WELCOME TO LabVIEW 1

ELECTRICAL ENGINEERING 20N
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

SIMON HONG, HSIN-I LIU, JONATHAN KOTKER, AND BABAK AYAZIFAR

1 Introduction

LabVIEW is a graphical programming language that uses icons instead of lines of text to create applications. In contrast to text-based programming languages, where instructions determine program execution, LabVIEW uses **dataflow programming**, where the flow of data determines execution order. LabVIEW is the educational vehicle of EE20N this semester, and will be used to help explore and understand concepts taught simultaneously in lecture. The purpose of this lab session is to explore basic LabVIEW concepts and features, in order to provide the foundation for future lab sessions.

1.1 Lab Goals

- To become comfortable with the LabVIEW environment and dataflow execution.
  - Front panels.
  - Block diagrams.
  - `Functions` and `Controls` palettes.
- To use built-in LabVIEW functions.
  - Use LabVIEW to solve problems.
  - Learn LabVIEW concepts.
- To find and use math and complex analysis functions.
- To work with data types.
- To display results.

1.2 Checkoff Points

2. Getting Started . . . .
3. Dataflow Programming . . . .

4. Creating a Simple VI .... (45 minutes)
   1. Terminology ....
   2. Example VI .... (10%)
   3. Self-Exercise .... (10%)
5. While Loop .... (30 minutes)
   1. Terminology ....
   2. Example VI .... (10%)
6. For Loop .... (45 minutes)
   1. Terminology ....
   2. Example VI .... (10%)
   3. Shift Register ....
   4. Selectors ....
   5. Self-Exercise .... (10%)
7. Acknowledgments ....

2 Getting Started

1. Please make sure to read the lab guide for lab 00, located in the bSpace site for the course, under `Resources → Labs`.
2. Log in to a computer in 105 Cory using your assigned account. This account is valid throughout the semester, and you should remember your login ID. Follow the instructions on the account sheet to change your password to a more memorable one.
3. Send an e-mail to your lab TA with your name, major, year, instructional account, and something amusing about yourself. Doing this *right now* will immediately give you 30% of the points alloted for this lab. E-mail will be your primary mode of communication with the EE20N course staff, whereby you can either raise short questions and problems, or make appointments to discuss more involved problems.

3 Dataflow Programming

DATAFLOW PROGRAMMING

LabVIEW uses a paradigm of programming known as **dataflow programming**. In the LabVIEW implementation of this paradigm, data flows from left to right, which passes through various blocks and modules that transform the data in interesting ways. This is analogous to a series of pipes connecting several machines together in, say, a factory; data can be imagined to be the liquid that flows through these pipes and machines. The user is provided with an interface through which to feed this data. The data is then processed and returned to the user through a similar interface.

4 Creating a Simple VI

4.1 Terminology

VIRTUAL INSTRUMENTS

LabVIEW programs are called **virtual instruments (VIs)**. As the name implies, you will be constructing an instrument that exists only in the computer, but otherwise similar to a tangible instrument, which will allow the user to send and receive meaningful data. Each VI has two windows:

1. **Front Panel**. This is where the user interacts directly with the VI. The front panel abstracts detail away from the user, and provides the user with as many inputs (or **controls**) and outputs (or **indicators**) required for the VI to function properly and meaningfully.

   FRONT PANEL
   CONTROLS
   INDICATORS

2. **Block Diagram**. The block diagram, in several ways, represents a flowchart. It receives the input data supplied to the controls on the front panel, manipulates them in interesting ways through **wires** and **blocks**, and then displays the output data through the indicators on the front panel. As a corollary, *every front panel control or indicator has a corresponding terminal on the block diagram*.

   BLOCK DIAGRAM
   WIRES
   BLOCKS

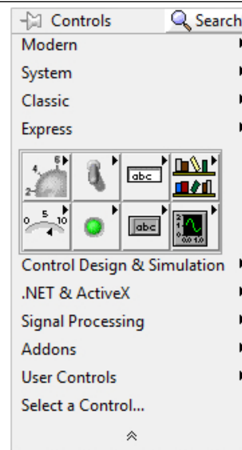
Another important part of any VI are the **icons** and the **connectors**. These allow your VI to communicate with other VIs elsewhere, and thus allows your VI to be used as a **sub-VI** in other VIs. This allows clean and modular block diagrams, where each sub-VI performs a certain task, and the inputs and results of each sub-VI are manipulated appropriately by the containing VI.

ICONS
CONNECTORS
SUB-VI

All of the available functions, blocks and VIs are located inside **palettes**. There are two main palettes: the `Controls` palette on the front panel and shown in Figure 1, and the `Functions` palette on the block diagram, as shown in Figure 2. Both of these palettes are accessible by right-click.

PALETTES

Figure 1 `Controls` Palette



As denoted by their names, the `Controls` palette contains both the controls and the indicators that are available for use on the front panel, while the `Functions` palette contains the blocks and the VIs that are available for use on the block diagram. Since there are so many blocks possible and available, they have been categorized, and both of the main palettes have search functionalities that allow you to find any block that you think might be useful.

4.2 Example VI

OK, enough talk: let's get down to business! The following exercise will demonstrate how to create a basic VI which calculates the area and perimeter of a circle, given its radius r.

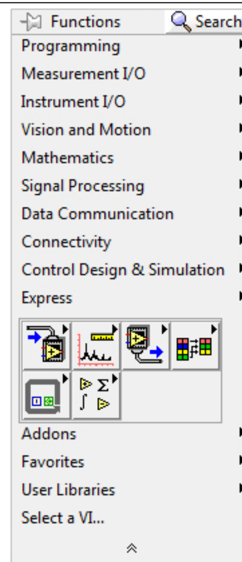
1. Open LabVIEW.

2. Open a new VI by clicking on `Blank VI` in the LabVIEW `Getting Started` window.

3. Save the VI as `Circle Area Perimeter.vi`:

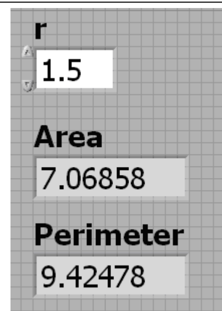
   - Select `File` $\rightarrow$ `Save`.

Figure 2 `Functions` Palette



- Navigate to the location where you wish to save the VI.
- Name the VI `Circle Area Perimeter.vi`.
- Click `OK`.

Figure 3 `Circle Area Perimeter.vi` Front Panel



4. Create the front panel in Figure 3.

   - Right-click on the front panel to open the `Controls` palette, and put down a `Numeric Control` from the `Modern` → `Numeric` subpalette, as shown in Figure 4.
   - Double-click on the label for this control and rename it to `r`.
   - Place down two numeric indicators from the `Modern` → `Numeric` subpalette and rename them `Area` and `Perimeter`.

     Note the distinction between an *indicator* and a *control*. A control receives input, while an indicator displays output.

   - You can change the font of the labels using the font drop-down menu near the menu bar, as shown in Figure 5.

Figure 4 `Numeric` Subpalette from the `Controls` Palette

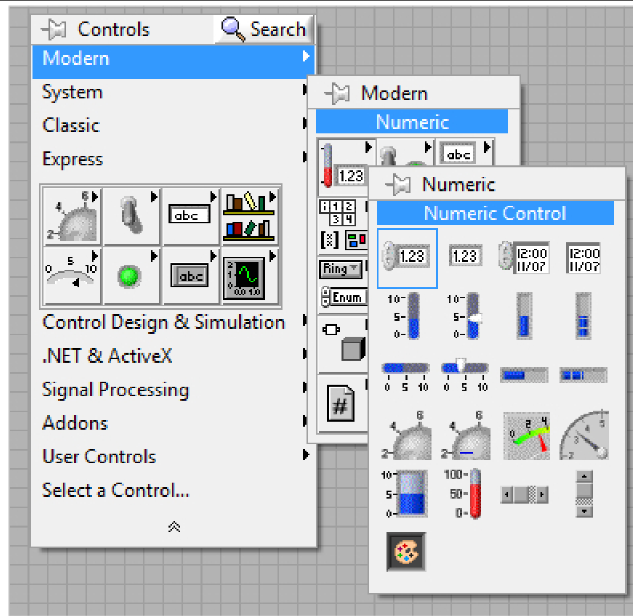
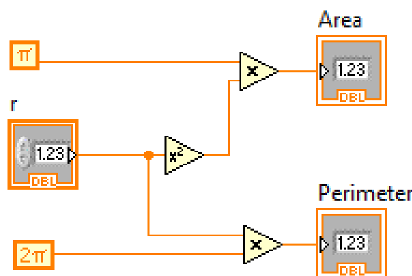


Figure 5 Font Drop-Down Menu

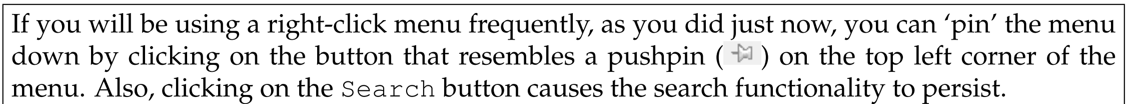
5. Open `Context Help` using the key combination `Ctrl-H`. When you hover over any block or connector, the `Context Help` will provide a brief description of its function or its type, depending on context. This will prove very helpful in creating VIs.

6. Create the block diagram for this VI as shown in Figure 6.

Figure 6 `Circle Area Perimeter.vi` Block Diagram

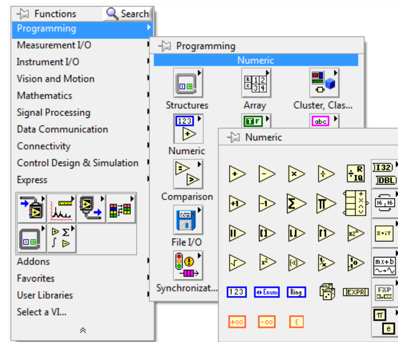


- Right-click on the block diagram to open the `Functions` palette and place down the necessary mathematical functions from the `Programming` → `Numeric` subpalette, as shown in Figure 7.



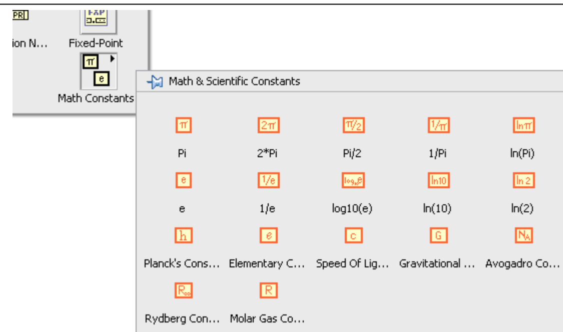
If you will be using a right-click menu frequently, as you did just now, you can 'pin' the menu down by clicking on the button that resembles a pushpin ( ) on the top left corner of the menu. Also, clicking on the `Search` button causes the search functionality to persist.

Figure 7 `Numeric` Subpalette from the `Functions` Palette



- Place down the necessary constants from the `Programming` → `Numeric` → `Math Constants` subpalette, as shown in Figure 8.

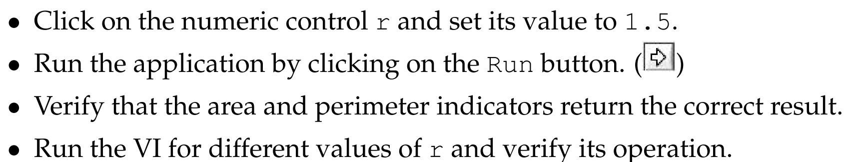
Figure 8 `Math Constants` Subpalette from the `Functions` Palette



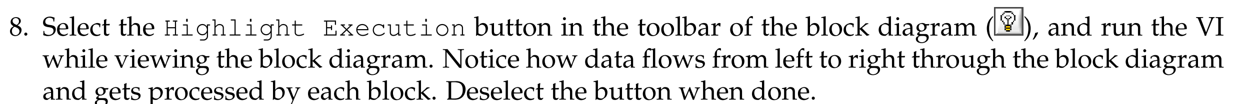
- Wire the functions as shown in Figure 6.

> The key combination `Ctrl-E` will help you toggle quickly between the front panel and the block diagram.

7. Run the VI and verify its operation.



   - Click on the numeric control `r` and set its value to `1.5`.
   - Run the application by clicking on the `Run` button. ( )
   - Verify that the area and perimeter indicators return the correct result.
   - Run the VI for different values of `r` and verify its operation.

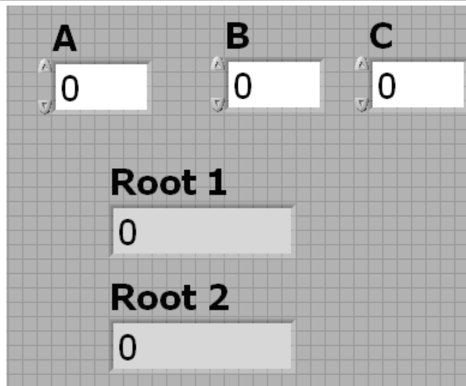


8. Select the `Highlight Execution` button in the toolbar of the block diagram ( ), and run the VI while viewing the block diagram. Notice how data flows from left to right through the block diagram and gets processed by each block. Deselect the button when done.

4.3 Self-Exercise

Now that you have learned how to build a simple VI, implement the following VI that calculates the roots of a quadratic equation $Ax^2 + Bx + C = 0$ based on its coefficients. Save this VI as `Quadratic Roots.vi`. The front panel is shown in Figure 9. For this simple exercise, the VI will not support complex roots.

Figure 9 `Quadratic Roots.vi` Front Panel



> Messy VI block diagram? Do not despair! The key combination `Ctrl-U` will clean it up for you. The key combination `Ctrl-B` removes all broken wires.

5 While Loop

5.1 Terminology

More than often, we would like an action to repeat until a condition is satisfied; for example, we may want to accept data from the user until the datum provided is of a particular value, at which point we stop processing any further data. This is where **loop structures** such as the `While Loop` come in handy.

LOOP STRUCTURES

A `While Loop` executes the functions it contains until the **conditional terminal** receives a specific Boolean value (`True` or `False`). By default, the conditional terminal is **Stop If True**(); this means that, when we connect a **Boolean control** (a control that returns either `True` or `False`) to the terminal, the `While Loop` will stop iterating as soon as the Boolean control returns `True`. Another kind of conditional terminal, called **Continue If True**, is available by right-clicking on the conditional terminal.

CONDITIONAL TERMINAL

BOOLEAN CONTROL

There is another terminal called the **iteration terminal**, which is an output terminal (i). Every run of the `While Loop` is considered one **iteration**. During, and after, execution, this terminal returns the number of iterations completed. The iteration terminal starts off at `0`; as a result, during the first iteration, the iteration terminal returns `0`.

ITERATION TERMINAL

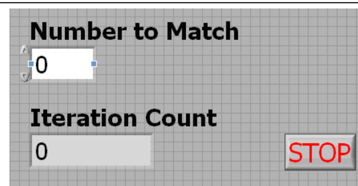
ITERATION

5.2 Example VI

The following exercise will demonstrate the use of a `While Loop` structure in a VI. This VI will keep generating a random integer between 0 and 100 in a loop until it matches a number defined by the user. An indicator will be used to count the number of iterations used to attain this match.

1. Open LabVIEW.
2. Open a new VI by clicking on `Blank VI` in the LabVIEW `Getting Started` window.
3. Save the VI as `Number to Match.vi`.
4. Create a random number generator, to generate integers between 0 and 100 inclusive, in a `While Loop`.

Figure 10 `Number to Match.vi` Front Panel

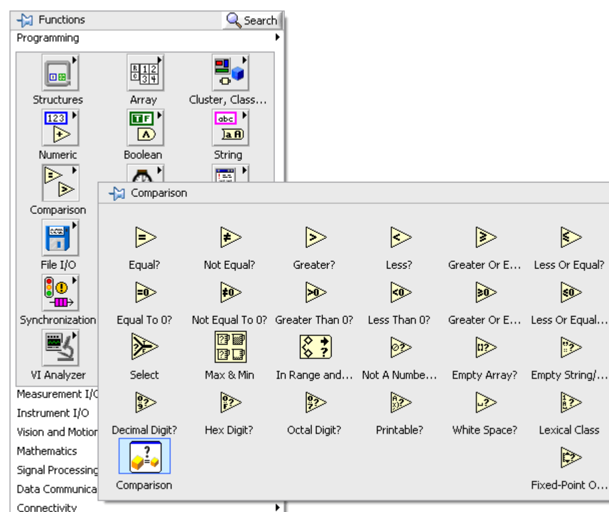


- From the `Functions` palette, place a `While Loop` from the `Programming` → `Structures` subpalette.
- Using the random number generator found in the `Programming` → `Numeric` subpalette, generate a random integer between 0 and 100. Since the random number generator generates random floating-point values between 0 and 1, use appropriate functions in the `Functions` palette to multiply the values by 100 and round them to an integer. You may find the `Round to Nearest` function useful, but you will need to look for it using the provided search functionality.

5. Create a control on the front panel that will be used to compare the results of the random number to a user-defined value, and display the number of iterations that were required to attain this match.

    - From the `Controls` palette, place a numeric control on the front panel and label it `Number to Match`.
    - From the `Controls` palette, place a numeric indicator on the front panel and label it `Iterations`.
    - Create an `Equal?` function from the `Programming` → `Comparison` subpalette and place it on the block diagram to compare the random number generated with the number present in `Number to Match`, as shown in Figure 11.

Figure 11 `Comparison` Subpalette from the `Programming` Palette



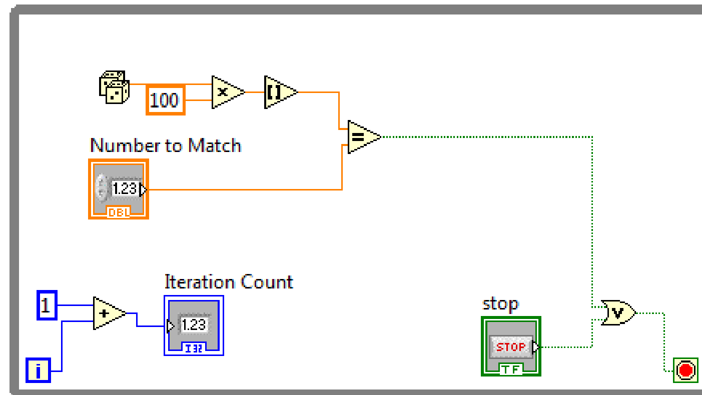
6. Create stop conditions for the `While Loop`.

    - It is good programming practice to allow users the ability to exit a loop structure using a control on the front panel. To this effect, create a stop button control on the front panel from the `Modern` → `Boolean` subpalette.

- To *also* allow the While Loop to terminate when a match has been found, use the appropriate logic functions from the Programming → Boolean subpalette to construct the necessary logic for the stop terminal of the loop. The overall block diagram should look similar to that shown in Figure 12.

Notice, in the block diagram, that 1 has been added to the iteration terminal (i); this is done because iteration counts for loop structures in LabVIEW begin at 0, whereas we need to start counting iterations from 1. To create a constant (in this case, 1), right-click on the terminal and select Create → Constant.

Figure 12 Number to Match.vi Block Diagram



7. Run Number to Match.vi and verify its operation. Run the VI again with the Highlight Execution button selected.

6 For Loop

6.1 Terminology

A For Loop, like a While Loop, is a loop structure and functions similarly; the sole exception is that a For Loop performs as many iterations as determined by the **count terminal** (N).

COUNT TERMINAL

6.2 Example VI

The following exercise will demonstrate the use of a For Loop structure in a VI. This VI will generate a random floating-point number between 0 and 1 every second and repeat for a finite number of iterations specified by the user. The VI will then return the largest number generated. An indicator will be used to count the number of iterations elapsed.

1. Open LabVIEW.
2. Open a new VI by clicking on Blank VI on the LabVIEW Getting Started window.
3. Save the VI as Max Search.vi.
4. Create a random number generator to generate floating-point numbers between 0 and 1 in a For Loop and display it on the front panel. Have this loop repeat every second.
    - In the block diagram, place a For Loop from the Programming → Structures subpalette.

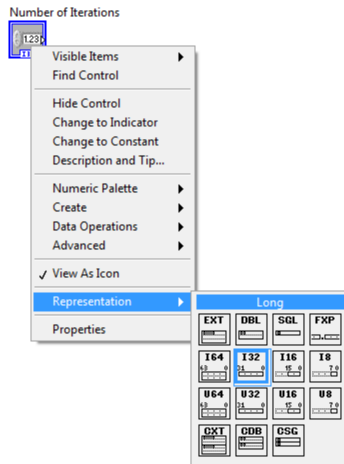
- Also in the block diagram, place a random number generator found in the `Programming` → `Numeric` subpalette.
- Create a numeric indicator labeled `Random Number` on the front panel, from the `Modern` → `Numeric` subpalette, to display the random numbers. In the block diagram, wire the random number generated to this indicator.
- Back in the block diagram, place a `Wait (ms)` function into the `For Loop` and wire it to a constant of `1000`. You can do this by right-clicking on the left terminal (the one labeled `milliseconds to wait`) and selecting `Create` → `Constant`. Again, use either the search functionality or the available categories to determine where the function itself is located in the `Functions` palette.

What have you just done? By adding the `Wait (ms)` block, you ensure that every iteration waits 1000 milliseconds, or 1 second, after completion before the next iteration starts.

5. Create a control on the front panel which will be used to define the number of iterations for the `For Loop` and place a numeric indicator to display the current iteration value.

   - From the `Controls` palette, place a numeric control on the front panel and label it `Number of Iterations`. Since this value can only be an integer, change the data representation of this control from `DBL` (which stores double precision floating-point numbers) to `I32` (which stores integers) by right-clicking on the control *in the block diagram*, selecting `Representation` and choosing `I32`, as shown in Figure 13. Connect this control to the iteration value `N` of the `For Loop`.

Figure 13 How to Change the Data Representation



- From the `Controls` palette, place a numeric indicator on the front panel and label it `Iterations`. Wire the iteration index `i` to this indicator in the block diagram.

We now need to keep track of the maximum number generated in the iterations that have completed, but this involves maintaining history. In other words, we need a way to remember past values. Before we continue with this exercise, we will explore the concept of a shift register.

6.3 Shift Register

A **shift register** will be one of the most useful and ubiquitous constructs that we will use in the labs to come. In order to understand how a shift register works, we draw the following analogy: Notice how the `For Loop` looks like a finite sheaf of papers. This represents the fact that whatever is present on the topmost sheet is duplicated onto the other sheets. As the VI runs, each of these sheets are 'run' one after the other: in effect, we are thus repeatedly running the code on the topmost sheet, as a `For Loop` must.

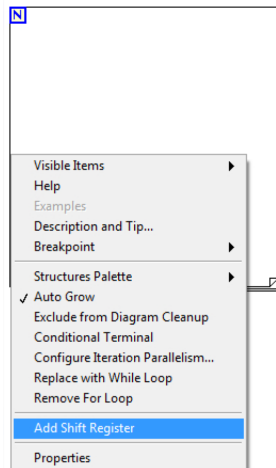
Essentially, a shift register is like a data pipeline connecting two adjacent sheets, or iterations, together; the right square saves the result of the current iteration so that the next iteration can use it, while the left one uses the result of the previous iteration. Formally, during the ith iteration, data sent forward from the $(i-1)^{\text{th}}$ iteration can be obtained from the left square of the shift register, manipulated or used in some way during the current iteration, and then sent forward to the $(i+1)^{\text{th}}$ iteration using the right square of the shift register. As a result, the shift register allows us to remember values from previous iterations. We will use this property for the current task.

In other words, the left square of the shift register can be seen as an input from the previous iteration, and the right square of the shift register can be treated as an output for the current iteration.

6. Find the largest random number generated.

    - Right-click on the left border of the `For Loop` and select `Add Shift Register`. You should now see a square on either side of the `For Loop`: the left one pointing down (), and the right one pointing up (); together, they constitute a shift register.

Figure 14 How to Add a Shift Register

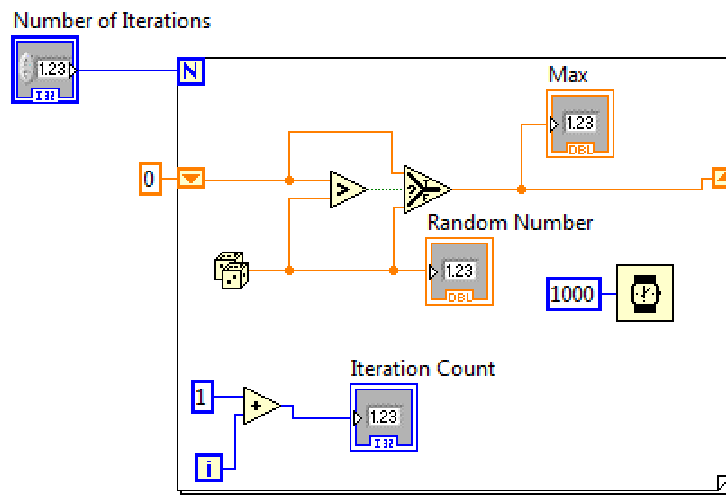


- Outside the `For Loop` but near the left square of the shift register, drag a numeric constant from the `Programming` → `Numeric` subpalette and initialize this constant to `0`. Then, connect this constant to the left square.

> Notice that the left square now has a constant attached to it (in this case, that constant is `0`). This is because, in the first iteration, there is no data from a previous iteration to be used, so an initial value must be attached to the left shift register; in this case, during the first iteration, the left shift register will produce the value `0`.
>
> If this initial value is not present, then running the VI the first time will cause LabVIEW to use a default initial value (usually `0` for numeric data), but running the VI subsequently will cause LabVIEW to use, as initial values, the values obtained from the previous runs of the VI, which is usually unintended behavior.

- Create a numeric indicator labeled `Max` on the front panel from the `Modern` → `Numeric` subpalette to display the maximum number.
- Using the necessary logic functions from the `Programming` → `Comparison` subpalette, implement an algorithm that will compare the random number with the previous maximum value and update the new maximum value. Figure 15 shows the final block diagram for `Max Search.vi`.

Figure 15 `Max Search.vi` Block Diagram



Wait, what is that large triangular block in the block diagram?

