Automated Composition of Hardware Components

James Smith
Stanford University
Gates Building, Room 326
Stanford, California 94305−9030
redskins@aglaia.stanford.edu

Giovanni De Micheli
Stanford University
Gates Building, Room 333
Stanford, California 94305−9030
nanni@galileo.stanford.edu

Abstract

In order to automate design reuse, methods for composing system components must be developed. The goal of this research is to automate the process of generating interfaces between hardware subsystems. The algorithms presented here can be used to generate a cycle−accurate, synchronous interface between two hardware subsystems given an HDL model of each subsystem. These algorithms have been implemented in the POLARIS *hardware composition tool and have been used to generate an interface between a MIPS microprocessor and the SRAM that comprises its secondary cache. Interface generation for the MIPS R4000 is described.*

1. Introduction

The increasing complexity of electronic systems is forcing designers to consider, if not implement, design reuse and intellectual property sharing. As this methodology matures, a new breed of tools will be required to automate component selection, subsystem scheduling, and system composition. This paper presents a mechanism for composing hardware blocks that communicate with different protocols, given an HDL description of these protocols, while providing hooks for implementing arbitration algorithms.

Interface synthesis has focused on optimizing high level communication between subsystems given a set of communication constraints ([ErHeBe93], [JeElOb94], [KaLe94]). Interface modeling languages such as that developed by [ObKuHe96] allow a designer to explore a interface design space and generate a synthesizable description of the interface. [GuRo94] describes a methodology for modeling an interface with a behavioral description suitable for high level synthesis. Other interface synthesis research, such as that performed in [GaGl96], has investigated specifying and scheduling communication between hardware and software subsystems. The research presented here focuses on generating a low−level, synthesizable description of synchronous interfaces between hardware components. Similarly, [ChOrBo95] describes a mechanism for creating the glue logic between two hardware components, but requires a functional description of component ports. In this paper, we present the POLARIS tool, which converts a subsystem's communication protocol into a standard scheme given an HDL description of the subsystem. An HDL model of the resulting interface is generated.

The basic purpose of an interface is to facilitate the movement of data. Data could be addresses, commands, values destined for a memory location, or some combination of these descriptions. In order to allow hardware subsystems that follow different protocols for moving data to communicate with one another, the tool presented here maps these protocols into a standard communication scheme. This scheme is then implemented in an interface architecture that is general enough to accomodate the requirements of any target interface. The terminology *client* is used in this paper to indicate a component that is sending data and *resource* indicates a component that is receiving data.

2. Overview

The algorithms presented here are used to generate synchronous component interfaces. The components may operate at different frequencies and may employ unidirectional or bidirectional busses. Bidirectional busses, such as those employed by PCI or VME, are handled by treating the bus as two unidirectional busses and combining the resulting interface controllers. Multi−way interfaces that allow multiple clients to interact with multiple resources are synthesized by dynamically establishing a point to point link between the client and the resource (Fig. 1).

The interface architecture is described in Section 3. From the component model described in Section 4.1, a state machine is synthesized to map the component's communication protocol into a standard protocol that other interfaces can understand (Section 4.2). The data formats that a client component employs are translated into formats that the resource component can understand (Section 4.3).

The POLARIS hardware composition tool requires the user to supply an HDL description of the component being interfaced to, the name(s) of the ports across which data is tranferred, and the names of ports that the interface does not have access to (*uncontrollable* ports). The names of the uncontrollable ports (e.g. *reset*) must be supplied so that POLARIS does not manipulate these ports when creating a component interface.

3. The Interface Architecture

Unlike the hardware interface synthesis research performed by [MaHa95], this research links hardware components through a standard architecture rather than by attempting to map one component interface into another. This allows interfaces to be synthesized for a broad range of components. In addition, it allows multiple components to be linked via the same interface.

3.1 Architectural Blocks

The interface architecture includes a state machine for protocol conversion, a send and receive buffer for transaction information (which must be saved while a resource is unavailable), and an arbiter to govern access to a resource (Fig. 1). Although hooks are provided to allow the implementation of an optimized arbitration algorithm, the details of the arbitration scheme are not required for interface synthesis.

When the state machine for protocol conversion detects that a component is sending data, the data is placed in the receive buffer for that interface. This data will be passed to the send buffer for the resource interface. Correspondingly, when the state machine detects that there is data in its send buffer, it executes the necessary signal assignments to transfer the data to the resource component. Sizing of these queues is currently a manual task, but could be automated in the

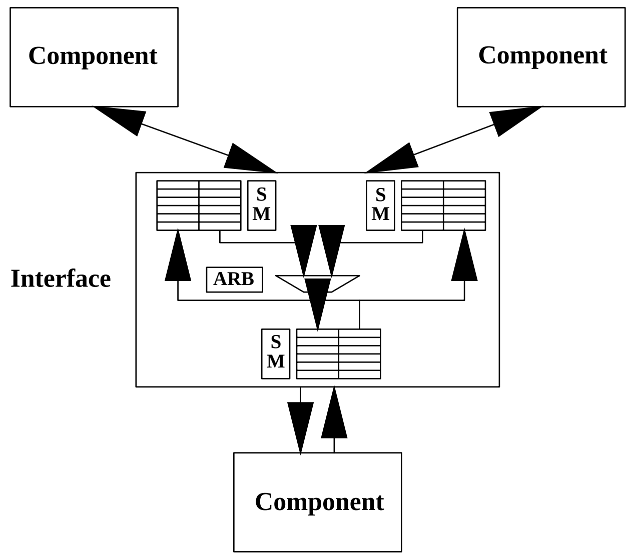

Fig. 1: High level view of a three component implementation of the interface architecture.

manner of [AmBo91]. The default arbiter implements a round robin arbitration scheme to select a client receive buffer that will transfer its data into the appropriate resource send buffer.

3.2 Communication Scheme

The standard protocol employs four control signals each for the receive buffer and the send buffer (Fig. 2). For the each buffer, the signals are implemented as follows: (1) *Request* − input to the buffer controller that indicates data is being sent to the interface; (2) *Stall* − output from the buffer controller that indicates that the buffer is full (the state machine must prevent other components or interfaces from sending any more data); (3) *Valid* − output from the buffer controller that indicates that valid data is in the buffer and ready to be transferred; (4) *Acknowledge* − input to the buffer controller that indicates that data has been read from the buffer (the buffer controller will increment the read address). Resources are scheduled in the arbiter by selectively *Acknowledge*ing the client *Valid* signal. Data is transferred through four unidirectional busses, two for sending data to and receiving data from the arbiter, and two for sending data to and receiving data from the external component.

4. State Machine Generation

Given a component model that describes bus functionality (or a superset of bus functionality), conditions for transferring data to or from that component are determined. A sequence(s) of assignments to component ports is determined that will cause these conditions to become true. After the required assignments to component ports have been determined, they are executed on the component model, so as to resolve the values of control ports that are inputs to the synthesized interface. Once the values of all necessary input and output ports have been resolved, a state machine is generated that executes the required assignments and monitors the necessary control ports. This state machine provides a mapping from the communication protocol for a system component to the standard protocol that allows inter−component communication. The collection of operations necessary to perform a data transfer is referred to as a *function*.

4.1 Component Model

The component is abstracted as a list of assignments to

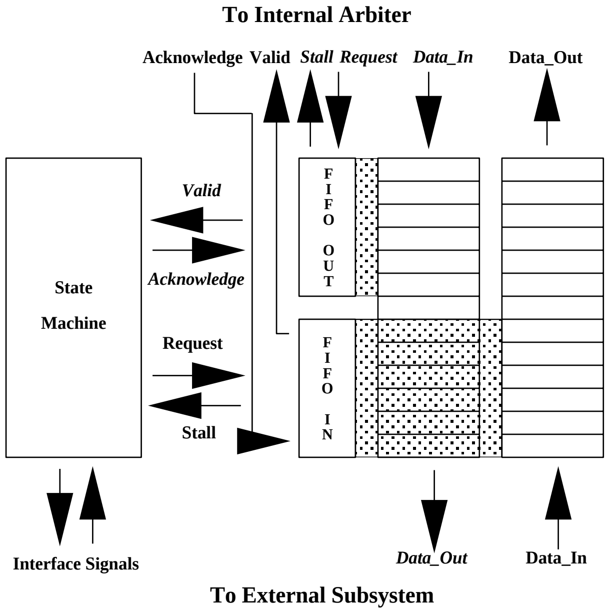

Fig. 2: Interface communication scheme.

variables and conditions for each assignment to be executed.

