Avalon Interface and Custom Components

Week 5 Lecture 2

Reminders

- Avalon and Custom Components
- Lab 2 is due this week
- Quiz due Sunday
  - **IO Devices, Custom Components**
- Demo due Friday
  - **Custom IP Demo**

- Exam 1 next Wednesday!!!
  - **In person during lecture**
  - **You may have a cheat sheet**
  - **Don’t forget to include assembly**

What is the Avalon Interface?

- Altera Avalon Interface
  - **Provides easy connection to components in Altera FPGA**
  - Standard interfaces are designed into Platform Designer/QSYS components
  - **Can use interface for your own custom components**
  - **Component can include any number Avalon interfaces**
    - Can also include multiple instances of the same interface type
  - **Avalon interfaces are an open standard**
    - No licenses or royalties are required to use
  - **Each Avalon interface defines its own signals & behavior**
    - Many signals are optional ➔ allows flexibility in implementation
    - Signals can be either active high or active low
    - In general each interface may have 1 signal of each signal role
    - Most signals are synchronous to clock and reset

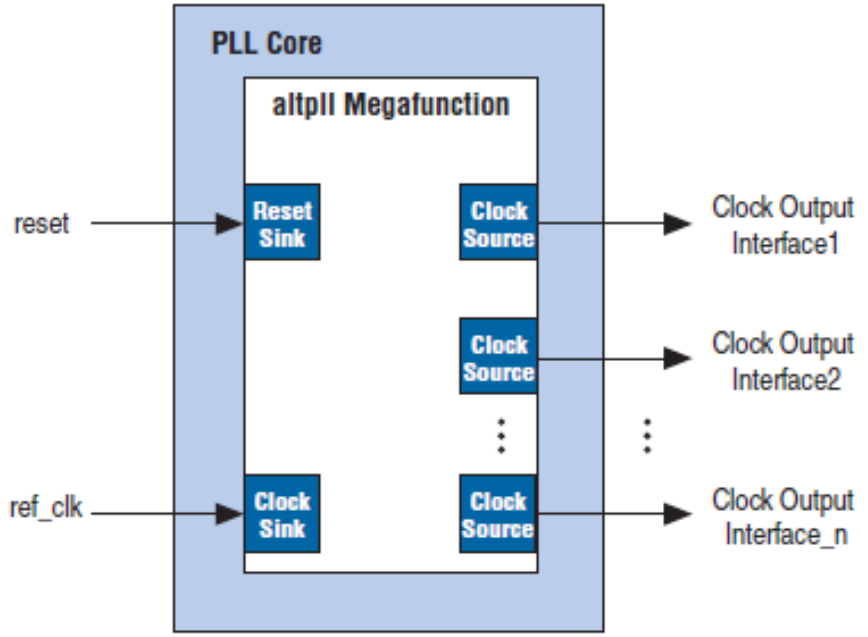
Interface Specifications

- Altera Avalon Interface
  - **No defined performance for Avalon Interfaces**
    - Depends on component design and system implementation
  - **Specification defines seven different interfaces**
    - Avalon Clock Interface
    - Avalon Reset Interface
    - Avalon Memory-Mapped
    - Avalon Interrupt Interface
    - Avalon Conduit
    - Avalon Conduit Tri-state
    - Avalon Streaming

Refer to Avalon Interface Specification for details

Clock Interface

- Avalon Clock Interface
  - **Define the clock or clocks used by component**
    - Components can have clock interfaces that are input, output or both
  - **Example block diagram**
    - Clock sinks are clock inputs
    - Clock source are output clocks



Reset Interface

- Avalon Reset Interface
  - **Provide both soft and hard reset functionality**
    - Soft reset logic typically reinitializes registers and memories without powering down the device
    - Hard reset logic initializes the device after power-on

Memory Mapped Interface

- Avalon Memory Mapped Interface (Avalon-MM)
  - Used for read & write interfaces on master/slave devices
    - Master and slave interfaces connected by an interconnect fabric
  - **Interface used for processors and memory devices**
    - Can be used on a wide variety of devices
  - Support simple fixed-cycle read and write transfers
  - Support complex pipelined interfaces w/ burst transfers
  - Only signals required for the component are implemented
    - Provides a very flexible interface
    - Only one instance of each signal role can be used
    - Minimum signal requirements
      - ***readdata*** *for a read-only interface*
      - ***writedata*** *and* ***write*** *for a write-only interface*

Memory Mapped Interface

- Avalon Memory Mapped Interface (Avalon-MM)
  - Only supports aligned accesses
    - Master can only issue addresses that are multiples of its data width
  - *Supports Dynamic bus sizing*
    - Dynamically manages transfers between master-slave pairs of differing data widths
    - If master is wider than the slave, data bytes in the master address space map to multiple locations in the slave address space
    - If master is narrower than the slave interconnect, Avalon System Interconnect Fabric manages the slave byte lanes using byte enables.

