Name- Moitrish Maity Roll-48 CSE 4th Sem

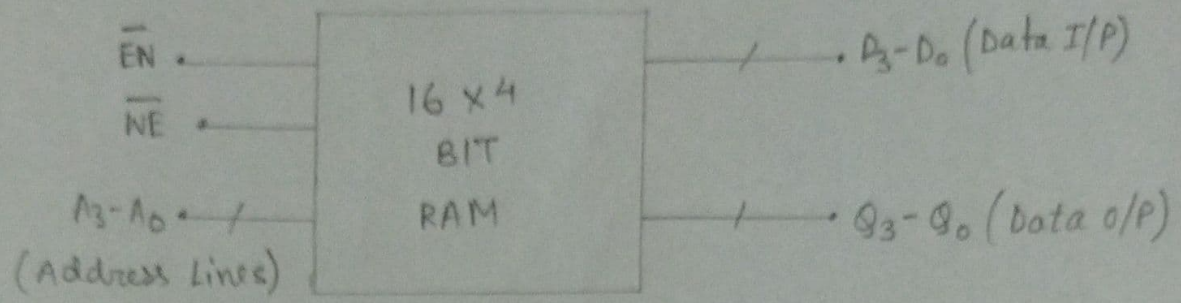
EXPERIMENT-12

- Title: Main memory
- Objective: Implementation of main memory
- Course Outcome: CO6
- Bloom's Level: Comprehension

(12.1) RAM Realization

- Problem statement: Write a verilog code to model a RAM memory cell with all the following consideration - Read/ write signal ('WE'), Address line (A), Select line ('EN' active low), Data I/P (DI), Data o/p (DO).
- Theory: RAM is a form of computer memory that can be read and changed at any order, typically used to store working data and machine codes. RAM is normally associated with volatile types of memory where stored information is lost if power is removed. RAM contains multiplexing and demultiplexing circuitry to connect the data lines to addressed storage for reading or writing the memory entries. Here, we are making 16 X 4 bit RAM. In this RAM, there are 16 different address locations each consist of four bits. Each address can store data of four bits. If chip select line became low, then RAM cell activates. If $\overline{WE}$ became low, RAM cell accepts data and if $\overline{WE}$ becomes high, 'READ' operation is performed.

Logic Diagram:



Truth Table

$\overline{EN}$	$\overline{WE}$	MODE	POWER
H	X	Not selected	Standby
L	L	Write	Active
L	H	Read	Active

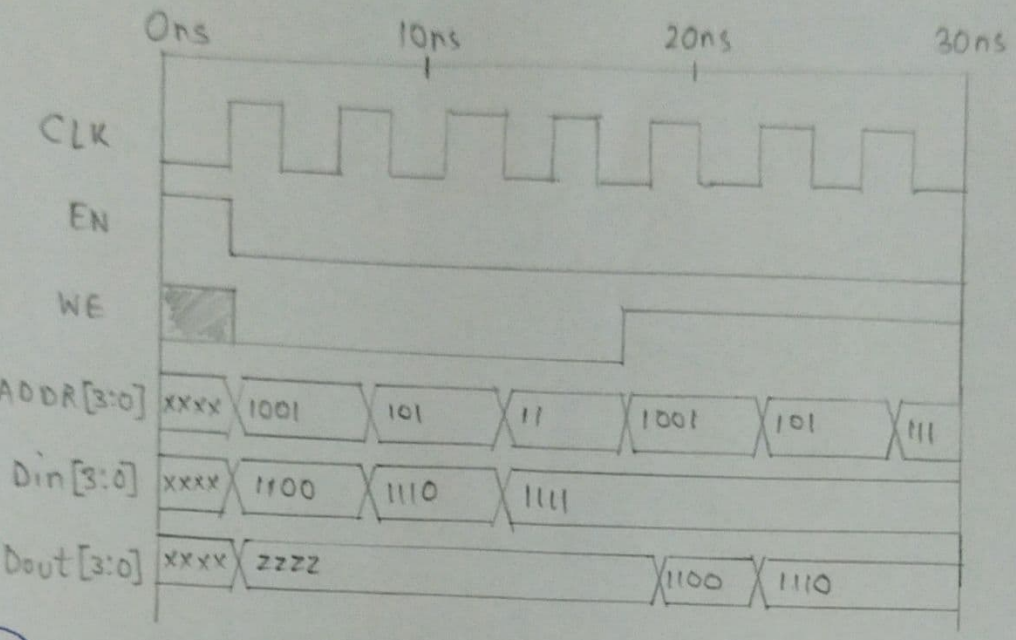
Design Code :

```
module ram_module(input we, input clk, input en, input [3:0] di,
                  input [3:0] addr, output [3:0] dout);

    reg [3:0] dout;
    reg [3:0] ram[15:0];
    always @(posedge clk)
      begin
        if (we == 0 && en == 0) begin
            ram[addr] <= di;
            dout <= 4'bzzzz;
        end
        else if (we == 1 && en == 0) begin
            dout <= ram[addr]; end
        else
          begin
            dout <= 4'bzzzz;
          end
      end
endmodule
```

- Testbench Code:

```
module ram_module_tb();
    reg EN, WE, CLK;
    reg [3:0] Din;
    reg [3:0] ADDR;
    wire [3:0] Dout;
    ram_module uut(.we(WE), .clk(CLK), .en(EN), .di(Din), .addr(ADDR),
                   .dout(Dout));

    initial begin
        $dumpfile("dump.vcd"); $dumpvars(1);
        CLK=0;
        EN=1; #2;
        EN=0; WE=0;
        ADDR = 4'b1001;
        Din = 4'b1100; #5;
        ADDR=4'b0101;
        Din = 4'b1110; #5;
        ADDR = 4'b0011;
        Din = 4'b1111; #5;
        WE=1;
        ADDR = 4'b1001; #5;
        ADDR = 4'b0101; #5;
        ADDR = 4'b0111; #5;
        ADDR = 4'b0011; #5;
        $finish;
    end
    always #2 CLK = ~CLK;
endmodule
```

□ Timing Diagram:



(12.2) RAM cascading

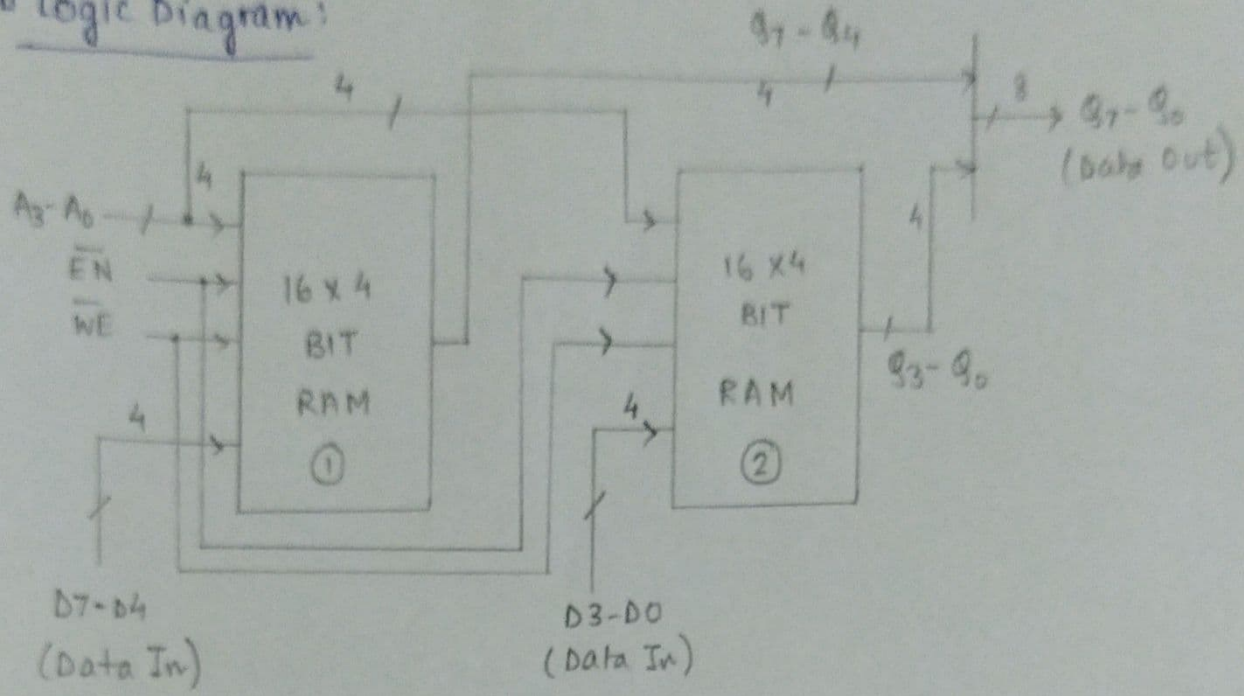
□ Problem statement: Write a verilog program to do the following expansions using 12.1 module initiations in coding.

(A) Horizontal Expansion
(B) Vertical Expansion
(C) Hybrid Expansion.

[A] Horizontal Expansion:

□ Theory: Whenever required data at each address is more than one chip data at each address, we have to do horizontal expansion of chips. In that case, each address line is fed to every chip at same address port. Here, we are making 16×8 bit RAM using two 16×4 bit RAM. Basically here we divide 8 data bits between two RAM chips. One chip can store 7th to 4th bits and another one stores 3rd to 0th bits of data in same address location. Thus, we store a 8 bit data in a 4 bit address space.

Logic Diagram:



Function Table:

$\overline{EN}$	$\overline{WE}$	ADDRESS				DATA-IN								DATA-OUT							
		A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0	q7	q6	q5	q4	q3	q2	q1	q0
H	x	x	x	x	x	x	x	x	x	x	x	x	x	z	z	z	z	z	z	z	z
L	L	L	L	L	H	H	L	H	L	H	H	L	H	z	z	z	z	z	z	z	z
L	L	L	L	H	L	L	L	L	L	L	L	L	L	z	z	z	z	z	z	z	z
L	L	L	H	H	L	L	H	L	H	L	H	L	H	z	z	z	z	z	z	z	z
L	L	H	L	L	L	H	H	H	H	H	H	H	H	z	z	z	z	z	z	z	z
L	H	L	L	L	H	x	x	x	x	x	x	x	x	H	L	H	L	H	H	L	H
L	H	L	L	H	L	x	x	x	x	x	x	x	x	L	L	L	L	L	L	L	L
L	H	L	H	H	L	x	x	x	x	x	x	x	x	L	H	L	H	L	H	L	H
L	H	H	L	L	L	x	x	x	x	x	x	x	x	H	H	H	H	H	H	H	H

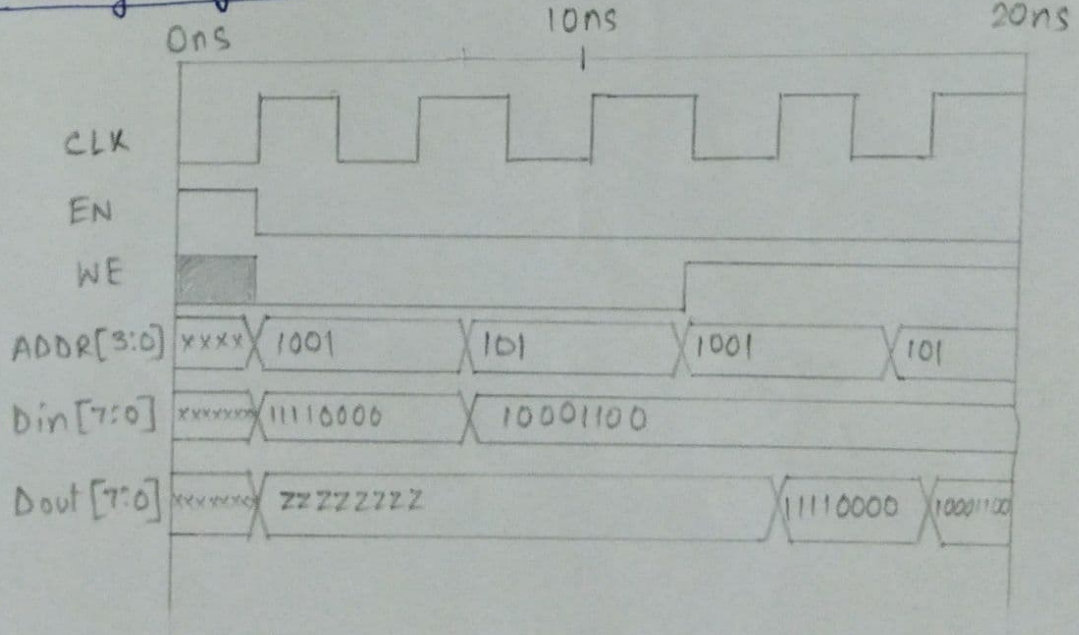
Design Code:

```
module horizontal_cascading (input we, input clk, input en,
                             input [7:0]di, input [3:0]addr, output[7:0] dout);
    ram_module f1(we, clk, en, di[7:4], addr[3:0], dout[7:4]);
    ram_module f2(we, clk, en, di[3:0], addr[3:0], dout[3:0]);
endmodule
```

Testbench Code:

```
module horizontal_cascading_tb();
    reg WE, CLK, EN;
    reg [7:0] Din;
    reg [3:0] ADDR;
    wire [7:0] Dout;
    horizontal_cascading uut (.we(WE),.clk(CLK),.en(EN),.di(Din),
                              .addr(ADDR),.dout(Dout));

    initial begin
        $dumpfile("dump.vcd"); $dumpvars(1);
        CLK=0;
        EN=1; #2; EN=0; WE=0;
        ADDR=4'b1001;
        Din=8'b11110000; #5;
        ADDR=4'b0101;
        Din=8'b10001100; #5;
        WE=1;
        ADDR=4'b1001; #5;
        ADDR=4'b0101; #5;
        $finish;
    end
    always #2 CLK=~CLK;
endmodule
```

Timing Diagram:



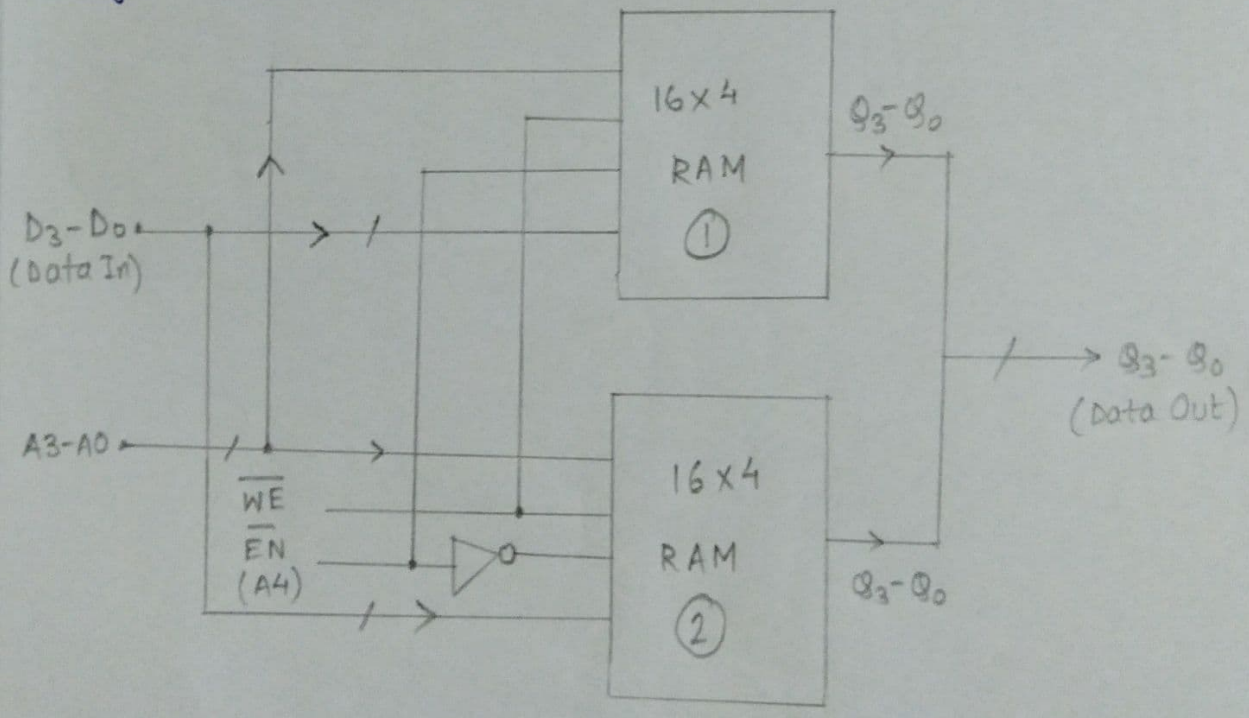
B Vertical Expansion

- **Theory:** Whenever required addresses are more than one single chip addresses we have to do vertical expansion. Here, we have to use 2 chips of 16×4 bit RAM to make a 32×4 bit RAM cell, which has 5 address lines.

1bit	4 bit

→ This 4 bit is fed to every memory cell of 16×4 bit. And when the MSB bit become '0' it will activate (as RAM select line $\overline{EN}$ is active low) the chip which shows first 0-15 addresses and when MSB become '1' it will activate another RAM cell which holds remaining 16-31 addresses. For this implementation, we need a 1 to 2 decoder or a simple 'NOT' operation to select the specific RAM cell. Thus, we get 32×4 bit RAM by using two 16×4 bit RAM cell.

- **Logic Diagram:**



Function Table:

$\overline{EN}$	$\overline{WE}$	ADDRESS LINES					DATA-IN				DATA-OUT			
		A4	A3	A2	A1	A0	D3	D2	D1	D0	Q3	Q2	Q1	Q0
H	X	X	X	X	X	X	X	X	X	X	Z	Z	Z	Z
L	L	L	L	H	L	H	L	H	H	L	Z	Z	Z	Z
L	L	L	L	L	L	L	L	L	L	H	Z	Z	Z	Z
L	L	H	L	H	H	L	H	H	L	L	Z	Z	Z	Z
L	L	H	H	H	H	H	H	L	L	L	Z	Z	Z	Z
L	H	L	L	H	L	H	X	X	X	X	L	H	H	L
L	H	L	L	L	L	L	X	X	X	X	L	L	L	H
L	H	H	L	H	H	L	X	X	X	X	H	H	L	L
L	H	H	H	H	H	H	X	X	X	X	H	L	L	L

Design Code:

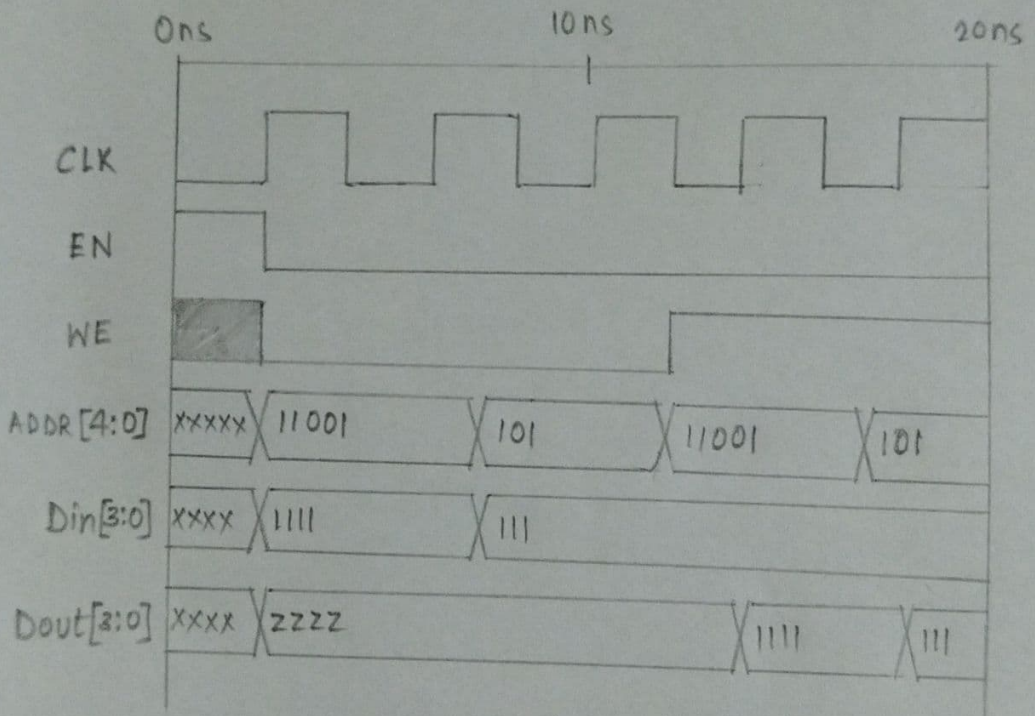
```
module vertical_cascading (input we, input clk, input en,
                           input [3:0]di, input [4:0]addr, output [3:0]out1,
                           output [3:0]out2, output [3:0]dout);

    ram_module f1 (we, clk, en, di[3:0], addr[3:0], out1[3:0]);
    ram_module f2 (we, clk, en, di[3:0], addr[3:0], out2[3:0]);
    assign dout = addr[4] ? out2 : out1;
endmodule
```

Testbench Code:

```
module vertical_cascading_tb();
    reg WE, CLK, EN;
    reg [3:0]Din;
    reg [4:0]ADDR;
    wire [3:0]Dout;
    vertical_cascading out(.we(WE), .clk(CLK), .en(EN), .di(Din),
                           .addr(ADDR), .dout(Dout));

    initial begin
        $dumpfile("dump.vcd"); $dumpvars(1);
        CLK=0;
        EN=1; #2; EN=0; WE=0;
```

```
        ADDR = 5'b11001;
        Din = 4'b1111; #5;
        ADDR = 5'b00101;
        Din = 4'b0111; #5;
        WE = 1;
        ADDR = 5'b11001; #5;
        ADDR = 5'b00101; #5;
        $finish;
    end
    always #2 CLK = ~CLK;
endmodule
```

□ Timing Diagram:

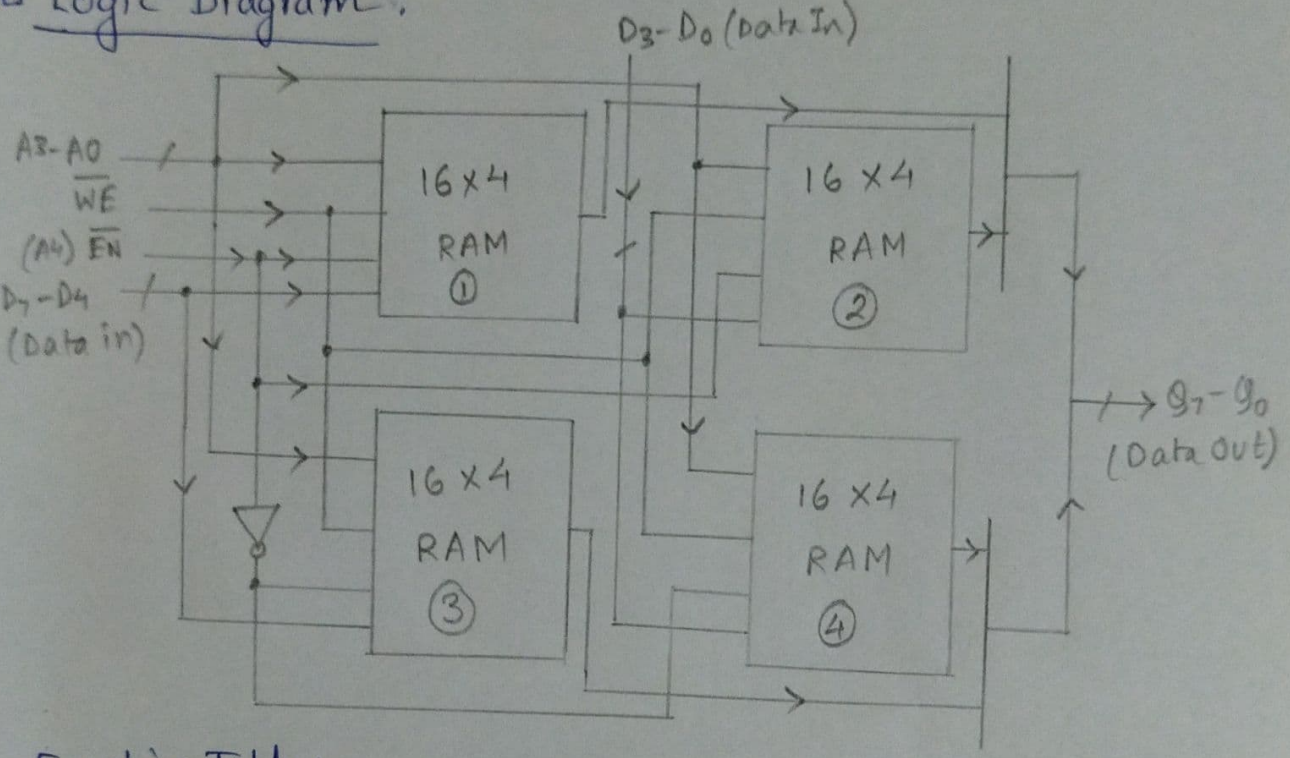


[c] Hybrid Expansion

□ Theory: Whenever required data and required address bits both are more than one single chip, then we have to do a hybrid cascading. It's actually a combination of vertical and horizontal cascading. Here, we are making a 32x8 bit RAM cell by using four $\left(\frac{32 \times 8}{16 \times 4}\right)$ 16x4 bit RAM cells. Here, in each rows, there are two 16x4 bit RAM cells. Total no. of rows is 2. Each RAM cells are

fed with same no. of address bits i.e. 4 at same address port. Ram cells in same rows are connected with a single select ($\overline{EN}$) line. So, at a time, a single row can be activated by the MSB bit of the 5 bit address line generated by CPU. Thus hybrid expansion is done.

Logic Diagram:



Function Table:

$\overline{EN}$	$\overline{WE}$	ADDRESS					DATA-IN								DATA-OUT							
		A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
H	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Z	Z	Z	Z	Z	Z	Z	Z
L	L	L	L	L	L	L	H	L	L	H	H	L	L	H	Z	Z	Z	Z	Z	Z	Z	Z
L	L	L	H	L	H	L	L	L	H	L	L	L	H	L	Z	Z	Z	Z	Z	Z	Z	Z
L	L	L	H	H	L	H	L	L	L	H	H	L	L	L	Z	Z	Z	Z	Z	Z	Z	Z
L	L	H	L	L	L	L	H	H	L	H	L	H	L	L	Z	Z	Z	Z	Z	Z	Z	Z
L	H	L	L	L	L	L	X	X	X	X	X	X	X	X	H	L	L	H	H	L	L	H
L	H	L	H	L	H	L	X	X	X	X	X	X	X	X	L	L	H	L	L	L	H	L
L	H	L	H	H	L	H	X	X	X	X	X	X	X	X	L	L	L	H	H	L	L	L
L	H	H	L	L	L	L	X	X	X	X	X	X	X	X	H	H	L	H	L	H	L	L

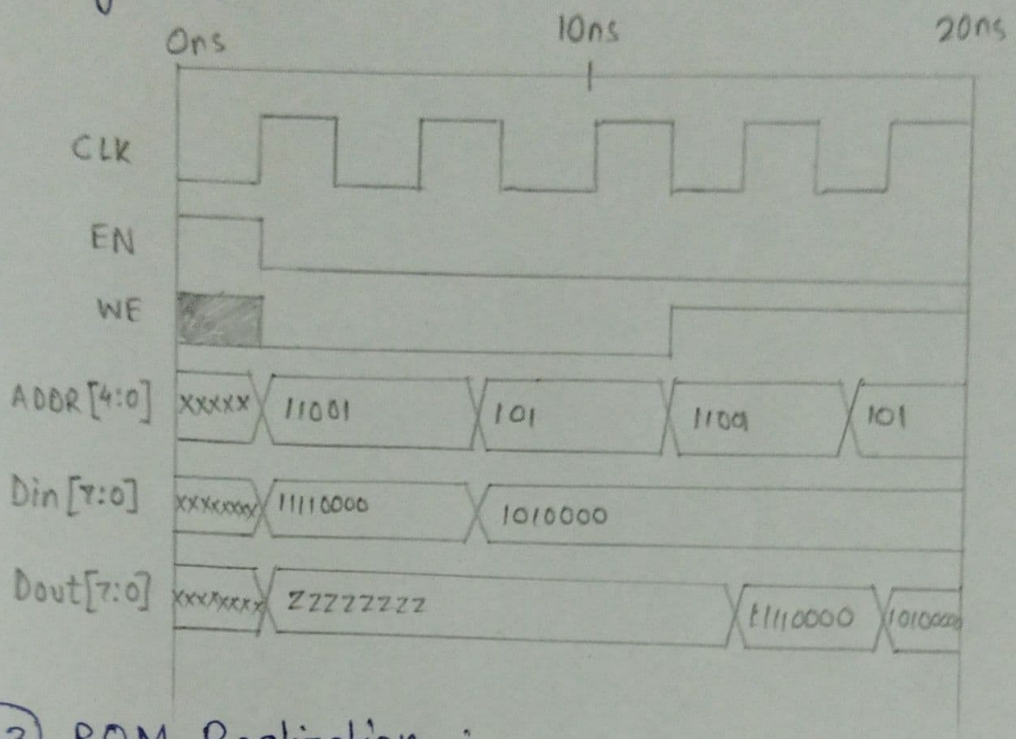
Design Code:

```
module horizontal_cascading (input we, input clk, input en,
                             input [7:0]di, input [3:0]addr,
                                 output [7:0]dout);
    ram_module f1(we, clk, en, di[7:4], addr[3:0], dout[7:4]);
    ram_module f2(we, clk, en, di[3:0], addr[3:0], dout[3:0]);
endmodule
module hybrid_cascading (input we, input clk, input en, input [7:0]di,
                         input [4:0]addr, output [7:0]dout, output [7:0]out1,
                             output [7:0]dout2);
    horizontal_cascading uut1(we, clk, en, di[7:0], addr[3:0], out1[7:0]);
    horizontal_cascading uut2(we, clk, en, di[7:0], addr[3:0], out2[7:0]);
    assign dout[7:0] = addr[4] ? out1[7:0] : out2[7:0];
endmodule
```

Testbench Code:

```
module hybrid_cascading_tb();
    reg WE, CLK, EN;
    reg [7:0] Din;
    reg [4:0] ADDR;
    wire [7:0] Dout;
    hybrid_cascading uut (.we(WE), .clk(CLK), .en(EN), .di(Din),
                          .addr(ADDR), .dout(Dout));
      initial begin
        $dumpfile("dump.vcd"); $dumpvars(1);
        CLK=0;
        EN=1; #2; EN=0; WE=0;
        ADDR=5'b11001;
        Din=8'b11110000; #5;
        ADDR=5'b00101;
        Din=8'b01010000; #5;
        WE=1;
        ADDR=5'b11001; #5;
        ADDR=5'b00101; #5; $finish; end
    always #2 CLK=~CLK;
endmodule
```

□ Timing Diagram:

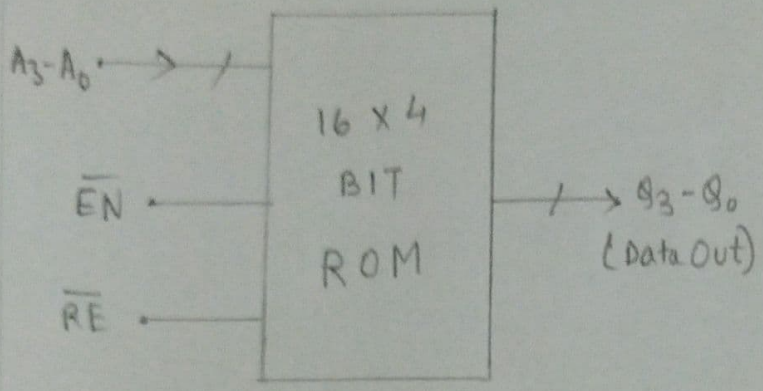


(12.3) ROM Realization :

□ Problem Statement: Write a verilog program to model a ROM memory cell with all following considerations : Read Signal ($\overline{RE}$), Address line (A), Select Line (EN' active low), Data O/P (DO).

□ Theory: ROM stands for Read Only Memory. The memory from which we can only read but can't write on it. A ROM stores such instructions that are required to start a computer. This operation is referred to as "bootstrap". Here, we are designing 16x4 bit ROM which has 4 address and 4 data lines. ROM cell can be activated by setting $\overline{EN}$ at zero as this is active low. ROM cell can read data by setting the value of $\overline{RE}$ at 0.

Logic Diagram:



Truth Table :

$\overline{EN}$	$\overline{RE}$	MODE	POWER
H	X	Not Selected	Standby
L	L	Read	Active
L	H	Not Selected	Standby

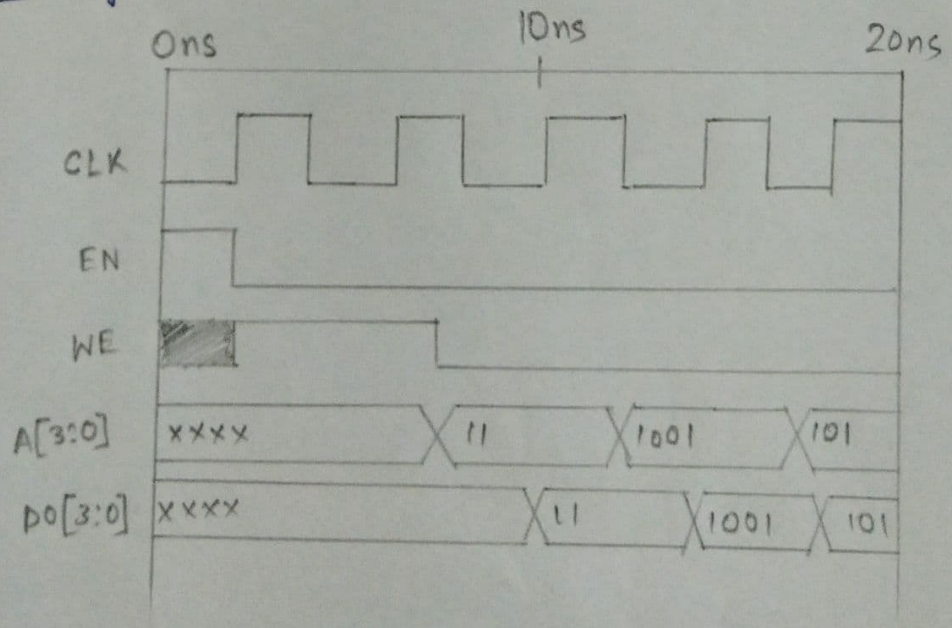
Design Code :

```
module rom_module (input re, input clk, input en, input [3:0]a,
                   output [3:0] dout);

    reg [3:0]dout;
    reg [3:0]rom [15:0];
    always @( posedge clk)
       begin
          if (re==0 && en==0) begin
              dout <= rom[a]; end

       end
    initial begin
       rom [4'b0000] = 4'b0000;
       rom [4'b0001] = 4'b0001;
       rom [4'b0010] = 4'b0010;
       rom [4'b0011] = 4'b0011;
       rom [4'b0100] = 4'b0100;
       rom [4'b0101] = 4'b0101;
       rom [4'b0110] = 4'b0110;
       rom [4'b0111] = 4'b0111;
       rom [4'b1000] = 4'b1000;
       rom [4'b1001] = 4'b1001;
       rom [4'b1010] = 4'b1010;
       rom [4'b1011] = 4'b1011;
       rom [4'b1100] = 4'b1100;
       rom [4'b1101] = 4'b1101;
       rom [4'b1110] = 4'b1110;
       rom [4'b1111] = 4'b1111;

    end
endmodule
```

Testbench Code:

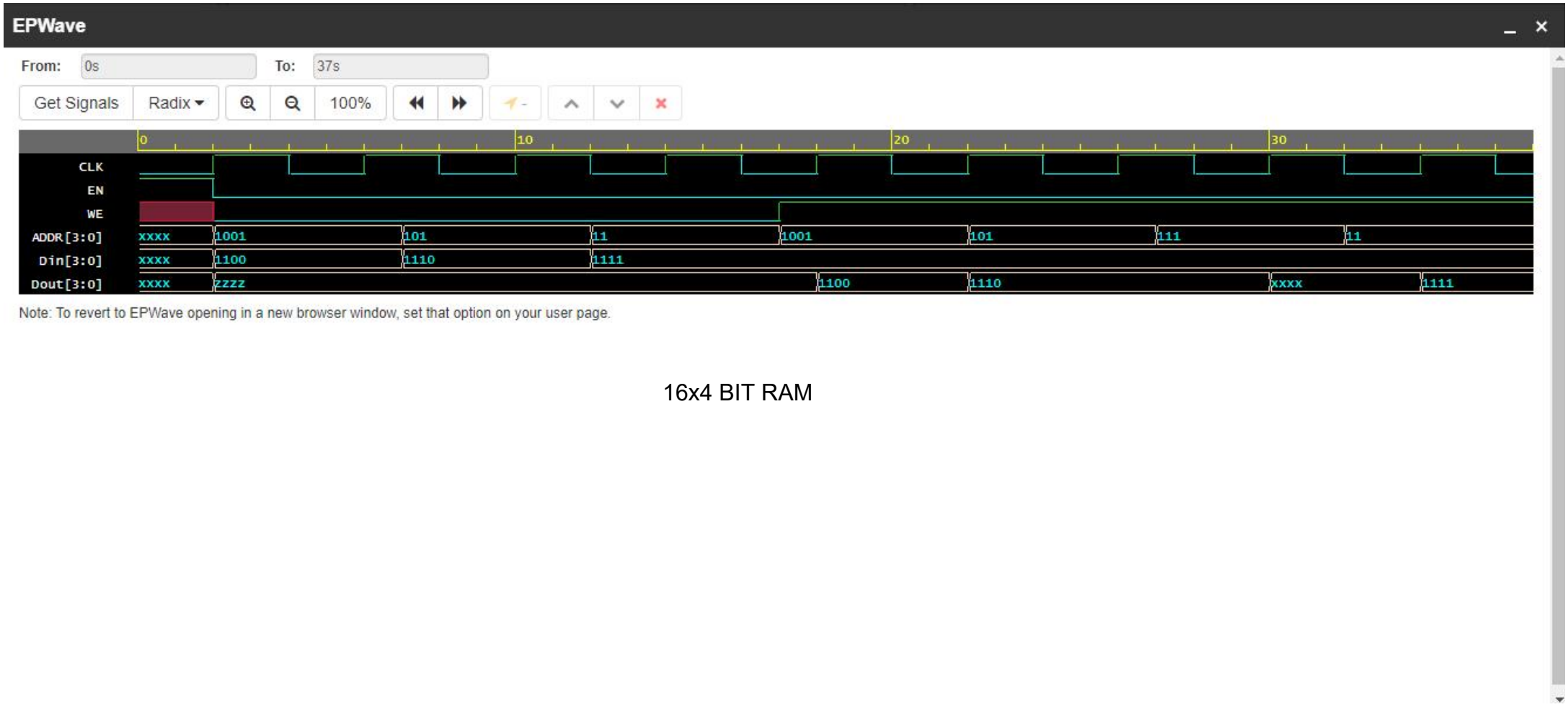
```
module rom_module_tb();
    reg RE, EN, CLK;
    reg [3:0] A;
    wire [3:0] DO;
    rom_module uut (.re(RE), .clk(CLK), .en(EN), .a(A), .dout(DO));
    initial begin
        $dumpfile("dump.vcd"); $dumpvars(1);
        CLK=0;
        EN=1; #2; EN=0;
        RE=1; #5;
        A=4'b1001; RE=0;
        A=4'b0011; #5;
        A=4'b1001; #5;
        A=4'b0101; #5;
        A=4'b0111; #5;
        $finish;
    end
    always #2 CLK=~CLK;
endmodule
```

Timing Diagram:

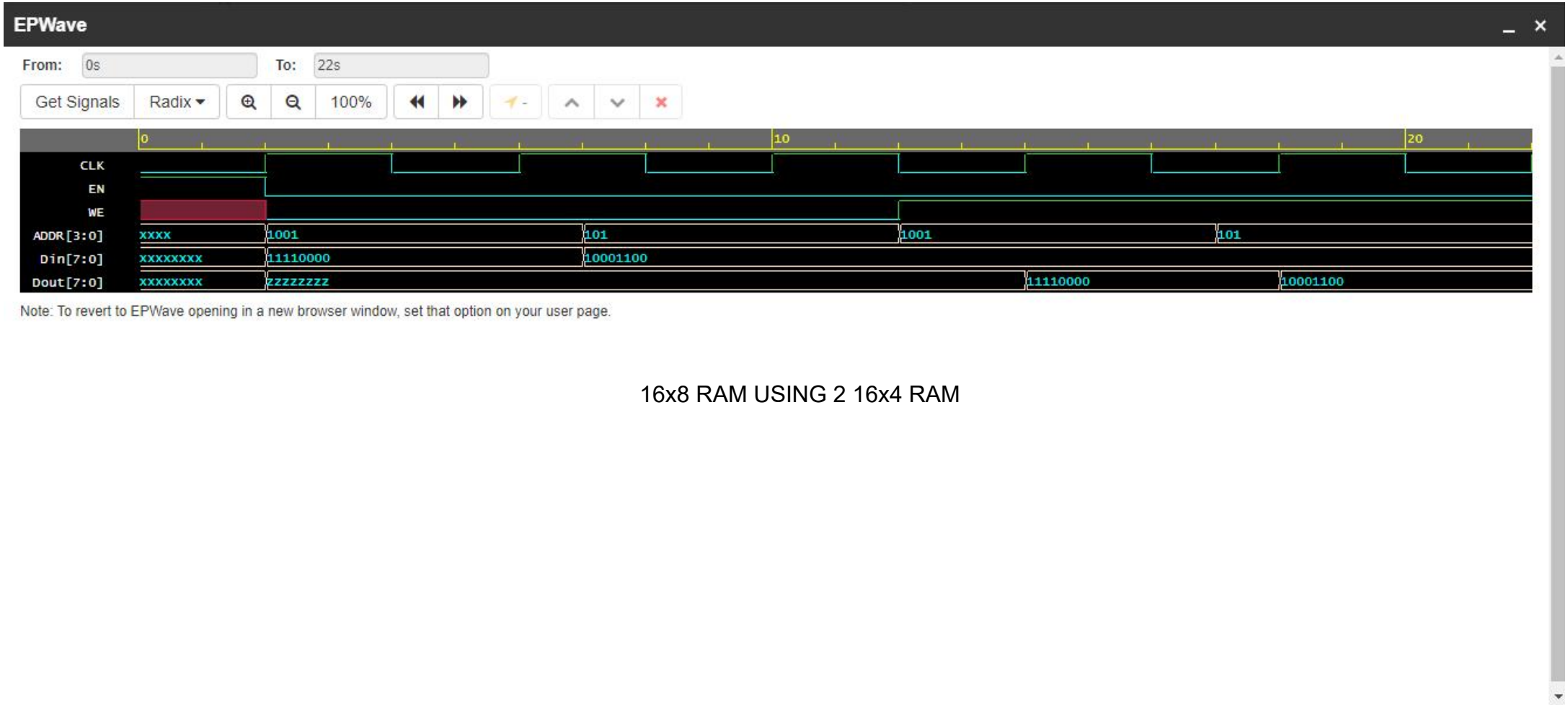


□ Discussion: In this experiment, we have implemented various memory systems like RAM, RAM expansions, ROM etc. We came to know that this memory modules are one of the most important parts of a computer system. We have learned various HDL terms also.

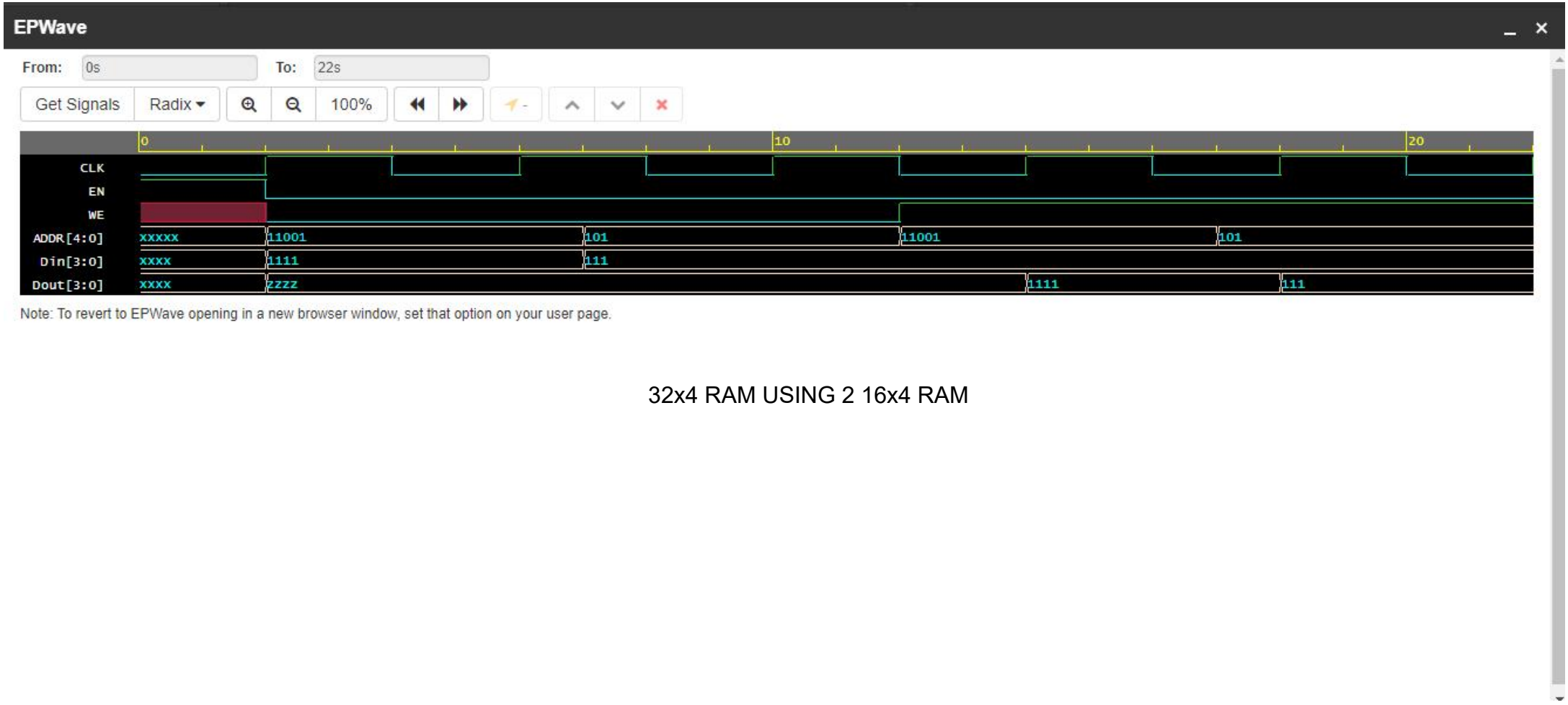
□ Justification of CO: In this experiment, we have designed various memory systems like RAM, ROM etc. by using verilog code which fulfils conditions of CO6. So, CO6 is justified.


