DISCLAIMER: non avendo alcuna soluzione ufficiale le risposte sono basate sulle slide e sui miei appunti. Potrebbero contenere errori e non coprono tutte le domande degli anni precedenti. Se trovate degli errori siete liberi di correggerli e ricondividere la vostra versione del documento (anche le aggiunte sono bene accette).

Se sapete usare git potete mandare una pull request con le modifiche a :

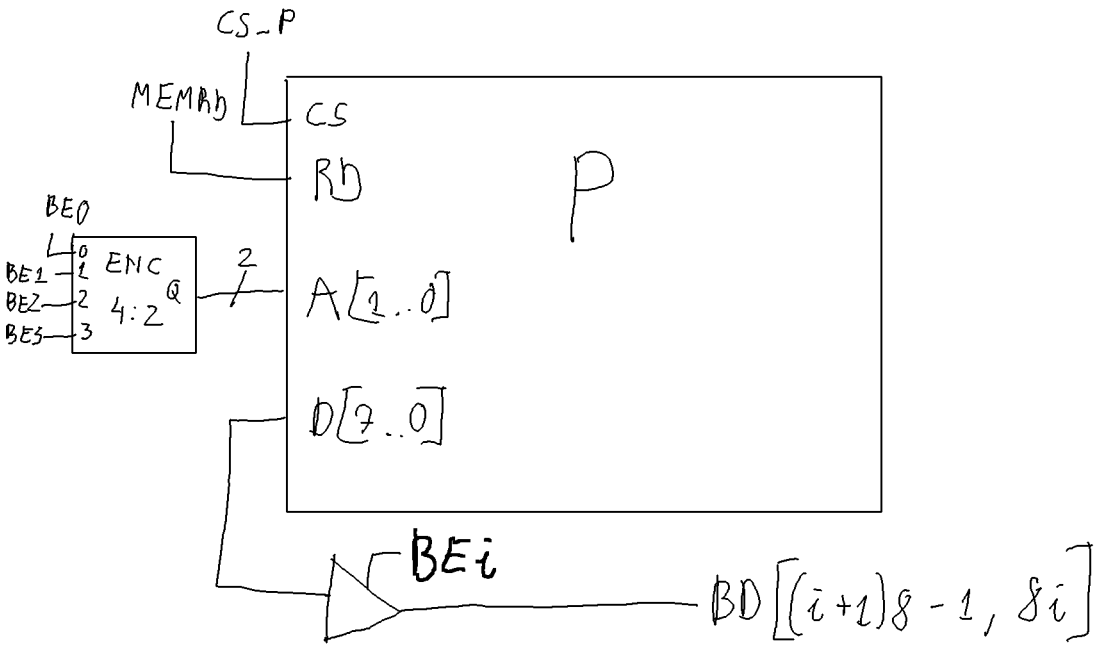
https://github.com/drank40/graal-calcolatori

Good luck!

Q2 2017_01_09)

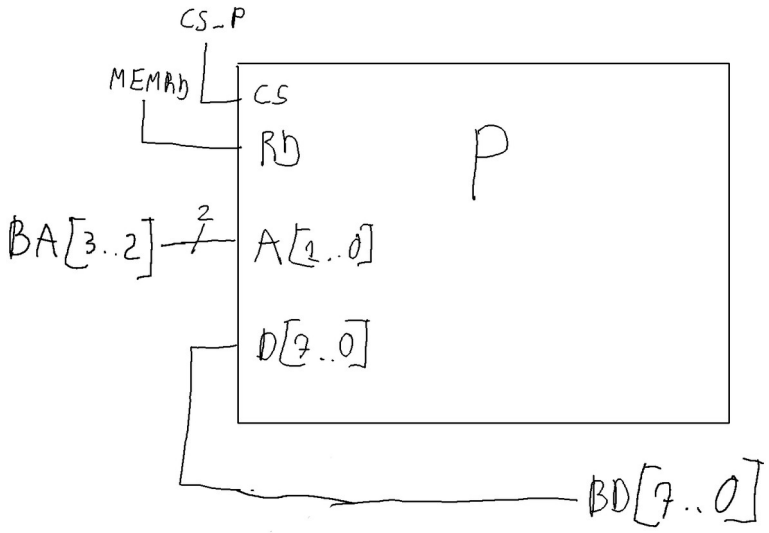
Descrivere, con un esempio, la differenza tra il mapping di periferiche a 8 bit (con 4 registri interni) a indirizzi contigui e non contigui nel caso di un processore con bus dati a 32 bit.

A) In un processore a 32 bit gli indirizzi contigui appartengono a banchi di memoria diversi, e questo vuol dire che per mappare il dispositivo ad indirizzi contigui bisogna usare i BE per generare l'input del decoder di secondo livello.



Supponendo di avere un dispositivo P, in sola lettura, con 4 registri P0-P3 allora il mapping contiguo sarebbe questo, dove in output ho 4 tristate condizionati dai diversi BE (l'utente non deve attivarne più di uno alla volta).

Il mapping non contiguo è più semplice :



Q2 2017_07_10)

Spiegare cosa sono le alee di controllo e quali strategie esistono per poterle gestirle.

A) Le alee di controllo sono causate dal fatto che le istruzioni che seguono un branch(conditional jump) dipendono dal branch stesso. Questo vuol dire che nella pipeline possono entrare delle istruzioni errate, e trattandosi di un sistema pipelined l'unità di controllo non può fare altro che trasformarle in NOP una volta valutata la condizione del branch, causando da 1 a 3 stalli in base alla strategia adottata dal DLX per la retroazione della condizione e dell'indirizzo di salto corretto.

La strategia più conservativa per gestire le alee di controllo è mandare in stallo la pipeline subito dopo aver decodificato un'istruzione di branch, per farla poi ripartire appena si è valutata la condizione di salto. Questo vuol dire che ogni branch causa il numero massimo di stalli, e rende questa strategia molto inefficiente.

Una strategia più efficiente è la Predict Not Taken che, come suggerisce il nome, lascia andare avanti il PC in modo speculativo come se il branch fosse not taken (PC=PC+4).

Quando la speculazione è corretta non ho stalli, altrimenti sì. In ogni caso non rischio nulla perchè nessuna istruzione inserita speculativamente nella pipeline raggiunge lo stadio di MEM o WB se è errata, e quindi non modifica alcun dato.

Una soluzione alternativa è la Delayed Branch, che sfrutta la posizione delle istruzioni in Assembly.
Se prima del Branch ho delle istruzioni che non influenzano il risultato del Branch o la sua esecuzione(un ulteriore Branch) un compilatore consapevole può riordinare queste istruzioni in modo da piazzarle dopo il Branch, dando quindi tempo al branch di essere valutato senza sprecare cicli di clock. Chiaramente queste istruzioni devono essere eseguite in ogni caso, quindi l'unità di controllo non deve dropparle. Nel caso non ci siano istruzioni del genere da riposizionare, il compilatore può inserire delle NOP, ottendo in pratica lo stesso risultato dela prima strategia.

La soluzione più efficiente e moderna è la Branch Target Buffer, che consiste nel conservare i risultati storici di una branch, identificata unicamente con il suo indirizzo. Realizzo quindi con una cache, messa nello stadio di IF, una struttura a tabella che associa agli indirizzi delle branch un PC predetto (essendo una cache dovrò avere anche una politica di cancellazione di dati, magari in modo casuale).

Se ho una HIT nella cache farò un fetch all'indirizzo predetto, altrimenti se ho una MISS speculerò su PC + 4.

Esistono diversi algoritmi per predire il PC in base allo storico delle branch, ma uno molto semplice consiste nel prevedere che un branch taken sarà taken anche la prossima volta che viene eseguito, e viceversa per il not-taken.

Q3 2017_01_09)

Descrivere il principio di funzionamento del branch target buffer. Dove e come agisce? Quali sono i problemi nel caso di loop annidati e quali strategie possono essere adottate in tal caso per aumentare le prestazioni?

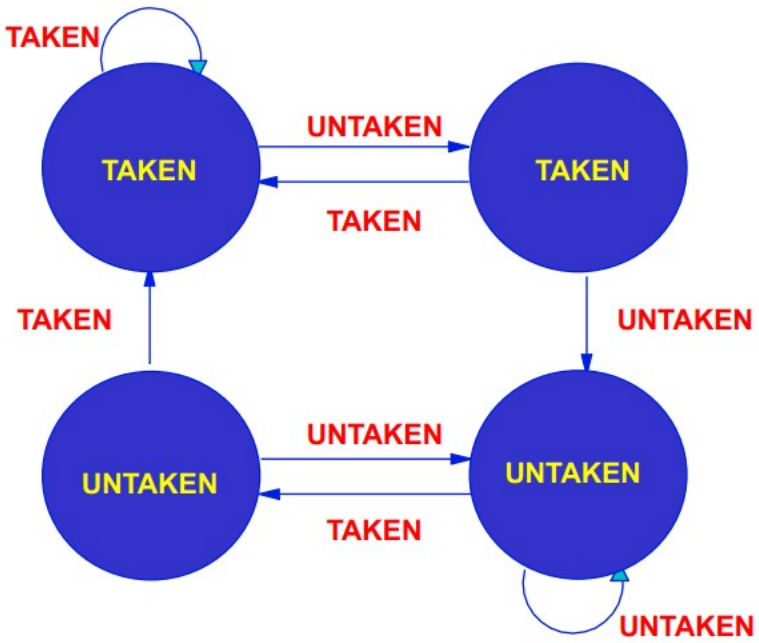
A) Vedi **Q2 2017_07_10)**, parte sul BTB.
Però un loop annidato risulta critico se adotto l'algoritmo descritto precedentemente : infatti ogni loop ha una certa condizione che lo porta a termine, dunque ogni loop esegue una branch prima di cominciare un ciclo e, annidando due loop, ho due branch annidati.

In particolare il loop “inner” ad ogni iterazione del loop “outer” seguirà questa sequenza di taken / not-taken

TAKEN
TAKEN
….
NOT-TAKEN (quando usciamo dal loop)

Se usassi l’algoritmo di prima a questo punto il branch predictor penserbbe che la prossima instanza dell’inner branch sarà not-taken, ma ciò non è verò perchè quando ricomincierà l’outer loop avrò un nuovo ciclo come prima, e il branch all’inizio sarà taken.

Per questo si preferisce alterare la previsione del branch predictior in risposta a due errori (mispredictions), seguendo un diagramma degli stati del tipo :



Q3 2017_02_15)

a) Quali soluzioni software esistono per gestire le alee di controllo?
b) E' necessario impostare l'unità di controllo in modo opportuno?

A) Vedi **Q2 2017_07_10**, parte sul Delayed Branch.

Q3 2021_06_16)

È possibile eseguire una JAL all'interno di codice chiamato da un'altra JAL?
1) Si o no?
2) In caso di risposta affermativa, come? In caso contrario, perché?

A) 1) Sì
2) Posso avere due JAL "innestate", devo solo prestare attenzione al fatto che ogni JAL salva il PC precedente in R31 : se le eseguissi in modo naïf senza salvare il PC restituito dalla prima JAL non potrei riprendere l'esecuzione del codice dopo la prima JAL.
Dovrei quindi scrivere un codice del tipo:

```
JAL first; primo salto
…; eseguo del codice

first :
    ADD R30, R31, R0; copio R31 in R30
    …; eseguo del codice
    JAL second; secondo salto
    …; eseguo del codice
    JR R30;

second :
    … ; eseguo del codice
    JR R31;
```

Q2 2017_02_15)

a) Cosa si intende per mapping di memorie a indirizzi non allineati?
b) Spiegare in dettaglio, con un esempio, come può essere realizzato.

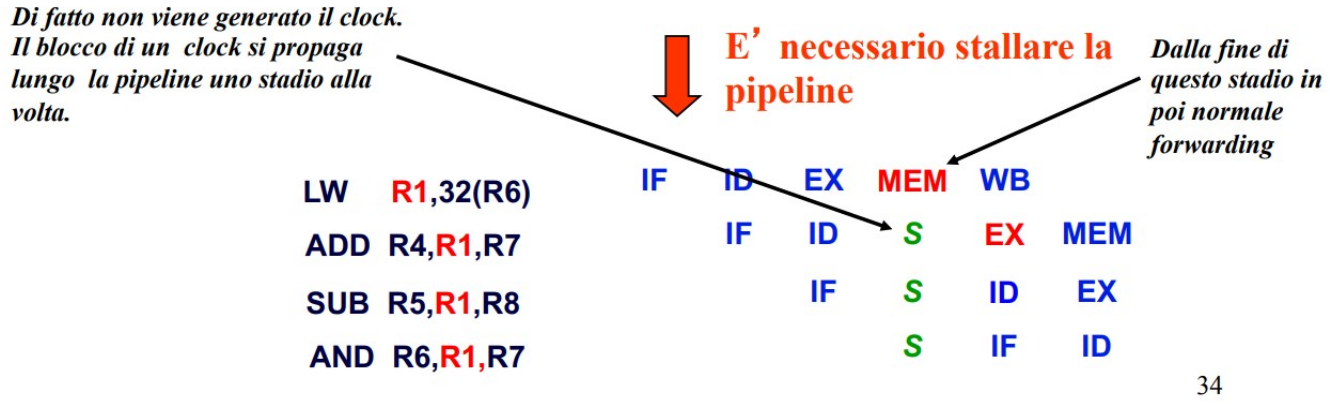
A) a) Dato un dispositivo D con N= 2^k byte indirizzabili, D si dice mappato ad un indirizzo A se gli indirizzi dei byte di D sono contenuti tra A e A + (n-1), e quindi A è il più basso indirizzo associato a D.
D è disallineato se A non è un multiplo di N.

b) Ad esempio se mappassi una memoria da 512 MB a 0x3000 0000-0x5000 0000 avrei un mapping disallineato con CS = $\overline{BA31}$ x ($\overline{BA30}$ x BA29 x BA28 + BA30 x $\overline{BA29}$ x $\overline{BA28}$)

Q2 2018_02_09)

Quali problemi implicano le alee di dato generate da letture in memoria e come sono gestite nel DLX?

A) Di norma le alee di dato sono gestite con la strategia del forwarding nel DLX pipelined. Le istruzioni di LOAD risultato problematiche da gestire con questa strategia dato che la lettura in memoria causa cicli di wait, che possono anche essere numerosi se il dato non è cached.
Per via del ritardo della memoria non posso semplicemente aggiungere un ulteriore ingresso ai mux della ALU dalla memoria perchè rischio di usare un dato che non è stato ancora letto.
Dunque l'alea non può essere eliminata con il forwarding e causa uno stallo.



Q3 2018_02_09)

Le istruzioni di scrittura in memoria di dati signed e unsigned sono differenziate nel DLX? Si o no? Motivare, sinteticamente e chiaramente, la risposta.

A) Quando scrivo un dato in memoria il dato scritto non deve mai essere esteso (al massimo viene troncato), perchè tutti i registri sono a 32 bit e posso solo scrivere BYTE, HW o WORD. Dunque non importa il se il dato è signed o unsigned.

Q2 2020_02_04)

A cosa serve il multiplexer posto sull'uscita del register file nel DLX pipelined?

A) Serve a gestire un' alea di dato dovuta a un'istruzione in WB contemporanea a una in ID : al posto di leggere il dato (stale) che è nel register file, leggo direttamente il risultato dello stadio di WB, usando un MUX controllato dalla forwarding unit.

