Rechnerarchitektur – Praktische Übungen

Übung 4: Bestimmung von CPI und MIPS Werten, Akkumulator-/ Stack-Maschine, Bedingte Befehle, Endianness

Aufgabe 4.1: Stack-Maschine (3 Punkte)

Gegeben sei eine Stack-Maschine mit folgenden Befehlen für die Grundrechenarten mit ganzen Zahlen:

- Grundrechenarten ADD, SUB, MUL, DIV und MOD (linker Operand liegt auf dem Stack unter dem rechten)
- Konstante auf den Stack: PUSH
- Entfernen des obersten Stack-Elementes: POP

Schreiben Sie jeweils ein Programm für die Stack-Maschine, das folgende Terme berechnet:

a) 40/5+2

b) (40/(5+2))+11

Aufgabe 4.2: Bedingte Befehle (3 Punkte)

Ergänzen Sie folgendes `MMIX`-Programm so, dass anschließend der Größte der drei Werte aus den Registern $1, $2, $3 im Register `max` steht. Verwenden Sie dazu keine Sprünge.

```
max         IS          $0
            LOC         #100
Main        SET         $1,19       Beliebige Startwerte…
            SET         $2,42
            SET         $3,11
```

Aufgabe 4.3: Endianness

a) Die 32-Bit Hexadezimalzahl 0xAFFE0815 soll beginnend bei Adresse 180 im Speicher eines Rechners abgelegt werden. Vervollständigen Sie die untenstehenden Tabellen für die Fälle, dass das System nach „big endian“ bzw. nach „little endian“ Verfahren arbeitet.

Big Endian	
Speicheradresse	**Wert**
180	
181	
182	
183	

Little Endian	
Speicheradresse	**Wert**
180	
181	
182	
183	

b) Recherchieren Sie, welche Byte-Ordnung bei folgenden Fällen verwendet wird:
   a. Intel 80x86
   b. MMIX
   c. ARM
   d. OpenRISC
   e. JPEG-Bild
   f. GIF-Bild
   g. IP-Adresse im IP Header eines IP-Paketes

c) Was müssen Sie tun, wenn Sie ein Dateiformat, welches „big endian“ Darstellung verwendet auf einem „little endian“ System lesen wollen?

Aufgabe 4.4 (praktisch): Bestimmung von CPI Werten

Im vorherigen Aufgabenblatt haben Sie mit Hilfe des Tools `orpmon` auf dem OpenRISC Modul die CPU-Benchmarks *coremark* und *Dhrystone* ausgeführt und dessen Leistung bestimmt. In dieser praktischen Aufgabe wird jetzt das Tool `orpmon` schrittweise um neue Befehle zur Bestimmung von CPI-Werten für ausgewählte Befehle erweitert. Um die/den zu messenden Maschinenbefehl exakt vorgeben zu können, wird dabei eine Möglichkeit genutzt, innerhalb von C-Code auch Assemblerbefehle einzufügen, die dann vom Compiler exakt so übernommen werden. Diese Methode wird auch als *„Inline-Assembler“* bezeichnet und findet bei der Programmierung von Treibern und ähnlichen hardwarenahen Programmen häufig Anwendung.

Schritt 1: Erzeugung einer eigenen orpmon-Variante für zusätzliche Testbefehle

- Wechseln Sie in das Verzeichnis mit dem orpmon-Quellcode: `~/soc-design/orpmon`
- Prüfen Sie durch den Aufruf von
  ```
  make clean
  make
  ```
  dass sich der Original-Quellcode ohne Fehler übersetzen lässt. Nach einer erfolgreichen Übersetzung des Programmes sollten Sie im Verzeichnis die Datei `orpmon.or32` finden. Diese

enthält den Binärcode des orpmon Programmes und lässt sich sowohl auf dem ORSoC-Modul als auch im Simulator ausführen.

- Unterhalb des Verzeichnisses `orpmon` finden Sie im Verzeichnis `cmds` die Quellcode-Dateien, die die Befehle von `orpmon` implementieren. Legen Sie dort eine neue Datei `hmcmd.c` an, in der später die neuen Befehle implementiert werden sollen. Kopieren Sie folgendes Gerüst in die Datei – sie finden die Datei auch auf Moodle zum Herunterladen.

```
#include "common.h"
#include "support.h"
#include "spi.h"

int hmcmd_cmd(int argc, char *argv[])
{
        if (argc == 0) {
                printf("usage:  […insert usage information here…] \n");
        } else if (argc == 1) {
                if (!strcmp(argv[0], "hello")) {
                        printf("Hello from University of Applied Sciences, Munich, Germany! \n");
                }
        } else {
                printf("Invalid number of arguments!\n");
        }

        return 0;
}

void module_hmcmd_init(void)
{
        register_command("hmcmd", "",
             "additional commands and tests by University of Applied Sciences Munich", hmcmd_cmd);
}
```

Figure 1: Vorgabe - Gerüst der Datei hmcmd.c

- Damit Ihre neue Datei auch kompiliert wird, muss sie dem `Makefile` im gleichen Verzeichnis hinzugefügt werden. Fügen Sie dort einen Verweis auf das zu erzeugende Objektfile an der passenden Stelle ein.
- Nun müssen Sie noch sicherstellen, dass die Initialisierungsmethode `module_hmcmd_init()` auch aufgerufen wird. Dies können Sie tun, indem Sie der Datei `orpmon/commons/commons.c` einen Aufruf von von `module_hmcmd_init()` an der passenden Stelle hinzufügen.
- Kompilieren und linken Sie Ihre modifizierte orpmon Variante durch den bereits vom Anfang bekannten Aufruf von `make`, so dass Datei `orpmon.or32` erneut erzeugt wird.
- Testen Sie Ihre modifizierte orpmon Variante nun innerhalb des Simulators
  `sim -f ~/soc-design/linux/arch/openrisc/or1ksim.cfg orpmon.or32`

  Wenn Sie alle Teile erfolgreich durchgeführt haben, sollte nun ein neues Fenster mit der Anzeige des Simulators geöffnet werden. Rufen Sie dort die orpmon Hilfe auf (hier sollte nun ein neuer Eintrag für `hmcmd` enthalten sein) und rufen Sie

  ```
  ORSoC ORDB2A> hmcmd hello
  ```

  auf.

Einschub: Einführung in Inline-Assembler

Die grundlegende Idee von Inline-Assembler ist, dass sich in der Regel der Großteil eines Programmes am angenehmsten in einer Hochsprache (wie C oder C++) programmieren lässt – welches man nur an einzelnen Stellen, an denen beispielsweise aus Gründen der Geschwindigkeit oder der Hardwarenähe volle Kontrolle über den erzeugten Maschinencode gewünscht ist, durch manuell vom Programmierer eingefügte Assemblerbefehle ergänzt. Dazu unterstützen alle gängigen C-Compiler sogenannte Inline-Assembler Instruktionen.

Einfaches Inline

Bei einer einfachen Inline-Assembler-Anweisung werden keine Parameter zwischen dem Assemblerbefehl und dem umgebenden Hochsprachenprogramm ausgetauscht. Hierzu nutzen Sie einfach das Schlüsselwort **`asm`** gefolgt von dem einzufügenden Assemblerbefehl.

```
asm("Assemblerbefehl");
```

Beispiele:

```
asm("l.nop");
```

Falls der Assemblerbefehl Registerwerte ändert, können Sie die modifizierten Register folgendermaßen angeben, so dass der Compiler dieses berücksichtigen kann:

```
asm("l.add r0,r1,r2" ::: "r0");
```

Nutzung des Schlüsselwortes `volatile`

In der Regel werden Inline-Assemblerbefehle in die Optimierung des Compilers einbezogen, d.h. dieser kann sie verschieben oder ggf. bei der Optimierung entfallen lassen, falls sie nicht benötigt werden. Wollen Sie dieses verhindern und sicherstellen, dass der Inline-Assemblerbefehl in jedem Fall ausgeführt wird, so können Sie das Schlüsselwort volatile voranstellen.

Beispiel:

```
asm volatile ("l.nop");
```

Ausgewählte Befehle der OpenRISC 1200 CPU

Einige wichtige Befehle aus dem Basic Instruction Set der OpenRISC CPU finden Sie in untenstehender Tabelle. Das vorgestellte „l." weist darauf hin, dass es sich um sogenannte Class I Befehle handelt, die von jeder OpenRISC CPU implementiert werden müssen. Die allgemeinen Register (General Purpose Register) haben die Namen r0, r1, r2, usw. Diese sind anstelle der Platzhalter rD, rA und rB zu verwenden.

Befehl	Bedeutung
`l.nop`	„no operation" - der Prozessor macht einen Takt lang nichts
`l.add rD,rA,rB`	Register rA und rB werden addiert, Ergebnis wird in Zielregister rD gespeichert
`l.div rD,rA,rB`	rA wird durch rB geteilt, das Ergebnis in rD gespeichert
`l.lbz rD, I(rA)`	Lädt ein Byte von der Adresse rA + Konstante `I`. `I` kann 0 sein.
`l.sfeq rA,rB`	„Set Flag if equal": vergleicht rA mit rB und setzt Flag, falls beide gleich sind.
`l.bf N`	„Branch if flag": Bedingter Sprung - falls flag gesetzt ist, Sprung zu Adresse N.

Eine vollständige Referenz finden Sie unter
http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf im OpenRISC 1000 Architecture Manual.

Schritt 2: Implementierung der CPI-Messung für `l.nop`

Implementieren Sie nun innerhalb des orpmon einen neuen Befehl `hmcmd cpi_nop [iterations]` welcher den CPI-Wert des Befehls `l.nop` mit Hilfe einer Messung bestimmt.

Dieser Befehl - der optional mit einer gewünschten Anzahl an Iterationen zur Festlegung der Länge der Messung aufgerufen werden kann - soll innerhalb einer einfachen Schleife einen `l.nop` Befehl über Inline-Assembler aufrufen. Die Schleife soll so häufig durchlaufen werden, wie über den Parameter `iterations` vom Nutzer gewünscht worden ist. Falls keine `iterations` angegeben worden sind, so nehmen Sie als Default-Wert 10^7 Iterationen an.

Messen Sie die zum Durchlaufen der Schleife insgesamt benötigte Zeit und berechnen Sie daraus am Ende der Messung den CPI-Wert des `l.nop` Befehls. Geben Sie den berechneten Wert auf der Konsole aus.

Hinweise:

- Zur Messung der Laufzeit können Sie auf eine global verfügbare Variable `timestamp` zurückgreifen, welche TICKS_PER_SEC mal pro Sekunde um 1 erhöht wird. Diese Variable und die Konstante TICKS_PER_SEC können Sie nutzen, sofern Sie wie in der Vorgabe die Include-Dateien `support.h` und `common.h` einbinden.
- Sie können für Ihre Berechnungen davon ausgehen, dass eine einfache Schleife der Form
```
for(i=0;i<iterations;i++){
   /* … */
}
```
für einen Durchlauf (selbstverständlich ohne den Schleifenkörper!) sechs Takte benötigt.
- Sie können Ihr Programm zunächst mit Hilfe des Simulators implementieren und die Lauffähigkeit testen. Die Messergebnisse stimmen bei Nutzung des Simulators jedoch nicht mit der Wirklichkeit überein, d.h. die Messungen müssen Sie zwingend auf einem echten ORSoC Modul durchführen. Hierzu kopieren Sie das von Ihnen modifiziert orpmon mit Hilfe des Debuggers gdb auf das Modul und starten es. Das Vorgehen hierbei ist identisch zu dem, was Sie beim letzten Übungsblatt für die Ausführung des Linux-Kernels verwendet haben.

Schritt 3: Erweiterung der CPI-Messung

Erweitern Sie den orpmon um drei weitere Befehle mit entsprechenden CPI-Messungen:

`hmcmd cpi_add`	Messung des CPI-Wertes des Befehls `l.add`
`hmcmd cpi_div`	Messung des CPI-Wertes des Befehls `l.div`
`hmcmd cpi_lbz`	Messung des CPI-Wertes des Befehls `l.lbz`

Beobachten Sie Unterschiede? Falls ja: Wie können Sie die beobachteten Unterschiede erklären?

Schritt 4: Implementierung eines eigenen einfachen CPU-Benchmarks zur MIPS Bestimmung

Im letzten Schritt dieses Praktikums soll nun ein eigener einfacher CPU-Benchmark zur Bestimmung des MIPS-Wertes der CPU implementiert werden. Dieser soll über den Befehl `hmcmd mips` gestartet werden und nach Beendigung der Messung den gemessenen MIPS Wert ausgeben.

Gehen Sie zur Implementierung folgendermaßen vor:

- Definieren Sie eine Methode `mips_measurement(…)`, die nach Erkennung des Befehls hmcmd mips ausgeführt wird.

- Implementieren Sie (in Form von C-Code) innerhalb der Methode eine Schleife innerhalb derer 3-4 arithmetische Operationen Ihrer Wahl sowie mindestens eine if-Anweisung ausgeführt werden.
- Fügen Sie direkt vor und nach der Schleife mit Hilfe von Inline-Assembler zwei `l.nop` Operationen ein. Diese zwei Operationen nutzen wir später als Markierung, um die Schleife leichter im erzeugten Binärcode wiederfinden zu können.
- Messen Sie die zum Durchlaufen der Schleife benötigte Laufzeit.
- Um den MIPS-Wert berechnen zu können, benötigen Sie die Anzahl der ausgeführten CPU-Instruktionen. Da diese rein anhand des von Ihnen implementierten C-Codes nicht sichtbar ist, lassen Sie sich vom Compiler den erzeugten Maschinencode in disassemblierter Form ausgeben. Hierzu editieren Sie im Verzeichnis `orpmon/cmd` das `Makefile` und fügen dort folgende Zeile unterhalb der Zeile mit `all:` am Anfang ein:

  ```
  all: $(LIB)
       $(CC) $(CFLAGS) -S -o hmcmd.S hmcmd.c
  ```

  Dieses bewirkt, dass der Compiler bei jedem Durchlauf eine Datei `hmcmd.S` im gleichen Verzeichnis erzeugt, die den erzeugten Binärkode in disassemblierter, d.h. menschenlesbarer, Form enthält
- Übersetzen Sie Ihr Programm erneut und öffnen Sie danach die erstellte Datei `hmcmd.S`. Sie enthält den Binärcode bestehend aus den vom Compiler aus Ihrem C-Code generierten Operationen sowie den von Ihnen direkt per Inline-Assembler eingefügten Operationen. Die per Inline-Assembler eingefügten Operationen sind dabei folgendermaßen markiert:

  #APP
  # [Zeilennummer] „Datei“
  Assemblerinstruktionen die per Inline-Assembler eingefügt worden sind
  #NO_APP
- Suchen Sie nach der Schleife Ihres MIPS-Tests und zählen Sie, wie viele Instruktionen innerhalb der Schleife durchlaufen werden. Hierbei müssen Sie einen Sonderfall berücksichtigen: die OpenRISC CPU führt den **nach** einem bedingten Sprungbefehl vorkommenden Befehl **immer aus (d.h. unabhängig davon, ob die Bedingung gilt oder nicht!)**, er zählt also zu der Schleife ggf. dazu. Die Ursachen für dieses Verhalten werden wir im Kapitel Pipelining untersuchen.
- Berechnen Sie aus der Gesamtzahl der ausgeführten Befehle sowie der gemessenen Laufzeit den MIPS-Wert.

Für die Abnahme dieser praktischen Aufgabe brauchen Sie:

- das lauffähige, von Ihnen modifizierte orpmon-Programm,
- den zugehörigen, übersetzbaren Quellcode,
- die Messwerte der einzelnen CPI-Messungen für die angegebenen Befehle,
- den Inhalt der Datei hmcmd.S
- den von Ihnen ermittelten MIPS-Wert