Compilation, Synthesis, and Simulation of Hardware Description Languages — The Compositional Models of HDL's

by

Szu-Tsung Cheng

B. Sc. (National Taiwan University, Taipei, Taiwan) 1989
M. Sc. (University of California, Berkeley, California, U.S.A.) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science
in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Alberto Sangiovanni-Vincentelli
Professor Dorit S. Hochbaum

Spring, 1998

The dissertation of Szu-Tsung Cheng is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring, 1998

Compilation, Synthesis, and Simulation of Hardware Description Languages — The Compositional Models of HDL's

Szu-Tsung Cheng

Abstract

Compilation, Synthesis, and Simulation of Hardware Description Languages — The Compositional Models of HDL's

by

Szu-Tsung Cheng

Doctor of Philosophy in
Computer Science
University of California, Berkeley

Professor Robert K. Brayton, Chair

With the advent of advanced CAD tools, people are now able to design multi-million gate chips. Generally, each of these tools has its specific view and model of the world. To represent designs in such a way that tools can understand and manipulate them, one needs to use certain languages. Hardware Description Languages (HDLs) like Verilog or VHDL are developed as description languages or simulator programming languages to describe the behavior of circuits at various abstraction levels. However, they suffer from the fact that they are based on the event-driven model, which does not match well with the Finite State Machine (FSM) model which is used by lots of synthesis, cycle-simulation, or verification engines. Therefore, one needs to maintain multiple "views" of a design in order to use different tools. This is inefficient and error-prone. On the other hand, there are languages that are based on FSMs or extensions of FSMs (sometimes these languages are also referred to as synchronous languages). Nonetheless, due to the causality (or combinational loops) problem that is intrinsic to these languages, one often needs to perform causality analysis (a global analysis, which is sometimes quite expensive for large designs) before feeding designs to the synthesis, verification, or simulation engines because lots of the core algorithms in these engines cannot take designs with combinational loops. Therefore, they do not scale well for large designs.

In this dissertation, we introduce a new hardware design language – V++. The heart of V++ is a new concept in hardware design languages: an automatically designed and implemented communications protocol, embedded by a compiler in the design itself.

This protocol plays a role in hardware design exactly equivalent to the role of the program stack in software: it permits transparent, automatic communication between modules in a hardware design, exactly as the program stack permits automatic, transparent communication between subprograms. This protocol is efficient, generalizes current design practice, and introduces little overhead in terms of the cycle time or the area of a typical system. Incorporating this protocol in the language itself eliminates a number of communication bugs, frees the designer from the task of writing communications code, and ensures that two communicating modules follow the same low-level protocol.

V++ is targeted at the general problem of hardware design: specifically, at the problem of representing and manipulating a design which is an assembly of a collection of predesigned components. V++ is thus designed with two objectives in mind:

1. Each program in V++ is directly interpreted as a network of communicating FSMs.
2. Programs in V++ are well-formed: the composition of two V++ programs is a V++ program, with well-defined, deterministic semantics.

At bottom, V++ is an extension of the synchronous languages to permit separate compilation for timing analysis, formal verification, cycle simulation, and synthesis.

We demonstrate V++'s major features. We address the problem of synthesizing hardware from V++ descriptions. We do so by first introducing a small set of kernel constructs and illustrating how circuitry can be built for these kernel constructs. We then show how V++ constructs can be compiled into these kernel constructs. We also provide an algorithm which can be used to check the legality (a program is legal if it does not contain loops which can be executed more than once in a clock cycle) of V++ programs. Communication device synthesis is then detailed. We show how to optimize the output circuits with a set of rule-based optimization techniques. We also show how V++ programs can be simulated. At last, V++ is applied to a real design – InfoPad, which is a wireless multimedia pager, and preliminary results are presented.

Professor Robert K. Brayton
Dissertation Committee Chairman

Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possbile without the help from many other people. First of all, I would like to give my deepest gratitude to my research advisor, Professor Robert K. Brayton, for his exceptional and inspirational guidance. I am grateful for his patience and kind help. He is the model from which I learn how to how to research, organize, and present ideas. Actually, I named my first son after Bob, hoping that he will someday follow Bob's foot steps.

I sincerely thank Professor Alberto Sangiovanni-Vincentelli for introducing me to the world of circuit simulation and logic synthesis. I also thank him for being on my qualification and thesis committees. Professor Dorit Hochbaum kindly agreed to be a memeber of my thesis committee. I am grateful to her for taking time to read my dissertation. I wish to thank Professor Katherine Yelick and Professor Shmuel Oren for being members of my qualification committee.

I have been priviledged to work with Dr. Patrick C. McGeer. He generously shared his thoughts with me and zealously guided me through the tedious process of getting the thesis done. To me, he is not only my manager, but also my advisor, and my best friend. I also want to give my special thanks to Tom Truman. He has provided several keen insights and comments from a designer's point of view for the V++ project.

I would like to thank the outstanding research scientists of Cadence Berkeley Laboratories for numerous informational and educational discussions. They are Patrick Scaglia, Alexander Saldanha, Felice Balarin, Kenneth McMillan, Jerry Burch, Professor Fabio Somenzi, Vigyan Singhal, Norris Yip, Luciano Lavagno, Mike Meyer, James Rowson, Howard Wong-Toi, Ellen Sentovich, Tiziano Villa, Yosinori Watanabe. I thank Kristine Lammanno from CBL and Flora Oviedo, Brad Krebs, Elise Mills from UCB for their kind and amiable help. Thanks to the many distinguished students, the Berkeley CAD group has become a place packed with intelligence and fun. They are, in alphabetic order, Adnan Aziz, Ramin Hojati, Timothy Kam, Sriram Krishnan, William Lam, Rajeev Murgai, Rajeev Ranjan, Tom Shiple, Serdar Tasiran, Tiziano Vialla, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pado, Shaz Qadeer, Shaker Sarwary, Gitanjali Swamy, Wilsin Gosti.

I would like to acknowledge the support from Cadence, Simens, CA Micro, and Fujitsu during the early stage of my graduate study. Among them, I want to give my special thanks to Cadence for their supporting the work.

My life in Berkeley would not have been so joyful without my friends. Thanks to my friends, Wan-teh Chang, Wei-Li and Lynn Li, Joseph King, Shan-Hsi Huang, Huey-Yih and Chiao-Lu Wang, Robert Wang, Chip-Po Wen, and Szu-Min Li.

Finally, I am grateful to my parents, Fu-Chun Chen and Hsi-Mien Tsai for their encouragement and supports. Without their persistent help and care, this thesis would not exist. I would also like to give my special thanks to my wife, Li-Ling Jeng for her patience, understanding, and love. I want to thank my sons, Bao-Jeng Robert and Mao-Heng Michael Cheng for giving me so much joy in my life.

Chapter 1

Introduction

Digital systems are highly complex. The process of digital design is difficult and error-prone. With the help of powerful tools like synthesizers ([Sea92]), simulators ([ver95, CMM+96]), verifiers ([McM94]), floor planners/routers, schematic editors, etc., it is now possible to design multi-million gate chips. Generally, each of these tools has its specific view and model of the world. To represent designs in such a way that a tool can understand and manipulate them, one needs to use certain languages. A Hardware Description Language (HDL) is a language used to describe the behavior of a piece of hardware so that it can be processed by the supporting CAD tools.

1.1 Traditional Hardware Description Languages

1.1.1 The Model

Historically, hardware designs have been entered using hardware description languages such as VHDL [vhd88] or Verilog [TM91]. As the name suggests, a HDL program was regarded as a description or model of the design. It also served as a simulation harness. These HDL's are based on a powerful, "universal" simulation mechanism - event queues. Event-driven simulation [VD75, Ulr78] is a simple yet powerful model with which digital behavior at many levels of abstraction can co-exist and be simulated.

1.1.2 The Tools

Verilog and VHDL are the most widely used languages in which designs are described, simulated, and exchanged. The powerful simulation mechanism makes the languages not only hardware description languages, but also programming languages. Designers can describe mixed levels of abstraction of circuits as well as test harnesses (the parts of designs which do not correspond to hardware, e.g., the I/O processing code which is used to set up stimulus to a circuit) in the same language, and the whole thing can be simulated seamlessly. Due to their versatility, these event driven simulators have become an indispensable part of current design flow.

1.1.3 The Semantic Problem

However, these powerful languages do have their problems. At the beginning, VHDL and Verilog (especially Verilog) suffered from the problem of being without a formal model. This means that given a program, it is hard to determine the meaning of a piece of code by simply reading the text without running the program. Even worse, the underlying simulation model can introduce nondeterminism, which makes it hard to determine if a bad simulation trace is a false alarm due to simulator induced nondeterminism or a design bug.

For example, given the following Verilog fragments:

always @(**posedge** clk) a = b;
always @(**posedge** clk) b = c;

The `always` processes say that, when the positive edge of `clk` arrives, assign the current value of `b` to `a` and `c` to `b`. Intuitively, these are two flip-flops in a row. Nonetheless, different schedules of the statements can give different answers. Suppose the first `always` statement is executed first, then `a` will get the current value of `b` and `b` will get the current value of `c`. On the other hand, if the second `always` statement is executed first, then `b` will first get the current value of `c` and then `a` gets `b`'s new value, which is equal to `c`. This kind of simulator introduced nondeterminism has increased the difficulty of debugging. [Bak93, BGM94, BFK94, Doh94, Eve94], etc. have tried to give semantics to these languages, formally or informally.

1.1.4 The Model Mismatch Problem

In the 80's, VHDL (Very High Speed Integrated Circuit HDL) was created by the U.S. Department of Defense (DOD) as a standard programming language for describing the function and structure of digital circuits. The goal was to make a circuit self documenting, so that blocks can be reusable with new technologies. In the mean time, Verilog HDL was created as a circuit simulator language by Philip R. Moorby. Due to their powerful underlying models, both languages are capable of describing designs at various abstraction levels. Because of their versatility, they soon became the most popular languages for digital design. In addition to being used as simulator programming languages so that designs can be simulated, these languages are also used as *contract languages* among companies. That is, these languages are used to sign-off chips (to specify "golden" behavior or timing models several parties agree upon).

However, another defect of these powerful languages comes from their strength – they are too powerful to match up perfectly with hardware. One can easily use these languages to write programs which have no corresponding hardware at all (e.g., test harnesses which read stimulus vectors). In effect, these languages are *simulator programming languages* where designs are programs that control the scheduling of events which model the circuit activities at various levels. Even though much effort has been invested to discover the formal semantics of such languages (or subsets of them) [Doh94, Eve94, BGM94], due to different models of computation, it does not help hardware extraction. Such languages cannot be the contract language between designers and CAD (Computer Aided Design) tools since no tool can derive hardware from an arbitrary description.

To use these languages for both simulation and hardware synthesis, one needs to build *hardware extractors* for HDL programs. As already pointed out, since the semantics of these languages are so rich, there are programs for which no hardware can exhibit the same behavior. Therefore, these extractors cannot handle the full set of these HDLs. In addition, due to the model mismatch between event-driven languages and hardware, these extractors need to do complex analysis to understand the role and the meaning of a construct in various contexts. By choosing different algorithms, different systems [syn94, GS96, vhd92, CB94, CBY$^+$95, CB96, CSM$^+$97], etc. have different capabilities for dealing with different combinations of HDL constructs. Moreover, each of these subsets is a semantic subset rather than syntactic subset, which again leads to several problems. First, without running a synthesizer, it is hard to guess if a program fragment can be correctly synthesized by the synthesizers as the semantics of a construct can change in different context. Second and even worse, sometimes it is impossible to know if the synthesized circuit is consistent with the simulation traces without doing expensive analysis (sometimes exhaustive simulation) on the pre-synthesis and post-synthesis versions of the designs. Consider the following scenario. Given a `case` statement in Verilog, the most straightforward translation of the statement is a priority decoder. However, in case a designer knows that only one branch can be matched at a time, the directive `synopsys_parallel_case` can be put so that the synthesizer can make use of the "don't care" condition (it will never happen that two or more branches get matched simultaneously) to produce more compact circuits. If this user provided directive is false, the synthesizer will interpret the circuit differently from the simulator. To determine if an overlapped condition can ever be reached, one needs to do reachability analysis in the most general settings. For real circuits, the complexity of the analysis can be formidably high. A similar situation also happens when one provides hints regarding which signals serve as synchronous/asynchronous set/reset during latch synthesis.

1.2 Synchronous Languages

1.2.1 The Model Supported By A Rich Set Of Tools

Fortunately, we do have models and languages that can be easily and unambiguously mapped to logic. They are the finite state machines (FSMs) and the synchronous languages. The fundamental assumption in these models/languages is that statements/logic execute arbitrarily fast in one "tick" (or one clock cycle) until they reach a stable state. In fact, this condition, which is similar to Martin's "Isochronic Fork", simply requires that the logic execute sufficiently quickly that the input to all latches are quiescent at the time latches are updated. This is a common condition in digital hardware, and all design methodologies, of which we are aware, insist upon it.

In the world of synthesis [SSL$^+$92], formal verification [ABB$^+$94, McM94, BHSv$^+$96a, BHSV$^+$96b], and cycle-based simulation [CMM$^+$96], finite state machines (FSMs) is the most widely used model. FSMs are the fundamental formalism under "synchronous languages" like BLIF-MV [BCH$^+$91], SMV [McM94], Esterel [BC84, BdS91], Signal [Hal93], Statecharts [Har87, DH89], etc. In addition, the *synthesizable subsets* of Verilog or VHDL [syn94, vhd92, CBY$^+$95] are also essentially FSMs.

The powerful algorithms in these CAD tools have automated and sped up the design process. The only major limiting factor for these tools seems to be the size of the circuits they can handle. One way to get around the problem is to decompose designs into smaller, manageable pieces so that tools can operate on sub-circuits and generate results that can be easily combined together. However, due to the following constraint, this approach may be limited in the general settings.

1.2.2 Causality Checking Problem

Traditional FSMs suffered from two major drawbacks – a flat, non-hierarchical structure and obscured concurrency. The synchronous languages described above have greatly overcome these drawbacks. However, they still suffer from one problem that is intrinsic to the underlying FSM model.

The problem with these FSM-based languages is that one can get undefined or nondeterministic behavior when composing two modules even though each by itself is well-defined. *The problem is due to the fact that unrestricted communication among modules is permitted.* The simplest example of this phenomenon is a RS latch formed by cross-coupling two NAND gates. When R=S=1, the state of the latch is indeterminate. Any language which permits a cross-coupled NAND gate today also permits loops of arbitrary size. This problem also arises when one composes two Mealy machines together. In the terminology of the synchronous languages, these loops are known as *causality violations* or *combinational cycles.*

The root of this difficulty is the so-called RMC Barrier Theorem [Bak96], which states, in essence, that no semantics can be simultaneously *Responsive* (i.e., value changes in one module are instantly visible in another), *Modular* (internals of each module are hidden), and *Causal* (composition of modules is well-defined).

Removal of combinational cycles is an important step to many CAD algorithms which can only accept inputs without combinational cycles. Detection and removal of causality violations generally requires a global analysis [Shi96, Mal93] of the design. While up to now, global analysis has been possible, if somewhat time-consuming, two factors are converging which limit the technique:

- The existence of very large designs. Currently, the largest designs are in the range of 5 million gates, or 20 million transistors. Within the next few years 20,000,000 gate designs will be common. Such designs will be associated with descriptions of millions of lines of code. Global analysis of such a design would be impractical, if not impossible.

- An emerging market is in predesigned blocks, or, "intellectual property". Most IP vendors have no desire to ship source code; rather, they wish to ship netlists and object code models of their blocks.

These two factors imply that global analysis of designs are no longer adequate. *Black-box* modularity (where the contents of each block are hidden from the rest of the design) will replace the current glass-box modularity over the next few years. Emerging tools must support black-box modularity. This implies that the communication between components of a design must be constrained to support black-box modularity.

1.3 Lessons Learned from Software Languages

Consider the problem of modularity and composition in the software community. The problem is to design a mechanism whereby subprograms communicate. The mechanism must isolate subprograms (so that programmers can write their routines without worrying about activity external to it), be reasonably efficient to implement, and relatively free of error. In addition, such mechanism must make debugging easy so that programmers can focus on the core computation of a program without looking at the details of how communication is done. The correct design of such a mechanism is far from obvious, and still less obvious is the notion that the communication mechanism can or should be done automatically and efficiently. Various mechanisms might be imagined, and most are used in the software community and were prevalent up through the past decade. The one that prevailed – through use of an intermediate storage mechanism, the stack – was among the least intuitive and the least likely to be manually designed. Nonetheless, it is ubiquitous today – and very few programmers have ever consciously used it. The mechanics of using the stack are somewhat complex, and at first glance fairly expensive. In principle, the items to be passed to a subprogram are placed on the stack, then consumed by the subprogram. However, all of this is transparent to the programmer who simply writes a function call and the compiler does the rest.

1.3.1 Stacks As A Universal Procedure Interface

The importance of this step is easy to underestimate. This simple innovation – the stack – was arguably a key enabling technology for the software explosion of the '70's and '80's. Before such a mechanism was in place, programs could not be easily assembled from components; software libraries could not be distributed; programs could not be written in multiple languages. Today, of course, all these practices are widespread. The abstraction of the communication mechanism via the stack, originally conceived to permit recursion, did far more than simply free programmers from the tedious and time-consuming task of providing the glue between subprograms. By providing a simple, standard interface it permitted software re-use.

The lesson from software languages is clear: in order for the world of block-based design to come into being, a simple, ubiquitous, automatically-generated communication mechanism must be implemented in finite-state systems. This mechanism must be efficient, lightweight, with clean semantics, well understood by system builders, and a generalization of current design practice.

By providing a standard communication mechanism, systems can be partitioned into a set of computation islands. The synthesized interfaces connect these islands by a very simple protocol: the sender puts the arguments on a bus before the current clock cycle ends, the receiver then has valid arguments to consume throughout the next cycle (or the cycles following that). By enforcing non-zero delay on these communication devices (we refer to these communication devices as *latch walls* later), the immediate influence of a gate is limited to the logical block containing the gate. That is, the scope of global analysis can be restricted to the islands. This limited global analysis provides several advantages.

First, for synthesis, the problem size for causality analysis can be limited (to each individual object, instead of the whole design). Second, this provides one way to limit the scope for algorithms like false path analysis [MB91, McG89], etc. In addition, latch walls also make it easy to put pieces together to form various sources.

For cycle simulation, having latch walls around islands gives us more freedom in scheduling islands. In essence, when compiling designs for simulation, one does not need to compile the whole design to get the whole schedule. Instead, each island can serve as the unit of scheduling and it can be arbitrarily scheduled. As we will see in the dissertation, the current version of the V++ (our new hardware design language) compiler only uses one of several possible schedules – a bottom-up schedule along the instantiation hierarchy. The bottom-up schedule was chosen simply due to the special rule we used to export combinational wires from children to parents.

Of course, the penalty here is the latency induced by message exchange. But given the advantages gained from the strategy, this approach is reasonable for large scale system designs for which design turn-around time, rather than performance, is the major concern. If latency between islands is a concern, the two or more islands can be clustered together to form one island (at the expense of complicating the design).

1.3.2 Structured Programming Enables Powerful Optimization

Another thing that makes software compilers successful is the adoption of powerful optimization techniques [ASU85, FRJL88] such as register allocation, loop optimization (e.g., loop invariant identification), copy propagation, dead code elimination, etc. Most of these techniques will not be applicable if the structure of a program is obscure to a compiler. In that case, the only way a compiler can understand the structure of a program is to use reachability analysis, which is undecidable for general programming languages.

The use of "weaker" control-flow operators such as `if/else` conditionals and `while` loops instead of the more general `goto` statements makes the flow of programs explicit and amenable to flow analysis and optimization. Even though most of these optimization algorithms tend to be conservative, for many cases the compiler optimized code is close to or even better than manually crafted code in terms of efficiency or memory usage.

The lesson is: by making the abstract flow of a program explicit and using conservative analysis, one can get compact and efficient results from large designs with moderate computation.

1.4 V++ – A Hardware Design Language

In this dissertation, a *Hardware Design Language*, V++ [CMM+97, CMM+98, CMS+98, CMSS98], is designed to fix the problems mentioned in Section 1.1 and 1.2. Specifically, the goals of V++ are as follows:

1. To be a mechanism for *capturing* the design, not *modeling* the design. Hardware description languages like VHDL or Verilog were principally targeted at design modeling;

the design itself was represented elsewhere. Contrast this with a high-level language program. In the programmer's mind, there is no distinction between the program and the implementation. *The program is the implementation.* Similarly, a V++ program should be the designer's view of the part to be implemented.

2. To present a single, well-defined, tool-neutral semantics to the user. Verilog abused its user community by having its semantics defined by the operations of the Verilog simulator [ver95]. VHDL did somewhat better: the standard body simultaneously defined the simulator and the language, so that at least there was an accurate reference manual [vhd88]. However, the semantics were still defined in terms of simulation, which led to inconsistencies between the behavior of a part's code and the behavior of that part. A better approach defines the semantics independent of an interpretation tool: indeed, in terms of the underlying implementation medium, digital hardware at some level of abstraction.

3. To solve the problem of compositionality – the behavior of two well-defined V++ parts, when composed, is well-defined.

4. To make the execution flow explicit – this enables the use of well-developed optimization techniques on larger designs.

V++ was designed with the lessons learned from software programming languages mentioned in Section 1.3. In particular, V++ was designed with a universal communication device, the stack, in mind.

1.4.1 The Hardware "Stack"

The characteristic of a hardware signal is that it is an infinite series of values, varying over time; the principal difficulty of an interface is in ensuring that the writer of a signal holds its value until all readers have thoroughly sampled it. The key, therefore, is that the synthesized interface takes the responsibility of ensuring that the sampled signal remains stable through the sampling interval. The simplest way to ensure this is with a latch, synthesized by the compiler; conceptually, each input of each module is latched. Each signal writer writes the latch input; the synthesized interface holds the signal. The reader of the signal is thus isolated from the writers, exactly in the manner that a subroutine is isolated from its caller via the stack.

What is novel in the language is the idea that a mechanism of arbitered FIFO's be implicit in the semantics of a Hardware Design Language in exactly the manner that a stack is implicit in the semantics of a programming language; and that this will permit the use of an Application Hardware Interface (AHI) hardware market analogous to the API's in the software market.

What is different from software programming languages is that in software, a routine uses program counters to know if it gets valid inputs to perform computation. In hardware, since all connections are made statically at compile/link time before it starts to run and since all components are executing simultaneously, an extra flag/token is used to indicate the transfer of control.

1.4.2 The World of V++

The fundamental unit in V++ is the *object*. An object corresponds to a collection of *ports* and *variables*. The set of variables defines the states of an object while the set of ports defines the external view of the object. An object is the fundamental unit of compilation in a design: that is, the full source code for an object, together with the interfaces of the objects with which it communicates in the design, is sufficient to realize a hardware implementation or complete simulation model of the object.

Communication between objects in a design is primarily via *messages* (or *port invocations*). Unlike messages in an object-oriented programming language [Str91, Gos96] where a message send is equivalent to a method or procedure invocation (which is *blocking*, i.e, a caller waits until the callee finishes its job before it can regain control), message-passing in V++ is just that: a message is sent to another object, or to itself. All message transmissions are *non-blocking*; a message is simply sent to the appropriate port of each object. Successful transmission of a message means that the synthesized communication device (*channel*) has contracted to deliver the message to the relevant recipient port and the message will be automatically held for the recipient until it is consumed. Some latency is associated with such a transmission: V++ does not, in general, guarantee instantaneously delivery of messages.

Note that V++ is a hardware design language. That is, the amount of resources needed to implement a design should be determined at compile/link time. Therefore, object creation is static and the structure of the communication network will be determined at

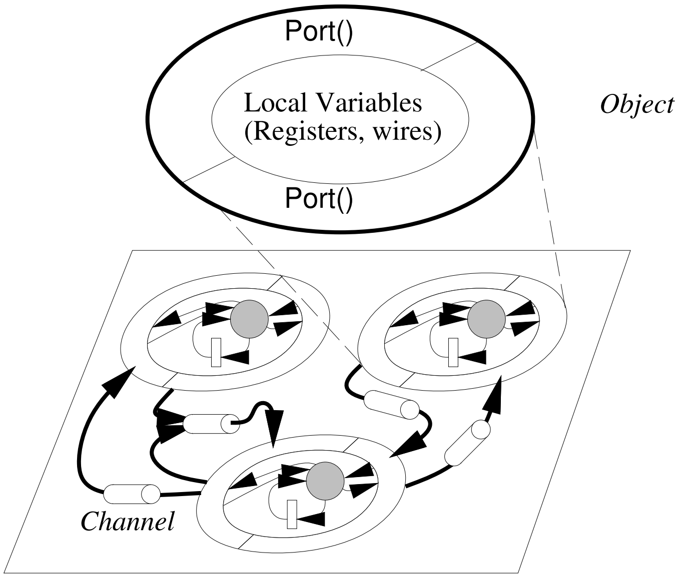


Figure 1.1: Objects, Port, Channels, and Variables - Thick arrows indicate message sends (port invocation). Hollow lines are determined after the whole design is compiled and linked (i.e., they are not dynamic). Gray circles denote combinational logic. Vertical rectangles denote latches.

link time. Also note that port invocation (message send) does not involve the instantiation of the recipient port. Therefore, "recursive" port invocation (a port sending messages to itself, directly or indirectly) is allowed. Refer to Figure 1.1 for a graphical illustration of the relationship among objects, ports, variables, and channels. Chapter 2 gives an overview of the V++ language.

1.4.2.1 Communication in V++

Communication via messages in V++ is fundamentally Moore-style: that is, communication through latches. As pointed out in [Bak96], this is sufficient to ensure compositionality. However, on occasions Mealy-style communication is desired. It is this unlatched communication which is the source of the difficulty in compositionality in previous languages. Thus, we desire to permit a restricted form of Mealy communication which provides the functionality desired but does not admit compositionality problems.

The mechanism for this is the *public wire variable*. A *variable* of a V++ object may only be assigned a value by code within the object. However, *public* variables are *visible* to *ancestors* of the object in the containment hierarchy.

To see that this restricted form of Mealy communication does not admit compositionality problems, note that such problems *always* are exhibited in the form of combinational loops: that is, a closed sequence of wired connections in a design without an intervening latch. In this form of Mealy-style communication, wires always run strictly *up* the design hierarchy: never down nor across the hierarchy. Thus, no loops are possible. Further, since the wired inputs to an object must come from the object's children, it is easy to schedule simulation code in V++ for levelized cycle simulation: schedule the code for the children of an object, then the code for the object itself.

These are the only two methods of communication in V++: since neither alone can cause a compositionality problem, and since they do not interact, V++ is compositionally safe, achieving our third major goal.

1.4.2.2 Computation in V++

As mentioned earlier, the synchronous languages are easily mapped to logic: therefore, unsurprisingly, within an object V++ is a synchronous language. That is, its model of computation is as a series of logic equations, which simultaneously solve out to a well-defined state. The restrictions on communication in V++ guarantee that the state is well-defined.

Any of the synchronous languages would suffice for our purposes: for reasons of convenience and clarity we chose a language that is pretty close to SMV [McM94]. SMV is a purely equational language. Users of synchronous languages may find it helpful to think of a V++ object as an SMV program: the message-sending mechanism implements a means by which SMV programs can communicate.

As pointed out earlier, an important assumption of this architecture is that *code within an object executes in zero time.* This "synchronous assumption" which is fundamental to the synchronous languages is also true for V++. This condition leads to a two-stage evaluation procedure: first, logic evaluates, i.e. arrives at its well defined solution. Second, latches update and messages are sent.

1.4.2.3 Flow of Execution in V++

In hardware, everything happens concurrently. However, people think sequentially. Therefore, when designers design circuits, they tend to abstract the behavior of the design as a sequence of sets of operations. For example, consider a traffic light controller. It sequentially turns on and off the green, yellow, and red lights in response to the car sensors and the internal timer. Conceptually, one can partition such a design into two parts – the sequencer (controller) and the data-path. At each instant, the sequencer activates appropriate data-path components to do the computation according to its current state and the sensor reading. That is, the sequencer controls the sequencing (flow) of data-path activation (execution). All activated components perform computation simultaneously. These data-path components in term define the values of variables.

The flow of execution in V++ is made explicit by using *processes* (Section 2.4). A V++ process is essentially a sequencer plus data-path components. In addition to the general data-path operators, V++ incorporates control constructs to make the flow of execution explicit. V++ provides constructs to explicitly express parallelism and organize the transitions in sequencers. V++ processes provide designers with a simple yet powerful way to describe multi-cycle operations. Some of these process constructs are inspired by the Esterel [BC84] language.

1.5 Organization of The Dissertation

The rest of the dissertation is organized as follows. Chapter 2 gives a tour of the major constructs of V++. Compilation and link algorithms are detailed in Chapter 3. Experiment results and conclusions are then shown in Chapter 4.

Chapter 2

A Tour of The V++ Language

In this chapter, we first give an overview of the constructs in the V++ language. In Section 2.1, the fundamental units of compilation, *objects*, is introduced. Section 2.2 then introduces *channels*, which are the communication devices that hook up objects. In Section 2.3, *attributes* are introduced. It can be used to fine-tune the hardware that are synthesized from V++ programs. Section 2.4 presents *processes*, which are powerful constructs to describe object reactions that span multiple cycles. In Section 2.5, *assertions* are used to specify properties of V++ designs. These properties can be checked by simulators or formal verifiers, or they can be used by synthesis tools as don't cares to generate more compact circuits. Then in Section 2.6, V++ is applied to various designs.

2.1 Objects

In V++, *objects* are the units of compilation. Each object contains a collection of *object members* (Section 2.1.1) which are *variables*, *clocks*, and *ports*. Similar to the traditional "modular" HDL like Verilog, VHDL, or SMV, an object is a bundle of definitions that can be reused (when the object is *instantiated*). What is different from these languages is that a V++ object does not have I/O wires. In V++, communication is done via *port invocations*. *Ports* in V++ are similar to member functions/methods [Str91, Gos96] in object-oriented languages. Ports define how objects can react to port invocations by modifying variables in the containing objects. They define the services an object provides.

2.1.1 Object Members

In an object, each variable is an instance of an object, a channel, or a fundamental data type. That is, a variable is a piece of hardware. The set of variables in an object define the state of the object. To change the state of an object, ports are provided. A port is a collection of control and datapath logic that manipulate variables.

The following example declares an object `FSM` which consists of one bit latch variable `state`. Note that keywords are in bold face.

```
object FSM {
   latch bit state;
}
```

A fundamental data typed variable can be either *latches* or *wires* by decorating the declaration with `latch` or `wire`. Latch variables are those which hold their values unless explicitly assigned. These variables sample their new values at discrete time points. Wire variables are unlatched and revert to default values (0) unless assigned. By default, a variable is a wire if it is not explicitly specified. In the above `FSM` example, if `state` is intended to be a wire, then it should be declared with:

```
wire bit state;
```

The visibility of variables and ports can be modified by *visibility modifiers* like `public` or `private`. If a variable is public, then that variable is visible to the outside world. If a variable is private, that variable can be accessed (read or written) only within the enclosing object. By default, all variables are public unless declared private.

Objects are instantiated by variable declarations with the types replaced by appropriate object names. For example:

```
object Controller {
   FSM foo = new FSM();
}
```

In the above example, in object `Controller`, an instance named `foo` of object `FSM` is created.

2.1.1.1 Constructors

A constructor is a piece of code that is invoked only when a new instance of an object is created. In V++, as all hardware instances are created at compile/link time, constructors are executed only by the compiler/linker. Actually, they are linker programs. Each constructor has the same name as the containing object. In addition, constructors are always public. Constructors are used to connect the instances of objects, to determine the initial values of latches, and to do initial port invocations. Consider the following example,

```
object Controller {
   FSM foo;
   Controller (FSM f) {
      self.foo = f;
   }
}
```

Like the previous example, `Controller` contains a variable `foo` of type `FSM`. However, unlike the previous example, `foo` is not a fresh instance. Instead, it "points" to another instance that is passed in (the formal argument `f` of `Controller`'s constructor) by the object that instantiates `Controller`. The keyword `self` refers to the current instance of the enclosing object (`Controller`).

Initializing Latches

Latches are initialized by assigning to the `init` fields of the variables in an object constructor. For instance:

```
object FSM {
   latch bit state;
   FSM() {
      state.init = 1;
   }
}
```

In the above example, latch `state` in object `FSM` is initialized to 1.

2.1.1.2 Parameter Passing – Generators

To facilitate the description of reusable objects, *parameters* are introduced. Parameters allow designers to change the widths of variables, the number of iterations of loops, or replace constants with programmable variables. What is different from ordinary variables is that parameters should be reducible to constants at compile/link time. If this is not the case, then the program will be rejected by the V++ linker.

For parameterized designs, the generator construct (`generate`) is used to define the values of parameters.

```
1    object FSM {
2       bit[3:0] width;
3       generate(bit[3:0] wid) {
4          self.width = wid;
5       }
6       latch bit[width-1:0] state;
7    }
8    object Controller {
9       FSM{2} foo = new FSM();
10   }
```

In the above example, `foo` is an instance of object `FSM`. Parameter (`{2}` at line 9) is passed to the `FSM` instance and is bounded to parameter `width` in the generator at lines 3-5 (this is done by first binding 2 to `wid`, the formal argument of port `generate`. When an object instance is created, the actual arguments that are enclosed in `{}` will be bound to the formal arguments of the `generate` port of the object that is being instantiated. Then in `generate`, `wid` is assigned to `width`). Therefore, the variable `state` (line 6) in `FSM` is a two-bit variable.

2.1.2 Ports

While the variables of an object define the state of the object, ports in the object define its behavior. A port receives zero or more arguments from senders and changes the values of variables in the enclosing object on the senders' behalf. It is the object's interface to the world. It exports the services provided by an object. As an example:

```
object FSM {
```

```
2    latch bit state; wire bit cond;
3    clock clk;
4    public count() @(clk) when (cond) {
5       // the port body, to be executed when 'count' is active.
6    }
7 }
```

Line 3 declares a variable `clk` of type `clock`. `clock` variables, which are essentially clock signals in circuits, are used to control the progress of systems. Lines 4-6 declare a port `count` that is visible to the external world. When the port is *active*, it consumes the input arguments and executes its port body statements. The port is synchronized by `clk`. That is, when one invokes `count`, it will not become *active* until `clk` arrives. The `when` clause defines the condition under which the body of the port may be active. It is called the *guard* of port `count`. The guard is conjuncted with the signal indicating the presence of input arguments to the port and the clock signal to determine if a port is active. More precisely, the port will be *active* only when (1) there is a valid argument that is not consumed by it yet, (2) `clk` is asserted, and (3) the *guard* `cond` is true.

2.1.2.1 Inside Ports

A port consists of a set of statements. These statements are the logic that changes the values of variables. The way statements change the values of variables is via assignments. For example, the `count` port in the FSM example may contain the following conditional statement.

```
public count() @(clk) {
   if (state < MAX) {
      state = state+1;
   } else {
      state = 0; // If counter reaches MAX, reset it and stop counting.
   }
}
```

The `if/else` statement says that if the current value of `state` is less than `MAX`, then the new value of `state` (which will be effective when the next `clk` tick comes) will be the current value of `state` plus 1. Otherwise `state` will get reset.

In a port, all assignments must follow the *single assignment rule* (multiple assignments from different ports do not need to follow this rule, refer to Section 2.1.2.5). That is, statements like:

```
foo = 1;
if (cond) foo = 0;
```

are illegal since `foo` is assigned twice when `cond` is true. On the other hand,

```
if (cond) foo = 0;
else foo = 1;
```

is legal since assignments to `foo` are under mutually exclusive conditions. Since the checking for mutual exclusiveness of assignment conditions is conservative, the following statements

```
if (cond) foo = 0;
if (!cond) foo = 1;
```

are also considered illegal.

Basically, V++ follows the "synchronous assumption" (Section 1.4.2.2). That is, all statements execute in zero time. If the LHS (left hand side) of an assignment is a wire, then the assignment is effective immediately. On the other hand, if the LHS of the assignment is a latch, then the assignment is not effective until the next clock cycle. For example,

```
latch bit a, b;
a = b;
b = a;
```

will swap the value of `a` and `b`, but only at the next clock tick. As another example, consider the following V++ fragment.

```
latch bit a, c;
wire bit b;
a = b;
b = c;
```

Even though `b` is assigned "after" (syntactically) it is read, at the next clock cycle, `a` will get the current value of `c`. This is due to the fact that all assignments to wires take effect instantaneously. Thus the values assigned to the wires are independent of the order of their appearance in the code.

2.1.2.2 Port Invocations – Sending Messages to A Port

A port (sender) can activate another port (receiver) by *port invocation*. In this dissertation, we also refer to the action of port invocation as *message sending*. What is done during port invocation is that the sender port sets up arguments (the *message*) to be passed to the receiver, and hands them over to a *channel*. Then the channel (essentially a communication device with internal buffers; more of channels in Section 2.2) delivers the arguments to the receiver port and turns on a flag indicating that a message is available so that the receiver can become active and consume the arguments when it is ready. As an example, the finite state machine `counter` may look like:

```
1  object FSM {
2     latch bit state; wire bit newState;
3     clock clk;
4     public count(bit keepCounting) @(clk) {
5        if (keepCounting) {
6           newState = state+1;
7           state = newState;
8           self.count(newState<MAX);
9        } else {
10          state = 0; // if counter reaches max, reset it and stop counting.
11       }
12    }
13 }
```

The `count` port takes a one-bit argument `keepCounting` which instructs if `count` should increase `state` or reset it. At line 8, port `count` invokes itself (`self.count`) with the argument `newState<MAX`. That is, whenever `count` is invoked by the external world, it starts counting by sending itself messages until it reaches the maximum (1 in this case), then it stops counting and the counter `state` is reset.

In a port, the *single port invocation rule* (it is forbidden to have multiple port invocations toward the same destination port at the same time) has to be followed. This rule is similar to the single assignment rule in Section 2.1.2.1. That is, V++ code segments like:

```
foo(bar);
foo(baz);
```

are forbidden.

Initial Port Invocation

To start the execution of a V++ program, one needs to do *initial port invocation* in the constructor of the containing object. In the above `FSM` object, if we want to initialize `state` to 0 and start counting from the beginning, then we may have constructors similar to the following code segment:

```
FSM () {
   state.init = 0;
   count(1);
}
```

Note that for initial port invocations, all that the V++ compiler/linker does is to set the initial states of the latches in the channels. That is, no extra hardware is allocated for these statements.

2.1.2.3 Conditional Statements

Conditional statements are useful in controlling the effectiveness of one or more groups of statements. In V++, there are two conditional statements – the `if/else` statements and `switch` statements.

The simplest form of conditional statements is the `if/else` statements:

```
if ( <conditional-expression> ) <statement1> [ else <statement2> ]
```

Note the square bracket around the `else` branch of the conditional statement indicates that the branch is optional. An `if` statement without matching `else` branch is said to be *incomplete*. If `<conditional-expression>` evaluates to 1 (true), then `<statement1>` is executed, otherwise `<statement2>` is executed. That is,

```
if (s) o = a; else o = b;
```

amounts to

```
o = s ? a : b;
```

In terms of hardware, the above statement corresponds to a multiplexer with output `o` and inputs `a` and `b`. The multiplexer is controlled by `s`. In the above conditional statement, if the `else` branch is omitted, then `o` will fall back to its default value when `s` is 0.

Nested conditional statements are allowed. Therefore, it may cause confusion when statements similar to the following are given:

```
if (c1)
   if (c2)
      o = a;
else
      o = b;
```

It is unclear whether the `else` branch belongs to the first `if (c1)` statement or the second `if (c2)` statement. In V++, it is defined that an `else` branch always corresponds to the "closest" (in syntactic order) incomplete `if` statement. Therefore, in the above example, the `else` branch belongs to the `if (c2)` statement.

A more complex form of conditional statements is the `switch` statements. In a `switch` statement, the first case tag that matches with the switch expression is executed. That is, a `switch` statement of the form:

```
switch (<switch-expr>) {
   case (<tag1>) : <statement1>
   case (<tag2>) : <statement2>
   ...
   case (<tagn>) : <statementn>
   default : <statement_default>
}
```

amounts to

```
if (<switch-expr> == <tag1>) <statement1>
else if (<switch-expr> == <tag2>) <statement2>
...
else if (<switch-expr> == <tagn>) <statementn>
else <statement_default>
```

2.1.2.4 Constructor Loops

In V++, `array` loops are available to describe arrays of regular structures. These loops are expanded by the compiler/linker at compile/link time. For example,

```
bit[2:1] a;
bit[2:1] b;
bit[2:0] c;
bit[2:1] s;
bit[1:0] i;
...
c[0] = 0;
array (i=1; i<=2; i=i+1) {
   c[i] = (a[i] & b[i]) | (b[i] & c[i-1]) | (a[i] & c[i-1]);
   s[i] = a[i] ^ b[i] ^ c[i-1];
}
```

is a two-bit carry ripple adder. `a` and `b` are the two operands, `c` is the carry signal (`c[0]` is carry-in and `c[2]` is carry-out), and `s` is the sum. Since `array` constructs are expanded statically, their bounds should be able to be reduced to constants at compile/link time.

2.1.2.5 Interaction Among Ports

In V++, a variable can be touched by several ports. In case there is contention, the value of the variable will be determined by a V++ compiler generated resolution logic. In the mean time, V++ warns about such contention among ports. Basically, the port with the highest priority determines the new value of the variable. By default, the port that is declared first in an object will have the highest priority. Priorities can be changed by using the `priority` keyword:

```
1 object FSM {
2     latch bit state;
3     clock clk;
4     public count() @(clk) {
5         if (state < MAX) {
6             state = state+1;
7             self.count();
8         } else {
9             state = 0; // if counter reaches max, reset it and stop counting.
10        }
11    }
12
13    public setCounter(bit value) @(clk) {
14        state = value;
15    }
16
17    priority setCounter > count;
18 }
```

In the example, a new port `setCounter` is added. When `setCounter` is invoked, it forces the next value of `state` to be `value`, which is `setCounter`'s argument. By default, if the priority statement at line 17 does not exist, when both `count` and `setCounter` are active, `setCounter` has no effect on `state` since `count` has higher priority (it is declared first). However, `priority` statement at line 17 makes `setCounter` have higher priority.

In case no port is assigning to a variable, the variable falls back to its default value. For a `latch` variable, the default value is its current value. For a `wire` variable, 0 is the default value. The default values for wires can be changed by attaching `default` clause to the declaration.

```
wire bit[3:0] foo default 1;
```

declares a 4-bit variable `foo` whose default value is 1.

2.1.2.6 Combinational Ports

Generally, there is a unit (or more) delay between when a port is invoked and the time it becomes active and executed. If a port is not synchronized by a clock (i.e., without the `@` clause), it is called a *combinational port*. Otherwise it is a *sequential port*. A combinational port can still be guarded by a `when` clause. Invoking a combinational port involves zero delay. That is, the channel that delivers arguments to a combinational port does not utilize latches to buffer arguments (refer to Section 2.2). In addition, senders to a combinational port cannot determine if the message has been successfully delivered. Therefore, in case a combinational port is blocked (e.g., the guarding condition is not met), the incoming request can get lost.

All combinational ports are private, i.e., they can only be accessed in the containing object. This restriction ensures that all objects are Moore machines at their boundaries.

The combinational port is powerful in the sense that one can squeeze lots of computation in one clock cycle. However, one should also be aware of the potentially long combinational paths introduced by a series of combinational port invocation.

The following object uses combinational ports to implement the FSM shown above.

```
object FSM {
    latch bit state;
    wire bit nextState;
    clock clk;
    public count() @(clk) { // a sequential port
        if (state < MAX) {
            self.count();
        }
        computeNS();
        state = nextState;
    }

    computeNS() { // a combinational port
        if (state < MAX) {
            nextState = state+1;
        } else {
```

```
            nextState = 0;
        }
    }
}
```

The port `computeNS` is a combinational port which updates the value of `nextState`. Port `count` picks up the value of `nextState` computed by `computeNS` and updates the value of `state`.

2.1.2.7 Functions

An alternative way of describing combinational logic is to use *functions*. Like combinational ports, all functions are private. Also, invocation of functions induces zero delay. A function is declared by the `function` keyword, followed by the width of the return value, the name of the function, the arguments to the function, and the function body. The following example implements the `FSM` object using functions.

```
object FSM {
    latch bit [1:0] state;
    clock clk;
    public count() @(clk) {
        if (state < MAX) {
            state = fullAdder(state[0], 1, 0);
            self.count();
        } else {
            state = 0;
        }
    }

    function bit[1:0] fullAdder(bit i1, bit i2, bit ci) {
        // a,b is variable concatenation
        fullAdder = ((i1&i2)|(i2&ci)|ci&i1),i1^i2^ci);
    }
}
```

In the above code, the adder at line 6 (of the `FSM` object in Section 2.1.2.2) is replaced by a function call to `fullAdder` whose definition is located at lines 13-16. Note that a function returns values by assigning to the name of the function (line 15).

Comparison between Combinational Ports and Functions

There are some differences between functions and combinational ports. Basically, each call to a function instantiates a non-shared copy of hardware which implements the functionality. On the other hand, a combinational port can be potentially shared by several ports. That is, via combinational ports, different ports can send messages to the same piece of combinational logic without duplicating the logic. However, all messages going into a combinational port need to go through a resolution logic. Whereas the inputs to functions are directly wired to the function body.

2.1.2.8 Port Sharing - When A Port Is Invoked by Multiple Sources

If a port is receiving messages from two or more ports, then the channel that is delivering messages has to resolve the possible contention that multiple ports can send messages to the same destination at the same time. Consider the following example:

```
object counter {
   clock clk;
   public start() @(clk) {
      doCount(4'b0000); // 4-bit literal, value 0000
   }
   doCount(bit[3:0] value) @(clk) {
      if (value != 4'b1111) {
         doCount(value+1);
      }
   }
   privilege(doCount) doCount > start ;
}
```

Each time port `start` is invoked, it invokes `doCount` to count from 0 to 15. `doCount` counts by sending itself messages. Once the counter value reaches 15, `doCount` stops counting.

port/variable	Operation					
	(1)	(2)	(3)	(4)	(5)	(6)
private in child	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\times$
public in child	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
private in parent	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\surd$
public in parent	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\surd$
port argument [1]	$\times$	$\times$	$\times$	$\times$	n.a. [2]	n.a. [2]

Table 2.1: Visibility Modifiers. The operations are: (1) read by child, (2) write by child, (3) read by parent, (4) write by parent, (5) invoked by child, and (6) invoked by parent. Operations (1)-(4) are applicable to data variables. Operations (5)-(6) are applicable to ports.

Instead of the storing the state of the counter in a latch variable, the current counter value is stored in the `value` argument of `doCount`. Note that port `doCount()` is invoked both from sequential port `start` (line 4) and sequential port `doCount` (line 8). If both ports try to invoke `doCount` at the same time, they need to be resolved. By default, in a compiler/linker synthesized channel, a priority decoder is used to arbitrate among multiple port invocations. If it is desired to explicitly control the priority among multiple port invocations with the same target port, then `privilege` construct can be used. In the above example, the `privilege` statement at line 11 states that, among those port invocations for `doCount`, port `doCount` has higher priority than port `start`. That is, once the counter starts counting, it will not be interrupted. If one wants to control more details of channels, *user defined channels* (Section 2.2.4) can be used.

Comparison of `priority` and `privilege`

In Section 2.1.2.5, keyword `priority` is introduced to explicitly specify the priorities of ports in terms of their ability in influencing the values of variables. Similarly, in Section 2.1.2.8, keyword `privilege` is used to specify the priorities of ports. However, `privilege` determines which port invocation can go through in case there are multiple contending port invocations toward the same destination port.

[1]Arguments of a port are visible only in the port. In addition, port arguments are only readable (only channels can write them).

[2]Not applicable.

2.1.3 Controlling What Is Visible from Outside An Object – The Variable Visibility Rules

In V++, by default, all hardware inside an object (wires, registers, or ports) are visible from outside the object. However, problems may arise when other objects are using variables that are intended for internal use only (imagine what will happen if another object is allowed to observe the internal of a encryption chip). To avoid the misuse of hardware resources, one can control the visibility of variables or ports by using *visibility modifiers*.

The visibility modifiers (`public` and `private`) specify the accessibility of member variables and ports in objects. Table 2.1 summarizes how visibility modifiers affect the accessibility of resources.

2.1.3.1 Latched Inputs

For certain designs, it is desirable that the sender and the receiver be loosely coupled in such a way that no explicit exchange of data is performed between the two. Instead, the sender simply drives a variable so that the receiver can latch it if it wants to. If this kind of communication happens inside one object, then a simple latch will serve the purpose. However, if this happens between a pair of ports belonging to different objects, then according Table 2.1, this cannot be done if a parent tries to drive a variable and makes it visible to its child. To get around the problem, *latched inputs* are introduced.

Latched inputs are latches declared inside an object that are modifiable to the parent of the object (in the instantiation hierarchy). The object containing the input latch can control the enable of the latch. That is, while the parent sets the value of the latch, the child determines when the latch samples the input by setting the enable signal associated with the latched input variable. By default, the enable signal is glued to constant one if not explicitly assigned. Consider the following program:

```
object foo {
   clock clk;
   bar bi = new bar(); // create a new instance of bar (Section 2.1.1)
   a(bit i) @(clk) {
      bi.iLat = i;
   }
```

```
7   }
8   object bar {
9       input latch bit iLat;
10  }
```

In the above example, whenever port `foo.a` is invoked, latch `bi.iLat` (variable `iLat` of object `bar`) is set to the argument `i` of port `foo.a` (line 5). In object `bar`, associated with input latch `iLat` is an enable signal named `iLat.enable`. `iLat.enable` is private to object `bar`. If this signal is asserted, then latch `iLat` will load whatever value that is assigned by its parent. Otherwise the latch retains its previous value.

2.2 Channels

When one port sends a message to another port, the message is delivered via a *channel*. Channels are communication networks by which ports can talk to each other. They take the responsibility of faithfully delivering messages from invoking ports to invoked ports once the messages are sampled. Unlike ports or objects, channels are not translated into hardware until the linkage phase when the whole design is available. In other words, objects direct compilers to generate hardware while channels direct linkers to generate hardware which is glued with the modules generated by the compiler.

2.2.1 Default Channels

The default channel generated by the V++ compiler is a multiple-writer-single-reader communication device with buffering queues. Consider the following example,

```
object two_guys {
    clock clk;
    latch bit[3:0] count;

    sender() @(clk) {
        self.receiver(count);
        count = count + 1;
    }
```

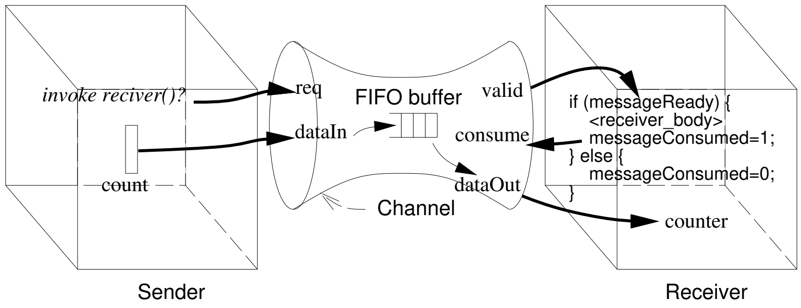


Figure 2.1: V++ compiler generated communication mechanism.

```
9     receiver(bit[3:0] count) @(clk) {
10    }
11  }
```

At line 6, port `receiver` is invoked with the current value of `count` as its argument (or we can say that a message is to be delivered from `sender` port to `receiver` port). By default, V++ generates a channel which uses FIFO buffers to queue the messages to be sent. When the message arrives at `receiver` and when `clk` is high, `receiver` becomes active and the message is consumed. Figure 2.1 illustrates how port invocation (line 6) is compiled. Note that the default port invocation is non-blocking. That is, the invocation always terminates in one clock cycle.

When the port invocation at line 6 is executed, `req` is asserted and the arbitration logic in channel has to determine if it has enough capacity to process the request. If so, the data (`count`) is stored in a FIFO that resides in the channel and the channel will assert `valid` signal for `receiver` in the next clock cycle. The `receiver` body statement will be enabled when `valid` is held high. On finishing the execution, the `receiver` asserts `consume` so that the channel can advance its FIFO pointer.

The default message send involves minimum interaction between the port invocation statement and the channel in the sense that the port invocation does not query the "return" value from a message delivery attempt. That is, the port invocation has no idea if the arguments has been successfully sampled. Therefore, in case the buffer is full when one tries to send a message, the message will get lost. In Section 2.2.2, *safe channels* are provided to prevent messages from getting lost.

2.2.2 Safe Channels - Channels That Do Not Drop Messages on The Floor

By default, when a V++ port sends a message through a channel, it asserts a request signal and holds data at the same time. If the message cannot be delivered, the sender will not be aware of the undelivered message. That is, if a sender sends a message without checking for acknowledgment, the data can potentially get lost if the channel cannot deliver the message (e.g., if another port with higher priority is also sending via the channel at the same time or the channel buffer is full).

To eliminate the possibility of message loss, a mechanism to catch message sending failure and control the digestion of port inputs is provided. The key is in a set of auxiliary latches called the *shadow registers*.

In a port, for each safe port invocation, a set of auxiliary latches (shadow registers) are allocated to store the values that are to be passed to the channel. If the message can be delivered in one clock cycle, the values stored in the registers will not be used. However, in case in the next clock cycle the arbitration logic in the channel determines that the data cannot be latched properly, the data will be latched into the shadow registers and the shadow registers will keep requesting the channel for message delivery in the following clock cycle(s) until it succeeds. When the shadow registers are trying to send the stored messages, the sending port blocks itself from consuming incoming data. Consider the following fragment.

```
object two_guys {
   clock clk;
   wire bit global;
   sender(bit in1) @(clk) {
      safe receiver(in1);
   }
   receiver(bit in2) @(clk) {
   }
}
```

It is compiled into the circuits similar to the ones shown in Figure 2.2. The flag `ack`, which is one clock lag behind the corresponding request, indicates if the channel has a chance to sample the data to be delivered. If `ack` is low one clock after a message send request is

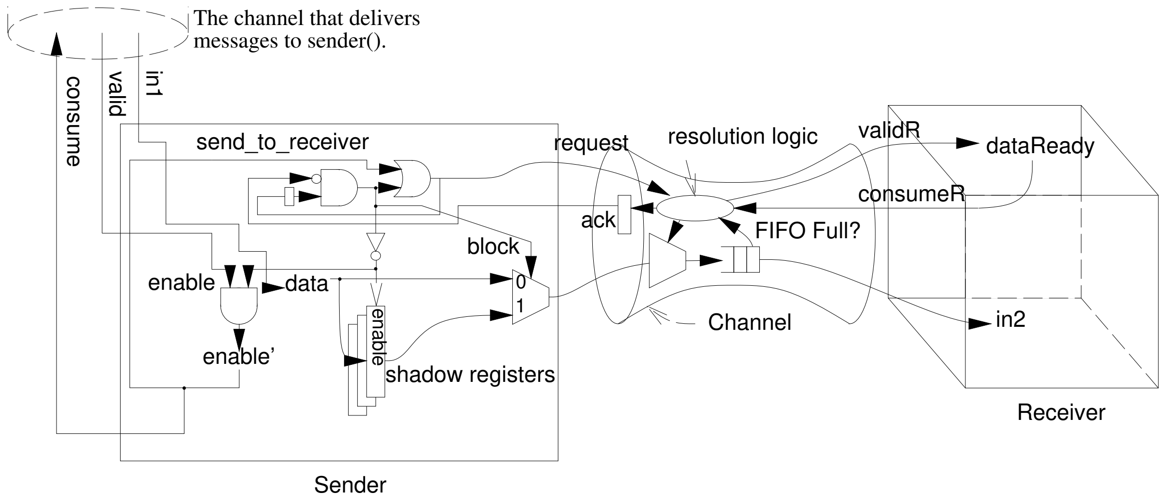


Figure 2.2: Safe send with shadow registers.

asserted, then the `sender` port switches to a busy sending mode in which it stops consuming any incoming messages (by setting `enable'` to low) and keeps sending the data (from the shadow registers) until `ack` is set high. The time `ack` is set, the sender can immediately resume execution. Note that if `sender` uses variable `global` as an argument to `receiver` at cycle i and is blocked, then the value of `global` at the i-th cycle will be used as the argument for the resends that follow (because the shadow register has latched the value of `global` at the i-th cycle), no matter what `global`'s values at cycles $i+1$, $i+2$, ... are going to be.

Waveforms of the above circuit are shown in Figure 2.3. In Figure 2.3, it is assumed that at the i-th cycle, the FIFO for the sender in Figure 2.2 is already full. Therefore, when `sender` tries to send a message at the i-th cycle, `ack` will be held low at the $i+1$-st cycle. `sender` is therefore blocked from consuming its input at the $i+1$-st cycle, if there is any. In the meantime, `sender` keeps trying the pending send by holding `request` and `data` lines. At the $i+1$-st cycle, since the `consume` bit from `sender` is held low, the input channel to `sender` has to hold the `valid` bit high (and input data to `sender`) at the $i+2$-th cycle, asserting that a valid token is available. Assume that at the $i+2$-th cycle, `ack` signal is set, indicating that data from the i-th cycle has been latched. At this time, `sender` then continues to consume the second message on its input channel and requests another message send. The waveforms show that the second request goes through without any trouble.

Now, suppose that a port contains a mixture of safe (blocking) and default (non-

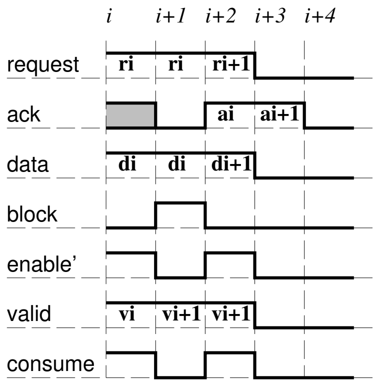


Figure 2.3: Waveforms for signals in Figure 2.2.

blocking) port invocations at the same time:

```
object two_guys {
    clock clk;
    latch bit l;
    wire bit w;
    sender(bit in1) @(clk) {
        safe receiverS(in1);
        l = in1 + 1;
        w = in1 - 1;
        receiverU(in1);
    }
    receiverS(bit in2) @(clk) { }
    receiverU(bit in3) @(clk) { }
}
```

Assume that at the i-th cycle `sender` tries to send a message to `receiverS` but fails (and `sender` will be blocked at the $i+1$-st cycle). `w` still gets assigned to `in1-1` at the i-th cycle and `l` gets `in1+1` (at the $i+1$-st cycle, since it is a latch). In addition, a message is sent to `receiverU` with the value of `in1` at the i-th cycle. Assume that there are one or more safe port invocations at the same time, then the sending port cannot take new messages until all the safe port invocations go through.

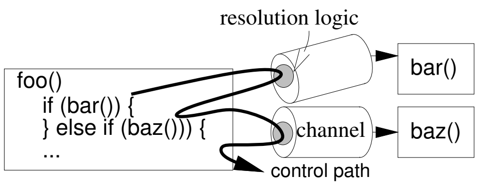
2.2.3 Alternative Safe Channel Proposals and Their Problems

Apart from the safe port invocation scheme presented in Section 2.2.2, several candidates implementing safe port invocations are also considered. In this section, we review these schemes and their problems.

2.2.3.1 Get Channel Status As Return Value

The most straightforward way to check if a message send is successful is to use the acknowledgment signal (unlatched) from the channel as the return value for a port invocation so that the V++ program flow can take into account the case that a message cannot be delivered. For example, in the following code segment, port `foo` first sends a message to `bar`. If the message cannot be sampled by the channel, then a 0 is returned, which will enable `foo` to try another port `baz`.

```
foo () @(clk) {
  if (bar()) {
  } else if (baz()) {
  }
}
```



The advantage of this approach is that the circuit for this is simple and the semantics is powerful. One can invoke a port, call another port (like error handler) if the first invocation can not succeed. All of these can be done in the same clock cycle. In addition, one can implement an arbitrarily complex sequence of "try-and-handle" actions using deeply nested if/else statements and port invocations. However, the "zig-zag" control wire (as shown in the above figure) that weaves together the channels and port `foo` can be arbitrarily long. The long combinational path can render a severe problem to the design.

2.2.3.2 Instantaneous Blocking Send

One way to get around the potentially long combinational path shown in the previous section is to use the acknowledgment signal from the channel to control the progression of the sending block as shown in Figure 2.4. However, this can lead to "backward combinational leakage" from the receiver to the sender port (dashed line). If `sender` and `receiver` invoke each other, then a combinational loop will be formed, even if the two ports are in different objects. This is the last thing we want in V++.

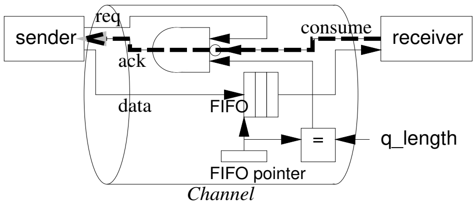


Figure 2.4: A leaking path from receiver to sender (dashed line) introduced by delay-less blocking send. In this figure, the `valid` signal to the `receiver` port is not drawn as it is irrelevant to the discussion.

One way to prevent the backward combinational path sneaking between senders and receivers is to insert a latch on the acknowledgment signal (`ack`). However, this extra latch will cut the performance of the sender to half of its top throughput. This is because for each safe port invocation, it has to spend at least another cycle waiting for the returning acknowledgment before it can proceed and do the next computation. Note that the safe port invocation mechanism shown in Section 2.2.2 does not have this performance degradation problem.

2.2.3.3 Prerequest

Inserting latches on the acknowledgment signal alleviates the backward combinational path problem shown in Section 2.2.3.2. However, as is indicated in Section 2.2.3.2, this can degrade the performance. One way to save the extra cycle for each safe send is to assert the request line (it is also referred to as "pre-request" in the rest of this section) one clock before the port really needs to use the channel. The following code implements this pre-request mechanism by attaching a small combinational circuit (`requireNext` clause) to the `sender` port to raise the request signal to the appropriate channel (channel `ch1`, which is the channel that delivers messages to `receiver` port, as is indicated by line 4) one cycle before `sender` actually does port invocation. The execution of the sender is then guarded by the acknowledgment from the channel (`ch1.ack`).

```
1 object two_guys {
2    sender() @(clk) when(ch1.ack) requireNext(ch1,1) {
3       latch bit[3:0] count;
4       ch1:  receiver(count);
```

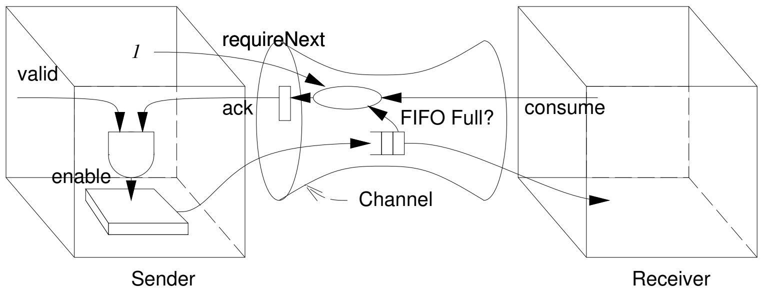


Figure 2.5: A safe channel implemented by guarding the execution of the sender by ready signal from the receiver.

```
5     }
6     receiver(bit[3:0] counter) @(clk) {
7     }
8 }
```

`requireNext` (line 2) is a keyword that keeps sending requests (the second argument, which should be a boolean expression) to the first argument (channel `ch1`). `requireNext` tells `ch1` that, in the next clock cycle, it is going to send a message and requests for permission. The channel `ch1` sends acknowledgment signal via variable `ch1.ack` (`when` clause, line 2) to guard the execution of `sender`. That is, if `ch1` does not grant the transmission (e.g., due to buffer overflow), `sender` cannot be active and initiate message sending. Therefore, messages will not get lost. However, the pre-request approach has the following drawback:

- Too conservative - One cycle before the execution of the port body, the sending port needs to lock the channels that will be used. However, as the body is not executed yet, all it can do is to lock all the channels that can potentially be used in the future. This can result in holding more resources than is necessary. Obviously, not only can resources be wasted, performance can also be degraded.

2.2.4 User Defined Channels

Like stacks in programming languages, channels perform like a standard communication glue binding different objects (ports). However, in the hardware world, the requirement for communication glues for different designs may vary drastically. It is therefore desirable to allow designers to create their own glues. *User defined channels* in V++

allow users to devise their own communication devices.

```
1 object two_guys {
2     clock clk;
3     arbiter a = new arbiter(); // arbiter is a user defined channel
4     latch bit[3:0] count;
5
6     sender() @(clk) {
7         self.receiver(count) via a;
8         count = count + 1;
9     }
10    receiver(bit[3:0] counter) @(clk) {
11        // message is received from the user defined channel a.
12    }
13 }
```

Note that the port invocation at line 7 is followed by a clause `via a`, where `a` is an instance of a user defined channel `arbiter`. This statement sends a message to `receiver` port by using channel `a`. Note that one port can take messages only from one channel. Therefore, if another port wants to send messages to `receiver`, the messages must be delivered by `a`, too. The following is the code for channel `arbiter`:

```
channel arbiter(clkSnd, req, ack, dataIn)
        (clkRcv, valid, consume, dataOut) {
    // left, right are compiler provided attributes for arrays.
    clock[] clkSnd;
    input bit[] req;
    output bit[] ack;
    input bit[][3:0] dataIn;
    clock[] clkRcv;
    output latch bit[] valid;
    input bit[] consume;
    output bit[][3:0] dataOut;
    wire bit[ack.left+1:ack.right][3:0] retVal;
```

```
13    latch bit[3:0] q; // the 'FIFO'
14    wire bit[ack.left+1:ack.right] reqPending;
15    wire bit[3:0] i;
16
17    run() @(clkR[0]) {
18       retVal[0] = 0; reqPending[0] = 0;
19       array (i=req.right; i<=req.left; i=i+1) {
20          // the resolution logic is simply a wired-or of all input data.
21          if (req[i]) {
22             retVal[i+1] = retVal[i] | dataIn[i];
23             reqPending[i+1] = 1;
24          } else {
25             retVal[i+1] = retVal[i];
26             reqPending[i+1] = reqPending[i];
27          }
28       }
29       // note:  new data always overwrite q.
30       q = retVal[req.left+1];
31       if (reqPending[req.left+1]) {
32          // retVal is sent (latched) to the receiver.
33          dataOut[0] = q;
34          valid[0] = 1;
35       } else if (consume[0]) {
36          valid[0] = 0;
37       }
38    }
39 }
```

The `arbiter` channel has two sets of arguments. The first set of arguments (*sender interface*, line 1) are from the senders that send messages via the channel. The second set of arguments (*receiver interface*, line 2) are glued to the receiver end.

For the first group of arguments, `clkSnd` is an array of clocks from sender ports, `req` is an array of send-requests, and `ack` is an array of acknowledgment signals from the channel (the pairs of square brackets [] in lines 4-7 represent arrays whose dimensions cannot not be determined yet. Actually, the dimensions of these variables will be determined by the linker.). `req` is asserted when a sender requests to send a message via the channel. `clkSnd`, `req`, and `ack` are not explicitly shown in the argument list of a port invocation. They will be used by the V++ compiler/linker to facilitate the handshaking between objects and channels. `dataIn` is an array of the data to be sent. One of its elements (`dataIn[0]` in this example) will be glued to variable `count` of port `sender` of object `two_guys` (line 7). The dimensions of these variables (`clkSnd`, `req`, `ack`, and `dataIn`) are equal to the number of senders that are hooked up to the channel.

For the second group of arguments, the first argument (`clkRcv`) is the receiver's controlling clock. The second argument (`valid`) indicates if a message is valid so that the receiving port can be activated. `valid` is latched and clocked by the receiver's controlling clock. The third argument (`consume`), which is an input to the channel, indicates if the receiver has consumed the message. If the message is consumed and there is no further message to be sent to the receiver, `valid` will get reset. The fourth argument (`dataOut`) is the data to be passed to the receiver. It is glued with the receiver port's input arguments.

`req.left` and `req.right` (line 19) are the leftmost and rightmost indices of array `req`, respectively. `req` has as many elements as the number of sending ports that are hooked up to the channel. Therefore, `req.left` and `req.right` will be determined only at link time.

The `array` statement at line 19 is simply a iterative constructor which is expanded according to the range determined by its lower bound and upper bound. The body of the `array` statement is copied as many times as specified by the range of the `array` construct.

In the user defined channel, there is a group of statements named `run` which is guarded by `@(clkR[0])` (line 17). This group of statements is executed only when the guarding clock is asserted. In a channel, there may be multiple `run` groups.

In the above channel example, the FIFO for buffering messages is a simple latch (`q` at line 13). The resolution logic is a wired-or of all the data from the ports that are trying to send messages. The connection between the ports (`sender` and `receiver`) and the user defined channel `arbiter` is similar to the one shown in Figure 2.1 (and Figure 3.13.a).

2.2.5 Broadcast - Sending Messages To Channels

The usual port invocation is appropriate for normal point-to-point communication. However, under the circumstances that one wants to do broadcast, the port invocation mechanism that has been provided so far may not suffice. To broadcast a message, one sends it to a channel, instead of sending it to a specific port and lets the channel to determine the right port(s) that will receive the message.

```
channel ch (cin, req, ack, din) (cout, valid, consume, dout) {
   clock[] cin;
   input bit[] req;
   output bit[] ack;
   input bit[] din;
   clock[] cout;
   output bit[] valid;
   input bit[] consume;
   output bit[] dout;
   ...
}
object foo {
   ch ci = new ch();
   wire ack;
   clock clk;
   a(bit in) @(clk) {
      ci(clk, 1, ack, in)();
   }
   x() @(clk) {
      bit validx; bit outx;
      ci()(clk, validx, 1, outx);
   }
   y() @(clk) {
      bit validy; bit outy;
      ci()(clk, validy, 1, outy);
   }
```

syntax	*meaning*
Behavior modifying attributes:	
depth(<number>)	The channel that delivers messages for the port will have an FIFO buffer with capacity <number>.
destructive	When the input buffer is full, new incoming message will destroy the last buffer entry.
Status checking attributes:	
running	This indicates whether the process is running and not suspended (due to suspended safe port invocation).
Variable characteristic:	
left/right	They apply to array variables and are used to determine the bounds of array variables.

Table 2.2: Attributes

```
27    }
```

In this example, the channel ci invocation (line 17) connects the arguments (clk, 1, ack, and in of port a) to the sender interface of channel ci (cin, req, ack, and din, respectively). The channel invocation does not contribute to the second group of arguments (the receiver interface). Therefore, the second group of argument is empty (() at line 17). When a is invoked, it requests ci to send data in. ci then has to figure out where the data should go. Similarly, by connecting to the receiver interface, one can do ci()(clk, valid, consume, out) in a port to get data out of channel ci. In the above example, both ports x and y take data from ci by connecting to the receiver interface (lines 21 and 25). If ci wants to broadcast to both x and y, it asserts both validx and validy (as well as outx and outy so that x and y can get the right data) at the same time. If desired, a channel invocation can contribute both to the sender interface and the receiver interface.

2.3 Attributes

There are occasions on which designers want to fine-tune the behavior of synthesized hardware. For example, one may want to allocate a specific number of FIFO entries in a specific channel. To access the details of a V++ design and fine-tune the behavior of synthesized hardware, *attributes* can be used. The attributes that are available in V++ are listed in Table 2.2.

There are three classes of attributes. First, there are attributes that designers can set to change the synthesized hardware. For example, one can use `depth` property to fine-tune the number of FIFO entries of channels:

```
public count() @(clk) depth(7) { }
```

In the previous example, the `count` port declaration explicitly specifies the size of the FIFOs of the channel which delivers messages to `count` to be 7.

Attributes can also be used to check the status of ports. For example:

```
if (count.running) {
   receiver();
}
```

says that if port `count` is running, then send a message to `receiver`.

Finally, attributes can be used to access the characteristics of variables. For example, given an array variable `foo`, `foo.left` and `foo.right` give the left and right boundary indices of its highest dimension, respectively.

2.4 Processes

V++ ports are suitable for describing a circuit's single cycle reaction on receiving a message. However, it is sometimes desirable to describe reactions that span multiple cycles. In this case, *processes* are provided to describe such multi-cycle reactions.

A process basically overlays a controller FSM over a set of data-path components. At each clock cycle, the controller selects some of the data-path components for execution. The selected data-path components then change the values of variables. The V++ language provides a set of powerful constructs to help organize the controller FSM transitions.

2.4.1 Introducing Delays in Processes

In a process, a `wait` statement is used to decompose a group of statements into multi-cycle execution sequences. Consider the following process.

```
process foo(bit in) @(clk) when(enable1) {
    <stmt1>
    wait(enable2);
```

```
4        <stmt2>
5    }
```

Assume that the process is invoked at the i-th cycle, then <stmt1> will be executed in that cycle. Then it encounters the wait statement at line 3 where it waits for enable2 (it can be any expression which evaluates to 0 or 1) to become true. Then <stmt2> is executed (at the same cycle when enable2 turns true). <stmt2> will be executed at no earlier than the $i+1$-st cycle. That is, if enable2 turns true at the i-th cycle, foo will not be able to sense it (and execute <stmt2>). If input in is referenced in <stmt2>, its value at the i-th cycle (when process foo is invoked) will be used.

2.4.2 Flow Control in Processes

if/else statements provide simple flow control in processes. Consider the following example:

```
<stmt1>;
if (cond) {
    <stmt2>;
    wait(trigger);
    <stmt3>;
} else {
    <stmt4>;
}
<stmt5>;
```

Assume that cond is true, <stmt1> and <stmt2> will get executed in the same cycle. Then <stmt3> (and <stmt5>) will get executed at least one cycle later (when trigger is asserted). If cond is false at the beginning, then <stmt1>, <stmt4>, and <stmt5> will get executed in the same cycle.

while statements provide loops in V++. A while statement introduces zero delay. That is, on entry, it evaluates the conditional expression. If the result is true, the first statement inside the while statement is executed. Otherwise the first statement that follows the while statement is executed. When execution flow reaches the end of a while statement, it restarts the while immediately (with zero delay). Consider the following example:

```
1   process foo(bit in) @(clk) {
2       <stmt1>;
3       while (cond) {
4           wait(trigger);
4           if (c)
5               <stmt2>;
6           else
7               break;
8       }
9       <stmt3>;
10  }
```

Assume that the process is invoked at the i-th cycle, <stmt1> is immediately executed in that cycle. Assume that cond is true, the while loop will be entered and the process waits for trigger to come true. Whenever trigger turns true, the if/else statement (line 4-7) will get executed. The break statement at line 7, when executed, aborts the while loop and resumes execution at the statement (<stmt3>) that follows the aborted while statement.

If cond is false at the i-th cycle, then only <stmt1> and <stmt3> will be executed in that cycle.

There are two ways to break out of a while loop. First, if the conditional expression of the while statement evaluates to false when it is executed, the loop is terminated. Second, if break statement is encountered in a while statement, the loop is immediately terminated and statement that follows the while statement gets executed (in the same cycle).

Another way to alter the flow of execution in loops is to used the continue construct. It re-executes the loop construct immediately when it is executed. However, if the conditional expression of the while loop construct does not hold, then the loop will terminate immediately.

Another form of loop statements is the whenever loop:

```
whenever(cond_expression) <body_statement>
```

```
1   while (cond) {
2       a = a+1;
3   }
```

(a)

```
1   while (cond) {
2       if (foo)
3           a = a+1;
4       else
5           wait(trigger);
6   }
```

(b)

```
1   while (cond) {
2       if (foo) {
3           a = a+1;
4           wait(trigger);
5       } else wait(trigger);
6   }
```

(c)

Figure 2.6: (a) An illegal loop. (b) Another illegal loop. (c) A legal loop.

A `whenever` loop never terminates. When a `whenever` statement is executed, it waits for its conditional expression to turn true. Then it executes the body statement. The above `whenever` statement amounts to the following `while` statement:

```
while (1) {
    wait (cond_expression);
    <body_statement>
}
```

2.4.2.1 Loops and Delays

V++ imposes some restrictions on the allowable combinations of loop constructs to ensure that any statement does not get executed more than once at any instant. The rule is that along any loop in a V++ program, there should be at least one `wait` statement (delay) in that loop.

Figure 2.6 gives a few examples. The code in Figure 2.6.a is illegal as when `cond` is true, line 2 will get executed infinitely many times. The code in Figure 2.6.b is still illegal because line $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is a zero delay loop. The code fragment in Figure 2.6.c is legal as all cycles are cut by at least one `wait` statement.

2.4.3 Creating Execution Threads

V++ allows designers to introduce *execution threads* (a thread is a sequence of statements that can be executed concurrently with other threads) in a process by using the `fork` statements. Basically, a `fork` statement spawns a set of concurrent sequences of statements, separated by `||` (each sequence of statement is also called a *branch* of the `fork` statement), that will be *synchronized* when they finish execution. The set of threads are executed in parallel when the `fork` statement is executed. The `fork` construct can be executed instantaneously. That is, it introduces zero delay on entering or exiting. Consider the code fragment.

```
<stmt1>;
fork {
   <stmt2>; || <stmt3>;
}
<stmt4>;
```

Assume that `<stmt1>` is executed at the i-th cycle, then `<stmt2>` and `<stmt3>` will get executed at the i-th cycle. `<stmt4>` will get executed whenever both `<stmt2>` and `<stmt3>` are finished. That is, assume that `<stmt2>` finishes at the j-th cycle and `<stmt3>` finishes at the k-th cycle, then `<stmt4>` can be executed at the $\max(j,k)$-th cycle. Therefore, if both `<stmt2>` and `<stmt3>` contain no `wait` statement, `<stmt1>`, `<stmt2>`, `<stmt3>`, and `<stmt4>` are all executed in a single (i-th) cycle.

2.4.4 Interactions between Safe Port Invocations and Processes

When a process does a safe port invocation, it does not know if the message can be successfully sampled until the next clock cycle. Assume that the sending process is informed that the safe port invocation cannot go through, then the process should suspend its execution. Suppose, on the contrary, that the suspended safe port invocation does not block the sending process from execution. Consider the example in Figure 2.7. Assume that line 3 is executed at the i-th cycle and `trigger` is asserted at the $i+1$-st cycle. If the safe port invocation does not go through, then the above code segment will try to do two safe port invocations from the $i+1$-st cycle (one starting from the i-th cycle, the other starting from the $i+1$-st cycle). This is unacceptable as V++ is a hardware design language. Therefore, the amount of hardware to implement a piece of code should be decidable at compile/link time.

In V++, when a process executes safe port invocation in one cycle (cycle i), the acknowledgment signal (from the safe port invocation) guards the effectiveness of the `wait` statement that will be active in the next cycle. That is, if the safe port invocation does not

```
1   while (1) {
2       wait(trigger);
3       safe foo();
4   }
```

Figure 2.7: A Fragment of A Process with Safe Port Invocation.

go through in the next clock cycle, then no `wait` statement will be sensitive to their waiting conditions. What this implies is that no statement (other than re-send) will be executed in cycle $i+1$. The process suspends its execution until all the safe port invocations executed in cycle i gets finished. Assume that the last safe port invocation finishes at cycle j (the acknowledgment signal is asserted at cycle j), then the process resumes execution at cycle j.

Consider again the example in Figure 2.7. Assume that `foo` is invoked at the i-th cycle (line 3). The control is, in the mean time, transferred to the `wait` statement at line 2, where it waits for `trigger` to become true starting from (including) the $i+1$-st clock cycle. If the safe `foo()` port invocation does not succeed at the $i+1$-st cycle, then the control flow will stop at line 3, waiting for the safe send to complete, even if `trigger` is true. Suppose that at the j-th cycle ($j > i$) the safe send is done; `wait` statement will then become effective (sensitive to `trigger`) immediately. That is, if `trigger` is asserted at the j-th cycle, then another safe port invocation will be made at the j-th cycle.

Similarly, for `fork` statements, if any branch of a `fork` statement does a safe port invocation, the `fork` statement still spawns threads and synchronizes them as if there were no safe port invocation at all. However, if there are pending safe port invocations, then the `wait` that will be executed in the next clock cycle will not be sensitive to its waiting condition until all safe port invocations are finished. Contrast the following two code segments:

```
fork {
    a = 1; || foo();
}
<stmt1>;
wait(trigger);
```

```
fork {
    a = 1; || safe foo();
}
<stmt1>;
wait(trigger);
```

Note that for both fragments, `<stmt1>` gets executed at the same cycle when `fork` is executed, no matter if the safe port invocation can succeed or not. The wait statement in the code fragment on the left side will be active starting from the next clock cycle. However, on the right hand side, the wait statement will not be active until the safe port invocation is finished.

2.4.5 Exceptions

To preempt the execution of a running process, there is a watch construct that immediately interrupts the execution of processes. For example,

```
{ <stmt> } watch(<cond>) <watch_stmt>;
```

When the watch clause's corresponding statement ({<stmt>}, the statement the watch clause is affixed to) is active, the clause continuously checks if the watching condition (<cond>) is true. If the condition is not met during the execution of the statement (<stmt>), then the statement simply behaves as if the watch clause does not exist. However, if the watching condition holds before statement <stmt> is finished, then an exception handling code (<watch_stmt>) will be invoked and the execution of the statement (<stmt>) will get aborted.

```
{
   <stmt1>
} watch(kill) { <stmt2>; }
<stmt3>;
```

Assume that <stmt1> is executed at the i-th cycle. If signal kill is asserted before <stmt1> is finished, then <stmt2> will be executed (the instant kill is asserted) and control will then be transferred to <stmt3> after <stmt2> is finished. Note that the watch clause becomes *active* (sensitive to the watching condition) whenever its corresponding statement ({<stmt1>}) is being executed.

watch clauses can be used with wait, while, whenever, {} (sequence of statements), and fork statements. When a watch clause is used with fork statements, all active branches will be preempted when the watching condition is met.

Note that if a statement is sending a message (safe or unsafe) while the outer (enclosing) watching condition is met, the message will not be sent. However, when there is a pending safe port invocation, watch clauses will be disabled (not sensitive to their watching conditions).

Similar to watch clauses, loops (while or whenever statements) can also have restart clauses. When the restart condition in a restart clause is true, the loop (that is attached to by the restart clause) gets restarted immediately. For example:

```
1   while
2       restart(again) { rerun = 1; }
3       (cond) {
4       wait(fire); go = 1;
5   }
```

Assume that the code stops at line 4, waiting for fire. Suppose that again is asserted, then the loop will get restarted (without setting go to 1) and rerun is set to 1, no matter fire is asserted or not. Assume that fire is asserted when again is absent, then go is set to 1 (and rerun is not set).

Remember that if a wait statement is executed at the i-th clock cycle, it will not be sensitive to the waiting condition until the $i+1$-st cycle (Section 2.4.1). However, if wait(w_cond) watch(p_cond) <p_stmt> is executed at the i-th cycle and p_cond is asserted in the same cycle, then the wait statement will be aborted immediately and the statement that follows the watch clause is executed (in cycle i). That is, it will not be waiting for w_cond at the $i+1$-st cycle at all.

Associated with each watch/restart clause is a fixed priority (the priority decides which exception handling code will be executed and what statement(s) will be aborted) that can be determined statically according to the syntactic structure of the program. In a process, the outer (syntactically) clause has the higher priority. When multiple watch/restart conditions are met, only the clause with the highest priority will be effective.

2.5 Assertions

Assertions can be put in objects or channels to monitor the behavior of a piece of code. They are statements that do not correspond to any hardware. However, they are properties of a hardware that can be checked by simulators or verifiers to help designers validate the design. They also can be used to generate don't care conditions to guide synthesizers to generate compact implementations. Consider again the `FSM` object shown in Section 2.1.2.2.

```
1 object FSM {
2     latch bit[1:0] state;
3     clock clk;
4     count() @(clk) {
5         if (state < 1) {
6             state = state+1;
7             self.count();
8         } else {
9             state = 0;
10        }
11    }
12
13    assert(state <= 1): stateProperty;
14 }
```

The `assert` statement at line 13 declares an invariance that the designer wishes to be true for the design. `stateProperty` is the label of the assertion. If the property holds, then the synthesizer can use it to optimize the implementation (e.g., only one bit is necessary for variable `state`). Unfortunately, the assertion cannot be satisfied by the above design. The violation can be found either by a formal verifier or a simulator. Once the error is flagged, designers can use the error trace to fix the bug.

2.6 V++ In Action

In this section, V++ is used to design various hardware components. First, a few circuit components like finite state machines, buses are designed using V++. The next example is Unger's traffic light controller. The last example is the reflex game [Hal93]. It demonstrates the power of processes and preemption constructs.

2.6.1 Finite State Machines

In V++, a FSM can be realized by a port which keeps sending itself messages. For example:

```
latch bit [1:0] state;
latch bit [1:0] in, out;
fsm() @ (clk) {
   switch (state) {
      case 0:  {state = 3; out=in;}
      case 1:  {state = in; out=1;}
      case 2:  {state = state; out=in+1;}
      case 3:  {state = 0; out=state;}
   }
   fsm();
}
```

In the above example, `fsm` is a port in the containing object (which is not shown). Note that this is an "active" FSM which does a next state computation on each clock cycle. If we want a "passive" FSM, which does nothing until the input stimulus is available, the following code segment is an example:

```
latch bit [1:0] state;
latch bit [1:0] out;
fsm(bit[3:0] in) @(clk) {
   switch (state) {
      case 0:  {state = 3; out=in;}
      case 1:  {state = in; out=1;}
      case 2:  {state = state; out=in+1;}
      case 3:  {state = 0; out=state;}
   }
}
```

Contrast this with the previous FSM, the trailing port invocation `fsm()` is removed. The FSM will not do any computation unless a message is present. When a new message is available, the `fsm` port computes the new value of `state` according to its `in` argument. On the other hand, the previous FSM samples the value of latch `in` and updates `state` (its current state) and `out` on each clock cycle.

