6

COMPOSITIONAL VERIFICATION

Composition is notation of distortion of what composers think they have heard before. Masterpieces are marvelous misquotations.

—Ned Rorem - The Paris Diary of Ned Rorem Braziller, 1966

Summary

This chapter presents a case study for the verification of a complex timed system. Most of the contents was already published in [PCSP02].

The system consists of an n-stage pulse-driven IPCMOS pipeline and the verification is carried out for any value of n greater than 0. Each stage is described by a circuit at transistor level and delay information is provided for each transistor. The correctness of the circuit strongly depends on the timed behavior of its components and the environment in which the circuit operates. The combination of the level of detail at which the circuit is described, together with its dependency on timing parameters, makes it attractive for verification.

To verify the system, three techniques have been combined: (1) the relative timing-based verification approach described in Chapter 4, (2) *assume-guarantee* reasoning to verify untimed *abstractions* of timed components, and (3) mathematical induction to verify pipelines of any length.

In first place, the IPCMOS architecture is described at high level for better comprehension of its overall behavior. Also, the different modules that form the architecture are described at low level in order to identify critical parts of the circuitry. High-level techniques for verification are introduced which are required in order to tackle the complexity of the verification. Then, the overall strategy for the verification of IPCMOS pipelines is developed. Finally, details on the verification of the circuitry that implements a single stage are provided.

6.1 Introduction

Chip performance, power consumption, noise reduction and clock synchronization are becoming critical challenges as microprocessor performance moves into GHz regimes. *Interlocked pipelined CMOS* (IPCMOS) circuits [SRC+00], provide an asynchronous clocking technique that can help to tackle these challenges. Measured results on an experimental chip that implemented a 64-bit floating point multiplier using this architecture, demonstrated robust operation at $3.3GHz$ under typical conditions and $4.5GHz$ under best-case conditions in a $0.18\mu m$ $1.5V$ CMOS technology. The circuit showed also robust operation with large variations in power supply voltage, operating temperature, threshold voltage, and transistor channel length.

The general concepts of interlocking, pipelining and asynchronous self-timing are not new and have been proposed in a variety of forms since [Sei80] and [Sut89]. However, the techniques used in those approaches are too slow, specially for blocks which receive data from many separate sources. In contrast, the performance achieved by IPCMOS circuits is due to their pulse-based communication mechanism. The protocol decouples the communication of a block with its predecessors and its successors, thus achieving a high degree of concurrency.

Although a single IPCMOS block that operates in a pipeline can be implemented using only 32 transistors, the concurrency achieved in the overall system leads to the state explosion problem even for a few interlocked blocks.

The correct operation of the system highly depends on its timing parameters, hence the complexity of the analysis is drastically affected by the time dimension. High-level techniques for verification have been typically used to handle the complexity of a system. For example, abstractions [Mel88] tackle the complexity by hiding those implementation details that are irrelevant for the verification of a given property.

In this chapter, the verification a complex timed system such as an IPCMOS pipelines is carried out by combining three techniques:

1. The relative timing-based iterative verification approach presented in Chapter 4 is used as the basic verification engine to perform all the required correctness proofs.

2. The assume-guarantee paradigm [Pnu84] is used to perform a hierarchical verification on large systems by means of abstractions.

3. Finally, mathematical induction is used to prove the correctness of infinite-state systems, such as an n-stage IPCMOS pipeline for $n > 0$.

The rest of the chapter is organized as follows. Section 6.2 presents the details of the IPCMOS architecture. Transistor-level descriptions of the circuits that implement the interlocked modules are provided. Section 6.3 introduces some background on techniques

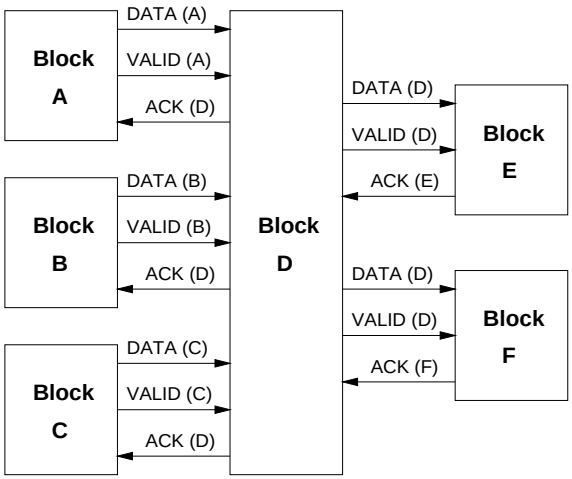


Figure 6.1 General block-level interlocking scheme.

used for compositional verification, such as abstraction, assume-guarantee and induction. The correctness of IPCMOS pipelines regardless of their length is demonstrated in Section 6.4. The proof includes the verification of the circuitry that implements a single stage. Details of this step are provided in Section 6.5.

6.2 The IPCMOS architecture

Figure 6.1 depicts the general block-level interlocking scheme of the IPCMOS architecture. In the figure, block D is interlocked with blocks A, B, C, E and F. In the forward direction, dedicated VALID signals emulate the worst-case performance path through each driving block (A, B and C), and thus determine when data can be latched within block D. In the reverse direction, ACK signals indicate when the data has been received by the subsequent blocks (E and F) and that new data may be processed within block D. In this interlocked approach, local clocks are generated only when there is an operation to perform.

6.2.1 IPCMOS pipelines

A single IPCMOS control block is relatively small and can be used to build meshes, pipelines and other scalable architectures. A linear version of the IPCMOS architecture is depicted in Figure 6.2. The system implements a pipeline composed of IPCMOS control blocks and latches that isolate the logic between stages. When input data for a stage is signaled to be available (VALID signal), the IPCMOS control block generates a local clock signal to latch the data (CLKE signal). Data receipt is confirmed to the sender and as soon as the data is processed (ACK signal). This eliminates the need of global clock distribution

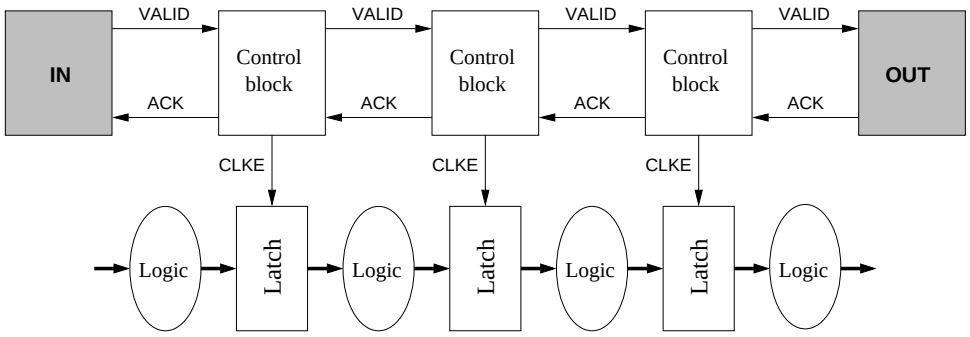


Figure 6.2 Linear IPCMOS architecture. Each stage is composed of a control block, a logic to handle data and a latch that isolates the stage.

and at the same time contributes to the power consumption reduction by clocking data only when it is available at the inputs.

The IPCMOS block communicates with other blocks via request signals (VALID), acknowledgment signals (ACK) and produces a local data clock signal (CLKE). VALID indicates data availability to the receiver(s), while ACK acknowledges to the sender(s) that data has been received. Generally IPCMOS blocks can be fed multiple ACK and VALID signals to enable safely processing data from multiple sources and feeding the result to multiple destinations. In a pipeline, only one VALID signal and only one ACK signal is sent/received by each stage.

IPCMOS circuits are *pulse-driven* or *edge sensitive*. Their operation is illustrated by means of the 2-stage pipeline in Figure 6.3 (a), where the structure of the control blocks is detailed. The figure also shows a waveform that illustrates how two data items propagate through the pipeline. Initially the pipeline is empty: all VALID and CLKE signals are high, whereas all CLKR and ACK signals are low. As soon as negative pulses are received at the VALID input of a stage, a positive pulse is generated at the ACK to acknowledge the data receipt. Input data is clocked by a negative pulse on the CLKE signal, produced by the *strobe* module. After some delay designed to match the worst case computation time of the logic attached to the stage, a negative pulse is generated by the *valid* module at the VALID output line indicating the data availability to the receiver. From this point on, the block waits for positive pulses to be received from the data consumer at the ACK input. The pulses are recorded by the *reset* module. When the acknowledgment is received, a positive pulse at CLKR is produced. This indicates the *strobe* and *valid* modules that the stage is ready to acknowledge new incoming data and to pass new processed data to the receiver, respectively. Meanwhile VALID input pulses indicating the new input data availability are also recorded. Hence, new data receipt at every stage is *interlocked*

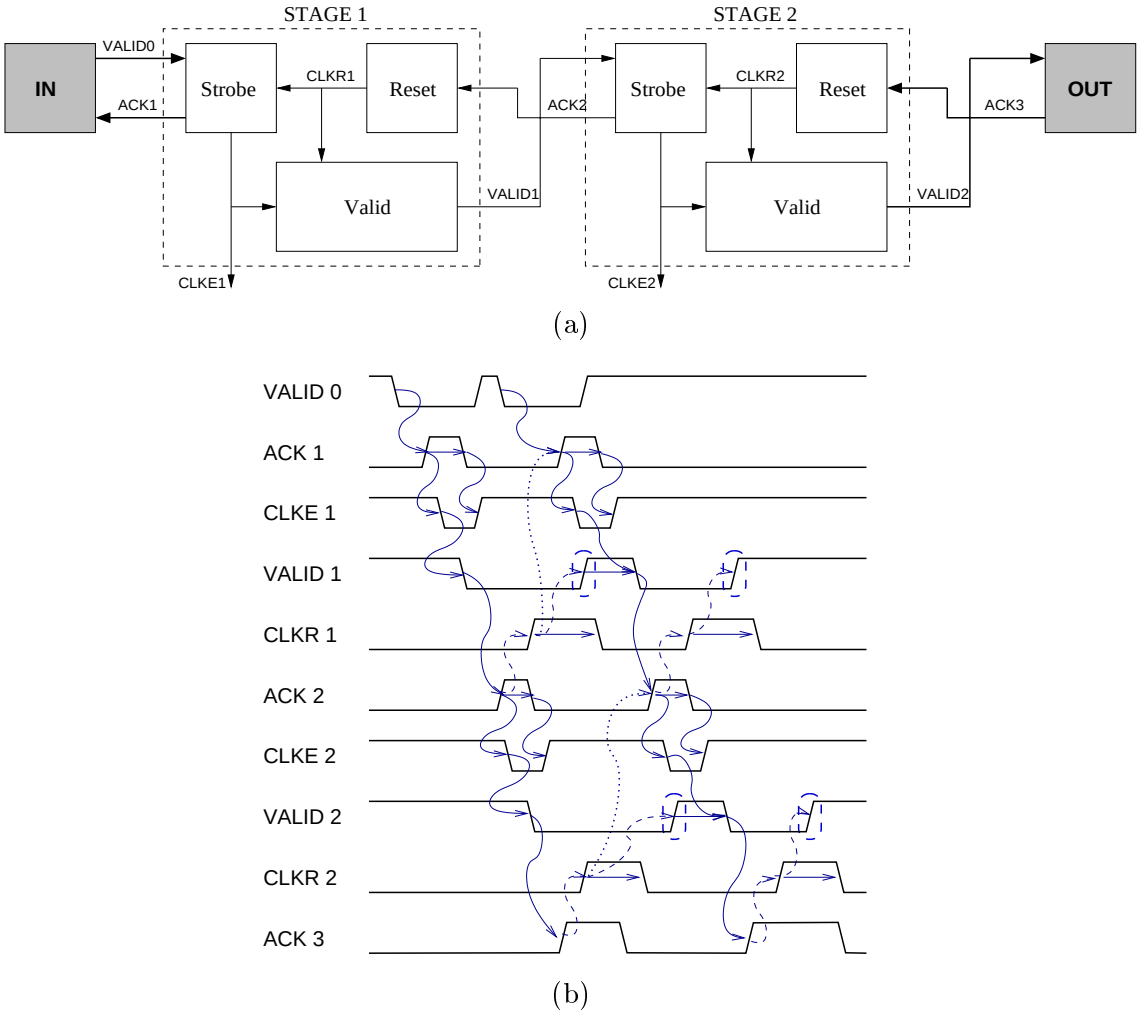


Figure 6.3 Detailed 2-stage IPCMOS pipeline (a) and waveform of its behavior (b). Communication at the extremes of the pipeline is pulse-based (thick lines) but the stages communicate through handshakes.

with the acknowledgment of the data by the following stages. For correct operation, the only restriction the IPCMOS modules pose on the environment is the pulse length of the incoming VALID and ACK signals.

Even though a pulse-driven environment is accepted by each pipeline stage, the internal communication between adjacent stages is performed in a partially handshaked protocol between the positive edges of the pulses (see Figure 6.4). These additional causality relations enable to abstract the behavior of such components when interacting among them, in such a way that internal timing information can be neglected. As we will see later on, this phenomenon considerably simplifies the verification of the pipeline.

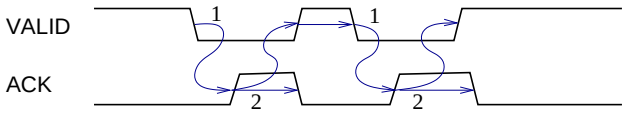


Figure 6.4 Two-phase handshake mechanism.

The dashed boxes in the waveform of Figure 6.3 (b) show the signal edges affected by the handshaking mechanism. The dashed arrows show the restricting causal dependencies that must not exist in the environment (*IN*, *OUT*) but take place in the IPCMOS stages. In the diagram we also show that all stages in a sequence cannot be filled with data at the same time, but "bubbles" (empty stages) are needed to propagate data in one direction and the acknowledgment in the other. The causal dependencies demonstrating this fact are highlighted in the diagram by means of dotted arrows.

The following sections provide details on the circuit implementations of each IPCMOS block, and the behavior of the environment modules *IN* and *OUT*.

6.2.2 Strobe circuit

The general structure of the *strobe* circuit is depicted in Figure 6.5 (a). The *strobe switch* circuit is shown in detail in the left of the figure. Figures 6.5 (b) and (c) illustrate the behavior of the *switch* and the *strobe* circuits by means of waveforms. Notice that when building an IPCMOS pipeline, the *strobe* circuit contains only one switch block. Thus, the waveform in Figure 6.5 (c) shows a single Vint signal.

The *strobe* circuit is responsible for accumulating the negative pulses from the VALID input lines. It is also responsible for generating the clock strobe CLKE that latches the input data as soon as it is available from all sources (all VALID input pulses are received). The clock strobe cannot be issued unless all the data receivers have acknowledged the previous processed data as indicated by CLKR.

Every *strobe switch* module is responsible for storing a single negative pulse from the corresponding VALID input line and for lowering the Vint signal afterwards.

The strobe circuit weak transistor pulls up the X signal as soon as all Vint signals are low (all VALID pulses are received), and the CLKR positive pulse has discharged Rint indicating the acknowledgment receipt from all the receivers. Finally, the switches are reset by the positive pulse sent through the ACK output line.

The p-type transistor that charges X is weak with respect to the n-type transistors that discharge it. Thus, in case Vint_n is the last signal to arrive, charging X does not produce a short-circuit. Signal Rint in also important since is prevents X to be discharged in case the acknowledge from the previous cycle has not been received yet.

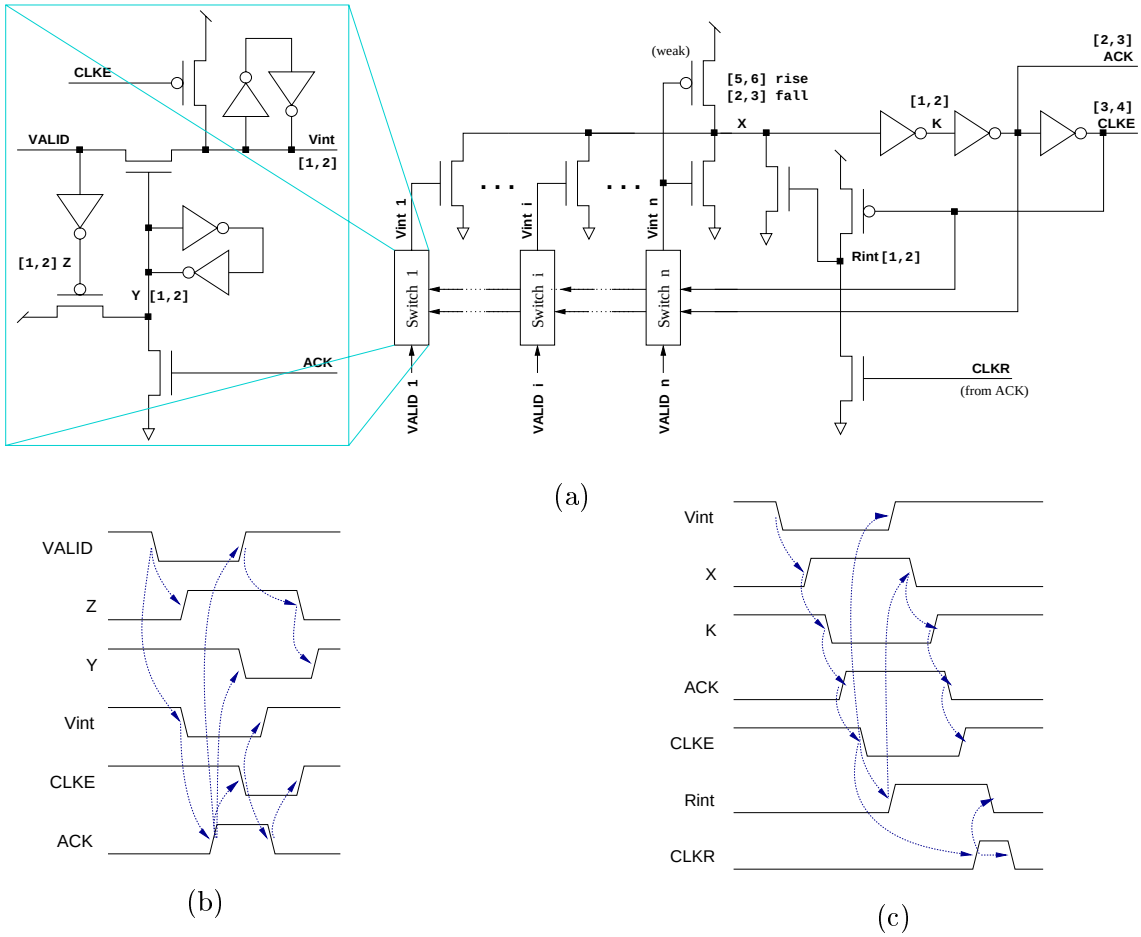


Figure 6.5 (a) The *strobe* circuit in detail with the *strobe switch* highlighted in the left. Waveforms describing the behavior of (b) the *switch* and (c) the *strobe* circuits.

No specific technology library is used, hence the delay bounds for each stack of transistors is set to be in the range of `[1,2]` delay units. Other appropriate delay ranges are used in case of weak transistors or fan-out considerations (see Figure 6.5 (a)). Thus, the delay of charging node X in the *strobe* circuit is set to be in the range `[5,6]`, whereas the discharge delay is set to be in the range `[2,3]`. Also, signals ACK and CLKE have delays in the range `[2,3]` and `[3,4]` respectively, due to fan-out considerations.

6.2.3 Reset circuit

The general structure of the *reset* circuit is depicted in Figure 6.6 (a). The *reset switch* circuit is shown in detail in the left of the figure. Figures 6.6 (b) and (c) illustrate the behavior of the *switch* and the *reset* circuits by means of waveforms. Notice that when

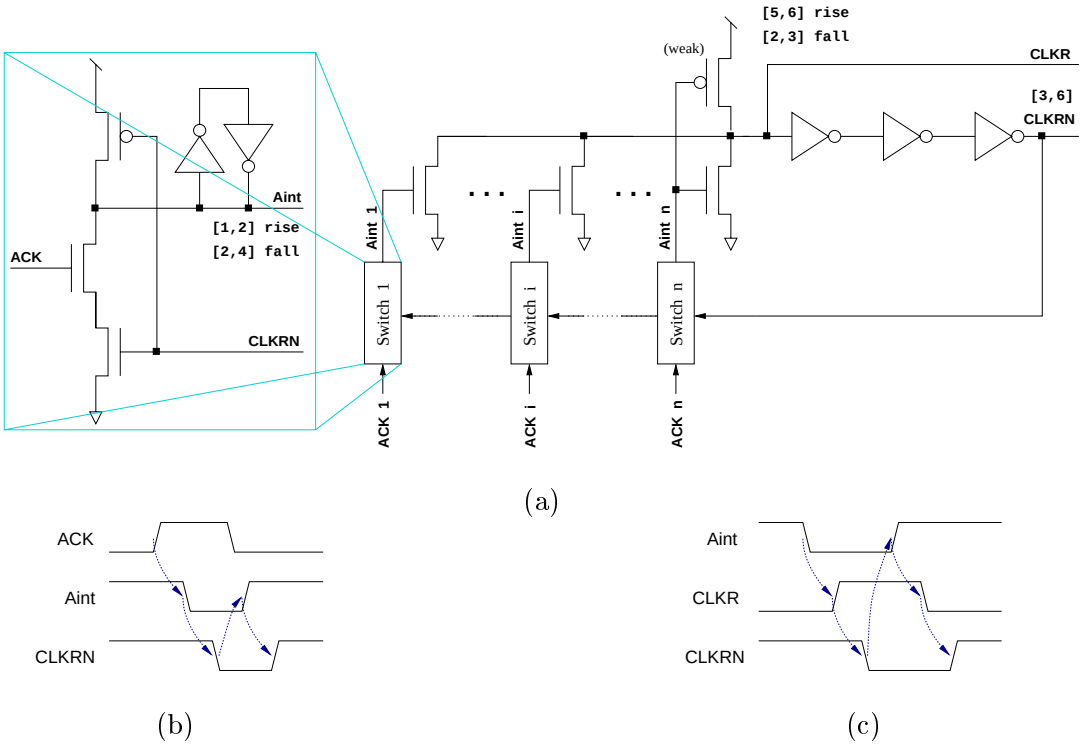


Figure 6.6 (a) The *reset* circuit in detail with the *reset switch* highlighted in the left. Waveforms describing the behavior of (b) the *switch* and (c) the *reset* circuits.

building an IPCMOS pipeline, the *reset* circuit contains only one switch block. Thus, the waveform in Figure 6.6 (c) shows a single Aint signal.

The switch circuit is aimed at detecting positive pulses of the input ACK lines coming from the successor blocks. Having received pulses from all the ACK lines it produces a positive pulse on CLKR for resetting the *valid* and the *strobe* circuits.

The detailed behavior of the *reset* circuit is as follows. Upon the receipt of a pulse from all ACK lines, all the Aint signals will be low, what allows for CLKR to be charged. Similarly to the *strobe* circuit, the p-type transistor that charges CLKR is weak with respect to the n-type transistors that discharge it. Thus, in case Aint$_n$ is the last signal to arrive, charging CLKR does not produce a short-circuit. Finally, after rising CLKR, CLKRN resets the Aint signals, which in turn discharge CLKR.

Similarly to the *strobe* circuit, the delay ranges of the *reset* circuit are set taking into account the weakness of the transistors (for signal CLKR) and fan-out considerations (for signal CLKRN). Signal Aint in the *switch* has a double fall delay corresponding to the two stacked n-type transistors. The delay intervals are annotated in Figure 6.6 (a).

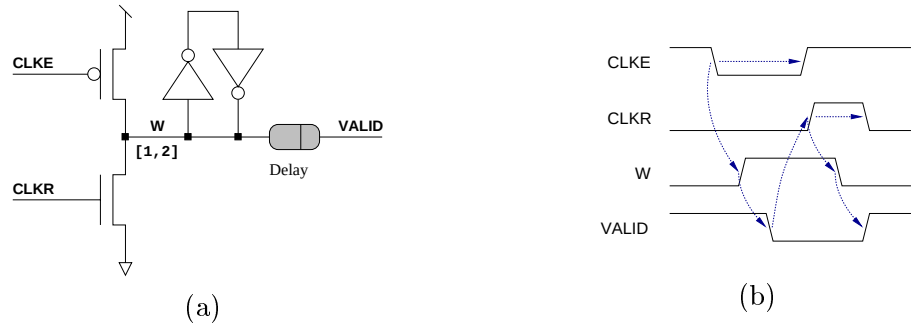


Figure 6.7 (a) The *valid* circuit and (b) waveform describing its behavior.

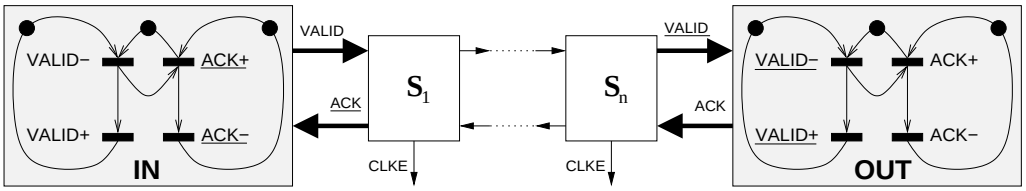


Figure 6.8 STGs modeling the pulse-based behavior of the *IN* and *OUT* modules.

6.2.4 Valid circuit

The details of the *valid* circuit are depicted in Figure 6.7.

The *valid* circuit incorporates a delay element matching the worst-case delay of the logic associated with the stage, so that it lowers the VALID output signal only after that delay is elapsed since latching the new data by the CLKE low pulse. On the other hand, the CLKR signal is fed to the *valid* circuit to ensure that the VALID output signal is raised again only after the data transfer has been acknowledged by the successor stage. Until that happens the VALID output signal is kept low. On the other hand, the delay associated to the changes in signal W is in the range `[1,2]`.

6.2.5 The environment modules

Figure 6.8 depicts the communication protocol implemented by the *IN* (left) and the *OUT* (right) parts of the environment, in the form of Signal Transition Graphs. The underlined transitions represent the behavior of the circuit stage signals. The *IN* and *OUT* modules operate in a pulse-based manner. This fact is highlighted by the thick lines that connect the environment modules with the pipeline.

The OUT module acknowledges the data available at the output of the pipeline by rising the ACK line. Once this happens, both the last stage of the pipeline and the OUT module reset the respective VALID and ACK lines independently. A restriction must be imposed to OUT to avoid early resetting of ACK. That is, if ACK− arrives too fast after ACK+, the falling edge of ACK may not be properly recorded by the *reset switch* circuit of the last stage of the pipeline. Therefore a minimum width is required to the positive pulse of ACK.

The IN module notifies new data availability at the input of the first stage of the pipeline by lowering the VALID line. The stage acknowledges the incoming data by rising the ACK line. The reset of both lines is carried out independently and no new data can be issued by IN until the first stage has acknowledged the previous data portion. Also a restriction must be imposed to IN to avoid early resetting of VALID. That is, if VALID+ arrives too fast after VALID−, the falling edge of VALID may not be properly recorded by the *strobe switch* circuit of the first stage of the pipeline. Therefore a minimum width is required to the negative pulse of VALID.

The aforementioned conditions on the width of the pulses produced by the environment modules, will be checked later on during the verification process.

6.2.6 About complexity

The complexity of an IPCMOS control block depends on the number of data suppliers and data receivers attached to it. Looking at the different circuits that implement the control block, the number of transistors that form it can be computed as: $N_{transistors} = 21 + 7 * N_{inputs} + 4 * N_{outputs}$. Thus, a single stage of a linear pipeline contains 32 transistors. Notice that data keepers are not taken into account because they do not introduce additional complexity to our model.

Although the number of transistors of a single stage is small, when a stage is composed to build a pipeline that interacts with the IN and OUT modules, the state explosion problem appears rather soon. Thus, for example, a simple 1-stage pipeline has 212040 untimed states, a 2-stage pipeline has about $2.5E+9$ untimed states, a 3-stage pipeline has about $3.0E+13$ untimed states, etc. In fact the state space of a 4-stage pipeline could not be computed in a $866MHz$ Pentium-III computer with 1GB of RAM running Linux, due to memory overflow. To our knowledge, no verification approach for timed systems can handle such amount of untimed states unless higher-level techniques for verification are used.

6.3 Compositional verification

In order to overcome the complexity of the verification, symbolic representations of the state space of the systems are commonly used. However, such representations do not

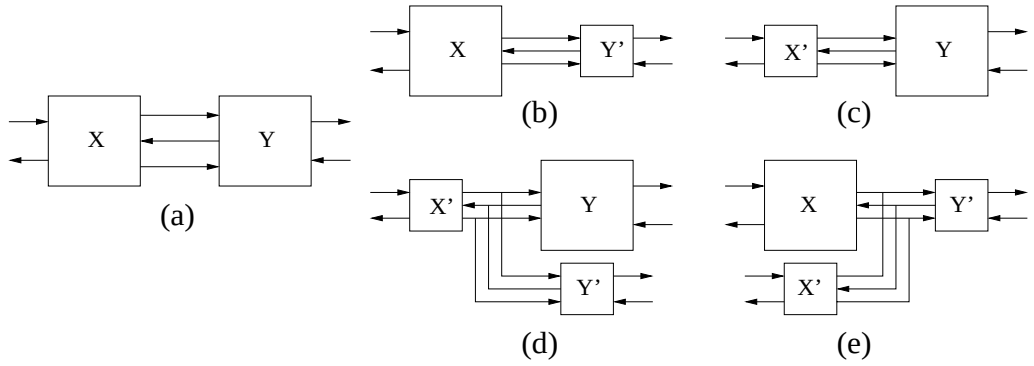


Figure 6.9 Assume-guarantee verification using abstractions.

suffice when complex realistic systems are analyzed. Several techniques have been proposed which, combined with symbolic representations, allow the analysis of larger systems. Those of interest to our purposes are: *abstraction*, *assume-guarantee* reasoning and *induction*.

Abstraction

The *abstraction* mechanism [Mel88, CGL92, TAKB96, DGG97] allows to reduce the size of the state space by removing details irrelevant for proving a given property. When performing abstraction, information about the exact behavior of the system is lost, therefore the truth of some properties cannot be determined by looking only at the abstracted system.

It is important that the verification methodology does not lead to false positive results. That is, if a given property holds in the abstraction a mechanism is required to show that the property actually holds in the non-abstracted system. A verification methodology with this property is said to be conservative. Notice that, in general, nothing can be concluded about what happens in the actual system if the property does not hold in the abstraction. This, will depend on the level of detail hidden by the abstraction procedure.

Assume-guarantee

The *assume-guarantee* paradigm [Pnu84, CLM89, Lon93, GL94, HRS98] exploits the modular structure of systems. It reasons about the correctness of the overall system by checking only the local properties of the components. Unfortunately, a component is designed to operate only in the environment of that system, thus it is unlikely to satisfy any interesting property unless analyzed together with such environment. However, such analysis would lead again to the state explosion problem.

The *assume-guarantee* technique tackles this intimate relation among the components of a system. In the example of Figure 6.9 (a), since the behavior of X depends on the behavior of Y, the correctness of X can be proved only if certain *assumptions* are satisfied

by Y. Then, one must *guarantee* that Y actually meets such assumptions. A similar reasoning can be done in the side of Y. By combining appropriately the assumed and guaranteed properties, it is possible to establish the correctness of the entire system, without building the global state space. Moreover, once a guarantee is proved it can be used as an assumption for a later stage in the verification process. To prevent from erroneous conclusions *circularity* must be avoided in the reasoning chain. Finally, the assumptions often come in the form of abstractions such that both techniques are combined.

Figure 6.9 depicts the verification scheme for the $X \| Y$ system, using assume-guarantee reasoning with abstractions: in (b) the local properties of X are verified assuming Y' is a valid abstraction of Y; in (c) the local properties of Y are verified assuming X' is a valid abstraction of X; in (d) Y' is checked to be a valid abstraction of Y in order to guarantee (b); and in (e) X' is checked to be a valid abstraction of X in order to guarantee (c). Each *guarantee* verification checks for language containment of the implementation X (Y) with respect to the abstraction X' (Y'). In our framework, these proofs can be performed by checking that any output produced by the implementation can also be produced by the abstraction under the same input *stimuli*.

Induction

Induction is used to prove properties on systems composed of a number of similar components, organized in some inductively definable structure like a pipeline, a matrix, etc. These techniques rely on the concept of *invariant* [BSV94, ES96] or the so-called *behavioral fixed point* [VK98], to reason about the behavior of systems with any number of components.

Another possibility is to use mathematical induction over the size of the system, by successively increasing the number of components that form it. This process can be carried out manually, or can be automated using automatic theorem provers [LG95].

6.3.1 Framework

Several formal frameworks have been presented in literature [GL94, McM97, HRS98] that support correct assume-guarantee reasoning with abstractions. These frameworks often rely on a *preorder relation* $\leq$ between processes, a *composition operator* $\|$ for processes and a *logic* to specify properties. The preorder relation $X \leq X'$ denotes that the abstraction X' captures more behaviors than X, *i.e.* X *refines* or *implements* X'. The composition operator must be monotonic with respect to the preorder, *i.e.* $X \leq X' \wedge Y \leq Y' \Rightarrow (X \| Y) \leq (X' \| Y')$. And the properties must be preserved through the preorder, *i.e.* $X \leq X' \wedge X'$ satisfies property $P \Rightarrow X$ satisfies property P.

Since we verify safety properties, the only condition we have to enforce for an abstraction to be appropriate, is that its state space is a superset of that of the original system. That

is, each state of the original system corresponds to a state in the abstraction. Therefore, the observable properties are preserved through the abstractions and false positive results cannot be produced. This, together with the relative timing-based verification approach presented in Chapter 4, provides a sound framework for performing assume-guarantee verification with abstractions.

6.4 Verification of IPCMOS pipelines

This section shows how abstraction, assume-guarantee and induction can be combined with our strategy for verification with relative timing, in order to proof the correctness of an IPCMOS pipeline regardless of its length.

6.4.1 Verification strategy

The correct operation of an IPCMOS pipeline initially empty, is given by the following informal specification (S):

> *"Every data item entered to the pipeline is acknowledged once and only once at every stage".*

We verify the correctness of the IPCMOS control circuit, *i.e.* the data path is assumed to be correct. Even though the previous property involves a liveness and a safeness condition, both can be modeled as safety conditions during the calculation of the state space. They can be modeled by means of a deadlock-freeness invariant in the control circuitry of the pipeline, such that the control deadlocks when either some data is not acknowledged or some data cannot move to the next stage.

Additionally, specific conditions about the correct behavior of CMOS circuits must be also ensured. These conditions are described in Section 6.5.1.

In particular, all properties required in this case study have been modeled with very simple temporal expressions that require at most the analysis of 1-step transitions. Therefore, it is not necessary a powerful engine to verify branching time or linear time logic for such task.

Due to the pulse-driven nature of the architecture, its correctness strongly depends on the delay margins associated to the components of the control stage.

The goal of the verification is to check whether an IPCMOS pipeline with any number of stages behaves correctly according to S. That is to check:

$$IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq S \tag{6.1}$$

for any value of $n > 0$, where I_i are identical instances of the circuit implementation I of a stage. The environment is formed by the data sender IN and the data receiver OUT described in Section 6.2.5. IN and OUT are indeed part of the specification in the sense that S also specifies the interface behavior of the pipeline, and IN and OUT can be obtained by simply mirroring such behavior.

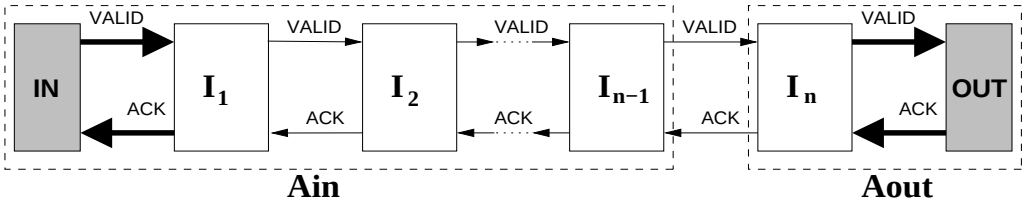


Figure 6.10 Pipeline verification using abstractions A_{in} and A_{out}.

The verification of (6.1) becomes exponentially more costly as n increases, specially because the communication protocol in both ends of a stage is highly decoupled. Thus, if the verification is carried out using the level of detail provided by I, in practice n cannot go beyond 2 stages. That is, although the number of untimed states can be computed for pipelines with several stages (see Section 6.2.6), when the time dimension is added for verification, the complexity is drastically affected. As a consequence, in order to overcome such complexity, the verification of longer pipelines must be carried out using abstractions.

IN and *OUT* operate according to the pulse-based protocol and so does the left side of a stage, whereas the right side of a stage operates according to a two-phase handshake protocol (see Section 6.2). Therefore, the communications between stages I_2 and I_{n-1} inside the pipeline use the handshake scheme (thin arrows in Figure 6.10), whereas the pulse-based behavior only appears at the extremes of the pipeline (thick arrows in Figure 6.10). We propose abstractions that hide the pulse-based behavior in a way that all the timing restrictions related to the correctness of such protocol are also encapsulated inside the abstractions (see A_{in} and A_{out} in Figure 6.10). Therefore, since A_{in} and A_{out} communicate through the handshake protocol, the abstractions can be untimed and *assume-guarantee* reasoning can be used. Hence, we pose the verification of (6.1) in terms of:

$$A_{in} \parallel A_{out} \leq S \tag{6.2}$$

We proceed as follows:

- Build abstractions A_{in} and A_{out} for the components shown in Figure 6.10. *IN* and *OUT* communicate by pulses respectively with I_1 and I_n inside the abstractions. On the other hand, the rest of the stages communicate by handshakes.
- Prove (6.2) *assuming* that A_{in} and A_{out} are correct abstractions of the respective parts of the pipeline.
- *Guarantee* the soundness of the abstractions. Discharge the assumptions by proving that A_{in} and A_{out} are indeed correct abstractions of $IN \parallel I$ and $I \parallel OUT$, respec-

Figure 6.11 STGs modeling the abstractions A_{in} (a) and A_{out} (b).

tively. Moreover, prove that A_{in} is also a good abstraction of $A_{in} \parallel I$, *i.e.* A_{in} is a *behavioral fixed point* that abstracts the sender IN and a chain of n stages.

- Finally, prove the correctness of a pipeline formed by a single fully detailed implementation (I) of a stage. The verification checks if I is a correct CMOS circuit and satisfies S in the given environment, that is $IN \parallel I \parallel OUT \leq S$.

The first three items are covered by the remaining of this section, whereas Section 6.5 shows in detail the use of the relative timing-based approach described in Chapter 4 to perform the proof in the last item.

6.4.2 Abstractions

The models A_{in} and A_{out} must describe the observable behavior of the abstracted parts of the pipeline at a higher level (see Figure 6.10). That is, A_{in} and A_{out} hide the internal communications inside the abstracted blocks. The two-phase handshake protocol in the communication interface of the abstractions is modeled by the fact that the rising edge of ACK to acknowledge a data portion is always interlocked within the falling edge and the next rising edge of VALID. The models for the environment (IN and OUT) used for the verification were already shown in Figure 6.8. Figure 6.11 depicts the models for the abstractions A_{in} and A_{out}, represented by Signal Transition Graphs. The underlined transitions represent inputs in their respective models.

A_{in}: abstraction of *IN*-$I_1 - \cdots - I_{n-1}$.
A_{in} hides the pulse-based communication between IN and the first stage of the pipeline. A_{in} signals the data availability at the input of the next stage of the pipeline by lowering the output VALID line. VALID is not raised again until the pipeline acknowledges the receipt of the data by rising the ACK line. The two-phase handshake protocol is completed by resetting the VALID line independently of the resetting of the ACK line by the pipeline.

A_{out}: abstraction of *I*-*OUT*.
A_{out} hides the pulse-based communication between the last stage of the pipeline and the OUT module. A_{out} samples the data available at the end of the pipeline signaled by the

low value of VALID, and acknowledges it by producing a positive ACK pulse. The resetting of the ACK and VALID lines to their initial state is done independently by the abstraction and the pipeline, respectively.

6.4.3 Assume-guarantee verification

The verification methodology described in Chapter 4 is used in this section to perform the experiments of the *assume-guarantee* strategy.

We want to prove that the abstract system built from A_{in} and A_{out} is a good abstraction of an IPCMOS pipeline, *i.e.* :

$$IN \parallel I_1 \parallel \cdots \parallel I_n \parallel OUT \leq A_{in} \parallel A_{out}$$

We use *assume-guarantee* reasoning in five steps in order to carry out the proof using the abstract models described above. Steps 3 and 4 are the ones that use induction to prove the correctness of an n-stage pipeline, for $n \geq 2$.

The second, third and forth verification steps are graphically depicted in Figure 6.12. The symbol $\Diamond$ models a component that checks that any event produced by the refinement is also produced by the abstraction (*i.e.* the language produced by the refinement is included in the language produced by the abstraction).

1. Assume: We must prove that the system formed by the abstractions meets the specification of the IPCMOS pipeline, that is: $A_{in} \parallel A_{out} \leq S$. Provided the models in Figure 6.11 this verification step is straightforward and completes successfully in less than a second of CPU time.

2. Guarantee correctness of A_{out}: We prove the correctness of A_{out} with respect to the system formed by the implementation of a stage of the pipeline I and the OUT module, when I communicates with the rest of the pipeline using the handshake protocol. That is: $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 6.12 (a) is built. The verification consists in checking the language containment of $I \parallel OUT$ with respect to A_{out}, which is reduced to checking that any output produced by $I \parallel OUT$ can also be produced by A_{out} at the same time instant. In this case, the only relevant output is signal ACK.

3. Guarantee correctness of A_{in} with one stage: We prove the correctness of A_{in} with respect to the system formed by the pulse-based IN module and the implementation of a stage of the pipeline I, when I communicates with the next stage in the pipeline using the handshake protocol. That is: $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this analysis, the system shown in Figure 6.12 (b) is built. The verification consists in checking that whenever I is ready to change the value of VALID, A_{in} is also ready for that, thus guaranteeing language containment of $IN \parallel I$ with respect to A_{in}.

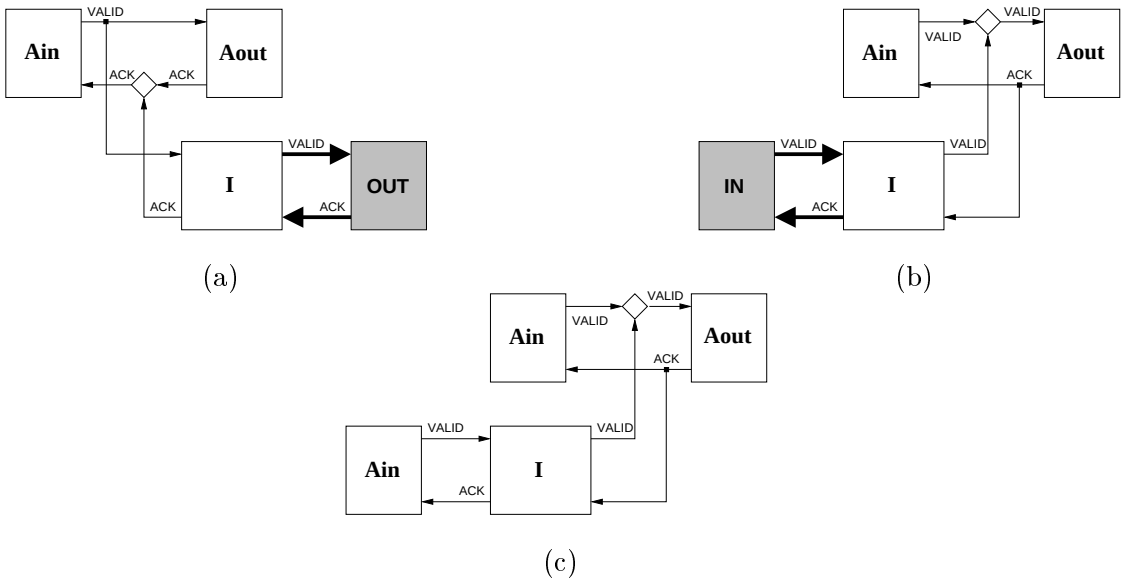


Figure 6.12 Scheme of the *guarantee* part of the verification to prove the correctness of the various abstractions: (a) $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$, (b) $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$ and (c) $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$.

4. Guarantee A_{in} is a *behavioral fixed point*: The previous proof only guarantees the correctness of A_{in} as an abstraction of IN and a single stage. However, that result serves as the induction hypothesis to prove that A_{in} is a correct abstraction of $IN \parallel I \parallel \cdots \parallel I_{n-1}$, for any $n \geq 2$, as shown in Figure 6.10. Namely, $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$. For this, the system shown in Figure 6.12 (c) is built and the verification is done similarly to the previous proofs, but now checking signal VALID.

A_{in} is an abstraction of the IN module and a chain of IPCMOS stages. Thus, A_{in} is said to be a behavioral fixed point [VK98], since no matter how large n is, A_{in} can be used as a correct abstraction. This, together with the previous proofs, demonstrate the correctness of an n-stage pipeline for $n \geq 2$.

5. Guarantee correctness of a 1-stage pipeline: The previous proofs demonstrate the correctness of IPCMOS pipelines with 2 or more stages. It is still needed to prove the correctness of a pipeline with a single stage, that is $IN \parallel I \parallel OUT \leq S$. This step is necessary to consider the case in which a stage is interacting with a pulse-driven environment at both sides. This is the step in which more timing constraints are required to guarantee a correct behavior of the components. Despite of its complexity, since this step also requires the refinement of the model at the level of transistors, we describe it in detail in Section 6.5.

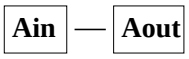
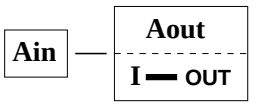
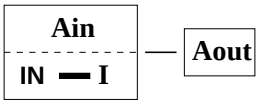
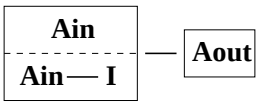
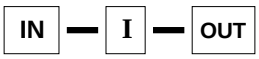
Proof	System	CPU time	Refinements
1. $A_{in} \parallel A_{out} \leq S$		$<$ 1 sec.	–
2. $A_{in} \parallel I \parallel OUT \leq A_{in} \parallel A_{out}$		28 min.	7
3. $IN \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$		9 min.	3
4. $A_{in} \parallel I \parallel A_{out} \leq A_{in} \parallel A_{out}$		10 min.	3
5. $IN \parallel I \parallel OUT \leq S$		35 min.	40

Table 6.1 Summary of the results for the 5 steps of the verification.

Table 6.1 summarizes the results of the five verification steps, using the TRANSYT tool implementing the relative timing-based verification approach described in Chapter 4. The first column shows the formula of the proof corresponding to each step of the verification strategy. The second columns depicts graphically the system built for each proof. The CPU times, indicated in the third column, have been rounded to minutes and correspond to executions in a $866MHz$ Pentium-III computer with 1GB of RAM running Linux. The number of refinements, indicated in the last column, correspond to the number of iterations needed by TRANSYT to successively incorporate the timing constraints that help pruning the failure traces from the state space of the corresponding models.

In the first experiment, no refinement is required since the verification only consists in computing the untimed state space of the abstractions involved and realizing that no violation of the specification arises. Notice also, that although experiments 2, 3 and 4 require a few refinements the CPU times are comparatively high with respect to that of experiment 5. This is due to the complexity of the required models (see Figure 6.12) and the resulting BDD explosion when doing reachability analysis. Finally, the last experiment requires a lot of refinements that correspond to all the constraints related to the timing-dependent pulse-based communication at both sides of the stage.

6.5 Verification of a stage

This section addresses the verification of the circuit implementation of an IPCMOS pipeline stage (I), in an environment formed by a data sender IN and a data receiver OUT. We will show the modeling mechanisms to describe the transistor-level circuits.

6.5.1 Modeling CMOS circuits

The behavior of the circuit is modeled by a TTS with a set of events that update the value of variables and therefore modify the state of the system. In particular we use a boolean variable to model each circuit node and several events that model the rising and falling transitions of the node value. For every event a transition relation is defined, including an enabling condition and a delay interval $[\delta^l, \delta^u]$ specifying the delay bounds of the signal switch once it becomes enabled.

A node in the circuit may be driven by stacks of pull-up and pull-down transistors, and possibly pass-transistors. Each stack is modeled by an event that sets the proper value to the variable (*one* for pull-up, *zero* for pull-down and copies the value of another variable for a pass-transistor). Delay intervals can be computed using the technology parameters (if they are available) and the fan-out conditions for each signal. The broader the delay intervals for which the circuit is proved to be correct, the more general is the verification, *i.e.* the more robust is the circuit.

As an example, consider the transition relations for signal Y in the *strobe switch* circuit (see Figure 6.5). Y+, takes place if Y is low and it is pulled up by the p-type transistor controlled by Z. Thus, the enabling condition is given by $En(\mathsf{Y}+) = \neg\mathsf{Y} \wedge \neg\mathsf{Z}$. Similarly, the enabling condition for Y− is given by $En(\mathsf{Y}-) = \mathsf{Y} \wedge \mathsf{ACK}$ because Y can only be pulled down by the n-type transistor controlled by the ACK signal.

We have carried out the verification process with no specific technology library in mind. Hence the delay bounds for each stack of transistors is assumed to be in the range of `[1,2]` abstract delay units. Other appropriate delay ranges are used in case of weak transistors or fan-out considerations. We want to remark that since our verification approach uses relative timing, the particular values of the delay bounds are only relevant to compute the relative differences between the accumulated delays of the critical paths related to a failure situations.

Tables 6.2, 6.3, 6.4, 6.5 and 6.6 summarize the models of each of the sub-circuits that compose a general IPCMOS control block. Thus, for example, the models of the *strobe* and the *reset* circuits consider multiple Vint and Aint input signals, respectively. Notice also that more than one event can be specified for the same signal switch if it is produced from different sources (*e.g.* the Vint+ event in Table 6.3). This allows potentially for a very detailed model in which an event can have different delays depending on what caused it. This is in direct correspondence with what happens in an actual circuit.

Enabling condition	Event	Delay	Comment
$\neg X \wedge \neg Rint \wedge \bigwedge_i \neg Vint_i$	X+	[5,6]	weak p-type transistor
$X \wedge (Rint \vee \bigvee_i Vint_i)$	X−	[2,3]	n-type transistor
$\neg K \wedge \neg X$	K+	[1,2]	inverter
$K \wedge X$	K−	[1,2]	inverter
$\neg ACK \wedge \neg K$	ACK+	[2,3]	inverter
$ACK \wedge K$	ACK−	[2,3]	inverter
$\neg CLKE \wedge \neg ACK$	CLKE+	[3,4]	inverter
$CLKE \wedge ACK$	CLKE−	[3,4]	inverter
$\neg Rint \wedge \neg CLKE$	Rint+	[1,2]	p-type transistor
$Rint \wedge CLKR$	Rint−	[1,2]	n-type transistor

Failure condition	Comment
$\neg CLKE \wedge CLKR$	Short-circuit at Rint

Initial state
$\neg X \wedge K \wedge \neg ACK \wedge CLKE \wedge \neg Rint$

Table 6.2 Model of the *strobe* circuit.

Enabling condition	Event	Delay	Comment
$\neg Rint \wedge \neg VALID$	Z+	[1,2]	inverter
$Z \wedge VALID$	Z−	[1,2]	inverter
$\neg Y \wedge \neg Z$	Y+	[1,2]	p-type transistor
$Y \wedge ACK$	Y−	[1,2]	n-type transistor
$\neg Vint \wedge Y \wedge VALID$	Vint+	[1,2]	n-type transistor
$\neg Vint \wedge \neg CLKE$	Vint+	[1,2]	p-type transistor
$Vint \wedge Y \wedge \neg VALID$	Vint−	[1,2]	n-type transistor

Failure condition	Comment
$\neg Z \wedge ACK$	Short-circuit at Y
$\neg VALID \wedge Y \wedge \neg CLKE$	Short-circuit at Vint

Initial state
$\neg Z \wedge Y \wedge Vint$

Table 6.3 Model of the *strobe switch* circuit.

Enabling condition	Event	Delay	Comment
$\neg$CLKR $\wedge$ $\bigwedge_i \neg$Aint$i$	CLKR+	[5,6]	weak p-type transistor
CLKR $\wedge$ ($\bigvee_i$ Aint$i$)	CLKR$-$	[2,3]	n-type transistor
$\neg$CLKRN $\wedge$ $\neg$CLKR	CLKRN+	[3,6]	3 inverters
CLKRN $\wedge$ CLKR	CLKRN$-$	[3,6]	3 inverters

Initial state
$\neg$CLKR $\wedge$ CLKRN

Table 6.4 Model of the *reset* circuit.

Enabling condition	Event	Delay	Comment
$\neg$Aint $\wedge$ $\neg$CLKRN	Aint+	[1,2]	p-type transistor
Aint $\wedge$ CLKRN $\wedge$ ACK	Aint$-$	[2,4]	2 n-type transistors
$\neg$failctl $\wedge$ CLKRN $\wedge$ ACK $\wedge$ $\neg$Aint	failctl+	[0,0]	
failctl $\wedge$ $\neg$ACK	failctl$-$	[0,0]	

Failure condition	Comment
failctl $\wedge$ $\neg$CLKRN $\wedge$ NS(CLKRN)	Slow resetting of ACK

Initial state
Aint $\wedge$ failctl

Table 6.5 Model of the *reset switch* circuit.

Enabling condition	Event	Delay	Comment
$\neg$W $\wedge$ $\neg$CLKE	W+	[1,2]	p-type transistor
W $\wedge$ CLKR	W$-$	[1,2]	n-type transistor
$\neg$VALID $\wedge$ $\neg$W	VALID+	[Delay]	delay(logic) - delay(*strobe*)
VALID $\wedge$ W	VALID$-$	[Delay]	delay(logic) - delay(*strobe*)

Failure condition	Comment
$\neg$CLKE $\wedge$ CLKR	Short-circuit at W

Initial state
$\neg$W $\wedge$ VALID

Table 6.6 Model of the *valid* circuit.

Provided this modeling mechanism, correctness of CMOS circuits can be posed in terms of the following properties:

Persistency. The temporal behavior of a gate is described by the *inertial* delay model. In this model, input pulses shorter than the lower delay bound δ^l are not propagated to the output. Pulses longer than the upper delay bound δ^u are always propagated. However, propagation of pulses with duration between δ^l and δ^u is uncertain and may produce *glitches*, hence signal *persistency* conditions are imposed. Persistency implies that every transition must fire once it is enabled and cannot be disabled by the firing of another transition.

Consider for example a given event e for which a persistency condition must be ensured. The following invariant condition must be satisfied:

$$\overline{EF(\mathsf{e}) \cdot \overline{EF'(\mathsf{e})} \cdot TR \setminus TR(\mathsf{e})}$$

That is, it must never happen that if event e is enabled in some state, the firing of another event of the system leads to a state where e is no longer enabled. In the expression, TR corresponds to the transition relation of the system, $TR(\mathsf{e})$ corresponds to the transition relation of event e, and EF and EF' are enabling functions expressed using current and next-state variables, respectively.

Notice that this condition for persistency is different to that presented in Section 5.3.6. There, the condition expressed the fact that the firing of an event x could induce non-presistency to any other event y in the system, *i.e.* x disables y. Conversely, the invariant presented here states the dual condition.

Short-circuits. Custom designs exploit the flexibility of CMOS technology, relaxing the complementarity between the pull-up and pull-down stacks. This introduces a potential source of short-circuits during circuit operation. Although short-circuits can be exploited by considering the pull-up/pull-down relative impedance, generally they are undesirable because they may leave the driven signals undefined, increase the power dissipation, or even cause a circuit damage. Therefore, the complementarity of the pull-up and pull-down conditions for each circuit node must be ensured.

Several potential short-circuits can happen in the *strobe* (signal Rint), the *strobe switch* (signals Y and Vint) and the *valid* (signal W) circuits. See Figures 6.5 and 6.7. Consider for example the potential short-circuits in the *strobe* circuit of Figure 6.5. They are identified by the following invariants:

1. $\neg\mathsf{Z} \wedge \mathsf{ACK}$: The Y node is pulled down by the n-type transistor controlled by ACK and pulled up by the p-type transistor controlled by Z. The short-circuit occurs if both transistors conduct.

2. $\neg$VALID $\wedge$ Y $\wedge \neg$CLKE : The VALID line is pulled down by the input and pulled up by the p-type transistor controlled by CLKE. The short-circuit occurs if the n-type transistor controlled by Y conducts.

Thus, the invariant conditions that must be satisfied during the verification are the negated of the short-circuit conditions. That is, (1) Z $\vee \neg$ACK and (2) VALID $\vee \neg$Y $\vee$ CLKE.

Failure conditions due to short-circuits in the different modules of an IPCMOS stage are summarized in Tables 6.2, 6.3 and 6.6.

6.5.2 Modeling IPCMOS circuits

Despite of the above conditions regarding the correctness of CMOS circuits in general, each circuit of the IPCMOS control block may exhibit its particular set of failure situations.

For example, in the *reset switch* circuit, a failure condition has been defined that models the situation in which signal ACK is slow to fall once the falling of Aint is produced. If ACK does not reset before CLKRN rises again, the circuit might produce a second falling edge in Aint, in response to a single rising edge in ACK. This situation cannot be captured with a boolean expression in terms of the current and next state signals of the circuit, since it involves a sequence of firings. Therefore, we have introduced an extra internal boolean variable failctl in the model. This variable is set to *one* when the falling edge in Aint is produced, and falls when ACK falls. In such a way the failure condition can be stated as that situation in which failctl is high (Aint fell) and CLKRN tries to rise. See Table 6.5.

Of special importance in the modeling are the bounds associated to the delay element in the *valid* circuit. Recall that such delay element mimics the operation time of the logic attached to the stage. The delay range should be general enough to allow correct operation with whatever logic attached to the stage, however the IPCMOS implementation imposes certain limitations on such possible delay.

Suppose a new data portion is already available at the stage inputs and so is notified by the low value of the incoming VALID line. On the other hand, the previous data portion is yet processed by the stage but the receiver has still not acknowledged it (outgoing VALID and incoming ACK are low). As soon as the acknowledgment arrives, a race is produced between to paths inside the stage. Namely, the sequence CLKR+ $\rightarrow$ W$-$ $\rightarrow$ VALID+ that resets the outgoing VALID signal to its initial high value, and the sequence CLKR+ $\rightarrow$ Rint$-$ $\rightarrow$ W+ $\rightarrow$ K$-$ $\rightarrow$ ACK+ $\rightarrow$ CLKE$-$ $\rightarrow$ W+ $\rightarrow$ VALID$-$ that notifies the availability of the new data portion to the receiver by lowering the outgoing VALID signal. In this situation, if the delay element behaves according to the inertial delay model, and the separation time of both sequences to complete is too small for the positive pulse of VALID to propagate, an undesirable data lost will occur. That is, VALID might be kept low such that no notification falling edge will be seen by the receiver. This situation would

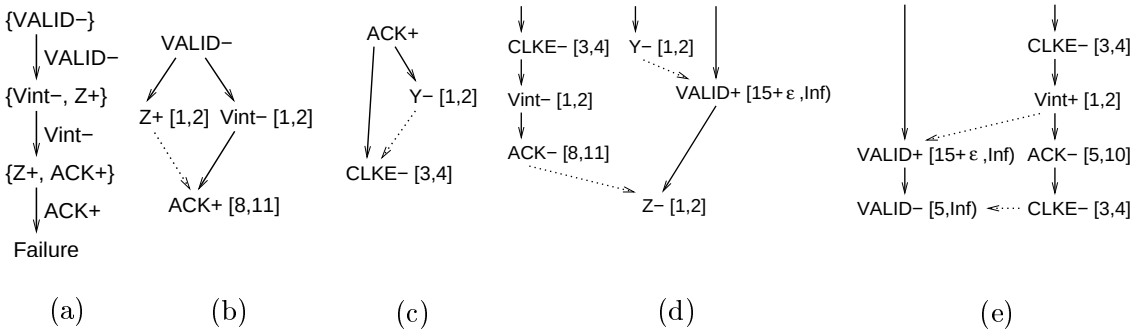


Figure 6.13 LzCES used to prove correctness of the *strobe switch* circuit: (a) Failure trace and (b) corresponding LzCES. (c)-(e) LzCESs showing other relative timing constraints (dotted arcs) for correctness.

be detected by the verification during the persistency check on the VALID signal. To avoid this faulty behavior the upper bound of the delay element in the *valid* circuit must be set in accordance to the difference between the accumulated upper delay bounds of the reset sequence, and the accumulated lower delay bounds of the notification sequence.

6.5.3 Verification results

Provided the models above, the verification succeeds proving that the invariants characterizing the circuit correctness conditions always hold. That is, the circuit implementing the IPCMOS stage operates correctly in the given *IN-OUT* environment and shows no short-circuits, no persistency violations and no deadlocks. The verification process finishes in about 35 minutes of CPU time in a $866MHz$ Pentium-III computer with 1GB of RAM running Linux.

The verification succeeds and also provides back-annotation indicating a set of sufficient timing relations between events that guarantee the correctness of the implementation. These relations are provided as the timed event structures obtained at every iteration of the verification process. This information permits to guess about the delay margins allowable to keep the correctness.

Figure 6.13 shows some of the timing relations obtained during the verification, that guarantee the correctness of the *strobe switch* module (see left side of Figure 6.5). Recall that signals ACK and CLKE are inputs coming from the *strobe* circuit, whereas VALID is an input signal coming from the environment.

For simplicity the chain of events Vint− → X+ → K− → ACK+ produced by the *strobe* circuit has been collapsed in figures (a), (b) and (d). Thus yielding an accumulated delay for ACK+ in the range [8,11]. Similarly, the chain of events Vint+ → X− → K+ → ACK− has been collapsed in figure (e), which yields an accumulated delay for ACK− in the range [5,10].

Figure 6.13 (a) shows a trace leading to a failure situation in which the early firing of ACK+ causes a short-circuit at node Y. Event structure (b) shows the actual ordering of Z+ and ACK+ in the timed domain proving that trace (a) is not timing consistent. This situation corresponds to the case where a falling edge of VALID occurs, followed by the fall of signal Vint and the rise of ACK to indicate the data receipt. Z+ must be faster than ACK+ to avoid the short-circuit at Y corresponding to invariant (1) of Section 6.5.1.

Event structure in Figure 6.13 (c) corresponds to a situation where after rising signal ACK, the transition Y− turns off the n-type transistor that isolates Vint from VALID. Thus Y falls before the VALID line is pulled up by CLKE−. This ordering is required to guarantee invariant (2) of Section 6.5.1.

Event structure in Figure 6.13 (d) shows a situation where event ACK− is ordered with Z− ensuring invariant (1). Indeed it shows that signal ACK always falls before Z thus avoiding the short-circuit. The delay of VALID+ is set so that the appropriate ordering of the events is guaranteed.

Event structure in Figure 6.13 (e) shows the ordering relation between CLKE+ and VALID−. The delay of VALID− is set to reset CLKE before the falling edge of VALID. This ordering contributes to guaranteeing invariant (2).

Similar event structures containing the timing constraints required to prove the correctness of the different parts of the circuit are generated during the verification. Instead of showing all of them, what would require a lot of space, we summarize some of the relevant conditions required for the circuit correct operation as follows.

Correctness of the *strobe switch* circuit

- Short-circuit at Y by early ACK+ in response to VALID− from the previous stage. Z+ must happen before ACK+, such that the p-type transistor driving Y opens, and prevents the short-circuit when ACK+ arrives. This requirement is met since it involves the delay of the inverter driving Z against a long chain of gates in both the *strobe switch* and the *strobe* circuits.

- Short-circuit at VALID once ACK+ is produced. In this case Y must discharge faster that CLKE coming from the *strobe* circuit. This requirement is met since it involves the delay of the transistor responsible of discharging Y, against the delay of the inverter driving CLKE, which is also a slow one due to fan-out considerations.

- Short-circuit at Y by slow ACK−. It must be guaranteed that ACK− is faster than Z−, *i.e.* the *strobe* circuit resets itself before the VALID line coming from the previous stage resets and charges Y again. This requirement is met by properly setting the lower delay bound of VALID+ in the *valid* circuit and the *IN* module.

- Short-circuit at VALID if a new VALID− from the previous stage is produced without allowing the proper resetting in CLKE of the previous cycle of operation. This requirement is met by properly the lower delay bound of VALID− in the *valid* circuit and the *IN* module.

Correctness of the *strobe* circuit

- Short-circuit at Rint due to an early CLKR+ in response to the rise of ACK. This is a problem if the self-resetting fall of ACK and the corresponding rise in CLKE are slower than CLKR. This requirement is met since CLKR+ is at the end of a long chain of events involving the *valid* circuit of the current stage, and also the *strobe* circuit of the next stage.
- Short-circuit at Rint due to a late CLKR− of the previous cycle of operation, against a new falling edge in CLKE. Solving this problem guarantees the complete resetting of the *strobe* circuit. This requirement is met since the new CLKE− comes from a long chain of events involving a new data VALID coming from the previous stage.

Correctness of the *reset* circuit

- Slow self-resetting of ACK in the *strobe* circuit against a complete cycle of operation in the *reset* circuit, right after ACK+ is produced. This requirement is met by balancing properly the delays in both paths, since the path leading to ACK− is just the self-resetting part of the cycle of operation of the *strobe* circuit, whereas the path in the *reset* circuit corresponds to a complete set and reset cycle of the involved signals.

Correctness of the *valid* circuit

- Persistency violation of VALID+ caused by an early firing of W+ after CLKR+ has fired. To meet this requirement the upper delay bound of VALID+ must be set below the minimum accumulated delay of the chain of events leading from CLKR+ to W+, minus the maximum delay of W−, who triggers VALID+ after CLKR+.

Despite of these list of conditions for circuit correctness, recall that the pulse-based environment modules must also satisfy certain conditions. In particular, a restriction is imposed to *OUT* to avoid early resetting of the ACK line. That is, if ACK− arrives too fast after ACK+, the falling edge of ACK may not be properly recorded by the *reset switch* circuit of the last stage of the pipeline. Therefore a minimum width is required to the positive pulse of ACK. Similarly, a restriction is imposed to *IN* to avoid early resetting of the VALID line. That is, if VALID+ arrives too fast after VALID−, the falling edge of VALID may not be properly recorded by the *strobe switch* circuit of the first stage of the pipeline. Therefore a minimum width is required to the negative pulse of VALID.

Finally, we want to remark the difficulty of our verification approach to check some of the invariants required for correctness. Namely, the condition corresponding to the short-circuit at W inside the *valid* circuit (see Table 6.6), needed more than ten refinements, each covering different excitation sequences. Similarly happens with the failure condition about the short-circuit at node Y inside the *strobe switch* circuit. Further analysis is required to clarify the reasons for such computational effort. We believe this effort can be significantly reduced by constructing better event structures which abstract away the complexity due to concurrency-causality combinations.

6.6 Conclusions

The verification of a complex timed system, the IPCMOS architecture, has been tackled in this chapter. The correctness of the system highly depends on the delays of the internal gates of the circuit, and also of the environment.

The verification has been carried out by combining the core verification algorithm presented in Chapter 4, together with the use of assume-guarantee reasoning to perform a hierarchical verification by means of abstractions, and the use of mathematical induction to prove the correctness of infinite-state systems. As a result, it has been proved the correctness of an IPCMOS pipeline regardless of the number of stages that conform it.

The use of the relative timing-based verification approach has been crucial to prove the correctness of such a complex system. Although some other parametrized systems have been verified in the past (see *e.g.* [LG95]), this is the first case in which delay information and refinements down to transistor level have been provided.

The abstractions of different components of the system have still been derived manually. Also, the chain of assume-guarantee proofs and the required systems for verification have been built manually. Automatic extraction of timed abstractions and automatic derivation of the subsequent chain of reasoning are important topics for future research in this area.