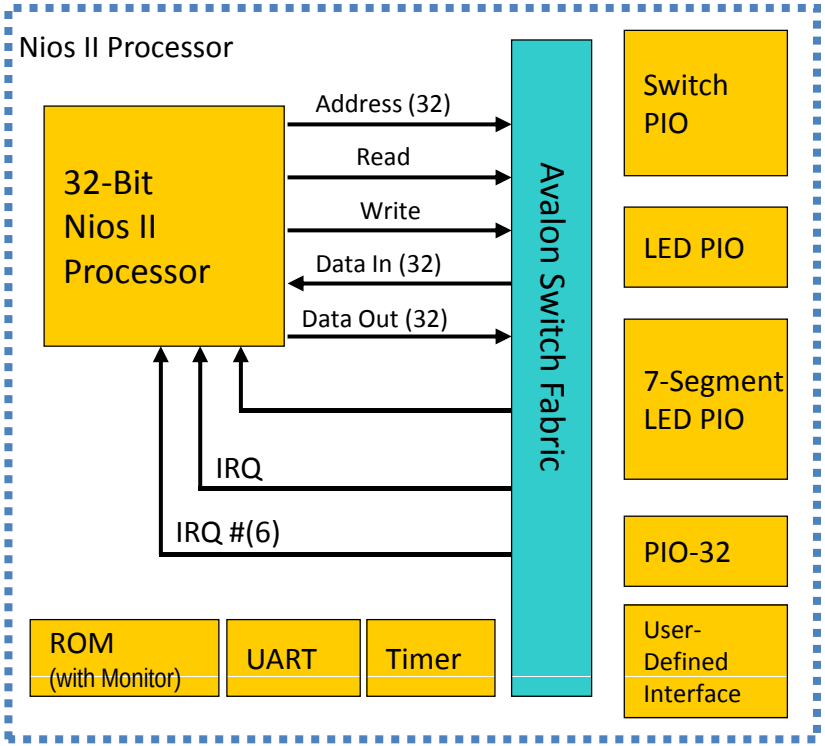
Avalon Switch Fabric

Avalon Switch Fabric

- Proprietary interconnect specification used with Nios II
- Principal design goals
  - Low resource utilization for bus logic
  - Simplicity
  - Synchronous operation
- Transfer Types
  - Slave Transfers
  - Master Transfers
  - Streaming Transfers
  - Latency-Aware Transfers
  - Burst Transfers



Avalon Switch Fabric

- Custom-Generated for Peripherals
  - Contingencies are on a Per-Peripheral Basis
  - System is Not Burdened by Bus Complexity
- SOPC Builder Automatically Generates
  - Arbitration
  - Address Decoding
  - Data Path Multiplexing
  - Bus Sizing
  - Wait-State Generation
  - Interrupts

Avalon Master Ports

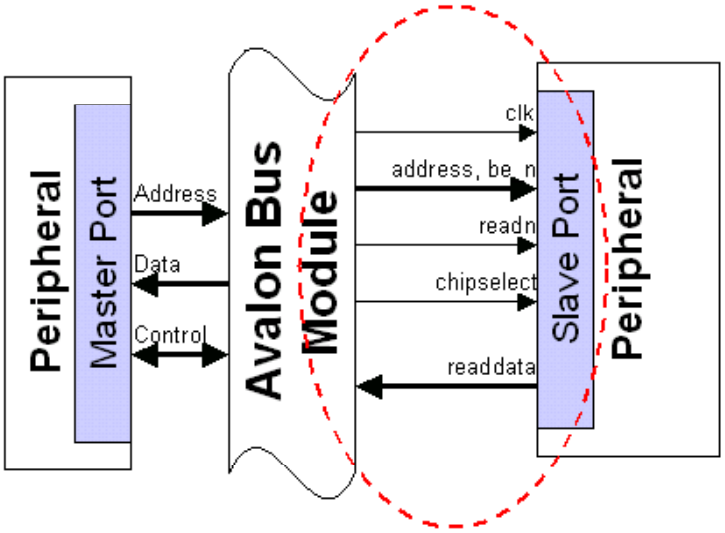
- Initiate Transfers with Avalon Switch Fabric
- Transfer Types
  - Fundamental Read
  - Fundamental Write
- All Avalon Masters Must Honor a waitrequest signal
- Transfer Properties
  - Latency
  - Streaming
  - Burst

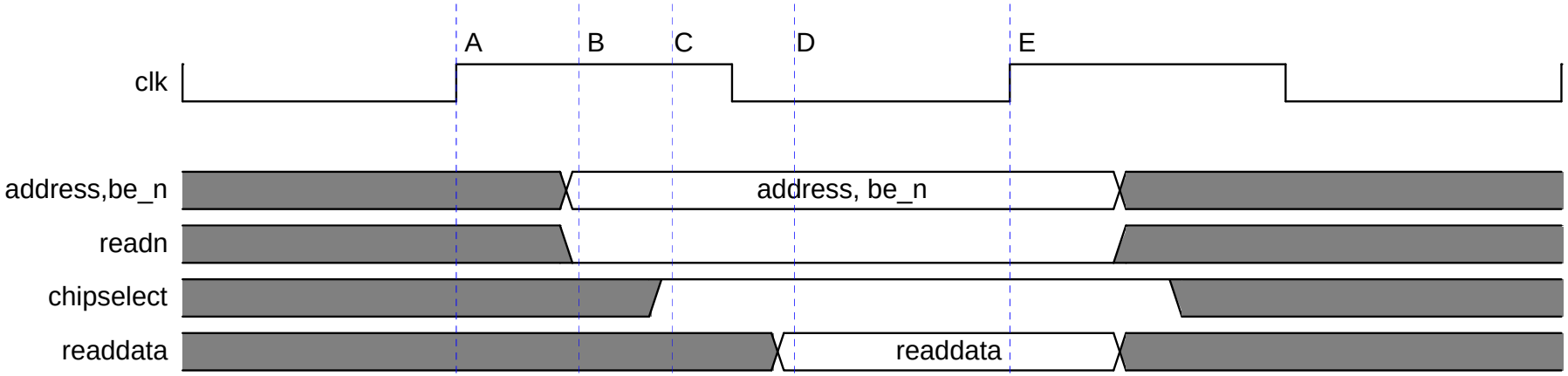
Avalon Slave Ports

- Respond to Transfer Requests from Avalon Switch Fabric
- Transfer Types
  - Fundamental Read
  - Fundamental Write
- Transfer Properties
  - Wait States
  - Latency
  - Streaming
  - Burst

Slave Read Transfer

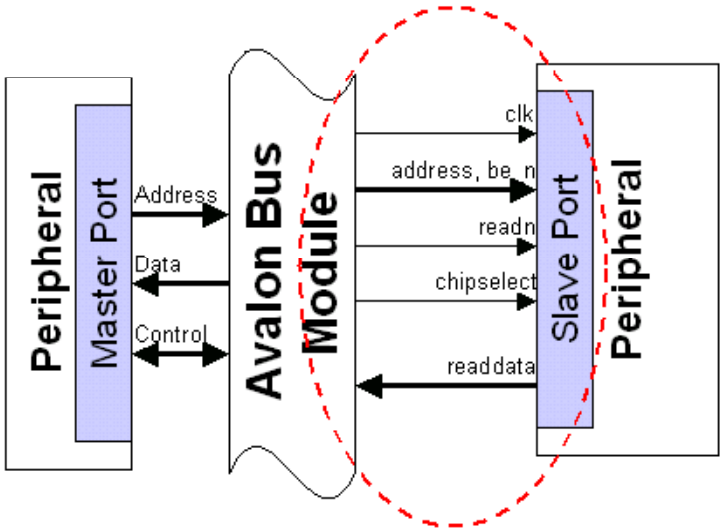
- 0 Setup Cycles
- 0 Wait Cycles

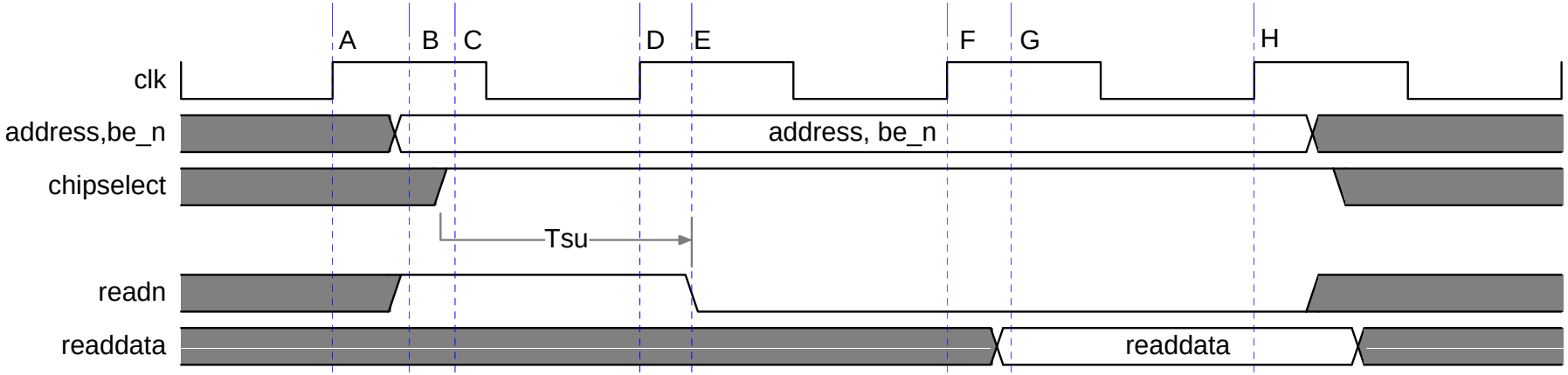




Slave Read Transfer with Wait States

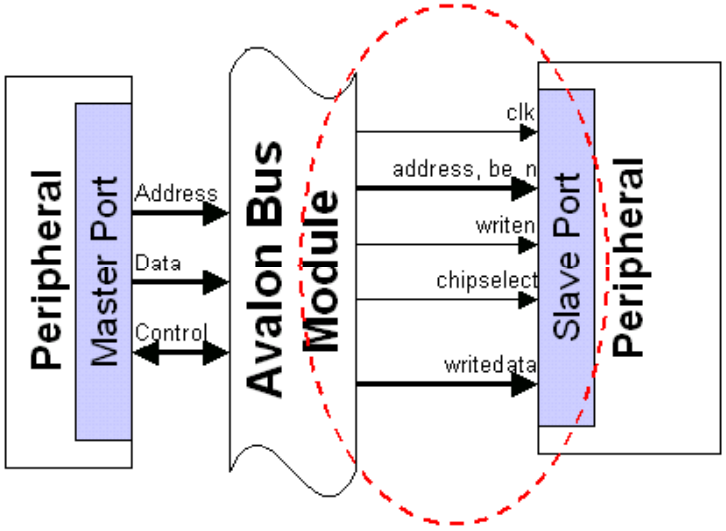
- 1 Setup Cycle
- 1 Wait Cycle

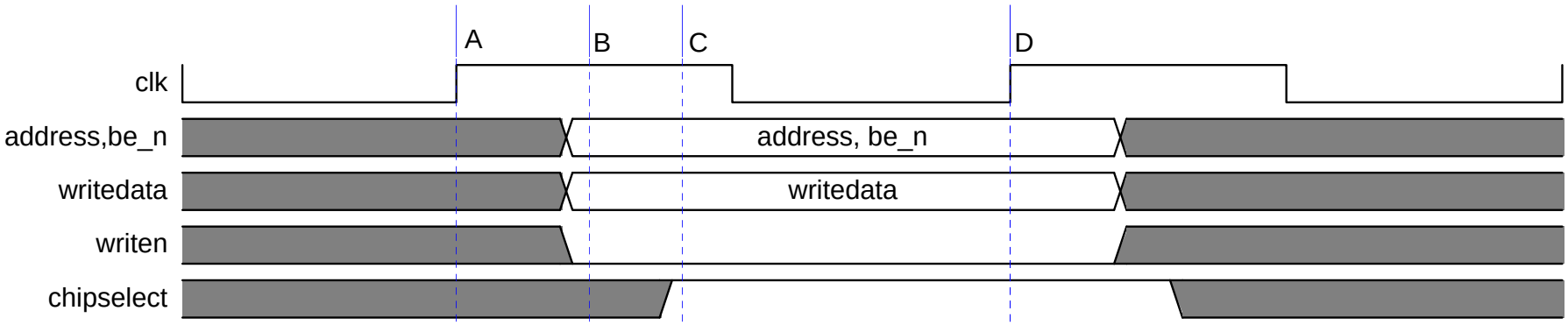




Slave Write Transfer

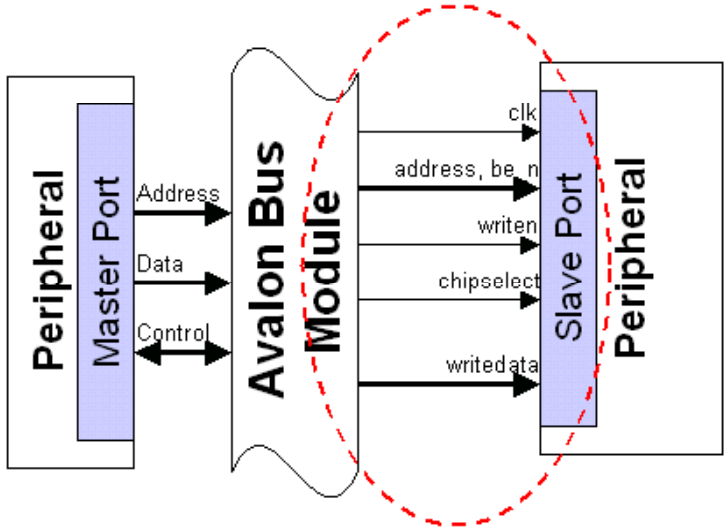
- 0 Setup Cycles
- 0 Wait Cycles
- 0 Hold Cycles

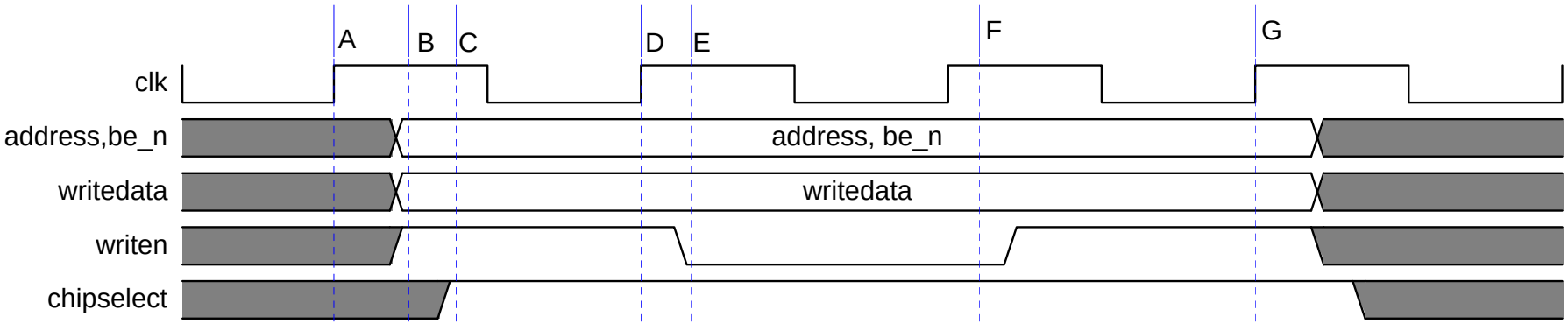




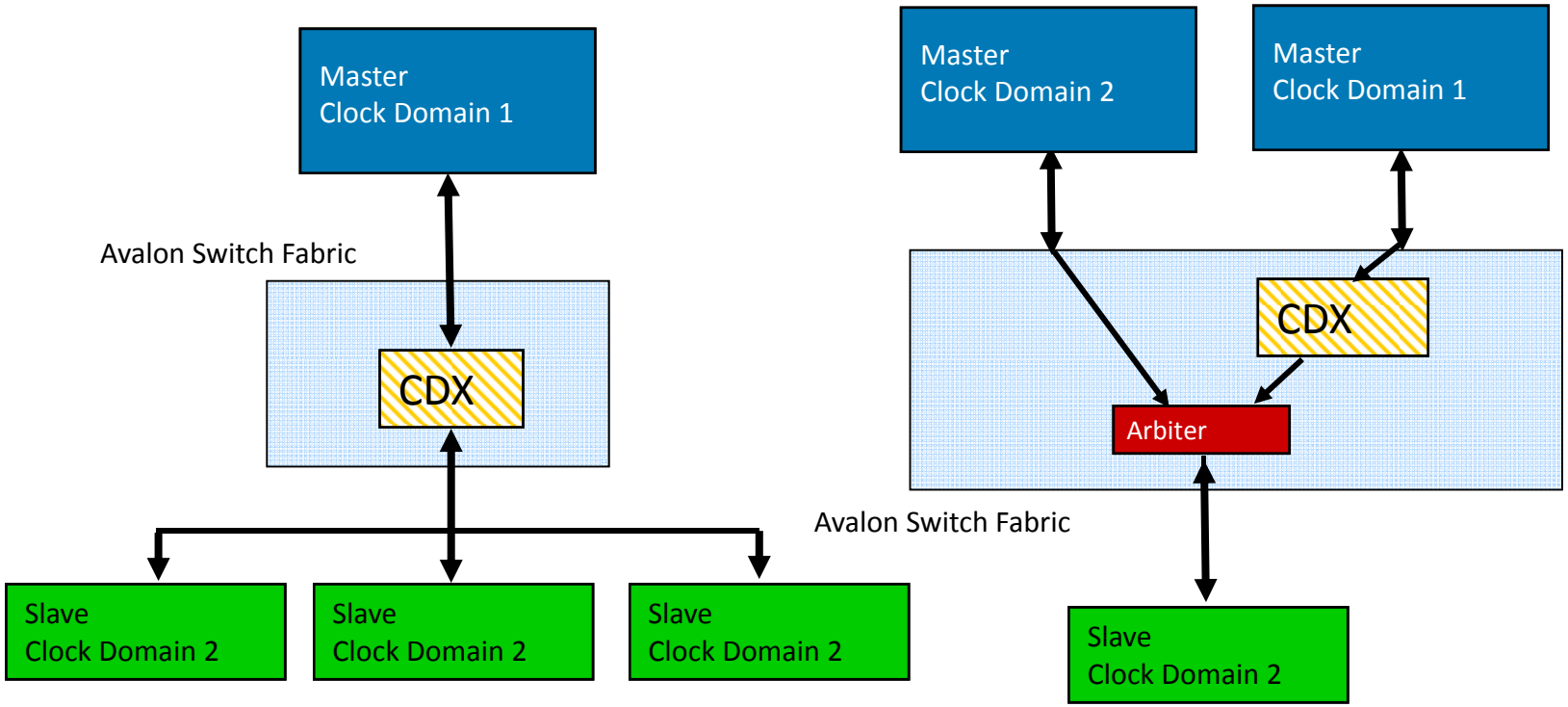
Slave Write Transfer with Wait States

- 1 Setup Cycle
- 0 Wait Cycles
- 1 Hold Cycle



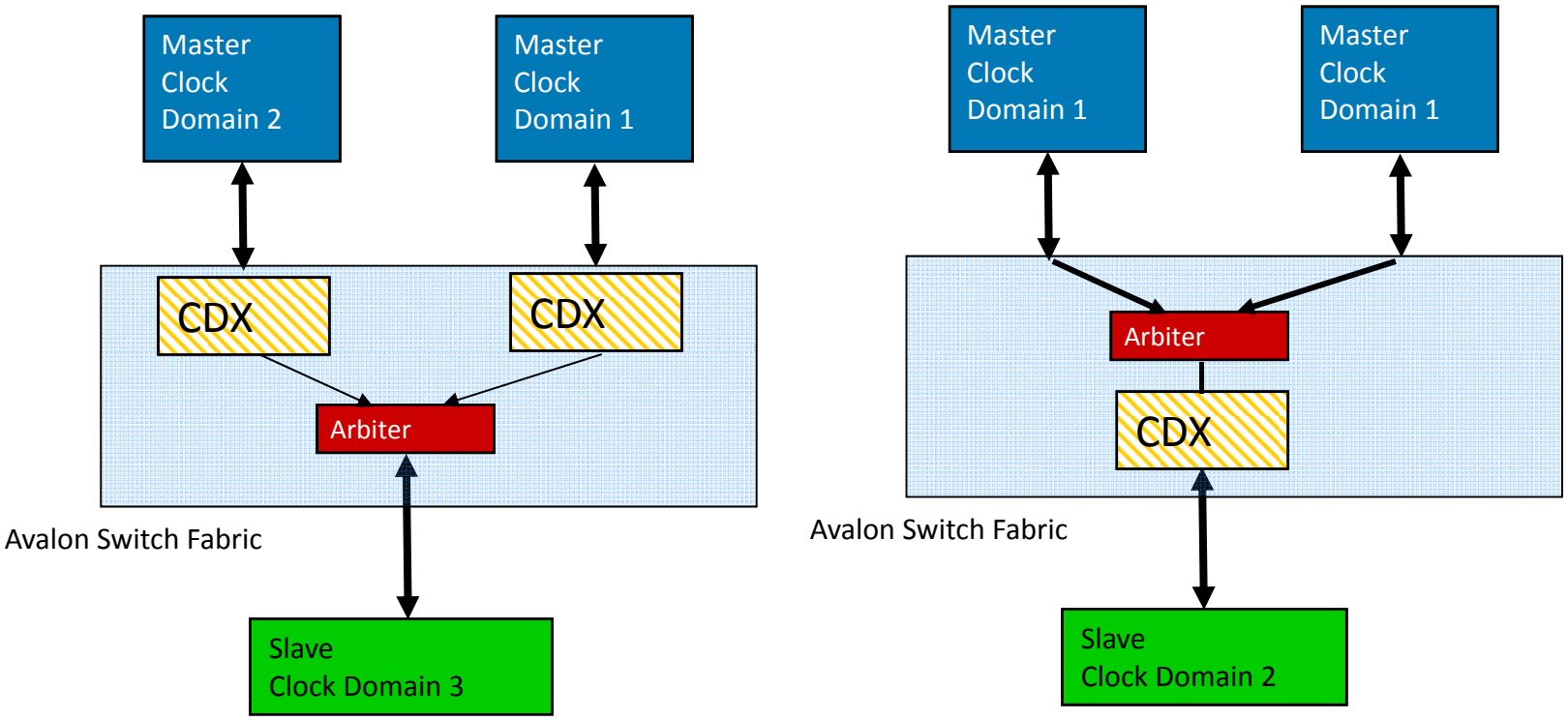


Multiple Clock Domains Supported



CDX = Clock Domain Crossing Logic (inserted automatically by SOPC Builder)

Multi-Clock Domain Support



CDX = Clock Domain Crossing Logic

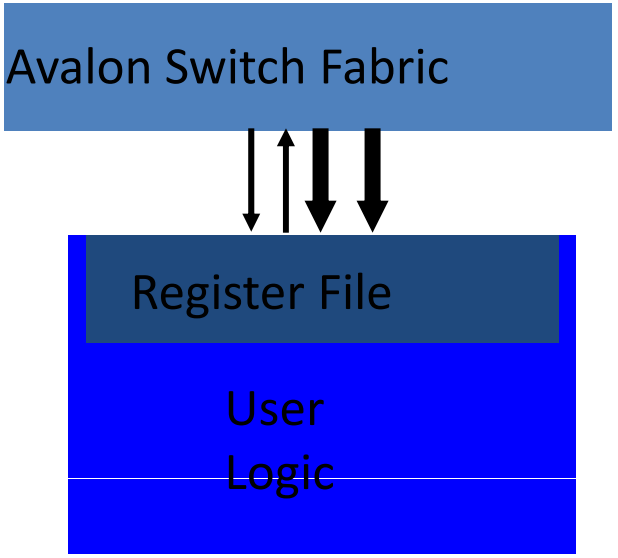
User-Defined Custom Peripherals

- What if I need to add a peripheral not included with the Nios II system?
  - user wants to add own peripheral to perform some kind of proprietary function or perhaps a standard function that is not yet included as part of the Nios kit
  - Expand or accelerate system capabilities

- We are now going learn how to connect our own design directly to the Nios II system via Avalon
  - As many peripherals contain registers we could also have chosen to connect to a PIO rather than directly to the bus

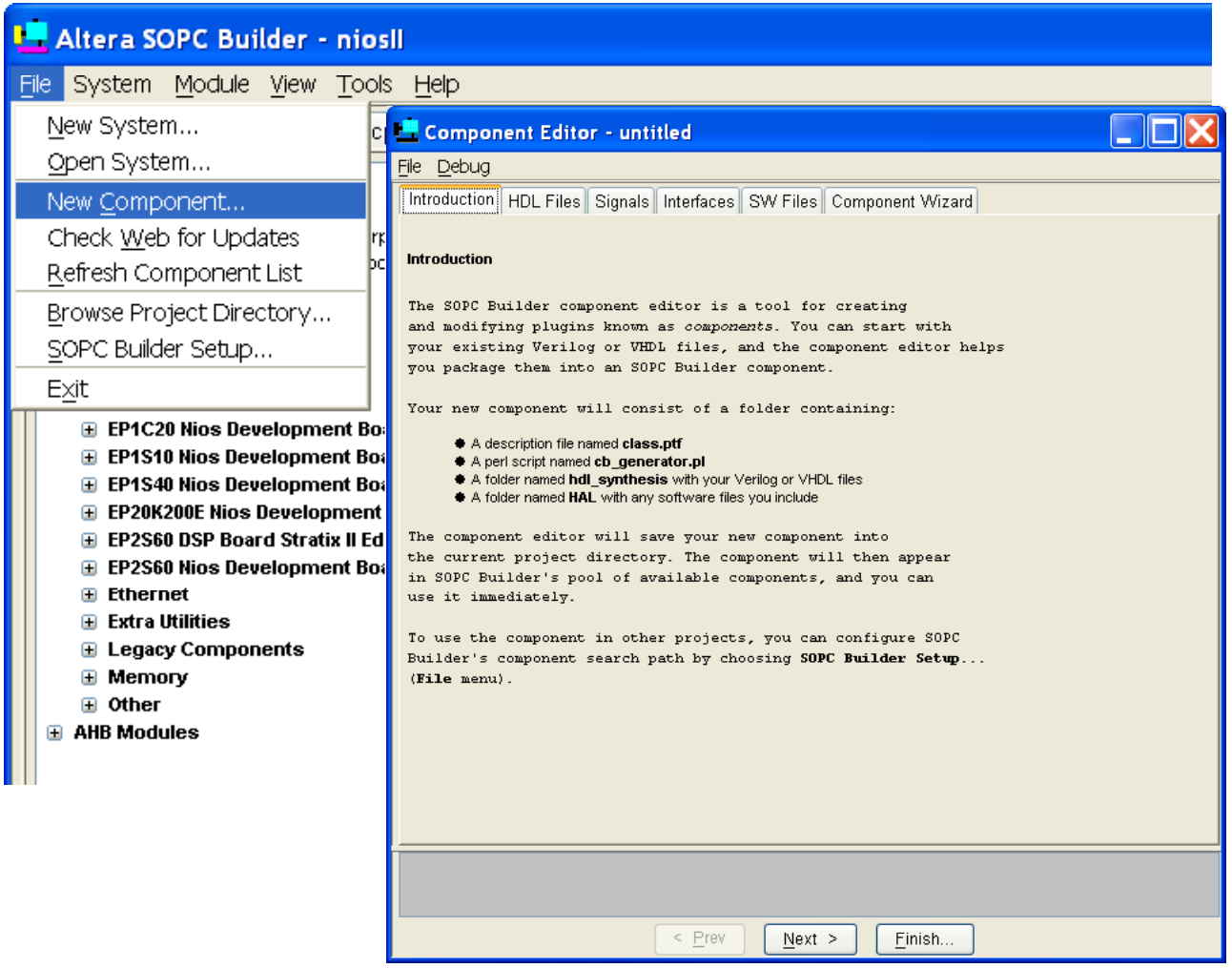
Creating Avalon Slave

- No Need to Worry about Bus Interface
- Implement Only Signals Needed
- Peripherals Adapted to by Avalon Switch Fabric
- Timing Handled Automatically
- Fabric Created for You
- Arbiters Generated for You



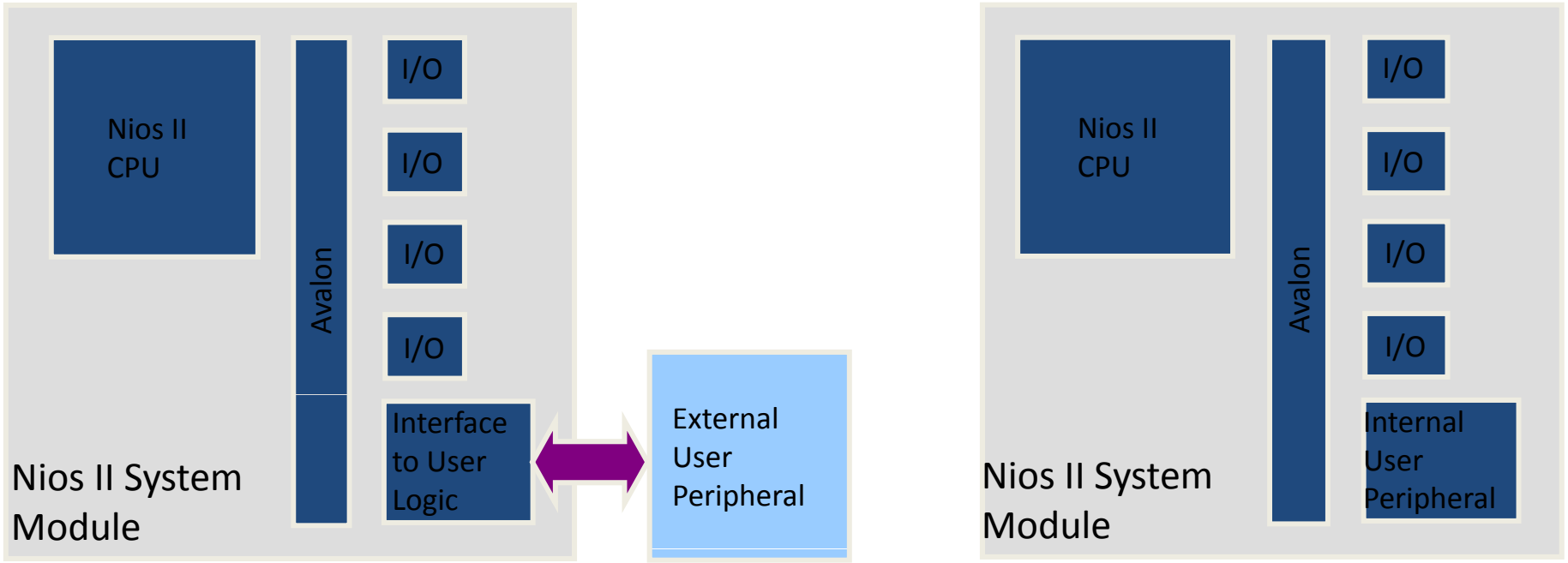
Concentrate Effort on Peripheral Functionality!

New Component Editor



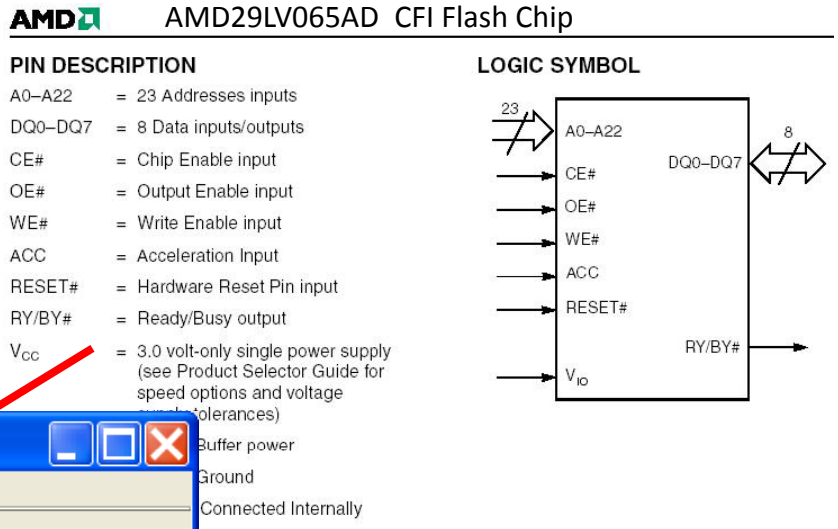
Creates Interface

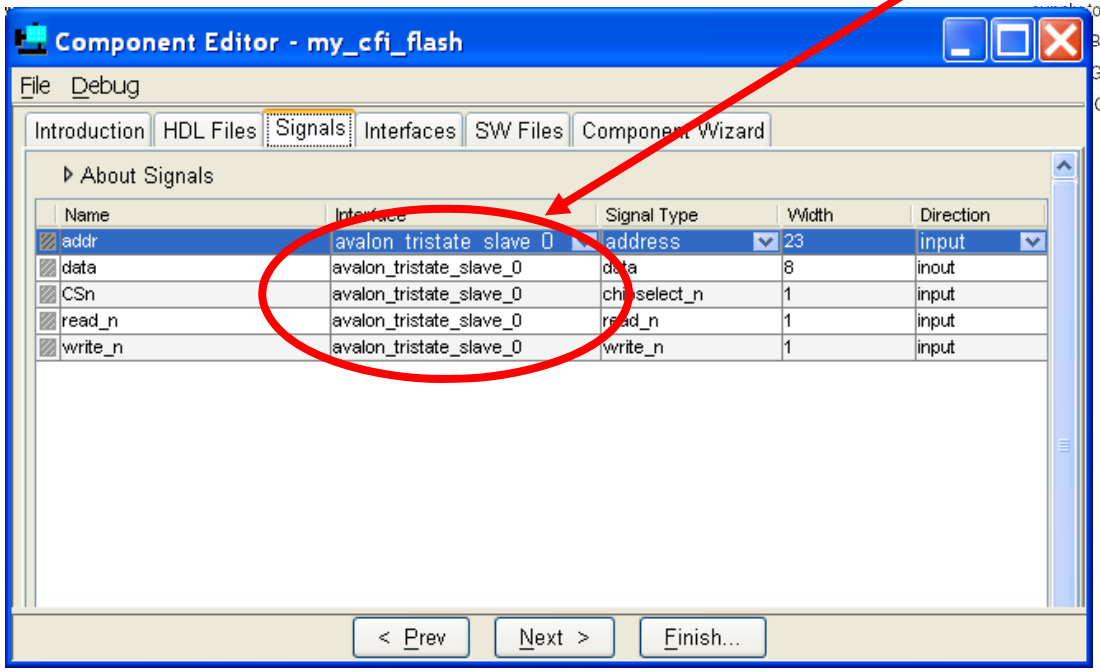
- Connect to Existing HDL or board component
- Map into Nios II Memory Space
- Can be “Inside” or “Outside” Nios II System



Create External Component Interface

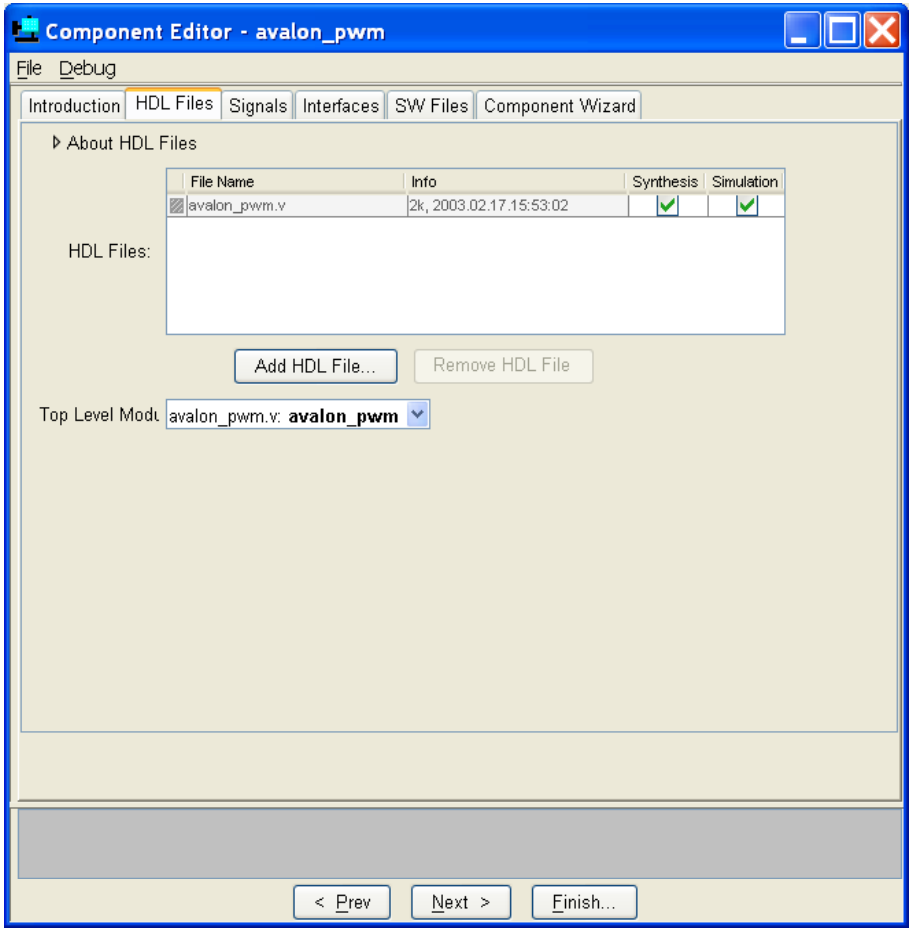
- To communicate with off-chip peripherals
- Base interface type on data sheet





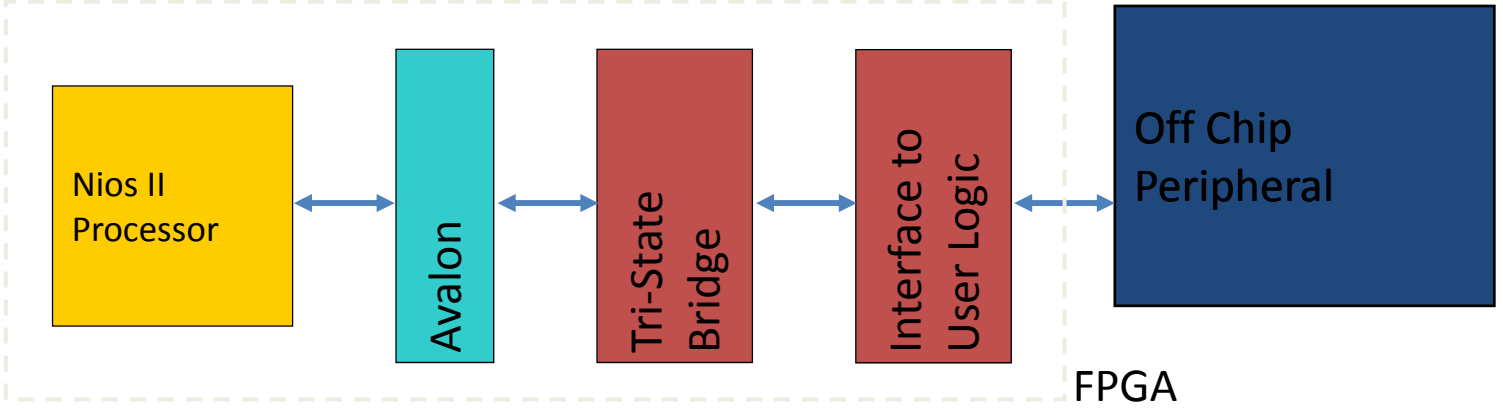
Or Add HDL Files

- For peripheral that has been encoded for FPGA



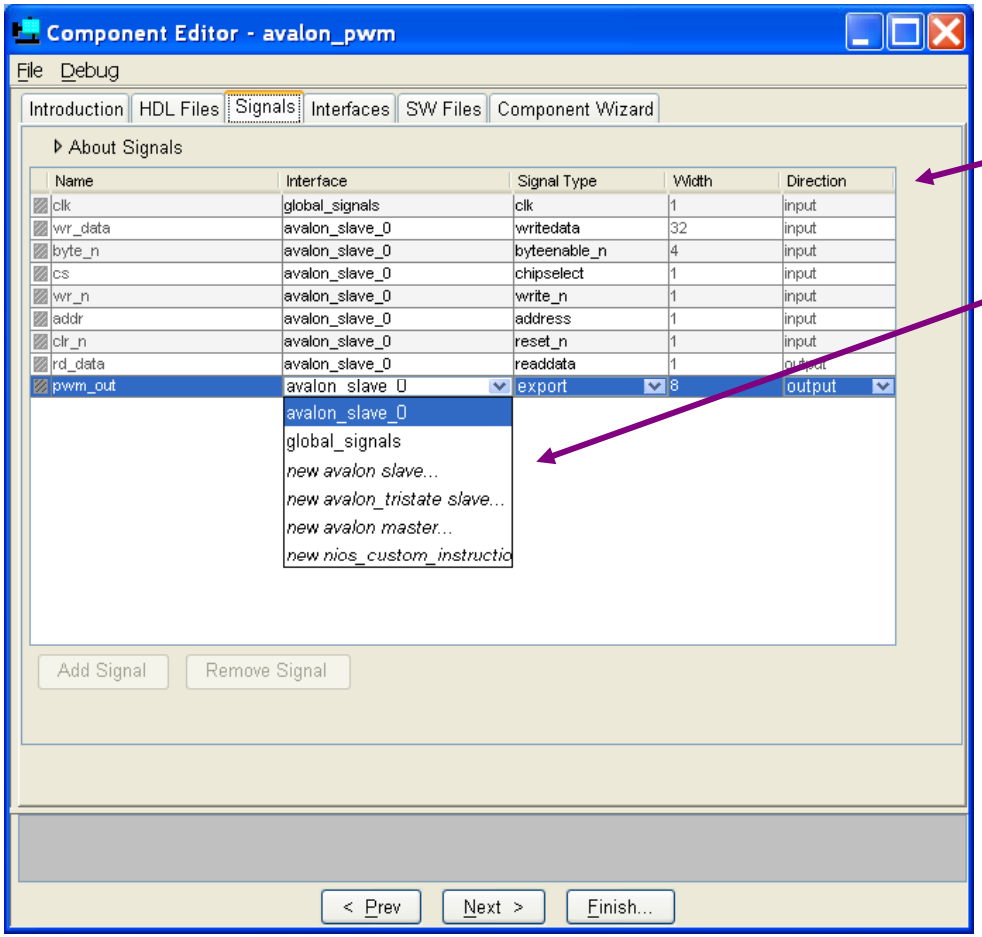
Tri-State Peripherals

- Require Tri-State Bridge
  - Available as an SOPC Builder component



- Tri-State peripheral is defined by the presence of a bi-direction data port
- Off-chip peripherals do not **have** to be tri-state

Define Component Signals

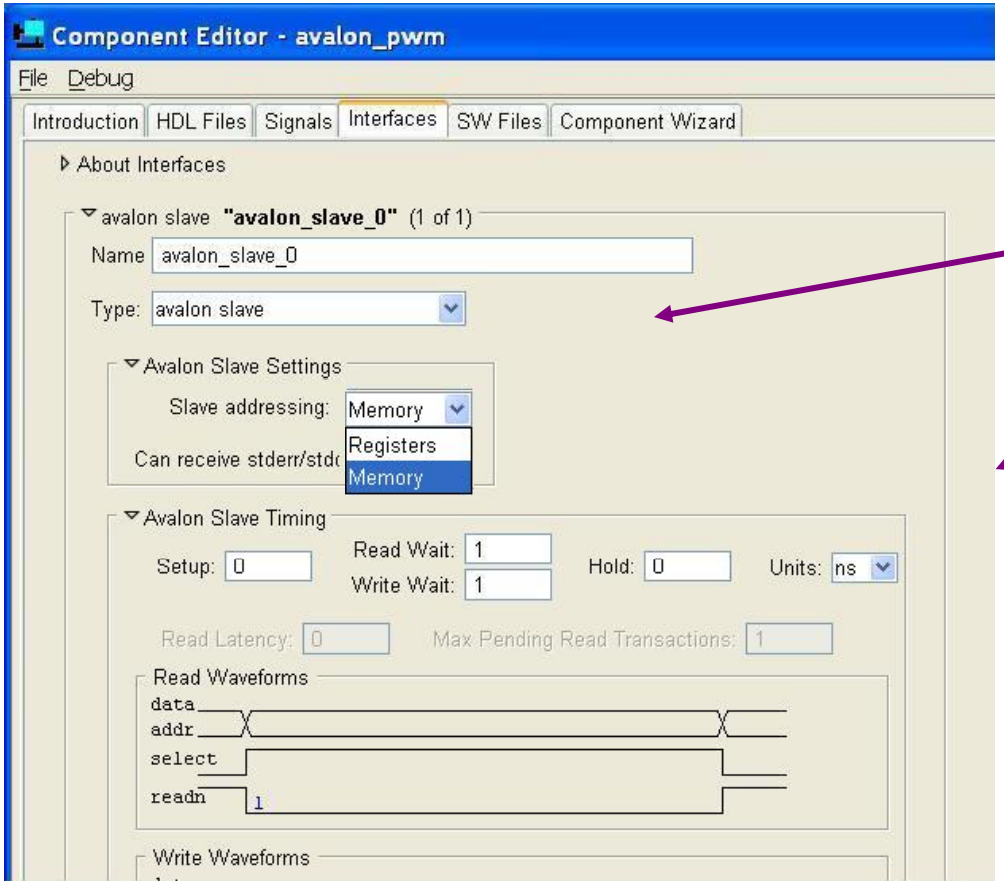


Automatically populates port table from design files

Enter port type here

Can also define ports manually

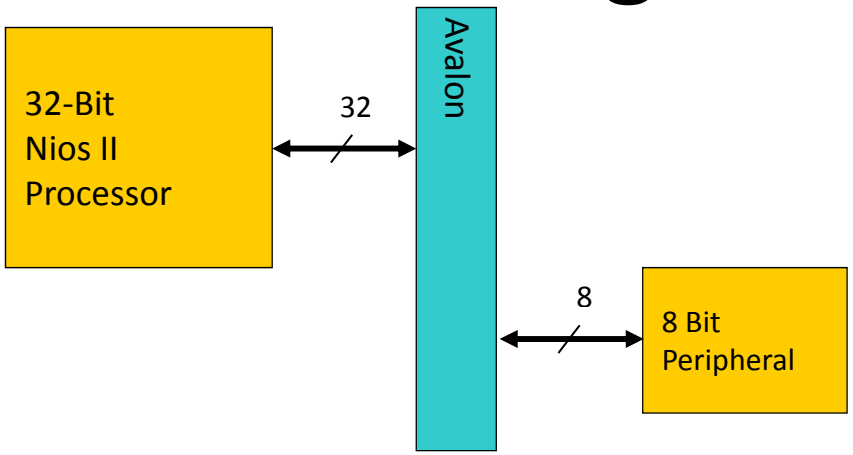
Define Interface for Each Signal Type



Choose interface type Register
- Slave uses native alignment,
- Memory Slave uses dynamic alignment

Control Read and Write Timing
- Add wait and hold states View waveforms

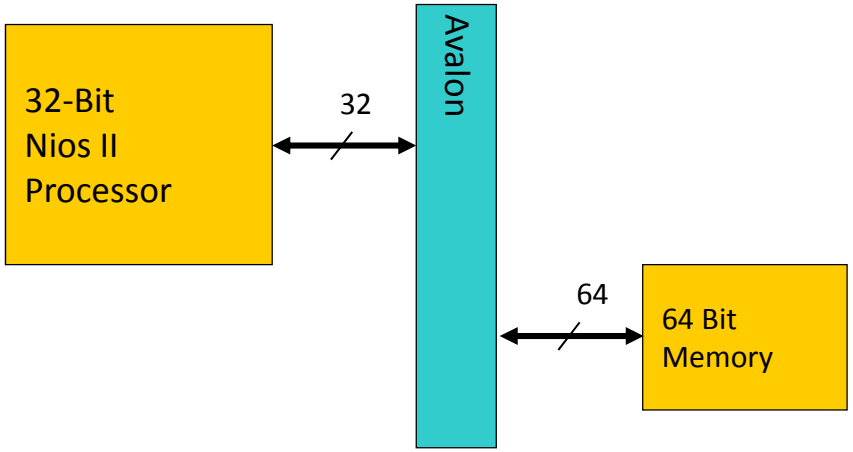
Address Alignment – Narrow Slave



Peripheral Registers	
Base	aa
Base + 0x1	bb
Base + 0x2	cc
Base + 0x3	dd
Base + 0x4	ee

- **Dynamic Address Alignment** (set as **Memory Slave**)
  - LD from Base + 0x0: dd cc bb aa
  - LD from Base + 0x4: uu uu uu ee
- **Native Address Alignment** (set as **Avalon Register Slave**)
  - LD from Base + 0x0: uu uu uu aa
  - LD from Base + 0x4: uu uu uu bb
  - LD from Base + 0x8: uu uu uu cc

Address Alignment – Narrow Master

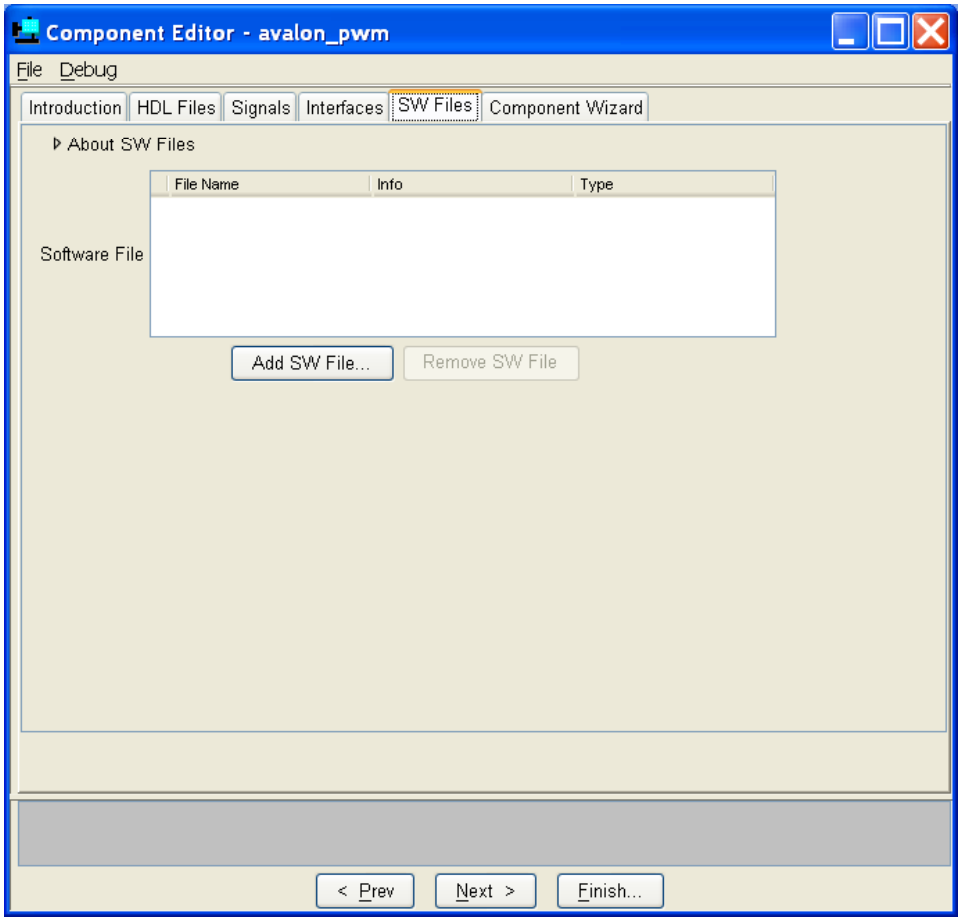


Memory Contents	
Base	77 66 55 44 33 22 11 00
Base + 0x8	ff ee dd cc bb aa 99 88
Base + 0x16	?? ?? ?? ?? ?? ?? ?? ??

