Chapter 2: Instructions: Language of the Computer

2.13 - 2.14: C sort example and array vs pointer

2.16 – 2.17: X86 and more info for RISC-V

2.20: Conclusion

ITSC 3181 Introduction to Computer Architecture

https://passlab.github.io/ITSC3181

Department of Computer Science

Yonghong Yan

yyan7@uncc.edu

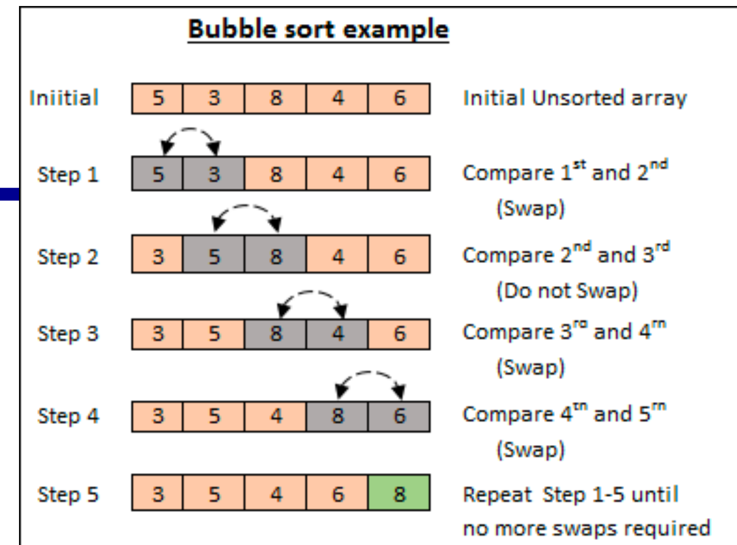
https://passlab.github.io/yanyh/

Chapter 2: Instructions: Language of the Computer

- Lecture
  - 2.1 Introduction
  - 2.2 Operations of the Computer Hardware
  - 2.3 Operands of the Computer Hardware
- Lecture
  - 2.4 Signed and Unsigned Numbers
  - 2.5 Representing Instructions in the Computer
- Lecture
  - 2.6 Logical Operations
  - 2.7 Instructions for Making Decisions
- Lecture
  - 2.8 Supporting Procedures in Computer Hardware
  - 2.9 Communicating with People
  - 2.10 RISC-V Addressing for Wide Immediate and Addresses

- Lecture
  - ~~2.11 Parallelism and Instructions: Synchronization~~
  - 2.12 Translating and Starting a Program
    - We covered before along with C Basics
  - 2.13 A C Sort Example to Put It All Together
  - 2.14 Arrays versus Pointers
    - We covered most before along with C Basics
  - ~~2.15 Advanced Material: Compiling C and Interpreting Java~~
  - ~~2.16 Real Stuff: MIPS Instructions~~
  - 2.17 Real Stuff: x86 Instructions
  - 2.18 Real Stuff: The rest of RISC-V
  - ~~2.19 Fallacies and Pitfalls~~
  - 2.20 Concluding Remarks
  - ~~2.21 Historical Perspective and Further Reading~~

The Sort Procedure in C
(Textbook Page 133)

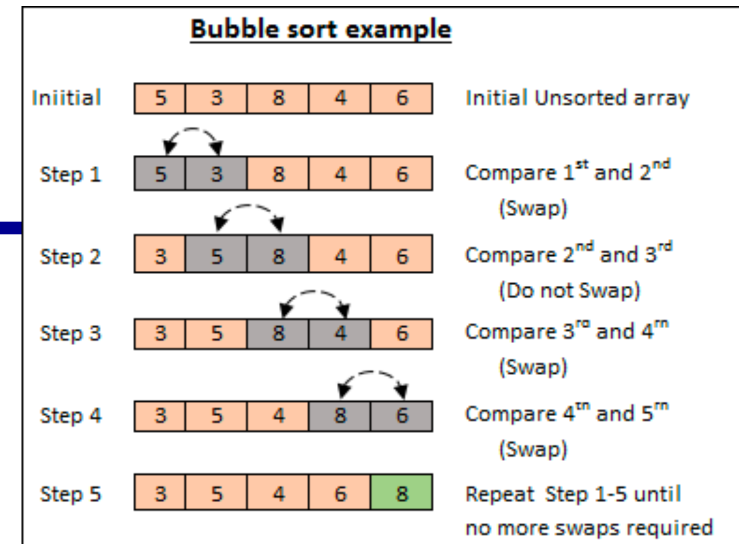


- Illustrates use of assembly instructions for a C bubble sort function
- Non-leaf (calls swap)

```
void sort (long long int v[], size_t n) {
  size_t 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 x10, n in x11, i in x19, j in x20

C Sort Example



- Swap procedure (leaf)

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

  – v in x10, k in x11, temp in x5

```
void sort (long long int v[], size_t n)
{
  size_t 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);
    }
  }
}
```

The Procedure Swap

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

- v in x10, k in x11, temp in x5

```
swap:
   slli x6,x11,3      // reg x6 = k * 8
   add  x6,x10,x6     // reg x6 = v + (k * 8)
   ld   x5,0(x6)      // reg x5 (temp) = v[k]
   ld   x7,8(x6)      // reg x7 = v[k + 1]
   sd   x7,0(x6)      // v[k] = reg x7
   sd   x5,8(x6)      // v[k+1] = reg x5 (temp)
   jalr x0,0(x1)      // return to calling routine
```

The Outer Loop

```
void sort (long long int v[], size_t n)
{
  size_t 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 x10, n in x11, i in x19, j in x20

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   mv   x21, x10      // store parameter x10 into x21
   mv   x22, x11      // store parameter x11 into x22 (not using st
   li   x19,0         // i = 0
for1tst:
   bge  x19,x11,exit1 //go to exit1 if x19≥x11(i≥n)

   (body of outer for-loop)

   addi x19,x19,1         // i += 1
   j    for1tst       // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j – = 1) { swap (v, j); }

```
    addi x20,x19,-1    // j = i –1
for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,3      // reg x5 = j * 8
    add  x5,x10,x5     // reg x5 = v + (j * 8)
    ld   x6,0(x5)      // reg x6 = v[j]
    ld   x7,8(x5)      // reg x7 = v[j + 1]
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
    mv   x10, x21      // first swap parameter is v
    mv   x11, x20      // second swap parameter is j
    jal  x1,swap   // call swap
    addi x20,x20,-1    // j –= 1
    j    for2tst   // branch to test of inner loop
exit2:
```

Preserving Registers

- Preserve saved registers:

```
  addi sp,sp,-40  // make room on stack for 5 regs
  sd   x1,32(sp)  // save x1 on stack
  sd   x22,24(sp) // save x22 on stack
  sd   x21,16(sp) // save x21 on stack
  sd   x20,8(sp)  // save x20 on stack
  sd   x19,0(sp)  // save x19 on stack
```

- Restore saved registers:

```
exit1:
  ld   x19,0(sp)  // restore x19 from stack
  ld   x20,8(sp)  // restore x20 from stack
  ld   x21,16(sp) // restore x21 from stack
  ld   x22,24(sp) // restore x22 from stack
  ld   x1,32(sp)  // restore x1 from stack
  addi sp,sp, 40  // restore stack pointer
  jalr x0,0(x1)
```

The Full Version

- Check the textbook

Saving registers		
	sort: addi sp, sp, -40	# make room on stack for 5 registers
	sd x1, 32(sp)	# save return address on stack
	sd x22, 24(sp)	# save x22 on stack
	sd x21, 16(sp)	# save x21 on stack
	sd x20, 8(sp)	# save x20 on stack
	sd x19, 0(sp)	# save x19 on stack
Procedure body		
Move parameters	mv x21, x10	# copy parameter x10 into x21
	mv x22, x11	# copy parameter x11 into x22
Outer loop	li x19, 0	# i = 0
	for1tst:bge x19, x22, exit1	# go to exit1 if i >= n
Inner loop	addi x20, x19, -1	# j = i - 1
	for2tst:blt x20, x0, exit2	# go to exit2 if j < 0
	slli x5, x20, 3	# x5 = j * 8
	add x5, x21, x5	# x5 = v + (j * 8)
	ld x6, 0(x5)	# x6 = v[j]
	ld x7, 8(x5)	# x7 = v[j + 1]
	ble x6, x7, exit2	# go to exit2 if x6 < x7
Pass parameters and call	mv x10, x21	# first swap parameter is v
	mv x11, x20	# second swap parameter is j
	jal x1, swap	# call swap
Inner loop	addi x20, x20, -1	j for2tst
	j for2tst	# go to for2tst
Outer loop	exit2: addi x19, x19, 1	# i += 1
	j for1tst	# go to for1tst
Restoring registers		
	exit1: ld x19, 0(sp)	# restore x19 from stack
	ld x20, 8(sp)	# restore x20 from stack
	ld x21, 16(sp)	# restore x21 from stack
	ld x22, 24(sp)	# restore x22 from stack
	ld x1, 32(sp)	# restore return address from stack
	addi sp, sp, 40	# restore stack pointer
Procedure return		
	jalr x0, 0(x1)	# return to calling routine

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

``
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
``
	``
	clear2(int *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	       p = p + 1)
	    *p = 0;
	}
	``

```
   li   x5,0        // i = 0
loop1:
   slli x6,x5,2     // x6 = i * 4
   add  x7,x10,x6   // x7 = address
                    // of array[i]
   sd   x0,0(x7)    // array[i] = 0
   addi x5,x5,1     // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

```
   mv x5,x10        // p = address
                    // of array[0]
   slli x6,x11,3    // x6 = size * 4
   add x7,x10,x6    // x7 = address
                    // of array[size]
loop2:
   sd x0,0(x5)      // Memory[p] = 0
   addi x5,x5,8     // p = p + 4
   bltu x5,x7,loop2
                    // if (p<&array[size])
                    // go to loop2
```

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

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
  - 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

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
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

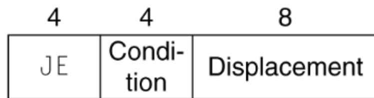
- Two operands per instruction

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

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

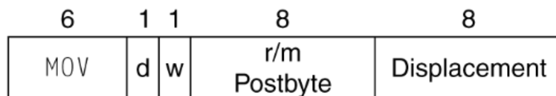
x86 Instruction Encoding

a. JE EIP + displacement

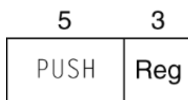


b. CALL

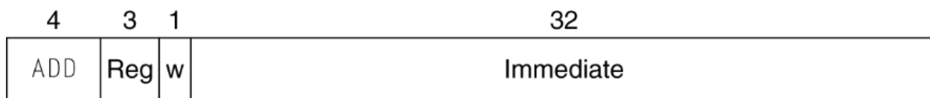
c. MOV EBX, [EDI + 45]



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

More Materials for RISC-V Instruction

- Slides for RISC-V intro and specification:
  - https://passlab.github.io/ITSC3181/notes/lectureXX_RISCV_ISA.pdf
- RISC-V instruction reference cards:
  - https://passlab.github.io/ITSC3181/resources/RISCVGreenCardv8-20151013.pdf
- Information for learning assembly programming
  - https://passlab.github.io/ITSC3181/resources/RISC-VAssemblyProgramming.html
- Resources from the official website including the standard
  - https://riscv.org/

Concluding Remarks

- Instruction Set Architecture are Hardware and Software Interface
- Three major classes of instructions
  - Arithmetic and logic instructions
  - Load/Store instructions
  - Control transfer (branch and jump/link)
  - Other helpful instruction, e.g. load immediate, etc.
- High-level language constructs to instruction sequence
  - Arithmetic and logic expression => Arithmetic and logic instructions
  - Array reference => address calculation and load/store
  - If-else/switch-case, for/while-loop => branch and jump
  - Function call => jump/link, store and restore registers
- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
  4. Make the common case fast

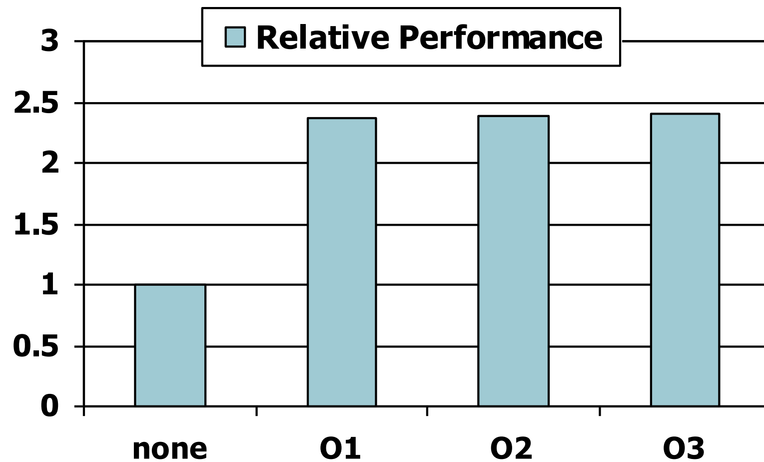
Contents Not Covered.

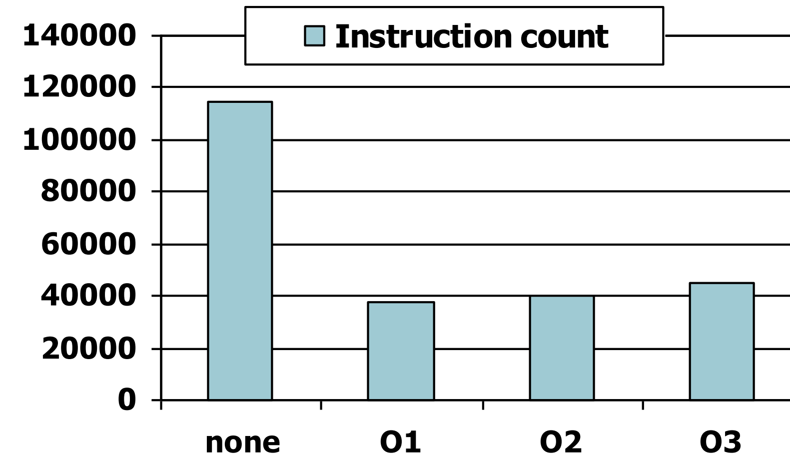
- struct person { int age; int height;} sam;

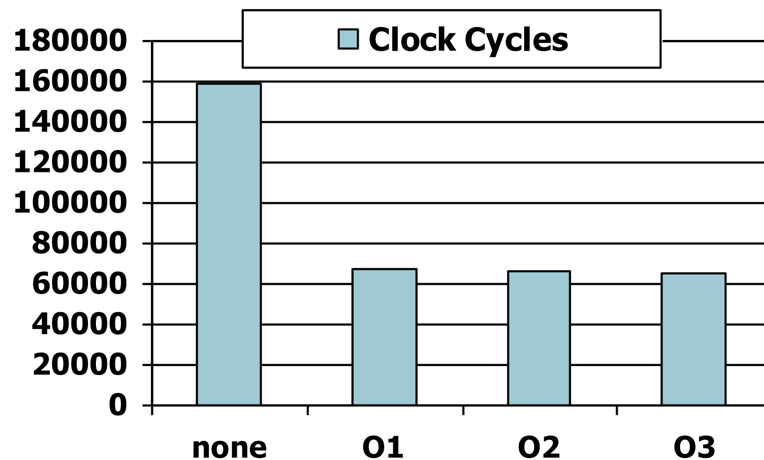
- Class:

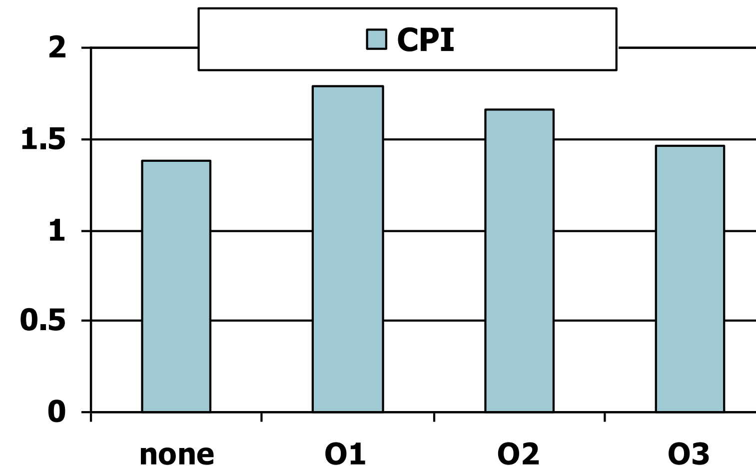
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

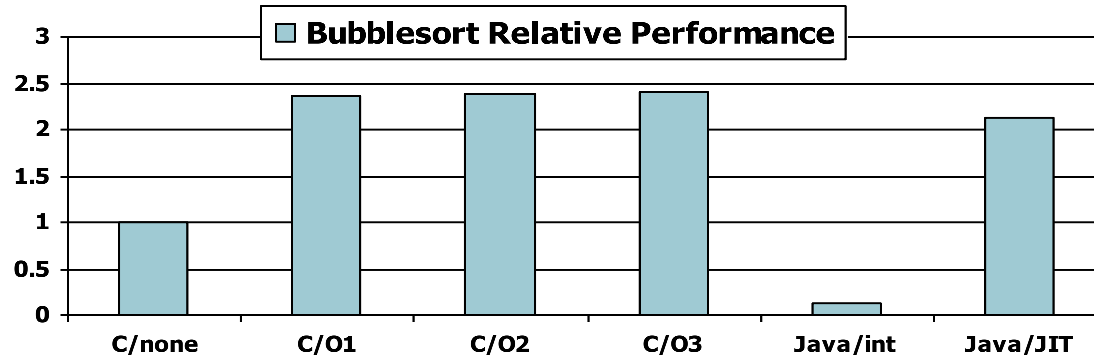


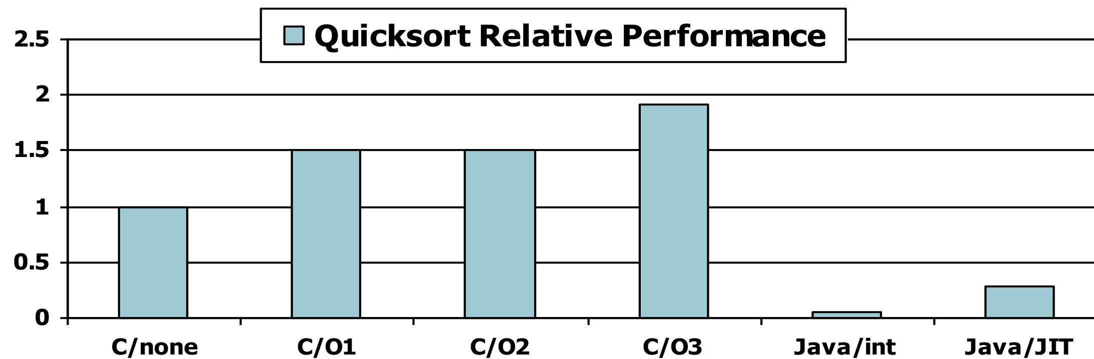


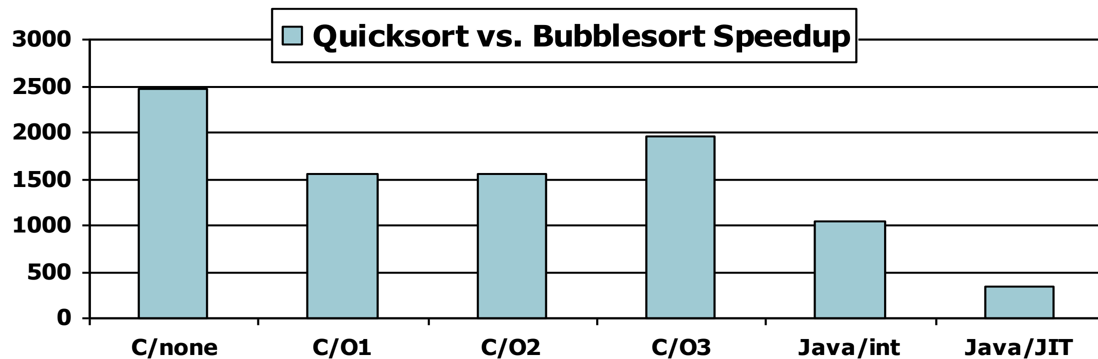




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

MIPS Instructions

- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- Different conditional branches
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

Instruction Encoding

Register-register

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 11	10 6	5 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31 20	19 15	14 12	11 7	6 0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

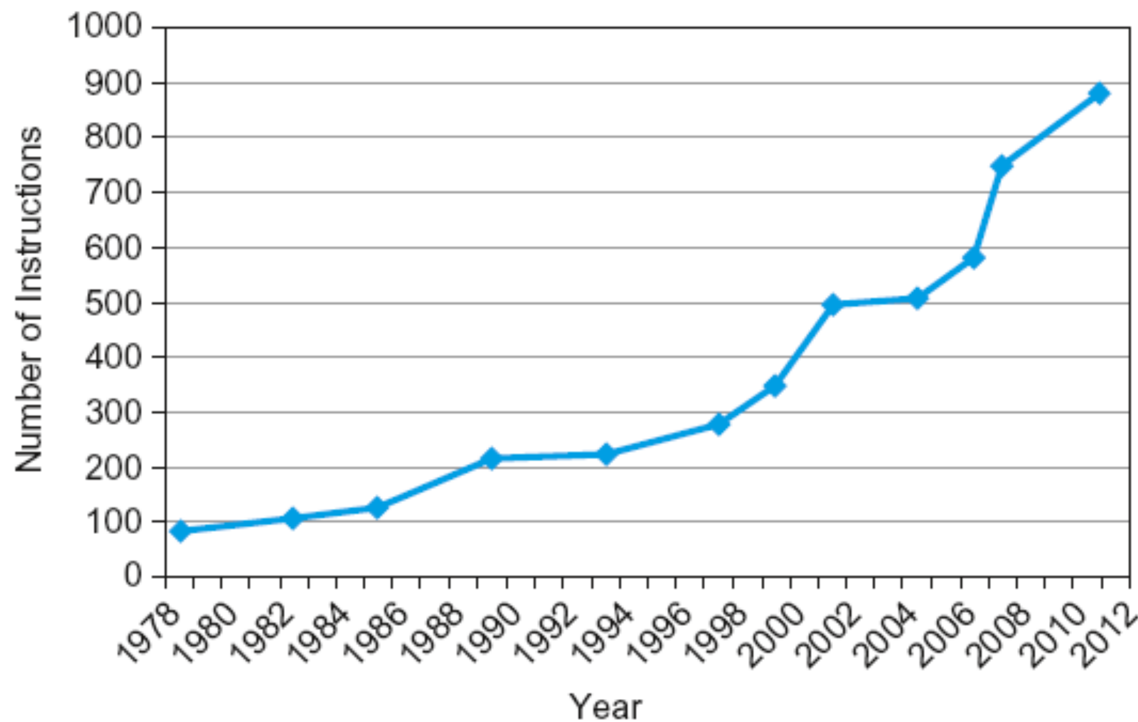
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

Fallacies

- Powerful instruction ⇒ 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 ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - 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