SAPIENZA
UNIVERSITÀ DI ROMA

Linguaggio ed interprete per la simulazione di dispositivi di I/O

Facoltà di Ingegneria dell' Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Stefano Palmieri
Matricola 1744854

Relatore

Prof. Alessandro Pellegrini

Anno Accademico 2019/2020

Linguaggio ed interprete per la simulazione di dispositivi di I/O
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: palmostefano@gmail.com

Sommario

L'obiettivo di questo progetto è quello di realizzare un interprete di un linguaggio che descriva il funzionamento di una macchina a stati che implementi il microcodice di periferiche generiche per architetture simil-x86. Il lettore verrà introdotto nel primo capitolo al problema di partenza e gli verrà presentata l'architettura di riferimento, successivamente verrà spiegato in generale il funzionamento dell'interprete e il suo ruolo. Infine verrà presentato l'interprete da me sviluppato, si mostrerà il codice implementato e il tipo di linguaggio che il suddetto compilatore può interpretare.

Indice

Capitolo 1

Introduzione al problema

L'intento è di realizzare un linguaggio che permetta di descrivere il funzionamento del microcodice di periferiche simil-x86 compatibili, compresa la comunicazione tra CPU e dispositivi di input/output (I/O). L'obiettivo è quindi quello di rappresentare gli stati in cui entrano le periferiche al momento della ricezione di eventi esterni e descrivere l'evoluzione al passaggio tra i vari stati. Dunque il progetto realizzato deve essere in grado di interpretare e simulare la SCO delle periferiche (nel paragrafo successivo introdurremo il significato di SCO). Il progetto realizzato consente allo studente di comprendere la logica dell'interazione e comunicazione tra i dispositivi di I/O e la CPU del sistema principale. Il motivo per cui ho deciso di sviluppare un parser e un interprete, invece di utilizzarne uno già esistente, è quello di poter avere a disposizione un linguaggio essenziale che risponda esclusivamente alle nostre necessità, ovvero le funzionalità della SCO, e che sia il più semplice possibile dal punto di vista lessicale in modo che lo studente possa concentrarsi principalmente sulla logica del funzionamento della SCO delle periferiche.

1.1 Cos'è una SCO

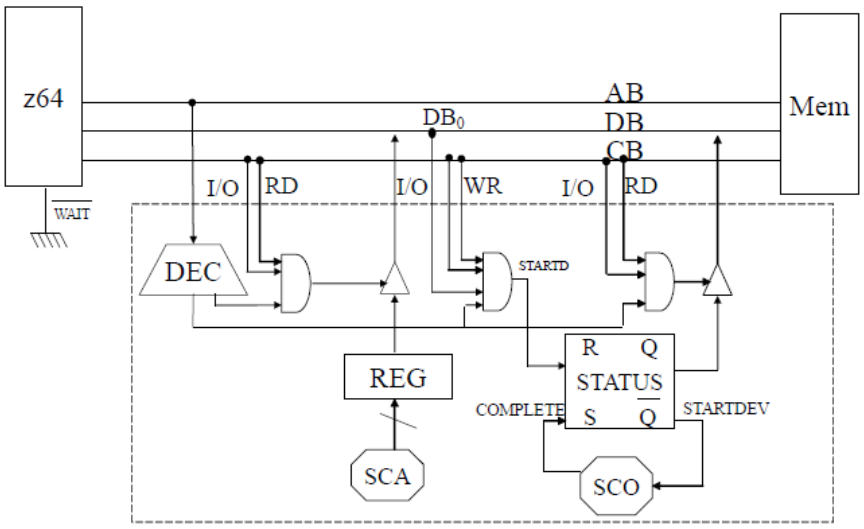
In un dispositivo di input/output solitamente andiamo a distinguere due sottosistemi: il primo lo chiameremo SCA (sottosistema di calcolo), un'unità che si occupa solo ed esclusivamente di svolgere operazioni di calcolo (ad esempio operazioni aritmetiche); il secondo lo chiameremo SCO (sottosistema di controllo) che gestisce le varie attività della periferica, tra cui coordinare la comunicazione con la CPU, rispondere alla ricezione di eventi esterni e dirigere le varie operazioni della SCA. Il lavoro di una SCO può essere descritto mediante una macchina a stati finiti, dove il passaggio da uno stato ad un altro può avvenire a seguito di un evento esterno (es. quando un utente preme un bottone della periferica), a causa di un segnale da parte della CPU o semplicemente dallo scadere di un tempo limite prefissato (es. colpo di clock).

1.2 Architettura a cui si fa riferimento

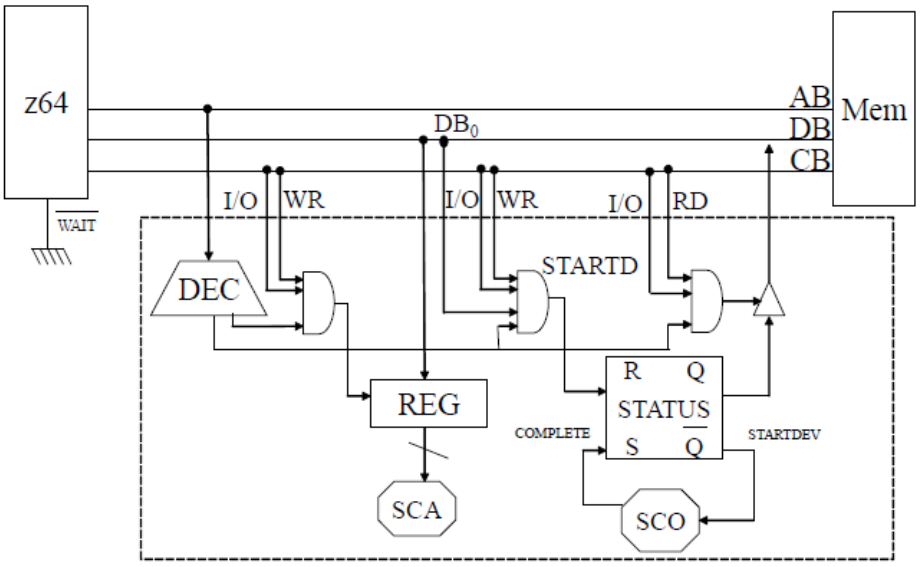
Per la realizzazione dell'interprete mi sono rifatto ad un'architettura RISC (Reduced Instruction Set Computer) a singolo BUS; le periferiche di riferimento sono dispositivi di I/O semplici con un singolo SCA e un singolo SCO in grado di ricevere eventi

esterni e di gestire operazioni elementari. Dal punto di vista logico tali periferiche utilizzano un flip/flop di stato mediante il quale il dispositivo di I/O instaura un protocollo di comunicazione con la CPU e viceversa; questo flip/flop può essere usato sia dalla CPU per comunicare alla periferica di avviare una determinata operazione, sia dalla periferica per comunicare alla CPU che essa è pronta o ha terminato una specifica operazione (il busy waiting viene coordinato attraverso quest'ultimo flip/flop). Un altro componente logico sempre presente nelle periferiche di riferimento è il flip/flop di interrupt_request che serve a richiamare l'attenzione della CPU. Questi due flip/flop sono componenti essenziali delle periferiche; possono essere presenti ulteriori registri e/o flip/flop per lo svolgimento delle operazioni proprie di una determinata periferica; ma quest'ultimi dipendono dal tipo di periferica scelta e da come viene imlementata al livello logico. Nella scrittura dell'intrerprete ho speso molta attenzione alla futura implementazione di protocolli per la comunicazione tra i due dispositivi. Prima di introdurre i due tipi di protocolli possibili è bene andare a spiegare il significato del termine protocollo; in informatica si definisce protocollo l'insieme coordinato di regole che consente a due interlocutori (un utente e un calcolatore elettronico, due utenti oppure due calcolatori) di scambiarsi rapidamente e univocamente dati e messaggi, cioè di colloquiare fra loro. Presenteremo ora due tipi di protocolli che useremo in questa trattazione per la comunicazione tra periferiche e CPU, cioè il protocollo sincrono e asincrono. Abbiamo un protocollo sincrono quando la comunicazione viene scandita da un medesimo clock (cioè quando i clock dei due dispositivi comunicanti sono coordinati tra di loro); altrimenti, nel caso in cui i clock siano differenti, il protocollo si dice asincrono e in generale tale protocollo non ha la caratteristica di avere un limite di tempo massimo per il completamento di una iterazione. Nel primo caso il dispositivo con cui la CPU necessita di interagire deve leggere e/o scrivere il dato nell'intervallo di tempo previsto dal SCO del processore per la lettura e/o la scrittura e quindi sincrono con la velocità di funzionamento del processore, come nel caso di interazione processore-memoria di lavoro (RAM statica). Invece, se la velocità di lettura/scrittura del dispositivo è inferiore a quella richiesta è necessario "rallentare" le attività del SCO di quest'ultimo, tale modalità si denota come “busy_waiting”, e in tal caso il trasferimento è asincrono. Di seguito mostrerò due esempi schematici di periferiche di input e di output:

Interfaccia di Input



Interfaccia di Output



Queste due immagini rappresentano rispettivamente una periferica di input e una di output; notiamo che entrambe sono dotate di un decoder (DEC) che permette l'identificazione di uno specifico componente circuitale all'interno della periferica con il quale la CPU vuole interagire, un registro contenente le informazioni elaborate o quelle che devono essere elaborate dalla periferica, un flip/flop di status, una SCA e una SCO collegate rispettivamente con il registro e il flip/flop. Guardando

attentamente possiamo notare che l'unica differenza tra le due è nel collegamento BUS-registro; infatti nel primo caso (input) abbiamo che il collegamento è uscente dalla periferica, pertanto sarà la periferica a passare informazioni alla CPU; nel secondo, invece, il collegamento è entrante e in questo caso sarà la CPU a passare informazioni al dispositivo di output.

Capitolo 2

Introduzione all'interprete

Prima di poter entrare nel fulcro dell'argomento è doveroso riportare alcuni concetti di base necessari alla comprensione del lavoro e quindi bisogna introdurre il concetto di compilatore.

2.1 Cos'è un compilatore e a cosa serve

Il compilatore è quel programma che legge le istruzioni in formato sorgente, in qualche linguaggio di programmazione e lo traduce in linguaggio oggetto che può essere o un altro linguaggio di programmazione oppure un linguaggio macchina. Il modello di compilazione più diffuso è quello basato su due fasi: analisi e sintesi. La parte di analisi serve a dividere il programma sorgente nelle sue parti costituenti e a creare una rappresentazione intermedia (formata da codici e strutture dati) utilizzata dalla fase di sintesi per creare il programma oggetto. La fase di analisi è divisa in 3 parti: lessicale, semantica e sintattica. Dopo le fasi di analisi il compilatore genera una rappresentazione intermedia del programma sorgente. Nella fase di sintesi si ricorre all'ottimizzazione del codice che permette di generare un codice a basso livello più efficiente, ricordo che fino a questo punto abbiamo un codice intermedio astratto. Durante lo sviluppo del mio compilatore mi sono fermato alla fase di analisi, il motivo di questa scelta è dovuto al fatto che ho realizzato è un interprete come obiettivo finale.
È importante allora capire bene che cos'è un interprete e cosa lo differenzia da un compilatore: mentre un compilatore produce un codice oggetto eseguibile da un'architettura reale, un interprete non produce un eseguibile, ma esegue direttamente (partendo dalla rappresentazione intermedia) le operazioni descritte dal programma sorgente.

2.2 Come implementare un compilatore

Scenderemo ora più nel dettaglio, nel tentativo di fare chiarezza sulle fasi introdotte precedentemente. L'analisi lessicale è affidata allo scanner che rappresenta un'interfaccia intermedia tra il programma sorgente e l'analizzatore sintattico o parser. Lo scanner, attraverso un esame carattere per carattere dell'input, separa il programma sorgente in frammenti chiamati token che rappresentano le unità indivisibili del

programma, come ad esempio: i nomi delle variabili, operatori, label, ecc. Nell' analisi sintattica, i token, identificati nella fase precedente, vengono passati ad un parser il quale li raggruppa gerarchicamente tramite delle strutture ad albero (dove le foglie sono i token) in modo da dare una struttura sintattica alle frasi del programma. Nella fase semantica, invece, il parser controlla che non ci siano errori semantici e acquisisce informazioni sui tipi che verranno usate nella fase di generazione del codice. La rappresentazione intermedia (generata dalla fase di analisi sintattica) viene usata per identificare operatori ed operandi di espressioni e statements. Due esempi di analizzatori lessicali e sintattici sono lex e flex per l'analisi lessicale e yacc e bison per l'analisi sintattica. L'analisi eseguita dal parser viene passata ad un interprete, che nel mio caso ho scritto in C, il quale analizza le informazioni ricevute dal parser ed esegue un programma con le informazioni ricevute.

Capitolo 3

Realizzazione dell'interprete

3.1 Tipo di Interprete sviluppato e logica di partenza

L'interprete che ho realizzato è in grando di leggere un linguaggio a stati rappresentativo del microprocessore di una SCO. Per implementare la struttura e la logica del mio interprete sono partito dall'analisi di alcuni circuiti delle interfacce di periferiche di I/O (vedi immagini del capitolo 1); sulla base di questi circuiti ho realizzato dei microcodici atti a implementare le attività delle SCO di tali ciruiti (un esempio di questi microcodici è disponibile nella sezione codice a fine libro). Sulla base di questi codici ho iniziato a scrivere il mio interprete.

3.2 Struttura del codice

Il codice implementato è formato da uno scanner scritto in flex, tale programma prende in input il microcodice della SCO riconosce i caratteri "speciali" (comandi e/o operazioni) e passa la sua analisi (formata da token) al parser. Il parser è stato scritto in bison e il ruolo che gioca è quello di costruire una struttura logica attraverso i dati raccolti dallo scanner. Il parser successivamente passerà la rappresentazione intermedia generata all'interprete, il quale analizzerà la struttura fornitagli dal parser e inizierà ad eseguire (simulare) il nostro microcodice. Per l'implementazione dell'intero programma mi sono appoggiato a due file inportanti: dichiarazioni.h e funzioni.c; il primo definisce tutte le strutture dati d'appoggio utilizzate dal parser per generare l'albero sintattico e il secondo definisce le varie funzioni necessarie per l'utilizzo di tali strutture.

3.3 Tipo di linguaggio interpretabile

Il linguaggio che ho ideato è costituito da 4 blocchi principali: i primi 3 sono dichiarativi e servono esclusivamente a inizializzare le variabili utilizzate, a definire i tipi di eventi che possono essere gestiti e i comandi che possono essere utilizzati. Il quarto blocco è il vero fulcro del programma, in questo vengono esplicitati tutti gli stati in cui può entrare una SCO e i passaggi di stato possibili; questo blocco è formato da una dichiarazione dello stato dove viene assegnato un nome a quest'ultimo;

state nome_stato{

successivamente segue un comando (actions) dove vengono elencate tutte le operazioni che la SCO della periferica deve eseguire appena entra nello stato;

state nome_stato{
actions{azione1, azione2, azione3};

tali azioni non sono altro che le operazioni che la periferica svolge in risposta alla ricezione di un evento esterno: se per esempio abbiamo come periferica un lettore di banconote allora nel momento in cui inseriamo una banconota nel lettore quest'ultimo dal suo stato iniziale di attesa passerà allo stato banconota_inserita e come operazione scriverà all'interno del suo registro il valore di tale banconota.
Dopo il comando actions segue un elenco di cambi di stato coordinato da alcune strutture di controllo ovvero IF, FOR, WHILE, ELSE. I cambi di stato sono delle semplici transizioni tra le condizioni in cui la periferica si può trovare, tali transizioni vengono scaturite da un evento esterno (pressione di un bottone) o interno (segnale da parte di una CPU) che la periferica può ricevere. I cambi di stato rappresentati dal mio codice, quindi, sono formati da: un evento che può ricereve la periferica seguito da un carattere speciale ('=>') e da uno stato che rappresenta quello in cui la periferică dovrà andare a seguito dell'input di tale evento; nel caso in cui la periferica non riceva alcun evento all'interno di un intervallo di tempo precedentemente stabilito, allora questa riceverà come evento un segnale di "timeout" che nel mio linguaggio viene rappresentato con l'assenza dell'evento davant al carattere speciale.

Es. evento=>nuovo stato:

bottone_premuto=>stato_premuto

Es. timeout=>nuovo stato:

=>timeout

Non avendo a disposizione periferiche sulle quali riprogrammare le SCO ho creato da me un emulatore che simulasse mediante delle interruzioni gli eventi esterni ricevuti; per fare queste interruzioni ho usato due timer: uno per gli eventi e uno per il timeout.

```
1 state nome_stato{
2   actions{azione1, azione2, azione3};
3   IF(var1>=var2){
4       =>nome_stato1;
5       }
6   evento=>nome_stato2;
7   ....
8 }
```

Listing 3.1. esempio

Andiamo ad analizzare brevemente quello che accade al codice di sopra riportato nel momento in cui verrà interpretato: lo stato dopo essere stata analizzato e scomposto dal programma in flex, viene incapsulato, dopo averne ricostruito l'albero logico attraverso il programma in bison, in una struttura dati che permetterà l'accesso alle informazioni dello stato quando la periferica riceverà l'evento ad esso associato. Di seguito viene mostrata la struttura dati "stato" della quale mi sono avvalso; si può notare che è una struttura complessa e quindi composta da più strutture: possiamo constatare come primo elemento il nome identificativo dello stato, successivamente sono presenti un insieme di liste: azioni, eventi, elenco condizioni (el_cond), elenco stati(el_stati), cicli che saranno riempite dal programma in bison quando il codice verrà interpretato.

```
typedef struct stato{
        String nome;
        struct action* azioni;
        struct event* eventi;
        struct elenco_cond* el_cond;
        struct elenco_stati* el_stati;
        struct cicli* cic;
       }stato;
```

Nel momento in cui l'emulatore della periferica riceve in input l'evento associato al nostro stato cerca la struttura contenente le informazioni necessarie a proseguire il programma e, una volta trovata, esegue come da principio le azioni specificate all'interno della sezione *actions*; una volta eseguite queste operazioni, l'emulatore verifica la presenza o meno d'istruzioni di controllo e se presenti ne valuta le condizioni per poi mettersi in attesa dell'ingresso di un evento. Se quest'ultimo arriva prima dello scadere del timeout l'emulatore cerca all'interno delle istruzioni di controllo in cui si trova un cambio di stato coerente con l'evento ricevuto; se così non accade o se il cambio di stato adeguato non è presente all'interno della struttura, l'emulatore eseguirà il cambio di stato associato al timeout (privo di evento). Solitamente una periferica se non riceve alcun tipo di evento esterno e/o interno si pone in uno stato di ready in cui non svolge alcuna operazione ma rimane in attesa della ricezione di un input; nel mio linguaggio tale stato standard viene identificato con il nome di *idle* ed è da questo che l'emulatore inizierà ad eseguire il codice, come una funzione *main* all'interno di un programma in C; questo perchè lo stato di "idle" è anche lo stato in cui si pone inizialmente una periferica dopo essere stata attivato. In assenza di questo stato, quindi,l'interprete non è in grado di eseguire il codice e pertanto invierà un messaggio di errore.

Capitolo 4

Conclusioni

L'obiettivo prestabilito per questa tesi è stato sicuramente raggiunto, l'interprete infatti riesce ad interpretare il codice in input (un microcodice delle SCO) e riesce ad eseguire le istruzioni corrette relative al microcodice; tuttavia il programma potrebbe essere ulteriormente esteso per apportare diverse migliorie. L'interprete implementato, infatti, ha diversi limiti a livello operazionale; quest'ultimo, infatti, supporta solo ed esclusivamente le 4 operazioni, mentre a livello di struttura di controllo l'interprete non è in grado di gestire condizioni o cicli annidati. Per quanto riguarda i limiti I limiti esposti potrebbero essere spunti di partenza per migliorare il codice per renderlo più robusto e arricchirlo di operazioni supportate. Come anticipato nel primo capitolo, non ho provveduto a fare nessuna ottimizzazione né dal punto di vista spaziale né temporale, diverse quindi sono le modifiche che potrebbero essere effettuate alla struttura per ridurre questi due importanti aspetti del programma.

Codice

Di seguito allego parte del codice; l'intero programma, invece, è disponibile al seguente link: `https://github.com/steno97/tesi`
Se si vuole fare un test, si può eseguire il file di prova digitando le seguenti istruzioni nella shell:
> make
> ./compilatore
In questo modo verrà eseguito il seguente programma:

```
1
2  global_variables{
3    int prima=1000;
4    int seconda=0;
5  }
6
7  events{
8    diminuiamo_prima;
9    aumentiamo_seconda;
10     errore;
11   ciclo;
12 }
13
14 commands{
15   incrementa;
16   decrementa;
17 }
18
19 state idle{
20   actions{print_schermo_message(stiamo_nell'idle)};
21   diminuiamo_prima=>dim;
22   aumentiamo_seconda=>incr;
23   errore=>error;
24   ciclo=>cic;
25   =>idle;
26 }
27
28 state dim{
29   actions{dec(prima)};
30   if(prima==0){
31     print_schermo_message(non_si_puo_piu_diminuire_prima);
32   }
33   aumentiamo_seconda=>incr;
34   errore=>error;
35   ciclo=>cic;
36   =>idle;
```

```
}


state incr{
  actions{inc(seconda)};
  if(seconda==5){
    print_schermo_message(seconda_e_arrivata_a_5);
  }
  diminuiamo_prima=>dim;
  errore=>error;
  ciclo=>cic;
  =>idle;
}

state cic{
  actions{print_schermo_message(stiamo_nel_ciclo)};
  while(prima>=900){
    dec(prima);
  }
  diminuiamo_prima=>dim;
  errore=>error;
  aumentiamo_seconda=>incr;
  =>idle;
}

state error{
  actions{print_schermo_message};
  =>idle;
}
```

Listing 1. Esempio linguaggio "prova.txt"

```

....

//programma principale e configurazione handler
int main(int argc, char **argv){
  if(!(yyin = fopen("prova.txt", "r"))) {
    printf("errore\n");
    return (1);
    }

  ////////////////////////////////////////////////////////////////
  //inizializzazione lettura file
    fp = fopen("eventi.txt", "r");
    if (fp == NULL){
        printf("errore nell'apertura file");
  }

  ////////////////////////////////////////////////////////////////
  // 1   interrupt
  struct sigaction sa;


  /* Install timer_handler as the signal handler for SIGVTALRM. */
  memset (&sa, 0, sizeof (sa));
```

```
25    sa.sa_handler = &timer_handler;
26    sigaction (SIGALRM, &sa, NULL);
27
28    /* Configure the timer to expire after 500 msec... */
29    timer.it_value.tv_sec = 0;
30    timer.it_value.tv_usec = 70000;
31    /* ... and every 500 msec after that. */
32    timer.it_interval.tv_sec = 0;
33    timer.it_interval.tv_usec = 700000;
34    /* Start a virtual timer. It counts down whenever this process is
35    executing. */
36    setitimer (ITIMER_REAL, &timer, NULL);
37
38
39    ////////////////////////////////////////////////////////////////
40    //2   interrupt
41    struct sigaction sa1;
42
43    /* Install timer_handler as the signal handler for SIGVTALRM. */
44    memset (&sa1, 0, sizeof (sa1));
45    sa1.sa_handler = &evento_handler;
46    sigaction (SIGVTALRM, &sa1, NULL);
47
48    /* Configure the timer to expire after 500 msec... */
49    timer1.it_value.tv_sec = 0;
50    timer1.it_value.tv_usec = 50000;
51    /* ... and every 500 msec after that. */
52    timer1.it_interval.tv_sec = 0;
53    timer1.it_interval.tv_usec = 500000;
54    /* Start a virtual timer. It counts down whenever this process is
55    executing. */
56    setitimer (ITIMER_VIRTUAL, &timer1, NULL);
57
58    ////////////////////////////////////////////////////////////////
59
60    int flag=0;
61    flag = yyparse();
62
63      fclose(yyin);
64      fclose(fp);
65      if (line){
66          free(line);
67    }
68      return flag;
69  }
```

Listing 2. Interprete

Bibliografia

[1] Enciclopedia Treccani: *http://www.treccani.it/vocabolario/protocollo/*.

[2] Prof Bruno Ciciani, *"I-O Z64 con due bus"*. Dispensa

[3] Dispense online: *"http://people.disim.univaq.it/orefice/Dispensa1.pdf"*.