Rechnerarchitektur Serie 1

Dominik Bodenmann 08-103-053
Orlando Signer 12-119-715

11. März 2014

1 Theorieteil

1.1 Aufgabe 1

5 Bytes, 4 Bytes für die Zeichen (T,e,s,t) und 1 Byte für \0.

1.2 Aufgabe 2

Listing 1: int-Array

```
1 int a[10];
2
3 int getAt(int i) {
4   return a[i];
5 }
6
7 int getAtWithPointer(int *a, int i) {
8   return *(a+i);
9 }
```

Listing 2: short-Array

```
1 short a[10];
2
3 short getAt(int i) {
4   return a[i];
5 }
6
7 short getAtWithPointer(short *a, int i) {
8   return *(a+i);
9 }
```

Bei Pointern beziehen sich die Rechenoperationen immer auf die Breite des Variablentyps (short 2 Byte, int 4 Byte). Somit zeigt auch *(a+i) auf die i-te Stelle im Array, egal ob es sich um ein short- oder int-array handelt.

1.3 Aufgabe 3

Listing 3: Ausgabe

```
1 bffff844
2 3ade68b1
3 68
4 de
5 bffff847
```

1. Der Wert von p: Die erste Speicheradresse von b.
2. p wird als long (4 Bytes) dereferenziert und danach inkrementiert. Da p vom Typ void ist, wird er nur um 1 erhöht.
3. p wird als char (1 Byte) dereferenziert und danach inkrementiert.
4. p wird als unsigned char (1 Byte) dereferenziert und danach inkrementiert.
5. Der Wert von p: Die 4. Speicheradresse von b.

1.4 Aufgabe 4

Preincrement: i = 1338, j = 1338
Postincrement: i = 1338, j = 1337

1.5 Aufgabe 5

1. Codestück

- 3. Zeile: x und px werden dereferenziert, es zeigen beide auf die 0. Stelle im Array. Ausgabe: "1 1"
- 5. Zeile: px wurde inkrementiert, zeigt jetzt auf die 1. Stelle. Ausgabe: "1 2"

2. Codestück

```
*(py--) = 10;
```

setzt zuerst y auf 10, danach wird py dekrementiert und auf 11 gesetzt. Die Ausgabe ist also "10 11"

1.6 Aufgabe 6

Grösse des structs: 4 * 1 Byte + 2 * 1 Byte + 2 Byte = 8 Bytes
Grösse des union: max(8 * 1 Byte, 4 Byte, 2 Byte) = 8 Bytes

Speicheraufteilung struct

d	d	c	b	$a[3]$	$a[2]$	$a[1]$	$a[0]$height

Speicheraufteilung union

$a[7]/d[3]$	$a[6]/d[3]$	$a[5]/d[2]$	$a[4]/d[2]$	$a[3]/d[1]/b$	$a[2]/d[1]/b$	$a[1]/d[0]/b$	$a[0]/d[0]/b$height

1.7 Aufgabe 7

Define ist eine Art Search-Replace Mechanismus und findet vor dem Compilieren statt. (praktisch für Konstanten)// Das Programmierstück gibt 3 aus.

2 Programmierteil

```
1  /* TODO: Task (b) Please fill in the following lines, then remove this line.
2   *
3   * author(s):   Dominik Bodenmann  08-103-053
4   *              Orlando Signer     12-119-715
5   *
6   * Please follow the instructions given in comments below.
7   * The file outputc1 shows what the output of this program
8   * should look like.
9   *
10  */
11
12 #include <stdlib.h>
13 #include <stdio.h>
14 #include <string.h>
15
16 /* Define abbreviations for the indices of the registers */
17 #define I_ZERO 0
18 #define I_AT 1
19 #define I_V0 2
20 #define I_V1 3
21 #define I_A0 4
22 #define I_A1 5
23 #define I_A2 6
24 #define I_A3 7
25 #define I_T0 8
26 #define I_T1 9
27 #define I_T2 10
28 #define I_T3 11
29 #define I_T4 12
30 #define I_T5 13
31 #define I_T6 14
32 #define I_T7 15
33 #define I_S0 16
34 #define I_S1 17
35 #define I_S2 18
36 #define I_S3 19
37 #define I_S4 20
38 #define I_S5 21
39 #define I_S6 22
40 #define I_S7 23
41 #define I_T8 24
42 #define I_T9 25
43 #define I_K0 26
44 #define I_K1 27
45 #define I_GP 28
```

```
46 #define I_SP 29
47 #define I_FP 30
48 #define I_RA 31
49
50 /* Define abbreviations for the operation codes (OC) and function codes (FC)
       */
51 #define OC_ADD    0x00
52 #define FC_ADD    0x20
53 #define OC_ADDI   0x08
54 #define OC_ADDIU  0x09
55 #define OC_ADDU   0x00
56 #define FC_ADDU   0x21
57 #define OC_AND    0x00
58 #define FC_AND    0x24
59 #define OC_ANDI   0x0C
60 #define OC_BEQ    0x04
61 #define OC_BNE    0x05
62 #define OC_DIV    0x00
63 #define FC_DIV    0x1A
64 #define OC_J    0x02
65 #define OC_JAL    0x03
66 #define OC_JR   0x00
67 #define FC_JR   0x08
68 #define OC_LBU    0x00
69 #define FC_LBU    0x24
70 #define OC_LHU    0x00
71 #define FC_LHU    0x25
72 #define OC_LUI    0x0F
73 #define OC_LW   0x23
74 #define OC_MULT   0x00
75 #define FC_MULT   0x18
76 #define OC_NOR    0x00
77 #define FC_NOR    0x27
78 #define OC_OR   0x00
79 #define FC_OR   0x25
80 #define OC_ORI    0x0D
81 #define OC_SLT    0x00
82 #define FC_SLT    0x2A
83 #define OC_SLTI   0x0A
84 #define OC_SLTIU  0x0B
85 #define OC_SLTU   0x00
86 #define FC_SLTU   0x2B
87 #define OC_SLL    0x00
88 #define FC_SLL    0x00
89 #define OC_SRL    0x00
90 #define FC_SRL    0x02
91 #define OC_SB   0x28
92 #define OC_SH   0x29
93 #define OC_SW   0x2B
94 #define OC_SUB    0x00
95 #define FC_SUB    0x22
96 #define OC_SUBU   0x00
97 #define FC_SUBU   0x23
98 /* Just a handy abbreviation */
```

```
99  #define OC_ZERO 0x00
100 /* Stop Operation --- not actual MIPS */
101 #define OC_STOP 0x3F
102
103 /* Number of operations, functions and registers */
104 #define OPERATION_COUNT 64
105 #define FUNCTION_COUNT 64
106 #define REGISTER_COUNT 32
107
108 /* Maximal length of operation and function names */
109 #define OP_NAME_LENGTH 4
110 #define FUNC_NAME_LENGTH 4
111
112 /* Define some types */
113 typedef unsigned long word;
114 typedef unsigned short halfword;
115 typedef unsigned char byte;
116
117 /* TODO Task (c) add bitfields InstructionTypeI, InstructionTypeJ and
        InstructionTypeR here */
118 typedef struct {
119   unsigned immediate:16;
120   unsigned rt:5;
121   unsigned rs:5;
122   unsigned opcode:6;
123 } InstructionTypeI;
124
125 typedef struct {
126   unsigned address:26;
127   unsigned opcode:6;
128 } InstructionTypeJ;
129
130 typedef struct {
131   unsigned funct:6;
132   unsigned shamt:5;
133   unsigned rd:5;
134   unsigned rt:5;
135   unsigned rs:5;
136   unsigned opcode:6;
137 } InstructionTypeR;
138
139 /* TODO Task (d) add union Instruction here */
140 typedef union {
141   InstructionTypeI i;
142   InstructionTypeJ j;
143   InstructionTypeR r;
144 } Instruction;
145
146 /* TODO Task (e) add enumeration InstructionType here */
147 typedef enum {
148   iType,
149   jType,
150   rType,
151   specialType
```

```
152 } InstructionType;
153
154 /* TODO Task (f) add structure Operation here */
155 typedef struct {
156   char name[OP_NAME_LENGTH];
157   InstructionType type;
158   void (*operation)(Instruction *instruction);
159 } Operation;
160
161 /* TODO Task (g) add structure Function here */
162 typedef struct {
163   char name[FUNC_NAME_LENGTH];
164   void (*function)(Instruction *instruction);
165 } Function;
166
167 /* Operation and function dispatcher */
168 Operation operations[OPERATION_COUNT];
169 Function functions[FUNCTION_COUNT];
170
171 /* Assembles the given parts of an I-type instruction into a single word*/
172 word create_itype_hex(unsigned short immediate, unsigned short rt, unsigned
        short rs, unsigned short opcode) {
173     return immediate + (rt << 16)+ (rs <<21) + (opcode << 26);
174 }
175
176 /* Assembles the given parts of an J-type instruction into a single word*/
177 word create_jtype_hex(unsigned long address, unsigned short opcode) {
178     return address + (opcode << 26);
179 }
180
181 /* Assembles the given parts of an R-type instruction into a single word*/
182 word create_rtype_hex(unsigned short funct, unsigned short shamt, unsigned
        short rd, unsigned short rt, unsigned short rs, unsigned short opcode) {
183     return funct + (shamt << 6) + (rd << 11) + (rt << 16) + (rs <<21) + (
            opcode << 26);
184 }
185
186 /* Assembles the given parts of an special-type instruction into a single
        word*/
187 word create_specialtype_hex(unsigned short opcode) {
188     return create_jtype_hex(0x0000, opcode);
189 }
190
191 /* Copies operation into the operation dispatcher */
192 void assignOperation(unsigned short opCode, const char name[OP_NAME_LENGTH
        +1], InstructionType type, void (*operation)(Instruction*)) {
193     strcpy(operations[opCode].name, name);
194     operations[opCode].type=type;
195     operations[opCode].operation = operation;
196 }
197
198 /* Copies functions into the function dispatcher */
199 void assignFunction(unsigned short funct, const char name[FUNC_NAME_LENGTH
        +1], void (*function)(Instruction*)) {
```

```
200     strcpy(functions[funct].name, name);
201     functions[funct].function = function;
202 }
203
204 /* Initialize the "hardware" and operation and function dispatcher */
205 void initialize() {
206     int i;
207     /* Initialize operations  with default values */
208     for (i=0; i<OPERATION_COUNT; ++i) {
209         assignOperation(i, "ndef", specialType, 0);
210     }
211
212     /* to deal with operations with OpCode = 0, i.e. R-Type */
213     assignOperation(OC_ZERO, "zero", rType, 0);
214
215     /* assign some actual operations */
216     assignOperation(OC_ADDI, "addi", iType, 0);
217     assignOperation(OC_J, "j", jType, 0);
218     assignOperation(OC_LUI, "lui", iType, 0);
219     assignOperation(OC_LW, "lw", iType, 0);
220     assignOperation(OC_ORI, "ori", iType, 0);
221     assignOperation(OC_SW, "sw", iType, 0);
222     assignOperation(OC_STOP,"stop", specialType, 0);
223
224     /* Initialize operations with OpCode = 0 and corresponding functions with
            default values*/
225     for (i=0; i<FUNCTION_COUNT; ++i) {
226         assignFunction(i, "ndef", 0);
227     }
228
229     /* assign some actual functions */
230     assignFunction(FC_ADD, "add", 0);
231     assignFunction(FC_SUB, "sub", 0);
232 }
233
234
235 void printInstruction(Instruction *i) {
236 /* TODO Task (h) complete printInstruction here */
237   Operation o = operations[i->i.opcode];
238   switch (o.type) {
239     case iType:
240       printf("%-4s %.2i %.2i 0x%.4x\n", o.name, i->i.rt, i->i.rs, i->i.
              immediate);
241       break;
242     case jType:
243       printf("%-4s %.8x\n", o.name, i->j.address);
244       break;
245     case rType: {
246       Function f = functions[i->r.funct];
247       printf("%-4s %.2i %.2i %.2i 0x%.4x\n", f.name, i->r.rd, i->r.rs, i->r.
              rt, i->r.shamt);
248       break;
249       }
250     case specialType:
```

```
251         printf("%s\n", o.name );
252         break;
253     }
254 }
255
256 void testPrint(word w) {
257     Instruction * instruction = (Instruction *) &w;
258     printInstruction(instruction);
259 }
260
261 int main(int argc, const char * argv[]) {
262     initialize();
263     testPrint(create_rtype_hex(FC_ADD, 0x0000, I_T0, I_T1, I_T2, OC_ADD));
264     testPrint(create_itype_hex(0x0001, I_T0, I_ZERO, OC_ADDI));
265     testPrint(create_jtype_hex(0xCD1234, OC_J));
266     testPrint(create_itype_hex(0xBBBB, I_T0, I_ZERO, OC_LUI));
267     testPrint(create_itype_hex(0xA03B, I_T0, I_T1, OC_LW));
268     testPrint(create_itype_hex(0x0101, I_T0, I_T0, OC_ORI));
269     testPrint(create_rtype_hex(FC_SUB, 0x0002, I_T0, I_T1, I_T2, OC_SUB));
270     testPrint(create_itype_hex(0xD070, I_T0, I_T1, OC_SW));
271     testPrint(create_specialtype_hex(OC_STOP));
272     return 0;
273 }
```