EPWave
From:
0s
To:
37s
Get Signals
Radix
100%
0
10
20
30
CLK
EN
WE
ADDR[3:0]
Din[3:0]
Dout[3:0]
xxxx
1001
101
11
1001
101
111
11
xxxx
1100
1110
1111
xxxx
zzzz
1100
1110
xxxx
1111
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

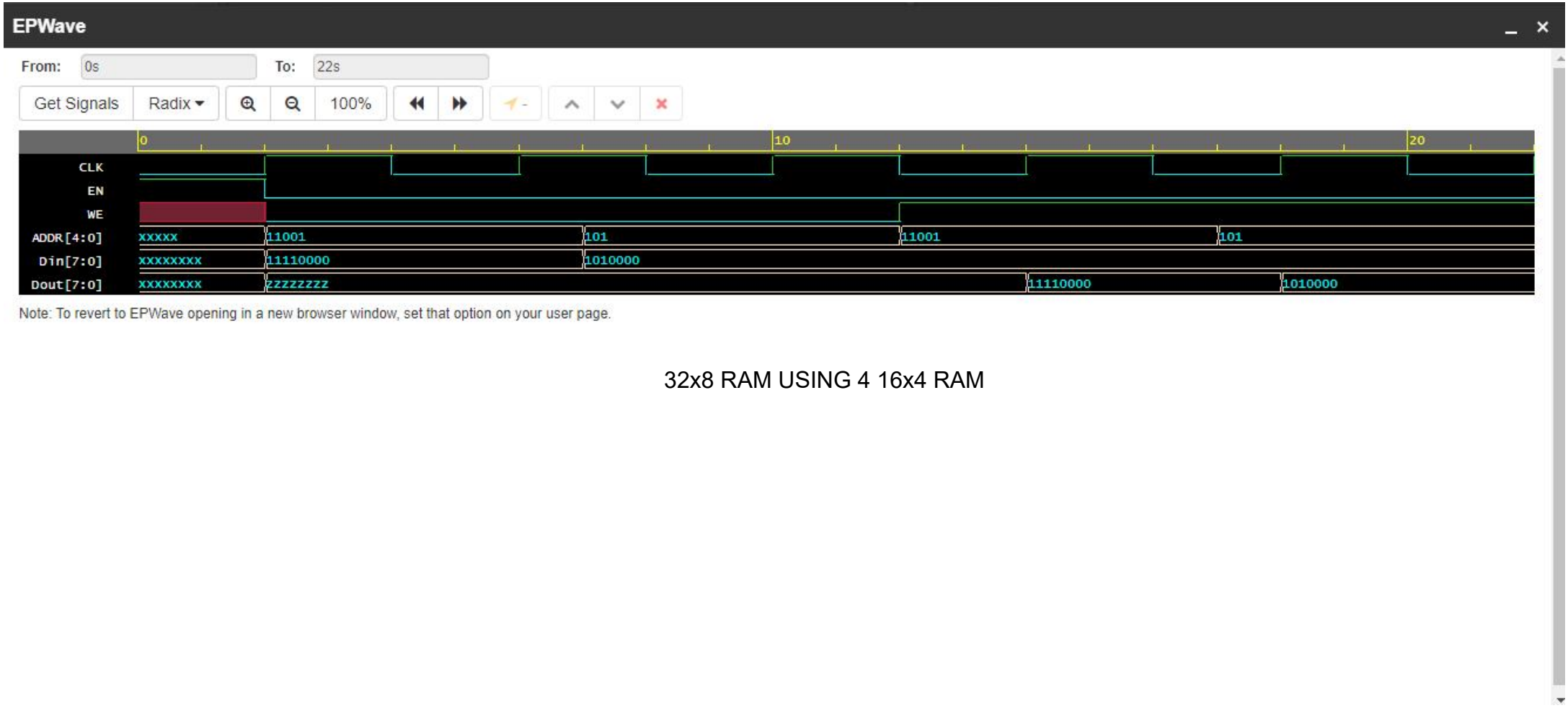
16x4 BIT RAM



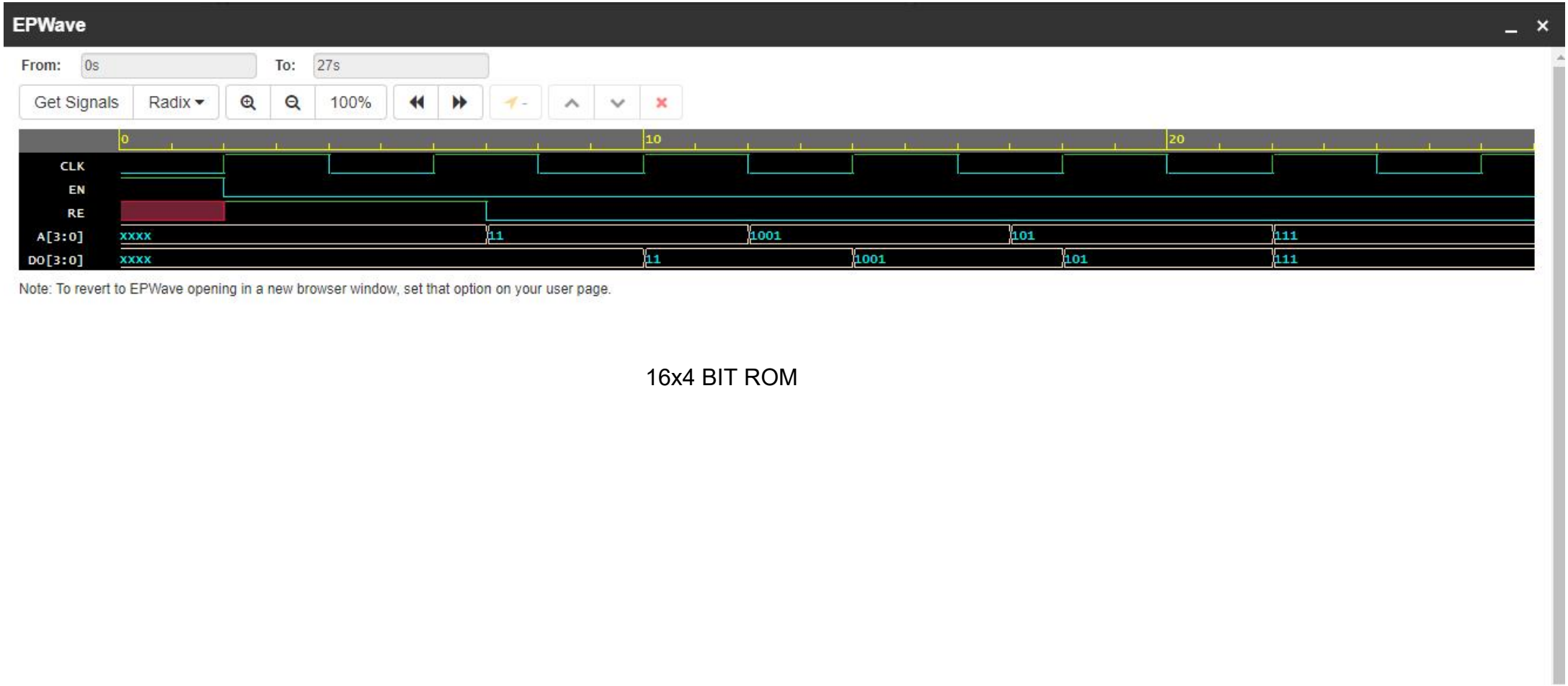
16x8 RAM USING 2 16x4 RAM



32x4 RAM USING 2 16x4 RAM



32x8 RAM USING 4 16x4 RAM


EPWave
From:
0s
To:
27s
Get Signals
Radix
100%
0
10
20
CLK
EN
RE
A[3:0]
DO[3:0]
xxxx
11
1001
101
111
xxxx
11
1001
101
111
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

16x4 BIT ROM