Assignment-2

Sarang Pramod Choudalwar |VLSI Architecture | 2020ht01012 (2020ht01012@wilp.bits-pilani.ac.in)

Problem statement: -

A 32 bit single cycle mips processor needs to be designed for executing the following predefined instructions.

```
 0 add add $s1,$s2,$s3                  //$s1 = $s2 + $s3
4 load word lw $s1,20($s2)              //$s1 = Memory[$s2 + 20]
8 store word sw $s1,20($s2)             //Memory[$s2 + 20] = $s1
12 nor nor $s1,$s2,$s3                  //$s1 = ~ ($s2 | $s3)
16 branch on equal beq $s1,$s2,5      //if ($s1 == $s2), execute nop
```

Designers thoughts/Observations/Assumptions:-

- Problem statement specify all instruction to be executed to be of type either r or I type instruction- There is no unconditional jump instruction is present, hence the hardware design of the jump instruction is being skipped in the Verilog implementation.
- The Data path and control unit considered for this problem statement is given below-

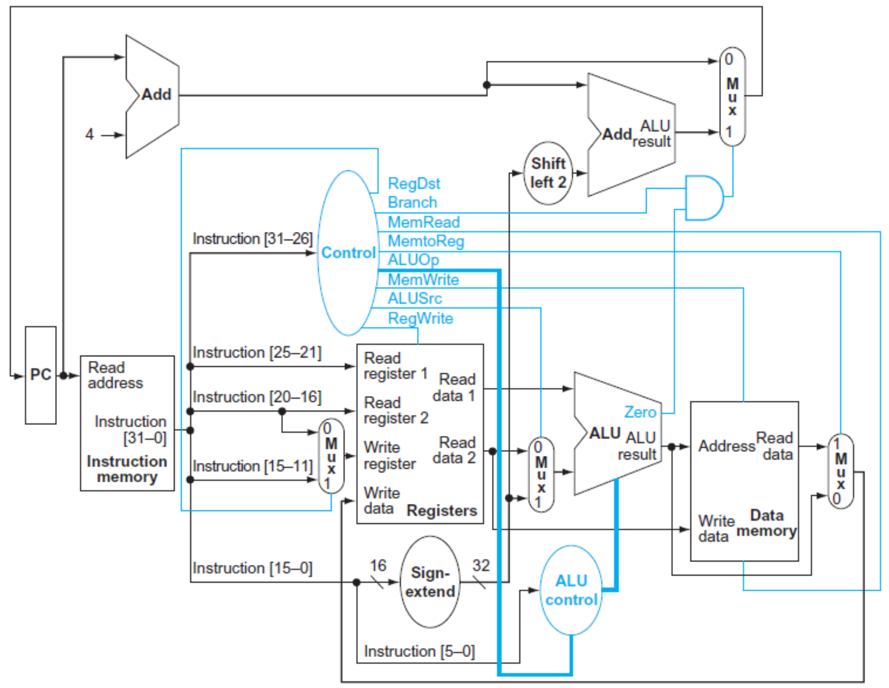
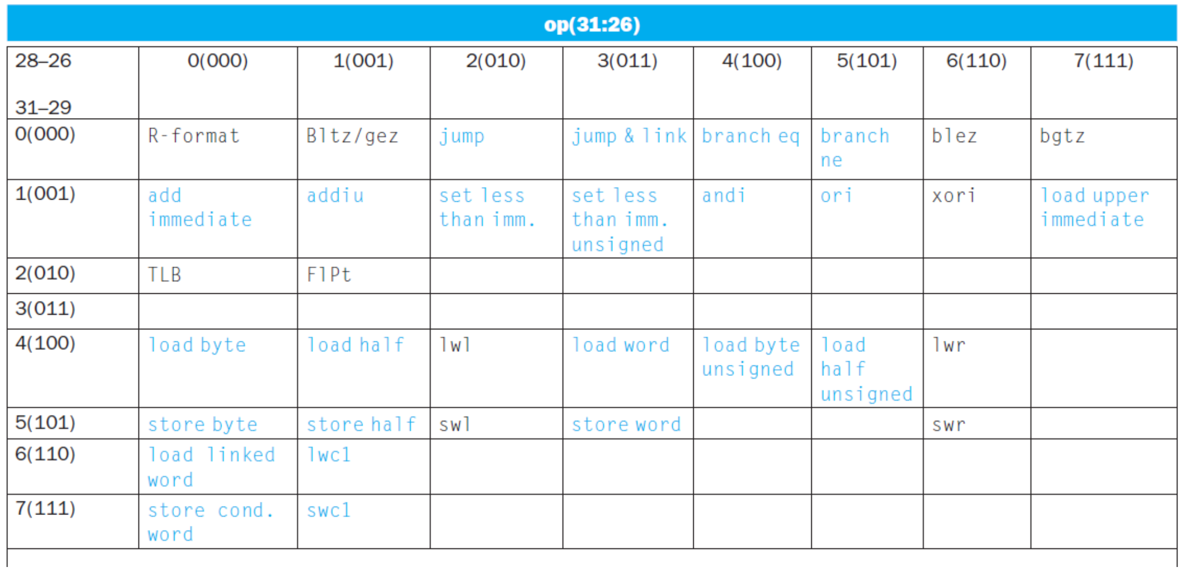


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

- In this architecture all the instruction are of 32 bit and contains all the information needed for executing the operation of the system.
- Each instruction needs to be manually decoded and then then the logic for implementation needs to be thought of.
- The table below shows the instruction format for all the possible instruction format.

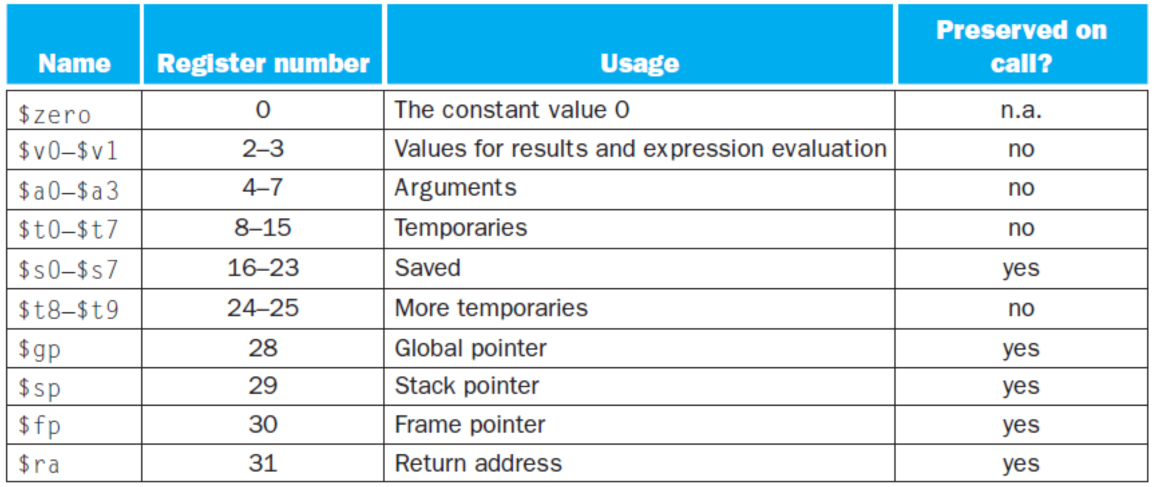
Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- The instruction consist of opcode source register, target register, destination register(only r-type), shift amount (only r-type), function field (only r-type), address/immediate value (only I-format), target address(only j-format).
- The values of the opcode can be decoded using the below table-



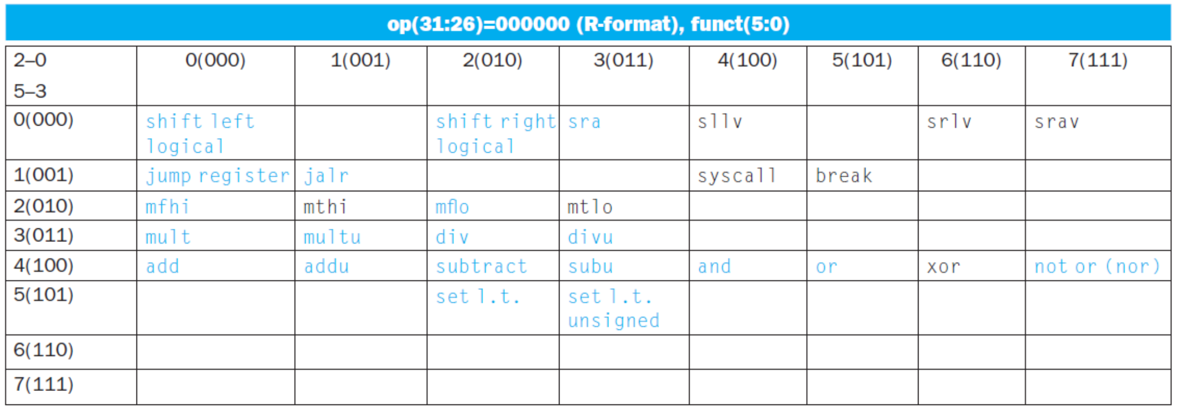
op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

- The value of registers can be decoded using the below table-



Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

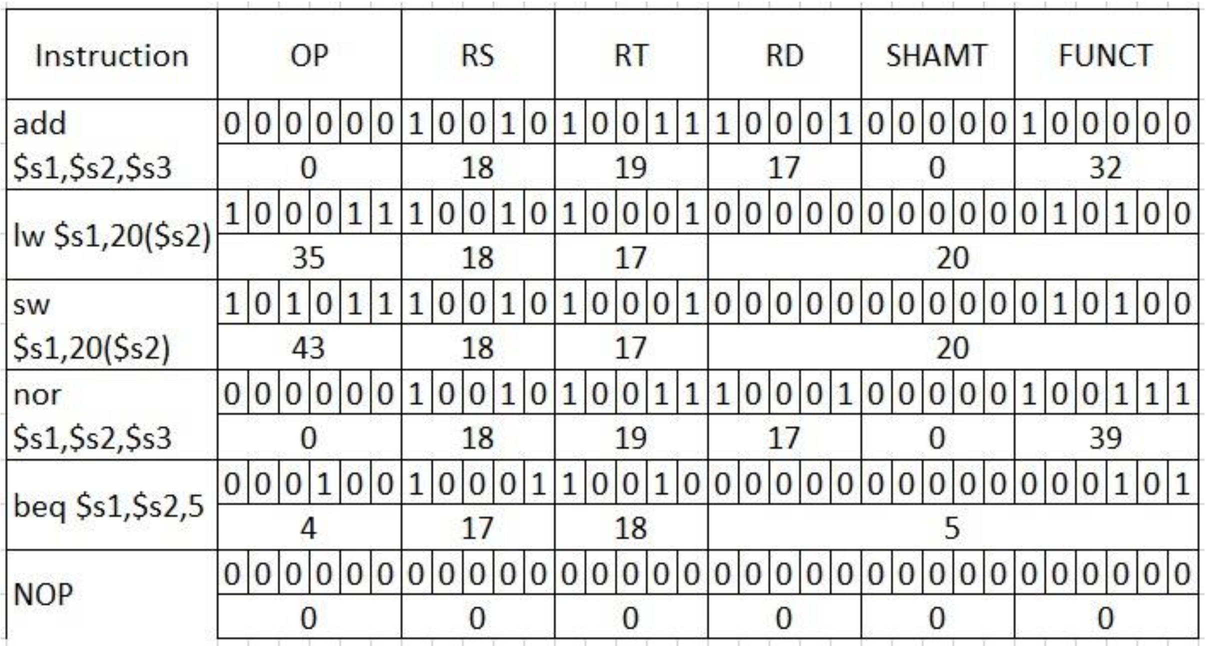
- The value of function field can be decoded using the below table-



op(31:26)=000000 (R-format), funct(5:0)

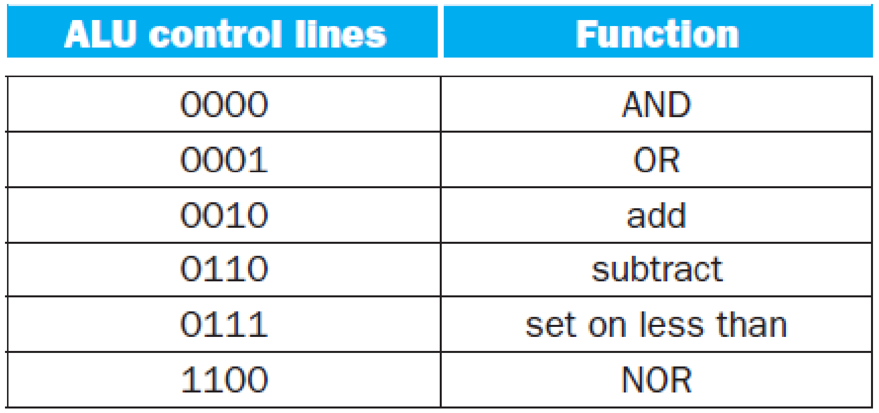
2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

- The given instruction is decoded into different parts using below table.



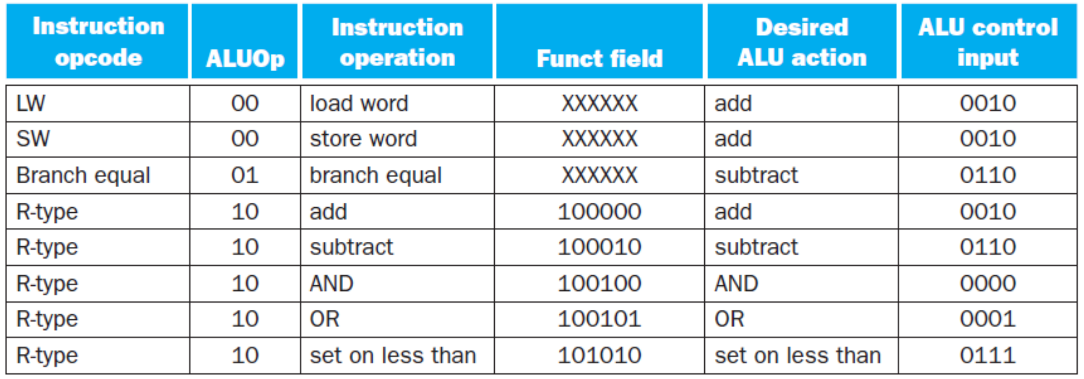
Instruction	OP	RS	RT	RD	SHAMT	FUNCT
add	0 0 0 0 0 0	1 0 0 1 0	1 0 0 1 1	1 0 0 0 1	0 0 0 0 0	1 0 0 0 0 0
$s1,$s2,$s3	0	18	19	17	0	32
lw $s1,20($s2)	1 0 0 0 1 1	1 0 0 1 0	1 0 0 0 1	0 0 0 0 0	0 0 0 0 0	0 1 0 1 0 0
	35	18	17	20		
sw	1 0 1 0 1 1	1 0 0 1 0	1 0 0 0 1	0 0 0 0 0	0 0 0 0 0	0 1 0 1 0 0
$s1,20($s2)	43	18	17	20		
nor	0 0 0 0 0 0	1 0 0 1 0	1 0 0 1 1	1 0 0 0 1	0 0 0 0 0	1 0 0 1 1 1
$s1,$s2,$s3	0	18	19	17	0	39
beq $s1,$s2,5	0 0 0 1 0 0	1 0 0 0 1	1 0 0 1 0	0 0 0 0 0	0 0 0 0 0	0 0 0 1 0 1
	4	17	18	5		
NOP	0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0
	0	0	0	0	0	0

- The ALU required for this implementation performs function according to the below control lines



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

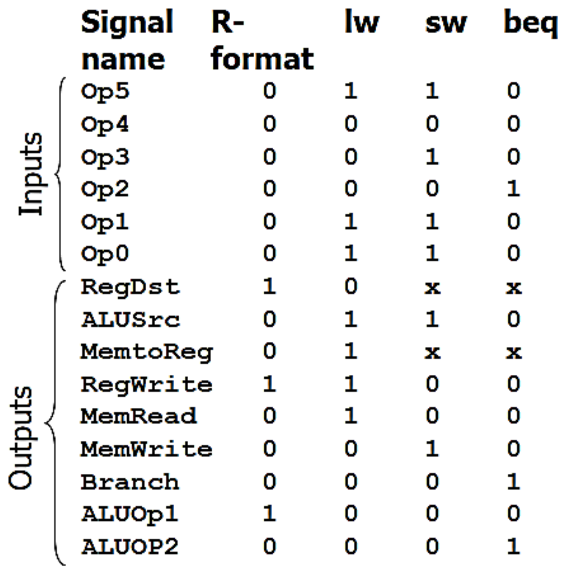
- The Table below shows the how the alu control input is driven using the alu op and the function field



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

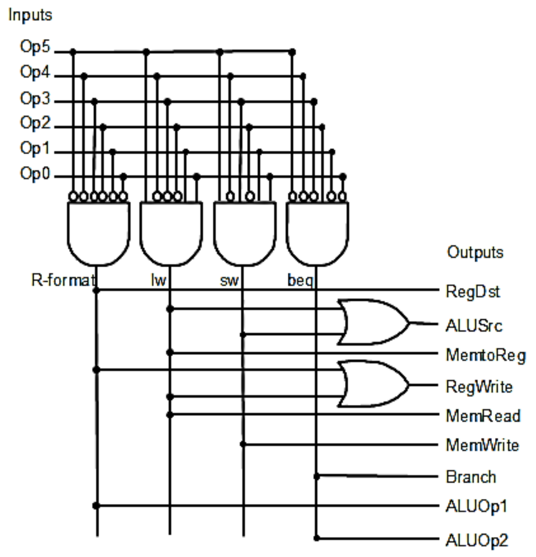
- The Main controller unit is implemented using the following diagram and truth table

Implementation: Main Control Block



	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

- The instruction and data memory are assumed to capable of writing and reading the data in the same cycle.
- Individual unit processing time is considered too small to be negligible.
- The code is not synthesized and its assumed that it is only for the simulation purpose.

Codes of different modules –

1. Top module for 32 bit mips processor

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - mips32.sv
* Function - Act as a top layer for 32 bit
*  Single Cycle MIPS processor.
* Language Used - System Verilog
*****************************************/

`timescale 1ns/1ns

`include "data_memory.sv"
`include "reg_file.sv"
`include "instruction_memory.sv"
`include "alu.sv"
`include "main_controller.sv"
`include "mux32.sv"
`include "mux5.sv"
`include "shift_left2.sv"
`include "sign_extender.sv"
`include "alu_adder32.sv"
`include "alu_controller.sv"

module mips32(input clk,
             input reset,
              output[31:0] alu_out);

  reg[31:0] pc_current;
  reg[31:0] pc_next;
  wire[31:0] wire_pc_next;
  wire[31:0] wire_pc_adder_out;
  wire[31:0] wire_instruction;
  wire[5:0] wire_opcode;
  wire      wire_rst;
  wire      wire_reg_dest;
  wire      wire_alu_src;
  wire      wire_mem_to_reg;
  wire      wire_reg_write;
  wire      wire_mem_read;
```

```verilog
  wire       wire_mem_write;
  wire       wire_branch;
  wire[1:0] wire_alu_op;
  wire[5:0] wire_op;
  wire[4:0] wire_rs;
  wire[4:0] wire_rt;
  wire[4:0] wire_rd;
  wire[15:0] wire_instruction_remain;
  wire[31:0] wire_read_reg_data_1;
  wire[31:0] wire_read_reg_data_2;
  wire[4:0]  wire_mux5_out;
  wire[5:0]  wire_function;
  wire[31:0] wire_sign_extended_value;
  wire[31:0] wire_mux32_unit1_out;
  wire[3:0]  wire_alu_ctrl;
  wire[31:0] wire_alu_result;
  wire       wire_zero;
  wire[31:0] wire_data_mem_read_data;
  wire[31:0] wire_mux32_unit2_out;
  wire[31:0] wire_shifted_value;
  wire[31:0] wire_alu_adder32_out;
  wire       wire_branch_if_zero;



  always @(posedge clk)
    begin
      if(reset)
        begin
          pc_current = 32'd0;
          pc_next    = pc_current+4;
        end
      else
        begin
          pc_current =pc_next;
          pc_next = pc_current+4;
        end
    end

  instruction_memory instruction_memory_unit(.read_address(pc_current),
                                              .instruction(wire_instruction),
                                              .reset(reset),
                                              .clk(clk));

  assign wire_op = wire_instruction[31:26];
```

```
assign wire_rs = wire_instruction[25:21];
assign wire_rt = wire_instruction[20:16];
assign wire_rd = wire_instruction[15:11];
assign wire_instruction_remain = wire_instruction[15:0];

main_controller main_controller_unit(.opcode(wire_op),
                   .rst(reset),
                   .reg_dest(wire_reg_dest),
                   .alu_src(wire_alu_src),
                   .mem_to_reg(wire_mem_to_reg),
                   .reg_write(wire_reg_write),
                   .mem_read(wire_mem_read),
                   .mem_write(wire_mem_write),
                   .branch(wire_branch),
                   .alu_op(wire_alu_op),
                   .clk(clk));



mux5 mux5_unit(.mux_out(wire_mux5_out),
               .a(wire_rt),
               .b(wire_rd),
               .sel(wire_reg_dest));

register_file register_file_unit(.clk(clk),
                      .rst(reset),
                      .reg_write(wire_reg_write),
                      .write_reg_addr(wire_mux5_out),
                      .write_reg_data(wire_mux32_unit2_out),
                      .read_reg_addr_1(wire_rs),
                      .read_reg_data_1(wire_read_reg_data_1),
                      .read_reg_addr_2(wire_rt),
                      .read_reg_data_2(wire_read_reg_data_2)
                );

assign wire_function = wire_instruction_remain[5:0];

sign_extender sign_extender_unit(.invalue(wire_instruction_remain),
                      .outvalue(wire_sign_extended_value));

mux32 mux32_unit_1(.mux_out(wire_mux32_unit1_out),
                   .a(wire_read_reg_data_2),
                   .b(wire_sign_extended_value),
                   .sel(wire_alu_src));
```

