# Rechnerarchitektur Serie 2

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

25. März 2014

## 1 Theorie-Teil

### 1.1 Aufgabe 1

**Listing 1: Ausgabe**

```
A: 10
B: 11
C: 12
```

### 1.2 Aufgabe 2

int * a: initialisiert in a einen Pointer auf einen Integer
int const * b: initialisiert in b einen Pointer auf einen konstanten Integer
int * const c: initialisiert in c einen konstanten Pointer auf einen Integer
int const * const d: initialisiert in d einen konstanten Pointer auf einen konstanten Integer

### 1.3 Aufgabe 3

Es sollte nur bis i=9 gehen (Indizes von 0 bis 9), d.h. es gibt einen Segmentation Fault.

### 1.4 Aufgabe 4

**Listing 2: C-Code**

```
while ($s2 != $zero){
  do something;
  $s2 -= $s1;
}
```

## 1.5 Aufgabe 5

**Listing 3: Erweiterung**

```
1 bge $s2, $s1, Label wird zu:
2 slt $at, $s2, $s1
3 beq $at, $zero, Label
```

## 1.6 Aufgabe 6

Big Endian (Start bei der Adresse 10004):
$(0110\ 1101\ 1000\ 1100\ 0010\ 0100\ 0000\ 0000)_2 = (1837900800)_{10}$
Little Endian (Start bei der Adresse 10007):
$(0000\ 0000\ 0010\ 0100\ 1000\ 1100\ 0110\ 1101)_2 = (2395245)_{10}$

