

Constructive Computer Architecture

Sequential Circuits

Arvind

Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-1

Contributors to the course material

- ◆ Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- ◆ Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012, 2013), 6.S078 (Spring 2012)
 - Andy Wright, Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- ◆ External
 - Prof Amey Karkare & students at IIT Kanpur
 - Prof Jihong Kim & students at Seoul Nation University
 - Prof Derek Chiou, University of Texas at Austin
 - Prof Yoav Etsion & students at Technion

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-2

Content

- ◆ Introduce sequential circuits as a way of saving area
 - Edge triggered Flipflop
 - Register
- ◆ New BSV concepts
 - state elements
 - rules and actions for describing dynamic behavior
 - modules and methods

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

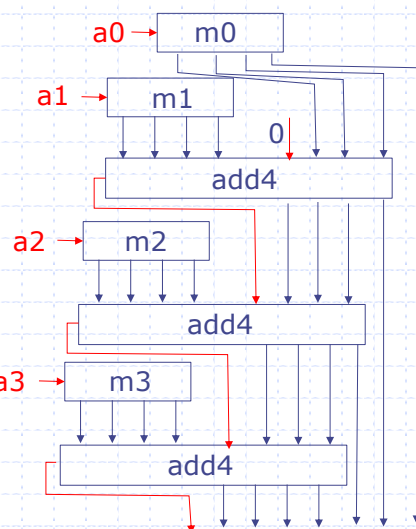
L02-3

Multiplication by repeated addition

```

b Multiplicand 1101 (13)
a Multiplier   * 1011 (11)
tp             0000
m0  + 1101
tp  01101
m1  + 1101
tp  100111
m2  + 0000
tp  0100111
m3  + 1101
tp  10001111 (143)
    
```

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



December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-4

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 32 add32 circuits

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-5

Design issues with combinational multiply

- ◆ Lot of hardware
 - 32-bit multiply uses 32 add32 circuits
- ◆ Long chains of gates
 - 32-bit ripple carry adder has a 31-long chain of gates
 - 32-bit multiply has 32 ripple carry adders in sequence!

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

Can we do better?

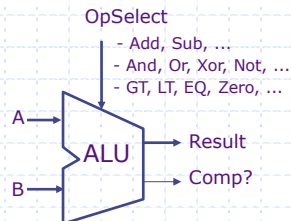
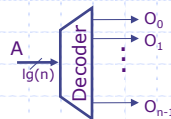
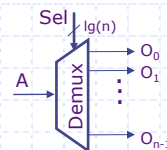
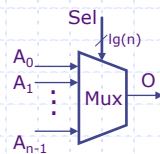
December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-6

We can reuse the same add32 circuit if we can store the partial results in some storage device, e.g., *register*

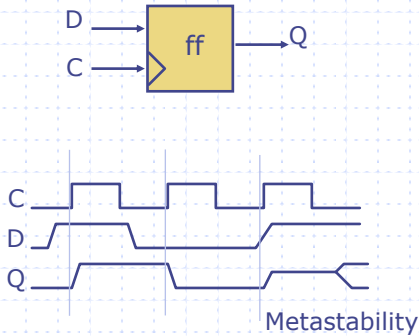
Combinational circuits



Such circuits have no cycles (feedback) or state elements

A simple synchronous state element

Edge-Triggered Flip-flop



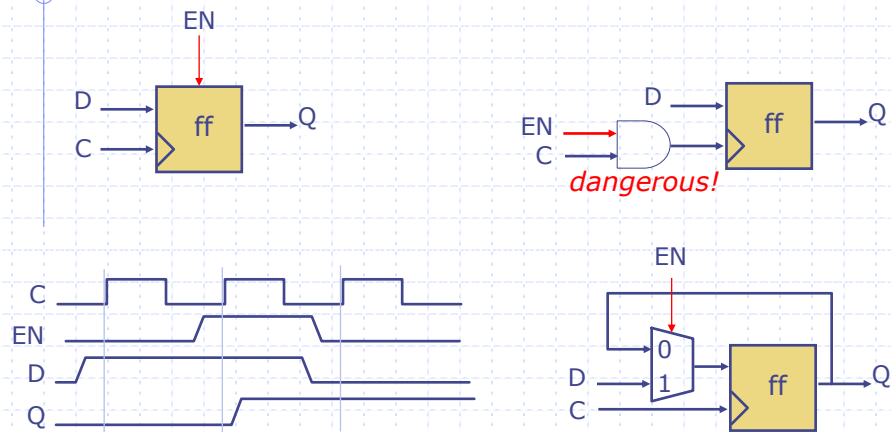
Data is sampled at the rising edge of the clock

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-9

Flip-flops with Write Enables



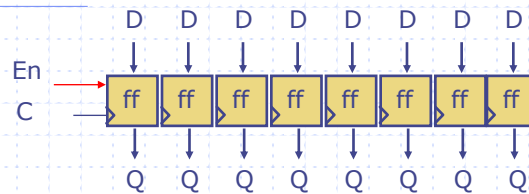
Data is captured only if EN is on

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-10

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)

We can build useful and compact circuits using registers

Circuits containing state elements are called *sequential circuits*

We will only use clocked or synchronous sequential circuits

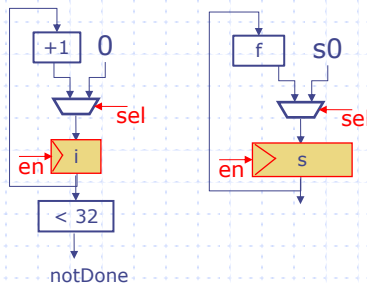
Expressing a loop using registers

```
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

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-13

Expressing sequential circuits in BSV

◆ Sequential circuits, unlike combinational circuits, are *not* expressed structurally (i.e., as wiring diagrams) in BSV

◆ For sequential circuits a designer defines:

- State elements by instantiating modules

```
Reg#(Bit#(32)) s <- mkRegU();
```

```
Reg#(Bit#(6)) i <- mkReg(32);
```

make a 32-bit register which is uninitialized

- Rules which define how state is to be transformed atomically

```
rule step if (i < 32);
```

```
  s <= f(s);
```

```
  i <= i+1;
```

```
endrule
```

make a 6-bit register with initial value 32

actions to be performed when the rule executes

the rule can execute only when its guard is true

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-14

Rule Execution

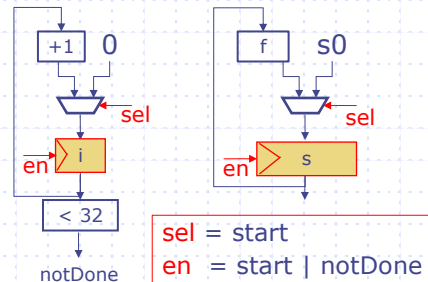
◆ When a rule executes:

- all the registers are read at the beginning of a clock cycle
- the guard and computations to evaluate the next value of the registers are performed
- at the end of the clock cycle registers are updated iff the guard is true

◆ Muxes are need to initialize the registers

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6)) i <- mkReg(32);
```

```
rule step if (i < 32);
    s <= f(s);
    i <= i+1;
endrule
```



Using registers in 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: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);

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

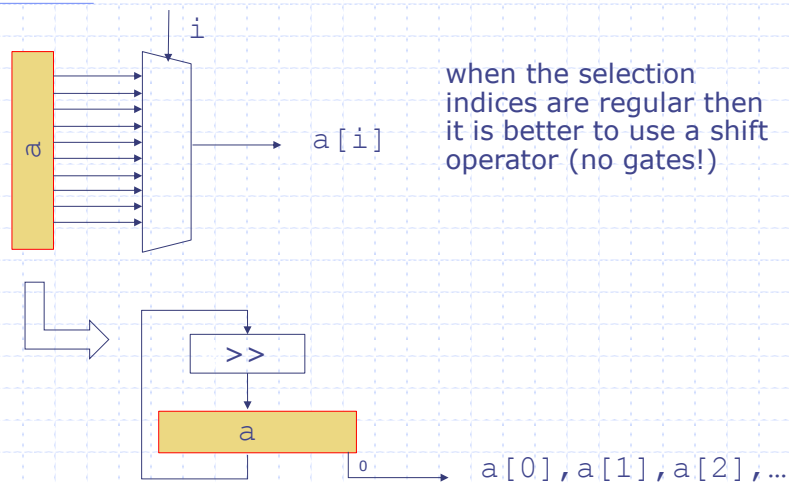
So that the rule won't fire until i is set to some other value

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-17

Dynamic selection requires a mux



December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-18

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

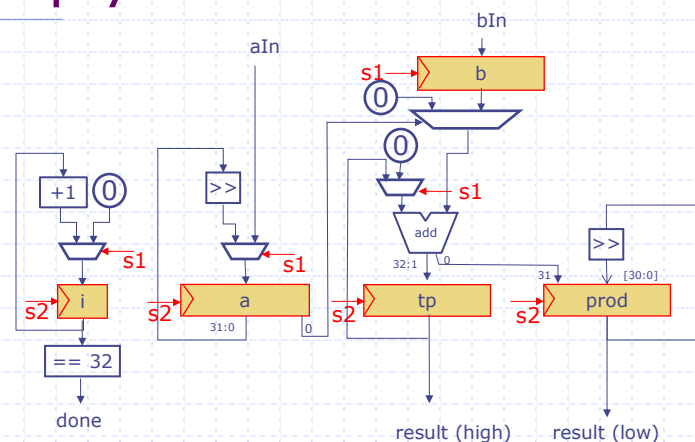
```

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-19

Circuit for Sequential Multiply



`s1 = start_en`
`s2 = start_en | !done`

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-20

Circuit analysis

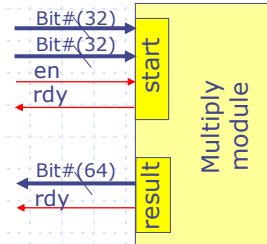
- ◆ Number of add32 circuits has been reduced from 32 to one, though some registers and muxes have been added
- ◆ The longest combinational path has been reduced from 32 serial add32's to one add32 plus a few muxes
- ◆ The sequential circuit will take 31 clock cycles to compute an answer

Modules

We often package sequential circuits into modules to hide the details

Multiply Module

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



- ◆ A module in BSV is like an object in an object-oriented language and can only be manipulated via the methods of its interface
- ◆ However, unlike software, a method in BSV can be applied only when it is “ready”
- ◆ Furthermore, application of an action method, i.e., a method that changes the state of a module, is indicated by asserting the associated enable wire

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-23

Multiply Module

```
module mkMultiply32 (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);

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

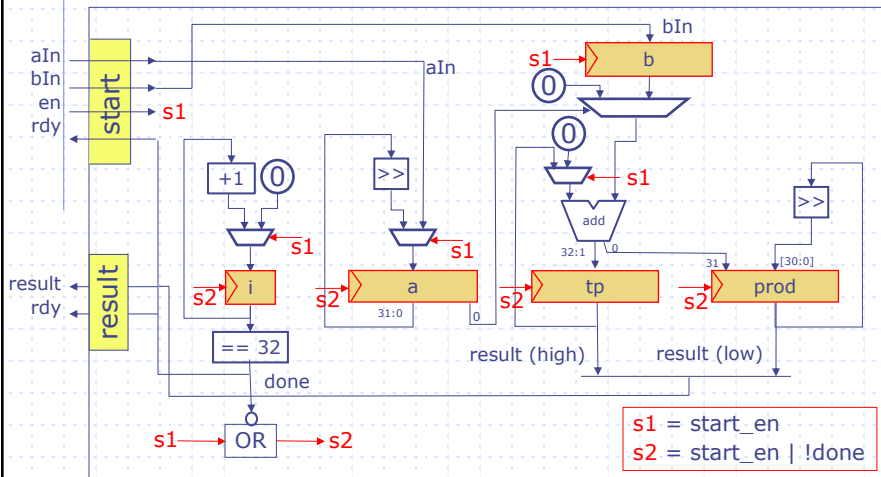
method guards

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-24

Module: Method Interface



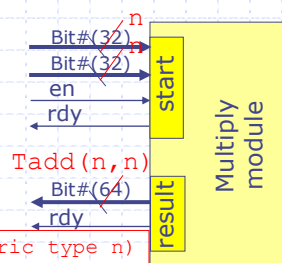
December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-25

Polymorphic Multiply Module

implicit conditions



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

```

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

◆ The module can easily be made polymorphic

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-26

Sequential n-bit multiply

```
module mkMultiplyN (MultiplyN#(n));
  Reg#(Bit#(n)) a <- mkRegU();
  Reg#(Bit#(n)) b <- mkRegU();
  Reg#(Bit#(n)) prod <-mkRegU();
  Reg#(Bit#(n)) tp <- mkReg(0);
  let nv = fromInteger(valueOf(n));
  Reg#(Bit#(TAdd#(TLog#(n),1))) i <- mkReg(nv);
  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[(nv-1):1]};
    tp <= sum[32:1]; 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
```

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-27

Multiply Module

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

- ◆ The same interface can be implemented in many different ways:

```
module mkMultiply (Multiply)
module mkBlockMultiply (Multiply)
module mkBoothMultiply (Multiply)...
```

December 30, 2013

<http://csg.csail.mit.edu/6.s195/CDAC>

L02-28