Rechnerarchitektur Serie 3

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

8. April 2014

1 Theorie-Teil

1.1 Aufgabe 1

	t	$Freq$	CPI	$Freq * CPI$	CPI	$Freq * CPI$	CPI	$Freq * CPI$
ALU	$5nsec$	25%	2	0.5	2	0.5	**1**	0.25
$LOAD$	$10nsec$	25%	4	1.0	**6**	1.5	4	1.0
$STORE$	$7.5nsec$	25%	3	0.75	3	0.75	1	0.75
$Branch$	$7.5nsec$	25%	3	0.75	3	0.75	1	0.75
				3.0		3.5		2.75height

1. Eine Maschine, die für die LOAD Instruktion 6 Taktzyklen braucht, ist also $3.5/3.0 - 1 = 16.7\%$ langsamer.

2. Eine CPU, bei der die ALU doppelt so schnell arbeitet, ist also $2.75/3.0 - 1 = 8.3\%$ schneller.

1.2 Aufgabe 2

1. Darauf kann die Rücksprungadresse für die Vortsetzung der Programmbearbeitung gespeichert werden.

2. Darauf können die Aufrufparameter gelegt werden, damit sie von der Subroutine gelesen werden können.

1.3 Aufgabe 3

Damit der Overflow behandelt werden kann:

$VorzeichenA$	$VorzeichenB$	$Binvert$	$Carryout$	$Vorz.Result.$	$Korr.Vorz.Res.$	$Overflow$
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	1	1	0
0	1	1	1	0	0	0
1	0	0	0	1	1	0
1	0	1	1	0	0	0
1	1	0	1	0	1	1
1	1	1	1	1	1	0

Die ersten 5 Argumente werden für die Overflowdetection gebraucht. Ist das B invert Bit nicht gleich dem Carry out Bit, so gibt es einen Overflow. (Bei ALU31, da da die Vorzeichen abgespeichert sind)

1.4 Aufgabe 4

- Beim slt-Befehl wird a-b gerechnet. Falls $a - b < 0$ erhält man im Vorzeichenbit (ALU31) eine 1.
- Die ALU unterstützt diesen Befehl, indem sie das Set des ALU31 Bits mit dem Less des ALU0 Bits verbindet. Alle anderen Less sind 0;

1.5 Aufgabe 5

Listing 1: MIPS push und pop

```
1 # Push und pop auf dem Stack. $sp bezeichnet den Stackpointer.
2 push:
3 subi $sp $sp 4
4 sw  $t0 0($sp)
5
6 pop:
7 lw $t0 0($sp)
8 addi $sp $sp 4
```

1.6 Aufgabe 6

1.7 Aufgabe 7

Befehl	A invert	B invert	Operation
and	0	0	0
or	0	0	1
add	0	0	1
subtract	0	1	1
slt	0	1	1
nor	1	1	0

and Mit dem Opcode 00 wird der erste Eingand des Operationsmuxes (das AND-Gatter) angesteuert, und somit muss A und B nicht invertiert werden.

or Gleich wie oben. Mit dem Opcode 01 kann direkt der zweite Eingang des Operationsmuxes (das OR-Gatter) angesteuert werden, und somit müssen A und B

wiederum nicht invertiert werden.

add Wiederum kann mit dem Opcode 10 der dritte Eingang des Operationsmuxes (Addition) angesteuert werden, und Aund B müssen wiederum nicht invertiert werden.

subtract Wenn man b negiert (also invertiert) kann man einfach noch eine Addition ausführen (Opcode 10)

slt Es wird wieder $a - b$ gerechnet (siehe subtract). Also B invertieren. Falls B grösser als A ist, soll das 31ste Bit den Wert 1 haben, welcher ebenfalls aufs erste Bit zurückgeführt wird. Damit kann man beim Operationsmux den vierten Eingang (Opcode 11) wählen, und erhält nun als Ausgabe eine 1, falls B grösser als A ist und sonst eine 0.

nor Da $aNORb = notaANDnotb$ ist, kann man A und B invertieren und das AND-Gatter (Opcode 00) anlegen.

2 Programmierteil

Listing 2: compile.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 "memory.h"
#include "mips.h"
#include "compiler.h"

int main ( int argc, char** argv ) {
    /* TODO: Task (c) implement main */
  if (argc != 3) {
    printf("usage: <%s> expression filename\n", argv[0]);
    return EXIT_FAILURE;
  }
  char * expression = argv[1];
  char * filename = argv[2];
  compiler(expression, filename);
  return EXIT_SUCCESS;
}
```

Listing 3: 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 <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "mips.h"
#include "memory.h"

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

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

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

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

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

void error(const char *functionName, const char *fileName, int lineNumber,
    char *message, ...) {
   /* TODO: Task (e) implement error */
   printf("%s in %s, line %i: %s\n", functionName, fileName, lineNumber,
       message);

   exit(EXIT_FAILURE);
}



/* Assembles the given parts of an I-type instruction into a single word*/
word create_itype_hex(unsigned short immediate, unsigned short rt, unsigned
    short rs, unsigned short 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 long address, unsigned short opcode) {
```

```
50    return address + (opcode << 26);
51  }
52
53  /* Assembles the given parts of an R-type instruction into a single word*/
54  word create_rtype_hex(unsigned short funct, unsigned short shamt, unsigned
        short rd, unsigned short rt, unsigned short rs, unsigned short opcode) {
55    return funct + (shamt << 6) + (rd << 11) + (rt << 16) + (rs <<21) + (opcode
          << 26);
56  }
57
58  /* Extends a 16 bit halfword to a 32 bit word with the value of the most
        significant bit */
59  word signExtend(halfword value) {
60      return (value ^ 0x8000) - 0x8000;
61  }
62
63
64  /* Extends a 16 bit halfword to a 32 bit word by adding leading zeros */
65  word zeroExtend(halfword value) {
66      return (value | 0x00000000);
67  }
68
69  /* To make some noise */
70  void printInstruction(Instruction *i) {
71      Operation o = operations[i->i.opcode];
72      Function f;
73      switch (o.type) {
74          case iType:
75              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 );
76              break;
77          case jType:
78              printf("%-4s 0x%08x\n", o.name, i->j.address);
79              break;
80          case rType:
81              f = functions[i->r.funct];
82              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);
83              break;
84          case specialType:
85              printf("%-4s\n", o.name);
86              break;
87      }
88  }
89
90
91  /* ================================================================== */
92  /* Initialize and run */
93
94  void assignOperation(unsigned short opCode, const char name[OP_NAME_LENGTH
        +1], char type, void (*operation)(Instruction*)) {
95      strcpy(operations[opCode].name, name);
```

```
96      operations[opCode].type=type;
97      operations[opCode].operation = operation;
98  }
99
100 void assignFunction(unsigned short funct, const char name[FUNC_NAME_LENGTH
        +1], void (*function)(Instruction*)) {
101     strcpy(functions[funct].name, name);
102     functions[funct].function = function;
103 }
104
105 /* Initialize the "hardware" and operation and function dispatcher */
106 void initialize() {
107   int i;
108   /* Initialize operations */
109   for (i=0; i<OPERATION_COUNT; ++i) {
110     assignOperation(i, "ndef", specialType, &undefinedOperation);
111   }
112   assignOperation(OC_ZERO, "zero", rType, &opCodeZeroOperation);
113   /* To stop the MIPS machine */
114   assignOperation(OC_STOP,"stop", specialType, &stopOperation);
115
116   assignOperation(OC_ADDI, "addi", iType, &mips_addi);
117   assignOperation(OC_LUI, "lui", iType ,&mips_lui);
118   assignOperation(OC_LW, "lw", iType, &mips_lw);
119   assignOperation(OC_ORI, "ori", iType, &mips_ori);
120   assignOperation(OC_SW, "sw", iType, &mips_sw);
121
122   /* Initialize operations with OpCode = 0 and corresponding functions */
123   for (i=0; i<FUNCTION_COUNT; ++i) {
124     assignFunction(i, "ndef", &undefinedFunction);
125   }
126   assignFunction(FC_ADD, "add", &mips_add);
127   assignFunction(FC_DIV, "div", &mips_div);
128   assignFunction(FC_MULT, "mult", &mips_mult);
129   assignFunction(FC_SUB, "sub", &mips_sub);
130
131       initializeMemory();
132
133   /* Initialize registers */
134   for (i=0; i<REGISTER_COUNT; ++i) {
135     registers[i]= 0;
136   }
137
138   /* Stack pointer */
139   SP=65535;
140
141   /* Initialize program counter */
142   pc = 0;
143
144   /* Yes, we want the machine to run */
145   doRun = TRUE;
146
147 }
148
```

```
149 /* Fetch and execute */
150 void run() {
151   while (doRun) {
152     /* Fetch Instruction*/
153     word w  = loadWord(pc);
154     Instruction *instruction = (Instruction *) &w;
155     pc += 4;
156     /* Execute Instruction*/
157     operations[instruction->i.opcode].operation(instruction);
158     /* In case you want to watch the machine */
159                 if (verbose) {
160                     printInstruction(instruction);
161                 }
162   }
163 }
164
165
166 /* ================================================================== */
167 /* "Special" operations --- only for "internal" usage */
168
169 /* To deal with "undefined" behaviour */
170 void undefinedOperation(Instruction *instruction) {
171     ERROR("Unknown opcode: %x", instruction->i.opcode);
172 }
173
174 /* To deal with "undefined" behaviour */
175 void undefinedFunction(Instruction *instruction) {
176     ERROR("Unknown funct: %x", instruction->r.funct);
177 }
178
179 /* To deal with operations with opcode = 0 */
180 void opCodeZeroOperation(Instruction *instruction) {
181   functions[instruction->r.funct].function(instruction);
182 }
183
184 /* To stop the machine */
185 void stopOperation(Instruction *instruction) {
186     doRun = FALSE;
187 }
188
189 /* ================================================================================
       */
190
191 /* ADD */
192 void mips_add(Instruction *instruction) {
193   InstructionTypeR r = instruction->r;
194   registers[r.rd] = (signed)registers[r.rs] + (signed)registers[r.rt];
195 }
196
197 /* ADDI */
198 void mips_addi(Instruction *instruction) {
199   InstructionTypeI i = instruction->i;
200   registers[i.rt] = (signed)registers[i.rs] + (signed)signExtend(i.immediate)
        ;
```

```
201 }
202
203 /* DIV --- Warning: actual DIV is a bit more complicated*/
204 void mips_div(Instruction *instruction) {
205   InstructionTypeR r = instruction->r;
206   registers[r.rd] = (signed)registers[r.rs] / (signed)registers[r.rt];
207 }
208
209 /* LUI */
210 void mips_lui(Instruction *instruction) {
211   InstructionTypeI i = instruction->i;
212   registers[i.rt] = i.immediate << 16;
213 }
214
215 /* LW */
216 void mips_lw(Instruction *instruction) {
217   InstructionTypeI i = instruction->i;
218   registers[i.rt] = loadWord(registers[i.rs] + (signed)signExtend(i.immediate
        ));
219 }
220
221 /* MULT --- Warning: actual MULT is a bit more complicated */
222 void mips_mult(Instruction *instruction) {
223   InstructionTypeR r = instruction->r;
224   registers[r.rd] = (signed)registers[r.rs] * (signed)registers[r.rt];
225 }
226
227 /* ORI */
228 void mips_ori(Instruction *instruction) {
229   InstructionTypeI i = instruction->i;
230   registers[i.rt] = registers[i.rs] | zeroExtend(i.immediate);
231 }
232
233 /* SUB */
234 void mips_sub(Instruction *instruction) {
235   InstructionTypeR r = instruction->r;
236   registers[r.rd] = (signed)registers[r.rs] - (signed)registers[r.rt];
237 }
238
239 /* SW */
240 void mips_sw(Instruction *instruction) {
241   InstructionTypeI i = instruction->i;
242   storeWord(registers[i.rt], registers[i.rs] + (signed)signExtend(i.immediate
        ));
243 }
```

Listing 4: memory.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 <time.h>
#include "memory.h"

MemoryRegion * memoryRegions[MEMORY_REGION_SIZE];
FILE * outputStream;

void emptyMemoryRegionInitializer() {}

/* ============================================================================
    */
/* DEFAULT MEMORY */
byte defaultMemoryData[RAM_SIZE];

void defaultMemoryInitializer() {
  int i;
  for (i=0; i<RAM_SIZE; ++i) {
    defaultMemoryData[i] = 0;
  }
}

void defaultMemoryWriteHandler(word address, byte value) {
  defaultMemoryData[address] = value;
}

byte defaultMemoryReadHandler(word address) {
  return defaultMemoryData[address];
}

MemoryRegion defaultMemory = {
    0,
    RAM_SIZE-1,
    &defaultMemoryInitializer,
    &defaultMemoryWriteHandler,
    &defaultMemoryReadHandler
};

/* NEW Macro */

/* ============================================================================
    */
/* printf memory */
```

```
51  void fprintfMemoryRegionInitializer() {
52    outputStream = stdout;
53  }
54
55  void fprintfMemoryWriteHandler(word address, byte value) {
56          fprintf(outputStream, "%c", value);
57  }
58
59  byte fprintfMemoryReadHandler(word address) {
60          /* ignore reads, only used to output */
61    return 0;
62  }
63
64  MemoryRegion fprintfMemory = {
65      FPRINTF_MEMORY_LOCATION,
66      FPRINTF_MEMORY_LOCATION,
67      &fprintfMemoryRegionInitializer,
68      &fprintfMemoryWriteHandler,
69      &fprintfMemoryReadHandler
70  };
71
72  /* ============================================================================
        */
73  /* time memory */
74
75  void timeMemorWriteHandler(word address, byte value) {}
76
77  byte timeMemoryReadHandler(word address) {
78    return clock() * 1000 / CLOCKS_PER_SEC;
79  }
80
81  MemoryRegion timeMemory = {
82      TIME_MEMORY_LOCATION,
83      TIME_MEMORY_LOCATION,
84      &emptyMemoryRegionInitializer,
85      &timeMemorWriteHandler,
86      &timeMemoryReadHandler
87  };
88
89  /* ============================================================================
        */
90
91  int checkMemoryRegionOverlap(MemoryRegion *region) {
92      int i;
93    for (i=0; i<MEMORY_REGION_SIZE; ++i) {
94          if (!memoryRegions[i]) {
95              break;
96          }
97      if (((memoryRegions[i]->min >= region->min) && ( region->min <=
          memoryRegions[i]->max))
98              || ((memoryRegions[i]->min >= region->max) && (region->max <=
                  memoryRegions[i]->max))) {
99        return FALSE;
100     }
```

```
101     }
102       return TRUE;
103   }
104
105   int memoryRegionCounter = 0;
106
107   void registerMemoryRegion(MemoryRegion *region) {
108       if (checkMemoryRegionOverlap(region)) {
109           memoryRegions[memoryRegionCounter] = region;
110           memoryRegionCounter++;
111           region->initialize();
112       } else {
113           ERROR("Overlapping memory regions");
114       }
115   }
116
117   void initializeMemory() {
118       int i;
119       memoryRegionCounter = 0;
120       for (i=0; i<MEMORY_REGION_SIZE; ++i) {
121           memoryRegions[i] = NULL;
122       }
123       registerMemoryRegion(&defaultMemory);
124       registerMemoryRegion(&fprintfMemory);
125       registerMemoryRegion(&timeMemory);
126   }
127
128   /* MEMORY FUNCTIONS ============================================================
        */
129
130   /* Load a file to memory */
131   void loadFile(char* filename) {
132       /* TODO: Task (d) implement loadFile */
133       int i;
134       FILE *file;
135       char *buffer = 0;
136       long length;
137       byte* ram;
138
139       // Reading the file in the buffer
140       file = fopen(filename, "rb");
141       if (file) {
142           fseek (file, 0, SEEK_END);
143           length = ftell (file);
144           fseek (file, 0, SEEK_SET);
145           buffer = malloc (length);
146           if (buffer)
147               fread (buffer, 1, length, file);
148       }
149       fclose(file);
150
151       // Create store characters in the memory
152       if (buffer) {
153           ram = defaultMemoryData;
```

```
154         for (i = 0; i < length; i++) {
155             *ram = (unsigned) buffer[i];
156             ram += 1;
157         }
158     }
159 }
160
161 /* ============================================================================
        */
162
163 MemoryRegion* getMemoryRegion(word location) {
164     int i;
165     MemoryRegion *memoryRegion = NULL;
166     for (i=0; i<MEMORY_REGION_SIZE && !memoryRegion; ++i) {
167         if (memoryRegions[i] && memoryRegions[i]->min <= location &&
                memoryRegions[i]->max >= location) {
168       memoryRegion = memoryRegions[i];
169   }
170     }
171     if (!memoryRegion) {
172         ERROR("Invalid memory access at 0x%x", (int) location);
173     }
174     return memoryRegion;
175 }
176
177
178 /* Store a word to memory */
179 void storeWord(word w, word location) {
180   MemoryRegion *memoryRegion = getMemoryRegion(location);
181   memoryRegion->write(location,   (w >> (8*3)) & 0xFF);
182   memoryRegion->write(location+1, (w >> (8*2)) & 0xFF);
183   memoryRegion->write(location+2, (w >> (8*1)) & 0xFF);
184   memoryRegion->write(location+3, (w >> (8*0)) & 0xFF);
185 }
186
187 /* Load a word from memory */
188 word loadWord(word location) {
189   MemoryRegion *memoryRegion = getMemoryRegion(location);
190   word w = 0;
191   w += (memoryRegion->read(location)   << (8*3));
192   w += (memoryRegion->read(location+1) << (8*2));
193   w += (memoryRegion->read(location+2) << (8*1));
194   w += (memoryRegion->read(location+3) << (8*0));
195   return w;
196 }
```