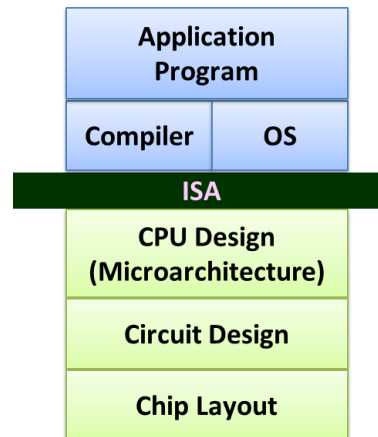


# The HW/SW Interface

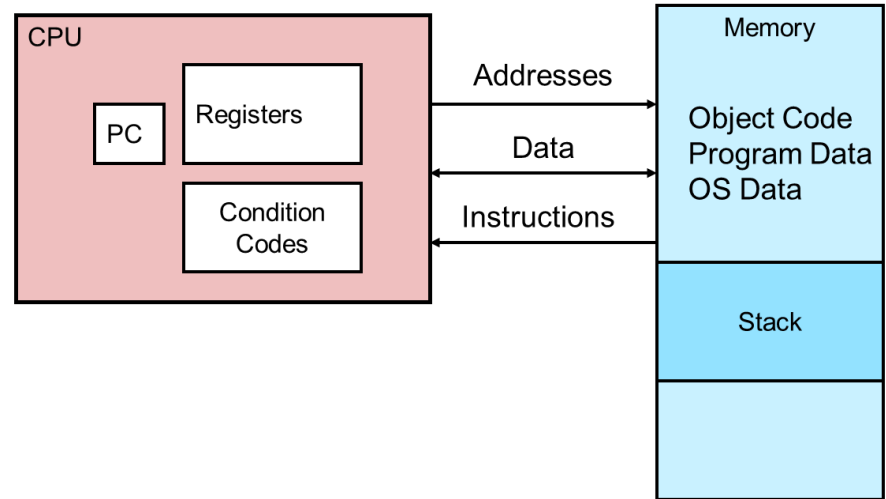
## RISC-V

### Machine-Level Programming Basics



# Module Outline

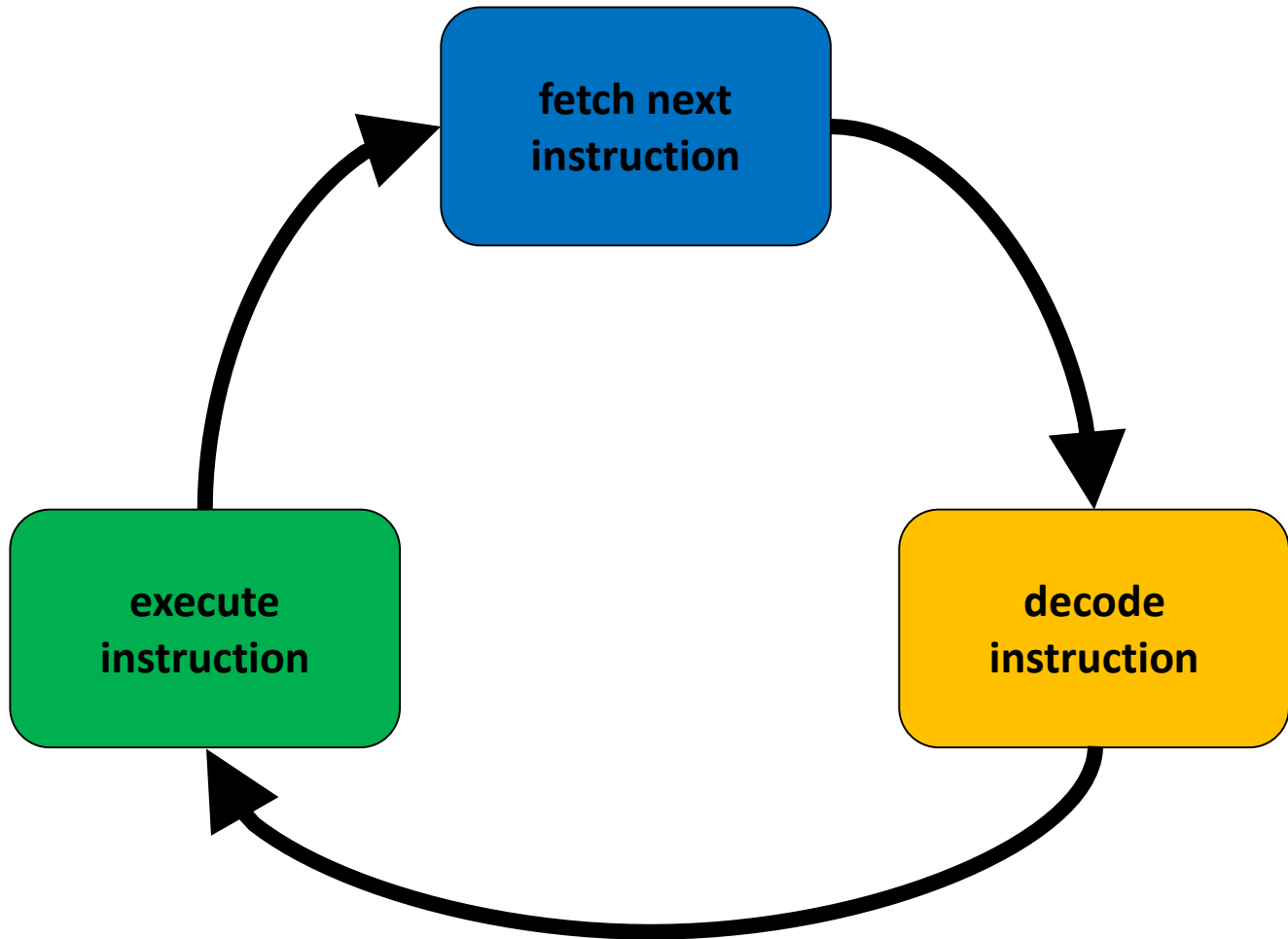
- **The Hardware-Software Interface**
- **Instruction Set Architecture: The Programmer's View**
- **The RISC-V ISA**
  - **Overview**
  - **First Steps**
  - **Arithmetic and Logical Operations**
  - **Data Transfer Operations**
- **Module Summary**



# The Hardware-Software Interface

# Life as a Processor

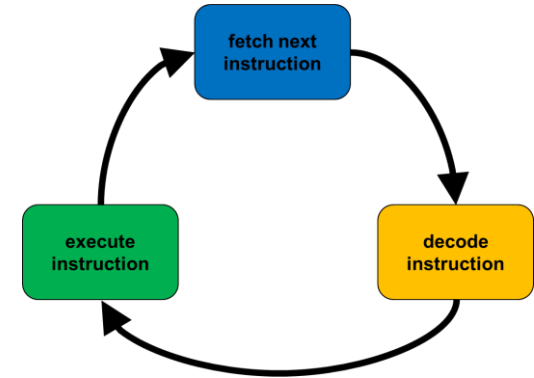
- As long as there is power

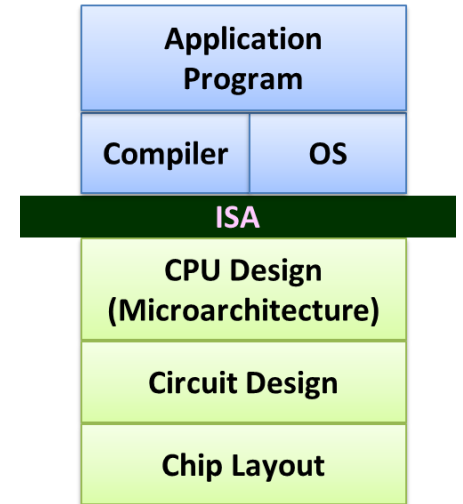


# Life as a Processor

## ■ Lots of interesting problems to solve

- fetch next instruction
  - ▶ “next instruction”
  - ▶ fetch from where
- decode instruction
  - ▶ instruction format
  - ▶ supported operations
  - ▶ supported operands
  - ▶ where do operands come from/go to?
- execute instruction
  - ▶ implementation





# Instruction Set Architecture: The Programmer's View

# Instruction Set Architecture (ISA)

**“the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation”**

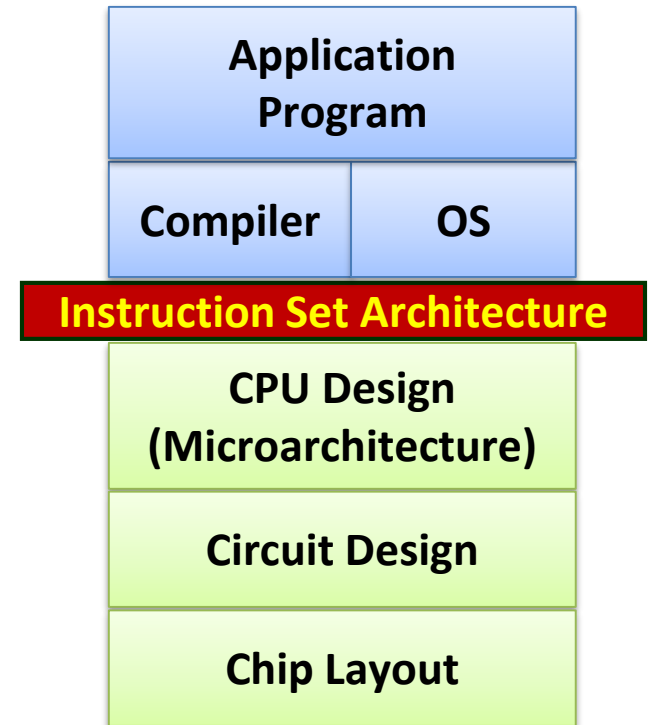
*-- Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360,  
IBM Journal of Research and Development, April 1964.*

