```
#include "spimcore.h"

/* ALU */
/* 10 Points */
void ALU(unsigned A, unsigned B, char ALUControl, unsigned *ALUresult, char *Zero)
{
  if (ALUControl == 0) { // addition
      // get result
      *ALUresult = A + B;
      // establisth zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 1) { // subtraction
      // get result
      *ALUresult = A - B;

      // establish zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 2) { // set less than
      // be sure to make integers signed
      // get result
      if ((signed)A < (signed)B)
        *ALUresult = 1;
      else
        *ALUresult = 0;

      // establish zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 3) { // set less than unsigned //double check this
      // get result
      if (A < B)
        *ALUresult = 1;
      else
        *ALUresult = 0;

      // establish zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 4) { // AND
      // get result
    if (A <= 1 && B <= 1)
        *ALUresult = 1;
    else
      *ALUresult = 0;
```

```
      // establish Zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 5) { // OR
      // get result
    if (A <= 1 || B <= 1)
      *ALUresult = 1;
    else
      *ALUresult = 0;
      // establish Zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 6) { // shift left 16 bits
      // get result
      *ALUresult = B << 16;
      // establish Zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  } else if (ALUControl == 7) { // NOT
      // get result
      *ALUresult = !A;
      // establish Zero
      if (*ALUresult == 0)
        *Zero = 1;
      else
        *Zero = 0;

  }
}

/* instruction fetch */
/* 10 Points */
int instruction_fetch(unsigned PC, unsigned *Mem, unsigned *instruction) {
  int halt;
  // halt condition: if word alignment is off then assert the need to halt
  if (PC % 4 != 0)
    halt = 1;
  else
    halt = 0;

  // get location
  *instruction = Mem[PC >> 2];

  return halt;
}

/* instruction partition */
/* 10 Points */
void instruction_partition(unsigned instruction, unsigned *op, unsigned *r1,
unsigned *r2, unsigned *r3, unsigned *funct, unsigned *offset, unsigned *jsec) {
```

```
  // seperate portions of instruction

  /*  operation: instruction [31-26]
      register source 1: instruction [25-21]
      register source 2: instruction [20-16]
      register destination: instruction [15-11]
      function: instruction[5-0]
      offset: instruction[15-0]
      jsec: instruction [25-0]*/

  *op = (instruction>>26)&63; // gets 5 bits needed for operation & shifts
  *r1 = (instruction>>21)&31; // gets 5 bits needed for 1st register source &
shifts
  *r2 = (instruction>>16)&31; // gets 5 bits needed for 2nd register source &
shifts
  *r3 = (instruction>>11)&31; // gets 5 bits needed for register destination &
shifts
  *funct = instruction & 63; // gets 16 bits needed for function
  *jsec = instruction & 67108863;  // gets 26 bits needed for jump command
  *offset= instruction&65535;
}

/* instruction decode */
/* 15 Points */
int instruction_decode(unsigned op, struct_controls *controls) {
  if (op == 0) { // R-format instruction
    controls->RegDst = 1;
    controls->Jump = 0;
    controls->Branch = 0;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 7;
    controls->MemWrite = 0;
    controls->ALUSrc = 0;
    controls->RegWrite = 1;
  } else if (op == 35) { // load word: 100011; I-type instruction
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 0;
    controls->MemRead = 1;
    controls->MemtoReg = 1;
    controls->ALUOp = 0; // add
    controls->MemWrite = 0;
    controls->ALUSrc = 1;
    controls->RegWrite = 1;
  } else if (op == 43) { // store word: 101011; I-type instruction
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 0;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 0; // add
    controls->MemWrite = 1;
    controls->ALUSrc = 1;
    controls->RegWrite = 0;
  } else if (op == 8) { // add immediate: 001000; I-type instruction
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 0;
```

```
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 0; // add
    controls->MemWrite = 0;
    controls->ALUSrc = 1;
    controls->RegWrite = 1;
  } else if (op == 4) {   // branch on equal: 000100; I-type instruction
    controls->RegDst = 0; // maybe a dont care value (2)
    controls->Jump = 0;
    controls->Branch = 1;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 8;
    controls->MemWrite = 0;
    controls->ALUSrc = 0;
    controls->RegWrite = 0;
  } else if (op == 10) { // set less than immediate: 001010; R-type instruction
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 1;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 7;
    controls->MemWrite = 0;
    controls->ALUSrc = 1;
    controls->RegWrite = 0;
  } else if (op == 11) { // set less than immediate (unsigned): 001011; R-type
instruction
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 1;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 7;
    controls->MemWrite = 0;
    controls->ALUSrc = 0;
    controls->RegWrite = 0;
  } else if (op == 2) { // jump: 000010; J-type instruction
    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;
  } else if (op == 15) { // load upper immediate: 001111
    controls->RegDst = 0;
    controls->Jump = 0;
    controls->Branch = 0;
    controls->MemRead = 0;
    controls->MemtoReg = 0;
    controls->ALUOp = 6;
    controls->MemWrite = 0;
    controls->ALUSrc = 1;
    controls->RegWrite = 1;
  } else
    return 1; // halt if operation code doesn't match any of the valid options
```

```
  return 0;
}

/* Read Register */
/* 5 Points */
void read_register(unsigned r1, unsigned r2, unsigned *Reg, unsigned *data1,
                   unsigned *data2) {
  // read addressed registers r1 & r2 and write values to data1 & data2
  *data1 = Reg[r1];
  *data2 = Reg[r2];
}

/* Sign Extend */
/* 10 Points */
void sign_extend(unsigned offset, unsigned *extended_value) {//aliza
  unsigned long int mask;
  if((offset>>15)==0){//sign bit is pos
      mask=0x0000FFFF;
      *extended_value = offset & mask;
      //for loop to extend the bits?
