Integer Multiplication and Division
for ENCM 369: Computer Organization

Steve Norman, PhD, PEng

Electrical & Computer Engineering
Schulich School of Engineering
University of Calgary

Winter Term, 2015
Contents

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
About these slides

These slides will **not** be presented in lectures, due to time constraints.

The basics of integer multiplication and division with computers are important and useful to know, but are easier to learn by self-study than either virtual memory or floating-point number formats.

To allow more time for the harder topics, your course instructors decided to leave integer multiplication and division out of lectures in 2015.
Relevant textbook material

On page 314, there’s a brief explanation of the MIPS integer multiply and divide instructions and the related Hi and Lo registers.

Sections 5.2.6 and 5.2.7 provide introductions to hardware designs for integer multiplication and division. There won’t be final exam questions on these hardware design ideas, but the reading is interesting and short.
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
Integer multiplication: concepts

Let's start by reviewing the algorithm taught to elementary school kids for base ten multiplication, with an example . . .

\[
\begin{array}{c}
\phantom{234} \\
2 3 4 \\
\times 1 1 3 \\
\hline
7 0 2 \\
2 3 4 \\
+ 2 3 4 \\
\hline
2 6 4 4 2
\end{array}
\]
Base two integer multiplication

The algorithm from the previous slide is easy to adapt to base two . . .

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \leftarrow \textit{multiplicand} \\
\times & 1 & 1 & 0 & 1 \leftarrow \textit{multiplier} \\
\hline
& 1 & 0 & 1 & 1 \\
& 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
+ & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \leftarrow \textit{product}
\end{array}
\]

It works!
\[11_{\text{ten}} \times 13_{\text{ten}} = 143_{\text{ten}}.\]

In base two the rules for generating rows of bits to add are simple: A row is either zero or a shifted copy of the multiplicand.
Base two integer multiplication: width of product

In this example, multiplying two 4-bit integers results in an 8-bit product.

In general the width of the product of two $N$-bit integers can be up to $2N$ bits.
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
Unsigned integer multiplication hardware: Problem definition

We’re trying to compute \( x \times y \).
Bits of \( x \) are \( x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \).
Bits of \( y \) are \( y_{n-1}, y_{n-2}, \ldots, y_1, y_0 \).
A product of 1 bit from \( y \) and 1 bit from \( x \) is \( y_i x_j \).
Getting product bits \( p_{2n-1}, p_{2n-2}, \ldots, p_1, p_0 \) is a big addition problem. For example, with \( n = 4 \), ...

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>y_0 x_3</th>
<th>y_0 x_2</th>
<th>y_0 x_1</th>
<th>y_0 x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>y_1 x_3</td>
<td>y_1 x_2</td>
<td>y_1 x_1</td>
<td>y_1 x_0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>y_2 x_3</td>
<td>y_2 x_2</td>
<td>y_2 x_1</td>
<td>y_2 x_0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>y_3 x_3</td>
<td>y_3 x_2</td>
<td>y_3 x_1</td>
<td>y_3 x_0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0 \]
Building blocks for a combinational unsigned multiplier

Multiplying bits $y_i$ and $x_j$ just takes an AND gate!

<table>
<thead>
<tr>
<th>$y_i$</th>
<th>$x_j$</th>
<th>$y_i x_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We will need a lot of 1-bit full adders …
4 × 4 unsigned combinational multiplier

To reduce clutter in this diagram, AND gates are not shown.
Scaling the combinational multiplier design for 32-bit inputs

Coding a $4 \times 4$ multiplier in VHDL and testing it in a PLD was a lab exercise in ENEL 353. It was a fun little toy but the point of it really was to show the structure of much larger multipliers.

The design for the $4 \times 4$ multiplier can be extended to $N \times N$ in a fairly obvious way.

A 1-bit adder can generate correct outputs a tiny fraction of a processor clock cycle after the adder inputs change.

In a $32 \times 32$ multiplier, when $x$ and $y$ change, it should be possible for product bits $p_{63}$ to $p_0$ to be correct within one or two processor clock cycles.
Scaling the combinational multiplier design for 32-bit inputs, continued

For $N = 32$, there are 31 rows of 32 adders, for a total of 992 adders.

