Instruction	Opcode	Format	Description
add	0000	R-Type	Adds \$rs and \$rt together using arithmetic addition, places the sum in \$rt
sub	0001	R-Type	Subtract \$rt from \$rs, places the difference in \$rt
and	0010	R-Type	Perform bitwise ANDing on \$rs and \$rt, place the result in \$rd
or	0011	R-Type	Perform bitwise ORing on \$rs and \$rt, place the result in \$rd
addi	0100	I-Type	Add \$rs to immediate value, place the result in \$rt
slt	0111	R-Type	Put 0x01 in \$rd if \$rs < \$rt, 0x00 otherwise

Instruction Set Architecture

We implement a 3-satge MIPS data-path which implements all R-Type instructions and ADDI.

Register 0.1: R-Type

15 12	11 10	9 8	7 6	5 0
op	rs	rt	rd	unused

Register 0.2: I-Type

15 12	11 10	9 8	7 0
op	rs	rt	value

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.

Control Table

Source Code

```
// @author   Matthew Warren & Christopher Schmitt
// @version  5.10.2020
// @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.
 *
```

Operation	RegDst	ALUSrc	MemToReg	RegWrite	MemWrite	Branch	ALUOp
add	1	0	0	1	0	0	00
sub	1	0	0	1	0	0	01
and	1	0	0	1	0	0	10
or	1	0	0	1	0	0	10
addi	0	1	0	1	0	0	00
lw	0	1	1	1	0	0	00
sw	0	0	0	0	1	0	00
slt	1	0	0	1	0	0	10
beq	0	0	0	0	0	1	01
bne	0	0	0	0	0	1	01

```
 * @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 compuation (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 - Weather 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
 * ouputs 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 circit that uses a pair of
 * DLatchs 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


/**
 * MuxBus behaves as a two by one mux, but has
 * busses as inputs.
 *
 * @param {wire} select - selects the input to be forwarded
 * @param {wire [15:0]} - left hand bus
 * @param {wire [15:0]} - right hand bus
 * @return {wire [15:0]} - selected output bus
 */
module MuxBus(
  input wire select,
  input wire [15:0] lhs,
```

```
  input wire [15:0] rhs,
  output wire [15:0] selected
);

  TwoOneMux mux_a(select, { lhs[0], rhs[0] }, selected[0]);
  TwoOneMux mux_b(select, { lhs[1], rhs[1] }, selected[1]);
  TwoOneMux mux_c(select, { lhs[2], rhs[2] }, selected[2]);
  TwoOneMux mux_d(select, { lhs[3], rhs[3] }, selected[3]);
  TwoOneMux mux_e(select, { lhs[4], rhs[4] }, selected[4]);
  TwoOneMux mux_f(select, { lhs[5], rhs[5] }, selected[5]);
  TwoOneMux mux_g(select, { lhs[6], rhs[6] }, selected[6]);
  TwoOneMux mux_h(select, { lhs[7], rhs[7] }, selected[7]);
  TwoOneMux mux_i(select, { lhs[8], rhs[8] }, selected[8]);
  TwoOneMux mux_j(select, { lhs[9], rhs[9] }, selected[9]);
  TwoOneMux mux_k(select, { lhs[10], rhs[10] }, selected[10]);
  TwoOneMux mux_l(select, { lhs[11], rhs[11] }, selected[11]);
  TwoOneMux mux_m(select, { lhs[12], rhs[12] }, selected[12]);
  TwoOneMux mux_n(select, { lhs[13], rhs[13] }, selected[13]);
  TwoOneMux mux_o(select, { lhs[14], rhs[14] }, selected[14]);
  TwoOneMux mux_p(select, { lhs[15], rhs[15] }, selected[15]);

endmodule // MuxBus


/**
 * 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
    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
```

```
      default: aluCtl <= 3'b101; // zero
    endcase
  endcase
endmodule // ALUControl


/**
 * CPU brings all of the modules together to create a
 * processor
 */
module CPU(
  input wire clk,
  output reg [15:0] pc,
  output reg [15:0] IF_ID,
  output reg [15:0] ID_EX,
  output reg [15:0] EX_MEM,
  output reg [15:0] MEM_WB,
  output wire [15:0] writeData
);

  // MEMORIES
  reg [15:0] IMemory [0:1023];
  reg [15:0] DMemory [0:1023];

  initial begin
    IMemory[0] = 16'b0101000100000000; // lw $1, 0x00($0)
    IMemory[1] = 16'b0101001000000001; // lw $2, 0x01($0)
    IMemory[2] = 16'b0000000000000000; // nop
    IMemory[3] = 16'b0000000000000000; // nop
    IMemory[4] = 16'b0000000000000000; // nop
    IMemory[5] = 16'b0111011011000000; // slt $3, $1, $2
    IMemory[6] = 16'b0000000000000000; // nop
    IMemory[7] = 16'b0000000000000000; // nop
    IMemory[8] = 16'b0000000000000000; // nop
    IMemory[9] = 16'b1000110000000101; // beq $3, $0, 0x05
    IMemory[10] = 16'b0000000000000000; // nop
    IMemory[11] = 16'b0000000000000000; // nop
    IMemory[12] = 16'b0000000000000000; // nop
    IMemory[13] = 16'b0110000100000000; // sw $1, 0x00($0)
    IMemory[14] = 16'b0110001000000001; // sw $2, 0x01($0)
    IMemory[15] = 16'b0000000000000000; // nop
    IMemory[16] = 16'b0000000000000000; // nop
    IMemory[17] = 16'b0000000000000000; // nop
    IMemory[18] = 16'b0101000100000000; // lw $1, 0x00($0)
    IMemory[19] = 16'b0101001000000001; // lw $2, 0x01($0)
    IMemory[20] = 16'b0000000000000000; // nop
    IMemory[21] = 16'b0000000000000000; // nop
    IMemory[22] = 16'b0000000000000000; // nop
    IMemory[23] = 16'b0001011011000000; // sub $3, $1, $2

    DMemory[0] = 16'h0005; // store 0x05 @ 0x00
    DMemory[1] = 16'h0007; // store 0x07 @ 0x01
  end

  // INSTRUCTION FETCH STAGE
  wire [15:0] next;
  wire [15:0] pcPlusOne;
  reg [15:0] IF_ID_pcPlusOne;
  reg [15:0] EX_MEM_target;
  reg EX_MEM_zero;
  reg EX_MEM_branch;

  and(shouldBranch, EX_MEM_branch, EX_MEM_zero);
  ALU fetchALU(pc, 16'h01, 3'b010, pcPlusOne, fetchZero);
  MuxBus pcMux(shouldBranch, EX_MEM_target, pcPlusOne, next);

  // INSTRUCTION DECODE STAGE
  wire [15:0] readDataA;
```

```
wire [15:0] readDataB;
wire [15:0] extended;
wire [7:0] control;
reg [15:0] ID_EX_pcPlusOne;
reg [15:0] ID_EX_readDataA;
reg [15:0] ID_EX_readDataB;
reg [15:0] ID_EX_extended;
reg [1:0] ID_EX_rt;
reg [1:0] ID_EX_rd;
reg [1:0] ID_EX_aluOp;
reg [1:0] MEM_WB_rd;
reg MEM_WB_regWrite;
reg ID_EX_regWrite;
reg ID_EX_memToReg;
reg ID_EX_branch;
reg ID_EX_memWrite;
reg ID_EX_aluSrc;
reg ID_EX_regDest;

RegisterFile regFile(clk, MEM_WB_regWrite, IF_ID[11:10], IF_ID[9:8], MEM_WB_rd,
      writeData, readDataA, readDataB);
MainControl mainControl(IF_ID[15:12], control);
assign extended = {{8{IF_ID[7]}}, IF_ID[7:0]};

// EXECUTE STAGE
wire [15:0] target;
wire [15:0] rhs;
wire [15:0] aluResult;
wire [2:0] aluControl;
wire [1:0] writeRegister;
reg [15:0] EX_MEM_aluResult;
reg [15:0] EX_MEM_readDataB;
reg [1:0] EX_MEM_rd;
reg EX_MEM_regWrite;
reg EX_MEM_memToReg;
reg EX_MEM_memWrite;

ALU branchALU(ID_EX_pcPlusOne, ID_EX_extended, 3'b010, target, branchZero);
ALU mainALU(ID_EX_readDataA, rhs, aluControl, aluResult, zero);
ALUControl aluController(ID_EX_aluOp, ID_EX[15:12], aluControl);
MuxBus rhsMux(ID_EX_aluSrc, ID_EX_extended, ID_EX_readDataB, rhs);
TwoOneMux writeRegisterA(ID_EX_regDest, { ID_EX_rd[0], ID_EX_rt[0] }, writeRegister[0]);
TwoOneMux writeRegisterB(ID_EX_regDest, { ID_EX_rd[1], ID_EX_rt[1] }, writeRegister[1]);

// MEMORY WRITE STAGE
wire [15:0] memOut;
reg [15:0] MEM_WB_memOut;
reg [15:0] MEM_WB_aluResult;
reg MEM_WB_memToReg;

assign memOut = DMemory[EX_MEM_aluResult];

always @(negedge clk) begin
  if (EX_MEM_memWrite) begin
    DMemory[EX_MEM_aluResult] <= EX_MEM_readDataB;
  end
end

// WRITE-BACK STAGE
MuxBus writeDataMux(MEM_WB_memToReg, MEM_WB_memOut, MEM_WB_aluResult, writeData);

// Initilize the pipeline
initial begin
  pc = 16'h0000;

  ID_EX = 16'h00;
  ID_EX_regWrite = 1'b0;
  ID_EX_memToReg = 1'b0;
```

```
    ID_EX_branch = 1'b0;
    ID_EX_memWrite = 1'b0;
    ID_EX_aluSrc = 1'b0;
    ID_EX_regDest = 1'b0;
    ID_EX_aluOp = 2'b00;

    IF_ID = 16'h00;

    EX_MEM_regWrite = 1'b0;
    EX_MEM_memToReg = 1'b0;
    EX_MEM_branch = 1'b0;
    EX_MEM_memWrite = 1'b0;
    EX_MEM_target = 16'h0000;

    MEM_WB_regWrite = 1'b0;
    MEM_WB_memToReg = 1'b0;
  end

  // Run the pipeline
  always @(negedge clk) begin

    // INSTRUCTION FETCH STAGE
    pc <= next;
    IF_ID_pcPlusOne <= pcPlusOne;
    IF_ID <= IMemory[pc];

    // INSTRUCTION DECODE STAGE
    ID_EX <= IF_ID;
    {
      ID_EX_regDest,
      ID_EX_aluSrc,
      ID_EX_memToReg,
      ID_EX_regWrite,
      ID_EX_memWrite,
      ID_EX_branch,
      ID_EX_aluOp
    } <= control;
    ID_EX_pcPlusOne <= IF_ID_pcPlusOne;
    ID_EX_readDataA <= readDataA;
    ID_EX_readDataB <= readDataB;
    ID_EX_extended <= extended;
    ID_EX_rt <= IF_ID[9:8];
    ID_EX_rd <= IF_ID[7:6];

    // EXECUTE STAGE
    EX_MEM <= ID_EX;
    EX_MEM_regWrite <= ID_EX_regWrite;
    EX_MEM_memToReg <= ID_EX_memToReg;
    EX_MEM_branch <= ID_EX_branch;
    EX_MEM_memWrite <= ID_EX_memWrite;
    EX_MEM_target <= target;
    EX_MEM_zero <= zero;
    EX_MEM_aluResult <= aluResult;
    EX_MEM_readDataB <= ID_EX_readDataB;
    EX_MEM_rd <= writeRegister;

    // MEMORY WRITE STATE
    MEM_WB <= EX_MEM;
    MEM_WB_regWrite <= EX_MEM_regWrite;
    MEM_WB_memToReg <= EX_MEM_memToReg;
    MEM_WB_memOut <= memOut;
    MEM_WB_aluResult <= EX_MEM_aluResult;
    MEM_WB_rd <= EX_MEM_rd;
  end

endmodule // CPU
```

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

  wire [15:0] pc;
  wire [15:0] IF_ID;
  wire [15:0] ID_EX;
  wire [15:0] EX_MEM;
  wire [15:0] MEM_WB;
  wire [15:0] writeData;

  reg clk;

  CPU cpu(clk, pc, IF_ID, ID_EX, EX_MEM, MEM_WB, writeData);

  always #1 clk = ~clk;

  initial begin
    $display("time pc IF_ID ID_EX EX_MEM MEM_WB writeData");
    $monitor("%2d %3d %h  %h  %h   %h   %h", $time, pc, IF_ID, ID_EX, EX_MEM, MEM_WB,
          writeData);
    clk = 1;
    #56 $finish;
  end

endmodule // TestBench
```

Machine Translation

```
0101000100000000 // lw  $1, 0x00($0)
0101001000000001 // lw  $2, 0x01($0)
0000000000000000 // nop
0000000000000000 // nop
0000000000000000 // nop
0111011011000000 // slt $3, $1, $2
0000000000000000 // nop
0000000000000000 // nop
0000000000000000 // nop
1000110000000101 // beq $3, $0, 0x05
0000000000000000 // nop
0000000000000000 // nop
0000000000000000 // nop
0110000100000000 // sw  $1, 0x00($0)
0110001000000001 // sw  $2, 0x01($0)
0000000000000000 // nop
0000000000000000 // nop
0000000000000000 // nop
0101000100000000 // lw  $1, 0x00($0)
0101001000000001 // lw  $2, 0x01($0)
0000000000000000 // nop
```

```
0000000000000000 // nop
0000000000000000 // nop
0001011011000000 // sub $3, $1, $2
```

Output (Branch Taken)

```
time pc IF_ID ID_EX EX_MEM MEM_WB writeData
 0    0  0000  0000  xxxx   xxxx   xxxx
 1    1  5100  0000  0000   xxxx   xxxx
 3    2  5201  5100  0000   0000   xxxx
 5    3  0000  5201  5100   0000   0000
 7    4  0000  0000  5201   5100   0007
 9    5  0000  0000  0000   5201   0005
11    6  76c0  0000  0000   0000   0000
13    7  0000  76c0  0000   0000   0000
15    8  0000  0000  76c0   0000   0000
17    9  0000  0000  0000   76c0   0000
19   10  8c05  0000  0000   0000   0000
21   11  0000  8c05  0000   0000   0000
23   12  0000  0000  8c05   0000   0000
25   15  0000  0000  0000   8c05   0000
27   16  0000  0000  0000   0000   0000
29   17  0000  0000  0000   0000   0000
31   18  0000  0000  0000   0000   0000
33   19  5100  0000  0000   0000   0000
35   20  5201  5100  0000   0000   0000
37   21  0000  5201  5100   0000   0000
39   22  0000  0000  5201   5100   0007
41   23  0000  0000  0000   5201   0005
43   24  16c0  0000  0000   0000   0000
45   25  xxxx  16c0  0000   0000   0000
47   26  xxxx  xxxx  16c0   0000   0000
49   27  xxxx  xxxx  xxxx   16c0   0002
51   28  xxxx  xxxx  xxxx   xxxx   xxxx
53   29  xxxx  xxxx  xxxx   xxxx   xxxx
55   30  xxxx  xxxx  xxxx   xxxx   xxxx
```

Output (Branch Not Taken)

```
time pc IF_ID ID_EX EX_MEM MEM_WB writeData
0     0  0000  0000  xxxx   xxxx   xxxx
1     1  5100  0000  0000   xxxx   xxxx
3     2  5201  5100  0000   0000   xxxx
5     3  0000  5201  5100   0000   0000
7     4  0000  0000  5201   5100   0005
9     5  0000  0000  0000   5201   0007
11    6  76c0  0000  0000   0000   0000
13    7  0000  76c0  0000   0000   0000
```

15	8	0000	0000	76c0	0000	0000
17	9	0000	0000	0000	76c0	0001
19	10	8c05	0000	0000	0000	0000
21	11	0000	8c05	0000	0000	0000
23	12	0000	0000	8c05	0000	0000
25	13	0000	0000	0000	8c05	0001
27	14	6100	0000	0000	0000	0000
29	15	6201	6100	0000	0000	0000
31	16	0000	6201	6100	0000	0000
33	17	0000	0000	6201	6100	0000
35	18	0000	0000	0000	6201	0001
37	19	5100	0000	0000	0000	0000
39	20	5201	5100	0000	0000	0000
41	21	0000	5201	5100	0000	0000
43	22	0000	0000	5201	5100	0005
45	23	0000	0000	0000	5201	0007
47	24	16c0	0000	0000	0000	0000
49	25	xxxx	16c0	0000	0000	0000
51	26	xxxx	xxxx	16c0	0000	0000
53	27	xxxx	xxxx	xxxx	16c0	fffe
55	28	xxxx	xxxx	xxxx	xxxx	xxxx