- The visible interface between software and hardware
- What the user (OS, compiler, programmer, ...) needs to know in order to reason about how the machine behaves
- Abstracted from the details of how it may accomplish its task

# Instruction Set Architecture (ISA)

## ■ The ISA abstracts the details of the processor implementation

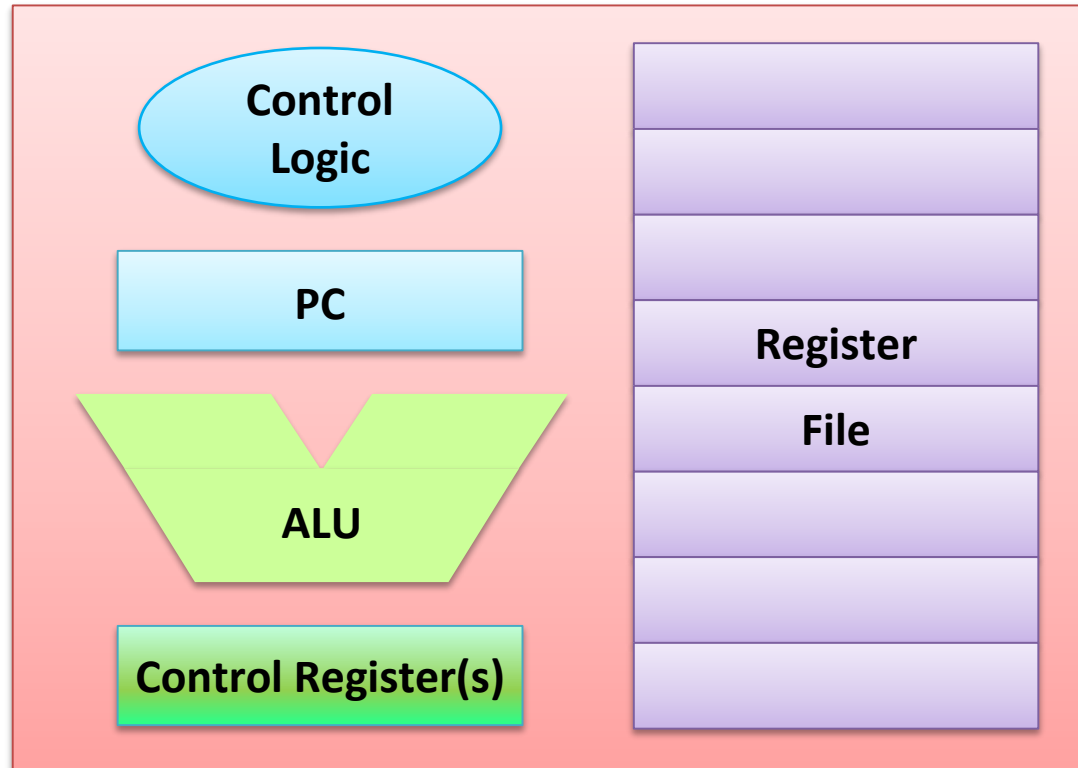
- Above: how to program machine
  - ▶ Processors execute instructions in sequence
- ISA: hardware/software interface
- Below: what needs to be built
  - ▶ Use variety of tricks to make it run fast





# Instruction Set Architecture (ISA)

- The ISA defines the programmer-visible state of a processor architecture



- the ISA tells us what we can expect the processor to do when we execute instructions (but not *how* the processor does it)

# Instruction Set Architecture (ISA)

- The ISA defines the programmer-visible state of a processor architecture
  - Memory addressing
    - ▶ interpretation of addresses
      - bytes, half words, words, double words, ...?
    - ▶ byte ordering in memory
      - little-endian vs. big-endian
    - ▶ addressing modes
      - how to access objects in memory

# Instruction Set Architecture (ISA)

## ■ The ISA defines the programmer-visible state of a processor architecture

### ● Type and size of operands

- ▶ specifying types
  - byte, half word, word, float, double...

### ● Operations

- ▶ arithmetic and logical
- ▶ data transfer
- ▶ control
- ▶ system
- ▶ floating point
- ▶ string
- ▶ multimedia

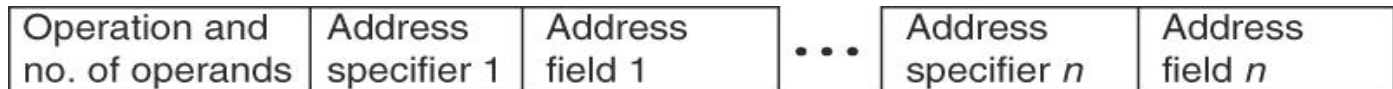
Rank	x86 operation	%
1	load	22
2	conditional branch	20
3	compare	16
4	store	12
5	add	8
6	and	6
7	sub	5
8	move reg-reg	4
9	call	1
10	return	1
	<b>total</b>	<b>96</b>

# Instruction Set Architecture (ISA)

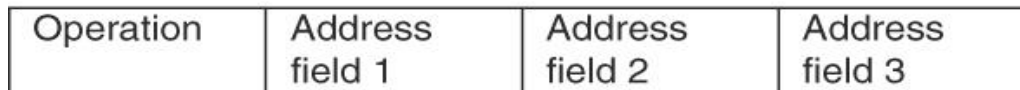
- The ISA defines the programmer-visible state of a processor architecture

- Instruction encoding

- ▶ defines binary representation for each operation and operand
- ▶ fixed size vs. variable size



(a) Variable (e.g., Intel 80x86, VAX)

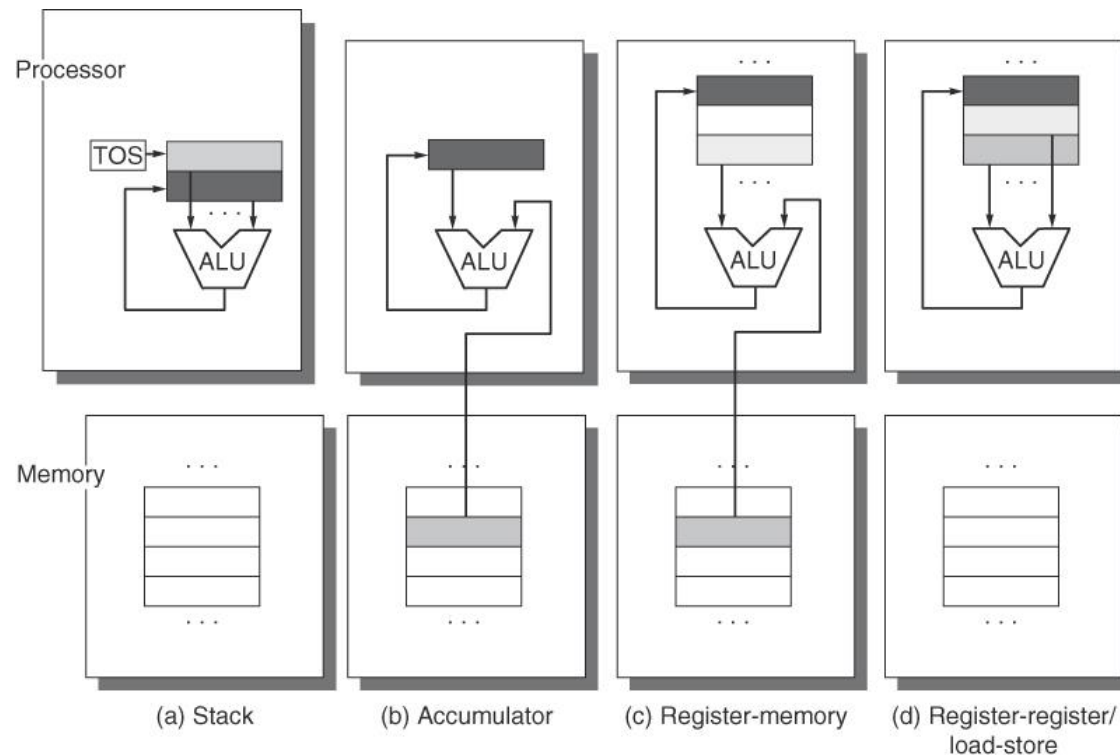
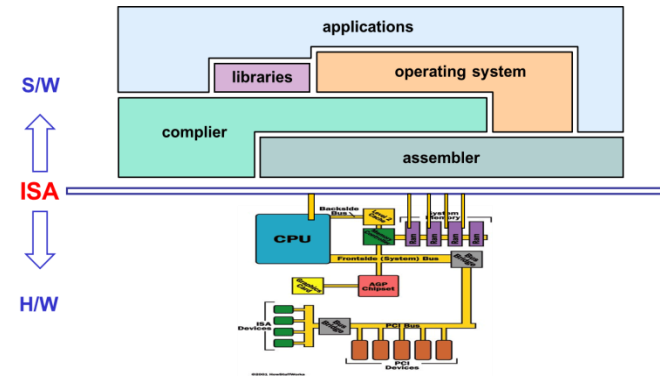


(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

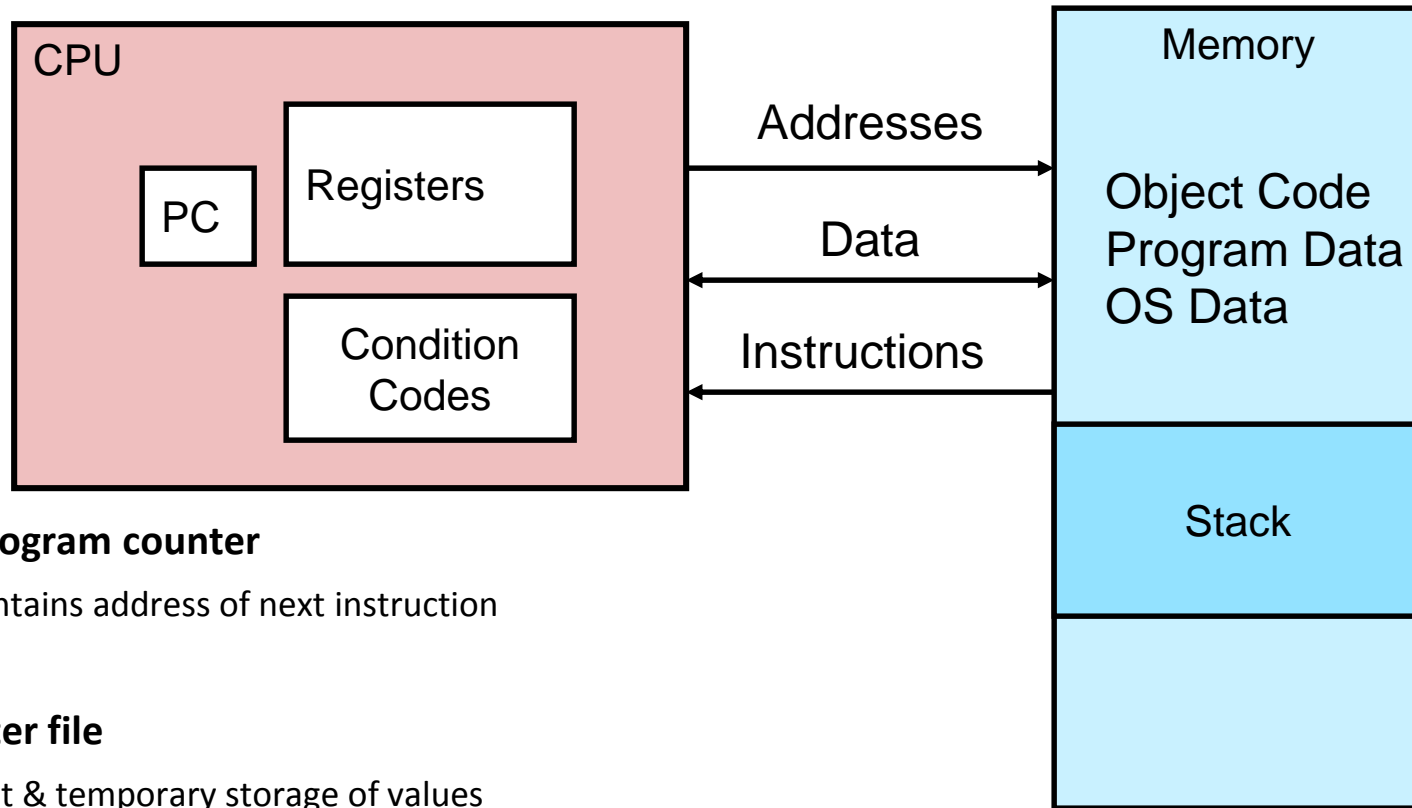
# Instruction Set Architecture (ISA)

## ISA classification

- RISC vs CISC
- general-purpose
  - ▶ two or three operands?
  - ▶ # of memory operands?
  - ▶ classification
    - load-store
    - register-memory
    - memory-memory
- stack
- accumulator



# Assembly Programmer's View



## ■ PC: Program counter

- Contains address of next instruction

## ■ Register file

- Fast & temporary storage of values

## ■ Condition codes

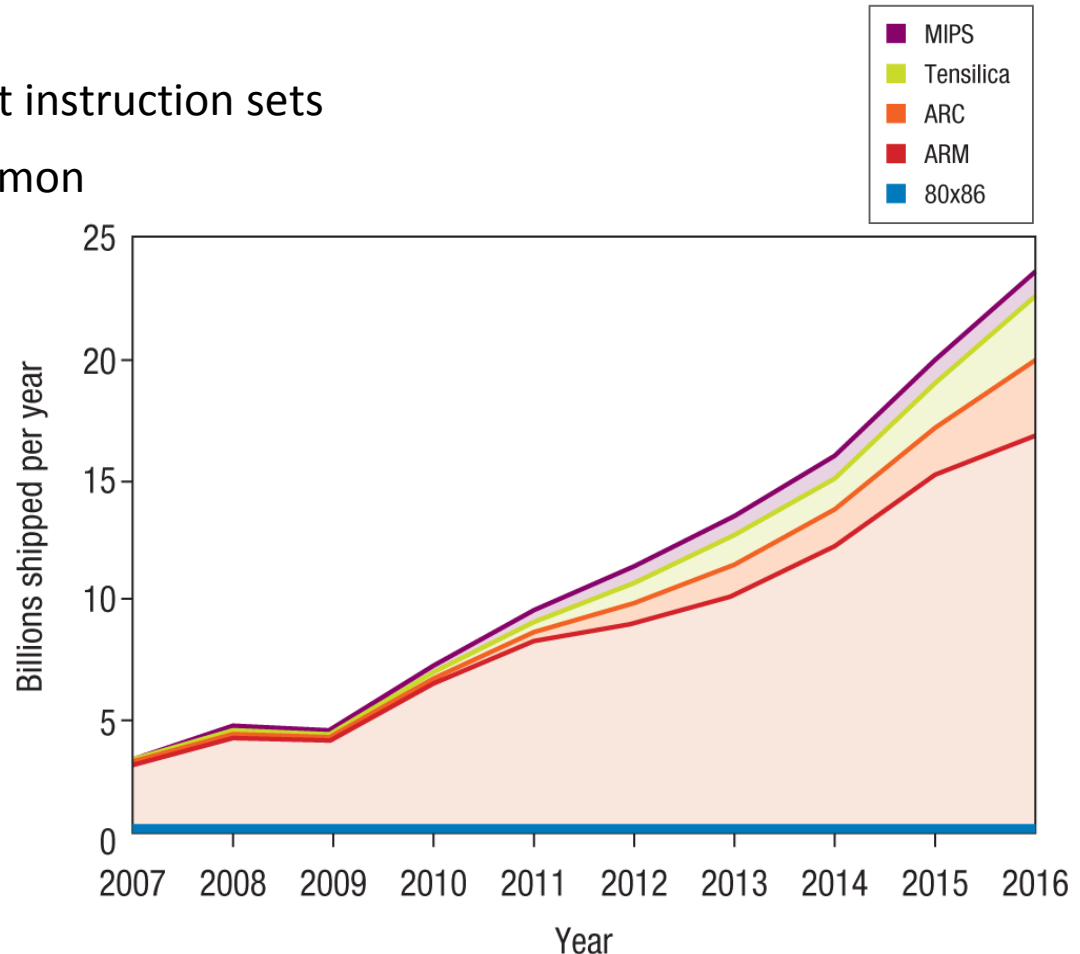
- Status information about most recent arithmetic operation
- Used for conditional branching

## ■ Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack to support procedures

# The 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
  - RISC (Reduced Instruction Set Computer)
- The Intel Architecture is an entirely different beast



Source: David Patterson, "Reduced Instruction Set Computers Then and Now," IEEE Computer, 2017.

# The Instruction Set

- ARM processors (ARMv8-A A64)
  - basic instructions: ~300
  - with bells and whistles: ~1300
  - [ARMv8-A A64 instruction set](#)
  
- RISC-V
  - base integer instruction set: 47 instructions
  - with bells and whistles: <200
  - [RISC-V ISA specification](#)

Source: David Patterson, "Reduced Instruction Set Computers Then and Now," IEEE Computer, 2017.



# The Instruction Set

## ■ Intel x86\_64

- basic instructions: ~300
- with bells and whistles: ~4000  
(includes different operand types)
  - ▶ Multimedia Extensions (MMX): 47 instructions
  - ▶ Streaming SIMD Extensions (SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2)  
About  $70 + 145 + 15 + 30 + 50 = 310$  new instructions
  - ▶ Advanced Vector Extensions (AVX)
    - AVX: ~100 instructions
    - AVX2: ~250 instructions
    - AVX-512: ~2,000 new instructions
- [Intel Architecture Reference Manual](#)

Source: David Patterson, "Reduced Instruction Set Computers Then and Now," IEEE Computer, 2017.



# The RISC-V Instruction Set Architecture

## Overview

# The RISC-V Instruction Set

- A completely open ISA that is freely available to academia and industry
- Fifth RISC ISA design developed at UC Berkeley
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), and RISC-V (2010)
- Now managed by the RISC-V Foundation (<https://riscv.org/>)
- Typical of many modern ISAs
  - See [RISC-V Reference Card](#) in textbook
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, ...

# Why Open the ISA at no Cost?

- Greater innovation via free-market competition
  - From many core designers, closed-source and open-source
- Shared open core designs
  - Shorter time to market, lower cost from reuse, fewer errors given more eyeballs, transparency makes it difficult for government agencies to add secret trap doors
- Processors becoming affordable for more devices
  - Help expand the Internet of Things (IoT), which could cost as little as \$1
- Software stack survive for long time
- Make architectural research and education more real
  - Fully open hardware and software stacks

Source: Krste Asanovic, RISC-V Tutorial at HPCA, 2015.

# RISC-V ISAs

## ■ Three base integer ISAs, one per address width

- RV32I, RV64I, RV128I
- RV32I: Only 40 instructions defined
- RV32E: Reduced version of RV32I with 16 registers for embedded systems

## ■ Standard extensions

- Standard RISC encoding in a fixed 32-bit instruction format
- C extension offers shorter 16-bit versions of common 32-bit RISC-V instructions (can be intermixed with 32-bit instructions)

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

# RISC-V Registers (32 registers)

#	Name	Usage
x0	zero	Hard-wired zero ( <i>always zero</i> )
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers) <i>free to use</i>
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	
x13	a3	
x14	a4	
x15	a5	

*function arguments*

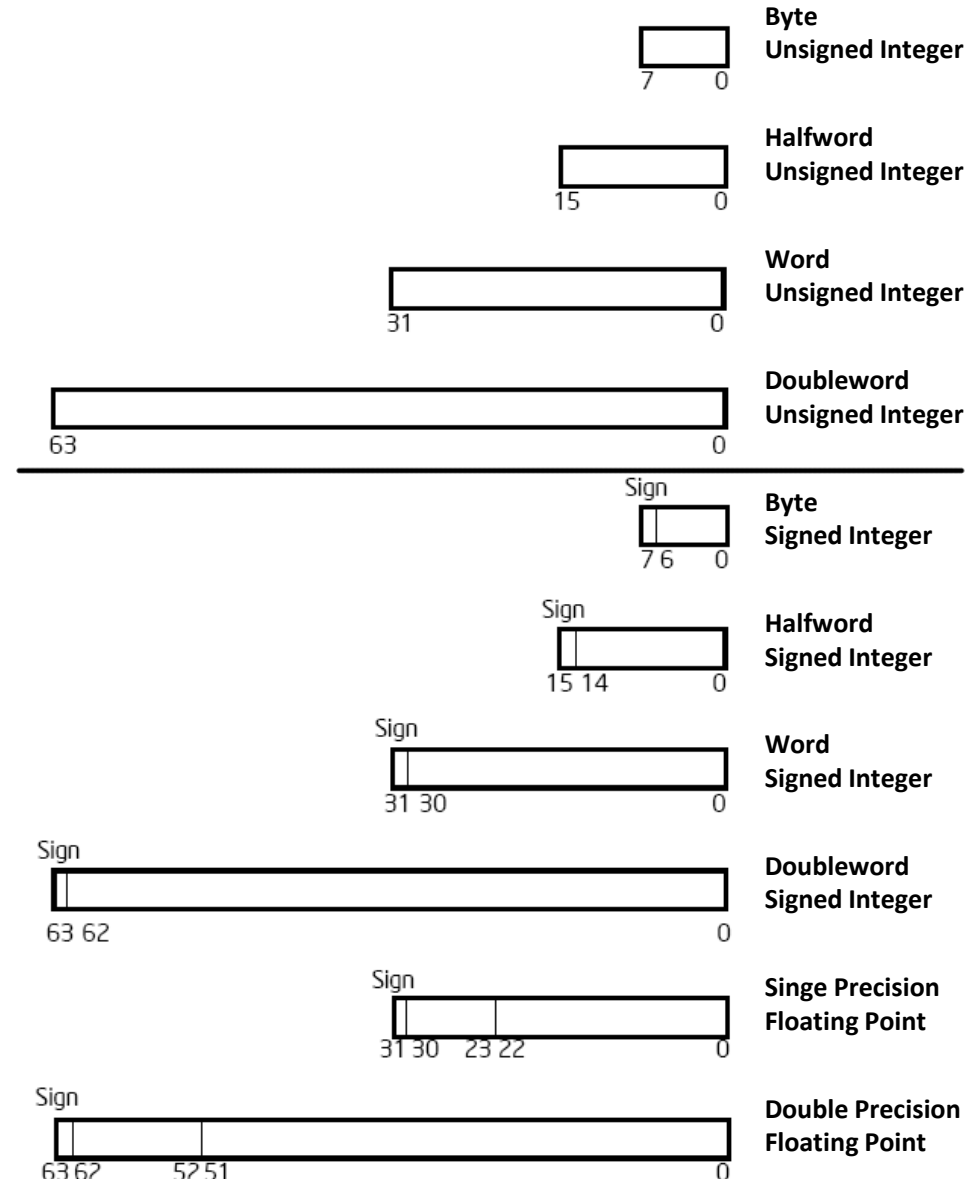
*like %RAX, %RDI of intel*

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
pc		Program counter

*address of instruction (ip)*

# Data Types

- Integer data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4 or 8 bytes (with F or D extension)
- No aggregated types such as arrays or structures
  - Just contiguously allocated bytes in memory

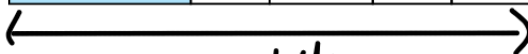
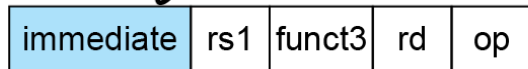


# Operand Types: RISC-V Addressing

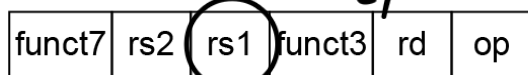
*directly moved to*

$x = \underline{78}$  (literal)

## 1. Immediate addressing

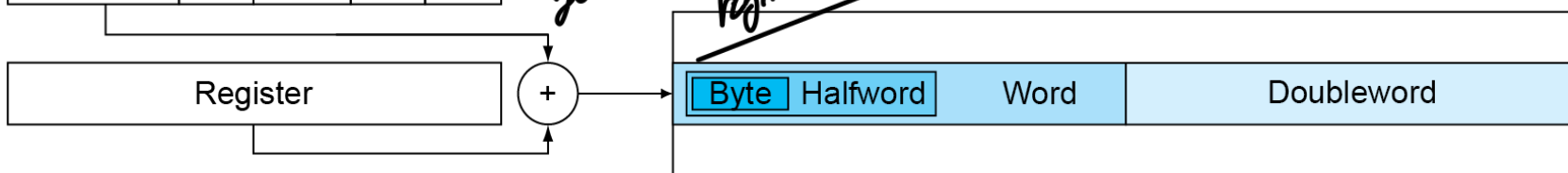
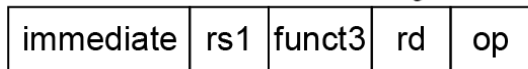


## 2. Register addressing (*pairing register*)

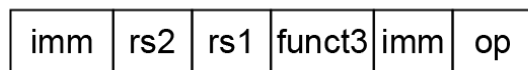


Size?   
 32 → 5bits   
 64 → 6bits

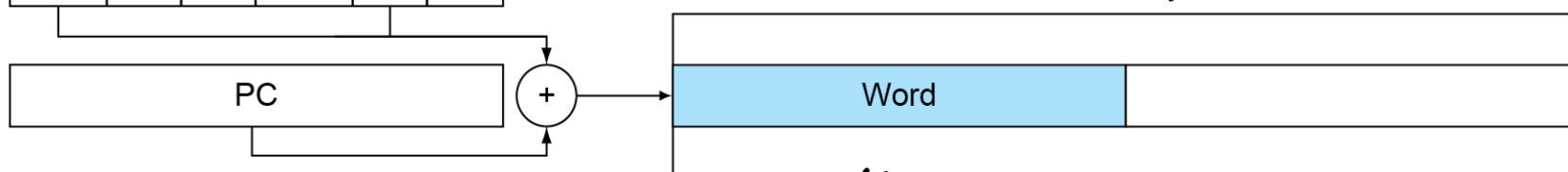
## 3. Base addressing (*memory*)



## 4. PC-relative addressing



*relative address from PC's address*



*intel using PC-relative addressing*



# Operations

- Perform an arithmetic or logical function on register data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jump
  - Conditional branch
  - Procedure call and return (*function*)

# Comparison

- ARMv8 (AArch64)
  - 32 general purpose registers
  - 1 reserved, 1 zero
- Intel IA32 (32-bit)
  - 8 general purpose registers
  - 2 of which have predetermined role
- Intel x86\_64 (64-bit)
  - 16 general purpose registers
  - 1 of which is reserved

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```



```
sum:
    add    a0,a0,a1
    ret
```

# The RISC-V Instruction Set Architecture

## First Steps

# From C to Machine Code

*gcc -v (#Path needed)*

Prepend all commands with  
risc64-unknown-elf-  
in the VM (cross-compiler)!

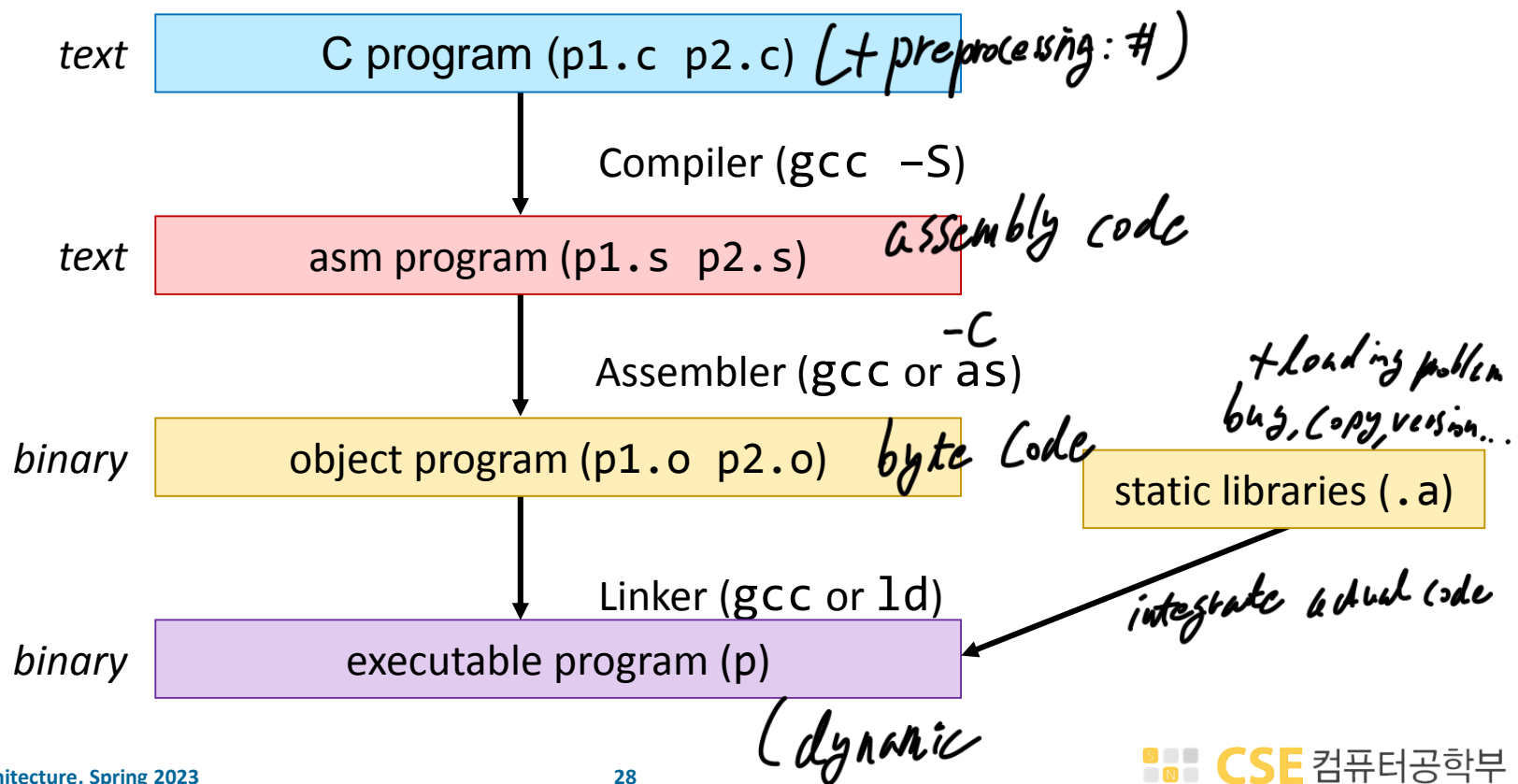
■ Code in files **p1.c** **p2.c**

■ Compile with command: **gcc -O p1.c p2.c -o p**

- Use optimizations (**-O**)

- Put resulting binary in file **p**

*gcc is not compiler*



# Compiling to 32-bit RISC-V

-O3 ↓ level of optimization  
-O2

C Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

*risc64-unknown-elf-objdump for read assembly*

\$ risc64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O -S sum.c  
produces the file sum.s:

sum:  
add a0,a0,a1  
ret

*not optimized*

→ code generated by gcc -march=rv32i -mabi=ilp32 (-O0) -S sum.c ?

# Compiling to 64-bit RISC-V

## C Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

\$ risc64-unknown-elf-gcc -march=rv64i -mabi=lp64 -O -S sum.c  
produces the file sum.s:

sum:

addw

ret

$x \rightarrow t$

(a0) a0, a1

# Compiling to 64-bit Intel (x86\_64)

C Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

\$ gcc -march=x86-64 -O -S sum.c

produces the file sum.s:

```
sum:
.LFB0:
    .cfi_startproc
    leal    (%rdi,%rsi), %eax
    ret
    .cfi_endproc
```

→ code generated by gcc -march=x86-64 -O0 -S sum.c ?

# Compiling to 32-bit Intel (i386)

## C Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

\$ gcc -m32 -march=i386 -O -S sum.c

produces the file sum.s:

```
sum:
.LFB0:
    .cfi_startproc
    movl    8(%esp), %eax
    addl    4(%esp), %eax
    ret
    .cfi_endproc
```



# Object Code

## Machine code for sum (32-bit RISC-V)

0x1014c <sum>:

0x00

0xb5

0x05

0x33

0x00

0x00

0x80

0x67

- 8 bytes total
- Each instruction 4 bytes
- Starts at address 0x1014c

### ■ Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

### ■ Linker

- Resolves references between files
- Combines with static run-time libraries
  - ▶ E.g., code for malloc, printf
- Some libraries are *dynamically linked*
  - ▶ Linking occurs when program begins execution

# Object Code

## 64-bit RISC-V

0x10154 <sum>:

0x00  
0xb5  
0x05  
0x3b  
0x00  
0x00  
0x80  
0x67

- 8 bytes total
- Each instruction 4 bytes
- Starts at address 0x1014c

*fixed*

## 64-bit Intel

0x113b <sum>:

0x8d  
0x04  
0x37  
0xc3  
0x90

- 5 bytes total
- Three instructions with 3, 1, 1 bytes
- Starts at address 0x113b

*Variable*

# Machine Instruction Example – RISC-V

```
int t = x+y;
```

## ■ C Code

- add two signed integers

*dest src1 src2*  
add     a0, a0, a1

## ■ Assembly

- add computes the sum of two registers and places the result in another register

- Operands:

x:    register        a0

y:    register        a1

t:    register        a0  
(return value also in a0)

```
0x1014c: 00b50533
```

## ■ Object Code

- 4-byte instruction
- Stored at address **0x1014c**

# Machine Instruction Example – IA32

```
int t = x+y;
```

*src*  
*%ebp+8*     *dest*

```
addl 8(%ebp),%eax
```

Intel code similar to expression:

$x += y$      *x=, /=, -=*  
                 *also support*

More precisely:

```
int eax;
```

```
int *ebp;
```

```
eax += mem[ebp + 8]
```

```
0x0804809a: 03 45 08
```

## ■ C Code

- add two signed integers

## ■ Assembly

- `addl` adds the 4-byte value located at memory address `%ebp+8` to register `%eax`
- Operands:
  - x:** register     **%eax**
  - y:** memory     **mem[%ebp+8]**
  - t:** register     **%eax**  
(return value also in %eax)

## ■ Object Code

- 3-byte instruction
- Stored at address **0x0804809a**

# Disassembling Object Code

```
$ riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O -o p main.c sum.c  
$ riscv64-unknown-elf-objdump -d p32
```

```
0001014c <sum>:  
    1014c:    00b50533      add    a0,a0,a1  
    10150:    00008067      ret
```

- Disassembler (objdump)
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces *approximate* rendition of assembly code
  - Can be run on either a.out (complete executable) or .o file

# Disassembling Object Code – IA32

```
$ gcc -m32 -c -O sum.c  
$ objdump -d sum.o
```

```
00000000 <sum>:  
  0:  55          push    %ebp  
  1:  89 e5       mov     %esp,%ebp  
  3:  8b 45 0c    mov     0xc(%ebp),%eax  
  6:  03 45 08    add     0x8(%ebp),%eax  
  9:  5d          pop     %ebp  
 a:  c3          ret
```

# Alternate Disassembly using gdb

- Within **gdb** Debugger

```
$ gcc -m32 -O -o p main.c sum.c
```

```
$ riscv64-unknown-elf-gdb p32
```

```
GNU gdb (GDB) 8.3.0.20190516-git
...
Reading symbols from p32...
(No debugging symbols found in p32)
(gdb) disassemble sum
Dump of assembler code for function sum:
    0x0001014c <+0>:      add      a0,a0,a1
    0x00010150 <+4>:      ret
End of assembler dump.
(gdb) x/8xb sum
0x1014c <sum>:  0x33 0x05 0xb5 0x00 0x67 0x80 0x00 0x00
(gdb)
```

# What Can be Disassembled?

```
% objdump -d WINWORD.EXE

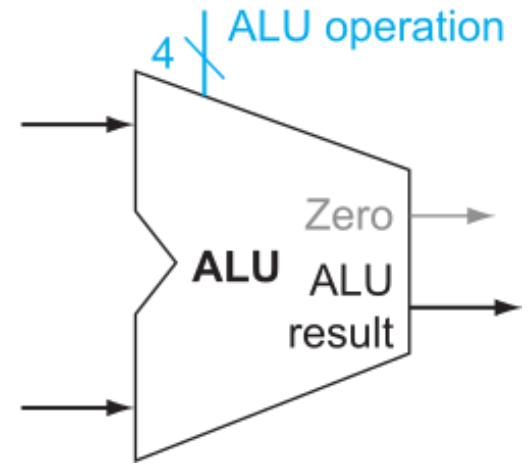
WINWORD.EXE:  file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                push    %ebp
30001001:  8b ec            mov     %esp,%ebp
30001003:  6a ff            push    $0xffffffff
30001005:  68 90 10 00 30    push    $0x30001090
3000100a:  68 91 dc 4c 30    push    $0x304cdc91
```

- (Almost) anything that can be interpreted as executable code (not everything makes sense, of course)
- Disassembler examines bytes and reconstructs assembly source





# RISC-V

## Arithmetic and Logical Operations

# Arithmetic Operations

- Add and subtract: three operands

- Two sources and one destination

*dest s1 s2*

add a, b, c	// $a \leftarrow b + c$
sub a, b, c	// $a \leftarrow b - c$

- All arithmetic operations have this form

- Design Principle 1: Simplicity favors regularity

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

# Register Operands

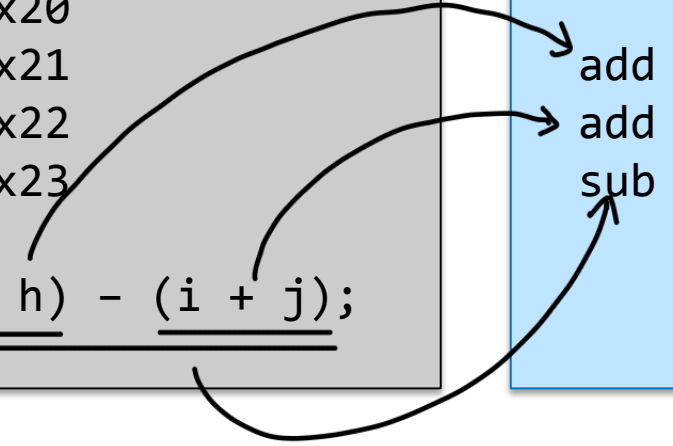
- Arithmetic instructions use register operands
- RISC-V has a 32 x 64-bit register file:  $x0 \sim x31$ 
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
  - 32-bit data is called a “word”
- Design Principle 2: Smaller is faster
  - cf. Main memory: millions of locations

# Register Operand Example

C code:

```
// f in x19
// g in x20
// h in x21
// i in x22
// j in x23

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



Compiled RISC-V code:

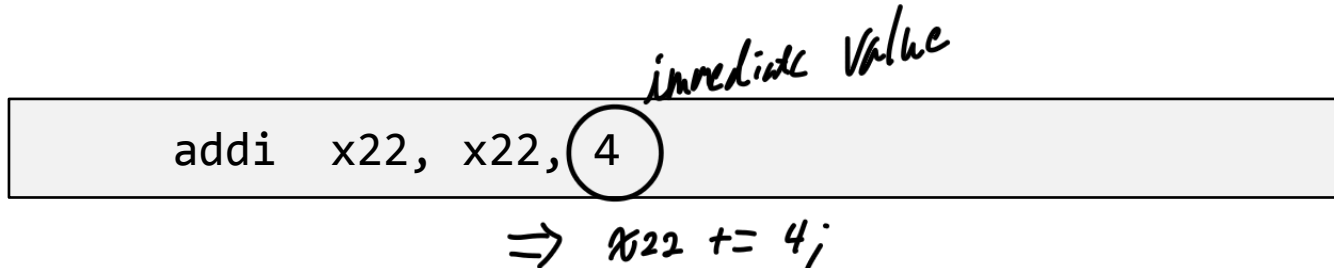
```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

# Registers vs. Memory

- Registers are faster to access than memory
- In RISC-V, data in memory cannot be directly addressed by ALU instructions
- Operating on memory data <sup>Lw</sup> <sup>Sw</sup> requires loads and stores
  - More instructions to be executed
- Compiler tries to use registers for variables as much as possible (cache)
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
- Good to know: the Intel ISA allows one memory operand (src1/dest, or src2)
  - $\text{mem}[\text{address}] += \text{reg}$
  - $\text{reg} += \text{mem}[\text{address}]$

# Immediate Operands

- Constant data specified in an instruction



- Make the common case fast
  - Small constants are common (limited to 12 bits, signed)
  - Immediate operand avoids a load instruction

- Good to know: the Intel x86\_64 ISA allows up to 8 byte (integer) constants

```
movabsq $0x123456789abcdef0, %rax
```

# Arithmetic Operations

Instruction	Type	Example	Meaning
Add	R	add rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
Subtract	R	sub rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
Add immediate	I	addi rd, rs1, imm12	$R[rd] = R[rs1] + \text{SignExt}(imm12)$
Set less than	R	slt rd, rs1, rs2	$R[rd] = (R[rs1] < R[rs2])? 1 : 0$
Set less than immediate	I	slti rd, rs1, imm12	$R[rd] = (R[rs1] < \text{SignExt}(imm12))? 1 : 0$
Set less than unsigned	R	sltu rd, rs1, rs2	$R[rd] = (R[rs1] <_u R[rs2])? 1 : 0$
Set less than immediate unsigned	I	sltiu rd, rs1, imm12	$R[rd] = (R[rs1] <_u \text{SignExt}(imm12))? 1 : 0$
Load upper immediate	U	lui rd, imm20	$R[rd] = \text{SignExt}(imm20 \ll 12)$
Add upper immediate to PC	U	auipc rd, imm20	$R[rd] = PC + \text{SignExt}(imm20 \ll 12)$

R: register

I: immediate

U: ?

16 bit

0x12345	0x000
---------	-------

여기에 넣는다

ex) lui 0x6 0x12345

# Example: arith

x in a0  
y in a1  
z in a2

```
long arith (long x,  
            long y,  
            long z) {  
    long t1 = x + y;  
    long t2 = z + t1;  
    long t3 = x + 4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rv = t2 - t5;  
    return rv;  
}
```

arith:

add	a5, a0, a1	# a5 = x + y (t1)
add	a2, a5, a2	# a2 = t1 + z (t2)
addi	a0, a0, 4	# a0 = x + 4 (t3)
slli	a5, a1, 1	# a5 = y * 2
add	a1, a5, a1	# a1 = a5 + y ) y * 3
slli	a5, a1, 4	# a5 = a1 * 16 (t4)
add	a0, a0, a5	# a0 = t3 + t4 (t5)
sub	a0, a2, a0	# a0 = t2 - t5 (rval)
ret		

shift left

no-longer use x, y

no-longer use x

even though using 3 instructions,  
more efficient (mul usually spend 3-5 cycles)



# Logical Operations

## ■ Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	sll, slli
Shift right (arithmetic)	>>	>>	sra, srai
Shift right (logical)	>>	>>>	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

# AND Operations

- Useful to mask bits in a word
  - Select some bits, clears 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 Operation

