Sequential Circuits in Minispec

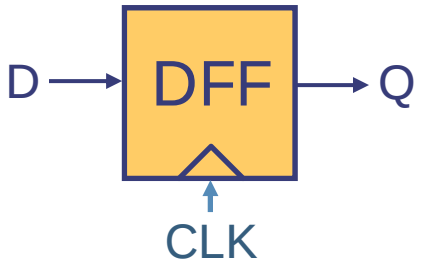
Lecture Goals

- 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

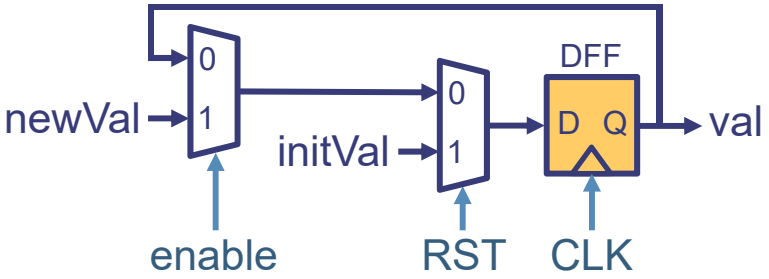
- Explore the advantages of sequential logic over combinational logic
  - Sequential circuits can perform computation over multiple cycles → handle variable amounts of input and/or output and computations that take a variable number of steps

Reminder: Sequential Circuits
State Elements

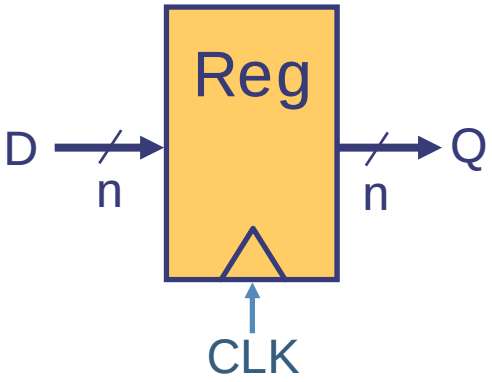
- D Flip-Flop (DFF): State element that samples its data (D) input at the rising edge of the clock



- Common DFF enhancements:
  - Reset circuit to set initial value
  - Write-enable circuit to optionally retain current value

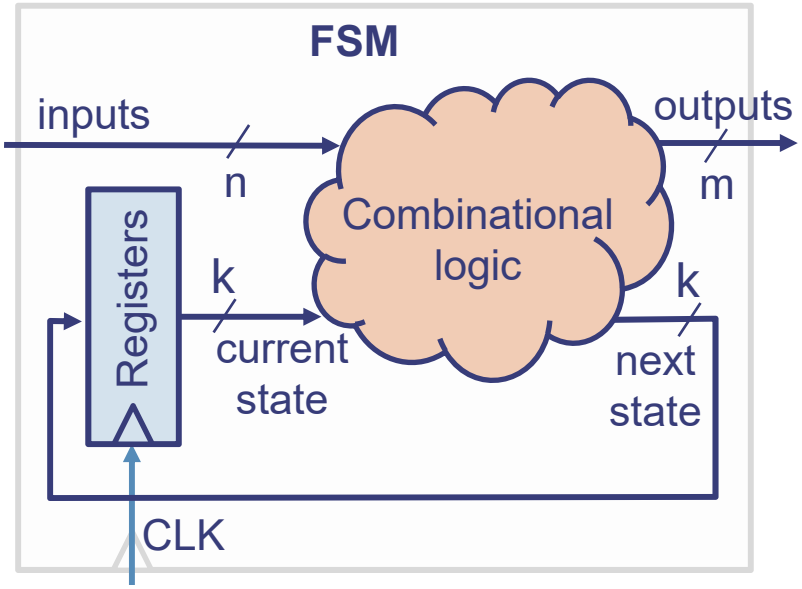


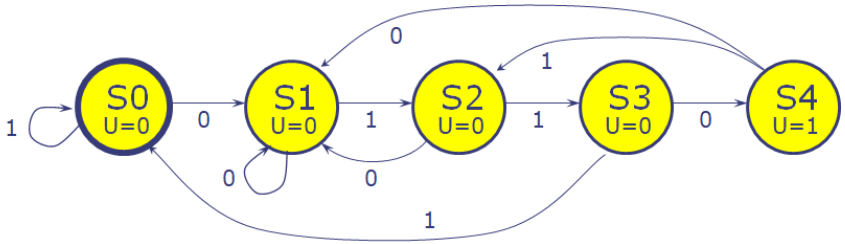
- Register: Group of DFFs
  - Stores multi-bit values



Reminder: Sequential Circuits
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



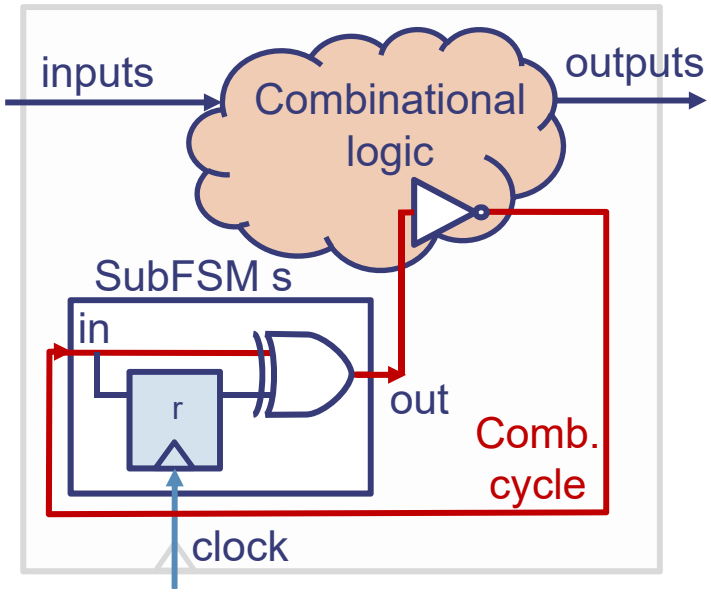


Problem: FSMs Don't Compose

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

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

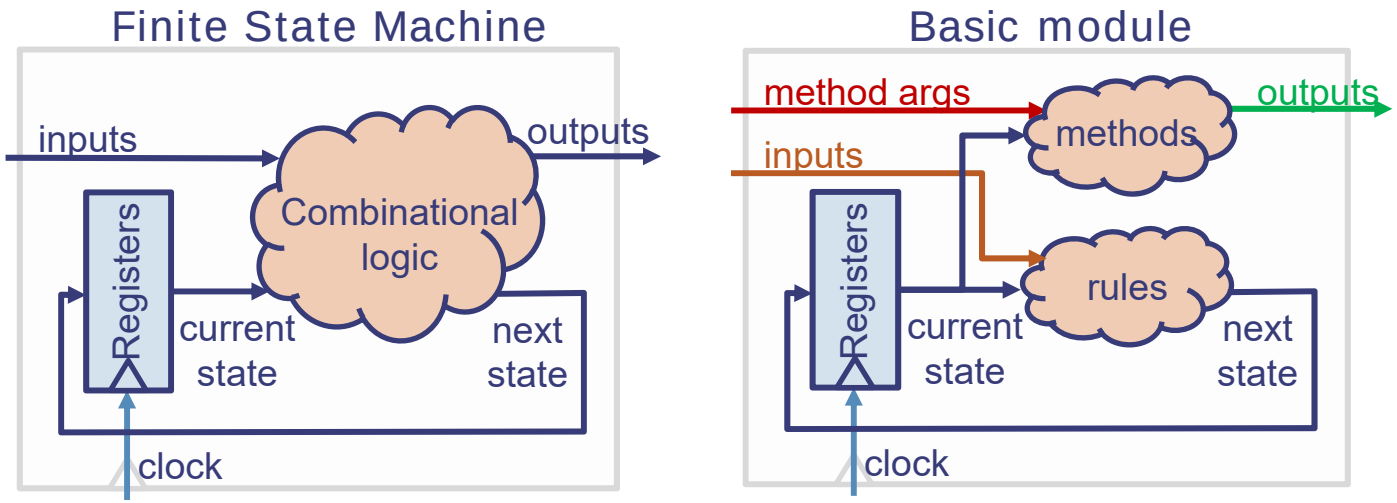
```
fsm Outer;
  Inner s;
  s.in = !s.out;
  …
```



- Most hardware description languages work this way
  - Just wire up FSMs however you want!
  - Got a cycle? ¯_(ツ)_/¯
    - If curious, read "Verilog is weird", Dan Luu, 2013

Modules

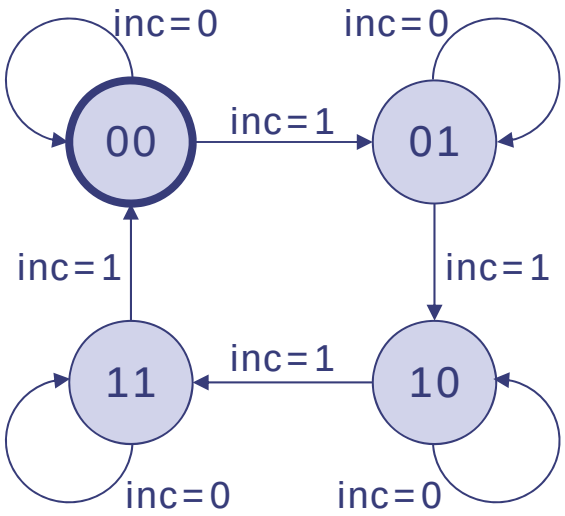
- Minispec modules add some structure to FSMs to make them composable

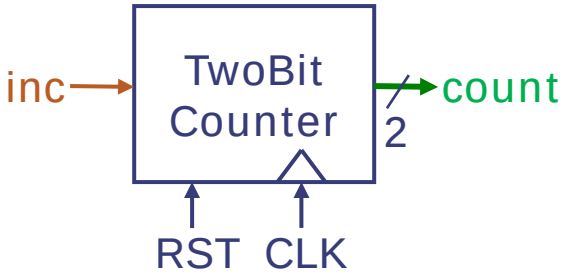


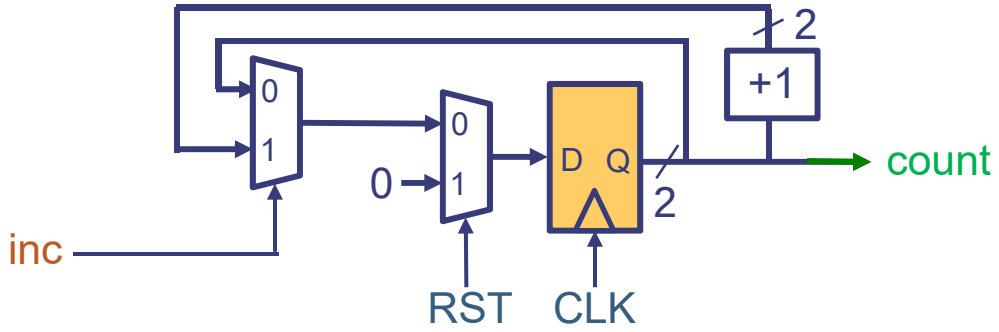
- Modules separate the combinational logic to compute the outputs and the next state
  - Methods compute outputs
  - Rules compute next state
  - Methods and rules use separate inputs

Reminder: Two-Bit Counter

Prev State	NextState	
q1q0	inc = 0	inc = 1
00	00	01
01	01	10
10	10	11
11	11	00







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 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 with =
  1. = assigns to variable immediately, but
     <= updates register at the end of the cycle
  2. Registers can be written at most once per cycle

Composing Modules

```
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
```

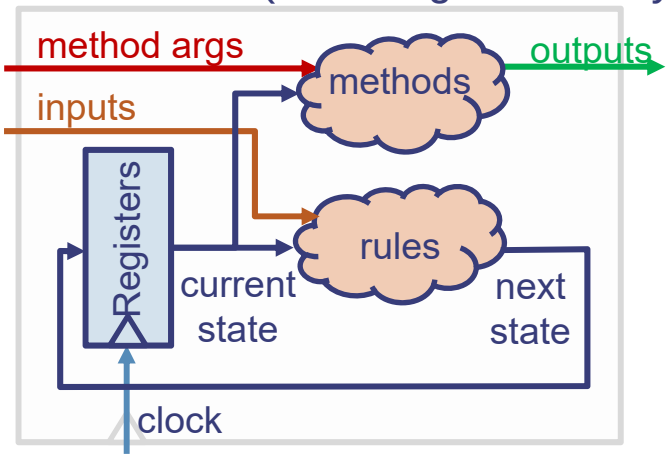
Instantiates a TwoBitCounter submodule named lower (stores lower 2 bits of our count)

increment rule sets the inputs of lower and upper submodules

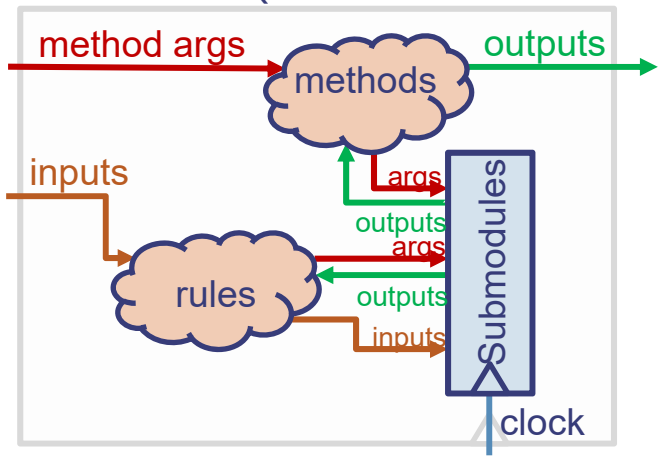
Increment upper counter when lower counter rolls over from 3 to 0

Module Components

Basic module (with registers only)



General module (with other submodules)



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

Modules Compose Cleanly

- In 6.004 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 tested module as a submodule
  - Drives its inputs through a sequence of test cases
  - Checks that outputs are as expected

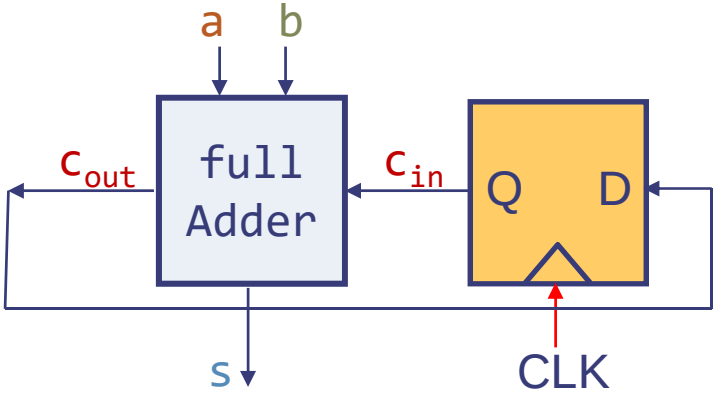
- System functions let testbench modules output results and control simulation
  - $display to print output
  - $finish to terminate simulation
  - System functions have no hardware meaning, are ignored when synthesized

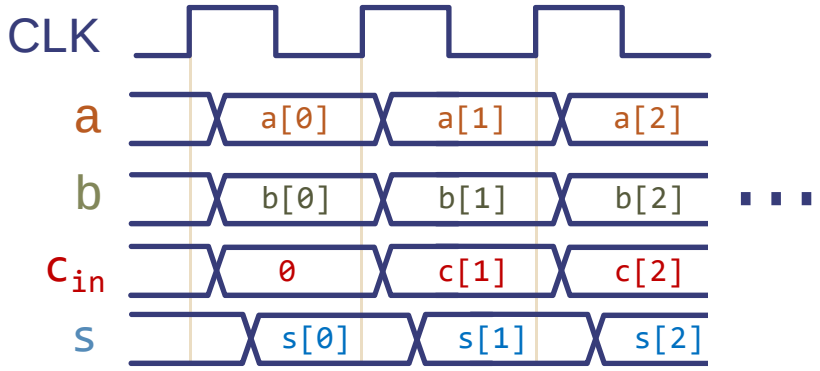
```
module FourBitCounterTest;
 FourBitCounter counter;
 Reg#(Bit#(6)) cycle(0);

 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, add one digit per cycle:





Example: GCD

- Euclid's algorithm efficiently computes the greatest common divisor (GCD) of two numbers:

```
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
```

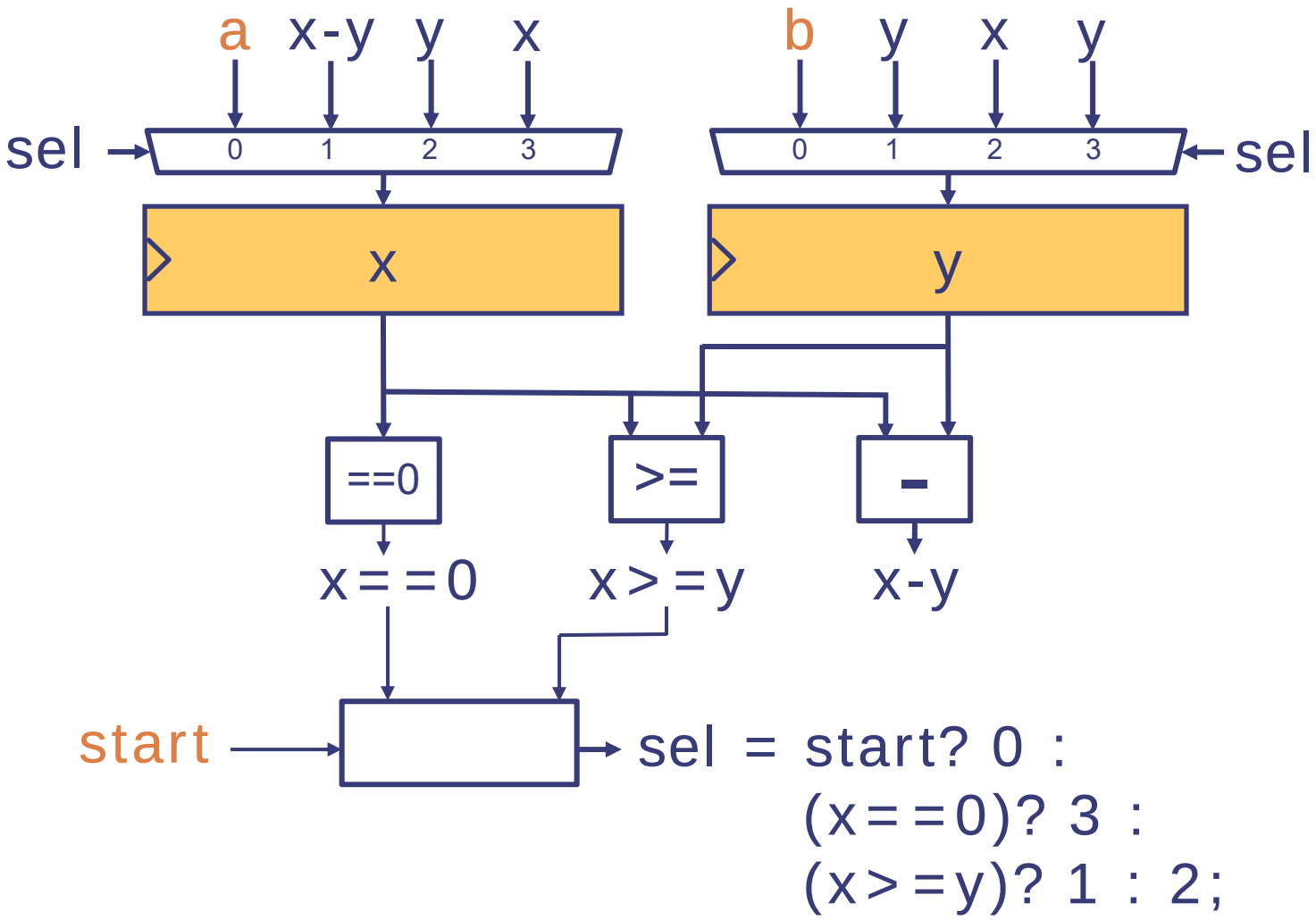
Example: gcd(15, 6)

x: 15	y: 6	
9	6	*subtract*
3	6	*subtract*
6	3	*swap*
3	3	*subtract*
0	3	*subtract*

result

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

GCD Circuit


a
x-y
y
x
b
y
x
y
sel
0
1
2
3
0
1
2
3
sel
x
y
==0
>=
-
x==0
x>=y
x-y
start
sel = start? 0 :
(x==0)? 3 :
(x>=y)? 1 : 2;

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
```

New GCD computation is started by setting start input to True and passing arguments through inputs a and b

Several cycles later, the module will signal it has finished through isDone. Then, the result gcd(a,b) will be available through the result method.

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isValid 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

```
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
```

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