Praktikum Rechnerstrukturen

Bogen 4

I/O – Stack
Erweiterungen

Department Informatik, AB TAMS
MIN Fakultät, Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

1 Ein- und Ausgabe

Gerade bei der Programmierung von Treibern zur Ansteuerung von I/O-Geräten wird Assemblerprogrammierung oft eingesetzt. Beim Konzept von *memory mapped I/O* werden die Geräte an bestimmten Adressen in den Adressraum des Prozessors eingeblendet und vom Prozessor direkt mit Lese- und Schreibbefehlen angesprochen. Zur Demonstration dieses Verfahrens enthält das Hades-Design `processor-io.hds` neben RAM und ROM zusätzlich zwei I/O-Komponenten: ein einfaches Register zur Ansteuerung einiger LEDs und ein ebenfalls parallel angesteuertes Terminal. Der Adressdekoder in diesem Entwurf ist so eingestellt, dass das ROM im Adressbereich von `0x0000..0x6ffe` aktiviert wird, das Terminal bei Adresse `0x7000`, die LEDs bei Adresse `0x7002` und das RAM im Bereich von `0x8000..0xfffe`.

Sie können aber diese Aufgaben aber auch mit dem Assembler lösen, der im Emulatorfenster ebenfalls eine Siebensegmentanzeige und ein Terminal hat.

Aufgabe 1.1: Ansteuerung der LEDs Schreiben Sie eine Funktion, um den Inhalt von `R10` auf den LEDs auszugeben. Demonstrieren Sie die Funktion, indem Sie in einer Endlosschleife einen Zähler inkrementieren und den aktuellen Wert dieses Zählers jeweils auf den LEDs ausgeben.

Aufgabe 1.2: Ansteuerung von Speicher vs. I/O Machen Sie sich an dieser Stelle klar, dass trotz der identischen Ansteuerung ein fundamentaler Unterschied zwischen Speicher und I/O-Komponenten wie dem Terminal oder einem Drucker besteht: mehrfaches Schreiben einer Speicherstelle mit demselben Wert bewirkt keine Änderung, während mehrfaches Schreiben eines Wertes in ein I/O-Geräts durchaus mehrfache Wirkung haben kann. Entsprechendes gilt für das Lesen; insbesondere dürfen Speicherbereiche mit I/O-Geräten normalerweise nicht vom Cache abgedeckt werden!

Dies gilt auch für die hier verwendete Ansteuerung des Terminals. Dieses verfügt neben den acht Datenleitungen, die direkt über die Datenbits `7..0` angesprochen werden, zusätzlich über zwei Steuerleitungen `nreset` und `strobe` (clk), die an die Datenbits `9` und `8` angeschlossen sind, siehe Abbildung 1. Für jedes Zeichen muss ein `clk`-Impuls erzeugt werden, indem das Datenbit `8` zuerst auf `0` und dann auf `1` gesetzt wird. Für den normalen Betrieb muss das `nreset` Bit auf `1` gesetzt werden (ansonsten wird der gesamte Bildschirm gelöscht).

Zum Beispiel müssen zur Ausgabe des Zeichens „A“ (Ascii-Code 65 bzw. 0x41) nacheinander die Werte `0x0241`, `0x0341`, `0x0241` an die Adresse `0x7000` geschrieben werden, wobei die dritte Ausgabe auch unterbleiben kann.

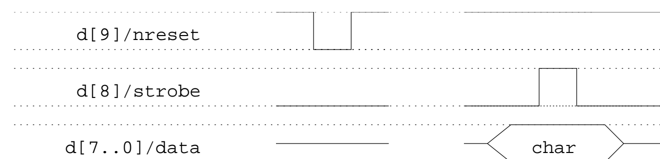

Abbildung 1: Ansteuerung des parallelen Terminals: alle Zeichen löschen (links), Ausgabe eines Zeichens (rechts)

Aufgabe 1.3: putc() Schreiben Sie ein Unterprogramm, um den Inhalt der unteren 8 Bits von `R10` auf das parallele Terminal (an Adresse `0x7000`) auszugeben.