- 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 Operation

- Differencing operation
  - Clear when bits are the same, set if they are different

```
xor x9, x10, x12
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
-----	----------	----------	----------	----------	----------	----------	----------	----------

x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
-----	----------	----------	----------	----------	----------	----------	----------	----------

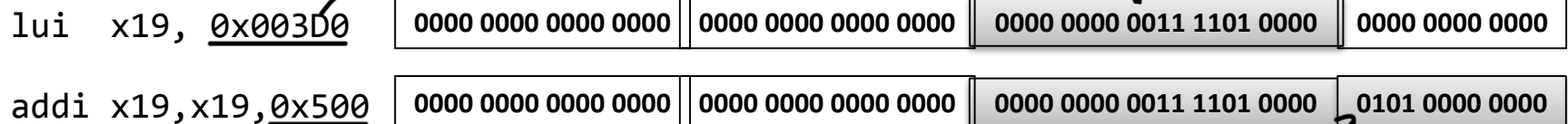
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111
----	----------	----------	----------	----------	----------	----------	----------	----------

# 32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant:
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

```
lui rd, constant
```

- Example:  $x19 \leftarrow 0x003D0500$



# Logical Operations

Instruction	Type	Example	Meaning
AND	R	and rd, rs1, rs2	$R[rd] = R[rs1] \& R[rs2]$
OR	R	or rd, rs1, rs2	$R[rd] = R[rs1]   R[rs2]$
XOR	R	xor rd, rs1, rs2	$R[rd] = R[rs1] \wedge R[rs2]$
AND immediate	I	andi rd, rs1, imm12	$R[rd] = R[rs1] \& \text{SignExt}(\text{imm12})$
OR immediate	I	ori rd, rs1, imm12	$R[rd] = R[rs1]   \text{SignExt}(\text{imm12})$
XOR immediate	I	xori rd, rs1, imm12	$R[rd] = R[rs1] \wedge \text{SignExt}(\text{imm12})$
Shift left logical	R	sll rd, rs1, rs2	$R[rd] = R[rs1] \ll R[rs2]$
Shift right logical	R	srl rd, rs1, rs2	$R[rd] = R[rs1] \gg R[rs2]$ (logical)
Shift right arithmetic	R	sra rd, rs1, rs2	$R[rd] = R[rs1] \gg R[rs2]$ (arithmetic)
Shift left logical immediate	I	slli rd, rs1, shamt	$R[rd] = R[rs1] \ll \text{shamt}$
Shift right logical imm.	I	srli rd, rs1, shamt	$R[rd] = R[rs1] \gg \text{shamt}$ (logical)
Shift right arithmetic immediate	I	srai rd, rs1, shamt	$R[rd] = R[rs1] \gg \text{shamt}$ (arithmetic)

# Example: logical

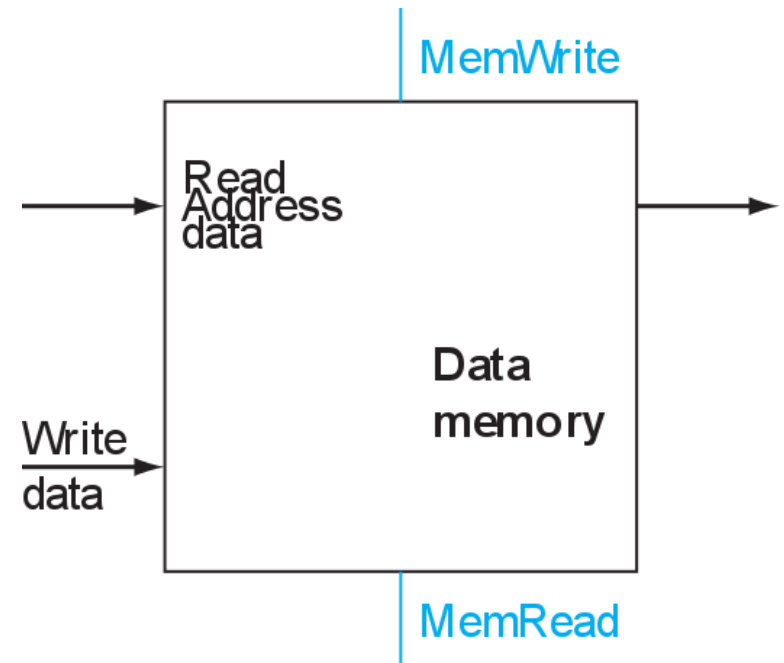
x in a0  
y in a1

```
long logical (long x, long y)
{
    long t1 = x ^ y;
    long t2 = t1 >> 17;
    long mask = (1 << 8) - 7;
    long rval = t2 & mask;
    return rval;
}
```

*constant value  
(calculate in compile time  
= constant propagation)*

logical:

```
xor    a0, a0, a1 # a0 = x ^ y (t1)
srai   a0, a0, 17 # a0 = t1 >> 17 (t2)
> andi a0, a0, 249 # a0 = t2 & ((1 << 8) - 7)
ret
```



# RISC-V

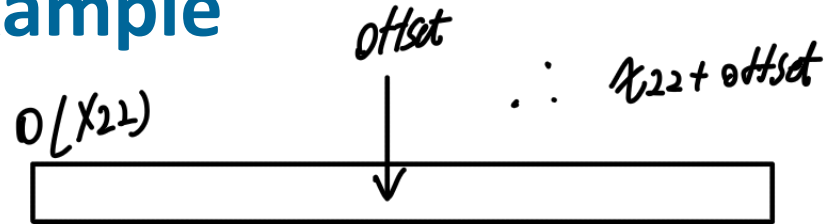
## Data Transfer Operations



# Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory to 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
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

# Memory Operand Example



C code:

```
// h in x21
// base address of A in x22
```

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

Compiled RISC-V code:

```
// 8 bytes per doubleword
```

```
// &A[8] = A + 64
```

```
ld    x9, 64(x22) : Load to x9
add    x9, x21, x9 : add h
sd    x9, 96(x22) : store inside A[12]
```

Handwritten annotations:  $8 \times 8$  (pointing to 64),  $8 \times 12$  (pointing to 96).

*Base-Addressing*

# Byte/Halfword/Word Operations

- Load byte/halfword/word:  
Sign extend to 64 bits in rd

lb	rd, offset(rs1)
lh	rd, offset(rs1)
lw	rd, offset(rs1)

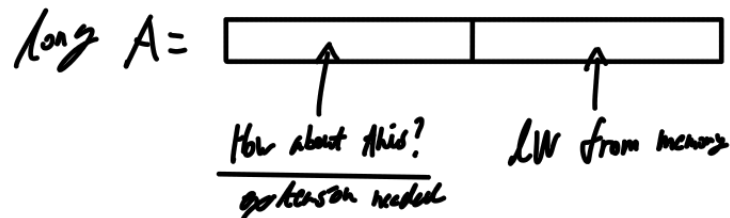
- Load byte/halfword/word:  
Zero extend to 64 bits in rd  
*unsigned load*

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)

*b 1 byte*  
*h 2 byte*  
*w 4 byte*  
*d 8 byte*



# Data Transfer Operations

Instruction	Type	Example	Meaning
Load doubleword	I	ld rd, imm12(rs1)	$R[rd] = \text{Mem}_8[R[rs1] + \text{SignExt}(\text{imm12})]$
Load word	I	lw rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_4[R[rs1] + \text{SignExt}(\text{imm12})])$
Load halfword	I	lh rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_2[R[rs1] + \text{SignExt}(\text{imm12})])$
Load byte	I	lb rd, imm12(rs1)	$R[rd] = \text{SignExt}(\text{Mem}_1[R[rs1] + \text{SignExt}(\text{imm12})])$
Load word unsigned	I	lwu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_4[R[rs1] + \text{SignExt}(\text{imm12})])$
Load halfword unsigned	I	lhu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_2[R[rs1] + \text{SignExt}(\text{imm12})])$
Load byte unsigned	I	lbu rd, imm12(rs1)	$R[rd] = \text{ZeroExt}(\text{Mem}_1[R[rs1] + \text{SignExt}(\text{imm12})])$
Store doubleword	S	sd rs2, imm12(rs1)	$\text{Mem}_8[R[rs1] + \text{SignExt}(\text{imm12})] = R[rs2]$
Store word	S	sw rs2, imm12(rs1)	$\text{Mem}_4[R[rs1] + \text{SignExt}(\text{imm12})] = R[rs2](31:0)$
Store halfword	S	sh rs2, imm12(rs1)	$\text{Mem}_2[R[rs1] + \text{SignExt}(\text{imm12})] = R[rs2](15:0)$
Store byte	S	sb rs2, imm12(rs1)	$\text{Mem}_1[R[rs1] + \text{SignExt}(\text{imm12})] = R[rs2](7:0)$

*doesn't need extension*

# Swap Example

$a0 = *xp$     $a1 = *yp$

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

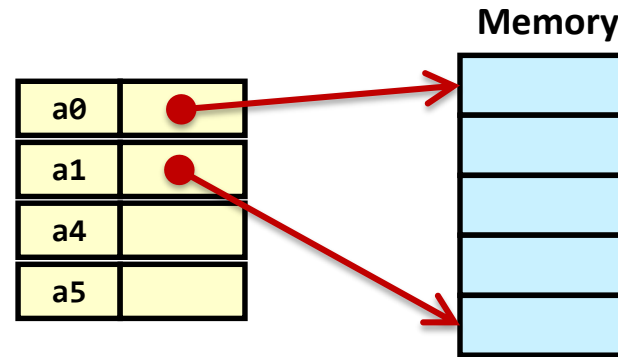
```
swap:
    ld    a4, 0(a0)
    ld    a5, 0(a1)
    sd    a5, 0(a0)
    sd    a4, 0(a1)
    ret
```

get data from memory  
stored in xp, yp address

a0, a1 point  
memory address

# Understanding Swap (1)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

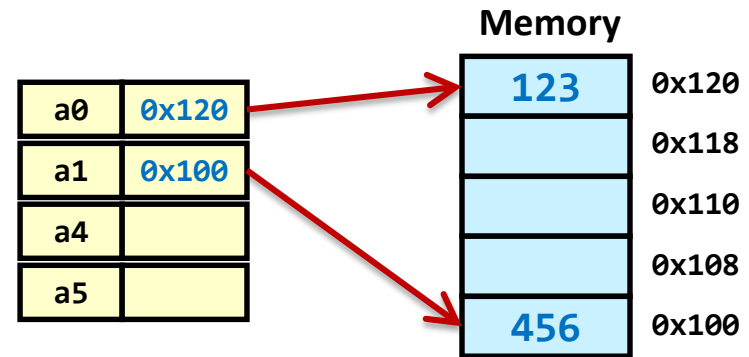
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

swap:

ld	a4, 0(a0)	# t0 = *xp
ld	a5, 0(a1)	# t1 = *yp
sd	a5, 0(a0)	# *xp = t1
sd	a4, 0(a1)	# *yp = t0
ret		

# Understanding Swap (2)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

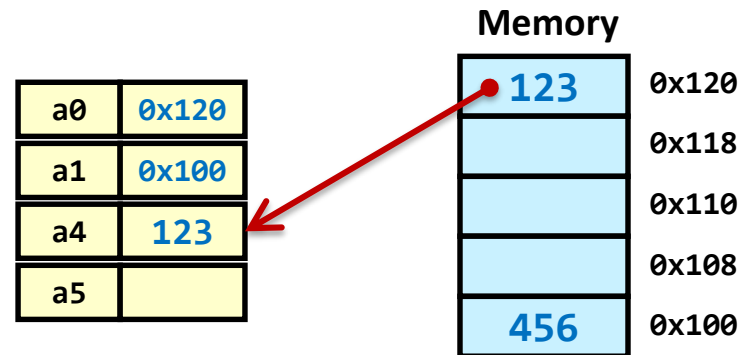
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

swap:

ld	a4, 0(a0)	# t0 = *xp
ld	a5, 0(a1)	# t1 = *yp
sd	a5, 0(a0)	# *xp = t1
sd	a4, 0(a1)	# *yp = t0
ret		

# Understanding Swap (3)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

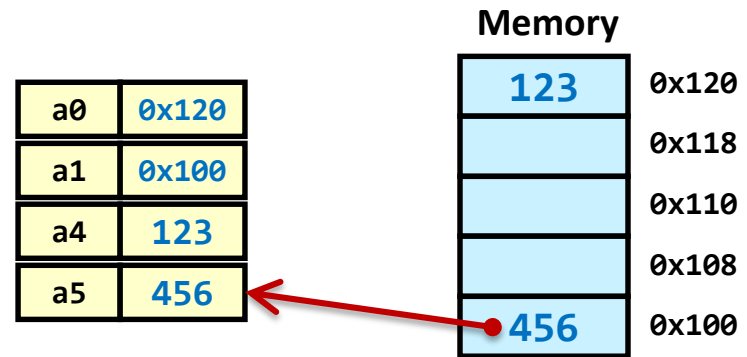
swap:

ld	a4, 0(a0)	# t0 = *xp
ld	a5, 0(a1)	# t1 = *yp
sd	a5, 0(a0)	# *xp = t1
sd	a4, 0(a1)	# *yp = t0
ret		



# Understanding Swap (4)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

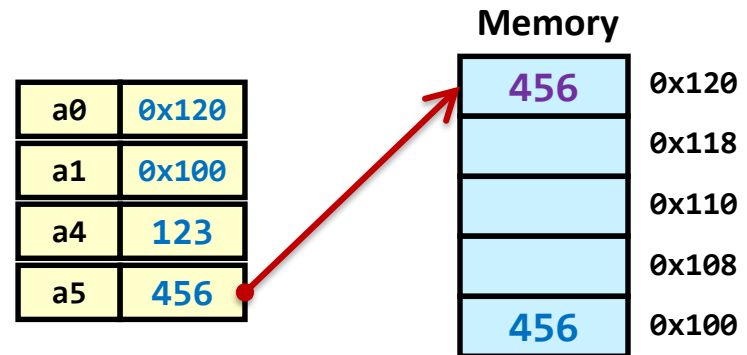
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

swap:

ld	a4, 0(a0)	# t0 = *xp
<b>ld</b>	<b>a5, 0(a1)</b>	<b># t1 = *yp</b>
sd	a5, 0(a0)	# *xp = t1
sd	a4, 0(a1)	# *yp = t0
ret		

# Understanding Swap (5)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

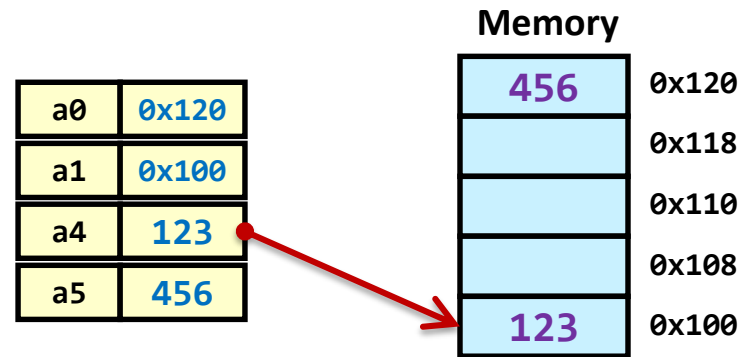
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

swap:

ld	a4, 0(a0)	# t0 = *xp
ld	a5, 0(a1)	# t1 = *yp
sd	a5, 0(a0)	# *xp = t1
sd	a4, 0(a1)	# *yp = t0
ret		

# Understanding Swap (6)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation  
(by compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

swap:

ld	a4, 0(a0)	# t0 = *xp
ld	a5, 0(a1)	# t1 = *yp
sd	a5, 0(a0)	# *xp = t1
<b>sd</b>	<b>a4, 0(a1)</b>	<b># *yp = t0</b>
ret		

# String Copy Example

```
void strcpy(char x[],
            char y[])
{
    size_t i;

    i = 0;
    while
        ((x[i]=y[i])!='\0')
        i += 1;
}
```

strcpy:

; x is in a0  
; y is in a1  
; i is allocated in t3

*&y[0] + t3  
due to char (byte)*

*get by add  
have*

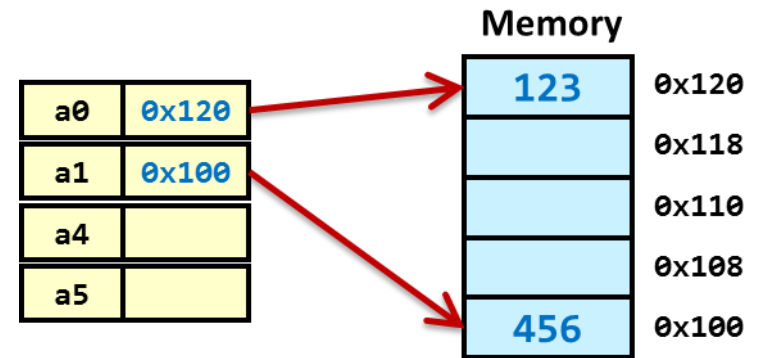
*set x add*

*copy*

*check t1==0*

*repeat*

```
L1: add    t3,zero,zero    ; t3 <- 0
    add    t0,t3,a1        ; t0 <- &y[i]
    lbu     t1,0(t0)        ; t1 <- y[i]
    add     t2,t3,a0        ; t2 <- &x[i]
    sb      t1,0(t2)        ; x[i] <- y[i]
    beq     t1,zero,L2      ; if (t1==0), goto L2
    addi    t3,t3,1         ; t3 += 1
    j       L1              ; goto L1
L2: ret                    ; return
```



# Module Summary

# Module Summary

## ■ Instruction Set Architecture (ISA)

- defines “the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior” of a computer system.
- ISAs and processors across all architectures are similar
  - ▶ program counter, register file, ALU, control logic, control registers
  - ▶ operation types, operands, memory access

## ■ RISC-V ISA

- 32 general-purpose registers (27 can be used more or less freely)
- three operand format: op dest, src1, src2
- arithmetic, logic, data operations (more to follow)

## ■ From C code to assembly

- Know how to produce assembly code and disassemble machine code!