



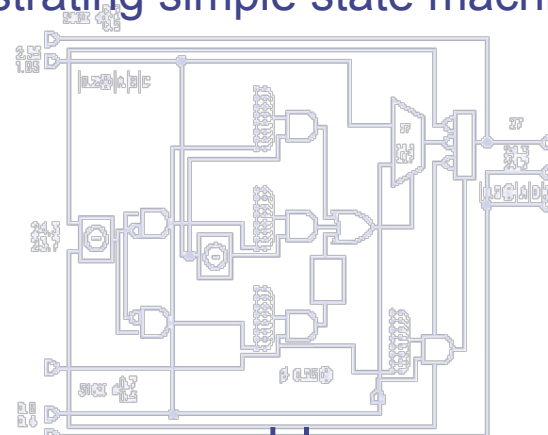
BSV Training

Eg02: Warmup exercise; simple state machines

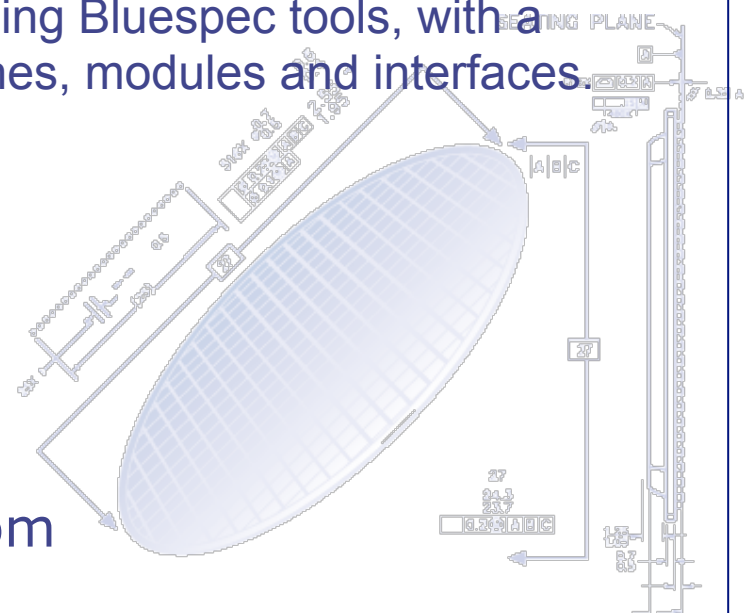
Introduction to “look and feel” of BSV and using Bluespec tools, with a simple example illustrating simple state machines, modules and interfaces.

```

import PIPCore
typedef BitA(24) (out0);
module ex_hdl_csrR_in[empty];
  Integer nfa_depth = 16;
  function BitA() distmpha_pump(stateTnd);
    return (out0);
  endfunction
  PIPCore(stateT) in0out0;
  out0out0(PIPCore(nfa_depth)) the_in0out0(in0out0);
  PIPCore(stateT) out0out0;
  out0out0(PIPCore(nfa_depth)) the_out0out0(out0out0);
  PIPCore(stateT) out0out0;
  out0out0(PIPCore(nfa_depth)) the_out0out0(out0out0);
  rule exp1 (true);
    stateT in_data = in0out0(nfa);
    PIPCore(stateT) out_data =
      distmpha_pump(nfa_data) = 0 ? out0out0 : out0out0;
    out_dataout0(nfa_data) =
      in0out0;
  endrule;
endmodule : ex_hdl_csrR_in
  
```



www.bluespec.com



Eg02a: A “Hello World” example

Ever since the classic book “The C Programming Language” by Kernighan and Ritchie in 1978, it has become customary to start with a very simple example to introduce the student to the basic “look and feel” of a language, and to get familiar with basic logistics: how programs texts are organized into files, how to compile them, execute them, and observe results. We shall do the same with BSV.

Our first BSV program just prints “Hello World!” (and a little more!) and halts.

The source code is in Example_Programs/Eg02a_HelloWorld/src_BSV/Testbench.bsv

BSV programs are organized into *modules*.

This program has one module only (“mkTestbench”).

All BSV modules have *interfaces*.

(This interface is the “Empty” *interface type*, which has no “methods” to interact with the environment.)

```
module mkTestbench (Empty);  
  
  rule rl_print_answer;  
    $display ("Deep Thought says: Hello, World! The answer is 42.");  
    $finish;  
  endrule  
endmodule
```

All behavior in BSV is expressed using rules. This program has one rule (“rl_print_answer”). A rule is a potentially infinite process, i.e., it may “fire” (execute) repeatedly, forever.

\$display is like “printf” in C/C++ (but it also always prints a final newline after the given output).

\$finish is like “exit()” in C/C++—it causes the whole program to halt immediately. Thus, in this example, the rule only fires once.

Compiling and running: Bluesim

Compiling, linking, and running is similar to compiling, linking and running a C/C++ program. Here, we show this using the “Bluesim” simulator.

The code can be found in: Example_Programs/Eg02a_HelloWorld/src_BSV/Testbench.bsv

You can type the commands as shown below. Or, for your convenience, there is also a Makefile in the Build/ directory, and you can invoke the commands by typing ‘make compile’, ‘make link’ and ‘make simulate’, respectively.

```
$ bsc -sim -g mkTestbench Testbench.bsv
checking package dependencies
compiling Testbench.bsv
code generation for mkTestbench starts
Elaborated module file created: mkTestbench.ba
All packages are up to date.
```

Or: `$ make compile`

```
$ bsc -sim -e mkTestbench -o ./mkTestbench_bsim
Bluesim object reused: mkTestbench.{h,o}
Bluesim object created: model_mkTestbench.{h,o}
Simulation shared library created:
mkTestbench_bsim.so
Simulation executable created: ./mkTestbench_bsim
```

Or: `$ make link`

```
$ ./mkTestbench_bsim
Deep Thought says: Hello, World! The answer is 42.
```

Or: `$ make simulate`

“bsc” is the Bluespec BSV compiler.

-sim: compile for Bluesim simulator

-g mkTestBench: top-level module

Testbench.bsv: source file

“bsc” is also the Bluespec BSV linker

-sim: link for Bluesim simulator

-e mkTestBench: top-level module

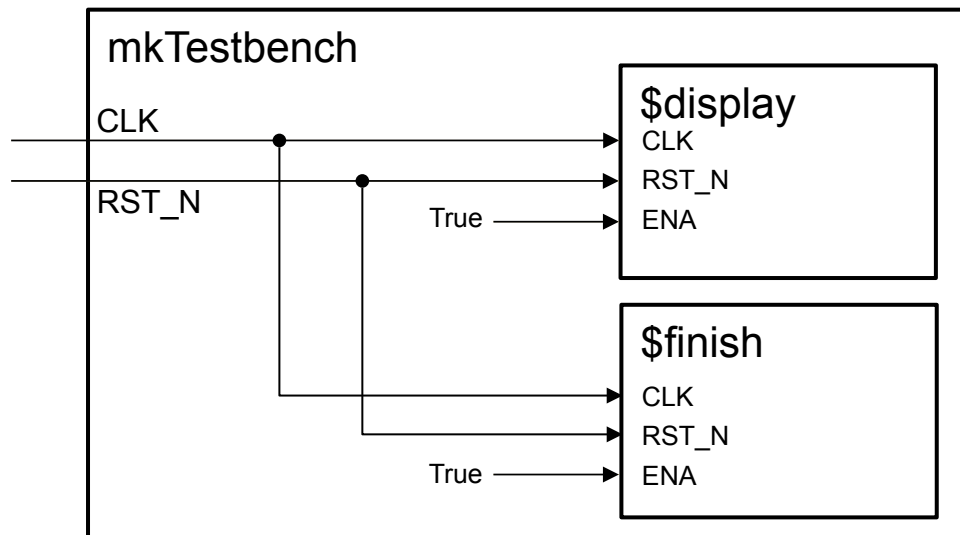
-o ./mkTestbench_bsim: name of output executable file

./mkTestbench_bsim: run the Bluesim executable like any executable.

“Deep ...”: \$display output is displayed on your screen.

A “hardware” view of our example

There is not much visible hardware in this example, since all the magic is in the \$display and \$finish primitives.



- The hardware generated by bsc is a *hierarchy of module instances* (module instances nested inside module instances).
- Every module has (at least) a CLK (clock) and RST_N (reset) input, and may have other inputs.
 - The environment asserts low (0, False) on RST_N for a short period after power is initially applied to the design
 - The environment oscillates CLK between low (0, False) and high (1, True) forever
- \$display and \$finish can be thought of as primitive modules that perform their action when the ENA signal is asserted. In this example, the ENA inputs are driven with the constant True.
 - Although \$display and \$finish are just used in simulation, one could actually build hardware modules that exhibit the same behavior

Compiling and running: Verilog simulation

Here, we show this using a Verilog simulator.

```
$ bsc -verilog -g mkTestbench Testbench.bsv
Verilog file created: mkTestbench.v
```

-verilog: generate verilog file ("mkTestbench.v")

-g mkTestBench: top-level module

Testbench.bsv: source file

Or: `$ make verilog`

```
$ bsc -verilog -e mkTestbench -o mkTestbench_vsim -vsim iverilog mkTestbench.v
Verilog binary file created: mkTestbench_vsim
```

-verilog: link for Verilog simulator

-e mkTestbench: top-level module

-o mkTestbench_vsim: name of output executable file

-vsim iverilog: use "iVerilog" Verilog simulator.

Alternatives: "modelsim" (Mentor), "ncverilog" (Cadence), "vcs" and "vcsi" (Synopsys), "cver" and "cvc" (Tachyon), "veriwel", "isim" (Xilinx)

Or: `$ make v_link`

```
$ ./mkTestbench_vsim
Deep Thought says: Hello, World! The answer is 42.
```

./mkTestbench_vsim: run the Verilog simulation executable like any executable.

"Deep ...": \$display output is displayed on your screen.

Or: `$ make v_simulate`

Synthesizing for FPGA or ASIC

(We won't actually do this, for this very simple example.)

The first step is the same as for Verilog simulation: generate Verilog files:

```
% bsc -verilog -g mkTestbench Testbench.bsv  
Verilog file created: mkTestbench.v
```

These Verilog files are then synthesized just like any other Verilog files using the synthesis tool of the target technology's vendor:

- ASIC: Design Compiler (Synopsys) or other vendor's RTL synthesis tool
- FPGA: Xilinx Vivado, Altera Quartus, or other vendor's RTL synthesis tool

Please consult the vendor's tools and training for details on how to do this.

Example variations

The supplied code includes three variations:

Eg02a_HelloWorld/	First version (previous slides)
Eg02b_HelloWorld/	Splits the first version into two separately compiled modules: "Testbench.bsv" and "DeepThought.bsv"
Eg02c_HelloWorld/	Adds some "state machine" functionality so that DeepThought "thinks for 7.5 million years" before yielding its answer ¹ , while the testbench waits. This will give a first view of rule conditions and method conditions.

We will now go through Eg02b and Eg02c.

¹You may have recognized that we are alluding to the book *The Hitchhiker's Guide to the Galaxy* by Douglas Adams (1979). In the book, a supercomputer named Deep Thought is asked to calculate the Answer to the Ultimate Question of Life, the Universe, and Everything. After 7.5 million years, it answers: "42".

Eg02b: Splitting into separately compiled modules

The code can be found in: `Example_Programs/Eg02b_HelloWorld/`

<code>Eg02a_HelloWorld/</code>	First version (previous slides)
<code>Eg02b_HelloWorld/</code>	Splits the first version into two separately compiled modules: "Testbench.bsv" and "DeepThought.bsv"
<code>Eg02c_HelloWorld/</code>	Adds some "state machine" functionality so that DeepThought "thinks for 7.5 million years" before yielding its answer ¹ , while the testbench waits. This will give a first view of rule conditions and method conditions.

Eg02b: Splitting into separately compiled modules

We split our previous program into two modules, a top-level ``testbench`` module that instantiates a ``design`` module. We define an interface for the design module, containing one ``method``. The testbench module invokes this method in the interface to interact with the design.

Each file is a separate package.
The filename must be ``packagename.bsv``
Here, package Testbench imports everything
defined in package DeepThought.

```
package Testbench;

import DeepThought :: *;

(* synthesize *)
module mkTestbench (Empty);

    DeepThought_IFC deepThought <- mkDeepThought;

    rule rl_print_answer;
    let x <- deepThought.getAnswer;
    $display ("Deep Thought says: Hello, World! The answer is %0d.", x);
    $finish;
    endrule
endmodule
```

Invoking an ActionValue method

Top-level module
creates an instance of
subordinate module

```
package DeepThought;

// Interface declaration

interface DeepThought_IFC;
    method ActionValue #(int)  getAnswer;
endinterface

// Module definition

(* synthesize *)
module mkDeepThought (DeepThought_IFC);

    method ActionValue#(int) getAnswer;
        return 42;
    endmethod
endmodule

endpackage
```

``synthesize`` tells bsc to
preserve this module
boundary when
generating Verilog (else it
would in-line it).

Compiling and running Eg02b: Bluesim

The code can be found in: Example_Programs/Eg02b_HelloWorld/

It contains a Makefile and some sub-directories.

The source code is in src_BSV/Testbench.bsv and src_BSV/DeepThought.bsv

The remaining sub-directories are initially empty; bsc will produce files in these directories.

```
$ bsc -u -sim -simdir build_bsim -bdir build_bsim -info-dir build_bsim
-keep-fires -aggressive-conditions -p ../src_BSV:~/Prelude:~/Libraries -
g mkTestbench src_BSV/Testbench.bsv
checking package dependencies
compiling ./src_BSV/DeepThought.bsv
code generation for mkDeepThought starts
Elaborated module file created: build_bsim/mkDeepThought.ba
compiling src_BSV/Testbench.bsv
code generation for mkTestbench starts
Elaborated module file created: build_bsim/mkTestbench.ba
All packages are up to date.
```

Only the top-level file and module need be mentioned; bsc will follow the ``import'' links and recompile whatever is needed

Or: `$ make compile`

```
$ bsc -e mkTestbench -sim -o ./mkTestbench_bsim -simdir build_bsim -bdir
build_bsim -info-dir build_bsim -p ../src_BSV:~/Prelude:~/Libraries
Bluesim object created: build_bsim/mkTestbench.{h,o}
Bluesim object created: build_bsim/mkDeepThought.{h,o}
Bluesim object created: build_bsim/model_mkTestbench.{h,o}
Simulation shared library created: mkTestbench_bsim.so
Simulation executable created: ./mkTestbench_bsim
```

Code generation and linking of all the modules for Bluesim.

Or: `$ make link`

```
% ./mkTestbench_bsim
Deep Thought says: Hello, World! The answer is 42.
```

Or: `$ make simulate`

Compiling Eg02b into Verilog

Here, we show this using a Verilog simulator.

```
bsc -u -verilog -vdir verilog -bdir build_v -info-dir build_v -elab -  
keep-fires -aggressive-conditions -no-warn-action-shadowing -p ../  
src_BSV:~/Prelude:~/Libraries -g mkTestbench src_BSV/Testbench.bsv  
checking package dependencies  
compiling ./src_BSV/DeepThought.bsv  
code generation for mkDeepThought starts  
Verilog file created: verilog/mkDeepThought.v  
Elaborated module file created: build_v/mkDeepThought.ba  
compiling src_BSV/Testbench.bsv  
code generation for mkTestbench starts  
Verilog file created: verilog/mkTestbench.v  
Elaborated module file created: build_v/mkTestbench.ba  
All packages are up to date.  
Compiling for Verilog finished
```

Creates separate Verilog modules (each in its own “.v” file, for each BSV module that had the “`synthesize” attribute.

Or: `$ make verilog`

You can of course link and simulate this in a Verilog simulator, as shown earlier for Eg02a.

In practice we mostly use Bluesim simulation, because it is much faster (10x-50x) and it has exactly the same cycle behavior as the corresponding Verilog simulation.

We typically generate Verilog only when we are ready to take it through post-RTL synthesis for ASIC or FPGA.

Eg02c: Adding some “state machine” functionality

The code can be found in: `Example_Programs/Eg02c_HelloWorld/`

Eg02a_HelloWorld/	First version (previous slides)
Eg02b_HelloWorld/	Splits the first version into two separately compiled modules: “Testbench.bsv” and “DeepThought.bsv”
Eg02c_HelloWorld/	Adds some “state machine” functionality so that DeepThought “thinks for 7.5 million years” before yielding its answer ¹ , while the testbench waits. This will give a first view of rule conditions and method conditions.

Eg02c: Adding some “state machine” functionality

Adds some “state machine” functionality so that DeepThought “thinks for 7.5 million years” before yielding its answer¹, while the testbench waits. This will give a first view of rule conditions and method conditions.

```
module mkTestbench (Empty);  
  
    DeepThought_IFC deepThought <- mkDeepThought;  
  
    rule rl_ask;  
        $display ("Asking the Ultimate Question of Life, The Universe and Everything");  
        deepThought.whatIsTheAnswer;  
    endrule  
  
    rule rl_print_answer;  
        let x <- deepThought.getAnswer;  
        $display ("Deep Thought says: Hello, World! the answer is %0d.", x);  
        $finish;  
    endrule  
endmodule
```

Rule “rl_ask” invokes the method “whatIsTheAnswer” to start a computation in the mkDeepThought module instance.

Some time later, rule “rl_print_answer” is able to invoke the method “getAnswer” and print the result.

```
// Interface definition  
  
interface DeepThought_IFC;  
    method Action whatIsTheAnswer;  
    method ActionValue #(int) getAnswer;  
endinterface  
  
// Module definition  
  
(* synthesize *)  
module mkDeepThought (DeepThought_IFC);  
  
    ... to be shown on next slides ...  
  
endmodule
```

Eg02c: Adding some "state machine" functionality

```
typedef enum { IDLE, THINKING, ANSWER_READY } State_DT  
deriving (Eq, Bits, FShow);
```

Define a type State_DT. The module will start in the IDLE state, move to THINKING, then to ANSWER_READY, and finally back to IDLE.

```
module mkDeepThought (DeepThought_IFC);
```

```
  Reg #(State_DT) rg_state_dt <- mkReg (IDLE);
```

Instantiate a register (variable) to hold the module state, initialized to IDLE

```
  Reg #(Bit #(4)) rg_half_millenia <- mkReg (0);
```

Instantiate register to count half-millenia

```
  let millenia = rg_half_millenia [3:1];
```

Define some useful values

```
  let half_millenum = rg_half_millenia [0];
```

```
  rule rl_think (rg_state_dt == THINKING);
```

Rule can fire whenever in THINKING state

```
    $write ("      DeepThought: ... thinking ... (%0d", millenia);
```

```
    if (half_millenum == 1) $write (".5");
```

```
    $display (" million years)");
```

Print the passing of the millenia

```
    if (rg_half_millenia == 15)
```

```
      rg_state_dt <= ANSWER_READY;
```

If seven and a half millenia, move to ANSWER_READY state

```
    else
```

```
      rg_half_millenia <= rg_half_millenia + 1;
```

else increment half millenia

```
  endrule
```

```
  method Action whatIsTheAnswer if (rg_state_dt == IDLE);
```

```
    rg_state_dt <= THINKING;
```

```
  endmethod
```

This method can be invoked when IDLE; then, move to THINKING state

```
  method ActionValue#(int) getAnswer if (rg_state_dt == ANSWER_READY);
```

```
    rg_state_dt <= IDLE;
```

```
    rg_half_millenia <= 0;
```

```
    return 42;
```

This method can be invoked when ANSWER_READY; then, return 42 and move to IDLE state

```
  endmethod
```

```
endmodule
```

Compiling and running Eg02c: Bluesim

The code can be found in: Example_Programs/Eg02c_HelloWorld/

```
$ make compile link
Compiling for Bluesim ...
bsc -u ...          as before

Compiling for Bluesim finished
Linking for Bluesim ...
bsc -e ...          as before

Linking for Bluesim finished
```

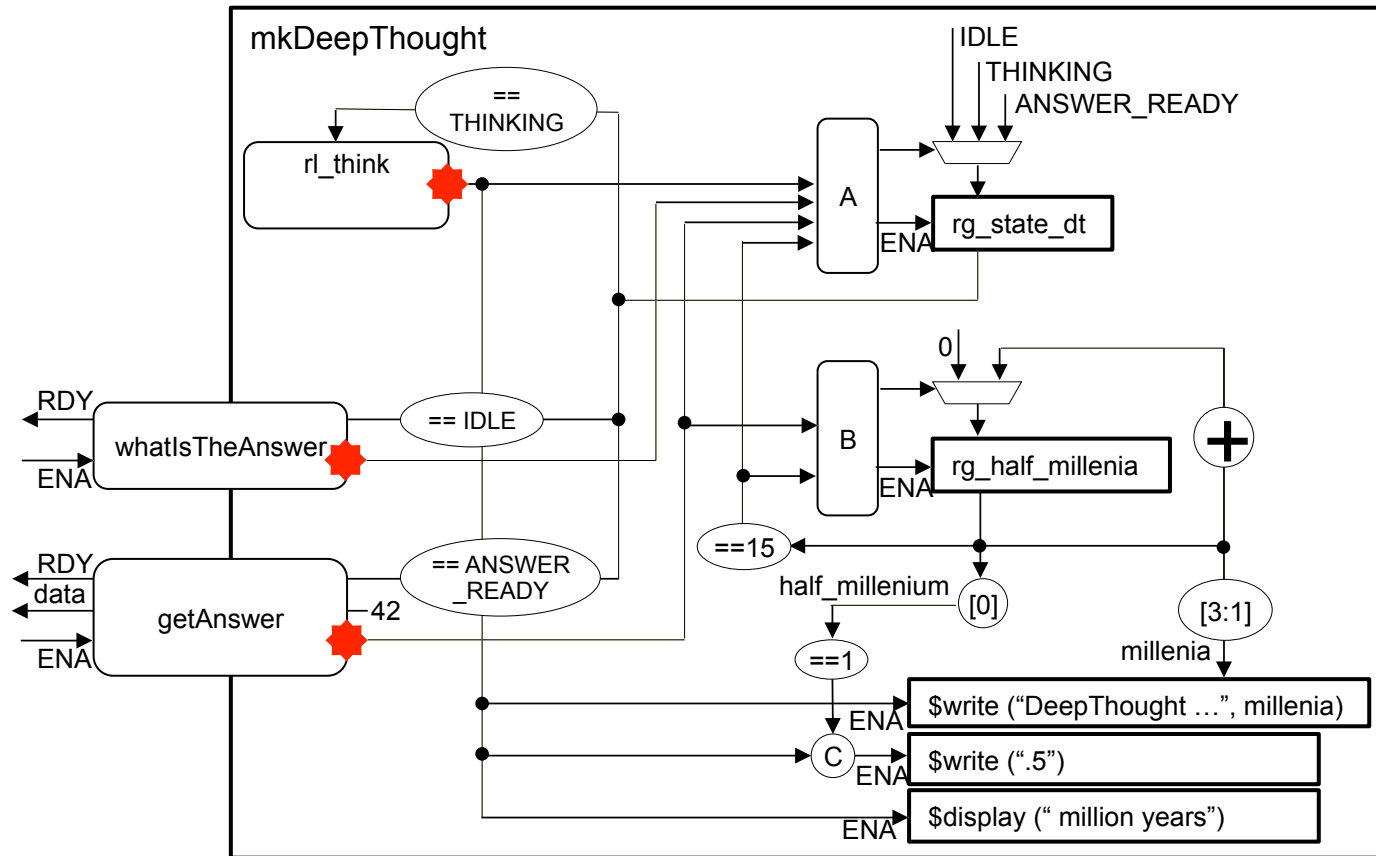
```
$ ./mkTestbench_bsim
Asking the Ultimate Question of Life, The Universe and Everything
    DeepThought: ... thinking ... (0 million years)
    DeepThought: ... thinking ... (0.5 million years)
    DeepThought: ... thinking ... (1 million years)
    DeepThought: ... thinking ... (1.5 million years)
    DeepThought: ... thinking ... (2 million years)
    DeepThought: ... thinking ... (2.5 million years)
    DeepThought: ... thinking ... (3 million years)
    DeepThought: ... thinking ... (3.5 million years)
    DeepThought: ... thinking ... (4 million years)
    DeepThought: ... thinking ... (4.5 million years)
    DeepThought: ... thinking ... (5 million years)
    DeepThought: ... thinking ... (5.5 million years)
    DeepThought: ... thinking ... (6 million years)
    DeepThought: ... thinking ... (6.5 million years)
    DeepThought: ... thinking ... (7 million years)
    DeepThought: ... thinking ... (7.5 million years)
Deep Thought says: Hello, World! The answer is 42.
```

From rule mkTestbench/rl_ask

From repeated firings of rule
mkDeepThought/rl_think

From rule mkTestbench/rl_print_answer

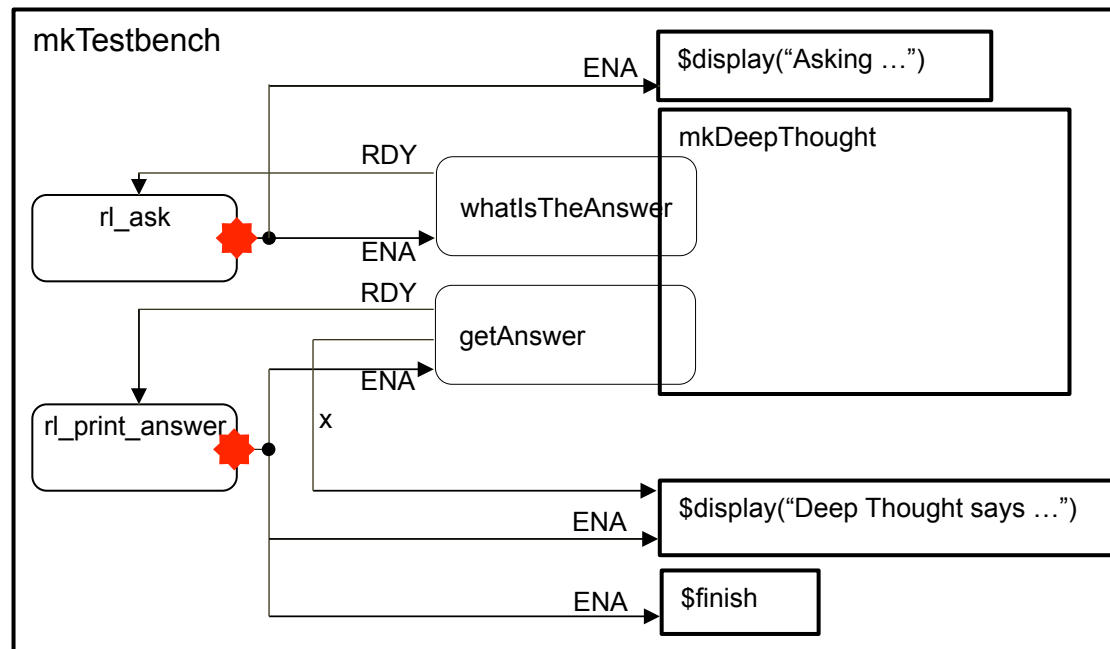
Hardware for Eg02c mkDeepThought



★ = "WILL_FIRE" signal of a rule/method (for a method, same as ENA)

A: controls `rg_state_dt`: selects input data (mux) and whether it is updated (ENA)
 B: controls `rg_half_millenia`: selects input data (mux) and whether it is updated (ENA)
 C: controls `$write` (ENA)
 In each case its output is a simple boolean combination of its inputs

Hardware for Eg02c mkTestbench



★ = "WILL_FIRE" signal of a rule

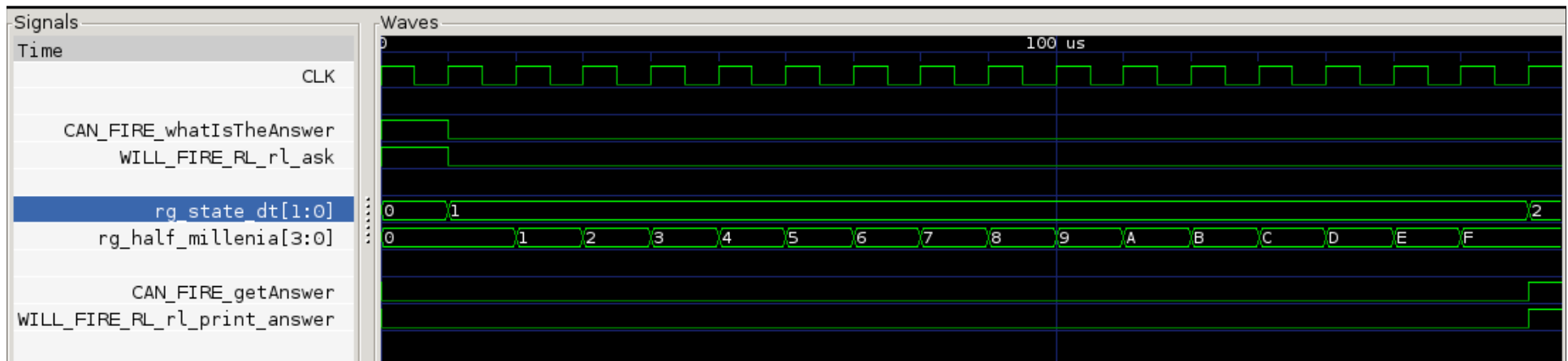
Waveforms from the circuit

```
% ./mkTestbench_bsim -V
Deep Thought says: Hello, World! The answer is 42.
```

-V: tells Bluesim simulation to dump waveforms from the circuit into “dump.vcd” file. Verilog simulators also have commands to capture VCDs. Note: you’ll get the same waveform whether from Bluesim or from Verilog sim.

```
% gtkwave dump.vcd
```

Displays the waves using “gtkwave” (you can use any convenient waveform viewer).



- The first wave shows the clock signal for the circuit (CLK)
- rg_state can be seen transitioning from 0 (IDLE) to 1 (THINKING) to 2 (ANSWER_READY)
- CAN_FIRE_whatIsTheAnswer shows that the method is enabled on the first clock, and WILL_FIRE_RL_rl_ask shows that the rule fires, invoking the method
- When rg_state is THINKING, rg_millenia and rg_half_millenia can be seen counting up. When they reach 7 and 1, respectively, rg_state transitions to ANSWER_READY (last clock)
- Then, CAN_FIRE_getAnswer is enabled, and WILL_FIRE_RL_rl_print_answer shows that the rule fires, invoking the method

Suggested exercises

In this and future examples, we suggest extra exercises to deepen your understanding of BSV

- In Eg02a, in rule `rl_print_answer`, exchange the two actions (`$display` and `$finish`). Is there any difference in behavior?
- In Eg02c, use the Makefile and build and run a Verilog simulation. Notice the “+bscvcd” flag in the `v_simulate` action. This causes a “dump.vcd” file to be created, just like when you gave the “-V” flag to Bluesim. View this in a waveform viewer and check that it has the same cycle behavior as Bluesim.
- Examine the generated Verilog files `mkTestbench.v` and `mkDeepThought.v` (in the “verilog/” directory).
 - Look at the input and output ports, and understand how they correspond to the BSV interface `DeepThought_IFC` and its methods.
 - Skim the interior of the Verilog module, and notice correspondences with the BSV source module (registers, rules, rule and method conditions, ...).
- In Eg02c, in module `mkDeepThought`, change the initial value of `rg_state_dt` from `IDLE` to `THINKING` and re-run the program. Change the initial value to `ANSWER_READY` and re-run. Discuss the behaviors.
- In the waveforms we saw that `IDLE`, `THINKING` and `ANSWER_READY` were encoded as 0, 1 and 2 respectively. Change the initial value of `rg_state_dt` from `IDLE` to 4, and try re-compiling. Discuss.

End

```

export PkPQst:
  typeof Kint(PQ) = BoolT;

  module of_int_out2_int2:typint;

  Integer Pk1_depth = 33;

  function Kint(PQ) distdepth_gauss(BoolT):
    return (pk1);
  endfunction;

  PkPQst(BoolT) initialize:
    pk1 = PkPQst(Pk1_depth) the_initialize_initialize;
    PkPQst(BoolT) out_bound:
      pk1 = PkPQst(Pk1_depth) the_outbound_initialize;
    PkPQst(BoolT) out_bound:
      pk1 = PkPQst(Pk1_depth) the_outbound_initialize;
  end PkPQst(BoolT) initialize;

  fun eval (True):
    BoolT in_state = initialize_init;
    PkPQst(BoolT) out_gauss =
      distdepth_gauss(in_state) = 0 ? out_bound : out_bound;
    out_initialize(in_state);
    initialize;
  end eval;

  endmodule; in of_int_out2_int

```

