Cedric Aebi 17-103-235

Nicolas Müller 17-122-094

RA FS 18 Serie 3

Qiyang Hu, Givi Meishvili, Adrian Wälchli

Die dritte Serie ist bis Dienstag, den 17. April 2018 um 15:00 Uhr zu lösen und auf Ilias hochzuladen. Für Fragen steht im ILIAS jederzeit ein Forum zur Verfügung. Allfällige unlösbare Probleme sind uns so früh wie möglich mitzuteilen, wir werden gerne helfen. Viel Spass!

Vorankündigung

Zum Lösen der Assembler-Serien (Serien 4 und 5) benötigen Sie einen Raspberry Pi, welchen wir am Dienstag, den 10. April 2018 verteilen werden. Sie erhalten das Raspberry Pi Kit gegen eine Kaution von 120 Franken. Bitte denken Sie daran, die Kaution am vorher genannten Datum mitzunehmen. Die Kaution wird am Dienstag, den 15. Mai 2018 bei der Rückgabe wieder zurückbezahlt. Sollte es Ihnen nicht möglich sein, bei der Ausgabe des Materials anwesend zu sein, setzen Sie sich bitte frühzeitig mit einem der Assistenten in Verbindung.

Theorieteil

Gesamtpunktzahl: 12 Punkte

1 Performance Berechnungen (2 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` 4nsec

`LOAD` 8nsec

`STORE` 6nsec

`Branch` 6nsec.

Sie können davon ausgehen, 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 ALU doppelt so schnell arbeitet?

2 Stackverwendung bei Subroutinen (2 Punkte)

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

3 ALU & Most Significant Bit (2 Punkte)

Warum muss die ALU für das *most significant bit* anders aufgebaut werden als für die restlichen Stellen?

4 ALU & `SLT` (2 Punkte)

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

5 Pop und push (1 Punkt)

Geben Sie an, wie ein `pop` und ein `push` mit MIPS-Befehlssatz umgesetzt werden kann.

6 `loadi` (1 Punkt)

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

7 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 Benutzung, 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-Binär-Datei in den Speicher gelesen und ausgeführt werden kann.

Sie können davon ausgehen, dass die MIPS-Binär-Datei genau dem erwarteten Abbild im Speicher entspricht, d.h. die einzelnen Wörter sind in Big Endian Bytereihenfolge 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 angeben in welcher Funktion, auf welcher Linie in welcher Datei der Fehler ausgetreten 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.

(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 Programmierteil 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 Programmierteil als erfüllt gilt.

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

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

A Theorie

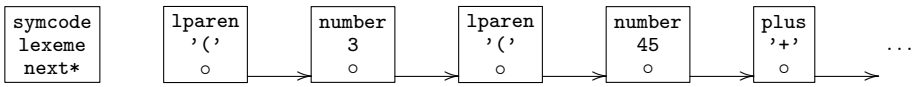
In `compiler.c` und `compiler.h` finden Sie die Implementation eines einfachen Compilers für arithmetische Ausdrücke. Unsere „Programmiersprache“ ist in der folgenden erweiterten Backus-Naur-Form (EBNF) gegeben:

```
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 Stackmachine 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 Programmstück wird z.B. (in Pseudo-Assembler) zu

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

Das Endresultat der Berechnung wird dabei wieder im Stack abgelegt.

1)

Op	Freq	CPI_i	Freq x CPI_i
ALU	25%	4	1
Load	25%	8	2
Store	25%	6	1,5
Branch	25%	6	1,5
			Σ = 6

4 1
6 1,5
6 1,5
6 1,5
5,5

new CPU Time = 5,5 x CC x IC 6/5,5 = 9,1% faster

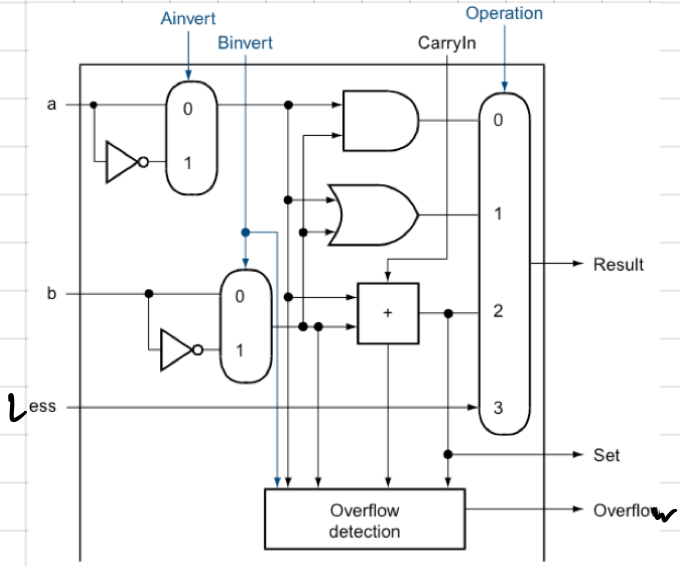
Op	Freq	CPI_i	Freq x CPI_i
ALU	25%	2	0,5
Load	25%	8	2
Store	25%	6	1,5
Branch	25%	6	1,5
			Σ = 5,5

6/5,5 9,1% faster

2) • Sobald eine Funktion (Subroutine) aufgerufen wird, wird Speicherplatz für lokale Variabeln reserviert. Dies geschieht auf den Stack. Zuden wird der FP auch abgelegt.

• Parameter, welche eine Unterfunktion braucht, werden auch auf den Stack für die Weiterverarbeitung abgelegt.

3)



Die slt Funktion muss realisiert werden können. Das msb gibt das less an die erste Stelle weiter. Zudem wird/muss geprüft werden, ob ein Overflow entstanden ist, was am msb geprüft wird.

4) Die Funktion slt gibt 1 zurück wenn $A < B$ ist, und 0 falls $A \geq B$. Da slt eine arithemische Operation ist, kann man sagen es handelt sich um eine Substraktion: $A - B < 0$ ist $A < B$.

Testen auf Gleichheit: $A - B = 0$ ist $A = B$

Wie es funktioniert:

Die "Operation" sagt den MUX 3. Das lsb bekommt nun als LESS, dass was bei den msb ausgerechnet wird. Alle anderen ALU's ausser das lsb bekommen als Less-input 0. Der Trick ist, dass nur das msb ALU eig. wichtig ist, denn wenn dieses 1 als set ausrechnet, ist die Zahl negativ und somit $A - B < 0 = A < B$

5) push: addiu $sp, $sp, -4 // Platz reservieren
sw $s0, ($sp) // abspeichern von Wert

pop: lw $s0, ($sp)
addiu $sp, $sp, 4

6) lui $t0, upper_16bits // dieser Befehl lädt die oberen 16bits der Konstante in t0.

ori $t0, $t0, lower_16bits // logisches Oder.

Skizze:

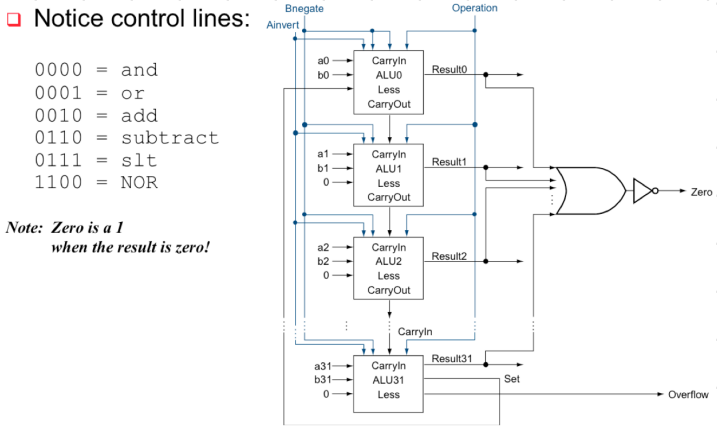
101010 .... | 0000 .... lui
0000 .... | 101010 ....
---------- ori
10101 .... | 10101 ....

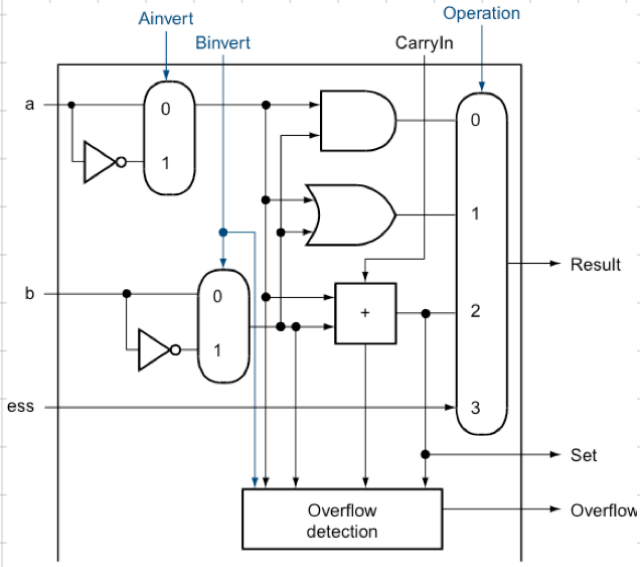
7 ALU: OPCodes (2 Punkte)

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.





and 0000 | Ainvert und Binvert auf 0, da es sich um ein einfaches bit-weise and handelt. Operation 00, da MUX-Eingang auf 0 sein muss. (0-Eingang ist and-Modul)

or 0001 | Ainvert, Binvert wieder 0 "..., Operation 01, da MUX-Eingang 1 sein muss für or-Gatter.

add 0010 | A", B" 0. Operation 10 für MUX-Eingang 2, da Volladdierer gebraucht wird.

sub 0110 | Ainvert 0, Binvert 1. Rest gleich. B-invert auf 1, da wenn man den 2ten Operanten negiert, man eine Subtraktion erhält. (2er Komplement ⇒ CarryIn auch 1)

slt 0111 | Ainvert 0, B 1, Operation 11 für MUX-Eingang 3. Somit verwendet die ALU dass Less für die Berechnung.

nor 1100 | Beide werden negiert und Operation führt and durch. Hier ist die DeMorgan-Regel richtig. (A nor B) = notA and notB.