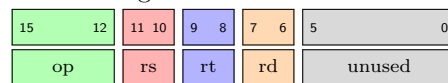


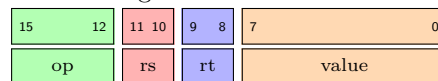
## Instruction Set Architecture

The set of supported instructions currently includes *add*, *sub*, *and*, *or*, *addi*, and *sll*. The following instruction are not currently implemented: *lw*, *sw*, *beq*, and *bne*. Each instruction is 16 bits wide. Instructions currently come in two flavors, R-Type and I-Type instructions.

Register 0.1: R-TYPE



Register 0.2: I-TYPE



In the above diagrams, *op* is the opcode, *rs* is the source register, *rt* is the target/destination register, and *rd* is the destination register. Value is the immediate value in I-Type instructions.

## Source Code

---

```
// @author  Matthew Warren & Christopher Schmitt
// @version 1.23.2020
// @licence MIT (c) @author

/**
 * The half adder performs addition of numbers. It uses a
 * single XOR gate to compute the sum and a single and gate
 * to compute the carry.
 *
 * @param {wire} lhs - The left hand side
 * @param {wire} rhs - The right hand side
 * @return {wire} sum - The sum of lhs + rhs
 * @return {wire} cout - The carry of lhs + rhs
 */
module HalfAdder(
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire cout
);
```

```
// Compute sum
xor(sum, lhs, rhs);

// Comput cout
and(cout, lhs, rhs);

endmodule // HalfAdder

/**
 * The adder combines two half adders to produce a single
 * circuit which accepts three inputs: carry in, lhs, and
 * rhs.
 *
 * @param {wire} cin - The carry in
 * @param {wire} lhs - The left hand side
 * @param {wire} rhs - The right hand side
 * @return {wire} sum - The sum of lhs + rhs
 * @return {wire} cout - The sum of lhs + rhs
 */
module Adder(
    input wire cin,
    input wire lhs,
    input wire rhs,
    output wire sum,
    output wire cout
);

    // Instantiate two half-adders
    HalfAdder halfAdderA(cin, lhs, buffer, carry_left);
    HalfAdder halfAdderB(buffer, rhs, sum, carry_right);

    // Compute cout
    or(cout, carry_left, carry_right);

endmodule // Adder

/**
 * The TwoOneMux selects one out of two input lines to
 * forward to the output line.
 *
 * @param {wire} select - Selects the input to forward
 * @param {wire [1:0]} data - The inputs to select from
 * @return {wire} result - The selected input
 */
```

```
module TwoOneMux(
    input wire select,
    input wire [1:0] data,
    output wire result
);

    // Invert select line
    not(select_, select);

    // Compute products
    and(lhs, data[0], select_);
    and(rhs, data[1], select);

    // Compute sum
    or(result, lhs, rhs);

endmodule // TwoOneMux

/**
 * The FourOneMux cascades several TwoOneMuxs together to
 * create a mux which selects between four inputs.
 *
 * @param {wire [1:0]} select - Selects the input to be forwarded
 * @param {wire [3:0]} data - The inputs to select from
 * @return {wire} result - The selected input
 */
module FourOneMux(
    input wire [1:0] select,
    input wire [3:0] data,
    output wire result
);

    // Instantiate layer one
    TwoOneMux muxA(select[0], {data[0], data[1]}, AB);
    TwoOneMux muxB(select[0], {data[2], data[3]}, BC);

    // Instantiate layer two
    TwoOneMux muxC(select[1], {AB, BC}, result);

endmodule // FourOneMux

/**
 * The OneBitALU performs basic arithmetic and logical
 * operations on all except for the most significant bit.
 */
```

```

*
* @param {wire} lhs - The left hand side of the computation
* @param {wire} rhs - The right hand side of the computation
* @param {wire} cin - The carry in from the computaion of
    previous bits
* @param {wire} less - The less input for the slt operation
* @param {wire} rhs_invert - Inverts the rhs (used for inverse
    operations)
* @param {wire [1:0]} op - The opcode representing the
    operation to perform
* @return {wire} result - The result of the computaion
* @return {wire} cout - The carry out of the computation (if
    applicable)
*/
module OneBitALU(
    input wire lhs,
    input wire rhs,
    input wire cin,
    input wire less,
    input wire rhs_invert,
    input wire [1:0] op,
    output wire result,
    output wire cout
);

    // Handle rhs inversion
    not(rhs_, rhs);
    TwoOneMux inverter(rhs_invert, {rhs_, rhs}, rhs_inverted);

    // Compute lhs & rhs
    and(lhs_and_rhs, lhs, rhs);

    // Compute lhs | rhs
    or(lhs_or_rhs, lhs, rhs);

    // Compute lhs + rhs
    Adder adder(cin, lhs, rhs_inverted, sum, cout);

    // Select operation based on opcode
    FourOneMux selector(op, {lhs_and_rhs, lhs_or_rhs, sum, less},
        result);

endmodule // OneBitALU

/**

```

```

* The OneBitALU_MSB performs basic arithmetic and logical
* operations on the most significant bit.
*
* @param {wire} lhs - The left hand side of the computation
* @param {wire} rhs - The right hand side of the computation
* @param {wire} cin - The carry in from the computaion of
*                   previous bits
* @param {wire} less - The less input for the slt operation
* @param {wire} rhs_invert - Inverts the rhs (used for inverse
*                           operations)
* @param {wire [1:0]} op - The opcode representing the
*                           operation to perform
* @return {wire} result - The result of the computaion
* @return {wire} sum - The sum of the most significant bits of
*                    lhs and rhs
* @return {wire} cout - The carry out of the compuation (if
*                      applicable)
*/
module OneBitALU_MSB(
    input wire lhs,
    input wire rhs,
    input wire cin,
    input wire less,
    input wire rhs_invert,
    input wire [1:0] op,
    output wire result,
    output wire cout,
    output wire sum
);

    // Handle rhs inversion
    not(rhs_, rhs);
    TwoOneMux inverter(rhs_invert, {rhs_, rhs}, rhs_inverted);

    // Compute lhs & rhs
    and(lhs_and_rhs, lhs, rhs);

    // Compute lhs | rhs
    or(lhs_or_rhs, lhs, rhs);

    // Compute lhs + rhs
    Adder adder(cin, lhs, rhs_inverted, sum, cout);

    // Select operation based on opcode
    FourOneMux selector(op, {lhs_and_rhs, lhs_or_rhs, sum, less},
        result);

```

```

endmodule // OneBitALU_MSB

/**
 * The ALU is the heart of the CPU, it performs arithmetic,
 * logical, and comparison operations. This ALU supports
 * add, sub, and, or, addi, and slt instructions. This ALU
 * is built by chaining together 15 OneBitALUs and 1
 * OneBitALU_MSB.
 *
 * @param {wire [15:0]} lhs - The left hand side value
 * @param {wire [15:0]} rhs - The right hand side value
 * @param {wire [2:0]} op - The operation to perform
 * @return {wire [15:0]} result - The result of the computation
 * @return {wire} zero - Whether the result is zero or not
 */
module ALU(
    input wire [15:0] lhs,
    input wire [15:0] rhs,
    input wire [2:0] op,
    output wire [15:0] result,
    output wire zero
);

    // Chain together 15 OneBitALUs
    OneBitALU oneBitALU_a(lhs[0], rhs[0], op[2], set, op[2],
        op[1:0], result[0], cout_a);
    OneBitALU oneBitALU_b(lhs[1], rhs[1], cout_a, 1'b0, op[2],
        op[1:0], result[1], cout_b);
    OneBitALU oneBitALU_c(lhs[2], rhs[2], cout_b, 1'b0, op[2],
        op[1:0], result[2], cout_c);
    OneBitALU oneBitALU_d(lhs[3], rhs[3], cout_c, 1'b0, op[2],
        op[1:0], result[3], cout_d);
    OneBitALU oneBitALU_e(lhs[4], rhs[4], cout_d, 1'b0, op[2],
        op[1:0], result[4], cout_e);
    OneBitALU oneBitALU_f(lhs[5], rhs[5], cout_e, 1'b0, op[2],
        op[1:0], result[5], cout_f);
    OneBitALU oneBitALU_g(lhs[6], rhs[6], cout_f, 1'b0, op[2],
        op[1:0], result[6], cout_g);
    OneBitALU oneBitALU_h(lhs[7], rhs[7], cout_g, 1'b0, op[2],
        op[1:0], result[7], cout_h);
    OneBitALU oneBitALU_i(lhs[8], rhs[8], cout_h, 1'b0, op[2],
        op[1:0], result[8], cout_i);
    OneBitALU oneBitALU_j(lhs[9], rhs[9], cout_i, 1'b0, op[2],
        op[1:0], result[9], cout_j);

```

```

OneBitALU oneBitALU_k(lhs[10], rhs[10], cout_j, 1'b0, op[2],
    op[1:0], result[10], cout_k);
OneBitALU oneBitALU_l(lhs[11], rhs[11], cout_k, 1'b0, op[2],
    op[1:0], result[11], cout_l);
OneBitALU oneBitALU_m(lhs[12], rhs[12], cout_l, 1'b0, op[2],
    op[1:0], result[12], cout_m);
OneBitALU oneBitALU_n(lhs[13], rhs[13], cout_m, 1'b0, op[2],
    op[1:0], result[13], cout_n);
OneBitALU oneBitALU_o(lhs[14], rhs[14], cout_n, 1'b0, op[2],
    op[1:0], result[14], cout_o);

// Chain 1 OneBitALU_MSB
OneBitALU_MSB oneBitALU_MSB(lhs[15], rhs[15], cout_o, 1'b0,
    op[2], op[1:0], result[15], cout_null, set);

// Compute zero
or(a_or_b, result[0], result[1]);
or(c_or_d, result[2], result[3]);
or(e_or_f, result[4], result[5]);
or(g_or_h, result[6], result[7]);
or(i_or_j, result[8], result[9]);
or(k_or_l, result[10], result[11]);
or(m_or_n, result[12], result[13]);
or(o_or_p, result[14], result[15]);

or(ab_or_cd, a_or_b, c_or_d);
or(ef_or_gh, e_or_f, g_or_h);
or(ij_or_kl, i_or_j, k_or_l);
or(mn_or_op, m_or_n, o_or_p);

or(abcd_or_efgh, ab_or_cd, ef_or_gh);
or(ijkl_or_mnop, ij_or_kl, mn_or_op);

nor(zero, abcd_or_efgh, ijkl_or_mnop);

endmodule // ALU

/**
 * The TwoFourDecoder takes in a binary-encoded number and
 * outputs the one-hot decoded value.
 *
 * @param {wire [1:0]} encoded - The encoded value
 * @param {wire [3:0]} decoded - The decoded value
 */
module TwoFourDecoder(

```

```

    input wire [1:0] encoded,
    output wire [3:0] decoded
);

    // Invert encoded
    not(encoded_a_, encoded[0]);
    not(encoded_b_, encoded[1]);

    // Compute products
    and(decoded[0], encoded_b_, encoded_a_);
    and(decoded[1], encoded_b_, encoded[0]);
    and(decoded[2], encoded[1], encoded_a_);
    and(decoded[3], encoded[1], encoded[0]);

endmodule // TwoFourDecoder

/**
 * The DLatch can store up to one bit of information by
 * "capturing" (latching) the input on the data wire.
 *
 * @param {wire} enable - Enable the latch
 * @param {wire} data - The bit to store in the latch
 * @return {wire} Q - The internal state of the latch
 */
module DLatch(
    input wire enable,
    input wire data,
    output wire Q
);

    not(data_, data);
    nand(x, data, enable);
    nand(y, data_, enable);
    nand(Q, x, Q1);
    nand(Q1, y, Q);

endmodule // DLatch

/**
 * The DFlipFlop is a clocked circuit that uses a pair of
 * DLatches to create a flip-flop that always copies the
 * data in the input line.
 *
 * @param {wire} clk - The input clock to sync with

