ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Controllo Numerico con Sviluppo FPGA

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Marco Negrini

Dicembre
2017/2018

Capitolo 1

Introduzione

Le macchine a Controllo Numerico Computerizzato sono molto usate nell' automazione industriale. La funzione più importante è il controllo del movimento. L' implementazione del software che gestisce queste macchine dipende dal tipo di processore che lo esegue, infatti lo sviluppo delle macchine a controllo numerico va di pari passo con quello dei computer.

Grazie allo sviluppo di soluzioni Open Surce in questo campo possiamo notare diversi approcci per l' architettura software. Le comunità interessate alla stampa 3D hanno optato per micro-controllori dato il loro basso costo, mentre per le macchine per la fresa storicamente viene usato un computer con un sistema operativo completo. Stampa 3D e frese, anche se compiono operazioni opposte, sono entrambe controllate numericamente, e quindi il software è lo stesso per entrambe.

I micro-controllori sono economici e possono essere usati per efficaci soluzioni stand-alone, ma faticano a gestire la complessità di interfaccie di rete o interfaccie video. Dall' altra parte, computer e sistemi operativi hanno kernel multi-thread e driver di rete e video, ma devono comunicare con controllori per la gestione continuata di eventi.

In particolare il progetto Machinekit: funziona su un sistema operativo GNU/Linux, ottenendo i vantaggi dei linguaggi di alto livello come python. Attualmente richiede piattaforme specifiche per interfacciarsi con i motori,

ma è possibile aggiungere nuove piattaforme.

Questa tesi si pone l' obiettivo di sviluppare una possibile piattaforma alternativa. Ed è stato posto il vincolo di utilizzare solo strumenti Open Source.

Indice

1 Introduzione **i**

2 Motion Control **1**

2.1 Automazione 1

2.1.1 Storia 1

2.2 Il progetto EMC 4

2.3 Il progetto RepRap 5

2.4 Interfaccia 6

3 La tool-chain NC **9**

3.1 Interprete 10

3.1.1 G-Code 10

3.2 Path planner 12

3.3 Interpolazione 18

4 Machinekit **19**

4.1 Hardware Abstraction Layer 20

4.2 La tool-chain NC in Machinekit 22

4.2.1 Interprete 22

4.2.2 Il componente Motion 22

4.3 Generazione dei passi 23

4.3.1 Device dedicati 24

4.3.2 Device Connessi 25

4.3.3 Il Progetto Klipper 25

4.4 SPI Slave Implementato in FPGA 25

5 FPGA Development **27**

5.1 The code structure 28

5.1.1 SPI Slave 28

5.1.2 FIFO Queue 30

5.1.3 Pulse generator 31

5.1.4 Compiling 32

5.2 Verifica 34

6 Conclusioni **37**

Bibliografia **39**

Elenco delle figure

3.1 Il percorso risultante 12
3.2 Profilo di velocità trapezoidale 13
3.3 Profilo di accelerazione cubica 14
3.4 path-planner punto a punto 15
3.5 path-planner continuo 16
3.6 Angolo arrotondato 16
3.7 Approssimazione di una curva 18

4.1 Struttura di Machinekit 23

5.1 Structura di un SPI-Slave 29
5.2 Gtkview dei segnali interni 35

Capitolo 2

Motion Control

In ambito industriale sono sempre più comuni macchine operate da controllori numerici. Il Controllo Numerico è un ramo dell' automazione ed è applicato ogni volta che è necessario monitorare la posizione di uno strumento avendo cura di aspetti come velocità o accelerazione.

Verrà qui descritta l' evoluzione di queste macchine, partendo dai primi automatistimi, fino ai controlli attuali.

2.1 Automazione

2.1.1 Storia

I primi controlli automatici

L' automazione è definita come la creazione di tecnologia con lo scopo di controllare e monitorare la produzione di vari beni e servizi. Per quanto riguarda la produzione industriale di beni, l' automazione ha subito un evoluzione da meccanica a elettronica, e ad oggi si utilizzano software per il controllo automatico. I primi controlli automatici, meccanici, sono stati usati per mantenere determinate valori all' interno di un intervallo. Questi valori erano per esempio temperatura, pressione, o velocità. I controlli possono essere divisi in due categorie: a circuito aperto o circuito chiuso.

Nei controlli a circuito aperto le decisioni prese dal controllore sono completamente indipendenti dallo stato attuale, o dal risultato effettivo. Sono tipicamente decisioni prese a priori, come per esempio un timer.

Quando il controllore ha un feed-back del risultato delle proprie azioni si dice che è a circuito chiuso, questo feed-back viene usato per correggere errori che non possono essere previsti a priori. Durante la storia son stati costruiti controllori meccanici a circuito chiuso, un esempio è il regolatore centrifugo usato per regolare il flusso di carburante nei primi motori. Il regolatore gestisce il flusso proporzionalmente alla velocità del motore.

L'inizio del ventesimo secolo vede un largo uso di controllori a circuito chiuso per voltaggio, corrente e frequenza [1]. Il mantenimento di questi, o altri, valori nell' intervallo desiderato è possibile utilizzando una semplice regola basata su tre parametri: proporzionale, integrativo e derivativo. Un' analisi teorica di questo controllo è stata presentata da Nicholas Minorsky (1885-1970) nel 1922.

Il controllo Proporzionale-Integrativo-Derivativo, abbreviato spesso con PID, è un controllo continuo e modulato, ad ogni istante di tempo calcola l' errore tra il valore impostanto e quello misurato, e applica una correzione basata sui termini proporzionale, integrale e derivativo. Lo pseudo-codice che rappresenta questo tipo di controllo è molto semplice e può essere descritto come mostrato nell' Algoritmo 1.

La teoria dietro il PID è arrivata a completezza negli anni 50, un controllo PID può essere implementato facilmente su una scheda elettronica, scegliendo i componenti in base ai coefficienti dei tre termini che si vogliono ottenere.

Micro-controllori

I processori diventarono popolari nelle industrie negli anni 80. I primi controllori dotati di processore sono stati i Controllori Logici Programmabili (in inglese Programmable Cogic Controllers, da cui PLC).

I PLC sono comparsi alla fine degli anni 60, il motivo principale della loro adozione è stato il costo di manutenzione, molto inferiore a quello dei, allora

Algorithm 1 Controllo PID

1: **procedure** PID(Kp, Ki, Kd)
2: while True:
3: *value* ← *measure()*
4: *error* ← *setpoint - value*
5: *accumulation error* ← *accumulation error + error * delta time*
6: *derivative error* ← *(error - last error) / delta time*
7: *last error* ← *error*
8: *proportional* ← *error * Kp*
9: *integral* ← *accumulation error * Kp*
10: *derivative* ← *derivative error * Kd*
11: *last error* ← *error*
12: *output value* ← proportional + integral + derivative
13: *set(output value)*

attuali, sistemi di relè.

Da allora una lenta, ma continua, crescita ha permesso alle industrie manufatturiere di utilizzare i PLC. Traendone vantaggi perché programmabili con un software che assomiglia molto alla logica dei precedenti rele, con cui i tecnici della manutenzione erano già familiari. Inoltre, un singolo PLC, inteso come prodotto, può essere prodotto in massa, riducendone il costo di produzione. Il compito di un PLC risulta molto semplice per un processore, tanto che quando è stato possibile si è passati a micro-controllori, perché più piccoli, economici, ma comunque capaci di gestire vari ingressi e uscite.

Utilizzare micro-controllori permette di trasformare in software una complicata rete di contatti, ma non solo. È anche possibile implementare comunicazioni seriali che portano a sistemi di controllo distribuiti e controllo dei processi produttivi. Si semplifica anche l' interfaccia uomo-macchina, grazie a interfacce video, impossibili con controlli solamente meccanici o elettronici.

Una tecnica interessante utilizzata per interfacciare circuiti digitali con segnali analogici è la modulazione dell' ampiezza tramite impulsi (in inglese

Pulse Width Modulation, da cui PWM), che permette di produrre un segnale analogico da un' uscita digitale. I segnali analogici hanno un intervallo continuo di possibili valori, mentre quelli digitali possono assumere solo valori discreti. In particolare possono assumere solo due possibili valori, zero e uno. Per produrre un valore intermedio è possbile generare degli impulsi, abbastanza velocemente, regolando il rapporto del tempo speso tra acceso e spento.

PWM e PID sono implementabili molto facilmente su un micro-controllore. Il PWM è nato per regolare la potenza o la velocità dei motori, e la disponibilità di controllori capaci di eseguire sequenze di comandi ha migliorato molto le macchine a Controllo Numerico.

2.2 Il progetto EMC

Il progetto EMC (Enhanced Motion Controller, dall' inglese Controllore Numerico Avanzato) è un progetto spinto dal governo americano alla fine degli anni 80[2]. Mira a "rivitalizzare l' industria americana delle macchine utensili che stava perdendo mercato contro la concorrenza straniera, principalmente tedesca e giapponese".

L' istituto nazionale degli standard e della tecnologia americano (NIST, National Institute of Standards and Technology) iniziò a lavorare sul progetto EMC negli anni 90. I lavori iniziarono da una libreria di message-passing chiamata Neutral Message Language che gestisce le comunicazioni interne del programma. La prima implementazione di EMC è stata completata nel 1996, operava utilizzando due computer windows, uno dei quali agiva da controllore mentre l' altro da semplice interfaccia. Inizialmente era un controllore capace di gestire solo una specifica macchina utensile.

Siccome EMC è un "progetto finanziato dal governo americano, esso non è soggetto a copyright ed è di dominio pubblico", e venne pubblicato quando Open Engineering ne chiese il codice.

Open Engineering adattò EMC per una diversa macchina utensile, rimuovendo molto codice specifico, e facendo girare sia il controllo che l' interfaccia sullo stesso computer. A seguito ci furono vari adattamenti che culminarono con una versione completa di EMC eseguibile su Linux, utilizzando un kernel real-time, ed eliminando il bisogno di periferiche esterne per il controllo dei motori.

EMC utilizzava NML per le sue comunicazioni interne. Questa eredità rimane ancora oggi, nonostante NML non sia più mantenuto. Ci si sta comunque spostando verso librerie più moderne come ZMQ o DDS.

La parte più importante del progetto EMC è un livello di astrazione hardware. Il quale permette di modularizzare il codice, semplificandone la gestione e la separazione tra codice real-time e non. Lo sviluppo di questa astrazione iniziò nel 2003, ed è ancora oggi in sviluppo.

Nel 2011 EMC cambiò nome in LinuxCNC per ragioni di copyright, e nel 2015 da LinuxCNC nasce il progetto Machinekit che si pone gli obiettivi di portare il codice su piattaforme ARM e di migliorare il livello di astrazione hardware.

Verrà usato il termine Machinekit per indicare il progetto nato come EMC e continuato dal 2015 col nome Machinekit.

2.3 Il progetto RepRap

La stampa 3D è un' applicazione comune del controllo numerico. Le comunità di stampa 3d hanno sviluppato i loro controllori numerici per piattaforme molto economiche.

Nel 2004 nasce il progetto RepRap, che si pone l' obiettivo di costruire una macchina capace di auto-riprodursi. Come primo passo si decise di costruire una stampante 3D capace di produrre i propri pezzi. Ciò richiede ancora assemblaggio manuale e aggiunta di pezzi (come per esempio il micro-controllore) che non è possibile stampare. Ogni pezzo non stampabile doveva essere il più economico possibile, per questo non era possibile gestire

la macchina con computer ma era necessario portare il software di controllo su un micro-controllore.

Il progetto nasce anche grazie alla scheda di sviluppo Arduino, un hardware open source, che nasceva proprio in quegli anni.

Grazie al progetto RepRap il costo di una stampante 3D è calato da qualche decina di migliaia di euro a qualche centinaia. Questo preché grazie a un progetto completamente aperto chiunque poteva costruirne una.

La disponibilità di un software liberamente modificabile ha portato alla nascita di vari controllori, i principali sono GRBL, Marlin e RepRap Frimware.

2.4 Interfaccia

Un micro-controllore permette di implementare un controllore numerico funzionante. negli ultimi anni si sta passando da arduino (a 8 bit) ad altri micro-controllori (a 32bit), ma mantenendo lo stesso software.

La parte problematica dei controllori numerici basati su micro-controllori è l' interfaccia uomo-macchina. Da questa interfaccia le operazioni che ci si aspetta sono:

- trasmettere il file con i comandi (chiamato file G-Code)
- ricevere aggiornamenti sullo stato della macchina e inviare commandi aggiuntivi

In una comune stampante 3d, governata da un ATMEGA2560, le possibili interfaccie che troviamo sono:

- un piccolo schermo LCD a matrice
- alcuni pulsanti
- uno slot per una scheda SD
- una porta USB

Per la trasmissione del file G-Code le opzioni quindi sono:

- scriverlo su una scheda SD e spostare la scheda
- inviare il file tramite USB

Scheda SD

La prima opzione è scrivere il file G-Code su una scheda SD e muovere fisicamente la scheda tra computer e stampante. La scheda SD è abbastanza grande per memorizzare almeno qualche file G-code

Questa soluzione è chiamata "stampante standalone" preché tutta l' interfaccia è presente presso la stampante, gli update vengono comunicati a schermo, e comandi aggiuntivi possono essere dati utilizzando i pulsanti. È un' interfaccia molto semplice, ma economica ed efficace.

communicazione via USB

La comunicazione USB permette un' interfaccia molto più completa, sopratutto perché è possibile collegare un computer e avere visualizzazione a schermo e un' interfaccia di rete. Un programma che fa proprio questo è Repetier-Server. Esso gestisce la comunicazione via USB con il microcontrollore e la utilizza per mandare il file G-Code o commandi aggiuntivi, e ricevere aggiornamenti sullo status. Questa soluzione è chiamata "tether".

Questa soluzione ha alcuni svantaggi: il computer è dedicato alla comunicazione con la stampante, se la connessione tra i due si interrompe la stampa fallisce. Ciò può accadere perché il computer si spegne o perché il cavo si può staccare. Per questa ragione spesso viene utilizzato un single board computer, economico, e dedicato a fare da interfaccia per la stampante. Il Raspbperry Pi è un ottimo single board computer per questo scopo.

Capitolo 3

La tool-chain NC

Il termine Controllo Numerico Computerizzato (CNC) si riferisce all' utilizzo di computer in applicazioni di controllo numerico. Storicamente il CNC è stato applicato principalmente a frese, ma la stessa tecnologia può essere applicata in molti altri casi. Per fare alcuni esempi, stampanti 3d o robot in generale richiedono lo stesso tipo di controllo.

Il termine Controllore Numerico (che in inglese diventa Numeric Controller, ca cui NC) verrà utilizzato per indicare il software che riceve i comandi e in base ad essi comanda un certo numero di motori. I comandi possono essere dati in molti modi, verranno cosiderati solo i NC che ricevono i comandi sotto forma di file G-Code, dato che è lo standard per stampanti 3d e frese.

Il NC può essere visto come una una tool-chain, una catena di diversi componenti software. Il loro compiti spaziano dalla lettura dei comandi, alla loro esecuzione, il tutto tenendo conto della configurazione della macchina, come è strutturata, i limiti di spazio, velocità e accelerazione che possono essere imposti.

La tool-chain è composta da:

- interprete
- path planner
- controllore del movimento

3.1 Interprete

L' interprete è un parser che scorre il file G-Code e decodifica i comandi. Il G-Code è un linguaggio di programmazione per controllo numerico, descrive tutte le possibile istruzioni che possono essere date alla macchina. Siccome esistono molti tipi di macchine automatiche il G-Code non ha uno standard universale, anzi, spesso vengono definiti comandi dedicati per la specifica macchina. Qui verrà descritto lo standard adottato da Machinekit[3].

3.1.1 G-Code

Il linguaggio G-Code è basato su linee di codice. Ogni linea (anche chiamata blocco) descrive un comando, che può anche descrivere istruzioni complesse. Un file, o programma, è un file G-Code valido se tutte le linee rispettano le specifiche definite.

L'interprete riceve quindi una lista di comandi G-Code, sotto forma di file, e deve capire quale istruzione è descritta in ciascuna linea. Successivamente ogni istruzione deve essere tradotta in una comunicazione interna con i successivi componenti della tool-chain che eseguiranno i comandi.

Questa è la struttura di un comando:

```
<Tipo di comando><Numero del comando>
{<parametro><valore>}
```

La prima parte indica il tipo del comando, è sempre una singola lettera, i tipi più comuni sono G e M, rispettivamente il tipo Generico e Misto. La coppia tipo e numero definisce univocamente un comando. I comandi di ogni tipo sono divisi in gruppi. I gruppi dei comandi generici possono essere, per esepio, comandi di movimento, o comandi di referenza. I comandi misti supportano le azioni di supporto, come rotazione, colata, o definiti dall' utente. A seguito del comando c' è una lista di coppie parametro e valore. Il significato di questi parametri dipende dallo specifico commando. In Machinekit

i comandi M accettano solo i parametri P e Q, mentre i comandi G possono avere un numero indefinito di parametri.

Un numero è riconosciuto come una sequenza di caratteri tra 0 e 9, con le seguengti proprietà¿

- può essere riconosciuto da

  ```
  [+|-]{0-9}[.]{0-9}
  ```

- ci sono due tipi di numeri, interi e reali. il tipo di un numero è definito dalla presenza (o assenza) della virgola
- l'unico limite alla lunghezza di un numero è la lunghezza massima del blocco.
- Un numero non nullo senza segno si assume essere positivo

All' interno del G-Code è anche possibile dichiarare variabili, l'unico tipo supportato sono i numeri reali. In ogni caso è possibile formulare espressioni con valori logici e valutarle utilizzando operatori di confronto.

Questo è un esempio di file G-code

```
G92
G0 X2 Y1 F1000
G0 X2 Y3 F1000
G0 X3 Y2 F200
M2
```

L' immagine 3.1 mostra il percorso risultante dai comandi G-Code dati in esempio: G92 è un comando che imposta la posizione attuale alle coordinate nulle; G0 è il comando usato per muovere la macchina in linea retta verso le coordinate comandate (ogni coordinata non specificata si assume resti costante).

Nelle prime implementazioni (di Machinekit) il NC funzionava in modalità **punto-a-punto**, il path planner gestiva un comando alla volta, fermandosi completamente al termine di ognuno. In questo modo tutti i componenti

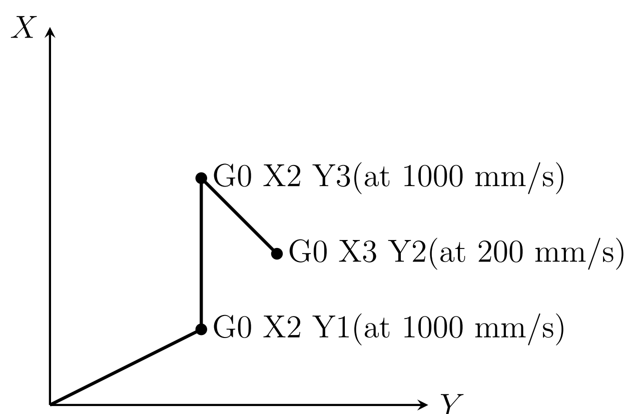

Figura 3.1: Il percorso risultante

della tool-chain erano sincronizzati: l' interprete legge un comando, attente che venga eseguito, e poi passa al comando successivo.

Ma questo limita il path-planner che può essere migliorato se potesse vedere avanti di qualche comando (questo verrà spiegato meglio nella prossima sezione). Oggi il path-planner è utilizzato in modalità **continua**, proprio per evitare di doversi fermare dopo ogni istruzione.

Quindi l'interprete deve semplicemente interpretare il comando e passarlo avanti, e non sa a che punto dell' esecuzione sia il path-planner, o dove sia posizionata la macchina.

3.2 Path planner

Il path-planner, o pianificatore della traiettoria, riceve dall' interprete le istruzioni, indicanti la posizione comandata.

Per spostarsi da una posizione all' altra la macchina deve accelerare e muoversi a una determinata velocià e accelerazione, entrambe potrebbero dover rispettare dei limiti, e il path-planner deve tenerne conto e regolare il movimento di conseguenza.

Il path-planner è un argomento complesso, che non verrà trattato nello specifico, verranno solo descritte alcune solo possibili soluzioni per mostrare

il tipo di calcoli che devono essere eseguiti.

Accelerazione

La soluzione più semplice è quella punto-a-punto, preché non c'è look-ahead. Durante il movimento il path planner deve considerare i limiti imposti alla posizione, velocità, accelerazione, e strappo (derivate dell' accelerazione nel tempo).

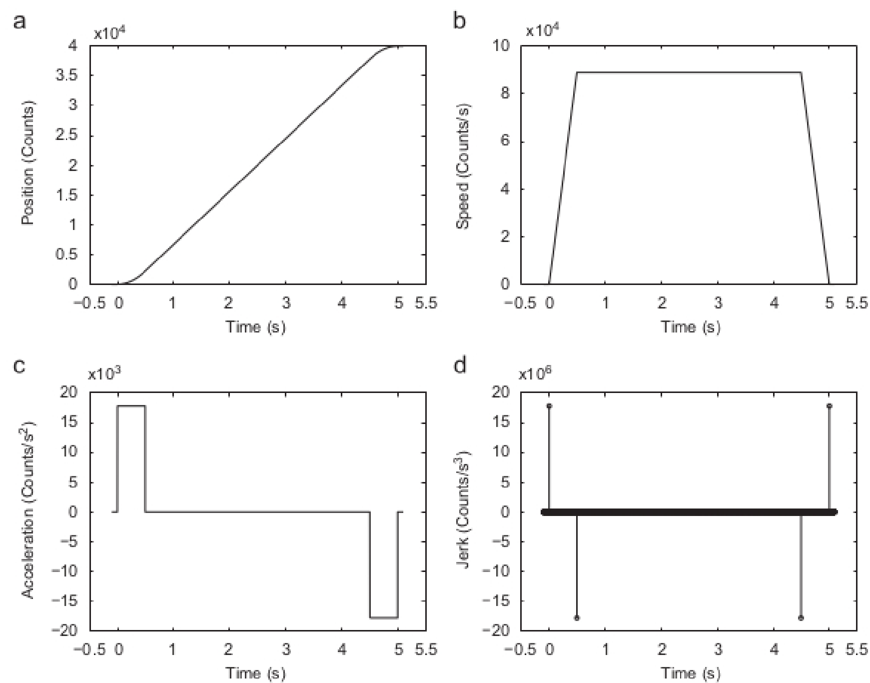

Figura 3.2: Profilo di velocità trapezoidale

Per esempi, l' immagine 3.2 mostra un profilo di traiettoria a **velocità trapezoidale**[5]. La macchina accelera al massimo possibile fino alla velocità massima, e decellera il più velocemente possibile appena necessario. Questo profilo viene usato in molti controllori commerciali, e viene usato come confronto con profili di accelerazione di grado maggiore. L'immagine 3.2a mostra il profilo della posizione, l' immagine 3.2b il pofilo di velocità

trapezoidale, l' immagine 3.2c il profilo di accelerazione e l' immagine 3.2d il profilo dello strappo. È evidente che, anche se la traiettoria della posizione sembra liscia e regolare, l' accelerazione ha delle discontinuità, che nel profilo dello strappo hanno chiari impulsi che stressano la meccanica.

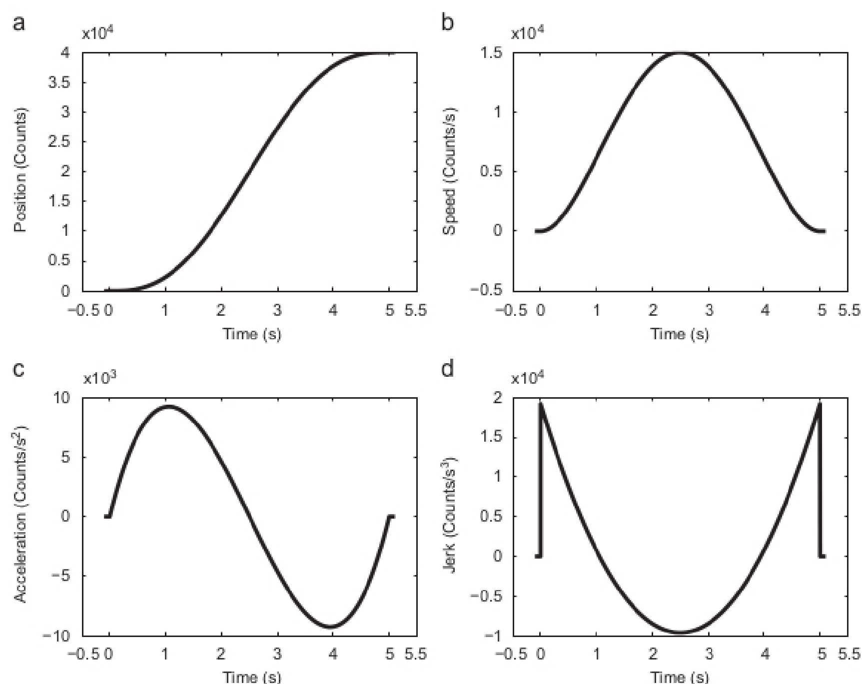

Figura 3.3: Profilo di accelerazione cubica

In confronto l' immangine 3.3 mostra un profilo di **accelerazione cubica**. Il profilo dell' accelerazione mostrato in figura 3.3c è una curva di terzo grado, da ciò si deduce che lo strappo è desrcitto da una curva di secondo grado, e la posizione sarà un profilo di quinto grado. È possibile notare che tutti i profili sono lisci, senza disconitnuità, ed è possibile anche imporre limiti sullo strappo.

Arrotondamento degli angoli

Qui verrà mostrata la differenza tra le modalità punto-a-punto e continua. Il risultato di un path-planner punto-a-punto può essere visto in figura 3.4, si può notare che allàngolo tra i due segmenti la machina si ferma completamente per poi ripartire. Si può notare che il profilo di velocità è trapezoidale.

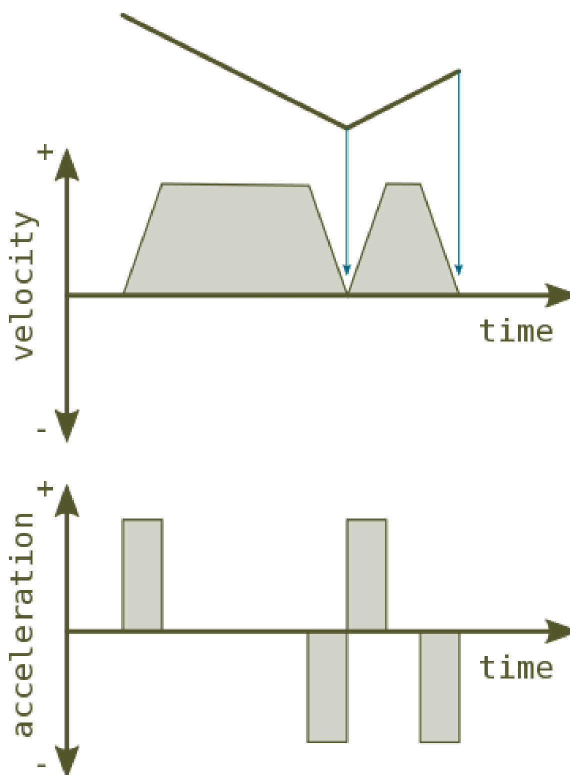

Figura 3.4: path-planner punto a punto

In confronto la figura 3.5 mostra il risultato di un path-planner continuo, l' angolo è arrotondato e non c'è sosta tra un segmento e l' altro.

L' arrotondamento dell' angolo può essere calcolato in vari modi. L' angolo potrebbe essere sia tra due segmenti che tra due movimenti circolari, visto che il G-Code permette di comandare anche quel movimento.

L' arco che sostituisce l' angolo può separarsi da esso di una certa massima distanza. L' arrotondamento non ha effetto solo sui segmenti che lo precedono

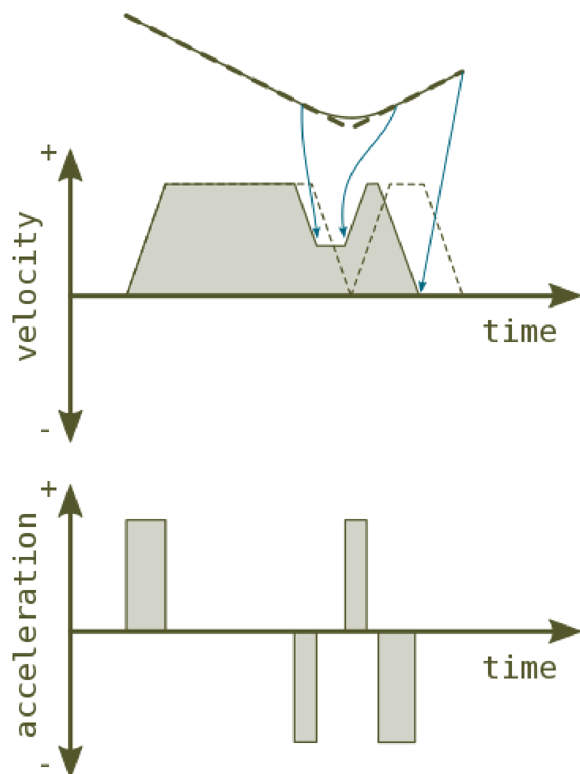

Figura 3.5: path-planner continuo

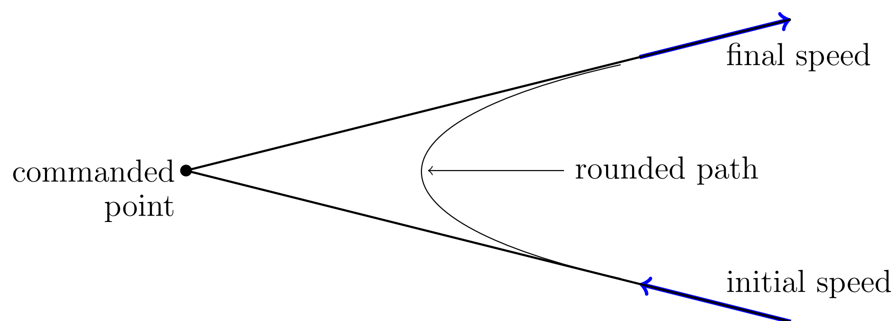

Figura 3.6: Angolo arrotondato

o seguono, sopratutto se i segmenti sono molto corti. Per questa ragione in path-planner deve avere un look-ahead di qualche segmento, per verificare che l' arrotondamento sia conforme con le istruzioni che seguono.

Esistono quindi due diverse posizioni, una è la posizione reale della macchina, e l' altra è quella virtuale raggiunta dall look-ahead del path-planner.

Ciascuna di queste può essere gestita da un diverso thread: la posizione reale richiede le garanzie di uno scheduling real-time; mentre il thread del path-planner non richiede un esecuzione a un tempo preciso.

Sistema di riferimento

Fin qui abbiamo considerato i movimenti in un sistema di riferimento cartesiano, ma molte macchine sono costruite su diversi sistemi di riferimento, o cinematiche.

Il G-Code astrae da questa configurazione, e si riferisce sempre a coordinate in spazione cartesiano.

Per riferirsi alle coordinate in spazio cartesiano e in quello specifico della macchina si usano due termini:

- si indica con **asse** la posizione lungo una delle coordinate in spazio cartesiano.

- si indica con **coppia cinematica** la posizione relativa tra due parti della macchina capaci di muoversi tra di loro

Ed esistono due tipi di trasformazione:

- in avanti: da coppia cinematica ad asse

- all' indietro: da asse a coppia cinematica

La trasformazione all' indietro è applicata dal path-planner, per calcolare il movimento di ogni singolo motore, responsabile di una coppia cinematica. La trasformazione in avanti è appicata per calcolare la posizione reale attuale. Il path-planner deve occuparsi anche di controllare che vengano rispettate velocità e accelerazione massime per ogni singola coppia cinematica, e non sono globalmente.

Esistono molti algoritmi possibili per risolvere questi problemi di path-planning, che però non sono argomento di questa tesi. La parte fondamentale

è la descrizione delle operazioni necessarie, dal punto di vista dell' architettura del NC: tutte queste operazioni richiedono operazioni floating-poing e sono abbastanza parallelizzabili.

3.3 Interpolazione

Il path-planner continuo calcola le curve ottimali, che la macchina cercherà di seguire. Alcune di queste curve risulteranno essere semplicemente dei segmenti, altre saranno semplici formule geometriche, e altre saranno definibili solamente tramite approssimazione. In ogni caso il problema fondamentale nel seguire queste curve è che sono continue, mentre il controllore digitale è discreto.

Per percorrere un percorso circolare, la macchina deve dividere il cerchio in una serie di brevi tratti che approssimano la curva, come mostrato dalla figura 3.7.

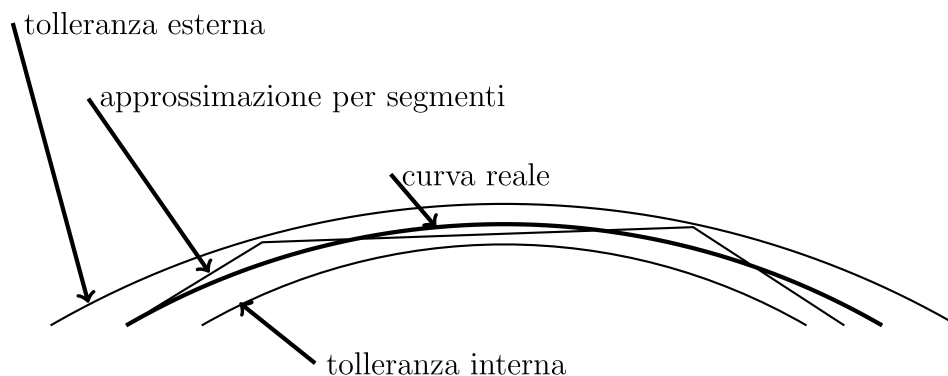

Figura 3.7: Approssimazione di una curva

Capitolo 4

Machinekit

Machinekit è un controllore general purpose, ovvero configurabile per gestire qualsiasi tipo di macchina. È configurabile tramite una logica a blocchi che ricorda molto la progettazione di schede elettroniche. I blocchi sono fatti per essere riutilizzabili e compatibili tra di loro. I blocchi vengono eseguiti sopra a un Livello di Astrazione Hardware (in inglese Hardware Abstraction Layer, da cui HAL), che si occupa di instanziarli, eseguirli, e farli comunicare tra di loro. Sia l' esecuzione che la comunicazione possono avvenire in real-time o in user space, a seconda di come è implementato il modulo.

Il processore non può interfacciarsi direttamente con in contatti di ingresso o uscita dei segnali, per ragioni di velocità e tempi di reazione. Un kernel real time permette di schedulare periodicamente i thread importanti, ma la gestione dei contatti deve essere fatta da micro-controllori più reattivi. Il micro-controllore può essere programmato per eseguire semplicemente le azioni comandate da Machikeit, o pu'ò ricevere istruzioni che richiedono elaborazione. Solitamente si cerca di tenere tutta la logica di controllo all' interno di Machinekit.

4.1 Hardware Abstraction Layer

È stato scelto Machinekit invece di altri controllori numerici per via di HAL: la modularità che offre semplifica il riutilizzo e la manutenzione del codice.

L' obiettivo di HAL è di permettere a un integratore di sistema di collegare un gruppo di componenti software per rispettare qualsiasi specifica di input/output sia necessaria.

HAL è basato sullo stesso approccio utilizzato per progettare circuiti elettronici: in elettronica componenti standard, come circuiti integrati, sono posizionati su una scheda con circuiti che collegano i contatti in qualsiasi funzione sia richiesta. Il singolo componente può essere semplice come un amplificatore-operazionale o complesso come un processore di segnali digitali. Ogni componente può venire testato singolarmente, per assicurarsi che funziona per come era stato progettato. Dopo che tutti i componenti vengono posizionati in un circuito collettivo, ogni segnale può essere monitorato per controllare e, nel caso, trovare i problemi.

Come nei componenti elettronici, ogni componente HAL ha dei contatti che possono essere connessi da segnali di un certo tipo. Generalmente un segnale viene aggiornato a intervalli regolari, questa è una differenza con l' elettronica reale, dove una modifica di un contatto si ripercuote immediatamente su tutto il circuito a cui è collegato.

Ogni modulo può essere caricato in real-time o in user space, e a seconda di ciò ha a disposizione dedicate funzioni, quindi quando viene progettato bisogna decidere quale sarà il suo ambiente di caricamento.

Questo è un comune file di configurazione di Machinekit:

```
# kinematics
loadrt trivkins

# trajectory planner
loadrt tp
```

```
# motion controller
loadrt motmod ... tp=tp kins=trivkins

# load low-level drivers
loadrt hal_bb_gpio output_pins=... input_pins=...
loadrt hal_pru_generic ... halname=hpg
loadrt pid count=2
loadrt limit1 count=2

loadusr -Wn Therm hal_temp_bbb ...

addf hpg.capture-position          servo-thread
addf bb_gpio.read                  servo-thread
addf motion-command-handler        servo-thread
addf motion-controller             servo-thread
addf pid.0.do-pid-calcs            servo-thread
addf pid.1.do-pid-calcs            servo-thread
addf limit1.0                      servo-thread
addf limit1.1                      servo-thread
addf hpg.update                    servo-thread
addf bb_gpio.write                 servo-thread
```

Alcune configurazioni specifiche sono state abbreviate perché relative alla specifica configurazione e non interessanti dal punto di vista globale. Nella prima parte carica i componenti per le cinematiche (sistema di riferimento, kinematics in inglese), il trajectory-planner, il motion controller e i driver necessari (hal bb gpio e hal pru generic). I driver devono essere necessariamente essere caricati in real-time.

Il comando **loadrt** viene utilizzato per caricare il componente nell' ambiente real-time, mentre **loadusr** carica il componente in user space.

Ogni componente mette a disposizione delle funzioni che vengono chia-

mate periodicamente. Nella seconda sezione queste funzioni vengono caricate sul servo-thread che le eseguirà, utilizzando il comando **addf**.

Tutte queste configurazioni possono venire dichiarate sia prima della partenza di Machinekit, sia dopo. Questa è una delle principali differenze in HAL tra Machinekit e LinuxCNC[6].

4.2 La tool-chain NC in Machinekit

Verrà ora descritto come Machinekit implementa la tool-chain NC.

4.2.1 Interprete

L' interprete è scritto in python, è un parser che analizza i comandi e produce errori oppure messaggi da passare al modulo successivo: task.

La task riceve tutti i comandi e li elabora. I comandi G-Code non sono tutti comandi di movimento, alcuni possono richiedere elaborazioni, anche complesse. Dopo il componente task c' è quello che si occupa del movimento, quindi ogni elaborazione deve avvenire qui.

Attualmente in Machinekit esiste un solo tipo di task, dedicata alla fresatura, dato il passato di Machinekit. È comunque possibile definire nuove task, per esempio una per la stampa 3d.

4.2.2 Il componente Motion

Il componente motion controlla tutto la tool-chain NC, la figura 4.1 mostra tutte le parti descritte nel capitolo precedente

Il componente task si interfaccia con la "motion interface" (motinf), da qui manda i comandi di movimento e rivece update sullo stato. Queste due comunicazioni sono basate su due strutture C, rispettivamente "motcmd" e "mot status".

Il modulo "command" si occupa solo di gestire il passaggio di messaggi, non fa nessuna delle azioni importanti descritte nella tool-chain NC. Com-

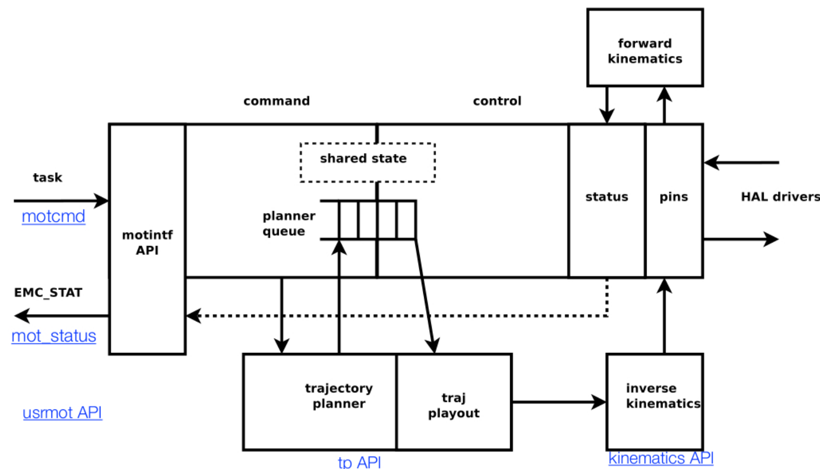

Figura 4.1: Struttura di Machinekit

mand manda i comandi al trajectory-planner, che lavora su un thread diverso (oggi un kernel real-time, anche se non ce ne sarebbe necessità). Il trajetory-planner risponde e i segmenti, arrotondati, e con accelerazioni accettabili, arrivano al modulo "control", il quale interpola approssimando le curve, e si occupa di rispettare il percorso prestabilito.

4.3 Generazione dei passi

Tutta questa parte della tool-chain NC calcola la velocità dei motori in ogni momento. Ma un' altra importante parte da considerare è come si interfaccia con i motori.

Guidare i motori coinvolge molti aspetti elettrici, che sono solitamente gestiti da un device dedicato, chiamato "driver". Questo device solitamente offre un' interfaccia a due contatti per comunicare la direzione e il movimento. Il movimento è comunicato tramite "passi", ogni fronte di salita del segnale comanda un passo nella direzione indicata in quel momento dal segnale dedicato (Ci sono 200 passi in un giro). Il processo di generazione dei fronti di salita è chiamato generazione dei passi.

Controllare questi segnali richiede necessariamente un controllore con risposte veloci. Un sistema operativo non riesce a fare context-switch abbastanza spesso per gestire i segnali di passo e direzione. In controllori numerici eseguiti su micro-controllori questo non è un problema. Ma Machinekit è eseguito su un processore, o più spesso su un micro-processore, quindi si deve trovare un modo per comunicare con un micro-controllore o con un dispositivo alternativo altrettanto valido.

L' integrazione del generatore di passi può essere implementata in due modi:

- usando particolari device che incorporano comnponenti capagi di generare passi

- comunicando con componenti esterni, via Ethernet, SPI, o altro

4.3.1 Device dedicati

Machinekit è stato portato per piattaforme specifiche capaci sia di eseguire il sistema operativo, sia di generare i passi. Queste piattaforme incorporano un micro-controllore oppure un FPGA nello stesso chip del processore Queste piattaforme sono la Beaglebone Black, e il De-0, hanno entrambe processori Cortex-A. Il porting è stato principalmente la scrittura di un driver HAL specifico per la piattaforma.

La Beaglebone ha due micro-controllori, chiamati Programmable Realtime Units, che eseguono il loro firmware, fuori dai context-switch del sistema operativo, e sono quindi adatti alla generazione dei passi.

Il De-0 ha una struttura simile, ma con un FPGA invece dei micro-controllori,anche gli FPGA sono adatti alla generazione dei passi.

4.3.2 Device Connessi

Driver SPI in Machinekit

Machinekit offre un driver per collegare via SPI un Raspberry Pi a un micro-controllore e delegare così la generazione dei passi.

Come ogni driver deve ricevere dal componente motion la velocità comandata e rispondere con la posizione attuale. Dall' altra parte questa stessa comunicazione deve avvenire con il micro-controllore. Il driver si occupa anche di trasformare la velocità comandata in frequenza dei passi. Comunica questa al micro-controllore, che risponde con la posizione attuale. Questa doppia comunicazione è possibile nello stesso momento grazie all' SPI che è un protocollo full duplex.

4.3.3 Il Progetto Klipper

Klipper è un progetto che si pone questo stesso problema, ha una struttura diversa da Machinekit, ma delega in maniera simile la generazione dei passi. Klipper parte da un' installazione di Octoprint su un Raspberry Pi, e implementa tutta la logica del path-planner sulk sistema operativo, invece che sul micro-controllore.

Il Raspberry Pi viene usato per memorizzare il G-Code e calcolare una schedule di eventi che vengono inviati al micro-controllore. Una volta che un evento viene schedulato il Raspberry non ne ha più il controllo, un' eventuale ri-valutazione dello schedule può essere fatta solo dal micro-controllore.

Il Raspberry ha un feed-back dello stato del micro-controllore, ma viene utilizzato solo per l' interfaccia uomo-macchina.

4.4 SPI Slave Implementato in FPGA

È stato deciso di implementare un generatore di passi utilizzando il driver SPI presente in Machinekit, e di collegarlo a un FPGA, per cui si deve implementare il generatore di passi.

È stato scelto il protocollo già presente in Machinekit perchè quello di Klipper non permette di tenere la logica del controllore all' interno di Machinekit.

Capitolo 5

FPGA Development

FPGA è l' acronimo di Field Programmable Gate array, è un particolare tipo di circuito integrato progettato per essere riconfigurato più volte. Esitono linguaggi dedicati alla programmazione degli FPGA, sono chiamati Lignuaggi di Descrizione Harware (HDL). In questi linguaggi è possibile descrivere il comportamento che deve avere un modulo oppure anche una rete di porte logiche connesse tra loro, a seconda del livello di astrazione.

Gli strumenti necessari per lavorare con gli FPGA sono spesso proprietari e forniti dalle aziende produttrici. Ma esiste un' alternativa Open Source.

Oggi esistono due grandi linguaggi di di descrizione hardware, Verilog e VHDL. Entrambi mostrano vantaggi e svantaggi. Si è scelto di utilizzare MyHDL, una libreria python per questo progetto. Il codice MyHDL permette di utilizzare tutte le librerie python, e per essere sintetizzato deve prima essere tradotto in Verilog, o VHDL. È stato scelto Verilog visto che gli strumenti aperti disponibili oggi permettono solo quella strada.

Il driver SPI di Machinekit è fatto per funzionare su un Raspberry Pi versione 1, quindi è stato necessario capire le differenze con l' attuale Raspberry Pi versione 3B. Sono state spostati gli indirizzi delle periferiche nella memoria, quindi è stato aggiornato il driver in modo che riconosca su quale Raspberry Pi si trova e che assegni correttamente gli indirizzi delle periferiche.

5.1 The code structure

Si avrà un modulo SPI Slave che leggerà la frequenza comandata da Machinekit, una FIFO in cui verranno memorizzati i comandi, e infine un modulo che genererà degli impulsi alla frequenza comandata, leggendola dalla FIFO.

5.1.1 SPI Slave

SPI protocol

L' Interfaccia Seriale Periferica è una comunicazione seriale sincrona. Dispositivi SPI possono communicare in full-duplex, utilizzando un architettura di tipo master-slave.

L' SPI è un bus seriale a 4 segnali, che sono:

- Clock: segnale di sincronizzazione, viene mandato dal master a tutti gli slave
- Master Output Slave input (MOSI): è il canale utilizzato dal master per mandare i dati agli slave
- Master Input Slave Output (MISO): è il canale untilizzato dagli slave per rispondere al master
- Slave Select (SS): è il segnale che il master usa per indicare a quale slave sta trasmettendo, deve esserci uno slave select per ogni slave.

Le linee dati (MOSI, MISO) devono essere sincronizzate con il clock. Esistono diverse modalità di invio per indicare se il valore deve essere letto sul fronte di salita o sul fronte di discesa, e sia master che slave devono essere pre-impostati sulla stessa modalità per comunicare. L' implementazione qui descritta utilizza la modalità zero.

SPI Slave code

Questo lavoro utilizza Rhea, una libreria di codice MyHDL di Christopher Felton, disponibile in licenza MIT[7].

Il modulo Slave SPI si interfaccia con master utilizzando 3 contatti fisici (Clock, MOSI e MISO) e con il resto dell' implementazione tramite un BUS FIFO. Il BUS FIFO è sia in lettura che in scrittura, queste due diverse operazioni sono implementate con due diverse code di memoria. Come mostrato in figura 4.1

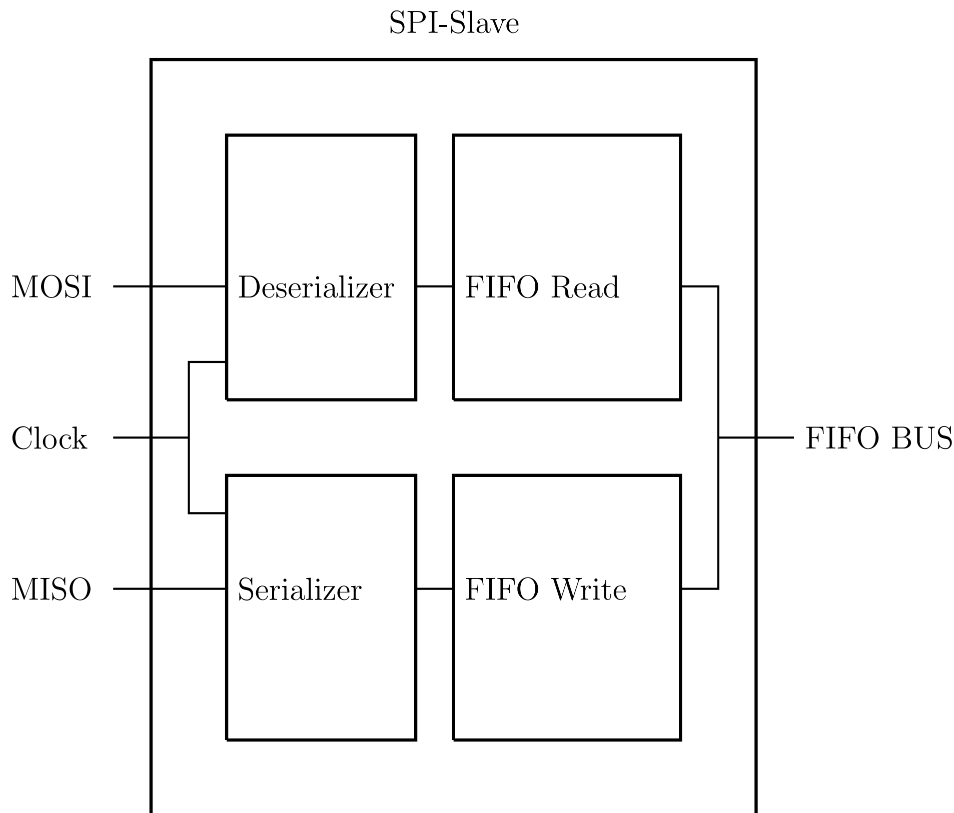

Figura 5.1: Structura di un SPI-Slave

Il BUS FIFO utilizzato viene descritto dal seguente codice MyHDL:

```
self.write_clock = Clock(0)
self.read_clock = Clock(0)
self.clear = Signal(bool(0))
```

```
self.write = Signal(bool(0))
self.write_data = Signal(intbv(0)[width:])
self.read = Signal(bool(0))
self.read_data = Signal(intbv(0)[width:])
self.empty = Signal(bool(1))
self.full = Signal(bool(0))
```

E può essere descritto così:

- read e write hanno due linee di clock separate
- ogni FIFO ha due linee dedicate, un segnale per il comando, e un BUS per la trasmissione dei dati
- empty e full sono due segnali utilizzati per indicare, rispettivamente se è possibile leggere o scrivere sulla specifica FIFO

5.1.2 FIFO Queue

La coda FIFO è implemetata utilizzando una memoria, la cui implementazione può essere trovata in fifomem.py

```
    memarray = Signals(intbv(0)[datasize:0], memsize)

    @always(clock_w.posedge)
    def beh_write_capture():
        wr.next = write
        addr_w.next = write_addr
        din.next = write_data

    @always(clock_w.posedge)
    def beh_mem_write():
        if wr:
            memarray[addr_w].next = din
```

```
    @always(clock_r.posedge)
    def beh_write_address_delayed():
        addr_wd.next = write_addr
        write_addr_delayed.next = addr_wd
```

In questo estratto di codice si può vedere tutta la logica che governa la memoria. La memoria viene dichiarata come un array di Signals, ovvero registri.

Il decoratore python **@always** indica che la funzione seguente deve essere utilizzata nella progettazione hardware. La funzione "beh write capture" è eseguita ogni volta che il segnale "clock w" produce un fronte di salita. E all' interno di quella funzione si può trovare la logica che gestisce la scrittura di un valore all' interno di un registro.

5.1.3 Pulse generator

```
@block
def pulsegen(
 # Ports
 clock,
 frequency,
 duration,
 out_pulse,
 # Parameters
 cnt_max = 10000000
):
 pulse_mem = Signal(intbv(0)[1:0])
 clk_cnt = Signal(intbv(0, min=0, max=cnt_max))
 @always(clock.posedge)
 def beh_strobe():
  if clk_cnt >= frequence:
   pulse_mem.next = 0
```

```
    clk_cnt.next = 0
   else:
    if clk_cnt >= duration:
     pulse_mem.next = 0
     clk_cnt.next = clk_cnt + 1
    else:
     pulse_mem.next = 1
     clk_cnt.next = clk_cnt + 1

 @always_comb
 def beh_map_output():
  out_pulse.next = pulse_mem

return beh_strobe, beh_map_output
```

Questo modulo può generare un impulso che dura "duration" cicli di clock, ogni "frequency" cicli di clock.

5.1.4 Compiling

Questo Makefile mostra come applicare tutte le operazioni necessarie, dalla traduzione del codice MyHDL in verilog, all' utilizzo di Icestorm per produrre un bitstream per programmare l' FPGA.

```
upload: tb.bin
        sudo iceprog tb.bin

tb.bin: tb.txt
        icepack tb.txt tb.bin

tb.txt: tb.blif
        arachne-pnr -d 8k -p example-8k.pcf \
        -o tb.txt tb.blif
```

```
pulse: spi_slave_pulsegen.v
        yosys -p "read_verilog spi_slave_pulsegen.v; \
        synth_ice40 -blif tb.blif"

led: spi_slave_led.v
        yosys -p "read_verilog spi_slave_led.v; \
        synth_ice40 -blif tb.blif"

spi_slave_led.v: spi_slave_led.py
        python spi_slave_led.py

spi_slave_pulsegen.v: spi_slave_pulsegen.py
        python spi_slave_pulsegen.py

clean:
        rm tb.* *.v *.vcd *.pyc

test-led: spi_slave_led.py
        python spi_slave_led.py --test

test-pulse: spi_slave_pulsegen.py
python spi_slave_pulsegen.py --test
```

Per compilare da verilog in un formato binario accettato dall' FPGA utilizzato (Lattice Ice40) bisogna prima sintetizzarlo utilizzando Yosys, ovvero ridurre tutto il progetto a una rete di porte logiche connesse tra loro. Successivamente queste porte logiche e le loro connessioni devono essere posizionate in modo quasi ottimale, e si utilizza Arachne Place n Route per questo. La risultante rete di porte logiche posizionate per lo specifico FPGA, devono infine venir tradotte in uno stream di comandi che programmerà l' FPGA.

Siccome il codice è stato scritto in MyHDL, deve prima essere tradotto

in Verilog.

```
module.config_sim(trace=True)
module.run_sim(10000)
module.convert('Verilog', initial_values=True)
```

Considerando “module” l’instanza di un blocco, prima viene configurata la simulazione in modo che produca un output per gtkwave, e successivamente viene generato il codice verilog.

5.2 Verifica

Il progetto è stato simulato con successo, come mostrato in figura 5,2. Nei segnali è possibile vederne alcuni esterni: il clock (sck), MISO, MOSI, CHIP SELECT, e l’ output del generatore di passi; alcuni interni, relativi alla FIFO interna: reading, fifobus empty, e fifobus read data; e clock count relativo alla generazione degli step. È possibile notare che inizialmente l’ SPI slave riceve la frequenza comandata sul canale MOSI. Il byte inviato è 0b00001010 ovvero, in decimale, 10. Il Chip select (CS, nella figura) rimane abbassato per tutta la durata della comunicazione, per rialzarsi al termine. Appena terminata la ricezione il modulo SPI Slave scrive il valore ricevuto sulla FIFO, e il segnale “fifobus empty” va a zero per indicare che non è più vuota.

Il generatore di impulsi interpreta la coda non vuota come la presenza di comandi da leggere, e alza il segnale di comando “reading”. La FIFO provvede a consegnare il dato, e ritorna vuota. Intanto il generatore di impulsi inizia a contare fino a 10. La lunghezza dell’ impulso è stata impostata a 2 (impostata in modo statico, ma è impostabile anche come dinamico). È infine possibile vedere il segnale out in uscita produrre gli impulsi desiderati.

Tutto il codice è disponibile su github `https://github.com/mngr0/rhea`

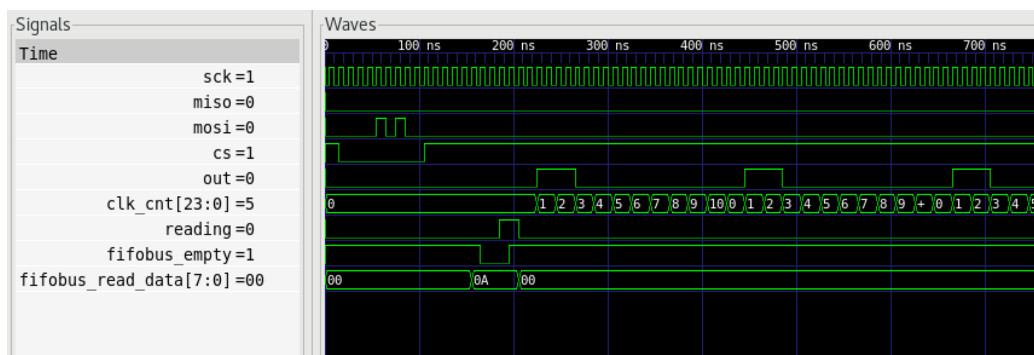

Figura 5.2: Gtkview dei segnali interni

Capitolo 6

Conclusioni

I controllori numerici disponibili oggi, come sorgente aperto si possono dividere in due categorie: quelli eseguiti su micro-controllori, e quelli eseguiti su interi sistemi operativi.

I compiti di un controllore numerico vanno dal path-planning alla generazione di passi, e interfacce complesse come video o Ethernet sono spesso richieste.

Il path-planning richiede calcoli floating-point che non hanno motivo di venire eseguiti su micro-controllori o in thread real-time. L' unica conseguenza è il rallentamento della generazione di passi, che invece richiede un hardware dedicato senza context-switch.

Quando si utilizza un micro-controllore per una stampante 3D, spesso come interfacce vengono utilizzati Octoprint e Repetier-Server. Questi programmi vengono eseguiti su un single board computer, come il Raspberry Pi, e agiscono come interfaccia per la stampante. Il Raspberry Pi deve essere connesso al micro-controllore e diventa critico per il processo di stampa. Dato il suo basso costo è un compromesso accettabile.

Ma ora il path-planner è in una posizione contraddittoria: viene eseguito sul micro-controllore, quando c'è un intero sistema operativo inattivo per la maggior parte del tempo. Mentre sarebbe più adeguato al tipo di operazioni necessarie, perché:

- può gestire operazioni floating-point con più precisione
- il path-planner può venire eseguito sul suo thread, liberando il micro-controllore da context-switch non necessario

On progetto basato esattamente su questo ragionamento già esiste: Klipper. Il progetto Klipper sposta esattamente il path-planner sul Raspberry Pi, e lo reimplementa in python. Calcola uno schedule di eventi e lo fa eseguire al micro-controllore.

Comunque, Klipper è dedicato alla stampa 3d, o macchine CNC al massimo. Il Raspberry Pi esegue la tool-chain NC e nulla di più, tutta la logica di controllo risiede ancora sul micro-controllore. Se si volesse, per esempio, aggiungere un encoder, bisognerebbe implementarne la gestione nel firmware del micro-controllore

Machinekit, d'altro canto, utilizza il micro-controllore solo per le specifiche azioni di input/output, e tutta la logica è tenuta all' interno del sistema operativo, e processa gli eventi con rapidità utilizzando thread real-time.

Sarebbe interessante comparare i tempi di risposta agli eventi (per esempio un encoder) tra:

- Machinekit eseguito su un kernel real-time
- un controllore numerico eseguito su un micro-controllore

Bibliografia

[1] A Brief History of Automatic Control, Stuart Bennet 1996, URL:http://ieeecss.org/CSM/library/1996/june1996/02-HistoryofAutoCtrl.pdf

[2] Use of Open Source Distribution for a Machine Tool Controller, William P. Shackleford and Frederick M. Protoctor URL:https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=821651

[3] Machinekit Manual, Gcode Overview, URL: http://www.machinekit.io/docs/gcode/overview

[4] Yosys Manual URL:http://www.clifford.at/yosys/files/yosys_manual.pdf

[5] FPGA implementation of higher degree polynomial acceleration profiles for peak jerk reduction in servomotors, Roque Alfredo Osornio-Rios a, René de Jesús Romero-Troncoso, Gilberto Herrera-Ruiz, Rodrigo Castaneda-Miranda, Robotics and Computer-Integrated Manufacturing 25 (2009) 379-392

[6] Machinekit manual, HAL basics, Loading a component, URL:http://www.machinekit.io/docs/hal/basic_hal/#loading-a-component

[7] A collection of MyHDL cores and tools for complex digital circuit design, Christopher Felton URL:https://github.com/cfelton/rhea

Ringraziamenti

Ringrazio il Professor Renzo Davoli per questo progetto basato su tecnologia FPGA, ringrazio Baldazzi Mauro Impianti per aver ospitato e finanziato questa ricerca, ringrazio la community che mantiene Machinekit per il lavoro di supporto in un grande progetto in cui vedo grandi potenzialità, ringrazio la mia famiglia e i miei amici per il supporto morale di ogni giorno.