EEC 483 Computer Organization (Spring 2006)

Chapter 3. Arithmetic for Computers

Chansu Yu

Cleveland State University

Table of Contents

Software interface

- Ch.1, 4 Introduction & Performance
- Ch. 2 Instruction: Machine Language
  - 2.1 Introduction — 2.2 Operations (arithmetic, memory operations)
  - 2.3 Operands — 2.4 Representing instructions
  - 2.5 Logical operations — 2.6 Control flow operations
  - 2.7 Supporting procedures — 2.8 Beyond numbers
  - 2.9 MIPS addressing — 2.10 Starting a program
- Ch. 3 CPU Implementation: Arithmetic
  - 3.1 Introduction — 3.2 Signed and unsigned numbers
  - 3.3 Addition and subtraction — **3.4 Multiplication**
  - **3.5 Division** — 3.6 Floating point
  - Appendix B Constructing an Arithmetic Logic Unit (ALU)
- Ch. 5 CPU Implementation: All others
- Ch. 6-9 Advanced topics

Key parts of this course

Hardware interface

Cleveland State University

c.yu91@csuohio.edu

3.4 Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
     00100   (multiplicand)
   x 01011   (multiplier)
```

- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

Multiplication

- Paper and pencil example (unsigned):

```
Multiplicand        00101
Multiplier          01011
                    -----
                    00101
                   00101
                  00000
                 00101
                00000
               ---------
Product        000110111
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)

Multiplication

- Let's look at 2 versions based on grade school algorithm

```
      00101   (multiplicand)
   x  01011   (multiplier)
```

Product (T) =0
If multiplier bit0=1, T=T+m'cand
If multiplier bit1=1, T=T+m'cand<<1
If multiplier bit2=1, T=T+m'cand<<2
If multiplier bit3=1, T=T+m'cand<<3
If multiplier bit4=1, T=T+m'cand<<4

- Multiplier is either 0 or 1
  - If 0: goto the next bit
  - If 1: just add the multiplicand after shift

Product (T) =0
If multiplier bit0=1, T=T+m'cand
m'cand=m'cand<<1
If multiplier bit1=1, T=T+m'cand
m'cand=m'cand<<1
If multiplier bit2=1, T=T+m'cand
m'cand=m'cand<<1
If multiplier bit3=1, T=T+m'cand
m'cand=m'cand<<1
If multiplier bit4=1, T=T+m'cand

Shift right multiplier one bit and test the last bit

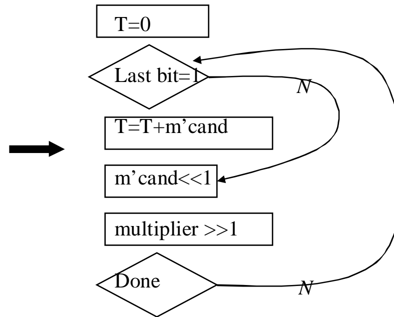
Shift left multiplicand one bit

Multiplication

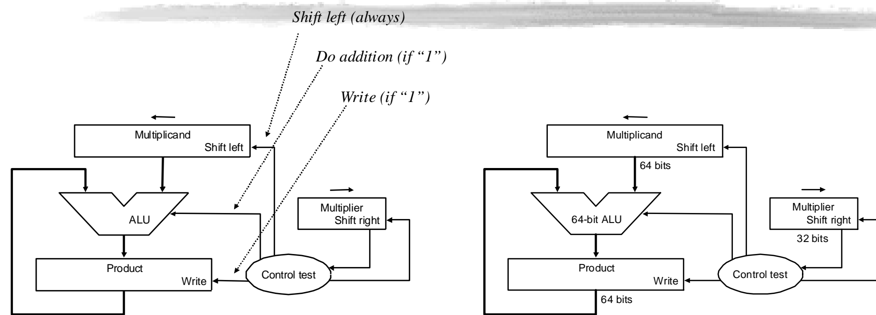
T=0

If multiplier bit0=1, T=T+m'cand
m'cand=m'cand<<1
multiplier>>1

… (repeat)



Multiplication: Implementation



Number of bits ???
- ***32-bit architecture***
- ***Multiplier: 32-bit***
- ***Multiplicand: 64-bit !!!***
- ***Product: 64-bit !!!***
- ***ALU: 64-bit ALU !!!***

It is actually a series of 32-bit add operations.
⇒Replace 64-bit ALU with 32-bit ALU
⇒Shift product (res) instead of shifting m'cand
⇒Next slice !!!

Cleveland State University
c.yu91@csuohio.edu

First version of the multiplication hardware is shown in Figure 3.5, page 177.
There are three registers (multiplicand, multiplier and product), a 64-bit ALU and a control logic. Consider the similar hardware for multiplying 5-bit numbers.
(i) Size of the three registers and the ALU
(ii) Fill the following table which shows the steps when multiplying 5x11
(multiplicand is 5 or 00101 and multiplier is 11 or 01011).

Iteration	Step	Multiplier (R)	Multiplicand (D)	Product (T)
0	Initial values	01011	00000 00101	00000 00000
1	1: T = T + D (since R0=1) 2: Shift left D 3: Shift right R	00101	00000 01010	00000 00101
2	1: T = T + D (since R0=1) 2: Shift left D 3: Shift right R	00010	00000 10100	00000 01111
3	1: no operation (since R0=0) 2: Shift left D 3: Shift right R	00001	00001 01000	
4	1: T = T + D (since R0=1) 2: Shift left D 3: Shift right R	00000	00010 10000	00001 10111
5	1: no operation (since R0=0) 2: Shift left D 3: Shift right R	00000	00101 00000	

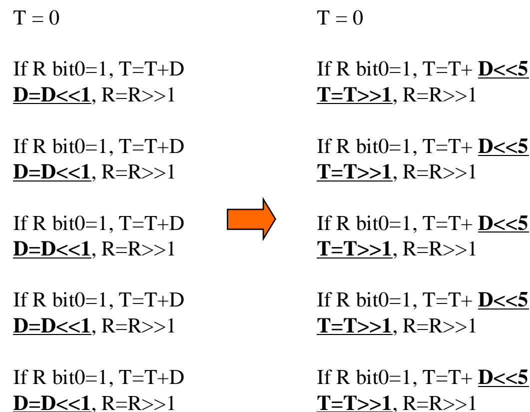
=> 55

Cleveland State University
c.yu91@csuohio.edu

How can we improve the design?

T =0				
If R bit0=1, T=T+D	T0=D<<5			T=T0
	T0>>1			T>>1
If R bit1=1, T=T+D<<1		T1=D<<5		T=T+T1
	T0>>1	T1>>1		T>>1
If R bit2=1, T=T+D<<2			T2=D<<5	T=T+T2
	T0>>1	T1>>1	T2>>1	T>>1
If R bit3=1, T=T+D<<3				…
	T0>>1	T1>>1	T2>>1	
If R bit4=1, T=T+D<<4				
	T0>>1	T1>>1	T2>>1	

How can we improve the design?



T = 0	T = 0
If R bit0=1, T=T+D **D=D<<1**, R=R>>1	If R bit0=1, T=T+ **D<<5** **T=T>>1**, R=R>>1
If R bit0=1, T=T+D **D=D<<1**, R=R>>1	If R bit0=1, T=T+ **D<<5** **T=T>>1**, R=R>>1
If R bit0=1, T=T+D **D=D<<1**, R=R>>1	If R bit0=1, T=T+ **D<<5** **T=T>>1**, R=R>>1
If R bit0=1, T=T+D **D=D<<1**, R=R>>1	If R bit0=1, T=T+ **D<<5** **T=T>>1**, R=R>>1
If R bit0=1, T=T+D **D=D<<1**, R=R>>1	If R bit0=1, T=T+ **D<<5** **T=T>>1**, R=R>>1

How can we improve the design?

Iteration	Step	Multiplier (R)	**Multiplicand (D)**	Product (T)
0	Initial values	01011	**00101**	00000 00000
1	1: T = T + D<<5 (since R0=1) 2: Shift right T 3: Shift right R	00101	**00101**	00101 00000 00010 10000
2	1: T = T + D<<5 (since R0=1) 2: Shift right T 3: Shift right R	00010	**00101**	00111 10000 00011 11000
3	1: no operation (since R0=0) 2: Shift right T 3: Shift right R	00001	**00101**	00001 11100
4	1: T = T + D<<5 (since R0=1) 2: Shift right T 3: Shift right R	00000	**00101**	00110 11100 00011 01110
5	1: no operation (since R0=0) 2: Shift right T 3: Shift right R	00000	**00101**	00001 10111

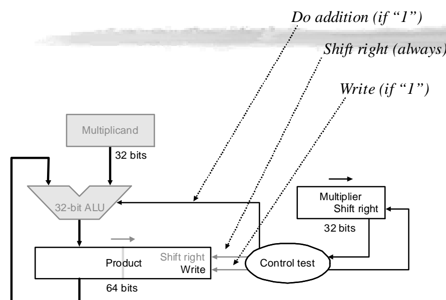
=> 55

Implementation

Do addition (if "1")

Shift right (always)

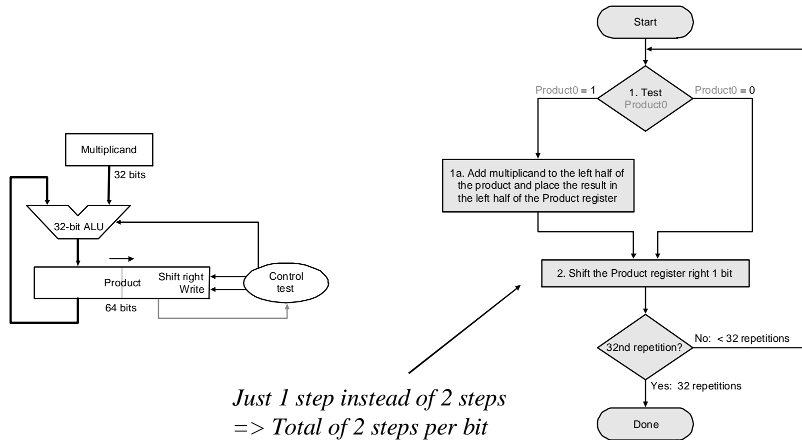
Write (if "1")



Number of bits ???
- ***32-bit architecture***
- ***Multiplier: 32-bit***
- ***Multiplicand: 64-bit => 32-bit***
- ***Product: 64-bit !!!***
- ***ALU: 64-bit ALU => 32-bit ALU***

Product register wastes space that exactly matches size of multiplier
Multiplier space can be saved.
⇒Combine Multiplier register and Product register
(Multiplier register stored in lower part of Product register
will be thrown away one bit at a time)
⇒Next slice !!!

Final Version



More on Multiplication: Booth Algorithm

- Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
  - can handle multiple bits at a time
- Want to increase the number of '0's so that we can bypass the add operation
  - Based on 'redundant' codes $\{1, 0, \bar{1}\}$
  - What does it mean by 'redundant' ?
- Example
  - 0 1 1 1 = 1+2+4 = 7
  - 1 0 0 $\bar{1}$ = -1 + 8 = 7

More on Multiplication: Booth Algorithm

x = 0111 1000 0111 1111 => 11 additions

x' = 1000 $\bar{1}$000 1000 000$\bar{1}$ => 2 additions & 2 subs

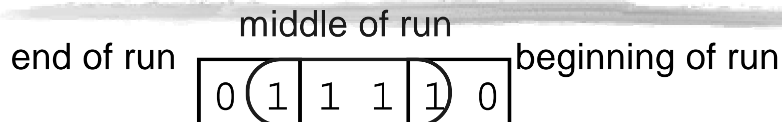
= 4 additions

(2 m'cand: +m & -m)

01111 = 10000 -1 *01111111 = 10000000-1*

0 (0) => 0
0 (1) => 1
1 (0) => -1
1 (1) => 0

Booth's Algorithm



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

```
   -1
+ 10000
  01111
```

Booths Example (2 x 7)

Operation	Multiplicand	Product	next?
0. initial value	**0010**	**0000 0111 0**	**10 -> sub**
1a. P = P - m	1110	+ 1110 1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+ 0010 0001 1100 1	shift
4b.	0010	0000 1110 0	done

Cleveland State University

17

c.yu91@csuohio.edu

Booths Example (2 x −3)

Operation	Multiplicand	Product	next?
0. initial value	**0010**	**0000 1101 0**	**10 -> sub**
1a. P = P - m	1110	+ 1110 1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1 + 0010	01 -> add
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0 + 1110	10 -> sub
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Cleveland State University

18

c.yu91@csuohio.edu

Multiply in MIPS

Instruction	Example	Meaning	Comments
❑ multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
❑ multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
❑ Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
❑ Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

3.5 Divide

- $2 / $3 = quotient ... remainder
  - Is quotient 32-bit or 16-bit?
  - Is remainder 32-bit or 16-bit?
- Example
  - $2=0111 1111 1111
  - $3=0000 0000 0000
  - Quotient=0111 1111 1111
  - Quotient must be 32-bit !
  - $2=0111 1111 1111
  - $3=0100 0000 0000
  - Quotient=1, Remainder=0011 1111 1111
  - Remainder must be 32-bit !
- n-bit / n-bit
  - n-bit quotient, n-bit remainder
  - More hardware (wasting)
- n-bit / n/2-bit
  - n/2-bit quotient, n/2-bit remainder
  - Overflow
- Two basic approaches
  - Restoring : conventional
  - Non-restoring

Division in MIPS

Instruction	Example	Meaning	Comments
❑ divide	div \$2,\$3	Lo = \$2 ÷ \$3,	Lo = quotient, Hi = remainder Hi = \$2 mod \$3
❑ divide unsigned	divu \$2,\$3	Lo = \$2 ÷ \$3,	Unsigned quotient & remainder Hi = \$2 mod \$3
❑ Move from Hi	mfhi \$1	\$1 = Hi	Used to get copy of Hi
❑ Move from Lo	mflo \$1	\$1 = Lo	Used to get copy of Lo

Implementation: Paper & Pencil

```
                 8          Quotient
Divisor   23 | 2057         Dividend
              -184
               217

                1           Quotient
Divisor 00023|02057         Dividend
          -000230000
          -000227943
              02057
```

We know where to start
But computer do not know and assume both dividend and divisor are 32-digit numbers

And start from the first possible digit
If the result is negative, move on to the next digit while recovering the dividend to the original value

Thus, start with
00023 0000 and 0000 02057
and shift right divisor each time

One nice thing with binary computation is that the quotient bit can be 1 or 0

If the subtraction gives positive, Qi=1
Otherwise, Qi=0 and ***restore*** *the dividend*

Implementation: Paper & Pencil

```
                  1001   Quotient
Divisor 1000  | 1001010  Dividend
               -1000
                  10
                  101
                  1010
                 -1000
                    10   Remainder (or Modulo result)
```

We know 10 is less than 1000. But ALU does not know until it subtracts 10-1000 and gets the negative result.
If it is negative, it needs to restore the dividend to the original value by adding 1000, i.e., (10-1000)+1000 = 10

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder

=> | Dividend | = | Quotient | + | Divisor |

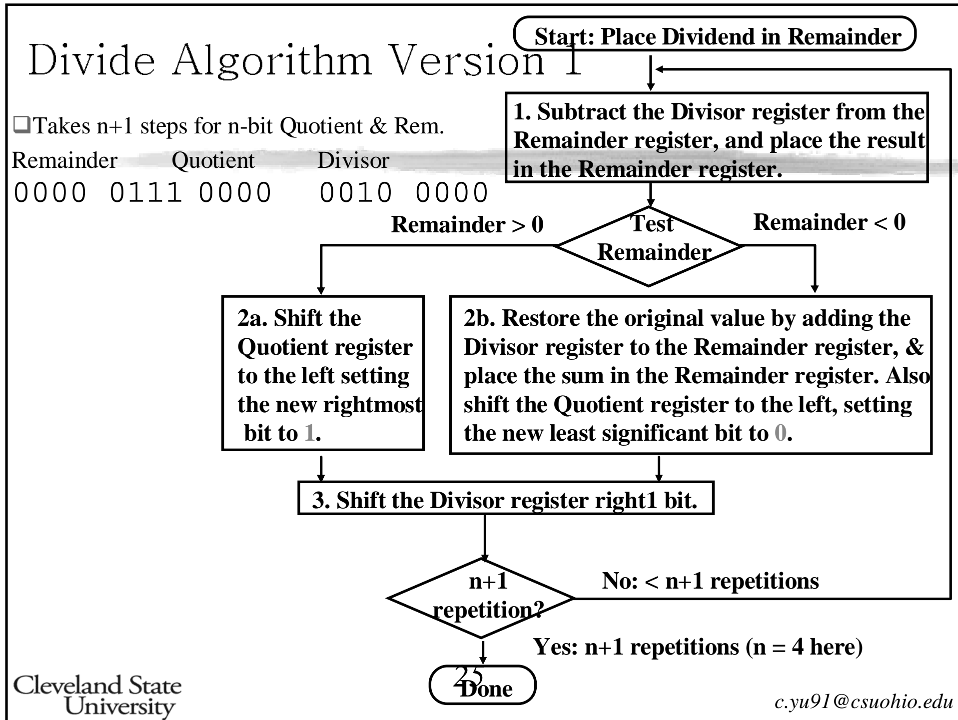
3 versions of divide, successive refinement

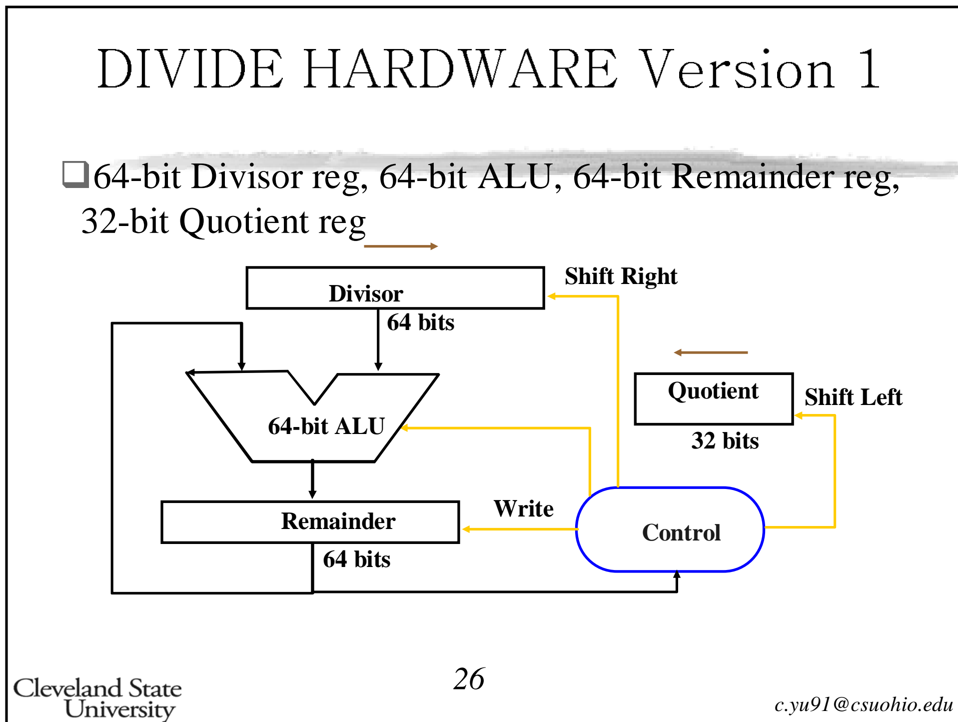
Dividend: 01011 (11), Divisor: 00101 (5)

Iteration	Step	Quotient (Q)	Divisor (D)	Remainder (R)
0	Initial values	00000	00101 **00000**	**00000** 01011
1	1: R = R - D 2: R<0 => Restore R, Shift left Q, Q0=0 3: Shift right D	0000**0**	00010 10000	11011 01011 00000 01011
2	1: R = R - D 2: R<0 => Restore R, Shift left Q, Q0=0 3: Shift right D	0000**0**	00001 01000	11101 11011 00000 01011
3	1: R = R - D 2: R<0 => Restore R, Shift left Q, Q0=0 3: Shift right D	0000**0**	00000 10100	11111 00011 00000 01011
4	1: R = R - D 2: R<0 => Restore R, Shift left Q, Q0=0 3: Shift right D	0000**0**	00000 01010	11111 10111 00000 01011
5	1: R = R - D 2: R>=0 => Shift left Q, Q0=1 3: Shift right D	0000**1**	00000 00101	00000 00001
6	1: R = R - D 2: R<0 => Restore R, Shift left Q, Q0=0 3: Shift right D	0001**0**	00000 00010	11111 11100 00000 00001

dividend — Restored (iterations 1–4), Not restored (iteration 5), Restored (iteration 6)

Quotient = 2 **remainder = 1**


Divide Algorithm Version 1
Takes n+1 steps for n-bit Quotient & Rem.
Remainder Quotient Divisor
0000 0111 0000 0010 0000
Start: Place Dividend in Remainder
1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.
Test Remainder
Remainder > 0
Remainder < 0
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, & 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 right1 bit.
n+1 repetition?
No: < n+1 repetitions
Yes: n+1 repetitions (n = 4 here)
Done
25
Cleveland State University
c.yu91@csuohio.edu

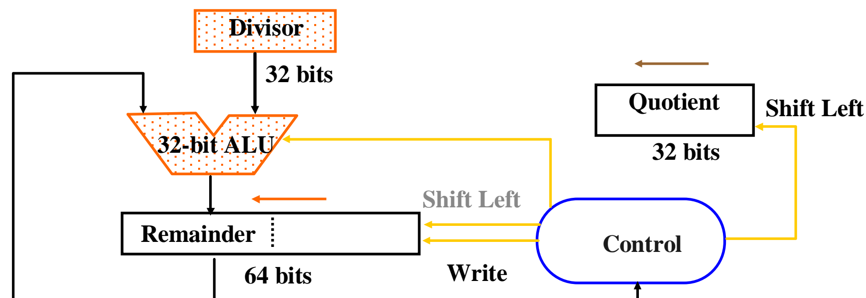

DIVIDE HARDWARE Version 1
64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg
Divisor
64 bits
Shift Right
64-bit ALU
Quotient
32 bits
Shift Left
Remainder
64 bits
Write
Control
26
Cleveland State University
c.yu91@csuohio.edu

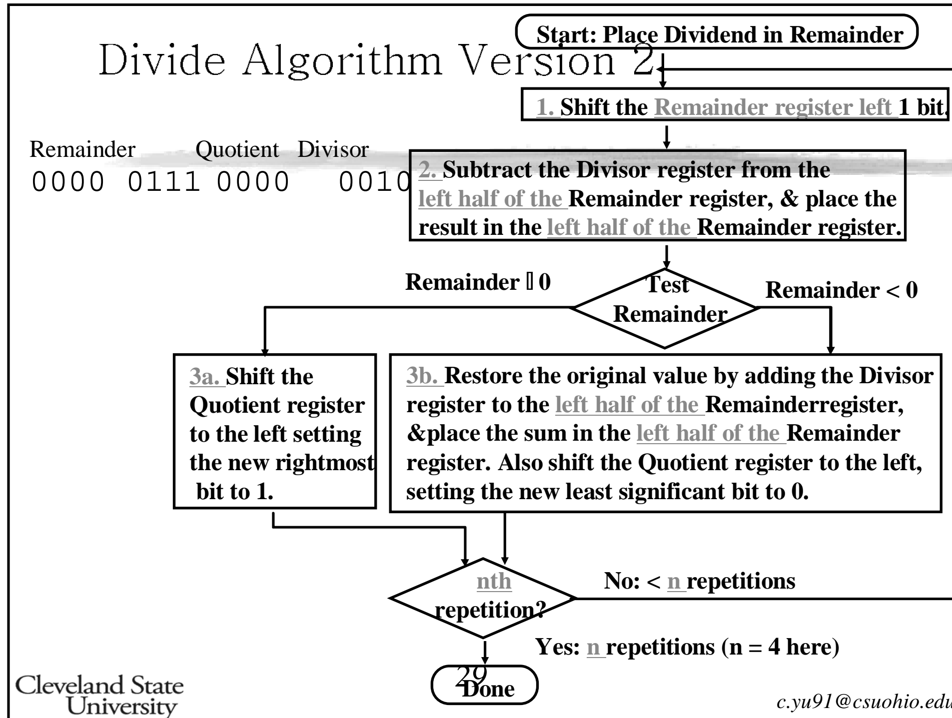
Observations on Divide Version 1

- 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?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save one iteration

DIVIDE HARDWARE Version 2

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





Observations on Divide Version 2

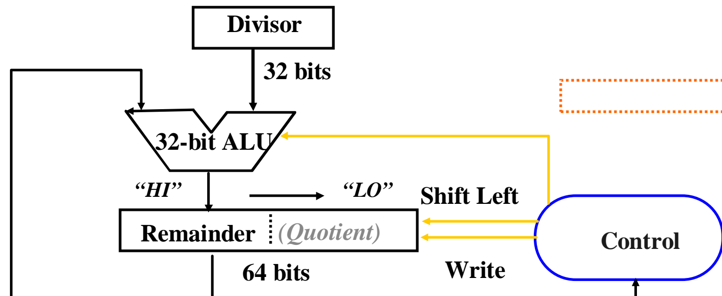
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

Cleveland State University

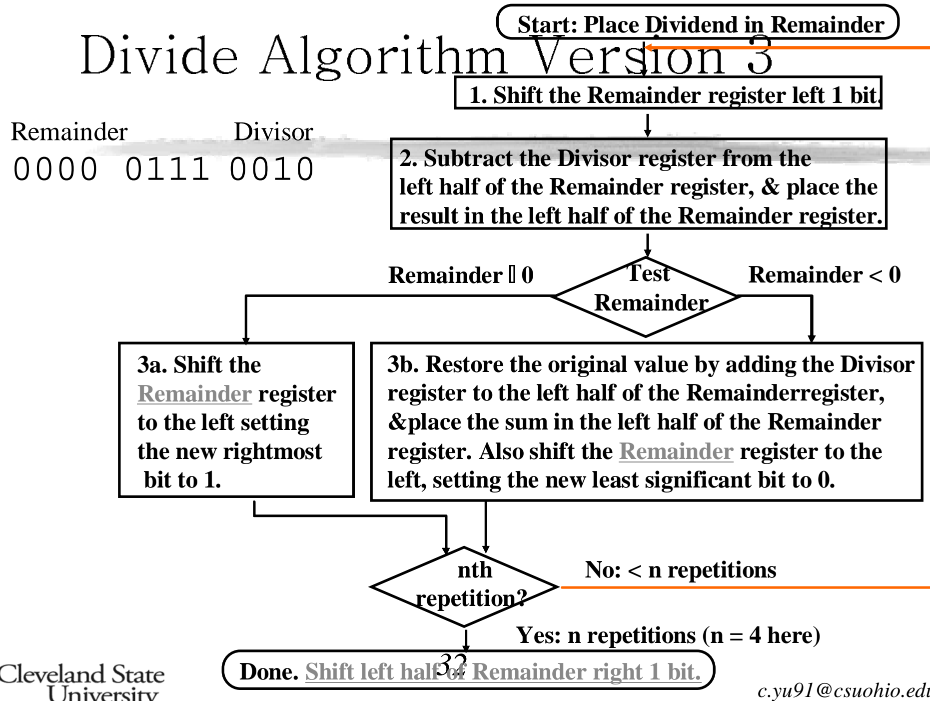
c.yu91@csuohio.edu

Divide Hardware: Final Version

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



Divide Algorithm Version 3



Observations on Divide Version 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 Divides: 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 ÷ 2 = –3, remainder = –1
- Possible for quotient to be too large: if divide 64-bit interger by 1, quotient is 64 bits ("called saturation")

Cleveland State University
c.yu91@csuohio.edu