EECS150: Homework 6, MIPS Processor, RAM

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

1. DDCA 8.24

   See Figure 1 for the MIPS code.

Figure 1 MIPS code for DDCA 8.24

```
#  MIPS code for Exercise 8.24
#  The LEDs are mapped to the 5 least significant bits at
#  memory address 0xFFFFFF14

       addi $t1, $0, 0x15    # the random pattern to display to the LEDs
Loop0: lw   $t0, 0xFF10($0)  # $t0 = button value
       beq  $t0, $0, Loop0   # if button is not pressed, keep checking
       sw   $t1, 0xFF14($0)  # if $t0 == 1, display pattern on LEDs
Loop1: lw   $t0, 0xFF10($0)  # $t0 = button value
       bne  $t0, $0, Loop1   # if button is still pressed, loop
       sw   $0, 0xFF14($0)   # clear LEDs when button is not pressed
       j    Loop0            # check when button is pressed
```

   See Figure 2 for the circuit schematic.

   See Figure 3 for the verilog.

2. Given two **256 x 32 single-port** RAMs, construct a **256 x 32 simple dual-port** RAM. You are allowed to use any additional gates, multiplexers or flip flops, but are limited to a total of 256 1-bit flip flops. Draw the schematic.

   To do this, we will first build a 256 x 1 **residency table** using our array of flip flops. The job of the residency table is to keep track of where (which of the two RAMs) we stored the data. Since flip flops are inherently simple dual-port structures, the residency table is essentially a 256 x 1 SDP RAM. See Figure 4 for the implementation. It is interesting to note that, if you were not restricted on the number of flip flops, you could build the entire 256 x 32 SDP RAM by putting 32 of the 256 x 1 SDP RAMs in parallel.

   The 256 x 32 SDP RAM we are trying to build will have both a read-only port and a write-only port, each with their own independent addresses. In order to implement the read-only port, we will need to add extra hardware that does the following things:

   (a) Read from the residency table at the read address in order to find out where we stored the data that wants to be read.

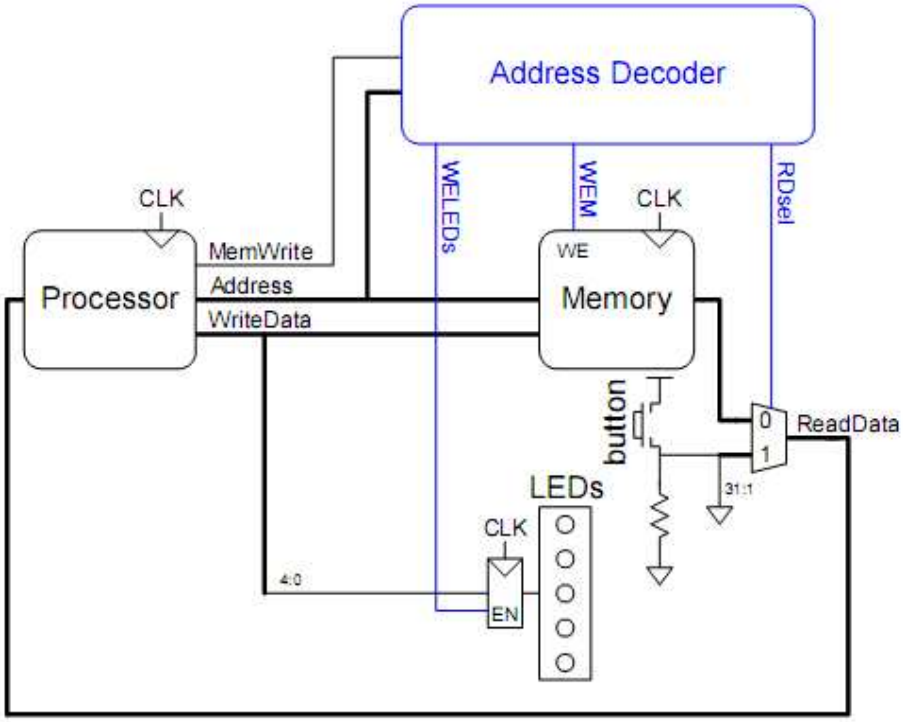
   (b) Read from the RAM that stores the data

   (c) Output the read data from the RAM that stores the data

   In order to implement the write-only port, the hardware will need to do the following things:

   (a) Using the data output from the residency table, find out which RAM we will be reading from this cycle.

Figure 2 Circuit Schematic for DDCA 8.24



(b) Write to the RAM that we are **not reading from**.

(c) Update the residency table with where we wrote the data.

The hardware that implements these functionalities is shown in Figure 5.

3. Given four **1k x 16 simple dual-port** RAMs, construct a **4k x 16 simple dual-port** RAM. You are allowed to use any additional gates or multiplexers. Draw the schematic.

   The idea behind this is simple. A 4k x 16 SDP RAM will have 12 address bits. We will tie the bottom 10 bits of the 12-bit address to both of the address ports on each of the 1k x 16 SDP RAMs.

   To handle the reads of the composite RAM correctly, we will use a 4:1 multiplexer. Each input of the multiplexer is the output of one of the 4 RAMs with the selector signal being the top 2 bits of the 12-bit address. To handle the writes of the composite RAM correctly, we will decode the top 2 address bits to figure out which RAM we will be writing to.

   The overall schematic is shown in Figure 6.

   As a side note, it does not really matter whether you split the RAM up using the top 2 bits, the bottom 2 bits, or even 2 arbitrary bits somewhere in the middle of the address. The only reason I used the top 2 bits is because it was easier for me to picture.

4. For this section, you will be taking another look at LUTs.

   (a) Implement a basic 4-input LUT down to the transistor level. Allow the LUT to be programmable with a new function whenever the `prog` signal is asserted. How many configuration bits does your LUT need?

   See Figure 7. Note that I am using **positive level-sensitive latches** instead of registers for storing the programmed values. The main reason is just due to transistor counts, as latches use fewer transistors than a register.

   We will need 16 configuration bits for the LUT, one bit for each latch.

Figure 3 Verilog for DDCA 8.24

```
module addrdec(input [31:0] addr, input memwrite,
               output reg   WELEDs, Mwrite,
               output reg   rdselect);

  parameter  B    = 16'hFF10;    // push button
  parameter  LEDs = 16'hFF14;    // LEDs

  wire [15:0] addressbits;

  assign addressbits = addr[15:0];

  always @ ( * )
    if (addr[31:16] == 16'hFFFF) begin
      // writedata control
      if (memwrite)
        if (addressbits == LEDs)
          {WELEDs, Mwrite, rdselect} = 3'b100;
        else
          {WELEDs, Mwrite, rdselect} = 3'b010;

      // readdata control
      else
        if ( addressbits == B )
          {WELEDs, Mwrite, rdselect} = 3'b001;
        else
          {WELEDs, Mwrite, rdselect} = 3'b000;
    end
    else
      {WELEDs, Mwrite, rdselect} =
        {1'b0, memwrite, 1'b0};

endmodule
```

(b) Modify your basic 4-input LUT such that it may also be used as a **16 x 1 Simple Dual Port RAM** (with synchronous writes) whenever the configuration bit `ram` is set. It should take `WriteEnable, WriteAddress, DataIn, Clock` as additional inputs. Draw the schematic of the new circuit down to the level of transistors.

In order to do this, we must add a 4-to-16 decoder for the write addresses and **change the latches we used into registers**. We will also need to add a mux in front of each register, so that they take data from `DataIn` instead of the programming data when `prog` is low. See Figure 8 for the implementation.

A RAM cell is implemented in Figure 9.

(c) Modify the new 4-input LUT such that it can also be used as a 16-bit shift register whenever the configuration bit `shift` is set.

This is trivial, expand upon the mux in front of the input of each register, such that they may also get data from the register below them. See Figure 10 for what each new cell looks like.

5. Using the Block RAM section of the Virtex-5 datasheet as a reference, **roughly estimate** the number of transistors it will take to implement a 36k Block RAM on the Virtex-5. Clearly explain and justify the number you came up with. (Hint: some factors you may wish to consider are width, depth, number of ports, etc.)

   Note: answers to this problem will vary, as we are making wild guesses about an implementation

Figure 4 Implementation of a 256 x 1 SDP Residency table using 256 flip flops. The gate-level implementation of the 8-to-256 decoder is shown on the right. In general, a $n - to - 2^n$ decoder can be made using 2^n n-input AND gates.

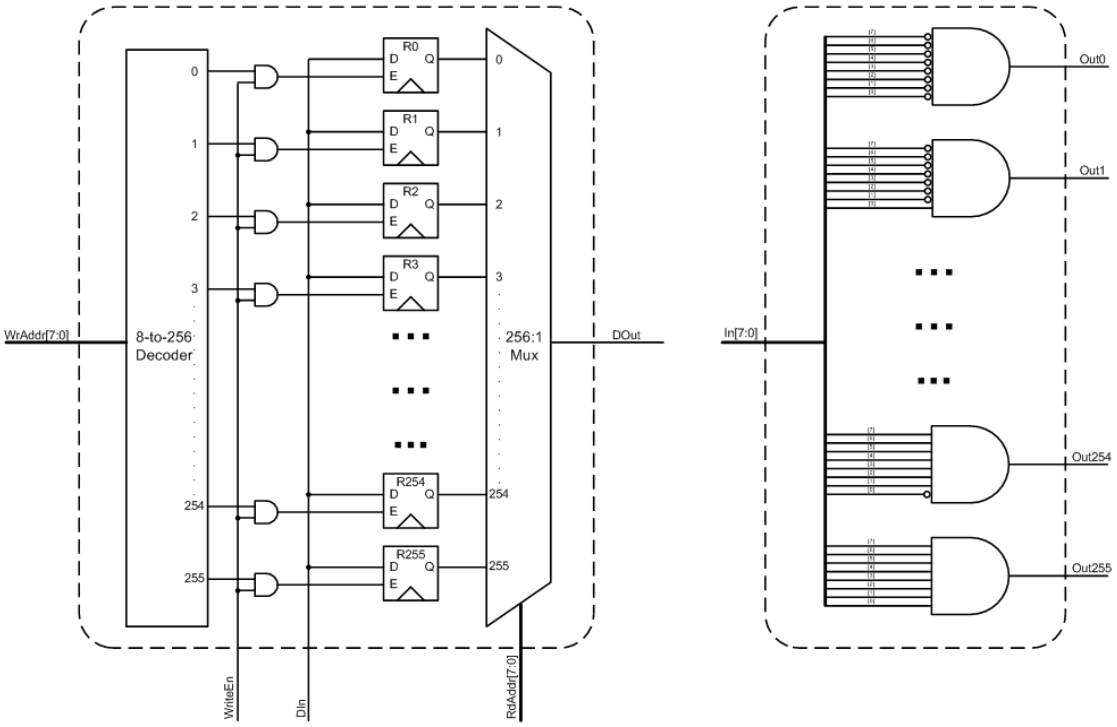


Figure 5 Implementation of a 256 x 32 SDP RAM using 2 instances of 256 x 32 Single-port RAMs

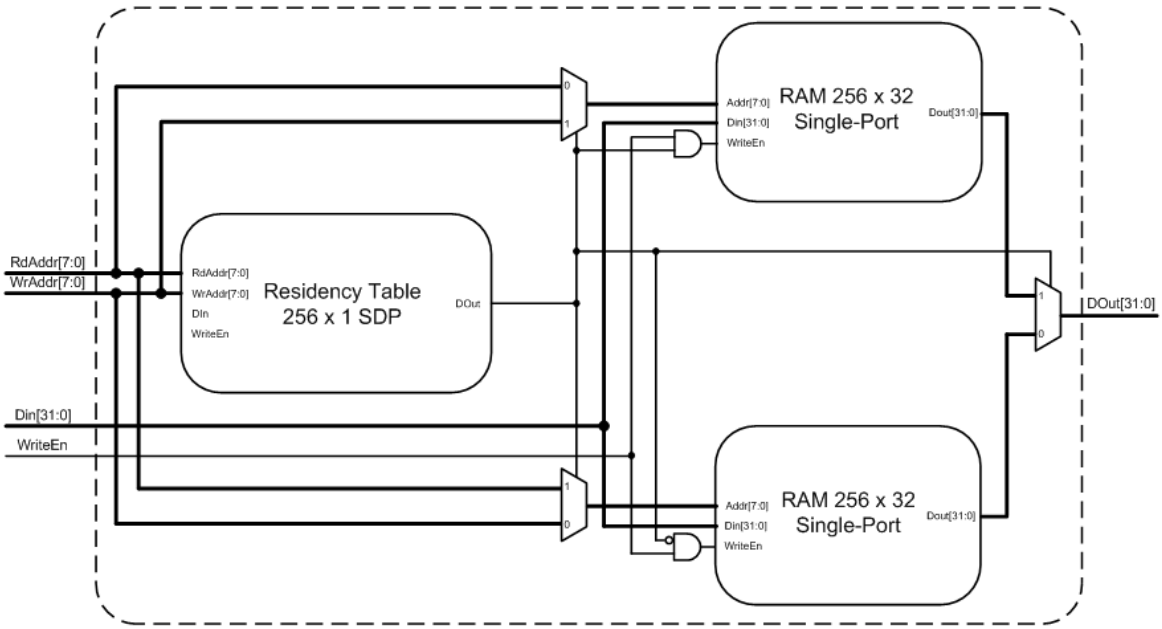
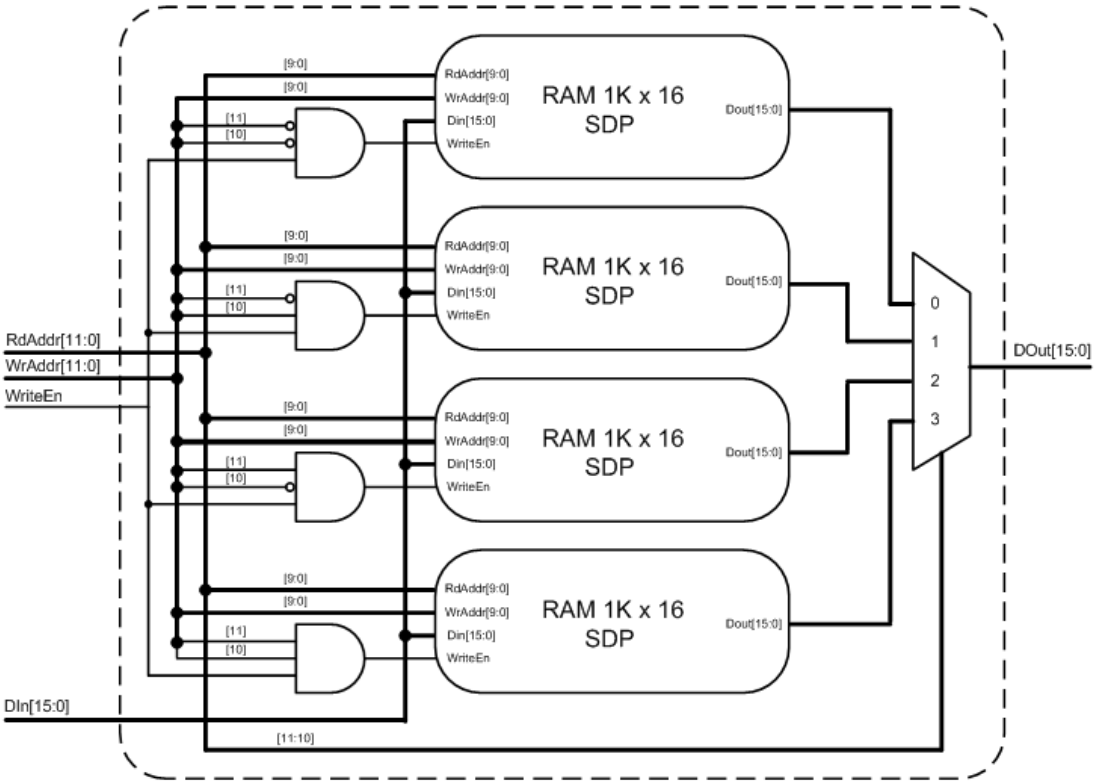


Figure 6 Implementation of a 4k x 16 SDP RAM using 4 instances of 1k x 16 SDP RAMs



giving its functionaility.

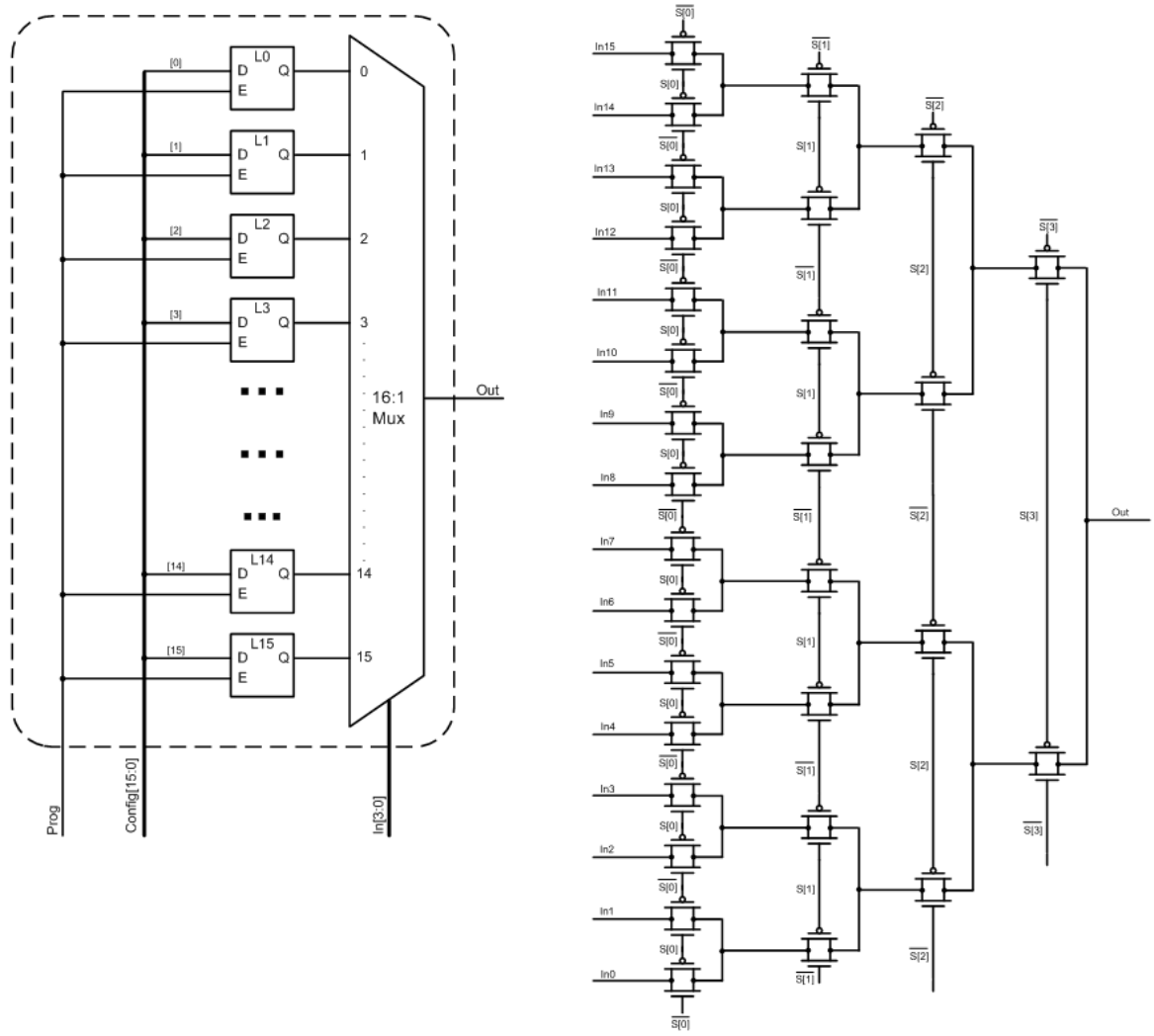
The first assumption we make is that the transistor count is dominated by three things different components - the RAM cells, the number of transistors needed to make the memory address decoder, and the number of transistors needed for the multiplexer. All the transistors from the analog components (sense amps, precharge circuitry, etc.) and configurable components (output registers, read-before-write circuitry, etc.) are considered negligible. Note also that many of the bits are error-check bits. To simplify our analysis even more by maintaining powers of 2's, we will analyze instead a **32k Block RAM**, and just multiply the transistor count of a **32k Block RAM** by $\frac{36k}{32k}$ to get the final transistor count.

We will also assume that the block RAM is organized in more of a square/rectangular structure with both a decoder and a column multiplexer rather than a long linear structure with just a decoder. To figure out how the address bits are split between the decoder and the multiplexer, we note that a 32k BlockRAM can be put in the 1k x 32, 2k x 16, 4k x 8, etc. configurations when kept in true dual-port mode and up to 512 x 64 in simple dual-port mode. This hints at the fact that 6 address bits are used for the column multiplexers, leaving 9 for the decoder. With this in mind, we can begin to crunch some numbers.

Each 1-bit RAM cell in a two-port memory structure takes **8 transitors**. With $32k = 2^{15}$ cells, that is a total of $2^{18} = 262144$ transistors used to implement all the 1-bit RAM cells.

For simplicity, lets assume that the 9-to-512 decoder is implemented using 512 instances of 9-input AND gates. Lets also assume that each 9-input AND gate is made using a 9-input NAND followed by an inverter. A 9-input NAND takes 18 transistors to make (you add 2 transistors for each additional input), and an inverter takes 2. This means 20 transistor per 9-input AND and a total of $512 \cdot 20 = 10240$ per decoder. Since we have two ports, there are **two** of these decoders and a total of 20480 transistors for both decoders.

Figure 7 On the left is the implementation of a 4-LUT. Refer back to HW5 solutions for a transistor-level implementation of a positive level-sensitive latch. On the right is the implementation of a 16:1 multiplexer, which is built by chaining together 2:1 multiplexers.



For the column multiplexers, we have to keep track of all the different memory widths possible. For simplicity, we will assume that no hardware can be shared between each configuration (i.e. a transistor used in one configuration sits idle in another memory width configuration). Recall that, in homework 5, we showed that a 2:1 multiplexer can be implemented using only 6 transistors. To handle the 32k x 1 RAM configuration, we will need 1 64:1 mux. To handle the 16k x 2 RAM configuration, we will need 2 32:1 muxes. For the 8k x 4 RAM, we need 4 16:1 muxes, etc. A 64:1 mux can be made using 63 2:1 multiplexers, a 32:1 mux uses 31 2:1 multiplexers, a 16:1 mux uses 15 2:1 multiplexers, etc. Summing all these up, each port requires 321 2:1 multiplexers, which takes 1926 transistors. With two ports, this makes a total of 3852 transistors.

If we add all these numbers up, we get:

$$262144 + 10240 + 3852 = 276236 \text{ transistors}$$

Adjusting this for the 36k block RAM, we get a grand total of:

$$276326 \cdot \frac{36k}{32k} = 310766 \text{ transistors}$$

Figure 8 Implementation of a 4-LUT with LUTRAM capabilities. This structure is similar to residency table used back Figure 4, albeit with fewer addresses.

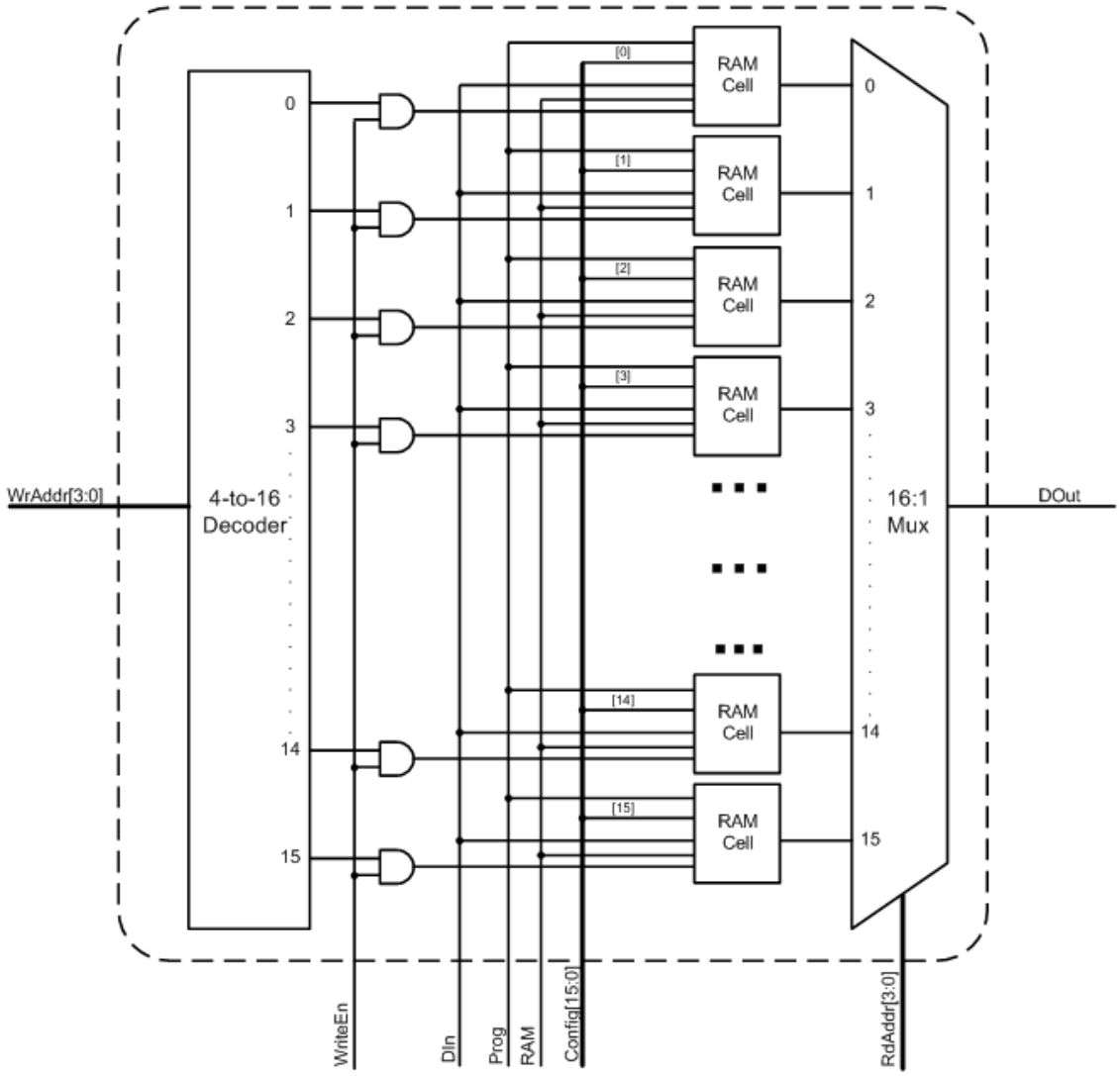


Figure 9 Implementation of a RAM cell.

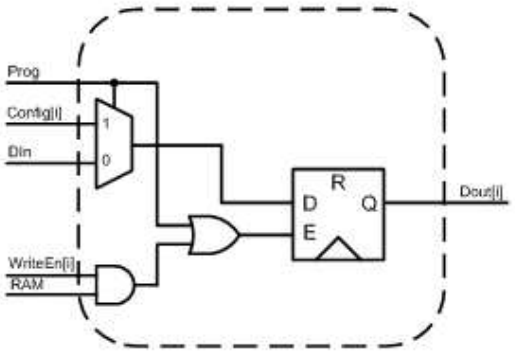
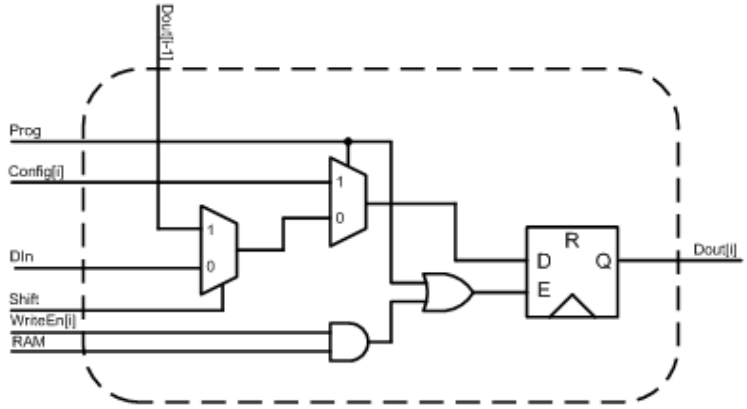


Figure 10 One cell of the 4-LUT with added shift register capabilities.

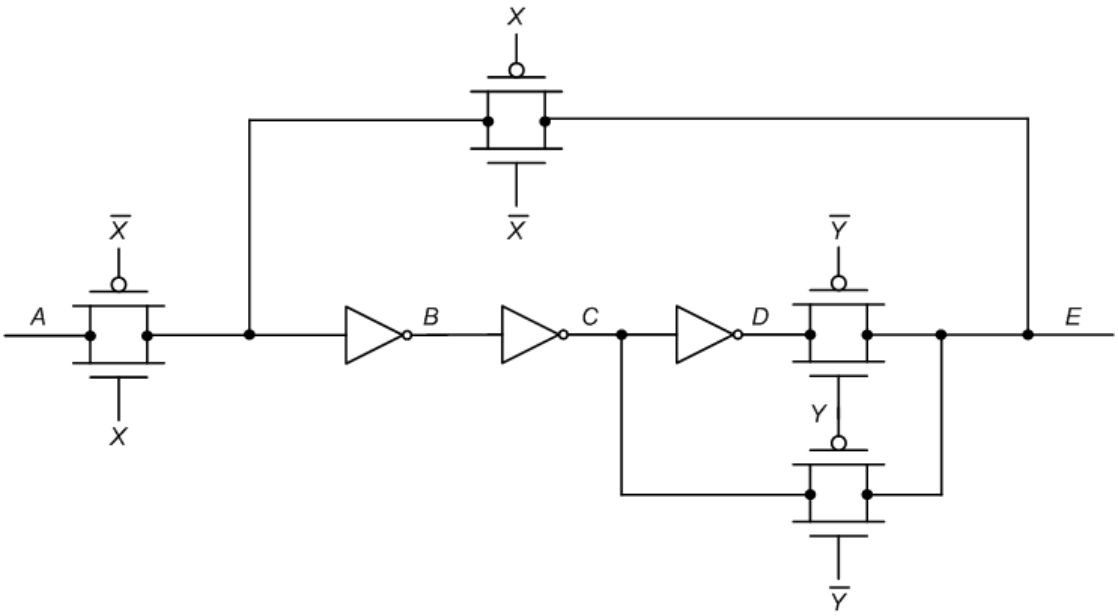


.

One thing to note about this result is that the transistor count total for the 1-bit RAM cells clearly dominate the transistor count for the entire Block RAM.

6. Consider the circuit pictured in Figure 11.

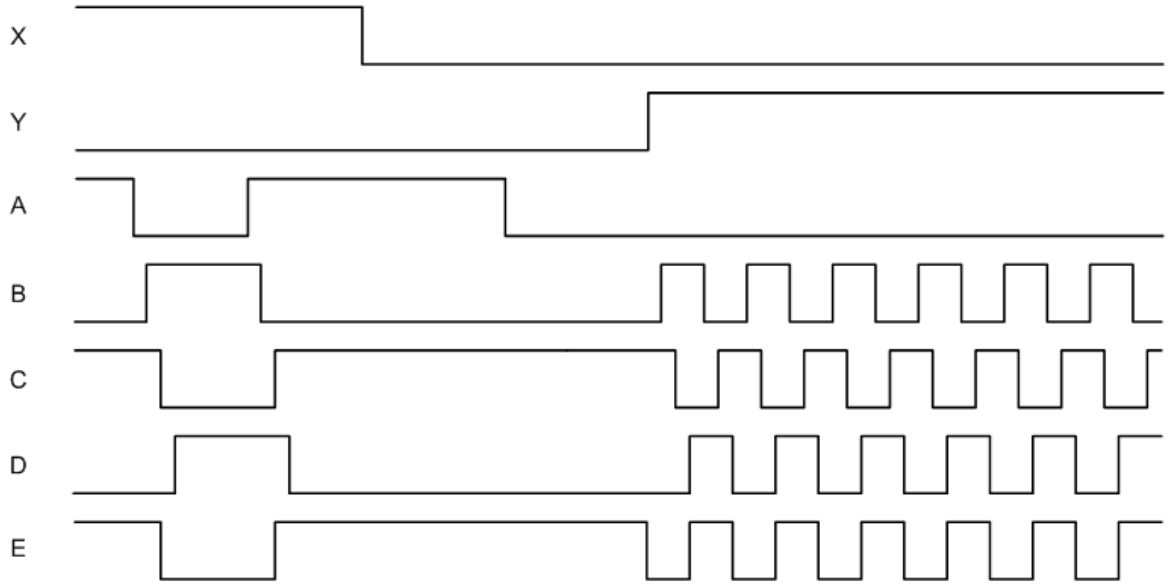
Figure 11 A weird circuit



(a) Complete the timing diagram in Figure 12 for the circuit. Note that it may be helpful for you to think about the inverters as having a little bit of delay. What does this circuit do if `X` is 0 and `Y` is 1? Why?

The circuit forms a combinational loop of 3 inverters when `X` is 0 and `Y` is 1. When we make a combinational loop with an odd number of inverters, we form an unstable circuit that will oscillate at a certain frequency. In fact, a circuit with an odd number of inverters in a loop is called a **ring oscillator**. See Figure 12 for what the waveform will look like.

Figure 12 Waveform for the ring oscillator



(b) How would the waveform for signal `E` change if the circuit had 5 inverters in series instead of 3? Why?

Adding more inverters in series in a ring oscillator simply increases the period (decreases the frequency) of the oscillation, since now it takes a longer time for a signal to propagate all the way through the ring.

7. **Come prepared for design review during your lab section!**