The On-Chip Peripheral Bus

Prof. Stephen A. Edwards

sedwards@cs.columbia.edu

Columbia University

Spring 2006

## The On-Chip Peripheral Bus

Developed by IBM

Part of their CoreConnect architecture designed for integrating on-chip "cores"

Something like "PCI on a chip"

Spec. allows for 32- or 64-bit addresses and data

Xilinx Microblaze variant uses 32-bit only

Intended System Architecture
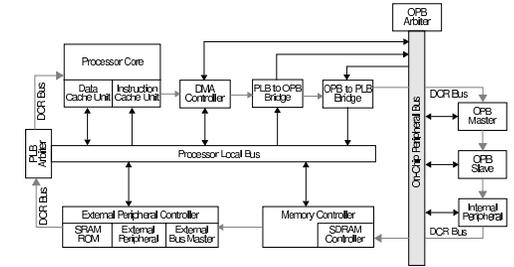
Source: IBM

Physical Implementation
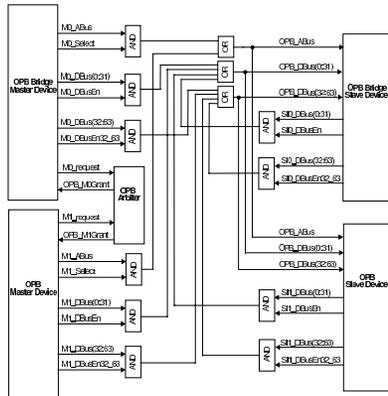
Masters and Slaves

Most bus protocols draw a distinction between

Masters: Can initiate a transaction, specify an address, etc. E.g., the Microblaze

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

---

Naming Conventions

For OPB slave devices,

prefix	meaning
OPB_	Signals from OPB bus logic to slave
Sln_	Signals from slave to OPB

OPB slave signals (Xilinx)

	Sln_xferAck	→
	Sln_toutSup	→
	Sln_retry	→
	Sln_DBus[0:31]	⇒
	Sln_errAck	⇒
Slave ←	OPB_Clk	**OPB**
←	OPB_Rst	
←	OPB_select	
←	OPB_RNW	
←	OPB_SeqAddr	
⇐	OPB_BE[0:3]	
⇐	OPB_ABus[0:31]	
⇐	OPB_DBus[0:31]	
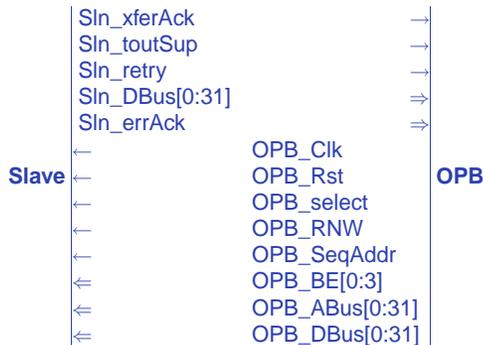
OPB Signals

OPB_Clk	Bus clock: master synchronization
OPB_Rst	Global asynchronous reset
OPB_ABus[0:31]	Address
OPB_BE[0:3]	Byte enable
OPB_DBus[0:31]	Data to slave
OPB_RNW	1=read from slave, 0=write to slave
OPB_select	Transfer in progress
OPB_seqAddr	Next sequential address pending (unused)
Sln_DBus[0:31]	Data from slave. Must be 0 when inactive
Sln_xferAck	Transfer acknowledge. OPB_select→0
Sln_retry	Request master to retry operation (=0)
Sln_toutSup	Suppress slave time-out (=0)
Sln_errAck	Signal a transfer error occurred (=0)

Bytes, Bits, and Words

The OPB and the Microblaze are big-endian:

0 is the most significant bit, 31 is the least

Bytes and halfwords are left-justified:

	msb						lsb
Byte	0		1		2		3
Bit	0	7	8	15	16	23	24	31

|			
Word	0		31
Halfword	0	15	
Byte	0	7	

In VHDL

```
entity opb_peripheral is
  generic (
    C_BASEADDR   : std_logic_vector(0 to 31) := X"FFFFFFFF";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"00000000";
    C_OPB_AWIDTH : integer                   := 32;
    C_OPB_DWIDTH : integer                   := 32);
  port (
    OPB_ABus   : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE     : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_Clk    : in  std_logic;
    OPB_DBus   : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW    : in  std_logic;
    OPB_Rst    : in  std_logic;
    OPB_select : in  std_logic;
    OPB_seqAddr: in  std_logic;
    Sln_DBus   : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck : out std_logic;
    Sln_retry  : out std_logic;
    Sln_toutSup: out std_logic;
    Sln_xferAck: out std_logic);
end entity opb_peripheral;
```

Typical OPB Read Cycle Timing
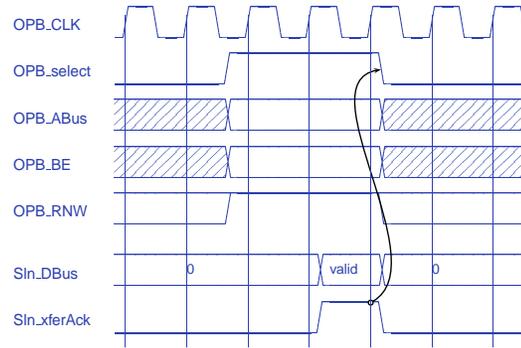
OPB signals arrive late; DBus and xferAck needed early.

Typical OPB Write Cycle Timing
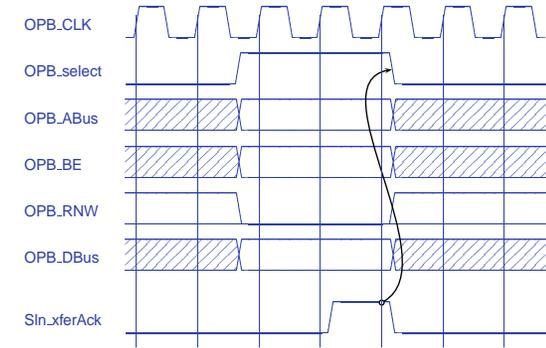
Xilinx Rules

OPB data and address busses are 32 bits

Byte-wide peripherals use data byte 0 and word-aligned addresses (0, 4, . . .)

Peripherals output 0 on everything when inactive

Xilinx does not support complete IBM OPB spec: Dynamic bus sizing is not used

Designing an OPB Peripheral

Let's design a peripheral that contains one of the BRAM blocks.
Reading and writing this peripheral will turn into reading and writing the BRAM.

Block Diagram
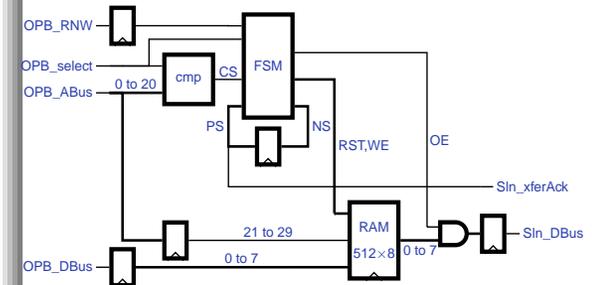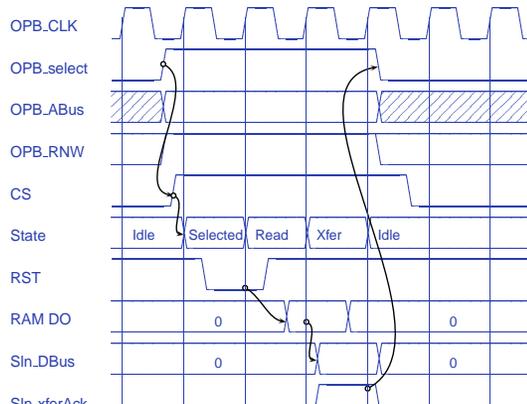
Read Cycle

Back-to-back Read Cycles
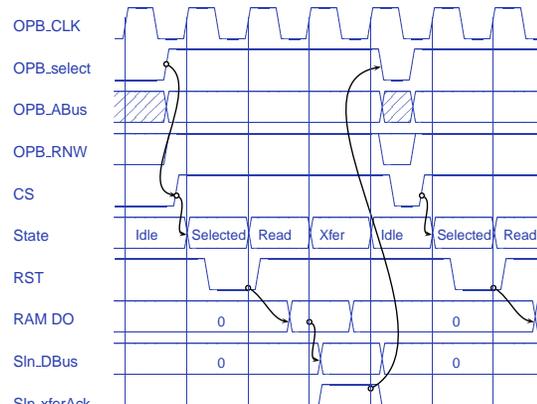
Aborted Read Cycle
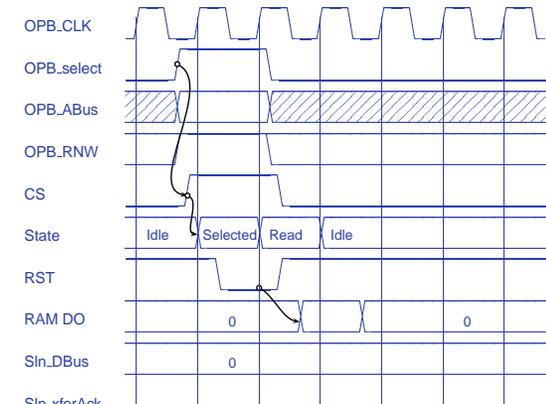
Write Cycle

OPB_CLK
OPB_select
OPB_ABus
OPB_RNW
CS
State   Idle   Selected   Xfer   Idle
WE
RAM DI
Sln_xferAck

