timon.kelter [☺] tu-dortmund.de

Übung zur Vorlesung
Rechnerarchitektur
Sommersemester 2014

Übungsblatt 3

(10 Punkte)

Besprechung am Montag, 5. Mai 2014

3.1 MMX/SSE-Programmierung (5 Punkte)

Speziell für die Signalverarbeitung eignen sich sogenannte Multimedia Befehle. Prominente Beispiele sind der MMX und der SSE Befehlssatz von Intel, welche auf allen heute gängigen CPUs der x86-Reihe vorhanden sind. Der MMX-Befehlssatz benutzt die FPU Register um damit Ganzzahl-SIMD-Funktionen zu realisieren, außerdem enthält er einige für DSPs typische Befehle wie z.B. *multiply/accumulate*, während der SSE-Befehlssatz SIMD-Funktionen für Gleitkommazahlen anbietet.
In dieser Aufgabe wenden wir diese Befehle exemplarisch zur digitalen Bearbeitung von Audiosignalen an. Die Aufgabe ist es, mithilfe von MMX und SSE zwei Filter zu implementieren und diese auf ein gegebenes Audiosignal anzuwenden. Die Filter sind:

- Ein Tiefpaßfilter mit der Übertragungsfunktion

$$f(x) = \sum_{k \in \mathbb{Z}} g(x-k)h(k) \quad \text{mit } x \in \mathbb{Z}$$

  wobei $f(x)$ das gefilterte Signal darstellt, $g(x)$ das Eingangssignal und $h(k)$ den Filterkern bezeichnet. Üblicherweise benutzt man finite Filterkerne, d.h. $\exists N \in \mathbb{N} : \forall |k| > N : h(k) = 0$. Der in diese Aufgabe zu verwendende Filterkern ist:

$$\begin{aligned} h(3) &= 0.378 \\ h(2) &= 0.623 \\ h(1) &= 0.623 \\ h(0) &= 0.378 \end{aligned}$$

  Der Einfachheit halber wird der Rand vernachlässigt, d.h. für die ersten drei Samples ist $f(x) = g(x)$.

- Ein Echo-Filter, mit der Übertragungsfunktion

$$f(x) = g(x) + \sum_{k \in [1,N]} \frac{g(x-kD)}{2^k} \quad \text{mit } x \in \mathbb{Z}$$

  wobei N die Anzahl der Echos und D die Verzögerung zwischen zwei Echos ist.

In der Dokumentation der x86 Architektur finden Sie eine Auflistung aller MMX- und SSE-Befehle (Volume 1, Chapter 9 und 10). Sie werden für diese Aufgabe ausschließlich die MMX- und SSE-Befehle brauchen, alles andere wird aus Gründen der Lesbarkeit in C geschrieben. Die MMX- und SSE-Befehle arbeiten auf speziellen Registern, die im Falle von MMX `%mm0` bis `%mm7` heißen und 64 Bit breit sind, und im Falle von SSE `%xmm0` bis `%xmm7` heißen und 128 Bit breit sind. x86-Instruktionen sind ein 2-Adress Code, bei dem der erste Operand auch gleichzeitig das Ziel der Operation ist. Fast alle MMX- oder SSE-Instruktionen folgen diesem Muster, d.h. in den Operationen

```
<mmx_op> %mm0, %mm1
<sse_op> %xmm0, %xmm1
```

werden jeweils die Register `%mm0` bzw. `%xmm0` beschrieben. Diese Notation (die im x86-Manual verwendet wird) ist die *Intel Assembler Syntax*.
Zur Durchführung von Berechnungen können sie dem oben angegebenen Muster folgen. Vor den Berechnugen werden Sie allerdings die Operanden laden und danach das Ergebnis speichern wollen. Das funktioniert direkt über den Speicher, über die folgenden Anweisungen, die jeweils das 64-Bit (128-Bit) Wort von der Adresse in Register `%eax` (Register-indirekte Adressierung) in ein MMX- bzw. SSE-Register laden:

```
movq %mm0, 0(%eax)
movups %xmm0, 0(%eax)
```

Beide Operationen können auch zum Speichern von Register-Inhalten benutzt werden, dazu einfach die Operanden vertauschen.
Alle Instruktionsbeispiele bis hierhin haben die *Intel Assembler Syntax* benutzt. Neben dieser Syntax gibt es auch die *AT&T Assembler Syntax* die großteils identisch zur Intel Syntax ist, den Zieloperanden jedoch immer als letzten Operanden notiert. D.h. eine Operation der Form `reg1 = (reg1 op reg2)` wird in den beiden Formen wie folgt ausgedrückt:

```
Intel Syntax: <op> <reg1> <reg2>
AT&T  Syntax: <op> <reg2> <reg1>
```

