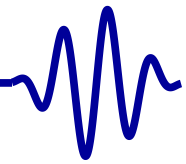
8. Using Block RAM

Gisselquist
Technology, LLC

Daniel E. Gisselquist, Ph.D.

Lesson Overview



Three types of FPGA memory

- Flip-flops
- Distributed RAM
- Block RAM

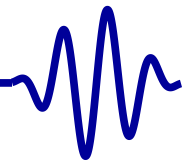
Block RAM is special within an FPGA

- It is fast and abundant
- Requires one clock to access
- Can only be initialized at startup
- Yet there are some logic requirements to use it

Objectives

- Be able to create block RAM resources
- Understand the requirements of block RAMs
- Learn how to verify a component containing a block RAM

Lesson Overview



Let's also take a quick look at synchronous resets

- Learn the two types of resets
- Reset logic follows one of two forms

Extra Objectives

- Know the two forms of synchronous reset logic
- Know how to verify a design with a synchronous reset

Design Goal

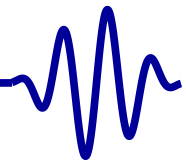
Let's rebuild our hello world design, but make the message longer

- We'll use a memory to capture our longer message

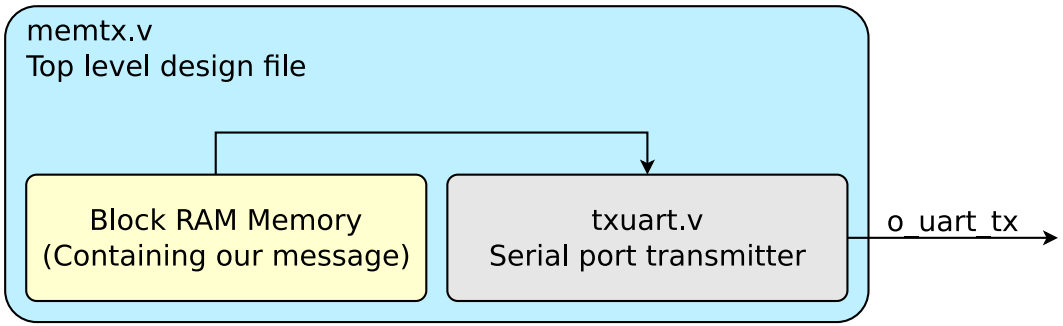
```
% ./memtx_tb
================================================================================
|                                                                              |
|  Psalm 1                                                                     |
|                                                                              |
|  Blessed is the man that walketh not in the counsel of the ungodly, nor      |
|    standeth in the way of sinners, nor sitteth in the seat of the scornful.  |
|  But his delight is in the law of the LORD; and in his law doth he meditate  |
|    day and night.                                                            |
|  And he shall be like a tree planted by the rivers of water, that bringeth   |
|    forth his fruit in his season; his leaf also shall not wither; and        |
|    whatsoever he doeth shall prosper.                                        |
|  The ungodly are not so: but are like the chaff which the wind driveth       |
|    away.                                                                     |
|  Therefore the ungodly shall not stand in the judgment, nor sinners in the   |
|    congregation of the righteous.                                            |
|  For the LORD knoweth the way of the righteous: but the way of the ungodly   |
|   shall perish.                                                              |
|                                                                              |
================================================================================


Simulation complete
%
```

- Then read from this memory, and . . .
- Transmit it out the serial port

Goal Design



Here's a basic block diagram



- We'll re-use the serial port transmitter, `txuart.v`
- We'll capture our message in a block RAM, and ...
- We'll use a top level module to coordinate it all, `memtx.v`
  - We'll infer the block RAM within our `memtx.v` design

But what is on-chip RAM and how shall we declare and use it?

On-Chip Memory

There's a special type of declaration for memory in Verilog:

```
reg      [W-1:0] ram      [0:MEMLN-1];
```

- This defines a memory of `MEMLN` elements,

  where each element is `W` bits long
- Verilog allows `MEMLN` to be anything
- Practically, `MEMLN` must only ever be a power of two, 2^N, in order to avoid simulation/hardware mismatch
- I tend to define my memories as

```
reg      [W-1:0] ram      [0:(1<<LGMEMSZ)-1];
```

  - This forces the power of two requirement
  - `LGMEMSZ` can also be used as the width of the address

Declaring On-Chip Memory

There's a special type of declaration for memory in Verilog:

```
reg      [W-1:0] ram      [0:(1<<LGMEMSZ)-1];
```

The synthesis tool will decide how to implement this

- Flip-Flops
  - Useful for small numbers of bits
  - Very inefficient for implementing memory on an FPGA
- Distributed RAM
  - Useful for small, localized RAM needs
  - Typically allocated one-bit at a time for memory sizes of 2^6 elements (Ex. Xilinx's SLICEM)
- Block RAM
  - Useful for larger and wider RAM needs
  - Using block RAM requires that you follow special rules

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block

```
always @(posedge i_clk)
if (write)
        ram[write_addr] <= write_value;

always @(posedge i_clk)
if (read)
        read_value <= ram[read_addr];
```

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once

```
always @(posedge i_clk)
if (i_reset)
begin
        // This is illegal!  Block
        // RAM cannot be re-initialized
        for(i=0; i<ramsize; i=i+1)
                ram[i] <= 0;
end else if (i_stb)
        ram[addr] <= value;
```

This is often an unexpected frustration for beginners.

- The solution is to rewrite your algorithm so you don't need to do this

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if

```
always @(posedge i_clk)
if (A)
        value <= // something;
else if (B)
        value <= // something else;
else if (C)
        // Don't do this either!
        value <= ram[addr];
else if (D)
        // logic continues ...
```

Such logic often ends up being replaced by flip-flops

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list

```
// Don't do this
output reg [W-1:0] ram [0:(1<<LGMEMSZ)-1];
```

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

```
// Many synthesizers will turn this into FF's
always @(posedge i_clk)
if (write_enable)
begin
        B <= // some logic;
        C <= // something else;
        ram[addr] <= value;
end
```

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow byte enables

```
always @(posedge i_clk)
if (write_enable)
begin
        if (en[1])
                ram[addr][15:8] <= value[15:8];
        if (en[0])
                ram[addr][ 7:0] <= value[7:0];
end
```

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow write-through

- Where the value being written may be read on the same clock

```
always @(posedge i_clk)
begin
        if (write_enable)
                mem[addr] = wvalue;
        rvalue = mem[addr];
end // Note the non-blocking notation!
```

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

Some synthesizers/hardware allow write-through

- Where the value being written may be read on the same clock
  - This would be ideal for a CPU register file
- It's not uniformly supported across our chosen tools/vendors
- Know your hardware, synthesizer, and simulator
- We'll pretend this feature does not exist in this tutorial

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

If you fail to follow these rules,

> You might get something other than block RAM, or
> You're design might fail to synthesize entirely

This is a common reason for synthesis failure

Block RAM Rules

If you want a block RAM, you need to follow certain rules:

1. Any RAM access should be contained in its own always block
2. RAM can only be initialized once
3. Don't put a RAM access in a cascaded if
4. Don't put a RAM in a port list
5. Don't put a RAM in a block with other things

If you fail to follow these rules,

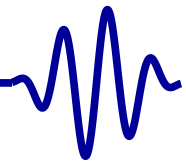
> You might get something other than block RAM, or
> You're design might fail to synthesize entirely

This is a common reason for synthesis failure

- Always keep an eye on your RAM and LUT usages
- Something out of bounds may be caused by this

If you suspect this is a problem, break your design into smaller and smaller components until you find out what's going on

Distributed RAM Rules



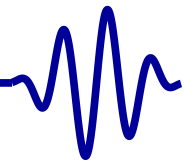
When is distributed RAM used?

- If the memory size is small (32 elements or less)
- If the memory is read without a clock

```
always @(*)
        rvalue = mem[addr];
// Or equivalently
assign  rvalue = mem[addr];
```

- Obviously, only if the device has distributed RAM
  - iCE40 devices have no distributed RAM

Initializing Memory



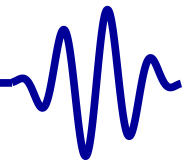
How might we initialize our RAM?

- We could use assignments within an initial block

```
reg [31:0]      ram     [0:8191];

integer k;
initial begin
        for(k=0; k <8192; k=k+1)
                ram[k] = 0;
        // We can also set specific values
        ram[5] = 7;
        ram[8190] = 5;
        // etc.
end
```

- When using Xilinx's ISE, this is the only way I've managed to initialize RAM

Initializing Memory



How might we initialize our RAM?

- We could use assignments within an initial block
  - Verilator (currently) complains about non-blocking **initial** assignments

```
// This will generate a Verilator warning
initial ram[8190] <= 5;
```

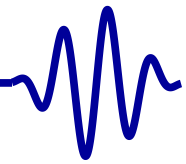
  - Yosys (currently) complains about blocking **initial** assignments

```
// This will generate a Yosys warning
initial ram[8190] = 5;
```

If you don't redefine any values, both will still work

- In this case, you may ignore the warnings

Initializing Memory



How might we initialize our RAM?

- We could use a `$readmemh` function call (recommended)

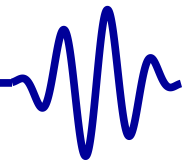
```
reg [31:0]      ram     [0:8191];
initial $readmemh(FILE_NAME, ram);
```

- Each line of the file `FILE_NAME` has format `%0*x`

```
012345678
....
```

- Separate each RAM word by white space
- Number of digits is based upon the width of the RAM word
  - Our example shows a 32-bit word
- Xilinx's ISE has a known bug that prevents `$readmemh` from working. Vivado doesn't have this bug.

Initializing Memory



How might we initialize our RAM?

- We could use a `$readmemh` function call (recommended)

```
reg [31:0]      ram     [0:8191];
initial $readmemh(FILE_NAME, ram);
```

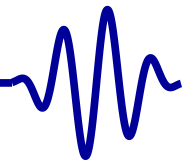
- Alternatively, lines can begin with (hexadecimal) addresses

```
@000000e0 2c 20 61 20 6e 65 77 20 6e 61 74 ...
@000000f0 63 6f 6e 63 65 69 76 65 64 20 69 ...
....
```

  - This example shows a series of 8-bit characters
  - Sixteen per line

- This form makes it possible to skip elements
- We'll build one of these files for our project later

Initializing Memory



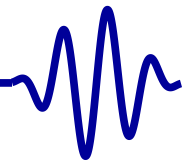
How might we initialize our RAM?

- We could use a `$readmemh` function call (recommended)

```
reg [31:0]      ram     [0:8191];
initial $readmemh(FILE_NAME, ram);
```

- On-chip RAM can only be initialized in an **initial** block
- Cannot re-initialize a block RAM in this fashion later without reconfiguring (i.e. reloading) the FPGA

Generating the Hex file

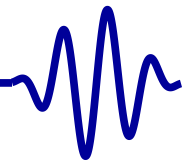


Let's generate a hex file that we can use with `$readmemh`

- Use a C++ program
- We'll call this program `genhex`
- Much of the program is boilerplate and error checking
- We'll skip much of this boilerplate now, and instead discuss only the interesting parts

You can find the entire `genhex` program with the course materials

Generating the Hex file

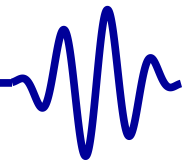


Let's build our hex file

- We'll prefix each line with an address

```
int      linelen = 0;
int      ch, addr = 0;

fprintf(fout, "@%08x ", addr);
linelen = 10;
```

Generating the Hex file

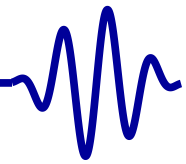


Let's build our hex file

- We'll prefix each line with an address
- Process one character at a time

```
// Read one character from our file
while((ch = fgetc(fp))!=EOF) {
    // and process it if we read
    // a non-empty character
    // ...
}
```

Generating the Hex file



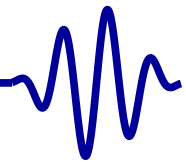
Let's build our hex file

- We'll prefix each line with an address
- Process one character at a time
- The values out are simply hex characters

```
// ...
while((ch = fgetc(fp))!=EOF) {
        fprintf(fout, "%0*x ",
            (nbits+3)/4,
            ch & 0x0ff);
        linelen += 3;
        addr++;
        // ...
```

- We can use **nbits** to make the width generic
- In this example, we only need two hex digits each, so **nbits** = 8.

