CS152
Computer Architecture and Engineering
Lecture 6

Verilog (finish)
Multiply, Divide, Shift

February 11, 2004
John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Review from last time

° Design Process
  - Design Entry: Schematics, HDL, Compilers
  - High Level Analysis: Simulation, Testing, Assertions
  - Technology Mapping: Turn design into physical implementation
  - Low Level Analysis: Check out Timing, Setup/Hold, etc

° Verilog – Three programming styles
  - Structural: Like a Netlist
    - Instantiation of modules + wires between them
  - Dataflow: Higher Level
    - Expressions instead of gates
  - Behavioral: Hardware programming
    - Full flow-control mechanisms
    - Registers, variables
    - File I/O, console display, etc

Verilog subtle ty: Blocking Assignments

° Blocking Assignments:
  - Assignments happen more like programming language (sequential code)
  - Both Right and left sides evaluated completely
  - Wait until assignment before going on
    - Can cause unexpected results when connecting output to logic in other always blocks.
    - Also a bit strange with delays on left hand side (LHS)

° Example:
reg E, C;
always @(posedge clk)
begin
  E = ~A;
  C = ~E;
end

Verilog subtlety: Non-Blocking Assignments

° Non-blocking Assignments:
  - All right-hand sides evaluated immediately
  - Then assignments occur
  - If no delays, often want output ports to be assigned with non-blocking assignments

° Example:
reg E, C;
always @(posedge clk)
begin
  E <= ~A;
  C <= ~E;
end
Sequential Logic (Revisited: better scheduling)

Must be careful mixing zero-time blocking assignments and edge-triggering: Probably won’t do what you expect when connecting it to other things!

```verilog
module FF (CLK,Q,D);
input D, CLK;
output Q; reg Q;
always @ (posedge CLK) Q <= D;
endmodule // FF

Good: Doesn’t output until “after edge”
```

```verilog
module FF (CLK,Q,D);
input D, CLK;
output Q; reg Q;
always @ (posedge CLK) Q = #5 D;
endmodule // FF

Good: Outputs 5 units “after edge”
```

Probably Not what you Expect:
- Hold time of 5 units
- glitches < 5 units ignored

A final word on Verilog

° Verilog does not turn hardware design into writing programs!
  • Since Verilog looks similar to programming languages, some think that they can design hardware by writing programs.
    - NOT SO.
  • Verilog is a hardware description language.
    - The best way to use it is to first figure out the circuit you want, then figure out how to describe it in Verilog.
    - The behavioral construct hides a lot of the circuit details but you as the designer must still manage:
      - the structure
      - data-communication
      - Parallelism
      - timing of your design.
    - Not doing so leads to very inefficient designs!

° Read the document on non-blocking assignment in Verilog that I put up on the handouts page. Lots of very interesting things!

How Program: FPGA Generic Design Flow

° Design Entry:
  • Create your design files using:
    - schematic editor or
    - hardware description language (Verilog, VHDL)

° Design “implementation” on FPGA:
  • Partition, place, and route (“PPR”) to create bit-stream file
  • Divide into CLB-sized pieces, place into blocks, route to blocks

° Design verification:
  • Use Simulator to check function.
  • Other software determines max clock frequency.
  • Load onto FPGA device (cable connects PC to board)
    - check operation at full speed in real environment.

Idealized FPGA Logic Block

° 4-input Look Up Table (4-LUT)
  • implements combinational logic functions

° Register
  • optionally stores output of LUT
  • Latch determines whether read reg or LUT
4-LUT Implementation

- n-bit LUT is actually implemented as a $2^n \times 1$ memory:
  - inputs choose one of $2^n$ memory locations.
  - memory locations (latches) are normally loaded with values from user's configuration bit stream.
  - Inputs to mux control are the CLB (Configurable Logic Block) inputs.

- Result is a general purpose “logic gate”.
- n-LUT can implement any function of n inputs!

LUT as general logic gate

- An n-lut as a direct implementation of a function truth-table
- Each latch location holds value of function corresponding to one input combination

Example: 4-lut

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>F(0,0,0,0)</th>
<th>F(0,0,0,1)</th>
<th>store in 1st latch</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Example: 2-lut

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Implements *any* function of 2 inputs.

How many functions of n inputs?

2/11/03 ©UCB Spring 2004 CS152 / Kubiatowicz

Additional application: Distributed RAM

- CLB LUT configurable as Distributed RAM
  - A LUT equals 16x1 RAM
  - Implements Single and Dual-Ports
  - Cascade LUTs to increase RAM size

- Synchronous write

- Synchronous/Asynchronous read
  - Accompanying flip-flops used for synchronous read

Block RAM (Extra RAM not using LUTs)

- Most efficient memory implementation
  - Dedicated blocks of memory
- Ideal for most memory requirements
  - Virtex-E XCV2000 has 160? blocks
    - 4096 bits per blocks (4K x 1, 2K x 4, 512 x 8, 256 x 16)
  - Use multiple blocks for larger memories

- Builds both single and true dual-port RAMs
- CORE Generator provides custom-sized block RAMs
  - Quickly generates optimized RAM implementation
Additional Application: Shift Register

- Each LUT can be configured as shift register
  - Serial in, serial out
- Saves resources: can use less than 16 FFs
- Faster: no routing
- Note: CAD tools determine with CLB used as LUT, RAM, or shift register, rather than up to designer

Example Partition, Placement, and Route

- Idealized FPGA structure:
  - collection of gates and flip-flops
- Example Schematic Circuit:
  - collection of gates and flip-flops
- Circuit combinational logic must be “covered” by 4-input 1-output “gates”.
  - Flip-flops from circuit must map to FPGA flip-flops.
  - (Best to preserve “closeness” to CL to minimize wiring.)
  - Placement in general attempts to minimize wiring.

Xilinx Virtex-E Routing Hierarchy

- Internal 3-state Bus
- Long lines and Global lines
- Buffered Hex lines (1/6 blocks)
- Single-length lines

- 24 single-length lines
  - Route GRM signals to adjacent GRMs in 4 directions
- 96 buffered hex lines
  - Route GRM (general routing matrix) signals to another GRMs six blocks away in each of the 4 directions
- 12 buffered Long lines
  - Routing across top and bottom and left and right

Virtex-E Configurable Logic Block (CLB)

- 2 “logic slices” / CLB, two 4-LUTs / slice
  => Four 4-LUTs / CLB
Virtex-E CLB Slice Structure

- Each slice contains two sets of the following:
  - Four-input LUT
    - Any 4-input logic function
    - Or 16-bit x 1 sync RAM
    - Or 16-bit shift register
  - Carry & Control
    - Fast arithmetic logic
    - Multiplexer logic
    - Multiplier logic
  - Storage element
    - Latch or flip-flop
    - Set and reset
    - True or inverted inputs
    - Sync. or async. control

Virtex-E Dedicated Expansion Multiplexers

- Since 4-LUT has 4 inputs, max is 2:1 Mux (2 inputs, 1 control line)
- MUXF5 combines 2 LUTs to create
  - 4x1 multiplexer
  - Or any 5-input function (5-LUT)
  - Or selected functions up to 9 inputs
- MUXF6 combines 2 slices to form
  - 8x1 multiplexer
  - Or any 6-input function (6-LUT)
  - Or selected functions up to 19 inputs
- Dedicated muxes are faster and more space efficient

Details of Virtex-E Slice

- Very fast ripple carry: (24-bit @ 100 MHz)
- Multiplexors to help combine CLBs into larger multiplexor

Administrivia

- Change in Sections (sorry!):
  - Section 1: 3107 Etcheverry  47 Evans
  - Section 2: 3107 Etcheverry  105 Latimer
- Starts tomorrow!
- Hopefully well on the way with Lab 2!
  - Some problem with schematics (which is fixed?)
- Start Reading Chapter 5
  - No class on Monday (Holiday)
- Next Time: Quick review of single-cycle processor
  - Also: First homework quiz next Wednesday
MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsigned</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Hi = $2 mod $3</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1000</td>
<td>01001</td>
</tr>
</tbody>
</table>

- m bits x n bits = m+n bit product

- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)

- 4 versions of multiply hardware & algorithm:
  - successive refinement

Unsigned Combinational Multiplier

- Stage i accumulates $A * 2^i$ if $B_i == 1$

How does it work?