There are $32 \times 32$ AND gates, for a total of 1024 AND gates.

A 2-input AND gate: 6 transistors.

A 1-bit full adder: 8–28 transistors, depending on the full adder design.

Rough estimate for the transistor count, using the worst-case number of transistors for each 1-bit adder:

$1000 \times (6 + 28) = 34000$. 
Is 34000 transistors affordable for a multiplier in a processor?

Let’s go back to 1985. The Intel 80386 had a total of 275,000 transistors. Using over 10% of the “transistor budget” for a multiplier would not be acceptable.

Now consider 2008. An Intel Atom processor used in a cheap “netbook” computer had 47,000,000 transistors. Here the multiplier would use less than 0.1% of the transistor budget—no problem!

This illustrates an important point: Huge improvements in integrated circuit technology have changed over and over again what it means for the design of a component—in this example, a multiplier—to be “too large to be practical”.
A 4 × 4 sequential multiplier

This circuit was the subject of an exercise in the last problem set in ENEL 353 in Fall 2013 . . .

A key point is that there are only 4 1-bit adders, not a 3 × 4 array of adders as in the combinational design.
Operation of $4 \times 4$ sequential multiplier

It takes 1 clock cycle to initialize the flip-flops, then 4 more to finish all the required adding and shifting . . .

<table>
<thead>
<tr>
<th>end of init. cycle</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>y₃</th>
<th>y₂</th>
<th>y₁</th>
<th>y₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>end of 1st add/shift cycle</td>
<td>0</td>
<td>y₀ₓ₃</td>
<td>y₀ₓ₂</td>
<td>y₀ₓ₁</td>
<td>y₀ₓ₀</td>
<td>y₃</td>
<td>y₂</td>
<td>y₁</td>
</tr>
<tr>
<td>end of 2nd add/shift cycle</td>
<td>intermediate result</td>
<td>y₃</td>
<td>y₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end of 3rd add/shift cycle</td>
<td>another intermediate result</td>
<td>y₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end of 4th add/shift cycle</td>
<td>p₇</td>
<td>p₆</td>
<td>p₅</td>
<td>p₄</td>
<td>p₃</td>
<td>p₂</td>
<td>p₁</td>
<td>p₀</td>
</tr>
</tbody>
</table>
Scaling the sequential multiplier for 32-bit inputs

The $4 \times 4$ sequential multiplier design can be scaled up to handle 32-bit inputs.

It will require 33 clock cycles to complete a multiplication.

How many transistors will it require? Assume, as before, 6 transistors per AND gate and 28 per 1-bit adder. Assume also 20 transistors per DFF and 6 per 2-input/1-bit mux.

<table>
<thead>
<tr>
<th>component</th>
<th>count</th>
<th>transistor count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>64</td>
<td>$64 \times 6 = 384$</td>
</tr>
<tr>
<td>1-bit adder</td>
<td>32</td>
<td>$32 \times 28 = 896$</td>
</tr>
<tr>
<td>D flip-flop</td>
<td>64</td>
<td>$64 \times 20 = 1280$</td>
</tr>
<tr>
<td>mux</td>
<td>32</td>
<td>$32 \times 6 = 192$</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td></td>
<td><strong>2752</strong></td>
</tr>
</tbody>
</table>
Comparison of $32 \times 32$ multiplier designs

The sequential circuit is *much slower* than the combinational circuit, but with fewer than 2800 transistors, the sequential circuit is *much smaller*, and would have been the correct choice for the 80386 back in 1985, with the limited transistor budget for the 80386.

**Attention:** All of the work on transistor counts in the preceding slide is *approximate*. Exact transistor counts would depend on a lot of detailed design considerations I haven’t gone into.

But the main point is valid—combinational circuits are relatively big and fast, while sequential circuits are relatively small and slow.
What about *signed* integer multiplication?

Reminder: An $N$-bit adder for *unsigned* addition “does the right thing” for *signed* addition as well.

The same idea *does not work* for *signed* versus *unsigned* integer *multiplication*. Relatively minor enhancements to the unsigned multiplier are needed to support signed multiplication. (Details of this are *not* covered in ENCM 369.)
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
MIPS instructions: mult and multu

The sources are GPRs.

Signed multiplication: mult src1, src2

Unsigned multiplication: multu src1, src2

The destinations are not specified in the instructions, and the destinations are not GPRs!

Bits 31–0 of the product go to a special-purpose register called Lo; bits 63–32 go to a special-purpose register called Hi.
Examples for mult and multu

Suppose $t0 = 0x0000_0002$ (2, signed or unsigned).
Suppose $t1 = 0xffff_ffff$ (−1 if signed, about 4.3 billion if unsigned).

What is the effect of mult $t0$, $t1$?
What is the effect of multu $t0$, $t1$?
Examples for mult and multu: answers

Signed arithmetic with **mult**: $2 \times -1 = -2$. The 64-bit two’s complement representation of $-2$ is

```
0xffff_ffff_ffff_fffe
```

So Hi gets $0xffff_ffff$ and Lo gets $0xffff_fffe$.

Unsigned arithmetic with **multu**: The result can be obtained by shifting $0xffff_ffff$ left by 1 bit: $0x1_ffff_fffe$. (If that’s not clear, do the work in base two to see how it works.)

So Hi gets $0x0000_0001$ and Lo gets $0xffff_fffe$.

Interesting fact, not proven in ENCM 369: Given the same two input GPR values, **mult** and **multu** may put different results in Hi, but will always put identical results in Lo.
mflo and mfhi: getting data out of Lo and Hi

\begin{align*}
\text{mflo} & \quad \text{dest} \quad \text{— copy 32-bit pattern from Lo to GPR dest} \\
\text{mfhi} & \quad \text{dest} \quad \text{— copy 32-bit pattern from Hi to GPR dest}
\end{align*}
Let $i, j, k$ be 32-bit ints in C or C++ code. *How will*
$k = i \times j$; *be implemented?*

*What does this mean for programmers doing multiplication with integers?*

*How would this work in MIPS? Suppose $i$ is in $s0$, $j$ is in $s1$, $k$ is in $s2$.***
C, C++, multiplication, and 32-bit ints: answers

\[ k = i \times j; \ldots \] In a typical system, \( k \) will get bits 31–0 of the 64-bit product, **even if that is incorrect** according to ordinary mathematics.

For example, if the values of \( i \) and \( j \) are both \( 70,000_{\text{ten}} \) (which is \( 0x11170 \)), the 64-bit product is \( 0x0000_0001_2410_1100 \).

\( k \) will get a value of \( 0x2410_1100 \), which is \( 605,032,704_{\text{ten}} \), not \( 4,900,000,000_{\text{ten}} \), which can’t be represented as a 32-bit int.

**What this means for programmers:** With integer multiplication (as with addition and subtraction) it is **your responsibility** to make sure that numbers don’t get so large that your integer types don’t represent them correctly!
If i is in $s0, j is in $s1, k is in $s2, then \( k = i \times j \); would get translated as

```
mult $s0, $s1 # do the multiplication
mflo $s2        # k = bits 31-0 of product
```

Bits 63–32 of the product, in the Hi register, would not be used.
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
Integer division: concepts

As you saw in ENGG 233 and ENCM 339, in computers, division with integers typically produces integer results.

In a typical programming system, $34 \div 6$ is not $5 \frac{2}{3} \approx 5.667$, but rather 5 with a remainder of 4.

The terms dividend, divisor, quotient, and remainder are important to know. In the above example, the dividend was 34, the divisor was 6, the quotient was 5, and the remainder was, as already stated, 4.

The next slide illustrates how those results can be obtained using long division in base two . . .
Base two division example

There is a sequence of steps. At each step there is a decision about whether or not to subtract a shifted copy of the divisor from an “intermediate remainder”.

\[
\begin{array}{c|c}
\text{divisor} & 1 & 1 & 0 \\
\hline
\text{dividend} & 1 & 0 & 0 & 0 & 1 & 0 \\
\text{remainder} & 1 & 0 & 0 & 0 & 1 & 0 \\
\text{quotient} & 1 & 0 & 1 & \end{array}
\]
Division hardware

A decision about whether or not to subtract a shifted copy of the divisor has to be made after each “intermediate remainder” is generated. This makes hardware implementations of long division slow.

