MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 4, Issued: Tuesday, Feb. 26

Overview of this week's work

In software lab

- Work through the software lab.
- Submit whatever work you have finished at the end of the lab into the tutor.

Before the start of your design lab on Feb 28 or 29

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section PS.5.2.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Take the nanoquiz in the first 15 minutes; don't be late.
- Work through the design lab with a partner, and take good notes on the results of your work.

At the beginning of your next software lab on Mar 4 or 5

- Submit online tutor problems in section PS.5.3.
- Submit written solutions to software lab, and to all the design lab questions. All written work must conform to the homework guidelines on the web page.

- **You will need SoaR in this software lab, so if you don't have SoaR installed on your own machine, use a lab laptop or Athena station.**
- **Get the lab4 files via `athrun 6.01 update` or from the home page. We have given you several that have `Skeleton` in the name. You should make a copy of those files and rename them to remove the `Skeleton`. You will get failing `imports` otherwise.**

Software Lab: Combinations of state machines

In class we discussed three methods of making new state machines out of old ones: serial composition, parallel composition, and feedback composition. Below (and in the file `SimpleSMSkeleton.py`) is the skeleton of the `SerialSM` class. We've provided the constructor method, which take state machines as input and construct a new, composite state machine.

```
class SerialSM:
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2

    # Be careful to keep the timing consistent.  The input to m2 has
    # to be the current output of m1, not the output after it is
    # stepped.
    def step(self, input=None):
        # Your code here

    def currentOutput(self):
        # Your code here
```

Question 1: Write the `step` and `currentOutput` methods for a serial SM. Test it by combining two `incr` machines (the definition is in `SimpleSMSkeleton.py`). Draw a picture showing how the machines are connected. First, figure out what the result *ought* to be by filling out a table like this one (remember that, by definition, $output_1 = input_2$:

step	$input_1$	$state_1$	$output_1$	$state_2$	$output_2$
0					
1					
2					
3					

Then test to be sure your machine is doing the right thing.

Question 2: Compare the results of composing a `sum` machine (defined in `SimpleSMSkeleton.py`), which outputs the sum of all of its inputs so far with an `incr` machine. Predict what the result ought to be by filling out a table like this one:

step	$input_1$	$state_1$	$output_1$	$state_2$	$output_2$
0					
1					
2					
3					

Then be sure you're getting the right result.

Feedback The following class defines a feedback state machine. Its `__init__` method takes a state machine, `sm`, and returns a state machine with no inputs, which consists of `sm` with its output fed back to its input. We'll define the output of the feedback machine to be the same as the output of `sm`.

```
class FeedbackSM:
    def __init__(self, sm):
        self.m = sm

    def step(self, input=None):
        return self.m.step(self.m.currentOutput())

    def currentOutput(self):
        return self.m.currentOutput()
```

Now, we can use this to couple two machines together, with the outputs of one machine serving as the inputs to the other, and vice versa.

```
def simulatorSM(m1, m2):
    return FeedbackSM(SerialSM(m1, m2))
```

Question 3: In the code file, you'll find the definition of `makeIncr`, which takes an initial state as an argument, and then thereafter updates its state to be its input plus 1. If we make the following coupled machine, we get a potentially surprising string of outputs:

```
>>> fizz = simulatorSM(makeIncr(10), makeIncr(100))
>>> run(fizz)
100
11
102
13
104
15
106
17
108
19
110
```

Explain why this happens. Draw a picture of the objects involved in this machine and say exactly what state variables they contain. Fill in a table describing the inputs, states, and outputs of each component machine at each step.