- At each stage shift A left ($\times 2$)
- Use next bit of B to determine whether to add in shifted multiplicand
- Accumulate 2n bit partial product at each stage
Carry Save addition of 4 integers
° Adding:
  \[A_2A_1A_0 + B_2B_1B_0 + C_2C_1C_0 + D_2D_1D_0\]
  \[= S_4S_3S_2S_1S_0\]

° Add Columns first, then rows!
° Full Adder = 3 2 element
° Can be used to reduce critical path of multiply
° Example: 53 bit multiply (for floating point):
  • At least 53 levels with naïve technique
  • Only 9 with Carry save addition!

Unisigned shift-add multiplier (version 1)
° 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

Multiply Algorithm Version 1

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.

Observations on Multiply Version 1
° 1 clock per cycle => ≈ 100 clocks per multiply
  • Ratio of multiply to add 5:1 to 100:1
° 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
° 0’s inserted in right of multiplicand as shifted
  => least significant bits of product never changed once formed
° Instead of shifting multiplicand to left, shift product to right?
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg

How to think of this?

Remember original combinational multiplier:

Multiply Algorithm Version 2

1. Test Multiplier
   - Multiplier0 = 1
   - Multiplier0 = 0

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions
Done

Simply warp to let product move right...

- Multiplicand stay's still and product moves right
Still more wasted space!

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

   1a. Add multiplicand to the left half of product & place the result in the left half of Product register

   - Product
   - Multiplier
   - Multiplicand

   0000 0000 0011 0010
   1:  0010 0011 0010
   2:  0001 0001 0010
   3:  0011 0000 0010
   4:  0001 0001 0010
   5:  0011 0000 0010
   6:  0001 0001 0010
   7:  0011 0000 0010
   8:  0001 0001 0010
   9:  0011 0000 0010
   10: 0001 0001 0010
   11: 0011 0000 0010
   12: 0001 0001 0010
   13: 0011 0000 0010
   14: 0001 0001 0010
   15: 0011 0000 0010
   16: 0001 0001 0010
   17: 0011 0000 0010
   18: 0001 0001 0010
   19: 0011 0000 0010
   20: 0001 0001 0010
   21: 0011 0000 0010
   22: 0001 0001 0010
   23: 0011 0000 0010
   24: 0001 0001 0010
   25: 0011 0000 0010
   26: 0001 0001 0010
   27: 0011 0000 0010
   28: 0001 0001 0010
   29: 0011 0000 0010
   30: 0001 0001 0010
   31: 0011 0000 0010
   32: 0001 0001 0010

2. Shift the Product register right 1 bit.

   3. Shift the Multiplier register right 1 bit.

   32nd repetition?

   No: < 32 repetitions
   Yes: 32 repetitions

Done

Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
- => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)

Multiply Algorithm Version 3

1. Test Product0
   - Product0 = 1
   - Product0 = 0

   1a. Add multiplicand to the left half of product & place the result in the left half of Product register

   - Product
   - Multiplicand

   0000 0110 0000 0010
   1:  0010 0011 0010
   2:  0001 0001 0010
   3:  0011 0000 0010
   4:  0001 0001 0010
   5:  0011 0000 0010
   6:  0001 0001 0010
   7:  0011 0000 0010
   8:  0001 0001 0010
   9:  0011 0000 0010
   10: 0001 0001 0010
   11: 0011 0000 0010
   12: 0001 0001 0010
   13: 0011 0000 0010
   14: 0001 0001 0010
   15: 0011 0000 0010
   16: 0001 0001 0010
   17: 0011 0000 0010
   18: 0001 0001 0010
   19: 0011 0000 0010
   20: 0001 0001 0010
   21: 0011 0000 0010
   22: 0001 0001 0010
   23: 0011 0000 0010
   24: 0001 0001 0010
   25: 0011 0000 0010
   26: 0001 0001 0010
   27: 0011 0000 0010
   28: 0001 0001 0010
   29: 0011 0000 0010
   30: 0001 0001 0010
   31: 0011 0000 0010
   32: 0001 0001 0010

2. Shift the Product register right 1 bit.

32nd repetition?

No: < 32 repetitions
Yes: 32 repetitions

Done
Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- How can you make it faster?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - apply definition of 2’s complement
    - need to sign-extend partial products and subtract at the end
  - Booth’s Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - can handle multiple bits at a time

Motivation for Booth’s Algorithm

- Example 2 x 6 = 0010 x 0110:
  \[
  \begin{array}{c}
  0010 \\
  \times \ 0110 \\
  + \ 0000 \\
  + \ 0010 \\
  + \ 0010 \\
  + \ 0000 \\
  \hline
  00001100
  \end{array}
  \]
  - shift (0 in multiplier)
  - add (1 in multiplier)
  - shift (0 in multiplier)

- ALU with add or subtract gets same result in more than one way:
  \[
  6 = -2 + 8
  \]
  \[
  0110 = -00010 + 01000 = 11110 + 01000
  \]
  - For example
  \[
  \begin{array}{c}
  0010 \\
  \times \ 0110 \\
  - \ 0010 \\
  . \ 0000 \ \text{shift (mid string of 1s)} \\
  . \ + \ 0010 \ \text{add (prior step had last 1)}
  \hline
  00001100
  \end{array}
  \]

Booth’s Algorithm

- end of run
  - middle of run
  - beginning of run

Booths Example (2 x 7)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplier</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 0111 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a.</td>
<td>P = P - m</td>
<td>1110</td>
<td>- 0000</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0011 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>2.</td>
<td>0010</td>
<td>1111 1001 1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>3.</td>
<td>0010</td>
<td>1111 1100 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td>4a.</td>
<td>0010</td>
<td>+ 0010</td>
<td>0001 1100 1</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>0000 0111 0</td>
<td>done</td>
</tr>
</tbody>
</table>
Booth Example (2 x -3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Multiplicand</th>
<th>Product</th>
<th>next?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. initial value</td>
<td>0010</td>
<td>0000 1101 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. P = P - m</td>
<td>1110</td>
<td>+ 1110</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b.</td>
<td>0010</td>
<td>1111 0110 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 0010</td>
<td></td>
</tr>
<tr>
<td>2a.</td>
<td></td>
<td>0001 0110 1</td>
<td>shift P</td>
</tr>
<tr>
<td>2b.</td>
<td>0010</td>
<td>0000 1011 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ 1110</td>
<td></td>
</tr>
<tr>
<td>3a.</td>
<td>0010</td>
<td>1110 1011 0</td>
<td>shift</td>
</tr>
<tr>
<td>3b.</td>
<td>0010</td>
<td>1111 0101 1</td>
<td>11 -&gt; nop</td>
</tr>
<tr>
<td>4a.</td>
<td>0010</td>
<td>1111 0101 1</td>
<td>shift</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>1111 1010 1</td>
<td>done</td>
</tr>
</tbody>
</table>

Radix-4 Modified Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current Bits</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Recode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>Middle of zeros</td>
<td>00 00 00</td>
<td>00 00 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>Single one</td>
<td>00 00 00</td>
<td>01 00 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>00 01 11</td>
<td>10 00 -2</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>00 01 11</td>
<td>11 00 -1</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>Ends run of 1s</td>
<td>00 00 11</td>
<td>11 00 1</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>Ends run of 1s</td>
<td>00 01 11</td>
<td>11 00 2</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>Isolated 0</td>
<td>00 11 10</td>
<td>11 00 -1</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>Middle of run</td>
<td>00 11 11</td>
<td>11 00 0</td>
</tr>
</tbody>
</table>

* Same insight as one-bit Booth’s, simply adjust for alignment of 2 bits. * Allows multiplication 2-bits at a time.

DIVIDE HARDWARE Version 1

° 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder

=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement
Divide Algorithm Version 1

Start: Place Dividend in Remainder

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.
2a. Shift the Quotient register to the left setting the new rightmost bit to 1.
2b. Restore the original value by adding the Divisor register to the Remainder register, & place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.
3. Shift the Divisor register right 1 bit.

Remainder ≥ 0

Test

Remainder < 0

2/11/03

©UCB Spring 2004 CS152 / Kubiatowicz

Lec6.45

Divide Algorithm I example (7 / 2)

Remainder Quotient Divisor

0000 0111 0000 0010 0000
1: 1110 0111 0000 0010 0000
2: 0000 0111 0000 0010 0000
3: 0000 0111 0000 0010 0000
1: 1111 0111 0000 0010 0000
2: 0000 0111 0000 0010 0000
3: 0000 0111 0000 0010 0000
1: 0000 0011 0000 0010 0000
2: 0000 0011 0000 0010 0000
3: 0000 0011 0000 0010 0000
1: 0000 0001 0000 0001 0000
2: 0000 0001 0000 0001 0000
3: 0000 0001 0000 0001 0000
1: 0000 0001 0000 0001 0000
2: 0000 0001 0000 0001 0000
3: 0000 0001 0000 0001 0000

Answer: Quotient = 3
Remainder = 1

Divide: Paper & Pencil

0 1010 Quotient

Divisor 0001 00001010 Dividend

00001

– 0001

0000

0001

– 0001

0 00 Remainder (or Modulo result)

No way to get a 1 in leading digit!
(this is an overflow, i.e quotient would have n+1 bits)
switch order to shift first and then subtract, can save 1 iteration

Observations on Divide Version 1

° 1/2 bits in divisor always 0
    => 1/2 of 64-bit adder is wasted
    => 1/2 of divisor is wasted

° Instead of shifting divisor to right, shift remainder to left?
### Divide Algorithm I example: wasted space

<table>
<thead>
<tr>
<th>Remainder</th>
<th>Quotient</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>1: 1110 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>2: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>3: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>1: 1111 1111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>2: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>3: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>1: 1111 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>2: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>3: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>1: 1111 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>2: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
<tr>
<td>3: 0000 0111</td>
<td>0000 0010</td>
<td>0000</td>
</tr>
</tbody>
</table>

### Divide HARDWARE Version 2

- **32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg**

![Diagram](image)

### Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half.
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register.
DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

<table>
<thead>
<tr>
<th>Control</th>
<th>Shift Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit ALU</td>
<td>“HI”</td>
</tr>
<tr>
<td>Remainder</td>
<td>“LO”</td>
</tr>
<tr>
<td>Divisor</td>
<td>32 bits</td>
</tr>
<tr>
<td>64 bits</td>
<td>Write</td>
</tr>
</tbody>
</table>

DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

Remainder = \[ \text{Dividend} - \text{Divisor} \times \text{Quotient} \]

Remainder < 0

Remainder ≥ 0

3a. Shift the Remainder register to the left setting the new rightmost bit to 1.

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register, & place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.

