Università degli studi di Modena e Reggio Emilia
Facoltà di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica (270/04)

PROVA FINALE

SIMULAZIONE E REALIZZAZIONE IN VHDL DI UNA CPU A 32 BIT

Tutor:
Prof. Ing. Costantino Grana

Candidato:
Guido Borghi

Anno Accademico 2011 - 2012

Indice

Introduzione **5**

Capitolo 1 - La CPU nei calcolatori elettronici **9**

1.1 Descrizione funzionale di una CPU 10

1.2 La CPU come rete logica 11

Capitolo 2 - Programmi utilizzati **13**

2.1 Logisim 14

2.2 ISE Design Suite 15

2.3 XStools 16

Capitolo 3 - Introduzione ad ADE32 ..... **17**

3.1 Scelte architetturali 18

3.2 Interfaccia esterna 18

Capitolo 4 - Microarchitettura di ADE32 ..... **21**

4.1 Schema funzionale 22

4.2 Datapath 22

4.2.1 Registri *22*

4.2.2 ALU (Arithmetic Logic Unit) *24*

4.2.3 Visione di insieme *31*

4.3 Istruzioni 33

4.3.1 Formato *33*

4.3.2 Microistruzioni *33*

4.3.3 Ciclo delle istruzioni *33*

4.3.4 Istruzioni monadiche *34*

4.3.5 Istruzioni per l'accesso in memoria *35*

4.3.6 Istruzioni aritmetico-logiche diadiche *35*

4.3.7 Parallelizzazione del caricamento del MAR *36*

4.3.8 Istruzioni di salto *38*

4.3.9 Codice operativo e tempi di clock *38*

4.4 Control Unit 39

4.4.1 Implementazione cablata *40*

4.4.2 Implementazione micro programmata *40*

4.4.3 Segnali di controllo *43*

Capitolo 5 - Assembly e Assembler 49

5.1 Caratteristiche dell'Assembly di ADE32 50

5.2 Assemblatore in C 50

5.3 Assemblatore con Flex e Bison 53

Capitolo 6 - Esempi di programmi per l'architettura 55

6.1 Somma di due numeri 56

6.2 Incremento da 0 a 10 57

Capitolo 7 - Sintesi del progetto 59

7.1 Introduzione al VHDL 60

7.2 FPGA 62

7.3 Realizzazione dei componenti 64

7.3.1 Registro 32, 20 e 4 bit 65

7.3.2 Buffer tri-state a 32 bit 67

7.3.3 Bit-extender 20-32 bit 68

7.3.4 Bit-extender 32-20 bit 68

7.3.5 ALU 69

7.3.6 Data path 71

7.3.7 Control Unit 72

7.3.8 ADE32 75

7.3.9 Motherboard 75

Conclusioni 79

Bibliografia 81

Sitografia 81

Introduzione

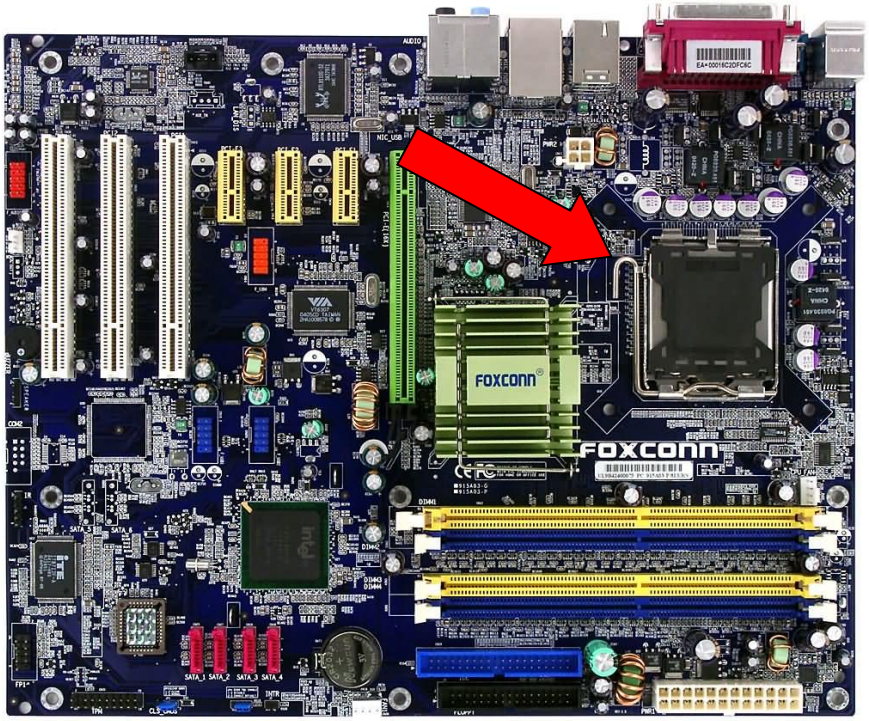

Figura 0.1: Locazione della CPU in una generica scheda madre di un computer

Osservando l'immagine riportata in Figura 0.1 probabilmente, per un non addetto ai lavori, prevale l'idea di una forte complessità: è difficile capire come un ammasso, in apparenza disordinato, di circuiti possa rendere possibile, per esempio, la visualizzazione di immagini e video su uno schermo, la navigazione in internet, la riproduzione di file audio, l'esecuzione di complessi calcoli ecc. Ed è forse ancora più difficile comprendere come il cosiddetto processore, detto anche CPU (*Central Processing Unit*) sia in grado, attraverso impulsi elettrici, di dare vita a un computer.
Anche uno studente di materie informatiche potrebbe chiedersi come realmente funzioni il cervello digitale di un calcolatore elettronico non fermandosi alle spiegazioni "panoramiche" fornite dal corso di studi, ma andando a investigare su ogni minimo componente della struttura del circuito o sulla vita e sul viaggio di ogni singolo bit di informazione attraverso i canali di comunicazione: in tale modo è davvero possibile comprendere come, in effetti, alla base del funzionamento anche del più moderno dei calcolatori ci siano dei semplici interruttori (ovvero transistor in ambito elettronico). Si giunge così a spogliare i circuiti di quella componente quasi "magica" e si comprende a fondo il loro funzionamento.
Con questo elaborato ci si propone, con le dovute semplificazioni, di comprendere a fondo la struttura e il funzionamento di un processore, attraverso la sua progettazione, simulazione e realizzazione effettuate con gli opportuni strumenti.

I temi trattati sono così suddivisi:

- **Capitolo 1**
  In apertura viene presentato il concetto di CPU, di cui viene data una descrizione funzionale, con qualche accenno alla sua evoluzione storica, per poi passare alla descrizione dal punto di vista di una rete logica.

Lo scopo è quello di esporre i concetti fondamentali alla corretta comprensione degli argomenti trattati nei capitoli successivi e di chiarire i termini che si utilizzano nel corso dell'elaborato: si cerca quindi di favorire la comprensione da parte del lettore.

- **Capitolo 2**
  Nel secondo capitolo vengono descritti i programmi che si sono utilizzati sia nella fase di progettazione e simulazione del processore sia nella fase di realizzazione. Di questi si presenta la schermata principale che generalmente viene suddivisa graficamente in più parti per facilitarne la spiegazione di utilizzo. I programmi presentati sono tutti gratuiti e liberamente scaricabili dai siti internet riportati nella Sitografia.
  I programmi sono *Logisim,* con il quale è stato realizzato il progetto e simulato virtualmente il funzionamento del processore, *ISE Design Suite*, per tutta la parte relativa al linguaggio di descrizione dell'hardware, ovvero il VHDL, ISim, facente parte dello stesso pacchetto di installazione di ISE Design Suite, che viene utilizzato per la simulazione dei circuiti logici digitali e infine *XStools*, programma necessario per il trasferimento fisico del *bitstream* generato tramite il codice VHDL sulla scheda utilizzata.

- **Capitolo 3**
  In questo capitolo si introduce in maniera specifica la CPU la cui progettazione e realizzazione è lo scopo della trattazione, ovvero *ADE32* (Architettura di Esempio a 32 bit).
  Nel primo paragrafo si illustrano le scelte architetturali che, come vedremo, si orientano alla semplicità e praticità di progettazione più che alle prestazioni e alla ottimizzazione dal punto di vista circuitale.
  Nel secondo paragrafo si descrive ADE32 come *black-box*, ovvero come scatola chiusa, in cui sono noti solamente le porte di ingresso e uscita del sistema: si elencano quindi i bus che collegano il processore ai dispositivi presenti sulla scheda madre, ovvero l'interfaccia esterna.

- **Capitolo 4**
  Il quarto capitolo prende in esame la microarchitettura di ADE32, ovvero la struttura interna del processore, elencando i dispositivi e i blocchi logici presenti. In modo particolare il processore viene diviso logicamente in due aree distinte: la prima area è deputata al controllo logico e prende il nome di *Control Unit* (CU), la seconda area ha altri compiti ugualmente importanti, come quelli di acquisire le istruzioni e i dati, passare le istruzioni all'Unità di Controllo, calcolare l'indirizzo successivo col quale reperire l'istruzione ed eseguire le operazioni di ALU, e prende il nome di *data path*. Nei paragrafi successivi vengono definite le istruzioni del processore, di cui si illustra il formato, la tipologia (istruzioni monadiche, per l'accesso in memoria, aritmetico-logiche diadiche e di salto condizionato e non condizionato) e le microistruzioni di cui sono composte. Infine si propone una tabella riassuntiva di tutte le istruzioni riconosciute dal processore con il loro codice operativo sia in formato binario che esadecimale.

- **Capitolo 5**
  In questo capitolo si delineano le caratteristiche del linguaggio *assembly* di ADE32, ovvero del linguaggio mnemonico che permette di dialogare con il

processore senza dover utilizzare il poco intuitivo linguaggio macchina (in formato binario).
Si introduce il concetto di *assembler*, ovvero il programma necessario a tradurre le istruzioni mnemoniche del linguaggio assembler in un *bitstream* riconosciuto dal processore.
Si descrive la realizzazione del programma che viene effettuata con due metodi distinti: il primo direttamente nel linguaggio C, il secondo attraverso l'uso dei *tools Flex* e *Bison*. Le due soluzioni proposte vengono confrontate e vengono messi a fuoco gli aspetti positivo e negativi di ciascun metodo.

- **Capitolo 6**
  Nel sesto capitolo vengono presentati due semplici programmi pensati per ADE32: una somma fra due numeri interi e un incremento unitario. Lo scopo è quello di osservare interamente il percorso che parte dall'idea del programma per arrivare al risultato prodotto dal processore. Infine proprio grazie a tali programmi è possibile compiere alcune osservazioni sul funzionamento reale di ADE32.

- **Capitolo 7**
  In questo capitolo viene descritta la sintesi del processore, ovvero la sua realizzazione *hardware* mediante il linguaggio VHDL e la scheda Xess ospitante una FPGA.
  Nel primo paragrafo è presente una breve introduzione al VHDL, che ha lo scopo di far familiarizzare il lettore con gli aspetti fondamentali della programmazione hardware. Successivamente si descrive il processo di implementazione dei circuiti digitali sulla FPGA, illustrandone tutti i passaggi. Infine si descrive la realizzazione dei componenti logici necessari alla realizzazione del processore, riportandone il codice VHDL e il grafico della simulazione eseguita con ISim.

Infine, per una lettura scorrevole, si propone di seguito una tabella contenente l'elenco dei principali acronimi, in ordine alfabetico, utilizzati nella stesura dell'elaborato.

ACC	Accumulator
ADE32	Architettura di Esempio a 32 bit
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CU	Control Unit
FPGA	Field Programmable Gate Array
I/O	Input Output
IR	Instruction Register
MAR	Memory Address Register
MDR	Memory Data Register
MUX	Multiplexer
PC	Program Counter
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

Capitolo 1 - La CPU nei calcolatori elettronici

In questo capitolo si introduce il concetto di CPU (*Central Processing Unit*), ovvero il "cuore pulsante" di ogni calcolatore elettronico. Ne verrà data inizialmente una descrizione funzionale, elencando i suoi compiti fondamentali, fornendo uno schema logico esemplificativo e anche qualche cenno storico; successivamente verrà descritta dal punto di vista di una rete logica, ovvero come insieme di dispositivi elettronici interconnessi che realizzano un'elaborazione elettronica.

1.1 Descrizione funzionale di una CPU

La CPU (*Central Processing Unit*, "unità centrale di calcolo") può essere vista come il cuore di ogni calcolatore digitale; si caratterizza principalmente per sovraintendere a tutte le funzionalità del calcolatore, dialogando, tramite appositi bus, con la memoria centrale e con gli altri dispositivi di I/O connessi.

Spesso ci si riferisce alla CPU con il termine processore o *microprocessore*: tale definizione nasce storicamente nel momento in cui risultò possibile integrare su un solo chip tutti i componenti della CPU che prima risiedevano su entità separate fisicamente tra di loro; dalla sua comparsa il microprocessore ha avuto una rapida evoluzione, seguendo con una buona approssimazione la legge di Moore (1965) che prevedeva il raddoppio del numero di transistor integrati sullo stesso chip ogni 18 mesi. Vale la pena notare che tale legge si è mantenuta più o meno valida fino ai nostri giorni, nonostante attualmente vi sia una disputa riguardo ai limiti della integrazione sul silicio. E' infine doveroso menzionare il primo microprocessore della storia, costruito dall'Intel nel 1971, il 4004.

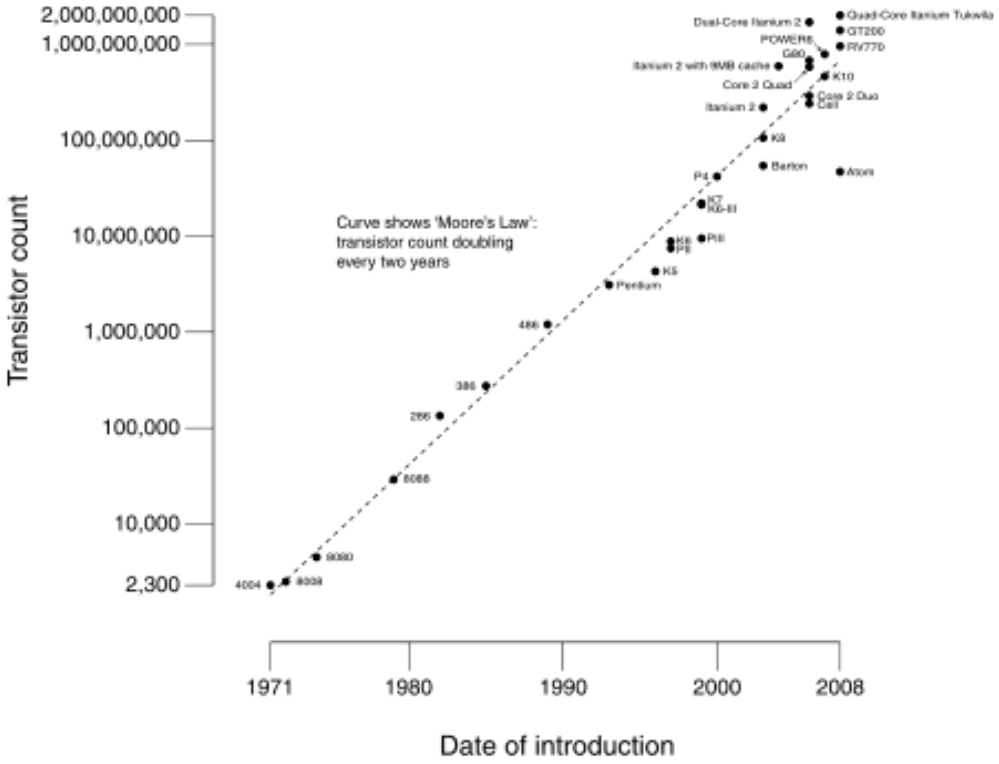

Figura 1.1: Legge di Moore e conteggio dei transistor integrati dal 1971 al 2008

Tornando alla descrizione funzionale, il compito di una CPU è quello di eseguire le istruzioni dei programmi che risiedono in memoria. Nello specifico ha quattro compiti fondamentali:

1. Gestione delle istruzioni;
2. Elaborazione dei dati;
3. Gestione della memorizzazione dei dati;
4. Gestione del trasferimento dei dati.

A fronte quindi di una serie di dati forniti in input si ha la produzione di un output formato dai dati elaborati internamente al processore; tale elaborazione dipende sostanzialmente dagli ingressi e dallo stato interno della CPU.
Tale funzionamento è rappresentato in forma schematica nella Figura 1.1.
Possiamo immediatamente notare che la CPU risulta a livello logico divisa in due parti: ALU (*Arithmetic Logic Unit*) e CU (*Control Unit*), connesse tra di loro mediante dei bus di comunicazione interni, ognuna dotata di un compito specifico che sarà analizzato in maniera dettagliata nei capitoli seguenti.

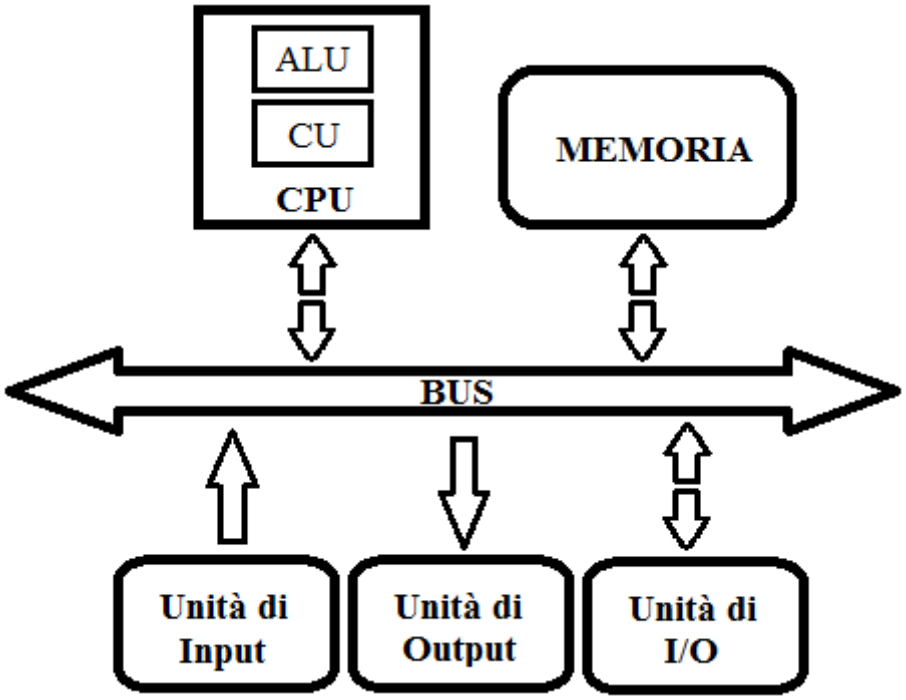

Figura 1.2: Schema funzionale di una CPU

1.2 La CPU come rete logica

La CPU si può vedere anche come una rete logica che esegue in modo ripetitivo e infinito un ciclo che ha un'importanza centrale in un qualsiasi calcolatore e che consiste in tre fasi:

- **Fetch**: reperimento dell'istruzione da eseguire: l'istruzione viene letta nella memoria tramite un apposito indirizzo e viene memorizzata in uno speciale registro solitamente chiamato *Instruction Register* (IR), mentre il suo indirizzo viene memorizzato in un altro registro tipicamente chiamato *Program Counter* (PC), per tenere traccia dell'indirizzo della successiva istruzione da caricare;
- **Decode**: l'istruzione viene decodificata, il processore attua i provvedimenti atti ad avviare correttamente l'esecuzione dell'istruzione stessa;
- **Execute**: l'istruzione viene eseguita; al termine di questa fase si procede generalmente con il caricamento di una nuova istruzione o dato dalla memoria centrale.

Nello specifico durante la fase di fetch la CPU fornisce un preciso identificativo per la cella di memoria alla quale accedere, ovvero l'indirizzo, con il quale preleva l'istruzione dalla memoria; in seguito, una rete logica interna al processore, la cosiddetta Control Unit, si occupa della decodifica dell'istruzione (fase di Decode) e predispone il

processore ad affrontare uno specifico tipo di operazione che poi verrà appunto eseguita durante la fase di execute.
Quest'ultima fase tipicamente prevede tre tipi di istruzioni che possono essere così classificate:

1. **Istruzioni di ALU**: hanno questo nome in quanto prevedono l'utilizzo del dispositivo logico-aritmetico e genericamente la presenza di uno o più operandi .
2. **Istruzioni di Load/Store**: sono istruzioni che prevedono l'accesso alla memoria centrale o, anche, ai dispositivi di I/O connessi al calcolatore.
3. **Istruzioni di Controllo**: sono particolari istruzioni che servono a gestire le unità interne alla CPU (o eventualmente altre unità interne al calcolatore); vengono utilizzate per esempio per l'esecuzione di salti condizionati, salti non condizionati, le chiamate al sistema operativo e operazioni simili.

Osservando la tipologia delle possibili istruzioni possiamo osservare che la CPU comunica con la memoria principalmente per tre motivi:

1. **Lettura di un operando**: dato un indirizzo, il processore legge il corrispondente dato in memoria;
2. **Scrittura di un operando**: dato un indirizzo, il processore scrive il dato in memoria;
3. **Reperimento di un'istruzione**: dato un indirizzo, il processore recupera l'istruzione relativa dalla memoria (la fase di Fetch comune a tutte le istruzioni).

Risulta quindi necessario prevedere un metodo per trasferire le istruzioni e dati dall'interno all'esterno del processore e viceversa, essendo le istruzione mai modificate all'interno della stessa CPU.

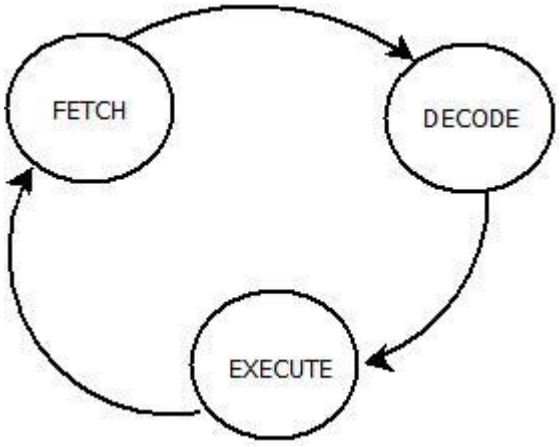

Figura 1.3: La CPU come rete logica

Capitolo 2 - Programmi utilizzati

In questo capitolo vengono presentati i software utilizzati per lo svolgimento dell'elaborato, *Logisim*, *ISE Design Suite* e *XStools*; di questi programmi non si vuole proporre una guida completa ed esaustiva ma si vuole soltanto fornire gli strumenti per avvicinarsi a questi ambienti di programmazione, per comprendere correttamente il significato degli schemi, delle figure e dei codici riportati nei paragrafi successivi.
Si analizza inoltre anche il programma *ISim*, un ambiente di test e simulazione dei circuiti digitali, contenuto nel pacchetto di installazione *WebPACK* assieme al programma ISE Design Suite.
Di tutti i programmi, tranne di XStools, si riportano le schermate più significative.

2.1 Logisim

Logisim è un programma scritto in Java, per cui non necessita di installazione, multipiattaforma, gratuito, ideato per la progettazione e la simulazione in tempo reale dei circuiti logici digitali.

Le sue principali qualità sono la semplicità e la velocità d'uso.

