Rechnertechnik **Übungszettel 9** **SS2014**

Vorbemerkung:

In dieser Aufgabe verwenden Sie einen kleinen Simulator, der das zeitliche Verhalten von Programmen auf einer idealisierten Maschine mit einem parametrisierbaren Instruction-Cache simuliert (Sie finden den Simulator auf Moodle im Paket Uebung9.zip: simulator.exe). Die zu simulierenden Programme werden dabei so vereinfacht, dass Sie lediglich eine Liste von Adressen bereitstellen müssen, die das Programm nacheinander an den Speicher legt um die Instruktionen zu laden. Der Simulator simuliert nun das Programm mit einem von Ihnen über die Kommando-Zeile übergebenen Satz an Parametern. Instruktionen selbst verbrauchen keine Zeit, sondern nur die Zeiten auf den Cache bzw. den Hauptspeicher werden simuliert. Das Nachladen einer Line geschieht im Simulator stets vollständig, erst dann kann der Prozessor weiterarbeiten. Als Ausgabe bekommen Sie die Anzahl der Zyklen ohne und mit der angegebenen Cache-Architektur, die Ausführungszeiten bei gegebener Taktfrequenz, die Miss-Rate des Programms sowie den Speedup, der durch den Cache verursacht wird. Die zu simulierenden Programme können Sie auch selbst schreiben, wenn Sie den Anweisungen in Anhang I folgen.

Die folgenden Eigenschaften bietet Ihnen der Simulator:

- Cachesize ist bis zu einer Größe von 32678 anzugeben, wobei dies stets eine 2er-Potenz sein muss
- Linesize ist bis zur Größe der Cachesize anzugeben, was aber wenig Sinn macht. Hier sollten Werte von 2-8 benutzt werden, allerdings wiederum nur 2er-Potenzen
- Assoziativität kann von 1 bis Cachesize/Linesize angegeben werden (also von Direct-Mapped-Cache bis zum vollassoziativen Cache), ebenfalls nur 2er-Potenzen
- Zugriffszeiten auf das DRAM können jeweils in ns angegeben werden (Vorladezeit und Zugriffszeit)
- Takt des Prozessors kann in MHz angegeben werden
- Assoziative Caches können die Taktfrequenz reduzieren (erhöhter Logikaufwand bei der Berechnung). Dies kann in % der Taktperiode angegeben werden
- Die Ersetzungstrategie kann gewählt werden (RoundRobin, Last Recently Used, Random)
- Der Simulator kann schrittweise betrieben werden. In jedem Schritt wird dann ein Dump des Cache-Inhaltes (nur die Adressen der Daten, die im Cache gespeichert wurden) ausgegeben.
- Für alle Parameter schauen Sie sich die Usage des Simulators an (Aufruf: simulator –h)

Aufgabe 1) Instruction-Cache – Simulation

Sie sollen nun verschiedene Beispielprogramme mit unterschiedlichen Assoziativitätsgraden und Replacement-Strategien simuliert werden. Die folgenden Eckdaten sollen für den Cache gelten:

- Die Cachegröße soll mit 64 Einträgen fest gegeben sein
- Die Linesize soll 4 Einträge betragen
- Die Speicherzugriffszeit nach dem Vorladen auf den Hauptspeicher ist mit X ns anzunehmen, wobei die Zeiten z.B. die folgenden sein können: DDR3-RAM: 1.25 ns, DDR2-RAM: 2.5 ns, DDR1-RAM: 5.0 ns. Hier dürfen Sie eine Größe auswählen.
- Die Vorladezeit des DRAM ist mit 30ns anzunehmen
- Die Speicherzugriffszeit auf den Cache-Speicher ist mit 1 Zyklus anzunehmen
- Für die Assoziativität sollen die Assoziativitätsgrade von 1,2,4,8 und 16 getestet werden
- Als Replacement-Strategien sollen die folgenden Strategien getestet werden:
    - **R**ound**R**obin, **L**ast**R**ecently**U**sed und **RAND**om

a) Führen Sie die Simulation für die drei Programme PRG1.DAT, PRG2.DAT und PRG3.DAT (finden Sie auf Moodle im Paket Uebung8.zip) für jeweils alle Assoziativitäten von 1 bis 8 jeweils für die drei Ersetzungsstrategien durch und beobachten Sie, wie sich die jeweilige Cachearchitektur bzgl. Laufzeit und Miss-Rate verhält. Versuchen Sie, die unterschiedlichen Ergebnisse anhand der Programmstruktur zu verstehen (die Programmstruktur können Sie in den Dateien PRG1.C, PRG2.C und PRG3.C anschauen – liegen ebenfalls auf Moodle im Paket Uebung8.zip; siehe hierzu Anhang 1 dieses Übungszettels).
Tragen Sie bitte in die 3 unten stehenden Tabellen die gemessenen Missraten ein.

