Introduction to Digital Design

as

Cooperating Sequential Machines

Arvind [1]
Rishiyur S. Nikhil [2]
James C. Hoe [3]
Silvina Hanono Wachman [1]

With contributions from the Staff of MIT Courses 6.004, 6.175 and 6.375

[1]MIT CSAIL (Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Lab)
[2]Bluespec, Inc.
[3]Carnegie Mellon University, Electrical and Computer Engineering

DRAFT v1.0, September 5, 2019

Acknowledgements

We would like to thank the staff and students of various recent offerings of this course at MIT and University of California at San Diego for their feedback and support. Professor Rajesh Gupta provided the initial encouragement and the opportunity to start the development of this material for an introductory logic design class at UCSD. Professor Daniel Sanchez made it possible to use Clifford Young's Yosys Hardware synthesis tool in conjunction with BSV; without Yosys, the lab experience would have been significantly poorer. Over the years, many students have contributed enormously to the development of lab exercises and pedagogy. Special thanks to Abhinav Agarwal, Thomas Bourgeat, Chanwoo Chung, Nirav Dave, Kermin Elliott Fleming, Sang-Woo Jun, Asif Khan, Myron King, Ming Liu, Alfred Man Cheuk Ng, Michael Pellauer, Daniel Rosenband, Richard Uhler, Muralidharan Vijayaraghavan, Andrew Wright, Shuotao Xu, Sizhuo Zhang. We also very thankful to Bluespec Inc for providing free access to Bluespec tools and fixing bugs in a timely manner.

Contents

Chapter 1

Introduction: Why a new introductory book on Digital Design?

This chapter discusses the context and rationale for this book. Readers keen on starting on the technical content can skip to the next chapter.

The foundations of Digital Design, Boolean Logic and Finite State Machines (FSMs) have not changed in more than 75 years. Numerous textbooks cover these basic concepts adequately, so what is the justification for writing a new introductory book on this topic? The following sections discuss how the landscape has changed, requiring changes in focus, emphasis, scale and methodology.

1.1 Modern digital designs are extremely large and complex

Modern digital systems are the most complex artefacts ever built by man. Advances in semiconductor technology have made it possible, in 2019, to build single chips containing several tens of billions of transistors. Electronic systems, in turn, are built with tens to hundreds of chips. The only way human designers can manage and control artefacts of such staggering scale and complexity is through powerful abstraction mechanisms for expressing designs, and powerful design automation tools that render such designs into silicon (actual hardware).

In introductory logic design courses it is common to use small contrived examples to motivate sequential circuits, like a controller for a set of traffic lights or a clothes-washing machine. The worst aspects of such an approach is that it leads to a dead end; students do not know what do with this knowledge.

Today, it should be (and is) feasible for students in an introductory digital design class to design and build simple microprocessors, an idea which would have seemed totally insane 50 years ago.

Although the design of small microprocessors has appeared in introductory digital design courses in the last few decades, the methodology is still old school: we teach students first how to design the "datapath", and then how to orchestrate data on it by controlling various muxes and register enable signals. Most students have difficulty connecting the fundamental concepts of sequential machines to the resulting hardware of microprocessors.

1.1.1 Modularity and compositionality are required for large, complex, evolving systems

The best way for human designers to tackle large complex systems is using modularity and compositionality. Starting with small, basic, often single-function modules ("primitives"), one should be able to design a more interesting module with more complex behavior. Then, just based on a summary description of this module and forgetting internal details, we should be able to use it in an even more complex design, and also to reuse it across diverse complex designs. Or, to look at it in a top-down manner, modules are *composed* of lower-level modules which, in turn, are themselves composed of even lower-level modules, and so on. At any level, we should not have to know details of the lower levels, *i.e.,* we should be able to summarize, or *abstract* the properties of lower-level modules into just what is needed for the current level.

Current Hardware Design Languages (HDLs) such as Verilog, SystemVerilog and VHDL, and even modern HDLs like Chisel, do indeed have a "module" construct from which one builds hierarchically structured designs. The problem is that, while their modularity and compositionality is fairly strong regarding *structure* (how components are wired together), they are very weak on compositionality of *behavior* (how is the behavior of a module explained systematically as a composition of the behaviors of its sub-modules).

This is a fundamental problem with the older methods of teaching Sequential Machines, based on the mature theory of Finite State Machines (FSMs). A module in Verilog is a clocked synchronous FSM. When used in a parent module, the ensemble is also an FSM, just a larger FSM (this is a well known theoretical property of FSMs—composition of FSMs produces an FSM). Unfortunately, this is not a modular description of behavior—it is hard to reason about the outer FSM without "plugging in" the inner FSM to produce the larger FSM. The usual techniques for describing small FSMs, such as state transition diagrams, simply do not scale as we get larger and larger FSMs.

This is also a problem that affects *evolution* and *refinement* of designs. In real life, one frequently has to change a design—to fix bugs, to improve some metric like performance or area, or to add some previously unanticipated functionality. A change to a module may mean that its behavior now is a different FSM. This can have a wide-ranging ripple effect because, by definition of FSM composition, the FSM for any module that uses this changed module has also changed. There is no systematic way to help isolate these changes.

When module M1 uses a sub-module M2, it is often the case that these modules are designed by different people at different times. The designer of M1 does not want to, nor should they need to know about all the internal details of M2. How should the designer of M2 communicate a description of M2 to the designer of M1, that contains just enough and no more detail than is necessary? Describing the full FSM for M2 is too detailed, since it may contain a lot of internal state and have many internal state transitions, and in some sense effectively exposes the internal implementation. So, with HDLs today, designers resort

to diagrams of waveforms on the input and output wires of their module, accompanied by much English text to describe behavior. These descriptions are notoriously incomplete and error-prone, and written in a variety of inconsistent styles.

There is no systematic way to abstract or to summarize the behavior of the inner FSM and to just use the summary while considering the behavior of the outer FSM. This issue of compositionality of behavior is not discussed systematically in any text book today.

A better, more scalable and more tractable approach would be if large digital circuits are viewed as a composition of many conceptually separate sequential machines (each module a separate sequential machine), which cooperate with each other in well-behaved ways to deliver the required functionality. The cooperation should be entirely characterized by well-structured interfaces with well-structured interface behavior. We can make internal changes to a module as long as this interface behavior is preserved. Thus, we need a systematic vocabulary and a theory of how to characterize interface behaviors.

1.1.2 We need appropriate computation models

When studying computational machines in Computer Science, one encounters a sequence of increasing expressive power: FSMs, stack machines, and Turing Machines. A modern computer system, technically, is just an FSM since it has a finite number of states (billons of bits, but still finite). But it is neither insightful nor productive to think of it in those terms. It is much more useful to view it as a Turing Machine.

Similarly, although each of our hardware designs is still technically a single (often giant) FSM, it will be much more insightful and productive to view them as collections of cooperating sequential machines that can be independently designed and verified, with simple and well-defined cooperation protocols.

1.1.3 We need HDLs suitable for formal verification

By "formal verification" we mean automatic proofs of correctness of a system, guaranteeing that its behavior meets expectations. For example, in a traffic-light controller we would want to prove that it never gets into a state where it is displaying Green for two intersecting flows. For an elevator (lift) controller we may want to prove that it never gets into a state where the elevator is moving with its door open. One can similarly imagine correctness properties for self-driving cars, or trains or planes, or medical devices that control dosage or radiation exposure. As we use computers more and more to control physical artefacts, it becomes increasingly important to prove correctness.

Experience in the software field has shown that formal proofs of correctness are facilitated:

- by higher-level languages for design;
- by design language with cleaner semantics that have been created with formal verification in mind from the beginning;
- by design languages that support abstraction, modularity, and compositionality.

Experience also shows that clean, high-level design languages facilitate manual and automated debugging. Unfortunately most existing HDLs fail to meet these criteria.

1.2 Why software engineers should learn about hardware design

In the past, teaching software and hardware design has often been done in disjoint silos, often coinciding with the unfortunate separation of Computer Science and Electrical Engineering departments. This disjointness is becoming increasingly untenable.

In principle, any algorithm can be implemented either in software or in hardware, *i.e.,* it is *an implementation choice*. Software of course provides maximum flexibility and changeability, and hardware can have a high up-front cost, but a direct hardware implementation:

- is often one to three orders of magnitude faster (10x to 1,000x), due to much higher levels of parallelism, removal of layers of interpretation, and extensive specialization, and
- often uses two to four orders of magnitude less energy (100x to 10,000x), due to removal of layers of interpretations and extensive specialization.

This is becoming an increasingly important issue today.

In 2019 we are approaching (or have reached) the limits of Moore's Law (ability to double the number of transistors on a chip about every two years) and Dennard Scaling (power consumption stays proportional to area). Consequently, it has become critical to move more and more critical algorithms from software implementations into direct hardware implementations. Your cell phone likely has many such purposeful hardware modules which, if done in software, would have killed your battery in a few minutes. Much of modern graphics computation and neural-net AI computation is done with special hardware. As we move towards the "Internet of Things", many devices will require direct hardware assists to stay within very limited power budgets. Modern Supercomputing (High Performance Computing) would not be possible without hardware accelerators. In other words, many major software applications have an increasing reliance on hardware assists.

Software engineers may view hardware design as the activity of designing chips, a daunting task, totally outside their domain of expertise or interest. However, in 2019, we now have Field Programmable Gate Arrays (FPGAs) using which we can create a hardware implementation in a few hours in the comfort of one's own office. Amazon already provides cloud-based servers that have attached FPGAs boards that software programmers are using today on a daily basis. Intel is creating new CPUs with FPGAs tightly integrated into x86 CPUs (for which they recently acquired Altera, one of the major FPGA vendors).

Thus, a modern software application designer needs to consider speed and power budgets, and whether it is productive to move parts of the application into direct hardware implementation, using FPGAs or custom chips. The software application designer needs to be concerned about the architecture of such hardware-software ensembles, and in particular how software and hardware components cooperate to solve a problem. Technically, we say that system design is evolving towards *hardware-software codesign*.

This course provides a minimum background in hardware design for the software engineer to address these modern challenges.[1]

[1] Ironically, the latest revision of ACM 2013 CS education curriculum guidelines devotes exactly zero hours of training in computer hardware, while making it an entirely upper-division subject!

1.3 Is so-called "High-Level Synthesis" (HLS) the answer?

In the last decade, a radically different approach to hardware design called High-Level Synthesis (HLS) has gained much attention. Here, one specifies an algorithm in C or C++, and an HLS tool converts that into implementation in hardware. One can think of it as an alternative compiler which, instead of targeting machine code for some instruction set like x86 or ARM, instead targets hardware logic gates and flipflops.

The commercial appeal of HLS is undeniable, especially for designing hardware accelerators for signal-processing algorithms. If the same language can be used for both hardware design and software implementations, it should reduce the cost of system development. Software engineers may find HLS more approachable because one codes in C or C++.

The reality is more complicated: the style of C one writes for hardware synthesis is quite different compared to the style of C one writes for software programming. Unless programmers have a good hardware model in their head, and know how to use the extra-lingual annotations provided by the HLS tool to express parallelism, their code is unlikely to produce useful hardware.

In general, HLS does reasonably well for algorithms that can be expressed as for-loops over dense rectangular arrays, but not for much else. The central problem is that complex hardware requires sophisticated understanding of hardware microarchitectures, and HLS tools are not good for specifying this.

1.4 The new approach in this book

In this book we teach digital design with a new approach addressing all the concerns listed above.

Modularity and compositionality are addressed using a new semantic model for hardware, that of cooperating sequential machines. Each module is a sequential machine, and a collection of modules cooperate through well-defined and well-structured behavioral interfaces. At the heart of behavioral descriptions are *Guarded Atomic Actions*. These semantics and the associated methodology scales well, from small designs (like traffic-light controllers) to very large designs (like the most advance high-performance microprocessors and hardware accelerators for neural networks and signal-processing algorithms).

Our vehicle for the exposition is BSV, a high-level hardware design language (HLHDL) that supports this way of thinking directly in the language semantics.[2] BSV incorporates ideas on types and abstraction and behavior from advanced software programming languages, notably:

- types, functions and functional programming from the modern functional programming language Haskell,
- object-orientation from object-oriented languages, and

[2]High-level semantics can always, in principle, be *coded* in a lower-level language by adopting a suitable coding discipline and style, but this is not really practical at scale. One could do object-oriented programming in C or Assembly Language, but it is vastly more convenient, understandable, changeable, correct and automated in C++. Similarly, the semantic concepts in BSV can, in principle, be manually coded in an existing HDL, but it is not practical at scale.

- atomic transactions from multithreaded and distributed programming languages.

BSV's module hierarchy mechanism, and its purely method-based communication between modules should be very familiar to software engineers who know object-oriented programming. Objects are connected to each other by atomic rules, where a rule is simply a composition of method calls. A rule must be executed *atomically*, that is, either each of its method calls is executed or none of them is executed. All the rules are active all the time, however, if two rules *conflict* then only a non-conflicting subset of rules can be executed, ie affect the state. This idea of atomic transactions should be familiar to software engineers who have had exposure to distributed systems, multithreaded or multiprocessing programs, or operating systems. In fact, atomicity of BSV rules and the methods they invoke in many ways makes the treatment of concurrency more powerful, and easier to use, than in many object-oriented software languages.

A compiler for BSV, *bsc*, translates our designs into lower-level hardware primitives in the older, more low-level HDL Verilog.[3] In BSV a rule execution is compiled into Verilog code that always completes in one clock cycle and after the execution of a rule or rules the state of the system changes. Just like in a synchronous sequential machine, the state of system changes only at clock edges. This generated Verilog can be simulated on a workstation (using a any Verilog simulation tool such as Verilator), or processed further by other tools into gates and transistors and ultimately into FPGAs or ASIC silicon.

In addition to BSV, the book relies on Yosys, a hardware synthesis tool that takes *bsc*'s output Verilog into gates, so the student can develop insight into the actual hardware cost of a design (measured in gates and time delay). Without such a tool it would be difficult to quantify the advantage of one hardware design over another.

BSV hardware descriptions using modules with guarded interfaces and atomic rules are significantly higher level or more abstract than typical hardware descriptions written in Verilog, and yet the synthesized hardware from such description is generally as efficient as hand coded RTL.

1.5 Traditional topics deliberately dropped

Teaching software design used to begin with Assembly Language programming, a long time ago, but we have long since moved away from that. This is not to say that Assembly Language programming is no longer important, just that it is no longer central to the theory and practice of modern sofware system design. Nowadays tools (compilers) manage the generation of Assembly code, calling conventions, register optimization, *etc.* Specialists who build those tools of course still study Assembly Language programming.

Similarly, in teaching hardware design, it is time to move away from obsessing over Truth Tables, State Transition Diagrams, Karnaugh Maps, state-machine minimization, logic optimization, and so on. Besides, logic optimization today is a complex multi-axis problem, looking at area, speed, power, latency, throughput, and other considerations; it requires a lot of mathematical sophistication, and it is too tedious and unproductive to do these by hand. Today, tools do these jobs for us, and specialists who build those tools still

[3] Conceptually, Verilog is *bsc*'s "Assembly Language", by analogy to software compilers that compile C/C++ to Assembly Language.

need to study those topics. The modern hardware designer needs to have general sense of what these tools do, but need not be an expert.

We also drop any discussion of semiconductors and semiconductor technology (*e.g.,* CMOS). Today, most modern hardware designers assume higher-level building blocks such as gates, flipflops, memory cells and so on. FPGA designers work with LUTs (lookup tables), flipflops, BRAMs and DSPs as primitives. ASIC designers uses vendor-supplied standard-cell libraries and memory compilers. It is important for a designer to understand the area-delay-power properties of his or her design but this does not require understanding how primitive design elements are implemented in CMOS. A modern digital designer deals with designs that require hundreds of thousands, if not millions, of gates to implement and can't afford to focus on the implementation of individual gates. Fortunately, hardware synthesis tools do all the low-level work for us.

1.6 Audience for this book, and their possible futures

This book has been written for MIT's introductory hardware design course (6.004), which teaches both digital design and computer architecture. This book is intended for the first half (digital design). 6.004 also has a significant laboratory component to cement the ideas in this book. Students create designs in BSV and execute them initially in simulation but then also on off-the-shelf FPGA boards.

At MIT, this course is required for all students of Electrical Engineering and Computer Science. In both cases, it assumes no hardware background in circuits or digital electronics. It is not surprising that EE majors take this course, and they may go on to deeper studies of computer architecture, hardware synthesis, VLSI, arithmetic circuits, communication or storage hardware, hardware systems, and so on.

For many CS majors, this may be the only course they take on hardware design, but they go into software fields with a solid understanding of the tradeoffs between implementing algorithms in software versus hardware, and are armed with the knowledge of how and when to exploit hardware implementations. As argued earlier, today it is very important even for software engineers to have this knowledge and perspective.

This book may be useful at other universities with similar curriculum goals.

This book will also be useful for those who already know or are experienced in hardware design using some other HDL such as Verilog, SystemVerilog, VHDL, or Chisel, or even HLS, and are curious about BSV and our approach. These readers may wish to start at Chapter 6, skip Chapter 7, and read the remaining chapters.

Chapter 2

Boolean Algebra

In elementary algebra, the values of the variables are numbers, and the primary operations are addition and multiplication. In Boolean algebra, the values of the variables are 1 (true) and 0 (false) and the main operations are conjunction ($\cdot$), disjunction ($+$), and negation ($\sim$). Boolean algebra has been fundamental in the development of digital electronics and is supported by all modern programming languages. It is also used in set theory, logic, and statistics.

2.1 Boolean Operators

2.1.1 Basic Boolean Operators

The basic operations of Boolean algebra are:

- `AND` (conjunction)
- `OR` (disjunction)
- `NOT` (negation)

Multiple notations can be used to specify Boolean operations. The three primary notations are:

1. Truth tables
2. Pictorial representations, also known as gates
3. Mathematical expressions

Truth tables enumerate all input combinations and provide the resulting output for each set of inputs. The truth tables of Table 2.1 show that the `AND` of two Boolean variables returns true only when both inputs are true and false otherwise. The `OR` of two Boolean variables returns true if at least one of the inputs is true and false if both inputs are false. The `NOT` of a Boolean variable returns true if the input was false, and false if the input was true.

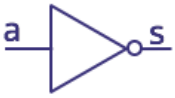
Operator	Truth Table	Pictorial Form	Mathematical Expression
AND	a b | s 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1		$a \cdot b$
OR	a b | s 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1		$a+b$
NOT	a | s 0 | 1 1 | 0		$\sim a$

Table 2.1: Definitions of AND, OR, and NOT Boolean Operators

A Mathematical, or Boolean, expression is built up from variables and the constants 0 and 1, using the operations $\cdot$, $+$, and $\sim$.

Any complex Boolean expression, also known as a Boolean function, can be described using a truth table or pictorially, as an acyclic interconnection of Boolean operators.

2.1.2 NAND, NOR, and XOR Boolean Operators

`NAND`, `NOR`, and `XOR` are additional Boolean operators that are frequently used. Their corresponding truth table, gate, and Boolean expressions are shown in Table 2.2.

The `XOR` truth table describes the behavior of the `XOR` operator which is that it produces a 1 when either (a=0) and (b=1), or (a=1) and (b=0). Hence, $a \oplus b = \sim a \cdot b + a \cdot \sim b$.

All complex gates, including the `NAND`, `NOR`, and `XOR` gates can be expressed using only `AND`, `OR`, and `NOT` gates as shown in Table 2.3. The `NAND` and `NOR` are straightforward to see. The `XOR` operator is defined to be true when either (a=0) and (b=1) which is covered by the upper `AND` gate, or when (a=1) and (b=0) which is covered by the lower `AND` gate. Taking the `OR` of the two `AND` gates thus results in the functionality of an `XOR` operator.

2.1.3 Universal Set of Operators/Gates

A set of operators or gates is considered *universal* if any boolean function can be describes using only those gates. Some examples of universal sets are: {`AND`, `OR`, `NOT`}, {`NAND`},

Operator	Truth Table	Pictorial Form	Mathematical Expression
NAND	a b s 0 0 1 0 1 1 1 0 1 1 1 0		$\sim(a \cdot b)$
NOR	a b s 0 0 1 0 1 0 1 0 0 1 1 0		$\sim(a+b)$
XOR	a b s 0 0 0 0 1 1 1 0 1 1 1 0		$a \oplus b$

Table 2.2: Definitions of NAND, NOR, and XOR Boolean Operators

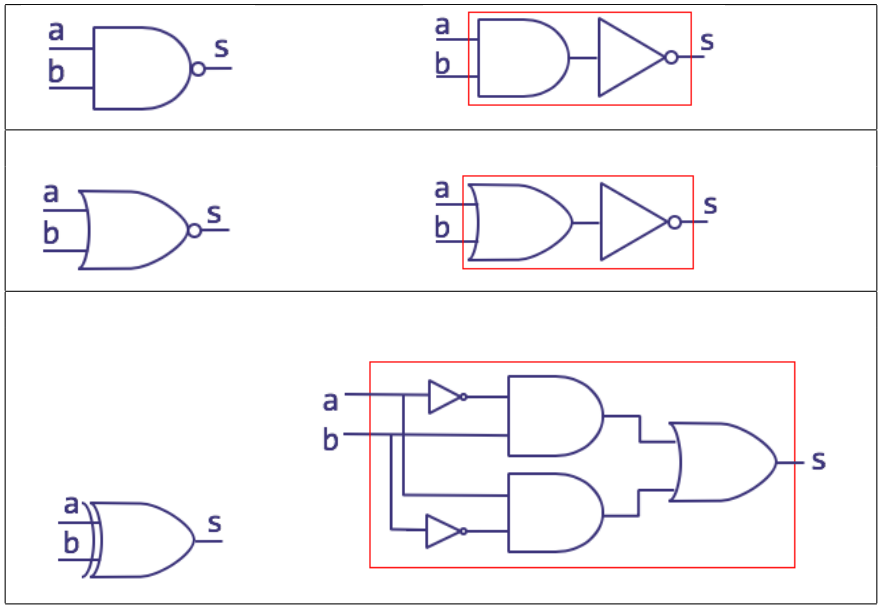


Table 2.3: NAND, NOR, and XOR using basic gates

and {NOR}. Section 2.2.1 described how to convert any truth table to a sum of products expression. By definition, a sum of products expression consists of only the AND, OR, and NOT operators, thus proving that the set AND, OR, NOT is universal.

Figure 2.1 provides a pictorial proof that using only a `NAND`, or only a `NOR`, operator, one can produce `NOT`, `AND`, and `OR` operators. Therefore, the sets `NAND` and `NOR` are universal as well.

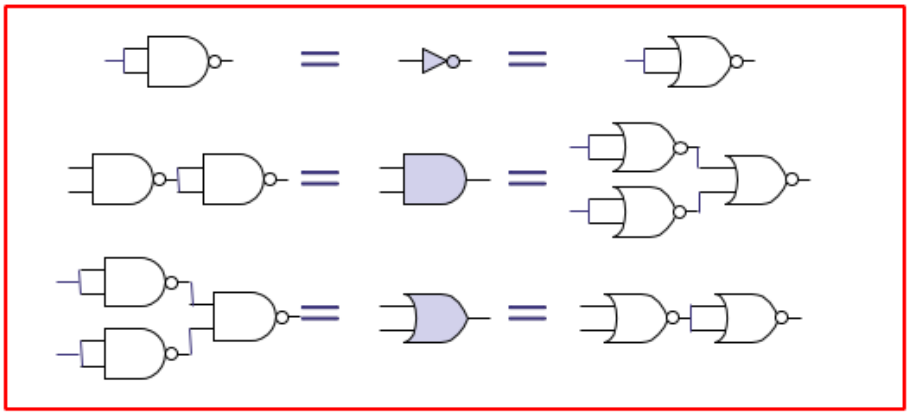


Figure 2.1: Universal Sets of Operators

2.2 Truth Tables

Truth tables can be used to define any Boolean function. For example, consider a function of three inputs `a`, `b`, and `c` and single output `f`, as shown in Figure 2.4. The truth table of Table 2.5 defines this function by specifying the value of the output for every possible input combination.

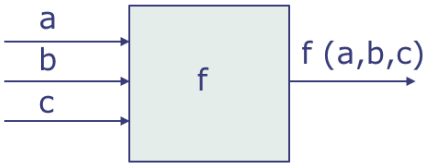


Table 2.4: Function of 3 inputs

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Table 2.5: Truth table for sample function

2.2.1 Truth Table to Boolean Expression

A sum-of-products (SOP) boolean expression can be generated for any truth table by writing a product term using only `AND`s and `NOT`s for each row in which the output is a 1, and then writing the sum of all such product terms.

For our sample truth table, this would lead to the Boolean expression:

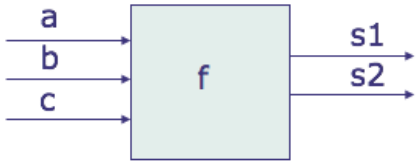


Figure 2.2: Two Output Function

f = (∼a)·(∼b)·(∼c) +a·(∼b)·(∼c) +a·(∼b)·c

An SOP expression generated this way is unique and is said to be the SOP normal form.

2.2.2 Boolean Expression to Truth Table

There is a unique truth table corresponding to every Boolean expression. To produce the truth table, enumerate all the combinations of input values, then evaluate the expression on each set of input values and record the answer in the output column of the truth table.

For example, the Boolean expression:

f = (∼b)·(∼c+a)

returns 1 when both (∼b) and (∼c+a) evaluate to 1. This occurs when b = 0 and either c = 0 or a = 1. Thus, this Boolean expression produces the same truth table as the one in Table 2.5.

2.2.3 Multi-output Truth Table

Sometimes, we specify multiple outputs for each set of input values as shown by the block diagram of Figure 2.2.

To avoid writing a new truth table for each output, a truth table can have multiple output columns as shown in Table 2.6. s1 and s2 are both outputs of the three inputs a, b, and c.

a	b	c	s1	s2
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	0

Table 2.6: Sample truth table with 3 inputs and 2 outputs

2.3 Laws Of Boolean Algebra

2.3.1 Boolean Algebra Axioms

identity	$a \cdot 1 = a$	$a + 0 = a$
null	$a \cdot 0 = 0$	$a + 1 = 1$
negation	$\sim 0 = 1$	$\sim 1 = 0$

Table 2.7: Boolean Algebra Axioms

Table 2.7 lists the three basic Boolean Algebra axioms known as *identity*, *null*, and *negation*. The *identity* axiom shows that if you `AND` any variable with a 1, you get back the variable. Similarly, if you `OR` any variable with a 0, you get back the variable itself. On the other hand, the *null* axiom shows that if you `AND` any variable with a 0, that will always return 0. Similarly, if you OR any variable with a 1 that will always return a 1. Finally, the *negation* axiom says that negating a 0 produces and 1 and negating a 1 produces a 0.

These axioms can be derived directly from the definitions of the `AND`, `OR`, and `NOT` Boolean functions.

2.3.2 Laws of Boolean Algebra

Table 2.8 provides a full list of laws that apply to Boolean Algebra. These laws can be derived from the definition of the `AND`, `OR`, and `NOT` functions and their axioms. They can be used to simplify complex Boolean expressions into simpler ones.

identity	a·1 = a	a+0 = a
null	a·0 = 0	a+1 = 1
negation	(∼0) = 1	(∼1) = 0
commutative	a·b = b·a	a+b = b+a
associative	a·(b·c) = (a·b)·c	a+(b+c) = (a+b)+c
distributive	a·(b+c) = (a·b)+(a·c)	a+(b·c) = (a+b)·(a+c)
complement	a·(∼a) = 0	a+(∼a) = 1
absorption	a·(a+b) = a	a+(a·b) = a
reduction	(a·b)+(a·(∼b)) = a	(a+b)·(a+(∼b)) = a
DeMorgan's Law	∼(a·b) = (∼a)+(∼b)	∼(a+b) = (∼a)·(∼b)

Table 2.8: Laws of Boolean Algebra

The *commutative* and *associative* laws are straightforward and are analogous to their mathematical counterparts. The next Boolean property states that the *distributive* property holds for `a·(b+c)` as well as `a+b·c`. Note that the latter holds true for Boolean algebra but not for integer algebra.

The *complement* property is very useful in simplifying complex Boolean expressions. It states that taking the `AND` of `a` and ∼`a` always results in 0 or false, whereas taking the `OR` of `a` and ∼`a` always results in 1 or true.

Absorption and *reduction* enable further simplification of complex Boolean expressions by noting that the value of `b` does not affect the value of the entire expression. The

absorption property states that a·(a+b) = a because if a = 0 the left product term (a) is 0 which ANDed with anything returns 0, whereas if a = 1 the right product term (a+b) is 1 making the entire expression equal to a which is 1. So the entire expression can be simplified into a. The converse of the *absorption* property where AND is replaced with OR and vice-versa can be explained in the same manner. The *reduction* property states that (a·b)+(a·∼b) = a because a appears in conjunction with both b and ∼b. Once again the same holds true for the converse of this statement.

The last law is *DeMorgan's Law* which is most easily understood by considering all input and output combinations for the left hand side and the right hand side of the expression and noting that they are always equal. For example, if both a=1 and b=1, then the left hand side is ∼(a·b) which is 0, and the right hand side is (∼a)+(∼b) which is also 0.

2.3.3 Principle of Duality

The dual of a Boolean expression is obtained by swapping all 0's and 1's as well as all ANDs and ORs. The principle of duality states that if expressions e1 and e2 are equal, then the dual of these expressions is also equal. In other words, (e1 = e2) implies (dual(e1) = dual(e2)).

Thus, for example, if for all a, a·1 = a then a+0 = a as well. Similarly, if for all a, a·(∼a) = 0 then a+(∼a) = 1. Note that the NOT operator is not replaced when producing the dual of the expression.

The two columns in the table of Boolean algebra laws, Table 2.8, are duals of each other. Hence, one only needs to remember one of the columns and then produce the other using the duality principle.

2.4 Boolean Simplification

The laws of Boolean algebra can be divided into two categories: those that reduce Boolean expressions and those that rearrange Boolean expressions. Table 2.9, indicates which rules can be used to reduce complex Boolean expressions into simpler ones as well as which rules can be used to rearrange the Boolean expression so that it may become easier to then apply one of the reduction laws.

2.4.1 Minimal Sum Of Products

A **minimal sum-of-products** is a sum-of-products expression that has the smallest possible number of AND and OR operators. Below is an example of using the reduction law to convert a complex sum-of-products (SOP) expression into a minimal SOP.

Unlike the normal form, a minimal SOP is not necessarily unique (i.e., a function may have multiple minimal SOPs).

Reduce	**identity**	a·1 = a	a+0 = a
	null	a·0 = 0	a+1 = 1
	negation	(∼0) = 1	(∼1) = 0
	complement	a·(∼a) = 0	a+(∼a) = 1
	absorption	a·(a+b) = a	a+(a·b) = a
	reduction	(a·b)+(a·(∼b)) = a	(a+b)·(a+(∼b)) = a

Rearrange	**commutative**	a·b = b·a	a+b = b+a
	associative	a·(b·c) = (a·b)·c	a+(b+c) = (a+b)+c
	distributive	a·(b+c) = (a·b)+(a·c)	a+(b·c) = (a+b)·(a+c)
	DeMorgan's Law	∼(a·b) = (∼a)+(∼b)	∼(a+b) = (∼a)·(∼b)

Table 2.9: Laws of Boolean Algebra

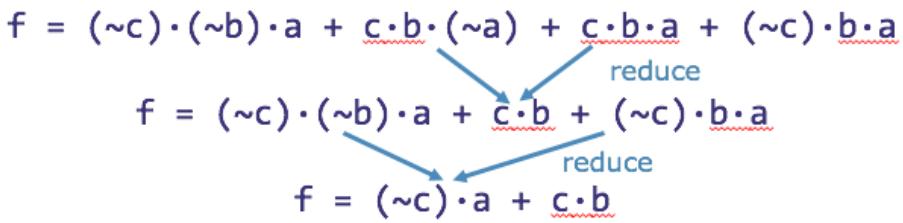


Figure 2.3: Reducing complex SOP expression to a minimal SOP

2.4.2 Common Subexpression Optimization (CSE)

We can often reduce the number of operators/gates by factoring out common subexpressions. For example, consider the minimal SOP

F = A·C +B·C +A·D +B·D

As is, it requires 4 `AND` operators and 3 `OR` operators. It can be rewritten as the following Boolean expression:

F = (A+B)·C +(A+B)·D

The common term (A+B) can be replaced with the variable X to produce

F = X·C +X·D where X = A +B

This new expression requires 2 `AND` operators and 2 `OR` operators. Pictorially, this new representation of the Boolean expression F is shown in Figure 2.4. Thus by using the common subexpression optimization the complexity of the Boolean expression has been reduced significantly. Note that using the common subexpression optimization turns a boolean expression from a tree into a directed acyclic graph.

2.4.3 Truth Table With Don't Cares

Another way to reveal simplification is to rewrite the truth table using *don't cares* (– or X) to indicate when the value of a particular input is irrelevant in determining the value of the

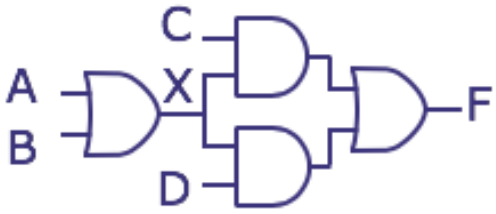


Figure 2.4: Common subexpression optimization of Boolean expression

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 2.10: Fully specified truth table

a	b	c	y	product term
0	X	0	0	
1	X	0	1	→ (∼C)·A
0	0	X	0	
1	1	X	1	→ C·B
X	0	1	0	
X	1	1	1	→ B·A

Table 2.11: Truth table with don't cares

output.

For example, consider the truth table of Table 2.10. Table 2.11 is an alternate representation for this same table using *don't cares*. For example, both input 000 and 010 produce a 0 output, thus both of the rows corresponding to these input combinations can be combined into a single row that includes a *don't care*. From this representation, it is more straightforward to see that the terms corresponding to inputs 100 and 110, or product terms A·(∼B)·(∼C) and A·B·(∼C), can be reduced to the simpler product term (∼C)·A.

Note that some input combinations (e.g., 000) are matched by more than one row in the *don't care* truth table. For a well-defined truth table, all matching rows must specify the same output value!

2.5 Satisfiability

A complex Boolean expression is said to be **satisfiable** if there exist some input combination that produces a true output. If no such input combination exists, then the expression is unsatisfiable. For example, consider the expression a·(∼a). Regardless of the value of a, this expression cannot produce a true output. So this expression is considered unsatisfiable. On the other, the expression (a +∼b)·c ·b is satisfiable by the input 111.

2.6 Take Home Problems

Take Home Problem 1

Write down a sum-of-products (SOP) boolean expression in normal form for the following truth table.

a	b	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Table 2.12: Take Home Problem 1

Take Home Problem 1 Solution

$s = (\sim a)\cdot b\cdot c + a\cdot(\sim b)\cdot(\sim c) + a\cdot b\cdot(\sim c)$

Take Home Problem 2

Generate the truth table corresponding to the following boolean expression.

$s = (\sim a)\cdot b\cdot c + a\cdot(\sim c)$

Take Home Problem 2 Solution

a	b	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Table 2.13: Take Home Problem 2 Solution

More Take Home Problems

- Boolean simplification

- Boolean equivalence checking
- Satisfiable expressions
- Universal sets of operators

Chapter 3

Binary Number Systems

We are all familiar with the representation of numbers in base-10 from daily life experience and mathematics. We also learn basic arithmetic operations like addition, subtraction and multiplication on numbers represented in base-10 notation. In the field of computer science, numbers are represented in binary, or base-2 representation, so that they can be encoded using only 0's and 1's. Basic arithmetic operations are also carried out directly in binary without numbers ever being converted back to base-10 representation. We will describe binary arithmetic in this chapter.

3.1 Encoding Positive Integers in Binary

To interpret the value of a decimal number, such as 132, we treat the right most bit as the 1's (or 10^0) value, the next bit to the left as the 10's (or 10^1) value, and the left bit as 100's (or 10^2). This leads to the interpretation of the number as 1 hundred plus 3 tens plus 2 ones or 132. To determine the value of a positive binary number, we use the same methodology replacing the base 10 with the base 2. So the binary number 110 is equal to $0 * (2^0) + 1 * (2^1) + 1 * (2^2) = 0 + 2 + 4 = 6$ in decimal. To avoid confusion, we prepend binary numbers with the prefix `0b` to indicate that the number represents a binary number rather than a decimal number.

More generally, in a binary representation (in right-to-left order), bit i is assigned weight 2^i. The value of an N-bit number is given by the formula $v = \sum_{i=0}^{N-1} 2^i b_i$. Thus, an N-bit string of 0's represents 0 while an N-bit string of 1's represents $2^N - 1$ which is the maximal value representable as an N-bit binary number.

Often we diagram such a representation, as shown in Figure 3.1, as an array of binary digits whose right-most bit b_0 is the least significant digit. This example diagrams a 12-bit binary representation of the (decimal) value 2000, as may be verified via the formula above. $V = 0 * (2^{11}) + 1 * (2^{10}) + 1 * (2^9) + ... = 1024 + 512 + 256 + 128 + 64 + 16 = 2000$.

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string. A popular choice is to use base-16, called *hexadecimal*. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

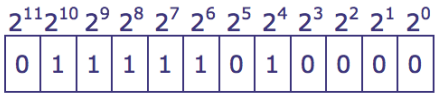


Figure 3.1: Encoding Positive Integers

Hexadecimal - base 16	
0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - A
0011 - 3	1011 - B
0100 - 4	1100 - C
0101 - 5	1101 - D
0110 - 6	1110 - E
0111 - 7	1111 - F

Table 3.1: Hexadecimal Values

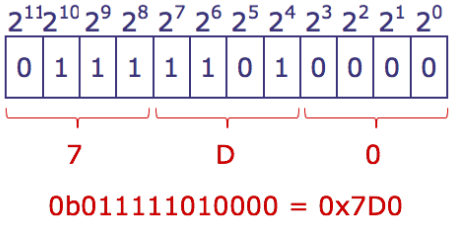


Table 3.2: Converting Binary to Hexadecimal

Figure 3.1 shows the mapping of 4-bit binary numbers into hexadecimal characters which range from 0-9 followed by the letters A-F in upper or lower case. Thus, the binary number of Figure 3.1 would be encoded as `0x7D0` in hexadecimal as illustrated in Figure 3.2. Note that in order to distinguish between binary and hexadecimal numbers in all cases, binary numbers are prepended with 0b and hexadecimal numbers are prepended with 0x. If the number is not prepended, then one assumes that it is using binary representation.

3.2 Binary Addition and Subtraction

Addition and subtraction in base 2 are performed just like in base 10 as illustrated by the example in Figure 3.2. Just as in base 10 addition adding 4 to 7 results in a 1 in the sum and a carry of 1 into the next position, similarly adding 1 to 1 in binary results in a sum of 0 and a carry of 1. When subtracting 7 from 4, we need to borrow 10 from the next column over, and when subtracting 1 from 0, we need to borrow a 2 from the next column over. In both cases, this results in decrementing the next column by 1 to account for this borrow.

A problem occurs when we try to subtract a larger number from a smaller number as shown in Figure 3.3. Initially, our procedure works for the first two columns, but when we attempt to subtract the third column, 0 minus 1 results in 1 after borrowing from the next column over. However, in this example there is no next column over so it is unclear where we are borrowing from. The problem occurs because presently, we don't yet have a way of representing negative binary numbers.

3.3 Encoding Negative Binary Numbers

3.3.1 Binary Modular Arithmetic

If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition). This is known as

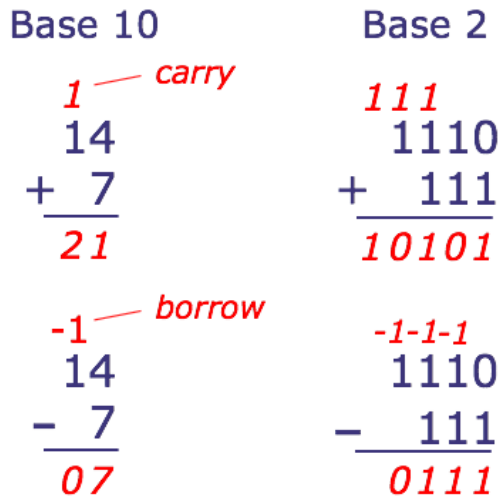


Figure 3.2: Binary Addition and Subtraction

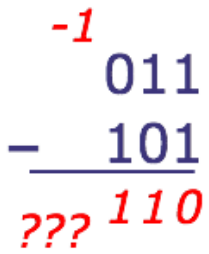


Figure 3.3: Attempt to Subtract Five from Three

overflow. A common approach for dealing with overflow is to ignore the extra bit. Doing this gives rise to *modular arithmetic* where using N-bit numbers, the resulting value is equivalent to the operation followed by `mod` 2^N. Visually, one can think of this as wrapping the numbers around as shown in Figure 3.3 where addition is performed by moving clockwise around the circle and subtraction is performed by moving counter-clockwise. Thus, if you have a 3-bit positive binary number, the smallest number is 0b000 (0) and the largest is 0b111 (7). Adding 1 to 7 results in wrapping back around to 0b000 because $(1 + 7)$ `mod` $2^N = 0$.

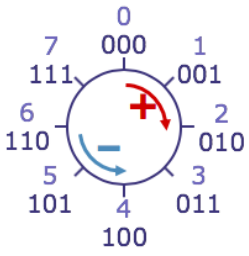


Table 3.3: Binary Modular Arithmetic

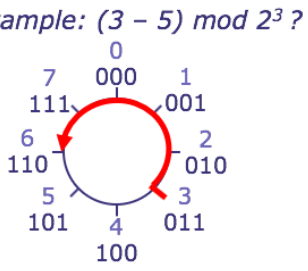


Table 3.4: Modular Arithmetic Example

3.3.2 Sign-Magnitude Representation

We use *sign-magnitude* representation for decimal numbers, encoding the number's sign (using "+" and "-") separately from its magnitude (using decimal digits). If we consider using this same approach for binary numbers, we could encode positive numbers using a 0 in the most significant position and negative numbers using a 1 in the most significant position, as shown in Figure 3.4.

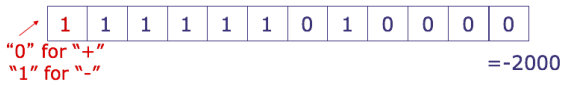


Figure 3.4: Sign Magnitude Representation of Binary Numbers

The problem with this representation is that first of all it results in two representations for the value 0 (+0 and -0). The other issue is that circuits for addition and subtraction are different and more complex than with unsigned numbers.

Instead, we can derive a better encoding known as two's complement encoding. This encoding simply relabels some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic, as shown in Figure 3.5.

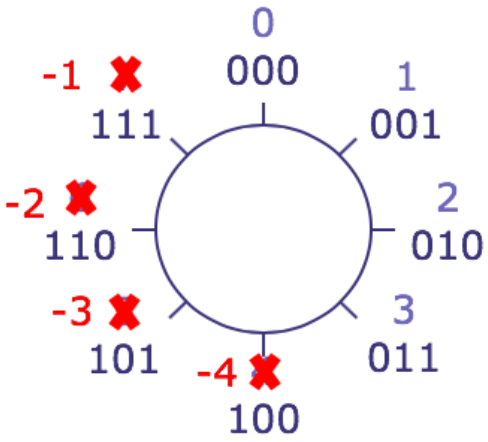


Figure 3.5: Two's Complement Encoding of Binary Numbers

More precisely, in two's complement encoding, the high-order bit of the N-bit representation has negative weight while all other bits have a positive weight so the value of a 2's complement number is $v = -2^{N-1}b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i$ as illustrated in Figure 3.6.

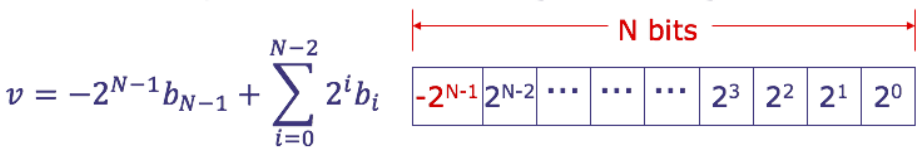


Figure 3.6: Two's Complement Encoding Formula

Using this formula, one can derive the most negative number that can be represented using N bits to be $10...0000 = -2^{N-1}$, and the most positive number $01...1111 = +2^{N-1}-1$. A couple of properties worth noting are that all negative two's complement numbers have a "1" in the high-order bit. Also, regardless of the width of your representation, if the bits are all 1, then the value of the number is -1.

3.3.3 Two's Complement Arithmetic

To negate a number (i.e., compute -A given A) in two's complement, we invert all the bits and add one. Figure 3.7 illustrates a proof for why this is the case. It begins with a mathematically correct equation which states that $-A + A = 0 = -1 + 1$. A is then subtracted from both sides of the equation to arrive at the equation that states that $-A = (-1 - A) + 1$. Next, we evaluate the expression $(-1 - A)$ by converting the values to binary where $-1 = 0b11...11$ and $A = A_{n-1}A_{n-2}...A_1A_0$. We then subtract the A from -1 beginning with the right most bit. If $A_0 = 0$, then $1 - A_0 = 1$. If $A_0 = 1$, then $1 - A_0 = 0$. In both cases $1 - A_0 = \sim A_0$. The remaining bits can be subtracted in the same manner arriving at the fact that $(-1 - A) = \sim A_{n-1}\sim A_{n-2}...\sim A_1\sim A_0$. Plugging this back into our equation for -A, we find that -A is equivalent to flipping all of the bits of A and adding 1.

$$-A + A = 0 = -1 + 1$$

$$-A = (-1 - A) + 1$$

$$\begin{array}{r} 1\ ...1\ 1 \\ -\ A_{n-1}...A_1A_0 \\ \hline \overline{A_{n-1}}...\overline{A_1}\overline{A_0} \end{array}$$

Figure 3.7: Negating Two's Complement Proof

In order to compute $A - B$, we can simply use addition and compute $A + (-B)$. This implies that we can use the same circuit to both add and subtract!

Let's consider a few two's complement arithmetic examples. Consider computing $3 - 6$ using 4-bit 2's complement addition. The number 3 in 4-bit two's complement is `0011` and 6 is `0110`. Using our negating formula, we can easily derive -6 in 4-bit 2's complement as `1010` by taking the representation for 6 (`0110`), flipping all the bits (`1001`) and adding 1 (`1010`).

Figure 3.8 shows the computation resulting from adding `0011` to `1010`, which results in `1101`. To convince ourselves that this is in fact equal to -3 as we would expect, we can flip the bits and add 1 to get `0011` which is equal to 3, thus `1101` is in fact equal to -3.

Now consider a second example using 3-bit 2's complement addition where we want to compute the result of $3 - 2$. The number 3 in 3-bit 2's complement is `011`, the number 2 is `010`, and -2 is `110`. When we add `011` to `110` we end up with a fourth sum bit. The correct answer comes from only the bottom 3 bits which are `001`. The reason that we get this

Figure 3.8: Two's Complement Example 1

extra one which should be ignored can be understood by examining Figure 3.9. It shows that adding `011` to `110` using modular arithmetic is equivalent to computing $(3 + 6)$ mod 8. Doing this is equivalent to working our way around the binary circle from 3 by moving 6 positions in the clockwise direction which crosses the zero value and results in `001`. The fact that we crossed the zero is the reason for the extra 1 in the fourth position which we ignore.

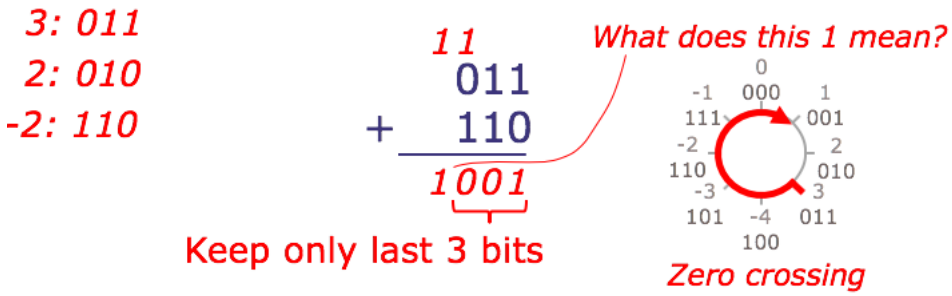


Figure 3.9: Two's Complement Example 2

There are a few interesting things to note about this example. First of all, we were trying to subtract 2 by adding -2 in 2's complement representation. Note that the 3-bit 2's complement encoding of -2 is equal to the 3-bit unsigned encoding of +6. This is because 3-2 in modular arithmetic is equivalent to $(3 + 6)$ mod 8. The second thing to note is that pictorially, subtracting 2 from 3 can be seen as moving two positions around the circle in a counter-clockwise direction. This too arrives at the same result of `001`.

3.4 How Boolean Algebra Relates to Binary Arithmetic

Boolean algebra can be used to describe binary arithmetic operations. For example, suppose that you want to describe the behavior of a 1-bit adder shown in Figure 3.10. A 1-bit adder, also known as a full adder (FA) has three inputs, the two 1-bit inputs that you are adding (A and B) plus any carry in from a less significant bit (C_{in}). The 1-bit adder has two outputs: a sum bit (S) and a carry out bit (C_{out}).

The truth table in Table 3.5 shows the relationship between the adder's three inputs and its outputs. Specifically, the sum output is one whenever either exactly one of the three inputs is one, or when all three inputs are one. The carry out is one when at least two of the inputs are one. The behavior of the 1-bit adder can be described using Boolean algebraic expressions that relate these inputs and outputs. The Sum-Of-Products representation of the sum output is:

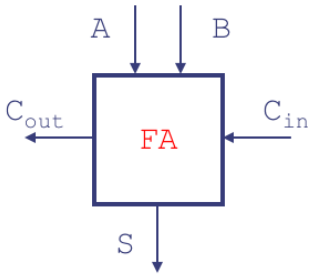


Figure 3.10: 1-bit Adder

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.5: 1-bit Adder Truth Table

$S = (\sim A)(\sim B)C_{in} + (\sim A)B(\sim C_{in}) + A(\sim B)(\sim C_{in}) + ABC_{in}.$

Note that for simplicity the `AND` $(\cdot)$ operator is implied and thus ommitted from each product term of the SOP.

The Sum-Of-Products representation of the carry out is:

$C_{out} = (\sim A)BC_{in} + A(\sim B)C_{in} + AB(\sim C_{in}) + ABC_{in}.$

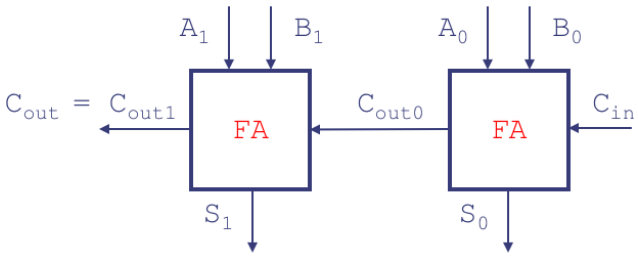


Figure 3.11: 2-bit Adder

One can describe the 2-bit adder of Figure 3.11 in a similar manner by concatenating two 1-bit adders back to back. Thus, the carry out of the least significant bit (C_{out0}) becomes the carry in of the next bit, and the carry out of the next bit (C_{out1}) becomes the carry out of the 2-bit adder. The rightmost full adder produces the least significant bit of the

2-bit sum, S_0, by adding inputs A_0, B_0, and the original carry in, C_{in}. The left full adder produces S_1 by adding inputs A_1, B_1, and C_{out0}.

Generalizing Fig. 3.11, we can create an w-bit adder for any width w by cascading more full adders, connecting the carry-out signal of one to the carry-in signal of the next. These are also called *ripple carry adders*, because of the way carry bits cascade from the lowest-order bit to the highest-order bit.[1]

3.5 Take Home Problems

Take Home Problem 1

1. Provide a truth table for a full adder (FA).
2. Specify a Sum of Products (SOP) representation for each of the full adder outputs.

Take Home Problem 1 Solution

Take Home Problem 2

1. Provide a truth table for a 2-bit adder.
2. Specify a Sum of Products (SOP) representation for each of 2-bit adder outputs.

Take Home Problem 2 Solution

[1] A consequence of this linear "ripple" is that the signals have to propagate through a sequence of gates and wires whose length is proportional to w. This may limit the clock frequency at which the adder can operate. We will address this question in later chapters with an improved *carry lookahead adder*.

Chapter 4

Digital Abstraction

In this chapter we will describe how we transform an inherently continuous signal into a digital signal and why it is a good idea to do so. It is because of this digital abstraction that we are able to build larger and larger systems without the loss of any fidelity even in the presence of electrical noise. This chapter will deepen our understanding of digital logic but is not necessary to understand the rest of this book.

4.1 Analog Versus Digital Systems

Analog systems represent and process information using continuous signals that represent real physical phenomena such as voltage, current, temperature, or pressure. Digital systems represent and process information using discrete symbols. Typically, the binary symbols 0 or 1, are encoded using ranges of a physical quantity such as voltage. Sample analog and digital signals are shown in Figure 4.1.

(a) Analog Signal (b) Digital Signal

Figure 4.1: Analog versus Digital Signals

4.1.1 Example: Analog Audio Equalizer

Consider the analog audio equalizer shown in Figure 4.2. Its goal is to convert a voltage signal representing pressure into a frequency-equalized voltage signal. To do this it applies multiple filters and then merges the signals together as shown in Figure 4.3. However, the resulting output does not exactly match the expected output. There are multiple reasons for this including electrical noise, manufacturing variations, and the degradation of components over time.

Figure 4.2: Analog versus Digital Signals

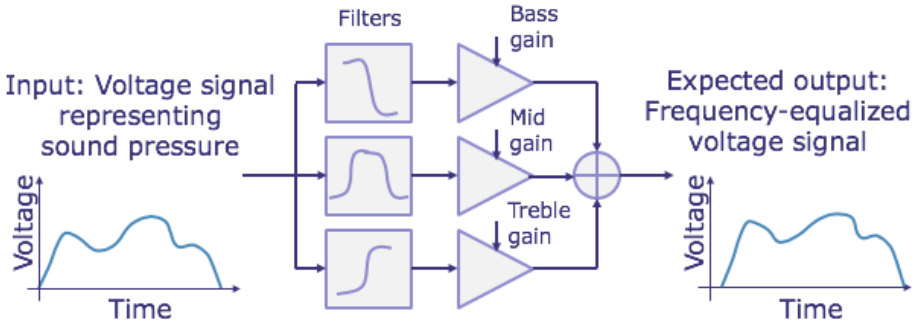


Figure 4.3: Analog versus Digital Signals

The significant advantage of digital signals over analog signals is that analog signals degrade due to electrical noise whereas digital signal can tolerate electrical noise without signal degradation.

However, the world is not digital, we would simply like to engineer it to behave that way. In the end, we must use real physical phenomena to implement digital designs. Specifically, we usage voltage to model digital signals.

4.2 Using Voltages Digitally

As a first attempt of modeling digital signals using voltages, consider a threshold voltage V_{th} that evenly divides the range of acceptable voltages in two, as shown in Figure 4.4. One can assign all voltage values $V < V_{th}$ to 0, and all voltages $V \geq V_{th}$ to 1. At first glance this seems like a reasonable approach, but consider what happens when V is close to V_{th}. The closer one gets to the threshold voltage, the harder it becomes to distinguish between a low and high value.

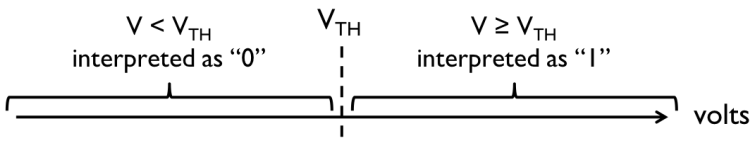


Figure 4.4: Single threshold voltage

For this reason, it is better to use two separate thresholds, one for low voltages where any $V \leq V_L$ is considered a valid 0, and one for high voltages where any $V \geq V_H$ is considered a valid 1 as shown in Figure 4.5. Note that for all voltages $V_L < V < V_H$ the value of the signal is undefined.

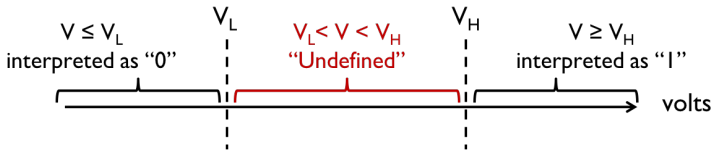


Figure 4.5: Low and high threshold voltages

A two threshold system will avoid confusing low and high values that are close to the threshold. However, consider the system shown in Figure 4.6. The desired behavior is that if one digital system produces a valid 0, or low output, and the output is then used as an input to another digital system, that the second system also consider its input to be a valid 0. However, if some electrical noise is introduced on the wire that carries the signal from the output of the first system to the input of the second, then if the signal was just under the V_L threshold, it's possible that it will be seen as slightly over V_L at the input to the second system which would be treated as an invalid input. To address this issue, it is desirable to have stricter constraints on digital outputs than on digital inputs so an output that has a small amount of noise introduced to it, can still be a valid input to the downstream system.

Thus, four threshold are defined as shown in Figure 4.7. An input is considered valid if $V_{in} \leq V_{IL}$ or $V_{in} \geq V_{IH}$. Analogously, an output is considered valid if $V_{out} \leq V_{OL}$ or $V_{out} \geq V_{OH}$.

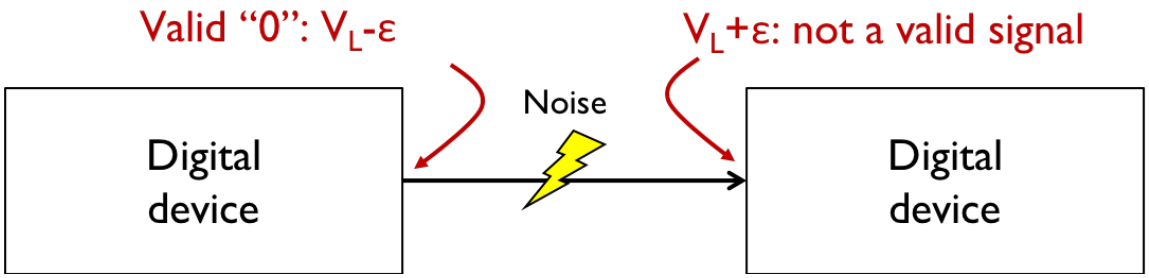


Figure 4.6: Electrical noise can make a valid signal become invalid

4.2.1 Noise Margins

The amount of noise that a system can tolerate is known as the systems noise margin. The low noise margin is defined as $V_{IL} - V_{OL}$, and the high noise margin is defined as $V_{OH} - V_{IH}$. The noise immunity of the system is the minimum of the two noise margins. In order to make our digital systems reliable, one would like to provide as large of a noise immunity as possible.

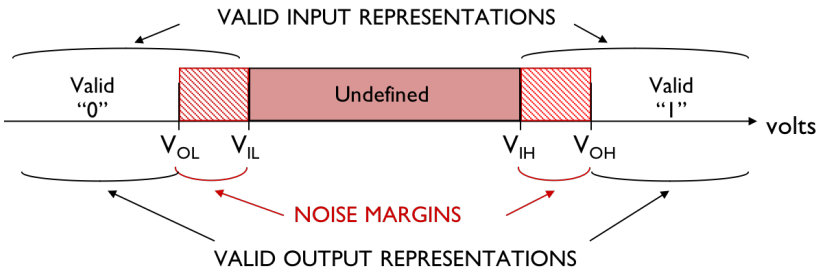


Figure 4.7: Noise margins

4.2.2 Static Discipline

The static discipline states that all digital systems must produce a valid output for every valid input. Thus, for all $V_{in} \leq V_{IL}$ or $\geq V_{IH}$, a digital system will produce an output where $V_{out} \leq V_{OL}$ or $\geq V_{OH}$.

4.2.3 Restorative Systems

One of the greatest advantages of digital systems over analog systems is their ability to cancel out noise as a result of their static discipline. Figure 4.8 shows that as a valid input traverses multiple analog systems, any noise it encounters is transmitted in a cumulative manner to the following analog system. Whereas Figure 4.9 shows that a digital system can tolerate noise because even if noise is introduced at the input to system `f`, the output of `f` is equal to what the output would have been if no noise was introduced (provided that the noise is smaller than the noise margin). In the analog system, the output of system `g` is degraded by the cumulative noise introduced at the inputs to both `f` and `g`. The digital system, however, behaves as if no noise had been introduced into the system at all.

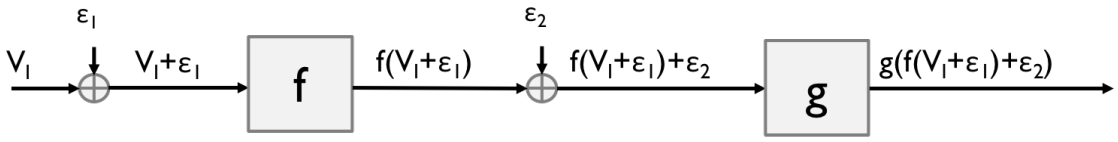


Figure 4.8: Noise in analog systems accumulates

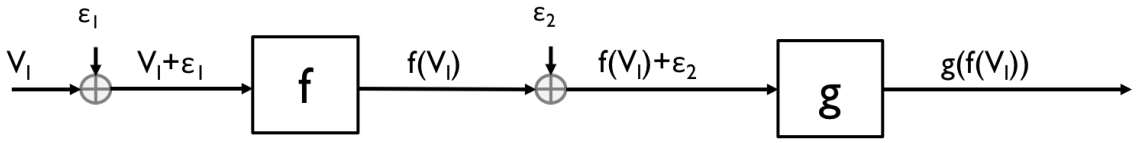


Figure 4.9: Digital systems cancel noise

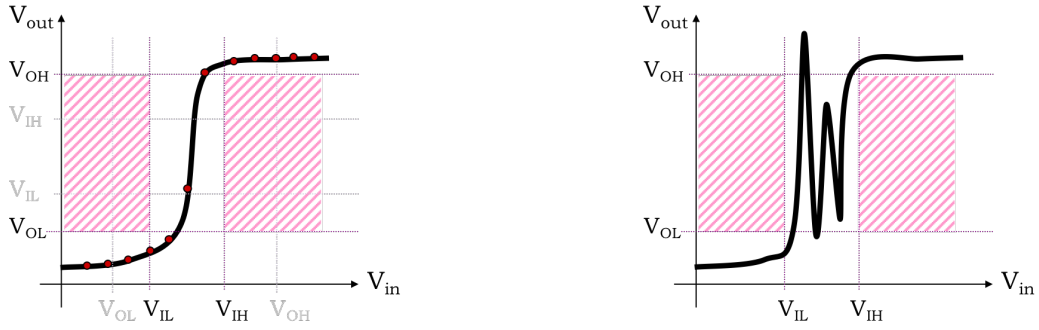
4.3 Voltage Transfer Characteristic

The relationship between the input voltage and the output voltage of a digital system is specified using a voltage transfer characteristic curve which identifies possible valid V_{out} for all valid V_{in}.

The blocks in red are known as the *forbidden regions* because they correspond to portions of the graph where inputs are valid but if the plot went through those regions then the system would be producing an invalid output which would break the static discipline.

In order to provide non-zero noise margins, it must be the case that $V_{OL} < V_{IL} < V_{IH} < V_{OH}$. This implies that $V_{OH} - V_{OL} > V_{IH} - V_{IL}$. Thus the slope of the curve is > 1 for $V_{IL} < V_{in} < V_{IH}$. This means that the system must have gain which implies that it must be built from active components like transistors rather than passive components like resistors.

Note that for $V_{IL} < V_{in} < V_{IH}$, the transfer curve may do anything as shown in the valid transfer characteristic of Figure 4.10b.



(a) Digital voltage transfer characteristic

(b) An alternate valid digital voltage transfer characteristic

Figure 4.10: Valid Voltage Transfer Characteristics (VTCs)

4.4 Take Home Problems

Take Home Problem 1 - VTC

Suppose that you measured the voltage transfer curve of a device, shown in Fig. 4.11. Can we find a signaling specification (V_{IL}, V_{IH}, V_{OL}, V_{OH}) that would allow this device to be a digital inverter? If so, give the specification that maximizes noise margin.

Take Home Problem 1 Solution

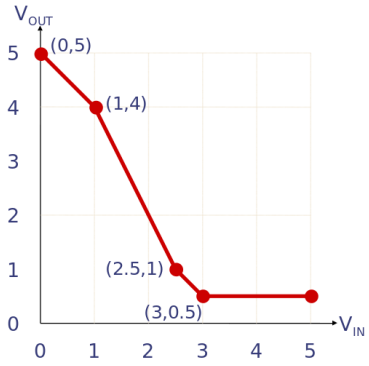


Figure 4.11: Voltage Transfer Curve of a device

Chapter 5

Combinational Circuits

We will begin by describing what is a combinational circuit and how we can compose combinational circuits to build larger combinational circuits with predictable functionality. We rely heavily on Boolean algebra to describe the functionality of combinational circuits and maintain one-to-one correspondence between combinational circuits and Boolean expressions. In addition to the functionality of a Boolean circuit, we have to pay attention to their physical properties, like area, propagation delay, and power consumption to design an optimal or even a good circuit.

The dominant technology today for building digital circuits is the CMOS (Complementary Metal Oxide Semiconductor) technology. How combinational circuits are built in CMOS is beyond the scope of this book. Typically, the manufacturer provides a standard cell library of primitive building blocks, i.e., gates, along with their physical properties, and the task of a digital designer is to implement the desired combinational circuit using the library elements. The standard cell library not only differs from manufacturer to manufacturer but it also changes by periodic improvement in the underlying CMOS technology. As we consider this problem in greater detail throughout this chapter, it will become clear that we need automated tools to implement a combinational circuit in a specific standard cell library.

We will begin by defining combinational circuits and their physical properties, and show by example the optimization problem in implementing the circuit using a standard gate library.

5.1 The Static Discipline

A *combinational device* is a circuit element that has:

- one or more **digital inputs**
- one or more **digital outputs**
- a **functional specification** that details the value of each output for every possible combination of valid input values

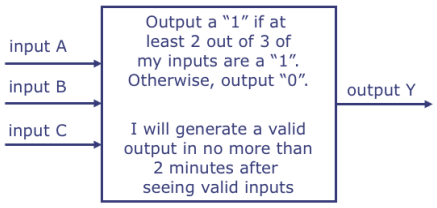


Figure 5.1: Combinational Device Specification

- a **timing specification** consisting (at a minimum) of a propagation delay (t_{PD}): an upper bound on the required time to produce valid, stable output values from an arbitrary set of valid, stable input values
- an **area specification** that specifies the actual physical size of the circuit

As stated in Section 4.2.2, the static discipline states that all digital systems must produce a valid output for every valid input. A *valid digital input* is defined as $V_{in} \leq V_{IL}$ or $\geq V_{IH}$. A *valid digital output* is defined as $V_{out} \leq V_{OL}$ or $\geq V_{OH}$.

In addition, every combinational device has a *functional specification* that specifies the value of each output for every possible combination of valid inputs. Figure 5.1 describes a combinational device whose functional specification is that it produces a 1 if at least two out of its three inputs are 1, and it produces 0 otherwise.

In order to specify the exact behavior of a combinational device, it must also include a *timing specification* which specifies the *propagation delay*, t_{PD}, which is an upper bound on the time required to produce valid, stable outputs given valid inputs. Figure 5.1 describes a digital device whose t_{PD} is 2 minutes.

Another important property of combinational devices is their size, or *area* specification. Section 5.2 will more fully describe how size constraints are taken into consideration when designing combinational circuits. Typically, one trades off propagation delay and area depending on the requirements of your circuit.

Note that combinational circuits have no memory or state, thus given the same set of inputs, they always produces the same outputs. In addition, since combinational devices must satisfy the digital abstraction, combinational circuits can tolerate noise and are restorative in that errors do not accumulate as explained in Section 4.2.3.

5.1.1 Boolean Gates

The Boolean operators that were introduced in Chapter 2 are examples of primitive combinational circuits which are also known as gates. Figure 5.2 illustrates the pictorial form of each of the basic gates together with its name and boolean algebraic description. These gates perform the operations of the `NOT`, `AND`, `OR`, `NAND`, `NOR`, and `XOR` Boolean operators.

5.1.2 Composing Combinational Devices

A set of interconnected elements is itself a combinational device if each circuit element is combinational and every input is connected to exactly one output or to a constant (0 or 1),

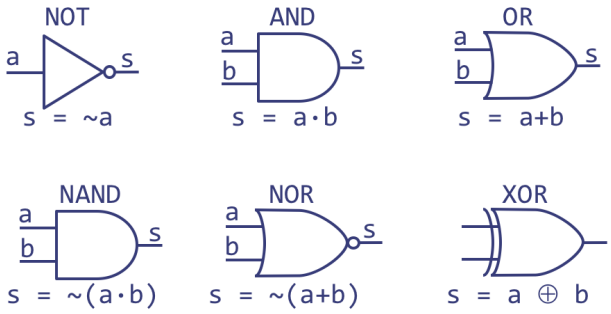


Figure 5.2: Basic Gates

and the circuit contains no directed cycles. Thus, the basic gates can be connected to form more complex combinational devices, and these complex devices can be further combined with any other combinational device to perform any Boolean function.

Figure 5.3 provides an example of combining three simpler combinational devices: A, B, and C into one larger combinational device. To convince ourselves that the composed circuit is also a combinational device, one must verify the five properties that define a combinational device.

1. The composed device has digital inputs - Since we know that devices A, B, and C are combinational devices, then their inputs must be digital inputs. The inputs to the composed device are made up of a combination of inputs to A, B, and C thus the inputs of the composed device are also digital.

2. The composed device has digital outputs - The output of the composed device is the output of device C which is known to have a digital output thus the composed device must also have digital outputs.

3. There exists a functional description of the composed device - Devices A, B, and C are digital thus there exists a functional description f_A, f_B, and f_C. The functional description of the composed device is $f_C(f_A(X,Y), f_B(f_A(X,Y),Z)$.

4. One can derive the propagation delay, t_{PD} of the composed device - If the basic gates A, B, and C have propagation delay $t_{PD,A}$, $t_{PD,B}$, and $t_{PD,C}$ respectively, then we can derive the propagation delay of the composed device as the sum of these three propagation delays since an input may need to traverse all three devices to arrive at the output. Thus, $t_{PD} = t_{PD,A} + t_{PD,B} + t_{PD,C}$.

   Arvind: This is a conservative estimate of propagation delay unless devices A, B and C are primitive

5. One can derive a rough estimate for the area of the composed device by summing the areas of the individual devices A, B, and C. Thus the $area \approx area_A + area_B + area_C$. In reality, this may underestimate the area since additional physical space may be required for the wires connecting the three individual devices.

Since the five properties of a combinational device are satisfied by the composed device, then it too is a combinational device.

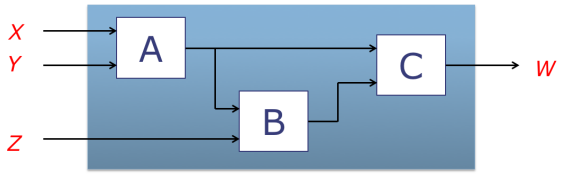


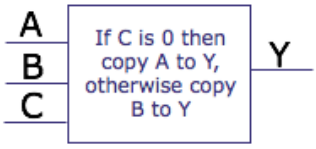
Figure 5.3: Composing Basic Combinational Devices into Larger Combinational Devices

5.1.3 Functional Specifications

There are many ways to specify the function of a combinational device.

So far, we have presented two systematic approaches as illustrated in Figure 5.4:

1. *Truth tables* enumerate the output values for all possible combinations of input values.
2. *Boolean expressions* are equations containing binary (0/1) variables and three operations: AND (·), OR (+), and NOT (∼).



(a) Textual Description

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(b) Truth Table

$$Y = ((\sim C)\cdot A)+(C\cdot B)$$

(c) Boolean Expression

Figure 5.4: Functional Specifications

In addition to truth tables and Boolean expressions, a functional specification can be provided by drawing an acyclic interconnection of Boolean gates whose composition result in the desired functionality. A more modern approach is to use a hardware description language where the description of the circuit functionality can be translated or compiled into a combinational circuit made up of primitive gates that are connected in an acyclic manner.

5.1.4 Propagation Delay

****

5.2 Combinational Circuit Properties

In addition to their functionality, combinational circuits have two important physical properties: propagation delay (t_{PD}) and area. The propagation delay is the time it takes for the

Gate	Delay (psec)	Area (μ^2)
Inverter	20	10
AND2	50	25
NAND2	30	15
OR2	55	26
NOR2	35	16
AND4	90	40
NAND4	70	30
OR4	100	42
NOR4	80	32

Table 5.1: Sample Standard Cell Library

output to stabilize after stable inputs are available. Area is the physical size of the circuit that implements the desired functionality. Both of these properties are derived from the properties of the gates used to build the circuit.

5.2.1 Standard Cell Library

Most circuits are built from some standard cell library implemented in a specific CMOS (Complementary Metal Oxide Semiconductor) technology. Table 5.1 provides an example of the information provided in a standard cell library. Specifically, it lists all of the available gates and their physical characteristics. Note that standard cell libraries typically include basic gates (i.e., inverter, 2-input `AND` (`AND2`), 2-input `OR` (`OR2`)) as well as gates with more inputs (e.g., `AND4` is a 4-input `AND` gate).

There are a couple of interesting properties to note from the standard cell library data. The first is that in current CMOS technology inverting gates (e.g., `NAND2`) are faster and smaller than non-inverting gates (e.g., `AND2`). The second is that delay and area grow with the number of inputs. Thus a 4-input `AND` gate is both slower and larger than a 2-input `AND` gate.

5.2.2 Design Tradeoffs: Delay vs Size

The delay and size properties of gates lead to design tradeoffs in implementing most combinational functions. For example, consider a circuit that takes the logical `AND` of 4 inputs. This could be implemented using a single `AND4` gate from our sample standard library. This gate has a propagation delay, $t_{PD} = 90psec$, and $area = 40\mu^2$. Alternatively, the same functionality could be achieved by using a 4-input `NAND` gate followed by an inverter. This would result in a $t_{PD} = 70 + 20 = 90psec$, and an $area = 30 + 10 = 40\mu^2$. These are both equivalent to the properties of a single `AND4` gate. A third way of implementing a 4-input `AND` gate is using two `NAND2` and one `NOR2` as shown in Figure 5.5. The propagation delay of this implementation is $t_{PD} = t_{PD,NAND2} + t_{PD,NOR2} = 30 + 35 = 65psec$. Note that although this implementation includes two `NAND2` gates, only one of the gates is on the path from any input to output and thus only one of them can affect the propagation delay. The area of this alternative implementation, however, is estimated by summing the area of the three constituent gates. Thus, $area = 2area_{NAND2} + area_{NOR2} = 2(15) + 16 = 46\mu^2$. This

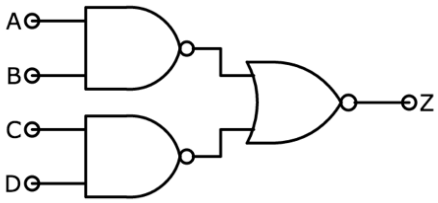


Figure 5.5: Alternate `AND4` Implementation

Figure 5.6: Alternate `NOR2` Implementation

implementation demonstrates that depending on how the circuit is implemented, one will end up with different values for propagation delay and area. One must generally trade off one for the other. Thus to get a smaller propagation delay, one often has to accept a larger area and vice-versa. Which implementation you pick depends on the design criteria which dictates whether you need to optimize for size of speed.

Using DeMorgan's Laws to Interpret Circuit Functionality

It may not be directly obvious that the functionality of Figure 5.5 is in fact equivalent to the functionality of a 4-input `AND` gate. DeMorgan's Laws state that $(\sim A)\cdot(\sim B) = \sim(A+B)$ and $(\sim A)+(\sim B) = \sim(A\cdot B)$. Pictorially the former can be visualized as a `NOR` gate being equivalent to an `AND` gate with its two inputs inverted as shown in Figure 5.6. If you replace the rightmost `NOR2` with this version of the gate, then you have two wires that have two inversions (shown as bubbles or small circles) in a row. The two inversions cancel each other out, thus the circuit is equivalent to two `AND2` gates feeding a third `AND2` gate which is logically equivalent to a 4-input `AND` gate.

5.2.3 Multi-Level Circuits

Suppose we are restricted to using only 2-input `AND` and `OR` gates, then a sum-of product (SOP) circuit cannot always to be realized using two levels of logic. For example, consider the function $F = A\cdot C + B\cdot C + A\cdot D + B\cdot D$. In order to implement this function using only 2-input gates, one would need to first compute two subfunctions $X1 = A\cdot C + B\cdot C$ and $X2 = A\cdot D + B\cdot D$. The function could then be rewritten as $F = X1+X2$. The implementation of this circuit using only 2-input gates is shown in Figure 5.7.

Fast microprocessors typically use 8 levels of logic, while slower processors use as many as 14-18 levels of logic!

5.3 Logic Synthesis: Mapping a Circuit to a Standard Cell Library

5.3.1 An Example

Consider the simple standard cell library shown in Figure 5.8. Suppose that you wanted to map the circuit of Figure 5.9 to this standard cell library with the goal of optimizing your

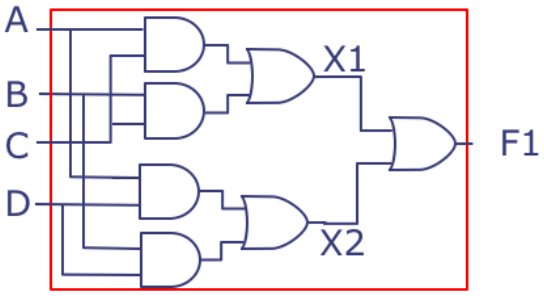


Figure 5.7: Multi-Level Circuit Implementation

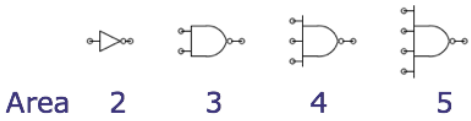


Figure 5.8: Simple Cell Library

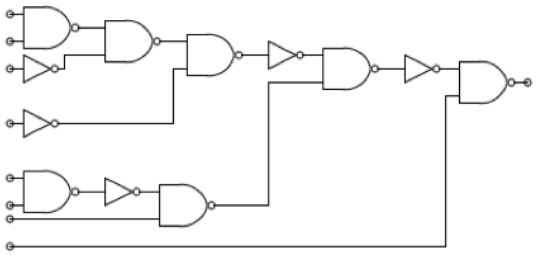
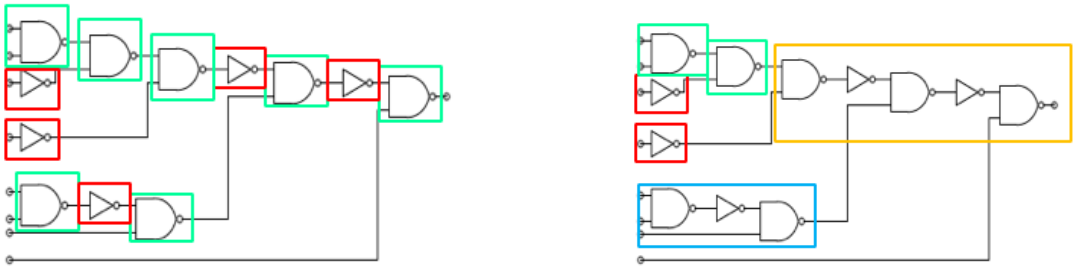


Figure 5.9: Sample Circuit



(a) Example 1

(b) Example 2

Figure 5.10: Various Mappings of Circuit to Standard Cell Library

circuit for minimum area. Mapping a circuit to a standard cell library with a particular goal in mind is known as *synthesizing* the circuit.

There is an analogy here to compiling a software programming language to machine code. Machine code is based on a standard set of instructions (the "instruction set") for a particular architecture, such as x86, ARM, POWER, Sparc, MIPS and so on. A compiler has to select suitable instructions from the given repertoire in order to map the behavior of the high-level language. Similarly, in hardware synthesis, the tool has to select gates from a standard repertoire in order to map the desired circuit behavior.

Figure 5.10 illustrates two possible implementations of our sample circuit using the standard cell library provided. Example 1 consists of seven 2-input `NAND` gates of area 3 and five inverters of area 2 for a total area = $7(3) + 5(2) = 31$. Example 2 implements the circuit using one 4-input `NAND` gate of area 5, 1 3-input `NAND` gate of area 4, two 2-input `NAND` gates of area 3, and two inverters of area 2 for a total area = $5 + 4 + 2(3) + 2(2) = 19$. Given the optimization goal of minimal area, one would prefer the implementation of Example 2 over that of Example 1.

5.3.2 Logic Synthesis Tools

As you may imagine, synthesizing an optimized circuit is a very complex problem. It involves:

- Boolean simplification
- Mapping to cell libraries with many gates
- Multi-dimensional tradeoffs (e.g., minimize are-delay-power product)

Doing this for all but the smallest circuits is infeasible to do by hand. Instead, hardware designers write circuits in a **hardware description language**, and use a **synthesis tool** to derive optimized implementations.

In practice, *tools* use Boolean simplification and other techniques to synthesize a circuit that meets certain area, delay, and power goals. Figure 5.11 illustrates the various pieces of information that feed a logic synthesis tool so that it can identify an optimized circuit implementation. Note that even for automated tools, it is not always feasible to find the absolute optimal circuit.

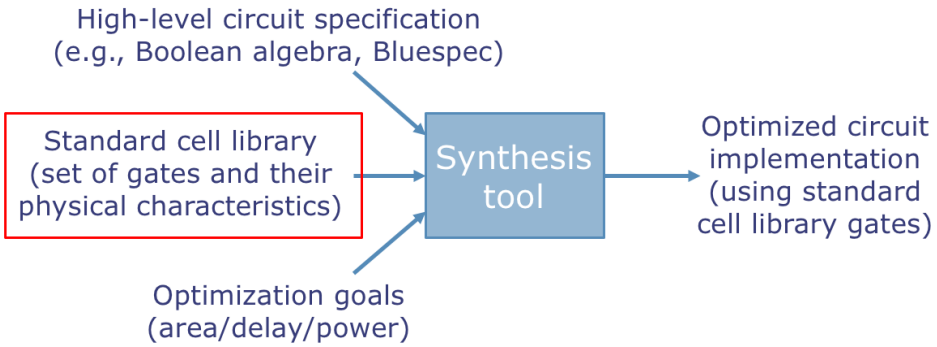


Figure 5.11: Logic Synthesis Tool

5.4 Take Home Problem

Take Home Problem 1

Find a minimal Sum-Of-Products (SOP) expression for the following Boolean (SOP) expression: A·(∼B)·(∼C) +A·(∼B)·C +(∼A)·(∼B)·C +(∼A)·B·C.

Then using the standard cell library of Table 5.1, find an implementation of this minimal SOP that minimizes area.

Take Home Problem 1 Solution

Chapter 6

Describing Combinational Circuits in BSV

In the preceding chapters, we approached the subject of combinational logic from several different perspectives, ranging from Boolean algebra formalism to logic circuit realizations. The motivation in this book for studying combinational logic—and sequential logic in later chapters—is to design interesting digital circuits with useful functionalities. For example, we already saw how to represent whole numbers in binary-valued digits and then how to design a ripple-carry adder to add two binary-represented values.

Until now, we made do with formal logic equations and informal logic circuit diagrams to express the combinational logic we had in mind (e.g., the adder). As we get into discussing more interesting and elaborate designs, we need a more clear and less clumsy way to express them. All the better if the description also can lead directly to implementation. In this chapter, we will learn how to describe combinational logic using the BSV hardware description language. The reader should find it much easier and faster to specify the desired combinational logic in BSV than drawing logic diagrams. At the same time, the BSV description is as precise as Boolean logic equations. With the help of design automation tools, we can simulate a BSV description to observe its operations, and we can synthesize a BSV description to ultimately produce physical implementations.

Just as we have so far focused on only combinational logic and circuits, this chapter will present only a basic subset of BSV directly relevant to the specification of combinational circuits. The features of BSV that truly set it part from contemporary digital design tools and practices will become apparent after we study sequential logic design. We will see in Chapter 7 the distinguishing features that shaped the new digital design thinking in this book.

6.1 A First Example: Ripple Carry Adder

We discussed in Section 3.4 how to cascade w full adders into a ripple-carry adder (Figure 6.1) to add two w-bit binary-value inputs and produce their w-bit binary-value sum as output. As a first example in describing combinational circuits in BSV, we will revisit the same ripple-carry adder construction. In this exercise, it is not sufficient we capture the

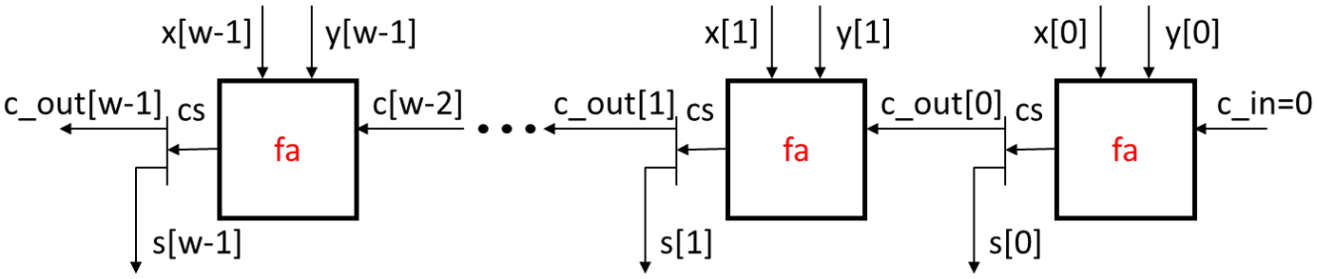


Figure 6.1: w-bit Ripple Carry Adder circuit, first seen in Figure 3.11

correct combinational logic function of an adder in the truth-table sense. The BSV description must also capture the exact circuit organization of the ripple-carry adder so that when synthesized the resulting implementation also has the intended implementation properties such as cost and delay.

6.1.1 Full Adder

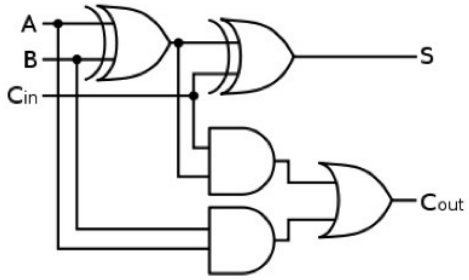


Figure 6.2: Full adder circuit, first seen in Figure 3.10

We begin by describing the full adder circuits in Figure 6.2 using the `function` construct in BSV.

A Description of the Full Adder in Figure 6.2

```
1  function Bit#(2) fa (Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
2     Bit#(1) temp;
3     Bit#(1) s;
4     Bit#(1) c_out;
5     Bit#(2) result;
6     temp = (a ^ b);
7     s = temp ^ c_in;
8     c_out = (a & b) | (c_in & temp);
9     result[0] = s;
10    result[1] = c_out;
11    return result;
12 endfunction
```

The keyword `function` in Line 1 starts the declaration of a BSV function named `fa`, and the declaration ends on line 11 with the keyword `endfunction`. The function has three

input arguments named `a`, `b`, and `c_in`. The other notations on Line 1 indicate the *data types* of the arguments and the result. The inputs and output of a BSV function used in this way correspond to input and output wires of the combinational circuits under design.

An input or output of a BSV function is not restricted to a single wire. An input or output can bundle multiple conceptually related wires (*e.g.,* those together convey a binary-value input or output). Thus a BSV function declaration must also specify the *type* of each input and the output. We will initially work with only the type `Bit#(`w`)`—think of a bundle of w wires (also sometimes called a w-bit *bus*). In Line 1, we see the inputs `a`, `b`, and `c_in` are each a single wire, as indicated by the type specifier `Bit#(1)` preceding each input. Between the `function` keyword and the function name `fa` is the type specifier `Bit#(2)` representing the data type of the output, which thus has 2 wires.

Line 2 declares the variable `temp` of the type `Bit#(1)`. Later in Line 6, the expression `(a ^ b)` is assigned to `temp`. In BSV, as in C, the `&`, `|` and `^` symbols stand for the bit-wise AND, OR, and XOR (exclusive OR) operators. Unlike C, where these bitwise operators only apply to operands or size 8, 16, 32 or 64 bits, in BSV they can be applied to any same-size type `Bit#(`w`)` operands to yield a type `Bit#(`w`)` result. In the current case, the expression `(a ^ b)` corresponds to the upper-left-most XOR gate in Figure 6.2. After the assignment, the variable `temp` can be used to refer to the output wire of that gate.

Similarly, Lines 3-5 declare variables `s`, `c_out` and `result`, and subsequent lines assign values to them. The right-hand-side expressions of the assignments correspond to the remaining combination circuits in Figure 6.2. A complex expression like `(a & b) | (c_in & temp)` corresponds to a commensurately complex multi-gate circuits. Notice in both the BSV function code and the corresponding circuit diagram, the combinational logic producing `s` and `c_out` use `temp` in their input.

In Line 11, the keyword `return` indicates the output of function `fa` (which was declared to have type `Bit#(2)` in Line 1) is `result` (which indeed has type `Bit#(2)`). The zero'th and first bit of `result` are assigned individually using the selection notation—`result[0]` for the zero'th bit and `result[1]` for the first bit. In general, the notation `x[`j`]` represents a selection of the j^{th} bit of `x`, with the common convention that $j = 0$ is the least significant bit. The two 1-bit output wires in Figure 6.2 are combined into a single 2-bit output in the BSV description to conform to the restriction that a BSV function has only one output (although the output could be of an arbitrarily elaborate type).

BSV is a strongly typed language: every variable and expression in a BSV description must have a type, which is either declared explicitly or inferrable unambiguously from usage by the BSV compiler. Assignments can only be made with matching types, and operators can only be applied to meaningfully typed operands. BSV compilation will report an error when there are type inconsistencies, and terminate the compilation. We will return to discuss the type system in BSV in Section 6.3.

The previous BSV description of `fa` was written in a purposely verbose style to make obvious the one-to-one correspondence with the logic diagram in Figure 6.2. There are many features in BSV to make a circuit description more succinct without losing details. The following is an equivalent description of the logic circuit in Figure 6.2 that would be more typically written by a practiced BSV user.

A Compact Description of the Full Adder in Figure 6.2

```
1 function Bit#(2) fa (Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
2    let s = (a ^ b) ^ c_in;
```

```
3     let c_out = (a & b) | (c_in & (a ^ b));
4     return {c_out,s};
5  endfunction
```

The type and interface declarations of `fa` in Line 1 above matches the first version we saw earlier. Instead of declaring `s` and `c_out` variables with explicit types, this version uses the `let` notation to introduce `s` and `c_out` without stipulating their types. The burden is on the BSV compiler to figure out their types based on how they are used and what are assigned to them. This version of the code is also more compact by omitting intermediate variables `temp` and `result`—the corresponding wires still exist even if they are not named explicitly. Interestingly, instead of `temp`, the expression `(a ^ b)` appears twice. The BSV compiler is sophisticated enough to recognize the repeated use of the same expression and would produce only one instance of the XOR gate. Chapter 10 will provide additional useful details—important to an effective BSV user—on how BSV descriptions are mapped into logic circuits. Lastly, Line 4 sets the expression `{c_out, s}` as the output of `fa`. The expression `{c_out, s}` represents a bit-concatenation of `s` and `c_out` and thus has the expected type `Bit#(2)`.

6.1.2 Two-bit Ripple Carry Adder

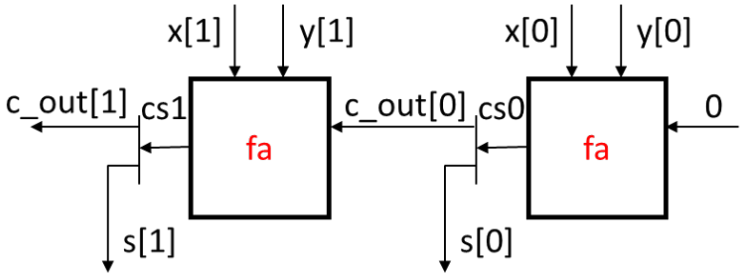


Figure 6.3: 2-bit Ripple Carry Adder circuit, first seen in Figure 6.3

Specifying combinational logic as a function in BSV creates a reusable design block that can be used in an enclosing design. We will now make use of `fa` as a black-box building block in the BSV function for a 2-bit ripple-carry adder. The BSV function below corresponds to the 2-bit ripple-carry adder circuit in Fig. 6.3.

2-bit Ripple Carry Adder

```
1  function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
2     Bit#(2) s;
3     Bit#(2) c_out;
4
5     Bit#(2) cs0 = fa(x[0], y[0], 0);
6     c_out[0] = cs0[1];  s[0] = cs0[0];
7
8     Bit#(2) cs1 = fa(x[1], y[1], c_out[0]);
9     c_out[1] = cs1[1];  s[1] = cs1[0];
10
```

```
11     return {c_out[1],s};
12  endfunction
```

Based on what we already learned about BSV, we can see the function declaration in Line 1 stipulates that the function `add2` has two `Bit#(2)` inputs, `x` and `y`, and produces a `Bit#(3)` output. Functionally, we expect `add2` should compute the 3-bit sum of the two inputs, treating `x` and `y` as 2-bit binary values. In general, adding two w-bit values produces a $w+1$-bit sum, accounting for the possible final "carry".

Besides instantiating two instances of the full adder `fa`, the body of `add2` does not contain any additional logic operators; it simply describes the wire connections in Figure 6.3. Lines 2 and 3 declare intermediate variables `s` and `c_out`. `s[`j`]` and `c_out[`j`]` will be used for the sum and carry outputs of the `fa` instance in bit position j.

Line 5 feeds the least-significant bit of `x` and `y` as the inputs to the first `fa` instance, with a default carry-in of 0. The `Bit#(2)` output of this `fa` instance is assigned to the variable `cs0`. In Line 6, `s[0]` and `c_out[0]` get assigned the corresponding wires selected from the compound output of `fa`. The same thing is repeated for the second `fa` instance at the second bit position. The carry-out `c_out[0]` of the first `fa` instance is the carry-in input of the second `fa` instance, forming a link of the ripple carry chain. Finally, Line 11 sets the `Bit#(3)` output of `add2` to `{c_out[1],s}`; the expression `{c_out[1],s}` is equivalent to `{c_out[1],s[1],s[0]}`.

Based on the 2-bit example, we should have confidence that we can, by extension, specify a ripple-carry adder (as in Figure 6.1) of any desired specific width $w = 2, 3, 4$ etc. Of course, this will require quite quite a lot tedious (almost) replicated code when w is large. Later we will show in Section 6.2.5 how to make use of the loop construct in BSV to simplify the description of combinational logic with regular, repetitive structure (such as in a ripple-carry adder). Moreover we will show in Section 6.4 how to describe a reusable adder design in a parameterized way so a single, compact description is all that is needed for ripple-carry adder of any width. Presently, we digress briefly to point out some important ideas to keep in mind when using BSV functions for combinational logic description.

6.2 More Combinational Examples in BSV

Besides adders, there are other combinational logic circuits that are used so commonly in digital design such that they are given special names. In this section, we will discuss a number of them with the help of BSV descriptions. The examples will also showcase additional BSV language features.

6.2.1 Selection

When discussing the full-adder in Section 6.1.1, we saw an example where we wanted to select one wire from a bundle of wires. The notation for selection is that given `x` of type `Bit#(`w`)`, `x[`j`]` represents a selection of the j^{th} bit of `x`, with the common convention that $j = 0$ is the least significant bit. It is invalid if $j > (w-1)$ or $j < 0$.

Similarly, the notation `x[`j`:`i`]` represents a selection a bundle/bus of bits `x[`j`]` through `x[`i`]`. This notation is valid for $0 \leq i \leq j < w$, and the result has type `Bit#(`$j - i + 1$`)`

In general, one can also declare *vectors* of any type, similar to arrays in programming languages. For example, we could have replaced the independently declared `cs0` and `cs1` in the 2-bit ripple-carry adder code (Section 6.1.2) by a single variable declaration `cs` of the type `Vector#(2,Bit#(2))`, i.e., a vector of two `Bit#(2)` elements. We can replace the use of `cs0` and `cs1` in the code by `cs[0]` and `cs[1]`. Correspondingly, `c_out[0] = cs[0][1]` and `s[0] = cs[0][0]`.

In the cases so far, the selections have been made with fixed indices that can be known at compile time—as opposed to values determined dynamically during execution of the resulting hardware. When selection is static, the resulting "circuit" is trivial: the compiler just connects the specified wire. Fig. 6.4 (left) shows a logic block for selecting one of the

Figure 6.4: Bit-select circuit, static index

four inputs to present as the output. For simplicity, assume inputs `x0`, `x1`, `x2`, and `x3` are simply `Bit#(1)`. Fig. 6.4 (right) shows, to make a static section, e.g., [2], the selected input `x2` is connected to the output; the remaining inputs are ignored. No logic gates are necessary. The circuits to allow dynamic selection at runtime is, however, more interesting.

6.2.2 Multiplexer

Fig. 6.5 (left) shows a logic block for selecting one of the four inputs to present as the output.

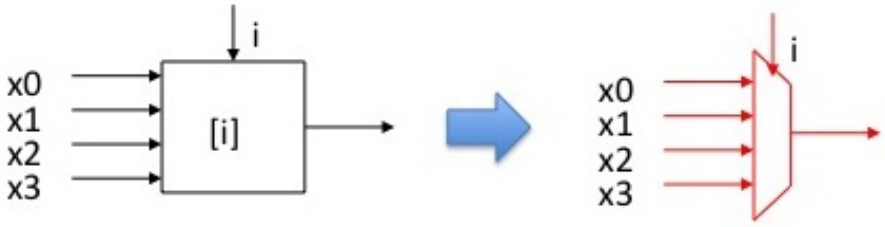


Figure 6.5: Bit-select circuit, dynamic index

This time, there is an additional input `i` to select which input should be connected to the output. The output should follow changes to the input, whether `x`'s or `i`, combinationally. This functionality is used so frequently it not only has a name, *multiplexer* or simply mux, but it is also drawn with the special symbol shown on the right of Fig. 6.5.

Fig. 6.6 (left) shows the symbol for a *2-way* multiplexer that takes two inputs, `A` and `B`, and forwards one of them to the output. The choice is made with a Boolean control input `s`. If `A` and `B` are single-bit, Fig. 6.6 (right) provides a gate-level implementation using AND and OR—as shown, the value of `A` is passed to the output if `s` is asserted. If `A` and `B` are

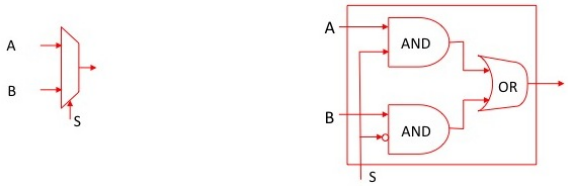


Figure 6.6: 2-way multiplexer

w-bit wide, we just replicate the 1-bit wide multiplexer w times with all of their `s` control inputs tied together.

Wider muxes to select from more inputs can in principle be created by cascading 2-way muxes. For example, Fig. 6.7 shows a 4-way mux created using three 2-way muxes in 2

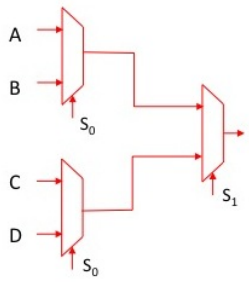


Figure 6.7: 4-way multiplexer

tiers. It now requires `s` of type `Bit#(2)` to select one of the four inputs. In general, for a n-way mux, we need a `Bit#(`$\lceil \lg_2 n \rceil$`)` control input `s` to identify one of the n inputs to forward. In general, n could be any number, not necessarily a power of two.

In BSV code muxes are usually expressed using classical programming language notation for alternatives: conditional expressions, "if-then-else" constructs, or "case" constructs, as shown in the following examples:

2-way Multiplexer using Conditional Expression

```
result = ( s ? a : b );
```

2-Way Multiplexer using If-Then-Else

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

4-way Multiplexer using Case

```
case (s2)
   0: result = a;
   1: result = b;
   2: result = c;
   3: result = d;
endcase
```

The differences in style in the above are mostly a matter of readability and personal preference, since the compiler will produce the same hardware circuits anyway. Both *bsc* and downstream tools perform a lot of optimization on muxes, so there is generally no hardware advantage to writing it one way or another.

6.2.3 Shifter

We will build up in steps to a general circuit for shifting a w-bit value right by n places.

Logical Shift Right by a Fixed Amount

Fig. 6.8 shows a circuit for shifting a 4-bit value right by 2 positions[1], filling in zeroes in the

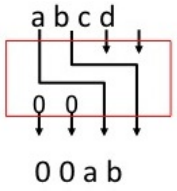


Figure 6.8: Logical shift right by 2

just-vacated most-significant bits.[2] When the shift amount is a constant known at compile time, you can see no logic gates are needed; it's just wiring. The same can be expressed in BSV using its built-in operators for shifting, which are the same as in C, Verilog and SystemVerilog:

Shift operators

```
1    abcd >> 2    // right shift by 2
2    abcd << 2    // left shift by 2
```

Even if shift operators were not built-in, it is very easy to write a BSV function for the same effect. Suppose we want to right-shift a 4-bit value right by 2 positions:

Shift function

```
1  function Bit #(4) logical_shift_right_by_2 (Bit#(4) arg);
2     Bit #(4) result = { 0, arg[3], arg[2] }; // alternatively: {0, arg[3:2]}
3     return result;
4  endfunction
```

Above, note that the "0" must be a two-bit value so that the result is of type `Bit#(4)`. The BSV compiler will automatically size it in this way. Alternatively, we can specify the width of constants explicitly like this: `2'b0` or `2'h0`. The "2" indicates the desired bit-width. The "b" and "h" indicate that the digits that follow are in binary or hexadecimal, respectively. Finally, the digits represent the value of the constant. The BSV compiler will complain if the value is too large to fit into the available size.

[1] What do "left" and "right" mean here? By convention, they mean "towards the most-significant bit" and "towards the least-significant bit", respectively.

[2] For arithmetic right shift, the vacated most-significant bits are filled by replicating the most-significant bit of the original input value in a procedure called sign-extending.

Logical Shift Right by n, a Dynamic Amount: Barrel Shifters

As mentioned before, the logic to support varying shift amounts controlled by a control input is more interesting.

There is an expensive way to do it, relying on only what we have seen so far. We could make a logic block to support varying shift amounts between 0 to $n-1$ positions by instantiating all n possible constant-shifts of the input, each one similar to Fig. 6.8. We then select one of them using an n-way multiplexer similar to Fig. 6.7, controlled by a "shift amount" input `sa`. The constant-shifts incur no gate cost (just wires), but the multiplexer does. See if you can figure out how the cost and critical path scale with n if the $(n+1)$-way multiplexers were to be constructed using 1-bit 2-way multiplexers. Ans: $O(n^2)$ and $O(\lg(n))$. While this would be functionally correct, there is a much more clever structure for this, relying on the binary-number interpretation of the shift-amount.

The shift-amount control input `sa` is given as a $\lceil \lg_2 n \rceil$-bit binary value. We saw in Chapter 3 a binary value can be interpreted as a summation of 2-power values 2^j whenever the j^{th} digit of the value is 1. Analogously, we can shift an input by `sa` positions by conditionally applying a series of shifts by 2^j positions if the j^{th} digit of `sa` is 1. As such, any shift amount between 0 and 31 can be achieved by applying conditionally five 2-power shifts—by 1, by 2, by 4, by 8 and by 16—as needed. If $sa = 5$ or 00101 in binary, the shift could be accomplished by first shifting by 1 position and then shifting the result by 4 positions. If $s = 31$ or 11111 in binary, all 5 shifts are needed.

A hardware interpretation of the above scheme is commonly referred to as a *barrel-shifter*. Generally, if the control input `sa` is of type `Bit#(n)`, we cascade n stages as follows:

- Stage 0: if `s[0]` is 0, pass through, else shift by 1 (2^0)
- Stage 1: if `s[1]` is 0, pass through, else shift by 2 (2^1)
- Stage 2: if `s[2]` is 0, pass through, else shift by 4 (2^2)
- ...
- Stage j: if `s[j]` is 0, pass through, else shift by 2^j
- ...

Each stage is a simple application of constant shifting and multiplexing controlled by one bit of `sa`. The j^{th} stage mux either passes the value through unchanged or shift it by 2^j positions as controlled by `s[`j`]`. Fig. 6.9 illustrates a 4-bit shifter where `sa` of type `Bit#(2)`

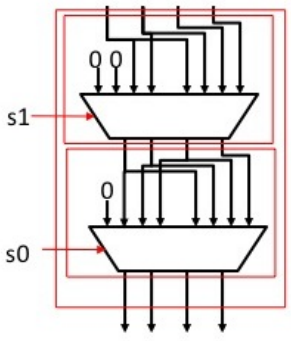


Figure 6.9: Shift by s = 0,1,2,3

can be between 0 and 3. The corresponding BSV function is as follows. (Note: the notation

`x[`j`:`k`]` selects a $(j-k+1)$-bit field between the i^{th} and j^{th} positions inclusive.)

Shift by Variable Amounts

```
function Bit #(4) logical_shift_right_by_n (Bit #(2) s, Bit#(4) arg);
   Bit #(4) temp = (s[0]==1) ? {1'b0,arg[3:1]} : arg ;
   Bit #(4) result = (s[1]==1) ? {2'b00,temp[3:2]} : temp
   return result;
endfunction
```

Other kinds of shifts: arithmetic and rotation

In the discussion so far, we have focus on so-called *logical shifts*, where the vacated bits—most-significant, for right-shifts and least-significant, for left-shifts—are replaced by zeroes.

In a so-called *arithmetic right-shift*, the vacated bits (most-significant) are replaced by copies of the sign bit. When the `Bit#(`w`)`argument is interpreted as signed number, then bit `[`$w-1$`]`, the sign bit, is 0 for positive numbers and 1 for negative numbers. For an arithmetic right-shift, this bit is replicated into the vacated bits.

We do not give any special name to arithmetic left-shifts, since the vacated bits are filled with zeroes, and so this is the same as a logical left-shift.

A *rotate left* operation is like a left-shift, except that the most-significant bits that are shifted out are wrapped around and shifted in at the least-significant end. Symmetrically, A *rotate right* operation is like a right-shift, except that the least-significant bits that are shifted out are wrapped around and shifted in at the most-significant end.

6.2.4 For-loops in BSV to express n-wide repetitive structures

In Sec. 6.2.3 we discussed barrel shifters (shifting by a dynamically-specified amount). We gave an example of a function that performs this shift on a 32-bit value, taking a `Bit#(2)` shift-amount argument; we repeat that code here:

Shift by Variable Amounts

```
function Bit #(4) logical_shift_right_by_n (Bit #(2) sa, Bit#(4) arg);
   Bit #(4) temp = (sa[0]==1) ? {1'b0,arg[3:1]} : arg ;
   Bit #(4) result = (sa[1]==1) ? {2'b00,temp[3:2]} : temp
   return result;
endfunction
```

What if `arg` were wider, say 64 bits? In general `sa` may need to be 6 bits wide, to represent any shift-amount in the range 0..63. Our example could be written as follows:

```
function Bit #(64) logical_shift_right_by_n (Bit #(6) sa, Bit#(64) arg);
   Bit #(64) temp0  = (sa[0]==1) ? (arg   >> 1)  : arg;
   Bit #(64) temp1  = (sa[1]==1) ? (temp0 >> 2)  : temp0;
   Bit #(64) temp2  = (sa[2]==1) ? (temp1 >> 4)  : temp1;
   Bit #(64) temp3  = (sa[3]==1) ? (temp2 >> 8)  : temp2;
```

```
   Bit #(64) temp4  = (sa[4]==1) ? (temp3 >> 16) : temp3;
   Bit #(64) result = (sa[5]==1) ? (temp4 >> 32) : temp4;
   return result;
endfunction
```

where, note, each line only performs a shift by a constant amount. Actually, it's not necessary to define new names (`temp`j) for every intermediate result. We could equivalently have written this code:

```
function Bit #(64) logical_shift_right_by_n (Bit #(6) sa, Bit#(64) arg);
   Bit #(64) result = arg         ;
   result = (sa[0]==1) ? (result >> 1)  : result;
   result = (sa[1]==1) ? (result >> 2)  : result;
   result = (sa[2]==1) ? (result >> 4)  : result;
   result = (sa[3]==1) ? (result >> 8)  : result;
   result = (sa[4]==1) ? (result >> 16) : result;
   result = (sa[5]==1) ? (result >> 32) : result;
   return result;
endfunction
```

In BSV an assignment like this:

`result =` *expression*;

merely says, "from this point onwards, `result` names the wires representing the output of the combinational circuit represented by the right-hand side *expression*." This is unlike most software programming languages where a variable names a memory location, and an assignment means storing the value of the right-hand side expression into the memory location named by the left-hand side variable. The last two examples of BSV code represent exactly the same ciruit.

Not only is the above code rather tedious, but we'd have write another function for shift-amounts of, say, 8 bits, and it would mostly be a copy of this one. The solution is to use a BSV for-loop. We start by just simplifying the previous example, staying with a 6-bit shift amount:

```
function Bit #(64) logical_shift_right_by_n (Bit #(6) sa, Bit#(64) arg);
   Bit #(64) result = arg;
   for (Integer j = 0; j < 6; j = j + 1)
       result = (sa [j]==1) ? (result >> (2**j)) : result;
   return result;
endfunction
```

In BSV, such for-loops are *unrolled completely* by the compiler in a step called *static elaboration*, so that this function is *exactly* equivalent to the previous one, where each shift was

written separately. We still get exactly the same combinational circuit. In particular, the variable j does not exist at all in the final circuit, not even as a wire carrying a value—it is purely a notational device to express the unrolled version succinctly. In other words, there is zero hardware or performance cost to writing it with a for-loop.[3]

Next, let us generalize it to n-bit shift-amounts:

```
function Bit #(64) logical_shift_right_by_n (Bit #(n) sa, Bit#(64) arg);
   Bit #(64) result = arg;
   for (Integer j = 0; j < valueOf (n); j = j + 1)
       result = (sa [j]==1) ? { 0, result [63:2**j]} : result;
   return result;
endfunction
```

In the function argument list, we have replaced `Bit#(6)` by `Bit#(n)`. In the for-loop, we have replaced "6" by `valueOf(n)`. The reason is that `n` here is a *type*, not a *value*. The notation `valueOf(n)` can be read as saying: "interpret the type as a value".

Because for-loops are unrolled completely by the compiler, the loop initial value, loop-termination condition, and loop-increment must all be known at compile-time. Here, even though the shift-amount `sa` is dynamic, its width `n` is known statically, and this is what controls the unrolling. We have a few more remarks on static elaboration in Sec. 6.6 at the end of this chapter.

6.2.5 32-bit Ripple-Carry Adder (RCA)

In Sec. 6.1.2 we saw how to describe a 2-bit ripple-carry adder. We repeat it here for convenience.

2-bit Ripple Carry Adder

```
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
   Bit#(2) s;
   Bit#(2) c_out;

   Bit#(2) cs0 = fa(x[0], y[0], 0);
   c_out[0] = cs0[1];  s[0] = cs0[0];

   Bit#(2) cs1 = fa(x[1], y[1], c_out[0]);
   c_out[1] = cs1[1];  s[1] = cs1[0];

   return {c_out[1],s};
endfunction
```

This example also has a repetitive structure, and so we can apply the idea of for-loops to specify a 32-bit ripple-carry adder in a more compact and easy to understand description.

[3] In software programming languages, a for-loop like this is typically executed dynamically, during program execution, so there can be a performance cost, although some compilers may also try to *unroll* the loop statically for optimization purposes.

32-bit Ripple Carry Adder

```
function Bit#(33) add32 (Bit#(32) x, Bit#(32) y);
   Bit #(32) s;
   Bit #(32) c_out;
   Bit #(2)  cs;
   Bit #(1)  c_in=0;
   for (Bit #(6) i=0; i<32; i=i+1) begin
      cs = fa (x[i],y[i],c_in);
      c_out[i] = cs[1];  s[i] = cs[0];
      c_in=c_out[i];
   end
   return {c_out[31],s};
endfunction
```

The first thing to notice is that when written with a for-loop, the adder code for 32-bit—or any other width for that matter—is not more lengthy than the code for the 2-bit adder. Moreover, with less redundancy, the version using a for-loop is easier to understand and it is arguably harder to make a mistake. As mentioned earlier, during compilation, the for-loop will be statically elaborated by the compiler. The end effect is the same as if the for-loop is unrolled 32 times.

32-bit Ripple Carry Adder with Unrolled Loop Body

```
function Bit#(33) add32 (Bit#(32) x, Bit#(32) y);
   Bit #(32) s;
   Bit #(32) c_out;
   Bit #(2)  cs;
   Bit #(1)  c_in=0;

   // i=0
   cs = fa (x[0],y[0],c_in);
   c_out[0] = cs[1];  s[0] = cs[0];
   c_in=c_out[0];
   // i=1
   cs = fa (x[1],y[1],c_in);
   c_out[1] = cs[1];  s[1] = cs[0];
   c_in=c_out[1];
   . . . .
   // i=31
   cs = fa (x[31],y[31],c_in);
   c_out[31] = cs[1];  s[31] = cs[0];

   return {c_out[31],s};
endfunction
```

Notice in both versions, `cs` and `c_in` are assigned repeatedly. As explained in Sec. 6.2.4, when reading the function as a circuit diagram description, each use of `fa` corresponds to a different instantiation of the combinational logic it represents. (If we liked, we could have

explicitly "inline" the `fa` code into the unrolled code as would be done in static elaboration.) Furthermore, a variable when assigned serves as a reference for the wire at the output of the combinational logic on the right-hand side of the assignment. When a variable is reassigned, subsequent use of the variable refers to the new wire. Convince yourself both versions of the code above describe thirty-two `fa`'s connected by a ripple carry chain as depicted in Figure 6.1.

6.2.6 Multiplier

Multiplication is not much harder to describe than addition, though it will turn out to be much more expensive in hardware. First let's begin by reviewing how to multiply. Suppose we are multiplying two 4-bit values 1101 and 1011. Our grade-school multiplication algorithm (translated from decimal to binary) looks like this:

Multiplication by repeated addition

```
1          1 1 0 1    // Multiplicand, b
2          1 0 1 1    // Multiplier,   a
3          -------
4          1 1 0 1    // b x a[0] (== b), shifted left by 0
5        1 1 0 1      // b x a[1] (== b), shifted left by 1
6      0 0 0 0        // b x a[2] (== 0), shifted left by 2
7    1 1 0 1          // b x a[3] (== b), shifted left by 3
8  ---------------
9  1 0 0 0 1 1 1 1
```

Thus, the j^{th} partial sum is `(a[j]==0 ? 0 : b) << j`, and the overall sum is just a for-loop to add these sums. The outcome of the multiplication is 8 bits ($= 4 + 4$, the sum of the widths of the arguments). Fig. 6.10 illustrates the above scheme in combinational logic

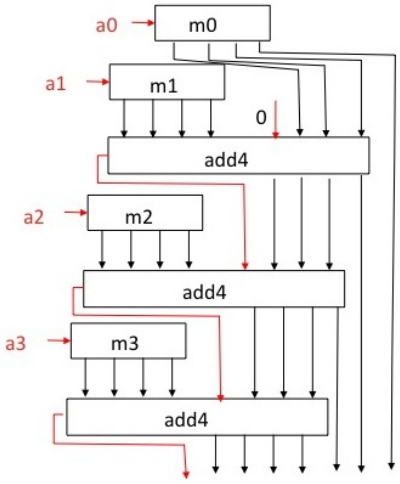


Figure 6.10: A combinational multiplier

circuits. The BSV code to express the circuit in shown below:

BSV code for combinational multiplication

```
function Bit#(8) mul4(Bit#(4) a, Bit#(4) b);
   Bit#(4) prod = 0;
   Bit#(4) tp = 0;
   for (Bit#(3) i=0; i<4; i=i+1) begin
      Bit#(4) m = (a[i]==0)? 0 : b;
      Bit#(5) sum = add4(m,tp,0);
      prod[i] = sum[0];
      tp = sum[5:1];
   end
   return {tp,prod};
endfunction
```

Take a moment to double check that the `mul4` function above indeed describes the circuit in Fig. 6.10. As in the 32-bit adder example, we can adjust the 4-bit multiplier to any desired width n by trivial substitutions. The result will lead to n^2 `fa` being instantiated. You need to be aware of what you are describing in circuits terms. A little bit of code can result in a lot of logic! Another interesting aside is to ask what is the critical input-to-output path delay of an n-by-n multiplication. In a literal reading of the BSV code, we would see that we are doing $O(n)$ additions which each takes $O(n)$ time using ripple carry. However, if we remember we are actually describing a combinational logic circuit where the operations of the adders are not serialized completely. Each `fa` begins its final evaluation as soon as its inputs are stable. The least significant `fa` of the last adder begins after the output is stable on the least significant `fa` of the second-to-last adder; this is about the same time as when the ripple-carry-chain of first adder is fully finished. All in all, no input-to-output paths involve traversing more than $2n$ `fa`'s.

6.2.7 ALU (Arithmetic-Logic Unit)

Datapath designs, particularly in computer CPUs, sometime call for a single Arithmetic and Logical Unit (ALU) that performs all the additions, subtractions, multiplications, ANDs, ORs, comparisons, shifts, and so on. Fig. 6.11 shows the symbol commonly used to represent

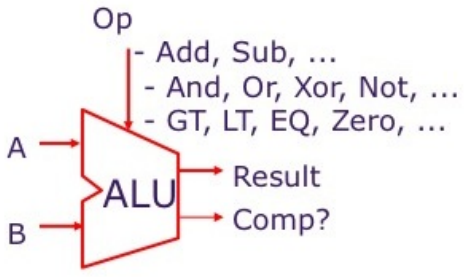


Figure 6.11: ALU schematic symbol

ALUs in circuit schematics. It has data inputs A and B, and an Op input by which we select the function (Add, subtract, logical AND/OR/NOT, comparisons, ...) performed by the ALU. It has a Result data output (and sometimes a separate output `Comp?` for results of comparisons).

BSV has built-in support for the operations we discussed above. In practical use, it often best to make use of BSV's built-in operators. We could have just as easily have said `s=x+y` or `s=x*y`. The compiler and synthesis tools will produce a working design. We do not know how the addition is actually done; we are then relying on the compiler and synthesis tools to make a good choice among the many different adder circuits with wide varying trade-offs. We see how to describe an ALU using BSV built-in operators below.

ALU for Arith, Logic, Shift ops

```
function Bit#(32) alu (Bit#(32) a, Bit#(32) b, Bit#(4) func);
  Bit#(32) res = case(func)
     0   : (a + b);
     1   : (a - b);
     2   : (a & b);
     3   : (a | b);
     4   : (a ^ b);
     5   : ~(a | b);
     6   : (a << b[4:0]);
     7   : (a >> b[4:0]);
     8   : signedShiftRight(a, b[4:0]);
  endcase;
  return res;
endfunction
```

Above, the output of the `alu` is a function of the inputs `a` and `b`. The 4-bit control input `func` selects one of the 9 functions; encodings 10 to 15 are unused. In the example, `(a+b)` could be replaced by `add32(a,b)` to instantiate our own adder design. Notice `>>` on `Bit#(`m`)` signifies a logical-right-shift. Arithmetic right shift is performed through the function `signedShiftRight()` left unspecified.

6.3 Types and Type-checking in BSV

BSV is a strongly typed language, following modern practice for robust, scalable programming. All expressions (including variables, functions, modules and interfaces) have unique types, and the compiler does extensive type checking to ensure that all operations have meaningful types. Type-checking in BSV is much stronger than in languages like C and C++. For example, in C or C++ one can write an assignment statement that adds a `char`, a `short`, a `long` and a `long long` and assign the result to a `short` variable; the compiler silently "casts" each right-hand side expression's result to a different (typically wider) type, performs the operation, and then silently casts the result value to the type required by the left-hand side (typically narrower). This kind of silent type "casting" (with no visible type checking error) is the source of many subtle bugs. In hardware, where we use values of many different bit-widths (not just 8, 16, 32, 64), these buts can be even more deadly. In BSV, type casting is never silent, it must be explicitly stated by the user in the code.

The central purpose of a strong static type system, whether in software programming languages or in BSV is to keep us (fallible humans) verifiably honest, so that we do not accidentally *misinterpret* bits, which ultimately are used to represent everything. For example

if a certain 64 bits is used to represent an "employee record", it makes no sense to apply the "square root" function to those bits, *i.e.,* to misinterpret those bits as if they represented a number.

A full treatment of the BSV type system is not appropriate in this book. Below, we will introduce the most important basics to make use of types productively in combinational logic descriptions. Additional topics will be introduced with sequential logic descriptions.

6.3.1 Types

As motivated in Section 6.1.1, instead of working with individual wires or bits, it is convenient to group multiple conceptually related wires (*e.g.,* those that together convey a binary value) to handle them as one entity. So far we have limited ourselves to working with the type `Bit#(`w`)`, quite literally a ordered-bundle—a vector—of wires. Already, there is a lot more to it.

Associated with a type are the supported operations on them. We have been manipulating variables and expressions of type `Bit#(`w`)` with logic operators (`&`, `|`, `^`, etc.), shifts (`>>` and `<<`), and concatenation (`{...}`). Toward the end of Section 6.2.5 we revealed the fact that you can actually perform arithmetic on `Bit#(`w`)` with built-in operators (`+`, `-`, `*`, etc.), interpreting the variables and expressions as unsigned binary values of width w.

As you become more advanced, you can learn on your own to use `Int#(`w`)` and `UInt#(`w`)` in BSV. They represent signed and unsigned integers, respectively, that can be represented in w bits. In addition to the normal arithmetic operators, you can also apply `negate` to a signed integer to invert its sign.

There is a Boolean type `Bool` which can take on 2 values (`True` or `False`). It has the expected built-in logic operators, `&&` ("AND") and `||` ("OR"), as well as unary negation using either `!` or `not`. In the BSV type system, `Bool` is not synonymous with `Bit#(1)`; you cannot operate on `Bool` variables and `Bit#(1)` variables together.

In Sec. 6.1 we were introduced to general vector types. For example, `Vector#(16,Bit#(8))` is a vector of size 16 containing `Bit#(8)`'s, and this is distinct from the type `Vector#(16,Bool)`, even though the underlying representation may occupy the same number of bits. Similarly, `Vector#(16,Bit#(8))` and `Vector#(8,Bit#(16))` represent distinct types, even though the underlying representation may occupy the same number of bits.

Certain types, such as `Bool` or `String` are straightforward. A type such a `Bit#(`w`)` has more structure. The symbol `#` indicates that `Bit` is parameterized. We need to specify a width to make a variable of this type concrete for synthesis, e.g., `Bit#(8)`, `Bit#(32)`, etc.

`Vector` is another example of a parameterized type, in this case taking two parameter types, one specifying the vector size, and the other specifying the type of the contents of each vector element. For example, `Vector#(16,Bit#(8))` is the type of vectors of 16 elements, each of which is a `Bit#(8)` value. As this example shows, types can be nested; thus `Vector#(16,Vector#(8,Bit#(2)))` represents:

A vector of 16 elements, each of which is
    a vector of 8 elements, each of which is
        a 2-bit value

Another useful example to learn is `Tuple2#(`$t1$`,`$t2$`)` which allows bundling of two arbitrary types into a new type, e.g., `Tuple2#(Bit#(8), Bool)`. These are useful for functions that return multiple results, to bundle them up in to a conceptually single entity. For those familiar with "struct" types in C or C++, a tuple is simply kind of struct where the fields/members are not explicitly named. BSV provides tuples of width 2,3,4,...

6.3.2 Type Synonyms

For types that are used frequently, we can create *type synonyms* as nicknames for them. We often do this to improve the readability of BSV code. As a rule, in BSV, a type name always begins with a capital letter, while a variable identifier begins with a lower-case letter.

Type Synonym Examples

```
1   typedef Bit#(8) Byte;
2
3   typedef Bit#(32) Word;
4
5   typedef 32 DataSize;
6
7   typedef Bit#(DataSize) Data;
8
9   typedef Tuple2#(a,a) Pair#(a);
10
11  typedef Pair#(Word) Cartesian;
```

Types and type synonyms can be parameterized by type variables. In the example, the type synonym `Pair#(a)` is parameterized by a type variable `a` to be specified only when it is used. A subtle issue to note is that, in BSV, the type language (which describes the "shape" of values) is completely separate from the language for hardware description, which describes values (carried on wires). In the above examples, the "8" and "32" are called *numeric types* because they are used as type parameters, even though they use the same notation as *numeric values*. We will say more about this shortly.

6.3.3 Enumerated Types

Enumerated Types are a very useful typing concept. Suppose we want a variable `c` to represent three different colors: Red, Green, Blue. With only what we know, we could declare the type of `c` to be `Bit#(2)` and impose a convention that 00 represents Red, 01 Blue, and 10 Green. The problem is that these two-bit values could be accidentally mixed in expressions with other two-bit values that have nothing to do with color, such as a hotel rating of Good, Better and Best. Further, operations like AND, OR, NOT, shift, *etc.* make no sense for this type.

Alternatively, we can create a new *enumerated* type just for this:

```
typedef enum {Red, Blue, Green} Color;
```

With this, we can declare a variable of type `Color` and assign the values `Red`, `Green`, `Blue` to it. One cannot perform logical or arithmetic operations on it, nor mix it with other enumerated types that may happen to be represented in 2 bits. Further, it makes our code more readable because we use the symbolic constants `Red`, `Green`, `Blue` for the values, instead of 2-bit constants `2'b00`, `2'b01`, `2'b10`.

6.3.4 Deriving Eq, Bits and FShow

Whenever we define a new type like `Color`, we almost always append this common "mantra" to the declaration:

```
typedef enum {Red, Blue, Green} Color
deriving (Eq, Bits, FShow);
```

The first element in the parentheses, `Eq`, tells the BSV compiler please to define the equality and inequality operators for this type automatically. Now we can write things like this:

```
1  Color c1, c2, c3;
2  c1 = Red;
3  c2 = Blue;
4  ...
5  c3 = c2;
6  ...
7  if (c1 == c2)
8      ...
9  else if (c2 != c3) ...
10     ...
```

In the first few lines, we declare variables `c1`, `c2` and `c3` of type `Color`, and perform some assignments on these using the symbolic constants `Red` and `Blue`. In the conditional statements, we are using the equality (`==`) and inequality (`!=`) operators on these variables, which are meaningful because of the `deriving(Eq)` clause.

The second element in parentheses, `Bits`, tells the BSV compiler please to choose the bit representations of the symbolic constants in the obvious way, namely, please use the minimum necessary bits to encode these values (in this case, 2 bits) and, further, encode them sequentially in the natural way: `Red` as 00, `Blue` as 01, and `Green` as 10.

The third element in parentheses, `FShow`, tells the BSV compiler please to define automatically a function that produces a natural string representation of the values to be used in `$display` statements. For example, if we were to write:

```
$display (c1,         ",", c2,         ",", c3);
$display (fshow (c1), ",", fshow (c2), ",", fshow (c3));
```

then, when we execute the code in a simulator, this would print the following two lines on the terminal:

0, 1, 1
Red, Blue, Blue

The reason that the compiler requires the BSV programmer to write the mantra:

```
deriving (Eq, Bits, FShow);
```

instead of just assuming it exists is that actually BSV provides complete flexibility in how one defines equality on a new type, on how one represents it in bits, and on how it is printed. This mantra is just a shorthand convenience to tell the compiler please to define these facilities for me in the obvious way.

In some other situation, the programmer may decide to use a so-called "one-hot" encoding for `Color` values, using 3 bits and encoding the values as 001, 010 and 100, respectively. BSV provides this power, but that is an advanced topic not relevant for this book.

Another example: we could declare an enumerated type to designate the functions performed by an ALU.

Enumerated types for ALU opcodes

```
1
2 typedef enum {Add, Sub, And, Or, Xor, Nor, LShift, RShift, Sra} AluFunc
3         deriving (Bits, Eq);
```

Compared with our first ALU example, using `AluFunc` instead of `Bit#(4)` for the control input `func` eliminates the error when a designer uses the ALU with the wrong encoding and makes it easier to add or remove operations later on.

ALU for Arith, Logic, Shift ops

```
1  function Bit#(32) alu (Bit#(32) a, Bit#(32) b, AluFunc func);
2    Bit#(32) res = case(func)
3       Add   : (a + b);
4       Sub   : (a - b);
5       And   : (a & b);
6       Or    : (a | b);
7       Xor   : (a ^ b);
8       Nor   : ~(a | b);
9       LShift: (a << b[4:0]);
10      RShift: (a >> b[4:0]);
11      Sra   : signedShiftRight(a, b[4:0]);
12   endcase;
13   return res;
14 endfunction
```

6.4 Parameterized Description (Advanced)

We will wrap up this chapter by generalizing the 32-bit ripple-carry adder to a w-bit ripple-carry adder. We begin with a not quite correct attempt by replacing the value "32" by the type variable "`w`" in the 32-bit ripple carry adder code from Section 6.2.5. We assume there is another type variable `lg2w` that is $\lceil \lg_2 \texttt{w} \rceil$.

w-bit Ripple Carry Adder (not quite right)

```
function Bit#(w+1) addN (Bit#(w) x, Bit#(w) y);
   Bit#(w) s;
   Bit#(w) c_out;
   Bit#(2) cs;
   Bit#(1) c_in=0;
   for(Bit#(lg2w+1) i=0; i<w; i=i+1) begin
      cs = fa (x[i],y[i],c_in);
      c_out[i] = cs[1];  s[i] = cs[0];
      c_in=c_out[i];
   end
   return {c_out[w-1],s};
endfunction
```

Mentally, we should see that by replacing `w` with a specific value, e.g., 32 or 2, the above function would describe an ripple carry adder of that width. Concretely, we can instantiate adders of specific sizes as follows.

w-bit Ripple Carry Adder ()

```
function Bit#(33) add32 (Bit#(32) x, Bit#(32) y)
    = addN (x,y);

function Bit#(3) add2 (Bit#(2) x, Bit#(2) y)
    = addN (x,y);
```

This defines `add32` and `add2` on 32-bit and 2-bit values, respectively. In each case the specific function just *calls* `addN`, but the compiler's type deduction will fix the value of w for that instance, and type checking will verify that the output width is 1 more than the input width. Alas, we have what we wanted, except for a small notational issue with the `addN` function given above.

The BSV compiler, like most compilers, is composed of several distinct phases such as parsing, type checking, static elaboration, analysis and optimization, and code generation. Type checking must analyze, verify and deduce numerical relationships, such as the bit widths in the example above. However, we cannot have arbitrary arithmetic in this activity since we want type-checking to be feasible and efficient and decidable at compile-time (arbitrary arithmetic would make it undecidable). Thus, the numeric calculations performed by the type checker are in a separate, limited universe, and should not be confused with ordinary arithmetic on values.

In the type expressions `Bit#(1)`, `Bit#(2)`, `Bit#(33)`, the literals `1`, `2`, `33`, are in the type universe, not the ordinary value universe, even though we use the same syntax as ordinary numeric values. The context will always make this distinction clear—numeric literal types only occur in type expressions, and numeric literal values only occur in ordinary value expressions. Similarly, it should be clear that a type variable like `w` in a type expression `Bit#(w)` can only stand for a numeric type, and never a numeric value.

Since type checking (including type deduction) occurs before anything else, type values are known to the compiler before it analyses any piece of code. Thus, it is possible to take

numeric type values from the types universe into the ordinary value universe, and use them there (but not vice versa). The bridge is a built-in pseudo-function called `valueOf(n)`. Here, `n` is a numeric type expression, and the value of the function is an ordinary `Integer` *value* equivalent to the number represented by that type.

Numeric type expressions can also be manipulated by numeric type operators like `TAdd#(t1,t2)` and `TMul#(t1,t2)`, corresponding to addition and subtraction, respectively (other available operators include min, max, exponentiation, base-2 log, and so on).

With this in mind, we can fix up our w-bit ripple carry adder code:

w-bit Ripple Carry Adder (corrected)

```
1  function Bit#(TAdd#(w,1)) addN (Bit#(w) x, Bit#(w) y);
2     Bit#(w) s;
3     Bit#(w) c_out;
4     Bit#(2) cs;
5     Bit#(1) c_in=0;
6     for(Integer i=0; i<valueOf(w); i=i+1) begin
7        cs = fa (x[i],y[i],c_in);
8        c_out[i] = cs[1];  s[i] = cs[0];
9        c_in=c_out[i];
10    end
11    return {c_out[valueof(w)-1],s};
12 endfunction
```

The differences from the earlier, almost correct version is that we have used `TAdd#(w,1)` instead of `w+1` in the type specification in lines 1, and we have used `valueOf(w)` in line 6 and line 11 to the type variable `w` in the circuit description. In the corrected version, we also made use of the special value type `Integer` for the loop counter `i` in line 6. `Integer` is only available as a static type for static variables that are used at compile time. (Recall, for-loop used in this way is evaluated by the compiler during static elaboration; no notion of `i` or "looping" remain in the combinational logic that results afterwards.) Semantically, `Integer` variables are true mathematical unbounded integers, not limited to any arbitrary bit width like 32, 64, or 64K (of course, they're ultimately limited by the memory of the system your compiler runs on). For dynamic integer values that maps to hardware, they must be typed with a certain definite bit width.

6.5 Takeaways

We have only seen the tip of the BSV iceberg. There is more to come in association with the design and description of sequential logic. There is also much sophistication in BSV beyond the scope of this book. Nevertheless, we have seen enough of the basics to accomplish many useful, even powerful, designs. At this point, there is no combinational logic circuits you can conceive in your mind that you cannot put down in BSV code to simulate and to synthesize.

To review, we learned how to use modern high-level programming constructs to describe combinational circuits in a special usage of BSV functions. As in software engineering, these functions enable a modular design methodology where we can reuse a design and we can hierarchically build-up to complex designs. The *piece de resistance* is the final parameterized

ripple-carry adder circuit description. As an exercise, you should create parameterized versions of the other examples we visited in this chapter.

We learned how to make use of types and type checking to prevent us from connecting functions and gates in illegal ways. Many users of BSV have found this especially helpful in flagging out subtle errors that only manifest in rare corner cases (e.g., inadvertent overflow or wraparound of values with mismatched bit width). When starting out, you are likely to find the type system unwieldy. The best way to learn about types is to try writing a few expressions and feeding them to the compiler.

It is important to appreciate the true nature of what we are describing, especially keeping in mind that all loops are unfolded and functions are in-lined during compilation. We will see much more of this later in Chapter 10.

6.6 Incredibly powerful static elaboration (Advanced)

(This section can be safely skipped without loss of continuity.)

6.6.1 Recursion

The for-loops introduced in Sec 6.2.4 are an example of using a high-level programming abstraction (loops) to specify certain repetitive hardware structures in a succinct and readable manner.

For example, consider this function to add all elements of a vector of n elements, each being a w-bit number (this function does not take care of overflow beyond w bits):

```
1 function Bit #(w) add_all (Vector #(n, Bit #(w)) vec);
2    Bit #(w) result = 0;
3    for (Integer j = 0; j < valueOf (n); j = j + 1)
4       result = result + vec [j];
5    return result;
6 endfunction
```

This would be unfolded into a linear array of w-bit adders, with the j'th adder adding `vec[j]` to the output of the previous adder. The 0'th adder's "previous output" is 0, and the $(n-1)$'th adder's result is the final output of the function.

Here is a different implementation of this function, using recursion instead of loops, in a binary divide-and-conquer fashion:

```
1 function Bit #(w) add_all (Vector #(n, Bit #(w)) vec);
2    return add_all_in_range (vec, 0, valueOf (n-1));
3 endfunction
4
5 function Bit #(w) add_all_in_range (Vector #(n, Bit #(w)) vec,
6                                     Integer i,
```

```
7                                       Integer j);
8     Bit #(w) result;
9     if (i == j)
10       result = vec [j];
11    else begin
12       Integer k = (i + j) / 2;
13       result = add_all_in_range (i, k) + add_all_in_range (k+1, j);
14    end
15    return result;
16 endfunction
```

The auxiliary function `add_all_in_range` returns the sum of all elements `vec[i]`, `vec[i+1]`, ... `vec[j]`. Suppose n was 8. The initial call to `add_all_in_range` has arguments `(vec, 0, 7)`. This performs two recursive calls:

```
add_all_in_range (vec, 0, 3)
+ add_all_in_range (vec, 4, 7)
```

which, in turn, results in the calls:

```
  (add_all_in_range (vec, 0, 1)
   + add_all_in_range (vec, 2, 3))
+ (add_all_in_range (vec, 4, 5)
   + add_all_in_range (vec, 5, 7))
```

which, in turn, results in the calls:

```
    ((add_all_in_range (vec, 0, 0)
      + add_all_in_range (vec, 1, 1))
  +  (add_all_in_range (vec, 2, 2)
      + add_all_in_range (vec, 3, 3)))
+   ((add_all_in_range (vec, 4, 4)
      + add_all_in_range (vec, 5, 5))
  +  (add_all_in_range (vec, 6, 6)
      + add_all_in_range (vec, 7, 7)))
```

Each of these, finally, hit the `(i==j)` condition and so there are no more recursive calls, so this produces:

```
    ((vec [0]
      + vec [1])
  +  (vec [2]
      + vec [3]))
+   ((vec [4]
      + vec [5])
  +  (vec [6]
      + vec [7]))
```

Notice that, while the previous function unfolded into a linear chain of adders, this function unfolds into a tree-like arrangements of adders. While the longest-path delay in the previous function was proportional to n, in this case it is proportional to $\log n$.

6.6.2 Higher-order functions

Suppose we wanted to compute the "AND" of all `Bit#(w)` elements in a vector. Or the "OR", maximum, minimum, *etc.* Each of these functions would look very similar to the `add_all` function in the previous section, the only difference being the 2-input operator we applied inside the function (`+`, `&`, `|` ...).

In BSV, we can write a single parameterized function to capture this "design pattern":

```
typedef
function Bit #(w) op_all (Vector #(n, Bit #(w)) vec,
                          function Bit #(w) op (Bit #(w) x, Bit #(w) y),
                          Bit #(w) init);
   Bit #(w) result = init;
   for (Integer j = 0; j < valueOf (n); j = j + 1)
      result = op (result, vec [j]);
   return result;
endfunction
```

This is just a generalization of the `add_all` function (the version using a loop; we could also have done it with the recursive version). We've changed the name of the function to `op_all` and added two more arguments. The first argument of `op_all` is the vector of values, as before. The new second argument, called `op`, is declared like this:

```
function Bit #(w) op (Bit #(w) x, Bit #(w) y)
```

indicating that it is itself a function, taking two `Bit#(w)` arguments and returning a `Bit#(w)` result. The third argument of `op_all` is a `Bit#(w)` value called `init`, representing the initial value of the accumulated result. For example, if `op` were the addition function, then `init` should be 0, and `op_all` will add all the elements. If `op` were the multiplication function, then `init` should be 1, and the function will multiply all the elements. If `op` were the "AND" function (`&`), then `init` should be all 1s, and the function will AND all the elements. If `op` were the "OR" function (`|`), then `init` should be all 0s, and the function will OR all the elements.

Our `add_all` function can then be defined succinctly as follows. We also show the other functions mentioned in the previous paragraph, using even more succinct notation:

```
function Bit #(w) add_all (Vector #(n, Bit #(w)) vec);
    return op_all (vec, \+ , 0);
endfunction

```

```
5 function Bit#(w) mult_all(Vector#(n, Bit#(w)) vec) = op_all (vec, \* , 1);
6 function Bit#(w) and_all (Vector#(n, Bit#(w)) vec) = op_all (vec, \& , '1);
7 function Bit#(w) or_all  (Vector#(n, Bit#(w)) vec) = op_all (vec, \| , 0);
```

The notation "`\+ `" (note the trailing space) allows us to disable the normal syntactic role of `+` as an infix operator expecting argument expressions on either side, and to treat it just as a 2-argument function. Similarly, "`\* `", "`\& `" and "`\| `" allows us to treat `*`, `&`, `|` as 2-argument multiplication, AND and OR functions, respectively.

Also, in the latter three definitions, instead of enclosing the function body with usual `endfunction`, we use the alternative form "= *expression*" (we normally use this alternative form only for very short function definitions).

The bottom line is, *we are parameterizing one combinational circuit structure with another combinational circuit*! The function `op_all` describes a combinational circuit structure that is common to the problem of adding, multiplying, ANDing or ORing a vector of elements. Into this common structure, wherever `op` is used, we plug in another specific combinational circuit, for addition, multiplication, ANDing or ORing. In programming language theory, a function whose argument (or result) can itself be a function is called a *higher-order function*. This capability can be found in many modern programming languages such as Python, Haskell, OCaml, Scala, *etc.* and, in a less disciplined way, even in C (function pointers).

6.6.3 Summary

In the full adder `fa` and 2-bit adder `add2` examples at the beginning of this chapter, we saw how combinational circuits can be described using BSV functions with full specificity and literal correspondence to a circuit diagram. Those simple functions are almost just a direct textual rendering of wiring diagrams, directly describing how primitive gates are connected with wires. A variable like `s` or `c_out` when assigned is just a name for the wires that are the output of the combinational circuit represented by the expression in the right-hand side of the assignment.

In this section we showed much more powerful uses of functions, where we rely on the compiler to "unfold" functions in sophisticated and intricate ways (loops, recursion, higher-order functions). However, we reach the same endpoint: a combinational circuit, which is just an acyclic interconnection of primitive gates. BSV functions can be seen as just a very powerful abstraction mechanism to describe combinational circuits. This can be put to powerful uses in the hands of an advanced BSV user.

The above notwithstanding, readers with programming background should have also realized that all the the functions we have described have perfectly sensible readings in the normal programming sense, where a function computes a return value based on the input arguments. This dual interpretation holds for all uses of BSV functions to describe combinational circuits. Indeed, when BSV is executed in a simulator, it is executed using the traditional computation interpretation, not in terms of AND and OR gates. The abstract perspective associated with a computational reading can be beneficial when working with more elaborate designs. As you become experienced with logic design and with BSV, you will find it useful to be able to move between the programming and circuits interpretations

as convenient. Initially though, you should first work to become practiced in thinking in terms of circuit description as this awareness is essential to producing high-quality circuits (in terms of size, speed, energy, etc.) over and above one that is merely functionally correct.

Chapter 7

Sequential (Stateful) Circuits

7.1 Introduction

So far, we have described combinational circuits which, in the end, are just acyclic interconnections of basic gates such as AND, OR or NOT gates. Quite complex structures can be created this way, and there is some common symbology used to depict such structures. Fig 7.1 shows some well-known combinational structures. At the top left is an n-input *multiplexer*, or "Mux" for short. It takes n inputs A_0, ..., A_{n-1} of a common width w, and one more input *Sel* of width $\log_2 w$. When *Sel* carries the value j, the output O of the circuit, also of width w, is equal to A_j.

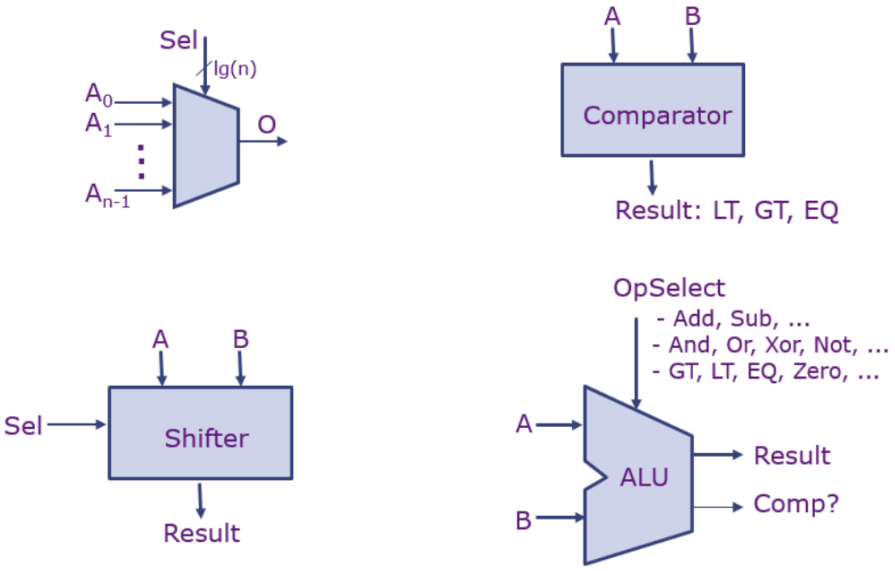


Figure 7.1: Common combinational circuits

At the top right we have a *comparator* that takes two inputs A and B of equal width w, and returns a 2-bit output result indicating whether $A < B$, $A = B$ or $A > B$. Some comparators interpret A and B as unsigned quantities, while some interpret them as signed (*e.g.,* 2's complement) quantities.

At the bottom left we have a *Shifter* that takes an input A of width w and another input B of width $\log_2 w$. The *Sel* input indicates the type of shift (left, right, logical, arithmetic). When B carries value j, the output result is the the value of A shifted by j bits.

The bottom right structure is the most complex. It is an ALU, or "Arithmetic and Logic Unit", and is the heart of any computer processor's CPU. It takes two inputs A and B, usually of some common width w, and another input *OpSelect* which is a code indicating which computation is desired: Addition, Subtraction, And, Or, Xor, Not, comparison, and so on. The *Result* output carries the result of the desired computation on A and B. Some ALUs have a separate *Comp?* output for the results of comparison operations. You can think of an ALU as a collection of circuits, one each to do addition, subtraction, And, Or, Not, comparison, *etc.* followed by a Mux driven by *OpSelect* to choose one of the results.

7.1.1 Sequential circuits

In this chapter we extend our notion of circuits from combinational circuits to *sequential circuits*. A combinational circuit has no "memory" of past inputs (we also say that it has no *state*). Each output wire, and indeed each internal wire, is a direct (boolean) function of the *current* inputs of the circuit (not counting any brief transitory period after inputs change, when changed signals may still be propagating through the circuit). Sequential circuits contain a second class of building blocks which we call *state elements*, *i.e.,* elements with "memory". The values carried by wires in such a ciruit depend on the past history of signals, because each state element "remembers" some past input. We thus call such circuits *sequential* circuits—their behavior is defined *temporally*, *i.e.,* relative to the sequence of signals in time.

7.2 The Edge-Triggered D Flip-Flop with Enable: a Fundamental State Element

The most commonly used state element is the "Edge-Triggered D Flip Flop with Enable". In the next few subsections we build up to this gradually from combinational circuits.

7.2.1 Combinational circuits with feedback: the D Latch

What happens if we deliberately break the rules of combinational circuits and introduce a cycle, *i.e.,* we add a connection so that the output of a gate feeds back to an input of the same gate, either directly or indirectly via other gates? Fig 7.2 shows two examples, using inverters (NOT gates). The first one (and in fact any cycle with an even number of inverters), is "stable" in one of two possible states. If you follow either the green or the red 0s and 1s around the cycles, each inverter seems to be doing the right thing, *i.e.,* inverting its input.

If it was in the state indicated by the green labels and we wanted to flip it to the state indicated by the red labels, how would we do it? If we could do that at will, we essentially would have a circuit that "remembers" the last flip (or flop) indefinitely. We also say that it has "memory" or "state".

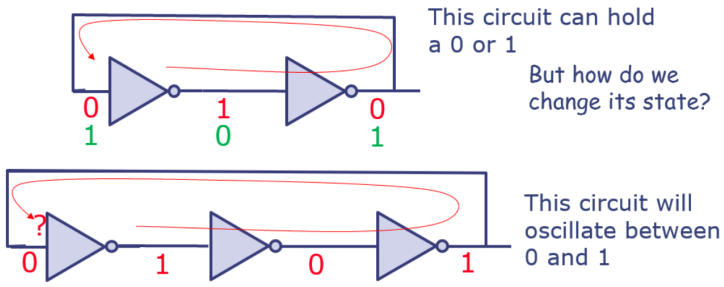


Figure 7.2: Circuits with cycles (not combinational)

In the second circuit, (and in fact any cycle with an odd number of inverters), we can find no stable state where every gate is inverting correctly, like the green and red labels of the first circuit. To explain what this circuit does we have to drop below the digital abstraction level and appeal to reasoning about the underlying analog circuits. Once we do that, and take into account the fact that the circuits have a propagation delay, we will conclude that this circuit will oscillate forever. Suppose we momentarily somehow forced the input of the left-most gate to zero. After three gate delays, it emerges on the right as 1. But this drives the input of the left-most gate so, after a gate delay, its output which was 1 becomes 0. A gate delay later, the middle gate's output, which was a 0, becomes 1. A gate delay later, the right-most gate's output which was a 1, becomes a 0, And this cycle repeats forever. The frequency of this oscillation of course depends on the gate delays and the number of gates in the odd-sized cycle.

Fig 7.3 shows a useful structure, well-known as the D Latch. It is based on a 2-input

Figure 7.3: The D Latch

mux (multiplexer); the mux input is selected using the signal `C`. When `C` is 0, the input D is selected and passed through to the `Q` output, and there is no cycle. When `C` transitions to 1, it selects the other input of the mux, which is the current value of Q, and this is a stable cycle. As long as `C` remains 1, `Q` will remain at that value, *even if* `D` *changes!*. In effect, the D Latch "remembers" what was the value of `D` when `C` transitions from 0 to 1 *no matter how long ago that transition occurred!* The D Latch gives us a usable memory cell! The `D` input is the value we remember, and the `C` input allows us to control when we want to "sample" `D`.

7.2.2 Cascaded D Latches give us an Edge-Triggered D Flip-Flop

Fig 7.4 shows two D latches cascaded in a certain way: they have the same `C` input, but

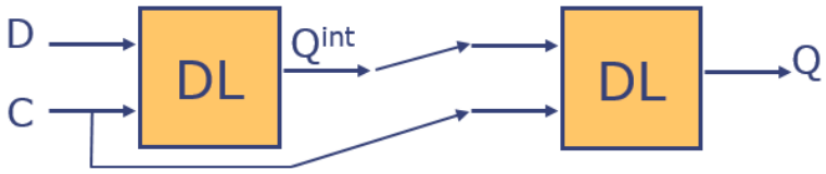


Figure 7.4: Two cascaded D Latches

the `Q` output of the first one feeds the `D` input of the second one. When `C` is 0, both latches *pass*-through their input signals, so `D` on the far left is passed all the way through to the `Q` on the far right. When `C` is 1, both latches *hold* their values, which is the just the most recent value of the leftmost `D` when `C` went from 0 to 1. Thus, they just behave like a single D latch.

Exercise 7.1: What happens if `C` oscillates with a time period comparable to the propagation delay of a single D latch? □

Fig 7.5 shows a slight variation: the `C` input of the second D latch is inverted from the `C`

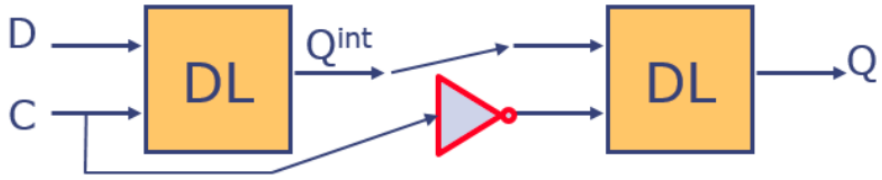


Figure 7.5: Two cascaded D Latches with inverted `C` inputs

input of the first one. When `C` is 0, the first latch is passing, so $\texttt{Q}^{int}$ (for "internal") follows `D`, and the second latch is holding. When `C` is 1, the first latch is holding $\texttt{Q}^{int}$, and the second latch is passing, so `Q` is equal to $\texttt{Q}^{int}$. In short, `Q` will not change when `C` is 0 or 1, but it can change only when `C` transitions from 0 to 1 (the so-called *rising edge* of `C`).

This structure is usually packaged as a basic building block, called a "D Flip-Flop", and has its own symbol, shown in Fig7.6. In pictures, the input with the little triangle always refers

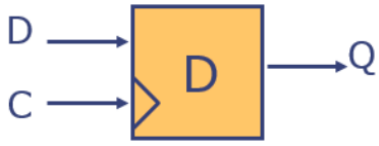


Figure 7.6: Symbol for an Edge-Triggered D-Flip Flop

to a *clock* input—it is the signal that controls (by its transitions) when data is "clocked in" to the D flip-flop. Fig7.7 describes the behavior of a D flip flop by showing waveforms for the input and output, and for the clock. At the first rising edge of the clock `C`, the current value of `D` (1, in the figure) is sampled and held, *i.e.,* appears at `Q` after a small propagation delay. At the second rising edge of `C`, the current value of `D` (0, in the figure) is sampled and held, *i.e.,* appears at `Q` after a small propagation delay.

What happens if `D` is changing at or around the rising edge of the clock, as suggested in the third rising edge of `C` in the figure? In this case we are violating the digital abstraction,

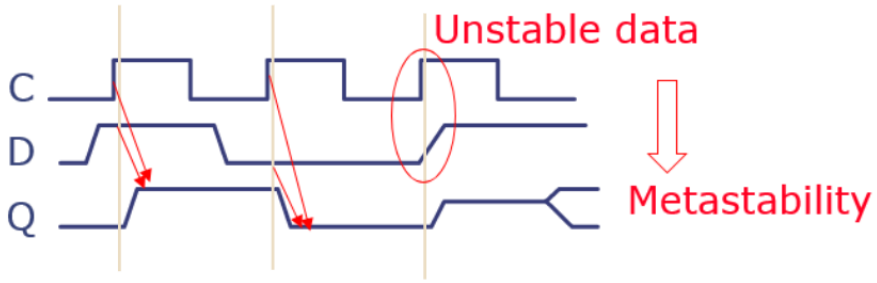


Figure 7.7: Behavior of an Edge-Triggered D-Flip Flop

where signals are supposed to be classifiable definitively as either a "0" or a "1". We are back in to analog circuit analysis territory, and in fact the flip-flop can go into a so-called *meta-stable* state, wandering at a voltage somewhere between our definitions of 0 and 1, for an indefinite period before it settles into a definitive 0 or 1. Every digital designer works to eliminate this indeterminate situation—we must ensure that the output of one state element (*e.g.,* a D flip flop) can propagate through all combinational circuits en route to an input of another state element well before the latter's next rising clock edge, so that it is stable when it is sampled there.

7.2.3 Controlling the Loading of a D Flip Flop with an Enable Signal

An edge-triggerd D flip flop samples its input on every rising edge of its clock. Suppose we want to sample a signal only on some condition, for example when a button is being pressed. One dangerous way to do this is to interpose an AND gate before the `C` input, fed by the clock and the button-press condition. This is extremely dangerous because we have disturbed the time at which the clock edge arrives at the flip flop, delaying it by the propagation delay of the AND gate. Further, if the button-press occurs coincident with a clock transition, the output of the AND gate can be unpredictable and can even "glitch" back and forth a couple of times. All this will lead to unpredictable and undesirable behavior of the flip flop.

A better way to do it, without messing with the clock, is shown in Fig7.8. When the

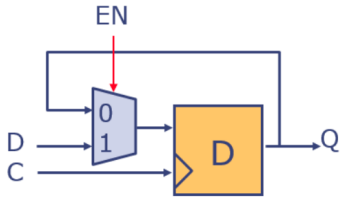


Figure 7.8: Implementation of an Edge-Triggered D-Flip Flop with an ENABLE signal

"enable" signal `EN` is 1, it behaves just like our original edge-triggered D flip flop. When `EN` is 0, it holds its value indefinitely, ignoring any subsequent changes in `D`. Fig 7.9 augments the symbol for the edge-triggered D flip flop to include an `EN` input signal, shown in red. Fig7.10 illustrates the behavior of the state element. Although `D` is 1 at both the first and second rising clock edge, it is sampled only at the second clock edge when `EN` is 1. Similarly, although `D` is 0 at the third clock edge, it is not sampled since `EN` is 0. Thus, the enable

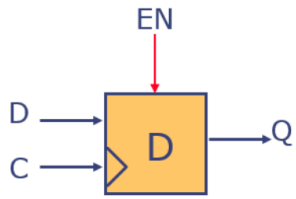
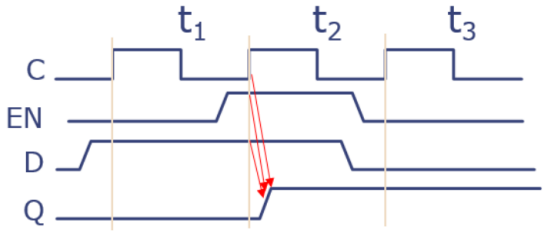


Figure 7.9: Symbol for an Edge-Triggered D-Flip Flop with ENABLE



EN	D	Q^t	Q^{t+1}	
0	X	0	0	hold
0	X	1	1	
1	0	X	0	copy input
1	1	X	1	

Figure 7.10: Behavior of an Edge-Triggered D-Flip Flop with an ENABLE signal

signal gives us precise control over exactly *when* we want to sample and input signal and remember it until the next sampling.

From now on, the edge-triggered D flip-flop with enable will be our basic state element. We often use the word "register", or phrase "1-bit register" synonymously. Some D flip-flops also have a "Reset" signal which re-initializes the flip-flop to a fixed value, say 0. Reset signals may be synchronous (they take effect on the next rising clock edge) or asynchronous (they take effect immediately, irrespective of clock edges).

7.2.4 A Register: a bank of D Flip-Flops

In the circuits we design, our state elements will typically hold n-bit values, for various values of n. This is achieved by simplifying banking several individual D flip flops together with common clock and enable signals, as depicted in Fig 7.11.

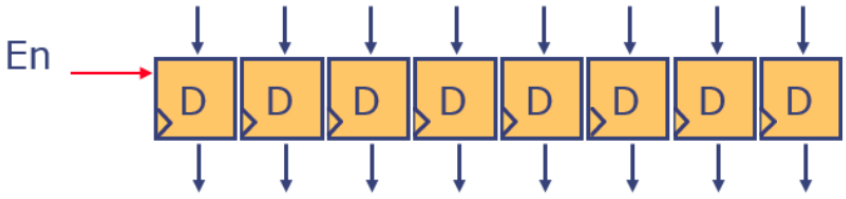


Figure 7.11: An n-bit Register: just a bank of D flip flops

7.2.5 Clocked Sequential Circuits in this book

In this book we restrict ourselves to pure clocked sequential circuits with a single global clock, that is, all state elements will be connected to the same clock, and they all sample their inputs on the rising edge of the clock. Because of this simplification, we will not even draw clock signals in our diagrams, in order to reduce clutter (we will just show the little

triangles on each state element to remind us that it is clocked). Further, we do not even mention clocks in BSV code.

Just for perspective: more advanced circuits can have multiple clocks; they are usually partitioned into so-called *clock domains*, where every state element in each domain has a common clock. Signals that cross clock domain boundaries have to be treated *very* carefully to avoid undesirable behaviors (metastability, unintended loss, duplication or random generation of data, *etc.*). State elements may be latched or clocked, and may also vary in whether they sample on rising or falling edges, *etc.*. BSV has notations and constructs to express such designs, but we do not cover those topics in this book.

7.3 Example: a Modulo-4 Counter and Finite State Machines (FSMs)

A Modulo-4 counter is a 2-bit counter that "wraps round" from the maximum value (3) back to the minimum value (0): 0, 1, 2, 3, 0, 1, 2, 3, ... Let us also assume an input `inc` by which we control whether the counter is actually counting or just holding steady at its most recent count. The behavior is specified in the following *state transition table*, where q1 and q0 are the two bits of the counter:

Current State	Next State	
q1 q0	inc = 0	inc = 1
00	00	01
01	01	10
10	10	11
11	11	00

A Modulo-4 counter is an example of a *Finite State Machine*, or "FSM" for short. It sequences through four states, 00, 01, 10 and 11. A second and equivalent way to specify an FSM is with a more pictorial notation called "bubble diagrams" or "state transition diagrams". The diagram specifying our Modulo-4 counter is shown in Fig 7.12. Each

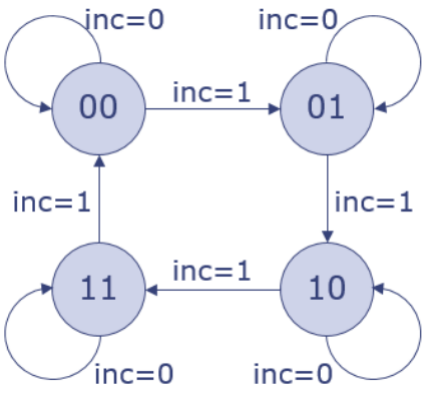


Figure 7.12: An FSM specifying the Modulo-4 counter

bubble represents one state, and each arrow depicts a transition between states. Each

arrow is labeled with the input(s) that trigger that transition. The loopback arrows show where the FSM remains in its current state.

A third and also equivalent way to specify and FSM is show boolean equations for the "next state" based on the current state and current inputs. Here are the equations for our Modulo-4 counter (the t and $t+1$ superscripts indicate values in the current state and next state, respectively).

$$\begin{aligned} q0^{t+1} &= \sim inc \cdot q0^t + inc \cdot \sim q0^t \\ &= inc \oplus q0^t \\ q1^{t+1} &= \sim inc \cdot q1^t + inc \cdot \sim q1^t \cdot q0^t + inc \cdot q1^t \cdot \sim q0^t \\ &= \sim inc \cdot q1^t + inc \cdot (q1^t \oplus q0^t) \end{aligned}$$

7.3.1 Implementing a Modulo-4 Counter using D flip-flops with enables

Fig 7.13

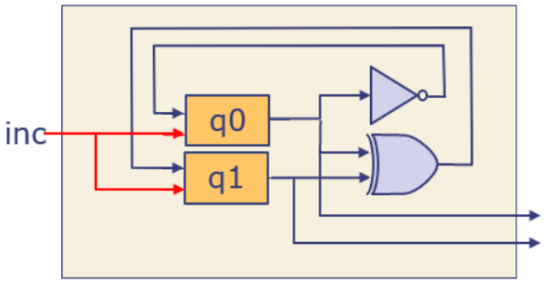


Figure 7.13: A circuit for the Modulo-4 counter

shows an implementation of a Modulo-4 counter. We use two D flip-flops with enables to hold the q0 and q1 bits. The inc input is connected to the flip-flop enables, and thus they change only when inc= 1. The remaining combinational logic is a direct transliteration of the boolean equations shown earlier:

$$\begin{aligned} q0^{t+1} &= inc \oplus q0^t \\ q1^{t+1} &= \sim inc \cdot q1^t + inc \cdot (q1^t \oplus q0^t) \end{aligned}$$

which we can further simplify to the situation where "inc" is 1:

$$\begin{aligned} q0^{t+1} &= q0^t \\ q1^{t+1} &= (q1^t \oplus q0^t) \end{aligned}$$

7.3.2 Abstracted view of all synchronous sequential circuits using registers (FSMs)

If we just redraw Fig 7.13 a bit and step back, its general structure follows the pattern of Fig 7.14. We have a bank of state elements (registers) holding current state, and they are all clocked by a common clock. We have some combinational logic whose inputs come

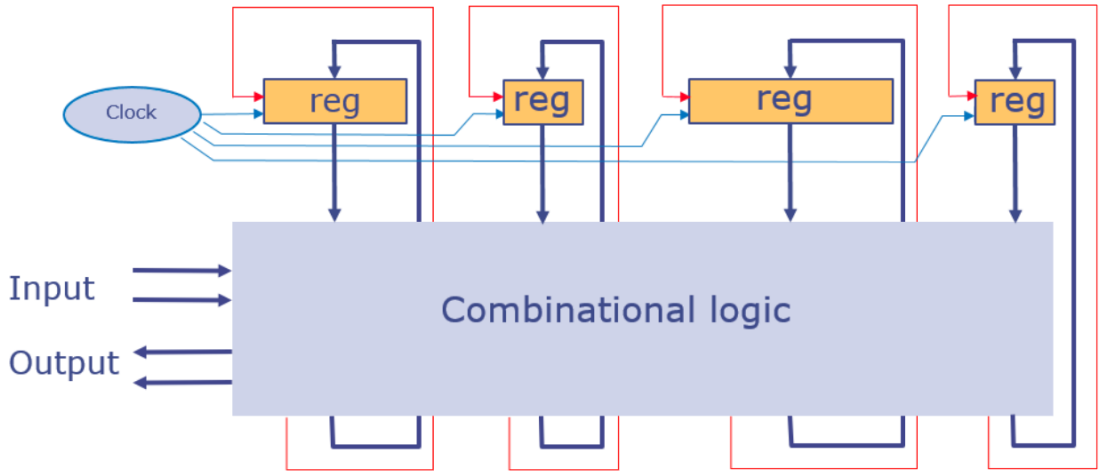


Figure 7.14: Abstract structure of sequential circuits

from the outputs of the state elements and from external inputs. The outputs of the combinational logic are connected to the inputs of the state elements and also provide the external outputs. The combinational circuits implement the next-state and output functions of the FSM. Fig 7.15 shows an even more abstract view of the same thing.

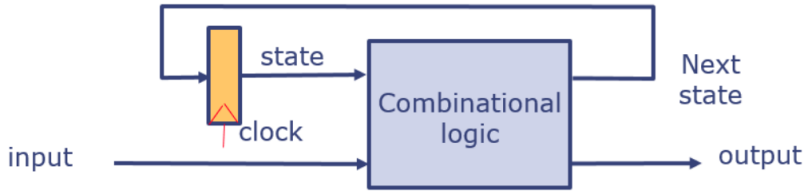


Figure 7.15: Even more abstract structure of sequential circuits

Synchronous sequential circuits are a way to implement FSMs in hardware. FSMs are a much studied topic in mathematics, just like Boolean Algebra. FSMs are used in software as well, particularly in network communication protocols such as socket connection, TCP/IP, UDP/IP and so on. In theory, all digital artifacts are FSMs, even your laptop and your mobile phone, since the number of bits of state (registers, memory, *etc.*) is large (in the gigabit or terabit range nowadays), but finite, although this is not a very tractable or practical characterization!

All our FSM notations—the state transition table, bubble diagrams and boolean next-state functions—are only practical for small FSMs. For a sequential circuit with n bits, the size of the specification (number of rows in the state transition table, number of bubbles in the bubble diagram, or right-hand sides of boolean equations) grows exponentially (2^n) and very quickly becomes unmanageable. Imagine trying to draw an FSM that characterizes your laptop that has, say, four gigabytes of memory (state elements)! There are many ways to abbreviate state transition tables and bubble diagrams, but they go only so far.

Instead, for large systems, the tried-and-true principle of "divide and conquer" is the best way to specify and comprehend them. Our systems will be specified/described/designed hierarchically (recursively). At every layer, we have a *simple compositions of simpler FSMs.*

BSV is used both to describe FSMs and their composition into more complex FSMs. Thus, real systems are more practically characterized as communicating FSMs, rather than as a single FSM.

7.4 Other commonly used stateful primitives

7.4.1 A Register File: a bank of individually selectable registers

A register file (such as the set of integer registers in a CPU), illustrated in Fig 7.16, is an

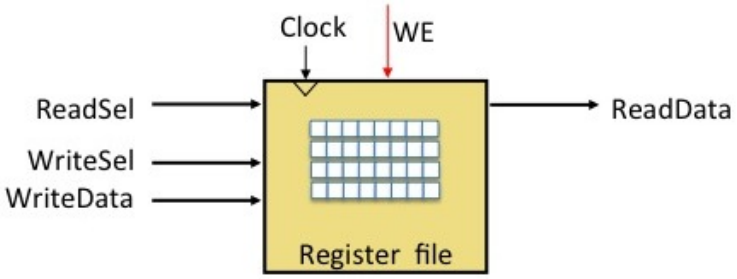


Figure 7.16: A Register File with four 8-bit registers

array of registers, with a way to selectively read and write an individual register identified by an index. The ReadSel signal is an index that identifies one of the n registers in the register file. The data in that register is presented on the ReadData output. This is usually a combinational function, just like reading a single register. The WriteSel signal identifies one of the registers, and if the WE enable signal is true, then WriteData is stored into that selected register.

Fig. 7.17 illustrates how a register file can be implemented. The read and write circuits

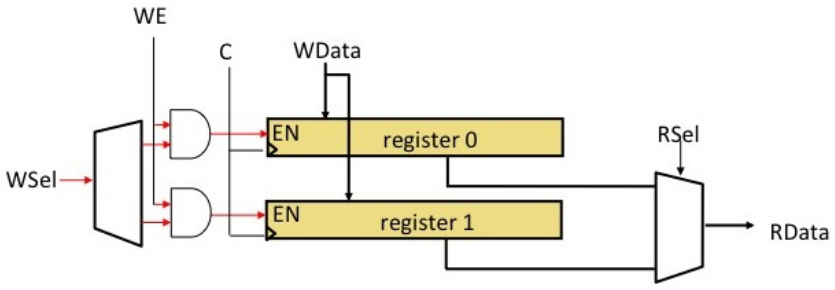


Figure 7.17: Register File implementation

are entirely separate. The read circuit is just a mux choosing the Q outputs of one of the registers. In the write circuit, Write Data is fed to the D inputs of all the registers. The Write Enable signal is fed to only one selected register, which captures the Write Data.

The read and write interfaces of the register file are called "ports". Many register files have more than one read and/or write port. Fig. 7.18 illustrates a register file with two read ports and one write port. It is easy to see how to extend the implementation of Fig. 7.17 for this—simply replicate the mux that implements the read circuit. When implementing

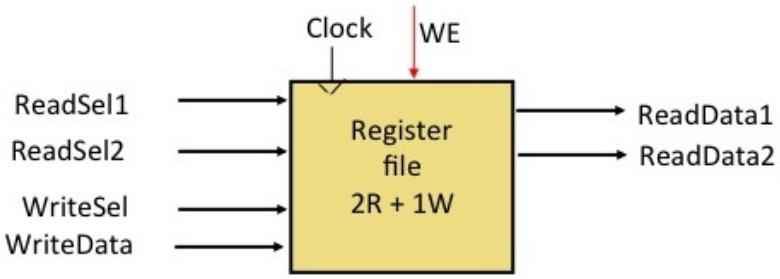


Figure 7.18: Multi-port Register File

multiple write ports, one has to define what it means if both Write Selects are the same, i.e., both ports want to write to the same register, possibly with different values. We may declare this illegal, or a no-op, or fix a priority so that one of them is ignored. All these schemes are easy to implement.

Why are multi-port register files interesting? In a CPU, the architectural register set may be implemented directly as a register file. Many instructions (such as ADD) read two registers, perform an op, and write the result to a register. If we wish to execute one such instruction on every clock cycle, then clearly we need a 2-read port, 1-write port register file. If we wish to execute two such instructions in every clock (as happens is modern "superscalar" designs), we'll need a 4-read port, 2-write port register file.

7.4.2 Memories

A register file is essentially a small "memory". The Read Select or Write Select input is an address identifying one location in the memory, and that location is read and/or written. However, building large memories out of register files is expensive in silicon area and power consumption. It is rare to see register files with more than a few dozen locations.

Typically, larger memories are built with special primitives that have been carefully engineered to be very dense (low area) and requiring much less power. The next step up in size from register files are on-chip SRAMs (Static RAMs), typically holding kilobytes to a few megabytes. They typically do not have more than one or two ports.

For even larger memories, we typically have to go to off-chip SRAMs (multi-megabytes) and DRAMs (Dynamic RAMs) in the megabyte to gigabyte range. On chip, you will typically have a "memory interface" circuit that communicates with the off-chip memories.

Unlike register files, SRAMs and DRAMs usually do not have combinational reads, i.e., data is only available 1 or more clock cycles after the address has been presented. And they typically do not have multiple ports for reading or writing.

Most modern computer systems have a *memory hierarchy*, with a few fast and expensive state elements at one end, and many slow and cheap elements at the other end. By "fast" and "slow" we are referring to clock speeds at which they run and latency of access. By "expensive" and "cheap" we are referring to silicon area and power consumption. D flip-flops are fast and expensive; SRAMS are slower and cheaper; and DRAMs are slowest and cheapest. And if we extend our notion of "state" to include FLASH memories and magnetic disk drives, we get elements that are even slower and cheaper, and usually even bigger.

Exercise 7.2: Design a circuit for a counter that will increment, decrement or preserve its value, depending on an input control signal. Does your circuit need a "busy" signal? □

Chapter 8

Sequential Circuits in BSV: Modules with Guarded Interfaces

8.1 Introduction

In the last chapter we introduced sequential circuits, base on the edge-triggered D flip-flop with an enable signal. Circuits on real chips contain thousands to millions of D flip-flops, so we need a way to impose a hierarchical structure on circuits so that we can comprehend and manage such complexity. In BSV, the fundamental construct for this is the *module* with its well-defined *interface*. All our circuits have a recursive modular structure—complex designs are written as a top-level module that is a composition of smaller modules that communicate with each other using their interfaces. These modules, in turn, may be composed of smaller modules, also communicating via interfaces. At the bottom of the hierarchy we have primitive modules that are pre-defined in the BSV language or supplied in BSV libraries.

To manage complexity, we aim for each module to have a specific, manageable function that we can easily articulate and comprehend. A module's interface consists of *methods* (similar to what we see in object-oriented languages such as C++, Java or Python) that are invoked from other modules. Most module interfaces also require a *protocol* about how methods are invoked, for correct operation. In BSV, these protocols are enforced by the language's type system and concurrent semantics, and so we also call them *guarded interfaces*.

8.2 Registers in BSV

The n-bit register depicted in Fig. 7.11 is created in BSV code using statements like this:

Declaring and Instantiating Registers

```
1  Reg #(Bit #(32))  x <- mkRegU;
2  Reg #(Bit #(32))  y <- mkReg (0);
```

In each line, the phrase `Reg #(Bit #(32))` specifies that we are instantiating a register that will hold 32-bit values, *i.e.,* that it will be a collection of thirty-two D flip-flops that

we treat as a single unit (they all have the same clock and same enable). The variables `x` and `y` are the local names by which we will refer to these registers in subsequent code. On the right-hand side of the instantiation symbol `<-`, in line 1 we use the register constructor `mkRegU` to create a register where we do not care about the initial contents (at "reset", or "the start of time"). In line 2 we use the register constructor `mkReg ()` with argument 0 to create a register that will contain the initial value 0. We also say that `mkReg()` instantiates a register with a particular *reset value*. The "U" in `mkRegU` name is mnemonic for "unspecified reset value".

We read registers in combinational expressions, and we assign new values into registers using register-assignment statements. Examples:

Register-assignment statements

```
1       y <= 42;
2       x <= x - y;
```

In each line, the left-hand side refers to a register, and the right-hand side is an arbitrary expression whose value specifies the new value to be assigned into the register. In the first line, we assign the constant 42 into register `y`. In the second line, we read registers `x` and `y`, perform the subtraction and assign the difference back into register `x`. The values of registers in any right-hand side expression refer to the *old*, or pre-assignment contents of registers.[1]

8.3 Example 1: The Modulo-4 counter in BSV

In Sec 7.3 we discussed a Modulo-4 counter, starting with an FSM (Finite State Machine) specification in Fig. 7.12, and then a circuit implementation in Fig 7.13. The inputs and outputs of the circuit were just left dangling. We will now encapsulate the circuit inside a module, and bring the inputs and outputs to the surface in well-defined interfaces.

In system design, whether in software or hardware, whenever we design a module, it is generally a good idea first to specify its interface, *i.e.,* the mechanisms and the semantics by which users or clients of the module are expected to use the module. This is, after all, the "contract" between the users of a module and the implementer of a module. Once this contract is clear, we can go about implementing the inner workings of the module. An interface specification describes *what* a module does, while the internals of a module describe *how* it does it.

Fig 8.1 represents the external view of a module that encapsulates our sequential circuit for a Modulo-4 counter. Every module has an internal *state* and an *interface*. The interface consists of *methods*. The internal state of the module can only be accessed or updated using its interface methods. Interfaces in BSV are declared using notation like the following for our Modulo-4 counter:

```
1  interface Counter;
2     method Action  inc;
```

[1] More accurately, the *pre-Action* contents, which becomes important when an Action can contain multiple statements, but more about that later.

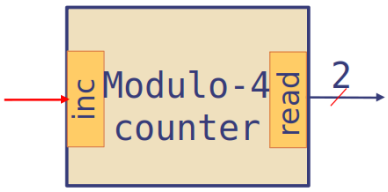


Figure 8.1: A Modulo-4 counter module

```
3     method Bit #(2) read;
4  endinterface
```

It contains two methods, an *action* method called `inc` and a *value* method called `read`. Action methods are used to update the internal state of the module and, when converted to hardware, will always have an *enable* wire by which the external environment tells the module when to perform the update (in this case, to increment the counter). Value methods are used for pure reads (side-effect free, *i.e.,* without updating the internal state) of internal state (in this case, to read the value of the counter).

An interface declaration just specifies the *external view* of a module—it says nothing at all about how the functionality is implemented inside the module. Indeed, one can have many modules that offer the same interface: one implementation may choose to optimize circuit area, another implementation may choose to optimize speed, and another may choose to optimize power consumption.

An interface in BSV is in fact a certain kind of *type* (just like scalar types, structs, or enums). Thus, a function can have interface arguments and return interface results; but more about that later.

Here is one possible implementation of our Modulo-4 Counter module:

```
1  module mkCounter (Counter);
2     Reg #(Bit #(2)) cnt <- mkReg (0);
3
4     method Action inc;
5        cnt <= { cnt [1] ^ cnt [0],  ~ cnt [0] };
6     endmethod
7
8     method Bit #(2) read;
9        return cnt;
10    endmethod
11 endmodule
```

In line 1, we declare the name of the module, `mkCounter`. Our style is to use the prefix "`mk`" in module names, pronounced "make", to emphasize that this is a module constructor—it can be instantiated many times to produce multiple module instances.

Following the module name, in parentheses, is the module's interface type, which we declared earlier. In line 2, the "`<-`", invokes the `mkReg` constructor to instantiate a register,

which we refer to using `cnt`. The `inc` method has one statement, a register assignment to register `cnt` with a value described by the combinational expression on the right-hand side. The `read` method merely returns the value of the register.

Fig 8.2 shows the circuit generated by the BSV compiler for the `mkCounter` module, which is just an encapsulation of the circuit of Fig. 7.13. Every time we assert the input wire of

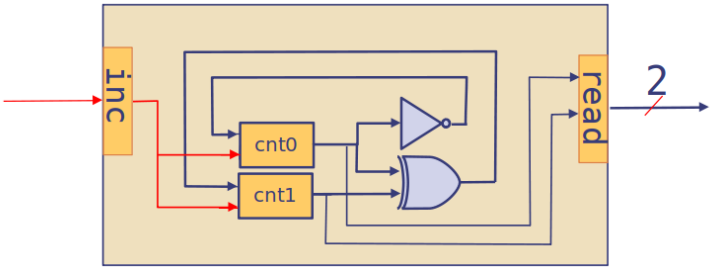


Figure 8.2: The circuit for the Modulo-4 counter module

the `inc` method, it enables the two registers `cnt0` and `cnt1` to update their contents based on the combinational circuit connecting their outputs to their inputs. The `read` method becomes a 2-bit wide output bus on which we read the register values.

8.3.1 Atomicity of methods

When invoked, an action method may update many state elements (such as registers) inside the module and, recursively, inside nested modules. A very important feature of BSV methods is that this collection of updates is done *atomically*. This means two things:

- Either all the specified state elements are modified, nor none of them are modified. It is never the case that some of them are modified and some remain unmodified.
- If two methods of a module are invoked concurrently, it is as if the methods were invoked in some particular order (we will discuss method ordering in more detail later), *i.e.,* neither method observes any "intermediate state" during the updates by the other method.

8.4 Example 2: A module implementing GCD using Euclid's algorithm

Euclid devised an algorithm to compute the GCD (Greatest Common Divisor) of two numbers a and b:

> Repeatedly subtract b from a as long a b is less than or equal to a, or swap them if b is greater than a, until a is zero; then, b contains the GCD.

Here is an example, computing the GCD of 15 and 6:

a	b	operation
15	6	subtract
9	6	subtract
3	6	subtract
6	3	swap
3	3	subtract
0	3	halt

We might write this in Python as:

```
    def gcd (a, b):
        if  a == 0:    return b
        elif  a >= b:  return gcd (a-b, b)
        else:          return gcd (b, a)
```

The interface contains four methods, defined using this type: Fig 8.3 shows a GCD module, its interface methods, and some interface wires in the generated hardware.

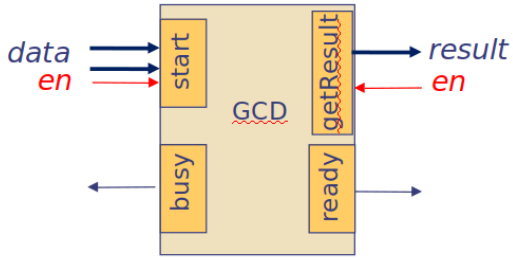


Figure 8.3: A GCD module

```
interface GCD;
   method Action                      start (Bit #(32) a, Bit #(32) b);
   method ActionValue #(Bit #(32))    getResult;
   method Bool                        busy;
   method Bool                        ready;
endinterface
```

The environment of this module must follow a certain protocol. In particular, the `start` method, to start a GCD computation, must not be invoked while the `busy` method returns `True`, and the `result` method must not be invoked while the `ready` method returns `False` (later, we will see how we can build these protocols into the methods themselves, so that they cannot be invoked incorrectly).

Here is the code to implement a GCD module with that interface.

```
module mkGCD (GCD);
   Reg #(Bit #(32)) x         <- mkReg (0);
```

```
3     Reg #(Bit #(32)) y         <- mkReg (0);
4     Reg #(Bool)      busy_flag <- mkReg (False);
5
6     rule gcd;
7       if (x >= y) begin
8          //subtract
9          x <= x - y;
10      end
11      else if (x != 0) begin
12         //swap
13         x <= y;
14         y <= x;
15      end
16    endrule
17
18    method Action start (Bit #(32) a, Bit #(32) b);
19       x         <= a;
20       y         <= b;
21       busy_flag <= True;
22    endmethod
23
24    method ActionValue #(Bit #(32))  getResult;
25       busy_flag <= False;
26       return y;
27    endmethod
28
29    method Bool busy  = busy_flag;
30
31    method Bool ready = (x == 0);
32 endmodule
```

In line 1, similar to our previous example, we give our module a name, `mkGCD`, and declare that its interface will have type `GCD`. In lines 2-4, we instantiate two 32-bit registers `x` and `y`, and a boolean register `busy_flag` that will be used to indicate that the module is busy performing a GCD computation.

Lines 18-22 define what the `start` method actually does. It stores its arguments `a` and `b` in registers `x` and `y` respectively, and sets the register `busy_flag` to `True`, thereby indicating that the module is now "busy". This method should only be invoked when the module is "not busy", which the environment can test by invoking the `busy` method.

Lines 6-16 define a *rule* called `gcd`. A rule is like an infinite process, executing its body repeatedly, forever. The body of the rule specifies the "subtract", "swap" and "halt" actions described earlier, each controlled by the appropriate condition. Note that when the GCD has been computed, `x`$= 0$ and `y`≥ 0, so neither of the if-then-else conditions will be true, and the rule has no action.

Lines 28-30 define the `getResult` method. This should only be invoked when the answer is ready, which the environment can test using the `ready` method. When `getResult` is

invoked, it sets `busy_flag` to false, thus readying the module for the next invocation of `start`, and returns the value of `y`.

The methods `busy` and `ready` in lines 29-31 have been written using short forms that are equivalent to:

```
1   method Bool busy;
2      return busy_flag;
3   endmethod
4
5   method Bool ready;
6      return (x == 0);
7   endmethod
```

In the module code, we are doing subtraction using the operator "`-`" which we usually think of as an operation on integers, whereas our operands here are of type `Bit #(32)`. In general we must be careful of such usage, since "`-`" can mean quite different things for different types (think of signed integers, polar coordinates, and other types). In this context it is ok because `Bit #(32)` can be viewed as similar to (more precisely, *isomorphic* to) unsigned 32-bit integers.

8.4.1 Methods and method types

The examples above illustrate the three major types of methods:

- *ActionValue* methods, like `getResult` `mkGCD`.
- *Action* methods, like `inc` in `mkCounter`, and `start` in `mkGCD`.
- *Value* methods, like `read` in `mkCounter`, and `busy` and `ready` in `mkGCD`.

ActionValue methods are the most general. They can have arguments and results and, most importantly, they can update the state of the module, which we also call "side-effects".

Action methods are a special case of ActionValue methods that do not return any result. They can have arguments and they can update the state of the module, *i.e.,* they can have "side-effects".

Value methods are pure functions, *i.e.,* they cannot have side effects. They can have arguments and results. They can (non-destructively) read internal state, but they cannot update any state. A value method, if invoked multiple times with the same arguments at the same time (for example within a rule or another method), will return the same result. Of course, if invoked multiple times at different times with the same arguments, they could return different results if they read underlying state that has changed in the meantime.

The hardware for Action and ActionValue methods always has an *enable* wire, which is how the external environment signals to the module *when* the method's internal action is to be performed.

Commentary on Action, ActionValue and Value types

[This commentary may be of interest to readers who are familiar with type systems in other programming languages, and can be safely skipped.]

In most languages with strong static type-checking, such as C++, Java and Go, the type of a function or method typically does not encode the property of whether or not it contains side-effects. An *int* → *int* function, from integers to integers, may or may not update state inside the function body.

In a few languages, notably Haskell, this property *is* captured in the type of a function or method, and strong type-checking guarantees, for example, that a side-effectful function cannot be used where a pure function is expected.

BSV follows Haskell's regime. Capturing side-effectfulness in types allows proper, statically checked composition of programs, guarantees the correctness of certain optimizations, and helps in formal reasoning about correctness of programs. For example, later we will see that execution of rules and methods can be controlled with conditions of type `Bool`; this type guarantees that a scheduling decision, about whether to execute a rule/method or not, does not itself change the state of the system.

Invoking methods: arguments and results

Arguments are provided to methods with the traditional notation, with parentheses and separated by commas. More interesting is how we collect results of method invocations or, to use more technical terminology, how we *bind* result values to variables.

Action methods (which have no results) are just written as statements:

module_instance`.`*method_name* `(` *arg_expression*`, ...,` *arg_expression*`)`

Examples:

```
mod4counter.inc;
gcd.start (10+3, 27);
```

For value methods, we invoke them as usual, by applying them to arguments. We can bind the result to a variable, or nest it in other expressions in the usual way. When we bind the result to a name, we can either declare the the type of the name, or use the `let` keyword to ask the compiler to work out the type. Examples:

```
Bool isGcdBusy = gcd.busy;
let r = gcd.ready
if (r && (mod4_counter.read == 0)) begin
   ...
end
```

ActionValue methods are invoked using a special notation involving "`<-`". Once again, the new variable's type can be declared explicitly, or one can use `let` to ask the compiler to work it out. Examples:

```
Bit #(32) result1 <- gcd1.getResult;
let       result2 <- gcd2.getResult;
```

There is nothing technically wrong with using "=" as a binder with an ActionValue method. In this case, you are just binding the variable to the ActionValue method itself, rather than invoking it and binding it to the result of the invocation. Example:

```
let m       = gcd2.getResult;    // m has type ActionValue #(Bit #(32)
let result2 <- m;                // result has type Bit #(32)
```

In the first line, we are binding `m` to the ActionValue method itself, and in the second line we're invoking it, and binding `result2` to the result of the invocation. The difference can be observed in the types of `m` and `result2`.

8.4.2 Rules and atomicity

Rules are where the action is!

The GCD example is our first example involving a *rule*. We repeat the code here, for convenience:

```
1  rule gcd;
2    if (x >= y) begin
3       //subtract
4       x <= x - y;
5    end
6    else if (x != 0) begin
7       //swap
8       x <= y;
9       y <= x;
10   end
11 endrule
```

As described earlier, a rule is a kind of infinite process, repeating forever. The body of rule may contain zero or more actions. In our example, all actions are just register-assignments. Later we will see that actions may be invocations of Action or ActionValue methods (in fact, a register assignment is just a special Action method invocation).

Each rule has a *name*, like `gcd` here, which are mostly for readability and do not play any semantic role.[2]

All the enabled Actions/ActionValues in a rule are performed in parallel, conceptually all at exactly the same instant. By "enabled" we mean all actions whose governing conditions are true, as in the conditions of the if-then-else structure in this example.

A rule's Actions are done *atomically*, just as we described in Section 8.3.1 regarding atomicity of methods. We repeat the definition here, using the more technical term `Action` instead of the more informal phrase "modify state elements":

[2]Rule names are also useful for readability and debugging, and for advising the compiler on how to schedule rules; more about that later.

- Either all the enabled Actions are performed, nor none of them are performed.
- If two separate rules execute concurrently, it is as if they are performed in some particular order, *i.e.,* the Actions of one before the Actions of the other, or vice versa (we will discuss method ordering in more detail later). Neither rule observes any "intermediate state" during the updates by the other rule.

When we say all the enabled Actions of a rule are done *simultaneously*, "semantically at the same instant", what does the following code fragment mean?

```
1     else ... begin
2        //swap
3        x <= y;
4        y <= x;
5     end
```

Here, there are two Actions, one that reads register `x` and writes it to register `y`, and the other that performs the opposite data movement. If we think of a rule execution as a "step", then all observed state values are the values preceding the step, and all updates specify what the value will be after the step. If we refer to these "before" and "after" times as t and $t+1$, we can annotate the code fragment to clarify which values they mean:

$\mathtt{x}^{t+1}$ <= $\mathtt{y}^{t}$;
$\mathtt{y}^{t+1}$ <= $\mathtt{x}^{t}$;

Thus, this is a parallel-read of `y` and `x`, respectively, and a parallel-write of the two values into `x` and `y`, respectively. Thus also, it would make no difference if we had written the statements in the opposite order:

$\mathtt{y}^{t+1}$ <= $\mathtt{x}^{t}$;
$\mathtt{x}^{t+1}$ <= $\mathtt{y}^{t}$;

Since all enabled actions in a rule are performed in parallel at the same instant, it makes no sense to compose actions in a rule that specify contradictory (inconsistent, conflicting) actions. For example:

```
1   rule one;
2      y <= 3;
3      x <= 5;
4      x <= 7;
5   endrule
```

The latter two assignments are contradictory, and it makes no sense for them to happen simultaneously. This "bad parallel composition of actions" is something that the BSV compiler will flag as an error during compilation.

Here is one subtle variation of the above example:

```
rule two;
   y <= 3;
   if (b)
      x <= 5;
   else
      x <= 7;
endrule
```

Even though there are two syntactic occurrences in the same rule of assignments to register `x`, they are semantically mutually exclusive, and so this is fine; and the BSV compiler will allow it. In fact, this example is equivalently written as:

```
rule two_variant;
   y <= 3;
   x <= (b ? 5 : 7);
endrule
```

However, consider the following rule:

```
rule three;
   y <= 3;
   x <= 5;
   if (b)
      x <= 7;
endrule
```

This is meaningless if the condition `b` is true, and this condition may not be statically known, *i.e.,* may depend on some dynamic (circuit execution-time) value. In this case, the BSV compiler will be conservative and disallow the program because of the potential conflict.

8.5 Example 3: A First-In-First-Out queue (FIFO)

In our next example, we will look at a hardware implementation of a *queue*, a structure into which we can enqueue and dequeue items. Because dequeued items emerge in the order in which they were enqueued, in the hardware community we usually refer to this structure as a FIFO, for 'first-in, first-out'[3]. Fig. 8.4 shows a commonly used symbol in diagrams to represent a FIFO.

FIFOs are used extensively in hardware to connect components, to communicate data from one component to another. A circuit that enqueues items into a FIFO is called a *producer*, and a circuit that dequeues items from a FIFO is called a *consumer*. In hardware,

[3]In contrast to a queue, a *stack* has LIFO behavior, for last-in, first-out; this can also be implemented in hardware.

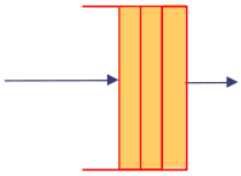


Figure 8.4: Symbol for a FIFO

FIFOs have a fixed maximum capacity, which is often as small as one. If the FIFO is currently full and unable to accept a new item, the producer circuit usually *blocks*, *i.e.*, waits, until a slot is freed up. Similarly, if the FIFO is currently empty, a consumer circuit typically blocks until an item is available. Thus, there is a natural "flow control" between producer and consumer—neither can get further ahead than the other, up to the capacity of the FIFO. A FIFO may also provide a way to test whether it is empty of full.

A FIFO interface can be defined in BSV as follows:

```
1  interface FIFO #(Bit #(n));
2     // enqueue side
3     method Bool       notFull;
4     method Action     enq (Bit #(n) x);
5
6     // dequeue side
7     method Bool      notEmpty;
8     method Action    deq;
9     method Bit #(n)  first;
10 endinterface
```

and the corresponding wires in the generated hardware are depicted in Fig. 8.5. The `x` and

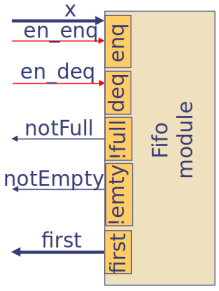


Figure 8.5: A FIFO module

`first` buses are n-bits wide, carrying values of type `Bit #(n)`.

A producer invokes the `notFull` method to test if the FIFO is currently full or not. If not, it can invoke the `enq` method to enqueue an n-bit value into the FIFO. A consumer invokes the `notEmpty` method to test if the FIFO has an item available or not. If available, it can invoke the `first` method to observe the n-bit value at the head of the queue, and it can invoke the `deq` method to perform the action of removing the item from the queue.

Note: it is important that the producer should only invoke `enq` when `notFull` returns true, and the consumer should only invoke `first` and `deq` when `notEmpty` returns true.

The code above defines a new *type*: `FIFO #(Bit #(n))`. This type can be used in many contexts where a type is expected. This FIFO holds items of a specific type: `Bit#(n)`. Further, the interface says nothing about the capacity of the FIFO. Later, we will see how we can parameterize the FIFO interface to specify both its capacity and a more general type for the items it holds (not just `Bit #(n)`).

Here is an implementation of a FIFO with the above interface, that has a capacity of one item:

```
1  module mkFIFO (FIFO #(Bit #(n)));
2     Reg #(Bit #(n)) d <- mkRegU;            // Data
3     Reg #(Bool)     v <- mkReg (False);     // Valid?
4
5     // enqueue side
6     method Bool  notFull = (! v)
7
8     method Action  enq (Bit #(n) x);
9        d <= x;
10       v <= True;
11    endmethod
12
13    // dequeue side
14    method Bool  notEmpty = v;
15
16    method Action  deq;
17       v <= False;
18    endmethod
19
20    method Bit #(n)  first = d;
21 endmodule
```

The internal state of the module consists of two registers that together represent the single "slot" in this one-element FIFO. Register `d` holds an item (an n-bit value) and `v` holds a boolean indicating whether the data is valid or not, *i.e.,* whether the slot is occupied or not, *i.e.,* whether the FIFO contains an item or not. The definitions of the methods, then, are a straighforward interpretation of these registers.

8.5.1 Using FIFOs to "stream" a function

Before we return to improving our FIFO interface and module, we take a moment to show how FIFOs can be used to wrap a combinational circuit so that, rather than just an input value and producing an output value, it can take a stream of input values and produce a stream of corresponding output values.

Fig. 8.6 is a sketch of how this is done. The following code sketches an implementation.

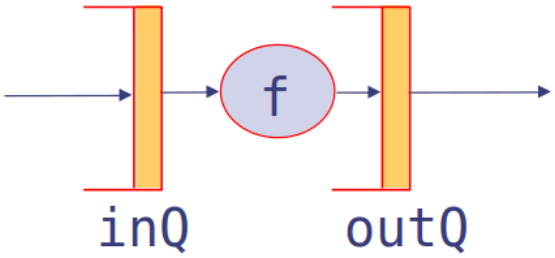


Figure 8.6: Streaming a combinational function using FIFOs

```
module mk_f_stream (FIFO #(Bit #(n)));
   FIFO #(Bit #(n))  inQ  <- mkFIFO;
   FIFO #(Bit #(n))  outQ <- mkFIFO;

   rule stream;
      if (inQ.notEmpty && outQ.notFull) begin
               // Collect a value from inQ
         Bit #(n) x = inQ.first;
         inQ.deq

         // Transform the value
         // f is some combinational function from Bit #(n) to Bit #(n)
         Bit #(n)  y = f (x);

         // Send the result out on outQ
         outQ.enq (y);
      end
   endrule

   // enqueue side
   method Bool       notFull          = inQ.notFull;
   method Action     enq (Bit #(n) x) = inQ.enq (x);

   // dequeue side
   method Bool      notEmpty = outQ.notEmpty;
   method Action    deq      = outQ.deq;
   method Bit #(n)  first    = outQ.first;
endmodule
```

Note that this module, `mk_f_stream` also has the same FIFO interface. When the environment enqueues an item `x` it goes into the input FIFO `inQ`. The rule continually dequeues items from `inQ` as they become available, transforms them with function `f`, and enqueues the results into `outQ`. The environment collects results from the module by collecting it from `outQ`.

This is small example of how FIFO interfaces are highly *compositional*. By "compo-

sitional" we mean that we can take smaller components with a particular interface and combine them into a larger component that has the same interface. In this case, actual FIFOs are our smaller components, and `mk_f_stream` is a larger component that also has a FIFO interface. Because of this, the technique can be applied recursively, and this is a key to being able to build large, complex systems incrementally in manageable chunks; it is a "divide-and-conquer" principle.

8.6 Guarded interfaces

In the previous examples, for correct operation, the environment had to follow a protocol in invoking methods.

- In the GCD example, the environment should not invoke the `start` method while `busy` returns true, and should not invoke the `getResult` method until `ready` returns True.
- In the FIFO example, then environment should not invoke `enq` until `notFull` returns true, and should not invoke `first` or `deq` until `notEmpty` returns true.

Not only is this tedious, but it is also a danger for correctness; what if we carelessly invoked a method when it is not supposed to be invoked?

Exercise 8.1: In the previous codes for GCD and the FIFO, hand-simulate a scenario when the protocol is not followed in invoking methods, and describe what happens. □

In BSV, we can combine many of these correctness conditions into methods so that the protocol is followed automatically, in fact guaranteeing that they will never be invoked erroneously. As a beneficial effect, it also reduces the tedium of writing such code and increases its clarity.

8.6.1 The FIFO again, this time with guarded interfaces

First, our interface simplifies into fewer methods:

```
1 interface FIFO #(Bit #(n));
2    // enqueue side
3    method Action  enq (Bit #(n) x);
4
5    // dequeue side
6    method Action    deq;
7    method Bit #(n)  first;
8 endinterface
```

Fig. 8.7 is a sketch of the corresponding interface wires in the hardware. In addition to the previous wires, each method now has an additional output *ready* wire carrying a boolean value (*i.e.,* 1 bit) that indicates whether that method can currently be invoked or not.

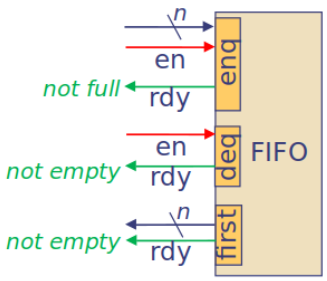


Figure 8.7: The FIFO interface again, with guards

Specifically, the ready-signal of a value method such as `first` can be seen as a "valid" bit indicating whether the output value is valid or not. The ready-signals of Action methods such as `enq` and `deq` can be seen as predicates indicating whether it is legal for the environment to assert the corresponding enable-signals. In our diagrams, we consistently show ready signals in green and enable signals in red.

We also refer to the ready-signals as *guards*, to BSV methods as guarded methods, and BSV interfaces as guarded interfaces.[4]

Note: although we have removed the `notFull` and `notEmpty` methods for this example, it is perfectly reasonable to keep those methods as well, as in the original definition, because some producers and consumers may wish to test the fullness/emptiness of the FIFO anyway, for example to decide whether to perform some other useful work instead of waiting.

Here is the revised code for the mkFIFO module.

```
1  module mkFIFO (FIFO #(Bit #(n)));
2     Reg #(Bit #(n)) d <- mkRegU;             // Data
3     Reg #(Bool)     v <- mkReg (False);      // Valid?
4
5     // enqueueue side
6     method Action  enq (Bit #(n) x) if (! v);
7        d <= x;
8        v <= True;
9     endmethod
10
11    // dequeue side
12    method Action deq () if (v);
13       v <= False;
14    endmethod
15
16    method Bit #(n)  first () if (v);
17       return d;
18    endmethod
19 endmodule
```

[4] Comparing BSV methods to methods in popular object-oriented languages, guards are a fundamental and important addition to method semantics. They are perhaps not that important in software programming languages that are sequential, but they are of fundamental importance in a hardware language, where we are dealing with massive, fine-grain concurrency.

In each method, following the methods' argument list, we have added a guard:
`if (`*condition*`)`
which is a boolean expression representing the condition under which it is legal to invoke the method. In the hardware, this value directly drives the ouput ready-signal.

Some syntax notes: for methods like `deq` and `first` which take no arguments, the empty argument list `()` is optional, but it can make the definition clearer for the human reader.

8.6.2 Streaming a function with guarded interfaces

We can now see how guards simplify our `mk_f_stream` module, which we studied in Sec. 8.5.1:

```
1  module mk_f_stream (FIFO #(Bit #(n));
2     FIFO #(Bit #(n)  inQ  <- mkFIFO;
3     FIFO #(Bit #(n)  outQ <- mkFIFO;
4
5     rule stream;
6        // Collect a value from inQ
7        Bit #(n) x = inQ.first;
8        inQ.deq
9
10       // Transform the value
11       // f is some combinational function from Bit #(n) to Bit #(n)
12       Bit #(n)  y = f (x);
13
14       // Send the result out on outQ
15       outQ.enq (y);
16    endrule
17
18    // enqueueue side
19    method Action  enq (Bit #(n) x) = inQ.enq (x);
20
21    // dequeue side
22    method Action    deq   = outQ.deq;
23    method Bit #(n)  first = outQ.first;
24 endmodule
```

The rule `stream` will not execute (we also say: will not "fire") until the guards of all the methods invoked in the rule are allowed to be invoked, *i.e.,* until all their ready-signals are true.

When we first introduced rules in Sec 8.4, we informally described them as a kind of infinite process. Here we see the next nuance: rules will not execute until the methods they invoke are allowed to be invoked.[5]

[5]There are further nuances when multiple rules compete for shared methods, which we will describe later.

8.6.3 GCD again, this time with guarded interfaces

Let us revisit our GCD example, adding guards to the methods. Here is the revised interface (it simply gets rid of the `busy` and `ready` methods):

```
interface GCD;
   method Action                      start (Bit #(32) a, Bit #(32) b);
   method ActionValue #(Bit #(32))    getResult;
endinterface
```

Fig. 8.8 is a sketch of the GCD interfaces in hardware. The left-hand side repeats Fig. 8.3 (without guards) and the right-hand side is the new interface with guards.

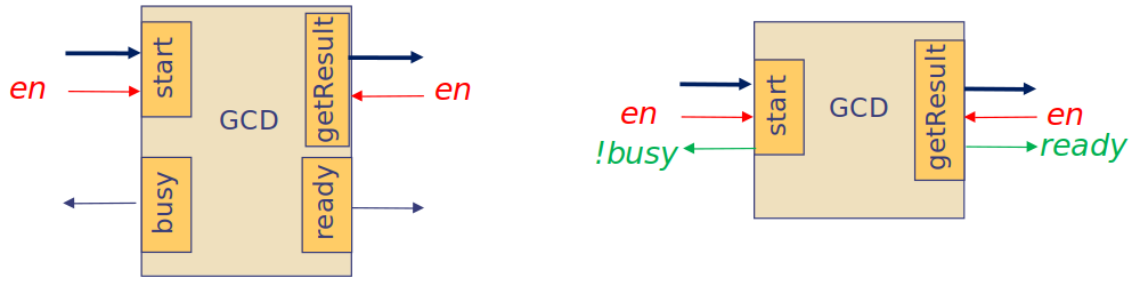


Figure 8.8: The GCD interface with and without guards

Here is the revised module implementation, moving the `busy` and `ready` conditions into the guards of the `start` and `getResult` methods.

```
module mkGCD (GCD);
   Reg #(Bit #(32)) x         <- mkReg (0);
   Reg #(Bit #(32)) y         <- mkReg (0);
   Reg #(Bool)      busy_flag <- mkReg (False);

   rule gcd;
     if (x >= y) begin
        //subtract
        x <= x - y;
     end
     else if (x != 0) begin
        //swap
        x <= y;
        y <= x;
     end
   endrule

   method Action start (Bit #(32) a, Bit #(32) b) if (! busy_flag);
      x         <= a;
```

```
20      y         <= b;
21      busy_flag <= True;
22   endmethod
23
24   method ActionValue #(Bit #(32))  getResult () if (x == 0);
25      busy_flag <= False;
26      return y;
27   endmethod
28 endmodule
```

8.6.4 Rules with guards

Earlier, we discussed how a rule cannot fire if the guard on any method it invokes is false. We call those conditions *implicit conditions* because they are not stated anywhere in the rule; they are in internal detail of how the method is implemented in a particular module implementation.

Rules can also have *explicit conditions*, which are additional boolean expressions that control whether a rule fires or not (it is AND-ed with the implicit conditions). For example, rule `gcd` could have been written as two rules with explicit conditions, as follows:

```
1    rule gcdSubtract (x >= y);
2         x <= x - y;
3    endrule
4
5    rule gcdSwap (x != 0);
6         x <= y;
7         y <= x;
8    endrule
```

Here, the rule name is followed by an explicit rule condition in parentheses. Omitting this, as in the original rule, is equivalent to having a rule condition that is always true. The first rule fires whenever `x` is $\geq$ `y`, and the latter rule fires whenever `x` $\neq 0$. Thus, despite there being two rules, and thus, conceptually, two concurrent infinite processes, it remains a sequential, one-rule-at-a-time algorithm due to the mutual exclusivity of the rules. Later we will see examples of rules whose conditions are *not* mutually exclusive, allowing concurrent execution of rules.

8.6.5 Generalizing the FIFO interface to arbitrary types and sizes

The FIFO interface described in Sec. 8.5 is for FIFOs that carry data of a particular type: `Bit #(n)`. This is somewhat flexible, since n may vary from one instance of a FIFO to another. But what if the items we want to communicate are of some other type, such as `Int #(n)`, or an enumeration or struct type?

A more general type for FIFO interfaces is: `FIFO #(t)` where `t` is any type t. This is in fact the standard definition for FIFO interfaces in the BSV library.

Another important feature of a FIFO is its capacity n, or size: how many items can it contain (our example earlier had $n = 1$). The higher the capacity, the more the "elasticity", "asynchronicity" or "decoupling" between the producer and consumer. A producer can run ahead and produced n items before the consumer is ready to consume even one. Conversely, if n items have been enqueued, the consumer can work on them even if the producer is subsequently unable to make progress. The BSV library provides several alternative FIFO modules where you can supply the desired capacity when you instantiate the module.

8.6.6 Streaming a module

In Sec. 8.5.1 we saw how we could take a combinational function and wrap it with FIFOs into a module that consumes a stream of inputs, applies the function to each input and produces the stream of corresponding outputs. Here we generalize the idea to wrapping an arbitrary module into a stream processor.

We use our GCD module as an example. We want to wrap it into a module that consumes a stream of pairs of numbers, computes the GCD of each pair, and produces a stream of corresponding results. Fig. 8.9 shows the desired wrapping. The interface for this

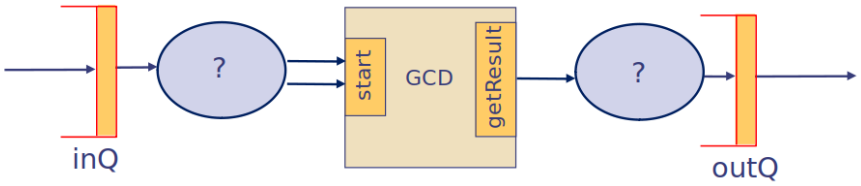


Figure 8.9: Streamed GCD

module is similar to a FIFO interface, but not exactly the same because the types of input and output data are different:

```
interface Streamed_GCD_IFC;
   // enqueueue side
   method Action  enq (Vector #(2, Bit #(32) v2));

   // dequeue side
   method Action    deq;
   method Bit #(32)  first;
endinterface
```

The incoming data is of type `Vector #(2, Bit #(32))`, *i.e.,* it is an array of two 32-bit elements. The outgoing data is of type `Bit #(32)`.

Here is a definition for our module:

```
module mkStreamed_GCD_IFC #(Streamed_GCD_IFC);
   FIFO #(Vector #(2, Bit #(32))) inQ  <- mkFIFO;
   FIFO #(Bit #(32))              outQ <- mkFIFO;
```

```
    GCD                              gcd  <- mkGCD;

    rule invokeGCD;
      let x = inQ.first [0];
      let y = inQ.first [1];
      gcd.start (x, y)
      inQ.deq;
    endrule

    rule getResult;
       let z <- gcd.getResult;
       outQ.enq (z);
    endrule

    // enqeueue side
    method Action  enq (Vector #(2, Bit #(32) v2)) = inQ.enq (v2)

    // dequeue side
    method Action    deq   = outQ.deq;
    method Bit #(32) first = outQ.first;
endmodule
```

We are using the versions of FIFO defined in Sec. 8.6.1 and GCD defined in Sec. 8.6.3, *i.e.*, the versions with guarded methods. Thus, rule `invokeGCD` can only fire when there is data available in `inQ` and when the GCD module is ready to accept an input (pair of numbers). The rule `getResult` can only fire when the GCD module has a result available, and when `outQ` has room to accept another datum.

If we had not used the guarded versions, the rule `invokeGCD` would need an explicit rule condition, like this:

```
    rule invokeGCD (inQ.notEmpty && (! gcd.busy);
       ...

    rule getResult (gcd.ready && outQ.notFull);
       ...
```

The implicit conditions get rid of all this clutter and, more importantly, avoids the situation where we might have carelessly omitted these conditions.

8.7 A method's type determines the corresponding hardware wires

It is possible, just by reading an interface declaration and with no additional context, to draw the interface wires that will be generated in the hardware. The mapping is so simple that BSV designers effortlessly think interchangeably at these two levels of abstractions (method types and hardware wires):

- Each method argument becomes an input bus to the module, just as wide as the number of bits needed to represent the data type of that argument.
- Each method result becomes an output bus from the module, just as wide as the number of bits needed to represent the data type of the result.
- Every method has an output READY wire indicating when it is legal to invoke it (we show these in green).
- Every `Action` and `ActionValue` method has an input ENABLE wire indicating to the module when the method is being invoked by the external environment, *i.e.,* telling the module when to perform the method's action (we show these in red).

8.8 The power of abstraction: composing modules

Separating the notion of an interface from a module that implements that interface is an example of *abstraction*. External environments ("clients") that use that interface need to know nothing more than the properties of that interface in order to use it correctly. Module implementations need to know nothing about clients and how they will use the interface, as long as they correctly implement the properties of the interface. This "substitutability" and "reusability", where a module can be substituted by another one without touching clients, and where a module can be instantiated in different clients, is a central requirement for the ability to build complex, scalable systems.

As an exercise to illustrate this point, let us build a new GCD module that has double the performance of our existing GCD module; further, we'll just reuse our existing GCD module as a building block.

Our existing GCD module becomes "busy" while computing a GCD; the client cannot supply the next input until the module is no longer busy (the current result has been consumed by the client). Our strategy to improve performance is to replicate our existing GCD module, so that two of them can be working at the same time, taking turns to handle successive inputs. Fig. 8.10 illustrates the structure. `gcd1` and `gcd2` are two instances of our

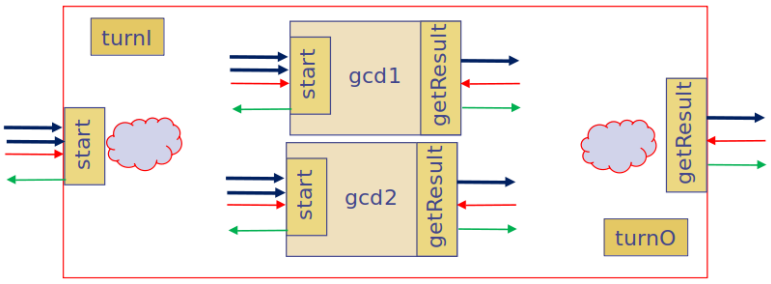


Figure 8.10: Multi-GCD

existing GCD module. The outer `start` method will use a boolean (1-bit) register `turnI` to determine whether to feed the input to `gcd1` or to `gcd2`, and will flip the bit each time. Similarly, the outer `getResult` method will use the boolean `turnO` register to determine whether to collect the result from `gcd1` or from `gcd2`, and will flip the bit each time. Here is the code for the module.

```
module mkMultiGCD (GCD);
   GCD         gcd1  <- mkGCD;
   GCD         gcd2  <- mkGCD;
   Reg #(Bool) turnI <- mkReg (False);
   Reg #(Bool) turnO <- mkReg (False);

   method Action start (Bit #(32) a, Bit #(32) b);
      if (turnI)
         gcd1.start (a,b);
      else
         gcd2.start (a,b);
      turnI <= (! turnI);
   endmethod

   method ActionValue (Bit#(32)) getResult;
      Bit#(32) y;
      if (turnO)
         y <- gcd1.getResult
      else
         y <- gcd2.getResult;
      turnO <= (! turnO);
      return y;
   endmethod
endmodule
```

Note that `mkGCD` and `mkMultiGCD` have exactly the same interface type. Thus, one can be substituted for the other, in any context, without changing functionality; The difference will be in non-functional properties, such as performance (here, throughput or bandwidth) and silicon area. These are exactly the kinds of tradeoffs that are contemplated in all hardware designs.

Exercise 8.2: Generalize `mkMultiGCD` to use four instances of `mkGCD`. □

Exercise 8.3: Generalize `mkMultiGCD` to use n instances of `mkGCD`, where n is a parameter. □

Exercise 8.4: Suppose `gcd1` is given an input whose computation takes a long time. Suppose `gcd2` is then given an input that completes quickly. `mkMultiGCD` is now stuck until the first input's computation completes, even if the next input is already available. This is typical of so-called "static allocation" of tasks to servers (here, we have two servers).

Change `mkMultiGCD` to do dynamic allocation of tasks. In the scenario described in the previous paragraph, the next input can be fed to any server $\texttt{gcd}_j$ that happens to be free.

Caveat: the overall input-output order should be maintained; one result should not be allowed to "overtake" another result just because it was computed more quickly. *Hint*: use a FIFO to remember which $\texttt{gcd}_j$ will produce the "next" result, by enqueueing j in the FIFO. □

Exercise 8.5: Identify a scenario where your solution to the previous exercise may still get stuck. How would you use more FIFOs to alleviate this? □

8.9 Summary

Modules with *rules* and *guarded interfaces* are a powerful way of expressing sequential circuits. Rules and guarded interfaces are a fundamentally different way to think about sequential circuits compared to other hardware design languages (HDLs), including Verilog, VHDL, and SystemVerilog.

A module instance, like an object-class in object-oriented languages, has a well-defined interface consisting of methods, and can be instantiated multiple times.

Unlike object-oriented languages, our methods here are guarded; they can be invoked only if "ready".

The compiler ensures that a method is invoked from the module's environment only when it is ready.

The fundamental "processes" in our computation model are *rules* which can fire repeatedly; there is no concept of a sequential thread that is found in all programming languages and hardware design languages. Rules are *atomic actions*, i.e., one rule can never observe "inconsistent intermediate state" due to concurrent execution of any other rule.

Systems are composed (glued together) structurally by modules instantiated in a hierarchy, and behaviorally by rules in one module (atomically) invoking methods in other modules.

A rule can fire only if its explicit condition, and all the conditions guarding all the methods it invokes, are true.

Chapter 9

Iterative circuits: spatial and temporal unfolding

The Space-Time Continuum

9.1 Introduction

Most interesting algorithms involve iteration (looping, repetition) of one kind or another. One aspect of an iteration is *control*— when should the loop stop or exit? The other aspect of a loop is the work that it performs on each iteration, which we call the *loop body*. Each of these components, the loop-termination condition and the loop body, can be implemented as a circuit, and our task then is to compose them into a circuit that performs the overall loop.

There are typically two interesting directions we can go:

- *Spatial unfolding*: We create a separate instance of the loop-body-circuit for each iteration of the loop, and connect them up so that data flows appropriately from the output of one instance (iteration) into the input of the next one. The values "before" the loop flow into the circuit for the first iteration, and the values "after" the loop flow out of the circuit for the last iteration.
- *Temporal unfolding*: We create just one instance of the loop-body-circuit. For each value that is updated in one iteration of the loop and used in a later iteration, we instantiate a register to hold that value between iterations. The outputs of these registers feed into the loop-body-circuit, and the outputs of the loop-body-circuit are used to update the registers. This is repeated for multiple clocks (in time) for as many iteration as are needed.

  The values "before" the loop are the initial contents of the registers. The values "after" the loop are the final contents of the registers.

Spatial unfolding likely takes more area (we are replicating the loop-body circuit), but likely delivers the result with less latency (just the propagation delay through all the loop-body

circuits). Temporal unfolding likely takes less area (we have one copy of the loop-body circuit), but likely delivers the result with greater latency (as many clocks as it takes to perform all iterations).

These are two extremes of a spectrum of design choices. An intermediate point in the space may be to create, say, 4 copies of the loop-body circuit (spatial) and repeat that for as many clocks as necessary (temporal).

9.1.1 Static vs. dynamic loop control

In some loops, the loop count (how many times it iterates) does not depend on input data. For example, adding all elements of a 100-element array has a fixed loop count of 100 no matter what data is in the array. We say the loop control is *static*.

In other loops, the loop count depends on input data. For example, finding the index of the first zero byte in an array. We say the loop control is *dynamic*.

Loops with static control can be spatially unfolded (if the area cost is acceptable). Loops with dynamic control cannot be fully spatially unfolded (we do not know how many copies of the loop-body are needed); but they can of course use bounded spatial unfolding (e.g., four copies of the loop body which are then temporally iterated).

Note this whole discussion is independent of the question of whether the loops in the algorithm are expressed using "for-loops", "while-loops" or recursion; the key question is whether the loop control is static or dynamic.

9.2 Example: A sequential multiplier

Recall the combinational "multiply" function from Sec. 6.2.6:

BSV code for combinational multiplication (not sequential!)

```
1  function Bit #(64) mul32 (Bit #(32) a, Bit #(32) b);
2     Bit #(32) prod = 0;
3     Bit #(32) tp = 0;
4     for (Integer i=0; i<32; i=i+1) begin
5        Bit #(32) m = (a[i]==0)? 0 : b;
6        Bit #(33) sum = add32 (m, tp, 0);
7        prod[i] = sum[0];
8        tp = truncateLSB(sum);
9     end
10    return {tp,prod};
11 endfunction
```

The for-loop in the code has static control—it always performs 32 iterations, no matter what are the values of the inputs to be multiplied. When this function is processed by the Bluespec compiler, it will perform spatial unfolding. It will unfold the loop into a combinational circuit that contains 32 instances of the `add32` circuit (and surrounding logic).

9.2.1 Example: A sequential multiplier for n-bit multiplication

We see that the values that change during the loop are `i`, `prod` and `tp`; we will need to hold these in registers when writing a sequential version of the loop: The values `m` and `sum`, although produced in the loop body, are immediately consumed in the same iteration; they are not carried from one iteration to the next, and therefore need not be saved in registers.

Here is an alternative, sequential implementation of the multiplier, using just one copy of the loop-body code.

BSV code for sequential multiplication

```
1   Reg #(Bit#(32)) a    <- mkRegU;
2   Reg #(Bit#(32)) b    <- mkRegU;
3   Reg #(Bit#(32)) prod <- mkRegU;
4   Reg #(Bit#(32)) tp   <- mkRegU;
5   Reg #(Bit#(6))  i    <- mkReg (32);
6
7   rule mulStep (i<32);
8      Bit#(32) m   = ((a[i]==0)? 0 : b);
9      Bit#(33) sum = add32(m,tp,0);
10     prod[i] <= sum[0];              // (not quite kosher BSV)
11     tp <= truncateLSB(sum);
12     i <= i + 1;
13  endrule
```

The register `i` is initialized to 32, which is the quiescent state of the circuit (no activity). To start the computation, something has to load `a` and `b` with the values to be multiplied and load `prod`, `tp` and `i` with their initial value of 0 (we will see this initialization later). Then, the rule fires repeatedly as long as `(i<32)`. The first four lines of the body of the rule are identical to the body of the for-loop in the combinational function at the start of this chapter, except that the assignments to `prod[i]` and `tp` have been changed to register assignments.

A small nuance regarding types

Why did we declare `i` to have the type `Bit#(6)` instead of `Bit#(5)`? It's because in the expression `(i<32)`, the literal value 32 needs 6 bits, and the comparision operator `<` expects both its operands to have the same type. If we had declared `i` to have type `Bit#(5)` we would have got a type checking error in the expression `(i<32)`.

What happens if we try changing the condition to `(i<=31)`, since the literal 31 only needs 5 bits?

Types subtlety

```
1  ...
2  Reg#(Bit#(5))  i <- ...
3
4  rule step if (i<=31);
5     ...
6     i <= i+1;
7  endrule
```

This program will type check and run, but it will never terminate! The reason is that when `i` has the value 31 and we increment it to `i+1`, it will simply wrap around to the value 0. Thus, the condition `i<=31` is always true, and so the rule will fire forever. Note, the same issue could occur in a C program as well, but since we usually use 32-bit or 64-bit arithmetic in C programs, we rarely encounter it. We need to be much more sensitive to this in hardware designs, because we usually try to use the minimum number of bits adequate for the job.

Some optimizations of circuit area

Although functionally ok, we can improve it to a more area-efficient circuit by eliminating dynamic indexing. Consider the expression `a[i]`. This is a 32-way mux: 32 1-bit input wires connected to the bits of `a`, with `i` as the mux selector. However, since `i` is a simple incrementing series, we could instead repeatedly shift `a` right by 1, and always look at the least significant bit (LSB), effectively looking at `a[0]`, `a[1]`, `a[2]`, ... Shifting by 1 requires no gates (it's just wires!), and so we eliminate a mux. Similarly, when we assign to `prod[i]`, we need a decoder to route the value into the i^{th} bit of `prod`. Instead, we could repeatedly shift `prod` right by 1, and insert the new bit at the most significant bit (MSB). This eliminates the decoder. Here is the fixed-up code:

BSV code for sequential multiplication

```
1   Reg #(Bit #(32)) a    <- mkRegU;
2   Reg #(Bit #(32)) b    <- mkRegU;
3   Reg #(Bit #(32)) prod <- mkRegU;
4   Reg #(Bit #(32)) tp   <- mkRegU;
5   Reg #(Bit #(6))  i    <- mkReg (32);
6
7   rule mulStep if (i<32);
8      Bit #(32) m = ((a[0]==0)? 0 : b);  // only look at LSB of a
9      a <= a >> 1;                       // shift a by 1
10     Bit #(33) sum = add32(m,tp,0);
11     prod <= { sum[0], prod [31:1] };   // shift prod by 1, insert at MSB
12     tp <= truncateLSB (sum);
13     i <= i + 1;
14  endrule
```

Fig. 9.1 shows the circuit described by the BSV code. Analyzing this circuit, and comparing it to the circuit for the combinational multiplier:

- The number of instances of the `add32` operator circuit has reduced from 32 to 1, but we have added some registers and muxes.
- The longest combinational path has been reduced from 32 cascaded `add32`s to one `add32` plus a few muxes (and so it can likely run at a higher clock speed).
- To complete each multiplication, the combinational version has a combinational delay corresponding to its longest path. The sequential version takes 32 clock cycles.

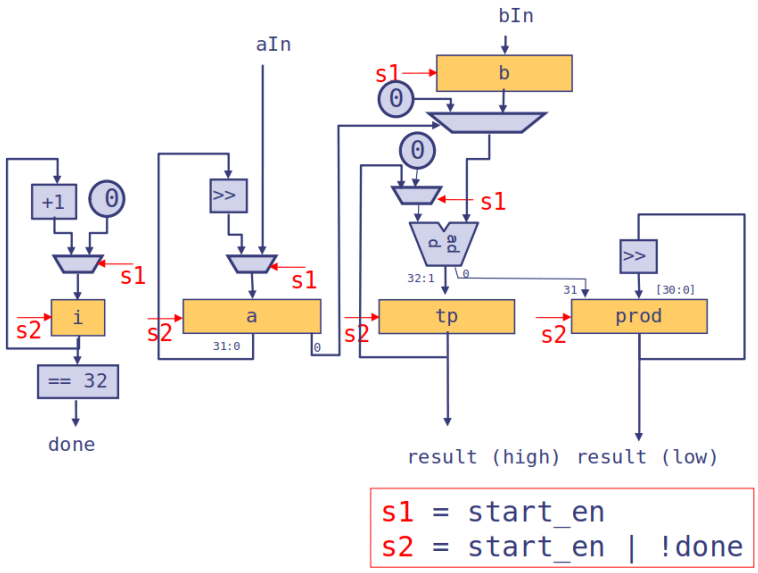


Figure 9.1: Sequential multiplication circuit

9.3 Latency-insensitive modules and compositionality

When building large systems, *latency-insensitivity* is a very useful property. Suppose the environment of a module gives it a request (by invoking a method); the module computes a result and, after some latency (which could be as low as zero), delivers the result back to the environment.

If either the module expects to receive a request on some particular cycle, or the environment expects to receive its result on some particular cycle, there is a certain built-in fragility. We cannot easily substitute the module with a different one that accept arguments or deliver results on different clock cycles, even though it may be better from the point of view of area or power or clock frequency. Similarly, we cannot use the original module in different environments that have different expectations of when arguments and results are communicated. Environments and modules that are resilient to such changes in exactly when arguments and results are communicated are called latency insensitive; it allows for independent evolution of environments and modules.

Latency-insensitive modules typically have FIFO-like interfaces, where one thinks of requests and responses like "packets" or "tokens" flowing in a pipe. We also refer to such modules as being more "elastic"

9.3.1 Packaging the multiplication function as a latency-insensitive module

Fig. 9.2

shows the interface of a module encapsulating the multiplication function, and here is its BSV code:

```
1 interface Multiply;
2    method Action                    start (Bit #(32) a, Bit #(32) b);
```

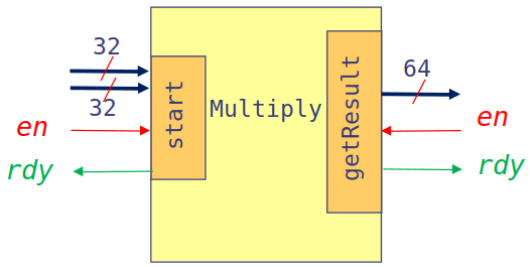


Figure 9.2: Interface of a multiplication module, with guards

```
   method ActionValue #(Bit #(64)) getResult;
endinterface
```

The astute reader may notice that it is almost exactly the same as the `GCD` interface on the right-hand side of Fig. 8.8, the only difference being in the width of the result.[1] Here is a module packaging up our sequential multiplier circuit with the required interface:

```
module mkMultiply (Multiply);
    Reg #(Bit #(32)) a    <- mkRegU;
    Reg #(Bit #(32)) b    <- mkRegU;
    Reg #(Bit #(32)) prod <- mkRegU;
    Reg #(Bit #(32)) tp   <- mkReg (0);
    Reg #(Bit #(6))  i    <- mkReg (32);
    Reg #(Bool)      busy <- mkReg (False);

   rule mulStep if (i < 32);
      Bit #(32) m = ((a [0]==0) ? 0 : b);
      a    <= a >> 1;
      Bit #(33) sum = add32 (m, tp, 0);
      prod <= {sum [0], prod [31:1]};
      tp   <= sum [32:1];
      i    <= i+1;
   endrule

   method Action start (Bit#(32) x, Bit#(32) y) if (!busy);
      a <= x;  b <= y;  i <= 0;
      busy <= True;
   endmethod

   method ActionValue Bit#(64) getResult if ((i==32) && busy);
      busy <= False;
      return {tp, prod};
   endmethod
```

[1] With a little more sophisticated use of types and polymorphism, BSV in fact enables us to unify the interfaces into a single declaration.

And Fig 9.3 shows the circuit of the module.

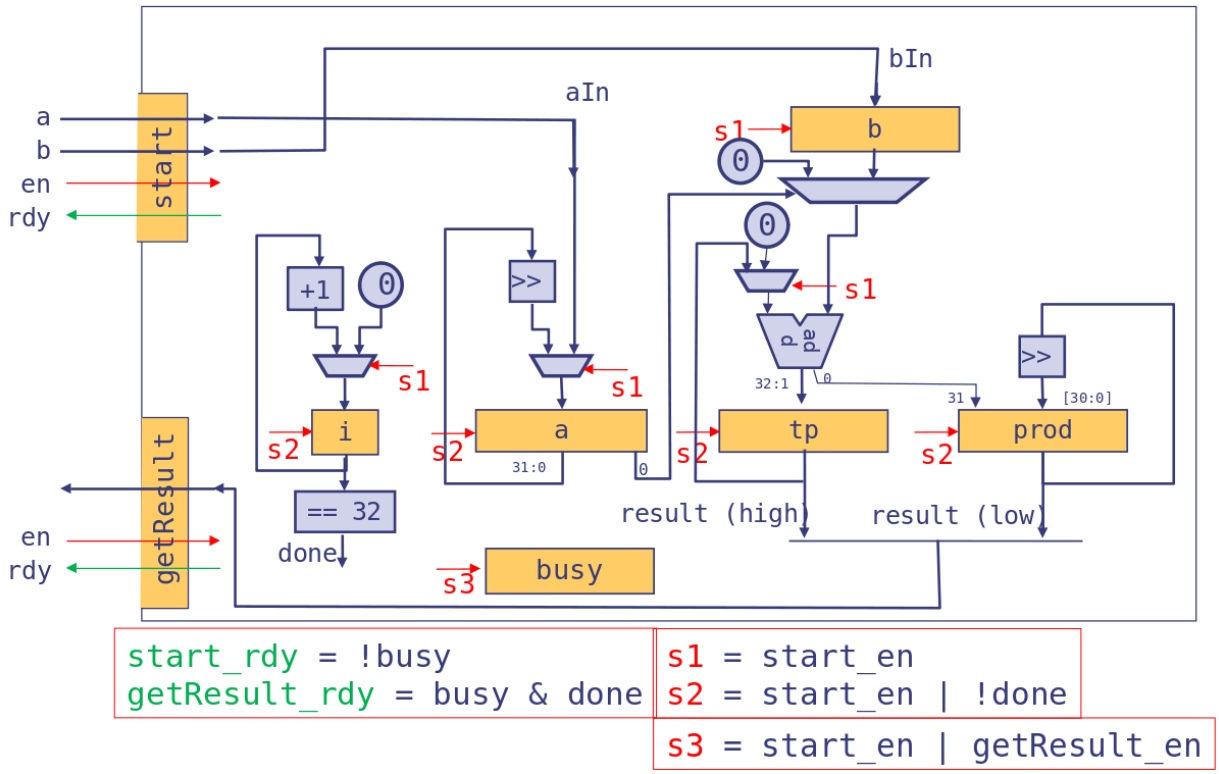


Figure 9.3: Module encapsulating the sequential multiplication circuit

If we encounter a different situation where we want a much lower latency multiplier and we're willing to pay the area cost, we could transparently substitute the above module with the following module that packages up the combinational multiplier instead.

```
function Bit #(64) mul32 (Bit #(32) a, Bit #(32) b);
   ... // combinational multiplier from Sec. 8.2
endfunction

module mkMultiply_Comb (Multiply);
    Reg #(Bit #(64)) prod <- mkRegU
    Reg #(Bool)      busy <- mkReg (False);

   method Action start (Bit#(32) x, Bit#(32) y) if (!busy);
      prod <= mul32 (x, y);
      busy <= True;
   endmethod

   method ActionValue Bit #(64) getResult if (busy);
      busy <= False;
      return prod;
   endmethod
```

Similarly, one can have many other implementations of the same multiplier interface, with different internal algorithms that offer various efficiency tradeoffs (area, power, latency, throughput):

Multiplier module, more alternatives

```
1  module mkMultiply_Block (Multiply);
2
3  module mkMultiply_Booth (Multiply);
```

"Block" multiplication and "Booth" multiplication are two well-known algorithms, or circuit structures, but we do not explore them here.

9.4 Summary

Iterative algorithms can be unfolded in a spatial manner (replicating the loop body for distinct "iterations") or in a temporal manner (re-using the loop body circuit repeatedly, saving intermediate values in registers), trading off latency, area, clock frequency and power.

Iterative algorithms with dynamic termination conditions will require some form of temporal structure, since hardware must have a fixed set of resources but can take a variable amount of time.

Latency-insensitive designs are less fragile in that modules can be developed independently from their environments, and can be used in many different environments, without compromising correctness. This makes it easier to compose reusable modules into complex systems.

9.5 Extra material: Typed registers

In BSV, the contents of registers can be of any type that has a bit representation, not just `Bit #(`n`)`. Examples:

Registers containing values of other types

```
1  Reg #(UInt #(32))     u     <- mkReg (17);
2  Reg #(EthernetHeader) ehdr1 <- mkReg (default_header);
3  Reg #(EthernetHeader) ehdr2 <- mkRegU;
```

In line 1 we create a register `u` containing unsigned 32-bit integer values, with reset value 17. In line 2 we create a register `ehdr1` containing values that are Ethernet packet headers (which is presumably a "struct" type declared elsewhere), initialized to hold a default Ethernet packet header (also presumably declared elsewhere). In line 3 we create register `ehdr2` containing values that are Ethernet packet headers, but with an unspecified reset value.

Registers in BSV are *strongly typed*, *i.e.,* it will not be possible to store a value whose type is `Bit #(32)` in register `u`, even though the "size is right" (32 bits). By "impossible" we mean that this will be checked by the compiler at compilation time and rejected as a type-error. Similarly, `ehdr1` and `ehdr2` will be constrained to only ever contain values of type `EthernetHeader`.

9.6 Extra material: Polymorphic multiply module

Let us now generalize our multiplier circuit so that it doesn't just work with 32-bit arguments, producing a 64-bit result, but works with n-bit arguments and produces a $2n$-bit result. Thus, we could use the same module in other environments which may require, say, a 13-bit multiplier or a 24-bit multiplier. The interface declaration changes to the following:

Polymorphic multiplier module interface

```
interface Multiply #(numeric type tn);
   method Action             start  (Bit#(tn) a, Bit#(tn) b);
   method Bit#(TAdd#(tn,tn)) result;
endinterface
```

And the module definition changes to the following:

Polymorphic multiplier module

```
module mkMultiply (Multiply #(tn));
   Integer n = valueOf (tn);

   Reg #(Bit#(n)) a    <- mkRegU;
   Reg #(Bit#(n)) b    <- mkRegU;
   Reg #(Bit#(n)) prod <- mkRegU;
   Reg #(Bit#(n)) tp   <- mkRegU;
   Reg #(Bit#(TAdd#(1,TLog#(tn)))) i <- mkReg (fromInteger(n));

   rule mulStep if (i<32);
      Bit#(n) m   = (a[0]==0)? 0 : b;
      a <= a >> 1;
      Bit#(Tadd#(tn,1)) sum = addN(m,tp,0);
      prod <= { sum[0], prod [n-1:1] };
      tp <= truncateLSB(sum);
      i <= i + 1;
   endrule

   method Action start(Bit#(n) aIn, Bit#(n) bIn) if (i==fromInteger(n));
      a <= aIn;
      b <= bIn;
      i <= 0;
      tp <= 0;
      prod <= 0;
   endmethod

   method Bit#(64) result if (i==fromInteger(n));
      return {tp,prod};
   endmethod
endmodule
```

Note the use of the pseudo-function `valueOf` to convert from an integer in the type domain to an integer in the value domain; the use of the function `fromInteger` to convert

from type `Integer` to type `Bit#(...)`, and the use of type-domain operators like `TAdd`, `TLog` and so on to derive related numeric types.

Chapter 10

Hardware Synthesis: from BSV to RTL

10.1 Introduction

In this chapter we discuss in a more detail how BSV code is *synthesized* into circuits (we have already seen aspects of this in previous chapters).

"Synthesis" is just a kind of compilation. A classical compiler translates a program in a high-level language (such as C, C++ or Python) into a lower-level language (assembly language, machine code, or bytecode). In our case, BSV is the high-level language, and we wish to translate the high-level constructs in BSV code (modules with guarded methods and rules) into lower-level constructs such as wires, gates and registers. This target (wires, gates and registers) is traditionally known as Register-Transfer Language (RTL).[1]

In this chapter, our lower-level language will just be diagrams (we show schematics of circuits), but in practice a BSV compiler translates into Verilog, the "assembly language" of hardware design. Just like, in software compilation, we have Assemblers that translate assembly language into machine code, similarly in hardware synthesis there are lower-level synthesis tools that will translate Verilog RTL into actual gates and wires.

10.1.1 General principles

Each BSV module in translated into a sequential circuit. For our present discussion, the only "primitive" module is a register; in general one can have an arbitrary library of primitives. The implementation of primitives is outside the scope of BSV.

A module instantiates registers and other modules. This results in a *module instance hierarchy*, *i.e.,* a tree of module instances below a top-level module instance, at the leaves of which are registers.

A module's BSV interface precisely determines a corresponding set of input and output wires in hardware. In other words, given an interface type declaration, with no additional information we can draw exactly the input and output wires into which it will be translated.

[1] Coding directly in RTL is still the predominant practice, using HDLs like Verilog, SystemVerilog and VHDL.

Each interface method is synthesized into a combinational circuit whose inputs include the method arguments (if any) and whose outputs include the method results (if any). For all methods, there is an additional output called the READY signal. For Action and ActionValue methods, there is an additional input called the ENABLE signal.

Each rule in a module also defines a combinational circuit. Its inputs and outputs come from results and arguments of methods it invokes, respectively. One of its outputs is a boolean representing the value of the rule's explicit condition as well as implicit conditions of methods invoked in the rule; this signal is the rule's READY signal, also called its `CAN_FIRE` signal.

The combinational logic of all the rules and methods are connected to the instantiated registers and modules using muxes (multiplexers).

10.2 Interface types precisely define input/output wires

A module's BSV interface precisely determines a corresponding set of input and output wires in hardware. In other words, given an interface type declaration, with no additional information we can draw exactly the input and output wires into which it will be translated. Consider our interface to the GCD module:

```
1 interface GCD;
2    method Action                    start (Bit #(32) a, Bit #(32) b);
3    method ActionValue #(Bit #(32)) getResult;
```

The corresponding input and output wires are shown in Fig 10.1.

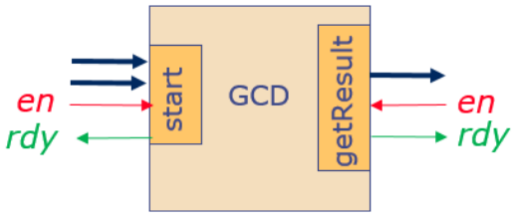


Figure 10.1: Interface wires for the GCD interface

- Each method argument becomes an input bus (bundle of wires). Some methods may not have arguments, in which case this bus is absent.
- Each method result becomes an output bus (bundle of wires). Value methods and ActionValue methods can have a result. Action methods have no result, and so this bus is absent.
- Each method has an output "ready" (RDY) wire (we will always show ready wires in green).
- Each Action or ActionValue method has an input "enable" (EN) wire (we will always show enable wires in red).

With a little practice, BSV programmers automatically visualize these wires when they read a BSV interface type.

10.3 Registers are just primitive modules, with interfaces

While we have not emphasized this so far, we now observe that even registers are just modules. Instantiating a register is conceptually the same as instantiating any other module. We call them "primitive" modules in that they are taken as "given" by fiat by the language; their internal implementation is not defined within BSV.

Like all modules, a register has an interface, shown below:

```
interface Reg #(numeric type n);
   method Action   _write (Bit #(n) x);
   method Bit #(n) _read;
```

It was a `_read` and a `_write` method. When we write a statement such as this:

```
   y <= x + 1;
```

this is just a syntactic convenience to make it look like a traditional assignment statement. Technically, using method-invocation notation, we would write:

```
   y._write (x._read + 1);
```

(This syntax is also acceptable, but we rarely use it.)

The input and output wires implied by the above interface type are shown in Fig. 10.2. However, for primitive registers, the READY signals are always True—the `_read` and

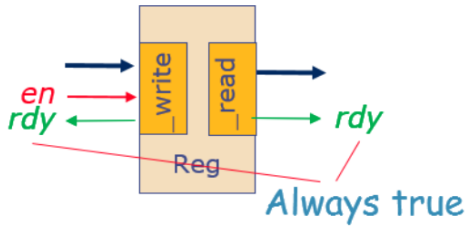


Figure 10.2: A register is just a primitive module

`_write` methods can be invoked at any time. Registers are instantiated using the BSV library modules `mkReg(i)` (with initial reset value i) and `mkRegU` (with no specified initial reset value). The user can also define their own modules that have the same interface (and sometimes it is indeed convenient to do so).

10.4 Schematic drawing conventions in this chapter

Drawing schematics is a useful pedagogical way to demystify what circuits we get from our BSV codes. But they are just that; no one actually designs circuits using schematics, except for initial rough cuts, sketches etc. At least since the 1980s people switched to text-based

languages like Verilog and VHDL because they are easier to scale to large systems, and to maintain.

Emphasizing the pedagogical role, we will use a few conventions to avoid clutter and to keep our schematics simple and understandable. Consider the following interface:

```
1 interface GFMI;
2    method Action                    start (Bit #(n) a);
3    method ActionValue #(Bit #(n)) getResult;
```

which can be the interface for a number of modules that accept some input using `start`, perform some computation F, and deliver a result using `getResult`. Here, "GFMI" stands for Generalized F module interface. From its interface type, we can visualize the input and output wires shown in Fig. 10.3. When we draw a module that implements this interface,

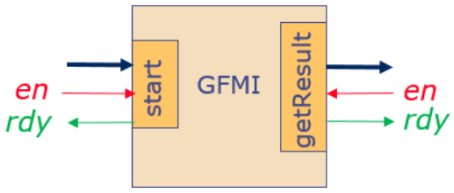


Figure 10.3: Input and output wires for the GFMI interface

we will often replicate the interface to separate out input and output wires, as shown in Fig. 10.4. The input signals (arguments and ENABLE signals) will be shown on the left,

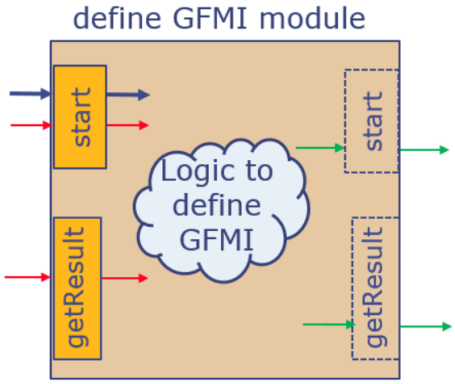


Figure 10.4: A module with a GFMI interface

and the output signals (results and READY signals) will be shown on the right.

Similarly, when we use such a module, we may replicate its methods, as shown in Fig. 10.5. The output signals (results and READY signals) will be shown on the left, and the input signals (arguments and ENABLE signals) will be shown on the right.

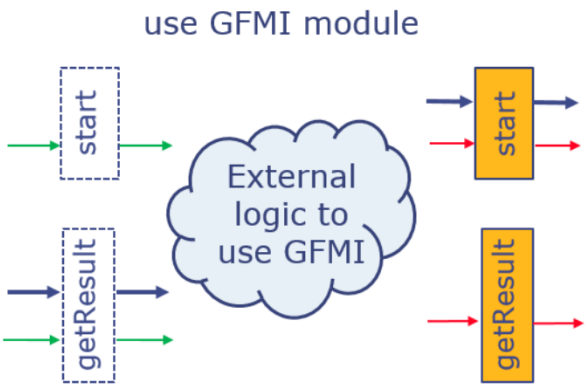


Figure 10.5: Using a module with a GFMI interface

10.5 General principles of the READY-ENABLE interface protocol

The READY output signal of a method in a module interface communicates from the module to the environment whether it is currently legal for the environment to invoke that method. The ENABLE input signal (for Action and ActionValue methods) communicates, from the environment, whether the environment is actually currently invoking the method or not. Thus, the ENABLE signal must depend on the READY signal; this is illustrated in Fig. 10.6.

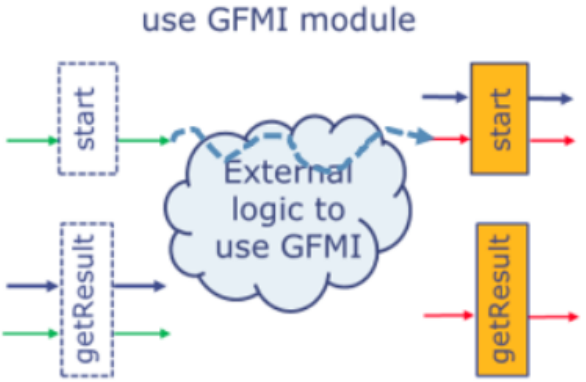


Figure 10.6: The ENABLE signal always depends on the READY signal

Conversely, the READY signal can never depend on the ENABLE signal; this is illustrated in Fig. 10.7. Otherwise, we would create a combinational cycle. The BSV compiler guarantees these properties in the circuits that it generates.

10.6 Example: A FIFO Module

Let us analyze our FIFO module code from Sec. 8.5 in detail to see the circuits produced. Here is its interface declaration:

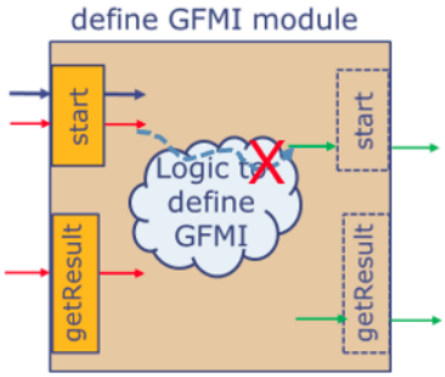


Figure 10.7: The READY signal can never depend on the ENABLE signal

```
interface FIFO #(Bit #(n));
   // enqueueue side
   method Action    enq (Bit #(n) x);

   // dequeue side
   method Action    deq;
   method Bit #(n)  first;
endinterface
```

and here is a module implementing it:

```
module mkFIFO (FIFO #(Bit #(n)));
   Reg #(Bit #(n)) d <- mkRegU;             // Data
   Reg #(Bool)     v <- mkReg (False);      // Valid?

   // enqueueue side
   method Action  enq (Bit #(n) x);
      d <= x;
      v <= True;
   endmethod

   // dequeue side
   method Action  deq;
      v <= False;
   endmethod

   method Bit #(n)  first = d;
endmodule
```

We are going to fill in the schematic for the module implementation gradually.

10.6.1 The interface methods

First, from the interface declaration, we can immediately read off the input-output wires, as shown in Fig. 10.8.

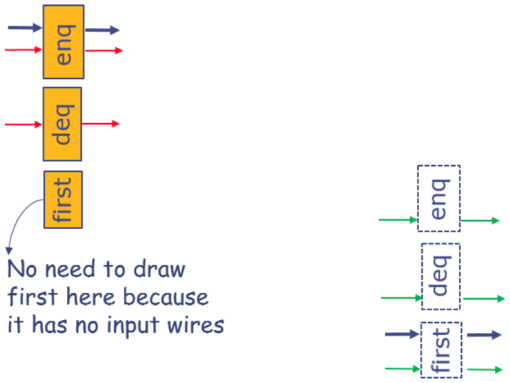


Figure 10.8: FIFO input and output wires

The `enq` method has an input argument and ENABLE (on the left) and an output READY signal (in the replicated dotted-line box on the right). The `deq` method has an input ENABLE (on the left) and an output READY signal (in the replicated dotted-line box on the right). The `first` method only has outputs—its READY signal and result value, shown only in the dotted-line box on the right; from now on we will omit the `first` box on the left since it has no input wires.

10.6.2 The internal state elements

Next, let us instantiate the `d` nd `v` registers, as shown in Fig. 10.9. The registers' `_read`

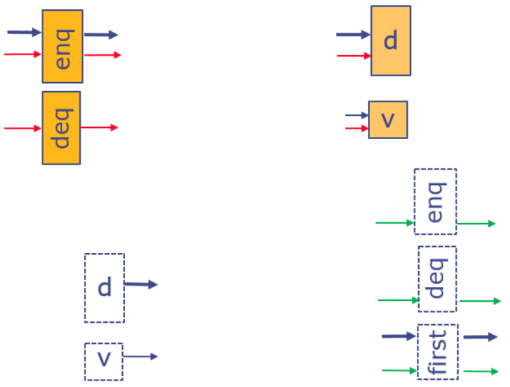


Figure 10.9: FIFO internal registers `d` and `v`

method signals are shown in the dotted-line boxes on the bottom left. Each register has an output value, and a READY signal (which, for a register, is always true). The registers' `_write` method signals are shown in the boxes on the top right. Each register has an input value (to be stored in the register), and an ENABLE signal. The `_write` method should also have a READY signal, but since this is always true in a register, we have not shown it in the diagram.

10.6.3 The circuits for the methods, individually

Next, let us look at the synthesis of each method, one at a time. We start with the `enq` method in Fig. 10.10. The `enq` method is ready whenever `v` is 0, thus the output of `v` is

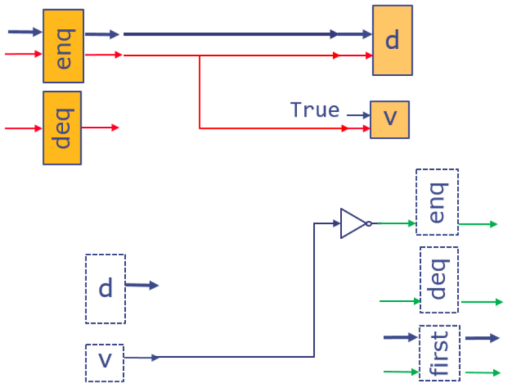


Figure 10.10: FIFO `enq` method

fed via an inverter into the READY signal of `enq`. The argument to `enq` goes straight into the `_write` method of register `d`. Whenever we enqueue, *i.e.*, the ENABLE signal of `enq` is true, we enable the `_write` method of `d`, and we also write the constant `True` (bit 1) into `v`.

Fig. 10.11 shows the synthesis of the `deq` method, viewed on its own. The `deq` method

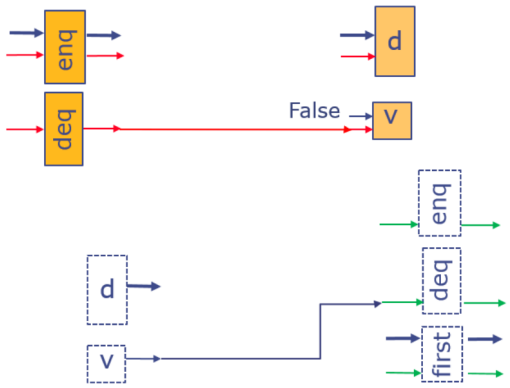


Figure 10.11: FIFO `deq` method

is ready whenever `v` is 1, thus the output of `v` is fed into the READY signal of `deq`. When the external environment invokes `deq`, it asserts its ENABLE signal, which we use to invoke the `_write` method of `v`, loading in the constant `False` (bit 0).

Finally, Fig. 10.12 shows the synthesis of the `first` method, viewed on its own. The `first` method is ready whenever `v` is 1, thus the output of `v` is fed into the READY signal of `first`. The value returned by `first` is the value in the `d` register, ie the output of the `_read` method on `d`.

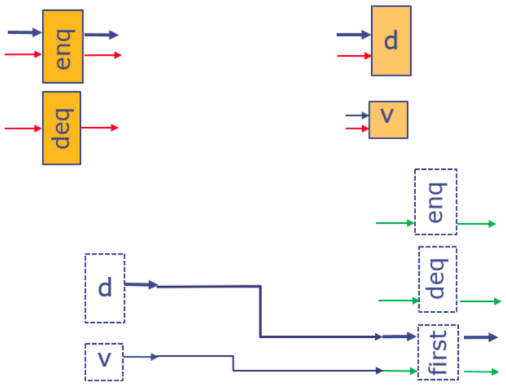


Figure 10.12: FIFO `first` method

10.6.4 The methods, combined

We must now combine the circuits we've shown separately for each method. Whenever we have an input that is fed from more than one input in the separate schematics, in the combined circuit we must "funnel" them through a mux (multiplexer). This is depicted in Fig. 10.13. The behavior of the mux is the following. First, we assume that at most one of

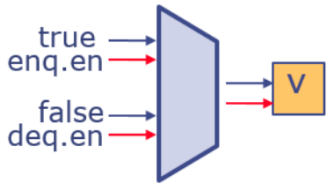


Figure 10.13: A mux (multiplexer) to combine signals

the input ENABLE signals `enq.en` and `deq.en` is true (that is indeed the case for our FIFO circuit; later we will see how we guarantee this in general). When either `enq.en` or `deq.en` is true, the output ENABLE signal is true (the ENABLE signal of the `_write` method of `v`). Further, the mux passes the corresponding data through to the `v` (the argument of the `_read`) method. Here, the data is either the constant `True` (1) of the constant `False` (0).

Fig. 10.14 shows how we could implement the mux. The signals `v1` and `v2` correspond

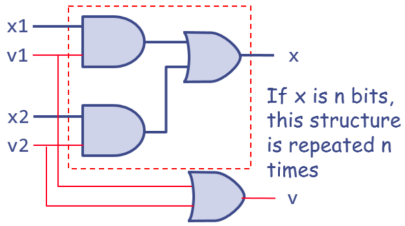


Figure 10.14: Implementation of the mux (multiplexer)

to our ENABLE signals—if either is true, the output `v` is true. When `v1` and `v2` are 0, the output `x` is 0. When `v1` is 1 and `v2` is 0, `x` is equal to `x1`. When `v1` is 0 and `v2` is 1, `x` is equal to `x2`. The outputs can also be expressed using these boolean equations:

x = (v1 & x1) | (v2 & x2)
v = (v1 | v2)

We can also see that in the case that we disallow, ie `v1` and `v2` both true, the output `x` will be an OR of `x1` and `x2` which may be a meaningless value. Also, although the mux is shown for one signal (`x1`, `x2`, `x`) we can simply replicate the structure in the dotted line if they have more bits.

In our FIFO, the only method-input that is fed by different signals in the individual method schematics is the input to the `v` register. Thus we can combine the method schematics into a composite schematic by using a mux there, as shown in Fig 10.15.

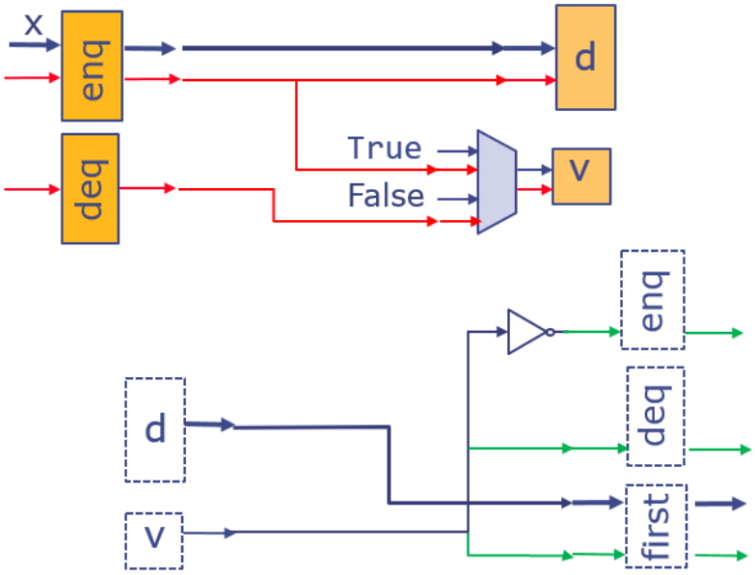


Figure 10.15: Combining the FIFO methods

As discussed earlier, the two control signals of the mux, coming from the ENABLE signals of the `enq` and `deq` signals should never be true simultaneously. We can reason that this will indeed be the case: in our FIFO module, the READY signals of `enq` and `deq` are mutually exclusive (only one of them can be true at a time), and by our READY-ENABLE protocol, therefore, since ENABLE signals depend on READY signals, only one of the ENABLE signals can be true.

Fig. 10.16 is simply a redrawing of our diagram with a more conventional layout. We can confirm that the READY signals only depend on the module's internal state. As indicated in our introduction, our module is indeed a sequential circuit, since it contains state elements (registers). However, this particular example contains no feedback into registers.

10.6.5 An impractical alternative circuit specification

As mentioned in the last chapter, the FIFO logic could have been specified as a next-state transition table, shown here:

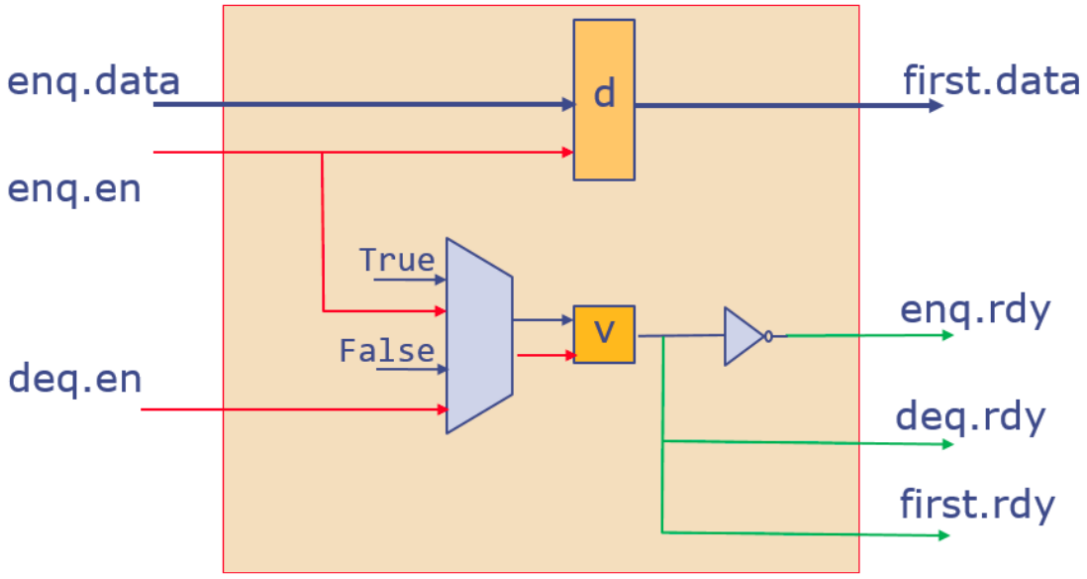


Figure 10.16: The FIFO, redrawn more conventionally

inputs			state		next state		outputs			
enq en	enq data	deq en	d^t	v^t	d^{t+1}	v^{t+1}	enq rdy	deq rdy	first rdy	first data
0	x	0	x	0	x	0	1	0	0	-
1	d	0	x	0	d	1	0	1	1	-
0	x	0	d	1	d	1	0	1	1	d
0	x	1	d	1	-	0	1	0	0	d

This can also be used to specify illegal inputs:

inputs			state		next state		outputs				Why illegal
enq en	enq data	deq en	d^t	v^t	d^{t+1}	v^{t+1}	enq rdy	deq rdy	first rdy	first data	
1				1			0				enq when not RDY
		1		0				0			deq when not RDY
1		1		0			1	0			deq when not RDY, and simul enq/deq
1		1		1			0	1			enq when not RDY, and simul enq/deq

As you can see, not only would it be very tedious and error-prone to construct such tables, especially for larger systems (more state elements, more inputs), but the behavior is not at all intuitive or obvious to the reader.

10.6.6 Summary

As Fig. 10.15 graphically illustrates, the circuit for a module is just a combinational circuit. The input of the module's combinational circuit come from:

- Arguments of the module's methods (Value, Action and ActionValue);
- ENABLE signals of the module's methods (Action and ActionValue);

- Results of sub-module methods (Value and ActionValue);
- READY signals of sub-module methods (Value, Action and ActionValue);

Conversely, the output of the module's combinational circuit go to:

- Results of the module's methods (Value and ActionValue);
- READY signals of the module's methods (Value, Action and ActionValue);

- Arguments of sub-module methods (Value, Action and ActionValue);
- ENABLE signals of sub-module methods (Action and ActionValue);

10.7 Example: GCD

In this section we look at synthesis of the GCD example which, unlike the FIFO example, also has an internal rule and feedback to internal registers. The code for GCD is recapped below. First, the interface declaration:

```
interface GCD;
   method Action                    start (Bit #(32) a, Bit #(32) b);
   method ActionValue #(Bit #(32)) getResult;
endinterface
```

and then a module implementing that interface:

```
module mkGCD (GCD);
   Reg #(Bit #(32)) x    <- mkReg (0);
   Reg #(Bit #(32)) y    <- mkReg (0);
   Reg #(Bool)      busy <- mkReg (False);

   rule gcd;
     if (x >= y) x <= x - y;
     else if (x != 0) begin
        x <= y;
        y <= x;
     end
   endrule

   method Action start (Bit #(32) a, Bit #(32) b) if (! busy);
      x <= a;
      y <= b;
      busy <= True;
   endmethod

   method ActionValue #(Bit #(32)) getResult if (busy && (x==0));
```

```
21        busy <= False;
22        return y;
23     endmethod
24  endmodule
```

As before, from the interface declaration and sub-modules (modules instantiated within the module) we can directly read off the input and output wires of the module, shown in Fig. 10.17. On the left, we have all the input wires from the module's interface methods

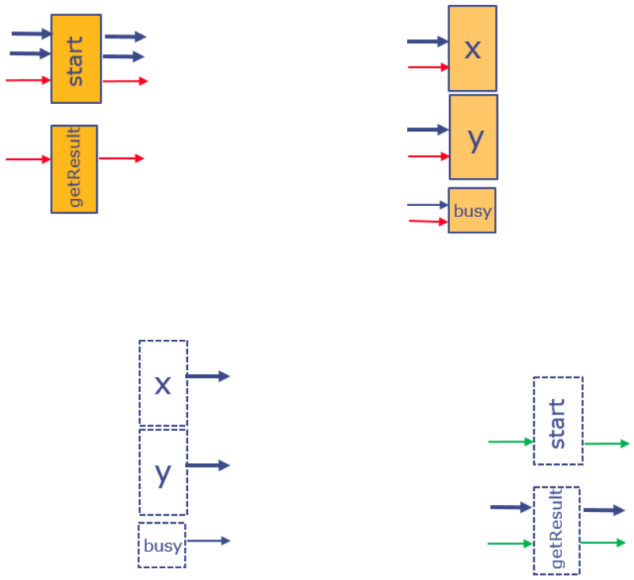


Figure 10.17: The inputs and outputs for the GCD circuit

(arguments and ENABLE signals), and all the output wires from the sub-module interface methods (results and READY signals), specifically, the `_read` methods of the registers `x`, `y` and `busy`.

On the right, we have all the output wires to the module's interface methods (results and READY signals), and all the input wires to the sub-module interface methods (arguments and ENABLE signals), specifically, the `_write` methods of the registers `x`, `y` and `busy`).

Next, we synthesize the circuits for the methods, individually, starting with the `start` method, shown in Fig. 10.18. We omit the boring nitty-gritty details of the combinational expressions, just depicting them by little "clouds", one representing the Actions performed by the `start` method and one representing the guard on the `start` method.

Next, we synthesize the `getResult` method, shown in Fig. 10.19.

Next, we synthesize the `gcd` rule, shown in Fig. 10.20. It is synthesized just like a method, except that none of its inputs come from the module's method; all of its inputs come only from sub-module outputs. Further, none of its outputs to the module's method outputs; all of its outputs go only to sub-module inputs.

Finally, we put these individual circuits together using muxes to combine multiple sources for each input. This is shown in Fig. 10.21.

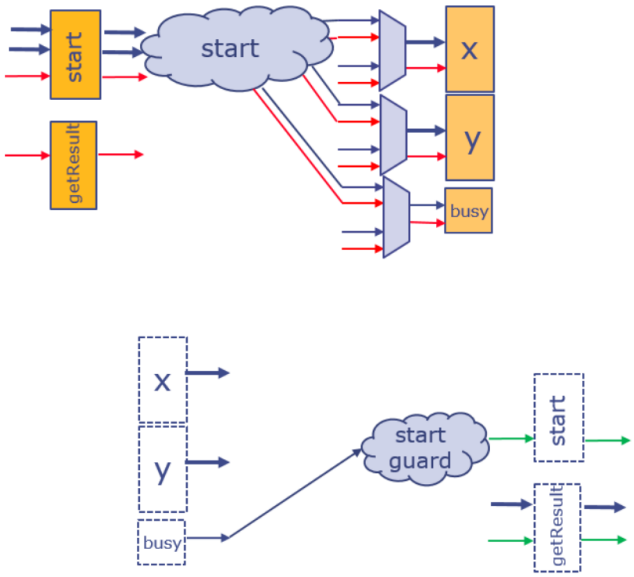


Figure 10.18: The GCD `start` method.

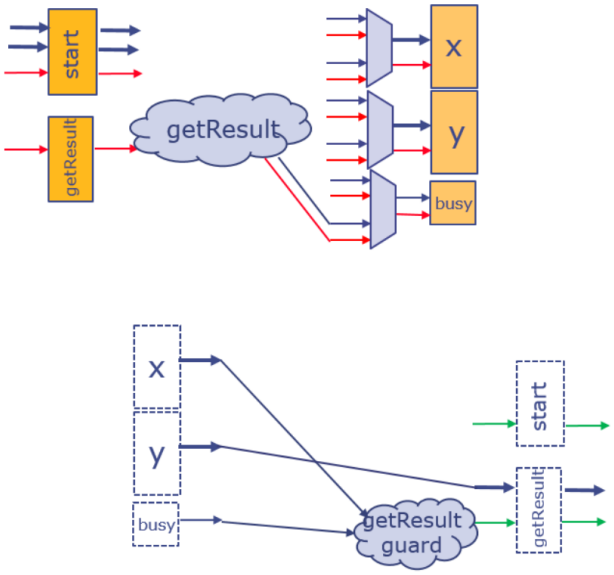


Figure 10.19: The GCD `getResult` method.

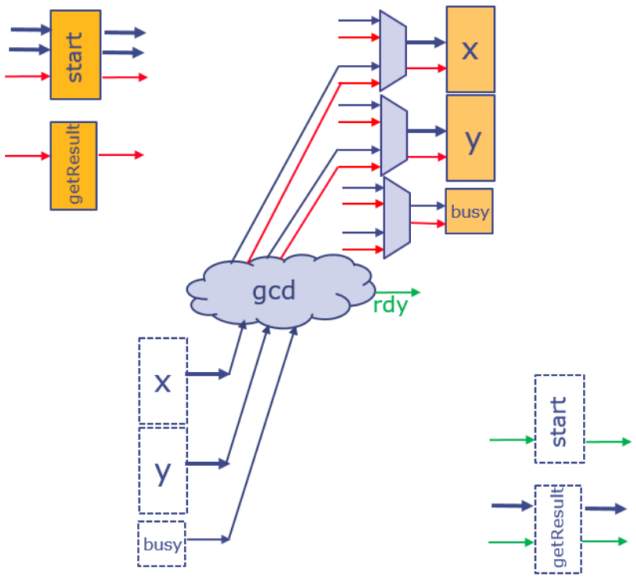


Figure 10.20: The GCD `gcd` rule.

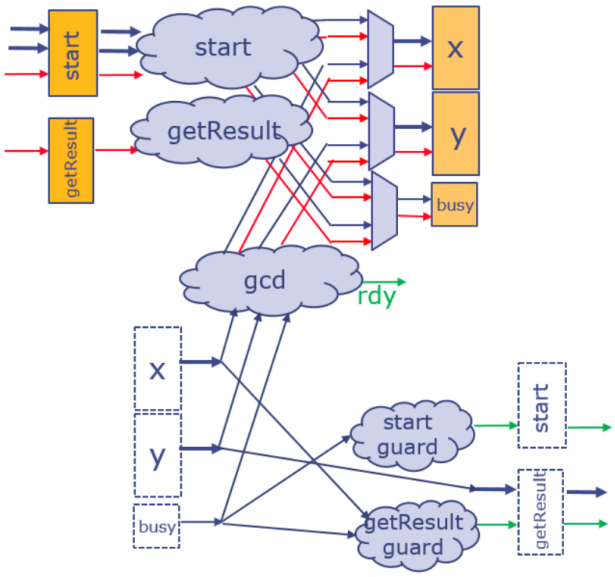


Figure 10.21: GCD combined circuit.

Once again, we can verify that, for each mux funneling data into registers, at most one of the ENABLE signals is true. The muxes for `x` and `y` are driven from the `start` method and `gcd` rules, whose READY signals are disjoint. For the mux driving `busy`, the ENABLEs are from the `start` and `getResult` methods, which depend on their READY signals, which are disjoint.

10.7.1 Example: Ready Signal in Multi-GCD

In Sec. 8.8 we studied a "multi-GCD" circuit where we replicated the GCD module inside a larger module that had the same GCD interface, in order to improve the "throughput" (how many GCDs can be computed in a fixed interval). The structure was depicted in Fig. 8.10.

In this section, we look at how the READY signal of the `start` method of the multi-GCD module is synthesized using the READY signals of the internal instances of the GCD module. The code for the outer `start` method is recapped below:

```
1  module mkMultiGCD (GCD);
2     ...
3     method Action start (Bit #(32) a, Bit #(32) b);
4        if (turnI)
5           gcd1.start (a,b);
6        else
7           gcd2.start (a,b);
8        turnI <= (! turnI);
9     endmethod
10    ...
11 endmodule
```

Fig. 10.22 shows the circuit for the READY signal of the `start` method of multi-GCD. In

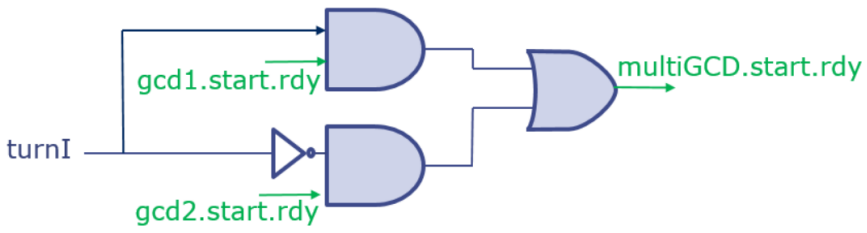


Figure 10.22: READY signal for `start` method of multi-GCD.

words: if `gcd1`'s start method is ready and `turnI` is true, then multi-GCD's `start` is ready to accept an input (which will be passed in to `gcd1`. If `gcd2`'s start method is ready and `turnI` is false, then multi-GCD's `start` is ready to accept an input (which will be passed in to `gcd2`.

10.8 Composition of sequential machines yields a sequential machine

Here is a module that packages an arbitrary combinational circuit f (a function) so that we can stream a sequence of inputs x_0, x_1, x_2, ... into the module, and it yields a corresponding stream of outputs $f(x_0)$, $f(x_1)$, $f(x_2)$, ... First, we declare its interface:

```
interface GMFI#(numeric type n);
   method Action                 start (Bit #(n) a);
   method ActionValue (Bit #(n)) getResult;
endinterface
```

A module to implement this is straightforward:

```
module mkstreamf (GMFI #(n));
   FIFO #(Bit #(n)) fifo  <- mkFIFO;

   method Action start (Bit #(n)  x);
      fifo.enq (f (x));
   endmethod

   method ActionValue (Bit#(n)) getResult;
      fifo.deq;
      return fifo.first ();
   endmethod
endmodule
```

When we invoke `start(x)` we immediately apply the function f to `x` and enqueue the result in the internal fifo. The `getResult` method simply yields these results. Fig. 10.23 shows the synthesis of the `start` method. The circuits can actually be simpler: given our

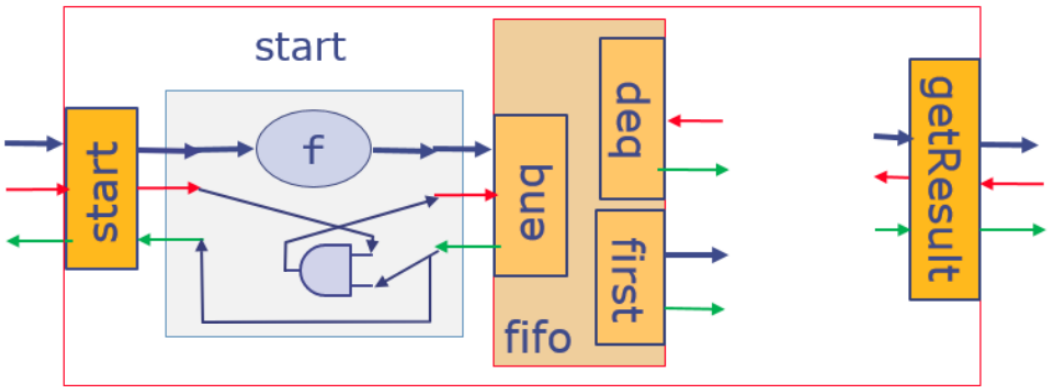


Figure 10.23: Input and output wires and methods in `mkstreamf`.

READY-ENABLE protocol, the AND gate in the `start` method is actually redundant and

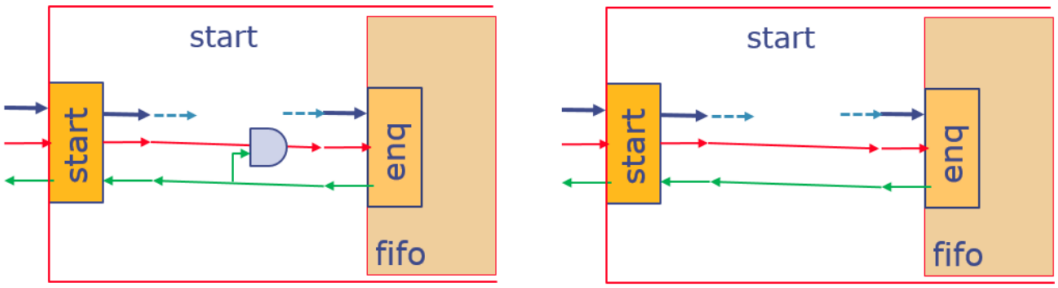


Figure 10.24: Simplifying the circuit for the `start` method of `mkstreamf`.

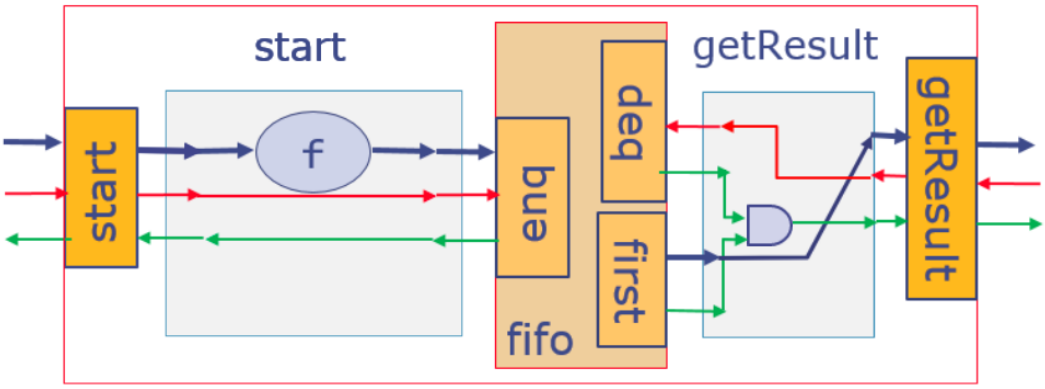


Figure 10.25: Adding the `getResult` method for `mkstreamf`.

can be removed, as shown in Fig. 10.24. Fig.10.25 shows the simplified `start` synthesis and also adds the synthesis of the `getResult` circuit.

Let us "inline" the FIFO module, *i.e.*, substitute it *in situ*. The circuit for the FIFO module was shown in Fig. 10.16. Fig. 10.26 shows the result, where we have also simplified

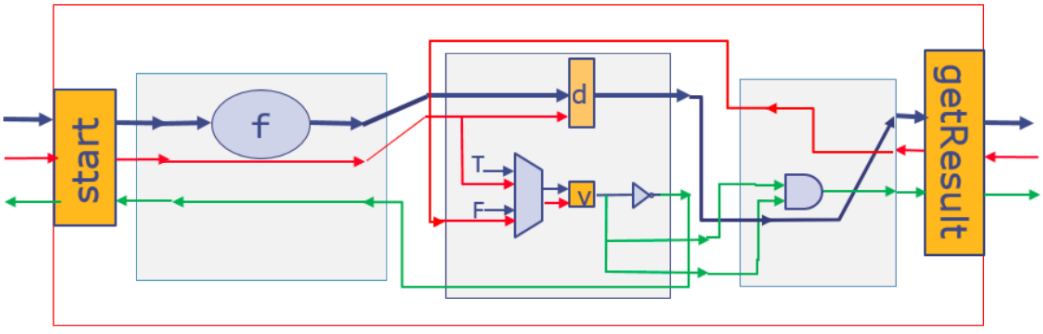


Figure 10.26: `mkstreamf` with the FIFO inlined.

the `getResult` circuit by removing a redundant AND gate just like we did for the `start` circuit.

In summary: the FIFO module is a sequential machine. The `mkstreamf` module is a sequential machine. Composing them together (by inlining) also results in a sequential machine.

10.9 Summary: The Module Hierarchy is a Hierarchical Sequential Circuit

Stepping back from the details of the examples in the last few sections, Fig. 10.27 shows

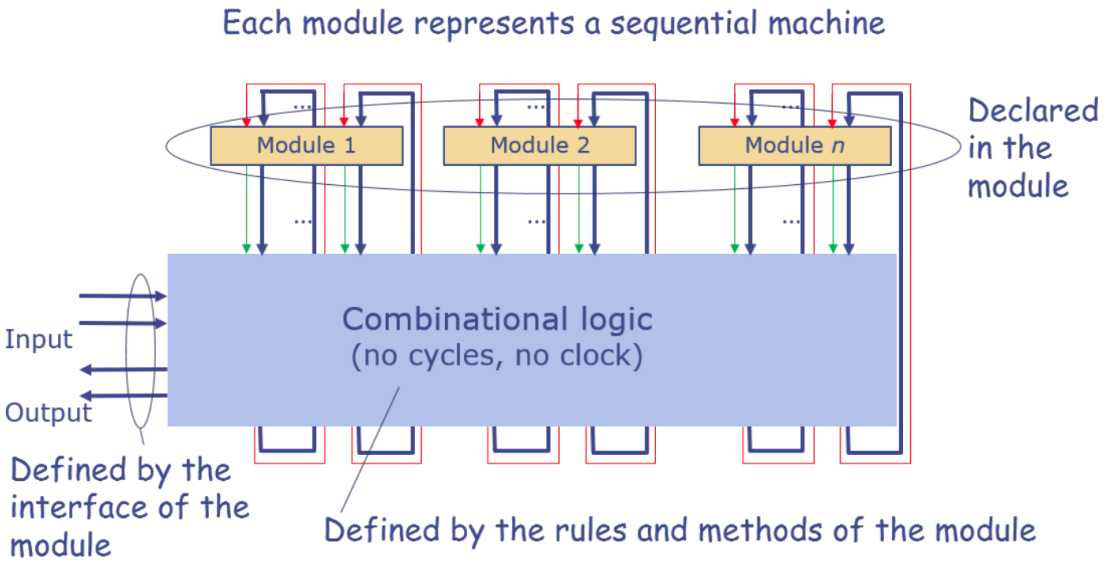


Figure 10.27: The general synthesis schema.

the general synthesis schema for a module.

A: Input wires of the module are determined from arguments of all interface methods, and ENABLE signals of Action and ActionValue methods.

B: Output wires of the module are determined from results of interface methods, and READY signals of all methods.

C: Inside the module, sub-modules are instantiated. For each interface method of each sub-module, that method also has input output wires. Arguments and ENABLE signals of those methods are inputs to the sub-module; results and READY signals of those methods are outputs of the sub-modules.

D: We can synthesize the combinational circuit for each rule and method of the current module individually, connecting the wires of A, Band C.

10.9.1 Synthesizing the guard of a method

Fig. 10.28 shows the general considerations in synthesizing the guard of a module's method.

Inside the guard expression, it likely invokes methods of sub-modules. However, because a guard expression always has type `Bool`, BSV's strong-typing system guarantees that the guard can never invoke an Action or ActionValue sub-module method, either directly or indirectly (in a sub-module's sub-module, for example). A guard expression can only ever invoke sub-module value methods. Further, synthesizing a guard never produces any ENABLE signals.

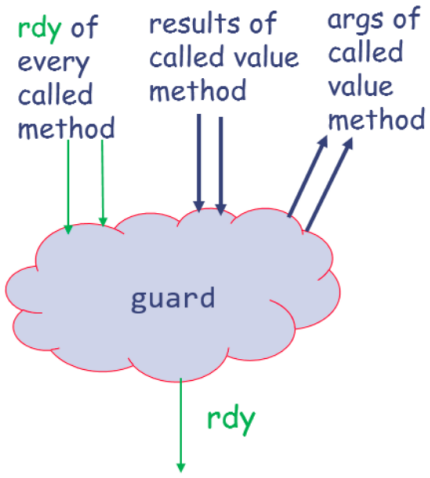


Figure 10.28: Synthesis of a method guard.

The guard synthesizes into a combinational circuit where:

- Inputs come from results of invoked sub-module methods.
- Inputs come from READY signal of invoked sub-module methods.
- The single boolean output is the value of the guard expression.

Note that a method's guard cannot involve the arguments of the method, since this can result in combinational cycles:

- The method's READY signal is the value of the guard;
- until this READY signal is true, the environment cannot invoke the method;
- until the method is invoked, its arguments are not valid;
- and, thus, a method's READY signal cannot depend on the method's arguments. It can only depend on sub-module methods (state elements).

10.9.2 Synthesizing the body of a value method

Fig. 10.29 shows the general considerations in synthesizing the body of a value method (*i.e.*,

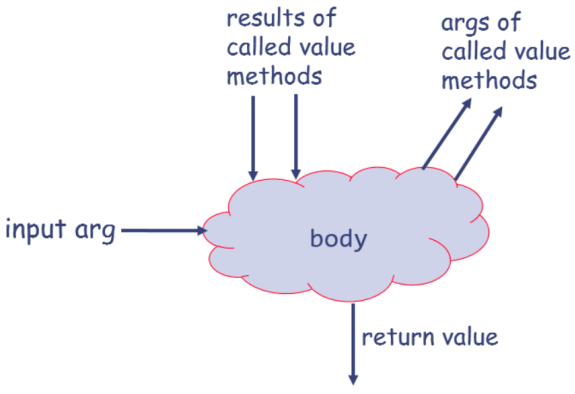


Figure 10.29: Synthesis of the body of a value method.

not and Action or ActionValue method). This is just like compiling any combinational

expression. Inside the method body, it likely invokes methods of sub-modules (but for course these could only be value methods of sub-modules). The method synthesizes into a combinational circuit where:

- Inputs come from results of invoked sub-module methods.
- Inputs come from method arguments.
- Outputs include the method output.
- Outputs include arguments of invoked sub-module methods.

10.9.3 Synthesizing the body of an Action or ActionValue method

Fig. 10.30 shows the general considerations in synthesizing the body of a module's method.

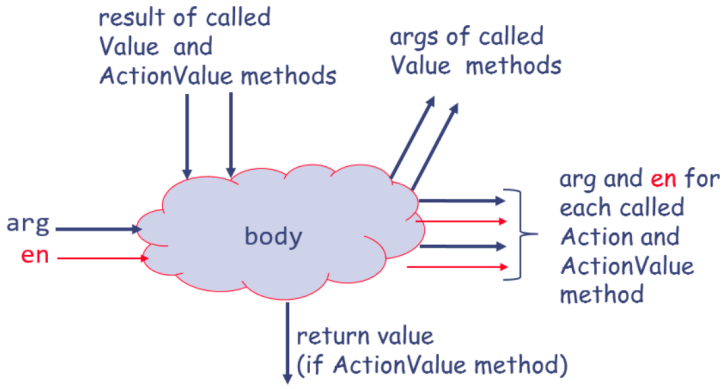


Figure 10.30: Synthesis of the body of an Action or ActionValue method.

Inside the method body, it likely invokes methods of sub-modules. The method synthesizes into a combinational circuit where:

- Inputs come from results of invoked sub-module methods.
- Inputs come from method arguments.
- Inputs come from method ENABLE signals (if this is an Action or ActionValue method).

- Outputs include the method output.
- Outputs include arguments of invoked sub-module methods.
- Outputs include the ENABLE signals of invoked sub-module Action and ActionValue methods.

10.9.4 Summary: synthesizing the method

Fig. 10.31 combines the components just described to provide an overview of a method's synthesis.

10.9.5 Synthesizing a rule in a module

A rule in a module is synthesized similarly to a method except that all its inputs and outputs connect only to sub-module methods, and never to any of the module's own methods.

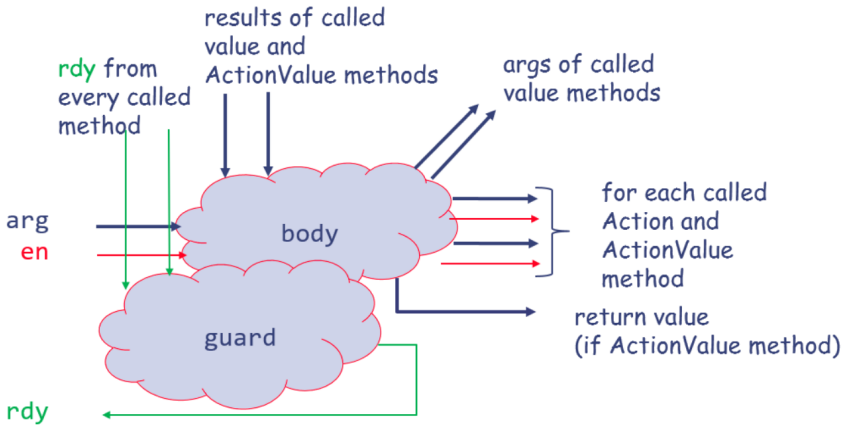


Figure 10.31: Synthesis of a method (summary).

A rule's guard is synthesized similar to a method's guard. Like a method's guard, it's type is `Bool` and the BSV type system guarantees, therefore, that it cannot invoke a sub-module's Action or ActionValue method, either directly or indirectly. The boolean value of the rule's guard is called its `CAN_FIRE` signal.

A rule's body is synthesized similar to an Action or ActionValue method's body. It's outputs will include ENABLE signals for sub-module Action and ActionValue methods.

Fig. 10.32 provides and overview of rule synthesis.

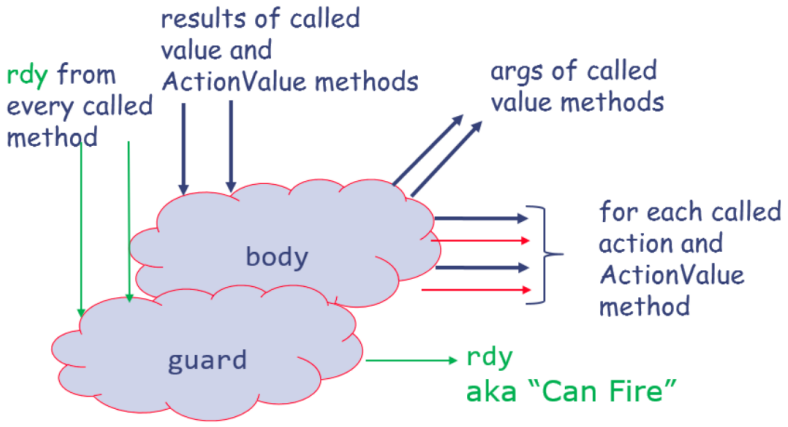


Figure 10.32: Synthesis of a rule.

10.9.6 A preview of rule scheduling

In Sec. 10.6.4 and Fig 10.14 we discussed "funneling" multiple inputs into a method using a mux (multiplexer). We further observed that, for an n input mux, either none or at most one of its inputs should be enabled to pass-through to the output; we should never enable two or more inputs at the same time. But how do we ensure this? Consider this made-up example:

```
module mkEx (...);

   Reg #(Bit #(n)) x <- mkRegU;

   rule foo if p(x);
      x <= e1;
   endrule

   rule baz if q(x);
      x <= e2;
   endmethod
endmodule
```

When $p(x)$ is true, rule `foo` wants to assign the value of `e1` to register `x`. When $q(x)$ is true, rule `baz` wants to assign the value of `e1` to register `x`. The circuit for this is shown in Fig. 10.33.

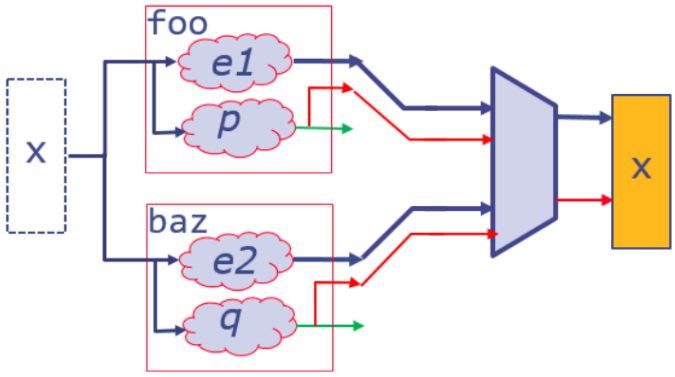


Figure 10.33: Conflicting assignments.

If the synthesis tool can guarantee that $p(x)$ and $q(x)$ are mutually exclusive (for example if one was $x < 32$ and the other was $x \geq 32$), then there is no problem—our requirement that at most one input of the mux should be enabled is met. But what if the synthesis tool cannot guarantee it, either because the conditions $p(x)$ and $q(x)$ are too complex for the tool's analysis, or because they also depend on some other external value unknown to the tool? In this case, the tool must produce circuitry that, in the case when both guards are true, suppresses one of the rules. One way to do this is shown in Fig. 10.34

Here, we have re-established our requirement that at most one input of the mux should be enabled by inverting $p(x)$ and ANDing it with $q(x)$ before feeding it to the mux. When $p(x)$ and $q(x)$ are both true, this extra circuitry will ensure that only `foo`'s data gets through. This brings up another bit of terminology. If we think of the output of $q(x)$ as the `CAN_FIRE` signal of rule `baz`, then the output of the AND gate is called the `WILL_FIRE` signal of rule `baz`.

Note that, in the above circuit, we have *prioritized* rule `foo` over `baz`. We could equally well have implemented the opposite priority by, instead, inverting $q(x)$ and ANDing it with

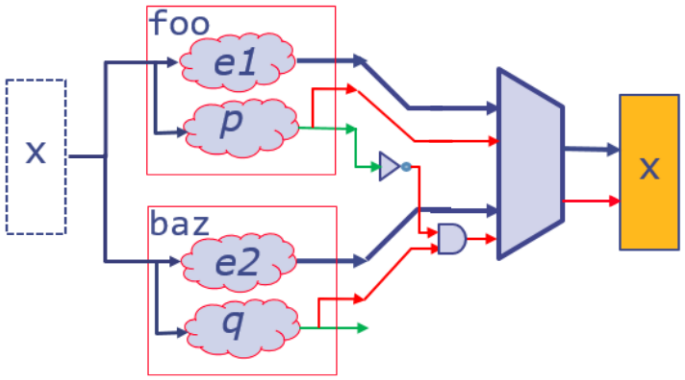


Figure 10.34: A resolution of conflicting assignments.

$p(x)$. This illustrates the general point that, when multiple rules *conflict* in some way, there often are many possible ways to the resolve the conflict. The different choices can affect the performance of the overall circuit, its area and its power, and so the "right answer" may depend on what is important for this application.

10.10 Example: An Up-Down Counter

An Up-Down counter is one where we can both increment and decrement the count. For example, in an automated car park, we might initialize the counter to 0, increment it each time a car enters and decrement each time a car leaves. The current counter value shows how many cars are currently in the car park, and this could perhaps drive an informational display showing the number of remaining (available) spaces. As usual, let's start with the interface declaration:

```
1  interface UpDownCounter;
2     method ActionValue #(Bit #(8)) up;
3     method ActionValue #(Bit #(8)) down;
4  endmodule
```

Here is a module implementation:

```
1  module mkUpDownCounter (UpDownCounter);
2
3     Reg #(Bit #(8)) ctr <- mkReg (0);
4
5     method ActionValue #(Bit #(8)) up if (ctr < 255);
6        ctr <= ctr+1;
7        return ctr;
8     endmethod
9
10    method ActionValue #(Bit #(8)) down if (ctr > 0);
11       ctr <= ctr-1;
```

```
12        return ctr;
13     endmethod
14  endmodule
```

An environment that uses the counter may look like this (in outline):

```
1   module ...
2
3      UpDownCounter x <- mkUpDownCounter;
4
5      rule rl_producer;    // car entering
6         ... x.up ...
7      endrule
8
9      rule rl_consumer;    // car leaving
10        ... x.down ...
11     endrule
12  endmodule
```

This is illustrated in Fig. 10.35. We must be careful to disallow the rules `producer` and

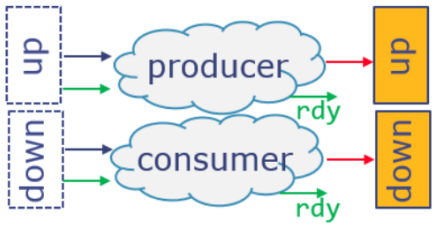


Figure 10.35: Using an Up-Down Counter.

`consumer` from executing concurrently, since that would result in a double-write: they invoke the methods `up` and `down`, respectively, and both methods write to register `ctr`.

Fig. 10.36 shows the internal circuit inside the module. The `up` method is READY when `ctr` is < 255. When invoked (ENABLEd) it updates `ctr` with $+1$ of its old value. The `down` method is READY when `ctr` is > 0. When invoked (ENABLEd) it updates `ctr` with -1 of its old value.

How do we prevent a double-write error? We do this in the environment, as shown in Fig. 10.37. We have inserted an inverter and an AND gate that allows the consumer to assert the `down` method's ENABLE signal only if the producer is not READY. In this circuit, if both producer and consumer are ready, the producer is given priority over the consumer. If they are ready at disjoint times, the extra logic has no effect. This extra circuitry to prevent double-writes will be automatically inserted by BSV's compiler.

10.10.1 Preventing double-writes and preserving atomicity

Consider this code fragment with two rules:

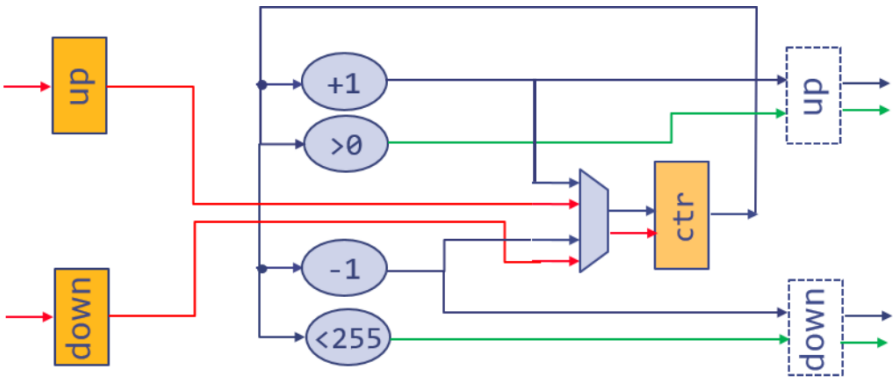


Figure 10.36: Up-Down Counter internal cicuit.

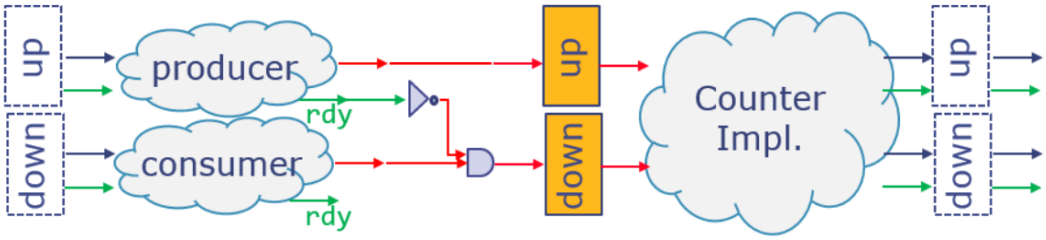


Figure 10.37: Suppressing the possibility of a double-write when using the Up-Down Counter.

```
1  rule ra;
2     x <= e1; y <= e2;
3  endrule
4
5  rule rb;
6     x <= e3; z <= e4;
7  endrule
```

Certainly, the if the rules were allowed to fire concurrently, they would double-write into register `x`. Following the example of the Up-Down Counter, we may think we could fix-up the circuit as shown in Fig.10.38. This indeed fixes the double-write on `x`, but it does not preserve the atomicity of rules. Remember, "atomicity" of a rule means it either performs *all* of its actions (here, register writes) or *none*. When both rules are ready, then this circuit writes the value of `e1` into `x` (because the producer is prioritized), and the value of `e2` into `y`; so far, so good. But the circuit also writes `e4` into `z`. Thus, we've performed one of the two actions in rule `rb`.

The circuit is easy to fix. By the definition of atomicity, if we suppress *any* action in a rule to avoid a double-write, we must suppress *all* actions of that rule at the same time. The fixed-up circuit is shown in Fig.10.39.

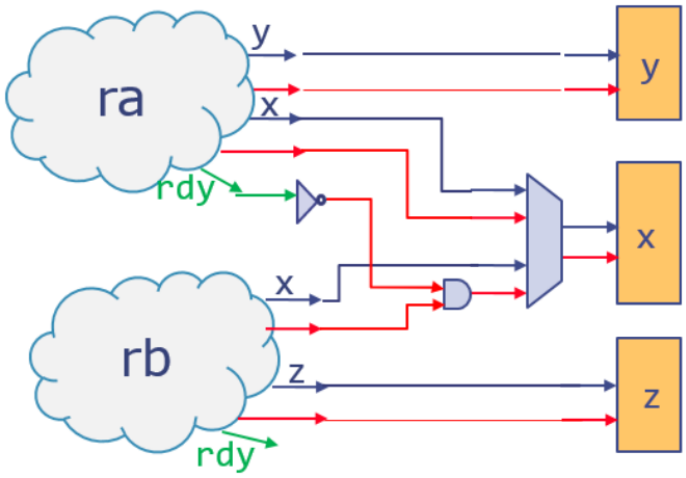


Figure 10.38: Circuit fixing double-write, but not preserving atomicity.

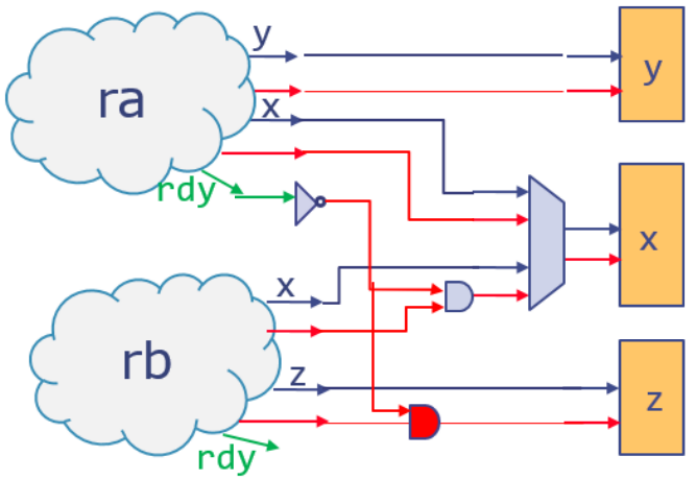


Figure 10.39: Circuit fixing double-write and preserving atomicity.

10.11 Generalizing the circuits that control rule-firing

We now generalize the examples of the last few sections. Note, fortunately all this work is done for us by BSV's compiler, but the purpose here is to develop an intuitive understanding of the kinds of circuits produced. Since these constructions are rather tedious, intricate and case-specific, they would also be error-prone if done manually; this gives us an appreciation for the extra confidence in correctness due to automatic construction by the compiler.

Fig.10.40 shows a sketch of two rules invoking two methods. Recall, we show each

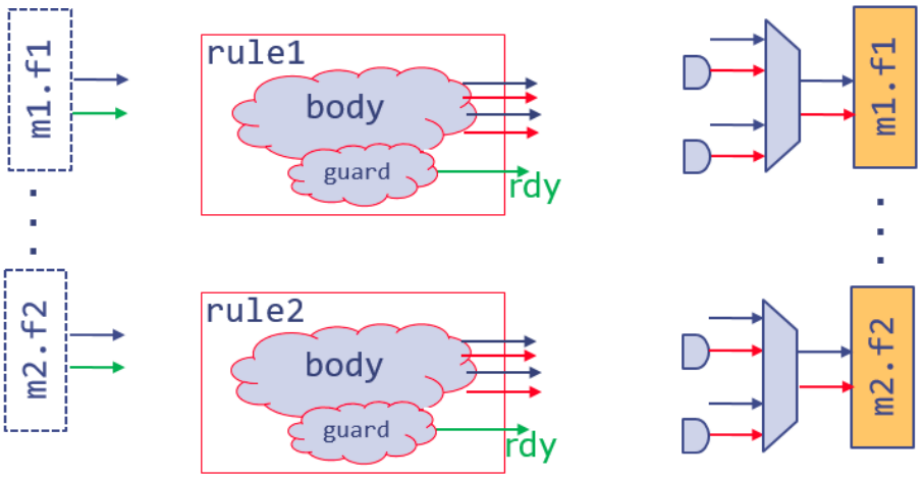


Figure 10.40: Two rules and two methods: the starting point.

method twice—the right-hand side shows input wires of the method going into the method's module and the left-hand side (dotted-line boxes) shows the same methods with output wires of the method coming out of the method's module. As always, READY signals are shown in green and ENABLE signals are shown in red. We have already prepended the methods with muxes to "funnel" in their data and ENABLE signals. The ENABLE signals have been prepended with AND gates in anticipation of suppression, as in the examples of the last section.

Next, in Fig.10.41 we show the "easy" connections—method outputs feeding the body and guard of each rule; data wires from the rule guards to the methods; data and ENABLE wires from the rule bodies to the methods.

In Fig.10.42 we introduce a new circuit which we call a "scheduler". Its inputs are the READY signals from all the rules, also known as the "CAN FIRE" signals. Finally, in Fig.10.43 we connect the output of the scheduler to the AND gates controlling each of the ENABLE signals feeding the muxes into the methods.

The scheduler, seeing all the rule CAN FIRE signals, has knowledge about which rules are concurrently ready. By selectively asserting the WILL FIRE signals, it can ensure that rules are executed atomically, *i.e.,* either all or none of the actions of a rule are performed. We can see that Fig.10.39 is just a special case of this figure, where we have not called out the scheduler as a separately identifiable circuit.

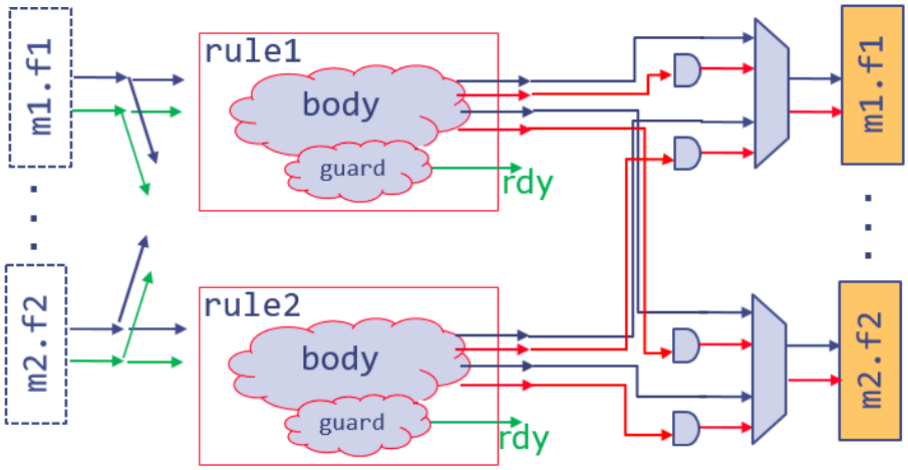


Figure 10.41: The "easy" connections.

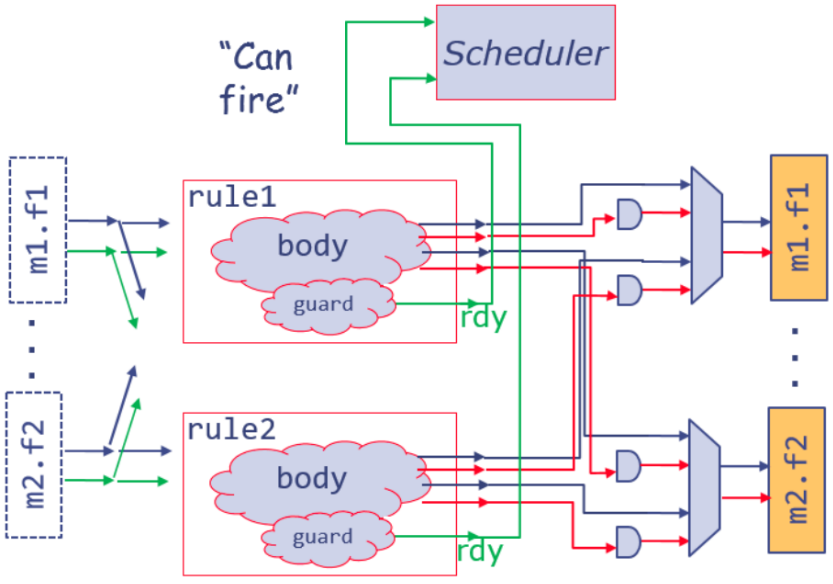


Figure 10.42: Introducing a "scheduler".

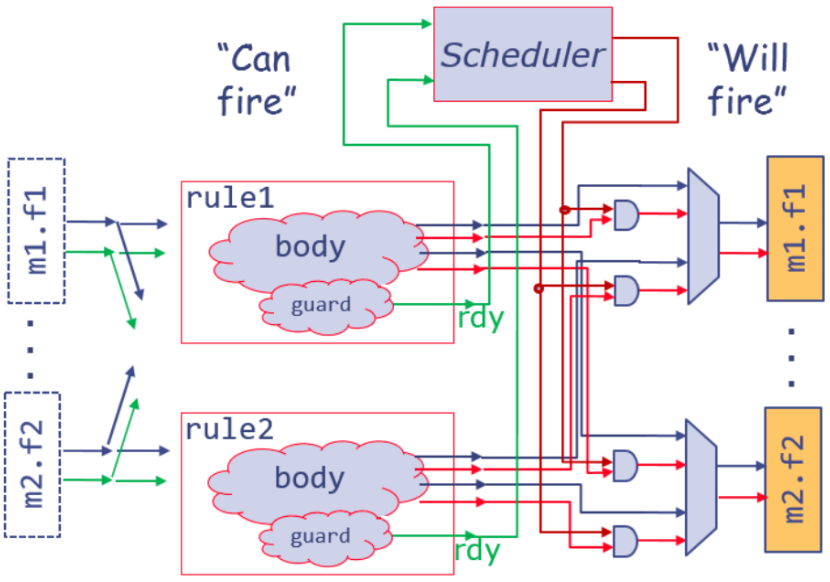


Figure 10.43: "WILL FIRE" signals from the scheduler control actions.

10.11.1 Looking inside the scheduler

The scheduler is a purely combinational circuit. It is typically custom built for each application based on an analysis of the specific rules in the application, their potential for double-writes and their atomicity requirements. All this is done automatically for us by BSV's compiler.

Fig.10.44 shows the simplest of schedulers for two rules which do not have any double-

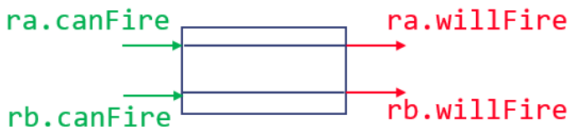


Figure 10.44: The simplest scheduler

write issue. Each CAN FIRE is simply carried through as the corresponding WILL FIRE.

Fig.10.45 shows a scheduler for two rules that do have a double-write issue, and where

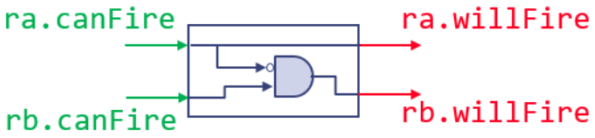


Figure 10.45: Scheduler prioritizing rule ra over rule rb

we have prioritized rule `ra` over rule `rb`. Fig.10.46 shows a scheduler with the opposite prioritization, rule `rb` over rule `ra`. BSV's compiler allows the programmer to supply "scheduling annotations" using which it can be guided to produce either of the above two schedulers.

These last few examples demonstrate that in any application, there may be many possible schedulers that prevent double-writes and preserve atomicity. Different choices of

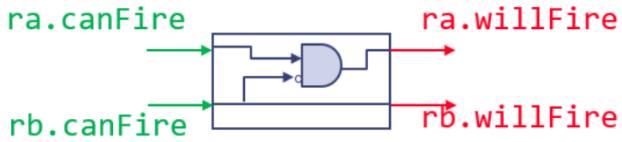


Figure 10.46: Scheduler prioritizing rule rb over rule ra

schedulers may produce circuits with different area, performance and power.

10.12 Summary

We have shown general principles by which one can generate a sequential machine corresponding to any user-written module. Remember, this work is done for us by the BSV compiler, but it is always useful to demystify the workings of compilers, so that we have a good mental model of the circuits obtained.

Sequential machines are connected to each other using atomic rules and methods. Sometimes, we need to prevent the execution of a rule to avoid double-write errors. If we prevent a double-write error from a rule, by atomicity we must also prevent all the rule's actions.

We can design hardware schedulers (which are just combinational circuits) to intervene and prevent the execution of any set of ready rules, whether for double-write error reasons or any other reason. Indeed, we will make use of this facility to enforce some more desirable properties of digital designs.

Exercise 10.1: In Sec. 10.7 we sketched the synthesis of a GCD module. Fill in the details and draw the complete circuit. □

Chapter 11

Increasing concurrency: Bypasses and EHRs

11.1 Introduction

So far we have concentrated only on the *functionality* of our designs ("what does it do?") and not so much on their performance ("how quickly does it do it?"). We now focus on the latter question. We will find that our designs so far lack a certain degree of concurrency (ability of multiple rules to execute in the same clock) thereby limiting performance. The solution is to introduce a standard hardware technique called *bypassing* which is a way to communicate values within a clock cycle (with just wires, and bypassing any intermediate state).

11.2 The problem: inadequate concurrency

11.2.1 Example: The Up-Down Counter

Here, again, is the code for our Up-Down Counter. First, the interface:

```
1 interface UpDownCounter;
2    method ActionValue #(Bit #(8)) up;
3    method ActionValue #(Bit #(8)) down;
4 endmodule
```

and then the module implementation:

```
1 module mkUpDownCounter (UpDownCounter);
2
3    Reg #(Bit #(8)) ctr <- mkReg (0);
4
5    method ActionValue #(Bit #(8)) up if (ctr < 255);
6       ctr <= ctr+1;
```

```
      return ctr;
   endmethod

   method ActionValue #(Bit #(8)) down if (ctr > 0);
      ctr <= ctr-1;
      return ctr;
   endmethod
endmodule
```

Since both the `up` and `down` methods write into register `ctr`, they cannot be invoked concurrently. Thus, in any clock, we can either increment or decrement, but not both. In many applications, this is not good enough.

11.2.2 Example: A pipeline connected with FIFOs

Consider the pipeline in Fig. 11.1, where the three stages transform data using the functions

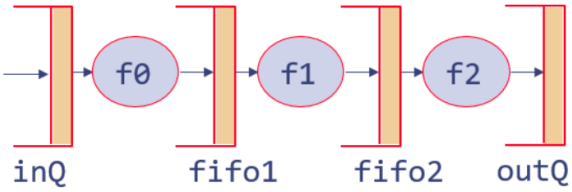


Figure 11.1: A 3-stage pipeline.

f_0, f_1 and f_2, respectively. Here is a code fragment for the pipeline:

```
rule stage1;
   fifo1.enq (f0 (inQ.first));
   inQ.deq;
endrule

rule stage2;
   fifo2.enq (f1 (fifo1.first));
   fifo1.deq;
endrule

rule stage3;
   outQ.enq (f2 (fifo2.first));
   fifo2.deq;
endrule
```

For this to act as a true pipeline, we would like all stages to advance concurrently (in each clock). This implies that, in each clock, we must be allowed to invoke each FIFO's `enq`, `first` and `deq` methods. If not, we only get limited pipelining, where only alternate stages can execute concurrently, and adjacent stages cannot. It would be a pipeline with "bubbles" separating data.

Here is our code for a 1-element FIFO:

```
module mkFifo (Fifo #(1, Bit #(n)));

  Reg #(Bit #(n)) d <- mkRegU;
  Reg #(Bool)     v <- mkReg (False);

   method Action enq (Bit #(n) x) if (! v);
      v <= True; d <= x;
   endmethod

   method Action deq if (v);
      v <= False;
   endmethod

   method Bit #(n) first if (v);
      return d;
   endmethod
endmodule
```

Methods `enq` and `deq` cannot execute concurrently for two reasons: they both write register `v` and they have mutually exclusive guards. Methods `enq` and `first` cannot execute concurrently because they have mutually exclusive guards.

Suppose we try to fix the problem using a 2-element FIFO instead, as sketched in Fig. 11.2, Assume that if there is only one element in the FIFO, it resides in data register

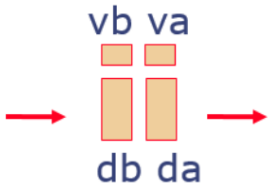


Figure 11.2: Sketch of internal state of a 2-element FIFO.

`da`, and the corresponding "valid" register `va` is true. If there are two elements in the FIFO, the data are in `va` and `vb` (in that order) and both valid registers are true. Initially, `va` and `vb` are set false, since the FIFO is empty.

And `enq` can be done as long as `vb` is false and it writes its value in the b registers. A `deq` can be done as long as `va` is true, and it takes its value from (and updates) the a registers. In this way, `enq` and `deq` do not write to any common register, and there is no double-write issue.

We eventually have to move a freshly enqueued value from the a registers to the b registers. This is done by an extra rule inside the module called "canonicalize". Here is the code:

```
module mkCFFifo (Fifo #(Bit #(n)));
   // ... instantiate da, va, db, vb (not shown)
```

```
3
4     rule canonicalize if (vb && (! va));
5        da <= db;
6        va <= True;
7        vb <= False;
8     endrule
9
10    method Action enq (Bit #(n) x) if (! vb);
11       db <= x;
12       vb <= True;
13    endmethod
14
15    method Action deq if (va);
16       va <= False;
17    endmethod
18
19    method Bit #(n) first if (va);
20       return da;
21    endmethod
22 endmodule
```

Unfortunately, although there is no longer any double-write problem or mutually exclusive guards involving `enq`, `deq` and `first`, we have just pushed the problem elsewhere. The methods still have a double-write issue and mutually exclusive guards with the rule. Thus, even though `enq` and `deq` and `first` can execute concurrently when there is exactly one element in the FIFO already in the a registers, when the rule is moving data from the b registers to the a registers, none of the methods can execute, thereby re-introducing our "bubble".

11.2.3 Summary: limitations of using registers to mediate all communication

If basic register is our only primitive state element, it is the only way for a method or rule to communicate a value to another method or rule. A method or rule writes a value into a register, and a method or rule reads from that register, and that can only happen in the next clock at the earliest. As seen in the above examples, this fundamentally limits concurrency.

Bypassing provides a means by which methods and rules can communicate within the same clock (concurrently). In effect, we are going to expand our set of basic primitives from just ordinary registers to include bypass registers and EHRs. These new primitives are "sufficient", *i.e.,* using them we can express any degree of concurrency that we wish.

11.3 Bypassing

In BSV one thinks of bypassing in terms of reducing the number of cycles it takes to execute two rules or methods that would conflict if they used ordinary registers to communicate. For

example: design a FIFO where, even when the FIFO is full, `deq` and `enq` can be performed concurrently. This requires signalling from the former to the latter method since, when the FIFO is full, an `enq` is only possible (is ready) if a `deq` happens (is enabled).

Another small example: transform the following rules:

```
1 rule ra;
2    x <= y+1;
3 endrule
4
5 rule rb;
6    y <= x+2;
7 endrule
```

such that they execute concurrently, and behave as if rule `ra` happened before `rb`, *i.e.,* `x` gets the old value of `y` +1, and `y` gets the old value of `y` +3. These examples are not possible using only ordinary registers to communicate between rules.

11.3.1 New kinds of primitive registers

Fig. 11.3, shows an ordinary register, viewed as a BSV module with a `Reg #(Bit #(n))`

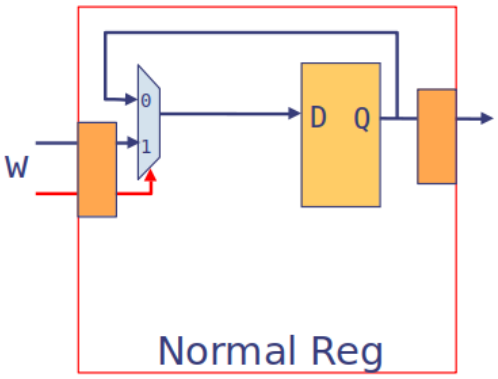


Figure 11.3: An ordinary register, seen as a BSV module with `Reg` interface.

interface, repeated below:

```
1 interface Reg #(Bit #(numeric type n))
2    method Action   _write (Bit #(n) x);
3    method Bit #(n) _read;
4 endinterface
```

The `_write` method is shown on the left and the `_read` method on the right.

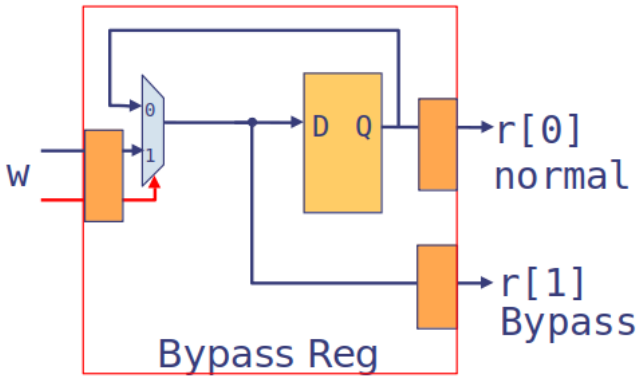


Figure 11.4: A bypass register.

The bypass register primitive module

Fig. 11.4 shows the circuit for a *bypass register*. Its interface is a slight generalization of the ordinary register interface, now having two read methods:

```
1 interface Reg #(Bit #(numeric type n))
2    method Action   _write (Bit #(n) x);
3    method Bit #(n) _read_0;    // normal
4    method Bit #(n) _read_1;    // bypass
5 endinterface
```

Note that the data value written by the `_write` method is directly bypassed out to the `_read_1` method. The `_read_0` method behaves like the `_read` method in an ordinary register. The value returned by `_read_0` is the old value ("before" the `_write`), and the value returned by `_read_1` is the new value ("after" the `_write`). We use the following notation to express this idea formally:

$$\texttt{\_read\_0} < \texttt{\_write}$$
$$\texttt{\_write} < \texttt{\_read\_1}$$

Note that if `_write` is not currenty being invoked, then both `_read_0` and `_read_1` return the same value, the value in the D flip flops. In other words, the new behavior only occurs when we concurrently invoke `_write` and `_read_1`.

The priority register primitive module

Fig. 11.5 shows the circuit for a *priority register*. Its interface is also a slight generalization of the ordinary register interface, this time having two write methods:

```
1 interface Reg #(Bit #(numeric type n))
2    method Action   _write_0 (Bit #(n) x);    // normal
3    method Action   _write_1 (Bit #(n) x);    // priority
4    method Bit #(n) _read;
5 endinterface
```

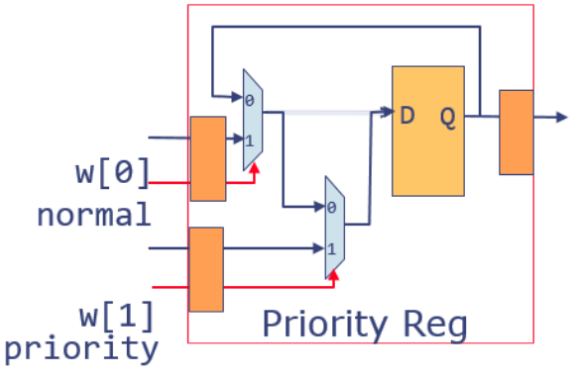


Figure 11.5: A priority register.

This is how we will arbitrate double writes. When `_write_0` and `_write_1` are invoked concurrently, we will essentially discard the value from the former and store the value from the latter, thus "prioritizing" the latter. The `_read` method's behavior is the same as in the ordinary register, with respect to both `_write_0` and `_write_1`. We can describe this formally as follows:

```
_read    < _write_0
_read    < _write_1
_write_0 < _write_1
```

Note that if `_write_0` and `_write_1` are not invoked concurrently, they each behave like the `_write` of an ordinary register.

The EHR primitive module

Our last new primitive module is the *EHR*[1] shown in Fig. 11.6. This is just a combination

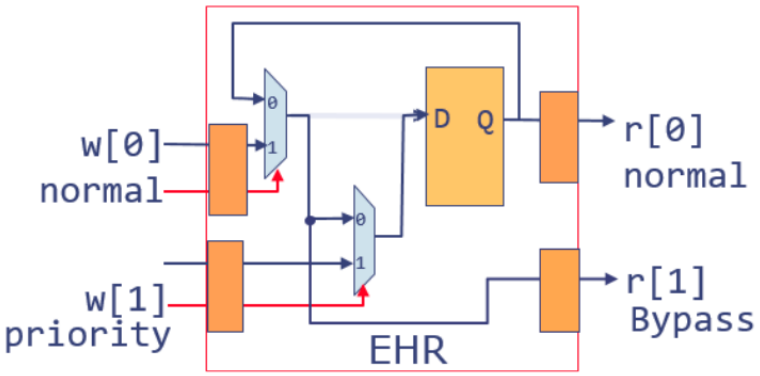


Figure 11.6: An EHR ("Ephemeral History Register".

of the bypass register and the priority register, and this is also reflected in the formalization of the method relationships:

[1]Originally, "Ephemeral History Register" in [1]. Nowadays we mostly use only the abbreviation EHR.

```
_write_0 < _write_1
_write_0 < _read_1
_read_0  < _write_0
_read_0  < _write_1
_read_1  < _write_1
```

Since EHRs subsume bypass and priority registers, we will only use EHRs from now on, just leaving unused methods unused.

Note that the bypassing or prioritizing behaviors only occur during concurrent invocations of methods. When methods are invoked individually (not concurrently), all the `_read_`j and `_write_`j methods behave like the `_read` and `_write` methods of an ordinary register. This is the beauty of EHRs—when analyzing a circuit for functional correctness, we think of one rule at a time, for which an EHR is identical to an ordinary register. We can separately analyze a circuit's performance by studying concurrent invocation of its EHR methods.

Note, while Fig. 11.6 shows an EHR with two read and two write methods, it can be extended to n read methods and m write methods using the same principles, for any n and m. In practice n and m are rarely larger than small single-digit numbers (say, 3 or 4), because each such extra method increases the number and length of combinational paths, thereby limiting the clock speed at which we can run the circuit.

11.4 Revisiting the Up-Down Counter, using EHRs for greater concurrency

It is very easy to change our code for the Up-Down counter in Sec. 11.2.1 to use an EHR instead of an ordinary register:

```
1  module mkUpDownCounter (UpDownCounter);
2
3     EHR #(Bit #(8)) ctr <- mkEHR (0);
4
5     method ActionValue #(Bit #(8)) up if (ctr [0] < 255);
6        ctr [0] <= ctr [0] + 1;
7        return ctr [0];
8     endmethod
9
10    method ActionValue #(Bit #(8)) down if (ctr [1] > 0);
11       ctr [1] <= ctr [1] - 1;
12       return ctr [1];
13    endmethod
14 endmodule
```

We have replaced the original instantiation of an ordinary registr:

```
Reg #(Bit #(8)) ctr <- mkReg (0);
```

by an instantiation of an EHR:

```
EHR #(Bit #(8)) ctr <- mkEHR (0);
```

In the `up` method, we have replaced each `ctr` method invocation by `ctr[0]`. In the `down` method, we have replaced each `ctr` method invocation by `ctr[1]`.

When `up` and `down` are called singly (not concurrently), the behavior is exactly as before, since an EHR behaves like an ordinary register when there is no concurrency. In fact, for analyzing functional correctness, we can simply treat `EHR` and `mkEHR` as a synonyms for `Reg` and `mkReg`, and erase all the square-bracket indexes. If we do that, we get the original code.

When `up` and `down` are called concurrently, the value written by `up` into `ctr[0]` is bypassed into the read of `ctr[1]` in the right-hand side of the register assignment in the `down` method, *i.e.,* it sees the incremented value. This value is decremented and written into `ctr[1]`. Thus, there is no net change in the value of `ctr` $(+1 - 1)$.

Note that the guard for `down` uses `ctr[1]`. In the original code (or in the non-concurrent case), if `ctr` had the value 0, the guard would prevent `down` from being invoked. However, here, if `up` is being invoked concurrently, then the guard for `down` would see the incremented value, 1, and will thus allow `deq` to be invoked concurrently. However, it is still like the original code in that `up` cannot be invoked if the counter value is 255.

Note, we could just as well have flipped the indexes, using `ctr[1]` in `up` and `ctr[0]` in `down`. This only affects the conceptual ordering of the two methods, a consequence of the conceptual ordering on the EHR methods. In the code shown above, `up` executes "before" `down`, thus allowing concurrent invocations even when the counter value is 0. If we flip the indexes, then conceptually `down` executes "before" `up`. In this case, a similar analysis will show that we can do concurrent invocations when the counter value is 255, and that `down` cannot execute when the counter value is 0.

Fig. 11.7 shows the circuit generated for the Up-Down counter with EHRs. There is

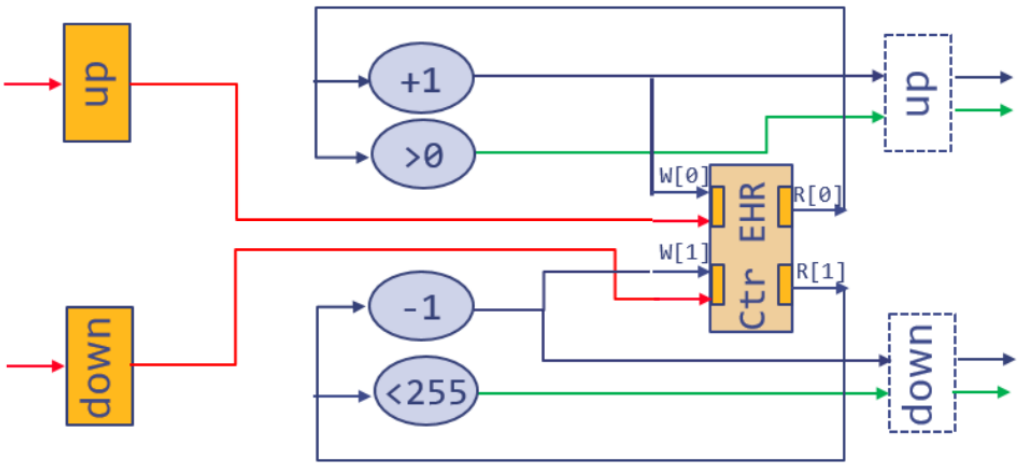


Figure 11.7: Circuit for Up-Down Counter with EHRs.

no longer any double-write problem, but it has introduced a longer combinational path. Specifically, the READY signal for the `down` method depends on the value from `ctr[1]`, which depends on the value being written into `ctr[0]`, which is the output of the +1 logic. Similarly, in the concurrent case, the `ctr[0]` output goes through both the +1 and the −1 logic before being written back to the D flip flops. This tradeoff comes with the territory—trying to do more in one clock (more concurrency) is likely to lengthen combinational paths.

Fig. 11.8 sketches the circuitry in the environment, *i.e.*, the surrounding circuit that

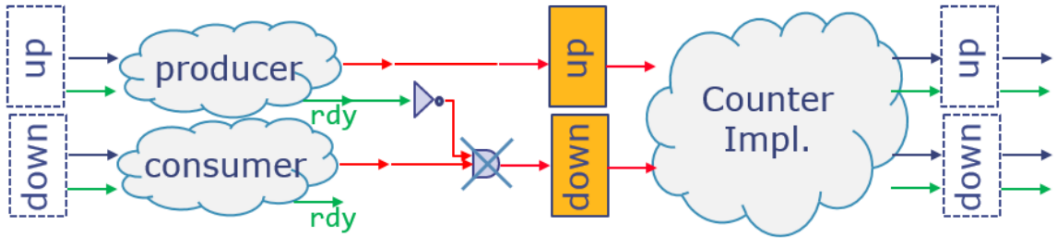


Figure 11.8: Environment ciruit for Up-Down Counter with EHRs.

instantiates and uses the Up-Down counter. This is the same as Fig. 10.37, except that we have crossed out the AND gate suppressing the consumer from invoking the `down` method concurrently with the producer invoking the `up` method; we no longer need it, since this implementation of the Up-Down counter allows concurrent invocation.

This is a useful general observation: note that the original code for the Up-Down counter using ordinary registers, and the new code using EHRs, have exactly the same BSV interface declaration. However, they differ in the concurrency that they allow on the `up` and `down` methods. In general, the concurrency allowed on methods of an interface depend on the particular module implementation behind that interface.

11.5 The Conflict Matrix: a formal characterization of method ordering

The "method ordering" ideas of the previous section can be formalized in a *Conflict Matrix*, which is a square matrix with both rows and columns labeled with the methods and rules of a module. An entry in the matrix at row r_a and column r_b specifies the ordering constraint between those two methods:

$r_a < r_b$	r_a and r_b can be executed concurrently; the net effect is as if r_a is executed before r_b. (We can also write this as $r_b > r_a$.)
r_a CF r_b	r_a and r_b can be executed concurrently; the net effect is the same as $r_a < r_b$ and $r_a > r_b$. (CF = "conflict free")
r_a C r_b	r_a and r_b cannot be executed concurrently; it would cause a double-write error, or the resulting effect is neither $r_a < r_b$ nor $r_a > r_b$. (C = "conflict")
r_a ME r_b	The guards of r_a and r_b are mutually exclusive and thus r_a and r_b can never be READY concurrently. (ME = "mutually exclusive")

Here is the conflict matrix for an ordinary register:

	`_read`	`_write`
`_read`	CF	<
`_write`	>	C

Method `_read` is conflict-free with itself (top left); it can be invoked concurrently in two rules/methods, and it imposes no ordering on those rules. Method `_write` conflicts with itself (bottom right); it can be never be invoked concurrently in two rules/methods (double-write). Method `read_` can be invoked concurrently with `_write` and it is as if they go in that order; this can be seen in the top right entry (the bottom left entry is the same; the matrix is symmetric around the diagonal).

Here is the conflict matrix for an EHR with two read and write methods:

	`_read_0`	`_write_0`	`_read_1`	`_write_1`
`_read_0`	CF	<	CF	<
`_write_0`	>	C	<	<
`_read_1`	CF	<	CF	<
`_write_1`	>	>	>	C

11.5.1 The Conflict Matrix for a module written in BSV

The conflict matrices for ordinary registers, EHRs, and indeed any primitive modules, are just "given" (although, if we look at the underlying D flip flops and logic from which they are built, we can see the rationale for the matrix entries). Since a primitive is a black box, the conflict matrix only involves its interface methods (there are no visible rules).

If we imagine a module instantiation hierarchy, the leaves ("height 0") of the hierarchy are primitive modules. Moving up one level, we get instances of modules written in BSV that use primitives (that instantiate primitives and invoke their methods). We can generate correct circuits for such a "height 1" module using the method relationships in the conflict matrices of the primitives it uses.

Further, we can systematically derive the conflict matrix for such a "height 1" module, based on the conflict matrices of the primitives it uses and the source code for the module. This conflict matrix relates all the methods and rules of this module.

This principle can be applied at the next level up and, recursively, at every level in the module hierarchy all the way up to the top-level module. To derive the conflict matrix for a "height j" module, we only need the conflict matrices for the methods of "height $j-1$ modules that it instantiates (we can ignore the rows and columns for the rules internal to the module). Indeed, a compiler for BSV does this systematically when compiling a BSV program.

For example, here is the conflict matrix for our Up-Down Counter implemented with ordinary registers:

	`up`	`down`
`up`	C	C
`down`	C	C

We saw earlier that for this implementation of the module, `up` and `down` cannot be invoked concurrently; thus the off-diagonal entries in the matrix are C (conflict). The diagonal entries are also C because neither `up` nor `down` can be invoked concurrently with itself (each case would result in a double-write).

Here is the conflict matrix for our Up-Down Counter implemented with EHRs:

	up	down
up	C	<
down	>	C

We saw earlier that for this implementation of the module, `up` and `down` can be invoked concurrently; thus the off-diagonal entries are <.

11.6 Designing concurrent FIFOs using EHRs

In Sec. 11.2.2 we introduced the need for FIFOs where we can invoke `enq` and `deq` concurrently. We will now design three such FIFOs, armed with the ability to use EHRs. All these FIFOs have the same FIFO interface, and differ only in the concurrency supported by their methods.

11.6.1 Pipeline FIFOs

A *Pipeline FIFO* allows concurrent `enq` and `deq` even if it is full. The method ordering is `deq`<`enq`; *i.e.,* the newly dequeued data conceptually vacates a space in the FIFO that is immediately occupied by newly enqueued data.

Here is the code for a one-element Pipeline FIFO, modifying the implementation in Sec. 11.2.2 by replacing the ordinary register for `v` by an EHR:

```
1  module mkPipelineFIFO (FIFO #(Bit #(n)));
2     Reg #(Bit#(n)) d <- mkRegU;
3     EHR #(Bool)    v <- mkEHR (False);
4
5     method Action enq (Bit #(n) x) if (! v[1]);
6        v[1] <= True;
7        d <= x;
8     endmethod
9
10    method Action deq if (v[0]);
11       v[0] <= False;
12    endmethod
13
14    method Bit #(n) first if (v[0]);
15       return d;
16    endmethod
17 endmodule
```

We invite the reader to analyze the code and verify that if `enq`, `first` and `deq` are invoked singly (not concurrently within the same clock), the module behaves exactly like like the original in Sec. 11.2.2 that was implemented with ordinary registers. Remember, without concurrency, we can treat EHRs just like ordinary registers and erase all the method indexes; then, this code becomes identical to the original.

When the FIFO is full, the "valid" EHR `v` is true, and the data is in `d`. The `first` and `deq` methods are READY (`v[0]`) is true). If we invoke these methods, the `deq` method writes false into `v[0]`.

Therefore, the `enq` method is ready—its guard is true since `v[1]` returns the just-assigned value of `v[0]`. Thus, we can invoke `enq` concurrently (in the same clock). The net change in the FIFO is that the newly enqueued data lands in `d`, and `v` remains true.

Here is the conflict matrix for a Pipeline FIFO:

	enq	deq	first
enq	C	>	>
deq	<	C	>
first	<	<	CF

11.6.2 Bypass FIFOs

A *Bypass FIFO* allows concurrent `enq` and `deq` even if it is empty. The method ordering is `enq`<`deq`; *i.e.,* the newly enqueued data is immediately bypassed through as the newly dequeued data.

Here is the code for a one-element Bypass FIFO, also modifying the implementation in Sec. 11.2.2 by replacing the ordinary register for `v` by an EHR:

```
1  module mkBypassFIFO (FIFO #(Bit #(n)));
2     EHR #(Bit#(n)) d <- mkEHR (?);    // reset value not relevant
3     EHR #(Bool)    v <- mkEHR (False);
4
5     method Action enq (Bit #(n) x) if (! v[0]);
6        v[0] <= True;
7        d[0] <= x;
8     endmethod
9
10    method Action deq if (v[1]);
11       v[1] <= False;
12    endmethod
13
14    method Bit #(n) first if (v[1]);
15       return d[1];
16    endmethod
17 endmodule
```

Again, we invite the reader to and verify that if `enq`, `first` and `deq` are invoked singly (not concurrently within the same clock), the module behaves exactly like like the original in Sec. 11.2.2 that was implemented with ordinary registers.

When the FIFO is empty, the "valid" EHR `v` is false. The `enq` method is READY (`v[0]` is false). If we invoke this method, it writes true into `v[0]`, and writes the newly enqueued data into `d[0]`.

Therefore, the `first` and `deq` methods are ready—their guards are true since `v[1]` returns the just-assigned value of `v[0]`. Thus, we can invoke these methods concurrently with `enq` (in the same clock). The value returned by `first`, read from `d[1]` is the value just enqueued by the `enq` method. The net change in the FIFO is that and `v` remains false.

Here is the conflict matrix for a Bypass FIFO:

	enq	deq	first
enq	C	<	<
deq	>	C	>
first	>	<	CF

11.6.3 Conflict-Free FIFOs

A *Conflict-Free FIFO* allows concurrent `enq` and `deq`, with no ordering constraints, as long as the FIFO is neither empty nor full. The effect of `enq` is not visible to `deq`, and vice versa, in the current clock.

Here is the code for a two-element Conflict FIFO, modifying the 2-element FIFO in Sec. 11.2.2, using EHRs instead of ordinary registers:

```
1  module mkCFFIFO (FIFO #(Bit #(n)));
2     EHR #(Bit #(n)) da <- mkEhr (?);
3     EHR #(Bool)     va <- mkEhr (False);
4     EHR #(Bit #(n)) db <- mkEhr (?);
5     EHR #(Bool)     vb <- mkEhr (False);
6
7     rule canonicalize (vb[1] && !va[1]);
8        da[1] <= db[1];
9        va[1] <= True;
10       vb[1] <= False;
11    endrule
12
13    method Action enq (Bit #(n) x) if (! vb[0]);
14       db[0] <= x;
15       vb[0] <= True;
16    endmethod
17
18    method Action deq if (va[0]);
19       va[0] <= False;
20    endmethod
21
22    method Bit #(n) first if (va[0]);
23       return da[0];
24    endmethod
25 endmodule
```

In the original code, rule `canonicalize` had a double-write conflict with all the methods. In this code, it uses the newly-written values from the methods concurrently (in the same

clock). The methods write their values into the [0] methods of the EHRs, and the rule reads these values immediately by using the [1] methods of the EHRs, and produces the final value in the registers by using the [1] methods. Note, if the FIFO is empty, `first` and `deq` are not READY, and if the FIFO is full, `enq` is not READY, as expected. In any cycle where the FIFO contains 1 element (neither full nor empty), `enq` and `deq` can be invoked concurrently.

Here is the conflict matrix for this Conflict-Free FIFO:

	`enq`	`deq`	`first`	`canonicalize`
`enq`	C	CF	CF	<
`deq`	CF	C	>	<
`first`	CF	<	CF	<
`canonicalize`	>	>	>	C

11.6.4 Revisiting our 3-stage pipeline

In Sec. 11.2.2, we showed a 3-stage pipeline (Fig. 11.1) and discussed how, to act as a true pipeline that is able to advance all stages together, we need FIFOs that allow concurrent invocation of `enq`, `deq` and `first` methods.

If we replace the `mkFIFO` instantiations with either `mkPipelineFIFO` or `mkCFFifo`, we achieve the desired concurrency. The latter (`mkCFFifo`) doubles the number of state elements compared to the former (`mkPipelineFIFO`), since each FIFO is a 2-element FIFO. On the other hand, using `mkPipelineFIFO` introduces a combinational path all the way through the pipeline:

- a `deq` on the far right affects whether stage 3 can invoke `enq` on its right; which affects whether stage3 can invoke `deq` on its left;
- That affects whether stage 2 can invoke `enq` on its right; which affects whether stage2 can invoke `deq` on its left;
- That affects whether stage 1 can invoke `enq` on its right; which affects whether stage1 can invoke `deq` on its left;
- That effects whether the `enq` on the far left can be invoked.

Note, Bypass FIFOs would not be appropriate here. If all FIFOs are empty, that would allow an input value x on the far left to pass straight through all three stages, delivering $f_2(f_1(f_0(x)))$ in a single clock to the far right which is interesting, but that is not what we want in pipeline behavior.

11.7 Example: A Register File and a Bypassed Register File

A register file is an array of registers, selected by an *index*. The interface for a so-called "two-read-port" register file containing 32 registers, each holding 32-bit data, is:

```
1 interface RFile;
2    method Action    wr  (Bit #(5) rindx, Bit#(32) data);
3    method Bit #(32) rd1 (Bit #(5) rindx);
```

```
4     method Bit #(32) rd2 (Bit #(5) rindx);
5  endmodule
```

The `wr` method takes a register index and a data value and writes that value into the corresponding register. The `rd1` and `rd2` methods have the same functionality—they take a register index and return the contents of the corresponding register. The reason for two read ports and one write is that this is typical in a modern CPU—most machine code instructions read two registers, perform an operation (such as ADD) and write the result back to a register.

Here is the code for a module implementing this, using ordinary registers:

```
1  module mkRFile (RFile);
2     Vector #(32, Reg #(Bit #(32)) rfile <- replicateM (mkReg (0));
3
4     method Action    wr  (Bit #(5) rindx, Bit#(32) data);
5        rfile[rindx] <= data;
6     endmethod
7
8     method Bit #(32) rd1 (Bit #(5) rindx) = rfile[rindx];
9
10    method Bit #(32) rd2 (Bit #(5) rindx) = rfile[rindx];
11 endmodule
```

The first line inside the module replicates the familiar `mkReg(0)` to instantiate 32 registers, and collects the resulting register interfaces in a vector. The `wr`, `rd1` and `rd2` method implementations are straightforward, just reading and writing the register selected by the given index argument. Based on the conflict matrix of ordinary registers, we can infer that:

$$\texttt{rd1} < \texttt{wr}$$
$$\texttt{rd2} < \texttt{wr}$$

If `rd1` (similarly `rd2`) and `wr` are invoked simultaneously with the same index argument, `rd1` returns the old value that is already in that register, and `wr` replaces it with a new value.

Now let us see if we can design a different register file module where

$$\texttt{wr} < \texttt{rd1}$$
$$\texttt{wr} < \texttt{rd2}$$

If `rd1` (similarly `rd2`) and `wr` are invoked simultaneously with the same index argument, `rd1` should returns the new value that is concurrently being written by `wr`.

Here is the code for this new register file implementation. The required modifications are almost trivial, using EHRs:

```
module mkBypassRFile (RFile);
   Vector #(32, EHR #(Bit #(32)) rfile <- replicateM (mkEHR (0));

   method Action    wr  (Bit #(5) rindx, Bit#(32) data);
      rfile [rindx] [0] <= data;
   endmethod

   method Bit #(32) rd1 (Bit #(5) rindx) = rfile[rindx] [1];

   method Bit #(32) rd2 (Bit #(5) rindx) = rfile[rindx] [1];
endmodule
```

11.8 Summary

EHRs can be used to design a variety of modules to reduce conflicts between methods and thereby increase concurrency. Examples include FIFOs (Pipeline, Bypass and Conflict Free), Register Files (bypassing), "scoreboards" for very high-performance CPUs that execute instructions in out-of-order or data-flow manner, high-performance memory systems, and so on.

The Conflict Matrix of a module's methods are sufficient to know what concurrency we can obtain with it; there is no need to peek into the module to see how it might use EHRs or other tricks; in this regard, it has the same status as primitive modules.

However, a caution is that EHRs can increase the length of combinational paths and thus affect achievable clock frequencies. A useful design methodology is first to design using ordinary registers and assure that our design is functionally correct, and then selectively to replace modules with refined implementations that use EHRs instead in order to increase performance by increasing concurrency.

Chapter 12

Serializability of Concurrent Execution of Rules

12.1 Introduction

Whether in software or in hardware, managing access to shared state from concurrent activities is a central problem. In software, concurrent activities occur as multiple virtual or physical threads in a CPU or tightly-coupled cluster of CPUs ("multi-threading", "multi-processing"), and from multiple computers communicating over networks ("distributed systems"). In software, people usually learn programming without any concurrency at all using sequential programming languages like C, C++, Python and so on. After gaining expertise, a few people may move on to concurrent programs (threads, processes, distributed systems). Sequential programs are often the "training wheels" from which one may later graduate to concurrent programming.

In hardware, it is next to impossible to avoid concurrency, from day one. Concurrency is often the very *raison d'Ã¤tre* of hardware—concurrency allows tasks to be completed much more quickly.

In both software and hardware systems, the single most important concept known in Computer Science for managing access to shared state from concurrent activities is the *atomic transaction*. An atomic transaction is a collection of updates to shared state performed by a process (concurrent activity), where the process can assume the absence of any interference from any other process. Further, atomicity semantics imply that either all of the updates or none are performed; the process does not leave the shared state in some questionable intermediate state.

A simple example from real life is transferring X dollars from a checking account to a savings account. The updates involve reading the checking balance, subtracting X and writing it back, and reading the savings balance, adding X and writing it back. Non-interference means that, during this transaction, the process need not concern itself with other processes trying to read and write these balances at the same time (interfering additions or subtractions may interleave badly and write bogus values back into the balances). All-or-none semantics implies that either the entire transfer succeeds, or none of it; we cannot mistakenly update one balance but not the other.

Atomicity allows us to analyze a system for correctness. For example, atomicity allows easily to reason that a collection of transfers between a checking and savings account will leave the total balance unchanged.

12.2 Linearizability and Serializability in BSV

In BSV, rules have the semantics of atomic transactions. This means that no other rule's accesses to shared state (reads and writes of share registers) appear interleaved with this rule's accesses. In other words, every other rule appears to execute either before or after this rule. This is the definition of *linearizability* of a rule.

One way to achieve atomicity is literally to execute one rule at a time since, in this case, the possibility of interference is moot. However, if we literally executed one rule per clock in hardware, in most applications we would not get any reasonable performance. We would like to execute multiple rules per clock ("concurrently").

We could say that concurrent execution of rules or methods is allowed as long as double-writes do not occur. Still, it is useful to preserve the one-rule-at-a-time view as the *logical semantics* of rules, because it simplifies the analysis of circuits to verify correct behavior. Thus, we impose the constraint of *serializability* on the concurrent execution of rules:

> *Serializability* means that a concurrent execution of rules must match some serial execution of rules, a.k.a. one-rule-at-a-time execution of rules.

In other words, rules can be executed concurrently provided the execution is *as if they were executed in some sequential order*, *i.e.,* functionally equivalent to a sequential execution.

Serializability is a very well-known and familiar concept to software engineers working with concurrent processes or distributed systems. Hardware engineers are less familiar with the concepts and vocabulary, even though the considerations are equally applicable (if not more so) in the hardware domain.

Our use of the term "linearizable" is just a local consideration, a property that a rule neither observes nor provides any interference with any other rule. Our use of the term "serialiable" is more global property, that execution of a collection of rules occurs as if they occurred in some serial order.

12.3 One-rule-at-a-time semantics of BSV

The one-rule-at-a-time *logical view* of BSV semantics is very simple:

> Repeatedly:
>
> - Select any rule that is ready to execute
> - Compute the state updates
> - Make the state updates

Any legal behavior of a BSV program can be explained by observing state updates obtained by applying one rule at a time.

However, as mentioned earlier, for performance we execute multiple rules concurrently, making sure that we do so in a manner consistent with one-rule-at-a-time semantics.

12.4 Concurrent execution of rules

To illustrate these concepts, let us look at the following three examples.

Example 1

```
rule ra;
    x <= x + 1;
endrule

rule rb;
    y <= y + 2;
endrule
```

Example 2:

```
rule ra;
    x <= y + 1;
endrule

rule rb;
    y <= x + 2;
endrule
```

Example 3:

```
rule ra;
    x <= y + 1;
endrule

rule rb;
    y <= y + 2;
endrule
```

Note that in each case, the rules write to different registers, and so there is no possibility of a double-write. But can we then, therefore, just execute them willy-nilly in the same clock?

Before we analyze serializability, let us remind ourselves about "old" and "new" values in a register assignment, by annotating the reads and writes with a time superscript:

Example 1

```
rule ra;
    x^{t+1} <= x^t + 1;
endrule

rule rb;
    y^{t+1} <= y^t + 2;
endrule
```

Example 2:

```
rule ra;
    x^{t+1} <= y^t + 1;
endrule

rule rb;
    y^{t+1} <= x^t + 2;
endrule
```

Example 3:

```
rule ra;
    x^{t+1} <= y^t + 1;
endrule

rule rb;
    y^{t+1} <= y^t + 2;
endrule
```

In any clock cycle, a register-read always reads the old value in the register (that was sampled at a previous clock edge), and a register-write produces a new value that is not visible until after the next clock edge. We designate these with superscripts ^{t} and $^{t+1}$, respectively. For any `x`, if there is no assigment defining $\texttt{x}^{t+1}$, then $\texttt{x}^{t+1}$ is the same as $\texttt{x}^{t}$.

To analyze serializability of concurrent execution, let us hand-execute the three examples in three different ways:

(1) "Blindly concurrent": ra and rb in the same clock
(2) Serially, with ra first, then rb (*e.g.,* in two successive clocks)
(3) Serially, with rb first, then ra (*e.g.,* in two successive clocks)

Assume that `x` and `y` are both initialized to contain 0. The table below shows the final values of `x` and `y` for the three examples nd for each of the above three "schedules":

	Example 1	Example 2	Example 3
Blindly concurrent	$x = 1$, $y = 2$	$x = 1$, $y = 2$	$x = 1$, $y = 2$
ra < rb	$x = 1$, $y = 2$	$x = 1$, $y = 3$	$x = 1$, $y = 2$
rb < ra	$x = 1$, $y = 2$	$x = 3$, $y = 2$	$x = 3$, $y = 2$

Figure 12.1: Outcomes under 3 schedules.

The "blindly concurrent" schedule is ok for example 1 because it is equivalent some serial schedule. In fact, it is equivalent to all possible serial schedules, and for this reason we say that the rules are *Conflict Free* (or *CF* for short).

The "blindly concurrent" schedule is not ok for example 2 because it is not equivalent to *any* serial schedule (the outcomes don't match the outcomes of any possible serial schedule). For this reason, we say that the rules *Conflict* (or *C* for short).

The "blindly concurrent" schedule is ok for example 3 because it is equivalent to some serial schedule, in particular the ra<rb schedule. It's outcome does not match that of rb<ra, but as long as it matches some serial schedule it's ok.

The problem with blind concurrency in Example 2 is that the accesses *interleave* in an order that is different from the order in which they occur in any serial schedule, producing an outcome unrealizable in any serial schedule. Bad interleavings violate atomicity.

Serializability allows to reason about correctness of a design. Returning to our bank-balance example, we might assert the following property:

> *In any collection of transfers, the net balance in the two accounts remains the same.*

We can analyze each transfer rule individually, and determine that it has this property. Then, automatically, we can infer that for any collection of these rules, they have this property, since it must be equivalent to a sequential execution of these rules. If we didn't have serializability, we would have to reason about what happens if reads and writes from different transfer rules interfere with one another, and that reasoning would be much more complex.

12.5 Deriving the Conflict Matrix for a module's interface

In Sec. 11.5 we introduced the concept of a "Conflict Matrix" which formalizes the relationship between the methods of a module. Recall that the Conflict Matrix is a square matrix whose rows and columns are labelled by the methods. Thus an entry CM[gi,gj] in the conflict matrix CM indicates the relationship between method gi and gj of the module—whether they conflict, are conflict-free, or should be ordered by $<$ or $>$. The matrix is symmetric around the diagonal: if CM[gi,gj] is conflict or conflict-free, we see the same entry in CM[gj,gi]. If CM[gi,gj] is $<$ (respectively $>$), then CM[gj,gi] has the opposite ordering $>$ (respectively $<$).

The four possible relationships are naturally viewed as a "lattice", shown in Fig. 12.2. This ordering permits us to take intersections of ordering information. For example:

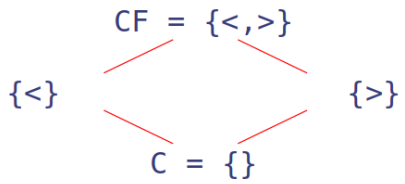


Figure 12.2: Lattice of rule/method relationships.

$$\{>\} \cap \{<, >\} = \{>\}$$
$$\{>\} \cap \{<\} \quad = \{\}$$

Intersections are also known as "meets" or "greatest-lower-bounds" (glbs) in lattices.

We derive conflict matrices in a bottom-up manner in the module hierarchy. The leaves of a module hierarchy (height 0) are all primitive modules (registers, EHRs, and so on), for each of which we are given a "built-in" conflict matrix. At height 1, we have module instances that internally only instantiate primitives. Given the conflict matrix of the primitives, we can deduce the conflict matrix of a height 1 module. From this point on, we can treat it just like a primitive, and now do the same for an enclosing height 2 module. In this way, recursively, we can deduce the conflict matrix for every module in the module hierarchy.

Let g1 and g2 be two methods in the interface of a module. Let us say that "mcalls(g1)" is the set of methods called inside the definition of method g1, *i.e.,* in its guard and in its body:

mcalls (g1) = g11, g12, ..., g1n
mcalls (g2) = g21, g22, ..., g1m

Then, CM[g1,g2] can be calculated by computing a the following function involving all pairwise relationships between a method in mcalls(g1) and a method in mcalls(g2), *i.e.,*

CM[g1,g2] = conflict (g11,g21) ∩ conflict (g11,g22) ∩ ... conflict (g1n,g2m) ∩
conflict (g12,g21) ∩ conflict (g12,g22) ∩ ... conflict (g12,g2m) ∩
...
conflict (g1n,g21) ∩ conflict (g1n,g22) ∩ ... conflict (g1n,g2m) ∩

where

conflict (x,y) = CM[x,y] if x and y are methods on the same sub-module
CF otherwise

Intuitively, CM[g1,g2] is the at or below (in the lattice) every ordering between any method in g1 and any method in g2.

The conflict matrix for two rules is computed in exactly the same way, by examining the methods called in the rules. For our example in Sec. 12.4, the methods they invoke are `x._read`, `x._write`, `y._read`, and `y._write`. Performing the analysis, we get the following conflict matrices:

Example 1		
	ra	rb
ra	C	CF
rb	CF	C

Example 2		
	ra	rb
ra	C	C
rb	C	C

Example 3		
	ra	rb
ra	C	<
rb	>	C

12.6 Example: The Two-Element FIFO

Let us derive the conflict matrix for the 2-element FIFO we introduced in Sec.11.2.2. Fig. 11.2 is repeated here as Fig. 12.3. The code for its `enq` and `deq` methods is given

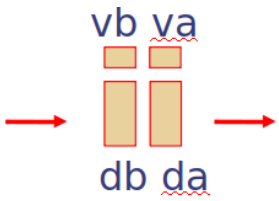


Figure 12.3: State elements in a 2-element FIFO.

below, where we have taken the `canonicalize` rule in the earlier version and incorporated its actions into the methods here:

```
module mkCFFifo (Fifo #(Bit #(n)));
   // ... instantiate da, va, db, vb (not shown)
   ...
   method Action enq (Bit #(n) x) if (! vb);
      if (va) begin
         db <= x;
         vb <= True;
      end
      else begin
         da <= x;
         va <= True;
      end
   endmethod

   method Action deq if (va);
      if (vb) begin
         da <= db;
         vb <= false;
      end
      else
         va <= False;
   endmethod
   ...
endmodule
```

We can derive a conservative conflict matrix for these methods by collecting all the methods called (ignoring the fact that some of them are called conditionally):

mcalls (enq) = { vb.r, va.r, db.w, vb.w, da.w, va.w }
mcalls (deq) = { va.r, vb.r, da.w, db.r, vb.w, va.w }

where we have abbreviated `._read` and `._write` as .r and .w for brevity.| From this, we can compute the conflict matrix entry:

We can derive a conservative CM by ignoring the conditionals mcalls(enq) = vb.r, va.r, db.w, vb.w, da.w, va.w mcalls(deq) = va.r, vb.r, da.w, db.r, vb.w, va.w

CM[enq,deq] =
CM[vb.r,va.r]∩CM[vb.r,vb.r]∩CM[vb.r,da.w]∩CM[vb.r,db.r]∩CM[vb.r,vb.w]∩CM[vb.r,va.w]
∩CM[va.r,va.r]∩CM[va.r,vb.r]∩CM[va.r,da.w]∩CM[va.r,db.r]∩CM[va.r,vb.w]∩CM[va.r,va.w]
∩CM[db.w,va.r]∩CM[db.w,vb.r]∩CM[db.w,da.w]∩CM[db.w,db.r]∩CM[db.w,vb.w]∩CM[db.w,va.w]
∩CM[vb.w,va.r]∩CM[vb.w,vb.r]∩CM[vb.w,da.w]∩CM[vb.w,db.r]∩CM[vb.w,vb.w]∩CM[vb.w,va.w]
∩CM[da.w,va.r]∩CM[da.w,vb.r]∩CM[da.w,da.w]∩CM[da.w,db.r]∩CM[da.w,vb.w]∩CM[da.w,va.w]
∩CM[va.w,va.r]∩CM[va.w,vb.r]∩CM[va.w,da.w]∩CM[va.w,db.r]∩CM[va.w,vb.w]∩CM[va.w,va.w]

= CF ∩ {<} ∩ CF ∩ {<} ∩ {>} ∩ {>} ∩ C ∩ C ∩ {>} ∩ C
= C

12.7 Concurrent execution of multiple rules

Generalizing from two rules to more rules, in order for rules r1, r2, ... and rk to execute concurrently, there must be an ordering r1<r2<...<rk such that executing the rules in that sequential order produces the same result as concurrent execution. The pairwise conflict matrices must be consistent with this orderin.

It is important to recognize that the < relation is not transitive, that is r1<r2 and r2<r3 does not imply r1<r3; the latter condition must be checked explicitly to validate the ordering r1<r2<r3.

12.8 Using the Conflict Matrix to Enforce Serializability

Suppose we are given a preferred rule ordering based on some other considerations of functionality or performance. We can use the conflict matrix to produce a "correct" execution of rules in that order. In each clock, we consider rules for execution in the given order. At the j'th rule, we can use the conflict matrix to check if this rule would conflict with any of the rules already executed earlier in clock and, if so, we skip the rule (it may be executable in some subsequent clock).

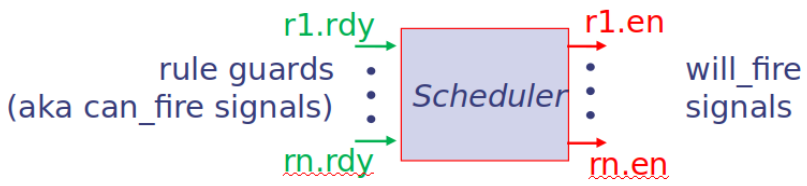


Figure 12.4: The rule scheduler

12.8.1 The rule scheduler

Fig.12.4 shows a *scheduler* circuit created by the BSV compiler for a BSV design. The guards r1.rdy ... rn.rdy of the rules in the design may be true simultaneously, and some pairs of rules may conflict with each other, which will be apparent in the conflict matrix for each pair of rules. The BSV compiler constructs a combinational scheduler circuit given a linear ordering on the rules, such that:

- Rule ri.en is true if ri.rdy is true, and
- For any rule rj before ri in the linear order, rj does not conflict with rj if rj.rdy is true.

12.9 Summary

The one-rule-at-a-time semantics of rules is a conceptual, logical view, which simplifies reasoning about the correctness of BSV designs. However, for good performance, we must execute many rules concurrently within each clock. The job of the compiler is to produce scheduling circuits that enable such concurrency, but in a controlled manner such that, for any collection of rules that execute in a clock, it is *as if* they were executed in a sequential order (has the same final state as they would have had were they executed in that order). The scheduler circuits effectively suppress a rule's execution in a clock if that would encounter a conflict with a rule already executed in that clock. The conflict matrix between each pair of rules and methods is a key tool used by the compiler in analyzing rules and producing a good scheduler circuit. With experience, BSV programmers also think about conflict matrices and use them to analyze performance for insight into how performance can be improved.

Exercise 12.1: Consider the pipeline in Fig.12.5. Can any pipeline stages fire concurrently

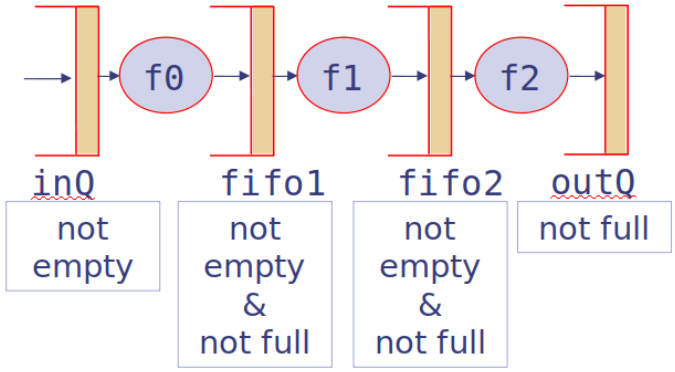


Figure 12.5: A pipeline

if the FIFOs do not permit concurrent `enq` and `deq`? □

Bibliography

[1] D. L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proc. MEMOCODE'04*, June 2004.