Altera’s Avalon Communication Fabric

Stephen A. Edwards

Columbia University

Spring 2014
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
  - SDRAM Controller
- 16-bit Avalon-MM Slave Port
  - UART
- 64-bit Avalon-MM Slave Port
  - Custom Logic

- 8-bit Avalon-MM Tristate Slave Port
  - Flash Memory Chip
- 16-bit Avalon-MM Tristate Slave Port
  - SRAM Memory Chip

SDRAM Memory Chip

RS-232

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

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

For a 16-bit connection that spans 32 halfwords,

\[
\begin{align*}
\text{ clk} & \quad \text{ reset} \\ 
\text{ chipselect} & \quad \text{ address}[4:0] \\ 
\text{ read} & \quad \text{ write} \\ 
\text{ byteenable}[1:0] & \quad \text{ writedata}[15:0] \\ 
\text{ readdata}[15:0] & \\ 
\text{ irq} & \quad \rightarrow
\end{align*}
\]
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.
module myslave(
    input logic clk,
    input logic reset,
    input logic [7:0] writedata,
    input logic write,
    input chipselect,
    input logic [2:0] address);

Bus cycle starts on rising clock edge.
Data latched at next rising edge.
Such a peripheral must be purely combinational.
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_LED Emulator Peripheral

```verilog
module VGA_LED(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, VGA_SYNC_n);

logic [7:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;

VGA_LED_Emulator led_emulator(.clk50(clk), .*);
```
The VGA_LED Emulator Peripheral

```verilog
always_ff @(posedge clk)
    if (reset) begin
        hex0 <= 8'b01100110; // 4
        hex1 <= 8'b01111111; // 8
        hex2 <= 8'b01100110; // 4
        hex3 <= 8'b10111111; // 0
        hex4 <= 8'b00111000; // L
        hex5 <= 8'b01110111; // A
        hex6 <= 8'b01111100; // b
        hex7 <= 8'b01001111; // 3
    end else if (chipselect && write)
        case (address)
            3'h0 : hex0 <= writedata;
            3'h1 : hex1 <= writedata;
            3'h2 : hex2 <= writedata;
            3'h3 : hex3 <= writedata;
            3'h4 : hex4 <= writedata;
            3'h5 : hex5 <= writedata;
            3'h6 : hex6 <= writedata;
            3'h7 : hex7 <= writedata;
        endcase
endmodule
```