Announcement

- Homework assignment #4: Due on Feb 22, before class
  - Readings: Sections 3.2, 3.3, B.5, B.6
  - Problems: 3.7, 3.9, 3.10, 3.12, 3.18-3.22

- Homework assignment #5, Due on March 1, before class
  - Readings: Sections 3.4, 3.5, 3.6, 3.7
  - Problems: 3.27 and 3.28 (do \(x \times y\) and \(x \div y\) only), 3.29-3.30, 3.35, 3.38, 3.40, 3.42-3.43, 3.44

- Do the following For Your Practice questions. The solutions are on-line at http://www.mkp.com/companions/1558606041/solutions/COD3_Chip_03_FMT_Solutions.pdf
  - Problems: 3.34, 3.47, 3.48, 3.50-3.53

- Project #2 is due on 11:59PM, March 10. Quiz #2 is March 1st, Tuesday

Csci 136 Computer Architecture II
- Multiplication and Division

Xiuzhen Cheng
cheng@gwu.edu

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>(S1 = S2 + S3)</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,3</td>
<td>(S1 = S2 - S3)</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>(S1 = \text{constant})</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>adiu $1,$2,3</td>
<td>(S1 = S2 + S3)</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,3</td>
<td>(S1 = S2 - S3)</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>multi $2,3</td>
<td>Hi, Lo = $2 \times S3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multiu$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 = quotient, Hi = remainder</td>
<td>Hi = (S2 \mod S3)</td>
</tr>
<tr>
<td>divide unsigned remainder</td>
<td>divu $2,3</td>
<td>Lo = $2 \div S3, Hi = quotient, Hi = remainder</td>
<td>Hi = (S2 \mod S3)</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfh $1</td>
<td>S1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mfl $1</td>
<td>S1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

Review on Multiplication/Division Algorithms

- **Add and Shift algorithm for multiplication**
  - Left shift or right shift?
  - Shift the Product? Or Multiplier? Or Multiplicand?

- **Shift and Subtract algorithm for Division**

Examples:

- Multiply \(010110 \times 110011\)
- Divide \(00101101 \div 0110\)

Unsigned Multiplication: Shift Multiplicand Left (1/3)

<table>
<thead>
<tr>
<th>Multiplicand Is shifted left</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check each bit of the multiplier</td>
</tr>
<tr>
<td>How to check? = shift right and then check the LSB</td>
</tr>
<tr>
<td>Three registers:</td>
</tr>
<tr>
<td>Multiplicand: 64 bits</td>
</tr>
<tr>
<td>Multiplier: 32 bits</td>
</tr>
<tr>
<td>Product: 64 bits</td>
</tr>
</tbody>
</table>

Unsigned Multiplication: First Implementation (2/3)

Datapath

Control

Mult/Div

Inputs:

- \(x\)
- \(y\)

Outputs:

- \(x \times y\)
- \(x \div y\)

Three registers:

- Multiplicand: 64 bits
- Multiplier: 32 bits
- Product: 64 bits
Unsigned Multiplication: Observations (3/3)

- 1 clock cycle per step => < 100 clocks per multiply
- 1/2 bits in multiplicand always 0 => 64-bit adder is wasted
- 0 is inserted when shift the multiplicand left => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

Unsigned Multiplication: Shift the Product Right

| 0 1 0 1 1 0 |
| 1 1 0 0 1 1 |

| Observations: |
| ALU can be 32 bits |
| Multiplicand register can be 32 bits |
| No need for a Multiplier register |
| Product register must be 64 bits |

Multiplication: Final Version

- Multiplier starts in the right half of product

| Multiplicand | 32 bits |
| Product | Write 64 bits |

Control

1. Test Product0
2. Shift the Product register right 1 bit
3. If < 32 repetitions, No: Yes: 32 repetitions

Final MULT Implementation: Questions

- How about signed multiplication?
  - Booth’s algorithm
- How to deal with Overflow?
- Does the hardware work for Booth’s algorithm?
  - YES
  - Do sign-extension

Final Version Multiplication: Observations

- Two steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MULTU
- What about signed multiplication?
  - Remember signs and do unsigned multiplication.
  - Booth’s Algorithm is an elegant way to multiply signed numbers using same hardware as before and save cycles
  - can handle multiple bits at a time

In-Class Exercise

- Used the same datapath (the hardware for the final multiplication implementation) for Booth’s algorithm, what will be the control?
Divide: Paper & Pencil

See how big a number can be subtracted, creating quotient bit on each step
Binary \( = 1 \times \text{divisor} \) or \( 0 \times \text{divisor} \)

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

Unsigned Division: First Implementation (1/3)

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

---

Unsigned Division: First Implementation (2/3)

- Takes 33 steps for 32-bit Quotient & Rem.
  - Why?
  - Dividend
  - Divisor
  - Quotient
  - Remainder

Start: Place Dividend in Remainder

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 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.

3. Shift the Divisor register right 1 bit.

---

Unsigned Division: First Implementation Observations (3/3)

- 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

---

Unsigned Division: Refined Implementation (1/3)

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

Start: Place Dividend in Remainder

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.

---

Unsigned Division: Refined Implementation (2/3)

Start: Place Dividend in Remainder

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.

---
Unsigned Division: Refined Implementation Observations (3/3)

- 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 Divide: 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 – software needs to take care of this.
- Divided by 0? – software do this too.

Summary

- Multiply/Divide: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Common hardware support

Questions?