Chapter 2

Instructions: Language of the Computer

Martin Schoeberl

Admin

- Please ask when something is unclear
- Slow me down if I am too fast
- Suggested reading and exercises at homepage
  - Will be good for preparation of the exam
- RISC-V assembly lab
  - https://www.kvakil.me/venus/
  - Show example

About C

- Old language
  - Coinvented with Unix
- Very close to the hardware
- System software (operating system) still written in C
- You need to know basic C

Learn C

- https://en.wikipedia.org/wiki/The_C_Programming_Language
- http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/

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

Instruction Set Architecture

- One instruction set
- Different possible implementations
  - Invented by IBM to sell different machines
  - Intel vs. AMD
- One compiler for different machines
- Common term now
  - ISA

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, …

RISC-V Instruction Set

- A free instruction set
  - Anyone is allowed to implement a RISC-V processor
  - Not the case for ARM, MIPS, x86
- Several implementations available
  - Open source and commercial
- Avoid paying royalty to e.g., ARM
- Vary active community
- We switched from MIPS to RISC-V

RISC-V Instruction Set

- RISC-V is a very simple ISA
- See the *green* sheet:
  - https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf
- You will implement a simulator for RISC-V
  - 39 different instructions
  - Many are very similar

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c   // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours 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
```

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Register x0 – x31
  - 32-bit data is called a “word”
  - 64-bit data is called a “doubleword”
- RISC-V in 32-bit and 64-bit variants
  - Not a big difference in the instructions
  - Book uses 32-bit, lab/Venus uses 32-bit

Register File

- All (but one) register are identical
- Register number 0 (x0) is hardwired to 0
  - Cannot be changed
  - Allows some optimization in the code

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

RISC-V Registers Convention

- 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 (= saved register)
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Alternative Names

- Assembler understands alternative names for registers, such as:
  - zero, ra, sp, gp, tp
  - t0-t7
  - fp (= s0)
  - a0-a7
  - s0-s11
- To simplify remembering usage convention
  - This is not hardware, but convention

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

- 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
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
  - But aligning words is more efficient

Memory Operand Example

- C code:

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

  - h in x21, base address of A in x22
    - Base address is where the array starts in memory
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
lw      x9, 32(x22)
add     x9, x21, x9
sw      x9, 48(x22)
```

Registers vs. Memory

- Registers are 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
```

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

Binary Numbers

- We are used to base 10 (10 fingers)
- Computers “have only two fingers”
  - Digital electronics **just** 0 or 1
- Use binary number system
  - Base 2

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
  - $0000\ 0000\ \ldots\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 64 bits: 0 to +18,446,774,073,709,551,615

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
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for positive numbers
- $-(-2^{n-1})$ can't be represented
- Positive 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

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 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

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

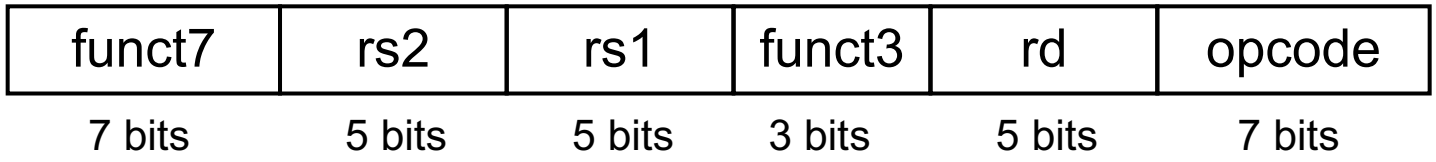
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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

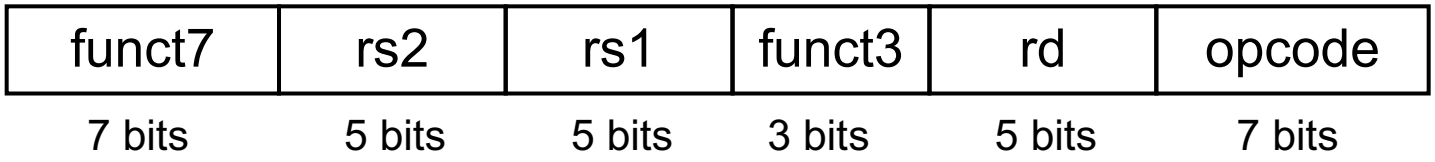
- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

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

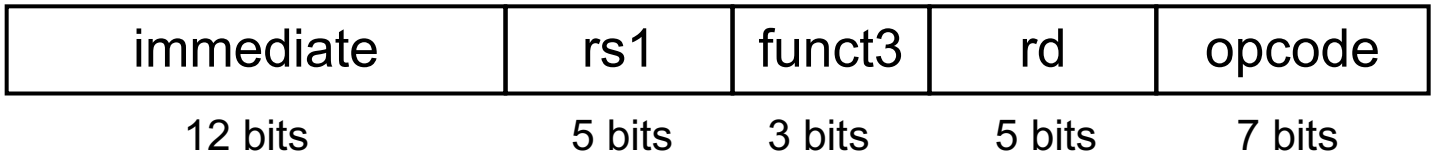



```
add x9,x20,x21
```

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

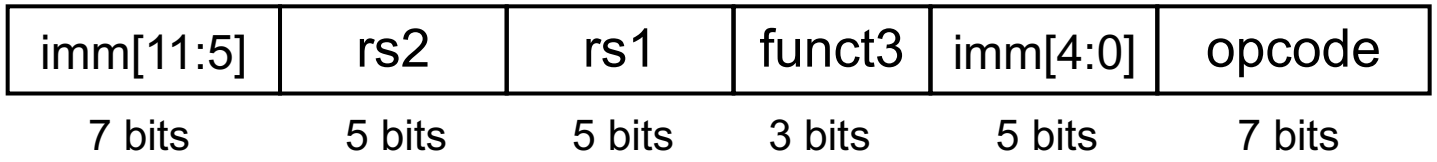
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two}$ =
$015A04B3_{16}$

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

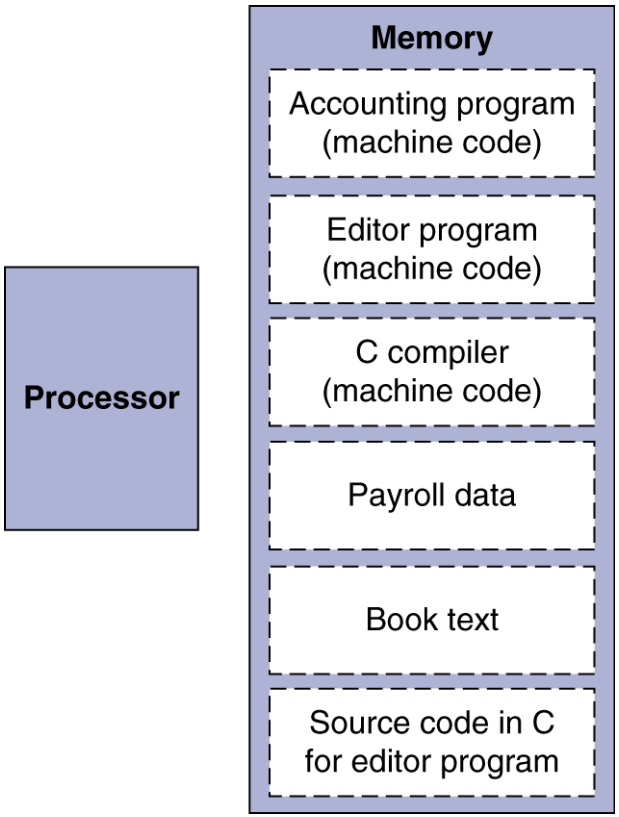
RISC-V S-format Instructions



- 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

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

Memory

- Organized in 8-bit bytes
- You can view it as an array of bytes
  - Like: byte[] memory = new byte[100000]
- May be accessed in different *sizes*
  - Byte (8-bit)
  - Halfword (16-bit)
  - Word (32-bit)
  - (Double word (64-bit))
- Most languages have those integers as data types (not Python, not JavaScript)

Only One Memory

- Also called Van Neuman architecture
  - Standard today
- Harvard architecture was two memories
  - One for instruction, one for data
  - How do you load a program?
- Processor explanation shows two memories
  - In reality these are two caches
- Have one memory in your simulator

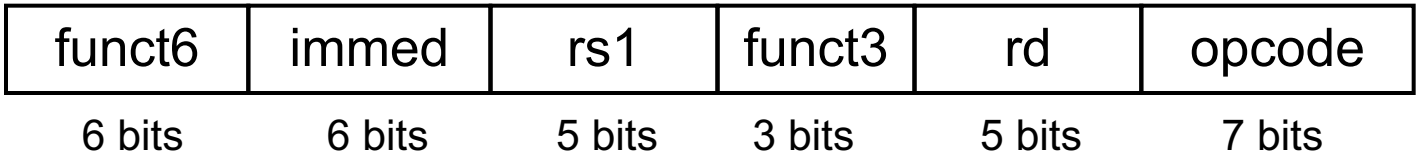
Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`sll, slli`
Shift right	>>	>>>	`srl, 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

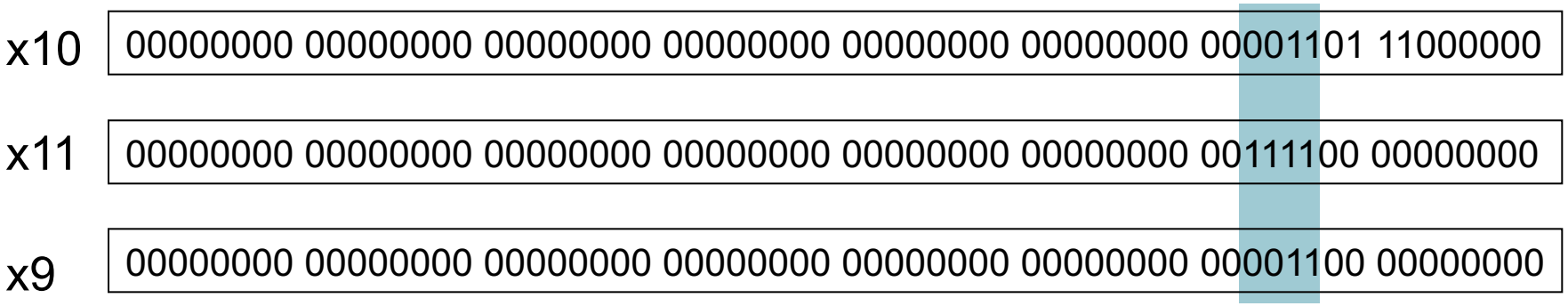


- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

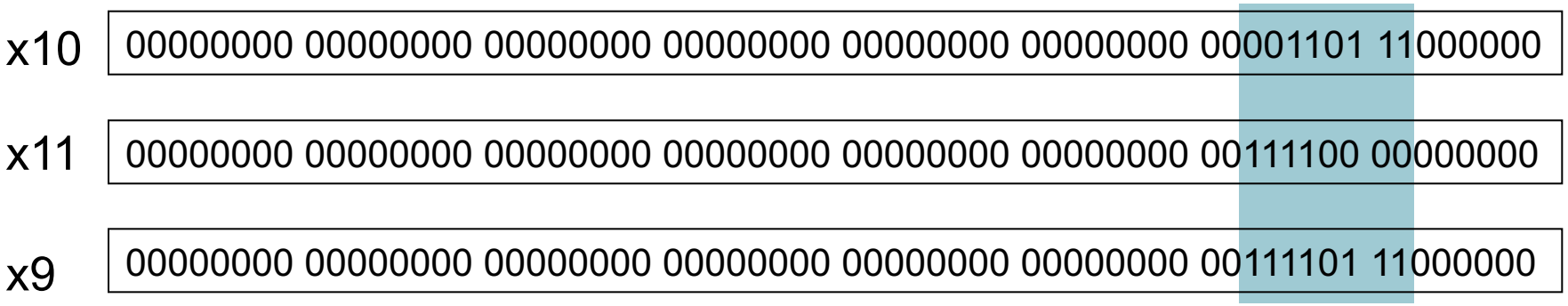
```
and x9,x10,x11
```



OR Operations

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

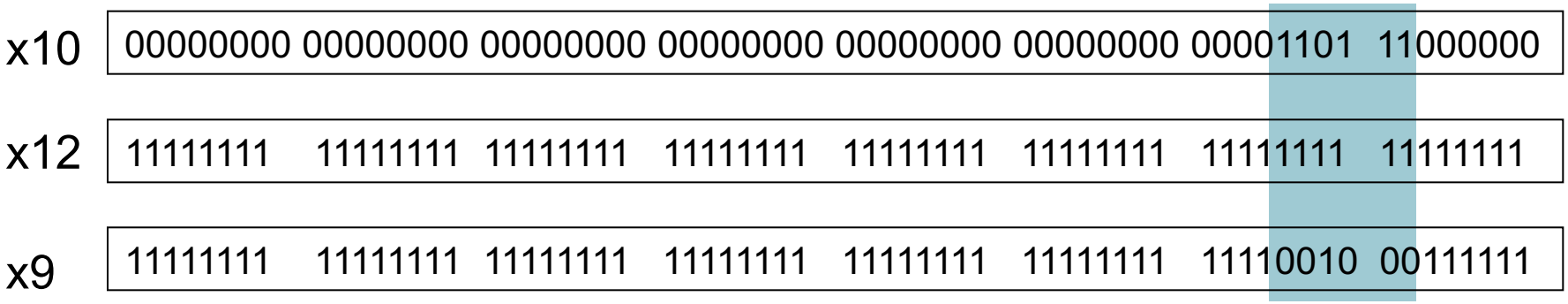
```
or x9,x10,x11
```



XOR Operations

- Differencing operation
  - Invert some bits, 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

Program Counter

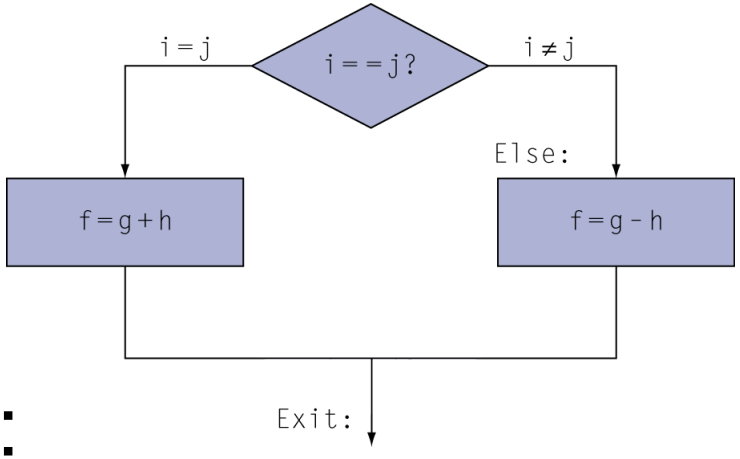
- A register that points to the current instruction
- Usually called PC
- Incremented by 4 to point to the next instruction
- Can be changed by branch and jump instructions
  - Not very *visible* in the instruction set

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:

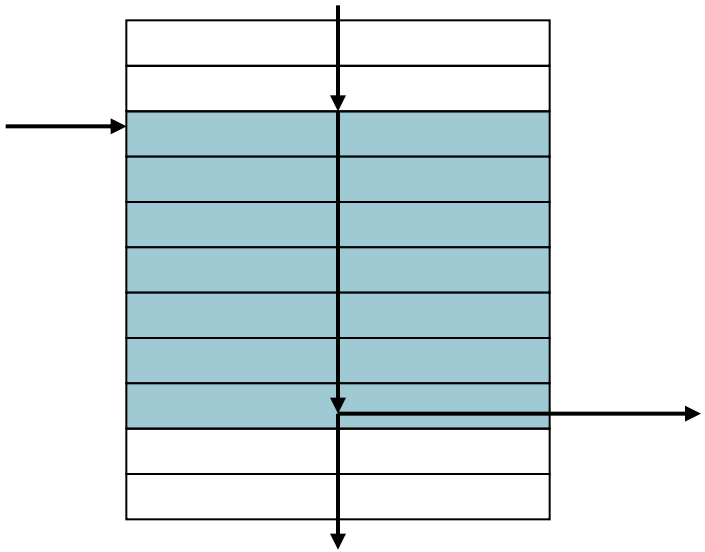
```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25
- Compiled RISC-V code:

```
Loop: slli x10, x22, 2
      add  x10, x10, x25
      lw   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

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
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. 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

- C code:

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

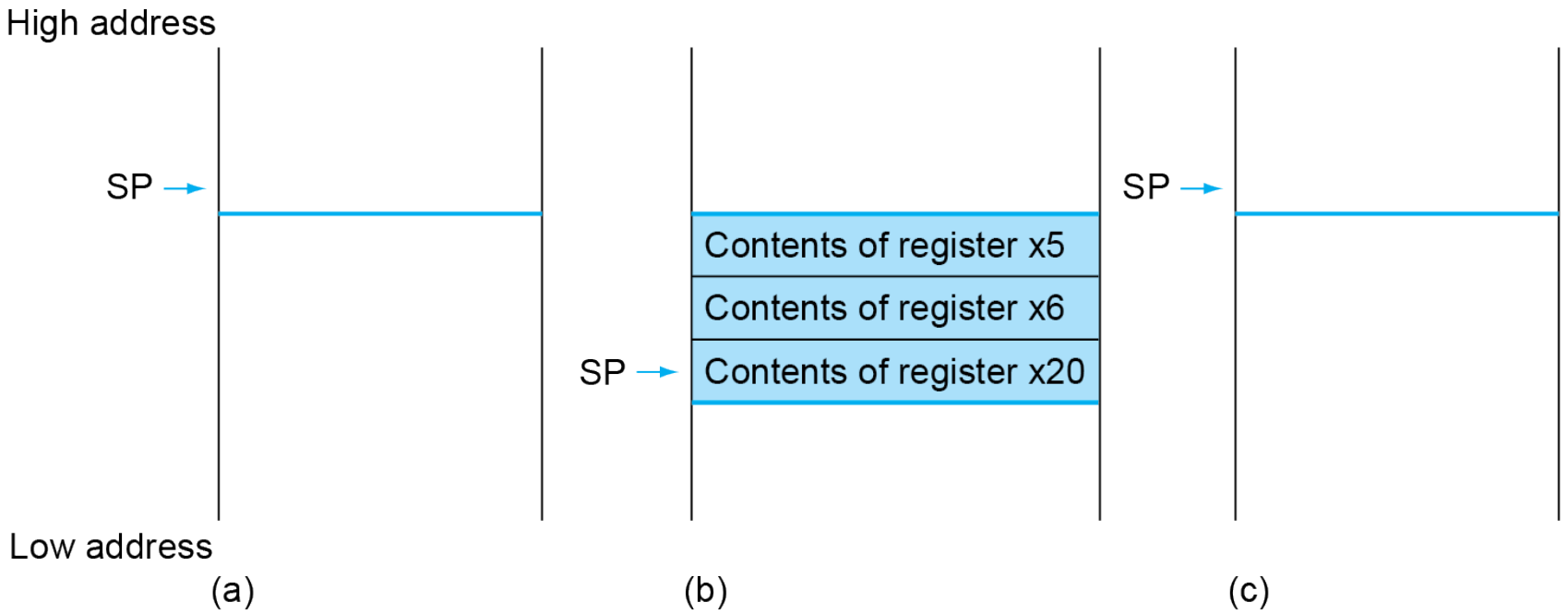
  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

Leaf Procedure Example

- RISC-V code:

```
leaf_example:
   addi sp,sp,-12        Save x5, x6, x20 on stack
   sw   x5,16(sp)
   sw   x6,8(sp)
   sw   x20,0(sp)
   add  x5,x10,x11       x5 = g + h
   add  x6,x12,x1        x6 = i + j
   sub  x20,x5,x6        f = x5 – x6
   addi x10,x20,0        copy f to return register
   lw   x20,0(sp)        Resore x5, x6, x20 from stack
   lw   x6,8(sp)
   lw   x5,16(sp)
   addi sp,sp,12
   jalr x0,0(x1)         Return to caller
```

Local Data on the Stack



Register Usage

- 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:

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

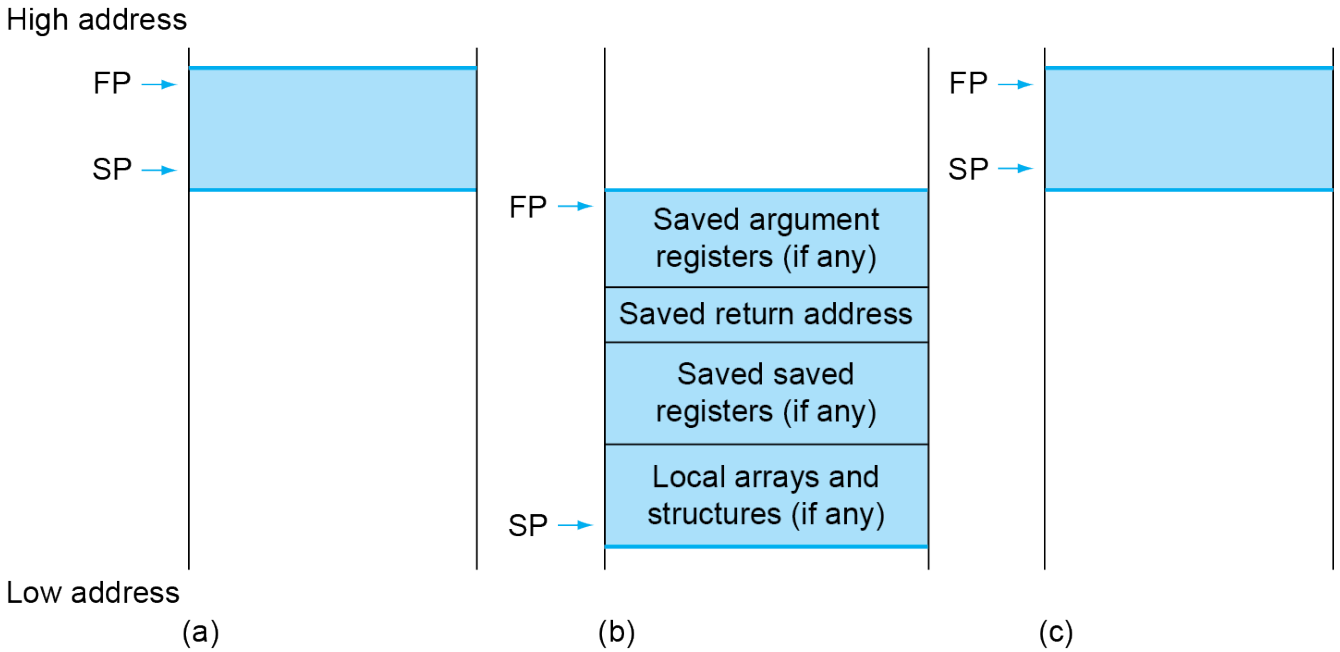
  - Argument n in x10
  - Result in x10

Non-Leaf Procedure Example

- RISC-V code:

```
fact:
    addi sp,sp,-8       Save return address and n on stack
    sw   x1,8(sp)
    sw   x10,0(sp)
    addi x5,x10,-1      x5 = n - 1
    bge  x5,x0,L1       if (n – 1) >= 0, 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)
    addi x6,x10,0       move result of fact(n - 1) to x6
    lw   x10,0(sp)      Restore caller's n
    lw   x1,8(sp)       Restore caller's return address
    addi sp,sp,8        Pop stack
    mul  x10,x10,x6     return n * fact(n-1)
    jalr x0,0(x1)       return
```

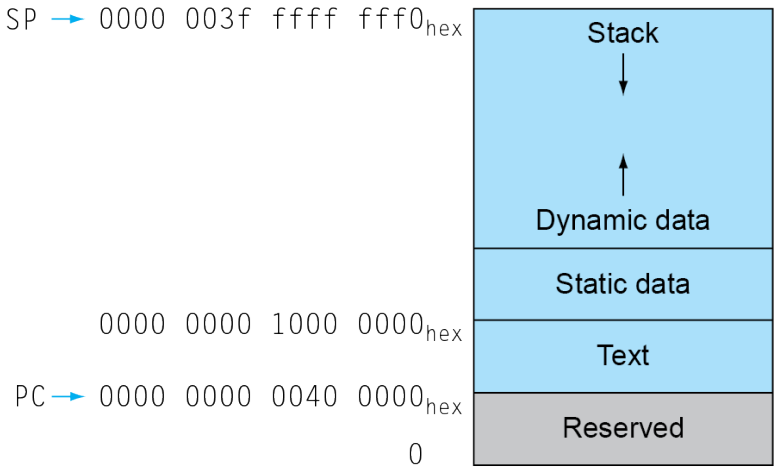
Local Data on the Stack



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

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



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)`

Some Words on the Lab

- We use the Venus RISC-V simulator
  - Runs in the browser
  - You can save it to run offline
- Implements the 32-bit version of RISC-V
  - No ld or sd, use lw, sw
- In a future lab we will use the full toolchain
  - Gcc, assembler, simulator
  - Runs best in Linux
  - Will provide a Ubuntu VM with installed tools

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