Generating the Hex file



Let's build our hex file

- We'll prefix each line with an address
- Process one character at a time
- The values are just simply hex characters
- After 56 bytes, start a new line with a new address

```
while((ch = fgetc(fp))!=EOF) {
    // ...
    if (linelen >= 56) {
        // New line starting with
        // the current address
        fprintf(fout, "\n@%08x ", addr);
        linelen += 10;
    }
} fprintf(fout, "\n");
```

Generating the Hex file

One task remains: adding the hexfile generation to our Makefile

- Our target is "memfile.hex"
- It depends upon **genhex**, and our text file, **psalm.txt**

```
memfile.hex: genhex psalm.txt
        ./genhex psalm.txt
```

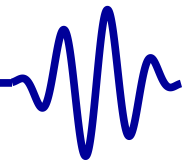
- **genhex** depends upon **genhex.cpp**, and must also be built

```
genhex: genhex.cpp
        g++ genhex.cpp -o genhex
```

- Don't forget to make sure **memfile.hex** is built before it's needed

Voila! A hex file that will change anytime **psalm.txt** does

Using the hexfile

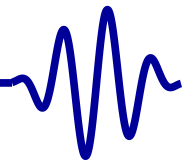


After all that work,

- We can now declare and initialize our memory

```
reg      [7:0]    tx_memory          [0:2047];

initial  $readmemh("memfile.hex", tx_memory);
```

Reset



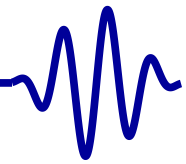
There are two types of resets

- Asynchronous resets

```
initial tx_index = 0;
always @(posedge i_clk or negedge i_areset)
if (i_areset)
        tx_index <= 0;
else begin
        // The rest of your logic
end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset

Reset



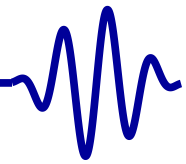
There are two types of resets

- Asynchronous resets

```
initial tx_index = 0;
always @(posedge i_clk or negedge i_areset)
if (i_areset)
        tx_index <= 0;
else begin
        // The rest of your logic
end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset
  - This is bad.

Reset



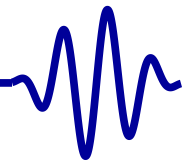
There are two types of resets

- Asynchronous resets

```
initial tx_index = 0;
always @(posedge i_clk or negedge i_areset)
if (i_areset)
        tx_index <= 0;
else begin
        // The rest of your logic
end
```

- These are more complex than their synchronous counterparts
- Often require being asserted for many cycles, and
- Released on a clock edge
- Poor design can lead to radio interference triggering an internal asynchronous reset
  - This is bad. We will avoid these in this tutorial

Reset



There are two types of resets

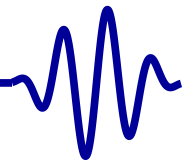
- Asynchronous resets, and
- Synchronous resets
  - These are set and released on clock tick

```
initial tx_index = 0;
always @(posedge i_clk)
if (i_reset)
        tx_index <= 0;
else begin
        // The rest of your logic
end
```

- These are simple to build and use

Let's implement a synchronous reset to this design

Synchronous Reset



Many designs use a synchronous reset

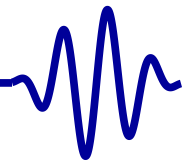
- Values responsive to a reset should also have an **initial** value
- The **initial** value and the reset value must match

```
initial tx_index = 0;
always @(posedge i_clk)
if (i_reset)
        tx_index <= 0;
else begin
        // The rest of your logic
end
```

- I like this form of a reset, but
- It requires that every register set by this block gets reset as well

The original Hello World design included no reset

Synchronous Reset



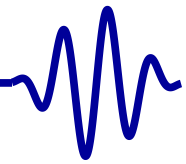
Many designs use a synchronous reset

- Values responsive to a reset should also have an **initial** value
- An alternate form of reset needs to be used if some values need to be reset within the block and others don't

```
initial tx_index = 0;
always @(posedge i_clk)
begin
        // Your logic would come
        // first, then ...

        if (i_reset)
                // Overrides the logic above
                tx_index <= 0;
end
```

- This is a more generic form, useful for all purposes

Synchronous Reset



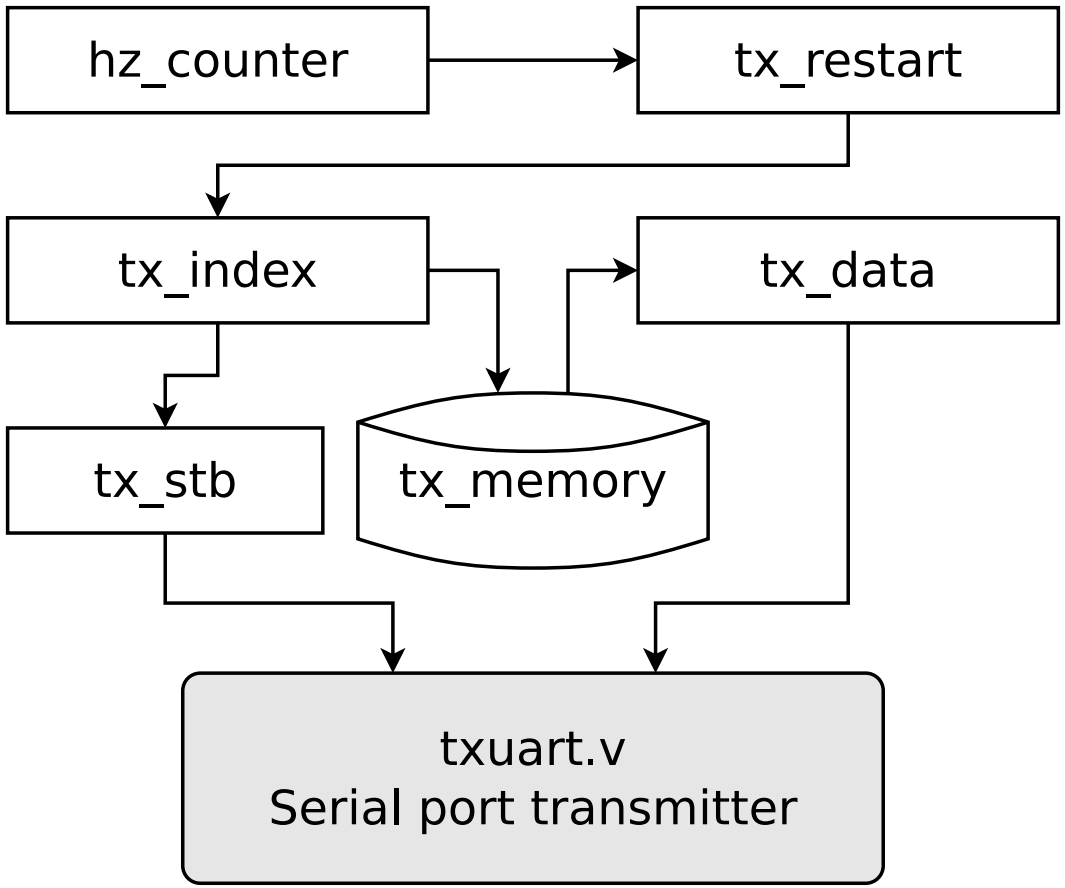
Why might you need a synchronous reset?

- Sometimes it just helps to start over
- Not all technologies support **initial** values
  - For example, if you want to create FPGA+ASIC support, you design will need a reset
- A (debounced) button can be used to create a reset
- Sometimes internal or external conditions will require a reset
  - Ex: An embedded CPU crash, or watchdog timer timeout might cause a CPU to need to be reset

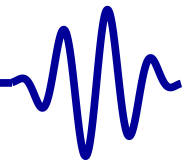
Let's use a synchronous reset in our design

Overview

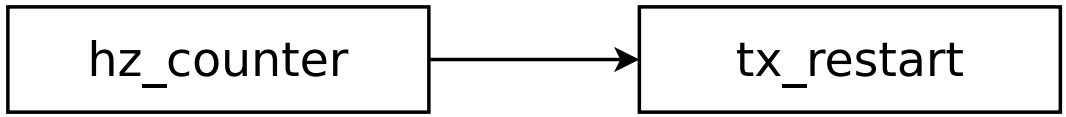
Let's examine a design overview



Overview



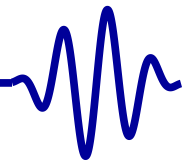
Here's how our design is going to work



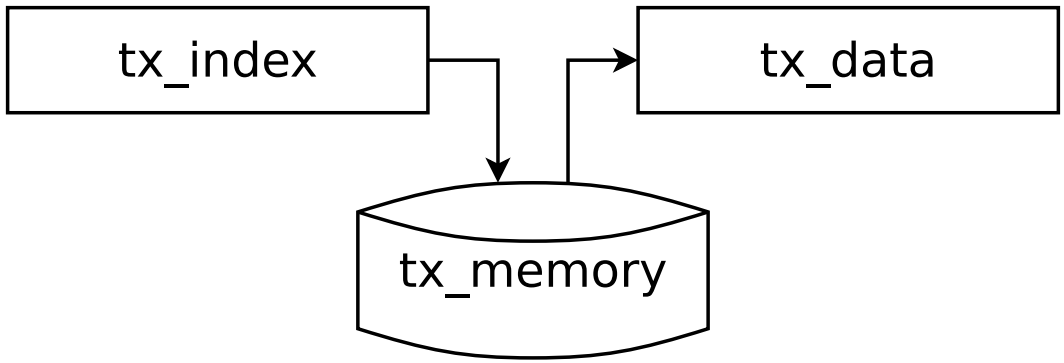
- We'll send our message once per second
- A counter, `hz_counter`, will count each second
- When `hz_counter` is zero, `tx_restart` will signal the rest of the design to start

This much should be fairly familiar

Overview



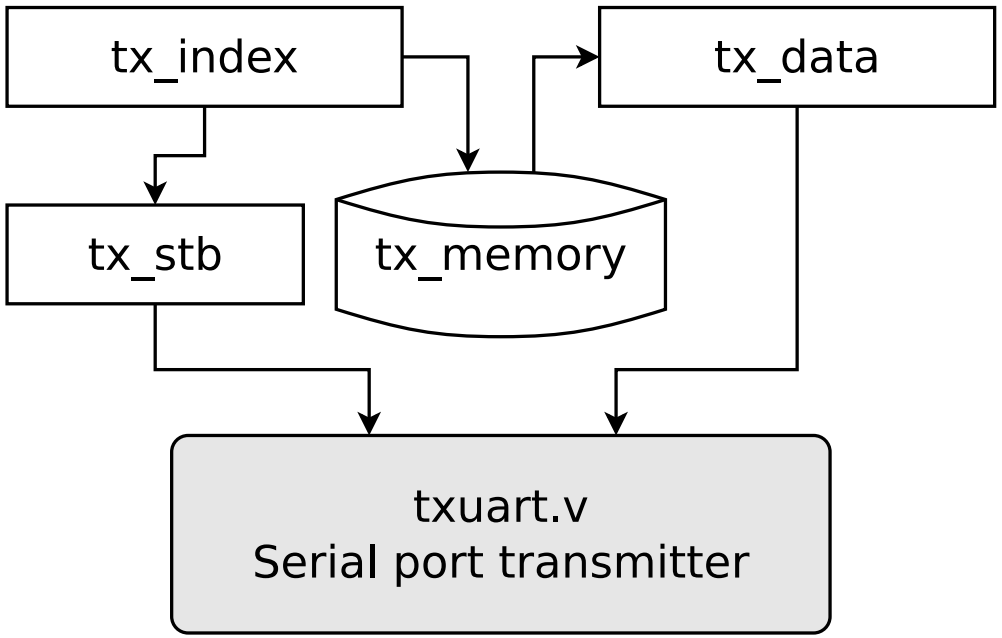
Here's how our design is going to work



- `tx_index` will capture our position in the message stream
- We'll read `tx_data` from memory, to know what to transmit

Overview

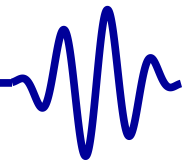
Here's how our design is going to work



- `tx_stb` will request a byte to be transmitted
- Once the whole message has been transmitted,
- `tx_stb` will deactivate until the next `tx_restart`

Are you ready to examine some Verilog?

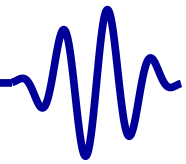
Restarting



Here's the one-second counter, `hz_counter`

```
// We'll start our counter just before the
// top of the second, to give everything
// a chance to initialize
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (i_reset)
        hz_counter <= 28'h16;
else if (hz_counter == 0)
        hz_counter <= CLOCK_RATE_HZ - 1'b1;
else
        hz_counter <= hz_counter - 1'b1;
```

Restarting

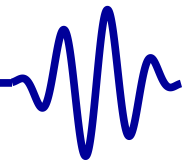


Here's the one-second counter, `hz_counter`

```
// We'll start our counter just before the
// top of the second, to give everything
// a chance to initialize
initial hz_counter = 28'h16;
always @(posedge i_clk)
if (i_reset)
        hz_counter <= 28'h16;
else if (hz_counter == 0)
        hz_counter <= CLOCK_RATE_HZ - 1'b1;
else
        hz_counter <= hz_counter - 1'b1;
```

- Question: What assertion does this logic require?

Restarting



Once a second, we'll set `tx_restart`

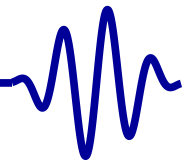
```
initial tx_restart = 0;
always @(posedge i_clk)
        tx_restart <= (hz_counter == 1);
```

Do you see a formal property hiding in here?

```
always @(*)
        assert(tx_restart == (hz_counter == 0));
```

Practice writing assertions as you see relationships!

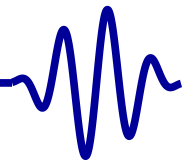
Mem Address



We'll need an address to read from memory

```
// Number of bytes in our message
parameter       MSGLEN = 1600;

initial tx_index = 0;
always @(posedge i_clk)
if (i_reset)
        tx_index <= 0;
else if ((tx_stb)&&(!tx_busy))
begin // Advance anytime a character was
        // accepted by the serial port,
        if (tx_index == MSGLEN-1)
                // End of message
                tx_index <= 0;
        else
                tx_index <= tx_index + 1'b1;
end
```

Reading from Memory



Memory reads take one clock

```
always @(posedge i_clk)
        tx_data <= tx_memory[tx_index];
```

Remember our rules from earlier?

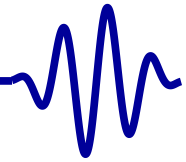
- We might have also chosen to use a read enable
- It wasn't necessary for this design though

When to transmit

As with hello world, `tx_stb` indicates we have a character to transmit

```
initial tx_stb = 1'b0;
always @(posedge i_clk)
if (i_reset)
        tx_stb <= 1'b0;
else if (tx_restart)
        // Start transmitting anytime
        // tx_restart is true
        tx_stb <= 1'b1;
else if ((tx_stb)&&(!tx_busy)
            &&(tx_index >= MSGLEN-1))
        // Stop when we get to the end
        // of the message
        tx_stb <= 1'b0;
```

Serial Port

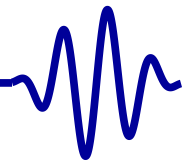


We'll skip the serial port details here

- We built this earlier
- We also showed how to abstract the serial port earlier
- Even our simulation script is nearly identical to Hello World

Feel free to go back and review if you don't remember these

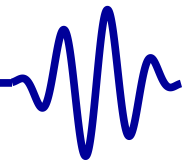
Next Steps



That's the basics of our design!

- We've already built our hex file, so
- We can now move on to formal verification!

Formal Verification



Formally verifying a component using memory requires:

- Assuming a constant address
- Asserting properties for the value at that address
- Usually requires examining no more than a single address

We can assume a constant value using the (* anyconst *) attribute

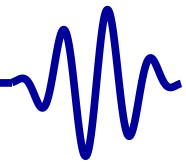
```
(* anyconst *) reg [10:0] f_const_addr;
```

- This allows the solver to pick any value for `f_const_addr`
- As long as it is constant
- If even one value can make your design fail,

    the solver will find it

Let's see how this works

Formal Verification



Let's create a value to match our memory at

```
(* anyconst *) reg [10:0] f_const_addr;
```

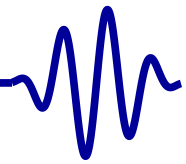
- We'll call this `f_const_value`

```
reg [7:0] f_const_value;
always @(posedge i_clk)
if (!f_past_valid)
    f_const_value <= tx_memory[f_const_addr];
else
    assert(f_const_value
        == tx_memory[f_const_addr]);
```

This value is constant because we are implementing a ROM

Now we can **assert** any properties associated with this address

Formal Verification



This design need only **assert** one memory property

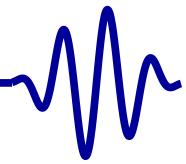
```
(* anyconst *) reg [10:0] f_const_addr;
                reg [ 7:0] f_const_value;
```

- When we transmit a value from `f_const_addr`, ...
- **assert** that it is the right value

```
always @(posedge i_clk)
if ((tx_stb)&&(!tx_busy)
                &&(tx_index == f_const_addr))
        assert(tx_data == f_const_value);
```

We'll come back to this memory verification approach again when we discuss FIFO's

Formal Verification



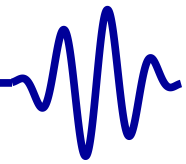
What other properties might we **assert**?

- That our index remains within bounds?
- That any time our index is within the memory bounds, `tx_stb` is high?

You should be familiar with these

- Let's pause to look at the reset
- Cover might need some attention as well

Reset Assertions

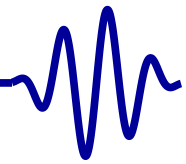


Synchronous reset properties have a basic pattern

- You may (or may not) assume an initial reset

```
always @(*)
if (!f_past_valid)
        assume(i_reset);
```

Reset Assertions



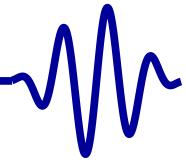
Synchronous reset properties have a basic pattern

- You may (or may not) assume an initial reset
- The **initial** value, held when `!f_past_valid`, and
  The value following a reset, i.e. when **$past**(`i_reset`)
  Should both be identical

```
// Check for anything with an initial
// or a reset value here
always @(posedge i_clk)
if ((!f_past_valid)||($past(i_reset)))
begin
        assert(hz_counter == 28'h16);
        assert(tx_index == 0);
        assert(tx_stb == 0);
end
```

- This verifies we met the rules of a synchronous reset

Cover



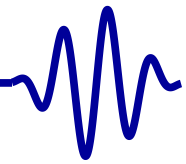
Unlike our Hello World design

- We can't cover the entire message
  - It's just too long
- We can only cover the first several steps
- Let's cover the first 30 characters

```
always @(posedge i_clk)
cover(tx_index == 30);
```

We'll need to simulate the rest

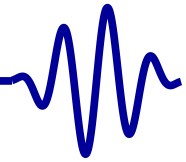
Simulation



Our simulation script is nearly identical to Hello World

```
// ...
#include "Vmemtx.h"
// ...
int     main(int argc, char **argv) {
        // ...
        TESTB<Vmemtx>   *tb = new TESTB<Vmemtx>;
        //
        tb->opentrace("memtx.vcd");
        for(unsigned clocks=0;
                clocks < 16*2000*baudclocks;
                 clocks++) {

                tb->tick();
                (*uart)(tb->m_core->o_uart_tx);
        } // ...
}
```

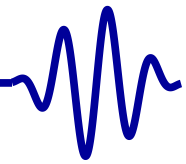
Exercise!



As with all of your designs, let's:

- Formally Verify this design
- Make sure it works in simulation

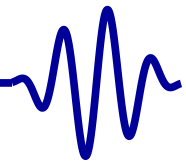
Bonus



If you have hardware to work with,

- Build this design for your hardware!
  - Be sure to compare the resource usage to Hello World
- Examine the serial port output
  - Does your terminal require carriage returns?
- How hard would it be to change the message?
  - Pick another message to send
  - I'm quite partial to the Sermon on the Mount, found in Matthew 5-7
  - What changes would need to be made to your design to support a longer message?
  - What's the longest message your hardware will support?
    - Would Psalm 119 fit?

Conclusion

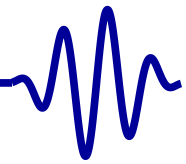


What did we learn this lesson?

- The Rules of using Block RAM
- How to generate a hex file for initializing memory
- Two forms of synchronous reset logic
- How to formally verify . . .
  - A component that uses RAM
  - A synchronous reset

Now we just need to build a serial port receiver

Conclusion



What did we learn this lesson?

- The Rules of using Block RAM
- How to generate a hex file for initializing memory
- Two forms of synchronous reset logic
- How to formally verify . . .
  - A component that uses RAM
  - A synchronous reset

Now we just need to build a serial port receiver

- That's next!