

UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Implementazione della CPU Z80 su FPGA: un primo approccio

Relatore:
Prof. Daniele Vogrig

Laureando:
Giacomo Biasin
1216276

ANNO ACCADEMICO 2021/2022

Data di laurea 21/09/2022

Indice

Capitolo 1 Introduzione	2
Capitolo 2 VHDL e FPGA.....	4
1. VHDL: Un linguaggio di descrizione hardware.....	4
2. FPGA e la scheda di sviluppo usata	9
Capitolo 3 Lo Z80.....	13
1. Generalità e storia del microprocessore Z80	13
2. Architettura dello Z80	16
3. Informazioni sull'organizzazione dello Z80.....	30
4. Implementazione del microprocessore Z80 su FPGA.....	36
A. Implementazione del ciclo principale.....	43
B. Implementazione delle sottomacchine	52
C. ALU e incrementers	59
D. Registri.....	62
E. Problemi noti.....	64
Capitolo 4 Memorie, interfacce verso l'esterno ed il controllore	65
1. Bus ed interfacce a registri	68
2. Memorie	69
3. Display a 7 segmenti	75
4. Controllore via UART.....	77
A. Controllore e interfaccia verso il PC	77
B. Generatore di clock variabile	87
C. Z80 snapper e Z80 reporter	87
Capitolo 5 Conclusione.....	89
Capitolo 6 Indice analitico	90
Capitolo 7 Indice delle figure	93
Capitolo 8 Bibliografia	95

Capitolo 1 Introduzione

Lo scopo di quest'elaborato è la descrizione del progetto in cui ho sviluppato e implementato su FPGA una versione HDL della CPU Z80 assieme ad un sistema che ne permettesse l'utilizzo come scheda di sviluppo. Lo scopo dell'implementazione della scheda di sviluppo era quello di creare un home computer facilmente programmabile e utilizzabile con la possibilità di espandersi senza difficoltà. La scelta della CPU Z80 deriva dalla storia degli home computer iniziata negli anni '80 e che vide questa CPU come una delle più usate [1] ed assieme alla famiglia di CPU derivate è usata ancora oggi, come nella calcolatrice TI-84 Plus [2].

La scelta dell'implementazione in HDL su FPGA dell'home computer deriva dal voler avere un ambiente facilmente modificabile e flessibile. L'uso di questa tecnologia permette di sviluppare più velocemente, rispetto alla realizzazione con i componenti discreti, l'intero sistema e di poterlo modificare semplicemente cambiando la descrizione hardware. L'unica difficoltà in più rispetto alla creazione con i componenti discreti era la necessità di sviluppare autonomamente la CPU Z80.

Per cui il primo passo del progetto è stato lo studio della CPU e la sua implementazione. Mi sono attenuto al comportamento descritto nei manuali e ho cercato di rispecchiare il suo comportamento visto dai pin cioè non attenendomi alla reale organizzazione interna che non è completamente compatibile con la realizzazione su FPGA.

Lo sviluppo del sistema che ne fa da contorno ha seguito due temi principali.

Il primo era la creazione di un ambiente in cui lo Z80 potesse funzionare. Per cui serviva l'implementazione in HDL delle memorie necessarie al funzionamento e di un output, compatibile con lo Z80, per farlo comunicare con l'esterno attraverso un display a sette segmenti.

Il secondo tema era il debug dell'intero ambiente. Per cui serviva un sistema che facesse un salvataggio dello stato degli ingressi e delle uscite del microprocessore e li rendesse disponibili ad un PC esterno. In aggiunta serviva poter controllare oltre che la velocità di esecuzione, quindi la frequenza del clock, anche il flusso del programma caricato nelle memorie avanzando per un numero preciso di cicli di clock o di istruzioni allo stesso modo di come si fa il debug passo-passo.

Nel corso dell'elaborato distinguerò tra architettura e organizzazione dei sistemi che analizzerò. Con la prima mi riferirò a tutti gli aspetti visibili al programmatore mentre con la seconda alle parti operative e alle loro interconnessioni per sviluppare l'architettura [3].

Ho deciso di organizzare quest'elaborato allo stesso modo in cui ho organizzato il progetto. Nel primo capitolo, *VHDL e FPGA*, fornisco delle informazioni utili per i capitoli successivi. Nella prima parte fornisco una breve visione del linguaggio HDL usato, allo scopo di introdurre termini e concetti che richiamerò nei capitoli successivi. Nella seconda parte, descrivo l'hardware usato e le scelte di implementazione che ho attuato per adattarsi a questo. Nel secondo capitolo, *Lo Z80*, descrivo lo studio e l'implementazione della CPU Z80. Parto dalle informazioni generali e per arrivare alla mia implementazione, tenendo conto dei risvolti che le scelte di progetto hanno portato.

Nel terzo capitolo, *Memorie, interfacce verso l'esterno e controllore*, descrivo la progettazione e l'implementazione della scheda di sviluppo associata allo Z80 e del controllore.

Nel capitolo conclusivo, analizzo i problemi incontrati e i possibili miglioramenti futuri.

Capitolo 2 VHDL e FPGA

1. VHDL: Un linguaggio di descrizione hardware

Lo Z80 venne rilasciato sul mercato nel 1976. È formato da circa 10 000 transistor [4] e quando venne progettato fu disegnato interamente a mano [5]. Confrontando questo numero di transistor con quello dei 20 miliardi del recente Apple M2 [6], risulta impensabile che non ci sia stato un cambiamento nella tecnologia di progettazione.

Con la crescita esponenziale della complessità dei sistemi e l'avvento della VLSI (Very Large Scale Integration) divenne indispensabile l'uso di programmi che assistano il progettista detti CAD (Computer-Aided Design tools). Questi programmi svolgono tre fasi principali: l'acquisizione di un design (capture), la sintesi (synthesis) e la conseguente verifica (verification) [3]. Però per fare tutto questo serve un mezzo per poter comunicare al programma l'organizzazione del design. A questo scopo sono stati creati i linguaggi di descrizione hardware, abbreviato HDLs (Hardware Description Languages). Con questi linguaggi si descrive per mezzo di una sintassi codificata il comportamento di un circuito logico possibilmente a tre diversi livelli di astrazione: strutturale (Structural), che descrive esplicitamente i collegamenti con gli elementi hardware base; register-transfer level RTL, che nasconde i dettagli tecnologici rispetto allo strutturale; comportamentale (Behavioural), in cui

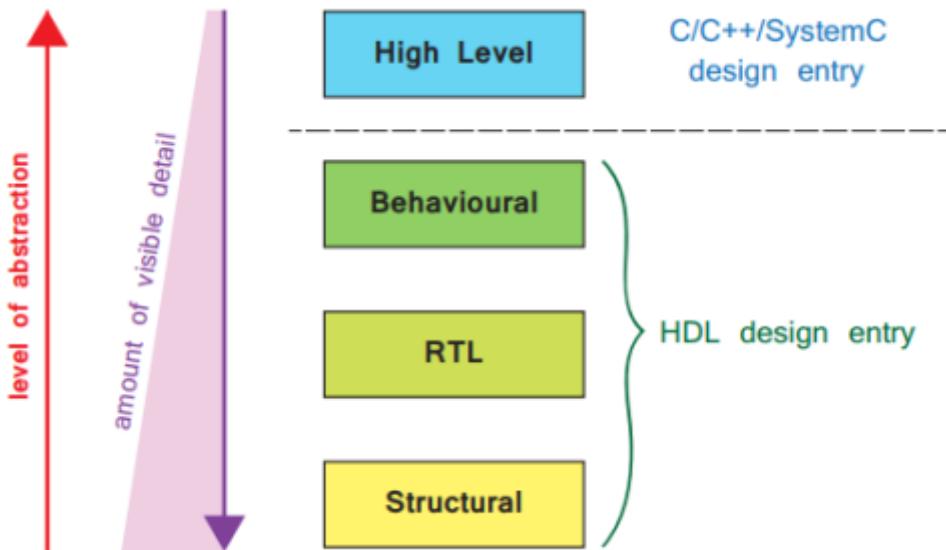


FIGURA 2-1 - LIVELLI DI ASTRAZIONE RELATIVI AGLI HDL [7]

si descrive il comportamento ai pin del circuito e si lascia al tool di sintesi il compito di definire l'hardware necessario.

Il VHDL è assieme al Verilog il linguaggio HDL più usato che permette di descrivere un circuito digitale. Questi circuiti vengono chiamati design entities, o più brevemente entities. Questa è composta da due parti l'entity declaration, in cui si definisce l'interfaccia dell'entity con il mondo esterno, e l'architecture body, che definisce le operazioni interne dell'entity e cioè l'implementazione della stessa, come mostrato nella Figura 2-2 - Esempio di design entity VHDL [7].

L'entity declaration è anche detta port poiché l'elenco di segnali di connessione è preceduta da questa parola chiave. Nell'entity declaration si può aggiungere anche un altro campo detto generic. In questo campo si possono definire delle variabili che generalizzano l'entity così da slegare il design da particolarità modificabili come la dimensione. Per esempio si può voler descrivere un sommatore e lasciare generico il numero di bit. Usando il campo generic, si definisce una variabile che rappresenta il numero di bit che verrà usata per definire i segnali del port e l'implementazione. Però questa rimane non definita sino all'istanziazione del componente, detta instantiation, in un entity di ordine superiore. Con questo vantaggio del VHDL si può riutilizzare la stessa entity in condizioni diverse.

All'interno dell'architecture body, detta anche architecture, possono essere definiti dei segnali, signals, utilizzati per collegare più parti interne. Queste possono essere a loro volta istanze di altre entities. I segnali possono rappresentare tipi elementari, come singoli bit, vettori di bit ordinati che rappresentano i bus, numeri interi oppure tipi più strutturati come array o record.

```

entity <entity_name> is
    port( <port_declarations> );
end <entity_name>

architecture <implementation_name> of <entity_name> is
    <component_declarations>
    <internal_signal_declarations>
begin
    <concurrent_statements>
end <implementation_name>;

```

FIGURA 2-2 - ESEMPIO DI DESIGN ENTITY VHDL [7]

L'assegnazione di valori ai segnali avviene in maniera simultanea per mezzo di assegnazione diretta, concurrent assignment, o condizionale, conditional assignment. Insieme definiscono l'elemento base dell'architecture, il concurrent statement. Per cui nel VHDL, a differenza di altri linguaggi di programmazione procedurale, tutte le entity e le assegnazioni simultanee sono attive ed eseguite allo stesso tempo.

Un tipo particolare di concurrent statement è il costrutto process. Questo costrutto può racchiudere delle assegnazioni o altri costrutti tipici dei linguaggi di programmazione con if-

then-else, while loop e for loop per svolgere operazioni più complesse. Un process viene attivato quando uno dei segnali definiti nella lista di segnali che segue la parola chiave, detta sensitivity list, cambia di stato cioè avviene un evento.

Il VHDL può essere usato sia per descrivere entities sintetizzabili, che si traducono poi in un vero e proprio circuito, o entities che fanno da banchi di prova per testare il funzionamento di altre entities. In questo caso, ci si riferisce all'entity che attua il test come testbench e alla testata come DUT (Device Under Test) per mantenere un'analogia con il mondo fisico. Nel testbench vengono definiti dei process con particolari funzioni che sfruttano l'esecuzione sequenziale e ciclica del codice presente all'interno svolto comunque in parallelo agli altri concurrent statements. Per cui ci sono solitamente dei process per generare dei segnali di temporizzazione per mezzo di direttive wait for ed altri process che generano gli stimoli da dare al DUT per verificarne il funzionamento. Per cui il VHDL e i linguaggi HDL più in generale permettono sia di sintetizzare che creare dei sistemi per simulare il comportamento di altri dispositivi.

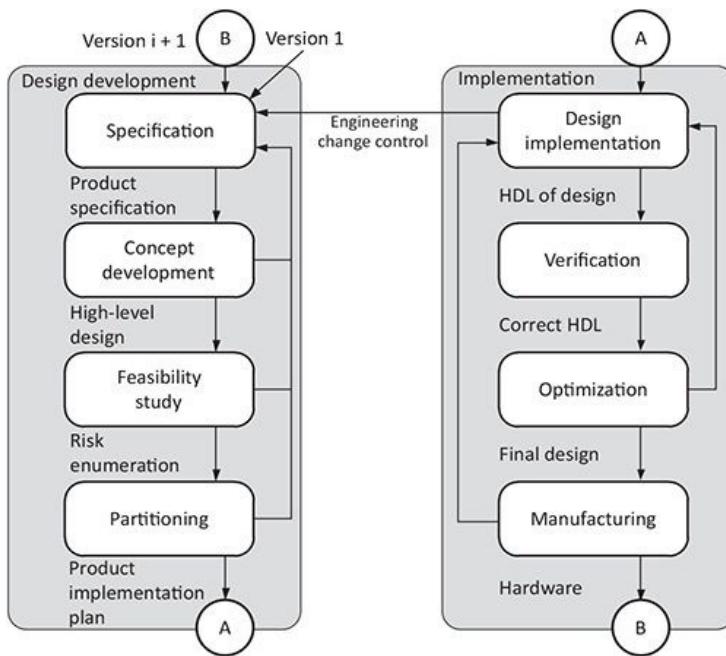


FIGURA 2-3 - PROCESSO ITERATIVO DI PROGETTAZIONE [3]

Avendo la possibilità di descrivere un sistema per mezzo di un CAD adatto, si può sviluppare interamente un sistema. Si può seguire un processo iterativo, come mostrato nella Figura 2-3 - Processo iterativo di progettazione , che partendo dalle specifiche si arriva alla descrizione del comportamento dell'entity, partizionarla in blocchi elementari più piccoli se necessario. Dopodiché si possono simulare le entities per mezzo di testbech dedicati e verificarne il funzionamento. Inoltre si può verificare il rispetto delle specifiche elencate e la conseguente ottimizzazione o approvazione. In questo modo si può progettare velocemente un sistema

arrivando all’implementazione fisica che è la parte più onerosa anche rispetto al tempo solo alla fine [3].

Uno svantaggio dell’uso di linguaggi HDL a livello comportamentale è che il tool di sintesi usa librerie standard di componenti che possono non essere la soluzione più ottimizzata per implementare efficientemente il design [3].

Per questo progetto ho adottato esattamente questo metodo. Partendo dall’esterno della scheda verso l’interno ho definito le specifiche dell’entity dove necessario. Nel caso dello Z80, le specifiche erano già definite nel datasheet. Dopodiché ho ricorsivamente diviso le entities in altre più piccole e semplici. Di seguito ho fatto l’implementazione del design in VHDL e la conseguente simulazione di verifica seguendo la metodologia iterativa. Solo nel momento in cui arrivavo a un sistema funzionante sulla simulazione lo sintetizzavo sull’FPGA e ne facevo una verifica sulla scheda poiché quest’ultimo passaggio richiedeva la maggior parte del tempo.

Un altro grande vantaggio dell’uso dei linguaggi HDL è la portabilità. Un’entity definita per un determinato ambiente può essere trasportata senza troppe difficoltà in un altro. Il tool di sintesi si occuperà di adattare la tecnologia al comportamento richiesto. Per far ciò serve uno standard di riferimento del linguaggio e di conseguenza ne vennero pubblicati vari. Tra questi il primo è IEEE Std 1076 del 1987, che definisce il linguaggio, e rinnovato secondo le nuove versioni, la cui più recente è VHDL-2008. Un altro standard importante è IEEE 1164 del 1993 in cui vengono standardizzati i valori logici usati e le conseguenti operazioni logiche per permettere una sintesi e simulazione più dettagliata.

Per poter usare l’FPGA su cui ho implementato la scheda di sviluppo, ho usato il CAD fornito dal produttore dell’FPGA, Xilinx. L’azienda mette a disposizione varie versioni dell’ambiente di sviluppo in base all’FPGA usata. Nel mio caso utilizzavo una Spartan 6 che permetteva solo l’uso di ISE 14.7 con VHDL-93. Quest’ambiente mette a disposizione oltre ai tool di sintesi anche un simulatore, un programmatore per l’FPGA e un analizzatore logico integrato detto ILA. Quest’ultimo se istanziato nel progetto, invia al PC lo stato dei segnali collegati all’ILA attraverso lo stesso protocollo di programmazione.

Lo stesso tool mette a disposizione un’interfaccia dedicata per inserire nel progetto degli IPs, chiamati IP cores. Gli IPs, abbreviazione di Intellectual Properties, sono entity che come mattoncini possono essere uniti al progetto principale per svolgere delle funzioni specifiche. Gli IPs coprono un vasto numero di funzioni: possono descrivere memorie, interfacce verso protocolli di comunicazione o svolgere compiti particolari come algoritmi di compressione.

Gli IPs vengono usati come black boxes facendo riferimento solo al loro port e al datasheet allegato. Un esempio noto di IP sono i core ARM che vengono autorizzati per essere usati nei vari design. [9]

Nel progetto ho fatto uso di alcuni IPs, messi a disposizione dal tool, per usare delle special features dell'FPGA quali Block RAMs e FIFOs e di un IP open-source per la gestione dell'interfaccia UART.

2.FPGA e la scheda di sviluppo usata

Una delle fasi del processo di progettazione è quella di scegliere quali parti comprare sul mercato già pronte e quali invece progettare autonomamente. In quel caso bisogna scegliere anche il mezzo su cui implementare il design. Nel caso di circuiti che richiedono funzionalità particolari e nel caso di elevati volumi di produzione si può creare un circuito integrato personalizzato detto ASIC (Application Specific Integrated Circuit). Le ASICs portano il vantaggio di poter scegliere autonomamente la tecnologia raggiungendo un'efficienza elevata. D'altra parte richiedono un corposo investimento per iniziare la produzione. Una via di mezzo tra l'uso di parti presenti sul mercato e quello delle ASICs è dato dalle FPGAs (Field-Programmable Gate Arrays) o più in generale dai dispositivi logici programmabili PLDs (Programmable Logic Devices). [3]

I PLDs in generale sono dispositivi logici regolari e riconfigurabili. A differenza degli altri dispositivi logici, al momento della produzione non hanno una funzione logica predefinita. Questi dispositivi si possono dividere in tre categorie di complessità crescente: simple programmable logic devices SPLDs, complex programmable logic devices CPLDs ed FPGAs. I SPLDs non differiscono molto da una ROM. In quel caso, in ogni riga sono codificate le uscite della funzione logica e le righe sono selezionate dalla giustapposizione degli ingressi della funzione. Dei SPLD, sono noti i PLAs (Programmable Logic Arrays) in cui la funzione logica è pressoché implementata per mezzo dei suoi mintermini ed ha una struttura come quella in Figura 2-4 - Esempio di PLA .

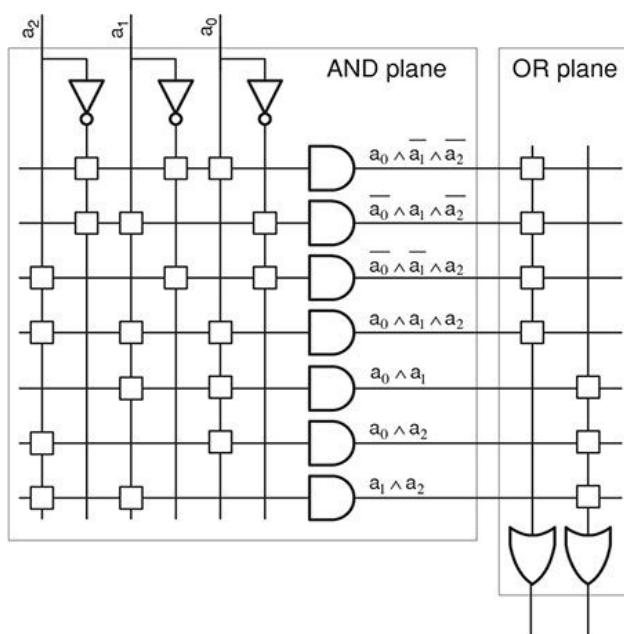
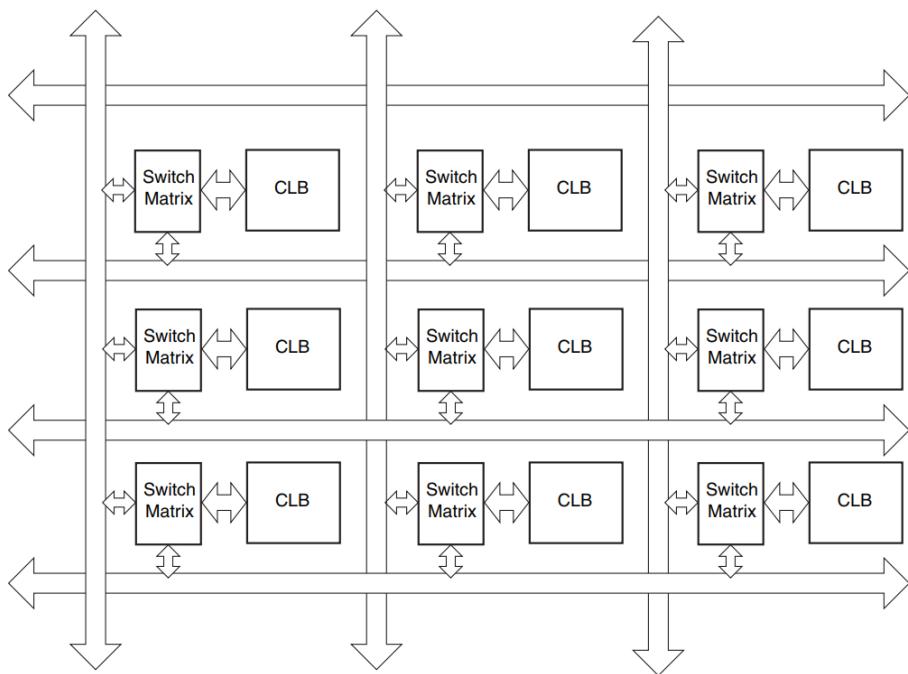


FIGURA 2-4 - ESEMPIO DI PLA [9]

Le FPGAs invece presentano uno schema a blocchi regolare. Al suo interno sono presenti dei blocchi logici configurabili CLBs (Configurable Logic Blocks). Quest'ultimi implementano le funzioni logiche per mezzo di look-up tables, LUTs, che sono generalmente delle memorie, ed altre funzioni aggiuntive come sommatori, multiplexer o stadi flip-flops, FFs. Oltre a questi blocchi ce ne sono altri con funzioni specifiche, a cui Xilinx si riferisce come special features, ad esempio memorie, gestione della temporizzazione o interfacce con l'esterno. Tutti questi blocchi sono disposti a matrice e collegati a dei bus bidirezionali per mezzo di connessioni programmabili, come si vede in Figura 2-5 - Schema dei CLBs e dei canali d'interconnessione . La struttura regolare e i collegamenti selezionabili delle FPGAs permettono di implementare un ampio numero di design andando a collegare e configurare questi blocchi elementari. Inoltre sono facilmente programmabili e riconfigurabili più volte in poco tempo.



ug384_29_012710

FIGURA 2-5 - SCHEMA DEI CLBs E DEI CANALI D'INTERCONNESSIONE [11]

Il CAD dopo aver svolto la sintesi del design si occupa di tradurre (translate) i componenti acquisiti con quelli di cui l'FPGA è costituita. Dopodiché si occupa di associarli (map) con quelli effettivamente presenti sulla FPGA ed infine di piazzarli e collegarli (place & route) creando il design richiesto. In questi passaggi si può andare incontro ad errori che possono condizionare la scelta dell'FPGA. Per esempio, si può scegliere un'FPGA con delle RAM già presenti per risparmiare CLBs od un'altra con un numero maggiore di CLBs poiché il design non riesce a essere associato correttamente.

Per svolgere il progetto ho usato la scheda di sviluppo AX309 prodotta da ALINX. Questa scheda presenta un ampio numero di periferiche già collegate. Tra queste, quelle di maggiore importanza, poiché sono state usate nel progetto, sono: il display da sei cifre a 7 segmenti, controllato per mezzo di un unico bus per i segmenti ed un bus per gli anodi; un’interfaccia USB-UART per comunicare con la scheda attraverso la micro USB di alimentazione; 4 LED e una porta JTAG. La porta JTAG serve per la programmazione dell’FPGA attraverso un programmatore compatibile ma anche per il debug attraverso le comunicazioni di un’ILA, se istanziata.



FIGURA 2-6 - SCHEDA DI SVILUPPO AX309

La scheda monta un FPGA Xilinx XC6LSX9 con clock a 50MHz. Quest’FPGA memorizza la configurazione su una memoria volatile. Per cui sulla scheda è presente una memoria FLASH, programmabile via JTAG, per immagazzinare la configurazione e ricaricarla ad ogni riavvio. L’FPGA presenta uno schema regolare dei CLBs che sono organizzati in colonne, come mostrato in Figura 2-5 - Schema dei CLBs e dei canali d’interconnessione . Ogni CLB è diviso internamente in due slices che non comunicano tra loro ma solo con i collegamenti al bus centrale. Queste possono essere di tre tipi: SLICEX, basilari che contengono quattro 6-Inputs LUTs e 8 FFs con clock ed enable comuni; SLICEL, che in aggiunta alle precedenti hanno la possibilità di gestire i riporti provenienti da altre slices; SLICEM, che presentano elementi di memoria per creare RAM distribuite e shift registers. Le slice nei CLBs seguono la seguente distribuzione: una slice di ogni CLBs è sempre una SLICEX mentre l’altra è alternata tra le altre due. Quest’ultime sono messe in colonna per permettere la propagazione del riporto. [10]

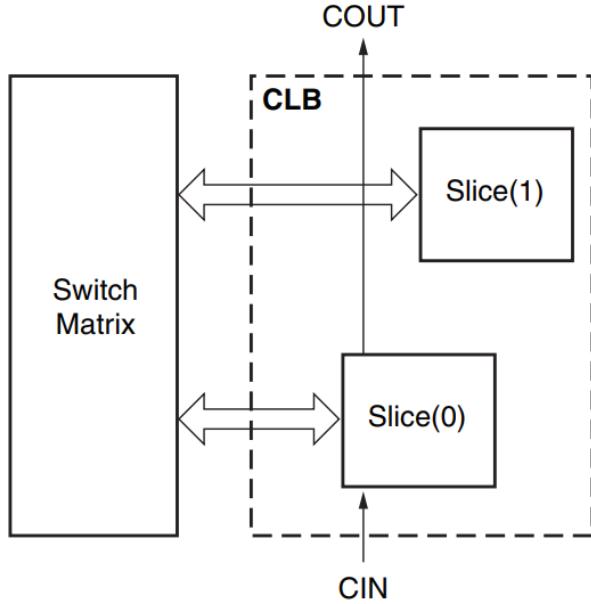


FIGURA 2-7 – DISPOSIZIONE DELLE SLICES ALL’INTERNO DI UN CLB [10]

Grazie agli elementi di memoria nelle SLICEMs, si possono creare delle RAM distribuite su più CLBs, anche dual-port, che hanno il vantaggio di essere facilmente utilizzabili e con lunghezze di parola arbitraria. La limitazione è nel numero poiché si possono implementare al massimo 90kb [10]. Per avere a disposizione più memoria, l’FPGA contiene già delle RAM, dette Block RAMs (BRAMs), che permettono un indirizzamento attraverso due port. L’FPGA che ho usato mette a disposizione 32 BRAMs da 18kb usabili come 2x9kb per un totale di 576kb [11]. L’uso di RAM dual-port permette di collegare due regimi a clock diversi all’interno del componente o, come nel mio caso, per permettere l’uso delle memorie sia da parte dello Z80 che del controllore.

Le BRAMs e le RAM distribuite sono utilizzabili nel progetto per mezzo di appositi IPs messi a disposizione dall’ambiente di sviluppo. Lo stesso ambiente mette a disposizione degli IPs che implementano delle FIFOs (First-In First-Out). Quest’ultime sono delle code che permettono di immagazzinare dati da un regime più veloce e renderli disponibili a uno più lento. Queste pile si basano su memorie RAM e l’ambiente permette di scegliere tra distribuite e BRAMs.

Nel progetto ho usato due BRAMs per implementare la ROM e la RAM collegate allo Z80, entrambe da 2kB per un totale di 32kb. Ho scelto di usare delle BRAMs per lasciare liberi i CLBs per altri scopi. Anche le FIFOs che ho usato le ho tutte implementate su BRAMs per lo stesso motivo.

Capitolo 3 Lo Z80

1. Generalità e storia del microprocessore Z80

Lo Z80 è un microprocessore a 8 bit prodotto dalla Zilog, con sede a Los Altos, California, e lanciata sul mercato nel 1976. La storia di questo microprocessore è strettamente legata a quella del primo microprocessore, l'Intel 4004. Come punto di partenza basti notare che entrambi i chip hanno due persone in comune: Federico Faggin e Masatoshi Shima.

Federico Faggin, fisico vicentino, laureato all'Università degli studi di Padova, fu a capo del "Progetto Busicom" che portò alla nascita nel 1971 del primo microprocessore ad Intel [12]. Nell'aprile del 1969, l'azienda giapponese di calcolatrici, Busicom, contattò Intel per produrre dei chip personalizzati per realizzare un set di calcolatrici senza dover modificare sostanzialmente l'hardware per ognuna [13]. Intel consigliò di ridurre il numero di chip concentrando la CPU tutta in un unico integrato. Però l'azienda americana, al tempo produceva per lo più memorie [12] e decise di assumere Federico Faggin nel novembre 1970. Faggin negli anni precedenti lavorava presso la Fairchild Semiconductor, azienda da cui provenivano i fondatori di Intel. Lì aveva sviluppato e perfezionato la tecnologia MOS Silicon Gate Transistor [14], indispensabile per produrre un circuito complesso come la CPU richiesta dalla Busicom e che gli diede notorietà. L'azienda giapponese alla fine commissionò 4 chip, che l'Intel chiamò famiglia 4000: 4001, una ROM; 4003, una RAM; 4002, uno shift register SIPO (Serial-In Parallel-Out) ed infine 4004, la CPU a 4 bit.

Faggin aveva il compito di progettare i quattro chip e di dirigere il progetto. A causa dell'impegno che si era assunta Intel di sviluppare lei la CPU, dovette anche occuparsi dell'organizzazione del 4004 e della metodologia di produzione, nuova per l'azienda [15]. L'azienda giapponese inviò poco dopo ad Intel Masatoshi Shima, l'ingegnere che era a capo della progettazione dell'organizzazione dei chip, per controllare l'avanzamento dei lavori [16]. A causa del ritardo nello sviluppo dei chip, Shima si offrì di aiutare Faggin per completare l'organizzazione della CPU mentre l'italiano già iniziava il layout del chip interamente a mano su tavole di mylar assieme ad altri tecnici [15]. Faggin firmò con le proprie iniziali i die come segno della sua opera.

Lavorando incessantemente, il primo wafer del 4004 funzionò nel gennaio 1971 e nacque così il primo microprocessore della storia. L'intera famiglia di componenti sviluppati per la Busicom venne commercializzata liberamente, senza più l'esclusiva dell'azienda giapponese, ed aprì la strada ai microprocessori. Così l'Intel produsse in rapida battuta nel 1972 l'Intel

8008, successore a 8 bit del 4004 e capostipite della famiglia x86 [17], e l'Intel 8080 che ebbe un successo immediato nel 1974 [13]. Faggin fece da capo ad entrambi i progetti dei due chip.

A novembre 1974, Faggin si licenziò dall'Intel per aprire, assieme ad un ingegnere suo collega, la propria azienda, la Zilog. L'azienda aveva lo scopo di produrre solo microprocessori, in contrasto con gli ideali del tempo di Intel. Il duo assunse Shima nell'aprile del 1975 per progettare l'organizzazione del microprocessore [5]. Questo fu disegna interamente a mano e in particolare per due terzi dallo stesso Faggin che ci lavorò incessantemente per tre mesi e mezzo.

Lo Z80 fu pronto per il mercato nel maggio del 1976 e venne pubblicizzato con lo slogan "The battle of the 80's" ("La battaglia degli 80"), Figura 3-1 - Pubblicità dello Z80 del maggio 1976.



FIGURA 3-1 - PUBBLICITÀ DELLO Z80 DEL MAGGIO 1976

Lo slogan richiamava la vicinanza dello Z80 con l'Intel 8080 sfruttando il gioco di parole con i suffissi dei nomi dei due chip. Lo Z80 fu progettato con lo scopo di essere compatibile con il mondo che si era creato attorno alla commercializzazione del microprocessore di Intel migliorando tutte le pecche del predecessore. Questo fu possibile perché entrambi vennero progettati dalla stessa persona.

Lo Z80 è costruito con tecnologia depletion MOS per cui richiede una sola alimentazione a 5V rispetto all'Intel 8080 che realizzato con enhancement MOS [18] richiedeva una triplice alimentazione +12V, +5V e -5V. Inoltre richiedeva due segnali di temporizzazione alla tensione +12V che non si sovrapponevano, come si vede nella Figura 3-2 - Forme d'onda degli

ingressi e delle uscite dell'Intel 8080° [19]. Lo Z80 invece richiedeva solo un clock a 5V. Un altro vantaggio dello Z80 erano gli mnemonici. La Zilog studiò attentamente gli mnemonici per la programmazione, in maniera tale che dessero più informazioni su cosa facessero, rendendo più comprensibile il listato [19]. Si fece anche attenzione a mantenere il set di istruzioni compatibile al codice binario con quello sviluppato per l'Intel 8080 permettendo di far girare il sistema operativo CP/M.

Un punto di forza considerevole per lo Z80 è la simbiosi con una famiglia di periferiche dedicate. Faggin progettò lo Z80 pensando già di integrarlo con almeno 4 periferiche che si protesero collegare senza logica aggiuntiva. Tra queste ci sono: Z80-DMA, il Direct Memory Access; Z80-PIO, interfaccia con due port parallel programmabili; Z80-SIO, interfaccia con due porte seriali full-duplex; Z80-CTC, modulo timer a tre canali per gestire la temporizzazione dei processi.

Per cui un acquirente che al tempo avesse voluto passare al microprocessore Z80 avrebbe avuto la possibilità di acquistare dallo stesso fornitore già un sistema funzionante. Inoltre lo Z80 presenta una gestione degli interrupt dedicata e ottimizzata per queste periferiche.

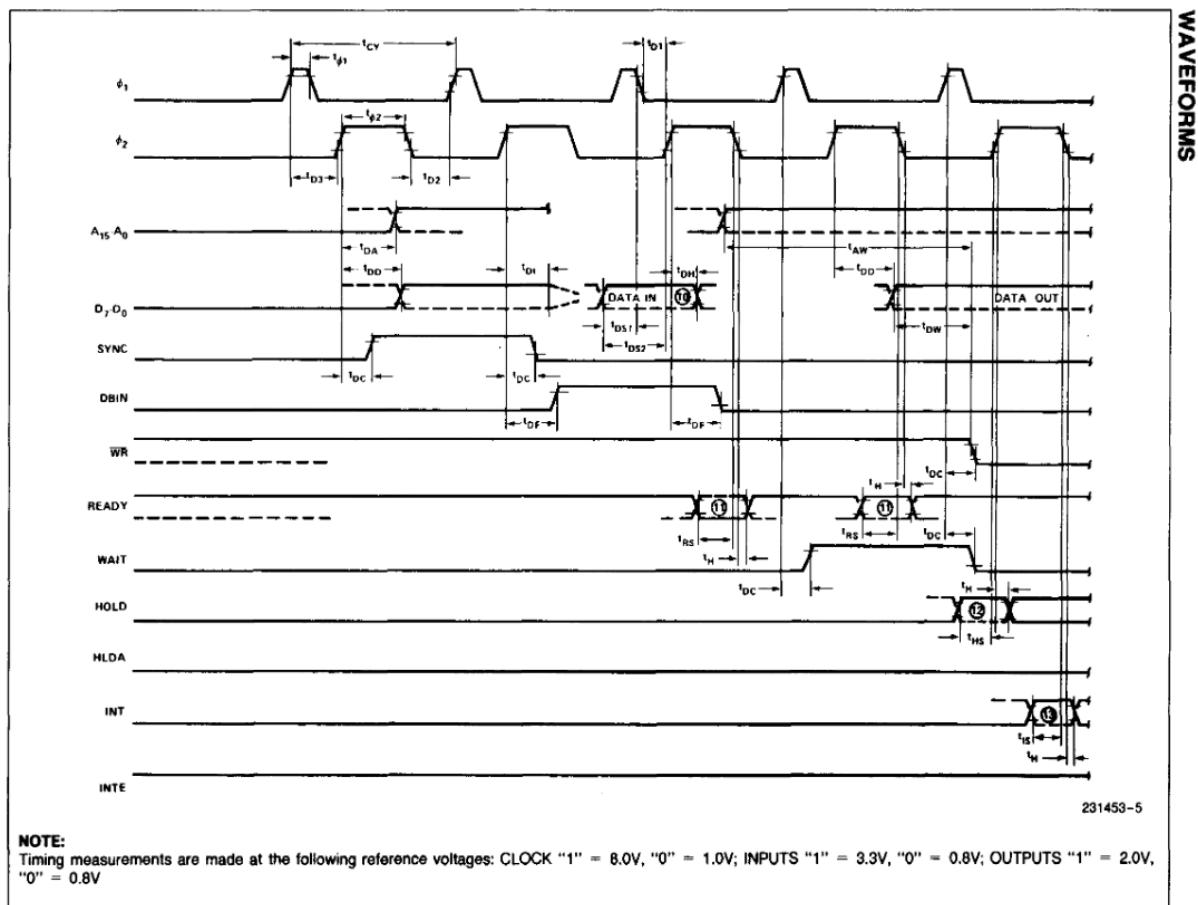


FIGURA 3-2 - FORME D'ONDA DEGLI INGRESSI E DELLE USCITE DELL'INTEL 8080° [20]

2. Architettura dello Z80

Per descrivere lo Z80 faccio uso della distinzione tra architettura e organizzazione.

Con architettura ci si riferisce all'insieme di attributi e risorse visibili al programmatore [20] o che hanno un impatto decisivo sull'esecuzione del programma [21].

Mentre con organizzazione ci si riferisce alle effettive unità operative e alle loro connessioni per sviluppare l'architettura [21].

A questo punto, si può analizzare lo Z80 secondo quattro elementi strettamente connessi tra loro: la piedinatura, o pinout; le risorse visibili al programmatore, detta per sineddoche architettura; la gestione degli eventi di interrupt; il set di istruzioni. Inizialmente farò riferimento a quanto riportato sul datasheet [22] e il manuale del microprocessore [23].

Pinout

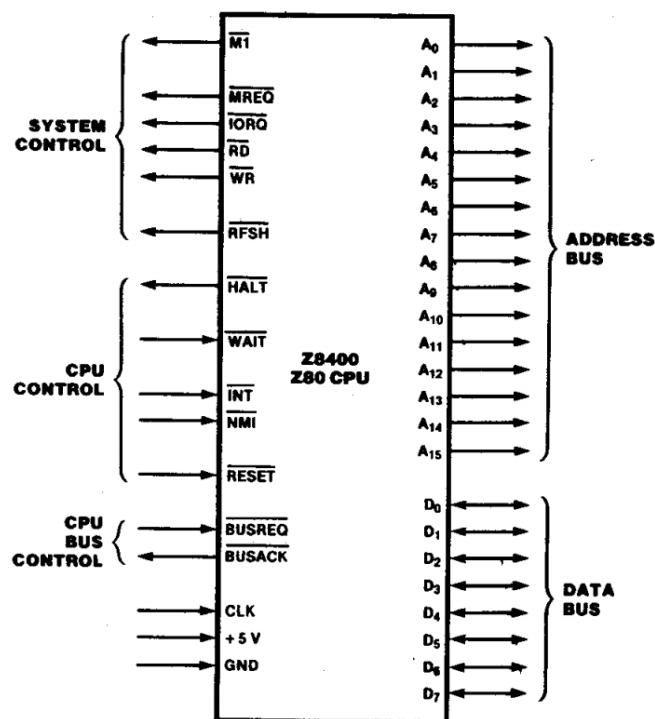


FIGURA 3-3 - PINOUT DELLO Z80 [22]

Lo Z80 ha la possibilità di indirizzare 64kB di memoria o registri di interfaccia con le periferiche di input e output (I/O). Usa un bus a 16 bit denominato address bus A che ha la possibilità di andare in alta impedenza (tecnologia three-state) per permettere l'uso dello stesso bus ad altri microprocessori o ad un DMA. Per comunicare con la memoria e gli altri dispositivi usa un bus bidirezionale dati da 8 bit detto Data Bus D, anch'esso three-state per lo stesso motivo di A.

Gli altri pin sono organizzati in tre gruppi e sono tutti attivi bassi.

Il gruppo System Control raggruppa i segnali d'uscita che permettono il controllo del sistema collegato alla CPU.

Per gestire la lettura e scrittura (R/W) con l'esterno fa uso di due pin distinti: nRD e nWR, con cui seleziona il verso della comunicazione. Quando nRD, abbrev. di Read, è attivo, la periferica selezionata deve mettere sul bus D il dato richiesto. Mentre quando nWR, abbrev. di Write, è attivo, la CPU ha fornito sul bus D un dato valido e comunica alla periferica che può essere letto.

Lo Z80 indirizza gli I/O allo stesso modo della memoria, per cui vede quest'ultimi come dei registri indirizzabili. Per distinguere tra le operazioni di R/W verso memoria o I/O utilizza due pin: nMREQ e nIORQ. Con nMREQ, abbrev. di Memory request, la CPU comunica che l'indirizzo presente su A è utilizzabile per indirizzare una memoria. Per cui può essere usato come segnale di abilitazione delle memorie. Mentre con nIORQ, abbrev. di I/O request, si possono comunicare due eventi. Il primo è la controparte di nMREQ, cioè comunica all'I/O che l'indirizzo presente sul bus A è valido per essere letto. Il secondo è per comunicare il ricevimento di una richiesta di interruzione, interrupt acknowledge. Durante l'interrupt acknowledgement non viene attivato il pin nRD sebbene la CPU stia richiedendo alla periferica un dato poiché si vuole che non venga fatta confusione un'operazione di normale lettura. Al contrario si attiva il pin nM1, abbrev. di Machine cycle one, che comunica proprio quest'avvenimento. Lo stesso pin viene usato per segnalare che l'operazione di lettura della memoria che si sta effettuando è un'operazione di recupero dell'istruzione, instruction fetch, o come viene chiamata nel datasheet opcode fetch. Quest'informazione è utile per le periferiche, specialmente quelle della famiglia Z80, che riconoscono lo stato della macchina. L'ultimo pin di questo gruppo è nRFSH, abbrev. di Refresh. Questo pin comunica alle memorie collegate ad A che sul bus è presente un indirizzo per le operazioni di refresh. Quest'operazione è indispensabile per le Dynamic RAM, DRAM, che altrimenti perderebbero il loro contenuto. La presenza di una logica di refresh già integrata nel microprocessore è stato un altro punto di forza dello Z80 rispetto all'Intel 8080. L'indirizzo di refresh presenta solo i 7 bit meno significativi poiché al tempo non era utile avere più bit per fare il refresh delle DRAM sul mercato.

Il secondo gruppo, CPU control, invece raggruppa quattro ingressi e un'uscita che controllano il comportamento del microprocessore.

Il pin nRESET, se mantenuto attivato per almeno tre cicli di clock, avvia la procedura di reset del microprocessore che azzera tutti i registri. Durante il reset della CPU, i bus A e D vanno in alta impedenza e i segnali di controllo diventano inattivi.

Con una particolare istruzione, si può portare la CPU in uno stato di attesa in cui continua ad eseguire operazioni nulle per continuare a svolgere il refresh della memoria. La CPU comunica l'ingresso in questo stato attivando l'uscita nHALT. Per uscire da questo stato bisogna fornire alla CPU un interrupt attivo.

Lo Z80 distingue tra due tipi di interruzioni, interrupt: quelli mascherabili via software che vengono richiamati abbassando il pin nINT; e quelli non mascherabili sul pin nNMI. I secondi hanno una priorità maggiore degli altri e vengono serviti per primi. Gli interrupt vengono serviti solamente alla fine di un'istruzione.

L'ultimo pin del gruppo è l'ingresso nWAIT. Questo pin viene attivato dalle memorie o dagli I/O su cui sta avvenendo un'operazione R/W. La periferica abbassa questo pin per comunicare che non è pronta ad accettare il dato e lo mantiene abbassato. La CPU da parte sua rimane in un ciclo di attesa fintanto che non vede disattivarsi il segnale. Il mantenere la CPU in uno stato di attesa per periodi prolungati può portare a problemi di refresh poiché la CPU non lo effettua con la periodicità corretta.

L'ultimo gruppo gestisce la possibilità di avere più microprocessori o un DMA collegati agli stessi bus e si chiama CPU bus control.

In questo gruppo ci sono due segnali: nBUSREQ, abbrev. di Bus request, e nBUSACK, abbrev. di Bus acknowledge.

Con il primo segnale una periferica richiede di poter usare i bus in modo esclusivo gestendo lei gli indirizzamenti, i dati e i controlli. Questo evento ha una priorità maggiore di nNMI e viene servito alla fine dei cicli macchina delle istruzioni.

Quando la CPU accetta la richiesta, lo comunica attivando il pin nBUSACK e mette i bus A, D e il gruppo System Control in alta impedenza. Durante questo periodo la CPU rimane in attesa non svolgendo nessuna attività per poi ripartire dallo stesso punto in cui si era interrotta. Come per i cicli di nWAIT, periodi prolungati di nBUSREQ possono portare a problemi di refresh poiché la CPU non effettua più l'operazione richiesta con la corretta periodicità.

Architettura

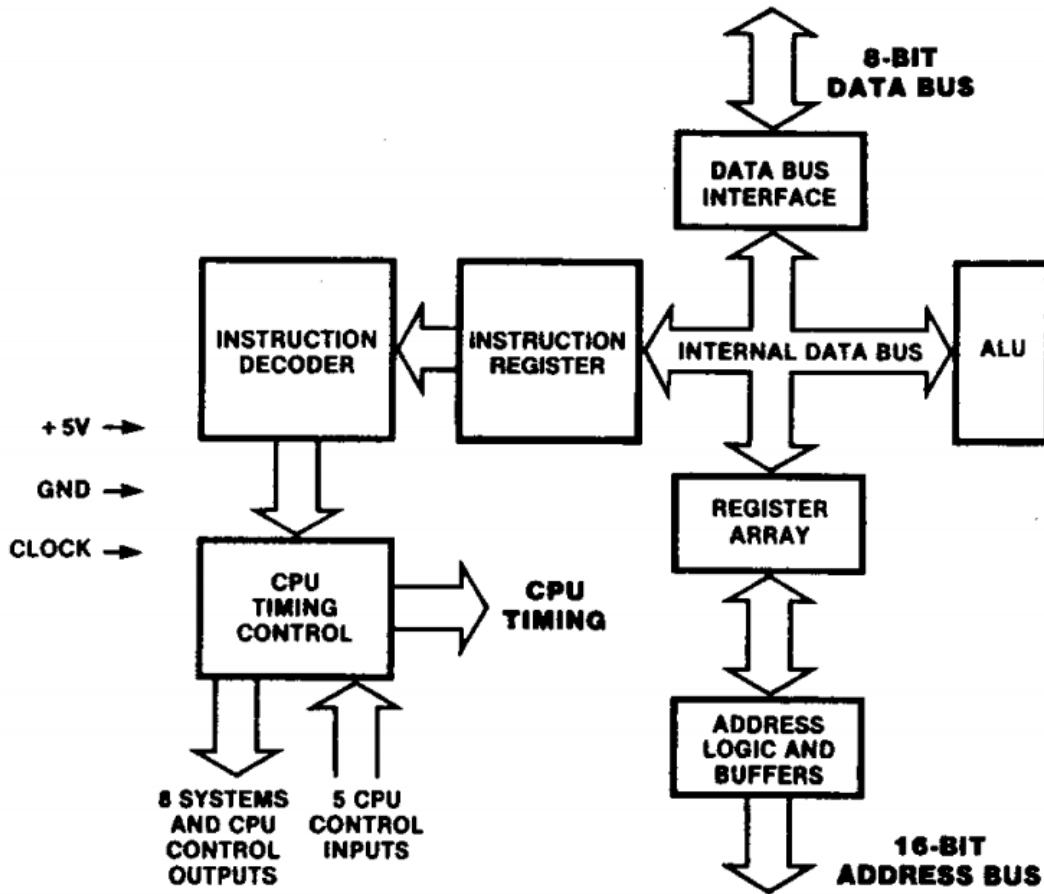


FIGURA 3-4 - DIAGRAMMA A BLOCCHI DELLO Z80 [22]

L'architettura dello Z80 è di tipo CISC, Complex Instruction Set Computer, perché presenta 158 istruzioni di lunghezza variabile la cui esecuzione può durare da uno a cinque cicli macchina [20]. Inoltre ci sono istruzioni che possono venire eseguite più volte fino al verificarsi di una condizione. Ad esempio una di queste è l'istruzione *LDI* che esegue una copia di una locazione di memoria in un'altra avanzando di locazione in locazione e decrementando un contatore. Quando quest'ultimo arriva a 0 si termina l'esecuzione e si passa all'istruzione successiva.

Lo Z80 divide il tempo in cui esegue un'istruzione in due modi: cicli di clock, detti T-cycles, che vanno da un fronte positivo del segnale di clock al successivo, e cicli macchina, detti M-cycles, che raggruppano i T-cycles in base alle operazioni che compie. Le operazioni di R/W su memoria o I/O, interrupt acknowledge e opcode fetch durano tutte un singolo ciclo macchina che però ha durata variabile da 3 a 5 cicli di clock.

I cicli macchina vengono numerati da 1 a 6 in base alla durata di un'operazione. Per esempio,

tutte le operazioni aritmetiche a 8 bit tra l'accumulatore e un altro registro durano tutte 1 M e 4 T. Mentre l'istruzione *EX (SP), HL*, che scambia il contenuto di HL con la testa della pila, dura 6 M e 23 T.

M1 è sempre un ciclo di opcode fetch. Nell'esecuzione delle istruzioni, ci possono essere dei cicli macchina di lunghezza variabile da 1 a 6 T per svolgere delle operazioni interne. Per questo motivo nel datasheet sono fornite le durate delle istruzioni sia in M-cycles sia T-cycles.

Lo Z80 usa la tecnica dell'I/O isolato poiché per comunicare con gli I/O usa delle istruzioni dedicate [21]. Inoltre il datasheet afferma che è usabile solamente la parte meno significativa dell'indirizzo per selezionare i dispositivi che di conseguenza possono esser al massimo 256.

Le operazioni aritmetiche e logiche a 8 bit sono basate sull'accumulatore. Per cui hanno sempre l'accumulatore come uno degli operandi e sempre lo stesso registro come risultato.

Mentre le operazioni a 16 bit usano i registri HL, IX o IY allo stesso scopo.

Gli indirizzi, come un generale i valori a 16 bit, vengono letti e scritti nella memoria secondo la tipologia Little Endian in cui all'indirizzo minore corrisponde la parte bassa del dato.

Come si vede dalla Figura 3-5 - Registri della CPU Z80 , all'interno dello Z80 vi sono tre blocchi adibiti al controllo della CPU scandendo il ritmo delle operazioni e decodificando le istruzioni lette. Questi blocchi sono connessi per mezzo di un bus interno alle interfacce dati e indirizzi ma anche all'unità aritmetica ALU e ai registri.

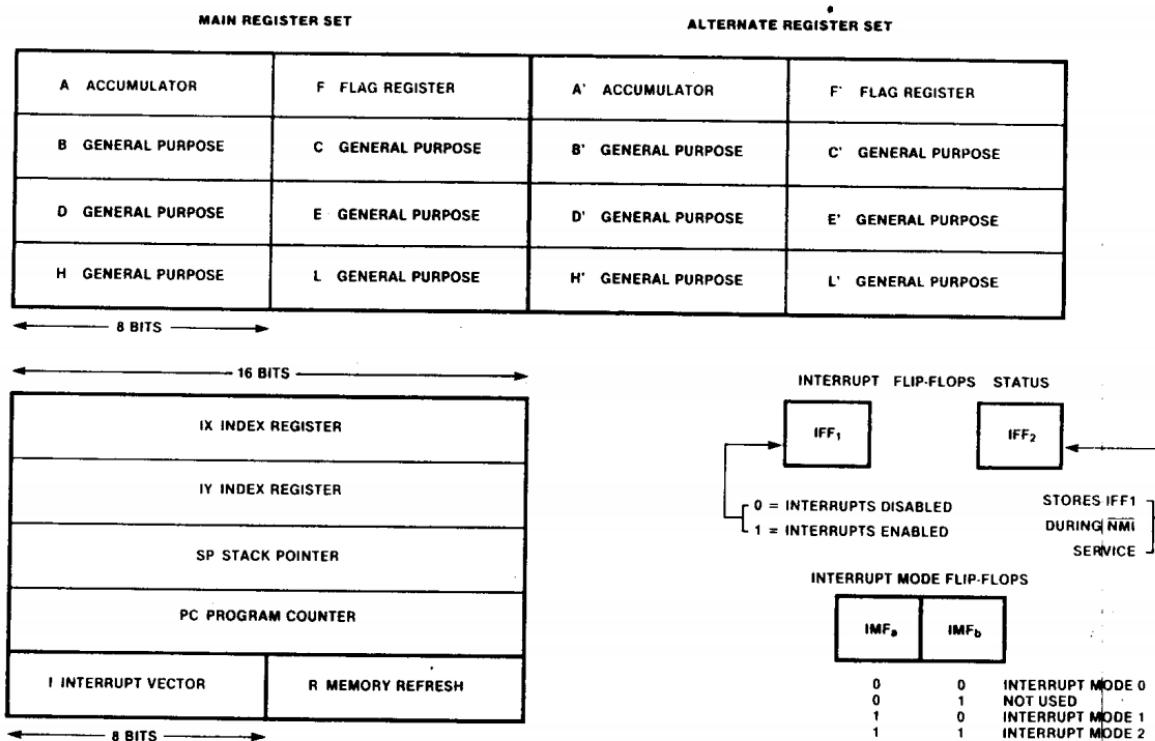


FIGURA 3-5 - REGISTRI DELLA CPU Z80 [22]

Lo Z80 mette a disposizione 6 registri nominati B, C, D, E, H, L. Questi sono registri a scopo generico, general purpose, che possono essere usati singolarmente a 8 bit o accoppiati a 16 bit come BC, DE, HL. I registri a 16 bit si possono usare per l'indirizzamento.

Assieme a questi registri general purpose c'è la coppia formata dall'accumulatore A e dal registro F che contiene i contrassegni derivanti dalle operazioni dell'ALU, detti flag. Assieme formano la coppia AF che non può essere usata per indirizzare. Nel registro F sono contenuti i flag:

- S sign flag, che segnala il segno del risultato;
- Z zero flag, che segnala se il risultato è 0;
- P/V parity-overflow flag, che segnala se nell'ultima operazione aritmetica è avvenuto un overflow o se il risultato dell'ultima operazione logica è pari;
- H half-carry flag, che segnala se nell'ultima operazione si è verificato un carry tra il primo o il secondo nibble, cioè tra i quattro bit meno significativi e i successivi;
- N subtract flag, che segnala se nell'ultima operazione è stata effettuata una sottrazione;
- C carry flag, che segnala se è stato prodotto un riporto dal bit più significativo.

Per ognuno di questi otto registri c'è una copia detti registri alternativi od ombra. I registri possono essere scambiati con quelli principali per mezzo di una coppia di istruzioni di scambio, Exchange. La presenza di questo set alternativo permette di risolvere velocemente un interrupt senza dover salvare il contesto su una pila oppure di svolgere delle operazioni con tecniche di background-foreground.

In aggiunta a questi registri ci sono altri registri con scopi precisi.

La coppia di registri d'indicizzazione indipendenti IX e IY, entrambi a 16 bit, sono usabili come indirizzi specialmente come base per puntare un'area di memoria attraverso delle istruzioni di indicizzazione.

Il registro SP, stack pointer, a 16 bit, punta alla prima posizione libera in testa ad una pila LIFO (Last-In First-Out). La pila è gestita attraverso una coppia di istruzioni *PUSH* e *POP*. La gestione automatica è utile per il salvataggio del contesto in caso di interrupt e il passaggio di parametri alle subroutine richiamate. Lo stack cresce verso gli indirizzi minori per cui un'operazione di *PUSH* decremente SP.

Il program counter PC è contenuto in un registro a 16 bit che può essere scritto e modificato con delle operazioni di salto. Il PC viene incrementato alla fine di ogni operazione di lettura o scrittura dalla memoria e sovrascritto in caso di salto.

Il registro R, a 7 bit, contiene l'indirizzo di refresh che viene messo sul bus. Viene

automaticamente incrementato alla fine dell'operazione e può essere gestito con un registro a 8 bit in relazione con l'accumulatore.

Si può gestire allo stesso modo di R, il registro I detto Interrupt Page Address. Il registro contiene la parte alta di un indirizzo che punta ad una voce di una tabella contenente informazioni per gestire una richiesta di interrupt collegata. Però questo avviene solo in una delle tre modalità di gestione degli interrupt.

Ai precedenti registri si aggiungono due coppie di flip-flops.

La prima coppia detta Interrupt Status FFs fa da maschera per gli interrupt normali. Il primo bit se abilitato permette la ricezione degli interrupt di nINT da parte della CPU. Il secondo bit serve come salvataggio del primo quando vengono servite le richieste da parte di nNMI.

La seconda coppia detta Interrupt Mode FFs definisce la tipologia di servizio degli interrupt usati.

Gestione degli eventi di interrupt

Come già detto in precedenza, lo Z80 distingue tra interrupt mascherabili, INT, e non, NMI. Entrambi gli interrupt vengono serviti solo nel caso in cui non sia stato attivato nBUSREQ in precedenza e alla fine delle istruzioni.

Gli interrupt non mascherabili, NMI, sono stati pensati per servire solo interrupt di alta priorità e che richiedono un servizio immediato. Un esempio è un problema all'alimentazione che richiede un immediato salvataggio del contesto da parte della CPU. Per questo non sono mascherabili via software. Essendo considerati come eventi unici hanno una sola strategia di servizio.

Quando viene accettato l'interrupt, viene eseguita in automatico l'istruzione *RST 66H*. L'istruzione *RST hh*, abbrev. di Restart, ha un solo parametro *hh* che appartiene ad un insieme di otto indirizzi da *00H* a *38H* distanti ognuno 8 byte. L'esecuzione è simile ad un'istruzione di *CALL* e consiste nel salvare il PC nello stack e caricare l'indirizzo puntato dal parametro nel PC.

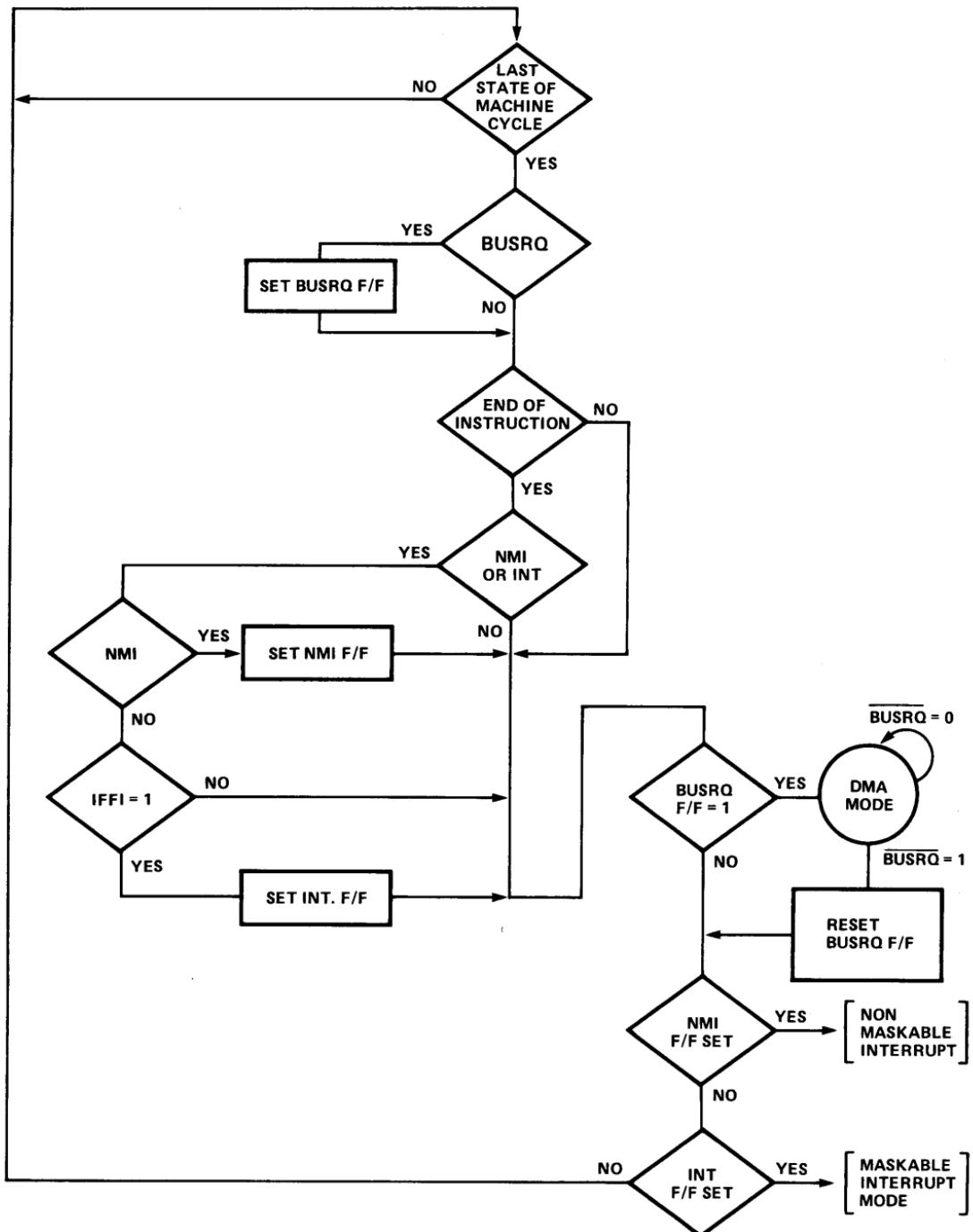


FIGURA 3-6 - DIAGRAMMA DI FLUSSO DELLA GESTIONE DEGLI INTERRUPT [26]

Nel caso di un NMI, come si vede nella Figura 3-6 - Diagramma di flusso della gestione degli interrupt , la CPU esegue un opcode fetch ignorando l'istruzione letta. Poi salva PC nello stack, disabilita gli INT resettando IFF1 e salvando nell'altro, IFF2, lo stato del primo per ripristinarlo alla fine della routine. Infine salta alla locazione 66H a cui si deve trovare la routine di servizio.

Per uscire da un NMI si usa l'istruzione apposita *RETN*, abbrev. per Return NMI. L'istruzione ricarica in PC il valore salvato nello stack, esattamente al contrario di RST hh. In aggiunta ad

una normale istruzione di ritorno RET, ricarica il valore di IFF2 in IFF1 così da ripristinare l'abilitazione degli INT allo stato prima dell'interrupt.

Gli interrupt mascherabili, INT, sono abilitati o meno dallo stato di IFF1. Lo stato dei bit di IFF è selezionato da due istruzioni *DI*, abbrev. di Disable Interrupt, che resetta i bit disabilitando gli interrupt e *EI*, abbrev. Enable Interrupt, che setta i bit abilitando gli interrupt. In ogni caso gli INT rimangono disabilitati fino alla fine dell'istruzione successiva. Questo per permettere il corretto funzionamento delle istruzioni della famiglia RET, che solitamente seguono le precedenti alla fine delle routine di servizio degli interrupt.

La possibile disabilitazione degli interrupt permette di garantire che alcune porzioni di codice vengano eseguite nel tempo corretto senza interruzioni. Inoltre, a differenza degli NMI, gli interrupt non vengono disabilitati automaticamente all'inizio della routine di servizio per cui si può implementare una logica di interrupt nidificato [20].

Lo Z80 per mantenere la compatibilità con i sistemi basati sull'Intel 8080, ma anche per avere una modalità preferenziale per le sue periferiche, permette tre modi diversi di gestire un interrupt. In ogni caso però l'asserzione avviene allo stesso modo: avviene una lettura di un dato attivando solo nIORQ e nM1 così da segnalare alla periferica interessata che la CPU sta prendendo in carico la richiesta. In aggiunta, la scelta della modalità avviene via software per mezzo di tre istruzioni dedicate: IM0, IM1 e IM2. Le quali settano il valore dei bit IMF.

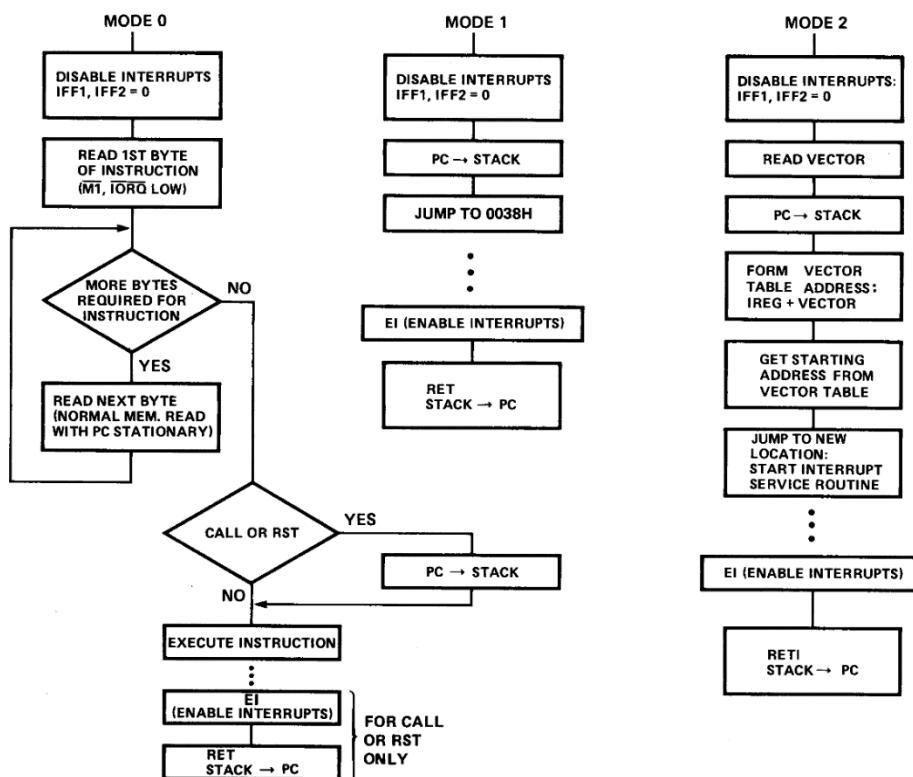


FIGURA 3-7 - SEQUENZE DI SERVIZIO DI UN INTERRUPT MASCHERABILE [24]

La prima modalità detta Mode 0, selezionata con IM0, è completamente compatibile con le periferiche dell'Intel 8080. Durante il ciclo di interrupt acknowledge, la periferica mette sul bus D un'istruzione che viene eseguita dalla CPU. Nel caso di istruzioni che richiedono più byte, solo la prima operazione di lettura è di interrupt acknowledge mentre le altre sono semplici letture dalla memoria. Per cui la soluzione migliore è che la periferica fornisca un'istruzione *RST hh* verso la propria routine. Ogni routine ha di base 8 byte in cui può avvenire la disabilitazione degli altri interrupt e il salvataggio del contesto prima di saltare all'effettiva routine. Nell'ipotesi vantaggiosa che le periferiche sfruttino l'istruzione *RST hh*, si possono servire 8 diversi interrupt.

La modalità Mode 1, selezionata con IM1, invece è simile a come vengono serviti gli NMI. Dopo il ciclo di interrupt acknowledge viene eseguito *RST 38H*, servendo quindi un unico tipo di interrupt.

La modalità Mode 2, selezionata con IM2, è quella tipica delle periferiche dello Z80 e attua un interrupt vettorizzato [20]. Durante il ciclo di interrupt acknowledge, la periferica carica sul bus l'indice di una entry in una tabella di cui verranno letti solo i sette bit più significativi. La CPU nel ciclo successivo salva nello stack il valore di PC. Poi legge un indirizzo contenuto in una tabella puntandolo per mezzo della giustapposizione tra il registro I e il valore fornito dalla periferica. Dopo aver letto il nuovo indirizzo, che punta alla routine di servizio adatta, effettua l'opcode fetch e ne inizia l'esecuzione. Con questa modalità si possono servire 128 interrupt diversi con una latenza minima di cinque cicli macchina.

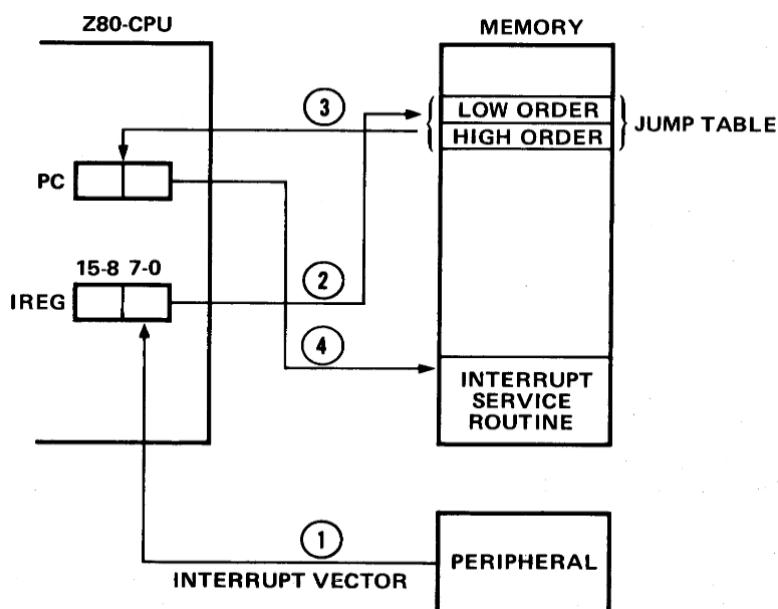


FIGURA 3-8 - SEQUENZA DI SERVIZIO DEGLI INTERRUPT VETTORIZZATI [24]

Il salvataggio del contesto non viene fatto automaticamente e può essere fatto attraverso l'uso dei registri ombra oppure sullo stack. Quest'ultima opzione è l'unica in caso di interrupt nidificati.

Per ritornare dalle routine di interrupt si possono usare due istruzioni di ritorno che svolgono la stessa funzione dal punto di vista della CPU: *RET* e *RETI*. Entrambe ricaricano in PC il valore salvato nello stack. La seconda è pensata appositamente per le periferiche della famiglia Z80 poiché riconoscono l'istruzione diversa e si comportano di conseguenza.

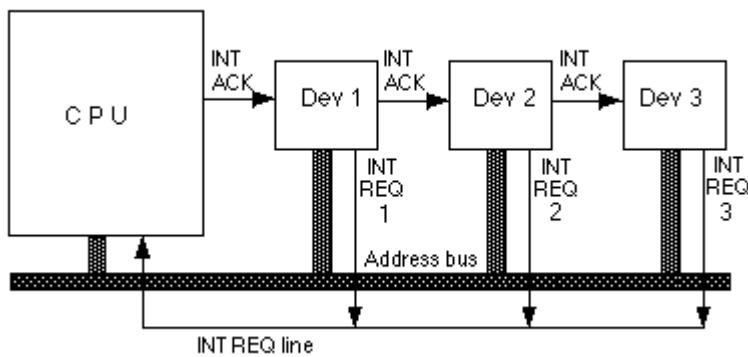


FIGURA 3-9 - ESEMPIO DI CONNESSIONE DAISY CHAIN.
NELL'IMMAGINE L'ADDRESS BUS VIENE SOSTITUITO CON IL DATA BUS
PER LO Z80.

Per permettere di collegare più INT con priorità differenti si usa una daisy chain. La daisy chain è una connessione in serie delle periferiche sulla stessa linea di controllo. Ogni nodo della daisy chain condivide una connessione al data bus, un'uscita per la richiesta di interrupt, un ingresso e un'uscita di un segnale di interrupt acknowledge. Tutte le uscite di richiesta interrupt sono collegate assieme per mezzo di una wired-AND. Mentre la linea di acknowledge esce dallo Z80 ed entra nel primo blocco, da questo esce un altro acknowledge e così via. La priorità tra gli interrupt viene dettata dalla vicinanza alla CPU: il più vicino ha priorità maggiore.

La CPU quando asserisce una richiesta, invia il segnale di interrupt acknowledge, che nel caso dello Z80 avviene con nIORQ e nM1 attivi contemporaneamente. Dal primo nodo, viene ricevuto l'acknowledge e se la periferica effettivamente ha generato un interrupt, mette sul bus il proprio indirizzo della tabella e non trasmette l'acknowledge al nodo successivo. Se invece non l'ha generato, trasmette al blocco successivo il segnale. Per servire altri interrupt, che continuano a tenere bassa la linea nINT, lo Z80 continua a svolgere i cicli di interrupt acknowledge.

Normalmente per ogni periferica bisogna creare una piccola interfaccia che svolga questa funzione. Nel caso delle periferiche della famiglia Z80, in particolare PIO, SIO e CTC, queste mettono a disposizione due linee: IEI, abbrev. di Interrupt Enable Input; IEO, abbrev. di Interrupt Enable Output. Questi pin sono collegati in modo che l'IEO della periferica precedente sia collegato con l'IEI della successiva. La prima periferica è collegata in modo da avere IEI sempre abilitato.

Ogni periferica riconosce autonomamente la condizione di interrupt acknowledge perché sono collegate al gruppo System Control. Se avviene un interrupt, la periferica interessata disabilita IEO altrimenti quest'ultimo ha lo stesso valore di IEI. Se IEI è disabilitato, nel caso di interrupt richiesto, la periferica non risponde all'interrupt acknowledge. Nel momento in cui una periferica, con IEI abilitato, risponde all'interrupt acknowledge, mette sul bus dati l'indirizzo della entry. Alla fine della routine, che si chiuderà con l'istruzione adatta *RETI*, la periferica riconosce l'istruzione e abilita IEO permettendo alla successiva periferica di servire il proprio interrupt.

Instruction set

Lo Z80 mette a disposizione 158 diverse istruzioni in cui l'opcode è per lo più contenuto in un solo byte. Le istruzioni con più byte di opcode, al massimo due, nel primo byte contengono un prefisso che solitamente segnala operazioni su IX e IY oppure rotazioni e traslazioni.

Il set di istruzioni contiene come sottoinsieme il set dell'Intel 8080. La lunghezza effettiva delle istruzioni dipende dal metodo di indirizzamento dei dati. Per cui si può passare da istruzioni di un singolo byte come *LD A, B* in cui l'indirizzamento dei registri è diretto sino a *LD (IX + d), n* che per contenere l'offset *d* per IX e il valore *n* occupa 4 byte.

Le istruzioni hanno la forma generale:

```
<mnemonico> {<operando_1>{ , <operando_2>} }
```

In base all'istruzione, possono essere presenti entrambi gli operandi, solo il primo oppure nessuno. Nel caso di istruzioni con due operandi, si ha sempre che il secondo va a modificare il primo. Ad esempio in *ADD HL, BC* si ha che in HL viene caricato il valore della somma tra HL stesso e BC a 16 bit.

```
<operando_1>, <operando_2>: <r> | (HL) | (IX + <d>) | (IY  
+ <d>) | (C) | (A) | <n> | <nn> | (<nn>) | (BC) | (DE) |  
<b> | <e> | <cc> | <qq> | <ss> | <pp> | <rr> | <s> | <m>
```

Instruction Notation Summary

Table 4 lists the operand notations and descriptions used in the Z80 Instruction Set.

Table 4. Instruction Notation Summary

Notation	Description
<i>r</i>	Identifies any of the registers A , B , C , D , E , H , or L
<i>(HL)</i>	Identifies the contents of the memory location, whose address is specified by the contents of the register pair HL
<i>(IX+d)</i>	Identifies the contents of the memory location, whose address is specified by the contents of the <i>Index</i> register pair IX plus the signed displacement <i>d</i>
<i>(IY+d)</i>	Identifies the contents of the memory location, whose address is specified by the contents of the <i>Index</i> register pair IY plus the signed displacement <i>d</i>
<i>n</i>	Identifies a one-byte unsigned integer expression in the range (0 to 255)
<i>nn</i>	Identifies a two-byte unsigned integer expression in the range (0 to 65535)
<i>d</i>	Identifies a one-byte signed integer expression in the range (-128 to +127)
<i>b</i>	Identifies a one-bit expression in the range (0 to 7) . The most-significant bit to the left is bit 7 and the least-significant bit to the right is bit 0
<i>e</i>	Identifies a one-byte signed integer expression in the range (-126 to +129) for <i>relative</i> jump offset from current location
<i>cc</i>	Identifies the status of the <i>Flag Register</i> as any of (NZ, Z, NC, C, PO, PE, P, or M) for the conditional jumps, calls, and return instructions
<i>qq</i>	Identifies any of the register pairs BC , DE , HL or AF
<i>ss</i>	Identifies any of the register pairs BC , DE , HL or SP
<i>pp</i>	Identifies any of the register pairs BC , DE , IX or SP
<i>rr</i>	Identifies any of the register pairs BC , DE , IY or SP
<i>s</i>	Identifies any of <i>r</i> , <i>n</i> , <i>(HL)</i> , <i>(IX+d)</i> or <i>(IY+d)</i>
<i>m</i>	Identifies any of <i>r</i> , <i>(HL)</i> , <i>(IX+d)</i> or <i>(IY+d)</i>

FIGURA 3-10 - SOMMARIO DEGLI OPERANDI DIRETTAMENTE DALLA GUIDA UTENTE DELLO Z80 [23]

Gli operandi invece possono essere direttamente valori, puntatori ad uno specifico registro o bit, oppure puntatori a locazioni di memoria, riconoscibili perché contenuti in parentesi tonde. Le istruzioni si dividono in 11 gruppi: caricamenti a 8 bit e a 6 bit; exchange, trasferimenti a blocchi e ricerca; operazioni a 8 bit e 16 bit; istruzioni di controllo della CPU; rotazioni e scorrimenti; operazioni sui bit; salti; chiamate, ritorni e restart; operazioni sugli I/O.

Di nota sono le operazioni di trasferimento di blocchi e ricerca. Queste sono operazioni che continuano ad essere eseguite fintantoché non si verifica una condizione. La condizione comune a tutte è quella che un contatore arrivi a 0. Nel caso in cui dopo ogni singola esecuzione non si verifichi la condizione, viene ricaricato nel PC la posizione dell'istruzione. Per cui la CPU continua a fare il fetch della stessa istruzione per eseguire il ciclo.

Inoltre lo Z80 presenta una specifica integrazione per svolgere sia somme che sottrazioni con

operandi in BCD. Per mezzo di un'istruzione di decimal adjust, *DAA*, che in base allo stato dei flag H ed N ed al valore dell'accumulatore, ricostruisce il risultato BCD dell'operazione.

Lo Z80 permette ben 10 metodi diversi di indirizzare gli operandi.

Permette l'indirizzamento immediato sia di valori a 8 bit o 16 bit fornendoli nei byte successivi all'opcode.

Allo stesso modo si possono indirizzare per mezzo di un campo dell'istruzione un registro, una coppia di registri o un singolo bit in un byte. Alcune istruzioni come quelle aritmetico-logiche a 8 bit, indirizzano implicitamente l'accumulatore.

Come indirizzamento indiretto lo Z80 permette di usare preferibilmente il registro HL per mantenere l'indirizzo oppure SP per quanto riguarda lo stack.

Nel caso particolare dei registri IX e IY, si può attuare un indirizzamento indicizzato. Si può per esempio puntare alla locazione $(IX + d)$ con d un indice da -127 a +128 da sommare a IX solo per creare il puntatore.

3. Informazioni sull'organizzazione dello Z80

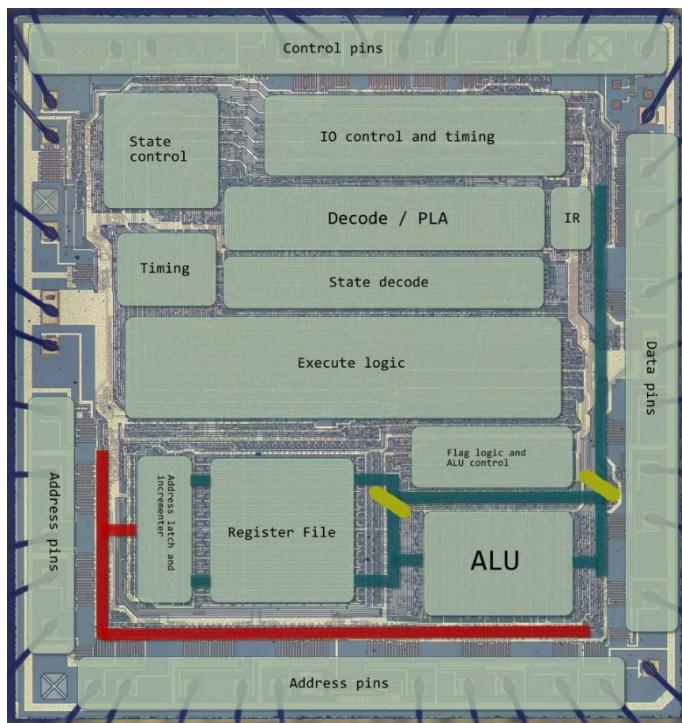


FIGURA 3-11 - DIE DELLO Z80 CON LE UNITÀ OPERATIVE EVIDENZIATE [27]

Per quanto sia chiaramente ben documentata l'architettura dello Z80, non ci sono informazioni ufficiali sulla sua organizzazione.

Tutte le informazioni che sono apparse negli anni successivi derivano da un processo di reverse engineering del die. Cioè si è analizzato al microscopio il chip e dalle tracce ottiche lasciate dai vari passaggi produttivi si è risaliti alla sua organizzazione a livello dei singoli transistor. Da lì si sono fatti i vari passaggi di astrazione sino a definire le unità operative e le loro connessioni. Di notevole importanza è stato il lavoro di Ken Shirriff, che nel suo blog ha scritto vari post al riguardo [25].

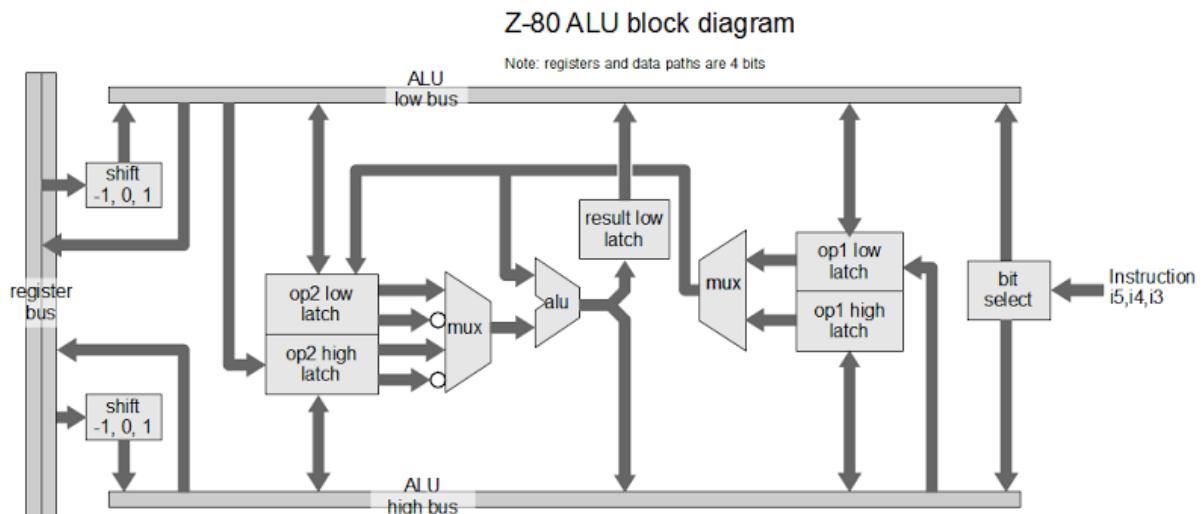


FIGURA 3-12 - SCHEMA BLOCCHI DELL'ALU [29]

Come prima scoperta si è visto che l'ALU dello Z80 non è a 8 bit, come lo sono le parole, ma a 4 bit. Di conseguenza la ALU svolge le operazioni un nibble alla volta. Per prima cosa la ALU carica gli operandi provenienti dal bus in due latch, uno per ogni operando. Ogni latch è diviso in nibble più significativo e meno significativo. Le uscite di ogni latch sono collegate a due multiplexer divisi per i due operandi, in questo modo l'ALU può scegliere con quale parte operare. Dopodiché la ALU prima compie l'operazione sui nibble inferiori e salva il risultato su un latch. Poi somma le parti superiori e mette sul bus sia il latch della parte inferiore che la parte appena calcolata.

I flag P di P/V, N, S e Z vengono generati direttamente dagli operandi. Il flag H viene generato al primo passaggio. Mentre il flag V di P/V e C vengono aggiornati solo nel secondo passaggio.

Per le operazioni sui bit come *SET*, *RST* e *BIT*, la ALU compie delle operazioni di AND e OR con delle maschere generate in base al bit selezionato. La decodifica della selezione del bit avviene direttamente nell'ALU leggendo il campo corrispondente dall'istruzione, invece di ricevere dei segnali di controllo dal decoder.

A differenza di altri processori come l'Intel 8085, successore dell'Intel 8080, lo Z80 non ha un'ALU dedicata per svolgere le operazioni di rotazione e scorrimento. Queste, essendo tutte ad un bit, sono svolte nel caricamento degli operandi, per cui l'ALU non deve svolgere nessuna operazione successiva. Per lo scorrimento come cifre BCD, che è svolto dalle istruzioni *RRD* e *RLD*, l'ALU compie la stessa operazione scambiando nei latch i nibble con quelli di un indirizzo in memoria [26].

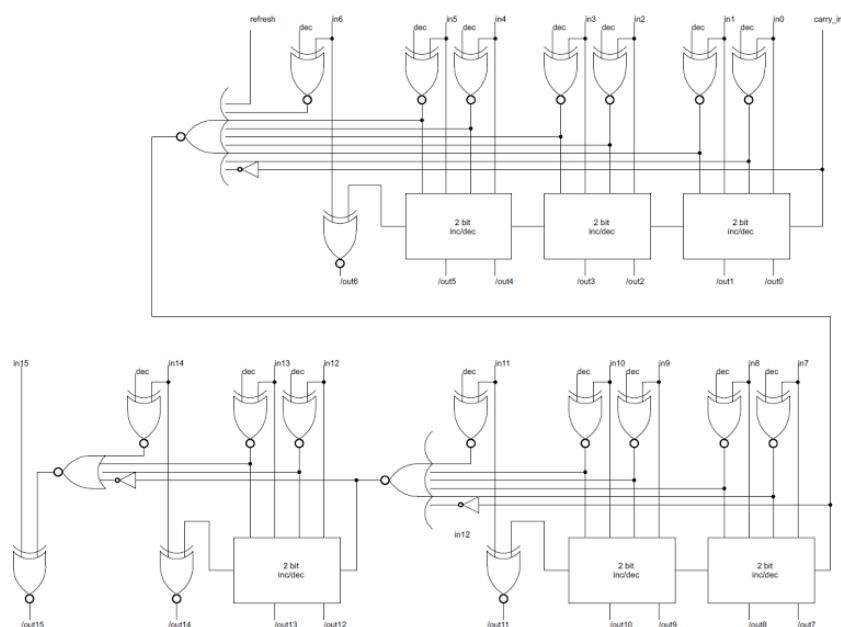


FIGURA 3-13 - CIRCUITO COMPLETO DELL'INC/DEC [30]

Vicino al latch degli indirizzi è presente un incrementer/decrementer dedicato, abbrev. inc/dec, a 16 bit per svolgere tutte le operazioni del gruppo *INC* e *DEC*. In questo inc/dec viene caricato il valore del bus indirizzi, dopo il latch dedicato, e fornisce il nuovo valore sul bus interno. La necessità di avere un inc/dec dedicato deriva dal poter svolgere un tipo rudimentale di pipeline. Questo consiste nella sovrapposizione delle fasi di fetch e decode. Queste si sovrappongono nell'ultimo T-cycle in cui si sta effettuando l'operazione di refresh e la decodifica o intera esecuzione dell'istruzione appena ottenuta. Per cui si sfrutta l'inc/dec per caricare in PC il nuovo valore subito dopo il fetch e per caricare il valore di R durante il primo T-cycle dell'esecuzione o decodifica.

La cella base dell'inc/dec è a due bit con carry in ingresso e sfrutta la tecnica del carry-skip per ridurre il ritardo di propagazione.

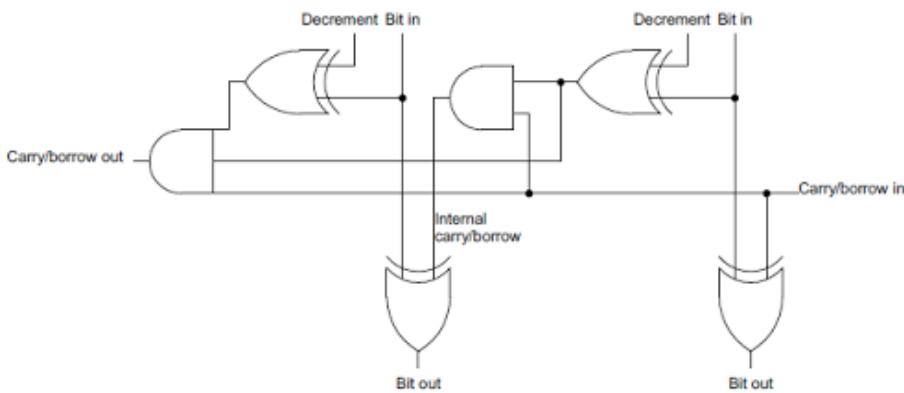


FIGURA 3-14 -CELLA INC/DEC A 2 BIT BASE [30]

Poi questi inc/dec a 2 bit sono collegati riportando il serie il carry in gruppi di due o tre. Per ognuno di questi gruppi c'è un circuito di carry-look ahead, CLA. Quest'ultimo sfrutta il fatto che se c'è almeno uno 0 non viene generato un carry in caso di incremento mentre se c'è almeno un 1 non viene generato un borrow in caso di decremento. Nel primo circuito c'è un segnale che se abilitato inibisce il carry.

Quest'ultima caratteristica è utile per l'incremento di R. R viene passato all'inc/dec come coppia I-R e quando viene incrementato non si deve intaccare I. In questo modo settando quel particolare bit non avviene il carry oltre il bit 6 per cui non si modifica il registro I e nemmeno i bit più alti di R.

All'interno dello Z80 non sono presenti due bus a 16 bit per i dati così da poter svolgere le operazioni di load sui registri in un'unica soluzione. L'unico presente è quello per gli indirizzi. Per cui un'altra caratteristica dell'inc/dec è la possibilità di non fare nessuna operazione. Questo, collegato al fatto che l'inc/dec per sua configurazione fa da retroazione del latch dell'indirizzo sul bus indirizzi interno, permette di salvare il valore di un registro a 16 bit e poi trasferirlo su un altro.

L'ultimo elemento in più collegato è un riconoscitore del valore 0001H. Questo è sfruttato

nelle operazioni di copia e ricerca di blocchi che ciclano fintantoché il contatore contenuto in BC arriva 0000H. In realtà lo Z80 controlla che il registro sia a 0001H per poi svolgere l'operazione come se fosse quella finale invece di controllare solo dopo il decremento. A differenza dell'ALU, l'inc/dec non modifica i flag [27].

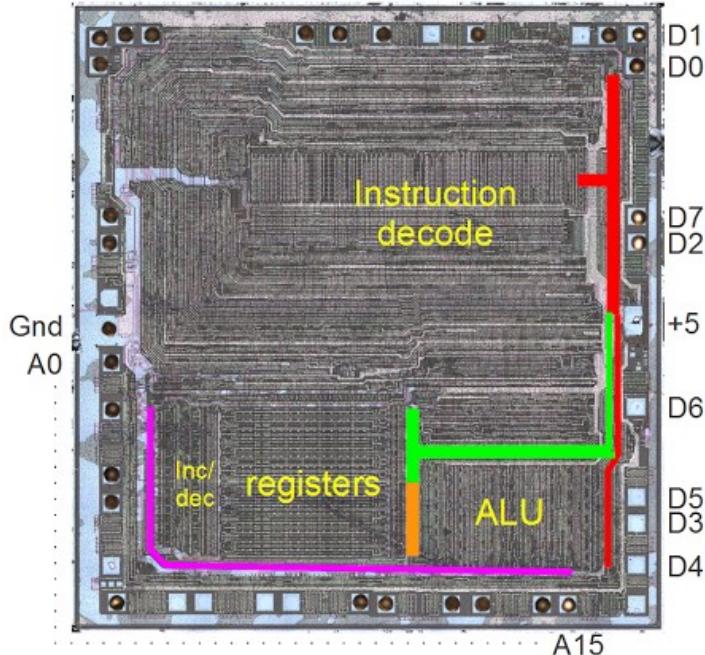


FIGURA 3-15 - DATA BUS INTERNI. IN ROSSO, VERDE E ARANCIO LE TRE SEZIONI DEL BUS A 8 BIT. IN FUCCSIA, IL BUS INDIRIZZI A 16 BIT [31]

Il data bus interno a 8 bit è diviso in tre parti. Una prima che dall'esterno va verso la logica di decodifica e l'ALU poiché da sola decodifica le istruzioni sui bit. Una seconda, parte dalla prima divisa per mezzo di un latch e va verso una pagina dei registri. Dalla seconda ne parte una terza, divisa anch'essa con un latch verso l'altra pagina dei registri e l'ALU. La presenza dei tre bus permette di svolgere operazioni in parallelo all'interno della CPU [28].

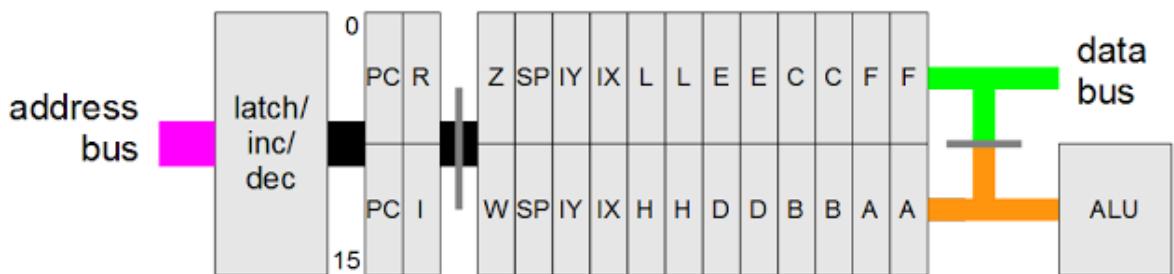


FIGURA 3-16 - STRUTTURA DEI REGISTRI DELLO Z80 [29]

La struttura interna dei registri non è esattamente come quella mostrata nella Figura 3-5 - Registri della CPU Z80 poiché non vi è una vera distinzione tra registri principali e ombra. Tutti i registri A, F, B, C, D, E, H ed L sono ravvicinati e collegati allo stesso bus a 16 bit come alla

stessa coppia di bus a 8 bit. I bus a 8 bit sono divisi in maniera tale che uno punti ai registri che rappresentano la parte alta quindi A, B, D ed H mentre l'altra ai rimanenti F, C, E ed L. I due bus sono separati da degli interruttori che possono unirli o meno così da permettere operazioni in simultanea sui due gruppi di registri.

Poi per ogni registro ce n'è uno identico indistinguibile dal primo. Per cui all'arrivo dell'istruzione *EXX*, che scambia tra registri principali ed ombra per BC, DE ed HL, viene semplicemente modificato un flag che seleziona un gruppo o l'altro senza distinzione. Allo stesso modo funziona l'istruzione *EX AF, AF'*, che scambia le coppie AF, e l'istruzione *EX DE, HL*, che con un altro flag instrada le istruzioni verso DE su HL e viceversa senza distinguere tra gli originali registri o meno.

Sempre collegati a questo gruppo ci sono i registri IX, IY ed SP. Vi sono anche due registri temporanei a 8 bit non visibili al programmatore chiamati W ed Z. Questi sono usati come appoggio nelle istruzioni di lettura di indirizzi, dati o per lo scambio. Vengono usati per esempio nell'istruzione *JP*, abbrev. di Jump, che esegue un salto verso un indirizzo a 16 bit che segue l'opcode. In quel caso l'indirizzo a 16 bit viene caricato durante la lettura su WZ, prima di caricarlo in PC.

A differenza dei precedenti, i registri PC e I-R sono separati dagli altri e isolabili dal bus a 16 bit per mezzo di interruttori per permettere le operazioni di incremento descritte prima. Infine il registro F presenta una copia nell'ALU. Questa viene caricata all'inizio delle operazioni per permettere il corretto utilizzo durante l'elaborazione dell'unità [29].

Per quanto riguarda le istruzioni che gestiscono l'I/O, nel datasheet è detto che possano indirizzare solo per mezzo degli 8 bit meno significativi. Invece, più avanti nello stesso datasheet, è affermato che l'indirizzamento diretto usa A come parte alta mentre l'indirizzamento indiretto per mezzo di C usa la coppia BC [22].

Lo Z80 presenta dei comportamenti non documentati nei datasheet ma dipendenti da residui nella progettazione del decoder delle istruzioni. Quest'analisi è stata ben documentata [30] e spazia dalla presenza di istruzioni particolari sino ad un comportamento non descritto per la gestione degli interrupt.

Di nota sono l'esecuzione dell'istruzione *DAA*, la presenza dell'istruzione *SLL*, abbrev. di Shift logical left, controparte logica dello scorrimento aritmetico a sinistra *SLA*, ed infine la presenza di flip-flops sui pin nINT, nNMI e nBUSREQ.

Per cui le richieste di interrupt possono avvenire in qualsiasi istante, la richiesta viene immagazzinata nei FFs e servita a tempo debito.

In conclusione, non si conosce completamente la vera organizzazione del decoder delle istruzioni. Dal die si riconosce una sezione di timing che conta i vari M-cycles e T-cycles. Il conteggio viene trasmesso a un PLA che fornisce i segnali di controllo. Di conseguenza si può dire solamente che è sicuro che lo Z80 sfrutti un controllo cablato e non microprogrammato.

4. Implementazione del microprocessore Z80 su FPGA

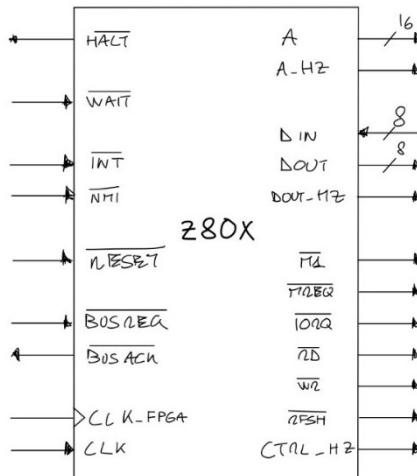


FIGURA 3-17 - PINOUT ENTITY Z80

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 rispondono 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

Nel diagramma di Figura 3-18 - Diagramma dell'organizzazione dello Z80X che mostra i collegamenti principali ho riassunto l'organizzazione interna dello Z80X. Ho omesso tutti i segnali di temporizzazione assieme a tutti i segnali di controllo in ingresso e 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.

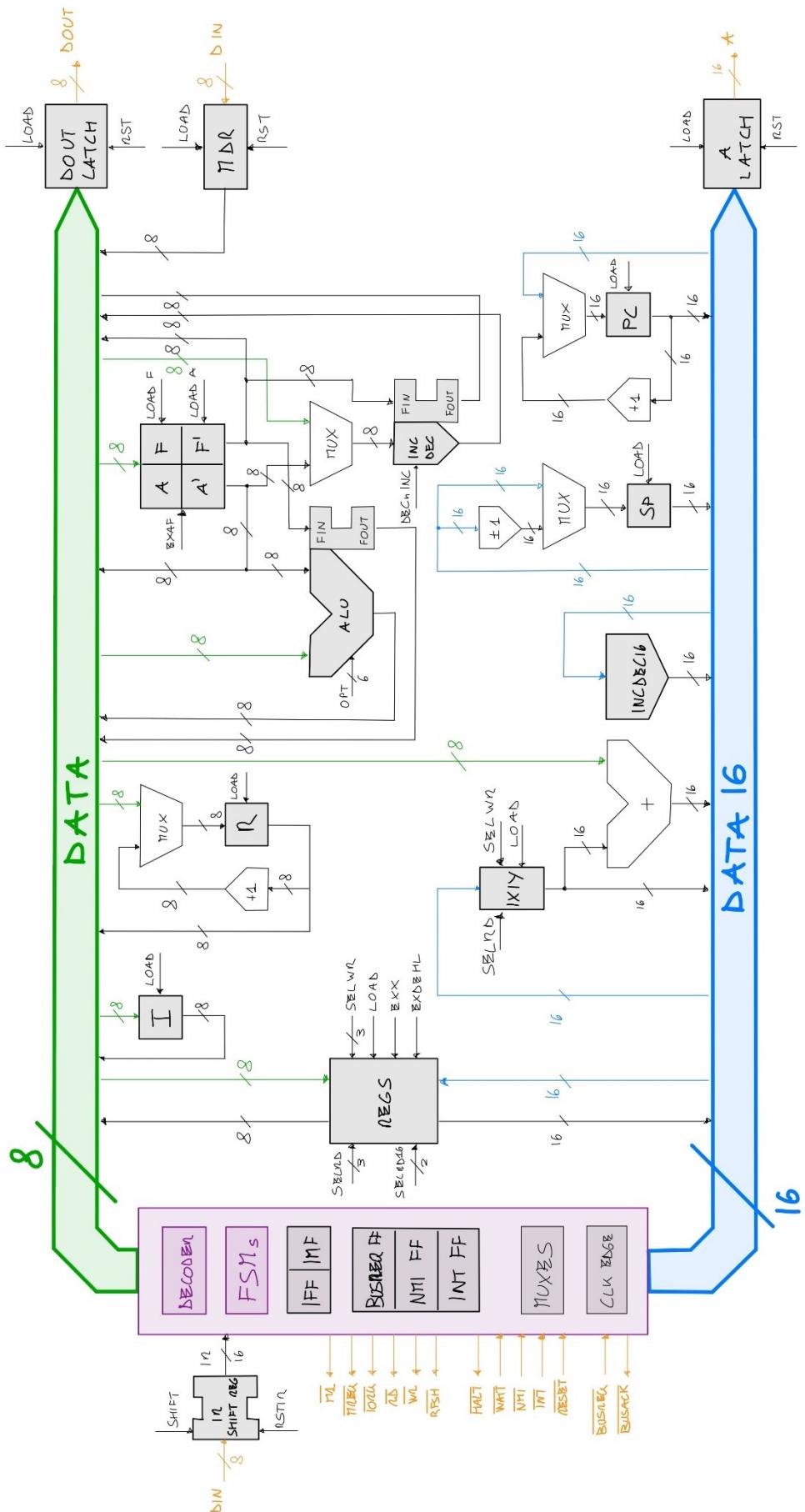


FIGURA 3-18 - DIAGRAMMA DELL'ORGANIZZAZIONE DELLO Z80X CHE MOSTRA I COLLEGAMENTI PRINCIPALI

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.

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 e con lo stesso nome a meno di un suffisso -FF. 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. Il FF di nINT viene settato solamente se IFF1 è attivo poiché fa da maschera all'ingresso, altrimenti il FF rimane invariato.

Affianco a questi FFs ve ne sono altri due chiamati NMIIIF, 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 base però non c'è un semplice contatore di M-cycles e T-cycles. Siccome ho notato una regolarità nell'esecuzione delle istruzioni, ho creato 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 caso delle FSMs in esame lo stato di arrivo è sempre determinato a priori (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. Questo perché 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. Il ritardo dovuto all'attesa del segnale di temporizzazione è trascurabile e facilmente risolvibile con qualche accorgimento durante il progetto. 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 in cui ogni livello controlla altre macchine sottostanti. Queste sono più piccole e svolgono funzioni più semplici 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;

la durata delle istruzioni è 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 aggiunge 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. Come se in quel caso si entrasse in un ciclo apposito e poi si continuasse con il comportamento ordinario;

l'ingresso nello stato di attesa per HALT o bus request è gestito come un ciclo assestante ognuno.

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 puntata da *nn* un valore a 16 bit che viene caricato in IX. L'istruzione è composta da 4 byte: un suffisso, l'opcode e due byte che contengono la 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 suffisso 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 bassa e alta 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 delle FSMs che ne deriva è la seguente: una FSM Master che in caso di bus request blocca il funzionamento delle FSMs sottostanti; una FSM Main 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 Main.

D'aiuto alla FSM Main 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 avverte 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 dell'ultimo T-cycle di un ciclo macchina, che lo segnala alzando un flag, chiamato MLAST;
- un rivelatore di transizione negativa di MLAST, che segnala per un ciclo di CLK che è appena finito l'ultimo T-cycle di un ciclo macchina. Avverte così la FSM Master di poter servire una richiesta del bus. Il segnale di chiama MLAST_NEDGE;
- un riconoscitore di prefissi, che avverte la FSM Main se è necessario fare il fetch del secondo byte come opcode per mezzo del segnale EXT_IR.

Il comportamento della FSM Main, specialmente per la durata e le operazioni svolte durante la fase di execute, è dettato da un gruppo di otto segnali chiamati FLOWCTRL. Questo gruppo è generato dall'unità adibita alla decodifica chiamata DECODER. L'unità è completamente combinatoria e viene attivata durante le fasi di decode ed execute e gestisce i multiplexer e i segnali di controllo in uscita dal gruppo di controllo. Per far questo usa lo stato della macchina Main, utile per scandire le varie fasi, l'istruzione dall'IR e dalla coppia di FFs NMIIF e INTIF.

Le fasi di esecuzione della gestione degli interrupt sono sempre eseguite dagli stessi stati della FSM Main che si occupano anche della normale esecuzione. Per cui con i due flag si segnala al DECODER che deve dare i segnali per l'esecuzione della gestione delle interruzioni. I flag vengono attivati alla fine delle fasi di interrupt acknowledge e vengono resettati alla successiva fase di fetch che segnala la fine della gestione.

A. Implementazione del ciclo principale

Per rappresentare i diagrammi di stato userò la seguente convenzione:

gli stati sono rappresentati per mezzo di ellissi e le transizioni con frecce su cui è riportata la condizione;

sotto agli stati, solitamente in verde, sono riportate le uscite che vengono attivate in quello stato. In arancio invece sono evidenziate quelle uscite attivate che sono utili alla struttura annidata delle macchine. In fucsia le uscite che hanno un comportamento particolare; alcuni insieme di stati sono raggruppati in rettangoli dai bordi smussati che presentano lo stesso comportamento di un normale stato;

per non appesantire troppo di transizioni il diagramma, alcuni salti a stati o gruppi vengono indicati con una freccia verso il nome della destinazione generalmente azzurro;

in diagrammi in cui è importante la relazione con i cicli di clock, questi sono evidenziati da sezioni viola che riportano il nome del T-cycle corrispondente e lo stato di CLK.

Il ciclo Main dello Z80X è gestito da due FSM: Master e Main.

Master

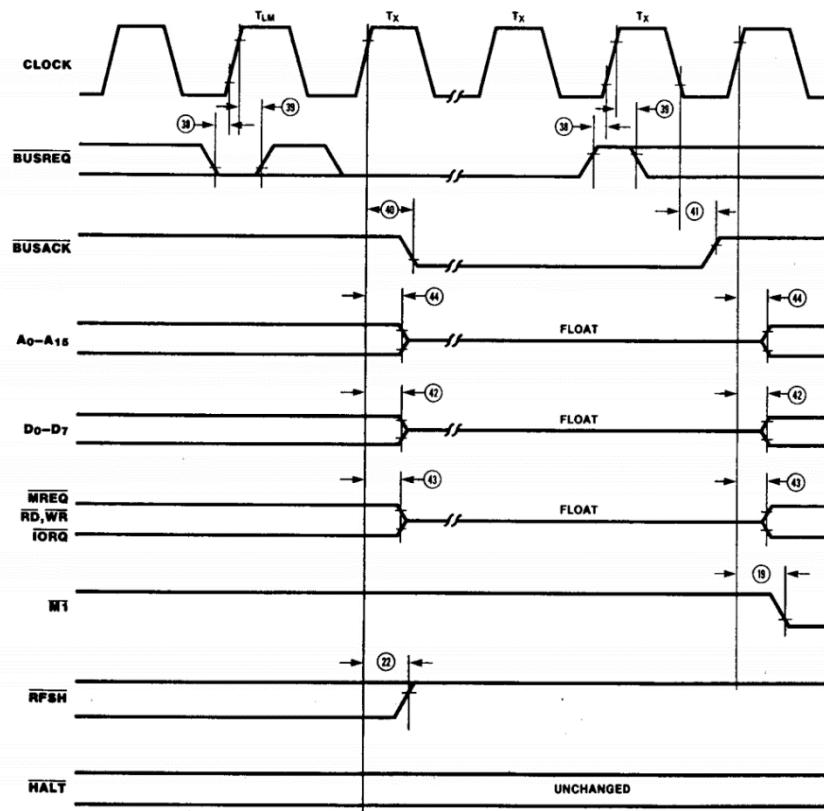


FIGURA 3-19 - BUS REQUEST/ACKNOWLEDGE CYCLE [22]

Master gestisce il comportamento del microprocessore quando si verifica una richiesta del bus e presenta solamente due stati: RUNNING e BUSREQUEST. La macchina è temporizzata solo sul fronte positivo di CLK, poiché le transizioni che deve fare avvengono solo su quell'evento a differenza di Main e delle altre che sono temporizzate su entrambi i fronti.

Durante il normale funzionamento di Z80X, Master si trova nello stato RUNNING. Come si vede dalla Figura 3-19 - BUS request/Acknowledge Cycle , dopo il ciclo di clock finale di un qualsiasi ciclo macchina, quindi quando si attiva MLAST_NEDGE, se l'ingresso nBUSREQ è attivo Master passa allo stato BUSREQUEST. In questo stato blocca il funzionamento della FSM Main, disattiva forzatamente i segnali nM1 e nRFSH e attiva tutti i segnali che mettono i bus in alta impedenza. Contemporaneamente segnala alla periferica richiedente che la richiesta è stata servita attivando nBUSACK.

La macchina esce dallo stato BUSREQUEST solamente nel se nBUSREQ è disattivato tornando allo stato RUNNING e facendo ripartire la FSM Main.

Questa configurazione permette di fermare il ciclo di funzionamento di Z80X ripartendo dallo stesso stato senza avere troppi stati aggiuntivi e permette di dare alle richieste la priorità massima.

Main

La FSM Main è la macchina che supervisiona il comportamento dello Z80X e può cambiare di stato solamente quando Master è in RUNNING.

La macchina controlla altre sottomacchine, dette anche μ FSM abbrev. di Micro FSM, attraverso due segnali TRIG e DONE seguiti dal suffisso della macchina a cui fanno riferimento.

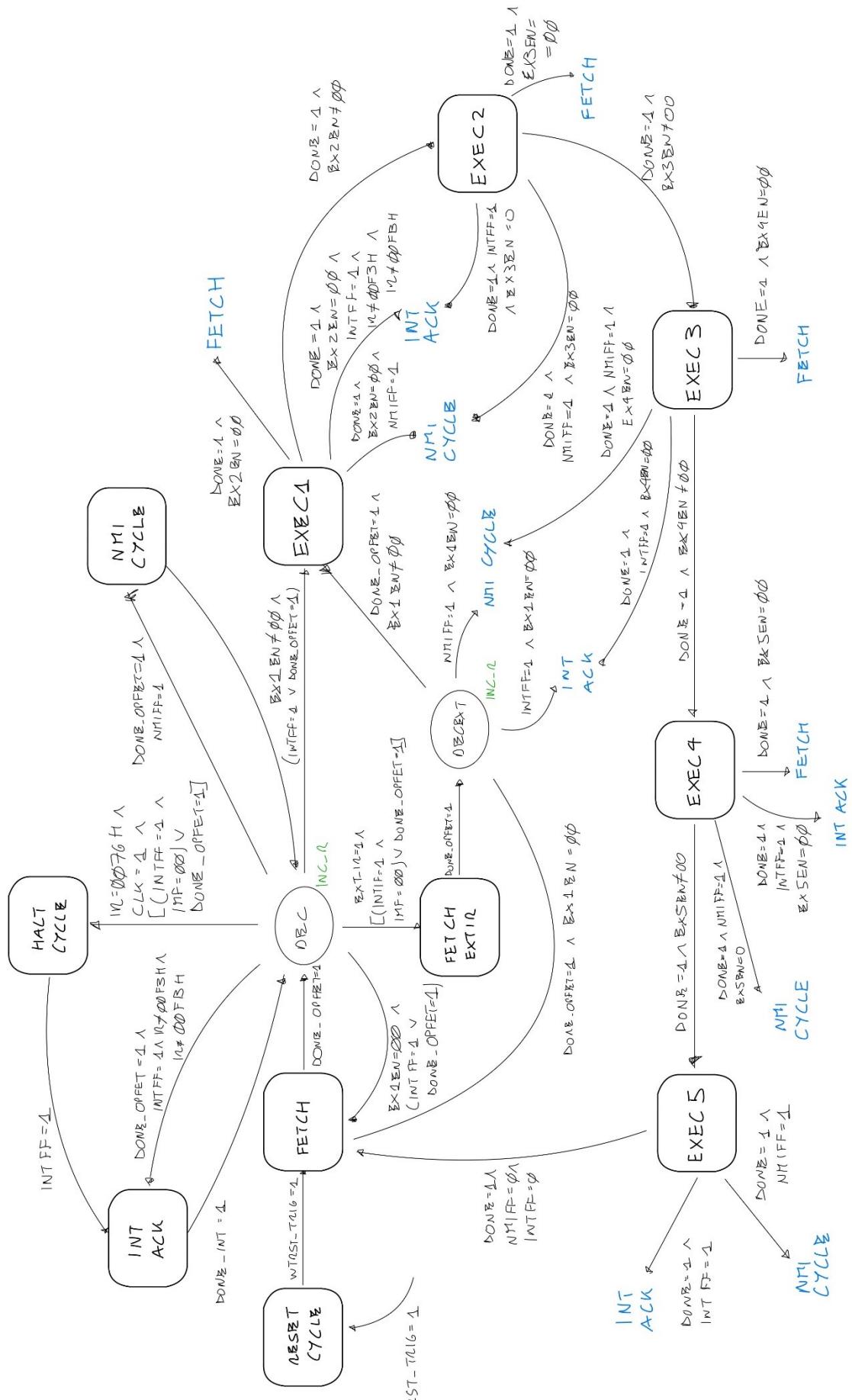


FIGURA 3-20 - DIAGRAMMA DI STATO DELLA FSM MAIN CON RAGGRUPPATI GLI STATI AFFINI

Main presenta un ciclo ordinario che può essere riassunto in tre sezioni principali: fetch – decode – execute.

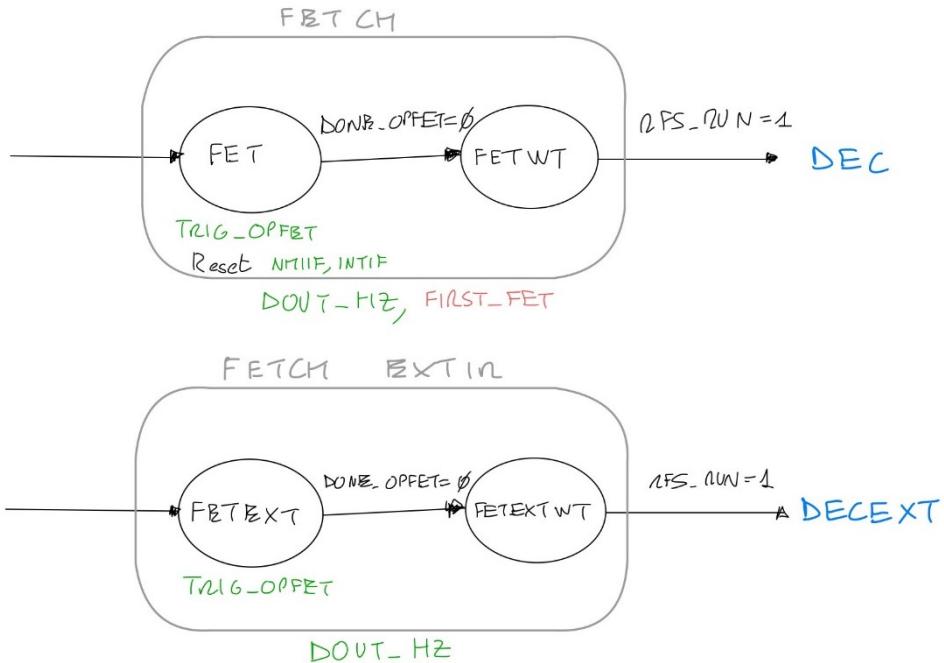


FIGURA 3-21 - DIAGRAMMI DI STATO DEI CICLI FETCH E FETCH EXTIR

Le fasi di fetch – decode possono essere singole o estese e sono raggruppate in due gruppi.

Entrambi i gruppi presentano la stessa struttura: prima uno stato, FET, che avvia la μ FSM per il fetch, chiamata OPFET, poi uno stato, FETWT, che attende la conclusione del ciclo di fetch e l'inizio del refresh seguito dallo stato di decodifica, DEC. Però i due gruppi differiscono poiché nel secondo caso, IR non viene resettato prima della lettura del dato così da ottenere l'istruzione estesa. Per questo gli stati del secondo gruppo presentano l'interfisso -EXT- che ne segnala la differenza.

Negli stati DEC e DECEXT, che eseguono la fase di decode, si attende che la macchina del fetch concluda completamente il suo ciclo e vengono incrementati sia PC che R ma non vengono svolte operazioni sull'esterno. Nel frattempo viene abilitato il DECODER che leggendo l'IR genera il gruppo FLOWCTRL. Fanno parte di questo gruppo cinque segnali chiamati EXiEN, abbrev. di Execution i Enable, in cui la i è sostituita dall'indice della fase di esecuzione corrispondente, assieme ai segnali IOnMEM, RDnWR, DONE_EX.

Può avvenire che un'istruzione sia istantanea cioè che l'operazione richiesta sia svolta direttamente nella fase di decode che quindi non coinvolge nessuna operazione sull'esterno. Un esempio sono le istruzioni di caricamento di un registro in un altro come *LD A, B* oppure le operazioni aritmetiche a 8 bit come *ADD A, B*. In questo caso, la macchina non entra nella fase di execute e torna direttamente al fetch.

La fase di esecuzione è formata da 5 blocchi elementari che sono identici a meno dell'indice che ne determina l'ordine. Sono cinque poiché al massimo lo Z80, e lo Z80X di conseguenza, esegue al massimo cinque operazioni R/W e di pura esecuzione differenti.

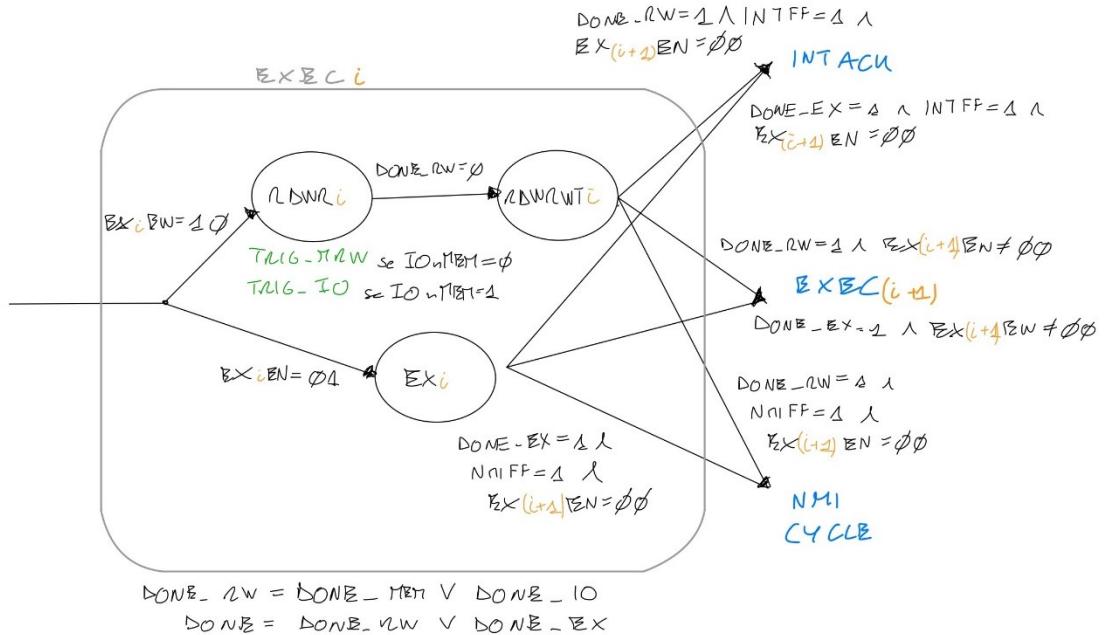


FIGURA 3-22 - DIAGRAMMI DI STATO DEI CICLI EXiEN, N INDICA L'INDICE DELLA FASE DI ESECUZIONE CORRISPONDENTE

Nei blocchi elementari ci sono due percorsi paralleli: uno gestisce le sottomacchine per la lettura o scrittura, mentre l'altro è uno stato di attesa di un'esecuzione generica, svolta internamente alla CPU senza coinvolgere l'esterno. Il primo è formato da due stati RDWRi e RDWRWTi, con i che indica sempre l'indice della fase. Nel primo stato avviene l'avvio della sottomacchina per le operazioni della memoria, MEMRDWR, se il segnale IOnMEM di FLOWCTRL è disattivato al contrario viene avviata la macchina per le operazioni sulle periferiche, IORDWR. Lo stato successivo attende il completarsi del ciclo ed esegue i controlli sui FFs degli interrupt e su EX(i+1)EN così da indirizzare lo stato verso il proseguimento corretto dettato da DECODER.

Lo stato di esecuzione EXi, con i sempre indice della fase, è regolato da un contatore che si incrementa ad ogni evento di CLK scandendo le azioni all'interno dello stato. Il DECODER attiva il segnale DONE_EX al raggiungimento di valori specifici che corrispondono alla loro effettiva durata, definita dall'istruzione che si sta svolgendo, e all'uscita dallo stato EXi. La selezione di un percorso od un altro è determinata dal valore di EXiEN che è un vettore di due bit in cui il più significativo abilita il percorso di R/W mentre l'altro il percorso di esecuzione interna. Se il vettore è pari a 00, quindi nessuna delle due è stata abilitata, significa che l'istruzione è arrivata al termine e si può procedere con la successiva fase di fetch,

richiudendo il ciclo principale.

Il caso dell'istruzione istantanea avviene con EX1EN = 00 per cui non è abilitato nessun percorso nella fase di esecuzione successiva.

Dall'ultima fase di esecuzione, EXEC5, non è presente nessun'altra fase successiva per cui la macchina torna alla fase di fetch.

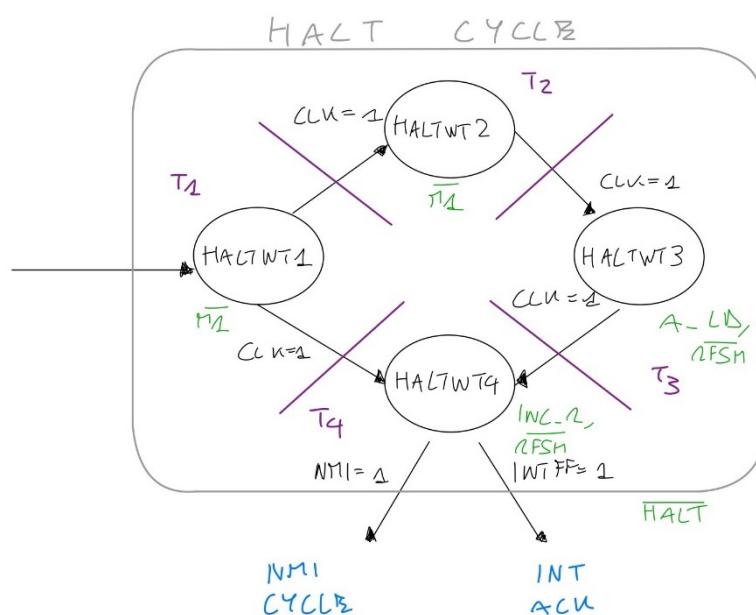
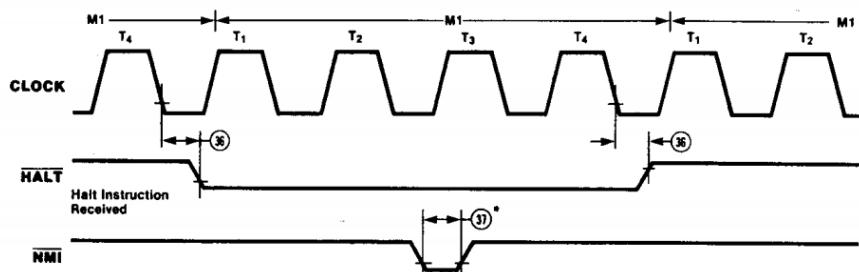


FIGURA 3-23 - DIAGRAMMI DI STATO DEL CICLO HALT CYCLE

Un'istruzione particolare è HALT che vale 0076H. Quest'istruzione viene eseguita direttamente dallo stato DEC che la riconosce e mette la macchina nel ciclo di sosta, HALT CYCLE. In questo ciclo ci sono quattro stati, uno per ogni T-cycle, che fanno eseguire allo Z80X delle operazioni NOP, abbrev. di No Operation, in cui avviene solamente il refresh della RAM. Quando la CPU entra in questo ciclo attiva il segnale nHALT per comunicare che è in attesa dall'esterno. Lo Z80X può uscire da questo stato solo per mezzo di un NMI o di un INT se abilitato oppure con nRESET. La lettura dei corrispondenti FFs viene fatta solo nel quarto ed ultimo stato del ciclo.

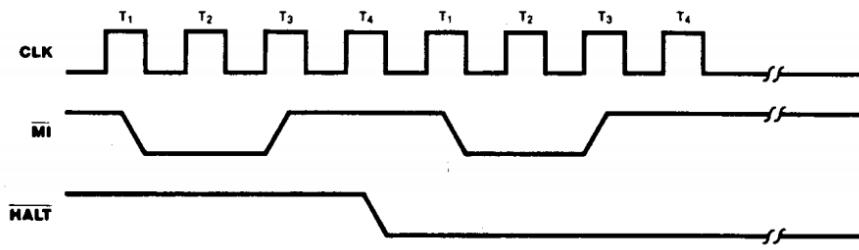


FIGURA 3-26 - POWER-DOWN ACKNOWLEDGE [22]

La presenza del ciclo HALT è utile per mettere in stand-by la CPU mantenendo comunque la RAM aggiornata.

Si può inoltre mettere la CPU in uno stato di power-down così da farle consumare meno corrente. Per far ciò, alla fine di un ciclo di HALT si deve togliere il segnale di CLK. In questo modo la CPU consuma il valore minimo di corrente ma non aggiorna la RAM. Per uscire da questo stato si fa allo stesso modo del precedente fornendo però prima il segnale di CLK.

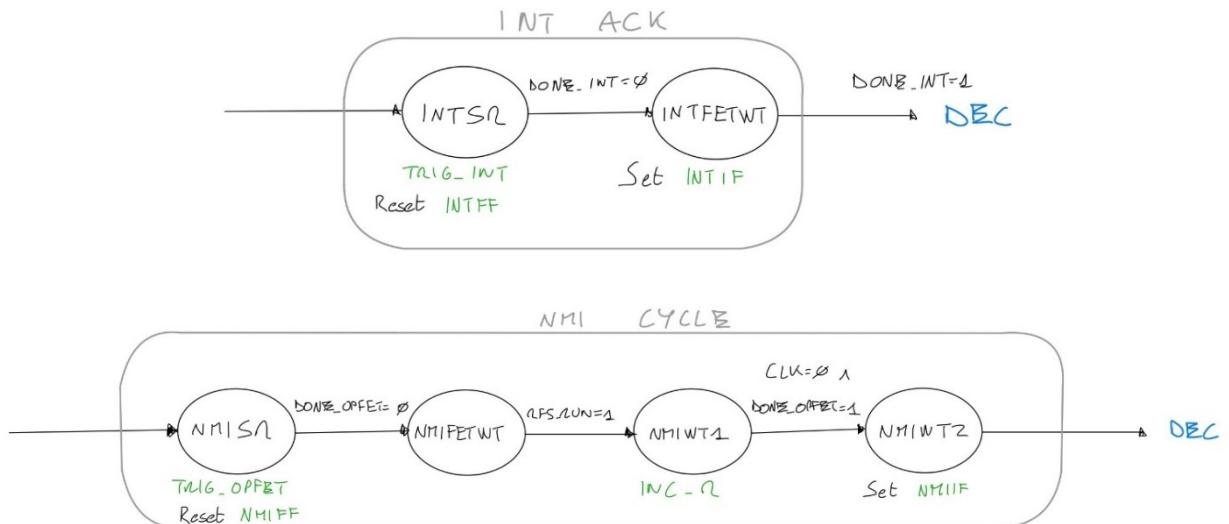


FIGURA 3-25 - DIAGRAMMI DI STATO DEI CICLI NMI CYCLE E INT ACK

Durante l'esecuzione di un INT in Mode 0, le operazioni di decode ed execute vengono svolte allo stesso modo di quelle normali.

Nel caso ci si trovi in uno stato terminale cioè DEC e DECEXT con EX1EN = 00 oppure RDWRWTi e EXi con EX(i+1)EN = 00, si può eseguire il servizio degli interrupt. Si servono prima gli NMI, per cui se NMIFF è attivato si passa al gruppo che gestisce la loro routine di servizio.

Nel primo stato, NMISR, viene avviato un ciclo di fetch, di cui viene ignorata l'istruzione recuperata, e viene anche resettato il flag NMIFF. Nel secondo stato, NMIFETWT, si attende l'inizio della fase di refresh e nel successivo, NMIWT1, si incrementa R attendendo la fine del fetch. L'ultimo stato, NMIWT2, serve per settare il flag NMIIF e attendere il ciclo

aggiuntivo al fetch, come di vede in Figura 3-27 - Non-Maskable Interrupt Request Operation. Alla fine di questa fase, si ritorna in DEC e il DECODER guida l'esecuzione di RST 0066H.

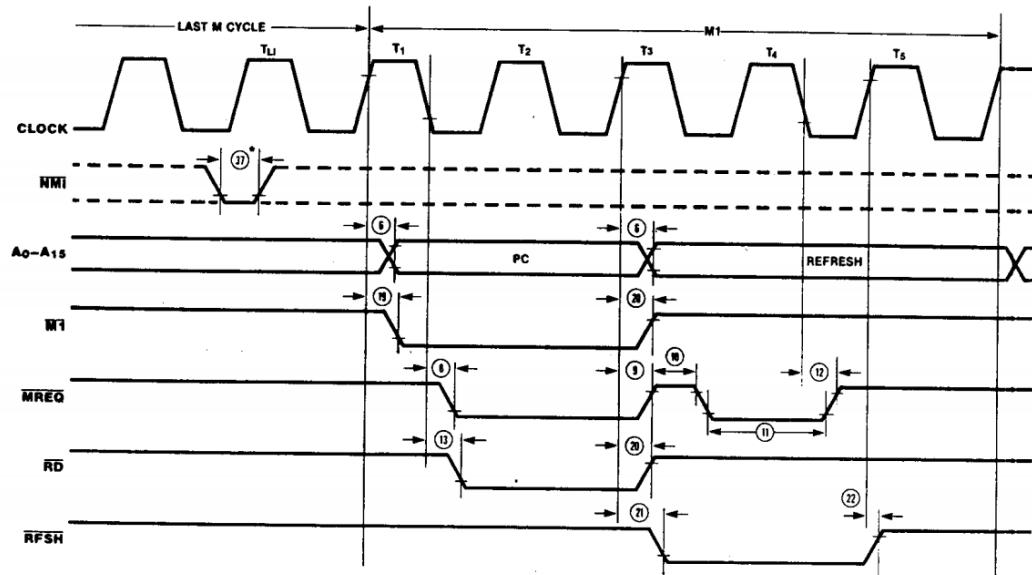


FIGURA 3-27 - NON-MASKABLE INTERRUPT REQUEST OPERATION [22]

Nel caso in cui non siano avvenuti NMI, si controllano gli INT. Se ne sono avvenuti e l'istruzione appena terminata non è EI, IR = 00FBH, o DI, IR = 00F3H, si va nel ciclo INT ACK. Questo avvia il ciclo di interrupt acknowledge nello stato INTSR che resetta anche il flag INTFF per poi passare allo stato INTFETWT che attende la fine del ciclo avviato e poi passa a DEC per l'esecuzione corrispondente.

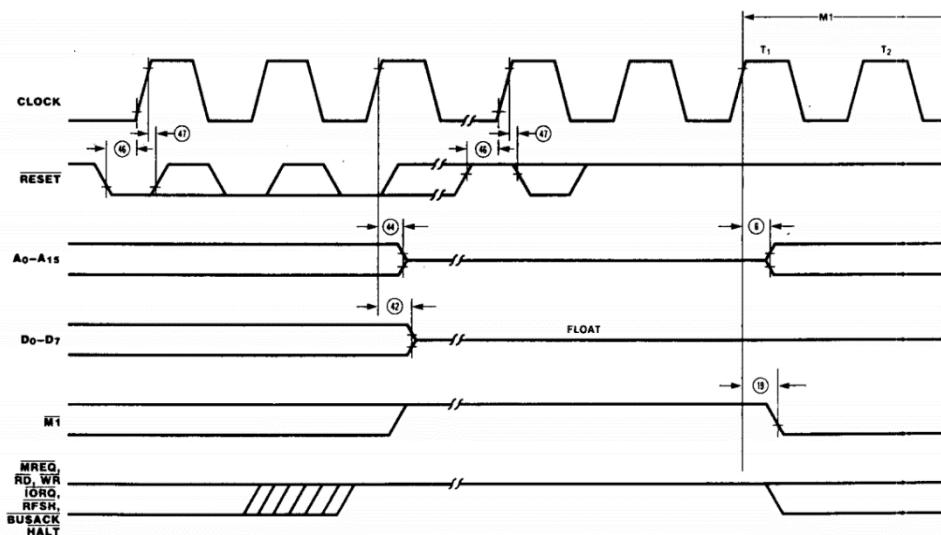


FIGURA 3-28 - RESET CYCLE [22]

Per gestire il reset, la FSM Main sfrutta un contatore. Il contatore serve perché la CPU deve accettare il comando di reset ed entrare in uno stato opportuno solo se il segnale nRESET rimane attivo per almeno tre cicli consecutivi di CLK. Il contatore conta questi cicli e al

raggiungimento del terzo attiva un segnale di trigger, RST_TRIG. Alla sua attivazione, la FSM Main entra nel gruppo RESET CYCLE, a prescindere dallo stato corrente, in cui vengono disattivati tutte le uscite e attivati i segnali A_HZ e DOUT_HZ.

La FSM entra nello stato WTRSET in cui si attende che il segnale nRESET si disattivi. Alla sua disattivazione si entra nello stato RSET in cui la CPU attende un ciclo di CLK prima di ripartire con la fase di fetch. Durante questo stato vengono attivati i segnali RST μ FSM, che resetta tutte le sottomacchine rendendole pronte a eseguire i loro cicli, e RSTREG, che resetta tutti i registri.

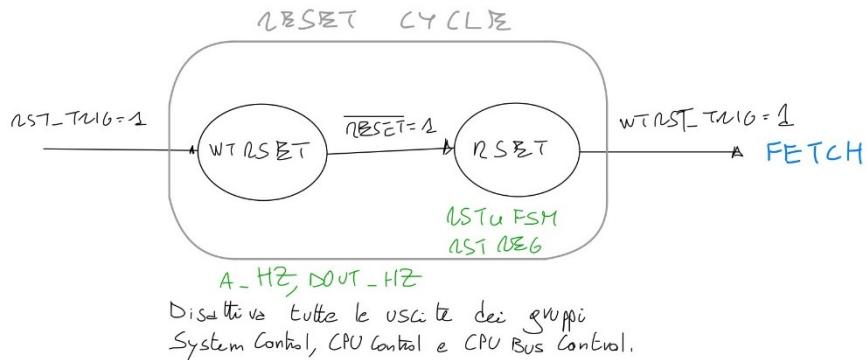


FIGURA 3-29 - DIAGRAMMA DI STATO DEL CICLO RESET CYCLE

B. Implementazione delle sottomacchine

Tutte le µFSM hanno lo scopo di eseguire un ciclo preciso svolto dalla CPU e di aggiornare dei valori internamente a Z80X. I cicli in esame sono gli unici che durante il normale funzionamento dello Z80X coinvolgono l'esterno dell'entity. Per cui nella descrizione che ne segue terrò conto del loro port, del ciclo che devono svolgere e dell'FSM che lo implementa e del loro compito all'interno dello Z80X.

Tutte le sottomacchine hanno uno schema e un comportamento comune.

Le µFSM presentano tutte la coppia di segnali TRIG e DONE, seguite dal suffisso della macchina, per il controllo all'interno della struttura annidata delle FSMs. Si aggiunge il segnale STOP che se attivato impedisce alla macchina di passare allo stato successivo.

Le sottomacchine presentano uno stato di attesa IDLE in cui se il segnale di TRIG è attivo e il CLK è a 1 iniziano il loro ciclo rimanendo insensibili ad altri stimoli su TRIG sino alla fine del ciclo. Per cui quando la macchina è in attesa e la FSM Main attiva il segnale TRIG inizia il ciclo collegato.

Si possono far andare le µFSM in continuazione tenendo TRIG attivo così la sottomacchina non entra mai nello stato IDLE ma dall'ultimo stato del ciclo passa direttamente al primo ricominciando.

La µFSM segnala la fine del proprio ciclo con il segnale DONE che rimane attivo anche durante l'attesa. In questo modo la FSM Main può sfruttare due eventi per la temporizzazione: l'inizio del ciclo richiamato quando DONE si disattiva e la fine del ciclo quando si attiva. Per cui Main mantiene il segnale di TRIG attivo fintantoché non vede disattivarsi DONE per avere la certezza di aver avviato la sottomacchina correttamente.

Il segnale di DONE viene generato quando la macchina si trova nello stato di attesa o negli ultimi stati del ciclo per permettere una corretta lettura del segnale da parte di Main. Il motivo è che siccome Main è stata implementata come una macchina di Moore presenta mezzo ciclo di CLK di ritardo sulle variazioni dei suoi segnali.

i. Instruction Opcode Fetch, *OPFET*

La macchina svolge il ciclo di fetch, che dura di base 4 T-cycles, in cui la CPU legge dalla memoria l'opcode all'indirizzo puntato da PC e lo carica in IR. Dopodiché esegue il refresh della RAM mettendo sul bus l'indirizzo di refresh, contenuto in R, e attivando gli opportuni segnali.

Con il segnale A_LD carica in A LATCH il PC o R selezionandoli con il segnale PCnRFSH. Mentre con il segnale DIN_LD carica in IR il valore di DIN, per cui è collegato al segnale SHIFT del registro.

Per permettere la sovrapposizione delle fasi di fetch e decode, la macchina fornisce alla FSM Main un ulteriore segnale, RFS_RUN, che si attiva nello stato precedente l'inizio della fase di refresh e rimane attivo fino alla fine della fase. L'attivarsi di questo segnale indica alla macchina di passare dallo stato di attesa a quello di decodifica: da FETWT a DEC o da FETEXTWT a DECEXT.

La macchina oltre al comportamento descritto dal diagramma in Figura 3-30 - DIGRAMMA DI STATO DELLA MFSM OPFET, resetta IR nello stato precedente il caricamento cioè in CHKWT, che è anche lo stato in cui campiona nWAIT per entrare o meno nel ciclo di attesa. Il RSTIR viene mascherato con il segnale FIRST_FET che viene attivato solo in FET. La macchina carica invece il valore letto in IR durante RD.

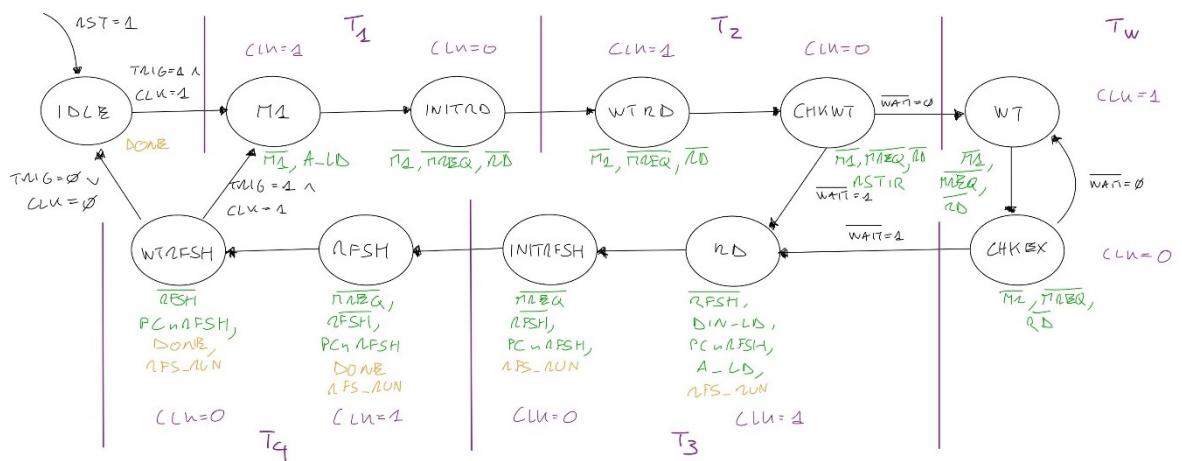


FIGURA 3-30 - DIGRAMMA DI STATO DELLA MFSM OPFET

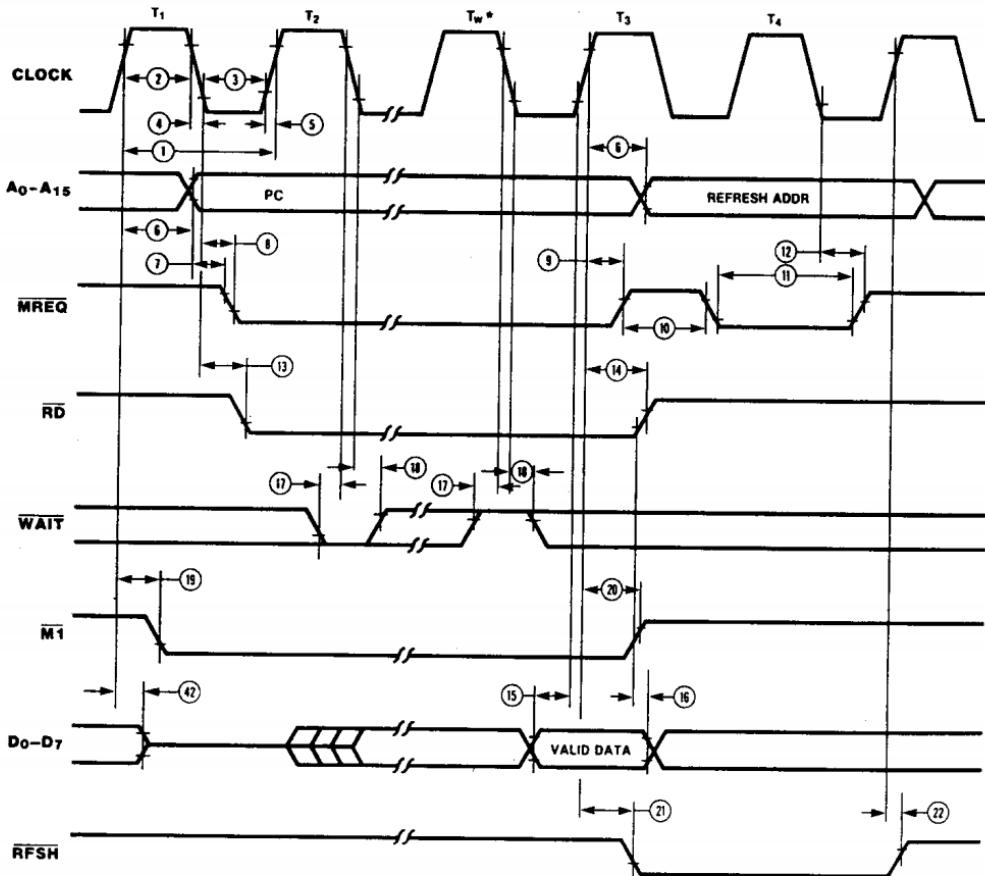


FIGURA 3-31 - INSTRUCTION OPCODE FETCH [22]

ii. Memory Read or Write Cycles, *MEMRDWR*

La macchina svolge il ciclo di lettura o scrittura su memoria, che dura di base 3 T-cycles.

In questo ciclo la CPU legge dalla memoria un dato all'indirizzo puntato dal valore sul bus interno DATA16 e lo carica in MDR. Oppure scrive all'indirizzo presente sul bus DATA16 il valore presente sul bus interno DATA.

Il verso dell'operazione è determinato dal segnale RDnWRFF, come si vede dal suffisso -FF è la versione campionata da un FF del segnale d'ingresso RDnWR. Il FF campiona il valore dell'ingresso mentre il segnale ENFF è attivo, cioè durante IDLE, WRA o RDWR.

Il ciclo della macchina rimane pressoché invariato in base al valore di RDnWRFF a meno della coppia di stati INITRD e WRD. Il primo avvisa la memoria che sta avvenendo una lettura e il secondo carica il valore di DATA sul DOUT LATCH. Per il resto cambia solamente che nRD sia attivato, nel caso in cui RDnWRFF = 1, o alternativamente che nWR sia attivato, con RDnWRFF = 0. Nel diagramma di Figura 3-32 - Diagramma di stato della uFSM MEMRDWR questo è mostrato per mezzo del segnale fittizio RDWR.

La scrittura del valore sul bus D avviene appunto nello stato WRD mentre quella dell'indirizzo nello stato WRA con l'attivazione di A_LD. La lettura e il caricamento di MDR avvengono nello stato RDWR se RDnWRFF = 1.

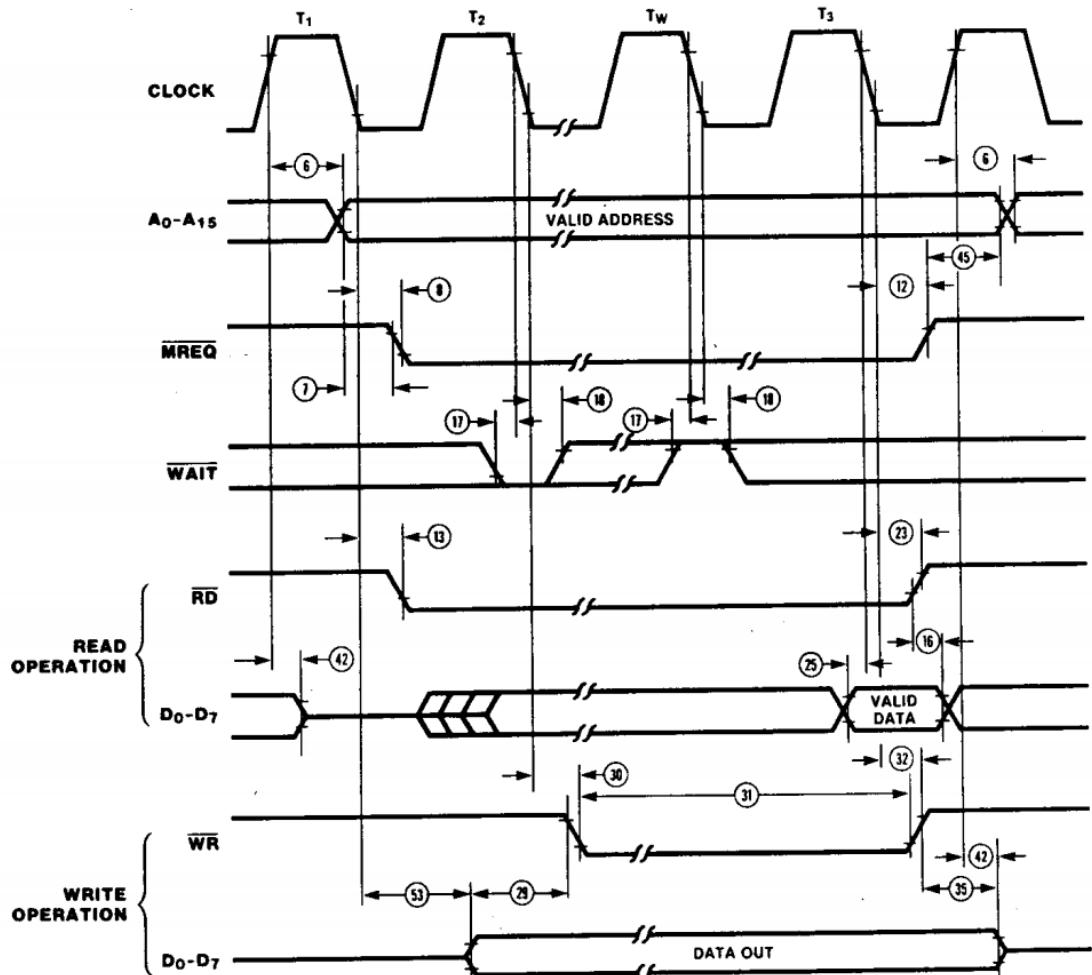


FIGURA 3-33 - MEMORY READ OR WRITE CYCLES [22]

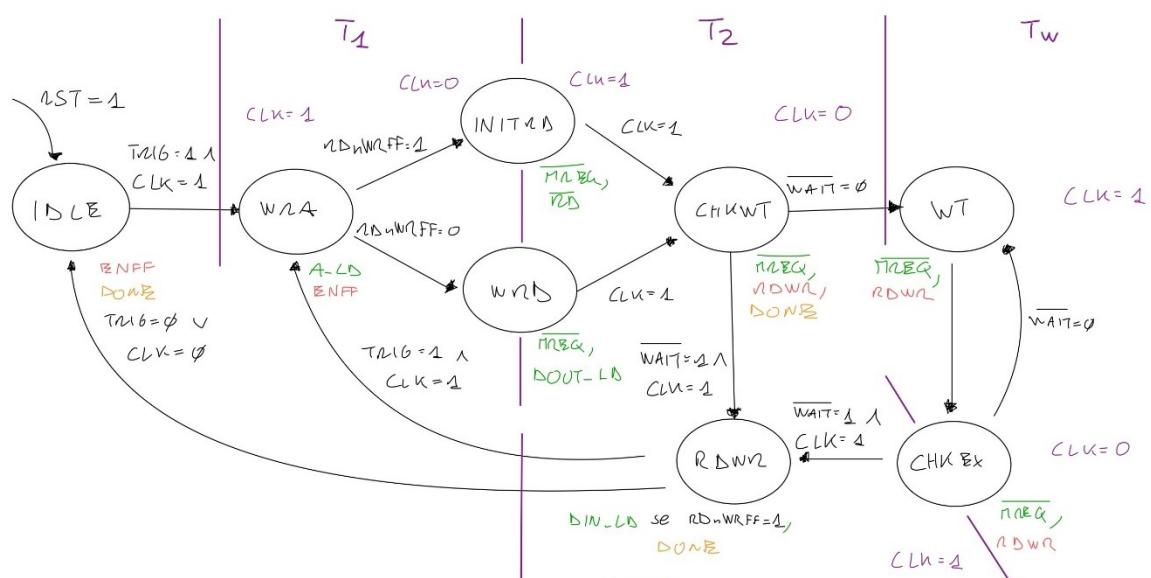


FIGURA 3-32 - DIGRAMMA DI STATO DELLA UFSM MEMRDWR

iii. Input or Output Cycles, *IORDWR*

La macchina svolge il ciclo di lettura o scrittura sulle periferiche, che dura di base 4 T-cycles.

La macchina è la controparte per gli I/O di MEMRDWR per cui ha lo stesso funzionamento a meno di avere un ciclo di attesa in più, WA, in cui non avviene nessun cambiamento delle uscite.

Inoltre controlla nIORQ invece di nMREQ e attiva i segnali nRD o nWR avviene nello stesso stato e non sfalsati. Per questo il ciclo della macchina è esattamente lo stesso indifferentemente dal valore di RDnWRFF.

Di conseguenza, la CPU legge un dato all'indirizzo puntato dal valore sul bus interno DATA16 e lo carica in MDR. Oppure scrive all'indirizzo presente sul bus DATA16 il valore presente sul bus interno DATA.

Il verso dell'operazione è determinato dal segnale RDnWRFF. Il FF campiona il valore dell'ingresso mentre il segnale ENFF è attivo, cioè durante IDLE, WRA o RDWR. Come per MEMRDWR, l'attivazione di nRD o nWR in base a RDnWRFF è mostrata per mezzo del segnale fittizio RDWR nella Figura 3-34 - Diagramma di stato della uFSM IORDWR.

La scrittura del valore sul bus D avviene appunto nello stato WRD se RDnWRFF = 1 mentre quella dell'indirizzo nello stato WRA con l'attivazione di A_LD. La lettura e il caricamento di MDR avvengono nello stato RDWR se RDnWRFF = 1.

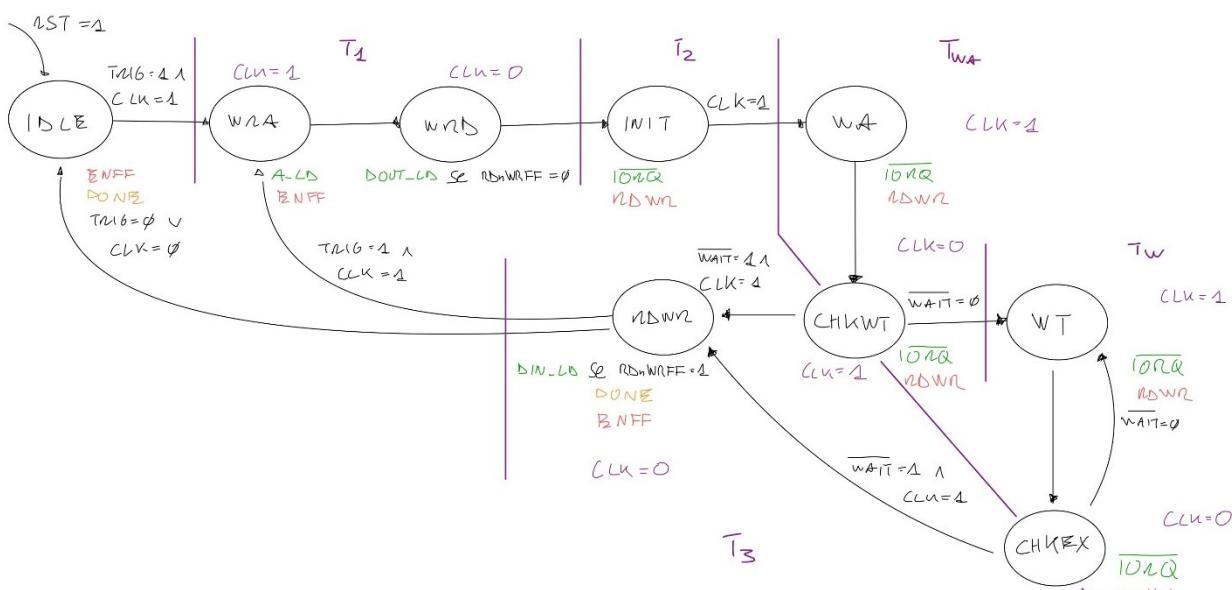


FIGURA 3-34 - DIGRAMMA DI STATO DELLA UFSM IORDWR

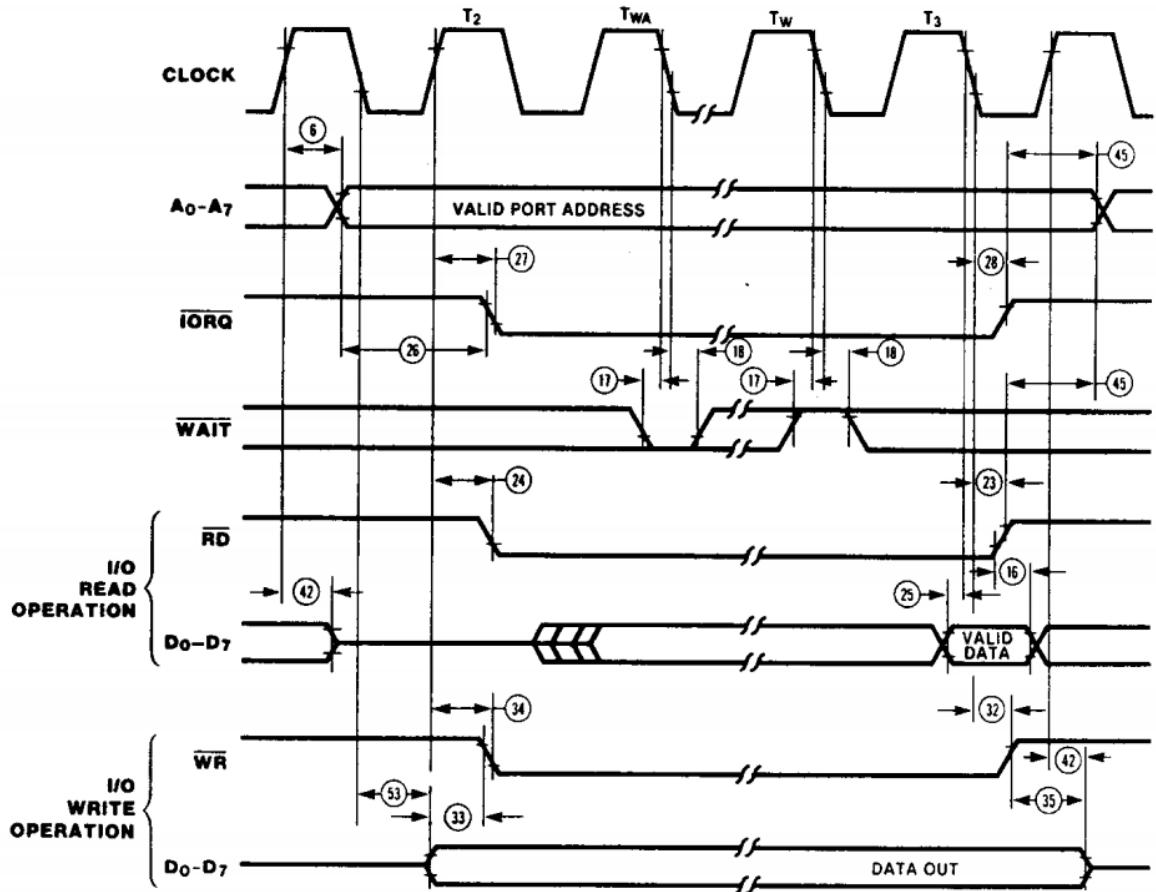


FIGURA 3-35 - INPUT OR OUTPUT CYCLES [22]

iv. Interrupt Request/Acknowledge Cycle, INTRQ

La macchina svolge il ciclo di interrupt acknowledge in caso sia avvenuto un INT, che dura di base 5 T-cycles.

La CPU esegue un ciclo simile a quello di fetch in cui carica il valore di PC anche se non è utile. Inoltre invia il segnale di interrupt acknowledge attraverso l'attivazione della combinazione di segnali nIORQ e nM1. Dopodiché legge dal bus il valore dell'istruzione da svolgere se in Mode 0 o la parte bassa dell'indirizzo se in Mode 2. In ogni caso il valore letto viene caricato in IR e poi sarà il DECODER a utilizzare sapientemente IR in base alla modalità impostata.

A differenza del ciclo di fetch, presenta due stati di attesa in più, TWA1 e TWA2. In realtà la macchina attende davvero in T2 e TWA1 mentre in TWA2 attiva il segnale nIORQ.

La scrittura dell'indirizzo avviene nello stato WRA con l'attivazione di A_LD. La lettura e il caricamento di MDR avvengono nello stato RD.

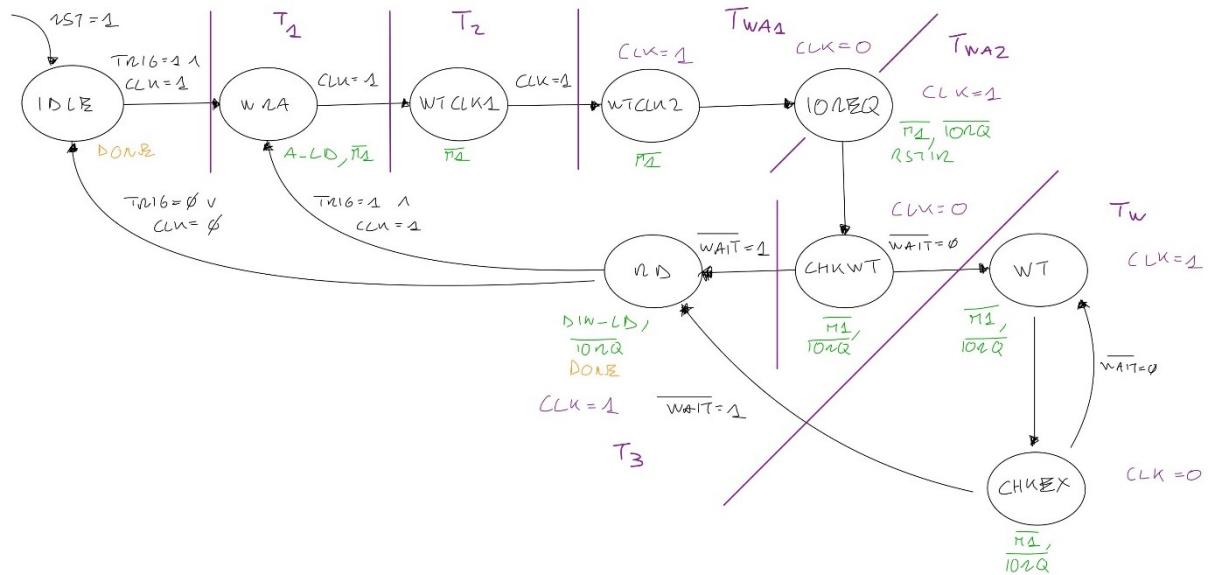


FIGURA 3-36 - DIGRAMMA DI STATO DELLA UFSM INTRQ

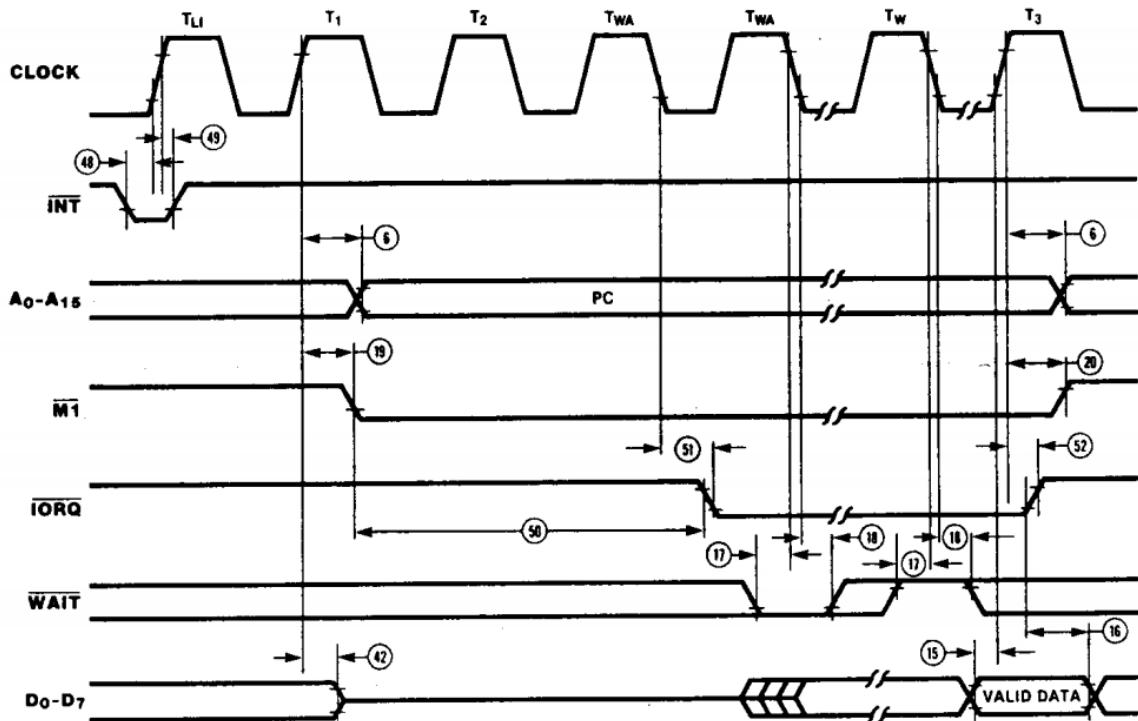


FIGURA 3-37 - INTERRUPT REQUEST/ACKNOWLEDGE CYCLE [22]

C.ALU e incrementers

L'ALU e due INC/DEC sono gli elementi principali di calcolo dello Z80X. Nell'architettura dello Z80 compare solo l'ALU mentre dalle informazioni sul reverse engineering si vede che l'ALU in realtà è a 4 bit e vi è un INC/DEC a 16 bit. Per semplicità di progetto, per le operazioni di incremento e decremento a 8 bit ho aggiunto un INC/DEC dedicato.

ALU

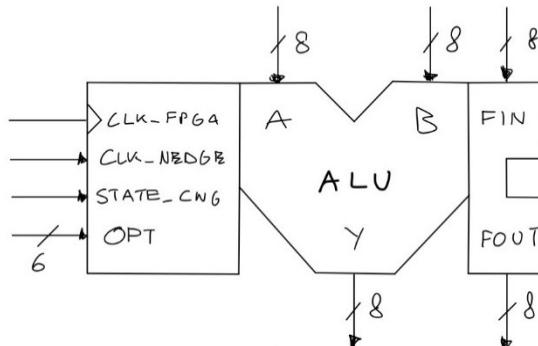


FIGURA 3-38 - SCHEMA DELL'ALU

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 che si attiva per un ciclo di clock se lo stato della FSM Main cambia.

L'implementazione dell'ALU considera sia le informazioni sull'organizzazione reale dello Z80 sia la semplicità di implementazione. Di conseguenza per le operazioni aritmetico-logiche, vi sono tre unità che lavorano in contemporanea. Una che calcola il risultato su 9 bit invece che 8 bit estendendo il segno, il risultato serve sia per generare il flag di overflow sia come risultato effettivo. Un'altra calcola la stessa cosa ma senza estendere il segno per cui il nono bit rappresenta il flag di carry out. Infine una a 5 bit che considera la parte bassa a 4 bit dei due operandi senza estensione del segno generando così il flag dell'half carry.

L'ALU svolge 27 differenti operazioni selezionabili per mezzo del vettore OPT, abbrev. di Option, attraverso codici riconducibili all'istruzione che effettua la chiamata. Nel caso in cui l'operazione selezionata non sia riconosciuta, l'ALU esegue un NOP.

Il primo gruppo contiene le 7 operazioni aritmetico-logiche a 8 bit e CP, abbrev. di Compare. Quest'ultimo viene implementato come una differenza che modifica i flag ma restituisce in

uscita il valore di A non modificato.

Le istruzioni vengono selezionate per mezzo della giustapposizione del prefisso 000- con il campo corrispondente, bit da 5 a 3, dell'istruzione.

Per la loro controparte a 16 bit, il DECODER prima fa eseguire l'operazione a 8 bit come richiesto e poi esegue l'operazione sugli altri 8 bit sempre con l'uso del carry in.

Il secondo gruppo contiene le 4 operazioni di rotazione dell'accumulatore. Queste operazioni vengono svolte per semplice riassortimento dei bit e calcola i bit del flag come se fosse una normale operazione aritmetico-logica. Il valore di OPT si genera con il prefisso 001- seguito dal campo corrispondente, bit da 5 a 3, dell'IR. Ad appendice a questo gruppo vi sono le 2 istruzioni di rotazione per cifre BCD, le istruzioni *RRD* e *RLD*.

Il terzo gruppo contiene 3 istruzioni di gestione dell'accumulatore: *DAA*, *CPL* e *NEG*.

DAA è l'abbrev. di Decimal Adjust Accumulator e serve a riportare in formato BCD l'accumulatore, se è stato generato per mezzo un'operazione aritmetica da due numeri BCD. Fa uso dei flag H, C ed N e dal reverse engineer si è visto che principalmente segue questa logica:

- se N è 1 tutte le operazioni successive sono di sottrazione invece che di somma;
- se la parte alta del risultato nell'accumulatore è maggiore di 9 o C è settato, viene sommato 60H;
- se la parte bassa del risultato nell'accumulatore è maggiore di 9 o H è settato, viene sommato 6H;
- nei casi previsti si eseguono entrambe le somme.

CPL è l'abbrev. di Complement ed esegue il complemento a 1 cioè la negazione logica dell'accumulatore mentre *NEG*, che è l'abbrev. di Negation, esegue il complemento a 2 cioè il cambiamento del segno.

Il quarto gruppo è formato dalle 3 istruzioni che effettuano operazioni sui bit: *BIT*, *RES* e *SET*.

BIT effettua solamente il test del bit selezionato per cui esegue un'AND tra il negato del valore da testare e una maschera di tutti 0 tranne un 1 nella posizione del bit corrispondente. Dopodiché il risultato non viene dato in uscita ma vengono modificati solo i flag in particolare il flag Z. Mentre *RES*, abbrev. di Reset, effettua un'AND tra il valore e una maschera con di tutti 1 tranne uno 0 nella posizione del bit corrispondente e *SET* effettua una OR tra il valore e una maschera con di tutti 0 tranne un 1 nella posizione del bit

corrispondente.

Il codice per la selezione si ottiene con la giustapposizione del prefisso 0111- seguito dal campo corrispondente, bit 7 e 6, della parte basse dell'IR.

L'ultimo gruppo contiene le 8 istruzioni che effettuano le operazioni di scorrimento e rotazione su registri diversi dall'accumulatore. Oltre alle 7 documentate ho aggiunta anche SLL che si inserisce perfettamente nei codici di selezione. Questi vengono creati dalla giustapposizione del prefisso 100- con il campo corrispondente, bit da 5 a 3, dell'IR.

INCDEC e INCDEC16

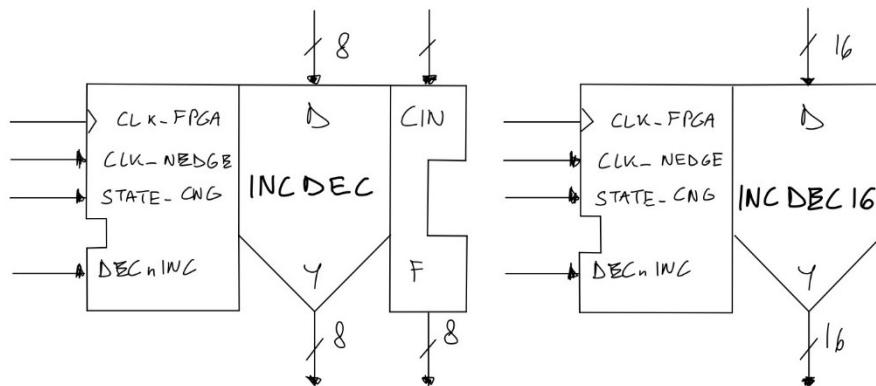


FIGURA 3-39 - SCHEMA DELL'INCDEC E INCDEC16

L'implementazione di INCDEC e INCDEC16 è la stessa a meno della differenza di lunghezza della parola. La selezione tra incremento e decremento avviene con la linea DECnINC che può essere collegata al bit 0 dell'IR che è 0 nel caso di incremento o è 1 altrimenti.

Le due entity presentano la stessa struttura a registri dell'ALU, per cui il valore del dato in ingresso viene campionato ad ogni attivazione di CLK_NEDGE e STATE_CNG ed il risultato viene solitamente letto in CLK_PEDGE.

INCDEC presenta anche un ingresso per il carry in poiché genera anche il proprio valore del registro di stato F e necessità del valore precedente di carry in.

D.Registri

Per l'implementazione dei registri ho guardato per lo più alla semplicità di utilizzo di questi all'interno delle operazioni piuttosto che alla completa adesione alle informazioni note sull'organizzazione dello Z80.

Per questo, come detto in precedenza, i registri SP, A-F, quelli d'indicizzazione e i general purpose non sono tutti nella stessa unità.

SP si trova isolato con un INC/DEC dedicato per facilitare e velocizzare le operazioni sul puntatore durante i *PUSH* e *POP*.

A ed F sono accoppiati perché vi è un sistema a FF che quando viene attivato il segnale EXAF, il che avviene durante l'istruzione *EX AF, AF'*, scambia i registri con la loro copia ombra allo stesso modo di come si fa con i registri general purpose e l'istruzione *EXX*.

Allo stesso modo i registri PC e I-R non sono assieme.

PC è isolato con un incrementer dedicato. Dal registro si può leggere il valore e metterlo sul bus a 16 bit oppure caricare un valore dal bus o il valore incrementato.

I ed R sono separati dagli altri e tra loro. Su I può avvenire la lettura sia come numero a 8 bit che come parte alta di un valore caricabile sul bus a 16 bit assieme a MDR o si può caricare un valore dal bus a 8 bit. Mentre R ha un incrementer dedicato, può essere caricato con un valore dal bus a 8 bit e usato come parte bassa di un numero a 16 bit preceduto da 0.

REGS

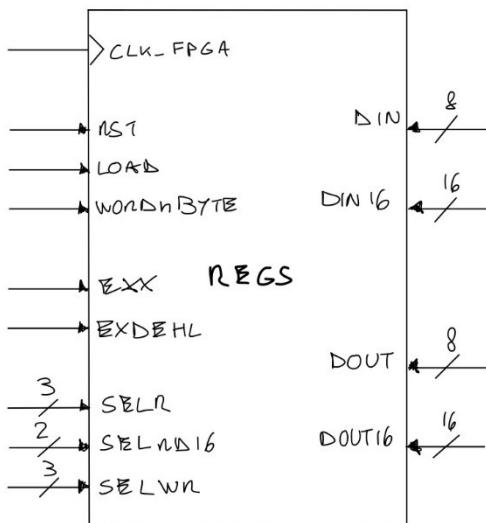


FIGURA 3-40 - SCHEMA DELL'ENTITY REGS

I registri general purpose B, C, D, E, H ed L sono contenuti nella stessa unità assieme ad una coppia di registri di appoggio W e Z. L'entity grazie a due FFs gestisce le due pagine di

registri per cui all'attivarsi del pin EXX, che corrisponde all'esecuzione dell'istruzione *EXX*, scambia le pagine e gestisce anche l'indirizzamento verso le coppie DE e HL che può essere scambiato all'attivarsi del pin EXDEHL, corrispondente all'istruzione *EX DE, HL*.

Data la forma dell'entity si può:

- leggere un registro a 8 bit selezionabile con il codice r corrispondente sul vettore SELRD;
- leggere un registro a 16 bit, ottenuto dalla giustapposizione di due registri contigui, cioè si può leggere BC, DE, HL e WZ, selezionabile con il codice dd corrispondente sul vettore SELRD16;
- scrivere un registro o una coppia di registri. Nel caso a 8 bit basta selezionare il registro con il codice r su SELWR, settare a 0 il segnale WORDnBYTE e attivare LOAD. Nel caso a 16 bit invece la selezione avviene con il codice dd sempre su SELWR, di cui vengono letti solo i due bit più significativi, e settando a 1 il segnale WORDnBYTE assieme a LOAD.

Viene permessa solo una scrittura alla volta per non incappare in problemi di interferenza delle due operazioni sullo stesso registro.

IXIY

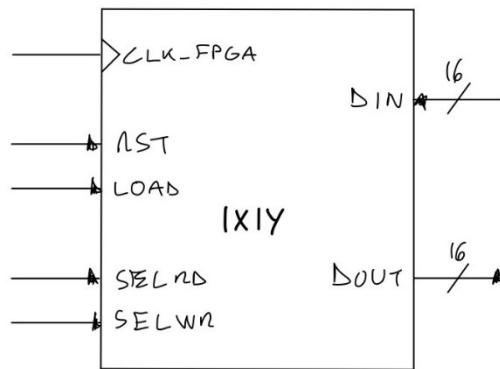


FIGURA 3-41 - SCHEMA DELL'ENTITY IXIY

La coppia di registri IX e IY è separata dagli altri registri per velocizzare le operazioni di indicizzazione.

Come REGS, permette di leggere un registro e di scriverne un altro contemporaneamente. La selezione avviene con i due segnali SELRD e SELWR rispettivamente.

E. Problemi noti

Quest'implementazione dello Z80 su FPGA presenta dei problemi, in alcuni casi facilmente risolvibili, di cui bisogna tenere conto durante l'utilizzo.

Per prima cosa bisogna far notare il rapporto tra le frequenze di CLK_FPGA e CLK. I due in questo caso sono in rapporto minimo pari a 20:1. Durante l'uso bisogna tenere conto che l'approssimazione dei latch con FFs temporizzati su CLK_FPGA vale fintantoché il rapporto tra le frequenze rimane elevato. Per cui se si volesse far andare Z80X fino ai 20MHz che sono garantiti nei datasheet delle ultime versioni CMOS dello Z80 [22] bisogna fornire un CLK_FPGA un segnale con frequenza almeno dieci volte maggiore.

L'istruzione *OUTI* è difettiva. L'istruzione esegue la copia di una locazione di memoria su una periferica incrementando l'indirizzo. L'istruzione nel decoder è eseguita dallo stesso blocco che esegue tutte le operazioni di copia di blocchi di memoria anche ripetuti cioè *OUTI*, *OTIR*, *OUTD*, *OTDR*, *LDI*, *LDIR*, *LDD* e *LDLR*. In questo insieme *OUTI* è l'unica che richiede 17 T-cycles contro i 16 delle controparti [22]. Per cui nello Z80X, l'istruzione viene eseguita come le altre con 16 T-cycles considerando la presenza del ciclo in più un errore.

Siccome la decodifica delle istruzioni è stata basata sul buon senso assieme alle informazioni presenti sul datasheet con l'intuizione della divisione in cicli simili, non è rispettata la reale divisione e durata dei singoli M-cycles.

Di conseguenza è sempre garantita la durata totale in T-cycles come da datasheet e spesso anche il numero di M-cycles corrispondenti senza però rispettare la vera divisione dei T-cycles per ogni M-cycles. Significa che lo Z80X messo a confronto con uno Z80 discreto, avrà la stessa durata delle istruzioni ma potrebbe non accogliere nello stesso momento le richieste del bus poiché i cicli macchina dello Z80X potrebbero iniziare e finire prima o dopo rispetto a quelli dello Z80. Questo problema è trascurabile poiché affligge solo il ritardo nel servizio delle richieste del bus. Il ritardo è al massimo di alcuni T-cycles e non crea danni al funzionamento complessivo.

In ultima analisi vi è l'occupazione di SLICEs dell'entity all'interno dell'FPGA.

La decisione di creare un'ALU con tre diverse sezioni per ogni operazione richiede uno spazio che può essere ridotto usandone solamente una assieme a più logica di controllo.

La struttura del DECODER occupa molto spazio a causa del controllo di tutte le condizioni che non avviene per mezzo di memorie ma di singole LUT all'interno delle SLICEs.

Capitolo 4 Memorie, interfacce verso l'esterno ed il controllore

Lo Z80 per funzionare deve essere connesso ad una memoria che contenga il programma da svolgere e i dati necessari. allo stesso modo lo Z80X deve essere connesso all'interno dell'FPGA con tutto il necessario per farlo funzionare, come le memorie. Per svolgere delle operazioni utili deve poter comunicare con l'utente e per avere un buon flusso di programmazione e di debug ci dev'essere un'interfaccia adatta e veloce.

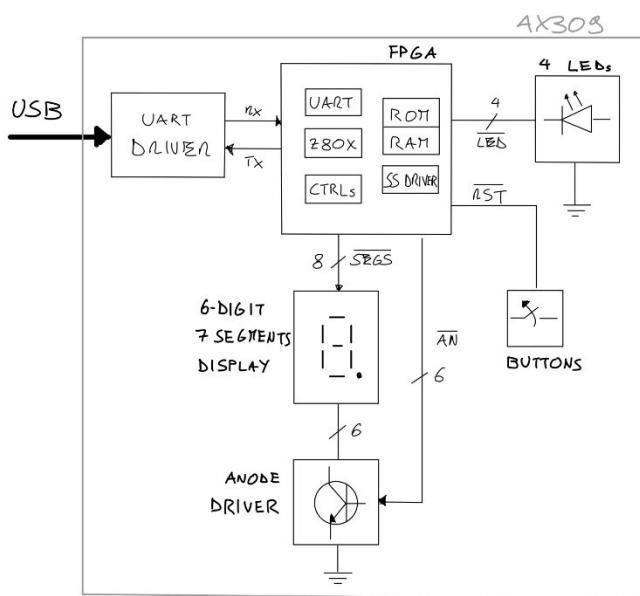


FIGURA 4-1 - SCHEMA DELLE UNITÀ USATE ALL'INTERNO DELLA SCHEDA AX309

Per questo ho creato delle entity che fanno funzionare lo Z80X e ne permettono il controllo. Per prima cosa, lo Z80X è controllabile attraverso la USB presente sulla scheda che viene convertita in protocollo UART e che invia serialmente dei comandi ad un entity dedicata. Lo Z80X comunica con l'esterno attraverso il display 7 segmenti a sei cifre presente sulla scheda. Per controllarlo si usano due bus. Il primo porta i segnali per i singoli segmenti di ogni cifra più il punto decimale, nSEGS, mentre il secondo è il comando degli anodi dei segmenti, nAN. Questo permette di avere una sola cifra accesa alla volta sfruttando la persistenza sulla retina così da risparmiare bus e corrente.

Ci sono anche 4 LEDs collegati che però sono usati per funzioni di debug poiché comunicano lo stato dell'intero sistema. I LEDs segnalano uno stato di errore del controllore, il mancato riconoscimento di un comando via UART, lo stato del clock che viene fornito allo Z80X e infine l'ultimo è collegato all'uscita nHALT dello Z80X.

L'unico ingresso diretto dalla scheda proviene da un bottone di reset che resetta il controllore e il sistema attorno allo Z80X ma non il microprocessore.

RUCS7

Il sistema di sviluppo ho deciso di chiamarlo con l'acronimo RUCS7 che contiene tutte le iniziali dei sistemi presenti: ROM/RAM – UART – Clock - Snapper – 7 segments.

Questi sistemi permettono rispettivamente: il funzionamento dello Z80X, il controllo dell'intero sistema, la generazione di un segnale di clock variabile, il salvataggio dello stato della macchina e il collegamento dello Z80X con il display 7 segmenti.

Lo scopo di avere un sistema controllabile via UART è quello di permettere una programmazione veloce delle memorie assieme ad un facile ed interattivo debugging del sistema. Per cui il sistema permette di poter leggere e scrivere le due memorie indipendentemente dall'utilizzo che ne fa lo Z80X, di controllare il clock dello Z80X anche sino al livello di un'istruzione alla volta e di avere un tracciato dello stato dei pin d'uscita dello Z80X per ricostruire il suo comportamento durante l'esecuzione.

Questi vantaggi sono difficilmente implementabili al difuori dell'FPGA, di conseguenza sono i punti di forza di quest'implementazione.

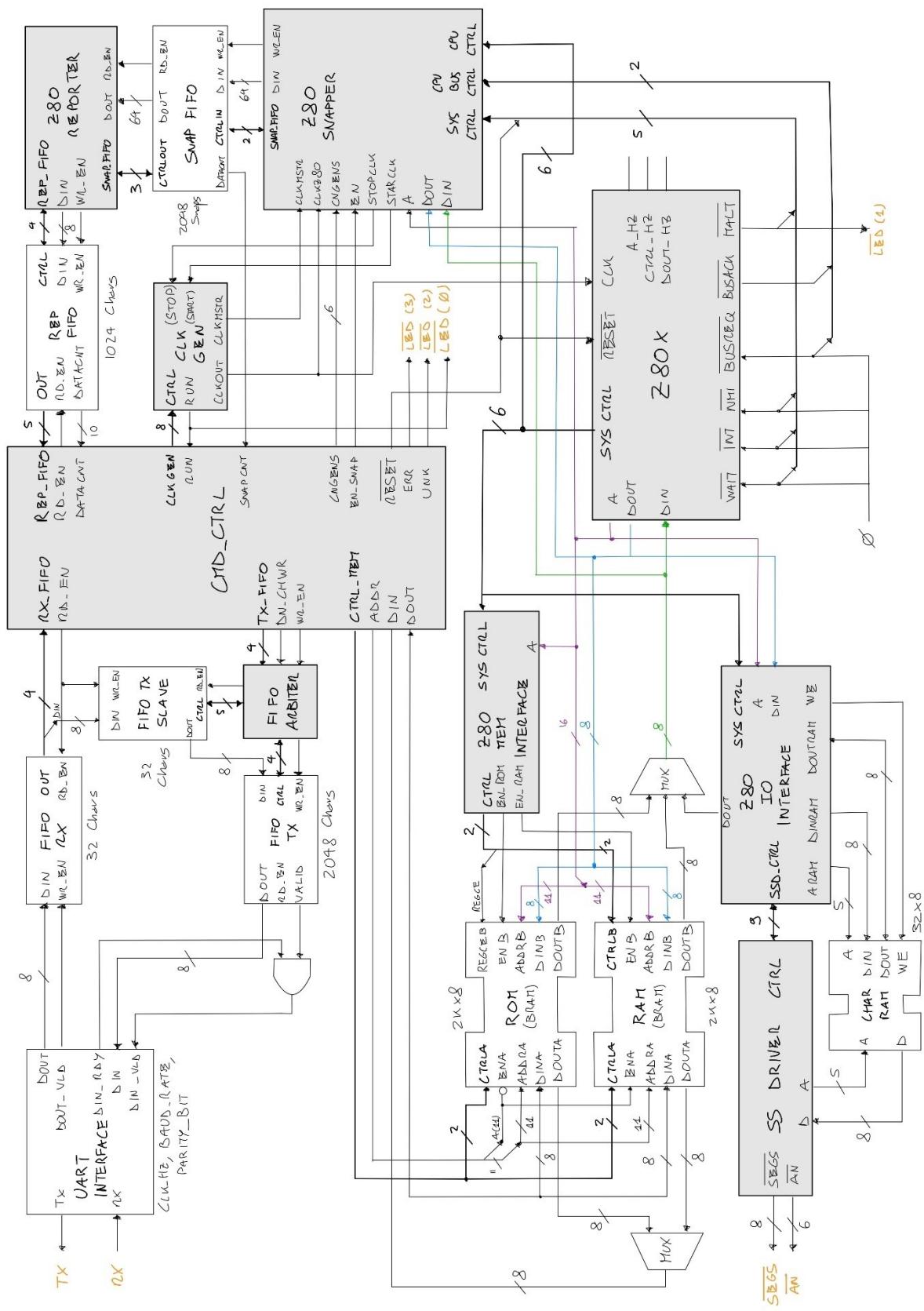


FIGURA 4-2 - SCHEMA DEL SISTEMA DI SVILUPPO RUCS7

1. Bus ed interfacce a registri

All'interno di RUCS7 ci sono molte interfacce per connettere assieme tutte le unità.

Di particolare interesse sono le interfacce tra lo Z80X e le memorie o il driver 7 segmenti.

L'interfaccia verso le memorie è gestita da Z80 MEM INTERFACE. Questo sistema è temporizzato per mezzo del segnale CLK_PEDGE e si occupa di attivare la ROM o la RAM. Principalmente abilita la linea REGCEB, abbrev. di Register Control Enable port B, delle due memorie se è attivo nRD, così da avere il valore aggiornato sul bus dati. Poi abilita WEB, abbrev. di Write Enable port B, della RAM se è attivo nWR, mentre lo stesso pin della ROM è sempre disattivato. L'abilitazione delle due memorie segue invece la seguente logica: se il dodicesimo bit del bus A è 0 viene attivata la ROM altrimenti viene attivata la RAM, sempre se la linea nMREQ è attivata altrimenti sono disattivate entrambe le memorie.

L'interfaccia Z80 IO INTERFACE è in realtà un sistema di più registri collegati al bus e selezionabili.

Ci sono due tipi di registri: quelli di sola lettura da parte di Z80X che sono collegati alle uscite delle periferiche e quelli di lettura e scrittura che sono collegati agli ingressi delle periferiche. Questi registri si abilitano quando nIORQ è attivo e sul bus A è presente l'indirizzo che vi corrisponde.

Il driver 7 segmenti presenta anche una RAM per immagazzinare al massimo 32 caratteri. Questa viene controllata per mezzo dei cinque bit meno significativi dell'indirizzo e la sua uscita viene fornita sul bus solo se l'indirizzo è nell'intervallo 0-31 e il segnale nRD è attivo. Mentre WE viene attivato solo se l'indirizzo corrisponde e nWR è attivo.

2. Memorie

Lo scopo delle memorie in questo design è duplice: separare due regimi con velocità differenti e memorizzare i dati e i programmi per lo Z80X.

Per sfruttare al meglio le potenzialità dell'FPGA, ho usato gli IP messi a disposizione.

Block RAMs, RAM e ROM

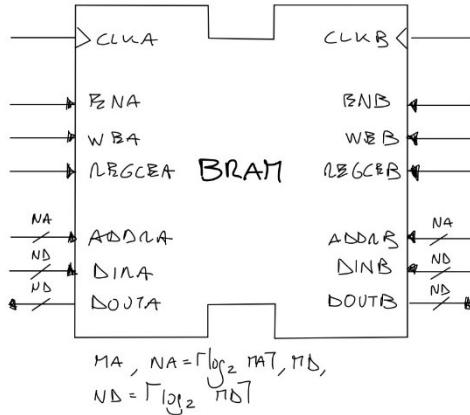


FIGURA 4-3 - SCHEMA DI UNA BRAM
GENERICA

Il design presenta due BRAM identiche ma con due scopi differenti. La prima rappresenta una ROM, per cui da parte dello Z80X non è scrivibile, mentre l'altra è una RAM completamente usabile dal microprocessore. Entrambe permettono l'uso completo da parte di due port, funzionalità real dual port. Il primo port, che ha priorità maggiore, è sempre collegato al controllore così da poter caricare il programma e leggere il valore delle memorie indipendentemente dalle operazioni che sta compiendo lo Z80X.

I port delle BRAM presentano la stessa struttura. Per prima cosa c'è il segnale di temporizzazione che in questo caso è il medesimo. Dopodiché vi sono i pin di controllo: EN, WE e REGCE. Con il primo si abilita la BRAM che altrimenti non risponde ai comandi, con il secondo invece si abilita la scrittura. Il terzo pin abilita i registri d'uscita che servono a mantenere l'uscita della BRAM costante quando non è utilizzata. Permettono anche di non avere un valore fluttuante sul bus dati durante la lettura.

Le BRAM presentano un tempo di accesso pari a 2 cicli di clock e un tempo di latenza di un ciclo [20]. Per permettere la giusta temporizzazione, il controllore fa uso di due entity dedicate, BRAM_READER e BRAM_WRITER, che implementano i cicli di lettura e scrittura. Queste entity permettono di avere un segnale di DONE nel momento in cui il dato è sicuramente stato elaborato correttamente a costo di un ciclo di clock in più.

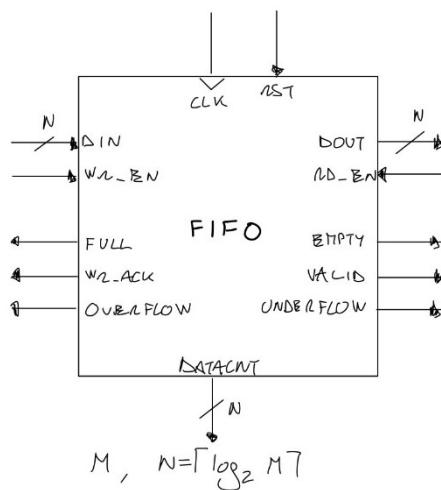
Dopo questi segnali ci sono il bus indirizzi ADDR assieme ai due bus dati DIN e DOUT. La dimensione di questi può essere molto variabile in base al numero di parole contenute e della loro lunghezza. Inoltre si può modificare la lunghezza delle parole in ingresso e uscita permettendo l'uso della memoria come una cache. Per permette un uso più semplice in questa configurazione, WE può diventare un vettore con cui selezionare il byte della parola da scrivere.

Distributed RAM, CHAR_RAM

L'unica RAM distribuita è la CHAR_RAM che serve ad immagazzinare i caratteri che il driver 7 segmenti mostrerà sul display. Ho scelto di implementarla come RAM distribuita poiché è di piccole dimensioni, 256 bit, e se avessi usato una BRAM avrei dovuto sprecare interamente una pagina di 9kb poiché è il taglio minimo.

CHAR_RAM è di tipo simple dual port poiché presenta due port indipendenti ma da uno si può sia leggere che scrivere mentre dall'altro è permessa la sola lettura. Il primo port è connesso all'interfaccia con lo Z80 mentre il secondo al driver.

FIFOs



**FIGURA 4-4 - SCHEMA DI UNA FIFO
GENERICA**

Le FIFOs messe a disposizione come IP presentano due port di controllo e sono implementabili sia con BRAM che con RAM distribuite. In comune sono presenti oltre al segnale di temporizzazione anche un segnale di reset dell'intera FIFO, RST, e un segnale che informa della quantità di dati presenti nella FIFO, DATA_CNT.

Il port di scrittura presenta i due segnali per il dato in ingresso, DIN, e l'abilitazione alla scrittura, WR_EN. Oltre a questi ci sono dei segnali che danno informazioni sullo stato della FIFO:

- FULL, informa che la FIFO è piena e non accetterà più dati;
- OVERFLOW, informa che nell'ultima operazione di scrittura la FIFO era piena ed il dato appena scritto è stato ignorato;
- WR_ACK, è la conferma della scrittura del dato sulla FIFO.

Il port di lettura presenta il bus per il dato in uscita, DOUT, che può avere dimensione diversa da DIN, e il segnale per leggere il dato successivo, RD_EN. Oltre a questi vi sono dei segnali di controllo per gestire il flusso di dati:

- EMPTY, comunica che la FIFO non ha più dati disponibili;
- VALID, comunica che il dato su DOUT può essere letto;
- UNDERFLOW, comunica che nella precedente lettura la FIFO era vuota e che in uscita non sta presentando un dato valido.

La presenza del segnale VALID assieme a EMPTY è particolarmente utile nel caso di FIFO non First-Word Fall-Through (FWFT). Nelle FIFO FWFT, appena un dato è stato scritto è già disponibile sull'uscita e non richiede di scorrere tutta la FIFO prima di averlo disponibile. Nel mio caso ho usato solo FIFO FWFT per la semplicità di utilizzo.

Ho usato le FIFOs principalmente per due scopi: separare i regimi con velocità differenti e per immagazzinare i dati mantenendo il loro ordine.

Al primo scopo ci sono le tre FIFOs per la comunicazione UART: FIFO_RX, FIFO_TX e FIFO_TX_SLAVE.

FIFO_RX serve ad immagazzinare un'intera riga di comando proveniente dall'interfaccia UART mantenendo l'ordine con cui è stata inviata. In questo modo se il controllore sta eseguendo una scrittura sull'UART e nel frattempo arriva un altro comando, la FIFO lo immagazzina. Così il controllore può leggere i caratteri del nuovo comando nel momento in cui l'invio è finito.

FIFO_TX_SLAVE viene scritta man mano che il controllore legge i caratteri da FIFO_RX. Questa FIFO serve ad immagazzinare il comando che il controllore sta eseguendo per poi renderlo disponibile sull'UART quando la scrittura del report è terminata. È utile poiché può succedere che la risposta del controllore sia abbastanza lunga da costringere lo stesso controllore, al comando successivo, ad attendere che la precedente scrittura termini prima di iniziare il nuovo report.

FIFO_TX è la memoria che contiene i caratteri che il controllore vuole inviare sull'UART. Il controllore ha necessità di questa FIFO per essere libero di svolgere altre funzioni mentre è in

corso la scrittura su UART, che dura decisamente di più di un singolo ciclo del controllore. Il collegamento delle due FIFO_TX e FIFO_TX_SLAVE è fatto per mezzo dell'entity FIFO ARBITER che ha il compito di direzionare i dati dal controllore o dalla FIFO slave sulla FIFO master. Principalmente l'arbiter da priorità alla scrittura da parte del controllore, cioè se DN_CHWR è disattivato. Altrimenti se DN_CHWR è attivo scarica la FIFO slave nell'altra. Nel caso in cui durante questo scarico, DN_CHWR si attiva, l'arbiter finisce l'operazione in corso non trasmettendo il segnale WR_ACK, che il controllore si attende dalla FIFO.

Le FIFOs SNAP_FIFO e REP_FIFO servono alla porzione di sistema che si occupa di fare le istantanee dello stato dello Z80X.

SNAP_FIFO contiene le istantanee, chiamate anche snaps, da parte di Z80_SNAPPER. Quest'ultima entity tramuta lo snap in una parola di 64 bit che viene immagazzinata nella FIFO. La FIFO è di dimensioni elevate, 2048 parole, perché lo snapper funziona a burst cioè se attivato fa andare lo Z80X sino al riempimento della memoria e poi blocca il clock verso il microprocessore fino a che la FIFO non viene svuotata.

REP_FIFO invece contiene i caratteri derivanti dalla conversione della parola dello snap memorizzato nella SNAP_FIFO. Per cui tramuta in una serie di caratteri direttamente stampabili sulla UART la parola immagazzinata e la rende disponibile sulla REP_FIFO. Così il controllore può semplicemente leggere un dato dalla REP_FIFO e metterlo sulla FIFO_TX in caso di lettura dagli snaps. Anche in questo caso la dimensione della FIFO è considerevole poiché ogni snap occupa 26 caratteri e per mantenere un buon flusso di dati serve che vi siano molti snap convertiti, già pronti per essere scritti.

Per gestire correttamente la lettura e la scrittura sulle FIFOs ho usato due entity dedicate.

CHAR_FEEDER_FIFO

Per leggere i dati dalla FIFO ho usato un'entity che si collega ai soli pin di controllo del port d'uscita della FIFO. Si occupa di segnalare all'entity a valle quando il dato è pronto e di avanzare con le parole quando necessario.

Per far ciò il port per l'entity a valle presenta tre segnali: RDY, GOT e VLD.

RDY, abbrev. di Ready, è attivato dall'entity che legge quando è pronta per il valore successivo. Viene disattivato durante l'elaborazione del dato.

Quando il dato presentato è stato letto ma non ancora processato, viene attivato il segnale GOT.

CHAR_FEEDER_FIFO attiva la linea VLD per segnalare che il dato presente sull'uscita della FIFO è valido alla lettura.

Per cui CHAR_FEEDER_FIFO prima attende che la FIFO non sia più vuota, WTEMPTY, e poi che il dato presente in uscita sia valido, WTVLD. Dopodiché attiva il segnale VLD in uscita, VON. A questo punto disattiva VLD se la entity a valle a recepito, VOFF, e quando questa è pronta, legge il valore successivo, RDEN. A questo punto, se si verifica un underflow, la macchina torna all'attesa che la FIFO non sia vuota, altrimenti attende solo che il dato sia valido.

Lo scopo di questa macchina, oltre a svolgere il semplice ciclo di lettura della FIFO, è anche quello di permette la lettura della stessa FIFO da parte di più entity in contemporanea e di gestire cicli di lettura di durata non costante. Specialmente nel controllore, dove sono presenti più riconoscitori di sequenze, quest'entity temporizza le fasi di lettura sulla durata dell'esecuzione più lunga, in corso tra i riconoscitori di sequenze.

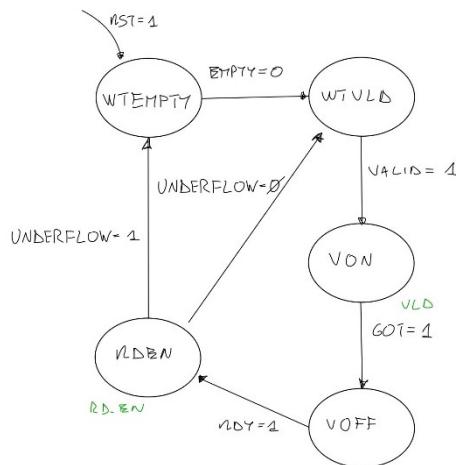


FIGURA 4-5 - DIAGRAMMA DI STATO DI CHAR_FEEDER_FIFO

CHAR_WRITER_FIFO

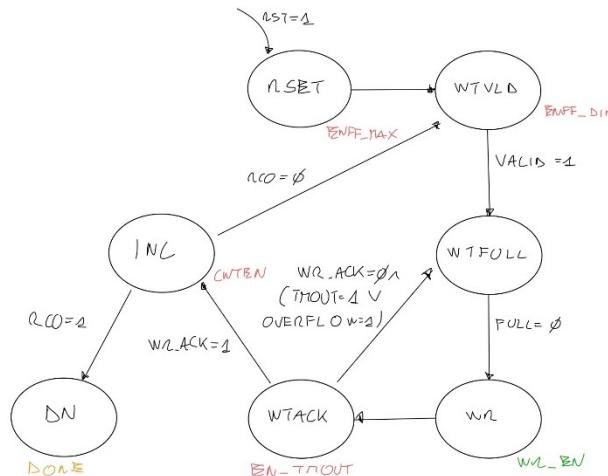
Per scrivere su una FIFO, l'entity usa l'intero bus di controllo e fornisce da lei il dato in uscita. Però presume di leggere i dati da una ROM.

Per cui ad ogni scrittura di un blocco di dati, l'entity dev'essere resettata e in quel caso campiona il valore presente all'ingresso MAX_ADDR che segnala quale sia la dimensione del blocco di dati. Il valore viene campionato per essere sicuri che non vari nel corso dell'elaborazione.

Dopodiché entra nel ciclo di trasferimento. Per prima cosa attende che il valore fornito dalla ROM sia valido attivando il segnale VALID e campiona il valore presente su DIN. Poi

attende che la FIFO su cui sta scrivendo abbia spazio libero. In quel caso fornisce sull'uscita il valore di DIN campionato ed emette un impulso su WR_EN per poi attendere che la scrittura sia andata a buon fine. Altrimenti ripete le fasi di attesa e scrittura.

La mancata attivazione di WR_ACK è considerata un fallimento della scrittura se scatta un timer di attesa o si attiva il segnale OVERFLOW. Se la scrittura va a buon fine si incrementa l'indirizzo con cui si legge dalla ROM. L'indirizzo è fornito da un contatore variabile che



**FIGURA 4-6 - DIAGRAMMA DI STATO DI
CHAR_WRITER_FIFO**

conta sino al valore campionario di MAX_ADDR e quando lo raggiunge attiva la linea RCO, abbrev. di Ripple-Carry Out. L'entity interpreta questo segnale come la fine del processo e l'ingresso nello stato DN che comunica all'esterno col segnale DONE.

3. Display a 7 segmenti

Lo scopo dell'uso del display a 7 segmenti è quello di far comunicare lo Z80X con l'utente per cui deve permettere un utilizzo molto flessibile. A questo scopo è stato sviluppato il driver per il display.

Il driver chiamato SS_DRIVER presenta una struttura a livelli in base alle azioni da compiere.

Nel livello più basso si trova un decoder da ASCII ai segnali di controllo dei segmenti.

Questo decoder chiamato SSegsASCII riceve in ingresso il codice ASCII a 7 bit del carattere da visualizzare e ritorna il segnale per il 7 segmenti a 8 bit in cui il più significativo rappresenta il punto decimale. La presenza di questo driver permette di non preoccuparsi su come mostrare una cifra, una lettera o un simbolo sul display ma basta solamente fornire il codice ASCII.

Al livello superiore vi è l'entity ASCIISSegsDriver che permette il controllo di tutte le cifre. Presenta due generic: DIGIT, che è il numero di cifre del display così da rendere universale il controllore; CLK_HZ, che è la frequenza in hertz del clock per generare correttamente i segnali di temporizzazione. L'entity si occupa di gestire le cifre per mezzo di una logica multiplexata con gli anodi.

Per cui genera un vettore con un bit per ogni cifra, in cui è presente al massimo un solo segnale di attivazione con cui attiva a turno ogni cifra. In questo modo mantenendo un tempo di circa 1 ms per ogni cifra si sfrutta la persistenza dell'occhio così far sembrare accese tutte le cifre contemporaneamente. Ma allo stesso tempo risparmiando sia bus per i segmenti che corrente. Inoltre l'entity fornisce anche il valore decodificato della cifra corrispondente attraverso SSegsASCII. Utilizzando poi il bit più significativo della cifra in ingresso, che rimarrebbe inutilizzato, come bit per l'attivazione del punto decimale permette di non sprecare una cifra come punto. I caratteri gli vengono passati ordinati per mezzo di un unico vettore di tanti byte quante sono le cifre.

Inoltre l'entity mette a disposizione un vettore per abilitare o meno le cifre così da spegnerne selettivamente una o più.

Sopra quest'entity vi è l'entity CircBuffSSegs che implementa un buffer circolare per il display. Basandosi su una RAM che immagazzina i dati, li legge uno per volta e li inserisce in uno shift register SIPO grande tanti byte quante sono le cifre. Così in caso di messaggi lunghi, i caratteri si possono fare scorrere sul display.

La RAM è indirizzata da 0 sino al valore dell'ingresso LIM che segna l'indirizzo dell'ultimo

carattere da visualizzare.

L'entity fornisce in uscita il segnale LAST che segnala al livello superiore quando si sta leggendo l'ultimo carattere e cioè quando l'indirizzo puntato è pari a LIM.

Lo shift dei caratteri è gestito dall'ingresso SH e quindi dall'entity del livello superiore.

L'ultimo livello è occupato da SS_DRIVER che gestisce quattro diversi comportamenti del display: la dimensione del messaggio e lo scorrimento, la velocità di scorrimento, il lampeggio selettivo delle cifre e la funzione di allineamento, detta adjust.

L'entity si aspetta che il messaggio sia caricato autonomamente sulla RAM dall'utilizzatore e che questo comunichi la dimensione del messaggio inserendo il numero di caratteri sul vettore NCHAR. L'entity abiliterà lo scorrimento dei caratteri se questi superano il numero di cifre del display. La frequenza dello scorrimento è selezionata dall'esterno per mezzo del vettore SHDIV come divisione di una frequenza fondamentale e massima. La frequenza massima è settata per mezzo del generic SH_MAX_FREQ_HZ assieme al numero di bit del divisore N_SH. L'utilizzatore può anche forzare lo scorrimento per mezzo dell'ingresso FSH, abbrev. di Force Shift. Il segnale di fine messaggio dell'entity sottostante è riportato all'esterno.

L'entity gestisce anche il lampeggio delle cifre selezionate dal vettore ENSTROBE tutte con la stessa frequenza impostata per mezzo di STROBEDIV come divisione di una frequenza massima. La frequenza e il numero di bit del divisore sono settati con i generic STROBE_MAX_FREQ_HZ e N_STROBE rispettivamente.

In aggiunta l'entity permette di aggiustare a sinistra l'inizio del messaggio sul display e di spegnere l'intero display con il segnale ENOUT. Queste operazioni sono utili quando si scrive un nuovo messaggio sullo schermo. Per cui prima si spegne la visualizzazione, si scrive il nuovo messaggio in RAM, si precarica il buffer circolare con i primi caratteri e poi si riattiva il display.

Per attuare l'adjust, l'entity fa uso di una piccola FSM. La macchina rimane nello stato IDLE fintantoché l'ingresso ADJ non è attivato. Quando ciò si verifica, entra nello stato ADJR in cui resetta il buffer circolare ed un contatore di modulo pari al numero di cifre. Poi nello stato ADJS abilita il contatore, che mantiene attivo il segnale di scorrimento del buffer circolare sino al completamento del conteggio. A questo punto la macchina passa allo stato ADJW in cui attende che il segnale ADJ si disattivi e nel frattempo attiva il segnale ADJDN, abbrev. di Adjust Done, disattivando lo scorrimento del buffer. Dopodiché torna in IDLE.

4. Controllore via UART

Al centro del sistema di sviluppo vi è la sezione controllata via UART e che fa capo al controllore. Questo sistema permette di controllare le funzionalità della scheda inviando comandi via UART e permette di non dover ricaricare il design sull'FPGA per modificare le memorie o la frequenza del clock.

Tutte le unità presenti le ho progettate e implementate perché non avessero necessità di interfacce ulteriori e con lo scopo di massimizzare la velocità. La possibilità del debug e del controllo sono garantite se le operazioni che compiono il controllore e le altre macchine rientrino all'interno di mezzo ciclo di CLK così che lo Z80X non riesca a notare le modifiche. L'unità che fa da interfaccia per il protocollo UART invece l'ho usata come un IP da “Simple UART for FPGA” di Jakub Cabal [32].

L'interfaccia UART non permette molti controlli sulla gestione della comunicazione ma è ideale per quello che deve fare all'interno di questo sistema. Permette di selezionare la velocità di trasmissione, detta baudrate, con il generic Baud_Rate, e la presenza e la tipologia del bit di parità, detto parity bit. Per questo design ho impostato la prima al massimo valore accettato dal driver USB cioè 115200 Bd e nessun parity bit per cui la trasmissione di un carattere dura circa 87 µs.

Il port dell'interfaccia mostra due sezioni, uno per l'invio e l'altra per la ricezione.

Per l'invio, è presente il bus per i dati, DIN, assieme ai segnali DIN_VLD e DIN_RDY. Con il primo si segnala all'interfaccia che il valore presente su DIN è stabile e può essere inviato, mentre con DIN_RDY l'interfaccia comunica di essere pronta per il prossimo invio.

Per l'uscita, sono presenti alcuni segnali che indicano errori nei valori ricevuti, quelli che interessano però sono i segnali DOUT e DOUT_VLD. DOUT è il bus che contiene il valore ricevuto e solamente quando avviene un impulso su DOUT_VLD si può considerare corretto il suo valore. Per cui si può usare DOUT_VLD come impulso di scrittura sulla FIFO_RX.

A. Controllore e interfaccia verso il PC

Il controllore, chiamato CMD_CTRL, è il centro di controllo dell'intera scheda di sviluppo. Si occupa di leggere il comando dalla FIFO_RX, interpretarlo, eseguirlo e inviare sull'UART una risposta, detta report.

La macchina presenta una forma a FSMs annidate per cui la FSM del controllore gestisce più macchine che in vari stadi lavorano in parallelo per riconoscere ed eseguire i comandi.

Il funzionamento del controllore può essere riassunto in tre fasi principali: fetch/decode – execute – report.

Le sottomacchine possono divise in due gruppi FD, abbrev. di Fetch/decode, e EX, abbrev. di Execute. Ogni macchina del primo gruppo è accoppiata con una del secondo gruppo e se una delle macchine del gruppo FD va a buon fine, viene attiva la sola macchina corrispondente nel gruppo EX.

Nella prima fase, il controllore si occupa solo di fornire i caratteri al gruppo FD. Queste macchine si occupano di riconoscere il comando e di estrapolare i campi con i dati. In un normale funzionamento, ogni macchina legge il comando ed una sola lo riconosce attivando il proprio segnale di MATCH.

Il controllore quando riconosce un terminatore di linea, cioè i caratteri LF e RT, che fanno attivare il segnale EOL, entra in uno stato di attesa. Da qui vi può uscire se arriva un carattere non EOL per cui è iniziato un altro comando o quando scatta un time-out che dura quanto la trasmissione di due caratteri.

Il controllore poi analizza la situazione:

- se una sola macchina ha dato un MATCH, entra nella fase di execute ed attiva la sola macchina corrispondente;
- se nessuna macchina ha dato un MATCH significa che il comando è sconosciuto e il controllore entra nel suo ciclo di gestione;
- se più di una macchina da un MATCH o almeno una macchina segnala un errore, si entra nel ciclo di errore.

Nel caso di esecuzione corretta, la macchina verrà avvertita con un segnale DONE_EX della fine della fase. Se non si verificano errori, per cui si entra nel ciclo di gestione, si va nella fase di report. Qui per prima cosa viene resettata l'entity che scrive sulla FIFO_TX, cioè la macchina CHAR_WRITER_FIFO. Poi si seleziona su un multiplexer l'unico messaggio da inviare, di conseguenza tutti i tipi possibili di messaggio di responso vengono sempre generati ma solo quello corretto viene instradato per essere scritto. Inoltre si dà all'entity che scrive anche il numero di caratteri proprio per ogni messaggio. Infine si attiva CHAR_WRITER_FIFO, che quando ha terminato attiva il segnale DONE_CHWR e il controllore torna alla fase di fetch/decode. Nel caso di errore o comando sconosciuto vengono

generati due segnali, ERR e UNK rispettivamente, e viene selezionato il messaggio corrispondente.

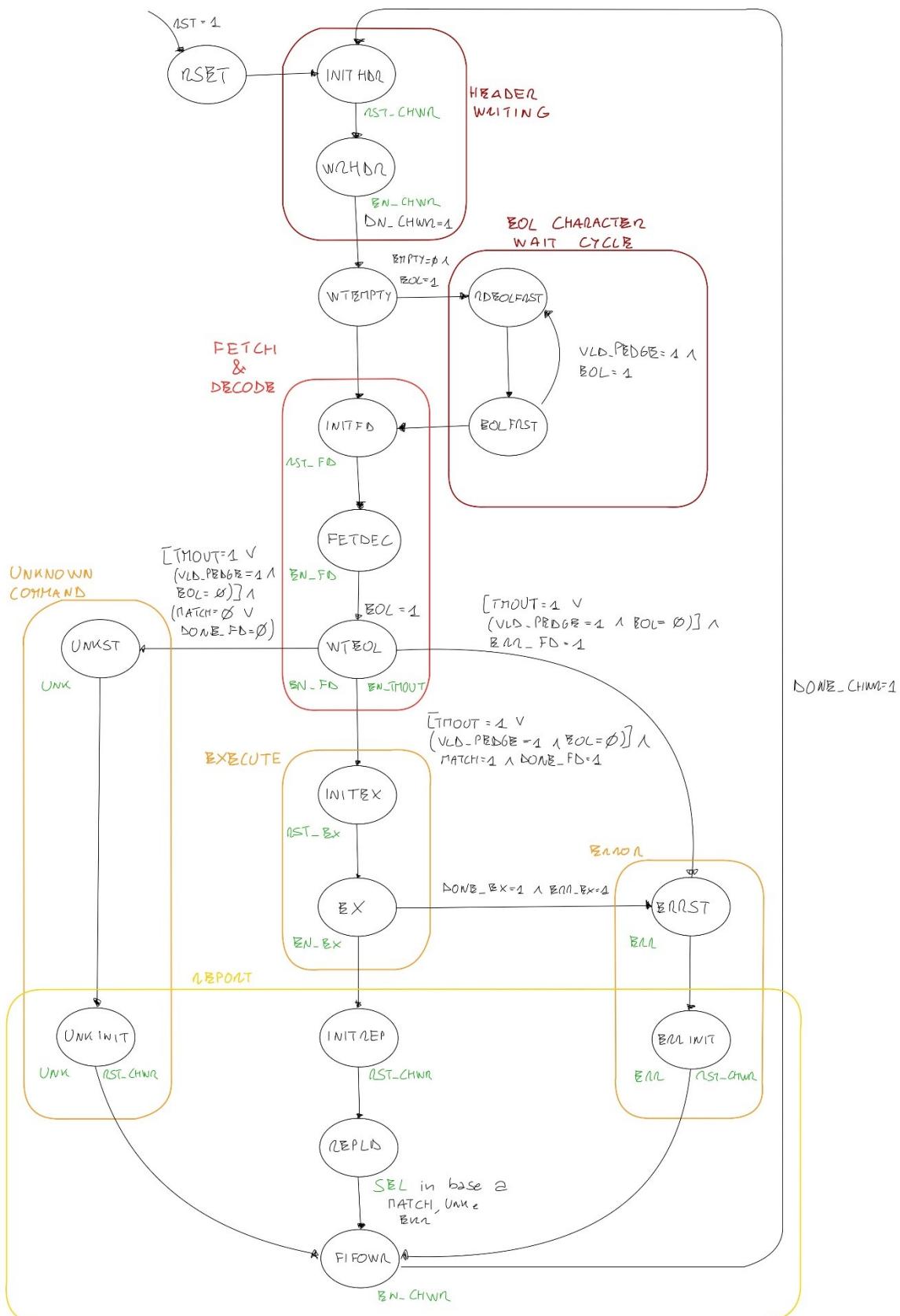


FIGURA 4-7 - DIAGRAMMA DI STATO DELLA FSM DI CMD CTRL

Le macchine del gruppo FD sono tutte dei riconoscitori di sequenze e presentano una struttura simile. Sono raggruppate in modo che riconoscano comandi che hanno funzioni simili tra loro. Tutte hanno in comune i segnali MATCH, DONE ed ERR_FD. Il primo si attiva se viene riconosciuto il comando corrispondente. Il secondo si attiva se è avvenuto un match o il messaggio è sconosciuto e di conseguenza comunica che questa macchina non ha più necessità di altri caratteri. Infine il terzo avverte di un errore.

Inoltre queste macchine hanno in comune i controlli per l'entity di lettura CHAR_FEEDER_FIFO, RDY e GOT. A livello del controllore questi segnali vengono combinati così che tutte le entity possano leggere i caratteri correttamente.

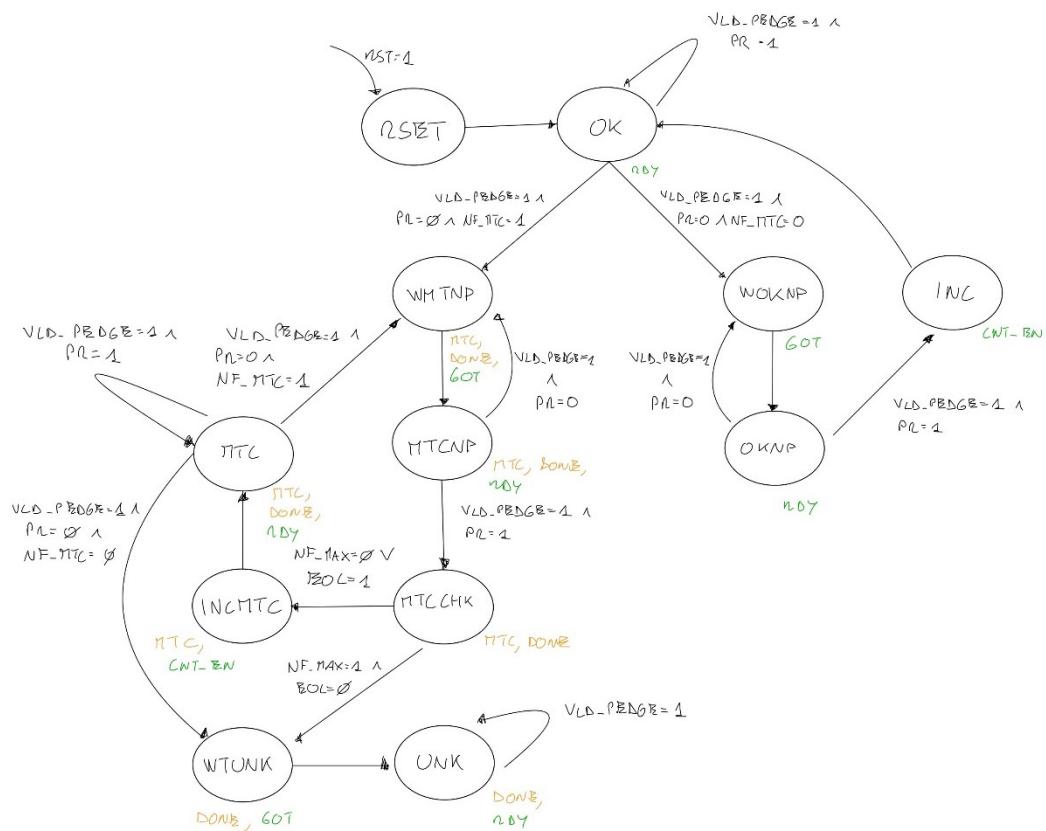


FIGURA 4-8 - DIAGRAMMA DI STATO DI FIELD_RECOGNIZER

Al centro di queste macchine vi è l'entity FIELD_RECOGNIZER. Quest'entity riconosce il numero di campi e scandisce l'ordine delle altre FSM collegate. Anche in questo caso vi è una struttura a FSMs annidate in cui ci sono tanti gruppi di macchine quanti il numero massimo di campi dei comandi riconoscibili.

Ogni istruzione va da un minimo di un campo a tre campi. La macchina li riconosce perché sono separati da caratteri non stampabili, come lo spazio o la tabulazione che sono individuati dalla disattivazione del segnale PR, abbrev. di Printable.

La macchina inizialmente viene resettata e poi entra nel ciclo di lettura dei caratteri, OK. Appena arriva un carattere non stampabile entra in un ciclo di attesa, WOKNP e OKNP, che termina appena arriva un carattere stampabile. Poi incrementa un contatore che fornisce un segnale one-hot di attivazione ai gruppi di macchine, INC. Il contatore fornisce anche in uscita il numero dei campi letti. Un solo gruppo di macchine è attivo alla volta e quando sono disattivate rimangono immobili senza modificare lo stato delle uscite.

Se all'arrivo del carattere non stampabile, l'indice del campo del comando rientra in un intervallo prefissato vi è il match del numero di campi e si entra nel ciclo che gestisce questo caso. Il ciclo funziona allo stesso modo del precedente solo con l'aggiunta di uno stato di controllo in più, MTCCHK, che controlla se al prossimo incremento si supera l'intervallo di campi consentiti. Di conseguenza, il comando è sconosciuto e si entra nel proprio ciclo di gestione e la macchina vi rimane intrappolata fino al prossimo riavvio.

I gruppi di macchine controllate da FIELD_RECOGNIZER sono formati da più istanze, con piccole variazioni, delle tre stesse macchine generiche. La prima riconosce un messaggio fisso salvato in una ROM e si chiama STATIC_RECOGNIZER. La seconda riconosce un campo di lunghezza variabile che può contenere valori diversi, come ad esempio l'indirizzo a cui leggere, e si chiama DYNAMIC_RECOGNIZER. Quest'ultima salva i caratteri del campo in uno shift register. La terza, che si chiama ONECHAR_RECOGNIZER, è una variazione della seconda e differisce solo perché il campo ha lunghezza unitaria e controlla un registro invece che uno shift register.

STATIC_RECOGNIZER

STATIC_RECOGNIZER è un classico riconoscitore di sequenze reso universale dal generic LEN assieme al collegamento ad una ROM dedicata che contiene i caratteri del messaggio da riconoscere.

Per prima cosa la macchina dev'essere riavviata e a questo punto, stato OK, ad ogni nuovo carattere lo confronta con quello presente in memoria e nel caso siano uguali attiva il segnale COR. Se il carattere è sconosciuto va in un ciclo di attesa sino al prossimo riavvio, RDUNK e UNK che sono separati per generare i segnali RDY e GOT. Altrimenti la macchina va in uno stato per generare l'impulso di GOT ed incrementare il contatore che contiene l'indirizzo della ROM. Se si arriva sino alla fine del messaggio, si passa agli stati WTMTC, per generare il segnale GOT, e poi a MTC che attiva il segnale MATCH. Se arriva un altro carattere, il messaggio è sconosciuto.

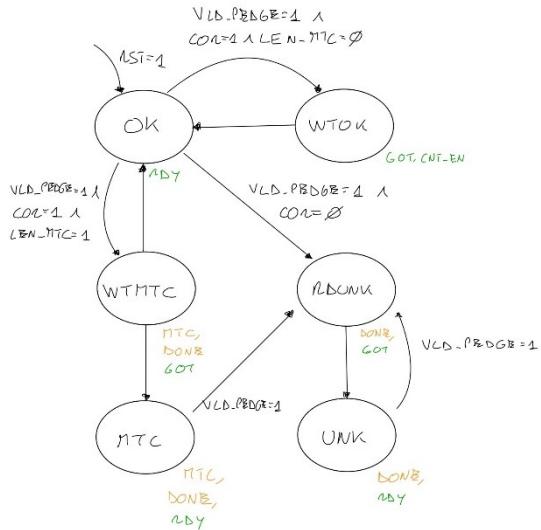


FIGURA 4-9 - DIAGRAMMA DI STATO DELL'ENTITY STATIC_RECOGNIZER

DYNAMIC_RECOGNIZER

A differenza di STATIC_RECOGNIZER, quest'entity deve riconoscere e salvare un campo di lunghezza e contenuto variabili. Inoltre, per rendere la macchina riutilizzabile, non genera da sola il segnale COR ma gli viene fornito dall'esterno. Per esempio, molti campi possono di numeri esadecimali, per cui la logica che genera COR controlla se il carattere in ingresso è una cifra esadecimale attivando il segnale o meno.

La macchina per funzionare dev'essere prima resettata, poi entra nel ciclo per i caratteri corretti, OK. Se arriva un carattere e COR è attivato, lo carica nello shift register attivando il segnale SHEN, abbrev. di Shift enable. Se COR è disattivato invece va nel ciclo per i comandi sconosciuti. Nel caso in cui la lunghezza del campo sia in un range definito con i generics, entra nel ciclo di match che attiva il segnale MATCH.

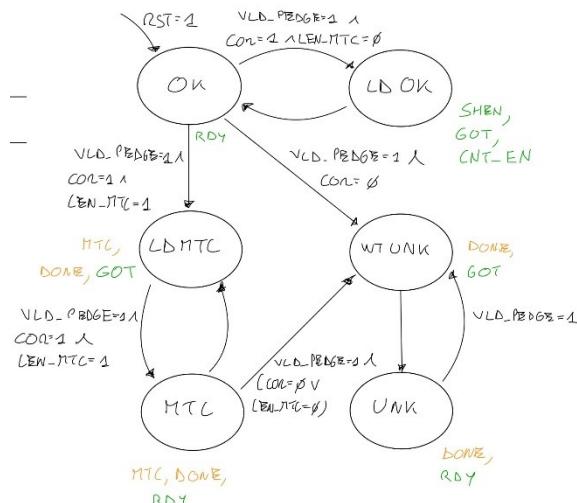


FIGURA 4-10 - DIAGRAMMA DI STATO DELL'ENTITY DYNAMIC_RECOGNIZER

Comandi

Il controllore riconosce 5 diversi gruppi di comandi: Write, Read, Clock, Report e Reset.

Il controllore per comunicare che è pronto a ricevere un comando, invia sull'UART

l'intestazione “>”, che è per l'appunto la prima operazione svolta all'inizio del ciclo della FSM. Dopodiché esegue un eco di battitura del comando che riesce a leggere.

Durante la fase di report, mette come intestazione un carattere speciale e fa seguire a tutto il messaggio inviato un'intera riga vuota. La presenza del carattere speciale servirebbe per una futura automatizzazione delle operazioni con il controllore poiché possono essere usati come tag per riconoscere l'operazione svolta.

Per tutti i caratteri, il controllore è case insensitive.

Gruppo Write

Il gruppo Write si occupa di scrivere i dati passati sulle memorie collegate allo Z80X e fa uso del carattere speciale “@” che significa asserzione al comando inviato.

Sono disponibili due modalità, una scrive una sola riga, l'altra permette di usare un comando più corto, quindi più veloce, e scrive il dato di seguito al precedente. L'ultima modalità è utile per scrivere molte righe di seguito e fa uso di un contatore per selezionare l'indirizzo.

Per scrivere una singola riga si fa uso del comando:

WR <dd> <aaaa>

<dd> : è il dato scritto come esadecimale di 1 o 2 caratteri.

<aaaa> : è l'indirizzo scritto come esadecimale, al massimo di 4 caratteri.

La macchina carica il valore aaaa nel contatore e scrive il dato dd all'indirizzo puntato.

Per la scrittura multipla, vi sono due istruzioni: la prima setta il contatore mentre la seconda esegue la scrittura all'indirizzo puntato dallo stesso.

WRL <aaaa>

Imposta il valore del contatore.

W <dd>

Scrive il valore dd all'indirizzo puntato da dal contatore.

In ogni caso dopo una qualsiasi operazione di scrittura, il contatore viene incrementato e la macchina risponde con “@”.

Gruppo Read

Il gruppo si occupa di stampare sull'UART il valore delle righe selezionate della memoria.

Prima dell'opportuna risposta, inserisce il carattere speciale “#” che identifica il gruppo.

Ci sono tre comandi:

RD <aaaa> Legge la sola riga puntata da aaaa.

RDL <aaaa₁> <aaaa₂> Legge l'intervallo di righe tra aaaa₁ e aaaa₂.

RDM ALL | ROM | ROM Legge l'intervallo della sezione selezionata.

Al primo comando la macchina legge la riga corrispondente e scrive sull'UART il dato come numero a due cifre esadecimale.

Negli altri due comandi invece scrive una o più righe secondo lo schema

<line_of_data>:

<line_of_data> : <dd> {<dd> ...}

Le <line_of_data> sono linee che contengono un numero prefissato di dati settato con il generic DATAxLINE. Per cui fintato che ci sono dati da inviare e il numero su quella linea è inferiore a DATAxLINE, vengono scritti i dati come coppie di cifre esadecimali seguite da due spazi. Se il dato è l'ultimo della linea, viene seguito dalla coppia di caratteri terminatori di linea.

Nel caso di RDM, con i tag vengono selezionati degli intervalli prefissati. Questi sono generati dal numero di bit dell'indirizzo della memoria che è passato all'entity come il generic N. Per cui si ha:

ALL :	aaaa ₁ = 0,	aaaa ₂ = 2 ^N - 1
ROM :	aaaa ₁ = 0,	aaaa ₂ = 2 ^{N-1} - 1
RAM :	aaa ₁ = 2 ^{N-1} ,	aaa ₂ = 2 ^N - 1

La macchina che ne esegue la decodifica dell'operazione può generare errore se aaaa_2 è minore di aaa₁ o almeno uno dei due esce dalle dimensioni della memoria, cioè fuori dall'intervallo [0, 2^{N-1}].

Gruppo Clock

Sono le istruzioni che controllano l'entity CLK_GEN e quindi il clock dello Z80X. Ne fanno parte quattro istruzioni.

CLK START STOP	Attiva o disattiva l'uscita del clock verso lo Z80X
CLKDIV <n>	Setta il divisore di frequenza con cui si ottiene il clock d'uscita
<n> :	è un valore esadecimale da 0 a $2^{\text{DIV_N}} - 1$. DIV_N è un generic. Il numero massimo di cifre si ottiene con la parte alta della divisione di DIV_N per 4.
CLKST <s>	Setta il numero di cicli di clock d'uscita per cui far andare il segnale prima di fermarsi.
<s> :	è un valore esadecimale da 0 a $2^{\text{STEP_N}} - 1$. STEP_N è un generic. Il numero massimo di cifre si ottiene con la parte alta della divisione di STEP_N per 4.
M1 <s>	Che setta il numero di fronti negativi di nM1 per cui far andare il segnale di clock prima di fermarsi.

A tutti i comandi viene risposto con il carattere di asserzione “@”.

La macchina di fetch/decode può generare errore se il valore di n o s è oltre il valore massimo. Questo perché con il numero di cifre a disposizione si possono rappresentare numeri più grandi di quelli consentiti.

Gruppo Reporter

Il gruppo reporter si occupa di gestire il report dello stato dello Z80X. Vi fanno parte tre istruzioni:

REPEN	Abilita la sezione di snap dello Z80X
REPSEL <hh>	Abilita o meno i trigger sul cambiamento dei campi dello snap
<hh> :	è una coppia di cifre esadecimali che rappresentano un numero binario a 7 cifre. Ogni cifra corrisponde al segnale di abilitazione di un campo specifico.

REP Inizia l'invio di tutti gli snap acquisiti sino all'invio di questo comando.

Ai primi due comandi, la macchina risponde con il carattere di asserzione “@”.

Al comando REP invece segue l'invio di tutte le righe di snap presenti. Le righe di snap sono formate nel seguente modo:

\$ <CNT_MSTR> <CNT_CLK> <CTRL_BUS> <A> <DIN> <DOUT>

Il carattere “\$” segnala l’inizio di una riga di snap. Segue un campo di 4 cifre esadecimali che rappresentano il conteggio su 16 bit dei cicli di clock master, a 4MHz, contati dall’inizio.

Questo permette di avere un riferimento temporale dei singoli snap. Il campo seguente è un altro riferimento temporale rispetto al clock dello Z80X, occupa una sola cifra ed è su 3 bit.

Segue una parola di 4 caratteri che rappresenta la giustapposizione di tutti i segnali di controllo dello Z80X , sia in ingresso che in uscita, rappresentati su 13 bit.

Come ultimi campi vi sono i valori dei bus A, DIN e DOUT rispettivamente su 4 e 2 cifre.

Ogni campo è separato da uno spazio e la riga termina con i due caratteri terminator.

Con l’istruzione REPSEL si selezionano quali campi attivino il trigger di Z80_SNAPPER per creare un altro snap. I gruppi di segnali che variando possono generare un segnale di trigger sono: nRESET, System Control, CPU Control meno nRESET, CPU Bus Control, A, DIN e DOUT. La macchina di fetch/decode genera errore se il valore di hh non è rappresentabile su 7 bit.

Gruppo Reset

In questo gruppo vi è l’unica istruzione RESET che attiva il ciclo di reset. Il ciclo inizia impostando gli step del segnale CLK di CLK_GEN su 5 cicli, così da essere sicuri che lo Z80X si resetti, e si mantiene attivo il segnale nRESET fintantoché il clock sta andando.

Errore e Comando sconosciuto

In caso avvenga un errore o il comando sia sconosciuto il controllore risponde sull’UART con due comandi dedicati.

Per l’errore risponde con:

&Err <ERR_CODE> Che con il carattere speciale “&” segnala un errore e con ERR_CODE informa chi l’ha generato.

In caso di comando sconosciuto il controllore risponde con:

!Unk cmd

In cui il carattere “!” segnala lo stato.

B. Generatore di clock variabile

Il controllore delega la gestione del clock dello Z80X all'entity CLK_GEN. Questa viene controllata con una serie di segnali. L'entity si comporta come un sistema ad orologeria che cambia funzione in base agli stimoli dall'esterno.

Di base c'è un divisore che genera il segnale CLK_MSTR che è un clock a 4MHz costanti. Poi per mezzo di un altro divisore variabile genera il segnale CLKOUT che può variare dai 4MHz di CLK_MSTR a 50Hz.

Se il segnale START viene attivato il divisore variabile viene abilitato altrimenti rimane fermo all'ultimo stato di CLKOUT. Con STOP invece si ferma il divisore se prima è stato attivato in un qualsiasi modo. Il valore di divisione è salvato in un registro che viene caricato al valore DIV se DIV_LD è attivo.

Nel caso del funzionamento a step, con STEPEN si abilita il divisore che viene bloccato appena il conteggio dei cicli di clock è arrivato al valore salvato in un opportuno registro. Lo stesso contatore conta, per mezzo di un multiplexer, i fronti negativi di nM1. Per cui il comportamento con M1EN è lo stesso di STEPEN ma riguardo a nM1. Il numero di passi viene caricato in un opportuno registro quando non sono in corso le due fasi e il valore viene campionato su STEP.

I segnali RUN e ERR informano il controllore. RUN è attivo quando il divisore che genera CLKOUT è abilitato. Mentre ERR si attiva quando l'unità riscontra un qualsiasi problema. Uno di questi è quando più di un segnale di abilitazione viene attivato contemporaneamente. In caso di errore l'unità si blocca e serve resettarla.

C. Z80 snapper e Z80 reporter

Come per la generazione del clock, il controllore delega la generazione, la memorizzazione e la traduzione degli snap a due entity Z80_SNAPPER e Z80_REPORTER.

L'entity Z80_SNAPPER è basata su una FSM.

La macchina rimane in uno stato di attesa, WT, fintantoché il controllore non attiva il segnale di enable, EN, e la SNAP_FIFO permette la scrittura, quindi FULL è disattivato.

In questo caso lancia un impulso di attivazione al CLK_GEN ed entra in uno stato di attesa, IDLE. Esce da questo stato quando il segnale interno CNG si attiva. Questo segnale è generato dalla variazione di un vettore particolare creato dalla giustapposizione di sette campi

modificati con delle maschere generate dai corrispondenti sette bit dell'ingresso CNGENS, abbrev. di Change Enables. I sette campi sono nRESET, System Control, CPU Control meno nRESET, CPU Bus Control, A, DIN e DOUT. In questo modo si può avere uno snap solo al variare di segnali precisi, come solo quelli del gruppo System Control, per ridurre il numero di snap.

Quando CNG si attiva, la macchina attiva il segnale WR_EN verso SNAP_FIFO scrivendo lo snap compattato in 64 bit. Se la FIFO, dopo quest'operazione, attiva FULL, lo snapper blocca CLK_GEN ed entra nello stato di attesa, IDLE.

Lo scopo dell'entity Z80_REPORTER è solo quello di tradurre in caratteri gli snap che sono presenti sulla SNAP_FIFO. Per cui appena SNAP_FIFO ha uno snap disponibile lo legge e genera il messaggio da stampare. Poi, fingendosi una ROM, lo fornisce a CHAR_WRITER_FIFO che lo carica carattere per carattere in REP_FIFO. Appena la scrittura finisce, Z80_REPORTER ricomincia da capo resettando l'entity.

Capitolo 5 Conclusione

Guardando retrospettivamente la realizzazione di questo minicomputer funzionante si possono notare delle migliori applicabili e si possono definire i prossimi passi.

Per prima cosa, si potrebbe creare un programma per PC così da semplificare ed automatizzare le operazioni con la scheda.

Poi, una futura implementazione migliore del DECODER dello Z80X può essere fatta usando una BRAM come se fosse un PLA e indirizzarla attraverso la parola generata dalla giustapposizione di tutti gli ingressi al DECODER. Questa soluzione ridurrebbe indubbiamente il numero di SLICEs usate però potrebbe incorrere in un altro problema cioè che le BRAM a disposizione possano non bastare per tenere tutte le parole richieste. Inoltre richiederebbe uno sforzo maggiore nella programmazione poiché si lavorerebbe con anonime parole di lunghezza considerevole ad anoniimi indirizzi di memoria aumentando la difficoltà di debug.

Il minicomputer al momento accetta come input solo la scrittura diretta di valori in zone precise della memoria. Per cui non è un input facilmente utilizzabile. Un prossimo passo, dopo aver alleggerito il design liberando spazio, sarebbe quello di inserire un'interfaccia per un altro driver USB a UART. Così si può attaccare direttamente alla scheda una tastiera. Servirebbe una logica che decodifichi gli scan codes ricevuti dalla tastiera in caratteri ASCII e li renda disponibili su una FIFO dedicata così che lo Z80X la veda come una periferica.

In aggiunta si potrebbe implementare un piccolo controllore per la porta VGA presente sulla scheda. L'ambiente di sviluppo della Xilinx mette a disposizione già un IP, ma servirebbe una memoria BRAM dedicata come periferica allo Z80X. Inoltre per velocizzare le operazioni servirebbe implementare Z80-DMA sull'FPGA così da sfruttare i vantaggi di un DMA per spostare i caratteri e le immagini.

Una volta aggiunta la tastiera si potrebbe implementare una calcolatrice. Per esempio si potrebbe passare all'HP-35 che fu una delle prime calcolatrici scientifiche a montare uno Z80 come processore.

L'implementazione di una calcolatrice, oltre ad essere un primo step è anche una forma di omaggio verso il passato di questi microprocessori e al lavoro di persone come Federico Faggin e Masatoshi Shima.

Capitolo 6 Indice analitico

A

A 16	
Accumulatore.....	21
A_HZ.....	37
AF.....	34
ALU	59
architecture	5
architecture body	5
architettura.....	16
ASCIISegsDriver	75
ASIC.....	9

B

B 21	
baudrate	77
BC	21
Block RAM	
BRAM	12
Block RAMs.....	69
BUS request/Acknowledge Cycle	44

C

C 21	
C carry flag.....	21
CAD	4
carry-look ahead	
CLA	32
CHAR_FEEDER_FIFO	72
CHAR_RAM.....	70
CHAR_WRITER_FIFO	73
CircBuffSSegs.....	75
CISC.....	19
CLB	10
CLK_EDGE	37
CLK_GEN.....	87
CLK_NEDGE	37
CLK_PEDGE	37
CMD_CTRL.....	77
concurrent assignement	5
concurrent statement.....	5
conditional assignment	5
CPU bus control	18
CPU control.....	17
CTRL_HZ	37

D

D 16; 21	
----------	--

DAA.....	60
daisy chain	26
DE	21
decimal adjust	29
DECODER.....	42
DONE	44
DOUT_HZ	37
DUT	6
Dynamic RAM	
DRAM.....	17
DYNAMIC_RECOGNIZER	82

E

E 21	
entity	5
entity declaration.....	5
exchange	28
EXT_IR.....	42

F

F 21	
FIFO.....	12; 70
FIFO_RX	71
FIFO_TX.....	71
FIFO_TX_SLAVE.....	71
First-Word Fall-Through	
FWFT	71
FPGA	9
FSMs annidate.....	41

G

general purpose	21
generic.....	5

H

H 21	
H half-carry flag.....	21
HALT	48
Halt Acknowledge.....	49
HDL	4
HL	21

I

I 22	
INCDEC.....	61
INCDEC16.....	61
Input or Output Cycles	56

instantiation.....	5
instruction fetch	
opcode fetch	17
Instruction Opcode Fetch.....	52
Instruction Register	
IR 39	
INT Mode 0	25
INT Mode 1	25
INT Mode 2	25
Interrupt Mode FFs	
IMF.....	22
Interrupt Request/Acknowledge Cycle	57
Interrupt Status FFs	
IFF	22
INTIF	40
INTRO.....	57
IORDWR	56
IP 7	
IX21	
IXIY	63
IY21	

L

L 21	
LIFO	21
LUT.....	10

M

macchina di Mealy.....	40
macchina di Moore.....	40
Main.....	44
Master	43
M-cycle	19
Memory Data Register	
MDR.....	39
Memory Read or Write Cycles.....	54
MEMRDWR	54
MLAST	42

N

N subtract flag.....	21
nBUSACK.....	18
nBUSREQ.....	18
nHALT.....	18
nINT.....	18
nIORQ.....	17
NMI.....	22
NMII.....	40
nMREQ.....	17
nNMI.....	18
Non-Mackable Interrupt Acknowledge.....	49
NOP	48
nRD.....	17
nRESET	17
nRFSH	17

nWAIT.....	18
nWR.....	17

O

OPFET.....	52
organizzazione	16

P

P/V parity-overflow flag.....	21
parity bit.....	77
PC	21
pinout.....	16
PLA	9
PLD	9
port.....	5
Power down acknowledge	49
process	5

R

R 21	
refresh.....	17
REGCEB	68
REGS	62
REP_FIFO	72
Reset Cycle.....	51
restart.....	28
ritorni	28
RUCS7.....	66

S

S sign flag	21
sensitivity list	6
signal	5
slice.....	11
SLICEL.....	11
SLICEM	11
SLICEX	11
SNAP_FIFO	72
sottomacchine	Vedi μ FSM
SP21	
SS_DRIVER.....	75
SSegsASCII	75
STATIC_RECOGNIZER	81
System Control	17

T

T-cycle.....	19
testbench	6
TRIG.....	44

V

VHDL	5
VLSI	4

W

W 34	
WZ.....	34

Z

Z 34	
Z zero flag	21

Z80 IO INTERFAC.....	68
Z80 MEM INTERFACE.....	68
Z80_REPORTER.....	88
Z80_SNAPPER.....	87

M

μ FSM	44
-----------------	----

Capitolo 7 Indice delle figure

Figura 2-1 - Livelli di astrazione relativi agli HDL [7]	4
Figura 2-2 - Esempio di design entity VHDL [7]	5
Figura 2-3 - Processo iterativo di progettazione [3].....	6
Figura 2-4 - Esempio di PLA [9]	9
Figura 2-5 - Schema dei CLBs e dei canali d'interconnessione [11].....	10
Figura 2-6 - Scheda di sviluppo AX309	11
Figura 2-7 – Disposizione delle slices all'interno di un CLB [10]	12
Figura 3-1 - Pubblicità dello Z80 del maggio 1976	14
Figura 3-2 - Forme d'onda degli ingressi e delle uscite dell'Intel 8080° [20].....	15
Figura 3-3 - Pinout dello Z80 [22].....	16
Figura 3-4 - Diagramma a blocchi dello Z80 [22]	19
Figura 3-5 - Registri della CPU Z80 [22]	20
Figura 3-6 - Diagramma di flusso della gestione degli interrupt [26].....	23
Figura 3-7 - Sequenze di servizio di un interrupt mascherabile [24]	24
Figura 3-8 - Sequenza di servizio degli interrupt vettorizzati [24]	25
Figura 3-9 - Esempio di connessione daisy chain. Nell'immagine l'address bus viene sostituito con il data bus per lo z80.....	26
Figura 3-10 - Sommario degli operandi direttamente dalla guida utente dello Z80 [23].....	28
Figura 3-11 - Die dello Z80 con le unità operative evidenziate [27]	30
Figura 3-12 - Schema blocchi dell'ALU [29].....	30
Figura 3-13 - Circuito completo dell'inc/dec [30]	31
Figura 3-14 -Cella Inc/dec a 2 bit base [30].....	32
Figura 3-15 - Data bus interni. In rosso, verde e arancio le tre sezioni del bus a 8 bit. In fucsia, il bus indirizzi a 16 bit [31]	33
Figura 3-16 - Struttura dei registri dello Z80 [29]	33
Figura 3-17 - Pinout entity Z80	36
Figura 3-18 - Diagramma dell'organizzazione dello Z80X che mostra i collegamenti principali.....	38
Figura 3-19 - BUS request/Acknowledge Cycle [22]	43
Figura 3-20 - Diagramma di stato della FSM Main con raggruppati gli stati affini	45
Figura 3-21 - Diagrammi di stato dei cicli FETCH e FETCH EXTIR	46
Figura 3-22 - Diagrammi di stato dei cicli EXiEN, n indica l'indice della fase di esecuzione corrispondente	47
Figura 3-23 - Diagrammi di stato del ciclo HALT CYCLE	48
Figura 3-24 - Halt Acknowledge [22].....	48
Figura 3-25 - Diagrammi di stato dei cicli NMI CYCLE e INT ACK	49
Figura 3-26 - Power-Down Acknowledge [22]	49
Figura 3-27 - Non-Maskable Interrupt Request Operation [22]	50
Figura 3-28 - Reset cycle [22]	50
Figura 3-29 - Diagramma di stato del ciclo RESET CYCLE	51
Figura 3-30 - DIGRAMMA DI STATO DELLA MFSM OPFET	53
Figura 3-31 - Instruction Opcode Fetch [22]	54
Figura 3-32 - Diagramma di stato della uFSM MEMRDWR.....	55
Figura 3-33 - Memory Read or Write Cycles [22].....	55
Figura 3-34 - Diagramma di stato della uFSM IORDWR	56
Figura 3-35 - Input or Output Cycles [22].....	57
Figura 3-36 - Diagramma di stato della uFSM INTRQ	58

Figura 3-37 - Interrupt Request/Acknowledge Cycle [22]	58
Figura 3-38 - Schema dell'ALU	59
Figura 3-39 - Schema dell'INCDEC e INCDEC16.....	61
Figura 3-40 - Schema dell'entity REGS	62
Figura 3-41 - Schema dell'entity IXIY	63
Figura 4-1 - Schema delle unità usate all'interno della scheda AX309.....	65
Figura 4-2 - Schema del sistema di sviluppo RUCS7.....	67
Figura 4-3 - Schema di una BRAM generica	69
Figura 4-4 - Schema di una FIFO generica.....	70
Figura 4-5 - Diagramma di stato di CHAR_FEEDER_FIFO	73
Figura 4-6 - Diagramma di stato di CHAR_WRITER_FIFO.....	74
Figura 4-7 - Diagramma di stato della FSM di CMD_CTRL	79
Figura 4-8 - Diagramma di stato di FIELD_RECOGNIZER	80
Figura 4-9 - Diagramma di stato dell'entity STATIC_RECOGNIZER	82
Figura 4-10 - Diagramma di stato dell'entity DYNAMIC_RECOGNIZER	82

Capitolo 8 Bibliografia

- [1] M. Balch, in "Digital Fundamentals". *Complete Digital Design: A Comprehensive Guide to Digital Electronics and Computer System Architecture. Professional Engineering.*, New York, McGraw-Hill Professional, 2003, p. 122.
- [2] «Texas Instruments TI-84 Plus C Silver Edition,» [Online]. Available: http://www.datamath.org/Graphing/TI-84PLUS_CSE.htm. [Consultato il giorno 13 09 2022].
- [3] W. J. Dally, R. C. Harting e T. M. Aamodt, «The practice of digital system design,» in *Digital Design using VHDL, a system approach*, Cambridge, Cambridge University Press, 2016, pp. 22-35.
- [4] F. Faggin, «Immagini centrali,» in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019.
- [5] F. Faggin, «La mia terza vita. Lo Z80-CPU,» in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019.
- [6] «Apple unveils M2 with breakthrough performance and capabilities,» [Online]. Available: [https://www.apple.com/newsroom/2022/06/apple-unveils-m2-w. \[Consultato il giorno 01 09 2022\].](https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/)
- [7] W. J. Dally, R. C. Harting e T. M. Aamodt, «VHDL descriptions of combinational logic,» in *Digital Design using VHDL, a system approach*, Cambridge, Cambridge University Press, 2016, pp. 130-153.
- [8] W. J. Dally, R. C. Harting e T. M. Aamodt, «The digital abstraction,» in *Digital Design using VHDL, a system approach*, Cambridge, Cambridge University Press, 2016, pp. 3-18.
- [9] W. J. Dally, R. C. Harting e T. M. Aamodt, «Combinational building blocks,» in *Digital Design using VHDL, a system approach*, Cambridge, Cambridge University Press, 2016, pp. 157-198.
- [10] «Spartan-6 FPGA. Configurable Logic Block,» Xilinx, 2010.
- [11] «Spartan-6 FPGA. Block RAM. User Guide,» Xilinx, 2011.
- [12] F. Faggin, «Il primo microprocessore. Intel e il progetto Busicom,» in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 75-83.
- [13] F. Faggin, «Il primo microprocessore. La storia del microprocessore in prospettiva,» in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 113-119.
- [14] F. Faggin, «La mia seconda vita. Il brevetto della SGT,» in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 66-74.

- [15] F. Faggin, «Il primo microprocessore. Funziona!», in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 83-91.
- [16] F. Faggin, «Il primo microprocessore. Il progetto dell'famiglia 4000», in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019.
- [17] F. Faggin, «Il primo microprocessore. Annuncio del microprocessore al mondo», in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 97-102.
- [18] H. Tohya, in *Switching Mode Circuit Analysis and Design: Innovative Methodology by Novel Solitary Electromagnetic Wave Theory*, Bentham Science Publishers, 2013, p. 4.
- [19] F. Faggin, «La mia terza vita. La battaglia degli '80», in *Silicio. Dall'invenzione del microprocessore alla nuova scienza della consapevolezza*, Milano, Mondadori, 2019, pp. 140-145.
- [20] S. Buso, Introduzione alle applicazioni industriali di Microcontrollori e DSP, Bologna: Società Editrice Esculapio, 2018, 2015.
- [21] W. Stallings, Computer Organization and Architecture. Designing for Performance. Tenth Edition, Harlow: Pearson Education Limited, 2016.
- [22] Zilog, «Datasheet Z8400/Z84C00 NMOS/CMOS», Zilog.
- [23] Zilog, «Z80 Microprocessor. Z80 CPU. User manual», Zilog, 2016.
- [24] «THE Z80 FAMILY PROGRAM. INTERRUPT STRUCTURE», 1978.
- [25] K. Shirrif, «Posts related to Z80», [Online]. Available: <http://www.righto.com/search/label/Z-80>. [Consultato il giorno 13 09 2022].
- [26] K. Shirrif, «The Z-80 has a 4-bit ALU. Here's how it works.», [Online]. Available: <http://www.righto.com/2013/09/the-z-80-has-4-bit-alu-heres-how-it.html>. [Consultato il giorno 13 09 2022].
- [27] K. Shirrif, «The Z-80's 16-bit increment/decrement circuit reverse engineered.», [Online]. Available: <http://www.righto.com/2013/11/the-z-80s-16-bit-incrementdecrement.html>. [Consultato il giorno 13 09 2022].
- [28] K. Shirriff, «Why the Z-80's data pins are scrambled.», [Online]. Available: <http://www.righto.com/2014/09/why-z-80s-data-pins-are-scrambled.html>. [Consultato il giorno 13 09 2022].
- [29] K. Shirriff, «Down to the silicon: how the Z80's registers are implemented.», [Online]. Available: <http://www.righto.com/2014/10/how-z80s-registers-are-implemented-down.html>. [Consultato il giorno 13 09 2022].
- [30] S. Young, «The Undocumented Z80 Documented.», 2005.

- [31] L. C. e. al., The Zynq Book: Embedded Processing Withe ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC, Strathclyde Academic Media, 2014.
- [32] Intel, «Datasheet Intel 8080A/8080A-1/8080A-2,» Intel, 1986.
- [33] G. Devic, «A Z80 From the Ground Up,» [Online]. Available: <https://baltazarstudios.com/z80-ground/>. [Consultato il giorno 13 09 2022].