Nicola Melloni (matr. 794324)

Guida al progetto di Architettura degli Elaboratori II

Introduzione

L' applicazione consiste di una calcolatrice a precisione multipla, cioè in grado di eseguire operazioni tra operandi di dimensione arbitraria e non limitati dalla dimensione dei registri del processore.

Tali operazioni vengono implementate tramite il riconoscimento dell' overflow e la gestione dei riporti tra operazioni aritmetiche nell' ISA MIPS, e si avvalgono di algoritmi per l' aritmetica a precisione multipla esposti in "The Art Of Computer Programming" di Donald E. Knuth, a cui si rimanda per un approfondimento più dettagliato delle nozioni matematiche e teoriche sottostanti.

Prima della descrizione delle varie funzioni contenute nel codice, riportiamo alcuni traguardi e benefici dell' implementazione data:

- **Efficienza:** Le funzioni implementate risultano molto efficienti dal momento che si avvalgono di istruzioni macchina e registri speciali non disponibili al programmatore di alto livello. Inoltre, tutte le funzioni si risolvono "in-place", cioè non utilizzano memoria addizionale oltre a quella data per il contenimento degli operandi e del loro risultato.

- **Sicurezza:** Le funzioni sono di sicur utilizzo, poichè la loro elaborazione avviene in modalità utente, l' overflow detection è realizzata attraverso codice non privilegiato anzichè tramite eccezioni hardware. Ciò evita che il progamma passi a codice kernel, col rischio di compromettere la stabilità e sicurezza del sistema operativo.

- **Scalabilità:** Le dimensioni degli operandi è limitata solo dalla quantità di memoria nel sistema. Maggiore memoria equivale a maggiori possibilità di calcolo, tuttavia gli algoritmi in questione sono eseguibili anche su macchine con memoria di capacità ridotta, come ad esempio in sistemi embedded.

- **Usabilità:** Le funzioni vengono utilizzate all' interno di un applicazione inerattiva ai comandi dell' utente, tuttavia è richiesto poco lavoro per la loro introduzione in una libreria C, utilizzbile per la stesura di codice ad alto livello.

Strategia implementativa

Un numero a multipla precisione big_int è implementato come array di word di 32 bit, e rappresenta un numero naturale espresso in notazione bianaria standard, dove i bit più significativi sono contenuti negli indirizzi più alti dell' array. Ad ogni elemento dell' array corrisponde una cifra; poichè una parola in memoria è di 32 bit, una cifra può avere 2^32-1 possibili valori.

Le procedure che operano tra big_int richiedono come argomenti due puntatori agli arrays che ospitano gli operandi e le rispettive dimensioni, più un puntatore ad un array in cui salvare il risulato. Tali funzioni, per ragioni di efficienza, non effettuano nessun controllo sui dati in input, in particolare sulla dimensione del buffer di destinazione, e pertanto possono incorrere in un buffer overflow; è responsabilità del chiamante utilizzare queste funzioni con parametri corretti.

Vengono riportate inoltre funzioni che operano tra un big_int e un intero unsigned contenuto in un registro. Queste procedure non vengono utilizzate dall' applicazione, possono tuttavia trovare spazio all' interno di una libreria per l' aritmetica a precisione multipla, ed essendo di fatto versioni semplificate degli algoritmi originali, possono risultare utili nella comprensione delle operazioni e nel debugging del programma

.

Descrizione del codice

Funzioni generiche

Funzione main, big_math_calculator e memzero

La funzione main non esegue nessun algoritmo specifico, limitandosi a salvare lo stato della macchina e a chiamare la funzione big_math_calculator, responsabile del flusso di controllo dell' intera applicazione. Tale scelta implementativa è data a garantire un inerfaccia più semplice e versatile all' invocazione della funzione principale,

La funzione big_math_calculator è responsabile di allocare la quantità di memoria necessaria alla computazione, leggere i dati immessi dall' utente ed eseguire l' operazione richiesta chiamando la procedura necessaria con i corretti parametri. Questa funzione ha un solo argomento *max_input_len* che specifica la dimensione dei numeri su cui si intende operare: attraverso la syscall **sbrk**, all' inizio della procedura vengono allocati due buffer di dimensione *max_input_len**4 bytes destinati a contenere gli operandi, più un terzo buffer di dimensione 2**max_input_len**4 bytes per il risultato. Vengono poi letti gli operandi in forma di stringa e convertiti in big_int attraverso la procedura *read_big_int*, che ne restituisce la dimensione esatta in bytes come numeri binari. Viene poi letto l' operatore e salvato in un registro, ed eseguita l' opportuna operazione. Viene inoltre data la possibilità di uscire dal calcolatore o di resettarlo, immettendo i comandi 'c' e 'q' in una qualsiasi fase d' elaborazione.

