Registers

- Registers are modules
- Need to be instantiated explicitly
- They're at the leaves of the module hierarchy
- Like all modules, registers have interfaces
- Register interfaces are parameterized types
  
  ```
  Reg #(Int#(32)) // interface to a register that contains Int#(32)
  Reg #(Bit#(16)) // interface to a register containing Bit#(16) values
  Reg #(State) // interface to a register containing a State value
  Reg #(Request) // interface to a register containing a Request value
  ```
- Registers are strongly-typed

Writing and reading registers

- The `Reg()` interface presents the methods to write and read from a register
  ```
  interface Reg#(type t);
  method Action _write(t a);
  method t _read;
  endinterface
  ```
- Any module register declares the `write` method as "store the value" and the `read` method as "return the value"

Instantiating registers follows standard module instantiation syntax
- The `mkReg()` module instantiates a register with a given reset value
- The initial value must, of course, have the correct type for the type of the register (else type-checking error)
- The `mkRegU` module instantiates a register with an unspecified reset value

```haskell
Reg #(Int#(32)) r1 <- mkReg (0); // Synchronously reset to 0
Reg #(Bit#(16)) r2 <- mkRegU; // unspecified initial value
Reg #(Request) r4 <- mkReg(Request { op: Load; addr: 0 });
```
Writing and reading registers

For ease of use:
- A register on the left of an assignment is treated as a write operation
- A register on the right side of an assignment is treated as a read operation

Reg #(Int#(32)) r1 <- mkReg(0);
...r1 <= 23;
   // is equivalent to r1._write(23);
...let a = r1;
   // is equivalent to let a = r1._read;

Rule

- All behavior in BSV is expressed using Rules
- A rule is a Declarative specification of a state transition
- A rule is an Action guarded by a Boolean condition
- Syntax:
  rule ruleName [( condition )];
  actions
  endrule [: ruleName]
- Over-simplified analogy to a Verilog “always” block:
  always @(posedge CLK)
  if cond begin
    actions
  end
  This analogy does not always hold.

Rules “Fire”

- Rules don’t control clocks
  - They only generate enable logic and muxing
  - You define all modules and state
- No clock or reset ??
  - those are wired directly to state elements.
- Firing means it is enabled and selected for this cycle.
- Rules always try to fire unless you or another rule tells it not to

When does a rule fire?

- Every cycle!
  - Unless you tell it not to (rule conditions)
  - Unless a child module tells it not to (ready conditions)
  - Unless a "more important" rule needs to fire instead (conflicts)
**Rule Conditions**

- Rule conditions are arbitrary expressions of type `Bool`.
- Such expressions are purely combinational.
- This is guaranteed by BSV's type-checking rules.
- If this condition is false, rule doesn't fire.
- No condition means default `True`.

```hs
rule rule1 (state == TRANSFER);
  ... rule actions ...
endrule : rule1

rule go (trigger == 1 && state != IDLE);
  ... rule actions ...
endrule : go
```

**Ready Conditions**

- Methods have “ready” signal.
- Classic examples: FIFOs.
  ```hs
 fifo.enq() − ready when !full
  fifo.first() − ready when !empty
  ```
- Ready can be always `True`.
- Ready signals are specified for each method in defining module.
- Rule doesn't fire unless all ready conditions are true.

```hs
rule rule1;
  // fifo.notFull;
  // fifo.notEmpty;
  ... if (cond)
  fifo.enq( 10 );
  let x = fifo.first();
  ...
endrule
endrule
```

**Conflicting Rules**

- A rule may not fire because it conflicts with other rules.
- Compiler will warn you.
- But what is a “conflict”?

```hs
rule rule1;
  x <= x + 1;
endrule

rule rule2;
  x <= x + 2;
endrule
```

**Rule Actions**

- The simplest rule action is a register assignment:
  `r <= ... expression ...`
- BSV uses `<=`, non-blocking assignment, which is a write action to a register.
- A rule body can contain multiple actions.
- There is no “sequencing” of actions in a rule body: all the actions happen simultaneously, in parallel.
- The following two rules are equivalent.

```hs
rule rule1 (st == T);
  valuea <= valuea + 1;
  mult <= valuea * 2;
endrule : rule1

rule rule2 (st == T);
  valuea <= valuea + 1;
  mult <= valuea * 2;
endrule : rule2
```
Because there is no sequencing of actions in a rule body (all the actions happen simultaneously, in parallel), it is meaningless to put conflicting actions in the same rule, as in the examples below:

```
rule rule3 (...);
valuea <= valuea + 1;
valuea <= valuea + 2;
...
endrule : rule3

rule rule4 (...);
fifo.enq (23);
fifo.enq (34);
...
endrule : rule4
```

The compiler will flag such errors.

The term “parallel composable” is used for actions that can be in the same rule body.

A rule also cannot contain resource conflicts, even if they are combinational.

For e.g., trying simultaneously to read two different registers in a register file using a single read-port.

The compiler will flag such errors:

```
rule rule5 (...);
let x = regFile.sel1(5);
let y = regFile.sel1(7);
...
endrule : rule5
```