Lecture 3:

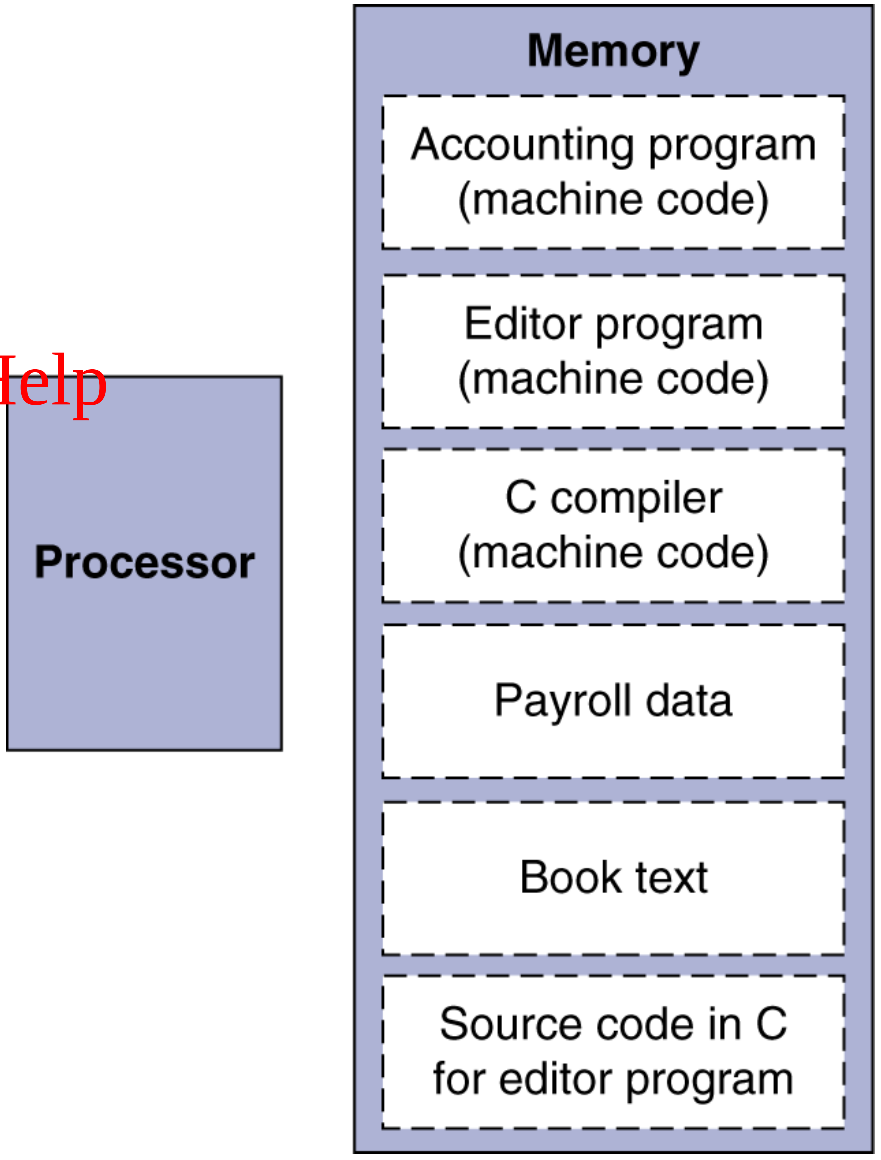
Instructions: Language of the Computer (2/2)

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

Stored Program Computers

The BIG Picture

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, . . .
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

From last time …

- What instructions look like
  - `add, sub, ld, sw, addi`
  - RISC-V: 32 bit instructions, different types (R, I, S)
  - RISC-V: Instructions either compute something or move something to/from memory
  - Converting bits <-> instructions
- Numbers
  - Integers, signed/unsigned integers, sign extension
  - Decimal, binary, hexadecimal
  - Converting bits <-> numbers

Logical Operations

- **Instructions for bitwise manipulation**

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

- ***Useful for extracting and inserting groups of bits in a word***

Shift Operations

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)
  - Also arithmetic right shifts that fill with sign bit (`srai`)
    - Why not an arithmetic left shift?

funct6	*immed*	*rs1*	*funct3*	*rd*	*opcode*
6 bits	*6 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0
- `and x9,x10,x11`

```
x10    00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x11    00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x9     00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9,x10,x11
```

x10	`00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000`
x11	`00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000`
x9	`00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000`

XOR Operations

- **Differencing operation**
  - **Set some bits to 1, leave others unchanged**

```
xor x9,x10,x12  // NOT operation
```

```
x10   00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x12   11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

x9    11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111
```

Conditional Operations

- Branch to a labeled instruction if a condition is true
    - Otherwise, continue sequentially

- `beq rs1, rs2, L1`
    - if (rs1 == rs2) branch to instruction labeled L1

- `bne rs1, rs2, L1`
    - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

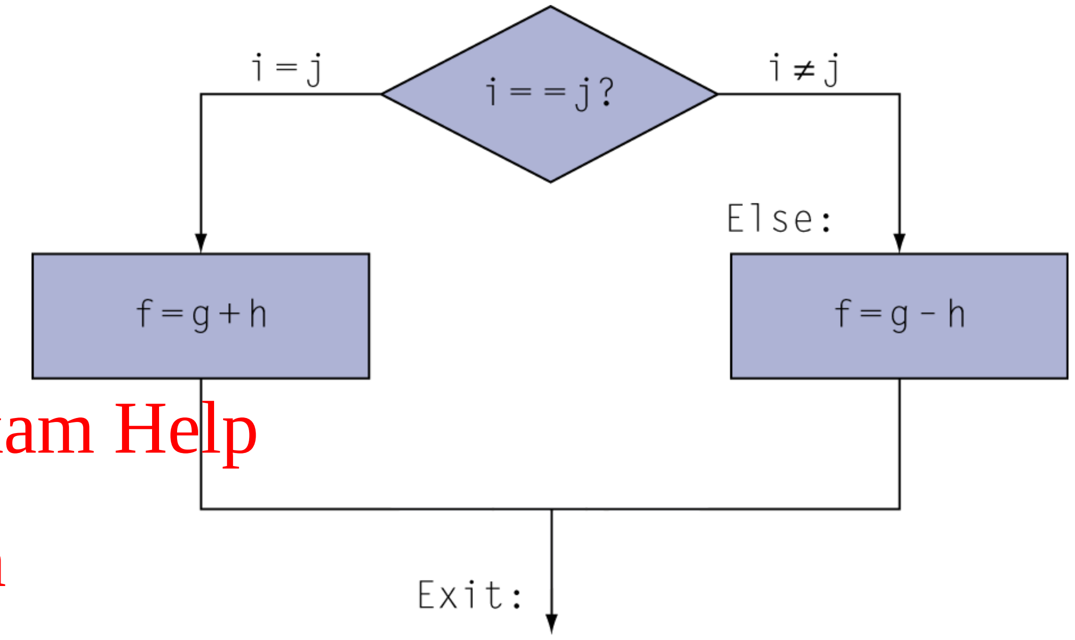
- **C code:**

```
if (i==j) f = g+h;
else f = g-h;
```

  - **f, g, … in x19, x20, …**

- **Compiled RISC-V code:**

```
      bne x22, x23, Else
      add x19, x20, x21
      beq x0, x0, Exit // unconditional
Else: sub x19, x20, x21
Exit: …
```



Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

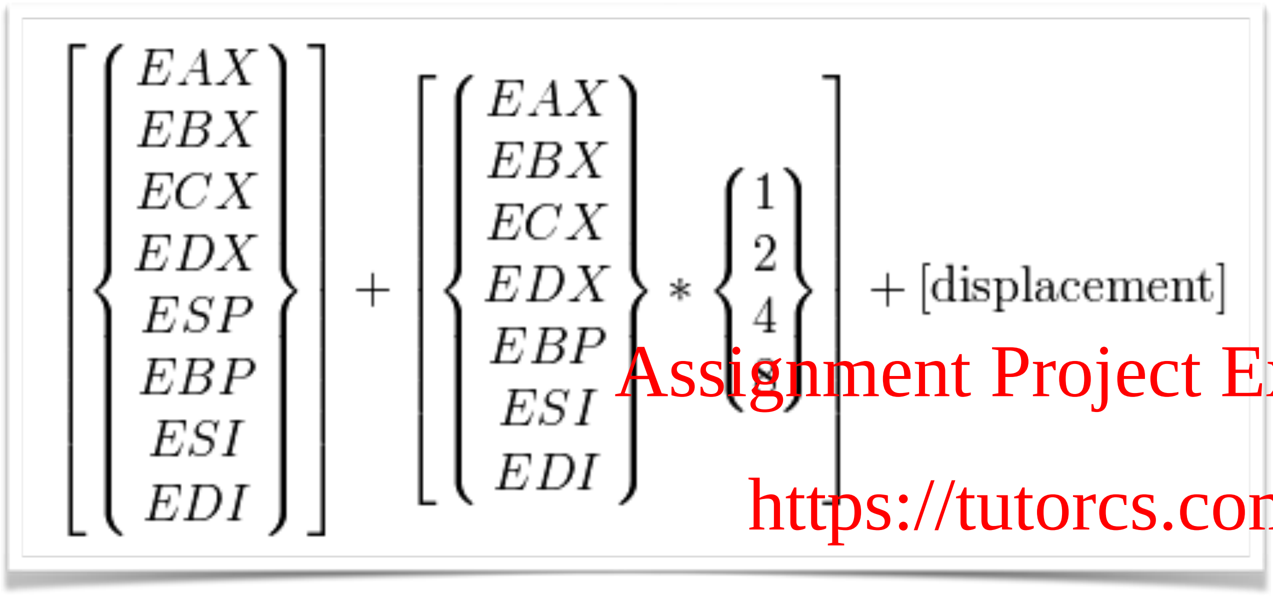
  - i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)     // could we optimize this with an immediate?
      bne  x9, x24, Exit

      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

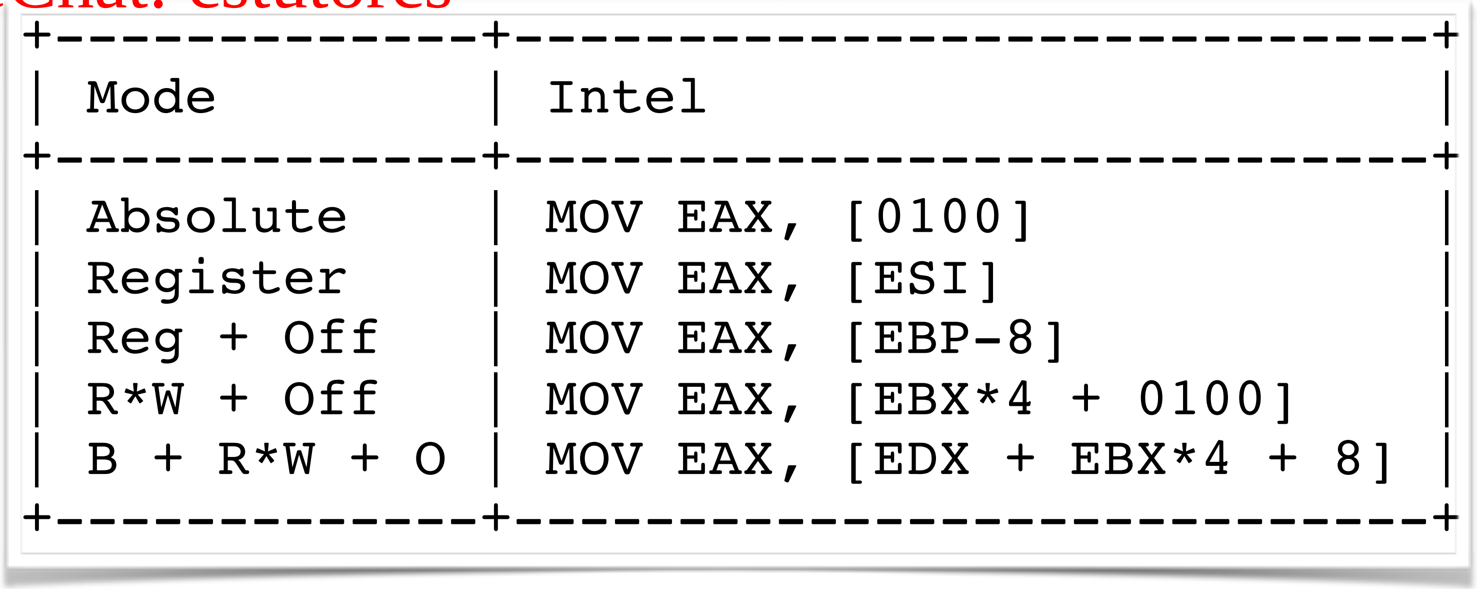
Aside on addressing modes

- x86 has many more addressing modes than RISC-V



$$\left[\begin{Bmatrix} EAX\\ EBX\\ ECX\\ EDX\\ ESP\\ EBP\\ ESI\\ EDI \end{Bmatrix}\right] + \left[\begin{Bmatrix} EAX\\ EBX\\ ECX\\ EDX\\ EBP\\ ESI\\ EDI \end{Bmatrix} * \begin{Bmatrix} 1\\ 2\\ 4\\ 8 \end{Bmatrix}\right] + [\text{displacement}]$$

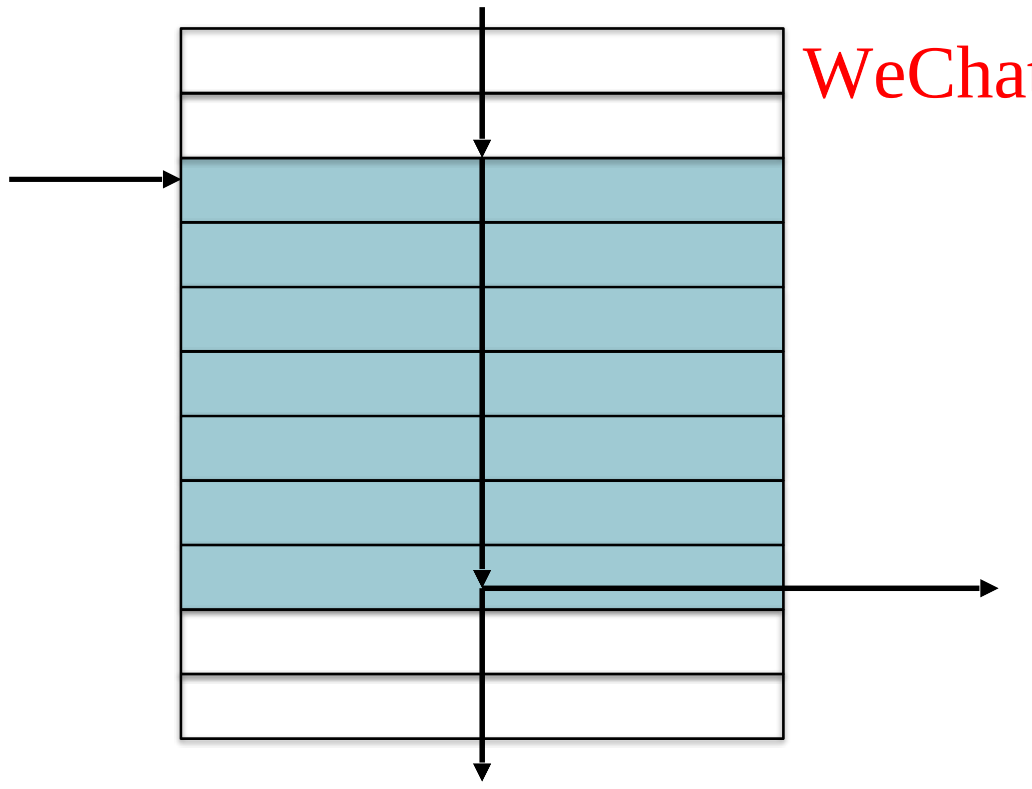
- RISC-V can do:
  - register
  - reg+off
  - (small) absolute



Mode	Intel
Absolute	MOV EAX, [0100]
Register	MOV EAX, [ESI]
Reg + Off	MOV EAX, [EBP-8]
R*W + Off	MOV EAX, [EBX*4 + 0100]
B + R*W + O	MOV EAX, [EDX + EBX*4 + 8]

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- *A compiler identifies basic blocks for optimization*
- *An advanced processor can accelerate execution of basic blocks*

More Conditional Operations

- `blt rs1, rs2, L1`
  - **if (rs1 < rs2) branch to instruction labeled L1**
- `bge rs1, rs2, L1`
  - **if (rs1 >= rs2) branch to instruction labeled L1**

- **Example**
  - `if (a > b) a += 1;` **// a in x22, b in x23**

```
    bge  x23, x22, Exit  // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23` // signed
    - –1 < +1
  - `x22 > x23` // unsigned
    - +4,294,967,295 > +1

Procedure Calling

- Steps required
  - Place parameters in registers x10 to x17
  - Transfer control to procedure
  - Acquire storage for procedure
    - “Storage” may be both register and memory space
  - Perform procedure’s operations
  - Place result in register for caller
  - Return to place of call (address in x1)

Procedure Call Instructions

- **Procedure call: jump and link**

  `jal x1, ProcedureLabel`

  - **Address of following instruction put in x1**
  - **Jumps to target address**

- **Procedure return: jump and link register**

  `jalr x0, 0(x1)`

  - **Like jal, but jumps to 0 + address in x1**
  - **Use x0 as rd (x0 cannot be changed)**
  - **Can also be used for computed jumps**
    - **e.g., for case/switch statements**

Aside: Data Types in C

- The actual size of the integer types varies by implementation. The standard only requires size relations between the data types and minimum sizes for each data type:
- The relation requirements are that the long long is not smaller than long, which is not smaller than int, which is not smaller than short. As char's size is always the minimum supported data type, no other data types (except bit-fields) can be smaller.
- The minimum size for char is 8 bits, the minimum size for short and int is 16 bits, for long it is 32 bits and long long must contain at least 64 bits.
- The type int should be the integer type that the target processor is most efficiently working with. This allows great flexibility: for example, all types can be 64-bit. However, several different integer width schemes (data models) are popular. Because the data model defines how different programs communicate, a uniform data model is used within a given operating system application interface.
- In practice, char is usually eight bits in size and short is usually 16 bits in size (as are their unsigned counterparts). This holds true for platforms as diverse as 1990s SunOS 4 Unix, Microsoft MS-DOS, modern Linux, and Microchip MCC18 for embedded 8-bit PIC microcontrollers. POSIX requires char to be exactly eight bits in size.

https://en.wikipedia.org/wiki/C_data_types

Leaf Procedure Example

"leaf procedures" make no function calls

- C code:

```
long long int leaf_example (
    long long int g, long long int h,
    long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

`long long int` *guarantees at least a 64-bit integer*

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Callee needs to save x5, x6, x20 on "stack" (magic data structure, we will describe shortly)

Leaf Procedure Example

- **RISC-V code:**

```
leaf_example:
   addi sp,sp,-24
   sd   x5,16(sp)
   sd   x6,8(sp)
   sd   x20,0(sp)
   add  x5,x10,x11
   add  x6,x12,x1
   sub  x20,x5,x6
   addi x10,x20,0
   ld   x20,0(sp)
   ld   x6,8(sp)
   ld   x5,16(sp)
   addi sp,sp,24
   jalr x0,0(x1)
```

Save x5, x6, x20 on stack (caller might need those values)

$x5 = g + h$

$x6 = i + j$

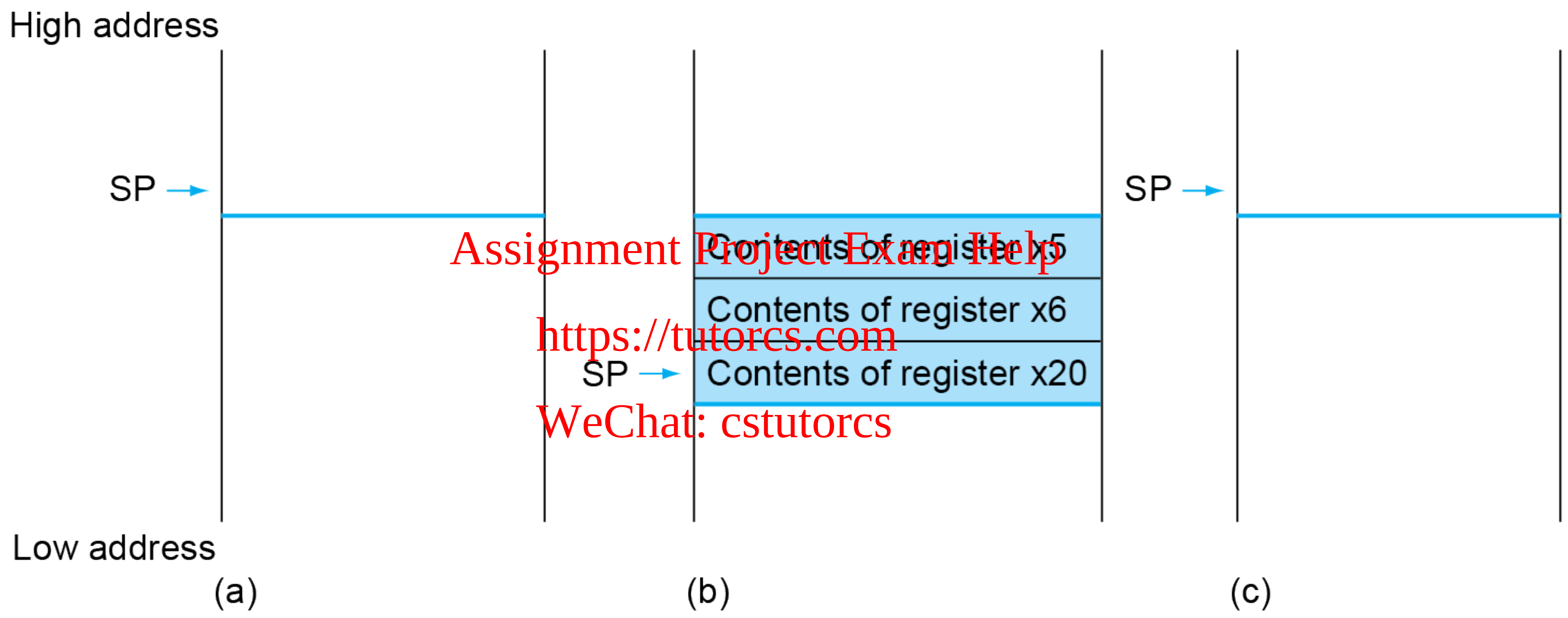
$f = x5 - x6$

copy f to return register

Restore x5, x6, x20 from stack

Return to caller

Local Data on the Stack



Register Usage (Convention)

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee

- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument *n* in x10
  - Result in x10

Leaf Procedure Example

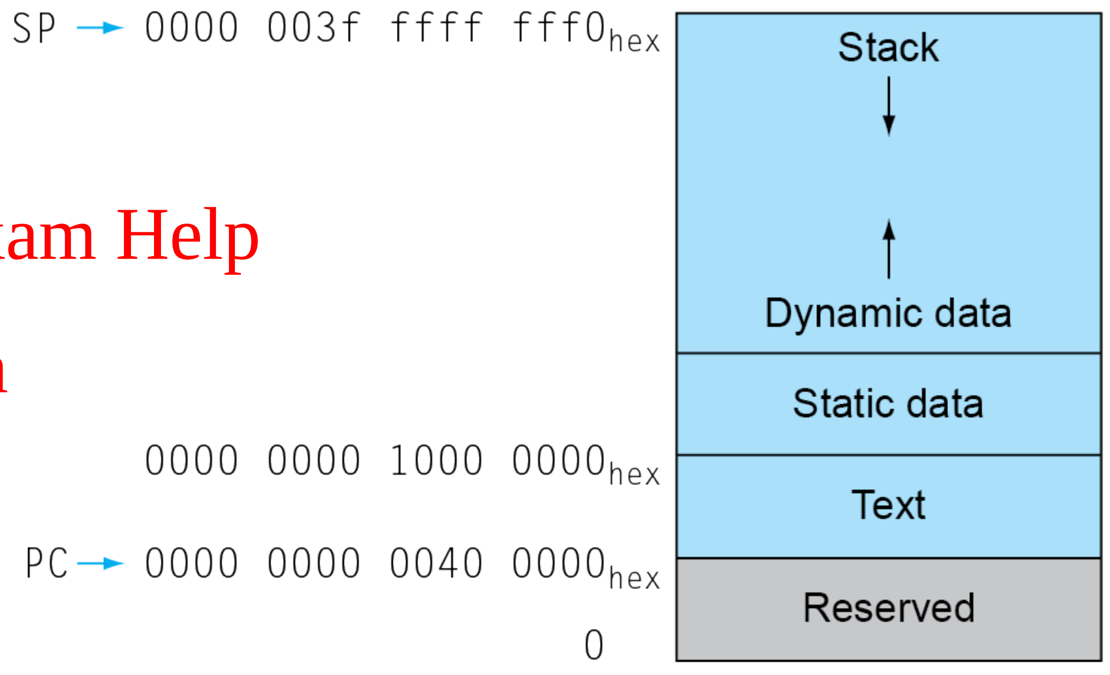
```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- **RISC-V code:**

`fact:`	
`addi sp,sp,-16`	
`sd   x1,8(sp)`	***Save return address and n on stack***
`sd   x10,0(sp)`	
`addi x5,x10,-1`	***x5 = n - 1***
`bge  x5,x0,L1`	***if n >= 1, go to L1***
`addi x10,x0,1`	***Else, set return value to 1***
`addi sp,sp,16`	***Pop stack, don't bother restoring values***
`jalr x0,0(x1)`	***Return***
`L1: addi x10,x10,-1`	***n = n - 1***
`jal  x1,fact`	***call fact(n-1), write next instruction's address into x1, result will be in x10***
`addi x6,x10,0`	***move result of fact(n - 1) to x6***
`ld   x10,0(sp)`	***Restore caller's n***
`ld   x1,8(sp)`	***Restore caller's return address***
`addi sp,sp,16`	***Pop stack***
`mul  x10,x10,x6`	***return n * fact(n-1)***
`jalr x0,0(x1)`	***return***

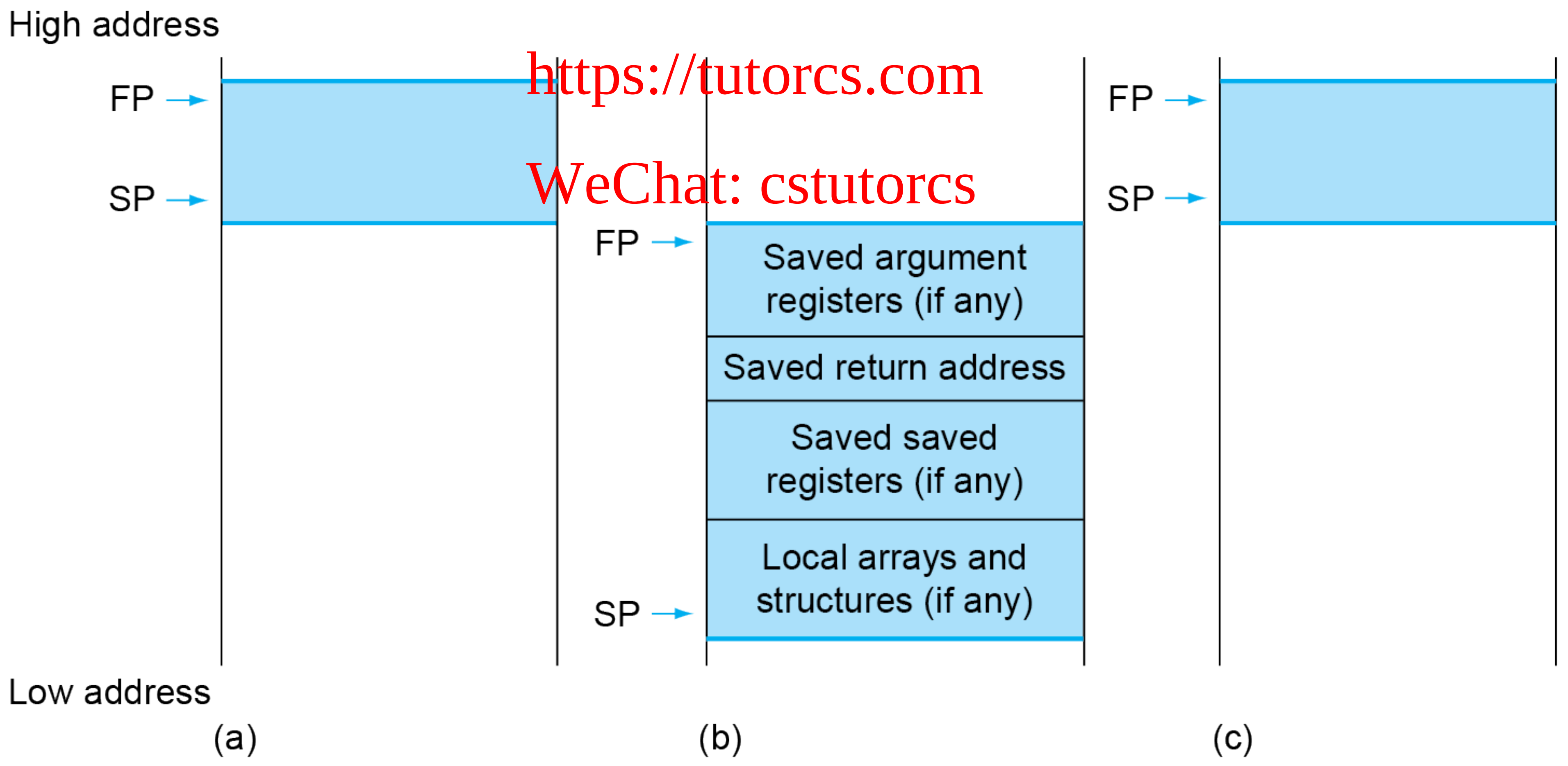
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage


High address
FP
SP
FP
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
SP
FP
SP
Low address
(a)
(b)
(c)

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[]) {
  size_t i;
  i = 0;
  while ((x[i]=y[i]) != ‘\0’)
    i += 1;
}

// C idiom: while (*x++ = *y++);
```

String Copy Example

- **RISC-V code:**

```
strcpy:
 addi sp,sp,-8   // adjust stack for 1 doubleword
 sd   x19,0(sp)  // push x19
 add  x19,x0,x0  // i=0
L1: add x5,x19,x10 // x5 = addr of y[i]
 lbu  x6,0(x5)   // x6 = y[i]
 add  x7,x19,x10 // x7 = addr of x[i]
 sb   x6,0(x7)   // x[i] = y[i]
 beq  x6,x0,L2   // if y[i] == 0 then exit
 addi x19,x19, 1 // i = i + 1
 jal  x0,L1      // next iteration of loop
L2: ld x19,0(sp) // restore saved x19
 addi sp,sp,8    // pop 1 doubleword from stack
 jalr x0,0(x1)   // and return
```

32-bit Constants

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

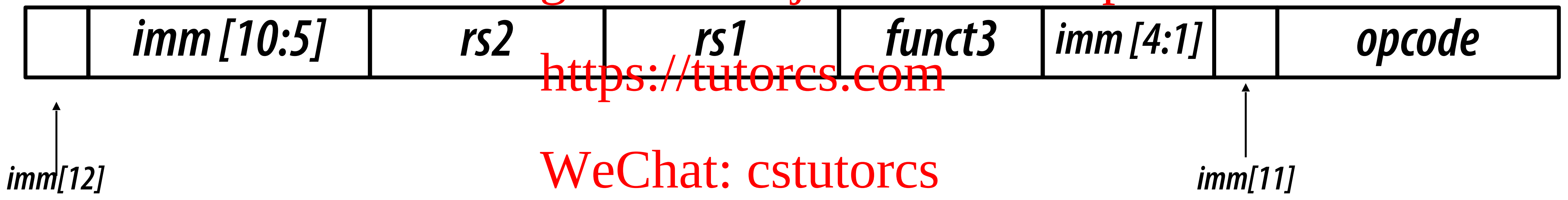
```
lui rd, constant
```

  - 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

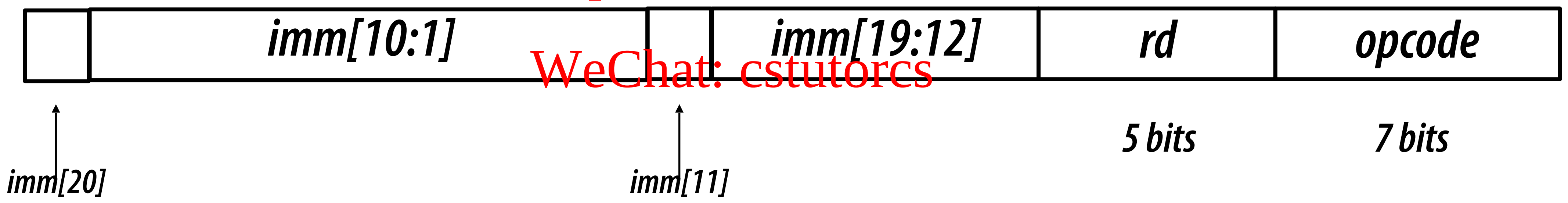
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:



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

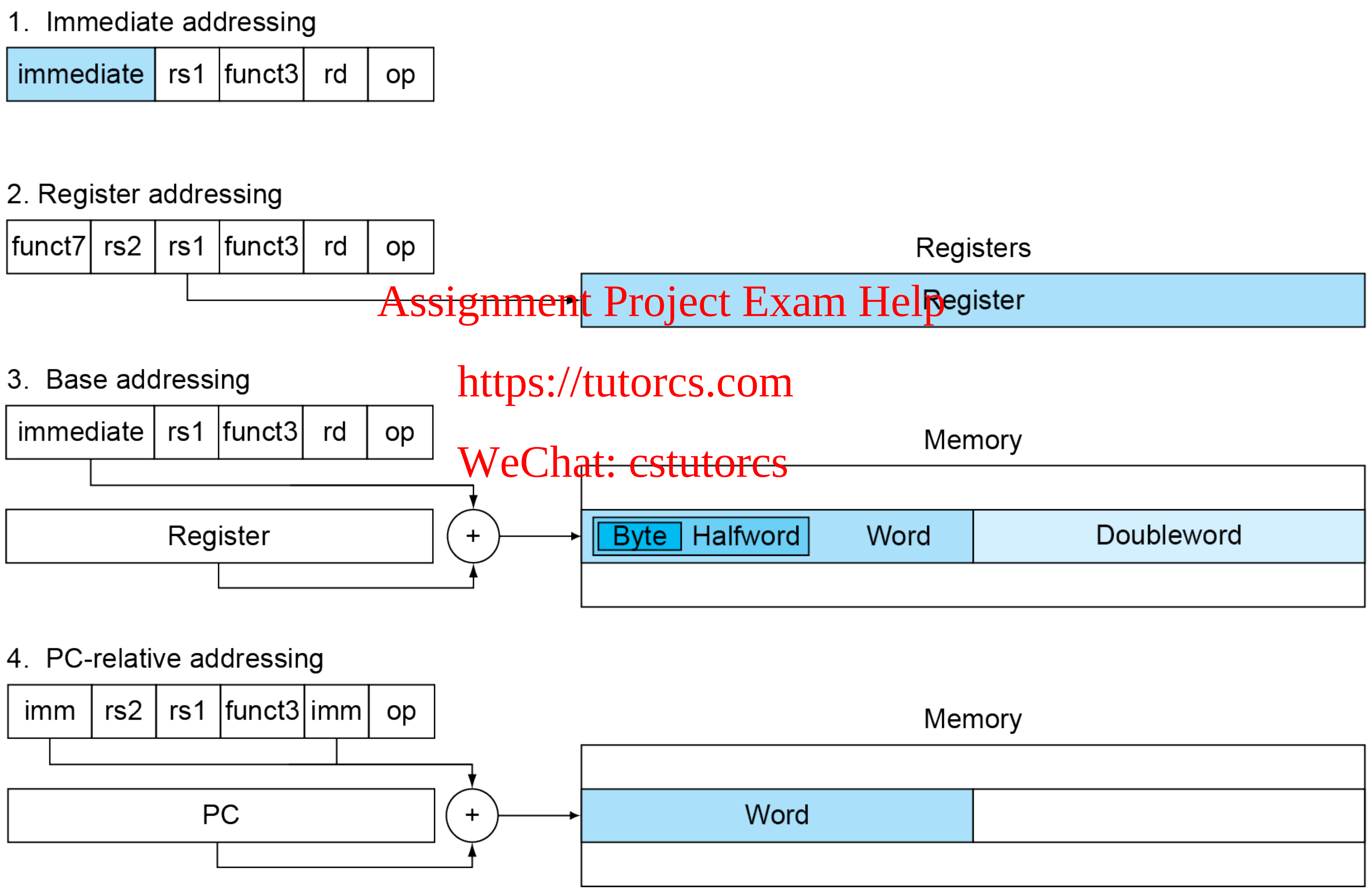
Jump Addressing

- 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)
- UJ format:



- 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



RISC-V Encoding Summary

Name	Field						Comments
(Field Size)	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

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):
  - load balance from memory to register
  - add $20 to register value
  - store balance from register to memory

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

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 ↔ memory
  - Or an atomic pair of instructions

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**

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
```

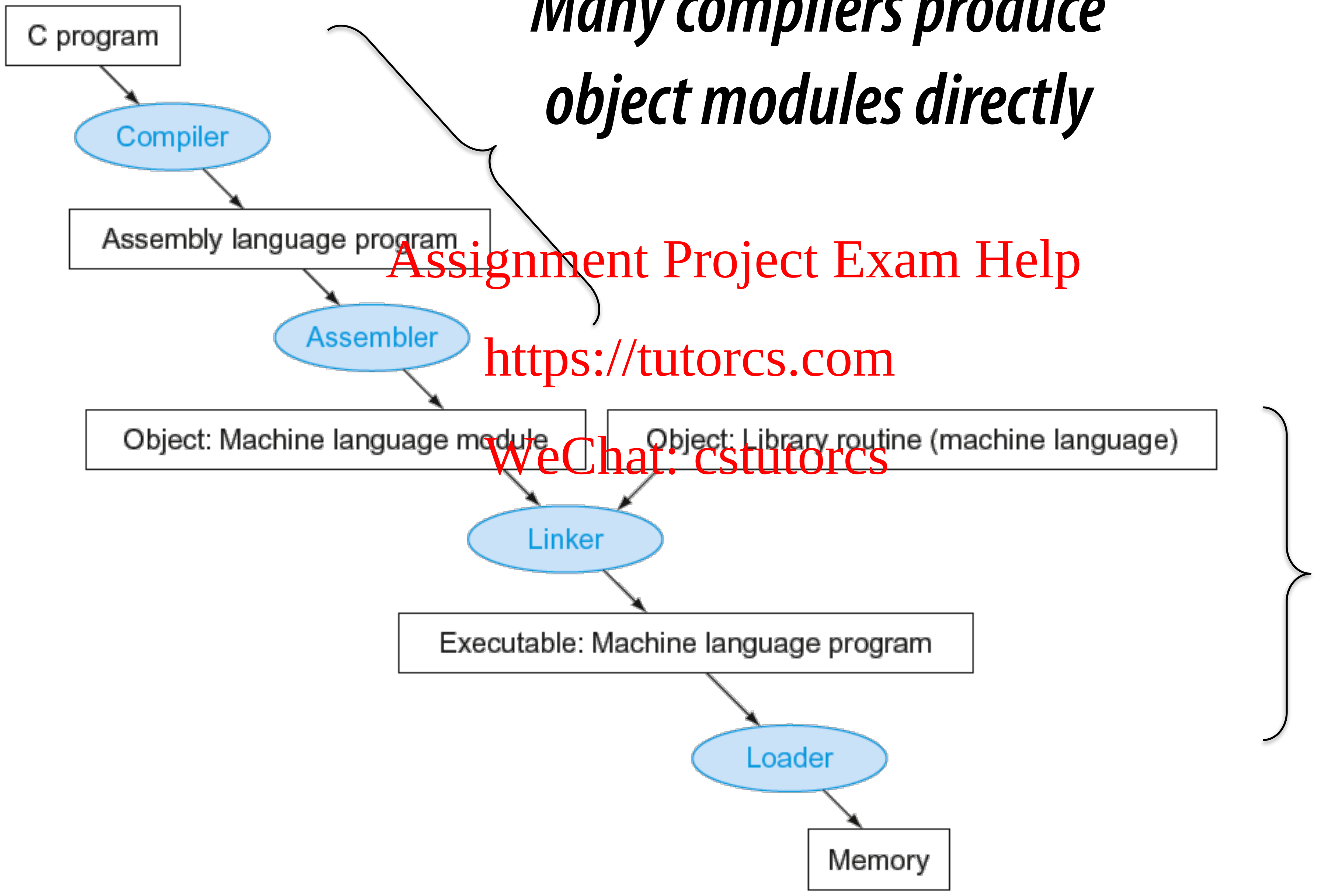
- **Example 2: lock**

```
        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


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

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)

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

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)

Loading a Program

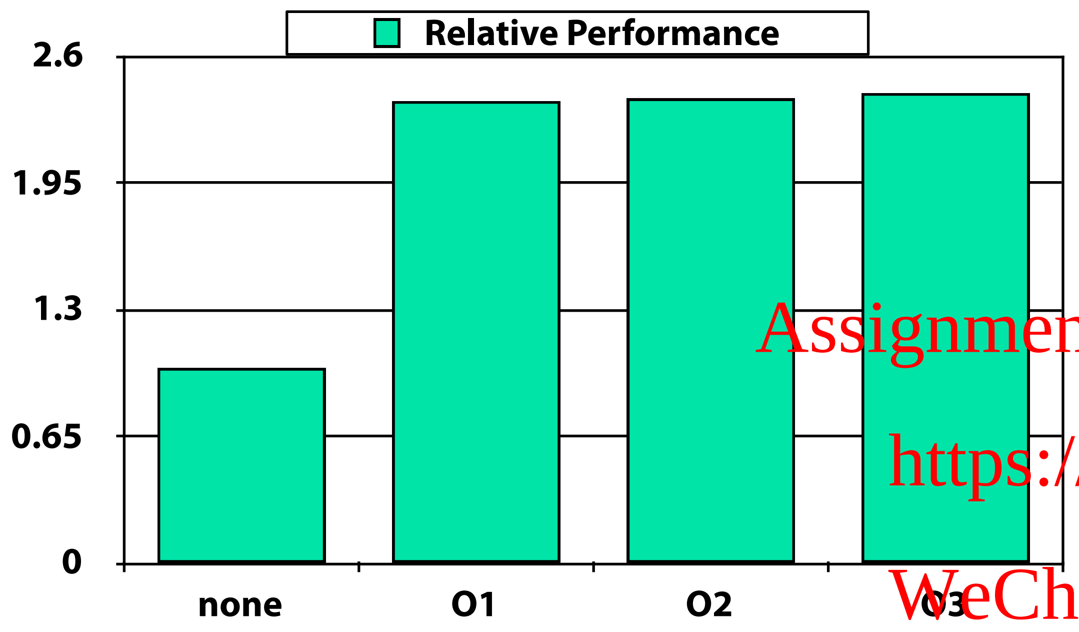
- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including sp, fp,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

