

**Lecture 4:**

# **Instructions: Language of the Computer (3/2)**

---

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

**Introduction to Computer Architecture**  
**UC Davis EEC 170, Fall 2019**

# From last time ...

## ■ What instructions look like

- RISC-V: 32 bit instructions, different types (R, I, S, and more)
- RISC-V: Instructions either compute something or move something to/from memory
- Last lecture: logical, branch, jump instructions

Assignment Project Exam Help

<https://tutorcs.com>

## ■ Calling procedures

WeChat: cstutorcs

- Arguments and return values
- Jump instructions and return addresses
- Saving registers and RISC-V register conventions
- The stack, and memory regions

# 32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

lui rd, constant

- Copies 20-bit constant to bits [31:12] of rd
- Extends bit 31 to bits [63:32]
- Clears bits [11:0] of rd to 0

lui x19, 976 // 0x003D0

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
---------------------	---------------------	--------------------------	----------------

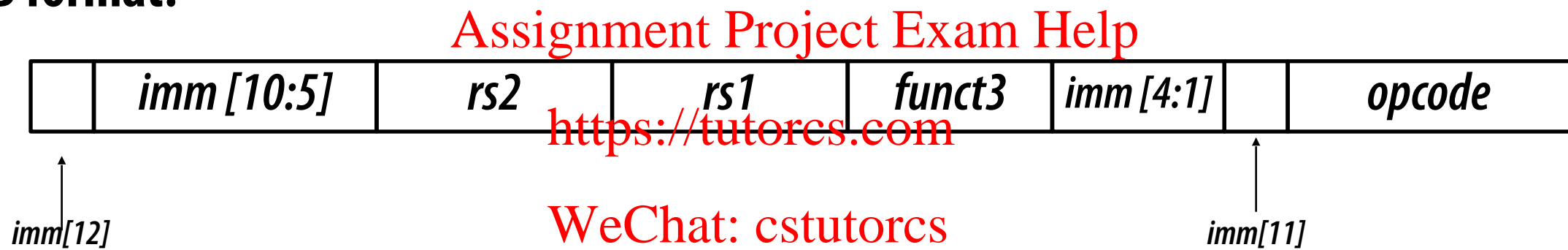
addi x19,x19,128 // 0x500

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000
---------------------	---------------------	--------------------------	----------------

# Branch Addressing

- **Branch instructions specify**
  - **Opcode, two registers, target address**
- **Most branch targets are near branch**
  - **Forward or backward**

- **SB format:**



- **“The address uses an unusual encoding, which simplifies data path design but complicates assembly.”**
- **PC-relative addressing**
  - **Target address = PC + immediate  $\times$  2**
  - **Why 2? “The RISC-V architects wanted to support the possibility of instructions that are 2 bytes long.”**

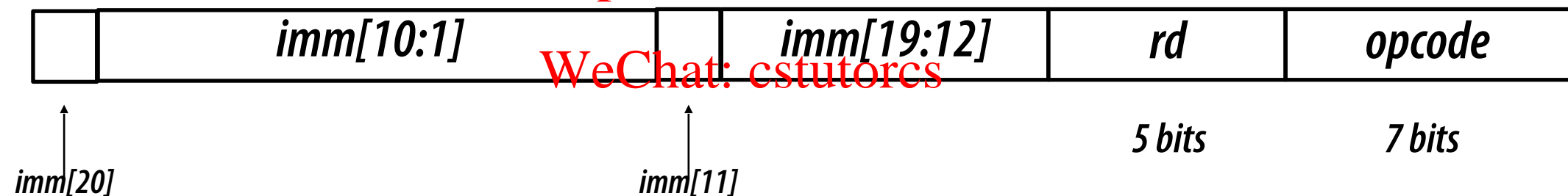
# Jump Addressing

- Jump and link (jal) target uses 20-bit immediate for larger range
  - Also uses PC-relative addressing
  - Use `jal x0, Label` to jump (goto) to Label (unconditional jump)

Assignment Project Exam Help

- UJ format:

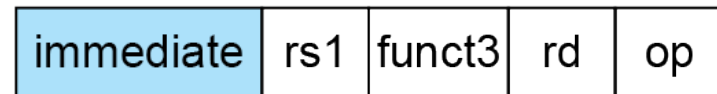
<https://tutorcs.com>



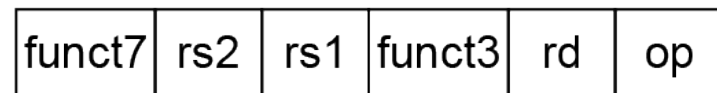
- For long jumps, eg, to 32-bit absolute address
  - `lui`: load address[31:12] to temp register
  - `jalr`: add address[11:0] and jump to target

# RISC-V Addressing Summary

## 1. Immediate addressing



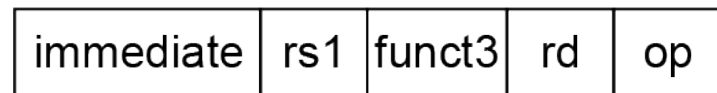
## 2. Register addressing



Registers

Register

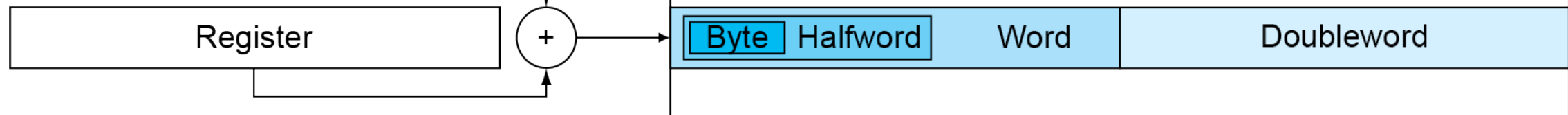
## 3. Base addressing



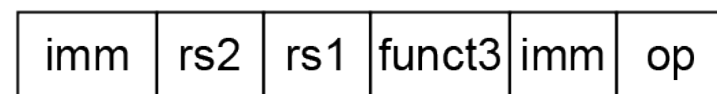
<https://tutorcs.com>

WeChat: estutores

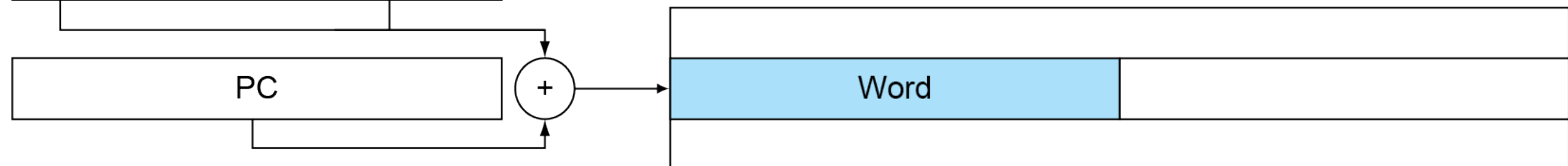
Memory



## 4. PC-relative addressing



Memory



# RISC-V Encoding Summary

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

WeChat: cstutorcs

# Synchronization

- **Two processors sharing an area of memory**
  - **P1 writes, then P2 reads**
  - **Data race if P1 and P2 don't synchronize**
    - **Result depends of order of accesses**
- **Example (next slide):** <https://tutorcs.com>
  - **load balance from memory to register**
  - **add \$20 to register value**
  - **store balance from register to memory**

Assignment Project Exam Help

WeChat: cstutorcs



# Synchronization example

Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the deposit() step typically breaks down into low-level processor instructions:

```
get balance (balance=0)
add 1
write back the result (balance=1)
```

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

```
A get balance (balance=0)
A add 1
A write back the result (balance=1)
B get balance (balance=1)
B add 1
B write back the result (balance=2)
```

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

```
A get balance (balance=0)
A add 1
A write back the result (balance=1)
B get balance (balance=0)
B add 1
B write back the result (balance=1)
```

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: csfutures

# Synchronization

- **Two processors sharing an area of memory**
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- **Hardware support required**
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- **Could be a single instruction**
  - E.g., atomic swap of register  $\leftrightarrow$  memory
  - Or an atomic pair of instructions

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Synchronization in RISC-V

- **Load reserved:** `lr.d rd, (rs1)`
  - Load from address in `rs1` to `rd`
  - Place reservation on memory address
- **Store conditional:** `sc.d rd, (rs1), rs2`
  - Store from `rs2` (the value to be stored) to address in `rs1`
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in `rd`
  - Fails if location is changed
    - Returns non-zero value in `rd`

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Synchronization in RISC-V

- **Example 1: atomic swap (to test/set lock variable)**

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 // X11 = status
        bne x11,x0,again  // branch if store failed
        addi x23,x10,0    // X23 = loaded value
                        // X23 and Mem[x20] have swapped
```

Assignment Project Exam Help

<https://tutorcs.com>

- **Example 2: lock**

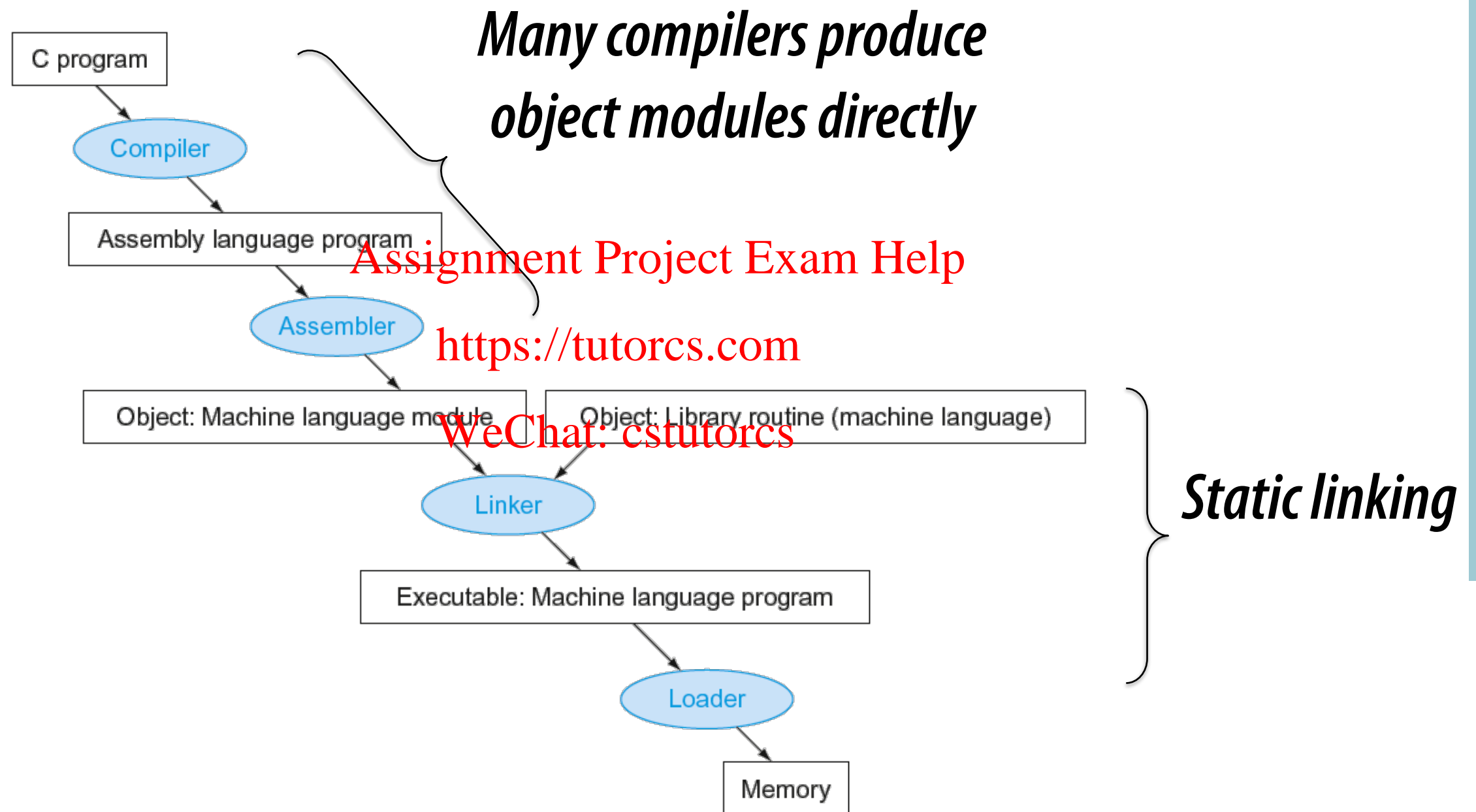
WeChat: cstutorcs

```
        addi x12,x0,1    // “locked” == 1
again:  lr.d x10,(x20)    // read lock
        bne x10,x0,again // check if it is 0 yet
        sc.d x11,(x20),x12 // attempt to store “locked” == 1
        bne x11,x0,again // branch if fails
```

- **Unlock:**

```
sd      x0,0(x20)        // free lock
```

# Translation and Startup



# Assembler tasks

- Translate assembly instructions into binary
- Do stuff that makes assembly writers' job easier
  - Translate labels to offsets (beq a1, a2, Label)
  - Pseudoinstructions:
    - `li` is "load immediate" (load a number into a register), not in instruction set
    - If it's small enough, assembler generates `addi`
    - If it's bigger, `lui` then `addi`
    - `mv` is a copy instruction (not in instruction set)

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Producing an Object Module

- **Assembler (or compiler) translates program into machine instructions**
- **Provides information for building a complete program from the pieces (following is Unix):**
  - **Header: describes contents of object module**
  - **Text segment: machine code**
  - **Static data segment: data allocated for the life of the program**
  - **Relocation info: which instructions/data words depend on absolute addresses in this program?**
    - **Address space layout randomization (e.g.) requires this**
  - **Symbol table: labels that are not defined (external references)**
  - **Debug info: for associating with source code**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Linking Object Modules

- Much faster to link than recompile
- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
- Nice example in the book (p. 128)

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs



# Loading a Program

- **Load from image file on disk into memory**
  1. **Read header to determine segment sizes**
  2. **Create virtual address space**
  3. **Copy text and initialized data into memory**
    - **Or set page table entries so they can be faulted in**
  4. **Set up arguments on stack**
  5. **Initialize registers (including sp, fp, gp)**
  6. **Jump to startup routine**
    - **Copies arguments to x10, ... and calls main**
    - **When main returns, do exit syscall**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Dynamic Linking

- **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**

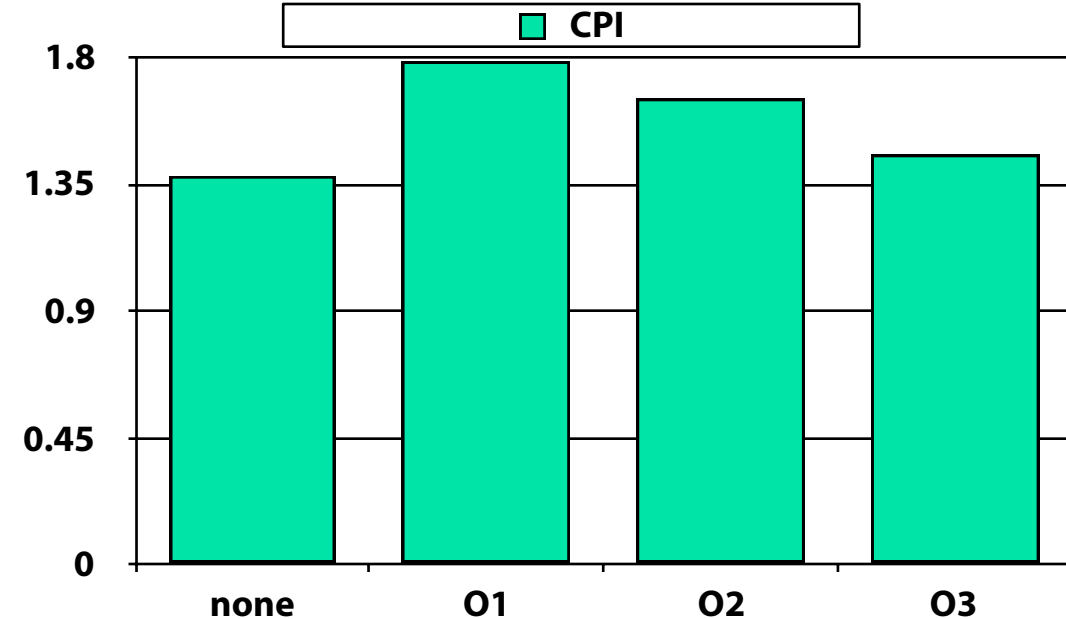
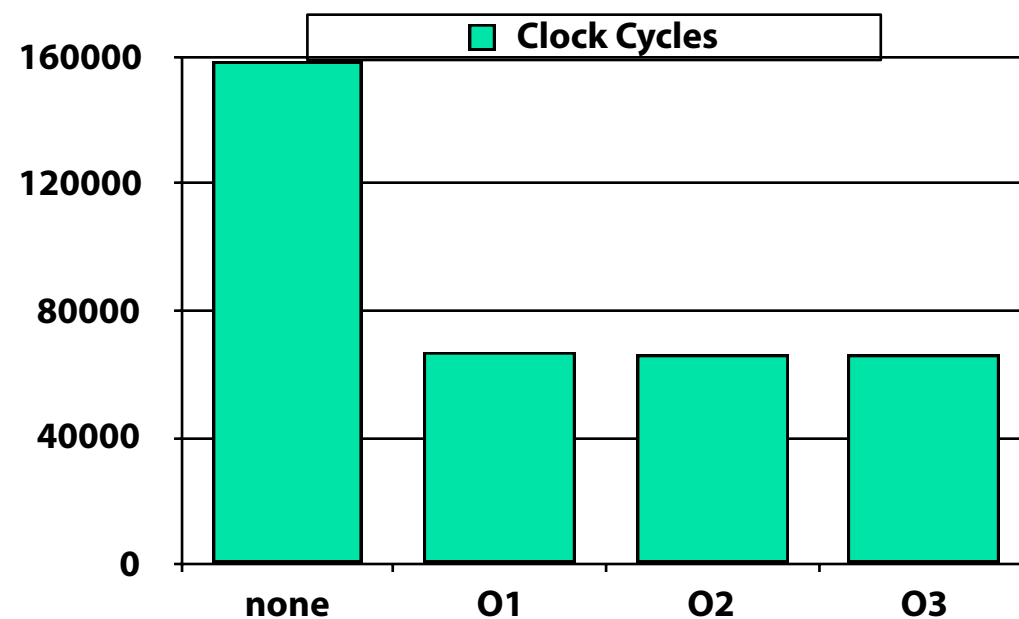
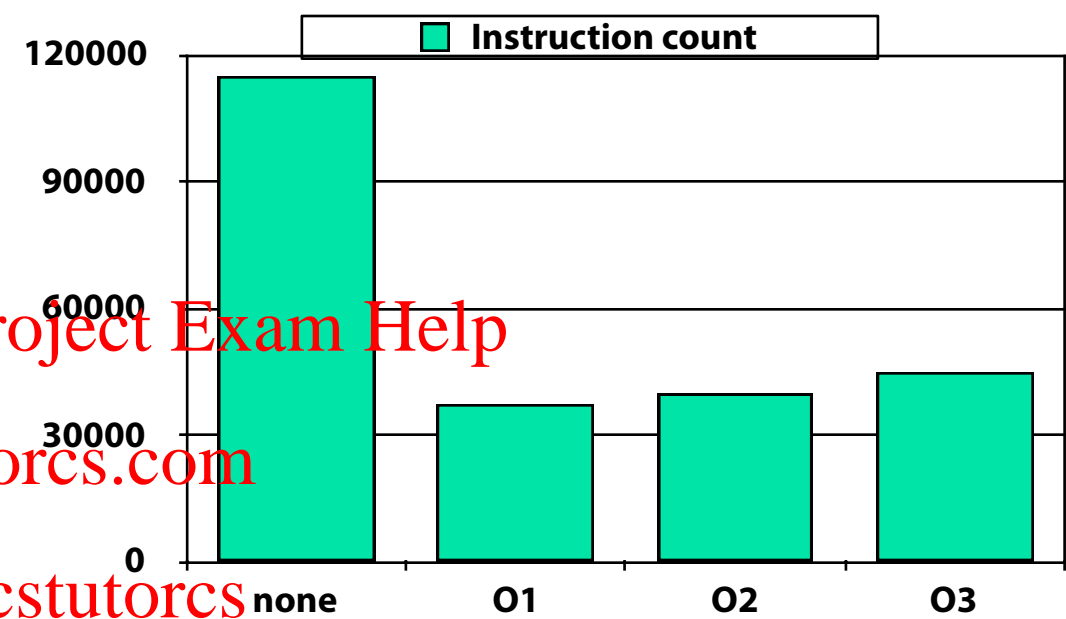
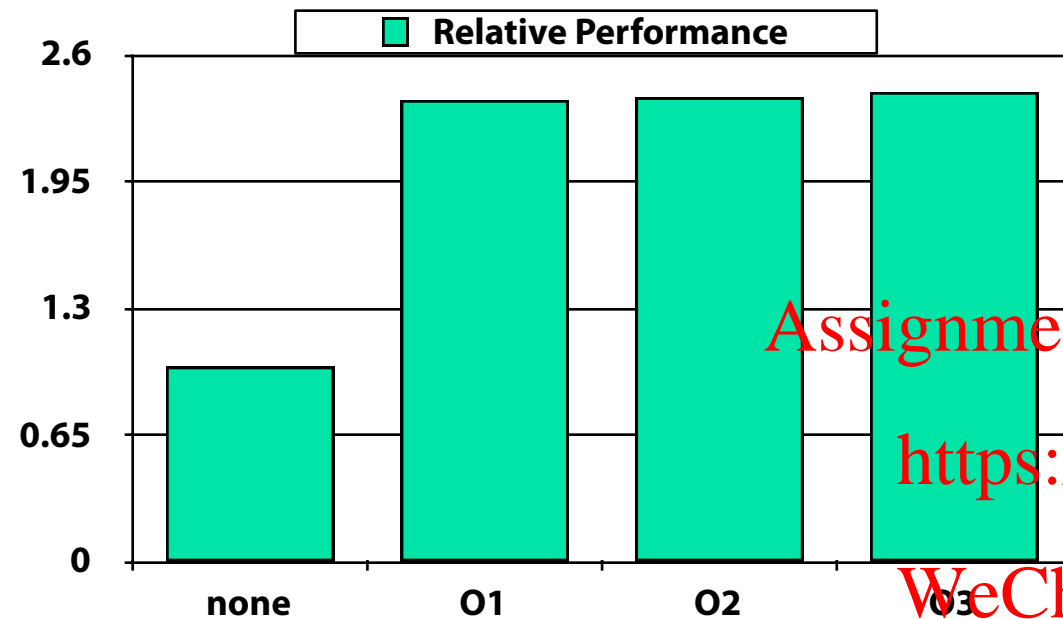
Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Effect of Compiler Optimization

## ■ Compiled with gcc for Pentium 4 under Linux



Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Effect of Language and Algorithm



Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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!**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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: RISC-V vs. MIPS

## 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																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												

## 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											
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																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						

# 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

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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  $\neq$  market success

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Basic x86 Addressing Modes

## ■ Two operands per instruction

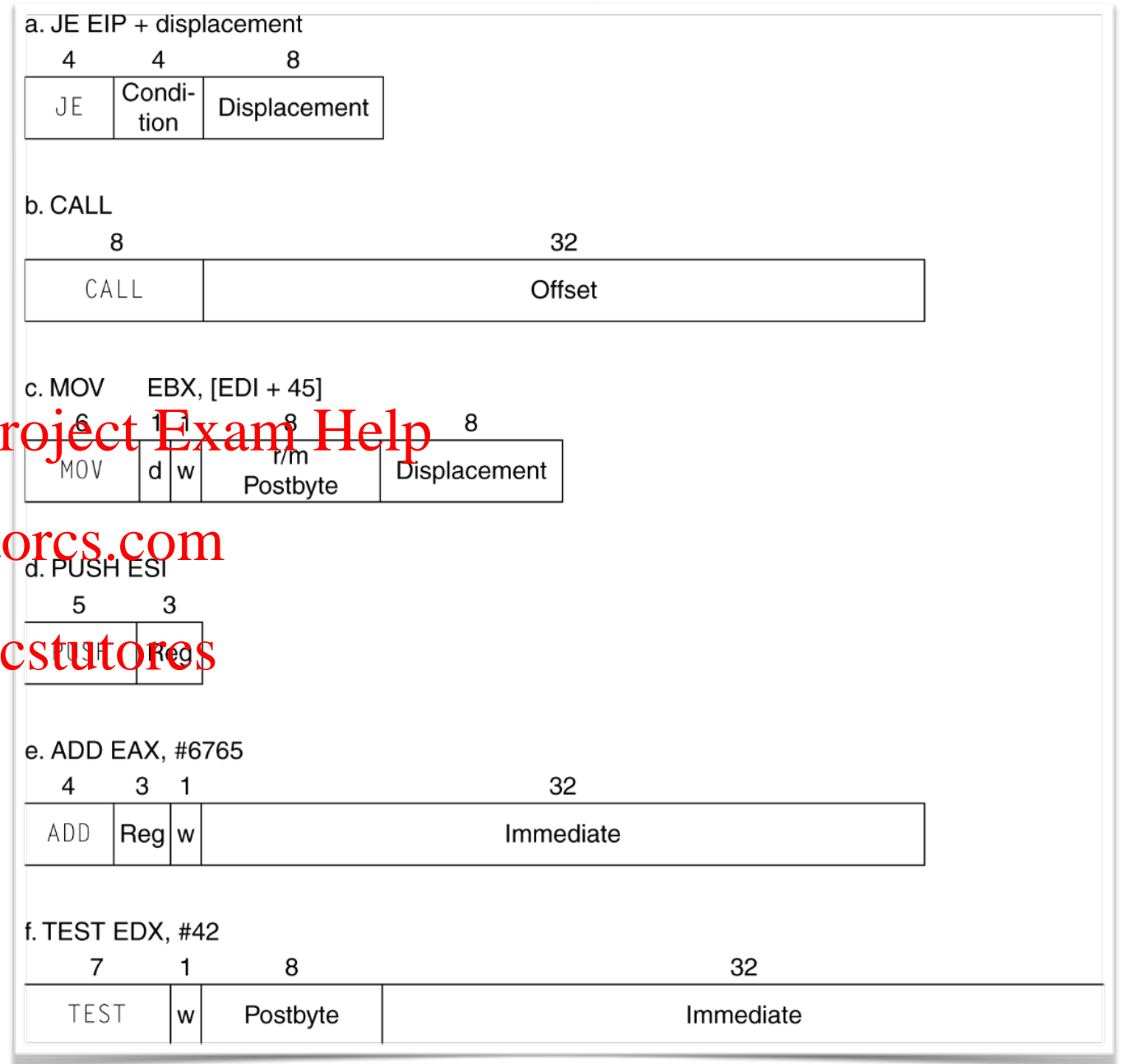
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

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

# x86 Instruction Encoding

## ■ Variable length encoding

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



Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: tutorcs

# Other RISC-V Instructions

## ■ Base integer instructions (RV64I)

- Those previously described, plus
- **auipc rd, imm** //  $rd = (imm \ll 12) + pc$ 
  - follow by **jalc** (adds 12-bit imm) for long jump
- **slt, sltu, slti, sltui**: set less than (like MIPS)
- **addw, subw, addiw**: 32-bit add/sub
- **sllw, srlw, srliw, 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**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs



# 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**

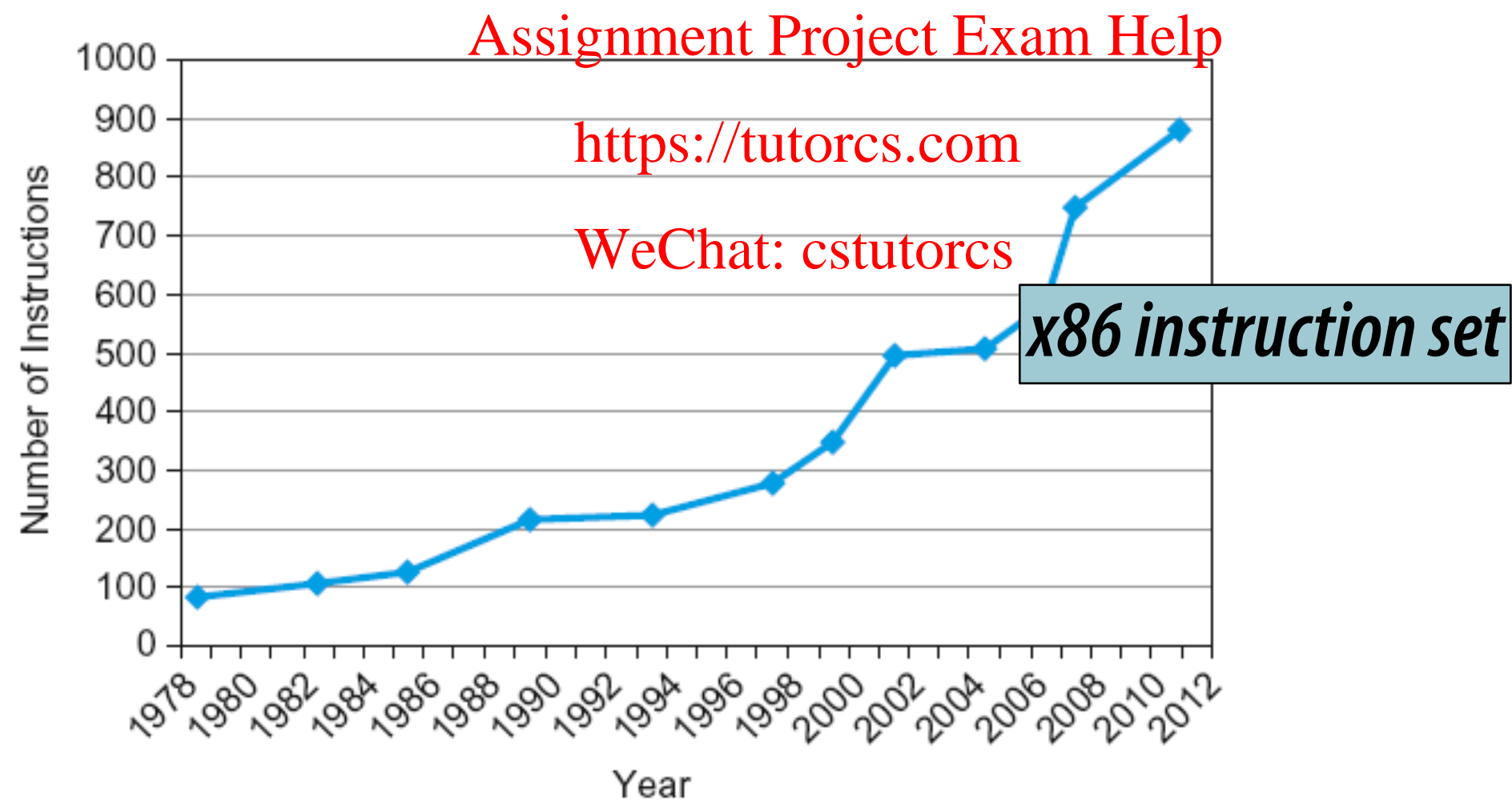
Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Fallacies

- **Backward compatibility  $\Rightarrow$  instruction set doesn't change**
  - **But they do accrue more instructions**



# Pitfalls

- **Sequential words are not at sequential addresses**
  - **MIPS-V addresses are byte addresses**
  - **Increment by 4 or 8, 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**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# Concluding Remarks

- **Design principles**
  - 1. **Simplicity favors regularity**
  - 2. **Smaller is faster**
  - 3. **Good design demands good compromises**
- **Make the common case fast**
- **Layers of software/hardware**
  - **Compiler, assembler, hardware**
- **RISC-V: typical of RISC ISAs**
  - **c.f. x86**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# We don't have time for the next few slides

- Great example though!

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# C Sort Example

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

# The Procedure Swap

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
```

<https://tutorcs.com>  
WeChat: cstutorcs

# The Sort Procedure in C

## ■ 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**

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs



# The Outer Loop

## ■ Skeleton of outer loop:

- `for (i = 0; i < n; i += 1) {`

```
li    x19, 0           // i = 0
```

```
for1tst:
```

```
bge   x19, x11, exit1   // go to exit1 if x19 ≥ x11 (i ≥ n)
```

<https://tutorcs.com>

[WeChat: cstutorcs](#)

```
(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){**

```
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 = i + (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   x21, x10      // copy parameter x10 into x21
```

```
mv   x22, x11      // copy parameter x11 into x22
```

```
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:
```

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

# 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:

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

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs