```
/*
 * author(s):   Michael Kohler - 11-108-289
 *              Lars SchÃ¼tz - 11-122-348
 * modified:    2011-03-27
 *
 */

#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 = FALSE;

/* 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, uns
igned rs, unsigned opcode) {
        return funct + (shamt << 6) + (rd << 11) + (rt << 16) + (rs <<21) + (opcode
 << 26);
}

/* Extends a 16 bit halfword to a 32 bit word with the value of the most significan
t bit */
word signExtend(halfword value) {
    return (value ^ 0x8000) - 0x8000;
}

/* Extends a 16 bit halfword to a 32 bit word by adding leading zeros */
word zeroExtend(halfword value) {
    return (value | 0x00000000);
}

/* To make some noise */
void printInstruction(Instruction *i) {
    Operation o = operations[i->i.opcode];
    Function f;
    switch (o.type) {
        case iType:
            printf("%-4s %02i=0x%08ux, %02i=0x%08ux, 0x%04x\n", o.name, i->i.rt, re
gisters[i->i.rt], i->i.rs, registers[i->i.rs], i->i.immediate );
            break;
        case jType:
            printf("%-4s 0x%08x\n", o.name, i->j.address);
            break;
        case rType:
            f = functions[i->r.funct];
            printf("%-4s %02i=0x%08ux, %02i=0x%08ux, %02i=0x%08ux, 0x%04x\n", f.nam
e, i->r.rd, registers[i->r.rd], i->r.rs, registers[i->r.rs],i->r.rt, registers[i->r
.rt],i->r.shamt);
            break;
        case specialType:
            printf("%-4s\n", o.name);
            break;
    }
}

/* ================================================================== */
/* Memory operations */

/* Store a word to memory */
void storeWord(word w, word location) {
        memory[location] = (byte)((w >> (8*3)) & 0xFF);
        memory[location+1] = (byte)((w >> (8*2)) & 0xFF);
        memory[location+2] = (byte)((w >> (8*1)) & 0xFF);
        memory[location+3] = (byte)(w & 0xFF);
}

/* Load a word from memory */
word loadWordFrom(word location) {
        word w = 0;
        w += (memory[location]   << (8*3));
        w += (memory[location+1] << (8*2));
        w += (memory[location+2] << (8*1));
        w +=  memory[location+3];
        return w;
}

/* ================================================================== */
/* Initialize and run */
void assignOperation(unsigned short opCode, const char name[OP_NAME_LENGTH+1], Inst
ructionType type, void (*operation)(Instruction*)) {
    strcpy(operations[opCode].name, name);
    operations[opCode].type=type;
    operations[opCode].operation = operation;
}

void assignFunction(unsigned short funct, const char name[FUNC_NAME_LENGTH+1], void
 (*function)(Instruction*)) {
    strcpy(functions[funct].name, name);
    functions[funct].function = function;
}

/* Initialize the "hardware" and operation and function dispatcher */
void initialize() {
        int i;
        /* Initialize operations */
        for (i=0; i<OPERATION_COUNT; ++i) {
                assignOperation(i, "ndef", specialType, &undefinedOperation);
        }
        assignOperation(OC_ZERO, "zero", rType, &opCodeZeroOperation);
        /* To stop the MIPS machine */
        assignOperation(OC_STOP,"stop", specialType, &stopOperation);

        assignOperation(OC_ADDI, "addi", iType, &mips_addi);
        assignOperation(OC_JAL, "jal", jType ,&mips_jal);
        assignOperation(OC_LUI, "lui", iType ,&mips_lui);
        assignOperation(OC_LW, "lw", iType, &mips_lw);
```

```
        assignOperation(OC_ORI, "ori", iType, &mips_ori);
        assignOperation(OC_SW, "sw", iType, &mips_sw);

        /* Initialize operations with OpCode = 0 and corresponding functions */
        for (i=0; i<FUNCTION_COUNT; ++i) {
                assignFunction(i, "ndef", &undefinedFunction);
        }
        assignFunction(FC_ADD, "add", &mips_add);
        assignFunction(FC_SUB, "sub", &mips_sub);

        /* Initialize memory */
        for (i=0; i<MEMORY_SIZE; ++i) {
                memory[i] = 0;
        }

        /* Initialize registers */
        for (i=0; i<REGISTER_COUNT; ++i) {
                registers[i]= 0;
        }

        /* Stack pointer */
        SP=65535;

        /* Initialize program counter */
        pc = 0;

        /* Yes, we want the machine to run */
        doRun = TRUE;
}

/* Fetch and execute */
void run() {
        while (doRun) {
                /* Fetch Instruction*/
                word w  = loadWordFrom(pc);
                Instruction *instruction = (Instruction *) &w;
                /* Please note: the program counter is incremented before the opera
tion is executed */
                pc += 4;
                /* Execute Instruction*/
                operations[instruction->i.opcode].operation(instruction);
                /* In case you want to watch the machine */
                if (verbose) {
                    printInstruction(instruction);
                }
        }
}

/* ====================================================================== */
/* "Special" operations --- only for "internal" usage */

/* To deal with "undefined" behaviour */
void undefinedOperation(Instruction *instruction) {
    printf("%s in %s, line %i: Unknown opcode: %x\n",__func__, __FILE__, __LINE__,
instruction->i.opcode);
    exit(0);
}

/* To deal with "undefined" behaviour */
void undefinedFunction(Instruction *instruction) {
    printf("%s in %s, line %i: Unknown funct: %x\n",__func__, __FILE__, __LINE__, i
nstruction->r.funct);
    exit(0);
}

/* To deal with operations with opcode = 0 */
void opCodeZeroOperation(Instruction *instruction) {
        functions[instruction->r.funct].function(instruction);
}

/* To stop the machine */
void stopOperation(Instruction *instruction) {
    doRun = FALSE;
}

/* ====================================================================== */
/* Implemented MIPS operations */

/* ADD */
void mips_add(Instruction *instruction) {
        InstructionTypeR instrTypeR = instruction->r;
        registers[instrTypeR.rd] = (signed)registers[instrTypeR.rs] +
                                        (signed)registers[instrTypeR.rt];
}

/* ADDI */
void mips_addi(Instruction *instruction) {
        InstructionTypeI instrTypeI = instruction->i;
    registers[instrTypeI.rt] = (signed)registers[instrTypeI.rs] +
                                    (signed)signExtend(instrTypeI.immediate);
}

/* JAL */
void mips_jal(Instruction *instruction) {
        InstructionTypeJ instrTypeJ = instruction->j;
  word w = pc;
  registers[I_RA] = (signed)(w);
  pc = (pc & 0xF0000000) | (instrTypeJ.address << 2);
}

/* LUI */
void mips_lui(Instruction *instruction) {
        InstructionTypeI instrTypeI = instruction->i;
  registers[instrTypeI.rt] = (instrTypeI.immediate << 16);
}

/* LW */
void mips_lw(Instruction *instruction) {
        InstructionTypeI i = instruction->i;
        registers[i.rt] = loadWordFrom(registers[i.rs] + (signed)signExtend(i.immed
iate));
}

/* ORI */
void mips_ori(Instruction *instruction) {
        InstructionTypeI i = instruction->i;
        registers[i.rt] = registers[i.rs] | zeroExtend(i.immediate);
}

/* SUB */
void mips_sub(Instruction *instruction) {
        InstructionTypeR r = instruction->r;
        registers[r.rd] = (signed)registers[r.rs] - (signed)registers[r.rt];
}

/* SW */
void mips_sw(Instruction *instruction) {
        InstructionTypeI instrTypeI = instruction ->i;
    storeWord(registers[instrTypeI.rt], registers[instrTypeI.rs] +
                                        (signed)signExtend(instrTypeI.immediate));
}
```

```
/*
 *
 * author(s):   Michael Kohler - 11-108-289
 *              Lars SchÃ¼tz - 11-122-348
 * modified:    2011-03-27
 *
 */

#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));
        assert(T0 == -1);
        test_execute(create_itype_hex(1, I_T0, I_T0, OC_ADDI));
        assert(T0 ==  0);

        test_execute(create_itype_hex(0xFFFF, I_T0, I_ZERO, OC_ADDI));
        assert(T0 == -1);
        test_execute(create_itype_hex(0xFFFF, I_T0, I_T0, OC_ADDI));
        assert(T0 == -2);

        test_execute(create_itype_hex(3, I_T0, I_ZERO, OC_ADDI));
        assert(T0 ==  3);
        test_execute(create_itype_hex(1, I_T1, I_T0, OC_ADDI));
        assert(T0 ==  3);
        assert(T1 ==  4);
}

/* JAL */
void test_jal() {
        int pcSaved;
        word w;
        Instruction* instruction;

        pc = 0x00000000;
        pcSaved = pc;
        test_execute(create_jtype_hex(0x0001, OC_JAL));
        assert(RA == pcSaved + 4);
        assert(pc == 4);

        /* The following test is executed manually as the desired pc is outside the
 memory,
         * i.e. the test needs to bypass actually storing the instruction in the me
mory.
         */
        initialize();
        pc = 0xAF000000;
        pcSaved = pc;
        w = create_jtype_hex(0x0001, OC_JAL);
        instruction = (Instruction *) &w;
        pc += 4;
        operations[instruction->i.opcode].operation(instruction);
        assert(RA == pcSaved + 4);
        assert(pc == 0xA0000004);
}

/* LUI */
void test_lui() {
    test_execute(create_itype_hex(0xFFFF, I_T0, I_ZERO, OC_LUI));
    assert(T0 == 0xFFFF0000);

    test_execute(create_itype_hex(0x0001, I_T0, I_ZERO, OC_LUI));
    assert(T0 == 0x00010000);
}

/* LW */
void test_lw() {
    word location1 = 0x000000010;
    word location2 = 0x000000014;

    word w = 0xFFF0;
    storeWord(w, location1);
    T1 = location1;
    test_execute(create_itype_hex(0x0000, I_T0, I_T1, OC_LW));
    assert(T0==w);

    w = 0x0010;
    T1 = location2;
    storeWord(w, location2);
    test_execute(create_itype_hex(0x0000, I_T0, I_T1, OC_LW));
    assert(T0==w);
}

/* ORI */
void test_ori() {
    test_execute(create_itype_hex(0xFFFF, I_T0, I_T1, OC_ORI));
    assert((T0 == 0xFFFF) | (T0 == T1));

    test_execute(create_itype_hex(0x00FF, I_T0, I_T1, OC_ORI));
    assert((T0 == 0xFF) | (T0 == T1));
}

/* SUB */
void test_sub() {
```

```
    T1 = 4;
    T2 = 5;
    test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
    assert(T0==1);

    T1 = 7;
    T2 = 3;
    test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
    assert(T0==-4);

    T1 = 4;
    T2 = -1;
    test_execute(create_rtype_hex(FC_SUB, 0x0000, I_T0, I_T1, I_T2, OC_SUB));
    assert(T0==-5);
}

/* SW */
void test_sw() {
    word location1 = 0x00001000;
    word location2 = 0x00001004;

    word w = 0xFFFFFFFF;
    T0 = w;
    T1 = location1;
    test_execute(create_itype_hex(0x0000, I_T0, I_T1, OC_SW));
    assert(loadWordFrom(location1) == w);

    w =0x12345678;
    T0 = w;
    T1 = location2;
    test_execute(create_itype_hex(0xFFFC, I_T0, I_T1, OC_SW));
    assert(loadWordFrom(location1) == w);
}

/* ============================================================================ */
/* make sure you've got a "fresh" environment for every test */
void execute_test(void (*test)(void)) {
    initialize();
    test();
}

/* executes all tests */
int main (int argc, const char * argv[]) {
        execute_test(&test_add);
        execute_test(&test_addi);
        execute_test(&test_jal);
        execute_test(&test_lui);
        execute_test(&test_lw);
        execute_test(&test_ori);
        execute_test(&test_sub);
        execute_test(&test_sw);
        return 0;
}
```