Tik-61.123 Computer Architecture
Lecture 4: Shifters, Divide, Floating Point

MIPS logical instructions

Instruction	Example	Meaning	Comment
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 Â $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

Shifters

Two kinds:

logical-- value shifted in is always "0"

"0" ▶ msb lsb ◀ "0"

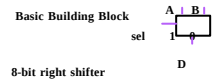
arithmetic-- on right shifts, sign extend

▶ msb lsb ◀ "0"

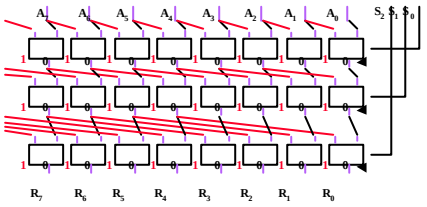
Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Review: Combinational Shifter from MUXes

Basic Building Block

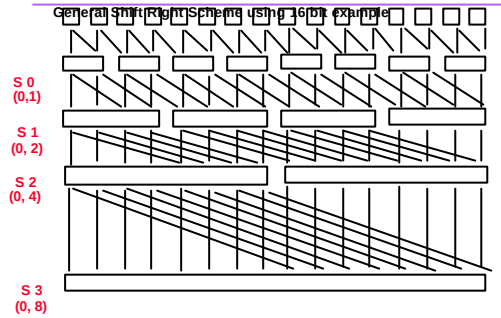


8-bit right shifter



- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes ?

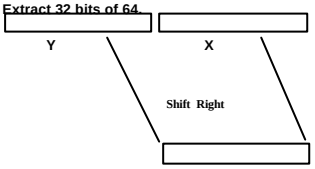
General Shift Right Scheme using 16 bit example



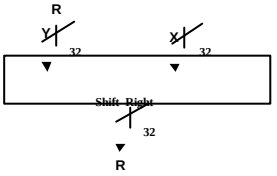
If added Right-to-left connections could support Rotate (not in MIPS but found in ISAs)

Funnel Shifter

Instead Extract 32 bits of 64

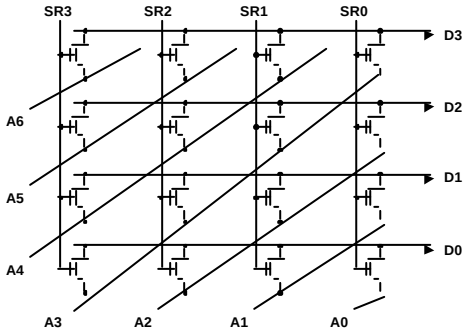


- Shift A by i bits (sa= shift right amount)
- Logical: Y = 0, X=A, sa=i
- Arithmetic? Y = _, X=_, sa=_
- Rotate? Y = _, X=_, sa=_
- Left shifts? Y = _, X=_, sa=_



Barrel Shifter

Technology-dependent solutions: transistor per switch



Divide: Paper & Pencil

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

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

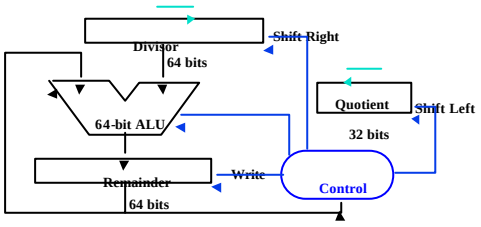
Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder
=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

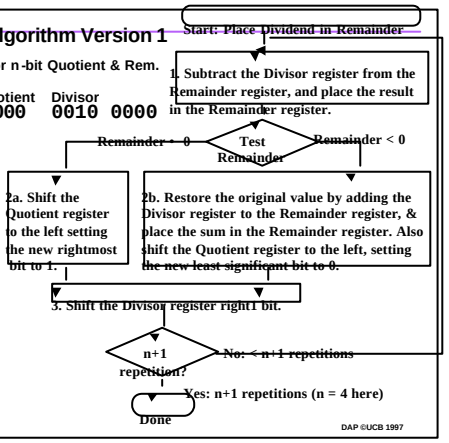
° 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

°Takes n+1 steps for n-bit Quotient & Rem.

Remainder Quotient Divisor
0000 0111 0000 0010 0000



Observations on Divide Version 1

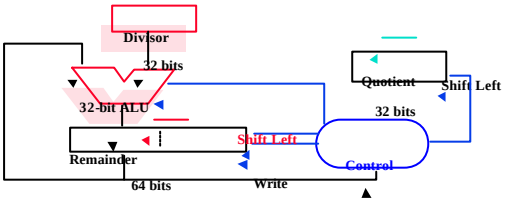
° 1/2 bits in divisor always 0
=> 1/2 of 64-bit adder is wasted
=> 1/2 of divisor is wasted

° Instead of shifting divisor to right, shift remainder to left?

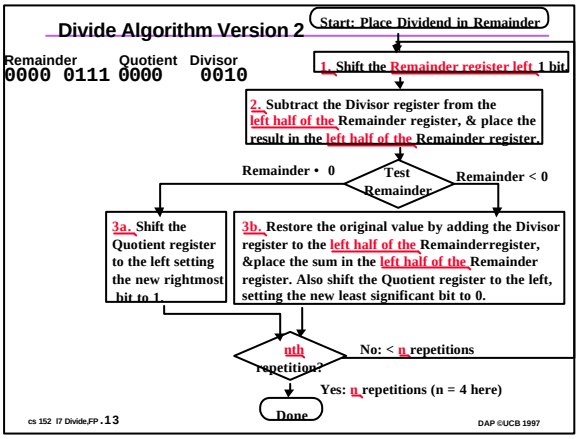
° 1st step cannot produce a 1 in quotient bit (otherwise too big)
=> switch order to shift first and then subtract, can save 1 iteration

DIVIDE HARDWARE Version 2

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



Divide Algorithm Version 2

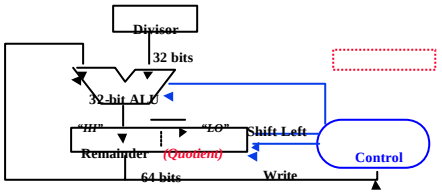


Observations on Divide Version 2

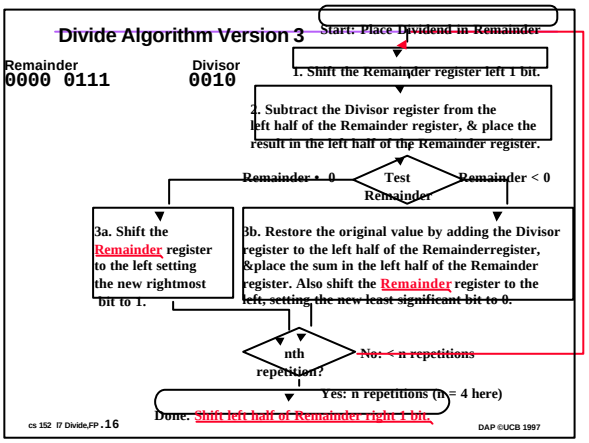
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The 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

DIVIDE HARDWARE Version 3

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



Divide Algorithm Version 3



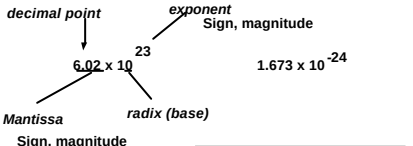
Observations on Divide Version 3

- 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”)

Floating Point

- What can be represented in N bits?
- Unsigned 0 to 2^N
- 2s Complement -2^{N-1} to $2^{N-1} - 1$
- 1s Complement $-2^{N-1}+1$ to $2^{N-1}-1$
- Excess M $-M$ to $2^N - M - 1$
  - (E = e + M)
- BCD 0 to $10^{N/4} - 1$
- But, what about?
  - very large numbers? 9,349,398,989,787,762,244,859,087,678
  - very small number? 0.00000000000000000000045691
  - rationals 2/3
  - irrationals ¦2
  - transcendentals e, š

Recall Scientific Notation



IEEE F.P. $\pm 1.M \times 2^{e - 127}$

° Issues:

- Arithmetic (+, -, *, /)
- Representation, Normal form
- Range and Precision
- Rounding
- Exceptions (e.g., divide by zero, overflow, underflow)
- Errors
- Properties (negation, inversion, if A ° B then A - B ° 0)

Review from Prerequisties: Floating-Point Arithmetic

Representation of floating point numbers in IEEE 754 standard:

single precision

exponent: excess 127 binary integer

mantissa: sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

actual exponent is e = E - 127

0 < E < 255

$N = (-1)^S 2^{E-127} (1.M)$

0 = 0 00000000 0 . . . 0 -1.5 = 1 01111111 10 . . . 0

Magnitude of numbers that can be represented is in the range:

$2^{-126} (1.0)$ to $2^{127} (2 - 2^{23})$

which is approximately:

1.8×10^{-38} to 3.40×10^{38}

(integer comparison valid on IEEE Fl.Pt. numbers of same sign!)

Basic Addition Algorithm

For addition (or subtraction) this translates into the following steps:

(1) compute Ye - Xe *(getting ready to align binary point)*

(2) right shift Xm that many positions to form Xm 2^{Xe-Ye}

(3) compute Xm 2^{Xe-Ye} + Ym

if representation demands normalization, then a normalization step follows:

(4) left shift result, decrement result exponent (e.g., 0.001xx…)
right shift result, increment result exponent (e.g., 101.1xx…)
continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(5) doubly biased exponent must be corrected:

Xe = 7 Xe = 1111 = 15 = 7 + 8
Ye = -3 Ye = 0101 = 5 = -3 + 8
Excess 8 10100 20 4 + 8 + 8

extra subtraction step of the bias amount

(6) if result is 0 mantissa, may need to set the exponent to zero by special step

Extra Bits for rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

How many extra bits?

IEEE: As if computed the result exactly and rounded.

Addition:

	1.xxxxx	1.xxxxx	1.xxxxx
+	1.xxxxx	0.001xxxxx	0.01xxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy
	post-normalization	pre-normalization	pre and post

° *Guard Digits*: digits to the right of the first p digits of significand to guard against loss of digits – can later be shifted left into first P places during normalization.

° Addition: carry-out shifted in

° Subtraction: borrow digit and guard

° Multiplication: carry and guard, Division requires guard

Rounding Digits

normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., B = 10, p = 3:

0 2 1.69 = 1.6900 * 10^{2-bias}

\- 0 0 7.85 = - .0785 * 10^{2-bias}

0 2 1.61 = 1.6115 * 10^{2-bias}

one round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

IEEE Standard:

four rounding modes: round to nearest (default)
round towards plus infinity
round towards minus infinity
round towards 0

round to nearest:
round digit < B/2 then truncate
> B/2 then round up (add 1 to ULP: unit in last place)
= B/2 then round to nearest even digit

it can be shown that this strategy minimizes the mean error introduced by rounding

Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

d0 . d1 d2 d3 . . . dp-1 0 0 0
\+ 0 . 0 0 X . . . X X X S
X X S

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

d0 . d1 d2 d3 . . . dp-1 0 0 0
\- 0 . 0 0 X . . . X X X 0
X X 0

d0 . d1 d2 d3 . . . dp-1 0 0 0
\- 0 . 0 0 X . . . X X X 1

generates a borrow

Rounding Summary:

Radix 2 minimizes wobble in precision

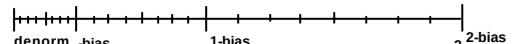
Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

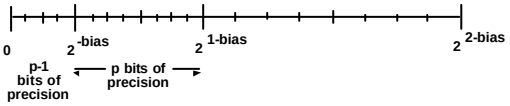
Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

Denormalized Numbers



The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.

IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.



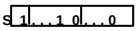
same spacing, half as many values!

NOTE: PDP-11, VAX cannot represent subnormal numbers. These machines underflow to zero instead.

Infinity and NaNs

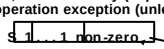
result of operation *overflows* , i.e., is larger than the largest number that can be represented

overflow is not the same as divide by zero (raises a different exception)

+/- infinity 

It may make sense to do further computations with infinity
e.g., X/0 > Y may be a valid comparison

Not a number, but not infinity (e.g. sqrt(-4))
invalid operation exception (unless operation is = or ≠)

NaN 

HW decides what goes here

NaNs propagate: f(NaN) = NaN

Summary

- Pentium: Difference between bugs that board designers must know about and bugs that potentially affect all users
  - Why not make public complete description of bugs in later category?
  - $200,000 cost in June to repair design
  - $500,000,000 loss in December in profits to replace bad parts
  - How much to repair Intel's reputation?
- What is technologists responsibility in disclosing bugs?
- Bits have no inherent meaning: operations determine whether they are really ASCII characters, integers, floating point numbers
- Divide can use same hardware as multiply: Hi & Lo registers in MPS
- Floating point basically follows paper and pencil method of scientific notation using integer algorithms for multiply and divide of significands
- IEEE 754 requires good rounding; special values for NaN, Infinity