4. Test Remainder

Remainder < 0

Remainder ≥ 0

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

5. Done. Shift left half of Remainder right 1 bit

MIPS logical instructions

- and $1,$2,$3 $1 = $2 \& $3 3 reg. operands; Logical AND
- or $1,$2,$3 $1 = $2 | $3 3 reg. operands; Logical OR
- xor $1,$2,$3 $1 = $2 \oplus $3 3 reg. operands; Logical XOR
- nor $1,$2,$3 $1 = \sim($2 |$3) 3 reg. operands; Logical NOR
- and immediate andi $1,$2,10 $1 = $2 \& 10 Logical AND reg, constant
- or immediate ori $1,$2,10 $1 = $2 | 10 Logical OR reg, constant
- xor immediate xori $1, $2,10 $1 = \sim$2 \&~10 Logical XOR reg, constant
- shift left logical sll $1,$2,10 $1 = $2 << 10 Shift left by constant
- shift right logical srl $1,$2,10 $1 = $2 >> 10 Shift right by constant
- shift right arith. sra $1,$2,10 $1 = $2 >> 10 Shift right (sign extend)
- shift left logical sllv $1,$2,$3 $1 = $2 << 3 Shift left by variable
- shift right logical srlv $1,$2,$3 $1 = $2 >> 3 Shift right by variable
- shift right arith. srav $1,$2,$3 $1 = $2 >> 3 Shift right arith. by variable

Remainder < 0

Remainder ≥ 0

nth repetition?

No: < n repetitions

Yes: n repetitions (n = 4 here)

5. Done. Shift left half of Remainder right 1 bit

MIPS logical instructions

- and $1,$2,$3 $1 = $2 \& $3 3 reg. operands; Logical AND
- or $1,$2,$3 $1 = $2 | $3 3 reg. operands; Logical OR
- xor $1,$2,$3 $1 = $2 \oplus $3 3 reg. operands; Logical XOR
- nor $1,$2,$3 $1 = \sim($2 |$3) 3 reg. operands; Logical NOR
- and immediate andi $1,$2,10 $1 = $2 \& 10 Logical AND reg, constant
- or immediate ori $1,$2,10 $1 = $2 | 10 Logical OR reg, constant
- xor immediate xori $1, $2,10 $1 = \sim$2 \&~10 Logical XOR reg, constant
- shift left logical sll $1,$2,10 $1 = $2 << 10 Shift left by constant
- shift right logical srl $1,$2,10 $1 = $2 >> 10 Shift right by constant
- shift right arith. sra $1,$2,10 $1 = $2 >> 10 Shift right (sign extend)
- shift left logical sllv $1,$2,$3 $1 = $2 << 3 Shift left by variable
- shift right logical srlv $1,$2,$3 $1 = $2 >> 3 Shift right by variable
- shift right arith. srav $1,$2,$3 $1 = $2 >> 3 Shift right arith. by variable
Shifters

Two kinds:

- **logical**: value shifted in is always "0"
  
  "0"\[\text{msb}\] \[\text{lsb}\] "0"

- **arithmetic**: on right shifts, sign extend
  
  msb \[\text{lsb}\] "0"

Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Combinational Shifter from MUXes

- **Basic Building Block**
  - sel \[\text{A}\] \[\text{B}\] \[\text{D}\]

- **8-bit right shifter**
  - \[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0\]

  - What comes in the MSBs?
  - How many levels for 32-bit shifter?
  - What if we use 4-1 Muxes?

General Shift Right Scheme using 16 bit example

- S0 \((0, 1)\)
- S1 \((0, 2)\)
- S2 \((0, 4)\)
- S3 \((0, 8)\)

If added Right-to-left connections could support Rotate (not in MIPS but found in ISAs)

Funnel Shifter

Instead Extract 32 bits of 64.

- Shift A by i bits
  - \(\text{sa}=\text{shift right amount}\)

- Logical: \(Y = 0, X = A, sa=i\)

- Arithmetic? \(Y = _, X = _, sa=_\)

- Rotate? \(Y = _, X = _, sa=_\)

- Left shifts? \(Y = _, X = _, sa=_\)
Barrel Shifter

Technology-dependent solutions: transistor per switch

SR3 SR2 SR1 SR0

D3

D2

D1

D0

A6

A5

A4

A3 A2 A1 A0

Summary

- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth’s algorithm to handle signed multiplies
  - There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)

- Booth Encoding: introducing multiple representations
  - Numbers can be encoded in different ways  
  - SRT Divide (for instance)
  - Handles multiplication of negative numbers transparently

- Divide can use same hardware as multiply: Hi & Lo registers in MIPS

- Shifter: success refinement 1/bit at a time shift register to barrel shifter

To Get More Information

- Chapter 4 of your text book: