Praktikum Rechnerarchitektur Sommersemester 2024

Praktikum 1

Ausgabedatum: 08.04.2024

Übung 0:

Machen Sie sich mit dem Mars MIPS-Simulator vertraut. Am besten, Sie installieren ihn auch auf Ihrem persönlichen Rechner, so dass Sie jederzeit damit üben können.
Er ist unter https://courses.missouristate.edu/KenVollmar/MARS/ downloadbar.

Übung 1:

a) Machen Sie sich mit dem arrayinit.asm Programm vertraut, laden Sie es in den Mars Assembler und führen Sie es aus. Es füllt ein Array von 100 Integer-Werten (in Assembler gibt es keine Arrays, also in Wirklichkeit nur einen zusammenhängenden Speicherbereich von 400 Byte mit 100 Integer Werten von 0 bis 99 und gibt diese aus.

b) Modifizieren Sie das Programm so, dass es die Quadratzahlen von 0 bis 99*99 ausgibt. Verwenden Sie dazu den Integermultiplikationsbefehl „`mul`“.

c) Das Programm aus Teil a) funktioniert, ist aber inkorrekt da es nicht der ABI entspricht: es verwendet die *saved* Register ohne diese auf dem Stack zu sichern. Korrigieren Sie diesen Fehler dadurch, dass Sie die entsprechenden Register sichern und vor das Programm endet, wiederherstellen.

Übung 2:

Implementieren Sie das Sieb des Eratosthenes um Primzahlen zu berechnen. Ihnen ist eine C Version des Algorithmus gegeben, das bereits so in kleine C Funktionen unterteilt wurde. dass es sich einfach in MIPS Assembler übertragen lässt. Sie können dazu auch das Gerüst aus Aufgabe 1 verwenden, in dem ja bereits ein Array für Sie angelegt wurde. Das Hauptprogramm, das unten steht, zeigt ihnen welche Hilfsfunktionen zu implementieren sind.

```
int main() {
    int current_prime = 2;
    init_array();

    do {
        eliminate_multiples(current_prime);
        current_prime = next_prime(current_prime);
    } while (current_prime * current_prime < size);

  print_primes();
  return 0;
}
```

Hier ist die Initialisierungsfunktion:

```
void init_array() {
 for (int i = 0; i != size; i++) {
    sieve[i] = 1;
  }
  sieve[0] = 0;
  sieve[1] = 0;
}
```

Wobei das Array und die size Variable global definiert sind:

```
#include<stdio.h>

int sieve[1000];
int size = 1000;
```

Weitere Funktionen sind:

```
void eliminate_multiples(int factor) {
    for (int i = factor + factor; i < size; i += factor) {
    sieve[i] = 0;
  }
}
```

welche Vielfache einer Zahl im Sieb als faktorisierbare Zahlen markiert und die Funktion

```
int next_prime(int p) {
    for (int i = p + 1; i < size; i++) {
      if (sieve[i]) {
        return i;
      }
    }
    // should not get here, but return -1 if it happens
    return -1;
}
```

welche das bereits teilweise berechnete Sieb verwendet, um die nächstgrößere Primzahl zu finden, um weiter zu sieben.
Folgende Funktion schließlich druckt die ausgesiebten Primzahlen aus:

```
void print_primes() {
    // now exactly the array indices of the array that are prime will
have contents of 1
    printf("Primes up to %d:\n", size);
    for (int i = 0; i != size; i++) {
        if (sieve[i]) {
            printf("%d ", i);
        }
    }
}
```

1. **WICHTIGE HINWEISE zur Einhaltung der ABI (Ihr Programm ist falsch, wenn Sie diese nicht einhalten, auch wenn es zufälligerweise funktionieren sollte.)**
   Damit Funktionsaufrufe nicht ihre lokalen Variablen überschreiben, sollten Sie current_prime in ein zu sicherendes Register (saved register) ablegen ($s0 - $s7).
   a. Denken Sie dann daran, dass wenn Sie ein zu sicherndes Register verwenden, dieses vor der Verwendung auf dem Stack sichern und vor Rückkehr aus der Funktion wieder herstellen müssen!

2. Auch Systemfunktionen wie z.B. die syscall Funktionen können Ihre Register überschreiben! Sollte das nicht passieren, haben Sie nur Glück gehabt, außer Sie haben zu sicherende Register für Werte verwendet, die Sie nach dem Funktionsaufruf (z.B. syscall) noch benötigen. Beachten Sie auch dann den Hinweis 1a.

3. Denken Sie daran, dass Funktionsargument in den Argumentregistern ($a0, $a1 etc. zu übergeben sind, also wenn es nur ein Funktionsargument gibt, wie in dem vorgegebenen C Programm, muss dieses in Register $a0 übergeben werden.
4. Funktionsresultate (z.B. der Ergebniswert von `next_prime`) müssen im Result-Register $v0 zurückgegeben werden.
5. Beachten Sie, dass der Assembler den Code in der ersten Funktion (also die zuerst im .text Segment steht) startet. Platzieren Sie also `main` zu Anfang oder fügen Sie ganz am Anfang einen unbedingten Sprung (`j main`) hinzu.

6. Eine Funktion müssen sie mit `jal` aufrufen (nicht `j`!) und zurück (return) kommen Sie mit `jr $ra`.
    a. Da `jal` das `$ra` Register mit der Rückkehradresse überschreibt, müssen Sie dieses sichern, wenn ihre Funktionen andere Funktionen aufrufen. Für syscall Aufrufe ist das allerdings nicht notwendig.
7. Das return-Statement von main müssen Sie (im Mars Simulator) nicht mit `jr` implementieren, rufen Sie einfach direkt den `exit` syscall auf.