**Parte 0**

**Cos’è l’FPGA?** Field Programmable Gate Array, è una scheda composta da tantissimi componenti (porte logiche, flip flop) programmabile sul campo, in pratica è come se fosse una lastra di silicio giá riempita di componenti che vanno soltanto collegati tra loro. Possono essere programmati indefinitivamente dall’utilizzatore finale. È un approccio molto piú efficiente rispetto ai sistemi **ASIC**, Application Specific Integrated Circuits, delle lastre di silicio programmabili in fabbrica e prettamente dedicate ad una specifica attivitá, non programmabili sul campo e generalmente piú complesse. Inoltre l’FPGA sintetizza una funzione direttamente a partire dalla tabella di veritá, quindi non è necessario effettuare la minimizzazione. Il linguaggio Verilog è preferito per il design degli ASIC, il VHDL invece per gli FPGA.

**Come influiscono gli aspetti tecnologici sul progetto?**  
La tecnologia realizzativa impatta fortemente sul progetto di un sistema digitale. È necessario orientare la scelta dei componenti logici fondamentali del design in modo da poterli sfruttare al meglio; valutare la convenienza ed effettuare eventualmente ottimizzazioni in fase di design; valutare le prestazioni di un design in termini di occupazione di area su silicio e tempi di risposta.

**Il VHDL:**  
Il VHDL (Very High Speed Integrated Circuits Hardware Description Language) è un linguaggio di descrizione dell’hardware: non descrive quali operazioni un esecutore deve svolgere per ricavare il risultato di una elaborazione, bensì descrive gli elementi che costituiscono il circuito digitale in grado di effettuare l’elaborazione richiesta. È un linguaggio concorrente in quanto tutto l’hardware è concorrente. Non è case sensitive.

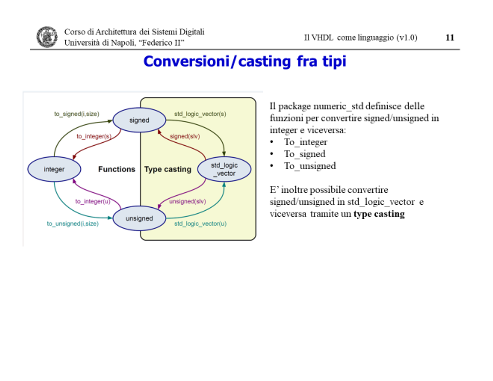
Una descrizione VHDL puó essere fatta a diversi livelli di astrazione:

* **Dataflow**: utilizzato tipicamente per descrivere circuiti combinatori, consiste nell’assegnare ad ogni segnale il valore risultante da un’espressione booleana;
* **Structural:** consiste nel definire la funzionalitá di un circuito in termini di componenti e loro interconnessioni; Il costrutto **component** consente di **dichiarare i componenti utilizzati da un design**, mentre **con port map si può istanziare il componente nella parte di descrizione funzionale.**
* **Behavioral**: consiste nel definire la funzionalitá di un circuito in termini di una descrizione del suo comportamento (come se il comportamento fosse descritto da un algoritmo con costrutti simili a quelli dei classici linguaggi di programmazione). Una descrizione di questo tipo potrebbe non sempre essere sintetizzabile. Viene usato anche nei testbench in quanto vogliamo una descrizione piú ad alto livello per uno strumento che non fa altro che istanziare altri componenti al suo interno, ciascuno descritti in modo piú specifico.
* **EXTRA-> RTL Register Transfer Level:** descrizione di un design in termini di registri e porte logiche e del flusso di segnali attraverso di essi. Flusso di dati attraverso i registri.

I numeri possono essere rappresentati interi o reali. Le stringhe di bit in binario, ottale o hex.

Cos’è un oggetto in vhdl? Un entitá che ha un nome, un tipo e un valore. Esistono variabili, costanti e segnali. Le variabili vano usate solo nei corpi sequenziali, i process, e devono essere inizializzate. Si usano per mantenere un informazione temporanea o lo stato nei process.

Il vhdl è fortemente tipizzato, si possono definire i propri tipi e vengono usati quelli disponibili dallo standard IEEE 1164 std\_logic (U,X,0,1,Z,W,H,L,-)

* Conversioni / casting fra tipi
  + Signed e unsigned sono creati a partire dagli standard logic. E’ proprio un casting, non è una funzione che trasforma una cosa in un’altra.
  + Uso la parolina signed(vector\_da\_trasformare)
  + Per gli integer posso usare to\_signed e to\_unsigned ma devo definire la size. Devo dire io il numero di bit

Ricorda tipo enumerazione : TYPE stato IS ('q0','q1','q2','q3')  
Tipo array: TYPE myArray IS ARRAY (7 DOWNTO 0) of std\_logic;

Un sistema digitale è progettato come se fosse composto da un insieme di moduli organizzati ad albero.

* C’è un top-level module (radice dell’albero) che istanzia e collega altri moduli.
* Un modulo può essere un componente o un top-level module del design.
* Ogni modulo in VHDL è descritto da due parti:
* \*\*entity\*\*: descrive l’interfaccia del modulo, ovvero i \*generics\*
* \*\*architecture\*\*: descrive il funzionamento del sistema, ovvero la relazione ingresso-uscita

**Per cosa viene usata una dichiarazione entity? Cosa include?**

Per dichiarare gli oggetti (segnali di IO ed eventuali parametri *generics*) che potrenno essere usati. specifica del componente in termine dei suoi porti, specificando nome, tipo e direzione. Può includere la specificazione di costanti per controllare la struttura e il comportamento dell’entity (tramite i *generics*). A che serve? A specificare ai tipi e i modi dei ports e dei generics. Quando il modulo sarà istanziato verrò stabilita la corrispondenza tra ports e segnali.

**Cosa sono i Generics?**

Li uso quando voglio fare un design usando parametri che possono variare. Sono specificati nell’interfaccia di una entity e permettono di parametrizzarne la descrizione. Il loro modo deve essere specificato in almeno uno dei seguenti modi:

* 1. Come valori di default nella dichiarazione di entity (recommended, \*\*rende il componente sintetizzabile\*\*)
* 2. All’atto della dichiarazione del component
* 3. All’atto dell’istanziazione del component.

Se usassi tutti e 3 i modi il valore usato è quello a maggiore prioritá, ovvero il terzo caso.

**Cosa sono i ports?**

Nome, tipo, modulo. Possono essere collegati ad un port solo segnali di tipo compatibile. I modi di collegamento disponibili sono IN, OUT, BUFFER, INOUT ma usiamo solo in e out e segnali di appoggio

**Possono esistere diverse architetture per una stessa interfaccia?**

Si, ogni architecture puó descrivere lo stesso modulo da diversi livelli di astrazione. Eventualmente ha valore soltanto l’ultima architettura scritta. Si parla quindi di binding. Un’architettura e composta da parte dichiarativa e body. L’istanziazione è uno statement concorrente, sempre attivo. Un componente si istanzia assegnando ordinatamente i segnali, sia usando una associazione esplicita per nome con label diverse.

**Qual è la differenza tra associazione nominale e posizionale?**

L’ordine in cui associamo i segnali. Nominale è del tipo a0=>nome, posizionale invece è segnale1, segnale2, segnale3 ecc.

**A cosa servono i costrutti iterativi?**

Il loro vantaggio è che non si muovono nel tempo ma nello spazio, sono utili per istanziare componenti opportunamente interconnessi. Per descrivere efficacemente queste situazioni il vhdl mette a disposizione il costrutto generate che permette di istanziare iterativamente un insieme di blocchi. Tale statement puó essere utilizzato sia con il FOR che con il IF.

**TECNICHE DI SIMULAZIONE DELL’HARDWARE:**

La simulazione hardware è una **fase critica nello sviluppo di sistemi digitali.** Consente agli ingegneri di verificare e validare il comportamento del design prima della realizzazione fisica, riducendo i costi e i tempi dovuti a possibili errori. Per circuiti complessi, **la simulazione deve essere non solo accurata ma anche efficiente**, al fine di gestire la grande quantità di componenti e le intricate interazioni tra di essi.

1. **SIMULAZIONE A TEMPO QUANTIZZATO**

Il tempo di simulazione procede per **incrementi di ampiezza prefissata**, detti quanti di tempo **(passo di quantizzazione)**; E’ **incapace di adattarsi alla dinamica del sistema,** dunque è utile per sistemi in cui i cambiamenti avvengono in maniera prevedibile e periodica. Dovrei fare dei quanti di tempo piccolissimi (più piccoli del ritardo) per vedere le variazioni del circuito, il che **può rendere la simulazione inefficiente,** specialmente se la maggior parte del tempo non succede nulla di rilevante.

PRO: facile da implementare in assenza di ritardi, puó simulare anche variazioni continue dei potenziali;

CONTRO: non si adatta alla dinamica del sistema, difficile gestire i tempi di propagazione delle porte, il passo di quantizzazione dovrebbe essere minore delle costanti di tempo del circuito → cresce l’onere computazionale delle simulazioni;

OTTIMIZZAZIONI: Determinazione dello scope di una variazione del potenziale di un nodo; il passo di quantizzazione puó essere variabile ed adattarsi all’evoluzione del circuito.

1. **SIMULAZIONE AD EVENTI**

Il tempo di simulazione procede per **incrementi di ampiezza variabile** (solo **quando ci sono variazioni in un nodo si calcola la propagazione di queste variazioni)**; Permette di **seguire la dinamica del sistema,** avanzando al verificarsi di eventi significativi. **Efficienza**: Il tempo di simulazione salta i periodi in cui non si verificano eventi, concentrando le risorse computazionali solo quando è necessario. Il tempo di simulazione avanza solo quando c’è qualche variazione del segnale → viene triggerato lo scheduling delle successive transazioni dei segnali. La simulazione salta i tempi dove non succede nulla

PRO: permette una simulazione veloce dei circuiti digitali (VHDL).

CONTRO: difficile da implementare.

**Il modello di tempificazione VHDL:**

Il VHDL permette di rappresentare il fatto che diversi oggetti ( assegnazioni di segnali, process…) evolvano contemporaneamente (“**concorrentemente**”); Una *assegnazione* su un segnale comporta l'aggiunta di una ***transazione*** (transaction) sul segnale target: una **transazione ha un valore e una componente di tempo (value, time) cioè il valore che verrá assegnato al segnale e dopo quanto tempo.** Non sempre **una assegnazione comporta una variazione del valore di un segnale. Si parla di evento** quando questo avviene, altrimenti di semplice transazione.

**DELTA CYCLES:**

Se si assegna un valore ad un segnale senza specificare il ritardo si assume che questo sará 0. La transazione è schedulata per lo stesso istante dell’assegnazione MA **concretamente il simulatore VHDL la eseguirà nel ciclo di simulazione successivo (→ il *delta cycle* successivo);** Il tempo della simulazione è del tipo: x\_1T+x\_2\delta dove **T** è un **tempo** (ns, ms, s) e **delta è un ciclo di simulazione** che **non corrisponde a tempo reale.** In altre parole, una transazione con tempo zero scade un “delta” dopo che è stata posta sul segnale;

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamenteI ritardi **inerziali** rappresentano un fenomeno di ritardo che però non propaga tutti gli impulsi (0→1→0, 1→0→1), ma solo quelli che durano più di una certa soglia; poco usati in vhdl.

I ritardi **transport** invece propagano sempre le transazioni indipendentemente dalla distanza temporale fra due transazioni successive;

**Cos’è una macchina combinatoria?**

Una macchina combinatoria è un circuito costituito da porte logiche le cui uscite in ogni istante assumono valori che dipendono unicamente dal valore degli ingressi nello stesso istante. Esse hanno un ritardo di propagazione dei segnali.

* Macchine per l**’istradamento di segnali** e la **connessione fra sistemi** (reti di priorità, encoder/decoder, multiplexer/demultiplexer)
* Macchine per la **conversione di codici** (reti di transcodifica)
* Macchine per la realizzazione di **funzioni aritmetiche e logiche** (sommatori, moltiplicatori, comparatori..)

**Come si progetta una macchina combinatoria?**

* Specifica in linguaggio naturale.
* Formalizzazione: \*\*tabelle di verità\*\*.
* Sintesi: espressione algebrica in forma canonica (somma di prodotti o prodotto di somme)
* Ottimizzazione: dalla forma canonica si ottiene un’espressione migliore.
* Realizzazione circuitale

Non è sempre conveniente e possibile procedere in questo modo. Talvolta si preferisce progettare per composizione di macchine giá esistenti.

**AUTOMI E MACCHINE SEQUENZIALI:**

Le ***macchine sequenziali*** o, più in generale, i sistemi sequenziali, sono sistemi in cui **le uscite** in un certo istante dipendono **non solo dal valore degli ingressi nello stesso istante, ma anche dagli ingressi precedenti.** La conoscenza degli ingressi precedenti costituisce lo **stato** che puó essere definito come la **memoria** del sistema.

Il **comportamento** di una macchina sequenziale può essere rappresentato in un modo non ambiguo tramite un **automa a stati finiti (ASF)**, un modello matematico di “sistema” caratterizzato da un **insieme finito di ingressi, uscite e stati.**

Immagine che contiene testo, schermata, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente E’ possibile rappresentare graficamente un ASF mediante un **grafo** detto **diagramma degli stati.** Esso è orientato ed ha V nodi, E archi, L etichette degli archi. Per gli automi di Moore l’uscita è associata direttamente allo stato.

Immagine che contiene testo, schermata

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

È possibile passare da Mealy a Moore usando appositi algoritmi di trasformazione. Tipicamente la rappresentazione di Moore contiene piú stati perché dovendo associare l’uscita non ad una transizione ma ad uno stato, bisognerebbe replicare tutti gli stati nei quali si può arrivare con uscite diverse creando stati diversi.

**Modello di Huffman:** separa esplicitamente questa memoria di stato dalla parte del sistema che calcola le funzioni di transizione e di uscita, che può essere modellata come una macchina combinatoria.

**Macchine sincrone e asincrone:**

* **Asincrone: partendo da qualsiasi stato e applicando un qualsiasi ingresso (anche per un tempo molto lungo), raggiungono uno stato cosiddetto *stabile*,** in cui cioè permangono fino all’applicazione di un nuovo ingresso (sono sensibili alle variazioni degli ingressi)
* **Sincrone:** avviene che almeno per uno stato e per un ingresso, a partire da tale stato e applicando tale ingresso esse continuano la loro evoluzione fra gli stati finché l’ingresso è applicato, senza fermarsi in uno stato stabile. Nella macchine sincrone lo stato in cui si trova la macchina in un dato istante dipende dalla durata del segnale di ingresso e dai tempi di reazione della macchina.

**Macchine sincrone:**

**Tempificazione**

*Per garantire il corretto funzionamento* della macchina, **è necessario considerare segnali di ingresso tali da causare una sola transizione di stato alla volta**. Ciò può essere ottenuto rispettando una delle seguenti condizioni: gli ingressi devono essere impulsivi, ossia devono **durare solo per un tempo limitato ben preciso** (proprietà legata alla caratteristica realizzativa del circuito) oppure gli ingressi sono a livelli ma il sistema deve **disporre di un’ulteriore ingresso di *sincronismo*** che renda possibile la transizione da uno stato all’altro solo in presenza di tale segnale.

Il modello piú semplice da realizzare è quello delle macchine a sincronizzazione esterna dove gli ingressi vengono campionati solo in corrispondenza di un segnale di sincronizzazione esterno impulsivo (clock). Poiché le transizioni avvengono in generale tra stati non stabili, **è necessario congelare lo stato prossimo calcolato in base al valore dello stato corrente e dell’ingresso in un dato istante (identificato dal segnale di sincronismo) all’interno di una memoria,** da cui esso viene letto al successivo segnale di sincronismo per calcolare la nuova transizione di stato e la nuova uscita. Il modello di Huffman si realizza allora utilizzando un vero e proprio elemento di memoria nelle linee di retroazione.

**Modello fondamentale**

Le macchine sincrone hanno degli **elementi di memoria sulla linea di retroazione per mantenere lo stato precedente.** Tale elemento di memoria è esso stesso una macchina asincrona in cui la retroazione è rappresentata da fili.

**Macchine asincrone:**

**Tempificazione**

Nelle macchine **asincrone** una variazione degli ingressi porta sempre in uno stato stabile e ció consente di realizzare la memoria di stato del modello di Huffman semplicemente con dei fili di retroazione. Lo stato viene codificato come una stringa binaria, e quindi è necessario un filo per ogni bit della stringa. Perché la macchina funzioni correttamente però, **è necessario che la durata degli ingressi sia tale da consentire la transizione da uno stato stabile ad un altro stato stabile.**

**Modello fondamentale**

* Sia gli ingressi che le uscite di una rete in generale possono essere a **livelli** o **impulsivi**
  + i primi sono segnali che si mantengono costanti fino a che non si modifica il valore logico, i secondi presentano dei picchi di breve durata in corrispondenza del valore 1 e assenza di impulso per lo 0 (in tal caso bisogna sapere quando osservare il segnale)
* Una macchina con soli ingressi a livelli funziona solo se è asincrona perché per ogni stato e per ogni ingresso si porta in uno stato stabile
  + ATTENZIONE: La durata di ciascun segnale di ingresso deve consentire alla macchina di transitare verso uno stato stabile passando per una serie di stati intermedi, quindi deve essere opportunamente determinata!

**RIASSUMENDO: Una macchina sincrona può essere realizzata solo utilizzando ingressi impulsivi (oppure ingressi a livelli e segnale di sincronismo), di durata appena sufficiente a far commutare lo stato e l’uscita**

**Latch** (significa CATENACCIO, quando non c’è variazione è trasparente, se A = 0 mantiene lo stato, se A = 1 segue l'ingresso).

**Edge Triggered** (Si aziona sui fronti di salita di A, legge ingresso alto due volte quindi resta SEMPRE alto in questo caso).

**Master Slave** (Campiona sul fronte di salita però non varia subito, solo quando arriva il fronte di discesa il valore viene rilasciato, quindi si ritrova in uscita. Poi fino al prossimo fronte omologo (in questo caso di discesa) non varia, al prossimo fronte di salita di A campiona di nuovo 1 quindi sul fronte di discesa successivo non varierà.

**Il FF ideale è il MS.** L’Edge Triggered su fronte di salita ci piace perchè ho foto nell'istante preciso e so in quel momento valore, però nel Latch vado bene solo con registro parallelo parallelo, con quello a scorrimento entro in crisi perchè ho la ripercussione di variazioni a cascata -> Edge triggered ha concetto di foto in un istante quindi va bene ma purtroppo la commutazione delle uscite avviene quando l'abilitazione A è alta -> Se immagino di avere il pezzo di circuito e a valle altre cose con abilitazione condivisa tra pezzi combinatori, sequenziali, ecc. potrei avere che qualcosa a valle può ancora reagire perchè quando chiudo la variazione non chiudo pure l'abilitazione -> **IL MASTER SLAVE FA PAURA, FA FOTO IN UN ISTANTE MA LA FOTO LA FA VARIARE SUL FRONTE DI DISCESA.** **Ma i MS non vengono fatti PERCHE’ IL MS è COMPLESSO -> Ha 6 stati**

**Negli FPGA ci sono i flip-flop D edge-triggered.**

**Progetto di macchine sequenziali**

Nell’ambito dei sistemi digitali, oltre che per la realizzazione di dispositivi di memorizzazione, le macchine sequenziali hanno larga applicazione nell’implementazione delle **unità di controllo** (a logica cablata o microprogrammata). L’unità di controllo di un sistema è la parte del sistema che ne **definisce il comportamento** (ad esempio implementando uno specifico algoritmo), impartendo opportuni comandi (*segnali di controllo*) alla cosiddetta **parte operativa,** costituita dall’insieme dei componenti architetturali che implementano praticamente le operazioni richieste (es. registri, unità aritmetiche, bus, ecc.)

Come abbiamo visto per le macchine combinatorie, anche per quelle sequenziali ci sono diversi approcci alla progettazione VHDL

* **Approccio comportamentale**: viene descritto il comportamento della macchina come definito dal relativo *automa*/diagramma degli stati (**finite state machine**)
* **Approccio strutturale**: vengono descritti i componenti della macchina e le relative interconnessioni. Da un punto di vista strutturale, il progetto prevede la descrizione dei componenti della rete logica (porte logiche e flip-flop) e delle loro interconnessioni.

Il costrutto fondamentale per descrivere una FSM è il ***process.*** Il modello più vicino alla realizzazione hardware di una FSM prevede **due processi**, uno puramente combinatorio che calcola la funzione stato prossimo e la funzione di uscita, e l’altro che rappresenta l’elemento di memoria (cioè il banco di flip-flop) che consente di retroazionare lo stato (ed eventualmente di registrare le uscite).

E’ possibile anche usare un **terzo processo** per definire il comportamento del segnale di uscita separatamente OPPURE è possibile modellare un automa con un unico processo la cui descrizione risulta piú immediata.

Descrizione behavioral di una macchina sequenziale SINCRONA

Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente

Volendo applicare il pattern con un singolo process, tutti i suoi ingressi vanno nella sensitivity list cioè stato corrente e ingresso i.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

I tool di sintesi incorporati negli IDE di sviluppo della Xilinx sono in grado di identificare propriamente una FSM e quindi di sintetizzarla purché vengano adottati degli accorgimenti nella programmazione:

È necessario identificare uno **stato iniziale,** che la macchina assume al power up

È necessario inserire un **segnale di RESET**, sincrono o asincrono

Se si utilizza la codifica con doppio process è necessario inserire nella sensitivity list del processo combinatorio (funzione stato prossimo e uscita) il segnale di stato (corrente) e tutti gli input della macchina

Se l’output deve essere inserito in un registro allora l’assegnazione va fatta nel processo sequenziale (quello che si occupa della retroazione)

Xilinx supporta diverse tecniche di **codifica degli stati** e può operare una ottimizzazione per ridurre l’area occupata o ottenere prestazioni migliori

* Auto State Encoding: XST cerca di trovare l’algoritmo più efficace in maniera automatica
* One-Hot State Encoding (default): associa un bit ad ogni stato e quindi in ogni istante solo una variabile di stato è asserita (appropriato per gli FPGA che hanno a disposizione molti flip-flop)
* Gray State Encoding: garantisce che cambi un solo bit alla volta quando si passa da uno stato all’altro (indicato quando si implementano controller con molti stati senza salti)
* Compact State Encoding: minimizza il numero di bit della codifica
* User State Encoding: usa la codifica specificata dall’utente

Sta roba degli stati puó essere indotta via codice ma in genere VIVADO se ne batte il ca quindi meglio agire dalle impostazioni del progetto (synthesis, fsm extraction).

**Guida alla scheda Artix7**

La scheda lavora con il concetto di **slices**, pezzi di componente composti da una parte combinatoria ed una parte sequenziale, banchi di ram distinti e memoria distribuita.

È possibile fornire potenza alla scheda in modi diversi selezionabili attraverso lo spostamento del jumper JP3 (vicino al jack di alimentazione). Noi useremo la modalitá di alimentazione USB.

Dopo aver acceso la scheda essa puó essere programmata in modi diversi: uno dei modi è caricare il file di programmazione direttamente attraverso il cavo che collega la scheda al pc. È possibile selezionare le modalitá di programmazione spostando il jumper JP1 ed il media selection jumper JP2. In questo modo è possibile scegliere se programmare la scheda prelevando il file bitstream dalla memoria flash all’accensione oppure da una micro sd esterna. Una vola programmata si accende il led DONE mentre premendo il pulsante PROG si ripristina la memoria di configurazione iniziale.

La scheda include un singolo oscillatore a cristallo da 100Mhz collegato al pint E3, quindi avremo un clock a 100MHz disponibile per tutti i componenti della scheda.

In generale comprende 2 led rgb, 16 switch a scorrimento, 6 pulsati, 16 led individuali, un display a 8 cifre da 7 segmenti ciascuna. Il bottone rosso cpu reset viene usato quando ci mappiamo un processore ma possiamo usarlo come reset del nostro design ma ATTENZIONE, funziona a logica inversa rispetto agli altri, cioè da1 quando non è premuto e 0 quando è premuto.

Per usare i vari componenti ho bisogno di specificare dei **vincoli fisici** cosí che vivado sia in grado di riconoscerli. Vado ad usare i file xdc dove andremo a de commentare le righe che ci servono ed a sostituire i nomi placeholders con i nomi usati nel nostro design per collegare le componenti ai nostri fili.

Per generare il bitstream clicco su generate bitstream, open hardware manager, open target, autoconnect, program device. Nella cartella runs, impl, troviamo il bitstream, program

**DEFINITIVO:**

**Come si realizza una macchina aritmetica?**

Una macchina aritmetica puó essere realizzata sia con un approccio combinatorio che con uno sequenziale. In particolare proprio i Moltiplicatori rappresentano una parte fondamentale nel progetto e nello sviluppo di molti modelli e algoritmi di machine learning, in particolare per quanto riguarda lo studio delle reti neurali.

Una macchina di questo tipo puó essere realizzata a livello porta, creando un circuito per ciascuna operazione, a livello comportamentale o a livello strutturale. I bit degli operandi possono essere presentati in modi diversi e tale scelta impatterá sulla natura, sulla complessitá e sulla velocitá della macchina aritmetica che si sta progettando. Si parla quindi di presentazione parallela (i bit degli op sono presentati in ingresso e i risultati sono calcolati contemporaneamente a meno di ritardi di programmazione), seriale pura (bit degli op e delle uscite vengono presentati in serie), seriale a gruppi o mista (un op è presentato in serie e l’altro in parallelo). In generale la presentazione in parallelo favorisce una maggior velocitá della macchina a causa dei ritardi che risultano minori (di una sola rete combinatoria).

**Cos’è un Half Adder? E un Full Adder?**

Un half adder è un componente che somma 2 BIT, presenta in uscita una somma S ed un riporto C. La differenza con il full adder è che questo somma due numeri binari, è realizzato con due half adder e una porta OR per gestire il riporto uscente e presenta in ingresso un riporto c0.

**Cos’è un Ripple Carry Adder?**

Un RCA è detto sommatore a propagazione di riporto e deriva dall’algoritmo manuale per la somma di due numeri interi positivi di n bit. Ha una struttura formata da un half adder in entrata e tanti full adder collegati in cascata. Alla prima cella a destra vengono dati in ingresso i primi due bit dei due operandi, tale cella produrrá una somma s0 e un riporto c1 che andrá in ingresso come carry al full adder successivo. Inoltre ponendo c0 pari a 0 è possibile eliminare il half adder ed avere tutti full adder in modo da rendere piú regolare il sistema.

Il ritardo risulta essere pari a 2∆n con ritardo di porta e l’area occupata è 5n pari al numero di porte logiche per ogni cella full adder della catena.

Uno dei limiti di questo sommatore sta proprio nel fatto che, a causa della propagazione dei riporti, si genera un ritardo non indifferente.

Puó essere usato anche come sottrattore, considerando l’operazione A-B come A+(-B) dove – B non è altro che il complemento a 2 di B che si ottiene facendo il complemento diminuito di B e aggiungendo 1. Per questo motivo la struttura sarebbe la stessa ma il riporto entrante alla prima cella sarebbe 1 e ci sarebbe una porta NOT in corrispondenza di ogni bit proveniente dal secondo operando.

**Come si realizza un sommatore/sottrattore? (NON RCA BASED)**

Sfruttando la porta XOR è possibile realizzare un sommatore/sottrattore, una macchina che puó funzionare in entrambi i modi all’occorrenza dell’utente. La porta XOR fornisce in uscita un valore negato se l’input viene posto in XOR con 1 (notY= Y XOR 1), viceversa fornisce l’input stesso se viene posto in XOR con 0. Sará quindi necessario far si che in entrata alla XOR vi sia un operando ed un segnale di substract che, nel momento in cui sará alto fará si che la XOR produca il negato dell’operando. L’uscita della XOR, insieme al secondo operando, entra in un classico RCA nel quale entra anche il segnale di subtract sotto forma di riporto (carry in ingresso) visto che quando si fa la sottrazione c’è bisogno di aggiungere 1 al complemento diminuito di un operando.

**Cos’è il sommatore Carry Look Ahead, come funziona e cosa cambia dall’RCA?**

Il Carry LA guarda in avanti, si propone di NON aspettare che avvenga tutta la propagazione classica dei riporti in un rca ma calcola tutti i riporti il prima possibile in modo da darli quanto prima insieme alle cifre da sommare ad ogni unitá di un rca.

A tal proposito si introducono le condizioni di propagazione e di generazione Pi = xi + yi e G^i = xiyi per cui posso scrivere, ricordando la regola del riporto C = XY + Z(X+oY) , il riporto in funzione rispettivamente di Propagate e Generate.

Si calcola il carry successivo-> Ci+1= xiyi + (xi+yi)= Gi+Pici dove ci

Ci = xi-1yi-1 +(xi-1+yi-1)ci-1 = gi-1 + pi-1ci-1

Da cui si ottiene Ci+1= xiyi + (xi+yi)= Gi+Pi(gi-1 + pi-1ci-1) = Gi + PiGi-1 + PiPi-1ci-1

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Sará quindi possibile scrivere tutti i riporti in qualsiasi stato in funzione di P e G che dipendono solo dagli addendi e da c0. Quindi mi calcolo prima tutti i G, tutti i P e tutti i ci . Alla fine mi basta sommare tutti i riporti e tutti gli addendi in full adder non connessi. Si calcolano generate e propagate usando una sola porta, quindi avremo 1∆, poi da Generate e Propagate si calcolano i carry, una OR e una AND quindi 2∆. Noti ii carry si calcolano le somme nei fa, 2 porte, quindi 1∆+2∆+2∆=5∆ ritardo complessivo.

L’area occupata sará pari a (n2+9n)/2. Il ritardo totale è sempre 5delta ma è ottimizzabile a 4∆ sostituendo i FA con porte XOR perché si puó scrivere la somma come S = x +o y +o Cin = P+o Cin e usando le XOR appunto al posto dei FA.

**Cos’è un sommatore Carry-Select?**

È un sommatore che si propone di risolvere il problema dell’rca della crescita della catena direttamente proporzionale all’aumento dei bit da sommare. L’idea di questo sommatore è quella di dividere la catena di full adder in P blocchi, ciascuno dei quali somma M bit. Il primo blocco sará un RCA di M bit che restituisce somma e riporto, i successivi saranno bocchi formati da 2 RCA che sommano GLI STESSI M BIT ma rispettivamente presentano riporto in ingresso pari a 0 e pari a 1. In questo modo non è necessario attendere che il riporto si propaghi attraverso ogni singolo adder ma abbiamo giá pronti i risultati corretti da selezionare per ogni blocco. Il ritardo totale risulta essere notevolmente ridotto perché si passa da un problema di propagazione sequenziale (in cui i ritardi si sommano) ad uno di propagazione parallela in cui il ritardo è DOMINATO dal blocco piú lento della catena + un piccolo overhead dovuto al MUX che andrá ad effettuare la selezione tra i due RCA di un blocco. Infatti, si definiscono TFA e TMUX i tempi di propagazione dei FA e del MUX:

Il carry in uscita dal primo blocco è disponibile al tempo M\*TFA perché è lungo M.

Somme e carry all’uscita del secondo saranno disponibili dopo M TFA+ TMUX perché c’è stata una selezione. Quindi alla ine avremo un tempo totale t= M TFA + (P-1)TMUX

**Cos’è un sommatore Carry save?**

Nasce per tentare di migliorare i tempi di propagazione ma in realtá risulta molto utile quando bisogna sommare piú operandi. Se quindi vogliamo effettuare S = x+y+z, se volessimo realizzare questa operazione con un classico rca per sommare le prime due stringhe e poi sommare la terza con il risultato avremo un ritardo significativo perché i riporti si devono propagare attraverso due sommatori.

Il carry save invece somma allo stesso tempo i bit corrispondenti dei 3 operandi producendo una stringa di somma e una stringa di riporto che andrá sommata alle cifre successive. La prima parte viene fatta da un full adder (cs). Da tale rete escono le due stringhe che dovranno essere sommate con un RCA. La particolarità sta nel fatto che i blocchi carry save operano in parallelo perché non sono soggetti alla propagazione del riporto.

**Differenza tra moltiplicatori paralleli e seriali?**

// se moltiplico una stringa di n bit con una di m bit l’uscita sará di n+m bit  
// La moltiplicazione di interi SENZA SEGNO avviene in decimale come in binario, stesso metodo.

I moltiplicatori paralleli effettuano il prodotto dele cifre + la somma delle righe e sono di tipo multiply and accumulate, quelli seriali sono basati su codifiche come Booth e derivano da una procedura manuale Robertson.

**Fammi un esempio di moltiplicatore Parallelo**

In un moltiplicatore parallelo l’operazione di moltiplicazione si divide in due fasi: una prima fase in cui si determina la matrice dei prodotti parziali utilizzando opportune AN; una seconda fase in cui avviene la somma dei riporti parziali, realizzabile con diversi approcci.

Un esempio è il prodotto somma per righe (ripple carry multiplication) in cui dopo aver calcolato la matrice dei prodotti parziali si calcola il prodotto come somma delle n righe della matrice con n-1 RCA. Le prime due righe della matrice vengono sommate con due HA e due FA, il primo e l’ultimo stadio invece presentano dei HA perché presentano solo entrate.

L’uscita dal primo RCA deve essere sommata alla terza riga, l’uscita dal secondo deve essere sommata alla quarta e ultima riga. . La struttura cosìi gestita porta problemi di propagazione di ritardo simili a quelli del singolo RCA a causa della propagazione dei riporti. Gli adders piú vicini agli operandi forniscono un risultato dopo 3T perché il primo T viene usato per il calcolo dei prodotti parziali e i successivi 2 T per il calcolo nel FA o HA. Il ritardo complessivo del moltiplicatore è 17T assumendo che il ritardo per ogni adder sia 1t dopo la disponibilità degli operandi. Possiamo calcolare l istante di tempo in cui sono disponibili le due uscite del FA come tcout,s=max(ta,tb,tcin) +1. Dato che i riporti si propagano tutti nella stessa direzione, i tempi richiesti per la moltiplicazione dipendono linearmente dal numero di bit dei fattori (i ritardi sono quindi circa doppi di quelli necessari per la somma).

Un altro esempio è il moltiplicatore che fa il prodotto come somma per diagonali sfruttando una struttura di tipo carry save in cui il riporto non viene piú propagato sulla riga ma in diagonale. Ha una struttura regolare e ben integrabile. Guardando una matrice dei prodotti parziali, tutte le righe tranne l’ultima non propagano il riporto ma lo mandano ogni volta allo stadio successivo. Quindi tutti i FA dello stesso livello daranno i risultati dopo lo stesso tempo, i ritardi che devono aspettare i blocchi del livello successivo saranno sempre gli stessi. In genere questo circuito è piú veloce del primo.

La differenza tra le due architetture sta nel fatto che nella seconda ci sono n sommatori, 1 per ogni riga e che questa risulta essere di tipo carry save. Il costo per le due è lo stesso ma la seconda viene eseguita piú rapidamente: 3 righe di 4 HA/FA contro 4 righe di 3 HA/FA.

**Cos’è un moltiplicatore a celle MAC?**

È un modello di moltiplicatore che si propone di unificare le due fasi di una moltiplicazione parallela.

Una cella MAC è composta da una porta AND per il prodotto parziale tra i bit iesimi dei due operandi e da un FA per la somma del riporto parziale con i riporti e le somme provenienti dai livelli adiacenti. È un’architettura regolare ed iterativa, simile a quella del prodotto come somma per diagonali. Ogni blocco condivide un termine con la stessa riga e un termine con la stessa diagonale.

Fondamentalmente non si ha un grande risparmio di tempistiche ma c’è un guadagno in termini di stabilità, riproducibilità ed interattività del circuito.

**Ma cos’è questo algoritmo manuale di cui si parla e dal quale deriva Robertson?**

// L’ALGORITMO MANUALE NON FUNZIONA PER NUMERI INTERI RELATIVI

È fondamentalmente il procedimento di base della moltiplicazione tra due numeri binari in colonna dove ad ogni passo devo shiftare il prodotto di j posizioni a sinistra per poter concorrere al calcolo del prodotto finale attraverso la somma di tutti i prodotti j-esimi. P=Sumj=0->n-1 xj2^jY. Quel 2 alla j sta ad indicare lo shift di j posizioni. Volendo si potrebbe implementare anche con un MUX che seleziona i risultati in base a 0 e 1 ed uno shift register che shifta ad ogni colpo di clock.

Esiste un modo peró per evitare di dover memorizzare tutti i prodotti Pj per la somma finale: ad ogni passo si puó calcolare una somma parziale dei prodotti pari a Pi+1=Pi + xj2^iY, P0=0, CONVIENE AD OGNI PASSO SHIFTARE P VERSO DESTRA piuttosto che il prodotto parziale. Quindi invece di sommare tutti insieme ne sommo uno alla volta, sommando la nuova stringa ad ogni iterazione con la somma parziale. In questo modo l ‘algoritmo è iterativo ma per implementarlo avrei bisogno di un sommatore, di un moltiplicatore, di uno shift register. Notiamo che la porzione P interessata nella moltiplicazione non è l’intera stringa ma solo la metá dei bit ogni volta traslata di una posizione. Quindi shiftiamo P verso destra di 1 ad ogni passo.

Se i numeri sono codificati in complemento a due una soluzione semplice come idea sarebbe quella del negare tutti gli operandi negativi, effettuare una operazione unsigned sui numeri positivi risultanti e poi negare il risultato se necessario ma per fare ció servono altri cicli di clock. Sulla base di alcune proprietá della rappresentazione in complementi è possibile applicare uno schema usando una tecnica di moltiplicazione unsigned apportando dele modifiche all’algoritmo manuale:

* Se X>0 e Y>0 si effettua una moltiplicazione tra unsigned
* Se X>0 e Y<0 ogni volta che Y moltiplica xj=/0 il prodotto parziale sará negativo, quindi il bit piú significativo di A, registro in cui sto appoggiando il risultato, deve essere sempre 1. Si puó usare un latch in cui viene inserito 1 e che viene usato per alimentare lo shift register in tutte le iterazioni
* Se X<0 e Y>0 per i primi n-1 prodotti parziali, il res è positivo mentre per l’ennesimo è necessario un passo di correzione con la sottrazione A-M.
* Se X<0 e Y<0 come nel caso 2, il bit piú significativo diA è 0 finché xj=0 e diventa 1 quando moltiplico xj=1 per Y. Per l’ennesimo prodotto è necessario fare la correzione A-M.

**Come funziona il moltiplicatore di Robertson?**

Il moltiplicatore di Robertson contiene: un registro M per il moltiplicando Y, un registro Q che inizialmente carica X e funge come shift register, concatenato ad A che funge da accumulatore del risultato parziale dell’operazione. F un fliflop/latch in testa allo shift register che tiene traccia del segno durante la moltiplicazione. Un adder/sottrattore parallelo, una control unit, un mux che seleziona tra M e 0 a seconda del bit corrente di Q durante la moltiplicazione ed un contatore che tiene traccia del numero di iterazioni e passaggi effettuati. La porta XOR serve in quanto vi entra il segnale substract che esce dalla cu come abbiamo visto nl sottrattore sommatore all’inizio.

M viene caricato con Y, Q con X e A inizializzato a 0 mentre il counter viene impostato al numero di iterazioni che servono in base alla lunghezza dei bit di X e Y; PER OGNI BIT DI Q, partendo dal meno significativo, se il bit corrente è 1, Il sommatore fa M+A altrimenti se è 0, A resta invariato (questa cosa è gestita dal mux). Il latch viene messo ad 1 se il bit piú significativo di M e il bit corrente di Q sono entrambi 1 perché dobbiamo riflettere il valore negativo.

Dopo ogni addizione (o dopo una operazione “vuota” se Q0 era 0), A e Q vengono shiftati a destra di 1 ed il contatore viene incrementato. Infine, se il bit meno significativo di Q è 1, indicando che il numero originale era negativo, si fa A-M per correggere il risultato finale che, dopo l’ultimo shift a destra, si troverá nel registro concatenato A.Q.

**E invece come funziona il moltiplicatore di Booth? E qual’è la differenza con Robertson?**

Consente di velocizzare le moltiplicazioni disegnando la CU saltando la fase di somma e risparmiando cicli di clock. Il moltiplicatore di booth si basa sulla codifica di Booth: sia X un intero che viene rappresentato in complementi a due come xn-1xn-2…x0, definiamo y0=-x0, y1=-x1+x0, y2=-x2+x1 ecc. allora moltiplicando ciascun yi per 2^i corrispondente (y0 per 2^0) si ottiene una rappresentazione posizionale, che altro non è che una codifica diversa per il complemento a due, che usa quindi un alfabeto diverso, -1 0 1. È come se trasformassi il mio numero 1111111 in 000000-1 risparmiando tante operazioni aumentando gli zeri.

I lvettorepuó essere rappresentato in forma di Booth-1 semplicemente sostituendo ciascuna coppia di bit adiacenti di x con un valore comepreso tra -1 0 1 secondo la regola:

* 00/11 ->0
* 01->1
* 10->-1

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Fondamentalmente basta modificare il moltiplicatore di Robertson facendo in modo che dentro Q devo guardare non solo Q0 ma anche un altro bit, devo aggiungere un bit a destra di Q0, che sarebbe Q-1 che fará da bit numero 1 per vedere la coppia. Sulla base dell’analisi di questi due bit si decide se si deve sommare, sottrarre o solamente shiftare ed in base a ció si metterá substract pari a 0 o a 1. In piú non c’è piú il passo di correzione alla fine perché sommo o sottraggo quando mi serve.

**Cos’è l’handshaking?**

È un protocollo di comunicazione, cioè un insieme di regole che definiscono la comunicazione tra due o piú entitá. In particolare, l’handshaking è asincrono (non c’è un segnale di tempificazione ma la comunicazione è gestita dai segnali di controllo che le due entitá si scambiano) e prevede lo scambio di un certo numero di messaggi il cui numero minimo è pari a 2. In questo caso infatti si usa un segnale di strobe che viene inviato da A a B per segnalare al sistema B che puó leggere il dato sul bus e B risponde con ack che significa che ha letto.

Puó essere implementato anche a 3 vie, in modo piú completo: c’è un segnale di REQ che segnala a B che A vuole inviare un dato, B risponde con ACK che significa che è disponibile e dopo aver effettuato la sua elaborazione, B invia un altro segnale DONE che significa operazione completata/dato ricevuto. È utile questo modello quando B deve fare operazioni lunghe.

Poi un protocollo puó essere anche sincrono o semisincrono. In quest’ultimo caso A e B hanno stesso riferimento temporale ma in caso di perdita di sincronismo B puó inviare un segnale di stato che significa aspetta.

Inoltre, con un protocollo di HS Semplice le due entitá si mettono accordo sullo scambio di un dato ma A non è in grado di sapere quando B ha terminato eventualmente una elaborazione a partire da quel dato. Si potrebbe usare un terzo segnale DONE con il quale B segnalerebbe ad A che ha finito ma ci sarebbe da interlacciare DONE con un altro segnale per avere la sicurezza che tutto vada bene. Per evitare ció si potrebbe semplicemente usare il fronte di discesa di accepted (della coppia data ready e accepted) che viene mantenuto alto fintanto che B elabora e viene abbassato quando ha terminato. In questo modo A capisce senza che sia aggiunta un'altra coppia di segnali.

**Cos’è la timing analysis?**

Il risultato dell’applicazione dei timing constraint durante l’implementazione è tipicamente quello di posizionare la logica in blocchi quanto più possibile vicini in modo da usare il meno possibile le risorse di routing. Primary clock: definisce un riferimento temporale per il progetto e viene utilizzato dalla timing engine per derivare i requisiti del timing path e la relazione di fase con altri clock. Il delay associato ad un primary clock viene calcolato dal pin/porto sorgente sul quale il clock è definito ai clock pin delle celle sequenziali ai quali esso viene collegato, per cui è opportuno definire i primary clock su segnali che appartengono al boundary del design.Se nel comando create\_clock non si specificano sorgenti (pin, ports,nets) per il clock con il comando get\_ports allora viene generato un **virtual clock**. Un **generated clock** è un clock derivato da un altro clock esistente, chiamato master clock, che viene trasformato attraverso un blocco logico che opera un divisione/moltiplicazione di frequenza, uno shift di fase, un’inversione, una modifica nel duty cycle, o una combinazione delle precedenti (es. attraverso un Clock Modifying Block come MMCM o PLL sui device 7 series).

L’analisi della tempificazione può essere effettuata già a valle della sintesi, però in quel caso i valori di tempificazione ottenuti saranno delle stime abbastanza approssimate poiché i tempi legati al routing, che di solito valgono tra i 45% e il 65% del tempo totale, non sono ancora disponibili. Per verificare quale sia la frequenza massima di funzionamento (FMAX) di un design è possibile diminuire progressivamente la frequenza del clock del design ed eseguire una timing analysis finchè non si ottiene uno slack negativo (worst negative slack). La FMAX non è fornita esplicitamente nei report ma può essere stimata calcolando 1/(T –WNS) dove T è il periodo del clock target e WNS è il worst negative slack nel caso intra-clock. Per ottenere una stima dei ritardi di un design puramente combinatorio, occorre inserire dei registri clockati a monte e a valle del design, che rappresentano rispettivamente gli input e gli output della macchina implementata. Il tool di sintesi cercherà di ottimizzare il ritardo fra i due registri (rispettando gli eventuali constraint forniti).

**Cos’è il controllo cablato e qual è la differenza col controllo microprogrammato?**

Le FSM rappresentano il modo più naturale per implementare la control unit in un sistema digitale: ***gli stati della FSM*** rappresentano i **diversi stati del controllore**, le ***uscite*** della FSM in ogni stato rappresentano i **segnali di controllo forniti al data path** (in tal caso si parla di **controllo cablato**). Questo fondamentalmente rappresenta il CONTROLLO CABLATO.

quando si parla di controllo microprogrammato, l’unitá di controllo usa una MicroROM in cui sono contenute le PAROLE corrispondenti alle MICROISTRUZIONI da eseguire. Quindi in ogni momento la ROM riceve un ingresso dal Program Counter e fornisce in uscita la parola contenuta nella locazione corrispondente. Fondamentalmente le microistruzioni corrispondono agli stati della FSM a controllo cablato e ogni microistruzione deve avere i segnali da settare verso l’esterno e l’indirizzo della prossima microistruzione. Poi ci sará bisogno di un circuito esterno alla ROM per modificare il flusso di controllo e consentire i salti.

I due tipi di controllo sono impiegati nell’implementazione della Control Unit all ‘interno della CPU. Nel caso del controllo cablato, ogni istruzione ha il suo circuito dedicato all’interno della CU, per cui è facile, veloce ma costoso e difficile da estendere in quanto si occupa tanto spazio su silicio. L’hardware di controllo è quindi realizzato come un FSM che passa da uno stato all’altro per ogni colpo di clock in base al contenuto dell istruction register, condition code e segnali esterni. Ne sono un esempio le architetture RISC (Le architetture RISC (Reduced Instruction Set Computer) sono un tipo di architettura di processore progettata per massimizzare le prestazioni eseguendo un insieme relativamente semplice di istruzioni in modo efficiente.

Nel caso del controllo microprogrammato invece per ogni istruzione c’é un insieme di parole nella microROM e la cu deve generare la sequenza di segnali di controllo per eseguire in modo corretto le microistruzioni. Quindi per ogni istruzione macchina c’è una sequenza di control words (microistruzioni) che viene detta micro-routine. Tutte le micro-routines sono memorizzate in una control store, esempio motorola 68k o intel8080.

**Come funziona una CPU in controllo microprogrammato?**

Una CPU con questa architettura ha il compito di eseguire nella giusta sequenza i segnali di controllo per l’esecuzione delle istruzioni. La sequenza di controllo è memorizzata all’interno di una memoria detta CONTROL STORE, sotto forma di microistruzioni (control word). Un decodificatore di istruzioni si occuperá poi di generare le control word adoperando un microPC, gestito dal clock e che punta sempre alle control word da dare in output. Viene usato un generatore di indirizzi di partenza per individuare l’inizio di ciascuna sequenza di microistruzioni in base all’op code dell’istruzione macchina. In piú un meccanismo esterno si occupa di modificare il flusso di controllo (in caso di salti).

La microprogrammazione puó essere di due tipi, organizzata in due modi:

* Organizzazione orizzontale: si usa un bit per ogni segnale, la decodifica è semplice e le microistruzioni sono molto lunghe con pochi 1 attivi contemporaneamente
* Organizzazione verticale: si raggruppano tutti i segnali mutuamente esclusivi e ció richiede la presenza di hw aggiuntivo per i decodificatori che viene compensato al fatto che ci sará una memoria piú piccola. Si puó usare uno schema di codifica binaria con decodificatori semplici. Attenzione perché volendo si potrebbero raggruppare proprio tutti i segnali, non solo quelli mutuamente esclusivi ma in questo caso si avrebbero CW di lunghezza minima ma circuiti di decodifica piú complessi.

**Come è organizzata la control store e cos’è il bit-oring?**

La cs deve contenere le microroutine per eseguire tutte le istruzioni, per questo risulta avere dimensioni elevate. Generalmente viene organizzata in modo da trovare e scrivere una sola volta tutte le microistruzioni comuni a operazioni diverse. In tal caso:

* È possibile condividere le parti comuni semplificando il modo in cui viene calcolato il valore da inserire nel microPC scegliendo indirizzo delle microistruzioni in modo tale che la distanza tra due microistruzioni con parte in comune sia al piú di un bit (bit ORING). Se la prima microistruzione diversa è memorizzata alla locazione x60(%0110 0000) conviene memorizzare la prima microistruzione con parte in comune in una locazione vicina che differisca solo per un bit, ad esempio x64(%0110 0100). In questo modo, giunti alla x60, al verificarsi di una certa condizione una porta OR cambierá il terzo bit da destra e si procederá al salto ed esecuzione della x64;
* È possibile aggiungere ad ogni microistruzione un campo indirizzo che indica la locazione della successiva. Ció si traduce in un aumento della lunghezza delle control word e in un incremento delle performance.

**Cos’è un livello microarchitetturale e cos’è il IJVM?**

Il livello microarchitetturale fornisce una astrazione delle unitá funzionali coinvolte all’interno di una CPU e delle loro interazioni. La IJVM è un sottoinsieme della Java Virtual Machine che opera su numeri interi. La microarchitettura che permette di implementarla è detta MIC1 ed è stata inventata da Andrew Tanenbaum.

**Qual è la struttura base della IJVM?**

Allora partendo dalla memoria, questa è di 4 GB organizzati in locazioni di 32 bit(l’area method è indirizzata a byte), questa è composta da 4 aree:   
Una **Constant pool** caricata quando un programma viene caricato in memoria, contiene costanti, stringhe e puntatori e non puó essere sovrascritta, la base di questa area è puntata dal CPP.  
Una **Local Variable Frame** che contiene le variabili locali di un metodo ed è allocato nello stack, la base è puntata dal puntatore LV. Le variabili sono accedute fornendo lo spiazzamento rispetto a LV. Un **Operand Stack** attiguo al local variable frame, ha una dim massima precalcolata dal compilatore, puntata dal SP che punta la cima e non la base. Viene usato per memorizzare gli operandi durante l’esecuzione di un’espressione aritmetica.  
Una **Method Area**, una area che contiene i metodi ed è gestita come un array di byte, puntata dal puntatore PC che punta alla prossima istruzione da prelevare.  
Le istruzioni sono corte e con pochi operandi, stack based. Ogni istruzione ha un opcode e al piú **un operando** come una costante o un memory offset. Le istruzioni consentono di inserire, prelevare dallo stack sia operandi che word, eseguire salti incondizionati o condizionati, fare scambi o rimuovere dalla cima dello stack. Invocare e ritornare da una procedura.

**Come funziona il meccanismo di chiamata a metodo in IJVM?**

Nella IJVM è possibile invocare metodi solo definiti nello stesso oggetto. Il chiamante mette sullo stack un puntatore all’oggetto da chiamare (per convenzione a JVM), mette nello stack i parametri della procedura e chiama INVOKERITUAL che ha come operando la posizione nella constant pool dell’indirizzo di partenza della procedura da chiamare, definita nella method area. All’indirizzo corrispondente si trovano 4 byte speciali, i primi due con il numero di parametri del metodo con il quale si accederà a OBJREF, i successivi 2 che contendono la dimensione della local variable area del metodo per allocare spazio sullo stack per il metodo. Si calcola quindi la base del nuovo LV frame, si calcola l’indirizzo dello stack dove SALVARE IL VECCHIO PC detto link pointer e inserisce questo indirizzo nella locazione puntata da LV. Salva sullo stack il vecchio PC e il vecchio LV. Inserisce in PC l ‘indirizzo della prima istruzione util del metodo.

Il return dalla procedura funziona invece cosí: si accede al link pointer e si ottiene l’indirizzo sullo stack del vecchio PC e del vecchio LV, si copia il valore nella locazione puntata da LV, si ripristinano i valori di PC e LV.

**Cos’è il MIC-1? Come è organizzata?**

Il mic 1 è il livello microarchitetturale atto all’implementazione della IJVM, implementato da Andrew Tanenbaum. È una macchina stack based, non dispone di registri generali.

I bus sono realizzati in vhdl con un singolo process che gestisce la logica di reset e quella di scrittura con il bus C. A ciascun bus corrisponde un signal.

La **ALU**  ha 6 linee di controllo, F0 F1 per l’operazione, ENA ed ENAB per abilitare gli inputs A e B, INVA per invertire l’ingesso A, INC per incrementare di 1.  
2 segnali di uscita, N e Z che segnalano che l’operazione effettuava ha prodotto un risultato negativo o nullo.  
Una operazione su due operandi inizia caricando il primo da uno dei registri in un bus B, si seleziona la configurazione di controllo che pone in uscita B che viene poi copiato sul bus C e poi nel registro H (HOLDING, mantiene il primo operando dell’alu) . Si connette il secondo operando col bus B. Lo shift register ha due linee di controllo proprie, SLL8 e SRAI, la prima shifta l’input a sx di un byte e riempie con tutti 0, il secondo shifta l’input di un bit a dx e lascia invariato il piú significativo.

La comunicazione con la memoria avviene o tramite un porto a 32 bit, controllato da MAR/MDR o tramite un porto a 8 bit controllato da PC/MBR. Ne sono due perché MAR e PC indirizzano due parti diverse della memoria. Attenzione che nell ‘implementazione fisica la memoria è byte addressable quindi il contenuto di MAR deve essere aggiustato, i due bit meno significativi vengono messi a0, i 2 piú significativi sono ignorati, il primo viene connesso al terzo bit dell’address bus.

I **Registri** sono: CPP,LV,SP usati per la gestione della memoria, PC, MAR che contiene l’indirizzo della word da leggere/scrivere, MDR word letta/scritta in memoria, TOS top of stack, cioè il valore puntato da SP, OPC reg temporaneo usato dalle istruzioni, MBR memorizzai byte dell’istruzione man mano che viene interpretata.

Tutto il datapath viene controllato da **29 segnali** divisi in 5 tipi:

9segnali per scrivere dal bus C ai registri, 9 per scrivere dai registri al bus B, 8 per Alu e Shift reg (6+2 giá visti), 2 segnali per indicare il read/write per MAR/MDR, 2 segnali per indicare il memory fetch PC/MBR.

La IJVM allora ha 2 segnali ulteriori di controllo: NEXT, indirizzo della microistruzione seguente, e JAM, codici di condizione per i salti.

La combinazione di questi segnali andrá a formare una microistruzione che conterrá dei bit aggiuntivi per indicare la prossima microistruzione. Di base una microistruzione è formata da: ADDR(next\_address)+JAM(JAMC/N/Z)+ALU(i primi 4+2 dello shift)+C(9 segnali)+MEM+B. Queste microistruzioni sono implementabili attraverso un linguaggio chiamato **MAL micro assembly language** che altro non è che una notazione sostitutiva per poter scrivere in forma leggibile i segnali di controllo.

Per quanto riguarda invece la **control store** essa è formata da 512 parole di 36 bit. MPC micro program counter indica ad ogni passo di controllo quale microistruzione leggere ed è virtuale, MIR micro istruction register contiene la microistruzione letta dall’indirizzo puntato da MPC. L’indirizzo nella control store della prima microistruzione di una istruzione ISA corrisponde al valore binario dell’op code dell’istruzione stessa.

Dopo ciascuna istruzione viene eseguita una microistruzione speciale **main** che ha il compito di prelevare un byte dalla memoria istruzioni; saltare alla prossima microistruzione.

La **logica di salto** è implementata con delle assegnazioni su un virtual register e dei semplici signal che rimangono stabili fino al fronte di clock successivo. Si usa JMPC che consente di implementare in maniera efficiente un multiway-branch cioè quando è =1, gli 8 bit MENO SIGNIFICATIVI di next\_address sono bassi, quindi in MPC va il valore di MBR con eventualmente 1 in testa. Di fatto si va ad inserire il valore dell’op code della prima microistruzione de eseguire per l’istruzione in esame.

JAMN e JAMZ determinano i bit di MPC[0]:

Immagine che contiene testo, diagramma, Parallelo, Disegno tecnico

Descrizione generata automaticamenteSe entrambi=0, MBR[0]=ADDR[0]  
Se JAMN=0 e JAMZ=1 MBR[0]=ADDR[0]+Z  
Se JAMN=1 e JAMZ=0 MBR[0]=ADDR[0]+N  
Se JAMN=1 e JAMZ=1 MBR[0]=ADDR[0]+N+Z

La **tempificazione del datapath si basa su un clock asimmetrico con un breve impulso,** ogni ciclo di data path parte dal fronte di discesa ed è diviso i 4 sotto cicli:  
1. Il registro MIR viene caricato e i segnali di controllo si stabilizzano  
2. I segnali di controllo vengono propagati e il bus B viene caricato col valore selezionato  
3. La ALU e lo Shiftreg elaborano il risultato  
4. I segnali sul bus C e sul bus verso la moemoria si stabilizzano  
Sul fronte di salita dell’impulso avviene il caricamento dei regstri dal bus X, dei flip flop N e Z , MBD e MDR.

Quindi riassumendo, l’esecuzione di ogni istruzione ISA consiste in: lettura dell’istruzione dalla memoria, salto al microcodice dell’istruzione, esecuzione del microcodice, ritorno al microcodice per il caricamento della prossima istruzione.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, Parallelo

Descrizione generata automaticamente

Immagine che contiene testo, schermata, numero, software

Descrizione generata automaticamenteImmagine che contiene testo, diagramma, schermata, Parallelo

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Parallelo

Descrizione generata automaticamente

**Cos’è la comunicazione seriale? Come funziona la UART?**

La comunicazione seriale è un tipo di comunicazione che vede lo scambio di informazioni tra due dispositivi UN BIT ALLA VOLTA,risparmiando risorse nella trasmissione. Un protocollo di questo tipo puó essere sincrono o asincrono ed i dati possono essere trasmessi in diversi modi, **simplex** (linea monodirezionale 1 trasmette 1 riceve), **halfduplex** (linea bidirezionale ma in ogni istante 1 trasmette 1 riceve), **full duplex** (linea bidirezionale entrambi possono trasmettere e ricevere contemporaneamente). Il tutto ruota attorno a dei parametri fondamentali, **baud rate, lunghezza del frame, endiannes(ordine in cui vengono trasmessi i bit), sincronizzazione, controllo errore.**

Per mantenere la sincronizzazione, il ricevitore campiona il segnale in arrivo su RX ad una frequenza superiore (16x o 64x) rispetto a quella di invio. Appena riconosciuto lo start bit, il ricevitore si posiziona a centro bit «contando» un certo numero di fronti di clock; I successivi campionamenti vengono effettuati posizionandosi sempre a centro bit per minimizzare gli errori. Gli errori possono essere di **overrun (**il ricevente non è in grado di elaborare un dato in arrivo prima che ne sopraggiunga un altro**), framing (**non c’è un bit di stop all’intervallo previsto**), paritá (**il valore del bit di paritá calcolato sul frame in arrivo non corrisponde con quello specificato nel frame**).**

Possono essere generate delle interrupt per evitare che un microcontrollore dotato di uart resti in attesa senza poter fare altre operazioni.

Quando un dispositivo UART deve connettersi ad un microcontrollore, che tipicamente utilizza valori di tensione detti TTL (Transistor-transistor logic), in cui lo 0 logico corrisponde a 0V e l’1 logico al valore Vcc (fra 3.3V e 5V), dovrà anche esso utilizzare segnali TTL. Una diversa specifica per i segnali è invece rappresentata dallo standard RS232, per il quale lo 0 logico è rappresentato da segnali la cui tensione varia nel range [+3V;+15V] e l’1 logico è rappresentato da segnali la cui tensione varia nel range [-3V;-15V] -> questi range piú elevati consentono di tollerare meglio il rumore e di avere comunicazioni su distanza maggiori rispetto ai segnali TTL.

Lo **standard RS232** è uno dei più vecchi e popolari standard di comunicazione seriale utilizzati nei prodotti commerciali, introdotto per trasmettere caratteri (tipicamente ASCII) tra un Data Transmissione Equipment (DTE) e un Data Communication Equipment (DCE). Fu creato inizialmente per connettere tra loro un telescrivente con un dispositivo per la gestione della comunicazione, un modem. La codifica dei caratteri, il framing, l’ordine dei bit e i protocolli di rilevamento errori sono definiti dall’hardware della porta seriale, tipicamente costituito da una UART. Lo standard non prevede di collegare tra loro due interfacce ma esistono cavi specifici che collegano tutti i fili e creano una connessione detta **null modem**. Nel caso piú semplice si collegano solo TX RX e GROUND.

Di base una UART è composta da una sezione di trasmissione e una di ricezione, ciascuna con un registro per i dati, un contatore per scandire i bit del frame, un contatore per gestire l’invio dei singoli bit in base alla baud rate concordata e una CU.

**Come funziona un controllore del trasmettitore in una UART? Ed il ricevitore?**

Il trasmettitore rimane inattivo in idle finché WR non diventa alto, appena diventa alto si passa allo stato TRANSFER in cui viene dato il segnale di load allo shift register che verrá caricat con la stringa ‘I’ & parity\_bit & DBIN ‘0’. In questo stato viene resettato il counter che conta i bit da trasmettere. Nello stato SHIFT viene fornito il segnale di shift che consente l uscita dei singoli bit dal registro, ogni bit viene inviato con una determinata velocitá. Appena il contatore di bit trasmessi diventa 12 si ritorna in idle.

Per quanto riguarda il ricevitore invece: quando è inattivo RXD viene mantenuto alto, appena diventa baso si passa allo stato EightDelay in cui si permane per 8 impulsi di conteggio in modo da posizionarsi a centro bit, appena il conteggio arriva a 8 si passa allo stato WaitFor0 seguito dallo stato WaitFor1 che assicurano che la macchina a stati venga ritardata esattamente per un tempo sufficiente a leggere il segnale RXD nel mezzo della trasmissione successiva. Lo stato successivo GetData incrementa il contatore dati e fornisce il segnale di shift. Appena arriva a 10 (8dati1parita1stop) viene attivato CheckStop che abilitá il controller degli errori.

**Cos’è la USART?**

È un dispositivo che supporta la comunicazione SINCRONA oltre quella asincrona. Quella sincrona prevede che vi sia un clock condiviso tra ricevitore e trasmettitore e che i dati vengano inviati in blocchi, eliminati i bit start e stop e paritá per aumentare prestazioni ed efficienza. La **sincronizzazione** può essere fatta o solo all’inizio della trasmissione oppure dopo un numero predefinito di caratteri utilizzando una sequenza di caratteri speciali **(byte di SYNC)**.

**Cos’è una NxM crossbar?**

È una rete di interconnessione molti a molti in cui gli N input e gli M output sono organizzati a formare una matrice ed in cui la ciascuna connessione fra un input e un output è realizzata da un componente elementare detto switch. Una rete del genere si dice non bloccante poiché consente connessioni multiple verso destinazioni diverse. È un approccio che in genere è poco affidabile visto che la connessione dipende dallo switch che collega due nodi

Un’alterativa è il MUX-DEMUX singolo stadio: è come sopra ma la matrice crossbar è realizzata mediante un mux e un demux in cascata, ha un solo stadio e quindi è veloce. Se si vuole garantire la connessione simultanea tra nodi diversi l’hardware deve essere replicato

Una architettura alternativa più scalabile si basa sull’utilizzo di più stadi intermedi per stabilire la comunicazione tra i nodi (un messaggio inviato da A a B viene instradato attraverso altri nodi intermedi). Uno switch elementare è quello che interconnette due sorgenti X1 e X2 con due destinazioni Y1 e Y2 utilizzando un mux 2:1 e un demux. Utilizzando blocchi elementari come questo è possibile realizzare un’interconnessione fra tutte le coppie di nodi comunicanti. (SWITCH ELEMENTARE)

**Cos’è un OMEGA NETWORK?**

È una architettura di interconnessione multistadio introdotta da Lawrie nel 1985, consiste di log2N stadi identici che sfruttano un’interconnessione fra i nodi basata sul **perfect shuffling,** un algoritmo derivante dal gioco delle carte e dal mischiare. Tale algoritmo si puó sfruttare per determinare gli accoppiamenti dei nodi nei singoli stadi e ottimizzare i percorsi.

Immagine che contiene diagramma, testo, linea, Carattere

Descrizione generata automaticamente

Nel caso in cui ci siano 2 comunicazioni simultanee e i due percorsi sono disgiunti, non si verificheranno conflitti. Se invece vi fosse un percorso condiviso vi sarebbe un conflitto che andrebbe gestito opportunamente.

Prima di parlare di come gestire un conflitto dobbiamo introdurre le tecniche di instradamento in rete:

* la **store and forward** è il metodo tradizionale e prevede che il pacchetto inviato sia ricevuto e archiviato da uno switch prima di essere inoltrato al nodo successivo. È un metodo lento ma affidabile e permette di controllare che non ci siano errori.
* la tecnica **wormhole** prevede che un pacchetto sia diviso in segmenti piú piccoli trasmessi individualmente. I primi pezzi contengono le info di routing che vengono usate per aprire una rotta, un corridoio, un wormhole che poi viene usato da tutti gli altri pezzi del pacchetto.

Proprio nel caso della tecnica wormhole si possono distinguere 4 modi di gestione dei conflitti:

1. **Blocco**: si interrompe l’avanzare del wormhole finché non si liberano le risorse necessarie
2. **Perdita** di pacchetti: i frammenti che non possono essere propagati vengono distrutti
3. **ReInstradamento**: si stabilisce un percorso alternativo a quello bloccato
4. **CutThrough**: i segmenti che non possono essere inoltrati vengono bufferizzati finché c’é spazio