MK MORGAN KAUFMANN

COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

5th Edition

Chapter 3

Integer Math

Recap - MIPS Instructions

- Consider a comparison instruction:

  slt $t0, $t1, $zero

  where $t1 contains the 32-bit number: 1111 01…01

  What gets stored in $t0?

- The result depends on whether $t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either **slt** or **sltu**

```
slt    $t0, $t1, $zero     stores  1 in $t0
sltu   $t0, $t1, $zero     stores  0 in $t0
```

Recap - Number Representations

- 32-bit signed numbers (2's complement):

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
...
0111 1111 1111 1111 1111 1111 1111 1110two = + 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = – 2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = – 2,147,483,647ten
...
1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```


- Converting < 32-bit values into 32-bit values
  - Copy the most significant bit (the sign bit) into the "empty" bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

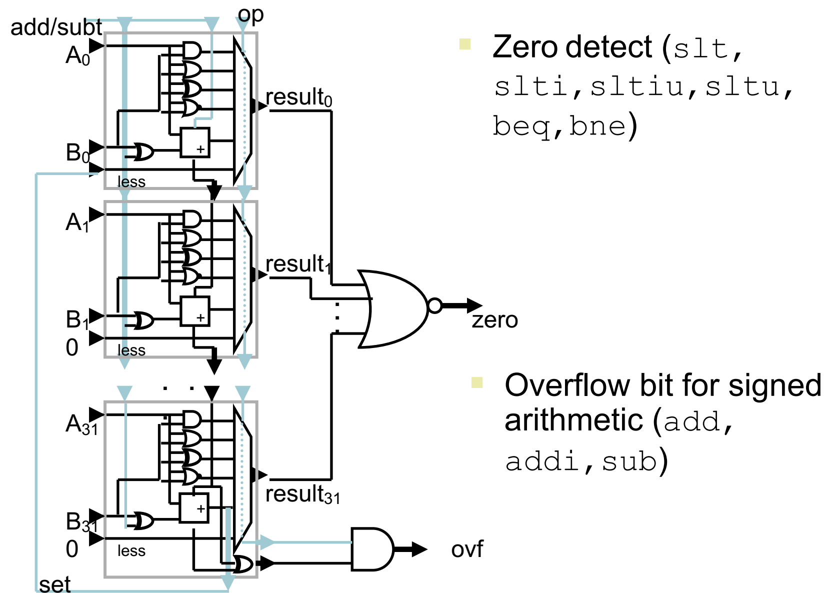
  - **Sign extend** versus **zero extend**

Integer Addition

- Example: 7 + 6

- Binary addition is similar to decimal addition.
- For subtraction, simply add the negative number:
  - A - B involves taking the two's complement of B (negating B's bits and adding 1) and adding to A.

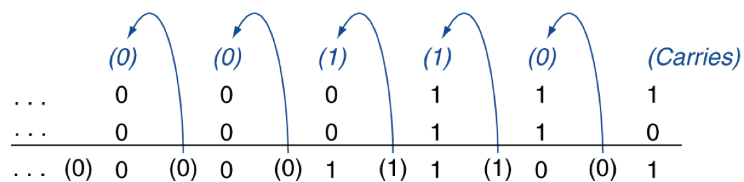
A MIPS ALU Implementation



- Zero detect (`slt, slti, sltiu, sltu, beq, bne`)
- Overflow bit for signed arithmetic (`add, addi, sub`)

Signed Integer Addition

- Example: 7 + 6



- An **Overflow** occurs if the result is out of range
  - Adding a + and - operand, no overflow possible.
  - Adding two + operands
    - Overflow if sign bit is 1.
  - Adding two – operands
    - Overflow if sign bit is 0.

Signed Integer Subtraction

- Add negation of second operand.
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two + or two – operands, no overflow.
  - Subtracting + from – operand
    - Overflow if result sign is 0.
  - Subtracting – from + operand
    - Overflow if result sign is 1.

Summary of Overflow Conditions

- Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a ***value*** bit of the result and not the proper ***sign*** bit.
  - When adding operands with different signs or when subtracting operands with the same sign, overflow can ***never*** occur.

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Summary of Overflow Conditions

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

- MIPS signals overflow with an exception – an unscheduled procedure call where the Exception Program Counter (EPC) contains the address of the instruction that caused the exception.
- MIPS **addu** and **subu** instructions *will not* cause an overflow – to detect the overflow, other instructions would have to be executed.

Detecting Overflow Logically

- When adding two's complement numbers, overflow will only occur if:
  - The numbers being added have the same sign;
  - The sign of the result is different then the sign of the two operands.
- If we perform the addition

$$\begin{array}{l} \;\; a_{n-1}\, a_{n-2} \ldots a_1\, a_0 \\ +\, b_{n-1} b_{n-2} \ldots b_1\, b_0 \\ \text{-----------------------------------} \\ =\, s_{n-1} s_{n-2} \ldots s_1\, s_0 \end{array}$$

- Overflow can be detected as

$$V = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$

- Overflow can also be detected as

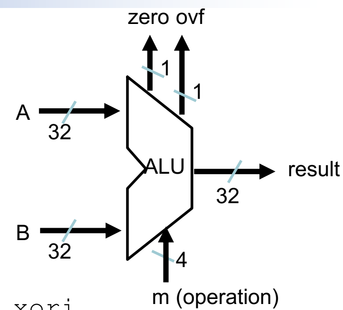
$$V = c_n \otimes c_{n-1}$$

where c_{n-1} and c_n are the carry in and carry out of the most significant bit.

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

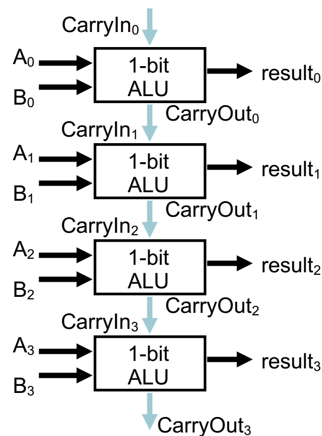
```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```



- With special handling for
  - Sign extend – `addi, addiu, slti, sltiu`
  - Zero extend – `andi, ori, xori`
  - Overflow detection – `add, addi, sub`

What about Performance?

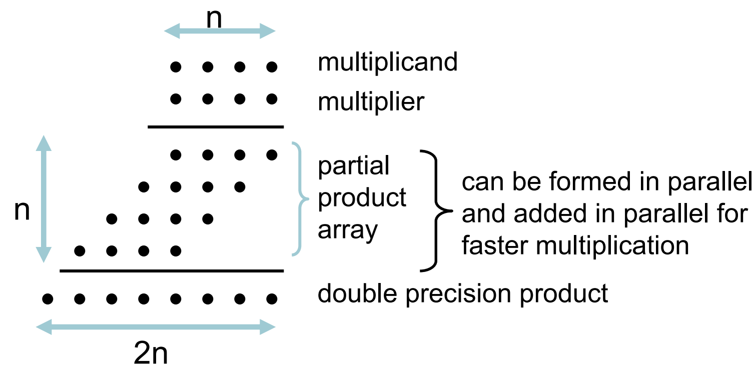
- Critical path of n-bit ripple-carry adder is *n(1-bit delay)*



- Solution – throw hardware at it (Carry Lookahead).

Multiply

- Binary multiplication can be just a ***bunch*** of right shifts and adds:

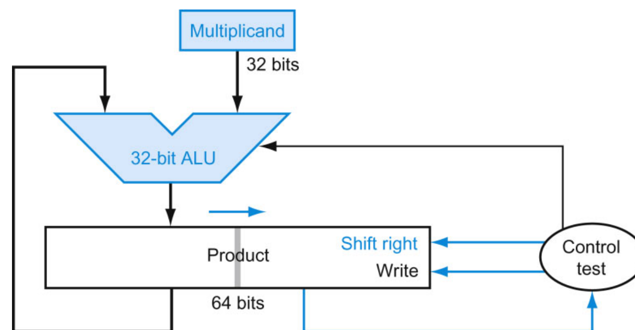


Multiplication Example

```
Multiplicand          1000
Multiplier          x 1001
                    --------
                      1000
                     0000
                    0000
                   1000
                   ---------
Product            1001000
```

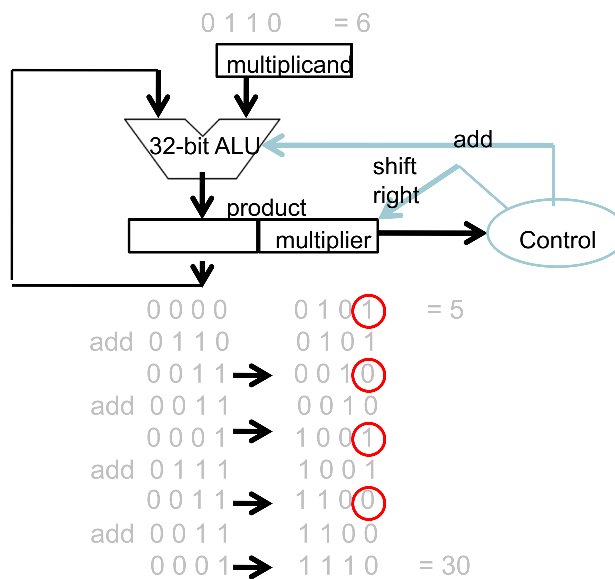
- In every step:
  - Multiplicand is shifted.
  - Next bit of multiplier is examined (also a shifting step).
  - If this bit is 1, shifted multiplicand is added to the product.

HW Multiplication Hardware



- 32-bit ALU and multiplicand are untouched.
- The sum keeps shifting right.
- At every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register.

Multiplication Example

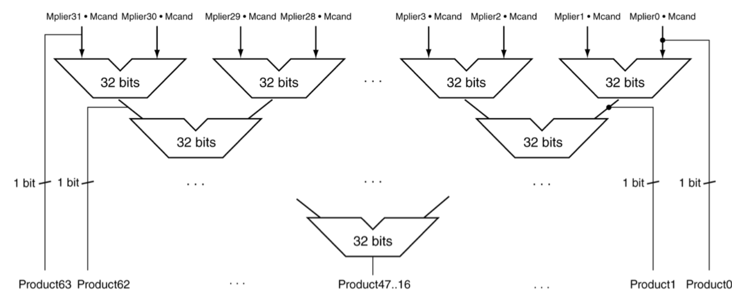


Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form).
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs are different.
- The product of two 32-bit numbers can be a 64-bit number -- hence, in MIPS, the product is saved in two 32-bit registers – HI and LO.
- At every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register.

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoffs
    - A clock is not required.
    - Much higher hardware cost.



- Can be pipelined
  - Several multiplications performed in parallel.

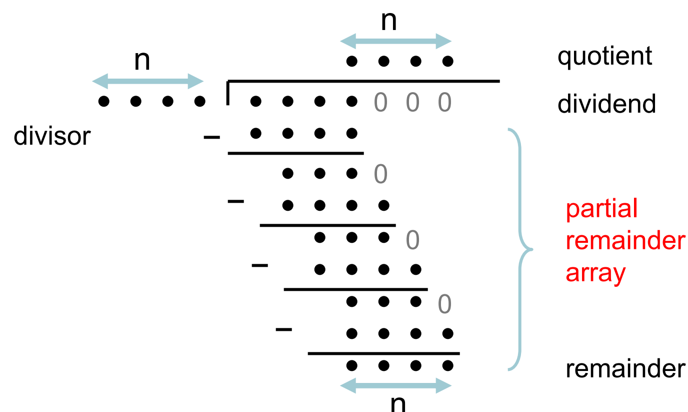
MIPS Multiplication Instructions

- Two 32-bit registers for product
  - HI: most-significant 32-bits.
  - LO: least-significant 32-bits.
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

Division

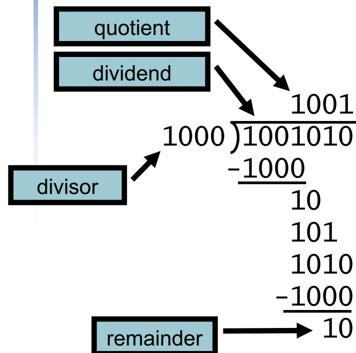
- Division is just a ***bunch*** of quotient digit guesses and left shifts and subtracts.

dividend = quotient x divisor + remainder



Division

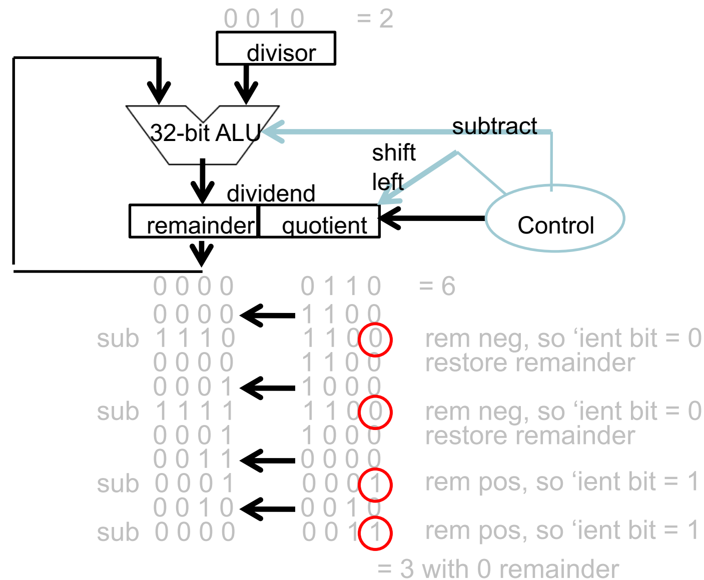
74/8 = 9 rem 2



n-bit operands yield *n*-bit quotient and *n*-bit remainder.

- Check for 0 divisor.
- Long division approach
  - If divisor bits ≤ dividend bits
    - 1 bit in quotient, subtract.
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit.
- Restoring division
  - Always do the subtract, and if remainder is < 0, add divisor back.
- Signed division
  - Divide using absolute values.
  - Adjust sign of quotient and remainder as required.

Division Hardware - Left Shift and Subtract



MIPS Divide Instruction

- Divide (`div` and `divu`) generates the remainder in `hi` and the quotient in `lo`

```
div    $s0, $s1        # lo = $s0 / $s1
                       # hi = $s0 mod $s1
```

0	16	17	0	0	0x1A

  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and remainder to registers in the register file.

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder.
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step (4 on today’s high-end processors)
  - Uses table lookup.
  - Still requires multiple steps.
- http://en.wikipedia.org/wiki/Division_(digital)

Next Class - Floating Point

- What can be represented in N bits?
  - Unsigned 0 to $2^{N}-1$
  - 2's Complement -2^{N-1} to $2^{N-1} - 1$
- But, what about--
  - Very large numbers?
    - 9,349,398,989,787,762,244,859,087,678
    - 1.23×10^{67}
  - Very small numbers?
    - 0.0000000000000000000000045691
    - 2.98×10^{-32}
  - Fractional values? 0.35
  - Mixed numbers? 10.28
  - Irrationals? π