# Chapter 2

## Instructions: Language of the Computer

指令：计算机的语言

# 目录

- 机器语言——指令集
  - 操作码、操作数（寄存器、内存地址、小常量）
- 数据的二进制表示
  - 无符号数、有符号数（二进制补码）
- 指令的二进制表示
  - 算术逻辑、访存、控制、函数调用
- 其他表示
  - 字符串、常量、数组与指针
- 并行与同步指令
- 程序的编译与运行

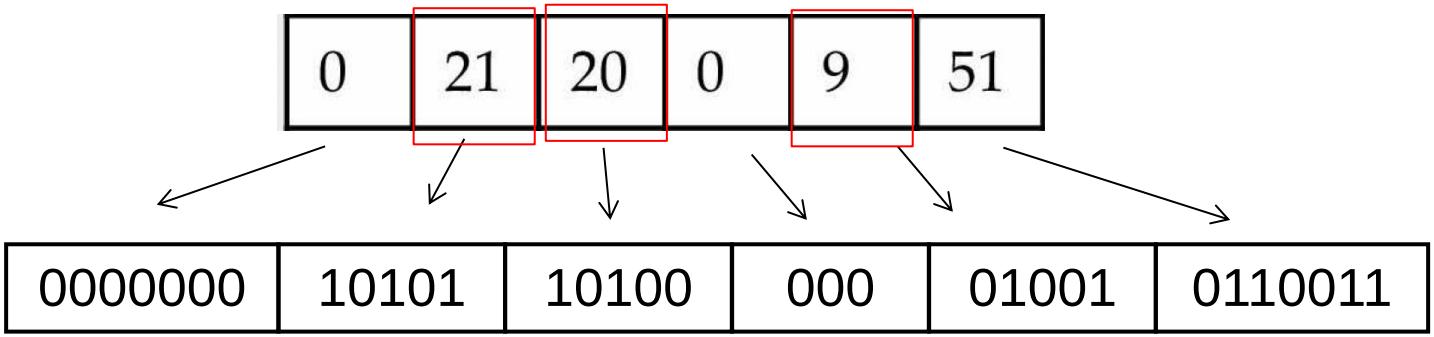
# 指令的表示（或编码）

- **指令**跟**数据**一样, 也是用二进制编码的
  - 也称**机器码**

- RISC-V指令
  - 编码成 32-bit 长的指令字
  - 用尽量简单的格式来编码
    - 操作码：operation code (opcode)
    - 寄存器编号：register numbers
    - 立即数：immediates…
    - 内存呢...?
  - Regularity! 规整!

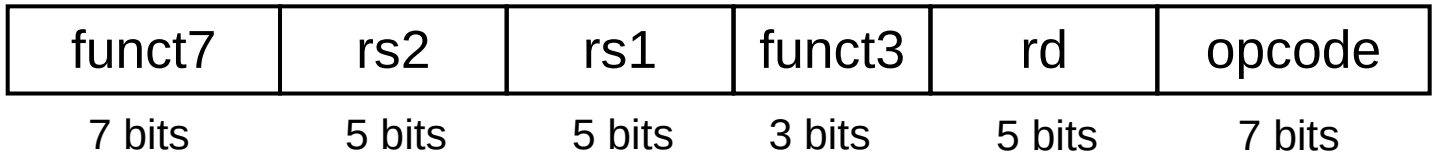
# 指令编码的例子

- Example
  - 汇编指令：add x9, x20, x21
  - 对应的机器语言或者机器码，如下




  - 有什么规律?
    - 操作码、寄存器?

# R型指令格式——寄存器型




- Instruction fields
  - opcode: operation code 操作码
  - **rd**: destination register number 目的寄存器
  - funct3: 3-bit function code (additional opcode) 其他操作码
  - **rs1**: the first source register number 源寄存器1
  - **rs2**: the second source register number 源寄存器2
  - funct7: 7-bit function code (additional opcode) 其他操作码
  - 例子
    - add rd, rs1, rs2
    - add x9, x20, x21

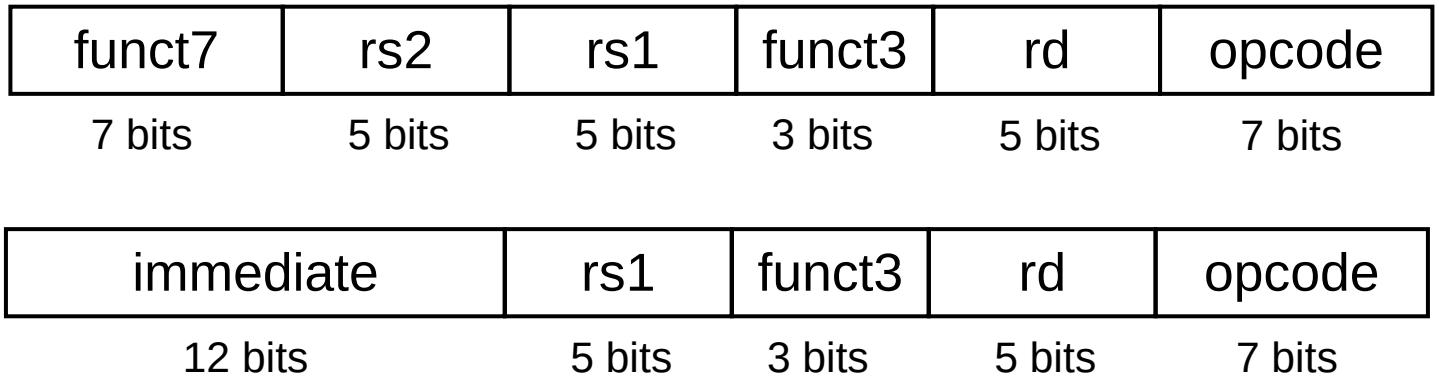
# R型指令格式的例子

- 指令格式
  - add rd, rs1, rs2
  - 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 |

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

# I型指令格式

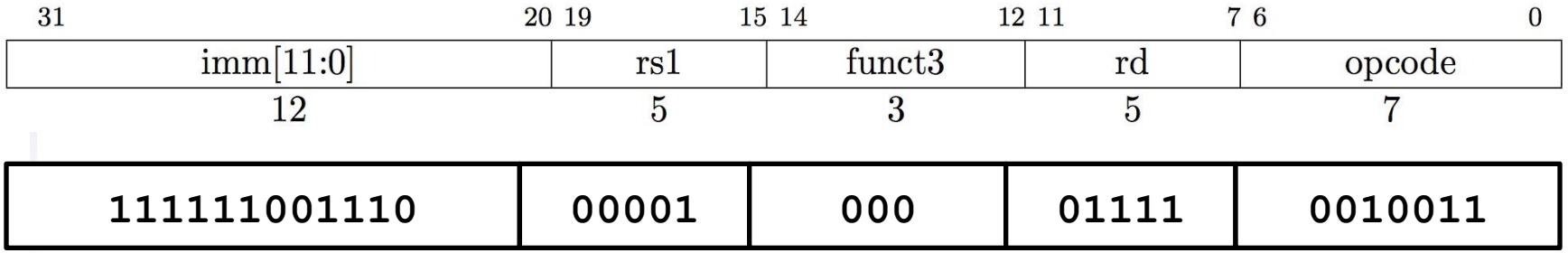



- 用于立即数算术指令和load 指令
  - rs1: 源寄存器，或者 基址寄存器
  - immediate: 常量操作数,或者 用于加在基址寄存器的偏移
    - 2进制补码表示，有符号扩展
- Design Principle 3: 好的设计需要妥协
  - Different formats complicate decoding, but allow 32-bit instructions uniformly 为了保持32位，有时候需要不一致
  - Keep formats as similar as possible 尽量一致
  - 例子: **`addi  x15,x1,-50`**

