a macchina

3. Data Segment
    a) Dati statici
4. Relocation information
5. Symbol table
6. Debugging information
    a) Consentono di capire cosa succede dentro il programma

Secondo compitino

Controllo del percorso dei dati

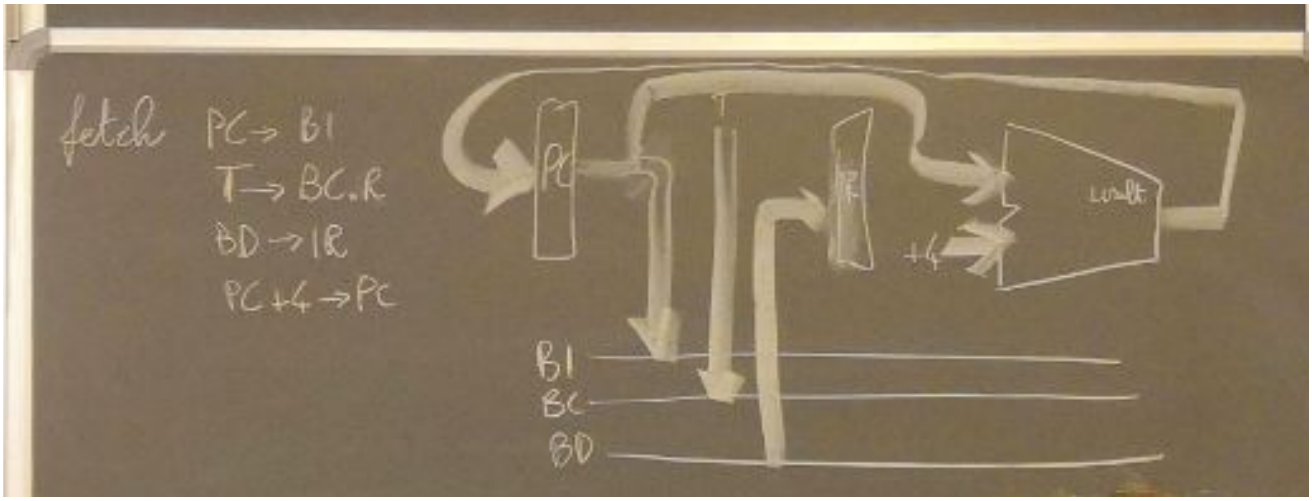
- Si studia come funziona la cpu all'interno, anche se per ora conosciamo tipo solo 3 componenti (perchè non aveva sbatta di spiegare il resto)
- Attualmente conosciamo a grandi linee, l'alu, register file, program counter e istruction register
- Qua s'andrà ad imparare l'estensore del segno. Come opera?
    - Se è uno 0 saranno repliche dalla massa, altrimenti se 1 saranno repliche dell'alimentazione
    - Entrano 16 bit e ne escono 32, poi ci si mette decoder o multiplexer dopo
- Oppure un circuito shiftLeft 2 che prende 26 e dà 28

Percorso dei dati

- Flusso logico che viene seguito per fare in modo che ciascuna istruzione sia eseguita
- Quali sono i passaggi dei dati attraverso i componenti interni? Lo scopo serve per poi effettuare anche un controllo dell'effettivo percorso dei dati
- Ci sono sovrapposizioni non risolvibili con saldature, bisognerà quindi utilizzare dei multiplexer, dei nuovi registri, insomma troviamo un modo di modificare i percorsi

Esempio: La fase di Fetch

Si aggiorna il program counter sul bus degli indirizzi e si assegna true al bus di controllo in read, poi ciò che arriva dal bus dati si mette nell'istruction register e si incrementa il program counter di 4



Il pezzo più lungo è l'attesa della memoria, quindi la propagazione lungo il bus dati, e poi pure il ritorno non scherza

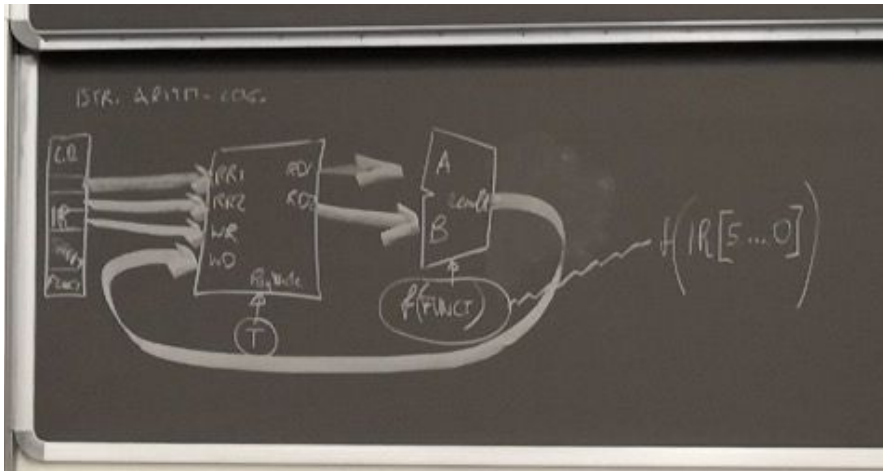
E' possibile notare come il segnale del TRUE venga lanciato proprio dal bus di controllo, ma da dove salta fuori? Il true praticamente fa parte del controllo dell'esecuzione

TUTTE le istruzioni effettuano la fase di fetch, pochi cazzi

Istruzione aritmetico logica

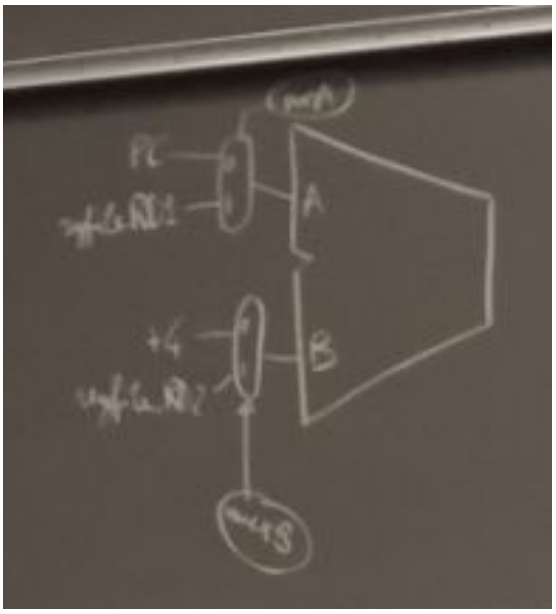
Si parte dall'istruction register, teniamo in mezzo anche il nostro register file con i suoi read data, write ecc

L'istruzione rimane come al solito con op code, istruz, shift, funct, insomma le solite cose



Dal register file i dati letti vengono sparati nel sommatore, che restituisce al register file, nel wd e regwrite va a true, anche se il

sommatore si becca i bit del campo function, che a sua volta becca un istruction register f(IR[5,….0])



E già qui è un bordello anche solo fare comunicare questi due bastardi, quindi con il clock come si svolge su diversi cicli di clock l'evoluzione dell'esecuzione dell'istruzione?

Questa cosa introduce grosse complicazioni, e sul libro manca il clock

Che ruolo gioca il clock quindi? Semplicemente il clock va a suddividersi, appunto ci son più cicli di clock

Se provassimo a dividere quindi questa istruzione in più parti potrebbe essere qualcosa del tipo aggiungere due variabili che prendono dai read data, e poi fare fino alll'alu out

Tutto dipende sicuramente da regWrite, se è true allora ok, ma tanto finchè è false nessuno viene toccato, quando scfriviamo il risultato dentro i registri a e b è false perchè su wd non c'è nulla da scrivere

Il clock c'è sempre eh, cioè è un filo che va contato, anche se non disegnato.

