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

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

For OPB slave devices,

**prefix** meaning

- **OPB_** Signals from OPB bus logic to slave
- **Sln_** Signals from slave to OPB

<table>
<thead>
<tr>
<th>OPB slave signals (Xilinx)</th>
<th>OPB Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave</td>
<td></td>
</tr>
<tr>
<td>Sln_xferAck</td>
<td>OPB_Clk</td>
</tr>
<tr>
<td>Sln_ToutSup</td>
<td>OPB_Rst</td>
</tr>
<tr>
<td>Sln_retry</td>
<td>OPB_ABus[0:3]</td>
</tr>
<tr>
<td>Sln_DBus[0:31]</td>
<td>OPB_BE[0:3]</td>
</tr>
<tr>
<td>Sln_errAck</td>
<td>OPB_DBus[0:31]</td>
</tr>
<tr>
<td>Sln_xferAck</td>
<td>Data to slave</td>
</tr>
<tr>
<td>Sln_ToutSup</td>
<td>1 read from slave, 0 write to slave</td>
</tr>
<tr>
<td>Sln_retry</td>
<td>Transfer in progress</td>
</tr>
<tr>
<td>Sln_DBus[0:31]</td>
<td>Next sequential address pending (unused)</td>
</tr>
<tr>
<td>Sln_xferAck</td>
<td>Data from slave. Must be 0 when inactive</td>
</tr>
<tr>
<td>Sln_ToutSup</td>
<td>0 request master to retry operation (–0)</td>
</tr>
<tr>
<td>Sln_retry</td>
<td>Suppress slave time-out (–0)</td>
</tr>
<tr>
<td>Sln_errAck</td>
<td>Signal a transfer error occurred (–0)</td>
</tr>
</tbody>
</table>

The OPB and the Microblaze are big-endian:
0 is the most significant bit, 31 is the least
Bytes and halfwords are left-justified:
In VHDL

```vhdl
entity opb_peripheral is
generic ( 
  C_BASEADDR : std_logic_vector(0 to 31) := "X"'FFFFF""; 
  C_HIGHEADOR : std_logic_vector(0 to 31) := "X"'00000000""; 
  C_OPB_WIDTH : integer := 32; 
  C_OPB_ADDR_WIDTH : integer := 32; 
);
port ( 
  OPB_ABus : in std_logic_vector(0 to C_OPB_ADDR_WIDTH-1); 
  OPB_BE : in std_logic; 
  OPB_RNW : in std_logic; 
  OPB_CLK : in std_logic; 
  OPB_SELECT : in std_logic; 
  OPB_xferAck : out std_logic; 
  Sln_DBus : out std_logic_vector(0 to C_OPB_ADDR_WIDTH-1); 
  Sln_xferAck : out std_logic; 
  Sln_errAck : out std_logic; 
); 
end entity opb_peripheral;
```

Xilinx Rules

OPB data and address busses are 32 bits
Byte-wide peripherals use data byte 0 and word-aligned addresses (0, 4, 8, ...)
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

![Block Diagram](https://via.placeholder.com/150)

Read Cycle

![Read Cycle](https://via.placeholder.com/150)

Back-to-back Read Cycles

![Back-to-back Read Cycles](https://via.placeholder.com/150)

Aborted Read Cycle

![Aborted Read Cycle](https://via.placeholder.com/150)
**RAM component**

```vhdl
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);
```

**FSM**

```vhdl
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);
```

**Input Registers**

```vhdl
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(0 to C_OPB_AWIDTH-3)-(RAM_AWIDTH-1)
to 0_OPB_AWIDTH-3);
  RNW <= OPB_RNW;
end if;
end process register_opb_inputs;
```

**Output Registers**

```vhdl
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**

```vhdl
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**

```vhdl
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**

```vhdl
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**

```vhdl
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;
      case present_state is
        when Selected =>
          if chip_select = '1' then
            next_state <= Xfer;
          else
            next_state <= Selected;
          end if;
        when Read =>
          if chip_select = '1' then
            next_state <= Read;
          else
            next_state <= Selected;
          end if;
      end case;
    end case;
  end if;
end process fsm_comb;
```
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;
end case;
end process fsm_comb;

-- 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;