

UP_APB3



November 18, 2024

Jay Convertino

Contents

1 Usage	2
1.1 Introduction	2
1.2 Dependencies	2
1.2.1 fusesoc_info Depenecies	2
1.3 In a Project	2
2 Architecture	5
3 Building	5
3.1 fusesoc	5
3.2 Source Files	6
3.2.1 fusesoc_info File List	6
3.3 Targets	6
3.3.1 fusesoc_info Targets	6
3.4 Directory Guide	6
4 Simulation	7
4.1 iverilog	7
4.2 cocotb	7
5 Module Documentation	8
5.1 up_apb3	9

1 Usage

1.1 Introduction

This core converts the APB3 bus to the uP bus. This allows any core with a uP bus to be interfaced with a APB3 bus. These busses are very similar and is done with combinatoral logic only.

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

1.3 In a Project

This core is made to interface APB3 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 relatively 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
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_tx_delay[
            //FIFO_DEPTH-1:1]};

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

```

2 Architecture

The only module is the up_apb3 module. It is listed below.

- **up_apb3** Convert APB3 to the Analog Devices uP BUS. (see core for documentation).

This core only uses combinatorial methods to convert a few signals between the uP bus the APB3.

Please see 5 for more information.

3 Building

The APB3 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_apb3.v
- tb
 - Type: verilogSource
 - tb/tb_apb3.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_apb3** APB3 to uP converter

The next sections document the module in great detail.

up_apb3.v

AUTHORS

JAY CONVERTINO

DATES

2024/03/19

INFORMATION

Brief

APB3 slave to uP interface

License MIT

Copyright 2024 Jay Convertino

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

up_apb3

```
module up_apb3 #(
    parameter
    ADDRESS_WIDTH
    =
    16,
    parameter
    BUS_WIDTH
    =
    4
) ( input clk, input rst, input [ADDRESS_WIDTH-1:0] s_apb_paddr, input [0:0]
```

APB3 slave to uP interface

Parameters

ADDRESS_WIDTH Width of the APB3 address port in bits.

parameter

BUS_WIDTH Width of the APB3 bus data port in bytes.

parameter

Ports

clk Clock

rst Positive reset

s_apb_paddr APB3 address bus, up to 32 bits wide.

s_apb_psel APB3 select per slave (1 for this core).

s_apb_penable APB3 enable device for multiple transfers after first.

s_apb_pready APB3 ready is a output from the slave to indicate its able to process the request.

s_apb_pwrite APB3 Direction signal, active high is a write access. Active low is a read access.

s_apb_pwdata APB3 write data port.

s_apb_prdata APB3 read data port.

s_apb_pslverror APB3 error indicates transfer failure, not implimented.

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

valid

```
assign valid = s_apb_psel & s_apb_penable
```

This will add an extra clock cycle. since enable happens after select. both are needed to use the device.

s_apb_pslverror

```
assign s_apb_pslverror = 1'b0
```

APB3 error is always 0, no error.

up_waddr

```
assign up_waddr = s_apb_paddr
```

up_waddr and s_apb_addr are a direct mapping.

up_raddr

up_raddr and s_apb_addr are a direct mapping.

up_wdata

```
assign up_wdata = s_apb_pwdata
```

up_wdata and s_apb_pwdata are a direct mapping.

s_apb_prdata

```
assign s_apb_prdata = up_rdata
```

s_apb_prdata and up_rdata are a direct mapping.

up_wreq

```
assign up_wreq = valid & s_apb_pwrite
```

uP write request is a combination of the APB3 valid and APB3 write select (active high is write).

up_rreq

```
assign up_rreq = valid & ~s_apb_pwrite
```

uP read request is a combination of the APB3 valid and APB3 write select (active low is read).

s_apb_pready

```
assign s_apb_pready = up_wack | up_rack | ~valid
```

Diagrams seem to indicate that we should indicate ready when not sel and enable, which is why valid is complimented.