La funzione memzero non fa altro che azzerare buffer di memoria: a questa vengono èassati come argomenti un puntatore a un indirizzo di memoria e il numero di bytes (dev'essere un multiplo di 4) che si desiderano azzerare a partire da questo. La funzione viene invocata dalla procedura principale per pulire i buffer sui cui intnde scrivere, garantendo la correttezza delle operazioni.

Funzioni aritmetiche

• add_reg, add_big_int e sub_big_int

Queste funzioni implementano l' addizione tra un big_int e un intero in registro, tra due big_int, la sottrazione tra due big_int e hanno tutte un funzionamento simile. L' addizione avviene tra ogni cifra, partendo dalle meno significative, e il resto dell' operazione viene ricordato e addizionato al passo successivo.

L' implementazione è realizzata attraverso le funzioni aritmetico-logiche MIPS tra numeri senza segno (e che non sollevano un overflow exception) e un meccanismo di individuazione dell' overflow permesso dall' istruzione **sltu (set less than unsigned)**: se la somma tra due numeri è strettamente inferiore a uno dei due operandi, allora è avvenuto un overflow, e un resto uguale a 1 va ricordato.

Specifiche

- **add_reg(loc(op1), len(op1), op2, loc(res)**
  parametro 1: puntatore ad array contentente il big_int
  parametro 2: lunghezza dell' array (in bytes)
  parametro 3: intero da sommare
  parametro 4: puntatore a buffer di destinazine (può coincidere con il parametro 1)

- **add_big_int(loc(op1), loc(op2), loc(res), len) e sub_big_int(…)**
  parametro 1: puntatore ad array contentente il big_int che è primo operando
  parametro 2: puntatore ad array contentente il big_int che è secondo operando
  parametro 3: puntatore al buffer di destinazione (deve avere dimensione almeno pari a len+1)
  parametro 4: lunghezza (in bytes) degli operandi

• mul_reg e mul_big_int

Queste funzioni implementano la moltiplicazione tra due big_int e un big_int e un intero in registro. Ancora una volta, la funzione è implementata attraverso la gestione dei riporti tra operazioni di moltiplicazione cifra per cifra, similmente a quanto si farebbe con carta e penna.

Questa volta il resto non è di un singolo bit bensì di una parola di 32 bit, e per l' implementazione della procedura risulta di cruciale importanza l' impiego dei registri speciali **$hi $lo** e dell' istruzione **maddu (multiply and add unisgned)**: il risultato dell moltiplicazioni cifra a cifra viene memorizzato viene in questi registri; la cifra del risultato attesso viene poi peevato dal registro $lo, mentre nel registro $hi sarà contenuto il resto, da sommare alla moltiplicazione successiva.

Specifiche

- **mul_reg(loc(op1), len(op1), op2, loc(res))**
  parametro 1: puntatore ad array contentente il big_int
  parametro 2: lunghezza dell' array (in bytes)
  parametro 3: intero da moltiplicare
  parametro 4: puntatore a buffer di destinazine (può coincidere col parametro 1)

- **mul_big_int(loc(op1), loc(op2), loc(res), len(op1), len(op2))**
  parametro 1: puntatore ad array contentente il big_int che è primo operando
  parametro 2: puntatore ad array contentente il big_int che è secondo operando
  parametro 3: puntatore al buffer di destinazione
  (deve avere dimensione alemeno pari len(op1)*len(op2))
  parametro 4: lunghezza (in bytes) del primo operando
  parametro 5 (su stack): lunghezza (in bytes) del secondo operando

• div_reg e div_big_int

L' implementazione delle operazioni di divisione e modulo risulta notevolmete più complessa poichè l' architettura MIPS di riferimento non fornisce meccanismi hardware atti a semplificarne lo sviluppo (come

ad esempio viene fatto dalle architetture x86 attraverso una dedicata istruzione di dvisione tra operandi a 64 bit). Per questa ragione, va in questo caso ridotta la dimensione delle cife di un big_int da 32 a 16 bit.

L' algoritmo di divisione implementato utilizza un meccanismo di normalizzazione e divisione degli operandi (realizzato tramite shift sinistro e destro, rispettivamente) al fine di ridurre il numero di tentativi da eseguire nel riconosimento delle cifre del quoziente: l' algoritmo infatti prova ad indovinare (con un errore al massimo pari a 2) quale possa essere la cifra del quoziente, effettuando poi una moltiplicazione e sottrazione del divisore al dividendo al fine di verificare l' esattezza della stima, decrementando la cifra approssimata se questa risulta troppo grande (cioè se la sottrazione va in overflow) e risommando il divisore al resto parziale della divisone. Anche in questo caso l' operazione avviene in modo molto simile a quanto faremmo con carta e penna.

