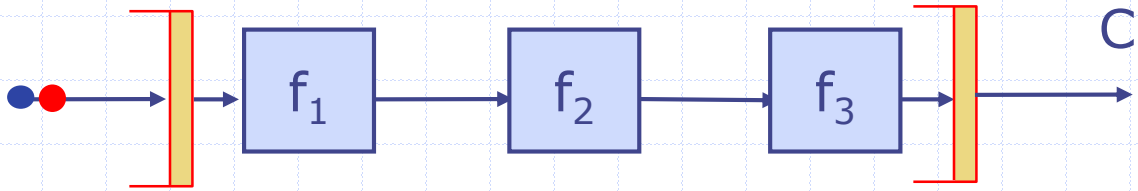
Constructive Computer Architecture

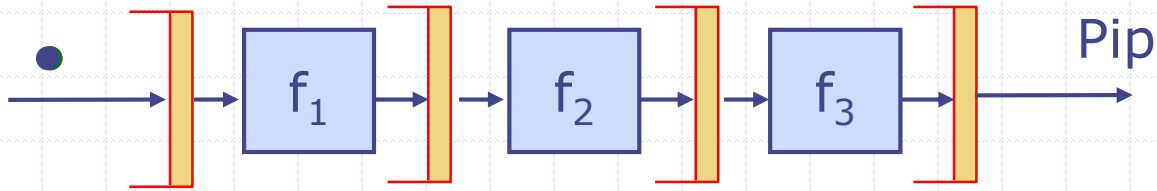
Folded "Combinational" circuits

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

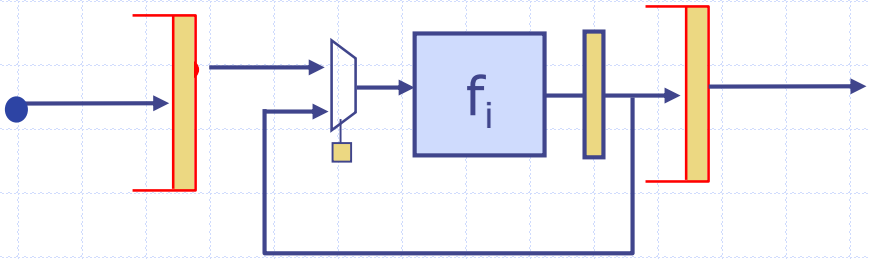
Design Alternatives



Combinational (C)



Pipeline (P)



Folded (F)
Reuse a block, multicycle

Clock: C < P ≈ F

Area: F < C < P

Throughput: F < C < P

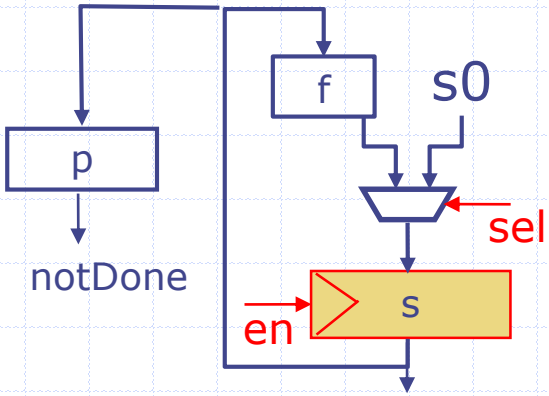
Content

- How to implement loop computations?
  - Need registers to hold the state from one iteration to the next
- Request-Response Latency-Insensitive modules
- A common way to implement large combinational circuits is by *folding or* as loops
  - Multiplication
- Polymorphic Multiply

Expressing a loop using registers

```
int s = s0;
while (p(s)) {
    s = f(s);
  }
return s;
```

C-code



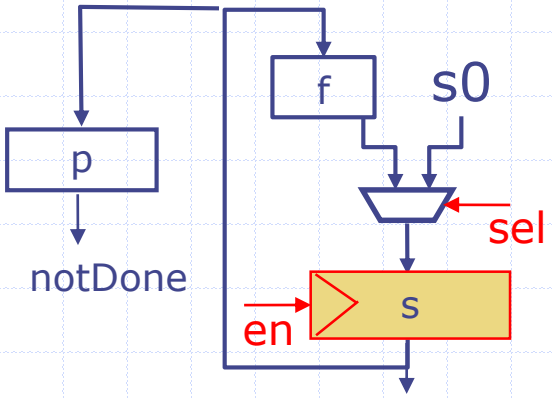
- Such a loop cannot be implemented by unfolding because the number of iterations is input-data dependent
- A register is needed to hold s from one iteration to the next
- s has to be initialized when the computation starts, and updated every cycle until the computation terminates

