Lecture 2:

Instructions: Language of the Computer (1/2)

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

Big picture for today

- *Stored program* computer: both programs and information are treated as data
  - "Information": text, pictures, videos, simulation data, etc.
- All data is stored in the "memory" of the machine
- We wish to manipulate this data:
  - Some data is instructions; we will treat that data as instructions and execute/evaluate them (but also treat them as data! e.g., linking against other code)
  - Some data is information; our instructions will operate on this information
- Today's topic: *What are these instructions?*

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

- Since the 1980s, the dominant philosophy has been toward simpler ("reduced", "RISC") instruction sets as opposed to complex ("CISC")
  - x86 is an exception, but x86 CPUs are RISC underneath

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
- `add a, b, c  // a gets b + c`
  - What are a, b, and c?
  - `add` only operates on integer data (what's an integer?)
  - Note: RISC V instructions put the destination first
- All arithmetic operations have this form
- Design Principle 1: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- **C code:**

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

- **Compiled RISC-V code:**

```
add t0, g, h    // temp t0 = g + h
add t1, i, j    // temp t1 = i + j
sub f, t0, t1   // f = t0 - t1
```

RISC-V Reference Card

- **Google it to find it**

Free & Open RISC-V Reference Card

- **It'll be attached to your exams**

Arithmetic ADD	R	`ADD`	`rd,rs1,rs2`
ADD Immediate	I	`ADDI`	`rd,rs1,imm`
SUBtract	R	`SUB`	`rd,rs1,rs2`
Load Upper Imm	U	`LUI`	`rd,imm`
Add Upper Imm to PC	U	`AUIPC`	`rd,imm`

- **What this says:**

  - **ADD is an "R" type instruction (you don't know this yet)**
  - **ADD's operands are `rd` (destination), `rs1` (source 1), `rs2` (source 2)**
  - **SUB is ~the same as ADD**

Register Operands

- Where is the data stored?
- Arithmetic instructions use *register* operands
  - Important! In RISC-V, all arithmetic instructions ONLY use register operands
- RISC-V has a 32 × 64-bit *register file*
  - Use for frequently accessed data
  - 64-bit data (in RISC-V) is called a "doubleword"
    - 32 × 64-bit general purpose registers x0 to x31
  - 32-bit data is called a "word"
- *Design Principle 2*: Smaller is faster
  - c.f. main memory: millions of locations

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- **C code:**

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

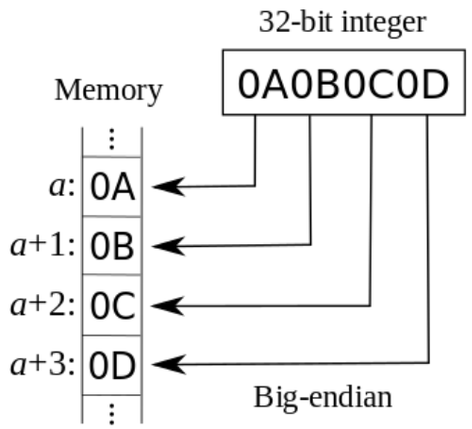
  - **f, …, j in x19, x20, …, x23**

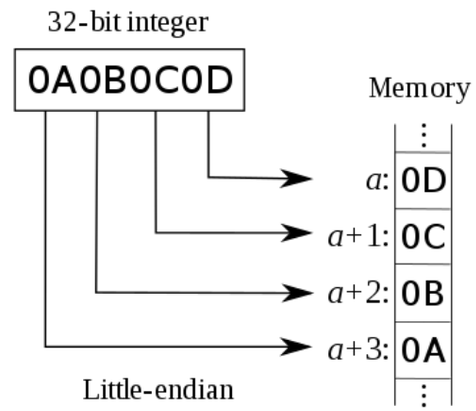
- **Compiled RISC-V code:**


```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory Operands (architectural decisions)

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations:
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word [processors]
  - c.f. Big Endian: most-significant byte at least address [network]
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs





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

Memory Operand Example

- **C code:**

```
A[12] = h + A[8];
```

  - **h in x21, base address of A in x22**

- **Compiled RISC-V code:**

  - **Index 8 requires offset of 64**

  - **8 bytes per doubleword**

```
ld   x9, 64(x22)  // x9 <- Mem[x22 + 64]
add  x9, x21, x9
sd   x9, 96(x22)  // Mem[x22 + 96] <- x9
```

- `ld dest, imm(src):`
  - `dest:` destination register
  - `imm:` immediate (integer)
  - `src:` base memory address, in register

Registers vs. Memory

- Registers are (much) faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- **Constant data specified in an instruction**

```
addi x22, x22, 4  // add the number 4 to
                  // x22, store back in x22
```


- **Make the common case fast**
  - **Small constants are common**
  - **Immediate operand avoids a load instruction**

Arithmetic ADD	R	`ADD`	`rd,rs1,rs2`
ADD Immediate	I	`ADDI`	`rd,rs1,imm`
SUBtract	R	`SUB`	`rd,rs1,rs2`
Load Upper Imm	U	`LUI`	`rd,imm`
Add Upper Imm to PC	U	`AUIPC`	`rd,imm`

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range: 0 to $+2^n - 1$

- Example
  - $0000\ 0000 \ldots 0000\ 1011_2$

    $= 0\times2^{31} + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 64 bits: 0 to +18,446,774,073,709,551,615

Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001…
  - decimal: 0 .. 2^n-1
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - RISC-V restrictions (e.g., no RISC-V subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?

Possible Representations

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- In all these, the most significant bit is the "sign bit"
- Issues: balance, number of zeros, ease of operations

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range: -2^{n-1} to $+2^{n-1}-1$

- Example

  - $1111\ 1111\ \ldots\ 1111\ 1100_2$

    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- Using 64 bits: –9,223,372,036,854,775,808

  to 9,223,372,036,854,775,807

2s-Complement Signed Integers

- Bit 63 (the *most significant* bit, MSB) is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
  - Complement means "flip all the bits"
  - $x + \bar{x} = 1111\ldots111_2 = -1$
  - $\bar{x} + 1 = -x$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_{two}$
  - $-2 = 1111\ 1111\ \ldots\ 1101_{two} + 1$
    $= 1111\ 1111\ \ldots\ 1110_{two}$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

- In RISC-V instruction set
  - `lb`: sign-extend loaded byte
  - `lbu`: zero-extend loaded byte

Loads	Load Byte	I	LB	rd,rs1,imm			
	Load Halfword	I	LH	rd,rs1,imm			
	Load Word	I	LW	rd,rs1,imm	L{D|Q}	rd,rs1,imm	
	Load Byte Unsigned	I	LBU	rd,rs1,imm			
	Load Half Unsigned	I	LHU	rd,rs1,imm	L{W|D}U	rd,rs1,imm	

Representing Instructions

- Instructions are encoded in binary
  - Called "machine code"
  - How do we get from `add x5, x20, x21` to binary?
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Big picture: We divide the 32-bit instruction word into "fields", each of a few bits, and encode different pieces information from the instruction into each field
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits ("nibble") per hex digit
  - 0x means "I'm hexadecimal"

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0x eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions

- **Instruction fields**
  - ***opcode*: operation code**
  - ***rd*: destination register number**
  - ***funct3*: 3-bit function code (additional opcode)**
  - ***rs1*: the first source register number**
  - ***rs2*: the second source register number**
  - ***funct7*: 7-bit function code (additional opcode)**

R-format Example

```
add x9,x20,x21
```

funct7	*rs2*	*rs1*	*funct3*	*rd*	*opcode*
7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*

0	*21*	*20*	*0*	*9*	*51*

0000000	*10101*	*10100*	*000*	*01001*	*0110011*

$$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} =$$

$$015A04B3_{16}$$

Opcode Map

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL

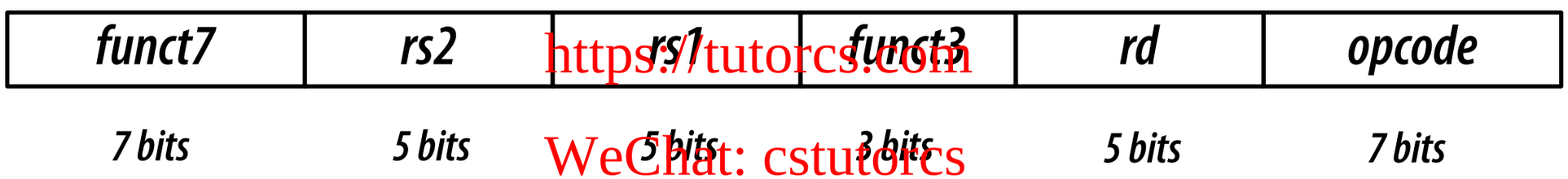
RISC-V I-format Instructions

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
  - How big can this immediate be?
  - Why did they pick this size?
  - Advantages/disadvantages of making it bigger/smaller?

