Misc Topics

Z. Jerry Shi
Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

- RISC vs CISC

- LA
- Arrays vs pointers
- Set less than
- Frame pointer (Section 2.8)

RISC-V ISA

- RISC-V is a typical of RISC ISAs
  - ARM and MIPS are also in this category
- x86 in Intel's processors is CISC
- Design principles in RISC-V
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

RISC: Reduced Instruction Set Computer

CISC: Complex Instruction Set Computer

CISC example: Intel's x86

- Only 8 registers at beginning
  - ax, bx, cx, dx, si, di, bp, and sp
- Variable instruction length
  - 1 byte to 15 bytes
- Operand can be in memory

```
dec   [ebx]          ; Mem[ebx]--
```

- Many memory addressing modes

```
mov eax, [edx + 4*ebx + 8]
; general form:  [reg + reg * size + offset]
```

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

Question

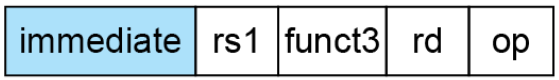
- What design decisions in RISC-V reflects the following principle?

Simplicity favors regularity

It is simpler to handle objects with regularity

RISC-V Addressing Mode

1. Immediate addressing



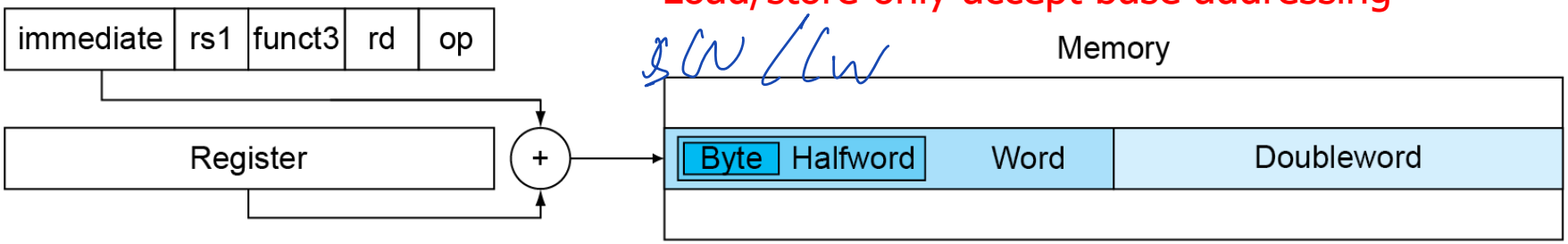
addi

2. Register addressing

JALR

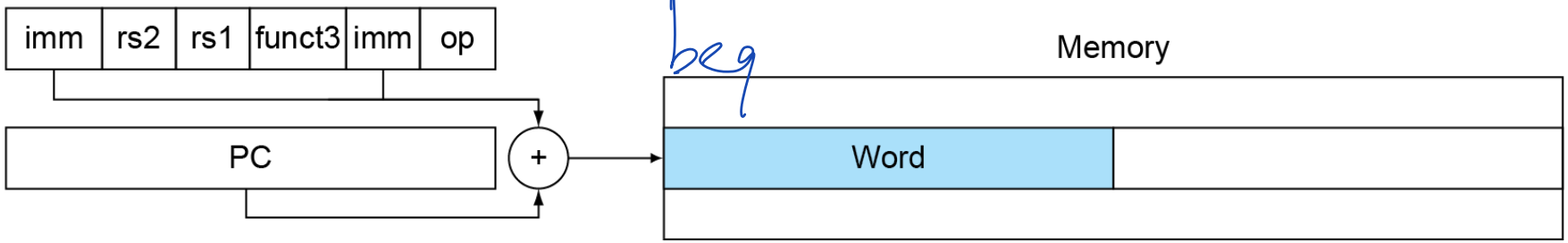
3. Base addressing

Load/store only accept base addressing



sw / lw

4. PC-relative addressing



beq

RISC-V instruction format

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Suppose we keep the size of opcode and funct3 the same but increase the number of registers to 64.

How many bits do we need for each register number?

How does this affect I-type instruction?

How does this affect other types of instructions?

Use of RISC-V instructions

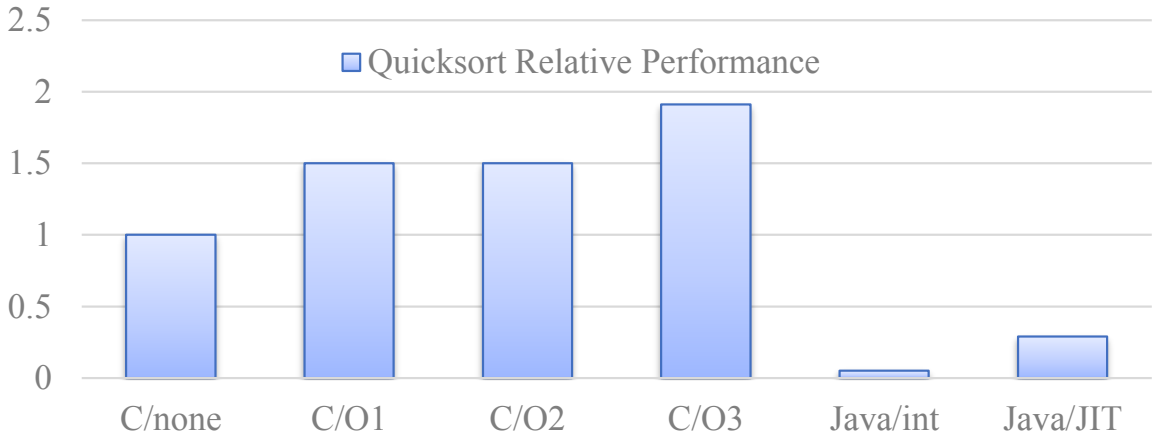
- Measure RISC-V instruction executed in SPEC CPU2006 CPU benchmarks

Instruction class	RISC-V examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, sb, lh, sh, lui`	35%	36%
Logical	`and, or, xor, sll, srl, sra`	12%	4%
Branch	`beq, bne, blt, bge, bltu, bgeu`	34%	8%
Jump	`Jal, jalr`	2%	0%

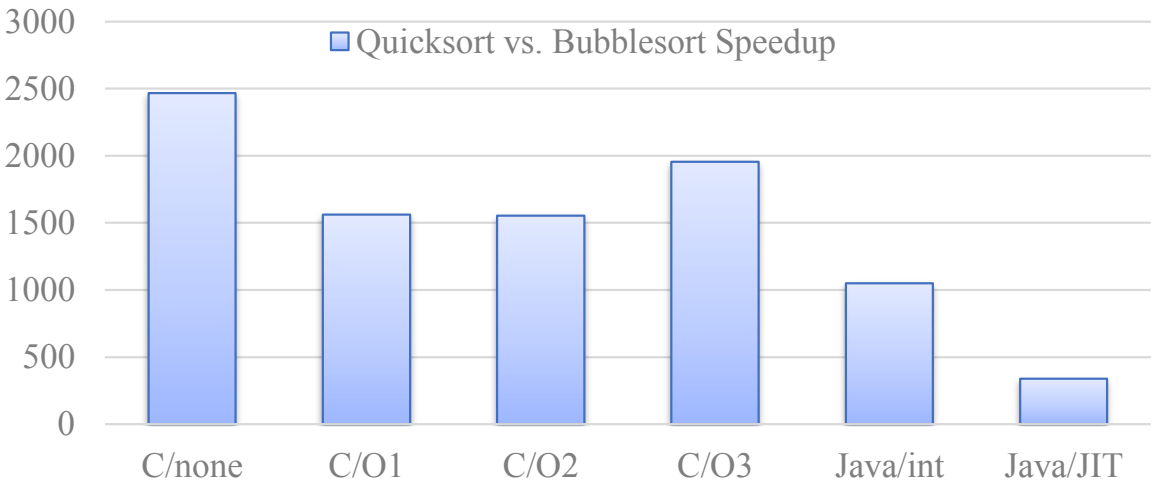
Figure 2.48

Effect of Language and Algorithm

Quicksort
in different languages



Quicksort vs Bubblesort



Nothing can fix a dumb algorithm!

Fallacies

- Use assembly code for high performance
  - Modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity
  - Hard to maintain

Question

- Which of the following ISAs is of a different type from other two?

RISC-V, ARM, x86

A. RISC-V
B. ARM
C. x86

Study the remaining slides yourself

How does LA work?

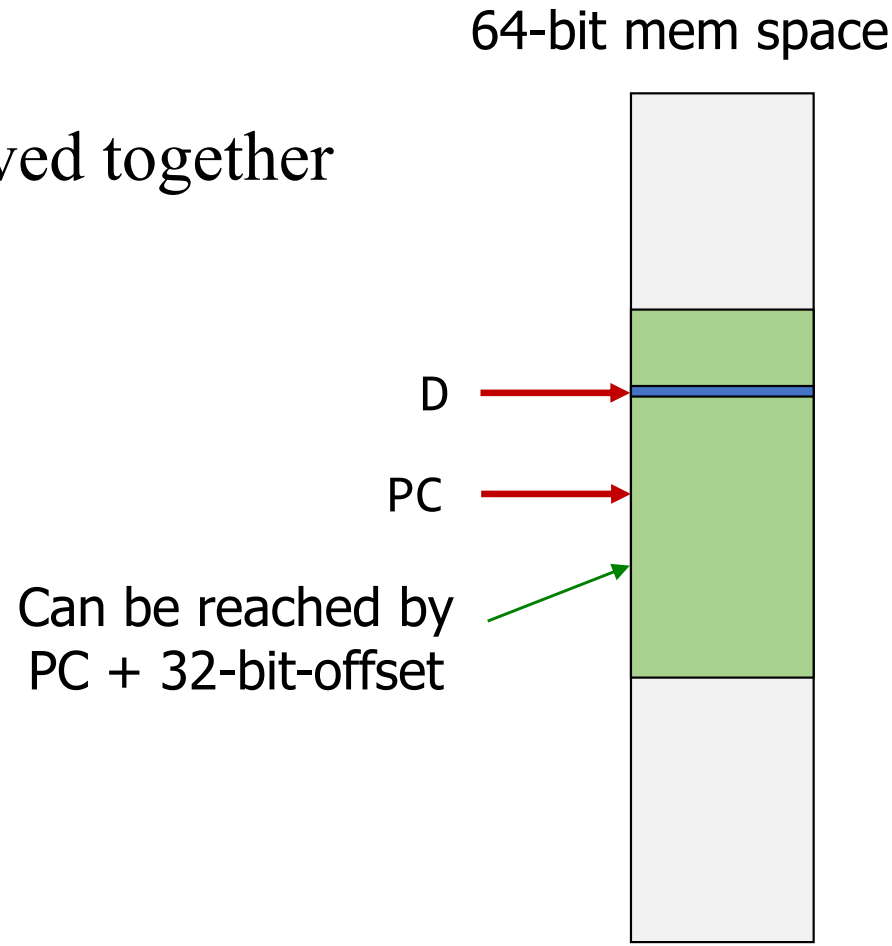
```
la   rd, var     # load var's addr into rd
```

- An address is a 32-bit value. We could use LUI and ADDI to get any 32-bit value into a register

- Not easy on 64-bit processors where addresses are of 64 bits
  - More instructions are needed
  - Often, we do not need to access memory far away from the instruction

PC-relative addressing

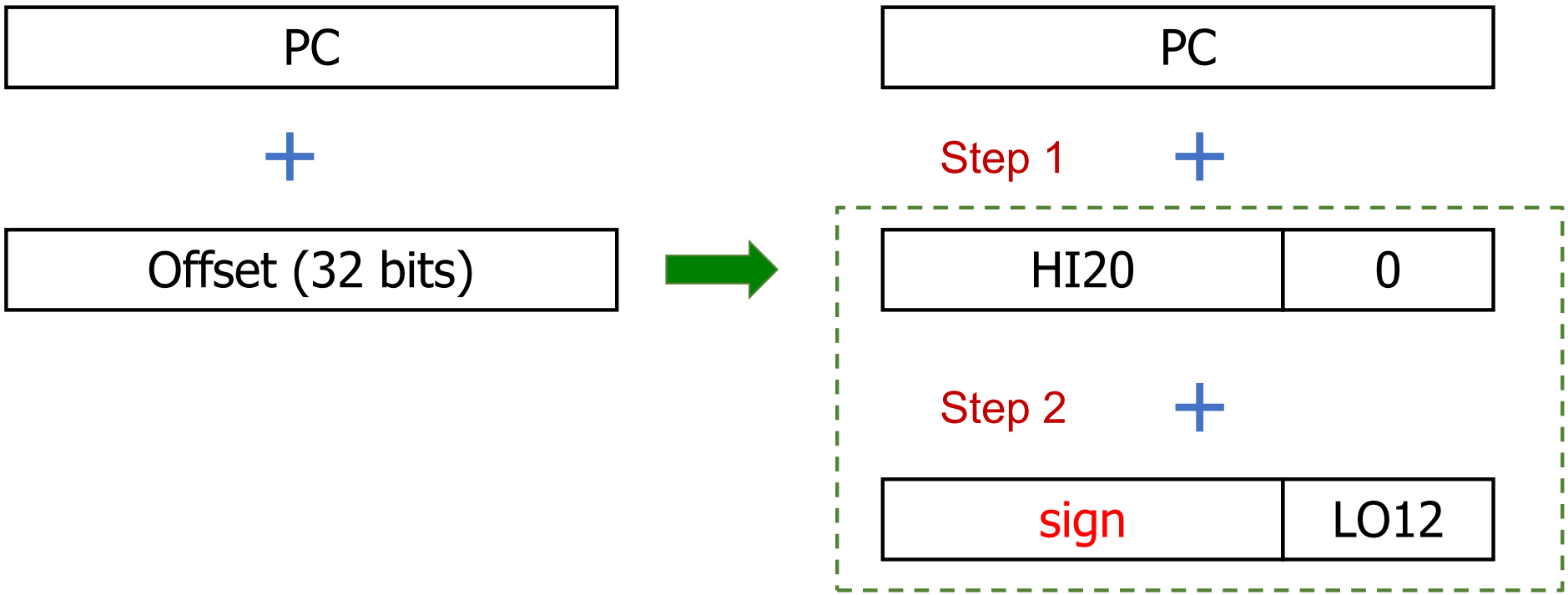
- Use PC-relative addressing to access data near the instruction
  - Add a 32-bit offset to PC
- Benefit: Data and code can be moved together



Can only access a tiny area of the memory space, but is good enough for common cases.

Adding 32 bits to PC

- We cannot specify 32 bits in one instruction, but can get 32 bits in a register with two instructions (LUI and ADDI)
- Do it in two steps: add higher 20 bits, then lower 12 bits



AUIPC

```
AUIPC rd, immd      # rd = PC + UI
```

- Operations (add upper immediate and PC):
  - Obtain an immediate like LUI:

    The higher 20 bits from machine code, the lower 12 bits are 0
  - Add the immediate and PC PC-relative addressing

```
# some assemblers convert LA to the following
auipc   x1, %pcrel_hi(var)
addi    x1, %pcrel_lo(var)      # Add the lower 12 bits

# RARS does not support %pcrel_hi
```

Encoding of AUIPC

- Which format would you use to encode AUIPC?

A. I

B. S

C. SB

D. U

E. UJ

```
AUIPC t1,0xABCDE
```

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Example

- What is in t0(x5) after auipc?
- What is in t0(x5) after the second ADDI instruction?
- Can you find the bits in each field in AUIPIC?

main:

Address	Code	Basic	
0x00400000	0x00000013	addi x0,x0,0x00000000	4: nop
0x00400004	0x00000297	auipc x5,0x00000000	5: la t0, main
0x00400008	0xffc28293	addi x5,x5,0xfffffffc	

Clearing an array – Pointer version

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

```
for (p = &a[0]; p < &a[8]; p += 1)
        *p = 0;
```

```
        p = &a[0]
        end = &a[8]
        # go to cond test
loop:
        Write 0 to p[0]
        Increment p, by ____
        If (p < end) goto loop
```

Address	Value
0x9024	
0x9020	a[8]
0x901C	a[7]
0x9018	a[6]
0x9014	a[5]
0x9010	a[4]
0x900C	a[3]
0x9008	a[2]
0x9004	a[1]
0x9000	a[0]

Arrays vs. Pointers Example: Clearing an Array

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

```
clear2(int *array, size_t 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]
   sw   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 = addr of array[0]
   slli x6,x11,2   // x6 = size * 4
   add x7,x10,x6   // x7 = address
                   // of array[size]
loop2:
   sw x0,0(x5)     // Memory[p] = 0
   addi x5,x5,4    // p = (int)p + 4
   bltu x5,x7,loop2
                   // if (p<&array[size])
                   // go to loop2
```

How many instructions are executed in each iteration, in clear1 and clear2?

Comparison of Array vs. Pointer

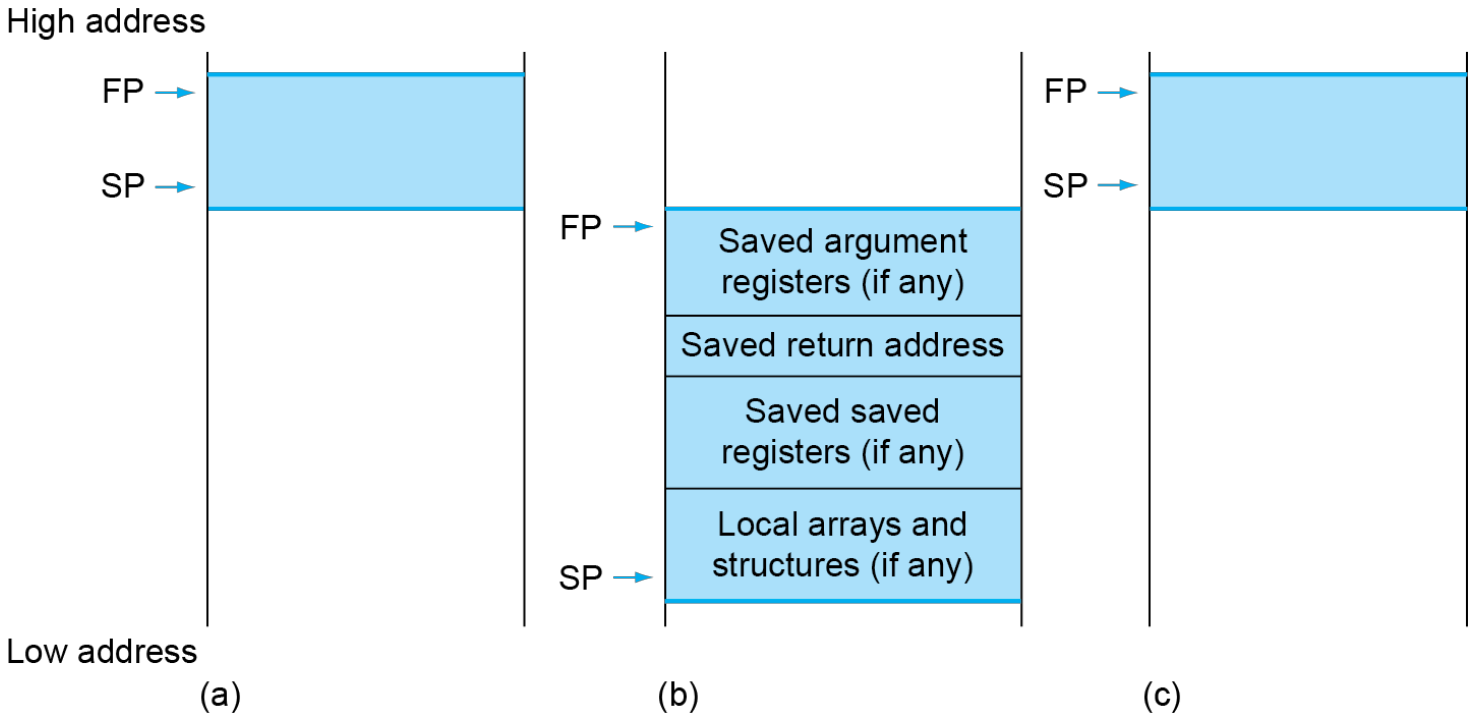
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - Multiply "strength reduced" to shift
  - Compared to incrementing pointer

- Compiler can achieve same effect as manual use of pointers (most of the time)
  - Better to make program clearer and safer

Array copy example:
cse3666/array_copy.md at master · zhijieshi/cse3666 (github.com)

Frame pointer

- Every function has a frame: procedure frame/activation record
- fp points to a fixed location in the procedure frame
  - sp may change within the function, but fp does not
  - Use fp as the base register for local variables



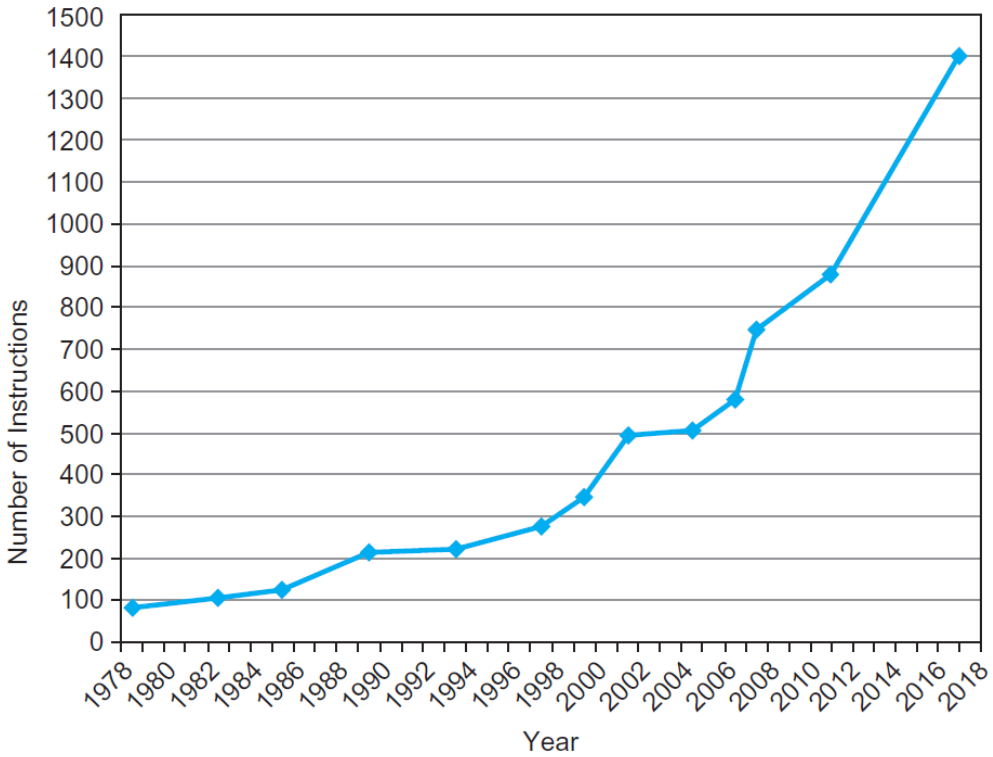
Example: set and restore frame pointer

```
addi    sp, sp, -4          # push fp
sw      fp, (sp)
addi    fp, sp, 0           # set fp

# function body is here
# use fp as the base register, e.g., -40(fp)
# sp may change, but fp does not


# before return, restore fp, and then sp
lw      fp, (sp)
addi    sp, sp, 4
ret
```

An example of compiled code, where fp is set to the old sp
cse3666/add.md at master · zhijieshi/cse3666 (github.com)

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Saving comparison results to a register

- Save the comparison result in a register
  - Can be used in combination with `beq, bne`

```
# if (rs1 < rs2) rd = 1; else rd = 0
slt   rd, rs1, rs2    # signed
sltu  rd, rs1, rs2    # unsigned


# compare with immediate
# if (rs1 < immd) rd = 1; else rd = 0
slti  rd, rs1, immd   # signed
sltiu rd, rs1, immd   # unsigned
```

immd is 12 bits long and sign extended, even for unsigned comparison!

Example

```
# pseudoinstruction seqz
seqz  t0, t1        # t0 = (t1 == 0)



# t0 = (s1 >= '0') && (s1 <= '9')
t0 = is_digit(s1)
```

Example

```
# pseudoinstruction seqz
seqz  t0, t1        # t0 = (t1 == 0)

sltui  t0, t1, 1

# t0 = (s1 >= '0') && (s1 <= '9')
t0 = is_digit(s1)

addi  t0, s1, -48       # '0' is 48
sltui t0, t0, 10
```