Memory Mapped Interface

- Avalon Memory Mapped Interface
  - Important Signals
    - Address

Masters: By default, the `address` signal represents a byte address. The value of the address must be aligned to the data width. To write to specific bytes within a data word, the master must use the `byteenable` signal. Refer to the `addressUnits` interface property for word addressing.

Slaves: By default, the interconnect translates the byte address into a word address in the slave's address space so that each slave access is for a word of data from the perspective of the slave. For example, `address`= 0 selects the first word of the slave and address 1 selects the second word of the slave. Refer to the `addressUnits` interface property for byte addressing.

Memory Mapped Interface

- Avalon Memory Mapped Interface
  - Important Signals

`read` `read_n`	Master → Slave	Asserted to indicate a `read` transfer. If present, `readdata` is required.
`readdata`	Slave → Master	The `readdata` driven from the slave to the master in response to a `read` transfer.
`write` `write_n`	Master → Slave	Asserted to indicate a `write` transfer. If present, `writedata` is required.
`writedata`	Master → Slave	Data for write transfers. The width must be the same as the width of `readdata` if both are present.

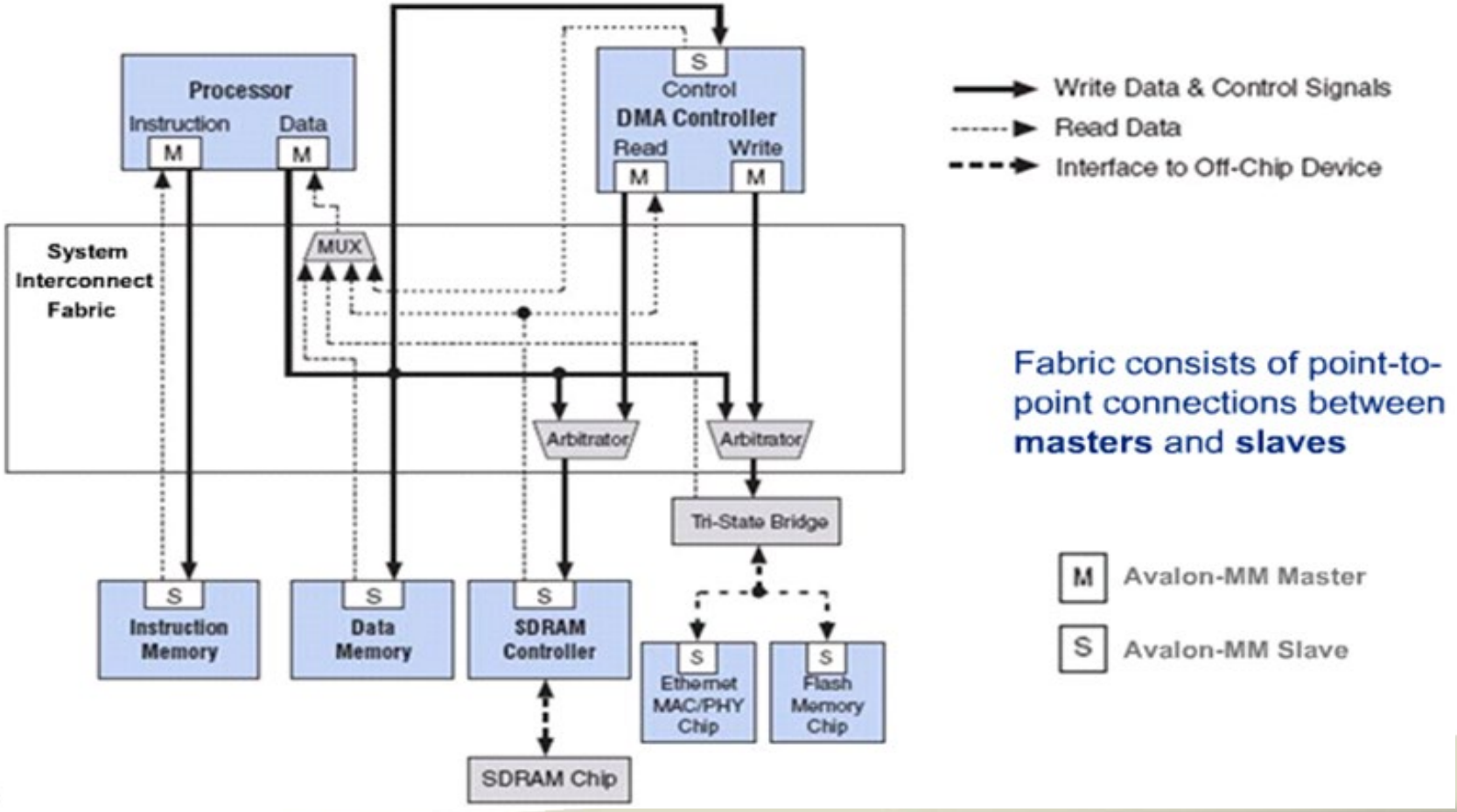
Memory Mapped Interface

- Avalon Memory Mapped Interface
  - Important Signals
    - Byte Enable
      - ***Enables specific byte lane(s) on port widths greater than 8 bits***
      - ***Strongly recommended to avoid unintended side effects***

1111	writes full 32 bits
0011	writes lower 2 bytes
1100	writes upper 2 bytes
0001	writes byte 0 only
0010	writes byte 1 only
0100	writes byte 2 only
1000	writes byte 3 only

Avalon Switch Fabric

- Avalon Memory Mapped Interface Example



Interrupt Interface

- Avalon Interrupt Interface
  - **Allow slave components to signal events to master**
    - Ex: JTAG UART, PIO, DMAC, etc.
  - **IRQ signal must be synchronous to rising edge of clock**
  - **IRQ signal** must be asserted until interrupt acknowledge
    - Implied level triggered
  - **IRQ signal can be active high or active low**
  - **Interrupt receiver typically determines response to event**

Streaming Interface

- Avalon Streaming Interface (Avalon-ST)
  - **Provides high bandwidth, low latency**
    - Data is unidirectional
    - Fire hose of interfaces
    - Ex: Streaming video
  - Supports single stream of data without acknowledge
  - **Supports** complex protocols capable of bursting and interleaving across multiple channels.
  - **Sideband signaling of channel**
    - Used for error, start/end of packet delineation

Conduit Interface

- Avalon Conduit
  - **Used to group together an arbitrary collection of signals**
  - Can specify any role for signals
    - Supports input, outputs, and bidirectional signals
  - **Used to route signals out of or into QSYS system**
    - Allow connection to external signals

Tri-state Conduit Interface

- Avalon Tri-state Conduit (Avalon-TC)
  - **Similar to Avalon Conduit**
  - Designed for on-chip controllers that drive off-chip components
    - Allows pins to be shared across multiple devices
  - Point-to-point interface
  - **Requires request and grant signals**

CUSTOM COMPONENTS

What are Custom Components?

- Written in VHDL and are slaves to the NIOS II processor
- Mainly used to interface the NIOS II processor to other modules instantiated on the FPGA and to other hardware peripherals external to the FPGA

What are Custom Components?

- Once a user writes the VHDL for a custom component, the component editor in the SOPC builder is used to create the necessary files and make the component part of the library.
- Once a component is part of the library, it can be added to the nios_system and linked to the processor in the same manner as the other components provided by Intel.

What are Custom Components?

- When the nios_system is generated, the hardware description for the custom component becomes part of the nios_system.vhd.
- ***NOTE* : The custom component becomes part of the nios_system.vhd. It should not be instantiated in the top_level VHDL ***
- Edits made to the custom component's VHDL file will be reflected in the next compile of the nios_system.

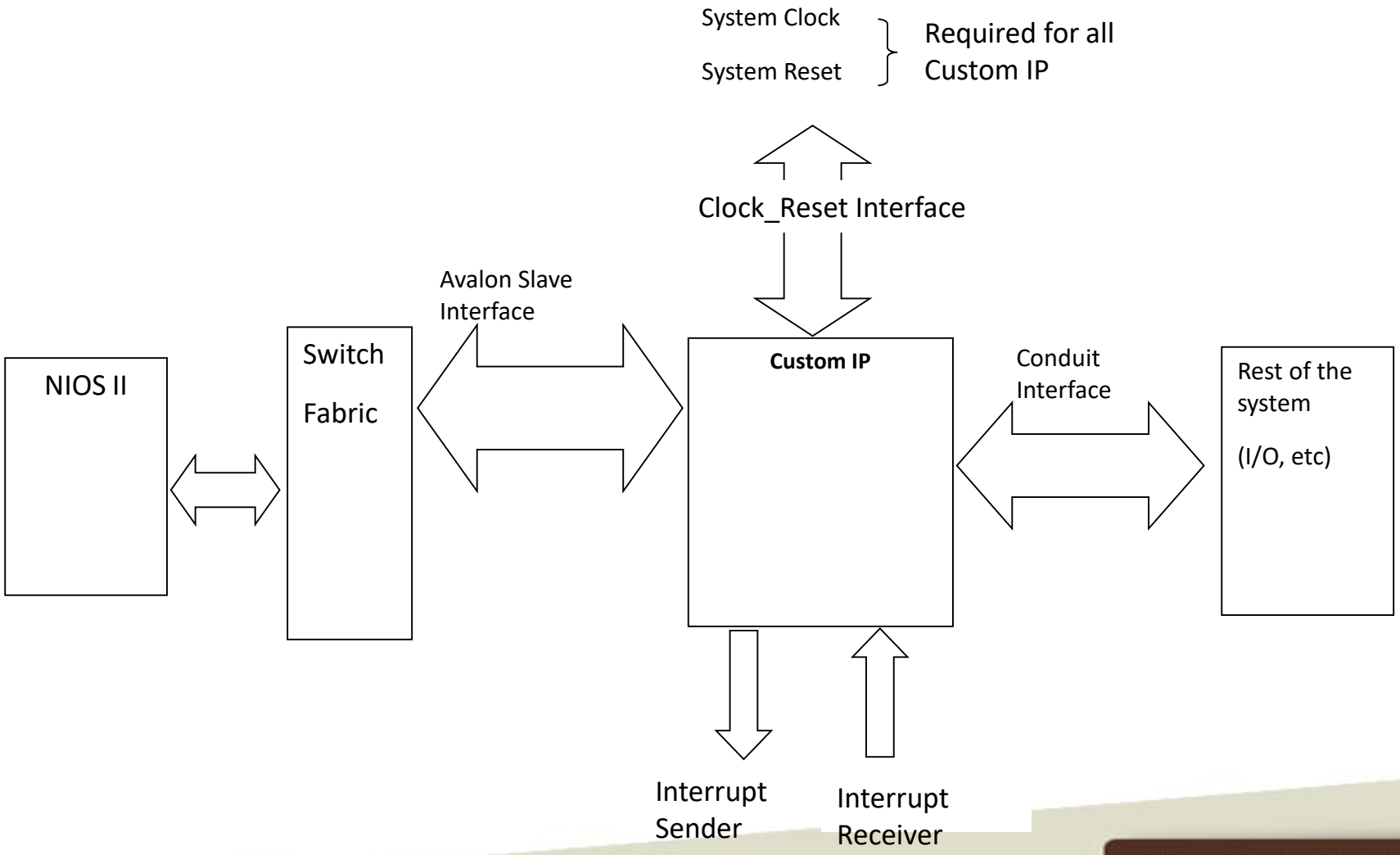
SOPC Component Editor

- The component editor provides a GUI to support the creation and editing of the Hardware Component Description File (_hw.tcl) file that describes a component to the SOPC Builder.
- Used for
  - **Specifying the HDL file(s) that describe your component**
  - **Specifying the signals for each of the component's interfaces and define the behavior of each interface signal**
  - **Specifying relationships between interfaces**

Component Hardware Structure

- The component editor creates components with the following characteristics:
  - **A component has one or more interfaces including**
    - Avalon memory mapped (MM) master and/or slave
    - Interrupt sender and/or receiver
    - Clock input and/or output
    - Conduit (for exporting signals to the top level)
  - **Each interface is comprised of one or more signals**

Custom Component Interfaces



Avalon Slave Interface

- This is the interface with the CPU that is used to get data into and out of the Custom Component.
- If there is only one piece of data (32 bits or less) that the processor will be writing to the component, a single register in the component is all that is needed to store that data. The port must contain a write enable signal (WE) and a write_data signal (bus).
- If there is more than one piece of data that the processor will be writing to the component, it must contain an array (inferred RAM) to store the data from the CPU. Data is written to and read from this array. In this case, the port must contain a write enable signal, a write_data signal, and an address signal.

Avalon Slave Interface

- There are several signal types in this interface. The following are ones that you will need to be familiar with:
  - **Write – this is an active high write signal. When this signal is driven high, the data on the writedata lines is written into the custom IP's data array (inferred RAM) at the address on the address lines.**
  - **Read – this is an active high read signal. When this signal is driven high, the component takes the data from the data array at the address specified by the address lines and puts the data on the readdata output**
  - **Writedata – This is data from the CPU that is stored in the custom IP's data array**
  - **Readdata – This is data that is read from the custom IP's data array back to the CPU**
  - **Address – This indicates which location in the custom IP's data array is being read from or written to. Assume that the data array is 32 bit (4 bytes) wide. If you reference the base address of the custom IP in your C program, you will be accessing the data at address 0. If you reference the base address + 1, you will be accessing the data at address 1. In the demo, we have an array of 8 32-bit values with addresses 0 through 7.**

Other Interfaces

- Conduit Interfaces – signals in this interface show up as input and output ports on the nios_system entity generated from the SOPC builder.
- Interrupt Sender – this is used if your custom IP generates an interrupt
- Interrupt Receiver – this is used if your custom IP can be interrupted. We will not use this in ESD.
- Clock_reset interface - this one is required and self-explanatory

Bottom Up Design

- With HDL Files tab you can specify VHDL file that describes the component logic
- Component editor analyzes the file
- If the file is successfully analyzed, the component editor's Signals tab lists all design modules in the Top Level Module list