## 1.7 Aufgabe 7

**Listing 4: Assembler**

```
1 lw $t3, $t1($t2)
2 mult $t3, $t0
3 sw $LO, $t1($t2)
```

## 1.8 Aufgabe 8

**Listing 5: Laden in Register**

```
1 lw $s2 9($s1)
```

# 2 Programmierteil

**Listing 6: mips.c**

```
/* TODO: Task (b) Please fill in the following lines, then remove this line.
 *
 * author(s):   Dominik Bodenmann
 *    Orlando Signer
 *
 * modified:    2010-01-07
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mips.h"

/* The "Hardware" */
byte memory[MEMORY_SIZE];
word registers[REGISTER_COUNT];
word pc;

/* To stop the MIPS machine */
int doRun;

/* In case you want to watch the machine working */
int verbose = TRUE;

/* Operation and function dispatcher */
Operation operations[OPERATION_COUNT];
Function functions[FUNCTION_COUNT];

/* ========================================================================= */
/* Some useful helpers */

/* Assembles the given parts of an I-type instruction into a single word*/
word create_itype_hex(unsigned immediate, unsigned rt, unsigned rs, unsigned
    opcode) {
  return immediate + (rt << 16)+ (rs <<21) + (opcode << 26);
}

/* Assembles the given parts of an J-type instruction into a single word*/
word create_jtype_hex(unsigned address, unsigned opcode) {
  return address + (opcode << 26);
}

/* Assembles the given parts of an R-type instruction into a single word*/
word create_rtype_hex(unsigned funct, unsigned shamt, unsigned rd, unsigned
    rt, unsigned rs, unsigned opcode) {
  return funct + (shamt << 6) + (rd << 11) + (rt << 16) + (rs <<21) + (opcode
      << 26);
}
```

```
47
48 /* Extends a 16 bit halfword to a 32 bit word with the value of the most
       significant bit */
49 word signExtend(halfword value) {
50     return (value ^ 0x8000) - 0x8000;
51 }
52
53 /* Extends a 16 bit halfword to a 32 bit word by adding leading zeros */
54 word zeroExtend(halfword value) {
55     return (value | 0x00000000);
56 }
57
58 /* To make some noise */
59 void printInstruction(Instruction *i) {
60     Operation o = operations[i->i.opcode];
61     Function f;
62     switch (o.type) {
63         case iType:
64             printf("%-4s %02i=0x%08ux, %02i=0x%08ux, 0x%04x\n", o.name, i->i.
                    rt, registers[i->i.rt], i->i.rs, registers[i->i.rs], i->i.
                    immediate );
65             break;
66         case jType:
67             printf("%-4s 0x%08x\n", o.name, i->j.address);
68             break;
69         case rType:
70             f = functions[i->r.funct];
71             printf("%-4s %02i=0x%08ux, %02i=0x%08ux, %02i=0x%08ux, 0x%04x\n",
                     f.name, i->r.rd, registers[i->r.rd], i->r.rs, registers[i->r
                    .rs],i->r.rt, registers[i->r.rt],i->r.shamt);
72             break;
73         case specialType:
74             printf("%-4s\n", o.name);
75             break;
76     }
77 }
78
79 /* ============================================================================ */
80 /* Memory operations */
81
82 /* Store a word to memory */
83 void storeWord(word w, word location) {
84   /* TODO: Task (c) implement storeWord here */
85   memory[location] = (w >> (8*3));
86   memory[location+1] = (w >> (8*2));
87   memory[location+2] = (w >> (8*1));
88   memory[location+3] = w;
89 }
90
91 /* Load a word from memory */
92 word loadWordFrom(word location) {
93   word w = 0;
94   w += (memory[location]   << (8*3));
95   w += (memory[location+1] << (8*2));
```

```
96    w += (memory[location+2] << (8*1));
97    w +=  memory[location+3];
98    return w;
99  }
100
101 /* ========================================================================== */
102 /* Initialize and run */
103 void assignOperation(unsigned short opCode, const char name[OP_NAME_LENGTH
        +1], InstructionType type, void (*operation)(Instruction*)) {
104     strcpy(operations[opCode].name, name);
105     operations[opCode].type=type;
106     operations[opCode].operation = operation;
107 }
108
109 void assignFunction(unsigned short funct, const char name[FUNC_NAME_LENGTH
        +1], void (*function)(Instruction*)) {
110     strcpy(functions[funct].name, name);
111     functions[funct].function = function;
112 }
113
114 /* Initialize the "hardware" and operation and function dispatcher */
115 void initialize() {
116   int i;
117   /* Initialize operations */
118   for (i=0; i<OPERATION_COUNT; ++i) {
119     assignOperation(i, "ndef", specialType, &undefinedOperation);
120   }
121   assignOperation(OC_ZERO, "zero", rType, &opCodeZeroOperation);
122   /* To stop the MIPS machine */
123   assignOperation(OC_STOP,"stop", specialType, &stopOperation);
124
