Using Crystal for Timing Analysis

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-0865
Arpanet address: ousterhout@berkeley
Uucp address: ucbvax!ousterhout

This user manual corresponds to Crystal version 2.

1. Introduction

Crystal is a program that analyzes the performance of VLSI circuits. Its input consists of a circuit description extracted from the mask layout by the Mextra program. Users also supply a few lines of text to guide the analysis. Crystal then determines how long each clock phase must be and outputs information about the portions of the circuit that cause the worst delays.

Crystal helps in performance tuning by pointing out paths that limit clock speed. It is intended for circuits designed using multiple non-overlapping clocks. It will determine the length of each clock phase, but will not check clock skew or set-up and hold times. Circuits using more complex timing disciplines may require additional timing analysis besides what Crystal provides.

This manual is a tutorial on how to use the Crystal commands to get accurate timing information. It should be used together with the Unix *man* page, which provides detailed syntax information along with more concise descriptions of the commands, options, and built-in tables. Crystal is easiest to understand if you try it out on simple test cases while you read the manual sections.

2. Timing Analysis versus Simulation

Crystal's approach is very different from simulation, so the way you'll use it is quite different from the way you use a simulator. The difference is that Crystal does not consider specific data values. When you use a simulator like SPICE, you invoke a run by giving specific values for all the inputs to the circuit. The simulator then tells you exactly what will happen at each node at each point in time. When using Crystal, the goal is to specify as little as possible about your circuit. You only give Crystal vague information about a few nodes in the circuits (usually the clocks). Wherever Crystal doesn't have specific information from you, it chooses the worst possible alternative. Crystal combines all these worst possibilities to find the overall slowest path through the circuit, which it presents to you.

Simulation results are only as good as the specific choice of test cases: if your test cases don't exercise a particular portion of the circuit, bugs in that portion may go undetected. The advantage of Crystal's value-independent approach is that it is guaranteed to find the worst-case timing behavior of the circuit. Crystal tries all possibilities at each point and picks the worst, so it doesn't depend on designer input to find the critical paths. Furthermore, Crystal does all this in a single run. Simulation requires separate runs for the different test cases, which can be expensive for large circuits.

The disadvantage of Crystal's approach is that it may examine paths that could not occur in the actual chip. For example, Crystal may examine a path whose first portion can occur only when signal A is zero and whose second portion can occur only when A is one. Unless the value of A has been specified explicitly, Crystal will assume that A could be zero in the first portion of the path and one in the second portion. False paths like this result in camouflage that may hide the true critical paths. To eliminate false paths, you restrict Crystal's analysis by fixing a few node values, by restricting the way that signals can flow through transistors, and by giving Crystal specific information about which nodes to watch. Sections 10 and 11 show how to do this. You should try to get by with as little additional information as you possibly can: if you restrict the analysis too much, you may accidentally prevent Crystal from examining the true critical path. The best way to use Crystal is to start out with no additional information, and then add only the bare minimum that's needed to eliminate the false paths.

Crystal is not a replacement for a simulator. Since it ignores most data values, it doesn't give any information about whether your circuit is functionally correct; it will merely tell you how fast it will run. However, by analyzing the timing behavior for you, it allows you to use a fast high-level simulator that ignores timing behavior (ESIM, for example) instead of a slow circuit-level simulator like SPICE. Crystal's models for timing are much simpler than SPICE's. This makes the program run fast, but produces less accurate results in some situations. Section 14 discusses Crystal's models in detail.

3. Signal Flow, Stages, and Delay Analysis

Crystal analyzes your circuit in terms of *signal flow*. "Signals" means zero or one signals, not current or electrons. Signals flow from sources to targets. Signal sources are the chip inputs and the Vdd and GND supply rails. In addition, nodes of the circuit that are labelled as busses are also considered to be signal sources in some situations (see Section 9). Signal targets are places where information is used: gates of transistors and the chip outputs. A *stage* is a path leading from an signal source through transistor channels and other nodes to a target. If all the transistors in a stage are turned on, then a signal can flow from the source to the target. See Figure 1.

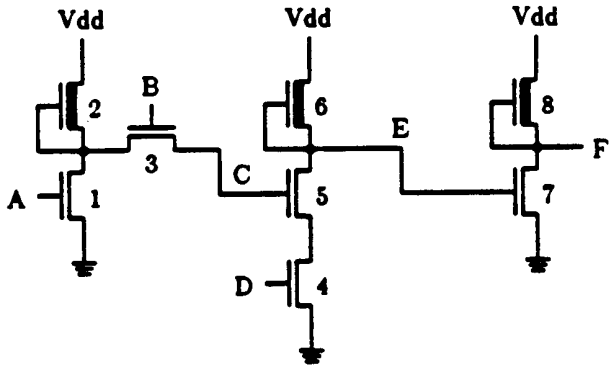

Figure 1. The path from Vdd to C through transistors 2 and 3 is a stage. If you tell Crystal that node A can fall at a certain time, Crystal will infer that node C might rise at a later time, node E might fall at a still later time, and node F rise latest of all.

To start a delay analysis, you give Crystal the time when some signal in your circuit rises or falls (usually this is the input pad for a clock signal). Crystal finds all the signal targets that can be reached from that node. For each target that is a gate, Crystal looks for stages that might be activated by the change in the gate. For each stage that it finds, Crystal computes the time when the stage's target will change value. Then if the target is a gate, Crystal repeats the whole analysis recursively by finding other stages that the gate change might activate. This continues until all possible consequences have been examined.

For example, in the circuit of Figure 1, if you tell Crystal that node A can rise at time 0, Crystal will realize that this change could activate a stage from GND to C through transistors 1 and 3. Whether or not this happens in the actual circuit depends on the value of B; if you haven't specified that value, Crystal will assume that it might be 1, so it will examine the stage. Using the information in the stage, Crystal will compute the delay to C, and use it to update the worst-case fall time for C. Since C connects to the gate of transistor 5, Crystal will then realize that when C falls, it could turn off the pulldown stage from GND through transistors 4 and 5 to E. This could activate the pullup stage from Vdd to E through transistor 6, so Crystal will examine that stage (of course, if node D is 0 then the pulldown stage was already turned off and the change in C has no effect; if you haven't explicitly told Crystal that D is 0, it will assume that it might be 1). Finally, when E rises it could activate the stage from GND through transistor

7 to F, so the worst-case fall time for F will be updated.

If the circuit has many input signals, you invoke the delay analysis again for each of them. Crystal remembers the worst delays seen in any of the analyses. After all the delay analysis has been done, you tell Crystal to print out the worst-case paths through the circuit. A path is a sequence of stages, each causing a change in the next. The worst-case path is the one whose final target reaches its final value later than any other node in the circuit. For example, the path from A to C to E to F is the worst case path in Figure 1. Information about the worst-case paths is recorded by Crystal as part of the delay analysis; you can control how many paths Crystal records.

4. Naming Nodes

Many of the Crystal commands take node names as parameters. A name can either refer to a single node or to a group of nodes. There are two forms for group names. The first form selects nodes whose names form a numerical sequence. The limits of the sequence are delimited by angle brackets (which are not part of the name). Thus, **Bit<1:4>** selects the nodes with names **Bit1**, **Bit2**, **Bit3**, and **Bit4**. To select a node whose name contains an angle bracket, use a backslash character in front of the bracket. For example, type **TrueIfX\<Y** to select the node whose name is **TrueIfX<Y**. To get a backslash in a node name, use two backslashes in a row.

The second form of group name selects all nodes whose names contain a given pattern. The name is specified as a star follwed by the pattern. Thus, ***abc** selects all nodes containing the pattern **abc**. Only simple pattern matching is done. The name ***** selects all nodes in the circuit.

5. How to Run Crystal

Invoke Crystal with the shell command

crystal *file*

where *file* is the name of a .sim file. If you want to modify Crystal's timing models, then you should not specify *file* on the command line; use the **build** command to read the file in after changing the models. The .sim file should have been created by Mextra. If Mextra was run with the **-o** switch (thereby generating "N" lines in the .sim file), then Crystal will know about parasitic capacitances and resistances associated with wires. If the **-o** switch wasn't specified to Mextra, then there will be "C" lines in the .sim file instead of "N" lines and Crystal will only know about parasitic capacitances. A .sim file shouldn't contain both "N" and "C" lines. Note: Crystal will not work with .sim files generated by Cifplot using its **-x** option.

Crystal reads its commands from standard input and writes its output to standard output. Each input line consists of a command name followed by arguments. The fields are separated by spaces or tabs. Any unique abbreviation for a command name is acceptable. If the first character of a command line is an

exclamation point, then the whole line is treated as a comment and ignored.

Commands are divided into seven groups, which should appear in the following order:

Model commands — These commands modify the timing models that Crystal uses to compute delays, and must appear before the circuit is read in. The model commands are **model**, **parameter**, and **transistor**. See Section 14 for information on how the models work and how to change them.

Circuit commands — Circuit commands are used to input the circuit and provide additional information about it, such as inputs and outputs. The circuit commands are **build**, **bus**, **capacitance**, **inputs**, **outputs**, and **resistance**.

Dynamic node command — This group includes the single command **markdynamic**, used to find and mark the dynamic memory nodes in the circuit. Section 13 describes how to use this command.

Check commands — There are two commands in this group, **check**, and **ratio**. They are used to examine the circuit's structure for suspicious looking electrical features, and may be useful in pointing out places where you need to provide extra information to Crystal. See Section 12.

Setup commands — Setup commands are used to restrict the paths that Crystal can examine in any given delay analysis. This group includes the **flow**, **precharged**, **predischarged**, and **set** commands.

Delay command — This group contains only a single command, **delay**, which performs the actual delay analysis.

Miscellaneous commands — These commands are used to set internal options and print out results and statistics. They can be invoked at any time. Commands in this group are: **alias**, **critical**, **dump**, **fillin**, **help**, **options**, **prcapacitance**, **prfets**, **prresistance**, **source**, **statistics**, and **undump**.

The only command outside these groups is the **clear** command, which resets information that was set by setup and delay commands. After **clear**, input may resume with anything except model commands. **Clear** is used to perform several different timing analyses (for example, for different clock phases) without having to read in the circuit again.

6. Simple Runs on Combinational Circuits

The simplest use of Crystal is for combinational (unclocked) circuits, where you are interested in knowing how long it takes for a change in an input to propagate throughout the circuit. Only four commands need be used: **inputs**, **outputs**, **delay**, and **critical**. First, you must identify to Crystal the circuit inputs (nodes that are driven by the outside world, such as input pads) and the circuit outputs (nodes whose values are used by the outside world). This information is used by Crystal in figuring out how signals can flow. For example,

inputs Bus<31:0> Select
outputs Overflow

identifies the 32 bus bits and the **Select** signal as inputs, and the **Overflow** signal as an output. See Section 9 for more on the **inputs** and **outputs** commands.

Delay commands are used to tell Crystal when input signals change value. For example,

delay BusBit 0 2

indicates that the latest time when **BusBit** will rise is time 0ns and the latest time when **BusBit** will fall is time 2ns. Crystal will then examine the consequences of this change to determine the latest possible rise and fall times for all other nodes affected directly or indirectly by **BusBit**. A negative time in a **delay** statement means that the transition never occurs:

delay Select -1 0

means that **Select** is initially 1, and will become 0 no later than time 0. Thus, only the falling transition of **Select** will be considered in the delay analysis. Many consecutive **delay** statements can be used where there are many inputs that change at different times.

After the **delay** commands, all that is needed is to print out the critical path. The **critical** command can be used for this. It requires no arguments.

7. Simple Runs on Clocked Circuits

Clocked circuits are handled like combinational circuits, except that there is a separate group of **delay** and **critical** commands for each clock phase. Typically, things in the circuit happen in response to the rising edges of clocks, and we'd like to know how long it takes for everything to stabilize once the clock phase has begun. Thus, there is usually a **delay** command of the form

delay Phi1 0 -1

in the group for each clock phase. If no other **delay** commands are given, it is assumed that all other input signals stabilize long before the clock rises.

The command

clear

is used between the commands for the different clock phases; it clears out old

delay information. An alternate way to handle different clock phases is with a completely separate Crystal run. However, for large chips it takes a long time to read in the circuit so it is usually faster to process all clock phases in a single run.

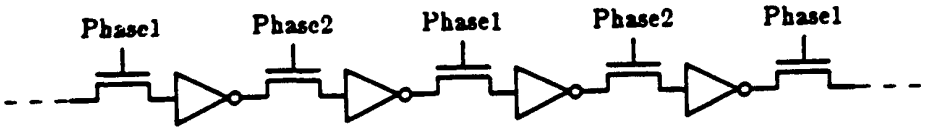

Figure 2. If Crystal doesn't know that Phase2 is zero, then during Phase1 analysis it will consider a path from the left end of the shifter all the way to the right end. If a **set** command is used to tell Crystal that Phase2 is zero, then Crystal won't propagate delays through the pass transistors that are turned off.

In addition to the **clear** commands between clock phases, **set** commands will be needed just before the **delay** commands for each phase. A **set** command indicates that a particular node will always have a particular value during the ensuing delay analysis. For example,

set 0 Phi<2:3>

indicates to Crystal that **Phi2** and **Phi3** will be 0 during the analysis. Crystal uses **set** information to avoid delay paths that cannot occur, as illustrated in Figure 2. The **clear** command will erase information from previous **set** commands; see Section 10 for more details on **set**.

Although the simple set of commands described above will work for many circuits, there are other circuits where it won't work very well. In particular, circuits with networks of transistors used for multiplexors or shifters require additional information that is discussed in Section 11. If only the simple commands are used for these circuits, Crystal will either produce pessimistic results or it will never finish. The sections below describe how to get more information out of Crystal and how to feed additional information into Crystal to produce more accurate results more quickly.

8. More on the Printing Commands

Besides the simple usage of the **critical** command, there are several additional ways that Crystal can print information. All of the printing commands are in the "miscellaneous" command group, so they can be invoked at any time.

8.1. Graphical Command Files

The printing commands will generate graphical command files if you wish. The command files can be used to highlight nodes and transistors using layout editors like Caesar, Magic, and Squid. The default for such files is Caesar format (the **options** command can be used to change the format to Squid or Magic style). The **-g** switch is used to generate the command files. For example,

critical -g dum

will generate in file **dum** a list of Caesar commands that will highlight the critical path. To use a Caesar command file generated in this way, do the following: first,

edit the circuit in Caesar; second, select a view that contains the entire circuit (using the **v** short command if necessary); third, use the **:source** long command to process the command file. The commands will place splotches of the error layer along with labels to identify "interesting points" on the circuit. Boxes are pushed on the box stack so that you can step from one interesting point to another using the **:popbox** long command. The interesting points and labels are different for different Crystal commands. In the **critical** command, for example, the points are the gates of transistors along the worst-case timing path, and the label for each point shows the delay to that point.

8.2. Critical Paths

The **critical** command prints out delay paths through the circuit and has the following form:

critical [**-g** *graphicsFile*] [**-s** *spiceFile*] [*textFile*] *number number* ...

For each *number* given, information about the *number*th slowest path in the circuit is output (Crystal only records a small number of the slowest paths; to change this number use the **options** command). If the **-g** switch is given, graphics output is generated. If the **-s** switch is given, a SPICE deck is generated for the critical path. If *textFile* is given, a textual description of the critical path is written to that file. If none of *graphicsFile*, *spiceFile*, or *textFile* is given, a textual description is output on standard output.

SPICE decks generated by Crystal contain circuit description cards and transient analysis cards, but no model cards; you should add your own model cards to the beginning of the deck. The circuit contains all the transistors and parasitic resistances and capacitances along the path, including gate-source and gate-channel capacitances for transistors that aren't part of the path but connect to it. Node 0 is used for GND, node 1 for Vdd, and node 2 for the substrate body. Crystal generates a card for Vdd, but it doesn't know what the body bias voltage is, so you must add your own card to the deck to generate it.

Crystal actually records three separate lists of slow paths, corresponding to different categories of nodes. The first list is for all nodes. The second list is for paths leading to memory nodes, and the third list is for paths leading to nodes that you have specially requested to be watched, using the **watch** command. Normally the *numbers* in the **critical** command refer to the overall list. However, if you end the number with the letter "m", then the the *number*th slowest path to a memory node is printed. For example, "1m" refers to the slowest path to a memory node. Similarly, the suffix "w" is used to refer to the list for watched nodes: "2w" refers to the next-to-slowest path leading to a watched node. The lists for memory and watched nodes are explained in Section 13.

8.3. Capacitance and Resistance Information

Prcapacitance and **prresistance** have similar syntax and are used to print out nodes with large capacitances or resistances:

prcapacitance [**-g** *graphicsFile*] [**-t** *threshold*] *node node ...*
prresistance [**-g** *graphicsFile*] [**-t** *threshold*] *node node ...*

For **prcapacitance** the threshold is in picofarads, and for **prresistance** the threshold is in ohms. The thresholds default to zero. If no nodes are specified, then the entire circuit is searched for nodes whose capacitance or resistance is greater than the threshold. If nodes are specified, then only those nodes are considered. A line of output is generated for each node exceeding the threshold. For example, **prcap abc** will print out the capacitance at node **abc**, and **prres -t 10000** will print out all nodes with lumped resistance greater than 10 kohms. The **-g** switch is used to generate a graphical command file. If Crystal encounters several nodes with exactly the same resistance or capacitance, only the first is printed. At the end of the printout, Crystal lists how many duplicate values were discarded.

8.4. Information about Transistors

The command

prfets *node node ...*

will print out lots of information about each transistor whose gate attaches to one of the *nodes*. If no *node* is given, then information is printed about all transistors.

9. Circuit Commands

Commands in the "circuit" group are used to read in the circuit and give Crystal additional information about it. Information from circuit commands lasts for the entire Crystal run, and isn't affected by **clear** or any other commands.

9.1. Reading in the Circuit

The command

build *file*

is used to read in the circuit. *File* is the name of a file in .sim format. If you type a filename on the command line when you invoke Crystal, then the **build** command is automatically invoked. However, if you wish to modify the circuit models, you must do so before reading in the circuit. In this case, don't give a filename on the command line, but use **build** instead. See Section 14 for information on changing the models.

If a node has been labelled several times, then Mextra picks one of those labels to identify the node. The other names are recorded in an alias file but are not used in the .sim file. If you'd like to use one of the aliases to refer to a node, rather than the name Mextra chose, you can use the command

alias *file*

to read in the ".al" file produced by Mextra and add the aliases to the Crystal's

name table.

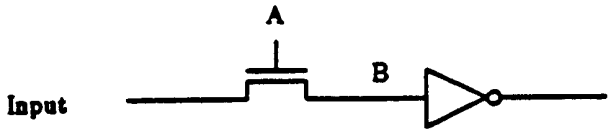

Figure 3. If Input isn't marked as an input, Crystal will not realize that it is a source of signals, and will mistakenly assume that a change at A has no effect on B.

9.2. Inputs and Outputs

These two commands were introduced in Section 6. They have the form

inputs *node node ...*
outputs *node node ...*

Crystal uses information about inputs and outputs to determine how signals can flow around the circuit: inputs and Vdd and GND are assumed to be sources of either a logic one or logic zero, and outputs and gates are assumed to be signal targets (places to which signals flow). If you forget to tell Crystal which nodes are inputs and/or outputs, it may miss some signal flows and overlook the critical path (see Figure 3). The **check** command can help to find nodes that should be marked as inputs.

Any input node that is not also an output node is assumed to be driven entirely from off chip. Crystal assumes that nothing on the chip can affect the value of the node, so if the node isn't used in a **delay** command, then Crystal will assume that its value never changes during the timing analysis. However, if a node is marked as both an input and an output, then Crystal will calculate delays to the node from the rest of the circuit. Usually only pads are marked as inputs, but this need not necessarily be the case. Marking a node as in input is roughly equivalent to applying a probe to the circuit at that point.

9.3. Changing Parasitic Values

Two commands are available to override Crystal's computation of parasitic capacitance and resistance:

resistance *ohms node node ...*
capacitance *pfs node node ...*

These commands will replace Crystal's computed value for the parasitic resistance or capacitance of one or more nodes with the specified value. There are at least two situations where this may be useful. For pads, there is relatively little capacitance on-chip, compared to the off-chip capacitance that must be driven. The **capacitance** command can be used to simulate the presence of the off-chip capacitance. The **resistance** command is used primarily to compensate for errors in the way Crystal computes resistances. To compute the internal resistance of a node, Crystal sums all of the internal resistances of all the wires connected to the node. All of the transistor gates attached to the node are assumed to be driven through all of the resistance. If a node has no branches this will give an accurate

result, but if the node has many branches then Crystal will substantially overestimate the resistance (this happens commonly for clock lines). The **resistance** command should be used to correct such situations. Since Crystal's resistance calculation is conservative, I suggest that you not use the **resistance** command until you discover that a bad resistance value is causing Crystal to overestimate the critical path.

9.4. Bus

There are a few occasions where, without guidance from the user, Crystal will chase around the circuit almost endlessly during a **delay** command without getting anywhere. This section describes once such scenario, and Section 11 describes another one that is even more serious.

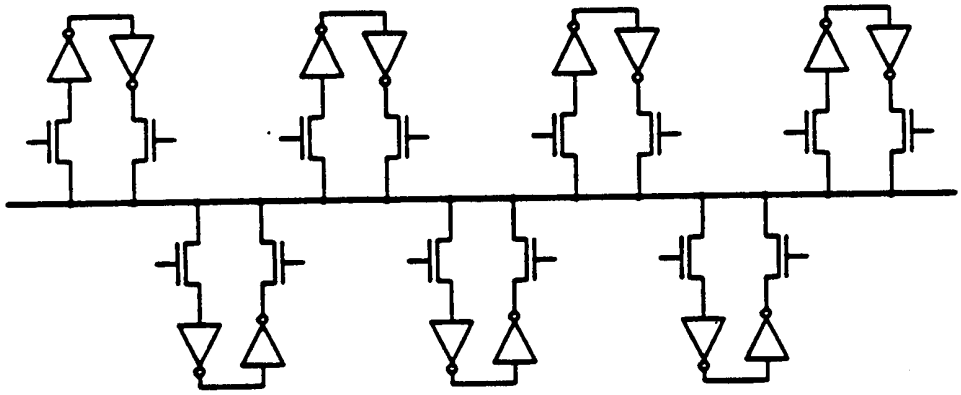

Figure 4. Without any additional information, Crystal will make a separate examination of every path from an output in one cell to an input in another cell. If Crystal knows about the presence of the bus, it first examines all paths from outputs to the bus, then examines paths from the bus to inputs. This makes the analysis much faster.

One situation where Crystal works too hard is the case of a bus with many elements attached to it. Figure 4 shows such a situation. During delay analysis, Crystal will check separately each path from the output of each bus element to the input of each other bus element, resulting in total work proportional to the square of the number of elements on the bus. If Crystal is told that the connecting node is a bus, then it breaks up the paths into separate stages from the elements onto the bus and from the bus to the inputs of the elements. For N elements on the bus, this results in 2N stages to examine instead of N^2. The **bus** command has the following syntax:

bus *node node ...*

Nodes marked as busses are treated both as signal sources and as signal targets.

It is only safe to mark a node as a bus if its capacitance is much greater than the internal capacitances of its elements. If this is not the case, then delays through the supposed bus will be underestimated. Crystal automatically marks all nodes with more than 2 pf of capacitance as busses (the threshold value can be changed with the **options** command; to prevent Crystal from automatically marking busses, use a very high threshold).

10. Setup Commands

Commands in the "setup" group are used to give Crystal additional information to restrict the paths it examines in **delay** commands. The **clear** command will erase any information provided by setup comands.

10.1. Set

The **set** command indicates that a node is fixed in value. Its syntax is

set 0/1 *node node ...*

When you tell Crystal that a node is fixed in value, Crystal performs a simple logic simulation to see if that fixed value causes other nodes to be fixed as well. For example, if an input of a NAND gate is set to 0, Crystal will deduce that the output is fixed at 1. If an input of a NOR gate is fixed at 1, then the output must be 0, and so on. See Section 14 for a description of how Crystal does the logic simulation. When processing delays, Crystal checks transistors to see if their gates are fixed in value. If a transistor's gate is forced to the value that turns the transistor off, then no signals can flow through the transistor.

If a node is forced to a value by a **set** command, then Crystal assumes that its value can never change during the timing analysis; that node will never appear in a critical path. Because of this, you should use **set** sparingly, lest you accidentally mask the critical path. Normally, **set** is used only to turn off all clock phases but one and to disable diagnostic circuitry such as scan-in-scan-out loops.

10.2. Precharging

The commands

precharged *node node ...*
predischarged *node node ...*

indicate to Crystal that the nodes are precharged to 1 or 0, respectively. When a node is precharged, Crystal assumes that it has an initial value of 1 and can only change to 0. Delays that would pull the node to 1 are ignored. When a node is predischarged, Crystal assumes that it has an initial value of 0 and can only change to 1. Delays that would pull the node to 0 are ignored. Precharged nodes are assumed to be highly capacitive, so they are treated like busses.

11. Pass Transistor Flow

As mentioned in Section 9.4, there are a few situations where Crystal can end up doing more work than necessary. The most severe examples of this concern pass transistors. Because Crystal does not generally have information about specific data values, it may examine impossible paths through pass transistors. Figures 5 and 6 show two cases. In Figure 5, Crystal will produce a pessimistic delay to Output2 by examining a path that passes forward and backward through

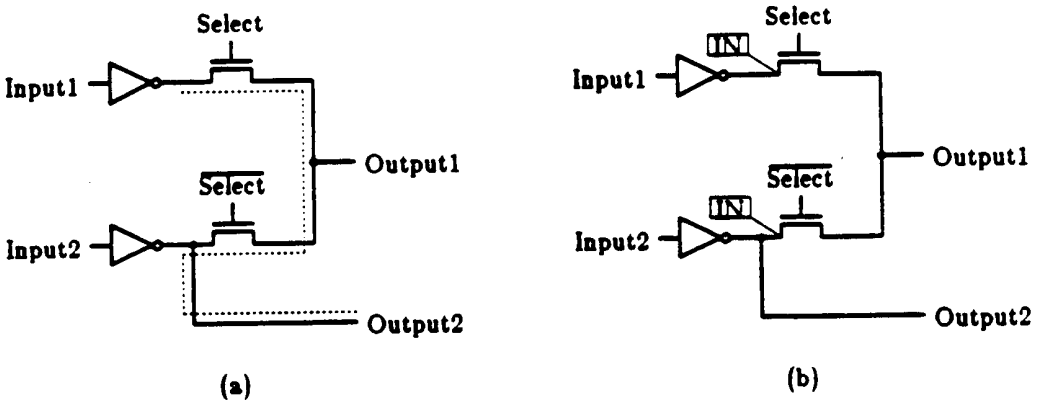

Figure 5. If Crystal doesn't know about pass transistor flow, it will consider the impossible path shown in (a). If the pass transistor flow is labelled with attributes, as in (b), then Crystal will consider paths from **Input1** to **Output1** and from **Input2** to both outputs, but it will not consider the path from **Input1** to **Output2**.

the multiplexor. In Figure 6, there is an enormous number of contorted paths through the shifter array. Crystal will attempt to examine every distinct path, even though the values on the control lines will prevent most of the paths from occurring in the actual circuit.

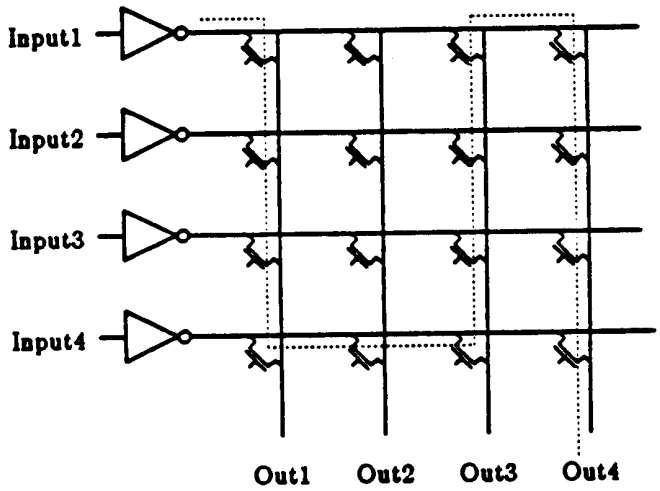

Figure 6. Crystal will consider long snake-like paths through this barrel shifter structure unless pass transistor flow information is provided.

To keep Crystal from chasing impossible paths, you must give it additional information about which way signals flow through pass transistors. Flow is indicated using *transistor attributes* in the CIF files that are input to Mextra. A transistor attribute is a label (CIF "94" construct, or a standard Caesar label) that touches the gate region of a transistor and ends in the character "$". Crystal ignores all attributes unless their first characters are either **Cr:** or **Crystal:**. To indicate the direction of signal flow, attach an attribute to a transistor's source or drain; this is done by placing the label exactly on the line between the gate and the source or drain (attributes placed entirely within the gate region are attached to the gate of the transistor and are used to identify the type of the transistor; see Section 14 for details).

For pass transistors that are unidirectional, two special attributes, **In** and **Out**, may be used. To use the **In** attribute, place a label of the form **Cr:In$** or **Crystal:In$** on the source or drain edge of a transistor gate. This indicates that whenever a 0 or 1 signal passes through the transistor, the source of the 0 or 1 is on the same side of the transistor as the **In** attribute (i.e. the 0 or 1 flows into the transistor from that side). The **Out** attribute indicates just the opposite, namely that 0's and 1's flow out of the transistor at that side.

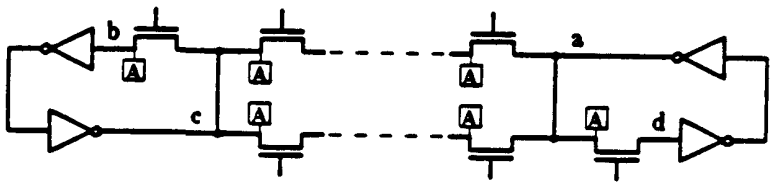

Figure 7. Named attributes can be used to control flow in bidirectional structures. In this case, paths from a to b and from c to d will be considered, but the path from a to c to d will not be considered (flow must be unidirectional with respect to tags of a given name).

Bidirectional pass transistors cause special problems. To handle bidirectional structures, one terminal of each pass transistor in the structure should be labelled with an attribute other than **In** or **Out**. See Figure 7. These attributes limit the way that signals may flow through the array: Crystal only allows signals to flow unidirectionally with respect to the attributes. This means that Crystal will consider any path through the array as long as the signal either a) flows into each transistor from the labelled side, or b) flows out of each transistor from the labelled side. A path will be ignored if a signal enters one transistor from the labelled side and leaves another from the labelled side. This allows signals to cross the structure in either direction, but will not allow them to criss-cross back and forth.

If different bidirectional structures are labelled with different attributes, then they are treated independently by Crystal. For example, Crystal will consider a path that enters at one transistor at a side labelled **Cr:A$**, and leaves another transistor at a side labelled **Cr:B$**. However, if the attribute **Cr:A$** is used for both transistors then the path is ignored.

Only a small number of transistors in any design should need to have flow attributes. These transistors can be identified in either of two ways. The easiest way is to use the **check** command, described in Section 12 below, to identify candidates for flow tagging. The hard way is just to run Crystal: if you haven't placed enough tags, then either Crystal will suggest impossible critical paths, or it will abort the delay analysis because it found too many paths. In the first case, it will be easy to identify the transistors that need flow tagging by looking at the critical path. In the second case, you'll have to examine the backtrace information printed after the abort to try to identify the transistors that need tagging (see Section 16).

11.1. Flow

The **flow** command allows you to restrict flow through named attributes, and has the form

flow *direction attribute attribute ...*

Direction must be one of **in**, **out**, **off**, **ignore**, or **normal**. If *direction* is **in** then Crystal treats each of the attributes as if it was an **In** attribute, and if *direction* is **out** then the attributes are treated as if they were **Out**. If **off** is specified then no flow is allowed through any transistors with the given attributes. If **ignore** is specified, Crystal will pretend that the attributes don't exist. If **normal** is given, the flow is reset to do the normal thing. All flow attributes are reset to **normal** by the **clear** command. The **flow** command has no effect on attributes **In** or **Out**.

12. Checking Commands

Two commands are provided by Crystal to perform a static electrical analysis of the circuit. They are only indirectly related to timing analysis, but are useful to find problems such as improper ratios, nodes that aren't marked as inputs, and transistors that should have flow attributes.

12.1. Check

The command

check

makes a series of static electrical checks on the circuit. It prints out information about nodes with no transistors connected to them, nodes that are not driven from anywhere, nodes that don't drive anything, transistors that are permanently forced off, and transistors connecting Vdd and GND directly. Each of these situations is probably an error. The **check** command also identifies transistors that are bidirectional (each side of the transistor has both a signal source and a signal target), but do not have any flow attributes attached. In most cases, bidirectional transistors should have flow attributes to keep Crystal from examining impossible paths.

12.2. Ratio

The **ratio** command has the form

ratio [*limit value*] [*limit value*] ...

and may be used for nMOS circuits to detect improper pullup/pulldown ratios. Normal logic gates are expected to have pullup/pulldown ratios between 3.8 and 4.2, while logic gates driven through pass transistors must have ratios between 7.8 and 8.2. Any ratios outside this range are printed out. If the same erroneous ratio occurs more than 20 times, only the first 20 are printed. The acceptable range may be changed using *limit-value* pairs. *Limit* is one of **normalhi**,

normallow, **passhi**, or **passlow**.

13. Multi-phase Signals, Memory Nodes, and Watched Nodes

Crystal treats clock phases in a very simple fashion: each clock phase is assumed to be long enough for the circuit to completely settle. The **critical** command indicates how long this takes. Although this approach will produce correct circuits, it is an overly pessimistic view of how clocks are used. In most clocked designs, some signals will settle over more than one clock phase. For example, the input latch for an ALU might be loaded during phase 1, and the output of the ALU might not be used until phase 2. In situations like this, Crystal will normally charge the ALU delay entirely to phase 1, leading to a pessimistic timing estimate.

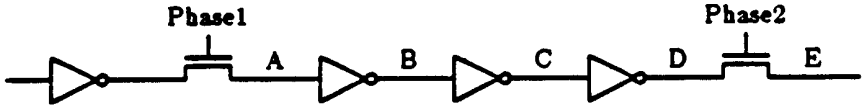

Figure 8. Because node **A** is a memory node, **Phase1** must be long enough for **A** to settle. Nodes **B**, **C**, and **D** need not settle during **Phase1**: they can settle anytime during **Phase1** or **Phase2**. However, if they don't settle during **Phase1**, enough time must be allowed during **Phase2** for them to settle and for the value at **E** to settle also.

For a circuit to function correctly, it isn't really necessary for everything to stabilize during each clock phase. All that matters is that clock phases are long enough for memory cells to be loaded correctly. This means that there can be some tradeoff between the lengths of the various clock phases: see Figure 8 for an example. Ideally, Crystal should deal only with memory nodes: when analyzing clock phase 1, Crystal should compute delays to memory cells loaded in phase 1, memory cells loaded in phase 2, and so on. Then, instead of outputting a single time and critical path, there would be separate times and critical paths for delays between the leading edge of phase 1 and the trailing edges of phase 1, phase 2, and so on. Unfortunately, Crystal doesn't provide this much detail. Instead, it uses memory nodes to provide a first-order approximation to this.

During the **delay** command, Crystal keeps three separate records of worst-case delays: one for all nodes, one just for memory nodes, and one for watched nodes. In the **critical** command, you can use the "m" suffix to print out memory nodes. For example, **critical 1m** will print out the path to the slowest memory node. This simple facility allows you to ignore signals that need not settle during the current clock phase. However, if a signal starts settling in one clock phase and is loaded into a memory cell in the next clock phase, Crystal will not check that the sum of the two phases is enough for this to happen safely. I suggest that you examine critical paths both for memory nodes and for all nodes: check to see that memory nodes will settle before the end of the current clock phase, and that all nodes will settle before the end of the next clock phase.

There are two kinds of memory nodes in a MOS circuit, static and dynamic (see Figure 9). Static memory nodes are those like cross-coupled NAND gates

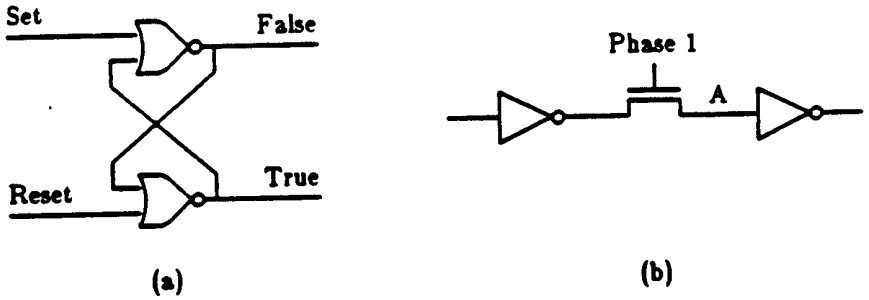

Figure 9. In (a), nodes **False** and **True** are static memory nodes. In (b), **A** is a dynamic memory node.

where there is an ever-present feedback path. Crystal detects such feedback paths during delay analysis and marks the memory nodes. However, Crystal cannot identify dynamic memory nodes without help from the user. At the beginning of analysis, you should use the **markdynamic** command to tell Crystal which nodes are dynamic memory. The command has the form

markdynamic *node value node value ...*

During the **markdynamic** command, Crystal sets each *node* to the given *value* just as if the **set** command had been used. Any nodes that are electrically isolated by these settings (i.e. all transistors connecting to them are forced off) are marked as dynamic memory. Normally, **markdynamic** is used by turning off all of the clock phases.

If Crystal's memory mechanism isn't discriminating enough to pick out all the important paths, there is one more mechanism available as a last resort. You can indicate certain nodes to be handled specially. These nodes are called "watched nodes" because you select them with the command

watch *node node ...*

A special third list of slow memory nodes will be used to record the slowest delays to watched nodes. This allows you to select key nodes and see the delays to those nodes, even if those delays aren't great enough to make the nodes appear on the overall list or the memory list. The danger of the watch mechanism is that it forces you to pick out the key nodes. If you forget a key node then you may end up missing the critical path. I recommend that you work as much as possible with the overall and memory lists, and only use the watch mechanism as a last resort.

14. The Models

Crystal's model of circuit behavior has two parts: one part is used to do logic simulation during the **set** command, and the other part is used to do delay calculations during the **delay** command. Both the simulation and delay models are based on transistor types: there are several types of transistors in the circuit, and each is parameterized by several values. The *man* page lists the predefined transistor types and the fields associated with each type. The subsections below tell how this information is used by Crystal, how to change the predefined

information, and how to define new transistor types.

14.1. Simulation

In order to do logic simulation, each type of transistor is given two integer strength values: **histrength** tells how strongly the transistor can pull to logic 1, and **lostrength** tells how strongly the transistor can pull to logic zero. The strength values are the same for all transistors of a given type, and are independent of the geometry of the transistor. For example, all nMOS enhancement transistors have a **lostrength** greater than the **histrength** of all nMOS depletion pullups.

During the **set** command, the nodes listed in the command are forced to a given value. Then Crystal sees if these settings cause any transistors to be forced on or off. If this happens, nodes on either side of the forced-on or forced-off transistors may be forced to a value. The strength values are used to see if this is the case. For a node to be forced to 1 in this way, two conditions must be met. First, there must be a path from the node to a source of logic level 1, all of whose transistors are forced on. Second, all paths from the node to sources of logic 0 must either contain a forced-off transistor or be weaker than the path to logic 1. The strength of a path is the strength of its weakest transistor.

This simple simulation model is powerful enough to handle a variety of nMOS and CMOS structures. Its weakness is that it doesn't take account of the sizes of transistors, so it may behave incorrectly if improper ratios are used.

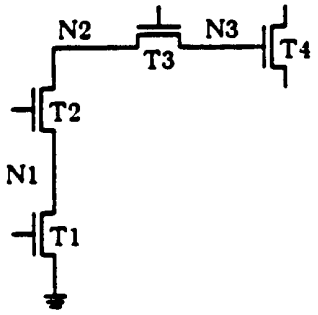

Figure 10. To calculate the delay along this path with transistor T2 as the trigger device, the resistances from T1, N1, T2, N2, T3, and N3 will be summed, and the capacitances from N2, T3, N3, and T4 will be summed. The delay will be the product of the two sums.

14.2. Delay Calculation: the RC Model

Crystal has been designed to include several different delay models and to permit the user to switch between them. At present, there are two delay models, **rc** and **slope**. In the rc model each transistor type is characterized by two resistances, **rup** and **rdown**. The transistor is assumed to have a fixed resistance value **rup** per square whenever it is used to transmit a 1 signal, and **rdown** per square whenever it is used to transmit a 0 signal.

To calculate the delay in a stage, the RC model divides the stage into two portions, separated by the transistor that turned on or off to activate the stage (this transistor is called the trigger for the stage). See Figure 10. All the resistances along the stage are summed, including **rup** or **rdown** for each transistor, plus the resistance of the interconnect. All the capacitances between the trigger and the target are also summed, including the gate-channel capacitance of each transistor along the stage, the parasitic capacitances of the interconnect, and the gate-source or gate-drain capacitances of unrelated transistors that connect to nodes along the stage. Crystal assumes that the trigger is the last transistor in the stage to turn on or off, so that all the charge between the trigger and the signal source has already been drained. The total delay for the stage is computed by multiplying the total capacitance by total resistance.

14.3. Delay Calculation: the Slope Model

The RC model is simple and efficient, but it often produces optimistic delay estimates. It assumes that the effective resistance of a transistor is independent of the waveform on the transistor's gate, and this simply isn't true in reality. If the gate voltage of a transistor rises or falls very slowly, the transistor has a much higher effective resistance than if the gate voltage changes instantaneously. The same transistor may vary in effective resistance by an order of magnitude or more, depending on the exact waveform on its input.

In the RC model, the waveform at a node is characterized solely by the time at which it rises or falls. In the slope model, an additional parameter is added: the rate at which the signal rises or falls. This is called the *edge speed*, and is measured in ns/volt at the instant in time when the signal crosses its logic threshold voltage (the logic threshold voltage is a model parameter and can be changed with the **parameter** command). Although this is only a first-order approximation to the actual waveforms, in Mead-Conway style digital circuits the waveforms tend to have about the same shape except for slope, so this characterization is fairly accurate.

The slope model characterizes the effective resistance of a particular type of transistor in terms of the ratio of two edge speeds: the input edge speed, and the output native edge speed. The output native edge speed is the edge speed that would occur on the output if the input rose or fell infinitely fast (edge speed 0). If the edge speed ratios are small (inputs much faster than output), or if they are uniform across the whole circuit, then the RC model is accurate.

Two tables are used to characterize each transistor type. One table is used when the transistor is pulling up, and the other is used when the transistor is pulling down (these are the **slopeparmsup** and **slopeparmsdown** fields in the transistor models). Each table consist of several triplets. Each triplet contains three values: an edge speed ratio, the transistor's effective resistance per square when that edge speed ratio occurs, and the output edge speed (per pf of capacitance driven and per square of transistor), when that edge speed ratio occurs. The table entries must be in increasing order of edge speed, and the first

entry must have a zero edge speed ratio. If Crystal ever encounters a ratio larger than the largest in the table, it issues a warning message and extrapolates from the largest values. To simplify the task of gathering all this model information, use the *Mkcp* ("make Crystal parameters") program.

Delay calculation in the slope model proceeds in much the same way as for the RC model, except that for the trigger transistor, Crystal interpolates in the tables to find the effective resistance. For transistors other than the trigger, the native resistance is used. In addition, the slope model computes an output edge speed contribution from each component along the path (transistor or node resistance), and sums these to compute the edge speed at the target. The edge speed contributions are computed for each componenet as if that component were driving the capacitance all by itself.

The slope model appears to be fairly accurate. Initial measurements suggest that it is usually within 5% of the times that SPICE predicts for the same circuits, and is rarely worse than 20% off. In contrast, the rc model often produces estimates that are optimistic by 40% or more. The slope model is almost as fast as the rc model, so there is little reason to use the rc model anymore, except for comparison.

14.4. Changing the Models

Crystal provides three commands that you can use to change its internal models. The command

model [*name*]

will set the current delay model to *name*, if it is specified. If *name* is omitted, then the command will print out the valid model names with two stars next to the current model.

The command

transistor [*name* [*field value(s)*] [*field value(s)*] ...]

is used to see and modify the values used to characterize each transistor. If **transistor** is invoked with no arguments, all the transistor types and their current values are printed. If only *name* is supplied with no fields or values, all the transistor type information for *name* is printed. Otherwise, fields for transistor type *name* are changed to the given values. The *man* page lists the predefined transistor types and the field names. If *name* isn't one of the predefined transistor types, then a new transistor type is created with the given field values.

The third command is used to see and set the model parameters that don't have to do with specific transistor types. At present, these parameters are used only for computing the parasitic resistance and capacitance of interconnect. The command has the form

parameter [*name*] [*value*]

If both *name* and *value* are specified, then the selected parameter is set to the given value. If *value* is omitted, then the value of the parameter is printed. If

neither *value* or *name* is given, then the values of all parameters for the current model are printed. See the *man* page for a listing of the parameter names.

14.5. Defining New Transistor Types

The **transistor** command can be used to define new transistor types besides the standard ones. To get Crystal to treat transistors in your circuit as one of the new ones you've defined, use transistor attributes. Normally, Crystal decides the type of each transistor based on its type in the .sim file (enhancement, depletion, p-channel, etc) and how it is used in the circuit. For example, depletion transistors with source or drain connected to Vdd and the other two terminals connected together are given type **nload**. If you want Crystal to use a type of your choosing for a transistor, place an attribute inside the gate area of the transistor. The name of the attribute will be taken by Crystal as the type of that transistor. For example, if you have defined a new transistor type **bootstrap**, then each of these devices should have an attribute **Cr:bootstrap$** on its gate.

15. Miscellaneous Commands

The **help** command prints out a list of the commands and their parameters. For information on the commands that is more detailed than **help**, and more concise than this document, see the *man* page.

The command

source *file*

will cause Crystal to read commands from *file* until its end is reached. Upon end-of-file, Crystal continues reading from the standard input. Source files may be nested.

The **options** command is provided so that you can change internal thresholds and switch settings used by Crystal. For example, one of the options is the threshold capacitance value at which Crystal automatically marks nodes as busses. Normal users shouldn't need to use this command very frequently. See the *man* page for details on its syntax and on the available options.

The **statistics** command prints out a variety of statistics gathered by Crystal as it runs. This information is probably not useful except to system maintainers.

The **quit** command causes Crystal to return to the shell.

16. Deciphering Crystal's messages

Crystal outputs a huge variety of error messages, bug messages and hints. Most of them are in response to syntax errors in the .sim file or errors in commands: these are relatively easy to understand. You should never see a message beginning with the words "Crystal bug:". If you do, report it to me or to your local Crystal wizard. There are several other messages whose meaning is not obvious. They generally indicate that something not-quite-right happened and

are hints that either you are not issuing the right commands or you need to use flow tags or **set** commands to restrict Crystal's analysis. Each of the following subsections describes one such message.

16.1. Aborting: no solution after examining 200000 stages

Crystal has a limit on how many stages it will examine in delay calculations. If the limit is reached, Crystal gives up in despair. When it gives up, it usually means that you need to add more flow control to pass transistors to restrict the set of paths Crystal has to analyze. Occasionally, the built-in limit isn't sufficient for a particular clock phase, even after all the necessary flow control has been added. In this case, use the **options** command to increase the limit.

When the limit is reached, Crystal outputs many messages, the first of which is the "Aborting:" message. Following this will be many messages of two forms: "ChaseVG giving up at xyz", and "ChaseGates giving up at abc". ChaseVG and ChaseGates are the two internal routines that trace out paths through the circuit during delay analysis. The messages indicate the path Crystal was examining when it gave up in despair, in backwards order from the node where it gave up to the node in the **delay** command. Often, the node names in the messages will identify the area where more flow control is needed.

If Crystal aborts a delay calculation, then the information in **critical** and similar commands may not be accurate, since the delay analysis wasn't completed. However, the path provided by **critical** may indicate the place where more flow control is needed. Another way to locate transistors that need flow tagging is to use the **check** command.

16.2. More than 8 transistors in series

During delay analaysis, if Crystal finds a single stage containing more than a certain number of transistors in series, it prints this message. The stage is also ignored (usually such stages cannot occur in practice anyway). A typical place where this might occur is in carry-chain precharging schemes where there are both parallel and serial paths to each node in the chain.

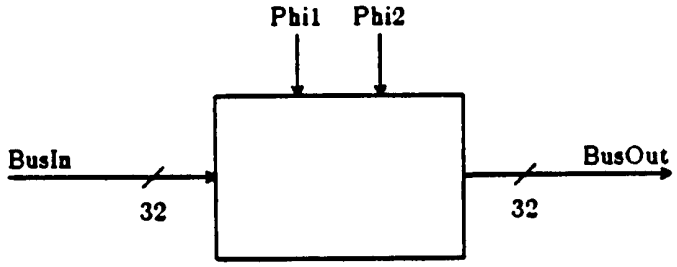

Figure 11. A simple circuit with two non-overlapping clock phases, 32 data inputs, and 32 data outputs.

17. An Example

For the circuit of Figure 11, the following Crystal commands might be used to do timing analysis, assuming that data is read into the circuit only during **Phi1** and that it stabilizes no later than 20ns into the clock cycle. The **BusIn** signals are unidirectional (if they could also be driven from on-chip then it would not be necessary to specify them in the **inputs** command). As a result of this set of commands, two Caesar command files will be created: phi1cmds and phi2cmds.

```
inputs BusIn<0:31> Phi1 Phi2
outputs BusOut<31:0>

set Phi2 0
delay Phi1 0 -1
delay BusIn<0:31> 20 20
critical -g phi1cmds

clear
set Phi1 0
delay Phi2 0 -1
critical -g phi2cmds
```