MSOC Lab #3

Yu-Sheng Lin

johnjohnlys@media.ee.ntu.edu.tw

Outline

- In Lab #3, we discuss about high-level RTL verification.
- You will learn VPI (Verilog Procedural Interface), which can connect C with Verilog.
- Using VPI + Python-C interface + Lab #2, you can write testbench in Python!
- Note: the environment is hard to set-up, so we provide a workstation account for you.

Example #0
RTL Used in Lab #3

The Module Used in This Lab

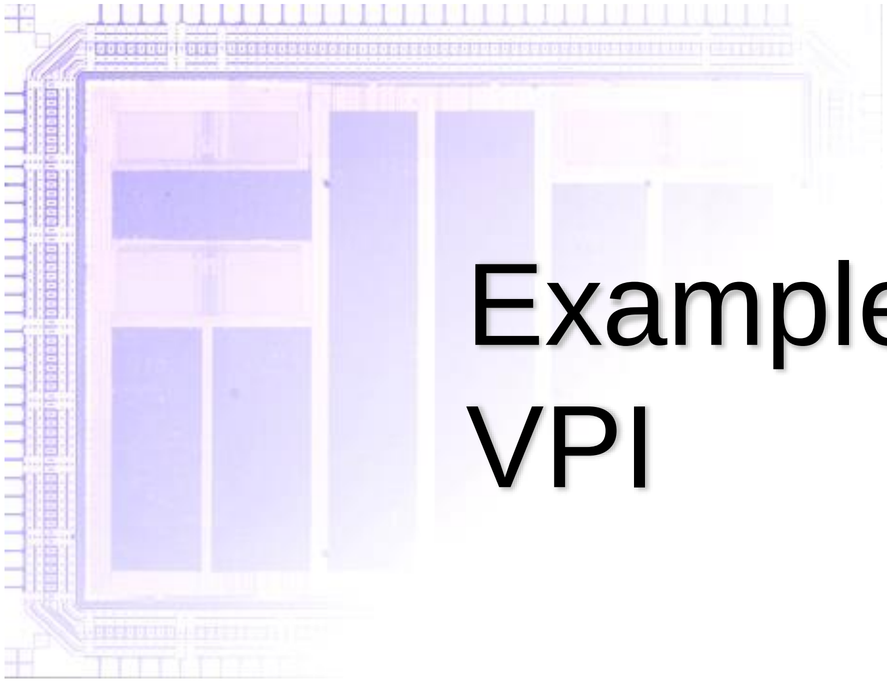
```
module ToUpper(
    input i_clk,
    input i_rst,
    input       i_valid,
    input [7:0] i_char,
    output logic       o_valid,
    output logic [7:0] o_char
);
```

Valid = 1: this cycle holds valid data
Valid = 0: this cycle doesn't hold valid data

The output uses the same protocol

Spec:
Convert all lower case characters to upper.
Any other characters should keep unchanged.
Convert input characters to upper case

It is in lab31_design.sv, and we leave it as a small homework.



Example #1
VPI

Verilog Procedural Interface

- Wikipedia gives a good example
  - https://en.wikipedia.org/wiki/Verilog_Procedural_Interface
- Sadly, it's very very difficult to find any other documents.
- Fortunately, we only have to understand a very minor part of VPI.

Verilog Procedural Interface

```
vpiHandle wire;
s_vpi_value wire_value;
int int_value;

wire_value.format = vpiIntVal;
vpi_get_value(wire, &wire_value);
int_value = wire_value.value.integer;


wire_value.value.integer = int_value + 1;
vpi_put_value(wire, &wire_value, NULL, vpiNoDelay);
```

vpiHandle: "pointer" to wire
s_vpi_value: Somethine like 32'd821

vpiIntVal: You have to provide a type such as integer, integer with z or x, string to read or write the value.

Verilog Procedural Interface

```
wire_value.format = vpiVectorVal;
vpi_get_value(wire, &wire_value);
int_value = wire_value.value.vector->aval;
xxx_value = wire_value.value.vector->bval;
```

vpiVectorVal: use vector->aval or bval to encode the 01xz

Verilog Testbench with VPI

```
initial while (1) @(posedge i_clk) #0 $Lab31Cycle;

initial begin
    $Lab31Start;
    ...
    #1000
    $Lab31Stop;
    $finish;
end

ToUpper u_to_upper(.i_clk(i_clk), .i_rst(i_rst));
```

Call this function every cycle

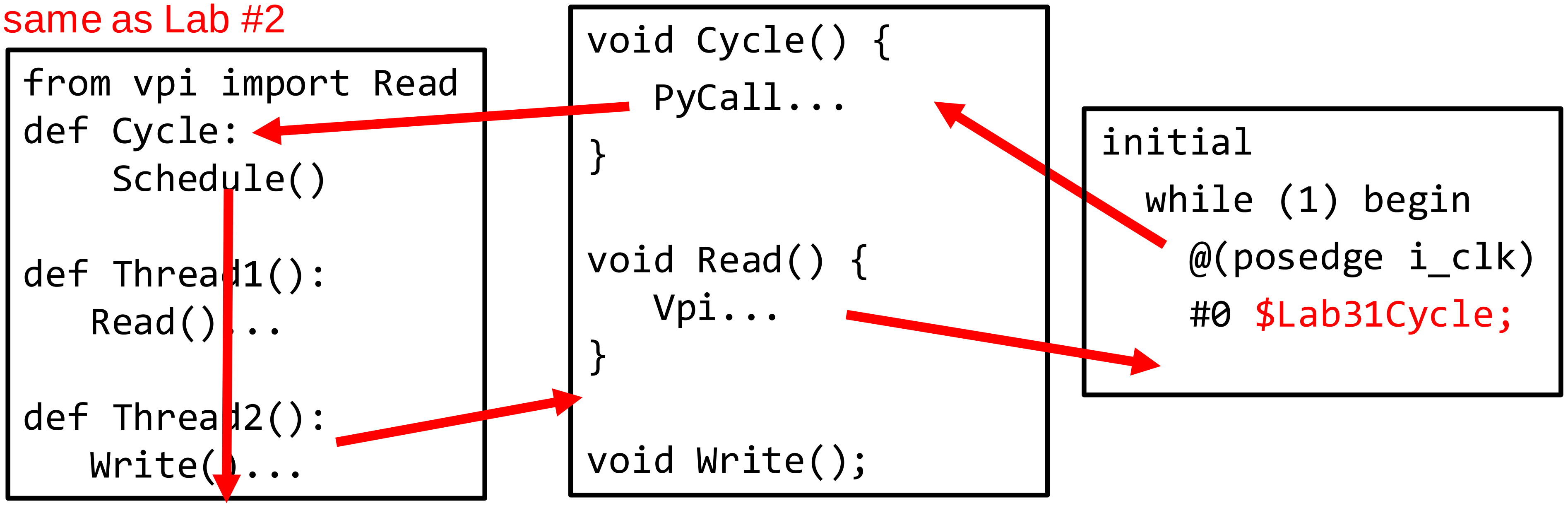
Connect all wires to C++ and Python

Your module

The Overall Architecture

Call Cycle from Verilog > C++ > Python

Schedule() is almost the same as Lab #2



```
from vpi import Read
def Cycle:
    Schedule()

def Thread1():
   Read()...

def Thread2():
   Write()...
```

```
void Cycle() {
   PyCall...
}

void Read() {
   Vpi...
}

void Write();
```

```
initial
  while (1) begin
    @(posedge i_clk)
    #0 $Lab31Cycle;
```

Read and Write is visible to Python

Read and Write can access Verilog wires (ncverilog +access+rw)

The C++ Part

```
from vpi import Read
def Cycle:
    Schedule()

def Thread1():
   Read()...

def Thread2():
   Write()...
```

```
void Cycle() {
   PyCall...
}

void Read() {
   Vpi...
}

void Write();
```

```
initial
  while (1) begin
    @(posedge i_clk)
    #0 $Lab31Cycle;
```

Write Verilog Signal

s_vpi_value hold a pointer to s_vpi_vecval.

```
s_vpi_value v;
s_vpi_vecval vecval;
s_vpi_time tm {vpiSimTime, 0, 0, 0};
v.format = vpiVectorVal;
vecval.bval = 0;
v.value.vector = ???;
```

To delay the write until all read are done

Parse the Write(1, 2) in Python

```
PyArg_ParseTuple(args, "II", &valid, &data)
```

Write 0 to i_valid

TODO: write the valid and data to Verilog throught VPI

```
vecval.aval = 0;
vpi_put_value(v_i_valid, &v, &tm, vpiInertialDelay);
```

Read Verilog Signal

```
unsigned valid, data;
valid = 1; data = 2;
return Py_BuildValue("II", valid, data);
```

TODO here
Use VPI to get the Verilog value

Equals to return (1, 2) in Python

Call Function upon Each Cycle

- How to call the "def Cycle():" in Python?
- Read the Python document, you have to call PyObject_CallFunction.
  - Callable = p_cycle_function (We prepare that for you).
  - Format = an empty string "", since Cycle() accepts no argument.

The Python Part

```
from vpi import Read
def Cycle:
    Schedule()

def Thread1():
   Read()...

def Thread2():
   Write()...
```

```
void Cycle() {
   PyCall...
}

void Read() {
   Vpi...
}

void Write();
```

```
initial
  while (1) begin
    @(posedge i_clk)
    #0 $Lab31Cycle;
```

Interfaces in Python

- You can use Python to control Verilog signals.

  -
    ```
    import lab31_vpi as V
    V.WriteBus(1, 100)
    V.ReadBus() # return 1, 20
    ```
- And this make this thread wait for a cycle
  - `yield`
- This is our test data.
  - `TEST_STRING = "JUST Monika, Hello moNIka"`
  - `GOLD_STRING = "JUST MONIKA, HELLO MONIKA"`

Cycle Function

```
def CycleGenerator():
    from itertools import zip_longest, repeat
    yield from zip_longest(Write(), Check())
    yield from repeat(None)

CycleObject = CycleGenerator()

def Cycle():
    ???
```

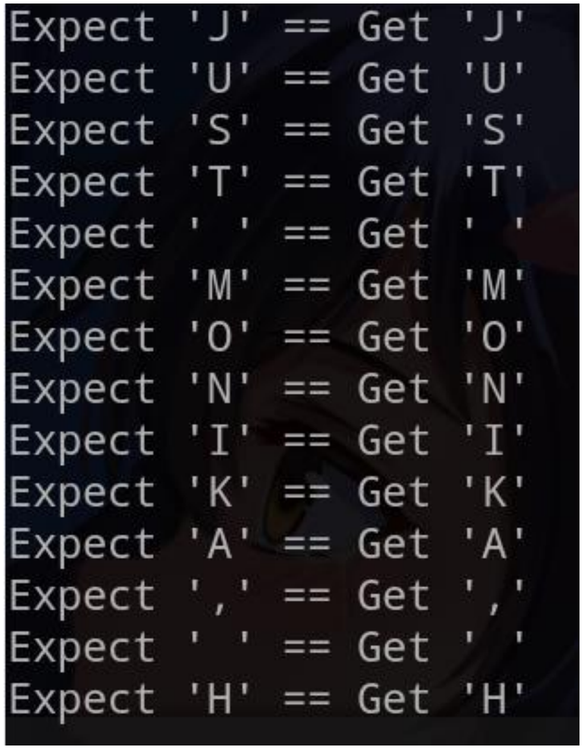
Infinite loop

The actual testbench.
You should implement them!
(You should be familiar with that in Lab #2)

TODO, advance the CycleObject generator here (HOW?)

A Sample Output

- In Write you call V.WriteBus every cycle to drive the input ports of module.
- In Check you call V.ReadBus to read whether the datum is valid and value is correct.
- Remember to yield to wait for a cycle in both of them.



```
Expect 'J' == Get 'J'
Expect 'U' == Get 'U'
Expect 'S' == Get 'S'
Expect 'T' == Get 'T'
Expect ' ' == Get ' '
Expect 'M' == Get 'M'
Expect 'O' == Get 'O'
Expect 'N' == Get 'N'
Expect 'I' == Get 'I'
Expect 'K' == Get 'K'
Expect 'A' == Get 'A'
Expect ',' == Get ','
Expect ' ' == Get ' '
Expect 'H' == Get 'H'
```

Example #2
Use Existing Frameworks

Python is Good, But...

- The wire names are hard-coded in C++.
  - We have to modify that every time.
- We have to implement every basic protocol.
  - Can we use the existing "transactors"?
- Co-simulation frameworks
  - cocotb is useful.
  - But there's also nicotb developed by TA that provide the same functionalities.
    - Surely I will use this as an example /lol/.

About Example #2

- No need to write C++!
- It requires a different testbench to run, but you can almost use the same testbenches across different simulations.

Testbench Set-Up (Verilog)

```
`Pos(rst_out, rst)
`PosIf(ck_ev, clk, rst)
always #1 clk = ~clk;
initial begin
    $fsdbDump...
    clk = 0; rst = 1;
    #1 $NicotbInit();
    ....
    #1000
    $NicotbFinal();
    $finish;
end

ToUpper u_to_upper(.i_clk(clk), .i_rst(rst));
```

Testbench Set-Up (Python)

```
iv, ic, ov, oc = CreateBuses([
    (("u_to_upper", "i_valid"),),
    (("u_to_upper", "i_char"),),
    (("u_to_upper", "o_valid"),),
    (("u_to_upper", "o_char"),),
])
```

Connect the wire through the API, not hard-coded

```
RegisterCoroutines([
    main(),
])
```

You don't to write the scheduler now. Just let the scheduler know to schedule it.

Use The Transactor in main()

This is the transactor for input ports.

```
master = OneWire.Master(iv, ic, ck_ev)
value = master.values
value.i_char[0] = 100
yield from master.Send(value)
```

The transactor provide good functions for operating the ports without knowing about the low-level wires.

Yet another transactor for output ports.

```
slave = OneWire.Slave(ov, oc, ck_ev, callbacks=[Check])
def Check(slave_data):
    value = slave_data.values
    print(value.o_char[0])
```

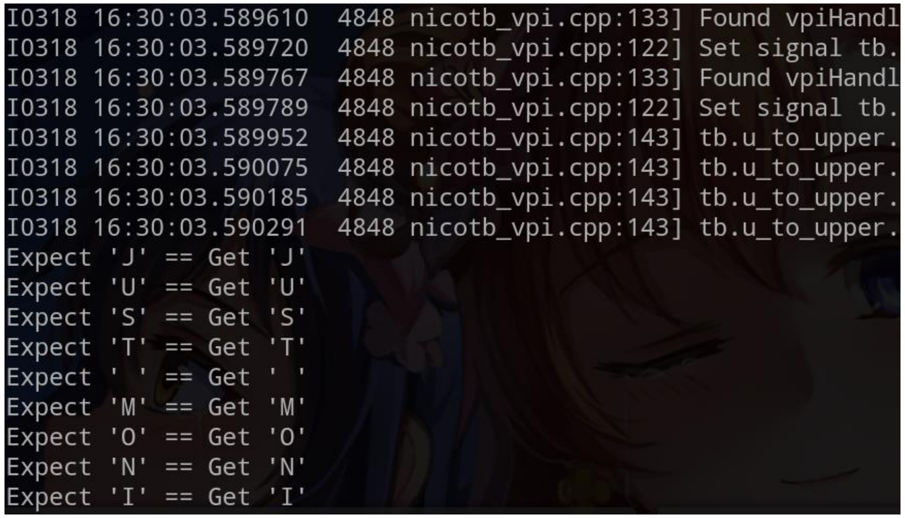
You can access the value in this way

callback ~ SC_METHOD, which is called every time a valid datum is observed at the output.

Hint: You will need these Python functions.
ord("A") == 65
chr(65) == "A"

A Sample Output

- Almost the same with Example #1
- The framework itselves also output some lines.



```
I0318 16:30:03.589610  4848 nicotb_vpi.cpp:133] Found vpiHandl
I0318 16:30:03.589720  4848 nicotb_vpi.cpp:122] Set signal tb.
I0318 16:30:03.589767  4848 nicotb_vpi.cpp:133] Found vpiHandl
I0318 16:30:03.589789  4848 nicotb_vpi.cpp:122] Set signal tb.
I0318 16:30:03.589952  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590075  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590185  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590291  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
Expect 'J' == Get 'J'
Expect 'U' == Get 'U'
Expect 'S' == Get 'S'
Expect 'T' == Get 'T'
Expect ' ' == Get ' '
Expect 'M' == Get 'M'
Expect 'O' == Get 'O'
Expect 'N' == Get 'N'
Expect 'I' == Get 'I'
```

Assignment #3

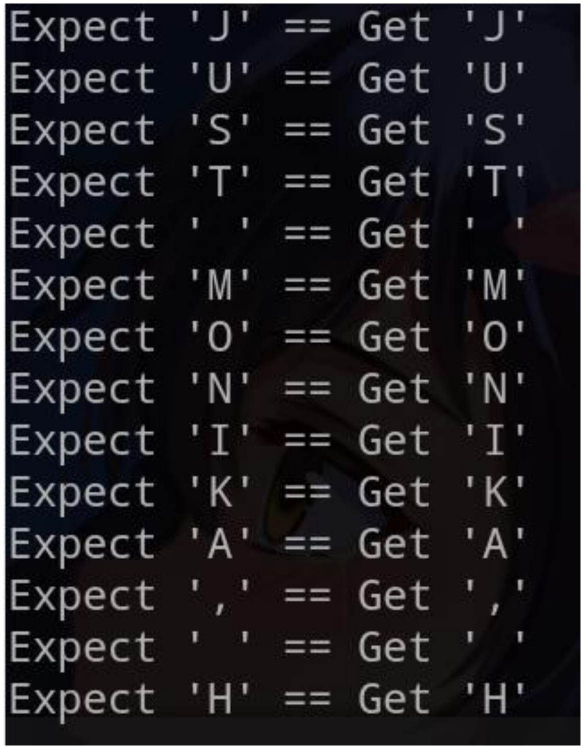
Requirements

Take-Home HW (Example #0)

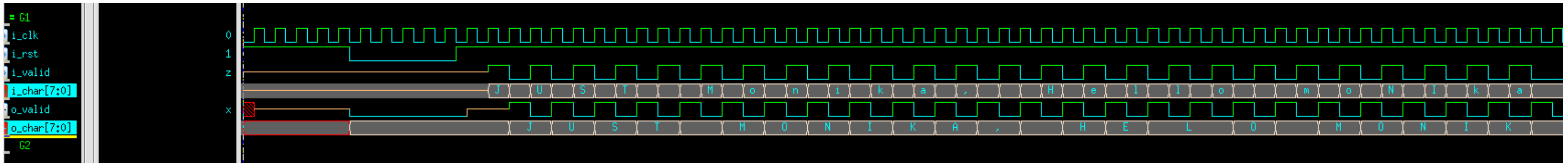
- Implement the RTL module that you will use later.
- Submission:
  - lab31_design.sv

Take-Home HW (Example #1)

- Command to run the lab
  - tool 2 (only type once every time you login)
    make lab31
- Note: your submission is considered invalid
  if it outputs something like Expect 'X' != Get 'Y'!
- Submission:
  - lab31_py.py
  - lab31.cpp

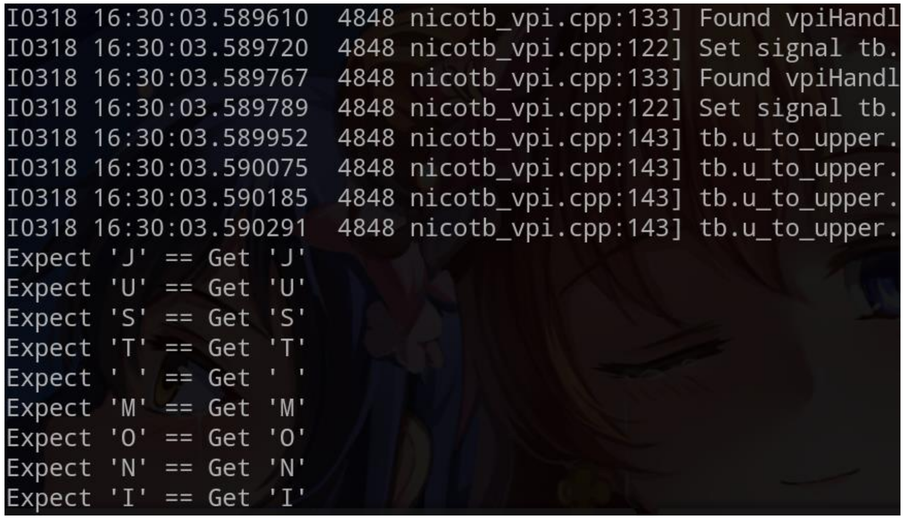


```
Expect 'J' == Get 'J'
Expect 'U' == Get 'U'
Expect 'S' == Get 'S'
Expect 'T' == Get 'T'
Expect ' ' == Get ' '
Expect 'M' == Get 'M'
Expect 'O' == Get 'O'
Expect 'N' == Get 'N'
Expect 'I' == Get 'I'
Expect 'K' == Get 'K'
Expect 'A' == Get 'A'
Expect ',' == Get ','
Expect ' ' == Get ' '
Expect 'H' == Get 'H'
```

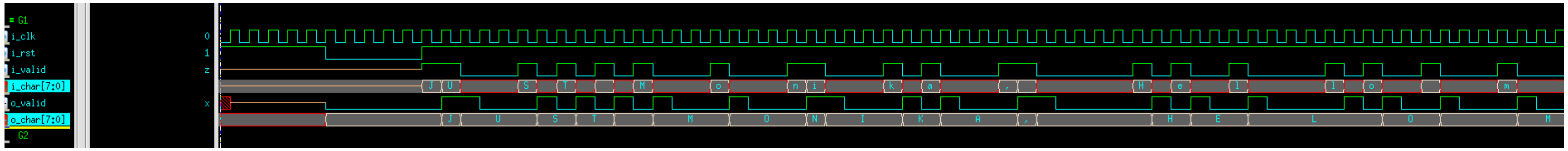

G1
i_clk 0
i_rst 1
i_valid z
i_char[7:0] J U S T M o n i k a , H e l l o m o N I k a
o_valid x
o_char[7:0] J U S T M O N I K A , H E L O M O N I K
G2

Take-Home HW (Example #2)

- Command to run the lab
  - make lab32
- The framework provides randomness and asserts X when signals are not valid.
- Submission
  - lab32_py.py



```
I0318 16:30:03.589610  4848 nicotb_vpi.cpp:133] Found vpiHandl
I0318 16:30:03.589720  4848 nicotb_vpi.cpp:122] Set signal tb.
I0318 16:30:03.589767  4848 nicotb_vpi.cpp:133] Found vpiHandl
I0318 16:30:03.589789  4848 nicotb_vpi.cpp:122] Set signal tb.
I0318 16:30:03.589952  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590075  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590185  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
I0318 16:30:03.590291  4848 nicotb_vpi.cpp:143] tb.u_to_upper.
Expect 'J' == Get 'J'
Expect 'U' == Get 'U'
Expect 'S' == Get 'S'
Expect 'T' == Get 'T'
Expect ' ' == Get ' '
Expect 'M' == Get 'M'
Expect 'O' == Get 'O'
Expect 'N' == Get 'N'
Expect 'I' == Get 'I'
```



Grading Rule

- Example #0 (10%)
- Example #1 (45%)
- Example #2 (45%)