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.

Outline of Today’s Lecture

○ Deriving the ALU from the Instruction Set

○ Multiply
### 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>$1 = \text{add}$</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = \text{subtract}$</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = \text{add immediate}$ + constant; exception possible</td>
<td></td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = \text{add unsigned}$</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = \text{subtract unsigned}$</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = \text{add imm. unsign.}$ constant; no exceptions</td>
<td></td>
</tr>
<tr>
<td>multiply</td>
<td>mul $2,$3</td>
<td>Hi, Lo = $2 \times S3$</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 \times S3$</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 \div S3$, Hi = $2 \mod S3$</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td>divide unsigned remainder</td>
<td>divu $2,$3</td>
<td>Lo = $2 \div S3$, Hi = $2 \mod S3$</td>
<td></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>

### 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>$1 = \text{and}$</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = \text{or}$</td>
<td>3 reg. operands; Logical OR</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = \text{xor}$</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = \text{nor}$</td>
<td>3 reg. operands; Logical NOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = \text{and immediate}$</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = \text{or immediate}$</td>
<td>Logical OR reg, constant</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1,$2,10</td>
<td>$1 = \text{xor immediate}$</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = \text{shift left logical}$</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = \text{shift right logical}$</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>sra $1,$2,10</td>
<td>$1 = \text{shift right arith.}$</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slt $1,$2,10</td>
<td>$1 = \text{shift left logical}$</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2,10</td>
<td>$1 = \text{shift right logical}$</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>srav $1,$2,10</td>
<td>$1 = \text{shift right arith.}$</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
Additional MIPS ALU requirements

- **Xor, Nor, Xorl**
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- **Sll, Srl, Sra**
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- **Mult, MultU, Div, DivU**
  => Need 32-bit multiply and divide, signed and unsigned

Add XOR to ALU

- **Expand Multiplexor**

![Diagram of 1-bit Full Adder](image)
Shifters
Three different kinds:

*logical*-- value shifted in is always "0"

\[0\] msb lsb ["0"

*arithmetic*-- on right shifts, sign extend

\[\text{msb lsb}\] ["0"

*rotating*-- shifted out bits are wrapped around (not in MIPS)

\[\text{left right}\]

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

Administrative Matters
MULTIPLY (unsigned)

° Paper and pencil example (unsigned):

| Multiplicand | 1000 |
| Multiplier   | 1001 |

\[
\begin{array}{c}
1000 \\
0000 \\
1000 \\
\end{array}
\]

Product \[01001000\]

° \( m \text{ bits} \times n \text{ bits} = m+n \text{ bit product} \)

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

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

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

```
Multiplier
32 bits
32-bit ALU

Product
64 bits
```

Multiply Algorithm Version 2

- Multiplier Multiplicand
- Product
- Multiplier Multiplicand
- Product

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
What’s going on?

- Multiplicand stays still and product moves right

Multiply Algorithm Version 2

1. Test Multiplier
   - 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.

- Product Multiplier Multiplicand
  - 0000 0000 0011 0010
  - 0010 0000
  - 0001 0000 0001 0010
  - 0011 00 0001 0010
  - 0001 1000 0000 0010
  - 0000 1100 0000 0010
  - 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

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.

2a. No: < 32 repetitions

32nd repetition?

32nd repetition?

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?
° 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:

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

\[
\begin{array}{c}
\text{shift (0 in multiplier)} \\
+ \text{add (1 in multiplier)} \\
+ \text{add (1 in multiplier)} \\
+ \text{shift (0 in multiplier)}
\end{array}
\]

\[
00001100
\]

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

\[
6 = -2 + 8 \quad \text{or} \quad 0110 = -0010 + 1000
\]

° 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. For example

\[
\begin{array}{c}
0010 \\
+ 0110 \\
\hline
0000 \\
- 0010 \quad \text{sub (first 1 in multiplier)} \\
+ 0000 \quad \text{shift (middle of string of 1s)} \\
+ 0010 \quad \text{add (prior step had last 1)}
\end{array}
\]

\[
00001100
\]

**Booth’s Algorithm Insight**

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Beginning of a run of 1s</td>
<td>0001111000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of a run of 1s</td>
<td>0001111000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of a run of 1s</td>
<td>0001111000</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of a run of 0s</td>
<td>0001111000</td>
</tr>
</tbody>
</table>

Originally for Speed since shift faster than add for his machine

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.
### 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</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</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</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>0010</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>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>
Booth’s Algorithm

1. Depending on the current and previous bits, do one of the following:
   00: a. Middle of a string of 0s, so no arithmetic operations.
   01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
   10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   11: d. Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

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</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</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = 2$</td>
<td>&amp; 10</td>
</tr>
<tr>
<td>or immediate</td>
<td>or $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1,$2,10</td>
<td>$1 = ^$2</td>
<td>&amp;~10</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 arith.</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>slv $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>srlv $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>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" → \[ \text{msb} \rightarrow \text{lsb} \rightarrow "0" \]

*arithmetic*-- on right shifts, sign extend

\[ \text{msb} \rightarrow \text{lsb} \rightarrow "0" \]

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

\[ \text{sel} \rightarrow \begin{array}{l} 1 \\ 0 \end{array} \rightarrow \begin{array}{l} A \\ B \end{array} \rightarrow D \]

8-bit right shifter

\[ \begin{array}{l} A_7 \\ A_6 \\ A_5 \\ A_4 \\ A_3 \\ A_2 \\ A_1 \\ A_0 \end{array} \rightarrow \begin{array}{l} R_7 \\ R_6 \\ R_5 \\ R_4 \\ R_3 \\ R_2 \\ R_1 \\ R_0 \end{array} \rightarrow \begin{array}{l} S_2 \\ S_1 \\ S_0 \end{array} \]

- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes?
361 ALU.33

361 ALU.34

Summary

- Instruction Set drives the ALU design
- 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)
- What’s Missing from MIPS is Divide & Floating Point Arithmetic