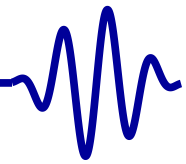
5. A Bus Scope

Gisselquist Technology, LLC

Daniel E. Gisselquist, Ph.D.

Lesson Overview



Objective: Learn how to design your own internal scope

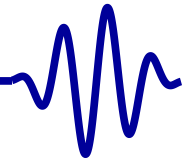
- Learn how to deal with bus interactions requiring multiple clock cycles
- A good internal logic analyzers is a *requirement* for hardware debugging
- Knowing how to build your own scope will make tailoring your own scope easier later
- Here's a story showing how valuable a good home-made internal scope can be

Work in Progress

This lesson is currently a work in progress.

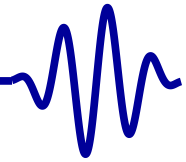
It will remain so until . . .

- I've filled out the simulation section, created an exercise, added host software, and . . .
- I've built the design myself



Project

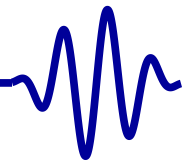
A Scope



So, what do I mean by a "scope"?

- A "scope" is something that collects and displays data
  - Data is often displayed in lines, or traces, across the screen
    We'll just capture the data today
    We can use GTKWave for the display
- Data often arrives faster than it can be displayed or viewed
- With a trigger, a scope can be made to sample relevant data
  - The trigger can be some event, such as an error condition
  - Data can be displayed up to the trigger
  - . . . or even after the trigger

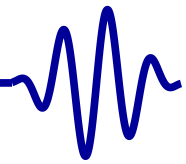
Uses for a scope



Hardware is notoriously difficult to debug

- This isn't software
  - A good software debugger stop your program on any breakpoint
  - While stopped, you can examine any variable in your program at any time
  - You can then step through your design
- Hardware doesn't stop
- It takes hardware to examine hardware
  - Seeing *everything* requires a lot of extra hardware
- A good scope can make it possible to get a glimpse of what's going on within your design

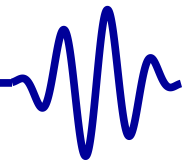
Uses for a scope



An "internal logic analyzer", or internal scope, can make hardware debugging ~~easier~~ possible

- You won't see much of what's going on in your design
  - FPGA RAM space is limited
  - You can't collect everything forever
- You might see enough
  - Sometimes simulation is just too slow
  - Sometimes you need to see how external peripherals interact with your design
  - "Seeing" what's going on can go a long way towards debugging it

Vendor bugs

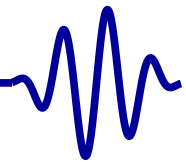


Need help with vendor IP?

- I've been known to wander various vendor forums
- Users report problems with their own and vendor designs on these forums
- (Interface problems are not uncommon)
- Without a trace illustrating the bug, bugs don't get isolated
- It's impossible to tell which component caused the bug

Traces, whether generated from simulation or actual hardware, are essentially *required* for isolating and solving user issues

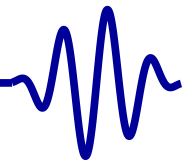
A Bus Scope



What makes a "Bus Scope" special?

- It has a bus based interface
  - The bus provides the infrastructure it needs
  - It's controlled from the bus–not JTAG
  - It's read from the bus
- An on-board CPU can control or trigger it (if desired)
- Unlike vendor-based scopes, a "bus scope" is controlled from within the design itself.

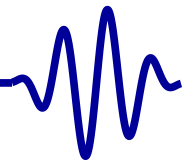
A Bus Scope



Example uses:

- You can hold off triggering until you do something you want to examine
- You can check whether or not an external trigger has taken place, and adjust (i.e. halt) your software at that time
- You can either read back via software, or the debugging bus
- For example, the ZipCPU's test S/W:
  - Manually triggers a CPU scope on any test failure
  - Then outputs details of the test failure
  - The scopes results can then be read and processed externally

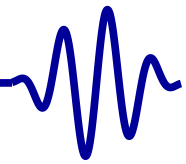
A Bus Scope



DSP designs have special needs

- DSP designs often have to deal with data rates slower than the clock rate
- These signals often include handshaking signals to indicate valid data
  - AXI Stream uses `S_AXIS_TVALID` and `S_AXIS_TREADY` to control such data
    This method often runs into trouble if the source, often an A/D digitizer, can't handle backpressure.
    (Backpressure exists when `VALID && !READY`)
  - I like to use a `CE` signal for this purpose. (I allow no back pressure.)
    This method can still run into trouble when driving a D/A where backpressure may be required

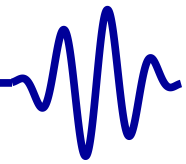
A Bus Scope



DSP Requirements:

- DSP designs can be a challenge to examine when looking at a trace
  - The data is only valid on specific cycles
  - This creates artifacts within traces that can be difficult to interpret
- If we only capture on valid data cycles, our result will be easier to understand
- Therefore, we'll want to *only* capture data when an external data valid signal is true.
- We can use `i_ce` for this purpose

Achille's Heel

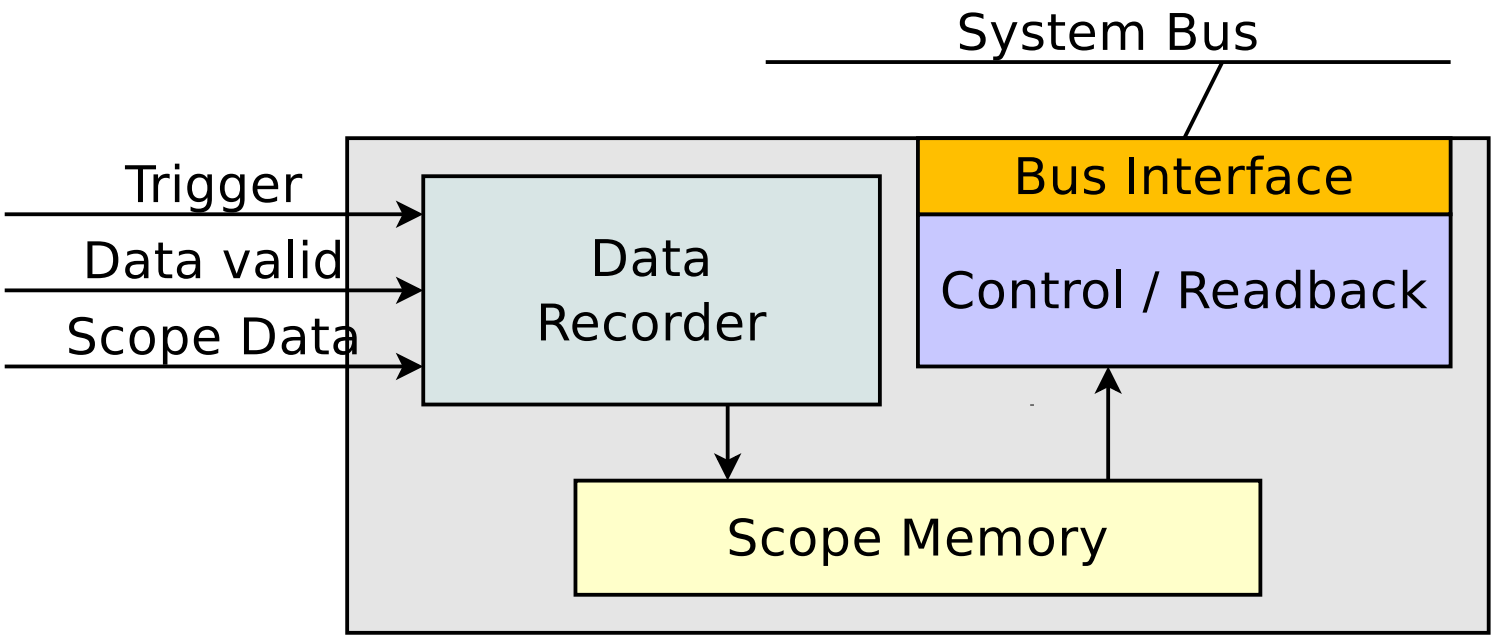


"Bus Scopes" have an Achilles heel:

- If the bus ever locks up, the debugging data becomes inaccessible
- A good formal proof will guarantee the bus doesn't lock up
  - Formal methods become essential here
  - Not just for the bus scope, but for the *entire design*

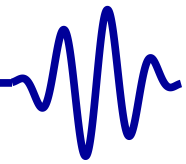
Project Structure

Let's capture these requirements in a drawing:



This should look very similar to our last project

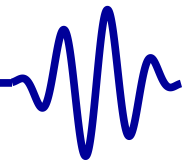
Design Requirements



Before diving in, let's enumerate some requirements

- Since we'll use this trace data for debugging, our data *must* be reliable
  - Since memory cannot be reset, this means we'll need to make certain that all memory is filled before we trigger any data capture
- Captures need to be triggered
  - Triggers can be either described in hardware, or written by the CPU
- The CPU must be able to:
  - Know if the scope has been triggered
  - Read out the results

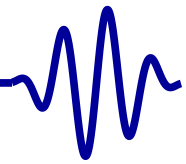
CPU Debugging



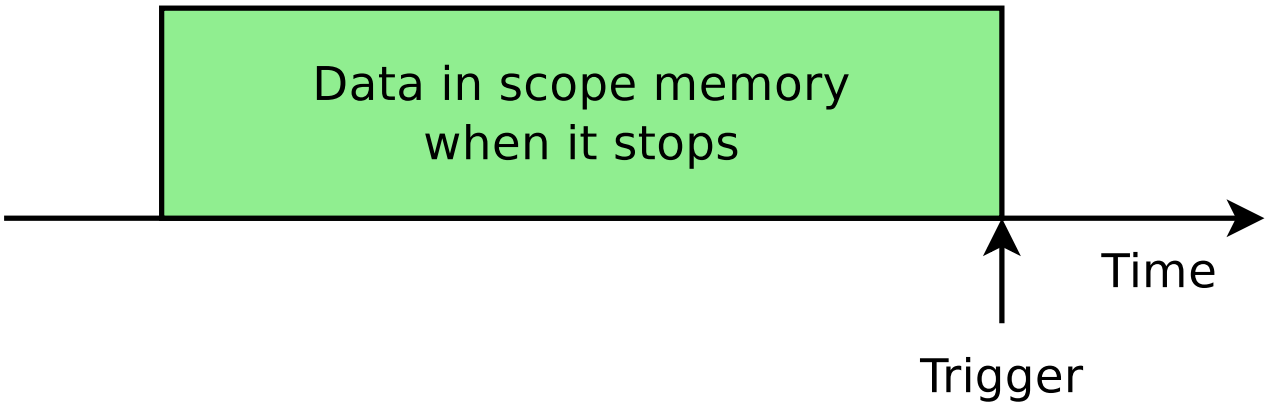
True story: Debugging with no debugging bus

- In one ZipCPU design, the FPGA had no room for a debugging bus
- The CPU, however, still needed debugging
  - *This was before I discovered formal methods*
- A watchdog timer rebooted the CPU if it ever locked up
  - The watchdog timer also triggered the scope
- On reset, the CPU read out the scope's data
  - Yes, this routine was written in assembly
  - Assembly allowed me to capture the CPU's registers on reset as well
- This allowed me to debug the CPU

Trigger Requirements



We want the ability to see into the past

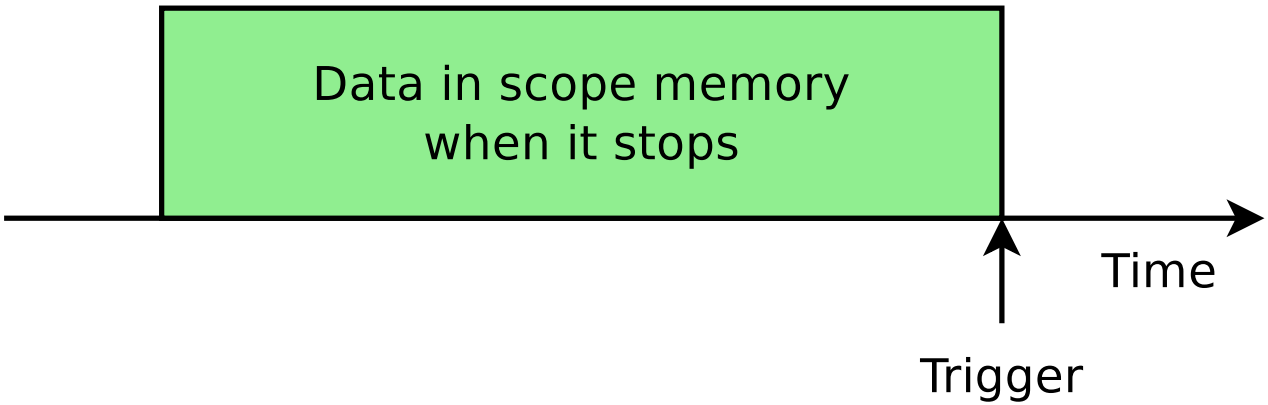


Use cases:

- The trigger describes a bug
- You want to know what lead up to the bug
- For example:
  - Your CPU hangs, and you want to know why

Trigger Requirements

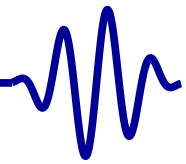
We want the ability to see into the past



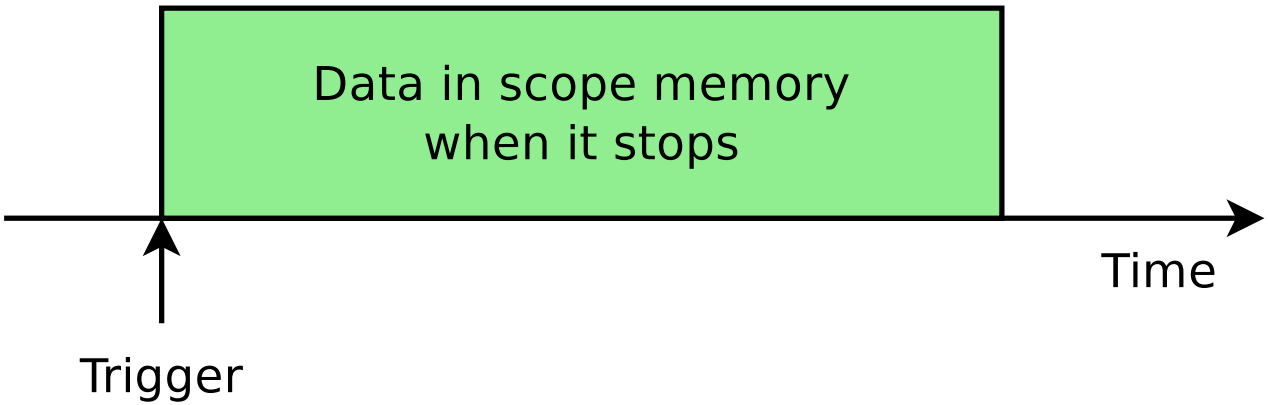
Use cases:

- For example:
  - A CPU self-test fails, and you want to know what happened
  - In this case, you can manually trigger the scope once the bug has been detected
  - The trace will tell you what lead up to the bug

Trigger Requirements



We want the ability to see what's going on now

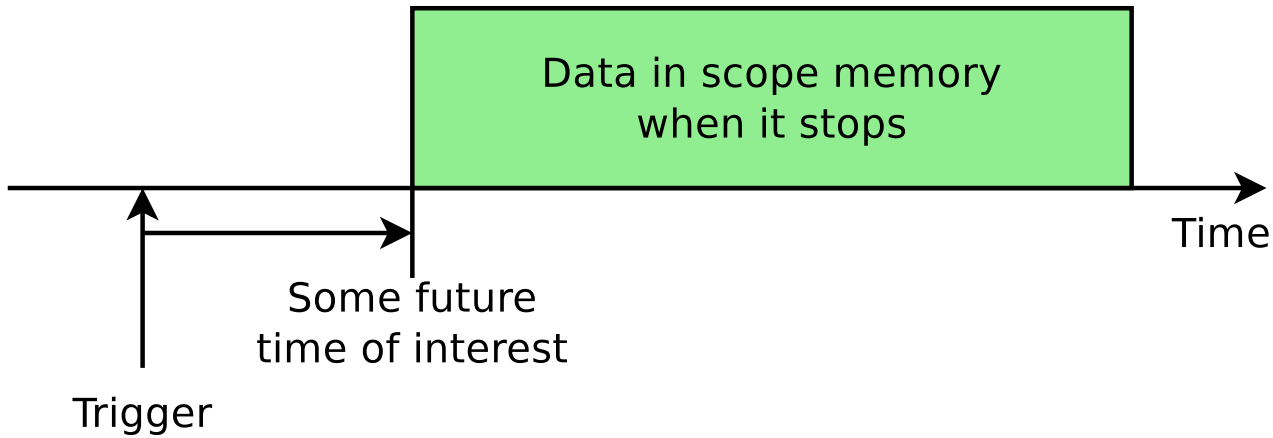


Use cases:

- The trigger describes the beginning of an event
- You want to know what happens next
- Example:
  - You write to the FPGA's configuration port (ICAPE)
  - You want to see and understand what happens next

Trigger Requirements

We want the ability to see what happens next

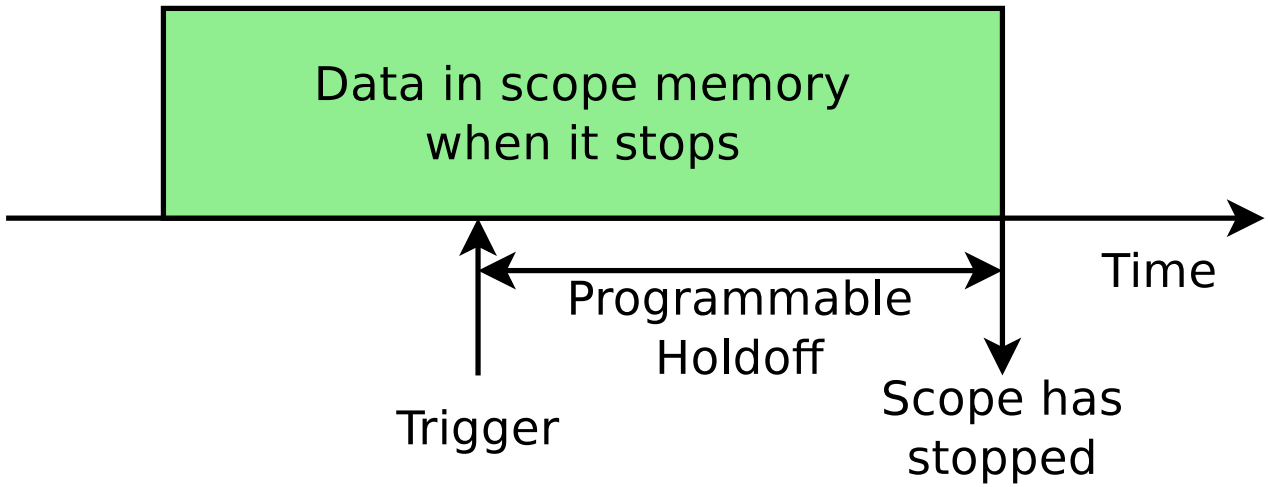


Use cases:

- The trigger describes the beginning of an event
- The capture duration isn't long enough to get all of what happens next
- Example: You are debugging an HDMI input stream
  - You trigger off of a start of frame signal
  - You want to capture the 80th row of video

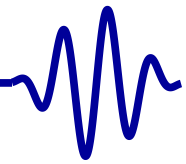
Trigger Requirements

This leads to a programmable holdoff requirement



This programmable holdoff will need to be controlled by the bus

Design Requirements



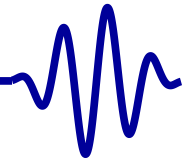
Let's go back to enumerating these requirements:

- Since we'll use this trace data for debugging, our data *must* be reliable
- Data capture is initially continuous
- Captures need to be triggered
  - Data capture stops some programmable time after the trigger
- We must be able to identify the trigger time later
- Data must be read oldest to most recent
- The CPU must be able to:
  - (Continued on the next page)

