```c
#include "spimcore.h"

/* ALU */
/* 10 Points */
void ALU(unsigned A, unsigned B, char ALUControl, unsigned *ALUresult, char *Zero)
{
	switch (ALUControl)
	{
	case 0:	//add
		*ALUresult = (signed)A + (signed)B;
		break;
	case 1:	//and
		*ALUresult = (A & B);	//Check if this needs to be signed...
		break;
	case 2:	//or
		*ALUresult = (A | B);	//Check if this needs to be signed...
		break;
	case 3:	//slt
		if ((signed)A < (signed)B)
		{
			*ALUresult = 1;
		}
		else
		{
			*ALUresult = 0;
		}
		break;
	case 4:	//sll
		*ALUresult = B << 16;
		break;
	case 5:	//sltu
		if (A < B)
		{
			*ALUresult = 1;
		}
		else
		{
			*ALUresult = 0;
		}
		break;
	case 6:	//sub
		*ALUresult = (signed)A - (signed)B;
		break;
	case 7:	//subu
		*ALUresult = A - B;
		break;
	default:
		//Maybe include error message here...
		break;
	}

	if (*ALUresult == 0)	//Set Zero from switch case results...
	{
		*Zero = 1;
	}
	else
	{
		*Zero = 0;
```

```
        }
}

/* instruction fetch */
/* 10 Points */
int instruction_fetch(unsigned PC, unsigned *Mem, unsigned *instruction)
{
        if (PC > 65535 | PC % 4 != 0)    //Bounds found in spimcore.c (line 3)...
        {
                return 1;    //Returns a flag if PC is out of bounds...
        }
        else    //Might have to get rid of else statement and just leave what's inside as-
is...
        {
                *instruction = Mem[PC >> 2];    //Fetch instructions...
                return 0;
        }
}

/* instruction partition */
/* 10 Points */
void instruction_partition(unsigned instruction, unsigned *op, unsigned *r1, unsigned
*r2, unsigned *r3, unsigned *funct, unsigned *offset, unsigned *jsec)
{
        //jsec == jump target, or just "j" in MIPS reference...

        /*
        Hexadecimal conversions (not necessary for program):
        0x0000003F == 0011 1111
        0x0000001F == 0001 1111
        0x0000FFFF == 1111 1111 1111 1111
        0x03FFFFFF == 0011 1111 1111 1111 1111 1111 1111
        */

        *op = (instruction & 0xFC000000) >> 26;    //Assigns opcode (6 bits)...
        *r1 = (instruction & 0x03E00000) >> 21;    //Assigns register 1 (5 bits)...
        *r2 = (instruction & 0x001F0000) >> 16;    //Assigns register 2 (5 bits)...
        *r3 = (instruction & 0x0000F800) >> 11;    //Assigns register 3 (5 bits)...
        *funct = instruction & 0x0000003F;    //Assigns function (6 bits)...
        *offset = instruction & 0x0000FFFF;    //Assigns offset (16 bits)...
        *jsec = instruction & 0x03FFFFFF;    //Assigns jump (26 bits)...
}

/* instruction decode */
/* 15 Points */
int instruction_decode(unsigned op, struct_controls *controls)
{
        int flag = 0;    //For unused opcode...

        switch (op)    //The "->" operator is used instead of "." (example:
controls.RegDst) because we are using pointers...
                                //controls struct found in spimcore.h (lines 7-18), and
following switch cases are opcodes in hexadecimal format...
                                //opcodes reference can be found under "opcodes" at
https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats...
        {
        case 0x00:    //R-types, except for mfc0...
                controls->RegDst = 1;
                controls->Jump = 0;
                controls->Branch = 0;
                controls->MemRead = 0;
                controls->MemtoReg = 0;
                controls->ALUOp = 8;
                controls->MemWrite = 0;
                controls->ALUSrc = 0;
                controls->RegWrite = 1;
                break;
        case 0x08:    //addi, I-type...
                controls->RegDst = 0;
                controls->Jump = 0;
                controls->Branch = 0;
                controls->MemRead = 0;
                controls->MemtoReg = 0;
                controls->ALUOp = 0;
                controls->MemWrite = 0;
                controls->ALUSrc = 1;
                controls->RegWrite = 1;
                break;
        case 0x0C:    //andi, I-type...
                controls->RegDst = 1;
                controls->Jump = 0;
                controls->Branch = 0;
                controls->MemRead = 0;
                controls->MemtoReg = 0;
                controls->ALUOp = 1;
                controls->MemWrite = 0;
                controls->ALUSrc = 1;
                controls->RegWrite = 1;
                break;
        case 0x04:    //beq, I-type...
                controls->RegDst = 2;
                controls->Jump = 0;
                controls->Branch = 1;
                controls->MemRead = 0;
                controls->MemtoReg = 2;
                controls->ALUOp = 6;
                controls->MemWrite = 0;
                controls->ALUSrc = 0;
                controls->RegWrite = 0;
        case 0x02:    //j, I-type...
                controls->RegDst = 0;
                controls->Jump = 1;
                controls->Branch = 0;
                controls->MemRead = 0;
                controls->MemtoReg = 0;
                controls->ALUOp = 0;
                controls->MemWrite = 0;
                controls->ALUSrc = 0;
                controls->RegWrite = 0;
                break;
        case 0x0F:    //lui, I-type...
                controls->RegDst = 1;
                controls->Jump = 0;
                controls->Branch = 0;
                controls->MemRead = 0;
                controls->MemtoReg = 0;
```

```
            controls->ALUOp = 3;
            controls->MemWrite = 0;
            controls->ALUSrc = 1;
            controls->RegWrite = 1;
            break;
    case 0x23:    //lw, I-type...
            controls->RegDst = 0;
            controls->Jump = 0;
            controls->Branch = 0;
            controls->MemRead = 1;
            controls->MemtoReg = 1;
            controls->ALUOp = 0;
            controls->MemWrite = 0;
            controls->ALUSrc = 1;
            controls->RegWrite = 1;
            break;
    case 0x0D:    //ori, I-type...
            controls->RegDst = 0;
            controls->Jump = 0;
            controls->Branch = 0;
            controls->MemRead = 0;
            controls->MemtoReg = 0;
            controls->ALUOp = 2;
            controls->MemWrite = 0;
            controls->ALUSrc = 1;
            controls->RegWrite = 1;
            break;
    case 0x0A:    //slti, I-type...
            controls->RegDst = 0;
            controls->Jump = 0;
            controls->Branch = 0;
            controls->MemRead = 0;
            controls->MemtoReg = 0;
            controls->ALUOp = 4;
            controls->MemWrite = 0;
            controls->ALUSrc = 1;
            controls->RegWrite = 1;
            break;
    case 0x0B:    //sltui, I-type...
            controls->RegDst = 0;
            controls->Jump = 0;
            controls->Branch = 0;
            controls->MemRead = 0;
            controls->MemtoReg = 0;
            controls->ALUOp = 5;
            controls->MemWrite = 0;
            controls->ALUSrc = 1;
            controls->RegWrite = 1;
            break;
    case 0x2B:    //sw, I-type...
            controls->RegDst = 2;
            controls->Jump = 0;
            controls->Branch = 0;
            controls->MemRead = 0;
            controls->MemtoReg = 2;
            controls->ALUOp = 0;
            controls->MemWrite = 1;
            controls->ALUSrc = 1;
            controls->RegWrite = 0;
            break;
        default:
            //Maybe insert error message here...
            flag = 1;    //Detects unused opcode to trigger halt...
            break;
        }
        return flag;    //Determines if switch case was default due to unused opcode,
triggering halt if flag == 1...