sel = start
en = start | notDone

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

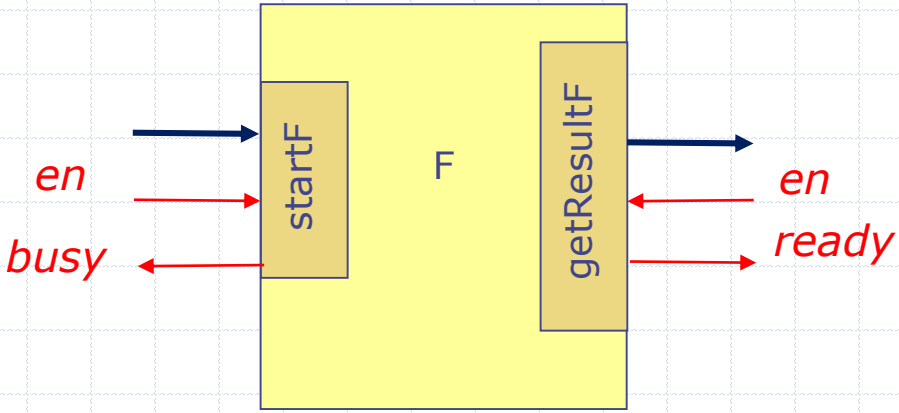
```
Reg#(t) s <- mkRegU();
rule step;
  if (p(s)) begin
      s <= f(s);
  end
endrule
```



How should this circuit be packaged for proper use?

sel = start
en = start | notDone

Packaging a computation as a Latency-Insensitive Module



Interface with guards

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

Request-Response Module

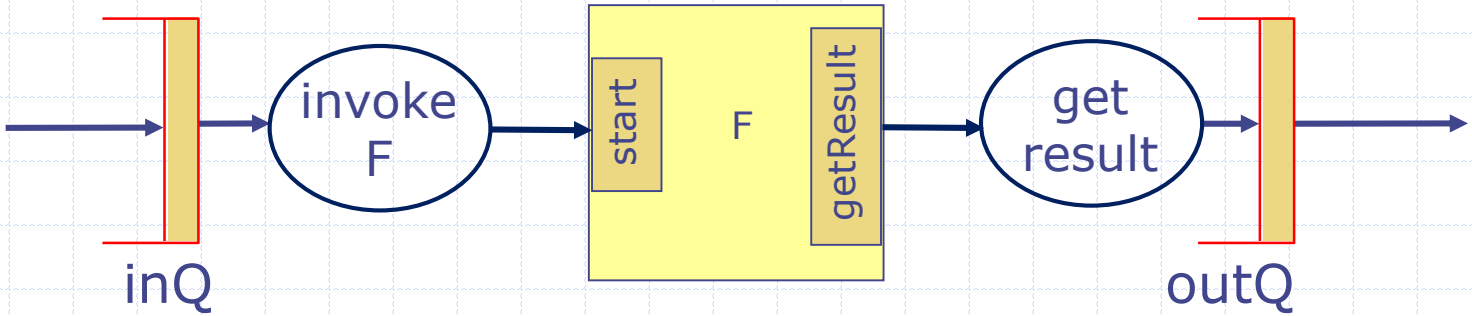
```
module mkF (F#(t));
    Reg#(t) s <-mkRegU();
    Reg#(Bool) busy <- mkReg(False);

   rule step;
     if (p(s)) begin
     s <= f(s);
     end
   endrule

method Action start(t a) if (!busy);
    s <= a; busy <= True;
endmethod

method ActionValue t getResult if (!p(s)&& busy);
  busy <= False; return s;
endmethod
endmodule
```

Using F




```
F#(t) f <- mkF(…)
```

```
rule invokeF;
  f.start(inQ.first); inQ.deq;
endrule
```

```
rule getResult;
   let x <- f.getResult; outQ.enq(x);
endrule
```

This system is *insensitive to the latency of F*

A rule can be executed only if guards of all of its actions are true

Combinational 32-bit multiply

```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
  Bit#(32) tp = 0;
  Bit#(32) prod = 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           = sum[32:1];
  end
  return {tp,prod};
endfunction
```

Combinational circuit uses 31 add32 circuits

We can reuse the same add32 circuit if we store the partial results in a *register*

Multiply 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:i] = sum[0];
     tp = sum[32:1];
  end
  return {tp,prod};
endfunction
```

Combinational version

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

Update the registers every cycle until we are done

Sequential Circuit for Multiply

```
Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <-mkRegU();
Reg#(Bit#(32)) tp <- mkReg(0);
Reg#(Bit#(6))  i <- mkReg(32);
```

state elements

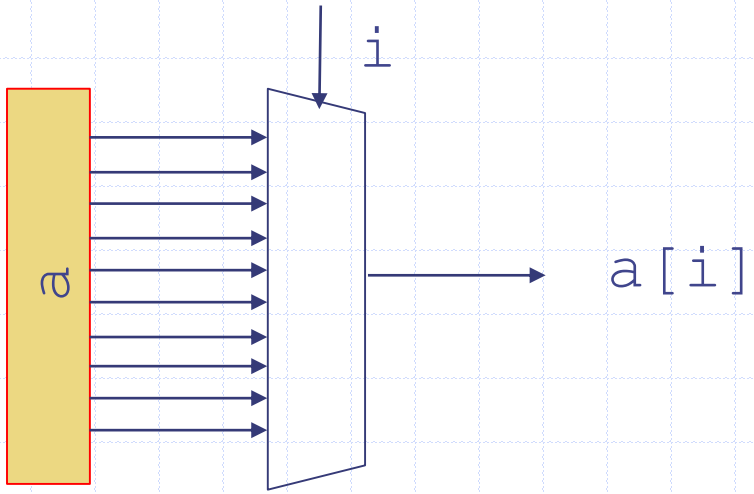
```
rule mulStep;
  if (i < 32) begin
    Bit#(32) m = (a[i]==0)? 0 : b;
    Bit#(33) sum = add32(m,tp,0);
    prod[i] <= sum[0];
    tp <= sum[32:1];
    i <= i+1;
  end
endrule
```

a rule to describe the dynamic behavior

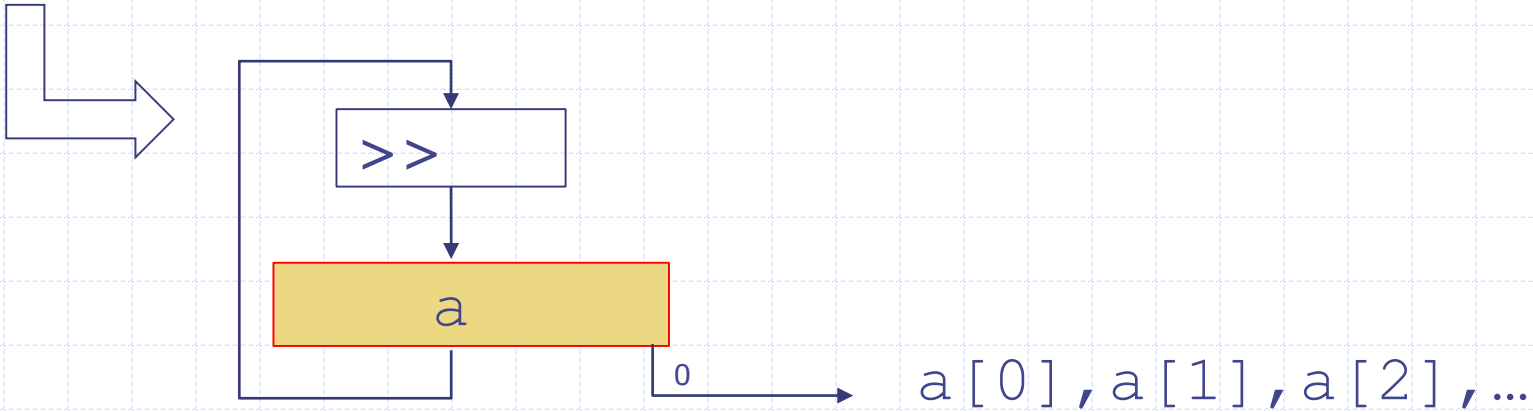
similar to the loop body in the combinational version

So that the rule has no effect until i is set to some other value

Dynamic selection requires a mux



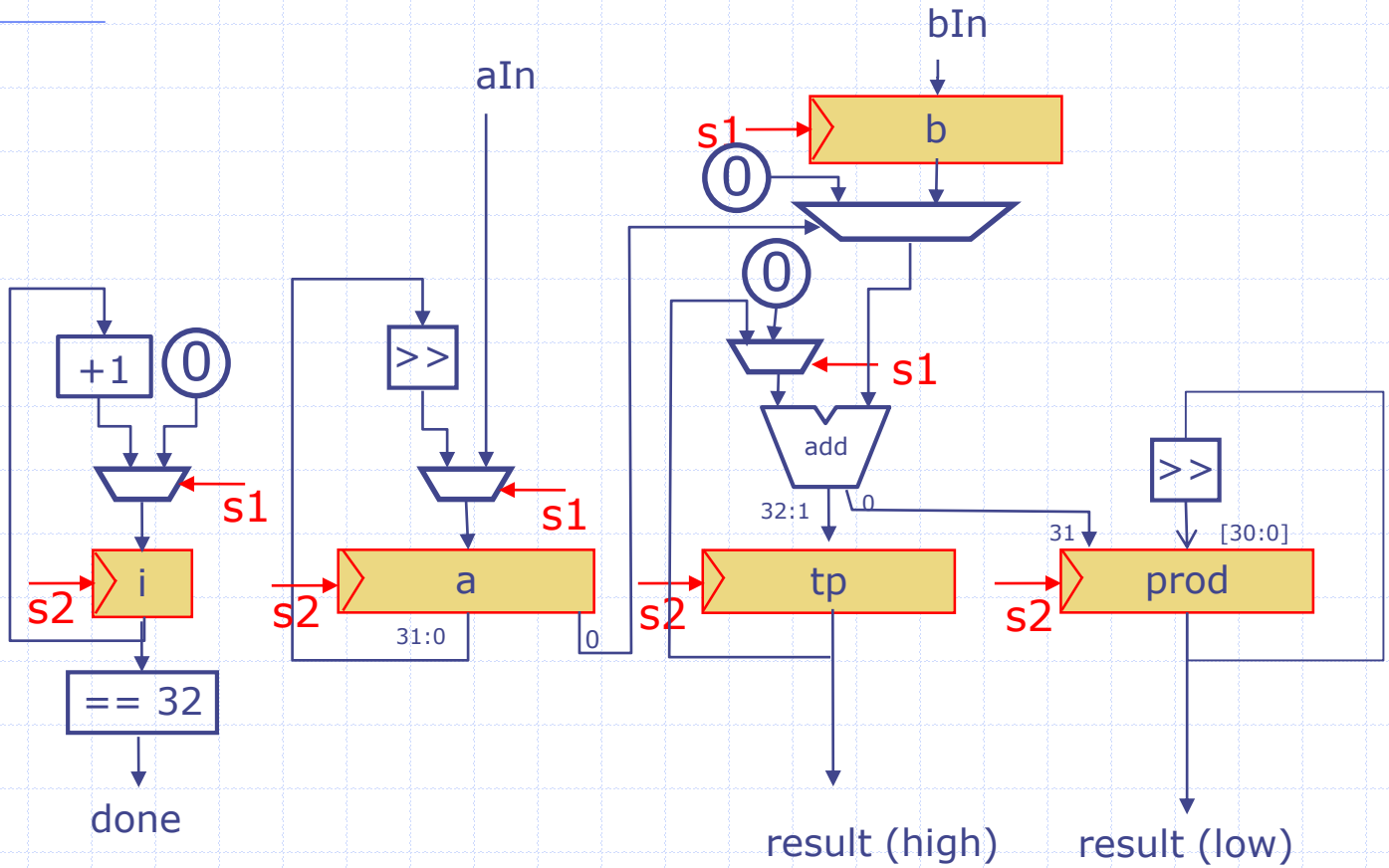
when the selection indices are regular then it is better to use a shift operator (no gates!)



Replacing repeated selections by shifts

