



UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

Softcore Minimale per FPGA

Relatore:

Prof: Marco Danelutto

Candidato:

Simone Stanganini

ANNO ACCADEMICO 2022/2023

Indice

1	Introduzione	5
1.1	Contesto	5
1.2	Obiettivo	5
1.3	Scaletta Tesi	6
2	Strumenti	7
2.1	Microblaze	8
2.2	Alveo U50	9
2.3	Vitis	10
2.4	OpenCL	11
2.5	C++/C	12
3	Progetto Logico	13
3.1	Funzionamento	15
4	Implementazione	19
4.1	Interprete Softcore	20
4.2	Interprete Versione Kernel	32
4.3	Interfaccia Host	35
4.4	Compilazione	40
4.5	Interprete Floating Point	42
4.6	Control Unit	44
4.7	Simulazione GPU con codice MIMD	48
5	Risultati	51
5.1	Verifica Funzionamento	51
5.2	Utilizzo FPGA	61
5.3	Execution Summary	66
6	Conclusioni	67
6.1	Bilancio Personale	67
A	Appendice	71
A.1	Codice Host 1 Kernel	71
A.2	Codice Interprete 1 Kernel	75

A.3	Codice Host GPU	88
A.4	Codice Interprete Floating Point	95
A.5	File Assembler	107

Capitolo 1

Introduzione

1.1 Contesto

Nel mondo dell'informatica l'utilizzo dei dispositivi FPGA (Field-Programmable Gate Array) è diventato sempre più comune per l'accelerazione di algoritmi e applicazioni specifiche attraverso hardware "customizzato", in modo da migliorare prestazioni, flessibilità e personalizzazione. Qui entrano in gioco i processori softcore, che sono diventati una risorsa essenziale per sfruttare al massimo le potenzialità dei dispositivi FPGA.

I processori softcore sono stati sviluppati per essere configurabili e programmabili in modo da adattarsi ad ogni specifica, questa loro caratteristica li rende adatti a una vasta gamma di compiti, dall'elaborazione dati all'elaborazione dei segnali, dall'elaborazione delle immagini all'automazione industriale, rendendo più semplice lo sviluppo di queste applicazioni anche per sviluppatori senza specifiche esperienze "hardware".

1.2 Obiettivo

Per questa mia tesi, mi è stato assegnato il compito di sviluppare un semplice interprete che sia in grado di eseguire un sottoinsieme delle istruzioni del processore Microblaze, softcore sviluppato da Xilinx e successivamente sperimentare questo softcore molto piccolo sulla FPGA. Lo scopo era quello di enfatizzare quanto questa soluzione sia flessibile e configurabile, misurarne l'occupazione sul dispositivo, verificarne il corretto funzionamento e determinare il numero massimo di istanze che possono essere inserite nel dispositivo.

In prospettiva, il softcore sviluppato dovrebbe essere utilizzato per simulare una GPU con core indipendenti e privi di controllore SIMD, mettendo in evidenza il potenziale di accelerazione offerto da questa configurazione, in particolare nella com-

putazione di applicazioni data parallel con thread di controllo "divergenti", ovvero che nello stesso istante eseguono computazioni localmente differenti tra di loro.

1.3 Scaletta Tesi

Il resto di questa tesi è organizzata quanto segue:

- **Capitolo 2:** Descrive gli strumenti usati per lo sviluppo del progetto.
- **Capitolo 3:** Fornisce una panoramica generale sul funzionamento del progetto, e riporta le possibilità di configurazione e estensione del progetto.
- **Capitolo 4:** Fornisce in dettaglio la spiegazione dell'implementazione delle componenti del progetto.
- **Capitolo 5:** Riporta i risultati ottenuti sull'occupazione della FPGA, come è stato verificato il funzionamento delle diverse versioni dell'interprete.
- **Capitolo 6:** Conclude la tesi riassumendo i risultati ottenuti durante il percorso di sviluppo del progetto. Inoltre offre un bilancio personale sul l'intero lavoro svolto.

Capitolo 2

Strumenti

In questo capitolo si descrivono gli strumenti usati per lo sviluppo del progetto:

- **Processore Microblaze**, il processore MicroBlaze è stato utilizzato come base per l'implementazione dell'interprete e del suo relativo set di istruzioni (sezione [2.1](#)).
- **Scheda Alveo U50**, un'FPGA di Xilinx, è stata utilizzata per condurre gli esperimenti di implementazione (sezione [2.2](#)).
- **Software Vitis**, fondamentale per programmare e sfruttare al massimo le capacità della scheda U50 (sezione [2.3](#)).
- **Il framework OpenCL** è stato impiegato per consentire la comunicazione tra l'host e l'FPGA, sfruttando al meglio le capacità di accelerazione hardware (sezione [2.4](#)).
- **Linguaggio di programmazione C++/C**, il linguaggio di programmazione C++ è stato utilizzato per gestire le chiamate OpenCL e altre parti dell'applicazione. Inoltre, il linguaggio C è stato impiegato per scrivere l'interprete stesso. (sezione [2.5](#))

2.1 Microblaze

Il MicroBlaze è un processore a microcontrollore configurabile (soft core) progettato da Xilinx, un'azienda specializzata in dispositivi programmabili come FPGA (Field-Programmable Gate Arrays) e SoC (System on Chip).

E' stato sviluppato per essere implementato all'interno delle schede FPGA Xilinx e svolge il ruolo di un processore personalizzabile, ovvero che può essere configurato in base alle specifiche esigenze dell'applicazione.

Alcune caratteristiche possono essere riassunte come segue:

1. **Configurabilità/Flessibilità.** Gli sviluppatori possono scegliere tra le varie versioni del Softcore e selezionare le funzionalità richieste per l'applicazione specifica che cercano di sviluppare, come la cache, le varie interfacce per le periferiche, l'unità di gestione degli interrupt e altro. Questo rende il processore utilizzabile in una varietà di applicazioni, tra cui controllo industriale, comunicazioni, video, sistemi embedded, e molto altro.
2. **Set di istruzioni.** Il MicroBlaze utilizza un set di istruzioni RISC (Reduced Instruction Set Computer) e può essere personalizzato per includere istruzioni personalizzate o estensioni per ottimizzare l'elaborazione specifica dell'applicazione.
3. **Consumo energetico.** Il MicroBlaze è stato progettato per essere efficiente dal punto di vista energetico, il che lo rende adatto per dispositivi a batteria e sistemi embedded in cui il consumo energetico è un aspetto critico.
4. **Supporto software.** Xilinx fornisce un ambiente di sviluppo software, come Xilinx SDK (Software Development Kit), che semplifica la programmazione e il debug di applicazioni.

In sostanza, il MicroBlaze è un processore personalizzabile che può essere adattato per adempiere a una vasta gamma di esigenze, ed è reso più facile da utilizzare grazie al supporto software fornito da Xilinx.

All'interno di questa tesi, abbiamo utilizzato solo una piccola parte delle istruzioni di questo processore. Abbiamo scelto di non considerare le sue caratteristiche hardware in quanto non erano necessarie per dimostrare la fattibilità di questa implementazione su un dispositivo FPGA.

2.2 Alveo U50

La Alveo U50 è una scheda di accelerazione FPGA sviluppata da Xilinx, è stata creata per consentire l'implementazione di software personalizzato che può essere accelerato tramite hardware. Questa combinazione offre efficienza e velocità che non sono raggiungibili attraverso il normale metodo di programmazione della CPU che ospita l'acceleratore.



Figura 2.1: Alveo U50

In altre parole, la Alveo U50 fornisce la capacità di adattare e ottimizzare il software per ottenere prestazioni eccezionali grazie all'elaborazione hardware su FPGA.

La Alveo U50 è caratterizzata da una buona potenza di calcolo e un'architettura molto configurabile, il che la rende adatta per una varietà di settori, tra cui intelligenza artificiale, elaborazione di dati, e molto altro. Inoltre, questa scheda è progettata per ridurre al minimo il consumo energetico, il che la rende ideale per applicazioni che richiedono una gestione efficiente delle risorse.

FEATURES	ALVEO U50
Architecture	UltraScale+
Form Factor	Half-Height, Half length single slot Low-Profile
Look Up Tables	872,000
HBM2 Memory	8GB
HBM2 Bandwidth	316GB/s ¹
Network Interface	1 x QSFP28 (100GbE) ²
Clock Precision	IEEE 1588
PCI Express	PCIe Gen3 x 16, dual PCIe Gen4 x 8, CCIX
Thermal Solution	Passive
Power (TDP)	75W

Figura 2.2: Specifiche Alveo U50 [1]

Nella Tabella 2.3 si trovano riassunte le principali caratteristiche e "dimensioni della scheda. Abbiamo scelto questa scheda perché era quella a disposizione nella macchina host usata per gli esperimenti di implementazione.

2.3 Vitis

Come scritto nella documentazione:

"The AMD Vitis™ software platform is a development environment for developing designs that includes FPGA fabric, Arm® processor subsystems, and AI Engines" [5].

In particolare Vitis Software Platform mette a disposizione i seguenti tool:

1. **Vitis Embedded** – per scrivere applicazioni in C/C++ e farle eseguire in codice per processori arm su piattaforme embedded.
2. **Compiler and simulators** – per implementare applicazioni usando l'AI Engine array.
3. **Vitis HLS** – per scrivere applicazioni C/C++ basate su gli IP blocks che hanno come target di esecuzione le schede FPGA.
4. **Vitis Model Composer** – "A model-based design tool that enables rapid design exploration within the MathWorks Simulink® environment" [5].
5. Un set di funzioni opensource tipo DSP, Vision, Solver, Ultrasound, BLAS, e altre ancora, che possono essere utilizzate nelle FPGA o con gli AI Engines in applicazioni "custom" .

Contrariamente all'approccio tradizionale RTL (Register-Transfer Level) di progettazione hardware, Vitis utilizza HLS (High-Level Synthesis), che rappresenta un cambiamento radicale di paradigma nella progettazione di FPGA.

Questo perché l'approccio HLS permette di scrivere applicazioni utilizzando codice in linguaggi di alto livello come C, C++, o OpenCL, invece di descrivere esplicitamente la logica hardware, così da tradurre il codice in un circuito hardware eseguibile dalla FPGA senza una vera e propria programmazione dettagliata di RTL.

Questo approccio rende accessibile la potenza di elaborazioni di un FPGA a sviluppatori che non hanno una profonda conoscenza del linguaggio RTL, così anche da rendere più semplice lo sviluppo di applicazioni in settori come l'intelligenza artificiale, elaborazione di immagini, e altro ancora.

2.4 OpenCL

"OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms." [9]



Figura 2.3: OpenCL logo

Il framework messo a disposizione da OpenCL crea uno standard per lo sviluppo di applicazioni parallele che richiedono di sfruttare a pieno la potenza dei calcolatori presenti al giorno di oggi, i quali sono molto eterogenei.

OpenCL velocizza l'esecuzione delle applicazioni eseguendo il codice più dispendioso dal punto di vista computazionale in acceleratori, in questa maniera gli sviluppatori possono scrivere dei kernel in C/C++, che saranno caricati tramite un device compiler per l'esecuzione parallela sui dispositivi di accelerazione.

Un applicazione OpenCL è divisa in due parti:

- **host:** questa parte è scritta in un linguaggio come C o C++, e compilata con i compilatori tradizionali per essere eseguita sulla CPU del calcolatore host.
- **device:** è la parte che può essere compilata "on the fly" (ovvero tramite l'utilizzo di chiamate speciali dell'API a tempo di esecuzione), oppure si può compilare prima che l'applicazione vada in esecuzione così da rendere portabile il binario generato in una rappresentazione intermedia chiamata SPIR-V di Khronos.

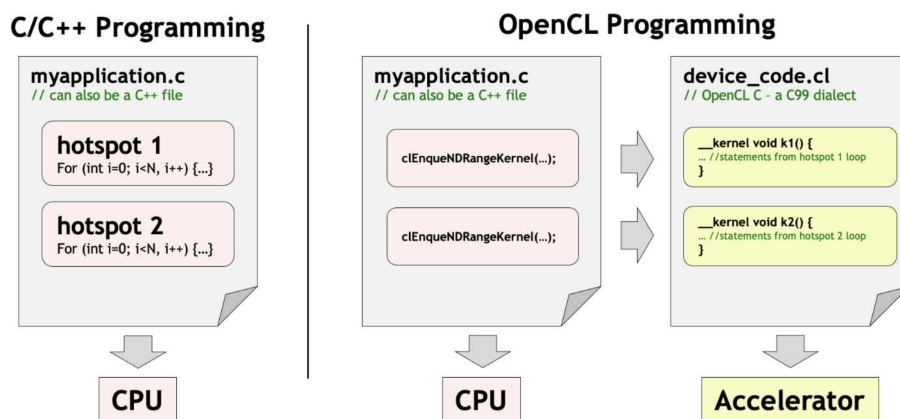


Figura 2.4: Traditional vs OpenCL programming paradigm

2.5 C++/C

I linguaggi di programmazione C e C++ sono due dei linguaggi più influenti e più utilizzati nella storia dell'informatica. Sono noti per la loro potenza, versatilità e velocità e sono utilizzati in una grande varietà di applicazioni, dai sistemi operativi al mondo del software per applicazioni generali.

Di seguito un'introduzione a entrambi:

Linguaggio C:

- È stato sviluppato negli anni 70 da Dennis Ritchie, ed è stato uno dei primi linguaggi ad alto livello della storia dell'informatica.
- È noto per la sua semplicità ed efficienza nell'accesso diretto alla memoria, ed è apprezzato per la possibilità di scrivere codice altamente ottimizzato.
- È ampiamente utilizzato per lo sviluppo di applicazioni embedded, sistemi operativi, compilatori e molte altre applicazioni a basso livello.

Linguaggio C++:

- C++ è una versione estesa di C sviluppata negli anni '80 da Bjarne Stroustrup, tra le varie feature che aggiunge, c'è il concetto di programmazione orientata agli oggetti (OOP), il quale consente la creazione di software più strutturato e modulare rispetto al linguaggio C.
- È ampiamente utilizzato nell'industria del software, nei videogiochi, nella elaborazione di immagini e molto altro.
- Dispone di una libreria standard che presenta molti strumenti per lo sviluppo software.

Entrambi sono linguaggi popolari, ma la scelta tra i due spesso dipende sia dall'applicazione specifica che dalle preferenze personali. Tutti e due offrono un grado alto di controllo sulla macchina, ma C++ aggiunge il paradigma ad oggetti e una sintassi diversa per affrontare problemi complessi in modo più efficiente.

Abbiamo scelto di usare i linguaggi C e C++ per due motivi, in primo luogo perché rientrano tra la lista di linguaggi supportati dall'ambiente Vitis e dal framework OpenCL, inoltre la mia familiarità con questi ha facilitato lo sviluppo del progetto.

Capitolo 3

Progetto Logico

In questo capitolo viene fornita una panoramica generale sul funzionamento del progetto e del comportamento delle sue parti principali. Inoltre viene data un'introduzione sull'estensibilità e la personalizzazione sia dell'interprete che della configurazione della scheda FPGA.

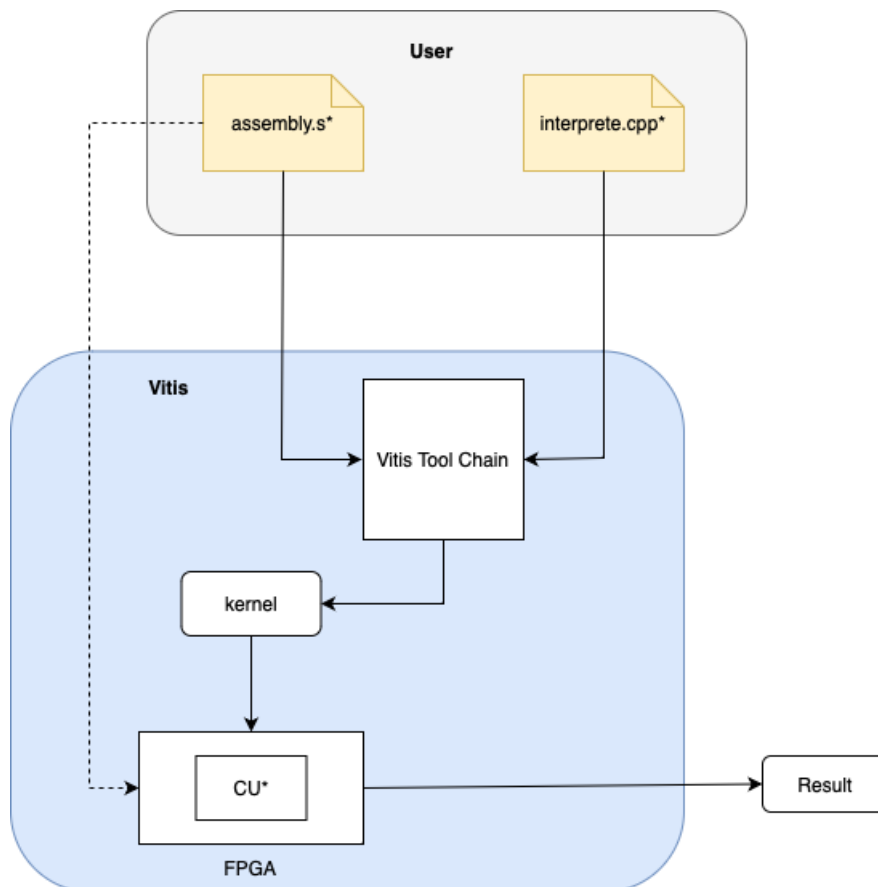


Figura 3.1: Schema Generale

Come si può osservare dal grafico 3.1, il progetto si basa sull'utilizzo un interprete scritto in un linguaggio di alto livello. Questo interprete attraverso la toolchain di Vitis viene trasformato in un kernel eseguibile all'interno dell'FPGA, di cui possono essere istanziate diverse copie (Control Unit), al fine di eseguire programmi anche diversi tra loro.

Insieme alla configurazione dell'interprete è necessario fornire un file scritto nel linguaggio assembly usato dall'interprete, il quale contiene le istruzioni che saranno eseguite dall'acceleratore, ovvero il programma da eseguire sul softcore. Durante l'esecuzione del Kernel, queste istruzioni presenti nel file assembler vengono caricate nella memoria allocata per il softcore sulla FPGA.

Dopo aver eseguito i calcoli sulla FPGA si otterranno i risultati desiderati. Questi risultati saranno estratti dalla memoria del softcore nella FPGA, e successivamente si potranno osservare nella memoria della macchina host.

Come si può notare dal grafico, questo progetto offre una grande possibilità di configurazione, con la capacità di effettuare modifiche nelle seguenti aree:

- **Interprete:** essendo un interprete scritto in linguaggio di alto livello, offre un alto grado di personalizzazione. È possibile specializzarlo scegliendo solo il set di istruzioni più adatto per l'applicazione specifica, oppure estenderlo per interpretare un insieme più grande di istruzioni assembly, o persino cambiare totalmente l'architettura interpretata.
- **Control Unit:** durante il processo della compilazione del kernel, è possibile scegliere quante istanze (CU) allocare all'interno della FPGA. Questo, permette di avere più "core logici" all'interno della scheda FPGA, da gestire in base alle esigenze dell'applicazione.
- **Assembly.s:** questo file, a patto che rispetti il set di istruzioni dell'interprete sviluppato, offre un grande libertà nella scrittura del codice, può contenere qualsiasi flusso di istruzioni consentito dall'interprete. Inoltre è possibile usare file assembly diversi per ciascuna delle CU, così da aver più core che eseguono flussi di istruzioni diverse.

Inoltre (come vedremo nel capitolo Implementazione, Cap. 4), l'interprete funziona con una memoria dati, e una memoria per i registri, le quali sono altamente anch'esse configurabili in modo da poter soddisfare esigenze diverse.

3.1 Funzionamento

Di seguito un grafico sul funzionamento generale delle componenti del progetto.

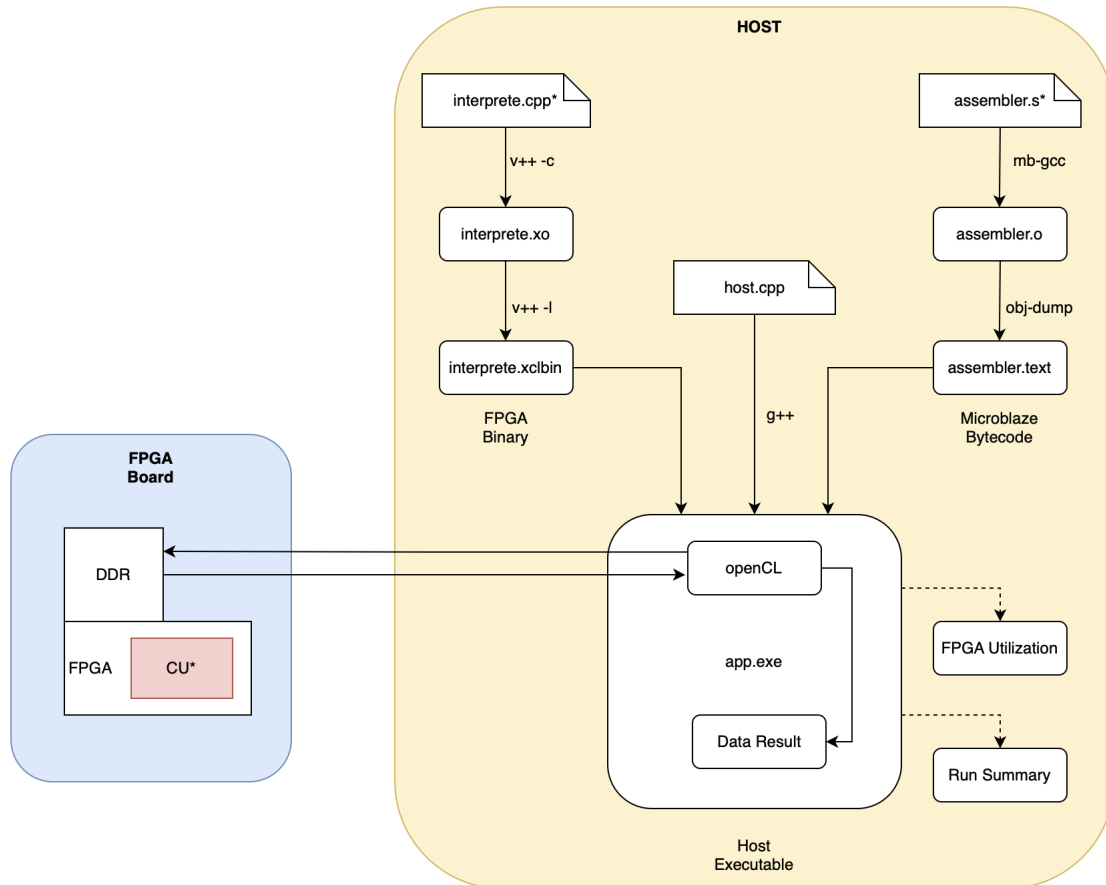


Figura 3.2: Grafico Funzionamento

Nel lato hardware del progetto, l'attività si svolge su una macchina host dove in cui è installata la scheda FPGA (la FPGA Board a sinistra in fig. 3.3). Le componenti software del progetto (a sinistra in fig. 3.3) sono suddivise principalmente in tre parti:

- **FPGA Binary**, questo file contiene il Bit Stream FPGA, il quale è l'implementazione hardware del kernel definito nel file `interprete.cpp`, questo kernel verrà eseguito successivamente dal chip della scheda FPGA.
- **Microblaze Bytecode**, il file contenente il bytecode generato dal compilatore `mb-gcc`, utilizzando le istruzioni scritte in linguaggio assembly Microblaze, il quale sarà interpretato dal kernel presente nella FPGA.
- **Host Executable**, questo eseguibile compilato tramite `g++`, viene eseguito sulla macchina host, ed è responsabile di effettuare le chiamate API di OpenCL, istanziare e caricare i dati nella memoria dell'FPGA (DDR), e gestire i risultati ottenuti dall'interpretazione delle istruzioni assembly.

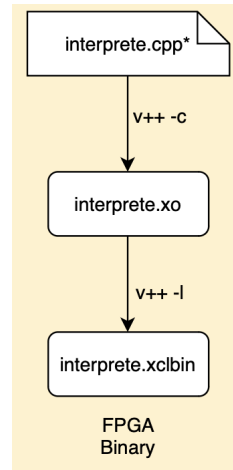


Figura 3.3: Flusso di Compilazione Interpretare

Per la generazione del Bit Stream, partiamo da un file chiamato **interprete.cpp**, dove è definita la funzione principale che verrà eseguita nella scheda FPGA.

All'interno di questo kernel è implementato l'intero interprete del processore Microblaze, il quale legge i dati dalla memoria globale della FPGA, ovvero la memoria dati, la memoria delle istruzioni da interpretare (precedentemente caricate dalla parte host), e la memoria dei registri. L'Interprete successivamente esegue le istruzioni e trasferisce la sua memoria dati nuovamente nella memoria globale della FPGA.

La creazione di questo file inizia da un file con all'interno una funzione `extern C`, il quale viene compilato tramite il compilatore fornito dalla toolchain di Vitis chiamato `v++`, con l'aggiunta della flag `-c`, la quale specifica di compilare il codice sorgente in un file `.xo`, il quale contiene tutti i dati necessari per la successiva generazione del bitstream. Notare che questo processo, che traduce il codice di alto livello del kernel nel linguaggio RTL, richiede un tempo dell'ordine dei minuti o anche secondi, quindi relativamente breve.

Successivamente si compila usando nuovamente `v++` utilizzando la flag `-l`, così da effettuare il linking del kernel compilato dal file `.xo`, con la piattaforma target, in modo da generare il bitstream che è contenuto nel file `.xclbin`, che contiene tutti i dati necessari per configurare l'FPGA con un kernel che implementa l'interprete. È da notare che la durata di questa fase può variare da minuti a ore, in base alla complessità e alla quantità di codice coinvolto. La durata di questo passaggio può variare da minuti a ore di tempo, a seconda della complessità e quantità di codice.

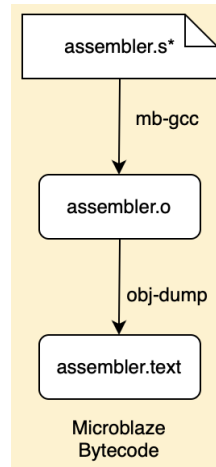


Figura 3.4: Flusso Assembler

Per la creazione del bytecode si inizia partendo da un file scritto in linguaggio assembly (vedi fig. 3.4).

Successivamente usando il compilatore fornito dalla toolchain di Xilinx chiamato `mb-gcc`, questo file viene compilato in un file "oggetto" `.o`. Questo file contiene il risultato della compilazione insieme ad altri meta dati aggiunti dal compilatore stesso.

Successivamente tramite l'utilizzo del tool chiamato `obj-dump` estraiamo la parte `.text` dal file. Questa sezione contiene le istruzioni assembly tradotte dal compilatore e convertite in uno stream di byte.

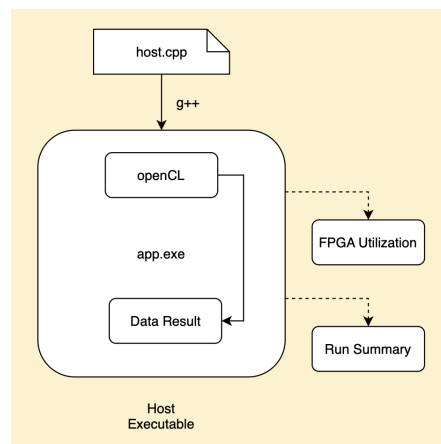


Figura 3.5: Flusso Host

Un'ultima parte del progetto riguarda la compilazione dell'eseguibile che sarà eseguito sulla CPU della macchina host (vedi fig.3.5). Questo file svolge un ruolo cruciale nella gestione della FPGA, compresa l'inizializzazione della memoria esterna dove verranno presi i dati, che includono i registri, la memoria dati e memoria delle istruzioni.

L'eseguibile è responsabile inoltre di caricare (tramite OpenCL) i kernel precedentemente compilati all'interno della scheda FPGA e successivamente di avviare il processo di esecuzione dell'interprete e verifica i risultati ottenuti dall'elaborazione.

Capitolo 4

Implementazione

In questo capitolo spiegheremo in dettaglio l'implementazione delle componenti del progetto. Cominceremo con un'analisi del Interprete del SoftCore e di tutte le sue componenti, per poi spiegare i cambiamenti richiesti per far funzionare il codice dentro un kernel compilabile per una FPGA. Successivamente descriveremo in dettaglio l'interfaccia lato host, e come si svolge il processo di compilazione del progetto. Esploreremo una versione dell'interprete con i registri in virgola Mobile (floating-point). Verrà analizzato come l'interfaccia host cambia con l'istanziamento di più Control Unit. Si conclude con una analisi della versione dell'interprete progettata per emulare una GPU senza controllore SIMD.

4.1 Interprete Softcore

In questa sezione, esploreremo le componenti dell'interprete. È necessario sottolineare che questa è la versione progettata per la compilazione ed esecuzione su una qualsiasi macchina host e scritta interamente nel linguaggio C, le modifiche necessarie per l'esecuzione sulla FPGA saranno dettagliate nella prossima sezione [4.2](#).

Il codice completo relativo alla sezione seguente è presente nell'appendice (ref a `cpu.c` e `cpu.h`)

Per lo sviluppo di questo interprete è stata seguita la documentazione ufficiale del Softcore Microblaze [\[3\]](#).

Nonostante questo interprete non replichi il funzionamento hardware effettivo del processore Microblaze, comunque rispetta il fondamentale paradigma di qualsiasi processore. Questo si basa sull'utilizzo di registri per mantenere lo stato della CPU e portare avanti la computazione e su l'uso di una memoria per immagazzinare e recuperare i dati.

È importante evidenziare che nella implementazione seguente si è optato per l'utilizzo di bit field per le strutture dati, così da migliorare la chiarezza e leggibilità delle dimensioni dei singoli campi, mantenendo al fedeltà alla documentazione.

Registri:

Cominciamo illustrando la struttura dei registri, partendo dalla loro implementazione:

```
1 struct Registers
2 {
3     int32_t r[32];
4     int16_t im;
5     bool c : 1;
6     int32_t pc;
7 };
```

Come è possibile vedere dalla implementazione, il processore dell'interprete è dotato dei seguenti registri:

- **r**: un array da 32 registri interi a 32 bit.
- **im**: ovvero un registro da 16 bit il quale viene usato dalla istruzione `imm` ([4.1](#)) per estendere l'immediato precedentemente istanziato da 16 a 32 bit.
- **c**: ovvero una flag da 1 bit il quale segnala il caso in cui l'istruzione precedente ha generato un overflow.

- **pc**: Program Counter, un registro che indica la prossima istruzione da eseguire.

Memoria:

```

1 struct Memory
2 {
3     int32_t *data;
4     int32_t size;
5 };

```

La memoria è dotata di un array di interi a 32 bit, con la relativa **size** per indicarne la dimensione.

Istruzioni:

L'interprete adotta lo standard di istruzioni RISC, le quali sono caratterizzate tutte da una lunghezza di 32 bit. Esistono due tipi di istruzioni, differenziate in base alla loro natura:

- Type A:



Figura 4.1: Type A instruction

- Type B:



Figura 4.2: Type B instruction

Dove le componenti sono:

- **Opcode**: 6 bit per distinguere il tipo di istruzione.
- **rd**: 5 bit per identificare il registro nel quale memorizzare il risultato dell'istruzione.
- **ra** e **rb**: ciascuno da 5 bit come nel caso di **rd**, per identificare i due registri dove nel caso di una istruzione di tipo A, vengono utilizzati per eseguire l'operazione richiesta dall'istruzione.

- **imm**: costituito **16 bit**, è presente solo nelle istruzioni di tipo B e viene utilizzato come fosse il registro **rb** nel caso delle istruzioni di tipo A. Questi bit identificano un valore immediato inserito direttamente nelle istruzioni assembler. Nel caso l'istruzione sia stata preceduta da una istruzione **imm** (4.1), viene esteso a 32 bit.

Di seguito l'implementazione della struttura che rappresenta le istruzioni:

```

1 struct Instruction
2 {
3     int8_t type : 1; /* 1 type A, 0 type B */
4     int8_t opcode : 6;
5     int8_t rd : 5;
6     int8_t ra : 5;
7     int8_t rb : 5;
8     int32_t im : 32;
9 };

```

L'interprete acquisisce le istruzioni da un file `.text` (sez 4.4). Queste istruzioni vengono caricate nell'interprete tramite la funzione (?? ref apice) sottoforma di un array di puntatori a 4 interi di 8 bit. Per elaborare le istruzioni l'interprete effettua il parsing di tali array per caricarli nella struttura 4.1 tramite la seguente funzione:

```

1 struct Instruction *parse_instruction(int8_t *instr, int8_t
   type, struct Instruction *res, int16_t *im)
2 {
3     if (type) /* Type A */
4     {
5         res->type = type;
6         res->rd = (instr[0] << 3) + ((instr[1] >> 5) & 0b00000111);
7         res->ra = instr[1] & 0b00011111;
8         res->rb = (instr[2] >> 3) & 0b00011111;
9     }
10    else /* Type B */
11    {
12        res->type = type;
13        res->rd = ((instr[0] << 3) & 0b00011000) + ((instr[1] >>
14        5) & 0b00000111);
15        res->ra = instr[1] & 0b00011111;
16        int16_t n = instr[2];
17        n = (n << 8) + (((int16_t)instr[3]) & 0b0000000011111111);
18
19        if (*im) /* imm instruction before */
20        {
21            res->im = (*im << 16) + ((int32_t)n & 0
22            b00000000000000001111111111111111);
23            *im = 0;

```

```
22     }
23     else
24         res->im = (int32_t)n;
25     }
26     return res;
27 }
```

Listing 4.1: Parse Instruction

Questa funzione accetta come parametri l'array della singola istruzione, il tipo della istruzione determinato a priori in base all'opcode, la struttura delle istruzioni da restituire dopo il parsing, e l'intero `im` (ovvero il registro in fig. 4.1). Nel caso l'istruzione sia preceduta da una istruzione `imm` (reference a `imm`), come indicato nell'implementazione, il parametro viene usato per estendere l'immediato presente nella struttura a 32 bit. Questo viene fatto utilizzando i 16 bit meno significativi dei bit dell'istruzione corrente, e i restanti bit più significativi dalla istruzione precedente `imm`. Si può notare come nel caso ci sia una istruzione di tipo A si rispetta la forma specificata in fig. 4.1, mentre nel caso di tipo B, si usa la specifica della fig. 5.2.

Prima di spiegare come alcune delle singole istruzioni sono state implementate, l'interprete al suo interno contiene delle funzioni di supporto per facilitare la scrittura di ogni singola implementazione di istruzione.

La prima funzione è la seguente:

```
1 void update_PC(struct Registers *reg, int32_t n, bool delay)
2 {
3     if (!delay)
4         reg->pc = reg->pc + n / 4;
5 }
```

Listing 4.2: Update PC

Questa funzione viene utilizzata per aggiornare lo stato del registro del Program Counter. Per facilità viene trattato come un numero intero. Tuttavia nella realtà e anche per il compilatore, per avanzare di un'istruzione (dato che ogni istruzione occupa 4 byte) dividiamo il valore di 4. Questo permette di mantenere coerenza tra la rappresentazione del PC register nell'interprete e i comandi del compilatore.

Inoltre la CPU dispone istruzioni di salto che possono includere un flag di Delay Slot ([8]). In caso questo flag di Delay Slot sia attivo, indipendentemente dal risultato del controllo dell'istruzione di salto, viene comunque eseguita l'istruzione successiva a quella in corso, a patto che questa non modifichi il Program Counter (come specificato nella documentazione [3]). Tramite l'utilizzo di questo parametro nella funzione `update_PC` manteniamo una corretta esecuzione del flusso del codice.

Un'altra funzione di supporto all'esecuzione dell'interprete è la seguente:

```
1 int8_t conv_reg(int8_t n)
2 {
3     return n & 0b00011111;
4 }
```

Questa funzione viene usata nel caso in cui il valore presente nella `struct Instruction` 4.1, deve essere usato per operazioni bit a bit.

Queste operazioni potrebbero non essere eseguite correttamente senza questa specifica conversione. Ciò è dovuto al fatto che qualsiasi operatore del linguaggio C, quando applicato come nel nostro caso a un intero `int8_t`, il quale in realtà è un bit field da 5 bit (come si nota in fig. 4.1), utilizza tutti gli 8 bit estendendo il nostro campo con dei bit non appropriati da 5 a 8. Questo può portare a risultati inaspettati durante le operazioni.

Per risolvere questo problema, la funzione `conv_reg` esegue una conversione, impostando correttamente i bit mancanti nel nostro campo. Questo assicura un utilizzo corretto degli operatori del linguaggio C senza gli errori derivanti da un estensione errata dei bit.

Come si può leggere nella documentazione [3], l'operazione fondamentale della CPU consiste nella somma di due registri. Questa operazione costituisce la base per gran parte delle istruzioni ed è l'unica che può causare overflow. Per gestire questo aspetto è stata implementata la seguente funzione:

```
1 int32_t add_Check_Overflow(int32_t a, int32_t b, bool *c)
2 {
3     int64_t res = (int64_t)a + (int64_t)b; /* only last 32 bit
4     */
5     if (res > INT32_MAX)
6     {
7         *c = true;
8         res = INT32_MAX;
9     }
10    else if (res < INT32_MIN) /* underflow */
11    {
12        *c = true;
13        res = INT32_MIN;
14    }
15    else
16    {
17        *c = false;
18    }
19
20    return (int32_t)res;
21 }
```


Questa funzione accetta due interi a 32 bit, e il booleano `c`, il quale rappresenta il flag di carry dei registri, che viene settato in base a se si è o no verificato l'Overflow. Questi due interi vengono convertiti a 64 bit per effettuare l'addizione, per poi controllare il risultato per vedere se sfiora il massimo valore degli interi da 32 bit oppure no. Nel caso di questo interprete viene anche gestito l'Underflow.

La base dell'interprete stesso è una funzione chiamata `run_instruction`. All'interno di questa funzione è presente uno switch, il quale in base all'opcode situato nei primi 6 bit dell'istruzione, seleziona il caso appropriato per eseguire l'operazione corrispondente.

L'implementazione della funzione è la seguente:

```

1 void run_instruction(int8_t *instruction,
2                     struct Memory *data,
3                     struct Registers *reg,
4                     int8_t **instructions,
5                     bool delay)
6 {
7     struct Instruction *instr = malloc(sizeof(struct Instruction
8     ));
9     bool carry = 0; //carry
10    op_code = (instruction[0] >> 2) & 0b00111111;
11    instr->opcode = op_code;
12    int32_t delayed_instruction, addr;
13    int8_t branch_type, is_delayed, is_absolute, is_link;
14    switch (op_code) {
15        ...
16    }
17 }
```

Listing 4.3: Run Instruction

Questa funzione accetta come parametri l'array della singola istruzione, lo stato corrente della memoria e dei registri, l'array contenente tutte le istruzioni (che viene utilizzato nelle istruzioni con delay per effettuare una ricorsione, come nella fig. 4.4), e un booleano Delay Slot. Quando viene eseguita un'istruzione con delay, il parametro impedisce l'aggiornamento del Program Counter nella funzione `update_pc`. (come visto nella fig. 4.2)

All'inizio della funzione viene creata un'istanza che rappresenta le istruzioni dopo il parsing della funzione `parse_instruction` (fig. 4.1). All'interno di questa struttura, il parametro `op_code`, viene inizializzato per motivi di facilità, poiché è utilizzato immediatamente. Successivamente vengono create le istanze delle variabili utilizzate all'interno dei vari case dello switch, le quali verranno spiegate successivamente.

Segue una spiegazione dei vari casi dello switch. Per evitare di ripetizioni, verranno

no illustrate solo le parti principali, poiché la maggior parte degli altri casi presentano solo delle variazioni minori.

La prima funzione dello switch è la `ADD` ed è implementata come segue:

```
1  case 0x0 :
2  { /* ADD 000000 */
3      instr = parse_instruction(instruction, TYPE_A, instr, &reg
        ->im);
4      reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra],
        reg->r[instr->rb], &carry);
5      update_PC(reg, 4, delay);
6      break;
7  }
```

Questo caso rappresenta in gran parte la struttura di tutte le istruzioni. Notare che i casi sono abbinati agli opcode scritti in esadecimale per una questione di leggibilità. L'esecuzione inizia chiamando la funzione per il parsing dell'istruzione, con i parametri appropriati. Secondo la documentazione, l'istruzione `ADD` è una di tipo `A`, per cui si chiama con la variabile `TYPE_A` definita all'inizio dell'interprete. Successivamente per eseguire l'operazione effettiva, viene utilizzata la funzione di supporto `add_Check_Overflow`, che assicura la gestione dell'overflow e dell'eventuale bit di carry.

È importante notare che, nel caso dell'istruzione `ADD`, la CPU Microblaze non tiene conto del bit di carry né per fare la somma e né per salvarlo in caso di carry effettivo. Possiamo osservare che il funzionamento effettivo consiste nell'eseguire la somma del contenuto del registro `ra` con il registro `rb`, e successivamente di memorizzare il risultato nel registro `rd`. Si osserva come al termine dell'esecuzione, venga effettuato l'aggiornamento del Program Counter per puntare alla istruzione successiva.

Per gestire questa situazione, è utile osservare l'implementazione che segue dell'istruzione `ADDCK`:

```
1  case 0x6:
2  { /* ADDCK 000110 */
3      instr = parse_instruction(instruction, TYPE_A, instr, &reg
        ->im);
4      reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra],
        reg->r[instr->rb] + reg->c, &carry);
5      reg->c = carry;
6      update_PC(reg, 4, delay);
7      break;
8  }
```

Possiamo notare come la struttura del rimane invariata rispetto alla istruzione. Co-

me anche il nome dell'istruzione implica, questa variante dell'istruzione ADD, tramite il flag C (nel nome della istruzione) tiene conto del bit di carry precedentemente salvato, per effettuare l'operazione di somma, e con il flag K ovvero "keep", implica il salvataggio dell'eventuale bit di carry nello stato dei registri generato dall'operazione.

Di seguito l'implementazione dell'istruzione RSUB:

```

1  case 0x1:
2  { /* RSUB 000001 */
3      instr = parse_instruction(instruction, TYPE_A, instr, &reg
        ->im);
4      reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb],
        add_Check_Overflow(~reg->r[instr->ra], 1, &carry), &carry);
5      update_PC(reg, 4, delay);
6      break;
7  }
```

Si noti una differenza fondamentale rispetto alla istruzione ADD: qui viene eseguita una somma tra il contenuto del registro `rb` e il not del contenuto del registro `ra` sommato a 1. Ovvero la sottrazione è implementata come somma del complemento a due del secondo operando. Questo approccio per effettuare la sottrazione è stato adottato per rispettare il funzionamento specificato nella documentazione [3]. Nonostante questo cambiamento il paradigma di esecuzione rimane invariato. Da notare che anche questa istruzione presenta le varianti RSUBC, RSUBK e RSUBCK.

Sia l'istruzione ADD che l'istruzione RSUB presentano le rispettive varianti ADDI e RSUBI per gestire il caso dei valori immediati, entrambe con le varianti con i vari flag per il bit di carry presenti. Di seguito, l'implementazione dell'istruzione ADDI:

```

1  case 0x8:
2  { /* ADDI 001000 */
3      instr = parse_instruction(instruction, TYPE_B, instr, &reg
        ->im);
4      reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r[
        instr->ra], &carry);
5      update_PC(reg, 4, delay);
6      break;
7  }
```

Possiamo notare pur essendo la stessa istruzione, al posto del registro `rb` è presente il valore immediato `instr->im`, il quale viene utilizzato come operando per l'operazione di somma. Per il resto il flusso dell'esecuzione rimane invariata.

L'interprete presenta la possibilità di eseguire operazioni bit a bit tramite le istruzioni offerte dall'architettura RISC, tra cui AND, OR, SRA, XOR, e così via. Anche queste presentano lo stesso paradigma di esecuzione delle istruzioni precedenti, come

è possibile osservare dall'implementazione seguente:

```

1 case 0x22:
2 { /* XOR 100010 */
3     instr = parse_instruction(instruction, TYPE_A, instr, &reg
4     ->im);
5     reg->r[instr->rd] = reg->r[instr->ra] ^ reg->r[instr->rb];
6     update_PC(reg, 4, delay);
7     break;
8 }

```

Nel caso in cui nel linguaggio assembly venga inserito un immediato che superi i 16 bit di grandezza, ovvero la dimensione riservata ai valori immediati nelle istruzioni di tipo B (fig. 5.2), il compilatore aggiunge, dopo il processo di compilazione, l'istruzione imm. Questa istruzione estende l'immediato dell'istruzione che la posticipa a 32 bit, nel modo specificato nella spiegazione della funzione `parse_instruction` (fig. 4.1). L'implementazione di tale istruzione è la seguente:

```

1 case 0x2C:
2 { /* IMM 101100 */
3     instr = parse_instruction(instruction, TYPE_B, instr, &reg
4     ->im);
5     reg->im = instr->im;
6     update_PC(reg, 4, delay);
7     break;
8 }

```

Le uniche istruzioni utilizzate da questo interprete per gestire la memoria dati sono LW e SW, entrambe con le varianti LWI e SWI per gestire il caso degli immediati. Segue la loro implementazione:

```

1 case 0x32:
2 { /* LW 110010 */
3     instr = parse_instruction(instruction, TYPE_A, instr, &reg
4     ->im);
5     addr = (uint32_t)(reg->r[instr->ra] + reg->r[instr->rb]);
6     reg->r[instr->rd] = data->data[addr];
7     update_PC(reg, 4, delay);
8     break;
9 }
10 case 0x36:
11 { /* SW 110110 */
12     instr = parse_instruction(instruction, TYPE_A, instr, &reg
13     ->im);
14     addr = (uint32_t)(reg->r[instr->ra] + reg->r[instr->rb]);
15     data->data[addr] = reg->r[instr->rd];

```

```

14     update_PC(reg, 4, delay);
15     break;
16 }

```

Osserviamo che entrambe queste istruzioni effettuano una somma tra il registro `ra` e il registro `rb` per ottenere l'indirizzo di memoria. Nel caso dell'istruzione `LW` salviamo nel registro `rd` il contenuto della memoria all'indirizzo precedentemente calcolato dentro la variabile `addr`, mentre nel caso di `SW` facciamo esattamente il contrario, ossia salviamo in memoria il contenuto del registro `rd` all'indirizzo precedentemente calcolato. Nella versione `SWI` e `LWI` l'unica differenza (oltre al tipo di istruzione, che diventa di tipo `B`), è la seguente riga:

```

1  ...
2  addr = (uint32_t)(reg->r[instr->ra] + instr->im);
3  ...

```

Si nota come al posto del registro `rb` venga usato semplicemente il valore immediato.

Notare come l'indirizzo `addr` viene sempre usato come un `unsigned int` a 32 bit. Questo perché la memoria dati è un array, e l'indicizzazione inizia da 0 e così via in maniera sequenziale, di conseguenza l'utilizzo di questi cast assicura la corretta esecuzione del codice.

Le istruzioni di salto gestite da questo interprete si distinguono, come le altre istruzioni, in due categorie: quelle che gestiscono gli immediati, contrassegnate dal flag `I` nel nome, e quelle senza immediati, ovvero con l'utilizzo di registri normali.

Di seguito la loro implementazione:

```

1  case 0x27:
2  { /* BEQ BGE BGT BLE BLT BNE 100111 */
3      instr = parse_instruction(instruction, TYPE_A, instr, &reg
4          ->im);
5      delayed_instruction = reg->pc + 1; /* prossima istruzione
6          da eseguire in caso di delay*/
7      branch_type = conv_reg(instr->rd) & 0b00001111;
8      is_delayed = conv_reg(instr->rd) & 0b00010000;
9
10     if ((branch_type == 0x0 && reg->r[instr->ra] == 0x0) || /*
11         BEQ D0000 */
12         (branch_type == 0x5 && reg->r[instr->ra] >= 0x0) || /* BGE
13         D0101 */
14         (branch_type == 0x4 && reg->r[instr->ra] > 0x0) || /* BGT
15         D0100 */
16         (branch_type == 0x3 && reg->r[instr->ra] <= 0x0) || /* BLE
17         D0011 */
18         (branch_type == 0x2 && reg->r[instr->ra] < 0x0) || /* BLT
19         D0010 */

```

```
13     (branch_type == 0x1 && reg->r[instr->ra] != 0x0))    /* BNE
D0001 */
14         update_PC(reg, reg->r[instr->rb], delay);
15     else
16         update_PC(reg, 4, delay);
17
18     if (is_delayed == 0x10) /* delayed slot */
19         run_instruction(instructions[delayed_instruction], data,
20                         reg, instructions, true);
21 break;
}
```

Listing 4.4: Istruzioni Branch

Si nota come l'opcode sia lo stesso per tutte le istruzioni di branch (come specificato nella documentazione [3]). Il tipo specifico di ciascuna lo si ricava dai 4 bit meno significativi del registro `rd` (come specificato nella documentazione [3]). Possiamo vedere come a seconda di quale sia il tipo di branch da eseguire, specificato nella variabile `branch_type`, andiamo ad eseguire il controllo appropriato del contenuto del registro `ra`, per poi aggiornare il valore del pc register con il valore contenuto nel registro `rb`. Successivamente, nel caso in cui l'istruzione abbia il flag D nel nome, che imposta a 1 il quinto bit del registro `rd`, si procede con l'esecuzione dell'istruzione successiva a quella del branch, senza tener conto se il controllo è andato a buon fine. Questo viene fatto impostando a true il parametro del delay slot, evitando così di aggiornare il pc register mentre eseguiamo questa istruzione, per poi riprendere il normale flusso del programma.

Fino a questo punto abbiamo esaminato le varie parti dell'interprete. Tuttavia nel contesto di un programma tutto ciò va utilizzato seguendo il flusso di una normale CPU. Per questo per sfruttare la funzione `run_instruction`, si utilizza un ciclo che prosegue fino a quando program counter non raggiunge l'ultima istruzione della lista. Di seguito è riportata l'implementazione di questo ciclo:

```
1 while (reg->pc < instructions_size) {
2     run_instruction(instructions[reg->pc],
3                   data,
4                   reg,
5                   instructions,
6                   false);
7 }
```

Listing 4.5: Ciclo Interprete

Inoltre per far funzionare l'interprete, è necessario disporre del bytecode delle istruzioni assembler generato dal compilatore, (come mostrato nella figura 3.3. Per fare ciò, il metodo usato è il seguente. Partendo da un file `.s`, di seguito viene riportato un esempio di codice assembler:

```
1  .text
2  .align 2
3  .globl main
4  .ent main
5  .type main, @function
6
7  main:
8      addi r2,r0,55
9      addi r3,r0,100
10     cmp    r4,r2,r3
11     addi r5,r0,2147483640
12
13     .end main
```

Successivamente utilizzando del compilatore fornito da Xilinx, `mb-gcc`, il codice viene compilato in un file `.o`.

Possiamo vedere il codice assembly disassemblato tramite il seguente comando:

```
1 mb-objdump -d assembler.o
```

Notare si utilizza il tool `objdump` offerto da Xilin. Il risultato del comando è il seguente:

```
1 00000308 <main>:
2 308: 20400037 addi r2, r0, 55
3 30c: 20600064 addi r3, r0, 100
4 310: 14821801 cmp r4, r2, r3
5 314: b0007fff imm 32767
6 318: 20a0fff8 addi r5, r0, -8
```

Notare come il compilatore aggiunge l'istruzione `imm` per estendere l'immediato della istruzione `addi`. Possiamo notare come questa sezione del file `.o` contiene il bytecode delle istruzioni tradotte dall'assembler a codice macchina. A questo punto è possibile estrarre solamente la sezione `.text` attraverso l'utilizzo del tool `objcopy`.

Questo processo avviene attraverso i comandi seguenti:

```
1 mbgcc -o assembler.o -c assembler.s
2 objcopy -j .text -O binary -I elf32-little assembler.o
   assembler.text
```

4.2 Interprete Versione Kernel

In questa sezione, saranno dettagliate tutte le modifiche apportate all'interprete per renderlo compilabile ed eseguibile sulla scheda FPGA.

Il codice completo relativo alla sezione seguente è presente nell'appendice (ref a vadd.cpp)

Finora, il codice dell'interprete sfrutta ampiamente i paradigmi offerti da un linguaggio di alto livello come il C. Tuttavia va sottolineato che non è possibile utilizzare molti di questi paradigmi per sviluppare un kernel compilabile ed eseguibile su una scheda FPGA. Questo perché la generazione tramite High-Level Synthesis (HLS) messa a disposizione da Vitis, impone restrizioni specifiche che devono essere rispettate per completare il processo di compilazione del kernel e garantire il suo corretto funzionamento.

Durante la conversione dell'interprete al paradigma di funzionamento di un kernel sono stati riscontrati due problemi. In primo luogo, non è permesso l'uso di doppi puntatori, e in secondo luogo, non è possibile utilizzare chiamate ricorsive. Questi vincoli hanno creato due problemi, il primo riguarda la dichiarazione delle istruzioni, le quali come già precedentemente descritto precedentemente (4.1), sono definite come un puntatore di puntatori a interi da 8 bit. Questo è stato risolto dichiarando a priori la dimensione massima del vettore delle istruzioni (ovvero il programma assembler) e quindi utilizzando la definizione di una matrice invece che un doppio puntatore. Come si può vedere nella seguente maniera:

```
1 #define MAX_INSTR 32
2
3 int8_t instr[MAX_INSTR][4];
```

Come si può notare, possiamo gestire fino 32 istruzioni, una quantità che comunque può essere cambiata, e che si è dimostrata più che sufficiente per gli scopi di test di questo interprete.

Il problema delle chiamate ricorsive è stato risolto eliminando le istruzioni che facevano uso del delay slot. Questa decisione è stata presa perché non era un obiettivo primario di questa tesi, e inoltre anche per mantenere l'implementazione relativamente semplice e focalizzata sugli aspetti essenziali del funzionamento di una FPGA. Una possibile soluzione consiste nel aggiungere un parametro nella funzione `run_instruction` (fig. 4.3). Questo parametro viene verificato ad ogni iterazione del ciclo dell'interprete (fig. 4.5), e se attivo, consente l'esecuzione dell'istruzione presente nel Delay Slot, per poi tornare al normale flusso di esecuzione.

Successivamente alla risoluzione di questi problemi, la procedura per sviluppare un kernel richiede la scrittura di una funzione che viene chiamata all'avvio dell'esecu-

zione sulla FPGA. Procediamo con il far vedere la segnatura della funzione:

```

1 void interprete(struct Memory *mem,
2                 struct Registers *reg,
3                 int32_t *out,
4                 ap_uint<32> my_size)
5 {
6     ...
7 }
```

Questa funzione prende come parametri la struttura della memoria, la struttura dei registri, un intero da 32 bit chiamato `out` per restituire il risultato e `my_size` che rappresenta la grandezza delle istruzioni. Questa scelta di utilizzare un singolo valore per verificare il corretto funzionamento è stata fatta poiché ai fini dei test iniziali è stato sufficiente restituire il risultato dell'esecuzione salvato in un singolo registro. Notare come in generale il risultato dell'esecuzione del codice sarà rappresentato nella memoria dati, che alla fine della computazione viene ricopiata nella memoria host, come vedremo nella sez. 4.7. Notare che per ridurre la quantità di parametri, le istruzioni sono state caricate insieme alla memoria `mem`. Notare che questi parametri sono dichiarati e inizializzati dal lato host, questo aspetto verrà spiegato nella sezione 4.3.

Appena all'inizio della funzione `interprete`, sono presenti le seguenti specifiche:

```

1 ...
2 #pragma HLS INTERFACE m_axi port = mem bundle = gmem
3 #pragma HLS INTERFACE m_axi port = reg bundle = gmem
4 #pragma HLS INTERFACE m_axi port = out bundle = gmem
5 #pragma HLS INTERFACE ap_ctrl_hs port = return
6 ...
```

Queste righe di codice sono delle direttive chiamate "HLS pragmas". Queste rappresentano delle specifiche HLS per il compilatore `v++`, utilizzato nella fase di sintesi hardware (4.4). Queste direttive specificano come avviene la creazione delle porte RTL, a partire dagli argomenti della funzione durante la sintesi dell'interfaccia. Queste porte rappresentano il punto di connessione tra l'hardware presente nel chip della FPGA e le strutture esterne, come in questo caso la memoria globale DDR presente nella scheda che ospita il chip dell'acceleratore. Facendo così il tool HLS determina automaticamente i protocolli I/O usati per gestire lo scambio di informazioni tra la parte acceleratore e i gli altri componenti del sistema.

L'utilizzo di queste direttive consente di determinare automaticamente i protocolli I/O utilizzati per la gestione dello scambio di informazioni tra la parte acceleratore e gli altri componenti. In questo caso specifico (come specificato nella documentazione [4]), le prime tre direttive `m_axi` definiscono le interfacce di tipo master AXI4 (Advanced Extensible Interface), per le strutture `mem`, `reg` e `out`. Tutte queste porte

sono tutti blocchi AXI, hanno tutti un bit che dice quando la computazione è terminata, la quarta direttiva `ap_ctrl_hs`), specifica di impostare a 1 il bit del blocco quando si esegue il comando `return`.

Successivamente alle direttive `#pragma` nella funzione è presente il seguente codice:

```

1  ...
2  struct Registers reg_copy;
3  struct Memory mem_copy;
4
5  // Copia dei registri dalla memoria globale alla memoria
   locale
6  for (int i = 0; i < 32; i++)
7      reg_copy.r[i] = reg->r[i];
8  reg_copy.c = reg->c;
9  reg_copy.pc = reg->pc;
10 reg_copy.im = reg->im;
11
12 // Copia della memoria dalla memoria globale alla memoria
   locale
13 for (int i = 0; i < 1024; i++)
14     mem_copy.data[i] = mem->data[i];
15
16 for (int i = 0; i < MAX_INSTR; i++)
17     for (int j = 0; j < 4; j++)
18         mem_copy.instr[i][j] = mem->instr[i][j];
19
20 // Creazione dei puntatori per accedere alle copie locali
21 struct Registers *reg_copy_pointer = &reg_copy;
22 struct Memory *mem_copy_pointer = &mem_copy;
23
24 // Ciclo interprete
25 while (reg_copy_pointer->pc < my_size)
26     run_instruction(mem_copy_pointer->instr[reg_copy_pointer->
   pc],
27         mem_copy_pointer,
28         reg_copy_pointer,
29         mem_copy_pointer->instr,
30         false);
31
32 // Restituzione del risultato al lato host
33 *out = reg_copy_pointer->r[1];
34 }

```

Listing 4.6: Funzione Interprete

Come è possibile notare dall'implementazione, viene eseguita una copia locale dei parametri provenienti dal lato host e caricati sulla DDR attraverso strutture allocate staticamente, così facendo queste strutture dati risulteranno allocate nei blocchi

RAM interni alla FPGA stessa (come sarà dimostrato nella sez. 5.2). Questo per evitare errori generati a tempo di esecuzione del kernel sulla FPGA, Successivamente le copie locali dei parametri, ovvero `reg_copy` e `mem_copy`, vengono usate tramite puntatori al fine di mantenere la leggibilità del codice e la coerenza. Successivamente è presente un ciclo, che come già precedentemente spiegato 4.1 permette di proseguire fino a quando il registro program counter non raggiunge l'ultima istruzione della lista.

Alla fine del ciclo viene eseguita una copia del registro `r1` sul parametro esterno `out` per restituire il valore del risultato della computazione.

Come già specificato precedentemente, il resto del codice rispetto alla versione spiegata nella sezione 4.1, rimane invariato.

4.3 Interfaccia Host

In questa sezione saranno dettagliate le componenti e il loro funzionamento dell'interfaccia usata sulla macchina host per gestire il trasferimento dei dati e la computazione della scheda FPGA.

Il codice completo relativo alla sezione seguente è presente nell'appendice (A.3)

In questa sezione del codice, avvengono tutte le chiamate OpenCl per interfacciarsi con l'acceleratore FPGA. Il funzionamento generale è riassunto nel seguente workflow:

1. Acquisire il file `.xclbin`, risultante dalla sintesi hardware del kernel dell'interprete (4.4) e il file `.text` derivante dall'estrazione del bytecode da un programma scritto in assembler (4.1).
2. Preparare l'ambiente di esecuzione e caricare i dati nella memoria DDR della FPGA.
3. Procedere con l'esecuzione della computazione all'interno dell'acceleratore FPGA.
4. In fine estrarre i risultati dalla memoria DDR e restituirli.

Per capirne il funzionamento iniziamo analizzando la porzione iniziale di codice dove vengono istanziate tutte le variabili necessarie per questa procedura:

```
1 int main(int argc, char **argv)
2 {
3     // Verifica numero argomenti
4     if (argc != 2)
5     {
6         std::cout << "Usage: " << argv[0] << " <XCLBIN File>" <<
            std::endl;
```

```

7     return EXIT_FAILURE;
8 }
9
10 // Dichiarazione Variabili Opencil
11 cl_int err;
12 cl::CommandQueue q;
13 cl::Context context;
14 cl::Kernel krnl;
15
16 bool valid_device = false;
17
18 // Allocazione spazio risultato
19 int32_t *result = (int32_t*)malloc(sizeof(int32_t));
20 *result = 0;
21
22 // Lettura dispositivo
23 auto devices = xcl::get_xil_devices();
24
25 // Creazione binario OpenCL
26 std::string binaryFile = argv[1];
27 auto fileBuf = xcl::read_binary_file(binaryFile);
28 cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()
29     }};
30 ...

```

Si nota che il file `.xclbin` viene passato come argomento alla funzione `main` all'avvio dell'eseguibile. Successivamente, vengono dichiarate le variabili OpenCL necessarie, tra cui:

- `cl_int err`: utilizzata per la verifica dei possibili errori derivanti dalle chiamate di funzione OpenCL.
- `cl::CommandQueue q`: rappresenta una coda dei comandi. In OpenCL una variabile di questo tipo viene utilizzata per inoltrare dei comandi all'acceleratore (FPGA in questo caso) per l'esecuzione di operazioni.
- `cl::Context context`: rappresenta il contesto OpenCL, ovvero un oggetto dove è possibile creare e gestire la memoria dedicata per l'acceleratore, caricare i kernel da eseguire, e altro ancora.
- `cl::Kernel krnl`: rappresenta un kernel OpenCL, ovvero il programma che sarà eseguito dall'acceleratore nel formato binario specifico del dispositivo.

In seguito, osserviamo alcune chiamate di funzione dalla libreria `xcl`, il quale è il runtime driver fornito da Xilinx. In generale il loro funzionamento è il seguente:

- `xcl::get_xil_devices()`: restituisce una lista di dispositivi compatibili con la piattaforma Xilinx.

- `xcl::read_binary_file(binaryFile)` restituisce un buffer a partire dal contenuto del file `.xclbin`.

Successivamente viene creato un oggetto `"cl::Program::Binaries bins"`, il quale rappresenta il contenuto binario di un programma OpenCL.

Di seguito il codice per effettuare la programmazione dei dispositivi di accelerazione:

```

1 ...
2 for (unsigned int i = 0; i < devices.size(); i++)
3 {
4     auto device = devices[i];
5
6     // Creazione della coda di comando e del contesto per i
    device presenti
7     OCL_CHECK(err, context = cl::Context(device, nullptr,
    nullptr, nullptr, &err));
8     OCL_CHECK(err, q = cl::CommandQueue(context, device,
    CL_QUEUE_PROFILING_ENABLE, &err));
9     std::cout << "Trying to program device[" << i << "]: " <<
    device.getInfo<CL_DEVICE_NAME>() << std::endl;
10    cl::Program program(context, {device}, bins, nullptr, &err
    );
11
12    if (err != CL_SUCCESS)
13    {
14        std::cout << "Failed to program device[" << i << "]"
    with xclbin file!\n";
15    }
16    else
17    {
18        std::cout << "Device[" << i << "]: program successful
    !\n";
19
20        //Creazione Kernel
21        OCL_CHECK(err, krnl = cl::Kernel(program, "interprete"
    , &err));
22        valid_device = true;
23        break; // Device valido trovato
24    }
25 }
26 ...

```

Listing 4.7: ricerca dispositivi

Possiamo osservare come per ogni dispositivo (nel nostro caso è sempre stato solo uno, ovvero l'FPGA), vengono creati il contesto e la coda di comando utilizzando le funzioni di supporto di OpenCL. Inoltre viene creato l'oggetto `program`, il quale rappresenta il programma che sarà eseguito nell'FPGA. Una volta creato il programma, viene utilizzato per creare il kernel, nel quale durante la creazione,

specifichiamo il nome della funzione che verrà chiamata all'avvio dell'esecuzione. Notare come eventuali errori vengo gestiti e registrati nella variabile `err`.

Successivamente vengono caricate le istruzioni dal file `.text`, vengono inizializzati memoria e registri dell'interprete, e le istruzioni vengono copiate all'interno della memoria dati. Di seguito è riportato il codice corrispondente:

```

1 ...
2  int32_t instr_size = 0;
3  int8_t **instr_vector = get_instructions_from_file(file, &
4      instr_size);
5
6  auto mysize = instr_size;
7
8  struct Memory *data = (struct Memory *)malloc(sizeof(struct
9      Memory));
10 struct Registers *reg = (struct Registers *)malloc(sizeof(
11     struct Registers));
12
13 // Inizializzazione Registri
14 reg = initialize_registers(reg);
15
16 // Inizializzazione Memoria
17 for (int i = 0; i < 1024; i++)
18     data->data[i] = 0;
19
20 // Copia istruzioni
21 for (int i = 0; i < mysize; i++)
22 {
23     data->instr[i][0] = instr_vector[i][0];
24     data->instr[i][1] = instr_vector[i][1];
25     data->instr[i][2] = instr_vector[i][2];
26     data->instr[i][3] = instr_vector[i][3];
27 }
28 ...

```

Successivamente si trova l'ultima sezione del codice dell'interfaccia host, dove continua la gestione della memoria e inizia la computazione dell'acceleratore, di seguito l'implementazione di questa:

```

1 ...
2 // Allocazione Buffer nella memoria globale della FPGA
3 OCL_CHECK(err, cl::Buffer buffer_out(context,
4     CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(int32_t),
5     result, &err));
6 OCL_CHECK(err, cl::Buffer buffer_data(context,
7     CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
8     Memory), data, &err));
9 OCL_CHECK(err, cl::Buffer buffer_reg(context,
10    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct

```

```

    Registers), reg, &err));
6
7 // Configurazione degli argomenti del kernel
8 OCL_CHECK(err, err = krnl.setArg(0, buffer_data));
9 OCL_CHECK(err, err = krnl.setArg(1, buffer_reg));
10 OCL_CHECK(err, err = krnl.setArg(2, buffer_out));
11 OCL_CHECK(err, err = krnl.setArg(3, mysize));
12
13 // Migrazione dei dati dalla memoria host alla memoria
    globale della FPGA
14 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_data,
    buffer_reg}, 0 /*from host*/));
15
16 // Inizio esecuzione kernel
17 OCL_CHECK(err, err = q.enqueueTask(krnl));
18
19 // Migrazione risultati dall'acceleratore alla memoria host
20 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_out},
    CL_MIGRATE_MEM_OBJECT_HOST));
21
22 // Attesa che il kernel completi l'esecuzione
23 OCL_CHECK(err, err = q.finish());
24
25 std::cout << "\n\n FPGA RESULT " << *result << "\n\n";
26
27 return EXIT_SUCCESS;
28 }

```

Si nota come inizialmente, si creano oggetti di tipo `cl::Buffer`, i quali rappresenta buffer di memoria sulla FPGA. Successivamente, gli argomenti della funzione iniziale presente nel kernel vengono configurati correttamente, in modo da fornire gli input nell'ordine appropriato. Viene quindi utilizzata la funzione `enqueueMigrateMemObjects` per aggiungere alla coda di comandi il comando che migra i dati dai buffer `buffer_data` e `buffer_reg` dalla memoria host alla memoria della FPGA (notare come l'argomento 0 specifica la direzione della migrazione). Successivamente, viene aggiunto alla coda di comando per avviare l'esecuzione del kernel. L'istruzione `q.finish()` è utilizzata per attendere che tutte le operazioni all'interno della coda finiscano prima di proseguire con il resto del codice. Una volta eseguite le operazione il risultato viene stampato e il programma termina.

4.4 Compilazione

Vitis mette a disposizione 3 modi di compilare il progetto tramite il compilatore da loro offerto v++:

- **Software Emulation:** in questa modalità, il kernel è compilato per essere eseguito sulla CPU della macchina host. Questo processo permette di rifinire facilmente il codice attraverso step iterativi di cicli build-and-run.
- **Hardware Emulation:** in questa modalità, il kernel viene compilato nel modello hardware (RTL), il quale viene eseguito in emulatore dedicato. Sebbene il processo di compilazione richieda più tempo, fornisce dettagli più approfonditi sull'esecuzione del kernel, e delle performance.
- **Hardware:** in questo processo il kernel viene compilato nell'hardware model (RTL) e viene implementato direttamente nella FPGA.

La prima fase del processo di compilazione, come indicato nella documentazione [2], consiste nel configurare l'ambiente per eseguire Vitis. Si utilizzano i seguenti comandi:

```
1 source /tools/Xilinx/Vitis/2023.1/settings64.sh
2 source /opt/xilinx/xrt/setup.sh
```

Successivamente è necessario configurare correttamente la seguente variabile di ambiente:

```
1 export PLATFORM_REPO_PATHS=/opt/xilinx/platforms/
   xilinx_u50_gen3x16_xdma_5_202210_1/
```

Questa variabile di ambiente indica il percorso per le piattaforme installate sulla macchina corrente. Possiamo notare come la nostra piattaforma sia la scheda u50.

Successivamente il processo continua tramite i seguenti comandi:

```
1 export XCL_EMULATION_MODE=sw_emu
2 g++ -g -std=c++17 -Wall -O0 src/host.cpp src/xcl2.cpp -o ./app
   .exe -I$XILINX_XRT/include/ -L$XILINX_XRT/lib -
   lxrt_coreutil -pthread -lOpenCL
3 emconfigutil --platform xilinx_u50_gen3x16_xdma_5_202210_1
4 v++ -c -t sw_emu --platform xilinx_u50_gen3x16_xdma_5_202210_1
   -k vadd -I/src src/vadd.cpp -o ./vadd.xo
5 v++ -l -t sw_emu --platform xilinx_u50_gen3x16_xdma_5_202210_1
   ./vadd.xo -o ./vadd.xclbin
```

È necessario far notare come:

- Vediamo come inizialmente impostiamo il metodo di emulazione.
- Il compilatore `g++` (come si può vedere nella fig. 3.3 e offerto da GNU [7]), è utilizzato per compilare la sezione del progetto, che racchiude l'interfaccia lato host spiegata nella sez. 4.3.
- `emconfigutil` genera un file di configurazione che definisce il tipo e la quantità dei dispositivi da emulare per la piattaforma specificata in precedenza.
- `v++ -c`: compila il codice sorgente del kernel, producendo il file `.xo`.
- `v++ -l`: effettua il link del kernel compilato con la piattaforma di destinazione e genera il file `.xclbin`.

Successivamente per avviare il programma (compilato in `sw_emu`) è sufficiente il seguente comando:

```
1 ./app.exe vadd.xclbin
```

Nel caso di voler usare la `hw_emu` è sufficiente sostituire le seguenti righe:

```
1 export XCL_EMULATION_MODE=hw_emu
2 v++ -c -t hw_emu --platform xilinx_u50_gen3x16_xdma_5_202210_1
  -k vadd -I/src src/vadd.cpp -o ./vadd.xo
3 v++ -l -t hw_emu --platform xilinx_u50_gen3x16_xdma_5_202210_1
  ./vadd.xo -o ./vadd.xclbin
```

Notare che il resto dei comandi rimane invariato. Per la compilazione con target l'hardware effettivo non è più richiesta la variabile di ambiente `XCL_EMULATION_MODE`, e le seguenti righe diventano:

```
1 v++ -c -t hw --platform xilinx_u50_gen3x16_xdma_5_202210_1 -k
  vadd -I/src src/vadd.cpp -o ./vadd.xo
2 v++ -l -t hw --platform xilinx_u50_gen3x16_xdma_5_202210_1 ./
  vadd.xo -o ./vadd.xclbin
```

4.5 Interpretare Floating Point

In questa sezione verranno mostrati i cambiamenti apportati al codice dell'interprete spiegato nella sez. 4.2, al fine di poter effettuare calcoli in virgola mobile.

Questi cambiamenti, come spiegato nella sez. 3.1, vengono resi possibili dal fatto che l'interprete è implementato in un linguaggio di alto livello. E' importante notare che abbiamo simulato il funzionamento delle istruzioni finora implementate in virgola mobile, allo scopo di osservare i cambiamenti che questo comportava e cosa sarebbe successo al kernel compilato all'interno della FPGA. Sebbene il processore MicroBlaze disponga già di istruzioni per il calcolo in virgola mobile dei registri, abbiamo scelto di apportare solo cambiamenti "minimali" per restare coerenti con il titolo della tesi, e mantenere un approccio centrato su un "core GPU". Inoltre questo cambiamento della rappresentazione interna in virgola mobile non implica la possibilità di inserire valori floating point direttamente nel linguaggio assembly

Iniziamo con i cambiamenti immediati, i quali includono le seguenti modifiche nel codice:

```
1  struct Registers
2  {
3      float r[32];
4      ...
5  };
6
7  struct Memory
8  {
9      float data[1024];
10     ...
11 };
12
13 struct Instruction
14 {
15     ...
16     float im;
17 };
```

Si nota come sia i registri che la memoria si dichiarano come `float`, rappresentando così numeri con la virgola mobile da 32 bit. Questo permette di eseguire le operazioni aritmetiche offerte dal linguaggio C per i numeri con la virgola.

Nelle istruzioni (`struct Instruction`) notiamo che la rappresentazione dell'immediato è in `float`.

Inoltre va notato che le operazioni bit a bit non sono direttamente supportate con i numeri in virgola mobile, quindi tutte le funzioni che usano questi operatori trattano il dato come un tipo `unsigned int` da 32 bit. Questo approccio garantisce la correttezza del codice in caso di operazioni bit a bit, di seguito un esempio della modifica all'istruzione AND seguendo questo paradigma:

```

1  case 0x21:
2  { /* AND 100001 */
3      instr = parse_instruction(instruction, TYPE_A, instr, &reg
        ->im);
4      reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) &
        f_to_u(reg->r[instr->rb]));
5      update_PC(reg, 4, delay);
6      break;
7  }

```

Possiamo notare come vengono usate le funzioni `f_to_u` e `u_to_f`, segue la loro implementazione:

```

1  uint32_t f_to_u(float n)
2  {
3      return *(uint32_t *)&n;
4  }
5
6  float u_to_f(uint32_t n)
7  {
8      return *(float *)&n;
9  }

```

Queste funzioni effettuano una conversione sull'intera rappresentazione in bit del tipo passato come argomento. Il motivo dell'utilizzo di queste funzioni rispetto ad un normale cast, è che se abbiamo un valore di tipo `float` e quindi un numero in virgola mobile di 32 bit, e vogliamo eseguire un cast a un `uint32_t` (ovvero un `unsigned int` da 32 bit), utilizzando la notazione `(uint32_t)`, non tutti i 32 bit del valore vengono utilizzati, ma si perdono i bit che rappresentano i numeri dopo la virgola. Visto che il nostro obiettivo è reinterpretare da `float` a `uint32_t` e viceversa utilizzando tutti i 32 bit, usiamo queste due funzioni anziché il cast offerto dal linguaggio C.

Inoltre un'altra modifica significativa riguarda la funzione di gestione dell'overflow, di seguito l'implementazione:

```

1  float add_Check_Overflow(float a, float b, bool *c)
2  {
3      double res = (double)a + (double)b; /* only last 32 bit*/
4      if (res > INT32_MAX)
5      {
6          *c = true;
7          res = INT32_MAX;
8      }
9      else if (res < INT32_MIN) /* underflow */
10     {
11

```

```
12     *c = true;
13     res = INT32_MIN;
14 }
15 else
16 {
17     *c = false;
18 }
19
20 return (float)res;
21 }
```

Questa funzione, analogamente alla funzione precedente, usa la rappresentazione successiva più grande di un `float`, ovvero un `double`, per facilitare la possibilità di determinare la presenza di un overflow. Il massimo valore rappresentabile è rimasto il massimo dei numeri `signed int` per una questione di praticità.

Una volta implementati questi cambiamenti si dispone di un interprete che esegue operazioni rappresentando i bit passati attraverso le istruzioni assembler con la virgola mobile.

4.6 Control Unit

In questa sezione, verrà illustrato il processo di creazione di più istanze, utilizzando il codice spiegato nella sez. 4.2. Queste "istanze", verranno eseguite separatamente per ogni Control Unit, le quali come possiamo vedere nella fig. 3.1, risiedono all'interno del chip FPGA, e funzionano come "core", i quali sono capaci di eseguire computazioni parallele, sia della solita istanza del kernel o anche di diverse a seconda della necessità.

Il processo è reso possibile dal file di configurazione `conf.cfg`, il quale passato come argomento al compilatore `v++` durante le fasi di compilazione e di linking, abilita questo tipo di configurazione. Di seguito le modifiche apportate al compilatore:

```
1 v++ -c -t hw --platform xilinx_u50_gen3x16_xdma_5_202210_1 --
    config conf.cfg -k vadd -I/src src/vadd.cpp -o ./vadd.xo
2 v++ -l -t hw --platform xilinx_u50_gen3x16_xdma_5_202210_1 --
    config conf.cfg ./vadd.xo -o ./vadd.xclbin
```

Notiamo che l'unica modifica consiste nell'aggiunta della flag `-config` seguita dal percorso al file `conf.cfg`.

Il contenuto `conf.cfg` del file è il seguente:

```
1 debug=1
2 save-temps=1
3
4 [connectivity]
```

```

5 nk=interprete:4:interprete_1,interprete_2,interprete_3,
  interprete_4
6
7 [profile]
8 data=all:all:all

```

Questo file in sostanza fornisce ulteriori specifiche al compilatore, oltre quelle passate come argomento.

Le prime due righe abilitano la creazione del file di debug e la conservazione dei file temporanei generati durante la fase di compilazione.

La sezione `profile` specifica di salvare tutte le metriche disponibili sulla memoria passata come argomento al kernel.

La sezione `connectivity` specifica quante istanze del kernel creare. Tramite la seguente stringa andiamo a creare 4 istanze del kernel `interprete`, di nome `interprete_1`, `interprete_2` e così via:

```

1 nk=interprete:4:interprete_1,interprete_2,interprete_3,
  interprete_4

```

Una volta creato il file di configurazione e modificati i parametri di compilazione, l'operazione successiva consiste nell'adattare l'interfaccia host in modo che sia in grado di gestire più Control Unit.

Per raggiungere questo obiettivo, è stato necessario evitare di ripetere il codice, creando una funzione per la gestione di tutto il processo di creazione e istanziazione della memoria per un singolo kernel. Prima di vedere l'implementazione di tale funzione, di seguito una modifica all'interno del codice listato nella fig. 4.7, dove si usa un array di kernel dichiarato nella seguente maniera:

```

1 cl::Kernel krnl[KERNEL_NUMBER];
2 int32_t result[KERNEL_NUMBER];

```

Possiamo notare come all'aumentare del numero di Control Unit, aumenta di conseguenza il numero risultati che devono essere estratti dalla FPGA.

All'interno del ciclo di ricerca dispositivi listato in fig. 4.7 viene usato il seguente approccio per configurare i kernel una volta trovato un dispositivo corretto:

```

1     ...
2     else
3     {
4         std::cout << "Device[" << i << "]: program successful!\n";
5         for (int i = 1; i <= KERNEL_NUMBER; i++)
6         {

```

```

7         std::string var = "interprete:{interprete_" + std::
to_string(i) + "}";
8         OCL_CHECK(err, krnl[i - 1] = cl::Kernel(program, var.
c_str(), &err));
9     }
10    valid_device = true;
11    break;
12 }
13 ...

```

Possiamo vedere come si usi un approccio modulare per istanziare i kernel con la forma `nome kernel: nome CU`.

Successivamente, segue l'implementazione della funzione che gestisce il singolo kernel per essere correttamente inserito nella coda di comando:

```

1 void enqueue_task(char const *file,
2                   cl::CommandQueue *q,
3                   cl::Kernel krnl,
4                   cl::Context context,
5                   int32_t *result)
6 {
7     cl_int err;
8     int32_t instr_size = 0;
9     int8_t **vector = get_instructions_from_file(file, &
instr_size);
10    auto mysize = instr_size;
11    struct Memory *data = (struct Memory *)malloc(sizeof(struct
Memory));
12    struct Registers *reg = (struct Registers *)malloc(sizeof(
struct Registers));
13    reg = initialize_registers(reg);
14
15    for (int i = 0; i < 1024; i++)
16    {
17        data->data[i] = 0;
18    }
19
20    for (int i = 0; i < mysize; i++)
21    {
22        data->instr[i][0] = vector[i][0];
23        data->instr[i][1] = vector[i][1];
24        data->instr[i][2] = vector[i][2];
25        data->instr[i][3] = vector[i][3];
26    }
27
28    OCL_CHECK(err, cl::Buffer buffer(context,
CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(int32_t),
result, &err));
29
30    OCL_CHECK(err, cl::Buffer buffer_data(context,

```

```

    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
    Memory), data, &err));
31
32 OCL_CHECK(err, cl::Buffer buffer_reg(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
    Registers), reg, &err));
33
34 OCL_CHECK(err, err = krnl.setArg(0, buffer_data));
35 OCL_CHECK(err, err = krnl.setArg(1, buffer_reg));
36 OCL_CHECK(err, err = krnl.setArg(2, buffer));
37 OCL_CHECK(err, err = krnl.setArg(3, mysize));
38
39 OCL_CHECK(err, err = q->enqueueMigrateMemObjects({
    buffer_data, buffer_reg}, 0 /* 0 means from host*/));
40
41 OCL_CHECK(err, err = q->enqueueTask(krnl));
42
43 OCL_CHECK(err, err = q->enqueueMigrateMemObjects({buffer},
    CL_MIGRATE_MEM_OBJECT_HOST));
44 }

```

Listing 4.8: enqueue task

La funzione accetta come argomenti il nome del file `.text`, che contiene la lista di istruzione dell'interprete, la coda di comando `q`, il kernel precedentemente creato, il contesto e la variabile risultato. Il resto del codice di questa funzione è stato spiegato nella sezione precedente ovvero sez. 4.3.

Successivamente al ciclo di ricerca di dispositivi, abbiamo il seguente codice, dove grazie alla funzione precedentemente spiegata ovvero `enqueue_task` passiamo tutti i comandi in ordine di kernel alla coda di comando, attendiamo che questa finisca il lavoro, e andiamo a stampare i risultati.

```

1 ...
2 char const *file[KERNEL_NUMBER] = {
3     "data.text",
4     "add.text",
5     "bitop.text",
6     "sub.text",
7 };
8
9
10 for (int i = 0; i < KERNEL_NUMBER; i++)
11 {
12     result[i] = 0;
13     enqueue_task(file[i], &q, krnl[i], context, &result[i]);
14 }
15
16 OCL_CHECK(err, err = q.finish());
17

```

```

18  for (int i = 0; i < KERNEL_NUMBER; i++)
19      std::cout << "\nKernel " << i << "  result:" << *result;
20
21  std::cout << "\n";
22  return EXIT_SUCCESS;
23 }

```

In questo modo abbiamo eseguito con successo 4 kernel distinti, ognuno con un file di istruzioni assembler diverso, all'interno della FPGA.

4.7 Simulazione GPU con codice MIMD

In questa sezione illustreremo la configurazione e le modifiche apportate all'interprete del softcore e all'interfaccia host, per simulare una GPU con core indipendenti, senza controllore SIMD. Con questo approccio si vuole mostrare come più istanze/-CU (Control Unit) del softcore all'interno della FPGA possano eseguire computazioni in modo data parallel, ma con codici distinti per ciascuna (quindi MIMD nel senso che sullo stesso dato a posizione diverse vengono effettuati calcoli diversi). In particolare, si potrà eseguire del codice "quasi SIMD", dove la stessa funzione `f` viene applicata su tutti i `v[i]` (il dato in posizione `i` del vettore `v` su cui applicare la funzione), ma con dei branch interni alla funzione `f`, i quali dipendono dal singolo dato `v[i]`.

In sostanza, oltre al fatto che le Control Unit sono diventate 8 invece che 4 come nella sezione 4.6, e che il risultato in questa sezione diventi tutta la memoria dell'interprete invece che restituire come risultato della computazione un intero da 32 bit, i cambiamenti sono i seguenti:

All'interno del kernel la lista di parametri e il modo in cui restituiamo il risultato cambiano come segue:

```

1  void vadd(struct Memory *mem, struct Registers *reg, struct
    Memory *outmem, ap_uint<32> my_size)
2  {
3  ...
4  ...
5      for (int i = 0; i < 1024; i++)
6          outmem->data[i] = mem_copy.data[i];
7  }
8  }

```

Inoltre è cambiata l'interfaccia lato host per gestire questa modifica. Di seguito è riportata l'implementazione:

```

1  ...
2  struct Memory *data_out[KERNEL_NUMBER];

```



```
3
4  for (int i = 0; i < KERNEL_NUMBER; i++)
5  {
6      data_out[i] = (struct Memory *)malloc(sizeof(struct Memory
7      ));
8      enqueue_task(file[i], &q, krnl[i], context, data_out[i]);
9  }
10
11  OCL_CHECK(err, err = q.finish());
12
13  for (int i = 0; i < KERNEL_NUMBER; i++)
14  {
15      print_vector(15, data_out[i], 1);
16      free(data_out[i]);
17  }
18
19  return EXIT_SUCCESS;
20 }
```

Il codice per il resto è rimasto praticamente invariato, con la principale modifica che riguarda gli argomenti di `enqueue_task`. Dove invece di gestire e caricare un intero gestiamo e carichiamo la memoria `data_out` all'interno del kernel. Inoltre al termine della computazione e del trasferimento dei file dei dati di tutti i kernel, vengono stampati i primi 15 elementi della memoria per valutare i risultati dell'esecuzione, e quindi il programma termina.

Capitolo 5

Risultati

In questo capitolo affronteremo il processo di verifica dei risultati per le diverse versioni dell'interprete spiegate nella sez. 4. Seguiranno una sezione sulla verifica del funzionamento di ciascuna versione, una sezione successiva per l'analisi sull'occupazione della FPGA per ogni versione dell'interprete, e infine concluderemo con una vista sull'esecuzione a run time dei vari kernel attraverso gli strumenti di Vitis.

5.1 Verifica Funzionamento

La verifica del funzionamento delle diverse versioni del codice è stata svolta utilizzando dei file in codice assembler per il processore Microblaze. Questi file sono stati eseguiti sull'interprete implementato nella sez. 4.1, eseguito sulla macchina host. Questi risultati sono stati confrontati con i risultati estratti dalle versioni compilate ed eseguite sul FPGA per verificare il corretto funzionamento dell'interprete.

Partiamo mostrando come i risultati sono stati prima verificati sulla macchina host. E' stato creato un file `test.c`, il quale per ogni file assembler di test, avvia l'interprete e confronta i risultati, la funzione utilizzata all'interno del file è la seguente:

```
1 bool test (char* file, int32_t n)
2 {
3     int32_t instructions_size = 0;
4     int8_t **instructions = get_instructions_from_file(file, &
5         instructions_size);
6
7     struct Memory *data = malloc(sizeof(struct Memory));
8     data = initialize_memory(MEMORY_SIZE, data);
9
10    struct Registers *reg = malloc(sizeof(struct Registers));
11    reg = initialize_registers(reg);
12
13    while (reg->pc < instructions_size)
```

```
13         run_instruction(instructions[reg->pc], data, reg,  
14         instructions, false);  
15     if (reg->r[1] == n)  
16         return true;  
17  
18     return false;  
19 }
```

Possiamo notare come la funzione prende in input il nome del file contenente le istruzioni, e un valore intero `n`, ovvero il risultato atteso dalla computazione. Dopo aver inizializzato tutto il necessario per avviare l'interprete sulla macchina host, la funzione avvia il ciclo dell'interprete. Al termine confronta il risultato ottenuto con quello atteso e restituisce un valore booleano.

Questa funzione all'interno del file `test.c` viene utilizzata nel seguente modo:

```
1  int main ()  
2  {  
3      if (test("test/add.text", ADD_RESULT))  
4          printf("ADD test OK\n");  
5      else  
6          printf("ADD test FAIL\n");  
7  
8      if (test("test/sub.text", SUB_RESULT))  
9          printf("SUB test OK\n");  
10     else  
11         printf("SUB test FAIL\n");  
12  
13     if (test("test/bitop.text", BIT_OP_RESULT))  
14         printf("BIT_OP test OK\n");  
15     else  
16         printf("BIT_OP test FAIL\n");  
17  
18     if (test("test/branch.text", BRANCH_RESULT))  
19         printf("BRANCH test OK\n");  
20     else  
21         printf("BRANCH test FAIL\n");  
22  
23     if (test("test/data.text", DATA_RESULT))  
24         printf("DATA test OK\n");  
25     else  
26         printf("DATA test FAIL\n");  
27  
28     return EXIT_SUCCESS;  
29 }
```

I file mostrati di seguito sono disponibili nell'appendice alla sez. [A.5](#). I risultati specificati nei define del codice sono i seguenti:

```
1 #define ADD_RESULT 159
2 #define SUB_RESULT -2147482620
3 #define BIT_OP_RESULT 200
4 #define BRANCH_RESULT 36
5 #define DATA_RESULT 199
```

Listing 5.1: Risultati Aspettati

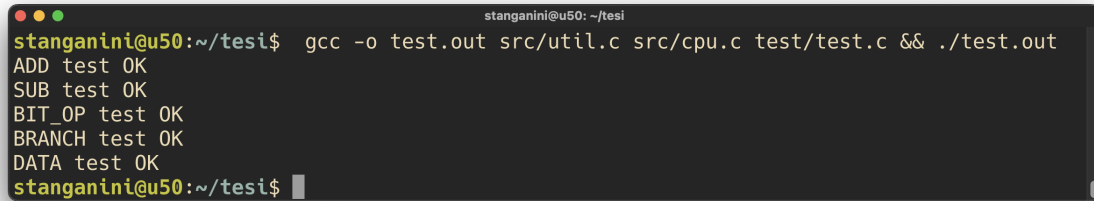
Si noti che questi risultati sono stati calcolati manualmente al fine di avere una verifica corretta:

Segue un esempio per il file `branch.text`:

```
1  .text
2  .align 2
3  .globl main
4  .ent main
5  .type main, @function
6
7  main:
8      addi r3,r0,8
9      addi r2,r0,-8
10     addi r1,r0,0
11     addi r1,r1,1
12     addi r3,r3,-1
13     bge r3,r2
14     nop
15     addi r3,r0,8
16     addi r1,r1,1
17     addi r3,r3,-1
18     bgei r3,-8
19     nop
20     addi r3,r0,8
21     addi r1,r1,1
22     addi r3,r3,-1
23     bgei r3,-8
24     nop
25     addi r3,r0,-8
26     addi r2,r0,-8
27     addi r1,r1,1
28     addi r3,r3,1
29     ble r3,r2 #r1 = 37+9 = 36
30
31     .end main
```

Questo file ha come risultato il valore 36 nel registro `r1`.

Dopo aver compilato ed eseguito l'interprete, è evidente notare come i risultati ottenuti siano coerenti con quelli aspettati. Di seguito l'immagine della stampa dell'esecuzione:



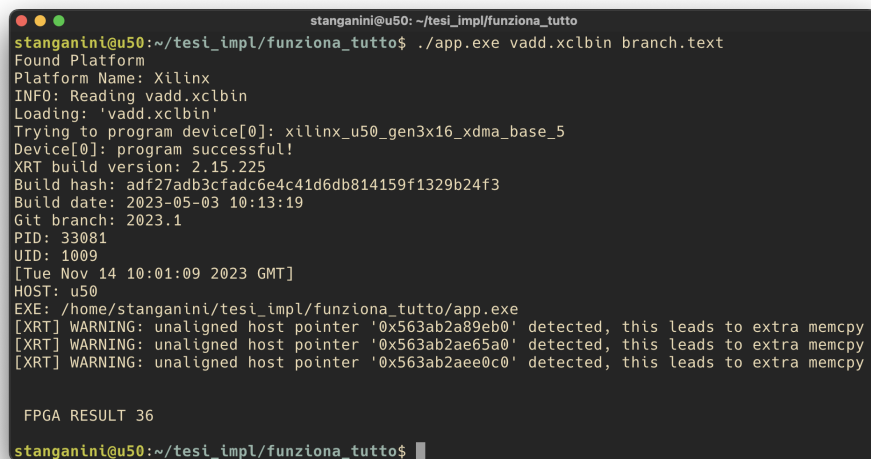
```
stanganini@u50: ~/tesi$ gcc -o test.out src/util.c src/cpu.c test/test.c && ./test.out
ADD test OK
SUB test OK
BIT_OP test OK
BRANCH test OK
DATA test OK
stanganini@u50: ~/tesi$
```

Figura 5.1: Interprete Softcore Risultati

Adesso consideriamo l'esecuzione dell'interprete spiegato nella sez. 4.2, e quindi una singola control unit eseguita all'interno della FPGA. Per mostrare il risultato è stato cambiato leggermente il codice per passare come argomento al main il nome del file assembler da eseguire. Possiamo vedere che i risultati ottenuti combaciano con i risultati aspettati:

```
1 #define BRANCH_RESULT 36
```

Listing 5.2: Risultato Branch



```
stanganini@u50: ~/tesi_impl/funzione_tutto$ ./app.exe vadd.xclbin branch.text
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.xclbin
Loading: 'vadd.xclbin'
Trying to program device[0]: xilinx_u50_gen3x16_xdma_base_5
Device[0]: program successful!
XRT build version: 2.15.225
Build hash: adf27adb3cfadc6e4c41d6db814159f1329b24f3
Build date: 2023-05-03 10:13:19
Git branch: 2023.1
PID: 33081
UID: 1009
[Tue Nov 14 10:01:09 2023 GMT]
HOST: u50
EXE: /home/stanganini/tesi_impl/funzione_tutto/app.exe
[XRT] WARNING: unaligned host pointer '0x563ab2a89eb0' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x563ab2ae65a0' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x563ab2aee0c0' detected, this leads to extra memcpy

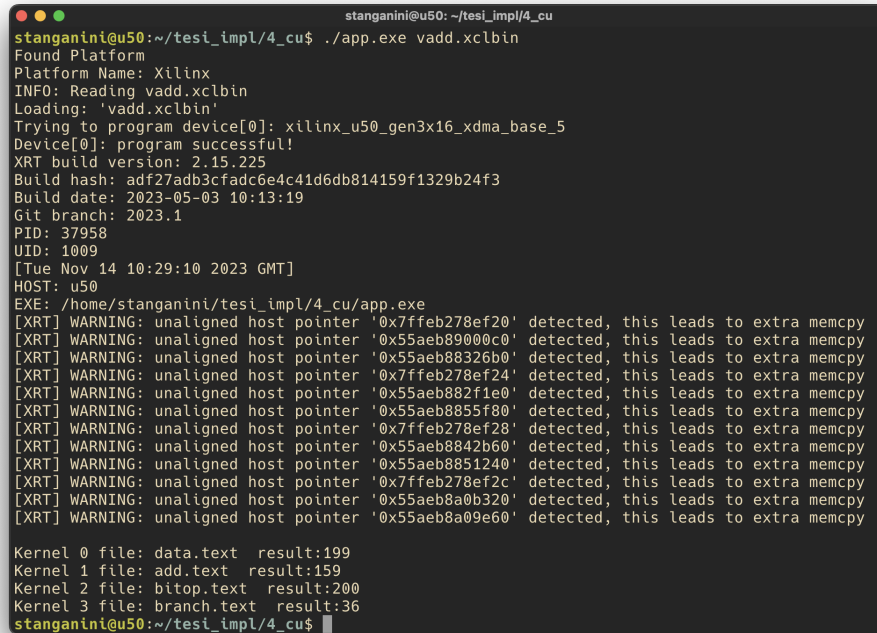
FPGA RESULT 36
stanganini@u50: ~/tesi_impl/funzione_tutto$
```

Figura 5.2: Interprete Kernel Risultati

Questo processo di verifica è stato ripetuto per tutti i file di test, confermando il corretto funzionamento del kernel implementato ed eseguito all'interno della FPGA.

Per la verifica della versione dell'interprete illustrata nella sez. 4.6, ovvero 4 Control Unit, sono stati usati i file assembler precedentemente spiegati nella fig. 5.1. In particolare, per ciascuna istanza del kernel istanziata, è stato usato un file distinto.

Di seguito un immagine dell'esecuzione:



```

stanganini@u50: ~/tesi_impl/4_cu
stanganini@u50:~/tesi_impl/4_cu$ ./app.exe vadd.xclbin
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.xclbin
Loading: 'vadd.xclbin'
Trying to program device[0]: xilinx_u50_gen3x16_xdma_base_5
Device[0]: program successful!
XRT build version: 2.15.225
Build hash: adf27adb3cfadc6e4c41d6db814159f1329b24f3
Build date: 2023-05-03 10:13:19
Git branch: 2023.1
PID: 37958
UID: 1009
[Tue Nov 14 10:29:10 2023 GMT]
HOST: u50
EXE: /home/stanganini/tesi_impl/4_cu/app.exe
[XRT] WARNING: unaligned host pointer '0x7ffeb278ef20' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb89000c0' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb88326b0' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x7ffeb278ef24' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb882f1e0' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb8855f80' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x7ffeb278ef28' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb8842b60' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb8851240' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x7ffeb278ef2c' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb8a0b320' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55aeb8a09e60' detected, this leads to extra memcpy
Kernel 0 file: data.text result:199
Kernel 1 file: add.text result:159
Kernel 2 file: bitop.text result:200
Kernel 3 file: branch.text result:36
stanganini@u50:~/tesi_impl/4_cu$

```

Figura 5.3: Interprete 4 CU Risultati

Possiamo constatare che i risultati ottenuti combaciano con i risultati attesi, confermando la corretta esecuzione delle 4 istanze dell'interprete all'interno della FPGA che eseguono ciascuna un suo programma diverso dagli altri.

Per la verifica dell'interprete in virgola mobile illustrato nella sez. 4.5, considerando che la rappresentazione dei valori in virgola mobile è solo interna all'interprete, per scrivere il file assembly utilizzato nei test, sono stati convertiti i numeri con la virgola usando la funzione di conversione `f_to_i`. Il funzionamento di questa funzione è lo stesso delle funzioni di conversione usate nella sez. 4.5. Di seguito il programma scritto in C utilizzato per testare il risultato e i valori corretti da usare all'interno delle istruzioni assembler:

```

1 void main()
2 {
3     float a = 177.13;
4     printf("a: %d\n", f_to_i(a));
5
6     float b = 231.9999;
7     printf("b: %d\n", f_to_i(b));

```

```

8
9     float c = a + b;
10
11     float d = 99.3;
12     printf("d signed: %d\n", f_to_i(d));
13     float e = u_to_f(f_to_u(c) | f_to_u(d));
14
15     float f = 20.8;
16     printf("f signed: %d\n", f_to_i(f));
17     float result = e - f;
18     printf("res signed: %d\n", f_to_i(result));
19     printf("res: %.6f\n", result);
20 }

```

Listing 5.3: Codice valori conversione floating

Di seguito il codice assembler usato per il test dell'interprete in versione floating point:

```

1     .text
2     .align 2
3     .globl main
4     .ent main
5     .type main, @function
6
7 main:
8
9     addi r1,r0,1127293256      # r1 = 177.13
10    addi r2,r0,1130889209      # r2 = 231.9999
11    add r1,r1,r2               # r1 = 409.129883
12    addi r3,r0,1120311706      # r3 = 99.3
13    or r1,r1,r3               # r1 = 413.200989
14    addi r4,r0,1101424230      # r4 = 20.8
15    rsub r1,r4,r1              # r1 = 392.401001
16
17    .end main

```

Listing 5.4: File Assembler per Test Floating Point

Possiamo notare come il file assembler listato in fig. 5.4 utilizza le stesse operazioni del codice in C listato in fig. 5.3. Il risultato dell'esecuzione è il valore in virgola mobile 392.401001, il quale rappresentato come un `signed int` è 1136931668. Di seguito l'esecuzione dell'interprete in versione Floating Point con il file assembler in listato in fig. 5.4:


```

stanganini@u50: ~/tesi_impl/Floating
stanganini@u50:~/tesi_impl/Floating$ ./app.exe vadd.xclbin
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.xclbin
Loading: 'vadd.xclbin'
Trying to program device[0]: xilinx_u50_gen3x16_xdma_base_5
Device[0]: program successful!
XRT build version: 2.15.225
Build hash: adf27adb3cfadc6e4c41d6db814159f1329b24f3
Build date: 2023-05-03 10:13:19
Git branch: 2023.1
PID: 553247
UID: 1009
[Wed Nov 22 09:24:50 2023 GMT]
HOST: u50
EXE: /home/stanganini/tesi_impl/Floating/app.exe
[XRT] WARNING: unaligned host pointer '0x55c3d3f77870' detected, this leads to extra memcpy
[XRT] WARNING: unaligned host pointer '0x55c3d3f807a0' detected, this leads to extra memcpy

FPGA RESULT 1136931668
stanganini@u50:~/tesi_impl/Floating$

```

Figura 5.4: Interprete Floating Point Risultati

L'ultima versione dell'interprete da verificare è la quella illustrata della sez. 4.7. Questo interprete, come precedentemente spiegato, restituisce l'intera memoria dati dopo l'esecuzione. Per questo, al fine di testare il corretto funzionamento di questo interprete, sono stati impiegati i seguenti file per ciascun "core". Di seguito è riportata la lista dei file utilizzati:

```

1 char const *file[KERNEL_NUMBER] = {
2     "gpu_1.text", // v[i] = v[i] + 1,
3     "gpu_2.text", // +5
4     "gpu_3.text", // -1
5     "gpu_4.text", // *2
6     "gpu_5.text", // *45
7     "gpu_6.text", // & 45
8     "gpu_7.text", // | 45
9     "gpu_8.text", // >> 1
10 };

```

È evidente come per ciascuna Control Unit sia stato creato un file specifico in linguaggio assembly. Questo file prende un vettore di 15 interi dalla memoria dati, precedentemente inizializzato con numeri casuali. Per ciascun elemento del vettore viene eseguita un'operazione diversa (come si può notare dai commenti affianco ai nomi dei file). Di seguito il codice assembly usato per compilare ed estrarre il file `gpu_2.text`:

```

1 .text
2 .align 2
3 .globl main
4 .ent main

```

```

5  .type main, @function
6
7  main:
8
9      addi    r1,r0,14
10     addi    r8,r0,1
11     addi    r2,r0,0
12     lwi     r3,r2,0
13     addi    r3,r3,5
14     swi     r3,r2,0
15     rsub    r1,r8,r1
16     addi    r2,r2,1
17     bgei    r1,-20
18
19     .end    main

```

Per inizializzare il vettore destinato ai test in memoria che sarà usato dal kernel, si parte dalla posizione 0 della memoria dati. Per eseguire questa operazione prima di caricare la memoria all'interno dell'interprete è stata apportata una modificata la funzione `enqueue_task` spiegata nella fig. 4.8, inserendo il seguente codice:

```

1  ...
2  for (int i = 0; i < 15; i++)
3  {
4      int32_t n = rand() % 1000;
5      data->data[i] = n;
6      vector_result[i] = n;
7  }
8  ...

```

È possibile notare che per questioni di semplicità, i numeri si limitano ad un massimo di 1000. Inoltre è evidente la presenza di un ulteriore vettore, ovvero `vector_result`, il quale viene inizializzato con gli stessi valori della memoria dati. Questo vettore viene calcolato dalla CPU della macchina host con le solite operazioni svolte da ogni istanza dell'interprete dentro la FPGA. Questa procedura ha lo scopo di confrontare il vettore con i risultati ottenuti in memoria per ogni kernel, in modo da verificare il corretto funzionamento.

Di seguito è riportato il codice usato nell'interfaccia host per effettuare questa verifica:

```

1  ...
2  struct Memory *data_out[KERNEL_NUMBER];
3  int vector_result[KERNEL_NUMBER][15];
4
5  for (int i = 0; i < KERNEL_NUMBER; i++)
6  {

```

```

7     data_out[i] = (struct Memory *)malloc(sizeof(struct Memory
8         ));
9     enqueue_task(file[i], &q, krnl[i], context, data_out[i],
10         vector_result[i]);
11 }
12
13 OCL_CHECK(err, err = q.finish());
14
15 std::cout << "\n";
16
17 for (int i = 0; i < KERNEL_NUMBER; i++)
18 {
19     make_result(vector_result[i], i);
20     compare(data_out[i], vector_result[i], i);
21     free(data_out[i]);
22 }
23
24 return EXIT_SUCCESS;
25 }

```

Per ciascun kernel è presente un vettore risultati, i quali come spiegato precedentemente vengono inizializzati all'interno della funzione `enqueue_task`. Successivamente attraverso la funzione `make_result()`, vengono calcolati i valori lato host per ogni kernel seguendo le operazioni specificate nei file. In fine, per verificare che i risultati ottenuti siano in linea con i risultati attesi all'interno di `vector_result`, viene utilizzata la funzione `compare`. Di seguito è riportata l'implementazione di tale funzione:

```

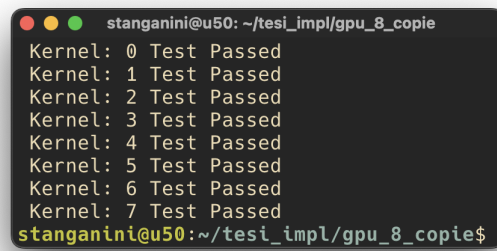
1 void compare(struct Memory *data, int32_t *vector_result, int
2     n)
3 {
4     bool cmp = true;
5
6     for (int i = 0; i < 15; i++)
7         if (data->data[i] != vector_result[i])
8             cmp = false;
9
10    if (cmp)
11        std::cout << " Kernel: " << n << " Test Passed\n";
12    else
13        std::cout << " Kernel: " << n << " Test Failed\n";
14 }

```

Possiamo notare come la funzione accetti come input la memoria restituita dall'interprete e il vettore dei risultati, calcolato sulla macchina host. Successivamente effettua un confronto tra i due e stampa il risultato del risultato ottenuto.

Di seguito è riportata l'immagine con il risultato dell'esecuzione del codice prece-

dentemente descritto:



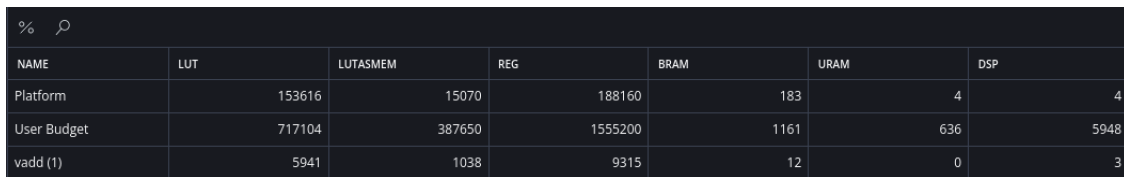
```
stanganini@u50: ~/tesi_impl/gpu_8_copie
Kernel: 0 Test Passed
Kernel: 1 Test Passed
Kernel: 2 Test Passed
Kernel: 3 Test Passed
Kernel: 4 Test Passed
Kernel: 5 Test Passed
Kernel: 6 Test Passed
Kernel: 7 Test Passed
stanganini@u50:~/tesi_impl/gpu_8_copie$
```

Figura 5.5: Interprete GPU Risultati

5.2 Utilizzo FPGA

Durante il processo di compilazione, il compilatore `v++` genera il file `.xclbin.link_summary`. Questo file contiene le informazioni sull'utilizzo effettivo delle risorse del chip della FPGA. Nel seguito di questa sezione verranno presentati i risultati ottenuti dalle diverse versioni dell'interprete. È importante notare che tutti i dati ottenuti durante questa fase siano stati letti utilizzando il software offerto da Vitis ovvero `vitis_analyzer`.

Di seguito i risultati relativi alla versione dell'interprete spiegata nella sez. 4.2, corrispondente ad una singola istanza dell'interprete all'interno della FPGA:



NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	153616	15070	188160	183	4	4
User Budget	717104	387650	1555200	1161	636	5948
vadd (1)	5941	1038	9315	12	0	3

Figura 5.6: Utilizzo FPGA 1 Control Unit

Nella tabella è possibile vedere i seguenti campi:

- **LUT** (Look-Up Table): Gli elementi fondamentali per una FPGA per realizzare circuiti logici. Le LUT sono delle tabelle di ricerca usate per implementare funzioni logiche. Presentano un numero di ingressi fisso e generano un'uscita in base ai valori memorizzati, che variano a seconda della funzione che si vuole realizzare [10].
- **LUTASMEM**: Sono una versione delle LUT che può funzionare come memoria lettura/scrittura.
- **REG** (Register): Nel contesto delle FPGA, i registri sono gruppi di flip-flop utilizzati per memorizzare i dati temporanei. Possono essere utilizzati per sincronizzare segnali, memorizzare risultati intermedi e altro ancora [6].
- **BRAM** (Block RAM): Blocchi di memoria RAM configurabili all'interno della FPGA. Queste memorie sono utilizzate per memorizzare i dati in modo efficiente, migliorando così le prestazioni dei circuiti implementati per le FPGA.
- **URAM** (Ultra RAM): Blocchi di memoria RAM ad alte prestazioni progettate per massimizzare larghezza di banda e latenza. Questi tipi di RAM sono disponibili solo su alcune schede FPGA [11].
- **DSP** (Digital Signal Processor): Sono delle ALU solitamente in Floating Point all'interno della FPGA dedicate alle operazioni matematiche complesse come moltiplicazioni, accumuli e altro ancora [12].

È possibile notare come le righe della tabella in fig. 5.6, riguardano la disponibilità totale delle risorse della piattaforma, quanto l'utente ha a disposizione in termini delle risorse totali e quanto il kernel compilato occupa di queste risorse. È importante sottolineare che la quantità delle risorse disponibile dipende direttamente dal tipo di scheda utilizzata, nel nostro caso la Alveo U50 2.3.

Di seguito sono riportati i risultati dell'occupazione della FPGA per la versione dell'interprete spiegata nella sez. 4.6, ovvero 4 istanze del kernel compilate all'interno della FPGA:

NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	168886	17875	219462	183	4	4
User Budget	701834	384845	1523898	1161	636	5948
vadd (4)	23764	4152	37260	48	0	12

Figura 5.7: Utilizzo FPGA 4 Control Unit

Mettendo a confronto le tabelle in fig. 5.6 e in fig. 5.9 possiamo concludere che:

- LUT: L'utilizzo è di circa 0.8% nella versione a 1 istanza del kernel, rispetto a circa 3.4% nella versione a 4.
- LUTASMEM: L'utilizzo è del 0.3% contro un 1% nella versione a 4 istanze del kernel.
- REG: L'utilizzo è del 0.6% dei registri rispetto a 2.5% per la versione a 4 kernel.
- BRAM: L' utilizzo è del 1% contro un 4%
- URAM: L'utilizzo è del 0% in tutti e due casi.
- DSP: L'utilizzo è del 0.05% contro 0.2%.

Va notato che le percentuali menzionate sono state approssimate per migliorare la leggibilità. Da questi risultati emerge che, l'occupazione delle risorse cresce con l'istanziamento di più kernel all'interno della FPGA. Inoltre si osserva che le risorse utilizzate non aumentano esattamente in modo lineare con il numero di kernel, ma che una parte di risorse è impiegata anche per gestire la complessità introdotta dall'aumentare delle componenti hardware all'interno del chip della FPGA.

Di seguito sono riportati i risultati di occupazione della versione dell'interprete descritta in sez. 4.7:

NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	190317	21615	261456	183	4	4
User Budget	680403	381105	1481904	1161	636	5948
vadd (8)	48096	8304	78504	136	0	24

Figura 5.8: Utilizzo FPGA 8 Control Unit

In questo caso notiamo un utilizzo del 7% delle LUT, 5% dei registri, e 12% di utilizzo delle BRAM. Questi risultati sono coerenti con il trend di crescita dell'utilizzo delle risorse osservato fin'ora. Inoltre va notato come con la scheda usata permetta di compilare ancora maggiore di kernel rispetto al massimo di 8 raggiunti nel codice citato in precedenza. Con l'utilizzo delle risorse osservato finora, è possibile stimare che circa 56 kernel possono essere inseriti nell'FPGA utilizzata per gli esperimenti (la scheda Alveo U50 2.2).

Di seguito sono riportati i risultati dell'occupazione della FPGA per la versione dell'interprete spiegata nella sez. 4.5, ovvero l'interprete con la rappresentazione interna dei valori in virgola mobile:

NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	132477	10917	176990	183	4	4
User Budget	738243	391803	1566370	1161	636	5948
vadd (1)	6993	725	11358	17	0	8

Figura 5.9: Utilizzo FPGA Floating Point

Possiamo notare, come, rispetto all'utilizzo della versione ad 1 CU senza virgola mobile, listata in fig. 5.6, è presente un maggior utilizzo dei DSP, questo perché come spiegato precedentemente nella definizione delle DSP, queste ALU vengono usate per il calcolo in virgola mobile. Di conseguenza, come prevedibile, si è registrato un aumento del loro utilizzo rispetto alla versione senza virgola mobile.

È importante sottolineare che, durante lo sviluppo di questo interprete, l'ottimizzazione nell'occupazione delle risorse della FPGA non era un obiettivo fondamentale. Nella documentazione del processore Microblaze [3], vengono mostrate delle statistiche sull'utilizzo delle risorse nelle varie schede FPGA del mercato con varie implementazioni del processore, sviluppate direttamente dagli sviluppatori del processore utilizzando un linguaggio RTL. Viene mostrato che il processore occupa mediamente circa 3000 LUT. Pertanto, possiamo concludere che anche se il nostro interprete implementi solo un sotto insieme di tutte le istruzioni assembler, e sia scritto in un linguaggio di alto livello e convertito in hardware tramite HLS offerto

da Vitis, con un utilizzo medio di 5600 LUT, possiamo considerarci soddisfatti, tenendo anche conto della facilità di configurazione e estensione che questa soluzione offre.

Durante lo sviluppo del progetto, è stata prestata abbastanza attenzione all'utilizzo delle BRAM all'interno della FPGA. Questo è stato fatto al fine di minimizzare l'uso della memoria globale DDR, la quale trovandosi fuori dal chip FPGA, aggiunge latenza e richiede circuiti hardware aggiuntivi per gestire il processo di comunicazione. Per verificare l'effettivo posizionamento dentro la BRAM delle strutture dati utilizzate all'interno del interprete, sono state sviluppate due versioni. In entrambe è presente un sottoinsieme ancora più piccolo di istruzioni per questioni di semplicità, ma in una versione per la computazione si utilizzavano direttamente i parametri passati dall'OpenCl e caricati nella memoria DDR. Nell'altra versione, si è seguito il paradigma illustrato nella sez. 4.2 e listato nella fig. 4.6 dove i parametri sono copiati su strutture dati dichiarate all'interno della funzione.

Di seguito il codice della versione che non effettua la copia locale dei parametri:

```

1 void interprete(struct Memory *mem, struct Registers *reg,
  uint32_t *out, ap_uint<32> my_size)
2 {
3   pragma HLS INTERFACE m_axi port = mem bundle = gmem
4   pragma HLS INTERFACE m_axi port = reg bundle = gmem
5   pragma HLS INTERFACE m_axi port = out bundle = gmem
6   pragma HLS INTERFACE ap_ctrl_hs port = return
7
8   while (reg->pc < my_size)
9     run_instruction(mem->instr[reg->pc],
10                  mem,
11                  reg,
12                  mem->instr,
13                  false);
14
15   out[1] = reg->r[1];
16 }

```

In questa implementazione, si noti l'uso diretto dei parametri istanziati dalla parte host e caricati nella DDR.

Di seguito i risultati di occupazione delle due versioni:

NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	153037	14781	186952	183	4	4
User Budget	717683	387939	1556408	1161	636	5948
vadd (1)	7742	639	9687	1	0	0

Figura 5.10: Utilizzo senza copia locale dei parametri

% ⌵

NAME	LUT	LUTASMEM	REG	BRAM	URAM	DSP
Platform	153616	15070	188160	183	4	4
User Budget	717104	387650	1555200	1161	636	5948
vadd (1)	4917	1003	8430	16	0	0

Figura 5.11: Utilizzo con copia locale dei parametri

È possibile notare come l'utilizzo delle BRAM sale da 1 a 16 con l'implementazione che prevede la copia locale dei parametri. Questo conferma che le strutture utilizzate per effettuare la copia dei parametri sono istanziate all'interno del chip della FPGA, consentendo un'esecuzione più efficiente e veloce del kernel.

5.3 Execution Summary

Attraverso l'utilizzo del file `xrt.ini`, come indicato dalla documentazione [2], è possibile abilitare la generazione del file `xrt.run_summary`. All'interno di questo file sono raccolte le informazioni sugli eventi registrati durante l'esecuzione della FPGA.

Di seguito l'immagine del `run_summary`, aperto tramite lo strumento `vitis_analyzer`, delle informazioni riguardanti dell'esecuzione del codice spiegato nella sez. 4.2 (1 istanza):



Figura 5.12: `run_summary` 1 Kernel

All'interno di questo grafico, possiamo notare come l'esecuzione del programma lato host, che comprende il caricamento del file `.xclbin`, l'inserimento e l'estrazione dei dati dalla memoria globale, e l'esecuzione effettiva del kernel, dura 413 ms. È interessante notare come gran parte del tempo di esecuzione è dedicato alle chiamate API dal lato host (in fig. 5.12 nella parte superiore), mentre il tempo effettivo dove inizia l'esecuzione della Control Unit (in fig. 5.12) evidenziata con nome **Executions**), avviene in un intervallo molto breve. Ciò è dovuto al fatto che il programma assembly utilizzato per l'esecuzione che possiamo osservare nel grafico in fig. 5.12, è composto di poche istruzioni e, di fatto, non contiene cicli di queste istruzioni.

In un'ottica futura, nel contesto di una possibile "soft GPU" che sfrutti questi softcore, il caricamento del kernel (contenuto nel file `.xclbin`) verrebbe eseguito una sola volta per tutti i core, per poi poter caricare volta per volta lo stato della memoria, dei registri, con i relativi programmi da eseguire. Questo permetterebbe di togliere l'overhead associato al dover programmare nuovamente l'FPGA ogni volta che si desidera eseguire un programma.

Capitolo 6

Conclusioni

L'obiettivo principale di questa tesi è stato lo sviluppo di un semplice interprete in grado di eseguire un sottoinsieme delle istruzioni del processore Microblaze, compilarlo ed eseguirlo sulla FPGA. Sono state create differenti versioni del interprete per dimostrare la flessibilità e configurabilità di questa soluzione. Inoltre una delle versioni dell'interprete è stata usata per simulare una GPU con core indipendenti e privi di controllore SIMD. Il funzionamento di ciascuna versione dell'interprete è stato verificato e validato attraverso test ad-hoc specifici. Inoltre è stata mostrata e discussa l'occupazione di ogni variante dell'interprete sulla FPGA. Questo ha permesso di stimare il numero massimo di istanze che è possibile inserire all'interno del dispositivo. In conclusione tutti gli obiettivi sono stati raggiunti con successo.

6.1 Bilancio Personale

Il lavoro svolto per questa tesi è stato impegnativo e stimolante. L'affrontare un contesto nuovo come quello delle FPGA, senza particolari conoscenze in materia, ha richiesto un piccolo sforzo nell'acquisire dimestichezza con gli strumenti e le metodologie usate, detto questo la documentazione disponibile grazie alla sua chiarezza e completezza ha costituito un grandissimo supporto durante questo processo. Inoltre, non è mancata la sfida nel risolvere i problemi che si sono presentati lungo il percorso. Tuttavia, ho potuto sempre contare sul supporto del mio professore. La sua disponibilità e competenza hanno contribuito in maniera significativa al successo di questo progetto. In definitiva, sebbene il lavoro svolto abbia richiesto impegno e dedizione, l'incontro con il mondo delle FPGA si è rivelato non solo una sfida da affrontare, ma anche un'opportunità unica di crescita personale e accademica.

Bibliografia

- [1] Inc. Advanced Micro Devices. Alveou50, 2022. <https://www.xilinx.com/products/boards-and-kits/alveo/u50.html#documentation>.
- [2] Inc. Advanced Micro Devices. Building and running vitis reference guide, 2023. <https://docs.xilinx.com/r/en-US/Vitis-Tutorials-Getting-Started/Building-and-Running-on-Alveo-U250-Data-Center-Accelerator-Card>.
- [3] Inc. Advanced Micro Devices. Microblaze reference guide, 2023. <https://docs.xilinx.com/v/u/en-US/ug984-vivado-microblaze-ref>.
- [4] Inc. Advanced Micro Devices. Pragma vitis reference guide, 2023. <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-interface>.
- [5] Inc. Advanced Micro Devices. Vitis unified software platform, 2023. <https://www.xilinx.com/products/design-tools/vitis.html>.
- [6] NATIONAL INSTRUMENTS CORP. Introduction to fpga hardware concepts, 2023. https://www.ni.com/docs/en-US/bundle/labview-fpga-module/page/lvfpgaconcepts/fpga_basic_chip_terms.html#:text=A
- [7] Inc. Free Software Foundation. Gcc, the gnu compiler collection, 2023. <https://gcc.gnu.org>.
- [8] Harris Harris. *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufman, 2016.
- [9] The Khronos® Group Inc. Opencl, 2023. <https://www.khronos.org/opencl/>.
- [10] Xilinx Inc. Lut, 2018. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034
- [11] Xilinx Inc. Ultra ram, 2023. <https://docs.xilinx.com/r/en-US/am007-versal-memory/UltraRAM>.
- [12] Wikipedia. Dsp, 2023. <https://it.wikipedia.org/wiki/DSP>.

Appendice A

Appendice

Tutto il Codice seguente è presente in versione integrale nella repo:

<https://github.com/StangaSimo/tesi>

A.1 Codice Host 1 Kernel

```
1 #include "xcl2.hpp"
2 #include <vector>
3
4 #define DATA_SIZE 256
5 #define MAX_INSTR 32
6
7 extern "C"
8 {
9 #include <stdlib.h>
10 #include <stdio.h>
11 #include <string.h>
12 #include <stdbool.h>
13 #include <stdint.h>
14
15 struct Registers
16 {
17     int32_t r[32];
18     int16_t im; /* for imm instruction */
19     bool c : 1;
20     bool i : 1;
21     int32_t pc;
22 };
23
24 struct Memory
25 {
26     int8_t instr[MAX_INSTR][4];
27     int32_t data[1024];
28     int32_t size;
```

```
29     };
30
31     struct Registers *initalize_registers(struct Registers *reg
32     )
33     {
34         for (int i = 0; i < 32; i++)
35         {
36             reg->r[i] = 0;
37         }
38         reg->c = false;
39         reg->i = false;
40         reg->pc = 0;
41         reg->im = 0;
42         return reg;
43     }
44
45     int8_t **get_instructions_from_file(char const *file_name,
46     int32_t *size)
47     {
48         FILE *file = fopen(file_name, "rb");
49         if (file == NULL)
50         {
51             fprintf(stderr, "Error: cannot open the file '%s'.\n",
52             file_name);
53             exit(1);
54         }
55
56         int8_t **instr = (int8_t **)malloc(sizeof(int8_t *));
57
58         fseek(file, 0, SEEK_SET);
59         while (true)
60         {
61             int8_t *buffer = (int8_t *)malloc(sizeof(int8_t) * 4);
62             int num_bytes = fread(buffer, sizeof(int8_t), 4, file);
63             if (num_bytes > 0)
64             {
65                 *size = *size + 1;
66                 instr = (int8_t **)realloc(instr, *size * sizeof(
67                 int8_t *));
68                 instr[*size - 1] = buffer;
69             }
70             else
71             {
72                 break;
73             }
74         }
75         fclose(file);
76         return instr;
77     }
78
79     int main(int argc, char **argv)
```



```

77 {
78     // Verifica numero argomenti
79     if (argc != 3)
80     {
81         std::cout << "Usage: " << argv[0] << " <XCLBIN File>" <<
            std::endl;
82         return EXIT_FAILURE;
83     }
84
85     // Istanziamento Variabili Opencl
86     cl_int err;
87     cl::CommandQueue q;
88     cl::Context context;
89     cl::Kernel krnl;
90     bool valid_device = false;
91
92     // Allocazione risultato
93     int32_t *result = (int32_t*)malloc(sizeof(int32_t));
94     *result = 0;
95
96     // Lettura device
97     auto devices = xcl::get_xil_devices();
98
99     //
100     std::string binaryFile = argv[1];
101     auto fileBuf = xcl::read_binary_file(binaryFile);
102     cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}
        };
103
104     for (unsigned int i = 0; i < devices.size(); i++)
105     {
106         auto device = devices[i];
107
108         // Creazione della coda di comando e del contesto per i
            device presenti
109         OCL_CHECK(err, context = cl::Context(device, nullptr,
            nullptr, nullptr, &err));
110         OCL_CHECK(err, q = cl::CommandQueue(context, device,
            CL_QUEUE_PROFILING_ENABLE, &err));
111
112         std::cout << "Trying to program device[" << i << "]: " <<
            device.getInfo<CL_DEVICE_NAME>() << std::endl;
113         cl::Program program(context, {device}, bins, nullptr, &err
            );
114
115         if (err != CL_SUCCESS)
116         {
117             std::cout << "Failed to program device[" << i << "] with
                xclbin file!\n";
118         }
119         else
120         {

```

```

121     std::cout << "Device[" << i << "]: program successful!\n
    ";
122
123     //Creazione Kernel
124     OCL_CHECK(err, krnl = cl::Kernel(program, "vadd", &err))
    ;
125     valid_device = true;
126     break; // Device valido trovato
127 }
128 }
129 if (!valid_device)
130 {
131     std::cout << "Failed to program any device found, exit!\n"
    ;
132     exit(EXIT_FAILURE);
133 }
134
135 //char const *file = "add.text";
136 // char const* file = "sub.text";
137 // char const* file = "cmp.text";
138 // char const* file = "data.text";
139 // char const* file = "bitop.text";
140 // char const* file = "absolute_branch.text";
141 char const* file = "branch.text";
142
143 int32_t instr_size = 0;
144 int8_t **instr_vector = get_instructions_from_file(argv[2],
    &instr_size);
145 auto mysize = instr_size;
146
147 struct Memory *data = (struct Memory *)malloc(sizeof(struct
    Memory));
148 struct Registers *reg = (struct Registers *)malloc(sizeof(
    struct Registers));
149
150 // Inizializzazione Registri
151 reg = initialize_registers(reg);
152
153 // Inizializzazione Memoria
154 for (int i = 0; i < 1024; i++)
155     data->data[i] = 0;
156
157 // Copia istruzioni
158 for (int i = 0; i < mysize; i++)
159 {
160     data->instr[i][0] = instr_vector[i][0];
161     data->instr[i][1] = instr_vector[i][1];
162     data->instr[i][2] = instr_vector[i][2];
163     data->instr[i][3] = instr_vector[i][3];
164 }
165
166 // Allocazione Buffer nella memoria globale della FPGA

```

```

167 OCL_CHECK(err, cl::Buffer buffer_out(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(int32_t),
    result, &err));
168 OCL_CHECK(err, cl::Buffer buffer_data(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
    Memory), data, &err));
169 OCL_CHECK(err, cl::Buffer buffer_reg(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
    Registers), reg, &err));
170
171 // Configurazione degli argomenti del kernel
172 OCL_CHECK(err, err = krnl.setArg(0, buffer_data));
173 OCL_CHECK(err, err = krnl.setArg(1, buffer_reg));
174 OCL_CHECK(err, err = krnl.setArg(2, buffer_out));
175 OCL_CHECK(err, err = krnl.setArg(3, mysize));
176
177 // Migrazione dei dati dalla memoria host alla memoria
    globale della FGPAy
178 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_data
    , buffer_reg}, 0 /* 0 means from host*/));
179
180 // Inizio esecuzione kerne
181 OCL_CHECK(err, err = q.enqueueTask(krnl));
182
183 // Migrazione risultati dall'acceleratore alla memoria host
184 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_out
    }, CL_MIGRATE_MEM_OBJECT_HOST));
185
186 // Attesa che il kernel completi l'esecuzione
187 OCL_CHECK(err, err = q.finish());
188
189 std::cout << "\n\n FPGA RESULT " << *result << "\n\n";
190
191 return EXIT_SUCCESS;
192 }

```

A.2 Codice Interprete 1 Kernel

```

1 #define MAX_INSTR 32
2 #define TYPE_A 1
3 #define TYPE_B 0
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdbool.h>
8 #include <stdint.h>
9 #include "ap_int.h"
10
11 extern "C"

```

```
12 {
13
14     struct Registers
15     {
16         int32_t r[32];
17         int16_t im;
18         bool c : 1;
19         bool i : 1;
20         int32_t pc;
21     };
22
23     struct Memory
24     {
25         int8_t instr[MAX_INSTR][4];
26         int32_t data[1024];
27         int32_t size;
28     };
29
30     struct Instruction
31     {
32         int8_t type : 1; /* 1 type A, 0 type B */
33         int8_t opcode : 6;
34         int8_t rd : 5;
35         int8_t ra : 5;
36         int8_t rb : 5;
37         int32_t im : 32;
38     };
39
40     int32_t add_Check_Overflow(int32_t a, int32_t b, bool *c)
41     {
42         int64_t res = (int64_t)a + (int64_t)b; /* only last 32 bit
43         */
44         if (res > INT32_MAX)
45         {
46             *c = true;
47             res = INT32_MAX;
48         }
49         else if (res < INT32_MIN) /* underflow */
50         {
51             *c = true;
52             res = INT32_MIN;
53         }
54         else
55         {
56             *c = false;
57         }
58
59         return (int32_t)res;
60     }
61
62     void update_PC(struct Registers *reg, int32_t n, bool delay)
```

```

63 {
64     if (!delay) /* Note: if the instruction is in a delay slot
65                  , it shouldn't modify the pc reg*/
66         reg->pc = reg->pc + n / 4;
67 }
68
69 int8_t conv_reg(int8_t n)
70 {
71     return n & 0b00011111;
72 }
73
74 struct Instruction *parse_instruction(int8_t *instr, int8_t
75                                     type, struct Instruction *res, int16_t *im)
76 {
77     if (type) /* Type A */
78     {
79         res->type = type;
80         res->rd = (instr[0] << 3) + ((instr[1] >> 5) & 0
81                                     b00000111);
82         res->ra = instr[1] & 0b00011111;
83         res->rb = (instr[2] >> 3) & 0b00011111;
84     }
85     else /* Type B */
86     {
87         res->type = type;
88         res->rd = ((instr[0] << 3) & 0b00011000) + ((instr[1] >>
89                                                         5) & 0b00000111);
90         res->ra = instr[1] & 0b00011111;
91         int16_t n = instr[2];
92         n = (n << 8) + (((int16_t)instr[3]) & 0b0000000011111111
93                     );
94
95         if (*im) /* imm instruction before */
96         {
97             res->im = (*im << 16) + ((int32_t)n & 0
98                                     b00000000000000001111111111111111);
99             *im = 0;
100         }
101         else
102             res->im = (int32_t)n;
103     }
104     return res;
105 }
106
107 void run_instruction(int8_t *instruction, struct Memory *
108                    data, struct Registers *reg, int8_t instructions[32][4],
109                    bool delay)
110 {
111     struct Instruction instr_data;
112     struct Instruction *instr = &instr_data;
113     bool carry = false; // carry
114     int8_t op_code = (instruction[0] >> 2) & 0b00111111; /*

```

```

remove last 11 bitfield is not working for the == operand*/
107   instr->opcode = (instruction[0] >> 2) & 0b00111111;
108   int32_t delayed_instruction, addr;
109   int8_t branch_type, is_delayed, is_absolute, is_link;
110
111   switch (op_code)
112   {
113       case 0x0:
114       { /* ADD 000000 */
115           instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
116           reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb], &carry);
117           update_PC(reg, 4, delay);
118           break;
119       }
120
121       case 0x4:
122       { /* ADDK 000100 */
123           instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
124           reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb], &carry);
125           reg->c = carry;
126           update_PC(reg, 4, delay);
127           break;
128       }
129
130       case 0x2:
131       { /* ADDC 000010 */
132           instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
133           reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb] + reg->c, &carry);
134           update_PC(reg, 4, delay);
135           break;
136       }
137
138       case 0x6:
139       { /* ADDCK 000110 */
140           instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
141           reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb] + reg->c, &carry);
142           reg->c = carry;
143           update_PC(reg, 4, delay);
144           break;
145       }
146
147       case 0x8:
148       { /* ADDI 001000 */
149           instr = parse_instruction(instruction, TYPE_B, instr, &

```

```

reg->im);
150     reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra], &carry);
151     update_PC(reg, 4, delay);
152     break;
153 }
154
155     case 0xA:
156     { /* ADDIC 001010 */
157         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
158         reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra] + reg->c, &carry);
159         update_PC(reg, 4, delay);
160         break;
161     }
162
163     case 0xC:
164     { /* ADDIK 001100 */
165         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
166         reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra], &carry);
167         reg->c = carry;
168         update_PC(reg, 4, delay);
169         break;
170     }
171
172     case 0xE:
173     { /* ADDICK 001110 */
174         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
175         reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra] + reg->c, &carry);
176         reg->c = carry;
177         update_PC(reg, 4, delay);
178         break;
179     }
180
181     case 0x1:
182     { /* RSUB 000001 */
183         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
184         reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
], add_Check_Overflow(~reg->r[instr->ra], 1, &carry), &
carry);
185         update_PC(reg, 4, delay);
186         break;
187     }
188
189     case 0x3:
190     { /* RSUBC 000011 */

```

```

191     instr = parse_instruction(instruction, TYPE_A, instr, &
192     reg->im);
193     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
194     ], add_Check_Overflow(~reg->r[instr->ra], 1, &carry) + reg
195     ->c, &carry);
196     update_PC(reg, 4, delay);
197     break;
198 }
199
200 case 0x7:
201 { /* RSUBCK 000111 */
202     instr = parse_instruction(instruction, TYPE_A, instr, &
203     reg->im);
204     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
205     ], add_Check_Overflow(~reg->r[instr->ra], 1, &carry) + reg
206     ->c, &carry);
207     reg->c = carry;
208     update_PC(reg, 4, delay);
209     break;
210 }
211
212 case 0x9:
213 { /* RSUBI 001001 */
214     instr = parse_instruction(instruction, TYPE_B, instr, &
215     reg->im);
216     reg->r[instr->rd] = add_Check_Overflow(instr->im,
217     add_Check_Overflow(~reg->r[instr->ra], 1, &carry), &carry);
218     update_PC(reg, 4, delay);
219     break;
220 }
221
222 case 0xB:
223 { /* RSUBIC 001011 */
224     instr = parse_instruction(instruction, TYPE_B, instr, &
225     reg->im);
226     reg->r[instr->rd] = add_Check_Overflow(instr->im,
227     add_Check_Overflow(~reg->r[instr->ra], 1, &carry) + reg->c,
228     &carry);
229     update_PC(reg, 4, delay);
230     break;
231 }
232
233 case 0xD:
234 { /* RSUBIK 001101 */
235     instr = parse_instruction(instruction, TYPE_B, instr, &
236     reg->im);
237     reg->r[instr->rd] = add_Check_Overflow(instr->im,
238     add_Check_Overflow(~reg->r[instr->ra], 1, &carry), &carry);
239     reg->c = carry;
240     update_PC(reg, 4, delay);
241     break;
242 }

```



```

230
231     case 0xF:
232     { /* RSUBIKC 001111 */
233         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
234         reg->r[instr->rd] = add_Check_Overflow(instr->im,
add_Check_Overflow(~reg->r[instr->ra], 1, &carry) + reg->c,
&carry);
235         reg->c = carry;
236         update_PC(reg, 4, delay);
237         break;
238     }
239
240     case 0x5:
241     { /* RSUBK CMP 000101 */
242
243         /* last 8 bit 00000011 then cmpu */
244         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
245
246         if (instruction[3] == 0x0)
247         {
248             instr = parse_instruction(instruction, TYPE_A, instr,
&reg->im);
249             reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->
rb], add_Check_Overflow(~reg->r[instr->ra], 1, &carry), &
carry);
250             reg->c = carry;
251         }
252         else
253         {
254             if (instruction[3] == 0x3)
255             {
256                 reg->r[instr->rd] = ((uint32_t)reg->r[instr->rb]) +
(~((uint32_t)reg->r[instr->ra]) + 1);
257
258                 if (((uint32_t)reg->r[instr->ra]) > ((uint32_t)reg->
r[instr->rb]))
259                     reg->r[instr->rd] = (reg->r[instr->rd] & 0
x7FFFFFFF) + 0x80000000; /* (rD)(MSB)      (rA) > (rB) */
260             }
261             else
262             {
263                 reg->r[instr->rd] = reg->r[instr->rb] + (~reg->r[
instr->ra] + 1);
264
265                 if (reg->r[instr->ra] > reg->r[instr->rb])
266                     reg->r[instr->rd] = (reg->r[instr->rd] & 0
x7FFFFFFF) + 0x80000000;
267             }
268         }
269         update_PC(reg, 4, delay);

```

```

270     break;
271 }
272 case 0x10:
273 { /* MUL 010000 */
274     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
275     reg->r[instr->rd] = (int32_t)(((int64_t)reg->r[instr->ra
276 ]) * ((int64_t)reg->r[instr->rb]) & 0xFFFFFFFF);
277     update_PC(reg, 4, delay);
278     break;
279 }
280 case 0x24:
281 { /* SRA 100100 */
282     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
283     reg->r[instr->rd] = reg->r[instr->ra] >> 1;
284     reg->c = carry;
285     update_PC(reg, 4, delay);
286     break;
287 }
288
289 case 0x20:
290 { /* OR 100000 */
291     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
292     reg->r[instr->rd] = reg->r[instr->ra] | reg->r[instr->rb
293 ];
294     update_PC(reg, 4, delay);
295     break;
296 }
297 case 0x21:
298 { /* AND 100001 */
299     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
300     reg->r[instr->rd] = reg->r[instr->ra] & reg->r[instr->rb
301 ];
302     update_PC(reg, 4, delay);
303     break;
304 }
305 case 0x22:
306 { /* XOR 100010 */
307     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
308     reg->r[instr->rd] = reg->r[instr->ra] ^ reg->r[instr->rb
309 ];
310     update_PC(reg, 4, delay);
311     break;
312 }

```

```

313     case 0x23:
314     { /* ANDN 100011 */
315         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
316         reg->r[instr->rd] = reg->r[instr->ra] & (~reg->r[instr->
rb]);
317         update_PC(reg, 4, delay);
318         break;
319     }
320
321     case 0x28:
322     { /* ORI 101000 */
323         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
324         reg->r[instr->rd] = reg->r[instr->ra] | instr->im;
325         update_PC(reg, 4, delay);
326         break;
327     }
328
329     case 0x29:
330     { /* ANDI 101001 */
331         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
332         reg->r[instr->rd] = reg->r[instr->ra] & instr->im;
333         update_PC(reg, 4, delay);
334         break;
335     }
336     case 0x2A:
337     { /* XORI 101010 */
338         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
339         reg->r[instr->rd] = reg->r[instr->ra] ^ instr->im;
340         update_PC(reg, 4, delay);
341         break;
342     }
343
344     case 0x2B:
345     { /* ANDNI 101011 */
346         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
347         reg->r[instr->rd] = reg->r[instr->ra] & (~instr->im);
348         update_PC(reg, 4, delay);
349         break;
350     }
351
352     case 0x27:
353     { /* BEQ BGE BGT BLE BLT BNE 100111 */
354         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
355         delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
356         branch_type = conv_reg(instr->rd) & 0b00001111;

```

```

357         is_delayed = conv_reg(instr->rd) & 0b00010000;
358
359         if (branch_type == 0x0 && reg->r[instr->ra] == 0x0) /*
BEQ D0000 */
360             update_PC(reg, reg->r[instr->rb], delay);
361         else if (branch_type == 0x5 && reg->r[instr->ra] >= 0x0)
/* BGE D0101 */
362             update_PC(reg, reg->r[instr->rb], delay);
363         else if (branch_type == 0x4 && reg->r[instr->ra] > 0x0)
/* BGT D0100 */
364             update_PC(reg, reg->r[instr->rb], delay);
365         else if (branch_type == 0x3 && reg->r[instr->ra] <= 0x0)
/* BLE D0011 */
366             update_PC(reg, reg->r[instr->rb], delay);
367         else if (branch_type == 0x2 && reg->r[instr->ra] < 0x0)
/* BLT D0010 */
368             update_PC(reg, reg->r[instr->rb], delay);
369         else if (branch_type == 0x1 && reg->r[instr->ra] != 0x0)
/* BNE D0001 */
370             update_PC(reg, reg->r[instr->rb], delay);
371         else
372             update_PC(reg, 4, delay);
373
374         // if (is_delayed == 0x10) /* delayed slot */
375         //     run_instruction(instructions[delayed_instruction],
data, reg, instructions, true);
376         break;
377     }
378
379     case 0x2F:
380     { /* BEQI BGEI BGTI BLEI BLTI BNEI 101111 */
381         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
382         delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
383         branch_type = conv_reg(instr->rd) & 0b00001111;
384         is_delayed = conv_reg(instr->rd) & 0b00010000;
385
386         if (branch_type == 0x0 && reg->r[instr->ra] == 0x0) /*
BEQI D0000 */
387             update_PC(reg, instr->im, delay);
388         else if (branch_type == 0x5 && reg->r[instr->ra] >= 0x0)
/* BGEI D0101 */
389             update_PC(reg, instr->im, delay);
390         else if (branch_type == 0x4 && reg->r[instr->ra] > 0x0)
/* BGTI D0100 */
391             update_PC(reg, instr->im, delay);
392         else if (branch_type == 0x3 && reg->r[instr->ra] <= 0x0)
/* BLEI D0011 */
393             update_PC(reg, instr->im, delay);
394         else if (branch_type == 0x2 && reg->r[instr->ra] < 0x0)
/* BLTI D0010 */

```

```

395     update_PC(reg, instr->im, delay);
396     else if (branch_type == 0x1 && reg->r[instr->ra] != 0x0)
/* BNEI D0001 */
397     update_PC(reg, instr->im, delay);
398     else
399     update_PC(reg, 4, delay);
400
401     // if (is_delayed == 0x10) /* delayed slot */
402     // run_instruction(instructions[delayed_instruction],
data, reg, instructions, true);
403     break;
404 }
405
406 case 0x26:
407 { /* BR BRD BRA BRLD BRAD BRALD 101110 */
408     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
409     delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
410
411     /* DAL00 */
412     is_absolute = conv_reg(instr->ra) & 0b00001000; /* A for
branch with absolute PC = rb */
413     is_link = conv_reg(instr->ra) & 0b00000100; /* L for
branch and link rd = PC */
414     is_delayed = conv_reg(instr->ra) & 0b00010000;
415
416     if (is_link == 0x4)
417         reg->r[conv_reg(instr->rd)] = reg->pc * 4;
418
419     if (is_absolute == 0x8)
420         reg->pc = (reg->r[conv_reg(instr->rb)]) / 4;
421     else
422         update_PC(reg, reg->r[instr->rb], delay);
423
424     // if (is_delayed == 0x10) /* delayed slot */
425     // run_instruction(instructions[delayed_instruction],
data, reg, instructions, true);
426     break;
427 }
428 case 0x2E:
429 { /* BRI BRAI BRID BRAID BRLID BRAILD 101110 */
430     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
431     delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
432
433     /* DAL00 */
434     is_absolute = conv_reg(instr->ra) & 0b00001000; /* A for
branch with absolute PC = rb */
435     is_link = conv_reg(instr->ra) & 0b00000100; /* L for
branch and link rd = PC */

```

```

436     is_delayed = conv_reg(instr->ra) & 0b00010000;
437
438     if (is_link == 0x4)
439         reg->r[conv_reg(instr->rd)] = reg->pc * 4;
440
441     if (is_absolute == 0x8)
442         reg->pc = instr->im / 4;
443     else
444         update_PC(reg, instr->im, delay);
445
446     // if (is_delayed == 0x10) /* delayed slot */
447     //     run_instruction(instructions[delayed_instruction],
448     data, reg, instructions, true);
449     break;
450 }
451
452 case 0x32:
453 { /* LW 110010 */
454     instr = parse_instruction(instruction, TYPE_A, instr, &
455 reg->im);
456     addr = (uint32_t)(reg->r[instr->ra] + reg->r[instr->rb])
457 ;
458     reg->r[instr->rd] = data->data[addr];
459     update_PC(reg, 4, delay);
460     break;
461 }
462 case 0x36:
463 { /* SW 110110 */
464     instr = parse_instruction(instruction, TYPE_A, instr, &
465 reg->im);
466     addr = (uint32_t)(reg->r[instr->ra] + reg->r[instr->rb])
467 ;
468     data->data[addr] = reg->r[instr->rd];
469     update_PC(reg, 4, delay);
470     break;
471 }
472
473 case 0x3A:
474 { /* LWI 111010 */
475     instr = parse_instruction(instruction, TYPE_B, instr, &
476 reg->im);
477     addr = (uint32_t)(reg->r[instr->ra] + instr->im);
478     reg->r[instr->rd] = data->data[addr];
479     update_PC(reg, 4, delay);
480     break;
481 }
482
483 case 0x3E:
484 { /* SWI 111110 */
485     instr = parse_instruction(instruction, TYPE_B, instr, &
486 reg->im);
487     addr = (uint32_t)(reg->r[instr->ra] + instr->im);

```

```

481     data->data[addr] = reg->r[instr->rd];
482     update_PC(reg, 4, delay);
483     break;
484 }
485
486 case 0x2C:
487 { /* IMM 101100 */
488     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
489     reg->im = instr->im;
490     update_PC(reg, 4, delay);
491     break;
492 }
493
494 default:
495 {
496     /* unknown instruction */
497     break;
498 }
499 }
500 }
501
502 void vadd(struct Memory *mem, struct Registers *reg,
uint32_t *out, ap_uint<32> my_size)
503 {
504 #pragma HLS INTERFACE m_axi port = mem bundle = gmem
505 #pragma HLS INTERFACE m_axi port = reg bundle = gmem
506 #pragma HLS INTERFACE m_axi port = out bundle = gmem
507 #pragma HLS INTERFACE ap_ctrl_hs port = return
508
509     struct Registers reg_copy;
510     struct Memory mem_copy;
511
512     for (int i = 0; i < 32; i++)
513         reg_copy.r[i] = reg->r[i];
514
515     reg_copy.c = reg->c;
516     reg_copy.i = reg->i;
517     reg_copy.pc = reg->pc;
518     reg_copy.im = reg->im;
519
520     for (int i = 0; i < 1024; i++)
521         mem_copy.data[i] = mem->data[i];
522
523     for (int i = 0; i < MAX_INSTR; i++)
524         for (int j = 0; j < 4; j++)
525             mem_copy.instr[i][j] = mem->instr[i][j];
526
527     struct Registers *reg_copy_pointer = &reg_copy;
528     struct Memory *mem_copy_pointer = &mem_copy;
529
530     while (reg_copy_pointer->pc < my_size)

```

```

531     run_instruction(mem_copy_pointer->instr[reg_copy_pointer
->pc],
532                     mem_copy_pointer,
533                     reg_copy_pointer,
534                     mem_copy_pointer->instr,
535                     false);
536
537     out[0] = reg_copy_pointer->r[1];
538 }
539 }

```

A.3 Codice Host GPU

```

1  #include "xcl2.hpp"
2  #include <vector>
3
4  #define KERNEL_NUMBER 8
5
6  extern "C"
7  {
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include <stdbool.h>
12 #include <stdint.h>
13
14  struct Registers
15  {
16      int32_t r[32];
17      int16_t im; /* for imm instruction */
18      bool c : 1;
19      bool i : 1;
20      int32_t pc;
21  };
22
23  struct Memory
24  {
25      int8_t instr[32][4];
26      int32_t data[1024];
27      int32_t size;
28  };
29
30  struct Registers *initalize_registers(struct Registers *reg
31  )
32  {
33      for (int i = 0; i < 32; i++)
34      {
35          reg->r[i] = 0;
36      }
37  }

```



```

36     reg->c = false;
37     reg->i = false;
38     reg->pc = 0;
39     reg->im = 0;
40     return reg;
41 }
42
43 int8_t **get_instructions_from_file(char const *file_name,
44     int32_t *size)
45 {
46     FILE *file_1 = fopen(file_name, "rb");
47     if (file_1 == NULL)
48     {
49         fprintf(stderr, "Error: cannot open the file_1 '%s'.\n",
50             file_name);
51         exit(1);
52     }
53
54     int8_t **instr = (int8_t **)malloc(sizeof(int8_t *));
55
56     fseek(file_1, 0, SEEK_SET);
57     while (true)
58     {
59         int8_t *buffer = (int8_t *)malloc(sizeof(int8_t) * 4);
60         int num_bytes = fread(buffer, sizeof(int8_t), 4, file_1);
61
62         if (num_bytes > 0)
63         {
64             *size = *size + 1;
65             instr = (int8_t **)realloc(instr, *size * sizeof(
66 int8_t *));
67             instr[*size - 1] = buffer;
68         }
69         else
70         {
71             break;
72         }
73     }
74     fclose(file_1);
75     return instr;
76 }
77
78 void make_result(int32_t *vector_result, int n)
79 {
80     switch (n)
81     {
82     case 0:
83         for (int i=0; i<15; i++)
84             vector_result[i] = vector_result[i]+1;
85         break;
86     case 1:

```

```

84     for (int i=0; i<15; i++)
85         vector_result[i] = vector_result[i]+5;
86     break;
87     break;
88 case 2:
89     for (int i=0; i<15; i++)
90         vector_result[i] = vector_result[i]-1;
91     break;
92 case 3:
93     for (int i=0; i<15; i++)
94         vector_result[i] = vector_result[i]*2;
95     break;
96 case 4:
97     for (int i=0; i<15; i++)
98         vector_result[i] = vector_result[i]*45;
99     break;
100 case 5:
101     for (int i=0; i<15; i++)
102         vector_result[i] = vector_result[i] & 45;
103     break;
104 case 6:
105     for (int i=0; i<15; i++)
106         vector_result[i] = vector_result[i] | 45;
107     break;
108 case 7:
109     for (int i=0; i<15; i++)
110         vector_result[i] = vector_result[i] >> 1;
111     break;
112
113 default:
114     break;
115 }
116 }
117 /*
118     *****
119     */
118 void compare(struct Memory *data, int32_t *vector_result, int
119             n)
119 {
120     bool cmp = true;
121
122     for (int i = 0; i<15; i++)
123         if (data->data[i] != vector_result[i])
124             cmp = false;
125
126     if (cmp)
127         std::cout << " Kernel: " << n << " Test Passed\n";
128     else
129         std::cout << " Kernel: " << n << " Test Failed\n";
130 }
131
132 /*

```

```

    /**
133
134 void print_vector(int n, struct Memory *data, int j, const
    char * file)
135 {
136     std::cout << " Kernel: " << j << " file: " << file << "
        vector: ";
137     for (int i = 0; i < n; i++)
138     {
139         std::cout << data->data[i] << " ";
140     }
141     std::cout << "\n";
142 }
143 /**
    /**
144
145 void free_data(struct Memory *data, struct Memory *data_out,
    struct Registers *reg)
146 {
147     free(data);
148     free(data_out);
149     free(reg);
150 }
151
152 /**
    /**
153
154 void enqueue_task(char const *file,
155                  cl::CommandQueue *q,
156                  cl::Kernel krnl,
157                  cl::Context context,
158                  struct Memory *data_out,
159                  int32_t *vector_result)
160
161 {
162     cl_int err;
163     int32_t instr_size = 0;
164     int8_t **vector = get_instructions_from_file(file, &
        instr_size);
165     auto mysize = instr_size;
166     struct Memory *data = (struct Memory *)malloc(sizeof(struct
        Memory));
167     struct Registers *reg = (struct Registers *)malloc(sizeof(
        struct Registers));
168     reg = initialize_registers(reg);
169
170     for (int i = 0; i < 1024; i++)
171     {
172         data->data[i] = 0;

```

```

173     data_out->data[i] = 0;
174 }
175
176 for (int i = 0; i < 15; i++)
177 {
178     int32_t n = rand() % 1000;
179     data->data[i] = n;
180     vector_result[i] = n;
181 }
182
183 for (int i = 0; i < mysize; i++)
184 {
185     data->instr[i][0] = vector[i][0];
186     data->instr[i][1] = vector[i][1];
187     data->instr[i][2] = vector[i][2];
188     data->instr[i][3] = vector[i][3];
189 }
190
191 OCL_CHECK(err, cl::Buffer buffer(context,
192     CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, sizeof(struct
193     Memory), data_out, &err));
194
195 OCL_CHECK(err, cl::Buffer buffer_data(context,
196     CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
197     Memory), data, &err));
198
199 OCL_CHECK(err, cl::Buffer buffer_reg(context,
200     CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, sizeof(struct
201     Registers), reg, &err));
202
203 // Set the Kernel Arguments
204 OCL_CHECK(err, err = krnl.setArg(0, buffer_data));
205 OCL_CHECK(err, err = krnl.setArg(1, buffer_reg));
206 OCL_CHECK(err, err = krnl.setArg(2, buffer));
207 OCL_CHECK(err, err = krnl.setArg(3, mysize));
208
209 // Copy input data to device global memory
210 OCL_CHECK(err, err = q->enqueueMigrateMemObjects({
211     buffer_data, buffer_reg}, 0 /* 0 means from host*/));
212
213 // Launch the Kernel
214 OCL_CHECK(err, err = q->enqueueTask(krnl));
215
216 // Copy Result from Device Global Memory to Host Local
217 // Memory
218 OCL_CHECK(err, err = q->enqueueMigrateMemObjects({buffer},
219     CL_MIGRATE_MEM_OBJECT_HOST));
220 }
221
222 /*
223 *****
224 */

```

```

214
215 int main(int argc, char **argv)
216 {
217     if (argc != 2)
218     {
219         std::cout << "Usage: " << argv[0] << " <XCLBIN File>" <<
220         std::endl;
221         return EXIT_FAILURE;
222     }
223
224     std::string binaryFile = argv[1];
225     cl_int err;
226     cl::CommandQueue q;
227     cl::Context context;
228     cl::Kernel krnl[KERNEL_NUMBER];
229
230     // OPENCL HOST CODE AREA START
231     // Create Program and Kernel
232     auto devices = xcl::get_xil_devices();
233
234     // read_binary_file() is a utility API which will load the
235     // binaryFile
236     // and will return the pointer to file_1 buffer.
237     auto fileBuf = xcl::read_binary_file(binaryFile);
238     cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()
239     }};
240     bool valid_device = false;
241
242     for (unsigned int i = 0; i < devices.size(); i++)
243     {
244         auto device = devices[i];
245         // Creating Context and Command Queue for selected Device
246         OCL_CHECK(err, context = cl::Context(device, nullptr,
247         nullptr, nullptr, &err));
248         OCL_CHECK(err, q = cl::CommandQueue(context, device,
249         CL_QUEUE_PROFILING_ENABLE, &err));
250
251         std::cout << "Trying to program device[" << i << "]: " <<
252         device.getInfo<CL_DEVICE_NAME>() << std::endl;
253         cl::Program program(context, {device}, bins, nullptr, &err
254         );
255
256         if (err != CL_SUCCESS)
257         {
258             std::cout << "Failed to program device[" << i << "] with
259             xclbin file_1!\n";
260         }
261         else
262         {
263             std::cout << "Device[" << i << "]: program successful!\n
264             ";

```

```

257     for (int i = 1; i <= KERNEL_NUMBER; i++)
258     {
259         std::string var = "vadd:{vadd_" + std::to_string(i) +
        "}"
260         OCL_CHECK(err, krnl[i - 1] = cl::Kernel(program, var.
        c_str(), &err));
261     }
262
263     valid_device = true;
264     break;
265 }
266 }
267 if (!valid_device)
268 {
269     std::cout << "Failed to program any device found, exit!\n"
        ;
270     exit(EXIT_FAILURE);
271 }
272
273 char const *file[KERNEL_NUMBER] = {
274     "gpu_1.text", // +1,
275     "gpu_2.text", // +5
276     "gpu_3.text", // -1
277     "gpu_4.text", // *2
278     "gpu_5.text", // *45
279     "gpu_6.text", // & 45
280     "gpu_7.text", // | 45
281     "gpu_8.text", // >> 1
282 };
283
284 struct Memory *data_out[KERNEL_NUMBER];
285 int vector_result[KERNEL_NUMBER][15];
286
287 for (int i = 0; i < KERNEL_NUMBER; i++)
288 {
289     data_out[i] = (struct Memory *)malloc(sizeof(struct Memory
        ));
290     enqueue_task(file[i], &q, krnl[i], context, data_out[i],
        vector_result[i]);
291     make_result(vector_result[i], i);
292 }
293
294 OCL_CHECK(err, err = q.finish());
295
296 std::cout << "\n";
297
298 for (int i = 0; i < KERNEL_NUMBER; i++)
299 {
300     compare(data_out[i], vector_result[i], i);
301     free(data_out[i]);
302 }
303

```

```

304     return EXIT_SUCCESS;
305 }

```

A.4 Codice Interprete Floating Point

```

1
2 #define TYPE_A 1
3 #define TYPE_B 0
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdbool.h>
8 #include <stdint.h>
9 #include "ap_int.h"
10
11 extern "C"
12 {
13
14     struct Registers
15     {
16         float r[32];
17         uint16_t im; /* 16 bit */
18         bool c : 1;
19         bool i : 1;
20         int32_t pc;
21     };
22
23     struct Memory
24     {
25         int8_t instr[32][4];
26         float data[1024];
27         int32_t size;
28     };
29
30     struct Instruction
31     {
32         int8_t type : 1; /* 1 type A, 0 type B */
33         int8_t opcode : 6;
34         int8_t rd : 5;
35         int8_t ra : 5;
36         int8_t rb : 5;
37         float im; /* had to remove the bit field */
38     };
39
40     float add_Check_Overflow(float a, float b, bool *c)
41     {
42         double res = (double)a + (double)b; /* only last 32 bit*/
43         if (res > INT32_MAX)
44         {

```

```

45     *c = true;
46     res = INT32_MAX;
47 }
48 else if (res < INT32_MIN) /* underflow */
49 {
50
51     *c = true;
52     res = INT32_MIN;
53 }
54 else
55 {
56     *c = false;
57 }
58
59     return (float)res;
60 }
61
62 uint32_t f_to_u(float n)
63 {
64     return *(uint32_t *)&n;
65 }
66
67 int32_t f_to_i(float n)
68 {
69     return *(int32_t *)&n;
70 }
71
72 float u_to_f(uint32_t n)
73 {
74     return *(float *)&n;
75 }
76
77 void update_PC(struct Registers *reg, int32_t n, bool delay)
78 {
79     if (!delay) /* Note: if the instruction is in a delay slot
80                  , it shouldn't modify the pc reg*/
81         reg->pc = reg->pc + n / 4;
82 }
83
84 int8_t conv_reg(int8_t n)
85 {
86     return n & 0b00011111;
87 }
88
89 struct Instruction *parse_instruction(int8_t *instr, int8_t
90 type, struct Instruction *res, uint16_t *im)
91 {
92     if (type) /* Type A */
93     {
94         res->type = type;
95         res->rd = (instr[0] << 3) + ((instr[1] >> 5) & 0
96 b00000111);

```



```

94     res->ra = instr[1] & 0b00011111;
95     res->rb = (instr[2] >> 3) & 0b00011111;
96 }
97 else /* Type B */
98 {
99     res->type = type;
100    res->rd = ((instr[0] << 3) & 0b00011000) + ((instr[1] >>
101    5) & 0b00000111);
102    res->ra = instr[1] & 0b00011111;
103    int16_t n = instr[2];
104    n = (n << 8) + (((int16_t)instr[3]) & 0b0000000011111111
105    );
106
107    /* TODO */
108
109    if (*im) /* imm instruction before */
110    {
111        uint32_t im_temp = ((uint32_t)*im << 16) + ((uint32_t)
112        n & 0b00000000000000001111111111111111);
113        res->im = *(float *)&im_temp;
114        *im = 0;
115    }
116    else
117    {
118        int32_t n_32 = (int32_t)n;
119        res->im = *(float *)&n_32;
120    }
121    return res;
122 }
123
124 void run_instruction(int8_t *instruction, struct Memory *
125 data, struct Registers *reg, int8_t instructions[32][4],
126 bool delay)
127 {
128     struct Instruction instr_data;
129     struct Instruction *instr = &instr_data;
130     bool carry = false; // carry
131     int8_t op_code = (instruction[0] >> 2) & 0b00111111; /*
132     remove last 11 bitfield is not working for the == operand*/
133     instr->opcode = (instruction[0] >> 2) & 0b00111111;
134     int32_t delayed_instruction, addr;
135     int8_t branch_type, is_delayed, is_absolute, is_link;
136
137     switch (op_code)
138     {
139     {
140     case 0x0:
141     { /* ADD 000000 */
142         instr = parse_instruction(instruction, TYPE_A, instr, &
143         reg->im);
144         reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
145         ], reg->r[instr->rb], &carry);

```

```

138     update_PC(reg, 4, delay);
139     break;
140 }
141
142 case 0x4:
143 { /* ADDK 000100 */
144     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
145     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb], &carry);
146     reg->c = carry;
147     update_PC(reg, 4, delay);
148     break;
149 }
150
151 case 0x2:
152 { /* ADDC 000010 */
153     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
154     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb] + reg->c, &carry);
155     update_PC(reg, 4, delay);
156     break;
157 }
158
159 case 0x6:
160 { /* ADDCK 000110 */
161     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
162     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->ra
], reg->r[instr->rb] + reg->c, &carry);
163     reg->c = carry;
164     update_PC(reg, 4, delay);
165     break;
166 }
167
168 case 0x8:
169 { /* ADDI 001000 */
170     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
171     reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra], &carry);
172     update_PC(reg, 4, delay);
173     break;
174 }
175
176 case 0xA:
177 { /* ADDIC 001010 */
178     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
179     reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra] + reg->c, &carry);

```

```

180     update_PC(reg, 4, delay);
181     break;
182 }
183
184 case 0xC:
185 { /* ADDIK 001100 */
186     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
187     reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra], &carry);
188     reg->c = carry;
189     update_PC(reg, 4, delay);
190     break;
191 }
192
193 case 0xE:
194 { /* ADDICK 001110 */
195     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
196     reg->r[instr->rd] = add_Check_Overflow(instr->im, reg->r
[instr->ra] + reg->c, &carry);
197     reg->c = carry;
198     update_PC(reg, 4, delay);
199     break;
200 }
201
202 case 0x1:
203 { /* RSUB 000001 */
204     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
205     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
], - reg->r[instr->ra], &carry);
206     update_PC(reg, 4, delay);
207     break;
208 }
209
210 case 0x3:
211 { /* RSUBC 000011 */
212     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
213     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
], - reg->r[instr->ra] + reg->c, &carry);
214     update_PC(reg, 4, delay);
215     break;
216 }
217
218 case 0x7:
219 { /* RSUBCK 000111 */
220     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
221     reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->rb
], - reg->r[instr->ra] + reg->c, &carry);

```

```

222     reg->c = carry;
223     update_PC(reg, 4, delay);
224     break;
225 }
226
227     case 0x9:
228     { /* RSUBI 001001 */
229         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
230         reg->r[instr->rd] = add_Check_Overflow(instr->im, - reg
->r[instr->ra], &carry);
231         update_PC(reg, 4, delay);
232         break;
233     }
234
235     case 0xB:
236     { /* RSUBIC 001011 */
237         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
238         reg->r[instr->rd] = add_Check_Overflow(instr->im, - reg
->r[instr->ra] + reg->c, &carry);
239         update_PC(reg, 4, delay);
240         break;
241     }
242
243     case 0xD:
244     { /* RSUBIK 001101 */
245         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
246         reg->r[instr->rd] = add_Check_Overflow(instr->im, - reg
->r[instr->ra], &carry);
247         reg->c = carry;
248         update_PC(reg, 4, delay);
249         break;
250     }
251
252     case 0xF:
253     { /* RSUBIKC 001111 */
254         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
255         reg->r[instr->rd] = add_Check_Overflow(instr->im, - reg
->r[instr->ra] + reg->c, &carry);
256         reg->c = carry;
257         update_PC(reg, 4, delay);
258         break;
259     }
260
261     case 0x5:
262     { /* RSUBK CMP 000101 */
263
264         /* last 8 bit 00000011 then cmpu */
265         instr = parse_instruction(instruction, TYPE_A, instr, &

```

```

reg->im);
266
267     if (instruction[3] == 0x0)
268     {
269         reg->r[instr->rd] = add_Check_Overflow(reg->r[instr->
rb], - reg->r[instr->ra], &carry);
270         reg->c = carry;
271     }
272
273     update_PC(reg, 4, delay);
274     break;
275 }
276 case 0x10:
277 { /* MUL 010000 */
278     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
279     reg->r[instr->rd] = reg->r[instr->ra] * reg->r[instr->rb
];
280     update_PC(reg, 4, delay);
281     break;
282 }
283
284 case 0x24:
285 { /* SRA 100100 */
286     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
287     reg->r[instr->rd] = u_to_f((f_to_u(reg->r[instr->ra]) >>
1));
288     reg->c = carry;
289     update_PC(reg, 4, delay);
290     break;
291 }
292
293 case 0x20:
294 { /* OR 100000 */
295     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
296     reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) |
f_to_u(reg->r[instr->rb]));
297     update_PC(reg, 4, delay);
298     break;
299 }
300
301 case 0x21:
302 { /* AND 100001 */
303     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
304     reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) &
f_to_u(reg->r[instr->rb]));
305     update_PC(reg, 4, delay);
306     break;
307 }

```

```

308
309     case 0x22:
310     { /* XOR 100010 */
311         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
312         reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) ^
f_to_u(reg->r[instr->rb]));
313         update_PC(reg, 4, delay);
314         break;
315     }
316
317     case 0x23:
318     { /* ANDN 100011 */
319         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
320         reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) &
(~f_to_u(reg->r[instr->rb])));
321         update_PC(reg, 4, delay);
322         break;
323     }
324
325     case 0x28:
326     { /* ORI 101000 */
327         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
328         reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) |
f_to_u(instr->im));
329         update_PC(reg, 4, delay);
330         break;
331     }
332
333     case 0x29:
334     { /* ANDI 101001 */
335         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
336         reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) &
f_to_u(instr->im));
337         update_PC(reg, 4, delay);
338         break;
339     }
340     case 0x2A:
341     { /* XORI 101010 */
342         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
343         reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) ^
f_to_u(instr->im));
344         update_PC(reg, 4, delay);
345         break;
346     }
347
348     case 0x2B:
349     { /* ANDNI 101011 */

```

```

350     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
351     reg->r[instr->rd] = u_to_f(f_to_u(reg->r[instr->ra]) &
(~f_to_u(instr->im)));
352     update_PC(reg, 4, delay);
353     break;
354 }
355
356 case 0x27:
357 { /* BEQ BGE BGT BLE BLT BNE 100111 */
358     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
359     delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
360     branch_type = conv_reg(instr->rd) & 0b00001111;
361     is_delayed = conv_reg(instr->rd) & 0b00010000;
362
363     if (branch_type == 0x0 && reg->r[instr->ra] == 0x0) /*
BEQ D0000 */
364         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
365     else if (branch_type == 0x5 && reg->r[instr->ra] >= 0x0)
/* BGE D0101 */
366         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
367     else if (branch_type == 0x4 && reg->r[instr->ra] > 0x0)
/* BGT D0100 */
368         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
369     else if (branch_type == 0x3 && reg->r[instr->ra] <= 0x0)
/* BLE D0011 */
370         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
371     else if (branch_type == 0x2 && reg->r[instr->ra] < 0x0)
/* BLT D0010 */
372         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
373     else if (branch_type == 0x1 && reg->r[instr->ra] != 0x0)
/* BNE D0001 */
374         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
375     else
376         update_PC(reg, 4, delay);
377
378     // if (is_delayed == 0x10) /* delayed slot */
379     //     run_instruction(instructions[delayed_instruction],
data, reg, instructions, true);
380     break;
381 }
382
383 case 0x2F:
384 { /* BEQI BGEI BGTI BLEI BLTI BNEI 101111 */
385     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
386     delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
387     branch_type = conv_reg(instr->rd) & 0b00001111;
388     is_delayed = conv_reg(instr->rd) & 0b00010000;

```

```

389
390     if (branch_type == 0x0 && reg->r[instr->ra] == 0x0) /*
BEQI D0000 */
391         update_PC(reg, f_to_i(instr->im), delay);
392     else if (branch_type == 0x5 && reg->r[instr->ra] >= 0x0)
/* BGEI D0101 */
393         update_PC(reg, f_to_i(instr->im), delay);
394     else if (branch_type == 0x4 && reg->r[instr->ra] > 0x0)
/* BGTI D0100 */
395         update_PC(reg, f_to_i(instr->im), delay);
396     else if (branch_type == 0x3 && reg->r[instr->ra] <= 0x0)
/* BLEI D0011 */
397         update_PC(reg, f_to_i(instr->im), delay);
398     else if (branch_type == 0x2 && reg->r[instr->ra] < 0x0)
/* BLTI D0010 */
399         update_PC(reg, f_to_i(instr->im), delay);
400     else if (branch_type == 0x1 && reg->r[instr->ra] != 0x0)
/* BNEI D0001 */
401         update_PC(reg, f_to_i(instr->im), delay);
402     else
403         update_PC(reg, 4, delay);
404
405     // if (is_delayed == 0x10) /* delayed slot */
406     // run_instruction(instructions[delayed_instruction],
data, reg, instructions, true);
407     break;
408 }
409
410 case 0x26:
411 { /* BR BRD BRA BRLD BRAD BRALD 101110 */
412     instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
413     delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
414
415     /* DAL00 */
416     is_absolute = conv_reg(instr->ra) & 0b00001000; /* A for
branch with absolute PC = rb */
417     is_link = conv_reg(instr->ra) & 0b00000100; /* L for
branch and link rd = PC */
418     is_delayed = conv_reg(instr->ra) & 0b00010000;
419
420     if (is_link == 0x4)
421         reg->r[conv_reg(instr->rd)] = reg->pc * 4;
422
423     if (is_absolute == 0x8)
424         reg->pc = (reg->r[conv_reg(instr->rb)]) / 4;
425     else
426         update_PC(reg, f_to_i(reg->r[instr->rb]), delay);
427
428     break;
429 }

```



```

430     case 0x2E:
431     { /* BRI BRAI BRID BRAID BRLID BRAILD 101110 */
432         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
433         delayed_instruction = reg->pc + 1; /* next instruction
for delayed branch*/
434
435         /* DAL00 */
436         is_absolute = conv_reg(instr->ra) & 0b00001000; /* A for
branch with absolute PC = rb */
437         is_link = conv_reg(instr->ra) & 0b00000100;      /* L for
branch and link rd = PC */
438         is_delayed = conv_reg(instr->ra) & 0b00010000;
439
440         if (is_link == 0x4)
441             reg->r[conv_reg(instr->rd)] = reg->pc * 4;
442
443         if (is_absolute == 0x8)
444             reg->pc = f_to_i(instr->im) / 4;
445         else
446             update_PC(reg, f_to_i(instr->im), delay);
447
448         break;
449     }
450
451     case 0x32:
452     { /* LW 110010 */
453         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
454         addr = f_to_u(reg->r[instr->ra]) + f_to_u(reg->r[instr
->rb]);
455         reg->r[instr->rd] = data->data[addr];
456         update_PC(reg, 4, delay);
457         break;
458     }
459     case 0x36:
460     { /* SW 110110 */
461         instr = parse_instruction(instruction, TYPE_A, instr, &
reg->im);
462         addr = f_to_u(reg->r[instr->ra]) + f_to_u(reg->r[instr->
rb]);
463         data->data[addr] = reg->r[instr->rd];
464         update_PC(reg, 4, delay);
465         break;
466     }
467
468     case 0x3A:
469     { /* LWI 111010 */
470         instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
471         addr = f_to_u(reg->r[instr->ra]) + f_to_u(instr->im);
472         reg->r[instr->rd] = data->data[addr];

```

```

473     update_PC(reg, 4, delay);
474     break;
475 }
476
477 case 0x3E:
478 { /* SWI 111110 */
479     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
480     addr = f_to_u(reg->r[instr->ra]) + f_to_u(instr->im);
481     data->data[addr] = reg->r[instr->rd];
482     update_PC(reg, 4, delay);
483     break;
484 }
485
486 case 0x2C:
487 { /* IMM 101100 */
488     instr = parse_instruction(instruction, TYPE_B, instr, &
reg->im);
489     reg->im = (uint16_t)f_to_u(instr->im);
490     update_PC(reg, 4, delay);
491     break;
492 }
493
494 default:
495 {
496     /* unknown instruction */
497     break;
498 }
499 }
500 }
501
502 void vadd(struct Memory *mem, struct Registers *reg,
uint32_t *out, ap_uint<32> my_size)
503 {
504 #pragma HLS INTERFACE m_axi port = mem bundle = gmem
505 #pragma HLS INTERFACE m_axi port = reg bundle = gmem
506 #pragma HLS INTERFACE m_axi port = out bundle = gmem
507 #pragma HLS INTERFACE ap_ctrl_hs port = return
508
509     struct Registers reg_copy;
510     struct Memory mem_copy;
511
512     for (int i = 0; i < 32; i++)
513         reg_copy.r[i] = reg->r[i];
514
515     reg_copy.c = reg->c;
516     reg_copy.i = reg->i;
517     reg_copy.pc = reg->pc;
518     reg_copy.im = reg->im;
519
520     for (int i = 0; i < 1024; i++)
521         mem_copy.data[i] = mem->data[i];

```

```

522
523     for (int i = 0; i < 32; i++)
524         for (int j = 0; j < 4; j++)
525             mem_copy.instr[i][j] = mem->instr[i][j];
526
527     struct Registers *reg_copy_pointer = &reg_copy;
528     struct Memory *mem_copy_pointer = &mem_copy;
529
530     while (reg_copy_pointer->pc < my_size)
531         run_instruction(mem_copy_pointer->instr[reg_copy_pointer
532             mem_copy_pointer,
533             reg_copy_pointer,
534             mem_copy_pointer->instr,
535             false);
536
537     out[1] = f_to_i(reg_copy_pointer->r[1]);
538 }
539 }

```

A.5 File Assembler

• add.s

```

1  .text
2  .align 2
3  .globl main
4  .ent main
5  .type main, @function
6
7  main:
8      addi r2,r0,55          #r2 = 55
9      addi r3,r0,100         #r3 = 100
10     add r4,r2,r3           #r4 = 155
11     addi r5,r0,2147483640   #r5 = 2147483640
12     addi r6,r0,8           #r6 = 8
13     addk r7,r5,r6          #r7 = 2147483640 + 8 c =1
14     addic r1,r4,1          #r1 = 155 + 1 + 1 = 157
15     addi r1,r1,2           #r1 = 157 + 2 = 159
16
17     .end ma

```

• branch.s

```

1  .text
2  .align 2
3  .globl main
4  .ent main

```

```

5  .type main, @function
6
7  main:
8      addi  r3,r0,8
9      addi  r2,r0,-8
10     addi  r1,r0,0
11     addi  r1,r1,1
12     addi  r3,r3,-1
13     bge  r3,r2
14     nop
15     addi  r3,r0,8
16     addi  r1,r1,1
17     addi  r3,r3,-1
18     bgei  r3,-8
19     nop
20     addi  r3,r0,8
21     addi  r1,r1,1
22     addi  r3,r3,-1
23     bgeid r3,-8
24     addi  r1,r1,1 #r1=18+9+9+1=37
25     nop
26     addi  r3,r0,-8
27     addi  r2,r0,-8
28     addi  r1,r1,1
29     addi  r3,r3,1
30     ble  r3,r2  #r1 = 37+9 = 46
31
32     .end  main

```

• data.s

```

1  .text
2  .align 2
3  .globl main
4  .ent  main
5  .type main, @function
6
7  main:
8      addi  r2,r0,55
9      addi  r3,r0,111
10     sw      r3,r2,r0
11     nop
12     nop
13     nop
14     lw      r4,r2,r0  #r4 = 111
15     addi  r7,r0,88
16     swi     r7,r0,99
17     nop
18     nop
19     nop

```

```

20      add    r3,r0,r0
21      lwi    r3,r0,99
22      add    r1,r3,r4    #111+88 = 199
23
24
25      .end    main

```

• sub.s

```

1      .text
2      .align  2
3      .globl  main
4      .ent    main
5      .type   main, @function
6
7  main:
8      addi    r2,r0,-55        #r2 = -55
9      addi    r3,r0,885       #r3 = 885
10     rsub    r4,r3,r2        #r4 = r2-r3 = -55-885 = -950
11     rsubi   r1,r4,88        #r1 = 88 - (- 850) = 1038
12     addi    r7,r0,-2147483647
13     addi    r8,r0,4
14     rsubc   r2,r8,r7        #r2= r7-r8 = -2147483647 - 4 =
        underflow = -2147483648
15     add     r1,r1,r2        #r1=r2+r1 = -2147482620
16
17     .end    main

```

• bitop.s

```

1      .text
2      .align  2
3      .globl  main
4      .ent    main
5      .type   main, @function
6
7  main:
8      addi    r2,r0,2999
9      addi    r3,r0,23939
10     and     r1,r2,r3        #2435
11     addi    r4,r0,112231
12     or      r1,r1,r4        #114663
13     sra     r1,r1           #57331
14     add     r4,r1,r0
15     andn    r1,r2,r3        #564
16     xor     r1,r1,r4        #56775
17     ori     r1,r1,99999     #122847
18     andi    r1,r1,232      #200

```

19

20 `.end main`