Lecture 3a:

Instructions: Language of the Computer (2/3)

John Owens

Introduction to Computer Architecture

UC Davis EEC 170, Winter 2021

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
- Numbers
  - Integers, signed/unsigned integers, sign extension
  - Decimal, binary, hexadecimal
  - Converting bits <-> numbers

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

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

- **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)**

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

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

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

- **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?**

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
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D|Q}	rs1,rs2,imm

immediate	*rs1*	*funct3*	*rd*	*opcode*
12 bits	*5 bits*	*3 bits*	*5 bits*	*7 bits*

RISC-V I-format vs. R-format

- **I-format:**

immediate	*rs1*	*funct3*	*rd*	*opcode*
12 bits	*5 bits*	*3 bits*	*5 bits*	*7 bits*

- **R-format:**

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

- ***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**

imm[11:5]	*rs2*	*rs1*	*funct3*	*imm[4:0]*	*opcode*
7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*

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
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D|Q}	rs1,rs2,imm

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

- **I-format:**

immediate	*rs1*	*funct3*	*rd*	*opcode*
12 bits	*5 bits*	*3 bits*	*5 bits*	*7 bits*

- **R-format:**

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

- **S-format:**

imm[11:5]	*rs2*	*rs1*	*funct3*	*imm[4:0]*	*opcode*
7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*

Logical Operations

- **Instructions for bitwise manipulation**

<table>
<tr><th>Operation</th><th>C</th><th>Java</th><th>RISC-V</th></tr>
<tr><td>Shift left</td><td><<</td><td><<</td><td><code>slli</code></td></tr>
<tr><td>Shift right</td><td>>></td><td>>>></td><td><code>srli</code></td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td><code>and, andi</code></td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td><code>or, ori</code></td></tr>
<tr><td>Bit-by-bit XOR</td><td>^</td><td>^</td><td rowspan="2"><code>xor, xori</code></td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td></tr>
</table>

- ***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?

Shifts	Shift Left	R	SLL	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt

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

Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm

Up to this point, we've made an assumption: what happens after we run instruction *n*?

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

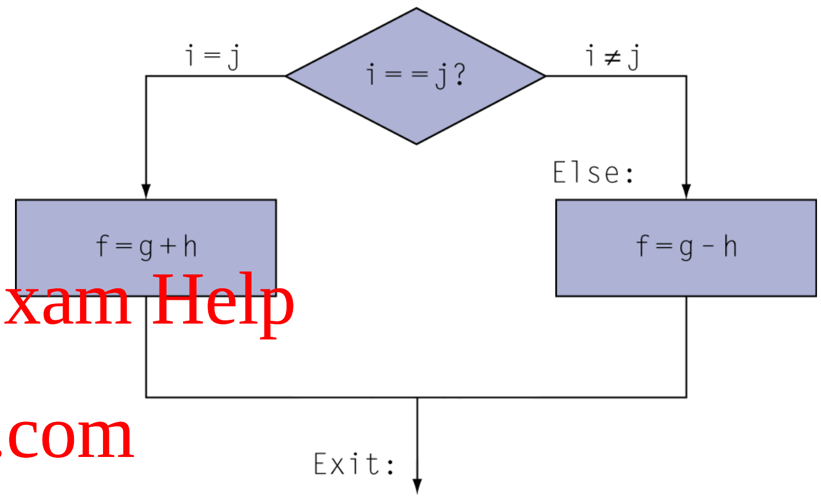
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

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

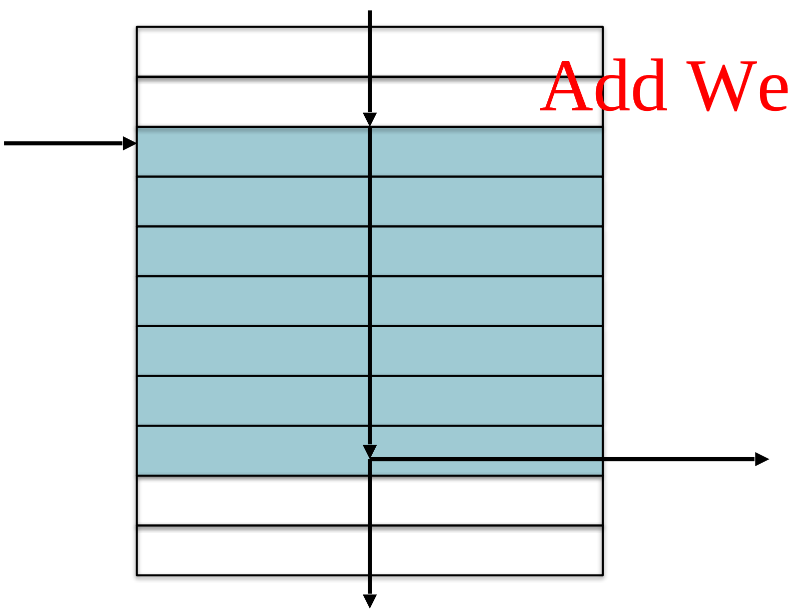
Mode	Intel
Absolute	MOV EAX, [0100]
Register	MOV EAX, [ESI]
Reg + Off	MOV EAX, [EBP-8]
R*W + Off	MOV EAX, [EBX*4 + 0100]
B + R*W + O	MOV EAX, [EDX + EBX*4 + 8]

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

Let's say you write an awesome procedure in RISC-V and I want to call it. You use registers, I use registers. What could go wrong?

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.

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;
}
```


  - 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)

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

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

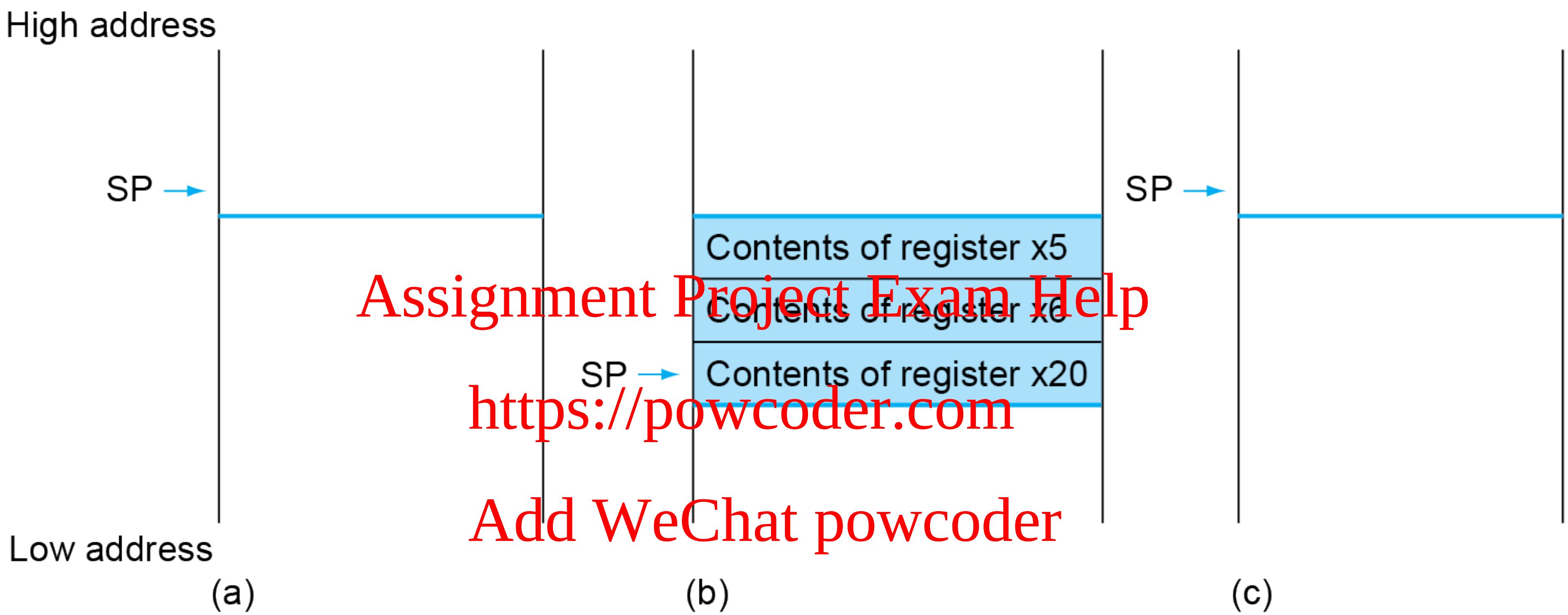
copy f to return register

Restore x5, x6, x20 from stack

Return to caller

What could a compiler do to optimize the previous code?

Local Data on the Stack



- **In RISC-V:**
  - **The stack pointer points to the "top" of the stack (the most recently used item)**
  - **The stack grows downward**

Register Usage (RISC-V 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

- Big picture: When a procedure call is made, some tasks are the responsibility of the caller and some are the responsibility of the callee

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**

Non-Leaf Procedure Example

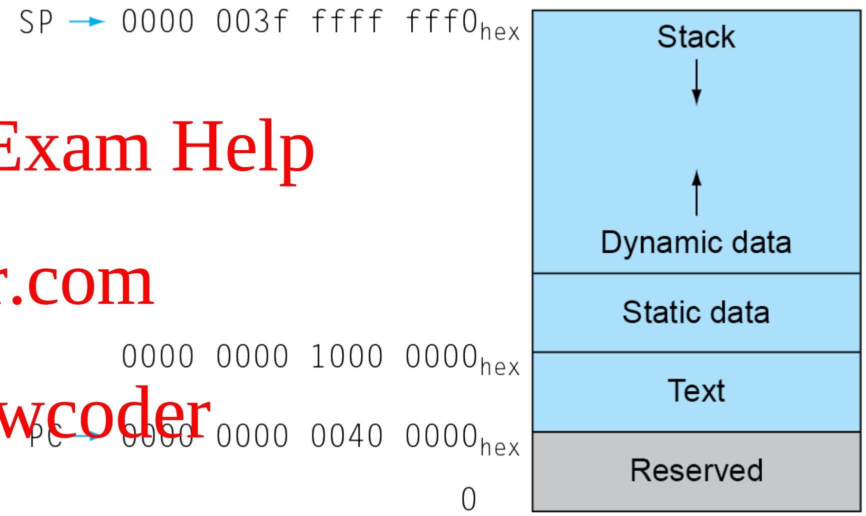
```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- **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
```

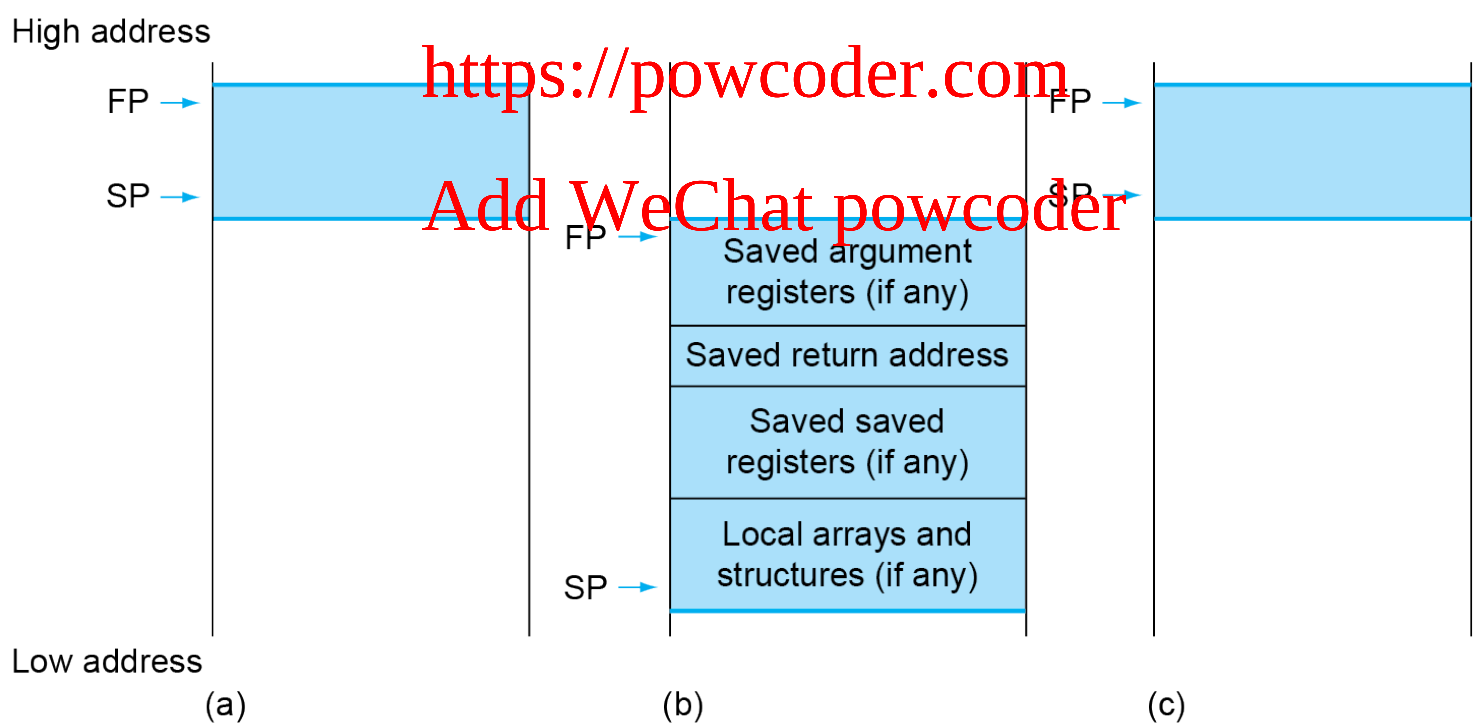
Memory Layout

- 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



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 (x19 contains i)
L1: add x5,x19,x10 // x10 = &y; x5 = addr of y[i]
 lbu  x6,0(x5)    // x6 = y[i]
 add  x7,x19,x11  // x11 = &x; 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
```