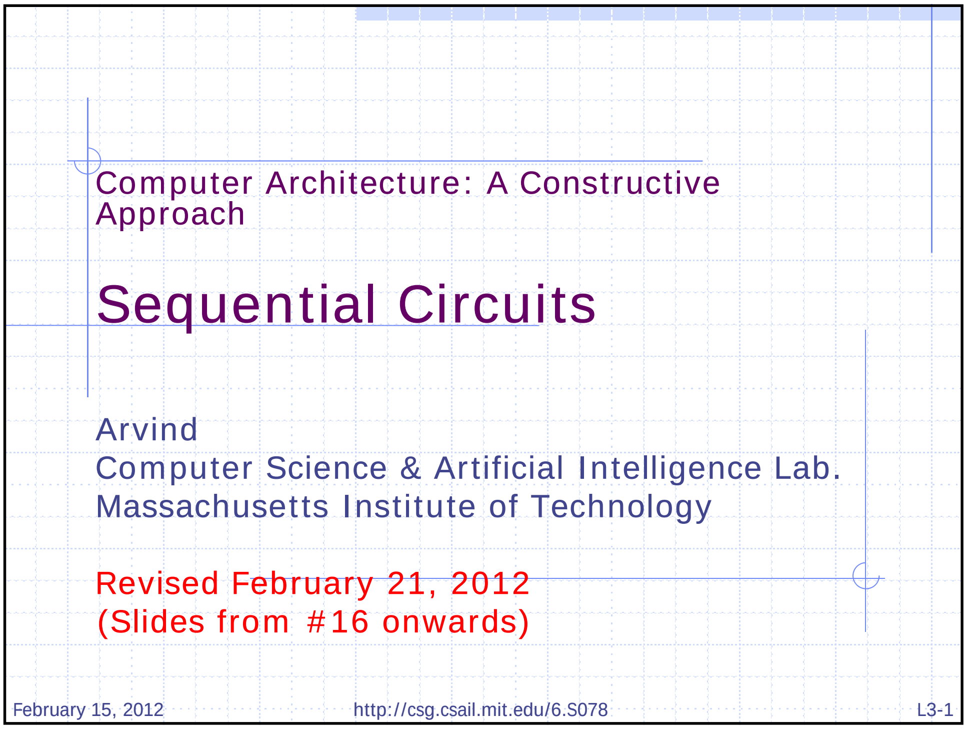


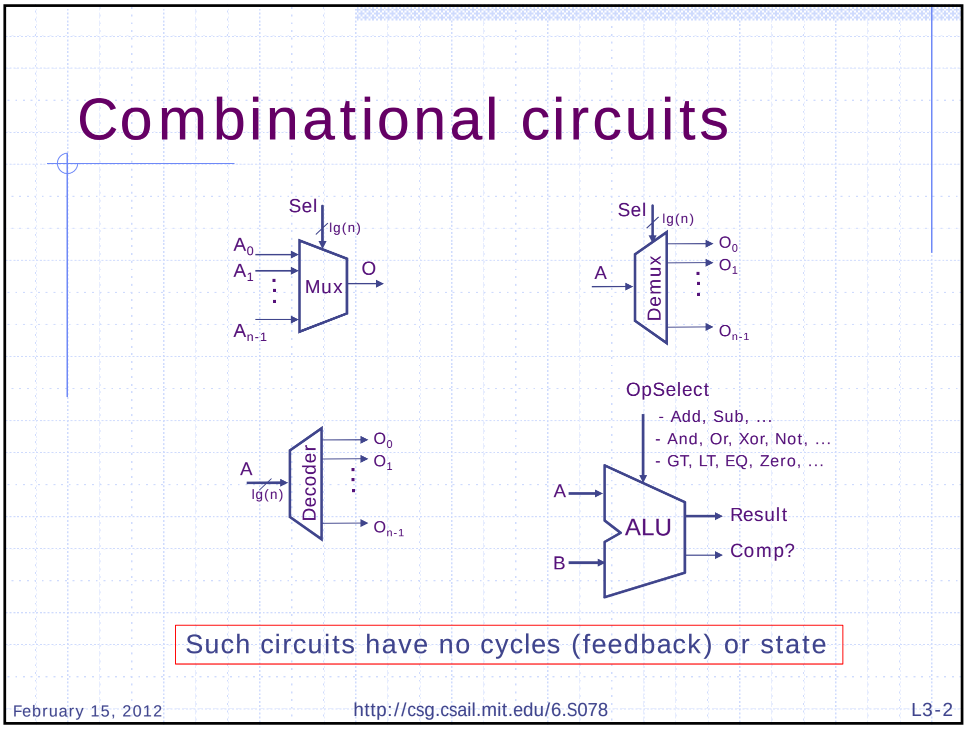
Computer Architecture: A Constructive Approach

Sequential Circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Revised February 21, 2012
(Slides from #16 onwards)

February 15, 2012 http://csg.csail.mit.edu/6.S078 L3-1



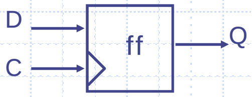
Combinational circuits

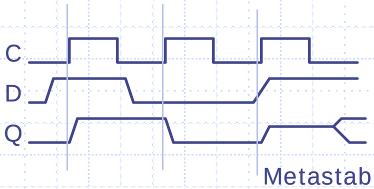
Such circuits have no cycles (feedback) or state

February 15, 2012 http://csg.csail.mit.edu/6.S078 L3-2

A simple synchronous state element

Edge-Triggered Flip-flop

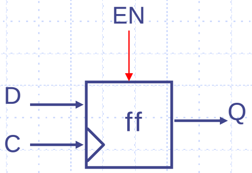


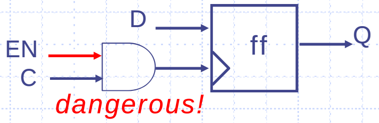


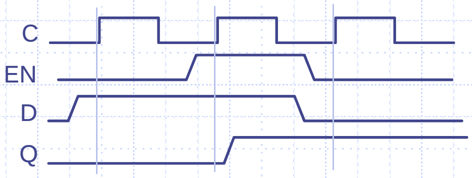
Metastability

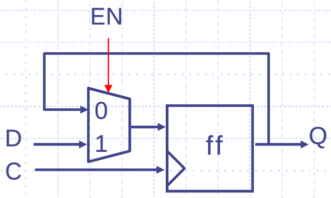
Data is sampled at the rising edge of the clock

Flip-flops with Write Enables



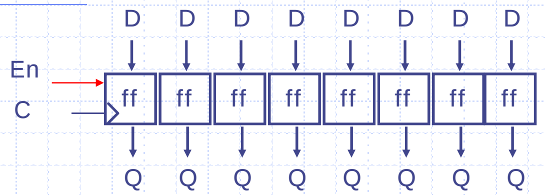






Data is captured only if EN is on

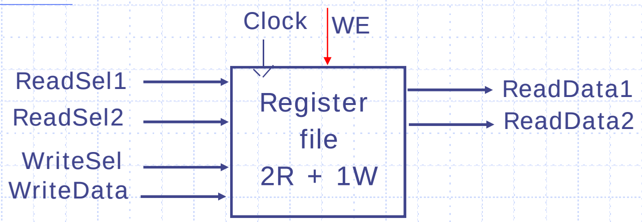
Registers

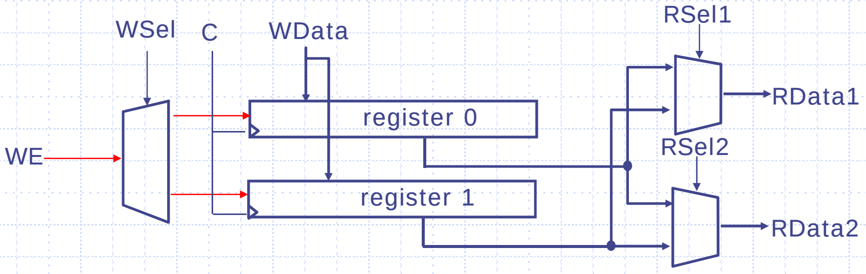


Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, input and output port(s)

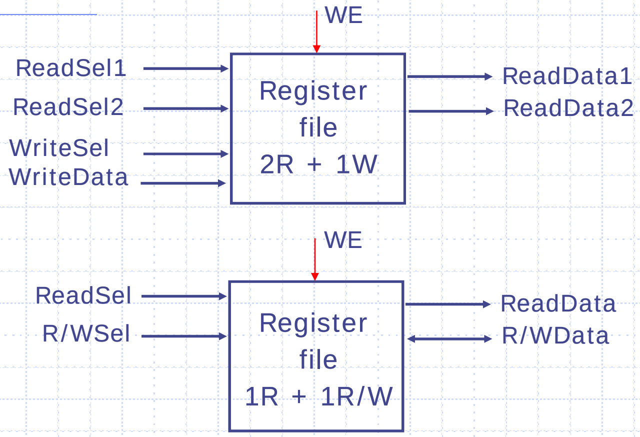
Register Files





No timing issues in reading a selected register

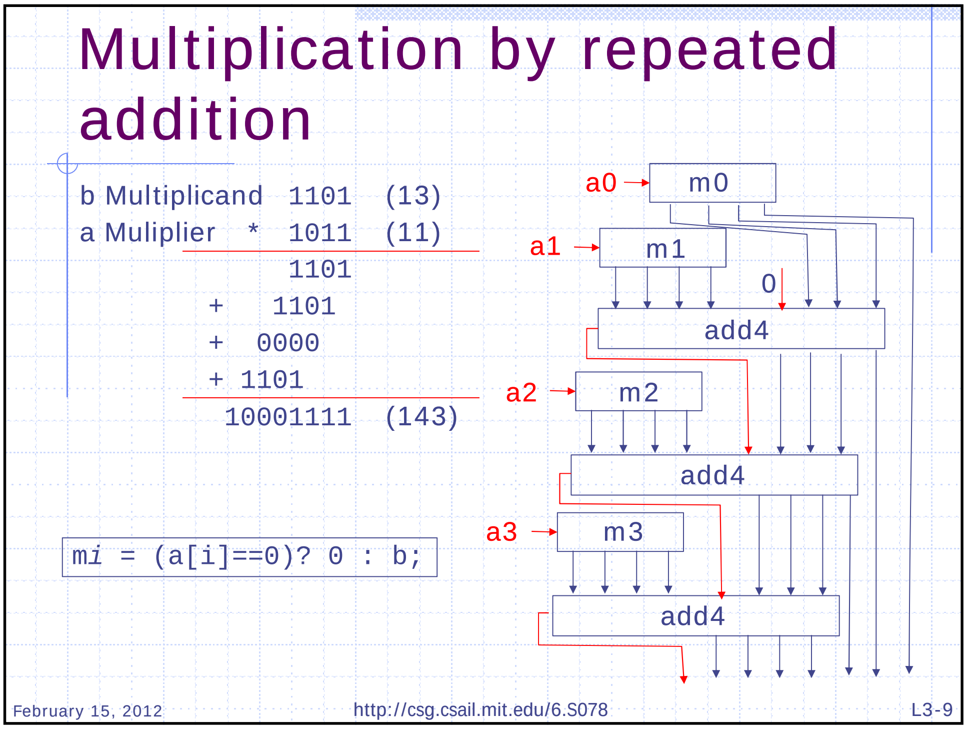
Register Files and Ports



Ports were expensive ⇒ multiplex a port for read & write

We can build useful and compact circuits using registers

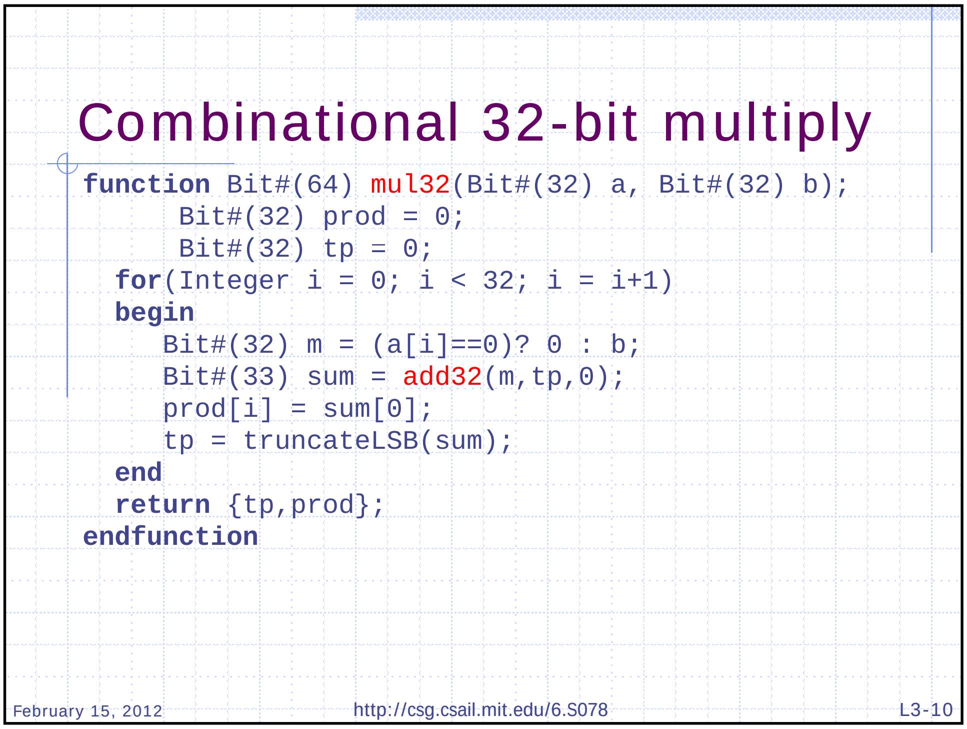
Example: Multiplication by repeated addition



Multiplication by repeated addition

```
b Multiplicand   1101   (13)
a Muliplier   *  1011   (11)
                 1101
          +     1101
          +    0000
          +   1101
             10001111   (143)
```

`mi = (a[i]==0)? 0 : b;`



Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

Design issues with combinational multiply

- Lot of hardware
  - 32-bit multiply uses 31 addN circuits
- Long chains of gates
  - 32-bit ripple carry adder has a 31 long chain of gates
  - 32-bit multiply has 31 ripple carry adders in sequence!

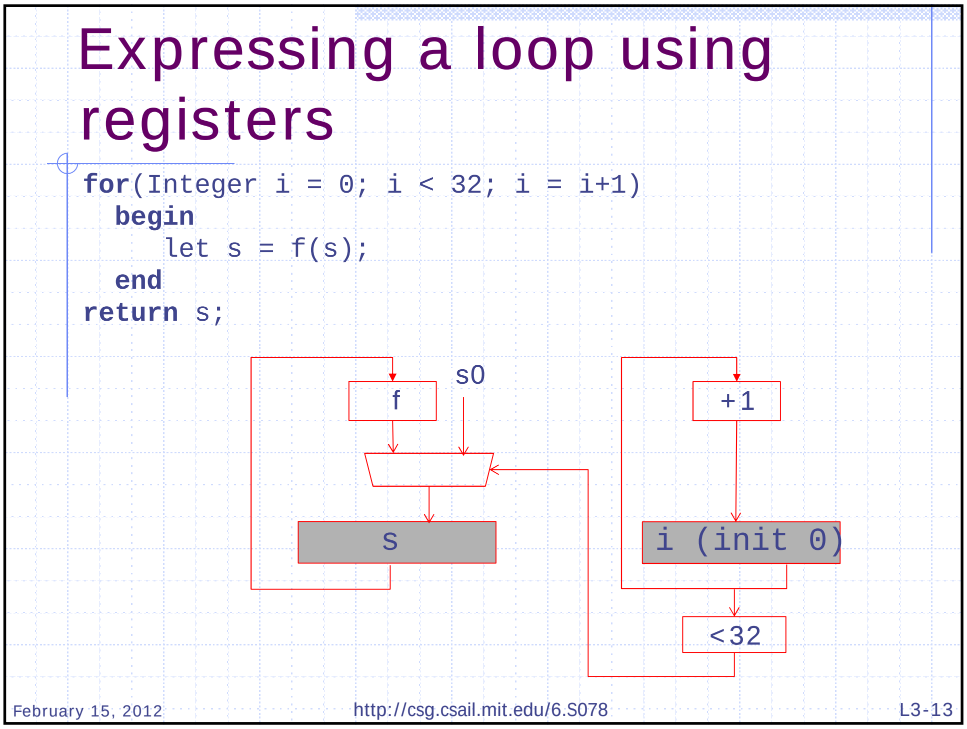
The speed of a combinational circuit is determined by its longest input-to-output path

Expressing a loop using registers

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
      Bit#(32) prod = 0;
      Bit#(32) tp = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] = sum[0];
     tp = truncateLSB(sum);
  end
  return {tp,prod};
endfunction
```

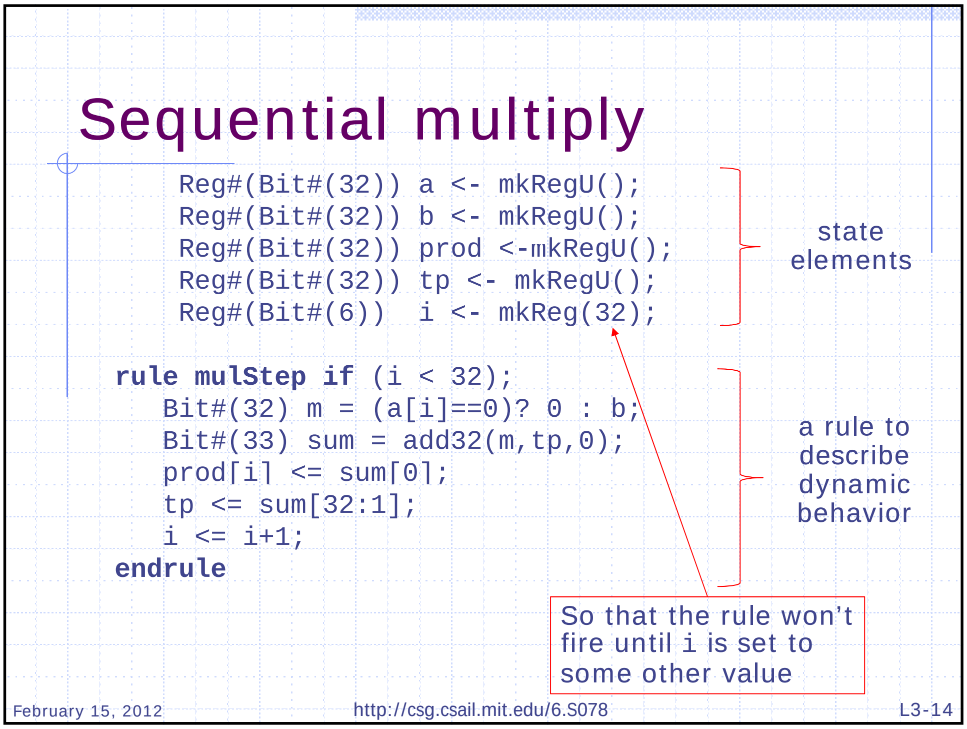
Need registers to hold a, b, tp prod and i

Update the registers every cycle until we are done



Expressing a loop using registers

```
for(Integer i = 0; i < 32; i = i+1)
  begin
     let s = f(s);
  end
return s;
```



Sequential multiply

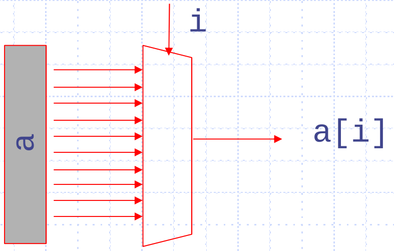
```
      Reg#(Bit#(32)) a <- mkRegU();
      Reg#(Bit#(32)) b <- mkRegU();
      Reg#(Bit#(32)) prod <-mkRegU();
      Reg#(Bit#(32)) tp <- mkRegU();
      Reg#(Bit#(6))  i <- mkReg(32);

  rule mulStep if (i < 32);
     Bit#(32) m = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i] <= sum[0];
     tp <= sum[32:1];
     i <= i+1;
  endrule
```

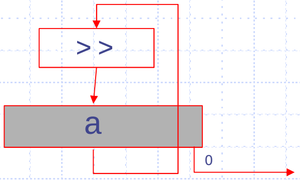
state elements

a rule to describe dynamic behavior

So that the rule won't fire until i is set to some other value

Dynamic selection requires a mux



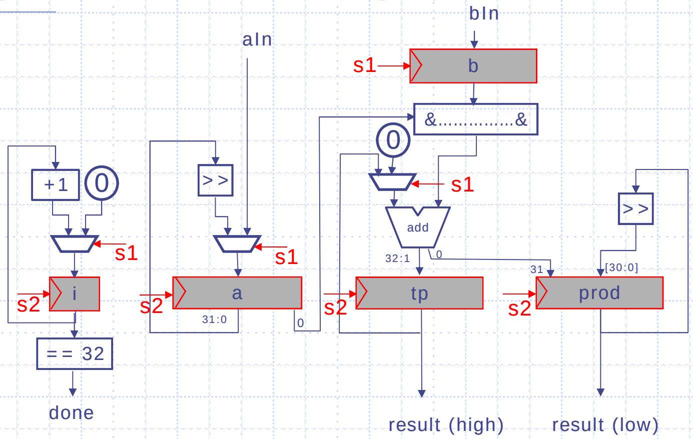


a[0],a[1],a[2],…

Replacing repeated selections by shifts

```
    Reg#(Bit#(32)) a <- mkRegU();
    Reg#(Bit#(32)) b <- mkRegU();
    Reg#(Bit#(32)) prod <-mkRegU();
    Reg#(Bit#(32)) tp <- mkRegU();
    Reg#(Bit#(6))  i <- mkReg(32);

rule mulStep if (i < 32);
   Bit#(32) m = (a[0]==0)? 0 : b;
   a <= a >> 1;
   Bit#(33) sum = add32(m,tp,0);
   prod <= {sum[0], (prod >> 1)[30:0]};
   tp <= sum[32:1];
   i <= i+1;
endrule
```

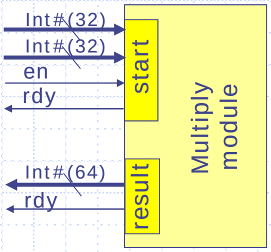
Sequential Multiply



s1 = start_en

s2 = start_en | !done

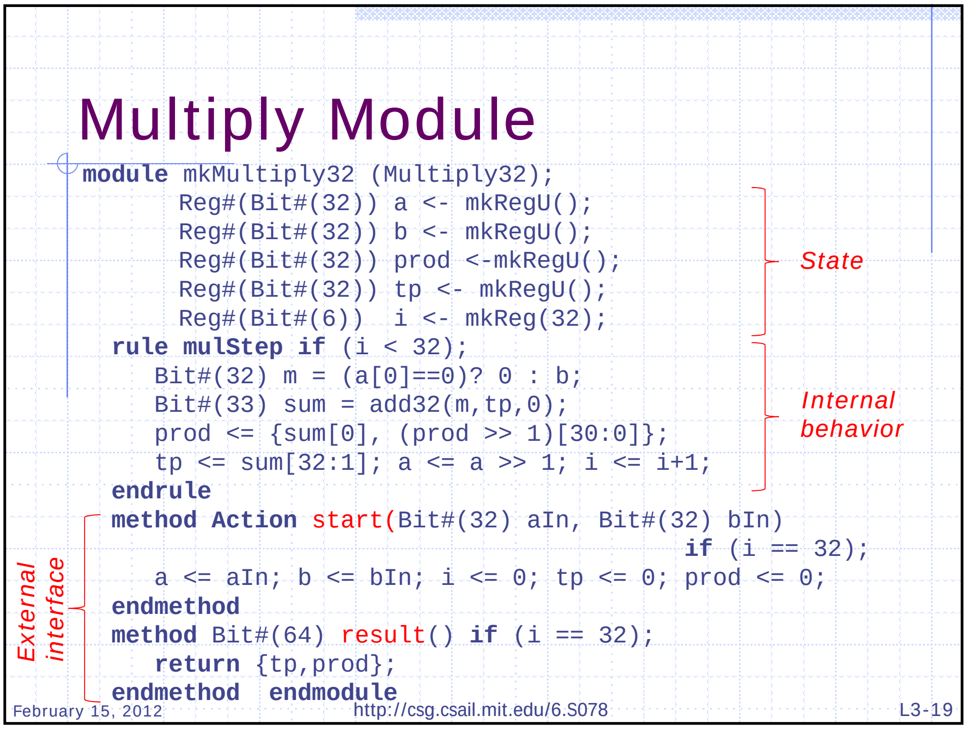
Multiply Module



implicit conditions

```
interface Multiply;
    method Action start (Int#(32) a, Int#(32) b);
    method Int#(64) result();
endinterface
```

- Many different implementations can provide the same interface: `module mkMultiply (Multiply)`



Multiply Module

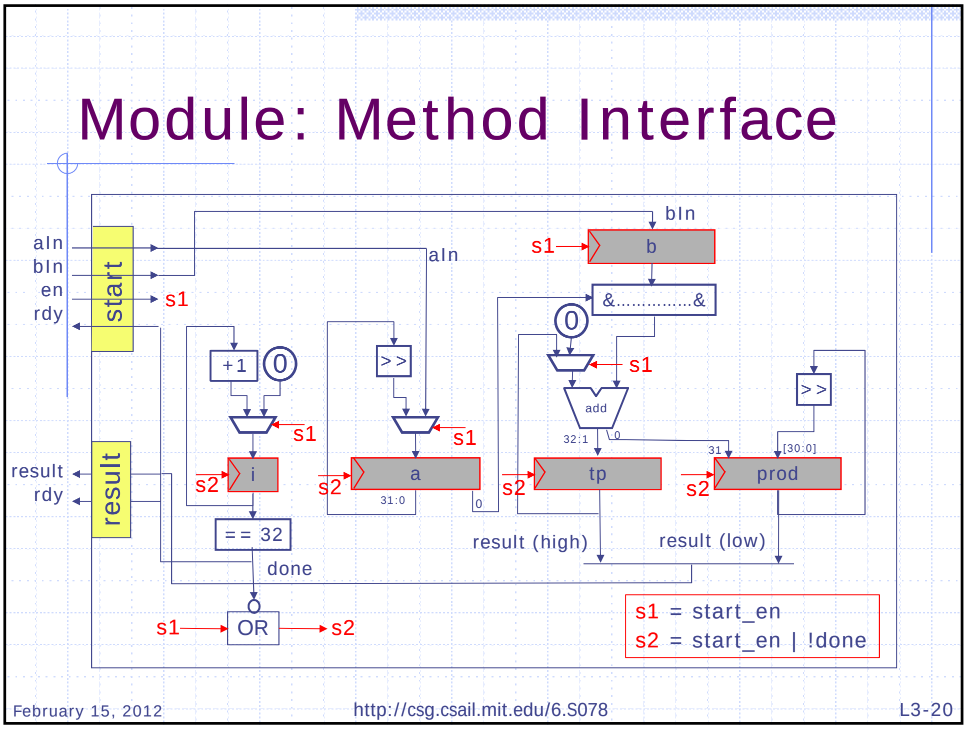
```
module mkMultiply32 (Multiply32);
       Reg#(Bit#(32)) a <- mkRegU();
       Reg#(Bit#(32)) b <- mkRegU();
       Reg#(Bit#(32)) prod <-mkRegU();
       Reg#(Bit#(32)) tp <- mkRegU();
       Reg#(Bit#(6))  i <- mkReg(32);
  rule mulStep if (i < 32);
     Bit#(32) m = (a[0]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod <= {sum[0], (prod >> 1)[30:0]};
     tp <= sum[32:1]; a <= a >> 1; i <= i+1;
  endrule
  method Action start(Bit#(32) aIn, Bit#(32) bIn)
                                        if (i == 32);
     a <= aIn; b <= bIn; i <= 0; tp <= 0; prod <= 0;
  endmethod
  method Bit#(64) result() if (i == 32);
     return {tp,prod};
  endmethod  endmodule
```

State

Internal behavior

External interface

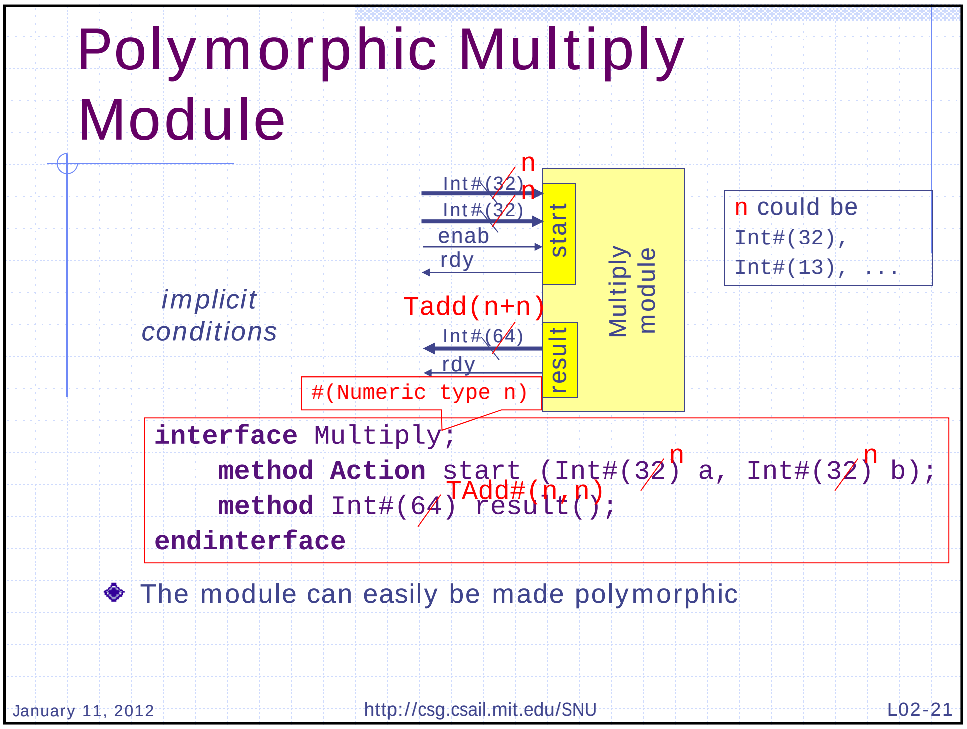
February 15, 2012 http://csg.csail.mit.edu/6.S078 L3-19



Module: Method Interface

s1 = start_en
s2 = start_en | !done

February 15, 2012 http://csg.csail.mit.edu/6.S078 L3-20



Polymorphic Multiply Module

Int#(32)
n
Int#(32)
n
enab
rdy
start
Multiply module
Tadd(n+n)
Int#(64)
rdy
result
implicit conditions

n could be
Int#(32),
Int#(13), ...

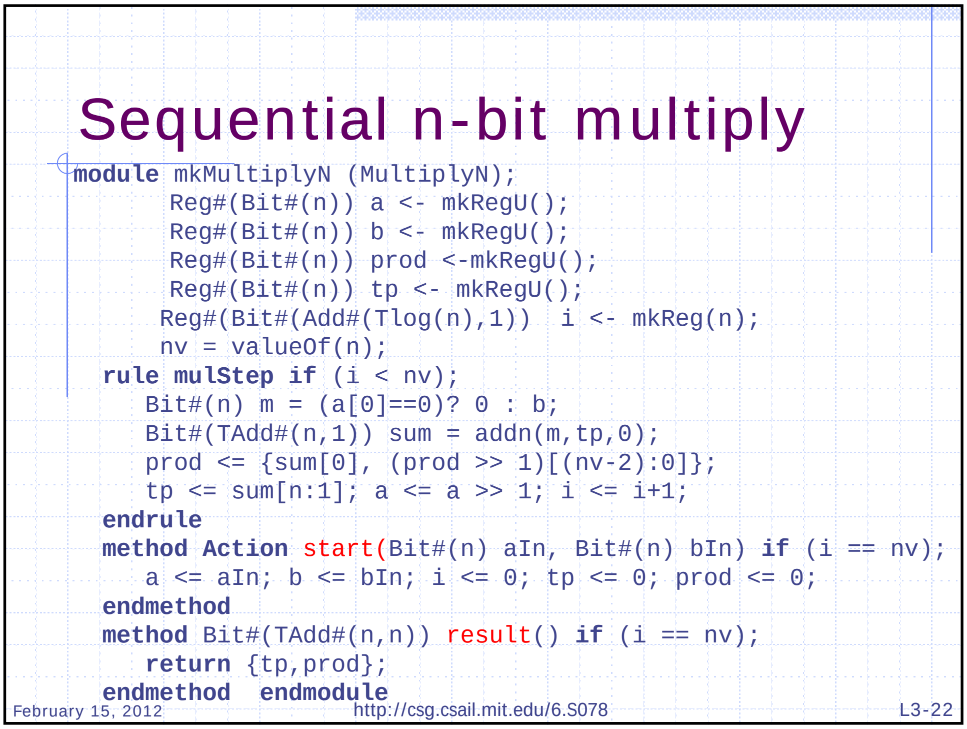
#(Numeric type n)

```
interface Multiply;
    method Action start (Int#(32) a, Int#(32) b);
    method Int#(64) result();
endinterface
```

n n TAdd#(n,n)

- The module can easily be made polymorphic

January 11, 2012 http://csg.csail.mit.edu/SNU L02-21



Sequential n-bit multiply

```
module mkMultiplyN (MultiplyN);
      Reg#(Bit#(n)) a <- mkRegU();
      Reg#(Bit#(n)) b <- mkRegU();
      Reg#(Bit#(n)) prod <-mkRegU();
      Reg#(Bit#(n)) tp <- mkRegU();
     Reg#(Bit#(Add#(Tlog(n),1))  i <- mkReg(n);
     nv = valueOf(n);
  rule mulStep if (i < nv);
     Bit#(n) m = (a[0]==0)? 0 : b;
     Bit#(TAdd#(n,1)) sum = addn(m,tp,0);
     prod <= {sum[0], (prod >> 1)[(nv-2):0]};
     tp <= sum[n:1]; a <= a >> 1; i <= i+1;
  endrule
  method Action start(Bit#(n) aIn, Bit#(n) bIn) if (i == nv);
     a <= aIn; b <= bIn; i <= 0; tp <= 0; prod <= 0;
  endmethod
  method Bit#(TAdd#(n,n)) result() if (i == nv);
     return {tp,prod};
  endmethod  endmodule
```

February 15, 2012 http://csg.csail.mit.edu/6.S078 L3-22