**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.

Di contro, 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 microistruzoni 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 modiicare 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. L'idea fondamentale dietro le architetture RISC è quella di semplificare il set di istruzioni del processore, eliminando istruzioni complesse a vantaggio di un set di istruzioni più piccolo e più semplice da eseguire.)

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?**

Allora una CPU con questa architettura deve 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 micro istruzioni (control word). Un decodificatore di istruzioni ha il compito di generare le control word usando un microPC gestito dal clock che punta ogni volta alle control word da dare in output, un generatore di indirizzi di partenza per individuare l’inizio di ciascuna sequenza in base al codice operativo dell’istruzione macchina e Un meccanismo per 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 compesnato al fatto che ci sará una memoria piú piccola. Si puó usare uno schema di codifica binaria con decodificatori semplici. Atteznione eprché volendo si potrebbero ragruppare proprio tutit i isegnali, non solo queli mutuamente escusivi 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 sará di dimensioni elevate. È comodo e necessario organizzare la memoria in modo da trovare e scrivere una unica volta tutte le microistruzioni comuni a operazioni diverse e ció si fa usando due approcci:

1. Condividere le parti comuni di istruzioni diverse semplificando il modo in cui viene calcolato il valore da inserire nel microPC ad esempio scegliendo gli indirizzi delle microistruzioni in modo che la distanza tra due microistruzioni con parti in comune sia di un solo bit. In tal caso si parla di bit-ORING: assumiamo d avere microistruzioni di 4byte e la cs byte addressable. 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 se si verifica una condizione specifica che triggera la seconda microistruzione, una volta arrivato alla x60 una porta OR cambia il bit 3 dell’indirizzo e si esegue quella.
2. Includere in ogni microistruzione un campo indirizzo che che indica la locazione della microistruzione successiva. Ció si traduce peró in un aumento della lunghezza delle control word anche se il sistema diventa piú performante.

**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?**

A differenza della JVM classica, 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 paramentri 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. Qui, 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, MBD 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 cotiene 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]:

Se 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.

Per eseguire una scrittura, l’indirizzo e il dato devono essere disponibili nei registri MAR e MDR al fronte di salita in cui è alto il segnale WE Write Enable. Quando si opera in lettura (WE basso), il dato è disponibile nel registro MDR al fronte di clock successivo rispetto a quello in cui l'indirizzo e posto nel registro MAR. Lettura e generazione del write enable sono implementate in un process

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, diagramma, Parallelo, Disegno tecnico

Descrizione generata automaticamente

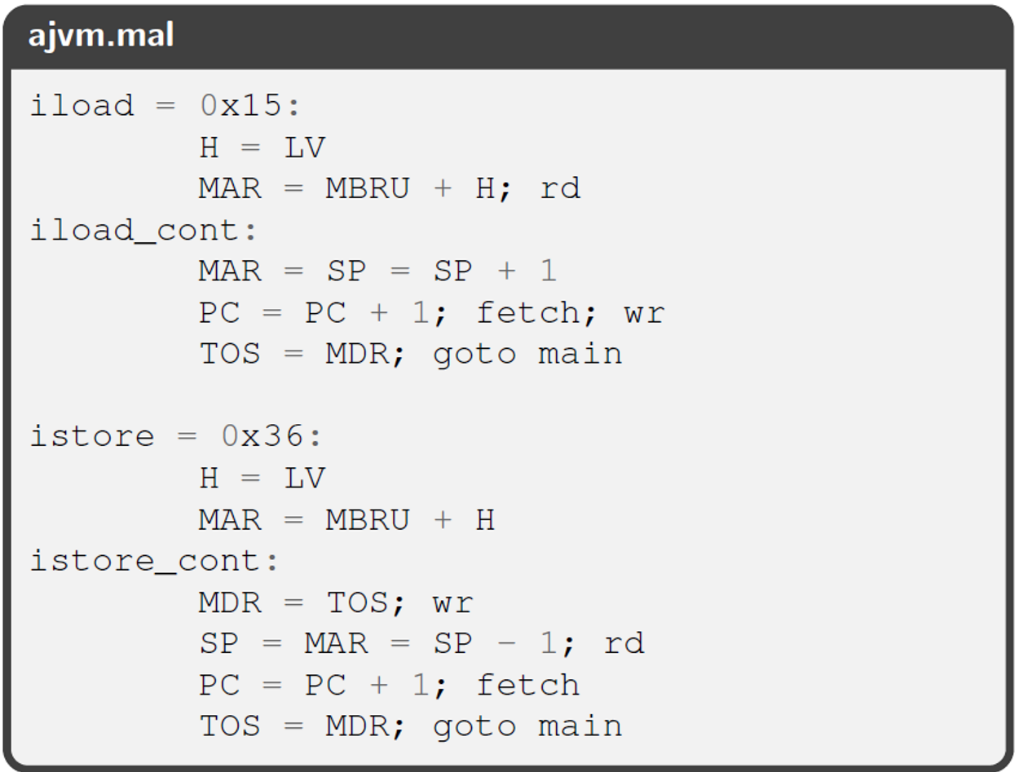
Immagine che contiene testo, schermata, Carattere, Parallelo

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma, Parallelo

Descrizione generata automaticamente

Immagine che contiene testo, diagramma, schermata, Parallelo

Descrizione generata automaticamente

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

Allora 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.**

La UART è acronimo di Universal Asynchronous Reciever Transmitter ed è un dispositivo hardware che supporta la comunicazione seriale asincrona. Consiste di due segnali TX e RX su cui viaggiano i dati trasmessi/ricevuti + un segnale di ground e prevede che la comunicazione avvenga secondo un protocollo che prevede una determinata struttura del frame trasmesso. La linea è mantenuta al valore logico alto (IDLE) fino all’inizio di una nuova comunicazione. La trasmissione inizia inviando uno start bit (valore logico basso). Successivamente vengono inviati i bit di dato (fra 5 e 9 bit). Opzionalmente viene inviato un bit di parità (parità pari o dispari in base alla configurazione) ed infine seguono 1 o 2 bit di stop (valore logico alto) per terminare la trasmissione del frame. **entrambi i dispositivi comunicanti devono essere configurati con la stessa baud rate.**

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 perane 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 macchin 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 argoritmo 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 sarebbe un percorso condiviso vi sarebbe un conflitto che andrebbe gestito opportunamente.

Prima di parlare di come getire un conflitto dobbiamo introdurre le tecniche di instradamneto 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 tecnicha wormhole si possono distinguere 4 modi di gestione dei conflitti:

1. Blocco: si interrompe l avanzare del wormhole finche 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