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
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.
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
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)
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: FSMs Don’t Compose

- Key strategy: Build large circuits from smaller ones
- Problem: Wiring up FSMs can introduce combinational cycles
Problem: FSMs Don’t Compose

- Key strategy: Build large circuits from smaller ones
- Problem: Wiring up FSMs can introduce combinational cycles
Problem: FSMs Don’t Compose

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

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

![Diagram showing combinational logic and FSM composition]
Problem: FSMs Don’t Compose

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

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

fsm Outer;
  Inner s;
  s.in != s.out;
  ...
```
Problem: FSMs Don’t Compose

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

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

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

- Most hardware description languages work this way
  - Just wire up FSMs however you want!
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? 廨__(ツ)___廨
Problem: FSMs Don’t Compose

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

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

clock
```

```plaintext
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?  
    ```text
    \(_/\(_/\(_/\(_/\(_/
    ```
    - If curious, read “Verilog is weird”, Dan Luu, 2013
Modules

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

<table>
<thead>
<tr>
<th>Prev State</th>
<th>NextState</th>
<th>NextState</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td>inc = 0</td>
<td>inc = 1</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>
Reminder: Two-Bit Counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>NextState inc = 0</th>
<th>NextState inc = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

[Diagram showing state transitions for a two-bit counter, where states 00, 01, 10, and 11 are connected with transitions labeled inc=0 and inc=1.]
Reminder: Two-Bit Counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>NextState inc = 0</th>
<th>NextState inc = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

TwoBit Counter

inc → count

RST, CLK
Reminder: Two-Bit Counter

<table>
<thead>
<tr>
<th>Prev State</th>
<th>NextState inc = 0</th>
<th>NextState inc = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1q0</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

TwoBit Counter

inc → count
RST  CLK

inc=0
inc=1
inc=0
inc=1
inc=0
inc=1
inc=1
inc=0
inc=1
inc=1
inc=0
inc=1
inc=0
inc=0

00
01
11
10

0
1
2

+1
2

0
1

D Q

RST  CLK

inc

count
Two-Bit Counter in Minispec

module TwoBitCounter;

dendmodule
Two-Bit Counter in Minispec

module TwoBitCounter;
  Reg#(Bit#(2)) count(0);
endmodule
Two-Bit Counter in Minispec

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

Instantiates a 2-bit register named count with initial value 0
Two-Bit Counter in Minispec

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

    method Bit#(2) getCount
        = count;

endmodule
```

Instantiates a 2-bit register named count with initial value 0
module TwoBitCounter;
    Reg#(Bit#(2)) count(0);

    method Bit#(2) getCount = count;

endmodule

Instantiates a 2-bit register named count with initial value 0

getCount method produces the output
Two-Bit Counter in Minispec

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

    method Bit#(2) getCount = count;

    input Bool inc;

endmodule
```

- Instantiates a 2-bit register named `count` with initial value 0
- `getCount` method produces the output
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
defines the method `getCount`
produces the output
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

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)
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
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
module FourBitCounter;

endmodule
Composing Modules

module FourBitCounter;
  TwoBitCounter  lower;
  TwoBitCounter  upper;
endmodule
Composing Modules

module FourBitCounter;
  TwoBitCounter lower;
  TwoBitCounter upper;
endmodule

Instantiates a TwoBitCounter submodule named lower (stores lower 2 bits of our count)
module FourBitCounter;
    TwoBitCounter lower;
    TwoBitCounter upper;

method Bit#(4) getCount =
    {upper.getCount, lower.getCount};
endmodule
module FourBitCounter;
    TwoBitCounter lower;
    TwoBitCounter upper;

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

input Bool inc;

endmodule
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
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

---

**Highlights**

- Composing modules: `FourBitCounter` is composed of `TwoBitCounter`
- Method `getCount` returns combined counts
- Increment rule updates submodules based on flag `inc` and current count state.
Module Components

Basic module (with registers only)

- Registers
- Current state
- Next state
- Clock
- Inputs
- Method args
- Methods
- Rules
- Outputs
Module Components

Basic module (with registers only)

- Inputs
- Registers
- Current state
- Next state
- Clock
- Rules
- Methods
- Outputs

General module (with other submodules)

- Inputs
- Outputs
- Arguments
- Clock
- Submodules
- Rules
- Methods
- Outputs
- Arguments
- Inputs
1. Submodules, which can be registers or other user-defined modules to allow composition of modules
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
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
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 6.004 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)
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:
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
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)
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
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
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

```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
```
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

```verbatim
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
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
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
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
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:
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:
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:
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:
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:

```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: GCD

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

\[
x: 15 \quad y: 6
\]
Example: GCD

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

\[
\begin{array}{cccc}
& x: 15 & y: 6 & \text{subtract} \\
9 & 6 &
\end{array}
\]
Example: GCD

- 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: gcd(15, 6)

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

subtract

```
9
3
```

October 10, 2019

MIT 6.004 Fall 2019
Example: GCD

- 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: gcd(15, 6)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
Example: GCD

- 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: gcd(15, 6)

<table>
<thead>
<tr>
<th>x:</th>
<th>y:</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>
| 9  | 6  | subtract
| 3  | 6  | subtract
| 6  | 3  | swap
| 3  | 3  | subtract
Example: GCD

- 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: `gcd(15, 6)`

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Subtract</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

October 10, 2019
Example: GCD

- 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: gcd(15, 6)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

result
Example: GCD

- 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: $\text{gcd}(15, 6)$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15</td>
</tr>
<tr>
<td>y</td>
<td>6</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
GCD Circuit

\[ \frac{x}{y} \]
GCD Circuit

\[ x \geq y = 0 \]

\[ x - y \]
GCD Circuit

Diagram showing a circuit for finding the greatest common divisor (GCD) of two numbers, x and y. The circuit includes selection logic (sel) and arithmetic operations such as comparison (==0, >=, -) to determine the GCD.
GCD Circuit

\[ a \] 
\[ b \]

\[ \text{sel} \rightarrow 0 \quad 1 \quad 2 \quad 3 \]

\[ > \]

\[ x \]

\[ \text{==0} \]

\[ x==0 \]

\[ >= \]

\[ x>=y \]

\[ - \]

\[ x-y \]

\[ \text{sel} \rightarrow 0 \quad 1 \quad 2 \quad 3 \]
GCD Circuit

\[ a \quad x-y \]
\[ b \quad y \]

\[ \text{sel} \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \]
\[ > \]
\[ x \]
\[ \rightarrow \]
\[ ==0 \]
\[ x==0 \]
\[ \rightarrow \]
\[ >= \]
\[ x>=y \]
\[ \rightarrow \]
\[ - \]
\[ x-y \]
GCD Circuit

\[ a \times - y \times y \]

\[ b \times y \times x \]

sel

\[ > \]

\[ x \]

\[ ==0 \]

\[ x==0 \]

\[ \geq \]

\[ x\geq y \]

\[ - \]

\[ x-y \]
GCD Circuit

\[ a \quad x-y \quad y \quad x \]

\[ b \quad y \quad x \quad y \]

\[ sel \]

\[ x \]

\[ ==0 \]

\[ x==0 \]

\[ >= \]

\[ x>=y \]

\[ - \]

\[ x-y \]
GCD Circuit

\[
\text{sel} = \text{start? 0 : (x==0)? 3 : (x>=y)? 1 : 2;}
\]
typedef Bit#(32) Word;
module GCD;
endmodule
typedef Bit#(32) Word;
module GCD;
    Reg#(Word) x(1);
    Reg#(Word) y(0);
endmodule
typedef Bit#(32) Word;
module GCD;
    Reg#(Word) x(1);
    Reg#(Word) y(0);
    input Bool start;
    input Word a;
    input Word b;
endmodule
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
endmodule
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
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.
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
Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
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!
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?*
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!
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
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
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
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
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

```c
typedef struct { Bool valid; T value; } Maybe\#(type T);
```
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
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

```c
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

```c
Maybe#(Word) x = Invalid; // no need to give value!
Maybe#(Word) y = Valid(42); // must specify a value
```
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);
endmodule
**Improved GCD Module Using Maybe Types**

```plaintext
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
    Reg#(Word) x(1);
    Reg#(Word) y(0);
    input Maybe#(GCDArgs) in;
endmodule
```

endmodule
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
endmodule
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
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)
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
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
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:
Design tradeoffs in sequential logic