prg1.dat: *Assoziativität*

Strategie	1	2	4	8	16
RR					
RAND					
LRU					

prg2.dat: *Assoziativität*

Strategie	1	2	4	8	16
RR					
RAND					
LRU					

prg3.dat: *Assoziativität*

Strategie	1	2	4	8	16
RR					
RAND					
LRU					

Wenn wir annehmen, dass unsere Programme nur so aussehen, wie die Beispielsprogramme, welchen Cache würden Sie warum einbauen?

b) Wenn Sie die ausgegebene Beschleunigung anschauen, so werden Sie feststellen, dass alle Programme mit jeder Cachearchitektur beschleunigt werden. Es kann jedoch Programme geben, die unter Verwendung eines Caches **langsamer** werden. Überlegen Sie sich, wie das Programm beschaffen sein muss, damit dieses Verhalten zu beobachten ist, schreiben Sie ein Programm(gemäß der Ausführungen in Anhang 1) mit mindestens 4 Adresszugriffen, das dieses Verhalten aufweist und simulieren Sie es.

Anhang 1

Da die Erstellung von Adresssequenzen eines Ablaufes eines Programmes manuell sehr umständlich ist, gibt es hierzu eine kleine Hilfe in Form einer Makro-Programmierung in C++. Sie müssen hierfür kein C++ verstehen, sondern es reichen die C- oder Java-Kenntnisse, die Sie aus PG1 mitbringen, sowie die hier aufgestellten Informationen, um eigene Programme zu schreiben und dann mit Ihrem Simulator zu testen. Die Angabe eines Programmes erfolgt dabei in einer Mischung aus PseudoCode und imperativem C-Code. Daraus kann dann mit Hilfe eines C++-Compilers eine ausführbare Datei erstellt werden, die die Adressliste als Textdatei erzeugt, die Sie im Cache-Simulator aus Aufgabe 1) laden können.

Ein solches Programm genügt folgender Syntax (hier die Grammatik):

PROGRAM ::= INCLUDE [FUNCS]* PROC

INCLUDE ::= "#include "makro.h""

FUNCS ::= *c-type* "FUNC(" *c-identifier* ")" BODY "ENDFUNC;"

PROC ::= "STARTPROC" BODY "ENDPROC;"

BODY ::= STM BODY | STM

STM ::= IFP_STM | IFC_STM | ORG_STM | LOOPFOR_STM | LOOPWHILE_STM |
GOTO_STM | INSTR_STM | RETURN_STM | *c-expressions*

IFP_STM ::= "IFP(" *integer-value* ")" BODY ["ELSEP" BODY] "ENDIFP;"

IFC_STM ::= "IFC(" *c-expression* ")" BODY ["ELSEC" BODY] "ENDIFC;"

ORG_STM ::= "ORG" *integer-value* ";"

LOOPFOR_STM ::= "LOOPFOR(" *integer-value* ")" BODY "ENDLOOPFOR;"

LOOPWHILE_STM ::= "LOOPWHILE(" *c-expression* ")" BODY "ENDLOOPWHILE;"

GOTO_STM ::= "GOTO" *c-label* ";"

INSTR_STM ::= "INSTRUCTION(" *c-string* "," *integer-value* ");"

RETURN_STM ::= "RETURN" *integer-value* ";"

Damit lässt sich die typische Struktur eines Programmes wiedergeben: Schleifen, Funktionsaufrufe, bedingte Anweisungen. Schleifen können entweder als Zählschleifen mit einer festen Anzahl an Durchläufen programmiert werden (LOOPFOR) oder als While-Schleifen, die an einer C-Expression entschieden werden (LOOPWHILE). Im letzten Fall muss im jeweiligen Rumpf der Schleife natürlich auch entsprechender C-Code stehen, damit die Schleife terminiert. Bedingungen können ebenfalls über C-Expressions entschieden werden (IFC). Zusätzlich gibt es noch die Möglichkeit bedingte Ausführungen von Wahrscheinlichkeiten von 0% bis 100% zu formulieren (IFP). Startadressen von Instruktionen werden mittels des ORG-Befehls gesetzt. Die Adresszugriffe des Programms schließlich werden mittels der INSTRUCTION-Anweisung gesetzt. Jeder Body sollte mindestens eine INSTRUCTION-Anweisung enthalten, ansonsten werden keine Adresszugriffe des Blockes simuliert. C-Expressions können überall im Code eingestreut werden, sie werden aber **nicht** bei den Adresszugriffen mit betrachtet, sondern dies geschieht **nur mittels der INSTRUCTION-Anweisung**.