2.6.2 Buses

The following example illustrates how a bus can be designed by using user defined channels. It can be used as a multiple-writer, single-reader channel. The channel uses a round robin scheme to schedule multiple requests. Imagine the set of writers are arranged on a ring. The round robin scheduler in the channel dynamically assigns priority to writers in a circular fashion. Once a writer finishes its request, the channel searches along the ring for the next writer which is asserting request and assigns the requesting writer with the highest priority. At each clock cycle, if there is no one requesting for message send, then the one that is next to the writer with the highest priority will have the highest priority in the next clock cycle. It is known that such a scheduler is fair. That is, assume that all requests can be finished in bounded time, once a writer asserts its request (and keeps asserting request until it is acknowledged), it will eventually get served.

```
object root {
   clock clk;
   RoundRobin ci = new RoundRobin();

   sender(bit[3:0] d) @(clk) {
      receiver(d) via ci;
   }

   receiver(bit[3:0] d) @(clk) {
   }
}

channel RoundRobin (clkS, reqS, ackS, dataS)
         (clkR, validR, consumeR, dataR) {
   input clock[] clkS;
   input bit[] reqS;
```

```
    output bit[] ackS;
    input bit[][3:0] dataS;

    input clock[] clkR;
    output latch bit[] validR;
    input bit[] consumeR;
    output bit[][3:0] dataR;

    latch bit[3:0] id; // the writer with the highest priority
    latch bit[3:0] q; // the 'FIFO'
    wire bit[clkS.left+1:clkS.right][3:0] id1;
    wire bit[clkS.left+1:clkS.right][3:0] id2;
    wire bit[clkS.left+1:clkS.right] found1;
    wire bit[clkS.left+1:clkS.right] found2;
    wire bit[clkS.left+1:clkS.right] tokenTaken1;
    wire bit[clkS.left+1:clkS.right] tokenTaken2;
    wire bit[clkS.left+1:clkS.right] requesting;
    wire bit[3:0] new_id;
    wire bit[3:0] i;
    run () @(clkR[0]) {
      // the two fixed boundary loops scan for sender with the highest
      // priority.
      id1[0] = 0;
      found1[0] = 0;
      tokenTaken1[0] = 0;
      array(i=clkS.right; i<=clkS.left; i=i+1) {
        found1[i+1] = (i==id) ? 1 : found1[i];
        tokenTaken1[i+1] = (found1[i+1] & reqS[i] & !tokenTaken1[i])
          ? 1 : tokenTaken1[i];
        id1[i+1] = (found1[i+1] & reqS[i] & !tokenTaken1[i]) ?
          i : id1[i];
      }
```

```
      id2[0] = id1[clkS.left+1];
      tokenTaken2[0] = tokenTaken1[clkS.left+1];
      found2[0] = 1;
      array(i=clkS.right; i<=clkS.left; i=i+1) {
         found2[i+1] = (i==id) ? 0 : found2[i];
         tokenTaken2[i+1] = (found2[i+1] & reqS[i] & !tokenTaken2[i])
            ? 1 : tokenTaken2[i];
         id2[i+1] = (found2[i+1] & reqS[i] & !tokenTaken2[i]) ?
            i : id2[i];
      }
      // acknowledgment
      if (tokenTaken2[clkS.left+1])
         new_id = id2[clkS.left+1];
      else
         new_id = id+1; // no one is requesting the bus
      id = new_id;
      requesting[0] = 0;
      array(i=clkS.right; i<=clkS.left; i=i+1) {
         // channel can take more inputs if:
         // (1) empty queue slot available, or
         // (2) receiver is digesting tokens.
         if (!validR[0] || consumeR[0]) {
            ackS[i] = (i==new_id && reqS[i]) ? 1 : 0;
         } else {
            ackS[i] = 0;
         }
         requesting[i+1] = (i==new_id && reqS[i]) ?
            1 : requesting[i];
      }
      // inform receiver that token is available -- valid signal
      validR[0] = ((validR[0] & !consumeR[0])|requesting[clkS.right]);
      dataR[0] = q;
```

```
81          q = dataS[new_id];
82      }
83  }
```

At line 3, channel `RoundRobin` is instantiated. At line 6, the port invocation indicates that messages will be delivered via channel instance `ci`. In channel `RoundRobin`, `id` (line 25) is the identifier of the channel that has the highest priority, and variable `q` (line 26) latches the data to be delivered. The first two `array` statements (lines 42-57) search for the writer that is requesting to send and has the highest priority. Lines 60-64 determine who will be the one with the highest priority in the next clock cycle. Lines 66-77 then signal the writer(s) if their requests have been granted. It also tells the reader if there are valid data available for consumption (lines 79-81).

Note that line 43 says that if `i` is equal to `id`, then `found1[i+1]` is equal to 1. Otherwise it is equal to `found1[i]`. At first glance, the statement seems to violate the single assignment rule (Section 2.1.2.1) as `found1` is assigned multiple times in a constructor loop. Actually, the assignment still follows the single assignment rule since assignments from different iterations touch different bits of `found1`.

2.6.3 Traffic Light Controller

The code appears below is Unger's traffic light controller. It is a mixture of active and passive FSM shown in Section 2.6.1. The traffic light controller fits the following scenario. There is a highway crossed by a farm road, and the intersection is governed by a traffic light. In general, the light permits traffic on the highway to pass, but when a car arrives at the farm road the light switches to permit traffic to pass along the farm road.

```
define(LONG_TIMEOUT,12)
define(SHORT_TIMEOUT,1)
define(MEDIUM_TIMEOUT,4)

define(GREEN,0)
define(YELLOW,1)
define(RED,2)
```

```
object TrafficLightController () {
   private tick() @(clk) depth(1) destructive {
      switch (count) {
         case SHORT_TIMEOUT + MEDIUM_TIMEOUT:
            { farmRoadLight[0] = farmRoadLight[1] = YELLOW; }
         case (2*SHORT_TIMEOUT) + MEDIUM_TIMEOUT: {
            farmRoadLight[0] = farmRoadLight[1] = RED;
            highwayLight[0] = highwayLight[1] = GREEN;
         }
         case SHORT_TIMEOUT: {
            highwayLight[0] = highwayLight[1] = RED;
            farmRoadLight[0] = farmRoadLight[1] = GREEN;
         }
      }
      if (count != LONG_TIMEOUT) {
         count++;
         self.tick();
      } else {
         count = 0;
      }
   }

   public senseCar() @(clk) when(count == 0) depth(1) destructive {
      highwayLight[0] = highwayLight[1] = YELLOW;
      self.tick();
   }
   private latch bit[0:31] count=0;
   public bit[0:1][1:0] farmRoadLight, highwayLight;
   public clock clk;
}
```

It is apparent to any reader that V++ is significantly different from other HDLs.

The most significant variant from standard HDL's is that there is no explicit connectivity through arguments and wires. Connectivity is simply given by the port declarations. This implies that the ports are bound to wires in the conventional parameter-like sense of most HDL's: this is because connectivity is largely, though not exclusively, the province of the V++ compiler.

Note also that the ports themselves have significant structures. This is a concept borrowed from object oriented software languages, where messages and methods are identical. Similarly, in V++, *ports* and *messages* are indistinguishable: a *port* defines both a message that may be received *and* the logic that processes the message. Thus, the port `tick` (line 10) both defined a message that may be received, and the logic that is executed when the message is received.

When we move to the declaration of `tick`, we note it is declared `private`. This of course means that it is invisible to the outside world, and only receives messages sent from the object itself.

Next, we come to the `@` clause immediately following the code for the port `tick`. This clause indicates the conditions that must apply before the message is received. Therefore, `tick` messages are received whenever `clk` is true (and, of course, `tick` messages are pending when `clk` is false); `senseCar` messages are received whenever `clk` is turned on and `count` is equal to 0.

The immediate question is, what happens to `senseCar` messages that come in when `count` is not 0? The answer is that the V++ compiler/linker provides storage to hold them until the *guard* condition: the argument of the `when` clause (line 31) – is true.

The `depth(1) destructive` modifiers of the `tick` and `senseCar` regulate this storage. The `depth(1)` modifier indicates that the port is only one-deep: i.e., there is no queue attached to this port, merely a simple latch. Only one message may be stored waiting for receipt. The `destructive` modifier indicates that the new messages overwrite the old one.

Taken together, these modifier indicate that the port is essentially a place (capable of holding only one token) in a Petri-net; i.e., an incoming message is essentially an enable, which is disabled once serviced.

Reading chronologically, the next curious construct we observe is `self.tick()` (line 25). This is what it appears to be: a message to *self* on the private port `tick`. This is in effect an instruction to self to continue the counting process at the next cycle. It is required since the semantics of V++ are fundamentally reactive in nature: an object responds to a message by executing the logic on the relevant port for a single cycle, and then is quiescent until the next message arrives.

With this in hand, we can see how the design works. Until a `senseCar` message arrives, nothing happens. Once `senseCar` message arrives, the `hightwayLights` are changed to `YELLOW` and the `tick` port is invoked by a message from `senseCar`. On the next and succeeding cycles, `tick` checks `count`: if it is less than `LONG_TIMEOUT` (on the first cycle it will be 1), it increments `count` and sends itself a message to continue; otherwise, it resets `count` to 0; the object then awaits next message to `senseCar`.

As can be seen from the example, the message-sending architecture has the side effect that control of a part designed in V++ tends to be distributed. *Note, however, that this architecture is not defined by a simulation algorithm. Rather, the guard code, the message delivery and storage structure, and all the code internal to a V++ object, is directly mapped into gates and latches and the mapping is part of the semantics of V++.*

2.6.4 The Reflex Game

In this section, V++'s process constructs are used to design a reflex game machine [Hal93].

2.6.4.1 Rules of The Game

The reflex game machine consists of two buttons (`ready` and `stop`) and one coin slot (when the player inserts a coin, the variable `coin` is set to 1). It has a numerical display (a `display()` port, whose definition is not shown in the code below) used to display how fast a player can react, a `warning` lamp used to signal that the player cheats during a game or abandons the game, a `game_over` lamp to signal the end of game, and a `go` lamp used to indicate that game machine has started counting. The game machine also has a bell which tells gamers that the wrong button has be pressed.

Initially, the `go` and `warning` lights are off, the numerical display is reset, and the `game_over` light is turned on. After accepting a coin, the machine enters the following sequence:

1. Turn off the `game_over` lamp.

2. Wait for the user to press `ready` button.
3. Once the `ready` button is pressed, wait for random number of cycles.
4. Turn on `go` lamp and start a counter measuring how fast the player can react.
5. Wait for the player to press `stop`.
6. Record/display the player's response time.

The above sequence is repeated `NUMBER` times and then the average response time is displayed and the game is terminated (the machine goes back to the initial state, waiting for more coins). However, there are a few exceptions that can affect the normal game flow shown above:

- At step 2, instead of the `ready` button, the player presses `stop`. In this case, a bell is rung signaling that the wrong button has been pressed. (exception I)
- At step 5, instead of the `stop` button, `ready` is pressed. Again, the bell is rung. (exception II)
- At step 3, the `stop` button is pressed. The `warning` light is turned on to warn the player not to cheat. (exception III)

2.6.4.2 The Program

```
object reflex {
    public wire bit go; // 'go' light
    public wire bit warning; // 'warning' light
    public wire bit ringBell; // make some noise
    public wire bit gameOver; // 'game_over' lamp
    public wire bit coin; // the gamer inserts a coin
    public wire bit ready; // ready button is pressed
    public wire bit stop; // stop button is pressed
    private wire bit letGo;
    latch bit[3:0] totalTime;
    latch bit[3:0] measureNumber;
```

```
    latch bit[3:0] averageTime;
    latch bit[3:0] time;
    clock clk;
    reflex() { time.init = 0; timer(); }
    updateAverage(bit[3:0] time) @(clk) {
       totalTime = totalTime + time;
       measureNumber = measureNumber + 1;
       averageTime = totalTime / measureNumber;
    }

    timer() @(clk) { // the process is only invoked in the constructor.
       time = time + 1;
       timer();
    }

    signalGo (bit[3:0] pausePeriod) @(clk) {
       if (pausePeriod == 0) letGo = 1;
       else signalGo(pausePeriod-1);
    }

    wire bit[3:0] i;
    process reflexGame() @(clk) {
       totalTime = 0; measureNumber = 0; averageTime = 0;
       go = 0; warning = 0; gameOver = 1; display(0);
       whenever
          restart (coin) { }
             (coin) {
             go = 0; warning = 0; gameOver = 1; display(0);
             totalTime = 0; measureNumber = 0;
             array (i=0; i<NUMBER; i=i+1) {
                // wait for the gamer to signal ready
                whenever
```

```
44              watch (ready) { }
45              (stop) { ringBell = 1; } // exception I
46            signalGo(random());
47            // sleep here and wait for letGo to turn true
48            whenever
49              watch (letGo) { go = 1; }
50              restart (stop) { warning = 1; } // exception III
51              (ready) { ringBell = 1; }
52            // now see how fast can the gamer react
53            time = 0;
54            whenever
55              watch (stop) {
56                display(time); updateAverage(time); go = 0;
57              }
58              (ready) { ringBell = 1; } // exception II
59          }
60          display(averageTime);
61        }
62    }
63    priority reflexGame > updateAverage;
64    priority reflexGame > timer;
65    priority reflexGame > signalGo;
66  }
```

The constructor at line 15 starts the `timer` port, which is a counter. The main body of the reflex game machine starts at line 33. It first initializes various counter variables (`totalTime`, `measureNumber`, and `averageTime`) and turns on/off appropriate status lights. The machine then waits for a gamer to insert a coin. When a coin is inserted, the game machine first waits for the gamer to press the `ready` button (the `ready` variable in object `reflex`). Once the `ready` button is pressed (line 44), the `whenever` statement at line 43 is preempted and the game machine then invokes (line line 46) a timer (`signalGo` port at line 27) which counts up to a random upper-bound. `random()` (line 46) is a function which returns positive random numbers. The definition of this function is not shown in the above example. Port `signalGo` “wakes” `reflexGame` up by setting signal `letGo`, which preempts the `whenever` statement at line 48. In addition, when the `whenever` statement (line 48) is aborted, the `go` light is lit. Now the game machine waits the gamer to push the `stop` button (line 55). Then it shows how long it takes the gamer to respond to the `go` signal (line 56). At the end of the game, the machine shows the average response time (line 60).

At line 37, the restart clause states that, if a coin is inserted in the middle of a game, the current game will be aborted and a new game will be started.

Chapter 3

Compilation, Synthesis, and Simulation of V++

In this chapter, we address the problem of compiling and linking V++ programs. The overall compilation/link process is illustrated in Figure 3.1. The input program is first parsed and transformed into a set of graphs, namely the parse trees, the precedence graphs (Section 3.2.5), and the NCFGs (*nested control flow graphs*, Section 3.2.2.1). Based on the precedence graphs, the compiler determines the resolution function for each variable. Based on the NCFGs, the circuitry for the V++ program is generated. The result of the compilation process is a set of intermediate representations which are then taken by the linker. The intermediate representations contain circuits that have been generated (e.g., the data-paths whose widths are known), the parameterized logic whose parameter cannot be determined yet, and the "holes" that need to be patched by the linker (e.g., references to ports of another object). The linker then builds a instantiation graph according to the instantiation hierarchy in the source program and evaluates the constructors and generators to determine the values of parameters for objects as well as the interconnections among objects. The V++ linker also patches the definition holes for object references. Note that, unlike programming language linkers, the V++ linker is also responsible for generating logic. To be more specific, the V++ linker has to generate logic in channels. The result of the whole compilation process is a network of FSMs or Java code (which can be used for simulation).

This chapter is organized as follows. In Section 3.1, a simple example (Fibonacci

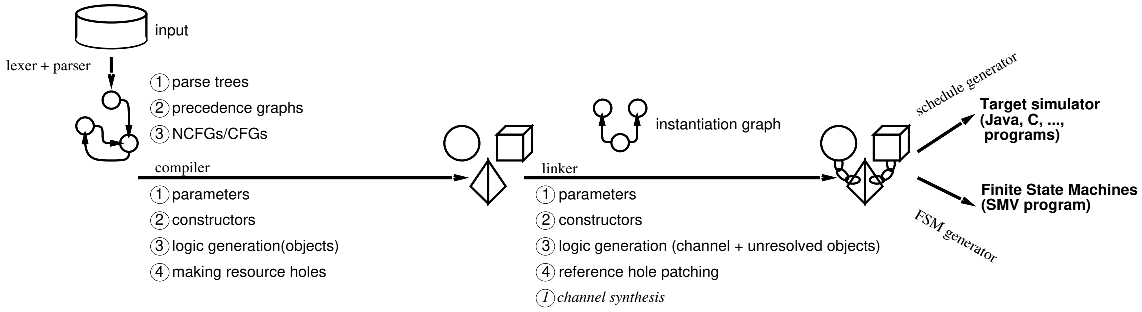


Figure 3.1: V++ compilation flow.

sequence generator) is given to demonstrate what circuits are generated for different V++ constructs. In Section 3.2, the process and algorithms for compilation are presented. Section 3.3 describes what tasks are required in linking together compiled objects. In Section 3.4, a few optimization techniques for synthesizing circuits from V++ programs are given. Scheduling V++ objects for simulation is detailed in Section 3.6.

3.1 Compiling V++ Programs

Conceptually, the V++ compiler generates circuits with the structure shown in Figure 3.2(a). For the body of each port or process, a piece of logic is created. If a port touches a variable, a resolution circuit will be generated to determine if the port can change the value of the variable. Similarly, if a port invokes another port, a communication device (channel) will be synthesized to arbitrate among multiple port invocations to the same destination. The channel is also responsible for sampling and delivering the arguments to the invoked port. For each port/process, there is exactly one channel that can deliver data to it.

Ports and channels communicate with each other via a set of control and data wires. The data wires carry the arguments the invoking process is trying to send to the invoked process. The control wires, namely *request*, *acknowledgment* (it is also referred to as *ack*), *valid*, and *consume*, are used to synchronize channels and ports. These control wires are shown in Figure 3.2(b). The logic in a port/process continuously monitors the channel logic that is responsible for delivering data for the port/process by checking the *valid* bit. If the *valid* bit of the channel is set, the port/process evaluates the logic of the port/process body. As soon as the statement body of the invoked port finishes execution,

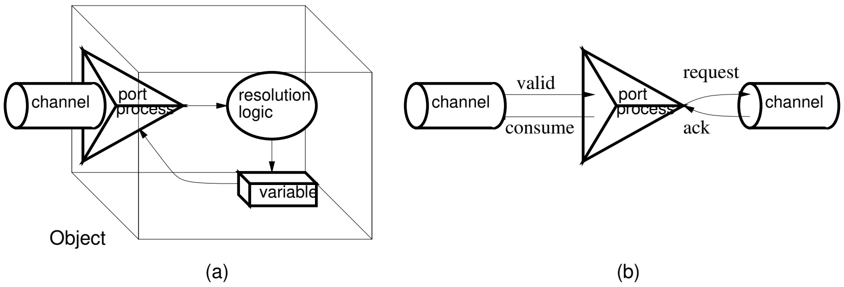


Figure 3.2: The structure of generated circuits for V++ programs

the port/process then asserts *consume* signal to notify the channel that the arguments have been digested. If there are FIFO's in the channel (a FIFO is always allocated in the compiler generated channel) for latching arguments, *consume* can be used to advance the FIFO pointer.

If a port invokes another port, then the *request* bit to the delivering channel will be asserted and the arguments provided to the channel. If the channel has enough capacity to deal with the request, it latches the arguments and asserts *acknowledgment* at the next clock cycle. Otherwise the acknowledgment line will be kept low. If the invoking port/process is doing safe port invocation, it needs to check this signal to determine if it can do more computation, or it has to suspend itself and keep trying the safe send.

The resolution logic (Figure 3.2.(a)) then takes the contending assignments from different ports/processes and determines the values of object variables. If the variable is a latch, the new value will be effective in the next clock cycle. If the variable is a wire, then the effect will be visible immediately.

The following examples highlight the process about how circuits are synthesized by the V++ compiler/linker. Assume that `fn`, `fn_1`, and `fn_2` are all 4-bit vectors. Given the following V++ segment,

```
fn = fn_1 + fn_2,
```

then the most straightforward translation is an adder with `fn_1` and `fn_2` as its inputs and `fn` as its output (Figure 3.3.(a)). Assume that the assignment is enclosed in a port `f`:

```
wire bit[3:0] fn;
f (bit[3:0] fn_1, bit[3:0] fn_2) @(clk) depth(1) { fn = fn_1 + fn_2; }
```

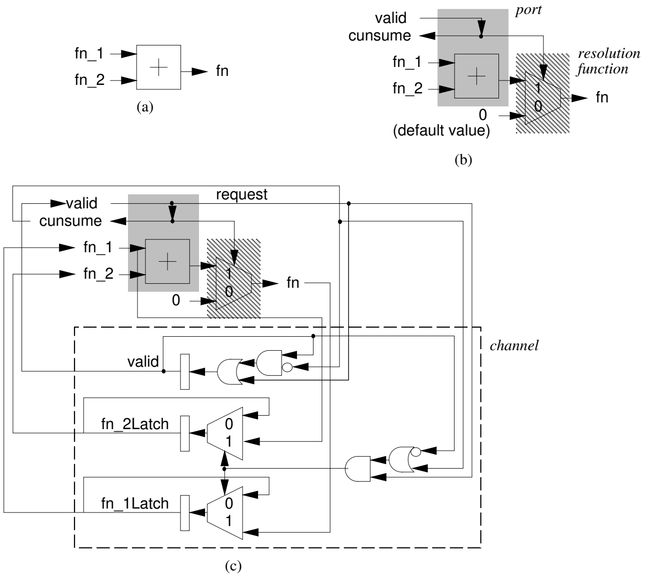


Figure 3.3: A Fibonacci sequence generator – Assume that `fn_1Latch` and `fn_2Latch` are both initialized to 1. Each clock cycle `fn` will give a new Fibonacci number.

The circuit for the above code is shown in Figure 3.3.(b). There are two compiler generated wires allocated for port `f`. The *valid* bit is set by the channel circuit to indicate that there is a fresh copy of `fn_1` and `fn_2` for `f`. This signal is also used to determine if the output of the adder will be used as the new value of `fn`. `f` uses *consume* to indicate if `fn_1` and `fn_2` have been used.

Assume that, in the port body statement, what follows the assignment is a port invocation of `f`:

```
f (bit[3:0] fn_1, bit[3:0] fn_2) @(clk) depth(1 ) { fn = fn_1 + fn_2; f(fn, fn_1); }
```

The channel circuitry as well as the port logic that will be created by the V++ compiler/linker are shown in Figure 3.3.(c). In the channel, two sets of latches (`fn_1Latch` and `fn_2Latch`) are generated for the arguments `fn_1` and `fn_2` of `f`. The latches are enabled under the conditions (1) `f` is trying to send a message (`request` is asserted), and (2) current content of the argument latches will be invalid ("stale") in the next clock cycle (*consume* is asserted or *valid* is not asserted). Note that as the port invocation is not a safe port invocation, no *ack* signal will be generated. The `valid` bit in the channel indicates if the contents of `fn_1` and `fn_2` are valid. It is set when (1) `f` is trying to send a message through the message (by asserting `request` line), or (2) the current value of `valid` latch is one and `f` has not consumed its arguments yet (`consume` is held low).

The above port can be modified to form a Fibonacci number generator:

```
1   object fibonacci {
2       public wire bit[3:0] fn;
3       clock clk;
4       fibonacci() { f(1,1); }
5       f(bit[3:0] fn_1, bit[3:0] fn_2) @(clk) depth(1) {
6           fn = fn_1 + fn_2;
7           f(fn, fn_1);
8       }
9   }
```

The constructor at line 4 does a initial port invocation to `f`. This initial invocation instructs the V++ linker to set the `valid` latch in the channel to 1. In addition, `fn_1Latch` and `fn_2Latch` in the channel are both initialized to 1.

3.2 Compiling V++ Ports and Processes

Now that we have a global picture of the compilation/linkage process, we focus on the details of the mechanisms and algorithms of the V++ compiler/linker. In this section, we work on the compiler.

Section 3.2.1 outlines the templates of I/O pins of the generated circuits for V++ ports/processes. Sections 3.2.2, 3.2.3, and 3.2.4 deal with the compilation of ports/processes. In Section 3.2.2, a set of *kernel process constructs* (or *kernel constructs* for simplicity) is given. The set forms the building block for more complicated (conceptually) V++ constructs. To represent the program structure of V++ programs in terms of these kernel constructs, a graphical representation (*NCFG*, Section 3.2.2.1) is then introduced. The complete compilation scheme for all the V++ control constructs is then detailed in Sections 3.2.3 and 3.2.4. In Section 3.2.3 we show what circuitry are generated for each kernel construct. We then show, in Section 3.2.4, how V++ constructs are compiled into kernel process constructs. Section 3.2.5 details resolution logic generation and hardware generation for variables.

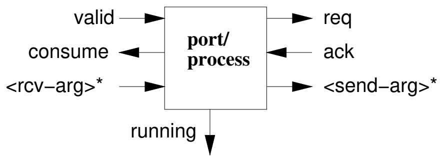


Figure 3.4: I/O Pins for Port/Process Logic

3.2.1 "Pins" For Ports/Processes

Given a port/process, a chunk of logic will be generated by the compiler (Sections 3.2.3 and 3.2.4). Conceptually, this chunk of logic has the I/O pins shown in Figure 3.4.

The `valid` signal comes from the channel that carries the arguments to the port. The `consume` bit, computed by the port/process, indicates if the argument has been taken by the port/process so that the channel can discard it. `<rcv-arg>*` is a list of data wires that are used as the arguments for the port/process. Whenever the port/process makes a port invocation, a set of `(req,ack,<send-arg>*)` signals are allocated. These signals will be glued to the channel that is responsible of carrying the arguments to the invoked port/process. `req` is asserted by the sending port/process indicating that it intends to invoke a certain port. `<send-arg>*` are the data to be delivered. `ack`, computed by the delivering channel, indicates if the channel has a chance to sample `<send-arg>*`. If the invoking port is doing a safe port invocation, the signal is used to block its execution.

The output pin `running`, computed by the port/process, indicates if the port/process is performing computation/port-invocation. If a port/process is making a safe port invocation and it is blocked, then this pin is set to low.

3.2.2 Kernel V++ Port/Process Constructs

To compile a V++ port/process, the V++ compiler needs to analyze the interaction among different constructs. For example, given the V++ fragment "{ `<statement>` } `watch (c)`", the generated circuit should be able to instantaneously abort the execution of `<statement>` whenever `c` is asserted. However, when `c` is asserted, `<statement>` might be executing its sub-statements. Therefore, `watch(c)` not only needs to abort the execution of sub-statements of `<statement>`, it also needs to change the execution flow so that the statement following `watch(c)` will get executed. That is, when the V++ compiler is compiling `<statement>` and its sub-statements, it should be aware that they can be preempted

<table>
<tr><th>Construct</th><th>Synopsis</th></tr>
<tr><td colspan="2">Control Flow Constructs</td></tr>
<tr><td>@(<expr>)</td><td>wait for <expr> to turn true</td></tr>
<tr><td><label>:</td><td>put a symbolic label for branch</td></tr>
<tr><td>branch(<cond_expr>) <label></td><td>conditionally transfer control to <label>, depending on the outcome of <cond_expr></td></tr>
<tr><td>{ <stmt1> ; ... ; <stmtn> }</td><td>serial composition</td></tr>
<tr><td>fork
  <stmt1> || ... || <stmtn>
join (<preempt_expr>)</td><td>parallel composition with weak preemption –
(1) Create concurrent execution threads.
(2) Control is transferred to the statement following the fork/join statement when preempt_expr is asserted. Execution of branches of fork/join is untouched when preempt_expr is asserted.</td></tr>
<tr><td colspan="2">Data Flow Constructs</td></tr>
<tr><td><lhs> = <rhs></td><td>assignment</td></tr>
<tr><td><id> (<expr_list>)</td><td>default port invocation, unsafe</td></tr>
<tr><td>safe <id> (<expr_list>)</td><td>safe port invocation</td></tr>
</table>

Table 3.1: Kernel V++ Port/Process Constructs

by signal c.

The V++ Hardware Design Language has a rich set of constructs. Analyzing the interaction among all V++ constructs will be complicated, tedious, and error-prone. To simplify the compilation process, a small set of *kernel constructs* is introduced. The kernel constructs are carefully chosen so that the V++ constructs can be described as combinations of the *kernel constructs*. Note that not all these kernel constructs are directly available in the V++ language. These kernel constructs are given in Table 3.1.

In addition to the constructs listed in Table 3.1, the kernel constructs provide all the operators available in the V++ language. Therefore, the translation of V++ expressions into kernels is simply making duplicates. The kernel constructs can be classified into two groups: the control flow constructs and the data flow constructs. The control flow constructs (wait, label, branch, serial composition, and parallel composition) are used to change the execution flow of V++ ports/processes. On the other hand, data flow constructs (assignment, default port invocation, and safe port invocation) are used to modify the values of variables or send messages to other ports/processes.

The wait construct (@(<expr>)) halts the execution of a port/process until <expr> becomes true. A wait statement takes at least one clock cycle to execute. That is, the time

@(<expr>) is executed (at cycle i), it is not sensitive to <expr>. It can be sensitive to <expr> only in the cycles that follow cycle i. Wait statement is the only kernel construct that explicitly introduces delays.

The label construct does nothing but put a symbolic label so that a branch construct can "jump to". The branch(<expr>) <label> chooses either the statement that immediately follows it (when <expr> is false) or the statement that immediately follows <label>: (when <expr> is true) for execution, depending on the outcome of <expr>.

The two constructs {} and fork/join are called *composite statements* as they group a set of statements (separated by ; and ||, respectively). The curly braces group a set of statements "sequentially" (separated by semi-colons). That is, suppose that

```
{<stmt1>; @(<expr1>); <stmt2> }
```

is selected for execution at the i-th cycle, then <stmt1> is executed at the i-th cycle, then it waits for <expr1> to become true from the $i + 1$-st cycle. When <expr1> turns true, <stmt2> can be executed at the very same cycle.

fork/join groups a set of statement "concurrently" (separated by ||). The statements that are separated by || are called the *branches* of the corresponding fork/join statement. A fork/join statement spawns a set of concurrent, independent branches which will be *synchronized* when they are finished. The synchronization takes zero time. That is, as soon the slowest branch of the fork/join statement finishes, the fork/join statement finishes (at the same instant).

Note that the kernel fork/join construct is different from the fork statement of the V++ language (Chapter 2) in that a kernel fork/join statement has an additional *weak preempting expression* following the fork/join keyword. A fork/join statement like fork <stmt1> || ... || <stmtn> join (<preempt_expr>) can be "weakly" preempted when the expression <preempt_expr> is asserted. The function of the weak preempting expression is to abort the fork/join statement even if if it is still waiting for some of its branches to finish. The statement following the fork/join statement can then immediately start execution. However, the execution of the unfinished branches is not affected by the weak preempting expression. Later, when these unfinished branches finish execution, the fork/join statement will simply ignores their completion.

Note the difference between fork/join and {}. Suppose that all the statements grouped by the two composite constructs contain no wait statement, then the two statements behave the same. However, suppose that the statements contain wait statement(s), then different branches in the `fork/join` statement will get executed independently. That is, suppose that one branch is locked in one of its wait statement(s), execution in other branches will not be affected. However, in a block of sequential composite statement, if the statement that precedes (in syntactic order) is locked in its wait statement(s), then the statements that follow will be excluded from execution (at current clock cycle).

The assignment `<lhs> = <rhs>` changes the value of a variable. When an assignment is executed, a new value (result of `<rhs>`) is suggested as the new value of the `<lhs>` variable. The actual new value of the variable will be determined by a compiler generated resolution logic.

The port invocation (`<id> ( <expr_list> )`), when executed, sends a message to port `<id>` via an appropriate channel. If the channel does not have enough capacity to deliver the message at the very same cycle, then the message will get lost. The safe port invocation (`safe <id> ( <expr_list> )`) has the same functionality except that if the message send does not go through, the execution of the sending party will be suspended until the channel has a chance to latch the data (`<expr_list>`) to be sent.

The label, branch, sequential composition, parallel composition, and wait constructs define the *control structures* of ports/processes. The assignment, port invocation, and safe port invocation constructs define the *data-paths* of ports/processes. Based on the values of conditional expressions in a port/process, the control structure of the port/process picks up the data-paths that can be executed. Intuitively, one can imagine a port/process operates in the following fashion. At the beginning of each clock cycle, a process is sleeping at one (or more) of its wait statements (it will be called the *source wait statement*), waiting for specific expressions to become true. When the expressions turn true, the port/process searches, along the control structure of the port/process, for a set of wait statements (referred to as the *destination wait statements*) where the control of the port/process can stay and wait for new conditions in the next clock cycle. All the data-paths on the paths, selected by conditional expressions, between the source wait statement and the destination wait statement are executed simultaneously.

3.2.2.1 Representation of V++ Ports/Processes

To represent the abstract control structure of V++ ports/processes, *Nested Control Flow Graphs* (NCFGs) are introduced in this section. Similar to *control flow graphs*

[ASU85] in traditional programming language compilers, NCFGs make the transfer of control explicit. Unlike software control flow graphs, in NCFGs, all statements in serial (without intervening wait statement) are executed simultaneously. The syntactic order among these statements does not suggest any data dependency (whether it is true data dependency, anti-dependency, or output dependency [HP90]). Dependencies are introduced only by variable references. That is, read from wire variables always refer to the values that have been assigned in the present cycle. Read from latch variables always refer to their present states while all writes to latch variables can only affect their next states (the new values of the latches in the next clock cycle).

Definition 3.2.1 (NCFG/CFG) *A* nested control flow graph *(NCFG) is a multi-graph $G = (V, E)$ where V is the set of nodes and E is the set of arcs. Associated with vertex $v \in V$ is its* kind, *denoted by $\kappa(v), \kappa : V \rightarrow \{$wait, dummy, branch, serial, parallel, assignment, send, safe-send $\}$ for construct wait, label, conditional branch, serial composition, parallel composition, assignment, default port invocation, and safe port invocation, respectively. We will also exchangeably use $v \in V$ and the corresponding construct/statement in the source V++ ports/processes if it does not cause confusion.*

To construct a NCFG for a V++ port/process, each construct is allocated a distinct vertex. In addition, there is a directed arc $e \in E, e : v_1 \rightarrow v_2$ iff

1. *v_1 and v_2 are separated by ; and v_1 precedes v_2 syntactically, or*

2. *v_1 is a conditional branch with v_2 as its target label.*

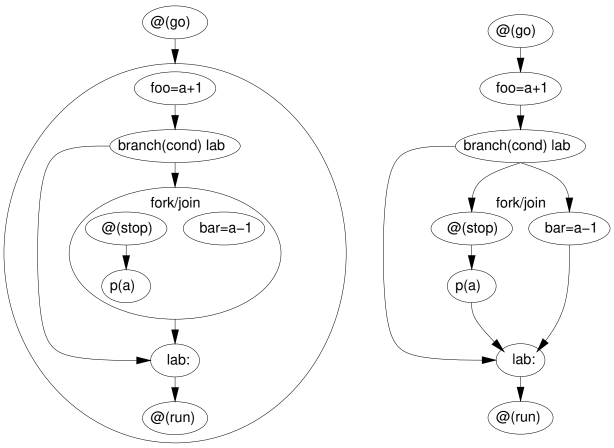
Note that if $\kappa(v_1) ==$ branch, a label is associated with e, denoted by $l(e)$, which is defined as:

$$l(e) = \begin{cases} f & \text{if } e \text{ is introduced by rule 1 shown above} \\ t & \text{if } e \text{ is introduced by rule 2 shown above} \end{cases}$$

If $\kappa(v) \in \{$ serial, parallel $\}$, then it is called a composite vertex. *If $v \in V$ is a sequential composition, then define $\gamma(v) = \{G'\}$ where G' is the NCFG for its sub-statements (the statements that are enclosed by $\{\ \}$). Similarly, if $v \in V$ is a parallel composition, then $\gamma(v)$ is the set $\{G' | G'$ is the NCFG of s, s is a branch of $v\}$.*

The Control Flow Graph *of a port/process can be derived from its NCFG by recursively expanding its composite (serial or parallel) vertices in the following way until there are no composite vertices.*

```
@(go);
{
   foo = a + 1;
   branch(cond) lab;
   fork
      @(stop); p(a)
      ||
      bar = a - 1
   join;
lab:;
   @(run)
}
```



(a) Kernel Code (b) NCFG (c) CFG

Figure 3.5: Kernel Code Segment, Its NCFG, and The CFG

1. *If vertex v is a serial composite vertex. Assume that α is the first vertex in $\gamma(v)$ and β is the last vertex in $\gamma(v)$, replace each edge $u \to v$ with $u \to \alpha$ and $v \to u$ with $\beta \to u$.*

2. *If vertex v is a parallel composite vertex. For each branch NCFG $G' \in \gamma(v)$, assuming that α is the first vertex in G' and β the last vertex in G', replace each edge $u \to v$ with $u \to \alpha$ and $v \to u$ with $\beta \to u$.*

Figure 3.5 illustrates a sample kernel code segment (Figure 3.5.a), its NCFG (Figure 3.5.b), and the CFG. Note that the `fork/join` vertex in the NCFG contains two sub-NCFGs, one for each of its branches.

3.2.3 Compiling V++ Port/Process Constructs

To compile a V++ port/process, the port/process constructs are first transformed into kernel constructs. Then the NCFG for the transformed code segment is built, and circuits are generated from the NCFGs. In this section, we show how to build circuits from kernel constructs. In Section 3.2.4, the details of compiling V++ constructs into kernels are given.

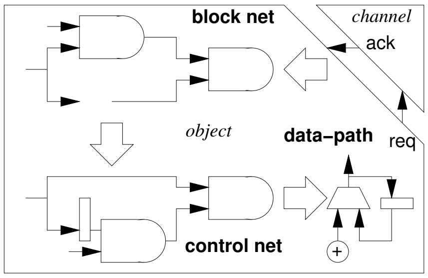


Figure 3.6: Interaction Among Control Net, Block Net, And Data-Path

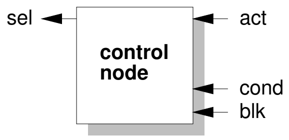
3.2.3.1 Building Circuits From NCFGs

Given an NCFG of a kernel program, the synthesized circuit contains three major parts – the *control net*, the *block net*, and the *data path*. Their interaction is shown in Figure 3.6. Both control and block nets are structurally translated from the NCFG. The data path is directly translated from the data flow constructs in the port/process. The control net is used to determine which data-path operators will be active and which port will be invoked. The block net checks if there is any pending safe send executed before the destination wait statement is executed (by looking at the `ack` signals returned from the channels that are responsible for delivering outgoing safe messages). If that is the case, then the block net will prevent the destination statement from being sensitive to the waiting condition until all the safe sends get finished.

In each clock cycle, the block net first computes, from the `ack` signals from safe send channels, if the destination wait statement can be activated (sensitive to its conditional expression). If there are still unfinished safe sends, then the wait statement will not be active. Otherwise, it will be waiting for the conditional expression to turn true. When the conditional expression of an active wait statement turns true, the control net will choose, according to the syntactic structure of the program, the set of data-path components that will be executed (by changing the values of variables or asserting `request` signal for sending messages). In the meantime, the control net also finds the set of destination wait statements at which the port/process will wait for new conditions in the following cycles.

Figure 3.7 shows the algorithm for building circuits from a NCFG. Before we go into the details of how the algorithm works, we first examine the routines that are used by the algorithm. Conceptually, compilation of expressions (generation of data-paths) is the most straightforward task. Basically, all we need to do is to build the dataflow graph according to the parse tree of an expression. For each expression, its sub-expressions are first compiled into circuits. The outputs from these circuits are treated as the results of sub-expression compilation. The operator of the expression is then built with the results from the sub-expression compilation as the inputs. The output of the circuit for the operator is then treated as the output/result of the expression compilation. For example, to compile statement s, `foo = a + b`, the sub-expressions `a` and `b` are first compiled, then the outputs (the wires carrying the values of `a` and `b`) are fed into an adder (the result of compiling `+`) and the output of the adder is taken as the result of compiling s, denoted by Data-Path(s).

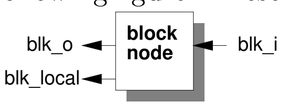
Circuits that are generated by routine `ControlNode(vertex_type)` follow a standard I/O pin definition as shown in the following figure. These circuits are called the *control nodes*.



The functionality of each pin is:

- The output pin `sel` (in case `vertex_type` is `branch`, there are two separate pins, `sel_t` and `sel_f`, instead of `sel`) indicates that the following statement will be chosen for execution in the current clock cycle.
- The input pin `act` indicates that the statement is chosen to be executed.
- The input `cond` carries a boolean value. If the `vertex_type` is `wait`, the pin denotes the condition the statement is waiting for. If `vertex_type` is `branch`, the `cond` pin determines which branch will get executed. If `vertex_type` is `parallel-composite`, the pin determines if the statement should get aborted immediately.
- The input `blk` is used to indicate if the safe sends executed before the statement have been finished.

Circuits that are generated by routine `BlockNode(vertex_type)` follow a standard I/O pin definition as shown in the following figure. These circuits are called the *block nodes*.



The functionality of each pin is:

```
BuildCircuit(G:NCFG) { // G = (V,E)
    // allocate circuit templates
    foreach v ∈ V do
        // allocate circuit node
        dp_v = Data-Path(v);
        cn_v = ControlNode(κ(v));
        bn_v = BlockNode(κ(v));
        // wiring circuits allocated for the same vertex
        glue blk_local pin of bn_v with blk pin of cn_v
        if (κ(v) == branch)
            glue the output of dp_v with cond of cn_v
        // next we deal with composite vertices, recursively
        if (κ(v) ==serial-composite ||κ(v) == parallel-composite)
            foreach G′ ∈ γ(v) do
                BuildCircuit(G′)
            od
    od
    // wiring vertices
    foreach e : v1 → v2 ∈ E do
        if (κ(v1) == branch)
            if (l(e) == t)
                allocate a wire connecting sel_t pin of cn_v1 to act pin of cn_v2
                allocate a wire connecting blk_o_t pin of bn_v1 to blk_i pin of bn_v2
            else if (l(e) == f)
                allocate a wire connecting sel_f pin of cn_v1 to act pin of cn_v2
                allocate a wire connecting blk_o_f pin of bn_v1 to blk_i pin of bn_v2
        else
            allocate a wire connecting sel pin of cn_v1 and act pin of cn_v2
            allocate a wire connecting blk_o pin of bn_v1 and blk_i pin of bn_v2
    od
}
```

Figure 3.7: The Algorithm for Building Control Nets, Block Nets, and Data-Paths.

- The output pin `blk_o` indicates that there is at least one pending safe port invocation to be finished after the current vertex is executed. (If `vertex_type` is `branch`, there are two separate pins, `blk_o_t` and `blk_o_f`, instead of `blk_o`. `blk_o_t` and `blk_o_f` are identical outputs.)

- The input pin `blk_i` indicates that there is one (or more) pending safe port invocation to be finished before the current vertex is executed.

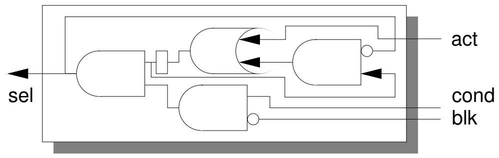
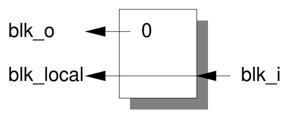
- The output pin `blk_local` is used only if `vertex_type` is `wait`. The pin carries a boolean value. It indicates if the safe sends preceding the wait statement are pending. It is used to control if the wait statement can be sensitive to the waiting condition. A wait statement, even though it is selected as the destination wait statement, cannot be sensitive to the waiting condition if there are pending safe sends executed before the wait statement.

The algorithm in Figure 3.7 can be used to structurally, recursively build control nets, block nets, and data-paths from a NCFG G. Depending on the kind of a vertex in G, it allocates predefined circuit templates in the control net, block net, and data-path for that vertex. The following subsection details the template building routines `ControlNode()`, `BlockNode()`, and `Data-Path()` for each kind of vertex. The allocated circuit nodes are then wired together according to the edges in G. In a control (block) net, if there is more than one wire connected to the input pin of a control (block) node, all the wires are or'ed together. The above process is recursively applied to all the composite vertices in the NCFG.

Next, we show how to synthesize circuits for each type of vertex.

Wait Vertex

Given a NCFG node v for a wait statement `@(<expr>)`, the circuits shown in the following table are allocated for the control net, block net, and data path. In the control node, the small rectangle in the middle is a latch controlled by the clock of the port/process. When the `wait` statement is not busy waiting for `cond`, the latch holds 0. Once the `wait` statement becomes the destination wait statement, the latch is loaded. From that point, it waits `cond` to turn true. Once the waiting condition expression becomes true and `blk` pin is false (i.e., there is no pending safe port invocation), the `sel` output pin is asserted and the latch will be discharged at the next cycle.

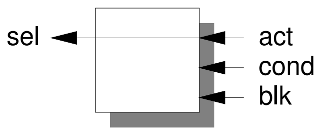
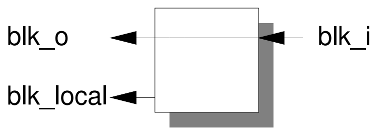
<table>
<tr><td>ControlNode(wait)</td><td>BlockNode(wait)</td><td>Data-Path(v)</td></tr>
<tr><td></td><td></td><td>Data-Path(<code><expr></code>)</td></tr>
</table>

The circuit in the block node is simply a wire from `blk_i` to `blk_local`, indicating that if there is a pending safe port invocation, then the `wait` statement is not sensitive to conditional expression of the wait statement (remember that `blk_local` of the block net is glued with the `blk` pin of the control net). In addition, `blk_o` is glued to zero, saying that when the statement following the wait statement is executed, there is no pending safe port invocation yet.

The `cond` pin of the control node is connected to the output of Data-Path(`<expr>`).

Dummy Vertex

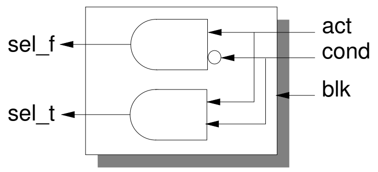
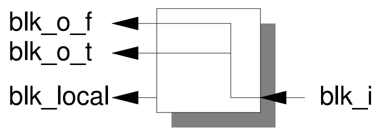
Dummy vertices are created for *label* kernel constructs (labels are the positions where `branch` statements can jump to). The circuits that are synthesized for dummy vertices are pretty simple. They are shown in the following table. The `act` pin of the control node is simply wired to the `sel` pin. That is, if the dummy vertex is chosen for execution, then the vertex following the dummy vertex will also be chosen. The `cond` pin of a dummy vertex is left open and the `blk` pin is not used. The `blk_i` pin of the block node is wired to the `blk_o` pin. That is, the vertex does not affect the boolean flag which indicates whether there is a pending safe port invocation. The `blk_local` wire of the block node is left undriven.

<table>
<tr><td>ControlNode(dummy)</td><td>BlockNode(dummy)</td><td>Data-Path(v)</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

No data-path logic is generated for a dummy vertex.

Branch Vertex

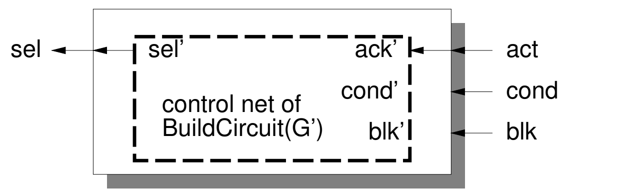
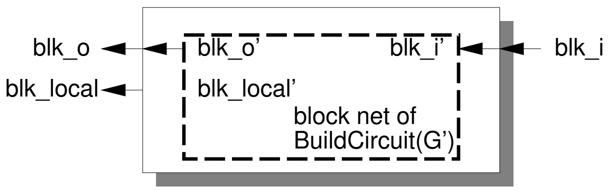
Given a NCFG node v for a branch statement `branch(<expr>) <label>`, the circuit nodes shown in the following table are generated. In the control node, based on the result of `<expr>`, it chooses one of the branches (`sel_t` or `sel_f`) for execution. The `blk` input to the control node is not used. In the block node, `blk_i` is connected to both `blk_o_t` and `blk_o_f`, indicating that if there is a pending safe port invocation executed before the branch vertex, then both branches will know of the pending port invocation. The `blk_local` is left open.

ControlNode(*branch*)	BlockNode(*branch*)	Data-Path(v)
		Data-Path(`<expr>`)

For branch vertex, the output the Data-Path(`<expr>`) is wired to the `cond` pin of the control node.

Serial-Composition Vertex

To compile a serial composition vertex v, the NCFG G' that is contained in $\gamma(v)$ is recursively compiled. As shown in the table below, the compiled control and block nets for G' (shown in dotted rectangle) are then glued with the pins of the control and block nodes for v. In the control net, the `act` pin of the control node of v is connected with the corresponding pin (`act'`) of the control node of the first vertex of G'. The `sel` pin for v is wired to the `sel` output (denoted by `sel'` in the table) of the control node of the last vertex in G'.

ControlNode($serial - composition$)

BlockNode($serial - composition$)

Data-Path(v)

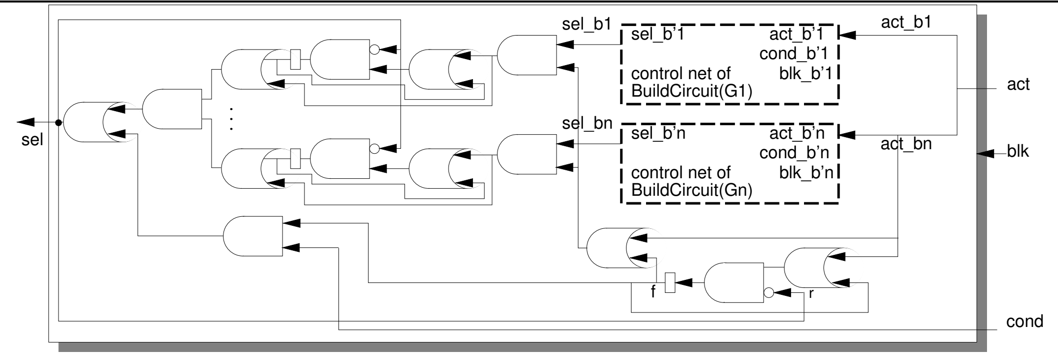
Similarly, in the block net, the `blk_i`, `blk_o` pins of the block node for v are connected to the `blk_i` pin (denoted by `blk_i'`) of the first vertex and `blk_o` pin (denoted by `blk_o'`) of the last vertex in G', respectively. The `blk_local` pin is left open.

There is no data-path logic generated for a serial composition.

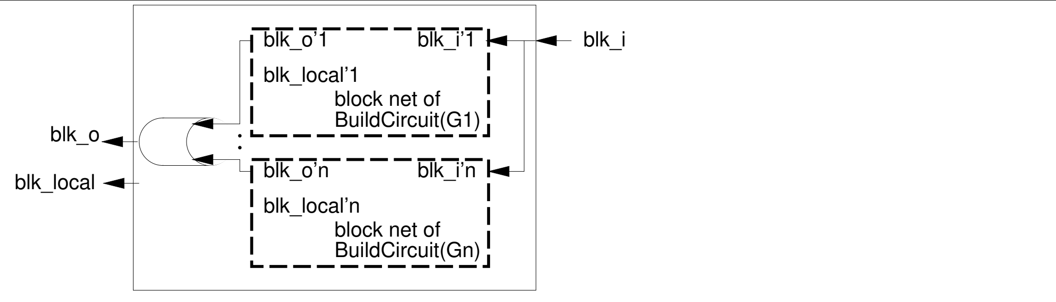
Parallel-Composition Vertex

Given a NCFG parallel composite vertex v for a kernel construct `fork <stmt1> || ... || <stmtn> join (<preempt_expr>)`, the circuit shown in the following table is synthesized. In the control net, one latch is allocated for each branch to memorize the fact that the branch has finished. That is, if a branch finishes execution, the latch for the branch get charged. The latch is discharged when the `fork/join` statement finishes execution (the instant when all branches are done).

ControlNode($fork/join$)



BlockNode($fork/join$)



Data-Path(v)

Data-Path(`<preempt_expr>`)

Note that there is another latch labeled `f` that is allocated for a `fork/join` statement. The purpose of the latch is to point out if the `fork/join` is still *active* (that is, some of its branches have not finished execution yet).

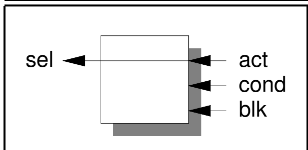
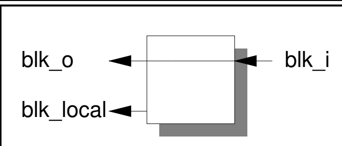
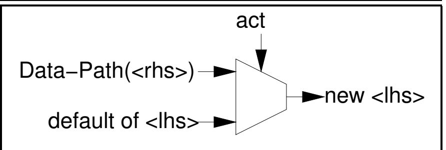
The `cond` pin of the control node is used to abort the execution of the `fork/join` statement. In other word, it is used to force the `fork/join` statement to hand over the control of program execution to the statement following the `fork/join` statement. However, it does not influence the execution of the branches.

Each branch of the `fork/join` statement is recursively compiled and connected to the control/block node of the `fork/join` statement as shown in the above table. Let G_i denote the NCFG of the i-th branch of v. In the control node, the `act` pin of the control node of the first vertex in G_i (denoted by `act_b'`$_i$) is wired with `act_b`$_i$. The `sel` pin of the control node of the last vertex in G_i (denoted by `sel_b'`$_i$) is wired with `sel_b`$_i$. Similarly, in the block node, the `blk_i` pin of the block node of the first vertex in G_i (denoted by `blk_i'`$_i$) is wired with `blk_i`. The `blk_o` pin of the block node of the last vertex in G_i (denoted by `blk_o'`$_i$) feeds an `or` gate. The output of the `or` gate is glued with the `blk_o` pin of the block node.

The output of Data-Path(`<preempt_expr>`) is connected to the `cond` pin of the control node for the `fork/join` statement.

Assignment Vertex

For an assignment `<lhs> = <rhs>`, the control and block nodes are generated the same way nodes are generated for dummy vertices. The circuit nodes are shown in the following table.

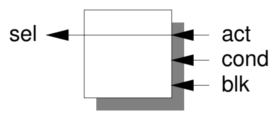
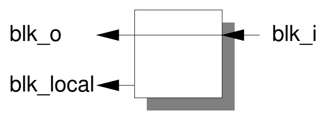
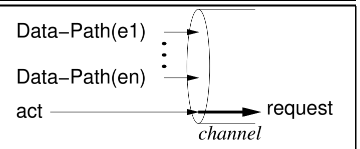
ControlNode($assign$)	BlockNode($assign$)	Data-Path(v)
		

Regarding data-path, what is generated for an assignment is a multiplexer choosing the default value for the LHS (left-hand-side) variable or the output of Data-Path(`rhs`). The control input to the multiplexer is derived from the `act` input pin of the control node. The outputs of these multiplexers are then collected by the compiler generated resolution logic to determine the values of variables.

Basically, the V++ compiler forms a priority graph (a directed graph) according to the `priority` statements in the V++ program. The graph is then topologically sorted to get a linearized priority list. According to the priority list, the V++ compiler can then generate resolution logic (or priority decoder) to resolve multiple assignments from multiple ports/processes to the same variable. Section 3.2.5 gives the details of resolution logic generation.

Port-Invocation Vertex

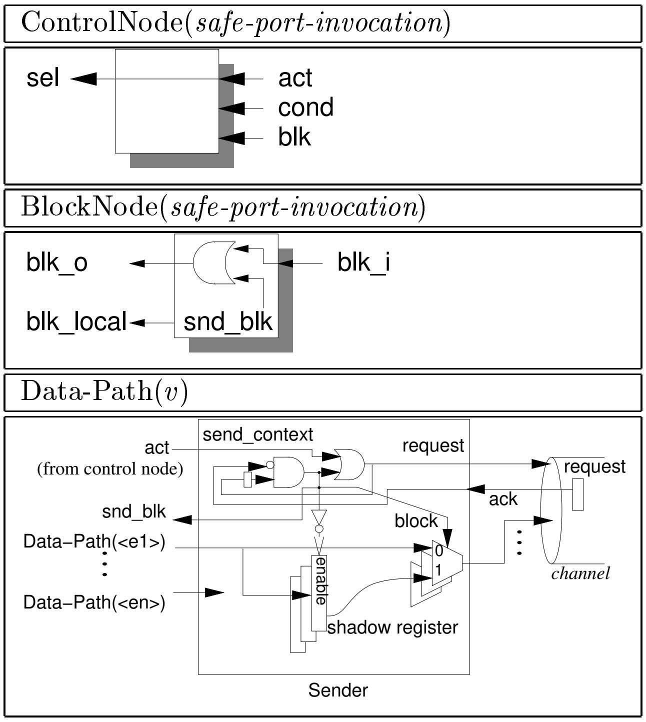
To send a message (<id>(<e1>,...<en>)), the data-paths for all the argument expressions are first generated. Outputs of these expressions are then wired with the channel that is responsible for delivering the message. The act pin of the control node is glued to the request pin of the channel. In the control node and block node, similar to the dummy vertex, the act pin of the control node is wired to the sel pin, and the blk_i input of the block node is connected to the blk_o pin.

ControlNode(*port-invocation*)	BlockNode(*port-invocation*)	Data-Path(v)
		

Safe-Port-Invocation Vertex

The following table shows the circuits that are generated for safe port invocations (safe <id>(<e1>,...,<en>)). In the data-path for a safe-port-invocation vertex, similar to port-invocation vertex, the data-paths for all the argument expressions are first compiled. In addition, a set of *shadow registers* are allocated to hold the arguments in case ack does not come back in the next clock cycle after the safe port invocation has been made (that is, the channel that is responsible for delivering the message cannot latch the data to be sent in time). The data-path computes a snd_blk signal which indicates if the port invocation has been suspended. If that is the case, then the shadow register(s) stops loading new data. In addition, the req is kept high signaling the channel that the port invocation is pending.

In the block node, the snd_blk signal computed by the data-path is ored with the blk_i pin, and the result is glued to blk_o. In the control node, the act input pin is used in the data path to determine if the statement is selected for execution.



3.2.3.2 Legal Processes

Before we jump into the details of compiling V++ constructs into kernel constructs, we should note that not all combinations of V++ constructs (or kernel constructs) have a good interpretation in terms of hardware. For example, consider the following code segment:

```
1    lab:;
2    a = a + b;
3    branch (a==8) lab;
```

The code is essentially a loop which keeps adding `b` to `a` until it is greater than 8. Moreover, all iterations should be done in the same clock cycle. If such programs were allowed, then to synthesize hardware for the code segment, one needs to find out, in the worst case, the number of times the loop body (line 3) will be executed. To achieve this, one needs to do reachability analysis, which is awfully expensive, if not unsolvable, for a V++ program. Actually, the above code segment indirectly violates the single assignment rule.

```
function checkLegality (G : (V, E)) returns {legal,illegal}
   σ ← ∅ // σ records the visited vertices
   foreach v ∈ V do
      if (v ∈ σ)
         continue;
      if (κ(v) = wait)
         continue;
      φ ← {v} // φ is the wave-front of vertices being visited
      ψ ← {v} // ψ can be reached from v without passing @
      foreach u ∈ φ do
         φ ← φ − {u}
         foreach e : u → w ∈ E do
            if (κ(w) = wait)
               continue;
            else if (w ∈ ψ)
               return illegal
            else if (w ∈ σ)
               continue;
            else
               φ ← φ ∪ {w}
               ψ ← ψ ∪ {w}
            fi
         od // foreach e : u → w ∈ E
      od // foreach u ∈ φ
      σ ← σ ∪ ψ
   od // foreach v ∈ V
   return legal
endfunction
```

Figure 3.8: The Algorithm for Checking The Legality of An NCFG

Therefore, a simple restriction is imposed on the allowed forms of kernel programs. The restriction is that, for each cycle in the CFG of a port/process, there should be at least one wait statement on the cycle. A NCFG/CFG that satisfies such a condition is called *legal*. If a program whose NCFGs/CFGs are all legal, then the program is *legal*. Whether a NCFG/CFG is legal or not can be checked by a graph traversal. Given a CFG $G = (V, E)$, the algorithm in Figure 3.8 checks whether a CFG is legal.

From this point on, we consider only legal programs.

3.2.4 Compiling V++ Process Constructs Into Kernel Constructs

To compile ports/processes, an intermediate construct is introduced. The intermediate construct, [<cond>:<handle>] (which is neither a kernel construct nor a V++ construct), can be regarded as a variation of the watch (preemption) clause. A statement like <stmt>[<cond>:<handle>] means that, when <stmt> is waiting, the execution can be preempted by asserting condition <cond>. That is, if <cond> is asserted when <stmt> is waiting for certain events, <stmt> is immediately "woken-up" and <handle> (the handler) is executed. If <handle> does not transfer flow of control to another place (by using branch), then <stmt> resumes execution right after the statement which just gets interrupted.

The differences between a watch clause and [] are:

1. watch clauses can only be attached to wait statement, loop statements (while or whenever statements) or composite statements (fork/join or {}). [] clauses can be attached to any statement (even though they may have no effects on the statement),

2. watch can affect "delay-less" statements. In addition, it aborts the statement once the preemption condition occurs.

   [] only searches for buried @ statements and inserts the preemption condition and handler to influence the execution of these @ statements. That is, when the preemption condition occurs, if the handler does not transfer control to other places, the port/process will resume execution right after the preempted @ statement.
   For example, consider the following two segments:

V++ Code	Kernel code
{ foo = bar+1; } watch (baz)	{ foo = bar+1; } [baz:{}]

   If baz is asserted, then the assignment on the left will not be executed. On the other hand, the assignment on the right will get executed (because there is no @ statement).

1:	*wait(<cond>)*	→	@(<cond>)
2:	*wait(<cond>) watch(d) <stmt>*	→	*if (d) { <stmt> }* *else {* @(<cond>) [<d>:<stmt>] }
3:	@(<cond>) [<d>:<stmt>]	→	@(<cond> | <d>); branch (!<d>) <label_done>; <stmt>; branch (1) <label_done>; <label_done>:

Table 3.2: Compilation rules for wait statements. The unconditional branch statement is used to ensure @ statements preempted by multiple [] will be translated correctly (i.e., only the exception handler with the highest priority will get executed). Refer to Figure 3.9 for examples.

As another example:

V++ Code	Kernel code
{ wait(cond); foo = bar+1; } watch (baz)	{ @(cond); foo = bar+1; } [baz:{}]

Assume that both statements are waiting for cond. For the above codes, if baz is asserted, execution of the wait statement is aborted and the assignment will not get executed. However, in the kernel code, the @ will get aborted (immediately when baz gets asserted) and the assignment will be executed.

We now show how V++ constructs get compiled into kernel constructs. In the following translation rules, all V++ constructs are in italic, all kernel constructs are in typewriter type style.

3.2.4.1 Compiling Wait Statements

wait-statements are simply compiled into the @ kernel construct according to the first rule in Table 3.2. The second rule compiles a wait statement with a watch clause.

The third rule states that a @ statement that is preempted by condition <d> is

```
@(cond)
   [<d1>:<s1>][<d2>:<s2>]
```

⇒

```
@(cond | <d1>)
   [<d2>:<s2>];
branch(!<d1>)
   <label_done>;
<s1>;
branch(1)
   <label_done>;
<label_done>:
```

⇒

```
@(cond | <d1> | <d2>);
branch(!<d2>)
   <label_done>;
<s2>;
branch(1)
   <label_done>;
branch(!<d1>)
   <label_done>;
<s1>;
branch(1)
   <label_done>;
<label_done>:
```

Figure 3.9: An example intermediate statement and the compilation sequence

translated into a `@` statement which is not only sensitive to `<cond>`, but also `<d>`. The exception handler will be the first statement to be executed when the `@` statement sees `<d>`. In the rule, `<label_done>` is a compiler-generated label that each `wait` statement has a unique `<label_done>`. At first glance, the second `branch` statement in the rule seems to be redundant (since it unconditionally transfers control to `<label_done>`). However, it is needed to ensure the correctness of translation for `@` statements that are preempted by multiple preemption clauses. We will see this in the example shown in Figure 3.9.

Figure 3.9 shows an example intermediate statement and the code that will be generated according to the rules in Table 3.2. Note that the preemption clause that is closer to the `@` statement has lower priority. That is, if both `<d1>` and `<d2>` are asserted at the same time, only `<s2>` will get executed.

Now we see why the unconditional `branch` statement is put in the third translation rule. Without it, when both `<d1>` and `<d2>` are true, both `<s1>` and `<s2>` will get executed. However, we only want `<s2>` (the one with the highest priority) to be executed.

3.2.4.2 Compiling Basic Statements

The compilation rules for basic statements (assignments, port invocations, and safe port invocations) are listed in Table 3.3. The first three rules show a straightforward compilation from V++ constructs to kernel constructs. Rules 4-7 show how preempted basic kernel statements (label, assignment, port-invocation, and safe-port-invocation) are

	Basic Statements:		
1:	*<lhs> = <rhs>;*	→	<lhs> = <rhs>;
2:	*<port>(...);*	→	<port>(...);
3:	*safe <port>(...);*	→	safe <port>(...);
4:	<label>: [d:<stmt>]	→	<label>:
5:	<lhs> = <rhs>; [d:<stmt>]	→	<lhs> = <rhs>;
6:	<port>(...); [d:<stmt>]	→	<port>(...);
7:	safe <port>(...); [d:<stmt>]	→	safe <port>(...);

Table 3.3: Compiling basic statements (assignments, port-invocations, and safe port-invocation) into kernel constructs

compiled. They indicate that the preemption clause has no effect on data-path elements. Basically, preemption conditions only affect control constructs.

3.2.4.3 Compiling Conditional Statements

Table 3.4 shows how conditional statements (`if/else` statements or `switch` statements) are compiled. The labels `<label_done>` and `<label_else>` are compiler-generated labels that are unique for each `if/else` statement.

Rules 1-2 compile `if/else` statements into `branch` kernel constructs. Rules 3-4 deal with `switch` statements. Instead of compiling `switch` statements directly into kernel construct, they are first translated into `if/else` statements in V++.

Rule 5 indicates that the preemption clause has no effects on `branch` construct.

3.2.4.4 Compiling Loop Statements

The loop constructs (`while` statements and `whenever` statements) are compiled according to the rules in Table 3.5. Note that when a `whenever` loop is entered, it will not be voluntarily broken out unless preempted by `watch` clauses. On the other hand, a `while` loop will be (re)entered only when its conditional expression is true.

In these rules, `<label_loop>` and `<label_exit>` are compiler-generated labels which denote the re-entrant and exit points for a loop statement, respectively. `<label_loop>` and `<label_exit>` are unique for each loop statement. If a `while/whenever` statement is attatched with more than one `watch/restart` clause, then they all refer to the same `<label_loop>` and `<label_exit>` labels. The `watch` and `restart` clauses of a loop statement will be processed according to their syntactic order. That is, the "innermost" clause

	Conditional Statements:		
1:	*if (cond)* *<stmt1>* *else* *<stmt2>*	→	`branch(!cond) <label_else>;` *<stmt1>;* `branch(1) <label_done>;` `<label_else>:` *<stmt2>;* `<label_done>:`
2:	*if (cond)* *<stmt1>*	→	`branch(!cond) <label_else>;` *<stmt1>;* `branch(1) <label_done>;` `<label_else>:` `<label_done>:`
3:	*switch (<cond_expr>) {* *<tag_expr1>: <stmt1>;* *. . .* *<tag_exprn>: <stmtn>;* *}*	→	*if (<cond_expr> == <tag_expr1>)* *<stmt1>;* *else if (<cond_expr> == <tag_expr2>)* *<stmt2>;* *. . .* *if (<cond_expr> == <tag_exprn>)* *<stmtn>;*
4:	*switch (<cond_expr>) {* *<tag_expr1>: <stmt1>;* *. . .* *<tag_exprn>: <stmtn>;* *default: <stmtd>;* *}*	→	*if (<cond_expr> == <tag_expr1>)* *<stmt1>;* *else if (<cond_expr> == <tag_expr2>)* *<stmt2>;* *. . .* *if (<cond_expr> == <tag_exprn>)* *<stmtn>;* *else* *<stmtd>;*
5:	`branch (<cond>) <label>` `[<d>:<stmt>];`	→	`branch (<cond>) <label>;`

Table 3.4: Compilation rules for conditional statements

	Loops:		
1:	*whenever(c)* <*stmt*>	→	<label_loop>:; @(c); <*stmt*>; branch(1) <label_loop>; <label_exit>:;
2:	*whenever(c)* *watch(d)* <*stmtw*> <*stmt*>	→	*if (d)* { <*stmtw*> } *else* { <label_loop>:; @(c); <*stmt*>[d:{<*stmtw*>; branch(1) <label_exit>}]; branch(1) <label_loop>; <label_exit>; }
3:	*whenever(c)* *restart(d)* <*stmtr*> <*stmt*>	→	*if (d)* { <*stmtr*> } <label_loop>:; @(c); <*stmt*>[d:{<*stmtr*>; branch(1) <label_loop>}]; branch(1) <label_loop>; <label_exit>:;
4:	*while (c)* <*stmt*>	→	<label_loop>:; branch(!c) <label_exit>; <*stmt*>; branch(1) <label_loop>; <label_exit>:;
5:	*while (c)* *watch(d)* <*stmtw*> <*stmt*>	→	*if (d)* { <*stmtw*> } *else* { <label_loop>:; branch(!c) <label_exit>; <*stmt*>[d:{<*stmtw*>;branch(1) <label_exit>}]; branch(1) <label_loop>; <label_exit>:; }
6:	*while (c)* *restart(d)* <*stmtr*> <*stmt*>	→	*if (d)* { <*stmtr*> } <label_loop>:; branch(!c) <label_exit>; <*stmt*>[d:<*stmtr*>; branch(1) <label_loop>]; branch(1) <label_loop>; <label_exit>:;

Table 3.5: Compilation rules for loops

(the clause that is closest to the loop construct) will be processed first and the "outermost" clause will have the highest priority.

What is not shown in Table 3.5 is how to compile `break` and `continue` statements. To compile `break` (`continue`), one simply replaces the statement with `branch(1) <label_exit>` (`branch(1) <label_loop>`), where `<label_exit>` (`<label_loop>`) is the compiler-generated label for the corresponding loop statement.

3.2.4.5 Compiling Composite Statements

Table 3.6 lists the compilation rules for composite statements (parallel composition, `fork/join`, and sequential composition, `{}`). The first three rules compile `fork` statements into kernel `fork/join` constructs. If a `fork/join` construct is preempted by more than one condition, the first one (the one that is closest to the `fork/join` construct) is compiled first and the last one has the highest priority.

The labels `<label_exit>` and `<label_done>` in the third and sixth rule are compiler-generated labels that are unique for each `fork`/sequential statement.

3.2.4.6 Compiling Ports/Processes

The translation of a port/process is actually pretty simple. It is translated into a while loop which is sensitive to the conjunction of the guarding condition of the port/process, the `valid` signal coming from the channel that is responsible for carrying the incoming arguments, and signal `block` (which is the `blk_o` pin of the block node of the last vertex in the NCFG of the port/process's body statement). Table 3.7 shows the translation rule. Signal `consume` (refer to Figure 3.4) is a compiler generated wire that is defaulted to 0 (if it is not explicitly assigned in a clock cycle, its value is 0). It is used to indicate if the arguments to the port/process have been digested. The `running` pin in Figure 3.4 is derived by taking the disjunction of all `act` pins of the control nodes in the port/process.

The `clk` signal in the table is used as the synchronizing clock of the `wait` statement of the translated code (the `while` statement). Whenever `<stmt>` refers to the variables in `<args>`, it will be redirected to the data pins of the channel that is responsible for delivering data to port `<id>`.

	Composite statements:		
1:	*fork* { *<stmt1>* || ... || *<stmtn>* }	→	`fork` *<stmt1>* `|| ...||` *<stmtn>* `join (0)`
2:	*fork* { *<stmt1>* || ... || *<stmtn>* } *watch (w) <stmtw> ;*	→	(`fork` *<stmt1>* `||` `...` `||` *<stmtn>* `join (0)`) `[w:`*<stmtw>*`]`
3:	(`fork` `<stmt1>` `||` `...` `||` `<stmtn>` `join (c)`) `[p:<stmtp>]`	→	`fork` `{<stmt1>; <label_done1>}` `[p:branch(1) <label_done1>]` `||` `...` `||` `{<stmtn>; <label_donen>}` `[p:branch(1) <label_donen>]` `join (c | p);` `branch(!p) <label_exit>;` `<stmtp>;` `branch(1) <label_exit>` `<label_exit>`
4:	{ *<stmt1>; ...; <stmtn>* }	→	{ *<stmt1>*`; ...;` *<stmtn>* }
5:	{ *<stmt1>; ... ; <stmtn>* } *watch(w) <stmtw>;*	→	*if (w)* { *<stmtw>* } *else* { *<stmt1>*`; ...;` *< stmtn>* } `[w:`*<stmtw>*`]`
6:	{ `<stmt1>; ...<stmtn>;` }`[d:<stmtd>]`	→	{ `<stmt1>[d:{<stmtd>;` `branch(1) <label_done>}];` `...` `<stmtn>[d:{<stmtd>;` `branch(1) <label_done>}];` `<label_done>:` }

Table 3.6: Compilation rules for composite statements

	ports:		
1:	*<id>(<args>) @(clk) when (cond)* *<stmt>*	→	*while(1) {* *wait(cond & valid & !block);* *<stmt>;* *consume=1* *}*

Table 3.7: Compilation rule for ports/processes

3.2.5 Synthesizing Resolution Functions

Once ports/processes are compiled, resolution logic is generated to resolve conflicting assignments from different ports/processes. Basically, every variable either defaults to a combinational value (wire) or its previous value (latch). Resolution functions are used to prioritize among assignments and determine the final result of the variable. The `priority` keywords specify the priorities of ports/processes in terms of their ability in influencing the values of variables. The logic generated for resolution functions are essentially priority decoders (a series of multiplexers) built from the *precedence graphs* constructed in the following manner.

For each variable α, a precedence graph (a directed graph) $P_\alpha = (V_\alpha, E_\alpha)$ is introduced. For each port that assigns to α, a vertex is allocated in V_α. In addition, the partial order among ports that is introduced by `priority` statement is translated into a set of edges in P_α. That is, for each $p_i < p_j$ in the `priority` statement, an edge from p_j to p_i is added to E_α if both p_i and p_j access α.

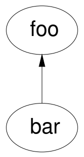
After the precedence graph is formed (the precedence graph is available when the whole object is parsed), a topological sort is performed on the precedence graph to get a linearized list. According to the linearized list, the priority decoder can then be constructed. Figure 3.10 is an example V++ code segment, the induced precedence graph, and the resolution logic that is generated by the V++ compiler.

3.3 Linkage

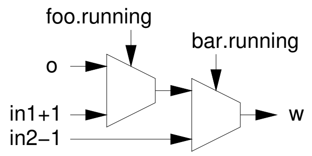
During compilation, logic inside objects are instantiated. However, there are a few things that have to wait till the whole design is available. For example, one cannot generate

```
wire bit w default 0;
foo(in1) @(clk) {
   w = in1 + 1;
}
bar(in2) @(clk) {
   w = in2 − 1;
}
priority bar > foo;
```

(a) V++ program.



(b) Precedence graph.



(c) Generated resolution logic.

Figure 3.10: Resolution logic generated by `priority`.

the resolution logic in channels until the number of ports that will be hooked up to the channel is determined.

To be more specific, the tasks that are performed by the V++ linker are:

1. Patching reference holes – If an object makes references to other objects/channels whose definitions are not available at compile time, then the compiler will create reference holes. The linker, on seeing these holes, directs the references to appropriate object/channel instances and creates necessary connections.

2. Finalizing parameters – Parameters are important constructs for reusable, parameterized designs like data-path components. However, it is an obstacle to separate compilation. In V++, parameters are passed via *generators*. Generators are places where expressions are evaluated to constants and these constants in term can determine the size of logic in an object.

   Consider the two objects in Figure 3.11. In object `root`, `bar` is instantiated with parameter `w` bound to 8. This is done by binding the formal argument (`width`) of the generator of `bar` with 8 (`{8}` at line 3). `width` is then assigned to variable `w` in `bar` (line 8). Therefore, instance `bi` has a variable `var` which is an eight-bit vector. Note that without compiling object `foo`, one cannot determine value of `w` (and the width of `var`) in object `bar`.

   One of the responsibilities of a V++ linker is to build the instantiation hierarchy and propagate parameters, in a top-down fashion, along the hierarchy. In a sense, the linker needs to do all the compilation if all the non-root objects have parameters that

```
object root {
   bit[3:0] width;
   bar{8} bi = new bar();
}
object bar {
   bit[3:0] w;
   generate(bit[3:0] width) {
      self.w = width;
   }
   bit[w-1:0] var;
}
```

Figure 3.11: Passing parameters between objects.

depend on their parents. One way to (partially) get around the problem and to make the compiler to do as much as possible (hopefully, most of the compilation work can be reused) is to compile the object with some default parameter(s) and record the parameter values and the corresponding compiled objects. In the above example, we can declare parameter `w` with `bit[3:0] w default 4` so that `w` gets default value 4. A master copy of the object definition has to be carried around so that if a new instance of the object needs to be created with parameters that are different from the recorded ones, then the master is compiled with the new parameters (by the linker) and the new pair (parameter values + compiled object) will be recorded.

3. Synthesizing channels/Wiring objects and channels – The main difference between the V++ linker and the traditional programming language linker is that the V++ linker is also responsible for target code (logic) generation.

   In programming languages, when one routine calls another routine, the calling routine is sometimes referred to as the *caller* and the called routines is referred to as the *callee*. For programming languages, the compiler can generate the code in the caller routine for argument preparation (the code that pushes arguments onto the stack), the "prologue" (the part of a callee routine that prepares the stack for local access), and the "epilogue" (the part of a callee routine that restores stack for the caller) of the callee routine at compile time, without knowing what the caller and the callee are. All that a compiler needs to know is the position and type of each argument, and the symbolic address of the callee routine (so that program loaders can fill in the actual address if the callee is relocatable).

For V++, even though all senders and receivers follow the same protocol in interfacing with channels, the compiler cannot generate channels before the whole design is hooked up. This is due to the fact that the compiler needs to know the number of ports that are hooked up to a channel in order to generate the arbitration logic in the channel.

The remainder of this section is devoted to channel synthesis.

3.3.1 Channel Synthesis

Basically, there are two ways objects can communicate with each other – ports and public variables. That is, an object can expose a subset of its variables so that the parent object can sample the values of these variables. An object can also send data to another object via port invocations. For each port, a *channel* is synthesized to collect/deliver the data to be sent to the port. In addition, the channel is also responsible for resolving multiple data sending requests and decoupling the sender from the receiver. Ports and channels communicate with each other via a set of standard interface signals. Between the sender port and the channel, two control signals, `request` and `ack` are used so that:

1. the sender can request for message send by asserting `request`,

2. the channel can tell the sender if the message has been latched by asserting `ack` signal.

Between the receiver port and the channel, `valid` and `consume` are provided so that:

1. the channel can tell the receiver that there is a valid message for it by asserting `valid` signal,

2. the receiver can signal the channel that the message has been consumed by asserting `consume`.

In addition to the above control signals, the port/channel interfaces also provide necessary data wires so that arguments of message sends can be latched/delivered by the channel.

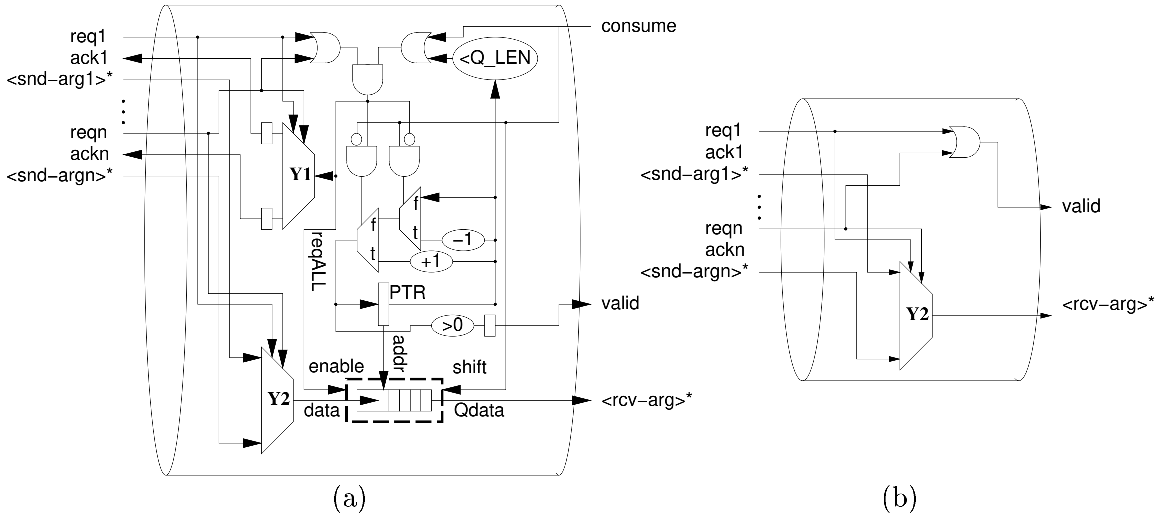


Figure 3.12: Synthesized logic for channels – (a) Receiver is a sequential port/process. (b) Receiver is combinational.

As an example, assume that port `foo` is trying invoke port `bar`. Assume that the channel cannot sample the data from `foo` because either another higher priority port is also trying to invoke `bar` at the same time or the buffer in the channel is full. In this case, the channel has to inform `foo` that the data cannot be sampled by deasserting the `ack` signal. If `foo` is doing a safe port invocation, then the deasserted `ack` will

- make `foo` to lock the data to be passed in the shadow registers,
- make `foo` to keep requesting for port invocation (`bar`), with the arguments coming from the shadow registers,
- prevent `foo` from "making progress" (or doing further computation, consuming more messages).

The logic that is synthesized by the V++ linker for a channel is shown in Figure 3.12. The circuit shown in Figure 3.12.(a) is a linker-generated channel which is connected to a sequential receiving port/process. It takes n port invocation requests (`req1`, `req2`, ..., `reqn`), resolves them, and signals them if the data (`<snd-arg1>*`, ..., `<snd-argn>*`) have been successfully latched by driving the acknowledgment signals (`ack1`, ..., `ackn`). It informs the receiver that `<rcv-arg>*` is a valid datum by asserting `valid`, and the receiver tells that the data have been digested by setting `consume`. In the synthesized channel logic, `Y1` is a de-multiplexer which, controlled by `req1`, ..., `reqn`, connects its input (`reqALL`) to one of its outputs while keeping the other outputs grounded. `Y2` is a multiplexer which, controlled by `req1`, ..., `reqn`, selects one of its inputs (`<snd-arg1>*`, ..., `<snd-argn>*`) and glues it to the output. The circuit in the dashed rectangle is an addressable FIFO with data input `data` and data output `Qdata` (which is derived from the rightmost FIFO cell). When `enable` is asserted, `data` is loaded into the location addressed by `addr`. When `shift` is asserted, the contents of the FIFO are shifted to the right by one cell. The content of the rightmost cell is dropped and the content of the leftmost cell will be zero (unless `data` is loaded into this cell at the same time). `Q_LEN` denotes the depth (number of cells) of the FIFO. This number can be set by designers. `PTR` is an address pointer which keeps track of the location of the first empty slot in the channel's FIFO.

Figure 3.12.(b) is a linker-generated channel which is connected to a combinational receiving port. Note that the FIFO (as well as the circuit for maintaining the FIFO pointer) in the circuit is removed as combinational port invocations take zero time. In addition, the `ack` and `consume` signals are removed. Therefore, one cannot do safe port invocation to a combinational port.

Figure 3.13 illustrates how the linker-generated channels are hooked up with various combinations of sender and receiver ports. Figure 3.13.a shows a sequential port invoking another sequential port. In Figure 3.13.b, a sequential port sends messages to a combinational port and in Figure 3.13.c, a combinational port sends messages to a sequential port. In Figure 3.13.d, messages are sent from a combinational port to another combinational port. Figure 3.13.e shows the interconnection for a safe port invocation.

The resolution functions in Figures 3.13.a, 3.13.c, and 3.13.e are the same as the circuits shown in Figure 3.12.a. The resolution functions in Figures 3.13.b and 3.13.d are the same as the combinational circuits in Figure 3.12.b.

3.4 Optimization

The previous sections deal with the generation of logic in the general settings. In this section, we turn our attention to the optimization of generated circuits for V++ programs. Consider the channel circuits generated by the V++ compiler as shown in Figure 3.14.a. Under certain circumstances, the generated circuits can be optimized:

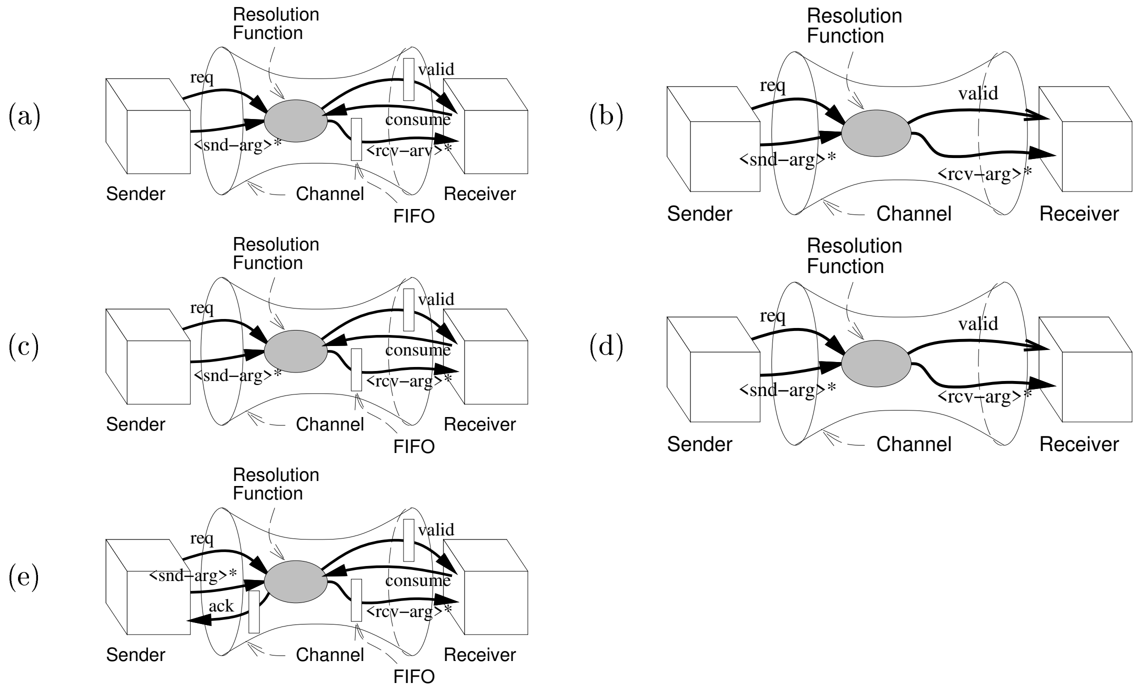


Figure 3.13: Connecting channels and ports – (a) A sequential port invokes another sequential port. (b) A sequential port invokes another combinational port. (c) A combinational port invokes another sequential port. (d) A combinational port invokes another combinational port. (e) A safe port invocation (This can happen only when both the sender and the receiver are sequential ports/processes).

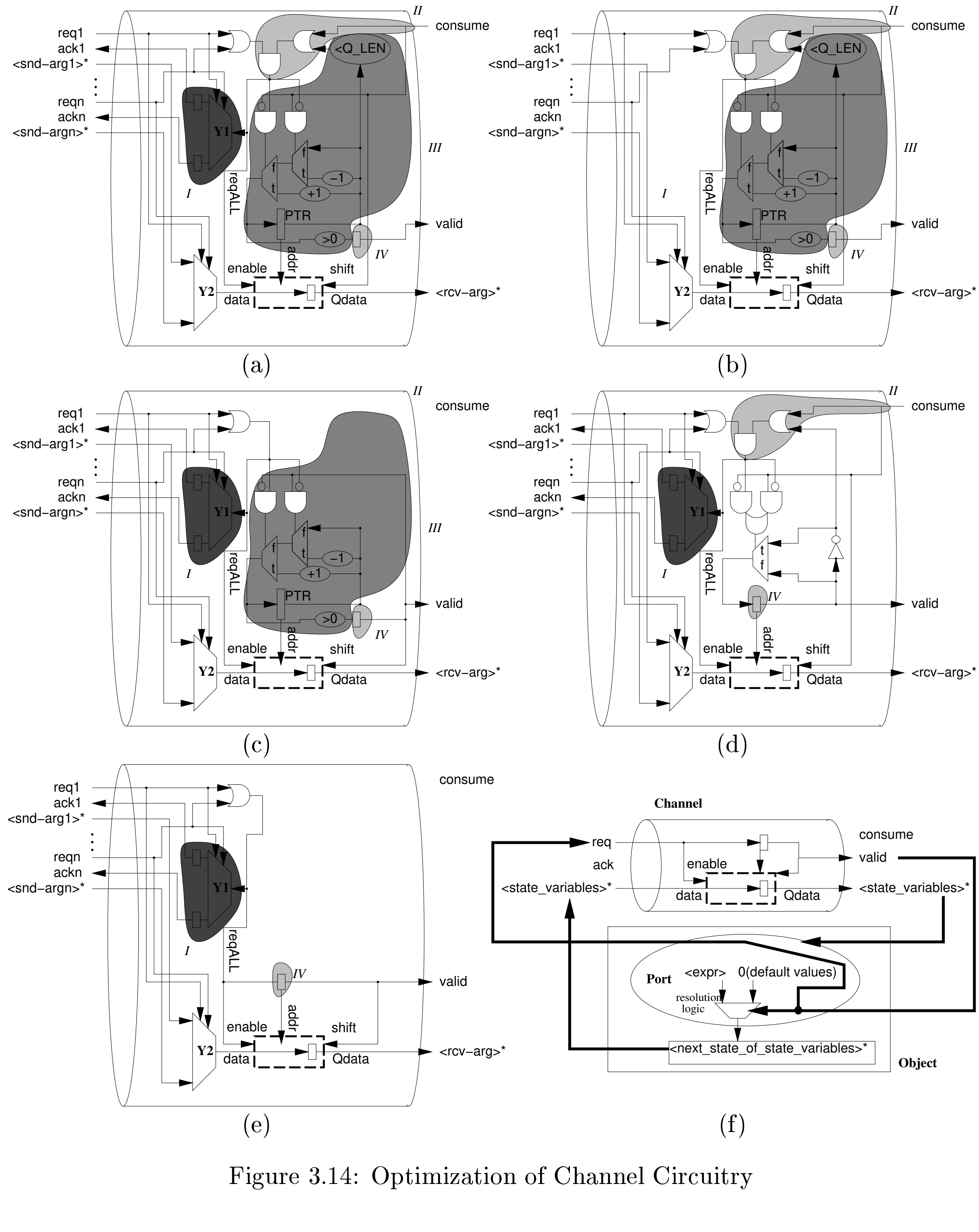


Figure 3.14: Optimization of Channel Circuitry

1. Assume that the invocations to the receiver port are all default (non-safe). In this case, there is no need for generating `ack` signals. Therefore, the circuit labeled I in Figure 3.14.a can be removed. The result is shown in Figure 3.14.b.

2. If the receiver does not suspend (that is, it does not do safe send and it is not guarded by guarding conditions. Therefore, `consume` will always be asserted when there is a `valid` signal), then the channel does not need to query the `consume` signal. That is, the circuitry labeled II in Figure 3.14.a can be discarded. The reason is that the circuitry is used to block the request signal coming from the sending ports when the FIFO is full and the receiving port is suspended. However, as the receiving port cannot be suspended (by assumption), this circuitry is not necessary. The result is shown in Figure 3.14.c.

   Note that the `shift` signal of the channel's FIFO now comes from `valid`.

3. Assume that the size of the FIFO queue is one, then the circuit labeled III in Figure 3.14 can be simplified. Figure 3.14.d shows the resulting circuit. Note that, in this case, the `PTR` register is identical to the `valid` latch (labeled with IV). Therefore, it is removed. In addition, as the `valid` register is only one bit, the adder, the subtracter, the comparator, and the multiplexers are simplified.

4. If both conditions 2 and 3 are true, then the FIFO pointer calculation circuitry can be removed as shown in Figure 3.14.e.

One of the most often used coding styles of specifying FSMs in V++ is:

```
fsm (<state_variables>*) @(clk) {
   <next_states_of_state_variables> = <expr>;
   fsm (<next_states_of_state_variables>*);
}
```

After applying the above rules, we can optimize the circuit as shown in Figure 3.14.f. Note that the next state of `valid` latch is simply itself. Therefore, `valid` can be replaced by a constant (0 if `fsm` is not invoked in the constructor, 1 if `fsm` is invoked in the constructor of the containing object). In this case, the "redundant logic" created by the V++ compiler/linker is the resolution function (which is a mux with its control glued to constant `valid`).

3.5 Related Compilation Techniques

Esterel [BC84, Ber92, Hal93] is one of the most popular system design languages. It was originally designed for reactive systems (systems that continuously react to their environment at a pre-determined speed). Esterel is an imperative parallel language with event manipulation and very high level control flow constructs. It is sometimes referred to as synchronous language as it is based on the perfect synchrony hypothesis: computation and communication take zero time.

What is similar between the V++ compiler and the Esterel compiler [Ber92, Ber96] is that both are based on the syntax directed compilation technique. Both compilers work by first translating the programs into kernel constructs. The kernels are then translated into circuit components/templates and the components are connected according to the syntax structure of the source program.

The major difference between the V++ compiler and the Esterel compiler is that the V++ compiler takes into account the interactions between the control flow constructs and the data flow constructs. In both languages, the control flow constructs determine which data-path components can be executed. In V++, in addition, the data-path elements can also suspend the execution of control flow constructs (e.g., unfinished safe port invocations can make wait statements insensitive to the conditional expressions). By consulting block nets, control nets can determine if the execution should be blocked or not. Furthermore, the encoding of the control nets of V++ is different from the *skeletons* (the structures formed by control wires) of Esterel. In Esterel, there is always a set of "hot wires" (*select* wires whose values are 1; *select* wires are control wires used to select constructs for execution). On the other hand, in V++, since the execution of a process can be suspended by pending safe port invocations, the control net can be totally "cold" (no data-path is selected for execution). The way V++ knows where to resume execution after safe port invocations are done is by charging the latches for the wait statements (Section 3.2.3.1).

V++ addresses the problem of compositionality by enforcing delays on inter-object communication. Therefore, one of the major responsibilities of the V++ linker is to synthesize channels – the inter-object communication devices. Esterel, like other synchronous languages, allows zero-delay inter-module communication. Consequently, expensive analysis like causality checking (e.g., [Shi96]) may be necessary for the whole design to ensure the behavior of the design is well-defined. What is novel in the V++ compiler is the way it compiles safe port invocations. By using shadow registers, it introduces a fully automatically generated message flow control mechanism without introducing unnecessary performance degradation. Also, V++ uses graph structures as intermediate representations for the control/data flow of kernel programs.

One interesting difference between the two compilers is the way they deal with control transfer constructs. In V++, the set of labels that a `branch` statement can jump to is recorded in a symbol table in the compiler so that the control (block) net can be formed by wiring the `sel_t` (`blk_o_t`) pin of the `branch` statement to the `act` (`blk_i`) pin of the target label. The symbol table is simply thrown away after the process is compiled. On the other hand, the Esterel compiler encodes the destinations for possible control transfer (the *continuations*) as pins of circuit templates. Therefore, if a statement is enclosed in an environment where there are ten possible locations it can potentially jump to, there will be ten pins (in addition to the regular control pins) on the circuit module that is generated for the statement.

3.6 Simulation

In the previous sections, we address the compilation, linking, and logic generation (synthesis/optimization) of V++ programs. In this section we show how a V++ design can be simulated. There are several ways to simulate a V++ program. For example, once the logic of the whole design is generated, one can use the techniques in [MMS$^+$95, CMM$^+$96] to flatten and simulate the whole circuit. For large circuits, these global compilation technique can be prohibitively time-consuming.

On the other hand, one can take advantage the structured communication in V++ to decompose the whole simulation task into several smaller sub-tasks. Excluding the public wire communication, the only way an object can communicate with another is via port invocation. In addition, all port invocations are realized by channels. Note that for *sequential port invocations* (where the receiver is a sequential port. These are the only allowed forms of inter-object port invocation), the channel uses latches to hold signal values for the receiver and it introduces at least one (or more) cycle of delay. These latches ensure that there is no combinational path from the logic of the sender port to the logic of the receiver port. Therefore, the sender and the receiver ports can be scheduled for evaluation in any order (if they belong to different objects).

```
Simulator(design)
   Simulate_Channels'(design)
   for simulation_cycle = 1, 2, ..., n
      foreach root r ∈ design
         Simulate_Object(root)
      Simulate_Channels(design)

Simulate_Object(object)
   foreach object instance o ∈ children(object)
      Simulate_Object(o);
   EvaluateLogic(object); // Combinational Channels are evaluated here
   UpdateLatch(object);

Simulate_Channels'(design) // this routine only evaluates logic once
   foreach channel c ∈ design
      EvaluateLogic(c);

Simulate_Channels(design)
   foreach channel c ∈ design
      EvaluateLogic(c);
      UpdateLatch(c);
      EvaluateLogic(c); // to propagate the new latch value so that
                        // the receiving ports can see it.
                        // the 'next' hardware cycle actually
                        // starts from here.
```

Figure 3.15: A V++ Hierarchical Simulator

However, public wires (which can be visible only to the parent of an object in the instantiation hierarchy) restrict the possible schedules. As combinational signals are allowed to propagate uni-directionally (upward along the instantiation hierarchy), the logic in children should be evaluated before their parents. Figure 3.15 shows one possible hierarchical simulator for a V++ design. Currently, the V++ compiler/linker can be instructed to generate Java programs (as the simulation code for V++ programs) with the structure shown in Figure 3.15.

The simulator works in the following manner. For each hardware cycle, the simulator first evaluates the combinational circuits in channels so that the values of the latch walls in channels can propagate towards channel outputs. The logic in objects are then evaluated and the latches in objects are updated. It is assumed (and enforced by current V++ compiler) that no latches in objects can directly drive inputs to channels. If they do, buffers will be inserted. This way, channels will not see the next states of the latches in objects. Before the end of the current hardware cycle, the logic of channels are evaluated again and the latches of channels are updated. Note that we need to evaluate the logic in channels one more time in order to propagate the inputs to channels to the latches in channels.

Chapter 4

Experimental Results

The algorithms presented in the previous chapters have been implemented. In this chapter, we put V++ into action. We apply the V++ design language to a real design – the InfoPad. We use V++ to design several variations of InfoPad to show what the trade-off (design time, area overhead, performance overhead, etc.) is by using V++.

4.1 The InfoPad Project

The InfoPad project [BBB+94, TPDB98] is a low power wireless multimedia communication device. InfoPad is not a computer. It is a hand-held, battery operated communication terminal like a multimedia cellular phone. It communicates with a net of closely spaced base stations to provide wireless communication with the backbone networks. InfoPad has a graphical display and a pen/keyboard input. It also supports bi-directional audio and full color video. However, it does not perform general purpose computing, storage, and communication, which are done by the servers on the network.

The architecture of a pad is shown in Figure 4.1. A low voltage bus (IP bus) serves as the backbone. It connects several logical chips. The chips talk to the IP bus via common bus interface modules to ensure interoperability. A chip that can serve as a data source (sink) is also called a *master* (*slave*). When a master has data to send, it requests bus access. At least one bus cycle later, after the access request is granted, the master can put data on the bus.

For example, when a user taps the pen on the screen, five bytes (a pen event packet) have to be sent from the pen device (master) to the transmitter chip (slave) to reflect the pen movement. After receiving the packet, the transmitter puts the packet in a packet buffer, and the packet number in a *ready FIFO*. The transmitter talks to the microprocessor to schedule the packet for transmitting over the wireless modem. The communication between the transmitter chip and the microprocessor can be done by either interrupts or polling. The transmitter chip uses a static RAM as buffers for packets coming from different chips. One of the major functionalities of the transmitter chip is to manage the buffer RAM for up to five streams (each stream is basically a logical device which can send packets to the transmitter). Once the processor detects that a packet is ready to send, it schedules the packet for transmitting by putting the packet number on an *ok FIFO* in the transmitter. The transmitter constantly monitors the *ok FIFO* for any packet that is available to be sent over the radio. The servers on the network can then compute the response based on the received pen events. If the response involves updating the pixels on the pad, the server will send packets to the pad. On the pad, when the receiver chip (master) gets bytes from the wireless modem, it redirects them to the appropriate destination (text/graphic chip) based on the type field of the packet. Basically, one can imagine the receiver chip as a byte interpreter. It analyzes the packet header to find out the target chip for the packet body and then forwards the data bytes to the destination without touching them.

All these data streams are sent over the bus using a common data-gram consisting of a start-of-packet byte (SOP), the data bytes, and the end-of-packet (EOP) byte. The InfoPad design partitions the address space on the bus into a set of pairs. When a master intends to send a stream of data to the slave, it first sends a byte to the high address of the address pair that is allocated to the slave indicating the start of a new packet. In the following cycles, it sends the data bytes to the low address of the address pair. At the end of the transaction, it sends another byte to the high address of the address pair signaling the end of the packet. A slave device can be associated with more than one address pair so that multiple masters can "virtually concurrently" send data to the same slave. The concurrency is not true as the bus can transfer only one byte at a time. Therefore the data transfers need to be interleaved. The data sources have programmable destinations which can be determined by the microprocessor at initialization time.

Above the transmitter/receiver interface is the XILINX chip which computes error correction code (ECC) and cyclic redundancy check code (CRC) for packet header/data. The chip is also responsible for synchronizing itself and the incoming bit-stream/packet from the receiving modem.

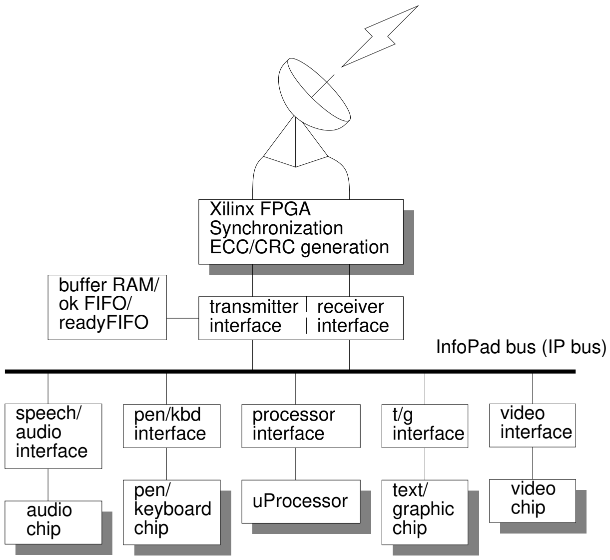


Figure 4.1: InfoPad Architecture

4.2 Implementing InfoPad Using V++

We use V++ to implement four versions of "InfoPad". We implement the major functionalities of the backbone communication structures on a pad with various V++ coding styles and subsets of V++ constructs. In this experiment, we implement the circuits that are necessary to carry out the main functionality of InfoPad. For example, we make sure that the pen packets will be buffered and finally transmitted over the radio. We also make sure that whenever the receiver chip gets packets for the text/video device, it will finally get displayed on the screen. There are parts of the design that are not implemented. For example, the parts of the circuits that are programmed at initialization time are wired to constants (e.g., data sources will send data to a fixed address, instead of the one indicated by its programmable control register). In addition, some parts are scaled down (and abstracted). For instance, the buffer RAM is shrunk to a 4-byte (32-bit) memory instead of a 64k-byte RAM. The pointers that address the RAM for buffer management are also scaled down. Since the devices that are hooked up with the IP bus are operating at different speeds, they are only modeled by nondeterministic packet generators/consumers. This is because currently V++ allows only one clock domain.

4.2.1 Reusing Chips In Different Implementations

In order to determine the overhead involved when compiler channels are generated for communication, in the following implementations of InfoPad, the V++ code for the chips/devices (pen/keyboard, text/graphic, tx/rx, and microprocessor) are reused. However, the interface they use to communicate with each other are slightly modified in order to accommodate various design styles.

The transmitter device is modeled by a packet sink and the receiver device is modeled by a nondeterministic packet generator. In the experiment, the receiver is programmed to generate packets targeted only for the text/graphic chip.

The pen/keyboard device is modeled as a nondeterministic keystroke generator. A packet is generated for each keystroke by wrapping the keystroke data with an appropriate header/tail. This job is accomplished by a "packetizer" FSM. After a packet is formed, the pen/keyboard interface requests bus access. For example, if the pen/keyboard device needs to send a pen event packet (5 bytes) to the transmitter, it first writes an SOP (start-of-packet) to a specific address on the IP bus that is reserved for the pen/keyboard. It then writes the bytes of the packet body in the following few bus cycles (the IP bus can accommodate one byte per bus cycle). Finally, the pen/keyboard device writes an EOP (end-of-packet) to the transmitter interface. Bytes of packets from different masters on the bus can be arbitrarily interleaved. But the order among bytes in the same packet should be maintained by the masters. The transmitter interface tells the beginning/end of packets by being written at the addresses that are reserved for signaling (SOP/EOP). The transmitter interface tells the source of the data bytes of packets by decoding the address that are being written (remember that for each master, a pair of addresses is reserved, one for signaling and the other for the packet body/data). This way, the transmitter can handle multiple streams of bytes from multiple devices. That is, at any point in time, the transmitter might have partially formed packets from different packet sources.

The microprocessor (an ARM processor) is modeled by a scheduler FSM with nondeterministic delay. The microprocessor can issue bus requests and it can receive service requests (interrupts). The processor serves the interrupt by reading the *ready FIFO* in the transmitter interface and then writes the packet number back into the *ok FIFO* of the transmitter interface once the packet is scheduled to be transmitted.

The text/graphic device is simply a data sink.

The major difference among the various implementations is the way data are delivered. In implementation I, a user designed channel is used to implement the IP Bus that is used in the InfoPad project. In implementation II, III, and IV, variations of V++ built-in channels are used to deliver data. In addition, due to the delays introduced by V++ built-in channels, extra buffer entries are allocated in implementation II, III, and IV. These extra FIFO entries are used to prevent bytes from getting lost when FIFO's are full. Details of the modifications to the sizes of FIFO buffers are provided in Section 4.2.3.

4.2.2 Implementation I : The InfoPad Bus

In the first implementation, we redo the InfoPad backbone in V++ (IP bus plus the interface chips) and try to match the original design in a gate-to-gate fashion. For this design, we restrict ourselves to use only the "level-0" V++, which is essentially SMV. Basically, level-0 V++ provides data operators (gates), assignments (naming of wires), conditional statements (multiplexors), instantiations. General port invocations are not allowed. The only allowed forms of port invocations are:

- A port can invoke itself via tail recursion – this kind of port invocation is only used to design finite state machines. Note that such a form of port invocation can be handled by the optimization techniques described in Section 3.4.
- Sending messages to user defined channels – the V++ compiler does not generate logic in this case. All that a compiler does is to glue request, acknowledgment, valid, and consume signals to the user defined channel. [1]

With these level-0 V++ constructs, one can describe gates, wires, and connections. We refrain from using the "advanced constructs" such as safe sends.

If an interface cannot lock the bus when it has data to send (the acknowledgment signal does not come back in time), it will keep requesting the bus with the same data. This is essentially what a safe send tries to do. The only difference is that the logic has to be crafted by hand in this implementation.

[1] In the V++ language, each channel, whether it is built-in or user defined, has to have at least four control signals – *request*, *acknowledgment*, *valid*, and *consume*. A channel uses *request* and *acknowledgment* to communicate with the sender port so that arbitration among multiple delivery requests can be made. The channel uses *valid* and *consume* to communicate with the receiving port. When the channel has data available for the port, it asserts *valid*. As soon as the receiving port samples the data, it asserts *consume* so that the channel can abandon the data it has latched for the receiver.

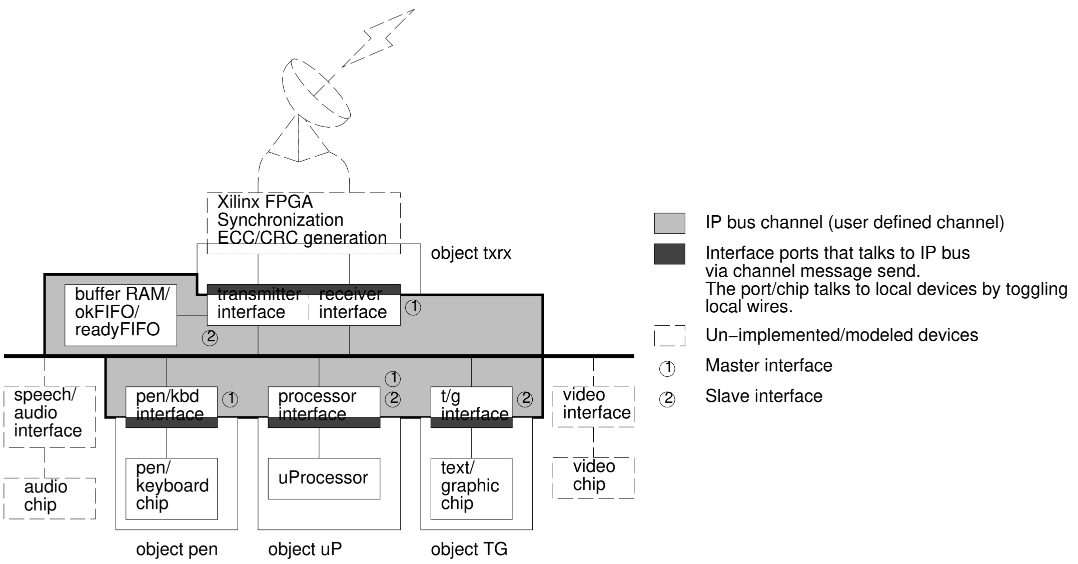


Figure 4.2: Implementation I : InfoPad using IP Bus.

The architecture is shown in Figure 4.2. The user defined channel arbitrates the incoming requests from master interfaces and determines which interface may put its data on the bus. The user defined channel (the light gray rectangle) is used to implement the IP bus and the bus interfaces. The white rectangles in the user defined channel are the parts of interfaces that are put in the user defined channel code. The black rectangles that are connected to these white rectangles are the parts of interfaces that sit in each individual object. They are basically channel invocations which glue wires that are local to each object to wires in the channel. The port/channel boundaries are drawn in such a way that there is no combinational path from one port (device) to another.

We implement/model the devices/interfaces depicted in solid lines. These are: the pen/keyboard chip, the microprocessor, the text/graphic chip, a small fraction of the XILINX FPGA (that is responsible for communicating the tx/rx interface), the pen/kbd interface, the processor interface, the t/g interface, and the transmitter/receiver interface (and their buffer RAM). The devices that are drawn in dashed lines are not modeled or implemented.

As mentioned above, the pen/keyboard interface is divided into two pieces, one (the white rectangle) resides in the user defined channel and the other (the black rectangle) resides in the pen object. The black rectangle is a V++ channel invocation which directly wires control/data signals in the user defined channel to signals in the pen object. The white rectangle participates in the arbitration concerning which interface gets access. It also has multiplexors, controlled by the output of the arbitration logic, to gate data/address to the IP bus.

The text/graphic interface simply routes the data on the IP bus to the text/graphic device. The address on the IP bus is decoded by the interface so that the text/graphic device knows if it is the intended receiver of the data on the bus.

The microprocessor interface routes the interrupt line to the microprocessor and contends for bus access whenever the microprocessor tries to use the IP bus. The microprocessor has the highest priority among all the masters.

The receiver interface is divided into two pieces. One (the black rectangle) sits in the `txrx` (transmitter/receiver) object and the other (the white rectangle) sits in the user defined channel. The black rectangle is simply a channel invocation. The white rectangle can be partitioned into the receiver logic and the transmitter logic. The receiver logic is a byte interpreter which redirects the packet body data to the appropriate destination device according to the type of the packet. The transmitter logic is basically a packet buffer manager which collects packet bytes in the buffer RAM until packets are complete. As was indicated in Section 4.2.1, the transmitter logic can handle multiple data sources. For each of these data sources, a region in the buffer RAM is reserved. Basically, each region is organized as a ring of buffers. When the transmitter receives a byte, it tells where the byte comes from by decoding the address that is being written. The transmitter then put the byte in the appropriate buffer ring. When the transmitter receives an EOP, it knows that the packet is complete. It sends interrupts to the microprocessor when there is a packet formed and ready to be sent. The processor then chooses which packet is allowed to be transmitted by writing the packet number into the ok FIFO. When the *ok FIFO* is not empty, the transmitter interface presents the packets to the the XILINX chip so that an appropriate error correction code can be generated and the packet can be delivered through the wireless modem. The buffer manager is the most complicated logic of all the interfaces. In implementation II, III, and IV, the buffer manager (as well as the buffer RAM) as well as the byte interpreter of the receiver interface are put in the `txrx` object.

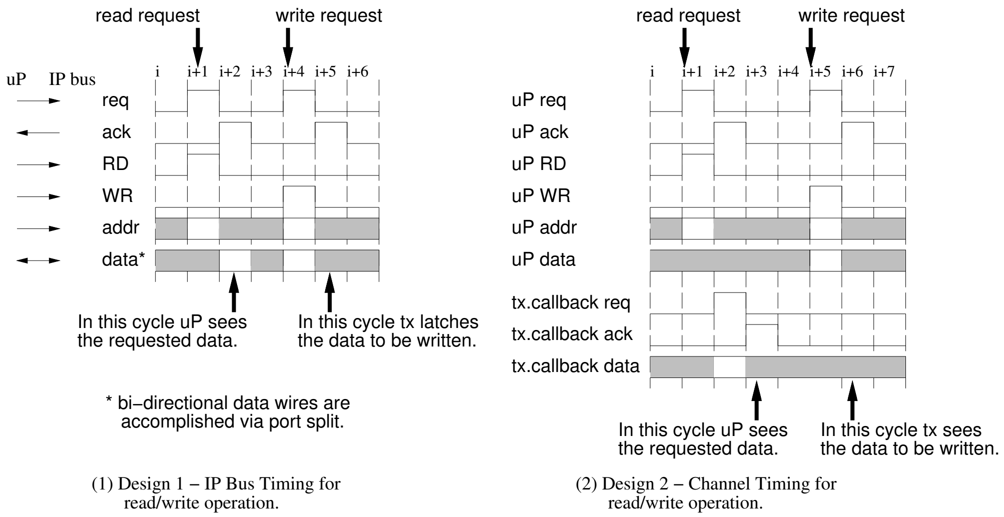


Figure 4.3: Timing for One Read Operation Followed by A Write Operations on (1) IP Bus and (2) V++ Generated Channel.

4.2.3 Implementation II : Making Use of V++'s Built-in Communication Mechanism

In this experiment, we take advantage of the built-in V++ message delivery mechanism to do the communication. The master devices sends data to the slaves by safe port invocations. The pen object sends bytes of keystroke packets directly to the `tx` port (transmitter) of the `txrx` (transmitter/receiver) object. The microprocessor object can either request to write into `tx`'s control registers or to read the control registers. When the processor requests to read data from the `tx` port, it will first send the read request to the port. At least a cycle later, when the `tx` has the data ready, it calls back the microprocessor with the data the processor requested. Note that interrupts are also delivered by safe port invocations. In this implementation, when the receiver interface has a packet for the text/graphic chip, it is delivered by a safe port invocation. Figure 4.4 shows the architecture of implementation II.

Note the difference in bus timing between implementation I (Figure 4.3.(1)) and implementation II (Figure 4.3.(2)). With the IP bus, if a read/write operation can go through without being blocked, it takes two bus cycles to accomplish the transaction. With the V++ built-in channels, it takes one cycle for the processor to send a read request to the transmitter and another cycle for the transmitter to call the processor back with the data

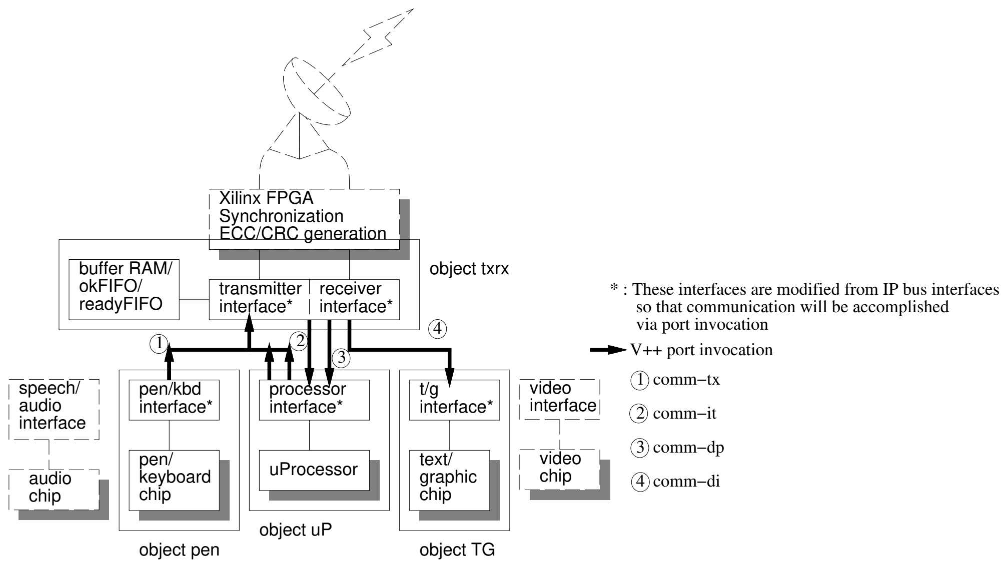


Figure 4.4: Implementation II : InfoPad using V++ Built-In Message Sends

requested. Therefore, the requested data is visible to the processor at the beginning of the third bus cycle after the data request is generated (which is the first cycle).

Similarly, timing of interrupts are also changed as they are delivered by V++ built-in channels. Due to the delay introduced by channels, the buffering mechanism implemented on the transmitter device needs to be modified. Consider the following scenario. If the transmitter collects a complete packet at cycle i, it will put the packet number in the ready FIFO indicating that a packet is ready to be sent. It will also interrupt the microprocessor regarding the availability of the new packet in the mean time. Assume that the ready FIFO gets full at cycle i. With the original IP bus implementation, the microprocessor responds to the interrupt at cycle $i+1$, i.e., at the $i+1$-st cycle, the processor will take over the IP bus and block other devices from accessing the bus. If the microprocessor fails to lock the bus at the $i+1$-st cycle (this is possible for implementation II), then in case there is a new packet formed (e.g., the pen/keyboard chip sends a EOP (end-of-packet) to the transmitter chip) at that cycle, the newly formed packet can potentially get dropped. In other words, by locking the bus at the $i+1$-st cycle, it is assured that no new packet will be formed at the $i+1$-st cycle and no packet will get dropped from the full FIFO.

One way to get around this problem is to enlarge the FIFO so that it alerts the processor before it is really full. This modification is made to implementations II, III, and

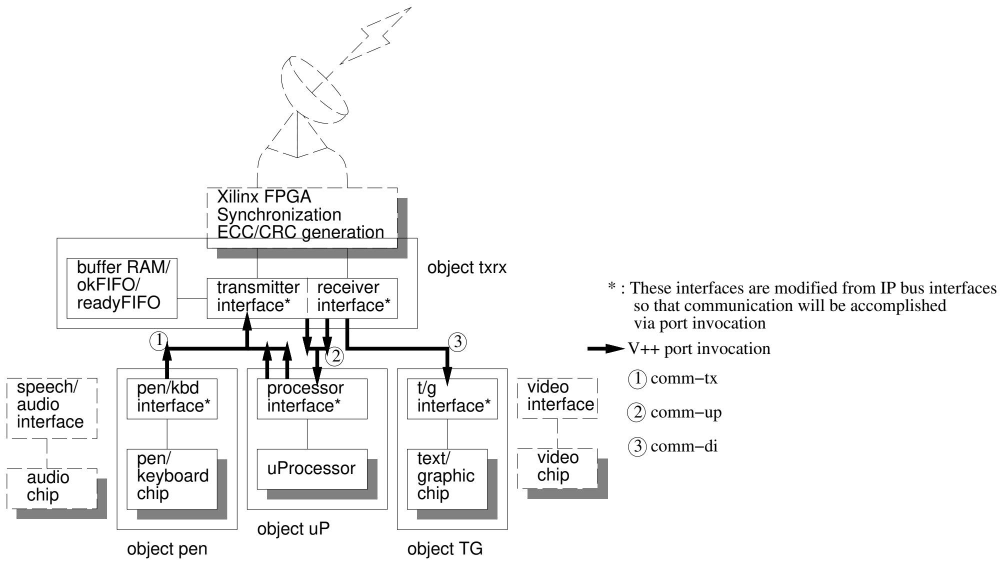


Figure 4.5: Implementation III : InfoPad Using V++ Built-In Message Sends as Buses

IV to ensure that no packets are dropped.

4.2.4 Implementation III : Using V++ Channels as Buses

Similar to implementation II, in this design we use V++'s built-in channels to do all the inter-device communication. Unlike implementation II, all communication destined at the same target device is implemented via invocation of the same port. Figure 4.5 illustrates the architecture of implementation III.

Contrast Figures 4.4 and 4.5. Basically, each object is now equipped with only one port for all the service(s) it provides. For example, the processor object provides one port through which it can be interrupted or called. The `txrx` object also provides only one port which serves all read/write requests for the object.

Using this coding style, the intent is to illustrate the use of the built-in V++ channels as "easy-to-use" buses that are generated by the V++ compiler on the fly. We want to understand what is the trade-off for using "compiler-generated buses" instead of the hand-crafted buses.

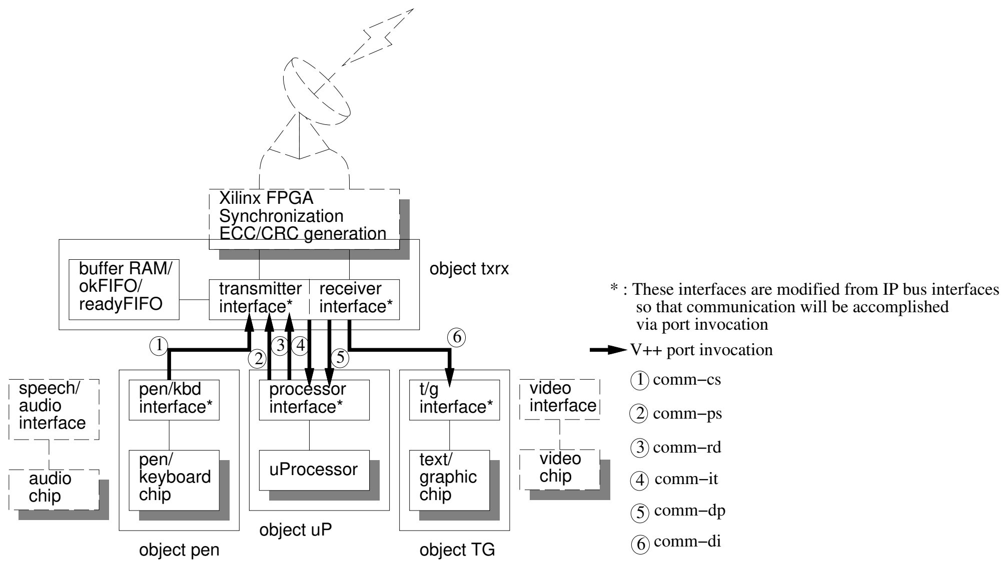


Figure 4.6: Implementation IV : InfoPad Using V++ Built-In Message Sends and All Channels Are Split

4.2.5 Implementation IV : Provide Separate Port for Each Service

Similar to implementation II, in this design we use V++ built-in channels to do all the inter-device communication; for each service provided by a device, a separate port is provided. This implementation is used as a contrast to implementation III (where all services provided by an object are accessed via a single port). For example, the transmitter allows the pen device to write into its signal/data registers to send pen packets. The transmitter also allows the microprocessor to read/write into its internal latches to program the control registers. In implementation IV, these services are provided through distinct ports. In implementation II, both services are provided by a single port and are resolved by the compiler generated channel when both parties request services at the same time.

4.3 Results

We implemented the four variations of InfoPad in V++ and compiled the designs into SMV. The SMV files are then translated into blif files via several intermediate format compilers. The generated blif files are then simulated in VIS [BHSV$^+$96b, BHSv$^+$96a]. We also mapped the blif files to various technology libraries using SIS [Sea92].

The experimental results are summarized in Table 4.1. There is one column for each variation of InfoPad. The rows are the size of the V++ source code (number of lines/number of bytes), the size of the SMV file generated by the V++ compiler (number of lines/number of bytes), the size of the blif files by flattening the SMV files, the area of the implementation when it is mapped to a circuit with cost functions that minimize delay using the MCNC technology library (`mcnc.genlib` + `mcnc_latch.genlib`), the area of the implementation when it is mapped to a circuit with cost functions that minimize area using the MCNC technology library, the area of the implementation when it is mapped to a circuit with cost functions that minimize delay using a standard cell technology library (`stdcell2_2.genlib`), the area of the implementation when it is mapped to a circuit with cost functions that minimize area using a standard cell technology library, the latch count, the literal count (reported from SIS [Sea92]), and the time (approximate) we spent on doing each design. The design time is further broken down into Table 4.2.

In this table, the circuits are mapped without any logic optimization. The reason of omitting the logic optimization is that, these implementations contain circuitry that are part of InfoPad but not used in the experiment. For example, there are control registers that can be programmed at initialization time by the microprocessor. However, in the experiment, the functionalities of the implementations that are controlled by these control registers are hard-wired. Nonetheless, the initialization circuitry and the circuitry used to access these control registers are still implemented in order to make the implementations as close to InfoPad as possible.

For those who are interested in applying logic optimization before circuits are mapped, we also apply script `script.rugged` to various components. The results can be found in Tables 4.3, 4.4, 4.5, and 4.6.

The breakdown of Table 4.1 is given in Table 4.2, 4.3, 4.4, 4.5, and 4.6. Table 4.2 shows the estimated time for doing the parts of variations of InfoPad. The rows represent the time it took us to design the transmitter interface (`tx`), the receiver interface (`rx`),

[1] Note that the design time for the last two designs (III and IV) does not reflect the real design time. It is the time we used to modify design II to derive design III and IV, respectively.

[2] The implementation is mapped to a minimum delay circuit using MCNC library gates.

[3] The implementation is mapped to a minimum area circuit using MCNC library gates.

[4] The implementation is mapped to a minimum delay circuit using standard cells.

[5] The implementation is mapped to a minimum area circuit using standard cells.

[6] The design time does not the time that was spent on reading InfoPad specification and understanding the InfoPad. This does not include the time for debugging.

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
Source Size (lines/bytes)	1227/39K	761/22K	777/22K	792/22K
SMV Size (lines/bytes)	4923/111K	2657/69K	2613/68K	2876/74K
blif Size (lines/bytes)	13104/227K	12100/226K	12307/230K	12357/225K
Mapped Area [2]	13085	13374	13623	13688
Mapped Area [3]	11612	11673	11997	11793
Mapped Area [4]	129616	135728	136768	136536
Mapped Area [5]	122200	123400	126128	125032
Latch Count	409	483	491	466
Literal Count	12232	11611	11672	11459
Design Time [6]	8.5 days	6 days	0.5 days [1]	0.5 days [1]

Table 4.1: Experiment Results of 4 Implementations of InfoPad Bus

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
Design Time(tx)	2.5 days	2.5 days	-	-
Design Time(rx)	1 day	1 day	-	-
Design Time(pen)	0.5 day	0.5 day	-	-
Design Time(t/g)	0.5 day	0.5 day	-	-
Design Time(uP)	1 day	1 day	-	-
Design Time(comm)	3 days	0.5 day	0.5 day	0.5 day

Table 4.2: Estimated Time for Designing InfoPad

the pen object (`pen`), the text/graphic object (`t/g`), the microprocessor object (`uP`), and the communication network (`comm`) among these objects. Note that for Design III and IV, the design time of certain components are marked with dash lines. It is because these components are essentially copied from Design II. However, minor modifications are made to the components in order to accommodate different design styles.

Tables 4.3, 4.4, 4.5, and 4.6 itemize the areas of components of various InfoPad implementations. Each entry is the areas of the component without/with technology independent logic optimization (script `script.rugged`) applied. The rows of these tables are: the area of the implementation when the whole design is mapped, the area of the `tx` chip (including the interface) when it is mapped along, the area of the `rx` chip (including the interface), the area of the `pen` device (including the interface), the area of the text/graphic (`t/g`) device (including the interface), the area of the microprocessor (`uP`, including the interface), and the area taken by the communication devices. For designs II, III, and IV, the areas of the communication devices are further broken down into their components (`comm-cs`, `comm-di`, `comm-dp`, `comm-it`, `comm-tx`, `comm-up`, `comm-ps`, `comm-rd`, refer to Fig-

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
	Standard Cell/Area			
total	122200/39712	123400/48768	126128/50600	125032/43240
tx	64600/20096	65488/21000	65488/21000	63832/19576
rx	11136/3896	10904/3576	10904/3576	10904/3576
pen	4776/1880	4896/2064	4896/2064	4896/2064
t/g	1456/640	1096/632	1096/632	1096/632
uP	7288/3304	7528/4592	8752/5232	8608/5184
comm	34624/11600	34304/21608	36560/23400	28752/17600
comm-cs				10648/6752
comm-di		7200/4064	7200/4064	7200/4064
comm-dp		3816/2112		3816/2112
comm-it		416/312		416/312
comm-tx		22872/15120	22872/15120	
comm-up			6488/4216	
comm-ps				5240/3480
comm-rd				1160/880

Table 4.3: Map InfoPad to The Standard Cell Library. The Technology Mapper Uses The Cost Function That Minimizes Area.

ures 4.4, 4.5, and 4.6).

Note that when built-in V++ channels are used (implementation II-IV) for communication devices, the size of the sources is about thirty percent smaller than the design that uses user defined channels (implementation I). This is because in implementation I, the designer has to specify all arbitration/resolution logic in the channel. The latch count shows that there is about a twenty percent increase in the number of latches when V++ built-in channels are used. We found that most of the extra latches come from the shadow registers used in safe sends. At first, it seems that implementations II-IV spend too much area overhead on compiler-generated circuits. However, after the whole implementations are mapped into circuits with various technology libraries and cost functions, we realize that the actual area overhead is quite small. For example, if we map the implementation onto MCNC technology library, the overhead is between 0.5 to 4.6 percent.

The main advantage of using V++ built-in communication mechanism seems to be the short design cycle. This is consistent with the sizes of various implementation. Basically, the larger the source code one needs to write, the longer it takes.

We also found that during the design process, using V++ built-in channels helps relieve ourselves from worrying about the bus timing. For example, assume that one object

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
	MCNC/Area			
total	11612/3708	11673/4644	11997/4816	11793/4123
tx	6068/1940	6205/1969	6205/1969	6074/1849
rx	1067/365	1026/342	1026/342	1026/342
pen	449/173	460/188	460/188	460/188
t/g	134/63	94/59	94/59	94/59
uP	676/301	694/402	808/470	804/471
comm	3267/1093	3217/2077	3461/2245	2759/1691
comm-cs				1040/648
comm-di		678/392	678/392	678/392
comm-dp		359/203		359/203
comm-it		38/29		38/29
comm-tx		2142/1453	2142/1453	
comm-up			641/400	
comm-ps				511/336
comm-rd				111/83

Table 4.4: Map InfoPad to The MCNC Technology Library. The Technology Mapper Uses The Cost Function That Minimizes Area.

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
	Standard Cell/Delay			
total	129616/42056	135728/50672	136768/52344	136536/44800
tx	68008/20640	70312/22120	70312/22120	69280/20048
rx	12080/4056	11400/3696	11400/3696	11400/3696
pen	4976/1960	5128/2160	5128/2160	5128/2160
t/g	1872/768	1880/760	1880/760	1880/760
uP	7520/3128	8088/4640	9448/5208	9616/5272
comm	39048/11784	37544/22016	39896/23856	34484/17984
comm-cs				12496/6816
comm-di		9056/4160	9056/4144	9056/4144
comm-dp		4768/2120		4768/2120
comm-it		440/320		440/320
comm-tx		23280/15416	23280/15416	
comm-up			7560/4296	
comm-ps				6264/3696
comm-rd				1248/888

Table 4.5: Map InfoPad to The Standard Cell Library. The Technology Mapper Uses Cost The Function That Minimizes Delay.

Design	I. IP	II. V++ (1)	III. V++ (2)	IV. V++ (3)
	MCNC/Delay			
total	13085/4004	13374/4876	13623/5084	13688/4434
tx	6727/2099	6915/2110	6915/2110	6964/1971
rx	1198/395	1138/358	1138/358	1138/358
pen	497/173	500/188	500/188	500/188
t/g	170/63	123/59	123/59	123/59
uP	741/331	785/457	939/511	927/517
comm	3626/1147	3830/2140	4092/2313	3208/1744
comm-cs				1224/655
comm-di		804/420	804/420	804/420
comm-dp		425/205		425/205
comm-it		38/30		38/30
comm-tx		2563/1485	2570/1485	
comm-up			718/408	
comm-ps				574/347
comm-rd				119/87

Table 4.6: Map InfoPad to The MCNC Technology Library. The Technology Mapper Uses The Cost Function That Minimizes Delay.

(e.g., a master) has some data to be delivered to another object (e.g., slave). The master has to keep requesting for bus access until it is granted. After the master is granted bus access, it places the data on the bus. In addition, the master should make sure that when it puts data on the bus, the slave is able to pick up the data. For InfoPad, the slaves are always available once the master gets the bus. In general, when a design involves a bus like structure, the correctness of a transaction depends on the relative timing of the devices involved in the transaction. For example, when a master has a piece of data to be sent to a slave, the master should explicitly check if the slave is able to receive the data (if the bus is owned by another higher priority master, or if the slave is busy and not listening to the bus, then the slave cannot receive the data) in order to prevent the data from getting lost. Using V++'s safe send, one simply does safe port invocation and the compiler does the rest. The compiler generated circuitry not only does handshaking between the master and the slave, it also buffers inputs to the slave in case the slave is busy.

Chapter 5

Conclusions and Future Directions

This dissertation addressed the problem of providing a language which is compositional and is consistent with the models used by most of the synthesis, formal verification, and cycle simulation engines.

In Chapter 2 we introduced a new Hardware Design Language, V++, which is based on FSMs. The heart of V++ is a new concept in hardware design languages: an automatically designed and implemented communications protocol, the *hardware stack* (arbitered FIFOs), embedded by a compiler/linker in the design itself. The protocol is efficient, generalizes current design practice, and introduces little overhead in terms of cycle time or area of a typical system.

We showed how to compile, optimize, and link V++ programs in Chapter 3. We compile V++ descriptions by first translating them into kernel constructs. NCFGs are then constructed from these kernel programs. We then show how to synthesize circuits from kernel constructs and how these circuit components can be connected according to the structures of the NCFGs. We show how to synthesize channel circuits for different port invocations. We also present some optimization techniques for V++ programs. In addition, we show how one can take advantage of the communication structures in V++ and build hierarchical simulators for V++ programs.

V++ is then applied to a real design, InfoPad, in Chapter 4 and various versions of InfoPad are implemented. All implementations are designed with V++. From the experimental results of Chapter 4, we found that V++'s simple communication mechanism is pretty efficient and actually introduces little resource overhead. Simplicity is exactly the reason we choose FIFOs as V++'s built-in communication mechanism. There are a huge number of more powerful alternatives that one may choose to implement inter-object communication. This is the reason we provide user defined channels to encourage people to design and experiment with their communication devices. In addition, the shorter design cycle and smaller source sizes can benefit large designs, improve maintainability, productivity, and provide correct designs more quickly. The simple and effective communication mechanism provided by the V++ compiler/linker relieves designers from the task of writing tedious and cumbersome communication code. Therefore it reduces the number of communication bugs.

On the other hand, V++ imposes the following constraints:

- In V++, all inter-object communication has to be accomplished via channels that induce at least one cycle of delay. This could impact the implementation of the design. For example, an interrupt will take at least one channel cycle to propagate. Therefore, if the interrupt is used to inform the processor that a buffer is full, then extra entries should be allocated in the buffer in order to compensate for the extra delay used to propagate the interrupt.
- In V++, a built-in channel can deliver only one message at at time. Therefore, if a compiler generated channel is used to implement the bus, then the potential "multi-tasking" that can take place on the bus (e.g., one device is generating an interrupt while the other device is sending data around) has to be modeled by multiple bus transactions. Therefore performance can be degraded.
- In V++, all arguments of a port invocation are uni-directional. In case bi-directional variables are desired, they need to be split into input only and output only variables.
- Currently, there can be only one clock domain in a V++ design. The only way to "model" the peripheral modules running at different clock rates is via nondeterminism. For instance, in the InfoPad example, we use nondeterministic event generators/consumers to model pen and text/graphic devices.

5.1 Future Directions

Current Hardware Description Languages either are based upon models that are incompatible with the ones that are used by many core synthesis, simulation, or verification engines, or they are not truly compositional in the sense that combining two well-defined, deterministic modules may result in nondeterministic or undefined behavior. V++ is based on the popular model (FSMs) that is used by all common synthesis, cycle-simulation, and verification engines. It gets around the compositionality problem by using the *hardware stacks* (arbitered FIFOs). However, by imposing delays on inter-object communication, designs may need to be modified in order to take advantage of what V++ can provide.

To make V++ more easily applicable to more designs, we foresee the following extensions to the language/compiler.

- For V++, one possible extension is to extend the current compilation/linkage mechanism so that port invocations can be mapped into various channel implementations according to various communication/delay/throughput requirements. Various implementations can be stored in a channel library so that designers can instruct the linker to pick up the desired channel implementation.

- Another possible extension is to allow multiple time domains in a design in order to facilitate the design of systems like multi-phase circuits. For example, if there exists a logical master clock (which is true for many practical systems), then the techniques introduced in [CB96] can be used to synthesize a "message sampler" between clock domains. Basically, the sampler coordinates message flow between sub-systems controlled by different clocks. It makes sure that when a message is passed from one clock domain to another, it will not get lost or duplicated.

- By applying V++ to numerous other designs (ASIC custom logic, microprocessors, IPs, etc.), we can learn valuable lessons of what is the strength and weakness of V++. This way, we can then share the lessons as style guides for designers who are interested in applying V++ to their designs but do not know how. In addition, the lessons also help us to further improve the V++ language and develop more powerful optimization techniques.

In the future, further investigation is necessary to integrate V++ into the design flow. Currently, V++ seems to be highly applicable to the following scenarios.

- Fast prototyping language – The third version of InfoPad in Chapter 4 is such an example. In this example, V++'s built-in channels are used to generate buses on the fly. This relieves designers from worrying about the tricky details of communication devices. Even though this approach sometimes introduces little area or cycle time overhead, it saves time spent on designing and debugging communication devices. For certain designs, once one can achieve the performance goal, there is little incentive in further improving the system performance. In addition, as the number of devices that can be put on a silicon chip grows dramatically, a small area overhead becomes acceptable. On the other hand, the key to the success of a product is the time to market. Therefore, it is vital to have a language that allows designers to quickly develop prototypes in order to shorten the design cycle.

V++ allows designers to prototype their idea without worrying too much about the communication details. In addition, V++ provides user defined channels so that designers can refine their designs and adopt their own communication devices if the built-in devices cannot provide desired performance/functionality.

- DSM (deep submicron) designs – Due to the advances in fabrication technology, feature sizes of devices on chips keeps shrinking. However, as sizes shrink, wire delays increase and gate delays decrease. Therefore, it may take more than one cycle for a module sitting on one side of a chip to talk to another module sitting on the opposite side. One way to design in such an environment is to design modules in a delay-tolerant fashion and insert latches so that computations can be pipelined.

  V++ facilitates such a design style by automating the synthesis of inter-module communication/arbitration devices.

- IP (Intellectual Property) exchange – Block-based design is a new design methodology where systems are integrated onto a single chip. Systems design houses devise chips by gluing intellectual properties (IPs, e.g., MPEG decoders, microprocessors, bus interfaces, peripheral controllers, etc.) from various vendors. Due to the increasing number of devices that can integrated onto a single chip, the cutting edge of such designs has high complexity and short design cycle.

  By using a simple inter-object communication protocol, V++ facilitates compositionality (when two well-defined, deterministic objects are glued together, the resulting design is guaranteed to be well-defined and deterministic) objects/IPs. It removes mysterious combinational cycles that are formed when composing objects. It gets rid of many tricky inter-object timing bugs. In addition, it relieves synthesis and cycle-simulation engines from the expensive, time-consuming global analysis at the chip level.

Bibliography

[ABB+94] Adnan Aziz, Felice Balarin, Robert K. Brayton, Szu-Tsung Cheng, Ramin Hojati, Sriram C. Krishnan, Rajeev K. Ranjan, Alberto L. Sangiovanni-Vincentelli, Thomas R. Shiple, Timothy Kam Vigyan Singhal, Serdar Tasiran, and Huey-Yih Wang. HSIS: A BDD-based environment for formal verification. In *DAC94*, San Diego, CA, June 1994.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Massachusetts, 1985.

[Bak93] Wendell C. Baker. An application of the synchronous/reactive semantics to the vhdl language. Memorandum UCB/ERL M93/10, University of California at Berkeley, 1993.

[Bak96] Wendell C. Baker. *Interfacing System Description Languages to Formal Verification.* PhD thesis, University of California at Berkeley, 1996.

[BBB+94] B. Barringer, T. Burd, F. Burghardt, A. Burstein, A. Chandrakasan, R. Doering, S. Narayanaswamy, T. Pering, B. Richards, T. Truman, J. Rabaey, and R. Brodersen. Infopad: A system design for portable multimedia access. In *Proceedings of Calgary Wireless 94 Conference*, 1994.

[BC84] G. Berry and L. Cosserat. The synchronous programming languages esterel and its mathematical semantics. In S. Brookes and C. Winskel, editors, *Seminar on Concurrency.* Springer Verlag Lecture Notes in Computer Science 197, 1984.

[BCH+91] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J.

Singh, and H.-Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Memorandum UCB/ERL M91/97, University of California at Berkeley, 1991.

[BdS91] F. Boussinot and R. de Sinone. The esterel language. In *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*. IEEE, September 1991.

[Ber92] G. Berry. A hardware implementation of pure esterel. In *Sadhana, Academy Proceedings in Engineering Sciences*. Indian Academy of Sciences, 1992.

[Ber96] G. Berry. The constructive semantics of pure esterel, draft version 2.0. 1996.

[BFK94] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. Proof theory and a validation condition generator for vhdl. In *Proceedings of Euro-DAC'94 with Euro'VHDL'94*, 1994.

[BGM94] Egon Borger, Uwe Glasser, and Wolfgang Muller. The semantics of behavioral vhdl'93 descriptions. In *Proceedings of Euro-DAC'94 with Euro'VHDL'94*, 1994.

[BHSv+96a] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Quadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. shiple, and Gitanjali Swamy. vis. In *FMCAD'96*, 1996.

[BHSV+96b] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In *Proceedings of the Conference on Computer Aided Verification*, July 1996.

[CB94] Szu-Tsung Cheng and Robert K. Brayton. Compiling verilog into automata. Memorandum UCB/ERL M94/37, University of California at Berkeley, 1994.

[CB96] Szu-Tsung Cheng and Robert K. Brayton. Synthesizing multi-phase hdl programs. In *Proceedings of International Verilog Conference*. IEEE, March 1996.

[CBY+95] Szu-Tsung Cheng, Robert K. Brayton, Gary York, Katherine Yelick, and Alexander Saldanha. Compiling verilog into timed finite state machines. In *Proceedings of International Verilog Conference*. IEEE, March 1995.

[CMM+96] Szu-Tsung Cheng, Patrick G. McGeer, Kennith L McMillan, Alexander Saldanha, Alberto Sangiovanni-Vincentelli, and Patrick Scaglia. Cycle simulation semanitcs of verilog and smv: Cobra from concept to simulator. In *Proceedings of Cadence Technical Conference*, 1996.

[CMM+97] Szu-Tsung Cheng, Patrick C. McGeer, Mike Meyer, Alberto Sangiovanni-Vincentelli, and Patrick Scaglia. The v++ systems design language. In *Proceedings of Cadence Technical Conference*, 1997.

[CMM+98] Szu-Tsung Cheng, Patrick C. McGeer, Mike Meyer, Tom Truman, Alberto Sangiovanni-Vincentelli, and Patrict Scaglia. The v++ system design language. In *The Proceedings of Design, Automation and Test in Europe*, 1998.

[CMS+98] Szu-Tsung Cheng, Patrick C. McGeer, Patrick Scaglia, Alberto Sangiovanni-Vincentelli, and Robert K. Brayton. Finite state machine communication in v++. Accepted by CTC'98, 1998.

[CMSS98] Szu-Tsung Cheng, Patrick McGeer, Fabio Somenzi, and Patrick Scaglia. Object-oriented simulation of hardware descriptions: Solving the namespace problem in a pure hardware language. Accepted by CTC'98, 1998.

[CSM+97] Szu-Tsung Cheng, Alexander Saldanha, Patrick C. McGeer, Patrick Scaglia, and Robert K. Brayton. A functional interpretation of verilog programs, 1997.

[DH89] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. In *IEEE Transactions on Computer-Aided Design*. IEEE, July 1989.

[Doh94] Gert Dohmen. Petri nets as intermediate representation between vhdl and symbolic transition systems. 1994.

[Eve94] Hans Eveking. (v)hdl-based verification of heterogeneous synchronous/ asynchronous systems. In *Proceedings of Euro-DAC'94 with Euro'VHDL'94*, 1994.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler.* The Benjamin/Cummings Publishing Company, Menlo Park, California, 1988.

[Gos96] James Gosling. *The Java Language Specificatioin.* Addison-Wesley Publishing Company, Menlo Park, CA, 1996.

[GS96] Brent L. Gregory and Russell B. Segal. Method for converting a hardware description of a logic circuit into hardware components. *United States Patent*, (5530841), June 1996.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Sytems.* Kluwer Academic Publishers, Norwell, MA, 1993.

[Har87] David Harel. *Science of Computer Programming*, volume 8, chapter Statecharts: A Visual Formalism For Complex Systems. Elsevier Science Publishers B.V. (North-Holland), 1987.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[Mal93] S. Malik. Analysis of cyclic combinational circuits. In *Proceedings of ICCAD*, 1993.

[MB91] Patrick C. McGeer and Robert K. Brayton. *Integrating Function and Temporal Domains in Logic Design: The False Path Problem and Its Implications.* 1991.

[McG89] Patrick C. McGeer. *On the Interaction of Functional and Timing Behavior of Combinational Logic Circuits.* PhD thesis, University of California at Berkeley, 1989.

[McM94] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1994.

[MMS+95] Patrick C. McGeer, Kenneth L McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *Proceedings of the 1995 ACM/IEEE International Workshop on Logic Synthesis*, 1995.

[Sea92] Ellen M. Sentovich and et. al. Sequential circuit design using synthesis and optimization. In *Proceedings of IEEE ICCD*. IEEE, October 1992.

[Shi96] Thomas Robert Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, 1996.

[SSL$^+$92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Memorandum UCB/ERL M92/41, University of California at Berkeley, University of California, Berkeley, May 1992.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Menlo Park, CA, 1991.

[syn94] *Synergy HDL Synthesizer and Optimizer Modeling Style Guild Version 2.0*. Cadence Design Systems, Inc., 1994.

[TM91] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[TPDB98] Tom Truman, T. Pering, Robert Doering, and Rober Brodersen. A port device for wireless information access. In *IEEE Transactions on Computers*. IEEE, 1998.

[Ulr78] Ernst G. Ulrich. Event manipulation for discrete simulations requiring large number of events. In *Communications of the ACM*, September 1978.

[VD75] Jean G. Vaucher and Pierre Duval. A comparison of simulatioin event list algorithms. In *Communications of the ACM*, 1975.

[ver95] *Verilog-XL Reference Manual, Version 2.3*. Cadence Design Systems, Inc., 1995.

[vhd88] *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., New York, 1988.

[vhd92] *Synopsys VHDL Compiler Reference Manual Version 3.0*. Synopsys, Inc., 1992.