```
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[31:1]};
    tp <= sum[32:1];
    i <= i+1;
endrule
```

Circuit for Sequential Multiply

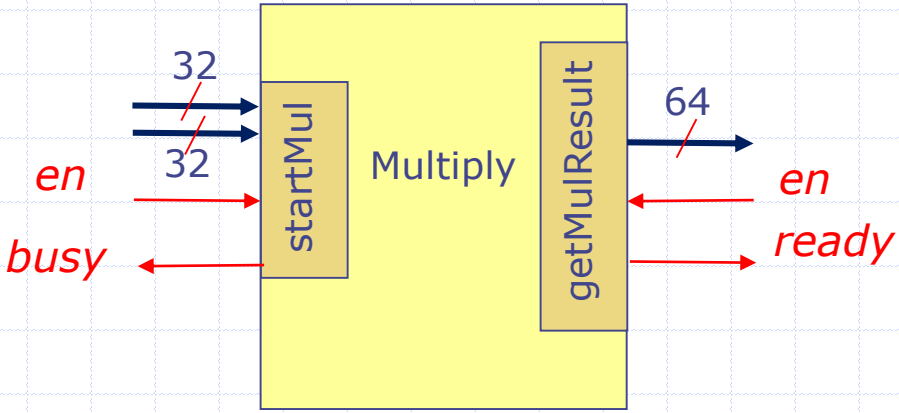


s1 = start_en
s2 = start_en | !done

Circuit analysis

- Number of add32 circuits has been reduced from 31 to one, though some registers and muxes have been added
- The longest combinational path has been reduced from 62 FAs to one add32 plus a few muxes
- The sequential circuit will take 31 clock cycles to compute an answer

Packaging Multiply as a Latency-Insensitive Module



Interface with guards

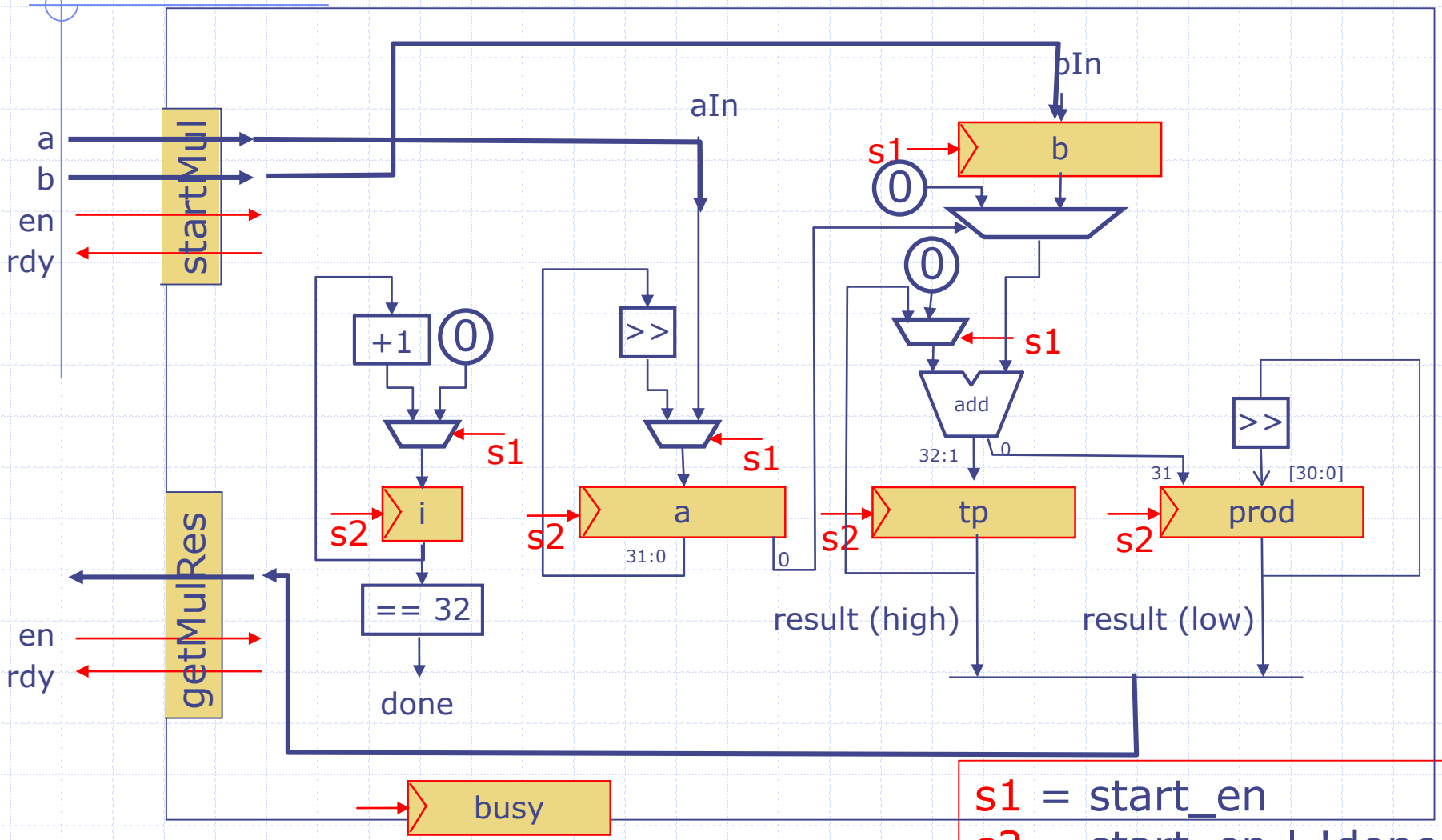
```
interface Multiply;
  method Action startMul (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(64)) getResultMul;
endinterface
```

Multiply Module

```
Module mkMultiply (Multiply);
    Reg#(Bit#(32)) a<-mkRegU(); Reg#(Bit#(32)) b<-mkRegU();
    Reg#(Bit#(32)) prod <-mkRegU();
    Reg#(Bit#(32)) tp <- mkReg(0);
    Reg#(Bit#(6))  i <- mkReg(32);
    Reg#(Bool) busy <- mkReg(False);

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[31:1]};
     tp <= sum[32:1]; i <= i+1;
endrule
method Action startMul(Bit#(32) x, Bit#(32) y) if (!busy);
  a <= x; b <= y; busy <= True; i <= 0;  endmethod
method ActionValue Bit#(64) getMulRes if ((i==32) && busy);
  busy <= False; return {tp,prod}; endmethod
```

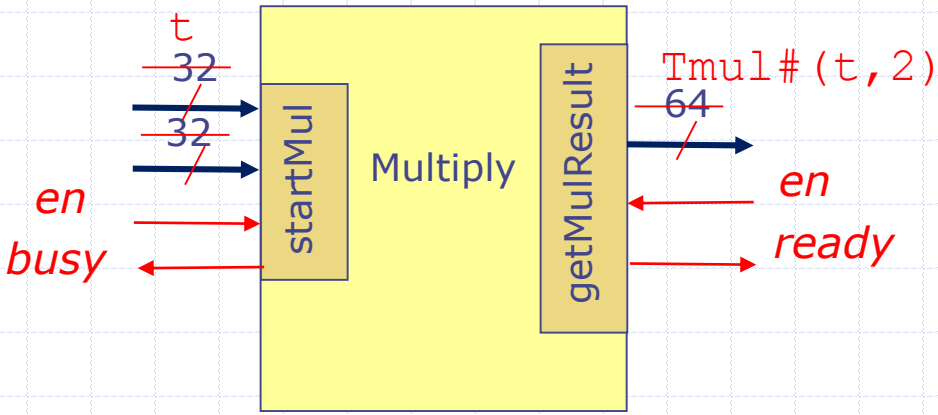
Circuit for Sequential Multiply



s1 = start_en

s2 = start_en | !done

Polymorphic Multiply Module



Polymorphic Interface

```
interface Multiply#(t ~~32~~);
  method Action startMul (Bit#(t ~~32~~) a, Bit#(t ~~32~~) b);
  method ActionValue#(Bit#(Tmul#(t,2) ~~64~~)) getResultMul;
endinterface
```

Polymorphic Multiply

```
Module mkMultiply (Multiply#(t));
    Reg#(Bit#(t)) a<-mkRegU(); Reg#(Bit#(t)) b<-mkRegU();
    Reg#(Bit#(t)) prod <-mkRegU();
    Reg#(Bit#(t)) tp <- mkReg(0); vt = valueOf(t);
    Reg#(Bit#(TAdd#(1, TLog#(t))))  i <- mkReg(vt);
    Reg#(Bool) busy <- mkReg(False);
rule mulStep if (i < vt);
     Bit#(t) m = (a[0]==0)? 0 : b;
     a <= a >> 1;
     Bit#(Tadd#(t)) sum = addN(m,tp,0);
     prod <= {sum[0], prod[(vt-1):1]};
     tp <= sum[vt:1]; i <= i+1;
endrule
method Action startMul(Bit#(t) x, Bit#(t) y) if (!busy);
  a <= x; b <= y; busy <= True; i<=0; endmethod
method ActionValue#(Bit#(TMul#(t,2)) getMulRes if
                                        ((i==vt)&& busy);
  busy <= False; return {tp,prod}; endmethod
```