Computer Systems II

Li Lu (卢立)

Room 319, Yifu Business and Management Building

Yuquan Campus

li.lu@zju.edu.cn

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

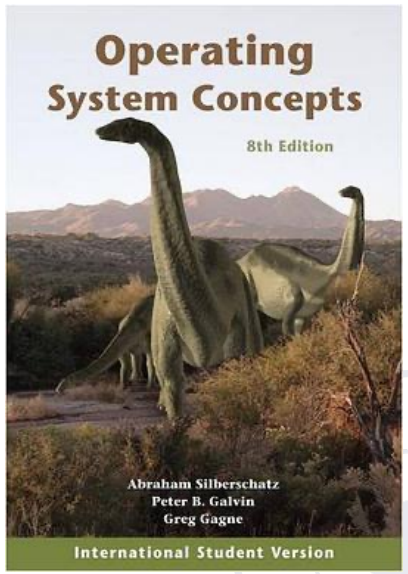
About the Class

Talk about this course

- What have you learned from the previous course?
  - Digital Logic, basic instruction set design, CPU design (mainly on single-cycle CPU), etc.
- What will be covered in this course and what you can get from this course?
  - Move forward one step to learn more complex CPU design
  - Begin to explore the principle of Operating Systems
  - Know not only what by also why

How to Prepare for the Class

- Textbook (Computer Organization and Design: The Hardware/Software Interface RISC-V Edition; Operating System Concepts)
- References
- Teaching Components
  - Lectures
  - Labs



Course Topics

- Instruction Classification and Design Principle ~ 1 week
- Concept, Category, Architecture and Design of Pipeline CPU ~ 1.5 weeks
- Hazard of Pipeline CPU ~ 2 weeks
- Software/Hardware Interfaces ~ 1 week
- Introduction of OS ~ 2 weeks
- Interrupt ~ 2 week
- Process and Thread ~ 2 weeks
- Scheduling, Synchronization and Deadlock ~ 2.5 weeks
- Final Review ~ 1 week

Instructor & TA

- Instructors
  - Li Lu 卢立 (li.lu@zju.edu.cn)
  - Wenbo Shen 申文博 (shenwenbo@zju.edu.cn)

- TAs
  - Xiaoran Pan 潘潇然
  - Yuanfan Zhang 张远帆
  - Xuanyong Lin 林轩永
  - Wenshuo Wang 王文烁
  - Chenghao Jiang 蒋城昊

Course Organization

- Lectures
  - Wednesday (Zijingang North 3-310)
    - 1:25 PM – 3:00 PM
  - Friday (Zijingang North 3-310)
    - 8:00 AM – 9:35 AM
- Labs
  - Friday (Zijingang North 3-310)
    - 10:00 AM – 12:25 AM
  - No group, please work alone

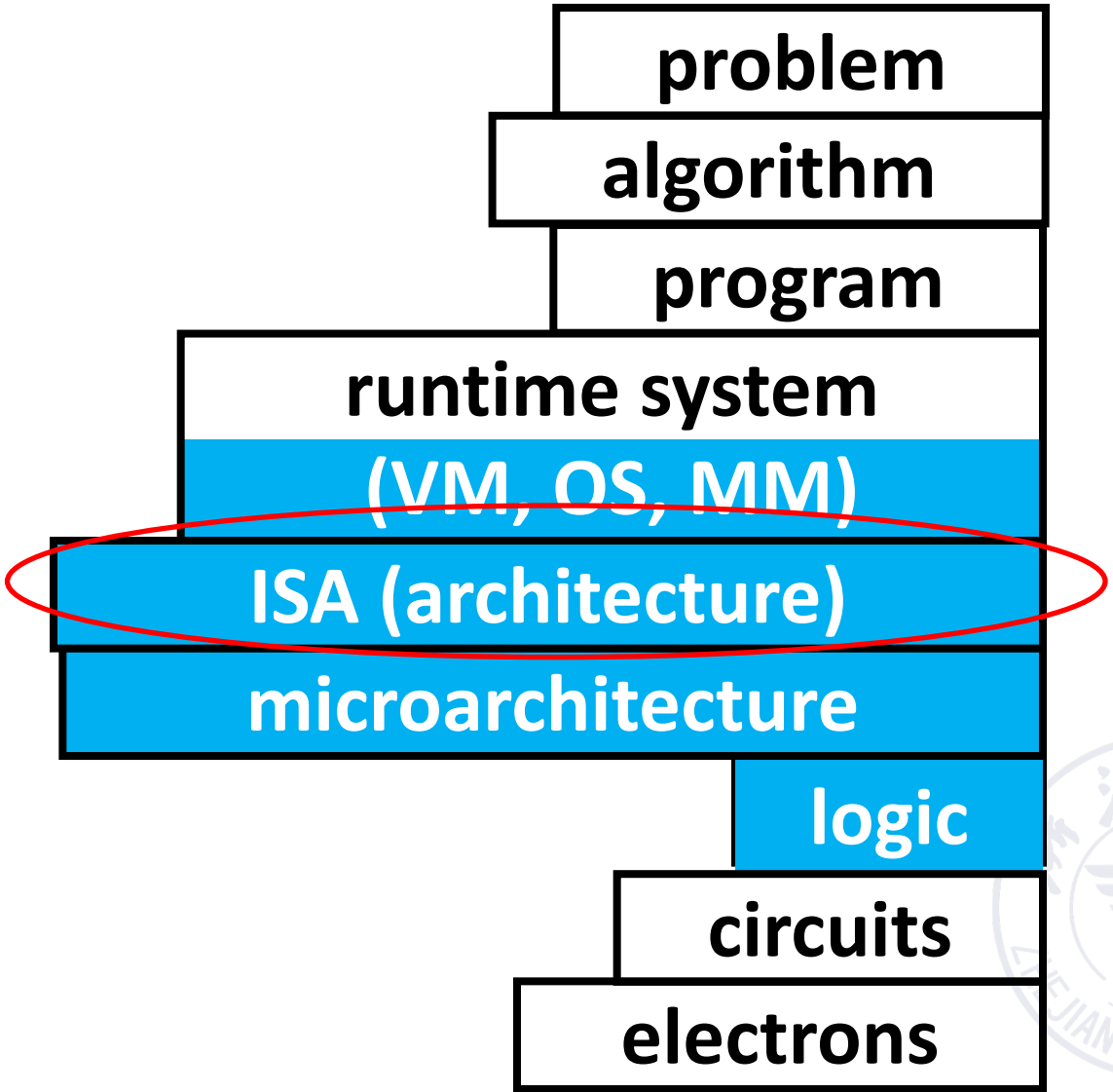
Course Policy

- Academic integrity
  - We will strictly enforce the university, college, and department policies against academic dishonesty
  - Plagiarism in any form will not be tolerated!
  - IMPORTANT: Plagiarism checking begins from 2024. Plagiarism once causes the specific lab credit loss, and twice causes the whole course failed!
- Unless otherwise noted, work turned in should reflect your independent capabilities
  - If unsure, note / cite sources and help
- Late work penalized 5%/day
  - No penalty for documented emergency or by prior arrangement in special circumstances

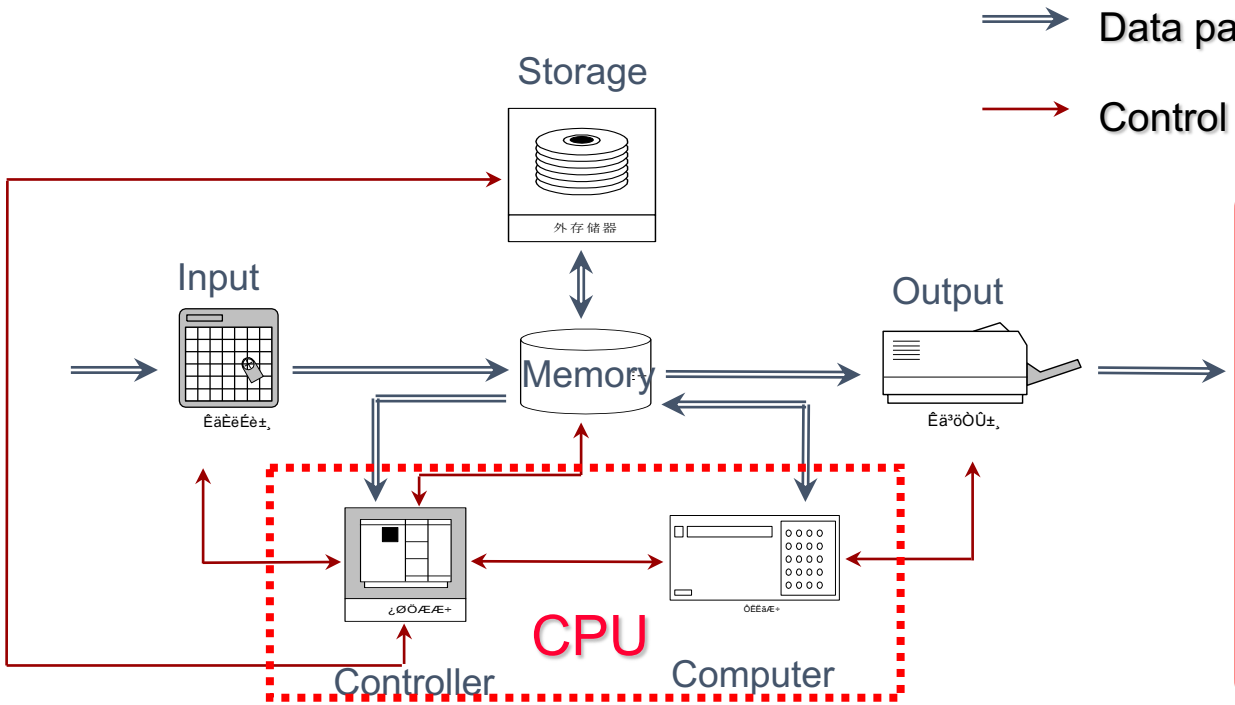
Systems I Review

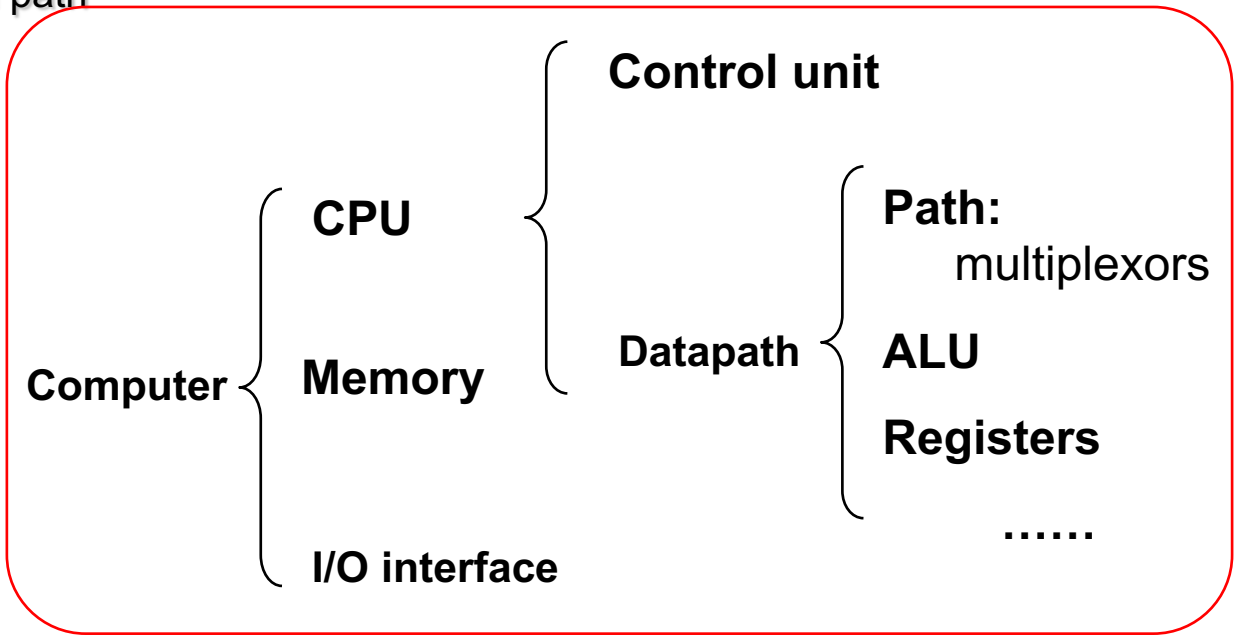
What is the architecture of computers?

Computer Architecture



What is the architecture of computers?





- Von Neumann structure: data and programs are in memory.
- CPU takes instructions and data from memory for operation and puts the results into memory.

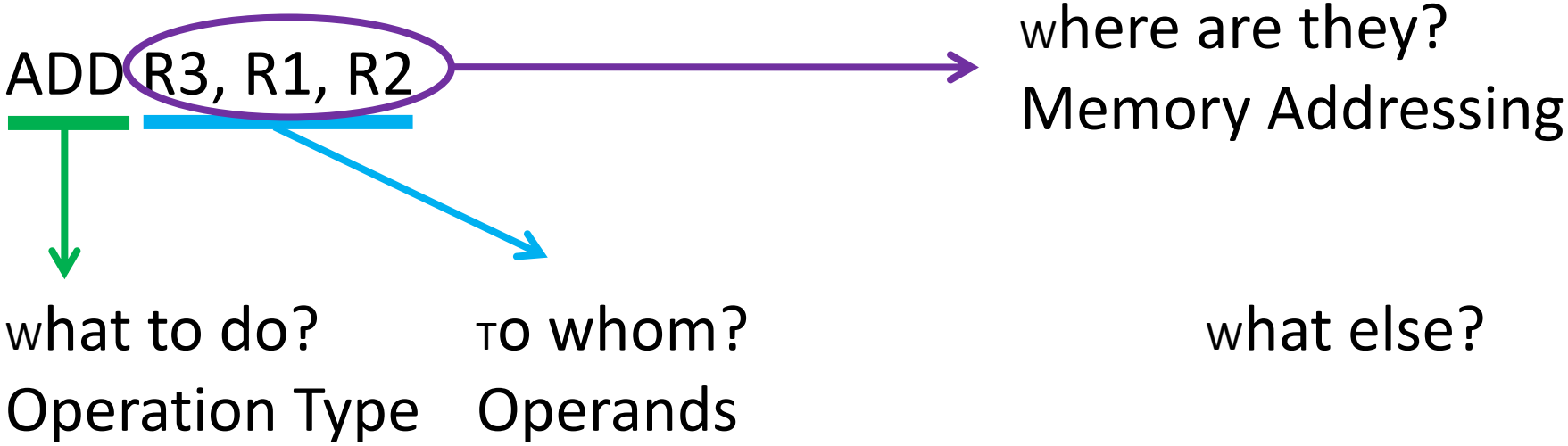
Von Neumann Structure

What are the basic principles of CPU/ISA design?

- Instruction Set
  - The instructions repertoire of a computer
  - Different computers have different instruction sets
    - with many aspects in common
  - Early computers had very simple instruction sets
    - with simplified implementation
  - Many modern computers also have simple instruction sets

What are the basic principles of CPU/ISA design?

- Instruction Set **Architecture**



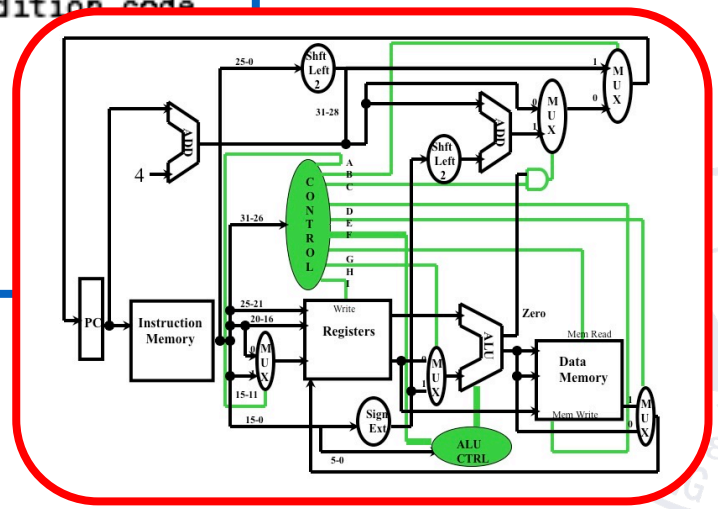
What are the basic principles of CPU/ISA design?

- Instruction Set Architecture (ISA)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void read(int *p);
4  int findmax(int *p);
5  #define N 10
6  int m,n;
```

Programmer-visible instruction set