Der Compiler, den Sie im folgenden verwenden sollen, verwendet die AT&T-Syntax, so dass Sie die Befehle in Intel-Syntax aus dem Manual entnehmen müssen, und in Ihrem Programm in AT&T-Syntax notieren müssen.
Das gegebene Rahmenprogramm ist in C geschrieben um den Code lesbar zu halten. Ihren Assemblercode werden Sie daher als *Inline-Assembler* schreiben, der direkt in die C-Datei geschrieben wird. Im frei verfügbaren GNU C Compiler (GCC) folgt Inline-Assembler (AT&T-Syntax!) dieser Form:

```
short input_operand_1[4] = ... ;
short input_operand_2[4] = ... ;
short output[4]; int i = ... ;
asm(
  "movq %[op1], %%mm0\n"
  "movq %[op2], %%mm1\n"
  "paddw %%mm1, %%mm0\n"
  "movq %%mm0, %[result]\n"
  : /* Ausgaben */
  [result] "=m"( output[i] )
  : /* Eingaben */
  [op1] "m"( input_operand_1[i] ),
  [op2] "m"( input_operand_2[i] )
  : /* Clobber */
  "mm0", "mm1" );
```

Der Assemblercode wird dabei mit Platzhaltern wie z.B. `(%[op1]` geschrieben, die erst in der Liste der Ausgaben und Eingaben aufgelöst werden. Dort werden die Platzhalter an C-Variablen gebunden. Normale Registernamen müssen zur Abgrenzung von den Platzhalten mit doppeltem `%` geschrieben werden. In diesem Beispiel bedeuten die Einträge der Eingabeliste, dass die Adresse der gegebenen Speicherinhalte (daher `"m"`) im Assemblercode anstelle des Platzhalters eingesetzt werden soll. Das `"="` gibt dabei an, dass der Operand beschrieben statt gelesen wird. Die *Clobber*-Liste muß alle Register enthalten, die von ihrem Code beschrieben werden, hier also `mmo` und `mm1`. Der obige Code lädt also die Inhalte der Eingabe-Arrays in MMX-Register `%mm0` und `%mm1` und berechnet dann parallel die Summe aller Array-Elemente, die mit dem letzten `movq` in das Ausgabe-Array gespeichert wird. Dieses Muster ist für die Bearbeitung dieser Aufgabe ausreichend, weitergehende Erklärungen finden Sie online.
Sie benötigen zu Bearbeitung der Aufgabe nach obigem Muster einen aktuellen GCC und daher ein Linux-System oder Cygwin unter Windows. Falls Ihnen keines von beiden zur Verfügung stehen sollte können Sie die Aufgabe auch an den Rechnerpools des Lehrstuhl 12 bearbeiten. Ein Rahmenprogramm zur Lösung der Aufgabe finden Sie auf den Webseiten der Übung. In dieses Rahmenprogramm müssen Sie nur noch an den mit `TODO` markierten Stellen Ihren Code zur Berechnung der Filterfunktion einfügen. Der Ordner enthält ein Makefile, mögliche Kommandos sind:

- **make all** - Kompiliert den Code einmal mit und einmal ohne SSE und testet ihn
- **make test** - Testet den compilierten Code und gibt die Zeitmessungen aus
- **make clean** - Löscht die generierten Dateien

Implementieren Sie mit diesen Vorgaben

a) Den Tiefpassfilter in ANSI-C (ohne MMX/SSE)

b) Den Tiefpassfilter mithilfe von SSE (hier werden 32-Bit Gleitkommazahlen multipliziert). Stellen Sie in diesem Fall die Koeffizienten direkt als `float`-Array dar, konvertieren Sie die Eingabe zu `float`s und rechnen Sie direkt auf dem `float`-Arrays.
Tipp: Wenn Sie die Summe über die 4 Arrayelemente mit dem `HADDPS`-Befehl berechnen, müssen Sie am Ende nur eine Zahl zurück nach `short` casten.

c) Den Echofilter mithilfe von MMX (hier werden nur 16-Bit Integerzahlen addiert).
Hinweis: Für diesen Teil müssen Sie den Schleifencode des Echofilters vorher entrollen.

3.2 VLIW- & EPIC-Prozessoren (5 Punkte)

In der Vorlesung wurden VLIW (Very Long Instruction Word) Architekturen vorgestellt. Diese Architekturen setzen eine starke Compilerunterstützung voraus, da die Parallelität in diesen Architekturen explizit im Instruktionswort codiert ist, d.h. der Compiler bestimmt welche Instruktionen parallel ausgeführt werden.
Im folgenden sollen die Schwierigkeiten, die sich hierbei für den Compiler ergeben untersucht werden. Dazu gehen wir von einer imaginären VLIW-Maschine aus, die in jedem Takt ein Paket aus

