Chapter 3: Computer Arithmetic

- Not implement ALU
  - Domain knowledge
- Do think about
  - Representing numbers (and ALU)
  - ISA from data perspective
- (optional) subword parallelism

Big Picture

- Part 1: what is computer, CSE, computer architecture?
  - Fundamental concepts and principles
- Part 2: ISA (externals)
  - Ch. 1: performance
    - Exe. time, benchmark, model, RISC, power, multicore
  - Ch. 2: language of computer; ISA
    - What is a good ISA? Today's RISC-style ISA (MIPS)
  - Ch. 3: computer arithmetic (key internal building block)
    - Data representation and ALU, ISA – data perspective
- Part 3: implementation of ISA (internals)
  - Ch. 4: processor
  - Ch. 5: memory system

Numbers

- Data types (INT and FP)
  - Numbers int,fp
  - Text, audio, image, video, and so on int
- Representing numbers and ALU
  - Positive and negative integers
    - Numbers are finite (overflow)
  - Scientific numbers (too big or too small)
- ISA from data perspective

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

010(2) + 110(-2) = 100
-> correct 필요

010(2) + 101(-2) = 111 (-0)

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

Possible Representations

Biased Notation:	Biased Notation:	Unsigned:
000 = -3	000 = -4	000 = +0
001 = -2	001 = -3	001 = +1
010 = -1	010 = -2	010 = +2
011 = 0	011 = -1	011 = +3
100 = +1	100 = 0	100 = +4
101 = +2	101 = +1	101 = +5
110 = +3	110 = +2	110 = +6
111 = +4	111 = +3	111 = +7
(Bias 3)	(Bias 4)	

MIPS (반복)

- **32 bit signed numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations (반복)

- Negating two's complement number: invert all bits and add 1
  - Remember: "negate" and "invert" are quite different!
- Signed and unsigned numbers
  - Memory address: start at 0 and end at the largest (e.g., FFFF)
  - Negative address make no sense
  - Some programming languages make this distinction
    - ⇒ C: **int** versus **unsigned int**
  - MIPS comparison instructions
    - ⇒ signed: **slt, slti**
    - ⇒ unsigned: **sltu, sltiu**

Signed and unsigned loads (반복)

- Singed load: copy sign repeatedly to fill rest of register
  - e.g., two's complement numbers
- Unsigned load: fill with 0s to left of data
  - When bit pattern is unsigned
- MIPS instructions
  - **lb** versus **lbu**
  - **lh** versus **lhu**
  - **lw**

† No concern to store

Addition & Subtraction

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

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

- Two's complement operations easy
  - subtraction using addition of negative numbers

```
   0111
 + 1010
```

- Arithmetic overflow (result too large for finite word):
  - e.g., adding two n-bit numbers not yield an n-bit number

```
   0111
 + 0001
   1000
```

Two's Complement Arithmetic Overflow

- No overflow when adding a positive and a negative number
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ? no
  - Can overflow occur if A is 0 ? yes B=2^n 일때
- Overflow detection using a single exclusive-or gate

two's complement overflow
양수 양수 0 1
음수 음수 1 0
상위 두개의 carry bit을 exclusive or 시키면 detect 쉽다.
-> detect 되면 interrupt (exception)

70	0110	0110	-70	1011	1010
+ 80	0101	0000	-80	1011	0000
150	1001	0110	-150	0110	1010

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Given overflow, what does OS do?
- Arithmetic overflow: program bug
  - What do programmers should do? variable 사용할때 표현 범위 유의
    - Flight control vs. homework assignment
- What about unsigned integers?
  - Commonly used for memory addresses (by compiler)
  - 2's complement arithmetic overflows are ignored
- MIPS solution: provide two kinds of arithmetic instructions

  add, addi, sub

  addu, addiu, subu // do not detect overflow

h/w 적으로 무시해라

MIPS Assembly Language: add, subtract

MIPS assembly language

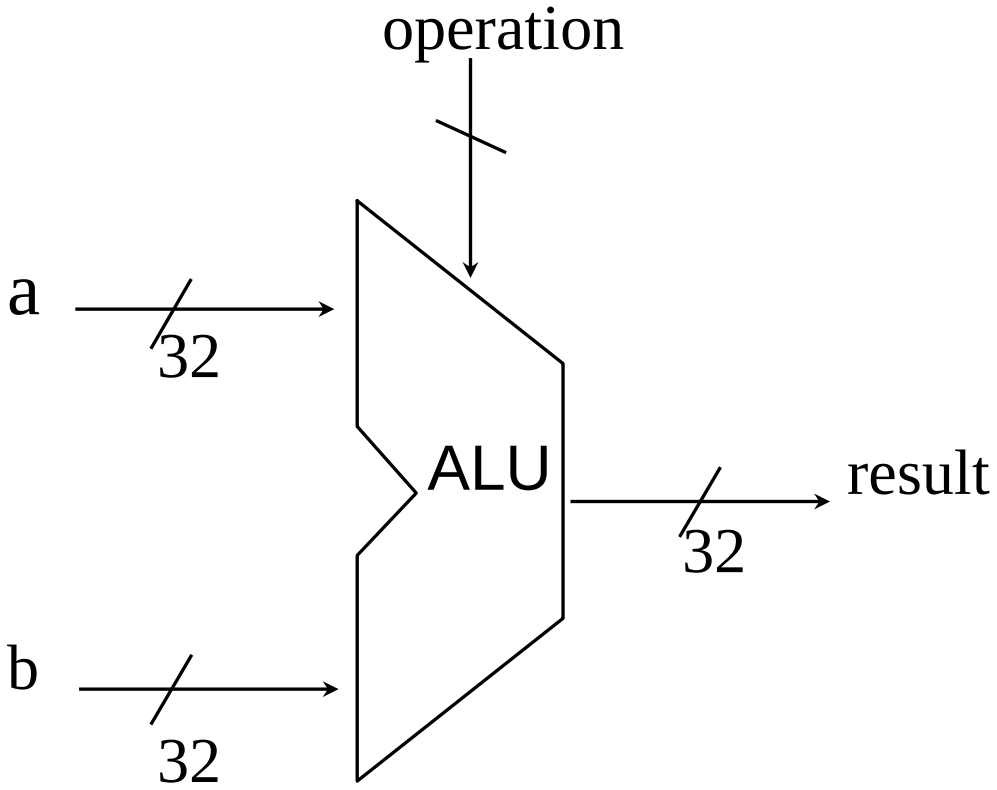
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	`$s1 = $epc`	Used to copy Exception PC plus other special registers

Build ALU

(Topic 1 에서 abstraction 이라는 개념을 설명하며 어느 정도 다룬 내용임)

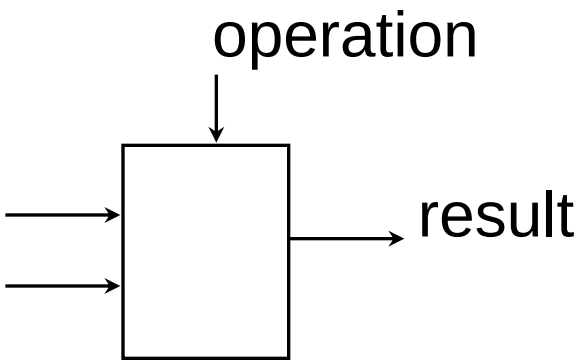
32-bit ALU Design

- Operations
  - Arithmetic: add, subtract, multiply, divide
  - Logical: bitwise AND, OR, NOT

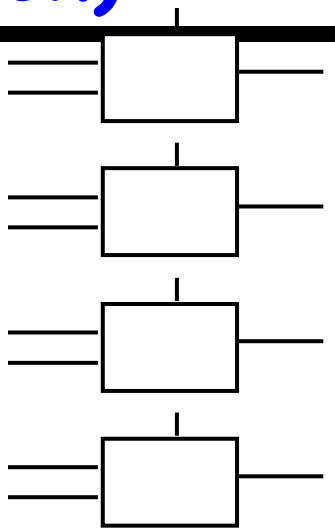


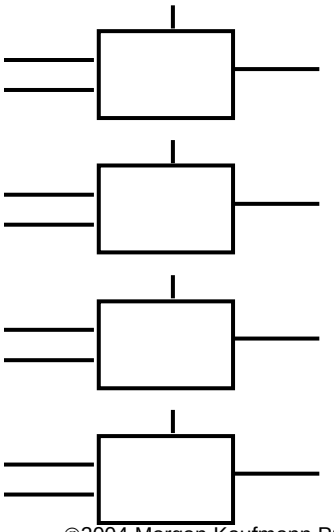
32-bit ALU Design (Abstraction)

- Build 1-bit ALU
  - Use 32 of them in parallel



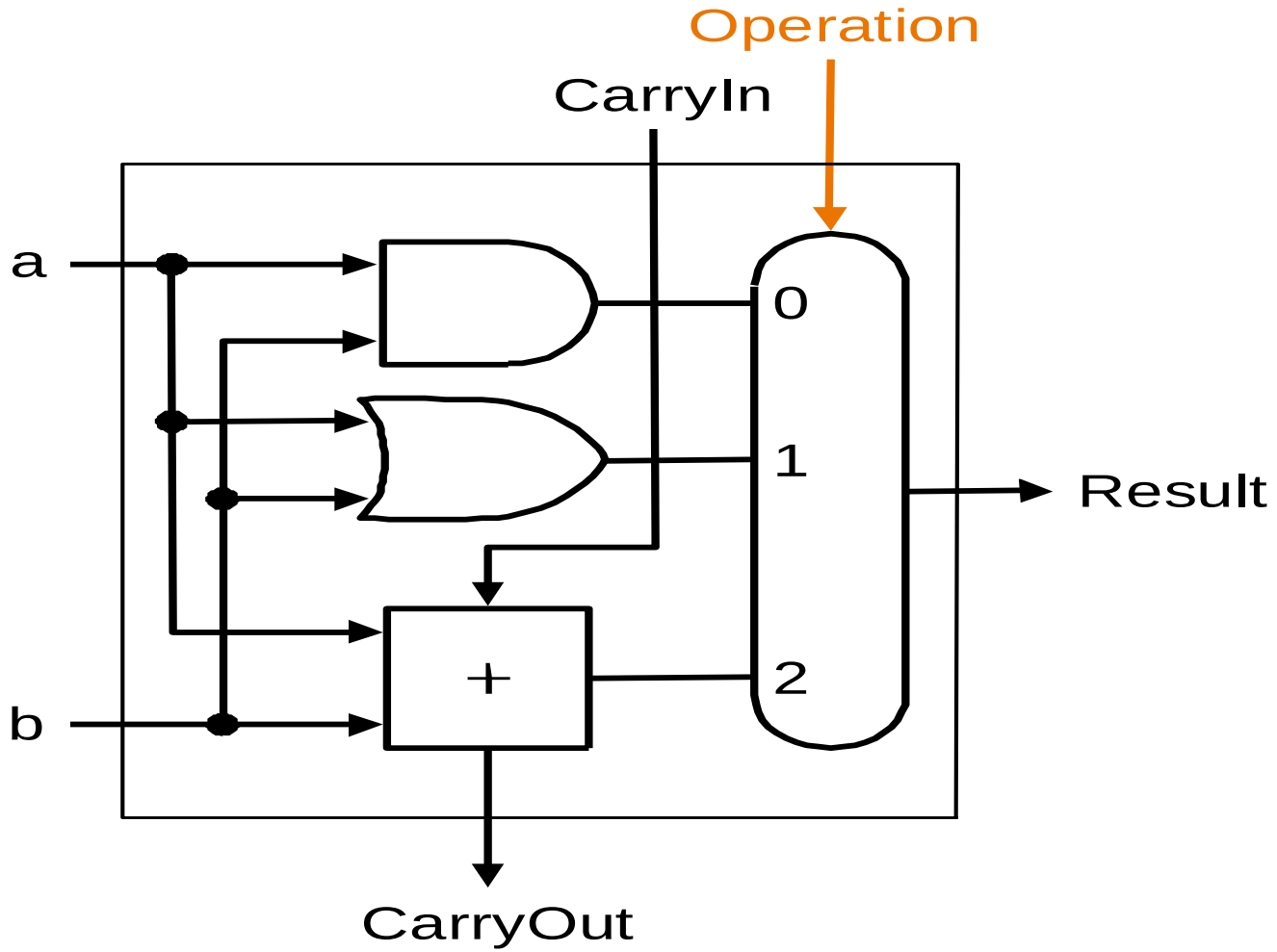
1-bit ALU





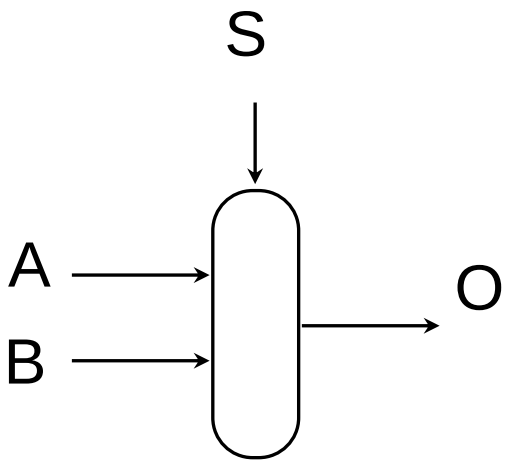
1-bit ALU Design (디지털논리설계; Abstraction)

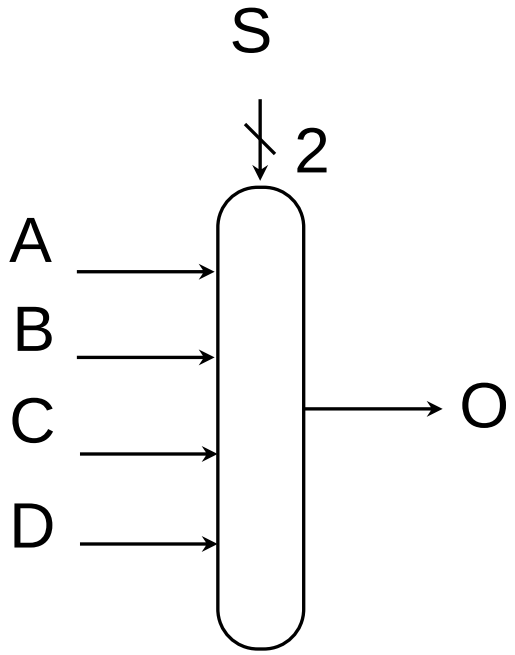
- 1-bit ADD, AND, OR



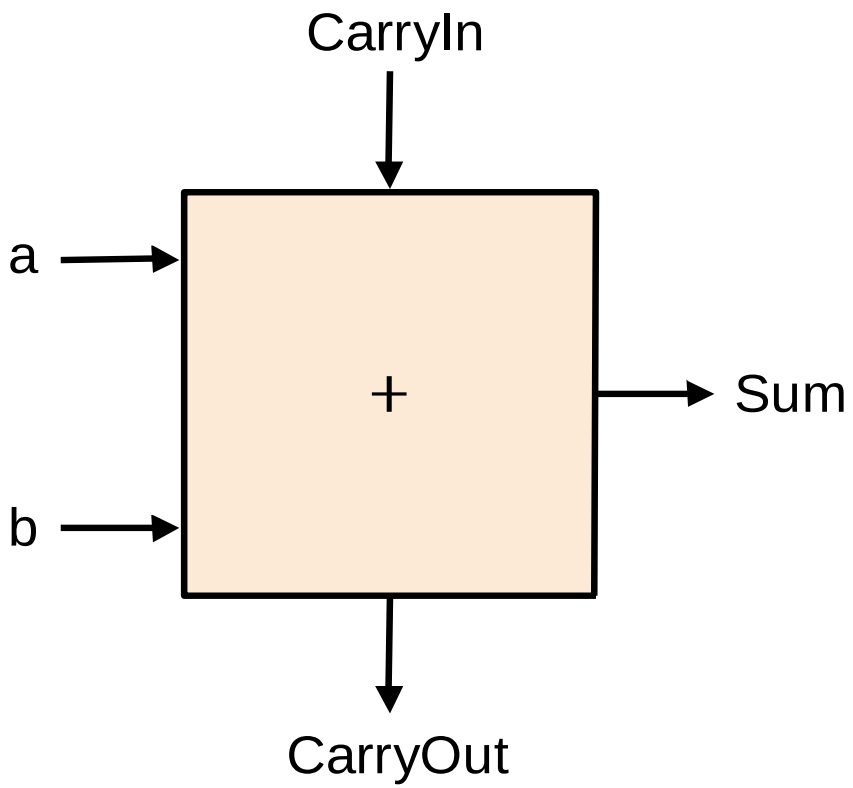
Multiplexor (Data Selector; Abstraction)

- 2-to-1 MUX, 4-to-1 MUX (c.f., Demultiplexer)





1-bit Full Adder Design (디지털논리설계)

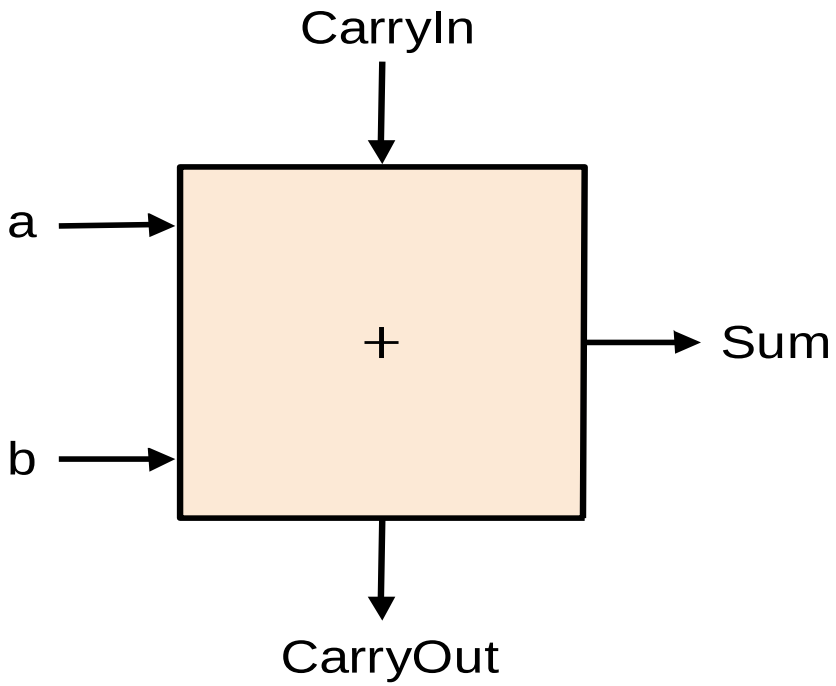


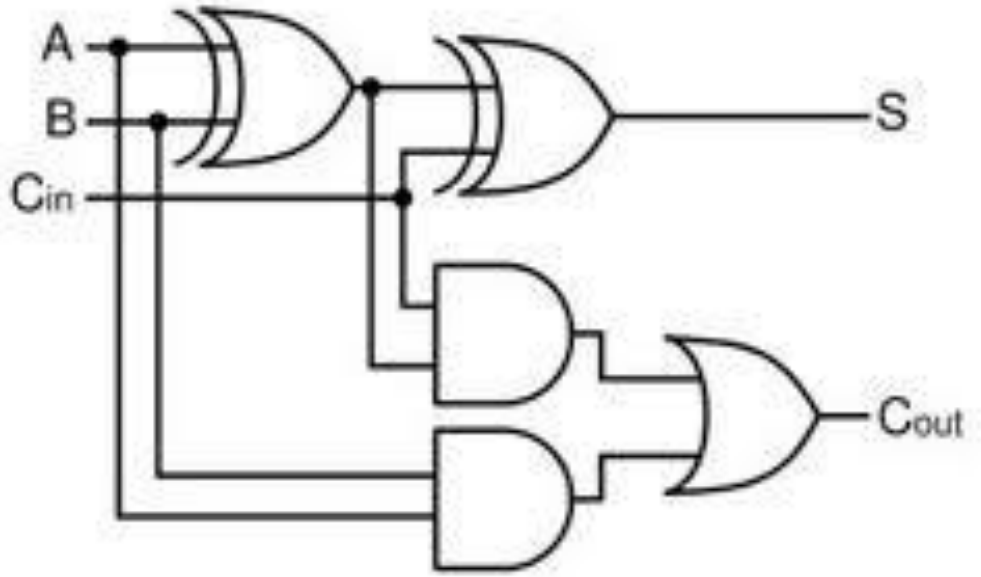
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

$$sum = a \text{ xor } b \text{ xor } c_{in}$$

1-bit Full Adder Design (Abstraction)





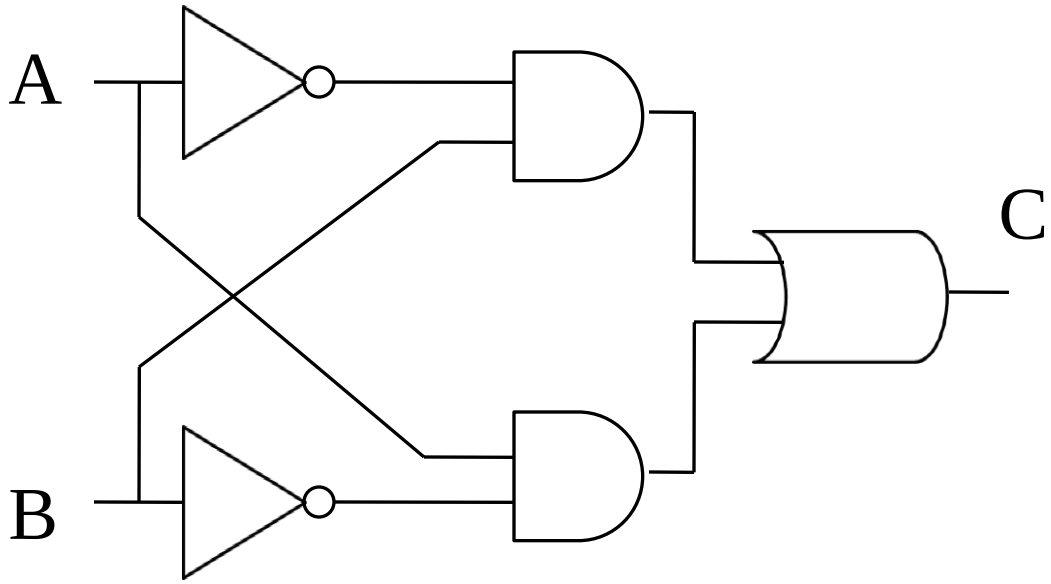
$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

$$sum = a \text{ xor } b \text{ xor } c_{in}$$

XOR (Exclusive-OR) Gate (Abstraction)

- $C = A \text{ XOR } B = A \oplus B$

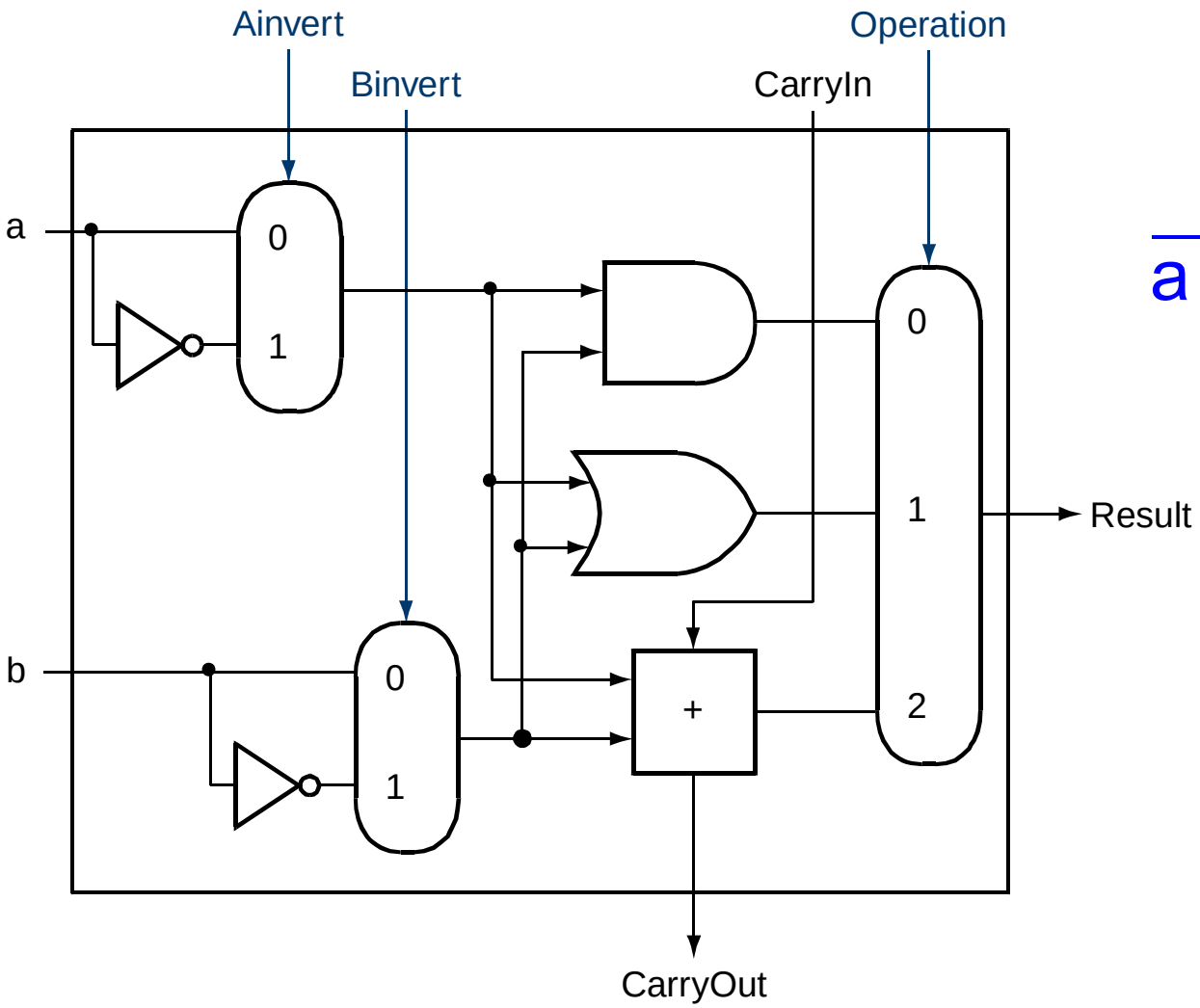
p	q	p ⊕ q
1	1	0
1	0	1
0	1	1
0	0	0



$$C = A \cdot \underline{B} + \underline{A} \cdot B$$

Adding a NOR function

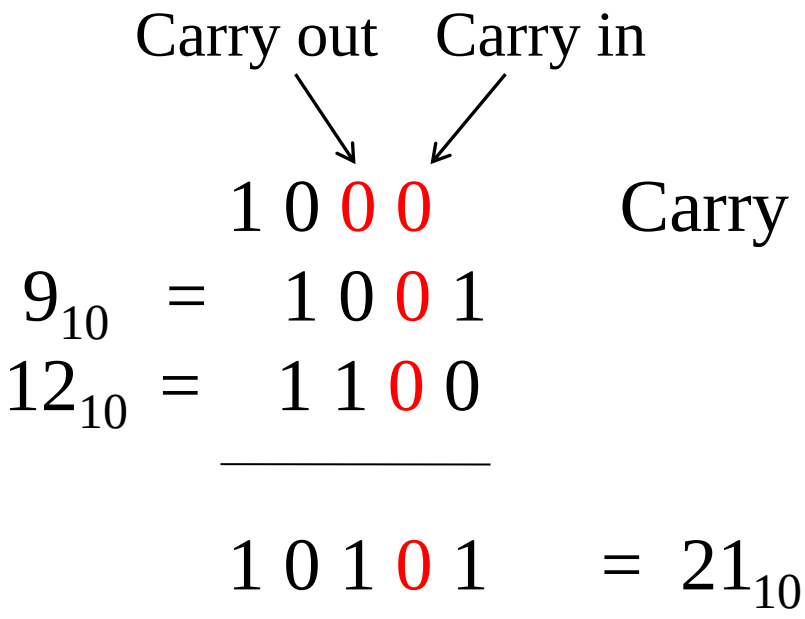
- Can also choose to invert **a**. How do we get "**a** NOR **b**" ?



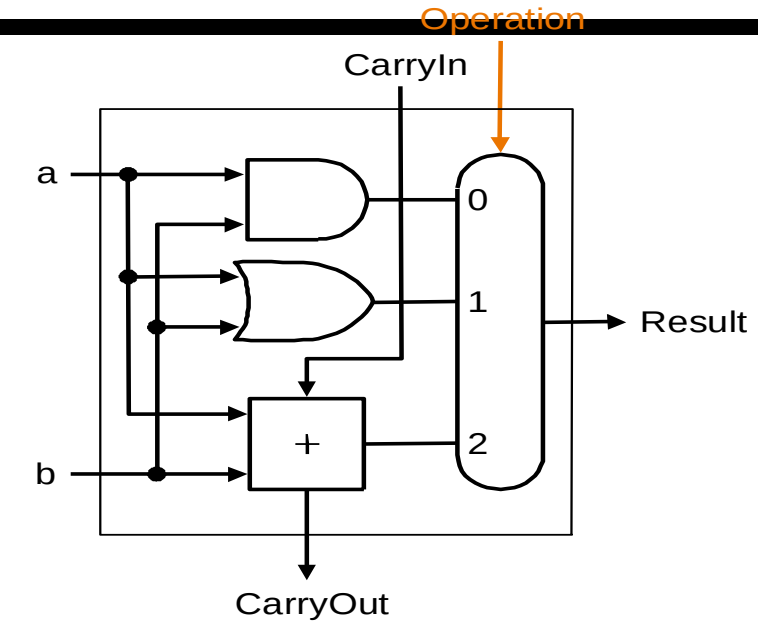
$$\overline{a \text{ or } b} = \overline{a} \text{ and } \overline{b}$$

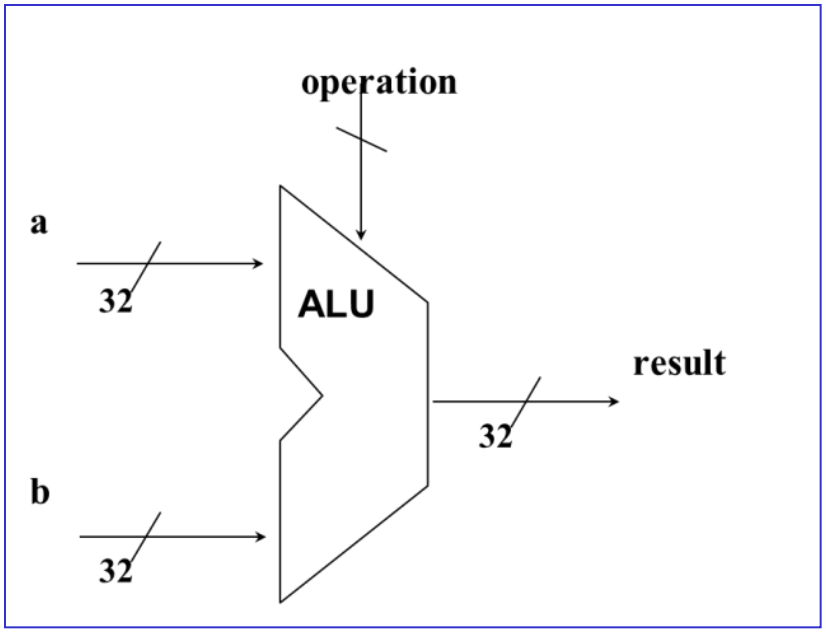
Add Binary Numbers

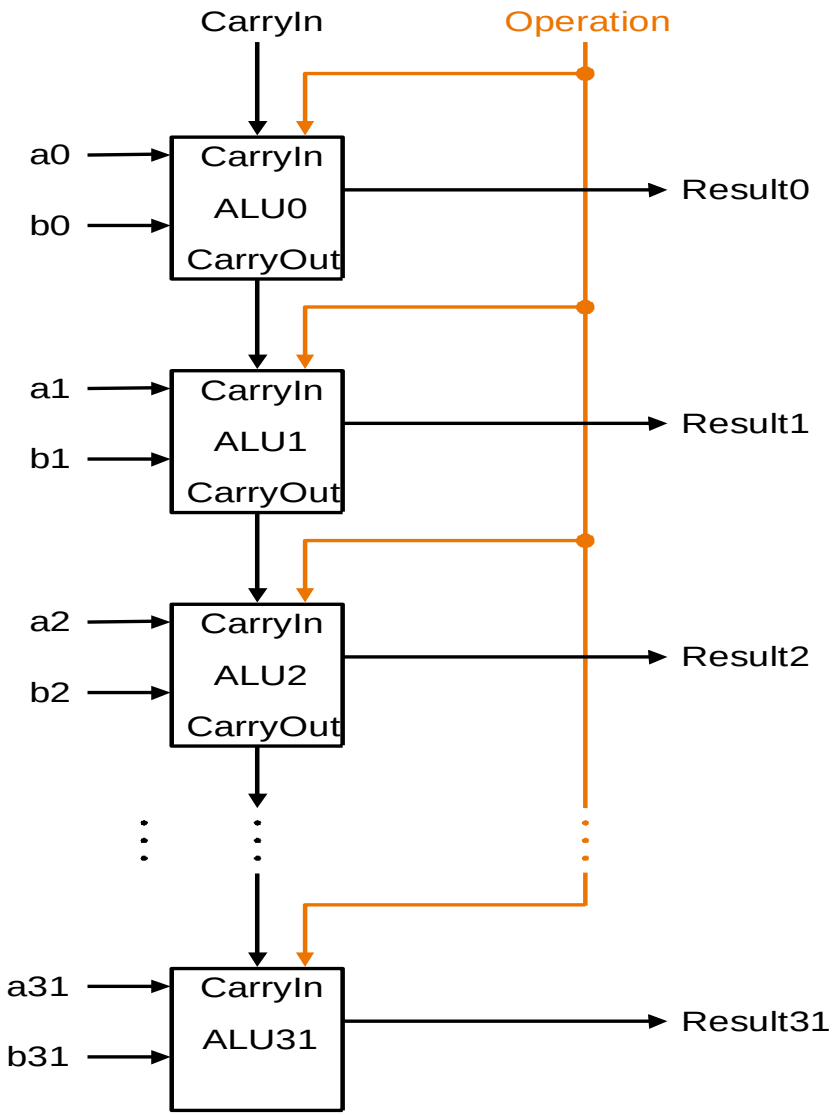
- Same with decimal add



32-bit ALU Design (Abstraction)

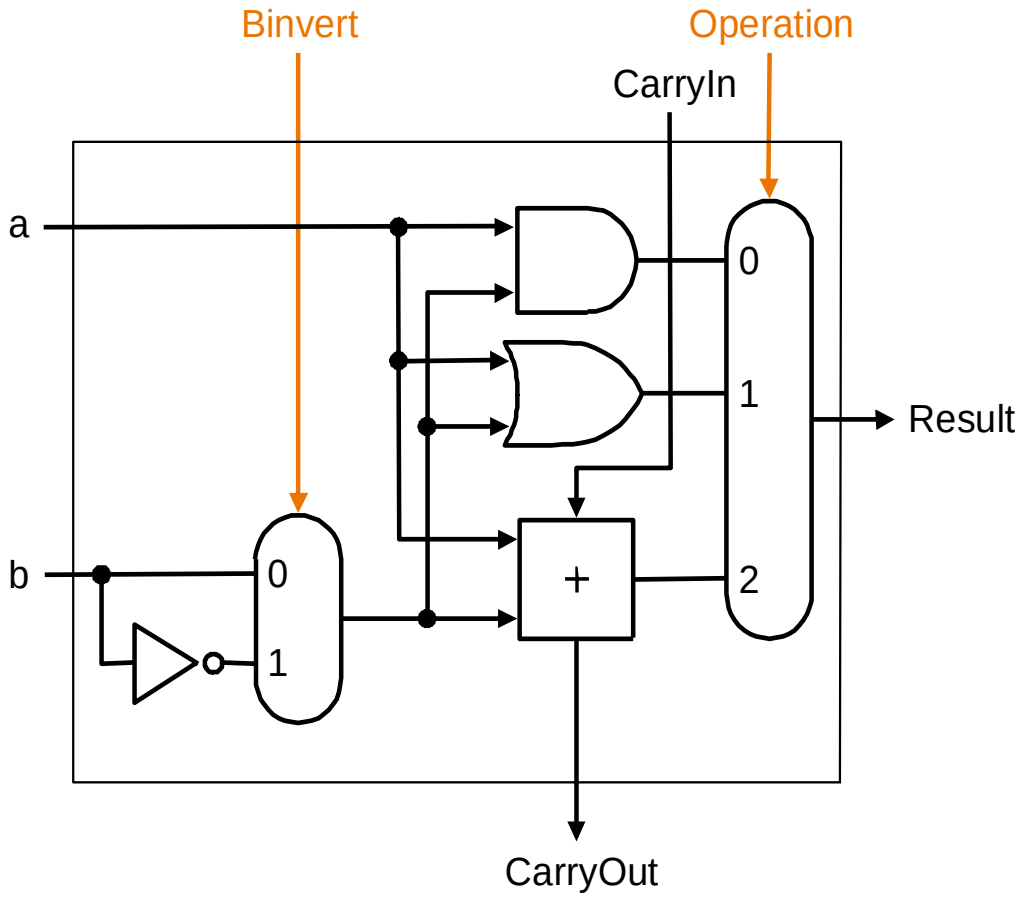






What about subtraction (a – b) ?

- Two's complement approach: negate **b** and add
  - Negate: bitwise NOT, then add 1

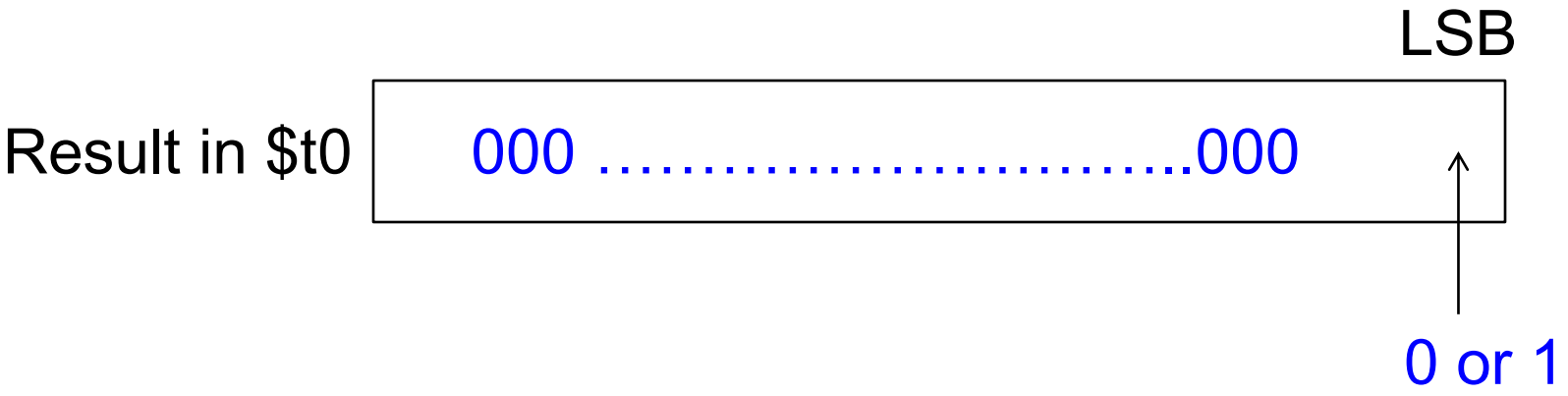


and, or, +, -

Tailoring the ALU to the MIPS

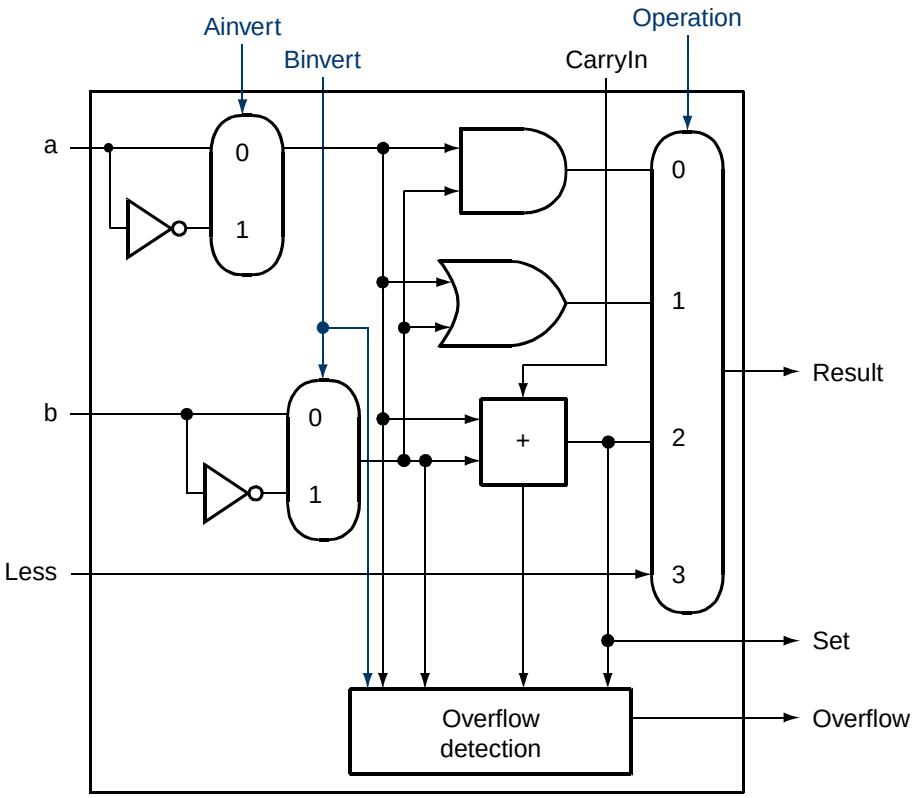
- Need to support the set-on-less-than instruction (**slt**)
  - Remember: **slt** is an arithmetic instruction
  - Produces a 1 if rs < rt and 0 otherwise
  - Use subtraction: (a-b) < 0 implies a < b

slt $t0, $t1, $t2

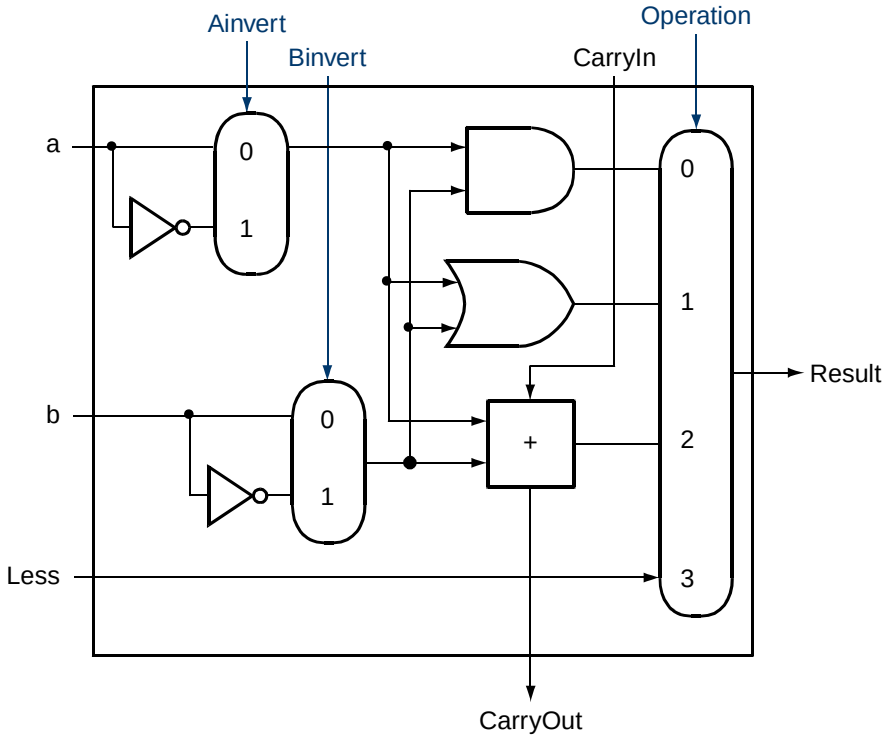


Supporting slt

- Can we figure out the idea?

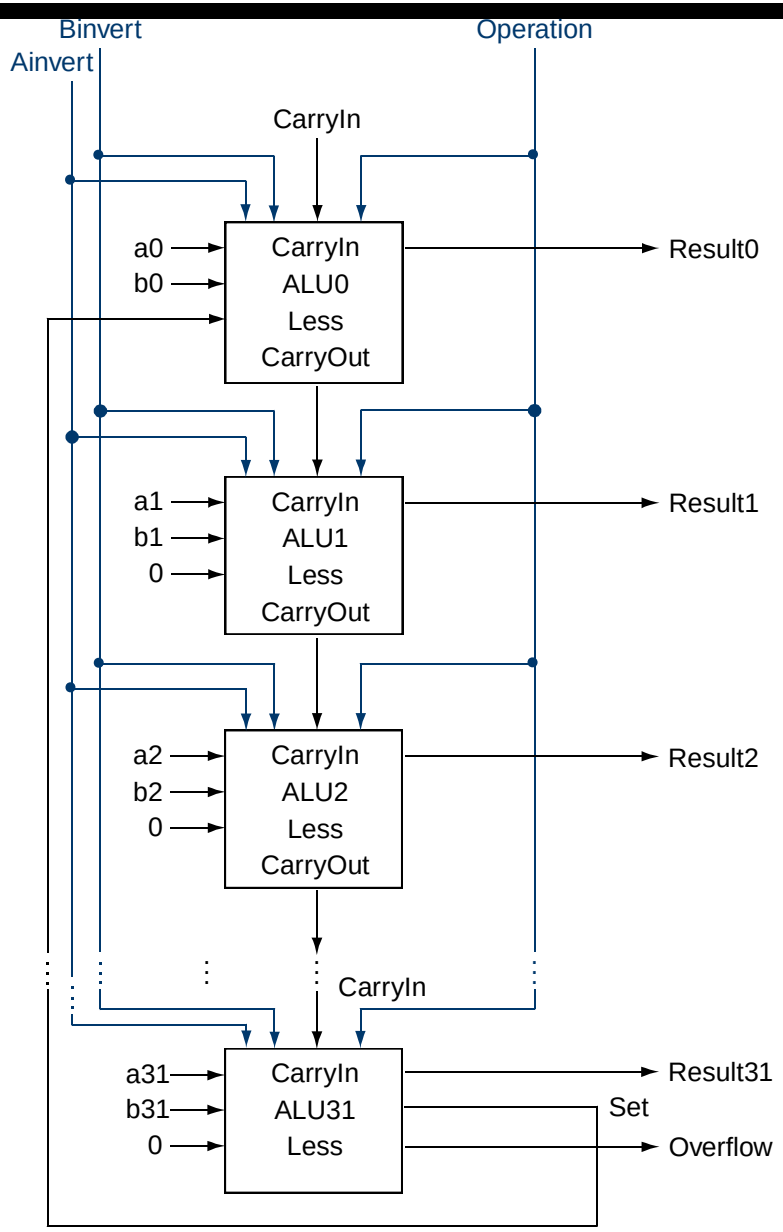


Adding most significant bits



Adding all other bits

Supporting slt



slt:

1) MUX는 less 선택
2) Bits 1-31: always 0
3) Bit 0: 0 or 1
   (크기비교 결과)

두 수 뺀후 MSB 넣으면 된다.
1. a<b이면 음수 MSB 1
2. a>=b이면 양수 or 0 MSB 0

Tailoring the ALU to the MIPS

- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

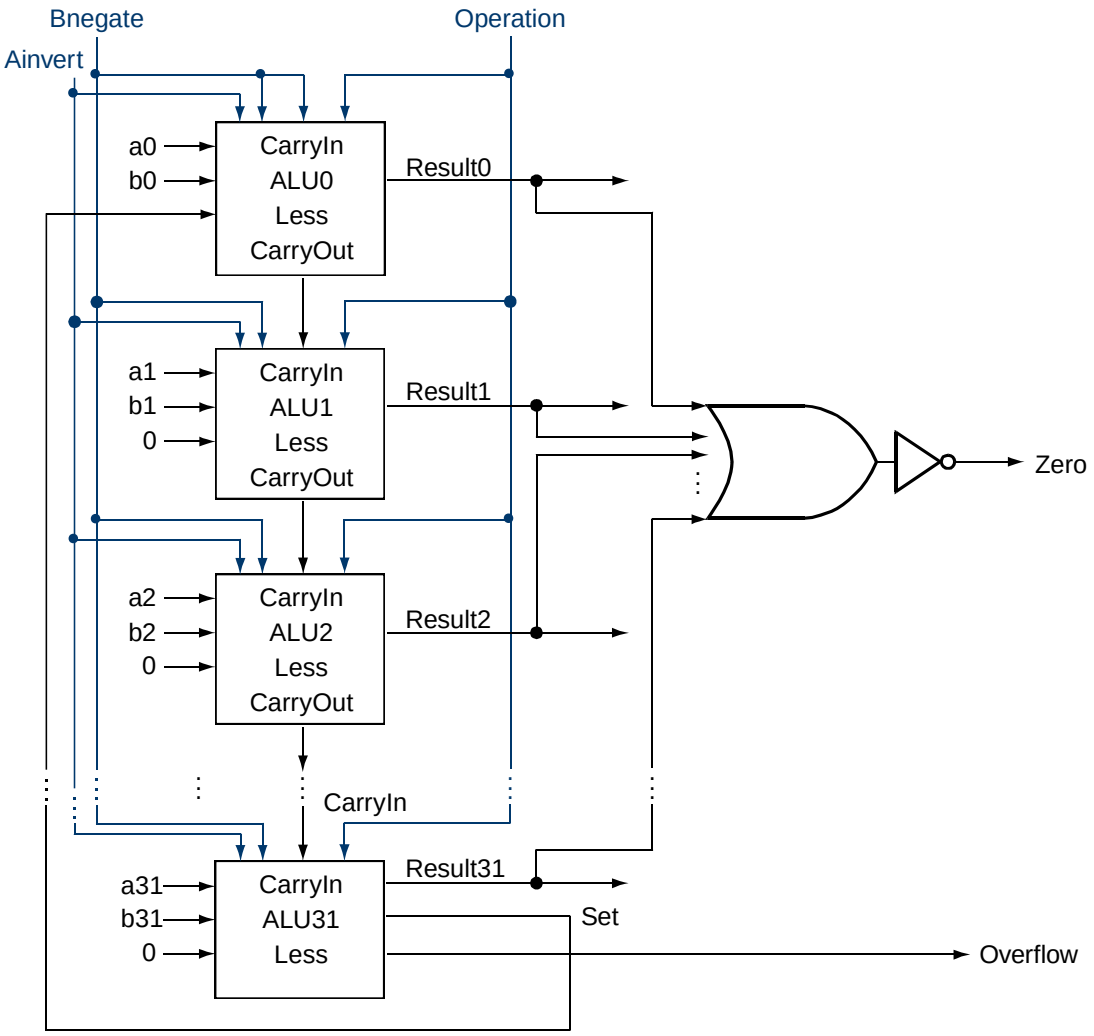
    느리기 때문에 대신, 각각의 비트가 같은지 확인

Test for equality

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

† *Note: zero is a 1 when the result is zero!*



Conclusion

- We can build an ALU to support the MIPS instruction set
  - Key idea: use multiplexor to select the output we want
  - Can efficiently perform subtraction using 2's complement
  - Can replicate a 1-bit ALU to produce a 32-bit ALU
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - Will see this in multiplication, let's look at addition now

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - Two extremes: ripple carry and sum-of-products

† Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

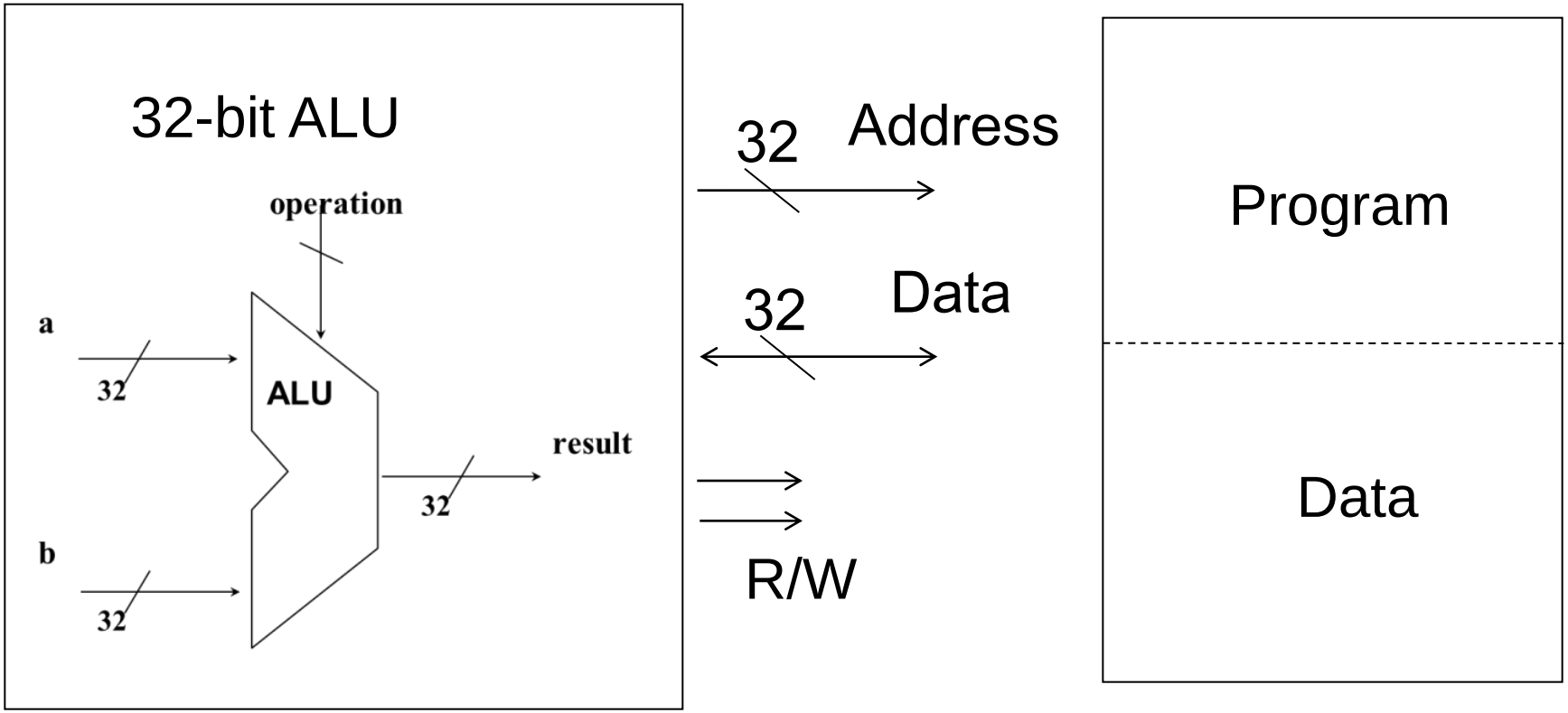
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$$

Not feasible! Why?

32-bit Computer



I/O: Monitor/keyboard, LAN-Internet, …

Computer Arithmetic and ALU

- Can you imagine
  - Many algorithms for addition,
    - Multiplication, division, FP operations as well
- Focus of building computer in 1945
  - Faster ALU
    - How to represent numbers
- Computer arithmetic
  - Matured in 1950s and 1960s
- Today, can buy ALU as IP (Intellectual Property)

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data), SSE
    - Subword parallelism
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - e.g., clipping in audio, saturation in video

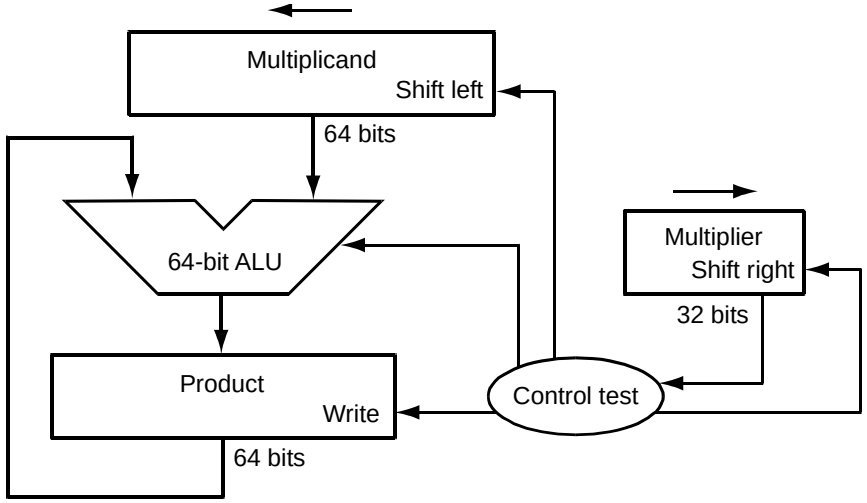
Multiplication

- More complicated than addition
  - Accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on a gradeschool algorithm

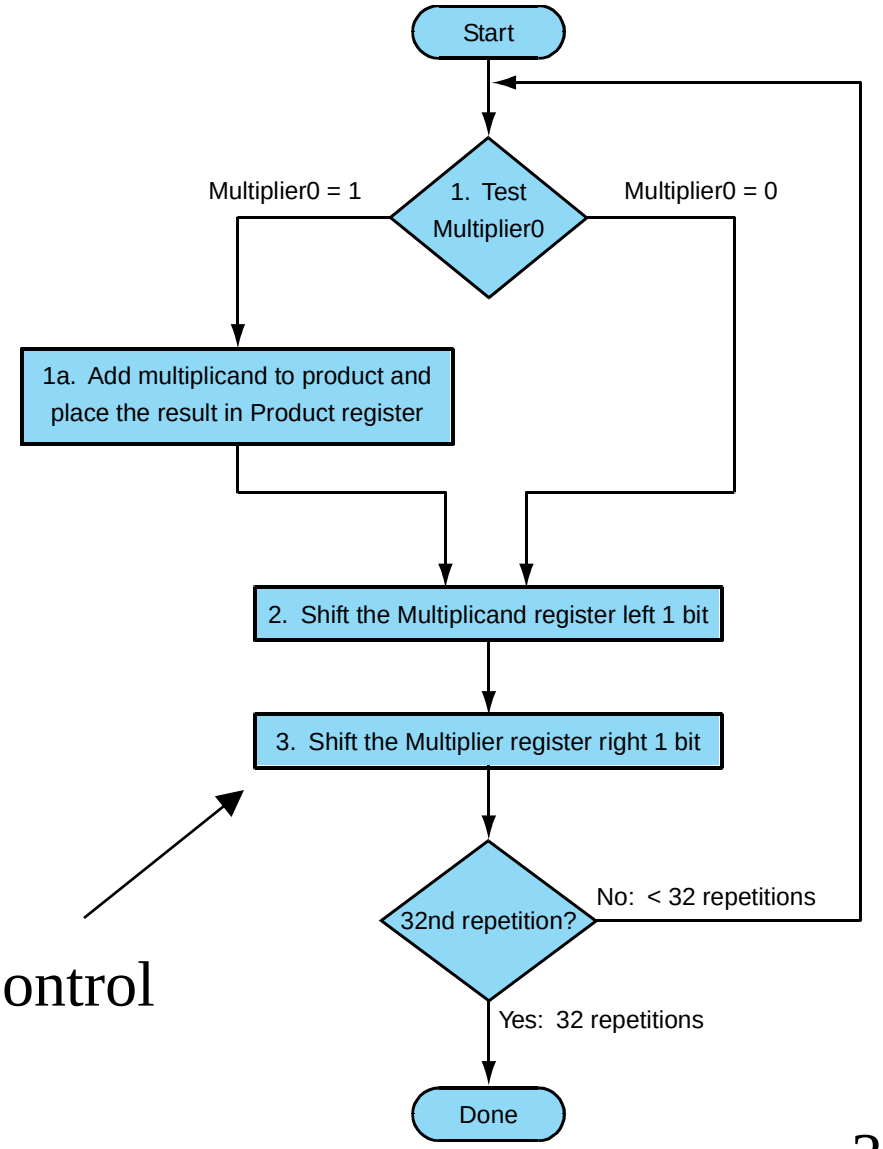
```
   0010   (multiplicand)
 x 1011   (multiplier)
```

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

Multiplication: Implementation



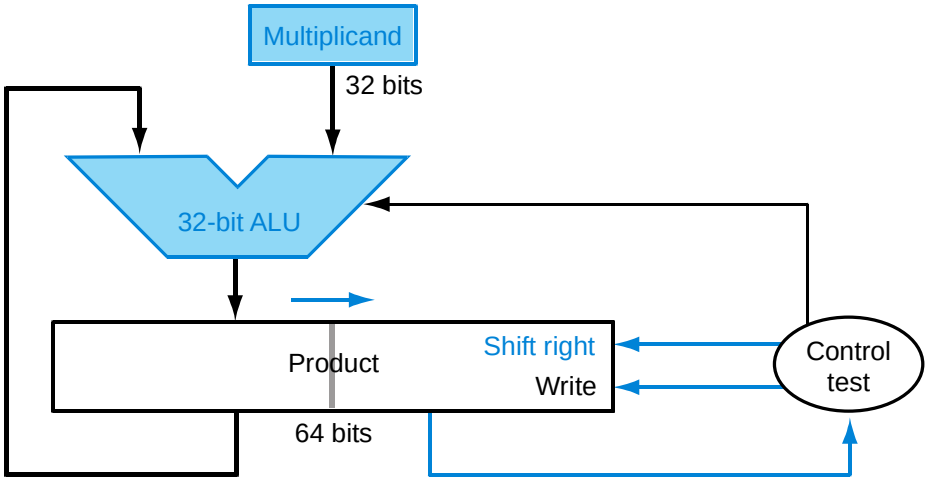
Datapath

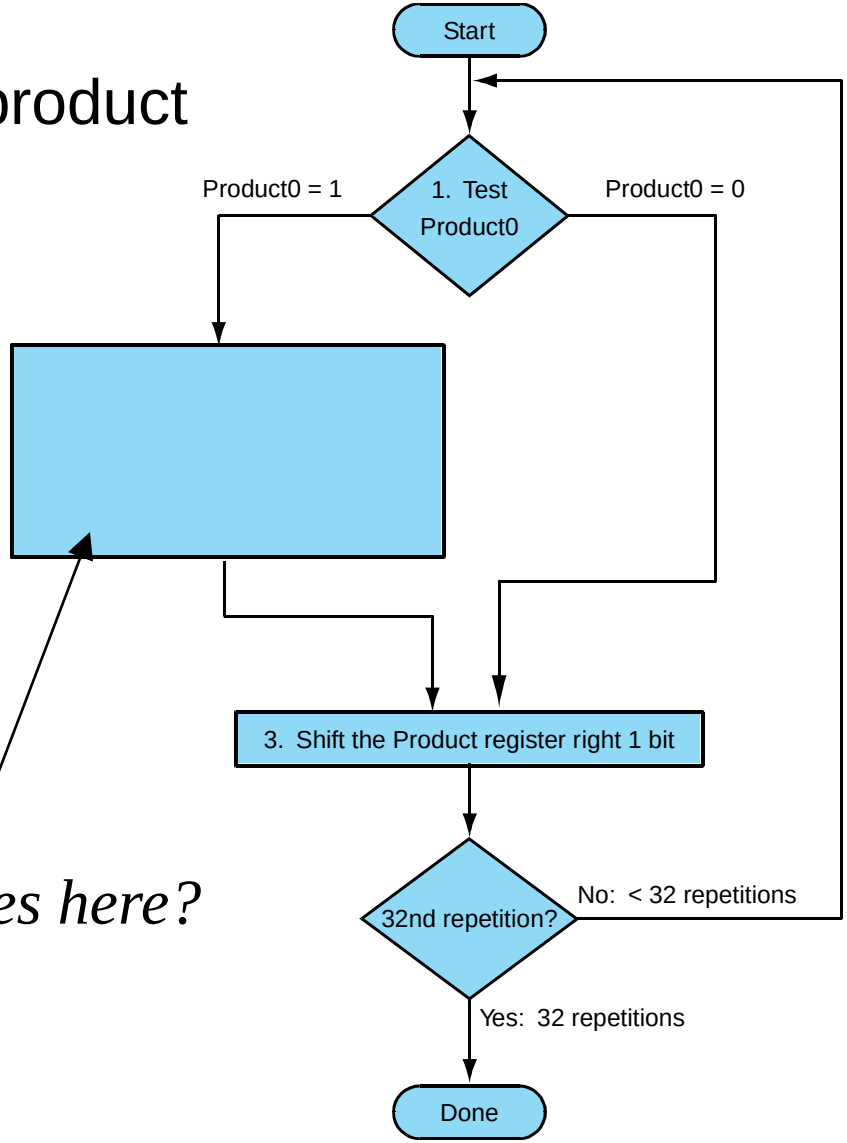


Control

Final Version

- Multiplier starts in right half of product





What goes here?

MIPS Assembly Language: multiply, divide

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

Floating-Point Numbers

(Why the name? Fixed-point numbers?)

Floating Point (a brief look)

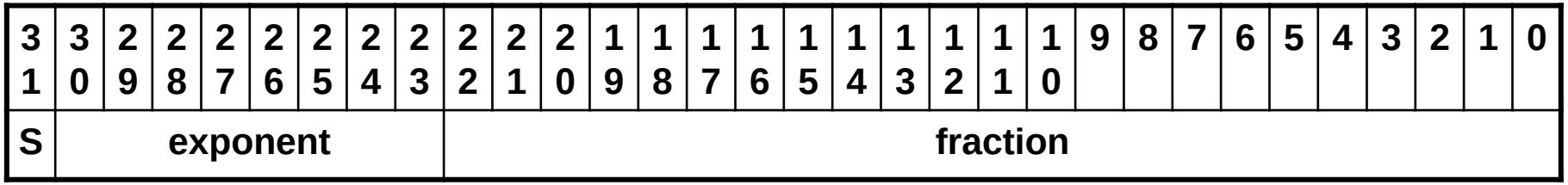
- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., 0.71×10^{-78}
  - very large numbers, e.g., 3.15576×10^{69}
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \cdot significand \cdot 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

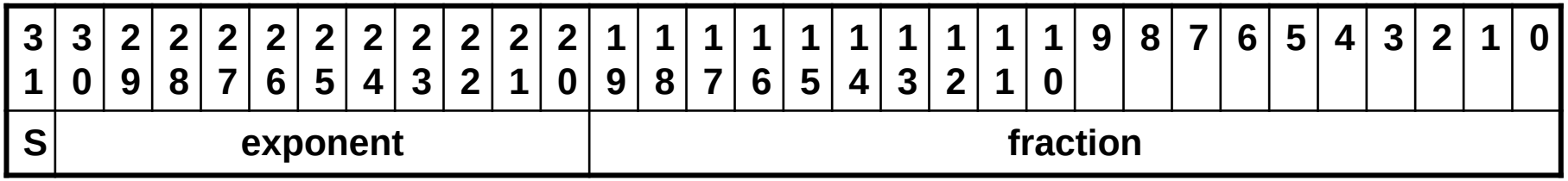
- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent all 1s is largest
    - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \cdot (1+significand) \cdot 2^{exponent - bias}$
- Example:
  - decimal: -.75 = - (½ + ¼)
  - binary: -.11 = -1.1 x 2^{-1} normalized된 표현 implicit 1 (bit 수 절약)
  - floating point: exponent = 126 = 01111110
  - IEEE single precision:

```
10111111 01000000 00000000 00000000
```

IEEE 754 floating-point standard



1 bit 8 bits 23 bits



1 bit 11 bits 20 bits

32 bits

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

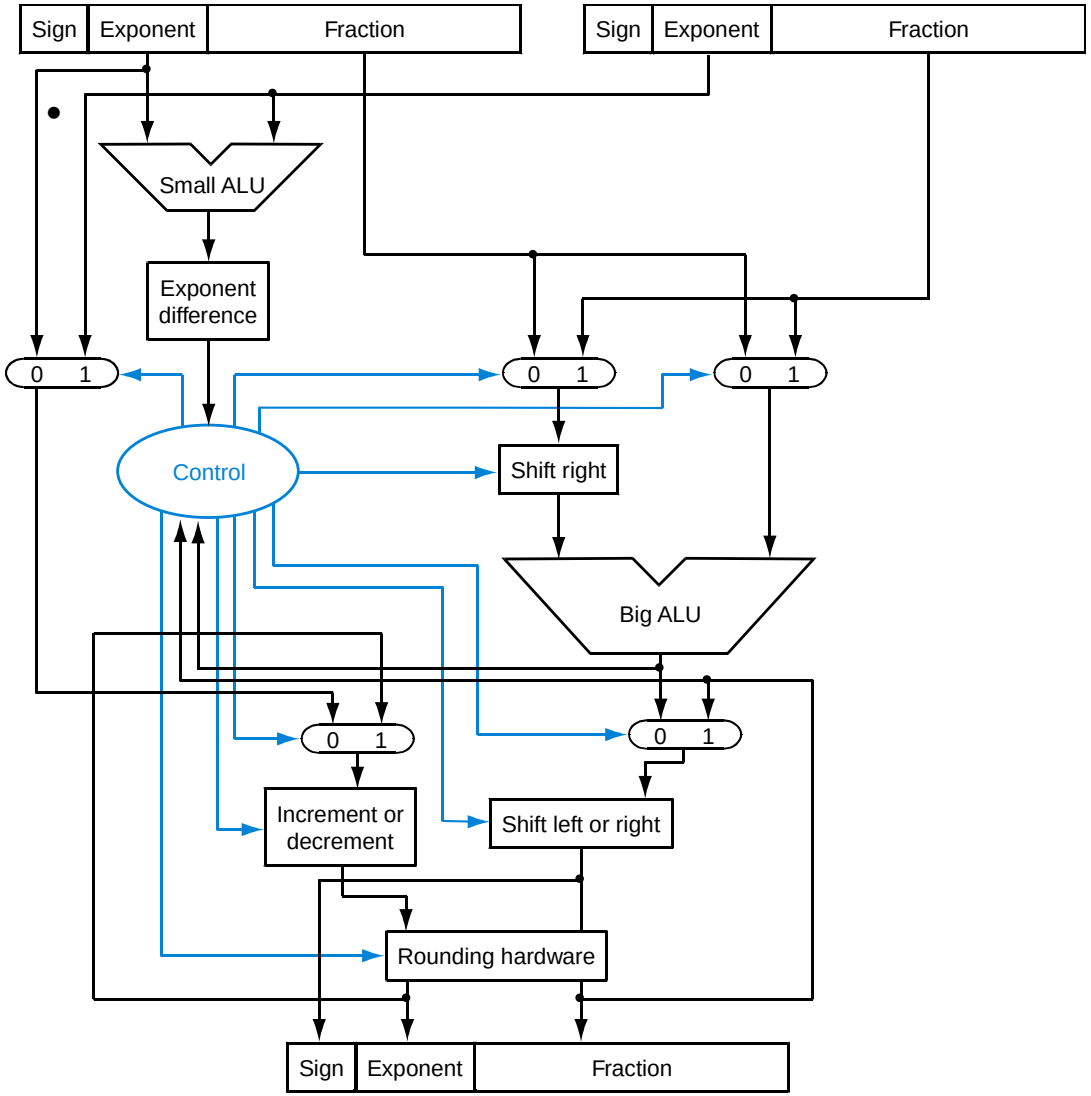
Double-Precision Range

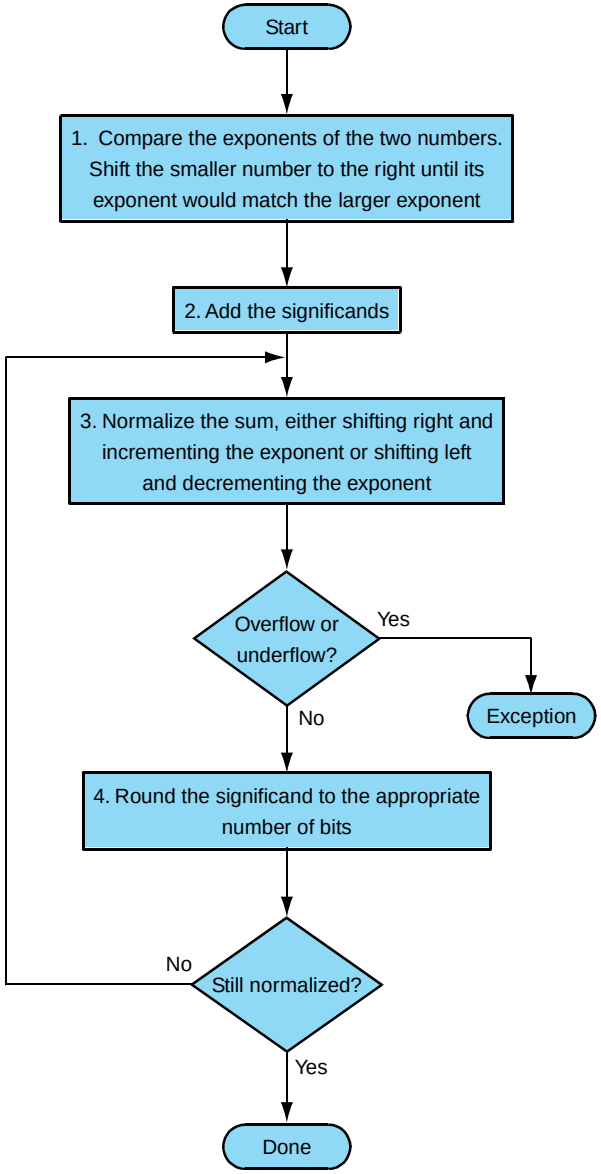
- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

IEEE 754 floating point standard

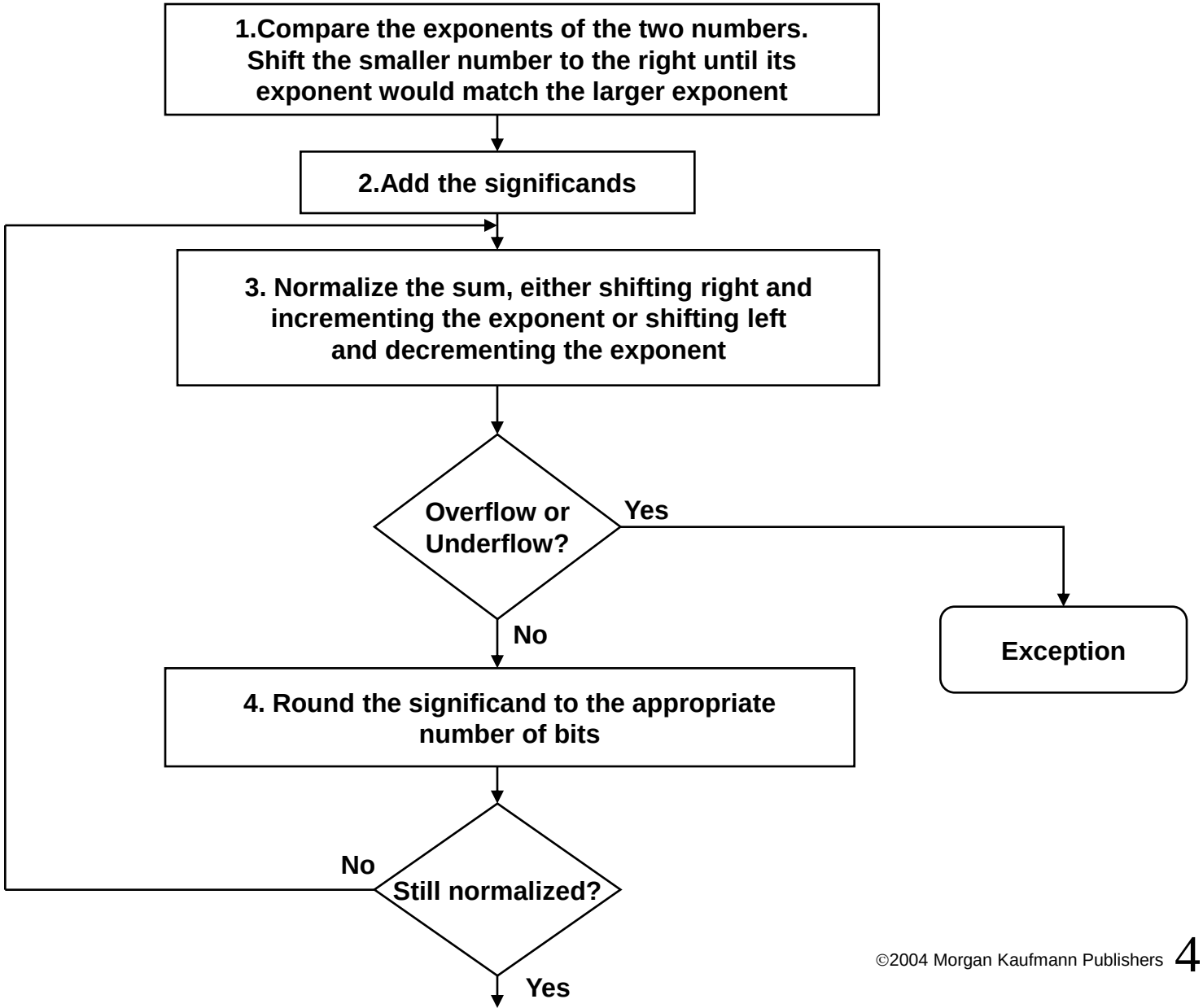
- Why do we use floating point representation?
  - Very large numbers, very small numbers
- What do we lose?
  - Accuracy, intervals not uniform
- Think about overflow, underflow
- In 32-bit standard, how many bits for significand?
  - Range vs. accuracy trade-off
- How do we represent 0, infinity?
  - Exp = 255, sig != 0 → NaN
  - Exp = 255, sig = 0 → +, - infinity
  - Exp = 0, sig = 0 → 0
  - Exp =0, sig != 0 → even smaller numbers

Floating point addition





Floating point addition

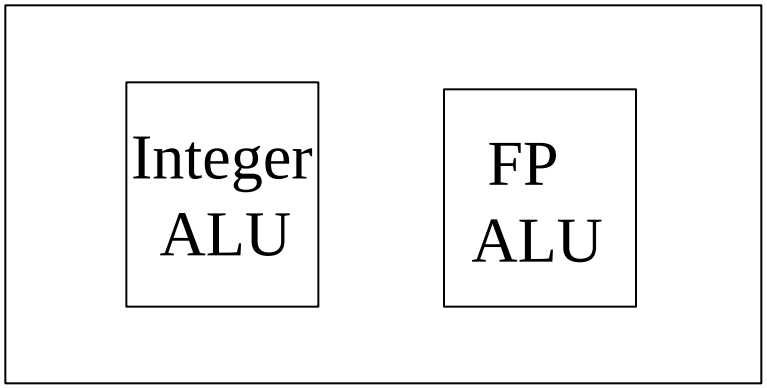


FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

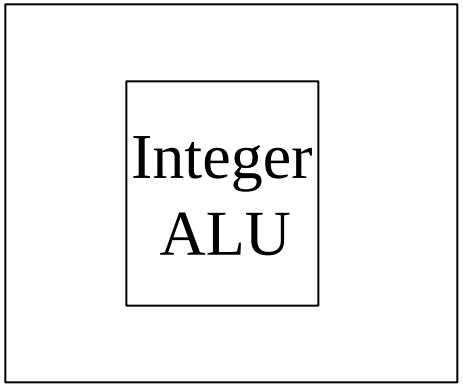
Integer and FP ALUs (반복)

- Processor for general-purpose computers:



Pentium

- Processors for embedded systems:



Smaller
Cheaper
Low-power

MIPS Assembly Language: floating-point

integer 연산과 구별하기 위해서 floating-point register 존재 홀수 번 - single 짝수 번 - double

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS Assembly Language: floating-point

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

To Think about

표준이 만들어졌을때 double 계산이 부담되었었다.

- When you do C, Java programming
  - When do you use **float**? imbedded system
  - When do you use **double**? 기본이 double 기계가 powerful 하기 때문에 가능
  - Do you ever think about long integer or arithmetic overflow?
- FP numbers and FP ALU
  - 계산을 위한 인공적인 고안의 결과
  - No beauty of integers
    - Not accurate from mathematics perspective

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism – numerical analysis
- FP numbers only approximations of real numbers

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Sections 3.6

Parallelism and Computer Arithmetic: Subword Parallelism

Sections 3.7

Real stuff: streaming SIMD extensions and advanced vector extensions in x86

Sections 3.8

Going faster: subword parallelism and matrix multiply

Sections 3.10

Concluding Remarks

(가볍게 읽어 보실 것)

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

MIPS Core, MIPS-32, Pseudo MIPS

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

Figure 3.26

MIPS IPS

Remaining MIPS-32	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R
exclusive or immediate	xori	I
shift right arithmetic	sra	R
shift left logical variable	sllv	R
shift right logical variable	srlv	R
shift right arithmetic variable	srav	R
move to Hi	mthi	R
move to Lo	mtlo	R
load halfword	lh	I
load byte	lb	I
load word left *(unaligned)*	lwl	I
load word right *(unaligned)*	lwr	I
store word left *(unaligned)*	swl	I
store word right *(unaligned)*	swr	I
load linked (*atomic update*)	ll	I
store cond. (*atomic update*)	sc	I
move if zero	movz	R
move if not zero	movn	R
multiply and add (S or *uns.*)	madds	R
multiply and subtract (S or *uns.*)	msubs	I
branch on ≥ zero and link	bgezal	I
branch on < zero and link	bltzal	I
jump and link register	jalr	R
branch compare to zero	bxz	I
branch compare to zero likely	bxzl	I
(x = lt, le, gt, ge)		
branch compare reg likely	bxl	I
trap if compare reg	tx	R
trap if compare immediate	txi	I
(x = eq, neq, lt, le, gt, ge)		
return from exception	rfe	R
system call	syscall	I
break *(cause exception)*	break	I
move from FP to integer	mfc1	R
move to FP from integer	mtc1	R
FP move (*s* or *d*)	mov.*f*	R
FP move if zero (*s* or *d*)	movz.*f*	R
FP move if not zero (*s* or *d*)	movn.*f*	R
FP square root (*s* or *d*)	sqrt.*f*	R
FP absolute value (*s* or *d*)	abs.*f*	R
FP negate (*s* or *d*)	neg.*f*	R
FP convert (*w*, *s*, or *d*)	cvt.*f*.*f*	R
FP compare un (*s* or *d*)	c.xn.*f*	R

Pseudo MIPS	Name	Format
absolute value	abs	rd,rs
negate *(signed or unsigned)*	negs	rd,rs
rotate left	rol	rd,rs,rt
rotate right	ror	rd,rs,rt
multiply and don't check oflw *(signed or uns.)*	muls	rd,rs,rt
multiply and check oflw *(signed or uns.)*	mulos	rd,rs,rt
divide and check overflow	div	rd,rs,rt
divide and don't check overflow	divu	rd,rs,rt
remainder *(signed or unsigned)*	rems	rd,rs,rt
load immediate	li	rd,imm
load address	la	rd,addr
load double	ld	rd,addr
store double	sd	rd,addr
unaligned load word	ulw	rd,addr
unaligned store word	usw	rd,addr
unaligned load halfword *(signed or uns.)*	ulhs	rd,addr
unaligned store halfword	ush	rd,addr
branch	b	Label
branch on equal zero	beqz	rs,L
branch on compare *(signed or unsigned)*	bxs	rs,rt,L
(x = lt, le, gt, ge)		
set equal	seq	rd,rs,rt
set not equal	sne	rd,rs,rt
set on compare *(signed or unsigned)*	sxs	rd,rs,rt
(x = lt, le, gt, ge)		
load to floating point (*s* or *d*)	l.*f*	rd,addr
store from floating point (*s* or *d*)	s.*f*	rd,addr

Figure 3.27

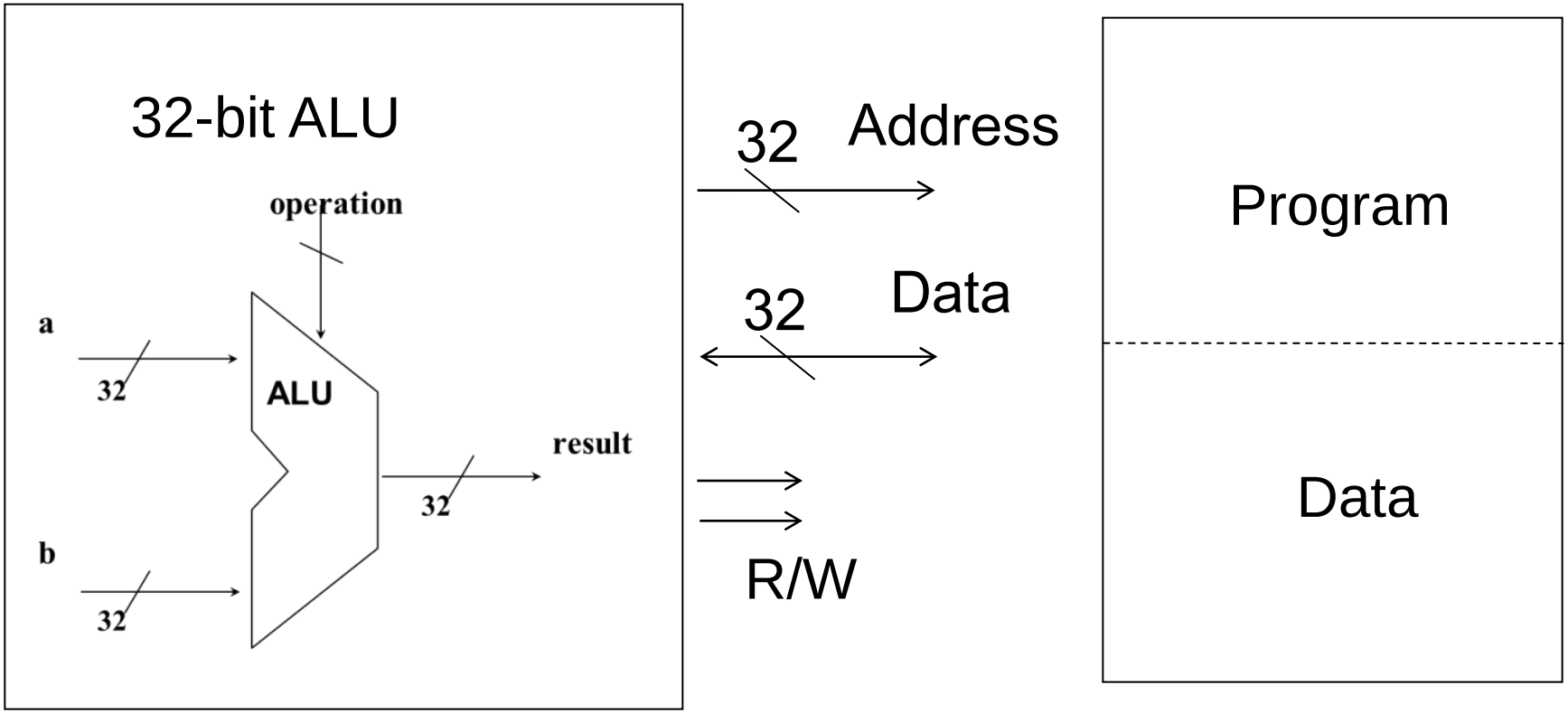
MIPS Core, MIPS-32, Pseudo MIPS

- Frequency of use in SPEC CPU2006 benchmark

Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

- For the rest of book, focus on MIPS core (integer instruction set excluding multiply and divide)
  - To make the explanation of computer design easier

32-bit Computer



I/O: Monitor/keyboard, LAN-Internet, …

Will implement fetch-decode-execute in Chapter 4