Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems[*]

Pai Chou Ross Ortega Gaetano Borriello

Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

Abstract

Microcontroller-based systems require the design of a hardware/software interface that enables software running on the microcontroller to control external devices. This interface consists of the sequential logic that physically connects the devices to the microcontroller and the software drivers that allow code to access the device functions. This paper presents a method for automatically synthesizing this hardware/software interface using a recursive algorithm. Practical examples are used to demonstrate the utility of the method and results indicate that the synthesized circuit and driver code are comparable to that generated by human designers. This new tool will be used by higher-level synthesis tools to evaluate partitionings of a system between hardware and software components.

1 Introduction

Microcontrollers are microprocessors with integrated general-purpose interfacing logic to facilitate the control of peripheral devices. This logic is encapsulated in I/O ports (i.e., collections of I/O pins and related interface logic) that can be written to or read from by program code. Microcontrollers are commonly used to implement digital control systems because they require minimal interfacing hardware. Peripheral devices range from user input and display units to memories and communication interfaces.

Designers of microcontroller-based systems must allocate devices to ports and customize subroutines to access the devices under the specific port assignment chosen. There are several issues in performing this task. Most importantly, additional hardware must be minimized so as to maintain the benefits of using a microcontroller, namely reducing part count. If there are more device pins than port pins, then the ports need to be time-multiplexed with the corresponding de-multiplexing performed in external hardware. Another important goal is to minimize code size. Exploiting the grouping of signals can reduce the number of instructions needed in the driver routines. In certain cases, memory-mapped I/O techniques are used to access the devices with the processor's address and data bus thereby bypassing most of the port interface logic. Since the address and data busses also use I/O pins to reach the devices, this further complicates the port assignment problem.

In this paper, we describe a tool that automates the synthesis of the hardware/software interface between a microcontroller and the devices it controls. Optimizations of the synthesis process focus primarily on the minimization of interface hardware and also the code size of the device driver routines. Section 2 specifies this interface synthesis problem. Section 3 describes the data structures used to represent the microcontroller, devices, and hardware and software primitives. Section 4 presents the port allocation algorithm. Section 5 highlights the features of this interface synthesis tool with two practical examples. Section 6 concludes the paper with an evaluation and discussion of future work.

2 Problem Specification

The input to the port allocation tool consists of a behavioral description of the system, peripheral device descriptions, and a microcontroller description. The ouput of the tool is a netlist, customized driver routines, and any necessary interface components.

The behavioral description is a high-level, imperative language program written by the user describing the necessary components of the circuit and its functionality. This program has a declarative section and an operational section. The declarative section allocates static storage for data and instantiates peripheral devices. The operational section computes functions and communicates with the peripheral devices via driver calls. We assume that the user program is single-threaded. This eliminates synchroniza-

[*]This work was supported by NSF under Grant MIP-8858782, DARPA under contract N00014-J-91-4041 , and Patricia Roberts Harris Fellowship.

tion as an issue and simplifies code generation.

The peripheral device description contains a list of device ports (collections of pins) and driver routines. The device ports must be connected directly or indirectly to either a microcontroller port or a power rail. The driver routines describe a sequence of microcontroller-independent actions required to communicate with the device. In this paper, we only consider communication protocols that rely solely on sequencing rather than exact timing relationships between signalling events. By only considering sequencing we can modify driver routines to set up interface hardware while preserving the correctness of the communication protocol.

The microcontroller description contains a list of controller ports and a set of port instructions. The only means the microcontroller has of communicating with peripheral devices is through its ports. The port instructions define the semantics of these external communications. A port may be input only, output only, or bidirectional. A port may also be partially addressable. For example, an 8-bit port may be written in either byte-mode or bit-mode if such instructions are available. A single instruction may alter the values of several ports. For example, a memory access will not only place data on a port, but also drive the address port and read/write enable ports.

The tool outputs a netlist which includes the microcontroller, devices, and any necessary interface hardware. Also, the tool automatically specializes the device access routines for the port assignment selected.

3 Representation: Data Structures

The information regarding the interface capabilities of the microcontroller and the interface requirements of the devices are stored in a class-instance data structure. An object-oriented management system creates and maintains the attributes for classes and instances of devices, controllers, and interface logic.

3.1 Device Representation

The device class attributes are stored in the device library. The attributes include the device port list, static pin rules (SPRs), and sequences (SEQs). The device port list contains all of the device ports that must be connected to controller ports. The SPRs represent hardware and the SEQs represent software. Figure 1 shows the representation for an LCD.

The SPRs capture the activation conditions for the device ports. They are essentially *guarded commands* in the form $X \rightarrow Y$ [3]. X is the guard, or a Boolean expression in terms of device ports. The Boolean terms may include edge-triggered conditions. For example, $x+$ means “rising

```
device Lcd-device(db, rs, rw, e)
    inout [7:0] db;
    input rs, rw, e
{   // *** Registers for SPRs ***
    reg [7:0] lcdReg;
    // *** SPRs ***
    !rw & e -> lcdReg := db;
    rw & e -> db := lcdReg;
    // *** SEQs ***
    seq Lcd-init ()
    {   e:= 0;
        par { rs := 1; rw := 0; db := 0x30; }
        e := 1; e := 0;
    }
    seq Lcd-write (data)
        input [7:0] data;
    {   par { rs := 1; rw := 0; db := data; }
        e := 1; e := 0;
    }
}
```

Figure 1: Example Device Representation (LCD)

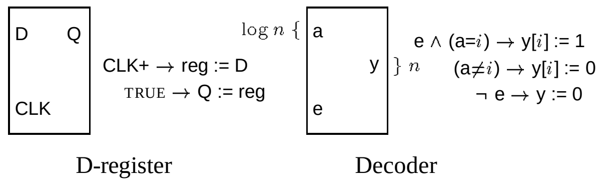

Figure 2: SPR examples for common components

x,” and $x-$ means “falling x.” Y is a list of assignment statements which are activated whenever X is true.

The SPRs express the direction and enabling characteristics of the pins. Direction is captured since data flows from the right to the left of the assignment. The guard expression may represent the clock in a flipflop, the enable of a tristate, or the select lines for a decoder or a multiplexor. An SPR with a guard that is always true represents a wire, a continuous input, or a continuous output. Two SPR examples are shown in Figure 2.

A SEQ represents the skeleton of a device-driver routine. It specifies the series of steps that the controller must take to communicate with the device. A SEQ is essentially a textual representation of the timing diagram, except that SEQs can also include parameters from the driver calls.

The SEQs consist of assignment statements involving the ports and parameters. SEQs contain no control constructs, like loops or conditionals. These constructs are expressed at the level of the behavioral description. Therefore, a SEQ can be viewed as a basic block implementing a

```
seq LCD write (data)
    input [7:0] data;
{   par {
        rs := 1;  rw := 0;
        db := data;
    }
    e := 1; e := 0;
}
```

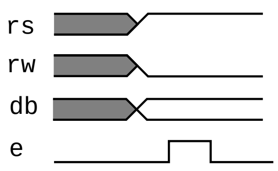

Figure 3: A sample SEQ and its timing diagram.

primitive operation that would normally be written directly in assembly language. A `par` construct lists signals that may be asserted in any order. The behavioral description will eventually be compiled and linked with these low-level basic blocks. Figure 3 shows a SEQ example.

3.2 Microcontroller Representation

The microcontroller attributes are stored in the controller library. These attributes include a controller port list, a set of communication instructions, and a specialized functions list. Figure 4 shows an example specification for the Intel 87C51 microcontroller.

The *controller port list* contains information about each port's name, size, and direction. The direction may be input-only, output-only, or bidirectional. The size of a port is the maximum number of bits that can be written to or read from in parallel during one communication instruction. Ports can have multiple communication instructions representing the different ways the controller can access them.

The *communication instructions* define the semantics of reading and writing the controller ports. Each instruction has a list of typed formal parameters. There are two categories of types: data-types and port-types. A data-type defines the width of a parameter. A port-type is defined by a list of ports and their addressability. Every port instruction contains at least one parameter of some port-type. Each parameter may also be input, output, or bidirectional. The direction of a data-type indicates whether a parameter value is passed into or returned from a SEQ. The direction of a port-type indicates whether a value is written out to or read in from a port.

Many microcontrollers include specific interface features that directly support connections to certain types of devices. Examples of this are built-in serial line controllers (UARTs) that support serial communication using common protocols. These highly specialized interface features are not easily discernible in a behavioral specification or in a low-level description of the controller's interface logic. Therefore, they are treated as special cases.

