For the recitation session for L07, instead of having tutorial problems, we will use a handout to go through various aspects in Bluespec when writing sequential circuits. The handout is attached to this cover page.
Guards vs Ifs
Take home problem

```plaintext
method Action enq(t x) if (!v);
  v <= True; d <= x;
endmethod

versus

method Action enq(t x);
  if (!v) begin v <= True; d <= x; end
endmethod
```

guard is !v; enq can be applied only if v is false

guard is True, i.e., the method is always applicable

if v is true then x would get lost

bad
Register is a primitive module in Bluespec

- Register implementation is defined outside the language

```plaintext
interface Reg#(type t);
    method Action _write(t x);
    method t _read;
endinterface
```

- A register is created using mkReg or mkRegU
  - Reg#(Bit#(32)) a <- mkReg(0);  creates a 32-bit reg with initial value 0
  - Reg#(Bit#(32)) b <- mkRegU;   creates an uninitialized 32-bit reg

- The guards of _write and _read are always true
  - The guard wires are not generated for registers

- Special syntax: We write
  - x <= e instead of x._write(e)
  - x instead of x._read in expressions

Rule

A module may contain rules

```plaintext
rule foo;
    x_{t+1} <= x - 1;
    y_{t+1} <= f(x_{t});
endrule
```

- A rule is a collection of actions, which invoke methods
  - register reads and writes are methods!
- All actions in a rule execute in parallel
- A rule can execute any time and when it executes all of its actions must execute atomicity
Parallel Composition of Actions & Double-Writes

- Parallel composition, and consequently a rule or a method containing it, is illegal if a double-write possibility exists.
- The Bluespec compiler rejects a program if there is any possibility of a double write in a rule or method.

GCD with Guards

```verbatim
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy <- mkReg(False);

  rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
  endrule

  method Action start(Bit#(32) a, Bit#(32) b) \times if (!busy);
    x <= a; y <= b; busy <= True;
  endmethod

  method ActionValue#(Bit#(32)) getResult \times if (busy && (x==0));
    busy <= False; return y;
endmodule
```

The Bluespec compiler rejects a program if there is any possibility of a double write in a rule or method.
Rule GCD

```
rule gcd;
   if (x >= y) begin x <= x - y; end //subtract
   else if (x != 0) begin x^{t+1} = y; y^{t+1} = x; end //swap
endrule
```

What is meaning of this? Swap!

Rule gcd executes repeatedly and perform subtract or swap but it has no effect if x == 0

---

Rules with guards

- Like a method, a rule can also have a guard

```
rule foo if (p);
   x1 <= e1; x2 <= e2;
endrule
```

Syntax: In rules, "if" is optional before the guard!

- A rule can execute only if it’s guard is true, i.e., if the guard is false the rule has no effect
- True guards can be omitted
- An alternative way to write the gcd rule is to split it into two rules:

```
rule gcdSubtract if (x >= y);
   x <= x - y;
endrule
```

These rules are mutually exclusive

```
rule gcdSwap if !(x >= y) && (x != 0);
   x <= y; y <= x;
endrule
```
Streaming a module

Rules are the glue that connects modules

```plaintext
rule invokeGCD ; if(inQ.first.rdy && inQ.deq.rdy) &
    let x = tpl_1(inQ.first); &
    let y = tpl_2(inQ.first);
    gcd.start(x,y);
    inQ.deq;
endrule

rule getResult ; if(gcd.getResult.rdy &
    let x <- gcd.getResult; &
    outQ.enq(x);
endrule
```

 Guards are implicit!

inQ is not empty
gcd is not busy

Expressing a loop using registers

- Such a loop cannot be implemented by unfolding because the number of iterations is input-data dependent
- A register is needed to hold s from one iteration to the next
- s has to be initialized when the computation starts, and updated every cycle until the computation terminates

```
s = s0;
while p(s):
    s = f(s)
return s;
```
Expressing a loop in BSV

- When a rule executes:
  - the register \( s \) is read at the beginning of a clock cycle
  - computations to evaluate the next value of the register and the \( s_{en} \) are performed
  - If \( s_{en} \) is True then \( s \) is updated at the end of the clock cycle

- A mux is needed to initialize the register

How should this circuit be packaged for proper use?

Packaging a computation as a Module

Interface with guards

```hs
interface F#(t);
  method Action start (t a);
  method ActionValue#(t) getResult;
endinterface
```

This interface is “latency-insensitive”; it does not depend upon how many cycles \( F \) takes to compute the result.