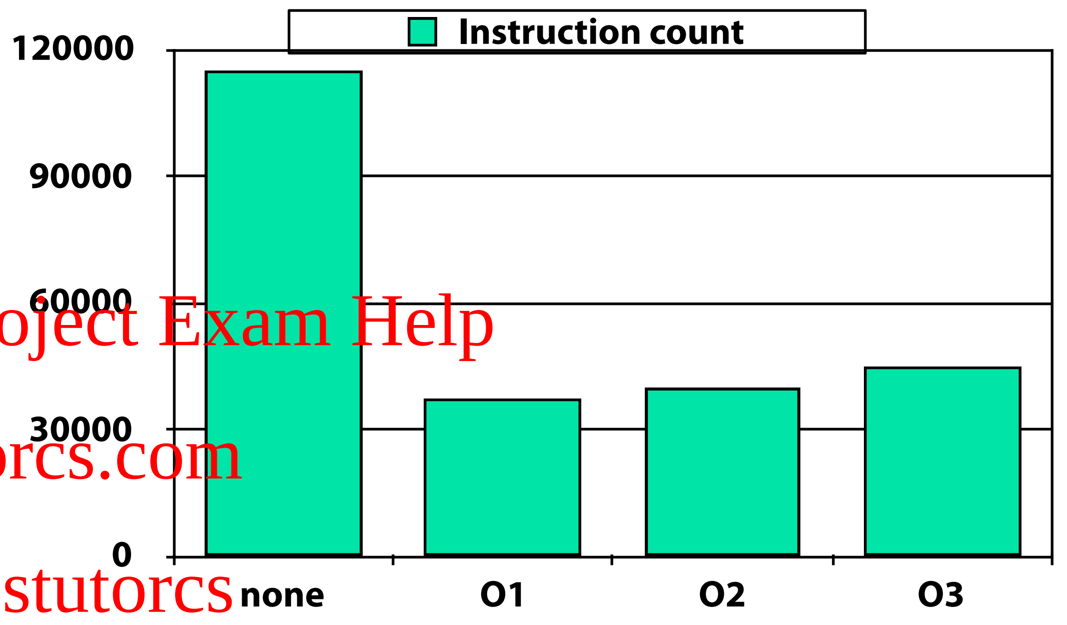
Dynamic Linking

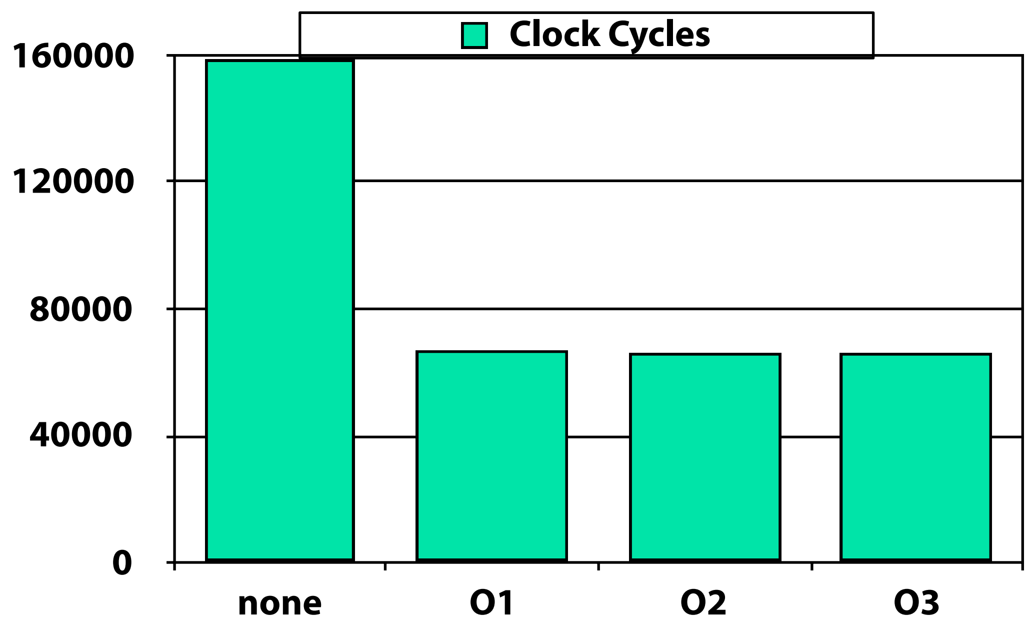
- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

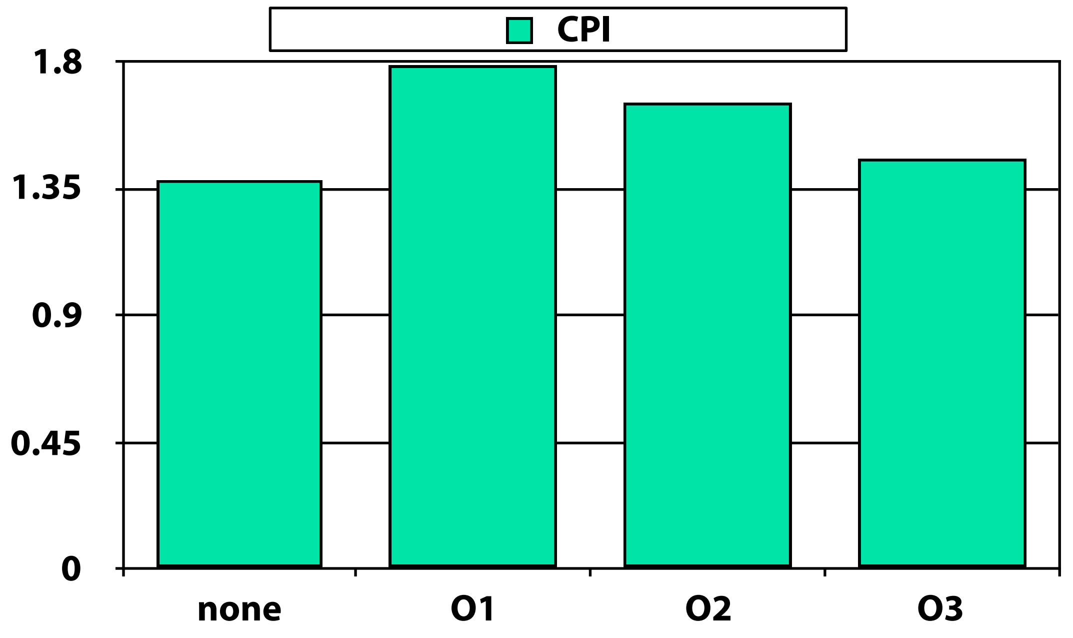
Effect of Compiler Optimization

- Compiled with gcc for Pentium 4 under Linux

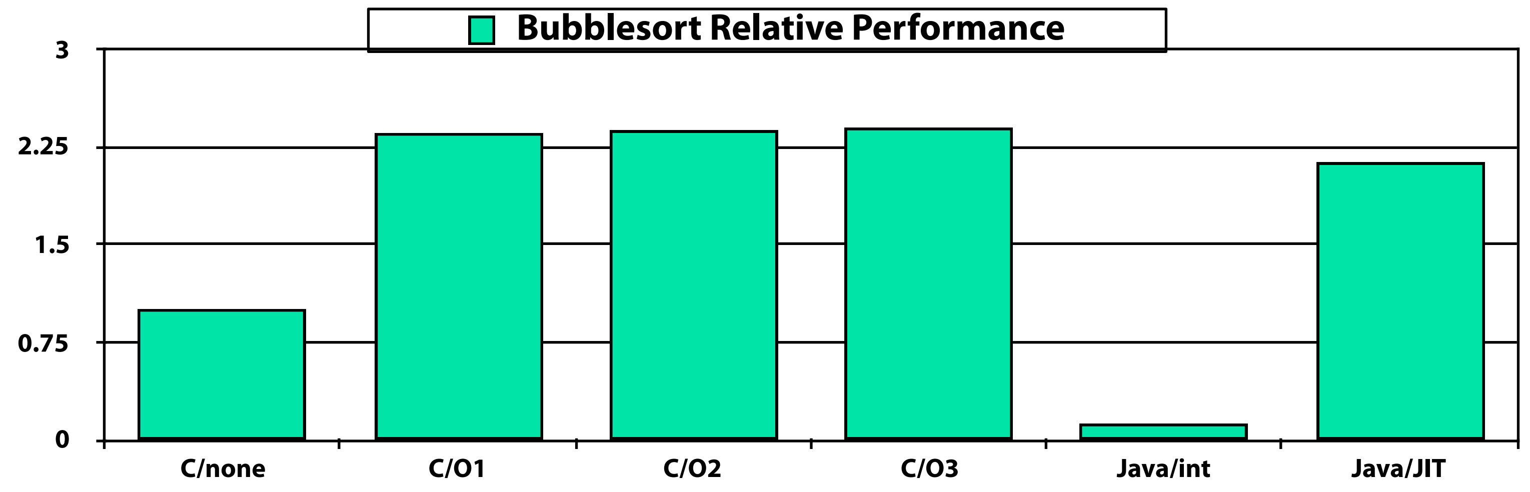


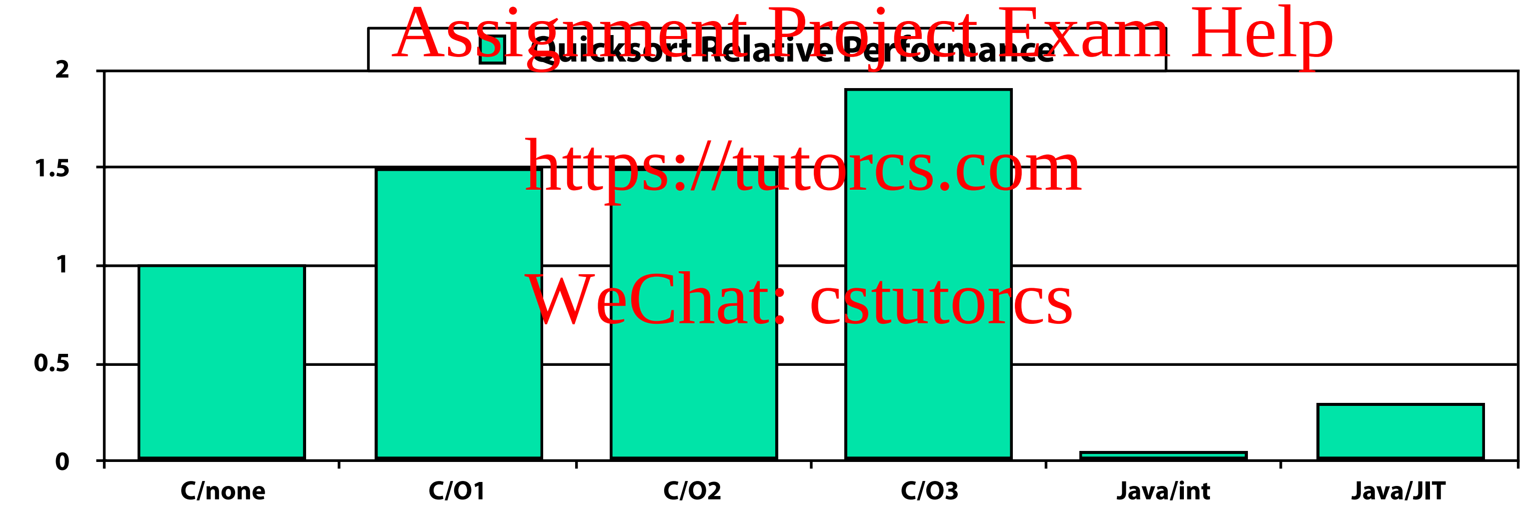


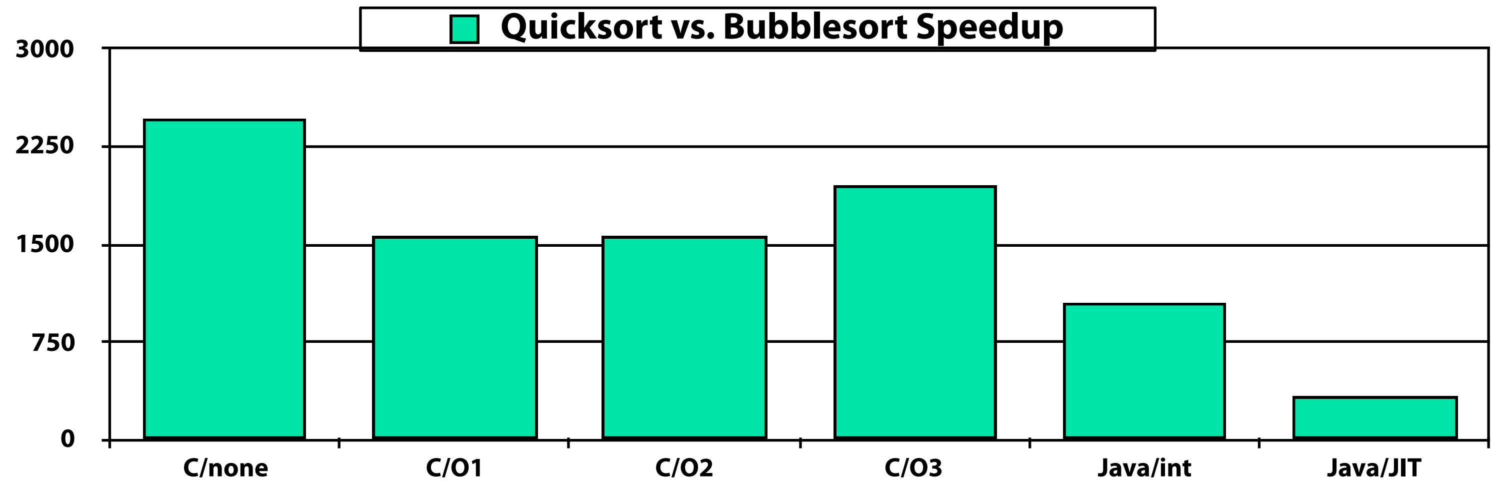




Effect of Language and Algorithm







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: 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 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)		

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

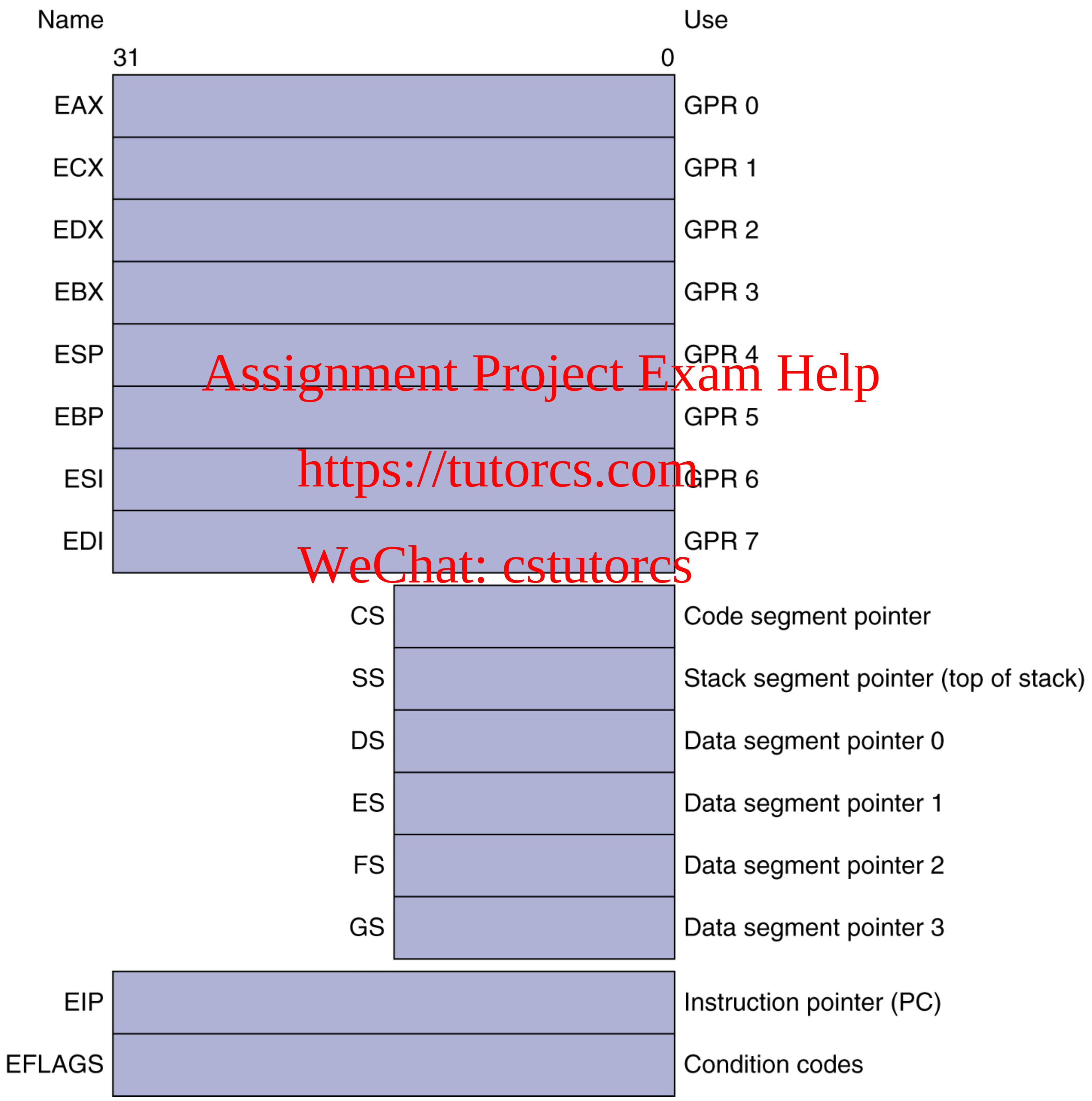
The Intel x86 ISA

- 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



Basic x86 Addressing Modes

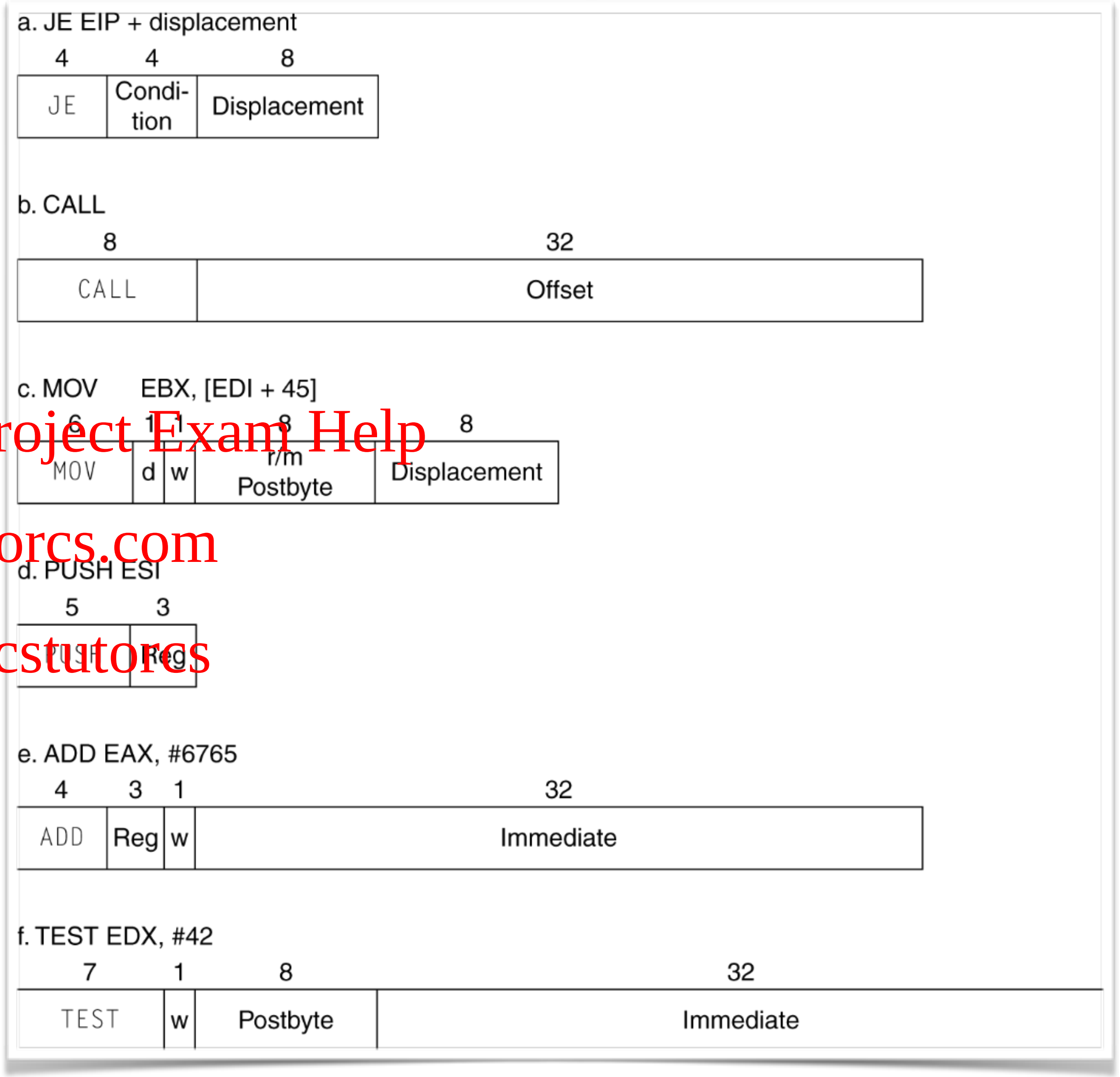
- **Two operands per instruction**

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

- 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

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

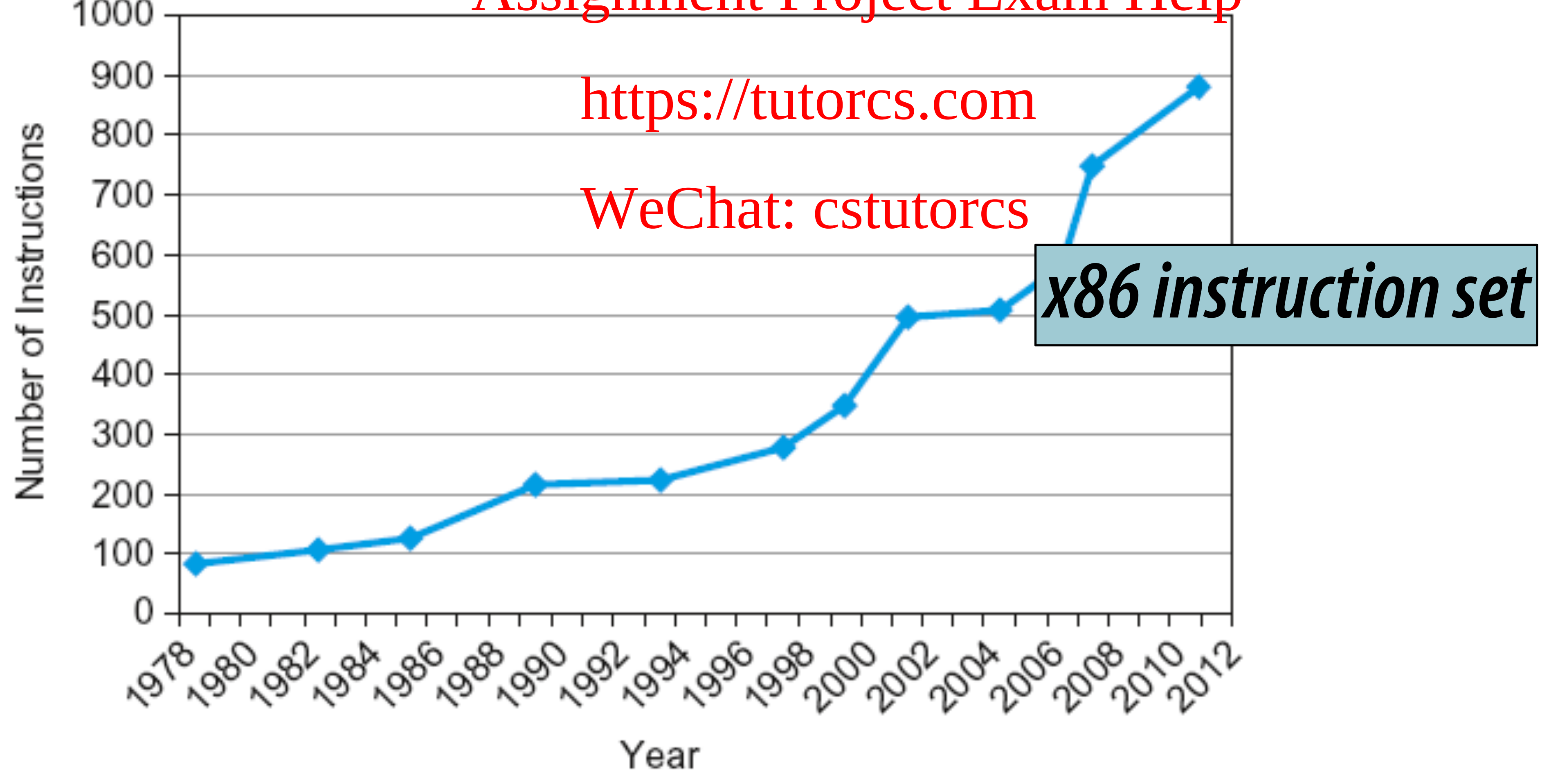
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

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 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

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

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
```

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

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)

   (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 = 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   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:
```

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)
```