Design Requirements

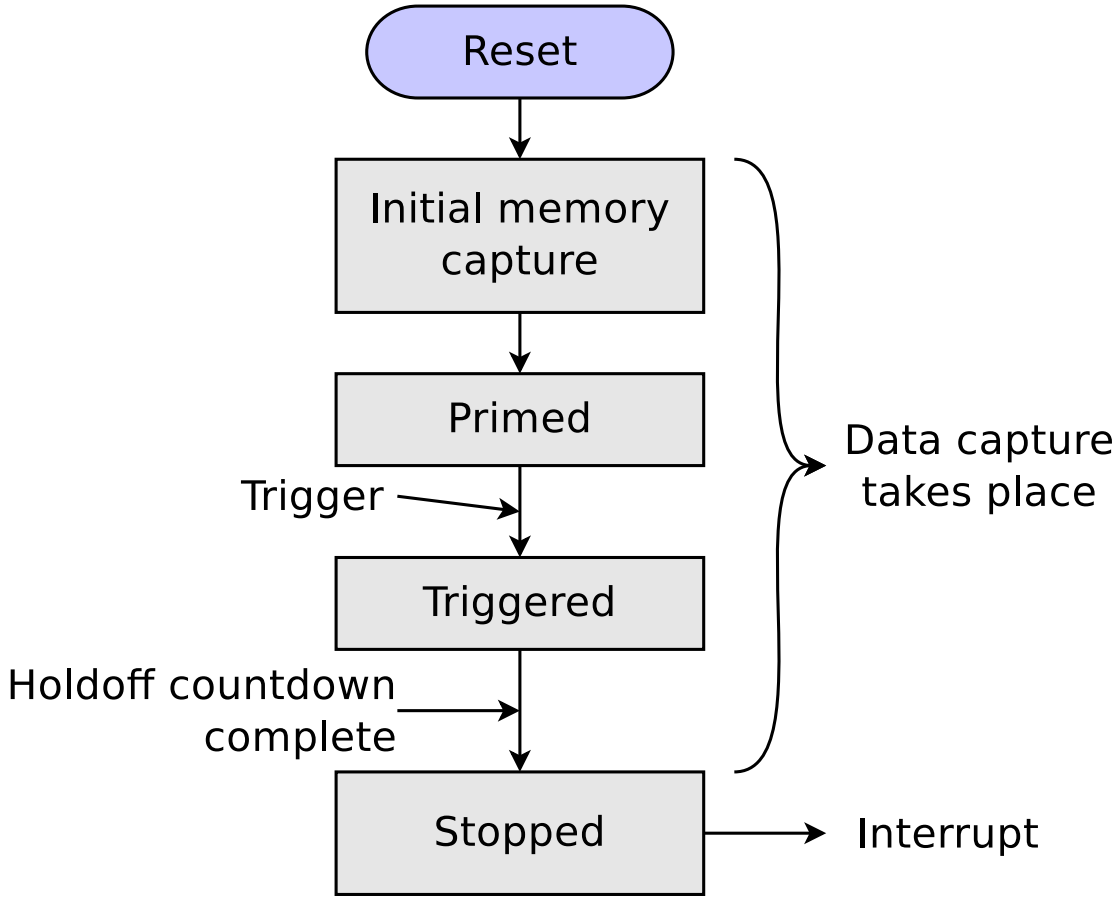
Let's go back to enumerating these requirements: *(Continued)*

- The CPU must be able to:
  - Reset the scope
  - Know when its memory has been filled
  - Know if the trigger has been hit
  - Manually trigger a capture
  - Control the holdoff amount
  - Read the data back
- This control may also be handled via our debugging bus



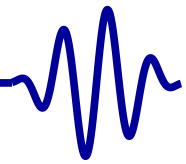
Design

State Machine



These design requirements lend themselves nicely to a basic state machine

State Machine



1. Reset
   - Memory pointers are initialized
2. Initial Memory Capture
   - Incoming data used to fill memory
3. Primed
   - At this point, all memory has been written to
   - Here is where we become sensitive to the trigger
4. Triggered
   - Once the trigger is received, we start a countdown timer
5. Stopped
   - When the countdown timer hits zero, we stop recording

State Machine

Although this design is easily described by a state machine . . .

- I've never built it like a state machine
  - There's no internal **case**(`state`) statement
  - While you could use one, I just never have
- Instead, I use flags to identify the various states
  - `s_reset`
  - `primed`
  - `triggered`
  - `stopped`

Basic Scope Design

Parts of this are easy enough we could almost start immediately:

1. We'll need a memory, a write pointer, and a flag to know if we've stopped collecting

```
parameter W = 32, // Match the bus width
          LGMEM = 12; // Log of the memory size

reg       [W-1:0]           mem [0:(1<<LGNA)-1];
reg       [LGMEM-1:0]       wr_addr;
reg                         stopped;
```

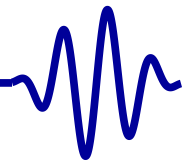
Basic Scope Design

Parts of this are easy enough we could almost start immediately:

1. We'll need a memory, a write pointer, and a flag to know if we've stopped collecting
2. We'll need to write to memory until we've been stopped

```
always @(posedge i_clk)
if (s_reset)
        wr_addr <= 0;
else if (!stopped && i_ce)
        wr_addr <= wr_addr + 1;

always @(posedge i_clk)
if (!stopped && i_ce)
        mem[wr_addr] <= i_scope_data;
```

Scope Design



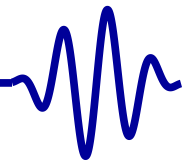
When is the memory full?

```
reg primed;

initial primed = 0;
always @(posedge i_clk)
if (s_reset)
        primed <= 0;
else if (i_ce && !primed)
        primed <= (&wr_addr);
```

Once `primed` becomes true,

- All memory is valid
- Incoming values are now overwriting prior (valid) memory values
- We can now respond to a trigger

Scope Design



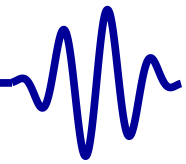
We can only trigger once we've been primed

```
reg     triggered;

initial triggered = 0;
always @(posedge i_clk)
if (s_reset)
        triggered <= 0;
else if (primed && !triggered)
        triggered <= w_trigger;
```

- We only ever trigger once–only a reset clears a trigger
- Once `triggered` becomes true,
  - We'll start our count-down timer

Note that `i_ce` is not required for a trigger to be recognized

Scope Design

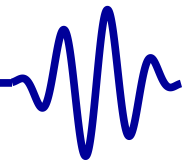


Collection stops `r_holdoff` time steps after the trigger

- This will require a `counter`

```
reg  [HOLDOFFBITS-1:0]  counter, r_holdoff;

initial counter = 0;
always @(posedge i_clk)
if (s_reset || !triggered)
        counter <= r_holdoff;
else if (i_ce && counter > 0)
        counter <= counter - 1;
```

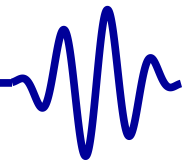
Scope Design



We can now generate our stopped flag.

```
reg     stopped;

initial stopped = 0;
always @(posedge i_clk)
if (s_reset || !primed)
        stopped <= 0;
else if (i_ce && !stopped)
begin
        if (w_trigger && r_holdoff == 0)
                // Trigger now
                stopped <= 1;
        if (triggered && counter == 0)
                // Countdown complete
                stopped <= 1;
end
```

Bus Interface



This design only needs two registers:

0: Control register: Actions include

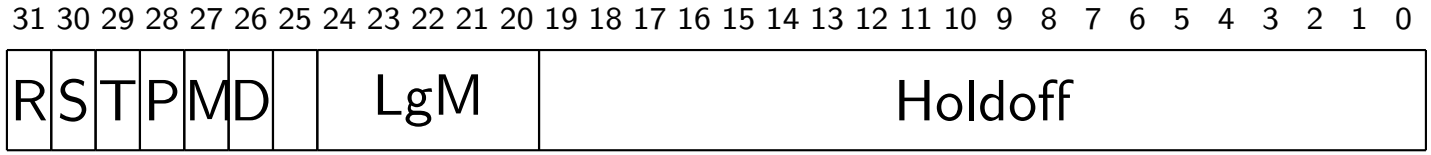
- Reset the scope
- Adjust the holdoff
- Query the state
- Manually trigger the capture
- Disable the hardware trigger

4: Data register

- Read scope data back once capture has stopped
  - Data is read from oldest to most recent
- BONUS: Read the current/active incoming data, before collection is complete

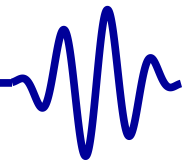
Control Requirements

We'll control our design with a couple of knobs:



R	Reset in progress. Writes automatically reset
S	Capture has stopped (Read only)
T	Design has been triggered (Read only)
P	Memory has been primed (Read only)
M	Manual trigger
D	Disable trigger (write only)
LgM	Log (based two) of the Memory Size (i.e. `LGMEM`)

Internal Reset



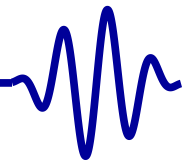
Our design needs two basic resets:

1. On any initial system reset
2. On a user command

On any write, we might adjust critical values and so require a reset

- Writes automatically trigger resets
- . . . *unless* a 1 is written to the reset bit to prevent this automatic reset

Internal Reset



Our design needs two basic resets:

1. On any initial system reset
2. On a user command

This forces us to have a separate reset register, `s_reset`

```
initial s_reset = 1;
always @(posedge i_clk)
if (i_reset)
        s_reset <= 1;
else if (i_stb && !o_stall && i_we && i_addr == 0)
        // Reset on every write,
//    ... unless told otherwise
        s_reset <= !i_sel[3] || !i_data[31];
else
        s_reset <= 0;
```

Trigger Control


```
initial { m_trigger, r_disabled } = 0;
always @(posedge i_clk)
if (i_reset)
begin
        m_trigger  <= 0; // Manual trigger
        r_disabled <= 0; // Disable trigger
end else begin
        if (s_reset)
                m_trigger <= 0;
        if (i_stb && !o_stall && i_we
             && i_addr == 0 && i_sel[3])
        begin
                if (i_data[27])
                        m_trigger <= 1;
                r_disabled <= i_data[26];
        end
end
```

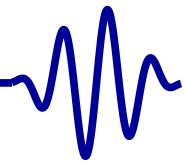
Trigger Control

Style question: When should signals be placed into the same process?

- If they are logically related, and
- If they use the same control structure
- I'm known for using a lot of processes, almost one per signal
  - I've learned to do this to minimize logic usage

In this case, the two signals were logically related and (almost) used the same control structure.

Trigger Control



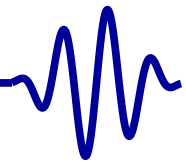
We can now generate an internal trigger signal

- We trigger on any external (hardware) trigger
  - But only if it hasn't been manually disabled
- We'll also trigger manually on any request

```
assign w_trigger = m_trigger
                 || (i_trigger && !r_disabled);
```

I prefixed this signal with `w_` for *wire*, to remind myself that this is a combinatorial signal. `triggered` is our registered copy of this signal.

Holdoff



Holdoff control follows the basic register write logic

```
initial r_holdoff = (1<<LGMEM)-4;
always @(posedge i_clk)
if (i_reset)
        r_holdoff  <= (1<<LGMEM)-4;
end else if (i_stb && !o_stall && i_we
            && i_addr == 0)
begin
        if (i_sel[0])
                r_holdoff <= i_data[7:0];
        if (i_sel[3])
                r_holdoff <= i_data[15:8];
        if (i_sel[3])
                r_holdoff <= i_data[19:16];
end
```

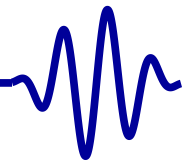
Control Register

Reading from the control register requires composing multiple bits together.

- I like to create a signal to hold these values

```
wire [31:0] w_control;

assign w_control = { s_reset, stopped, triggered,
                     primed, m_trigger, r_disabled,
                     1'b0, LGMEM[4:0], r_holdoff };
```

Reading Data



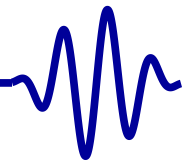
Reading data is a bit more of a challenge

- The address to be read is an offset from the write pointer
  - We need to do this to put the data back in order
  - Read from last to first–regardless of where last and first are in memory
- Our memory rules require that all memory reads require their own clock cycle
  - This can't be merged with the bus address selection
- This gets harder under AXI, but with Wishbone it's pretty easy
- To keep it easier, we'll insist that all bus accesses take the same number of clock cycles

Reading Data


```
always @(posedge i_clk)
if (!stopped)
        // Point to the next write address
        //   This will be the oldest mem location
        rd_addr <= wr_addr + (i_ce) ? 1:0;
else if (i_stb && !o_stall && !i_we && i_addr[0])
        rd_addr <= rd_addr + 1;

always @(posedge i_clk)
if (i_stb && !o_stall && !i_we && i_addr[0])
        rd_data <= mem[rd_addr];
```

Reading Timing



Let's think through this read timing:

1. On the first clock cycle, `i_stb && !o_stall` will be true
   - `rd_addr` will also be valid
2. On the second clock cycle, `rd_data` will be valid
   - To keep everything aligned, we'll need to remember what address was requested in `r_addr`
   - We'll also need to remember we are responding to a request. We can put this into a register `pre_read`
     - Don't forget this register needs to be sensitive to the bus reset!
3. On the third clock cycle we can return data
   - This is also the clock cycle when we'll need to set `o_ack`

Bus Reads


```
always @(posedge i_clk)
        r_addr <= i_addr;

always @(posedge i_clk)
case(r_addr)
0: o_data <= w_control;
1: o_data <= (stopped) ? rd_data : i_scope_data;
endcase

initial { o_ack, pre_read } <= 2'b00;
always @(posedge i_clk)
if (i_reset)
        { o_ack, pre_read } <= 2'b00;
else
        { o_ack, pre_read } <= { pre_read,
                i_stb && !o_stall };

assign o_stall = 1'b0; // Never stall
```

Bus Reads

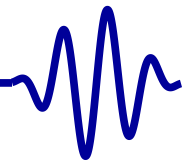

```
always @(posedge i_clk)
if (!stopped)
        rd_addr <= wr_addr;
else if (i_stb && !o_stall && !i_we && i_addr[0])
        rd_addr <= rd_addr + 1;

always @(posedge i_clk)
if (i_stb && !o_stall && !i_we && i_addr[0])
        rd_data <= mem[rd_addr];
```

Interrupts

We may wish to interrupt our processor when the scope has stopped.

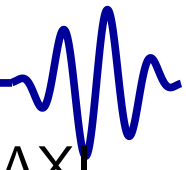
```
assign o_interrupt = stopped;
```

This is easy.
The challenge will come when we wish to build an interrupt controllers that can handle multiple (potential) interrupt sources.



AXI-Lite notes

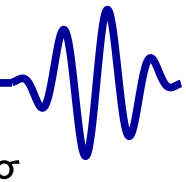
AXI Differences



There are a couple of differences if you are working with AXI

- I've used `i_addr`[0] above, since Wishbone uses word addressing
  - This would become `S_AXI_AWADDR`[2] in AXI-lite, since AXI uses octet (byte) addressing.
    Yes, these is a subtle difference between bytes and octets: Bytes aren't always 8-bits. Indeed, the ZipCPU originally made bytes into 32-bits.
- Otherwise, you can replace:
  - `i_stb && !o_stall && i_we` with `axil_write_ready`
  - `i_stb && !o_stall && !i_we` with `axil_read_ready`
  - Reference our AXI-lite notes for more information
- The big problem is back pressure.

Back-Pressure

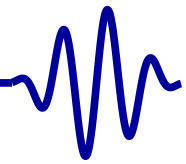


Back Pressure makes AXI pipelines a bit more challenging

- `S_AXI_ARREADY` will only stall the beginning of our pipeline
- Getting this to work requires a couple possibilities. We can either:
  - Only ever allow one item into the pipeline
  - Stall each stage of our pipeline independently
  - Use a (small) FIFO to handle backpressure
- In this design, writes do not suffer from backpressure
  - Our original write ready logic still works

```
assign axil_write_ready=skd_awvalid && skd_wvalid
                && (!S_AXI_BVALID || S_AXI_BREADY);
```

Back-Pressure



Allowing only one read request into the pipeline is easy

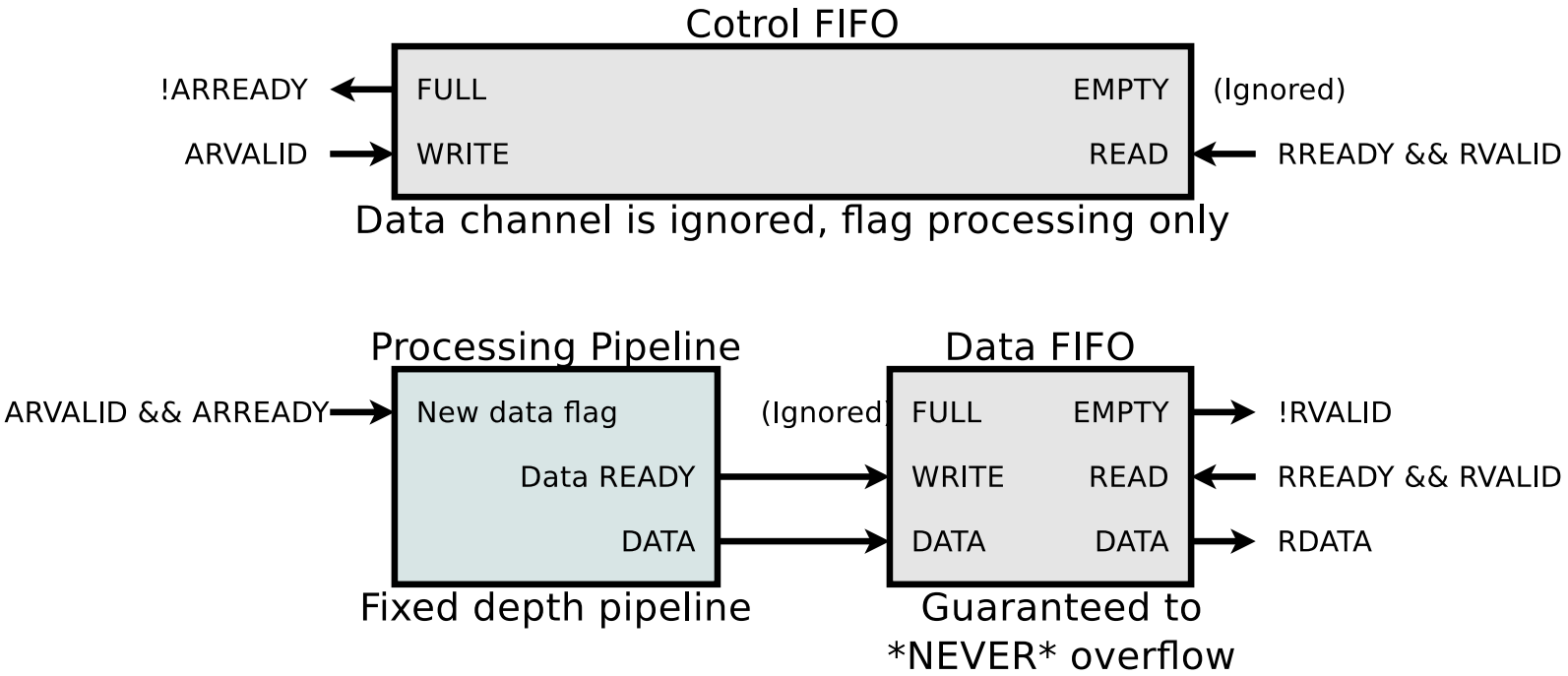
```
assign axil_read_ready = skd_arvalid
                && (!S_AXI_RVALID || S_AXI_RREADY)
                && !pre_read;
```

The problem? This drops our maximum throughput to 50%.

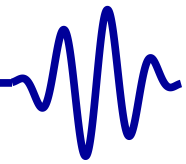
- Is this really a problem? It depends.
- If the scope is for debugging purposes, it might be designed to be rarely read. Slow reads, though important, might not matter.
- If you are routinely reading data from a fast capture, faster reads might be a requirement

Back-Pressure

Another approach is to use two FIFOs



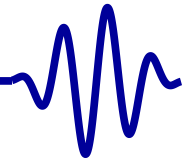
Back-Pressure



This double FIFO approach maintains 100% throughput

- Only the control signals are used on the first FIFO
  - This tells us how to set `S_AXI_ARREADY`
  - Any data channel through the control FIFO is ignored
- The pipeline then feeds a data FIFO
  - The control FIFO guarantees the data FIFO never overflows

This is a common AXI structure. If you are working with AXI, you should become familiar with it.



Formal Verification

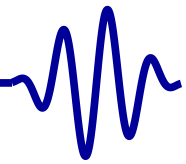
Property Files

Verifying bus components always starts with a bus property file

- You should already know how to do the basics of this
- The new key here is that you'll need to correlate the number of outstanding transactions with the number of items in your pipeline

```
always @(*)
    assert(fwb_outstanding == o_ack + pre_ack);
```

Contract Checks

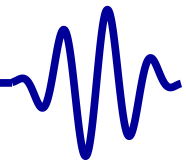


The "Contract" checks that the core works as designed

- Let the solver pick a data input
- Count which input that is
- Verify that the same input can be read back at the right read count
  - You may assume the user doesn't read until the design has stopped.

BONUS: A better check would be the twin write FIFO check

Induction Checks



Let's also verify our control structure:

- `triggered` should never be set if not `primed`
- `stopped` should never be set if not `triggered`
- Pick a value in memory.
  - Verify that it is written to between `s_reset` and `primed`.
- Count the clocks from the trigger to when `stopped` is asserted.
  - Verify that it matches the holdoff

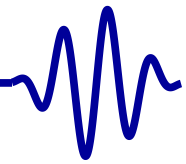
Induction Checks

The assertion for the state machine flags has a basic structure:

```
always @(*)
case({ stopped, triggered, primed })
3'b000: begin end
3'b001: begin end
3'b011: begin end
3'b111: begin
        // We can even add per-state checks here
        assert(counter == 0);
        end
default: assert(0);
```

You are likely to see this again.

Do not pass Go



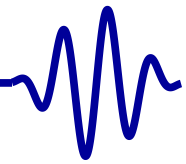
I've said this before, but:

- *Do not proceed to integration until you know your core works!*

Take whatever time you need get it your core to pass

- This applies especially to your bus interfaces
- Do what you can with the rest
- If you miss a bug later, then adjust your properties to catch it next time and come back here and re-do this step

Debugging only gets harder from here on out



AutoFPGA

Bus connection

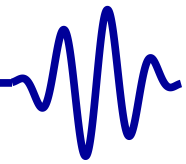
As before, let's use AutoFPGA to wire up this design component

- Because of the pipeline, we can't use any of the canned slave types

  `@PREFIX=busscope`
  `@SLAVE.BUS=wb` *Connect to bus named wb*
  `@SLAVE.TYPE=OTHER` *Nothing special*
- We have only two bus addresses:

  `@NADDR=2` *Two word addresses*
- And one interrupt

Interrupts



We haven't discussed AutoFPGA and interrupts before

- AutoFPGA can create an N-element vector for you to contain interrupt signal sources
- You can then feed this vector to your interrupt controller–whatever it is.

Let's create one of these interrupt vectors:

- The required structure is prefixed with `PIC`
  - `PIC` (*Programmable Interrupt Controller)*

`@PREFIX=buspic`	*An AutoFPGA component*
`@PIC.BUS=int_vector`	*The Verilog name of our int vector*
`@PIC.MAX=15`	*Max # of interrupts in this vector*

Interrupts

Once we have a programmable interrupt control (wire vector) defined, we can now assign interrupts to it.

- There are three tags for this purpose:
  - `@INT.`*NAME*`.WIRE`: The Verilog name of the wire containing the interrupt source. AutoFPGA will create the definition of this wire.
    - *NAME* in this case is your name for the interrupt.
    - It is typically in all caps
  - `@INT.`*NAME*`.PIC`: The name of the PIC to which this interrupt is to be assigned.
  - `@INT.`*NAME*`.ID`: This is optional. If given, it will force the interrupt to have a given position in the interrupt vector.

Interrupts

Once we have a programmable interrupt control (wire vector) defined, we can now assign interrupts to it.

- There are three tags for this purpose: `WIRE`, `PIC`, and `ID`
- Let's assign these:

`@INT.SCOPE.PIC=buspic`	*AutoFPGA PIC PREFIX*
`@INT.SCOPE.WIRE=@$(PREFIX)_int`	*Interrupt wire name*

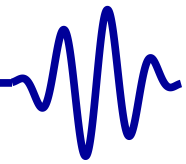
Bus connection

We'll also need to instantiate this scope within our design

@MAIN.INSERT=

```
busscope
@$(PREFIX)i(i_clk, i_reset,
        //
        // The port list for the memory port
        @$(SLAVE.PORTLIST),
        //
        // Scope trigger and debug wire
        @$(TRIGGER), @$(DEBUG),
        //
        // Our interrupt
        @$(PREFIX)_int);
```

Scope connections



So far, I've left the two scope connections undefined.

- These are the `@TRIGGER` and `@DEBUG` inputs

  `@TRIGGER=@$(PREFIX)_trigger` *A (default) trigger def'n*

  `@DEBUG=@$(PREFIX)_debug` *A (default) data def'n*
- They don't need to be AutoFPGA variables
  - However, AutoFPGA has an inheritance capability
  - If we make them AutoFPGA variables, they can then be overridden
  - The following uses our bus scope definition file, `scope.txt` to provide default definitions in a separate AutoFPGA file

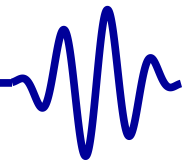
    `@INCLUDEFILE=scope.txt` *Includes the scope definition*
  - In this way one scope configuration can define many scope instances

Scope connections

For example, a scope to examine a flash controller's debug output might look like:

`@PREFIX=flashscope`	*Example: Examine a flash*
`@INCLUDEFILE=scope.txt`	*Includes the scope definition*
`@TRIGGER=flash_trigger`	*Specific to flash controller*
`@DATA=flash_data`	*Debug data Verilog name*

Register Address



We'll also want to know the ultimate address of our core

- This is determined by AutoFPGA
  - It's used internally to configure the interconnect
  - We also want this value in several output files
- The following will put these addresses into a `regdefs.h` file

```
@DEVID=SCOPE                          A register name prefix
@REGS.N=2                             Two named addresses
@REGS.0=0 R_@$(DEVID) @$(DEVID)       Control address
@REGS.1=1 R_@$(DEVID)D @$(DEVID)D     Data address
```

CPU Header

To use this scope from a CPU, we'll need to update our board header

- We'll need some values to interact with this IP
- The following will be copied into a `board.h` file

`@BDEF.DEFN=` *Define our data structure*

```
#ifndef BUSSCOPE_H
#define BUSSCOPE_H
#define BUSSCOPE_NO_RESET   0x80000000u
#define BUSSCOPE_STOPPED    0x40000000u
#define BUSSCOPE_TRIGGERED  0x20000000u
#define BUSSCOPE_PRIMED     0x10000000u
#define BUSSCOPE_TRIGGER \
            (BUSSCOPE_NO_RESET | 0x08000000u)
#define BUSSCOPE_MANUAL     BUSSCOPE_TRIGGER
#define BUSSCOPE_DISABLE    0x04000000u
// Continued next page ...
```

CPU Header

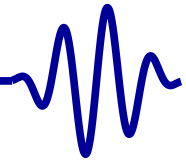
Our design structure consists of two memory addresses: a control register and a data register. Let's define a structure containing these.

```
// Continued ...
typedef struct BUSSCOPE_S {
  unsigned s_ctrl, s_data;
} BUSSCOPE;
#endif
```

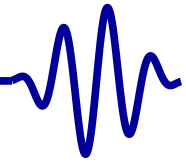
We need to know one more piece: where to find this scope in memory

`@BDEF.OSVAL=` *Define our memory's base address*

```
static volatile BUSSCOPE *const @(PREFIX)
        = ((BUSSCOPE *)@[0x%08x](REGBASE));
```



Simulation



Host Control

Data Capture

Using our debugging bus, it's easy to capture the data from this scope

```
// Wait for the scope collection to stop
while((m_fpga->readio(R_SCOPE)
                    & BUSSCOPE_STOPPED)==0)
        ;

// Allocate memory
unsigned *scopdata = new unsigned[(1<<LGMEM)];

// Read the data from the FPGA
//   Reference lesson 1 on the debugging
//   bus for more info
m_fpga->readz(R_SCOPED, (1<<LGMEM), scopdata);
```

Data Output

The easy way to output this data is just to write it to the console

```
for(int k=0; k<(1<<LGMEM); k++) {
        printf("%4d: 0x%08x\n", k, scopdata[k]);
```

While I've debugged data like this, the resulting output is a challenge to work with

