UNIVERSITÄT SALZBURG

Rechnerarchitektur

SS 2018

Übungszettel 1

1. In einer 32-Bit-Stackmaschine laufen alle Operationen über den Stack ab. Nur `push` (Datum auf den Stack, 3 Taktzyklen) und `pop` (Datum vom Stack, 3 Taktzyklen) greifen auf den Speicher zu. Weitere Kenndaten der Rechnerarchitektur sind:

   - Der Operationscode ist 1 Byte lang.
   - Alle Speicheradressen haben eine Länge von 2 Bytes.
   - Alle Operanden sind 4 Bytes lang.
   - Die Variablen A und B stehen vom Beginn an im Speicher.

   (a) Schreiben Sie ein Programmstück für die Stackmaschine, das $A^2 - B^2$ berechnet und das Ergebnis in die Variable C speichert. Es stehen die Befehlscodes `add` (2 Taktzyklen), `sub` (2 Taktzyklen) und mult (8 Taktzyklen) zur Verfügung. Sie sollten dabei mit *einer* Multiplikation auskommen.

   (b) Wieviele Befehle und Taktzyklen benötigt Ihr Programm? Wieviele Bytes (Instruktionen und Daten) werden beim Programmablauf vom Speicher in die Maschine transferiert?

   (c) Es stehen nun die zusätzlichen Befehle `dup` (dupliziert das oberste Stackelement, 1 Taktzyklus) und `exc` (vertauscht die beiden obersten Stackelemente, 1 Taktzyklus) zur Verfügung.

   (d) Ermitteln Sie die Parameter Ihres neuen Programms (analog zu b).

2. Schreiben Sie ein Programmstück für eine Stackmaschine, das die Fakultät einer ganzen Zahl $n \geq 0$ berechnet. Die Zahl steht an der Speicherstelle n und das Ergebnis $n!$ soll an der Speicherstelle res abgelegt werden. Der Wert von n muss nicht im Speicher erhalten bleiben. Wir nennen das oberste Stackelement top und den wert darin *topvalue*.

   Sie haben folgende Befehle zur Verfügung (die Zahlen $+1, 0, -1$ beziehen sich auf die Änderung der Anzahl der Stackelemnte):

   - `push x` (Datum von Speicherstelle x auf top, +1)
   - `pop x` (top value an Speicherstelle x, -1)
   - `inc` (erhöht top value um eins, 0)
   - `dec` (verringert top value um eins, 0)
   - `mult` (top value = top value * (top - 1) value, -1)
   - `bz label` (springt zu label, wenn top value gleich null ist, 0)
   - `jmp label` (springt zu label, 0).

3. Für zwei vorzeichenlose Zahlen $Z1 = A \cdot 2^{16} + B$, $Z2 = C \cdot 2^{16} + D$ soll der mathematische Ausdruck $E = A \cdot C \cdot 2^{32} + (B \cdot C + A \cdot D) \cdot 2^{16} + B \cdot D$

   (a) mit einer Akku-Maschine: LOAD x, STORE x, ADD x, MULT x, SHL i

   (b) mit einer Speicher-Maschine: ADD x, y, z, MULT x, y, z, SHL x, y, i

   unter ausschließlicher Verwendung der jeweils angegebenen Opcodes berechnet werden. i ist ein Immediate-Wert (z.B. 16). Wir nehmen an, dass die Speicherstellen (A – E) und der Akku alle Ergebnisse aufnehmen können. Zwischenergebnisse können in zusätzlichen Speicherstellen ab F gespeichert werden. Sie sollten die Anzahl der Stores jedoch minimal halten. Geben Sie jeweils den Assembler Code zur Berechnung obigen Ausdruckes an. Vergleichen Sie die Anzahl der Instruktionen und der notwendigen Speicherzugriffe.

UNIVERSITÄT SALZBURG

Rechnerarchitektur

SS 18

Übungszettel 2

4. Zum Vergleich der Effizienz von Speicherzugriffen bei vier verschiedenen Architekturen betrachten wir folgende Maschinen:

   (a) Stack-Maschine (0-Adressmaschine)

   (b) Akku-Maschine (1-Adressmaschine)

   (c) Speicher-Maschine (3-Adressmaschine)

   (d) Load-Store-Maschine (Register-Maschine) mit
   16 Registern, Register-Befehle haben 3 Operanden, die Registerkennung ist 4 Bit lang

   Zum Vergleich der Speicherzugriffe treffen wir für alle Architekturen folgende Annahmen:

   - Der Operationscode ist immer 1 Byte lang.
   - Alle Speicheradressen haben eine Länge von 2 Bytes.
   - Alle Operanden sind 4 Bytes lang.
   - Die Länge einer Instruktion ist immer eine ganzzahlige Byteanzahl.
   - Die Variablen A, B, und C stehen vom Beginn an im Speicher.
   - Mit einem Speicherzugriff können maximal 4 Bytes übertragen werden.

   Schreiben Sie für jede der vier obigen Architekturen Assembler–Code für die Anweisung `A = B + C` . Verwenden Sie dazu die (syntaktisch der entsprechenden Architektur angepassten) Assemblerbefehle `load, store, add, push, pop, move` und ermitteln Sie die Anzahl der Befehlsbytes und die Anzahl der Datenbytes für jede Code-Sequenz. Welche Architektur hat den effizientesten Code und welche die wenigsten Speicherzugriffe (= Befehle und Daten)?

5. Ein Audiosignal mit Dauer τ soll digital abgetastet und analysiert werden. Das *Nyquist*-Kriterium besagt, dass für die exakte Rekonstruktion eines Signals mit Bandbreite B die Abtastrate mindestens $2B$ betragen muss.

   In der Digitalen Signalverarbeitung wird zur Signalanalyse oft des Fourierspektrum herangezogen. Die Komplexität der Diskreten Fouriertransformation (DFT) beträgt $O(n^2)$, d.h. es sind grob n^2 Multiplikationen, wobei n die Anzahl der Abtastpunkte bezeichnet, notwendig. Die Komplexität der schnellen Fouriertransformation (FFT) beträgt lediglich $O(n \log n)$.

   Berechnen Sie die (ungefähren) Rechenzeiten für DFT und FFT auf einem Prozessor mit der Taktrate f_T. Eine Floating-Point-Operation benötigt k Taktzyklen. Zum Vergleich werden lediglich die Anzahl der Floating-Point-Multiplikationen berücksichtigt. Spezialisieren Sie das Ergebnis mit $\tau = 1\text{s}$, $B = 20\text{KHz}$, $f_T = 1\text{GHz}$ und $k = 4$.

6. Gegeben ist eine CPI-Rate eines Prozessors von 2.5. 50% der Befehle verursachen einen Speicherzugriff und 5% sind FPU-Befehle. Folgende Verbesserungen des Prozessors sollen geprüft werden:

   (a) Beseitigung von *Structural Hazards*: Bei $\frac{1}{4}$ der Speicherzugriffe wird 1 Taktzyklus gespart.

   (b) Erhöhung der Taktfrequenz um 10%. Dadurch benötigen aber FPU-Befehle 2 Taktzyklen mehr.

(c) FPU-Befehle um 2 Taktzyklen kürzen.

Welche Maßnahme bringt die größte Verbesserung?

7. Die CPI-Rate eines Prozessors sei 2.4. 15% der Befehle sind Sprungbefehle. Es werden zwei Verbesserungen ins Auge gefasst, von denen aber nur eine realisiert wird.

   (a) Jeder Sprungbefehl wird um zwei Taktzyklen verringert.

   (b) Die Taktfrequenz des Prozessors wird um $\frac{3}{7}$ erhöht, wodurch FPU–Befehle um einen Taktzyklus verlängert werden müssen.

   Bei welchem Anteil von FPU–Befehlen sind beide Verbesserungen gleich?

UV

Rechnerarchitektur

SS 2018

Übungszettel 3

8. Lesen Sie die Einführung in die DLX-Programmierung und starten Sie `openDLX` (im Ordner `apps/dlx_apps` sind einige Testprogramme) und beantworten Sie folgende Fragen:

   (a) Welche Bedeutung haben `.data` und `.text`?

   (b) Was ist der Unterschied zwischen `add`, `addu`, `addi`, `addui`, `addf` und `addd`?

   (c) Was sind Pointer und wozu braucht man sie?

   (d) Was macht der Befehl `jal`?

9. Schreiben Sie ein DLX-Programm, das die abgerundete Wurzel b einer natürlichen Zahl a mittels Intervallhalbierung berechnet, entsprechend folgendem Java-Codeausschnitt:

```
int a = 81, b;
int u = 0, o = a, m = o/2;
while (m != u)
{ if (m*m > a) o = m; else u = m;
m = (o+u)/2;
}
b = m;
```

10. Schreiben Sie ein DLX-Programm, das das ggT (den größten gemeinsamen Teiler) c zweier positiver Zahlen a, b mit dem euklidischen Algorithmus berechnet. Dabei wird der Divisionsrest r von a/b (d.h. $a = nb + r, r < b$) berechnet und dann auf gleiche Weise mit b und r fortgefahren, bis der Rest $r = 0$ ist. Halten Sie sich an folgenden Java-Code:

```
int a = 15, b = 6, c;
while (b > 0)
{
while (a >= b) { a = a - b; }  // a = Divisionsrest
int tmp = a; a = b; b = tmp;   // a und b vertauschen
}
c = a;
```

11. Schreiben Sie ein DLX-Programm, das die n-te Fibonacci-Zahl berechnet. Halten Sie sich am besten an folgendes Gerüst und folgenden Algorithmus:

```
.data
n:      .word 10
f:      .word 0

.text
main:         ; Programm folgt hier
```

```
a0 = 0;
a1 = 1;
n = n - 1;
while (n != 0) {
a2 = a0 + a1;
a0 = a1;
a1 = a2;
n = n - 1;
}
f = a1;
```

UNIVERSITÄT SALZBURG

Rechnerarchitektur

SS 18

Übungszettel 4

12. Schreiben Sie ein DLX–Unterprogramm, das die Korrektheit eines Zeichens im 7–Bit ASCII Code mit *even parity* überprüft. Überlegen Sie zuerst, wie einzelne Bits in einem Register (mit logischen Operationen) geprüft werden können.

    a. Geben Sie das Unterprogramm mit folgenden Übergabeparametern an: `R1` enthält das zu überprüfende Zeichen. Das Ergebnis wird in `R3` übergeben. Das Unterprogramm soll nach dem *Callee–Save*–Prinzip gestaltet sein.

    b. Geben Sie ein Codefragment an, das es dem Unterprogramm auch erlaubt Zeichen im 7–Bit ASCII Code mit *odd parity* zu prüfen. Dazu wird in `R2` 0 für ein Zeichen mit even parity und 1 für odd parity übergeben. Kennzeichnen Sie die Stelle in a., wo dieses eingefügt werden soll.

    c. Testen Sie das Unterprogramm im DLX–Simulator.

13. Ein A/D–Wandler liefert einen digitalisierten Spannungswert in Form eines 1–aus–16 Codes, der in das Register `R1` einer DLX–Maschine geladen wurde.

    a. Schreiben Sie ein DLX-Unterprogramm (mit *Callee–Save*), das die Korrektheit des Digitalwerts prüft und geben Sie das Resultat der Überprüfung in `R2` zurück.

    b. Was müsste in ihrem Programm a. geändert werden, wenn der A/D–Wandler einen 1–aus–32 Code liefert?

    c. Testen Sie das Unterprogramm im DLX–Simulator.

14. Schreiben Sie ein DLX-Unterprogramm (ohne *register save*), das die Anzahl des Auftretens eines Bytes in einem Speicherbereich ermittelt. Der Wert des Bytes wird in `R1` übergeben. Der zu überprüfende Speicherbereich wird durch die Anfangsadresse in `R2` und die Endadresse in `R3` definiert. Geben Sie die Anzahl des Auftretens in `R1` zurück.

UNIVERSITÄT SALZBURG

Rechnerarchitektur

SS 2018

Übungszettel 5

15. Ein data hazard, der einen stall von 2 Zyklen verursacht, soll durch *forwarding* umgangen werden. Dazu wird die CPU mit zusätzlicher Hardware ausgestattet, wodurch aber die Taktrate um 5% verringert werden muss. Der data hazard tritt bei 10% der Befehle auf, kann aber nicht immer verhindert werden. Die durchschnittliche CPI-Rate vor der Verbesserung ist 1.8.

    (a) Es können nur 20% der data hazards verhindert werden. Berechnen Sie die neue CPI-Rate und den Speedup (ein Bruch genügt).

    (b) Sei h der Anteil der data hazards, die verhindert werden können. Wie groß muss h mindestens sein, damit eine Verbesserung erzielt wird.

16. Gegeben ist folgender Programmabschnitt für die DLX-Pipeline:

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

    Wie viele Taktzyklen werden für die Ausführung dieses Programmabschnitts benötigt

    (a) in der klassischen Pipeline (d. h. ohne Forwarding und Takthalbierung)?

    (b) wenn Takthalbierung möglich ist?

    (c) wenn in der Pipeline-Unit Forwarding-Hardware vorhanden ist?

17. Die unten angegebene Befehlssequenz wird auf einer DLX mit Pipeline ausgeführt, wobei die Pipeline folgende Eigenschaften aufweist:

    i. Speicherzugriffe auf Daten und Instruktionen sind gleichzeitig möglich (d.h. IF und MEM im gleichen Takt).

    ii. Forwarding ist nicht vorhanden.

    iii. Der Takt ist in Schreib– und Lesephase für Registerinhalte geteilt (d.h. Register schreiben in Phase WB und Register lesen in Phase ID ist gleichzeitig möglich).

```
LW  R1, b
LW  R2, c
ADD R2, R1, R2
LW  R1, e
SW   a, R2
LW  R2, f
SUB R1, R1, R2
SW   d, R1
TRAP 0x0
```

    (a) Welcher Hazard–Typ kann wegen Eigenschaft i. vermieden werden? Begründen Sie Ihre Aussage.

    (b) Wann steht der Wert des Zielregisters eines ALU– oder Loadbefehls frühestens zur Verfügung?

    (c) Beschreiben Sie den Ablauf der gegebenen Sequenz durch ein Taktzyklusdiagramm. Geben Sie die CPI–Rate der Sequenz an.

    (d) Beantworten sie die obigen Punkte b. und c. unter der Annahme, daß nun Forwarding eingeschaltet ist. Dies bedeutet bei der DLX in diesem Fall, daß Forwarding für ALU–, Load– und Store–Befehle möglich ist (und daß bei Store–Befehlen der zu speichernde Wert erst direkt nach der MEM–Phase der Store–Instruktion geliefert werden kann). Markieren Sie im Taktzyklusdiagramm, wo Forwarding stattfindet.

UNIVERSITÄT SALZBURG

Rechnerarchitektur

SS 18

Übungszettel 6

18. Geben Sie ein Taktzyklendiagramm und die Anzahl der Zyklen und Stalls für das gegebene Codestück unter folgenden Bedingungen an:

```
ADDI    R1, R0, #4
LW      R3, 0(R2)
MULT    R3, R3, R3
SUB     R4, R1, R5
AND     R5, R5, R8
SW      0(R2), R5
OR      R4, R3, R8
```

   (a) Die DLX Pipeline arbeitet mit Takthalbierung in den Stufen `IF`, `ID`, `MEM` und `WB`.

   (b) Versuchen Sie, die Anzahl der Stalls für Fall a) durch Codeoptimierung (Umordnen der Befehle) zu minimieren.

19. Folgendes Programm zum Kopieren eines Speicherbereichs läuft durch eine einfache DLX Pipeline (ohne Verbesserungen).

```
      .data
len:  .word 100          ;length of data array
a:    .space 100         ;source
b:    .space 100         ;destination
      .text
main: addi r1, r0, a     ;source address
      addi r2, r0, b     ;destination address
      lw   r3, len       ;load length
loop: lb   r4, 0(r1)
      addi r1, r1, #1    ;increment source pointer
      sb   0(r2), r4
      addi r2, r2, #1    ;increment destination pointer
      subi r3, r3, #1    ;decrement counter
      bnez r3, loop
end:  trap 0
```

   (a) Geben Sie Art und Anzahl der Stalls an.

   (b) Optimieren Sie den Code so, dass die Anzahl der Stalls minimiert wird.

20. Schreiben Sie ein DLX-Unterprogramm, das die Fakultät einer ganzen Zahl $n \geq 0$ berechnet. Die Zahl n wird in `R1` übergeben und das Ergebnis $n!$ in `R2` zurückgegeben.

   (a) Schreiben Sie das Unterprogramm in möglichst kompakter Form.

   (b) Durch eine Verbesserung in der DLX Pipeline steht eine Sprungadresse nicht mehr nach MEM, sondern schon nach ID zur Verfügung, sodass ein Jump/Branch nur mehr einen Stall erzeugt. Eine weitere Verbesserung ist ein *Delay Slot*. Hier führt die Pipeline den Befehl nach dem Jump/Branch **immer** aus und führt eine eventuelle Verzweigung erst nach dem Befehl im delay slot aus. Dies bedeutet aber auch, dass im delay slot immer ein korrekter Befehl stehen muss.

   Schreiben Sie Ihr Programm so um, dass es auf einer Pipeline mit delay slot korrekt läuft und die Rechenzeit (im Vergleich zu a) verkürzt.