```
alu_controller alu_controller_unit(.alu_op(wire_alu_op),
                                   .func_code(wire_function),
                                   .alu_ctrl(wire_alu_ctrl),
                                   .reset(reset),
                                   .clk(clk));

alu alu_unit(.main_alu_input_a(wire_read_reg_data_1),
             .main_alu_input_b(wire_mux32_unit1_out),
             .alu_control(wire_alu_ctrl),
             .result(wire_alu_result),
             .zero(wire_zero),
             .reset(reset),
             .clk(clk));

data_memory data_memory_unit(.data_memory_read_data(wire_data_mem_read_data),
                             .data_memory_address(wire_alu_result),
                             .data_memory_write_data(wire_read_reg_data_2),
                             .mem_write(wire_mem_write),
                             .mem_read(wire_mem_read),
                             .rst(reset),
                             .clk(clk));


mux32 mux32_unit_2(.mux_out(wire_mux32_unit2_out),
                   .a(wire_alu_result),
                   .b(wire_data_mem_read_data),
                   .sel(wire_mem_to_reg));

shift_left2 shift_left2_unit(.shift_out(wire_shifted_value),
                             .value(wire_sign_extended_value));

alu_adder32 alu_adder32_unit(.alu_out(wire_alu_adder32_out),
                             .alu_in_pc_next(pc_next),
                             .alu_in_shift_2(wire_shifted_value));

and and_unit(wire_branch_if_zero,wire_branch,wire_zero);

mux32 mux32_unit3(.mux_out(wire_pc_next),
                  .a(pc_next),
                  .b(wire_alu_adder32_out),
                  .sel(wire_branch_if_zero));

always @(*)
  begin
  pc_next <= wire_pc_next;
```

```
    end

  assign alu_out = wire_alu_result;

endmodule
```

2.Instruction memory

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - instruction_memory.sv
* Function - 32 bits MIPS Instruction memory
* implementation.
* using different read and write port.
* Language Used - System Verilog
*****************************************/

//instruction memory

`timescale 1ns/1ns

module instruction_memory(output[31:0] instruction,
                          input[31:0] read_address,
                          input reset,
                          input clk
                         );

  reg[31:0] instruction;
  wire[7:0] temp_addr;
  reg[31:0] data[63:0];
  integer i;

  always @(posedge clk or reset)
    begin
      if(reset)
        begin
          data[0]=32'b00000010010100111000100000100000; //add $s1,$s2,$s3
          data[1]=32'b10001110010100010000000000010100; //lw $s1,20($s2)
          data[2]=32'b10101110010100010000000000010100; //sw $s1,20($s2)
          data[3]=32'b00000010010100111000100000100111; //nor $s2,s2,s3
          data[4]=32'b00010010001100100000000000000101; //beq $s1,$s2,5
          for(i=5;i<64;i=i+1)
            begin
              data[i] = 32'b00000000000000000000000000000000;
            end

        end
      else
        instruction = data[read_address[7:2]];
    end
endmodule
```

3.Main controller-

```
/******************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - alu.sv
* Function - Main controller Implementation
* Responsible for generating diffent control
* signals required for entire system to work.
* Language Used - System Verilog
******************************************/


`timescale 1ns/1ns

module main_controller(input[5:0] opcode,
                        input rst,
                        output reg_dest,
                       output alu_src,
                       output mem_to_reg,
                       output reg_write,
                       output mem_read,
                       output mem_write,
                       output branch,
                        output[1:0] alu_op,
                        input clk
```

```
                    );
reg reg_dest;
reg alu_src;
reg mem_to_reg;
reg reg_write;
reg mem_read;
reg mem_write;
reg branch;
reg[1:0] alu_op;

always@(*)
  begin
    if(rst)
      begin
          reg_dest    =  1'b0;
          alu_src     =  1'b0;
          mem_to_reg  =  1'b0;
          reg_write   =  1'b0;
          mem_read    =  1'b0;
          mem_write   =  1'b0;
          branch      =  1'b0;
          alu_op      =  2'b00;
      end
    else
    case (opcode)
      6'b000000:
        begin
          reg_dest    =  1'b1;
          alu_src     =  1'b0;
          mem_to_reg  =  1'b0;
          reg_write   =  1'b1;
          mem_read    =  1'b0;
          mem_write   =  1'b0;
          branch      =  1'b0;
          alu_op      =  2'b10;
        end
      6'b100011:
        begin
          reg_dest    =  1'b0;
          alu_src     =  1'b1;
          mem_to_reg  =  1'b1;
          reg_write   =  1'b1;
          mem_read    =  1'b1;
          mem_write   =  1'b0;
          branch      =  1'b0;
          alu_op      =  2'b00;
        end
      6'b101011:
        begin
          reg_dest    =  1'b0;
          alu_src     =  1'b1;
          mem_to_reg  =  1'b0;
          reg_write   =  1'b0;
          mem_read    =  1'b0;
          mem_write   =  1'b1;
          branch      =  1'b0;
          alu_op      =  2'b00;
        end
      6'b000100:
          begin
          reg_dest    =  1'b0;
          alu_src     =  1'b0;
          mem_to_reg  =  1'b0;
          reg_write   =  1'b0;
          mem_read    =  1'b0;
          mem_write   =  1'b0;
          branch      =  1'b1;
          alu_op      =  2'b01;
        end
      default:
        begin
          reg_dest    =  1'b0;
          alu_src     =  1'b0;
          mem_to_reg  =  1'b0;
          reg_write   =  1'b0;
          mem_read    =  1'b0;
          mem_write   =  1'b0;
          branch      =  1'b0;
          alu_op      =  2'b00;
        end
    endcase
  end
endmodule
```

4. mux with bit input bit width of 5

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - mux5.sv
* Function - multiplexer with 5 bit bus width.
* Language Used - System Verilog
*****************************************/

`timescale 1ns/1ns

module mux5(output[4:0] mux_out,
             input[4:0] a,
             input[4:0] b,
             input sel);
  assign mux_out=(sel==0)?a:b;
endmodule
```

5.Register file

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - reg_file.sv
* Function - Register file implementation
* with 2 read port and 1 write port.
* (32 different registers with 32 bit width
* is considered.)
* using different read and write port.
* Language Used - System Verilog
*****************************************/

`timescale 1ns/1ns

module register_file( input clk,
                     input rst,
                     //write port
                     input reg_write,
                     input[4:0] write_reg_addr,
                     input[31:0] write_reg_data,
                     //read port 1
                     input[4:0] read_reg_addr_1,
                     output[31:0] read_reg_data_1,
                     //read port 2
```

```
                    input[4:0] read_reg_addr_2,
                    output[31:0] read_reg_data_2
                   );

  reg[31:0] reg_array[31:0]; //32 registers with 32 bit width

  always @(*)
    begin
      if(rst)
        begin
          reg_array[0] <= 32'b0;
          reg_array[1] <= 32'b0;
          reg_array[2] <= 32'b0;
          reg_array[3] <= 32'b0;
          reg_array[4] <= 32'b0;
          reg_array[5] <= 32'b0;
          reg_array[6] <= 32'b0;
          reg_array[7] <= 32'b0;
          reg_array[8] <= 32'b0;
          reg_array[9] <= 32'b0;
          reg_array[10] <= 32'b0;
          reg_array[11] <= 32'b0;
          reg_array[12] <= 32'b0;
          reg_array[13] <= 32'b0;
          reg_array[14] <= 32'b0;
          reg_array[15] <= 32'b0;
          reg_array[16] <= 32'b0;
          reg_array[17] <= 32'h30; //s1 = 30 hex
          reg_array[18] <= 32'h10; //s2 = 10 hex
          reg_array[19] <= 32'h20; //s3 = 20 hex
          reg_array[20] <= 32'b0;
          reg_array[21] <= 32'b0;
          reg_array[22] <= 32'b0;
          reg_array[23] <= 32'b0;
          reg_array[24] <= 32'b0;
          reg_array[25] <= 32'b0;
          reg_array[26] <= 32'b0;
          reg_array[27] <= 32'b0;
          reg_array[28] <= 32'b0;
          reg_array[29] <= 32'b0;
          reg_array[30] <= 32'b0;
          reg_array[31] <= 32'b0;
        end
      else begin
        if(reg_write)
```

```
          begin
            reg_array[write_reg_addr] = write_reg_data;
          end
      end
    end

  assign read_reg_data_1 = (read_reg_addr_1 == 0)? 32'b0 : reg_array[read_reg_addr_1];
  assign read_reg_data_2 = (read_reg_addr_2 == 0)? 32'b0 : reg_array[read_reg_addr_2];

endmodule
```

6. Sign extender

```
/******************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - sign_extender.sv
* Function - Sign extension of 16 bit value
* to 32 bit value.
* Language Used - System Verilog
******************************************/
`timescale 1ns/1ns

module sign_extender(input[15:0] invalue,
                     output[31:0] outvalue);
  assign outvalue = $signed(invalue);
endmodule
```

7.Mux with 32 bit input capacity

```
/******************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - mux32.sv
* Function - 32 bit multiplexer.
* Language Used - System Verilog
*****************************************/

`timescale 1ns/1ns

module mux32(output[31:0] mux_out,
             input[31:0] a,
             input[31:0] b,
             input sel);
  assign mux_out=(sel==0)?a:b;
endmodule
```

8.Alu controller.

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - alu_controller.sv
* Function -  Alu controller generates the
* - opcodes for alu depending upon which ALU
* performs the operation - (arithmatic/logical)
* Language Used - System Verilog
*****************************************/


`timescale 1ns/1ns

module alu_controller(input[1:0] alu_op,
                   input[5:0] func_code,
                      output[3:0] alu_ctrl,
                     input reset,
                     input clk);

  reg alu_ctrl;
  reg[7:0] alu_control_in;


  always @(*)
    begin
    alu_control_in={alu_op,func_code};
       casex(alu_control_in)
          8'b00xxxxxx: alu_ctrl = 4'b0010; //2 Add lw/sw
          8'b01xxxxxx: alu_ctrl = 4'b0110; //2 sub beq
          8'b10100000: alu_ctrl = 4'b0010; //6 Substract //fun
          8'b10100010: alu_ctrl = 4'b0110; //2 Add //fun
          8'b10100100: alu_ctrl = 4'b0000; //0 AND //fun
          8'b10100101: alu_ctrl = 4'b0001; //1 OR //fun
          8'b10101010: alu_ctrl = 4'b0111; //7 slt //fun
          8'b10100111: alu_ctrl = 4'b1100; //12 nor //fun
    default: alu_ctrl =4'b0000; // should not happen;
      endcase
    end

  always @(reset)
    begin
       alu_ctrl = 4'b0000;
    end
endmodule
```

9.Alu.

```
/******************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - alu.sv
* Function - Implementation logic for 32 bit
* ALU. Support 6 - different type of operation.
* Language Used - System Verilog
*****************************************/
`timescale 1ns/1ns

module alu(input[31:0] main_alu_input_a,
           input[31:0] main_alu_input_b,
           input[3:0] alu_control,
           output[31:0] result,
           output zero,
          input reset,
          input clk);

  reg zero;
  reg result;

  always@(*)
    if(reset)
```

```
      begin
        zero <= 1'b0;
        result <= 32'b0;
      end
    else
      begin
       case (alu_control)
        4'b0000 : result <= main_alu_input_a & main_alu_input_b; // and
        4'b0001 : result <= main_alu_input_a | main_alu_input_b; // or
        4'b0010 : result <= main_alu_input_a + main_alu_input_b; // add
        4'b0110 : result <= main_alu_input_a - main_alu_input_b; // sub
        4'b0111 : result <= main_alu_input_a < main_alu_input_b ? 1'b1:1'b0; //set on less than
        4'b1100 : result <= ~(main_alu_input_a | main_alu_input_b); //nor
        default : result <= 0; //default
       endcase
        zero=(result==0)?1'b1:1'b0;
      end
endmodule
```

10.Data Memory

```
/*******************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - data_memory.sv
* Function - Data memory with possibility
* of reading and writing the data simultaneously
* using different read and write port.
* Language Used - System Verilog
*******************************************/

`timescale 1ns/1ns

module data_memory(output[31:0] data_memory_read_data,
                   input[31:0] data_memory_address,
                   input[31:0] data_memory_write_data,
                   input mem_write,
                   input mem_read,
                   input rst,
                   input clk);
```

```
  reg[31:0] data_memory_read_data;
  reg[31:0] ram[63:0];
  integer i;

  always @(rst or posedge clk )
    begin
      if(rst)
        begin
          for(i=0;i<64;i=i+1)
            begin
              ram[i] <= 10;
            end
        end
    end

  always @(mem_write, mem_read)
    begin
      if(mem_write)
        begin
          ram[data_memory_address[7:2]] <= data_memory_write_data;
        end

      if(mem_read)
        begin
          data_memory_read_data <=ram[data_memory_address[7:2]];
        end
    end
endmodule
```

11. Shift left 2

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - shift_left_2.sv
* Function - shifts input by 2 bit to left
* passes it to output
* Language Used - System Verilog
*****************************************/
`timescale 1ns/1ns

module shift_left2(output[31:0] shift_out,
                   input[31:0] value
              );
  assign shift_out=value<<2;
endmodule
```

12. 32 bit alu adder

```
/*****************************************
* Author - Sarang Pramod Choudalwar
* Subject - VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - alu_adder32.sv
* Function -  32 bit ALU adder
* - used for calculating new pc address
* in case of branch needs to be taken
* Language Used - System Verilog
*****************************************/

`timescale 1ns/1ns

module alu_adder32(output[31:0] alu_out,
                   input[31:0] alu_in_pc_next,
                   input[31:0] alu_in_shift_2);
```

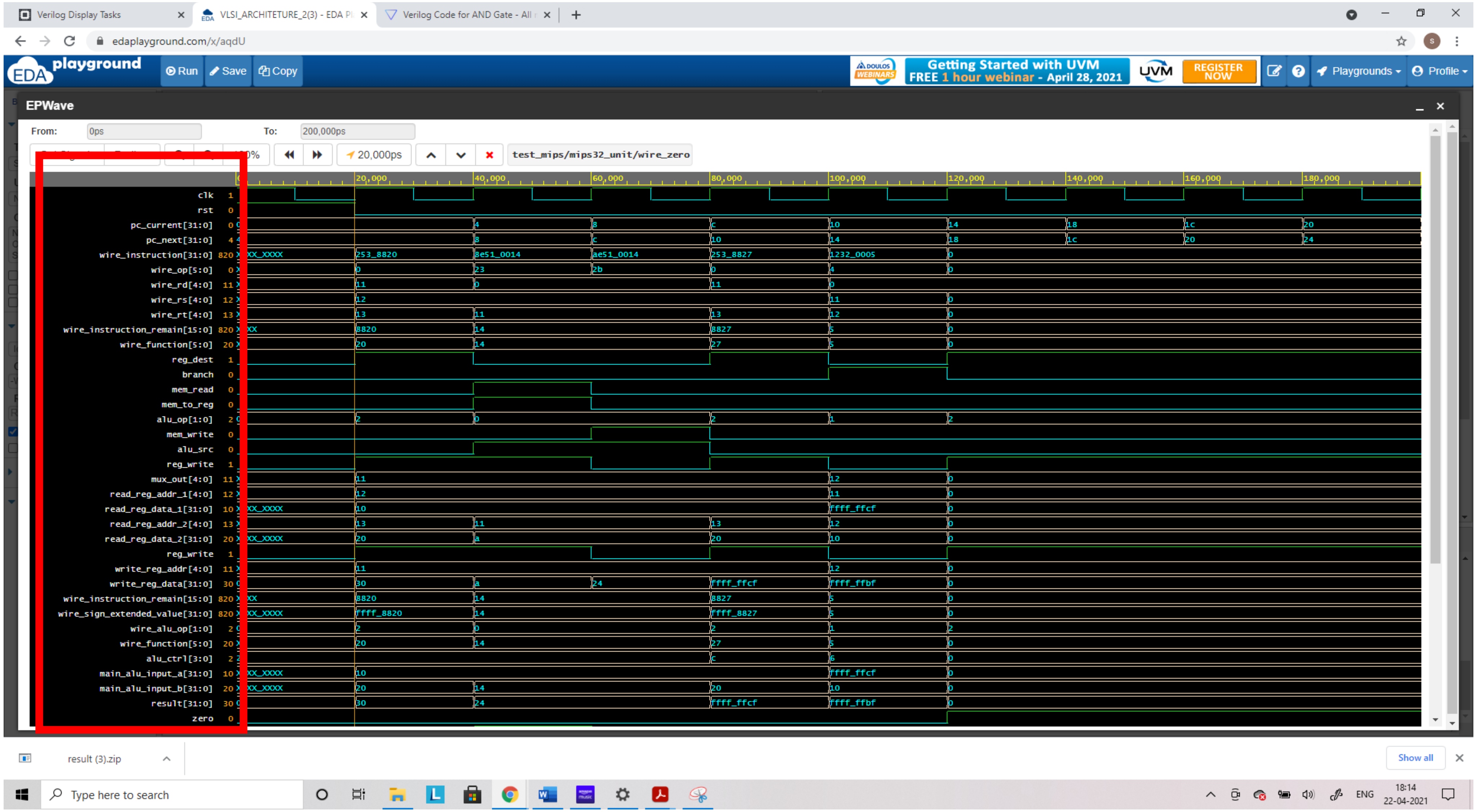
```
  assign alu_out=alu_in_pc_next+alu_in_shift_2;
endmodule
```

Test-bench

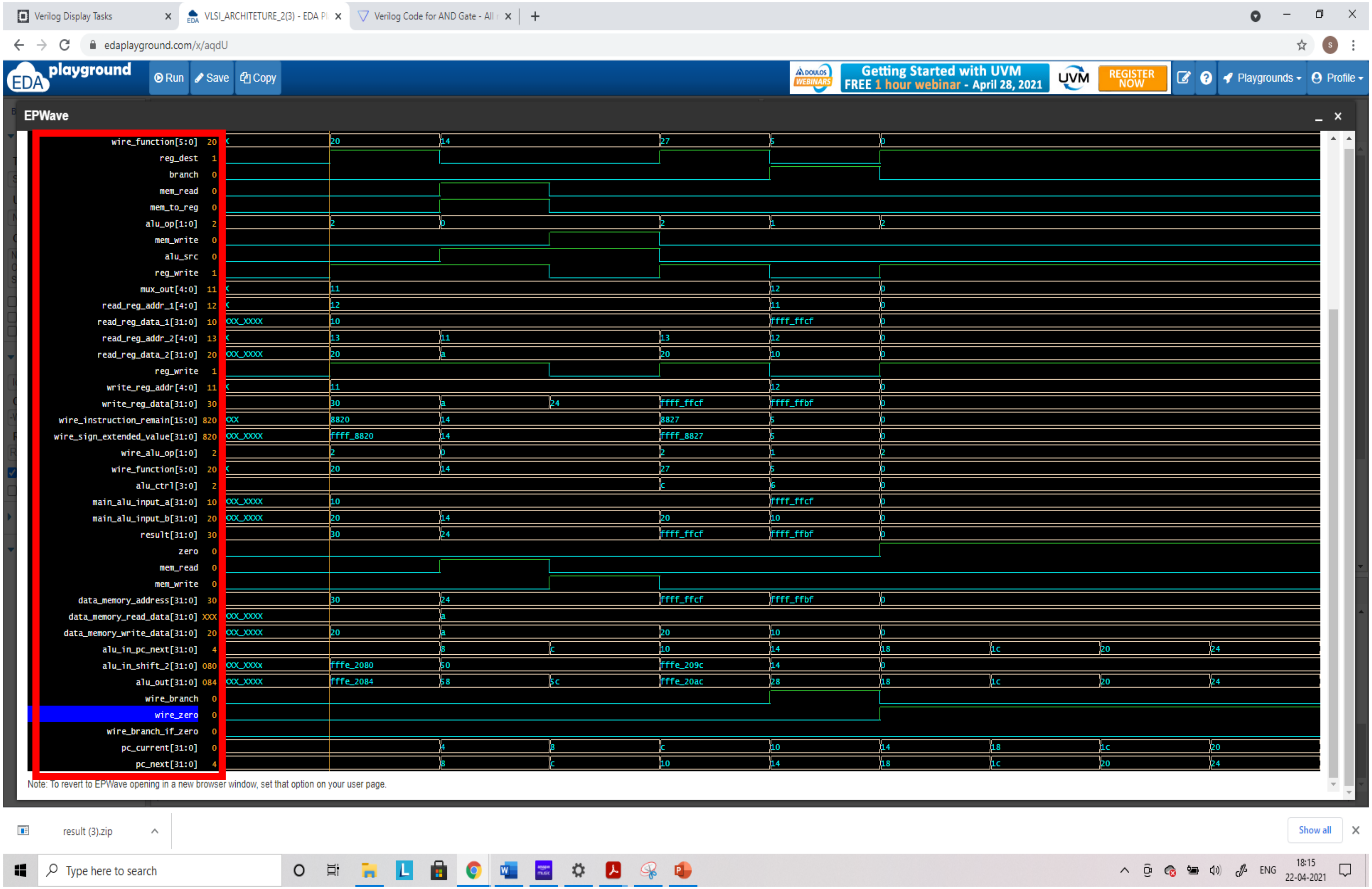
```
/*****************************************
* Author - Sarang Pramod Choudalwar
* VLSI - ARCHITECTURE
* Bits-pilani - wilp - sem-2
* Roll number - 2020ht01012
* Module - testbench.sv
* Function - Testbench for single cycle mips.
* Language Used - System Verilog
*****************************************/

`include "mips32.sv"

`timescale 1ns/1ps
module test_mips();

  reg tb_clk;
  reg tb_reset;
  wire[31:0] tb_alu_result;

  mips32 mips32_unit(.clk(tb_clk),
                     .reset(tb_reset),
                     .alu_out(tb_alu_result));
    initial begin
      $dumpfile("dump.vcd");
      $dumpvars();
    end

  initial
    begin

    tb_clk=1;
    forever #10 tb_clk = ~tb_clk;
  end

  initial begin
    tb_reset =1;
    @(posedge tb_clk)
    tb_reset =0;
  end
  initial begin
    #200 $finish;
  end


endmodule
```

Output of different instructions:- (Refer the red block for values of different internal signals)

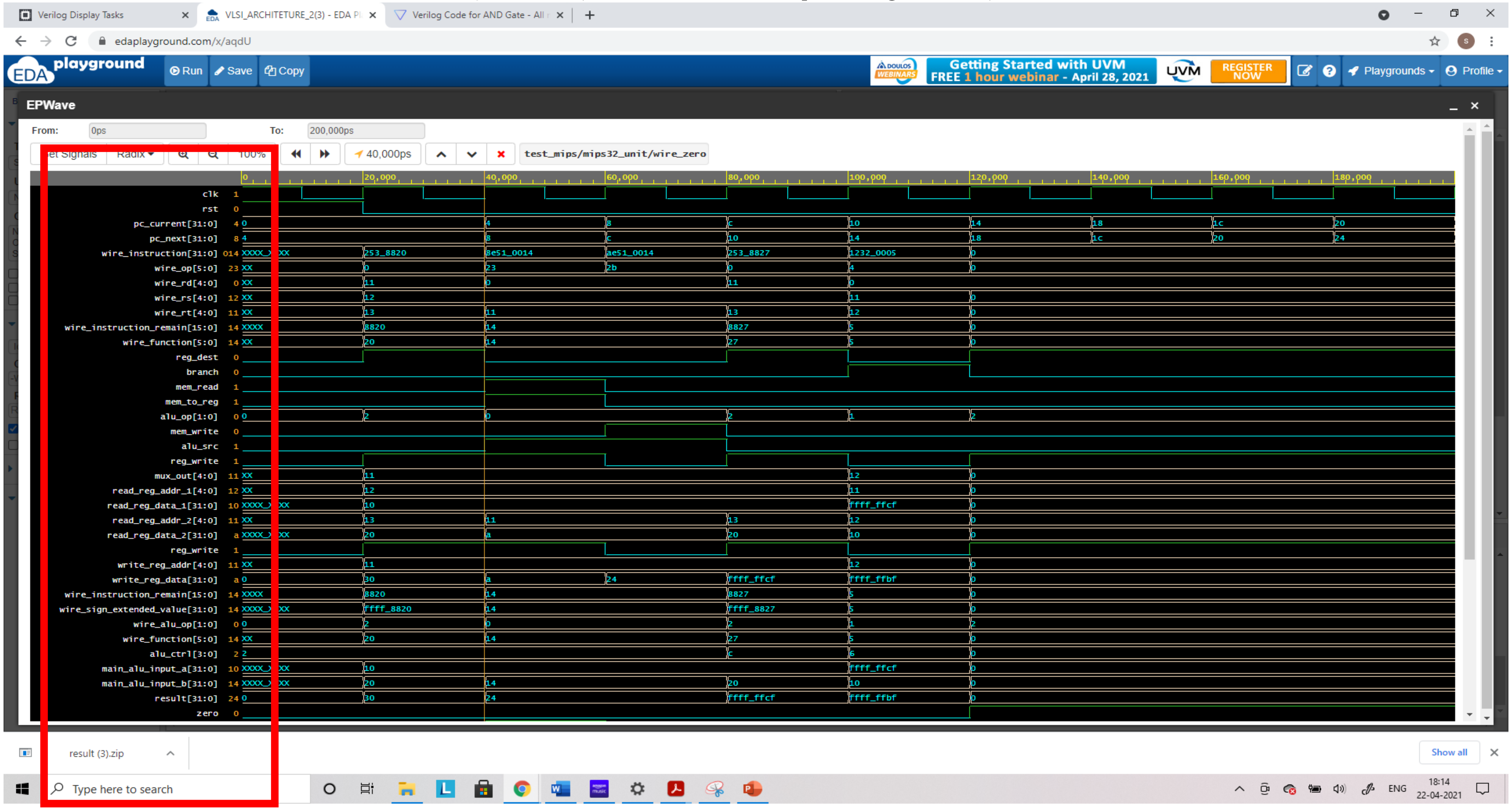
0 add add $s1,$s2,$s3 //$s1 = $s2 + $s3



0 add add $s1,$s2,$s3 //$s1 = $s2 + $s3 (Red block indicates values of different signals)

