CSCI 2500 — Group Project

As we grouped to tackle the challenge of constructing a unique computer architecture, we ensured to account for the Seven Great Ideas in Computer Architecture via our proposed implementations. Based on Appendix A for our given Team ID of 19, we saw how we could utilize implementations from MIPS in our designs given the 32-bit architecture.

For Design 1, to support our minimalistic design, we decided to format our direct addressing mode using Instruction[31-27] bits as opcodes, and Instruction[26-0] bits designated for holding the address of value required for instruction. This design choice is effective, allowing for a large range of memory addresses to be available for the minimal instructions provided by our architecture, essential for large-scale applications. For our choice in endianness, we decided to go for Big Endian implementation, taking inspiration from the MIPS ISA we studied previously during the semester. Given the high level of comfort when working with Big Endian ISAs, we also felt it would allow for a smooth development experience. A Big Endian implementation is catered for our left-to-right reading order, making debugging much simpler. It's also commonly used in network protocols, which can allow simplifying our data exchange with other systems if needed.

To calculate the clock rate of the CPU in Design 1 and 1a, we analyzed our Verilog modules, waveforms, and prior notes to estimate the latencies of other components in our implementations based on the provided default latencies. Assume that the clock rate of any CPU can be determined via summation of all applicable latencies during a critical path over one. We estimated that the latency of ALU should be 5 ns, the latency of registers involved in the fetch-decode-execute cycle should be 7 ns, and multiplexors involved in the execution of specific instructions should have a latency of 5 ns. The period is 20ns+7ns+5ns+5ns, which is 37 ns. We find the clock rate by taking the inverse of 37 ns, which gives us a roughly estimated CPU clock rate of 27 MHz. This is a reasonable clock rate for a simple 32-bit CPU that has a minimal ISA for the given project at hand. All future clock rate adjustments will account for SRAM implementation and any possible miss penalty latencies.

The clock rate of the CPU in Design 1b will be the same as the standard clock rate, but only if the cache has no misses. This is unrealistic of course, so a worst-case latency for the processor would be 20ns+7ns+5ns+5ns+7ns, = 44 ns, given that a miss would result in the loading of a different register, and that register loads take 7ns. The clock rate then is 1/44ns = 22.7MHz, which reflects the slowdown caused by a cache miss.

For Design 1a we modified our instruction layout to accommodate immediate instructions, as specified in the Team ID table. Our new instruction template uses Instruction[31] to determine whether this instruction represents a direct or immediate instruction. Instruction[30:27] denotes the opcode of the instruction to follow, and Instruction[26:0] is reserved for the data, regardless of whether it's immediate or direct. This uniform instruction style allows for a streamlined implementation in the Verilog.

For Design 1b we implemented a data cache capable of holding 524,288 bits of data. Our entry in the Team ID table required that we use a maximum of 600,000 bits of data in the cache, so our design is compliant in that regard. The cache is a direct mapped cache with a write-through policy. We chose this design due to the simplicity and cost-effectiveness of direct-mapped caches with write-through policies. The cache is formatted with 2^{14} indices, each 32 bits wide.

Given our Team ID of 19, we were assigned specific restrictions on memory management and ordered to implement an additional addressing mode. Our memory organization was given 256Mi * 16 with a total size of 1Gi. This made our design choice in our RAM to utilize 24 bits for the address width to account for 256 Mi and a data width of 32 bits to account for our 32-bit word sizes. For a data width of 32 bits, we must use two RAM chips to store any given variable, with a total of four rows of these two-chip columns. The cache was specified to not be implemented within this early stage, so no considerations were made until we reached further design implementations.

For our instruction set, we implemented all of the required instructions plus a few extra credit instructions as well. We used the mandatory add, halt, load, store, clear, skip, jump, and addi instructions as specified in the project overview. We then created the subtract, and, or, not, subi, andi, and ori instructions as well.

The following code below is the Verilog code implementation of our proposed designs for all designs. All documentation for the proper references is included in code snippets and also has been appended to our references section at the end of this document.

Verilog Code - Design 1

alu.sv

```
// Adopted from https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html
`timescale 1ns / 1ps

module alu(
        input [31:0] A,B,  // ALU 32-bit Inputs
        input [2:0] ALU_Sel,// ALU Selection
        output [31:0] ALU_Out // ALU 32-bit Output
  );
  reg [31:0] ALU_Result;
  wire [31:0] tmp;
  assign ALU_Out = ALU_Result; // ALU out
  always @(*)
  begin
     case(ALU_Sel)
     3'b001: // Addition
        ALU_Result = A + B ;
     3'b010: // Subtraction
        ALU_Result = A - B ;
     3'b011: // Clear
        ALU_Result = 0;
      3'b000: // AND
         ALU_Result = A & B;
      3'b100: // OR
         ALU_Result = A | B;
     default: ALU_Result = A;
     endcase
  end

endmodule
```

decoder.sv

```
// Adopted from https://circuitcove.com/design-examples-decoders/
`timescale 1 ns / 1 ps

module decoder
  # (parameter ENCODE_WIDTH = 4,
     parameter DECODE_WIDTH = 2**ENCODE_WIDTH
    )

  (
    input  [ENCODE_WIDTH-1:0] in,
    output [DECODE_WIDTH-1:0] out
  );

  localparam latency = 1;

  assign #latency out = 'b1<<in;

endmodule
```

ram.sv:

```
// Adopted from https://www.chipverify.com/verilog/verilog-single-port-ram
`timescale 1 ns / 1 ps

module single_port_sync_ram
  # (parameter ADDR_WIDTH = 12,
     parameter DATA_WIDTH = 16,
     parameter LENGTH = (1<<ADDR_WIDTH)
    )

  (   input clk,
      input [ADDR_WIDTH-1:0] addr,
      inout [DATA_WIDTH-1:0] data,
      input cs,
      input we,
      input oe
  );

  reg [DATA_WIDTH-1:0] tmp_data;
  reg [DATA_WIDTH-1:0] mem[LENGTH];

  always @ (posedge clk) begin
    if (cs & we)
      mem[addr] <= data;
  end

  always @ (negedge clk) begin
    if (cs & !we)
      tmp_data <= mem[addr];
  end

  assign data = cs & oe & !we ? tmp_data : 'hz;
endmodule
```

ram_large.sv:

```
// Adopted from https://www.chipverify.com/verilog/verilog-single-port-ram
`include "ram.sv"
`include "decoder.sv"

`timescale 1 ns / 1 ps

module single_port_sync_ram_large
  # ( parameter ADDR_WIDTH = 14, // used to be 14 -- 28
      parameter DATA_WIDTH = 32,
      parameter DATA_WIDTH_SHIFT = 1
    )

  (   input clk,
      input [ADDR_WIDTH-1:0] addr,
      inout [DATA_WIDTH-1:0] data,
      input cs_input,
      input we,
      input oe
  );

  wire [3:0] cs;

  decoder #(.ENCODE_WIDTH(2)) dec
  (   .in(addr[ADDR_WIDTH-1:ADDR_WIDTH-2]),
      .out(cs)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u00
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[0]),
      .we(we),
      .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u01
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
      .cs(cs[0]),
      .we(we),
      .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u10
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[1]),
    .we(we),
    .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u11
  (   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[1]),
    .we(we),
    .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u20
  (   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[2]),
    .we(we),
    .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u21
  (   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[2]),
    .we(we),
    .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u30
  (   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[3]),
    .we(we),
    .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u31
  (   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[3]),
    .we(we),
    .oe(oe)
  );

Endmodule
```

cache.sv:

```
// Adopted from https://www.chipverify.com/verilog/verilog-single-port-ram
`timescale 1 ns / 1 ps

module cache
  (   input clk,
      inout [31:0] data,
      inout found,
      input we,
      input oe
  );

  reg [31:0] tmp_data;
  reg [31:0] mem[15'b100000000000000]; // 16384 spaces
  reg temp_found;

  always @ (posedge clk) begin
    if (we) begin
      mem[data[13:0]][31:14] <= data[31:14]; // tag
      mem[data[13:0]][0] <= 1;              // valid
    end
  end

  always @ (negedge clk) begin
    if (!we) begin
            if(mem[data[13:0]][0] == 1'b1 && mem[data[13:0]][31:14] == data[31:14]) // verify tags against each
other
            begin
                    tmp_data[31:14] <= mem[data[13:0]][31:14];   // tag
                    tmp_data[13:0] <= data[13:0];                // index
                    temp_found <= 1;
            end
            else begin
                    temp_found <= 0;  // make 'we' = 1 in the superclass after this.
            end
    end

  end

  assign data = oe & !we ? tmp_data : 'hz;
  assign found = temp_found;
endmodule
```

Testbenches - Design 1

test_alu.sv:

```
// Adopted from https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html
`timescale 1ns / 1ps

module test_alu;
//Inputs
reg[31:0] A,B;
reg[1:0] ALU_Sel;

//Outputs
wire[15:0] ALU_Out;
// Verilog code for ALU
integer i;
alu u0(
        A,B,  // ALU 16-bit Inputs
        ALU_Sel,// ALU Selection
        ALU_Out // ALU 16-bit Output
    );
   initial begin
   $dumpfile("dump.vcd");
   $dumpvars;
   A = 'hFA;
   B = 'h02;
   ALU_Sel = 2'h0;

     for (i=0;i<4;i=i+1)
     begin
      ALU_Sel = ALU_Sel + 2'h1;
      #10;
     end;
   end
endmodule
```

test_cpu.sv:

```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 26; // used to be 14
  parameter DATA_WIDTH = 32;
```

```
reg osc;
localparam period = 10;

wire clk;
assign clk = osc;

reg cs;
reg we;
reg oe;
integer i;
reg [ADDR_WIDTH-1:0] MAR;
wire [DATA_WIDTH-1:0] data;
reg [DATA_WIDTH-1:0] testbench_data;
assign data = !oe ? testbench_data : 'hz;

single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
(   .clk(clk),
 .addr(MAR),
    .data(data[DATA_WIDTH-1:0]),
    .cs_input(cs),
    .we(we),
    .oe(oe)
);

reg [31:0] A;
reg [31:0] B;
wire [31:0] ALU_Out;
reg [2:0] ALU_Sel;
alu alu16(
  .A(A),
  .B(B),  // ALU 16-bit Inputs
  .ALU_Sel(ALU_Sel),// ALU Selection
  .ALU_Out(ALU_Out) // ALU 16-bit Output
   );

reg [31:0] PC = 'h100;
reg [31:0] IR = 'h0;
reg [31:0] MBR = 'h0;
reg [31:0] AC = 'h0;

initial osc = 1;  //init clk = 1 for positive-edge triggered
always begin  // Clock wave
   #period  osc = ~osc;
end

initial begin

   $dumpfile("dump.vcd");
   $dumpvars;
```

```
// Multiplication by addition program

/*
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h101; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110D;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h103; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h310B;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210D;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h105; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h310F;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h107; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h8400;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h109; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h9102;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h7000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10B; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0005;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0007;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10D; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10F; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hFFFF;
$display("%h HELLO\n", MAR);
*/


/*
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
$display("%h And some other stuff\n", MAR);
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
```

```
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
$display("%h HALT HERE %b\n", MAR, MBR);
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h7800000A;
$display("%h MAYBE HALT HERE %b\n", MAR, MBR);
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
$display("%h\n", MAR);
*/


// Immediate Addressing:
// New code, and I'm POSITIVE this version works.
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;
```

```
// Direct Addressing Only:
/*
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h38000122;
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;
@(posedge clk) MAR <= 'h122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;
*/


@(posedge clk) PC <= 'h100;

for (i = 0; i < 135; i = i+1) begin

$display("%h\n", MAR);

    // Fetch
    @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
    @(posedge clk) IR <= data;
    @(posedge clk) PC <= PC + 2;
    // Decode and execute
    case(IR[31])
    1'b0: begin          // Register addressing


    case(IR[30:27])
      4'b0000: begin  // add
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= data;
           @(posedge clk) ALU_Sel <= 'b001; A <= AC; B <= MBR;
           @(posedge clk) AC <= ALU_Out;
      end
   4'b0001: begin  // halt
           $display("%h THIS HALTS!!!\n", MAR);  // This never halts.
           @(posedge clk) PC <= PC - 2;



      end
      4'b0010: begin   // load
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= data;
           @(posedge clk) AC <= MBR;
           #1
           $display("%b AC %h %d\n", AC, MAR, AC[11:0]);
      end
      4'b0011: begin    // store
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= AC;
           @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
           //#1
           //$display("%b AC %h %d\n", AC, MAR, AC[11:0]);
      end
      4'b0100: begin  // clear
          @(posedge clk) AC <= 0;

      end
      4'b0101: begin  // skip
          //$display("%h THIS SKIPS!!!\n", MAR);
          @(posedge clk)
          if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 2;
          else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 2;
          else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 2;
          else if(IR[11:10]==2'b01 && AC[26:0] == 27'b111111111111111111111111111) PC <= PC + 2;
          $display("Hello World! %b %b\n", AC, IR[11:10]);
      end
      4'b0110: begin // jump
        @(posedge clk) PC <= IR[26:0];
      end

      4'b0111: begin // subtract
      @(posedge clk) MAR <= IR[26:0];
      @(posedge clk) MBR <= data;
      @(posedge clk) ALU_Sel <= 'b010; A <= AC; B <= MBR;
      @(posedge clk) AC <= ALU_Out;
      end

      4'b1000: begin // and
      @(posedge clk) MAR <= IR[26:0];
      @(posedge clk) MBR <= data;
      @(posedge clk) ALU_Sel <= 'b000; A <= AC; B <= MBR;
      @(posedge clk) AC <= ALU_Out;
      end
```

```
    4'b1001: begin // or
    @(posedge clk) MAR <= IR[26:0];
    @(posedge clk) MBR <= data;
    @(posedge clk) ALU_Sel <= 'b100; A <= AC; B <= MBR;
    @(posedge clk) AC <= ALU_Out;
    end

    4'b1000: begin // not
    @(posedge clk) AC <= ~AC;
    end

   endcase

   end
   1'b1: begin        // Immediate addressing

   case(IR[30:27])
    4'b0000: begin  // addi
          @(posedge clk) AC <= AC + IR[26:0];
    end

    4'b0111: begin // subi
      @(posedge clk) AC <= AC - IR[26:0];
    end

    4'b1000: begin // andi
      @(posedge clk) AC <= AC & IR[26:0];
    end

    4'b1001: begin // ori
      @(posedge clk) AC <= AC | IR[26:0];
    end

   endcase




   end
   endcase


end


@(posedge clk) MAR <= 'h10D; we <= 0; cs <= 1; oe <= 1;
```

```
      @(posedge clk)

     #20 $finish;
    end

  endmodule
```