RISC-V I-format vs. R-format

- **I-format:**

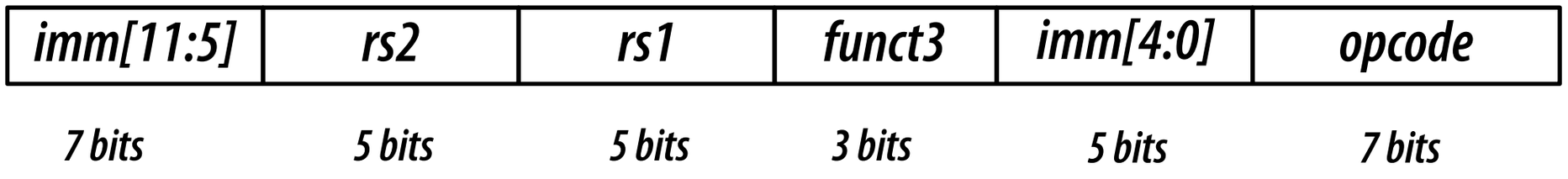
- **R-format:**



- ***Design Principle 3*: Good design demands good compromises**
  - **Different formats complicate decoding, but allow 32-bit instructions uniformly**
  - **Keep formats as similar as possible**

RISC-V S-format Instructions

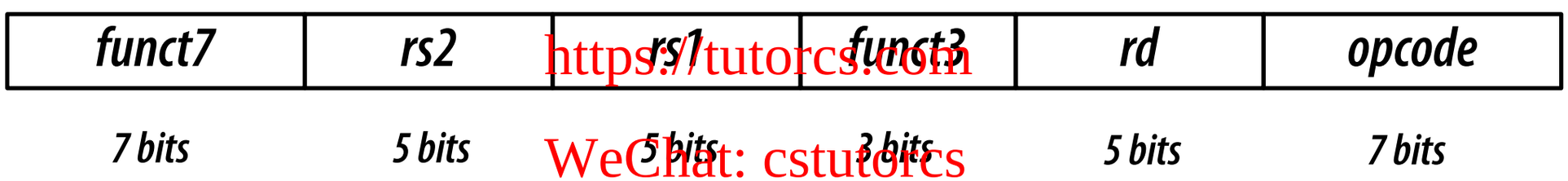
- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place



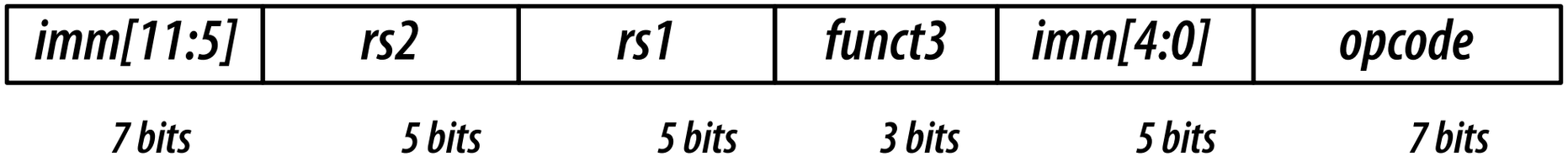
RISC-V I-format vs. R-format vs. S-format

- **I-format:**

- **R-format:**

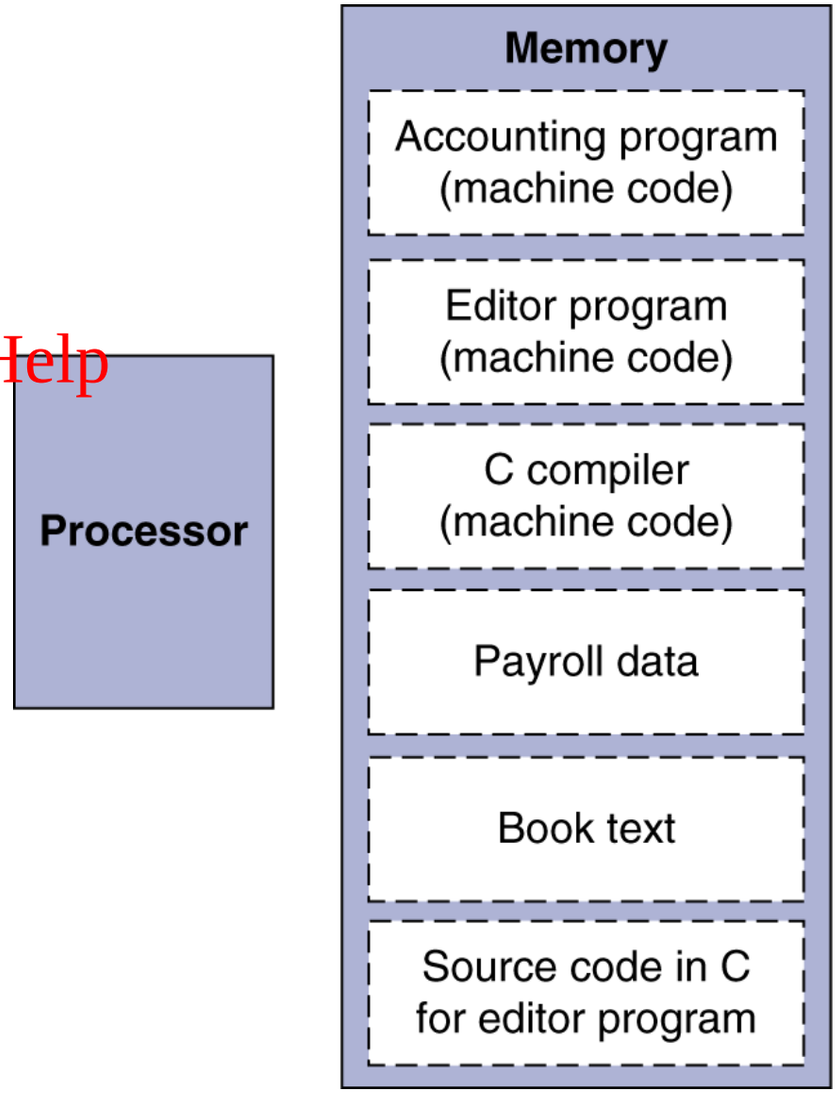