6.4 Selectors

A selector acts as a sluice gate, and as such, allows us to choose which data should flow through a wire. It is analogous to the `if`-construct in programming languages. A selector has three input terminals: `t`, `s`, and `f`. The `s` terminal accepts a wire carrying Boolean data, which is data that is either `True` or `False`. If the datum is `True`, then the data contained in the wire connected to the `t` terminal is allowed to flow through; if the datum is `False`, then the data contained in the wire connected to the `f` terminal is allowed to flow through.

In the example shown in Figure 15, the `Greater Than` comparison block produces a `True` Boolean value, if the value flowing into its upper terminal is strictly greater than the value flowing into its lower terminal. This Boolean value feeds into the `s` terminal of the selector. With this in mind, we interpret the block diagram as follows: if a past value, provided by the shift register, is greater than the number randomly generated in this iteration, then that number is propagated to the next iteration; otherwise, the new random number is propagated. As a result, the shift register remembers the maximum value generated so far.

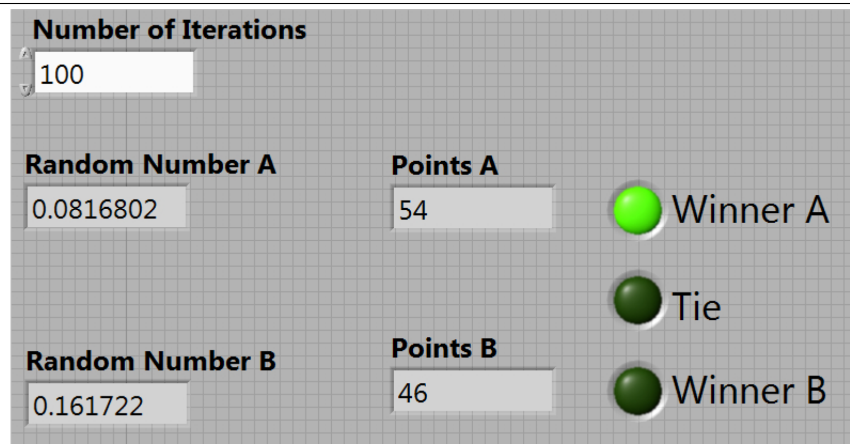
7. Run `Max Search.vi` and verify its operation. Run the VI again with the `Highlight Execution` button selected.

6.5 Self-Exercise

Now that we learned how to use a `For Loop`, implement the following VI which generates two random numbers between -0.5 and 0.5 for a set amount of time defined by the user. Denote the first random number as `Random Number A` and the second as `Random Number B`. Points are attributed to a random number whenever its value is greater than the other. Write a VI that keeps track of the points for each random number generator and displays the winner, with the help of Boolean indicators. Also, include an indicator in the event of a tie. Note that the winner should only be displayed at the end of the competition (once all iterations have finished). Save this VI as `Random Competition.vi`.

> You will probably need two shift registers and two selectors to achieve this task, one each for `A` and for `B`. Think about what each shift register should remember, and how these values should be updated; for instance, if `A` gets a higher score than `B`, which shift register should be updated? How about the other shift register?
>
> Also, using the analogy of data flow, where would you obtain the final scores of `A` and of `B`, and how would you use this information to determine the winner?

Figure 16 `Random Competition.vi` Front Panel



7 Acknowledgments

Special thanks go out to the teaching assistants (TAs) of the Spring 2009 semester (Vinay Raj Hampapur, Miklos Christine, Sarah Wodin-Schwartz), of the Fall 2009 semester (David Carlton, Judy Hoffman, Mark Landry, Howard Lei, Feng Pan, Changho Suh), and of the Spring 2010 semester (Xuan Fan, Brian Lambson, Kelvin So) for providing suggestions, ideas, and fixes to this lab guide. Portions of this lab guide were derived from material contained in the LabVIEW reference guides published by National Instruments.

> This is not a box. You might *think* it is a box, but you're mistaken.