A.6 Built-In Chips

The hardware simulator features a library of built-in chips that can be used as internal parts by other chips. Built-in chips are implemented in code written in a programming language like Java, operating behind an HDL interface. Thus, a built-in chip has a standard HDL header (interface), but its HDL body (implementation) declares it as built-in. Figure A.2 gives a typical example.

The identifier following the keyword BUILTIN is the name of the program unit that implements the chip logic. The present version of the hardware simulator is built in Java, and all the built-in chips are implemented as compiled Java classes. Hence, the HDL body of a built-in chip has the following format:

```
BUILTIN Java class name;
```

where Java class name is the name of the Java class that delivers the chip functionality. Normally, this class will have the same name as that of the chip, for example Mux.class. All the built-in chips (compiled Java class files) are stored in a directory called tools/builtIn, which is a standard part of the simulator's environment.

Built-in chips provide three special services:

■ *Foundation:* Some chips are the atoms from which all other chips are built. In particular, we use Nand gates and flip-flop gates as the building blocks of all combinational and sequential chips, respectively. Thus the hardware simulator features built-in versions of Nand.hdl and DFF.hdl.

```
/** 16-bit Multiplexor.
If sel = 0 then out = a else out = b.
This chip has a built-in implementation delivered by an external
Java class. */
CHIP Mux16 {
   IN a[16], a[16], sel;
   OUT out[16];
   BUILTIN Mux;  // Reference to builtIn/Mux.class, that
                 // implements both the Mux.hdl and the
                 // Mux16.hdl built-in chips.
}
```

Figure A.2 HDL definition of a built-in chip.

■ *Certification and efficiency:* One way to modularize the development of a complex chip is to start by implementing built-in versions of its underlying chip parts. This enables the designer to build and test the chip logic while ignoring the logic of its lower-level parts—the simulator will automatically invoke their built-in implementations. Additionally, it makes sense to use built-in versions even for chips that were already constructed in HDL, since the former are typically much faster and more space-efficient than the latter (simulation-wise). For example, when you load RAM4k.hdl into the simulator, the simulator creates a memory-resident data structure consisting of thousands of lower-level chips, all the way down to the flip-flop gates at the bottom of the recursive chip hierarchy. Clearly, there is no need to repeat this drill-down simulation each time RAM4K is used as part in higher-level chips. Best practice tip: To boost performance and minimize errors, always use built-in versions of chips whenever they are available.

■ *Visualization:* Some high-level chips (e.g., memory units) are easier to understand and debug if their operation can be inspected visually. To facilitate this service, built-in chips can be endowed (by their implementer) with GUI side effects. This GUI is displayed whenever the chip is loaded into the simulator or invoked as a lower-level part by the loaded chip. Except for these visual side effects, GUI-EMPOWERED chips behave, and can be used, just like any other chip. Section A.8 provides more details about GUI-empowered chips.

A.7 Sequential Chips

Computer chips are either combinational or sequential (also called *clocked*). The operation of combinational chips is instantaneous. When a user or a test script changes the values of one or more of the input pins of a combinational chip and reevaluates it, the simulator responds by immediately setting the chip output pins to a new set of values, as computed by the chip logic. In contrast, the operation of sequential chips is clock-regulated. When the inputs of a sequential chip change, the outputs of the chip may change only at the beginning of the next time unit, as effected by the simulated clock.

In fact, sequential chips (e.g., those implementing counters) may change their output values when the time changes even if none of their inputs changed. In contrast, combinational chips never change their values just because of the progression of time.

A.7.1 The Clock

The simulator models the progression of time by supporting two operations called tick and tock. These operations can be used to simulate a series of time units, each consisting of two phases: a tick ends the first phase of a time unit and starts its second phase, and a tock signals the first phase of the next time unit. The real time that elapsed during this period is irrelevant for simulation purposes, since we have full control over the clock. In other words, either the simulator's user or a test script can issue ticks and tocks at will, causing the clock to generate series of simulated time units.

The two-phased time units regulate the operations of all the sequential chip parts in the simulated chip architecture, as follows. During the first phase of the time unit (*tick*), the inputs of each sequential chip in the architecture are read and affect the chip's internal state, according to the chip logic. During the second phase of the time unit (*tock*), the outputs of the chip are set to the new values. Hence, if we look at a sequential chip "from the outside," we see that its output pins stabilize to new values only at *tocks*—between consecutive time units.

There are two ways to control the simulated clock: manual and script-based. First, the simulator's GUI features a clock-shaped button. One click on this button (a tick) ends the first phase of the clock cycle, and a subsequent click (a tock) ends the second phase of the cycle, bringing on the first phase of the next cycle, and so on. Alternatively, one can run the clock from a test script, for example, using the command repeat n {tick, tock, output ; }. This particular example instructs the simulator to advance the clock n time units, and to print some values in the process. Test scripts and commands like repeat and output are described in detail in appendix B.

A.7.2 Clocked Chips and Pins

A built-in chip can declare its dependence on the clock explicitly, using the statement:

```
CLOCKED pin, pin, . . . , pin;
```

where each pin is one of the input or output pins declared in the chip header. The inclusion of an *input pin x* in the CLOCKED list instructs the simulator that changes to x should not affect any of the chip's output pins until the beginning of the next time unit. The inclusion of an output *pin x* in the CLOCKED list instructs the simulator that changes in any of the chip's input pins should not affect *x* until the beginning of the next time unit.

Note that it is quite possible that only some of the input or output pins of a chip are declared as clocked. In that case, changes in the nonclocked input pins may affect the nonclocked output pins in a combinational manner, namely, independent of the clock. In fact, it is also possible to have the CLOCKED keyword with an empty list of pins, signifying that even though the chip may change its internal state depending on the clock, changes to any of its input pins may cause immediate changes to any of its output pins.

The "Clocked" Property of Chips How does the simulator know that a given chip is clocked? If the chip is built-in, then its HDL code may include the keyword CLOCKED. If the chip is not built-in, then it is said to be clocked when one or more of its lower-level chip parts are clocked. This "clocked" property is checked recursively, all the way down the chip hierarchy, where a built-in chip may be explicitly clocked. If such a chip is found, it renders every chip that depends on it (up the hierarchy) implicitly clocked. It follows that nothing in the HDL code of a given chip suggests that it may be clocked—the only way to know for sure is to read the chip documentation. For example, let us consider how the built-in DFF chip (figure A.3) impacts the "clockedness" of some of other chips presented in the book.

Every sequential chip in our computer architecture depends in one way or another on (typically numerous) DFF chips. For example, the RAM64 chip is made of eight RAM8 chips. Each one of these chips is made of eight lower-level Register chips. Each one of these registers is made of sixteen Bit chips. And each one of these Bit chips contains a DFF part. It follows that Bit, Register, RAM8, RAM64 and all the memory units above them are also clocked chips.

```
/** D-Flip-Flop.
If load[t-1]=1 then out[t]=in[t-1] else out does not change. */
CHIP DFF {
  IN in;
  OUT out;
  BUILTIN DFF;        // Implemented by builtIn/DFF.class.
  CLOCKED in, out;    // Explicitly clocked.
}
```

Figure A.3 HDL definition of a clocked chip.

It’s important to remember that a sequential chip may well contain combinational logic that is not affected by the clock. For example, the structure of every sequential RAM chip includes combinational circuits that manage its addressing logic (described in chapter 3).

A.7.3 Feedback Loops

We say that the use of a chip entails a feedback loop when the output of one of its parts affects the input of the same part, either directly or through some (possibly long) path of dependencies. For example, consider the following two examples of direct feedback dependencies:

```
Not (in=loop1, out=loop1)    // Invalid
DFF (in=loop2, out=loop2)    // Valid
```

In each example, an internal pin (loop1 or loop2) attempts to feed the chip's input from its output, creating a cycle. The difference between the two examples is that Not is a combinational chip whereas DFF is clocked. In the Not example, loop1 creates an instantaneous and uncontrolled dependency between in and out, sometimes called data race. In the DFF case, the in-out dependency created by loop2 is delayed by the clocked logic of the DFF, and thus out (t) is not a function of in (t) but rather of in (t-1). In general, we have the following:

Valid/Invalid Feedback Loops When the simulator loads a chip, it checks recursively if its various connections entail feedback loops. For each loop, the simulator checks if the loop goes through a clocked pin, somewhere along the loop. If so, the loop is allowed. Otherwise, the simulator stops processing and issues an error message. This is done in order to avoid uncontrolled data races.