Insomma ci son diversi cicli di clock, ma durante i cicli vengono svolte cose diverse, quindi non sempre voglio memorizzare nel reg file, A MENO CHE sia il risultato appena calcolato, e non posso scrivere in write data in fase di fetch, e la stessa roba anche per a e b, scriverebbero ogni volta

Questa cosa del write va riproposta anche per i registri insomma, introduciamo quindi pc write e ir write

---

PcWrite e IR write

Pc si mette sul program counter mentre iR vien messo sull'istruction register, su a e b non mettiamo nulla, risparmiamo, non servono in questo caso

---

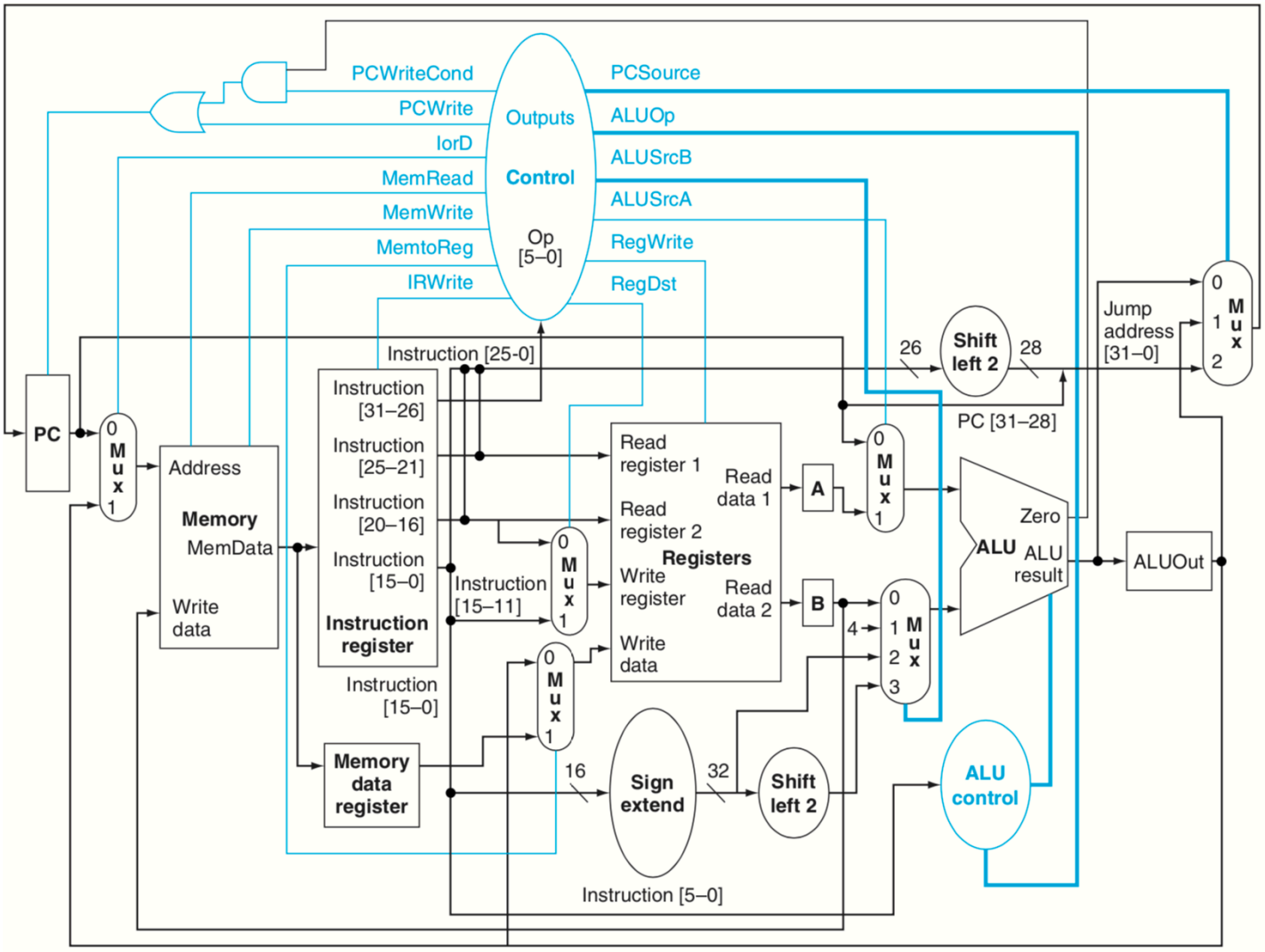
La Control Unit

E' un componente chiaramente hardware che in ingresso riceve i bit dell'istruzione ed in uscita ha tutti i controlli, quindi sì, anche pcwrite, irwrite, tutto quanto quello detto finora, le uscite sono uguali ai controlli presenti nella cpu

