STUDIO DEL CASO

Problema

Secondo quanto dettato dalle linee guida del progetto, il programma deve analizzare un codice binario dato in ingresso e restituire la relativa istruzione MIPS.
Il codice binario in questione, secondo lo standard MIPS per l'architettura a trentadue bit, deve essere lungo trentadue bit.
L'implementazione da noi realizzata segnala con un messaggio di errore qualora il codice sia troppo breve o contenente caratteri diversi da '1' o '0' ed ignora i caratteri che eccedono la lunghezza massima.

Eccezioni

La consegna ci fa assumere che il codice dato in ingresso rappresenti un'istruzione MIPS valida.
Tuttavia il codice da noi redatto riesce ad identificare anche le istruzioni non valide.
Qualora il programma riconosca il verificarsi di quest'ultimo caso, interroga l'utente sulla volontà di arrestare l'esecuzione dello stesso o continuare ed inserire un nuovo codice binario.

Analisi

Avendo il MIPS una rosa di istruzioni piuttosto estesa abbiamo deciso di creare un immaginario linguaggio sliMips che supporti venti istruzioni MIPS base, ne abbiamo analizzato i codici binari e le abbiamo organizzate in sottogruppi in base al loro codice operativo.
In nove istruzioni, avendo codice operativo posto a 000000, dobbiamo analizzare il codice funzione per identificarle.

STRUTTURA

.data

Il *.data* viene popolato inserendo:

1) i messaggi di interazione utente con localizzazione it_IT;
2) i nomi delle istruzioni riconosciute;
3) *registersSet*: un vettore (più propriamente un insieme) contenente i nomi dei registri ordinati dallo zero al trentadue;
4) *instructionsSet*: un vettore che simula un array bidimensionale avente come indice il codice operativo e come contenuto l'indirizzo (inserito dinamicamente mediante la procedura *setInstructionHandler*) dell'etichetta che gestisce tale codice;
5) *instructionsZeroSet*: identico ad *instructionsSet*, memorizza le istruzioni con codice operativo '000000';
6) *hexString*: una stringa utile per la visualizzazione degli indirizzi in formato esadecimale;
7) *binaryCode*: una word riservata al contenimento dell'indirizzo dello stack in cui è memorizzato il codice binario immesso dall'utente.

.text (main*)

Il *main**, punto di partenza del programma, è suddiviso in stage, ognuno con un compito specifico:

1. *main*: è una fase di bootstrap utile a popolare dinamicamente il vettore *instructionsSet*. Una volta eseguita non viene più reinvocata;
2. *main_start*: è dedicata all'interazione con l'utente, al controllo dell'input ed allo smistamento delle competenze per l'analisi della stringa stessa. Questa fase può essere rieseguita qualora l'utente immetta un codice binario errato e scelga di poterne reimmettere un altro;
3. *main_exit*: ha il compito di terminare l'esecuzione del programma. Questa fase viene eseguita unicamente quando il programma giunge a naturale fine o quando, riscontrata una anomalia nel codice binario, sia lo stesso utente a chiederne la terminazione.

.text (print*)

Le procedure il cui nome inizia per *print*:

- *print_string*
- *print_integer*
- *print_space*
- *print_separator*
- *print_immediate*
- *print_immediate_hex*
- *print_register_one*
- *print_register_two*
- *print_register_three*

sono delle facilities per la visualizzazione a schermo. Eccezion fatta per le prime due, nessuna delle su elencate procedure accetta parametri in ingresso e restituisce valori in ritorno.
Per un'analisi più approfondita si rimanda alla sezione PROCEDURE od alla documentazione nel codice.

.text (convert*)

Le procedure il cui nome inizia per *convert*:

- *convertUBinFromStringToDec*
- *convertBinFromStringToDec*
- *convertFromDecToHex*

sono destinate alla conversione tra formati (binario, decimale, esadecimale).
Per un'analisi più approfondita si rimanda alla sezione PROCEDURE od alla documentazione nel codice.

.text (_instruction*)

Le procedure il cui nome inizia per *_instruction* sono i gestori delle istruzioni (instuctions handlers) ed hanno il compito, una volta identificato il codice operativo e se necessario il codice funzione, di analizzare il codice binario e visualizzarne in output il corrispettivo in linguaggio MIPS.

PROCEDURE

void setInstructionHandler(* $a0, * $a1)

Non viene ritornato alcun valore.

$a0 contiene l'indirizzo del vettore bidimensionale che associa al codice operativo l'indirizzo della procedura che lo gestisce;
$a1 contiene l'indirizzo della procedura da inserire nel vettore $a0.

Assegna l'indirizzo contenuto in $a1 alla word successiva a quella che contiene il codice operativo.
Incrementa $a0 di otto e ripassa l'esecuzione al chiamante.

Il vettore puntato da $a0 presenta la struttura che segue:

4 byte	4 byte	4 byte	4 byte	4 byte
Codice Operativo	Indirizzo Procedura	Codice Operativo	Indirizzo Procedura	...

pointer getInstructionHandler(* $a0, integer $a1)

Viene ritornato un indirizzo.

$a0 contiene l'indirizzo del vettore bidimensionale che associa al codice operativo ($a1) l'indirizzo della procedura che lo gestisce;
$a1 contiene il codice operativo in decimale di cui si vuole ottenere il gestore.

La procedura esegue un ciclo sul vettore puntato da $a0 alla ricerca dell'indirizzo associato al codice operativo contenuto in $a1. L'indirizzo viene ritornato al chiamante. Qualora il codice cercato non dovesse essere trovato, verrebbe visualizzato un messaggio di avvertimento e verrebbe terminata l'esecuzione del programma.

Per approfondimenti sulla struttura di $a0 si rimanda all'analisi della procedura *setInstructionHandler.*

Nota Implementativa:
Il vettore puntato da $a0 presenta gli indici, rappresentati dai codici operativi, ordinati in senso crescente. Per questo l'indice n sarà sempre seguito dall'indice n+i e preceduto dall'indice n-i, dove 'n' ed 'i' sono dei numeri naturali. A tal proposito, è stato implementato un algoritmo per la ricerca che termina la sua esecuzione non appena il valore dell'indice corrente al ciclo risulta maggiore dell'indice cercato con $a1. Questo al fine di accelerare la ricerca nel vettore.

void checkString(* $a0)

Non viene ritornato alcun valore.

$a0 contiene l'indirizzo, nella posizione del bit più significativo, della stringa di trentadue byte immessa da console.

Effettua un ciclo di trentadue iterazioni scorrendo carattere per carattere, dal più significativo al meno, tutta la stringa. Per ogni carattere estratto, viene chiamata la procedura *checkString_byte* (con l'indirizzo del carattere come parametro) che verifica che il carattere sia 48 o 49 (in codifica ASCII), gli sottrae 48 (per avere il numero in decimale), lo memorizza nella locazione di memoria puntata da $a0 e restituisce il controllo alla funziona chiamante.
Qualora il carattere non sia valido o la stringa sia troppo breve viene mostrato un messaggio di errore all'utente e gli viene chiesto se voglia inserire una nuova stringa (inserendo '1') o uscire dal programma (inserendo '0').
Non è presente una procedura apposita per il controllo della lunghezza della stringa in quanto questa verifica viene svolto implicitamente dalla procedura *checkString_byte* che, qualora gestisse una stringa più breve di trentadue caratteri, genererebbe un errore incontrando in maniera prematura il carattere di fine stringa.

integer convertUBinFromStringToDec(* $a0, integer $a1)

Viene ritornato un valore intero.

$a0 contiene l'indirizzo, nella posizione del bit più significativo, della stringa di $a1 byte da convertire;
$a1 contiene la lunghezza della stringa da convertire.

Converte una stringa binaria in decimale senza tenere in considerazione il segno e la ritorna al chiamante.

integer convertBinFromStringToDec(* $a0, integer $a1)

Viene ritornato un valore intero.

$a0 contiene l'indirizzo, nella posizione del bit più significativo, della stringa di $a1 byte da convertire;
$a1 contiene la lunghezza della stringa da convertire.

Converte una stringa binaria, codificata in complemento a due, in decimale tenendo in considerazione il segno e la ritorna al chiamante.

pointer convertFromDecToHex(integer $a0)

Viene ritornato un indirizzo.

$a0 contiene il valore decimale da convertire;

Questa procedura in realtà non esiste in questa forma, ma è stata inserita all'interno di *print_immediate_hex*.

integer pow(integer $a0, integer $a1)

Viene ritornato un valore intero.

$a0 contiene la base da elevare a potenza;
$a1 contiene la potenza a cui elevare la base.

Come la omonima funzione C, la procedura pow esegue l'elevazione ad una generica potenza $a1 di una generica base $a0 e ritorna il risultato al chiamante.

pointer getRegisterString($a0)

Viene ritornato un indirizzo.

$a0 contiene l'indice in cui andare a cercare la stringa che descrive in linguaggio MIPS un registro.

Ritorna l'indirizzo della stringa memorizzata in .data, precisamente nel vettore *registersSet*, corrispondente all'indice passato in $a0 (es. a 2 corrisponde $v0). Ogni stringa memorizzata in *registersSet* ha una lunghezza di esattamente 4 byte, tre per la stringa che codifica il registro MIPS e uno per il carattere di terminazione. Per questo, l'accesso al record cercato viene effettuato moltiplicando l'indice passato in $a0 per il fattore quattro. Tale comportamento risulta possibile in quanto i registri MIPS sono identificati da numeri interi consecutivi (da zero a trentadue).

FUNZIONAMENTO

main

In questa fase viene invocata la procedura *setInstructionHandler* (per dodici volte) per riempire il vettore *instructionsSet* (vedi l'analisi della procedura *setInstructionHandler*) con undici gestori di istruzioni primarie (ovvero riconoscibili unicamente mediante il codice operativo) più la meta-istruzione *_instructionZero* che ha il compito, mediante l'analisi del codice funzione, di riconoscere le rimanenti sotto-istruzioni.

main_start

1) viene richiesto all'utente di inserire una stringa di trentadue bit;
2) viene allocato spazio (trentasei byte), in valore multiplo di quattro, nello stack per la memorizzazione della stessa;
3) viene letta una stringa di trentatré bit da console. Il valore trentatré per la syscall è stato scelto in base alla necessità di acquisire trentadue caratteri propri del codice binario più uno per il carattere di terminazione. In caso di input maggiore di trentadue caratteri la stringa viene troncata al trentaduesimo;
4) viene eseguita la procedura *checkString* per controllare la correttezza della stringa e normalizzarla dalla codifica ASCII ai rispettivi valori interi '0' e '1'. Qualora vengano riscontrate anomalie nel formato della stessa quali una lunghezza inferiore a trentadue o caratteri diversi da '0' o '1', viene data la possibilità all'utente di reinserirne una nuova o terminare l'esecuzione del programma;

5) viene stampato a schermo un messaggio di intestazione per la futura istruzione MIPS;
6) viene affidata l'analisi della stringa, in particolare del codice operativo, alla procedura generica *dispatchWork* che prende in ingresso l'indirizzo della stringa da analizzare, la lunghezza del codice operativo e l'indirizzo del vettore contenente i codici operativi riconosciuti. Se il codice è fra quelli supportati viene estratto l'indirizzo della procedura che lo gestisce e le viene passato il controllo per l'analisi dei rimanenti bit altrimenti viene visualizzato un messaggio di avvertimento e viene terminata l'esecuzione del programma.

DERIVAZIONI

Sono riportati alcuni alberi di derivazione secondo il normale flusso del programma in più situazioni.

Simboli:
+---# operazioni non precisate
+<--> jal
> j, jr o salti condizionati

Codice binario malformato:

```
main
    +<--> setInstructionHandler
    +<--> ... x10
    +<--> setInstructionHandler
main_start
    +---# read a string
    +<--> checkString
        +<--> checkStringByte
        +<--> ...x?
            > _checkStringByte_error
              _checkString_byte_error_continue
            # immissione da console di '1'
            > main_start
            +---# read a string
            +<--> ...
```

Istruzione non riconosciuta:

```
main
    +<--> setInstructionHandler
    +<--> ... x10
    +<--> setInstructionHandler
main_start
    +---# read a string
    +<--> checkString
        +<--> checkStringByte
        +<--> ...x30
        +<--> checkStringByte
    +<--> dispatchWork
        +<--> convertUBinFromStringToDec
            +<--> pow
            +<--> ... x4
            +<--> pow
        +<--> getInstructionHandler
            > _getInstructionHandler_error
            > main_exit
```

Istruzione riconosciuta:

```
main
    +<--> setInstructionHandler
    +<--> ... x10
    +<--> setInstructionHandler
main_start
    +---# read a string
    +<--> checkString
        +<--> checkStringByte
        +<--> ...x30
        +<--> checkStringByte
    +<--> dispatchWork
        +<--> convertUBinFromStringToDec
            +<--> pow
            +<--> ... x4
            +<--> pow
        +<--> getInstructionHandler
```

```
        +<--> $variableAddress
```

Istruzione BEQ (000100):

```
              _instructionBeq
            > _instructionB_abstract
            +<--> print_string
            +<--> print_space
            +<--> print_register_one
            +<--> print_separator
            +<--> print_register_two
            +<--> print_separator
            +<--> print_immediate
main_exit
```

Istruzione ADD (000000...100000):

```
              _instructionZero
            +<--> dispatchWork
                +<--> setInstructionHandler
                +<--> ... x7
                +<--> setInstructionHandler
                +<--> convertUBinFromStringToDec
                    +<--> pow
                    +<--> ... x4
                    +<--> pow
                +<--> getInstructionHandler
                +<--> $variableAddress
                      _instructionAdd
                    > _instructionRrr_abstract
                    +<--> print_string
                    +<--> print_space
                    +<--> print_register_three
                    +<--> print_separator
                    +<--> print_register_one
                    +<--> print_separator
                    +<--> print_register_two
main_exit
```

ISTRUZIONI

Segue la mappatura delle istruzioni riconosciute dallo sliMips:

NOOP

32 bit
0000 0000 0000 0000 0000 0000 0000 0000

JR $r1

6 bit	5 bit	21 bit
Codice Operativo	Registro $r1	0 0000 0000 0000 0000 1000

MULT $r1, $r2

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r1	Registro $r2	0000 0000 0001 1000

DIV $r1, $r2

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r1	Registro $r2	0000 0000 0001 1010

ADD $r1, $r2, $r3

6 bit	5 bit	5 bit	5 bit	11 bit
Codice Operativo	Registro $r2	Registro $r3	Registo $r1	00 00010 0000

SUB $r1, $r2, $r3

6 bit	5 bit	5 bit	5 bit	11 bit
Codice Operativo	Registro $r2	Registro $r3	Registo $r1	00 00010 0010

AND $r1, $r2, $r3

6 bit	5 bit	5 bit	5 bit	11 bit
Codice Operativo	Registro $r2	Registro $r3	Registo $r1	00 00010 0100

OR $r1, $r2, $r3

6 bit	5 bit	5 bit	5 bit	11 bit
Codice Operativo	Registro $r2	Registro $r3	Registo $r1	00 00010 0101

SLT $r1, $r2, $r3

6 bit	5 bit	5 bit	5 bit	11 bit
Codice Operativo	Registro $r2	Registro $r3	Registo $r1	00 00010 1010

J indirizzo

6 bit	26 bit
Codice Operativo	indirizzo

JAL indirizzo

6 bit	26 bit
Codice Operativo	indirizzo

BEQ $r1, $r2, offset

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r1	Registro $r2	offset

BNE $r1, $r2, offset

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r1	Registro $r2	offset

ADDI $r1, $r2, offset

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

SLTI $r1, $r2, offset

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

ANDI $r1, $r2, offset

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

LW $r1, offset($r2)

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

LB $r1, offset($r2)

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

SW $r1, offset($r2)

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

SB $r1, offset($r2)

6 bit	5 bit	5 bit	16 bit
Codice Operativo	Registro $r2	Registro $r1	offset

NOTE CONCLUSIVE

Non sono state riscontrate difficoltà rilevanti nella realizzazione del progetto o nell'implementazione del codice.

Gli unici problemi riscontrati sono derivati da alcune lievi incompatibilità fra i due emulatori di architettura MIPS presi in esame, ovvero Mars ed il noto Spim.
Quest'ultimo infatti ha mostrato di non supportare il noto carattere di escape "\" (in alcune occasioni particolari) ed ha avuto un comportamento inaspettato, del tutto fuori le linee guida della documentazione MIPS, in occasione della syscall per la lettura di un carattere da console.

Entrambi le incompatibilità sono comunque state superate efficacemente con dei workaround che hanno reso possibile l'uso del codice su entrambi gli emulatori.

Francesco Ricci
Daniele Orlando