Definition 1. A *component* is described by the list of tuples $\{C_i := <\nu_i, X_i, A_i, \sigma_i>\}$ where $X_i = \{\chi_{ij}\}$ are the conditions under which the values $A_i = \{\alpha_{ij}\}$ are assigned to the variable ν_i. For the assignment to ν_i, we assume there are n_i possible values $\{\alpha_{ij}$, j= 1,2, ... , $n_i\}$ each selected by one and only one condition $\chi_{ij} \varepsilon \{0, 1\}$, and σ_i indicates whether the assignment is combinational or synchronous. Thus,

$$\nu_i = \Sigma\, \chi_{ij} * \alpha_{ij}.$$

Variables that are not component ports are referred to as *internal* variables.

4.2 Protocol Conversion Algorithm

The state machine for protocol conversion is represented, analogous to the Moore model, as a collection of state assignments and conditions for state transitions.

Definition 2. The *state machine for protocol conversion* is described by the tuple $SM := <S, T>$ where

$S := \{S_i\}$ is a list of states,
$T := \{(\tau_{ij} = 1) => S_i -> S_j\}$ is a list of conditions governing state transitions.

The algorithm for generating the state machine for protocol conversion is completed in five steps: (1) generate a sequence S_f of *functional states* that cause a function to be executed, (2) generate a sequence S_x of *exit states* that cause an executing function to be halted ($S = S_f \cup S_x$), (3) generate the conditions $\{\tau_{ij}\}$ that govern the state transitions, (4) combine state sequences for multiple functions, and (5) reduce the number of states. The name of the component's data bus, referred to as the *target* variable, is supplied to initiate generation of the interface state machine. The algorithm is outlined in Fig. 3.

If the target variable is an input or bidirectional port, the component model is searched for a use of this variable such as:

```
if signalA = valueA then
        signal <= target

if target = value and
  signalB <= valueB then ...
```

The conditions under which the target variable (*target*) is used are the *zeroth order input conditions*.

Definition 3. The set of *zeroth order input conditions* for the target variable ν_x is $I_0 := \{\chi_{ij}\}$ where χ_{ij} or α_{ij} is dependent on ν_x.

In the previous example, $\nu_x = \{target\}$ and $I_0 = \{signalA = valueA, signalB = valueB\}$

If the target variable is an output or bidirectional port, the component model is searched for an assignment to this port such as:

```
if signalA = valueA then
                target <= signal
```

The conditions for these assignments are the *zeroth order output conditions.*

Definition 4. The set of *zeroth order output conditions* for the target variable ν_x is $O_0 := \{\chi_{xj}\}$.

In the previous example, $\nu_x = \{target\}$ and $O_0 = \{signalA = valueA\}$.

4.2.1 Executing a Function

Assignments that satisfy zeroth order conditions must be executed as part of the functional state sequence. That is, if $I_0 = \{signalA = valueA\}$, then state S_0 must contain the assignments $\{signalA <= valueA\}$. If *signalA* is a component port, then a single assignment can be made to allow that function to be completed. However, if *signalA* is an internal variable of the component, then conditions must be determined that will cause that variable assignment. For example, if a component description contained the sequential statements:

```
if externalSignalA = valueA and
    internalSignal = value then
                signal <= target

if externalSignalB = valueB then
                internalSignal <= value
```

then the following state sequence is generated:

State 1: *extSignalB* <= *valueB*
/* causes *intSignal* <= *value* in State 0 */

State 0: *extSignalA* <= *valueA*

These conditions that must be satisfied to cause the internal variable assignment are the *nth order conditions*.

Definition 5. The set of *nth order input conditions* for the target variable ν_x is $I_n := \{\chi_{kl}\}$ where, given the (n−1)th order input conditions for ν_x, i.e. I_{n-1},
there exists an $\chi_{ij} \varepsilon I_{n-1}$
such that $(\chi_{kl} = 1) => (\chi_{ij} = 1)$.

In the previous example, $\nu_x = \{target\}$, $I_0 = \{externalSignalA = valueA \text{ and } internalSignal = value\}$ and $I_1 = \{externalSignalB = valueB\}$. An analogous definition applies to *nth order output conditions*. If conditions of all orders can be satisfied by a sequence of assignments to component ports, the interface state machine can deterministically drive a component into functional states.