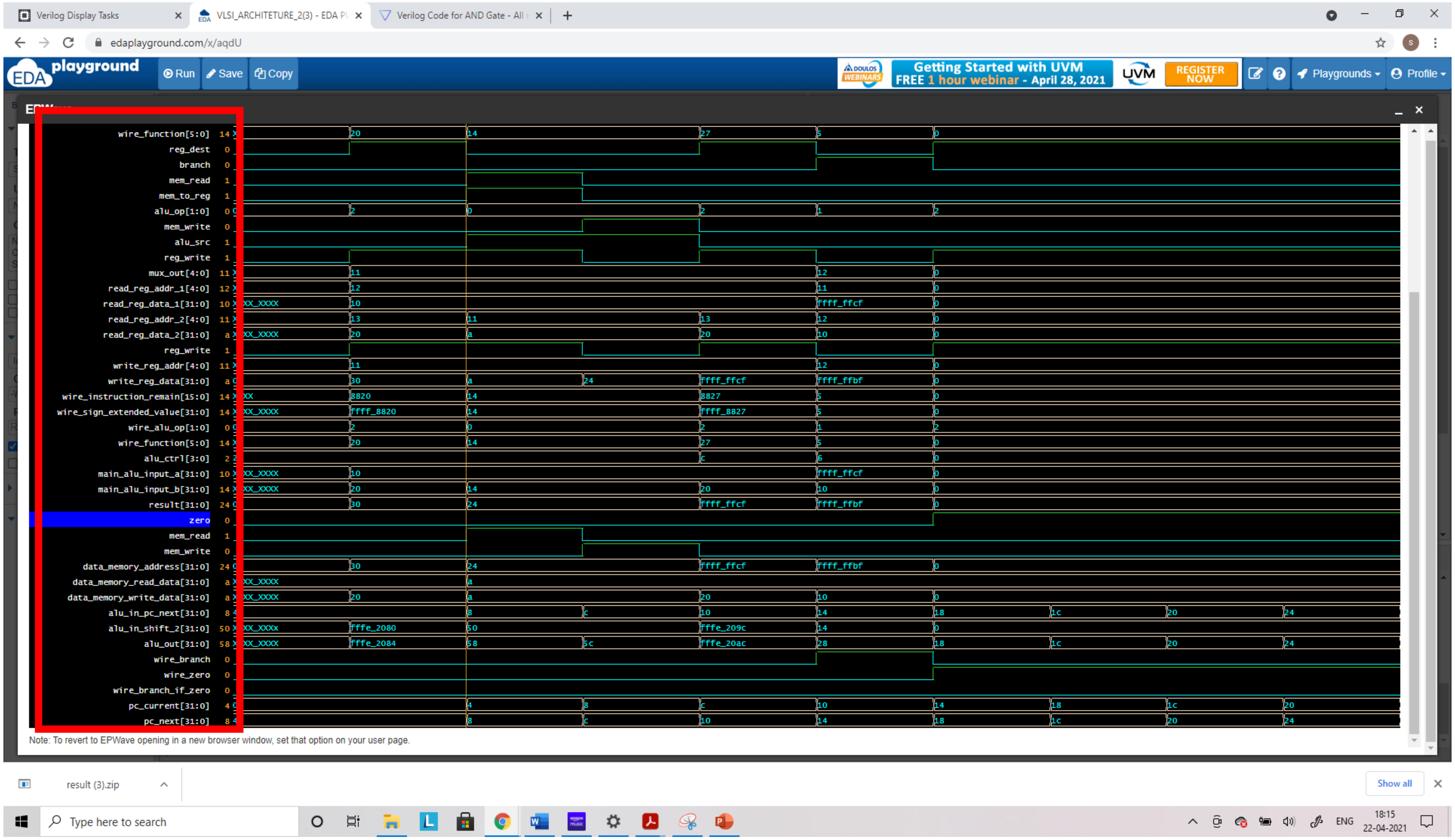


4 load word lw $s1,20($s2)   //$s1 = Memory[$s2 + 20] (Red block shows the important signal values)



```
clk                               1
rst                               0
pc_current[31:0]                  4
pc_next[31:0]                     8
wire_instruction[31:0]          014
wire_op[5:0]                     23
wire_rd[4:0]                      0
wire_rs[4:0]                     12
wire_rt[4:0]                     11
wire_instruction_remain[15:0]    14
wire_function[5:0]               14
reg_dest                          0
branch                            0
mem_read                          1
mem_to_reg                        1
alu_op[1:0]                       0
mem_write                         0
alu_src                           1
reg_write                         1
mux_out[4:0]                     11
read_reg_addr_1[4:0]             12
read_reg_data_1[31:0]            10
read_reg_addr_2[4:0]             11
read_reg_data_2[31:0]             a
reg_write                         1
write_reg_addr[4:0]              11
write_reg_data[31:0]              a
wire_instruction_remain[15:0]    14
wire_sign_extended_value[31:0]   14
wire_alu_op[1:0]                  0
wire_function[5:0]               14
alu_ctrl[3:0]                     2
main_alu_input_a[31:0]           10
main_alu_input_b[31:0]           14
result[31:0]                     24
zero                              0
```

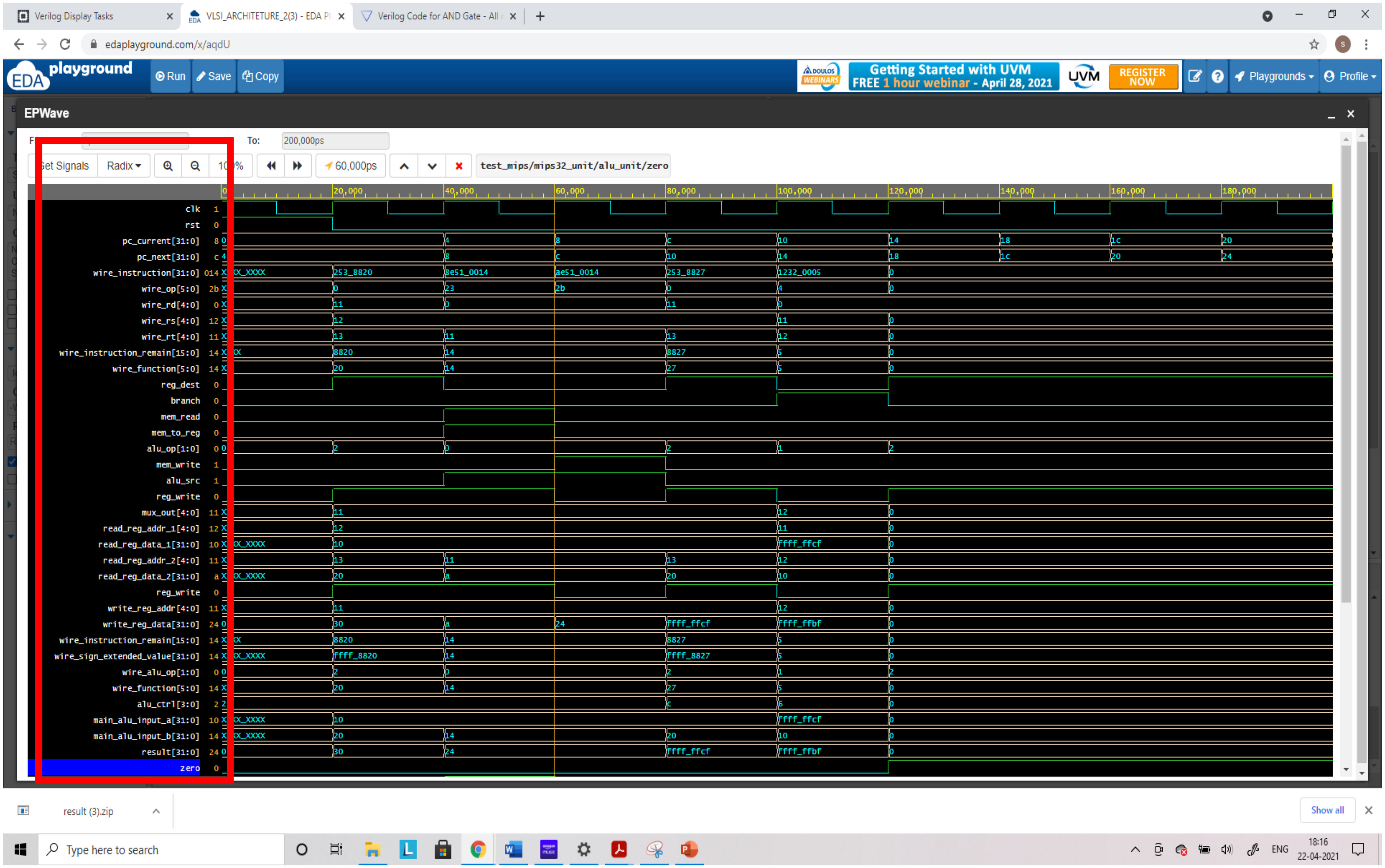
4 load word lw $s1,20($s2) //$s1 = Memory[$s2 + 20] (Red block shows the important signal values)



```
wire_function[5:0]               14
reg_dest                          0
branch                            0
mem_read                          1
mem_to_reg                        1
alu_op[1:0]                       0
mem_write                         0
alu_src                           1
reg_write                         1
mux_out[4:0]                     11
read_reg_addr_1[4:0]             12
read_reg_data_1[31:0]            10
read_reg_addr_2[4:0]             11
read_reg_data_2[31:0]             a
reg_write                         1
write_reg_addr[4:0]              11
write_reg_data[31:0]              a
wire_instruction_remain[15:0]    14
wire_sign_extended_value[31:0]   14
wire_alu_op[1:0]                  0
wire_function[5:0]               14
alu_ctrl[3:0]                     2
main_alu_input_a[31:0]           10
main_alu_input_b[31:0]           14
result[31:0]                     24
zero                              0
mem_read                          1
mem_write                         0
data_memory_address[31:0]        24
data_memory_read_data[31:0]       a
data_memory_write_data[31:0]      a
alu_in_pc_next[31:0]              8
alu_in_shift_2[31:0]             50
alu_out[31:0]                    58
wire_branch                       0
wire_zero                         0
wire_branch_if_zero               0
pc_current[31:0]                  4
pc_next[31:0]                     8
```