The *specialized-functions-list* contains devices that either have predesigned interface parts or are replaceable by

```
processor Intel87c51 (p0, p1, p2, p3, ale, psen)
  inout [7:0] p0, p1, p2, p3;
  output ale; input psen;
{ // port type definitions
  byte-port = (p0, p1, p2, p3);
  bit-port = Index-op (byte-port, 0, 7);
  // communication instructions
  instruction write-byte(data, port)
    input [7:0]data;
    output byte-port port;
  { port := data; }
  instruction read-bit(variable, port)
    output variable;
    input bit-port port;
  { variable := port; }
  ....
  // specialized functions
  special uart ()
  { substitute (p3.0, RxD); substitute (p3.1, TxD); }

  special ram (addressWidth, dataWidth)
    input int addressWidth, dataWidth;
  { deviceref addrLatch, RAM;
    addrLatch := instantiate(latch);
    RAM := instantiate(ram,addressWidth,dataWidth);
    connect (RAM.dataBus, p0);
    connect (RAM.hiAddress, p2);
    connect (RAM.loAddress, addrLatch.Q);
    connect (p0, addrLatch.D);
    connect (ale, addrLatch.CLK);
    connect (RAM.we, p3.6);
    connect (RAM.oe, p3.7);
  }
}
```

Figure 4: Example Microcontroller Representation

a built-in function. In the first case, the list contains the interface parts and the netlist for the predefined connections. An example of this is a RAM for the Intel 87C51 which requires a latch for the lower order address byte, so that port p0 may also be used for data. Such a connection is shown in Figure 6. In the second case, the list contains the controller ports which replace an external device. For example, the 87C51 has a built-in UART. The ports p3.0 and p3.1 may be configured as RxD and TxD for serial communication. Such a connection is shown in Figure 7.

3.3 Interface Parts Representation

The interface parts library contains hardware and software templates. The hardware templates include registers, tristates, multiplexors, decoders, and registered-decoders (*i.e.* a decoder whose outputs are latched). They are represented with SPRs similar to the other devices. The software templates are the SEQs required to communicate with these hardware components.

The port-allocation process introduces the interface parts to allow sharing or encoding of the controller ports. Sharing, or time-multiplexing, means a controller port is connected to more than one device port. Encoding, in the context of this paper, means any transformation between n one-hot signals and its $\log n$-bit representation using either a multiplexor, an encoder, or a decoder.

4 Main Algorithm

The port-allocation algorithm described in this section makes a pass over the list of device ports that must be connected to microcontroller ports. It proceeds linearly but may need to backtrack when it discovers an early decision that prevents later optimizations. The backtracking depth is bounded by a constant, the number of controller ports. Therefore, the algorithm is $O(n^2)$ in the worst-case (which rarely occurs) where n is the number of device ports.

The first step of the algorithm initializes the internal data structures. The *netlist*, which contains all point-to-point connections between ports, is initialized to the empty set. The *binding-list*, which contains all connections involving a controller port, is initialized to the empty set. The *free-list*, which keeps track of unconnected controller ports, is initialized to the set of all microcontroller ports. The *device-list*, which contains the attributes of all instantiated devices, is initialized to all of the devices that the user program instantiates. The *dport-list*, which contains all of the device ports requiring a connection, is initialized to all of the device ports of the *device-list* sorted by width.

The second step of the algorithm is *specialized function mapping*. It exploits special built-in functions of the

```
MainAlgo()
{   initialize data structures;
    special functions mapping;
    Recurse(fiveLists);
    if (FAIL) MemoryMappedIO(fiveLists);
}
Recurse(fiveLists)
{   while (dportList $\neq$ EMPTY) {
        $D$ := Dequeue(dportList);
        CList := SelectControllerPort($D$);
        NormalAlloc(CList,$D$,$C$);
        if(SUCCESS) {
          if($C$ is 1st dedicated controller port of its size) {
            Bind(fiveLists);
            DriverLinkAndUpdate(fiveLists);
            $L$ := Recurse(fiveLists);
          // *** this is the backtracking point ***
            if (not FAIL) return $L$;
          }
        }
        if (FAIL) ForcedSharing(fiveLists);
        if (FAIL) EncodingTransform(fiveLists);
        if (FAIL) return FAIL;
        Bind(fiveLists);
        DriverLinkAndUpdate(fiveLists);
    }
    return netlist + drivers + interface parts;
}
```

Figure 5: Main Algorithm

microcontroller or connects devices with predesigned interfaces. It checks if any member of the *dport-list* is also in the *specialized-functions-list*. In order of decreasing width, the algorithm attempts to allocate the specialized controller port to the device port. Upon success, the five lists are updated.

The third step of the algorithm is a recursive call (section 4.1) which attempts to allocate a controller port for every device port from the *dport-list* in order of decreasing width. If this step fails then the algorithm attempts *memory-mapped I/O* (section 4.7). If successful the algorithm returns the *netlist*, *specialized driver routines*, and any necessary interface hardware. This output can be used to synthesize software and hardware. A compiler can generate code with the user program and the SEQs as input. A technology-mapping system can use the SPRs to synthesize the necessary interface hardware components.

4.1 Recurse

The procedure `Recurse` allocates controller ports to device ports using three methods: `NormalAlloc`, `ForcedSharing`, and `EncodingTransform`. It iterates over members D of *dport-list* and calls `Select-ControllerPort` to return a list of suitable controller ports *CList*. `NormalAlloc` (section 4.3) attempts to connect D to a member of *CList* without modifying any SEQs. If successful, it decides whether or not to store the backtracking point.

Backtracking is a method for the algorithm to return to an earlier branching point after making an infeasible allocation decision. It allows the algorithm to explore progressively more expensive but potentially feasible solutions. As a heuristic, this algorithm marks a backtracking point when it allocates the first dedicated controller port of a particular width. The intuition is that it is considered a luxury for a device port to own a dedicated controller port. To mark a backtracking point, the algorithm recurses so that the activation record saves the program state on the stack. If the recursive call is successful then a feasible solution has been found, and all recursive calls return to `Main-Algo`.

If either `NormalAlloc` fails or if `Recurse` backtracks, then `ForcedSharing` (section 4.4) is attempted. If this fails, then `EncodingTransform` (section 4.5) is attempted. If this still fails then `Recurse` backtracks. If there are no more backtrack points, then `Main-Algo` attempts memory-mapped I/O.

If any of the methods is successful, then `Bind` and `DriverLinkAndUpdate` are executed to reflect the allocation decision. When *dport-list* is empty, all device ports have been successfully connected directly or indirectly to the microcontroller.

4.2 Controller Port Selection

`SelectControllerPort` chooses a controller port C for the device port D if it meets the following two conditions. First, C must have a width greater than or equal to that of D. We assume that the width of every device port not on the *specialized-functions-list* is less than or equal to the width of the largest controller port. If a port is too wide, then it must be converted into smaller ports with a higher level transformation, which involves de-multiplexors and changes to the device SEQs. Second, the directionality of C must be compatible with that of D. The list of eligible controller ports $CList$ is sorted by allocated/free ports. This allows `NormalAlloc` to evaluate all potential sharing possibilities before sacrificing any yet unused controller ports.

4.3 Normal Allocation

`NormalAlloc` decides which controller port C should be used to communicate with the given device port D. It iterates over sorted members of $CList$ from the previous step. If a member C of $CList$ passes the *interference check*, then it can be allocated to D without modifying any SEQs.

Interference check determines whether a device port D can be connected to a controller port C without *bus contention* or *sequence clash*. *Bus contention* occurs when two or more device ports connected to the same controller port are *active* at the same time. A device port is *active* if it is inputting or outputting a value. A *sequence clash* occurs when two ports of the same device are both *active* in the same SEQ, and they are both connected, directly or indirectly, to the same controller port.

Bus contention is detected by examining the SPR guards of each device port connected to the same controller port. A guard defines the conditions under which the device port is *active*. The steps of bus contention check are *weakest guard extraction*, *post-sequence condition evaluation*, and *contention evaluation*.

A *weakest guard* is defined as the minimum cover of a disjunctive (sum of products) expression. For example, suppose S_1 equals $abc + ae + af$. Then the weakest guard is a. As another example, let S_2 equal $abc + bd + e$. Then the weakest guard is $b + e$. The weakest guard can be thought of as a master enable for the remaining signals in the set of SPRs. Using the weakest guard provides more sharing opportunities for controller ports.

The *post-sequence condition* (PSC) is defined as the set of values that a device port retains after the execution of any SEQ that references it. Note that we are only interested in the PSCs of those device ports that are weakest guards. The PSC of a guard G is the union of the last assignment to G in all SEQs.

Contention evaluation determines whether any two device ports from different devices connected to the same controller port are mutually exclusive. That is, only one may be *active* at a time. To check if a port D is active outside of a SEQ call, the PSC set is substituted into the *weakest guard* set. If any of the guard expressions evaluates to true, then D remains active after that particular SEQ. Therefore, connecting D to C will result in contention.

The *sequence clash check* determines whether two device ports from the same device connected to the same controller port are mutually exclusive. In addition to the point-to-point connections, the *sequence clash check* must also examine those device ports indirectly connected to the same controller port. This knowledge can be inferred from the *netlist*. If two such ports appear in the same SEQ, then they have a sequence clash.

4.4 Forced Sharing

`ForcedSharing` eliminates bus contention by strengthening the guard for the device port to be shared. Guard strengthening is accomplished by introducing interface hardware in some combination of tristates and D-registers.

A controller port is a candidate for *forced sharing* if it has an existing connection and a width greater than one. Among the candidates, a shared port is preferred over a dedicated port for forced sharing. The justification is that a shared port is already guarded and requires at most one interface component. On the other hand, a dedicated port may not be guarded and may require two interface parts.

Based upon the directionality of D, `ForcedSharing` generates suitable interface hardware. For an input-only device port, a D-register is chosen. For an output only device port, a tristate is chosen. For a bidirectional device port, a bidirectional latch is chosen. Once the interface component is selected, it is added to the *device-list*. Both the controller port and the device port are connected to this component. In the *binding* step described below, everywhere a SEQ references this device port is replaced with an interface routine call.

4.5 Encoding Transformations

The algorithm uses three *encoding transformation* techniques to encode single-bit device ports. They are decoder, registered-decoder, and multiplexor. These techniques encode n single-bit device ports in $\log n$ bits of a controller port. The example in section 5.2 applies both decoder and registered-decoder transformations.

The decoder transformation technique encodes a set of single-bit, one-hot, input-only ports. The encoded controller ports are connected to the input of an enabled decoder. Each device port is connected to an output of the decoder. In addition, a separate enable port is required to prevent glitching. This enable port is added to the *dport-list* for later allocation.

To determine if a set of device ports is one-hot, the algorithm evaluates the PSCs for each port. If the port has a constant PSC, then it can be one-hot encoded. If the candidate ports have different constant values, then polarity conversion will be necessary.

The registered-decoder transformation technique encodes a set of n single-bit, input-only device ports which are not one-hot. Note that registered-decoders are more expensive in both software and hardware than plain decoders. The $\log n$ encoded controller ports are input to an enabled decoder. The outputs of the decoder are connected to the latch enable inputs of n D-registers. Each device port is connected to a register output. A single data bit from the controller is fanned out to each register's input. To prevent glitching of the register enables, a decoder enable port is required. The data port and the decoder enable port are added to the *dport-list* for later allocation.

The multiplexor transformation technique encodes a set of n single-bit output-only ports. Unlike the input-only case, the algorithm cannot determine if the outputs of device ports are one-hot. Each of the n device ports are connected to the multiplexor's input. The $\log n$ encoded controller ports are connected to select one of these n device ports.

4.6 Bind, Link, and Update

Pairs of (controller port C, device port D) are passed to `Bind` upon successful allocation. The binding step updates the five lists to reflect the connection decisions.

After the *binding* step, the same (C, D) pairs are passed to `DriverLinkAndUpdate`. The *linking* step is similar to linking in program compilation, where all references are bound to addresses. In the driver's case, device port references in the SEQs are bound to the controller port references in the SEQs and the SPRs. If interface hardware H is introduced for D, then the *update* step will replace all assignments involving D with the SEQ calls to H.

4.7 Memory-Mapped I/O

Memory-mapped I/O is attempted when all other transformations have failed. It is expensive in terms of hardware and performance, because it involves complex, controller specific memory sequencing and a consistent allocation of the memory space. We are currently developing a more general algorithm.

A memory transaction includes an address cycle followed by a data cycle. A read or write signal is asserted to signify the data cycle. We assume that the address is held throughout the data cycle. The address bits are decoded to control the interface hardware introduced for memory-mapping.

Transformations for input-only and output-only device ports are analogous to the encoding transformations described above. In the bidirectional case, the device port must be isolated from the controller's data bus with an I/O latch such as that used in Figure 7.

5 Examples

This section presents practical examples to illustrate the utility of the algorithm. Results indicate that the synthesized systems are comparable to implementations designed by humans. The first example is an electronic phonebook. The second example is a software/hardware interface for an interactive tester.

5.1 Electronic Phonebook

This example is a phonebook with a dialer. Names and phone numbers are downloaded into RAM via the serial

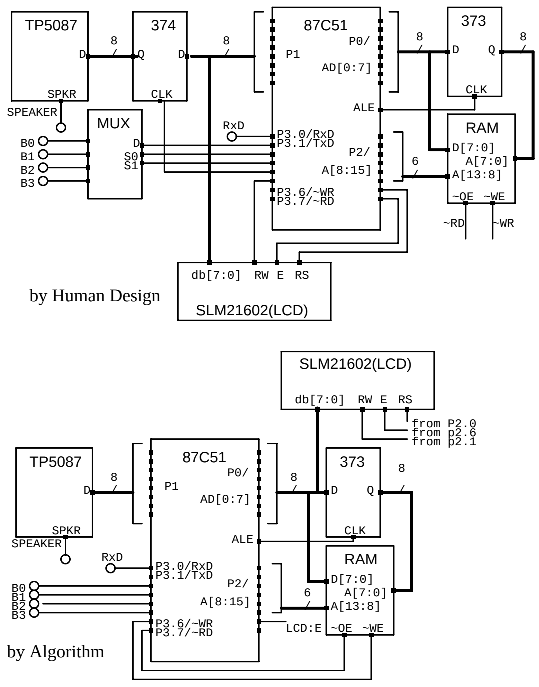

Figure 6: Electronic Phonebook

port. Required components include a TP5087 dial-tone generator, four switches, an SLM 21602 LCD, a 16K RAM and a UART all controlled by an Intel 87C51 microcontroller.

The 87C51 is a microcontroller with 32 single-bit I/O ports. They can be addressed individually, or in groups of eight. The groups are p0, p1, p2, and p3. When addressed individually, they are referenced by their group name followed by an index. For example, p0.0 or p3.7.

The LCD has four ports: an 8-bit bidirectional data port named DB, and three input-only ports RW, RS, and E. RW selects the direction of DB for reading or writing. RS selects between the instruction register and the data register in the LCD. E signals the start of a read or a write operation. The TP5087 has an 8-bit input only data port. Each of the four switches has an output only port. The RAM has a 14-bit address port, an 8-bit data port, a read enable port, and a write enable port. The UART has a data-in port and a data-out port.

The *specialized-functions-list* of the Intel 87C51 contains a RAM and a UART. The RAM has a predesigned interface consisting of a level-sensitive latch to isolate the lower address bits from the data bits. Unlike the RAM which is an external device, the UART is a built-in function of the 87C51. Therefore, no physical UART device is instantiated. The ports p3.0 and p3.1 correspond to the RxD and TxD ports of the UART.

The dialtone generator is not sharable so it is allocated its own controller port. The DB port of the LCD is sharable, therefore it is assigned the same controller port as the RAM. The E port of the LCD is not sharable. RS and RW are sharable since they are both guarded by E. The four switches are not sharable, so they are allocated their own controller ports.

The solution generated by the algorithm has comparable performance to the human design while using two fewer hardware components. The algorithm detected a subtle sharing opportunity between the RAM's data port and the LCD's data port. This sharing decision enables the switches to be connected directly to the controller. The human design forces the tone generator and the LCD to share p1 using a register. The register's CLK requires a dedicated controller port leaving only three free ports. A multiplexor is required to connect the four switches.

5.2 Hardware/Software Interface for Interactive Tester

This example allows a logic simulator or tester program running on a personal computer to read and write values to 256 physical pins connected to an actual circuit. The computer sends commands to the microcontroller via the serial port. The controller then samples or sets the pins accordingly. The pins are organized in 32 groups of eight. The controller communicates with 32 8-bit bidirectional latches connected to these pins. The required components include an 87C51, 32 8-bit bidirectional latches, and a UART.

Each bidirectional latch has two 8-bit bidirectional data ports and three 1-bit control ports. The data port D is read or written by the controller. The data port B connects to the pins of the device under test. The three control ports are write, enable, and read. The write signal causes D to be latched internally. The enable signal causes the internal latch to output its value on B. When enable is low, B is tristated. The read signal causes D to be driven with the value of B.

All of the D ports are sharable, so they are all connected to p0. The remaining 96 read, write, and enable ports are not sharable. The read and write signals have constant PSCs and no sequence clashes. Therefore the algorithm uses a 6:64 enabled decoder to address them. Since enable has a variable PSC, the algorithm uses an enabled 5:32 registered-decoder.

The solution generated by the algorithm after technology mapping with similar components uses ten fewer hardware packages than the human design. The extra components in the human solution support a sequential access pattern of the pins which minimizes controller ports required. It has slightly higher performance because fewer instructions are needed per access. A higher level transformation would be needed for an interface synthesis algo-

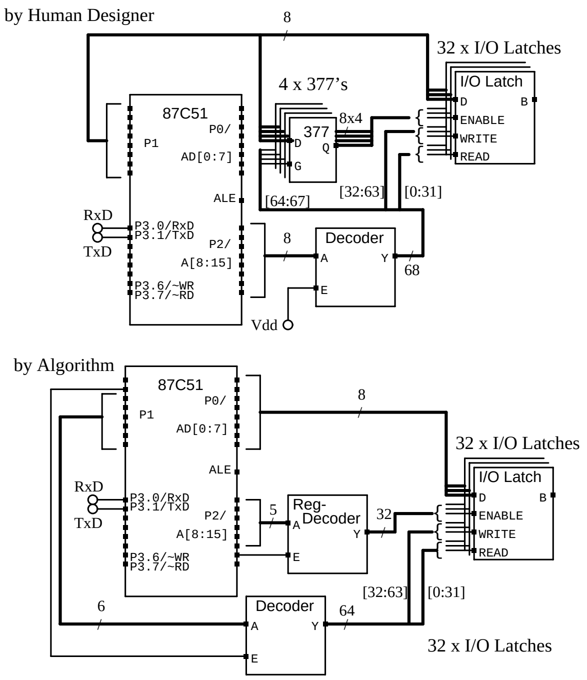

Figure 7: Hardware/Software Interface for Interactive Tester

rithm to exploit such access patterns.

6 Conclusion and Future Work

We have presented an algorithm that interfaces a micro-controller with its peripheral devices. It synthesizes necessary interface hardware, and outputs specialized driver software. The solutions generated are comparable to systems designed by hand as demonstrated by the examples.

Several related problems will be addressed in future work. We are still developing transformation techniques for memory-mapped I/O. While the current algorithm is effective for a wide class of problems, the real-time issues (e.g., timing-constrained sequencing) must still be addressed. Higher-level transformations are required to meet performance and real-time constraints as well as optimizing the use of hardware and software resources. Specifically, the communication patterns suggest possible performance optimizations. This algorithm will be used as a subroutine by a hardware/software co-synthesis system to evaluate the various tradeoffs and the many possible transformations.

References

[1] Intel. *8-Bit Embedded Controller Handbook*, Intel Corporation, 1990.

[2] A.J. Martin. *Programming in VLSI: From Communicating Processes To Delay-Insensitive Circuits*, Department of Computer Science, California Institute of Technology, Caltech-CS-TR-89-1, 1989.

[3] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice/Hall International, Englewood Cliffs, NJ, 1985.

[4] M. Srivastava, R. Brodersen. “Rapid Prototyping of Hardware and Software in a Unified Framework”, *Proc. of the International Conference on Computer-Aided Design*, 1991.

[5] J. Sun, R.W. Brodersen, “Design of System Interface Modules”, Submitted to ICCAD-92.

[6] F. Vahid, D. Gajski, “Specification Partitioning for System Design”, *29th ACM/IEEE Design Automation Conference*, June 1992.

[7] Shelley & Associates. *SLM21602 LCD Data Book*, 1990.