Lorenzo Ferron, Gabor Galazzo, Alderico Gallo
Primo anno del Corso di Laurea in Informatica

ESERCIZIO 1: **Estensione del microinterprete**

Architettura degli Elaboratori 2
Esercizi di gruppo validi come esonero per la parte pratica dell'esame

Alessandria
Università degli Studi del Piemonte Orientale "Amedeo Avogadro"
A. A. 2017-2018

IL GRUPPO

Il nostro gruppo è costituito da tre persone:

- Lorenzo Ferron, il coordinatore;
- Gabor Galazzo;
- Alderico Gallo

Il lavoro è stato svolto utilizzando il software di controllo versione distribuito git sulla piattaforma online GitHub, sul quale è stata creata una repository contenete i tre esercizi proposti, che è raggiungibile al seguente link.

Data la scalabilità di ogni progetto vi era la possibilità che ogni componente potesse scegliere di affrontare ogni aspetto di esso in base alle competenze possedute. Si è proceduto quindi a suddividere i tre esercizi assegnandone ognuno ad ogni componente del gruppo: Lorenzo esercizio 1, Alderico esercizio 2 e Gabor esercizio 3. Ovviamente durante lo sviluppo dei singoli esercizi gli altri membri facevano da supporto nella realizzazione. Per ogni esercitazione si è proceduto nell'effettuare un'analisi collettiva del progetto, proponendo di conseguenza soluzioni e riflessioni all'incaricato, infine si è proceduto con una minuziosa suddivisione delle competenze per singolo esercizio.

Al termine di ogni step realizzativo dell'esercizio avvenivano dei confronti collettivi al fine di chiarire la situazione attuale e proporre miglioramenti e/o correzioni.

SPECIFICA DELLA NUOVA ISTRUZIONE

Operazione

Salta se la comparazione tra numeri interi (`int`) ha successo

Formato

`if_icmplt`
`byte1`
`byte2`

Indirizzo

if_icmplt = 161 (0xa1)

Operazione sullo stack

..., $mem[SP-1]$, $TOS \rightarrow$
...

Descrizione

Entrambi mem[SP − 1] e TOS devono essere numeri interi (`int`). Loro sono entrambi estratti dallo stack e comparati. La comparazione è segnata. Il risultato della comparazione è uguale:

- *if_icmplt* ha successo se e soltanto se $mem[SP-1] < TOS$

Se il confronto ha esito positivo, il *`byte1`* e *`byte2`* senza segno vengono utilizzati per creare un offset a 16 bit con segno, in cui l'offset è calcolato per essere (*`byte1`* `<<` 8) | *`byte2`*. L'esecuzione procede quindi a quell'offset dall'indirizzo dell'opcode dell'istruzione *if_icmplt*. L'indirizzo di destinazione deve essere quello di un codice operativo di un'istruzione all'interno del metodo che contiene questa istruzione *if_icmplt*.

Altrimenti, l'esecuzione procede all'indirizzo dell'istruzione che segue questa istruzione di *if_icmplt*.

DISCUSSIONE DEI PROBLEMI DA AFFRONTARE E DELLA SOLUZIONE INDIVIDUATA

L'estensione del linguaggio MAL è avvenuta in due momenti distinti. Dapprima si è sviluppata una versione che non considerava la possibilità di overflow. Questa prima implementazione è stata sottoposta a test in IJVM non automatizzati, verificato il suo corretto funzionamento si è proceduto nel miglioramento del codice sorgente MAL per considerare la **possibile** presenza di overflow, che sarà discusso in seguito.

Prima di illustrare l'algoritmo risolutivo, si faranno alcune considerazioni. La prima considerazione riguarda la natura dell'operazione, che estende il micro-interprete ovvero IF_ICMPLT. Essa realizza, come già illustrato nella precedente sezione, il confronto $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$. Si consideri adesso il risultato del confronto al variare dei segni dei due operandi:

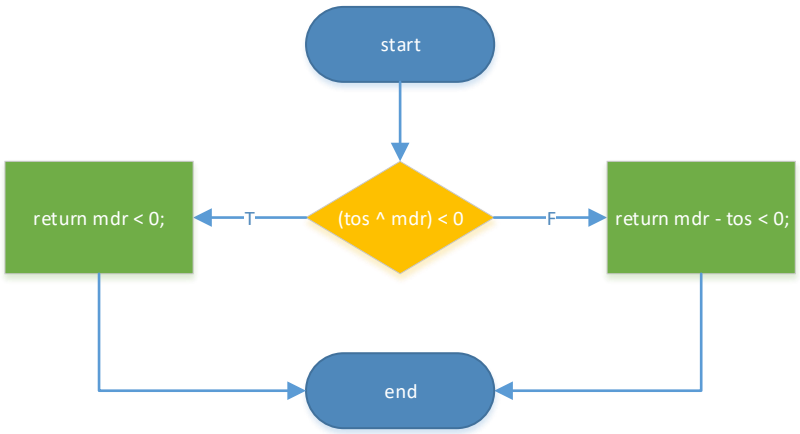
- se $\mathrm{TOS} < 0$ e $\mathrm{mem}[\mathrm{SP}-1] < 0 \Rightarrow \mathrm{mem}[\mathrm{SP}-1] - \mathrm{TOS} < 0$ **non** può generare overflow perché l'operazione di sottrazione è tra operandi di segno discorde;
- se $\mathrm{TOS} > 0$ e $\mathrm{mem}[\mathrm{SP}-1] > 0 \Rightarrow \mathrm{mem}[\mathrm{SP}-1] - \mathrm{TOS} < 0$ **non** può generare overflow (si veda il punto precedente);
- se $\mathrm{TOS} < 0$ e $\mathrm{mem}[\mathrm{SP}-1] > 0 \Rightarrow \mathrm{mem}[\mathrm{SP}-1] - \mathrm{TOS} < 0$ **può** generare overflow, eccedendo la dimensione di 4 byte (=32 bit) su cui un intero è rappresentato in IJVM; facendo però una considerazione matematica è intuitivo dire che il risultato di questo confronto è **falso**: infatti nessun numero negativo è maggiore di un numero positivo;
- se $\mathrm{TOS} > 0$ e $\mathrm{mem}[\mathrm{SP}-1] < 0 \Rightarrow \mathrm{mem}[\mathrm{SP}-1] - \mathrm{TOS} < 0$ **può** generare overflow, anche questo caso si comporta come il precedente, con l'unica differenza che il risultato del confronto è **vero**: infatti un numero positivo è sempre maggiore di qualsiasi numero negativo;

Dalle quattro casistiche esposte si può notare come l'overflow si **può** avere solo in presenza di operandi con segni discordi.

Dalle considerazioni fatte possiamo definire in linguaggio naturale l'algoritmo risolutivo. L'algoritmo può essere brevemente codificato nei seguenti punti:

1) determino se gli operandi sono di segno discorde o concorde: nel primo caso vado al punto 2, altrimenti vado al punto 3;
2) conoscendo la discordanza dei segni, quindi la possibilità di overflow, e tenendo a mente le considerazioni matematiche fatte, basta valutare il segno del primo operando ($\mathrm{mem}[\mathrm{SP}-1]$). Solo nel caso in cui si ha operando negativo vale $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$.
3) sapendo che non può esserci overflow, il risultato del confronto $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$ è determinato dal segno della differenza tra i due operandi. Solo nel caso in cui si ha risultato negativo vale $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$.

Quanto detto nel precedente paragrafo si può sintetizzare nel seguente flowchart:



e nel seguente frammento di codice C (disponibile nel file allegato *mal-test.c*):

```
bool cmp(int tos, int mdr) {
    if ((tos ^ mdr) < 0) /* equivale alle labels: if_icmplt6, POS, NEG */
        /* equivale a: OVERFLOW N = OPC; if (N) goto T; else goto F */
        return mdr < 0;
    /* equivale a: NO_OVERFLOW N = OPC - H; if (N) goto T; else goto F */
    return mdr - tos < 0;
}
```

L'algoritmo proposto risulta solo un'idea della soluzione, che non può essere implementata direttamente in MAL questo perché non si ha l'operazione di XOR (^). Vista la restrizione del linguaggio MAL, l'operazione di XOR sarà sostituita dai confronti sugli operandi per capire il loro segno (per una spiegazione approfondita del codice si faccia riferimento al codice del micro-interprete allegato).

COME È STATO SVOLTO IL TEST

La verifica del corretto funzionamento dell'istruzione implementata è avvenuta in tre fasi distinte, ricollegabili alle due versioni proposte. In una prima fase si è sviluppato un test non automatizzato, reperibile nel file *test1-IF_ICMPLT.jas*, eseguito sulla prima versione dell'implementazione, ovvero quella che non gestiva la possibilità di overflow. Il secondo test (si veda file *test1-IF_ICMPLT-overflow.jas*) è stato eseguito sulla versione definitiva dell'implementazione, cioè quella che tiene in considerazione il possibile overflow.

I test finora esposti presentano l'enorme svantaggio di richiedere una revisione del codice qualora si voglia cambiare il valore degli operandi da confrontare. Per ovviare a questo problema una prima soluzione sarebbe generare, con un linguaggio di alto livello (C, Python, Java, …), file jas contenenti operandi casuali, il confronto da fare e il risultato aspettato, inserito dal linguaggio ad alto livello. Questa tipologia di test risulta difficile da effettuare anche perché richiedere l'interazione dell'utente, visto che bisogna compilare ogni file jas prodotto e solo successivamente caricarlo nell'emulatore.

La soluzione da noi proposta (si veda *test2-IF_ICMPLT.jas*) si basa, ancora una volta, su osservazioni matematiche. Riprendendo il confronto di IF_ICMPLT, $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$, potremmo considerare TOS e $\mathrm{mem}[\mathrm{SP}-1]$ come due insiemi legati dalla seguente relazione:

"ogni elemento di TOS **è sempre maggiore** di qualsiasi elemento di $\mathrm{mem}[\mathrm{SP}-1]$"

Per definire gli insiemi (TOS e $\mathrm{mem}[\mathrm{SP}-1]$) si fanno uso di quattro costanti, due per insieme ciascuna delle quali indica il massimo e il minimo dell'insieme. Iterando con due cicli annidati si possono confrontare tra loro tutti gli elementi degli insiemi e qualora non vi sia fallimento terminare stampando "DONE", altrimenti non stampando nulla, ma permettendo di osservare la memoria al fine di trovare gli errori. È intuitivo che la condizione di errore, a meno di tale, non potrà mai essere incontrata dato che gli insiemi confrontati soddisfano sempre $\mathrm{mem}[\mathrm{SP}-1] < \mathrm{TOS}$.

Vi è ancora un'ulteriore considerazione che si può fare sul test, riguardo l'adattabilità alle altre istruzioni IJVM di confronto: IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IF_ICMPNE e IF_ICMPEQ ma questo implica una ridefinizione della relazione che intercorre tra gli insiemi (TOS e $\mathrm{mem}[\mathrm{SP}-1]$) a seconda dell'istruzione da testare.

Seppur corretto quest'ultimo test proposto, risulta efficiente solo quando si confrontano insiemi arbitrariamente piccoli, altrimenti è inefficiente. L'inefficienza è data dal fatto che il codice IJVM deve essere prima compilato e successivamente eseguito dall'emulatore, implicando che non vi è un'esecuzione diretta del codice da parte della CPU. Per quest'ultimo motivo si è deciso di effettuare un'ulteriore test, al fine di considerare insiemi sufficientemente grandi da garantire il corretto funzionamento dell'istruzione. Si è così implementata una versione del codice MAL in C (si veda *mal-test.c*), la quale è stata strutturata, al meno concettualmente, il più vicino possibile al codice MAL proposto, infatti: si è addirittura fatto uso del costrutto `goto`, oramai in disuso, per rispecchiare quello presente in MAL. Il vantaggio datoci dal codice C è la compilazione di quest'ultimo, la quale favorisce un'esecuzione diretta del codice da parte della CPU, ma soprattutto, grazie agli opportuni parametri di ottimizzazione passati al compilatore (si veda `-O2` in gcc) permette un'esecuzione fuori sequenza risultando, quindi, più veloce rispetto al codice IJVM.

COMMENTI CONCLUSIVI

La prima versione del codice MAL presentava un errore dovuto al fatto che, se i due operandi in cima allo stack erano ambedue negativi ed il TOS coincideva con il massimo valore rappresentabile in complemento a due su 32 bit, il risultato del confronto diventava imprevedibile. L'errore e la documentazione relativa sono presenti come commento all'interno del file *mic1ijvmC_SePT_OVERFLOW.mal* alla label NEG.

Di base il lavoro di gruppo e proceduto tranquillamente senza rilevanti intoppi.

Solo per il MAL sono state impiegate circa 12 ore di lavoro non sono state contate le ore di test, documentazione e revisione. Complessivamente lo svolgimento di questo esercizio ha richiesto 10 giorni.