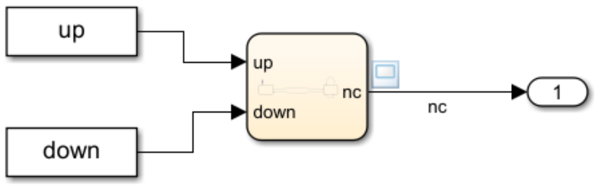
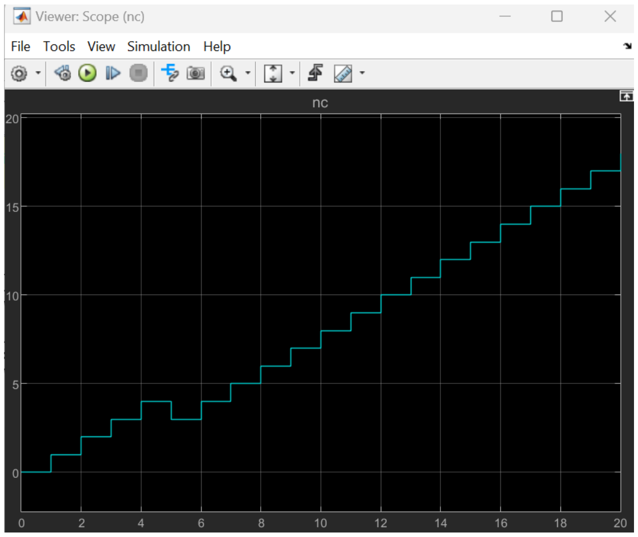
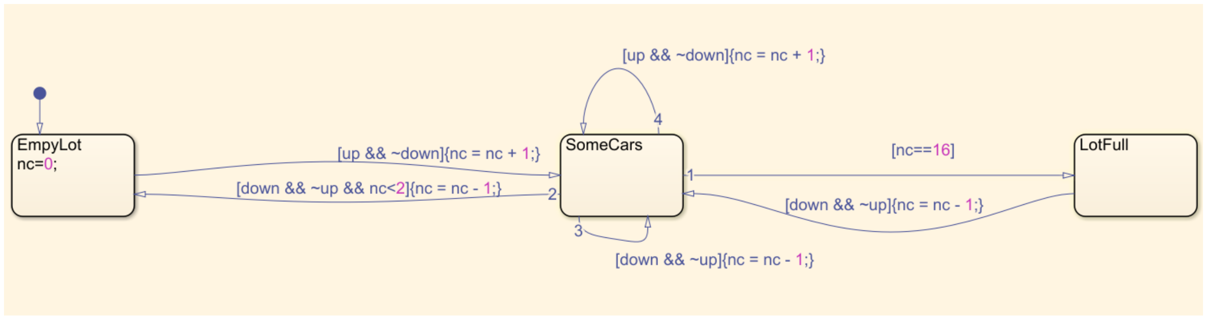
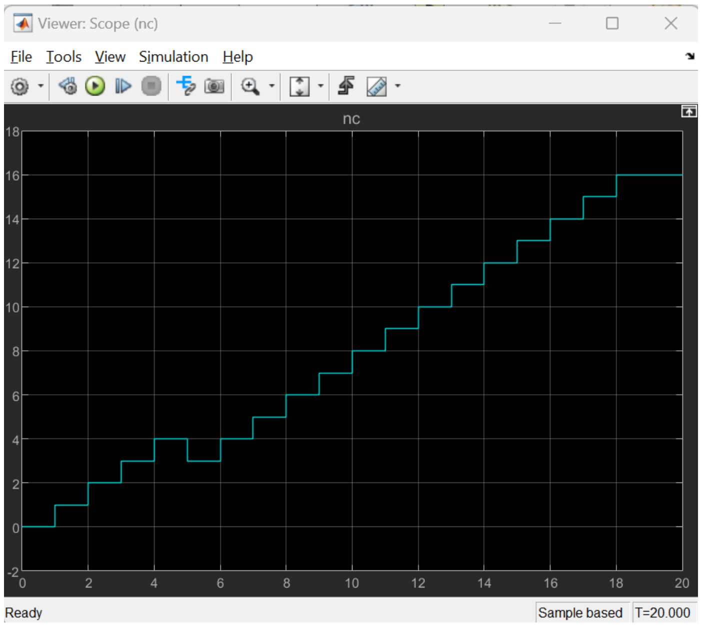
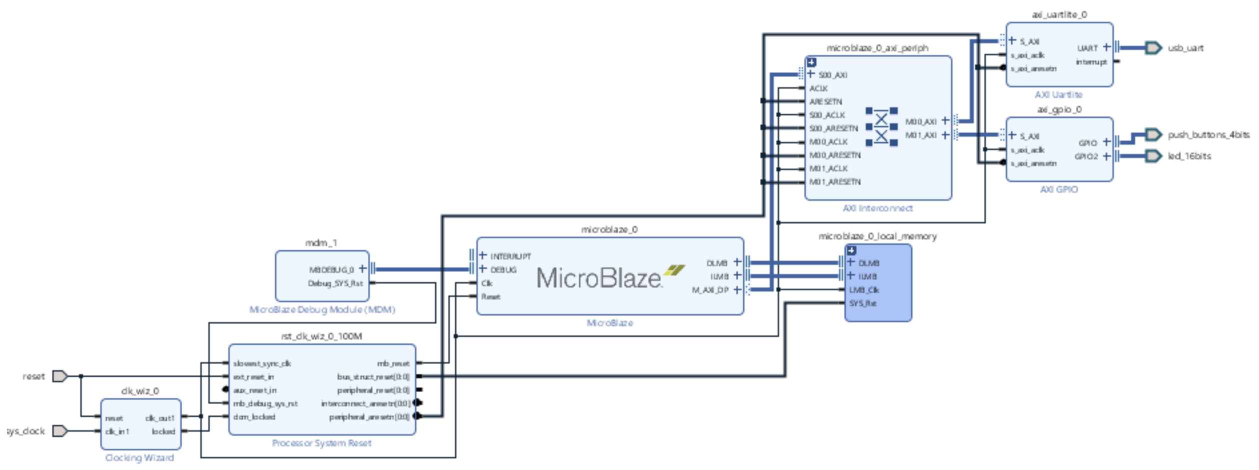
Lab 3: Getting Started with Stateflow and Peripheral Interfacing with GPIO

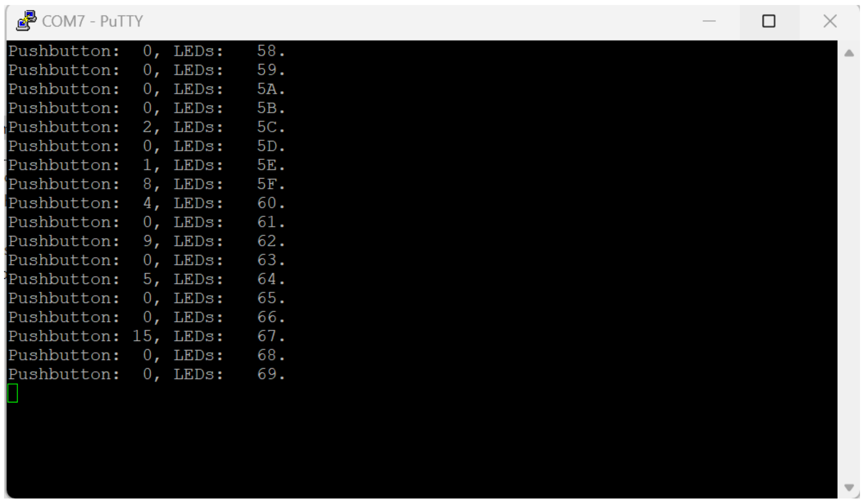





EmpyLot
nc=0;
SomeCars
LotFull
[up && ~down]{nc = nc + 1;}
[up && ~down]{nc = nc + 1;}
[down && ~up && nc<2]{nc = nc - 1;}
[nc==16]
[down && ~up]{nc = nc - 1;}
[down && ~up]{nc = nc - 1;}
1
2
3
4







```
Pushbutton:  0, LEDs:   58.
Pushbutton:  0, LEDs:   59.
Pushbutton:  0, LEDs:   5A.
Pushbutton:  0, LEDs:   5B.
Pushbutton:  2, LEDs:   5C.
Pushbutton:  0, LEDs:   5D.
Pushbutton:  1, LEDs:   5E.
Pushbutton:  8, LEDs:   5F.
Pushbutton:  4, LEDs:   60.
Pushbutton:  0, LEDs:   61.
Pushbutton:  9, LEDs:   62.
Pushbutton:  0, LEDs:   63.
Pushbutton:  5, LEDs:   64.
Pushbutton:  0, LEDs:   65.
Pushbutton:  0, LEDs:   66.
Pushbutton: 15, LEDs:   67.
Pushbutton:  0, LEDs:   68.
Pushbutton:  0, LEDs:   69.
```

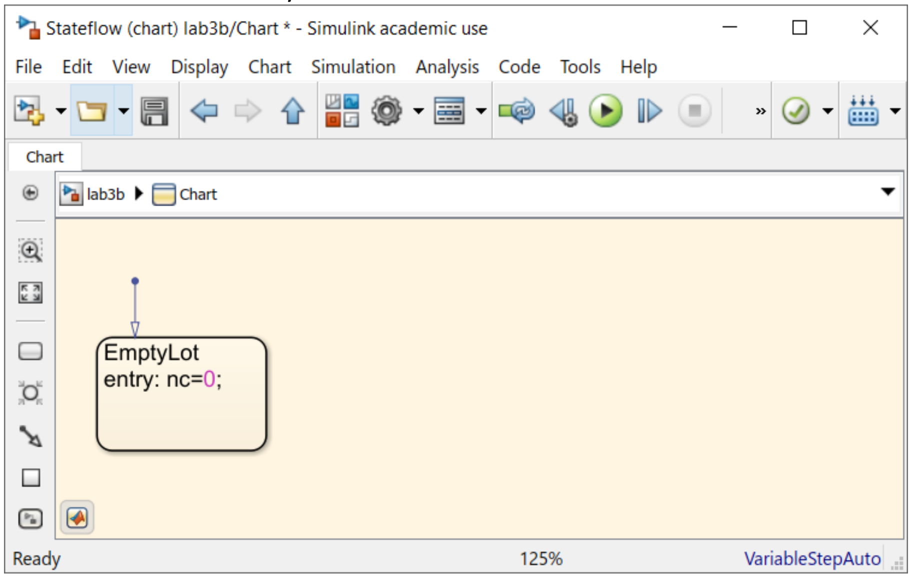
```c
    /* GPIO driver initialization */
    Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    /* Set the direction for the LEDs to output */
    XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x00);

    while (1) {
    /* Write output to the LEDs. */
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led);
    /* Wait a small amount of time so LED blinking is visible. */
    for (Delay=0; Delay<LED_DELAY; Delay++);
    btn = XGpio_DiscreteRead(&Gpio, BTN_CHANNEL);

    if (btn==1 && led<1){
        led++;
    }
    else if (btn==1 && led<0xFFFF){
        led = (led<<1) | 0x0001;
    }
    else if (btn==8 && led>0){
        led = (led>>1);
    }
    xil_printf("Pushbutton: %2d, LEDs: %4x.\r\n",btn,led);
    }
}
```

Prerequisites
Prior to starting this guide make sure to install MATLAB and Simulink. For more information, see the files "MATLAB_installation_instructions.pdf" or "COES_Virtual_Desktop_Instructions.pdf". If installing MATLAB on your personal laptop, it's recommended that you install all products, but you definitely must select Stateflow.

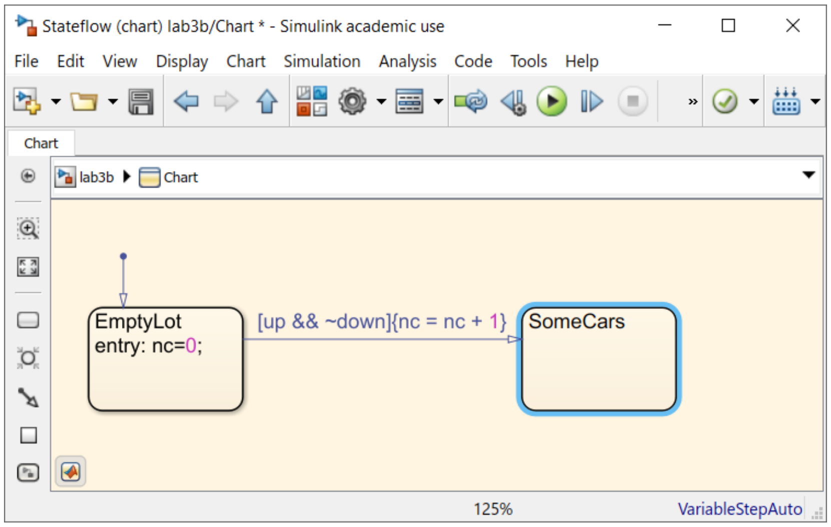
Part 1: Modeling a Car Garage Counter:

Using Stateflow, build a FSM for the car garage model discussed during class lecture. Your first FSM for this part should have two states labeled "EmptyLot" and "SomeCars", three guarded transitions, and a default transition. Your inputs, "up" and "down", are Boolean variables that register whenever a car has passed over an entry sensor (up==1) or an exit sensor (down==1). Your output, "nc" is a count of the number of cars presently in the garage, which will be displayed to the lot attendant using LEDs.
If you are already fairly familiar with Stateflow, you may proceed with the exercise on your own; otherwise, you may follow the detailed instructions below.

1. Click the Windows icon in the bottom-left corner of your screen and type "MATLAB" to find your installation. Select your latest version of MATLAB (e.g. R2019b).
2. In the MATLAB command window, enter the following command:
   ```
   >> sfnew
   ```
   This makes a new model called "untitled". Select the "Save" icon ( ) to save the model as "lab3a.slx".
3. Double-click the Chart block ( )to enter into the Stateflow chart. Begin the model by adding a State. On the left toolbar, select the State ( ) button to add a new state. Click somewhere in the left of your chart window. You should see that a new state with a default transition has been added and the text "?" is highlighted. Type "EmptyLot", press the Enter key, and then type "nc=0;" to define both the State name and an entry action statement:

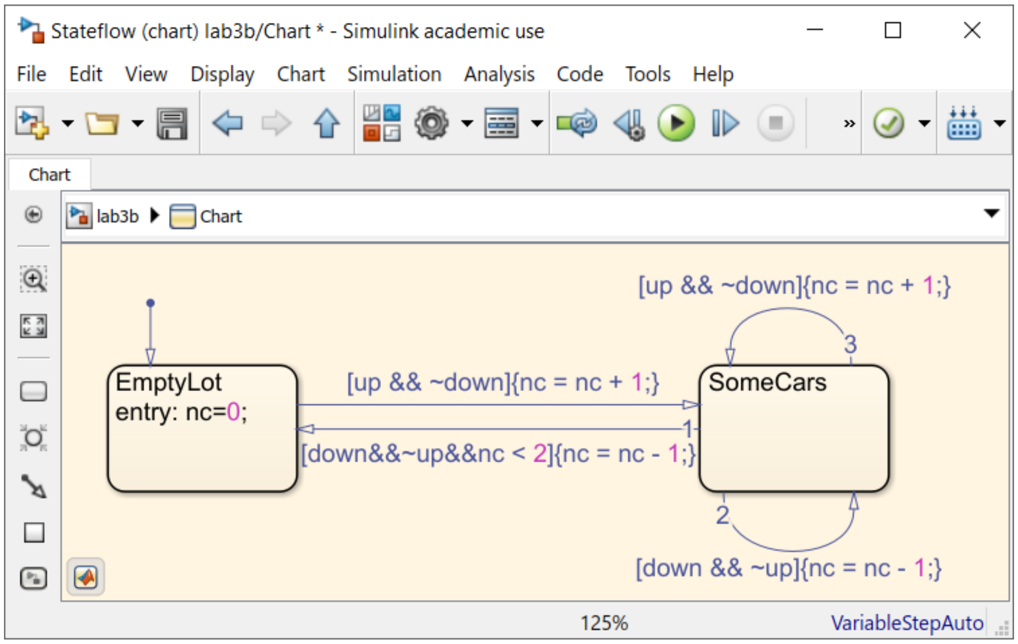

Stateflow (chart) lab3b/Chart * - Simulink academic use
File Edit View Display Chart Simulation Analysis Code Tools Help
Chart
lab3b
Chart
EmptyLot
entry: nc=0;
Ready
125%
VariableStepAuto

4. Add another state to the right and label it "SomeCars". To draw a transition from EmptyLot to SomeCars, click on the right edge of the 1st state block, and drag the mouse to the left edge of the 2nd state block. An arrow appears that connects the two states from left to right with a default description of "[ ] { }". Within the square brackets [], enter a **guard condition** of "`up && ~down`". Within the curly braces {}, enter a **transition action** of "`nc++;`"; the text will automatically change to read "`nc = nc+1;`".

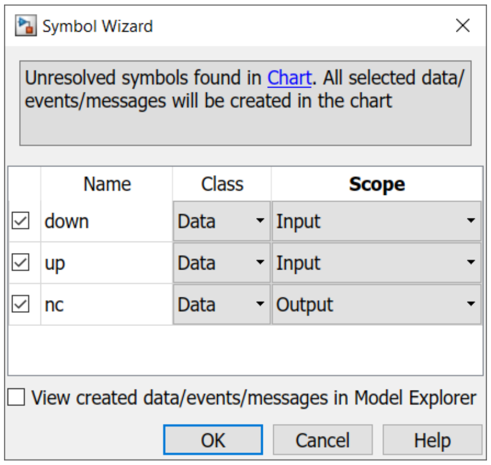

Stateflow (chart) lab3b/Chart * - Simulink academic use
File Edit View Display Chart Simulation Analysis Code Tools Help
Chart
lab3b ▶ Chart
EmptyLot
entry: nc=0;
[up && ~down]{nc = nc + 1}
SomeCars
125%
VariableStepAuto

5. Using the same method for drawing transitions, draw another transition from SomeCars to EmptyLot with a guard condition of "**`[down && ~up && nc<2]`**" and a transition action of "**`{nc--;}`**", which is automatically changed to "**`{nc = nc - 1;}`**".

6. Next, draw two self-transitions for the SomeCars state. For the 1st self-transition, click on the bottom-left edge of the state, and drag the mouse to the bottom-right corner. Add a guard condition of "**`[down && ~up]`**" and a transition action of "**`{nc--;}`**", which is automatically changed to "**`{nc = nc - 1;}`**". For the 2nd self-transition, click on the bottom-left edge of the state, and drag the mouse to the bottom-right corner. Add a guard condition of "**`[up && ~down]`**" and a transition action of "**`{nc++;}`**", which is automatically changed to "**`{nc = nc + 1;}`**".
   a. **Note**: the numbers automatically added on top of the transition arrows represent **Execution Order**. Why is execution order important for the transition to Empty Lot and the 1st self-transition?


Stateflow (chart) lab3b/Chart * - Simulink academic use
File Edit View Display Chart Simulation Analysis Code Tools Help
Chart
lab3b ▶ Chart
[up && ~down]{nc = nc + 1;}
3
EmptyLot
entry: nc=0;
[up && ~down]{nc = nc + 1;}
SomeCars
1
[down&&~up&&nc < 2]{nc = nc - 1;}
2
[down && ~up]{nc = nc - 1;}
125%
VariableStepAuto

7. From the top menu, select "Model → Update Model" or type the Ctrl-D shortcut. The "Symbol Wizard" window appears:



Symbol Wizard

Unresolved symbols found in Chart. All selected data/events/messages will be created in the chart

	Name	Class	Scope
☑	down	Data	Input
☑	up	Data	Input
☑	nc	Data	Output

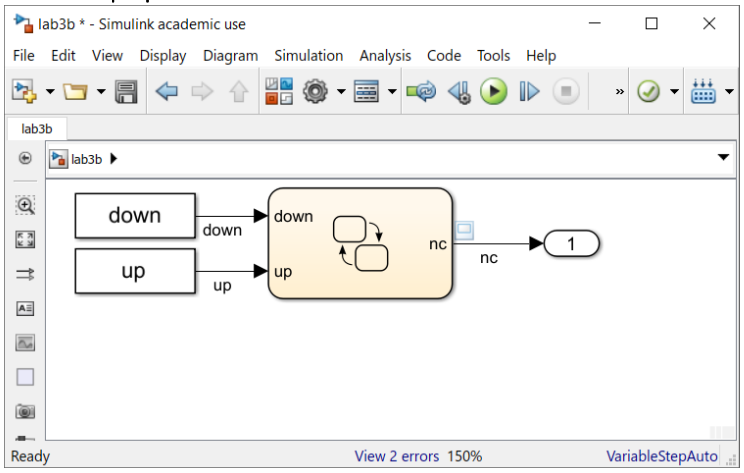
☐ View created data/events/messages in Model Explorer

Ensure that symbols "down" and "up" are set to Class "Data" (but note the option "Event") and Scope "Input". Set the symbol "nc" to Class "Data" and Scope "Output". Optionally, you may select the "View created data/events/messages in Model Explorer" check box to see where these symbols may be viewed or changed for future reference.

8. In the toolbar, select the "Up to Parent" ( ) button or click "lab3a" to return to the Simulink model at the next upper layer. Open the Simulink Library Browser ( ). In the left pane, under "Simulink", select "Sinks". Drag the "Out1" block from the Library browser into the "lab3a" model window, to the right of the Chart block. Ignore the blue box that appears. Drag the Out1 block (or select it and press your keyboard arrow keys) so that its output port connects to the output port "nc" of the Chart block.

9. Return to the Simulink Library Browser. In the left pane, under "Simulink", select "Sources". Drag the "From Workspace" block from the Library browser into the "lab3a" model window, to the left of the "down" port of the Chart block. When the blue box appears, enter "down". Double-click the block and set the "Sample Time" parameter to **1**. To connect the two blocks, first select the "down" block, then hold down the "Ctrl" Key, and select the Chart block. You can also drag an arrow from the output of "down" to the input of Chart. Double-click the arrow to label the signal as "down", or something similar.

10. Right-click on "down" and drag the block down to create a copy. Double-click the block and set the "Data" parameter to "up" and the "Sample Time" parameter to **1**. Connect the output of the "up" block to the "up" port of the Chart block. Double-click the arrow the label the signal as "up" or similar.

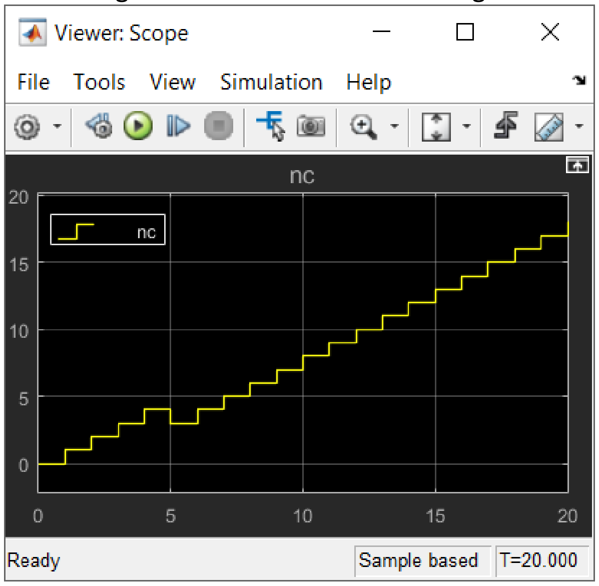


11. Label the output signal "nc". To see the system output, you can attach a signal viewer to the "nc" signal. To do so, right-click on the signal arrow, and from the context menu select "Create & Connect Viewer" → "Simulink" → "Scope". A "Viewer: Scope" window appears. A light blue scope box appears above the signal at the left.

12. In the MATLAB command window, enter commands to create the variables "**`down`**" and "**`up`**". Each variable should have 21 rows (one row for each time step) and two columns (one column for time and the other for signal values). To assign an initial set of input values, enter the following commands:
```
>> t = (0:20).'
>> down = [t,zeros(21,1)]; down(6,2) = 1;
>> up = [t,ones(21,1)]; up([1,6],2) = 0;
```
In the Workspace browser, you should see that new variables have been created named "**`down`**" and "**`up`**". Double-click each variable to see its values with respect to time in the Variable Editor.
*****Note:** The **' mark cannot be cut-and-pasted in**. You will have to manually type it***

13. Run the simulation until a stop time of 20 seconds. In the toolbar of your "lab3a" window, enter a value of "20" in the edit box to the left of the word "Normal". Then, select the Play button ( ).

14. In the Scope window that appears, you should see a yellow line showing the number of cars in the parking garage (a.k.a. the values of nc) over time. You can also add a legend to the Scope window by selecting the "Configuration Properties" button ( ), selecting the "Display" tab, and selecting the "Show Legend" checkbox. When the legend is added, the plot should appear as shown:



15. In the MATLAB workspace, note that new variables "out" and have been created to capture the simulation output because of the "Out1" block. View it by typing "**`out.plot`**". You can save these variables to a MAT-file by entering the following command at the MATLAB command prompt:
```
>> save lab3a out
```

Once you've completed these steps, you may call over the instructor to mark you down for Part 1, or you can wait until you've completed the entire Lab before calling him over.

Part 2: Modeling a Space-Limited Car Garage Counter

For Part 2, update the Stateflow model to include a 3rd state named "LotFull". Your FSM should enter this state when the number of cars reaches the total number of parking spaces. For this part, you can assume that the parking garage has a constant 16 spaces. Save this revised model as "lab3b.slx". Ensure to re-run the simulation to show your modified output behavior.

16. In the MATLAB workspace, note that new variable "out" has been updated to capture the simulation output from the Out1 block. View by typing "`out.plot`". You can save these variables to a MAT-file by entering the following command at the MATLAB command prompt:

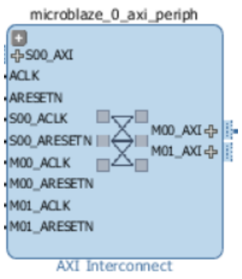
```
>> save lab3b out
```

Once you've completed this, you may call over the instructor to mark you down for Part 2, or you can wait until you've completed the entire Lab before calling him over.

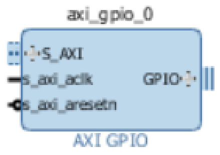
Part 3: Synthesizing GPIO in Vivado and Interpreting I/O Values

1.) Start a new project called "Lab3" in Vivado. You will need to re-run Lab 2 steps 11-15 to build a new Microblaze wrapper. Do not just copy your lab2 files over as this causes continuity issues! Start with a fresh build.

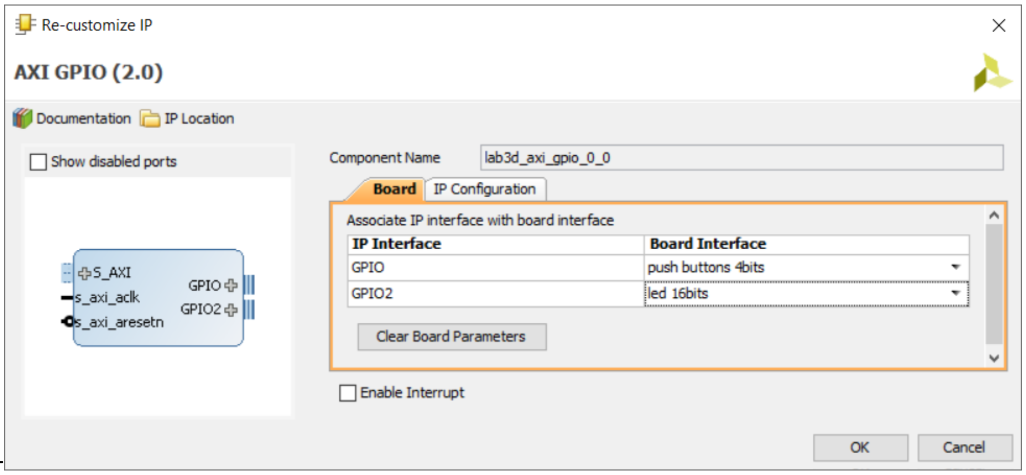
2.) Double-click the "AXI Interconnect" block. Set "Number of Master Interfaces" to 2. You should now see another port on the right side of the block labeled "M01_AXI".



3.) Select the Add IP ( ) button, or right-click and select "Add IP…". Type "GPIO", and then double-click "AXI GPIO". You should see the "AXI GPIO" block has now been added to the block diagram.



4.) Double click the AXI GPIO block. Set the 1st entry whose IP Interface is "GPIO" to have a Board Interface of "push_buttons_4bits" and set the 2nd entry whose IP Interface is "GPIO2" to "led_16bits". Click OK.


Re-customize IP

AXI GPIO (2.0)

Documentation IP Location

Show disabled ports

Component Name: lab3d_axi_gpio_0_0

Board | IP Configuration

Associate IP interface with board interface

IP Interface	Board Interface
GPIO	push buttons 4bits
GPIO2	led 16bits

Clear Board Parameters

Enable Interrupt

OK Cancel

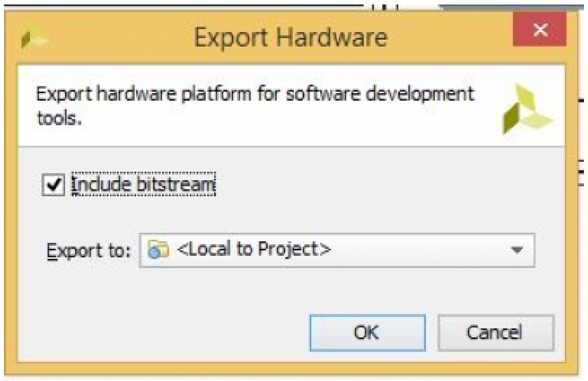
+

5.) Run connection automation. Select "All Automation" checkbox, and click OK. External ports are created, connections made. Then, select the Regenerate Layout ( ) button to clean up the block diagram. Next, select the Validate Design ( ) button to check for errors.

6.) In your "Sources" browser, below "Design Sources", select your block diagram given by the icon, right-click, and select "Create HDL Wrapper". In the "Create HDL Wrapper" window that appears, select "Let Vivado manage wrapper and auto-update", and then click OK.

7.) In the Flow Navigator pane on the left side, select Run Synthesis to run Synthesis. Select Run Implementation to Run Implementation. Select Generate Bitstream to generate a bitstream.

8.) On the main toolbar, click **File** and select **Export→Export Hardware**. Check the box to **Include Bitstream** and click **OK**. This will export the hardware design with system wrapper for the Software Development Tool - Vivado SDK.


Export Hardware

Export hardware platform for software development tools.

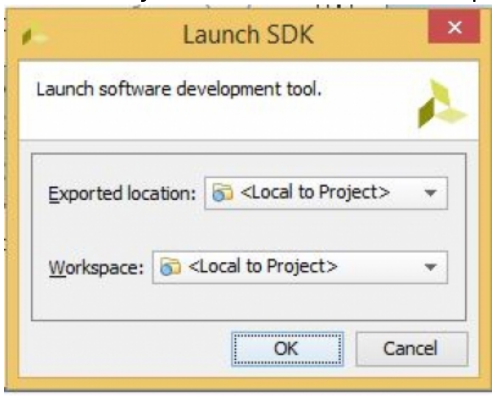
☑ Include bitstream

Export to: <Local to Project>

OK Cancel

A new file directory will be created under **Lab3.SDK** similar to the Vivado hardware design project name. Two other files, *.sysdef* and *.hdf* are also created. This step essentially creates a new SDK Workspace.

9.) In the Vivado top menu, click **File** and then **Launch SDK**. Leave both dropdown menus as their default *Local to Project* and click **OK**. This will open Xilinx SDK and import your hardware.



Launch software development tool.

Exported location: <Local to Project>

Workspace: <Local to Project>

10.) Creating New Application Project in SDK

In the SDK top menu, click the **New** dropdown arrow and select **Xilinx→Application Project**.
Give your project a name that has no empty spaces (e.g. "Lab3f") and click **Next**.
On the Templates page, select "Hello World" and click **Finish**.

11.) In the **Project Explorer**, right-click helloworld.c under the **src** folder and rename it (e.g. "**lab3h.c**").
On line 51, after the existing #include statements, add the following new #include statements.

```
/* Include Files */
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"
```

12.) On line 57, beneath the #include statements, add the following new #define statements.

```
/* Definitions */
#define GPIO_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID /* GPIO device for LEDs*/
#define LED 0x0000          /* Initial LED value */
#define LED_DELAY 10000000  /* Software delay length: 10 mil loops   */
#define BTN_CHANNEL 1       /* GPIO port for pushbuttons */
#define LED_CHANNEL 2       /* GPIO port for LEDs */
#define printf xil_printf   /* A smaller, optimized printf */
XGpio Gpio;                 /* GPIO Device driver instance */
```

13.) On line 66, beneath the #define statements, add the following new function declaration, **LEDOutputExample()**, and define its local variables.

```
int LEDOutputExample(void) {
     volatile int Delay;
     int Status;
     uint16_t led = LED;   /* Hold current LED value. */
     uint8_t btn;          /* Hold current pushbutton value. */
     uint8_t ncar = 0U;    /* Hold number of cars. */
}
```

14.) On line 72, after the local variable initialization, add new AXI GPIO Initialization commands.

```
/* GPIO driver initialization */
Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}
/* Set the direction for the LEDs to output */
XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x00);
```

15.) On line 79, after the AXI GPIO Initialization, add a while loop, with GPIO write and read calls. Also insert a one-line `for` loop to pause for a fixed amount of time before the next iteration. Consider raising/lowering the **LED_DELAY** variable to allow the program to wait for more or less time between readings.

```
while (1) {
    /* Write output to the LEDs. */
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led);
    /* Wait a small amount of time so LED blinking is visible. */
    for (Delay=0; Delay<LED_DELAY; Delay++);
    btn = XGpio_DiscreteRead(&Gpio, BTN_CHANNEL);
}
```

16.) On line 85, within the while loop and after the GPIO read/write calls, display the current pushbutton and LED values on the terminal using the `xil_printf()` function. Either display them as decimal (%d) or hexadecimal (%x), depending on your preference. Change the LED value to display alternate hex symbols at the terminal.

```
xil_printf("Pushbutton: %2d, LEDs: %4x.\r\n",btn,led);
led++;
```

17.) On line 90, in the **main()** function, add a call to the **LEDOutputExample()** function.

```
int main()
{
   init_platform();
   int Status;
   /* Execute the LED output. */
   Status = LEDOutputExample();
   if (Status != XST_SUCCESS) {
    xil_printf("GPIO output to the LEDs failed!\r\n");
   }
   return 0;
}
```

18.) Save the C file. Program FPGA. Build executable.

19.) Start Terminal window using the same settings as in Lab 1. Ensure that you receive the LED and pushbutton readings. Make notes about what the different Hexadecimal values mean for the pushbuttons and LEDs, respectively.

Once you've completed these steps, you may call over the instructor to mark you down for Part 3.

Part 4: Implementing the Car Garage Counter using Pushbuttons and LEDs

For part 4, you will implement the car garage counter whose system behavior you had modeled in Part 2. When the Up button is pressed, this should signify a car entering the parking garage, and the number of lit LEDs lighted should increment. When the Down button is pressed, this should signify a car exiting the parking garage, and the number of lit LEDs should decrement. Note the following hints (if you'd like):

- Each hexadecimal symbol represents 4 bits.
- There are 4 pushbuttons, so all four can be represented by a single hexadecimal digit (values 0-15).
- There are 16 LEDs, which can be represented by four hexadecimal digits (e.g. 0xABCD).
- In C programming, you can use the binary AND (&) operator to mask the pushbutton variable to find whether a specific button was pressed.
- In C programming, you can use the binary left-shift (<<) operator to shift a sequence of bits to the left (e.g. led<<1 shifts left by 1 place). Doing so would easily allow you to display another lit LED if you combine it with the binary OR (|) operator to light the least significant bit (0x0001).
- In C programming, you can use the binary right-shift (<<) operator to shift a sequence of bits to the right (e.g. led>1 shifts right by 1 place). Doing so would easily allow you to display one LED fewer.

Once you've completed these steps, you may call over the instructor to mark you down for Part 4.