# I型指令格式的例子

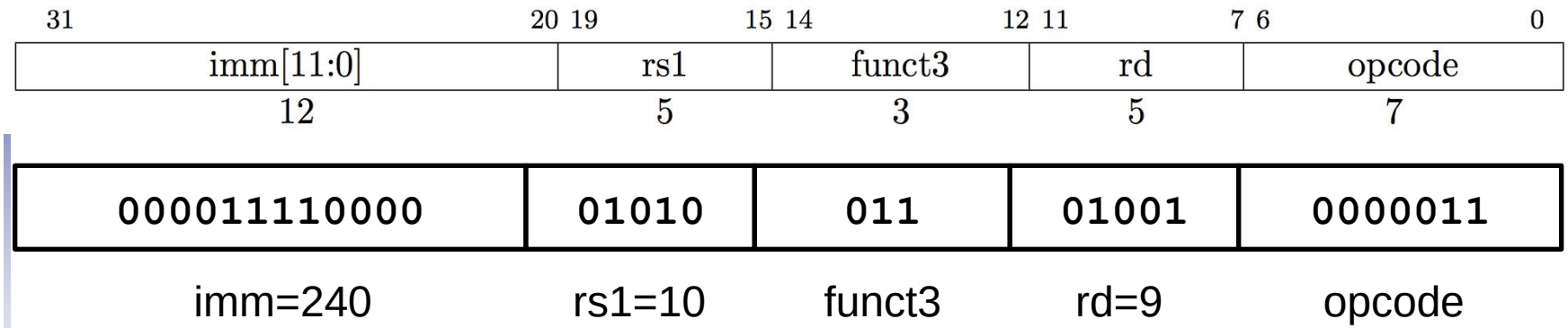
RISC-V 汇编指令:

```
addi   x15,x1,-50
addi   rd, rs1, immediate
```



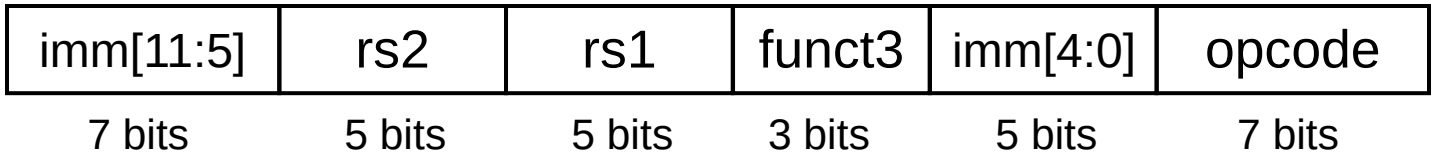

- imm[11:0] 表示值的范围： [$-2048_{ten}$ , $+2047_{ten}$ ]
- 在进行算术运算时，立即数总是先符号扩展到32位，再参加运算

# Load 指令也是I-Type




- ld x9, 240(x10)　　？可以访问多大数组?
- ld rd, imm(rs)
- 在load指令中，12位的立即数先符号扩展，再跟寄存器rs1中的基地址相加，得到最终的内存地址
- This is very similar to the add-immediate operation but used to create **address** not to create **final result**
- The value loaded from memory is stored in register **rd**

# S型指令格式——store指令



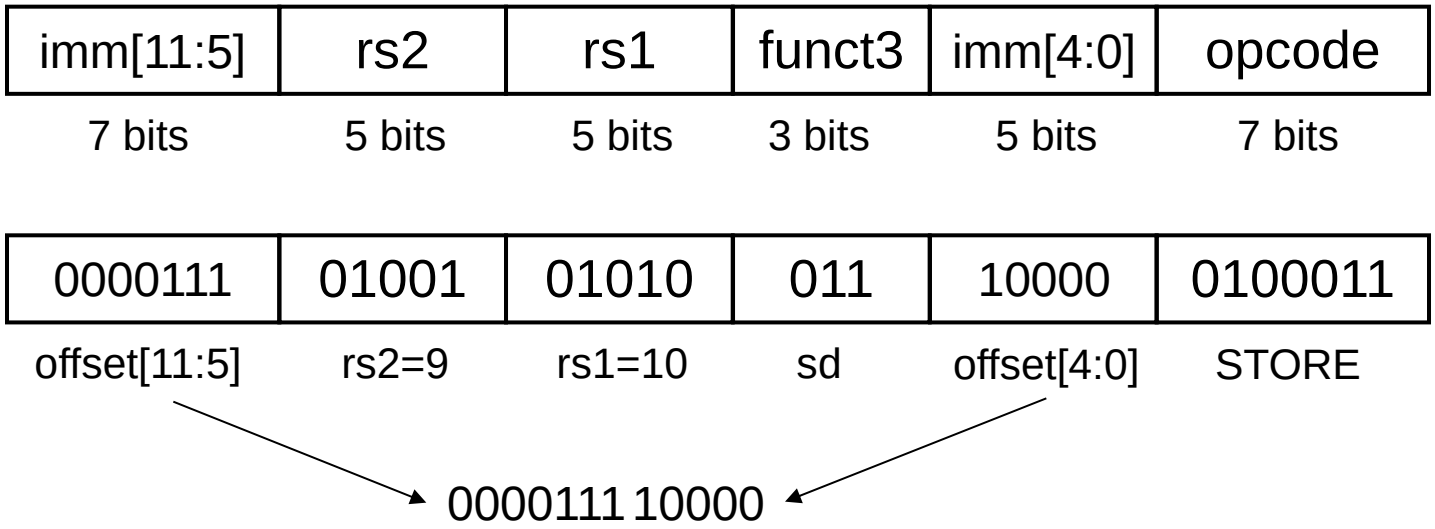

- store instructions不同的立即数格式
  - rs1: base address register number （基址寄存器）
  - rs2: source operand register number
  - immediate: offset added to base address
    - 留意：立即数被分成2块，为了让 rs1和rs2 字段保持同样位置
  - 例子
    - **sd x9, 240(x10)**

# S型指令格式的例子

- RISC-V 汇编指令:

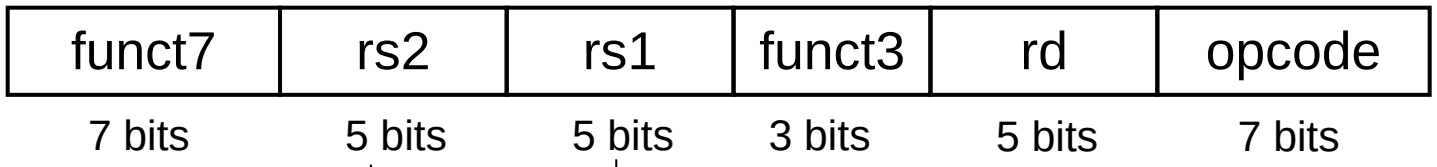
  **sd x9, 240(x10)**

  **sd  rs2, imm(rs1)**

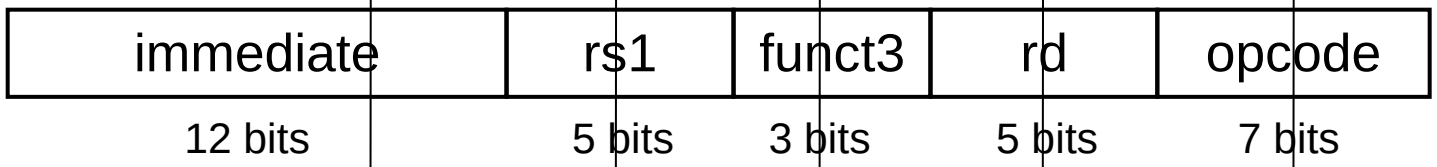


# 三种指令格式——尽量一致

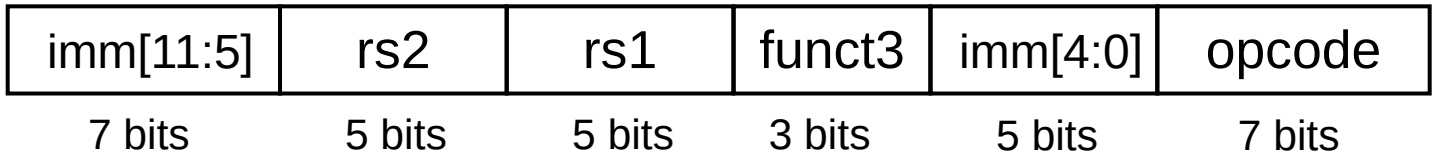
- R-format 指令




- I-format 指令




- S-format 指令

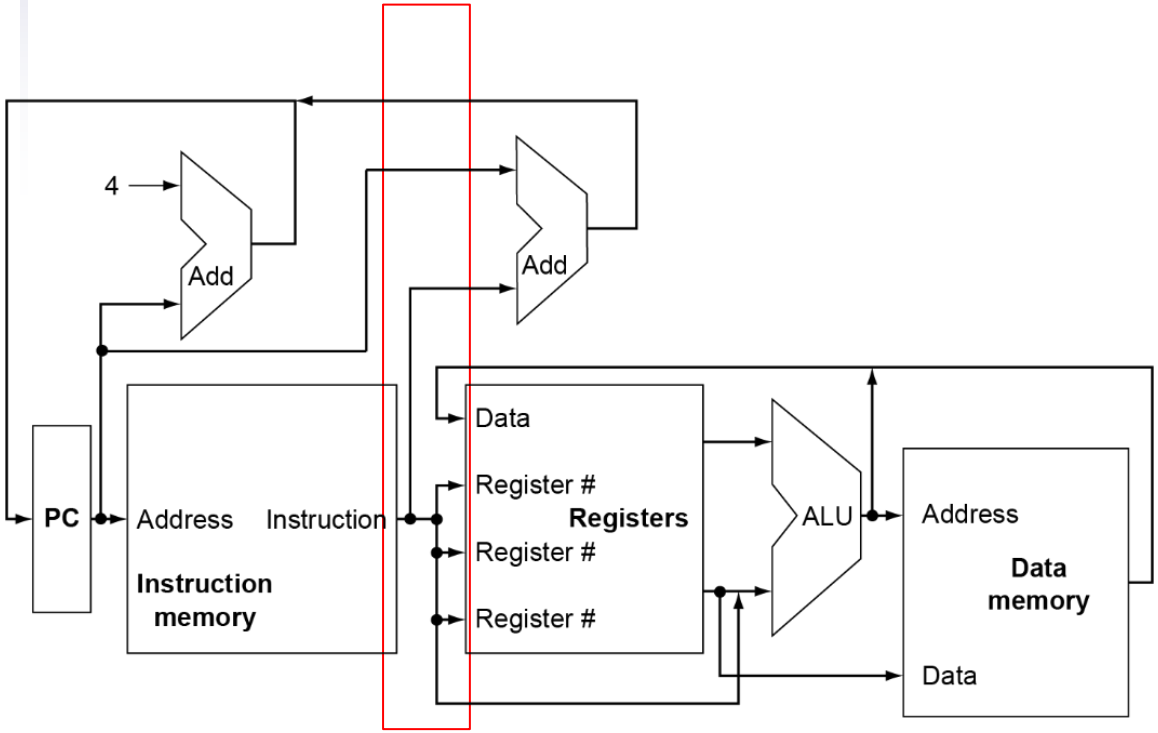


# 让寄存器保持同样位置...

- 所有指令都需要从寄存器中读取值
- 通过让所有read寄存器保持在同样位置，CPU可以：
  - 不需要知道指令类别，都可以毫不犹豫地去读取寄存器
  - 如果后面发现读的值无效（比如，I型指令不需要读取rs2），在后面阶段忽略掉该值就好
- 其他RISC类指令可能有略微不同的编码方案，后果是
  - 需要在读寄存器之前，先有一些硬件逻辑来识别指令的类别
- 这是RISC-V 中进行的许多小修改的一个示例

# 操作码

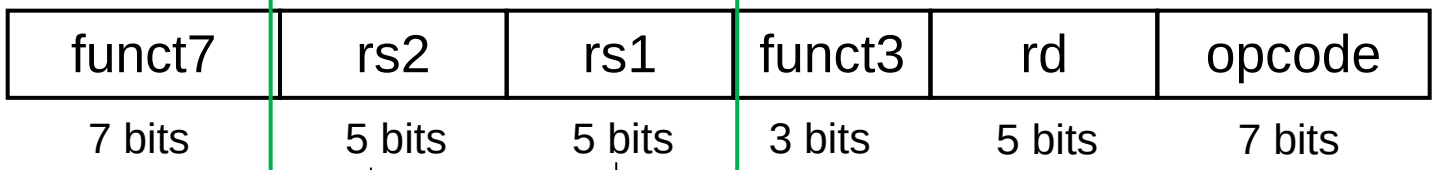
- 用于区分不同的指令格式，理论上需要最先解码
- 但是，现在可以一边解码一边读取寄存器



# RISC-V Instruction Formats

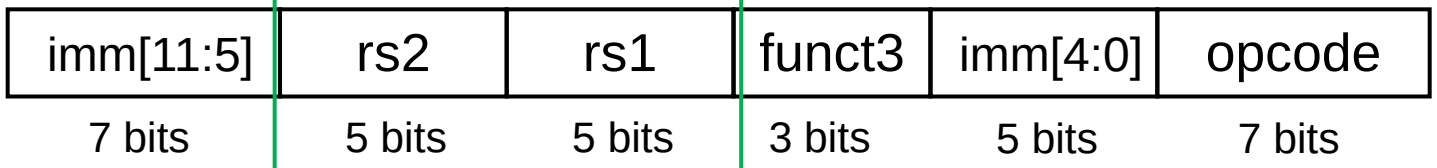
解码opcode同时，总能将24~15位视为读寄存器开始读取

- R-format Instructions




- I-format Instructions


- S-format Instructions



# 汇编指令到机器码的映射

- 汇编指令<-> 机器码：一一映射
  - 汇编：将汇编指令翻译成机器码（二进制）
    - ld x9, 240(x10)
  - 指令格式

| immediate | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

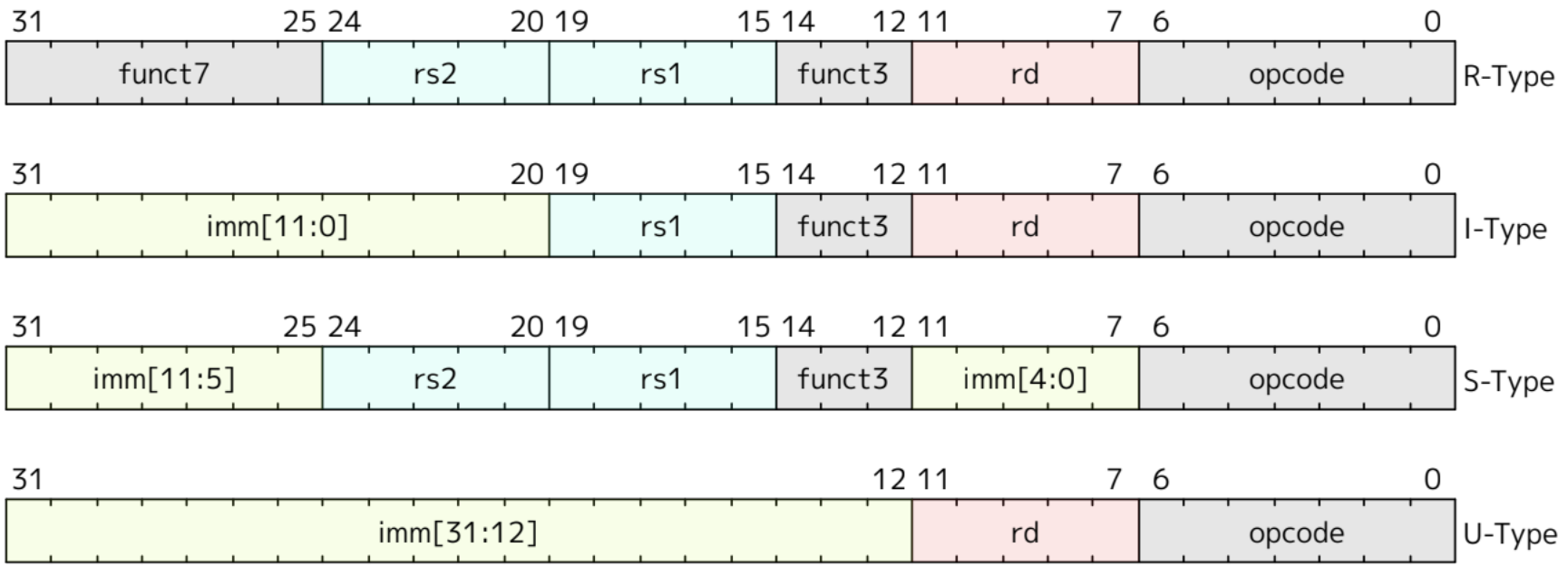
  - 反汇编：将机器码翻译成汇编指令

| immediate | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 000011110000 | 01010 | 011 | 01001 | 0000011 |

| Instruction | Format | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|---|---|
| add (add) | R | 0000000 | reg | reg | 000 | reg | 0110011 |
| sub (sub) | R | 0100000 | reg | reg | 000 | reg | 0110011 |
| **Instruction** | **Format** | **immediate** | | **rs1** | **funct3** | **rd** | **opcode** |
| addi (add immediate) | I | constant | | reg | 000 | reg | 0010011 |
| ld (load doubleword) | I | address | | reg | 011 | reg | 0000011 |
| **Instruction** | **Format** | **immed -iate** | **rs2** | **rs1** | **funct3** | **immed -iate** | **opcode** |
| sd (store doubleword) | S | address | reg | reg | 011 | address | 0100011 |

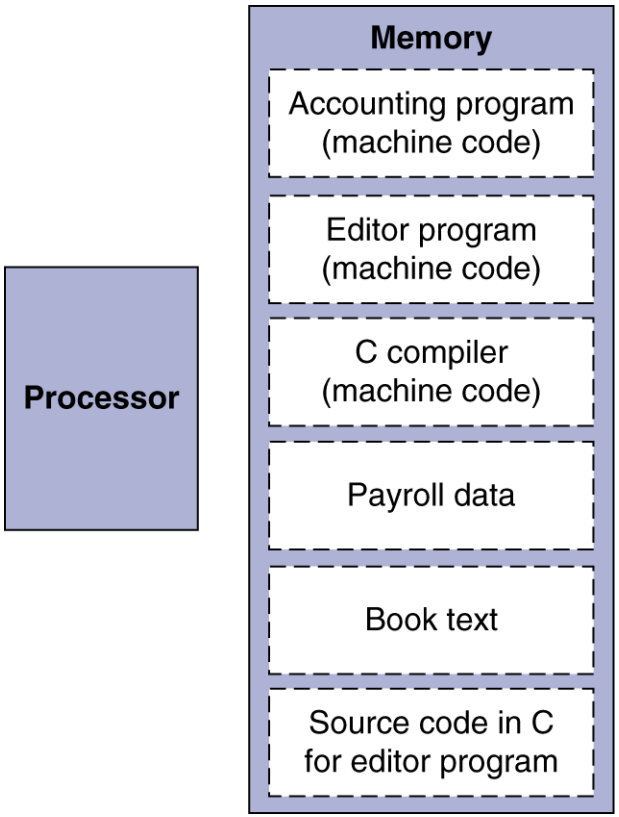
# 哪里看完整的指令编码方案

- riscv
  - https://riscv.org/technical/specifications/



# Stored Program Computers 存储程序计算机

The BIG Picture




- 指令（程序代码）像数据一样，用二进制表示，并存储在内存
- 因此，
  - 一台机器M可以通过下载一个程序的二进制文件P，运行任何程序
  - 只要该二进制文件P可以用该机器M的指令集来解释
- 程序可以用来处理另外的程序
  - 比如，编译器，链接器，…
- 指令也有内存地址:
  - 存于PC寄存器（不在x0～x31序列）

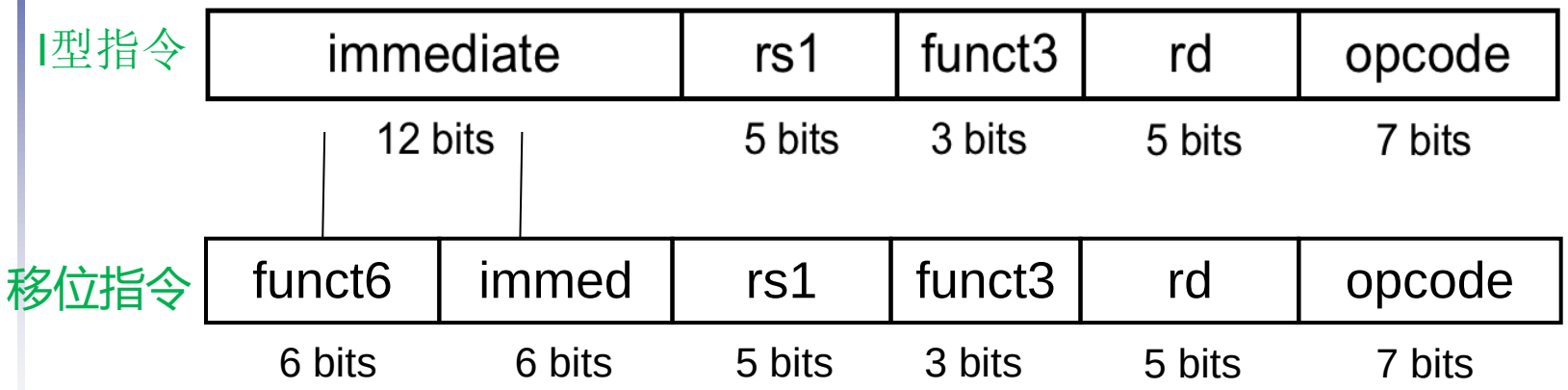
# 逻辑运算

- 用于bit级操作的逻辑运算指令

| 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 packing/unpacking groups of bits in a word
  - 可以用于访问宽度小于一个字的数据

# Shift Operations移位运算



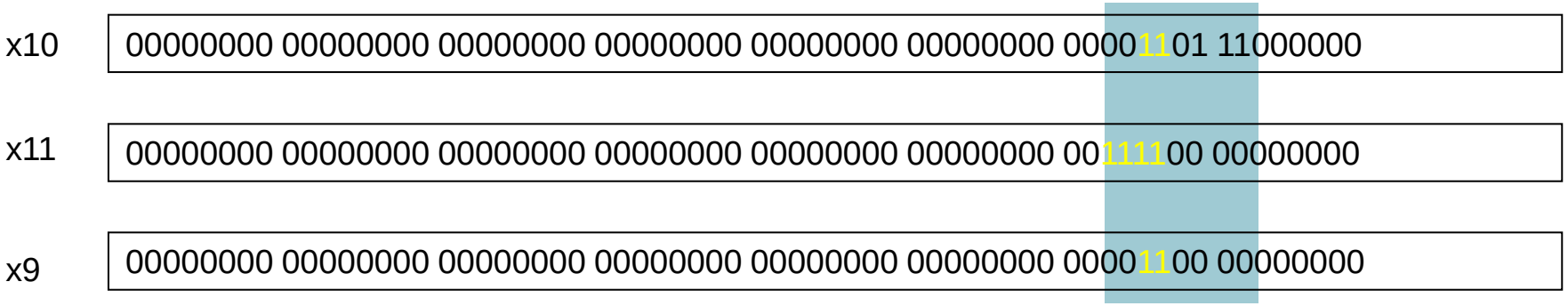

- immed: how many positions to shift
- Shift left **logical 逻辑左移**
  - 左移，右边补0
  - 左移i位，等于乘以$2^i$
- Shift right **logical 逻辑右移**
  - 右移，左边补0
  - 右移i位，等于除以$2^i$ (仅适用于无符号)

# 更多的移位运算

- Shift right arithmetic 算术右移
  - srai 右移，左边补符号位
  - 比如，假设寄存器x10中的值为
    1111 1111 1111 1111 1111 1111 1110 0111two= -25ten
  - 如果运行 sra x10, x10, 4, 则结果为:
    1111 1111 1111 1111 1111 1111 1111 1110two= -2ten
- 然而，右移n位，并不总是等于除以 $2^n$
  - 当被除数是奇负数时，会失效
  - C语言算术的语义，除法结果应该**向0取整**（截断小数部分）
    - -5/4 = -1(-1.25 )
    - -7/4 = -1(-1.75)
  - 然而，右移是**向下取整**
    - -5 >> 2 = -2(-1.25 )
    - -7 >> 2 = -2(-1.75)

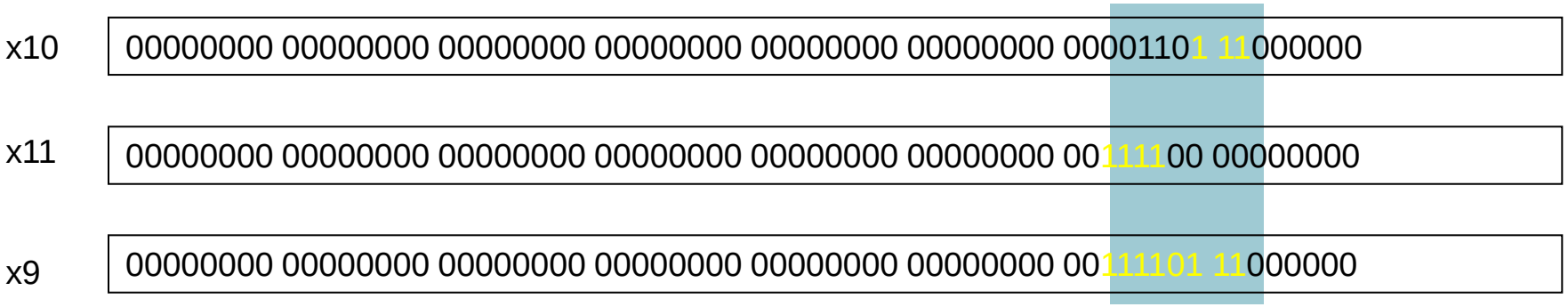
# AND 运算

- 用作掩码，掩盖word中的特定bit位
  - 可以保留一些bit位，清零另外的bit位
  - and x9,x10,x11
    - x10可以用作x11的掩码，x11也可以用作x10的掩码



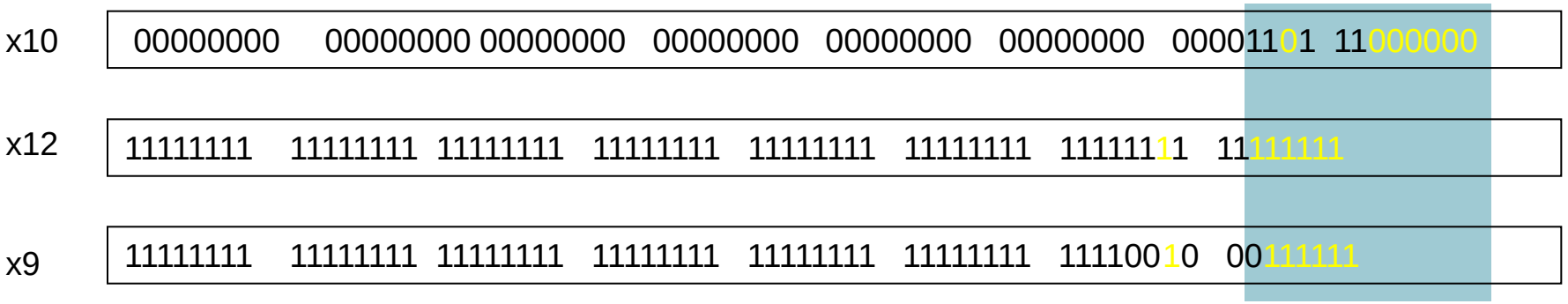
# OR 运算

- 用于引入一些bit位值到word中
  - 将字中的某些位设置位 1, 其他位不变
  - `or x9,x10,x11`
    - 合并x10和x11中bit位为1的值



# XOR 运算

- 俩操作数对应bit位，相同为0，不同为1
  - xor x9,x10,x12  // how to implement NOT operation?
    - 0跟x异或，得x
    - 1跟x异或，得x的非



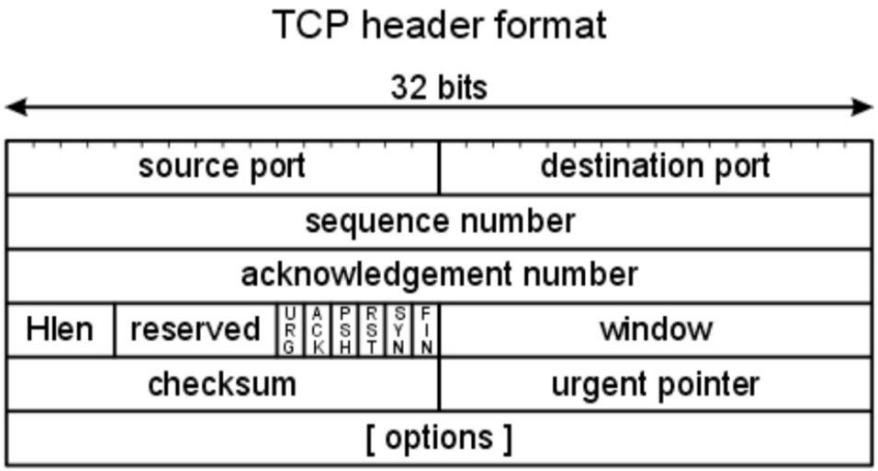
# 为什么需要移位和逻辑运算指令?

- 有时必须读写word内部的bit值
- e.g, in C:

```
int *packet;
packet[0] = src_port << 16 | dest_port;
```

- 对应的汇编代码(假设：packet in x1, src_port in x2, dest_port in x3)

```
slli x4, x2, 16
or x4, x4, x3
sw x4, 0(x1)
```

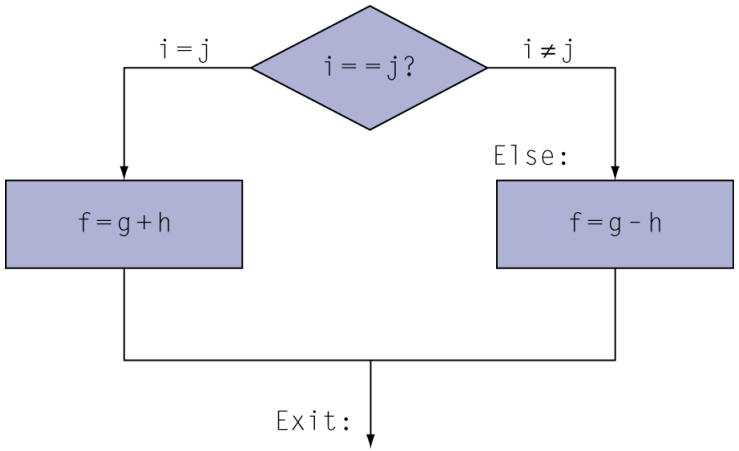


# Conditional Operations 条件运算

- 如果条件成立，则跳转到一条特定标记的指令
  - 否则，顺序往后执行
- beq rs1, rs2, L1
  - if (rs1 == rs2) 跳转到标记为L1的指令
- bne rs1, rs2, L1
  - if (rs1 != rs2) 跳转到标记为L1的指令

# Compiling If Statements 编译if语句

- C code:
- if (i==j) f = g+h;
  else f = g-h;
  - 假设f, g, … in x19, x20, …




- 编译后的 RISC-V 汇编指令:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // 无条件跳转
Else: sub x19, x20, x21
Exit: …
```

交给汇编器来计算标记对应的地址

# Compiling Loop Statements编译循环语句

- C code:

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

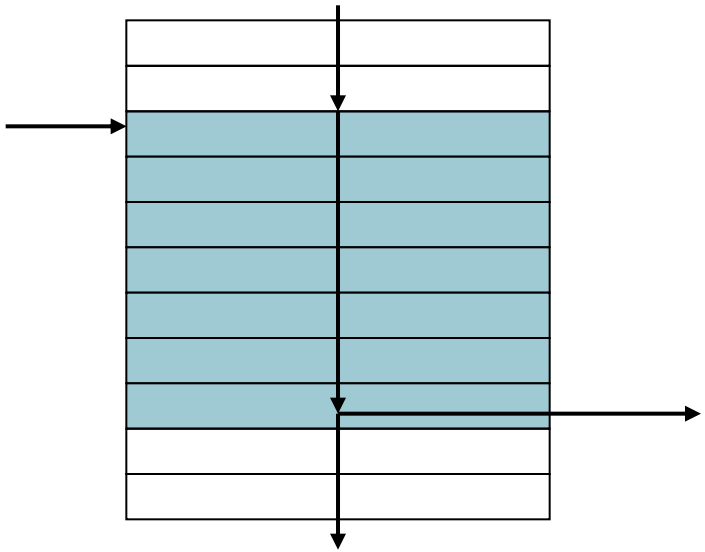
  - 假设:
    - i in x22, k in x24, save的基址 in x25
    - save数组每个元素8字节

- 编译后的RISC-V 指令:

```
Loop: slli x10, x22, 3    #i*8-> x10
      add  x10, x10, x25     # save + i*8,
即&(save[i])
      ld   x9, 0(x10)  # 加载save[i]到x9寄
存器
      bne  x9, x24, Exit     # 比较x9和x24
       addi x22, x22, 1
      beq  x0, x0, Loop    #实现了无条件转移
Exit: …
```

# Basic Blocks基本块

- 一个基本块是一段连续的指令序列
  - 内部没有跳转指令 (除了结尾)
  - 内部没有跳转的目标标签 (除了开始)



- 编译器识别基本块，以便优化
- 高级处理器能加速基本块的执行

# More Conditional Operations更多条件运算

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) 跳转到标记为L1的指令
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) 跳转到标记为L1的指令
- Example
  - if (a > b) a += 1;
  - 假设a in x22, b in x23

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

# Signed vs. Unsigned 区分有/无符号比较

- 有符号比较分支指令: blt, bge
- 无符号比较分支指令: 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

# More instructions 更多指令

- slt x3, x4, x5
  - if x3 < x4, **x5 -> 1**,
  - otherwise, **x5 -> 0**

**Additional Instructions in RISC-V Base Architecture**

| Instruction | Name | Format | Description |
|---|---|---|---|
| Add upper immediate to PC | auipc | U | Add 20-bit upper immediate to PC; write sum to register |
| Set if less than | slt | R | Compare registers; write Boolean result to register |
| Set if less than, unsigned | sltu | R | Compare registers; write Boolean result to register |
| Set if less than, immediate | slti | I | Compare registers; write Boolean result to register |
| Set if less than immediate, unsigned | sltiu | I | Compare registers; write Boolean result to register |
| Add word | addw | R | Add 32-bit numbers |
| Subtract word | subw | R | Subtract 32-bit numbers |
| Add word immediate | addiw | I | Add constant to 32-bit number |
| Shift left logical word | sllw | R | Shift 32-bit number left by register |
| Shift right logical word | srlw | R | Shift 32-bit number right by register |
| Shift right arithmetic word | sraw | R | Shift 32-bit number right arithmetically by register |
| Shift left logical word immedate | slliw | I | Shift 32-bit number left by immediate |
| Shift right logical word immediate | srliw | I | Shift 32-bit number right by immediate |
| Shift right arithmetic word immediate | sraiw | I | Shift 32-bit number right arithmetically by immediate |

**FIGURE 2.37 The remaining 14 instructions in the base RISC-V instruction set architecture.**

# 过程调用 & "ABI" 约定

- **Application Binary Interface** 定义了调用约定
  - 如何在指令级（二进制级）调用另外一个函数
- 一个关键的约定就是：
  - 在过程调用时，如何使用32个寄存器
  - 谁来负责保护寄存器?
- ABI 规定了一个契约: 当过程f调用g时：
  - f将参数放哪，g将返回值放哪?
  - g执行过程中，如何保证不会覆盖f放在寄存器和内存的数据?
  - g结束后，返回到哪?
- 为了方便使用，另有一套寄存器的命名
  - So going forward, no more x3, x6... type notation

# RISC-V 寄存器及约定

| Register | ABI Name | Description | Saved By Callee? |
|---|---|---|---|
| x0 | zero | Always Zero | N/A |
| x1 | ra | Return Address | No |
| x2 | sp | Stack Pointer | Yes |
| x3 | gp | Global Pointer | N/A |
| x4 | tp | Thread Pointer | N/A |
| x5-7 | t0-2 | Temporary | No |
| x8 | s0/fp | Saved Register/Frame Pointer | Yes |
| x9 | s1 | Saved Register | Yes |
| x10-x17 | a0-7 | Function Arguments/Return Values | No |
| x18-27 | s2-11 | Saved Registers | Yes |
| x28-31 | t3-6 | Temporaries | No |

# RISC-V 函数调用约定

- Registers faster than memory, so use them
- **a0–a7** (x10-x17):
  - 8个参数寄存器，2个返回值寄存器 (**a0-a1**) (caller saved)
  - 更多寄存器则必须放栈上
  - Technically we could return in **a2-a7** as well, but we're mostly dealing with C and not python or golang...
- **ra**(x1): 一个返回地址寄存器，记录返回点 (caller saved)
- **sp**: 指向栈顶 (callee saved)

# Procedure Calling 过程调用的工作原理

- 步骤
  1. 参数传递：Place parameters in registers x10 to x17
     - i.e., a0 ~ a7
  2. 控制传递：Transfer control to callee procedure
     - Acquire storage for callee procedure 为callee分配存储资源
     - Perform callee procedure's operations 完成计算任务
     - Place result in register for caller (a0~a1) 返回值传递
  3. 控制传回：Return to place of call (address in x1)
     - 即ra寄存器
  4. ...caller的后续工作
     - （可选）使用返回值：Access return value in caller

# 调用约定: caller与callee之间的契约...

- The “Calling Convention” in the ABI is the format/usage of registers in a way between the function ***caller*** and function ***callee***, if all functions implement it, everything works out
  - It is effectively a contract between functions
- Registers are two types
  - ***caller-saved***
    - The function invoked (the callee) can do whatever it wants to them!
  - ***callee-saved***
    - The function invoked must restore them before returning (if used)
  - 为何这么麻烦?

# 更多约定

- **s0-s11**
  - Callee saved registers: Preserved across function calls
  - caller的视角：可以假设callee不会修改这些寄存器
- **fp**
  - Frame Pointer: Pointer to the top of the call frame
  - Also is **s0**, the first saved register, callee saved
  - Frame pointer can often be omitted by the compiler, but we will use it because it makes things clearer how functions are translated.
- **t0-t6**
  - 临时存储: Caller saved
  - callee的视角：可以假设caller不需要使用这些寄存器的值

# 过程调用指令

- 过程调用: jump and link
  - `jal x1, ProcedureLabel`
    - 将下一条指令的地址放到 x1(ra)
    - 跳转到目标指令地址 (PC + immediate)
- 过程返回: jump and link register
  - `jalr x0, 0(x1)`:
    - Like jal, but jumps to 0 + address in x1（即ra寄存器）
    - Use x0 as rd (x0 cannot be changed)
  - 可以用于运行时可以更改的跳转目标
    - 比如 case/switch statements
    - 比如函数指针

# Caller和Callee的分工——完整版

| caller | callee |
| --- | --- |
| • **调用前**，保存caller-saving寄存器到caller栈上（t0～t6, ra, a0~a7）<br>• 将callee所需实际参数准备好，即拷贝到约定的参数位置（a0~a7）<br>• **调用时**，jal x1， calleeEntry指令：将返回地址填入约定的x1寄存器，同时将PC寄存器更新为calleeEntry | 1. 分配栈空间<br>2. 保护callee-saving寄存器到栈上(s0~s11)<br>3. 从约定的位置（a0～a7）读取参数值<br>4. 完成计算任务<br>5. 将结果存储到约定的返回值寄存器（a0）<br>6. 从栈上恢复callee-saving寄存器<br>7. 释放栈空间<br>8. 利用jalr x0， 0(x1)指令，跳转到约定的返回地址寄存器（x1） |
| • 调用后，从callee约定的返回值寄存器（a0）获取返回值<br>• 从栈上恢复caller-saving寄存器<br>• 往后执行... | |

# 叶子函数的例子

- 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;
}
```
- 假设
  - 参数 g, …, j in a0, …, a3
  - f in s4
  - temporaries s2, s3
  - Need to callee-save s2, s3, s4 on stack
```

# 函数调用前后，如何保护、恢复寄存器?

- 寄存器保护
  - 由于caller和callee共用寄存器，需要一个地方，
    - 在函数调用前 保护 寄存器的旧值
    - 在函数调用返回时，恢复 寄存器的旧值
  - 理想是使用一个 *stack* 结构: 一个后进先出的队列
    - Push: 将一个数据放在**栈顶**
    - Pop: 从**栈顶**取走一个数据
    - 栈只能放在内存中，因此需要一个寄存器来记录栈顶的地址
- **sp** is the *stack pointer* in RISC-V (**x2**)
  - **sp** 永远指向栈顶（最近使用过的位置）
  - 通常约定，栈从高地址往低地址增长
    - *Push* 减小 **sp**的值, *Pop* 增加 **sp** 的值

# 叶子函数的例子

- RISC-V code:

```
leaf_example:
   addi sp,sp,-24
   sd   s2,16(sp)
   sd   s3,8(sp)
   sd   s4,0(sp)
   add  s2,a0,a1
   add  s3,a2,a3
   sub  s4,s2,s3
   addi a0,s4,0
   ld   s4,0(sp)
   ld   s3,8(sp)
   ld   s2,16(sp)
   addi sp,sp,24
   jalr x0,0(x1)
```

1. 分配栈空间
2. Save s2, s3, s4 on stack

3-4. 读取参数值，并进行计算

t0 = g + h

t1 = i + j

f = t0 – t1

5. copy f to return register
6. Resore s2, s3, s4 from stack

7. 释放栈空间
8. Return to caller

# 非叶子函数

- 调用其他函数的函数
- 如何实现嵌套的函数调用
  - 在调用前，在栈上保存:
    - 返回地址（ra寄存器的旧值）
    - 调用之后仍然需要使用的：任何参数和临时值
  - 在调用后，从栈上恢复这些值

# 非叶子函数的例子

- C code:

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

  - 参数 n in x10(a0)
  - 返回值 in x10(a0)

# Caller和Callee的分工——完整版

| caller | callee |
| --- | --- |
| 1. **调用前**，保存caller-saving寄存器到caller栈上（t0～t6, ra, a0~a7)<br>2. 将callee所需实际参数准备好，即拷贝到约定的参数位置（a0~a7）<br>3. **调用时**，jal x1， calleeEntry指令：将返回地址填入约定的x1寄存器，同时将PC寄存器更新为calleeEntry<br><br>4. 调用后，从callee约定的返回值寄存器（a0）获取返回值<br>5. 从栈上恢复caller-saving寄存器<br>往后执行..... | a) 分配栈空间<br>b) 保护callee-saving寄存器到栈上(s0~s11)<br>c) 从约定的位置（a0～a7）读取参数值<br>d) 完成计算任务<br>e) 将结果存储到约定的返回值寄存器（a0）<br>f) 从栈上恢复callee-saving寄存器<br>g) 释放栈空间<br>h) 利用jalr x0， 0(x1)指令，跳转到约定的返回地址寄存器（x1） |

# 非叶子函数的例子

- RISC-V code: 橙色：callee操作流程；红色：caller操作流程

```
fact:
    addi sp,sp,-16      a) 分配栈空间
    sd   ra,8(sp)       1. Save return address on stack (以便递归调用callee时使用)
    sd   a0,0(sp)       1. Save 参数 n on stack (以便递归调用callee时使用)
    addi t0,a0,-1       c) t0 = n - 1
    bge  t0,x0,L1       d) if n-1 >= 0, go to L1
    addi a0,x0,1        e) Else, set return value to 1
    addi sp,sp,16       g) 释放栈空间，免去恢复寄存器——没有发生调用，没有破坏
    jalr x0,0(ra)       h) Return

L1: addi a0,a0,-1       2. 准备参数：n = n - 1
    jal  ra,fact        3. 调用call fact(n-1)
    addi t1,a0,0        4. 获取返回值 of fact(n - 1) to t1
    ld   a0,0(sp)       5. 恢复a0 caller's 参数n
    ld   ra,8(sp)       5. 恢复 caller's return address
    addi sp,sp,16       释放栈空间
    mul  a0,a0,t1       return n * fact(n-1)
    jalr x0,0(ra)       return
```

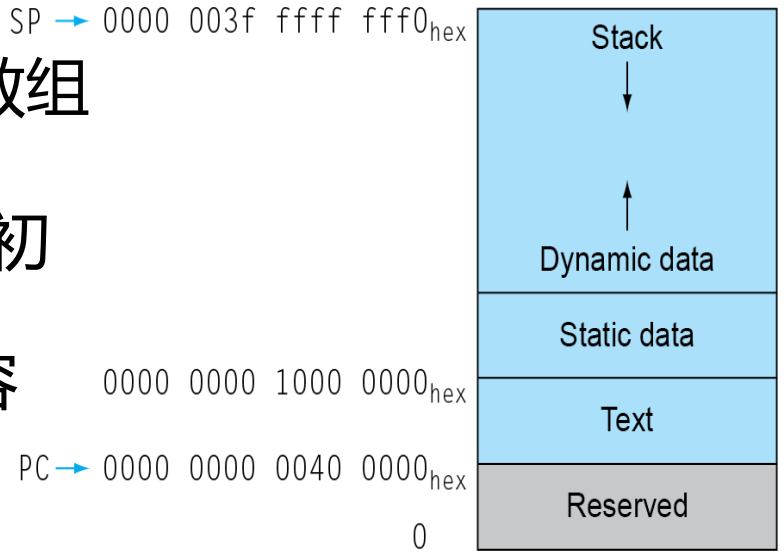
- C code:

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

  - Argument n in a0
  - Result in a0

# 内存的布局（内存地址的分布）

- Text: 程序的代码
- Static data: 全局变量
  - C语言中的static变量, 常量数组和字符串
  - 用x3 (global pointer)寄存器初始化为这段的基址，然后用±offsets来访问这个段的内容
- Dynamic data: 堆
  - E.g., **malloc** in C, **new** in Java/C++
- Stack: 自动存储



# 在栈上分配空间

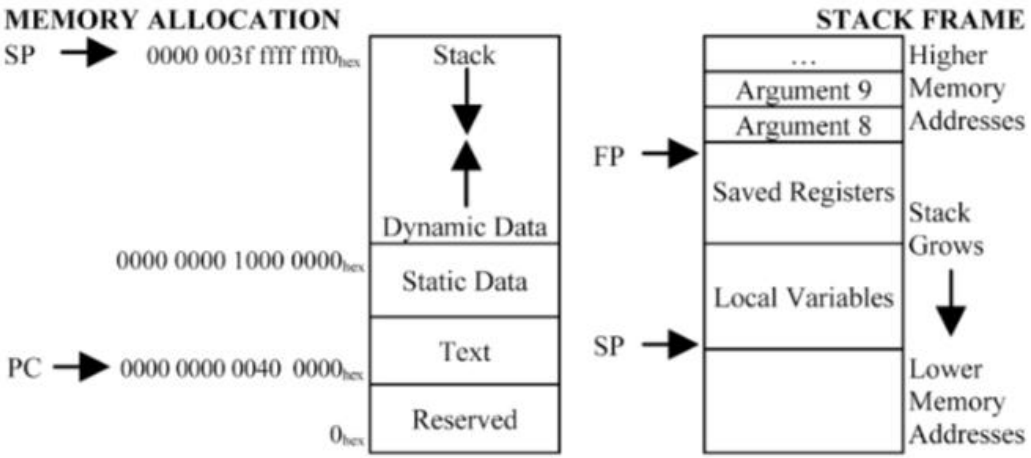
- C语言有两种存储类别: **automatic** and **static**
  - *Automatic* 变量是函数的局部变量，当函数结束时，无需保留
  - *Static* 变量在整个程序运行期间都需要保留值
- automatic变量（局部变量）尽量放寄存器，然后才放栈上
- *Procedure frame* or *activation record*:
  - 栈上用于存储**单次**函数调用的局部变量和寄存器保护的一段区域

# 栈上可以存储局部变量...

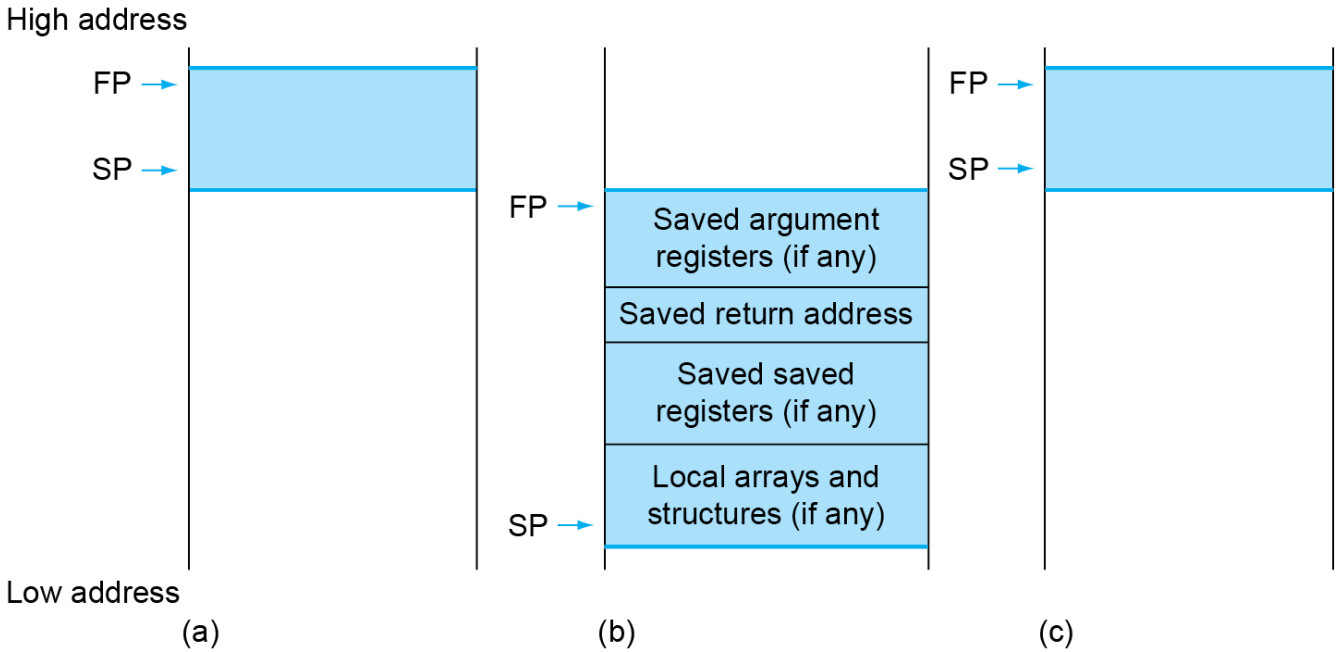
- e.g.
  **char[20] foo;**
- 需要在栈上分配
  - 可能需要填充（以便对齐到特定地址）
- 然后，可以用下列指令将foo的基址传递给a0寄存器
  - **addi a0 sp *offset-for-foo-off-sp***
    - *or,* If you are using the frame pointer...
  - **addi a0 fp *offset-for-foo-off-fp***

# 栈上可以存储参数

- 参数 1-8 用寄存器 a0-a7传递
- 那么第9+个参数呢?
  - 只能用栈空间传递!
- When the function is called,
  - 0(fp) -> arg #9
  - 4(fp) -> arg #10...



# 栈上的局部数据




- callee分配的局部变量
  - e.g., C语言中automatic 变量
- Procedure frame (activation record)
  - 一些编译器用来管理栈上的存储

# 练习

Which two statements are FALSE?

A. RISC-V uses **jal** to invoke a function and **jalr** to return from a function

B. jal saves PC+1 in ra （应该+4）

C. The callee can use **temporary registers** ($t_i$) without saving and restoring them

D. The caller can rely on **save registers** ($s_i$) without fear of callee changing them