Embedded Architecture Co-Synthesis and System Integration

Bill Lin  Steven Vercauteren  Hugo De Man
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract — Embedded system architectures comprising of software programmable components (e.g., DSP, ASIP, and micro-controller cores) and customized hardware co-processors, integrated into a single cost-efficient VLSI chip, are emerging as a key solution to today's microelectronics design problems. This trend is being driven by new emerging applications in the areas of wireless communication, high-speed optical networking, and multimedia computing. A key problem confronted by embedded system designers today is the rapid prototyping of application-specific embedded system architectures where different combinations of programmable processors and hardware components must be integrated together, while ensuring that the hardware and software parts communicate correctly. In this paper, we present a solution to this embedded architecture co-synthesis and system integration problem based on an orchestrated combination of architectural strategies, parameterized libraries, and software CAD tools.

1 Introduction

Telecommunication and multimedia computing are among the fastest growing segments of the microelectronics market today. These market sectors are being fueled by new emerging business and consumer applications that are now possible with recent advances in wireless communication, videoprocessing, and integrated networking technologies. The design of VLSI chips in these applications are often subject to stringent requirements in terms of processing performance and power dissipation.

To facilitate flexible low-cost designs in short design time, emerging designs are based on heterogeneous embedded system architectures, as depicted in Figure 1, that integrate software programmable components, e.g., DSP and microprocessor cores, together with customized and "pre-designed" hardware processing components on a custom IC. Programmability is introduced in these system-on-silicon architectures, while maintaining most of the advantages of customized VLSI architectures, such as the potential to optimize the processing performance and power dissipation.

To enable the rapid design of embedded systems, significant advances are required on a number of key system design problems. These problems include system modeling, co-simulation, performance estimation, partitioning, re-targetable code generation, and embedded architecture co-synthesis and system integration. Each of these problems is being investigated by several ongoing efforts world-wide [3, 5, 6, 7, 2, 10, 1, 4, 9, 15].

In this paper, we aim to address the latter problem of supporting the designer in constructing an application-specific embedded system architecture from a high-level and in integrating the different system components together. We refer to this problem as embedded architecture co-synthesis and system integration. Depending on the application requirements, different application-specific

![Diagram](attachment:Figure_1.png)

Figure 1: Heterogeneous embedded system architectures.
channel. The sender must block until the receiver is ready to receive, and vice versa. This rendezvous semantics ensures that both parties are synchronized with each other before the data transfer takes place. We have chosen the CSP communication model because it has a rigorously defined semantics along with a well defined algebra to reason about the communication behavior, supported by existing formal verification methods. To support other communication models, such as buffered communication, we have chosen to mimic these models by using intermediate component units that implement that communication behavior.

A processor component unit can either be a hardware component or a software programmable component (e.g. DSP, ASIP, or micro-controller core). In the case of hardware, the processor component unit can either be a "pre-designed" library component, including parameterized communication components like buffers, or a hardware processor that has still to be implemented. The hardware component may consist of internal storage. In the case of a software programmable component, the processor component unit consists of the processor core itself, an internal memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the communication interface to its external environment, as depicted in Figure 2(b). The I/O unit acts as a "hardware wrapper" that effectively encapsulates a software programmable component into a hardware component. The I/O unit is driven by the processor core via the software program that executes on it.

3 Implementation of Channels

To ensure different processor components can be integrated together at the "implementation" level, they must communicate with each other in a well-known and consistent manner at the "circuit" level. Our strategy to this problem is to define a common circuit level protocol that will be used by all components to implement the control mechanism for synchronizing the channel transfers.

In choosing this protocol scheme, several important factors must be considered. We start from the simple observation that most of today's digital hardware operates synchronously. This is not surprising since synchronous circuits are far easier to design and implement when compared to their asynchronous counterparts.

In synchronous edge-triggered designs, the clock signal plays an essential role in synchronizing data transfers. The protocol scheme chosen should take advantage of this property to ensure a high sustainable transfer speed between communicating components and simple control logic. Also, when the communicating components operate on the same clock, which is often the case, then the protocol scheme chosen should not hinder existing synchronous hardware synthesis techniques from optimizing across component boundaries.

For performance and power considerations, the different components in an embedded architecture often operate at different clock frequencies. However, still in most cases today, the different clocks are derived from the same system clock $\phi_C$. This property should also be taken into consideration when designing the protocol scheme, again to ensure high transfer speed and simple control circuitry.

Considering the increasing difficulties with clock distribution, we must also consider the integration of synchronous components that are clocked by an unrelated clock (i.e. derived from a different crystal) or asynchronous components. The integration of asynchronous components is often necessary when communicating with external backplane system buses (e.g. VME bus) and other "off-chip" components.

The rest of this section is organized as follows. We first consider fully synchronous embedded architectures where all components are clocked by clocks that are derived (e.g. via clock division) from the same global system clock $\phi_C$, assuming negligible skew. Under this scenario, components may operate at different clock frequencies, tuned for performance and power, but with the restriction that all clocks are derived from the same clock. This is the most common situation today. We then next consider a more general setting where components may operate under unrelated clocks or asynchronously.

3.1 Synchronous wait transfer mode

In this section, we will consider the first scenario where the communicating components are synchronous and the clocking discipline used is a set of derived clocks that have an exact phase relationship with each other. We will further assume that the designs are positive edge-triggered and the components have registered outputs, which are common assumptions with today's commercial hardware synthesis tools (e.g. [11]).

Based on these assumptions, a CSP rendezvous style communication channel can be implemented in hardware using a simple synchronous transfer protocol called a synchronous wait protocol. We will first explain the protocol further assuming that both partners are operating on the same clock. We defer to sections 3.2 and 3.3 to describe how synchronization is automatically handled for coupling components that operate on different derived clock frequencies, unrelated clocks, or in unlocked asynchronous modes.

![Figure 2: a Channel based Component Architecture Model.](image)

![Figure 3: Channel implemented using a synchronous wait protocol.](image)
In this protocol, the sender and receiver partners synchronize the communication by a pair of sendRdy and recvRdy signals, as shown in Figure 3. The sender partner implements a send operation by setting its sendRdy signal high and placing valid data on the data lines. This is shown in Figure 4. If the receiver is not yet ready, as indicated by the input recvRdy signal being low, then the receiver enters into a "wait state" until the receiver is ready. This ensures synchronization. When the receiver is ready, as indicated by the input recvRdy signal being high, then the transfer is assumed to be completed in that clock cycle; thus the completion of the data transfer is implicit.

Similarly, the receiver partner implements a receive operation by setting its recvRdy signal high. This is shown in Figure 5. If the sender is not yet ready, as indicated by the input sendRdy signal being low, then the receiver enters into a wait state until the sender is ready. When the sender is ready, as indicated by the input sendRdy signal being high, then the receiver latches the data and moves to the next state.

The timing diagrams in Figure 6 further illustrate this protocol. In Figure 6(a), both partners are ready in the same state. In Figure 6(b), the receiver partner is ready, but it has to wait until the sender is also ready before the communication occurs. In Figure 6(c), the sender partner is ready first, but it has to wait until the receiver is ready before the communication occurs. While in a wait state, the sender ensures that the data remain valid on the data lines. This simple transfer protocol ensures that communication occurs only when both partners are ready. Note that in this protocol, there is no real distinction between which partner is the "master" and which is the "slave". Both the sender and receiver are in fact initiating the transfer by indicating to the other that it is "ready" for the transfer.

Despite its simplicity, the synchronous wait protocol offers several important advantages. One advantage is that the completion of communication is implicit. This means that when both partners are "ready", the communication behaves like a single cycle register transfer operation between the two components. "Burst" transfer modes, where the sender transfers consecutively a sequence of data to the receiver, can be implemented very efficiently. In fact, a new data can be transferred at a sustained rate of one item per cycle. This is in contrast to for example a handshaking protocol like the four-phase request-acknowledge scheme where the completion of the transfer has to be explicitly acknowledged. This protocol is widely used in asynchronous circuit implementations [14, 13] to synchronize communication in the absence of a clock. But when employed in a synchronous positive edge-triggered design, this request-acknowledge scheme requires at least two clock cycles for each transfer to account for the explicit acknowledgment. This problem can be partly circumvented if the channel control logic is clocked at twice the clock frequency by using both phases of the clock to trigger the logic. However, this will make the circuit more difficult to test and less amendable to conventional resynthesis by existing synchronous hardware synthesis techniques.

3.2 Synchronization between derived clocked components

When the communicating components are clocked by different clock frequencies, but the clocks used are derived from the same global system clock \( \phi_c \), then the above synchronous wait protocol can still be used. In our approach, we automatically synthesize a small channel adapter to handle the frequency conversion. This is shown in Figure 7. Using this adapter approach, we can integrate "pre-designed" reusable library components into a new custom embedded system architecture without the need to modify the description or the behavior of the reusable component itself, if the component has been
For handling this situation, we can use two strategies. In the first strategy, we use a higher clock frequency than the two clocks that is a common multiple of the two clock frequencies. Following on the \( \phi_S = 20\text{Mhz} \) and \( \phi_R = 30\text{Mhz} \) example, a common multiple clock frequency is a 60Mhz clock. This happens to be the system clock \( \phi_C \). We are guaranteed to be able to find a common multiple clock frequency because we assumed in the first place that all component clocks are derived from the system clock \( \phi_C \) by clock division. However, there may be another derived clock available in the system that has a lower common multiple clock frequency. In this case, the user can choose either clock to use for the adapter.

Let us refer to the chosen common multiple clock as \( \phi_M \). This clock is an integer multiple \( P \) times faster than the sender's clock \( \phi_S \), and an integer multiple \( Q \) times faster than the receiver's clock \( \phi_R \). In this scenario, the problem degenerates to a "Case 2" channel adapter for converting \( \phi_S \) to \( \phi_M \) and a "Case 1" channel adapter for converting \( \phi_M \) to \( \phi_R \). Both channel adapters are clocked at \( \phi_M \). They are then composed together to form a single channel adapter.

In a second strategy, we use instead a lower clock fre-
quence than the two clocks that is a common divisor of the two clock frequencies. Following again on the $\phi_S = 20\text{MHz}$ and $\phi_R = 30\text{MHz}$ example, a common divisor clock frequency can be a $10\text{MHz}$ clock. If there is already another derived clock available in the system that can be used as a common divisor clock, then that clock can be chosen. Otherwise, one can be derived by dividing either the $\phi_S$ or $\phi_R$ clock.

Let us refer to the chosen common divisor clock as $\phi_D$. Similar to the first strategy, the problem degenerates to a "Case 1" channel adapter for converting $\phi_S$ to $\phi_D$ and a "Case 2" channel adapter for converting $\phi_D$ to $\phi_R$. In this case, the sender's channel adapter is clocked at $\phi_S$ and the receiver's channel adapter is clocked at $\phi_R$. They are then composed together to form a single channel adapter.

3.3 Synchronization with unrelated clocked and unclocked components

In this section, we describe how we implement a channel adapter for interconnecting two communicating components that are operating on clocks that are unrelated (i.e. derived from a different crystal) or when one of the two components is an asynchronous self-timed component. Here we need to consider two cases that we will refer to as "Case 4" and "Case 5".

- Case 4. The sender and the receiver operate on unrelated clocks that have no known deterministic phase relationship with each other.
- Case 5. Either the sender or the receiver component is an unclocked "self-timed" asynchronous component.

These two cases are enumerated below.

Case 4. In this case, simply "stretching" a send or receive operation into a $N$-cycle operation is not sufficient. We need an explicit request-acknowledge scheme to indicate the initiation and completion of the communication. For this purpose, we use a four-phase request-acknowledge scheme as the intermediate handshake protocol. However, the components are still implemented, possibly stored in a library, using the synchronous wait protocol, and we can use them without modifying them. Instead, we implement a channel adapter that converts the synchronous wait protocol to a four-phase handshake protocol. This is depicted in Figure 8.

In a four-phase handshake protocol, channel communication is controlled by a pair of request and acknowledge signals, one for signaling a request, the other for signaling an acknowledge to that request [14, 13]. Here, we use a push version of the protocol where the sender assumes the responsibility for initiating the transfer. This protocol is shown in Figure 9.

As shown in Figure 8(a), the channel adapter for the sender is a finite state machine that is clocked with the sender's clock, which can be further optimized together with the sender. Similarly, the channel adapter for the receiver is a finite state machine this is clocked with the receiver's clock, which also can be further optimized together with the receiver. Because the clocks are unrelated, synchronizers are required to synchronize the req signal from the sender to the receiver, and the ack signal from the receiver to the sender. We assume that a robust synchronizer is available as a library element that can accept an asynchronous input signal and produce an output signal that is synchronous to the input clock. We can avoid inserting synchronizers for the data lines because the adapting clock ensures that the data is held stable during the entire four-phase period. The channel adapters for converting to a four-phase protocol is automatically synthesized in our approach. The two synthesized channel adapters can then be composed together to form an overall channel adapter, as shown in Figure 8(c).

Case 5. In the case that we need to communicate with an asynchronous component, we assume that component communicates using the four-phase handshake protocol. If not, a wrapper can be placed around the communication channel to convert the internal protocol to a four-phase protocol, using the techniques described in [13]. To interconnect a synchronous component that uses the synchronous wait protocol to an asynchronous component that uses the four-phase protocol, we use the method in "Case 4" to synthesize a channel adapter on the synchronous side to a four-phase protocol. Then we can connect it to the asynchronous component.

4 Software Component Architecture

4.1 Processor Template

A processor component unit may be a software programmable processor core (e.g. DSP or micro-controller cores). Our approach to incorporating software processors into a custom target embedded architecture is based on building a parameterized "architecture template" around the processor core. This architecture template is shown in Figure 10. There are three main components to this architecture template: the processor core itself, an internal memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the hardware communication interface to the external environment. These components are interconnected via the "processor bus", which consists of
Using an architecture template to model a software processor, a software component is seen to other components in our component architecture model simply as another hardware processor component that communicates via dedicated channels, and the channels are implemented using a common circuit-level channel protocol, i.e., the synchronous wait protocol. From the user’s perspective, the architecture template can be customized to provide a specified number of "physical" channels that support user-defined directions and data widths. In fact, it is actually implemented these physical channels for interconnection with other components. Based on this processor architecture model, a software processor component unit can be integrated into a custom target embedded architecture in the same way as another hardware processor component.

4.2 Details of the Processor Template

In our architecture template, the I/O unit implements the communication protocol between the processor core and the external environment. Communication with the external environment is accomplished through one or more input or output ports attached to the I/O unit. These input or output ports are connected to communication channels that are connected to other processor components. These ports implement channel control using the implementation protocol described in Section 3.1. A more detailed architecture template using the ARM processor is shown in Figure 11. The input and output of data to and from the processor core is accomplished through one of two different methods: memory-mapped I/O or instruction-programmed I/O.

Memory-Mapped I/O. Memory-mapped I/O provides a data-transfer mechanism that is convenient because it does not require the use of special processor instructions, and can implement practically as many input or output ports as desired. In memory-mapped I/O, portions of the address space are assigned to input and output ports. Reads and writes to those addresses are interpreted as commands to the I/O ports. "Sending" to a memory-mapped location involves effectively executing a "Store" instruction on a pseudo-memory location connected to an output port, and "Receiving" from a memory-mapped location involves effectively executing a "Load" instruction on a pseudo-memory location connected to an input port. When these memory operations are executed on the portions of address space assigned to memory-mapped I/O, the memory system ignores the operation. The I/O unit, however, sees the operation and performs the corresponding operation to the connected I/O ports.

For custom embedded architecture synthesis, the number of memory locations assigned for memory-mapped I/O will depend on the number of "channels" that a software processor component has to "physically" implement. Here, the assignment of address locations to channels can be user-defined. However, it is usually preferred that the address locations assigned for memory-mapped I/O be a "contiguous" memory region. This greatly simplifies the address decoding logic for the I/O unit. In this case, address location assignment is automatically performed by offsetting from a user-defined base address location.

Instruction-Programmed I/O. Some processors also provide special instructions for accessing special I/O ports provided with the processor itself. Using this scheme, these special communication ports of the processor are connected to the external channels via the I/O unit. We use a simple greedy algorithm (like in [3]) that uses these programmed I/O ports first if they are less expensive than using memory-mapped I/O. If communication via special programmed I/O instructions is more expensive, or not available, then only memory-mapped I/O will be used.

Interrupt Control. In addition to providing hardware support for memory-mapped and instruction-programmed I/O, the I/O unit also provides support for hardware interrupt control. Interrupts are used for different purposes, including the coordination of interrupt-driven I/O transfers, as described in Section 4.4. Different processors provide different degree of hardware interrupt support. Some processors provide direct access to a number of dedicated interrupt signals. Our I/O unit architecture makes use of these signals when available. If more interrupt "channels" are required, as for example required to support a number of interrupt-driven communication channels, we use the strategy of interrupt vectors. Interrupt vectors are pointers or addresses that tell the processor core where to jump to for the interrupt service routine.

Direct Memory Access Support. Optional to our
I/O unit architecture template is the addition of a direct memory access (DMA) controller. This DMA controller can access directly the data memory of the software processor component unit via the processor bus. It can be used to support high-speed data exchange directly between the (external) communication channels and the processor data memory without intervention of the processor. The processor initiates the DMA transfer by indicating to the DMA controller the start address of memory, the number of items to be transferred, the (external) communication channel where the transfer will take place, and the direction of transfer (i.e., reads or writes). Once initiated, the DMA transfers are accomplished between the specified communication channel and the memory. Upon completion, the DMA controller will interrupt the processor to indicate completion. If the DMA controller option is used, we will also produce automatically the necessary bus arbitration logic to manage the bus contention on the processor bus and the control logic to enable the processor to control the DMA controller.

4.3 Generation of I/O Unit

The I/O unit must be connected to the processor bus of the processor core. Different processors use different processor bus protocols, usually described as timing diagrams in data books, for implementing their memory "read" or "write" operations, or their special programmed I/O operations. Our interface synthesis tools can also automate the design of the necessary protocol matching hardware for communication with a specific processor bus protocol [13].

4.4 Software Communication Synthesis

On the software side, the processor can be programmed using the C or C++ language. To facilitate external communication, we automatically generate a custom library of C/C++ "call-able" functions with send and receive operations. This library is automatically generated depending on the number of channels that the software component must support, their directions, and their data width. The library can be thought of as a "customized" communication kernel. From the programmer's perspective, the application communicates with the external world by using the appropriate send and receive operations. This abstraction isolates the programmer from the low-level details of how the processor actually interacts with the environment. The send and receive routines are implemented using the memory "read" and "write" instructions if memory-mapped I/O is used for the specific channel, or the corresponding special programmed I/O instructions if instruction-programmed I/O is used. Because we use rendezvous semantics to implement the channels in our component architecture model, a communication operation to another processor component unit may take an arbitrary amount of time since both the "sender" and the "receiver" must be "ready". To avoid unnecessary "idling", we use an interrupt-driven I/O scheme. In this scheme, the "send" and "receive" routines in software are each split into two operations: the initiation and continuation operations. The initiator routine is responsible for getting an I/O operation started. In the case of a "send" operation, the processor transfers the data to the I/O unit. Once the data has been transferred to the I/O unit, the processor can proceed with other tasks if it is using a multi-tasking kernel, or a compiler can insert instructions at compiled time after the initiator operation that don't depend on the completion of the send operation. When the actual send operation to the external environment is completed, the I/O unit interrupts the processor. The continuation routine is then responsible for notifying the calling routine that the send operation has completed. Similarly, in the case of a "receive" operation, the processor initiates the operation by notifying the I/O unit. The I/O unit then coordinates the receive operation via the I/O ports along the specified channel. When the actual receive operation from the external environment is completed, the I/O unit interrupts the processor. The continuation routine is then responsible for transferring the data to the processor. To support the above I/O operations, the necessary interrupt controller functionalities are also synthesized into the I/O unit hardware. If DMA support is desired, then the necessary software routines for coordinating "burst" DMA transfers are also automatically generated. In addition to interrupt-driven I/O, polling and processor disabling are also supported.

5 Hardware Component Architecture

Hardware Communication Synthesis. In our target architecture model, a hardware processor component in an application-specific embedded system architecture may still need to be implemented. In this case, we automatically generate for the user a "container" that essentially implements a "communication wrapper" around the hardware that the user will later provide. From a user's perspective, the designer only needs to declare the number of communication channels required for the hardware processor component, their directions, and the data width that they must support. We then automatically generate a customized VHDL package that implements the communication channels, according to the synchronous wait protocol described in Section 3.1. This VHDL package provides a set of send and receive operations that can be "called" by an application program in VHDL, which provides an abstraction of the external communication. The designer can then "program" this "container" by writing a VHDL program that uses the send and receive operations provided by the VHDL package for external communication. If the designer chooses to use another programming language to program the hardware, e.g., Silage [8], then VHDL and the VHDL packages generated can be used as an intermediate interface to a lower level hardware implementation trajectory.

Parameterized Libraries. Reusable library components can be modeled and integrated into a custom embedded architecture. The library component can be in the form of synthesizable VHDL or already at the circuit level. The main requirement is that all external communication with the outside world must be implemented using the CSP channel concept (cf. Section 2), and the implementation protocol is the synchronous wait protocol described in Section 3.1. The components may operate under different clocks. Such components can be automatically integrated into a custom architecture, without the need to modify the description or the behavior of the reusable component itself, by synthesizing channel adapters using the techniques described in Section 3.2 and Section 3.3. To represent parameterizable components, we use the parameterization features of VHDL.
Communication Components. An important class of parameterized library hardware components is the class of communication components. For example, we keep in the library a parameterized synthesizable VHDL model of a FIFO communication buffer. This model is parameterized by depth and data width. It is designed to communicate using the CSP channel model and the synchronous wait protocol. This hardware "component" can be instantiated and integrated like any other hardware component. This way, we can mimic different communication models.

6 Application Experience

Our approach to the embedded architecture co-synthesis and system integration problems is embodied in a system called Symphony. It provides designers with tools to implement application-specific heterogeneous architectures and hardware/software communication layers. This system is part of a larger heterogeneous system codeign environment called CoWare under construction at IMEC [4]. In this section, we discuss the application of the proposed techniques to a simple error corrector for the Compact Disc player [12]. The error decoding algorithm consists of two phases. In the first phase, syndrome computation, a sequence of 28 or 32 symbols are read, depending on whether the sequence has been encoded once or twice. Then four 8-bit wide syndrome values are computed that contain the necessary information for error location and detection in the second phase. The reader can refer to [12] for more details.

We considered a partitioning, as is depicted in Figure 12, where the syndrome calculator and the error locator are implemented as two separate hardware components, while the error correction algorithm is implemented in software on an ARM core [16]. The software running on the ARM sends a sequence of symbols to the syndrome calculator along a 32-bit wide channel. In turn, the syndrome calculator sends its four syndrome values along four 8-bit wide channels to the error locator, both implemented in hardware. The latter then sends the necessary error location information along a 14-bit wide channel back to the ARM. Using our proposed method, we produced the design shown in schematic form in Figure 13. In this design, the ARM processor operates at the system clock frequency of 40MHz. The error correction algorithm was implemented in C on the ARM using the generated library of C-callable send and receive functions for communication. Both hardware components, the syndrome calculator and the error locator, operate at a 20MHz clock, which is derived from the system clock. These hardware components were programmed in VHDL using the send and receive operations from the VHDL communication packages synthesized.

Acknowledgements. We thank our colleagues K. Van Rompaey, I. Bolsens, G. Goossens, F. Catthoor, D. Verkest, B. Gyselinckx, K. Croes, T. Rolka, J. Silva, and C. Ykman for numerous insightful discussions on the system integration problem.

References