Project Two
RISC Processor Implementation
ECE 485

Chenqi Bao
Peter Chinetti

November 6, 2013

Instructor: Professor Borkar

1 Statement of Problem

This project requires the design and test of a RISC processor in VHDL. It focuses especially on the datapath design of the processor, and its implementation. In this groups' specific case, the required instructions[1] were:

Name	Abrev.	Type
Load Word	lw	I
Store Word	sw	I
Add	add	R
Branch On Equal	beq	I
NAND	nand	R
OR Immediate	ori	I
OR	or	R
AND Immediate	andi	I

2 Background

2.1 Instruction Types

The MIPS ISA defines three instruction types, R, I, and J type instructions. Only R and I type instructions will be covered here, as they are the only instructions that are to be implemented for this project.

[1] NAND does not exist in the MIPS ISA, so the ISA was extrapolated to fill out the table

2.1.1 R Type

R type, or register type instructions are the most common form of MIPS instructions. In this instruction format, the 32 bits of the instruction are split as follows:

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-11}	B_{10-6}	B_{5-0}
opcode	rs	rt	rd	shamt	funct

In these instructions, the opcode is always 000000_2, and the function code (funct) is used to determine the specific instruction. rs and rt are the two registers the operation is working on, and rd is the destination register. For some instructions, a shift amount (shamt) is needed, so it is specified.

2.1.2 I type

I type, or immediate type instructions are also very common. In this instruction format, the 32 bits of the instruction are split as follows:

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
opcode	rs	rt	immediate

In these instructions, the op code field actually encodes the specific instruction. rt is the destination register, and rs is the register on which the operation acts. The immediate field holds the immediate data that serves as the other operand.

2.2 Multicycle Datapath

The microprocessor logically comprises two main components: datapath and control. The datapath performs the arithmetic operations, and control tells the datapath, memory and I/O devices what to do according to the wishes of the instructions of the program [1].
When executing an instruction, the microprocessor steps through five main stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Operations (MEM) and Write Back (WB). Multicycle datapath implementations takes advantage of the fact that the stages of the operation can share the same hardware. Rather than use for example, a separate ALU for PC incrementing and addition of two registers, the same ALU can have its input switched from PC incrementation to register reads. This reuse saves on components in the processor, which can cost less.
Multicycle, however, requires some additional work in the form of multiplexers to select between inputs and outputs of each stage. Although this is a non-trivial amount of work, it is still better than duplicating components for each step.

2.3 VHDL

VHDL is a hardware description language that can be used to prototype digital systems. According to [2], "VHDL includes facilities for describing logical structure and function of digital system at a number of levels of abstraction, from system level down to the gate level."

3 Implementation

3.1 Design Decisions

3.1.1 Instruction Set

The first design decision was what to use as the format of the instructions requested. Generally, we used the format specified in the MIPS ISA, but, as mentioned earlier, NAND is not implemented in the MIPS ISA. Below is a list of our choices for opcodes and function codes:

OpCode	Function Field	Instruction	Operation
100011	000000	lw	lw $t3,200($t2)
101011	000000	sw	sw $t3,0($t2)
000000	100000	add	add $t1,$t1,$t1
000100	000000	beq	beq $t1,$t4,15
000000	100101	or	or $t0,$t1,$t0
001100	000000	andi	andi $t0,$t0,5
000000	000001	nand	nand $t0,$t0,$zero
001101	000000	ori	ori $t6,$t6,61680

3.1.2 Memory

Memory was implemented as a simple array of 256 words in this implementation. Larger memory sizes are possible, but they are unnecessarily complicated for a simple demonstration such as this.

3.2 Optimization

Little optimization was done on this project other than to not include obviously useless code. This processor is not pipelined, and as such, it is very much kept back from the optimization that make modern processors so quick.

3.3 Improvements

This processor has many ways to improve. Out of the large many ways, a few are most obvious: implement a complete instruction set, add piplineing, and increase the memory size. Currently the processor exists solely to serve as an educational demonstration, but could grow to be a complete implementation of the MIPS ISA given much improvement.

3.4 Failures

Thankfully, we have no failures to report.

3.5 Block Diagram

See figure 1.

3.6 Simulation

The output of the simulations can be found in figures 2-9. The simulation was done sequentially, in the order of presentation, so the values going into subsequent instructions are often dependent on the output of the previous command.

3.7 Code Listing

3.7.1 Datapath

```
entity MIPS is
  Port (
    clock :in bit;                                  --clock record
    PC0 : out integer;                             --PC counter (32 bits
    )
    SET : in bit;
    Memval : out bit_vector (31 downto 0);       --mem word
    addressable
    Instrval : out bit_vector (31 downto 0);     --Instruction 32
    bits wide
    Output : out BIT_VECTOR (31 downto 0);       --We are working in
     Word size
    Port1,Port2,Port3,Port4 : out bit_vector (31 downto 0));
end MIPS;

architecture INSTRUCTION of MIPS is
------------------------ Data types
 signal internal_state: integer;
 subtype word is bit_vector(31 downto 0); -- 32-bit words
 type regfile is array (0 to 31) of word; -- 32 words
type ram is array (0 to 255) of word; -- toy sized ram for testing
 subtype reg_addr is bit_vector(4 downto 0); -- 2^5 can store 32
    regs
subtype halfword is bit_vector(15 downto 0); -- 16-bit entities i.e
    . Immediate value
subtype byte is bit_vector(7 downto 0); -- if we need bytes
constant bvc : bit_vector (0 to 1) := "01";   --Binary value
-------------------int -> bits
procedure int2bits(int :in integer; bits :out bit_vector) is
variable temp: integer;
variable result: bit_vector(bits'range);
begin
  temp := int;
  if int < 0 then
    temp := -int - 1;
```

```
  end if;
  for index in bits'reverse_range loop
    result(index) := bvc(temp rem 2);
    temp := temp/2 ;
  end loop;
  if int < 0 then
      result := not result;
      result(bits'left) := '1';
  end if;
  bits := result;
  end int2bits;
--------------------bits -> unsigned int
function bits2int (bits : in bit_vector) return integer is
variable result : integer := 0;
begin
  for index in bits'range loop
    result := result * 2 + bit'pos(bits(index));
  end loop;
  return result;
end bits2int;

  ---------------------- Sign Extend
  function sign_ext(imm : in halfword) return word is
  variable extended : word;
  begin
    if imm(imm'left) = '1' then
      extended := (31 downto 16 => '1')& imm;
    else
      extended := (31 downto 16 => '0')& imm;
    end if;
    return extended;
end sign_ext;
--------------------+/-
procedure alu_add_subtract (addsel: in bit; result : inout word; a,
     nb : in word; V,N : out bit) is -- Overflow -> Cout
variable sum : word;
variable carry : bit := '0';
variable b: word;
begin
  if addsel ='1' then
  b:=Not nb;
  carry := '1';
  else b := nb;
end if;
for index in sum'reverse_range loop
sum(index) := a(index) xor carry xor b(index);
carry := ( a(index) and b(index) ) or ( carry and ( a(index) xor b(
    index) ) );
end loop;
result := sum;
V := carry ;--= '1';
end procedure alu_add_subtract;

-------------------------------------------------------------------
Begin Proc: Process(clock)
variable i: integer:=0; -- Execution cycle counter
Begin
```

```
85    if clock = '1' and clock'event then
          if i = 5 OR SET = '1' then -- reset on SET or 5 cycles
87          i := 0;
          end if;
89        i:=i+1;
          internal_state <= i;
91        end if;
  end process Proc;
93
95
97  Datapath: Process(internal_state)
    variable result,Instr,op1,op2,op3,maddr : word;
99  variable opcode, funct : bit_vector(5 downto 0);
    variable rs,rt,rd,dstreg,shamt : reg_addr;
101 variable state : integer:=0; -- =='cycle'

103 variable PC : integer:= 0;
    variable Imm : halfword;
105 variable mem_index : byte; -- only need 8 bits
    variable reg : regfile:= (9 => X"0000_0001", 10 => X"0000_0002",12
        => X"0000_0002", others => X"0000_0000");
107 variable mem : ram := (
    0 => X"8D4B_00C8", -- lw $t3,200($t2) [Load $7FFF_FFFF to $t3]
109 1 => X"AD2B_0000", -- sw $t3,0($t2)   [Store $7FFF_FFFF to memory
        address 2]
    2 => X"0129_4820", -- add $t1, $t1, $t1 [ doing 1+1 and store the
        result in $t1]
111 3 => X"112C_000B", -- beq $t1,$t4,15   [ If $t1=2, go to instr. 15]
    15 => X"010B_4025", -- or $t0, $t3, $t0 [or 7FFF_FFFF with 0000
        _0000]
113 16 => X"3108_0005", -- andi $t0,$t0,5   [ and 7FFF_FFFF with 5]
    17 => X"0100_0021", -- nand $t0,$t0,$zero [ nand 0000_0005 with
        0000_0000 => FFFF_FFF2]
115 18 => X"35CE_F0F0", -- ori $t0,$zero,61608   [or 0000_F0F0 with
        0000_0000 => FFFF_FFFF]
    others => X"0000_0000");
117 variable mem_rw : boolean; -- Mem Access
    variable mem_r : boolean; -- Mem Read
119 variable i: integer:=0; -- Exec cycle counter
    variable Dmem : ram := (
121 202 => X"7FFF_FFFF",
    others => X"0000_0000");
123 variable V,N,RST : bit;


125
    Begin state:=internal_state;
127 case state is
    when 1 =>
129 -- IF
      Instr := mem(PC); PC := PC + 1; --If PC is an int, incremeting by
          1 works
131   RST := '0';  -- init
      mem_rw := false;  -- init
133   when 2 =>
      -- ID
```

```
135   opcode := Instr(31 downto 26);
      rs := Instr(25 downto 21);
137   rt := Instr(20 downto 16);
      rd := Instr(15 downto 11);
139
      dstreg := rt;
141   Imm := Instr(15 downto 0);
      shamt := Instr(10 downto 6);
143   funct := Instr(5 downto 0);
      op1 := reg(bits2int(rs)); -- after filtering to an int, store
145   op2 := reg(bits2int(rt));
      op3 := sign_ext(Imm); -- this is the immediate value after being
        sign extended
147
    when 3 =>
149 -- EX
    case opcode is -- switch on opcode
151   when "100011" => --lw
        alu_add_subtract('0',maddr,op1,op3,V,N);
153     mem_rw :=true;
        mem_r :=true;
155   when "101011" => --sw
        alu_add_subtract('0',maddr,op1,op3,V,N);
157     mem_rw :=true;
        mem_r :=false;
159   when "000100" => --beq
        alu_add_subtract('1',result,op1,op2,V,N);
161     if result = X"0000_0000" then--if our ALU had a zero output,
        take the branch
        PC := PC + bits2int(op3);
163     RST := '1';
        end if;
165   when "001101" => --ORI
      result := op1 OR op3;
167   when "001100" => --ANDI
      result := op1 AND op3;
169
      when "000000" => --0 op code, therefore R type
171    dstreg := rd; --R types always have rd as the dest
       case funct is
173       when "100000" => --Add
            alu_add_subtract('0', result, op1,op2,V,N);
175       when "100001" => --NAND
            result :=op1 NAND op2;
177       when "100100" => --AND
            result := op1 AND op2;
179       when "100101" => --OR
            result := op1 OR op2;
181       when others =>
    end case;
183 when others =>
    end case;
185
      when 4 => --MEM
187     if mem_rw = true then -- These flags got set above when
        decoding lw and sw
          if mem_r = true then --set on read
```

```
189           result := Dmem(bits2int(maddr));
              else -- cleared on write
191           Dmem(bits2int(maddr)) := op2; -- reg2 written to mem
              RST := '1';
193           end if;
        end if;
195
      when 5 => -- Write-back cycle
197   if RST = '0' then  -- if we didn't write to mem
        reg(bits2int(dstreg)) := result; -- writeback value to dest.
        register
199   end if;
      when others =>
201   end case;

203   Output <= result;
      Memval <= mem(bits2int(maddr));
205   PC0 <= PC;
      InstrVal <= Instr;
207   Port1 <= op1;
      Port2 <= op2;
209   Port3 <= op3;
      Port4 <= reg(bits2int(dstreg));
211   end process Datapath;
      end INSTRUCTION;
```

MIPS.vhd

3.7.2 Simulator

```
1
ENTITY sim2 IS
3 END sim2;

5 ARCHITECTURE simulation OF sim2 IS
COMPONENT MIPS
7   PORT ( clock :In bit;
    SET : In bit;
9   Output : Out BIT_VECTOR (31 DownTo 0);
    PC0 : Out INTEGER;
11  Memval : Out BIT_VECTOR (31 DownTo 0);
    Instrval : Out BIT_VECTOR (31 DownTo 0);
13  Port1,Port2,Port3,Port4 : Out BIT_VECTOR (31 DownTo 0)
    );
15  END COMPONENT; --

17 SIGNAL Clock : bit := '0';
   SIGNAL SET : bit := '0';
19 SIGNAL Output : BIT_VECTOR (31 DownTo 0) := "
       00000000000000000000000000000000";
   SIGNAL PC0 : INTEGER := 0;
21 SIGNAL Memval : BIT_VECTOR (31 DownTo 0) := "
       00000000000000000000000000000000";
   SIGNAL Instrval : BIT_VECTOR (31 DownTo 0) := "
       00000000000000000000000000000000";
```

```
SIGNAL Port1,Port2,Port3,Port4 : BIT_VECTOR (31 DownTo 0) := "
    00000000000000000000000000000000";

--||||||||||||||||||||||||||||||Simulation begins
    ||||||||||||||||||||||||||||||!
BEGIN
UUT : MIPS
PORT MAP (
clock => clock,
SET => SET,
Instrval => Instrval,
Output => Output,
PC0 => PC0,
Memval => Memval,
Port1 => Port1,
Port2 => Port2,
Port3 => Port3,
Port4 => Port4
);
 PROCESS
BEGIN
CL : LOOP
clock <= '0';
WAIT FOR 50 ns;
clock <= '1';
WAIT FOR 50 ns;
END LOOP CL;
END PROCESS;
PROCESS
BEGIN
WAIT FOR 5000 ns;
END PROCESS;


END simulation;
```

simulator.vhd

References

[1] David A. Patterson, John L. Hennesy, *Computer Organization and Design*. Morgan Kaufmann, Massachusetts, 4th Revised Edition, 2012.

[2] Peter J. Ashden, *VHDL Tutorial*. Elsevier Science, USA, 2004

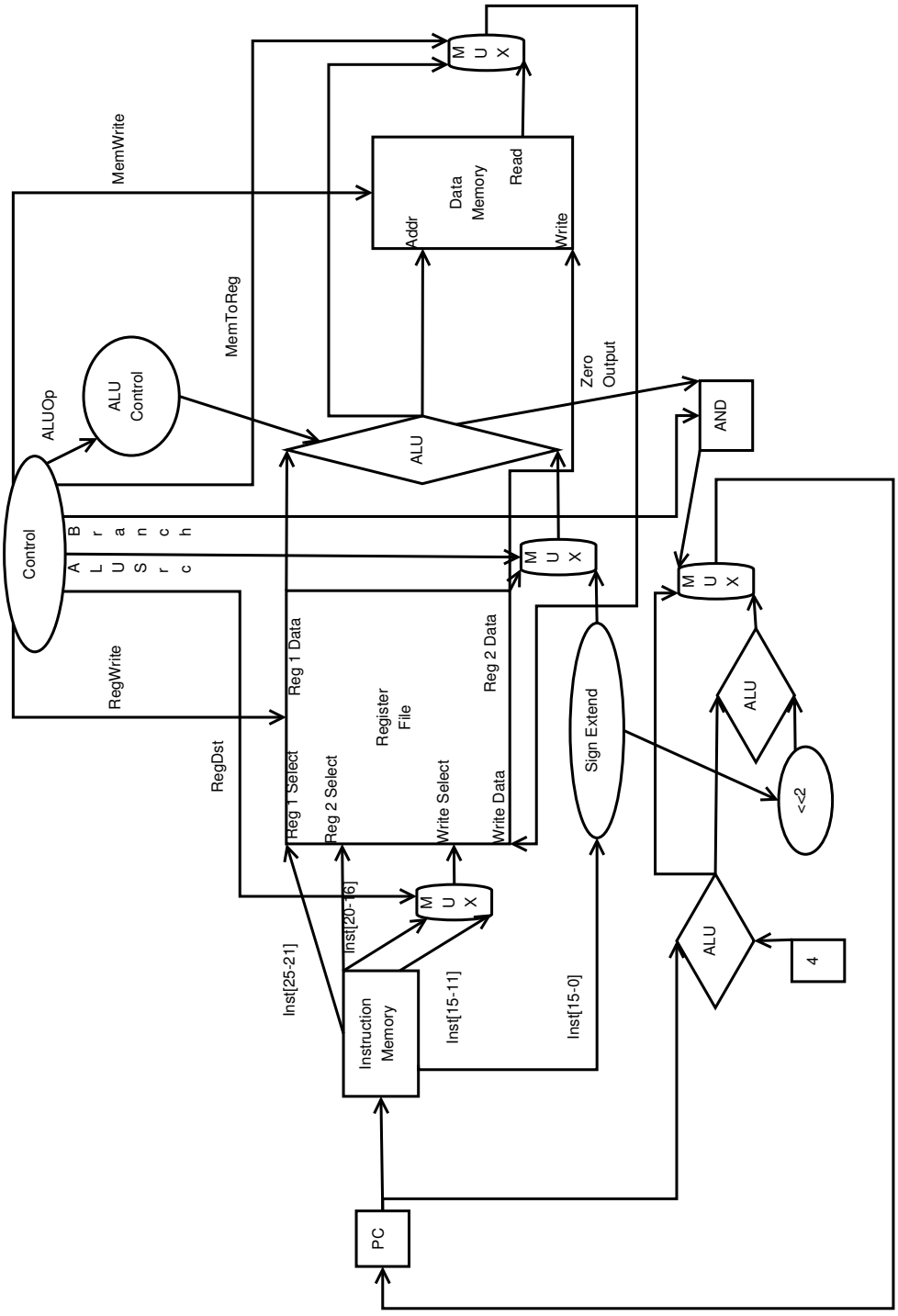

Figure 1: Block Diagram

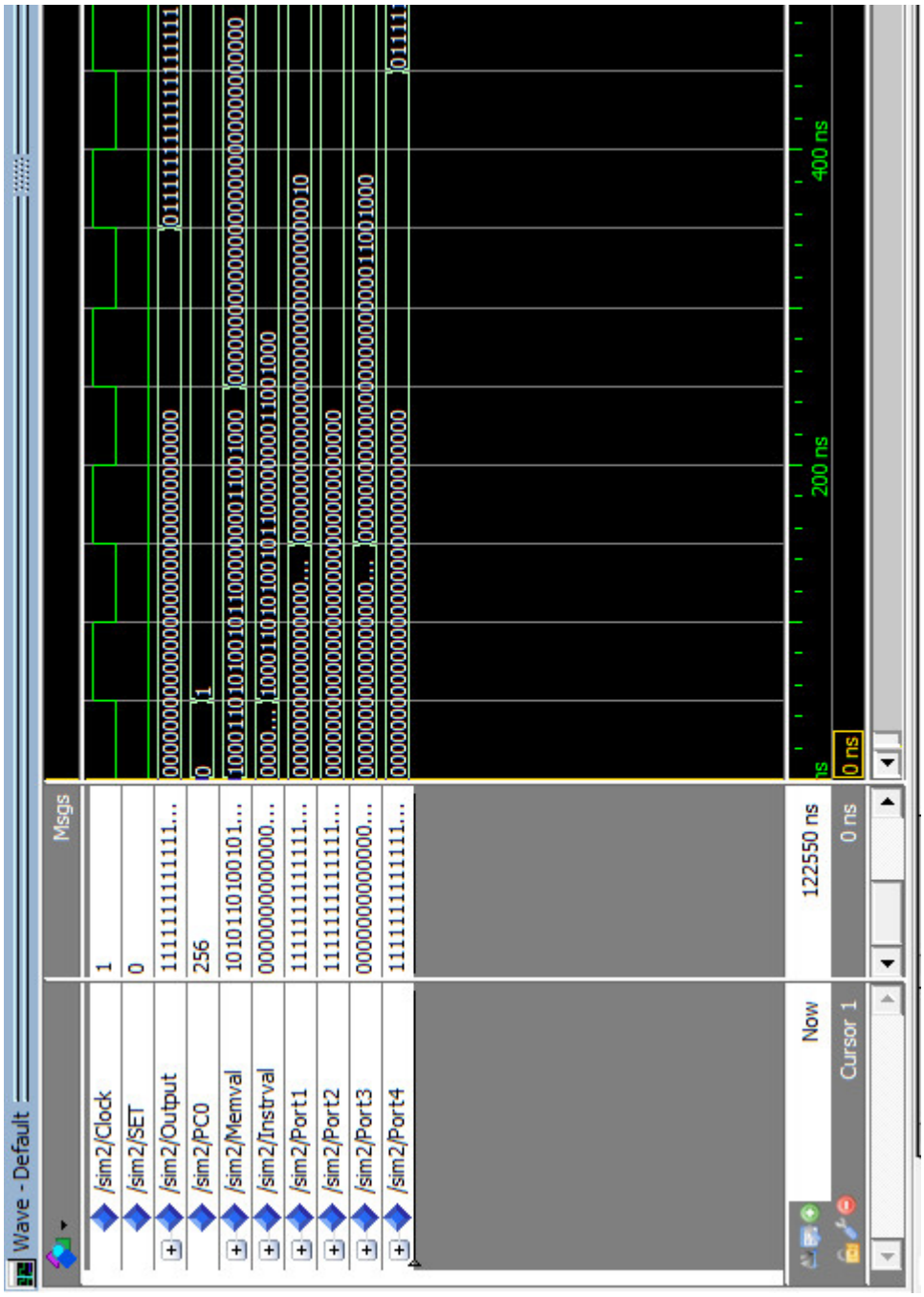

Figure 2: lw $t3,200($t2); *Loading 7FFF FFFF*

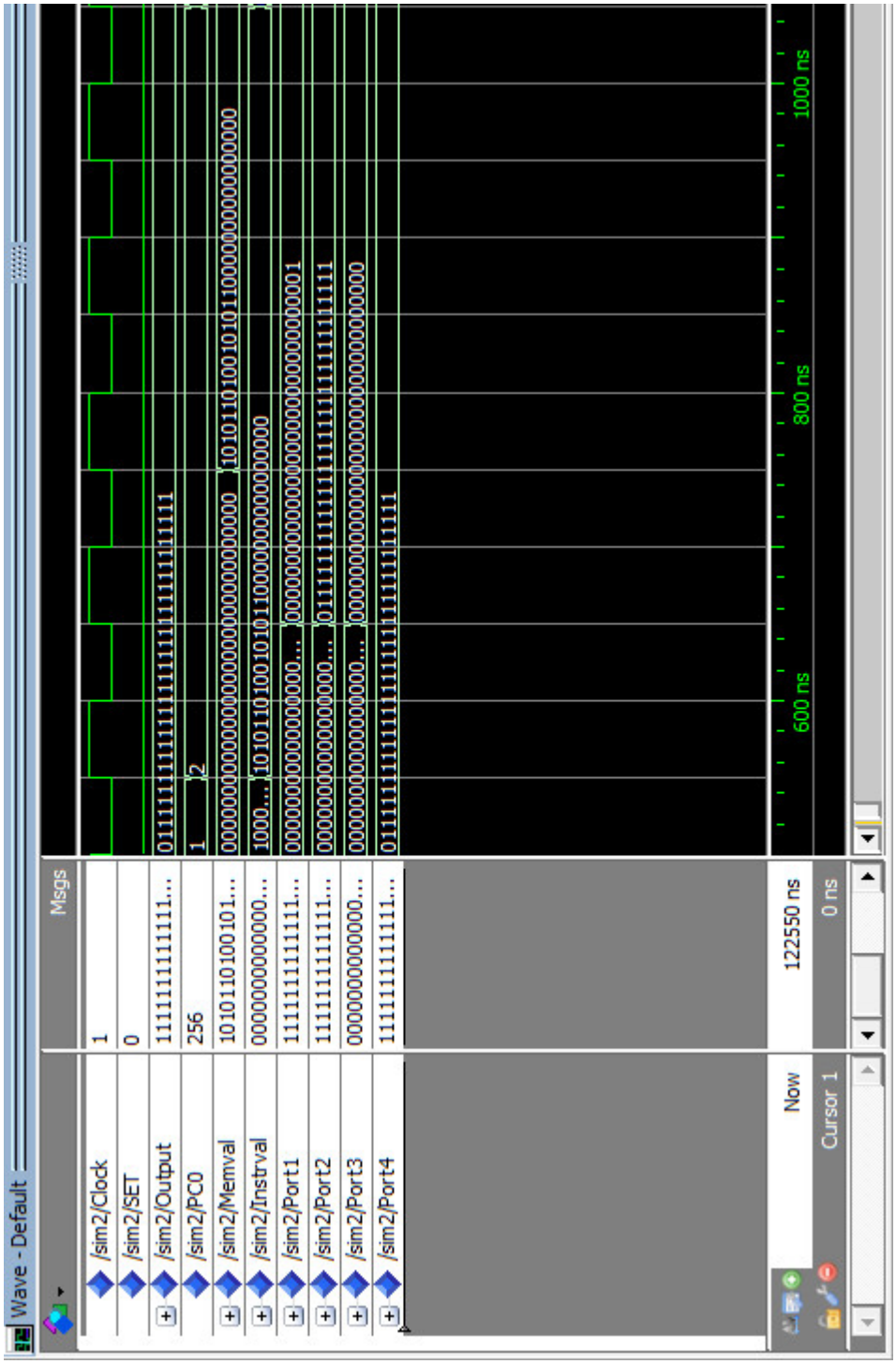

Figure 3: sw $t3,0($t2); *Storing 7FFF FFFF*

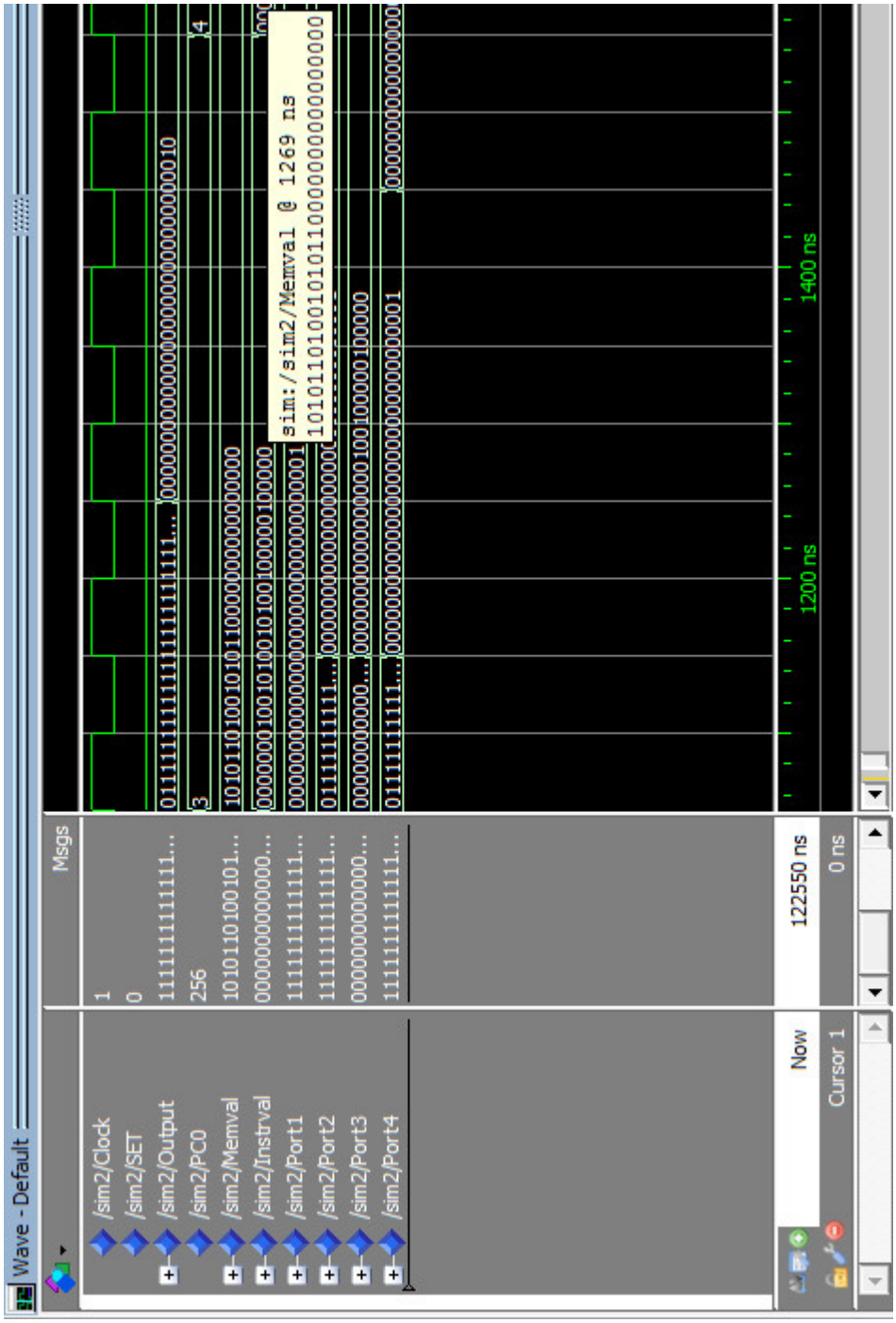

Figure 4: add $t1,$t1,$t1; *$t1 initialized to 1*

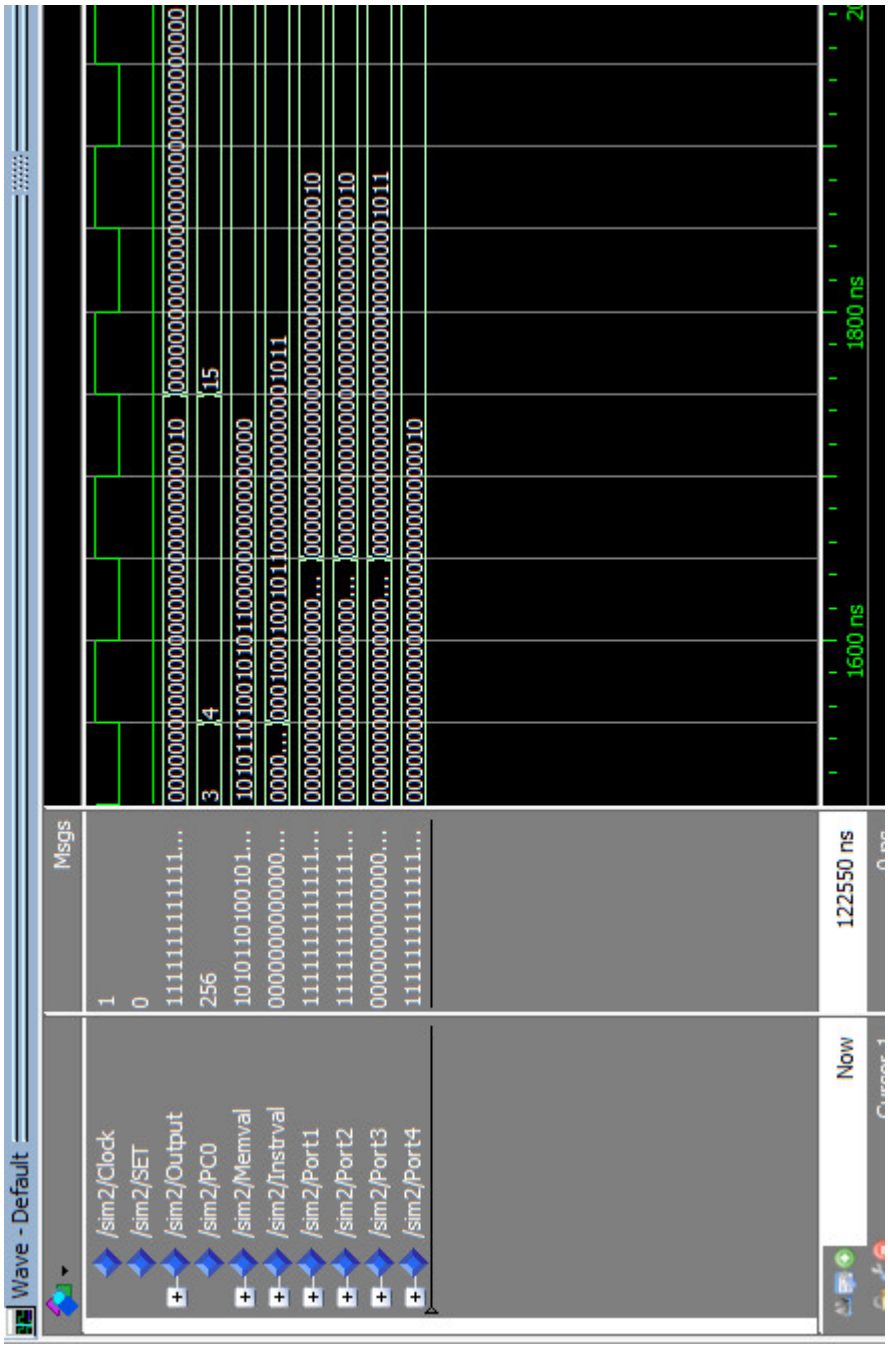

Figure 5: beq $t1,$t4,15; *$t4 initialized to 2, $t1 = 2*

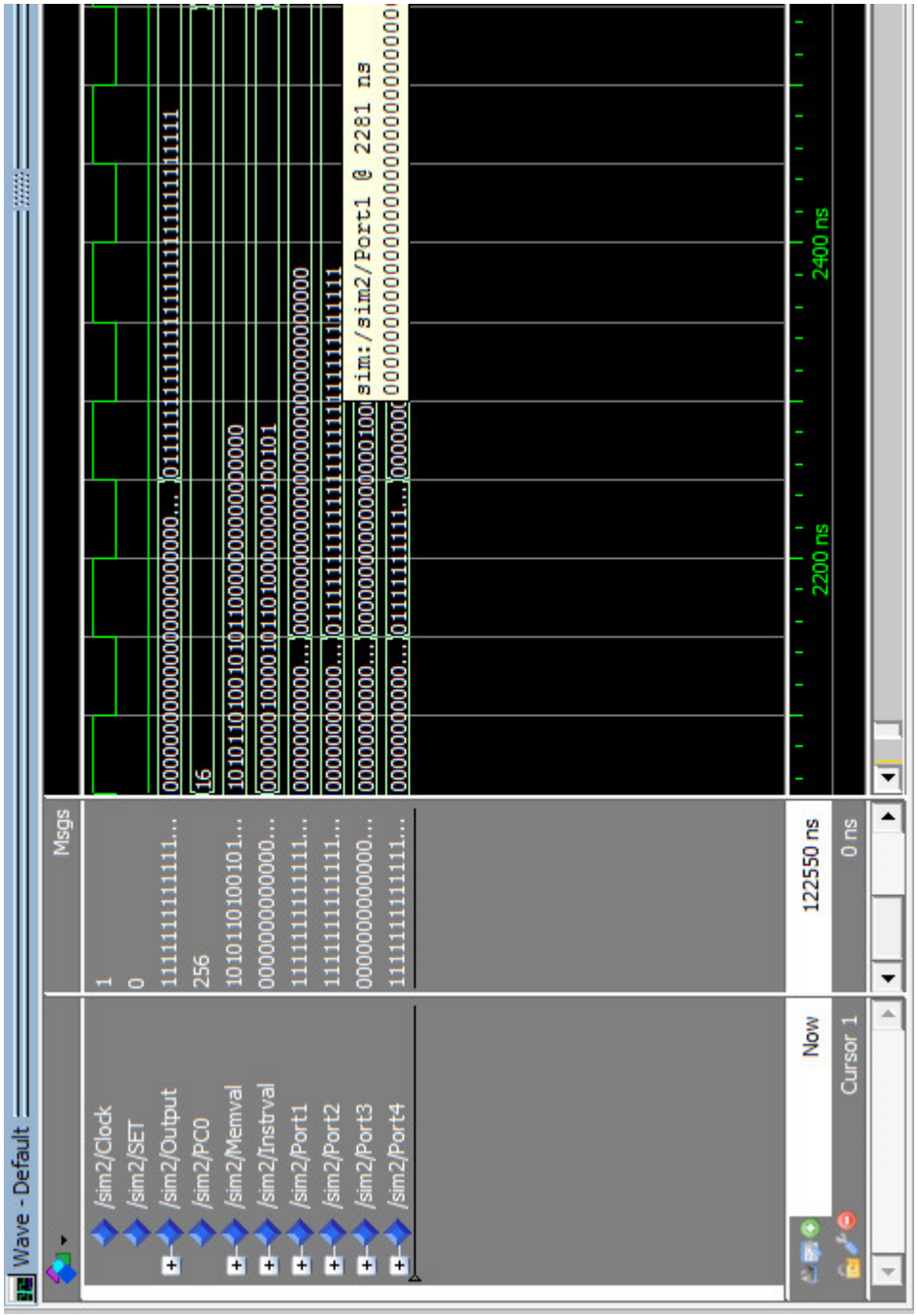

Figure 6: or $t0,$t3,$t0; *$t3 = 7FFF FFFF, $t0 = 0*

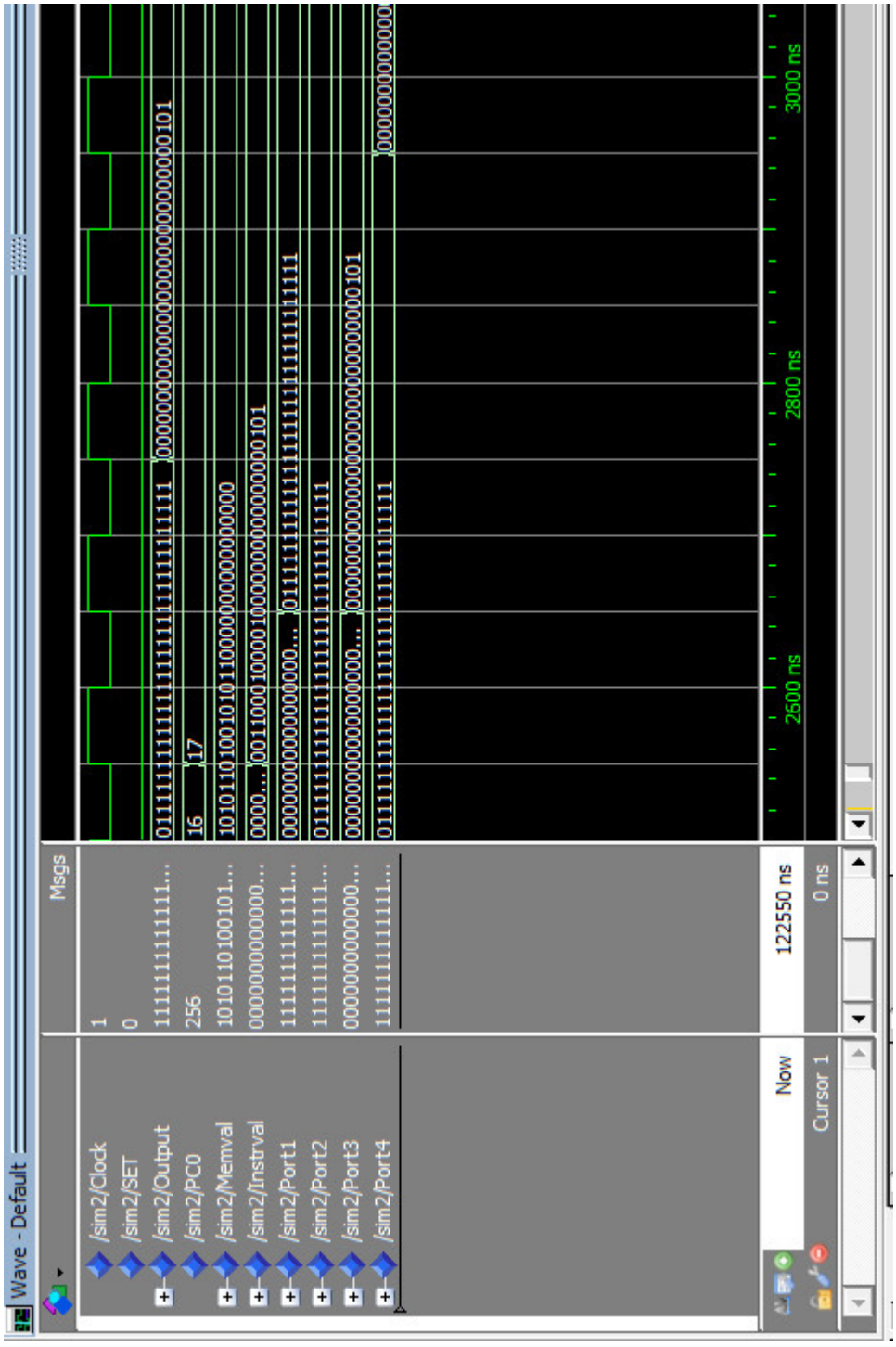

Figure 7: andi $t0,$t0,5; *$t0 = 7FFF FFFF*

