Message-Based Hardware/Software Communication in HDL/C Environments

Linus Tauro
Quickturn Design Systems, Inc.
440 Clyde Ave.
Mountain View, CA 94043
tauro@cs.ucr.edu

Frank Vahid
Department of Computer Science
University of California
Riverside, CA 92521
vahid@cs.ucr.edu, www.cs.ucr.edu

Abstract

Implementing communication between hardware and software components can be a time-consuming and error-prone task. We describe a set of VHDL routines, with accompanying C routines, that can be used to greatly simplify the high-level specification and implementation of communication between hardware and software processor components. The routines come close to presenting the programmer with the abstraction of message-passing communication using send/receive primitives. Because they use existing language constructs, the routines are fully simulatable using standard VHDL simulators, and can be converted to implementations using existing VHDL synthesis tools. We demonstrate the use of the routines in several examples involving the PC ISA bus protocol. We also demonstrate the need for additional routines, beyond just send/receive, to support communication in real design examples.

1 Introduction

Systems often consist of a combination of processing components, such as microprocessors, microcontrollers, and custom ASIC/FPGA processors. Ideally, a system designer would be able to write and simulate software (C, C++) and hardware (VHDL, Verilog) programs using abstract send/receive primitives for communication between processes running on those processors; such communications could occur not only within software or within hardware, but also between hardware and software. Later, the designer would bind those communications to physical ports using particular protocols. Finally, the designer would compile the C or C++ and synthesize the VHDL or Verilog to implementations.

Common languages like VHDL, Verilog, C and C++, do not include such communication primitives. To solve this shortcoming, we are developing a set of communication libraries in C, C++ and VHDL. These libraries consist of routines and data objects built from existing language constructs without any language extensions, and therefore can be input to existing compilers, simulators, and synthesis tools. For a given existing communication protocol, such as the PC ISA bus protocol, a collection of data and routines is created (we'll call this collection a "class," from C++'s object terminology) and added to the library. A designer needs to declare and initialize a communication channel of a given class, and can then apply what appear to be send/receive primitives over that channel. In reality, the primitives are actually routines that carry out the communication over ports, so are simulatable in existing environments. Those primitives have also been pre-tested (through synthesis or compilation), so provide a simple path to implementation. We refer to the library as OOCL (Object-Oriented Communication Library). OOCL classes are being developed for numerous common protocols, such as I2C, PC ISA, RS232 serial, PC parallel, PCI, as well as custom protocols typically used between multiple custom hardware processors.

The OOCL approach complements related codesign interface research. In [1], a solution is proposed that automates the hardware-software interface using minimum glue logic, while satisfying timing constraints. To communicate with a peripheral device, the processor generates a sequence of signals (SEQs) that read and write the device's ports. Symphony [2] defines a standard communication protocol for all processor components, using send and receive operations with a synchronous wait protocol, where the sender asserts a ready signal and then waits for the receiver to assert its own ready signal. In [3], a codesign methodology is discussed using process communication primitives that allow three types of process interaction: synchronized data transfer, unsynchronized data transfer and synchronization without data transfer. In [4], a system design methodology is discussed using abstract send/receive channels for communication. In all of these approaches, OOCL can be used to encapsulate the communication using send/receive primitives, while implementing the protocol, without any modification to the C or VHDL languages. Other related work includes techniques to interface incompatible protocols by generating an interface process [5] or by synthesizing interface hardware [6]. Some other works have suggested extensions to existing languages.

In this paper, we present details of OOCL routines for one particular protocol, the PC ISA bus, demonstrate their practical use on two examples, and show why some routines in addition to just send/receive are necessary to support somewhat complex examples.

2 ISA communication

The PC-XT/AT bus, also called the industry standard architecture (ISA) bus [7], supports 8 and 16 bit datapaths, and 20 to 24 bit addressing. Today's increasingly popular PCI standard includes within the PCI subsystem, a PCI/ISA bridge that supports the current 8 and 16 bit peripherals used in PCs. Most PC cards are ISA compatible. The PC is typically the master of the bus, initiating reads or writes of memory or input/output (I/O) devices by placing an address on the bus and asserting the appropriate control signals.

OOCL classes have been developed for the ISA bus 8 and 16-bit I/O device read and write cycles. On the PC side exists a C++ class consisting of routines to send or receive data to or from any device. On the hardware device side exist complementary VHDL routines. The C++ class for sending is shown in Figure 1, while the VHDL class for receiving is shown in Figure 2; the C++ receive and VHDL send routines are not shown. We use an apostrophe in our figures to indicate an active-low VHDL signal, although VHDL syntax would require a different indentifier.

Declare Channel

```
class CL_SEND_M_ISA
{
   // Ports are pre-defined on the ISA bus
   // expansion slots and connections are made
   // by an adapter board
public:
   // Initialize ports to default values
   void InitDefault();
   // Send the message(character)
   void Send(uint addr, uchar message);
};
```

Initialize channel (default)

```
// Port initialization is done at  initialization or
// reset usually as part of the BIOS initilaization
// routines
void CL_SEND_M_ISA::InitDefault()     {;}
```

Send message (character)

```
void CL_SEND_M_ISA::Send(uint addr, uchar
                                        message)
{
   asm {
          mov dx, addr;
          mov al,  message;
          out   dx, al;
       }
}
```

Figure 1: OOCL master ISA bus C code.

Each class consists of the following:

1. **Declarations:** The PC's C++ send class is called *CL_SEND_M_ISA*, while the VHDL receive class is called *CL_REC_S_ISA*; the M and S refer to the master (initiator) and servant (responder) of the communication. For the PC, the ISA ports are pre-defined on the expansion slots, and the connections are made by an adapter. The port access is internally handled by the operating system. For the VHDL, the ISA ports are declared as part of the entity declaration, by the user. The VHDL class has a data member to store the 16-bit receiver address.

Declare Channel

```
type CL_REC_S_ISA is
   --The ISA ports used in the communication library
   --i.e. data_p, addr_p, reset_p, iow_p', ior_p',
   -- aen_p, clk_p are declared by the user as part of
   --entity declaration
   record
   addr : bit16;                    -- receiver address
   end record;
```

Initialize channel (default)

```
procedure CL_InitDefaultRecSrvISA(
   signal chan   :  out CL_REC_S_ISA;
   signal data_p : out STD_LOGIC8;
   signal clk_p   : in bit) is
begin
   -- Allocate a default ISA address to receiver
   chan.addr <= CL_DEFAULT_ADDR;
   data_p      <= "ZZZZZZZZ"; -- high impedance
   wait until clk_p'event and clk_p='1';
end CL_InitRecSrvISA;
```

Receive message (character)

```
procedure CL_RecSrvISA(
   signal chan          : in CL_REC_S_ISA;
   variable message  : out STD_LOGIC8;
   signal addr_p        : in bit16;
   signal data_p        : in STD_LOGIC8;
   signal aen_p         : in bit;
   signal reset_p       : in bit;
   signal iow_p'        : in bit;
   signal clk_p         : in bit) is
begin
   -- Wait for read request and my address on the bus
   wait until (clk_p'event and clk_p='1' and aen_p='0'
                  and reset_p='0' and iow_p'='0' and
                                  addr_p = chan.addr;
   -- Read the data while iow_p' is asserted
   message   :=  data_p;
   wait until clk_p'event and clk_p='1' and    iow_p'='1';
   data_p      <= "ZZZZZZZZ"; -- high impedance
   wait until clk_p'event and clk_p='1';

end CL_RecSrvISA;
```

Figure 2: OOCL servant ISA bus VHDL code.

2. **Initializations:** Initialization configures the channel's hardware. OOCL usually provides a default initialization routine with no or few parameters, along with a custom initialization routine with numerous user-configured parameters. In this example, the PC ports are initialized at startup or during reset by the BIOS routines, so there are no actions in the initialization routine. The receiver default initialization sets the ISA receiver channel address to be an unused I/O port address. The *STD_LOGIC* resolved data type from the *IEEE.STD_LOGIC_1164* library is used for the bi-directional data ports. These are set to high impedance during initialization.
3. **Send/Receive:** Figures 1 and 2 show the PC send and VHDL receive routines for a byte transfer. The PC initiates an I/O write cycle, by executing an OUT instruction. The OOCL receive procedure proceeds for the I/O write cycle.