FSM

!CS

Idle
000

CS   !SEL

!SEL

Selected
001

/xferAck

SEL & RNW / !RST

SEL & !RNW / WE

Read
011

SEL / OE

Xfer
111
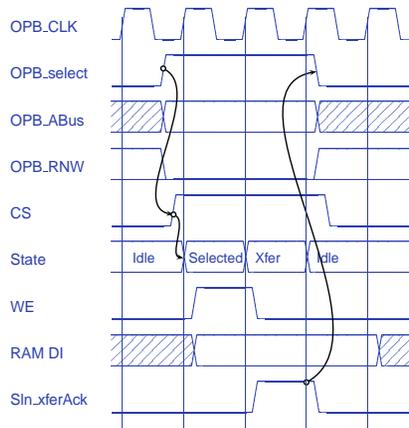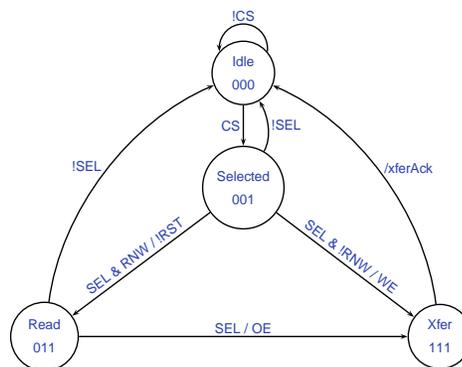
RAM component

```
signal WE, RST : std_logic;
signal RAM_DI, RAM_DO
   : std_logic_vector(0 to RAM_DWIDTH-1);
signal ABus
   : std_logic_vector(0 to RAM_AWIDTH-1);

RAMBlock : RAMB4_S8
port map (
   DO   => RAM_DO,
   ADDR => ABus,
   CLK  => OPB_Clk,
   DI   => RAM_DI,
   EN   => '1',
   RST  => RST,
   WE   => WE);
```

Input Registers

```
register_opb_inputs: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    RAM_DI <= (others => '0');
    ABus <= (others => '0');
    RNW <= '0';
  elsif OPB_Clk'event and OPB_Clk = '1' then
    RAM_DI <= OPB_DBus(0 to RAM_DWIDTH-1);
    ABus <=
      OPB_ABus(C_OPB_AWIDTH-3-(RAM_AWIDTH-1)
               to C_OPB_AWIDTH-3);
    RNW <= OPB_RNW;
  end if;
end process register_opb_inputs;
```

Output Registers

```
register_opb_outputs: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    Sln_DBus(0 to RAM_DWIDTH-1) <=
        (others => '0');
  elsif OPB_Clk'event and OPB_Clk = '1' then
    if output_enable = '1' then
      Sln_DBus(0 to RAM_DWIDTH-1) <=
          RAM_DO;
    else
      Sln_DBus(0 to RAM_DWIDTH-1) <=
          (others => '0');
    end if;
  end if;
end process register_opb_outputs;
```

Chip Select

```
chip_select <=
   '1' when OPB_select = '1' and
     OPB_ABus(0 to C_OPB_AWIDTH-3-RAM_AWIDTH) =
     C_BASEADDR(0 to C_OPB_AWIDTH-3-RAM_AWIDTH)
   else '0';
```

FSM: Declarations

```
constant STATE_BITS : integer := 3;
constant Idle
 : std_logic_vector(0 to STATE_BITS-1) := "000";
constant Selected
 : std_logic_vector(0 to STATE_BITS-1) := "001";
constant Read
 : std_logic_vector(0 to STATE_BITS-1) := "011";
constant Xfer
 : std_logic_vector(0 to STATE_BITS-1) := "111";

signal present_state, next_state
 : std_logic_vector(0 to STATE_BITS-1);
```

FSM: Sequential

```
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    current_state <= Idle;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    current_state <= next_state;
  end if;
end process fsm_seq;
```

FSM: Combinational

```
fsm_comb : process(OPB_Rst, present_state,
                   chip_select, OPB_Select, RNW)
begin
  RST <= '1';
  WE <= '0';
  output_enable <= '0';
  if OPB_RST = '1' then
    next_state <= Idle;
  else
    case present_state is
      when Idle =>
        if chip_select = '1' then
          next_state <= Selected;
        else
          next_state <= Idle;
        end if;
```

FSM: Combinational

```
when Selected =>
   if OPB_Select = '1' then
     if RNW = '1' then
        RST <= '0';
        next_state <= Read;
     else
        WE <= '1';
        next_state <= Xfer;
     end if;
   else
     next_state <= Idle;
   end if;

when Read =>
   if OPB_Select = '1' then
     output_enable <= '1';
     next_state <= Xfer;
   else
     next_state <= Idle;
   end if;
```

FSM: Combinational

```
   -- State encoding is critical here:
   --   xfer must only be true here
   when Xfer =>
     next_state <= Idle;

   when others =>
     next_state <= Idle;
   end case;
 end if;
end process fsm_comb;
```

For more information...

Xilinx Processor IP Reference Guide

IBM On-Chip Peripheral Bus Architecture Specification