```
// File: disassembler.c
// Student: Austin J. Alexander
// Assignment: Project 1
// Course: MET CS472 (FALL 2014)

#include <stdio.h>
#include <string.h>

int main()
{
  // the first instruction begins at address 0x0007A060;
  // the rest follow in 4 byte intervals;
  // this program will output the address and assembly instruction translation
  // (e.g., 0x0007A060  lw $10, 12($20))

  // rs: first register source operand
  // rt: second register source operand
  // rd: register destination operand
  // shamt: shift amount

  unsigned int memory_address = 0x0007A060; // will be incremented

  unsigned int instructions[] = { 0x022DA822,
                                  0x8EF30018,
                                  0x12A70004,
                                  0x02689820,
                                  0xAD930018,
                                  0x02697824,
                                  0xAD8FFFF4,
                                  0x018C6020,
                                  0x02A4A825,
                                  0x158FFFF6,
                                  0x8E59FFF0 };

  // loop through instructions
  for (int instr_index = 0; instr_index < 11; instr_index++) {

    // display machine instruction in hex
    printf("\n<--- Machine instruction (in hex): 0x%x --->\n\n",
            instructions[instr_index]);

                        /////// R & I FORMATS ///////
```

```
    /******* ADDRESS *******/

    printf("Memory Address: 0x%x\n", memory_address);


    /******* OPCODE [6 bits] *******/

    // properly positioned mask for opcode
    // |1111 11|00 0000 0000 0000 0000 0000 0000
    unsigned int opcode_mask = 0xFC000000;
    // proper shift amount for opcode
    int opcode_right_shift = 26;
    // (logical AND) then perform bit shift to the right
    unsigned int opcode = (instructions[instr_index] & opcode_mask)
                            >> opcode_right_shift;

    // display opcode value in hex
    printf("Opcode hex value: 0x%x\n", opcode);
    // display instruction type
    char opcode_type = ' ';
    if (opcode == 0) {
      opcode_type = 'R';
    }
    else {
      opcode_type = 'I';
    }
    printf("Instruction type: %c Format\n", opcode_type);


    /******* FIRST REGISTER (source) [5 bits] *******/

    // properly positioned mask for first register
    // 0000 00|11 111|0 0000 0000 0000 0000 0000
    unsigned int rs_mask = 0x03E00000;
    // proper shift amount for first register
    int rs_right_shift = 21;
    // (logical AND) then perform bit shift to the right
    unsigned int rs = (instructions[instr_index] & rs_mask)
                                >> rs_right_shift;
    // display first register value in hex
    printf("First register (source) hex value: 0x%x\n", rs);
```

```
/******* SECOND REGISTER (R: source | I: source/dest) [5 bits] *******/

// properly positioned mask for second register
// 0000 0000 000|1 1111| 0000 0000 0000 0000
unsigned int rt_mask = 0x001F0000;
// proper shift amount for second register
int rt_right_shift = 16;
// (logical AND) then perform bit shift to the right
unsigned int rt = (instructions[instr_index] & rt_mask)
                            >> rt_right_shift;
// based on opcode, follow R or I format,
// and display second register value in hex
char rt_type[20];
if (opcode == 0) {
  strcpy(rt_type, "source");
}
else {
  strcpy(rt_type, "source/dest");
}
printf("Second register (%s) hex value: 0x%x\n", rt_type, rt);


// based on opcode, follow R or I format
if (opcode == 0) {
                     /////// R FORMAT ///////

  /******* THIRD REGISTER (destination) [5 bits] *******/

  // properly positioned mask for third register
  // 0000 0000 0000 0000 |1111 1|000 0000 0000
  unsigned int rd_mask = 0x0000F800;
  // proper shift amount for third register
  int rd_right_shift = 11;
  // (logical AND) then perform bit shift to the right
  unsigned int rd = (instructions[instr_index] & rd_mask)
                              >> rd_right_shift;
  // display third register value in hex
  printf("Third register (destination) hex value: 0x%x\n", rd);
```

```
    // ignore shamt


    /******* FUNCTION [6 bits] *******/

    // properly positioned mask for function
    // 0000 0000 0000 0000 0000 0000 00|11 1111|
    unsigned int funct_mask = 0x0000003F;
    // logical AND (no need to shift)
    unsigned int funct = (instructions[instr_index] & funct_mask);
    // display function value in hex
    printf("Function value: 0x%x\n", funct);

    // |R| add: 0x00000020 |R| sub: 0x00000022
    // |R| and: 0x00000024 |R|  or: 0x00000025
    // |R| slt: 0x0000002A

    char funct_name[30];
    switch (funct) {
      case 0x00000020:
        strcpy(funct_name, "add");
        break;
      case 0x00000022:
        strcpy(funct_name, "sub");
        break;
      case 0x00000024:
        strcpy(funct_name, "and");
        break;
      case 0x00000025:
        strcpy(funct_name, "or");
        break;
      case 0x0000002A:
        strcpy(funct_name, "slt");
        break;
      default:
        strcpy(funct_name, "ERROR! Something went wrong!");
        break;
    }

    printf("Function name: %s\n", funct_name);
```

```
      /******* FINAL OUTPUT *******/

      printf("\n---> Assembly instruction: ");
      printf("0x%x: %s $%d, $%d, $%d <---\n\n",
              memory_address, funct_name, rd, rs, rt);

    } // end: if (opcode == 0)
    else {
                        /////// I FORMAT ///////

      /******* CONSTANT/OFFSET [16 bits] *******/

      // properly positioned mask for constant/offset
      // 0000 0000 0000 0000 |1111 1111 1111 1111|
      unsigned int constant_offset_mask = 0x0000FFFF;
      // logical AND (no need to shift, but do need a short for negative
values)
      short constant_offset = (instructions[instr_index] &
constant_offset_mask);
      // display constant/offset value in hex
      printf("Constant/Offset hex value: 0x%x\n", constant_offset);


      /******* OPERATION NAME (OPCODE) [6 bits] *******/

      // |I|  lw: 0x00000023 |I|  sw: 0x0000002B
      // |I| beq: 0x00000004 |I| bne: 0x00000005

      char operation_name[30];
      switch (opcode) {
        case 0x00000023:
          strcpy(operation_name, "lw");
          break;
        case 0x0000002B:
          strcpy(operation_name, "sw");
          break;
        case 0x00000004:
          strcpy(operation_name, "beq");
          break;
        case 0x00000005:
          strcpy(operation_name, "bne");
          break;
```

```
        default:
          strcpy(operation_name, "ERROR! Something went wrong!");
          break;
      }

      printf("Operation name: %s\n", operation_name);


      /******* FINAL OUTPUT *******/

      printf("\n---> Assembly instruction: ");
      // if a branch:
      // program counter (PC) / instruction pointer (IP) would be pointed
      // at next instruction
      int instruction_pointer = (memory_address + 4);
      // branch_to instruction is based on IP,
      // with the constant/offset shifted right by two bits,
      // so shift left the constant/offeset to recover actual byte offset
      int branch_to = (instruction_pointer + (constant_offset << 2));
      if (opcode == 0x00000004 || opcode == 0x00000005) {
        printf("0x%x: %s $%d,$%d (branch to: 0x%x) <---\n\n",
                memory_address, operation_name, rs,
                rt, branch_to);
      }
      else {
        printf("0x%x: %s $%d,%d($%d) <---\n\n",
                memory_address, operation_name, rt,
                constant_offset, rs);
      }

    } // end: else (i.e., opcode != 0)


    // increment memory address before next loop
    memory_address += 4;

  } // end: for (int instr_index = 0; instr_index < 11; instr_index++)

  return 0;
}
```