UP_WISHBONE_CLASSIC

November 18, 2024

Jay Convertino

Contents

1 Usage

1.1 Introduction

This core converts the Wishbone Classic bus to the uP bus. This allows any core with a uP bus to be interfaced with a Wishbone Classic bus. Combination of combinatorial logic and synchronous logic.

1.2 Dependencies

The following are the dependencies of the cores.

- fusesoc 2.X
- iverilog (simulation)
- cocotb (simulation)

1.2.1 fusesoc_info Depenecies

- dep
  - AFRL:utility:helper:1.0.0
  - ::wb_common:1.0.3

1.3 In a Project

This core is made to interface Wishbone Classic bus to uP based device cores. This is part of a family of converters based on Analog Devices uP specification. Using this allows usage of Analog Devices AXI Lite core, AFRL APB3, AFRL Wishbone Classic, and AFRL Wishbone Pipeline converters. Meaning any uP core can be easily customized to any bus quickly. These are made for relativly slow speed bus device interfaces. An example of a Verilog uP interface provided below.

```
//output signals assigned to registers.
assign up_rack  = r_up_rack & up_rreq;
assign up_wack  = r_up_wack & up_wreq;
assign up_rdata = r_up_rdata;
assign irq      = r_irq;

assign s_rx_ren = ((up_raddr[3:0] == RX_FIFO_REG) &&
    ↪ up_rreq ? r_up_rack & r_rx_ren : 0);

//up registers decoder
always @(posedge clk)
```

```
begin
  if(rstn == 1'b0)
  begin
    r_up_rack   <= 1'b0;
    r_up_wack   <= 1'b0;
    r_up_rdata  <= 0;

    r_rx_ren    <= 1'b0;

    r_overflow  <= 1'b0;

    r_control_reg <= 0;
  end else begin
    r_up_rack   <= 1'b0;
    r_up_wack   <= 1'b0;
    r_tx_wen    <= 1'b0;
    r_rx_ren    <= 1'b0;
    r_up_rdata  <= r_up_rdata;
    //clear reset bits
    r_control_reg[RESET_RX_BIT] <= 1'b0;
    r_control_reg[RESET_TX_BIT] <= 1'b0;

    if(rx_full == 1'b1)
    begin
      r_overflow <= 1'b1;
    end

    //read request
    if(up_rreq == 1'b1)
    begin
      r_up_rack <= 1'b1;

      case(up_raddr[3:0])
        RX_FIFO_REG: begin
          r_up_rdata <= rx_rdata & {{(BUS_WIDTH*8-
              ↪ DATA_BITS){1'b0}}, {DATA_BITS{1'b1}}};
          r_rx_ren <= 1'b1;
        end
        STATUS_REG: begin
          r_up_rdata <= {{(BUS_WIDTH*8-8){1'b0}},
              ↪ s_parity_err, s_frame_err, r_overflow,
              ↪ r_irq_en, tx_full, tx_empty, rx_full,
              ↪ rx_valid};
          r_overflow <= 1'b0;
        end
        default:begin
```

```
            r_up_rdata <= 0;
          end
        endcase
      end

      //write request
      if(up_wreq == 1'b1)
      begin
        r_up_wack <= 1'b1;

        //only allow write once ack (Analog Devices does
            ↪ the same)
        if(r_up_wack == 1'b1) begin
          case(up_waddr[3:0])
            TX_FIFO_REG: begin
              r_tx_wdata  <= up_wdata;
              r_tx_wen    <= 1'b1;
            end
            CONTROL_REG: begin
              r_control_reg <= up_wdata;
            end
            default:begin
            end
          endcase
        end
      end
    end
  end

  //up control register processing and fifo reset
  always @(posedge clk)
  begin
    if(rstn == 1'b0)
    begin
      r_rstn_rx_delay <= ~0;
      r_rstn_tx_delay <= ~0;
      r_irq_en <= 1'b0;
    end else begin
      r_rstn_rx_delay <= {1'b1, r_rstn_rx_delay[
          ↪ FIFO_DEPTH-1:1]};
      r_rstn_tx_delay <= {1'b1, r_rstn_rx_delay[
          ↪ FIFO_DEPTH-1:1]};

      if(r_control_reg[RESET_RX_BIT])
      begin
        r_rstn_rx_delay <= {FIFO_DEPTH{1'b0}};
```

```
      end

      if(r_control_reg[RESET_TX_BIT])
      begin
        r_rstn_tx_delay <= {FIFO_DEPTH{1'b0}};
      end

      if(r_control_reg[ENABLE_INTR_BIT] != r_irq_en)
        r_irq_en <= r_control_reg[ENABLE_INTR_BIT];
      end
    end
  end
```

2 Architecture

The only module is the up_wishbone_classic module. It is listed below.

- **up_wishbone_classic** Convert Wishbone Classic to the Analog Devices uP BUS. (see core for documentation).

This core has two generate blocks, and two always blocks. They are listed below.

generate:

1. Part Select Write generates uP signals that have the non-selected elements blanked out with zeros.
2. Part Select Read generates Wishbone classic signals that have non-selected elements blanked out with zeros.

always:

- Burst State Control
  1. Based on Wishbone Classic CTI state, change request logic.
  2. If CTI burst does not match a configuration, no request is allowed.
  3. Check the address state, if we are currently requesting burst mode then change to self incrementing address.
  4. If we hit the last in burst mode, switch out of it.
- Reset Hold, holds reset for 8 more clock cycles to comply with Wishbone Standard.

Please see 5 for more information.

3 Building

The Wishbone Classic core is written in Verilog 2001. They should synthesize in any modern FPGA software. The core comes as a fusesoc packaged core and can be included in any other core. Be sure to make sure you have meet the dependencies listed in the previous section.

3.1 fusesoc

Fusesoc is a system for building FPGA software without relying on the internal project management of the tool. Avoiding vendor lock in to Vivado or Quartus. These cores, when included in a project, can be easily integrated and targets created based upon the end developer needs. The core by itself is not a part of a system and should be integrated into a fusesoc based system. Simulations are setup to use fusesoc and are a part of its targets.

3.2 Source Files

3.2.1 fusesoc_info File List

- src

  Type: verilogSource
  - src/up_wishbone_classic.v

- tb

  Type: verilogSource
  - tb/tb_wishbone_slave.v

3.3 Targets

3.3.1 fusesoc_info Targets

- default

  Info: Default for IP intergration.
  - src
  - dep

- sim

  Info: Base simulation using icarus as default.
  - src
  - dep
  - tb

3.4 Directory Guide

Below highlights important folders from the root of the directory.

1. **docs** Contains all documentation related to this project.
    - **manual** Contains user manual and github page that are generated from the latex sources.
    - **specs** Contains specifications for the bus.
2. **src** Contains source files for the core
3. **tb** Contains test bench files for iverilog and cocotb
    - **cocotb** testbench files

4 Simulation

There are a few different simulations that can be run for this core.

4.1 iverilog

iverilog is used for simple test benches for quick verification, visually, of the core.

4.2 cocotb

Future simulations will use cocotb. This feature is not yet implemented.

5 Module Documentation

There is a single async module for this core.

- **up_wishbone_classic** Wishbone Classic to uP converter

The next sections document the module in great detail.

up_wishbone_classic.v

AUTHORS

JAY CONVERTINO

DATES

2024/03/01

INFORMATION

Brief

Wishbone Classic slave to uP interface

License MIT

up_wishbone_classic

```
module up_wishbone_classic #(
parameter
ADDRESS_WIDTH
 =
16,
parameter
BUS_WIDTH
 =
4
) ( input clk, input rst, input s_wb_cyc, input s_wb_stb, input s_wb_we, inp
```

Wishbone Classic slave to uP up_wishbone_classic

Parameters

ADDRESS_WIDTH parameter	Width of the APB3 address port in bits.
BUS_WIDTH parameter	Width of the APB3 bus data port in bytes.

Ports

clk	Clock
rst	Positive reset
s_wb_cyc	Bus Cycle in process
s_wb_stb	Valid data transfer cycle
s_wb_we	Active High write, low read
s_wb_addr	Bus address
s_wb_data_i	Input data
s_wb_sel	Device Select
s_wb_bte	Burst Type Extension
s_wb_cti	Cycle Type
s_wb_ack	Bus transaction terminated
s_wb_data_o	Output data
s_wb_err	Active high when a bus error is present
up_rreq	uP bus read request
up_rack	uP bus read ack
up_raddr	uP bus read address
up_rdata	uP bus read data
up_wreq	uP bus write request
up_wack	uP bus write ack
up_waddr	uP bus write address
up_wdata	uP bus write data

VARIABLES

s_next_address

```
assign s_next_address = wb_next_adr(
r_address,
r_wb_cti &
s_wb_cti,
s_wb_bte,
BUS_WIDTH *
8
)
```

Use the fusesoc wb_next_adr function to generate a address when wishbone classic is in a burst mode.

valid

```
assign valid = s_wb_cyc & s_wb_stb & ~r_rst[0]
```

Indicate valid request from wishbone.

up_rreq

```
assign up_rreq = ~s_wb_we & r_req
```

Convert wishbone read requests to up requests

up_wreq

```
assign up_wreq = s_wb_we & r_req
```

Convert wishbone write requests to up requests

s_wb_err

```
assign s_wb_err = 1'b0
```

TODO:check for burst address errors

up_raddr

```
assign up_raddr = (
                                                    ~s_wb_we & ~r_rst[0] ? (
address_state == init_address ? s_wb_addr : s_next_address)
:
0
)
```

assign address to read address port if selected

up_waddr

```
assign up_waddr = (
                                                    s_wb_we & ~r_rst[0] ? (
address_state == init_address ? s_wb_addr : r_address)
:
0
)
```

assign address to write address port if selected

up_ack

```
assign up_ack = (
up_rack |
up_wack
)
```

ack is ack for both, or them so either may pass

s_wb_ack

```
assign s_wb_ack = up_ack
```

combined uP ack is wishbone ack.