Using Single Port RAM on a Cyclone V

Liam Ramsey

July 6, 2020

This document will go over setting up and using a 1-Port RAM module on a CylcloneV FPGA. Reading the memory is shown with two methods, both of which utilized the JTAG interface with different frontends.

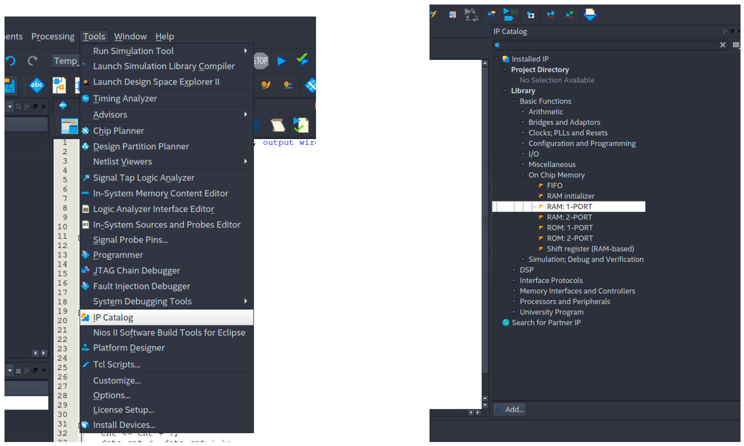
1 Adding the RAM to Your Project

The primary way to add a 1-Port RAM module to your project is to use the IP Megafunction Wizard.

1.1 Using the IP Catalog

If you do not already have the IP Catalog open in Quartus, you can get to it by going to Tools→IP Catalog. From the IP Catalog you can go to Library→Basic Functions→On Chip Memory and double click RAM: 1-PORT. This opens the megafunction wizard for the module.

With the megafunction wizard open, you can specify the size in bits stored at each memory address by setting the width of the output bus "q". Setting how many words of memory specifies how many addresses we have to use, so total memory capacity will be number of words times how many bits per word. Leaving the rest to "Auto" and "Single clock" are fine, unless you know you need otherwise.



(a) IP Catalog (b) RAM: 1-PORT

Figure 1

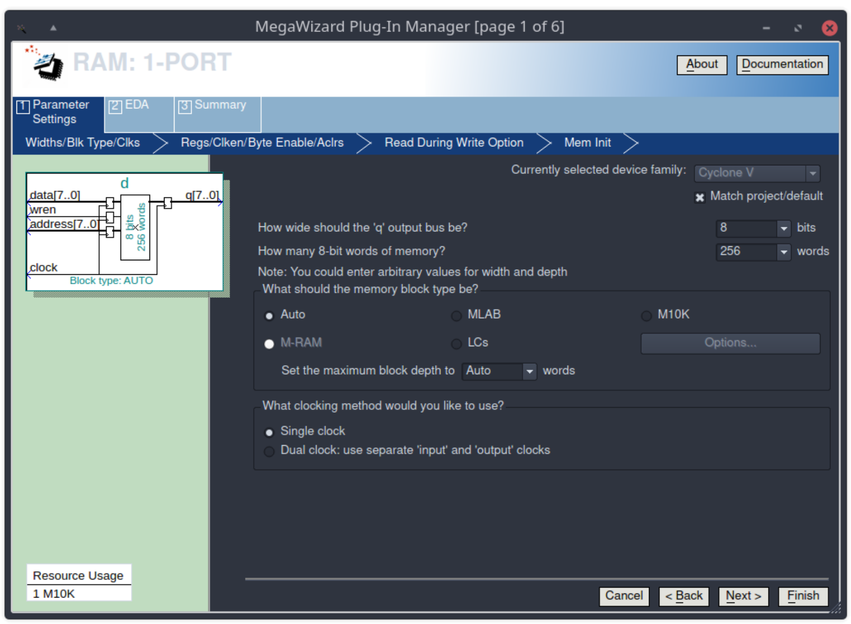


Figure 2: MegaWizard page 1

For pages 2 and 3 options can be left at default for a simple implementation. Things to note are the addition of an all clear signal and a read enable signal.

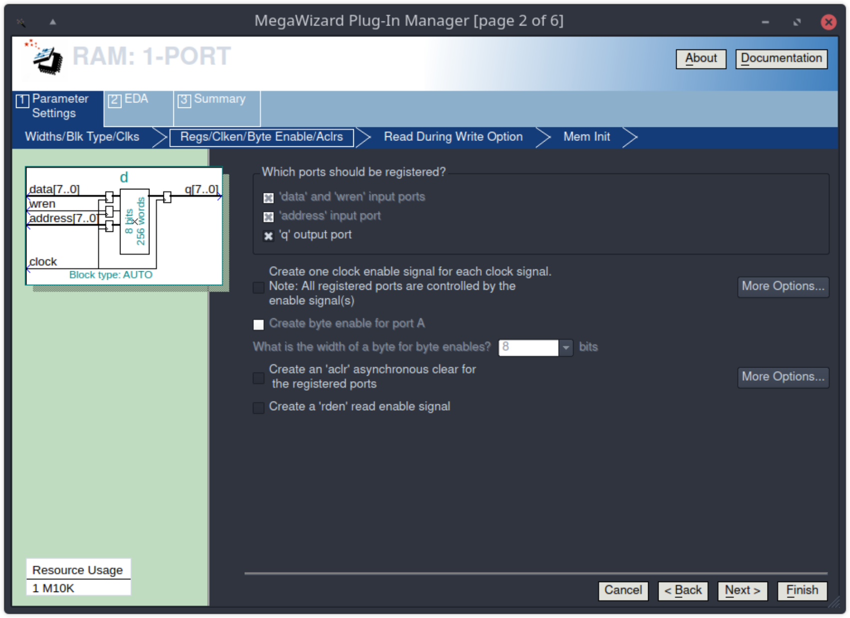


Figure 3: MegaWizard page 2

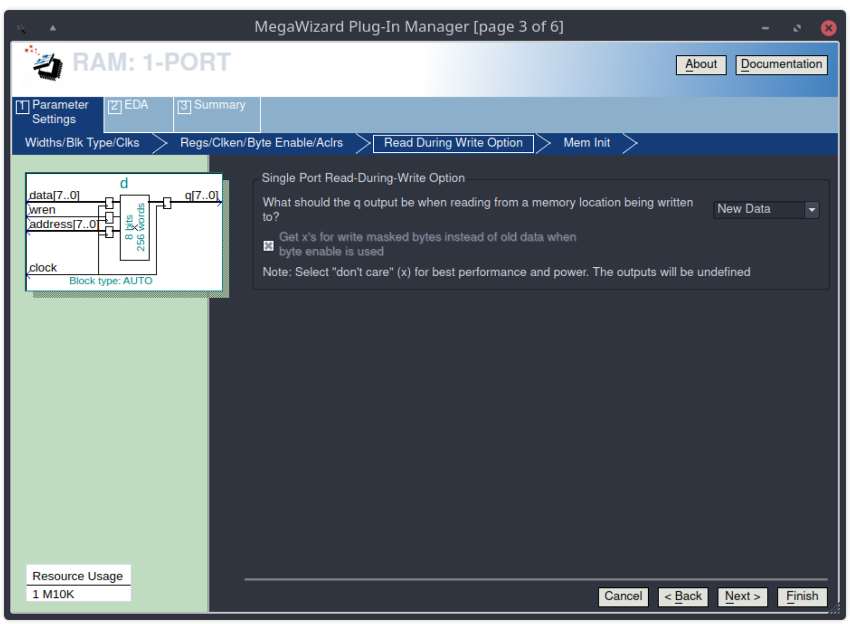


Figure 4: MegaWizard page 3

On page 4 here you'll want to check the box for "Allow In-System Memory Content Editor ..." so that the memory is accessable by external means. Here you can also specify initial contents of the memory. A .mif file can be generated using the mif python package.

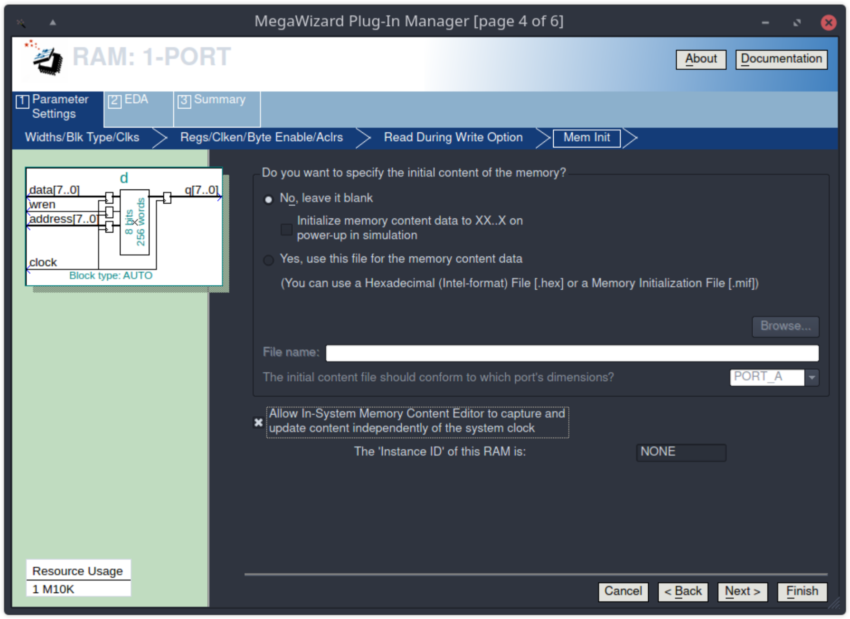


Figure 5: MegaWizard page 4

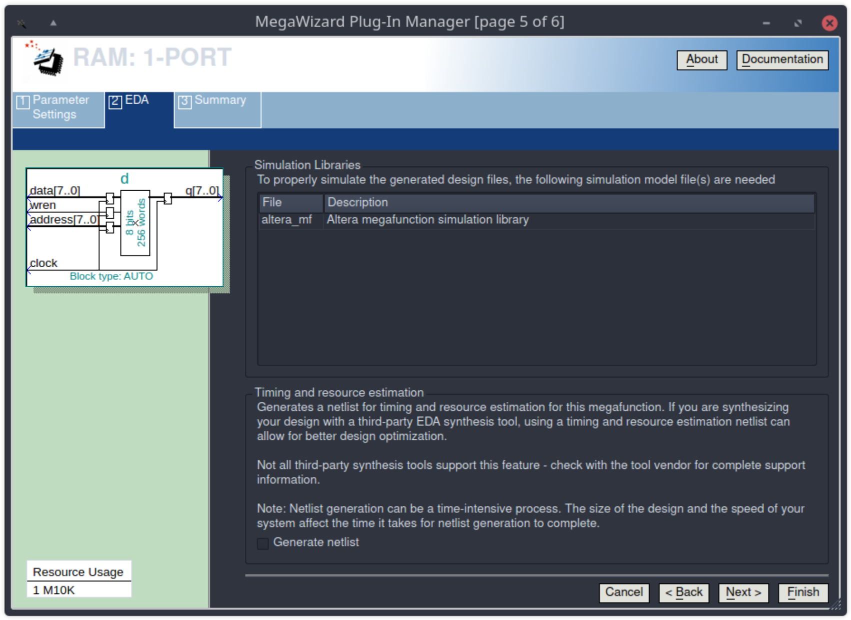


Figure 6: MegaWizard page 5

Here you can define which files Quartus will generate for you. The main <name>.v and <name>_bb.v files are checked by default. Additionally Quartus can generate <name>_inst.v, which contains an example if instantiating the module, you do not need to select this, as I will provide instantiation code here. After clicking finish, Quartus will prompt you to add the IP to your project.

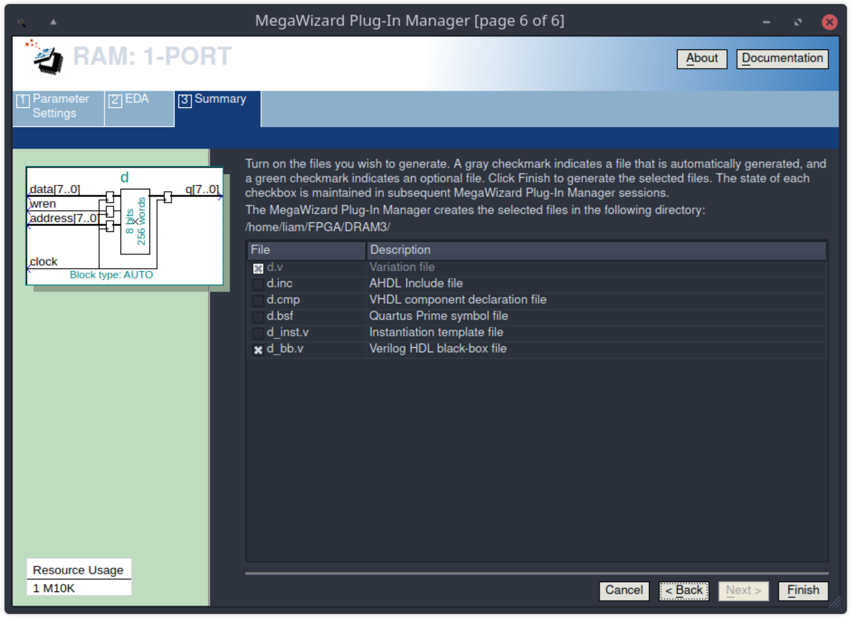


Figure 7: MegaWizard page 6

2 Using the Memory

2.1 Instantiating the module and writing to it

The memory can be instantiated with the following verilog. Something important to note, instantiating multiple times from the same IP will create multiple RAM instances with different indices but the same instance ID name (See the columns in the In-System Memory content editor). This is important to keep in mind when using the python interface as the find_instance function will only find one of them.

```
<name> ram_inst (
    .address ( address_sig ),
    .clock ( clock_sig ),
    .data ( data_sig ),
    .wren ( wren_sig ),
    .q ( q_sig )
);
```

You need to define the inputs and outputs earlier in the code with the names in the parentheses. Also replace <name> with the name that you assigned the IP module. The register sizes must match the values specified when the module was created. For example the declarations for a RAM module with 8-bit words and a 8-bit address space would be:

```
reg address_sig[7:0];
reg data_sig[7:0];
wire wren_sig;
wire clk_sig;
reg q_sig[7:0];

assign clk_sig = clk;
```

Where clk is the input clock to your main module, and <name>is the name given to the RAM module. To write a value to memory, first the wren_sig wire must be set high (you could assign a register to it), then the desired values need to be put into the address_sig and data_sig registers. For an example of a complete verilog file see Appendix B.

2.2 Interfacing With the Memory Using the In-System Memory Content Editor

The simplest way to read and write to the memory on the FPGA is to use Quartus's In-System Memory Content Editor. However, more functionality can be achieved by using the Python packaged described in the next section.

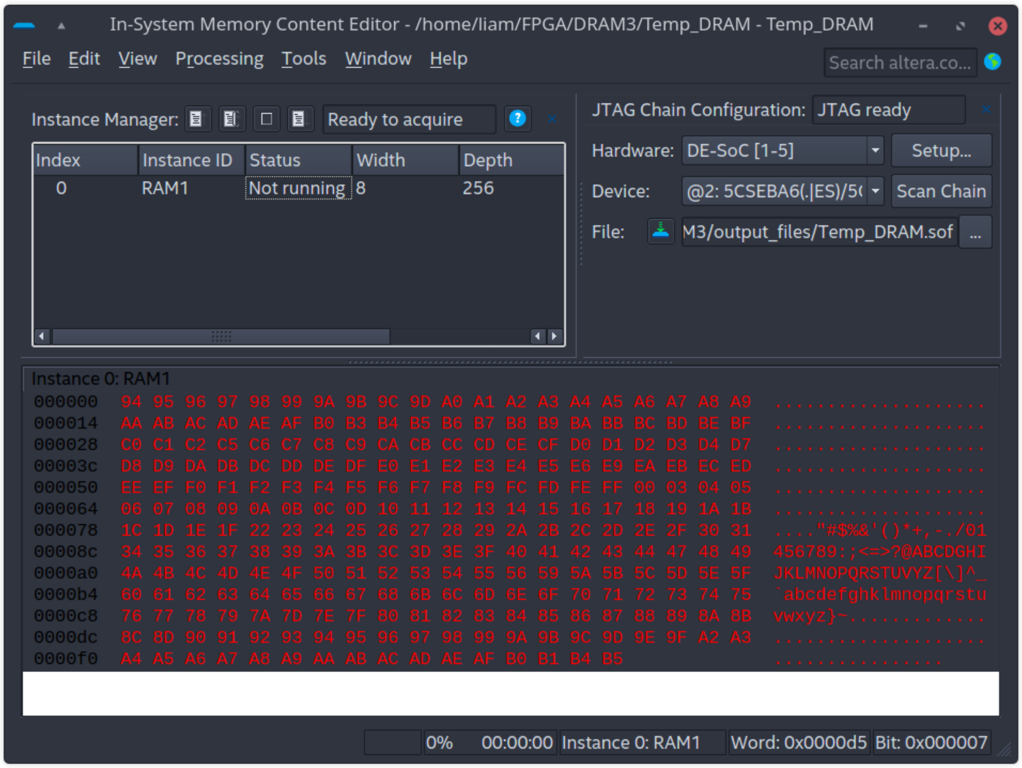


Figure 8: In-System Memory Content Editor

In the upper right section of the window, we have the basic functionality of the device programmer. Clicking the button directly to the right of "File:" will program the FPGA with the specified .sof file. The top left section lists the memory instances Quartus has found on the board. Right clicking an instance provides different options to read and write to the memory. The bottom section contains the memory currently selected in hex values on the left, and ASCII decoded on the right.

2.3 Using the mif and quartustcl Python Packages

In order to interface with the memory using these packages we will be using the QuartusMemory class. In order to use the quartustcl package that QuartusMemory is based on in Windows, you must add the quartus scripts to your environment path variable. This can be done by first locating the quartus install directory, typically at C:\IntelFPGA_lite\<version>\quartus\bin64.

This needs to be added to the Path environment variable. This can be done by windows searching for "Environment Variables" and editing the path variable as shown below.

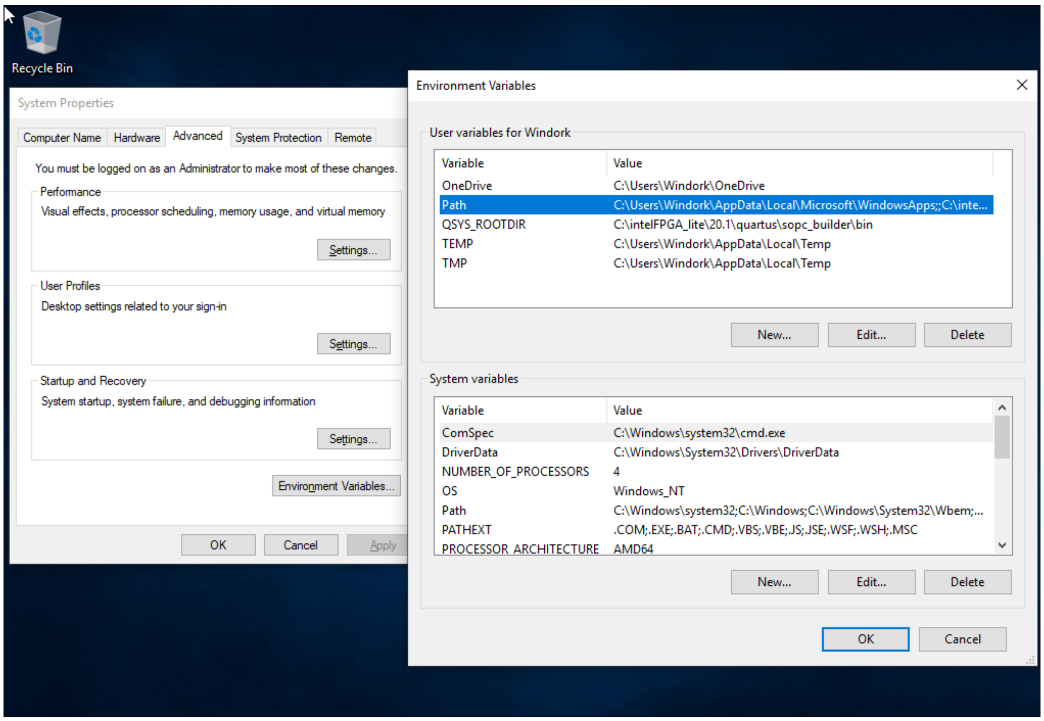


Figure 9: Editing the PATH environment variable

With this done, quartus scripts can now be used via command line and the python package quartustcl will function. Code for the QuartusMemory class is provided in Appendix A. Below is an example program using the class.

```python
from QuartusMemory import QuartusMemory

q = QuartusMemory()

# Find index of instance named RAM1
inst = q.find_instance('RAM1')

```

```
# Read memory from device
arr = q.read_mem(inst,True)

# Print contents of address 0x04
print('Arr1:')
for k in arr[4]:
    print(str(k) + ' ',end='')
print()

# Copy the array and change one of the bits in 0x04
arr2 = arr
arr2[4][1] = 1

# Print the new array
print('Arr2:')
for k in arr2[4]:
    print(str(k) + ' ',end='')
print()

# Write the memory to the device
q.write_mem(inst,arr2,True)

# Read memory into a new array
arr3 = q.read_mem(inst,True)

# Print to confirm that memory was changed
print('Arr3:')
for k in arr3[4]:
    print(str(k) + ' ',end='')
print()
```

The code here simply reads the memory from the device into an array, changes a bit in it, writes it back to the device, then lastly reads to confirm the write executed correctly. The boolean argument at the end of each function call is whether to delete the .mif file generated. Both read and write default to not delete it. It is worth noting that running this script multiple times will overwrite and delete previously generated .mif files. If saving them is desired they should be renamed. The array format is of a two dimensional numpy array consisting of 1's and 0's with type uint8 (unsigned 8-bit integer). This code can be found at https://github.com/Noeloikeau/TDC/tree/Liam in the Python folder.

A QuartusMemory Python Class

Code for the QuartusMemory class. Also available at https://github.com/Noeloikeau/TDC/tree/Liam in the Python folder.

```python
import os
import math
import mif
import quartustcl
import copy
import numpy as np



class QuartusMemory():

    def __init__(self,chip_number=0,fpga_number=1):

        self.quartus = quartustcl.QuartusTcl()
        self.hwnames = self.quartus.parse(self.quartus.get_hardware_names())
        self.hwname = self.hwnames[chip_number] # Picking first chip
        self.devnames = self.quartus.parse(self.quartus.get_device_names(hardware_name=self.hwname))
        self.devname = self.devnames[fpga_number] # Skip SOC chip, which is index 0
        self.path=''
        self.name='mem{0}.mif'



    #Below finds instance index given a name (string)

    def find_instance(self,inst_name,N_levels=2):

```

```
        self.memories_raw = self.quartus.get_editable_mem_instances(hardware_name=self.hwname,\
            device_name=self.devname)
        self.memories = self.quartus.parse(self.memories_raw, levels=N_levels)
        found_memid = None

        for memid, depth, width, rw, type, name in self.memories:
            if name == inst_name:
                found_memid = memid

        if found_memid is None:
            raise RuntimeError('Could not find memory '+inst_name)
        return int(found_memid)



    #Below reads memory from instance and returns as an array
    #Generates intermediary MIF file which is then optionally deleted

    def read_mem(self,inst,delete_mif=False):

        fname=self.path+'r'+self.name.format(inst)

        self.quartus.begin_memory_edit(hardware_name=self.hwname,\

            device_name=self.devname)

        self.quartus.save_content_from_memory_to_file(

            instance_index=inst,
```

```python

            mem_file_path=fname,

            mem_file_type='mif'

        )

        with open(fname, 'r') as f:
            data = mif.load(f)
            f.close()
        self.quartus.end_memory_edit()

        if delete_mif:
            os.remove(fname)

        return data


    # Below writes memory to an instance from an array by writing data to mif file, then to instance
    # Optionally will not delete temporary .mif

    def write_mem(self,inst,data,delete_mif=False):

        fname=self.path+'w'+self.name.format(inst)

        self.quartus.begin_memory_edit(hardware_name=self.hwname, device_name=self.devname)

        try:
            with open(fname, 'w') as f:
                mif.dump(data, f)
                f.close()
            self.quartus.update_content_to_memory_from_file(
                instance_index=inst,
```

```
                mem_file_path=fname,
                mem_file_type='mif',
            )
            self.quartus.end_memory_edit()
        except:
            self.quartus.end_memory_edit()
        if delete_mif:
            os.remove(fname)
```

B Verilog Example

An example of a 1-Port RAM module being filled with the output of a counter.

```
module Temp_DRAM (input wire clk, output wire [7:0] LED);

reg [31:0] cnt;
reg [7:0] address_sig;
reg [7:0] data_sig;
wire clock_sig;
wire wren_sig;
reg wren_reg;
reg [23:0] data_cnt;

ram ram_inst (
        .address ( address_sig ),
        .clock ( clock_sig ),
        .data ( data_sig ),
        .wren ( wren_sig ),
        .q ( q_sig )
        );

initial begin
        cnt <= 32'b0;
        address_sig <= 8'b0;
        wren_reg <= 1;
        data_sig <= 8'b0;
```

```
        data_cnt <= 24'b111111;
end

assign wren_sig = wren_reg;
assign LED[0] = data_cnt[23];

always @(posedge clk) begin
        cnt <= cnt + 1;
        data_cnt <= data_cnt + 1;

        data_sig <= data_sig + 1;
        address_sig <= address_sig + 1;

        if (data_sig == 8'b0) begin
                data_sig <= data_sig + 8'b11;
        end


end


endmodule
```