test_decoder.sv

```
`timescale 1 ns / 1 ps

module test_decoder;
  parameter ENCODE_WIDTH = 2;
  parameter DECODE_WIDTH = 2**ENCODE_WIDTH;

  reg osc;
  reg  [ENCODE_WIDTH-1:0] in;
  reg [DECODE_WIDTH-1:0] out;

  localparam period = 10;

  wire clk;
  assign clk = osc;

  decoder #(.ENCODE_WIDTH(ENCODE_WIDTH)) u0
  (   .in(in),
      .out(out)
  );

  integer i;

  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    {osc, in} <= 0;

    for (i = 0; i < 4; i = i+1) begin
      @(posedge clk) in = i;
    end
    #(period*4) $finish;
  end

endmodule
```

test_ram.sv

```
// Adopted from https://www.chipverify.com/verilog/verilog-single-port-ram
`timescale 1 ns / 1 ps

module test_ram;
  parameter ADDR_WIDTH = 14;
  parameter DATA_WIDTH = 8;

  reg clk;
  reg cs;
  reg we;
  reg oe;
  reg [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;

  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH)) u0
  (   .clk(clk),
      .addr(addr),
      .data(data),
      .cs(cs),
      .we(we),
      .oe(oe)
  );

  always #20 clk = ~clk;
  assign data = !oe ? testbench_data : 'hz;

  integer i;
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    {clk, cs, we, addr, testbench_data, oe} <= 0;

    repeat (2) @ (posedge clk);

    for (i = 0; i < 16; i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 0; i < 16; i= i+1) begin
    repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
   end

   @(posedge clk) cs <= 0;

   #360 $finish;
  end
endmodule
```

test_ram_large.sv

```
// Adopted from https://www.chipverify.com/verilog/verilog-single-port-ram
`timescale 1 ns / 1 ps

module test_ram;
  parameter ADDR_WIDTH = 14;
  parameter DATA_WIDTH = 16;

  reg clk;
  reg cs;
  reg we;
  reg oe;
  reg [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) u0
  (   .clk(clk),
      .addr(addr),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );


  always #20 clk = ~clk;
  assign data = !oe ? testbench_data : 'hz;

  integer i;
  initial begin
   $dumpfile("dump.vcd");
   $dumpvars;
   {clk, cs, we, addr, testbench_data, oe} <= 0;

   repeat (2) @ (posedge clk);
```

```
    // Write
    for (i = 2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**(ADDR_WIDTH-1)-4; i < 2**(ADDR_WIDTH-1); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2)-4; i <
2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**ADDR_WIDTH-4; i < 2**ADDR_WIDTH-4; i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    // Read
    for (i = 2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**(ADDR_WIDTH-1)-4; i < 2**(ADDR_WIDTH-1); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2)-4; i <
2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**ADDR_WIDTH-4; i < 2**ADDR_WIDTH-4; i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    #40 $finish;
  end
endmodule
```

test_cpu_cache.sv:

```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 26; // used to be 14
  parameter DATA_WIDTH = 32;

  reg osc;
  localparam period = 10;

  wire clk;
  assign clk = osc;

  reg cs;
  reg we;
  reg oe;
  integer i;
  reg [ADDR_WIDTH-1:0] MAR;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;


  // assign data = !oe ? testbench_data : 'hz;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
  (   .clk(clk),
   .addr(MAR),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );



  reg found;
  wire [DATA_WIDTH-1:0] cache_data;
  reg [DATA_WIDTH-1:0] cache_reg;  // because Verilog is finnicky and you can't just set the value of a wire to
something.

  assign cache_data = cache_reg; // See above.

  reg cwe;
  reg coe;

  cache the_cache
  ( .clk(clk),
    .data(cache_data),
    .found(found),
    .we(cwe),
    .oe(coe)
  );
```

```
assign data = (!oe && !found) ? testbench_data : 'hz;
assign cache_data = (!oe && !found) ? testbench_data : 'hz;
assign data = (!oe && found) ? cache_data : 'hz;


/*
always @(*) begin
  if(!oe && (found == '0))
  begin
    data = testbench_data;
    assign cache_data = testbench_data;
  end
  else if(!oe && found)
    assign data = cache_data;
  else
    assign data = 'hz;
end
*/



reg [31:0] A;
reg [31:0] B;
wire [31:0] ALU_Out;
reg [2:0] ALU_Sel;
alu alu16(
  .A(A),
  .B(B),  // ALU 16-bit Inputs
  .ALU_Sel(ALU_Sel),// ALU Selection
  .ALU_Out(ALU_Out) // ALU 16-bit Output
   );

reg [31:0] PC = 'h100;
reg [31:0] IR = 'h0;
reg [31:0] MBR = 'h0;
reg [31:0] AC = 'h0;

initial osc = 1;  //init clk = 1 for positive-edge triggered
always begin  // Clock wave
   #period  osc = ~osc;
end

initial begin


  /*
  if(!oe && (found == '0))
  begin
    data <= testbench_data;
  cache_data <= testbench_data;
end
else if(!oe && found)
  data <= cache_data;
else
  data <= 'hz;
*/

 $dumpfile("dump.vcd");
 $dumpvars;
// Multiplication by addition program

/*
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h101; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110D;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h103; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h310B;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210D;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h105; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h110E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h310F;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h107; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h210E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h8400;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h109; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h9102;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h7000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10B; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0005;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0007;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10D; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10F; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hFFFF;
$display("%h HELLO\n", MAR);
*/


/*
```

```
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
$display("%h And some other stuff\n", MAR);
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
$display("%h HALT HERE %b\n", MAR, MBR);
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
$display("%h MAYBE HALT HERE %b\n", MAR, MBR);
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;
$display("%h\n", MAR);
@(posedge clk) MAR <= 'h122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
$display("%h\n", MAR);
*/


// New code, and I'm POSITIVE this version works.
@(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
@(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
```

```
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;

@(posedge clk) PC <= 'h100;

for (i = 0; i < 500; i = i+1) begin

$display("%h\n", MAR);

    // Fetch
    @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
    @(posedge clk) IR <= data;
    @(posedge clk) PC <= PC + 2;
    // Decode and execute
    case(IR[31])
    1'b0: begin           // Register addressing


    case(IR[30:27])
      4'b0000: begin  // add
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= data;
           @(posedge clk) ALU_Sel <= 'b001; A <= AC; B <= MBR;
           @(posedge clk) AC <= ALU_Out;
      end
   4'b0001: begin  // halt
           $display("%h THIS HALTS!!!\n", MAR);  // This never halts.
           @(posedge clk) PC <= PC - 2;



      end
      4'b0010: begin   // load
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= data;
           @(posedge clk) AC <= MBR;
           #1
           $display("%b AC %h %d\n", AC, MAR, AC[11:0]);
      end
      4'b0011: begin    // store
           @(posedge clk) MAR <= IR[26:0];
           @(posedge clk) MBR <= AC;
```

```
        @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
  end
  4'b0100: begin  // clear
      @(posedge clk) AC <= 0;

  end
  4'b0101: begin  // skip
      //$display("%h THIS SKIPS!!!\n", MAR);
      @(posedge clk)
      if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 2;
      else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 2;
      else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 2;
      else if(IR[11:10]==2'b01 && AC[26:0] == 27'b111111111111111111111111111) PC <= PC + 2;
      $display("Hello World! %b %b\n", AC, IR[11:10]);
  end
  4'b0110: begin // jump
    @(posedge clk) PC <= IR[26:0];
  end

  4'b0111: begin // subtract
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b010; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1000: begin // and
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b000; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1001: begin // or
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b100; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1000: begin // not
  @(posedge clk) AC <= ~AC;
  end

endcase

end
1'b1: begin          // Immediate addressing
```

```
          case(IR[30:27])
            4'b0000: begin  // addi
                  @(posedge clk) AC <= AC + IR[26:0];
            end

            4'b0111: begin // subi
              @(posedge clk) AC <= AC - IR[26:0];
            end

            4'b1000: begin // andi
              @(posedge clk) AC <= AC & IR[26:0];
            end

            4'b1001: begin // ori
              @(posedge clk) AC <= AC | IR[26:0];
            end

          endcase





          end
          endcase


    end


    @(posedge clk) MAR <= 'h10D; we <= 0; cs <= 1; oe <= 1;

    @(posedge clk)

   #20 $finish;
  end

endmodule
```

Verilog code - Design 1a / 1b

> alu.sv

```
`timescale 1ns / 1ps

module alu(
           input [31:0] A,B,  // ALU 32-bit Inputs
           input [2:0] ALU_Sel,// ALU Selection
           output [31:0] ALU_Out // ALU 32-bit Output
    );
    reg [31:0] ALU_Result;
    wire [31:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    always @(*)
    begin
        case(ALU_Sel)
        3'b001: // Addition
           ALU_Result = A + B ;
        3'b010: // Subtraction
           ALU_Result = A - B ;
        3'b011: // Clear
           ALU_Result = 0;
         3'b000: // AND
            ALU_Result = A & B;
         3'b100: // OR
            ALU_Result = A | B;
        default: ALU_Result = A;
        endcase
    end

endmodule
```

> cache.sv

```
`timescale 1 ns / 1 ps

module cache
  (   input clk,
      inout [31:0] data,
      inout found,
      input we,
      input oe
  );

  reg [31:0] tmp_data;
  reg [31:0] mem[15'b100000000000000]; // 16384 spaces
  reg temp_found;

  always @ (posedge clk) begin
    if (we) begin
      mem[data[13:0]][31:14] <= data[31:14]; // tag
      mem[data[13:0]][0] <= 1;               // valid
    end
  end

  always @ (negedge clk) begin
    if (!we) begin
          if(mem[data[13:0]][0] == 1'b1 && mem[data[13:0]][31:14] == data[31:14]) // verify tags against
each other
          begin
                    tmp_data[31:14] <= mem[data[13:0]][31:14];   // tag
                    tmp_data[13:0] <= data[13:0];                // index
                    temp_found <= 1;
          end
          else begin
                    temp_found <= 0;  // make 'we' = 1 in the superclass after this.
          end
    end

  end
  assign data = oe & !we ? tmp_data : 'hz;
  assign found = temp_found;
endmodule
```

> decoder.sv

```
`timescale 1 ns / 1 ps

module decoder
  # (parameter ENCODE_WIDTH = 4,
     parameter DECODE_WIDTH = 2**ENCODE_WIDTH
    )

  (
    input  [ENCODE_WIDTH-1:0] in,
    output [DECODE_WIDTH-1:0] out
  );

  localparam latency = 1;

  assign #latency out = 'b1<<in;

endmodule
```

> ram.sv

```
`timescale 1 ns / 1 ps

module single_port_sync_ram
  # (parameter ADDR_WIDTH = 24, // used to be 12 -- 26
     parameter DATA_WIDTH = 16,
     parameter LENGTH = (1<<ADDR_WIDTH)
    )

  (   input clk,
      input [ADDR_WIDTH-1:0] addr,
      inout [DATA_WIDTH-1:0] data,
      input cs,
      input we,
      input oe
  );

  reg [DATA_WIDTH-1:0] tmp_data;
  reg [DATA_WIDTH-1:0] mem[LENGTH];

  always @ (posedge clk) begin
    // Used when debugging a multiplication program
    // if (addr=='h10D) $display("Product is %d", mem[addr]);
    if (cs & we)
      mem[addr] <= data;
  end

  always @ (negedge clk) begin
    if (cs & !we)
      tmp_data <= mem[addr];
  end

  assign data = cs & oe & !we ? tmp_data : 'hz;
endmodule
```

> ram_large.sv

```
`include "ram.sv"
`include "decoder.sv"

`timescale 1 ns / 1 ps

module single_port_sync_ram_large
  # ( parameter ADDR_WIDTH = 26, // used to be 14 -- 28
      parameter DATA_WIDTH = 32,
      parameter DATA_WIDTH_SHIFT = 1
    )

  (   input clk,
      input [ADDR_WIDTH-1:0] addr,
      inout [DATA_WIDTH-1:0] data,
      input cs_input,
      input we,
      input oe
  );

  wire [3:0] cs;

  decoder #(.ENCODE_WIDTH(2)) dec
  (   .in(addr[ADDR_WIDTH-1:ADDR_WIDTH-2]),
      .out(cs)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u00
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[0]),
      .we(we),
      .oe(oe)
  );
```

```
single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u01
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[0]),
    .we(we),
    .oe(oe)
);

single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u10
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[1]),
    .we(we),
    .oe(oe)
);
single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u11
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[1]),
    .we(we),
    .oe(oe)
);

single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u20
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[2]),
    .we(we),
    .oe(oe)
);
single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u21
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
    .cs(cs[2]),
    .we(we),
    .oe(oe)
);

single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u30
(   .clk(clk),
    .addr(addr[ADDR_WIDTH-3:0]),
    .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
    .cs(cs[3]),
    .we(we),
    .oe(oe)
 );
 single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u31
 (   .clk(clk),
     .addr(addr[ADDR_WIDTH-3:0]),
     .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
     .cs(cs[3]),
     .we(we),
     .oe(oe)
 );

endmodule
```

> test_cpu.sv

```
`timescale 1 ns / 1 ps

module test_cpu;
 parameter ADDR_WIDTH = 26; // used to be 14
 parameter DATA_WIDTH = 32;

 reg osc;
 localparam period = 44;

 wire clk;
 assign clk = osc;

 reg cs;
 reg we;
 reg oe;
 integer i;
 reg [ADDR_WIDTH-1:0] MAR;
 wire [DATA_WIDTH-1:0] data;
 reg [DATA_WIDTH-1:0] testbench_data;
 assign data = !oe ? testbench_data : 'hz;

 single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
 (   .clk(clk),
  .addr(MAR),
     .data(data[DATA_WIDTH-1:0]),
     .cs_input(cs),
     .we(we),
     .oe(oe)
 );

 reg [31:0] A;
 reg [31:0] B;
 wire [31:0] ALU_Out;
 reg [2:0] ALU_Sel;
```

```
alu alu16(
  .A(A),
  .B(B),  // ALU 16-bit Inputs
  .ALU_Sel(ALU_Sel),// ALU Selection
  .ALU_Out(ALU_Out) // ALU 16-bit Output
   );

reg [31:0] PC = 'h100;
reg [31:0] IR = 'h0;
reg [31:0] MBR = 'h0;
reg [31:0] AC = 'h0;

integer fib = 1;

initial osc = 1;  //init clk = 1 for positive-edge triggered
always begin  // Clock wave
   #period  osc = ~osc;
end

initial begin

   $dumpfile("dump.vcd");
   $dumpvars;
  // Multiplication by addition program


  // Immediate Addressing:
  // New code, and I'm POSITIVE this version works.
  @(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
  @(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
  @(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
  @(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
  @(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
  @(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
  @(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
  @(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
  @(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
  @(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
  @(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
  @(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
  @(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
  @(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
  @(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
  @(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
  @(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;


  @(posedge clk) PC <= 'h100;
```

```
for (i = 0; i < 135; i = i+1) begin

// $display("%h\n", MAR);

   // Fetch
   @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
   @(posedge clk) IR <= data;
   @(posedge clk) PC <= PC + 2;
   // Decode and execute
   case(IR[31])
   1'b0: begin          // Register addressing


   case(IR[30:27])
    4'b0000: begin  // add
        @(posedge clk) MAR <= IR[26:0];
        @(posedge clk) MBR <= data;
        @(posedge clk) ALU_Sel <= 'b001; A <= AC; B <= MBR;
        @(posedge clk) AC <= ALU_Out;
        @(posedge clk) fib = fib + 1;

        #1
        $display("Fib:%0d -> AC:%d\n",fib, AC[11:0]);
    end
  4'b0001: begin  // halt
        @(posedge clk) PC <= PC - 2;


    end
    4'b0010: begin   // load
        @(posedge clk) MAR <= IR[26:0];
        @(posedge clk) MBR <= data;
        @(posedge clk) AC <= MBR;
        #1
        // $display("%b AC %h %d\n", AC, MAR, AC[11:0]);
    end
    4'b0011: begin    // store
        @(posedge clk) MAR <= IR[26:0];
        @(posedge clk) MBR <= AC;
        @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
    end
    4'b0100: begin  // clear
      @(posedge clk) AC <= 0;

    end
    4'b0101: begin  // skip
      @(posedge clk)
      if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 2;
      else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 2;
      else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 2;
      else if(IR[11:10]==2'b01 && AC[26:0] == 27'b111111111111111111111111111) PC <= PC + 2;
  end
  4'b0110: begin // jump
    @(posedge clk) PC <= IR[26:0];
  end

  4'b0111: begin // subtract
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b010; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1000: begin // and
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b000; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1001: begin // or
  @(posedge clk) MAR <= IR[26:0];
  @(posedge clk) MBR <= data;
  @(posedge clk) ALU_Sel <= 'b100; A <= AC; B <= MBR;
  @(posedge clk) AC <= ALU_Out;
  end

  4'b1000: begin // not
  @(posedge clk) AC <= ~AC;
  end

endcase

end
1'b1: begin          // Immediate addressing

case(IR[30:27])
  4'b0000: begin  // addi
        @(posedge clk) AC <= AC + IR[26:0];
  end

  4'b0111: begin // subi
    @(posedge clk) AC <= AC - IR[26:0];
  end

  4'b1000: begin // andi
    @(posedge clk) AC <= AC & IR[26:0];
  end
```

```
            4'b1001: begin // ori
              @(posedge clk) AC <= AC | IR[26:0];
            end

          endcase





          end
          endcase


    end


    @(posedge clk) MAR <= 'h10D; we <= 0; cs <= 1; oe <= 1;

    @(posedge clk)

   #20 $finish;
  end

endmodule
```

> test_cpu_cache.sv

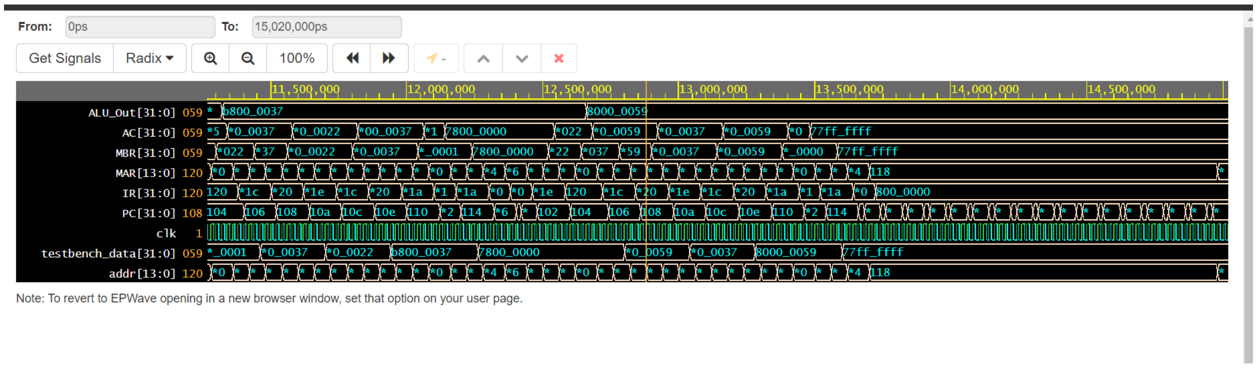
```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 26; // used to be 14
  parameter DATA_WIDTH = 32;

  reg osc;
  localparam period = 44;

  wire clk;
  assign clk = osc;

  reg cs;
  reg we;
  reg oe;
  integer i;
  reg [ADDR_WIDTH-1:0] MAR;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;
  integer fib = 1;


  // assign data = !oe ? testbench_data : 'hz;
  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
  (   .clk(clk),
   .addr(MAR),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );



  reg found;
  wire [DATA_WIDTH-1:0] cache_data;
  reg [DATA_WIDTH-1:0] cache_reg;  // because Verilog is finnicky and you can't just set the value of a
wire to something.

  assign cache_data = cache_reg; // See above.

  reg cwe;
  reg coe;

  cache the_cache
  ( .clk(clk),
    .data(cache_data),
    .found(found),
    .we(cwe),
    .oe(coe)
  );


  assign data = (!oe && !found) ? testbench_data : 'hz;
  assign cache_data = (!oe && !found) ? testbench_data : 'hz;
  assign data = (!oe && found) ? cache_data : 'hz;


  /*
  always @(*) begin
    if(!oe && (found == '0))
    begin
      data = testbench_data;
      assign cache_data = testbench_data;
    end
    else if(!oe && found)
      assign data = cache_data;
    else
      assign data = 'hz;
  end
  */




  reg [31:0] A;
  reg [31:0] B;
  wire [31:0] ALU_Out;
  reg [2:0] ALU_Sel;
  alu alu16(
    .A(A),
    .B(B),  // ALU 16-bit Inputs
    .ALU_Sel(ALU_Sel),// ALU Selection
    .ALU_Out(ALU_Out) // ALU 16-bit Output
     );

  reg [31:0] PC = 'h100;
  reg [31:0] IR = 'h0;
  reg [31:0] MBR = 'h0;
  reg [31:0] AC = 'h0;

  initial osc = 1;  //init clk = 1 for positive-edge triggered
  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin


    /*
    if(!oe && (found == '0))
    begin
      data <= testbench_data;
      cache_data <= testbench_data;
    end
    else if(!oe && found)
      data <= cache_data;
    else
      data <= 'hz;
    */

     $dumpfile("dump.vcd");
     $dumpvars;
    // Multiplication by addition program

    // New code, and I'm POSITIVE this version works.
    @(posedge clk) MAR <= 'h100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011E;
    @(posedge clk) MAR <= 'h102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000120;
```

```
@(posedge clk) MAR <= 'h104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011C;
@(posedge clk) MAR <= 'h106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000120;
@(posedge clk) MAR <= 'h108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011E;
@(posedge clk) MAR <= 'h10A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011C;
@(posedge clk) MAR <= 'h10C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h18000120;
@(posedge clk) MAR <= 'h10E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000011A;
@(posedge clk) MAR <= 'h110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hB8000001;
@(posedge clk) MAR <= 'h112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1800011A;
@(posedge clk) MAR <= 'h114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h28000400;
@(posedge clk) MAR <= 'h116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h30000100;
@(posedge clk) MAR <= 'h118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h08000000;
@(posedge clk) MAR <= 'h11A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000009;
@(posedge clk) MAR <= 'h11C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h11E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000000;
@(posedge clk) MAR <= 'h120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h78000001;

@(posedge clk) PC <= 'h100;

for (i = 0; i < 500; i = i+1) begin

   // Fetch
   @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
   @(posedge clk) IR <= data;
   @(posedge clk) PC <= PC + 2;
   // Decode and execute
   case(IR[31])
   1'b0: begin        // Register addressing


   case(IR[30:27])
    4'b0000: begin  // add
        @(posedge clk) MAR <= IR[26:0];
        @(posedge clk) MBR <= data;
        @(posedge clk) ALU_Sel <= 'b001; A <= AC; B <= MBR;
        @(posedge clk) AC <= ALU_Out;
        @(posedge clk) fib = fib + 1;

        #1
        $display("Fib:%0d -> AC:%d\n",fib, AC[11:0]);
    end
  4'b0001: begin  // halt
        @(posedge clk) PC <= PC - 2;



    end
    4'b0010: begin   // load
        @(posedge clk) MAR <= IR[26:0];
        @(posedge clk) MBR <= data;
```

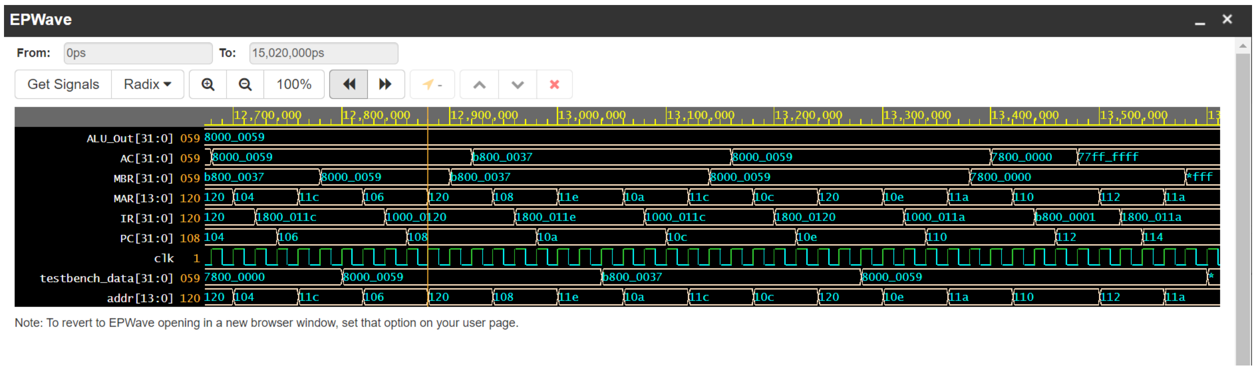
```
      @(posedge clk) AC <= MBR;
      // #1
      // $display("%b AC %h %d\n", AC, MAR, AC[11:0]);
end
4'b0011: begin    // store
      @(posedge clk) MAR <= IR[26:0];
      @(posedge clk) MBR <= AC;
      @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
end
4'b0100: begin  // clear
    @(posedge clk) AC <= 0;

end
4'b0101: begin  // skip
    //$display("%h THIS SKIPS!!!\n", MAR);
    @(posedge clk)
    if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 2;
    else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 2;
    else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 2;
    else if(IR[11:10]==2'b01 && AC[26:0] == 27'b111111111111111111111111111) PC <= PC + 2;
    // $display("Hello World! %b %b\n", AC, IR[11:10]);
end
4'b0110: begin // jump
  @(posedge clk) PC <= IR[26:0];
end

4'b0111: begin // subtract
@(posedge clk) MAR <= IR[26:0];
@(posedge clk) MBR <= data;
@(posedge clk) ALU_Sel <= 'b010; A <= AC; B <= MBR;
@(posedge clk) AC <= ALU_Out;
end

4'b1000: begin // and
@(posedge clk) MAR <= IR[26:0];
@(posedge clk) MBR <= data;
@(posedge clk) ALU_Sel <= 'b000; A <= AC; B <= MBR;
@(posedge clk) AC <= ALU_Out;
end

4'b1001: begin // or
@(posedge clk) MAR <= IR[26:0];
@(posedge clk) MBR <= data;
@(posedge clk) ALU_Sel <= 'b100; A <= AC; B <= MBR;
@(posedge clk) AC <= ALU_Out;
end

4'b1000: begin // not
@(posedge clk) AC <= ~AC;
```

```
          end

        endcase

        end
        1'b1: begin          // Immediate addressing

        case(IR[30:27])
          4'b0000: begin  // addi
                @(posedge clk) AC <= AC + IR[26:0];
          end

          4'b0111: begin // subi
            @(posedge clk) AC <= AC - IR[26:0];
          end

          4'b1000: begin // andi
            @(posedge clk) AC <= AC & IR[26:0];
          end

          4'b1001: begin // ori
            @(posedge clk) AC <= AC | IR[26:0];
          end

        endcase




        end
        endcase


    end



    @(posedge clk) MAR <= 'h10D; we <= 0; cs <= 1; oe <= 1;

    @(posedge clk)

   #20 $finish;
  end

endmodule
```

Verilog Waveforms



Note: To revert to EPWave opening in a new browser window, set that option on your user page.