L'ambiente di programmazione integra l'ambiente di lavoro, che dividiamo, per una maggiore chiarezza, idealmente in tre parti riportate in Figura 2.1.

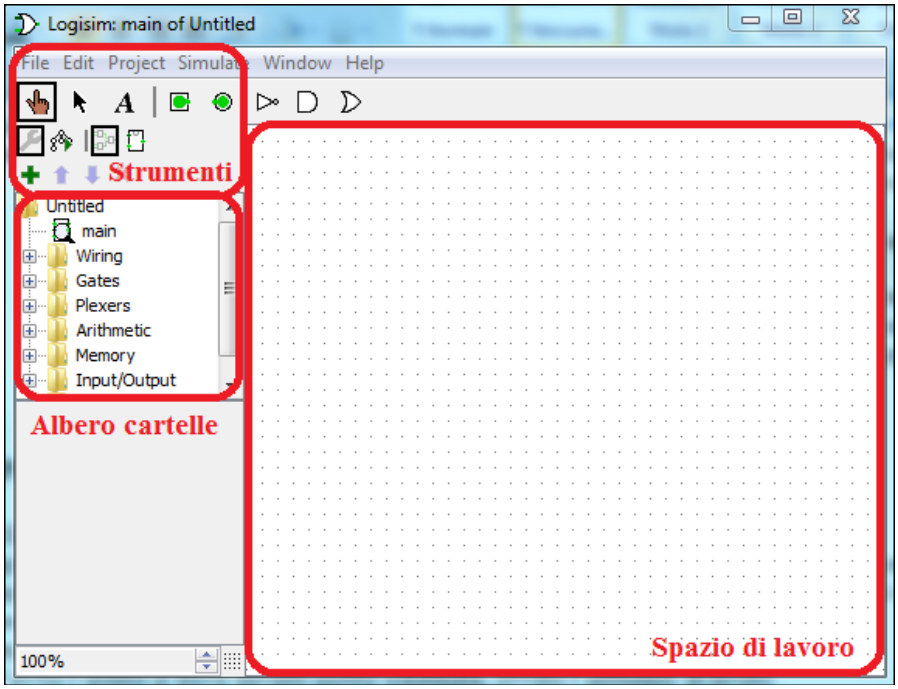

Figura 2.1: Ambiente di lavoro di Logisim

A sinistra, nell'*Albero cartelle*, vengono mostrati tutti i progetti di cui si compone il progetto principale e una serie di cartelle che contengono gli elementi che possono essere utilizzati nella realizzazione del circuito logico; per esempio si hanno a disposizione elementi di memoria (RAM, ROM ecc.), vari tipi di porte logiche (AND, OR, NAND, NOR ecc.), multiplexer, demultiplexer e componenti aritmetici (Half Adder, Full Adder ecc).

Nello *Spazio di lavoro* l'utente disegna gli schemi logici attraverso un semplice *drag-and-drop* degli elementi che si trovano a lato.

Infine nella parte *Strumenti*, oltre alle solite opzioni di apertura, salvataggio ecc, si trovano le opzioni per creare etichette, per gestire gli schemi e per abilitare o disabilitare l'emulazione in tempo reale del circuito disegnato.

Nel complesso uno circuito digitale avrà sempre uno o più input, rappresentati da dei quadrati riportanti all' interno il loro valore logico, e una o più uscite, rappresentate da dei cerchi anch'essi riportanti all'interno il valore assunto, come possiamo vedere in Figura 2.2 (in questo esempio sono input ed output da 1 bit).

Figura 2.2: Esempio di pin di input ed output a 1 bit

I dispositivi sono collegati tra di loro mediante linee che possono assumere diversi colori in base al valore che assumono durante l'emulazione; nel caso di creazione di bus di comunicazione l'unico colore disponibile è il nero.
Eventuali errori di progetto vengono segnalati mediante apposite scritte sul fondo dello spazio di lavoro e da particolari colori (arancione) nel disegno del progetto.

2.2 ISE Design Suite

ISE Design Suite è un ambiente di programmazione realizzato dalla casa produttrice *Xilinx* ed esiste sia in versione gratuita (quella che noi utilizzeremo) che in versione a pagamento che, naturalmente, offre un maggior numero di funzionalità rispetto a quella di base; è inserito in un pacchetto di tools più ampio chiamato WebPACK, scaricabile dal sito della stessa casa produttrice.

L'aspetto dell'ambiente di lavoro, una volta avviato il programma, risulta essere semplice: è diviso idealmente in quattro sezioni, quella più grande, lo *Spazio di lavoro*, ospita l'editor del codice VHDL e offre la possibilità di agire direttamente sul testo con alcune funzioni come l'indentazione e l'inserimento di commenti; a fianco di questa finestra ci sono due piccoli specchietti, *Albero componenti* e *Funzioni*, che permettono di prendere visione di tutti i componenti che formano il progetto, di navigare fra i vari progetti dell'utente e di abilitare particolari funzioni, come il controllo sintattico del codice scritto e il lancio della simulazione.
I messaggi forniti dal sistema all'utente, compresi eventuali messaggi di errore e di warning vengono stampati nella finestra *Console errori* posta in basso.

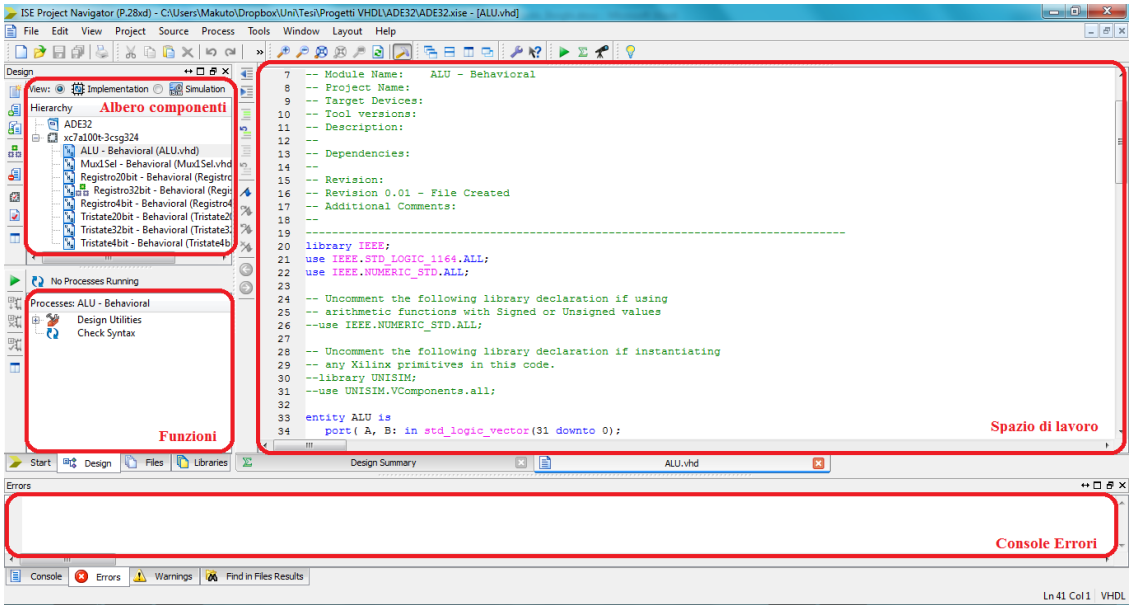

Figura 2.3: Ambiente di lavoro di ISE Design Suite

Una caratteristica molto interessante è rappresentata dal fatto che è possibile progettare un circuito sia attraverso l'uso del VHDL, sia attraverso l'utilizzo di un ambiente di programmazione grafico, molto simile come concetto a quello di Logisim, che permette di lavorare con componenti di base definiti dall'utente o precaricati come esempi.
E' inoltre possibile testare virtualmente i circuiti realizzati tramite il lancio di un software che è contenuto assieme a ISE Design Suite all'interno del WebPACK, ovvero ISim.

ISim costruisce un grafico nel quale sono riportati tutti i segnali di ingresso con il loro valore e i relativi segnali di uscita del circuito, in modo tale che l'utente ha la possibilità di verificare il suo corretto funzionamento e controllare in maniera accurata l'andamento dei segnali durante i cicli di funzionamento del dispositivo realizzato.
In Figura 2.4 viene riportato a titolo di esempio il grafico generato da uno dei tanti test condotti durante la fase di sintesi del processore ADE32.

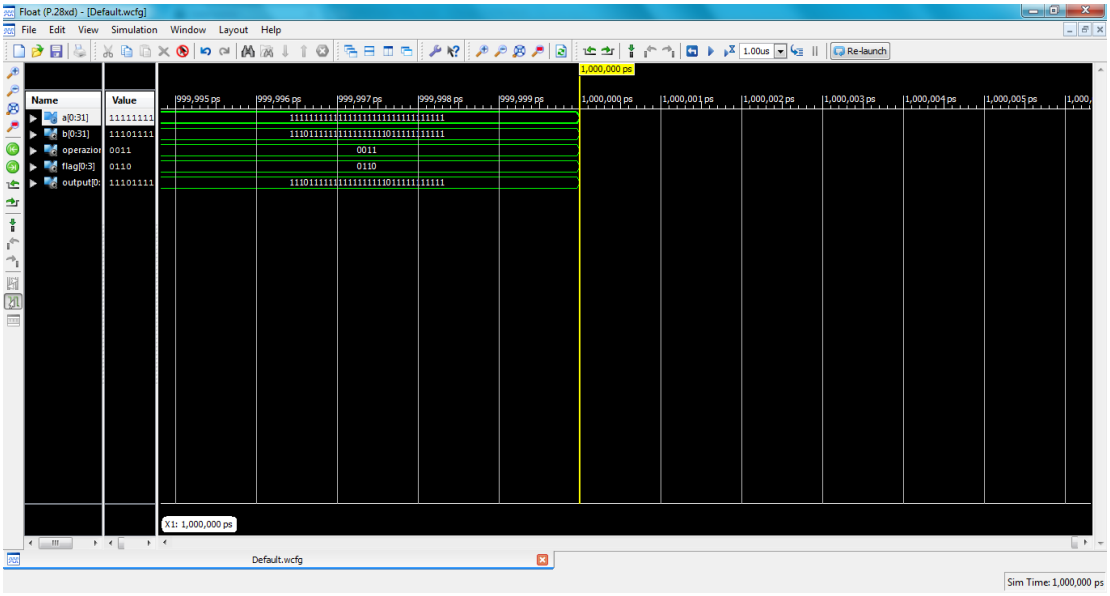

Figura 2.4: ISim, Ambiente di simulazione e test dei circuiti realizzati

2.3 XStools

XStools è un pacchetto di programmi, scaricabile gratuitamente dal sito della *Xess*, che include le seguenti utilities:

1. **GXSTESTS**: permette di testare il funzionamento della propria scheda e il riconoscimento da parte del computer connesso;
2. **GXSSETCLK**: permette di settare la frequenza di clock degli oscillatori presenti sulle schede da valori diversi da quelli di default;
3. **GXSLOAD**: si incarica del trasferimento fisico del bitstream, generato ad esempio tramite ISE Design Suite, sulla scheda (cfr. Capitolo 7);
4. **GXSPORT**: permette di inviare dei segnali logici ai pin della scheda commutando i segnali generati dal computer.

Tutti i programmi presentano un'interfaccia *user-friendly* e sono di rapido apprendimento; vengono installati tutti assieme sul computer con un'unica installazione. Ovviamente sono stati progettati e realizzati per operare con le schede prodotte dalla stessa casa, la Xess.

Capitolo 3 - Introduzione ad ADE32

In questo capitolo si introducono gli aspetti fondamentali dell'Architettura di Esempio ADE32.

Nel primo paragrafo si illustrano le scelte progettuali ed architetturali e fatte le relative conseguenze nel campo della struttura del processore.

Nel secondo paragrafo viene presentato ADE32 come *black-box*, ovvero come scatola chiusa in cui sono noti solamente i segnali di ingresso e di uscita; infine vengono illustrati i dispositivi presenti sulla scheda madre e i bus di comunicazione (bus dati e bus indirizzi) che collegano questi alla CPU.

3.1 Scelte architetturali

L'Architettura di Esempio ADE32 nasce per scopi didattici, privilegia quindi la semplicità e la chiarezza all'ottimizzazione del progetto e alla velocità di funzionamento.
Chiaramente non ha scopi commerciali.
Tendendo ai principi sopra riportati l'ADE32 nasce come un'architettura ad accumulatore, ovvero è presente un unico registro nel quale si accumulano tutti i risultati delle operazioni di ALU; inoltre tale registro è utilizzato nelle operazioni di ricezione e invio dati.
E' una struttura infatti molto semplice da realizzare, ma al tempo stesso rappresenta anche il collo di bottiglia del sistema.
Da notare come ormai la maggior parte dei moderni calcolatori adotti architetture a *set di registri*, molto flessibili e generali, che prevedono l'utilizzo di più locazioni di memorie temporanee che posso essere accedute o lette in maniera quasi del tutto indipendente.
Il processore ha un parallelismo interno a 32 bit, ovvero il numero di bit di parola che vengono elaborati in modo parallelo dall'unità di elaborazione e che vengono memorizzati e trasferiti è 32.
Si segue a livello generale l'architettura di *Von Neumann*, ovvero dati e istruzioni sono entrambi contenuti in una memoria leggibile e scrivibile esterna al processore: l'accesso alla memoria si esegue in base all'indirizzo e non al tipo di dato; questa tipologia impone che venga utilizzato un unico bus per la comunicazione tra processore e memorie esterne, motivo per cui non è possibile caricare un dato mentre si ha la lettura di un'istruzione.
Nel caso in cui dati e istruzioni risiedessero in due memorie separate si parlerebbe di architettura di *Harvard*, che quindi è dotata di due differenti bus per la comunicazione tra processore, memoria dati e memoria istruzioni e può eseguire il caricamento di un'istruzione e di un dato in modo parallelo.
Questo tipo di scelta, come si vedrà in seguito, avrà notevoli influenze sul progetto.

3.2 Interfaccia esterna

Il processore comunica con l'esterno mediante un bus dati a 32 bit e un bus indirizzi a 20 bit, quindi la massima memoria indirizzata è pari a 1 Mega (2^{20} bit).
Tale scelta risponde alla volontà di evitare istruzioni con lunghezza variabile.
Il bit più significativo del bus degli indirizzi (quindi il 20) viene inviato all'unità di Controllo I/O e viene sdoppiato attraverso il suo negato.
La memoria ha un parallelismo a 32 bit, quindi a ogni indirizzo corrisponde una parola a 32 bit.
Il processore riceve in ingresso altri due segnali, uno di reset per azzerare il contenuto dei registri interni (fondamentale per esempio in fase di avvio) e il segnale di clock, necessario per la sincronizzazione dei componenti interni ed esterni al processore stesso.
In particolare il segnale di reset è collegato a ogni registro di ogni sotto circuito che dispone appunto di un ingresso di reset del suo contenuto; il segnale di clock arriva alla memoria RAM e, come nel caso del reset, a ogni singolo registro per abilitare il campionamento dei dati che avviene a ogni fronte di salita del clock, e a quasi tutti gli altri dispositivi presenti.
Ci sono infine altre due unità esterne al processore: è presente un controllo di I/O per abilitare il dispositivo sul quale il processore compie la lettura e la scrittura (che quindi può essere la RAM o un unità generica di output come un display a sette segmenti

esadecimale, quest'ultimo utilizzato nel progetto) e un registro di I/O che permette sia di scrivere ma anche leggere il valore presente nell'unità di output.
La struttura del registro I/O può essere osservata in Figura 3.1. L'aggiunta del circuito si è resa necessaria per poter leggere il valore contenuto nel registro di output.

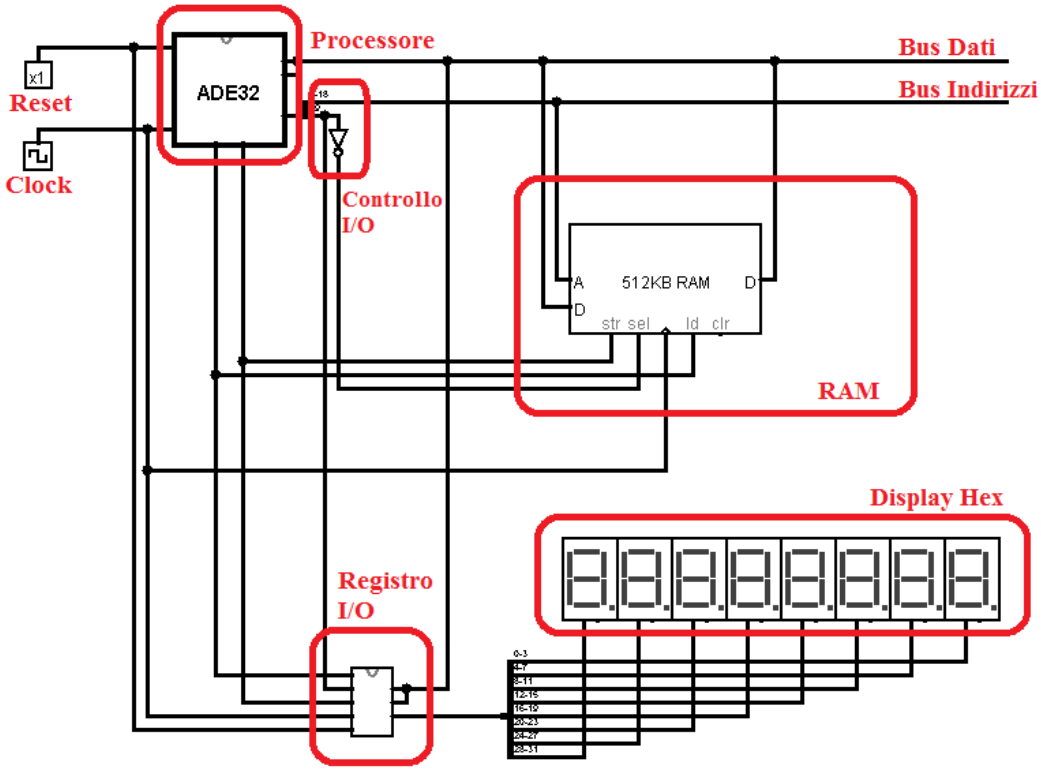

Figura 3.1: Interfaccia esterna - Scheda Madre ADE32

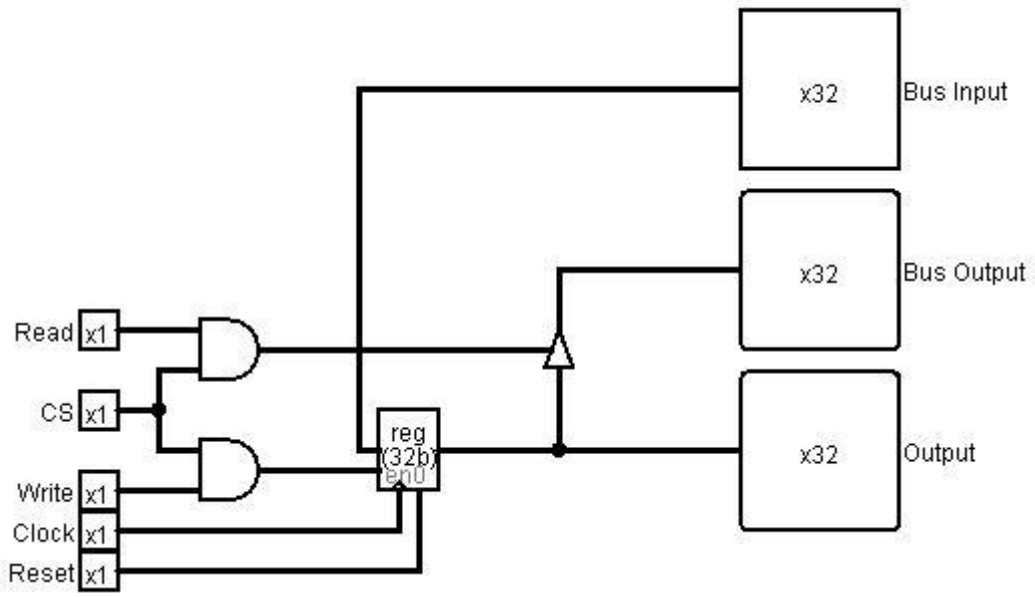

Figura 1.2: Schema logico del Registro I/O

Capitolo 4 - Microarchitettura di ADE32

In questo capitolo viene illustrata la microarchitettura del processore ADE32, in cui compaiono i blocchi funzionali interni (come i registri, l'Unità Aritmetico Logica, la Control Unit) con le relative dipendenze e funzionamento, i bus di comunicazione, il percorso dei dati, concretizzando in tale modo le scelte architetturali esposte nel capitolo precedente.

Viene dedicato spazio anche al discorso delle istruzioni per il processore: se ne illustra il formato e il ciclo, si analizzano le microistruzioni, se ne presentano le varie tipologie, ovvero le istruzioni monadiche, le istruzioni per l'accesso in memoria, le istruzioni aritmetico-logiche diadiche e le istruzioni di salto condizionato e non condizionato.

Infine si analizza l'implementazione micro programmata, fornendone schemi esemplificativi, dell'Unità di Controllo, sia nella versione senza salti che in quella con i salti condizionati.

4.1 Schema funzionale

Come abbiamo visto in precedenza la CPU al suo interno è costituita a livello logico da due parti funzionali: la Control Unit (CU) e il data path; sono entrambe poste in comunicazione con dei bus: un bus si occupa del passaggio dal data path alla Control Unit dei bit del registro Flag, il secondo bus si occupa, sempre nel medesimo verso, del passaggio dell'opcode, infine un terzo bus è impiegato per trasportare il segnale di controllo dalla Control Unit al data path.
Il bus dati e il bus indirizzi che abbiamo visto nel paragrafo precedente sono direttamente collegati al data path. In uscita inoltre si hanno anche i segnali di controllo per la gestione della memoria, ovvero Read e Write.
Infine sono presenti in entrata due segnali fondamentali, ovvero il CLOCK, che come abbiamo visto ha il compito di far funzionare e sincronizzare tutti i dispositivi all'interno della CPU, e il RESET, che ha il compito di resettare il contenuto di tutti i registri ponendolo a zero.