This can be done either sequentially, or with a slow combinational circuit, as illustrated in textbook Figure 5.20 (page 254).

Division is usually much slower than addition, subtraction, or combinational multiplication!

Details of division hardware are not an ENCM 369 topic.
Basic Rule of Integer Division

There are two parts to the basic rule:
- quotient $\times$ divisor + remainder = dividend
- $|\text{remainder}| < |\text{divisor}|

The basic rule completely specifies results for unsigned division.

What about signed division? For example, what are the quotient and remainder for $(-14)/3$?

It turns out that for $(-14)/3$ that two pairs of results satisfy the basic rule:
- quotient = $-5$, remainder = 1
- quotient = $-4$, remainder = $-2$
Signed integer division: Messy details

As seen in the example on the previous slide, for division when one or both of dividend and divisor are negative, two different quotient-and-remainder pairs satisfy the Basic Rule (except when the remainder is zero).

Over history, computer designers have not agreed on which results should be considered correct.

So not all C and C++ standards fully specify results of integer division involving negative numbers.
In addition to the Basic Rule, most modern processor designs implement an extra rule: $|A/B|$ must be equal to $|A|/|B|$.

For example this extra rule says for $(-14)/3$ that the quotient is $-4$ and the remainder is $-2$.

The extra rule is equivalent to saying that integer quotients are exact quotients rounded toward zero.

(But, if you’re a Python fan, see http://python-history.blogspot.ca/2010/08/why-pythons-integer-division-floors.html for an explanation why round-toward-zero is not what you get with integer division in Python.)
Out-of-range results in integer division (assuming that the divisor is not zero!)

In adding, subtracting, or multiplying a pair of $N$-bit numbers, sometimes the “everyday math” result can’t be represented as an $N$-bit number.

This **never** happens in **unsigned division**.

In **signed division** there is only one pair of numbers that causes a problem—the dividend is the “most negative” two’s-complement number, and the divisor is $-1$. See your compiler documentation and/or your instruction set documentation if you care about what happens in that unusual case.
Contents of “Integer Multiplication and Division”

About these slides

Integer Multiplication: Concepts

Hardware design for integer multiplication

MIPS Instructions for Integer Multiplication

Integer Division: Concepts

MIPS Integer Division Instructions and How C and C++ Compilers Use Them
MIPS instructions: `div` and `divu`

The instructions are

- `div` \texttt{dividend, divisor} for signed division
- `divu` \texttt{dividend, divisor} for unsigned division

For both instructions: the dividend and divisor must be GPRs; the quotient goes to the Lo register; the remainder goes to the Hi register. These are the same Lo and Hi registers written to by the `mult` and `multu` instructions.

Notes . . .

- Behaviour on divide-by-zero is undefined—programmers should not expect meaningful bit patterns in Hi or Lo.
- The MIPS Instruction Set Reference does not say what happens for `div` when \(0x8000_0000 / 0xffff_ffff\) is attempted.
MIPS instructions generated by C/C++ compilers

Suppose i in $s0, j in $s1, and k in $s2 are all of type unsigned int.

### C code

```c
k = i / j;
```

### possible translation

```assembly
bne $s1, $zero, L1

L1: divu $s0, $s1
```

```
mflo $s2
```

```c
k = i % j;
```

```assembly
as above with mfhi in place of mflo
```

What if i in $s0, j in $s1, and k in $s2 are all of type “plain” int instead of unsigned int? Replace divu with div.
MIPS instructions generated by C and C++ compilers, continued

For integer addition, subtraction and multiplication, C and C++ compilers typically allow programs to continue when the arithmetic results are incorrect in terms of “everyday math”.

But integer divide-by-zero typically causes a program-terminating exception.

Why is there a difference in behaviour?

A guess at an answer: For addition, subtraction and multiplication, there are somewhat rare cases in which it makes sense to continue with a result that is wrong in the “everyday math” sense. It’s hard to think of a situation in which the result of integer divide-by-zero is meaningful.