

Computer Systems II

Li Lu

Room 605, CaoGuangbiao Building

li.lu@zju.edu.cn

<https://person.zju.edu.cn/lynnluli>



Preliminary of Following Labs on OS

- Semesters that need to learn before lab
 - <https://missing-semester-cn.github.io/>
 - 1/13: 课程概览与 shell
 - 1/14: Shell 工具和脚本
 - 1/21: 命令行环境
 - 1/22: 版本控制(Git)
 - 1/23: 调试及性能分析



Instruction Set Principles

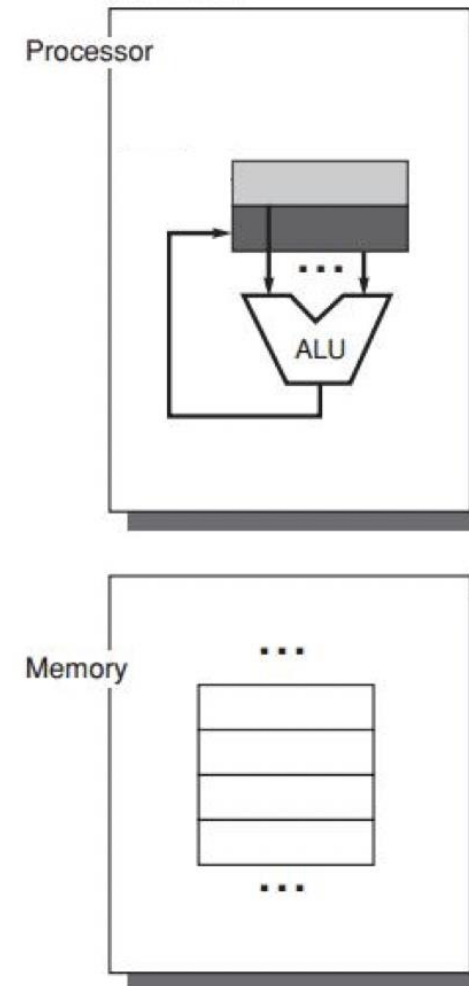


ISA Classification Basis

- The types of internal storage:

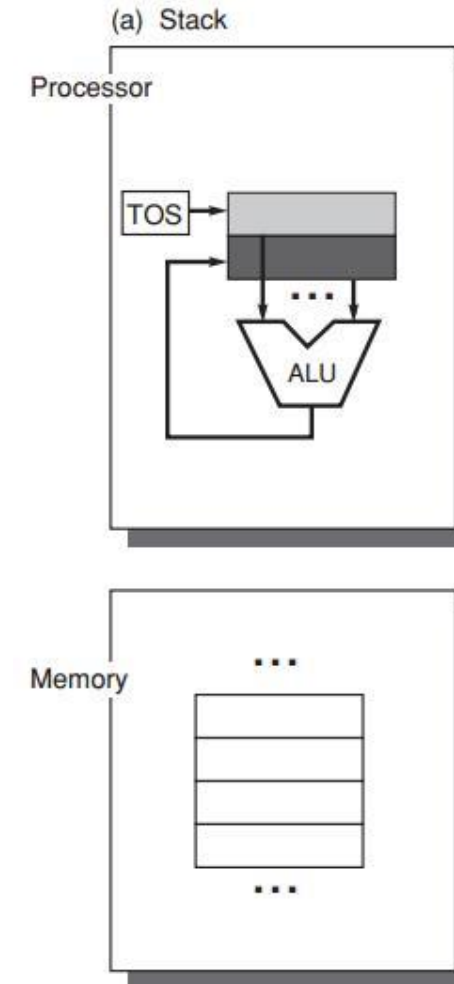
- Stack
- Accumulator
- Register

In processor, stores
data fetched from
memory or cache



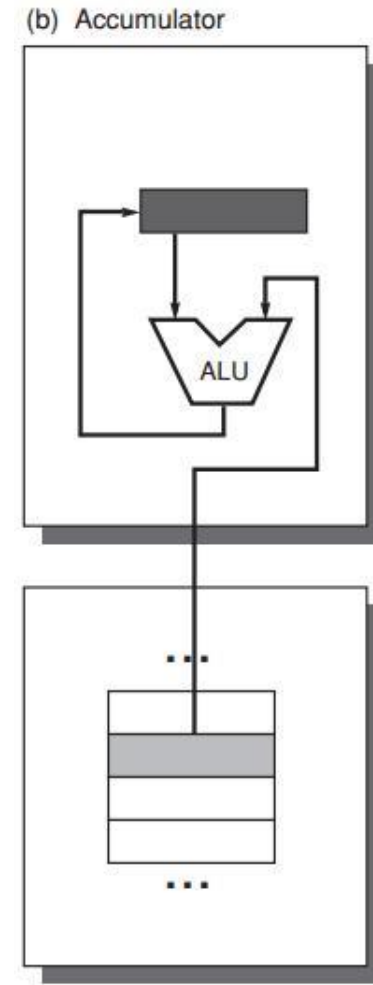
ISA Classes: Stack Architecture

- Implicit Operands
on the **T**op **O**f the **S**tack (TOS)
- $C = A + B$ (memory locations)
Push A
Push B
Add
Pop C



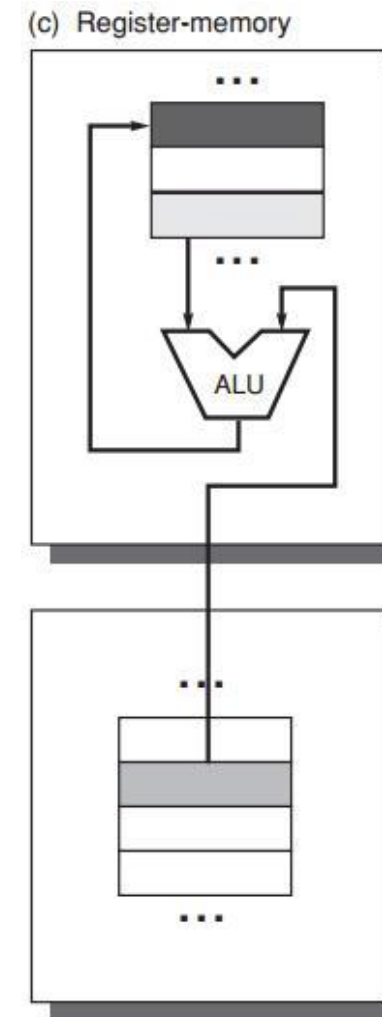
ISA Classes: Accumulator Architecture

- One implicit operand: the accumulator
one explicit operand: mem location
- $C = A + B$
Load A
Add B
Store C
- Accumulator is both an implicit input operand and a result



GPR: Register-Memory Arch

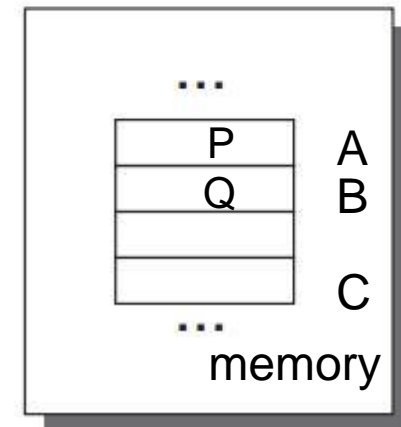
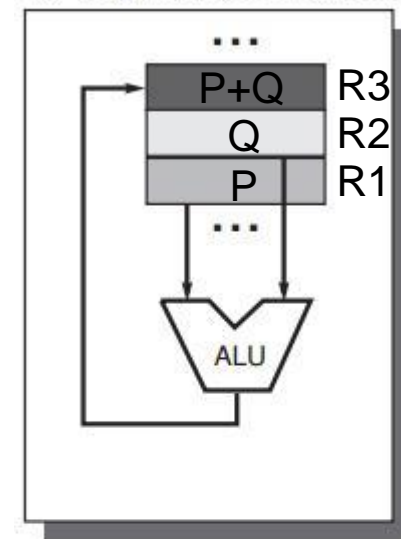
- Register-memory architecture
(any instruction can access memory)
- $C = A + B$
Load R1, A
Add R3, R1, B
Store R3, C



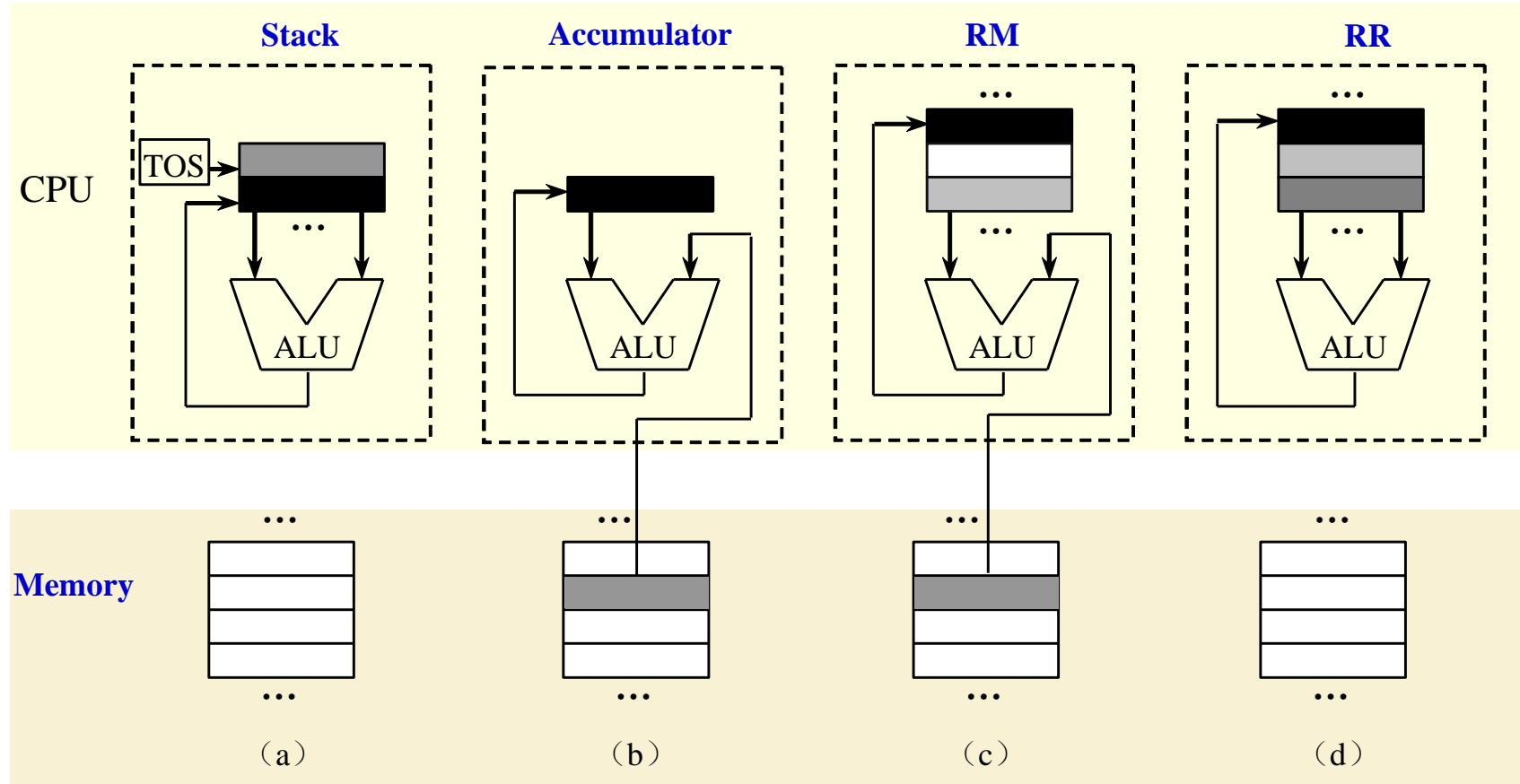
GPR: Load-Store Architecture

- Load-Store Architecture
 - only load and store instructions
 - can access memory
- $C = A + B$
 - Load R1, A
 - Load R2, B
 - Add R3, R1, R2
 - Store R3, C

(d) Register-register/load-store



GPR Classification



More about ISA



Where to find operands?



Interpret Memory Address

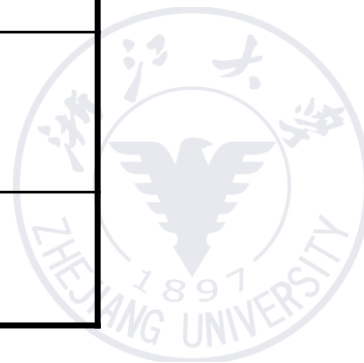
- Byte addressing

byte	– 8 bits
half word	– 16 bits
word	– 32 bits
double word	– 64 bits



Operand Type and Size

Type	Size in bits
ASCII character	8
Unicode character, Half word	16
Integer, word	32
Double word, Long integer	64
IEEE 754 floating point – single precision	32
IEEE 754 floating point – double precision	64
Floating point – extended double precision	80



Interpret Memory Address

- Byte ordering in memory: 0x12345678

Little Endian: store least significant byte in the smallest address

78 | 56 | 34 | 12

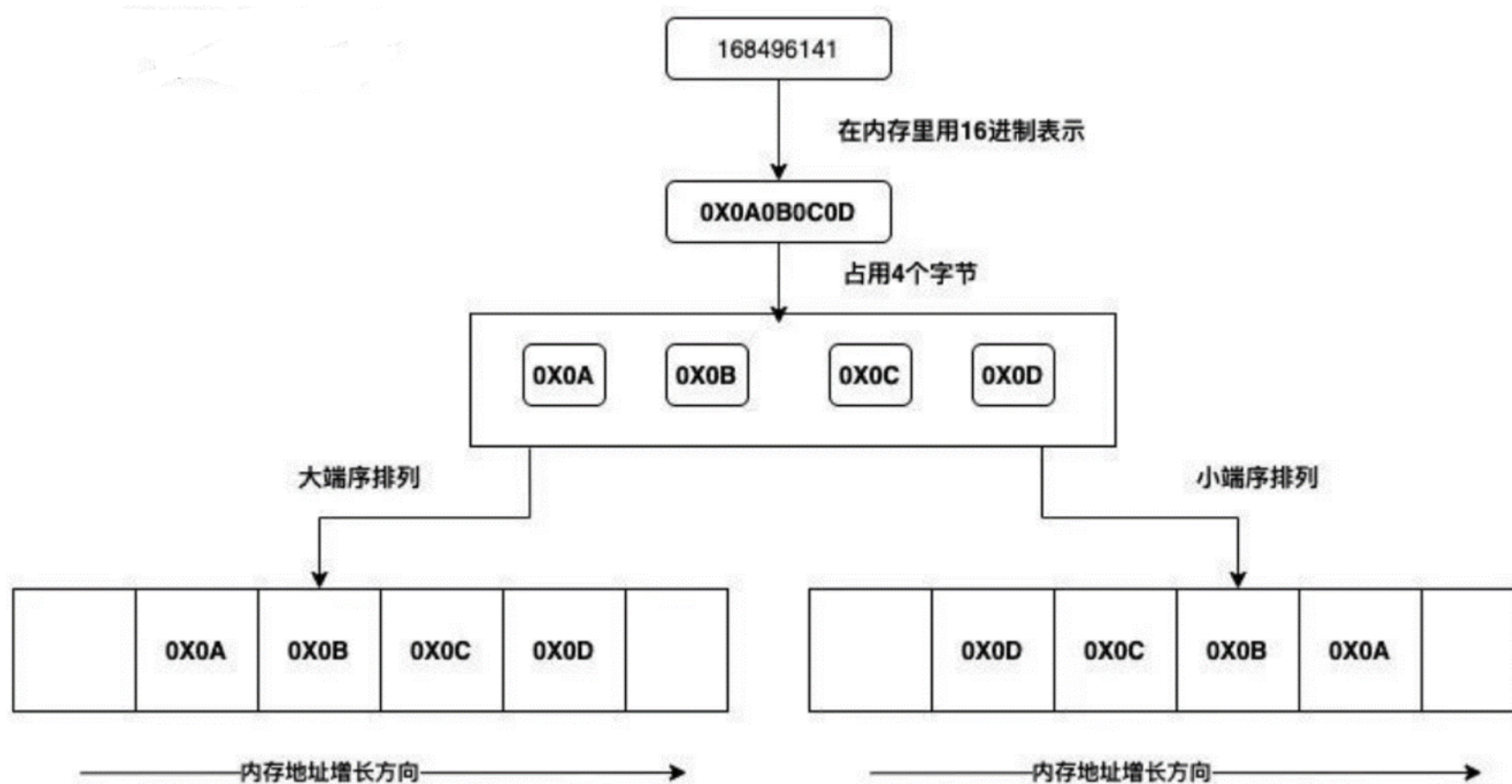
Big Endian: store most significant byte in the smallest address

12 | 34 | 56 | 78



Interpret Memory Address

- A more explicit example



Interpret Memory Address

- Address alignment

object width: s bytes

address: A

aligned if $A \bmod s = 0$



Interpret Memory Address

- Address alignment

object width: s bytes

address: A

aligned if $A \bmod s = 0$

- *Why to align addresses?*



Each misaligned object requires two memory accesses

	Value of 3 low-order bits of byte address								
Width of object	0	1	2	3	4	5	6	7	
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned		
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned	
4 bytes (word)	Aligned				Aligned				
4 bytes (word)		Misaligned				Misaligned			
4 bytes (word)			Misaligned				Misaligned		
4 bytes (word)				Misaligned					Misaligned
8 bytes (double word)	Aligned								
8 bytes (double word)		Misaligned							
8 bytes (double word)			Misaligned						
8 bytes (double word)				Misaligned					
8 bytes (double word)					Misaligned				
8 bytes (double word)						Misaligned			
8 bytes (double word)							Misaligned		
8 bytes (double word)								Misaligned	



Addressing Modes

- How instructions specify addresses of objects to access?
- Types
 - constant --- immediate
 - register
 - memory location – effective address



frequently
used

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4,#3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4,100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1+R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$.
Autoincrement	Add R1,(R2)+	$\begin{aligned} \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \\ \text{Regs}[R2] &\leftarrow \text{Regs}[R2] + d \end{aligned}$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d .
Autodecrement	Add R1,-(R2)	$\begin{aligned} \text{Regs}[R2] &\leftarrow \text{Regs}[R2] - d \\ \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \end{aligned}$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers.



How to operate operands?



Operations

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations



Control Flow Instructions

- Four types of control flow change:

P48

Conditional branches – most frequent

Jumps

Procedure calls

Procedure returns



Control Flow Instructions

- Explicitly specified destination address
(exception: procedure return *as target is not known at compile time*)
- PC-relative
destination addr = PC + displacement
- Dynamic address: for returns and indirect jumps with unknown target at compile time
e.g., name a register that contains the target address



Procedure Invocation Options

- Control transfer + State saving
- Return address must be saved
in a special link register or just a GPR

How to save registers?



Procedure Invocation Options: Save Registers

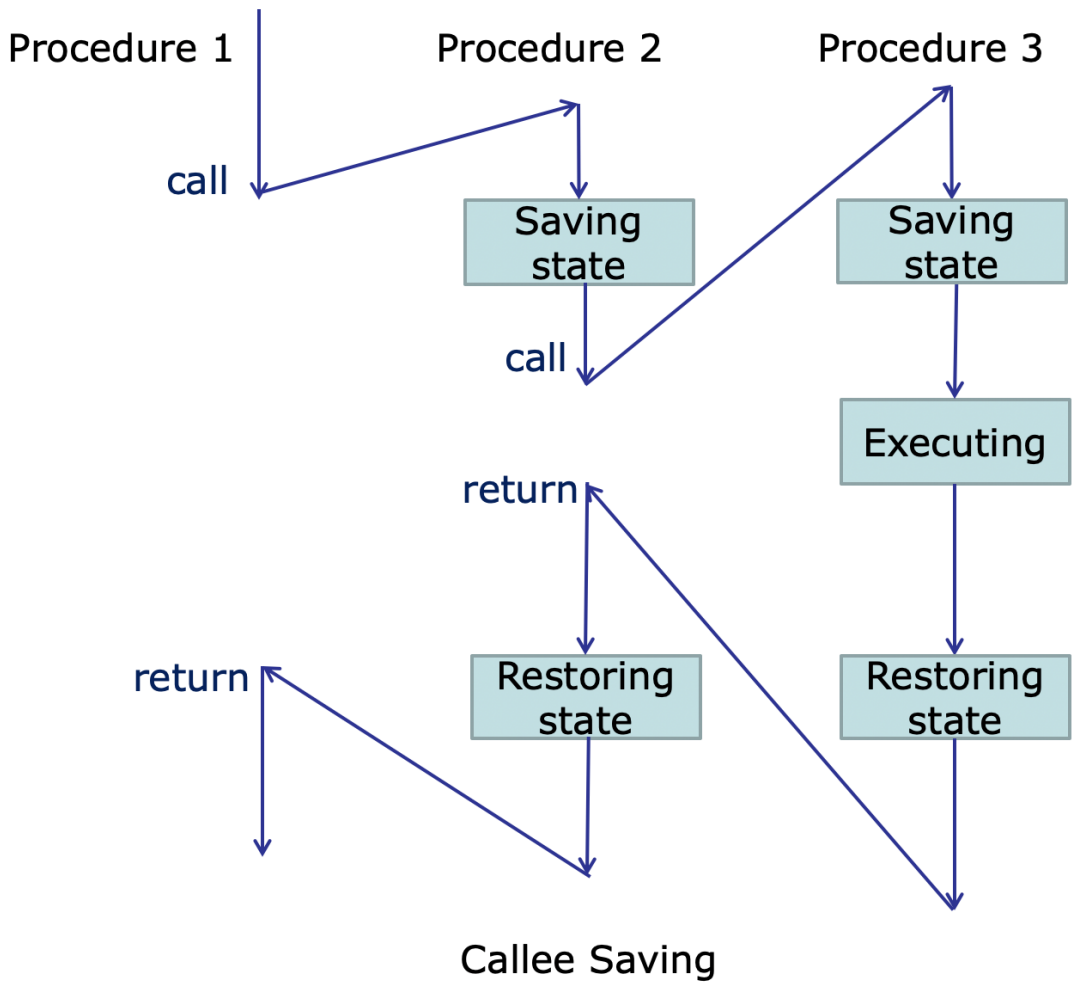
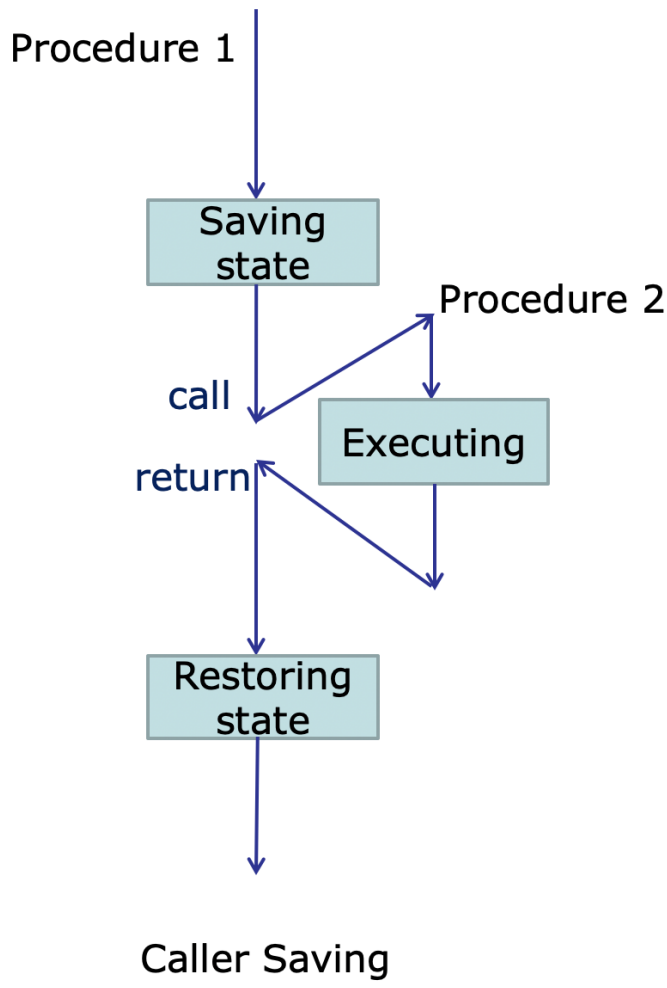
- Caller Saving

the calling procedure saves the registers that it wants preserved for access after the call

- Callee Saving

the called procedure saves the registers it wants to use





The improvement of ISA

- CISC
 - Disadvantage

- RISC
 - Advantage
 - characteristic



RISC ARCHITECTURE

- CISC (Complex Instruction Set Computer)
 - Intel x86
 - Variable length instructions, lots of addressing modes, lots of instructions.
 - Recent x86 decodes instructions to RISC-like micro-operations.
- RISC (Reduced Instruction Set Computer)
 - MIPS, Sun SPARC, IBM, PowerPC, ARM, RISC-V
- RISC Philosophy
 - fixed instruction lengths, uniform instruction formats
 - load-store instruction sets
 - limited number of addressing modes
 - limited number of operations



Brief History of RISC-V

- Origin: Research at UC Berkeley for Parallel Computing (From May 2010)
 - Why not X86, ARM or MIPS?
 - Expensive license fee, closed source or difficult to be expanded.....
- Development:
 - The **Chisel** hardware construction language that was used to design many RISC-V processors was also developed in the Par Lab
 - The RISC-V Foundation (www.riscv.org) was founded in 2015
 - Collaboration with the Linux Foundation (November 2018)
- For more: <https://riscv.org/about/history/>



RISC-V ISA

- A RISC-V ISA is defined as a base integer ISA
 - The base integer ISAs are very similar to that of the early RISC processors(such as MIPS)
 - No branch delay slots
 - Support for optional variable-length instruction encodings
- Goal: A ***standard free*** and ***open*** architecture for industry implementations



RISC-V Instruction Modularization

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

- RISC-V ISA modules
 - I: Integer Compute + Load/Store + Control Flow
 - M: Multiplication + Division extension
 - A: Atomically read, modify, and write memory for inter-processor synchronization
 - F: Floating-point registers, single-precision computation + load/store
 - D: Floating-point registers, double-precision computation + load/store
 - ...



Formats of Instruction

- Six Basic Instruction formats (similar to MIPS, but optimized)
 - Reduce combinational logic delay
 - Extend addressing range

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0		
funct7				rs2			rs1		funct3		rd			opcode		R-type
imm[11:0]						rs1		funct3		rd			opcode		I-type	
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]	imm[11]	opcode		B-type
imm[31:12]									rd			opcode			U-type	
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type	

Register-Register
Register Immediate(16bits)
+ Load
Store
Branch
Register Immediate(20bits)
Jump



RISC-V Instruction Encoding(RV32I)

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	iw	0000011	010	n.a.
	id	0000011	011	n.a.
	ibu	0000011	100	n.a.
	ihu	0000011	101	n.a.
	iwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

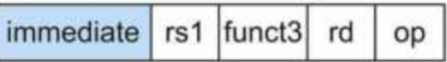
Great Idea in RISC-V ISA Design:
Make the Common Case Fast

- *Jal* in RV32I, *J* in RV32C (Why?)

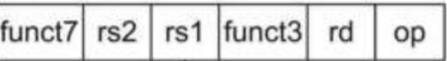


RISC-V Address Mode

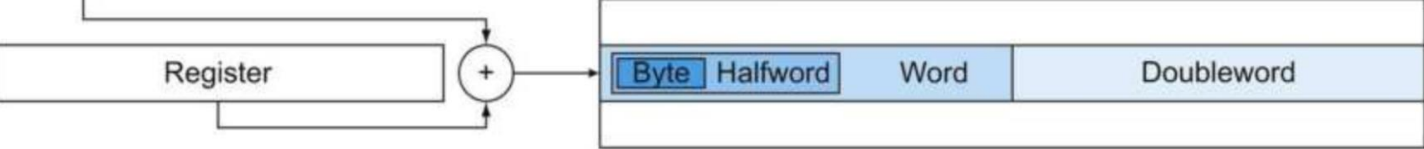
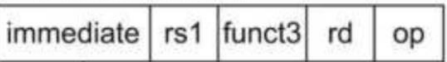
1. Immediate addressing



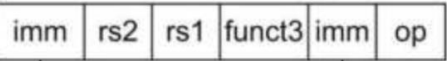
2. Register addressing



3. Base addressing



4. PC-relative addressing



- No Pseudodirect Address (Why?)
- Example: *Jal*
- MIPS: jal offset $\rightarrow 2^{26}$
- RISC-V: jal offset(rd) $\rightarrow 2^{32} \rightarrow$ More Address Space + Support for half word address



Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”

- **Assembler names**

- **x0: constant 0**
- x1: link register
- x2: stack pointer
- x3: global pointer

原则上遵循

- x4: thread pointer
- x5-x7, x28-x31: temporary
- x8-x9, x18-x27: save
- x10-x17: parameter/result



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
- Words are aligned in memory
 - Address must be a multiple of 4
- RISC-V is **Little Endian**
 - Least-significant byte at least address
 - *c.f.* Big Endian: most-significant byte at least address of a word



Memory Operand Example 1

- C code:

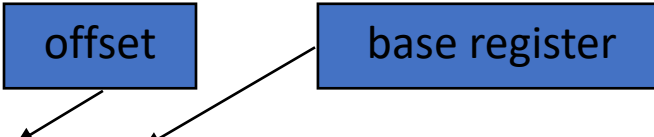
`g = h + A[8];`

`g` in `x8`, `h` in `x9`, base address of `A` in `x18`

- Compiled RISC-V code:

- Index 8 requires offset of 32

- 4 bytes per word 每个元素 32 位，8 个字节


`lw x5, 32(x18) # load word`
`add x8, x9, x5`



Memory Operand Example 2

- C code:

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

- h in $x8$, base address of A in $x9$

- Compiled RISC-V code:

- Index 8 requires offset of 32

`lw x5, 32(x9)` # load word

`add x5, x8, x5`

`sw x5, 48(x9)` # store word



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 x8, x8, 4`

- No subtract immediate instruction

- Just use a negative constant

`addi x8, x9, -1`

- *Design Principle 3: Make the common case fast*

- Small constants are common
 - Immediate operand avoids a load instruction



Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

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



Shift Operations

op	rd	func3	rs1	imm[11:0]
7 bits	5 bits	3 bits	5 bits	12 bits

- Instructions for bitwise manipulation
- Useful for extracting and inserting groups of bits in a word



AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0
- Example: and x5, x6, x7

\$x7	0000 0000 0000 0000 0000 1101 1100 0000
\$x6	0000 0000 0000 0000 0011 1100 0000 0000
\$x5	0000 0000 0000 0000 0000 1100 0000 0000



OR Operations

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

- Example: or x5, x6, x8

\$x8	0000 0000 0000 0000 0000 1101 1100 0000
\$x6	0000 0000 0000 0000 0011 1100 0000 0000
\$x5	0000 0000 0000 0000 0011 1101 1100 0000



Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- ***beq x5, x6, L1***
 - if (rs == rt) branch to PC+L1;
- ***bne x5, x6, L1***
 - if (rs != rt) branch to PC+L1;
- ***jal x1, L1***
 - unconditional jump to PC+L1, and store the previous address PC+4 in x1 for return

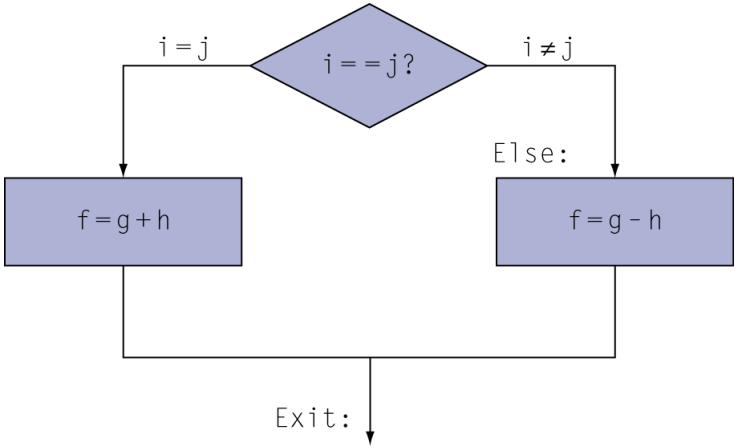


Compiling If Statements

- C code:

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

- f, g, h, i, j in x5, x6, x7, x8, x9



- Compiled RISC-V code:

- bne x8, x9, 12
- add x5, x6, x7
- jal x1, 8
- sub x5, x6, x7
- Exit: ...

Assembler calculates addresses



Compiling Loop Statements

- C code:

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

- i in x3, k in x5, address of save in x6

- Compiled RISC-V code:

- Loop: sll x8, x3, 2
 add x8, x8, x6
 lw x7, 0(x8)
 bne x7, x5, 12
 addi x3, x3, 1
 jal x1, -20

- Exit: ...



More Conditional Operations

- Set result to 1 if a condition is true
 - Otherwise, set to 0
- ***slt x5, x6, x7***
 - if ($x6 < x7$) $x5 = 1$; else $x5 = 0$;
- ***slti x5, x6, immediate***
 - if ($x6 < \text{immediate}$) $x5 = 1$; else $x5 = 0$;
- Use in combination with beq, bne
 - slt x5, x6, x7 # if ($x6 < x7$)
 - bne x5, x0, L1 # branch to PC+L1



Branch Instruction Design

Why not blt, bge, etc?

- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise
- RISC-V supports blt, bge, etc.



Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
 - $x5 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x6 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - `slt x2, x5, x6 # signed`
 - $-1 < +1 \Rightarrow x2 = 1$
 - `sltu x2, x5, x6 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow x2 = 0$



Procedure Calling

- Steps required
 1. Place parameters in registers
 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



Procedure Call Instructions

- Procedure call: jump and link

jal x1, ProcedureOffset

- Address of following instruction put in x1
- Jumps to target address, i.e., PC+Offset

- Procedure return: jump register

jalr x3, x1, offset

- Copies the address in x1 plus offset to program counter
- Address of following instruction put in x3



Leaf Procedure Example

- C code:

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

- Arguments g, ..., j in x10, ..., x13
- f in x7 (hence, need to save x7 on stack)
- Result in x17



Leaf Procedure Example

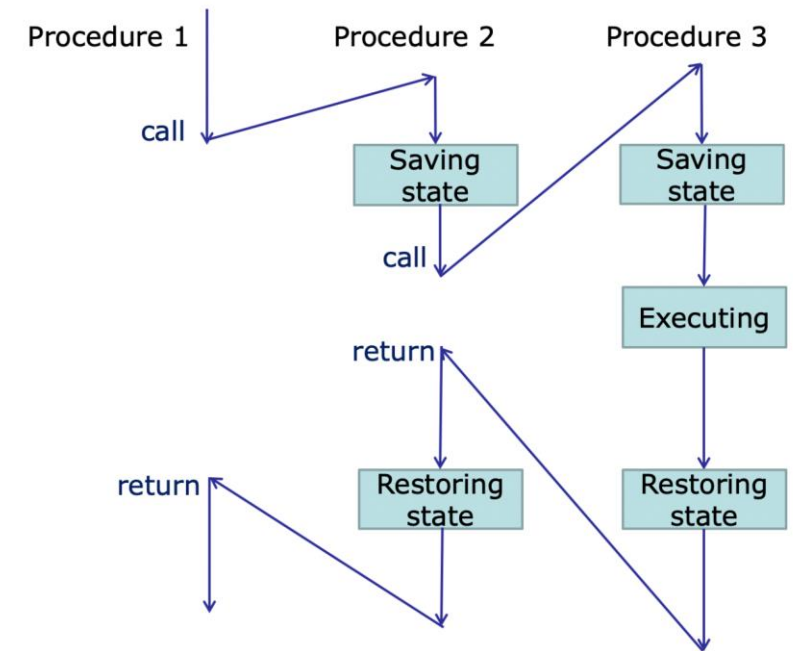
RISC-V code:

<i>leaf_example:</i>	
<i>addi x2, x2, -4</i> <i>sw x7, x2, 0</i>	<i># Save x7 on stack</i>
<i>add x5, x10, x11</i> <i>add x6, x12, x13</i> <i>sub x7, x5, x6</i>	<i># Procedure body</i>
<i>add x17, x7, x0</i>	<i># Result</i>
<i>lw x7, x2, 0</i> <i>addi x2, x2, 4</i>	<i># Restore x7</i>
<i>jalr x0, x1, 0</i>	<i># Return</i>

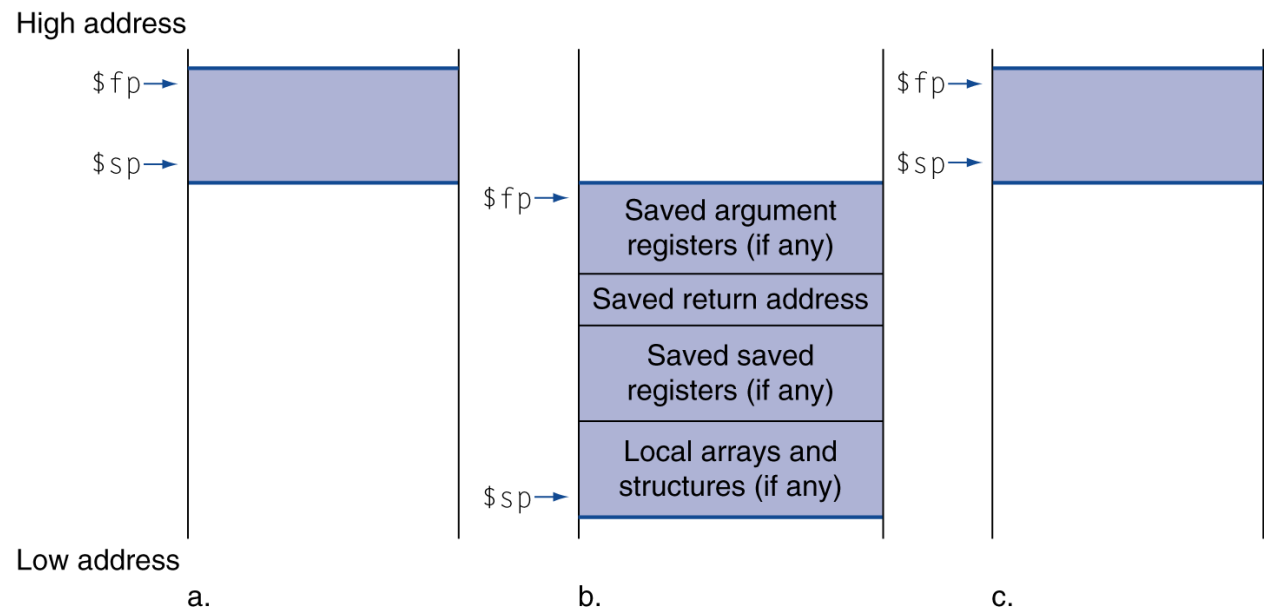


Leaf Procedure Example

- 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



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



Non-Leaf Procedure Example

- C code:

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

- Argument n in x5
- Result in x17



Non-Leaf Procedure Example

RISC-V code:

<i>fact:</i>	
<i>addi x2, x2, -8</i>	<i># adjust stack for 2 items</i>
<i>sw x1, x2, 4</i>	<i># save return address</i>
<i>sw x5, x2, 0</i>	<i># save argument</i>
<i>slti x6, \$x5, 1</i>	<i># test for n < 1</i>
<i>beq x6, x0, L1</i>	
<i>addi x17, x0, 1</i>	<i># if so, result is 1</i>
<i>addi x2, x2, 8</i>	<i># pop 2 items from stack</i>
<i>jalr x0, x1, 0</i>	<i># and return</i>
<i>L1: addi x5, x5, -1</i>	<i># else decrement n</i>
<i>jal x1, -36</i>	<i># recursive call</i>
<i>lw x5, x2, 0</i>	<i># restore original n</i>
<i>lw x1, x2, 4</i>	<i># and return address</i>
<i>addi x2, x2, 8</i>	<i># pop 2 items from stack</i>
<i>mul x17, x5, x17</i>	<i># multiply to get result</i>
<i>jalr x0, x1, 0</i>	<i># and return</i>



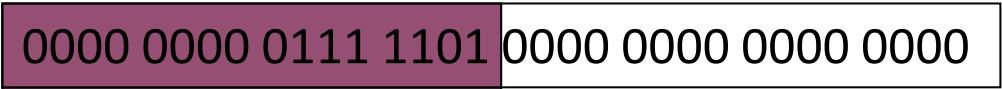
32-bit Constants

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

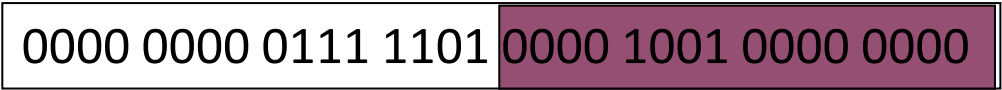
lui rt, constant

- Copies 16-bit constant to left 16 bits of rt
- Clears right 16 bits of rt to 0

lui x5, 61

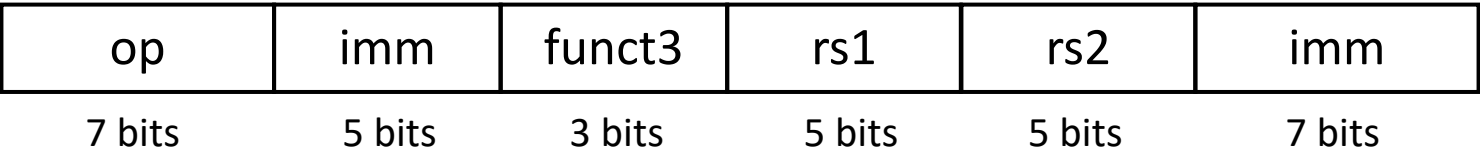


ori x5, x5, 2304



Branch Addressing

- Branch instructions specify
 - Opcode, two registers, offset to target address
- Most branch targets are near branch
 - Forward or backward

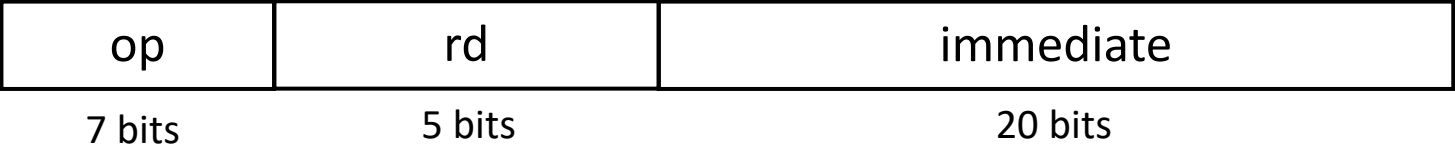


- PC-relative addressing
 - Target address = PC + offset \times 4
 - PC already incremented by 4 by this time



Jump Addressing

- Jump (jal) targets could be anywhere in text segment
 - Encode full address in instruction



Addressing Mode Summary

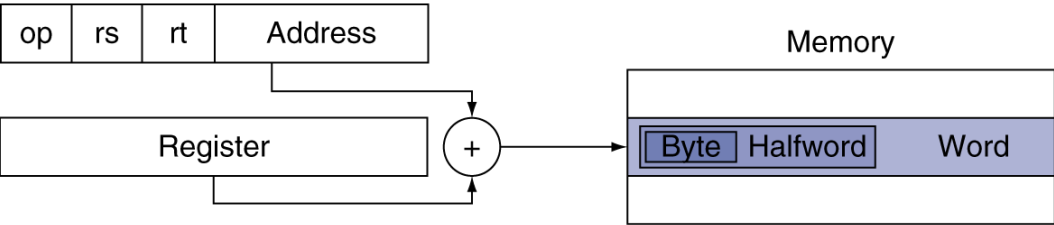
1. Immediate addressing



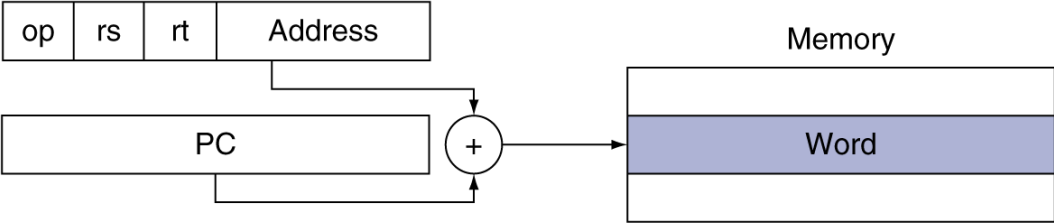
2. Register addressing



3. Base addressing



4. PC-relative addressing



The Four ISA Design Principles

1. Simplicity favors **regularity**
 - Consistent *instruction size, instruction formats, data formats*
 - Eases implementation by simplifying hardware, leading to higher performance
2. Smaller is faster
 - Fewer bits to access and modify
 - Use the register file instead of slower memory
3. Make the **common case fast**
 - e.g. Small constants are common, thus small immediate fields should be used.
4. Good design demands good compromises
 - Compromise with special formats for important exceptions
 - e.g. A long jump (beyond a small constant)



The Four ISA Design Principles

- **Simplicity favors regularity**

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

- C code:

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

- Compiled RISC-V code:

add x5, g, h	# temp x5 = g + h
add x6, i, j	# temp x6 = i + j
sub f, x5, x6	# f = x5 − x6



The Four ISA Design Principles

- **Smaller is faster**

- C code:

$f = (g + h) - (i + j)$

f, g, h, i, j in $x8, x9, x18, x19, x20$

- Compiled RISC-V code:

add x5, x9, x18

add x6, x19, x20

sub x8, x5, x6



The Four ISA Design Principles

- **Make the common case fast**

- Constant operators appear at high frequencies.
- Compared to taking constants from memory, arithmetic instructions containing constants run faster with less overhead.
- RISC-V register 0 (x0) is the constant 0
 - ✓ Cannot be overwritten
 - ✓ Useful for common operations
 - ✓ E.g., move between registers
 - add x5, x8, x0



The Four ISA Design Principles

- **Good design demands good compromises**

Keep all instructions with the same length, but different types of instructions are in different instruction formats.

31	25 24	20 19	15 14	12 11	7 6	0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]					rd	opcode	U-type

Although hardware is complicated by multiple instruction formats, maintaining the similarity of instruction formats can reduce complexity.



Example: Optimized representation of opcode

- There is a model machine with 7 instructions in total. The frequency of instruction usage is shown in the following table. Try to give the instruction code length for this machine.

Instruction	Frequency	Instruction	Frequency
I1	0.40	I5	0.04
I2	0.30	I6	0.03
I3	0.15	I7	0.03
I4	0.05		

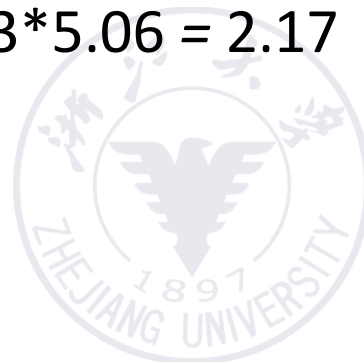


Example: Optimized representation of opcode

- [Method 1] Information entropy H : The average amount of information contained in the information source. When various instructions appear independent of each other:

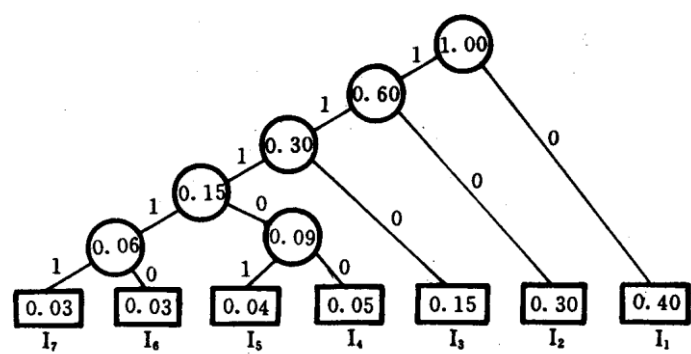
$$H = \sum_{i=0}^N -p_i \log_2 p_i$$

- Use fixed-length binary code:
 - Average code length = 3
 - $H = 0.4 * 1.32 + 0.3 * 1.74 + 0.15 * 2.74 + 0.05 * 4.32 + 0.04 * 4.61 + 0.03 * 5.06 + 0.03 * 5.06 = 2.17$
 - Information redundancy: $(3 - 2.17) / 3 \approx 28 \%$



Example: Optimized representation of opcode

- [Method 2] Huffman coding



Instruction	Frequency	Huffman Code	op Length
I1	0.40	0	1
I2	0.30	1 0	2
I3	0.15	1 1 0	3
I4	0.05	1 1 1 0 0	5
I5	0.04	1 1 1 0 1	5
I6	0.03	1 1 1 1 0	5
I7	0.03	1 1 1 1 1	5

- Average code length: 2.2
- Information redundancy: $(2.20 - 2.17) / 2.20 \approx 1.36 \%$
- Problem: The opcode is very irregular, neither easy to decode nor practical.



Example: Optimized representation of opcode

- [Method 3] Extended compression coding: It is a coding method between fixed-length binary coding and Huffman compression coding. The operation code is not fixed-length, but there are only a few.

Instruction	Frequency	Huffman Code	op Length	Extended Compression Code	op Length
I1	0.40	0	1	0 0	2
I2	0.30	1 0	2	0 1	2
I3	0.15	1 1 0	3	1 0	2
I4	0.05	1 1 1 0 0	5	1 1 0 0	4
I5	0.04	1 1 1 0 1	5	1 1 0 1	4
I6	0.03	1 1 1 1 0	5	1 1 1 0	4
I7	0.03	1 1 1 1 1	5	1 1 1 1	4

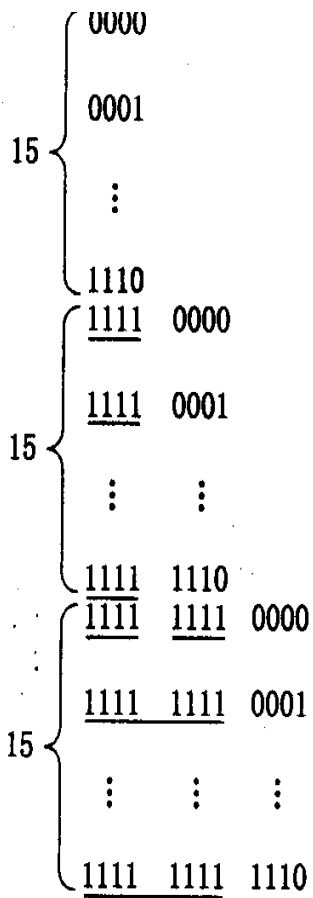
- Average code length: 2.30
- Information redundancy: $(2.30 - 2.17) / 2.30 \approx 5.65 \%$



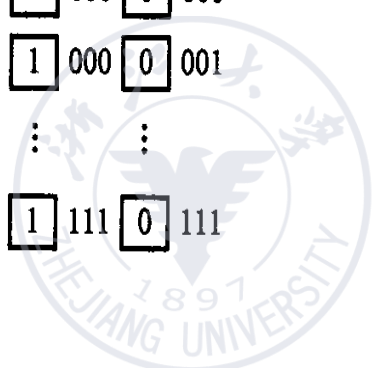
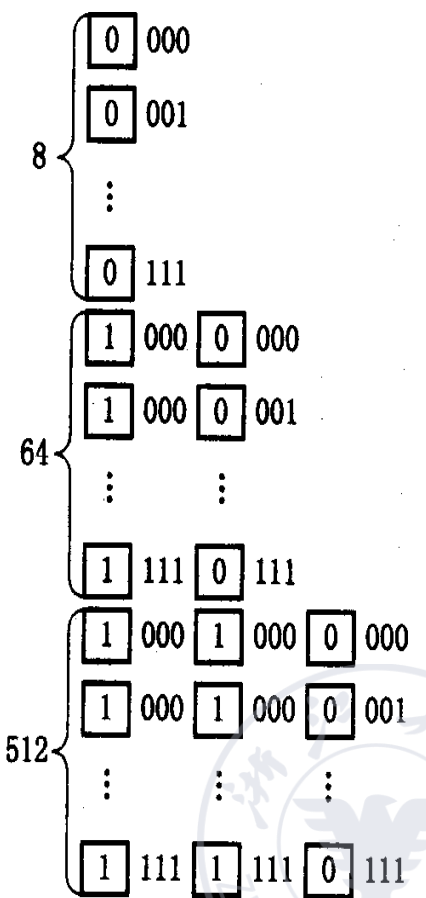
Example: Optimized representation of opcode

- [Method 3] Extension method: generally use equal length extension, such as 3-6-9, 4-8-12 extension. For example, there are two ways to extend 4-8-12:
- Expand by code: set aside one code point for each layer for expansion, such as 15/15/15/.....
- Bitwise expansion: each layer sets aside half of the code points for expansion, such as 8/64/512/.....

15/15/15 Encoding



8/64/512 Encoding



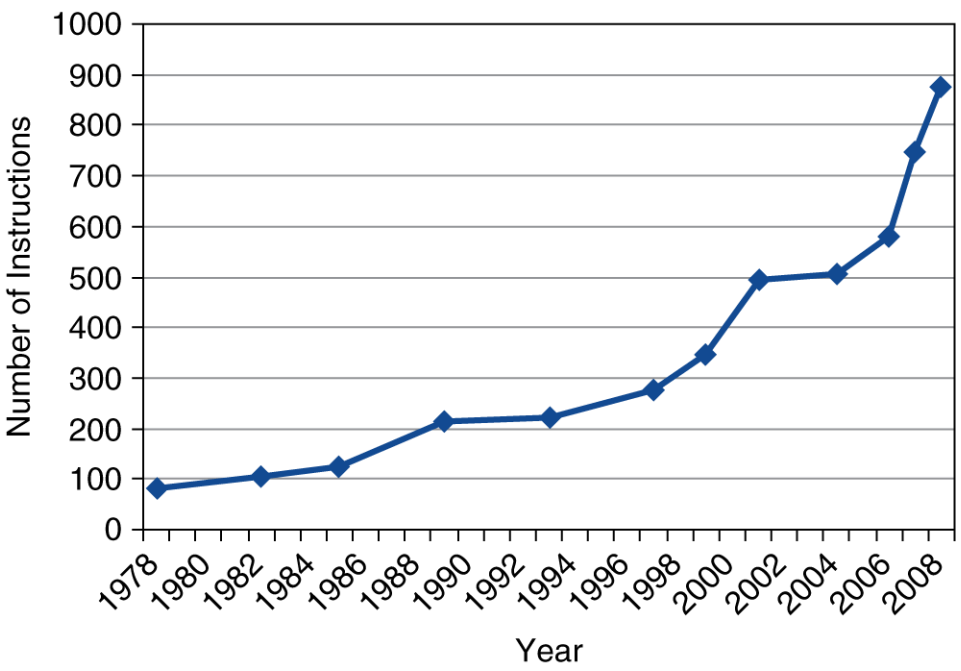
Fallacies

- Powerful instruction \Rightarrow 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 \Rightarrow more errors and less productivity



Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set



Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, 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. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- RISC-V: open source RISC ISAs