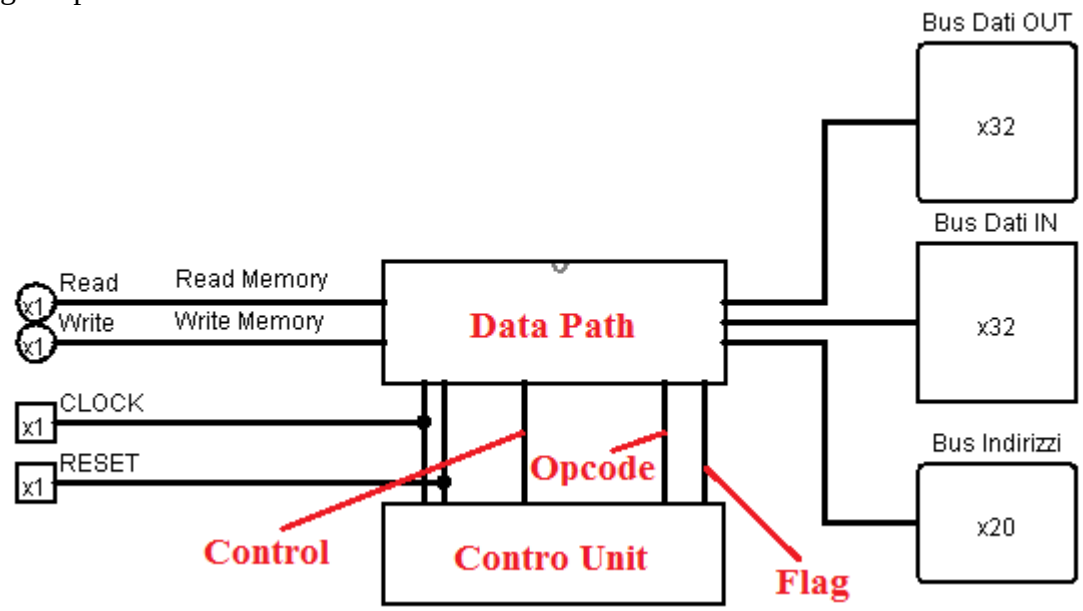

Figura 4.1: Schema funzionale di ADE32

4.2 Datapath

Il data path è denominato anche Unità di Elaborazione, poiché ha il compito di acquisire le istruzioni e i dati, passare le istruzioni all'unità di controllo, calcolare l'indirizzo successivo col quale reperire l'istruzione ed eseguire le operazioni di ALU.
Può essere visto come un insieme di unità di calcolo e di memorizzazione come per esempio i registri, bus di comunicazione e componenti logico-aritmetiche.

4.2.1 Registri

Analizziamo ora nello specifico la struttura interna del Datapath di ADE32.
L'Unità di Elaborazione è dotata di un bus interno a 32 bit che ha la funzione di collegare i registri alla ALU.
I registri, ovvero piccole unità di memorizzazione temporanee che campionano il dato in input al fronte di salita del clock, sono otto e hanno funzioni precise. Sono i seguenti:

- **ACC** (Accumulatore): registro a 32 bit di accumulo dei dati e degli operandi utilizzati durante le operazioni di ALU e durante le operazioni di lettura e

scrittura della memoria. E' il registro fondamentale nell'ambito di un'architettura ad accumulatore.

- **IR** (Instruction Register): contiene l'istruzione che è in fase di esecuzione, memorizza il valore durante la fase di reperimento dell'istruzione (fetch). Essendo le istruzioni di lunghezza fissa tale registro è a 32 bit (non vi può essere nessuna istruzione la cui lunghezza supera i 32 bit).
- **MDR** (Memory Data Register): porta intermedia tra la memoria e il bus interno della CPU; ha il compito di memorizzare temporaneamente il dato prelevato o letto dalla memoria. E' un registro a 32 bit (poiché naturalmente deve essere della stessa dimensione del bus dati).
- **MAR** (Memory Address Register): porta di passaggio degli indirizzi in uscita della CPU diretti alla memoria. E' un registro a 20 bit (deve essere della stessa dimensione del bus degli indirizzi).
- **PC** (Program Counter): contiene l'indirizzo della prossima istruzione da eseguire; è quindi aggiornato automaticamente tramite un'operazione di ALU per puntare sempre all'istruzione successiva, anche in caso di istruzione di salto o di controllo. Assieme al MAR è l'unico registro a 20 bit.
- **ALUA e ALUOUT**: sono registri da 32 bit relativi all'ALU; in particolare il primo contiene il dato che deve essere passato come primo input all'unità logica, il secondo invece contiene il risultato prodotto in output dalla stessa unità; vedremo come il registro ALUOUT svolga una funzione regolatrice nei confronti dell'attività dell'ALU.
- **FLAG**: è un piccolo registro a 4 bit che contiene i bit di condizione (Zero, Carry, Negativo e Overflow), generati dalle operazioni logico-aritmetiche eseguite nell'ALU; è un registro importante in quanto viene utilizzato direttamente dall'Unità di Controllo per l'esecuzione delle istruzioni di salto condizionato.

I registri, ad eccezione di FLAG, sono tutti connessi direttamente con il bus interno.
La regolazione dei processi di lettura o scrittura nei confronti del bus interno è affidata a dei segnali di abilitazione in lettura (che indichiamo con la sigla OE, ovvero *Output Enable*), che sono collegati a dei buffer tri-state che si comportano come un interruttore comandato. Tutti i registri sono dotati di tali segnali ($ACC_{OE,}$ $MDR_{OE,}$ $MAR_{OE,}$ $IR_{OE,}$ $PC_{OE,}$ $ALUOUT_{OE}$); inoltre i registri sono dotati anche di un segnale di abilitazione in scrittura (che indichiamo con la sigla IE, ovvero *Input Enable*), anch'essi collegati a dei buffer tri-state ($ACC_{IE,}$ $MDR_{IE,}$ $MAR_{IE,}$ $IR_{IE,}$ PC_{IE}, $ALUA_{IE,}$ $ALUOUT_{IE}$).
Il registro ALUA è sempre abilitato in scrittura verso la ALU.
Il registro MDR rappresenta un caso particolare in quanto dispone anche di due segnali ($MDR_{BUSOE,}$ MDR_{BUSIE}) che regolano l'attivazione dell'input o output nei confronti della memoria e dei dispositivi di output generici e vengono portati all'esterno del processore come segnali di Write (WR) e Read (RD): naturalmente, per il corretto funzionamento della memoria, non possono mai essere attivi contemporaneamente.

4.2.2 ALU (Arithmetic Logic Unit)

Volgiamo ora il nostro sguardo alla componente logico-aritmetica, l'ALU.

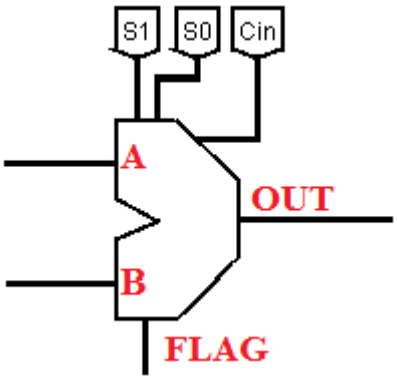

Figura 4.2: Schema logico di un ALU

L'ALU è caratterizzata da due porte di input (A e B), dalle quali entrano gli operandi, e da una porta di uscita (OUT) dalla quale esce il risultato elaborato.
In entrata della porta A dell'ALU è presente, anche se non riportato in Figura 4.2, un multiplexer a due ingressi comandato dal segnale ZA che consente di inviare in input uno zero oppure l'output di ALUA: come si vedrà in seguito, questa selezione di input è fondamentale per le operazioni interne all'ALU.
La porta B invece è sempre connessa al bus interno.
Una seconda uscita dell'ALU è costituita dal registro FLAG, dotato di un segnale di abilitazione dell'input ($FLAG_{IE}$), che campiona i bit di condizione provenienti dall'ALU, che verranno impiegati dall'Unità di Controllo.
Possiamo notare come la ALU non sia regolata nel suo funzionamento da un segnale di clock esterno, motivo per cui questo circuito elabora in maniera continua gli operandi che riceve in ingresso dalle porte A e B; tale comportamento però non costituisce un problema in quanto i risultati prodotti dall'ALU vengono prelevati solo nel momento in cui risulta attivo in input il registro ALUOUT, che quindi si occupa di regolare il funzionamento dell'Unità logico-aritmetica campionando il risultato solo nel momento necessario.
La ALU, oltre alle due porte che abbiamo visto poco sopra, ha in ingresso un segnale di riporto (Cin) e due segnali (S1 e S0) che consentono di eseguire due operazioni di tipo logico e due operazioni di tipo aritmetico come possiamo vedere nella tabella qui sotto:

S1	S0	Operazione
0	0	A and B
0	1	A or B
1	0	A + B + Cin
1	1	A + B' + Cin

Invertiamo ora momentaneamente il nostro approccio *top-down* fino a questo punto seguito per adottare un approccio *bottom-up*, che ci risulta particolarmente utile per analizzare la struttura interna dell'ALU.
Andremo prima a costruire una ALU a 1 bit per poi arrivare alla sintesi di un ALU a 32 bit attraverso l'unione dei blocchi precedentemente realizzati.
La Figura 4.3 rappresenta il blocco fondamentale per la costruzione della ALU, nello specifico è un'ALU a 1 bit.
In alto a sinistra ci sono i due input da 1 bit, Input A e Input B, e l'input di Carry, Carry In.

Nel settore in basso a sinistra è presente un decodificatore che, in base ai due segnali di controllo S1 e S2, attiva una delle quattro operazioni possibili attraverso una porta logica AND posta in uscita di ogni linea di output. Le operazioni logiche di AND ed OR vengono effettuate dalle relative due porte presenti nell'Unità Logica, mentre nella parte in basso a destra è presente un sommatore (Full Adder), che tiene conto dell'eventuale riporto in ingresso, per le operazioni di tipo aritmetico; un eventuale riporto viene trasmesso in output attraverso il segnale di output Carry Out.
Il negato del secondo input è ottenuto mediante l'attivazione della porta logica EXOR posta dinanzi a Input B.

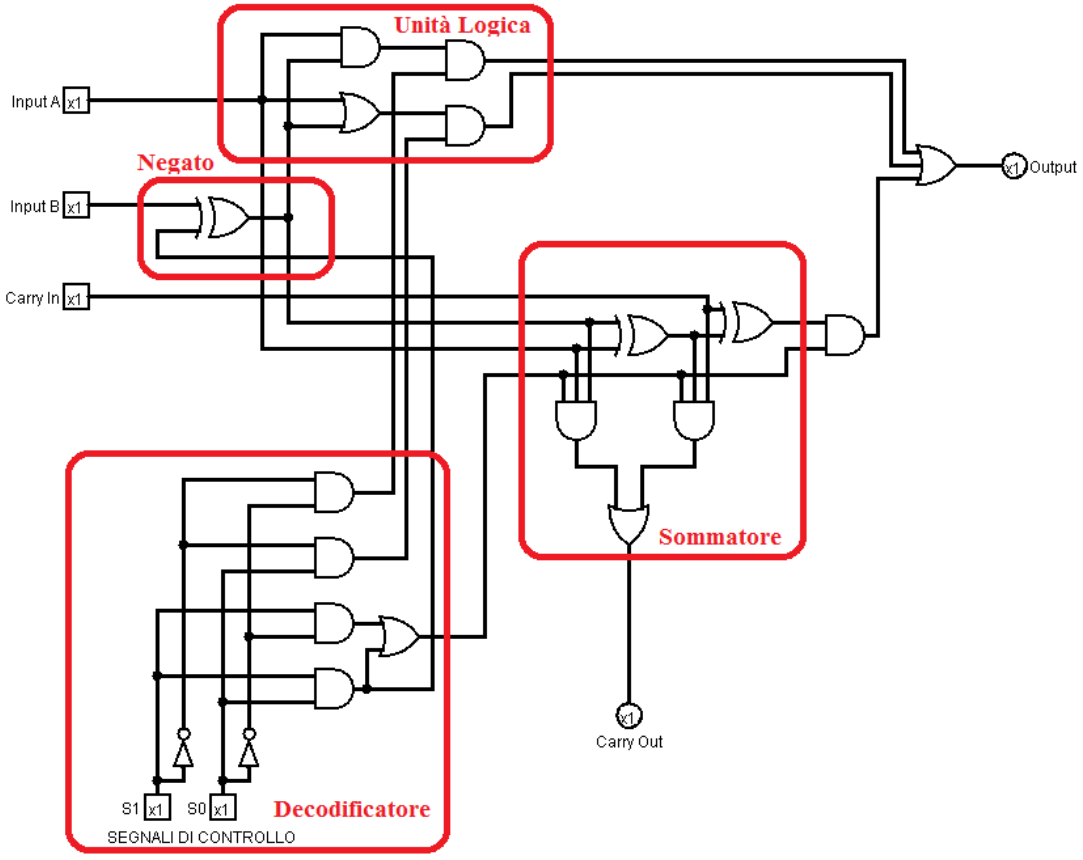

Figura 4.3: Schema logico di una ALU a 1 bit

Per meglio comprendere il funzionamento dei blocchi riportiamo il simbolo e le tabelle di verità delle porte logiche e di alcuni componenti che sono stati impiegati per la realizzazione del circuito.

INPUT		**OUTPUT**
A	**B**	A **AND** B
0	0	0
0	1	0
1	0	0
1	1	1

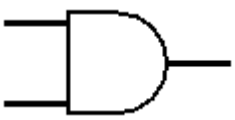

Figura 4.4: Simbolo AND

INPUT		OUTPUT
A	**B**	A **OR** B
0	0	0
0	1	1
1	0	1
1	1	1

Figura 4.5: Simbolo OR

INPUT		OUTPUT
A	**B**	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

Figura 4.6: Simbolo XOR

INPUT	OUTPUT
0	1
1	0

Figura 4.7: Simbolo NOT

INPUT			OUTPUT	
A	**B**	**Cin**	**SOMMA**	**Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

La vista del circuito come modulo appare in questo modo:

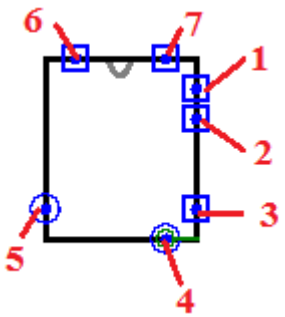

Figura 4.8: Modulo ALU a 1 bit

La mappatura delle porte è la seguente:

1. S1
2. S2
3. Cin
4. Output
5. Cout
6. Input B
7. Input A

La costruzione dell'ALU a 8 bit risulta a questo punto immediata, in quanto è sufficiente mettere assieme 8 blocchi di ALU a 1 bit, ottenendo il risultato riportato in Figura 4.9.
Abbiamo quindi in ingresso due segnali (A e B) a 8 bit, il segnale di riporto (Cin) e i segnali di controllo (S1 e S0) che devono ovviamente essere comuni a tutti i blocchi. In uscita da ognuno di questi abbiamo il segnale di riporto (Cout) collegato in cascata: in questo modo il riporto si trasmette fino all'ultimo blocco presente con il segnale Cout[31], che verrà impiegato per rilevare la presenza del carry nelle operazioni aritmetiche.
I segnali rimanenti sono utilizzati solo nell'ultima ALU a 8 bit e servono per generare i segnali di flag. Sono Cout[30], ovvero il penultimo segnale di riporto, BS, ovvero il bit più significativo dell'Output e Zero, ottenuto inviando gli output dei blocchi in una porta NOR che rileva se l'operazione ha prodotto un risultato uguale a zero.

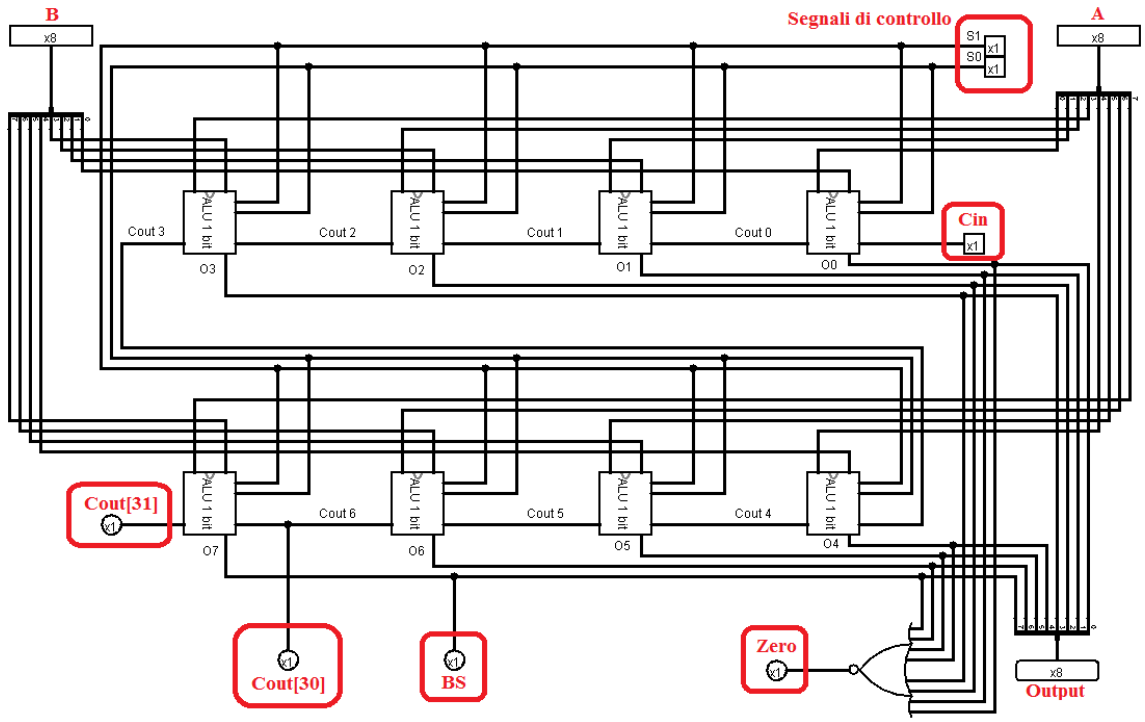

Figura 4.9: Schema logico di una ALU a 8 bit

Riportiamo per completezza e per una migliore comprensione la tabella di verità e il simbolo della porta logica NOR:

INPUT		OUTPUT
A	**B**	A **NOR** B
0	0	1
0	1	0
1	0	0
1	1	0

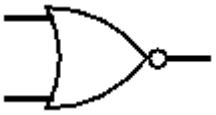

Figura 4.10: Simbolo NOR

La vista del circuito come modulo appare in questo modo:

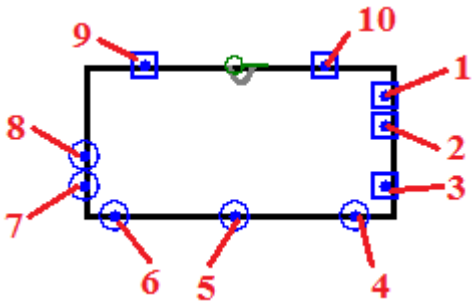

Figura 4.11: Modulo ALU a 8 bit

La mappatura delle porte è la seguente:

1. S1
2. S0
3. Cin
4. Zero
5. Output
6. BS
7. Cout[31]
8. Cout[30]
9. Input B
10. Input A

Ricordiamo che le porte 8 (Cout[30]) e 6 (BS) risultano essere utilizzato solamente nell'ultimo blocco, quello che riguarda i bit più significativi della parola di 32 bit.

Come è facile intuire la realizzazione dell'ALU a 32 bit si ottiene unendo 4 blocchi di ALU a 8 bit che abbiamo creato in precedenza.
Abbiamo ora due segnali di ingresso a 32 bit (A e B), i soliti due segnali di controllo (S1 e S0) e il segnali di riporto, un segnale di uscita a 32 bit (Output) e un bus a 4 bit che raccoglie i bit di stato generati dalle operazioni dell'ALU.
Il primo bit è il bit di zero che è ottenuto tramite la somma logica (Porta 1) dei segnali di Zero provenienti dalle 4 unità da 8 bit. Il secondo bit è quello di segno ed è semplicemente formato dal bit più significativo (il bit 31). Il terzo bit riporta la presenza di un eventuale riporto in uscita; infine il quarto bit, ottenuto tramite lo XOR del Carry Out 30 e 31 (Porta 2), identifica un eventuale stato di Overflow, che si verifica nel momento in cui il risultato non può essere rappresentato dai 32 bit a disposizione del segnale di output.

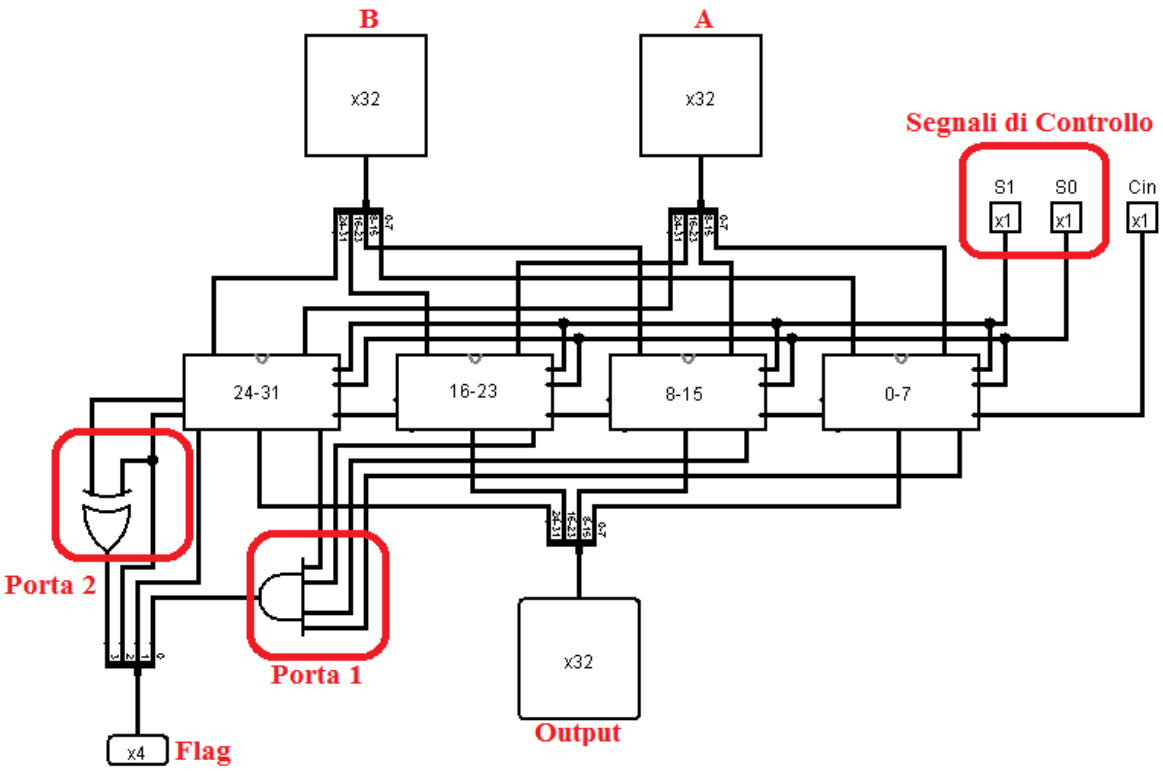

Figura 4.12: Schema logico di una ALU a 32 bit

La vista del circuito come modulo appare in questo modo:

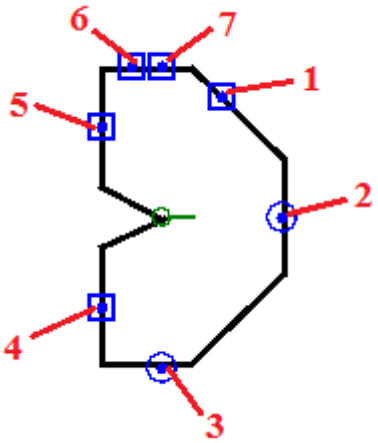

Figura 4.13: Modulo ALU a 32 bit

La mappatura delle porte è la seguente:

1. Cin
2. Output
3. Flag
4. Input B
5. Input A
6. S1
7. S0

Per riassumere, i segnali di controllo necessari all'interno del data path sono i seguenti:

0	ACC_{OE}
1	ACC_{IE}
2	MDR_{OE}
3	MDR_{IE}
4	MDR_{BUSOE}
5	MDR_{BUSIE}
6	MAR_{OE}
7	MAR_{IE}
8	IR_{OE}
9	IR_{IE}
10	PC_{OE}
11	PC_{IE}
12	$ALUA_{IE}$
13	$ALUOUT_{OE}$
14	$ALUOUT_{IE}$
15	$FLAG_{IE}$
16	S1
17	S2
18	ZA
19	CIN

Mettendo assieme i segnali tutti i segnali che riguardano direttamente l'ALU, ovvero S1, S0, ZA e Cin, si può definire l'intero insieme della tipologia delle istruzioni implementabili e implementate all'interno di ADE32, riportato nella tabella che segue.

S1	S0	ZA	Cin	Operazione	Risultato
0	0	0	-	0 AND B	Sempre 0 (può servire per avere uno zero da usare con il decremento)
0	0	1	-	A AND B	AND
0	1	0	-	0 OR B	B (inutile)
0	1	1	-	A OR B	OR
1	0	0	0	0 + B	B (inutile)
1	0	0	1	0 + B + 1	Incremento
1	0	1	0	A + B	Somma
1	0	1	1	A + B + 1	Somma e incrementa (inutile)
1	1	0	0	0 + B'	NOT
1	1	0	1	0 + B' + 1	Negativo, cioè –B
1	1	1	0	A + B'	Sottrae e decrementa (può servire per fare un decremento)
1	1	1	1	A + B' + 1	Sottrazione (A - B)

Ogni operazione di ALU necessita di un registro abilitato in lettura sul bus interno per avere un dato nella porta B.
A causa dell'adozione di una architettura ad accumulatore le istruzioni di ALU vengono sempre scritte specificando prima il dato presente nel canale A (che può essere il contenuto del registro ALUA o zero), poi viene indicata l'operazione , poi il canale B,

ovvero il registro abilitato in lettura ed infine, se necessario, il valore del riporto in ingresso Cin.
Il risultato viene sempre scritto in ALUOUT.

La rappresentazione dei numeri negativi avviene mediante la celebre codifica del *Complemento a due* che ha il grande vantaggio di utilizzare un solo tipo di circuito, un sommatore, per realizzare le operazioni di addizione e sottrazione, rendendo quindi particolarmente semplice e snello il progetto del processore.
Con tale codifica il bit più significativo determina il segno del numero binario, ha quindi peso sia negativo che positivo, da cui deriva che tutti i numeri binari positivi iniziano con lo 0 e, specularmente, tutti i numeri binari negativi iniziano con un 1. Per tale motivo i numeri rappresentabili con n bit vanno da -2^{n-1} a 2^{n-1}.
Il Complemento a due si basa essenzialmente sul calcolo dell'opposto di un numero binario che si ottiene invertendo tutti i sui bit (applicando quindi l'operazione logica NOT) e aggiungendo 1 (ad esempio: $(31)_{10} = (00011111)_2$, $(-31)_{10} = (11100001)_2$).
Ha il vantaggio di avere una rappresentazione univoca dello zero, proprietà che non si ha con un'altra celebre codifica utilizzata per la gestione dei numeri negativi binari, ovvero la codifica *Modulo e segno*, che prevede solo l'aggiunta di un bit di segno in testa al numero binario per indicare se questo è positivo o negativo.
Senza addentrarci eccessivamente nello studio dell'algebra booleana e della codifica Complemento a due, possiamo affermare che la sottrazione fra due numeri (per esempio A e B) viene implementata come una semplice addizione tra il primo addendo e l'opposto del secondo, a cui si aggiunge il valore 1 ($(A - B)_{10} = (A + B' + 1)_2$).

4.2.3 Visione di insieme

Riportiamo ora la visione di insieme del data path che è costituita dall'unione dei componenti che abbiamo analizzato fino a questo punto, ovvero la ALU a 32 bit e i vari registri (ACC, IR, MDR, MAR, PC, ALUA, ALUOUT), collegati tra di loro dal bus interno a 32 bit. I collegamenti con l'esterno sono affidati al bus dati a 32 bit e al bus indirizzi a 20 bit. Per motivi legati alla progettazione su Logisim, in Figura 4.14 sono riportati due bus dati, uno per l'input e l'altro per l'output.
Da notare come per collegare i registri MAR e PC, entrambi da 20 bit, con il bus interno a 32 bit, sia necessario utilizzare dei bit-extender, ovvero particolari dispositivi che hanno il compito di accettare in ingresso valori a 20 bit e trasformali in valori a 32 bit mediante l'aggiunta di zeri in testa al numero (e viceversa per il passaggio da 32 bit a 20 bit).
Nel settore in basso a destra sono presenti i segnali di controllo provenienti dall'Unità di Controllo che per motivi di chiarezza sono stati collegati ai buffer tri-state tramite dei tunnel, una sorta di collegamenti "wireless" virtuali.

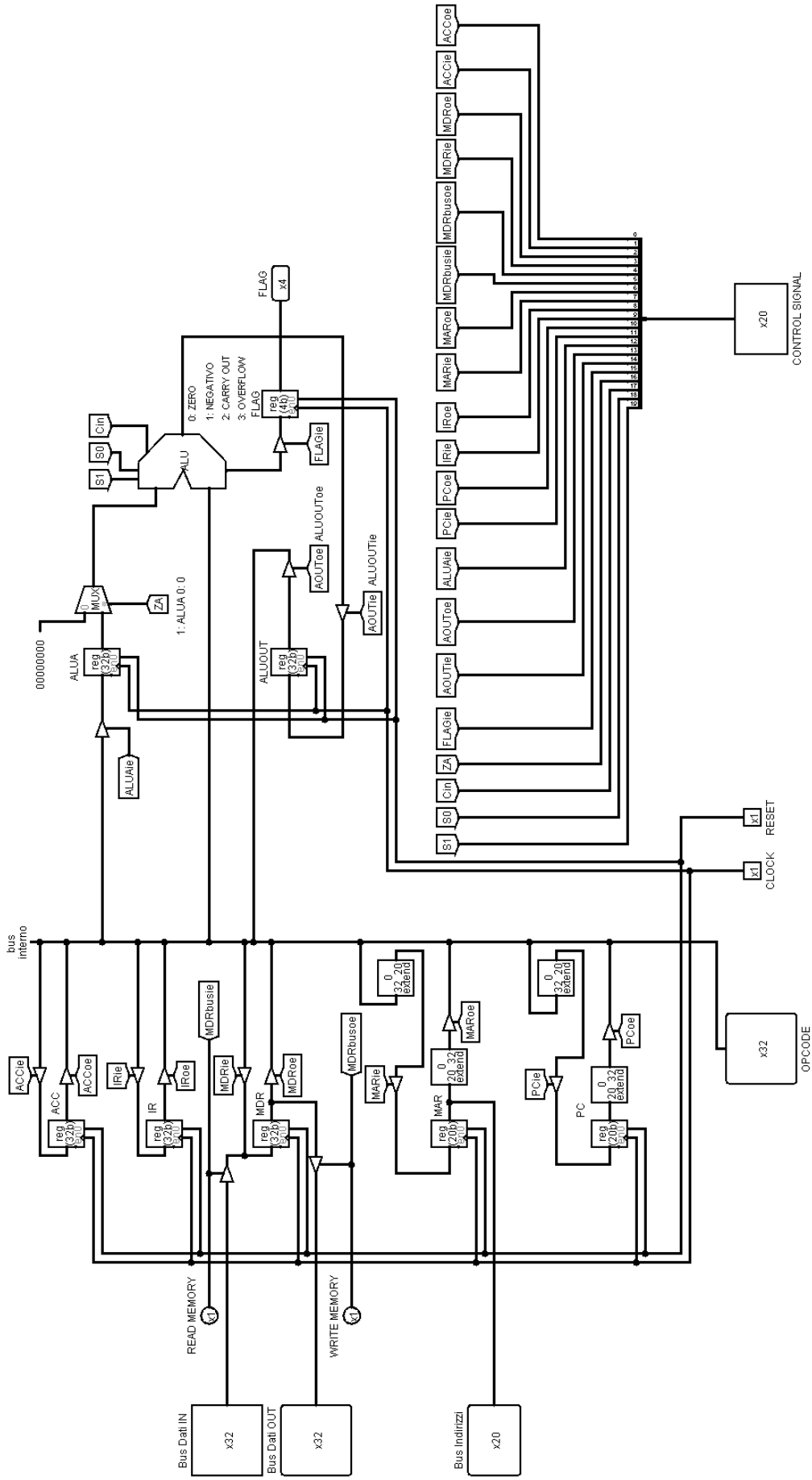

Figura 4.14: Data path di ADE32

4.3 Istruzioni

4.3.1 Formato

Il formato delle istruzioni è orientato alla semplificazione dell'unità di controllo.
Come prima si è affermato le istruzioni hanno lunghezza fissa a 32 bit; questi bit sono suddivisi in due sottocampi, il primo campo a 12 bit contiene il codice operativo (Opcode), ovvero i bit impiegati dall'Unità di controllo per l'elaborazione dei segnali di controllo, e il secondo campo a 20 bit che contiene l'indirizzo, la cui funzione dipende dall'istruzione implementata (può essere per esempio un operando di un'operazione aritmetica, come può anche essere un indirizzo a una certa locazione di memoria).
La semplicità del formato delle istruzioni naturalmente va a discapito del realismo del processore.

31 **20**	**19** **0**
Codice Operativo (opcode)	Indirizzo (addr)

4.3.2 Microistruzioni

Le microistruzioni sono le azioni che coordinano il funzionamento dei vari componenti del data path, come i registri, l'ALU e la lettura e scrittura sulla memoria: possono essere viste anche come l'insieme dei segnali presenti all'interno del data path che vengono attivati in maniera controllata e sincronizzata.
Ogni istruzione comporta l'esecuzione di una o più microistruzioni; un insieme di microistruzioni viene definito microprogramma. La parte hardware che si occupa della traduzione delle istruzioni nelle loro rispettive microistruzioni è l'Unità di Controllo.

4.3.3 Ciclo delle istruzioni

Come abbiamo visto nel paragrafo 1.2 ogni operazione è il risultato del susseguirsi di tre fasi tra di loro distinte: fetch, decode ed execute.
In questa architettura la fase di decode, che a volte viene considerata una sottofase di fetch, è limitata all'incremento del Program Counter, dato che la decodifica non coinvolge nessun registro di stato.
Per la descrizione delle microistruzioni facciamo uso di una semplice notazione riportata nella tabella seguente:

M[add] ← dato	Scrittura di un dato a un dato indirizzo [add] in memoria
dato ← M[add]	Lettura a un dato indirizzo [add] in memoria
(FLAG)	Aggiornamento del registro Flag

Fetch

Le microistruzioni di questa fase sono comuni a tutte le istruzioni:

MAR ← PC
MDR ← M[MAR]
IR ← MDR

Bisogna inoltre incrementare il valore del Program Counter (fase di Decode):

ALUOUT ← 0 + PC + 1
PC ← ALUOUT

La fase di fetch definitiva quindi diventa:

MAR ← PC, ALUOUT ← 0 + PC + 1
MDR ← M[MAR], PC ← ALUOUT
IR ← MDR

Execute
Essendo questa un'architettura ad accumulatore tutte le istruzioni lavoreranno sempre con la memoria, ad eccezione delle istruzioni monadiche che riguardano il solo accumulatore. Sono quindi necessarie due operazioni di lettura e scrittura dell'accumulatore e le operazioni aritmetiche e logiche tra l'accumulatore e un dato in memoria.

4.3.4 Istruzioni monadiche

Elenchiamo di seguito le istruzioni monadiche, ovvero quelle istruzioni che prevedono il solo utilizzo dell'accumulatore e nessuna operazione con la memoria, sia in lettura che in scrittura.

- **Set (SET)**: imposta un valore a 20 bit, esteso a 32 bit con degli zeri, nell'accumulatore. Ovviamente il valore impostato deve essere positivo. Una soluzione semplice e abbastanza flessibile per eseguire questa istruzione è quella di eliminare i 12 bit più significativi dell'istruzione memorizzando l'istruzione stessa nel MAR (registro a 20 bit), per poi trasferire il suo contenuto, ovvero i 20 bit meno significativi, nell'accumulatore.
  MAR ← IR(add)
  ACC ← MAR

- **Incremento (INC)**: incrementa di uno il contenuto dell'accumulatore e non prevede l'utilizzo di nessun parametro. E' una classica istruzione ALU priva di parametri.
  ALUOUT ← 0 + ACC + 1 (FLAG)
  ACC ← ALUOUT

- **Decremento (DEC)**: utilizza la sottrazione con decremento vista nella tabella, ovvero somma all'accumulatore il negato di zero che in complemento a due è -1. Il canale B della ALU deve essere impostato a zero, che viene generato mediante l'uso di un AND con zero. Da notare che non è necessario attendere il corretto trasferimento di ACC in ALUA per preparare ALUOUT con uno zero, essendo il canale B assolutamente ininfluente e quindi si può utilizzare ACC per pilotare la porta B della ALU e caricare ALUA. In questo caso ALUOUT viene sia utilizzato come registro sorgente che come registro destinazione. Infine l'aggiornamento del registro Flag è presente solamente sulla seconda operazione di ALU.
  ALUA ← ACC, ALUOUT ← 0 and ACC
  ALUOUT ← ALUA + ALUOUT' (FLAG)
  ACC ← ALUOUT

- **Negativo (NEG)**: pone nell'accumulatore il negato aritmetico, ovvero l'operazione ACC = 0 - ACC.
  ALUOUT ← 0 + ACC' + 1 (FLAG)
  ACC ← ALUOUT

- **Complemento (NOT)**: pone nell'accumulatore il suo complemento logico bit a bit; per fare questo sfruttiamo la possibilità di negare la porta B da parte dell'ALU.
  ALUOUT ← 0 + ACC' (FLAG)

ACC ← ALUOUT

4.3.5 Istruzioni per l'accesso in memoria

- **Lettura (LD)**: legge il contenuto della memoria all'indirizzo specificato come parametro e lo scrive nell'accumulatore.
  MAR ← IR(add)
  MDR ← M[MAR]
  ACC ← MDR

- **Scrittura (ST)**: similmente all'istruzione di load, scrive il contenuto dell'accumulatore in memoria all'indirizzo specificato come parametro.
  MAR ← IR(add)
  MDR ← ACC
  M[MAR] ← MDR

- **Lettura indiretta (tramite puntatore) (LDI)**: legge il contenuto in memoria all'indirizzo specificato come parametro, ottenendo un puntatore che viene utilizzato come indirizzo con il quale leggere in memoria il dato e portarlo nell'accumulatore.
  MAR ← IR(add)
  MDR ← M[MAR]
  MAR ← MDR
  MDR ← M[MAR]
  ACC ← MDR

- **Scrittura indiretta (tramite puntatore) (STI)**: in modo simile all'istruzione precedente, legge il contenuto in memoria all'indirizzo specificato come parametro, ottenendo un puntatore che viene utilizzato come indirizzo con il quale scrivere in memoria il contenuto dell'accumulatore.
  MAR ← IR(add)
  MDR ← M[MAR]
  MAR ← MDR
  MDR ← ACC
  M[MAR] ← MDR

4.3.6 Istruzioni aritmetico-logiche diadiche

Tutte queste istruzioni hanno una struttura simile, ovvero il caricamento di ALUA con il contenuto del registro ACC, l'invio alla seconda porta dell'ALU del contenuto del registro MDR e la scrittura del risultato in ALUOUT; si può notare inoltre come sia possibile rendere parallelo il momento di lettura dell'operando e il caricamento del registro ALUA con il contenuto dell'accumulatore.

- **Prodotto logico (AND)**
  MAR ← IR(add)
  MDR ← M[MAR], ALUA ← ACC
  ALUOUT ← ALUA and MDR (FLAG)
  ACC ← ALUOUT

- **Somma logica (OR)**
  MAR ← IR(add)
  MDR ← M[MAR], ALUA ← ACC
  ALUOUT ← ALUA OR MDR (FLAG)
  ACC ← ALUOUT

- **Somma (ADD)**
  MAR ← IR(add)
  MDR ← M[MAR], ALUA ← ACC
  ALUOUT ← ALUA + MDR (FLAG)
  ACC ← ALUOUT

- **Sottrazione (SUB)**
  MAR ← IR(add)
  MDR ← M[MAR], ALUA ← ACC
  ALUOUT ← ALUA + MDR' + 1 (FLAG)
  ACC ← ALUOUT

- **Comparazione (CMP)**
  MAR ← IR(add)
  MDR ← M[MAR], ALUA ← ACC
  ALUOUT ← ALUA + MDR' + 1 (FLAG)

4.3.7 Parallelizzazione del caricamento del MAR

Possiamo operare un'importante ottimizzazione di quanto illustrato finora osservando che le istruzioni SET, LD, ST, LDI, STI, AND, OR, ADD, SUB E CMP hanno la prima microistruzione in comune, ovvero il caricamento dell'indirizzo presente nell'IR nel MAR. Inoltre abbiamo già notato che se un registro sta pilotando un bus, più di un dispositivo può leggere e memorizzare il contenuto. Infine possiamo facilmente constatare che il contenuto del MAR è ininfluente sulle altre micro operazioni, in quanto non va a modificare istruzioni che non utilizzano la memoria (INC, DEC, NEG e NOT). Quindi si potrebbe sfruttare la terza microistruzione del fetch per caricare il contenuto di MDR nel MAR ("bypassando" il passaggio per il registro IR) per averlo già pronto nelle operazioni successive.
Riscriviamo quindi la fase di fecth in questo modo:

MAR ← PC, ALUOUT ← 0 + PC + 1
MDR ← M[MAR], PC ← ALUOUT
IR ← MDR, MAR ← MDR

Possiamo ora riscrivere in forma sintetica le microistruzioni delle varie istruzioni.

SET
ACC ← MAR

INC
ALUOUT ← 0 + ACC + 1 (FLAG)
ACC ← ALUOUT

DEC
ALUA ← ACC, ALUOUT ← 0 and ACC
ALUOUT ← ALUA + ALUOUT' (FLAG)

ACC ← ALUOUT

NEG
ALUOUT ← 0 + ACC' + 1 (FLAG)
ACC ← ALUOUT

NOT
ALUOUT ← 0 + ACC' (FLAG)
ACC ← ALUOUT

LD
MDR ← M[MAR]
ACC ← MDR

ST
MDR ← ACC
M[MAR] ← MDR

LDI
MDR ← M[MAR]
MAR ← MDR
MDR ← M[MAR]
ACC ← MDR

STI
MDR ← M[MAR]
MAR ← MDR
MDR ← ACC
M[MAR] ← MDR

AND
MDR ← M[MAR], ALUA ← ACC
ALUOUT ← ALUA and MDR (FLAG)
ACC ← ALUOUT

OR
MDR ← M[MAR], ALUA ← ACC
ALUOUT ← ALUA or MDR (FLAG)
ACC ← ALUOUT

ADD
MDR ← M[MAR], ALUA ← ACC
ALUOUT ← ALUA + MDR (FLAG)
ACC ← ALUOUT

SUB
MDR ← M[MAR], ALUA ← ACC
ALUOUT ← ALUA + MDR' + 1 (FLAG)
ACC ← ALUOUT

CMP
MDR ← M[MAR], ALUA ← ACC
ALUOUT ← ALUA + MDR' + 1 (FLAG)

4.3.8 Istruzioni di salto

Le ultime, ma non meno importanti, istruzioni da introdurre sono quelle relative ai salti condizionati e non condizionati, che consentono di modificare il flusso di esecuzione di un programma, elemento fondamentale per implementare funzioni logiche complesse. In particolare le istruzioni di salto condizionato sono quelle che eseguono il cambiamento del flusso di esecuzione di un programma solo in una determinata situazione (possiamo pensare per esempio ai costrutti presenti nei vari linguaggi di programmazione come if then, else, for...).
Le istruzioni di salto si basano sui risultati delle operazioni aritmetiche precedenti che vengono memorizzati nel registro FLAG e vengono inviati all'Unità di controllo che li utilizza per realizzare le funzioni logiche opportune.
L'istruzione di salto incondizionato è la seguente:

JMP
PC ← IR(add)

Come possiamo osservare l'implementazione è davvero semplice: si scrive il contenuto di IR in PC così che al prossimo ciclo di clock il PC è già stato modificato opportunamente e l'esecuzione quindi prosegue secondo l'istruzione puntata da PC.
Le istruzioni di salto condizionato sono molto simili: l'implementazione è identica, aggiungono soltanto un controllo, eseguito dall'unità di controllo, che verifica la condizione richiesta e che quindi poi permette il salto.
Le istruzioni di salto condizionato sono le seguenti:

JZ	Salta se il flag Z vale 1
JNZ	Salta se il flag Z vale 0
JN	Salta se il flag N vale 1
JNN	Salta se il flag N vale 0
JC	Salta se il flag C vale 1
JNC	Salta se il flag C vale 0
JO	Salta se il flag O vale 1
JE	Salta se è uguale
JNE	Salta se non è uguale
JGS	Salta se è maggiore (signed)
JGES	Salta se è maggiore o uguale (signed)
JLS	Salta se è minore (signed)
JLES	Salta se è minore o uguale (signed)
JGU	Salta se è maggiore (unsigned)
JGEU	Salta se è maggiore o uguale (unsigned)
JLU	Salta se è minore (unsigned)
JLEU	Salta se è minore o uguale (unsigned)

4.3.9 Codice operativo e tempi di clock

Riportiamo nella tabella sottostante l'elenco finale delle istruzioni viste, con riportati i cicli di clock che richiede la loro esecuzione, considerando anche nel secondo caso l'operazione di fetch (che è comune a tutte le istruzioni).

Gli indirizzi con i quali le istruzioni sono registrate nella memoria verranno riportati più avanti.

Dec	Mnemonico	Clock	Con Fetch
0	SET	1	4
1	INC	2	5
2	DEC	3	6
3	NEG	2	5
4	NOT	2	5
5	LD	2	5
6	ST	2	5
7	LDI	4	7
8	STI	4	7
9	AND	3	6
10	OR	3	6
11	ADD	3	6
12	SUB	3	6
13	JMP	1	4
14	JZ	1	4
15	JNZ	1	4
16	JN	1	4
17	JNN	1	4
18	JC	1	4
19	JNC	1	4
20	JO	1	4
21	JE	1	4
22	JNE	1	4
23	JGS	1	4
24	JGES	1	4
25	JLS	1	4
26	JLES	1	4

4.4 Control Unit

La Control Unit, ovvero l'Unità di Controllo del processore, gestisce il funzionamento dei vari componenti dell'Unità di Elaborazione mediante l'invio di segnali attraverso il Control Bus che vengono generati in base all'input fornito mediante il bus dei flag.
Prendiamo ora in considerazione due possibili implementazioni della Control Unit, l'implementazione cablata e l'implementazione micro programmata: nello specifico per il processore ADE32 si utilizza la seconda soluzione proposta, che quindi verrà analizzata in maniera più approfondita.
Vale la pena notare che tale scelta influenza in maniera profonda la natura stessa del processore in quanto, senza voler entrare nei dettagli, contribuisce alla sua entrata nella famiglia delle macchine RISC (*Reduced Instruction Set Computer*) o CISC (*Complex Instruction Set Computer*) che si caratterizzano principalmente per le scelte eseguite nei confronti dell'implementazione dell'Unità di Controllo (e di conseguenza sul parco istruzioni ecc.).

4.4.1 Implementazione cablata

L'implementazione cablata dell'unità di controllo, definita anche *hardwired*, prevede la realizzazione di appositi circuiti di controllo che interpretano il codice operativo (opcode) e inviano appositi segnali al data path. E' quindi tipicamente un'implementazione caratterizzata da una elevata velocità essendo realizzata interamente in hardware.
Vediamo ora brevemente come potrebbe essere realizzata all'interno di ADE32.
Il campo opcode di IR è direttamente connesso ad un decoder nell'unità di controllo che permette ad ogni istruzione di generare un segnale corrispondente che viene poi inviato al data path; l'insieme dei segnali che vengono generati dalle istruzioni non vengono riportati in questo paragrafo ma in quello successivo.
Per sintetizzare i segnali di controllo è necessario mettere in AND l'indicatore dell'istante in cui il segnale si deve attivare con il segnale relativo all'istruzione corrente (una soluzione già adottata nella sintesi dell'ALU a 1 bit).
Questa implementazione viene volutamente analizzata in modo superficiale in quanto non viene adottata per la realizzazione dell'Unità di Controllo di ADE32, ma si riporta solamente per completezza.

4.4.2 Implementazione micro programmata

L'implementazione micro programmata dell'unità di controllo prevede di memorizzare in un'apposita memoria, la *Memoria del Microcodice* (tipicamente non riscrivibile, come la ROM, la EEPROM e simili), per ogni microistruzioni l'insieme di tutti i segnali di controllo con il loro relativo valore.
Quindi nel momento in cui viene letta dalla memoria RAM una istruzione, viene inviato il suo opcode all'unità di controllo dai cui quest'ultima ricava l'elenco delle microistruzioni e i relativi segnali da attivare per eseguire l'istruzione stessa.
Al termine di ogni ciclo di clock è necessario ovviamente modificare l'indirizzo analizzato per prelevare i segnali corretti: questo può essere ottenuto mediante un *micro* Program Counter (µPC), un registro del tutto analogo al registro PC visto all'interno del data path. Tale registro può essere incrementato per passare alla microistruzione successiva, può essere azzerato per tornare alla fase di fetch (comune a tutte le istruzioni e che quindi compare all'indirizzo iniziale della memoria) e può essere caricato con il contenuto dell'opcode per diventare un indirizzo nella memoria delle microistruzioni.
Nella figura 4.15 si riporta solo uno schema della soluzione adottata (in quanto su Logisim è stato adottato un circuito più complesso, che vedremo in seguito). I segnali di micro controllo (MC1, MC0) sono utilizzati come bit di controllo per il multiplexer a tre uscite per selezionare l'uscita: essendo presenti solo tre uscite a fronte di 4 (2^2) teoricamente controllabili, la quarta configurazione dei segnali di µcontrollo risulta essere non valida.
Ricordando il funzionamento di un multiplexer possiamo quindi ridirigere la seguente tabella:

MC1	MC0	Uscita selezionata
0	0	µsalto al fetch
0	1	µsalto all'opcode
1	0	Incremento
1	1	(non valido)

Risulta quindi necessario aggiungere alle microistruzioni i due segnali MC1 ed MC0 (rispettivamente il 20 e il 19).

Attendiamo di completare lo sviluppo dell'implementazione micro programmata per riportare una tabella riassuntiva delle istruzioni con le relative microistruzioni e segnali di controllo.

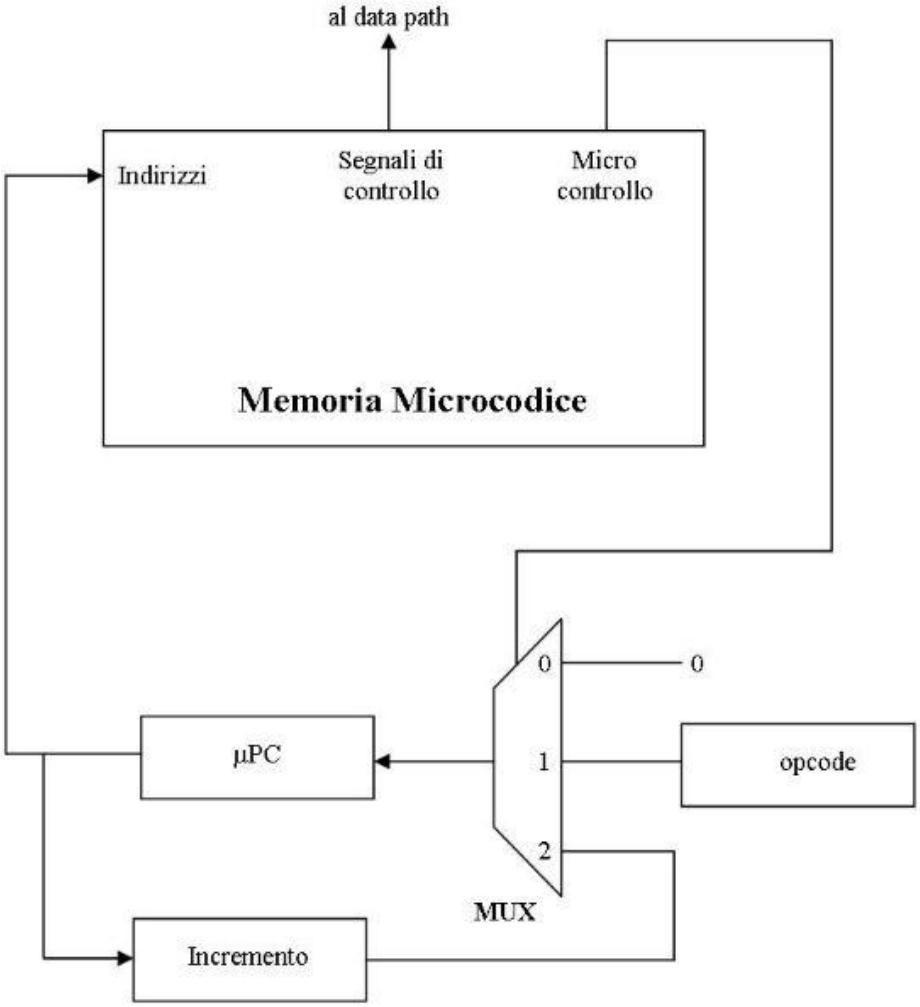

Figura 4.15: Schema Unità di Controllo (senza salti)

L'incremento del µPC viene effettuato da un circuito logico molto simile a un Full Adder, come si può vedere in Figura 4.16. Una costante di valore unitario viene aggiunta al primo bit dell'input da 12 bit, il riporto, generato dalle porte AND, viene trasmesso in cascata a tutte le porte XOR. Da notare che l'ultimo riporto, essendo inutile, è stato omesso risparmiando in tale modo l'implementazione di un ulteriore porta logica.

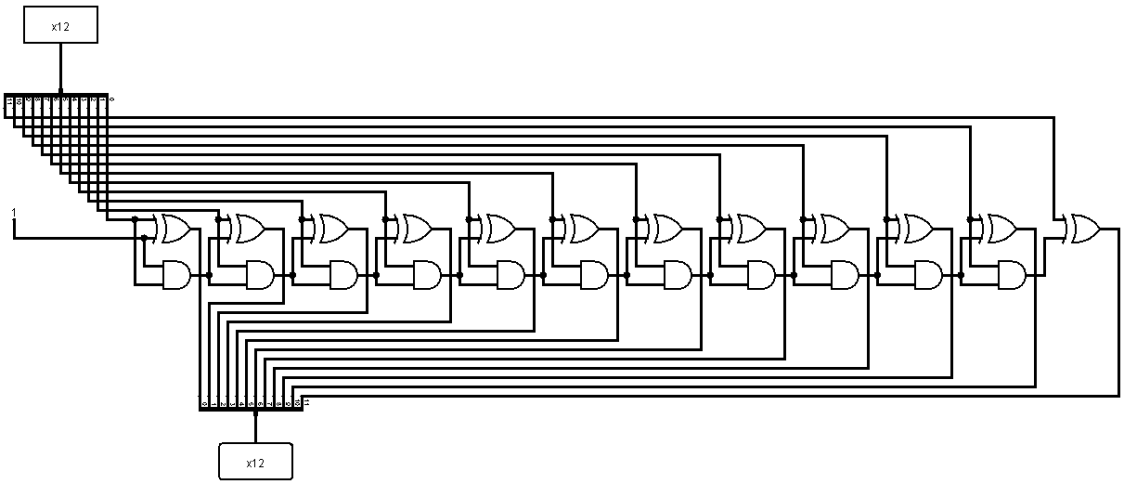

Figura 4.16: Schema logico dell'Incrementatore

Introduciamo ora nella Control Unit il meccanismo di controllo dei salti. L'idea di fondo è che se alcuni eventi devono essere condizionati dal valore del registro FLAG, bisogna allora pensare a un meccanismo in grado di ricevere i segnali in ingresso e decidere quale abilitare per l'uscita. Solitamente, per raggiungere questo scopo, si lavora con dei multiplexer connessi in cascata tra di loro.
In figura 4.17 vediamo chiaramente la soluzione adottata. Sono presenti tre multiplexer, chiamati Mux A, Mux B e Mux C, connessi tra di loro in cascata. Al Mux A vengono passati i segnali di controllo provenienti dal registro FLAG e i loro negati (O, N, N', C, C', Z, Z') per le prime istruzioni di salto che abbiamo visto, ovvero JO, JN, JNN, JC, JNC, JZ, JNZ; con i segnali provenienti dal registro FLAG si formano anche gli altri ingressi del multiplexer corrispondenti alle funzioni logiche delle istruzioni di salto:

- JGU = N' and Z'
- JGS = Z and (N xnor O)
- JGES = N xnor O
- JLEU = N or Z
- JLS = N xor O
- JLES = Z or (N xor O)

I segnali JE, JNE, JLU, JGEU corrispondono a segnali già presenti, rispettivamente a JZ, JNZ, JN, JNN. Infine sono presenti 3 ingressi, due posti a zero e uno posto a 1 per le microistruzioni che prevedono l'incremento .
I 4 bit provenienti dalla memoria ROM abilitano sono una di queste uscite, è quindi proprio questo meccanismo che consente di verificare eventuali salti dell'esecuzione del programma. L'output di Mux A funge da controllo per il Mux B, i cui due ingressi sono formati dallo zero per il fetch, e dall'uscita dell'incrementatore, che ha una struttura e funzione del tutto identica a quanto visto prima. Infine l'ultimo multiplexer, Mux C, è controllato dall'AND dei segnali provenienti dalla ROM e ha in ingresso il risultato di Mux B e l'Opcode. L'unica microistruzione in grado di abilitare il secondo ingresso del Mux C è la terza microistruzione del fetch, che avendo tutti i bit di controllo posti a 1, riesce ad abilitare l'AND posto sotto il Mux A.
Riportiamo nella tabella seguente tutte le possibili configurazione dei quattro segnali di controllo con le loro rispettive funzioni:

MC3	MC2	MC1	MC0	Funzione
0	0	0	0	Fetch
0	0	0	1	Incremento (prossima µistruzione)
0	0	1	0	Se Z incremento, altrimenti fetch
0	0	1	1	Se Z' incremento, altrimenti fetch
0	1	0	0	Se N incremento, altrimenti fetch
0	1	0	1	Se N' incremento, altrimenti fetch
0	1	1	0	Se C incremento, altrimenti fetch
0	1	1	1	Se C' incremento, altrimenti fetch
1	0	0	0	Se O incremento, altrimenti fetch
1	0	0	1	Se > incremento, altrimenti fetch (unsigned)
1	0	1	0	Se > incremento, altrimenti fetch (signed)
1	0	1	1	Se > o = incremento, altrimenti fetch (signed)
1	1	0	0	Se < o = incremento, altrimenti fetch (unsigned)
1	1	0	1	Se < incremento, altrimenti fetch (signed)
1	1	1	0	Se < o = incremento, altrimenti fetch (signed)
1	1	1	1	Salto all'opcode

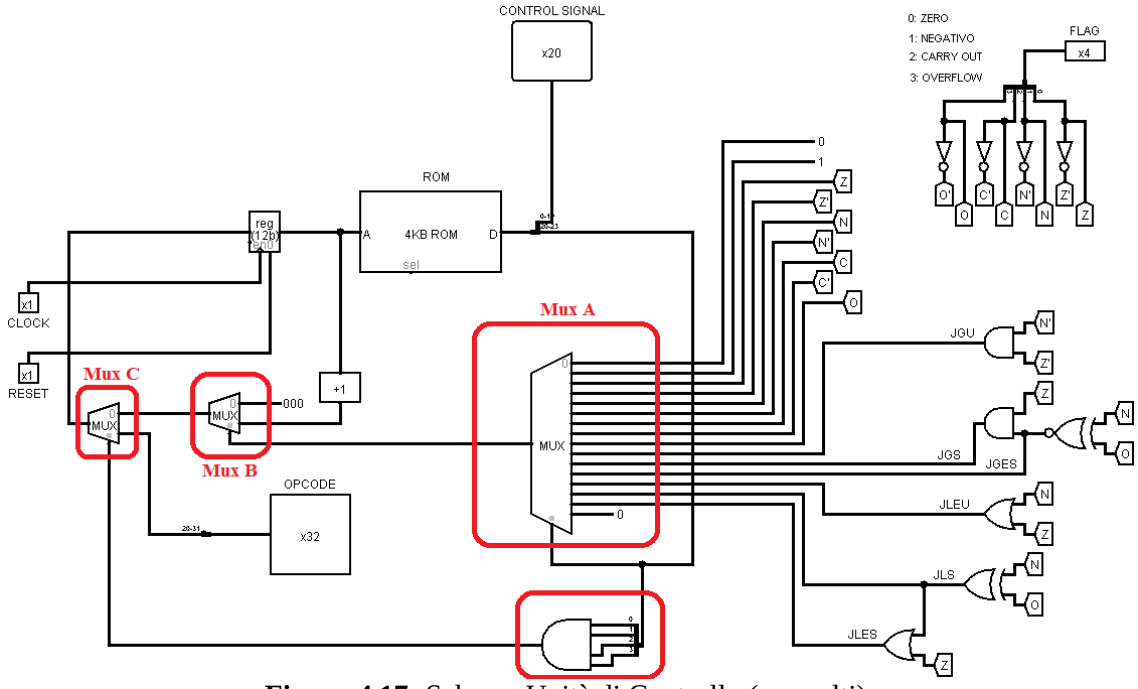

Figura 4.17: Schema Unità di Controllo (con salti)

4.4.3 Segnali di controllo

Avendo ora definito tutto il parco di istruzioni disponibili per l'architettura ADE32, è possibile andare a specificare i segnali di controllo specifici per ogni microistruzione.
Per ogni microistruzione si dovranno attivare i segnali che abbiamo visto nei capitoli precedenti, segnali che provengono dalla ROM e che sono selezionati dall'opcode di ogni istruzione.

SET

ACC ← MAR	ACC_{IE}, MAR_{OE}

INC

ALUOUT ← 0 + ACC + 1 (FLAG)	$ALUOUT_{IE}$, ACC_{OE}
	(S1,S0,ZA,Cin)=1001, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

DEC

ALUA ← ACC, ALUOUT ← 0 and ACC	$ALUA_{IE}$, ACC_{OE}, $ALUOUT_{IE}$,
	(S1,S0,ZA,Cin)=0000
ALUOUT ← ALUA + ALUOUT' (FLAG)	$ALUOUT_{IE}$, $ALUOUT_{OE}$,
	(S1,S0,ZA,Cin)=1110, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

NEG

ALUOUT ← 0 + ACC' + 1 (FLAG)	$ALUOUT_{IE}$, ACC_{OE},
	(S1,S0,ZA,Cin)=1101, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

NOT

ALUOUT ← 0 + ACC' (FLAG)	$ALUOUT_{IE}$, ACC_{OE},
	(S1,S0,ZA,Cin)=1100, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

LD

MDR ← M[MAR]	MDR_{BUSIE}
ACC ← MDR	ACC_{IE}, MDR_{OE}

ST

MDR ← ACC	MDR_{IE}, ACC_{OE}
M[MAR] ← MDR	MDR_{BUSOE}

LDI

MDR ← M[MAR]	MDR_{BUSIE}
MAR ← MDR	MAR_{IE}, MDR_{OE}
MDR ← M[MAR]	MDR_{BUSIE}
ACC ← MDR	ACC_{IE}, MDR_{OE}

STI

MDR ← M[MAR]	MDR_{BUSIE}
MAR ← MDR	MAR_{IE}, MDR_{OE}
MDR ← ACC	MDR_{IE}, ACC_{OE}
M[MAR] ← MDR	MDR_{BUSOE}

AND

MDR ← M[MAR], ALUA ← ACC	MDR_{BUSIE}, $ALUA_{IE}$, ACC_{OE}
ALUOUT ← ALUA and MDR (FLAG)	$ALUOUT_{IE}$, MDR_{OE}
	(S1,S0,ZA,Cin)=0010, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

OR

MDR ← M[MAR], ALUA ← ACC	MDR_{BUSIE}, $ALUA_{IE}$, ACC_{OE}
ALUOUT ← ALUA or MDR (FLAG)	$ALUOUT_{IE}$, MDR_{OE}
	(S1,S0,ZA,Cin)=0110, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

ADD

MDR ← M[MAR], ALUA ← ACC	MDR_{BUSIE}, $ALUA_{IE}$, ACC_{OE}
ALUOUT ← ALUA + MDR (FLAG)	$ALUOUT_{IE}$, MDR_{OE}
	(S1,S0,ZA,Cin)=1010, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

SUB

MDR ← M[MAR], ALUA ← ACC	MDR_{BUSIE}, $ALUA_{IE}$, ACC_{OE}
ALUOUT ← ALUA – MDR' + 1 (FLAG)	$ALUOUT_{IE}$, MDR_{OE}
	(S1,S0,ZA,Cin)=1111, $FLAG_{IE}$
ACC ← ALUOUT	ACC_{IE}, $ALUOUT_{OE}$

CMP

MDR ← M[MAR], ALUA ← ACC	MDR_{BUSIE}, $ALUA_{IE}$, ACC_{OE}
ALUOUT ← ALUA – MDR' + 1 (FLAG)	$ALUOUT_{IE}$, MDR_{OE}
	(S1,S0,ZA,Cin)=1111, $FLAG_{IE}$

JMP

PC ← IR(add)	PC_{IE}, IR_{OE}

Come sopra riportato, i segnali che la Control Unit genera in seguito all'arrivo di un'istruzione di salto condizionato sono identici ai segnali di JMP, con la sola differenza che il primo ciclo di clock è impiegato per la logica interna alla CU.

Possiamo ora riportare nella pagina seguente la tabella riassuntiva di tutti i comandi disponibili, con i relativi indirizzi con i quali sono memorizzati in memoria e i segnali di controllo sia in forma binaria che esadecimale (quest'ultima supportata da Logisim).

Operazione	Indirizzi	MC3	MC2	MC1	MC0	S1	S0	Cin	ZA	$FLAG_{IE}$	ALUO	ALUO	$ALUA_{IE}$	PC_{IE}	PC_{OE}	IR_{IE}	IR_{OE}	MAR_{IE}	MAR_{OE}	MDR_{BU}	MDR_{BU}	MDR_{IE}	MDR_{OE}	ACC_{IE}	ACC_{OE}	HEX
		23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
fetch	**000**	0	0	0	1	1	0	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1A4480
	001	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	102820
	002	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	F00284
SET	**010**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	000042
INC	**020**	0	0	0	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1A4001
	021	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
DEC	**030**	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	105001
	031	0	0	0	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1DE000
	032	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
NEG	**040**	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1EC001
	041	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
NOT	**050**	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1C4001
	051	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
LD	**060**	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	100020
	061	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	000006
ST	**070**	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	100009
	071	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	000010
LDI	**080**	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	100020
	081	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	100084
	082	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	100020
	083	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	000042
STI	**090**	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	100020
	091	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	100084
	092	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	100009
	093	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	000010
AND	**0A0**	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	101021
	0A1	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	11C004
	0A2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
OR	**0B0**	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	101021
	0B1	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	154004
	0B2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
ADD	**0C0**	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	101021
	0C1	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	194004
	0C2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
SUB	**0D0**	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	101021
	0D1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1FC004
	0D2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	002002
CMP	**0E0**	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	101021
	0E1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0FC004
JMP	**0F0**	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JZ (JE)	**100**	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	200000
	101	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
NZ (JNE)	**110**	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	300000
	111	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JN (JLU)	**120**	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	400000
	121	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
NN(JGEU)	**130**	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	500000
	131	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JC	**140**	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	600000
	141	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JNC	**150**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	700000
	151	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JO	**160**	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	800000
	161	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JGU	**170**	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	900000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JGS	**180**	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A00000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JGES	**190**	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	B00000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JLEU	**1A0**	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C00000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JLS	**1B0**	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D00000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900
JLES	**1C0**	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	E00000
		0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	000900

Il difetto evidente di questa implementazione deriva dal fatto che è necessario inserire un ciclo di clock in ogni istruzione di salto condizionato che serve solo a verificare lo stato dei segnali ed eseguire un μsalto al fetch in caso di condizione falsa.
Questo difetto è però controbilanciato dalla forte semplicità della Contro Unit, motivo per il quale si è scelto questo compromesso.

Capitolo 5 - Assembly e Assembler

In questo capitolo si presenta il linguaggio *Assembly* del processore ADE32 costruito sulla base delle istruzioni presentate fino a questo momento; tale linguaggio permette di programmare il processore attraverso un linguaggio mnemonico, sicuramente maggiormente maneggevole rispetto al linguaggio macchina.
In un secondo momento si introduce il concetto di Assembler, il software che si occupa della traduzione dal linguaggio mnemonico *Assembler* al linguaggio macchina: se ne presentano due diversi sviluppi, uno in pure linguaggio C e uno realizzato mediante i *tools* Bison e Flex.

5.1 Caratteristiche dell'Assembly di ADE32

L'insieme delle istruzioni che abbiamo visto nei capitoli precedenti costituisce il linguaggio assembly dell'Architettura ADE32.
L'assembly è un linguaggio di programmazione particolarmente vicino al linguaggio macchina: in effetti abbiamo visto che la corrispondenza tra le istruzioni mnemoniche e le istruzioni del processore è di 1:1; tale linguaggio permette di non programmare il processore tramite comandi espressi in formato binario (o tutt'al più in formato esadecimale) scomodi e di difficile memorizzazione, ma appunto con delle sigle mnemoniche, dotate di significato, molto più semplici da memorizzare e utilizzare.
Come è facile immaginare ogni costruttore di processori (Intel, AMD...) definisce un proprio linguaggio Assembly che quindi è supportato da una documentazione esaustiva, in grado di mettere gli sviluppatori nelle condizioni di programmare su una determinata architettura.
Nel nostro caso dobbiamo quindi definire il linguaggio assembly di ADE32: per fare ciò dobbiamo stabilire le regole di base.
Su ogni riga di codice bisogna specificare il codice operativo simbolico e gli eventuali operandi che possono essere numerici e simbolici: non è necessario calcolare l'indirizzo delle variabili in memoria dato che può farlo un programma per noi che, come vedremo, prende il nome di Assembler (o Assemblatore). Inoltre all'inizio di ogni riga può essere presente una etichetta seguita da ":" (due punti), che rappresenta simbolicamente un indirizzo di memoria.
Ogni istruzione occupa solo una riga di codice; le istruzioni vengono eseguite in maniera sequenziale, ovvero nell'ordine in cui sono scritte (a meno di costrutti che modificano il flusso delle istruzioni).
Programmare in Assembly significa quindi programmare a un basso livello di astrazione logica. Esistono per tale motivo i linguaggi di programmazione ad alto livello (come il C, C++, Java, Python...) generalmente più adatti per la programmazione di software di grandi dimensioni e/o maggiormente complessi.
Poiché un programma scritto in assembly non può essere eseguito direttamente dal processore è necessario disporre di uno strumento che si occupa della traduzione dal formato mnemonico a linguaggio macchina corrispondente, ovvero l'assembler.
Il termine deriva dal fatto che le istruzioni tradotte in binario vengono montate, assemblate appunto, una accanto all'altra.
Prendiamo ora in considerazione due sviluppi di assembler, uno in linguaggio C e l'altro realizzato col supporto di *Bison* e *Flex*.

5.2 Assemblatore in C

Una prima soluzione semplice ma poco flessibile è quella di sviluppare il compilatore in C senza l'utilizzo di nessun programma intermedio.
Il C è un linguaggio particolarmente adatto a questo scopo, essendo tra tutti i linguaggio di programmazione ad alto livello quello di livello logico più basso.
Il programma creato prende in input un file di testo (f) in cui è scritto il codice assembly, lo elabora e produce in output due file, il primo (g) contente la traduzione in linguaggio macchina (che è espresso in forma esadecimale per essere interpretato da Logisim) e il secondo (log) contenente un report di log, per il controllo di eventuali errori. In particolare il file di testo dato in ingresso viene scansionato due volte tramite la funzione *fscanf*: durante la prima volta vengono cercate le etichette e se trovate vengono memorizzate all'interno di un array; nella seconda scansione si traduce ogni stringa che si incontra cercando di definirne la tipologia (numero, etichetta, istruzione, indirizzo).

Questa soluzione, proprio per il suo metodo di scansione stringa per stringa, risulta essere di limitata flessibilità; ad esempio i due punti dell'etichetta necessitano di essere seguiti sempre da uno spazio, altrimenti il programma non riesce a distinguere l'etichetta dall'istruzione che segue; o, ancora, l'etichetta deve essere sempre presente all'inizio della riga di codice a cui è associata.
Inoltre risulta difficile applicare modifiche al programma in seguito a un'eventuale espansione del parco delle istruzioni del processore o altre piccole modifiche alle regole di costruzione sintattica del codice.
Si avverte quindi la necessità di cambiare il metodo di scansione, di avere un'analisi sintattica e lessicale più accurata e dinamica, maggiormente libera dai vincoli rigidi che abbiamo appena visto.
Riportiamo di seguito il nucleo principale del codice del programma, ovvero la funzione che si occupa della traduzione dell'input in linguaggio macchina (in formato esadecimale):

```
while(fscanf(f, "%s", etichetta) > 0){//leggo la prima parola
della riga
            if(strpbrk(etichetta, ";"))//se incontro un ; è un
commento che non scansiono
                  fscanf(f, "%[^\n]s", vuoto);
            else if(strpbrk(etichetta, ":")){//se contiene : è
un'etichetta
                  strcpy(vetic[i], etichetta);//memorizzo
etichetta nel vettore
                  i++;
                  fscanf(f, "%[^\n]s", vuoto);//vado a fine
riga, le altre parole non interessano
            }
            else {//se non contiene : non c'è etichetta
                  strcpy(vetic[i], "()");//memorizzo un valore
nullo nel vettore
                  i++;
                  fscanf(f, "%[^\n]s", vuoto);//vado a fine
riga
            }
      }//fine while
      rewind(f);//riparto a leggere il file dall'inizio
      while(fscanf(f, "%s", parola) > 0){
            ok_et=0; ok_num=0; ok_com=0; ok_ist=0;
            if(strpbrk(parola, ";")){//se incontro un ; è un
commento che non scansiono
                  fscanf(f, "%[^\n]s", vuoto);
                  ok_com=1;
            }
            else if(strpbrk(parola, ":" )){//se trovo
un'etichetta non faccio nulla
                  fprintf_s(log, "Trovata etichetta: %s\n",
parola);
                  ok_et=1;
            }
            else if (parola[0]>='0' && parola[0]<='9'){//se
trovo un numero lo converto in esadecimale
                  fprintf_s(log, "Trovato numero: %s\n",
parola);
                  str_to_hex(parola, g);
                  ok_num=1;
            }
            else if(ok_com==0 && ok_et==0 &&
ok_num==0){//controllo istruzione
```

```
                  for(j=0;j<28;j++){
                        if(strcmp(istr[j].nome, parola)==0){
                              fprintf(g, "%s", istr[j].code);
                              fprintf(log, "Trovata istruzione:
%s\n", parola);
                              ok_ist=1;
                        }
                  }
            }
             if( ok_et==0 && ok_com==0 && ok_num==0 &&
ok_ist==0){//se arrivo qui la parola è sicuramente un simbolo
                  strcpy(simbolo, parola);
                  strcat(simbolo, ":");
                  for(k=0;k<i;k++)
                        if (strcmp(simbolo, vetic[k]) == 0){
                              fprintf_s(log, "Trovato simbolo:
%s\n", parola);
                              int_to_hex(k, g);
                              sim++;}
                        if(sim==0){
                              fprintf_s(log, "***Elemento [%s]
non trovato!\n", parola);
                              err++;}
                        sim=0;
            }
      }
      fclose(log);
      if(err==0)printf("\nProcesso terminato
correttamente.\nVisualizzare file di log per debug.\n");
      else printf("\nProcesso terminato con %d problemi.
Errori riportati su file di log.\n", err);
```

Da notare che la presenza di eventuali errori e il loro numero sono segnalati in chiusura del programma. Le funzioni esterne chiamate dal programma, *str_to_hex* e *int_to_hex*, servono, come si può intuire, a convertire la stringa o l'intero in numeri esadecimali.
Si riporta inoltre, per completezza, la funzione *main()* del programma;

```
int main(int argc, char **argv){
      FILE *f, *g;

      if(argc==1 || argc>4){
            printf("Uso: ADE32_asm -l/-b file_input file_output");
      }

      else if(strcmp(argv[1], "-l")==0){
            if((f=fopen(argv[2], "r")) == NULL){
                  fprintf_s(stderr, "Errore apertura file 1\n");
                  return 1;}
            if((g=fopen(argv[3], "w")) == NULL){
                  fprintf_s(stderr, "Errore apertura file 2\n");
                  return 2;}
            logisim(f, g, argv[3]);
            fclose(f);
            fclose(g);
      }

      else if(strcmp(argv[1], "-b")==0){
            if((f=fopen(argv[2], "r")) == NULL){
                  fprintf_s(stderr, "Errore apertura file 1\n");
                  return 1;}
```

```
            if((g=fopen(argv[3], "w")) == NULL){
                  fprintf_s(stderr, "Errore apertura file 2\n");
                  return 2;}
            binario(f, g);
            fclose(f);
            fclose(g);
      }
      else printf("Uso: ADE32_asm -l/-b file_input file_output");
}
```

5.3 Assemblatore con Flex e Bison

Solitamente il parser, ovvero il programma che si occupa dell'analisi sintattica, non viene scritto a mano, a causa dell'eccessiva complessità, ma viene generato da dei generatori di parser, come ad esempio Bison, che generalmente viene utilizzato assieme a Flex.

Flex e Bison sono appunto i due tools utilizzati per la creazione dell'Assembler come seconda soluzione. Non è obiettivo di questo scritto analizzare nel dettaglio le dinamiche e il funzionamento di questi due programmi: per una trattazione accurata e completa si rimanda alla documentazione ufficiale.

Flex è un analizzatore lessicale (o scanner) che raggruppa i caratteri letti dal file sorgente in unità lessicali, definiti *token* (e per tale motivo viene anche definito come *tokenizer*); nella programmazione di tale strumento non serve definire come deve essere eseguita la scansione, ma cosa si vuole scansionare, definendo i token validi; la maggior parte dei dettagli riguardanti la scansione è gestita in maniera automatica.

Flex, come abbiamo detto, è pensato principalmente per lavorare con Bison, il suo output è il file *Flex.yy.c*, un programma in C privo di main() che contiene la routine di scanning *yyFlex()* assieme ad altre routine ausiliari e macro, che viene dato in ingresso a Bison.

Bison è un analizzatore sintattico (o parser) che ha il compito di raggruppare i token, passati da Flex, in frasi grammaticali e produrre in output il programma in C che funge da compilatore, l'Assembler.

Rispetto alla soluzione implementata in C questa è sicuramente migliore dal punto di vista della flessibilità e robustezza. Inoltre a fronte di input anche molto complessi, il programma si mantiene su livelli di complessità discreti.

Capitolo 6 - Esempi di programmi per l’architettura

In questo capitolo vengono presentati due semplici programmi scritti per il processore ADE32: il primo è una somma fra due numeri interi a 32 bit, mentre il secondo è un incrementatore che parte dal valore 0 per arrivare al valore 10.
Hanno il compito di mostrare il funzionamento del processore, in particolare mettono in evidenza il funzionamento delle operazioni su un’architettura ad accumulatore, e l’utilizzo di istruzioni di salto condizionato.

6.1 Somma di due numeri

Questo programma esegue la somma fra due valori numerici a 32 bit A e B inseriti in memoria durante la fase di caricamento del programma.
Il risultato viene memorizzato nella variabile C.

Codice assembly

```
        LD    A
        ADD   B
        ST    C

A:      00001234h
B:      00005678h
C:      0
```

In maniera simile agli altri linguaggi assembly i valori esadecimali vengono terminati con il carattere *h*.
Il codice assembly viene dato in ingresso all'assembler e il risultato è il seguente (in formato esadecimale, caricato in memoria a partire dall'indirizzo 0):

Codice esadecimale

```
v2.0 raw
6000003 c000004 7000005 1234 5678 0
```

Questo output, essendo pensato esplicitamente per Logisim, conserva in apertura una stringa necessaria per la sua corretta comprensione da parte del programma.

Codice binario

```
00000110000000000000000000000011
00001100000000000000000000000100
00000111000000000000000000000101
00000000000000000001001000110100
00000000000000000101011001111000
00000000000000000000000000000000
```

Confrontando il punto di partenza, ovvero il codice assembly, con il punto di arrivo, il codice binario, ben si capisce l'utilità di poter scrivere in un linguaggio di tipo mnemonico e affidare a un programma la traduzione delle istruzioni.

6.2 Incremento da 0 a 10

Questo programma esegue un incremento di una unità partendo dal valore 0 e arrivando a 10. E' un programma interessante in quanto vengono utilizzate istruzioni monadiche, di salto condizionato e di comparazione.

Codice assembly

```
SET    0
loop:
   ST      80000h
   INC
   CMP   max
   JLEU  loop

fine:
   JMP    fine

MAX: Ah
```

Altri elementi degni di nota in questo programma sono la stampa del valore incrementale nel display esterno al processore (raggiungibile mediante l'indirizzo 80000h) e il ciclo infinito al termine del programma per evitare che il processore prosegua col fetch di una nuova istruzione, che serve quindi per fermare il processore nella sua esecuzione.

Codice esadecimale

```
v2.0 raw
1000000 7080000 2000000 e000000 1a000001 f000005
```

Come nel caso precedente c'è la stringa iniziale che serve al programma Logisim per caricare correttamente il codice in memoria.

Codice binario

```
00000001000000000000000000000000
00000111000010000000000000000000
00000010000000000000000000000000
00001110000000000000000000000000
00011010000000000000000000000001
00001111000000000000000000000101
```

Capitolo 7 - Sintesi del progetto

In questo capitolo si attua l'implementazione del progetto del processore ADE32 su scheda programmabile dal punto di vista hardware.
A tal fine si presenta il linguaggio VHDL utilizzato per la progettazione dell'hardware, fornendone una breve descrizione delle caratteristiche e del suo impiego; inoltre illustra la strumentazione utilizzata, ovvero la scheda Xess ospitante l'FPGA *Spartan3* prodotto dalla *Xilinx*.
Si mostra inoltre la costruzione dei vari componenti con il VHDL e le loro connessioni per formare ADE32, realizzate sulla base del progetto ottenuto con Logisim.

7.1 Introduzione al VHDL

Il VHDL (ovvero *VHSIC Hardware Description Language*, VHSIC a sua volta significa *Very High Speed Integrated Circuits*) è, assieme al Verilog, il linguaggio maggiormente utilizzato per la progettazione di sistemi elettronici digitali.
Viene utilizzato per descrivere il funzionamento e la struttura dei circuiti digitali, è un linguaggio per la descrizione dell'*hardware*. Permette per esempio di modellare l'interazione tra i blocchi funzionali di un sistema, tramite la definizione dei segnali di ingresso e di uscita di ogni componente e il suo funzionamento logico.
Per alcuni aspetti è molto simile a un linguaggio di programmazione, prevede infatti l'utilizzo dei tipici costrutti di programmazione per modellare il funzionamento di un circuito digitale (come if then, else, when, case...), ma per altri è completamente differente: una specifica VHDL non è un programma eseguibile, infatti descrive la costituzione di un componente, anche se tuttavia può essere simulato mediante degli specifici strumenti (come ISim, di cui è stato scritto nel secondo capitolo).
Due sono le principali fasi di progettazione di un dispositivo in VHDL: si descrive prima la sua configurazione esterna, ovvero l'interfaccia, inserendo queste informazioni all'interno del costrutto *entity*: essa è costituita dai segnali di ingresso, di uscita, il tempo di ritardo e le caratteristiche dei bus connessi; in seguito si descrive il blocco dal punto di vista funzionale, ovvero come il dispositivo funziona, all'interno del costrutto *architecture*.
La sintassi generale di una *entity declaration* è la seguente:

```
entity entity_name is
[generic( generic_list );]
[port( port_list );]
end entity_name;
```

In `entity_name` viene inserito il nome del blocco mentre in `port_list` vengono inseriti tutti i segnali di ingresso e di uscita del componente seguendo tale forma:

```
port_name[,port_name,...]: {in|out|inout} port_type;
```

Come si può notare per ogni segnale dichiarato bisogna specificare la sua tipologia che può essere unidirezionale (`in` e `out`) e bidirezionale (`inout`). In `generic_list`, a differenza delle porte, abbiamo valori senza alcuna direzione, che possono essere anche tipi complessi, come in virgola mobile.
La sintassi generale di una *architecture declaration* è la seguente:

```
architecture architecture_name of entity_name is
[declarations]
begin
[implementation]
end architecture_name;
```

Appare immediato che un'architecture è associata a una sola entity, mentre invece una entity può essere associata a più architecture.
Nel blocco opzionale `declarations` vengono riportate tutte le dichiarazione delle costanti, dei segnali, dei tipi e dei componenti che verranno utilizzati nel costrutto; nel blocco `implementation` viene inserita la descrizione riguardo alle funzionalità che il blocco deve avere.
Quest'ultima parte solitamente segue due differenti stili di descrizione, *behavioural* o *structural*. Con la prima si descrive il funzionamento del dispositivo mediante funzioni

logiche ed algoritmi, con la seconda invece si rappresenta la struttura interna del dispositivo formata da componenti logici di basso livello e si mostrano i loro collegamenti. Nel nostro caso utilizzeremo solamente la descrizione di tipo behavioural.
Un'altra nota, importante per comprendere al meglio i componenti realizzati, è data dalla principale caratteristica del VHDL, ovvero la *concorrenzialità*: a differenza degli altri linguaggi di programmazione, le diverse parti del codice VHDL vengono eseguite parallelamente poiché nel contesto reale i segnali effettivamente entrano e vengono elaborati in modo simultaneo dai componenti logici, a meno che il flusso dell'esecuzione del codice non venga forzato da costrutti di programmazione. Si avranno quindi costrutti di tipo parallelo e costrutti di tipo sequenziale.
Si riportano inoltre i tipi di dato messi a disposizione dal linguaggio:

1. **Bit**: è il tipo di dato più semplice, rappresenta un valore binario che può assumere solo i valori logici di 0 e 1 (nel codice indicati fra gli apici ('0', '1').
2. **Integer**: rappresenta valori interi a 32 bit, per default considerati senza segno; nella nostra implementazione non vengono utilizzati.
3. **IEEE**: è una libreria standard che introduce diverse tipologie di segnali molto utili in fase di sintesi del progetto. I valori logici su cui si basa sono:
   - **'0'** Valore logico 0.
   - **'1'** Valore logico 1.
   - **'Z'** Alta impedenza.
   - **'X'** Indeterminato. Può essere 0 o 1.
   - **'U'** Indefinito. Il valore non è mai stato assegnato.
   - **'W'** Segnale debole. Non è possibile interpretarlo come 0 o 1.
   - **'L'** Segnale debole. Interpretabile come 0.
   - **'H'** Segnale debole. Interpretabile come 1.
   - **'-'** Don't care

Come ogni libreria nel VHDL la IEEE è suddivisa in *packages* ognuno dei quali definisce dei tipi di dato che possono essere utilizzati in fase di sintesi, in particolare riportiamo il package *std_logic_1164* che contiene i tipi *std_logic* e *std_logic_vector* (oltre ai corrispondenti *std_ulogic* e *std_ulogic_vector*), ampiamente utilizzati nell'implementazione di ADE32.
Per istanziare un bus si utilizza la seguente sintassi:

```
signal_name( index1 {to|downto} index2 )
```

Infine il VHDL supporta tutte le espressioni logiche, le variabili booleane sono rappresentate dai dati di tipo bit e std_logic.

Operatore logico	Operatore VHDL
AND, °	and
OR, +	or
NOT, ', !	not
XOR, ⊕	xor

Gli operatori di assegnamento sono "<=" e "=>" per i segnali e ":=" per le altre variabili. Infine i commenti sono introdotti con due linee (--).

Il progetto dell'architettura ADE32, fino ad ora confinato all'ambito di Logisim, viene implementato tramite il linguaggio VHDL su una scheda FPGA.

7.2 FPGA

Un *Field Programmable Gate Array*, spesso abbreviato in FPGA, è un circuito integrato digitale programmabile direttamente dall'utente via software. Come si può osservare in Figura 7.1, contiene al suo interno un alto numero di porte logiche disposte a matrice che vengono utilizzate per implementare funzioni logiche complesse, mentre sul perimetro sono presenti bei blocchi di I/O (piedini) che vengono utilizzati per inviare o ricevere segnali dalla scheda.

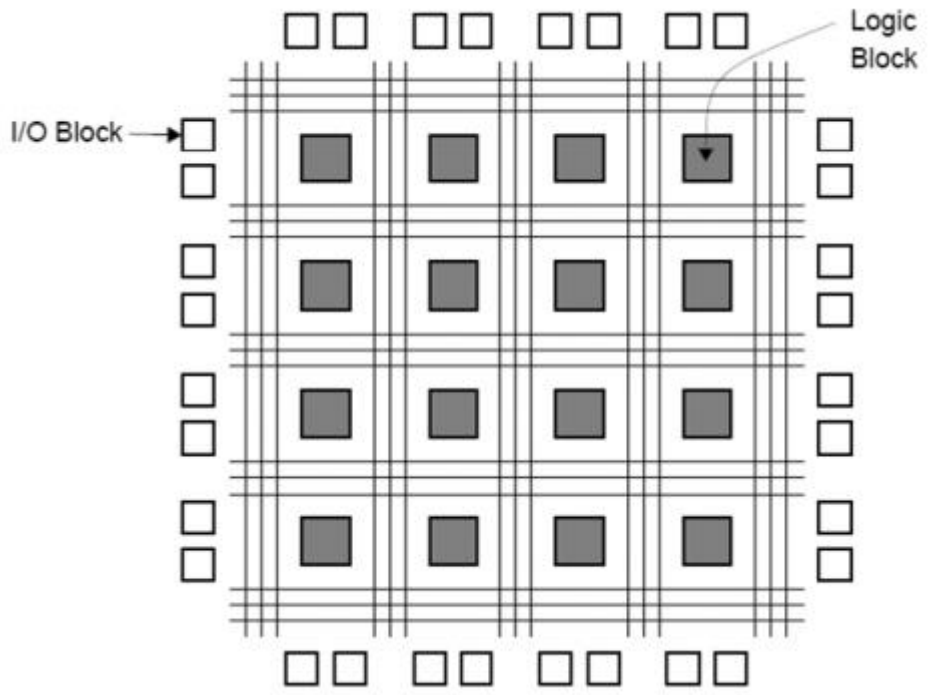

Figura 7.1: Schema di una generica FPGA

La programmazione avviene solitamente tramite il linguaggio VHDL o *Verilog*, ma esistono anche altri metodi che qui non vengono elencati.
Queste schede programmabili risultano quindi essere molto comode in fase di prototipazione in quanto permettono la cancellazione del loro contenuto e la loro riconfigurazione in maniera rapida ed efficiente.

L'FPGA generalmente viene inserita su di una scheda dotata di varie tipologie di dispositivi che si connettono direttamente o indirettamente al circuito digitale programmabile, espandendo in tal modo le possibilità di utilizzo e programmazione dell'utente.
Per l'implementazione di ADE32 la scheda utilizzata è la *XSA-3S1000* della Xess (Figura 7.2); al suo interno ospita l'FPGA *Spartan3* della Xilinx e altri dispositivi come un oscillatore da 100 MHz, un circuito CPLD, una memoria Flash, una memoria SDram da 32 MB, alcuni tasti e un semplice display led a 7 segmenti; la scheda comunica con l'esterno tramite diverse porte, come la porta VGA per uno schermo e una porta per la connessione di un mouse o una tastiera; infine la scheda è espandibile attraverso diversi moduli che vengono venduti separatamente e possono essere connessi in maniera diretta alla scheda.
Nel nostro caso la scheda è provvista di un modulo aggiuntivo che permette la connessione con il computer tramite normale presa usb.

La scheda per poter funzionare deve essere alimentata dall'esterno con una tensione che va dai 5 ai 9 V; inoltre alcuni led aiutano a verificare la corretta connessione con il computer e il corretto caricamento del programma in memoria.
Per le specifiche tecniche si rimanda alla documentazione ufficiale scaricabile gratuitamente dal sito della casa Xess.

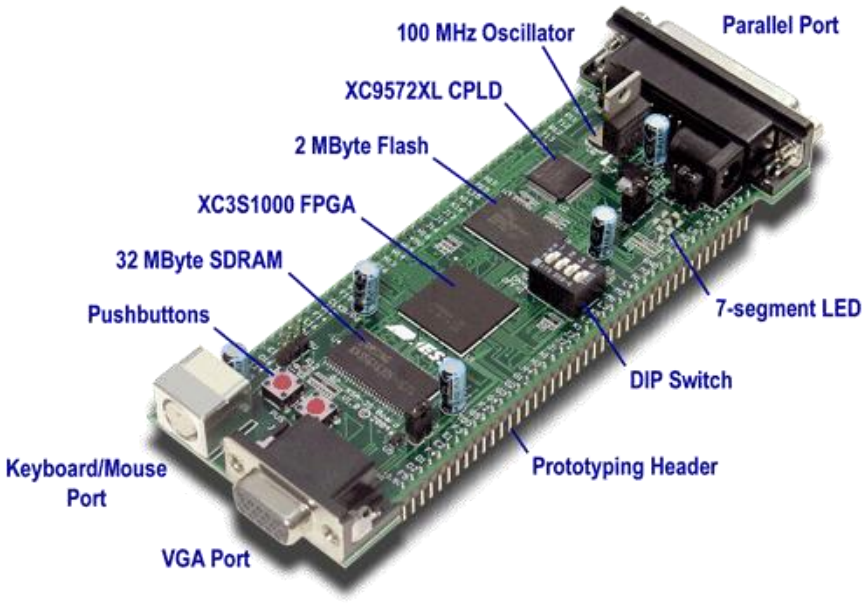

Figura 7.2: La scheda XSA-3S1000

Il processo di implementazione di un circuito logico digitale (e quindi anche di ADE32) su FPGA si compone principalmente di 5 passaggi che riportiamo di seguito:

1. Definizione del circuito logico mediante il linguaggio di descrizione dell'hardware (VHDL) e, volendo, una volta definiti i blocchi base, mediante schemi grafici creati grazie all'apposito editor di schemi contenuto sempre in ISE Design Suite.
2. Utilizzo di un sintetizzatore logico per trasformare il codice VHDL in una *netlist*, ovvero una descrizione dei componenti che fanno parte del circuito digitale e le loro connessioni.
3. Fase di implementazione vera e propria, in cui appositi *tool* mappano le porte logiche e le loro connessioni, mediante specifici processi, all'interno dell'FPGA, andando a configurare la matrice logica.
4. Un programma dedicato si occupa della trasformazione del risultato prodotto dalla fase 3 in un *bitstream*, ovvero una stringa di bit che non è altro che la digitalizzazione della configurazione matriciale del circuito FPGA.
5. Il *bitstream* prodotto dalla fase 4 viene fisicamente trasportato sulla scheda di lavoro in cui vi sono vari circuiti che riconoscono le informazioni portati dalla stringa di bit e che implementano fisicamente il circuito digitale.

Si riporta ora una schematizzazione grafica dei punti precedentemente esposti:

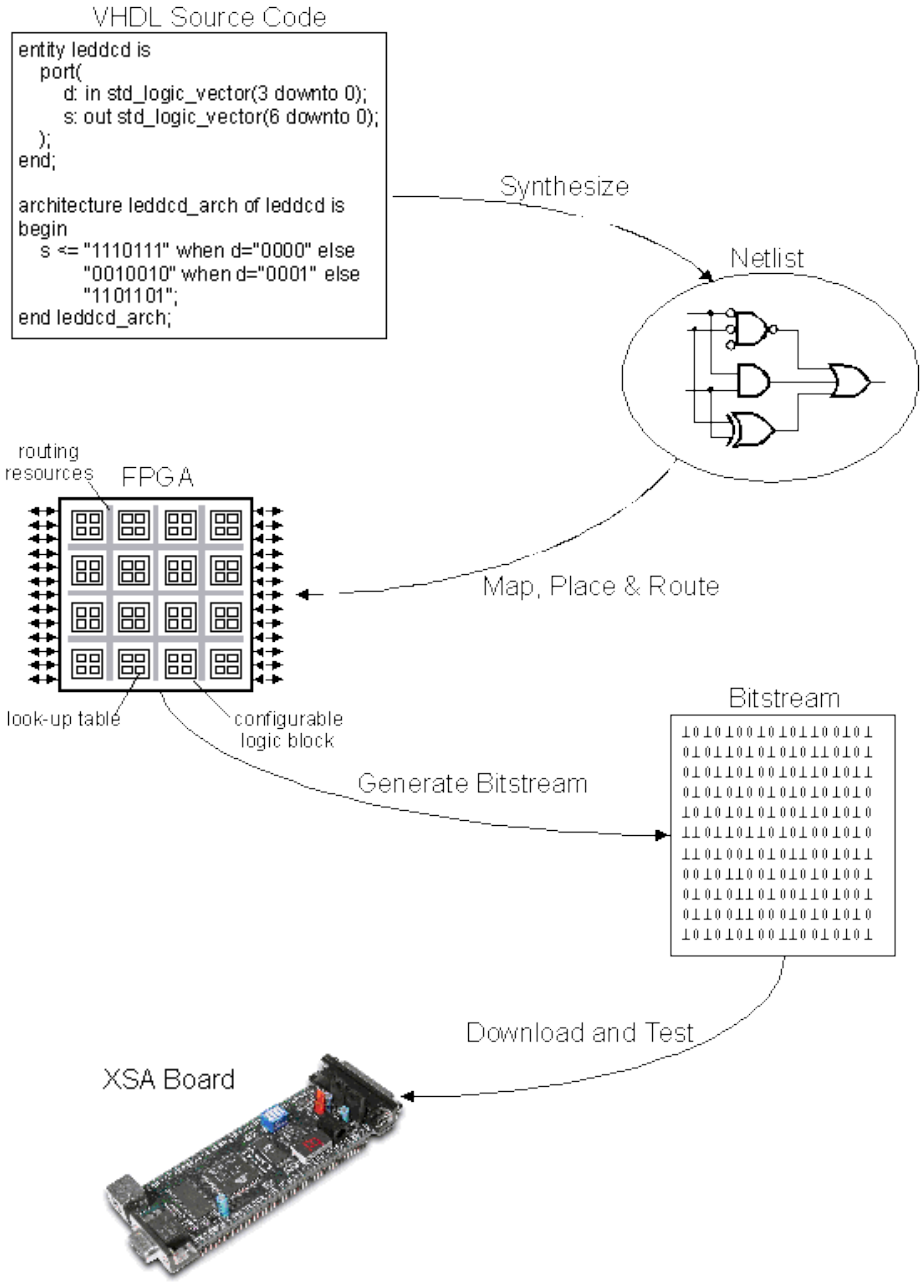

Figura 7.3: Schematizzazione dei 5 passaggi per l'implementazione su FPGA

I punti 1, 2, 3 e 4 sono realizzabili grazie agli strumenti forniti dalla Xilinx all'interno del WebPACK (ISE Design Suite), mentre il punto 4 è attuabile tramite i programmi e le schede fornite dalla Xess (XStool), che abbiamo visto nel Capitolo 2.

7.3 Realizzazione dei componenti

Viene ora riportata la realizzazione in VHDL dei singoli componenti necessari alla realizzazione dell'architettura ADE32.
Di ognuno si riporta il codice sorgente VHDL e il grafico del test effettuato con ISim.
Si segue sostanzialmente il progetto ottenuto con Logisim.

7.3.1 Registro 32, 20 e 4 bit

Il progetto di Logisim prevede l'utilizzo di tre tipi di registri: a 32 bit per i registri ACC, IR, MDR, ALUA, ALUOUT; a 20 bit per i registri PC e MAR; infine a 4 bit per il registro FLAG.

Codice per il registro a 32 bit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Register32bit is
    Port ( INPUT: in  STD_LOGIC_VECTOR(31 downto 0);
           ie: in  STD_LOGIC;
           oe: in  STD_LOGIC;
           clk: in  STD_LOGIC;
           rst: in  STD_LOGIC;
           OUTPUT: out STD_LOGIC_VECTOR(31 downto 0)
                  );
end Register32bit;

architecture Behavioral of Register32bit is
      signal value: STD_LOGIC_VECTOR (31 downto 0);
begin
      process (clk)
      begin
            if (clk'event and clk='1') then
                  if(rst = '1') then
                        value <= "00000000000000000000000000000000";
                  else
                        if(ie = '1') then
                              value <= INPUT;
                        end if;
                  end if;
            end if;
      end process;
      OUTPUT <= value when (oe = '1') else (others => 'Z');
end Behavioral;
```

Codice per il registro a 20 bit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Register20bit is
    Port ( INPUT: in  STD_LOGIC_VECTOR(19 downto 0);
           ie: in  STD_LOGIC;
           oe: in  STD_LOGIC;
           clk: in  STD_LOGIC;
           rst: in  STD_LOGIC;
           OUTPUT: out STD_LOGIC_VECTOR(19 downto 0)
                  );
end Register20bit;

architecture Behavioral of Register20bit is
      signal value: STD_LOGIC_VECTOR (19 downto 0);
begin
      process (clk)
      begin
            if (clk'event and clk='1') then
                  if(rst = '1') then
                        value <= "00000000000000000000";
                  else
                        if(ie = '1') then
                              value <= INPUT;
```

```
                        end if;
                    end if;
                end if;
        end process;
        OUTPUT <= value when (oe = '1') else (others => 'Z');
end Behavioral;
```

Codice per il registro a 4 bit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Register4bit is
    Port ( INPUT: in  STD_LOGIC_VECTOR(3 downto 0);
           ie: in  STD_LOGIC;
           oe: in  STD_LOGIC;
           clk: in  STD_LOGIC;
           rst: in  STD_LOGIC;
           OUTPUT: out STD_LOGIC_VECTOR(3 downto 0)
                 );
end Register4bit;

architecture Behavioral of Register4bit is
      signal value: STD_LOGIC_VECTOR (3 downto 0);
begin
      process (clk)
      begin
            if (clk'event and clk='1') then
                  if(rst = '1') then
                        value <= "0000";
                  else
                        if(ie = '1') then
                              value <= INPUT;
                        end if;
                  end if;
            end if;
      end process;
      OUTPUT <= value when (oe = '1') else (others => 'Z');
end Behavioral;
```

Come è possibile notare il codice dei vari registri è molto simile, cambia solamente la dimensione del bus dati in input e in output mentre il funzionamento generale rimane inalterato: quando il segnale di abilitazione dell'input (`ie`, input enable) vale 1 e si ha il fronte di salita del clock (`clk`) il registro memorizza il segnale presente in input, altrimenti non si ha il campionamento del valore. Se il segnale di output (`oe`, output enable) vale 1 il registro fornisce in uscita il valore memorizzato, altrimenti l'uscita è in alta impedenza (`Z`).
Il segnale reset (`rst`) posto a 1 azzera il contenuto del registro.
Riportiamo il test eseguito con ISim sul registro a 32 bit:

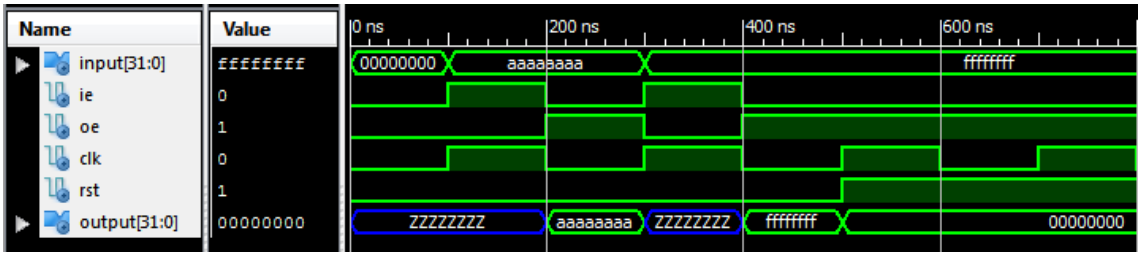

Figura 7.4: Test del registro a 32 bit

Si può notare come il registro campioni correttamente il valore in input al fronte di salita del clock e segnale di input attivo; con l'arrivo del segnale di reset il contenuto invece si azzera (e si mantiene azzerato in quanto il segnale di reset si mantiene sul valore logico 1 dai 500 ns in poi).

7.3.2 Buffer tri-state a 32 bit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Tristate32bit is
port(INPUT: in STD_LOGIC_VECTOR(31 downto 0);
     OUTPUT: out STD_LOGIC_VECTOR(31 downto 0);
     enable: in STD_LOGIC
        );
end Tristate32bit;

architecture Behavioral of Tristate32bit is
signal nullo: STD_LOGIC_VECTOR(31 downto 0):=
"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
begin
      OUTPUT <= nullo;
      process(enable)
      begin
      if(enable = '1') then
            OUTPUT <= INPUT;
      else
            OUTPUT <= nullo;
      end if;
      end process;
end Behavioral;
```

I buffer tri-state servono per abilitare o meno il passaggio di dati su di un bus di comunicazione; in particolare trasferiscono i dati dalla loro porta input alla porta output in presenza del segnale di abilitazione (`enable`), altrimenti forniscono in uscita valori di alta impedenza (`Z`).
Nello specifico, a differenza del progetto di Logisim, vengono impiegati solamente due tri-state per regolare l'input ed output del bus dati (collegati ai segnali MDR_{BUSIE} e MDR_{BUSOE}) in quanto i registri implementati, come abbiamo visto, contengono già al loro interno la logica tri-state fornendo un'uscita in alta impedenza nel momento in cui il segnale di output vale 0.
Si riporta di seguito il test eseguito sul tri-state a 32 bit con ISim:

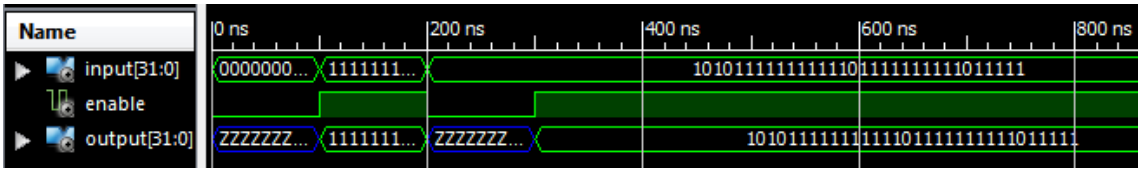

Figura 7.5: Test del tri-state a 32 bit

Come si può notare dal grafico quando il segnale `enable` vale 1 il valore in input viene correttamente trasferito in output, mentre quando il segnale di `enable` vale 0 in uscita abbiamo un valore posto in alta impedenza, sfruttando il tipo dato IEEE (evidenziato dal colore blu e dal simbolo `Z`).

7.3.3 Bit-extender 20-32 bit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ext20_32 is
    Port ( INPUT : in  STD_LOGIC_VECTOR(19 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR(31 downto 0)
                 );
end ext20_32;

architecture Behavioral of ext20_32 is
signal temp: STD_LOGIC_VECTOR(11 downto 0);
begin
      temp <= "000000000000";
      OUTPUT <= temp & INPUT ;
end Behavioral;
```

Il bit-extender 20-32 bit estende una parola di 20 bit in una parola di 32 bit aggiungendo in testa 12 zeri che non vanno a modificare, se non per la lunghezza in bit, il valore originario. In particolare viene impiegato per collegare l'output dei registri a 20 bit (MAR e PC) al bus interno del data path a 32 bit.
L'operatore di concatenamento in VHDL è l'ampersand (`&`), in questo caso si concatena una stringa composta da 12 zero con la stringa del valore ricevuto in input.
Si riporta di seguito il test eseguito con ISim:

Figura 7.6: Test del bit-extender 20-32 bit

Il valore in input a 20 bit viene correttamente esteso con degli zero a 32 bit.

7.3.4 Bit-extender 32-20 bit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ext32_20 is
    Port ( INPUT : in  STD_LOGIC_VECTOR(31 downto 0);
           OUTPUT : out  STD_LOGIC_VECTOR(19 downto 0)
                 );
end ext32_20;

architecture Behavioral of ext32_20 is
begin
      OUTPUT <= INPUT(19 downto 0);
end Behavioral;
```

In maniera analoga al dispositivo descritto sopra, il bit-extender 32-20 bit riduce la lunghezza di una parola di 32 bit a 20 bit, troncando i 12 bit più significativi della parola originaria. Tale dispositivo viene impiegato per collegare l'input dei registri a 20 bit (MAR e PC) al bus interno del data path a 32 bit.

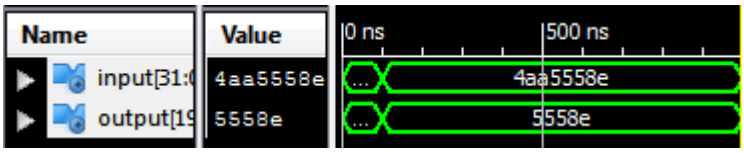

Figura 7.7: Test del bit-extender 32-20 bit

Come si può osservare dal grafico, il valore in input a 32 bit viene correttamente ridotto a 20 bit, eliminando i bit più significativi.

7.3.5 ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU is
      port( A, B: in STD_LOGIC_VECTOR(31 downto 0);
            operazione: in STD_LOGIC_VECTOR(3 downto 0);
            flag: out STD_LOGIC_VECTOR(3 downto 0);
            OUTPUT: out STD_LOGIC_VECTOR(31 downto 0)
            );
end ALU;

architecture Behavioral of ALU is
signal zero: STD_LOGIC_VECTOR(31 downto 0);
begin
      attivazione: process(A, B)
      variable temp: std_logic_vector(32 downto 0):=
"00000000000000000000000000000000";
      begin
      zero <= "00000000000000000000000000000000";
      case operazione is
            when "0000" =>
                  temp := zero and B;
            when "0001" =>
                  temp := zero and B;
            when "0010" =>
                  temp := A and B;
            when "0011" =>
                  temp := A and B;
            when "0100" =>
                  temp := zero or B;
            when "0101" =>
                  temp := zero or B;
            when "0110" =>
                  temp := A or B;
            when "0111" =>
                  temp := A or B;
            when "1000" =>
                  temp := std_logic_vector(0 + unsigned(B));
            when "1001" =>
                  temp := std_logic_vector(0 + unsigned(B) + 1);
            when "1010" =>
                  temp:= std_logic_vector((unsigned(A)+unsigned(B)));
            when "1011" =>
                  temp :=std_logic_vector((unsigned(A)+unsigned(B)+1));
            when "1100" =>
                  temp := std_logic_vector(0 + unsigned(not B));
            when "1101" =>
                  temp := std_logic_vector(0 + unsigned(not B) + 1);
```

```
            when "1110" =>
                  temp :=std_logic_vector(unsigned(A)+unsigned(not B));
            when others =>
                  temp:=std_logic_vector((unsigned(A)+unsigned(notB)+1));
      end case;
      --genero i segnali di FLAG
      if (temp = "00000000000000000000000000000000") --flag(0)->Zero
            then flag(0) <= '1';
      else flag(0) <= '0';
      end if;

      if (temp(31) = '1') --flag(1)->Negativo
            then flag(1) <= '1';
      else flag(1) <= '0';
      end if;

      if (temp(32) = '1') --flag(2)->Carry Out
            then flag(2) <= '1';
      else flag(2) <= '0';
      end if;

      if (temp(31)='1' and A(31)='0' and B(31)='0')--flag(3)->Overflow
            then flag(3) <= '1';
      else flag(3) <= '0';
      end if;

      OUTPUT <= temp(31 downto 0);

      end process;
end Behavioral;
```

Nella prima parte del codice si eseguono le operazioni di tipo logico-aritmetico (la codifica avviene mediante il costrutto `case`) mentre nella seconda parte si elaborano i segnali di flag Zero, Negativo, Carry Out, Overflow, che corrispondono a `flag(0)`, `flag(1)`, `flag(2)`, `flag(3)`.
Il bus `operazione` si incarica di portare in ingresso i bit necessari alle operazioni dell'ALU, a differenza del progetto di Logisim in cui i 4 segnali sono mantenuti distinti tra di loro. Come riportato nel quarto capitolo l'ALU funziona in modo asincrono, ovvero non viene pilotata dall'esterno da un segnale di sincronizzazione (clock) ma elabora in modo continuo i segnali posti in ingresso sulle sue due porte (`A` e `B`).
Il segnale `output` (32 bit) viene ricavato dalla variabile `temp` che avendo una lunghezza di 33 bit necessita di essere ridotta di 1 bit prima dell'assegnamento.
Riportiamo il test eseguito con ISim in cui viene testata l'operazione di AND:

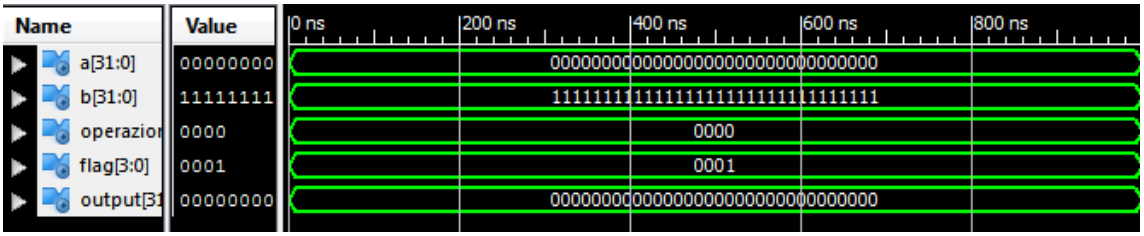

Figura 7.7: Test dell'ALU a 32 bit

Come si può vedere l'operazione "`0000`" corrisponde all'AND logico (cfr. Capitolo 7); il registro di flag giustamente riporta il valore "`0001`" in quanto il primo bit rileva la presenza di un risultato nullo, come appunto abbiamo ottenuto in questo caso.

7.3.6 Data path

Lo schema del data path segue sostanzialmente lo schema del progetto di Logisim ed è stato ottenuto utilizzando l'editor grafico presente in ISE Design Suite.
I componenti di base utilizzati sono quelli descritti in precedenza.

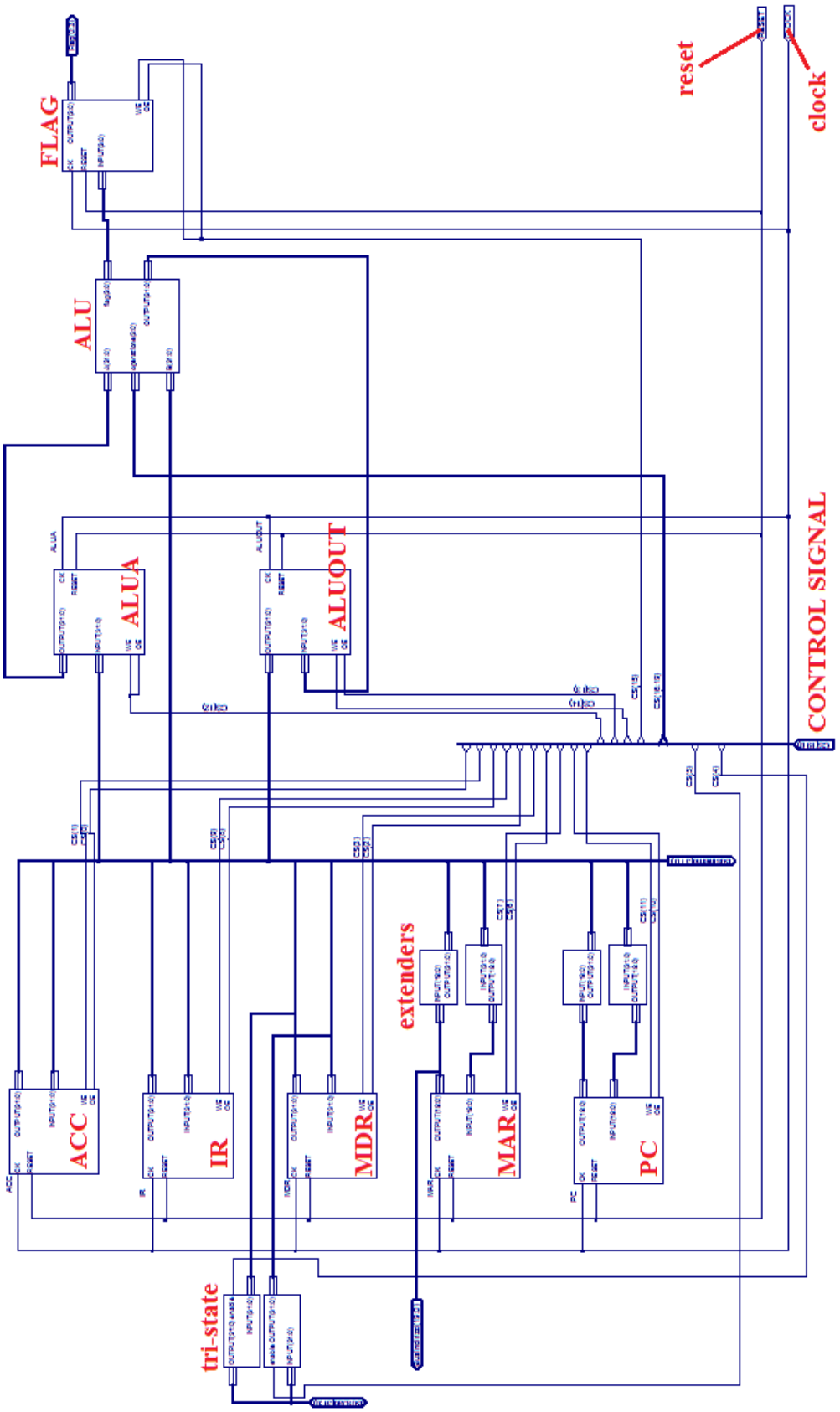

Figura 7.8: Schema grafico del data path ottenuto con ISE Design Suite

7.3.7 Control Unit

A differenza del progetto di Logisim non si è creata l'Unità di Controllo mediante un unico componente ma con tre blocchi distinti: una ROM, un registro a 12 bit e un circuito di controllo; per la loro sincronizzazione è stato necessario portare a ogni blocco il segnale di clock.
Riportiamo subito lo schema creato tramite l'editor grafico, poi definiamo i blocchi da cui è composto:

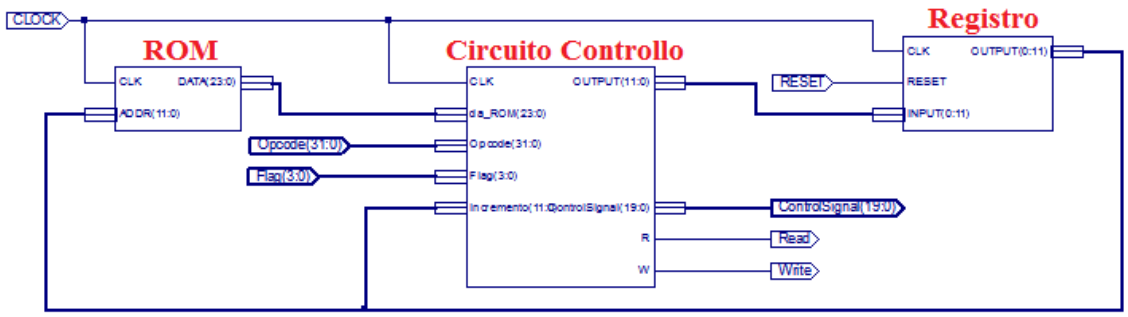

Figura 7.9: Schema grafico della Control Unit ottenuta con ISE Design Suite

Riportiamo il codice del registro a 12 bit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Registro12bit is
      port ( INPUT: in STD_LOGIC_VECTOR(0 to 11);
             OUTPUT: out STD_LOGIC_VECTOR(0 to 11);
             clk: in STD_LOGIC;
             rst: in STD_LOGIC
                   );
end Registro12bit;

architecture Behavioral of Registro12bit is
begin
      attivazione: process(clk, rst)
      begin
            if(rst = '1') then
                  OUTPUT <= (others => '0');
            elsif (clk'event and clk = '1') then
                  OUTPUT <= INPUT;
            end if;
      end process;
end Behavioral;
```

