Communication Software Code Generation

Andreas Gerstlauer

Technical Report ICS-00-46
August 1, 2000

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

gerstl@cecs.uci.edu
http://www.cecs.uci.edu/~gerstl

Abstract

This report describe the implementation of system-level communication on a programmable processor. First, the issues are introduced using the example of communication software on a Motorola DSP. Then, the problem is generalized and defined for the general case of system-level communication on a programmable processor.

Contents

List of Figures

Communication Software Code Generation

A. Gerstlauer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describe the implementation of system-level communication on a programmable processor. First, the issues are introduced using the example of communication software on a Motorola DSP. Then, the problem is generalized and defined for the general case of system-level communication on a programmable processor.

1 Introduction

This document describes the processor/software aspects of communicating data items between a part of an application running as software on a processor and a part of the application running in another component, e.g. a custom hardware block. After partitioning, data items are transferred between hardware and software. The following sections cover the issues of implementing this communication on the processor side. For demonstrational purposes, the examples show the implementation on a Motorola DSP56600 processor core [1].

In the partitioned specification, a data transfer is initiated through communication primitives (e.g. `send()` and `receive()`). The specification contains primitives with different semantics (blocked vs. non-blocked) and different data types. In the following we will describe the implementation of blocked, synchronous message passing primitives for arbitrary data types. Other semantics are special versions of this general case, e.g. non-blocking sends/receives or non-typed transfers (simple event notification/wait without data transfer).

In the implementation on the processor side, the communication primitives are realized through two layers of protocols:

1. High-level handshaking to ensure the synchronous, blocking semantics of the communication primitives and to implement the transfer of arbitrary data items over the low-level protocol.

2. Low-level handshaking for performing a single transfer of a basic data item (as supported by the instruction set) over the actual address, data and control lines according to the processor bus protocol.

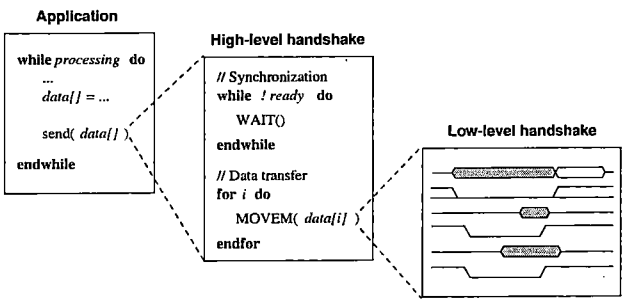

Figure 1: Communication and handshaking levels.

The two protocol layers hierarchically build on each other in the sense that the high-level layer uses the services provided by the interface of the low-level layer (Figure 1). The layers are clearly separated and independent from each other.

2 Low-Level Handshaking

Assuming that both communication ends have been synchronized to agree on transferring a data item, the actual data value has to be communicated over a bus consisting of control, data and address lines. The bus protocol handshaking guarantees that the data is safely transferred to the receiving end.

Data is transferred to other components over one of the processor busses. On the software side, data transfers over the processor busses are supported by the I/O instructions of the instruction set. Depending on the bus the component is connected to, a transfer is performed through a normal `MOVE`-type of instruction (memory-mapped I/O over the processor data bus) or a special I/O instruction (port I/O over a peripheral bus). In both cases, the semantics of the instructions are to transfer a data item between the processor memory or one of the processor registers to external memory or an external register.

The processor busses in general are assumed to support the transfer of a single data item (e.g. a machine word) as supported by the instruction set. In case of a long word transfer (two machine words) a transfer might consist of multiple bus cycles or a bus burst cycle. All transfers over the processor busses are initiated by executing corresponding I/O instructions on the software side. Hence, the processor is considered the master on the bus and all other components are slaves listening to requests at the time of a transfer (through high-level handshaking, see Section 3).

The bus protocols are typically standard memory or simple register file access protocols. In general, bus protocols are fixed and given for each processor. The hardware has to be synthesized to conform with this bus protocol. However, the bus protocol timing might be parameterizable through certain processor control registers during processor initialization (reset).

In general, processor bus protocols support a scheme to address different locations in the external memory or port space. Addressing is needed in case of multiple components connected to the same bus or in case of multiple communication points inside the same component (e.g. multiple tasks communicating in parallel). Each of the communication endpoints is assigned an address or a range of addresses on the bus. By decoding bus addresses the components determine the communication target.

In case of the DSP56600, the external bus protocol of the processor is a memory-access protocol and data transfers over the external bus are handled via memory-mapped I/O on the instruction side. The corresponding timing diagram of the external bus is shown in Figure 2. On the software side, the external address range is mapped into the program memory space and the instruction set supports transfers of single machine words (16-bit) via the `MOVEM` instruction.

The Motorola processor allows the external data transfer protocol to be parameterized by the number of wait states (*WS*) added during each transfer. The *WS* parameter is set by programming a certain register in the processor as part of processor initialization during reset. It influences a number of protocol timing constraints for both read and write accesses.

2.1 Timing Constraints

The timing diagram including the timing constraints of the DSP56600 processor for an external write access is shown in Figure 3 and for an external read access in Figure 4.

The timing characteristics for the read and write accesses depend on the processor clock period T_C and on the number of programmed wait states *WS*:

Access time T_{100} The time the address is valid on the bus

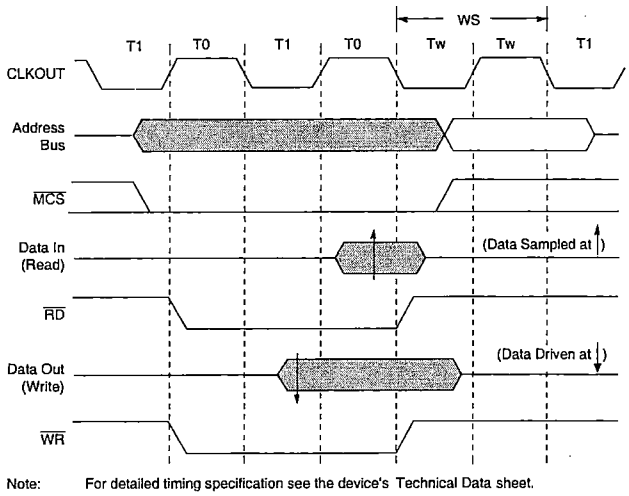

Figure 2: Timing diagram for single `MOVEM` data transfer.

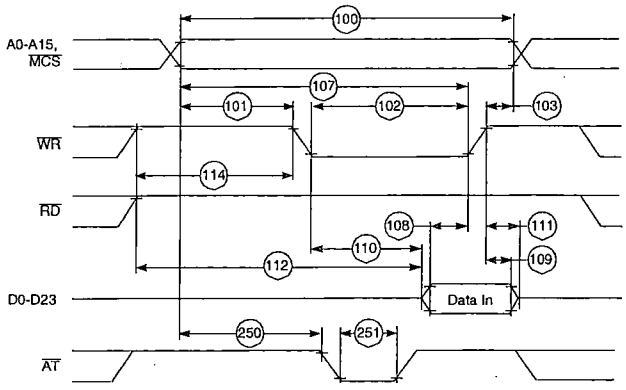

Figure 3: Timing diagram for external write.

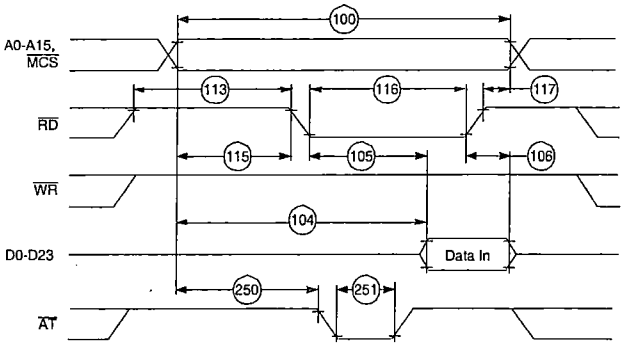

Figure 4: Timing diagram for external read.

(time between address and data becoming valid):

$$\begin{array}{ll} 1 \leq WS \leq 3: & T_{100} > (WS+1) \times T_C - 4.4 \\ 4 \leq WS \leq 7: & T_{100} > (WS+2) \times T_C - 4.4 \\ WS \geq 8: & T_{100} > (WS+3) \times T_C - 4.4 \end{array}$$

Write assertion time T_{101} Time between address valid and write assertion:

$$\begin{array}{ll} WS = 1: & T_{101} > 0.25 \times T_C - 3.7 \\ 2 \leq WS \leq 3: & T_{101} > 0.75 \times T_C - 4.4 \\ WS \geq 4: & T_{101} > 1.25 \times T_C - 4.4 \end{array}$$

Write assertion width T_{102} Time between write assertion and write deassertion:

$$\begin{array}{ll} WS = 1: & T_{102} > 1.5 \times T_C - 5.7 \\ 2 \leq WS \leq 3: & T_{102} > WS \times T_C - 4.4 \\ WS \geq 4: & T_{102} > (WS - 0.5) \times T_C - 4.4 \end{array}$$

Data hold time T_{109} Time data is valid after write deassertion:

$$\begin{array}{ll} 1 \leq WS \leq 3: & T_{109} > 0.25 \times T_C - 3.8 \\ 4 \leq WS \leq 7: & T_{109} > 1.25 \times T_C - 3.8 \\ WS \geq 8: & T_{109} > 2.25 \times T_C - 3.8 \end{array}$$

Read assertion time T_{115} Time between address valid and read assertion:

$$T_{115} > 0.5 \times T_C - 4.0$$

Read assertion width T_{116} Time between read assertion and read deassertion:

$$T_{116} > (WS + 0.25) \times T_C - 3.8$$

Output enable time T_{105} Time between read assertion and input data valid:

$$T_{105} < (WS + 0.5) \times T_C - 8.5$$

Therefore, given a certain number of wait states the constraints for synthesizing the hardware talking to the processor can be directly derived. On the other hand, given the timing of the synthesized hardware, the number of wait states needed on the processor side is determined such that all timing constraints as stated above are satisfied.

Given the timing constraints $T_{i_{min}}$ of the hardware and the piece-wise linear equations $WS_i(T_i)$ of the processor's minimal number of wait states given a timing constraint, the minimal number of wait states WS_{min} needed is calculated:

$WS_{min} = 1;$
for all T_i **do**
  if $WS_i(T_{i_{min}}) < WS_{min}$ **then**
    $WS_{min} = WS_i(T_{i_{min}});$
  endif
endfor

Depending on the access type the following input constraints $T_{i_{min}}$ and corresponding equations $WS_i(T_i)$ are needed:

- Read Access
  - Access time T_{100}
  - Read assertion time T_{115}
  - Read assertion width T_{116}
  - Output enable time T_{105}
- Write Access
  - Access time T_{100}
  - Write assertion time T_{101}
  - Write assertion width T_{102}
  - Data hold time T_{109}

Note that the WS calculation (through the access time parameter, T_{100}) includes the bus cycle time supported by the hardware, i.e. the time needed by the hardware until it can accept the next word of a multi-cycle transfer in case of communicating a multi-word data item (see Section 4.1.2). Hence, no further synchronization at this level is necessary.

3 High-Level Synchronization

Based on transferring a single data item using the low-level bus handshaking protocol, at the next level it has to be ensured that the data processing rates at both communication ends match when processing and transferring multiple data items in a loop-like fashion out of the application. The communication partners have to be synchronized such that the slave is ready when the master is about to initiate a transfer.

Single data transfers over the bus are initiated by the software on the processor, i.e. the processor is the master on the bus and the hardware reacts as a slave (see Section 2). This implies that read and write data transfers are only performed at the request of the software on the processor. Hence, the software has to be synchronized in accordance with the hardware processing rate to initiate transfers only when it is guaranteed that the hardware is ready to supply or accept the next data time in case of a bus read or bus write, respectively.

For every transfer there has to be a task on side waiting to send an item and a task on the other end waiting to receive it. The purpose of high-level handshaking is to make sure that the two tasks meet to perform the actual transfer. In both cases (send or receive) and on each end, the flow is to

1. Signal ready state and wait until the other end becomes ready.
2. Perform read or write data transfer using low-level protocols.

On the processor, while waiting for the other communication end, other tasks might execute in a multi-tasking fashion as determined by the operating system environment chosen for the software part.

Due to the master/slave nature of the bus, the processor is also in control of the handshaking process. A low-level transfer is initiated through the processor and, in addition to the actual transfer, constitutes an event sent to the hardware signaling the processor being ready. Upon reaching a send or receive communication point, the hardware just sits and listens for a corresponding read or write transfer on the bus to/from the assigned address. Hence, high-level synchronization only has to ensure that the hardware is ready before the processor initiates a transfer.

In the following sections we will describe the different possibilities for implementing high-level synchronization.

3.0.1 Delay Software

If the timing relation between hardware and software is known (in the worst case), the software side can be artificially delayed such that the hardware is guaranteed to be ready by the time the software initiates a transfer. Software timing is adjusted, for example, by inserting appropriate amount of NOP instructions, sleeping the processor for the required amount of time or executing other tasks for a certain time. In contrast to the other schemes, this requires that the timing of both the software and the hardware is known.

Given the best-case and worst-case start times of the corresponding communication operations on the software and hardware side, respectively (as a result of scheduling), the worst-case delay between software and hardware is determined. If the delay is negative no modifications are necessary. If the delay is positive, the number of cycles (and hence the number of NOP operations, for example) required is computed by dividing the delay by the processor clock period and subtracting the delay loop overhead:

```
  do #nn,L1          ; delay loop
  NOP
L1
  MOVEM  a,p:(Rx)    ; data transfer
```

The inputs for an implementation of delay handshaking are:

- Communication operation start times (best case/worst case).
- Processor clock period.
- Delay loop overhead cycles.

3.0.2 Polling

Software on the processor is synchronized with the hardware by polling. The software reads a register/flag in the hardware at regular intervals until the hardware signals its ready state:

```
L1
  MOVEM  $xxxx,b     ; poll reg./flag
  TST    b           ; test for zero
  JNE    L1          ; keep repeating

  MOVEM  a,p:(Rx)    ; data transfer
```

Polling the register/flag is handled as a normal data transfer over the external bus with a specially assigned address.

Note that in addition to the actual data processing FSMD, this requires an additional FSMD in the hardware which runs concurrently to the computation FSMD and answers polling requests from the processor while the main computation is still running. The FSMD implements polling register reads from the processor by listening on the bus, waiting for a bus read cycle with the assigned address and supplying the polling register value on such requests.

Parameters related to the polling handshaking scheme are:

- Address for the polling register/flag read cycle.

3.0.3 Interrupt-Based

The hardware signals ready state to the software through an interrupt. One of the processor's interrupt inputs is reserved for hardware handshaking. The corresponding interrupt handler sets a global variable in the processor memory whenever the hardware interrupt is received:

```
  global FIntHW_Handler
FIntHW_Handler
  MOVE   a,y:(r6)+   ; save reg
  MOVE   #1,a
  MOVE   a,y:FREADY  ; set READY flag
  MOVE   y:-(r6),a   ; restore reg
```

```
  RTI

  ; set interrupt vector to point
  ; to interrupt handler
  org p:$INTx_Vector
  JSR    >FIntHW_Handler
```

The data processing task on the processor then checks the flag before each data transfer. If the hardware is not ready yet, the processor is suspended until the corresponding interrupt has been received. Then, the interrupt flag is reset and the data transfer is performed:

```
L1
  MOVE  y:FREADY,b ; read READY flag
  TST   b          ; test for zero
  JEQ   L2         ; break if flag
  WAIT             ; wait for int
  JMP   L1         ; test flag again
L2
  CLR   b
  MOVE  b,y:FREADY ; clear flag

  MOVEM a,p:(Rx)   ; data transfer
```

Since single data transfers are in itself synchronized with the hardware and since the hardware doesn't issue the next interrupt before the actual data transfer has been performed, no additional handshaking is necessary and it is guaranteed that no interrupt is lost.

Parameters related to the polling handshaking scheme are:

- Interrupt vector of the interrupt the hardware is connected to.
- Interrupt parameters like edge-sensitive vs. level-sensitive, etc.

4 Problem Statement

The problem to solve is, given a processor out of the IP database and a component connected to the processor bus, to create code for implementation of the software part of the given specification for the processor—in this case the DSP56600.

The communication code generation tool (see Figure 5) in the end will generate assembly code for the communication layers to be linked to the rest of the application running on the processor. Code is generated by the tool using assembly code templates stored in the IP database together with the processor data. The inputs to the code generation tool fall into the following categories:

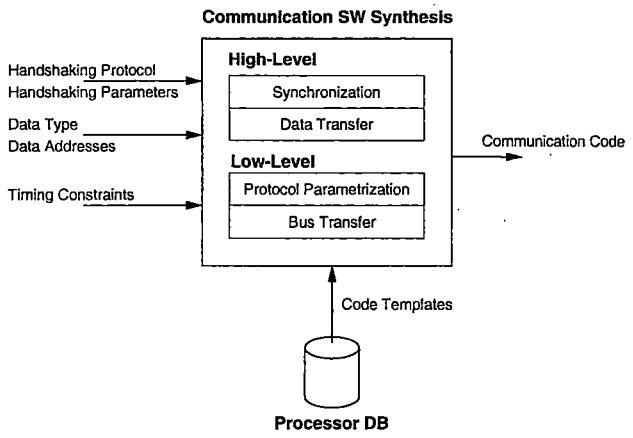

Figure 5: Communication Code Generation.

- the selected high-level handshaking protocol and its parameters (Section 3),
- type and location of the data item to be transferred in the processor memory.
- address of the communication partner on the processor bus, and
- the external bus protocol supported by the hardware and the associated timing constraints.

The problem consists of several subproblems at the two different layers of the communication hierarchy. The two subtasks of the code generation process are:

1. create low-level I/O code
   - code for transferring a single data item over the processor bus as supported by the bus and instruction set
2. create high-level handshaking functions
   - synchronization code
   - code to transfer arbitrary data types using low-level protocol functions

The two layers of code will sit on top of each other in the generated code, i.e. the high-level functions are implemented during code generation utilizing the low-level routines generated previously by the first code generation subtask.

Code generation produces an implementation of the `send()` and `receive()` communication primitives on the software side by creating corresponding `send()` and `receive()` functions which will be called from the application. In addition, code generation can add to the runtime library that will be linked to the final application during compilation. The runtime library source code provides

designated space for inserting interrupt handlers or processor initialization/reset code, for example. All in all, the generated code becomes part of the operating system kernel the application will be linked against.

In the following sections the inputs and approaches for each subproblem will be defined. In all cases, communication code which became part of the operating system layer on the processor is created.

4.1 High-Level Handshaking

Create code for implementation of the communication primitives in the application. For each `send()` and `receive()` communication primitive of different data type a function is created. The communication primitives in the specification will then be implemented as calls to the created functions in the code generated for the processor.

In general, the communication functions consist of code for synchronization between software and hardware followed by code for transferring of the given data item over the processor bus using the low-level protocol. The communication functions are assembled from those two parts. Hence, high-level handshaking code generation is subdivided into generation of synchronization code and generation of data transfer code subtasks described in the following sections.

4.1.1 Synchronization

Create code for implementation of the communication primitives synchronization and blocking semantics. Input to code creation is the type of the desired handshaking implementation (Section 3). Based on that the corresponding code template stored in the database is used. The templates are customized using a set of parameters depending on the type of handshaking:

- No handshaking: no parameters.
- Software delay handshaking:
  - Number of delay cycles
    The number of cycles is calculated from the difference between software and hardware start times. The number of cycles equals the delay between software and hardware divided by the processor clock period (minus delay loop overhead, see Section 3.0.1).
- Polling:
  - Address for external polling register/flag
- Interrupt-based handshaking:
  - Interrupt vector of the interrupt the hardware is connected to

The template code is inserted at the beginning of the high-level handshaking function body. The parameters in the code templates are replaced with the actual values to create the final code for the synchronization part of the communication function.

Finally, in case of an interrupt-based handshaking, code for the interrupt handler and initialization code for setting up interrupt vectors is generated in a similar manner from corresponding templates stored in the processor database. This code is inserted into the interrupt handler table and processor reset routine of the runtime library, respectively.

4.1.2 Data Transfer

Create code for transferring the given data item using the low-level routines. The interface provided by the low-level protocol supports transferring a single data item over the processor bus (see Section 4.2.2).

In order transfer an arbitrary data item as specified by the application, code is produced to split the data transfer into multiple data transfers as supported by the low-level protocol. Input parameters are:

- Address of data item in processor memory
- Type information of data item (symbol table entry)
- Address of data item on the external processor bus
- List of data transfers supported by the low-level protocol
  - data type
  - instruction/function name

The generated data transfer code becomes part of the communication functions following synchronization.

Code to transfer complex data types consisting of multiple machine words in processor memory depends on the data serialization convention, i.e. on the order in which the different words of the complex data item are transmitted:

Processor Memory Layout Data words are transmitted in the order in which they are stored in processor memory. Code is generated that loops over all consecutive addresses occupied by the data item (based on the address and size of the data item). Data is transmitted in the largest chunks supported by the low-level protocol.