Q3 2020_02_04)

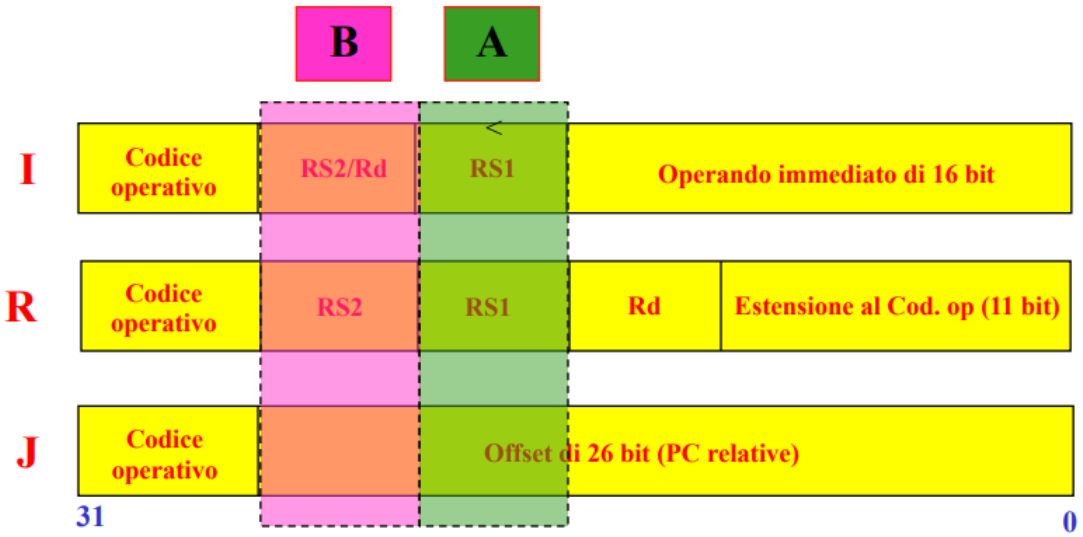
a) Quali strategie è opportuno adottare nella gestione dei registri nella scrittura di un interrupt handler?
b) Cosa può accadere se non si adottano tali strategie?

A) Un interrupt handler può essere chiamato in un momento arbitrario durante l'esecuzione, e per questo è importante considerare quali registri andrà a modificare l'handler.
Se questi registri vengono utilizzati in altre parti del codice utilizzandoli nell'handler rischio di alterare l'esecuzione di quel codice. Per questo è importante o dedicare alcuni registri esclusivamente all'handler oppure salvare il valore originale dei registri utilizzati per poi rimpostarli a quel valore alla fine dell'handler.

Q2 2020_06_16)

A cosa servono i due registri A e B nel DLX sequenziale?

A) I registri A e B conservano i due operandi sorgente estratti dal register file. Sono estratti in modo speculativo (sfruttando il formato coerente delle istruzioni della ISA DLX) durante la fase di decode.



Q3 2020_06_16)

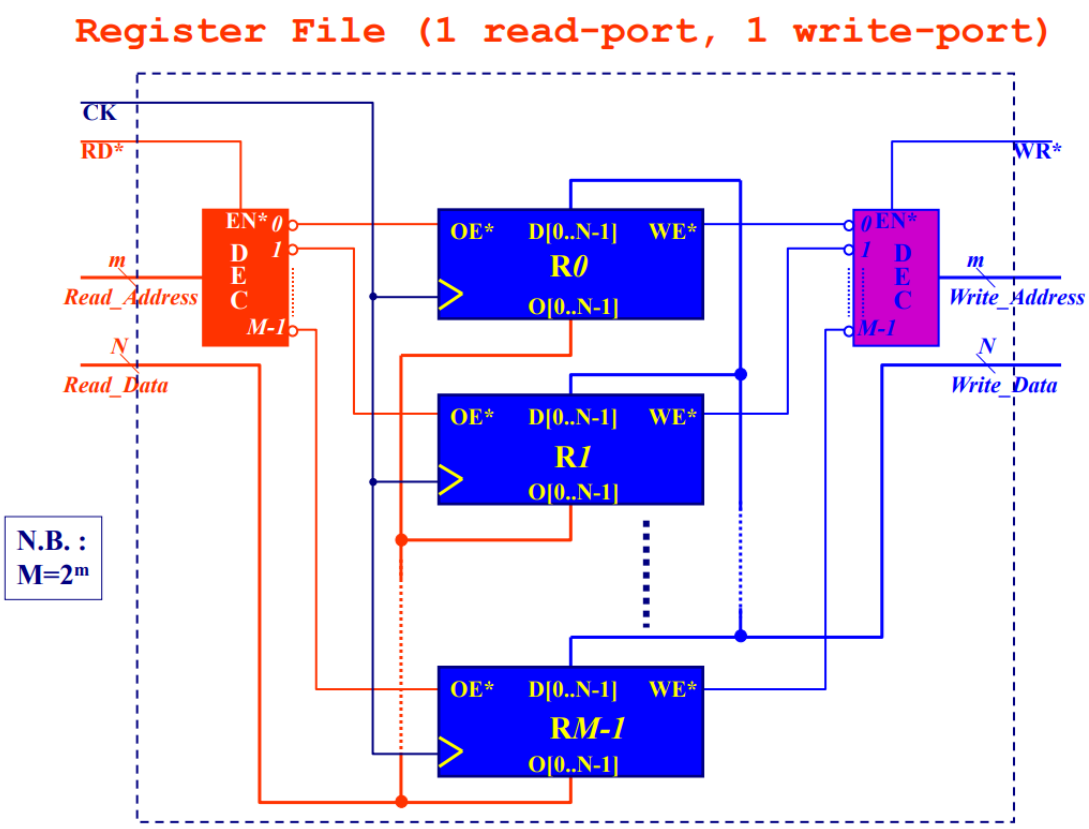
È possibile annidare interruzioni nel DLX?
In caso affermativo, come? In caso negativo, perché?

A) No, in modo superficiale perchè la IEN(Interrupt Enable Flag) viene disabilitata durante l'esecuzione di un interrupt handler. Il motivo più profondo di questa limitazione del DLX è l'assenza di uno *stack (last in, first out)*, che sarebbe necessario per conservare in modo appropriato gli indirizzi di ritorno dalle interruzioni annidate.

Q2 2017_06_21)

a) Spiegare la funzione del register file nel DLX indicando chiaramente quali sono tutti i segnali di input e output di tale modulo
b) Descrivere in dettaglio la struttura interna di un register file

A) a) Il register file del DLX ospita i 32 registri general purpouse a 32 bit del DLX. Tutti questi registri sono edge triggered, per questo il register file prende in input un clock.
Inoltre dal register file si possono leggere due registri e scriverne uno ad ogni fronte del clock, e quindi ho 2 OE, 2 indirizzi di lettura, 2 output a 32 bit corrispondenti, 1 WE, 1indirizzo di scrittura, e un input a 32 bit corrispondente.
b) Il register file internamente è composto da 32 registri edge triggered a 32 bit il cui output enable è controllato da 2 decoder che interpretano gli indirizzi di lettura, mentre il loro write enable è controllato da un decoder che interpreta gli indirizzi di scrittura.



!!! In figura è rappresentato un RF con una singola read port, andrebbe aggiunto un altro decoder per l'output !!

Q3 2017_06_21)

a) Quali sono le fasi comuni a tutte le istruzioni definite mediante il diagramma degli stati del controller?
b) Indicare possibili varianti al punto a) e le conseguenti modifiche hardware alla struttura del DLX sequenziale

A) Tutte le istruzioni condividono le fasi di instruction fetch e instruction decode : logicamente prima di terminare l'instruction decode non so che tipo di istruzione sto trattando, quindi eseguo le stesse operazioni.
All'inizio di ogni istruzione eseguo quindi

MAR ← PC
IR ← M [MAR]

PC ← PC + 4
A ← RS1
B ← RS2

b) Posso unire le prime due microoperazioni in una singola operazione IR ← M [PC] se connetto il PC direttamente agli indirizzi della memoria con un MUX.