125   assignOperation(OC_ADDI, "addi", iType, &mips_addi);
126         assignOperation(OC_JAL, "jal", jType ,&mips_jal);
127   assignOperation(OC_LUI, "lui", iType ,&mips_lui);
128   assignOperation(OC_LW, "lw", iType, &mips_lw);
129   assignOperation(OC_ORI, "ori", iType, &mips_ori);
130   assignOperation(OC_SW, "sw", iType, &mips_sw);
131
132   /* Initialize operations with OpCode = 0 and corresponding functions */
133   for (i=0; i<FUNCTION_COUNT; ++i) {
134     assignFunction(i, "ndef", &undefinedFunction);
135   }
136   assignFunction(FC_ADD, "add", &mips_add);
137   assignFunction(FC_SUB, "sub", &mips_sub);
138
139         /* Initialize memory */
140   for (i=0; i<MEMORY_SIZE; ++i) {
141     memory[i] = 0;
142   }
143
144   /* Initialize registers */
145   for (i=0; i<REGISTER_COUNT; ++i) {
146     registers[i]= 0;
147   }
```

```
148
149   /* Stack pointer */
150   SP=65535;
151
152   /* Initialize program counter */
153   pc = 0;
154
155   /* Yes, we want the machine to run */
156   doRun = TRUE;
157 }
158
159 /* Fetch and execute */
160 void run() {
161   while (doRun) {
162     /* Fetch Instruction*/
163     word w  = loadWordFrom(pc);
164     Instruction *instruction = (Instruction *) &w;
165     /* Please note: the program counter is incremented before the operation
          is executed */
166     pc += 4;
167     /* Execute Instruction*/
168     operations[instruction->i.opcode].operation(instruction);
169     /* In case you want to watch the machine */
170                 if (verbose) {
171                     printInstruction(instruction);
172                 }
173   }
174 }
175
176 /* ================================================================== */
177 /* "Special" operations --- only for "internal" usage */
178
179 /* To deal with "undefined" behaviour */
180 void undefinedOperation(Instruction *instruction) {
181     printf("%s in %s, line %i: Unknown opcode: %x\n",__func__, __FILE__,
          __LINE__, instruction->i.opcode);
182     exit(0);
183 }
184
185 /* To deal with "undefined" behaviour */
186 void undefinedFunction(Instruction *instruction) {
187     printf("%s in %s, line %i: Unknown funct: %x\n",__func__, __FILE__,
          __LINE__, instruction->r.funct);
188     exit(0);
189 }
190
191 /* To deal with operations with opcode = 0 */
192 void opCodeZeroOperation(Instruction *instruction) {
193   functions[instruction->r.funct].function(instruction);
194 }
195
196 /* To stop the machine */
197 void stopOperation(Instruction *instruction) {
198     doRun = FALSE;
```

```
199 }
200
201 /* ============================================================================
        */
202 /* Implemented MIPS operations */
203
204 /* ADD */
205 void mips_add(Instruction *instruction) {
206     /* TODO: Task (e) implement ADD here */
207     InstructionTypeR r = instruction->r;
208     word rt = registers[r.rt];
209     word rs = registers[r.rs];
210     registers[r.rd] = rt + rs;
211 }
212
213 /* ADDI */
214 void mips_addi(Instruction *instruction) {
215   /* TODO: Task (e) implement ADDI here */
216     InstructionTypeI i = instruction->i;
217     word rs = registers[i.rs];
218     word immediate = (signed) signExtend(i.immediate);
219     registers[i.rt] = rs + immediate;
220 }
221
222 /* JAL */
223 void mips_jal(Instruction *instruction) {
224   /* TODO: Task (e) implement JAL here */
225     InstructionTypeJ j = instruction->j;
226     /* We dont need to add 4 to the PC as it is incremented before the
            operation. */
227     RA = pc;
228     pc = (pc & 0xF0000000) + (j.address << 2);
229 }
230
231 /* LUI */
232 void mips_lui(Instruction *instruction) {
233   /* TODO: Task (e) implement LUI here */
234     InstructionTypeI i = instruction->i;
235     registers[i.rt] = i.immediate << 16;
236 }
237
238 /* LW */
239 void mips_lw(Instruction *instruction) {
240   InstructionTypeI i = instruction->i;
241   registers[i.rt] = loadWordFrom(registers[i.rs] + (signed)signExtend(i.
          immediate));
242 }
243
244 /* ORI */
245 void mips_ori(Instruction *instruction) {
246   InstructionTypeI i = instruction->i;
247   registers[i.rt] = registers[i.rs] | zeroExtend(i.immediate);
248 }
249
```

```
250 /* SUB */
251 void mips_sub(Instruction *instruction) {
252   InstructionTypeR r = instruction->r;
253   registers[r.rd] = (signed)registers[r.rs] - (signed)registers[r.rt];
254 }
255
256 /* SW */
257 void mips_sw(Instruction *instruction) {
258   /* TODO: Task (e) implement SW here */
259     InstructionTypeI i = instruction->i;
260     word location = registers[i.rs] + (signed) signExtend(i.immediate);
261     storeWord(registers[i.rt], location);
262 }
```

