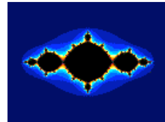
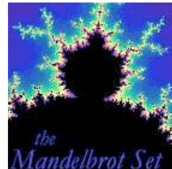
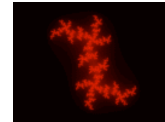






Lab 2 Fractals!

Lecture 5 Revised

Kunle Olukotun
Stanford EE183
Jan 27, 2003

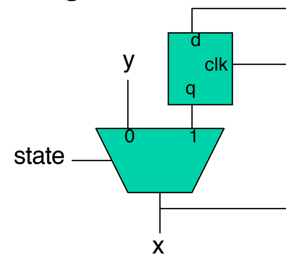
Lab #1

- Due ***Friday*** at 6pm
- No class on Friday!!!
- You can demo anytime before then
- I'll be in the lab around 5pm for demos
- Report due next Monday 1/27 at midnight

Lab #1 Issues

- Gated clocks
  - Don't gate clocks causes skew problems
  - Need a single synchronous system
  - Slow clocks should be used to control enable
- Unspecified nextstate/output logic

```
if (state == 1'b0)
  x = y;
```



Synchronization: Why care?

- Digital Abstraction depends on all signals in a system having a valid logic state
- Therefore, Digital Abstraction depends on reliable synchronization of external events

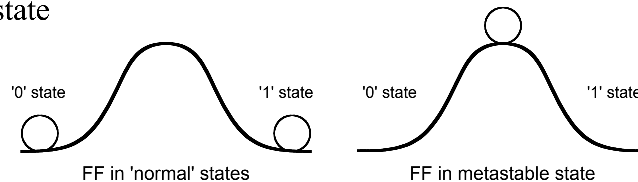
The Real World

- Real World does not respect the Digital Abstraction!
  - Inputs from the Real World are usually asynchronous to your system clock

    - Inputs that come from other synchronous systems are based on a different system clock, which is typically asynchronous to your system clock

Metastability

- When asynchronous events enter your synchronous system, they can cause bistables to go into metastable states
- Every real life bistable (such as a D-latch) has a metastable state



- Once a FF goes metastable (due to a setup time violation, say) we can't say when it will assume a valid logic level or what level it might eventually assume
- The only thing we know is that the probability of a FF coming out of a metastable state increases exponentially with time

Mean Time Between Failures

- For a FF we can compute its MTBF, which is a figure of merit related to metastability.

$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o f a}$$

t_r resolution time (time since clock edge)
f sampling clock frequency
a asynchronous event frequency
τ and T_o FF parameters

For a typical .25um ASIC library FF
t_r = 2.3ns
τ = 0.31ns
T_o = 9.6as

For f = 100MHz,
a = 1MHz

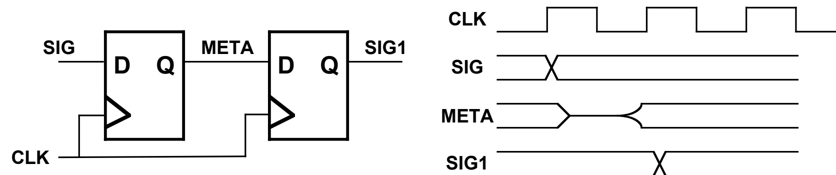
MTBF = 20.1 days

Synchronizer Requirements

- Synchronizers must be designed to reduce the chances system failure due to metastability
- Synchronizer requirements
  - Reliable [high MTBF]
  - Low latency [works as quickly as possible]
  - Low power/area impact

Single signal Synchronizer

- Traditional synchronizer
  - SIG is asynchronous, and META might go metastable from time to time
  - However, as long as META resolves before the next clock period SIG1 should have valid logic levels
  - Place FFs close together to allow maximum time for META to reslove



Single Synchronizer analysis

- MTBF of this system is roughly:

$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o fa} \times \frac{e^{(t_r/\tau)}}{T_o f}$$

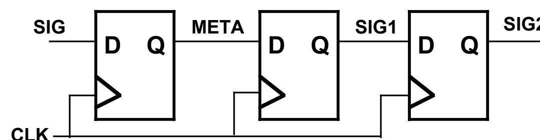
For a typical .25um ASIC library FF

t_r = 2.3ns
τ = 0.31ns
T_o = 9.6as

For f = 100MHz, a = 1MHz

MTBF = 9.57×10^{10} years
Age of Earth = 5×10^{9} years

- Can increase MTBF by adding more series stages



Course Logistics

- Labs due every two weeks
  - Prelab report due a week before demo
    - The prelab is important
  - Demo due on Fridays at **6pm**
  - Report due by following Monday at midnight

Lab	Pre-Lab Report Due	Demo Due by 5 pm	Final Report Due
1	Jan 17	Jan 24	Jan 27
2	Jan 31	Feb 7	Feb 10
3	Feb 14	Feb 21	Feb 24
4	Feb 28	Mar 7	Mar 10

- All reports are submitted by email using PDF

A Few More Words about Verilog

- Comment your verilog!
  - Make it self-documenting
  - Include a header that says what the block does
- Declaring FSMs

```
    // State Machine Declarations...
    `define FSM_NUM_DFF 2
    `define STATE0 2'b00
    `define STATE1 2'b01
    `define STATE2 2'b10
    `define STATE3 2'b11
reg [`FSM_NUM_DFF-1:0] next_state_d; // input to FF
wire [`FSM_NUM_DFF-1:0] state_q; // output from FF
```

Documentation

- Read requirements in Tao of EE183
- What things would a competent engineer in the field need to understand to modify or use this design?
  - More often than not, that engineer is ***YOU*** in 12 months.
- Incrementally work on the documentation—don't leave it for after the design is complete!!
  - Prelab helps with this
- All documentation submitted electronically in PDF format

Documentation Requirements

- Abstract- overview of what you did
- Design - how you did it
  - Hierarchy and design decisions
  - descriptions of FSMs and how they interact
  - Design aids (such as CoreGen modules) and how you used them and how they work.
- Results - what you achieved
  - including top speed and area usage
- Conclusions - what you thought of it
  - Was it a good/bad design/ implementation?
  - What would you do differently next time?
- A1. Simulations
  - show simulations and scripts with annotations
- A2. Implementation
  - your verilog code with comments
- A3. Performance metrics
  - a screen shot of your layout on the FPGA, timing and usage reports.

Mandelbrot Fractal

- The Mandelbrot set is the set of points in the complex *c*-plane that do not go to infinity when iterating $z_{n+1} = z_n^2 + c$ starting with $z = 0$. One can avoid the use of complex numbers by using $z = x + iy$ and $c = a + ib$, and computing the orbits in the *ab*-plane for the 2-D mapping

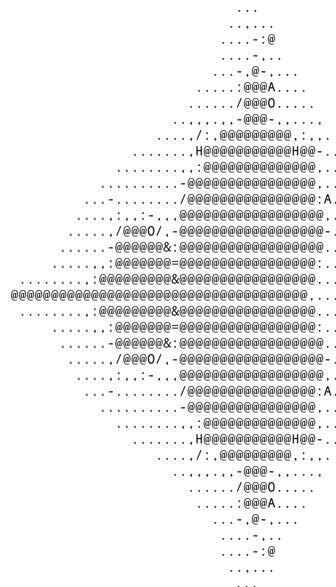
$$x_{n+1} = x_n^2 - y_n^2 + a$$
$$y_{n+1} = 2x_n y_n + b$$

with initial conditions $x = y = 0$ (or equivalently $x = a$ and $y = b$). It can be proved that the orbits are unbounded if $|z| > 2$ (i.e., $x^2 + y^2 > 4$).
  - http://www.olympus.net/personal/dewey/mandelbrot.html
  - http://www.jade-leaves.com/mandelbrot_set/index.shtml

Julia Set

- Very similar for the next state generation except the (a,b)
  - these are constants throughout the calculation
- Sample code for matlab and perl is in
  - http://www.stanford.edu/class/ee183/fractals/

Perl Mandelbrot Output

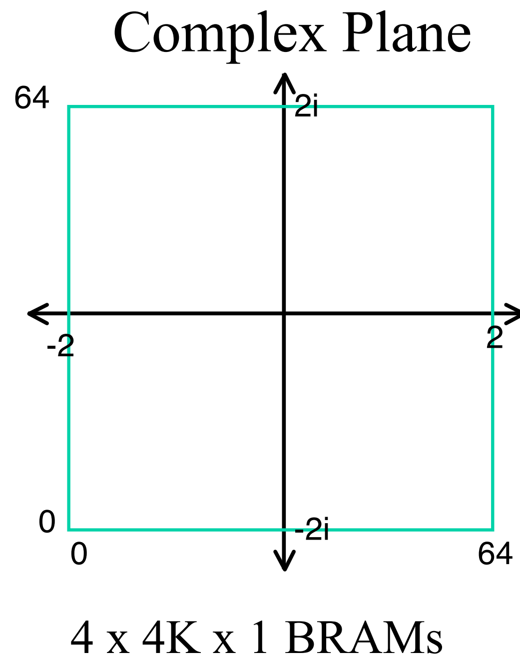


Note: the perl ouput looks funny because ascii character dimensions are not proportional

Lab 2

- Display the Mandelbrot Fractal on the VGA monitor
- Use the Sega Controller to select the constant for the Julia set and then draw it
- Calculate both from –2 to 2 on a 64x64 grid with 4 bit color
- Demonstration:
  - http://www.unca.edu/~mcmcclur/java/Julia/
- Create a Julia Animation

Complex Plane



4 x 4K x 1 BRAMs

Game Plan

- Reuse concepts from Lab1
  - Sega Gamepad frontend
  - Dual Port BRAM to calculate into one port and VGA scans the other port
    - Only need 1 DPBRAM because we don't need any state to compute *from*
  - Cursor location to select Julia constant
- Translate Perl/Matlab into hardware
  - Special purpose hardware

Mandelbrot Perl Code

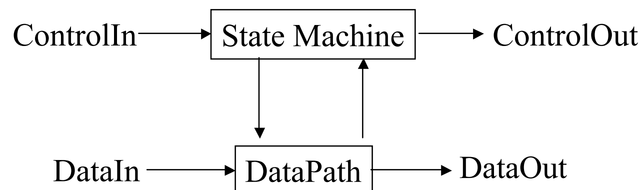
$x_{n+1} = x_n^2 - y_n^2 + a$
$y_{n+1} = 2x_ny_n + b$

```
#!/usr/local/bin/perl
$Cols=64; $Lines=64;
$MaxIter=63;
$MinRe=-2.0; $MaxRe=2.0;
$MinIm=-2.0; $MaxIm=2.0;
@chars=(' ','.',',','-
',':','/','=','H','O','A','M','%','&','$','#','@','_');

for($Im=$MaxIm;$Im>=$MinIm;$Im-=($MaxIm-$MinIm)/$Lines)
{ for($Re=$MinRe;$Re<=$MaxRe;$Re+=($MaxRe-$MinRe)/$Cols)
  { $zr=$Re; $zi=$Im;
    for($n=0;$n<$MaxIter;$n++)
    { $a=$zr*$zr; $b=$zi*$zi;
      if($a+$b>4.0) { last; }
      $zi=2*$zr*$zi+$Im; $zr=$a-$b+$Re;
    }
    print $chars[$n/4];
  }
  print "\n";
}
```

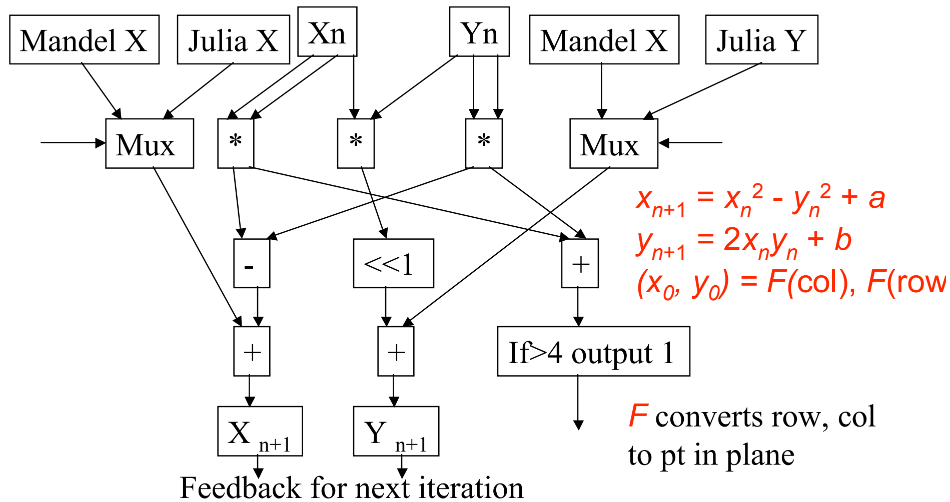
Separation of Control & Datapath

- You have already been doing this
  – Remember the tutorial

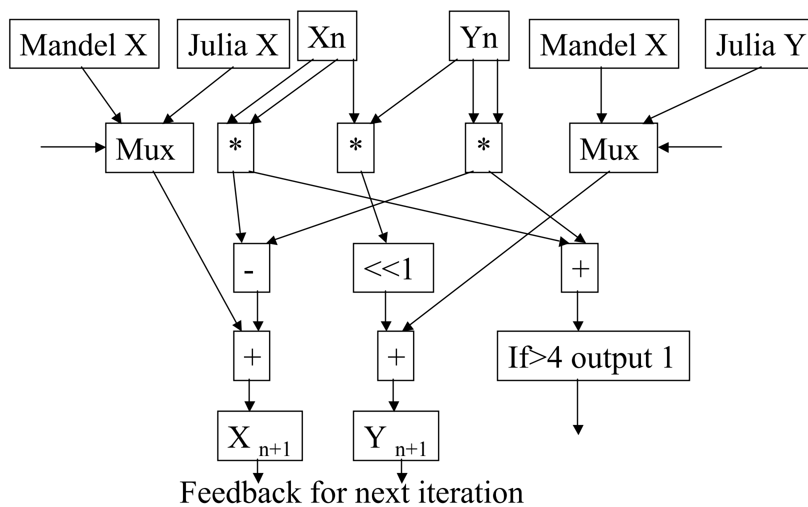


Fractal Generation Datapath

Use counters for row, col, and n



Multicycle Datapath

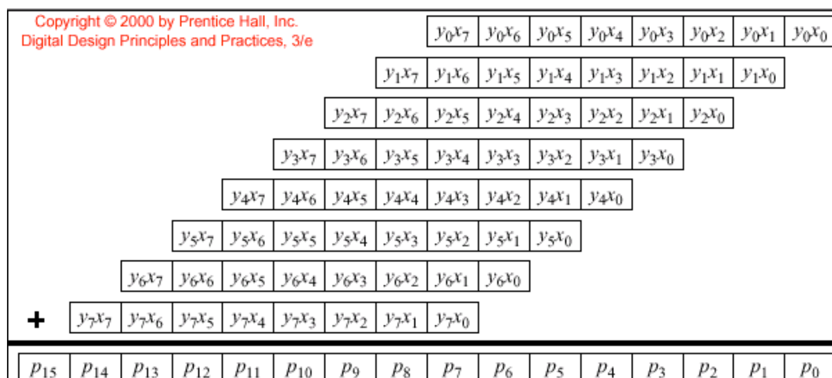


We need multipliers!

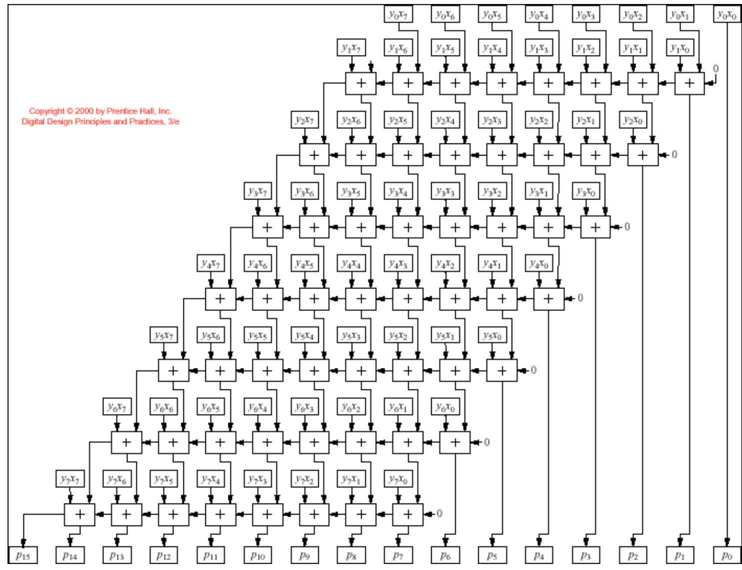
- All other elements in datapath we know how to build..
- We discussed multipliers in EE121…

Multipliers

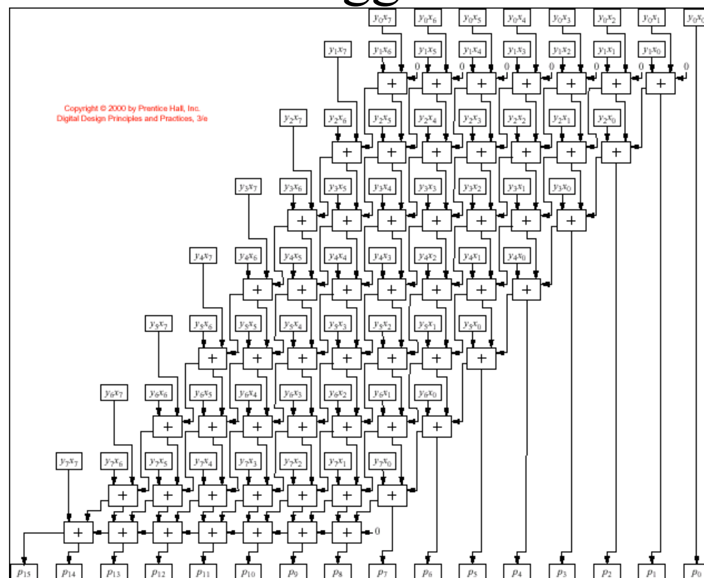
- Repeated Addition



Initial Architecture



More Aggressive



Multipliers Summary

- Lots of clever architectures out there. They all do the same thing—multiply!
- Consider routing delay in addition to logic delay.

Fractions?

- Those were all Integer Multipliers
  - Signed operands in twos complement work fine
- Our algorithm calls for fractional arithmetic
  - Normally implemented as Floating Point Math
    - Very painful
  - So use Fixed Point Math
    - Assume numbers are always in the form: X.Y where X and Y have constant width

Fixed Point Math

- Addition/Subtraction same as integers
- Multiplication is a little different
- Note that the coregen multiplier has several pipeline options

Multiplying Fixed Point Numbers

- The maximum number of digits you can have in the product is the sum of the number of digits in the multiplier and the multiplicand.
- The number of digits of fraction in the product is the sum of the number of digits of fraction in the multiplier and the number of digits of fraction in the multiplicand.
- An integer is a fixed point number with 0 digits of fraction.
- This means that when you multiply a fixed point number in an M.N format by an integer the result is a number in M.N format. For example let's multiply integer 7 by 3 in 2.2 fixed point format.
- **`0007 * 03.00 = 21.00`**
- When you multiply 2 fixed point numbers in M.N format you get a result that has 2N digits of fraction. For example:
- **`07.00 * 03.00 = 21.0000`**
- To get back to an M.N format number you throw away the low order N digits of the fraction. That is, you shift the number right by N digits. Truncating the low order digits is a source of error in fixed point arithmetic, but don’t worry about it

Arithmetic Right Shift

- Multiplication requires an *arithmetic* right shift after the computation
  - Divide to remove the least significant digits
    - Restore the location of the decimal point
  - Verilog >> is logical shift
    - Construct Arithmetic Shift from conditional (?:)
- Aside: Verilog 2000 has >>>/<<< as arithmetic shifts.

Fixed Point Partition?

- How big should the integer part or fractional part be?
  - Want to minimize these since multipliers grow quickly in size and latency with operand size
  - Don't want them so small that overflow of the integer part occurs (results in aliasing) or that the fractional part has large quantization error
- We stop the loop when magnitude is greater than 4
  - Use that knowledge to approximate size of intermediate operands

Julia Set Animation Constants

- Previous gif: “It's a sequence of Julia sets for the points at 5° intervals around the unit circle.”
  - Why should they be on a circle? What other trajectories would be “interesting?”
- Could calculate these values on the fly
  - Use the Sine-Cosine generator in Coregen
- Or precompute the values and store them in a ROM
- Use the CORDIC algorithm
  - http://www.opencores.org/projects/cordic/

Julia Set Animation

- How long does each Julia image take to create?
- (64*64*64*7*1/50e6) = 0.036s
- So can calculate them in real time
- Would double buffering help?
  - Not sure…

http://homepages.enterprise.net/scruss/julia_anim.html