Midterm Exam

- The midterm exam is on Wednesday March 16, 2016 at 10:00 a.m. in class.
- The midterm will cover Chapters 1, 2, and 3. Please review the lecture notes, book, assignments, and quizzes.
- The exam will be online (cougar courses) with multiple choice questions. You will use the safe browser exam to access the exam.
- It is closed book, closed note, only MIPS sheet allowed. Please make sure you bring a copy of the MIPS sheet.
- You may use calculators.

Chapter 1
Performance

Defining Performance

- For some program running on Computer X,

$$Performance_X = 1 / Execution\ or\ CPU\ time_X$$

- “X is ***n*** times faster than Y” **Pay attention x/y not y/x**

$$n = Performance_X / Performance_Y$$
$$= Execution\ or\ CPU\ time_Y / Execution\ or\ CPU\ time_X$$

- CPU time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$
$$= \underbrace{IC \times CPI}_{\text{Clock Cycles}} \times CT$$

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$Clock\ Cycles = \sum_{i=1}^{n} (CPI_i \times Instruction\ Count_i)$$

Instruction Count for i^{th} Class

Average cycles per instruction for i^{th} class

- Weighted average CPI

$$CPI = \frac{Clock\ Cycles}{Instruction\ Count} = \sum_{i=1}^{n} \left(CPI_i \times \underbrace{\frac{Instruction\ Count_i}{Instruction\ Count}}_{\text{Relative frequency}} \right)$$

Pitfalls

- MIPS: Millions of Instructions Per Second

$$MIPS = \frac{Instruction\ count}{Execution\ time \times 10^6} = \frac{Clock\ rate}{CPI \times 10^6}$$

  - Does not account for capability/complexity of instructions
    - Can not compare computers with different ISA
    - Can not compare programs on same computer

- Amdahl's law: Expecting improvement of one aspect of a computer to increase overall performance by an amount proportional to size of improvement

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

Example

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

Sequence 1:

IC = 5

$$Clock\ Cycles = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

$$= 2\times1 + 1\times2 + 2\times3 = 10$$

Must have HIGHER CPI

Avg. CPI = Clock Cycles/IC

= 10/5 = 2.0

Sequence 2:

IC = 6

Clock Cycles = 4×1 + 1×2 + 1×3 = 9

Avg. CPI = 9/6 = 1.5

Example

- Consider a program on a computer of two classes of instructions
  - A: CPI = 2, frequency = 40%
  - B: CPI = 4, frequency = 60%

- What's the CPI of this machine?

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times frequency_i \right) = 2 \times 40\% + 4 \times 60\% = 3.2$$

- If CPI of the instruction class B is reduced to 3 without changing clock rate, how much faster is the new machine? What's its CPI?

  CPI = 2 × 40% + 3 × 60% = 2.6.

  Because both have the same number of instructions and clock rate, the ratio of execution time is the ratio of the CPI:

  Speedup = 3.2 / 2.6 = 1.23 times faster

Example

- A: CPI = 2, frequency = 40%
- B: CPI = 4, frequency = 60%
- Avg CPI = 3.2

- If we reduce number of class A instructions to 50% of the original for the program (and CPI of class B instructions reduced to 3), how much faster is the new machine? What's its CPI?

Assume originally there are IC_1 instructions, then Clock Cycles = $3.2 \times IC_1$

The number of cycles after reducing CPI of B and instructions of A is:

$$\mathit{Clock\ Cycles}_{new} = \sum_{i=1}^{n}(CPI_i \times IC_i) = 2 \times 20\% \times \mathrm{IC}_1 + 3 \times 60\% \times \mathrm{IC}_1 = 2.2 \times \mathrm{IC}_1$$

Speedup = 3.2 / 2.2 = 1.45

$\mathrm{CPI}_{new} = \text{Clock Cycles}_{new} / \mathrm{IC}_{new} = 2.2 \times \mathrm{IC}_1 / 0.8 \times \mathrm{IC}_1 = 2.75$

Amdahl's Law Example

- A program takes 10 seconds to run on the current computer. The program spends 40% of its time on floating-point operations, 40% on integer operations, and 20% on I/O operations.
- If you can make the floating-point operations 2 times faster, what is the overall speedup of the program?

T_{new}= 8 sec → 1.25 times speedup

- If you want the whole program to run 2 times faster, how much do you need to improve the speed of integer operations?

