香港中文大學
The Chinese University of Hong Kong

CENG3420

L05: Arithmetic and Logic Unit

Bei Yu

byu@cse.cuhk.edu.hk

(Latest update: January 25, 2018)

Spring 2018

Overview

Overview

Addition

Multiplication & Division

Shift

Floating Point Number

Overview

Overview

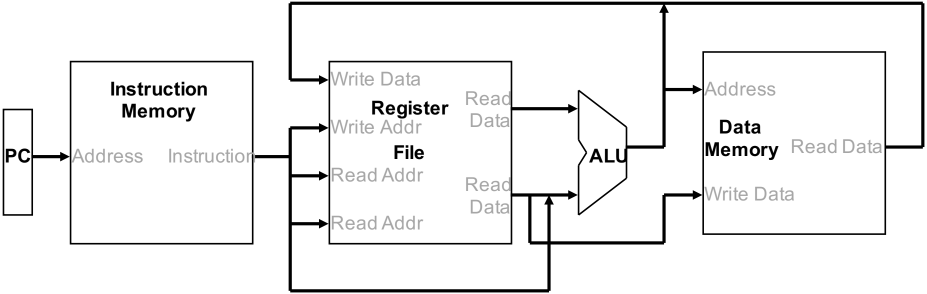
Addition

Multiplication & Division

Shift

Floating Point Number

Abstract Implementation View



Arithmetic

Where we've been: **abstractions**

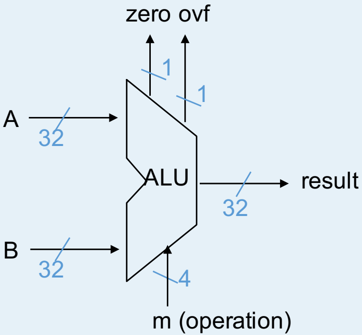
- Instruction Set Architecture (ISA)
- Assembly and machine language

Arithmetic

Where we've been: **abstractions**

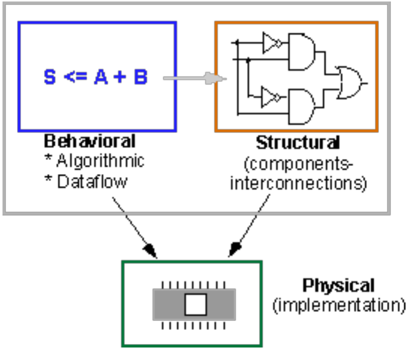
- Instruction Set Architecture (ISA)
- Assembly and machine language

What's up ahead: Implementing the ALU architecture



Review: VHDL

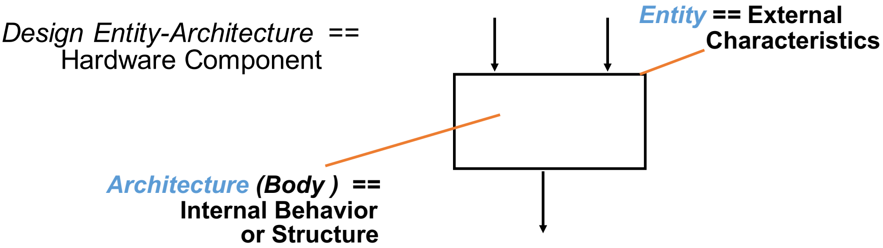
- Supports design, documentation, simulation & verification, and synthesis of hardware
- Allows integrated design at behavioral & structural levels



Review: VHDL (cont.)

Basic Structure

- Design entity-architecture descriptions
- Time-based execution (discrete event simulation) model



Review: Entity-Architecture Features

Entity

defines externally visible characteristics

- Ports: channels of communication
  - signal names for inputs, outputs, clocks, control
- Generic parameters: define class of components
  - timing characteristics, size (fan-in), fan-out

Review: Entity-Architecture Features (cont.)

Architecture

defines the internal behavior or structure of the circuit

- Declaration of internal signals
- Description of behavior
  - collection of Concurrent Signal Assignment (CSA) statements (indicated by $<=$);
  - can also model temporal behavior with the delay annotation
  - one or more processes containing CSAs and (sequential) variable assignment statements (indicated by $:=$)
- Description of structure
  - interconnections of components; underlying behavioral models of each component must be specified

ALU VHDL Representation

```
entity ALU is
  port(A, B:  in std_logic_vector (31 downto 0);
          m:  in std_logic_vector (3 downto 0);
          result: out std_logic_vector (31 downto 0);
          zero: out std_logic;
          ovf: out std_logic)
end ALU;

architecture process_behavior of ALU is
. . .
begin
    ALU: process(A, B, m)
    begin
        . . .
        result := A + B;
        . . .
    end process ALU;
end process_behavior;
```

Machine Number Representation

- Bits are just bits (have no inherent meaning)∗
- Binary numbers (base 2) – integers

Of course, it gets more complicated:

- storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)
- have to be able to represent negative numbers, e.g., how do we specify -8 in

```
addi    $sp, $sp, -8     #$sp = $sp - 8
```

- in real systems have to provide for more than just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