```



```

* @param {wire} data - The bit to copy into internal state
* @return {wire} Q - The internal state of the flip-flop
*/
module DFlipFlop(
    input wire clk,
    input wire data,
    output wire Q
);

    not(clk_, clk);

    DLatch d0(clk, data, y);
    DLatch d1(clk_, y, Q);

endmodule // DFlipFlop

/**
 * Register is a general purpose CPU register with a width
 * of 16 bits. Each register is built out of 16 DFipFlops.
 *
 * @param {wire} clk - The clock signal to sync to
 * @param {wire [15:0]} write - The data to write
 * @return {read [15:0]} read - The state of the register
 */
module Register(
    input wire clk,
    input wire [15:0] write,
    output wire [15:0] read
);

    // Instantiate 16 DFlipFlops
    DFlipFlop flipFlop_0(clk, write[0], read[0]);
    DFlipFlop flipFlop_1(clk, write[1], read[1]);
    DFlipFlop flipFlop_2(clk, write[2], read[2]);
    DFlipFlop flipFlop_3(clk, write[3], read[3]);
    DFlipFlop flipFlop_4(clk, write[4], read[4]);
    DFlipFlop flipFlop_5(clk, write[5], read[5]);
    DFlipFlop flipFlop_6(clk, write[6], read[6]);
    DFlipFlop flipFlop_7(clk, write[7], read[7]);
    DFlipFlop flipFlop_8(clk, write[8], read[8]);
    DFlipFlop flipFlop_9(clk, write[9], read[9]);
    DFlipFlop flipFlop_a(clk, write[10], read[10]);
    DFlipFlop flipFlop_b(clk, write[11], read[11]);
    DFlipFlop flipFlop_c(clk, write[12], read[12]);
    DFlipFlop flipFlop_d(clk, write[13], read[13]);

```

```
    DFlipFlop flipFlop_e(clk, write[14], read[14]);
    DFlipFlop flipFlop_f(clk, write[15], read[15]);

endmodule // Register

/**
 * RegisterFile manages the four registers in 16 bit MIPS.
 * The register file also manages reads and writes to each
 * of the registers.
 *
 * @param {wire} clk - The clock signal to sync to
 * @param {wire} write - Enables write mode
 * @param {wire [1:0]} readDestA - The register to read from
 * @param {wire [1:0]} readDestB - The register to read from
 * @param {wire [1:0]} writeDest - The register to write to
 * @param {wire [15:0]} data - The data to write
 * @return {wire [15:0]} readA - The data in the first selected
 *                               register
 * @return {wire [15:0]} readB - The data in the second selected
 *                               register
 */
module RegisterFile(
    input wire clk,
    input wire write,
    input wire [1:0] readDestA,
    input wire [1:0] readDestB,
    input wire [1:0] writeDest,
    input wire [15:0] data,

    output wire [15:0] readA,
    output wire [15:0] readB
);

    wire [15:0] Q_a;
    wire [15:0] Q_b;
    wire [15:0] Q_c;

    // Instantiate registers
    Register register_a(wa_clk, data, Q_a);
    Register register_b(wb_clk, data, Q_b);
    Register register_c(wc_clk, data, Q_c);

    // Handle write inputs
    TwoFourDecoder writeDecoder(writeDest, {wc, wb, wa, w0});
```

```
and(register_clock, write, clk);

and(w0_clk, w0, register_clock);
and(wa_clk, wa, register_clock);
and(wb_clk, wb, register_clock);
and(wc_clk, wc, register_clock);

// Output port a
FourOneMux readMuxA_0(readDestA, {1'b0, Q_a[0], Q_b[0],
    Q_c[0]}, readA[0]);
FourOneMux readMuxA_1(readDestA, {1'b0, Q_a[1], Q_b[1],
    Q_c[1]}, readA[1]);
FourOneMux readMuxA_2(readDestA, {1'b0, Q_a[2], Q_b[2],
    Q_c[2]}, readA[2]);
FourOneMux readMuxA_3(readDestA, {1'b0, Q_a[3], Q_b[3],
    Q_c[3]}, readA[3]);
FourOneMux readMuxA_4(readDestA, {1'b0, Q_a[4], Q_b[4],
    Q_c[4]}, readA[4]);
FourOneMux readMuxA_5(readDestA, {1'b0, Q_a[5], Q_b[5],
    Q_c[5]}, readA[5]);
FourOneMux readMuxA_6(readDestA, {1'b0, Q_a[6], Q_b[6],
    Q_c[6]}, readA[6]);
FourOneMux readMuxA_7(readDestA, {1'b0, Q_a[7], Q_b[7],
    Q_c[7]}, readA[7]);
FourOneMux readMuxA_8(readDestA, {1'b0, Q_a[8], Q_b[8],
    Q_c[8]}, readA[8]);
FourOneMux readMuxA_9(readDestA, {1'b0, Q_a[9], Q_b[9],
    Q_c[9]}, readA[9]);
FourOneMux readMuxA_a(readDestA, {1'b0, Q_a[10], Q_b[10],
    Q_c[10]}, readA[10]);
FourOneMux readMuxA_b(readDestA, {1'b0, Q_a[11], Q_b[11],
    Q_c[11]}, readA[11]);
FourOneMux readMuxA_c(readDestA, {1'b0, Q_a[12], Q_b[12],
    Q_c[12]}, readA[12]);
FourOneMux readMuxA_d(readDestA, {1'b0, Q_a[13], Q_b[13],
    Q_c[13]}, readA[13]);
FourOneMux readMuxA_e(readDestA, {1'b0, Q_a[14], Q_b[14],
    Q_c[14]}, readA[14]);
FourOneMux readMuxA_f(readDestA, {1'b0, Q_a[15], Q_b[15],
    Q_c[15]}, readA[15]);

// Output port b
FourOneMux readMuxB_0(readDestB, {1'b0, Q_a[0], Q_b[0],
    Q_c[0]}, readB[0]);
FourOneMux readMuxB_1(readDestB, {1'b0, Q_a[1], Q_b[1],
    Q_c[1]}, readB[1]);
```

```

FourOneMux readMuxB_2(readDestB, {1'b0, Q_a[2], Q_b[2],
    Q_c[2]}, readB[2]);
FourOneMux readMuxB_3(readDestB, {1'b0, Q_a[3], Q_b[3],
    Q_c[3]}, readB[3]);
FourOneMux readMuxB_4(readDestB, {1'b0, Q_a[4], Q_b[4],
    Q_c[4]}, readB[4]);
FourOneMux readMuxB_5(readDestB, {1'b0, Q_a[5], Q_b[5],
    Q_c[5]}, readB[5]);
FourOneMux readMuxB_6(readDestB, {1'b0, Q_a[6], Q_b[6],
    Q_c[6]}, readB[6]);
FourOneMux readMuxB_7(readDestB, {1'b0, Q_a[7], Q_b[7],
    Q_c[7]}, readB[7]);
FourOneMux readMuxB_8(readDestB, {1'b0, Q_a[8], Q_b[8],
    Q_c[8]}, readB[8]);
FourOneMux readMuxB_9(readDestB, {1'b0, Q_a[9], Q_b[9],
    Q_c[9]}, readB[9]);
FourOneMux readMuxB_a(readDestB, {1'b0, Q_a[10], Q_b[10],
    Q_c[10]}, readB[10]);
FourOneMux readMuxB_b(readDestB, {1'b0, Q_a[11], Q_b[11],
    Q_c[11]}, readB[11]);
FourOneMux readMuxB_c(readDestB, {1'b0, Q_a[12], Q_b[12],
    Q_c[12]}, readB[12]);
FourOneMux readMuxB_d(readDestB, {1'b0, Q_a[13], Q_b[13],
    Q_c[13]}, readB[13]);
FourOneMux readMuxB_e(readDestB, {1'b0, Q_a[14], Q_b[14],
    Q_c[14]}, readB[14]);
FourOneMux readMuxB_f(readDestB, {1'b0, Q_a[15], Q_b[15],
    Q_c[15]}, readB[15]);

endmodule // RegisterFile

/**
 * IMemory stores the CPU instructions. Each memory
 * address is 16 bits wide.
 *
 * @param {wire} clk - The clock signal to sync with
 * @param {wire [15:0]} pc - The program counter
 * @param {wire [15:0]} data - the data to write at an address
 * @param {wire [15:0]} addr - the address to write to
 * @return {wire [15:0]} value - The value at an address
 */
module IMemory(
    input wire clk,
    input wire write,
    input wire [15:0] data,

```

```

    input wire [15:0] addr,
    output wire [15:0] value
);

    reg [15:0] ram [0:65535];

    assign value = ram[addr];

    initial begin
        ram[0] = 16'b0101000100000000; // lw $1, 0x00($0)
        ram[1] = 16'b0101001000000001; // lw $2, 0x01($0)
        ram[2] = 16'b0111011011000000; // slt $3, $1, $2
        ram[3] = 16'b1000110000000010; // beq $3, $0, 0x02
        ram[4] = 16'b0110000100000001; // sw $1, 0x01($0)
        ram[5] = 16'b0110001000000000; // sw $2, 0x00($0)
        ram[6] = 16'b0101000100000000; // lw $1, 0x00($0)
        ram[7] = 16'b0101001000000001; // lw $2, 0x01($0)
        ram[8] = 16'b0001011001000000; // sub $1, $1, $2
    end

endmodule // IMemory

/**
 * DMemory stores the working memory. Each memory
 * address is 16 bits wide.
 *
 * @param {wire} clk - The clock signal to sync with
 * @param {wire [15:0]} pc - The program counter
 * @param {wire [15:0]} data - the data to write at an address
 * @param {wire [15:0]} addr - the address to write to
 * @return {wire [15:0]} value - The value at an address
 */
module DMemory(
    input wire clk,
    input wire write,
    input wire [15:0] data,
    input wire [15:0] addr,
    output wire [15:0] value
);

    reg [15:0] ram [0:65535];

    assign value = ram[addr];

    initial begin

```

```

        ram[0] = 16'h5; // store 0x05 @ 0x00
        ram[1] = 16'h7; // store 0x07 @ 0x01
    end

endmodule // DMemory

/**
 * The MainControl unpacks the opcode and function code
 * into instructions for the CPU's internal components.
 */
module MainControl(
    input wire [3:0] opcode,
    output reg [7:0] control
);

    // Control bits: RegDst, ALUSrc, MemtoReg, RegWrite, MemWrite,
    // Branch, ALUOp
    always @(opcode) case (opcode)
        4'b0000: control <= 8'b10010000; // add
        4'b0001: control <= 8'b10010001; // sub
        4'b0010: control <= 8'b10010010; // and
        4'b0011: control <= 8'b10010010; // or
        4'b0100: control <= 8'b01010000; // addi
        4'b0101: control <= 8'b01110000; // lw
        4'b0110: control <= 8'b01001000; // sw
        4'b0111: control <= 8'b10010010; // slt
        4'b1000: control <= 8'b00000101; // beq
        4'b1001: control <= 8'b00000101; // bne
    endcase
endmodule // MainControl

/**
 * The ALUControl unpacks the opcode and function code into
 * instructions for the CPU's ALU.
 */
module ALUControl(
    input wire [1:0] aluCode,
    input wire [3:0] funCode,
    output reg [2:0] aluCtl
);

    always @(aluCode, funCode) case (aluCode)
        2'b00: aluCtl <= 3'b010; // add
        2'b01: aluCtl <= 3'b110; // subtract
    endcase
endmodule

```

```
        2'b10: case (funCode)
            4'b0000: aluCtl <= 3'b010; // add
            4'b0001: aluCtl <= 3'b110; // sub
            4'b0010: aluCtl <= 3'b000; // and
            4'b0011: aluCtl <= 3'b001; // or
            4'b0111: aluCtl <= 3'b111; // slt
        endcase
    endcase
endmodule // ALUControl

/**
 * CPU brings all of the modules together to create a
 * processor
 *
 * @param {wire} clk - The clock signal to sync with
 * @return {wire [15:0]} readA - The output of the register file
 * @return {wire [15:0]} readB - The output of the register file
 */
module CPU(
    input wire clk,
    output wire [15:0] readA,
    output wire [15:0] readB,
    output wire [15:0] aluResult,
    output reg [15:0] pc
);

    wire [15:0] nextPc;
    wire [15:0] addr;
    wire [15:0] increment;
    wire [15:0] jump;
    wire [15:0] inst;
    wire [15:0] load;
    wire [1:0] readDestA;
    wire [1:0] readDestB;
    wire [1:0] writeDest;
    wire [15:0] data;
    wire [15:0] lhs;
    wire [15:0] rhs;
    wire [1:0] aluOp;
    wire [2:0] aluCtl;
    wire [15:0] extended;
    wire [15:0] offset;

    initial pc = 0;
```

```

assign extended = {{16{inst[7]}}}, inst[7:0]};
assign writeDest = (regDest) ? inst[7:6] : inst[9:8];
assign rhs = (aluSrc) ? extended : readB;
assign nextPc = (branch && zero) ? jump : increment;
assign data = (memToReg) ? load : aluResult;

ALU fetchALU(pc, 16'b1, 3'b010, increment, fetchZero);
ALU mainALU(readA, rhs, aluCtl, aluResult, zero);
ALU branchALU(pc, extended, 3'b010, jump, branchZero);
ALU dataALU(readA, extended, 3'b010, addr, dataZero);

RegisterFile regs(clk, regWrite, inst[11:10], inst[9:8],
    writeDest, data, readA, readB);
MainControl mainControl(inst[15:12], {regDest, aluSrc,
    memToReg, regWrite, memWrite, branch, aluOp});
ALUControl aluControl(aluOp, inst[15:12], aluCtl);
IMemory IMemory(clk, 1'b0, 16'b0, pc, inst);
DMemory DMemory(clk, memWrite, readB, addr, load);

always @(negedge clk) begin
    pc <= nextPc;
end

endmodule // CPU

/**
 * This testbench instantiates a CPU and provides it with a
 * clock. The testbench executes the example assembly.
 */
module TestBench();

    reg clk;

    wire [15:0] pc;
    wire [15:0] readA;
    wire [15:0] readB;
    wire [15:0] aluResult;

    CPU cpu(clk, readA, readB, aluResult, pc);

    always #1 clk = ~clk;

    initial begin
        $display("clk\treadA\t\t\treadB\t\t\tALU_Result\t\t\tpc");
    end

```



```

    $monitor("%b\t%b\t%b\t%b\t%d", clk, readA, readB, aluResult,
             pc);
    clk = 1;
    #16 $finish;
end
endmodule

```

---

## Machine Translation

---

```

0100000100001111 // addi $1, $0, 0x0F # $1 <= 15
0100001000000111 // addi $2, $0, 0x07 # $2 <= 7
0010011011000000 // and $3, $1, $2 # $3 <= 7
0001011110000000 // sub $2, $1, $3 # $2 <= 8
0011101110000000 // or $2, $2, $3 # $2 <= 15
0010101111000000 // add $3, $2, $3 # $3 <= 22
0111111001000000 // slt $1, $3, $2 # $1 <= 0
0111101101000000 // slt $1, $2, $3 # $1 <= 1

```

---

## Output

---

| clk | ReadA            | ReadB            | ALU_Result       | pc |
|-----|------------------|------------------|------------------|----|
| 1   | 0000000000000000 | xxxxxxxxxxxxxxxx | 0000000000001111 | 0  |
| 0   | 0000000000000000 | xxxxxxxxxxxxxxxx | 000000000000111  | 1  |
| 1   | 0000000000000000 | xxxxxxxxxxxxxxxx | 000000000000111  | 1  |
| 0   | 0000000000001111 | 000000000000111  | 000000000000111  | 2  |
| 1   | 0000000000001111 | 000000000000111  | 000000000000111  | 2  |
| 0   | 0000000000001111 | 000000000000111  | 000000000001000  | 3  |
| 1   | 0000000000001111 | 000000000000111  | 000000000001000  | 3  |
| 0   | 0000000000001000 | 000000000000111  | 000000000001111  | 4  |
| 1   | 0000000000001000 | 000000000000111  | 000000000001111  | 4  |
| 0   | 0000000000001111 | 000000000000111  | 000000000000111  | 5  |
| 1   | 0000000000001111 | 000000000000111  | 000000000000111  | 5  |
| 0   | 0000000000000111 | 000000000000111  | 000000000000001  | 6  |
| 1   | 0000000000000111 | 000000000000111  | 000000000000001  | 6  |

---

## Diagrams

