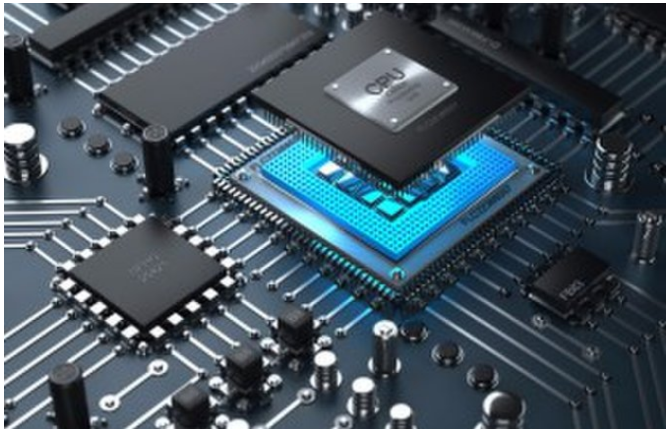


Arithmetic for Computers 1

CS 154: Computer Architecture
Lecture #8
Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Lab 4 underway…
- **Syllabus (Schedule Section) has been updated**

Midterm Exam (**Wed. 2/12**)

What's on It?

- Everything we've covered in lecture from start to Monday, 2/10

What Else?

- Closed book – some notes (details to follow)
- Random seat assignments – come to class EARLY!

Lecture Outline

- MIPS Instructions: Arrays vs. Pointers

- Arithmetic
  - Addition / Subtraction
  - Multiplication / Division

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array (the classic way)

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

Example: Clearing an Array (using a pointer)

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2  # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Comparison of the Two…

```
      move  $t0,$zero       # i = 0                    move  $t0,$a0        # p = & array[0]
loop1:sll   $t1,$t0,2       # $t1 = i * 4              sll   $t1,$a1,2      # $t1 = size * 4
      add   $t2,$a0,$t1     # $t2 = &array[i]          add   $t2,$a0,$t1    # $t2 = &array[size]
      sw    $zero, 0($t2)   # array[i] = 0      loop2: sw    $zero,0($t0)   # Memory[p] = 0
      addi  $t0,$t0,1       # i = i + 1                addi  $t0,$t0,4      # p = p + 4
      slt   $t3,$t0,$a1     # $t3 = (i < size)         slt   $t3,$t0,$t2    # $t3=(p<&array[size])
      bne   $t3,$zero,loop1# if () go to loop1         bne   $t3,$zero,loop2# if () go to loop2
```

- Version on the left must have the "multiply" and add inside the loop

- Memory pointer version on the right increments the pointer p directly.
- Moves the scaling shift and the array bound addition *__outside__* the loop
- It reduces instructions executed per iteration from 6 to 4.

- This is how a lot of compilers optimize code like this.

Arithmetic Overview 1

- Addition / subtraction
  - Carry out vs. Overflow – remember the difference!

Examples in 8-bit adders:

- 0x24 + 0xB0 *0xD4, C = 0, V = 0*
- 0x7F + 0x66 *0xE5, C = 0, V = 1*
- 0x15 + 0xFB *0x10, C = 1, V = 0*
- 0x87 + 0xAA *0x31, C = 1, V = 1*

Dealing with Overflow in C/C++

- Some languages (e.g., C/C++, Java) ignore overflow
- What happens when you do:

  0x87000000 + 0xAA000000 in C++?

  (i.e. –2,030,043,136 + –1,442,840,576?)

  - You get 822,083,584… discuss…

- In MIPS, you'd use

  addu, **addui**, **subu** instructions to not trigger overflow

  (this is what a C/C++ compiler would issue)

- Why?
  - Checking for overflow for every calculation can be demanding on CPU run time

Dealing with Overflow in Other Languages

- Other languages (e.g., Ada, Fortran – older ones) require raising an exception

- In MIPS, you'd use MIPS **`add`**, **`addi`**, **`sub`** instructions

- What actually happens?
  - On overflow, an "exception handler" is invoked
  - PC is saved in exception program counter (EPC) register
  - Jump executed to predefined handler address
  - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic Overview 2

- Multiplication
  - Left bit shifting by N bits ←→ Multiplying by 2^N
  - Using **`mult`** / **`multu`** and **`mflo`** (or **`mfhi`**)

- Division
  - Right bit shifting by N bits ←→ Integer divide by 2^N
  - Using **`div`** / **`divu`** (again, with **`mflo`** or **`mfhi`**)
    - No checking for overflow or divide-by-zero
  - Raises questions about floating point...
    - Will be coming up…

Multiplication in Computers: The Algorithm using a Decimal Example

- Let **P** be the partial *product*,
  M be the *multiplicand*,
  and **N** be the *multiplier*
  - *i.e. P eventually will be = M * N*

Initially, P is 0
Loop:
    If N is 0, then P = the result, exit Loop
    Else, P += (the rightmost digit of N) times M
    Shift N <u>right</u> once, and M <u>left</u> once
Repeat Loop

Example with Decimals
803 * 151 *(which we expect to be 121,253)*

P	M	N
0	803	151

1. N is not 0
2. P **+=** (rightmost digit of N[1]) * M[803]
   Shift N right once, M left once
   N is not 0
3. P **+=** (rightmost digit of N[5]) * M[8030]
   Shift N right once, M left once
   N is not 0
4. P **+=** (rightmost digit of N[1]) * M[80300]
   Shift N right once, M left once
   N IS 0 ; END

Example with Decimals
803 * 151 *(which we expect to be 121,253)*

P	M	N
0	803	151
803	8030	15
40953	80300	1
121253	803000	0

1. N is not 0

2. P **+=** (rightmost digit of N[1]) * M[803]
   Shift N right once, M left once
   N is not 0
3. P **+=** (rightmost digit of N[5]) * M[8030]
   Shift N right once, M left once
   N is not 0
4. P **+=** (rightmost digit of N[1]) * M[80300]
   Shift N right once, M left once
   N IS 0 ; END

Multiplication in Computers: The Algorithm using a **Binary** Example

- …Even easier than the decimal example:
  Shown here for 32 bits

Initially, P is 0
Loop 32 times:
    If $N_{bit0} = 1$, then P += M
    Shift N right once, and M left once

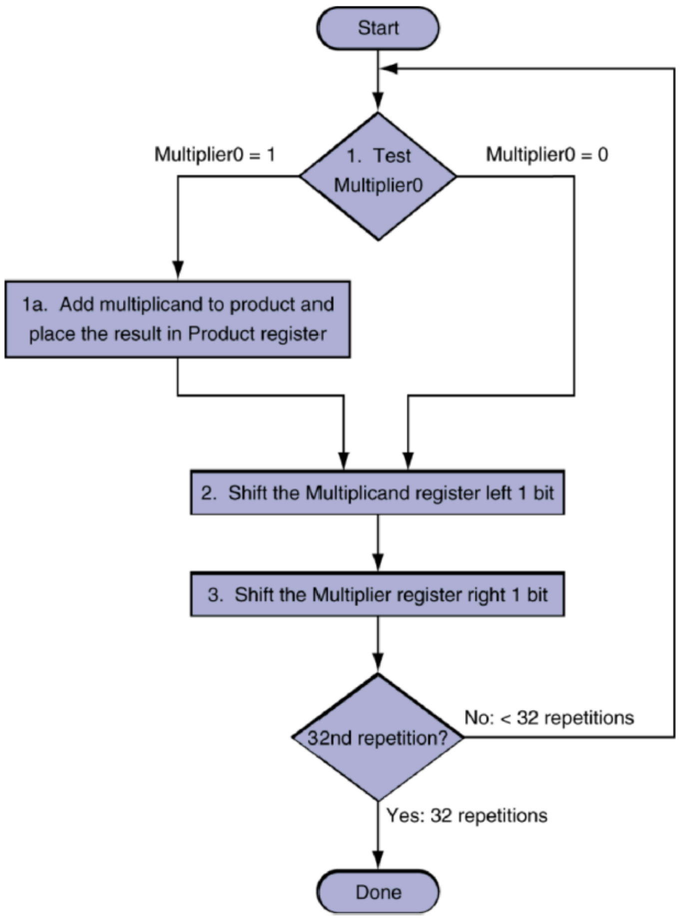
Simple Example using 8 bits

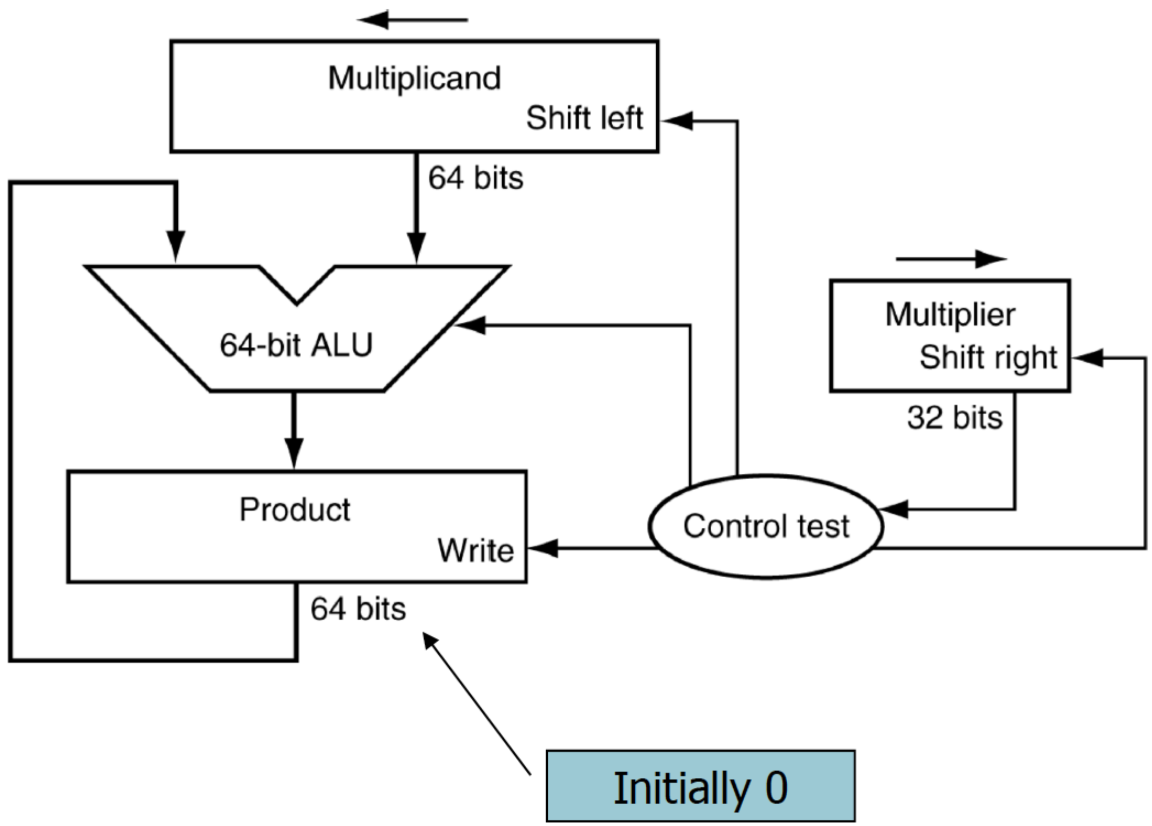
M = 0x04 = 0000 0100 (multiplicand)

N = 0x05 = 0000 0101 (multiplier)

- P = 0
- N_0 = 1 → P += 0x04 = 0x04, N = 0000 0010, M = 0000 1000
- N_0 = 0 → P = 0x04 (unchanged), N = 0000 0001, M = 0001 0000
- N_0 = 1 → P += 0x10 = 0x14, N = 0000 0000, M = 0010 0000
- Exit with **P = 0x14** (correct answer, since 0x14 = 20)

Multiplication Hardware



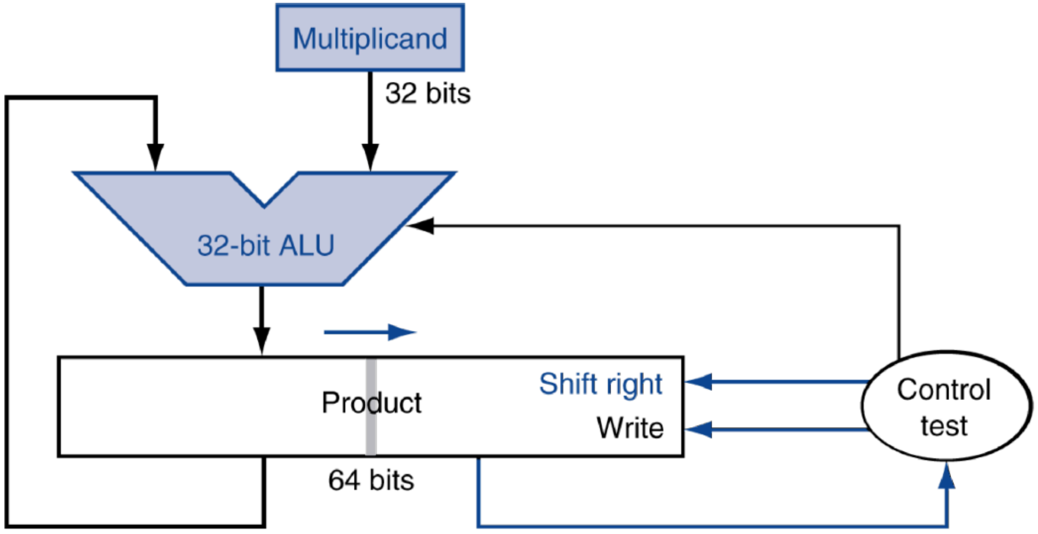


Can be further optimized with added HW

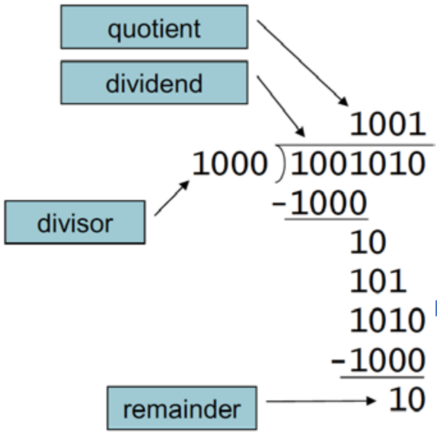
Optimization of HW for Multiplication

- You can perform some steps in parallel: add/shift

- One cycle per partial-product addition is ok to do, if frequency of multiplications in program is low



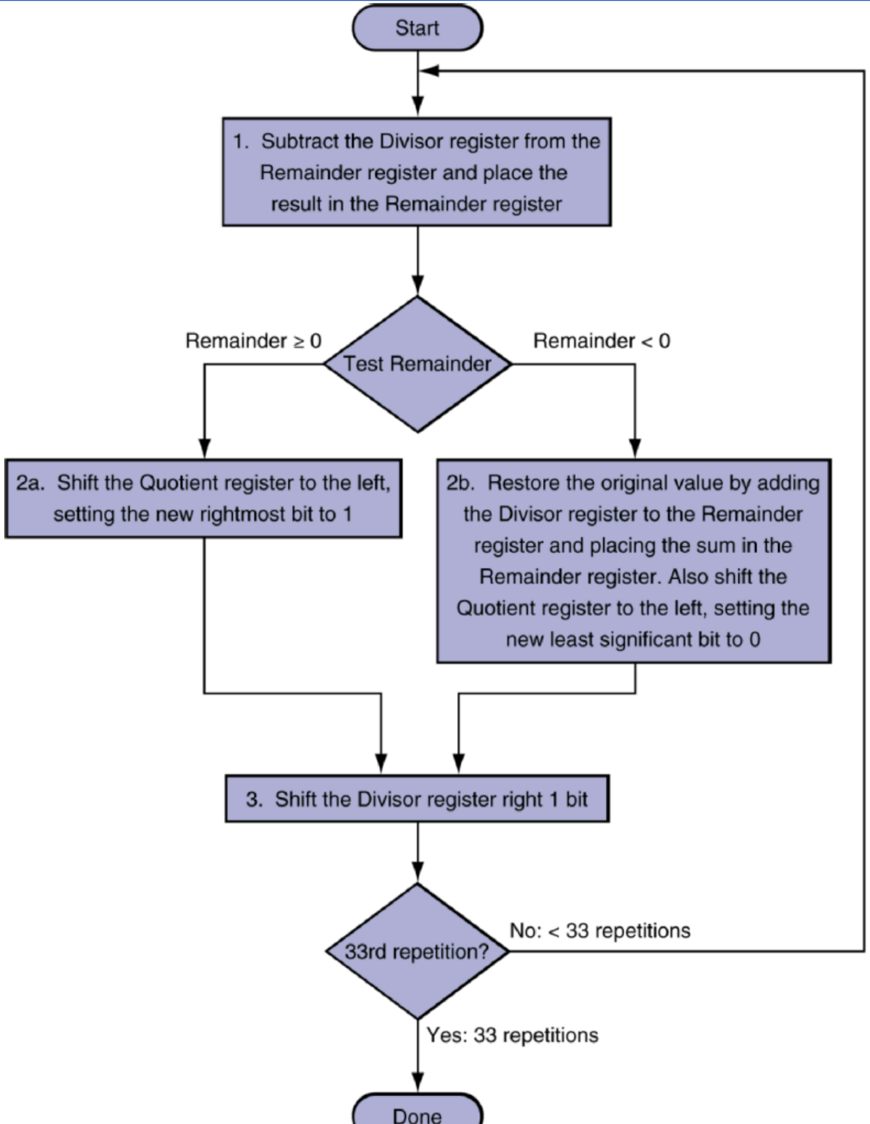
Division in Computers: The Algorithm

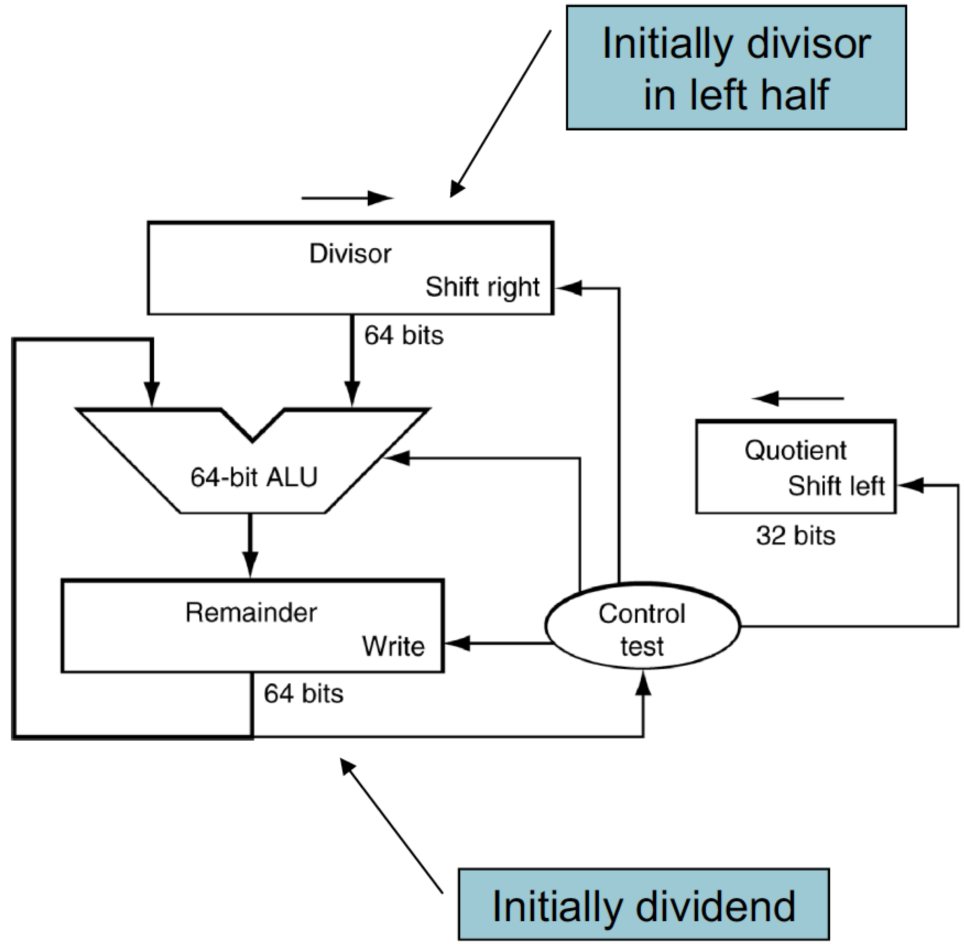


- Dividend (N) ÷ Divisor (D)= Quotient, Remainder

```
Initially, R = N
Loop 32 times:
        R = R – D
        If R ≥ 0, then
                shift Q to left 1 bit
                set LSB to 1 (that is, Q | 1)
        Else
                R = R + D
                shift Q to left 1 bit
        Shift D 1 bit to right,
```

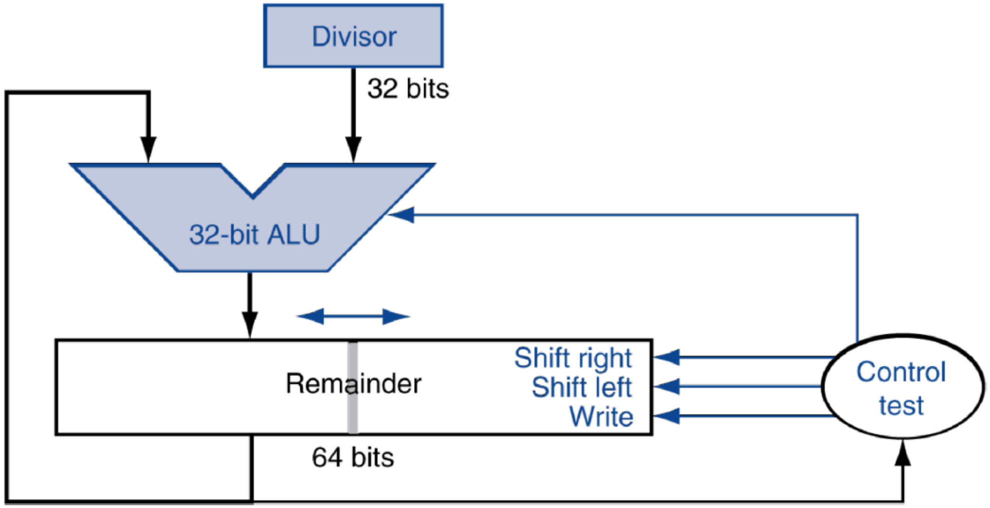
Division Hardware





Optimization of HW for Division

- One cycle per partial-remainder subtraction



- Looks a lot like a multiplier!
- In fact, we can use the same hardware for both…

Floating Point

- Representation for non-integral numbers
- Including very small and very large numbers

- Usually follows some "normalized" form of scientific notation
- Example: -2.34×10^{6} (ok) *vs.* -234×10^{4} (not ok)

- In binary, the form is: $\pm\ \mathbf{1.xxxxxxx_{(base\ 2)} \times 2^{yyyy}}$
- Types `float` and `double` in C/C++

- More in next lecture…

YOUR TO-DOs for the Week

- Readings!

- Work on Lab 4!

</LECTURE>