Data Decoding

You can also adjust this output for your purposes

- Imagine your data contained Wishbone bus information
- It might help to decode this to make it more readable
- Key requirement: Line your fields up in columns for easier readability

```
for(int k=0; k<<(1<<LGMEM); k++) {
    printf("%4d: 0x%08x %s %s %s\n",
        k, scopdata[k],
        (scopdata[k] & 0x8000) ? "CYC":"   ",
        (scopdata[k] & 0x4000) ? "STB":"   ",
        (scopdata[k] & 0x2000) ? "WE":"  ");
```

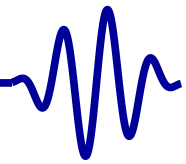
This is better, but it's still a challenge

VCD Generation

If we want to view our trace in GTKWave, we'll need to write a VCD file

- VCD files aren't that hard to write
- They're just text files
- Basic components
  1. File Header
     - Data Definitions
  2. Data section consists of repeated sections of:
     - Clock time
     - Data lines: (Value) (Data)
- Let's look at each piece in turn

VCD Header



For the first part, we need just three lines

- The first just identifies the program creating the VCD file

```
$version Generated by MyBusScope $end
```

- The next line identifies when the file was created
  - We can use **ctime**() to create this string

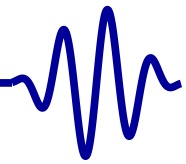
```
$date Mon xx Mon Year HH:MM:SS xM xxT $end
```

- The last line tells the viewer what time scale we are using

```
$timescale 1ns $end
```

  - This says that all of the times we generate will be in nanoseconds
  - If you had a reason to, you could also use 1ps, 10ns, etc.

VCD Header

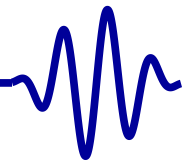


A fourth line is useful, but optional

```
$timezero <TriggerTime> $end
```

- <**TriggerTime**> here is the internal file time of the trigger
- This is (really) optional, but I like using it to identify where the trigger took place for easier viewing

VCD Data Definition

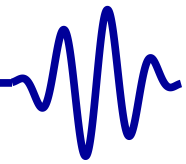


We now need to define our data

- The data definition section starts with a `$scope` line

```
$scope module MyBusScope $end
```

VCD Data Definition



Data definition (continued)

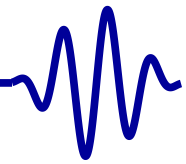
- Signal definitions are contained in lines starting with `$var wire`
- One signal is defined per line
  - This definition has three parts: a width, an abbreviation, and a full name
  - We get to choose what abbreviation we'd like

```
$var wire <WID> <ABBRV> <NAME> $end
```

- For example, we'll need to define our clock signal, the raw data we captured, and our trigger signal

```
$var wire  1 xC i_clk $end
$var wire 32 xD _raw_data[31:0] $end
$var wire  1 xT _trigger $end
```

VCD Data Definition



Data definition (continued)

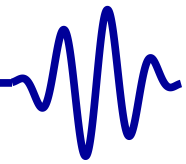
- When all definitions are complete, we'll move up a scope and complete the definitions section

```
  $upscope $end
$enddefinitions $end
```

  - This also completes the header

All that remains is to fill our data file with values

VCD Data Lines



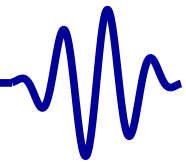
There are two types of data lines

- Time lines
  - These start with a # followed by a number specifying the time within the collect.
  - For a 10ns clock, these times might be #0, #10, #20, etc.

```
#10
```

- Data lines: (Next page)

VCD Data Lines



There are two types of data lines

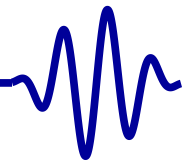
- Time lines
- Data lines: Here again there are two types
  - Binary (1,0) data lines contain the value followed by the abbreviation

```
1xC
1xT
```

  - Wider data lines begin with a b, followed by (width) digits of (1,0), then the signal's abbreviation

```
b01101100111110111100010101101010O0 xD
```

VCD Data Lines



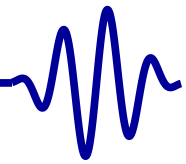
There are two types of data lines

- Time lines
- Data lines: Binary and wider
- Values not defined in any given time step keep their value from the prior timestep
- To make the clock look right, you'll need to have a time step where it's high, and another where it's low

```
#15
0xC
```

- Time lines must be in sorted order, you can't go backwards!

ScopeCls



I have a C++ SCOPE class I use for this purpose

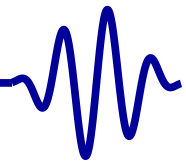
- To use, first create your own class inheriting from it

```
class   MYSCOPE : public SCOPE  {
public:
        MYSCOPE(DEVBUS *fpga, unsigned addr)
                : SCOPE(fpga, addr,
                        false, // Compressed?
                        true) {};

        virtual void define_traces(void);
}
```

- Four parameters need to be defined

1. The first is a pointer to the DEVBUS interface

ScopeCls



I have a C++ SCOPE class I use for this purpose

- To use, first create your own class inheriting from it
- Four parameters need to be defined
  1. The first is a pointer to the DEVBUS interface
  2. The second is the address of the scope's bus interface
  3. The third is an option for a compressed scope. We'll just set this to false for now.
  4. The final option controls if `readz()` or `readio()` is used.
     - `readz()` (true) is faster, and to be preferred
       - `readz()` reads multiple items at a time
       - All items are read from the same address
     - `readio()` is often easier to get working first
       - `readio()` reads one item at a time

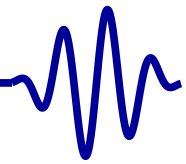
ScopeCls

You'll then need to override the `define_traces()` method

- To override `define_traces()`: Define each component of your incoming data value, give it a name, a width, and the bit it starts from

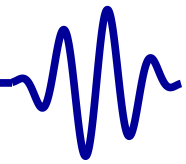
```
void define_traces(void) {
    register_trace("signame", 1, 30);
    register_trace("sigtwo",  2, 28);
    register_trace("third",   4, 24);
    // etc.
}
```

main()



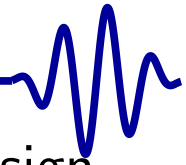
A simple main program is all that remains to use this

```
FPGA *m_fpga;

int main(int argc, char **argv) {
    FPGAOPEN(m_fpga); // Connect to FPGA/sim
    MYSCOPE *scope = new MYSCOPE(m_fpga,
        R_SCOPE);
    if (!scope->ready()) {
        printf("Scope hasn't stopped yet\n");
    } else
        scope->writevcd("trace.vcd");
}
```



Hardware

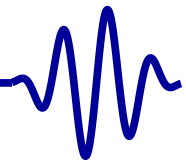
Build it!



You should now be able to include this scope into any design

- Add a scope to your wavetable design
- Does the resulting waveform look like a sine wave?
- Is it at the right frequency?
- If not, then why not?

Compression



Waveform traces can become really long

- Our demo does nothing to compress the data it collects
- A simple run-length compression isn't that hard to build
  - Clear bit 31 if bits 30-0 contain data
  - Set bit 31 to indicate the last value is repeated 1+`data`[30:0] times
- Now you can debug crazier things:
  - SPI Flash devices
  - Serial ports
    - GPS data streams
    - Does the GPS PPS come before, or after, the time given in the serial port?
  - I2C interactions, such as HDMI EDID ports