Figura X – Pinout entity Z80 ##DA SISTEMARE##

Per la mia implementazione dello Z80 ho tenuto conto delle limitazioni che l’uso dell’FPGA induce e non mi sono attenuto a rispettare tutte le informazioni sull’organizzazione interna poiché lo scopo era quello di ricreare da punto di vista comportamentale il microprocessore. Per analizzare l’implementazione farò riferimento all’entity in VHDL, a cui mi riferirò con Z80X, affrontando per prima cosa il port e poi addentrandomi nell’organizzazione interna.

Port

Il port di Z80X rispecchia per lo più quello originale aggiungendo alcuni segnali per adattare l’entity ad essere usata all’interno dell’FPGA.   
Per prima cosa, Z80X presenta due segnali di temporizzazione: CLK\_FPGA e CLK. Il primo è quello fornito dall’esterno all’FPGA ed è fisso a 50MHz mentre il secondo è generato per divisione dal primo all’interno del design e può variare da circa 4MHz a circa 50Hz. Di conseguenza i due sono in relazione di fase. Però CLK è il clock usato per temporizzare le azioni del microprocessore.  
Il vero segnale di temporizzazione dell’entity è CLK\_FPGA . Questa scelta è stata dettata da tre motivi principali:  
mantenere lo stesso clock su tutto il design all’interno dell’FPGA per non incorrere in problemi di metastabilità tra i diversi regimi di clock;  
poiché non è buona pratica fornire alle SLICEs un clock con frequenza variabile nel tempo, come lo è CLK, ho preferito temporizzare l’entity su CLK\_FPGA che invece è a frequenza costante;  
l’organizzazione reale dello Z80 richiederebbe dei latch e dei flip-flops triggerati su entrambi i fronti del segnale di temporizzazione, FFs double-edge. I primi è buona pratica non usarli all’interno dei design su FPGA mentre i secondi non sono realizzabili poiché i FFs presenti nell’FPGA rispondo a solo un fronte del segnale.  
Di conseguenza ho sfruttato il fronte di salita di CLK\_FPGA come evento di trigger di tutti i FFs. I latch li ho realizzati per mezzo di FFs di tipo D con un segnale di abilitazione solitamente denominato come LOAD. Per le temporizzazioni sui fronti del segnale CLK ho sfruttato dei rivelatori di fronte. Nel caso di fronti positivi attivano per un periodo di CLK\_FPGA il segnale CLK\_PEDGE, abbrev. di Clock Positive Edge, mentre sul fronte negativo attivano per lo stesso periodo il segnale CLK\_NEDGE, abbrev. di Clock Negative Edge. Il segnale CLK\_EDGE, che segnala l’avvenimento di un cambiamento su CLK, è ottenuto per mezzo di OR dei due precedenti segnali. Usando dei FFs con segnale di abilitazione collegato a CLK\_EDGE ho ottenuto lo stesso effetto dei FFs double-edge. Grazie alla grande differenza di frequenza, il ritardo indotto dalla temporizzazione su CLK\_FPGA è trascurabile dal punto di vista di Z80X.

Ho separato il bus dati secondo il verso: ingresso, DIN, ed uscita, DOUT. In questo modo è più semplice gestire lo scambio di dati sul bus dati.  
Ho scelto di non usare i bus A e D e il gruppo System Control con la logica three-state ma piuttosto di fornire in uscita dei segnali che comunicano se quest’ultimi devono essere considerati in alta impedenza. I segnali sono A\_HZ per A, DOUT\_HZ per DOUT e CTRL\_HZ per il gruppo System Control. Si possono usare questi segnali come segnali di selezione per dei multiplexer o come abilitazioni per porte three-state.

Organizzazione

Figura XX – Diagramma dell’organizzazione dello Z80X che mostra i collegamenti principali ##DA SISTEMARE##

Nel diagramma di Figura XX ho riassunto l’organizzazione interna dello Z80X. Ho omesso tutti i segnali di temporizzazione assieme a tutti i segnali di controllo in ingresso ed uscita da Z80X, quest’ultimi perché vengono gestiti e generati dalla sezione di controllo. Inoltre tutti i segnali di controllo delle unità presenti sono tutti generati dallo stesso gruppo di controllo. Durante la seguente analisi prederò in esame le principali differenze con le informazioni sull’organizzazione dello Z80.

Per prima cosa ho deciso di usare due bus interi e separati a 8 e 16 bit per i dati e gli indirizzi. Nel diagramma questi bus sono segnati differentemente dalle altre connessioni. Il motivo è che le connessioni in ingresso a questi bus sono gestite dal gruppo di controllo, che in base allo stato e all’istruzione sceglie quale segnale instradare sul bus. Per cui i segnali che vanno verso il bus in realtà passano prima per la sezione di controllo.  
Ho deciso di separare i registri A ed F dal blocco di registri REGS per gestire più facilmente il passaggio dei flag e dell’accumulatore. Il blocco REGS, che contiene B, C, D, E, H, L, W e Z, si occupa di mantenere le due pagine di registri e permette la contemporanea lettura di un singolo registro a 8 bit, una coppia a 16 bit e una scrittura. Però con questo metodo non si riesce ad accedere velocemente a registri non in relazione tra loro.  
Per questo anche IX ed IY non sono insieme agli altri. Inoltre ho aggiunto un sommatore dedicato che serve per generare gli indirizzi in caso di indirizzamento indicizzato. Allo stesso modo SP è isolato con un inc/dec dedicati per facilitare l’aggiornamento in caso di operazioni di PUSH e POP.  
I registri I ed R sono separati. I è solo mentre R ha un incrementer dedicato.  
Ci sono tre diversi latch per i bus dati e indirizzi. A\_LATCH e DOUT\_LATCH servono a mantenere stabile il valore sui bus d’uscita. Invece MDR, abbrev. di Memory Data Register, mantiene l’ultimo valore letto durante un’operazione di lettura e non fetch.

Il registro IR, abbrev. di Instruction Register, è in realtà uno shift register SIPO. Viene resettato all’inizio dell’operazione di opcode fetch del primo byte e caricato con il valore presente su DIN. Per caricarlo si effettua un’operazione di scorrimento. Nel caso in cui si legga un prefisso e serva il resto dell’opcode, basta semplicemente fare il fetch della seconda parte e caricarlo normalmente, così l’istruzione con il suo prefisso sarà presente all’uscita del registro.

L’ALU presenta dei registri interni che campionano il valore dell’accumulatore, del secondo operando e del registro di stato F. Questi registri vengono aggiornati quando CLK\_NEDGE è attivo e sull’uscita il risultato è sempre presente anche se solitamente viene campionato con CLK\_PEDGE. Questo permette di usare i bus ad altri scopi. I latch dell’ALU per necessità di temporizzazione con altri eventi sono aggiornati anche da STATE\_CNG che è un segnale generato dalla sezione di controllo.

Per semplicità nella gestione degli operandi durante l’esecuzione delle istruzioni INC e DEC, sono presenti due unità dedicate INCDEC e INCDEC16 rispettivamente a 8 e 16 bit.

Sezione di controllo

Nella sezione di controllo ho raggruppato tutte le unità che servono a scandire il ritmo delle operazioni svolte dalla CPU, a gestire gli interrupt e generare i segnali di controllo per le altre unità. In particolare ci sono i generatori dei segnali CLK\_PEDGE, CLK\_NEDGE e CLK\_EDGE, usati nell’intero circuito in sostituzione del riconoscimento dei corrispondenti fronti di CLK, e i multiplexer per i bus dati interni.

Come nell’organizzazione dello Z80 sono presenti tre FFs di tipo Set-Reset con i segnali di set collegati rispettivamente a nINT, nNMI e nBUSREQ. Le uscite dei FFs vengono usate per entrare nei rispettivi cicli di servizio. Durante questi cicli i FFs vengono resettati con un impulso per cui se vi è una richiesta ancora pendente, questa viene comunque letta al prossimo ciclo.  
Affianco a questi FFs ve ne sono altri due chiamati NMIIF, abbrev. di NMI Instruction Flag, e INTIF, abbrev. di INT Instruction Flag. Questi vengono settati durante l’inizio delle corrispettive routine di servizio per segnalare al decoder che dev’essere eseguita una speciale istruzione per la gestione dei rispettivi interrupt. Alla loro conclusione vengono resettati.

Come da architettura ci sono due coppie di FFs per immagazzinare i valori di IFF e IMF che sono collegati ai due bit meno significativi del bus interno a 8 bit.

Il controllo dei processi e la decodifica ho deciso di farla sempre del tipo cablato alla cui base però non c’è un semplice contatore di M-cycles e T-cycles. Siccome ho notato una regolarità nell’esecuzione delle istruzioni ho creare un sistema con macchine a stati finiti, abbrev. in FSMs, Finite States Machines, annidate.

Le macchine a stati finiti sono automi caratterizzati da un numero limitato di possibili stati assumibili. La macchina può passare dallo stato in cui si trova in un altro solo al verificarsi di condizioni note a priori e nel delle FSMs in esame lo stato di arrivo è sempre determinato, macchine deterministiche. Per cui il comportamento di una FSM è sempre completamente descrivibile per mezzo del suo diagramma degli stati che esprime quali sono gli stati e le condizioni per il passaggio da uno ad un altro.  
Data la forma, nelle FSMs si riconoscono tre elementi principali:  
un registro di stato, che mantiene lo stato attuale e lo aggiorna all’arrivo di un evento di temporizzazione;  
una logica combinatoria che in base agli ingressi e allo stato corrente calcola lo stato successivo che viene dato al registro di stato;  
una logica combinatoria che genera le uscite.

Vi è una distinzione in base alla tipologia dell’ultima logica. Se questa usa solo lo stato corrente come ingresso si dice che la FSM è una macchina di Moore altrimenti se usa anche gli ingressi è una macchina di Mealy.  
Le macchine di Mealy hanno tendenzialmente meno stati rispetto alla controparte di Moore e rispondono più velocemente alle variazioni degli ingressi poiché non vi è in ritardo dovuto all’attesa del segnale di temporizzazione per cambiare di stato. D’altra parte per le macchine di Moore è più facile controllare il comportamento in base al valore delle uscite.

Per scelta progettuale ho deciso di usare solo macchine di Moore per la facilità di verifica e il ritardo dovuto all’attesa del segnale di temporizzazione è trascurabile. Questo perché tutte le azioni che compie il microprocessore non dipendono istantaneamente dagli ingressi che invece vengono campionati in momenti precisi e di conseguenza in stati adeguati.

La struttura a FSMs annidate consiste in più livelli di macchine che ne controllano altre più piccole e che svolgono funzioni più semplici e ripetute varie volte. Questa struttura l’ho elaborata facendo queste osservazioni:  
la struttura e la temporizzazione delle azioni di opcode fetch, memory R/W, I/O R/W, interrupt acknowledge, halt e reset sono sempre uguali e ripetute più volte in ordine diverso in base alla necessità. Di conseguenza è verosimile siano implementate per mezzo di singole macchine;  
le durate delle istruzioni sono sempre di minimo 1 M-cycle e 4 T-cycles corrispondente alla fase di fetch-decode, per ogni operazione sulla memoria si aggiunge 1 M-cycle e 3 T-cycles, per ogni operazione sugli I/O si aggiuge 1 M-cycle e 4 T-cycles. Per cui l’ordine e la quantità di operazioni viene gestita da una macchina apposita;  
in caso di interrupt solo il primo M-cycle è di interrupt acknowledge qualsiasi sia l’interrupt e la modalità;  
l’ingresso nello stato di attesa per HALT o bus request è gestito come un ciclo assestante.

Un esempio della corrispondenza tra durata in M/T-cycles e le operazioni svolte è l’istruzione LD IX, (nn). L’istruzione legge dalla locazione punta da nn un valore a 16 bit che viene caricato in IX. È composta da 4 byte: un suffiso, l’opcode e due byte che contegono parte bassa e alta dell’indirizzo nn. Sul datasheet è riportata una durata di 6 M-cycles per un totale di 20 T-cycles.  
Ipotizzando la possibile esecuzione dell’istruzione si vede che devono essere eseguite due fasi di fetch, una per il suffiso e una per l’opcode, per un totale di 2 M-cycles e 8 T-cycles. Queste sono seguite da due letture da memoria per le due parti dell’indirizzo che occupano in totale 2 M-cycles e 6 T-cycles. L’istruzione termina con la lettura della parte alta e bassa del dato direttamente caricato in IX occupando 2 M-cycles e 6 T-cycles. Il totale è di 6 M-cycles e 20 T-cycles come riportato sul datasheet.

La struttura di FSMs che ne segue è la seguente: una FSM Master che in caso di bus request blocca il funzionamento delle FSMs sottostanti; una FSM Principale che gestisce i cicli per gli interrupt, il reset, l’halt e la normale esecuzione controllando la partenza delle FSM sottostanti; quattro FSMs secondarie che svolgono ognuna un ciclo preciso e vengono fatte partire da un segnale di trigger dato dalla FSM Principale.

D’aiuto alla FSM Principale ci sono:  
un contatore di periodi di reset, che conta per quanti cicli di CLK consecutivi il segnale nRESET è attivato e se supera la soglia dei tre cicli avverta la FSM. Tiene conto anche per quanti cicli nRESET non è attivo e lo comunica alla FSM;  
un contatore di cicli di clock di esecuzione, che conta i cicli di CLK in cui la FSM rimane in uno stato di pura esecuzione in cui non esegue nessuna operazione sulla memoria o sugli I/O;  
un rivelatore di ciclo macchina finale, che segnala alzando un flag, chiamato M\_LAST, che la FSM ha raggiunto l’ultimo ciclo macchina dell’istruzione;  
un rivelatore di transizione negativa di M\_LAST, che segnala per un ciclo di CLK che è appena finita un’istruzione e da il tempismo alla FSM Principale per servire gli interrupt;  
un riconoscitore di prefissi, che avverte la FSM Principale per mezzo del segnale EXT\_IR se è necessario fare il fetch del secondo byte come opcode.