}

/* Read Register */
/* 5 Points */
void read_register(unsigned r1, unsigned r2, unsigned *Reg, unsigned *data1, unsigned
*data2)
{
        //Reg is an array found in spimcore.c (line 12)...
        *data1 = Reg[r1];
        *data2 = Reg[r2];
}

/* Sign Extend */
/* 10 Points */
void sign_extend(unsigned offset, unsigned *extended_value)
{
        if (offset & 0x00008000)
        {    //Check if offset is a negative number for sign extension...
            *extended_value = offset | 0xFFFF0000;    //Hexadecimal 0xfff0000 == binary
65,535 which is the PC boundary...
        }
        else
        {
            *extended_value = offset;    //May have to change add & 0x0000ffff...
        }
}

/* ALU operations */
/* 10 Points */
int ALU_operations(unsigned data1, unsigned data2, unsigned extended_value, unsigned
funct, char ALUOp, char ALUSrc, unsigned *ALUresult, char *Zero)
{
        // R-type functions like 0x20 found at reference source listed earlier (line
108)...
        if (ALUSrc == 8)
        {
            switch (funct) {
            case 0x21:    //add
                ALUOp = 0;
                break;
            case 0x22:    //sub
                ALUOp = 6;
                break;
            case 0x23:    //subu
                ALUOp = 7;
                break;
            case 0x24:    //and
                ALUOp = 1;
                break;
```

```
			case 0x25:	//or
				ALUOp = 2;
				break;
			case 0x2A:	//slt
				ALUOp = 4;
				break;
			case 0x2B:	//sltu
				ALUOp = 5;
				break;
			default:
				return 1;
			}
			ALU(data1, data2, ALUOp, ALUresult, Zero);	//Might have to create a
tempVal for ALUOp to be converted to binary...
		}

		// I-type...
		else if (ALUSrc == 1) {
			ALU(data1, extended_value, ALUOp, ALUresult, Zero);
		}
		return 0;
}

/* Read / Write Memory */
/* 10 Points */
int rw_memory(unsigned ALUresult, unsigned data2, char MemWrite, char MemRead, unsigned
*memdata, unsigned *Mem)
{
		if ((MemWrite == 1 || MemRead == 1) && ALUresult % 4 != 0) //PC needs to be
divisible by 4...
		{
			//Maybe insert error or halt message...
			return 1;
		}
		if (MemWrite == 1)
		{
			Mem[ALUresult >> 2] = data2;
		}
		if (MemRead == 1)
		{
			*memdata = Mem[ALUresult >> 2];
		}
		return 0;
}

/* Write Register */
/* 10 Points */
void write_register(unsigned r2, unsigned r3, unsigned memdata, unsigned ALUresult, char
RegWrite, char RegDst, char MemtoReg, unsigned *Reg)
{
		if (RegWrite == 1)
		{
			if (MemtoReg == 1)
			{
				if (RegDst == 1)
					Reg[r3] = memdata; //Write memdata to rd
				else
					Reg[r2] = memdata; // Write memdata to rt
		}
		else if (MemtoReg == 0)
		{
			if (RegDst == 1)
			{
				Reg[r3] = ALUresult; // Write ALU results to rd
			}
			else
			{
				Reg[r2] = ALUresult; // Write ALU results to rt
			}
		}
		else
		{
			//Maybe insert error message...
		}
	}
}

/* PC update */
/* 10 Points */
void PC_update(unsigned jsec, unsigned extended_value, char Branch, char Jump, char Zero,
unsigned *PC)
{
	*PC = *PC + 4;    //Start PC update by adding 4 to PC...
	if (Jump == 1)
	{
		*PC = (jsec << 2) | (*PC | 0xF0000000);    //PC gets jump shifted left 2
bits + (PC + 1111 0000 0000 0000 0000 0000 0000 0000)...
	}
	else if (Branch == 1 && Zero)
	{
		*PC += (extended_value << 2);    //PC gets PC + extended_value shifted left
2 bits...
	}
}
```