If an internal variable assignment requires an nth order condition that in turn requires the same assignment, the variable assignment is *uncontrollable* and can not be deterministically driven to that value. For example, if a component description contained the sequential statements:

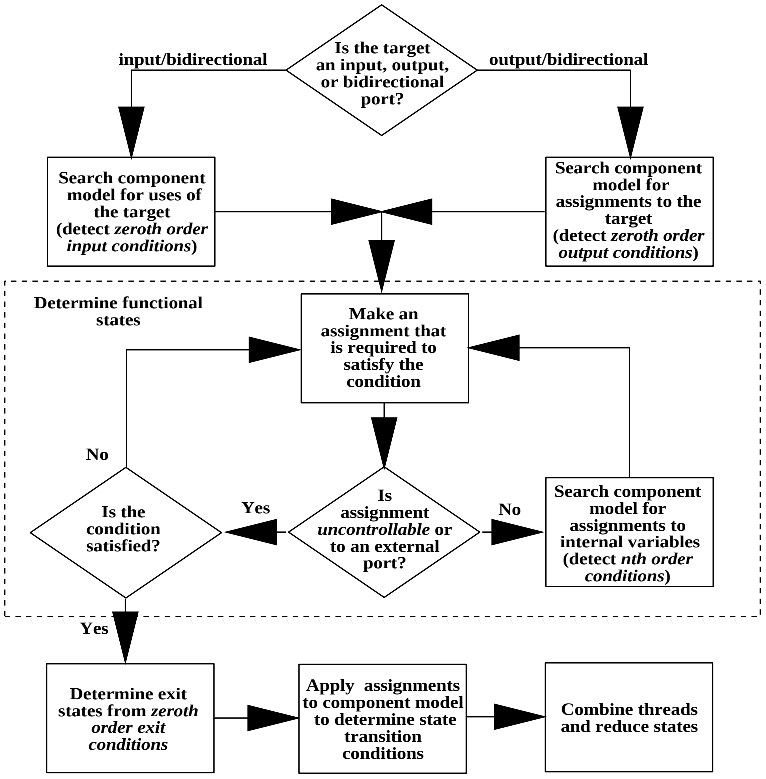
Is the target an input, output, or bidirectional port?
input/bidirectional
output/bidirectional
Search component model for uses of the target (detect zeroth order input conditions)
Search component model for assignments to the target (detect zeroth order output conditions)
Determine functional states
Make an assignment that is required to satisfy the condition
No
Is the condition satisfied?
Yes
Is assignment uncontrollable or to an external port?
No
Search component model for assignments to internal variables (detect nth order conditions)
Yes
Determine exit states from zeroth order exit conditions
Apply assignments to component model to determine state transition conditions
Combine threads and reduce states

Fig. 3: The algorithm for state machine generation

```
if reset = TRUE or intSignalA = valueA then
            target <= value
            intSignalB <= valueB,

if intSignalB = valueB then
            intSignalA <= valueA,
```

then *intSignalA* <= *valueA* is uncontrollable if *reset* is uncontrollable.

Definition 6. An internal variable assignment $(\nu_k = \alpha_{kl})$ is *uncontrollable* if $\chi_{kl} \varepsilon I_n$ and
(1) $\chi_{kl} \varepsilon I_m$ where m < n, or
(2) χ_{kl} is dependent on another uncontrollable assignment.

If an uncontrollable internal variable assignment is encountered, no nth order conditions for that assignment are generated. The thread of states that required the assignment is thus aborted (Fig. 3). This prevents the algorithm from looping indefinitely on a component description such as the one described in the previous example.

A component can not be deterministically driven to a functional state when an uncontrollable signal is encountered. However, the functional states can be detected by examining component control signals (Section 4.2.3).

4.2.2 Exiting a Function

In the same way that a sequence of states is determined to execute a function, another sequence of states is determined to end that function. That is, when the data transfer is completed, the interface must exit its corresponding data transfer state.

A component exits a data transfer state when an assignment is made that contradicts a zeroth order condition. This can be achieved with assignments to component ports or internal variables. For example, if a component description contained the statement:

```
if externalSignalA = valueA and
    internalSignal = value then
                signal <= target
```

then satisfying the condition *not (externalSignalA = valueA and internalSignal = value)* must cause the data transfer state in the interface to be exited. Such conditions are the *zeroth order exit conditions*.

Definition 7. The set of *zeroth order exit conditions* for the target variable ν_x is $E_0 := \{\chi_{kl}\}$ where, given the zeroth order input (or output) conditions for ν_x, i.e. I_0,
there exists an $\chi_{xj} \varepsilon I_0$,
such that $(\chi_{kl} = 1) \Rightarrow (\chi_{xj} = 0)$.

Nth order exit conditions are generated and satisfied in the same manner that nth order input and output conditions are generated and satisfied.

The exit state sequence is combined with the data transfer state sequence to obtain a state sequence that can execute a function and be reset.

4.2.3 Generating State Machine Conditions

After the conditions for executing and exiting a function have been completely satisfied or found to be uncontrollable, the required assignments are executed on the component model. If the component drives a port to a valid value in a cycle, then a conditional statement governing the state transition must be added to the state corresponding to that cycle. For example, if the component description contained the sequential statements:

```
if externalSignalA = valueA and
    internalSignal = value then
                signal <= target

if uncontrollableSignal = valueX and
    externalSignalB = valueB then
                internalSignal <= value
                externalSignalC <= valueC
```

then the condition τ_{10} = {*externalSignalC = valueC*} must be satisfied to allow the state transition from state S_1 = {*externalSignalB <= valueB*} to state S_0 = {*externalSignalA <= valueA*}.

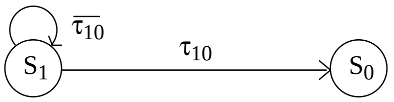

Fig. 4: Conditions governing state transitions.

Thus, if the interface state machine can not deterministically drive a component into a particular state (in the example above, the component state corresponding to S_1), it will be able to determine when the component has reached that state by evaluating its status signals.

4.2.4 Combining Multiple Threads of Execution

A function's execution may require or allow more than one sequence to be executed in parallel. For example, the component description,

```
if internalSignalA = valueA and
   internalSignalB = valueB then
                signal <= target
```

requires multiple variables (*internalSignalA*, *internalSignalB*) to be set in tandem. This results in multiple threads of execution being generated, which must be combined into a single state machine.

ANDing two state sequences is a straightforward combination of state assignments and state transition conditions. The final states in a sequence and their predecessors are ANDed. An example is shown in Fig. 5.

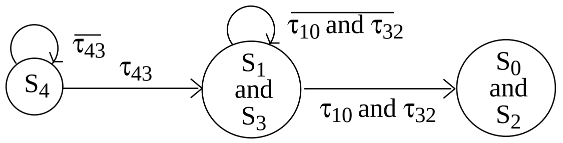

Fig. 5: The results of ANDing the two state sequences (S_0, S_1) and (S_2, S_3, S_4).

If any two ANDed states contain contradictory assignments the threads are discarded.

In ORing two threads, only the head states are combined. Previous states are not ORed to prevent state sequences that contain a portion of each thread from being executed. An example is shown in Fig 6.

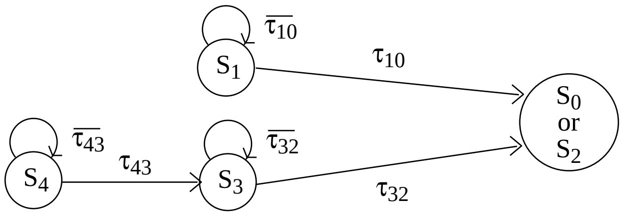

Fig. 6: The results of ORing the the two state sequences (S_0, S_1) and (S_2, S_3, S_4).

Duplicated states are removed as shown in the next section.

4.2.5 State Reduction

A component will frequently share states between execution threads. For example, a component may contain a state in which it polls its subsystems for writes, and executes a different sequence for each write:

```
if state = POLL_STATE then
            case (subsystem)
            subA_write:
                    state <= subA_WRITE_STATE
            subB_write:
                    state <= subB_WRITE_STATE
```

The number of states is reduced by joining threads of execution at points where their respective states are *congruent*.

There are two sets of requirements that states can satisfy to be considered congruent. First, two states are congruent if they stem from congruent previous states and the conditions for entry into both states are the same. Second, two states are considered to be congruent if they contain the same variable assignments and one of them is an exit state. The second requirement allows a state machine to be reset once it has executed a functional sequence.

Definition 8. States S_i and S_j are *congruent* if either
(1) for all x, there exists a y
such that $\tau_{xi} = \tau_{yj}$ and
and S_x and S_y are congruent, or
(2) assignments of S_i = assignments of S_j
and S_i satisfies E_0.

```
module r4600_interface (SysADi, SysADCi, SysCmdi, SysCmdPi,
          SysADo, SysADCo, SysCmdo, SysCmdPo,
          RdRdy, WrRdy, ExtRqst, Release, ValidIn, ValidOut,
          Reset, Clock, SysOe,
          CPUwr, CPUrd, CPUrsp, CPUdataI, CPUdataO, CPUack, CPUvalid, CPUaddr);

 input [63:0] SysADi;
 input [7:0] SysADCi;
 input [8:0] SysCmdi;
 input SysCmdPi;
 output [63:0] SysADo;
 output [7:0] SysADCo;
 output [8:0] SysCmdo;
 output SysCmdPo;
 input RdRdy, WrRdy;
 input ExtRqst;
 output Release;
 input ValidIn;
 output ValidOut;
 input Reset;
 input Clock;
 output SysOe;
 input CPUwr;
 input CPUrd;

 input CPUrsp;
 input [63:0] CPUdataI;
 output [63:0] CPUdataO;
 output CPUack;
 output CPUvalid;
 input [63:0] CPUaddr;
 reg [63:0] SysADo;
 reg [7:0] SysADCo;
 reg [8:0] SysCmdo;
 reg SysCmdPo;
 reg Release;
 reg ValidOut;
 reg SysOe;
 reg [63:0] CPUdataO;
 reg CPUack;
 reg CPUvalid;
 reg [3:0] bus_state;

 parameter S_SEND_IDLE = 4'b0000, S_SEND_ADDR = 4'b0001, S_SEND_DATA = 4'b0010,
        S_SEND_SPIN = 4'b0011, S_RECV = 4'b0100, S_RECV_SPIN = 4'b0111;

 parameter H_READ = 2'b00, H_WRITE = 2'b01, H_NULL = 2'b10;

always @ (posedge Clock) begin
  ValidOut = 0;
  Release = 0;
  SysOet = 1;
  CPUack = 0;
  if (Reset) begin
    bus_state = S_SEND_IDLE;
  end else begin
    case (bus_state)
      S_SEND_IDLE: begin
        if ((CPUwr && WrRdy) || (CPUrd && RdRdy)) begin
          bus_state = S_SEND_ADDR;
        end else if (ExtRqst) begin
          Release = 1;
          bus_state  = S_RECV_SPIN;
          SysOe = 0;
        end
      end
      S_SEND_ADDR: begin
        if (CPUwr) begin
          SysCmdo = 9'b001000000;
          bus_state = S_SEND_DATA;
        end else if (CPUrd) begin
          SysCmdo = 9'b000000000;
          bus_state = S_SEND_IDLE;
        end
      end
      S_SEND_DATA: begin
        SysADo = CPUdataI;
        if (!CPUwr) begin
          SysCmdo = 9'b100000000;
          bus_state = S_SEND_IDLE;
        end else begin
          SysCmdo = 9'b110000000;
        end
        ValidOut = 1;
        CPUack = 1;
      end
      S_SEND_SPIN: begin
        SysOe = 0;
        bus_state = S_SEND_IDLE;
      end
      S_RECV: begin
        SysOe = 0;
        if (ValidIn) begin
          if (SysCmdi[8]) begin
            CPUdataO = SysADi;
          end else case (SysCmdi[6:5])
            H_NULL:
              bus_state = S_SEND_SPIN;
          endcase
        end
      end
      S_RECV_SPIN: begin
        SysOe = 0;
        bus_state = S_RECV;
      end
    endcase
  end
 end
endmodule
```

Fig. 7: Verilog description of a simple MIPS SysAD interface (bidirectional pads for SysAD not shown).

Congruent states are combined by creating a state in which the assignments of both states are executed. If one of the states is an exit state, then the conditions for entrance into the newly created state are ORed.

4.3 Datapath Translation

The state machine for protocol conversion allows two components to communicate by translating their control signals into the standard interface. However, components often employ different datapaths that must be reconciled if they are to communicate with one another. Translating datapaths between interfaces imposes two requirements: (1) datapath widths must be reconciled, and (2) addresses must be extracted from a transaction so that client data can be directed to the appropriate resource.

Datapath widths are determined from the component description. When data is read from a client receive buffer, it is read into a register that is the width of the resource datapath. If the resource datapath is wider than the client datapath, an *Acknowledge* is returned to the client interface for every word that is popped off the client receive buffer. If the resource datapath is thinner than the client datapath, an *Acknowledge* is returned to the client interface when the resource register is filled.

Address extraction is currently a manual task. It can be performed using a simplified version of the structures that ([ChOrBo95], [MaHa95]) used to achieve interface synthesis. These two employed structures called "signal sequences" (SEQs) and "protocol flow graphs" (PFGs), respectively, to model the bit patterns that are required to interact with a component. In the case of POLARIS, a sequence of higher level descriptions can be provided to allow the interface to detect which cycles or bits are transmitting an address.

5. Example − Simple MIPS SysAD Interface

This example demonstrates how POLARIS generates a state machine that converts the communication protocol of a MIPS processor with a simplified SysAD interface to the standard protocol. This generated interface can communicate with a similarly synthesized interface for another component such as RAM, a DSP, etc. (not described here). The HDL model of the SysAD interface (not including the bidirectional buffer for SysAD) is given in Fig. 7. In addition to the HDL model, the bus for data transferral (*SysAD*) and a list of uncontrollable ports (Reset, *CPUwr, CPUrd, etc.*) is supplied to the tool.

Upon determining that the target variable (*SysAD*) is a bidirectional port, POLARIS first creates the state machine for input through the target variable. The component model is searched for the for the assignment * <= *SysAD*. The set of zeroth order input conditions I_0 = {*Reset = 0 and bus_state = S_RECV and Valid_In = 1 and SysCmdi[8] = 1*} is returned. State S_0 = {*Reset = 0, bus_state = S_RECV, Valid_In = 1, SysCmdi[8] = 1*} is created. Since *bus_state* is an internal variable and not uncontrollable, it is made the new target bus and the component model is searched for the assignment *bus_state <= S_RECV*. The set of first order input conditions I_1 = {*Reset = 0 and bus_state = S_RECV_SPIN*} is returned. State S_1 = {*Reset = 0, bus_state = S_RECV_SPIN*} is created, and the component model is searched for the assignment *bus_state <= S_RECV_SPIN*. The set of second order input conditions I_2 = {*Reset = 0 and bus_state = S_SEND_IDLE and ((CPUwr and WrRdy) or (CPUrd and RdRdy)) = 0 and ExtRqst = 1*} is returned and states S_2 = {*Reset = 0, bus_state = S_SEND_IDLE, WrRdy = 0, ExtRqst = 1*} and S_3 = {*Reset = 0, bus_state = S_SEND_IDLE, RdRdy = 0, and ExtRqst = 1*} are created (note that CPUwr and CPUrd are uncontrollable and states corresponding theirassignment are not generated). This process continues until nth order conditions can be completely controlled from external ports or require uncontrollable assignments (e.g. *Reset = 0, bus_state = S_SEND_IDLE*).

The zeroth order exit conditions are determined by negating the zeroth order input conditions. The set of zeroth order exit conditions is E_0 = {*ValidIn = 0 or SysCmd[8] = 0 or bus_state != S_RECV or Reset = 1*}. This exit condition generates three exit states, two of which satisfy the first congruency criteria ({*ValidIn=0*} and {*SysCmd[8] = 0*}).

Now that all possible complete sequences of states have been determined, the state assignments are executed to determine the valid values on external ports during each state. Since *Release* and *ValidOut* are driven to valid values during state S_1, condition τ_{10} = (*Release = 0 and ValidOut = 0*) must

```
if (Reset) begin
   state = S_0;
end else begin
   case (state)

      S_3: begin
         if (valid)
            state = S_2;
      end;

      S_2: begin
         WrRdy = 0;
         RdRdy = 0;
         ExtRqst = 1;
          if (Release && !ValidOut)
            state = S_1;
      end;

      S_1: begin
         if (!Release && !ValidOut)
            state = S_0;
      end;

      S_0: begin
         if (valid) begin
            ValidIn = 1;
            SysCmd[8] = 1;
            SysAD = Data_Out;
            acknowledge = 1;
         end else  begin
            if (...) begin
               ValidIn = 1;
               SysCmd[8] = 1;
               SysCmd[6:5] = H_NULL;
               state = S_3;
            end else begin
               ValidIn = 0;
               SysCmd[8] = 0;
            end;
         end;
      end;

   endcase;
end;
```

Fig. 8: Resulting state machine that allows writes to the MIPS core through the SysAD bus.

be satisfied to complete the transition $S_1 \rightarrow S_0$. Similarly, $\tau_{21} = \tau_{31}$ = (*Release = 1 and ValidOut = 0*) must be satisfied to complete the transitions $S_2 \rightarrow S_1$ and $S_3 \rightarrow S_1$.

Given the assignments for each state S and the conditions T for transitioning between states, the state sequences are joined at congruent states. The three sequences generated by the exit states all contain congruent states leading up to the functional state. Thus, the threads are joined at each of these states. However, the exit states that perform {*ValidIn = 0*} and {*ValidIn = 1, SysCmd[8] = 0, SysCmd[6:5] = H_NULL*} are mutually exclusive, thus there are multiple branches exiting the functional state. According to the second congruency criteria, the exit state from {*ValidIn = 0*} is congruent to state S_0 and the exit state from {*ValidIn = 1, SysCmd[8] = 0, SysCmd[6:5] = H_NULL*} is congruent to state S_3. The state machine for transferring data to the MIPS processor is given in Fig. 8.

The process above is repeated for data transferred from the MIPS processor to the interface. The only difference being that, initially, the functional states are determined by searching the expression list for *SysAD* <= *. The output state machine and input state machine are joined at states S_2 and S_3 in state reduction. The physical characteristics of the synthesized interface, not including the send and receive buffers, are shown in Fig. 9. The speed, area and power consumption of the bus control logic synthesized above is not optimized in this research because this logic is rarely a significant factor in the overall speed, area and power consumption of the interface.

6. Conclusion

Reusing existing high level blocks is becoming a necessity for designing complex systems. Composing blocks that are developed by different design groups with different communication protocols is an imperative in automating design reuse and IP sharing. The composition architecture presented here provides a means to compose synchronous blocks while providing hooks for optimizing system performance by prioritizing component communication. The example provided illustrates its ability to generate an interface between hardware blocks.

The techniques described above allow a designer to automatically generate an HDL model of an interface between two or more blocks given an HDL description of the corresponding blocks. In implementing these techniques, parsing of the input HDL has been completed in a front end module that can be adapted to different coding styles or represenatations. While overspecified input HDL descriptions can be used to geneate an interface, incompletely specified input blocks will yield interfaces with ambiguous states. However, these states are detectable. Interfaces between blocks with multiple busses can be generated when

Process	LSI LCB007
Gate Count	1464
Max. Operating Frequency	166 MHz
Est. Area	.92 mm^2
Est. Power Consumption	348mW

Fig. 9: Physical Charactersistics of the Automatically Generated Interface to a MIPS R4000 (Does not include 256B receive and send buffers)

the control of these busses is separate. Extensions for blocks that contain multiple busses with shred control is a future topic of research.

Future research will seek to expand the techniques presented here to generate interfaces between software and hardware components in a system that implements memory mapped I/O. The communication protocol for software interfaces is restricted by the instruction set architecture of the microprocessor on which the software is running. Instead of communicating with a hardware subsystem by assigning values to and reading values from ports, the software driver communicates by writing and reading hardware registers. Thus, the hardware/software composition requires the interface state machine generator described above and another layer, similar to [BoDeLi96], that encapsulates port assignments in microprocessor operations.

Acknowledgements

This research is sponsored by ARPA under grant DATB63−95−C0049.

References

[ErHeBe93] R. Ernst, J. Henckel, and T. Benner, "Hardware/Software Cosynthesis for Microcontrollers", *IEEE Design and Test of Computers*, p. 64−75, December 1993.

[JaElOb94] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, and H. Tenhunen, "Hardware/Software Partitioning and Minimizing Memory Interface Traffic", *Proceedings of EuroDac*, p. 226−231, 1994.

[KaLe94] A. Kalavade and E.A. Lee, "A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem", *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, p. 42−48, 1994.

[ObKuHe96] Johnny Oberg, Anshul Kumar, Ahmed Hemani, "Grammar−based Hardware Synthesis of Data Communication Protocols", *9th International Symposium on System Synthesis*, p. 14−19, 1996.

[GuRo94] P. Gutberlet, W. Rosenstiel, "Specification of Interface Components for Synchronous Data Paths", *7th International Symposium on System Synthesis*, p. 134−139, 1994.

[GaGl96] Michael Gasteier, Manfred Glesner, "Bus−Based Communication Synthesis on System−Level", *9th International Symposium on System Synthesis*, p. 65−70, 1996.

[ChOrBo95] Pai Chou, Ross B. Ortega, Gaetano Borriello, "Interface Co−Synthesis Techniques for Embedded Systems", *Proceedings of the IEEE/ACM International Conference on Computer−Aided Design*, pp.280−287, 1995.

[MaHa95] Jan Madsen and Bjarne Hald, "An Approach to Interface Synthesis", *8th International Symposium on System Synthesis*, p. 16−21, 1995.

[AmBo91] T. Amon and G. Borriello, "Sizing Synchronization Queues: A Case Study in Higher Level Synthesis", *Proceedings of the 28th Design Automation Conference*, Jun 1991.

[BoDeLi96] I. Bolsens, H. DeMan, B. Lin, K. Van Rompaey, S. Vercauteren, D. Verkest, "Hardware−Software Codesign of Digital Telelcommunication Systems", Proceedings of the IEEE, p 391−418, March 1997.