- 2 arithmetisch/logischen Befehlen und
- 2 Load/Store-Befehlen

verarbeiten kann. Die einzelnen Befehle aus den Befehlspaketen haben das Standard-MIPS-Format[1], ein Befehlspaket ist also $4 * 32$ Bit groß. Als Befehle sind hier nur die ”echten” MIPS-Befehle zulässig, die in der MARS-Hilfe als ”Basic Instructions” gekennzeichnet sind. Die Maschine hat einen Branch-Delay-Slot, der exakt ein Befehlspaket pro Sprung aufnimmt, und die Abarbeitung der Befehle erfolgt synchron, d.h. alle Befehle lesen gleichzeitig ihre Operanden und schreiben ihre Ergebnisse nach erfolgter Rechnung gleichzeitig zurück. Es ist zulässig, in einem Befehlspaket einen Sprung mit 3 weiteren Operationen zusammenzufassen. Alle Befehle benötigen genau einen Takt Bearbeitungszeit. Notieren Sie Assemblerbefehle für diese VLIW-Maschine mit der Syntax

```
<ALU-Befehl 1> ; <ALU-Befehl 2> ; <Load/Store-Befehl 1> ;  <Load/Store-Befehl 2>
```

a) Auf der Maschine wird das folgende Stück C-Code ausgeführt. Sie können voraussetzen, daß die Addition in dem verwendeten Zahlenbereich assoziativ ist.

```
int p, q, r, s;

...

p = p + q + r + s;
```

Geben Sie zwei verschiedene VLIW-Instruktionsfolgen an, die diese C-Anweisung auf der Maschine ausführen. Die Variablen `p`, `q`, `r` und `s` seien schon in Registern `$p`, `$q`, `$r` und `$s` geladen. Was ist die günstigste Umsetzung dieser Anweisung?

[1] Eine MIPS-Befehlsreferenz finden Sie im MARS-Simulator (http://courses.missouristate.edu/KenVollmar/MARS/)

b) Übersetzen Sie das folgende C-Fragment möglichst optimal in VLIW-Instruktionen. Die Semantik der Anweisungen muß erhalten bleiben, insbesondere dürfen Instruktionen erst dann ausgeführt werden, wenn sie auch im C-Fragment ausgeführt worden wären (keine spekulative Ausführung). Sie können dabei annehmen, daß die Basisadressen der verwendeten Integer-Arrays hinreichend klein sind um als Offsets der Load/Stores verwendet werden zu können (also: `LW $2, X($1)`). Verwenden Sie bei der Übersetzung symbolische Bezeichner für MIPS-Register wie z.B. `$a` für Variable `a`.

```
int a, b, c, X[N], Y[N], Z[N];

...

int i = 0;
while ( i < 100 ) {
  a    = X[i];
  b    = a / r;
  a    = a + r;
  c    = b + r;
  b    = r - b;
  X[i] = a;
  Y[i] = b;
  Z[i] = c;
  i    = i + 1;
}
```

Wie viele VLIW-Befehle benötigen Sie für eine einzelne Iteration (inklusive der Auswertung der Abbruchbedingung) der Schleife?

c) Entrollen Sie die Schleife aus Ihrem Ergebnis aus b) indem Sie den Körper der Schleife duplizieren. Finden Sie eine möglichst optimale Zuordnung der einzelnen Befehle zu VLIW-Instruktionen, indem Sie Abhängigkeiten zwischen den beiden abgerollten Schleifeniterationen durch passende Umbenennung der verwendeten Register eliminieren. Geben Sie das entstehende Programm in der obigen VLIW-Assemblersyntax an. Wie viele VLIW-Befehle benötigen Sie nun für eine einzelne Iteration der Originalschleife (Laufzeit einer Iteration der neuen Schleife geteilt durch zwei)?

Hinweise:

- Das Umbenennen von Registern sollte nach dem Schema erfolgen:

```
<Lese-/Schreibzugriffe auf $x>      <Lese-/Schreibzugriffe auf $x>
<Schreibzugriff auf $x>         →   <Schreibzugriff auf $x2>
<Lese-/Schreibzugriffe auf $x>      <Lese-/Schreibzugriffe auf $x2>
```

- Durch das Umbenennen von Registern lassen sich nur Antidaten- und Ausgabeabhängigkeiten eliminieren.

Allgemeine Hinweise: Die Übungstermine und weitere Informationen finden Sie unter http://ls12-www.cs.tu-dortmund.de/daes/de/lehre/lehrveranstaltungen/sommersemester-2014/rechnerarchitektur.html. Die Übungszettel werden zum Semesterbeginn online gestellt und sollen eigenständig bis zum jeweiligen Stichtag gelöst werden. Die Lösungen werden in den Gruppen besprochen. Auf Wunsch kann für diese Veranstaltung ein Übungsschein ausgestellt werden. Hierzu müssen die selbst erstellten Lösungen jeweils vor der Besprechung der Aufgaben beim Übungsgruppenleiter abgegeben werden. Dabei müssen 45% der Gesamtpunkte bei den Übungszetteln erreicht und eigene Lösungen in der Übungsgruppe präsentiert werden. Für die Teilnahme an der Klausur nach BPO 2013 / der Fachprüfung nach DPO 2001 ist der Übungsschein *nicht* erforderlich.