Testen Sie Ihr Unterprogramm, indem Sie in einem Hauptprogramm in einer Endlosschleife einen Zähler inkrementieren und den aktuellen Wert dieses Zählers jeweils auf des Terminal ausgeben.

Falls beim Testen Ihres Programms mit HADES Probleme auftreten, weil das Terminal in einen undefinierten Zustand übergeht, kontrollieren Sie den Microcode für den `stw`-Befehl: Der Datenbus darf sich nicht gleichzeitig mit der `nWE` oder `nOE` Leitung ändern.

Aufgabe 1.4: puts() Schreiben Sie jetzt ein Unterprogramm `puts()`, um unter Verwendung des Unterprogramms `putc()` eine Zeichenkette auf einen Drucker (bzw. ein Terminal) auszugeben. Das Argument mit dem Zeiger auf die auszugebende Zeichenkette wird wiederum in `R10` übergeben.

2 Speicherbereiche und Stack

Da beim von-Neumann-Rechner sowohl die Programme als auch alle Daten im Hauptspeicher liegen, ist die Organisation des Speichers von zentraler Bedeutung. Die in Unix übliche Konvention zur Einteilung der Speicherbereiche ist in Abbildung 2 gezeigt. Dabei werden die folgenden Speicherbereiche (*Segmente*) unterschieden:

- Das *Textsegment* enthält den eigentlichen Programmtext mit allen Befehlen. Sofern keine selbstmodifizierende Programme zum Einsatz kommen, bleibt das Textsegment während des Programmablaufs unverändert. Es wird häufig ab unteren Ende des Speichers abgelegt.
- der *constant pool* (Konstantenbereich) nimmt alle Konstanten und statischen Variablen des Programms auf. Typ und Anzahl dieser Variablen ergeben sich unmittelbar aus dem Programm. Der Speicherplatz für diese Variablen wird normalerweise direkt oberhalb des *Textsegments* angelegt.
- der *Heap* (Halde) nimmt alle dynamisch zur Laufzeit des Programms erzeugten Variablen bzw. Objekte auf. Der Heap wird oberhalb des Konstantenpools angelegt und wächst nach oben. Für den Heap werden (Betriebssystem-) Funktionen benötigt, um freie Speicherbereiche für neu anzulegende Variablen zu finden und diese auch wieder freigeben zu können.
- der *Stack* (Stapel) wird für die Parameterübergabe zwischen Funktionen und für die Speicherung der lokalen Variablen der einzelnen Funktionen benutzt. Der Stack wird häufig ab oberen Ende des zur Verfügung stehenden Speichers angelegt und wächst mit jedem Aufruf nach unten.

Der im Befehlssatz des D·CORE definierte Befehl `JSR` speichert die Rücksprungadresse immer in Register `R15`. Das bedeutet, dass ohne weitere Maßnahmen immer nur höchstens ein Unterprogramm aufgerufen werden kann, da sonst der zweite Aufruf die Rücksprungadresse des ersten Aufrufs überschreibt. Für geschachtelte Aufrufe muss daher ein *Stapel* bereitgestellt und vom Anwenderprogramm aus verwaltet werden.

Wie bei fast allen RISC-Prozessoren (außer SPARC), gibt es im Befehlssatz keine weitere Unterstützung für die Stack-Verwaltung. Die Motivation ist, dass der Compiler oft in der Lage ist, soweit

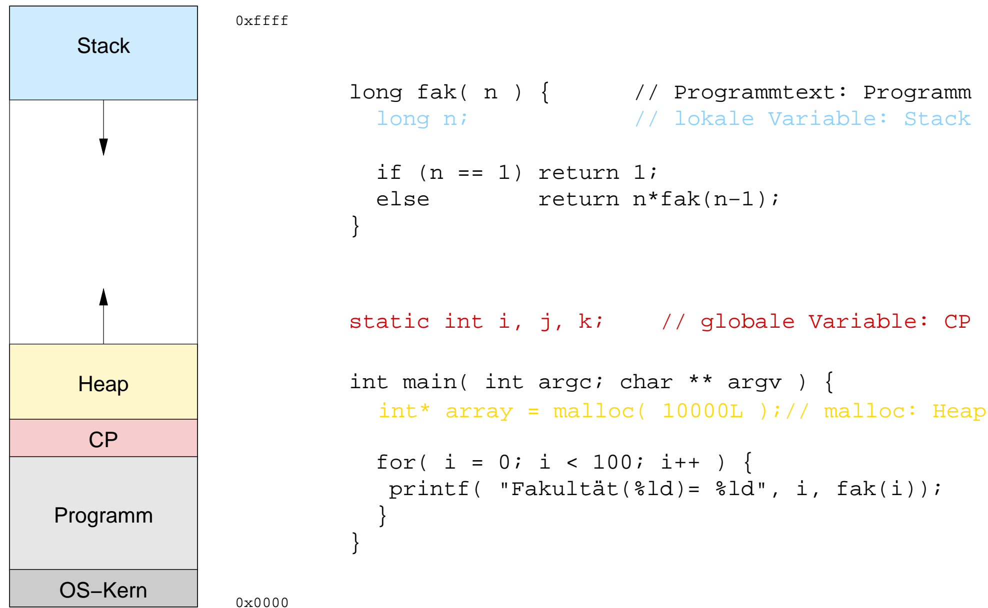

Abbildung 2: Speicherbereiche im Hauptspeicher: Textsegment, Konstantenpool, Heap, Stack

möglich alle Parameter über Register zu übergeben und den Stack nur verwendet, wenn sich dies nicht vermeiden lässt.

Aufgabe 2.5: Stack Machen Sie sich aus den Vorlesungsskripten die Funktion eines Stacks klar. Was könnten in diesem Zusammenhang folgende Begriffe bedeuten, wenn es um Informationen (z.B. Registerinhalte) geht, die noch benötigt, bzw. überschrieben werden:
caller save:
callee save:

Mit welchen Befehlen kann der D·CORE-Stackpointer auf den in Abbildung 3 verwendeten Wert von `0xfffe` initialisiert werden?

Aufgabe 2.6: push() In den meisten Situationen müssen nicht alle sondern nur einige Register auf den Stack gesichert werden. Notieren Sie als Beispiel die Assemblerbefehle, um den Inhalt der Register `R4`, `R5`, `R10` auf den Stack zu sichern. Per Konvention soll Register `R0` als Stackpointer verwendet werden. Wie behandeln Sie den Stackpointer?

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			

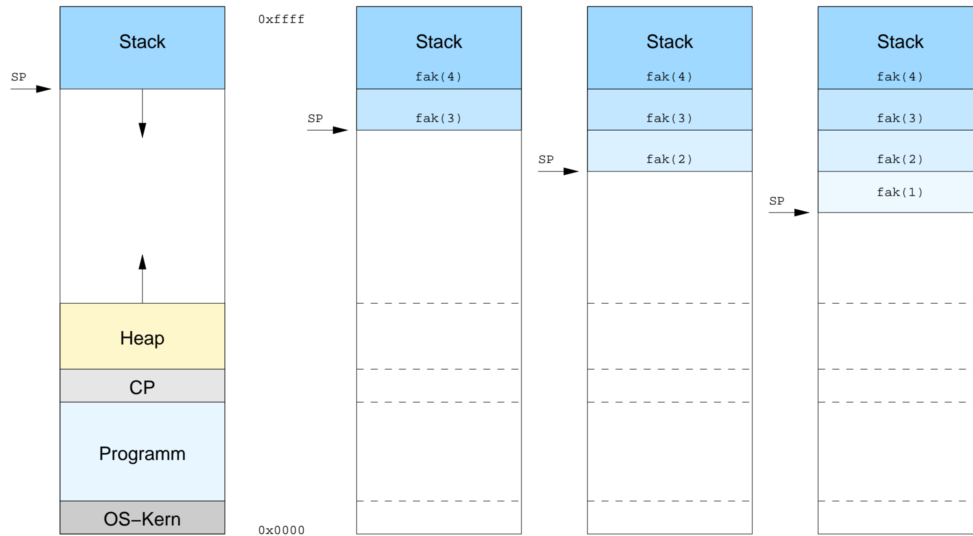

Abbildung 3: Rekursiver Aufruf der Faktultätsfunktion.

Aufgabe 2.7: pop() Notieren Sie die notwendigen Assemblerbefehle, um den Inhalt der Register `R4`, `R5`, `R10` vom Stack wiederherzustellen:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			

Viele gute Assembler stellen entsprechende Macros zur Verfügung, wobei die betroffenen Register als Argumente übergeben werden. Das gilt für das von uns verwendete Programm. Hier heißen sie `.push` und `.pop`. Als Stackpointer wird als Default das Register R0 angenommen, das auf die zuletzt beschriebene Speicherstelle zeigt. Falls Sie ein anderes Register als Stackpointer verwenden möchten (z.B. das R14), können sie dies dem Assembler mit `.stack R14` mitteilen. Vergessen Sie bitte nicht, ihren Stackpointer auf einen definierten Wert (z.B. 0) zu initialisieren.

Aufgabe 2.8: Rekursive Unterprogramme

Setzen Sie folgendes C-Programm in ein Assembler-Programm für den D·COREum und testen Sie es. Auch wenn Sie nur JAVA kennen, sollten Sie das Programm nachvollziehen können.

Multiplikation und Division durch 2 lässt dabei auf Bitebene leicht durch einen einzigen Assemblerbefehl realisieren.
Das Ergebnis soll am Schluss in einem Register stehen.

```
  int rpm( int a, int b) {

    if (b == 0)
      return 0;
    else {
      int erg = rpm( 2*a, b / 2);

      if ((b % 2) == 1)     // Ist b ungerade?
          erg= erg + a;
      return erg;
    }
  }

void main() {
    rpm( 4, 6);
    }
```

Testen Sie Ihr Programm auch noch für andere Eingaben.

Was berechnet die Funktion `rpm`?

3 Erweiterungen / Verbesserungen

Eigentlich wäre es wünschenswert, nach der *Assembler-* jetzt die darüber liegende *Compilerebene* zu betrachten.

Wir werden dies schon allein deshalb nicht tun, weil es zwar relativ leicht ist, einen Assembler zu schreiben, während das bei einem Compiler ganz anders aussieht.

Stattdessen sollen Sie an dieser Stelle die Gelegenheit haben, eigene Erweiterungen oder Verbesserungen für unseren Prozessor vorzuschlagen und (so weit möglich) zu implementieren.

Mögliche Fragen wären:

a) Lässt sich der Befehlssatz durch weitere Befehle sinnvoll erweitern?

Machen Sie sich dabei immer klar, ob sich diese neuen Befehle auf der gegebenen Hardware ausführen – was natürlich wünschenswert wäre – oder muss man auch diese erweitern, und wenn ja, wie.
Überlegen Sie sich das Befehlsformat (Kodierung) Ihrer neuen Befehle. das Mikroprogramm, das sie realisiert, und auch ein kleines Testprogramm.

Achtung: Erweiterungen der ALU zählen nicht mit, weil sich diese nicht sinnvoll in unser Design einbauen und testen lassen!

b) Ist irgendeine Komponente in unserem Prozessor überflüssig und lässt sich der Prozessor schneller machen, indem man auf Sie verzichtet?

c) Lässt sich der Prozessor schneller machen, indem man die Hardware etwas umorganisiert?

Machen Sie **zwei** solcher Vorschläge. Bevor Sie sie realisieren, diskutieren Sie sie bitte mit den Betreuern.