Pipelining & Verilog

- Division
- Latency & Throughput
- Pipelining to increase throughput
- Retiming
- Verilog Math Functions
Sequential Divider

Assume the Dividend (A) and the divisor (B) have N bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction at a time and then cycle the circuit N times. This circuit works on unsigned operands; for signed operands one can remember the signs, make operands positive, then correct sign of result.

Init: P←0, load A and B
Repeat N times {
    shift P/A left one bit
    temp = P-B
    if (temp > 0)
        {P←temp, A_{LSB}←1}
    else A_{LSB}←0
}
Done: Q in A, R in P
// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
// sign -- 0 for unsigned, 1 for twos complement

// It uses a simple restoring divide algorithm.

module divider #(parameter WIDTH = 8)
    (input clk, sign, start,
    input [WIDTH-1:0] dividend, input [WIDTH-1:0] divider,output reg [WIDTH-1:0] quotient,output [WIDTH-1:0] remainder;output ready);

    reg [WIDTH-1:0]  quotient_temp;
    reg [WIDTH*2-1:0] dividend_copy, divider_copy, diff;reg negative_output;

    wire [WIDTH-1:0] remainder = (!negative_output) ?
                        dividend_copy[WIDTH-1:0] : ~dividend_copy[WIDTH-1:0] + 1'b1;

    reg [5:0] bit;
    reg del_ready = 1;wire ready = (!bit) & ~del_ready;

    wire [WIDTH-2:0] zeros = 0;
    initial bit = 0;
    initial negative_output = 0;

    always @( posedge clk ) begin
        del_ready <= !bit;
        if( start ) begin
            bit = WIDTH;
            quotient = 0;
            quotient_temp = 0;
            dividend_copy = (!sign || !dividend[WIDTH-1]) ?
                            {1'b0,zeros,dividend} :  {1'b0,zeros,~dividend + 1'b1};
            divider_copy = (!sign || !divider[WIDTH-1]) ?
                            {1'b0,divider,zeros} :{1'b0,~divider + 1'b1,zeros};

            negative_output = sign &&
                              ((divider[WIDTH-1] && !dividend[WIDTH-1])
                           |  (!dividend[WIDTH-1] | !divider[WIDTH-1]));

            end
        else if ( bit > 0 ) begin
            diff = dividend_copy - divider_copy;
            quotient_temp = quotient_temp << 1;
            if( !diff[WIDTH*2-1] ) begin
                dividend_copy = diff;
                quotient_temp[0] = 1'd1;
            end
            quotient = (!negative_output) ?
                        quotient_temp :
                        ~quotient_temp + 1'b1;
            divider_copy = divider_copy >> 1;
            bit = bit - 1'b1;
        end
    end
endmodule

L. Williams MIT '13
Math Functions in Coregen

Wide selection of math functions available
Coregen Divider

Details in data sheet.

not necessary many applications
Coregen Divider

Chose minimum number for application

Ready For Data: needed if clocks/divide > 1
Performance Metrics for Circuits

Circuit Latency (L): time between arrival of new input and generation of corresponding output.

For combinational circuits this is just $t_{PD}$.

Circuit Throughput (T): Rate at which new outputs appear.

For combinational circuits this is just $1/t_{PD}$ or $1/L$. 
Coregen Divider Latency

Latency dependent on dividend width + fractional reminder width

Table 4: Latency of Fixed-point Solution Based on Divider Parameters

<table>
<thead>
<tr>
<th>Signed</th>
<th>Fractional</th>
<th>Clks/Div</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>1</td>
<td>M+2</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>&gt;1</td>
<td>M+3</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>1</td>
<td>M+F+2</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>&gt;1</td>
<td>M+F+3</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>1</td>
<td>M+4</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>&gt;1</td>
<td>M+5</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>1</td>
<td>M+F+4</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>&gt;1</td>
<td>M+F+5</td>
</tr>
</tbody>
</table>

Note: M=dividend width, F=fractional remainder width.

The divclk_sel parameter allows a range of choices of throughput versus area. With divclk_sel = 1, the core is fully pipelined, so it will have maximal throughput of one division per clock cycle, but will occupy the most area. The divclk_sel selections of 2, 4 and 8 reduce the throughput by those respective factors for smaller core sizes.
For combinational logic:

\[ L = t_{PD}, \]
\[ T = \frac{1}{t_{PD}}. \]

We can’t get the answer faster, but are we making effective use of our hardware at all times?

F & G are “idle”, just holding their outputs stable while H performs its computation.
Retiming: A very useful transform

Retiming is the action of moving registers around in the system
- Registers have to be moved from ALL inputs to ALL outputs or vice versa

Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of the edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.

Benefits of retiming:
- Modify critical path delay
- Reduce total number of registers
Retiming Combinational Circuits aka “Pipelining”

Assuming ideal registers:
- \( t_{PD} = 0, t_{SETUP} = 0 \)

\[ L = 45 \]
\[ T = 1/45 \]

\[ t_{CLK} = 25 \]
\[ L = 2 \times t_{CLK} = 50 \]
\[ T = 1/t_{CLK} = 1/25 \]
The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.
Pipeline Conventions

DEFINITION:
A **K-Stage Pipeline** ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.

A **COMBINATIONAL CIRCUIT** is thus an 0-stage pipeline.

CONVENTION:
Every pipeline stage, hence every K-Stage pipeline, has a register on its **OUTPUT** (not on its input).

ALWAYS:
The **CLOCK** common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register $t_{PD}$ PLUS (output) register $t_{SETUP}$.

---

The **LATENCY** of a K-pipeline is K times the period of the clock common to all registers.

The **THROUGHPUT** of a K-pipeline is the frequency of the clock.
Ill-formed pipelines

Consider a BAD job of pipelining:

For what value of $K$ is the following circuit a $K$-Pipeline? _________ none

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed $K$ pipeline!
A pipelining methodology

Step 1:
Add a register on each output.

Step 2:
Add another register on each output. Draw a cut-set contour that includes all the new registers and some part of the circuit. Retime by moving regs from all outputs to all inputs of cut-set.

Repeat until satisfied with T.

STRATEGY:
Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).

\[
\begin{align*}
A & \quad 4 \text{nS} \\
B & \quad 3 \text{nS} \\
C & \quad 8 \text{nS} \\
D & \quad 4 \text{nS} \\
E & \quad 2 \text{nS} \\
F & \quad 5 \text{nS} \\
\end{align*}
\]

\[T = 1/8\text{ns}\]
\[L = 24\text{ns}\]
Pipeline Example

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don’t improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>
No pipeline
assign y = G(x); // logic for y
assign pixel = C(y) // logic for pixel

Pipeline
always @(posedge clock) begin
  y2 <= G(x); // pipeline y
  pixel <= C(y2) // pipeline pixel
end

Lab 3 Pong
- G = game logic 8ns tpd
- C = draw round puck, use multiply with 9ns tpd
- System clock 65mhz = 15ns period - opps

Latency = 2 clock cycles!
Implications?
Increasing Throughput: Pipelining

Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.

Throughput = $1/4t_{PD,FA}$ instead of $1/8t_{PD,FA}$

= register

$\triangleright$ = register
How about $t_{PD} = 1/2t_{PD,FA}$?

= register
Timing Reports

65mhz = 27mhz*2.4

Multiple: 7.251ns

Total Propagation delay: 34.8ns
History of Computational Fabrics

- Discrete devices: relays, transistors (1940s-50s)
- Discrete logic gates (1950s-60s)
- Integrated circuits (1960s-70s)
  - e.g. TTL packages: Data Book for 100's of different parts
- Gate Arrays (IBM 1970s)
  - Transistors are pre-placed on the chip & Place and Route software puts the chip together automatically – only program the interconnect (mask programming)
- Software Based Schemes (1970’s-present)
  - Run instructions on a general purpose core
- Programmable Logic (1980’s to present)
  - A chip that be reprogrammed after it has been fabricated
  - Examples: PALs, EPROM, EEPROM, PLDs, FPGAs
  - Excellent support for mapping from Verilog
- ASIC Design (1980’s to present)
  - Turn Verilog directly into layout using a library of standard cells
  - Effective for high-volume and efficient use of silicon area
Reconfigurable Logic

- **Logic blocks**
  - To implement combinational and sequential logic
- **Interconnect**
  - Wires to connect inputs and outputs to logic blocks
- **I/O blocks**
  - Special logic blocks at periphery of device for external connections

**Key questions:**
- How to make logic blocks programmable? (after chip has been fabbed!)
- What should the logic granularity be?
- How to make the wires programmable? (after chip has been fabbed!)
- Specialized wiring structures for local vs. long distance routes?
- How many wires per logic block?
Programmable Array Logic (PAL)

- Based on the fact that any combinational logic can be realized as a sum-of-products
- PALs feature an array of AND-OR gates with programmable interconnect
RAM Based Field Programmable Logic - Xilinx

Programmable Interconnect

Configurable Logic Blocks (CLBs)

I/O Blocks (IOBs)
LUT Mapping

- N-LUT direct implementation of a truth table: any function of n-inputs.
- N-LUT requires $2^N$ storage elements (latches)
- N-inputs select one latch location (like a memory)

4LUT example

Latches set by configuration bitstream
Configuring the CLB as a RAM

Memory is built using Latches not FFs

Read is same a LUT Function!

Figure 4: 16x2 (or 16x1) Edge-Triggered Single-Port RAM
Xilinx 4000 Interconnect

Figure 28: Single- and Double-Length Lines, with Programmable Switch Matrices (PSMs)
Xilinx 4000 Interconnect Details

Wires are not ideal!
Add Bells & Whistles

The Virtex II CLB (Half Slice Shown)
Adder Implementation

\[ Y = A \oplus B \oplus \text{Cin} \]

LUT: \( A \oplus B \)

Dedicated carry logic

1 half-Slice = 1-bit adder
#### FPGA's

<table>
<thead>
<tr>
<th>Part Number</th>
<th>LH75T</th>
<th>LH130T</th>
<th>LH190T</th>
<th>LH240T</th>
<th>LH965T</th>
<th>LH550T</th>
<th>LH790</th>
<th>SK310T</th>
<th>SK470T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Cells</td>
<td>74.8k</td>
<td>128k</td>
<td>200k</td>
<td>241k</td>
<td>364k</td>
<td>50.1k</td>
<td>76.3k</td>
<td>97.2k</td>
<td>47.2k</td>
</tr>
<tr>
<td>CLB Flip-Flops</td>
<td>931k</td>
<td>160k</td>
<td>252k</td>
<td>201k</td>
<td>455k</td>
<td>667k</td>
<td>548k</td>
<td>394k</td>
<td>596k</td>
</tr>
<tr>
<td>Maximum Distributed RAM (kbits)</td>
<td>1,045</td>
<td>1,740</td>
<td>3,040</td>
<td>3,550</td>
<td>4,180</td>
<td>5,700</td>
<td>8,260</td>
<td>5,090</td>
<td>7,640</td>
</tr>
<tr>
<td>Block RAM/FIFO w/ ECC (96k bits each)</td>
<td>156</td>
<td>394</td>
<td>844</td>
<td>416</td>
<td>416</td>
<td>667</td>
<td>720</td>
<td>704</td>
<td>1,064</td>
</tr>
<tr>
<td>Total Block RAM (kbits)</td>
<td>5,816</td>
<td>9,504</td>
<td>12,984</td>
<td>14,578</td>
<td>14,578</td>
<td>22,752</td>
<td>25,590</td>
<td>29,204</td>
<td>29,204</td>
</tr>
<tr>
<td>Mixed Mode Clock Managers (AMCM)</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>12</td>
<td>18</td>
<td>18</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>DSP48E1 Slices</td>
<td>298</td>
<td>480</td>
<td>640</td>
<td>765</td>
<td>576</td>
<td>364</td>
<td>364</td>
<td>1,344</td>
<td>2,916</td>
</tr>
<tr>
<td>PCI Express Interface Blocks</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10/100/1000 Ethernet MAC Blocks</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX Low-Power Transceivers</td>
<td>12</td>
<td>20</td>
<td>20</td>
<td>24</td>
<td>24</td>
<td>36</td>
<td>0</td>
<td>24</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Package</th>
<th>Area (Pitch)</th>
<th>Maximum User I/O: Select I/O Interface Pins (GTX Transceivers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF484</td>
<td>24 x 24 mm (0.6 mm)</td>
<td>240 (8)</td>
</tr>
<tr>
<td>FF704</td>
<td>92 x 92 mm (0.8 mm)</td>
<td>400 (12)</td>
</tr>
<tr>
<td>FF1156</td>
<td>25 x 25 mm (1.0 mm)</td>
<td>600 (20)</td>
</tr>
<tr>
<td>FF1766</td>
<td>42.5 x 42.5 mm (1.0 mm)</td>
<td>720 (24)</td>
</tr>
<tr>
<td>FF1760</td>
<td>42.5 x 42.5 mm (1.0 mm)</td>
<td>1,200 (0)</td>
</tr>
</tbody>
</table>

* Preliminary product information, subject to change. Please contact your Xilinx representative for the latest information.

**CLB**

| Virtex 2  | 8,448 | 1,056 kbit | 2,592 kbit | 144 (18 x 18) |
| Virtex 6  | 667,000 | 6,200 kbit | 22,752 kbit | 1,344 (25 x 18) |
| Spartan 3E | 240 | 15 kbit | 72 kbit | 4 (18 x 18) |
| Artix-7 A100 | 7,925 | 1,188 kbit | 4,860 kbit | 240 (25 x 18) |
Design Flow - Mapping

- Technology Mapping: Schematic/HDL to Physical Logic units
- Compile functions into basic LUT-based groups (function of target architecture)

```verilog
always @(posedge clock or negedge reset)
begin
  if (! reset)
    q <= 0;
  else
    q <= (a&b&c) || (b&d);
end
```
Design Flow – Placement & Route

- **Placement** - assign logic location on a particular device

- **Routing** – iterative process to connect CLB inputs/outputs and IOBs. Optimizes critical path delay – *can take hours or days for large, dense designs*

  Iterate placement if timing not met

  Satisfy timing? → Generate Bitstream to config device

Challenge! Cannot use full chip for reasonable speeds (wires are not ideal).
Typically no more than 50% utilization.
Example: Verilog to FPGA

module adder64 (
    input [63:0] a, b;
    output [63:0] sum);

assign sum = a + b;
endmodule

Virtex II – XC2V2000
How are FPGAs Used?

- **Prototyping**
  - Ensemble of gate arrays used to emulate a circuit to be manufactured
  - Get more/better/faster debugging done than with simulation

- **Reconfigurable hardware**
  - One hardware block used to implement more than one function

- **Special-purpose computation engines**
  - Hardware dedicated to solving one problem (or class of problems)
  - Accelerators attached to general-purpose computers (e.g., in a cell phone!)
Summary

• FPGA provide a flexible platform for implementing digital computing
• A rich set of macros and I/Os supported (multipliers, block RAMS, ROMS, high-speed I/O)
• A wide range of applications from prototyping (to validate a design before ASIC mapping) to high-performance spatial computing
• Interconnects are a major bottleneck (physical design and locality are important considerations)
module sample_tf;
  // Inputs
  reg bit_in;
  reg [3:0] bus_in;

  // Outputs
  wire out_bit;
  wire [7:0] out_bus;

  // Instantiate the Unit Under Test (UUT)
  sample uut (
    .bit_in(bit_in),
    .bus_in(bus_in),
    .out_bit(out_bit),
    .out_bus(out_bus)
  );

  initial begin
    // Initialize Inputs
    bit_in = 0;
    bus_in = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

    end

endmodule