3 Examples

3.1 FPGA coprocessor

In this section, a coprocessor example is used to demonstrate OOCL use. The example is that of computing the greatest common divisor (GCD) of two numbers on an FPGA. Two different polling approaches are used to indicate the computation's completion to the PC, one using a flag, and one using a sentinel value.

Figure 3(a) shows the flag approach, in which a distinct address is assigned to a flag that will be set when the computation is complete. The OOCL ISA ports are declared as part of the entity declaration. Three distinct addresses are assigned from the ISA unused block of addresses to indicate whether the ISA master(PC) is sending raw data for the GCD process, reading the result or reading the flag address. The master reads the result address only after it determines the result is ready by detecting a '1' at the flag address (*gcd_done_addr*). In other words the PC first sends the raw GCD data, polls the GCD flag address until it reads a "one", and then finally reads the result. Three servant channels are declared as VHDL global signals, one for sending the data to GCD, one for receiving the flag, and one for receiving the GCD result data. Consistent with the OOCL approach, the channels are declared, initialized, and then used for the data transfers. In the main GCD process, after initialization of the GCD send and receive channels, the process enters a loop. The GCD flag is set to zero, the input data (x and y) is received from the ISA master, and then the GCD is computed. Although the computation is not shown here because of lack of space, the reader can refer to Figure 5(a). After the computation is complete, the GCD flag is set, and the result is then sent when the ISA master queries the GCD result address.

In the meantime, the flag address process first initializes the flag address channel, and then continuously returns a "zero" whenever the ISA master queries the flag address, as long as the GCD flag is not set by the main process. As soon as the GCD has been computed by the GCD process, as indicated by the GCD flag, the flag process returns a "one" when the ISA master queries the flag address. The flag process then waits for the GCD flag to be set back to zero before it repeats the loop.

Figure 3(b) shows the block diagram for the ISA interface for a 16-bit ISA data bus. The examples deal with an 8-bit data bus, the only difference being that *iocs16_p* is asserted by the ISA servant to indicate to the ISA bus master that it can support a 16-bit transfer.

Figure 3(c) shows a sentinel process approach. The entity, architecture declarations, and the GCD process are the same as in the flag approach, except that there is no need for a flag address. The sentinel process replaces the flag process in Figure 3(a). In this approach, the ISA bus master first sends the input data for the GCD computation, after which it queries the GCD coprocessor for the result. A result of "zero" means that the result is not ready yet. The first non-zero result is assumed to be the GCD result.

3.2 FPGA bus-controller for multiple processes

The example discussed in the previous section had only one computation process. Often, two or more computations are performed simultaneously on the FPGA, in which case it would be advantageous to have a bus-controller process handle all the communications with the environment and co-ordinate data transfer between the other processes. Using a bus-controller can reduce the interconnect area as well as simplify the high-level communication model. Only the bus controller external interface needs to interface to the ISA bus, while internally a much simpler bus, such as one using a two-phase handshake protocol, can be used.

Figure 4(a) shows the bus controller process. The entity declaration is the same as in the GCD example discussed in the previous section and includes the OOCL ISA ports. Four distinct ISA addresses are used, all of them having the first 12 bits in common. The FPGA module address is composed of these 12 bits. The next four bits determine which process is being addressed, i.e., GCD or RSA, and whether data is being sent or received. The bus controller process communicates with the GCD and RSA processes, using a two-phase bus with 8 data lines, the ports for which are internally declared as global signals. The *STD_LOGIC* resolved data type from the *IEEE.STD_LOGIC_1164* is used for these signals, since the data ports in the OOCL ISA channels use this type. The bus controller acts as the master for the two-phase interface. Four distinct master channels are declared for sending and receiving data from the GCD and RSA processes, and the corresponding servant channels are declared for the GCD and RSA processes. All these channels are declared as global signals.

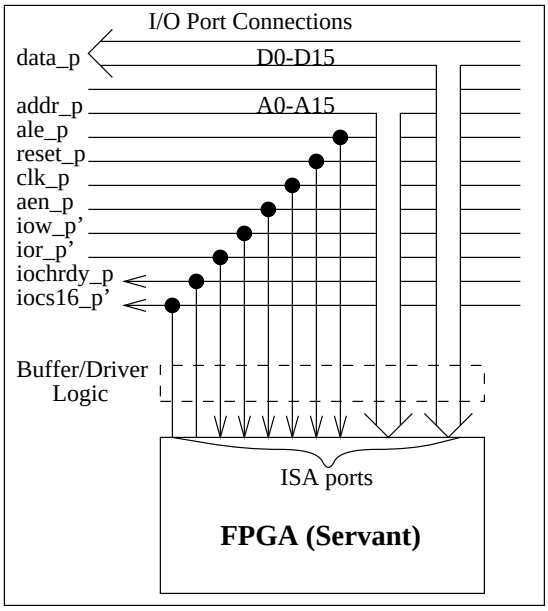
The same bus controller process acts as the servant for the ISA transfer, and the OOCL ISA send and receive channels are also declared globally. The bus controller process begins by initializing the ISA send and receive channels. Next it enters a loop where it waits for an ISA read or write command to be issued as determined by the ISA servant channel's *Ready()* member function, which has been added to OOCL.

Figure 5(c) shows the procedure for *ReadySend()*; *ReadyRec()* would be similar. If the FPGA module address is on the ISA bus, and *ior_p'* is asserted, the address is returned so that bits other than module address can be examined to determine the actual channel being addressed. To ensure timing constraints are met, *iochrdy_p* is asserted to request a bus cycle extension.

Coming back to Figure 4(a), the bus controller pro-

```
--Flag address approch for the gcd example using the
-- ISA protocol and the FPGA as a co-processor
use work.CL_ISA.all;
-- The ports for the transfer are declared in the  entity
entity CL_S_ISA is
        port(addr_p       : in bit16;
             aen_p        : in bit;
             reset_p      : in bit;
             clk_p        : in bit;
             data_p       : inout STD_LOGIC8;
             iow_p        : in bit;
             ior_p        : in bit);
end CL_S_ISA;
architecture ACL_S_ISA of CL_S_ISA is
-- Using  addresses from unused  block 0300-0377h
constant gcd_send_addr   : bit 16  := X"0300";
constant gcd_rec_addr    : bit 16  := X"0301";
constant gcd_done_addr   : bit 16  := X"0302";
signal gcd_done_s        : bit  := '0';-- gcd done global
-- Declare the send, receive and the gcd done channels
signal send_chan_s       : CL_SEND_S_ISA;
signal rec_chan_s        : CL_REC_S_ISA;
signal gcd_done_chan_s   : CL_SEND_S_ISA;
begin
   gcd:process
      variable x,y          : STD_LOGIC8;
   begin
                                        -- Initializations
      CL_InitSendSrvISA(send_chan_s, data_p,
                        gcd_send_addr, clk_p);
      CL_InitRecSrvISA(rec_chan_s, data_p,
                       gcd_rec_addr, clk_p);
      loop
         gcd_done_s    <=  '0';
         --Receive x and y
         CL_RecSrvISA(rec_chan_s, x, addr_p, data_p,
                      aen_p, reset_p, iow_p, clk_p);
         CL_RecSrvISA(rec_chan_s, x, addr_p, data_p,
                      aen_p, reset_p, iow_p, clk_p);

         --Compute the gcd from x and y (not shown)
         gcd_done_s    <= '1';          --Set the gcd flag

         --Send the result
         CL_SendSrvISA(send_chan_s, x, addr_p. data_p,
                       aen_p, reset_p, ior_p, clk_p);
      end loop;
   end process;
   flag: process
   begin
                                        --Initialization
      CL_InitSendSrvISA(gcd_done_chan_s, data_p,
                                gcd_done_addr, clk_p);
      loop
         while (gcd_done_s = '0') loop   -- gcd not done
            CL_SendSrvISA(gcd_done_chan,s, X"00",
            addr_p, data_p, aen_p, reset_p, ior_p, clk_p);
      end loop;
      CL_SendSrvISA(gcd_done_s, X"01", addr_p,
                    data_p, aen_p, reset_p, ior_p, clk_p);
      wait until clk_p'event and clk_p='1' and
                                        gcd_done_s = '0';
      end loop;
   end process;
end ACL_S_ISA;
```

(a)



(b)

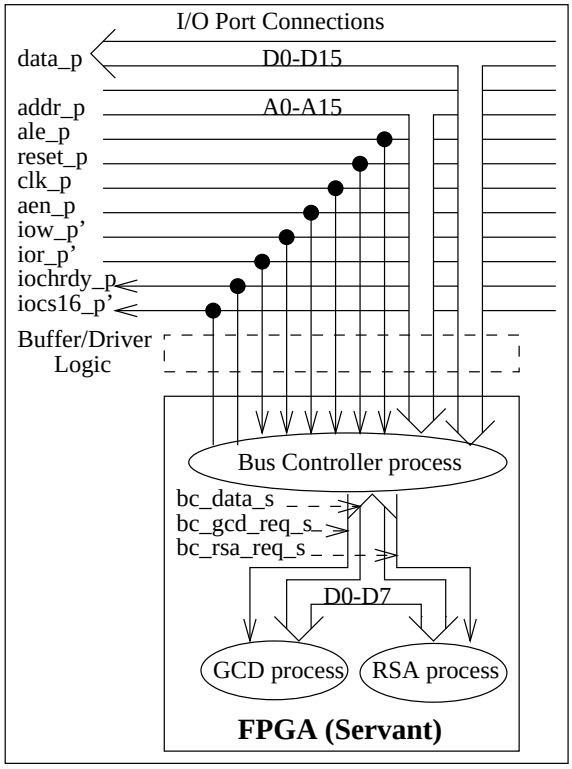
```
--Sentinel approch for the gcd example using the
-- ISA protocol and the FPGA as a co-processor

-- The entity, architecture declarations and the
-- gcd process are exactly the same as in the flag
-- approach (the only difference is that there is no
-- need for a flag address (gcd_done_addr)

-- Instead of the flag process the sentinel process
-- described below is used

sentinel: process
begin
                                        --Initialization
   CL_InitSendSrvISA(gcd_done_chan_s, data_p,
                                gcd_send_addr, clk_p);
   loop
      -- Tell master that gcd result is not ready yet
      while (gcd_done_s = '0') loop
         CL_SendSrvISA(gcd_done_chan,s, X"00",
         addr_p, data_p, aen_p, reset_p, ior_p, clk_p);
      end loop;
   wait until clk_p'event and clk_p='1' and
                                        gcd_done_s = '0';
   end loop;
end process;
```

(c)

Figure 3: GCD example on the FPGA using the ISA protocol: (a) using a flag address, (b) using a sentinel.

```
--Bus controller approach for multiple FPGA processes
use work.CL_ISA.all;              -- VHDL  ISA and
use work.CL_CUSTOM.all;           -- custom libraries

-- The entity declaration is the same as in the gcd exmaple

architecture ACL_S_ISA of CL_S_ISA is
--Using addresses from unused block 0300-0377h
-- The first 12 bits common, used for module address
constant ISA_module_addr    : bit16 := X"0300"
constant  gcd_send_addr     : bit4 := "0000";
constant  gcd_rec_addr      : bit4 := "0001";
constant  rsa_send_addr     : bit4 := "0010";
constnat  rsa_rec_addr      : bit4 := "0011";

--Ports for the D8P2 bus between bc and processes
signal bc_data_s            : STD_LOGIC8;
signal bc_gcd_req_s'        : STD_LOGIC;
signal bc_rsa_req_s'        : STD_LOGIC;
-- Sentinels for the gcd and rsa process
signal gcd_done_s           : bit   := '0';
signal rsa_done_s           : bit   := '0';

--Send, receive channels for the D8P2 interface
signal gcd_send_chan_D8P2_s     : CL_SEND_S_D8P2;
signal gcd_rec_chan_D8P2_s      : CL_REC_S_D8P2;
signal rsa_send_chan_D8P2_s     : CL_SEND_S_D8P2;
signal rsa_rec_chan_D8P2_s      : CL_REC_S_D8P2;
signal bc_gcd_send_chan_D8P2_s: CL_SEND_M_D8P2;
signal bc_gcd_rec_chan_D8P2_s   : CL_REC_M_D8P2;
signal bc_rsa_send_chan_D8P2_s : CL_SEND_M_D8P2;
signal bc_rsa_rec_chan_D8P2_s   : CL_REC_M_D8P2;

--Send, receive cghannels for the ISA intereface
signal bc_send_chan_ISA_s       : CL_SEND_S_ISA;
signal bc_rec_chan_ISA_s        : CL_REC_S_ISA;

-- The functions to test if a particular bit in a bit vector
-- is set and to encode a message for the RSA process
-- go here (not shown)

begin
   -- The bus controller (servant) uses the ISA protocol
   -- to talk with the PC and acts as the master for the
     -- D8P2 protocol it uses to talk to other processes
   bus_controller : process
      variable ISA_Ready               : bit    := '0';
      variable ISA_module_addr_lsb     : int2 := 4;
      variable read_addr               : bit16;
      varaible char_data               : STD_LOGIC8;
   begin
                                  -- Initilaizations
      CL_initSendSrvISA(bc_send_chan_ISA_s, data_p,
                                  ISA_module_addr, clk_p);
      CL_InitRecSrvISA(bc_rec_chan_ISA_s, data_p,
                                  ISA_module_addr, clk_p);
      loop
         -- Wait until a read or write command  is issued
         while (ISA_Ready = '0') loop
            CL_ReadySendSrvISA(bc_send_chan_ISA_s, ISA_
              Ready, ISA_module_addr_lsb, read_addr,
              addr_p, aen_p, reset_p, ior'_p, iochrdy_p, clk_p);
            CL_ReadRecSrvISA(bc_rec_chan_ISA_s, ISA_
              Ready, ISA_module_addr_lsb, read_addr,
              addr_p, aen_p, reset_p, ior'_p, iochrdy_p, clk_p);
         end loop;
```

(a)



(b)

```
         --Read raw data or send result as per the address
         case read_addr(3 downto 0) is
            when gcd_send_addr =>  -- Read gcd raw data
               CL_GetDataSrvISA(char_data, data_p
                                  iochrdy_p, iow_p, clk_p);
               CL_SendMstD8P2(bc_gcd_send_chan_D8P2
              _s, char_data, bc_data_s, bc_gcd_req_s, clk_p);
            when gcd_rec_addr =>
            -- If the gcd has been computed send the result
            -- from gcd process, otherwise send sentinel
            if (gcd_done_s = '0') then
               CL_PutDataSrvISA(X"00", data_p,
                                  iochrdy_p, ior_p', clk_p);
            else
               CL_RecMstD8P2(bc_gcd_rec_chan_D8P2_s,
                 char_data, bc_data_s, bc_gcd_req_s', clk_p);
               CL_PutDataSrvISA(char_data, data_p,
                                  iochrdy_p, ior_p', clk_p);
            end if;
         --The cases for rsa_send_addr and rsa_rec_addr
         -- are similar (not shown)
         when others => -- Error erport can be asserted
         end case;
      end process;
   end loop;
   -- The GCD and RSA processes go here and are shown
   -- in a subsequent figure
end ACL_S_ISA;
```

Figure 4: Bus controller for multiple FPGA processes: (a) bus controller process, (b) block diagram.

```
--GCD process for the bus contoller example
gcd : process
   variable x            : STD_LOGIC8;
   variable y            : STD_LOGIC8;
begin
   --Initializations
   CL_InitSendSrvD8P2(gcd_send_chan_D8P2_s,
      bc_data_s, bc_gcd_req_s', clk_p);
   CL_InitRecSrvD8P2(gcd_rec_chan_D8P2_s,
       bc_data_s,bc_gcd_req_s',clk_p);
   loop
      gcd_done_s <= '0';
      --Receive x
      CL_RecSrvD8P2(gcd_rec_chan_D8P2_s, x,
         bc_data_s, bc_gcd_req_s', clk_p);
      -- Receive y
      CL_RecSrvD8P2(gcd_rec_chan_D8P2_s, y,
         bc_data_s, bc_gcd_req_s', clk_p);
      --Compute the gcd
      while (x /= y) loop
         if (x < y) then
            y := y - x;
         else x := x - y;
         end if;
      end loop;
      gcd_done_s  <= '1';        -- Set the GCD global
      -- Send the result
      CL_SendSrvD8P2(gcd_send_chan_D8P2_s, x,
                     bc_data_s, bc_gcd_req_s', clk_p);
   end loop;
end process;
```

(a)

```
-- RSA process for the bus controller example
rsa : process
   variable pubkey_d             : STD_LOGIC32;
   variable pubkey_n             : STD_LOGIC32;
   variable msg_item_raw         : STD_LOGIC8;
   variable msg_item_encoded     : STD_LOGIC32;
begin
   --Initialzations
   CL_InitSendSrvD8P2(rsa_send_chan_D8P2_s,
                      bc_data_s, bc_rsa_req_s', clk_p);
   CL_InitRecSrvD8P2(rsa_rec_chan_D8P2_s,
                      bc_data_s, bc_rsa_req_s', clk_p);
   --Receive the public keys
   CL_RecLongSrvD8P2(rsa_rec_chan_D8P2_s,
            pubkey_d, bc_data_s, bc_rsa_req_s', clk_p);
   CL_RecLongSrvD8P2(rsa_rec_chan_D8P2_s,
            pubkey_n, bc_data_s, bc_rsa_req_s', clk_p);
   loop
      rsa_done_s    <= '0';
      --Receive a character
      CL_RecSrvD8P2(rsa_rec_chan_D8P2_s,
         msg_item_raw, bc_data_s, bc_rsa_req_s', clk_p);
      --Encode the message
      msg_item_encoded := EncodeMsg(msg_item_raw,
                                    pubkey_d, pubkey_n);
      rsa_done_s    <= '1';          --Set the RSA global
      --Send the result
      CL_SendLongSrvD8P2(rsa_send_chan_D8P2_s, msg
        _item_encoded, bc_data_s, bc_rsa_req_s', clk_p);
   end loop;
end process;
```

(b)

```
-- The communication library procedure for checking
-- if it is the servant's turn to write data to the ISA bus.
-- The companion  for receiving is ReadyRecSrvISA
procedure CL_ReadySendSrvISA(
   signal chan           : in CL_SEND_S_ISA;
   variable ready        : out bit;
   variable addr_LSB     : in int4;
   variable read_addr    : out bit16;
   signal addr_p         : in bit16;
   signal aen_p          : in bit;
   signal reset_p        : in bit;
   signal ior_p'         : in bit;
   signal iochrdy_p      : out bit;
   signal clk_p          : in bit) is
begin
   wait until clk_p'event and clk_p='1';
   if(aen_p='0' and reset_p='0' and ior_p'='0' and
      addr_p(15 downto addr_lSB) = chan.addr(15
                                 downto addr_LSB) then
      read_addr   := addr_p;      --Read the address
      iochrdy_p   <= '0';         -- Request an extension
      ready       := '1';         -- Receiver's turn
      wait until clk_p'event and clk_p='1';
   else
      ready       := '0';
   end if;
end CL_ReadySendSrvISA;
```

(c)

```
-- Library procedure to read data from the ISA bus after
-- the ReadyRecSrvISA procedure has been called

-- The companion procedure for sending is PutData
-- SrvISA

procedure CL_GetDataSrvISA(
   variable char_received        : out STD_LOGIC8;
   signal data_p                 : out bit;
   signal iochrdy_p              : out bit;
   signal iow_p'                 : in bit;
   signal clk_p                  : in bit) is
begin
   char_received     := data_p;
   iochrdy_p         <= '1';
   wait until clk_p'event and clk_p='1'; and iow_p'='1';
end CL_GetDataSrvISA;
```

(d)

Figure 5: Bus controller (cont'd): (a) GCD process, (b) RSA process, (c) addition to OOCL.

cess then examines the relevant bits, to determine which channel the ISA bus master is addressing, and accordingly routes the data to or from the GCD or RSA process, using the two-phase OOCL bus. The switch statement shows the cases for the GCD process, the ones for the RSA process are identical. The *GetData()* or *PutData()* routines are used for the servant receive and send respectively and have been added to OOCL.