Specifiche

- **div_reg(loc(op1), len(op1), op2, loc(res))**
  parametro 1: puntatore ad array contentente il dividendo
  parametro 2: lunghezza del dividendo (in bytes)
  parametro 3: divisore
  parametro 4: puntatore a buffer di destinazine per il quoziente (può coincidere col parametro 1)

- **div_big_int(loc(op1), loc(op2), loc(quot), loc(rem), len(op1), len(op2))**
  parametro 1: puntatore ad array contentente il divisore
  parametro 2: puntatore ad array contentente il dividendo
  parametro 3: puntatore al buffer di destinazione per il quoziente
  (di dimensione almeno paria a len(op1))
  parametro 4: puntatore al buffer di destinazione per il resto
  (di dimensione almeno paria a len(op1))
  parametro 5 (su stack): lunghezza (in bytes) del divisore
  parametro 6 (su stack): lunghezza (in bytes) del dividendo

Funzioni di input/output

• write_big_int

La funzione write_big_int permette la conversione di un big_int in una stringa di cifre decimali. Per il calcolo di tali cifre viene utilizzato l' aloritmo simile a quello di Euclide: il numero viene diviso per 10, e sostituito dal quoziente; il resto della divisione rappresenta una cifra decimale del numero in questione, e viene salvata sullo stack come carattere ASCII per essere poi stampato a video nella corretta posizione. Il procedimento viene ripetuto fino a che il quoziente non diventa pari a zero.

Va infatti notato che la funzionne "consuma" il suo operando. Per tal motivo , nel caso sia d' interesse riutilizzare tale numero (ma non lo è per la nostra applicazione), questo deve essere prima copiato in una zona di memoria dedicata.

La divisone per dieci è resa possibile da un algoritmo più più efficace e intuitivo di quello descritto sopra per la divisione di un bigint e un intero, noto come ***short division,*** che ha tuttavia lo svantaggio di poter operare sulla nostra architettura con un divisore non superiore a 2^16-1.

Specifiche

- **write_big_int(loc(buf), len(buf)**
  parametro 1: puntatore ad array contentente il big_int
  parametro 2: dimensione in bytes del buffer
  valore di ritorno (in $v0): dimensione reale del big_int in buffer

• read_big_int

La funzione read_big_int converte una stringa decimale in un big_int. Il suo funzionamento si basa su un meccanismo di addizione e moltiplicazione per 10 non molto diverso da quello dato al passo di moltiplicazione/sottrazione nella divisione tra big_int: le cifre della stringa decimale vengono addizionate una alla volta, moltiplicando per 10 il risulato precedentemente ottenuto. La procedura inoltre si accerta dlla validità del suo argomento tramite strutture di controllo, ritornando nei registri $v0 e $v1 valori numerici identificativi dell' errore.

Specifiche

- **read_big_int(loc(str), len(str), loc(buf), len(buf))**
  parametro 1: puntatore a una stringa decimale
  parametro 2: dimensione della stringa
  parametro 3: puntatore a buffer di destinazione
  parametro 4: dimensione in bytes del buffer di destinazione
  valori di ritorno:
  - $v0 = n, $v1 = 0: la funzione ha successo e n è la dimensione reale del big_int letto
  - $v0 = -1, $v1 = 0: la funzionne fallisce a causa di un buffer overflow
  - $v0 = -1, $v1 = 1: la funzione fallisce a causa dell' immissione un carattere errato

Conclusione

Il codice può essere migliorato ed espanso in vari modi: va in primo luogo notato che è stata favorita, in fase di sviluppo, la chiarezza e semplicità implementativa all' efficienza, motivo per cui il codice manca di tante piccole ottimizzazioni; ho ritenuto infatti che queste fossero da applicare a un' ipotetica seconda versione del programma, a seguito di una verifica di correttezza degli algoritmi espressi nella forma più chiara possibile e il più possibile fedeli alla loro formulazione originale sul testo di Knuth.

Una possibile espansione dall' applicazione potrebbe essere data dall' implementazione di funzioni per l' aritmetica tra numeri arbitrariamente grandi con segno. Potrebbe inoltre essere interessante creare un interfaccia C per le funzioni implementate, ad esempio avvalendosi dei costrutti struct e typedef o delle funzioni per l' allocazione dinamica della memoria.