2. AXI-Lite

Gisselquist Technology, LLC

Daniel E. Gisselquist, Ph.D.

Lesson Overview

Objective: To learn and become familiar with using a AXI-lite bus

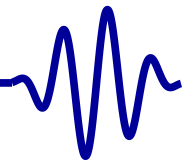
- AXI is complicated. AXI-lite is (somewhat) better.
  - The handshaking can be hard to get right
  - Handshaking is common to both
- We'll use a AXI-lite bus simplifier in this lesson
  - This will convert AXI-lite into something that handles handshaking across multiple components
  - A similar AXI simplifier also exists
  - Doesn't impact throughput
- Modify our basic general purpose I/O controller for AXI-Lite
- Discuss and invoke bus bridges
  - Wishbone to AXI-Lite
  - AXI-Lite to Wishbone

Work in Progress

This lesson is currently a work in progress.

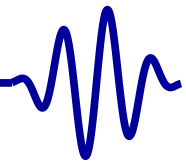
It will remain so until . . .

- I have a verified the instructions and
- Used them to generate a working example on my own



AXI-Lite Simplified

5 Channels



AXI and AXI-Lite both use five separate channels

AW Write address channel
- Sends the address to be written to

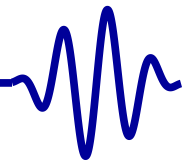
W Write data channel
- Prefixed with W
- Sends the data to be written to that address
- The slave will need to synchronize this with the write address channel

B Write response channel

AR Read address channel
- Requests a read from a given address

R Read data channel
- Returns any values read

Handshaking



Each channel has associated VALID and READY signals

- When the source wishes to send information, it sets VALID
- When a sink is ready to receive information, it sets READY
  The AXI specification recommends leaving READY high if possible To prevent deadlock, the VALID signal is not allowed to wait for READY to be high
- A transaction takes place whenever `VALID && READY`
- A channel is stalled whenever `VALID && !READY`
  In this case, the source *may not change* any associated data
- All VALIDs must be cleared on reset

Backpressure

Backpressure: when the master isn't ready to receive the return

- Ex: master requests several writes, but holds BREADY low
- Ex: master requests several reads, but holds RREADY low
- When sufficiently backed up, the slave must lower AxREADY

The correct logic *should* look something like,

```
always @(*)
begin
    AWREADY = AWVALID && WVALID
                && (!BVALID || BREADY);
    WREADY  = AWREADY;

    ARREADY = ARVALID && (!RVALID || RREADY);
end
```

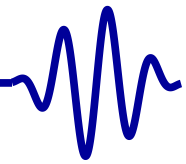
Such combinatorial logic is disallowed by the AXI4 specification

Registered

The AXI4 specification *requires* that outputs cannot depend combinatorially upon inputs

- All bus signals *must* be registered
- This includes AxREADY
- It will take one clock from when the slave detects an output stall until it can lower AxREADY
- During that clock period, if AxREADY is high, a request could be lost

Handshake Solutions



Solutions to this problem include:

- Holding AxREADY low until AxVALID is true

```
always @(posedge S_AXI_ACLK)
if (!S_AXI_ARESETN)
        S_AXI_AWREADY <= 1'b0;
else if (S_AXI_AWVALID && S_AXI_WVALID
        && (!S_AXI_BVALID || S_AXI_BREADY))
        S_AXI_AWREADY <= 1'b1;
else
        S_AXI_AWREADY <= 1'b0;

always @(posedge S_AXI_ACLK)
if (!S_AXI_ARESETN)
        S_AXI_BVALID <= 1'b0;
else if (!S_AXI_BVALID || S_AXI_BREADY))
        S_AXI_BVALID <= S_AXI_AWREADY;
```

Handshake Solutions

Solutions to this problem include:

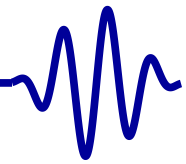
- Holding AxREADY low until AxVALID is true
  - This will limit any core to 50% throughput at best
  - This was the solution Xilinx attempted

Handshake Solutions

Solutions to this problem include:

- Holding AxREADY low until AxVALID is true
- Using a skid buffer
  - Buffers the transaction that would be otherwise lost
  - Can be used to achieve 100% throughput
  - Building a skid buffer is a sublesson in itself
  - We'll come back to this in another lesson

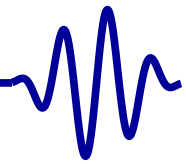
Handshake Solutions



Solutions to this problem include:

- Holding AxREADY low until AxVALID is true
- Using a skid buffer
- Simplifying the specification
  - If a group of slaves all shares the same basic logic
  - The handshaking work can be shared
  - Internal buffers can be shared
  - AutoFPGA will do this for `SLAVE.TYPE`s of both SINGLE and DOUBLE
  - Still achieve 100% throughput

Simplified



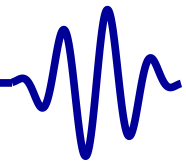
Simplified AXI protocol rules:

- All ready signals are held high
- All returns come one clock after xVALID
  - The simplifier ignores the slave's `BVALID`, `RVALID`, `AxREADY` and `WREADY` signals
  - Any slave that would otherwise set its outgoing valid signals with a one clock delay will work here
  - You can also cheat ...
  - We'll cheat today, to get you up and running quickly
- Write address and data channels synchronized
- Concurrent read and write support is required

This takes all the hard work out of building an AXI-Lite slave

- The difficult synchronization tasks are left to the simplifier
- The bus simplifier can be made application agnostic

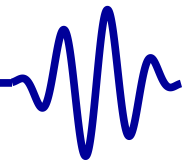
Simplified



Xilinx offers a similar bus simplifier

- They call it “IPIF”
- It only allows one read or one write transaction at a time
- It will get at best 25% throughput

Two simplifiers



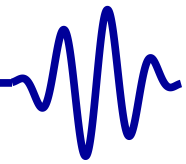
Two types of AXI-Lite simplifiers in this repository

- AXI-Lite SINGLE
  - Slaves may only ever have one register
  - Simplifies the bus for a group of slaves
  - We'll discuss these today
- AXI-Lite DOUBLE
  - Slaves may have many registers
  - Must still respond the clock following AxVALID
  - Simplifies the bus for a group of slaves
  - These will be the focus of lesson three

There's also a similar AXI4 simplifier in the same repo

- Supports burst access, and multiple transaction IDs
- Even supports exclusive bus access

AutoFPGA

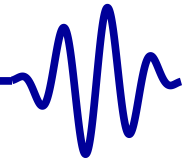


AutoFPGA will automatically connect bus simplifiers

- Groups all slaves of a given type on the bus
- Slaves of `@SLAVE.TYPE=OTHER` will not be simplified

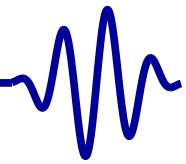
`@BUS.TYPE`	`@SLAVE.TYPE`	Simplifier
AXI4-Lite	`SINGLE`	axilsingle.v
AXI4-Lite	`DOUBLE`	axildouble.v
AXI4	`SINGLE`	axisingle.v (not yet written)
AXI4	`DOUBLE`	axidouble.v
WB	`SINGLE`	(added in-line)
WB	`DOUBLE`	(added in-line)

For these groups, only one master can access the group at a time



Updated Controller

Wishbone Writes

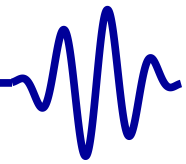


Here was our wishbone write logic:

```
// Bus slave logic
assign  o_wb_stall = 1'b0;
assign  o_wb_ack   = i_wb_stb;
assign  o_wb_data  = { 31'h0, o_led };

always @(posedge i_clk)
if (i_wb_stb && i_wb_we)
        o_led <= i_wb_data[0];
```

AXI-Lite Writes



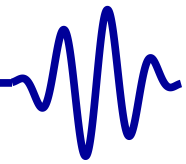
Here is our modified AXI-Lite write logic:

```
// Always return "okay"
always @(*)
        S_AXI_BRESP <= 2'b00;

// Actually set our I/O
always @(posedge i_clk)
if (S_AXI_AWVALID && S_AXI_WSTRB[0])
        o_led <= S_AXI_WDATA[0];
```

The simplifier takes care of everything else

AXI-Lite Writes



For completeness sake, you can set the other signals

```
always @(*)
begin // Always be ready
        S_AXI_AWREADY <= 1'b1;
        S_AXI_WREADY  <= 1'b1;
end

// Acknowledge on the clock following AWVALID
always @(posedge i_clk)
if (!S_AXI_ARESETN)
        S_AXI_BVALID <= 0;
else
        S_AXI_BVALID <= S_AXI_AWVALID;
```

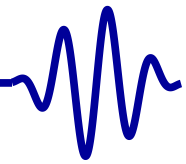
They'll be ignored, but at least they'll be (roughly) valid

Wishbone Reads

Here was our wishbone read logic:

```
        // Bus slave logic
        assign  o_wb_stall = 1'b0;
        assign  o_wb_ack   = i_wb_stb;
        assign  o_wb_data  = { 31'h0, s_button };
```

AXI-Lite Reads

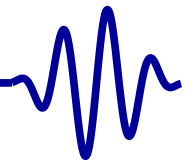


Here's our modified AXI-Lite read logic:

```
always @(*)
        S_AXI_RRESP = 2'b00;

always @(posedge S_AXI_ARESETN)
        S_AXI_RDATA <= { 31'h0, s_button };
```

AXI-Lite Reads

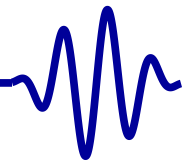


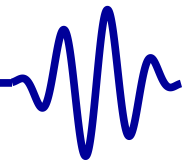
For completeness, you can set the other signals

```
always @(*) // Always be ready
        S_AXI_ARREADY <= 1;

// Return one clock after any request
always @(posedge S_AXI_ACLK)
if (!S_AXI_ARRESET)
        S_AXI_RVALID <= 1'b0;
else
        S_AXI_RVALID <= S_AXI_ARVALID;
```

Module declaration





Bridging buses

AutoFPGA

Two approaches to integrating this module into our design

- We can start w/ WB and bridge to AXI-Lite

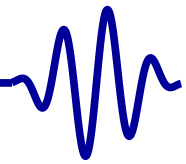
AutoFPGA

Two approaches to integrating this module into our design

- We can start w/ WB and bridge to AXI-Lite, or
- We could start w/ AXI-Lite and bridge to WB

PICTURE

Bridge to AXI-Lite

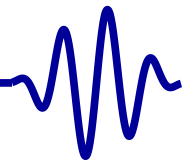


Let's add a bridge to AXI-Lite

- The WB2AXIP repo contains a WB to AXI-Lite bridge
- We need to tell AutoFPGA about it
  - We'll start by defining a new bus we'll call `axil`
  - It'll also be of type `axil`, for AXI-Lite

```
@PREFIX=wb2axil
@BUS.NAME=axil
@BUS.TYPE=axil
@BUS.CLOCK=clk
```

Bridge to AXI-Lite



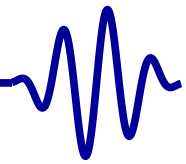
Let's bridge from WB to AXI-Lite

- We can add the bridge description to the same component
- A bridge has both master and slave interfaces
- The master interface is that of a SUBBUS
  - This allows the slave's address width to be determined by the subbus'es address width

```
@SLAVE.BUS=wb
@SLAVE.TYPE=OTHER
@MASTER.BUS=axil
@MASTER.TYPE=SUBBUS
@RTL.MAKE.GROUP=WB2AXIL
@RTL.MAKE.SUBD=../wb2axip
@RTL.MAKE.FILES=wbm2axilite.v
```

Tell make about these files

Bridge to AXI-Lite

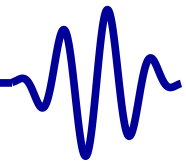


Let's bridge from WB to AXI-Lite

- Depending on how your design build is set up
- You might want to add `wbm2axilite.v` to your Makefile
- This can be done via AutoFPGA as well

```
@RTL.MAKE.GROUP=WB2AXIL
@RTL.MAKE.SUBD=../wb2axip
@RTL.MAKE.FILES=wbm2axilite.v
```

Bridge to AXI-Lite



Let's bridge from WB to AXI-Lite

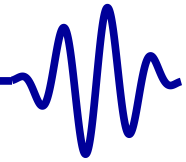
- The last step is the actual logic of the bridge
- `MASTER.BUS.AWID` is the bus'es address width
  - Determined by the sub-bus'es slave address widths

`@MAIN.INSERT=` *// copy this code into main.v*

```
wb2axilite #(
        .C_AXI_ADDR_WIDTH(@$(MASTER.BUS.AWID)))
@$(PREFIX)i(i_clk, i_reset,
        @$(SLAVE.PORTLIST),
        @$(MASTER.PORTLIST));
```

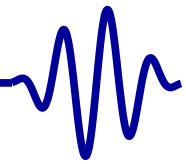
- `MASTER.BUS.AWID` is the bus'es address width
  - Determined by the sub-bus'es slave address widths

Consequences



Bridges have consequences

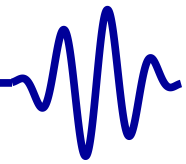
Consequences



Bridges have consequences

- Not all features translate
  - AxPROT field records data vs instruction access
    - No WB equivalent
  - `AWVALID` & `ARVALID` might both be true
    - WB only allows read or write, never both
  - Wishbone achieves lock by holding `CYC` high
    - AXI-Lite has no equivalent
    - AXI has AxLOCK, but semantics don't translate
  - AXI byte zero is xDATA[7:0]
    - Wishbone has no explicit endianness
    - May require an endian swap

Consequences

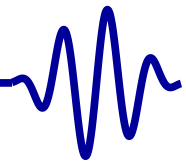


Bridges have consequences

- Not all features translate
- Yes, each of these bridges is high performance
  - Each can achieve 100% throughput
- You'll pay for every bridge
  - Each bridge introduces a lag
  - Not all features will translate
  - I haven't mentioned the logic cost

Don't bridge across buses if you don't need to

Bridge to Wishbone

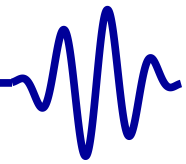


We could also bridge from AXI-Lite to WB

- The process is the same
- The core is configured a touch different
  - We can either bridge from AXI-Lite write and read channels separately, or
  - We can arbitrate between the two and bridge them together into one WB channel
  - If we do the latter, a good crossbar would still allow us two separate channels of WB
- Either way the AutoFPGA configuration starts the same way

```
@PREFIX=axil2wb
@SLAVE.BUS=axil
@SLAVE.TYPE=OTHER
@MASTER.BUS=wb
@MASTER.TYPE=SUBBUS
```

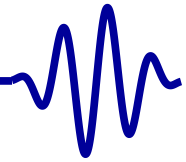
AXI-Lite Hexbus



We would need an original AXI-Lite master

- There's one in the dbgbus repo
- It's called `hbaxil.v`
- You should be able to modify `hexbus.txt` to work with `hbaxil.v`
  - Replace `hbbus` with `hbaxil`
  - Add `hbaxil.v` and `hbexecaxil.v` to the file lists
  - Change the `MASTER.BUS` to read `axil`
  - You can leave the `wb` bus definition in place–it won't hurt anything, and we'll still need it anyway

Your turn



Your turn: Do your AXI-Lite LED and buttons work?

- Try the AXI-Lite `blinker`
- Does the LED still turn on when you press the button?
- Does the LED still turn off when you release the button?
- Can you bridge from WB to AXI-Lite?
- Can you bridge from AXI-Lite to WB?