EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Vorlesung Informatik der Systeme
SS 2015
Prof. Dr. M. Menth
Dipl.-Inform. W. Braun

Exercise 6

19. Mai 2015

Abgabe: 2. Juni 2015, 10.00 Uhr

In diesem Übungsblatt soll zwischen Assemblersprache und Maschinensprache für den DLX-Befehlssatz übersetzt werden. Der Befehlssatz ist ähnlich zu dem von MIPS, der in der Vorlesung bereits vorgestellt wurde. Um das Übungsblatt bearbeiten zu können, brauchen Sie aber noch weitere Informationen: Der DLX-Prozessor hat 32 Register (`r0` - `r31`). wobei das Register `r0` immer den Wert 0 hat (diese "Konstante" kann in einigen Befehlen geschickt genutzt werden).
Jeder DLX-Befehl besteht auf Maschinenebene aus 32 Bit, in denen die gewünschte Operation und die Operanden codiert sind. In Tabelle 1 (letzte Seite) sind einige DLX-Assembler-Befehle zusammen mit dem jeweiligen Befehlswort-Pattern aufgelistet. Bei Befehlen des I-Typs und des J-Typs bestimmen die ersten 6 Bit jeder Instruktion die Funktion, die dieser Befehl ausführt. Beim R-Typ sind diese Bits alle auf Null; die letzten 6 Bit geben die Operation an.
In untenstehendem Listing sind von einem kurzen Programm sowohl die Assemblerbefehle als auch die entsprechenden Maschinensprachen-Instruktionen aufgeführt. Der erste Befehl beginnt mit 100011, es handelt sich also um den `lw`-Befehl. Der zweite Befehl ist vom R-Typ (beginnt mit 000000), es ist der `xor`-Befehl (endet auf 100110).
Welche Quell- und Zielregister verwendet werden sollen, wird jeweils durch die 5 Bit lange binäre Darstellung der Registernummer codiert. Der `add`-Befehl speichert in Register $rd =$ `r2` das Ergebnis der Addition von $rs =$ `r2` und $rt =$ `r1`. Im Opcode wird deswegen der Abschnitt `sssss` mit 00010 (=2) gefüllt, `ttttt` mit 00001 (=1) und `ddddd` mit 00010 (=2). Genauso werden für I- und J-Typ *Immediate Values* (16 Bit) und *Offsets* für Sprünge (16 Bit bzw. 26 Bit) im Zweierkomplement codiert gespeichert.
Im Assembler-Code werden die Ziele der Jump-Befehle (hier: `j` und `beqz`) durch *Labels* (hier: `loop` und `end`) markiert: Nach dem Sprung geht es mit dem Befehl weiter, der auf das Label folgt. In den Maschinensprache-Instruktionen hingegen wird das Sprungziel relativ zur aktuellen Adresse angegeben. Der Wert, der in der Instruktion gespeichert wird, gibt an, um wie viele Byte der *Program Counter* verändert werden muss. Da jedes Befehlswort 4 Byte benötigt, entspricht ein Sprung um n Befehle ein Sprung um $4n$ Bytes. Außerdem muss berücksichtigt werden, dass der Befehlszähler automatisch nach jedem Befehl um 4 Byte erhöht wird (damit er auf den nächsten Befehl in Code-Reihenfolge zeigt): Das Sprungziel des `beqz`-Befehls liegt 4 Befehle hinter `beqz`. Der Befehlszähler würde automatisch auf den *add*-Befehl zeigen, er muss um 3 Befehle, also 12 Byte erhöht werden. Dieser Wert ist als Offset bei `beqz` codiert. Beim `j`-Befehl würde der Befehlszähler danach auf `sw` zeigen, er muss um 4 Befehle (16 Byte) nach hinten geschoben werden. Daher beträgt der Offset hier -16 (im Zweierkomplement: ... 1110000).

```
1      lw r1, 0x1000 ( r0 ) # 10001100000000010001000000000000 ; 0x8c011000
2      xor r2, r2, r2       # 00000000010000100001000000100110 ; 0x00421026
3  loop:
4      beqz r1, end         # 00010000001000000000000000001100 ; 0x1020000c
5      add r2, r2, r1       # 00000000010000010001000000100000 ; 0x00411020
6      subi r1, r1, 1       # 00101000001000010000000000000001 ; 0x28210001
7      j loop               # 00001011111111111111111111110000 ; 0x0bffffff0
8  end:
9      sw 0x1004 ( r0 ), r2 # 10101100000000010000100000000100 ; 0xac021004
```

Um die Instruktionen in kompakterer Form darzustellen, können sie natürlich in hexadezimaler statt binärer Darstellung notiert werden, wie in der letzten Spalte des Listings. Diese Form ist allerdings weniger gut geeignet um eine Instruktion zu entschlüsseln, da die einzelnen Abschnitte dann nicht mehr direkt ablesbar sind.

Problem 6.1: *Lesen von Maschinencode*

Wandeln Sie folgendes Stück Maschinencode zurück in Assemblercode:

```
1  10001100000000010001000000000000
2  00000000001000000001000000100000
3  00100000000000110000000000000001
4  00010000010000000000000000001100
5  00000000010000110001100000011001
6  00101000010000100000000000000001
7  00001011111111111111111111110000
8  10101100000000110001000000000100
```

Welche Funktion implementiert dieser Code? 8 Points

Problem 6.2: *Umwandlung von Assemblercode zu Maschinencode*

Übersetzen Sie das folgende Listing in Maschinencode für den DLX-Prozessor. Geben Sie jedes Maschinencode-Befehlswort auch in kompakter hexadezimaler Darstellung an.

```
1          lw r1, 0x1000(r0)
2          lw r2, 0x1004(r0)
3  loop:   beqz r1, end
4          slt r3, r1, r2
5          bnez r3, branch
6          sub r3, r1, r2
7          add r1, r2, r0
8          add r2, r3, r0
9          j loop
10 branch: sub r3, r2, r1
11         add r2, r1, r0
12         add r1, r3, r0
13         j loop
14 end:    sw 0x1008(r0), r2
```

12 Points

Total: 20 Points

Opcode-Pattern	Befehl	Erklärung
`100011sssssddddd iiiiiiiiiiiiiiii`	**lw** rd, imm(rs)	$rd \leftarrow MEM[rs + imm]$
`101011sssssddddd iiiiiiiiiiiiiiii`	**sw** imm(rs), rd	$MEM[rs + imm] \leftarrow rd$
`00111100000ddddd iiiiiiiiiiiiiiii`	**lhi** rd, imm	$rd[31{:}16] \leftarrow imm$
`001000sssssddddd iiiiiiiiiiiiiiii`	**addi** rd, rs, imm	$rd \leftarrow rs + imm$
`001001sssssddddd iiiiiiiiiiiiiiii`	**addui** rd, rs, imm	$rd \leftarrow rs + imm$
`001010sssssddddd iiiiiiiiiiiiiiii`	**subi** rd, rs, imm	$rd \leftarrow rs - imm$
`001011sssssddddd iiiiiiiiiiiiiiii`	**subui** rd, rs, imm	$rd \leftarrow rs - imm$
`001100sssssddddd iiiiiiiiiiiiiiii`	**andi** rd, rs, imm	$rd \leftarrow rs \wedge imm$
`001101sssssddddd iiiiiiiiiiiiiiii`	**ori** rd, rs, imm	$rd \leftarrow rs \vee imm$
`001110sssssddddd iiiiiiiiiiiiiiii`	**xori** rd, rs, imm	$rd \leftarrow rs \oplus imm$
`011000sssssddddd iiiiiiiiiiiiiiii`	**seqi** rd, rs, imm	$rd \leftarrow 1$ if $(rs = imm)$ else 0
`011001sssssddddd iiiiiiiiiiiiiiii`	**snei** rd, rs, imm	$rd \leftarrow 1$ if $(rs \neq imm)$ else 0
`011010sssssddddd iiiiiiiiiiiiiiii`	**slti** rd, rs, imm	$rd \leftarrow 1$ if $(rs < imm)$ else 0
`011011sssssddddd iiiiiiiiiiiiiiii`	**sgti** rd, rs, imm	$rd \leftarrow 1$ if $(rs > imm)$ else 0
`011100sssssddddd iiiiiiiiiiiiiiii`	**slei** rd, rs, imm	$rd \leftarrow 1$ if $(rs \leq imm)$ else 0
`011101sssssddddd iiiiiiiiiiiiiiii`	**sgei** rd, rs, imm	$rd \leftarrow 1$ if $(rs \geq imm)$ else 0
`000000ssssstttttddddd00000100000`	**add** rd, rs, rt	$rd \leftarrow rs + rt$
`000000ssssstttttddddd00000100001`	**addu** rd, rs, rt	$rd \leftarrow rs + rt$
`000000ssssstttttddddd00000100010`	**sub** rd, rs, rt	$rd \leftarrow rs - rt$
`000000ssssstttttddddd00000100011`	**subu** rd, rs, rt	$rd \leftarrow rs - rt$
`000000ssssstttttddddd00000011000`	**mult** rd, rs, rt	$rd \leftarrow rs * rt$
`000000ssssstttttddddd00000011001`	**multu** rd, rs, rt	$rd \leftarrow rs * rt$
`000000ssssstttttddddd00000011010`	**div** rd, rs, rt	$rd \leftarrow rs/rt$
`000000ssssstttttddddd00000011011`	**divu** rd, rs, rt	$rd \leftarrow rs/rt$
`000000ssssstttttddddd00000100100`	**and** rd, rs, rt	$rd \leftarrow rs \wedge rt$
`000000ssssstttttddddd00000100101`	**or** rd, rs, rt	$rd \leftarrow rs \vee rt$
`000000ssssstttttddddd00000100110`	**xor** rd, rs, rt	$rd \leftarrow rs \oplus rt$
`000000ssssstttttddddd00000000100`	**sll** rd, rs, rt	$rd \leftarrow rs \ll rt$
`000000ssssstttttddddd00000000110`	**srl** rd, rs, rt	$rd \leftarrow rs \gg rt$
`000000ssssstttttddddd00000101000`	**seq** rd, rs, rt	$rd \leftarrow 1$ if $(rs = rt)$ else 0
`000000ssssstttttddddd00000101001`	**sne** rd, rs, rt	$rd \leftarrow 1$ if $(rs \neq rt)$ else 0
`000000ssssstttttddddd00000101010`	**slt** rd, rs, rt	$rd \leftarrow 1$ if $(rs < rt)$ else 0
`000000ssssstttttddddd00000101011`	**sgt** rd, rs, rt	$rd \leftarrow 1$ if $(rs > rt)$ else 0
`000000ssssstttttddddd00000101100`	**sle** rd, rs, rt	$rd \leftarrow 1$ if $(rs \leq rt)$ else 0
`000000ssssstttttddddd00000101101`	**sge** rd, rs, rt	$rd \leftarrow 1$ if $(rs \geq rt)$ else 0
`000100sssss00000oooooooooooooooo`	**beqz** rs, offset	if $rs = 0 : PC \leftarrow PC + offset[+4]$
`000101sssss00000oooooooooooooooo`	**bnez** rs, offset	if $rs \neq 0 : PC \leftarrow PC + offset[+4]$
`000010oooooooooooooooooooooooo`	**j** offset	$PC \leftarrow PC + offset[+4]$

Tabelle 1: Einige DLX-Befehle mit dazugehörenden Maschinensprache-Instruktionen