      //checks it 16th bit is npos in tht case the other 16 bits will be filled w
0?
  }
  else if((offset>>15)==1){//sign bit is neg
      mask=0xFFFF0000;
      *extended_value=offset|mask;
      //fill with 1?
  }

  //take 16 bit immediate value or offset and make it 32 bit long (fill either w 1
or 0)
}

/* ALU operations */
/* 10 Points */
int ALU_operations(unsigned data1, unsigned data2, unsigned extended_value,
unsigned funct, char ALUOp, char ALUSrc, unsigned *ALUresult, char *Zero) {
    switch(ALUSrc){
      case 1:
        ALU(data1,extended_value, ALUOp, ALUresult, Zero);
        return 0;
      case 0:
        switch(ALUOp){
          case 0:
            return 0;
          case 6:
            ALU(data1, extended_value, ALUOp, ALUresult, Zero);
            return 0;
          case 8:
            ALUOp = 1;
            ALU(data1, data2, ALUOp, ALUresult, Zero);
            return 0;
          case 7:
            switch(funct){
              case 32:
                ALUOp = 0;
                break;
              case 34:
```

```
                ALUOp = 1;
                break;
              case 42:
                ALUOp = 2;
                break;
              case 43:
                ALUOp = 3;
                break;
              case 36:
                ALUOp = 4;
                break;
              case 37:
                ALUOp = 5;
                break;
            }
          ALU(data1, data2, ALUOp, ALUresult, Zero);
          return 0;
        }
      default: return 1;
    }


}

/* Read / Write Memory */
/* 10 Points */
int rw_memory(unsigned ALUresult, unsigned data2, char MemWrite, char MemRead,
              unsigned *memdata, unsigned *Mem) {
    // Ensure the address is word-aligned and within memory bounds
    /*if (ALUresult % 4 != 0 || ALUresult >= 0xFFFF) {
        return 1; // Halt condition
    }*/

    if (MemWrite == 1) {
        Mem[ALUresult / 4] = data2; // Write memory
    }
    else if (MemRead == 1) {
        *memdata = Mem[ALUresult / 4]; // Read memory
    }

    return 0; // No halt condition
}

/* Write Register */
/* 10 Points */
void write_register(unsigned r2, unsigned r3, unsigned memdata, unsigned ALUresult,
char RegWrite, char RegDst, char MemtoReg, unsigned *Reg) {
    //write data to a register adressed by r2 or r3
    if (RegWrite == 1){
      if(MemtoReg == 1)
        if(RegDst == 1)
          Reg[r3] = memdata;
        else
          Reg[r2] = memdata;

      else if(MemtoReg == 0){
        if(RegDst == 1)
          Reg[r3] = ALUresult;
        else
```

```
          Reg[r2] = ALUresult;
      }
      else {
        if(RegDst == 0)
          Reg[r2] = ALUresult;
        else
          Reg[r3] = ALUresult;
      }
    }


}

/* PC update */
/* 10 Points */
void PC_update(unsigned jsec, unsigned extended_value, char Branch, char Jump, char
Zero, unsigned *PC) {//Aliza
  if(Branch == 0 && Jump == 0)
    *PC += 4;

  if(Branch == 1 && Zero == 1 & Jump==0)
    *PC = (extended_value * 4) + (*PC + 4);

  if(Jump == 1 && Branch == 0){
    unsigned jShift = jsec * 4;
    unsigned mask = (*PC + 4) & 0xF000000;
    *PC = mask | jShift;
  }

}
```