∗conventions define the relationships between bits and numbers

MIPS Representation

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

$$
\begin{array}{lcl}
\texttt{0000 0000 0000 0000 0000 0000 0000 0000}_{\texttt{two}} & = & \texttt{0}_{\texttt{ten}} \\
\texttt{0000 0000 0000 0000 0000 0000 0000 0001}_{\texttt{two}} & = & \texttt{+ 1}_{\texttt{ten}} \\
\texttt{0000 0000 0000 0000 0000 0000 0000 0010}_{\texttt{two}} & = & \texttt{+ 2}_{\texttt{ten}} \\
\texttt{...} & & \\
\texttt{0111 1111 1111 1111 1111 1111 1111 1110}_{\texttt{two}} & = & \texttt{+ 2,147,483,646}_{\texttt{ten}} \\
\texttt{0111 1111 1111 1111 1111 1111 1111 1111}_{\texttt{two}} & = & \texttt{+ 2,147,483,647}_{\texttt{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0000}_{\texttt{two}} & = & \texttt{- 2,147,483,648}_{\texttt{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0001}_{\texttt{two}} & = & \texttt{- 2,147,483,647}_{\texttt{ten}} \\
\texttt{1000 0000 0000 0000 0000 0000 0000 0010}_{\texttt{two}} & = & \texttt{- 2,147,483,646}_{\texttt{ten}} \\
\texttt{...} & & \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1101}_{\texttt{two}} & = & \texttt{- 3}_{\texttt{ten}} \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1110}_{\texttt{two}} & = & \texttt{- 2}_{\texttt{ten}} \\
\texttt{1111 1111 1111 1111 1111 1111 1111 1111}_{\texttt{two}} & = & \texttt{- 1}_{\texttt{ten}}
\end{array}
$$

What if the bit string represented addresses?

- need operations that also deal with only positive (unsigned) integers

Two's Complement Operations

- Negating a two's complement number – complement all the bits and then add a 1
  - remember: "negate" and "invert" are quite different!

- Converting n-bit numbers into numbers with more than n bits:
  - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
  - sign extend: copy the most significant bit (the sign bit) into the other bits

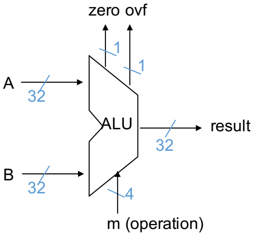
```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - sign extension versus zero extend (`lb` vs. `lbu`)

Design the MIPS Arithmetic Logic Unit (ALU)

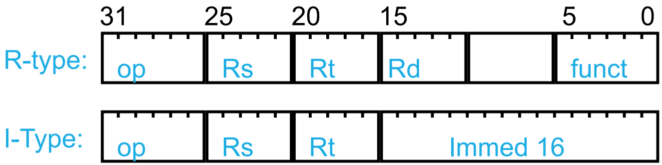
- Must support the Arithmetic/Logic operations of the ISA

```
add, addi, addiu, addu, sub, subu
mult, multu, div, divu, sqrt
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 detected: `add, addi, sub`

MIPS Arithmetic and Logic Instructions



Type	op	funct
ADDI	001000	xx
ADDIU	001001	xx
SLTI	001010	xx
SLTIU	001011	xx
ANDI	001100	xx
ORI	001101	xx
XORI	001110	xx
LUI	001111	xx

Type	op	funct
ADD	000000	100000
ADDU	000000	100001
SUB	000000	100010
SUBU	000000	100011
AND	000000	100100
OR	000000	100101
XOR	000000	100110
NOR	000000	100111

Type	op	funct
	000000	101000
	000000	101001
SLT	000000	101010
SLTU	000000	101011
	000000	101100

Overview

Overview

Addition

Multiplication & Division

Shift

Floating Point Number

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
  0111        0111        0110
+ 0110      - 0110      - 0101
-------     ------      -------
```

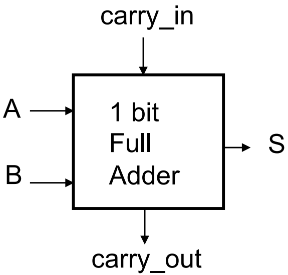
- Two's complement operations are easy: do subtraction by negating and then adding

```
  0111    ->      0111
- 0110    ->    + 1010
------          -------
```

- Overflow (result too large for finite computer word). E.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
------
```

Building a 1-bit Binary Adder



A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

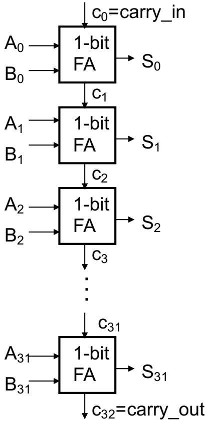
S = A xor B xor carry_in

carry_out = A&B | A&carry_in | B&carry_in

(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

Building 32-bit Adder



- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect ...

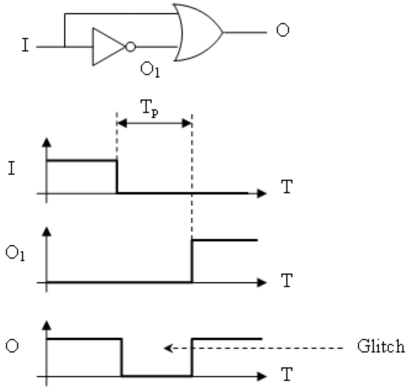
- Ripple Carry Adder (RCA)
  - ☺: simple logic, so small (low cost)
  - ☹: slow and lots of glitching (so lots of energy consumption)

Glitch

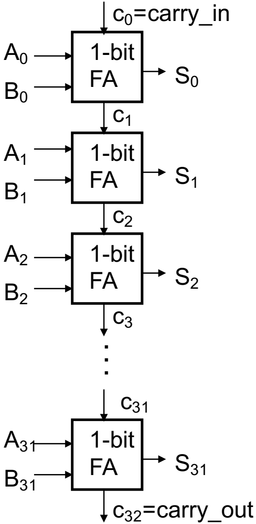
Glitch

invalid and unpredicted output that can be read by the next stage and result in a wrong action

Example: Draw the propagation delay



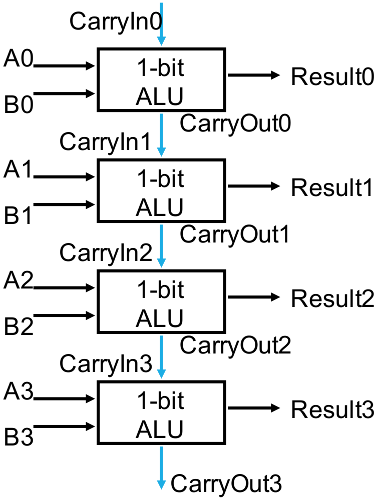
Glitch in RCA



A	B	carry_in	carry_out	S
0	0	**0**	0	**0**
0	0	**1**	0	**1**
0	1	**0**	0	**1**
0	1	**1**	1	**0**
1	0	**0**	0	**1**
1	0	**1**	1	**0**
1	1	**0**	1	**0**
1	1	**1**	1	**1**

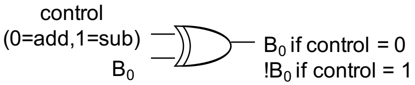
But What about Performance?

- Critical path of n-bit ripple-carry adder is $n \times CP$
- Design trick: throw hardware at it (Carry Lookahead)



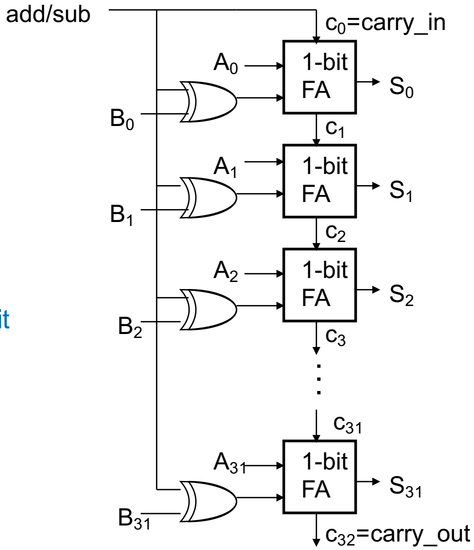
A 32-bit Ripple Carry Adder/Subtractor

- complement all the bits



- add a 1 in the least significant bit

```
A    0111   ->    0111
B  - 0110   ->  + 1001
     ----
     0001            1
                  ----
                1 0001
```



Minimal Implementation of a Full Adder

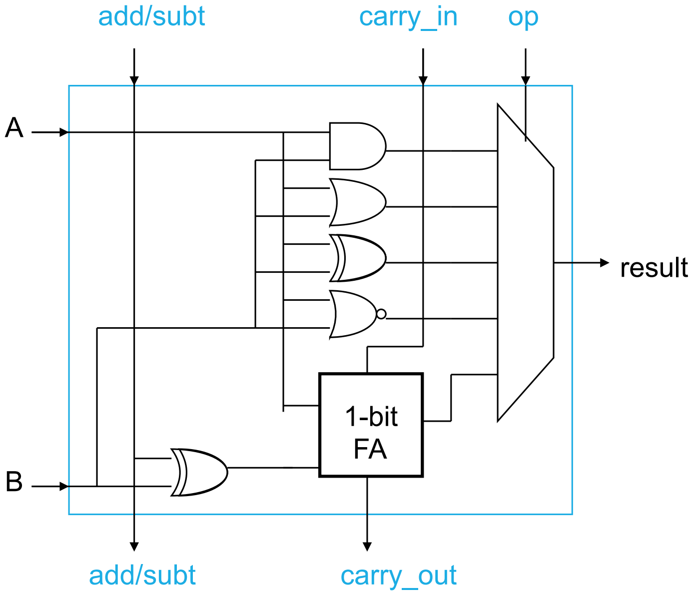
Gate library: inverters, 2-input NANDs, or-and-inverters

```
architecture concurrent_behavior of full_adder is
   signal t1, t2, t3, t4, t5: std_logic;
begin
     t1 <= not A after 1 ns;
     t2 <= not cin after 1 ns;
     t4 <= not((A or cin) and B) after 2 ns;
     t3 <= not((t1 or t2) and (A or cin)) after 2 ns;
     t5 <= t3 nand B after 2 ns;
     S <= not((B or t3) and t5) after 2 ns;
     cout <= not((t1 or t2) and t4) after 2 ns;
end concurrent_behavior;
```

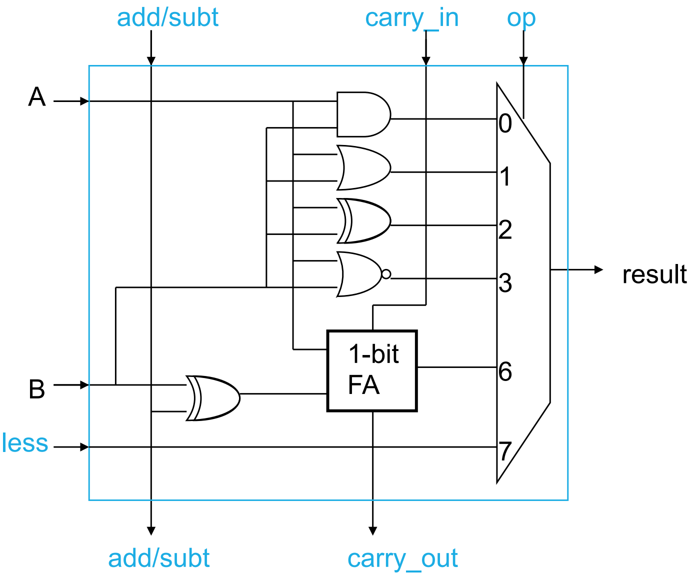
Tailoring the ALU to the MIPS ISA

- Also need to support the logic operations (`and`, `nor`, `or`, `xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function and a mux to choose the output
- Also need to support the set-on-less-than instruction (slt)
  - Uses subtraction to determine if $(a - b) < 0$ (implies $a < b$)
- Also need to support test for equality (`bne`, `beq`)
  - Again use subtraction: $(a - b) = 0$ implies $a = b$
- Also need to add overflow detection hardware
  - overflow detection enabled only for `add, addi, sub`
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

A Simple ALU Cell with Logic Op Support



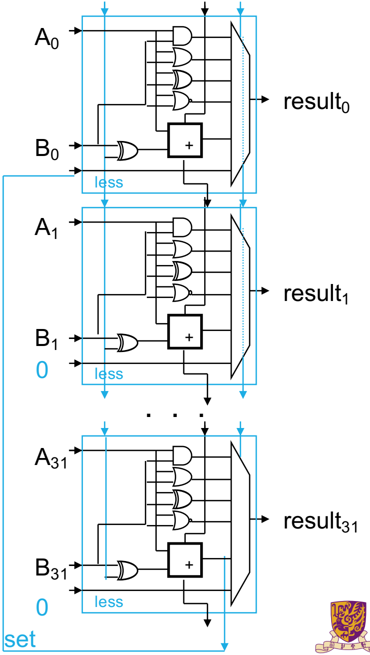
A Simple ALU Cell with Logic Op Support



Modifying the ALU Cell for `slt`

Modifying the ALU for `slt`

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result
- Tie the most significant sum bit (sign bit) to the low order less input



Overflow Detection

Overflow occurs when the result is too large to represent in the number of bits allocated

- adding two positives yields a negative
- or, adding two negatives gives a positive
- or, subtract a negative from a positive gives a negative
- or, subtract a positive from a negative gives a positive

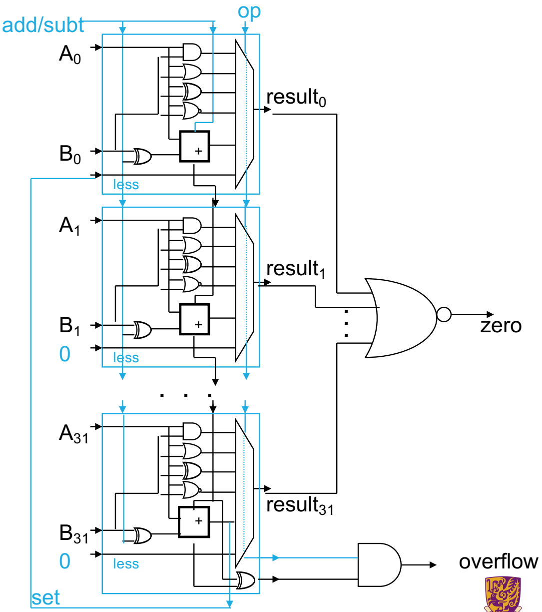
Question: prove you can detect overflow by:

Carry into MSB xor Carry out of MSB

	0	1	1	1		
		0	1	1	1	7
+		0	0	1	1	3
		1	0	1	0	– 6

	1	0				
		1	1	0	0	–4
+		1	0	1	1	– 5
		0	1	1	1	7

Modifying the ALU for Overflow



- Modify the most significant cell to determine overflow output setting
- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)

Overflow Detection and Effects

- On overflow, an exception (interrupt) occurs
- Control jumps to predefined address for exception
- Interrupted address (address of instruction causing the overflow) is saved for possible resumption
- Don't always want to detect (interrupt on) overflow

New MIPS Instructions

Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add unsigned	0 and 21	addu $s1, $s2, $s3	$s1 = $s2 + $s3
	sub unsigned	0 and 23	subu $s1, $s2, $s3	$s1 = $s2 - $s3
	add imm.unsigned	9	addiu $s1, $s2, 6	$s1 = $s2 + 6
Data Transfer	ld byte unsigned	24	lbu $s1, 20($s2)	$s1 = Mem($s2+20)
	ld half unsigned	25	lhu $s1, 20($s2)	$s1 = Mem($s2+20)
Cond. Branch (I & R format)	set on less than unsigned	0 and 2b	sltu $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than imm unsigned	b	sltiu $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0

- Sign extend: `addi, addiu, slti`
- Zero extend: `andi, ori, xori`
- Overflow detected: `add, addi, sub`

Overview

Overview

Addition

Multiplication & Division

Shift

Floating Point Number

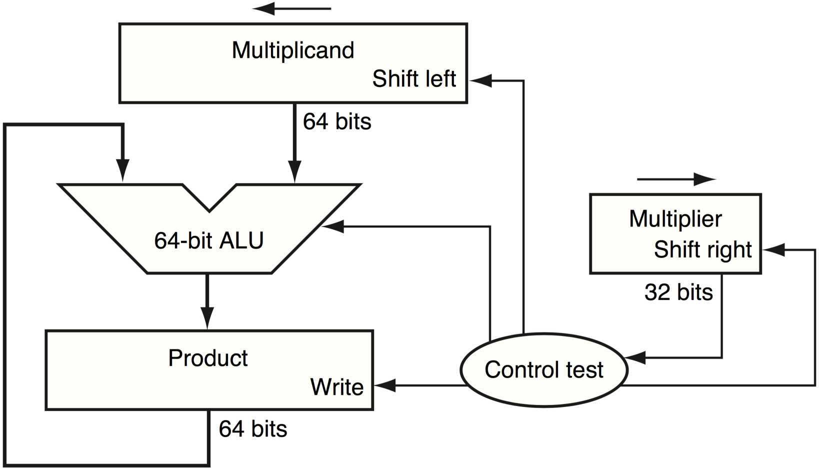
Multiplication

- More complicated than addition
- Can be accomplished via shifting and adding

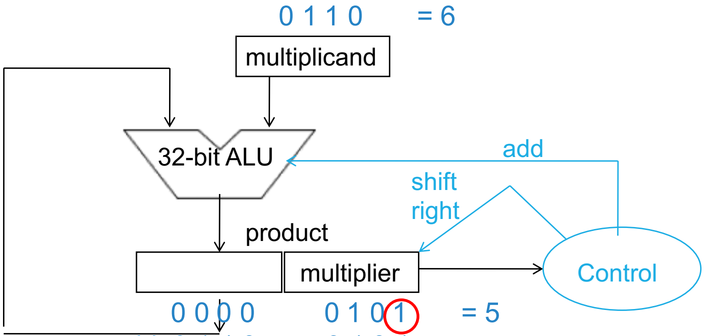
```
        0010    (multiplicand)
     x  1011    (multiplier)
       0010
      0010      (partial product
     0000         array)
    0010
    00010110    (product)
```

- Double precision product produced
- More time and more area to compute

First Version of Multiplication Hardware



Add and Right Shift Multiplier Hardware



	0 0 0 0		0 1 0 1	= 5
add	0 1 1 0		0 1 0 1	
	0 0 1 1	→	0 0 1 0	
add	0 0 1 1		0 0 1 0	
	0 0 0 1	→	1 0 0 1	
add	0 1 1 1		1 0 0 1	
	0 0 1 1	→	1 1 0 0	
add	0 0 1 1		1 1 0 0	
	0 0 0 1	→	1 1 1 0	= 30

MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product

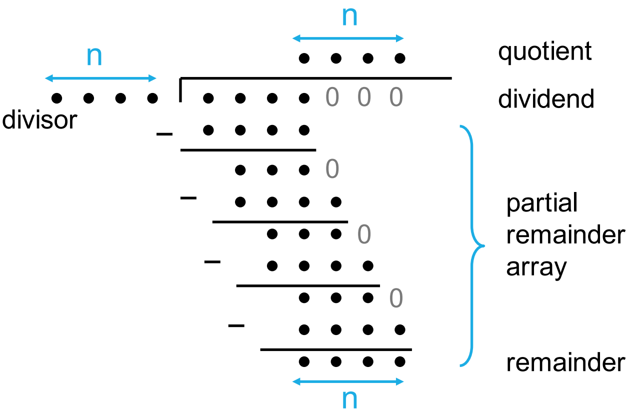
```
mult    $s0, $s1        # hi||lo = $s0 * $s1
```

0	16	17	0	0	0x18

- Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
- Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file
- Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Division

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



Question: Division

Dividing 1001010 by 1000

MIPS Divide Instruction

- Divide generates the reminder in `hi` and the quotient in `lo`

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

op	rs	rt	rd	shamt	funct

- Instructions `mflo rd` and `mfhi rd` are provided to move the quotient and reminder to (user accessible) registers in the register file
- As with multiply, divide ignores overflow so software must determine if the quotient is too large.
- Software must also check the divisor to avoid division by 0.

Overview

Overview

Addition

Multiplication & Division

Shift

Floating Point Number

Shift Operations

- Shifts move all the bits in a word left or right

```
sll  $t2, $s0, 8 #$t2 = $s0 << 8 bits
srl  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
sra  $t2, $s0, 8 #$t2 = $s0 >> 8 bits
```

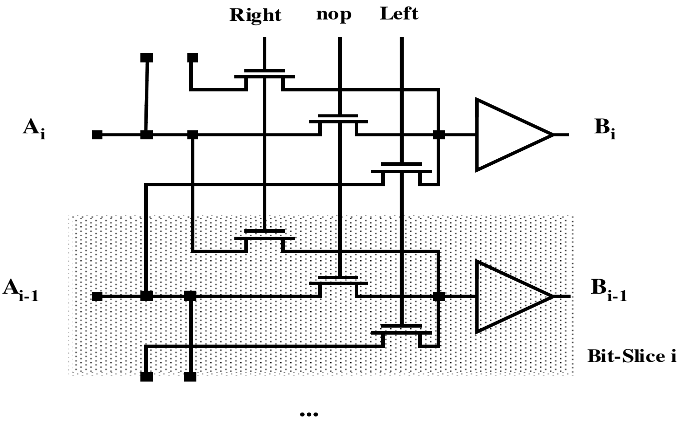
op	rs	rt	rd	shamt	funct

- Notice that a 5-bit `shamt` field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions
- Logical shifts fill with zeros, arithmetic left shifts fill with the sign bit

The shift operation is implemented by hardware separate from the ALU

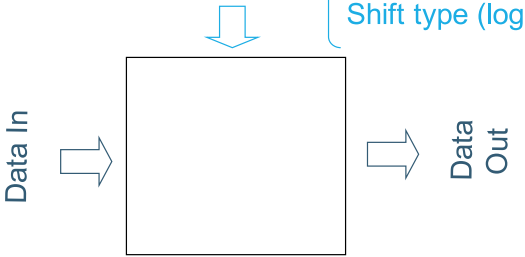
Using a barrel shifter, which would takes lots of gates in discrete logic, but is pretty easy to implement in VLSI

A Simple Shifter

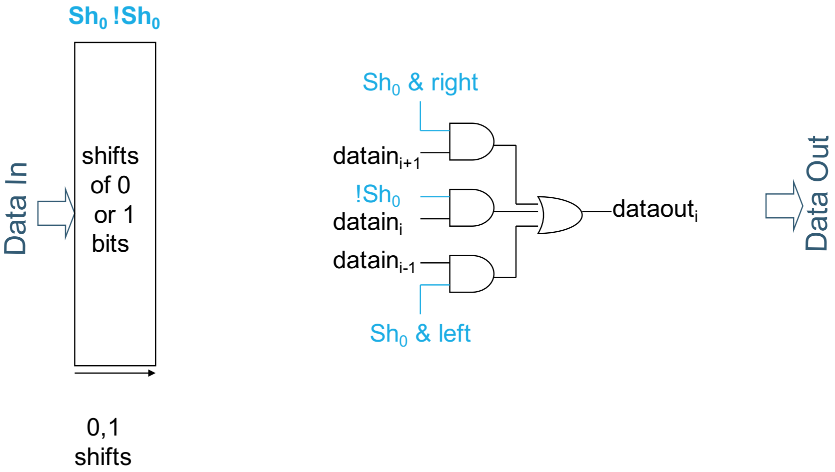


Parallel Programmable Shifters

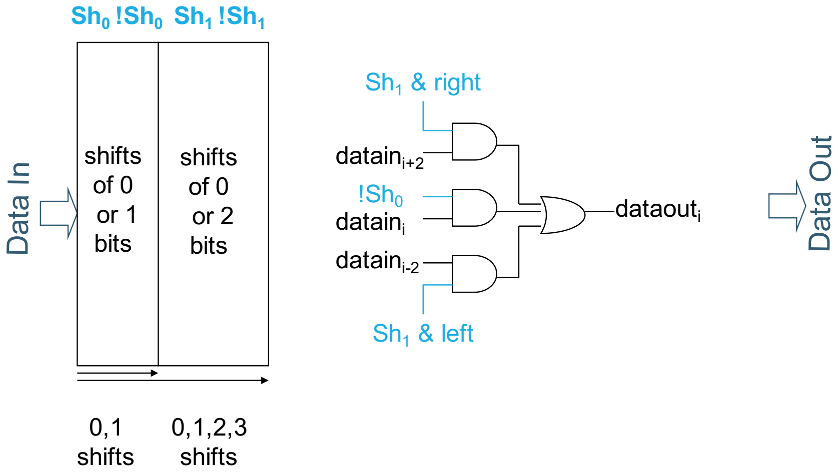
Control =
- Shift amount ($Sh_4Sh_3Sh_2Sh_1Sh_0$)
- Shift direction (left, right)
- Shift type (logical, arithmetic)



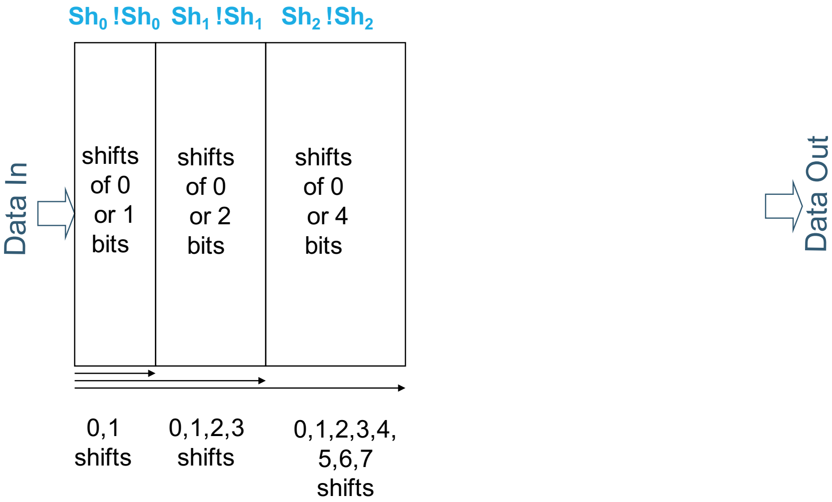
Logarithmic Shifter Structure



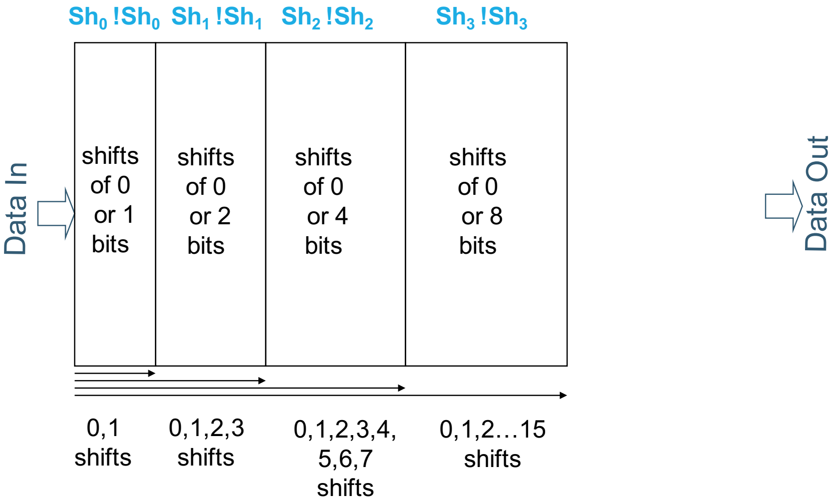
Logarithmic Shifter Structure



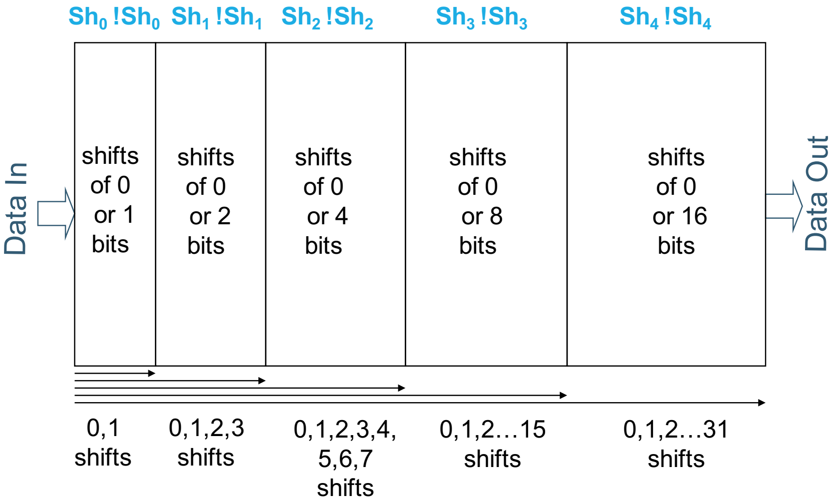
Logarithmic Shifter Structure



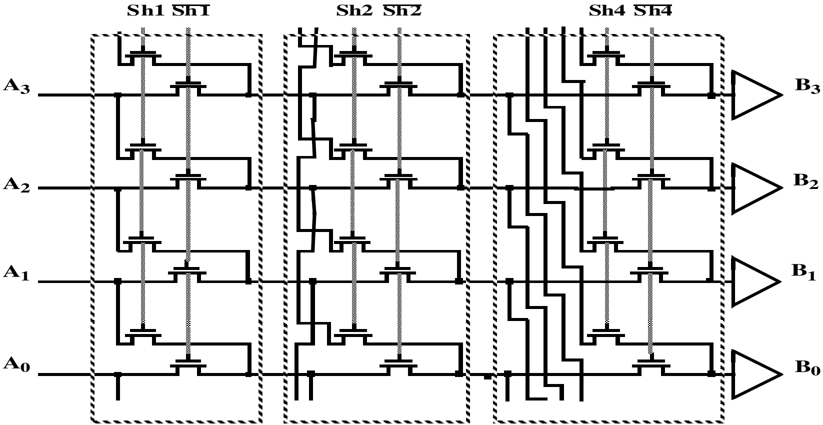
Logarithmic Shifter Structure



Logarithmic Shifter Structure



Logarithmic Shifter Structure



Overview

Overview

Addition

Multiplication & Division

Shift

Floating Point Number

Floating Point Number

Scientific notation: 6.6254×10^{-27}

- A normalized number of certain accuracy (e.g. 6.6254 is called the mantissa)
- Scale factors to determine the position of the decimal point (e.g. 10^{-27} indicates position of decimal point and is called the exponent; the **base** is implied)
- Sign bit

Normalized Form

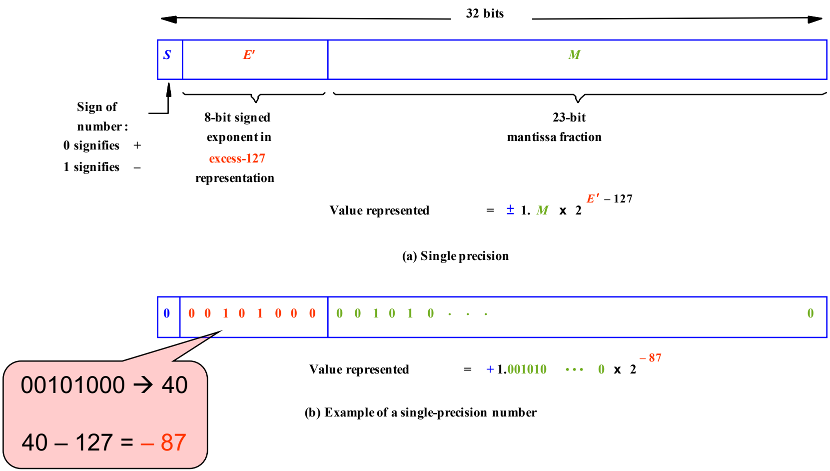
- Floating Point Numbers can have multiple forms, e.g.

$$
\begin{aligned}
0.232 \times 104 &= 2.32 \times 10^3 \\
&= 23.2 \times 10^2 \\
&= 2320. \times 10^0 \\
&= 232000. \times 10^{-2}
\end{aligned}
$$

- It is desirable for each number to have a unique representation => Normalized Form
- We normalize Mantissa's in the Range $[1..R)$, where R is the Base, e.g.:
  - $[1..2)$ for BINARY
  - $[1..10)$ for DECIMAL

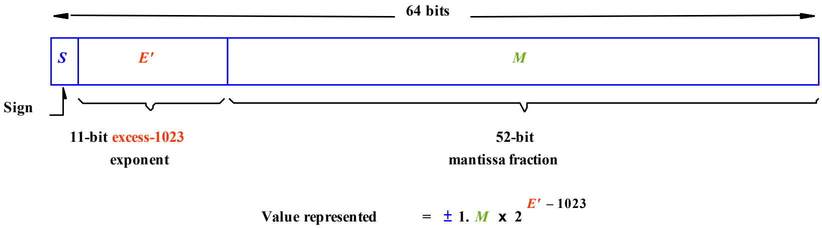
IEEE Standard 754 Single Precision

32-bit, float in C / C++ / Java



IEEE Standard 754 Double Precision

64-bit, float in C / C++ / Java



(c) Double precision

Question:

What is the IEEE single precision number $40C0\ 0000_{16}$ in decimal?

Question:

What is the IEEE single precision number 40C0 0000_{16} in decimal?

- Binary: 0100 0000 1100 0000 0000 0000 0000 0000
- Sign: +
- Exponent: 129 - 127 = +2
- Mantissa: 1.100 0000 $\ldots_2 \rightarrow 1.5_{10} \times 2^{+2}$
- $\rightarrow$ +110.0000 $\ldots_2$
- Decimal Answer = $+6.0_{10}$

Question:

What is -0.5_{10} in IEEE single precision binary floating point format?

Question:

What is -0.5$_{10}$ in IEEE single precision binary floating point format?

- Binary: $1.0... \times 2^{-1}$ (in binary)
- Exponent: 127 + (-1) = 01111110
- Sign bit: 1
- Mantissa: 1.000 0000 0000 0000 0000 0000
- Binary representation: 1011 1111 0000 0000 0000 0000 0000 0000

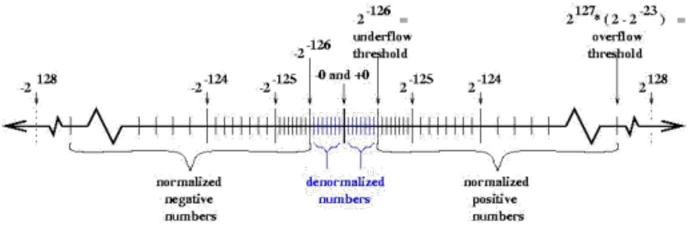
Ref: IEEE Standard 754 Numbers

- Normalized +/– 1.d…d x 2^{exp}
- ***Denormalized*** +/– 0.d…d x 2^{min_exp} → to represent *near-zero* numbers
  e.g. + 0.0000…0000001 x 2^{-126} for Single Precision

Format	# bits	# significant bits	macheps	# exponent bits	exponent range
Single	32	1+23	2^{-24} ($\sim 10^{-7}$)	8	$2^{-126} - 2^{+127}$ ($\sim 10^{\pm 38}$)
Double	64	1+52	2^{-53} ($\sim 10^{-16}$)	11	$2^{-1022} - 2^{+1023}$ ($\sim 10^{\pm 308}$)
Double Extended	>=80	>=64	$\le 2^{-64}$ ($\sim 10^{-19}$)	>=15	$2^{-16382} - 2^{+16383}$ ($\sim 10^{\pm 4932}$)

(Double Extended is *80 bits on all Intel machines*)

macheps = Machine Epsilon = ε_{mach} $= 2^{-(\#\text{ significand bits})}$



Special Values

Exponents of all 0's and all 1's have special meaning

- E=0, M=0 represents 0 (sign bit still used so there is ± 0)
- E=0, M$\neq$0 is a denormalized number $\pm$0.M $\times 2^{-127}$ (smaller than the smallest normalized number)
- E=All 1's, M=0 represents $\pm$`Infinity`, depending on Sign
- E=All 1's, M$\neq$0 represents `NaN`

Other Features

+, -, x, /, sqrt, remainder, as well as conversion to and from integer are correctly rounded

- As if computed with infinite precision and then rounded
- Transcendental functions (that cannot be computed in a finite number of steps e.g., sine, cosine, logarithmic, , e, etc.) may not be correctly rounded

Exceptions and Status Flags

- Invalid Operation, Overflow, Division by zero, Underflow, Inexact

Floating point numbers can be treated as "integer bit-patterns" for comparisons

- If Exponent is all zeroes, it represents a denormalized, very small and near (or equal to) zero number
- If Exponent is all ones, it represents a very large number and is considered infinity (see next slide.)

Dual Zeroes: +0 (0x00000000) and -0 (0x80000000): they are treated as the same

Other Features

`Infinity` is like the mathematical one

- `Finite` / `Infinity` $\rightarrow$ 0
- `Infinity` $\times$ `Infinity` $\rightarrow$ `Infinity`
- Non-zero / 0 $\rightarrow$ `Infinity`
- `Infinity` $^{\{\texttt{Finite}\ \text{or}\ \texttt{Infinity}\}}$ $\rightarrow$ `Infinity`

`NaN` (Not-a-Number) is produced whenever a limiting value cannot be determined:

- `Infinity` - `Infinity` $\rightarrow$ `NaN`
- `Infinity` / `Infinity` $\rightarrow$ `NaN`
- 0 / 0 $\rightarrow$ `NaN`
- `Infinity` $\times$ 0 $\rightarrow$ `NaN`
- If x is a `NaN`, x $\neq$ x
- Many systems just store the result quietly as a `NaN` (all 1's in exponent), some systems will signal or raise an exception

Inaccurate Floating Point Operations

- E.g. Find 1st root of a quadratic equation
  - r = (–b + sqrt(b*b – 4*a*c)) / (2*a)

Sparc processor, Solaris, gcc 3.3 (ANSI C),

```
Expected Answer   0.00023025562642476431
double            0.00023025562638524986
float             0.00024670246057212353
```

- Problem is that if c is near zero,

$$\text{sqrt}(b*b - 4*a*c) \approx b$$

- Rule of thumb: use the highest precision which does not give up too much speed

Catastrophic Cancellation

- (a – b) is inaccurate when a $\approx$ b
- Decimal Examples
  - Using 2 significant digits to compute mean of 5.1 and 5.2 using the formula (a+b) / 2:

    a + b = 10 (with 2 sig. digits, 10.3 can only be stored as 10)
    10 / 2 = 5.0 (the computed mean is less than both numbers!!!)
  - Using 8 significant digits to compute sum of three numbers:

    $(11111113 + (-11111111)) + 7.5111111 = 9.5111111$

    $11111113 + ((-11111111) + 7.5111111) = 10.000000$
- Catastrophic cancellation occurs when

$$\left|\frac{[round(x)\text{"}\bullet\text{"}round(y)] - round(x \bullet y)}{round(x \bullet y)}\right| >> \varepsilon_{mach}$$