Not possible

Chapter 2
Instructions: Language of the Computer

Unsigned Binary Integers

- Some examples:
  - 1100_2

    12
  - $0001\ 1011_2$

    27
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100_2$

    12
  - 7_d

    0111
  - 23_d

    1 0111

2s Complement Signed Integers

- Some examples:
  - 0

    0000 0000 … 0000
  - –1

    1111 1111 … 1111
  - -5

    1111 1111 … 1011
  - 8

    0000 0000 … 1000

Hexadecimal Examples

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Convert from Hexadecimal to binary: e c a 8 6 4 2 0

  1110 1100 1010 1000 0110 0100 0010 0000

- Convert from binary to Hexadecimal :

  1111 1011 1000 1010 1010 0010 0011 0001

  f b 8 a a 2 3 1

MIPS Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	function arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary
. . .		
15	t7	

16	s0	saved temporary
. . .		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

Instruction Types and Formats

- Data operation
  - Arithmetic, Logical
- Data transfer
  - Load, Store
- Instruction sequencing
  - Branch (conditional), Jump (unconditional)

R	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**
I	**op**	**rs**	**rt**	**16 bit address**		
J	**op**	**26 bit address**				

R-format Example

add $t0, $s1, $s2

(Pay attention to reg order, names!!)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Shift Operations

- Shift left logical or right logical using **`sll`** and **`srl`**
  - **`sll $t2, $s0, 2`** `# $t2 = $s0 << 2`
  - **`srl $t2, $s0, 2`** `# $t2 = $s0 >> 2`

  - Shift and fill with 0 bits
  - shamt: how many positions to shift (here 2)
  - `sll` by i bits multiplies by 2^i
  - `srl` by i bits divides by 2^i

- Example: **`sll $t2, $s0, 3`**

$S0: 0110 0000 0000 0000 1100 1000 0000 1111
$t2: 0000 0000 0000 0110 0100 0000 0111 1000

More Logical Operations

- Logical Operations
  - **AND** bit-wise AND between registers
    - `and $t1, $s0, $s1`
  - **OR** bit-wise OR between registers
    - `or $t1, $s0, $s1`
  - **NOR** Bit-wise NOR between registers
    - `nor $t1, $s0, $s1`
    - `nor $t1, $t0, $0        # $t1 = NOT($t0)`
  - Immediate modes
    - `andi and ori` (**Zero Extend**)

Logical Operations Example

- How can we isolate the byte in red?

0000 0010 1000 1100 0000 1101 1100 0000

1. Using AND

   with 0000 0000 0000 0000 0000 1111 0000 0000

2. Using a shift left followed by shift right

   sll 1101 1100 0000 0000 0000 0000 0000 0000

   srl 0000 0000 0000 0000 0000 0000 0000 1101

Loading Larger Constants?

- 2 instructions to load a 32 bit constant into a register: **can not do it in one instruction!**
  - **1010 1010 1010 1010 0010 1010 1010 1010**
  - Load upper immediate: `lui $t0, 1010101010101010`

    filled with zeros

1010101010101010	0000000000000000

  - Get the lower order bits right: `ori $t0, $t0, 0010101010101010`

	1010101010101010	0000000000000000
ori	0000000000000000	0010101010101010
	1010101010101010	0010101010101010

Memory Instructions

- Load word
  - From memory location to register
  - `lw $t1, offset($t0)`
- Store word
  - From register to memory location
  - Has destination last
  - `sw $t1, offset($t0)`
  - **NEED to COMPUTE ADDRESS in separate instruction!**

C code:

```
A[8] = h + A[8];
```

- h in \$s2
- base address of word array A in \$s3

MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

Instruction Sequence Operations

- Conditional Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially

  `beq rs, rt, L1`
  - Go to the statement labeled L1 if the value in rs equals the value in rt

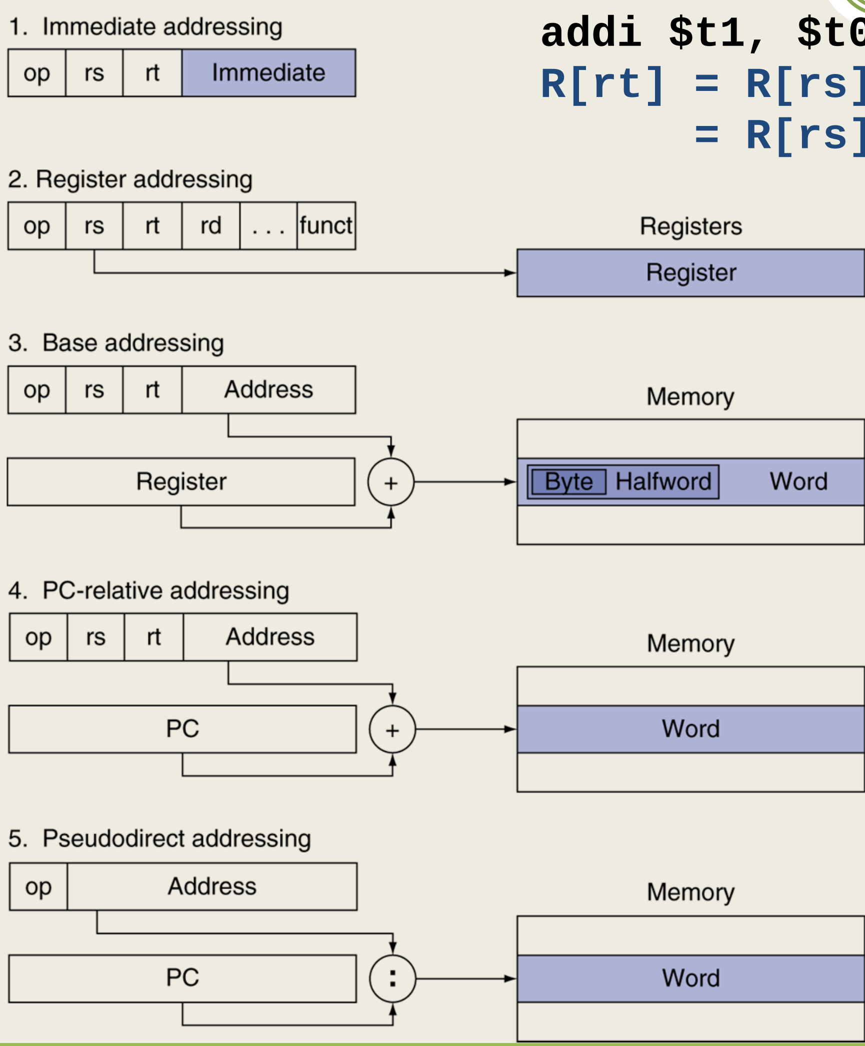
  `bne rs, rt, L1`
  - Go to instruction labeled L1 if the value in rs *is not* equal the value in rt

- Unconditional Operations

  `j L1`
  - Unconditional jump to instruction labeled L1

  `jr $t0`
  - "jump register". Jump to the instruction specified in register $t0

MIPS Addressing Mode Summary




```
addi $t1, $t0, 20
R[rt] = R[rs] + SignExtImmediate
      = R[rs] + {16{Immediate[15]}, Immediate}
```

```
add $s1, $s0, $t0
R[rd] = R[rs] + R[rt]
```

```
lw $s0, 40($t0)
R[rt] = M[R[rs] + SignExtAddress]
```

```
beq $t0, $t1, L1
If (R[rs] == R[rt])
  PC = PC+4 +
    {14{Address[15]}, Address, 2'b0}
```

```
j L2
PC = {PC+4[31:28], Address, 2'b0}
```

Chapter 3
Arithmetic for Computers

Integer Addition

- Example: 7 + 6

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

Overflow if result out of range

Adding +ve and –ve operands: No overflow

Adding two +ve operands: Overflow if result sign is 1

Adding two –ve operands: Overflow if result sign is 0

2's Complement Integer Subtraction

- Add negation (2s complement) of second operand

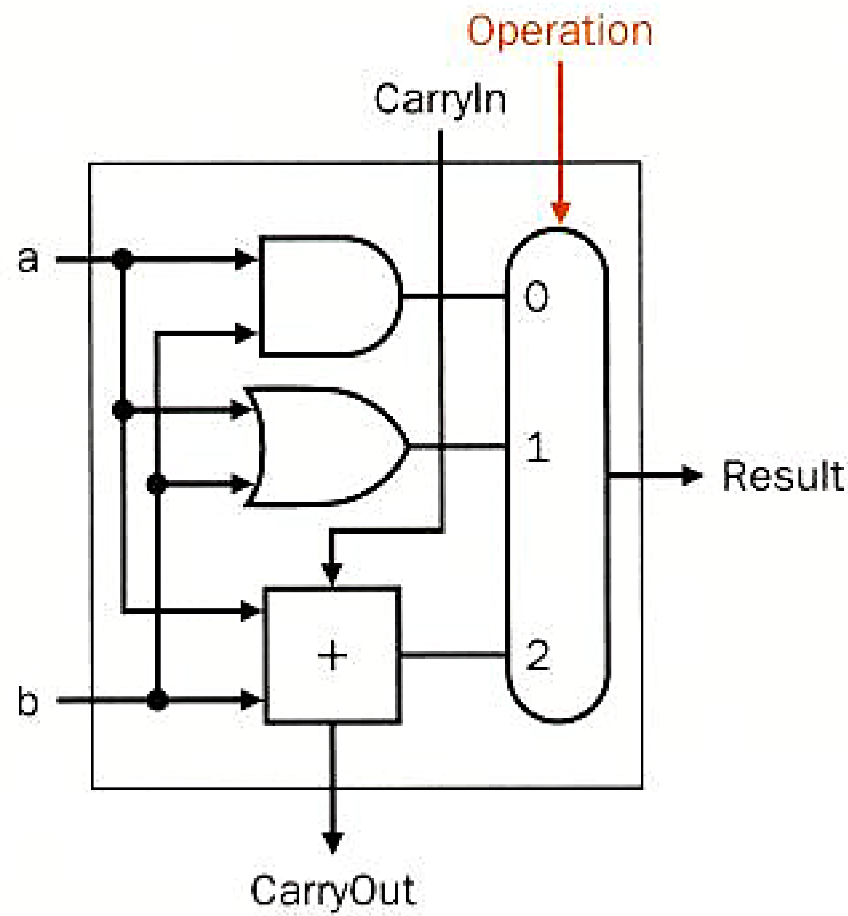
- Example: 7 – 6 = 7 + (–6)

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

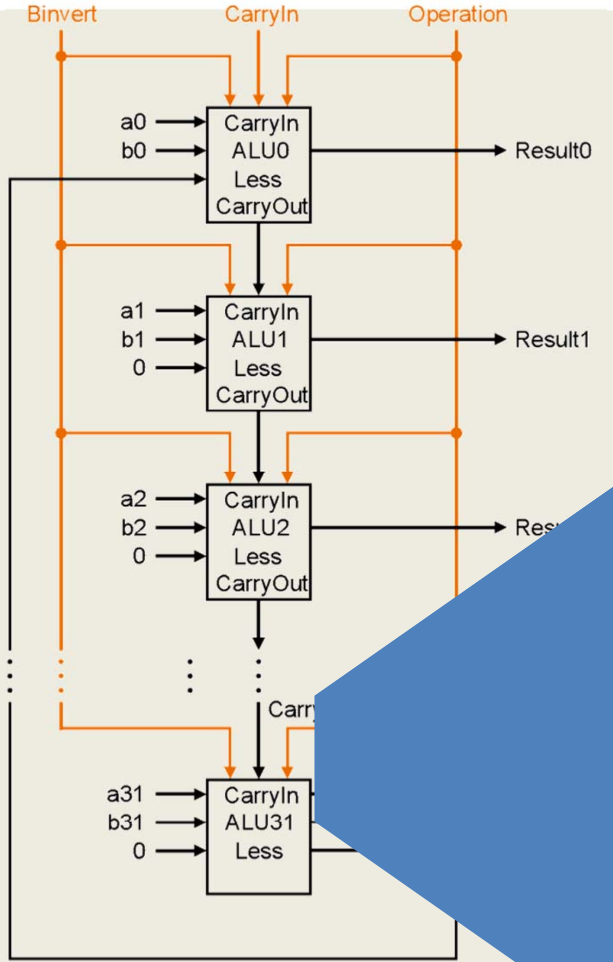
- Overflow if result out of range

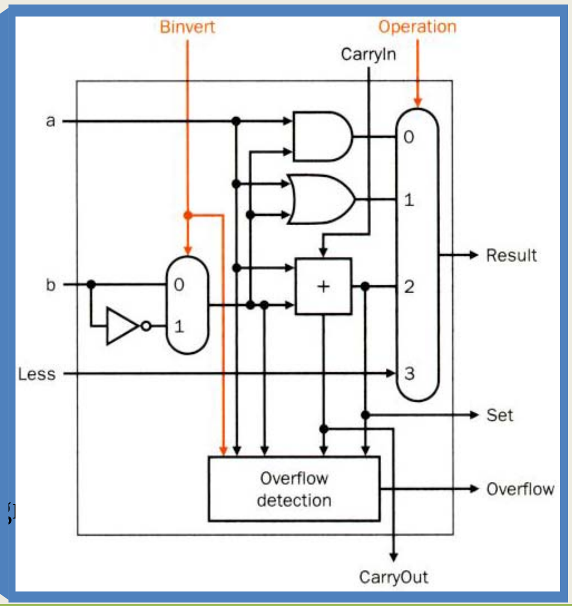
1 bit ALU

- ALU so far
  - AND
  - OR
  - ADD



Full ALU





what signals accomplish ADD?

	Binvert	CIn	Oper
A	1	0	2
B	0	1	2
C	1	1	2
D	0	0	2
E	More than One Above		

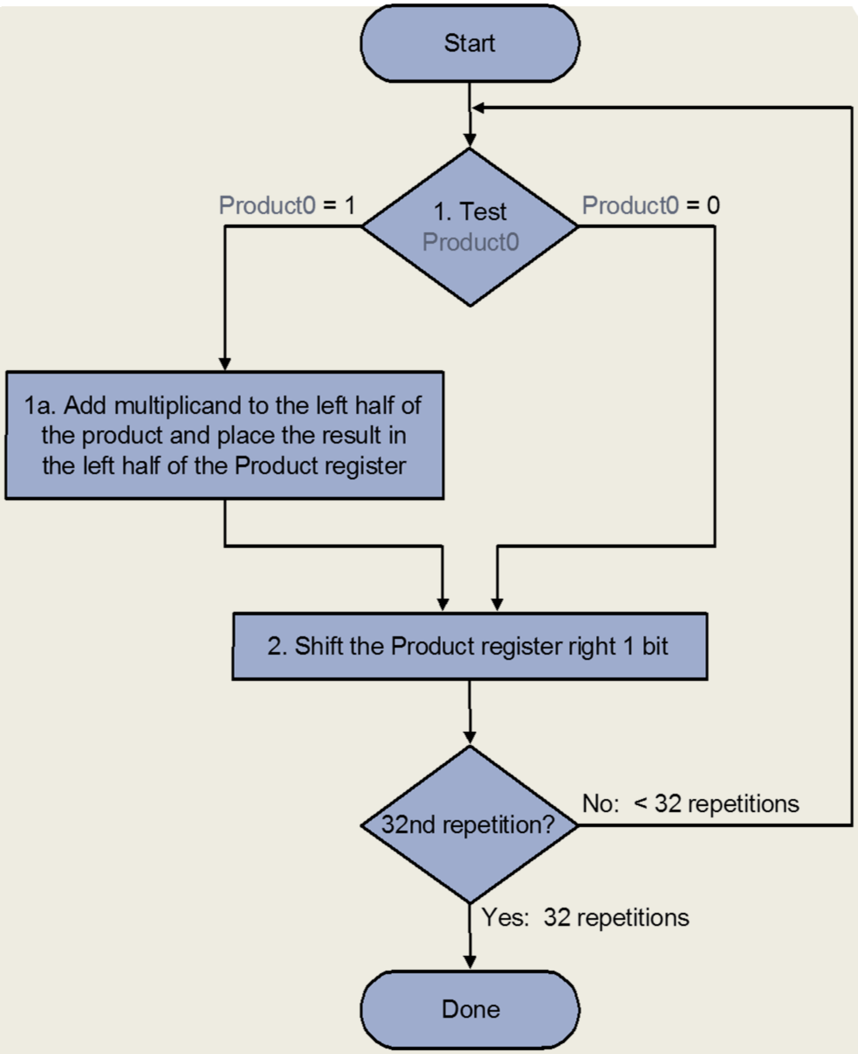
Multiplication

```
    1000    multiplicand
 ×  1001    multiplier
 -------
    1000
   0000
  0000
 1000
 -------
 1001000    product
```

64 bit product
32 bit multiplier and multiplicand

- Start with product=0 and accumulate partial products
- Look at current bit position of multiplier
  - If multiplier is 1
    - Add multiplicand to product
  - Else add 0
  - Shift multiplicand left 1 bit or shift product right 1 bit

Final Version

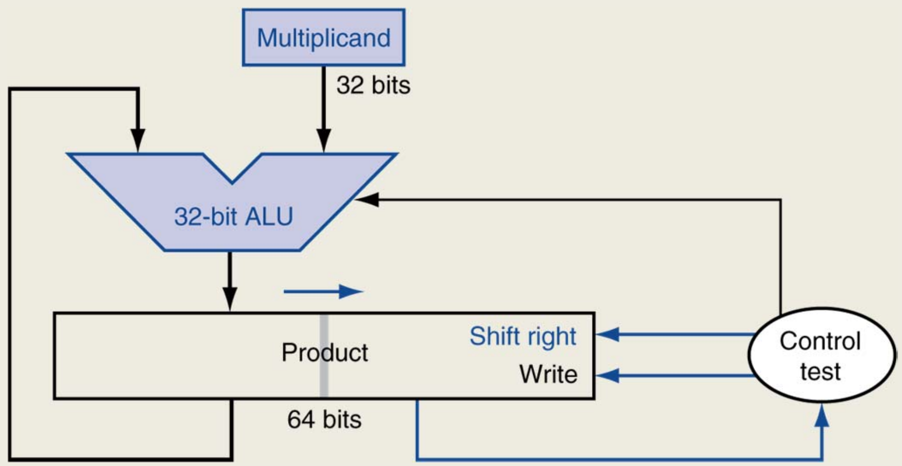


- Initially, product is '0'
- Initial lower 32 bits of product register shifted out by the end of multiplication

→ Use these 32 bits for multiplier

→ At every iteration, check lowest bit of product register (multiplier bit)

→ Either add & shift or shift only



Example

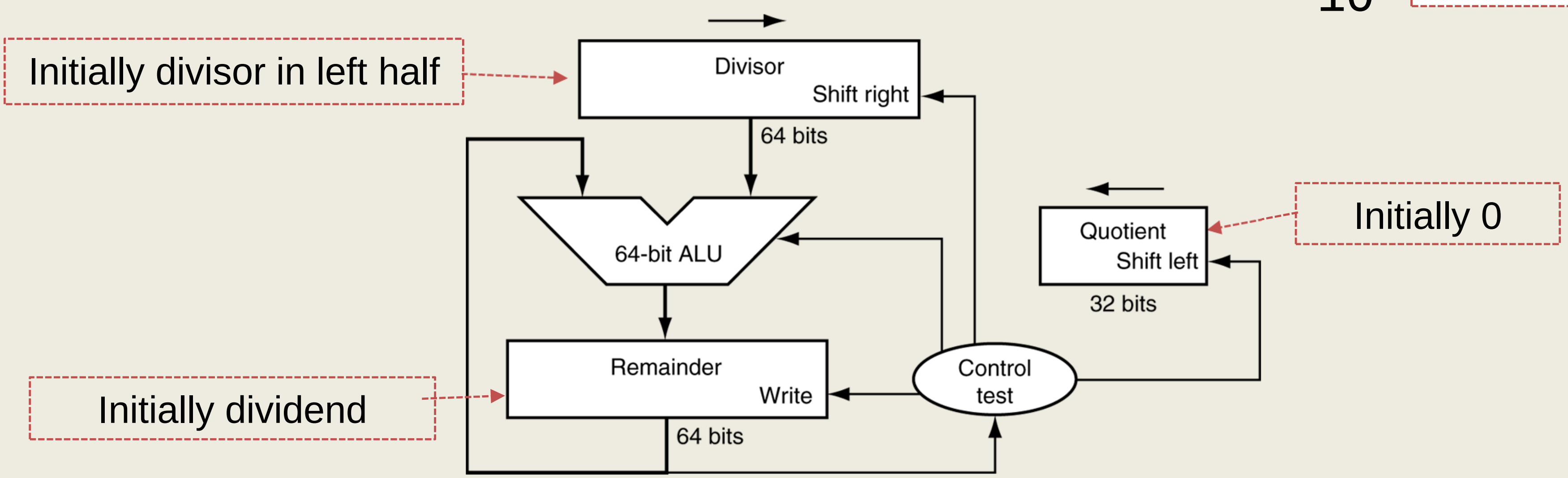
- 2 × 3 or 0010 × 0011

Iteration	Step	Multiplicand	Product
0	Initial Values	0010	0000 0011
1	1a: 1 =>Prod=Prod+Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 =>Prod=Prod+Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 =>no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 =>no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

Restoring Division

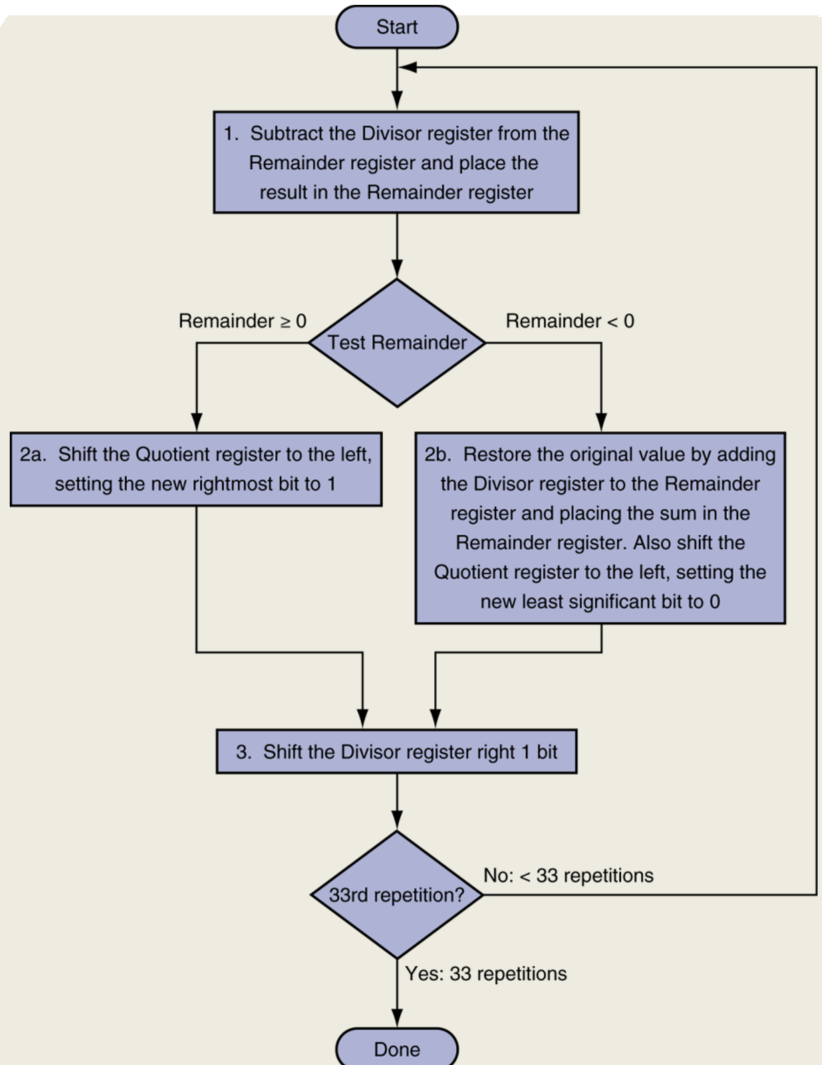
- Subtract divisor from dividend
- If remainder goes < 0
  - Add divisor back and put 0 bit in quotient
- Else
  - 1 bit in quotient
- Shift divisor right 1 bit
  - Align divisor relative to dividend for next iteration

```
            1001      quotient
divisor
  1000 )1001010
        -1000         dividend
         0010
          0101
           1010
          -1000
             10       remainder
```



Division Version 1

- Each step
  - Subtract divisor from Dividend (stored in remainder register)
  - Depending on remainder
    - Leave or Restore (reverse subtraction)
    - Write '1' or '0' to quotient
  - Shift Divisor right
    - Align divisor relative to dividend for next iteration



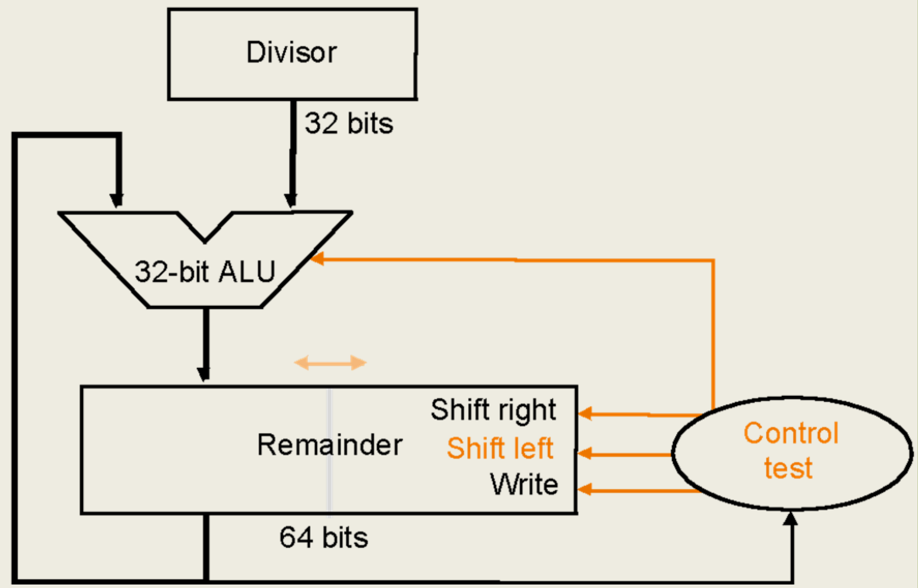
Example

- 7/2 or 0000 0111 / 0000 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

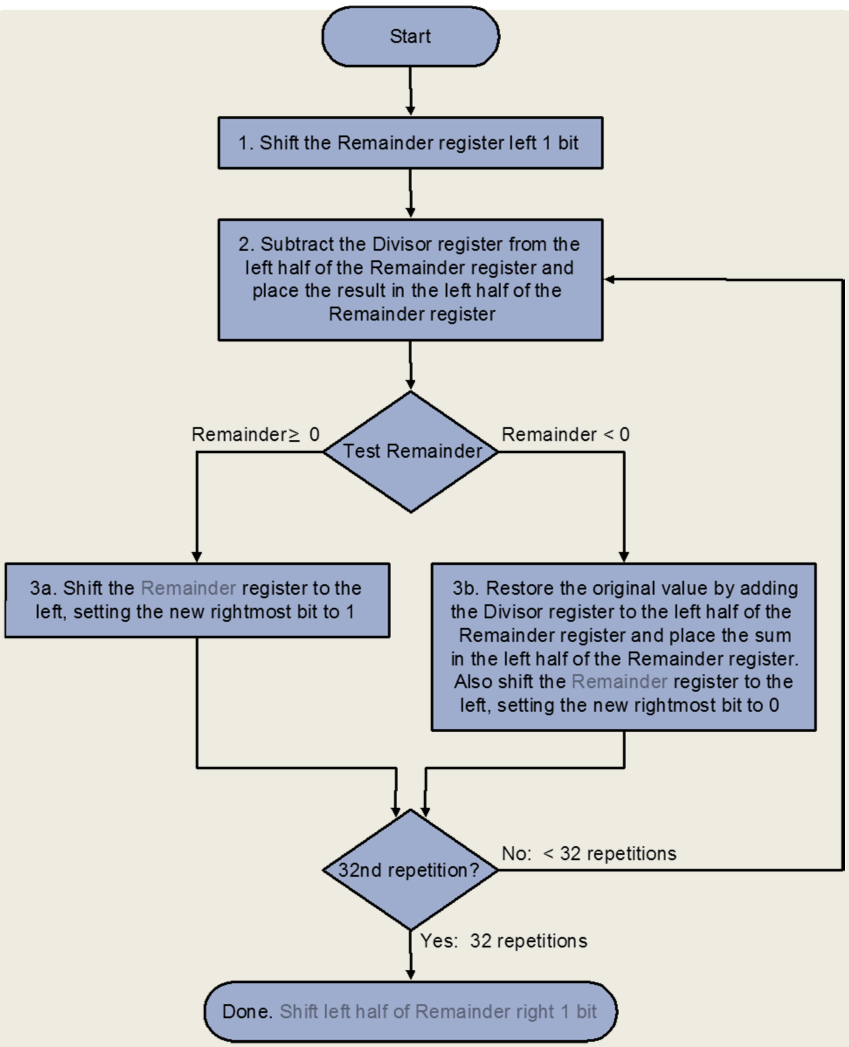
Optimized Divider

- Switch order to shift first and then subtract can save 1 iteration
  - Reduction of divisor and ALU width by half
- Remainder originally zero so keep quotient in remainder register
  - No register for quotient



Looks a lot like multiplier: Same hardware can be used for both!

Improved Divide Algorithm



- Shift the remainder left
- Only one shift per loop
  - The remainder will be shifted left one time
- The final correction step shifts back only the remainder in the left half of the register

Example

- 7/2 or 0000 0111 / 0010

iteration	step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left	0010	0000 1110
1	2: rem = rem - div	0010	1110 1110
	3b: restore, sll R, R0 = 0	0010	0001 1100
2	2. rem = rem - div	0010	1111 1100
	3b: restore, sll R, R0= 0	0010	0011 1000
3	2. rem = rem - div	0010	0001 1000
	3a: leave, sll R, R0= 1	0010	0011 0001
4	2. rem = rem - div	0010	0001 0001
	3a. leave, sll R, R0=1	0010	0010 0011
	Shift left half of rem right 1	0010	0001 0011

IEEE Floating-Point Format

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

	S	Exponent	Fraction
Single Precision:	1bit	8 bits	23 bits
Double Precision:	1bit	11 bits	52 bits

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Fraction or mantissa
- Exponent
  - Actual exponent + Bias (Bias = 127 for single; 1023 for double)

Example 1

- Show the binary representation of -0.75 in IEEE single precision format
  - Binary representation: - 0.11
  - Normalized representation: $- 1.1 \bullet 2^{-1}$

- Floating point
  - $(-1)^{\text{sign}} \bullet (1 + \text{fraction}) \bullet 2^{\text{exponent} - \text{bias}}$

- Sign bit = 1
- Significand = 1 + .1000 ....
- Exponent = (-1 + 127) = 126

→ 1 01111110 100 0000 0000 0000 0000 0000

Example 3

- What is the value of following IEEE binary floating point number?

0 1000 0000 011 0000 0000 0000 0000 0000

  - S = 0
  - Exponent = 10000000_2 = 128
  - Fraction = $01100...00_2$

$$x = (-1)^0 \times (1 + 011_2) \times 2^{(128 - 127)}$$
$$= (1) \times 1.375 \times 2^1$$
$$= 2.75$$

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points
   - Shift number with smaller exponent
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

   Shift right n times → x 2^n
   Shift left n times → x 2^{-n}

2. Add significands
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)

1. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
4. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change)
5. Determine sign: +ve × –ve ⇒ –ve
   - $-1.110_2 \times 2^{-3} = -0.21875$