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]
- Area
- Logic Levels
- Fan-in/Fan-out
- Delay
- Power
- Pin Out
- Cost
- Design time

Representation Languages

Hardware Representation Languages:
- Block Diagrams: FUs, Registers, & Dataflows
- Register Transfer Diagrams: Choice of busses to connect FUs, Regs
- Flowcharts
- State Diagrams
  - Two different ways to describe sequencing & microoperations

Fifth Representation "Language": Hardware Description Languages
- E.G., ISP', VHDL, Verilog
  - hw modules described like programs
  - with i/o ports, internal state, & parallel execution of assignment statements

Descriptions in these languages can be used as input to
- simulation systems
- "software breadboard"
- synthesis systems
  - generate hw from high level description

"To Design is to Represent"

Review: Cost/Price and Online Notebook

- Cost and Price
  - Die size determines chip cost: cost = die size\((\alpha + 1)\)
  - Cost v. Price: business model of company, pay for engineers
  - R&D must return $8 to $14 for every $1 invester

- On-line Design Notebook
  - Open a window and keep an editor running while you work; cut&paste
  - Refer to the handout as an example
  - Former CS 152 students (and TAs) say they use on-line notebook for programming as well as hardware design; one of most valuable skills

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 electrical behavior; accurate waveforms.

**Circuit** 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** == External Characteristics

**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 an infinite loop

- **VLBit is 0, 1, X or Z**

---

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)
Modeling Delays

- **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

- **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

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

Arithmetic Operations

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

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

```vhdl
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 process;
end behavior;
```

---

**Administrative Matters**

- Broken-spim was online Saturday
- Final class list is online.
- On-line lab notebook is such a good idea, it’s required! (starting with Lab 3)
- Reading Chapter 4 now

---

**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. 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, Hi = $2 mod $3</td>
<td>Quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unssigned 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>

---

**MULTIPLY (unsigned)**

- Paper and pencil example (unsigned):

<p>| Multiplicand | 1000 |</p>
<table>
<thead>
<tr>
<th>Multiplier</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>01001000</td>
</tr>
</tbody>
</table>

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

- Binary makes it easy:
  - $0 \Rightarrow \text{place } 0$ (0 x multiplicand)
  - $1 \Rightarrow \text{place a copy }$ (1 x multiplicand)

- 4 versions of multiply hardware & algorithm:
  - successive refinement
Unsigned Combinational Multiplier

- Stage $i$ accumulates $A \times 2^i$ if $B_i = 1$
- Q: How much hardware for 32-bit multiplier? Critical path?

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

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 $Multiplier_0$
   - $Multiplier_0 = 1$
   - $Multiplier_0 = 0$
2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.
4. Product = $Multipler \times Multiplier_0$
   - $0000\ 0000\ 0011$ Multiplier
   - $0000\ 0010\ 0000\ 0100$ Multiplier
   - $0000\ 0110\ 0000\ 1000$ Multiplier
   - $0000\ 0110$ Multiplier

Start

Control

Write

Multiplier $= \text{datapath} + \text{control}$
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:

Simply warp to let product move right...

* Multiplicand stay’s still and product moves right
Multiply Algorithm Version 2

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

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

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 0000 0011 0010
2: 0001 0000 0011 0010
3: 0001 1000 0001 0010
1: 0011 0000 0001 0010
2: 0001 1000 0000 0010
3: 0000 1100 0000 0010
1: 0000 1100 0000 0010
2: 0000 0110 0000 0010
3: 0000 0110 0000 0010

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**
   - \( \text{Product0} = 1 \)
   - \( \text{Product0} = 0 \)

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

2. **Shift the Product register right 1 bit.**

3. **32nd repetition?**
   - Yes: 32 repetitions
   - No: < 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: \( 2 \times 6 = 0010 \times 0110 \):
  - \( \begin{array}{c}
    0010 \\
    \times 0110 \\
    + 0000 \\
    + 0010 \\
    + 0000 \\
  \end{array} \)
  - \( \begin{array}{c}
    \text{shift (0 in multiplier)} \\
    \text{add (1 in multiplier)} \\
    \text{shift (0 in multiplier)}
  \end{array} \)
  - \( \begin{array}{c}
    00001100
  \end{array} \)
- ALU with add or subtract gets same result in more than one way:
  - \( 6 = -2 + 8 \)
  - \( 0110 = -00010 + 01000 = 11110 + 01000 \)
- For example:
  - \( \begin{array}{c}
    0010 \\
    \times 0110 \\
    + 0000 \quad \text{shift (0 in multiplier)} \\
    + 0010 \quad \text{shift (mid string of 1s)} \\
    + 0010 \quad \text{add (prior step had last 1)}
  \end{array} \)
  - \( \begin{array}{c}
    00001100
  \end{array} \)

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>\text{sub}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>( 0001111000 )</td>
<td>\text{none}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>( 001111000 )</td>
<td>\text{add}</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>( 001111000 )</td>
<td>\text{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
  - \( \begin{array}{c}
    -1 \\
    \text{+ } 10000
  \end{array} \)
  - \( \begin{array}{c}
    01111
  \end{array} \)
### 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>0001 1100 1</td>
<td>01 -&gt; add</td>
</tr>
<tr>
<td>4a.</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 110</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</td>
<td>0110 10 11</td>
<td>shift</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 1</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>

### MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>3 reg. operands; Logical OR</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ¬($2 ⊕ $3)</td>
<td>3 reg. operands; Logical NOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg. constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>Logical OR reg. constant</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = ¬($2 &amp; $3)</td>
<td>Logical XOR reg. constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>sr $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,$2,3</td>
<td>$1 = $2 &lt;&lt; 3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>sr $1,$2,3</td>
<td>$1 = $2 &gt;&gt; 3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>sr $1,$2,3</td>
<td>$1 = $2 &gt;&gt; 3</td>
<td>Shift right by variable</td>
</tr>
</tbody>
</table>

### Shifters

Two kinds:

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

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

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

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

**Basic Building Block**

8-bit right shifter

```
  sel 1 0
  A    B
  D

A7 A6 A5 A4 A3 A2 A1 A0
```

```
  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
  R2 R1 R0 R4 R3 R2 R1 R0
```

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**General Shift Right Scheme using 16 bit example**

**Funnel Shifter**

- Instead Extract 32 bits of 64.

```
  Y
  X
```

Shift Right

\[ Y = 0, \quad X = A, \quad sa = i \]

- Logical
- Arithmetic
- Rotate
- Left shifts

**Barrel Shifter**

Technology-dependent solutions: transistor per switch
### Divide: Paper & Pencil

**Divisor** 1000  
**Dividend** 1001010  
**Quotient** 1001  
**Remainder** 10

See how big a number can be subtracted, creating quotient bit on each step  
- Binary => 1 * divisor or 0 * divisor

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

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

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**

#### 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 Algorithm Version 1

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

#### Divide Algorithm Flowchart

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.

\( n+1 \) repetitions?  

No: \(< n+1 \) repetitions

Yes: \( n+1 \) repetitions (\( n = 4 \) here)

Done
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.

nth repetition? No: < n repetitions

Yes: n repetitions (n = 4 here)

Observations on Divide Version 2

- Eliminate Quotient register by combining with Remaider 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.

Test Remainder

Remainder < 0

Remainder

nth repetition?

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., \(-7 \div 2 = -3\), 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 text book: