Today’s Outline

- Review of Last lecture
- Intro to VHDL
- Administrative Issues
- on-line lab notebook
- Designing a Multiplier
- Booth’s algorithm
- Shifters

Review: ALU Design

- Bit-slice plus extra on the two ends
- Overflow means number too large for the representation
- Carry-look ahead and other adder tricks

Review: Elements of the Design Process

- Divide and Conquer (e.g., ALU)
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)
- Generate and Test (e.g., ALU)
  - Given a collection of building blocks, look for ways of putting them together that meets requirement
- Successive Refinement (e.g., multiplier, divider)
  - Solve “most” of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.
- Formulate High-Level Alternatives (e.g., shifter)
  - Articulate many strategies to “keep in mind” while pursuing any one approach.
- Work on the Things you Know How to Do
  - The unknown will become “obvious” as you make progress.
Review: Summary of the Design Process

Hierarchical Design to manage complexity

Top Down vs. Bottom Up vs. Successive Refinement

Importance of Design Representations:
- Block Diagrams
- Decomposition into Bit Slices
- Truth Tables, K-Maps
- Circuit Diagrams
- Other Descriptions: state diagrams, timing diagrams, reg xfer, ...

Optimization Criteria:
- Gate Count
- [Package Count]
- Logic Levels
- Fan-in/Fan-out
- Pin Out
- Cost
- Design time
- Area
- Delay
- Power
- Top
- down
- bottom
- up

Why should you keep an design notebook?

- Keep track of the design decisions and the reasons behind them
  - Otherwise, it will be hard to debug and/or refine the design
  - Write it down so that you can remember in long project: 2 weeks ->2 yrs
  - Others can review notebook to see what happened
- Record insights you have on certain aspect of the design as they come up
- Record of the different design & debug experiments
  - Memory can fail when very tired
- Industry practice: learn from others mistakes

Why do we keep it on-line?

- You need to force yourself to take notes
  - Open a window and leave an editor running while you work
    1) Acts as reminder to take notes
    2) Makes it easy to take notes
  - 1) + 2) => will actually do it
- Take advantage of the window system’s "cut and paste" features
- It is much easier to read your typing than your writing
- Also, paper log books have problems
  - Limited capacity => end up with many books
  - May not have right book with you at times vs. networked screens
  - Can use computer to search files/index files to find what looking for

How should you do it?

- Keep it simple
  - DON’T make it so elaborate that you won’t use (fonts, layout, ...)
- Separate the entries by dates
  - Type "date" command in another window and cut&paste
- Start day with problems going to work on today
- Record output of simulation into log with cut&paste; add date
  - May help sort out which version of simulation did what
- Record key email with cut&paste
- Record of what works & doesn’t help team decide what went wrong after you left
- Index: write a one-line summary of what you did at end of each day
On-line Notebook Example

Refer to the handout:

“Example of On-Line Log Book” on cs152 home page (handouts section)
I verified the bug. Here's a viewsim of the bug as it appeared...

\[ \text{equal should be 0 instead of 1} \]

```
SIM>stepsize 10ns
SIM>v a_in A[31:0]
SIM>v b_in B[31:0]
SIM>w a_in b_in equal
SIM>a a_in ffffffff
SIM>a b_in fffffff7
SIM>sim
```

```
time = 10.0ns  A_IN=FFFFFFFF  B_IN=FFFFFFF7  EQUAL=1
```

Ah, I've discovered the bug. I mislabeled the 4th net in the comp32 schematic.

I corrected the mistake and re-checked all the other labels, just in case.

I re-ran the old diagnostic test file and tested it against the bug Bart found. It seems to be working fine. Hopefully there aren't any more bugs:

---

On second inspection of the whole layout, I think I can remove one level of gates in the design and make it go faster. But who cares! The comparator is not in the critical path right now. The delay through the ALU is dominating the critical path, so unless the ALU gets a lot faster, we can live with a less than optimal comparator.

I e-mailed the group that the bug has been fixed.

Mon Sep 11 14:03:41 PDT 1995

```
---
```

• Perhaps later critical path changes:
  what was idea to make comparator faster? Check log book!

---

**Added benefit: cool post-design statistics**

Sample graph from the Alewife project:

- Initial set of "silly" bugs
- John Kubiatowicz at ISCA '92 (Debugging grade in a hat)
- Directed-vertex testing
- Change in target technology
- Use of test "tasters" (Really nifty bugs)

**For the Communications and Memory Management Unit (CMMU)**

- These statistics came from on-line record of bugs

---

**Representation Languages**

**Hardware Representation Languages:**

- Block Diagrams: FUs, Registers, & Dataflows
- Register Transfer Diagrams: Choice of busses to connect FUs, Regs
- Flowcharts
- State Diagrams

**Fifth Representation "Language": Hardware Description Languages**

- VHDL
- Verilog

**Descriptions in these languages can be used as input to**

- simulation systems
- synthesis systems

"To Design is to Represent"
Simulation Before Construction

- "Physical Breadboarding"
  - discrete components/lower scale integration preceeds actual construction of prototype
  - verify initial design concept
- No longer possible as designs reach higher levels of integration!
- Simulation Before Construction
  - high level constructs implies faster to construct
  - play "what if" more easily
  - limited performance accuracy, however

Levels of Description

- Architectural Simulation
  - models programmer’s view at a high level; written in your favorite programming language
- Functional/Behavioral
  - more detailed model, like the block diagram view
- Register Transfer
  - commitment to datapath FUs, registers, busses; register xfer operations are clock phase accurate
- Logic
  - model is in terms of logic gates; higher level MSI functions described in terms of these
- Circuit
  - electrical behavior; accurate waveforms

Schematic capture + logic simulation package like Powerview
Special languages + simulation systems for describing the inherent parallel activity in hardware

VHDL (VHSIC Hardware Description Language)

- Goals:
  - Support design, documentation, and simulation of hardware
  - Digital system level to gate level
  - "Technology Insertion"

- Concepts:
  - Design entity
  - Time-based execution model.

Interface

- Externally Visible Characteristics
  - Ports: channels of communication
    - (inputs, outputs, clocks, control)
  - Generic Parameters: define class of components
    - (timing characteristics, size, fan-out)
    --- determined where instantiated or by default
- Internally Visible Characteristics
  - Declarations:
  - Assertions: constraints on all alternative bodies
    - (i.e., implementations)

Design Entity == Hardware Component
Architecture (Body) == Internal Behavior or Structure
VHDL Example: nand gate

ENTITY nand is
  PORT (a,b: IN VLBIT; y: OUT VLBIT)
END nand

ARCHITECTURE behavioral OF nand is
BEGIN
  y <= a NAND b;
END behavioral;

- Entity describes interface
- Architecture give behavior, i.e., function
- `y` is a signal, not a variable
  - it changes when ever the inputs change
  - drive a signal
  - NAND process is in a infinite loop
- VLBit is 0, 1, X or Z

Modeling Delays

ENTITY nand is
  PORT (a,b: IN VLBIT; y: OUT VLBIT)
END nand

ARCHITECTURE behavioral OF nand is
BEGIN
  y <= a NAND b after 1 ns;
END behavioral;

- Model temporal, as well as functional behavior, with delays in signal statements; Time is one difference from programming languages
- `y` changes 1 ns after a or b changes
- This fixed delay is inflexible
  - hard to reflect changes in technology

Generic Parameters

ENTITY nand is
  GENERIC (delay: TIME := 1ns);
  PORT (a,b: IN VLBIT; y: OUT VLBIT)
END nand

ARCHITECTURE behavioral OF nand is
BEGIN
  y <= a NAND b AFTER delay;
END behavioral;

- Generic parameters provide default values
  - may be overridden on each instance
  - attach value to symbol as attribute
- Separate functional and temporal models
- How would you describe fix-delay + slope * load model?

Bit-vector data type

ENTITY nand32 is
  PORT (a,b: IN VLBIT_1D (31 downto 0); y: OUT VLBIT_1D (31 downto 0))
END nand32

ARCHITECTURE behavioral OF nand32 is
BEGIN
  y <= a NAND b;
END behavioral;

- VLBIT_1D (31 downto 0) is equivalent to powerview 32-bit bus
- Can convert it to a 32 bit integer
Arithmetic Operations

ENTITY add32 is
PORT (a,b: IN VLBIT_1D (31 downto 0);
y: OUT VLBIT_1D (31 downto 0)
END add32

ARCHITECTURE behavioral OF add32 is
BEGIN
y <= addum(a, b) ;
END behavioral;

° addum (see VHDL ref. appendix C) adds two n-bit vectors to produce an n+1 bit vector
  • except when n = 32!

Control Constructs

entity MUX32X2 is
generic (output_delay : TIME := 4 ns);
port(A,B: in vlbit_1d(31 downto 0);
DOUT: out vlbit_1d(31 downto 0);
SEL: in vlbit);
end MUX32X2;

architecture behavior of MUX32X2 is
BEGIN
mux32x2_process: process(A, B, SEL)
begin
if (vlb2int(SEL) = 0) then
DOUT <= A after output_delay;
else
DOUT <= B after output_delay;
end if;
end if;
end process;
end behavior;

° Process fires whenever is “sensitivity list” changes
° Evaluates the body sequentially
° VHDL provide case statements as well

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>add $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. unsign.</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, Hi = $2 mod $3</td>
<td>quotient, remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</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>

Administrative Matters

° Remember that first homework due next Tuesday
  • First homework quiz at BEGINNING of class.
  • No late homework (No exceptions)
° First Lab due next Wednesday at midnight via submit program
° On-line lab notebook is such a good idea, its required! (starting with Lab 3)
° Reading Chapter 4 now
**MULTIPLY (unsigned)**

- Paper and pencil example (unsigned):
  - **Multiplicand**: 1000
  - **Multiplier**: 1001
    - **Product**: 01001000

- 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 $A \times 2^i$ if $B_i = 1$

---

**How does it work?**

- at each stage shift A left (x 2)
- use next bit of B to determine whether to add in shifted multiplicand
- accumulate 2n bit partial product at each stage

---

**Unsigned shift-add multiplier (version 1)**

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

---

**Multiplier = datapath + control**
Multiply Algorithm Version 1

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

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

2. Shift the Multiplicand register left 1 bit.
   - Multiplier 0000 0010
   - Multiplicand 0000 0110

3. Shift the Multiplier register right 1 bit.
   - Product 0000 0000
   - Multiplier 0000 0011
   - Multiplicand 0000 0110

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

Done

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 left 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.**
   - No: < 32 repetitions
   - Yes: 32 repetitions

3. **Shift the Multiplier register right 1 bit.**
   - 32nd repetition?

**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

![Diagram of multiply hardware](image)

### Multiply Algorithm Version 3

1. **Test Product0**
   - **Product0 = 1**
     1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   - **Product0 = 0**

2. Shift the Product register right 1 bit.

3. **32nd repetition?**
   - No: < 32 repetitions
   - Yes: 32 repetitions

### 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?**
- What about signed multiplication?
  - 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:**
  
  \[
  \begin{array}{c}
  \text{0010} \\
  \times \ \text{0110} \\
  \end{array}
  \]
  
  \[
  \begin{array}{c}
  + \ \text{0000} \quad \text{shift (0 in multiplier)} \\
  + \ \text{0010} \quad \text{add (1 in multiplier)} \\
  + \ \text{0000} \quad \text{shift (0 in multiplier)} \\
  \end{array}
  \]
  
  \[
  \text{00001100}
  \]

- **ALU with add or subtract gets same result in more than one way:**
  
  \[
  \begin{array}{c}
  \text{6} \quad = -2 + 8 \\
  \text{0110} \quad = -\text{00010} + \text{01000} = \text{11110} + \text{01000}
  \end{array}
  \]

- **For example**
  
  \[
  \begin{array}{c}
  \text{0010} \\
  \times \ \text{0110} \\
  \end{array}
  \]
  
  \[
  \begin{array}{c}
  - \ \text{0000} \quad \text{shift (0 in multiplier)} \\
  + \ \text{0010} \quad \text{sub (first 1 in multiplier)} \\
  + \ \text{0000} \quad \text{shift (mid string of 1s)} \\
  + \ \text{0010} \quad \text{add (prior step had last 1)}
  \end{array}
  \]
  
  \[
  \text{00001100}
  \]
Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>0001111000</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>0001111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>001111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>0001111000</td>
<td>none</td>
</tr>
</tbody>
</table>

Originally for Speed (when shift was faster than add)

° Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

-1
+ 10000
01111

Booths Example (2 x 7)

<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 0111 0</td>
<td>10 -&gt; sub</td>
</tr>
<tr>
<td>1a. P = P - m</td>
<td>1110</td>
<td>+ 1110 0111 0</td>
<td>shift P (sign ext)</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 0001 1100 1</td>
<td>shift</td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>0000 1110 0</td>
<td>done</td>
</tr>
</tbody>
</table>

Booths 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 1110 1101 0</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>2a.</td>
<td>0001 0110 1</td>
<td>shift P</td>
<td></td>
</tr>
<tr>
<td>2b.</td>
<td>0010</td>
<td>0000 1011 0</td>
<td>10 -&gt; sub</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>1111 0101 1</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>4b.</td>
<td>0010</td>
<td>1111 1010 1</td>
<td>done</td>
</tr>
</tbody>
</table>

MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>* and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
</tr>
<tr>
<td>* or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
</tr>
<tr>
<td>* xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
</tr>
<tr>
<td>* nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~($2</td>
</tr>
<tr>
<td>* and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
</tr>
<tr>
<td>* or immediate</td>
<td>or $1,$2,10</td>
<td>$1 = $2</td>
</tr>
<tr>
<td>* xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = ~$2 &amp; ~10</td>
</tr>
<tr>
<td>* shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
</tr>
<tr>
<td>* shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
</tr>
<tr>
<td>* shift right arithm. sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>* shift left logical</td>
<td>sllv $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 3</td>
</tr>
<tr>
<td>* shift right logical</td>
<td>srli $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 3</td>
</tr>
<tr>
<td>* shift right arithm. sraw $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; 3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
Shifters

Two kinds:

- **logical**—value shifted in is always "0"
  - "0"  
  - msb  
  - lsb  
  - "0"

- **arithmetic**—on right shifts, sign extend
  - msb  
  - 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
  - A  
  - B  
  - sel  
  - D

- 8-bit right shifter
  - A_7  
  - A_6  
  - A_5  
  - A_4  
  - A_3  
  - A_2  
  - A_1  
  - A_0  
  - S_2  
  - S_1  
  - S_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

- S_0  
  - (0, 1)
- S_1  
  - (0, 2)
- S_2  
  - (0, 4)
- S_3  
  - (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.
  - Y  
  - X

- Shift A by i bits (sa= 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

```
<table>
<thead>
<tr>
<th></th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Divide: Paper & Pencil**

1001 Quotient

```
<table>
<thead>
<tr>
<th>Divisor 1000</th>
<th>1001010</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1000</td>
<td>10</td>
</tr>
<tr>
<td>101</td>
<td>1010</td>
</tr>
<tr>
<td>-1000</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Remainder (or Modulo result)

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

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient \times Divisor + Remainder

\[\Rightarrow |\text{Dividend}| = |\text{Quotient}| + |\text{Divisor}|\]

3 versions of divide, successive refinement

**DIVIDE HARDWARE Version 1**

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

- Start: Place Dividend in Remainder

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

2. **Shift the Quotient register to the left setting the new rightmost bit to 1.**

3. **Shift the Divisor register right 1 bit.**

**Divide Algorithm Version 1**

- Takes \(n+1\) steps for \(n\)-bit Quotient & Rem.

Remainder | Quotient | Divisor |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0111</td>
<td>0000</td>
</tr>
</tbody>
</table>

\(0\) repetitions? \(\Rightarrow \)

- No: < \(n+1\) repetitions
- Yes: \(n+1\) repetitions (\(n = 4\) here)

Done
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?

° 1st step cannot produce a 1 in quotient bit  
  (otherwise too big)  
  => switch order to shift first and then subtract,  
  can save 1 iteration

DIVIDE HARDWARE Version 2

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

Divide Algorithm Version 2

1. Shift the Remainder register left 1 bit  

2. Subtract the Divisor register from the  
left half of the Remainder register, & place the  
result in the left half of the Remainder register.

3a. Shift the Quotient 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 Quotient register to the left, setting the new least significant bit to 0.

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

Divide Algorithm Version 3

1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
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.

Remainder < 0
- Test Remainder
- No: < n repetitions
- Yes: n repetitions (n = 4 here)

Done. Shift left half of Remainder right 1 bit

Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., \(-7 + 2 = -3\), remainder = \(-1\)
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (“called saturation”)

Summary

- Intro to VHDL
  - a language to describe hardware
    - entity = symbol, architecture ~ schematic, signals = wires
  - behavior can be higher level
    - \(x \Leftarrow \text{boolean_expression}(A,B,C,D)\)
  - Has time as concept
  - Can activate when inputs change, not specifically invoked
  - Inherently parallel

- 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)

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

- What’s Missing from MIPS is Divide & Floating Point Arithmetic: Next time the Pentium Bug
To Get More Information

- Chapter 4 of your textbook:
