1 Lezione del 22-11-24

1.1 Microprogrammazione della parte di controllo

Le tecniche di microprogrammazione ci permettono di sintetizzare la parte di controllo di reti complesse. In particolare, associata una codifica ad ogni stato del registro STAR, e chiamata questa codifica per ogni STAR **μ-indirizzo** dello stato, possiamo creare una tabella:

	μ-istruzione			
μ-indirizzo	**μ-codice**	c_{eff}	**μ-indirizzo T**	**μ-indirizzo F**
00	00	1	00	01
01	11	0	10	01
...				

dove si associa ad ogni μ-indirizzo, quindi ad ogni stato, una **μ-istruzione**: intendiamo una μ-istruzione come un insieme di variabili di comando associate a quello stato (il **μ-codice**), una **variabile efficiente** c_{eff} sulla base della quale si effettuano i μ-salti, e due μ-indirizzi, T e F, che determinano il salto successivo sulla base del valore, rispettivamente vero o falso, della c_{eff}.

A partire da una tabella del genere, possiamo sintetizzare la PC secondo due modalità:

- **Modello basato sui μ-indirizzi:** mi sa non riesco ora
- **Modello basato sulle μ-istruzioni:**

1.1.1 Reintrodurre i salti a più vie

Nel caso di salti a più di due vie, si dovranno considerare più condizioni in cicli di clock differenti. fai esempio Questo diventa poco efficiente quando i salti sono a un elevato numero di vie, in quanto per n possibili μ-salti si perdono $\sim n$ cicli di clock.

Nella pratica, i processori sono spesso progettati per compiere salti a due vie, tranne che in due casi particolari:

- All'**inizio** della fase di fetch, cioè quando si legge il **formato** dell'opcode, dove si dovrà saltare a un blocco μ-codice diverso a seconda della posizione degli operandi fai esempio assembler. Si perderanno quindi $\sim f$ cicli per f formati possibili delle istruzioni;
- Alla **fine** della fase di fetch, cioè quando si determina il salto al blocco di μ-codice che gestisce la **fase di esecuzione** dell'istruzione. Si perderanno quindi $\sim i$ cicli per i istruzioni possibili.

Una soluzione al problema dei salti a più vie e data quindi dal **Multiway Jump Register**.

1.1.2 Multiway Jump Register

L'MJR non è un gruppo punk americano ma un **registro operativo** destinato a contenere indirizzi di salto. Generiamo l'ingresso del MJR attraverso la parte operativa della sintesi, e lo utilizziamo nella parte di controllo.

Per codificare la presenza del MJR, nella ROM della sintesi della parte di controllo dovremmo introdurre una nuova uscita, il **μ-tipo**. Il valore del μ-tipo determina il tipo

di salto che vorremo eseguire: μ-tipo a 0 significherà salto standard a 2 vie, e μ-tipo a 1 significherà salto basato sul MJR.

parla del b_k di sovrascrittura del MJR

1.2 Sottoliste

Talvolta può convenire strutturare una descrizione di RSS con sottoliste simili a **sotto-programmi**. Porzioni di μ-programma diverse potranno quindi essere raggiunte da stati di partenza diversi, che riporteranno allo stato di partenza stesso al termine della loro esecuzione attraverso un processo simile a quello delle CALL e RET viste sull'assembly (integra). Questo può essere implementato nella pratica, inserendo il μ-indirizzo successivo all'esecuzione della sottolista nel MJR, cioè impostando b_k a 0 per quello stato, e inserendo quindi il μ-indirizzo dell'inizio della sottolista in STAR. A questo punto la rete di controllo "eseguirà" il μ-codice ed effettuerà i μ-salti specificati dalla sottolista fino al passo finale, che rimetterà MJR in STAR, e quindi riprenderà l'esecuzione dal μ-indirizzo memorizzato prima della "chiamata" della sottolista.

Due limitazioni di questo approccio sono che MJR diventa inutilizzabile durante l'esecuzione della sottolista, e sopratutto che un singolo MJR ci permette un solo livello di annidamento di sottoliste. Per avere più livelli avremo bisogno di una **pila di MJR**, che però non è trattata in questo corso.

1.3 Struttura del calcolatore

Siamo arrivati ora a a poter descrivere in Verilog un **sistema completo** di:

- Processore;
- Memoria;
- Interfacce;
- Dispositivi di I/O

collegati fra di loro attraverso una rete di interconnessione.

All'interno del **sottosistema di ingresso/uscita** distinguiamo **interfacce** e **dispositivi**. Gli ultimi si occupano effettivamente di ottenere codifiche di dati dal mondo esterno, mentre le prime gestiscono i dispositivi in modo che questi possano colloquiare col processore. Le interfacce contengono un piccolo numero di **registri di interfaccia** su cui il processore può leggere o scrivere.

La **memoria principale** sarà formata in larga parte da memoria RAM, e conterra in ogni istante le **istruzioni** e i **dati** che questo elabora. Una parte della memoria principale dovrà essere implementata attraverso memoria ROM, in quanto c'è da risolvere il problema dello stato di avvio del processore introducendo dati predefiniti che vengono puntati per primi dall'istruction pointer. (credo) Il modello che andremo a studiare poi sarà dotato di memoria video, che conterrà le immagini visualizzate sullo schermo, e sarà anch'essa in diretta comunicazione col processore.

Il **processore** eseguirà il ciclo **fetch-execute**, prelevando dala memoria principale **istruzioni operative** e **istruzioni di controllo**. Dovrà partire in una determinata configurazione dei registri, ottenuta collegando opportunamente piedini di `/preset` e `/preclear` alla linea di `/reset`, in modo da inizializzare (come detto prima) l'instruction pointer a puntare ad una locazione di memoria nota che lanci un determinato programma in memoria, detto **bootstrapper**.

Per quanto ci riguarda, il calcolatore sarà formato da una serie di RSS, e il processore potrà essere sintetizzato attraverso la separazione PO/PC.

1.3.1 Memoria

La nostra memoria sarà formata da uno spazio lineare di 2^{24} lozazioni di memoria da un byte, per un totale di 16 MB indirizzati su 24 bit (3 byte). Lo spazio di I/O sarà formato da uno spazio lineare di 2^{16} locazioni di memoria da un byte, per un totale di 64 B indirizzati su 16 bit (2 byte).

1.3.2 Processore

Il processore sarà dotato di 3 tipi di registri:

- **Registri accumulatore:** AH e AL, da 8 bit ciascuno;
- **Registro dei flag:** 8 bit con 4 significativi: CF (0), ZF (1), SF (2), OF (3);
- **Registri puntatore:** da 24 bit (3 byte) ciascuno (devono contenere indirizzi di memoria), sono:
  - **IP**: l'instruction pointer;
  - **SP**: lo stack pointer;
  - **DP**: il data pointer, che come vedremo contiene le locazioni degli operandi di istruzioni.

Come avevamo visto, non programmeremo il nostro processore attraverso il linguaggio macchina, ma con un linguaggio assembler che codifica le istruzioni macchina, nella forma già vista:

```
OPCODE source, destination
```

Questo linguaggio sarà simile a quello già studiato, cioè dei processori Intel x86. La differenza sarà che avremo come problema il dover effettivamente codificare ciò che scriviamo in assembler in istruzioni in linguaggio macchina da fornire al processore (adesso non stiamo solo *programmando*, ma anche *progettando* il processore).

1.3.3 Modalità di indirizzamento

Avevamo visto le seguenti modalità di indirizzamento per le istruzioni:

- Di registro;
- Immediato;
- Di memoria;
- Delle porte di I/O riempi

listone istruzioni