Sequential Circuits in Minispec
Lecture Goals

- Learn how to implement sequential circuits in Minispec
  - Design each sequential circuit as a module
  - Modules are similar to FSMs, but are easy to compose

- Explore the advantages of sequential logic over combinational logic
  - Sequential circuits can perform computation over multiple cycles → handle variable amounts of input and/or output and computations that take a variable number of steps
Reminder: State Elements

- D Flip-Flop (DFF): State element that samples its data input at the **rising edge** of the clock

- Common DFF enhancements:
  - Write-enable circuit to optionally capture new input value
  - Reset circuit to set initial value

- Register: Group of DFFs
  - Stores multi-bit values
Reminder: Finite State Machines

- Synchronous sequential circuits: All state kept in registers driven by the same clock
- This allows discretizing time into cycles and abstracting sequential circuits as finite state machines (FSMs)
- FSMs can be described with state-transition diagrams or truth tables
Problem: FSMs Don’t Compose

- **Key strategy:** Build large circuits from smaller ones
- **Problem:** Wiring up FSMs can introduce combinational cycles

```verilog
fsm Inner;
  Reg r;
  out = in ^ r.q;
  ...

fsm Outer;
  Inner s;
  s.in = !s.out;
  ...
```

- Most hardware description languages work this way
  - Just wire up FSMs however you want
  - Allows combinational cycles and many other bugs!
    - If curious, read “Verilog is weird”, Dan Luu, 2013
Minispec modules add some structure to FSMs to make them composable.

Modules separate the combinational logic to compute the outputs and the next state:
- **Methods** compute outputs
- **Rules** compute next state
- Methods and rules use separate input wires (method *arguments* vs rule *inputs*)
Reminder: Two-Bit Counter
Two-Bit Counter in Minispec

```
module TwoBitCounter;
  Reg#(Bit#(2)) count(0);

  method Bit#(2) getCount = count;

input Bool inc;

rule increment;
  if (inc)
    count <= count + 1;
endrule
endmodule
```

- Instantiates a 2-bit register named `count` with initial value 0
- `getCount` method produces the output
- `increment` rule computes the next state: if `inc` input is True, updates `count` to `count + 1`

Rules execute automatically every cycle.
The Reg#(T) Module

- Reg#(T) is a register of values of type T
  - e.g., Reg#(Bool) or Reg#(Bit#(16)), not Reg#(16)

- Register writes use a special register assignment operator: <=
  - e.g., count <= count + 1, not count = count + 1

- <= has two key differences from =
  1. = assigns to variable immediately, but <= updates register at the end of the cycle
  2. Registers can be written at most once per cycle
module FourBitCounter;
  TwoBitCounter lower;
  TwoBitCounter upper;

method Bit#(4) getCount =
  {upper.getCount, lower.getCount};

input Bool inc;

rule increment;
  lower.inc = inc;
  upper.inc = inc && (lower.getCount == 3);
endrule
endmodule

Instantiates a TwoBitCounter submodule named lower (stores lower 2 bits of our count)

increment rule sets the inputs of lower and upper submodules

Increment upper counter when lower counter wraps around from 3 to 0
Module Components

1. **Submodules**, which can be registers or other user-defined modules to allow composition of modules
2. **Methods** produce outputs given some input arguments and the current state
3. **Rules** produce the next state and submodule inputs given some external inputs and the current state
4. **Inputs** represent external inputs controlled by the enclosing module
In 6.191 we will only use strict hierarchical composition, which obeys two restrictions:

1. Each module interacts only with its own submodules
2. Methods do not read inputs (only their own arguments)

These conditions guarantee two nice properties:

1. It is impossible to get combinational cycles
2. Very simple semantics: System behaves as if rules fire sequentially, outside-in (i.e., first the outermost module, then its submodules, and so on)

Minispec supports non-hierarchical composition (with similar guarantees), but we will not use it
Simulating and Testing Modules

- Modules can be simulated/tested with testbenches
  - Another module that uses the tested module as a submodule
  - Drives its inputs through a sequence of test cases
  - Checks that outputs are as expected

- System functions let testbench modules output results and control simulation
  - `$display` to print output
  - `$finish` to terminate simulation
  - System functions have no hardware meaning, are ignored when synthesized

```verilog
module FourBitCounterTest;
    FourBitCounter counter;
    Reg#(Bit#(6)) cycle(0);

    rule test;
        // Increment only on odd cycles
        counter.inc = (cycle[0] == 1);

        // Print the current count
        $display("[cycle %d] getCount = %d", cycle, counter.getCount);

        // Terminate after 32 cycles
        cycle <= cycle + 1;
        if (cycle >= 32) $finish;
    endrule
endmodule
```
Multi-Cycle Computations
Time is More Flexible Than Space

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

- Example: Build a circuit that adds two numbers of arbitrary length
  - Combinational: Can’t, inputs/outputs must have fixed width
  - Sequential: Trivial, e.g., add one bit per cycle:
Euclid’s algorithm efficiently computes the greatest common divisor (GCD) of two numbers:

```python
def gcd(a, b):
    x = a
    y = b
    while x != 0:
        if x >= y:  # subtract
            x = x - y
        else:       # swap
            (x, y) = (y, x)
    return y
```

Example: \( \text{gcd}(15, 6) \)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>subtract</th>
<th>subtract</th>
<th>swap</th>
<th>subtract</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Takes a variable number of steps
- Approach: Build a sequential circuit that performs one iteration of the while loop per cycle
GCD Circuit

\[
\begin{align*}
\text{def } \text{gcd}(a, b): \\
x &= a \\
y &= b \\
\text{while } x \neq 0: \\
\quad \text{if } x \geq y: \\
\quad \quad x &= x - y \\
\quad \text{else:} \\
\quad \quad (x, y) &= (y, x) \\
\text{return } y
\end{align*}
\]
General Recipe for Multi-Cycle Circuits
Deriving a sequential circuit from code

1. Specify circuit interface: Data inputs and outputs; also, control signals (in GCD: a, b, result; start, isDone)
2. Implement state elements: Every value that spans multiple cycles needs to be in a register (x, y)
3. Implement combinational logic that performs all needed operations on register values (x==0, x>=y, x-y)
4. Implement a mux at the input of each register, and determine the possible values of each register case by case. Use each of these values as an input to the muxes (in GCD, four possible next values → 4-input muxes)
5. Derive the mux select signals; may require some combinational logic (sel)
6. Derive outputs (result, isDone)
GCD in Minispec
First version

typedef Bit#(32) Word;
module GCD;
    Reg#(Word) x(1);
    Reg#(Word) y(0);
    input Bool start;
    input Word a;
    input Word b;
rule gcd;
    if (start) begin
        x <= a;  y <= b;
    end else if (x != 0) begin
        if (x >= y) begin // subtract
            x <= x - y;
        end else begin // swap
            x <= y;  y <= x;
        end
    end
endrule
method Word result = y;
method Bool isDone = (x == 0);
endmodule

Poor interface: Several inputs and outputs are closely coupled
• New GCD computation is started by setting start input to True and passing arguments through inputs a and b
• Several cycles later, the module signals that it has finished by having isDone return True; only then, the result method returns the correct result for gcd(a,b)
Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. Why?
    - e.g., may forget to check isDone and read wrong result!
  - Tedious to use. Why?
    - e.g., if start is False, we still have to set the a and b inputs, even though they are not used!

- To design good interfaces,
  group related inputs and outputs
  - In our case, GCD should have:
    - A single output that is either invalid or a valid result
    - A single input that is either no arguments or arguments
  - This requires we learn about one last type...
The Maybe Type

- **Maybe#(T)** represents an **optional** value of type T
  - Either Invalid and no value, or Valid and a value

- Possible implementation: A value + a valid bit
  ```
typedef struct { Bool valid; T value; } Maybe#(type T);
  ```
  - Although we could implement our own, optional values are so common that Maybe#(T) has a few built-in operations

```plaintext
Maybe#(Word) x = Invalid;  // no need to give value!
Maybe#(Word) y = Valid(42); // must specify a value

if (isValid(y))  // check validity
    Word z = fromMaybe(?, y); // extract valid value
```
Improved GCD Module
Using Maybe Types

typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
  input Maybe#(GCDArgs) in;
rule gcd;
  if (isValid(in)) begin
    let args = fromMaybe(?, in);
    x <= args.a; y <= args.b;
  end else if (x != 0) begin
    if (x >= y) begin // subtract
      x <= x - y;
    end else begin // swap
      x <= y; y <= x;
    end
  end
endrule
method Maybe#(Word) result =
  (x == 0)? Valid(y) : Invalid;
endmodule

Single input and output:
• New GCD computation is started by setting a Valid input in (which always includes a and b)
• When GCD computation finishes, result becomes a Valid output
Summary

- Modules implement FSMs in a composable way
  - Extra structure to FSMs: Combinational logic split into rules (produce next state) and methods (produce outputs)
  - Clean hierarchical composition: No combinational cycles, system behaves as if rules execute outside-in

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

- To build simple, easy-to-use module interfaces, group related inputs and outputs
Thank you!

Next lecture:
Arithmetic Pipelines