Today's Outline

- Review of Last lecture
- on-line lab notebook
- Intro to VHDL
- Administrative Issues
- 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
- Area
- Logic Levels
- Fan-in/Fan-out
- Cost
- Design time
- Delay
- Power
- Package Count
- Pin Out
- 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 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
    - Acts as reminder to take notes
    - Makes it easy to take notes
  - Separate the entries by dates
    - Type “date” command in another window and cut&paste
  - 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 time 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)

1st page of On-line notebook (Index + Wed. 9/6/95)

Index

Wed Sep 6 00:47:28 PDT 1995 - Created the 32-bit comparator component
Thu Sep 7 14:02:21 PDT 1995 - Tested the comparator
Mon Sep 11 12:01:45 PDT 1995 - Investigated bug found by Bart in comp32 and fixed it

Wed Sep 6 00:47:28 PDT 1995

Goal: Layout the schematic for a 32-bit comparator

I've layed out the schematics and made a symbol for the comparator.
I named it comp32. The files are
 ~/wv/proj1/sch/comp32.sch
 ~/wv/proj1/sch/comp32.sym

Wed Sep 6 02:29:22 PDT 1995

Add 1 line index at front of log file at end of each session: date+summary
Start with date, time of day + goal
Make comments during day, summary of work
End with date, time of day (and add 1 line summary at front of file)

2nd page of On-line notebook (Thursday 9/7/95)

-stract

Thu Sep 7 14:02:21 PDT 1995

Goal: Test the comparator component

I've written a command file to test comp32. I've placed it
in ~/wv/proj1/diagnostics/comp32.cmd.

I ran the command file in viewsim and it looks like the comparator
is working fine. I saved the output into a log file called
~/wv/proj1/diagnostics/comp32.log

Notified the rest of the group that the comparator
is done.

Thu Sep 7 16:15:32 PDT 1995

-stract

3rd page of On-line notebook (Monday 9/11/95)

Mon Sep 11 12:01:45 PDT 1995

Goal: Investigate bug discovered in comp32 and hopefully fix it
Bart found a bug in my comparator component. He left the following
email:

-------------------
From bart@simpsons.residence Sun Sep 10 01:47:02 1995
Received: by wayne.manor (NX5.67e/NX3.0S)
        id AA00334; Sun, 10 Sep 95 01:47:01 -0800
Date: Wed, 10 Sep 95 01:47:01 -0800
From: Bart Simpson <bart@simpsons.residence>
To: bruce@wayne.manor, old_man@gokuraku, hojo@sanctuary
Subject: [cs152] bug in comp32
Status: A

Hey Bruce,
I think there's a bug in your comparator.
The comparator seems to think that ffffffff and ffffff7 are equal.
Can you take a look at this? Bart

-------------------
I verified the bug. Here's a viewsim of the bug as it appeared.

```
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\h
SIM>a b_in fffffff7\h
SIM>sim
time = 10.0ns  A_IN=FFFFFFFF\h  B_IN=FFFFFFF7\h  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:

- 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
- 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
- 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

Less Abstract
More Accurate
Slower Simulation

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 characterisitcs, size, fan-out)

- Internally Visible Characteristics
  - Declarations:
  - Assertions: constraints on all alternative bodies
    - (i.e., implementations)
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
- \( \text{VLBit is 0, 1, X or Z} \)

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;
```

- \( \text{VLBIT}_1D \) (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!

MIPS arithmetic instructions

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

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 process;
end behavior;

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

Administrative Matters

° Remember that first homework due next Wednesday
  • First homework quiz at BEGINNING of class.
  • No late homework (No exceptions)
° First Lab due next Wednesday at midnight via submit program
  • No late labs!
  • Mount filesystem as \fileservice\cs152
° 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):

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Product</td>
<td>01001000</td>
</tr>
</tbody>
</table>

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

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

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

---

**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 Combinational Multiplier**

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

- Q: How much hardware for 32 bit multiplier? Critical path?

---

**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 Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

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

2. Shift the Multiplicand register left 1 bit.

3. Shift the Multiplier register right 1 bit.

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

Observations on Multiply Version 1

- 1 clock per cycle => \( \approx 100 \) 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 stays still and product moves right.

Multiply Algorithm Version 2

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   - 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

Observations on Multiply Version 2

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

Still more wasted space!
MULTIPLY HARDWARE Version 3

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

Multiply Algorithm Version 3

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:

\[
\begin{array}{c}
0010 \\
\times 0110 \\
\hline
+ 0000 \\
+ 0010 \\
+ 0000 \\
\hline
00001100
\end{array}
\]

° ALU with add or subtract gets same result in more than one way:

\[
\begin{array}{c}
6 \\
= -2 + 8 \\
0110 = -00010 + 01000 = 11110 + 01000
\end{array}
\]

° For example

\[
\begin{array}{c}
0010 \\
\times 0110 \\
\hline
- 0000 \text{ shift (mid string of 1s)} \\
+ 0010 \text{ add (prior step had last 1)} \\
\hline
00001100
\end{array}
\]
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>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>000111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>00111000</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>000111000</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

Booth’s 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</td>
<td>shift P (sign ext)</td>
</tr>
<tr>
<td>1b. 0010</td>
<td>1111 0111</td>
<td>0</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>2. 0010</td>
<td>1111 1001</td>
<td>1</td>
<td>11 -&gt; nop, shift</td>
</tr>
<tr>
<td>3. 0010</td>
<td>1111 1100</td>
<td>01</td>
<td>-&gt; add</td>
</tr>
<tr>
<td>4a. 0010</td>
<td>+ 0010</td>
<td>0001 1100</td>
<td>1</td>
</tr>
<tr>
<td>4b. 0010</td>
<td>0000 1110</td>
<td>0</td>
<td>done</td>
</tr>
</tbody>
</table>

Booth’s 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. 0010</td>
<td>1111 0110</td>
<td>01</td>
<td>-&gt; add</td>
</tr>
<tr>
<td>2a. 0001</td>
<td>0110</td>
<td>1</td>
<td>shift P</td>
</tr>
<tr>
<td>2b. 0010</td>
<td>0000 1011 0</td>
<td>10 -&gt; sub</td>
<td></td>
</tr>
<tr>
<td>3a. 0010</td>
<td>1110 1011 0</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>3b. 0010</td>
<td>1111 0101</td>
<td>11</td>
<td>-&gt; nop</td>
</tr>
<tr>
<td>4a. 0010</td>
<td>1111 0101</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>4b. 0010</td>
<td>1111 1010</td>
<td>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 ^ $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>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,$3</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srvl $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithmetic.</td>
<td>srav $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; 10</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**

8-bit right shifter

<table>
<thead>
<tr>
<th>sel</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**What comes in the MSBs?**

**How many levels for 32-bit shifter?**

**What if we use 4-1 Muxes?**

Funnel Shifter

Instead Extract 32 bits of 64.

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=_

R
**Barrel Shifter**

Technology-dependent solutions: transistor per switch

![Diagram of Barrel Shifter]

**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:
