

The Elevator Lab

"The Elevator Pitch"

This lab will lead you through slightly more advanced topics of the Bluespec System Verilog language and the usage of the Bluespec compiler, Bluesim simulator and the Bluespec development workstation. In each part you will find a description of the actual code you will have to generate and how to use and invoke each of the previously mentioned tools.

Note 1: *The directory $BLUESPECDIR/../doc/BSV contains very useful documentation that includes the Language Reference Guide, User Guide, Known Problems and Solutions and Style Guide among others.*
In the BLUESPECDIR/../training/BSV/examples directory you will find some useful examples for future reference.
The full set of training materials, documentation, and examples can be accessed from the file $BLUESPECDIR/index.html.

Note 2: *you might want to run simulations of the generated Verilog code using your favorite Verilog simulator.*

bsc -e <topName> -vsim <simulator> <verilogFile>

You provide the name of the simulator after the -vsim option. Currently the natively supported simulors are: vcs, vcsi, ncverilog, modelsim, cver, iverilog, and veriwell.

The Elevator System

A block diagram of a full elevator system is shown in the next figure:

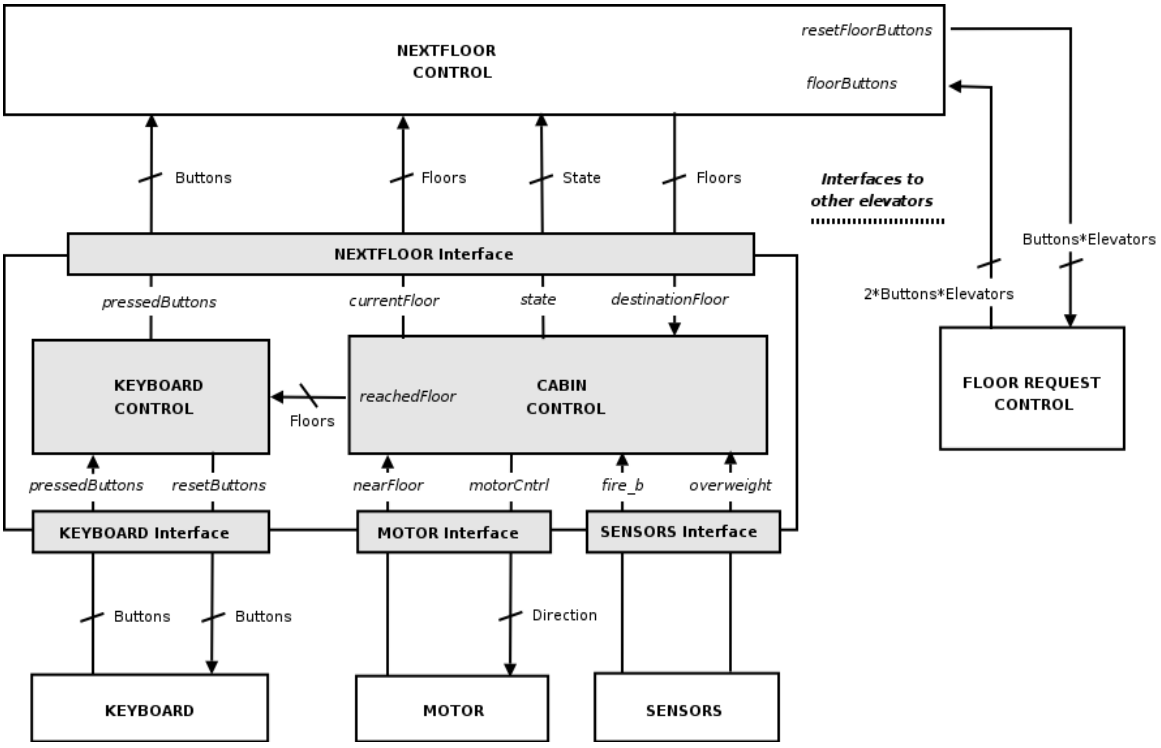

Figure 1 – Elevator System

In this lab we will focus in the CABIN SYSTEM, which comprises the CABIN CONTROL module and the KEYBOARD CONTROL module, grayed parts in the figure. The CABIN SYSTEM interfaces to the external NEXTFLOOR CONTROL, MOTOR, KEYBOARD and SENSORS modules which will be actually part of out testbenches.

CABIN SYSTEM Interfaces explanation:

- KEYBOARD interface:
  - *pressedButtons* is a bus whose width is the number of floors in the building. Each line in the bus indicates if the correspondent button has been pressed in the cabin's keyboard. Each bit in the button bus is sticky, i.e. once is set, it will remain in this state until it is reset by the corresponding *resetButtons* signal.

- *resetButtons* is a bus whose width is the number of floors in the building. Each line in the bus indicates if the correspondent button has to be reset in the cabin's keyboard

- CABIN CONTROL to KEYBOARD CONTROL interface
  - *reachedFloor* is a bus that indicates the reached floor. Its width is the minimum needed to index all the floors in the building. This signal is used by the KEYBOARD CONTROL to generate the *resetButtons* signal

In order to simplify our exercise let's consider the case of a building with 10 floors from 0 up to 9. Therefore we could define the Types Buttons as Bit#(10) and Floors type as Bit#(4).

- NEXTFLOOR CONTROL interface
  - *pressedButtons* indicates the buttons that are requested in the cabin.
  - *currentFloor* indicates the last floor to which the elevator arrived
  - *destinationFloor* indicates the floor to which NEXTFLOOR CONTROL commands the elevator to proceed
  - *state* is an output indicating the current elevator state. We will define the *CabinState* space in the set {Up, Down, Stopped}.

- MOTOR interface
  - *motorCntrl* is a bus that commands the motor. For simplicity we can define the Direction type as the *CabinState*, meaning:
    Stopped the motor does not work
    Up the motor is moving upwards
    Down, the motor is moving downwards
  - *nearFloor* indicates that a floor has been detected. It will be used by CABIN CONTROL to count floors and stop the cabin.

- SENSORS interfaces
  - *fire_n* indicates a fire situation whenever it is False
  - *overweight* indicates too much load for the cabin whenever is True

The basic functionality of the CABIN CONTROL and the KEYBOARD CONTROL will be explained in the next paragraphs.

Part 1

In this part you will learn:
- How to use user defined types
- Deriving
- How to describe an FSM using rules and methods
- How to reduce latency with RWire
- always_enabled and always_ready attributes

A) Create a project

Note: This step is only required when using the development workstation.

- Create a new project in the development workstation.
- Review the Project Options. Make sure the editor is set to your preferred text editor (Editor tab) and that Compile to is set to Verilog (Compile Tab). You will specify the Top File later.

B) Programming the design

When using the development workstation, you can create files from the File menu in the Project Files window.

- Create a file cabinCntrl.bsv that contains the interface *IfcCabinCntrl*:

```
interface IfcCabinCntrl#(type floors, type direction);
  // Interface with NEXTFLOOR CONTROL
  method Action      destinationFloor(floors a);

  // Interface with the MOTOR
  method Action      nearFloor;
  method direction   motorCntrl;
endinterface
```

- Declare the type enumerated *CabinState* with the next values: Up, Down and Stopped. The *cabinState* register will be of type *CabinState*.[1]
- Declare the type *Direction* as an alias of *CabinState*. this type will be used as the returned type by the *motorCntrl* method.

[1] When creating user defined types as "**typedef enum**", "**typedef struct** " and "**typedef union tagged**", you will need to use **deriving(Bits, Eq)**, meaning to the compiler to apply the obvious definition for pack/unpack and relation operators to the defined type.

- As said before, let's imagine our building has 10 floors from 0 up to 9, therefore we will need 4 bits in order to index the floors. Declare the alias type *Floors* as Bit#(4).
- Create a module *mkCabinController* using the *IfcCabinCntrl* to which you will pass the *Floors* and *Direction* types.

- Create the necessary rules and methods in order to describe the next behavior[2]:

  1) Initially the cabin will be on the floor 0 and stopped, i.e. the register *cabinControl* will be reset to Stopped and the *currentFloor* register will be reset to 0.
  2) Lets assume the first requested floor wins and that when in motion, the cabin control does not take into account subsequent requests. This could be summarized as:
     "If a new floor different than *currentFloor* is requested through the *destinationFloor* method and the cabin is *stopped,* then update the *requestedFloor* register".
  3) Once a new *requestedFloor* has been registered, the cabin is still stopped. If the *currentFloor* register content is greater than the *destinationFloor* then the cabin status will be changed to Down, otherwise Up. In this moment the cabin control also calculates the number of floors the elevator will have to move and store it in the *floorCounter* register.
  4) The *motorCntrl* method will output in any moment the direction of the cabin: Stopped, Up or Down.
  5) If the cabin is moving and *nearFloor* is detected, i.e. the method *nearFloor* gets externally invoked, the cabin control decrements the number of remaining floors *floorCounter* and recalculates *currentFloor*. After having moved the number of needed floors, *cabinState* register will change to Stopped.

C) Once you have programmed all the above statements in BSV, proceed as follows:

- Check that ALL of the above described actions are mapped in terms of rules and methods. This will be a proof of functional correctness.
- Compile your design for a Verilog back end, until you obtain a clean compilation. Please make sure you set the (* synthesize *) attribute on top of your module.
  - If compiling from the command line, use the -verilog flag.
  - If using the workstation, make sure that Verilog is set on the Compile tab in the Project Options window. Also make sure you've defined the top file on

[2] This set of conditions is complete, i.e. by making a direct translation of each point into BSV the behavior of the cabin control will be fully contained.

the Files tab.

- After you obtain a clean compilation, view the schedule information; pay special attention to rule and method conflicts.
  - If working from the command line, recompile your design with -show-schedule flag to view the schedule information.
  - If using the workstation, the Schedule Analysis window and the scheduling graphs show multiple views of the schedule information.
- Resolve ALL the conflicts in a proper way in order to obtain the intended hardware.

D) The method *motorCntrl* is designed in a way that it is always driving an output in each cycle.

- Take a look at the generated Verilog file.
  In the workstation Project Options ->Files tab, you can set which files to display in the Project Files window
- Set the attribute *(* synthesize, always_enabled = "motorCntrl", always_ready = "motorCntrl" *)* just above the mkCabinCntrl module.
- Recompile and inspect the Verilog code generated. What has changed? What happens if you set these attributes for another method that you know are not always_enabled and/or not always_ready?

E) You can run a simulation by using the provided testbench cabinCntrlTest.bsv.
In the workstation you'll need to change the top file to cabinCntrTest.bsv and set the top module to mkTest and rebuild.

F) Improving the performance of the design

- Use RWire in order to eliminate the cycle of latency between the Action method *destinationFloor* and the state change from Stopped to Up or Down. The register *requestedFloor* will still be required, you will need to instantiate an RWire that sets the value in the method *requestedFloor* and reads it in the rule that runs while in Stopped.

Part 2

In this part you will learn:

- sub-interfaces declaration and usage
- descending_urgency attribute
- How to reduce latency with Wire
- use of fire_when_enabled and no_implicit_conditions

A) Programming the design

- Copy your design files in a new directory called Part2
- Create two interfaces as follows:

```
interface MOTOR#(type direction);
        method Action       nearFloor;
        method direction    motorCntrl;
endinterface
interface SENSORS;
        method Action       fire_n (Bool b);
        method Action       overweight (Bool b);
endinterface
```

- We can now redefine our interface IfcCabinCntrl as:

```
interface IfcCabinCntrl#(type floors, type direction);
        // Interface with NEXTFLOOR CONTROL
        method Action               destinationFloor(floors a);
        method floors               inFloor;
        method direction            state;

        // Interface with the motor
        interface MOTOR#(direction) ifcMotor;

        // Interface with Sensors
        interface SENSORS           ifcSensors;

        // Interface from CABIN CONTROL to KEYBOARD CONTROL
        method floors               reachedFloor;
endinterface
```

- Modify the module mkCabinCtrnl in order to define the interface *ifcMotor*.

Basically you will need to enclose the related methods that you already have between interface/endinterface. For instance, for MOTOR:

```
interface MOTOR ifcMotor;
        // Method nearFloor functionality
        method Action       nearFloor if (...);
                ....
        endmethod

        // Method motorCntrl functionallity
        method Direction    motorCntrl;
                ...
        endmethod
endinterface.
```

- Create the method *reachedFloor* which will output the *destinationFloor* when it has been reached.
- Create the methods *inFloor* and *state* which will output the *currentFloor* and the *cabinState* respectively.
- Define the *ifcSensors* interface functionality as it was done for MOTOR. In this case in addition you will need to create the new methods used by the interface. The methods of the interface SENSORS will introduce the next modifications:

    1. if *fire_n* is False, Stop the cabin immediately and reset *destinationFloor* and *floorCounter* registers. Remain in Stop until *fire_n* is True again.
    2. if *cabinState* is Stopped and *overweight* is True then remain Stopped until *overweight* becomes False again regardless on any requested floor.
    3. A fire situation must be treated as high priority one

B) Repeat points B and D as mentioned in Part1. Use fire_when_enabled and no_implicit_conditions in your rules to make sure these assertions will be met for your chosen rules.

C) Improving the performance of the design

- Use Wire in order to eliminate the cycle of latency between the Action methods *fire_n* and *overweight* and the state change to Stopped.

Part 3

In this part you will learn:

- Connecting Modules
- Returning interfaces

A) Programming the design

- Copy your design files in a new directory called Part3
- Create a file keyboardCntrl.bsv that contains the interface IfcKeyboardCntrl, and the KEYBOARD interface:

```
interface KEYBOARD#(type buttons);
        method Action    pressedButtons (buttons  bs);
        method buttons   resetButtons_n;
endinterface

interface IfcKeyboardCntrl#(type buttons, type floors);
        // interface with the CABIN CONTROL
        method Action  reachedFloor(floors value);

        // Interface with keyboard
        interface KEYBOARD#(buttons)  ifcKeyboard;

        // Interface with the NEXTFLOOR CONTROL
        method buttons  pressedButtons;
endinterface
```

- Create the module mkKeyboardCntrl which will use the interface IfcKeyboardCntrl, with type *Buttons* as an alias of Bit#(10) and *Floors* as a synonym of Bit#(4). Remember Deriving operators on type definitions.
- Define the method *reachedFloor* to store the value of the floor reached in the reachedFloor register.
- Create the methods of ifcKeyboard as follows:
  1. Method *pressedButtons* simply takes the input and stores it in the *requestedFloors* register
  2. Method *resetButtons_n*: if the *requestedFloor* line corresponding to the *reachedFloor* register is set, then produce a pulse for one clock period in the corresponding bit of *resetButtons_n*.
- Create the method *pressedButtons* that simply outputs the *requestedFloors* register. Consider the possibility of using RWire in order to remove the latency.

B) Repeat points B and D as mentioned in Part1.
C) Creating the Cabin System

- Create a file called cabinSystem.bsv
- Define the interfaces ifcNextFloor and IfcCabinSystem as follows:

```
interface NEXTFLOOR#(type buttons, type floors);
        method buttons      pressedButtons;
        method Action       destinationFloor(floors a);
        method floors       currentFloor;
        method CabinState   state;
endinterface

interface IfcCabinSystem#(type buttons, type floors, type direction);
        // interface with the NEXTFLOOR CONTROL
        interface NEXTFLOOR#(buttons, floors)   ifcNextFloor;

        // Interface with the motor
        interface MOTOR#(direction)             ifcMotor;

        // Interface with Sensors
        interface SENSORS                       ifcSensors;

        // Interface with keyboard
        interface KEYBOARD#(buttons)            ifcKeyboard;
endinterface
```

- Create a module mkCabinSystem which uses IfcCabinSystem interface with types Buttons as Bit#(10), *Floor* as Bit#(4) and *Direction* as enum of {Up, Down, Stopped}.
- Instantiate the modules mkCabinCntrl and mkKeyboardCntrl with their respective interfaces and parameters.
- Create a rule that connects the *reachedFloor* methods of both modules.
- Define the rest of the interfaces as direct connections to the corresponding interfaces in the mkCabinCntrl and mkKeyboardCntrl module instantiation.