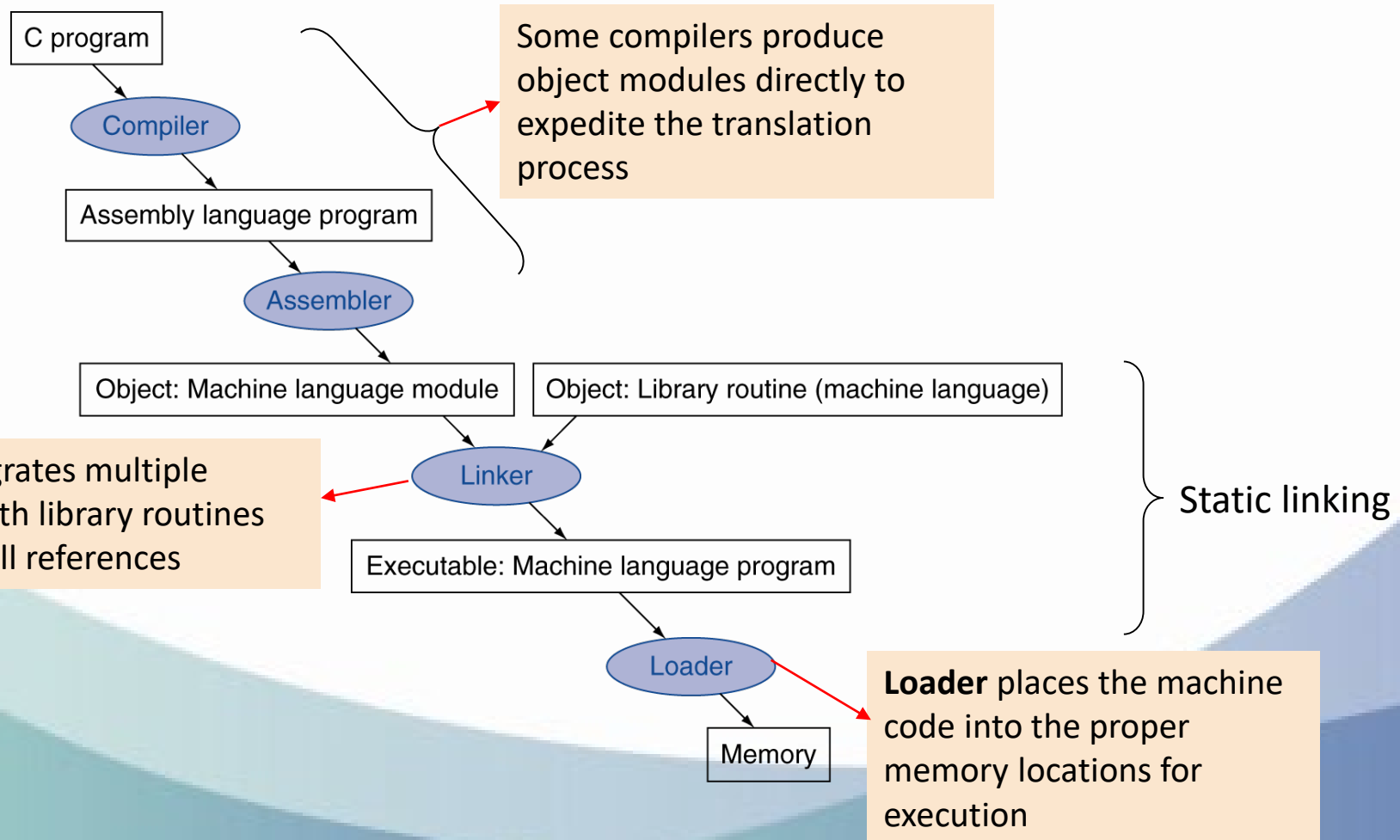


Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

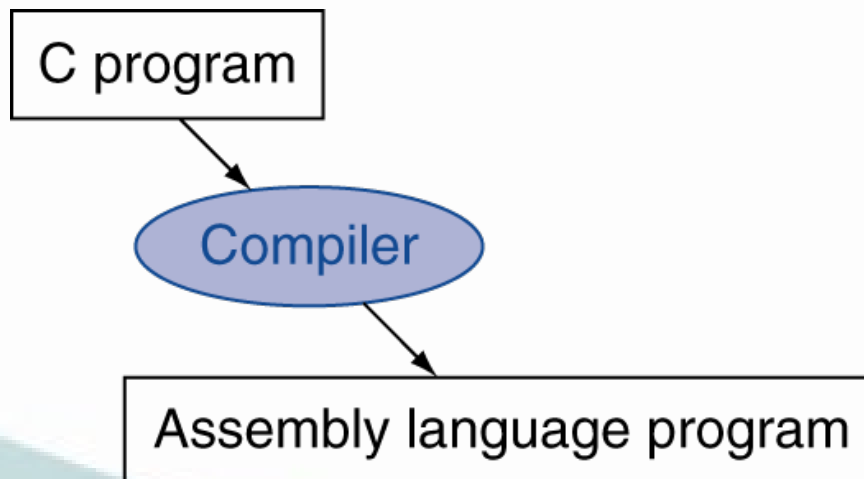
DEPT. OF COMPUTER SCIENCE, NCCU

Translating and Starting a Program



Compiler

- In 1975, many operating systems and assemblers were written in assembly language due to small memories and slow compilers.
- Nowadays, the compiler transforms a high-level program (e.g. c program) into an assembly language program



Assembler

- An interface to higher-level software
- The assembler can deal with common variations of machine language instructions

- E.g. The MIPS assembler accepts some instructions even though it is not found in the MIPS architecture
 - `move $t0, $t1` **# \$t0 ← \$t1** will be converted into `addi $t0, $zero, $t1` **# \$t0 ← 0+\$t1**

- So, we have

- `blt` = `slt` + `bne`
- `bgt`, `bge`, `ble`, etc....

`blt $t0, $t1, L`



`slt $at, $t0, $t1`
`bne $at, $zero, L`
\$at (register 1): assembler temporary

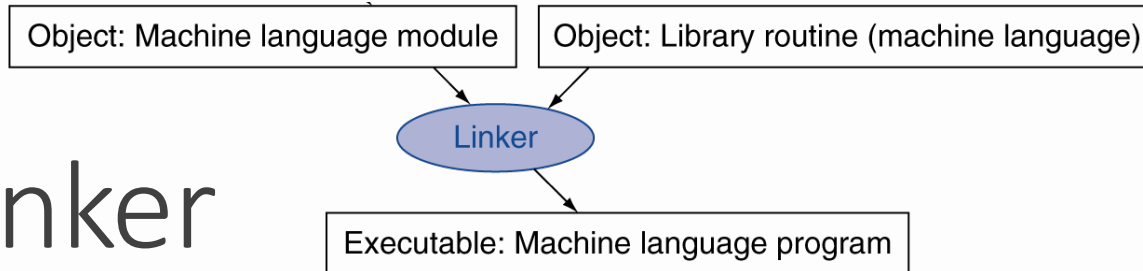
Assembly language program

Assembler

Assembler needs to determine the addresses of all used labels

Object: Machine language module

Object: Library routine (machine language)



Linker

- To avoid a fact that a single change to one line of one procedure requires re-compiling and re-assembling the entire program
- Producing an executable image
 - Merging segments
 - Resolving labels (filling up their addresses)
 - Patching location-dependent and external refs
- Compile and assemble each procedure independently
 - Linker takes all the independently assembled programs and link them together
 - Note that the assembler cannot know where the independently programs are

Steps:

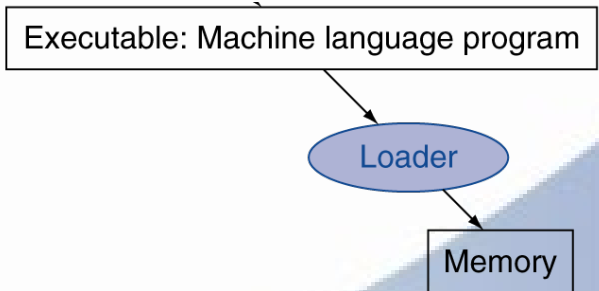
1. Place code and data modules symbolically in memory
2. Determine the addresses of data and instruction labels
3. Patch both the internal and external references

Producing Object File

- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Loading a Program

- Load an executable file into memory
 1. Read segment sizes from header
 2. Find address space (could be virtual)
 3. Copy text and initialized data into memory (in virtual memory, set page table entries null to incur a page fault, so data can be moved in)
 4. Set up arguments on stack
 5. Initialize registers (including \$sp, \$fp, \$gp)
 6. Jump to startup routine
 - Copies arguments to \$a0, ... and calls main
 - When main returns, call the exit syscall



Link the two object files

An object file contains:

1. Object file header
 - Size and position of the file
2. Text segment
 - Machine language code
3. Static data
 - Data allocated
4. Relocation info
 - Instructions and data words that depends on absolute addresses
5. Symbol table
 - Labels (external references)

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

What the linker does is to edit object files and make them an executable file

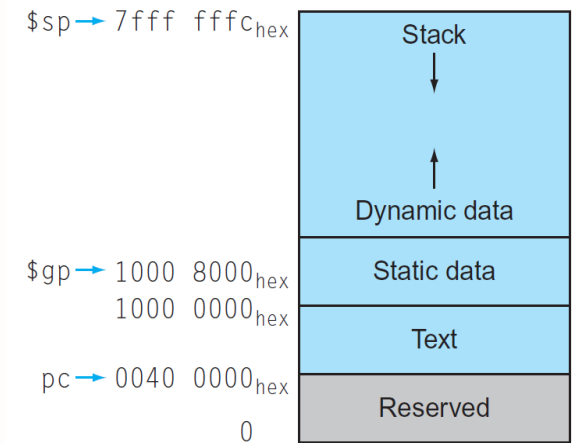
Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
	0040 0104 _{hex}	jal 40 0000 _{hex}

Data segment	Address	
	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)

X
B
Y
A



lw \$a0, 8000(\$gp)

#1000 8000 + 8000 (negative) = 1000 8000 (gp) + FFFF8000 = 1000 0000

Integer Subtraction

- Add negation of second operand
- Example: $7 - 6 = 7 + (-6)$

$$\begin{array}{r}
 \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_{\text{two}} = 7_{\text{ten}} \\
 - \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

$$\begin{array}{r}
 \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0111_{\text{two}} = 7_{\text{ten}} \\
 + \quad 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1010_{\text{two}} = -6_{\text{ten}} \\
 \hline
 = \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

Dynamic Linking

■ Motivation

- The library routines become part of the executable code, so if a patch/an update is released, the statically linked program keeps using the old version
- It loads all routines in the library even though they may not be executed (called)
- These lead to dynamically linked libraries (DLLs)

■ Only link/load library procedure when it is called

- Requires procedure code to be relocatable
- Avoids image bloat caused by static linking of all (transitively) referenced libraries
- Automatically picks up new library versions

More Examples: Sorting

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in \$a0, k in \$a1, temp in \$t0

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                               # (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in \$a0, k in \$a1, i in \$s0, j in \$s1

The Procedure Body

	move \$s2, \$a0	# save \$a0 into \$s2	Move params
	move \$s3, \$a1	# save \$a1 into \$s3	
for1tst:	move \$s0, \$zero	# i = 0	Outer loop
	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
for2tst:	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	Inner loop
	addi \$s1, \$s0, -1	# j = i - 1	
	slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 (j < 0)	
	bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
	sll \$t1, \$s1, 2	# \$t1 = j * 4	
	add \$t2, \$s2, \$t1	# \$t2 = v + (j * 4)	
	lw \$t3, 0(\$t2)	# \$t3 = v[j]	
	lw \$t4, 4(\$t2)	# \$t4 = v[j + 1]	
	slt \$t0, \$t4, \$t3	# \$t0 = 0 if \$t4 ≥ \$t3	
	beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	Pass params & call
	move \$a0, \$s2	# 1st param of swap is v (old \$a0)	
	move \$a1, \$s1	# 2nd param of swap is j	Inner loop
	jal swap	# call swap procedure	
exit2:	addi \$s1, \$s1, -1	# j -= 1	Outer loop
	j for2tst	# jump to test of inner loop	
	addi \$s0, \$s0, 1	# i += 1	
	j for1tst	# jump to test of outer loop	

Full Procedure

```
sort:      addi $sp,$sp, -20      # make room on stack for 5 registers
           sw $ra, 16($sp)       # save $ra on stack
           sw $s3,12($sp)       # save $s3 on stack
           sw $s2, 8($sp)       # save $s2 on stack
           sw $s1, 4($sp)       # save $s1 on stack
           sw $s0, 0($sp)       # save $s0 on stack
           ...                   # procedure body
           ...
exit1:     lw $s0, 0($sp)        # restore $s0 from stack
           lw $s1, 4($sp)        # restore $s1 from stack
           lw $s2, 8($sp)        # restore $s2 from stack
           lw $s3,12($sp)        # restore $s3 from stack
           lw $ra,16($sp)        # restore $ra from stack
           addi $sp,$sp, 20      # restore stack pointer
           jr $ra               # return to calling routine
```



Arrays vs. Pointers in MIPS

- Array indexing (calculating address)
 - Base address + index*element size
 - A[4]
- Pointers can directly handle memory addresses
 - *(A+4)

Arrays

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

array in \$a0, size in \$a1




```
move    $t0,$zero        # i = 0  
loop1:  sll  $t1,$t0,2      # $t1 = i * 4  
        add $t2,$a0,$t1    # $t2 = &array[i]  
        sw  $zero, 0($t2)  # array[i] = 0  
        addi $t0,$t0,1     # i = i + 1  
        slt $t3,$t0,$a1    # $t3 = (i < size)  
        bne $t3,$zero,loop1 # if (i < size)  
                           # goto loop1
```

Six instructions in the loop

Pointers

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size]; p = p + 1)  
        *p = 0;  
}
```

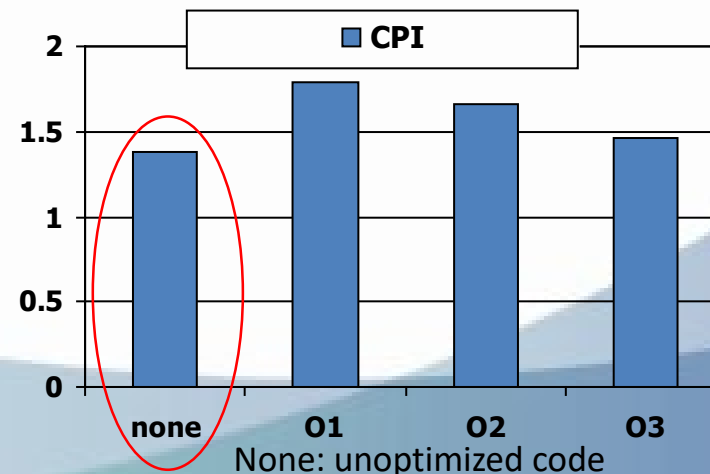
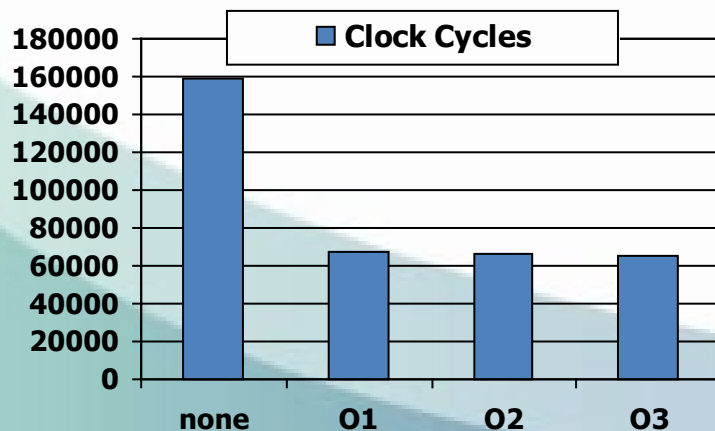
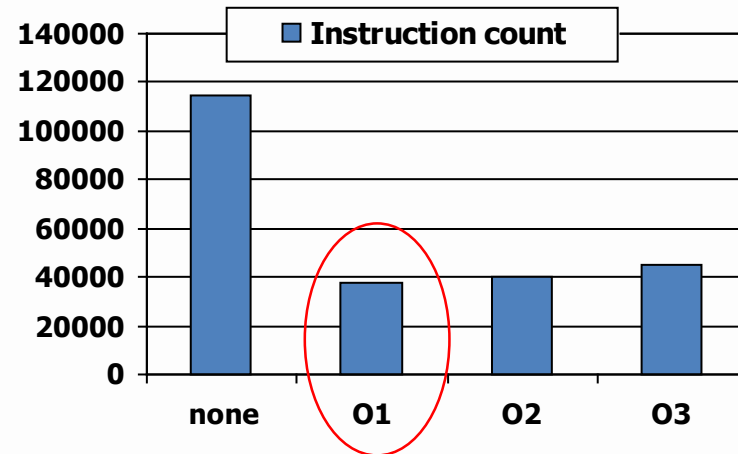
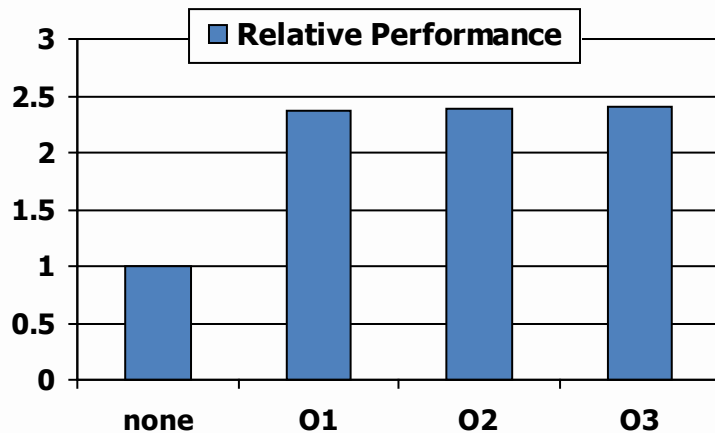


```
move $t0, $a0          # p = & array[0]  
sll $t1, $a1, 2        # $t1 = size * 4  
add $t2, $a0, $t1      # $t2 = &array[size]  
                        # (array end)  
loop2: sw $zero, 0($t0) # Memory[p] = 0  
addi $t0, $t0, 4       # p = p + 4  
slt $t3, $t0, $t2      # $t3 = (p < &array[size])  
bne $t3, $zero, loop2  # if (p < &array[size])  
                        # goto loop2
```

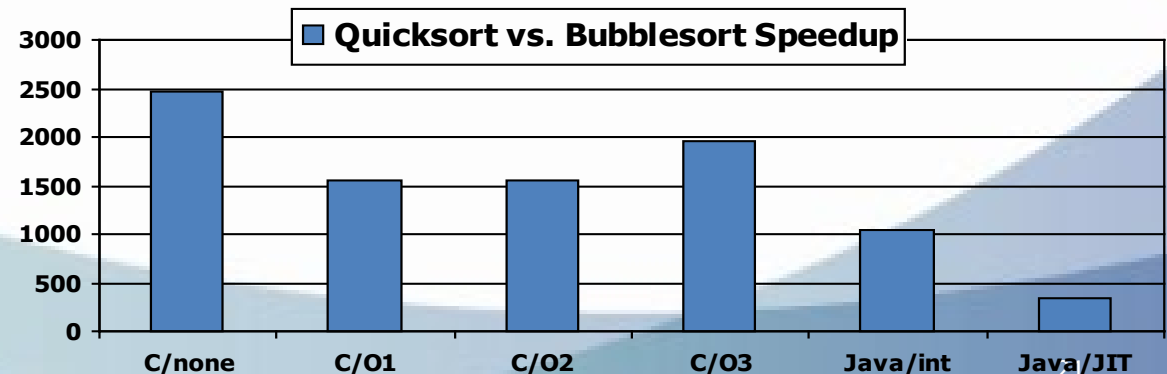
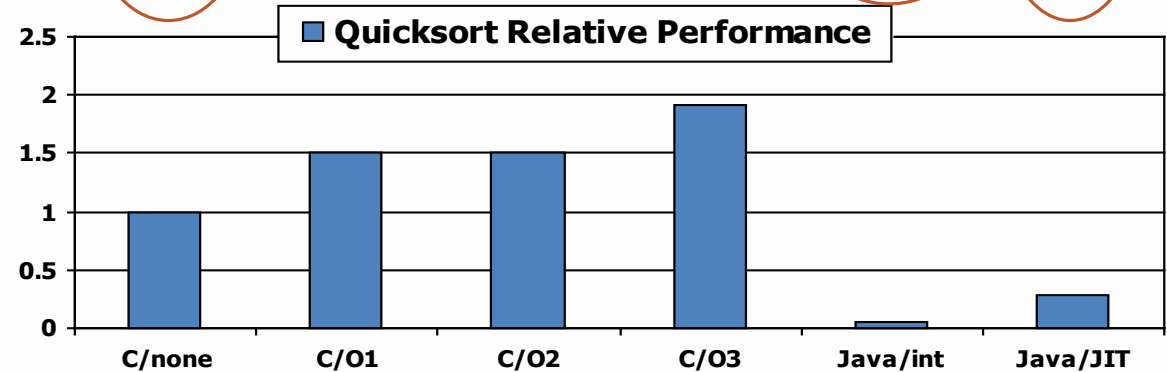
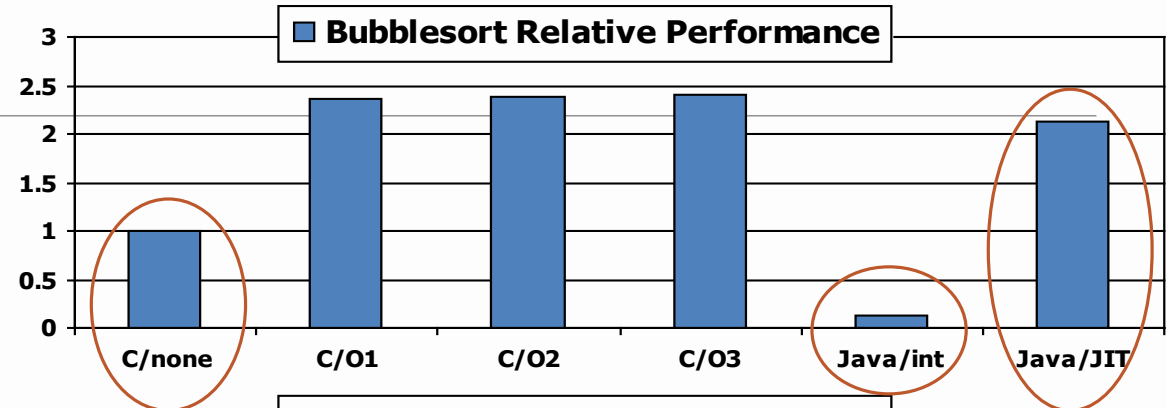
Only four instructions in the loop
Good Compiler should do this for you

Effect of Compiler Optimization

Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20



Impact of Language and Algorithm



- C is 8.3 times faster than Java/Int, but 2.1 times slower than Java/JIT
 - Java/Int: JVM bytecode
 - Java/Just-In-Time (JIT) compiler: compile bytecode to machine code with optimization

Lessons Learnt

- Instruction count and CPI cannot represent performance well
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
 - Comparable to optimized C in some cases
- Efficient algorithms are more important than compiler optimization

ARM and x86 ISA

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

ARM & MIPS Similarities

- ARM register-register and data transfer instructions compared to those of MIPS

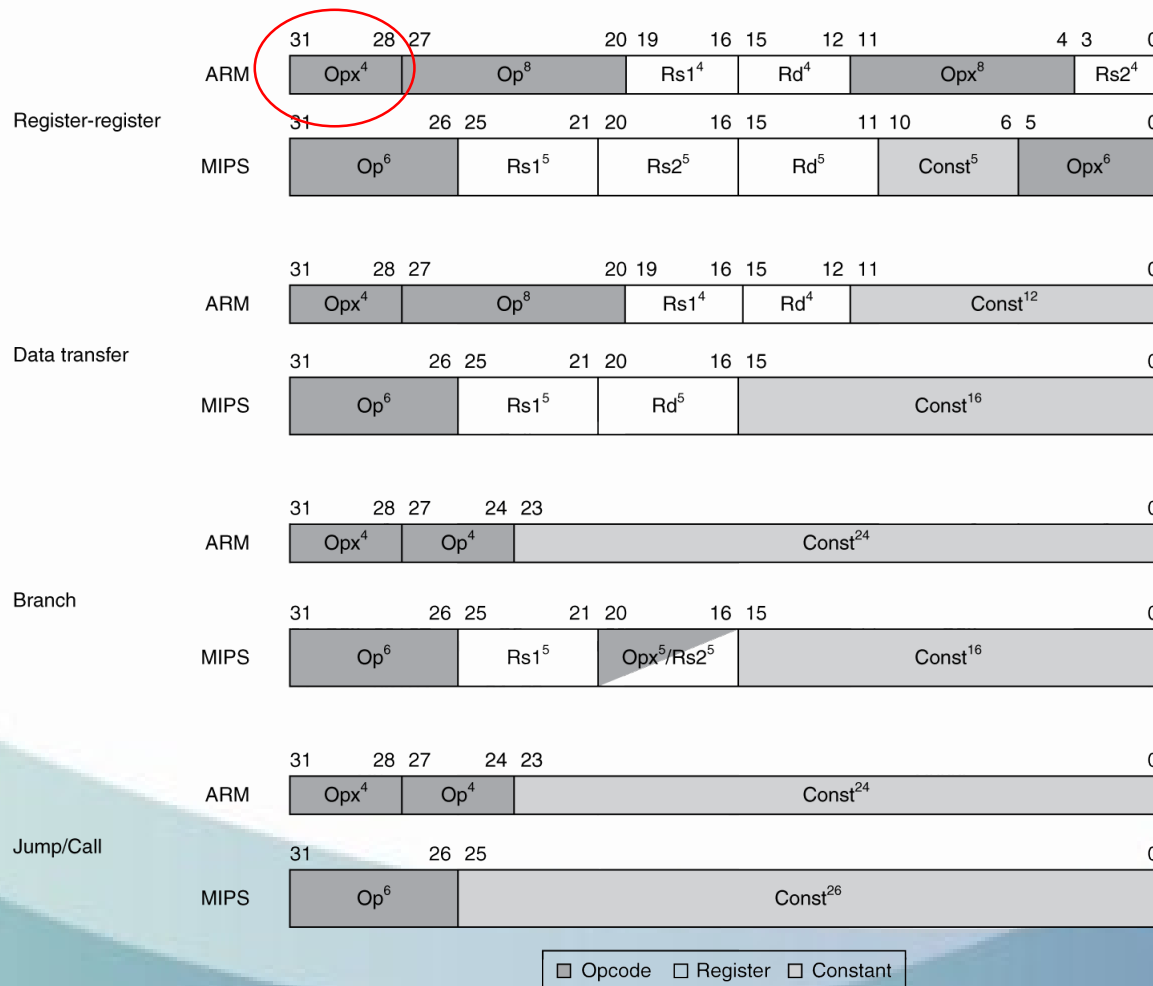
	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl ¹	sllv, sll
	Shift right logical	lsr ¹	srlv, srl
	Shift right arithmetic	asr ¹	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

FIGURE 2.32 from
Computer Org. and
Design, 5th edition

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
 - Negative, zero, carry, overflow
 - Compare instructions to set condition codes without keeping the result
- Each instruction can be executed conditionally
 - Top 4 bits of instruction word: condition value (with nop)
 - Can avoid branches over single instructions
 - Requires less code space and time to simply conditionally execute one instruction.

Instruction Encoding



The Intel x86 ISA

- Evolution with backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080
 - **Complex instruction set (CISC)**
 - 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments

The Intel x86 ISA

■ Further evolution...

- i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - The infamous FDIV bug (hardware bug affecting the floating-point unit)
- Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions

The Intel x86 ISA

- And further...
 - **AMD64 (2003): extended architecture to 64 bits**
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - **AMD64 (announced 2007): SSE5 instructions**
 - Intel declined to follow, instead...
 - Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance ≠ market success

Basic x86 Registers

Intel 80386

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
<div> <div>general-purpose registers</div> <div>MIPS programs can use four times as many GPRs than x86.</div> </div>			
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

Basic x86 Addressing Modes

- Two operands per instruction

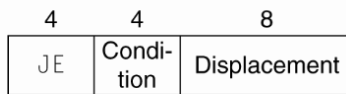
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes

- Address in register
- $\text{Address} = R_{\text{base}} + \text{displacement}$
- $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
- $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

x86 Instruction Encoding

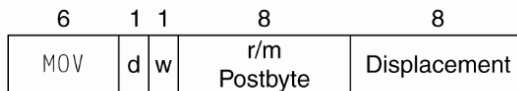
a. JE EIP + displacement



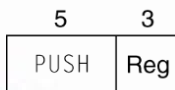
b. CALL



c. MOV EBX, [EDI + 45]



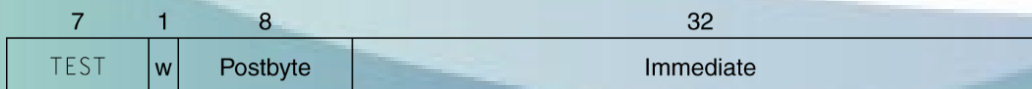
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



■ Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
 - Operand length, repetition, locking, ...

Implementing IA-32

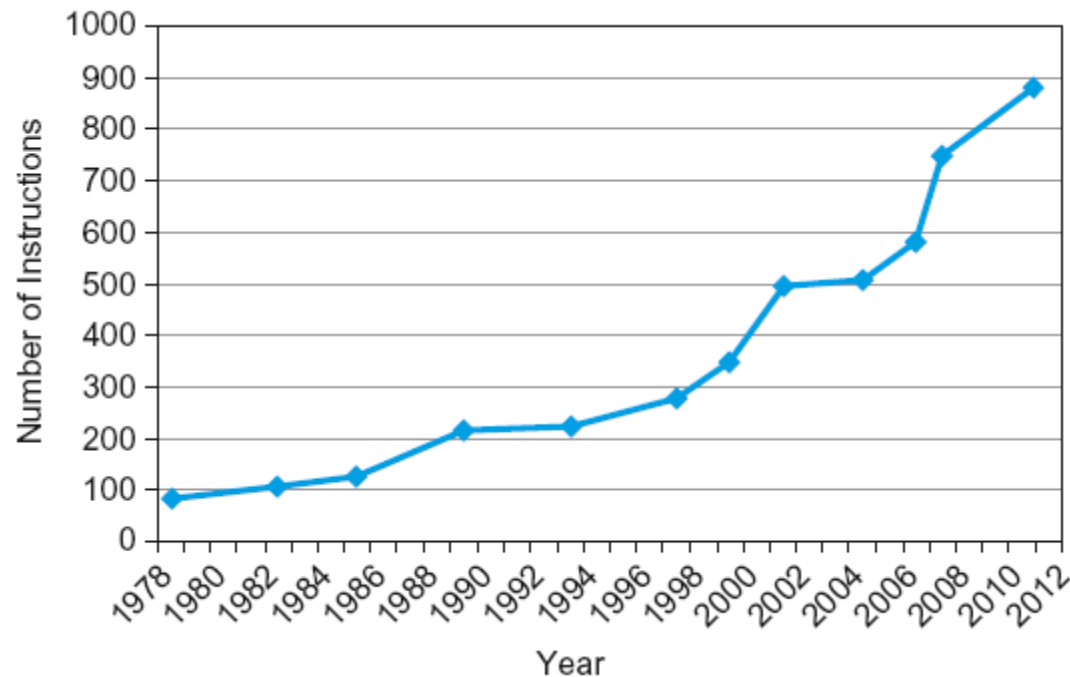
- Complex instruction set makes implementation difficult
 - Hardware translates instructions to simpler microoperations
 - Simple instructions: 1–1
 - Complex instructions: 1–many
 - Microengine similar to RISC
 - Market share makes this economically viable
- Comparable performance to RISC
 - Compilers avoid complex instructions

Fallacies

- Powerful instruction \Rightarrow higher performance
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change (X)
 - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped

Concluding Remarks

■ Design principles

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises

■ Layers of software/hardware

- Compiler, assembler, hardware

■ MIPS: typical of RISC ISAs

- c.f. x86

Profiling

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statements	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	If statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, returns, and case/switch statements	2%	0%

Arithmetic for Computers: Basics

Arithmetic for Computers

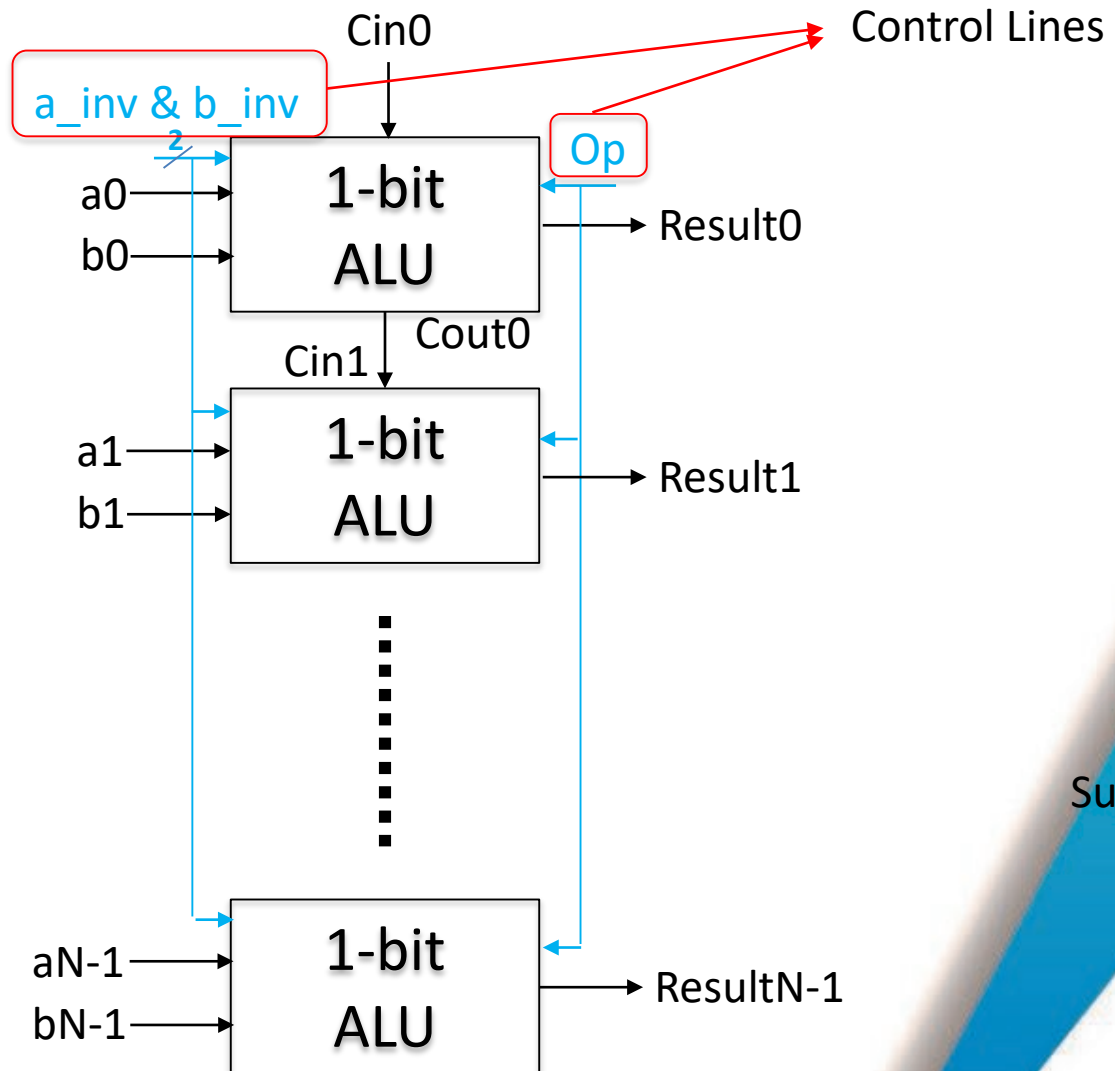
- Handled by ALU
- Operations on integers
 - Addition and subtraction
 - Dealing with overflow
 - Multiplication and division
- Floating-point real numbers
 - Representation and operations

Constructing ALU

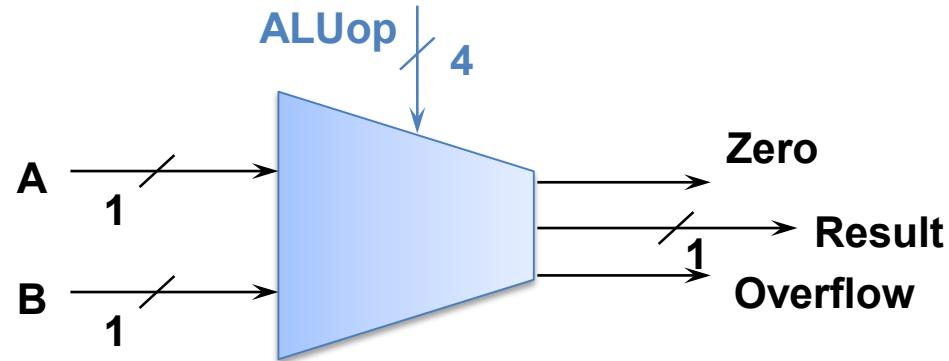
- Arithmetic Logic Unit (ALU)
 - It preforms addition, subtraction, logical operations (AND, OR, NOR), set on less than
 - $\text{NOR}(a, b) : (a + b)' = a' \cdot b'$
- MIPS works with 32 bits. Thus, a 32-bit-wide ALU is needed
- Starting with designing a 1-bit ALU and connect 32 of them to construct a 32-bit ALU

N-bit ALU

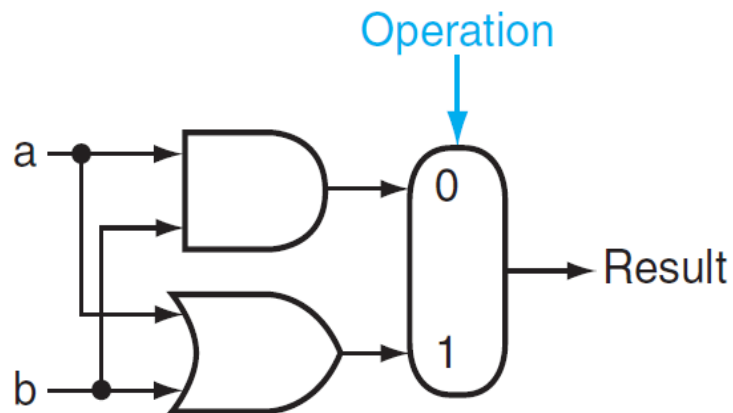
divide and conquer



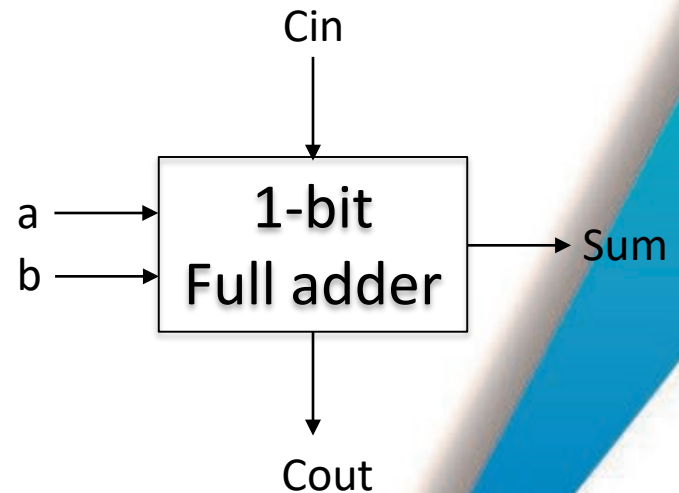
A 1-Bit ALU



It can be broken down into parts:



Logical operations



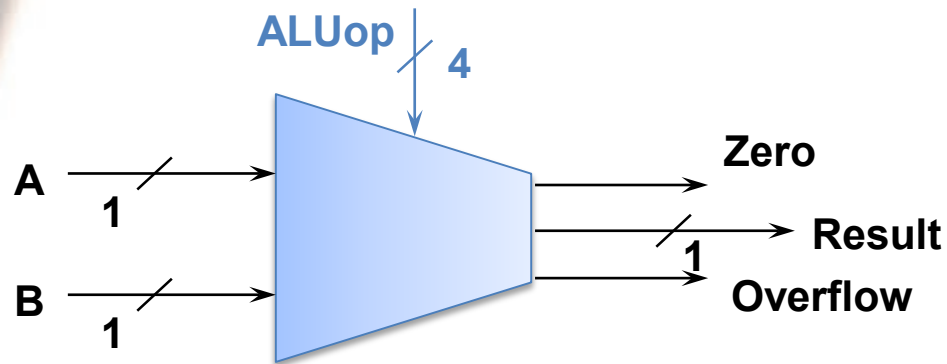
1-bit Full Adder

Inputs			Outputs	
a	b	Cin	Cout	Sum
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

$$\text{Cout} = (b \cdot \text{Cin}) + (a \cdot \text{Cin}) + (a \cdot b)$$

$$\text{Sum} = (a \cdot b' \cdot \text{Cin}') + (a' \cdot b \cdot \text{Cin}') + (a' \cdot b' \cdot \text{Cin}) + (a \cdot b \cdot \text{Cin})$$

ALU Function Control



ALUop

ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

1-bit ALU

ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

NOR : $(a + b)' = a' \cdot b'$

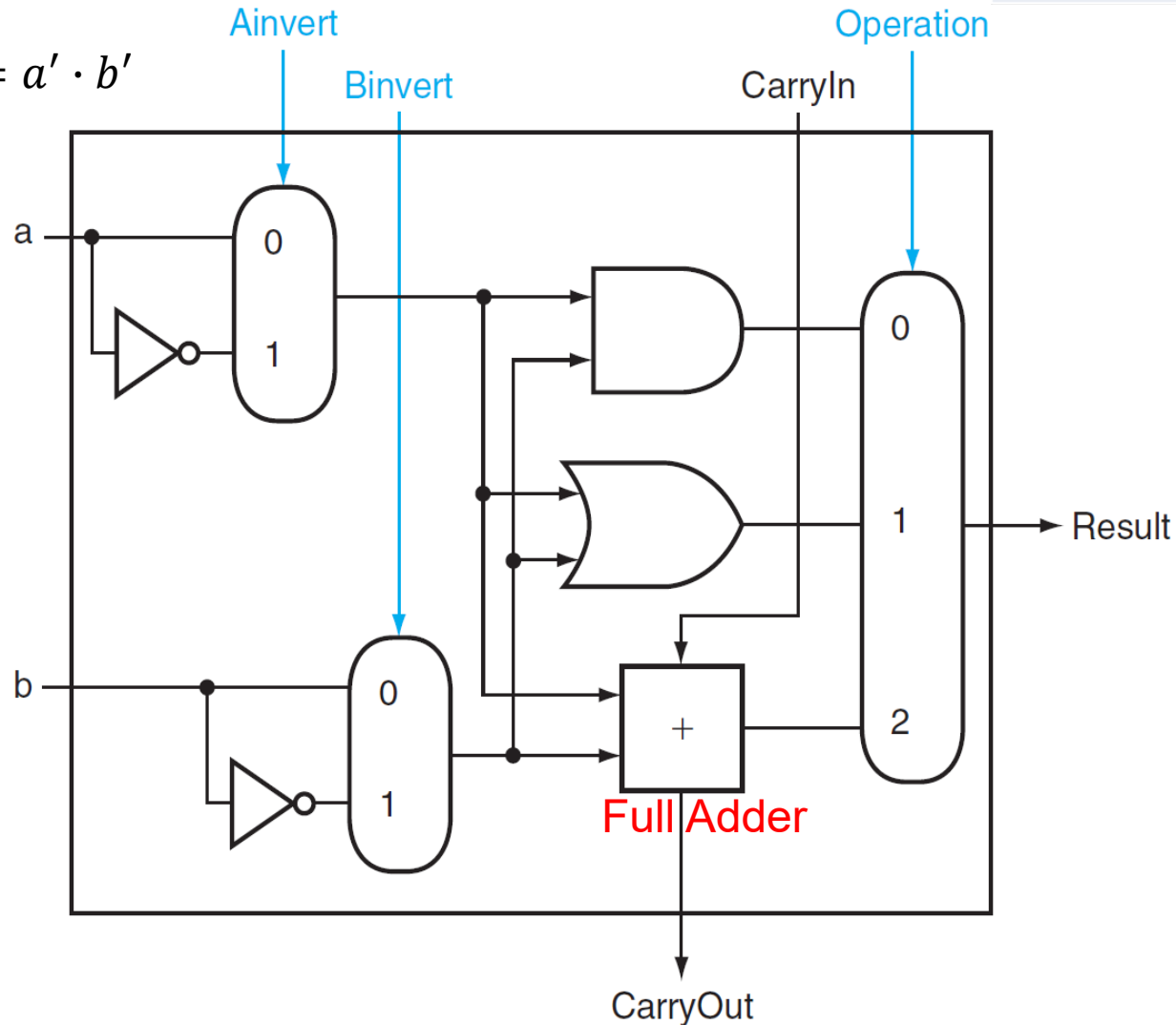
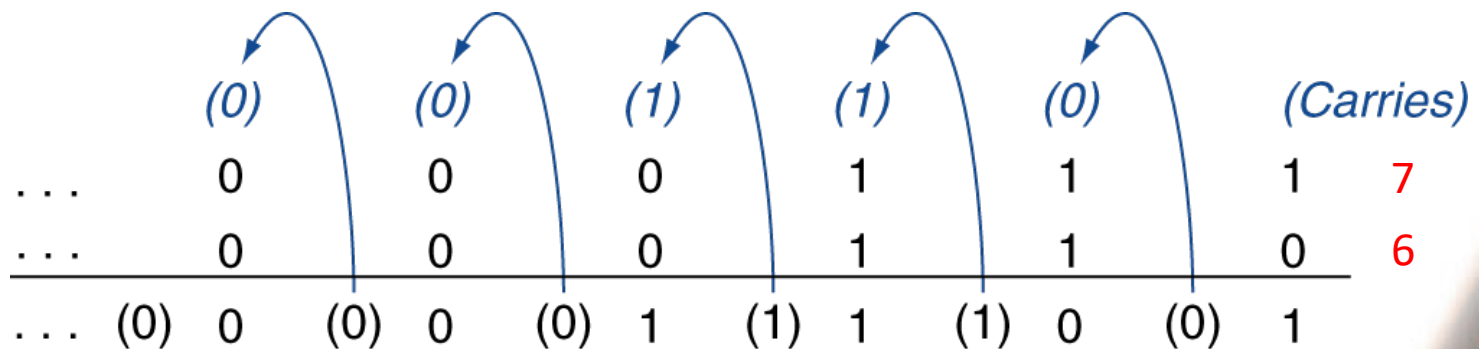


FIGURE B.5.9 from
Computer Org. and
Design, 5th edition

Addition

- Integer Addition

Example: $7 + 6$



Integer Subtraction

- Add negation of second operand
- Example: $7 - 6 = 7 + (-6)$
- Using 2's complement
 - $a - b = a + (-b) = a + (b' + 1)$

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ - \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\ \hline = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}} \end{array}$$

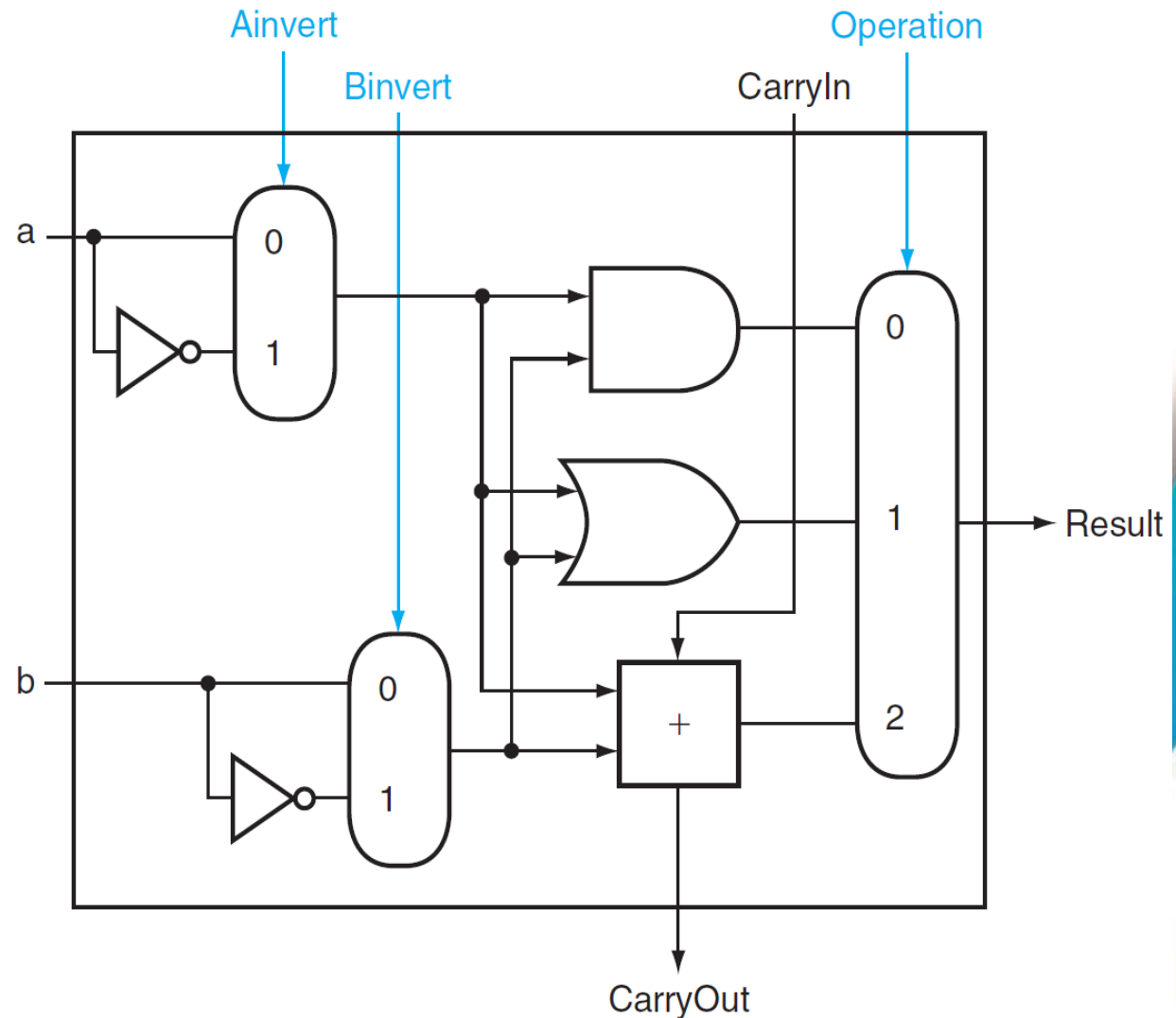
$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ + \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\ \hline = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}} \end{array}$$

Subtraction

$$\begin{aligned} a - b \\ &= a + (-b) \\ &= a + (b' + 1) \end{aligned}$$

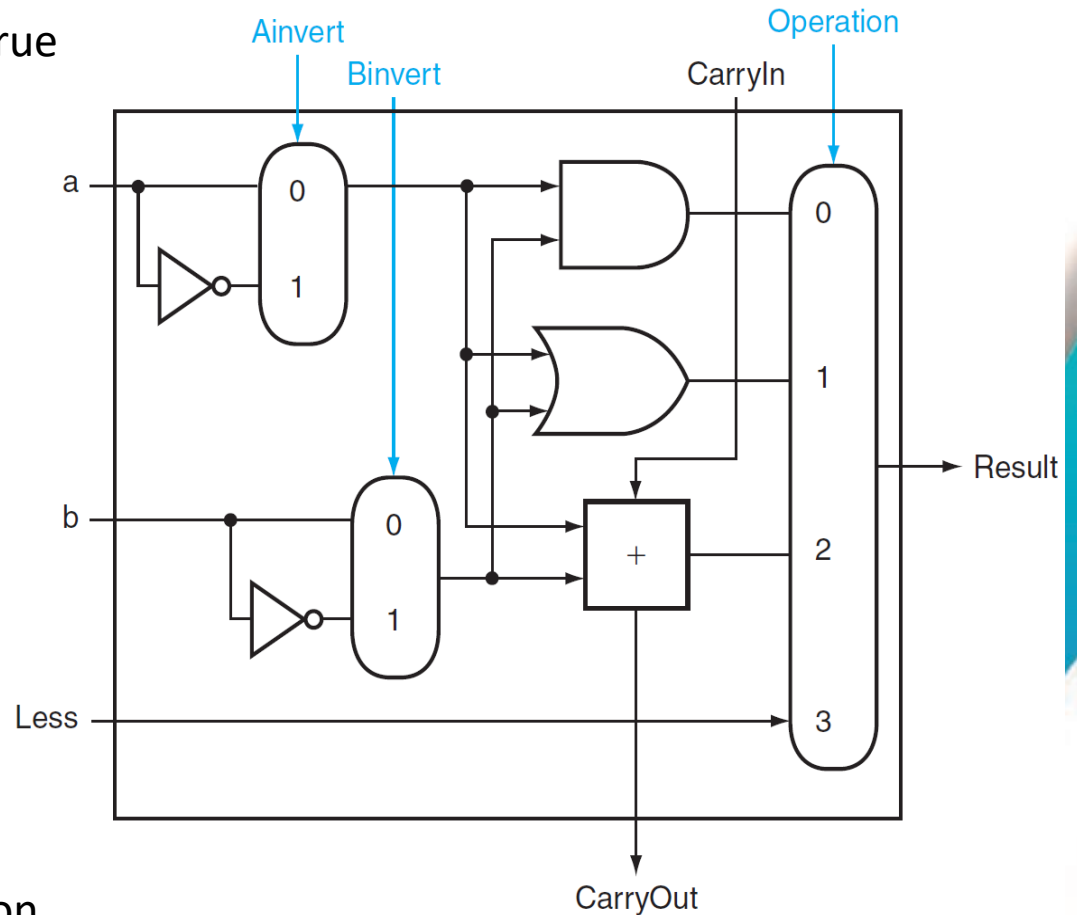
For the 1-bit ALU of LSB, set $\text{CarryIn} = \text{Binvert} = 1$

FIGURE B.5.9 from
Computer Org. and
Design, 5th edition



Set on Less Than

- Set result to 1 if the condition is true
 - Otherwise, set to 0
- `slt rd, rs, rt`
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;



Less: Set 0 for bits 1-31

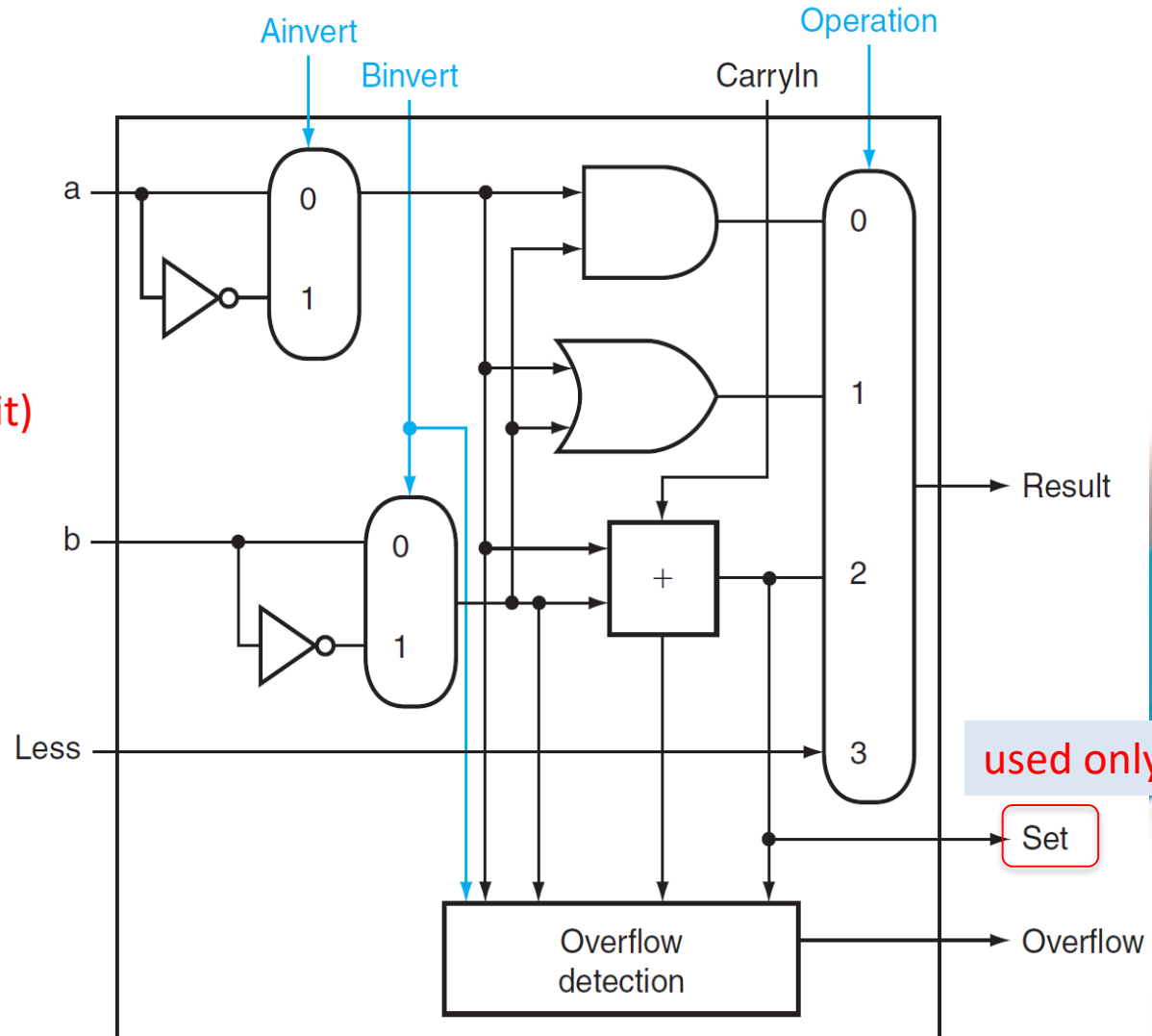
FIGURE B.5.10 from
Computer Org. and Design, 5th edition

Set on Less Than

ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

1-bit ALU for the most significant bit (31st bit)

If $rs - rt < 0$, Set = 1 (sign bit)



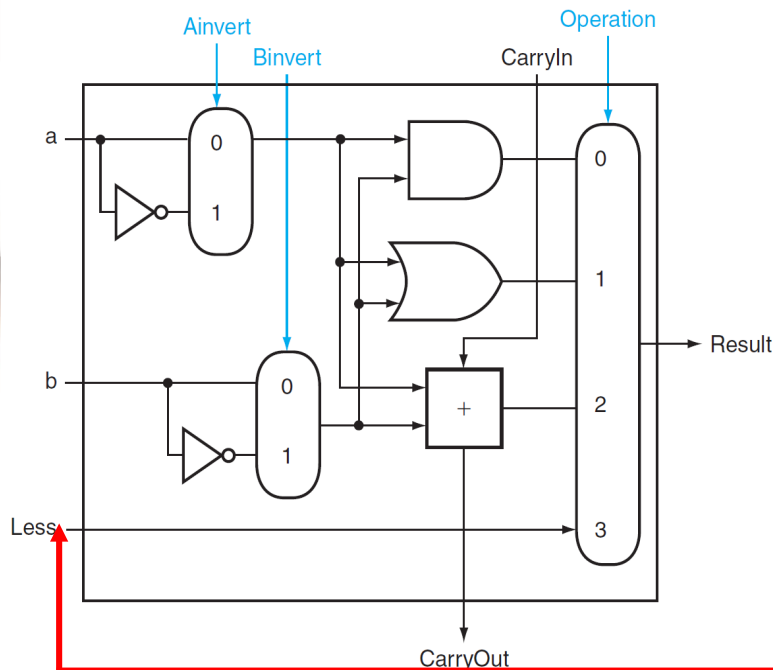
used only for slt

Set

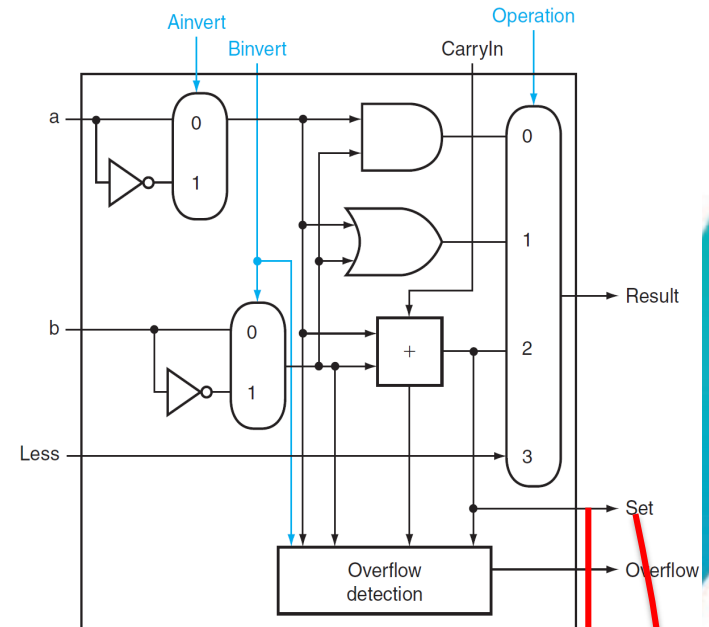
Overflow

Set on Less Than (32-bit)

bit 0



bit 31



Less = $\begin{cases} 0 & \text{for bits 1-31} \\ \text{Set} & \text{for bit 0} \end{cases}$

Arithmetic Overflow

- The condition occurs when a calculation of arithmetic operation(s) results in a result causing a given register to wrongly represents it

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0