Altera’s Avalon Communication Fabric

Stephen A. Edwards

Columbia University

Spring 2023
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

Avalon-MM System

- Processor
  - 32-bit Avalon-MM Master Port
- Ethernet MAC
  - 32-bit Avalon-MM Master Port
- Custom Logic
  - 64-bit Avalon-MM Master Port

System Interconnect Fabric

- 32-bit Avalon-MM Slave Port
- 16-bit Avalon-MM Slave Port
- 64-bit Avalon-MM Slave Port

- SDRAM Controller
- UART
- Custom Logic

- 8-bit Avalon-MM Tristate Slave Port
- 16-bit Avalon-MM Tristate Slave Port

- Flash Memory Chip
- SDRAM Memory Chip
- RS-232

Source: Altera
Controller and Responder

Most bus protocols draw a distinction between

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

Responders: Respond to requests from controllers, can generate return data. E.g., a video controller

Most peripherals are responders

Controllers speak a more complex protocol

Bus arbiter decides which controller gains control
The Simplest Responder Peripheral

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

For a 16-bit connection that spans 32 halfwords,

\[
\text{Responder} \quad \leftarrow \quad \begin{align*}
&\text{clk} \\
&\text{reset} \\
&\text{chipselect} \\
&\text{address}[4:0] \\
&\text{read} \\
&\text{write} \\
&\text{byteenable}[1:0] \\
&\text{writedata}[15:0] \\
&\text{readdata}[15:0] \\
&\text{irq}
\end{align*} \quad \rightarrow \quad \text{Avalon}
\]
Avalon Responder Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>Main 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

```verilog
module myresponder(
    input logic clk,
    input logic reset,
    input logic [7:0] writedata,
    input logic write,
    input logic chipselect,
    input logic [2:0] address);
```
Basic Responder Read Transfer

Bus cycle starts on rising clock edge
Data latched at next rising edge
Such a peripheral must be purely combinational
Responder 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. Responder 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. Responder Write w/ 1 Wait State

Bus cycle starts on rising clock edge
Peripheral latches data two cycles later
For slower peripherals
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), .*);
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