- **S-format:**



Stored Program Computers

- 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



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	~	~	

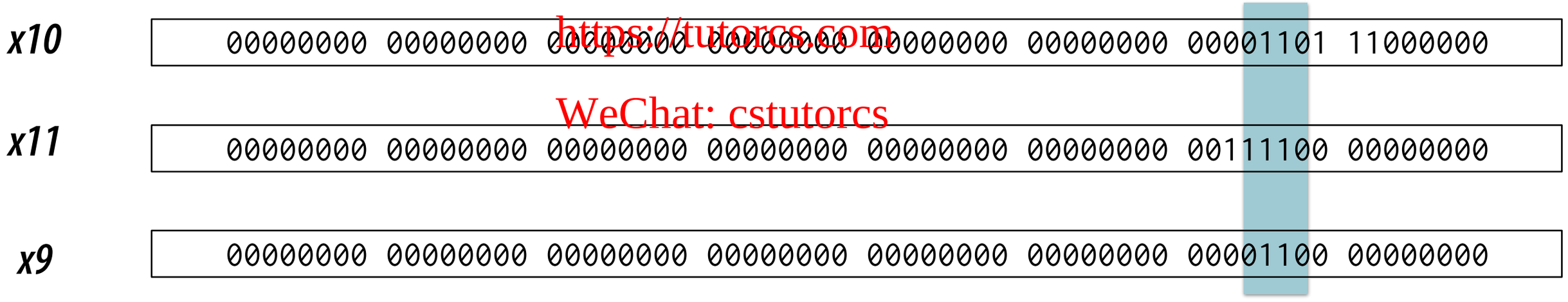
- ***Useful for extracting and inserting groups of bits in a word***

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?

AND Operations

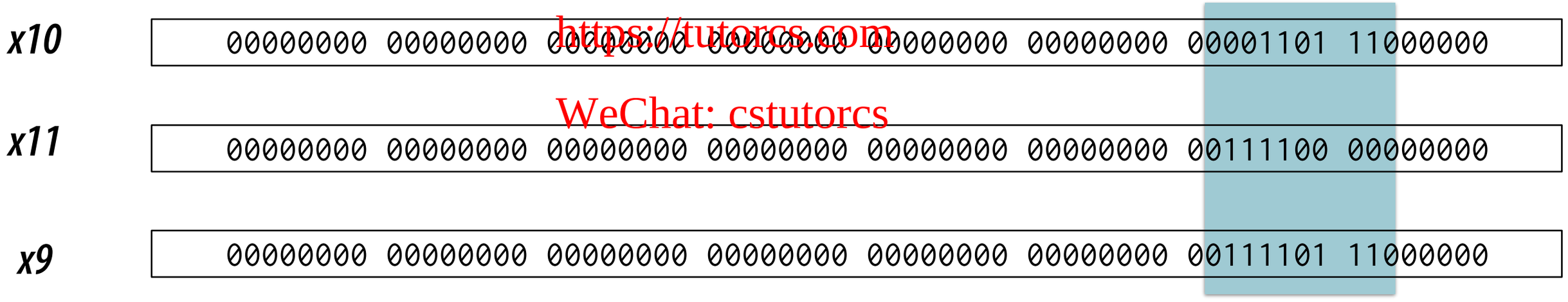
- Useful to mask bits in a word
  - Select some bits, clear others to 0
- `and x9,x10,x11`



OR Operations

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

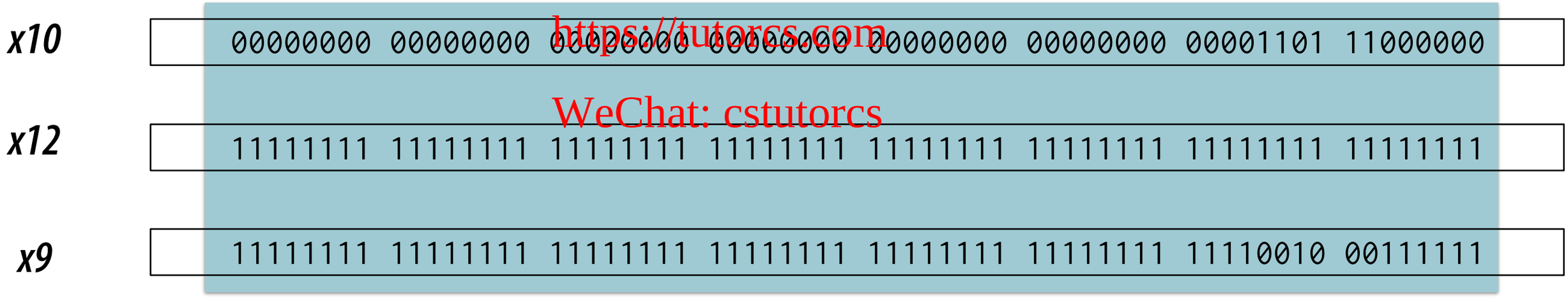
```
or x9,x10,x11
```

XOR Operations

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

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

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

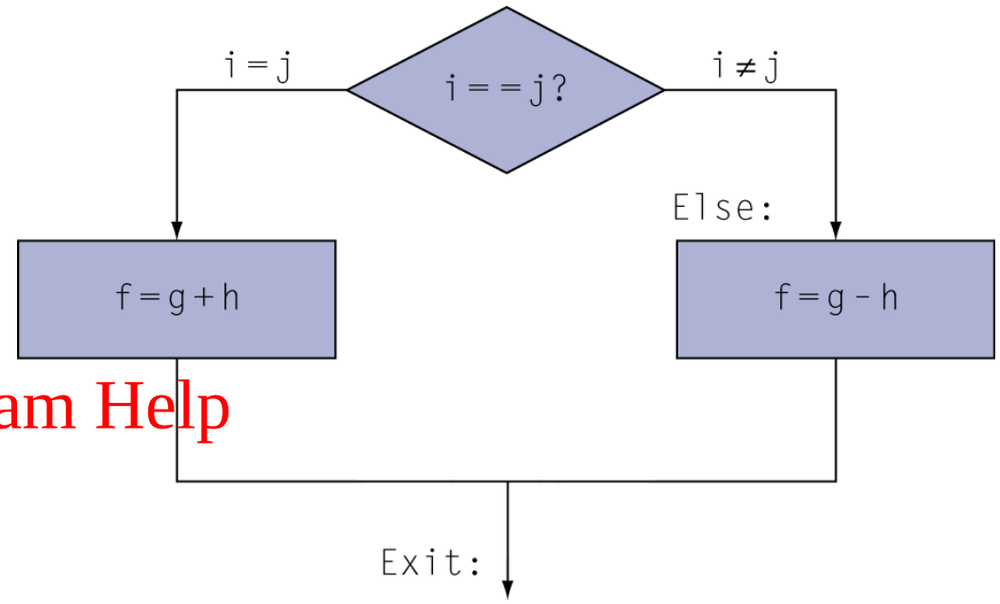
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

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

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

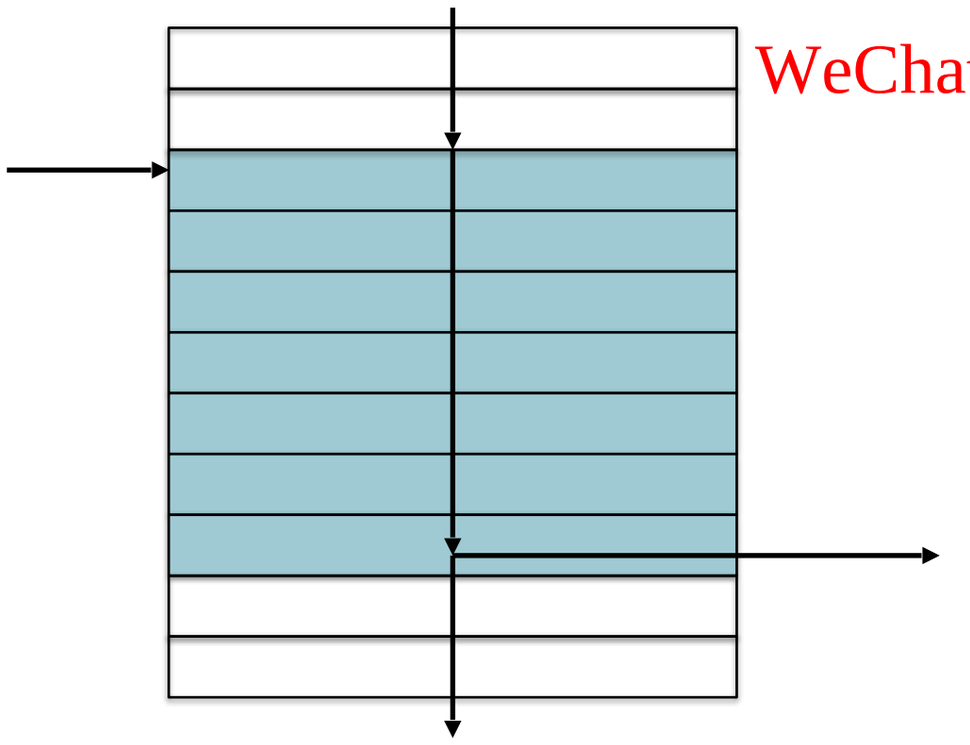
- RISC-V can do:
  - register
  - reg+off
  - (small) absolute

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]

https://cs.nyu.edu/courses/fall10/V22.0201-002/addressing_modes.pdf