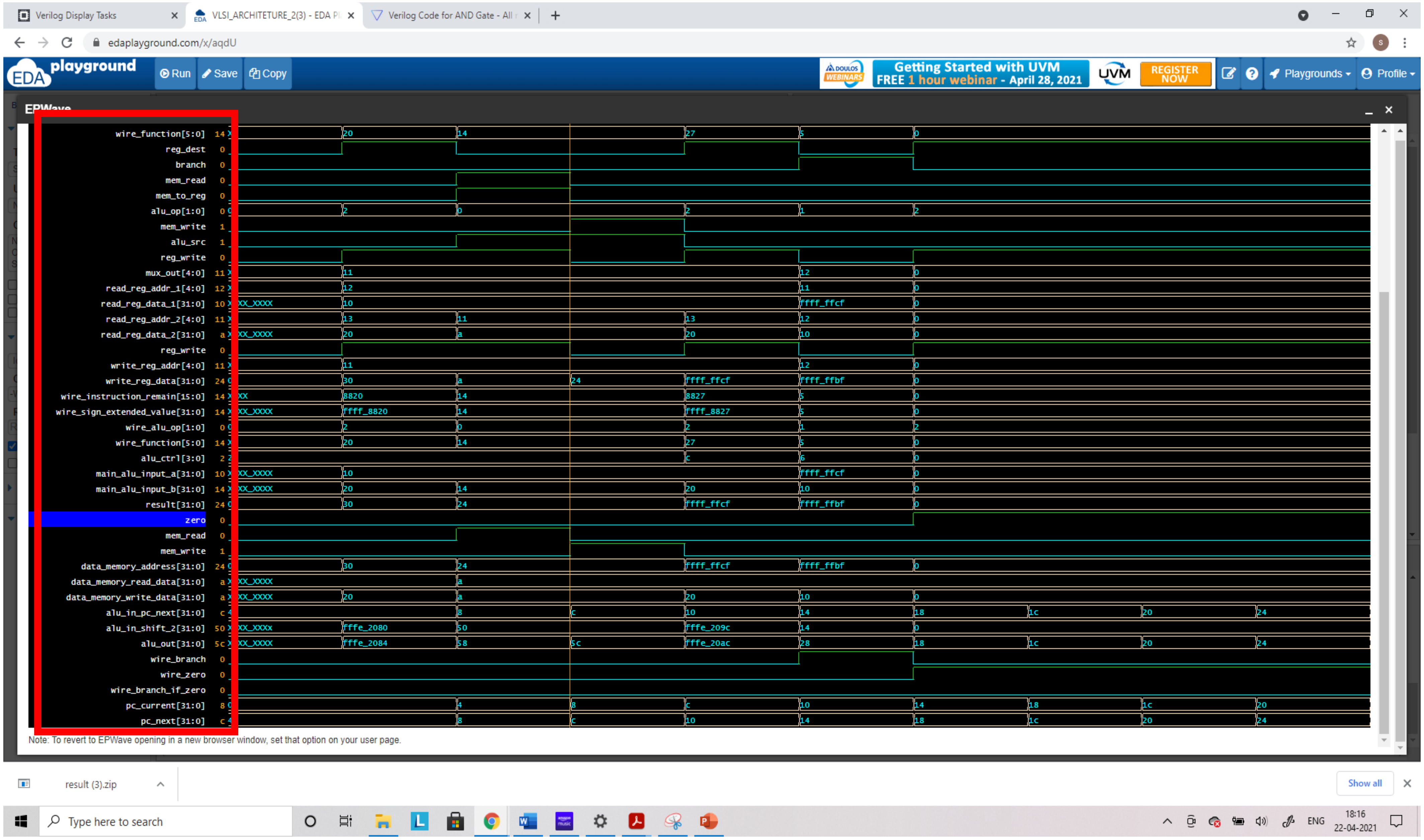
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

8 store word sw $s1,20($s2)   //Memory[$s2 + 20] = $s1 (Red block shows the important signal values)



```
clk                              1
rst                              0
pc_current[31:0]                 8
pc_next[31:0]                    c
wire_instruction[31:0]           014
wire_op[5:0]                     2b
wire_rd[4:0]                     0
wire_rs[4:0]                     12
wire_rt[4:0]                     11
wire_instruction_remain[15:0]    14
wire_function[5:0]               14
reg_dest                         0
branch                           0
mem_read                         0
mem_to_reg                       0
alu_op[1:0]                      0
mem_write                        1
alu_src                          1
reg_write                        0
mux_out[4:0]                     11
read_reg_addr_1[4:0]             12
read_reg_data_1[31:0]            10
read_reg_addr_2[4:0]             11
read_reg_data_2[31:0]            a
reg_write                        0
write_reg_addr[4:0]              11
write_reg_data[31:0]             24
wire_instruction_remain[15:0]    14
wire_sign_extended_value[31:0]   14
wire_alu_op[1:0]                 0
wire_function[5:0]               14
alu_ctrl[3:0]                    2
main_alu_input_a[31:0]           10
main_alu_input_b[31:0]           14
result[31:0]                     24
zero                             0
```

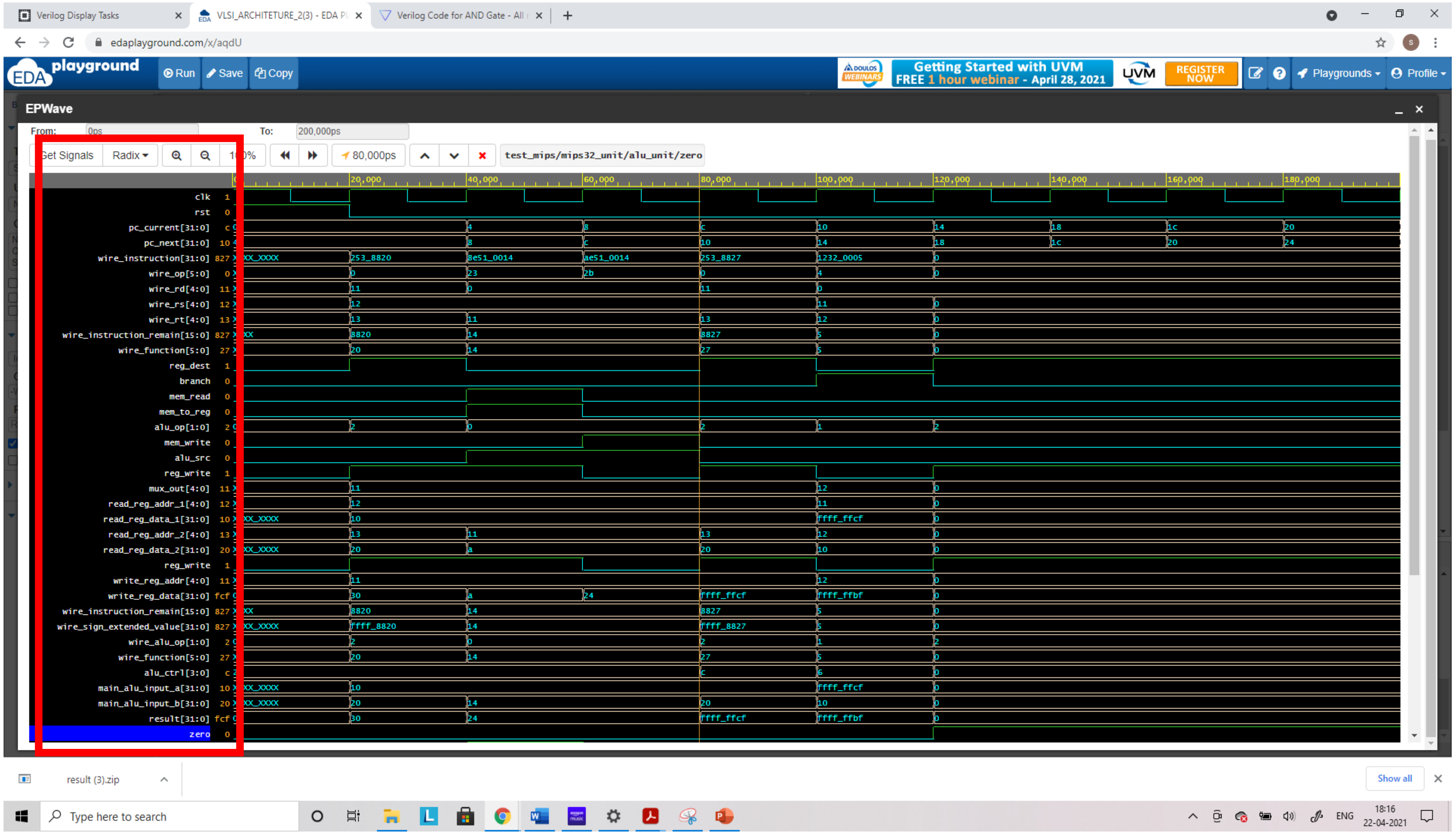
8 store word sw $s1,20($s2)   //Memory[$s2 + 20] = $s1 (Red block shows the important signal values)



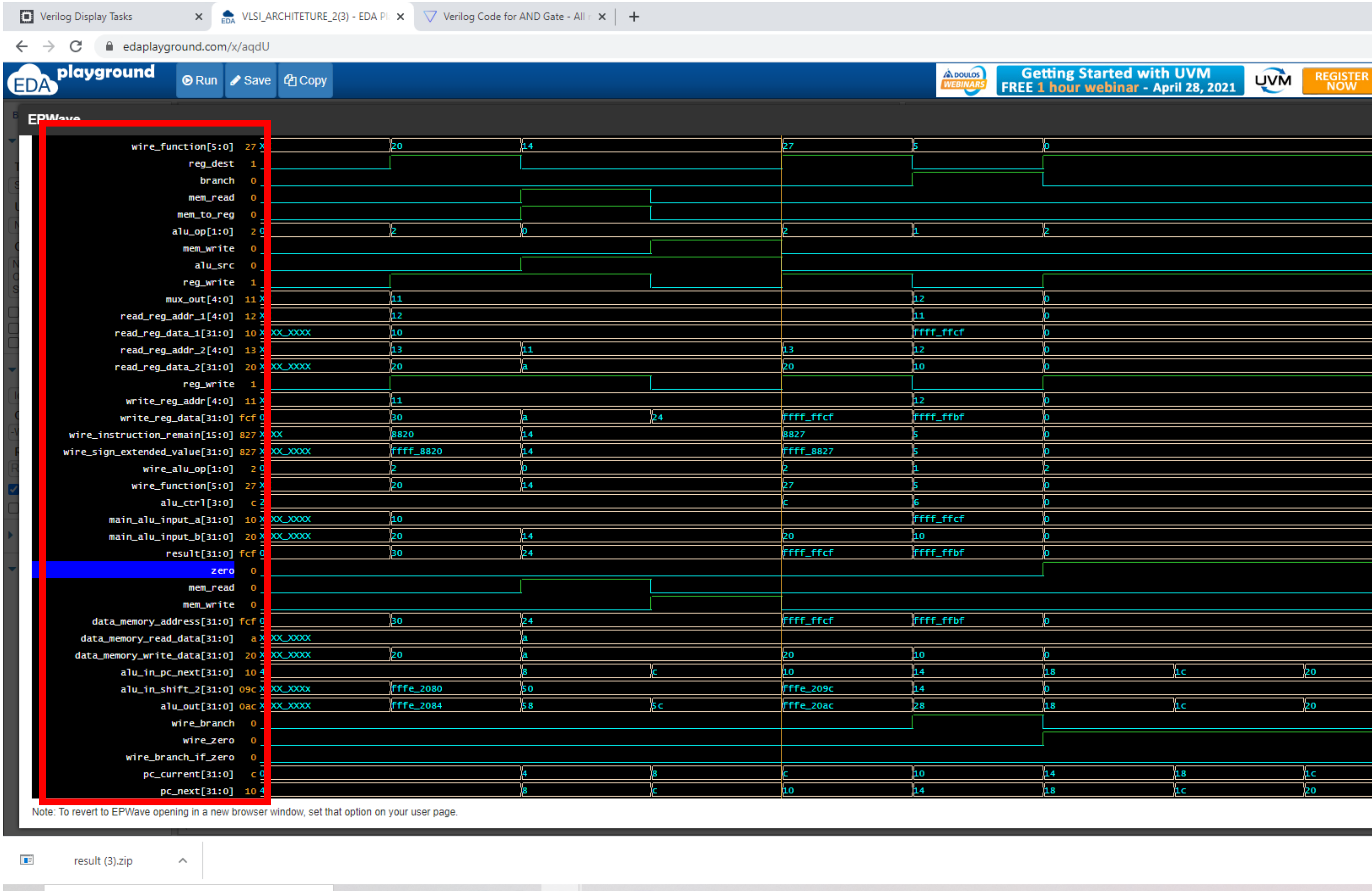
```
wire_function[5:0]            14
reg_dest                       0
branch                         0
mem_read                       0
mem_to_reg                     0
alu_op[1:0]                    0
mem_write                      1
alu_src                        1
reg_write                      0
mux_out[4:0]                  11
read_reg_addr_1[4:0]          12
read_reg_data_1[31:0]         10
read_reg_addr_2[4:0]          11
read_reg_data_2[31:0]          a
reg_write                      0
write_reg_addr[4:0]           11
write_reg_data[31:0]          24
wire_instruction_remain[15:0] 14
wire_sign_extended_value[31:0] 14
wire_alu_op[1:0]               0
wire_function[5:0]            14
alu_ctrl[3:0]                  2
main_alu_input_a[31:0]        10
main_alu_input_b[31:0]        14
result[31:0]                  24
zero                           0
mem_read                       0
mem_write                      1
data_memory_address[31:0]     24
data_memory_read_data[31:0]    a
data_memory_write_data[31:0]   a
alu_in_pc_next[31:0]           c
alu_in_shift_2[31:0]          50
alu_out[31:0]                 5c
wire_branch                    0
wire_zero                      0
wire_branch_if_zero            0
pc_current[31:0]               8
pc_next[31:0]                  c
```

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

12 nor nor $s1,$s2,$s3   //$s1 = ~ ($s2 | $s3) (Red block shows the important signal values)



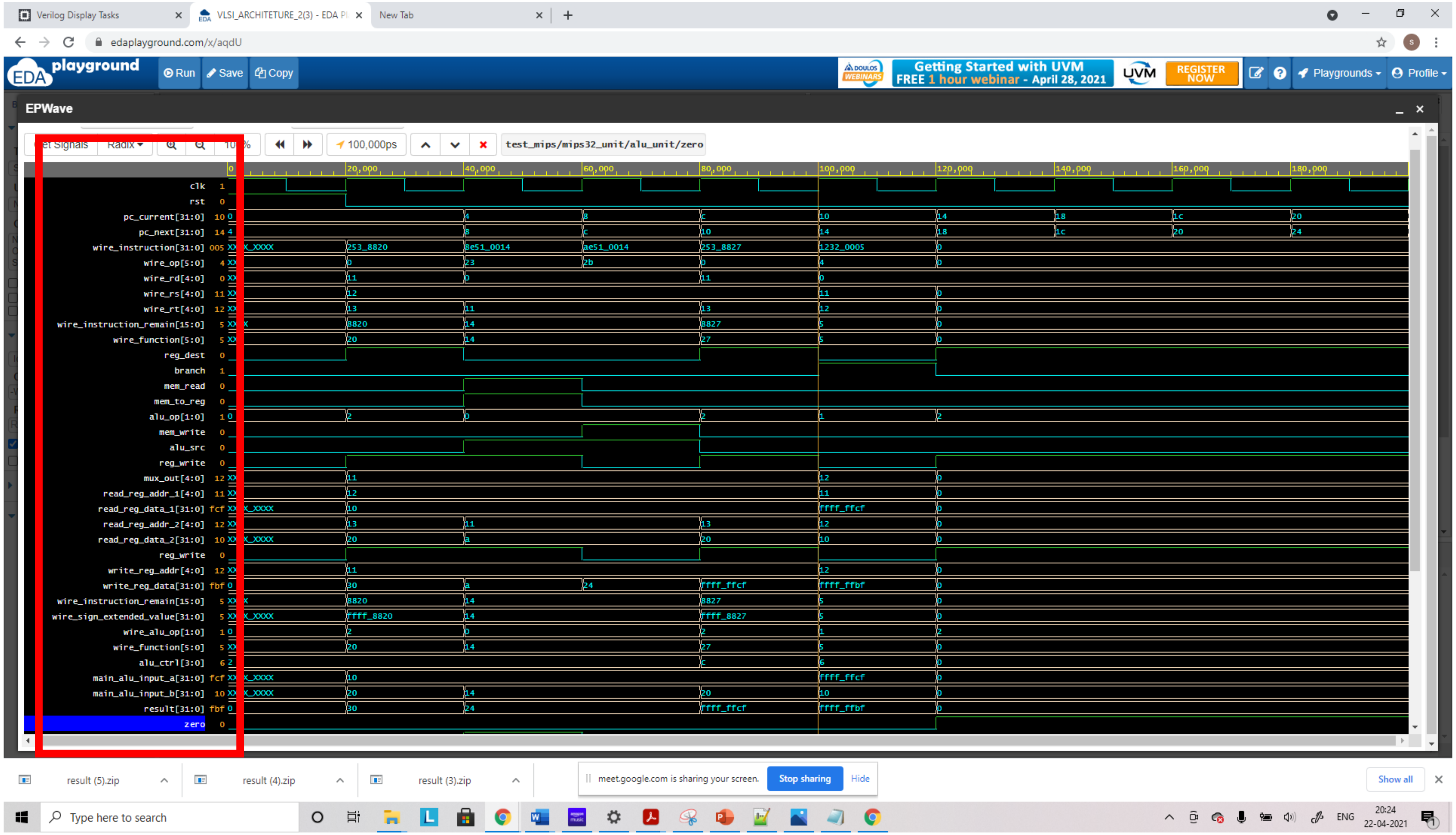
12 nor \$s1,\$s2,\$s3 //\$s1 = ~ (\$s2 | \$s3) (Red block shows the important signal values)



```
wire_function[5:0]          27
reg_dest                     1
branch                       0
mem_read                     0
mem_to_reg                   0
alu_op[1:0]                  2
mem_write                    0
alu_src                      0
reg_write                    1
mux_out[4:0]                11
read_reg_addr_1[4:0]        12
read_reg_data_1[31:0]       10
read_reg_addr_2[4:0]        13
read_reg_data_2[31:0]       20
reg_write                    1
write_reg_addr[4:0]         11
write_reg_data[31:0]       fcf
wire_instruction_remain[15:0] 827
wire_sign_extended_value[31:0] 827
wire_alu_op[1:0]             2
wire_function[5:0]          27
alu_ctrl[3:0]                c
main_alu_input_a[31:0]      10
main_alu_input_b[31:0]      20
result[31:0]               fcf
zero                         0
mem_read                     0
mem_write                    0
data_memory_address[31:0]  fcf
data_memory_read_data[31:0]  a
data_memory_write_data[31:0] 20
alu_in_pc_next[31:0]        10
alu_in_shift_2[31:0]       09c
alu_out[31:0]              0ac
wire_branch                  0
wire_zero                    0
wire_branch_if_zero          0
pc_current[31:0]             c
pc_next[31:0]               10
```

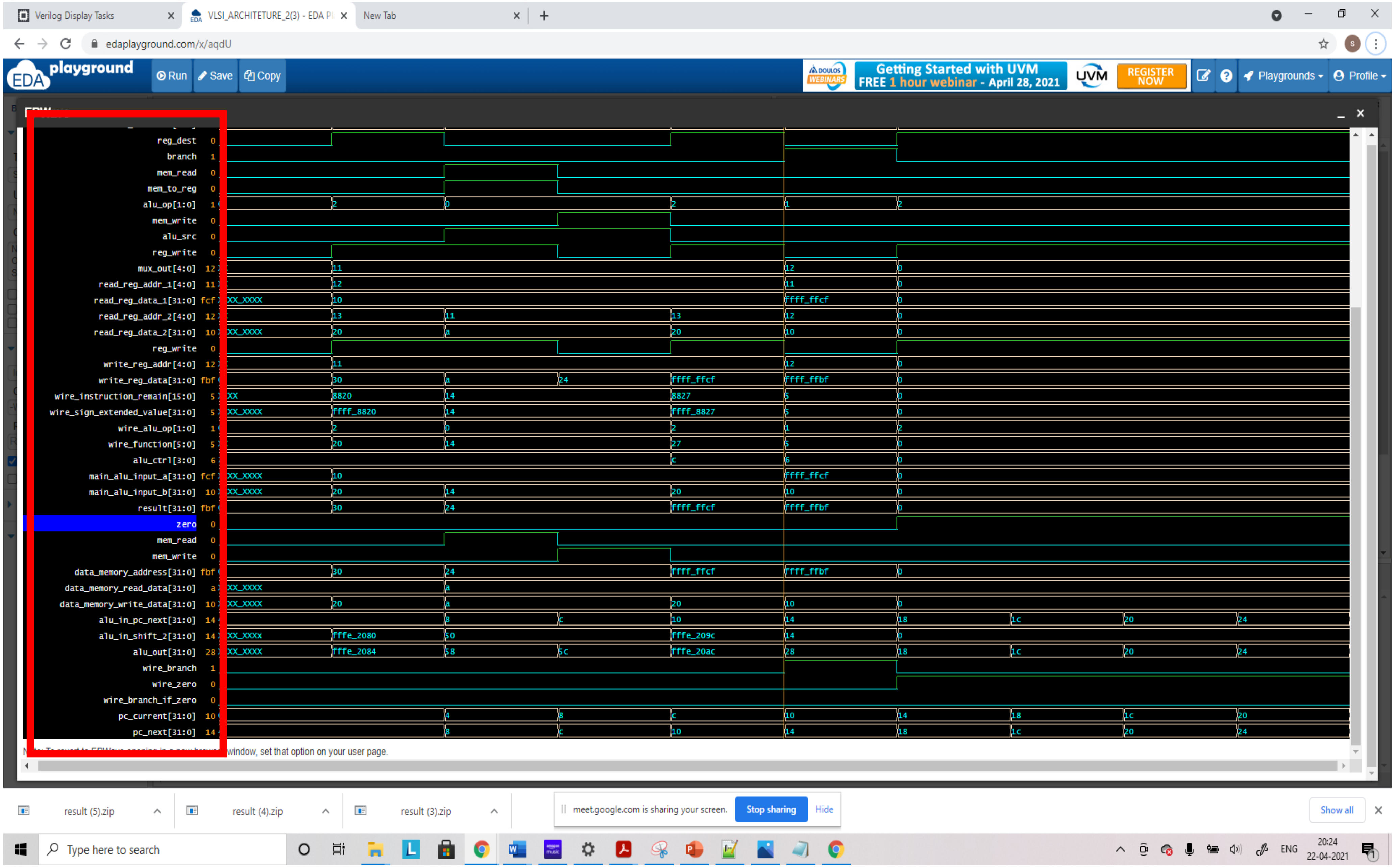
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

16 branch on equal beq $s1,$s2,5 //if ($s1 == $s2), execute nop (Red block shows the important signal values)



```
clk                               1
rst                               0
pc_current[31:0]                 10
pc_next[31:0]                    14
wire_instruction[31:0]          005
wire_op[5:0]                      4
wire_rd[4:0]                      0
wire_rs[4:0]                     11
wire_rt[4:0]                     12
wire_instruction_remain[15:0]     5
wire_function[5:0]                5
reg_dest                          0
branch                            1
mem_read                          0
mem_to_reg                        0
alu_op[1:0]                       1
mem_write                         0
alu_src                           0
reg_write                         0
mux_out[4:0]                     12
read_reg_addr_1[4:0]             11
read_reg_data_1[31:0]           fcf
read_reg_addr_2[4:0]             12
read_reg_data_2[31:0]            10
reg_write                         0
write_reg_addr[4:0]              12
write_reg_data[31:0]            fbf
wire_instruction_remain[15:0]     5
wire_sign_extended_value[31:0]    5
wire_alu_op[1:0]                  1
wire_function[5:0]                5
alu_ctrl[3:0]                     6
main_alu_input_a[31:0]          fcf
main_alu_input_b[31:0]           10
result[31:0]                    fbf
zero                              0
```

16 branch on equal beq $s1,$s2,5 //if ($s1 == $s2), execute nop (Red block shows the important signal values)

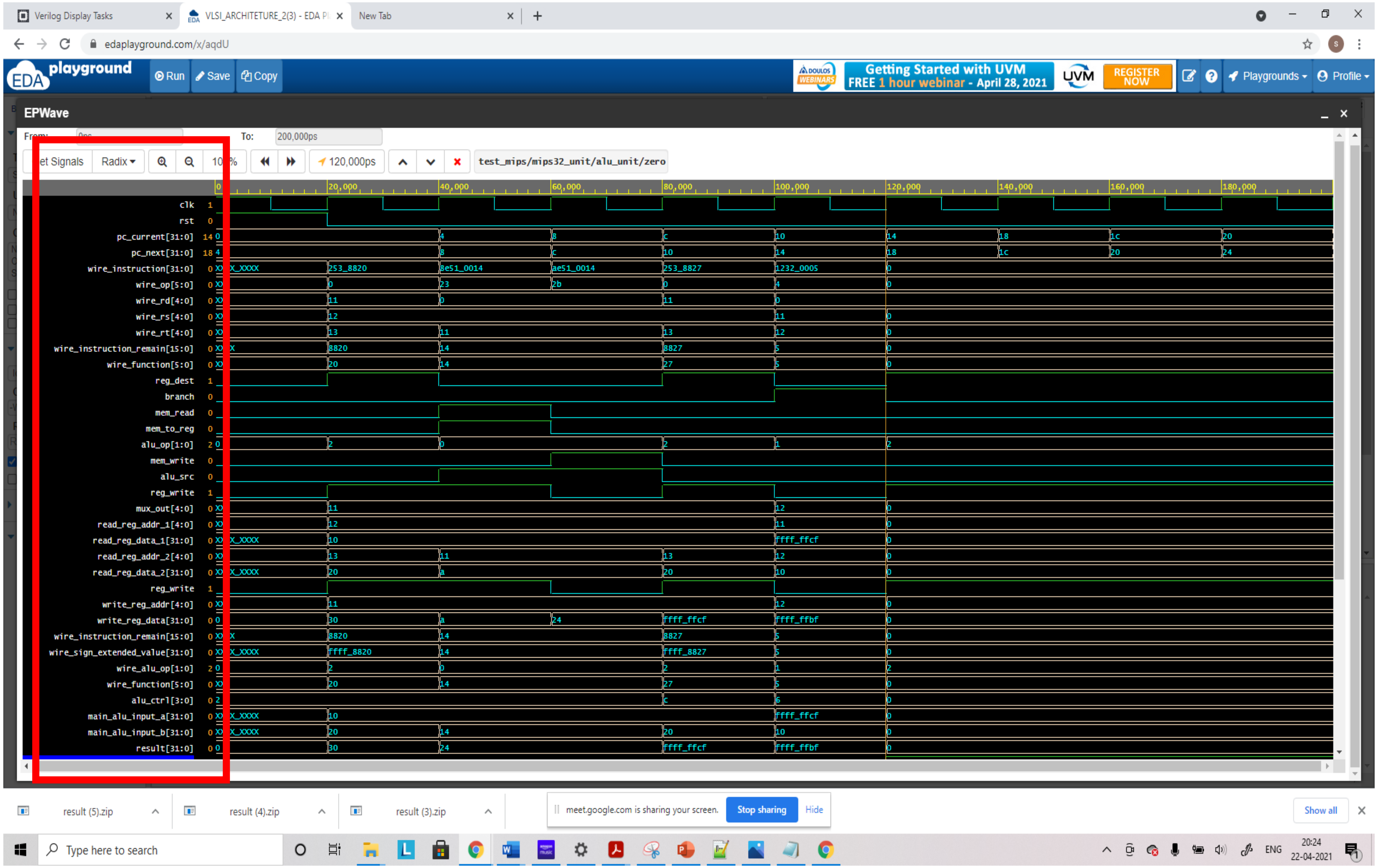


edaplayground.com/x/aqdU

```
reg_dest                        0
branch                          1
mem_read                        0
mem_to_reg                      0
alu_op[1:0]                     1
mem_write                       0
alu_src                         0
reg_write                       0
mux_out[4:0]                   12
read_reg_addr_1[4:0]           11
read_reg_data_1[31:0]         fcf
read_reg_addr_2[4:0]           12
read_reg_data_2[31:0]          10
reg_write                       0
write_reg_addr[4:0]            12
write_reg_data[31:0]          fbf
wire_instruction_remain[15:0]   5
wire_sign_extended_value[31:0]  5
wire_alu_op[1:0]                1
wire_function[5:0]              5
alu_ctrl[3:0]                   6
main_alu_input_a[31:0]        fcf
main_alu_input_b[31:0]         10
result[31:0]                  fbf
zero                            0
mem_read                        0
mem_write                       0
data_memory_address[31:0]     fbf
data_memory_read_data[31:0]     a
data_memory_write_data[31:0]   10
alu_in_pc_next[31:0]           14
alu_in_shift_2[31:0]           14
alu_out[31:0]                  28
wire_branch                     1
wire_zero                       0
wire_branch_if_zero             0
pc_current[31:0]               10
pc_next[31:0]                  14
```

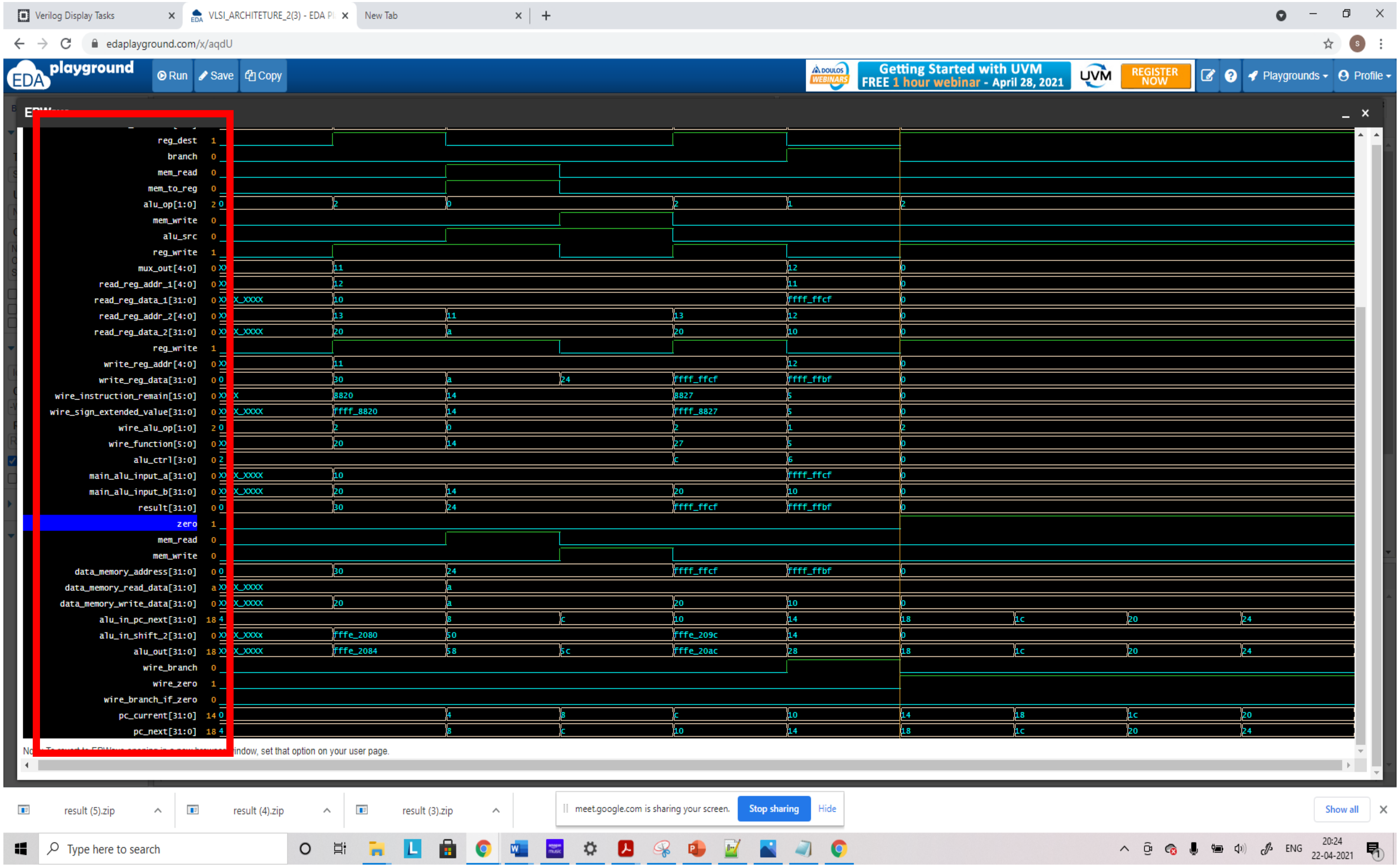
window, set that option on your user page.

20 nop //no operation (Red block shows the important signal values)



```
test_mips/mips32_unit/alu_unit/zero

clk                                1
rst                                0
pc_current[31:0]                  14
pc_next[31:0]                     18
wire_instruction[31:0]             0
wire_op[5:0]                       0
wire_rd[4:0]                       0
wire_rs[4:0]                       0
wire_rt[4:0]                       0
wire_instruction_remain[15:0]      0
wire_function[5:0]                 0
reg_dest                           1
branch                             0
mem_read                           0
mem_to_reg                         0
alu_op[1:0]                        2
mem_write                          0
alu_src                            0
reg_write                          1
mux_out[4:0]                       0
read_reg_addr_1[4:0]               0
read_reg_data_1[31:0]              0
read_reg_addr_2[4:0]               0
read_reg_data_2[31:0]              0
reg_write                          1
write_reg_addr[4:0]                0
write_reg_data[31:0]               0
wire_instruction_remain[15:0]      0
wire_sign_extended_value[31:0]     0
wire_alu_op[1:0]                   2
wire_function[5:0]                 0
alu_ctrl[3:0]                      0
main_alu_input_a[31:0]             0
main_alu_input_b[31:0]             0
result[31:0]                       0
```

20 nop //no operation (Red block shows the important signal values)



```
reg_dest                         1
branch                           0
mem_read                         0
mem_to_reg                       0
alu_op[1:0]                      2
mem_write                        0
alu_src                          0
reg_write                        1
mux_out[4:0]                     0
read_reg_addr_1[4:0]             0
read_reg_data_1[31:0]            0
read_reg_addr_2[4:0]             0
read_reg_data_2[31:0]            0
reg_write                        1
write_reg_addr[4:0]              0
write_reg_data[31:0]             0
wire_instruction_remain[15:0]    0
wire_sign_extended_value[31:0]   0
wire_alu_op[1:0]                 2
wire_function[5:0]               0
alu_ctrl[3:0]                    0
main_alu_input_a[31:0]           0
main_alu_input_b[31:0]           0
result[31:0]                     0
zero                             1
mem_read                         0
mem_write                        0
data_memory_address[31:0]        0
data_memory_read_data[31:0]      a
data_memory_write_data[31:0]     0
alu_in_pc_next[31:0]             18
alu_in_shift_2[31:0]             0
alu_out[31:0]                    18
wire_branch                      0
wire_zero                        1
wire_branch_if_zero              0
pc_current[31:0]                 14
pc_next[31:0]                    18
```