Appendix A: Hardware Description Language (HDL)[1]

Intelligence is the faculty of making artificial objects,
especially tools to make tools.

Henry Bergson (1859-1941)

A *Hardware Description Language* (HDL) is a formalism for defining and testing *chips*: objects whose interfaces consist of input and output pins that carry Boolean signals, and whose bodies are composed of interconnected collections of other, lower-level, chips. This appendix describes a typical HDL, as understood by the hardware simulator supplied with the book. Chapter 1 (in particular section 1.1) provides essential background without which this appendix does not make much sense.

How to use this appendix: This is a technical reference, and thus there is no need to read it from beginning to end. Instead, we recommended focusing on selected sections, as needed. Also, HDL is an intuitive and self-explanatory language, and the best way to learn it is to play with some HDL programs using the supplied hardware simulator. Therefore, we recommend to start experimenting with HDL programs as soon as you can, beginning with the following example.

A.1 Example

Figure A.1 specifies a chip that accepts two three-bit numbers and outputs whether they are equal or not. The chip logic uses Xor gates to compare the three bit-pairs, and outputs true if all the comparisons agree.

```
/** Checks if two 3-bit input buses are equal */
CHIP EQ3 {
   IN  a[3], b[3];
   OUT out;  // True iff a=b
   PARTS:
   Xor(a=a[0], b=b[0], out=c0);
   Xor(a=a[1], b=b[1], out=c1);
   Xor(a=a[2], b=b[2], out=c2);
   Or(a=c0, b=c1, out=c01);
   Or(a=c01, b=c2, out=neq);
   Not(in=neq, out=out);
}
```

FIGURE A.1: HDL Program example.

[1]From *The Elements of Computing Systems* by Nisan & Schocken (draft ed.), MIT Press, 2005, www.idc.ac.il/tecs

Each internal part Xxx invoked by an HDL program refers to a stand-alone chip defined in a separate `Xxx.hdl` program like the one listed above. Thus the chip designer who wrote the `EQ3.hdl` program assumed the availability of three other lower-level programs: `Xor.hdl`, `Or.hdl`, and `Not.hdl`. Importantly, though, the designer need not worry about *how* these chips are implemented. When building a new chip in HDL, the internal parts that participate in the design are always viewed as black boxes, allowing the designer to focus only on their proper arrangement in the current chip architecture.

Thanks to this modularity, all HDL programs, including those that describe high-level chips, can be kept short and readable. For example, a complex chip like RAM16K can be implemented using a few internal parts (e.g. RAM4K chips), each described in a single HDL line. When fully evaluated by the hardware simulator all the way down the recursive chip hierarchy, these internal parts are expanded into many thousands of interconnected elementary logic gates. Yet the chip designer need not be concerned by this complexity, and can focus instead only on the chip's topmost architecture.

A.2 Conventions

File Extension: Each chip is defined in a separate text file. A chip whose name is Xxx is defined in file `Xxx.hdl`.

Chip structure: A chip definition consists of a *header* and a *body*. The header specifies the chip *interface*, and the body its *implementation*. The header acts as the chip's API, or public documentation. The body should not interest people who use the chip as an internal part in other chip definitions.

Syntax conventions: HDL is case-sensitive. HDL keywords are written in uppercase letters.

Identifier naming: Names of chips and pins may be any sequence of letters and digits not starting with a digit. By convention, chip and pin names start with a capital letter and a lowercase letter, respectively. For readability, such names can include uppercase letters.

White space: Space characters, newline characters, and comments are ignored.

Comments: The following comment formats are supported:

```
//  Comment to end of line
/*  Comment until closing */
/** API documentation comment */
```

A.3 Loading Chips into the Hardware Simulator

HDL programs (chip descriptions) are loaded into the hardware simulator in three different ways. First, the user can open an HDL file interactively, via a "load file" menu or GUI icon. Second, a test script (discussed below) can include a `load Xxx.hdl` command, which has the same effect. Finally, whenever an HDL program is loaded and parsed, every chip name Xxx listed in it as an internal part causes the simulator to load the respective `Xxx.hdl` file, all the way down the recursive chip hierarchy. In every one of these cases, the simulator goes through the following logic:

```
if Xxx.hdl exists in the current directory
    then load it (and all its descendents) into the simulator
else
    if Xxx.hdl exists in the simulator's builtIn chips directory
        then load it (and all its descendents) into the simulator
    else
            issue an error message.
```

The simulator's `builtIn` directory contains executable versions of all the chips specified in the book, except for the highest-level chips (CPU, Memory, and Computer). Hence, one may construct and test every chip mentioned in the book before all, or even any, of its lower-level chip parts have been implemented: The simulator will automatically invoke their built-in versions instead. Likewise, if a lower-level chip Xxx has been implemented by the user in HDL, the user can still force the simulator to use its built-in version instead, by simply moving the `Xxx.hdl` file out from the current directory. Finally, in some cases the user (rather than the simulator) may want to load a built-in chip directly, e.g. for experimentation. To do so, simply navigate to the `tools/builtIn` directory – a standard part of the hardware simulator environment -- and select the desired chip from there.

A.4 Chip Header (Interface)

The header of an HDL program has the following format:

```
CHIP chip name {
  IN input pin name, input pin name, ... ;
  OUT output pin name, output pin name, ... ;
  // Here comes the body.
}
```

- **CHIP declaration:** The `CHIP` keyword is followed by the chip name. The rest of the HDL code appears between curly brackets.
- **Input pins**: The `IN` keyword is followed by a comma-separated list of input pin names. The list is terminated with a semicolon.
- **Output pins**: The `OUT` keyword is followed by a comma-separated list of output pin names. The list is terminated with a semicolon.

Input and output pins are assumed by default to be single-bit wide. A multi-bit *bus* can be declared using the notation *pin name*`[`*w*`]`. (e.g. `a[3]` in `EQ3.hdl`). This specifies that the pin is a bus of width *w*. The individual bits in a bus are indexed *0* ... *w*-1, from right to left (i.e. index 0 refers to the least significant bit).

A.5 Chip Body (Implementation)

A.5.1 Parts

A typical chip consists of several lower-level chips, connected to each other and to the chip input/output pins in a certain "logic" (connectivity pattern) designed to deliver the chip functionality. This logic, written by the HDL programmer, is described in the chip body using the format:

> `PARTS:`
> *internal chip part*`;`
> *internal chip part*`;`
> `. . .`
> *internal chip part*`;`

Where each *internal chip part* statement describes one internal chip with all its *connections*, using the syntax:

> *chip name*`(`*connection*`, ... ,` *connection*`);`

Where each *connection* is described using the syntax:

> *part's pin name = chip's pin name*

(Throughout this appendix, the presently defined chip is called *chip*, and the lower-level chips listed in the `PARTS` section are called *parts*).

A.5.2 Pins and Connections

Each *connection* describes how one pin of a part is connected to another pin in the chip definition. In the simplest case, the programmer connects a part's pin to an input or output pin of the chip. In other cases, a part's pin is connected to another pin of another part. This internal connection requires the introduction of an *internal pin*, as follows.

Internal pins: In order to connect an output pin of one part to the input pins of other parts, the HDL programmer can create and use an *internal pin*, say `v`, as follows:

```
Part1(..., out=v);     // out of Part1 is piped into v
Part2(in=v, ...);      // v is piped into in of Part2
Part3(a=v, b=v,...);   // v is piped into both a and b of Part3
```

Internal pins (like `v` above) are created as needed when they are specified the first time in the HDL program, and require no special declaration. Each internal pin has fan-in 1 and unlimited fan-out, meaning that it can be fed from a single source only, yet it can feed (through multiple connections) many other parts. In the above example, the internal pin `v` simultaneously feeds both `Part2` (through `in`) and `Part3` (though `a` and `b`).

Input pins: Each input pin of a part may be fed by one of the following sources:

- An input pin of the chip;
- An internal pin;
- One of the constants `true` and `false`, representing 1 and 0, respectively.

Each input pin has fan-in 1, meaning that it can be fed by one source only. Thus `Part(in1=v,in2=v,...)` is a valid statement, whereas `Part(in1=v,in1=u,...)` is not.

Output pins: Each output pin of a part may feed one of the following destinations:

- An output pin of the chip;
- An internal pin.

A.5.3 Buses

Each pin used in a connection -- whether input, output, or internal -- may be a *multi-bit bus*. The *widths* (number of bits) of input and output pins are defined in the chip header. The widths of internal pins are deduced implicitly, from their connections.

In order to connect individual elements of a multi-bit bus input or output pin, the pin name (say `x`) may be subscripted using the syntax `x[i]` or `x[i..j]=v`, where `v` is an internal pin. This means that only the bits indexed *i* to *j* (inclusive) of pin *x* are connected to the specified internal pin. An internal pin (like `v` above) may not be subscripted, and its width is deduced implicitly from the width of the bus pin to which it is connected the first time it is mentioned in the HDL program.

The constants `true` and `false` may also be used as buses, in which case the required width is deduced implicitly from the context of the connection.

Example:

```
CHIP Foo {
   IN in[8]   // 8-bit input
   OUT out[8] // 8-bit output
   // Foo’s body (irrelevant to the example)
}
```

Suppose now that `Foo` is invoked by another chip using the part statement:

```
Foo(in[2..4]=v, in[6..7]=true, out[0..3]=x, out[2..6]=y)
```

Where `v` is a previously declared 3-bit internal pin, bound to some value. In that case, the connections `in[2..4]=v` and `in[6..7]=true` will bind the `in` bus of the Foo chip to the following values:

`in:`

Bit:	7	6	5	4	3	2	1	0
Contents:	1	1	?	v[2]	v[1]	v[0]	?	?

Now, let us assume that the logic of the Foo chip returns the following output:

`out:`

Bit:	7	6	5	4	3	2	1	0
Contents:	1	1	0	1	0	0	1	1

In that case, the connections `out[0..3]=x` and `out[2..6]=y` will yield:

`x:`

Bit:	3	2	1	0
Contents:	0	0	1	1

`y:`

Bit:	4	3	2	1	0
Contents:	1	0	1	0	0

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.

```
/** 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: A built-in chip definition.

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`.
- **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 `RAM16K.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.

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, e.g. 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, i.e. 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.

```
 /** 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: A clocked chip definition.

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.

It's important to remember that a sequential chip may well contain combinational logic which 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.

A.8 Visualizing Chip Operations

Built-in chips may be “GUI-empowered.” These chips feature visual side effects, designed to animate chip operations. A GUI-empowered chip can come to play in a simulation in two different ways, just like any other chip. First, the user can load it directly into the simulator. Second, and more typically, whenever a GUI-empowered chip is used as a part in the simulated chip, the simulator invokes it automatically. In both cases, the simulator displays the chip’s graphical image on the screen. Using this image, which is typically an interactive GUI component, one may inspect the current contents of the chip as well as change its internal state, when this operation is supported by the built-in chip implementation.

The present version of the simulator features the following set of GUI-empowered chips:

ALU: Displays the Hack ALU’s inputs and output as well as the presently computed function.

Registers (There are three of them: ARegister -- address register, DRegister -- data register, and PC -- program counter): Displays the contents of the register and allows modifying its contents.

Memory chips (ROM32K and various RAM chips): Displays a scrollable array-like image that shows the contents of all the memory locations, and allows modifying them. If the contents of a memory location changes during the simulation, the respective entry in the GUI changes as well. In the case of the `ROM32K` chip (which serves as the instruction memory of our computer platform), the GUI also features a button that enables loading a machine language program from an external text file.

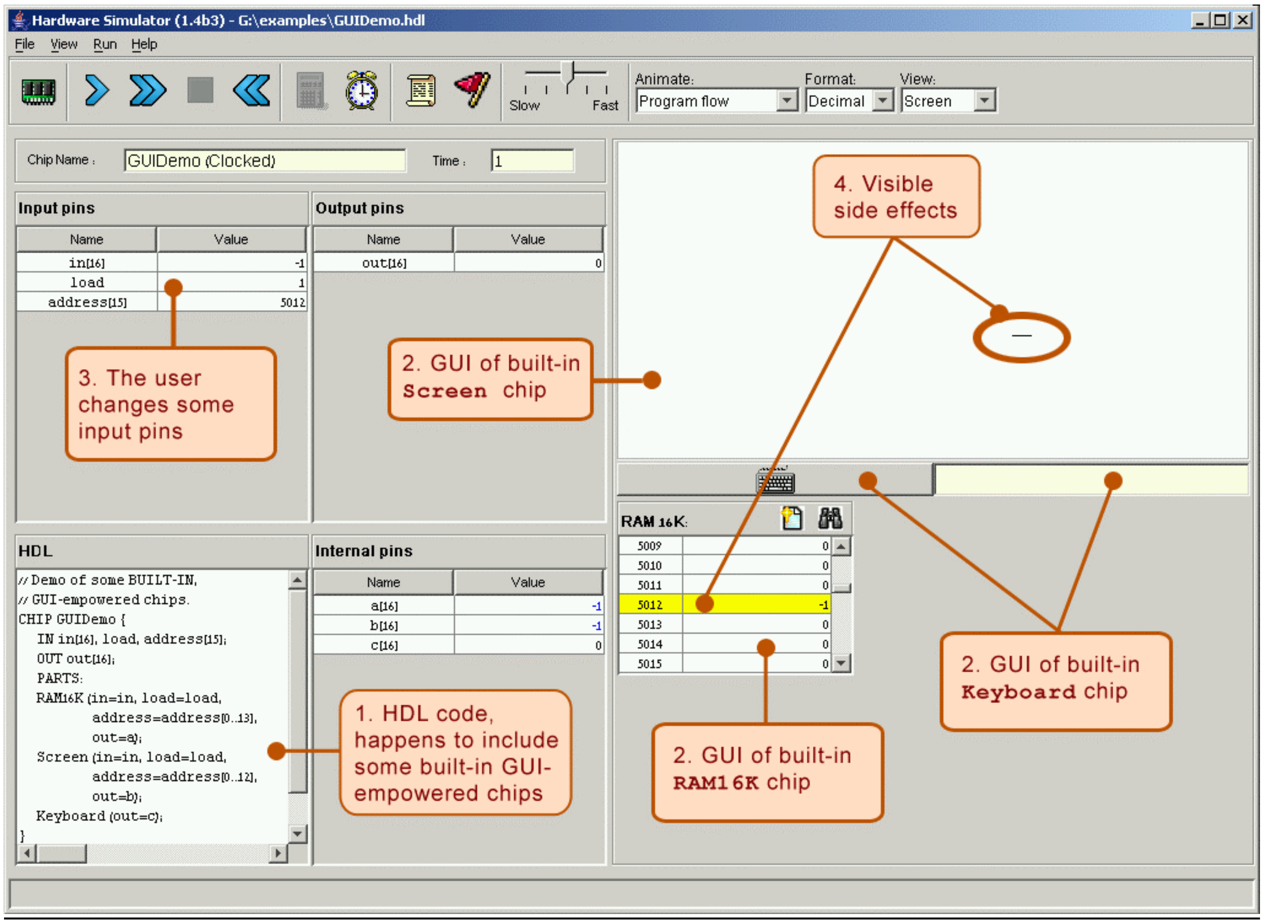
Screen chip: If the HDL code of a loaded chip invokes the built-in Screen chip, the hardware simulator displays a 256 rows by 512 columns window that simulates the physical screen. When the RAM-resident memory-map of the screen changes during the simulation, the respective pixels in the screen GUI change as well, via a “refresh logic” embedded in the simulator implementation.

Keyboard chip: If the HDL code of a loaded chip invokes the built-in Keyboard chip, the simulator displays a clickable keyboard icon. Clicking this button connects the real keyboard of your computer to the simulated chip. From this point on, every key pressed on the real keyboard is intercepted by the simulated chip, and its binary code is displayed in the keyboard’s RAM-resident memory-map. If the user moves the mouse focus to another area in the simulator GUI, the control of the keyboard is restored to the real computer. Figure A.3 illustrates many of the features just described.

```
// Demo of GUI-empowered chips.
// The logic of this chip is meaningless, and is used merely to
// force the simulator to display the GUI effects of some other chips.
CHIP GUIDemo {
  IN in[16], load, address[15];
  OUT out[16];
  PARTS:
  RAM16K(in=in, load=load, address=address[0..13], out=a);
  Screen(in=in, load=load, address=address[0..12], out=b);
  Keyboard(out=c);
}
```

FIGURE A.4: A GUI-empowered chip.

The chip logic in figure A.4 feeds the 16-bit in value into two destinations: register number *address* in the RAM16K chip and register number *address* in the Screen chip (presumably, the HDL programmer who wrote this code has figured out the widths of these address pins from the documentation of these chips). In addition, the chip logic routes the value of the currently pressed keyboard key to the internal pin c. These meaningless operations are designed for one purpose only: illustrating how the simulator deals with built-in GUI-empowered chips. The actual impact is shown in figure A.5.


Hardware Simulator (1.4b3) - G:\examples\GUIDemo.hdl
File View Run Help
Slow Fast
4. Visible side effects
3. The user changes some input pins
2. GUI of built-in Screen chip
1. HDL code, happens to include some built-in GUI-empowered chips
2. GUI of built-in RAM16K chip
2. GUI of built-in Keyboard chip
RAM 16K:

Animate: Program flow Format: Decimal View: Screen

Chip Name : GUIDemo (Clocked) Time : 1

Input pins

Name	Value
in[16]	-1
load	1
address[15]	5012

Output pins

Name	Value
out[16]	0

HDL

```
// Demo of some BUILT-IN,
// GUI-empowered chips.
CHIP GUIDemo {
    IN in[16], load, address[15];
    OUT out[16];
    PARTS:
    RAM16K (in=in, load=load,
            address=address[0..13],
            out=a);
    Screen (in=in, load=load,
            address=address[0..12],
            out=b);
    Keyboard (out=c);
}
```

Internal pins

Name	Value
a[16]	-1
b[16]	-1
c[16]	0

RAM 16K:

5009	0
5010	0
5011	0
5012	-1
5013	0
5014	0
5015	0

FIGURE A.5: GUI-empowered Chips. Since the loaded HDL program uses GUI-empowered chips as internal parts (step 1), the simulator draws their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4). The circled horizontal line is the visual side effect of storing –1 in memory location 5012. Since the 16-bit 2's complement binary code of –1 is 1111111111111111, the computer draws 16 pixels starting at the 320th column of row 156, which happen to be the screen coordinates associated with address 5012 of the screen memory map (the exact memory-to-screen mapping is given in Chapter 4).

A.9 Supplied and New Built-In Chips

The built-in chips supplied with the hardware simulator are listed in figure A.6. These Java-based chip implementations were designed to support the construction and simulation of the Hack computer platform (although some of them can be used to support other 16-bit platforms). Users who wish to develop hardware platforms other than Hack would probably benefit from the simulator's ability to accommodate new built-in chip definitions.

Developing user-level built-in chips: The hardware simulator can execute any desired chip logic written in HDL; the ability to execute new built-in chips (in addition to those listed in figure A.6) written in Java is also possible, using a chip-extension API. Built-in chip implementations can be designed by users in Java to add new hardware components, introduce GUI effects, speed-up execution, and facilitate behavioral simulation of chips that are not yet developed in HDL (an important capability when designing new hardware platforms and related hardware construction projects). For more information about developing user-level built-in chips, see chapter 13.

Chip name	Specified in chapter	Has GUI	Comment
`Nand`	1		Foundation of all combinational chips
`Not`	1		
`And`	1		
`Or`	1		
`Xor`	1		
`Mux`	1		
`DMux`	1		
`Not16`	1		
`And16`	1		
`Or16`	1		
`Mux16`	1		
`Or8way`	1		
`Mux4way16`	1		
`Mux8way16`	1		
`DMux4way`	1		
`DMux8way`	1		
`HalfAdder`	2		
`FullAdder`	2		
`Add16`	2		
`ALU`	2	☑	
`Inc16`	2		
`DFF`	3		Foundation of all sequential chips
`Bit`	3		
`Register`	3		
`ARegister`	3	☑	Identical operation to `Register`, with GUI
`DRegister`	3	☑	Identical operation to `Register`, with GUI
`RAM8`	3	☑	
`RAM64`	3	☑	
`RAM512`	3	☑	
`RAM4K`	3	☑	
`RAM16K`	3	☑	
`PC`	3	☑	Program counter
`ROM32K`	5	☑	GUI allows loading a program from a text file
`Screen`	5	☑	GUI connects to a simulated screen
`Keyboard`	5	☑	GUI connects to the actual keyboard

FIGURE A.6: All the built-in chips supplied with the present version of the Hardware Simulator. A built-in chip has an HDL interface but is implemented as an executable Java class.