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**

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 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

x5 = g + h

x6 = i + j

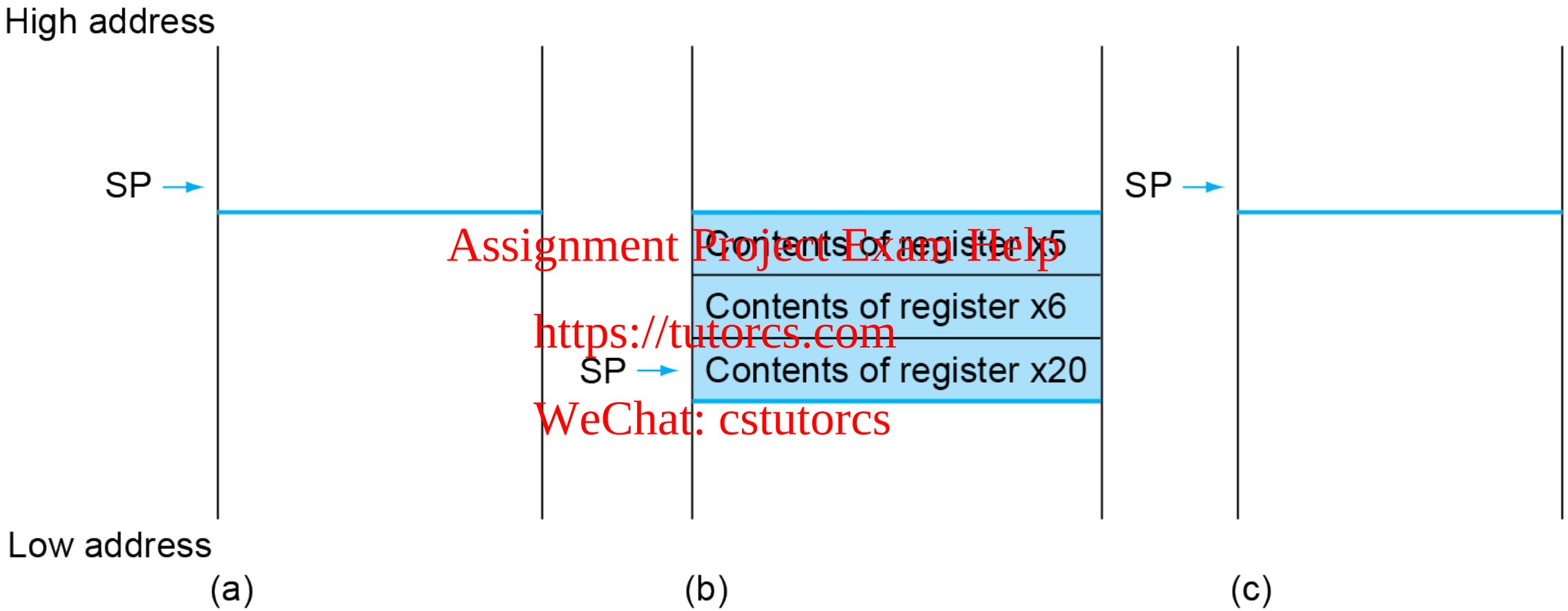
f = x5 − x6

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

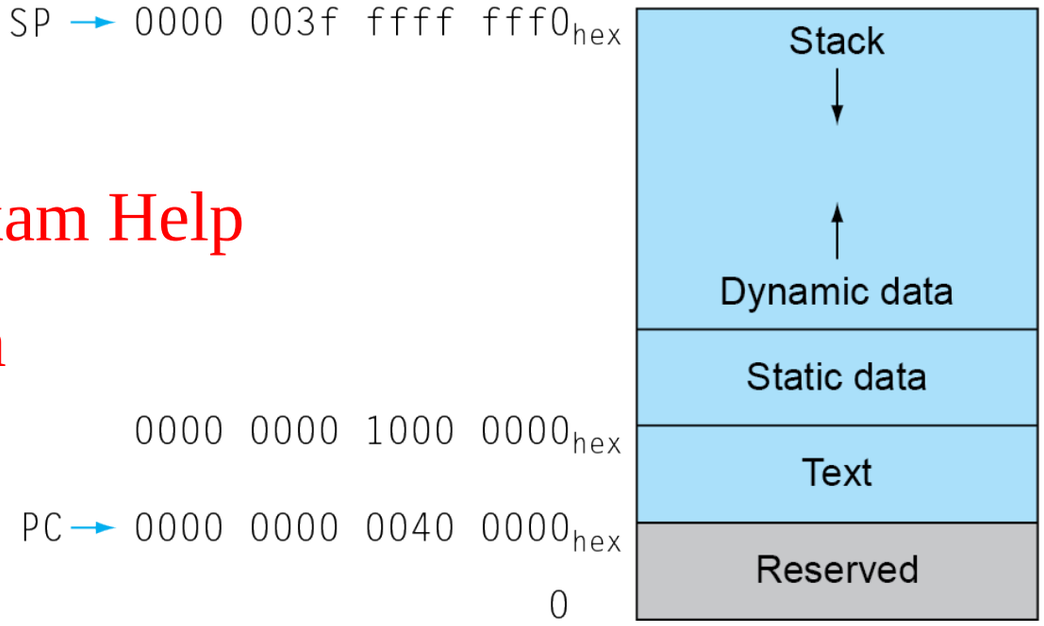
```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- **RISC-V code:**

```
fact:
     addi sp,sp,-16      | Save return address and n on stack
     sd   x1,8(sp)       |
     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