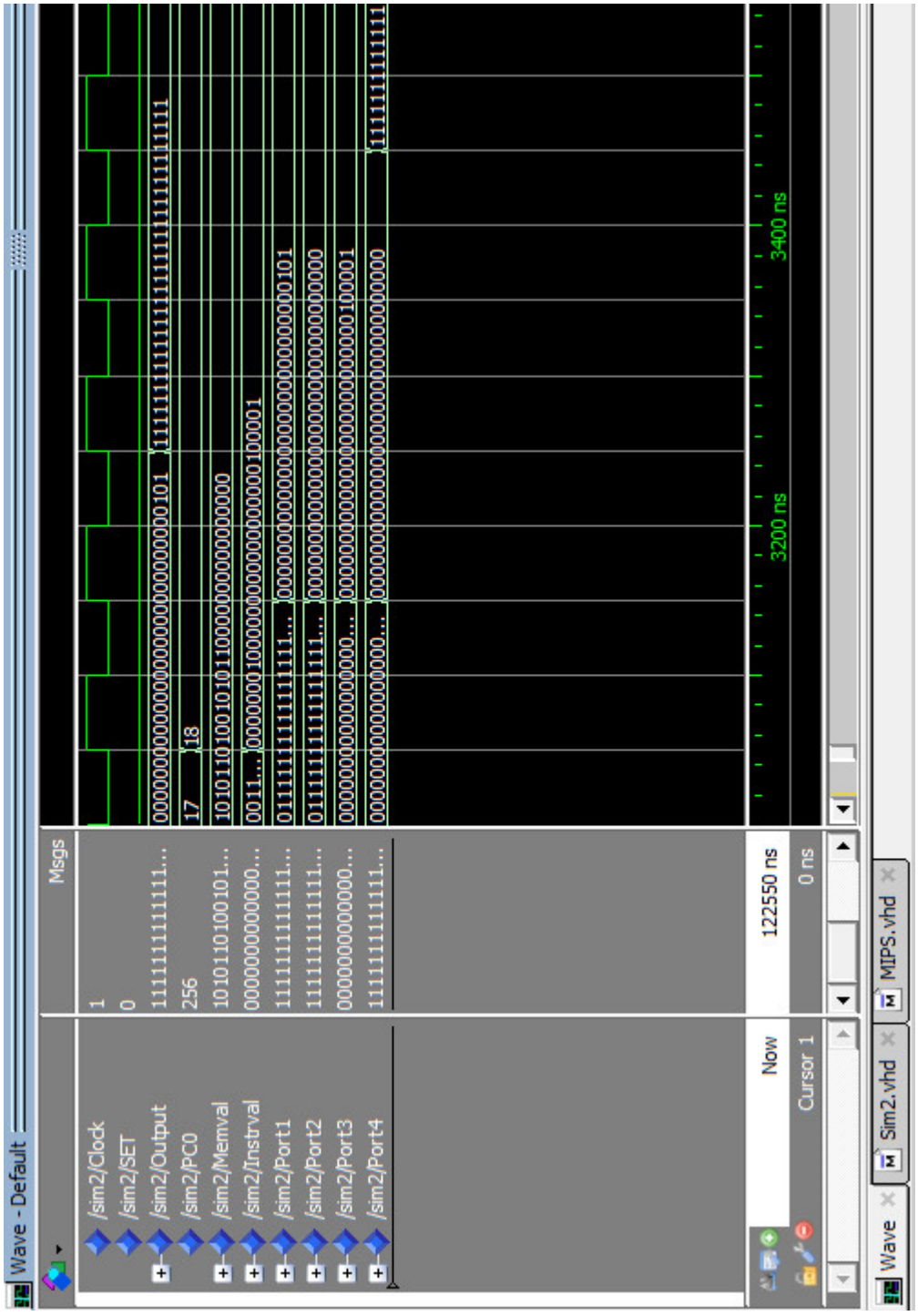

Figure 8: nand $t0,$t0,$zero; *$t0 = 5*

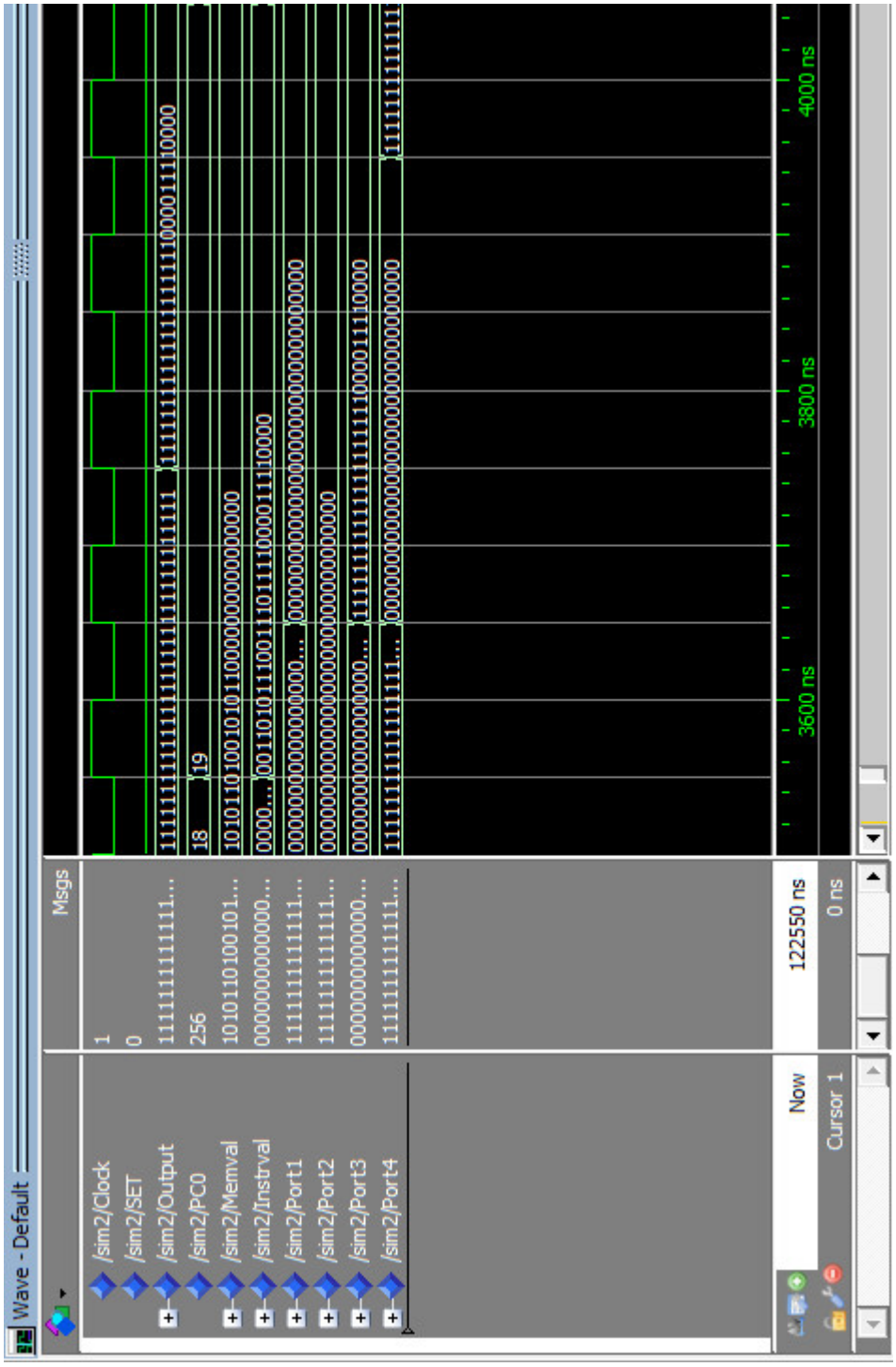

Figure 9: ori $t6,$t6,61680; *$t6 = 0*