Altera’s Avalon Communication Fabric

Stephen A. Edwards

Columbia University

Spring 2022
Altera’s Avalon Bus

Something like “PCI on a chip”

Described in Altera’s *Avalon Memory-Mapped Interface Specification* document.

Protocol defined between peripherals and the “bus” (actually a fairly complicated circuit).
Intended System Architecture

Source: Altera
Masters and Slaves

Most bus protocols draw a distinction between

**Masters**: Can initiate a transaction, specify an address, etc. E.g., the Nios II processor

**Slaves**: Respond to requests from masters, can generate return data. E.g., a video controller

Most peripherals are slaves.

Masters speak a more complex protocol

Bus arbiter decides which master gains control
The Simplest Slave Peripheral

Avalon-MM Peripheral

- writedata[15..0]
- write
- chipselect
- clk
- D
- CLK_EN
- Q
- pio_out[15..0]

Basically, “latch when I’m selected and written to.”
Slave Signals

For a 16-bit connection that spans 32 halfwords,

<table>
<thead>
<tr>
<th>Slave</th>
<th>Avalon</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>clk</td>
</tr>
<tr>
<td>←</td>
<td>reset</td>
</tr>
<tr>
<td>←</td>
<td>chipselect</td>
</tr>
<tr>
<td>←</td>
<td>address[4:0]</td>
</tr>
<tr>
<td>←</td>
<td>read</td>
</tr>
<tr>
<td>←</td>
<td>write</td>
</tr>
<tr>
<td>←</td>
<td>byteenable[1:0]</td>
</tr>
<tr>
<td>←</td>
<td>writedata[15:0]</td>
</tr>
<tr>
<td>←</td>
<td>readdata[15:0]</td>
</tr>
<tr>
<td>irq</td>
<td>⇒</td>
</tr>
<tr>
<td>irq</td>
<td>→</td>
</tr>
</tbody>
</table>
### Avalon Slave Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Master clock</td>
</tr>
<tr>
<td>reset</td>
<td>Reset signal to peripheral</td>
</tr>
<tr>
<td>chipselect</td>
<td>Asserted when bus accesses peripheral</td>
</tr>
<tr>
<td>address[..]</td>
<td>Word address (data-width specific)</td>
</tr>
<tr>
<td>read</td>
<td>Asserted during peripheral→bus transfer</td>
</tr>
<tr>
<td>write</td>
<td>Asserted during bus→peripheral transfer</td>
</tr>
<tr>
<td>writedata[..]</td>
<td>Data from bus to peripheral</td>
</tr>
<tr>
<td>byteenable[..]</td>
<td>Indicates active bytes in a transfer</td>
</tr>
<tr>
<td>readdata[..]</td>
<td>Data from peripheral to bus</td>
</tr>
<tr>
<td>irq</td>
<td>peripheral→processor interrupt request</td>
</tr>
</tbody>
</table>

All are optional, as are many others for, e.g., flow-control and burst transfers.
In SystemVerilog

module myslave(input logic clk,
               input logic reset,
               input logic [7:0] writedata,
               input logic write,
               input logic chipselect,
               input logic [2:0] address);
Basic Slave Read Transfer

Bus cycle starts on rising clock edge
Data latched at next rising edge
Such a peripheral must be purely combinational
Slave Read Transfer w/ 1 Wait State

Bus cycle starts on rising clock edge
Data latched two cycles later
Approach used for synchronous peripherals
Basic Async. Slave Write Transfer

Bus cycle starts on rising clock edge
Data available by next rising edge
Peripheral may be synchronous, but must be fast
Basic Async. Slave Write w/ 1 Wait State

Bus cycle starts on rising clock edge
Peripheral latches data two cycles later
For slower peripherals
The Vga_ball Peripheral

module vga_ball(input logic clk,
                 input logic reset,
                 input logic [7:0] writedata,
                 input logic write,
                 input chipselect,
                 input logic [2:0] address,
                 output logic [7:0] VGA_R, VGA_G, VGA_B,
                 output logic VGA_CLK, VGA_HS, VGA_VS,
                 output logic VGA_BLANK_n,
                 output logic VGA_SYNC_n);

logic [10:0] hcount;
logic [9:0] vcount;

logic [7:0] background_r, background_g, background_b;

vga_counters counters(.clk50(clk), .*);
The Vga_ball Peripheral

```verilog
always_ff @(posedge clk)
  if (reset) begin
    background_r <= 8'h0;
    background_g <= 8'h0;
    background_b <= 8'h80;
  end else if (chipselect && write)
    case (address)
      3'h0 : background_r <= writedata;
      3'h1 : background_g <= writedata;
      3'h2 : background_b <= writedata;
    endcase

always_comb begin
  {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
  if (VGA_BLANK_n )
    if (hcount[10:6] == 5'd3 &&
      vcount[9:5] == 5'd3)
      {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff};
    else
      {VGA_R, VGA_G, VGA_B} =
        {background_r, background_g, background_b};
  end
```