Review 1/5: Steps in Executing MIPS Subset

1) Fetch Instruction and Increment PC
2) Read 1 or 2 Registers
3) Mem-ref: Calculate Address
   Arith-log: Perform Operation
   Branch: Compare if operands ==
4) Load: Read Data from Memory
   Store: Write Data to Memory
   Arith-og: Write Result to Register
   Branch: if operands ==, Change PC
5) Load: Write Data to Register

Outline
° Review Datapath, ALU
° Including subtract in a 32-bit ALU
° Review Binary Multiplication
° Administrivia, “What’s this Stuff Good For”
° Multiplier
° Review Binary Division
° Divider
° Conclusion
Review 3/5: Hardware Building Blocks

**AND Gate**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
</table>
| A → □ → C | \( \begin{array}{c|c|c|c}
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array} \) |

**OR Gate**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
</table>
| A → □ → C | \( \begin{array}{c|c|c|c}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array} \) |

**Inverter**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
</table>
| A → □ → C | \( \begin{array}{c|c|c}
0 & 1 \\
1 & 0 \\
\end{array} \) |

**Multiplexor**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
</table>
| A → 0 → C | \( \begin{array}{c|c|c|c}
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array} \) |

**Review 4/5: Add 1-bit Adder to 1-bit ALU**

° Now connect 32 1-bit ALUs together

**2's comp. shortcut: Negation (Lecture 7)**

° Invert every 0 to 1 and every 1 to 0, then add 1 to the result

- Sum of number and its inverted rep. (“one’s complement”) must be 111...111\textsubscript{two}
- 111...111\textsubscript{two}= -1\textsubscript{ten}
- Let \( \overline{x} \) mean the inverted representation of \( x \)
- Then \( x + \overline{x} = -1 \Rightarrow x + \overline{x} + 1 = 0 \Rightarrow \overline{x} + 1 = -x \)

° Example: -4 to +4 to -4

\( x : \) \( 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100 \textsubscript{two} \)

\( \overline{x} : \) \( 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0011 \textsubscript{two} \)

\( +1: \) \( 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0100 \textsubscript{two} \)

\( : \) \( 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1101 \textsubscript{two} \)

\( +1: \) \( 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100 \textsubscript{two} \)

° What about subtract?
How Do Subtract?

° Suppose added input to 1-bit ALU that gave the one’s complement of B
° What happens if set CarryIn0 to 1 in 32-bit ALU?
  • 32-bit Sum = A + B + 1
° Then if select one’s complement of B \( \overline{B} \), Sum is
  \[ A + \overline{B} + 1 = A + (\overline{B} + 1) = A + (-B) = A - B \]
° If modify 1-bit ALU, can do Subtract as well as And, Or, Add

1-bit ALU with Subtract Support

<table>
<thead>
<tr>
<th>Binvert</th>
<th>Op</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A and B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A and B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A or B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A or B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A + B + CarryIn</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A + B + CarryIn</td>
</tr>
</tbody>
</table>

32-bit ALU

° 32-bit ALU made from AND gates, OR gates, Inverters, Multiplexors
° Performs 32-bit AND, OR, Add, Sub (2’s complement)
° Op, Binvert, CarryIn are control lines; control datapath

MULTIPLY: From Lecture 9

° Paper and pencil example:
  
  \[
  \begin{array}{c}
  \text{Multiplicand} \quad 1000 \\
  \text{Multiplier} \quad 1001 \\
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  & 0 & 1 & 0 & 0 \\
  & 1 & 0 & 0 & 0 \\
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  & 0 & 1 & 0 & 0 \\
  \end{array}
  \]

  Product = 01001000

  • m bits x n bits = m+n bit product

° MIPS: mul, mulu puts product in pair of new registers hi, lo; copy by mfhi, mflo
  • 32-bit integer result in lo
MULTIPLY

• Binary multiplication is easy:
  • 0 → place 0 (0 x multiplicand)
  • 1 → place a copy (1 x multiplicand)
  • Shift the multiplicand left before adding to product

• 3 versions of multiply hardware & algorithm:
  • Successive refinement to get to real version
  • Go into first version in detail, give idea of others

Shift-Add Multiply Algorithm V. 1

1. Test Multiplier
   1. Test Multiplier

2. Shift the M’cand register left 1 bit
   2. Shift the M’cand register left 1 bit

3. Shift the M’plier register right 1 bit
   3. Shift the M’plier register right 1 bit

31nd repetition? No: < 32 repetitions

Done

Multiplier = datapath + control

Administrivia

• Project 5: Due today

• Next Readings: 3.12, 4.9
  • Optional: RISC Survey (including HP ISA) ftp://mkp.com/COD2e/Web_Extensions/survey.htm

• 9th homework: Due Friday (Ex. 7.35, 4.24)

• 10th homework: Due Wednesday 4/21 7PM
  • Exercises 4.43, 3.17 (assume each instruction takes 1 clock cycle, performance = no. instructions executed * clock cycle time, ignore CPI comment)
Administivia: Rest of 61C
F 4/16 Intel x86, HP instruction sets; 3.12, 4.9
W 4/21 Performance; Reading sections 2.1-2.5
F 4/23 Review: Procedures, Variable Args
(Due: x86/HP ISA lab, homework 10)
W 4/28 Processor Pipelining; section 6.1
F 4/30 Review: Caches/TLB/VM; section 7.5
(Due: Project 6-sprintf in MIPS, homework 11)
M 5/3 Deadline to correct your grade record
W 5/5 Review: Interrupts/Polling
F 5/7 61C Summary / Your Cal heritage
Sun 5/9 Final Review starting 2PM (1 Pimintel)
W 5/12 Final (5PM 1 Pimintel); mds@cory

“What’s This Stuff Good For?”
Swimmers looking for an edge over the competition can now use tiny half-ounce sensors that give them instant feedback on their power and their stroke rate, length and velocity, shown as real-time performance graphs generated by the PC. Unlike most, this swimmer generates more arm power in his butterfly than in his freestyle. “If he learns to apply that strength to his freestyle, he’ll almost certainly decrease his time.”
One Digital Day, 1998
www.intel.com/onedigitalday

Observations on Shift-Add Multiply V.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?

Shift-add Multiplier Version 2 (v. slide 14)
°32-bit Multiplicand reg, 32-bit ALU,
64-bit Product reg, 32-bit Multiplier reg

Multiplier = datapath + control
Shift-Add Multiplier Version 3

- Product reg. wastes space that exactly matches size of Multiplier reg
  ⇒ combine Multiplier reg and Product reg
  ⇒ "0-bit" Multiplier reg

Multiplier = datapath + control

Divide: From Lecture 9

Divisor 1000 1001010
Dividend
-1000
10
101
1010
-1000
10 Remainder
(or Modulo result)

Dividend = Quotient x Divisor + Remainder

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

DIVIDE HARDWARE Version 1

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

- Put Dividend into Remainder to start, collect Quotient 1 bit at a time

Divide Algorithm V. 1

Start: Place Dividend in Remainder

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

Remainder ≥ 0
Remainder < 0
Test
Remainder

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, and 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 repetition?
Yes: n+1 repetitions (n = 4 here)
No: < n+1 repetitions

Done
Example Divide Algorithm V. 1

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

<table>
<thead>
<tr>
<th>Remainder</th>
<th>Quotient</th>
<th>Divisor</th>
<th>(-Divisor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0111</td>
<td>0000</td>
<td>0010 0000</td>
<td>(1110 0000)</td>
</tr>
<tr>
<td>1 1110 0111</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0000 0111</td>
<td>0000</td>
<td>0001 0000</td>
<td>(1111 0000)</td>
</tr>
<tr>
<td>1 1111 0111</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0000 0111</td>
<td>0000</td>
<td>0000 1000</td>
<td>(1111 1000)</td>
</tr>
<tr>
<td>1 1111 1111</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0000 0111</td>
<td>0000</td>
<td>0000 0100</td>
<td>(1111 1100)</td>
</tr>
</tbody>
</table>

Example Divide Algorithm V. 1 (cont’d)

° Original division of \( 0111_{two} \) (7) by \( 0010_{two} \) (2)
gives a quotient of \( 0011_{two} \) (3) plus a remainder of \( 0001_{two} \) (1)

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 HARDWARE Version 2 (v. slide 28)

° 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
DIVIDE HARDWARE Version 3

- Eliminate Quotient register by combining with Remainder as shifted left 32-bit
  ⇒ “0-bit” Quotient reg

Multiply, Divide Algorithms Explain MIPS

- \( \text{mul} \) puts product in pair of regs \( \text{hi}, \text{lo} \);
  - 32-bit integer result in \( \text{lo} \)

- \( \text{MIPS}: \text{div}, \text{divu} \) puts Remainder into \( \text{hi} \), puts Quotient into \( \text{lo} \)

And in Conclusion..

- Subtract included to ALU by adding one’s complement of B

- Multiple by shift and add

- Divide by shift and subtract, then restore by add if didn’t fit

- Can Multiply, Divide simply by adding 64-bit shift register to ALU

- MIPS allows multiply, divide in parallel with ALU operations

- Next: Intel and HP Instruction Set Architectures