- **Dynamic Address Alignment**
  - LD from Base + 0x0: 33 22 11 00
  - LD from Base + 0x4: 77 66 55 44
  - LD from Base + 0x8: bb aa 99 88
- **Native Address Alignment**
  - LD from Base + 0x0: 33 22 11 00
  - LD from Base + 0x4: bb aa 99 88
  - LD from Base + 0x8: ?? ?? ?? ??
  - *High bytes are unobtainable – warning issued*

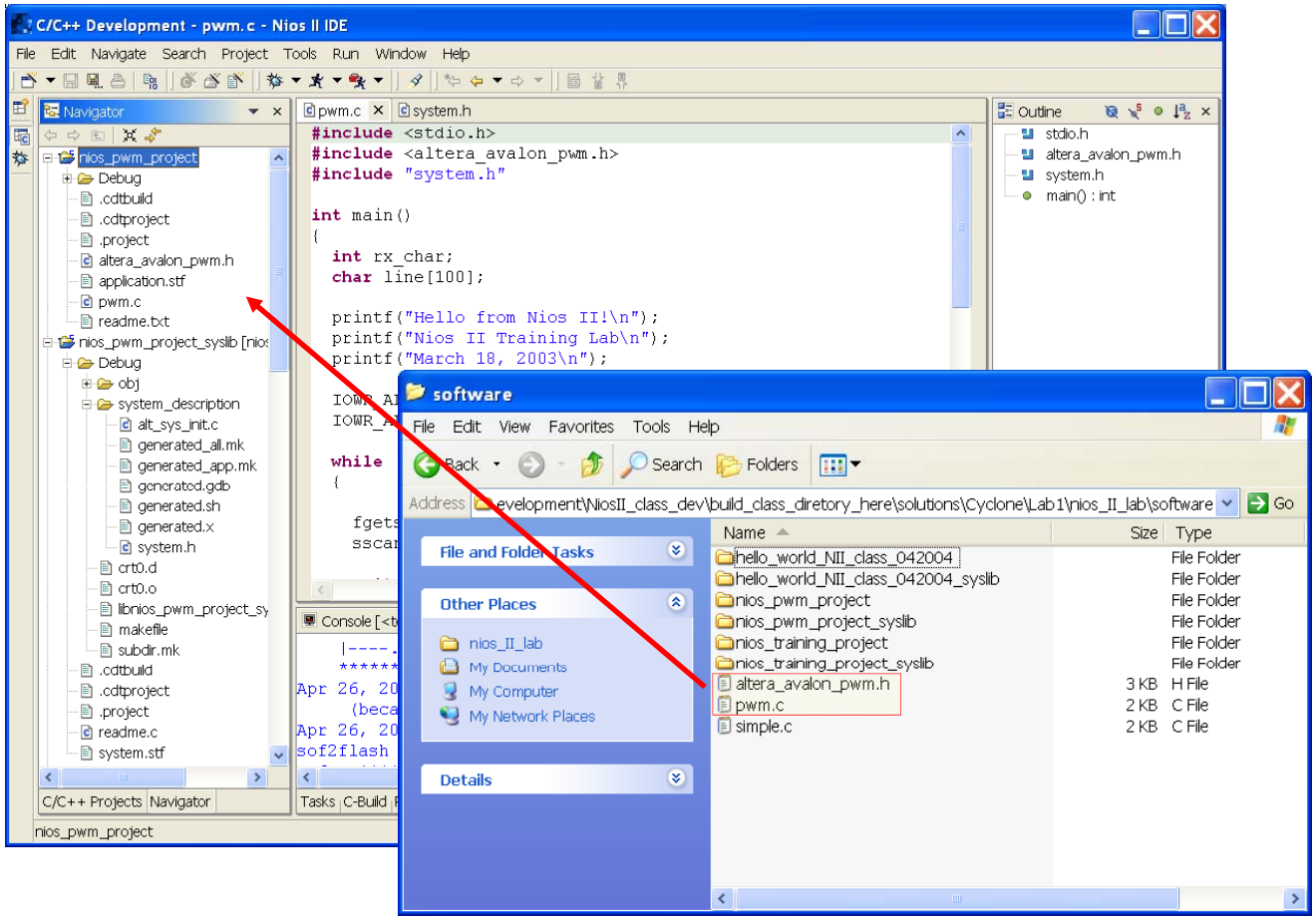
Add Software Files

- ie. Header files and drivers



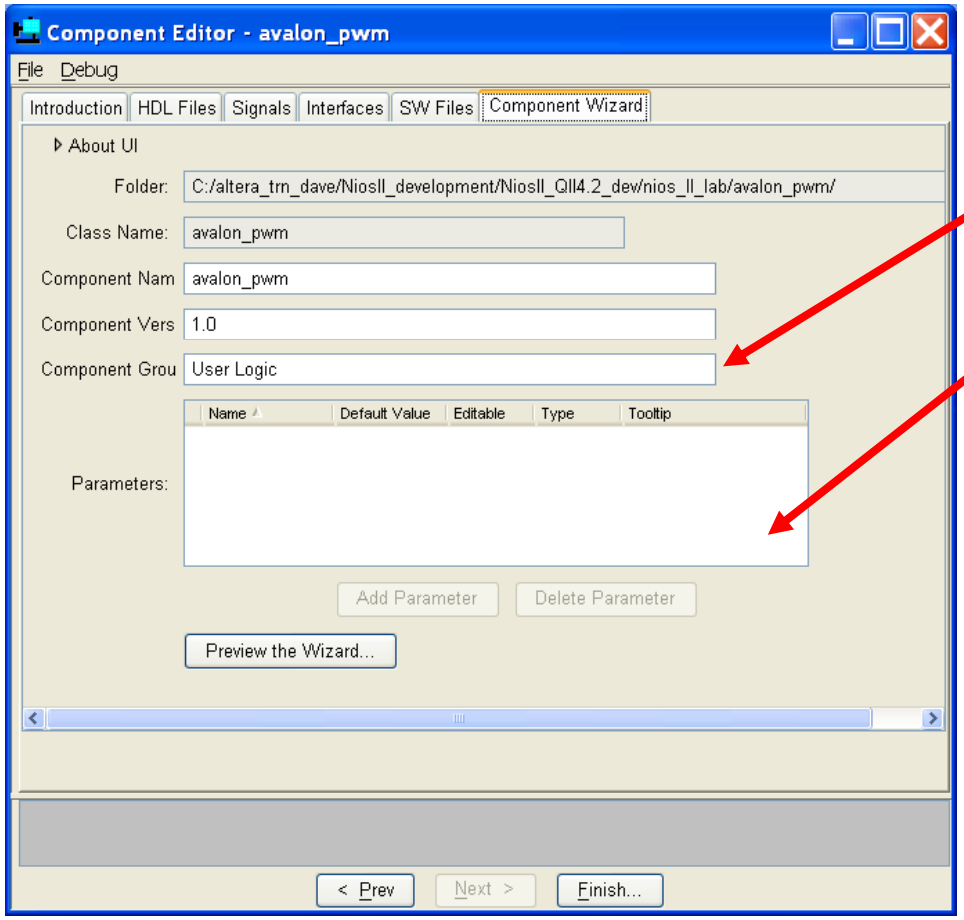
Add Software Files

- Header file and drivers can also be added directly to Application Project



Create Component Wizard

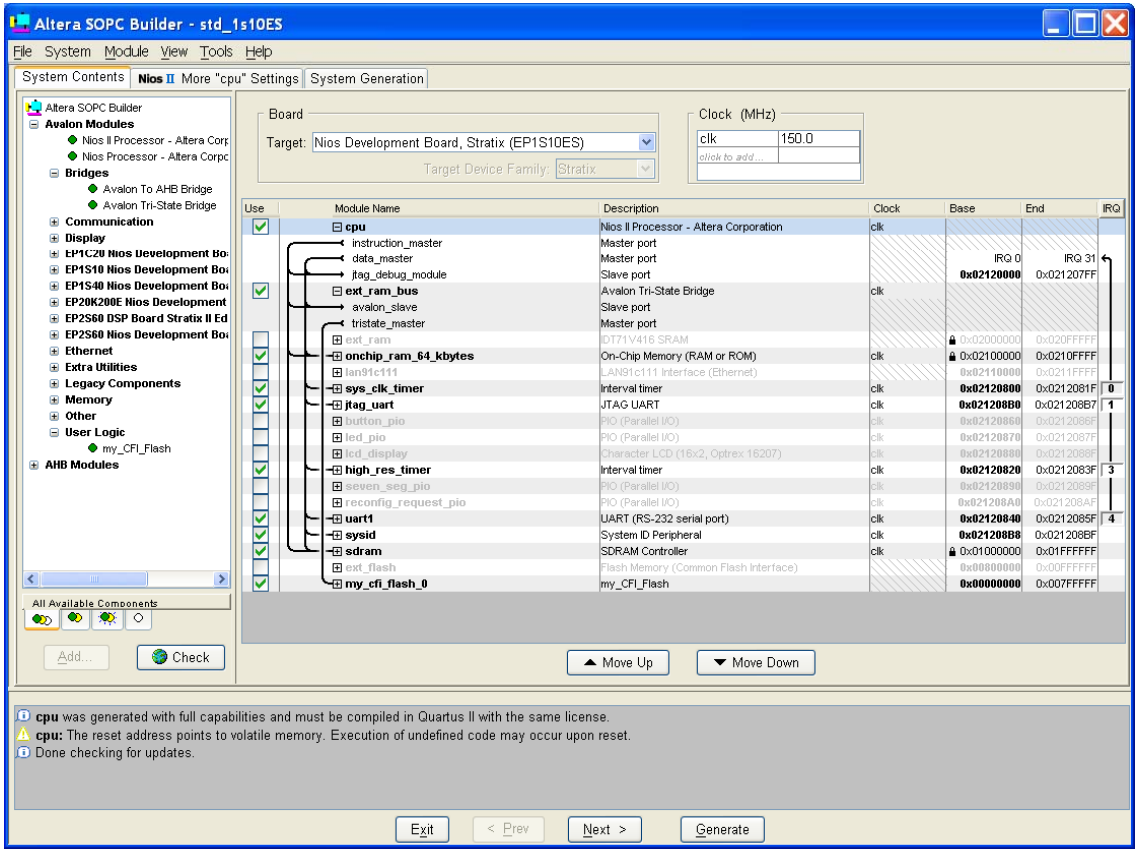
- Publish and create a wizard for your component