Il codice del registro risulta diverso da quello dei registri implementati nel data path, questo perché nell'Unità di Controllo non è necessario fornire un'uscita in alta impedenza in quanto non ci sono bus condivisi da pilotare.

Il Circuito di Controllo è così definito in VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Circuito_Controllo is
    Port ( da_ROM: in STD_LOGIC_VECTOR (23 downto 0);
           clk: in std_logic;
           Opcode: in STD_LOGIC_VECTOR (31 downto 0);
```

```
            Flag: in STD_LOGIC_VECTOR (3 downto 0);
            Incremento: in STD_LOGIC_VECTOR (11 downto 0);
            R: out std_logic;
            W: out std_logic;
            OUTPUT: out STD_LOGIC_VECTOR (11 downto 0);
            ControlSignal: out STD_LOGIC_VECTOR (19 downto 0));
end Circuito_Controllo;

architecture Behavioral of Circuito_Controllo is
      signal out1: STD_LOGIC; --uscita primo mux
      signal out2: STD_LOGIC_VECTOR(11 downto 0); --uscita secondo mux
      signal contr_3mux: STD_LOGIC; --controllo terzo mux

begin
      ControlSignal <= da_ROM(19 downto 0);
      R <= da_ROM(4);
      W <= da_ROM(5);
      process(clk)
      begin
      case da_ROM(23 downto 20) is --MULTIPLEXER 1
            when "0000" =>
                  out1 <= '0';
            when "0001" =>
                  out1 <= '1';
            when "0010" =>
                  out1 <= Flag(0);--Z
            when "0011" =>
                  out1 <= (not Flag(0));--Z'
            when "0100" =>
                  out1 <= Flag(1);--N
            when "0101" =>
                  out1 <= (not Flag(1));--N'
            when "0110" =>
                  out1 <= Flag(2);--C
            when "0111" =>
                  out1 <= (not Flag(2));--C'
            when "1000" =>
                  out1 <= Flag(3);--O
            when "1001" =>
                  out1 <= (not Flag(1) and not Flag(0));
            when "1010" =>
                  out1 <= ((Flag(1) xnor Flag(3)) and Flag(0));
            when "1011" =>
                  out1 <= (Flag(1) xnor Flag(3));
            when "1100" =>
                  out1 <= (Flag(1) or Flag(0));
            when "1101" =>
                  out1 <= (Flag(1) xor Flag(3));
            when "1110" =>
                  out1 <= ((Flag(1) xor Flag(3)) or Flag(0));
            when others =>
                  out1 <= '0';
      end case;

      case out1 is --MULTIPLEXER 2
            when '0' =>
                  out2 <= (others => '0');
            when others =>
                  out2 <= std_logic_vector(unsigned(Incremento) + 1);
      end case;
```

```
      contr_3mux  <=  da_ROM(23)  and  da_ROM(22)  and  da_ROM(21)  and
da_ROM(20);

      case contr_3mux is --MULTIPLEXER 3
            when '0' =>
                  OUTPUT <= out2;
            when others =>
                  OUTPUT <= Opcode(31 downto 20);
      end case;
      end process;
end Behavioral;
```

I tre multiplexer messi in cascata tra di loro nel progetto di Logisim vengono qui sostituiti con l'utilizzo del costrutto `if` e `case`. Il segnale `da_ROM` in ingresso viene diviso in due parti, la prima, composta da 20 bit viene inviata al data path come bus dei segnali di controllo mentre la seconda parte, composta da 4 bit, viene utilizzata per il funzionamento del Circuito di Controllo.
I segnali di lettura e scrittura della memoria (`R` e `W`) vengono inviati direttamente in uscita, senza quindi essere mandati prima al data path (da cui non è possibile portarli in uscita per esigenze di sintesi a basso livello).

Riportiamo infine il codice VHDL della memoria ROM:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ROM is
port (ADDR : in std_LOGIC_VECTOR(11 downto 0);
      CLK : in STD_LOGIC;
      DATA : out STD_LOGIC_VECTOR(23 downto 0));
end ROM;

architecture syn of ROM is
    type rom_type is array (63 downto 0) of STD_LOGIC_VECTOR (23
downto 0);
    signal ROM : rom_type:=
                     (X"00200A", X"000300", X"008101", X"004000", X"008601", X"02033A",
                      X"000300", X"008602", X"002310", X"00203B", X"008300", X"040002",
                      X"008201", X"000500", X"004001", X"002500", X"000340", X"000241",
                      X"004002", X"008300", X"008201", X"000500", X"008101", X"000602",
                      X"004003", X"00241E", X"000301", X"000102", X"002122", X"002021",
                      X"000301", X"000102", X"002222", X"004001", X"000342", X"00232B",
                      X"000900", X"000302", X"000102", X"004002", X"000900", X"008201",
                      X"002023", X"000303", X"002433", X"000301", X"004004", X"000301",
                      X"001002", X"021307", X"020036", X"000301", X"001002", X"002237",
                      X"004004", X"000304", X"004040", X"002500", X"002500", X"002500",
                      X"00030D", X"F00284", X"102820", X"1A4480");

begin
       process (CLK, ADDR)
       begin
    DATA <= ROM(conv_integer(ADDR));
       end process;
end syn;
```

Il contenuto della memoria è inizializzato solo per la prima operazione di fetch.
Come si può notare a fronte di un segnale di indirizzo (ADDR) fornito in ingresso il circuito fornisce il valore memorizzato nell'array nella posizione equivalente all'indirizzo (che viene convertito tramite la funzione `conv_integer`).

7.3.8 ADE32

Viene ora riportata la struttura di ADE32 creata sempre tramite l'editor grafico presente in ISE Design Suite. Il processore è diviso in due parti a livello logico che interagiscono fra di loro mediante dei bus di comunicazione (Bus Interno a 32 bit e Flag a 4 bit).
Dal data path parte il Bus Indirizzi e il Bus dati (bidirezionale). Come abbiamo visto in precedenza i segnali di `Read` e `Write` vengono inviati all'esterno direttamente dall'Unità di Controllo.

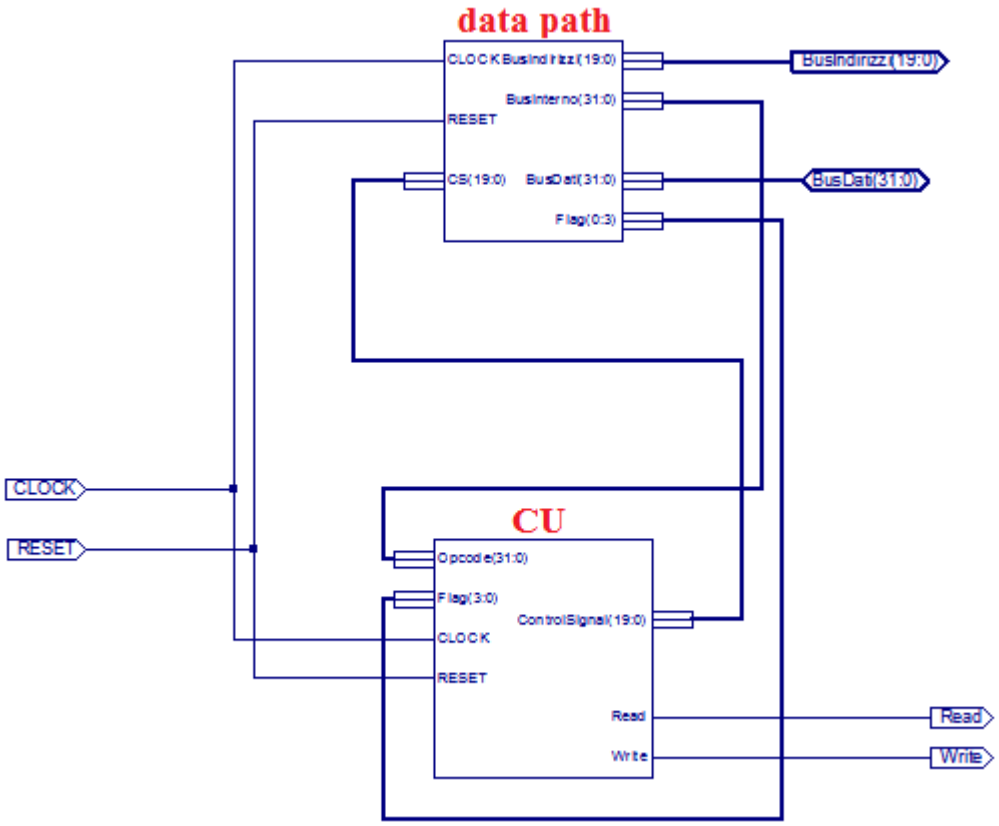

Figura 7.10: Schema grafico di ADE32 ottenuta con ISE Design Suite

7.3.9 Motherboard

La scheda madre (cfr. Figura 0.1, Introduzione) ospita i componenti che si interfacciano con il processore e che quindi fanno parte del calcolatore; nel nostro caso sono presenti una memoria RAM da cui vengono lette le istruzioni e i dati e un I/O Register che si occupa dell'output effettuato su dei display led a 7 segmenti: è un output a 32 bit ma di cui vengono soltanto utilizzati i primi 4 in quanto la scheda XSA-3S1000 ospita al suo interno un solo display led.
Il blocco chiamato nella Figura 7.11 Display LED ospita la logica necessaria alla traduzione dell'input a 4 bit e dell'output a 7 bit, necessario per far funzionare correttamente il display a sette segmenti.
Come da progetto, il funzionamento dell'unità di display esterna e della memoria viene regolata mediante il bit più significativo del Bus Indirizzi (quindi il bit 20) che viene utilizzato anche in forma negata grazie all'apposito inverter.
Risulta quindi possibile scrivere e leggere valori dall'I/O Register.
Infine sono presenti i soliti segnali di Clock (CLOCK) e reset (RESET) che si propagano a tutti i dispositivi che abbiamo visto in precedenza.

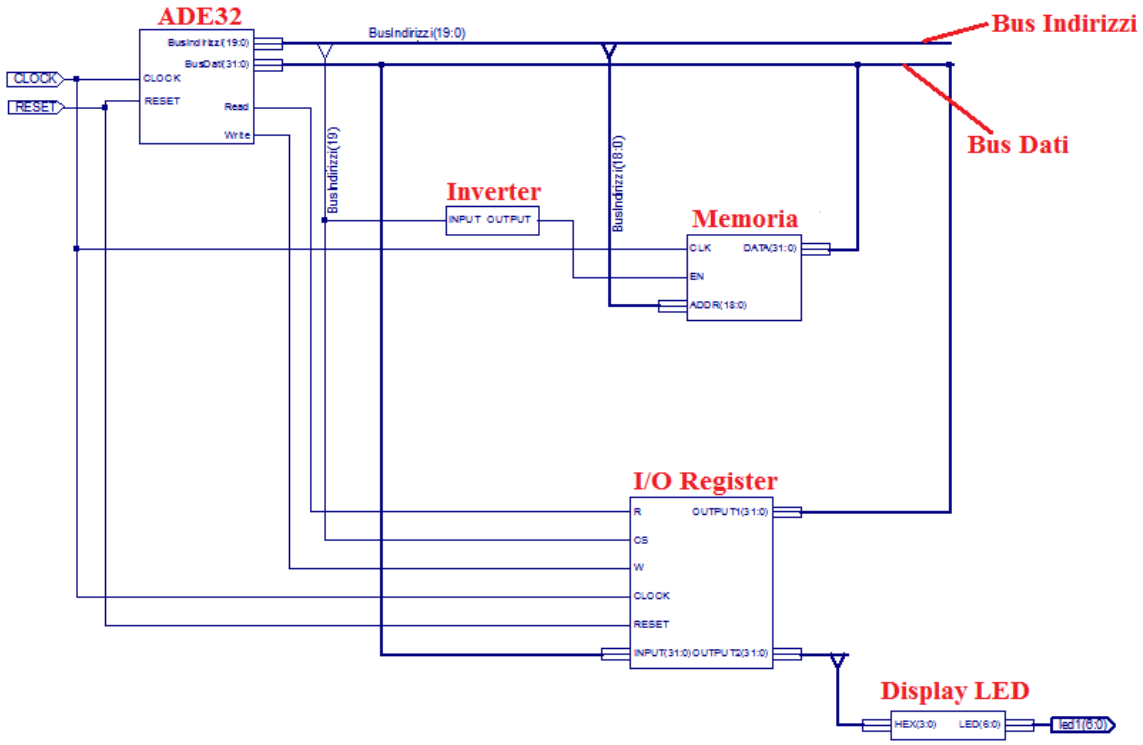

Figura 7.11: Schema grafico della Motherboard ottenuta con ISE Design Suite

Riportiamo ora il codice, molto semplice, dell'Inverter:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity inverter is
    Port ( INPUT: in  STD_LOGIC;
           OUTPUT: out  STD_LOGIC);
end inverter;

architecture Behavioral of inverter is
begin
      OUTPUT <= not INPUT;
end Behavioral;
```

Il codice dell'I/O Register invece è il seguente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ioRegister is
    Port ( R : in  STD_LOGIC;
           CS : in  STD_LOGIC;
           W : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           RESET : in  STD_LOGIC;
           INPUT : in  STD_LOGIC_VECTOR (31 downto 0);
           OUTPUT1 : out  STD_LOGIC_VECTOR (31 downto 0);
           OUTPUT2 : out  STD_LOGIC_VECTOR (31 downto 0));
end ioRegister;

architecture Behavioral of ioRegister is
signal contenuto : STD_LOGIC_VECTOR(31 downto 0);
begin
      contenuto <= "00000000000000000000000000000000";
      attivazione: process(clk, RESET, CS)
      begin
```

```
      if(R='1' and CS='1') then
            OUTPUT1 <= contenuto;
      end if;
      if(W='1' and CS='1') then
            contenuto <= INPUT;
      end if;
      if(RESET = '1') then
            contenuto <= "00000000000000000000000000000000";
      end if;
      if(CS = '0') then
            OUTPUT1 <=   "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      end if;
      OUTPUT2 <= contenuto;
      end process;

end Behavioral;
```

L'uscita che porta i valori al display risulta essere sempre attiva (di base mostra il valor 0) mentre l'uscita che porta i valori al bus dati è attiva solamente con il segnale di selezione del dispositivo (CS, chip select) attivo posto in and con il segnale di scrittura (W).
E infine riportiamo il codice del Display LED:

```
library ieee;
use ieee.std_logic_1164.all;
entity display7led is
   Port (HEX: in STD_LOGIC_VECTOR (3 downto 0);
         LED: out STD_LOGIC_VECTOR (6 downto 0)
                );
end display7led;
--
-- segment encoinputg
--      0
--     ---
--  5 |   | 1
--     ---   <- 6
--  4 |   | 2
--     ---
--      3
architecture Behavioral of display7led is
begin
   LED<= "0000111" when HEX="0001" else   --1
         "1011011" when HEX="0010" else   --2
         "1001111" when HEX="0011" else   --3
         "1100110" when HEX="0100" else   --4
         "1101101" when HEX="0101" else   --5
         "1111101" when HEX="0110" else   --6
         "0000111" when HEX="0111" else   --7
         "1111111" when HEX="1000" else   --8
         "1101111" when HEX="1001" else   --9
         "1110111" when HEX="1010" else   --A
         "1111100" when HEX="1011" else   --b
         "0111001" when HEX="1100" else   --C
         "1011110" when HEX="1101" else   --d
         "1111001" when HEX="1110" else   --E
         "1110001" when HEX="1111" else   --F
         "0111111";   --0
 end Behavioral;
```

Conclusioni

L'obiettivo di questo progetto era la simulazione e realizzazione di una CPU a 32 bit; possiamo quindi dividere in due parti le conclusioni del lavoro svolto:

- **Simulazione della CPU**
  Dal punto di vista della simulazione del processore, mediante il programma Logisim, l'esperienza è stata del tutto soddisfacente, i risultati conseguiti sono stati molto buoni: si è giunti ad ottenere un modello perfettamente funzionante.
  Attraverso la simulazione si è verificata l'effettiva funzionalità del processore e si sono andate a controllare le varie caratteristiche di progetto.
  Non secondario a mio parere il fatto che il costo dell'intera parte risulti essere nullo.

- **Realizzazione della CPU**
  Questa seconda parte prevedeva l'acquisizione di un nuovo linguaggio, il VHDL, che non rientra tra quelli visti durante i tre anni del corso di laurea, e l'utilizzo di una scheda FPGA per l'implementazione fisica del processore.
  Si è simulato con successo, mediante l'ambiente di ISim, il comportamento dei blocchi logici realizzati con il VHDL.
  Conoscenze più approfondite riguardo alla sintesi di basso livello avrebbero sicuramente contribuito a implementare in maniera perfettamente funzionante ADE32 sull'FPGA.

Questo progetto potrebbe essere un valido "trampolino di lancio" per attività future. La stessa natura *general purpose* di ADE32, come del resto delle moderne CPU, permette di immaginare sviluppi notevoli anche se confinati probabilmente sempre nell'ambito didattico (sarebbe difficile fare concorrenza a case come l'Intel, la Motorola, l'AMD...).
Si potrebbe ad esempio pensare a cominciare ad avvicinarsi ad un'architettura di tipo vettoriale, ovvero un'architettura che prevede l'elaborazione contemporanea di più dati elementari, inserendo per esempio nella struttura delle *pipeline*.
O ancora si potrebbe inserire un'unità di memoria *cache* con la quale velocizzare il reperimento delle istruzioni da parte del processore: sarebbe interessante poi testare realmente il comportamento del processore evidenziando la differenza di prestazioni.
Infine si potrebbero inserire validi miglioramenti atti ad aumentare il realismo del processore, come ad esempio una lunghezza variabile per le istruzioni, il che naturalmente comporterebbe un notevole aumento della complessità della realizzazione della CPU.

Nello svolgimento di questa relazione si è fatto ricorso a molte delle nozioni acquisite tramite il percorso di studi di questi tre anni presso la Facoltà di Ingegneria: in particolar modo le nozioni provengono dai corsi che trattavano la programmazione in C (Fondamenti di Informatica I e Fondamenti di Informatica II) e il corso di Calcolatori Elettronici e Lab. per tutta la parte riguardante la CPU, la sua architettura, microarchitettura ecc.

Bibliografia

- Andrew S. Tanenbaum: *Architettura dei Calcolatori - Un approccio strutturale*, Pearson Education Italia S.r.l, 2006

- Carlo Brandolese: *Introduzione al linguaggio VHDL - Aspetti teorici ed esempi di progettazione*, disponibile gratuitamente all'indirizzo http://static.gest.unipd.it/~mattavelli/FPGA_lab/Manuali_pdf/VHDL_intro2.pdf (verificato al 12/12)

- Peter J. Ashenden: *The VHDL Cookbook*, disponibile gratuitamente all'indirizzo http://www.cs.ucr.edu/~vahid/courses/vhdl_cookbook/ (verificato al 12/12)

Sitografia

- Sito della Xilinx su cui è possibile trovare, in lingua inglese, abbondante documentazione riguardo sia alla parte software che hardware (FPGA).
  Ospita inoltre il download del pacchetto di programmi WebPACK, in cui è incluso il programma utilizzato in questo elaborato ISE Design Suite.
  http://www.xilinx.com/ (verificato al 12/12)

- Sito di Logisim da cui è possibile scaricare gratuitamente il programma e visionare la documentazione in lingua inglese.
  http://ozark.hendrix.edu/~burch/logisim/ (verificato al 12/12)

- Sito della Xess sul quale si possono trovare le specifiche tecniche delle schede da loro prodotte, compresi i manuali gratuitamente scaricabili in pdf e i software necessari all'implementazione su scheda.
  http://www.xess.com/index.php (verificato al 12/12)

Video

- Alcuni video riguardanti la parte di simulazione su Logisim possono essere visionati al seguente indirizzo:
  http://www.youtube.com/watch?v=OYG0kUXL7io (verificato al 12/12)
  http://www.youtube.com/watch?v=v7RtTm9m3zc (verificato al 12/12)