**Listing 7: test.c**

```
/* TODO: Task (b) Please fill in the following lines, then remove this line.
 *
 * author(s):   Dominik Bodenmann
 *              Orlando Signer
 * modified:    2010-01-07
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "mips.h"

/* executes exactly the given instrution */
void test_execute(word instr) {
  word w;
  Instruction *instruction;

  /* Store the executable word  */
  storeWord(instr, pc);

  /* Fetch the next Instruction */
  w  = loadWordFrom(pc);
  instruction = (Instruction *) &w;
  pc += 4;

  /* Execute the fetched instruction*/
  operations[instruction->i.opcode].operation(instruction);
  assert(ZERO == 0);
}

/* ADD */
void test_add() {
  T1=1;
  T2=1;
  test_execute(create_rtype_hex(FC_ADD, 0x0000, I_T0, I_T1, I_T2, OC_ADD));
  assert(T0==2);

  T1=1;
  T2=-1;
  test_execute(create_rtype_hex(FC_ADD, 0x0000, I_T0, I_T1, I_T2, OC_ADD));
  assert(T0==0);

  T1=-1;
  T2=-1;
  test_execute(create_rtype_hex(FC_ADD, 0x0000, I_T0, I_T1, I_T2, OC_ADD));
  assert(T0==-2);
}

/* ADDI */
void test_addi() {
  test_execute(create_itype_hex(0xFFFF, I_T0, I_ZERO, OC_ADDI));
```

```
53    assert(T0 == -1);
54    test_execute(create_itype_hex(1, I_T0, I_T0, OC_ADDI));
55    assert(T0 ==  0);
56
57    test_execute(create_itype_hex(0xFFFF, I_T0, I_ZERO, OC_ADDI));
58    assert(T0 == -1);
59    test_execute(create_itype_hex(0xFFFF, I_T0, I_T0, OC_ADDI));
60    assert(T0 == -2);
61
62    test_execute(create_itype_hex(3, I_T0, I_ZERO, OC_ADDI));
63    assert(T0 ==  3);
64    test_execute(create_itype_hex(1, I_T1, I_T0, OC_ADDI));
65    assert(T0 ==  3);
66    assert(T1 ==  4);
67  }
68
69  /* JAL */
70  void test_jal() {
71      int pcSaved;
72      word w;
73      Instruction* instruction;
74
75      pc = 0x00000000;
76      pcSaved = pc;
77    test_execute(create_jtype_hex(0x0001, OC_JAL));
78    assert(RA == pcSaved + 4);
79      assert(pc == 4);
80
81      /* The following test is executed manually as the desired pc is outside
            the memory,
82       * i.e. the test needs to bypass actually storing the instruction in the
             memory.
83       */
84    initialize();
85      pc = 0xAF000000;
86    pcSaved = pc;
87      w = create_jtype_hex(0x0001, OC_JAL);
88      instruction = (Instruction *) &w;
89      pc += 4;
90      operations[instruction->i.opcode].operation(instruction);
91    assert(RA == pcSaved + 4);
92      assert(pc == 0xA0000004);
93  }
94
95  /* LUI */
96  void test_lui() {
97      test_execute(create_itype_hex(0xFFFF, I_T0, I_ZERO, OC_LUI));
98      assert(T0 == 0xFFFF0000);
99
100     test_execute(create_itype_hex(0x0001, I_T0, I_ZERO, OC_LUI));
101     assert(T0 == 0x00010000);
102 }
103
104 /* LW */
```

```
105 void test_lw() {
106     /* TODO: Task (d) add test for LW here */
107     word location1 = 0x00001000;
108 
109     word w = 0x87654321;
110     T1 = location1;
111 
112     storeWord(w, location1);
113     test_execute(create_itype_hex(0x0000, I_T0, I_T1, OC_LW));
114     assert(w == T0);
115 }
116 
117 /* ORI */
118 void test_ori() {
119     /* TODO: Task (d) add test for ORI here */
120     word w = 0xAAAAAAAA; /* 0b1010.... */
121     T1 = w;
122 
123     test_execute(create_itype_hex(0x5555, I_T0, I_T1, OC_ORI)); /* 0x5555 0
            b0101... */
124     assert(0xAAAAFFFF == T0);
125 
126     w = 0xFFFF0000;
127     T1 = w;
128     test_execute(create_itype_hex(0xFFFF, I_T0, I_T1, OC_ORI));
129     assert(0xFFFFFFFF == T0);
130 }
131 
132 /* SUB */
133 void test_sub() {
134     /* TODO: Task (d) add test for SUB here */
135     /* T0 = T2 -T1 */
136     T1=1;
137   T2=1;
138   test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
139   assert(T0==0);
140 
141   T1=1;
142   T2=-1;
143   test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
144   assert(T0==-2);
145 
146   T1=-1;
147   T2=1;
148   test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
149   assert(T0==2);
150 
151   T1=-1;
152   T2=-1;
153   test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
154   assert(T0==0);
155 }
156 
157 /* SW */
```

```
158 void test_sw() {
159     word location1 = 0x00001000;
160     word location2 = 0x00001004;
161 
162     word w = 0xFFFFFFFF;
163     T0 = w;
164     T1 = location1;
165     test_execute(create_itype_hex(0x0000, I_T0, I_T1, OC_SW));
166     assert(loadWordFrom(location1) == w);
167 
168     w =0x12345678;
169     T0 = w;
170     T1 = location2;
171     test_execute(create_itype_hex(0xFFFC, I_T0, I_T1, OC_SW));
172     assert(loadWordFrom(location1) == w);
173 }
174 
175 /*
    ============================================================================
     */
176 /* make sure you've got a "fresh" environment for every test */
177 void execute_test(void (*test)(void)) {
178     initialize();
179     test();
180 }
181 
182 /* executes all tests */
183 int main (int argc, const char * argv[]) {
184   execute_test(&test_add);
185   execute_test(&test_addi);
186   execute_test(&test_jal);
187   execute_test(&test_lui);
188   execute_test(&test_lw);
189   execute_test(&test_ori);
190   execute_test(&test_sub);
191   execute_test(&test_sw);
192   return 0;
193 }
```