La storeword

Si ha come al solito l'instruction register e di fianco il register file

Praticamente ho il campo immediate, ed i due campi quelli da 5 quello più in alto va in read register 1, mentre i secondi vanno in write register



Alla fine la control unit è a tutti gli effetti una rete sequenziale

Le eccezioni

Finora si è parlato di esecuzioni normali senza contare i famosi elementi eccezionali, ci sono una serie di classi di possibili eccezioni

L'eccezione in sè è un elemento imprevisto che ha luogo con l'esecuzione di un'istruzione, praticamente viene sincorno con essa. Esempio: Scrivo una loadword di 4 dopo uno spazio da 3, rimarrà dispari, quindi eccezione

Ogni isctruction set usa il suo modo, nel mips sono tutte eccezioni, per altri sono interruzioni, ma non sono proprio la stessa cosa

Interrupt ed eccezioni son cose diverse. L'interrupt è un evento asincrono esterno che generalmente nasce dagli utenti stessi. Un esempio di interrupt è la syscall, concettualmente il codice della syscal in qtspim è proprio fatto dallo stesso programma

In realtà è la macchina simulata che ha queste syscall automatiche, infine poi ci sono le trap che sono altre cose molto simili, bisogna capire praticamente in base al datasheet di ogni processore ma l'oggetto del discorso è: come vengono gestite?

Gestione delle eccezioni

Esiste un handler o gestore delle eccezioni, ma il discorso è chi lo ha scritto? Perchè l'eccezione è rilevata dall'hardware, gli si deve dire di far partire con una determinata procedura solo quando c'è l'eccezioni.

Come si attiva? Il concetto è che è caricato in memoria in una posizione particolare nota a priori, l'hw sa che è lì caricato, effettua un'alterazione del flusso, popo molla tutto e va lì diretto. Tutto noto stabilito nell'hw, MA attenzione è caricato dal loader di sistema all'accensione

Ok puoi disattivarlo hackerandolo, in modo che alla prima eccezione esplode tutto. Il gestore è software, mentre il fatto di trovare l'eccezione, e saltarci è hardware. Come fa il gestore (sequenza di istruzioni) a capire l'eccezione? Il meccanismo per veicolarla è un registro o semiregistro del processore scritto nel momento in cui si verifica l'eccezione

In pratica c'è il registro cause scritto dall'hardware appena c'è l'eccezione, lo riceve il gestore che tra l'altro ha anche in mano il program counter, oltretutto non c'è anche l'indirizzo di inizio eccezione, quindi ci sono registro cause, e exception program counter. Il gestore potrebbe essere anche stupido e troncare a prescindere l'esecuzione del programma

---

Perchè usare una gestione vettorizzata invece che singola

Va ragionato per clock lenti, lo svantaggio è nel fatto che ogni volta che ho l'identificativo di causa, quando voglio verificare quale handler far partire, devo fare diversa roba, e questo richiede tempo

Con i clock moderni cambia poco, posso permettermi la vettorizzazione, che usa più risorse ma ce ne sono parecchie, il mips usa 8 word per gestire le eccezioni, non è vettorizzato.

A parte che anche se sono solo 8 word, nulla mi vieta di fare una branch in giro per la memoria

Dopo la gestione dell'eccezione il registro cause viene azzerata la causa gestita, nel senso, azzeri quel che hai fatto, non altro

---

Gestione dell'input/output

Riprendiamo il modello di von neuman con in basso il bus di sistema che collega ram cpu e periferiche, e soffermiamoci sulle periferiche

Ogni periferica ha la sua struttura, comunemente che ha un'interfaccia, un registro dati, e un registro di stato, i dati van dal calcolatore al registro dati della periferica, poi attraverso le interfacce beh, i dati son comunicati all'esterno, oppure in una variabile di programma, comunque da lì salta fuori in qualche modo. Quindi le periferiche sono di uscita se dalla variabile di programma i dati vanno verso l'esterno, mentre se da fuori la roba entra dentro beh.. Input insomma.

Questo registro come vien visto? Come viene interfacciato al calcolatore? L'utente schiaccia un tasto, il codice ascii della lettera premuta deve andare in memoria, no? Esistono due modi per farlo.

- Mappatura in memoria delle periferiche
- O periferiche mappate in uno spazio indirizzamento distinto per l'ingresso e l'uscita

Che vuol dire mappata? Praticamente è tipo il fatto di avere una struttura fisica della memoria fatta in un modo o in un altro. Se sul bus di indirizzi gira una certa configurazione di bit, solo quello che è mappato risponde. Alla fine si fa in modo che il sistema di decodifica sia attivato in maniera opportuna quando di questi circuiti ce n'è più di uno. Insomma si fa in modo che si attivi solo il circuito di riferimento.

La memoria è sempre mappata in memoria, nel secondo caso lo spazio di indirizzamento distinto è esterno circa, in realtà non sono mappate in memoria ma in uno spazio di indirizzamento. E' un uso improprio di termini come al solito.

C'è una ulteriore circuiteria di selez ione oltre al registro di stato e dati

Tecnica di gestione dell'ingresso uscita

Il controllo di programma

Nel registro di stato c'è un bit che si chiama Ready che dice che la periferica è pronta, quando uno costruisce la periferica fa in modo che questo succede così.

A livello logico if(Ready) do else do not do. Proprio una cosa semplice, è un branch if equals portandosi con una lw in $t0 per esempio il valore del registro di stato. Se non è ready cicla, e continua a ciclare tantissimo tipo all'infinito finchè non è ready true

Il ciclo di attesa prende il nome di Busy waiting, quindi concettualmento la cpu è occupata in quest'attesa, è definito tipo Polling sta robba qua, cioè continuare a ripetere finchè

Se io per ipotesi vado più veloce della mia periferica, viene contato solo l'ultima delle mie configurazioni, tipo non so faccio una parola, mi tiene solo le ultime lettere

Quando i dati escono, di solito con la load word o comunque qualcosa di simile vien scritto 0 nel registro di stato, se ne occupa la periferica, è tutto predisposto

Banda passante e latenza

La banda passante praticamente sarebbe la quantità di dati per unità di dati tipo gb al secondo ecc

La latenza invece è l'intervallo di tempo che passa da quando la periferica è pronta, a quando prelevo i dati e rendo 0 il ready

Tornando al modello di Von Neumann

Gli interrupt

Nel datapath, in cui si considerano solo l'overflow e il codice operativo non esistente, in realtà ci sono un casino di eccezioni tra cui l'interrupt, l'interruzione

Dove si verifica l'interruzione? Come si fa a capire? Viene verificata all'inizio del percorso, prima del fetch, e a quel punto c'è un gestore, un unico gestore. Nel mips praticamente è solo un solo codice possibile

Teniamo in mente che una cpu era mono thread, quando si trova l'interruzione, il gestore esegue il codice di soluzione e ritorna dove il programma è stato lasciato.

Alla fine della gestione delle interruzioni ripristiniamo i registri che vengono salvati prima della gestione, in pratica è come le eccezioni, tutto uguale.

C’è da contare una robina, praticamente qua è grande la banda passante, la latenza è alta

Come facciamo per fare in modo che non ci siano interruzioni nel gestore eccezioni? Cioè durante una soluzione all’eccezione, non devo interrompere. Quindi c’è un bit di flag e una porta and con l’interrupt request, quindi che cosa succede? Se l’and è vero allora esegue altrimenti semplicemente no.

Nella parte exec dell’alterazione di flusso che porta al gestore. Contiamo di avere più interrupt request (con la and che conferma se è vera, se va bene) e alla fine tutto va dentro una or, se la or è >0 allora ok

---

Registro di stato della cpu

Si trova nella cpu, attenzione che non è lo stesso delle periferiche, ha varie funzioni ma c’è un gruppo di bit chiamati bit di mascheramento delle linee di interruzioni, poi ci sono macchine che hanno un registro di mascheramento, nel mips questo accade all’interno del registro di stato.

Quei bit chi li mette lì dentro? generalmente sono tutti 1 ma magari uno può interrompere un gestore interruzione, beh, fine, interrompe

---

Vettorizzazione

Alternativa della formula a singolo vettore, praticamente il gestore testa i valori presenti nel registro causa, con questa struttura risparmiamo le istruzioni del gestore che servono a capire quale periferica e quale causa, toglie tutti gli if insomma

Base + causa * 4 (se son 4 bit) si ottiene l’indirizzo di inizio gestore causa che è un intero genericamente

---

Memoria cache

All’atto pratico il sistema di memoria è costituito da un largo quantitativo di memoria ad accessi lenti, ed una piccina con tempi veloci. Come si fa per fare apparire quindi una sola memoria veloce definitiva?

Esiste una località che si chiama cache che consente di fare apparire la memoria come “più veloce”. Avendo una memoria grande e lenta, e volendo avere una memoria grande veloce con l’uso di una memoria piccina, come spostiamo le robe nella memoria veloce?

Come l'nvme che fa da supporto all'hdd, Quanto è grande ciò che si sposta dalla memoria lenta alla cache? Quest'insieme di dati si chiama blocco, ed ogni sistema ha il suo concetto di blocco come dimensione

Ogni cache ha i suoi blocchi quindi, vicino alla cpu c'è la cache fisicamente vicino, costerebbe tantissimo allargare le dimensioni, del tipo che un giga fa 500 -> 1000 euro.

Concettualmente gli accessi della cpu vanno direttamente nella cache, non nella memoria lenta. Il blocco è un quantitativo che viene copiato dalla ram alla cache. La cache deve comunque poter dire se un'istruzione richiesta non è presente nel blocco. Viene mandato un hit o un miss, che sarebbero cel'ho o noncel'ho, fine

La frequenza con cui l'hitrate o missrate si verificano è variabile. Ma che accade quando vien ricevuto un no? C'è un hit time, tempo per ottenere il dato richiesto se esso è presente nella cache (sia per verifica che per l'ottenimento) mentre invece il miss penalty è il tempo della sostituzione del blocco + accorgimento che il dato non c'è

Mappatura diretta

In ciascuno dei blocchi ci va a finire uno solo dei blocchi della ram, quindi è necessario dividere la ram in blocchi (esempio 4 byte a blocco che son 32 bit alla fine)

Come si verifica che un indirizzo ci sia in cache? Mi basta guardare quale dei blocchi è presente nella cache, che quindi ha una serie di bit che si chiamano tag che specificano quale tra i blocchi esistenti è quello caricato. Qua si parla di blocchi ora, la cache ha un certo numero di blocchi quindi.

Dati n blocchi della cache ed m nella ram, il tag consente di sapere quale degli m blocchi è scritto nel n-esimo blocco di cache, chiaramente 3 bit di questi aggiuntivi servono per l'indirizzo della cache, mentre invece gli altri 2 son bit di tag. Viene salvato in corrispondenza di ogni blocco a parte (presente nella cache).

Si confronta la parte più significativo dell'indirizzo con il tag, dipende dal tipo di mappatura, ma comunque il tag rimane.

Aumentando la dimensione di un blocco in che modo influenzo la prestazione? Blocco piccolo = tempi più rapidi di copiatura dalla ram alla cache MA più accessi mentre invece al contrario se il blocco è ciccione e gigante allora in quel caso potrebbe essere vantaggioso perchè riesci a ritrovare più istruzioni ma il problema è che poi se ti capita un miss allora i tempi saranno maggiori per reperire il nuovo blocco

Write through e write buffer

write through, finchè non c'è lo stesso dato sia in cache che in ram, mi incazzo e non mi muovo, mi devo adeguare quindi alla ram sperando sia veloce ma almeno ho sempre coerenza dei dati in cache e memoria

Il buffer dice che la scrittura in cache è ok, il write buffer è contemporanea, posso andare avanti e per avere il dato aggiornato devo aspettare i tempi necessari di attesa

Infine c'è il writeback che praticamente la scrittura si fa solo in cache, si rischia rispetto all'incoerenza tipo tantissimo, nel senso, prima porto in cache, modifico, spacco disfo faccio robe, poi rimando modificato di pacco