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

ISA Operations Supported by Our Architecture:

Operation	Microoperations	Description
add X	MAR ← X # load X into MAR MBR ← M[MAR] # load value stored at address X into MBR AC ← AC + MBR # add value in AC with MBR value and store it back into AC	Adds value in AC with the value at address X and stores the result into AC
subtract X	MAR ← X MBR ← M[MAR] AC ← AC - MBR	Subtracts the value at address X from the value in AC and stores the re- sult into AC

and X	MAR ← X
	MBR ← M[MAR]
	AC ← AC and MBR
		Performs Boolean ”and” on the value in AC and the value at address X and stores the result into AC

or X	MAR ← X MBR ← M[MAR] AC ← AC or MBR	Performs Boolean ”or” on the value in AC and the value at address X and stores the result into AC
not	AC ← not AC	Performs Boolean ”not” on the value in AC and stores the result into AC
halt		Ends the program (Implied that this loops the halt instruction which does nothing except stall)
load X	MAR ← X # load X (address) into MAR MBR ← M[MAR] # load value stored at address into MBR AC ← MBR # load value in MBR into AC	Loads the value from address X into the AC
store X	MAR ← X # load address into MAR MBR ← AC # load AC value into MBR M[MAR] ← MBR # writes MBR value into the Memory of address indicated by the MAR	Stores the current value from the AC into address X
clear	AC ← 0	Writes 0 into AC
nop	N/A	Do nothing
skip C	If (C), PC ← (PC) + 1	Skips the next instruction based on C. if (C) is: 0: Skips if AC < 0 2: Skips if AC = 0 4: Skips if AC > 0
jump X	PC ← X	Jumps to address X

Each operation within the table has been implemented.

Design Specifications ID 00019

ID mod 32	Word size	Main Memory Size	Main Memory Organization	Max # of bits to be used by L1 cache	Additional Addressing Mode
19	32 bits	1Gi bits	256Mi x 16	600,000	Immediate

- 256Mix16 means that we have to implement the data split where one stack of memory represents the most significant 16 bits of our 32 bit string, and one stack represents the least significant 16.
    - Each of the stacks should be 256Mi long.

- That makes the cache itself 32 bits wide, but < 600,000 bits long.

- Notes on memory formatting:
    - $Main\ memory\ size\ is\ 1Gi\ \rightarrow\ 4\ *\ 256Mi,\ so\ we\ need\ 4\ rows\ of\ chips$
        - 256Mi / 16 = 16777216 bits, $\log_2(16777216) = 24$ bits
            - So we need 2^{24} rows of memory per RAM chip
    - $Word\ size\ is\ 32\ bits$
    - $L1\ Cache\ Lines\ =\ Size\ of\ Cache/\ size\ of\ line\ =\ 600K\ /\ 16\ =\ <\ 37,500\ cache\ lines$
        - That makes 2^{14} lines of cache, to be safe, since cache index sizes are powers of 2.

Immediate Addressing:

i.e. addi, subi (not in MIPS, but extra credit), andi, ori.

```
Addi X:
        AC ← AC + X
Subi X:
        AC ← AC - X
Andi X:
        AC ← AC AND X
Ori X:
        AC ← AC OR X
```

Design 1a and 1b Instruction format:

[1 bit I-code][4 bit opcode][27 bit data]

I-code:

0 -- direct addressing
1 -- Immediate addressing

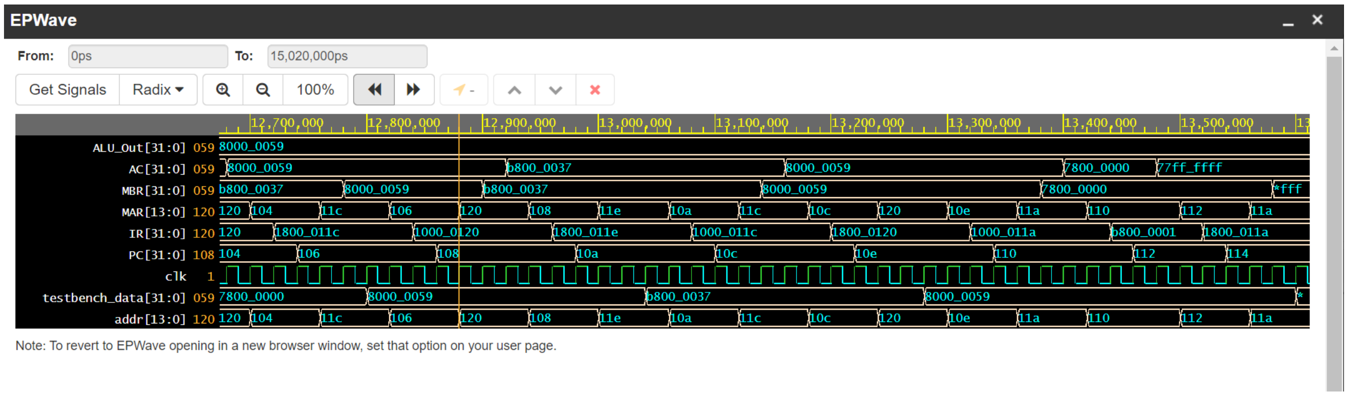
Opcodes:

0000: add / addi
0001: halt
0010: load
0011: store
0100: clear
0101: skip
0110: jump
0111: sub / subi
1000: and / andi
1001: or / ori
1010: not
1111: nop

- Justification: given we have 13 operations, one bit can represent the addressing mode, 4 bits can account for all operations, and leave room for 27 bits to store the address of the given value to operate.
    - Given the size of the address bits, this leaves us with a larger range of values to read from the main memory.

Benchmark Programs

For our Benchmark Program, we implemented the calculation of the eleventh Fibonacci number. Our code works, as you can see in this waveform:



Here at MAR of 120 (the value of the return variable), the value in AC is 0x059, which is 89 in decimal. Since 89 is the 11th Fibonacci number, this code successfully computed that result.

The exact code we used for the benchmark is as follows:

```
100 Load A        // AC = A
102 Add B         // AC = A + B
104 Store temp    // temp = AC = F_2
106 Load B        // AC = B
108 Store A       // A = B = F_1
10A Load temp     // AC = temp
10C Store B       // B = temp = F_2
10E Load control  // AC = control
110 Subi 1        // control--
112 Store control // AC -> control
114 Skip 01 control // 01 specifies skip on equal
116 Jump 100
118 Halt
11A control Dec 9
11C temp Dec 0
11E A Dec 0       // F_0
120 B Dec 1       // F_1
```

This assembly code can be transformed into the following machine code:

```
1000011E        // Load A
00000120        // Add B
1800011C        // Store temp
10000120        // Load B
1800011E        // Store A
1000011C        // Load temp
18000120        // Store B
1000011A        // Load control
B8000001        // Subi 1
1800011A        // Store control
28000400        // Skip 01
30000100        // Jump 100
08000000        // Halt
78000009        // control Dec 10
78000000        // temp Dec 0
78000000          // A Dec 0
78000001        // B Dec 1
```

This code represents the more efficient algorithm we developed to find the 11th Fibonacci number using immediate instructions (subi). We also created a version that uses only direct instructions, in compliance with Design 1, which is as follows:

```
100 Load A       // AC = A
102 Add B        // AC = A + B
104 Store temp    // temp = AC = F_2
106 Load B       // AC = B
108 Store A      // A = B = F_1
10A Load temp     // AC = temp
10C Store B      // B = temp = F_2
10E Load control  // AC = control
110 Sub One       // control--
112 Store control // AC -> control
114 Skip 01 control // 01 specifies skip on equal
116 Jump 100
118 Halt
11A control Dec 10
11C temp Dec 0
11E A Dec 0      // F_0
120 B Dec 1      // F_1
122 One Dec 1
```

```
1000011E
00000120
1800011C
10000120
1800011E
1000011C
18000120
1000011A
38000122
1800011A
28000400
30000100
08000000
78000009
78000000
78000000
78000001
78000001
```

Executing Benchmarks Process

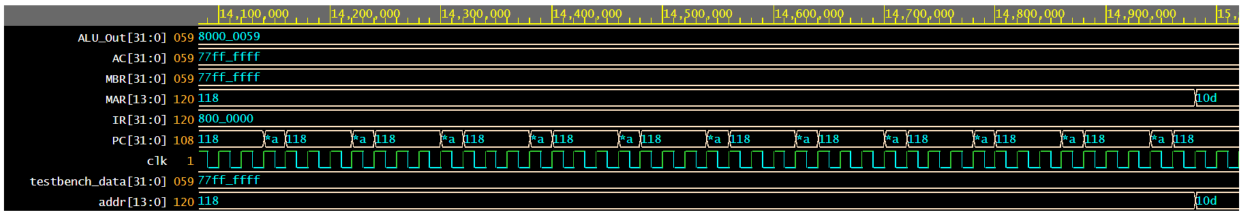
The process of executing the benchmarks mainly involves an ALU selection bit. For our CPU, we loop through the fetch, decode, etc. processes and use the IR to determine which set of instructions to follow.

We used both Icarus Verilog and EDAPlayground to run the test benches that simulated our Benchmarks. Both compilers showed the same results, but EDAPlayground was helpful with its timing diagrams to test performance.

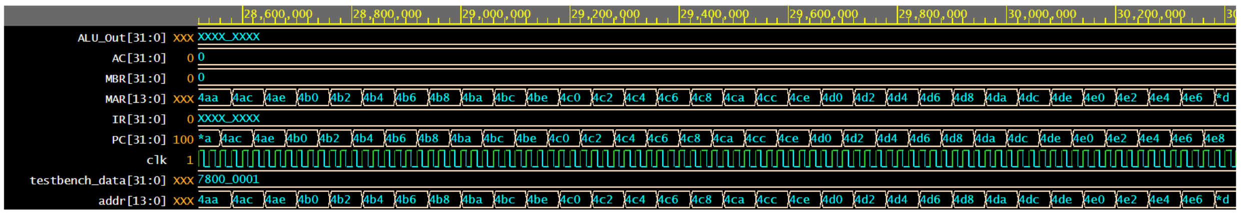
Performance Comparisons

The performance of the processor without the cache runs circles around the performance of the processor with the cache. The version with the cache takes too much time to set up the cache itself and struggles to maintain the values in the cache on time. Because of this, the cached version stalls much more frequently than the non-cached version.

Non-cached:



Cached:


28,600,000
28,800,000
29,000,000
29,200,000
29,400,000
29,600,000
29,800,000
30,000,000
30,200,000
ALU_Out[31:0] xxx xxxx_xxxx
AC[31:0] 0 0
MBR[31:0] 0 0
MAR[13:0] xxx 4aa 4ac 4ae 4b0 4b2 4b4 4b6 4b8 4ba 4bc 4be 4c0 4c2 4c4 4c6 4c8 4ca 4cc 4ce 4d0 4d2 4d4 4d6 4d8 4da 4dc 4de 4e0 4e2 4e4 4e6 *d
IR[31:0] 0 xxxx_xxxx
PC[31:0] 100 *a 4ac 4ae 4b0 4b2 4b4 4b6 4b8 4ba 4bc 4be 4c0 4c2 4c4 4c6 4c8 4ca 4cc 4ce 4d0 4d2 4d4 4d6 4d8 4da 4dc 4de 4e0 4e2 4e4 4e6 4e8
clk 1
testbench_data[31:0] xxx 7800_0001
addr[13:0] xxx 4aa 4ac 4ae 4b0 4b2 4b4 4b6 4b8 4ba 4bc 4be 4c0 4c2 4c4 4c6 4c8 4ca 4cc 4ce 4d0 4d2 4d4 4d6 4d8 4da 4dc 4de 4e0 4e2 4e4 4e6 *d

Note: To revert to EPWave opening in a new browser window, set that option on your user page

The waveform provided by EDAPlayground shows that the non-cached version takes 15,000,000 time units to run, while the cached version ran two times slower, at about 30,200,000 total time units. This slowdown is due to frequent misses, requiring the processor to stall while it reads data from the main memory most of the time when it tries to access the cache.

Contributions

We all met up multiple times to discuss and work together. We discussed ways to debug and fix errors we ran into. We didn't individually divide up the workload, rather we all worked on the same thing and discussed it from there. We shared the code on GitHub so it was easier to share and edit the code.

In terms of the report, we added what we found and accumulated our research and logic.

- Seth:
    - Worked on the ALU, decoder, RAM, large RAM, cache, and CPU files
    - Wrote assembly code for the Fib_{11} calculation and converted it to hexadecimal machine code
    - Developed the instruction format
- Jeffrey:
    - Worked on documentation, code organization, code refactoring, and unit testing of all computer architecture implementations
- Izzy:
    - Consulted with Seth about coding implementation, operation, bug fixing, and solutions to deep-rooted technical issues
- Yash:
    - Worked on the project report
    - Attempted ALU coding
    - Ran simulated code testing
- Christopher:
    - Setup and organized repository directories
    - Added to and cleaned up project report
    - Spent time running simulation code testing

References

- https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html
- https://www.youtube.com/watch?v=0Pbw-GS7-x0&ab_channel=IntellCity
- https://www.youtube.com/watch?v=ZPaQTN6Xp0c&ab_channel=GEEK
- https://booksite.elsevier.com/9780124077263/downloads/COD_5e_Greencard.pdf
- https://submitty.cs.rpi.edu/courses/f23/csci2500/course_material/Slides/Appendix-B_Verilog-slides.pdf
- https://stackoverflow.com/questions/32473143/disable-statement-not-killing-other-block-in-the-fork-statement
- https://stackoverflow.com/questions/29628469/how-to-store-input-into-reg-from-wire-in-verilog

- https://stackoverflow.com/questions/61740386/mux-in-iverilog-unable-to-bind-parameter-cannot-evaluate-genvar-expression-erro
- https://stackoverflow.com/questions/16424726/what-is-the-difference-between-verilog-and
- https://stackoverflow.com/questions/13506652/create-an-a-x-b-expanded-grid-memory-efficiently
- https://stackoverflow.com/questions/20317820/icarus-verilog-dump-memory-array-dumpvars
- https://stackoverflow.com/questions/36058714/instancing-modules-verilog/36059317#36059317
- https://stackoverflow.com/questions/24591396/access-top-level-resources-outside-of-hierarchy
- https://stackoverflow.com/questions/5265467/how-to-use-display-without-initial-or-always-blocks
- https://stackoverflow.com/questions/74715850/how-to-declare-integer-variable-in-verilog-to-keep-track-of-a-value-to-be-used-i
- https://stackoverflow.com/questions/16424726/what-is-the-difference-between-verilog-and
- https://web.engr.oregonstate.edu/~traylor/ece474/beamer_lectures/verilog_operators.pdf
- https://www.chipverify.com/verilog/verilog-data-types
- https://hardwarebee.com/ultimate-guide-verilog-test-bench/
- https://learn.zybooks.com/zybook/RPICSCI2500KuzminFall2023
- https://submitty.cs.rpi.edu/courses/f23/csci2500/course_material/Slides/Appendix-B_Verilog-slides.pdf
- https://submitty.cs.rpi.edu/courses/f23/csci2500/course_materials