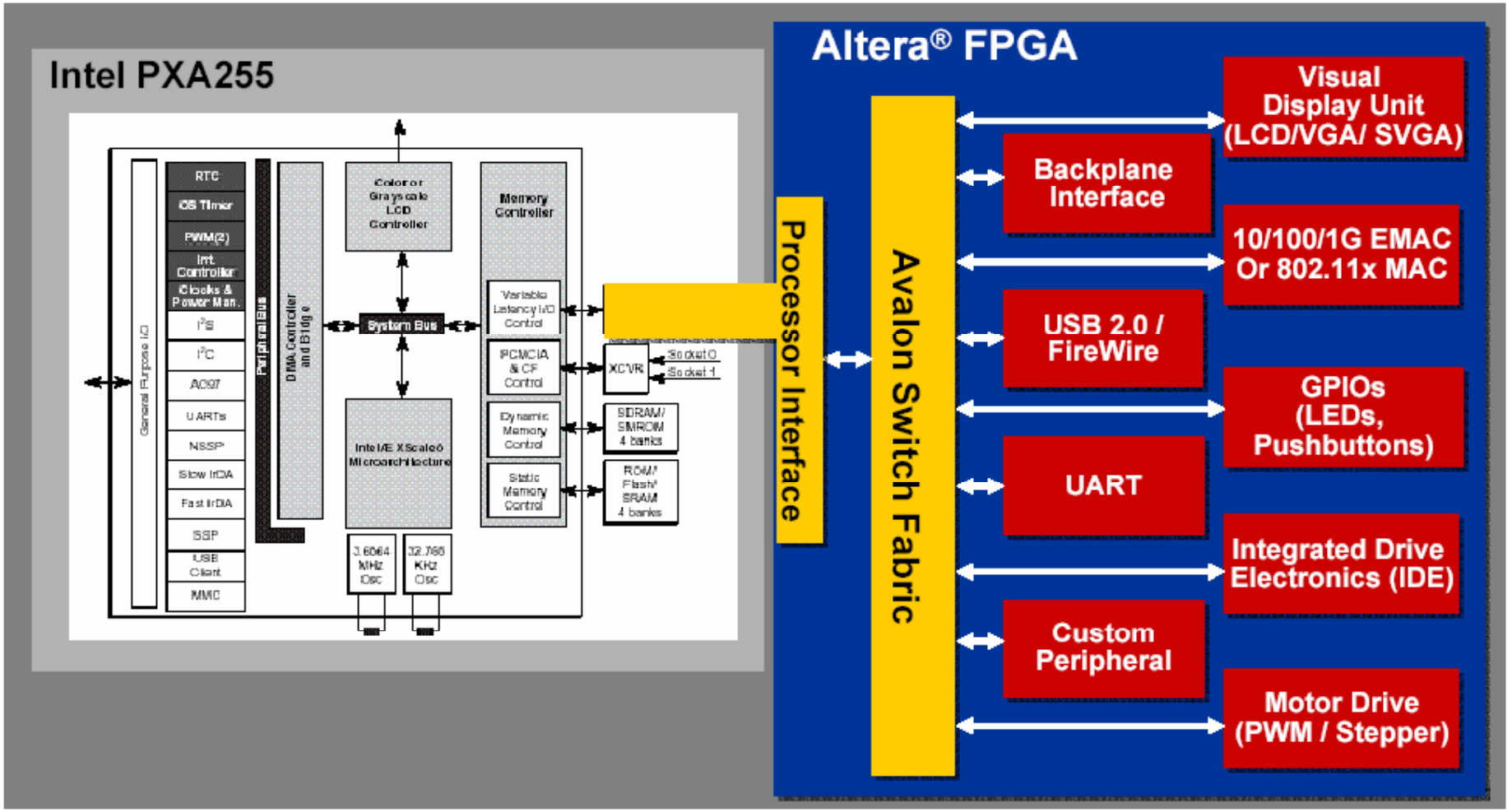
- Fill in fields
- Add component to SOPC Builder portfolio
- Can add parameterizing capability to component

Add Component to SOPC System

- Default location is the **User Logic** folder

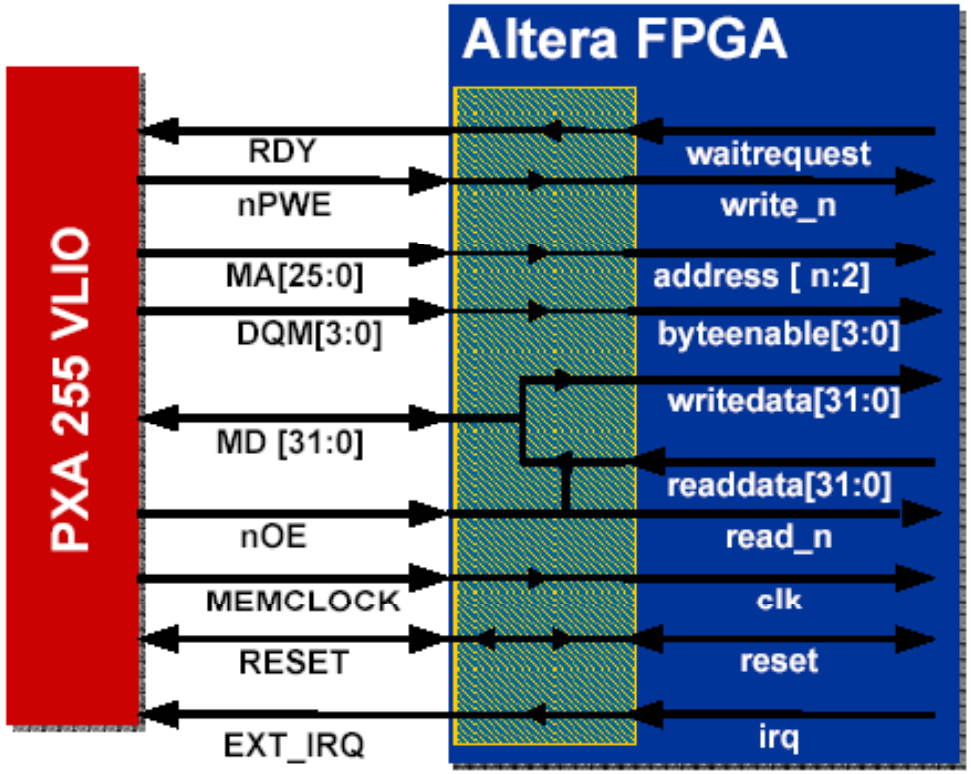


Intel PXA255 Example



VLIO as an Avalon Master Port VLIO

- Intel PXA255 Variable Latency I/O (VLIO) Uses a Bi-Directional Data Path, RDY Signal to Add Wait States
- Interface Separates DATA into Read Data & Write Data Paths



VLIO	Avalon
RDY	waitrequest
MA [25:0])	address
nPWE	write_n
MD[31:0]	writedata
nOE	read_n
MD [31:0]	readdata
DQM[3:0]	Byteenble[3:0]
EXT_IRQ	irq
MEMCLOCK	clk
RESET	reset

Relevant Verilog Code to Relevant Verilog Code to Implement

```
module ext_proc_if (
    //port declarations
    )
    //signal declarations
    assign a_write_n = e_pwe_n;
    assign a_read_n = e_oe_n;
    assign a_addr = {e_ma, 2'b0};
    assign e_rdy = a_waitrequest;
    assign a_be_n = e_dqm_n;
    // work out the bi-directional data bus
    // if output enable is low, then get the data from the readdata path of the Avalon switch fabric
    assign e_data = (!e_oe_n &&  !e_cs_n)? a_data_read : 'bz;
    // assign the Avalon Switch Fabric write data path to athe a_data_write net (i.e. the incoming data..
    assign a_data_write = a_write_data;
    always @(!e_cs_n)
          begin
         if (!e_pwe_n ) a_write_data = e_data;
          end
    endmodule
```