

Constructive Computer Architecture

Sequential Circuits - 2

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

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-1

Content

- ◆ So far we have seen modules with methods which are called by rules outside the body
- ◆ Now we will see examples where a module may also contain rules
 - gcd
- ◆ A common way to implement large combinational circuits is by *folding* where registers hold the state from one iteration to the next
 - Implementing imperative loops
 - Multiplication

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-2

Programming with rules: A simple example

Euclid's algorithm for computing the Greatest Common Divisor (GCD):

15	6	
9	6	<i>subtract</i>
3	6	<i>subtract</i>
6	3	<i>swap</i>
3	3	<i>subtract</i>
0	<i>answer:</i> 3	<i>subtract</i>

GCD module

Euclidean Algorithm

```

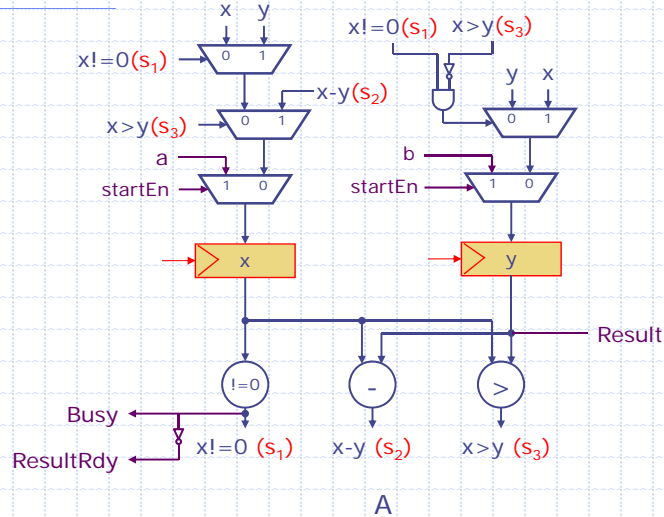
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
rule gcd;
  if (x >= y) begin
    x <= x - y;
  end else if (x != 0) begin
    x <= y; y <= x;
  end
endrule
method Action start(Bit#(32) a, Bit#(32) b);
  x <= a; y <= b; endmethod
method Bit#(32) result; return y; endmethod
method Bool resultRdy; return x == 0; endmethod
method Bool busy; return x != 0; endmethod
    
```

A rule inside a module may execute anytime

If x is 0 then the rule has no effect

- ◆ Start method should be called only if busy is False.
- ◆ The result is available only when resultRdy is True.

Circuits for GCD



September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-5

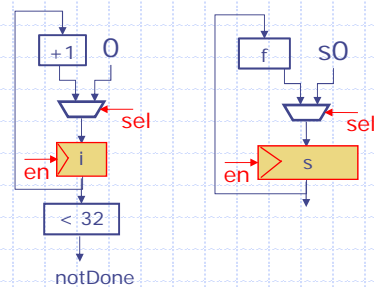
Expressing a loop using registers

```
int s = s0;
for (int i = 0; i < 32; i = i+1) {
    s = f(s);
}
return s;
```

C-code

We need two registers to hold s and i values from one iteration to the next.

These registers are initialized when the computation starts and updated every cycle until the computation terminates



sel = start
en = start | notDone

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-6

Expressing a loop in BSV

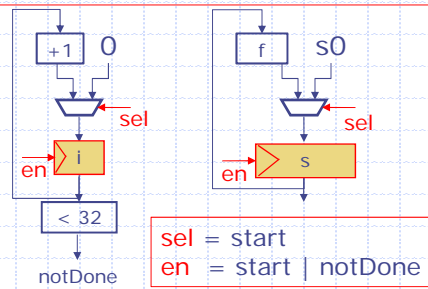
◆ When a rule executes:

- all the registers are read at the beginning of a clock cycle
- computations to evaluate the next value of the registers are performed
- Registers that need to be updated are updated at the end of the clock cycle

◆ Muxes are needed to initialize the registers

```

Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6)) i <- mkReg(32);
rule step;
  if (i < 32) begin
    s <= f(s); i <= i+1;
  end
endrule
    
```



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: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

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-9

Sequential Circuit for Multiply

```
Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <-mkReg(0);
Reg#(Bit#(32)) tp <- mkReg(0);
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
the
dynamic
behavior

similar to the
loop body in the
combinational
version

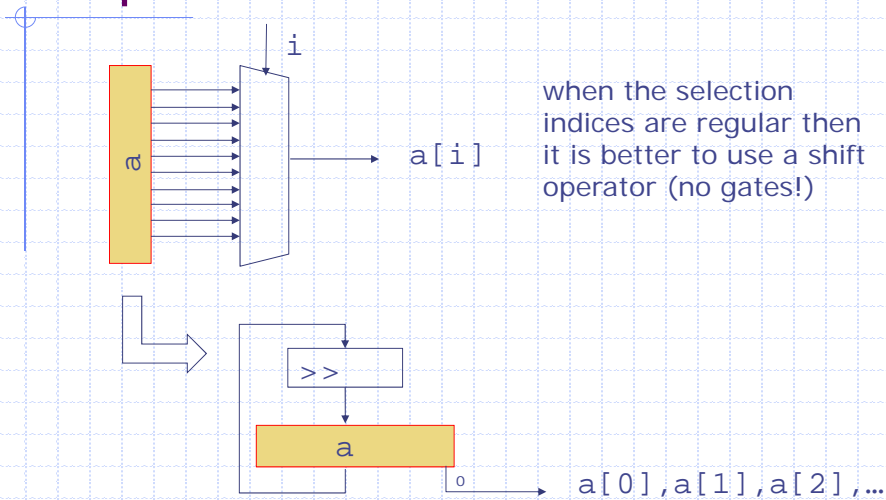
So that the rule has
no effect until i is set
to some other value

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-10

Dynamic selection requires a mux



September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-11

Replacing repeated selections by shifts

```
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);

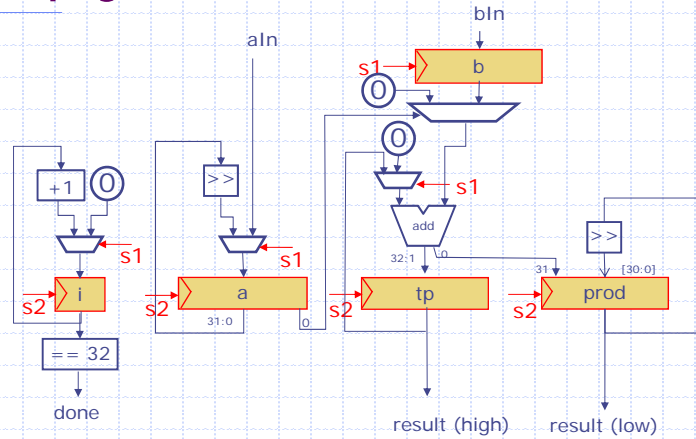
rule mulStep if (i < 32);
  Bit#(32) m = (a[0]==0)? 0 : b;
  a <= a >> 1;
  Bit#(33) sum = add32(m,tp,0);
  prod <= {sum[0], prod[31:1]};
  tp <= sum[32:1];
  i <= i+1;
endrule
```

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-12

Circuit for Sequential Multiply



s1 = start_en
s2 = start_en | !done

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-13

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

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-14

Observations

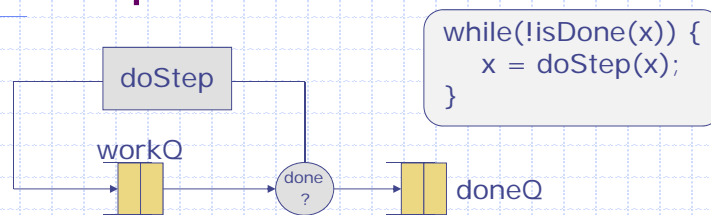
- ◆ These programs are not very complex and yet it would have been tedious to express these programs in a state table or as a circuit directly
- ◆ BSV method calls are not available in Verilog/VHDL, and thus such programs sometimes require tedious programming
- ◆ Even the meaning of double-write errors is not standardized across tool implementations in Verilog

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-15

A subtle problem



```
let x = workQ.first;
workQ.deq;
if (isDone(x)) begin
  doneQ.enq(x);
end else begin
  workQ.enq(doStep(x));
end
```

Double write problem
for 1-element Fifo

Later we will design FIFOs
to permit simultaneous enq
and deq

September 18, 2015

<http://csg.csail.mit.edu/6.175>

L05-16

Illegal Actions – Double Write

- ◆ `x <= e1; x <= e2;`
- ◆ `x <= e1; if(p) x <= e2;`

- ◆ `if(p) x <= e1; else x <= e2;` Not an error

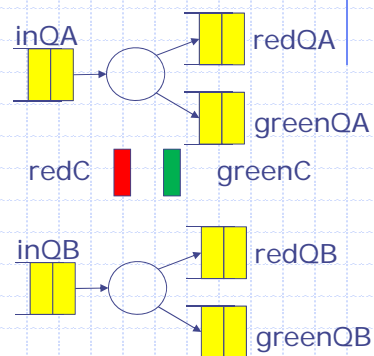
Parallel composition of two actions is illegal if it creates the possibility of a *double-write error*, that is, if its constituent (sub)actions invoke the same action method

Shared counters

```

if (inQA.first.color == Red) begin
    redQA.enq(inQA.first.value);
    inQA.deq; redC <= redC+1;
end else begin
    greenQA.enq(inQA.first.value);
    inQA.deq; greenC <= greenC+1;
end;
if (inQB.first.color == Red) begin
    redQB.enq(inQB.first.value);
    inQB.deq; redC <= redC+1;
end else begin
    greenQB.enq(inQB.first.value);
    inQB.deq; greenC <= greenC+1;
end
    
```

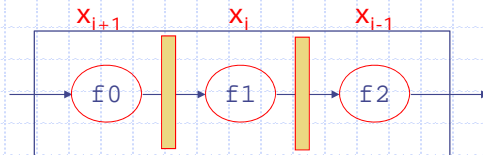
Ignoring full/empty conditions



What is wrong with this code?

Double write error

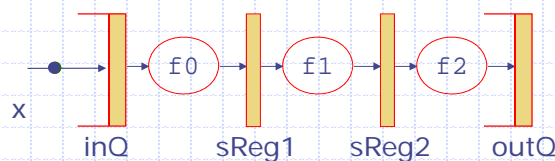
Pipelining Combinational Functions



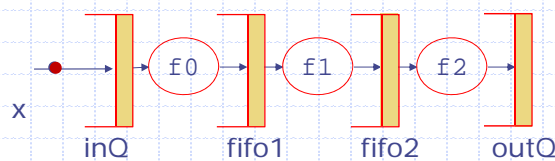
3 different datasets in the pipeline

- ◆ Lot of area and long combinational delay
- ◆ Folded or multi-cycle version can save area and reduce the combinational delay but throughput per clock cycle gets worse
- ◆ Pipelining: a method to increase the circuit throughput by evaluating multiple inputs

Inelastic vs Elastic pipeline



Inelastic: all pipeline stages move synchronously



Elastic: A pipeline stage can process data if its input FIFO is not empty and output FIFO is not Full

Most complex processor pipelines are a combination of the two styles