

RA FS 21 Serie 3

Abdellhak Lemkenter, Adrian Wächli, Sepsch Sameni

Die dritte Serie ist bis Dienstag, den 20. April 2021 um 15:00 Uhr zu lösen und auf ILIAS hochzuladen. Für Fragen steht ein Piazza jederzeit ein Forum zur Verfügung.
Allfällige unklare Probleme sind uns so früh wie möglich mitzuteilen, wir werden gerne helfen.
Viel Spass!

Theorieteil

Gesamtpunktzahl: 11 Punkte

1 Performance Berechnungen (3 Punkte)

Nehmen Sie an, eine CPU sei mit 500Mhz getaktet. Nehmen Sie weiter an, dass besagte CPU die folgenden Operationen (mit angegebener Zeitdauer) durchführt:

ALU 4nscc

LOAD 8nscc

STORE 12nscc

Branch 6nscc.

- (a) Wir nehmen zu Beginn an, dass alle Operationen gleich häufig durchgeführt werden.
- Wieviel schneller/langsamer ist eine Maschine, die für die LOAD Instruktion 6 Taktzyklen braucht?
 - Wieviel schneller ist eine CPU bei der die STORE doppelt so schnell arbeitet?
- (b) Die Operationen werden mit folgender Häufigkeiten ausgeführt: ALU 50%, LOAD 15%, STORE 25%, BRANCH 10%. Berechnen Sie den idealen CPI (ohne zu berücksichtigen, dass der Zugriff auf den Hauptspeicher Stalls erfordert).

2 Stackverwendung bei Subroutinen (2 Punkte)

Geben Sie zwei mögliche Gründe an, wieso man den Stack bei Assembler-Subroutinen braucht?

3 ALU & SLT (2 Punkte)

- Was passiert beim `slt` Befehl in der ALU?
- Wie unterstützt die ALU den `slt` Befehl?

4 `loadi` (2 Punkte)

Geben Sie an, wie das Laden einer (32 Bit langen) Konstanten in ein MIPS-Register mit dem MIPS-Befehlssatz umgesetzt werden kann.

5 ALU: OPCODEs (2 Punkte)

Beschreiben Sie, wie die Ansteuerbits der ALU für die folgenden Befehle gesetzt werden müssen:

`and` `or` `add` `subtract` `slt` `nor`

Erläutern Sie weiter den Zusammenhang zwischen diesen Bits/Befehlen und den einzelnen Elementen der ALU.

Programmierteil

Ihre Aufgabe ist es, das gegebene Programmgerüst wie folgt zu vervollständigen:

- (a) Laden Sie die zur Serie 3 gehörigen Dateien von IlIAS herunter und studieren diese aufmerksam. Versuchen Sie zu verstehen, was die bereits vorhandenen Teile bedeuten.¹
- (b) Tragen Sie Ihren Namen sowie den Namen einer allfälligen Übungspartnerin oder eines allfälligen Übungspartners an den vorgesehenen Stelle in den Dateien `compile.c`, `mips.c` und `memory.c` ein.
- (c) Vervollständigen Sie die Funktion `main` in `compile.c`, so dass auf der Kommandozeile der zu kompilierende Ausdruck und der Dateiname für die Datei, in der das kompilierte Programm gespeichert wird, als Parameter übergeben werden können. Bei falscher Eingabe soll ein Hinweis zur Benützung, wie z.B.

`usage: <Befehlsname> expression filename`

ausgegeben werden (wobei anstelle von `<Befehlsname>` der tatsächliche Programmname steht, auch wenn das Programm umbenannt wird). Bei korrekter Eingabe, also z.B.

`./compile '3*(45+6)*12' test.mips`

soll

`Input: (3*(45+6))*12`

`Postfix: 3 45 6 + * 12 +`

`MIPS binary saved to test.mips`

ausgegeben werden.

Hinweis:

- Die Hauptarbeit übernimmt die bereits implementierte Funktion `void compiler(char* exp, char *filename)` (in `compiler.c`). Sie müssen nur sicherstellen, dass `main` mit der richtigen Anzahl Argumente aufgerufen und diese korrekt an `compiler` weitergegeben werden
 - http://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html
- (d) Vervollständigen Sie die Funktion `loadFile(char*)` in `memory.c`, so dass eine erzeugte MIPS-Binary-Datei in den Speicher gelesen und ausgeführt werden kann.
- Sie können davon ausgehen, dass die MIPS-Binary-Datei genau dem erwarteten Ablauf im Speicher entspricht, d.h. die einzelnen Wörter sind in Big Endian Byte-Reihenfolge ohne Unterbruch in der Datei abgelegt.

Hinweis:

- Betrachten Sie `void store(word w)` in `compiler.c`
- Erzeugen Sie (z.B. mittels `./compile 1+1 test.mips`) einige Dateien und betrachten Sie diese mit einem Hex-Editor, um das Dateiformat besser zu verstehen.
- http://publications.gbdirect.co.uk/c_book/chapter9/input_and_output.html und folgende Abschnitte

- (e) Vervollständigen Sie die Funktion `error` in `mips.c`, so dass die Fehlermeldungen korrekt formatiert ausgegeben werden können. Betrachten Sie dazu auch das Makro `ERROR` in `mips.h`. Die Fehlermeldung soll jeweils ausgehen in welcher Funktion, auf welcher Linie in welcher Datei der Fehler aufgetreten ist sowie eine detaillierte Fehlermeldung ausgeben. Ein Aufruf

`ERROR("Unknown opcode: %x", instruction->i_opcode);`

in `mips.c`, Funktion `undefinedOperation` auf Zeile 166 soll z.B. folgende Ausgabe ergeben

`undefinedOperation in mips.c, line 166: Unknown opcode 2b`

¹`compiler.c` und `compiler.h` müssen nicht ausführlich studiert werden, wichtige Stellen werden unten erwähnt, bei Interesse für die restlichen Teile findet sich jedoch ein wenig Theorie im Anhang.

Memory.c we already have a pointer to Memory (defaultMemoryData)

(angenommen, `instruction->i_opcode == 0x2b`).

Nach der Ausgabe des Fehlers soll das Programm mittels `exit(EXIT_FAILURE)` beendet werden.

Hinweis:

- <http://linux.die.net/man/3/vfprintf>
- http://linux.die.net/man/3/va_start

- (f) Stellen Sie sicher, dass Ihre Implementation ohne Fehler und Warnungen kompilierbar ist, überprüfen Sie dies mit `make`
- Dies ist eine notwendige Voraussetzung, damit der Programmteail als erfüllt gilt.
- (g) Stellen Sie sicher, dass Ihre Implementation die gegebenen und Ihre eigenen Tests ohne Fehler und Warnungen absolviert, überprüfen Sie dies mit `make test`
- Dies ist eine notwendige Voraussetzung, damit der Programmteail als erfüllt gilt.

- (h) Erstellen Sie aus Ihrer Lösung eine Zip-Datei namens `<achname>.zip` (wobei `<achname>` natürlich durch Ihren Namenen zu ersetzen ist).

- (i) Geben Sie die Datei elektronisch durch Hochladen in ILIAS ab.

A Theorie

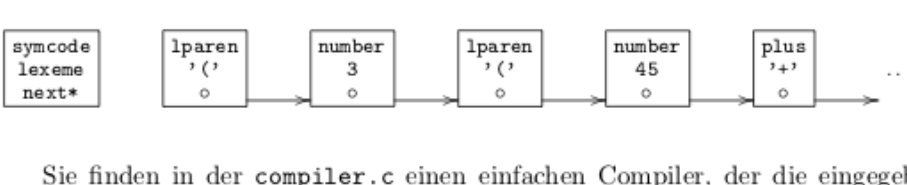
In `compiler.c` und `compiler.h` finden Sie die Implementation eines einfachen Compilers für arithmetische Ausdrücke. Unsere „Programmiersprache“ ist in den folgenden erweiterten Backus-Naur-Form (EBNF) gegeben:

`digit` = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
`number` = `digit`, (`digit`) ;
`factor` = `number` | "+" | "-", `expression`, ")",
`term` = `factor`, { ("*" | "/"), `factor` } ;
`expression` = `term`, { ("+" | "-"), `term` } ;

Ein mögliches „Programm“ wäre also z.B.

`(3*(45+6))*12`

Diese Eingabe wird von einem lexikalischen Scanner (auch *Lexer* genannt) zuerst in einzelne Bestandteile (in diesem Fall Symbole und Zahlen), sogenannte Tokens, aufgeteilt. Diese werden mit Zusatzinformationen versehen und als Liste abgespeichert, d.h. der oben stehende Ausdruck wird beispielsweise zu



Sie finden in der `compiler.c` einen einfachen Compiler, der die eingegebenen Ausdrücke in Postfix-Notation umwandelt, d.h. der oben stehende Ausdruck wird zu

`3 45 6 + * 12 +`

Ausdrücke in Postfixnotation lassen sich einfach mit einer Stackmaschine abarbeiten. Hierzu wird jedes Argument auf einen Stack gelegt und Operationen werden auf den obersten Stackelementen angewendet. Hierzu werden die Stackelemente in Register des Prozessors geladen, die Operation angewendet und schliesslich das Resultat wieder zurück auf den Stack gelegt.

Ausdrücke in Postfixnotation lassen sich also einfach in Assemblercode umwandeln, das oben stehende Programmteail wird z.B. (in Pseudo-Assembler) zu

```
push 3 // Lage 3 auf den Stack
push 45 // Lage 45 auf den Stack
push 6 // Lage 6 auf den Stack
pop B // Hole obersten Wert auf Stack nach Register A
add A, A, B // Addiere die Register A und B, speichere das Resultat in A
push A // Lage den Wert von Register A auf den Stack
pop B // Hole obersten Wert auf Stack nach Register A
mult A, A, B // Multipliziere Register A und B, speichere das Resultat in A
push A // Lage den Wert von Register A auf den Stack
push 12 // Lage 12 auf den Stack
pop B // Hole obersten Wert auf Stack nach Register A
add A, B // Addiere den Wert auf Stack nach Register B
add A, A, B // Addiere die Register A und B, speichere das Resultat in A
push A // Lage den Wert von Register A auf den Stack
stop // Beende das Programm
```

Das Enderesultat der Berechnung wird dabei wieder im Stack abgelegt.

Nehmen Sie an, eine CPU sei mit 500Mhz getaktet. Nehmen Sie weiter an, dass besagte CPU die folgenden Operationen (mit angegebener Zeitdauer) durchführt:

ALU 4nscc

LOAD 8nscc

STORE 12nscc

Branch 6nscc.

- (a) Wir nehmen zu Beginn an, dass alle Operationen gleich häufig durchgeführt werden.
- Wieviel schneller/langsamer ist eine Maschine, die für die LOAD Instruktion 6 Taktzyklen braucht?
 - Wieviel schneller ist eine CPU bei der die STORE doppelt so schnell arbeitet?
- (b) Die Operationen werden mit folgender Häufigkeiten ausgeführt: ALU 50%, LOAD 15%, STORE 25%, BRANCH 10%. Berechnen Sie den idealen CPI (ohne zu berücksichtigen, dass der Zugriff auf den Hauptspeicher Stalls erfordert).

1.)

OPERATION	FREQUENZ	CPI	FREQ x CPI
ALU	25 %	4	1
LOAD	25 %	8	2
STORE	25 %	12	3
Branch	25 %	6	1.5

$$\sum = 7,5$$

a)

OPERATION	FREQUENZ	CPI	FREQ x CPI
ALU	25 %	4	1
LOAD	25 %	6	1.5
STORE	25 %	12	3
Branch	25 %	6	1.5

$\sum = 7 \implies \text{CPU ist ca. } 7,3\% \text{ schneller}$

OPERATION	FREQUENZ	CPI	FREQ x CPI
ALU	25 %	4	1
LOAD	25 %	8	2
STORE	25 %	6	1.5
Branch	25 %	6	1.5

$\sum = 6 \implies \text{CPU ist ca. } 20\% \text{ schneller}$

b.)

OPERATION	FREQUENZ	CPI	FREQ x CPI
ALU	50 %	4	2
LOAD	15 %	8	1.2
STORE	25 %	12	3
Branch	10 %	6	0.6

$\underline{\underline{\sum = 6,8}}$

- 2.) Ein Stack ist notwendig da beim Aufruf von Subroutinen Speicherplatz für lokale Variablen der Funktion irgendwo reserviert werden muss.

Zudem können Funktionsparameter im Stack abgelegt werden damit sie von der Subroutine später weiterverwendet werden können.

- 3.) $SLT \implies A < B \iff A \text{ ist } 0$
 $A \geq B \implies A - B \geq 0$

Daum slt oder auch set on less than wird in der "Operation"-control der Schalter auf 3 gesetzt. Alle dadurch wird beim höchstwertigen bit das set auf das tiefstwertigen bit des less übertragen. Alle bits ausser dem niederwertigsten haben bei less 0 als Input.

Um nun die Bedingung $A < B$ zu prüfen kann man einfach überprüfen ob $A - B$ positiv ist oder negativ. Das höchstwertige bit gibt das Vorzeichen an und so kann $A < B$ einfach bestimmt werden. Falls $A - B$ negativ ist muss das bit übertragen werden.

- 4.) Mit lui können die 16 oberen bits gesetzt werden
Mit ori die 16 unteren bits gesetzt werden

lui \$r3, upper16bits
ori \$r3, \$r3, lower16bits

- 5.)
- | OPERATION | OPCODE | FLANCT |
|-----------|---------|---------|
| and | 000 000 | 100 100 |
| or | 000 000 | 100 101 |
| add | 000 000 | 100 100 |
| subtract | 000 000 | 100 010 |
| slt | 000 000 | 101 010 |
| nor | 000 000 | 100 111 |
- Bitwert und A-Bwert werden auf 0 gesetzt, "Operation" = Multiplikator wird das erste resultat verwendet (AND)
Gleich wie beim and, ausser das beim Multiplikator das zweite resultat verwendet wird, (OR)
A-Bwert, B-wert sind 0, Multiplikator drittes resultat. (HALBADDIERER)
A-Bwert 0, B-wert 1, Multiplikator drittes resultat. (HALBADDIERER)
Gleich wie sub, ausser das bei der Operation der Wert auf 3 gesetzt wird \implies resultat wird weiterverwendet von ALU
A-Bwert 1, B-wert 1, Multiplikator erstes resultat