ACHTUNG! Kommentare bitte nur in der Form `/* Kommentar */` verwenden!

Im Detail tun die Instruktionen das folgende:

- IFC : Bedingte Verzweigung mit C-Expression. Entspricht einer normalen If-Anweisung mit optionalem else-Teil (ELSEC)
- IFP: Wahrscheinlichkeits-bedingte If-Anweisung; hier wird gewürfelt, d.h. IFP(70) heißt, es gibt eine 70%ige Chance, dass der if-Teil ausgeführt wird und eine 30%ige Chance, dass der ELSEP-Teil ausgeführt wird, falls vorhanden
- LOOPFOR: Die Schleife wird exakt so oft durchlaufen, wie beim Parameter angegeben
- LOOPWHILE: Die Schleife wird solange durchlaufen, wie die C-Expression wahr ist; entspricht normaler while-Schleife
- GOTO: Verzweigt augenblicklich zum C-Label, verlässt dabei auch Schleifen. Sollte niemals als Aussprung aus einer Funktion genutzt werden.
- RETURN: entspricht dem bekannten return
- INSTRUCTION: Bildet sequentielle Blöcke von Instruktionen, die im Adressraum aufeinander folgen. Die Anzahl der Instruktionen wird mit übergeben. Jeder Instruktionsblock muss einen eindeutigen String als Kennung erhalten. Vorsicht! Ist der String nicht eindeutig, ist die erzeugte Adresssequenz nicht korrekt!

Wenn Sie ein Programm geschrieben haben, können Sie es compilieren und ausführen. Dazu öffnen Sie bitte eine Command-Shell (schnellster Weg: Filebrowser für Ihr Arbeitsverzeichnis aufmachen und oben beim Pfad „cmd“ eintippen). Stellen Sie sicher, dass die Datei „makros.h“ ebenfalls in diesem Verzeichnis liegt. Dann übersetzen mit einem C++-Compiler, z.B. für mingw mittels „g++ prg.c –o prg.exe“. Die entstandene ausführbare Datei dann starten und mit einem Argument versehen, das den Namen der Textdatei für den Simulator enthält, also z.B. „prg.exe prg.txt“. Voilá! In der nun entstandenen Datei prg.txt stehen die Daten, die der Simulator aus Aufgabe 1) benötigt.

Achten Sie bitte unbedingt darauf, dass Sie ELSEP/ELSEC und ENDIFP/ENDIFC immer richtig mit IFP/IFC kombinieren, da es ansonsten nicht funktionieren wird. Da die Implementierung Makros aus C++ nutzt, gibt es keine Möglichkeit einen Syntax-Check einzubauen!

`break` kann verwendet werden, **`continue`** hingegen **nicht**!

Nun noch ein etwas komplizierteres Beispiel zwecks Illustration:

```
#include "makros.h"


int FUNC(function1)          /* Verwendet C-Typ als Rückgabetyp */
                             /* weitere Parameter sind nicht möglich */
   ORG 1024;                 /* Startadresse der folgenden Instr. Setzen */
   INSTRUCTION("function1.1",4);
   IFP(50)
      INSTRUCTION("function1.1.1",5);
      RETURN 0;
   ENDIFP;
   RETURN 1;
ENDFUNC;


STARTPROC
   bool condition1=false;      /* Das sind C-Expressions */
   bool condition2=false;      /* Das sind C-Expressions */
   ORG 0;
   INSTRUCTION("1",12);        /* Block von 12 Instruktionen setzen */
   LOOPFOR(100)
      IFP(20)  /* Mit 20%iger Wahrscheinlichkeit wird if-Teil ausgeführt */
         condition2=true;      /* Das sind C-Expressions */
         IFP(70)
            int a=1;           /* Das sind C-Expressions */
            LOOPWHILE(a<42)    /* LOOPWHILE nutzt C-Expression */
               INSTRUCTION("while_loop.1",2);
               a = a*2;        /* Das sind C-Expressions */
            ENDLOOPWHILE;
            INSTRUCTION("1.1.1",3);
         ELSEP
            INSTRUCTION("1.1.2",6);
         ENDIFP;
      ELSEP
          INSTRUCTION("1.2.3",10);
          condition1 = true;   /* Das sind C-Expressions */
      ENDIFP;
      IFC(condition1||condition2) /* IFC nutzt C-Expression */
         INSTRUCTION("1.2",5);
      ENDIFC;
      int x = function1();     /* Das sind C-Expressions */
      IFP(1)        /* ziemlich kleine Wahrscheinlichkeit zum raushüpfen */
        GOTO Label1;
      ENDIFP;
   ENDLOOPFOR;
Label1:                        /* Das ist C: Das setzen eines Labels */
   INSTRUCTION("2",5);
ENDPROC;
```