Canonical Serialization Data is transmitted in a canonical, predetermined order. The processor database contains a library of serialization functions for each

basic and complex C data type. Serialization functions for basic types are implemented using optimal low-level routines according to the data type. Serialization functions for complex data types (arrays and structs) recursively call the serialization function for the corresponding base type. Code is generated by calling the appropriate serialization function depending on the type of the data item.

Arbitrary Serialization The order of data serialization is arbitrary as defined by external factors (e.g. the hardware). Inputs to code creation are:

- For each basic data type (integral and floating-point types) a definition of the number, type and order of low-level transfers (e.g. big-endian vs. little-endian order). For each basic data type a sequence of low-level transfer has to be defined.
- For each complex data type (arrays and structs) the order in which the elements of the base type are to be transmitted (e.g. row-major or column-major in case of arrays).

Code for custom serialization functions is then created. Each function serializes transmission of a certain data type according to the definition. The functions are then called in the same manner as in the canonical case.

4.2 Low-Level Handshaking

4.2.1 Protocol Parameterization

Create code for initializing the registers associated with bus protocol parameters of the processor. Initialization code to be inserted into the processor reset routine of the runtime library is created from a template stored in the IP database for the processor. The code template modifies the processor control registers according to the selected parameters.

For each processor bus a hardware module is connected to, the following information is needed:

- list of protocol parameters
- list of bus timing constraints
- list of equations for calculating a parameter as a function of each timing constraint
- actual delay values of the hardware for each timing constraint

The different protocol parameters are then computed by evaluating the parameter functions using the given delay values and selecting the optimal value over all constraints (see the DSP56600 example in Section 2).

4.2.2 Bus Transfer

Create code for transferring a single data item over the processor bus. Depending on the complexity the code for transferring a single data item is either inlined into the high-level communication functions (see Section 4.1.2) or becomes a separate function which is called from inside the high-level functions.

The set of I/O instructions needed to implement a data transfer is taken from the processor database in the form of a code template. Usually, however, processor bus transfers are supported by a single processor I/O instruction that will be inlined into the communication functions. In case of the DSP56600, for example, a data transfer is implemented using a single `MOVEM` instruction.

5 Conclusions

In this report we described and defined the problem of generating code for implementation of the specification's communication functionality for parts of an application running on a processor. The problem and approach was described on the example of the Motorola DSP56600 processor. However, the general problem statement applies equally to other processors.

The problem can be extended and generalized in a couple of ways: The approach presented here assumes that the communication is handled over the processor bus as supported directly by the processor architecture and the instruction set. However, the processor core might include (possibly optional) I/O peripherals that implement extended I/O functionality and more complex external protocols. Depending on the protocol selected for communication between the processor and other components, a processor core might be allocated that supports the required protocol natively in such a way.

In this case the processor database would contain the necessary driver code to implement low-level data transfers over using the I/O module. However, since the driver or the hardware in the I/O module can support extended functionality like blocking, synchronization or buffering the semantics of the low-level interface for code generation have to become more general. The high-level code generation then has to take this additional parameter into account when implementing the semantics as required by the specification.

On the other hand, the I/O peripheral is itself connected to one of the processor busses. Hence, the peripheral driver communicates with the peripheral hardware over the processor bus, using the processor's I/O instructions to read/write the peripheral registers. Therefore, the peripheral module can be modeled as an additional hardware

component interfacing between the processor and the other components. At the specification level corresponding code modeling the peripheral behavior and its communication with the processor is inserted. Since the added code will in turn be based on standard communication primitives it sits one level above communication code generation as described in this report.

Finally, the processor might include a peripheral module for general I/O that allows the software direct control over some of the processor core's input and output pins. This allows implementation of any arbitrary protocol (within the restrictions of the processor state machine) on those processor pins. In this case, generation of the low-level bus transfer code becomes general in the sense that the software side is synthesized to conform with a given hardware protocol. Given a description of the hardware protocol in a suitable format (e.g. Protocol Sequence Graph, PSG), the problem is to generate low-level handshaking code for implementation of that protocol over the processor pins.

References

[1] Motorola, Inc., Semiconductor Products Sector, DSP Division, *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.