Figure 5(d) shows the *GetData()* procedure, the *PutData()* procedure is identical. Data is read from the ISA bus and after the *iochrdy_p* port is de-asserted, the procedure waits until the cycle is completed. The sentinel approach is used for both the GCD and the RSA computations, for sending the result, as discussed in the previous section. However, in this case a separate sentinel process is not needed, since the bus-controller examines the corresponding global, and accordingly sends either the sentinel or the result.

Figure 4(b) shows the block diagram for the interface. The bus controller process acts as the ISA servant, and co-ordinates the data transfer to and from the other processes on the FPGA module, by acting as the bus master for the two-phase bus. A separate request line is used for each channel on the two-phase bus instead of using the addressed mode. Since each channel is point-to-point, no address is necessary, leading to faster internal transfers. An address would only be necessary if there were very many processes to which the bus controller was interfacing. A brief note on naming conventions: each channel type begins with *CL* for communication library, followed by either *SEND* or *REC* for sender or receiver, followed by either *M* or *S* for either master or servant, followed by the protocol type, such as ISA for the ISA bus protocol, or D8P2 for a custom protocol with 8 data lines and 2-phase handshaking.

Figure 5(a) shows the GCD process. This has already been discussed in Section 3.1. The only difference is that instead of receiving or sending data over the ISA channels to the PC directly, the data transfer to the bus controller takes place over the internal two-phase bus.

Figure 5(b) shows the RSA process. Distinct OOCL servant send and receive channels are declared and initialized. Next the public keys are received from the bus controller, after which the RSA process enters into a loop. The RSA global is set to zero, after which raw data is received over the *rsa_rec_chan*. This data is encoded and then sent back over the *rsa_send_chan*. Again, it is assumed that the relevant operators have been overloaded to support the *STD_LOGIC* data types.

To summarize, in addition to supporting the initialization and the send and receive methods, the OOCL servant channels have been enhanced with routines that determine if the channel is ready to send or receive data, and with other routines that then proceed to complete the transfer. This enables the OOCL methodology to be flexible enough to be applied in a multi-tasking environment in both hardware and software.

4 Conclusions

OOCL bridges the gap between the abstract message passing primitives desired by the user, and the underlying communication implementation. It provides a consistency across diverse protocols and hardware-software components. Because OOCL is based on libraries, no additional tools are needed to generate the interface, and no language extensions are necessary. In addition to supporting the initialization, send and receive routines, the OOCL servant channels have been enhanced with routines that determine if the channel is ready to send or receive data, and with other routines that then proceed to complete the transfer. This enables the OOCL methodology to be applied to multiprocessing examples. In this paper, we demonstrated the OOCL routines for the PC ISA bus, and showed that the routines support communication for two reasonably complex communication examples.

The current library includes C libraries for the PC ISA bus transfer, interrupt-driven PC serial port transfer, and handshaked PC serial port transfer; C libraries for the Intel 8051 for PC serial port transfer, I2C bus transfer, and custom protocols for communication with other 8051's or with custom hardware components; and VHDL packages for PC ISA bus transfer and custom protocols. Future directions include expanding the library to include other common protocols, and investigating automated generation of the library routine implementations to simplify library development and porting to new architectures.

References

[1] P. Chou, R. Ortega, and G. Borriello, "Interface co-synthesis techniques for embedded systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 280–287, 1995.

[2] S. Vercauteren, B. Lin, and H. D. Man, "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proceedings of the Design Automation Conference*, 1996.

[3] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[4] D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung, "System design methodologies: Aiming at the 100 h design cycle," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 4, no. 1, pp. 70–82, 1996.

[5] S. Narayan and D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Proceedings of the 32nd Design Automation Conference*, pp. 468–473, 1995.

[6] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[7] L. S. Eggebrecht, *Interfacing to the IBM Personal Computer*. Sams, second ed., 1990.