```
        LOAD    R1,&a       ; R1 <-- contents of 'a'
        LOAD    R2,&a       ; R2 <-- contents of 'a'
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JNE     @L1         ; goto L1 if not equal
        ADD     R1,R2       ; R1 <-- R1 + R2
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JGE     @L2         ; goto L2 if R1 >= R2
        JMP     @END        ; goto END
@L1     ADD     R1, R2      ; R1 <-- R1 + R2
        JMP     @END        ; goto END
@L2     ADD     R1, R2      ; R1 <-- R1 + R2
@END    SUB     R2, R3      ; R2 <-- R2 - R3
```



How to Design a CPU?

- Understand ISA
  - Taking RISC-V as an example
- 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 ISA

32-bit Instruction Formats

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

16-bit (RVC) Instruction Formats

	15–13	12	11–10	9–8	7–6	5–4	3–2	1–0
CR	funct4		rd/rs1			rs2		op
CI	funct3	imm	rd/rs1			imm		op
CSS	funct3	imm				rs2		op
CIW	funct3	imm					rd′	op
CL	funct3	imm		rs1′	imm		rd′	op
CS	funct3	imm		rs1′	imm		rs2′	op
CB	funct3	offset		rs1′	offset			op
CJ	funct3	jump target						op

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	B	BEQ rs1,rs2,imm	
	Branch ≠	B	BNE rs1,rs2,imm	
	Branch <	B	BLT rs1,rs2,imm	
	Branch ≥	B	BGE rs1,rs2,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm	
Jump & Link	J&L	J	JAL rd,imm	
	Jump & Link Register	I	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

Control Status Register (CSR)			
Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm
			+RV64I	
			LWU	rd,rs1,imm
			LD	rd,rs1,imm
			SD	rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Float Load Word SP	CL	C.FLW rd',rs1',imm	FLW rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP rd,imm	FLW rd,sp,imm*8
	Float Load Double	CL	C.FLD rd',rs1',imm	FLD rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP rd,imm	FLD rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Float Store Word	CS	C.FSW rs1',rs2',imm	FSW rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP rs2,imm	FSW rs2,sp,imm*8
	Float Store Double	CS	C.FSD rs1',rs2',imm	FSD rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP rs2,imm	FSD rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
	AND	CR	C.AND rd,rs1	AND rd,rd,rs1
	AND Immediate	CI	C.ANDI rd,imm	ANDI rd,rd,imm
	OR	CR	C.OR rd,rs1	OR rd,rd,rs1
	eXclusive OR	CR	C.XOR rd,rs1	AND rd,rd,rs1
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI rd,imm	SRAI rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI rd,imm	SRLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word strores) plus:

- ADD Word (C.ADDW)
- ADD Imm. Word (C.ADDIW)
- SUBtract Word (C.SUBW)
- Load Doubleword (C.LD)
- Load Doubleword SP (C.LDSP)
- Store Doubleword (C.SD)
- Store Doubleword SP (C.SDSP)

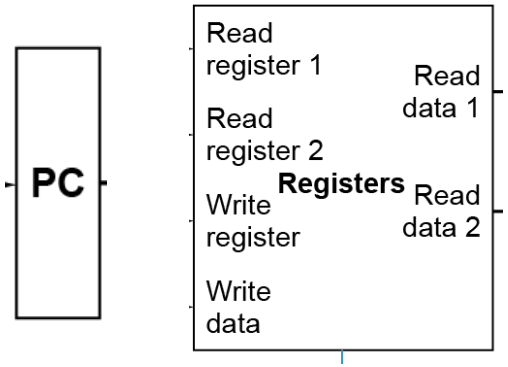
How to Design a CPU?

- Understand instruction execution of CPU
  - **Fetch** :
    - Take instructions from the instruction memory
    - Modify PC to point the next instruction
  - **Instruction decoding & Read Operand**:
    - Will be translated into machine control command
    - Reading Register Operands, whether or not to use
  - **Executive Control**:
    - Control the implementation of the corresponding ALU operation
  - **Memory access**:
    - Write or Read data from memory
    - Only ld/sd
  - **Write results to register**:
    - If it is R-type instructions, ALU results are written to rd
    - If it is I-type instructions, memory data are written to rd
  - **Modify PC** for branch instructions

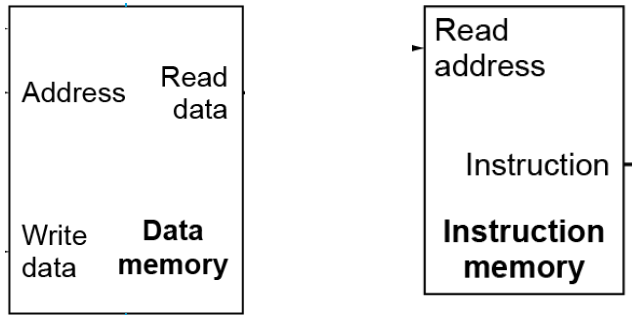
How to Design a CPU?

- Knowing the elements

Registers

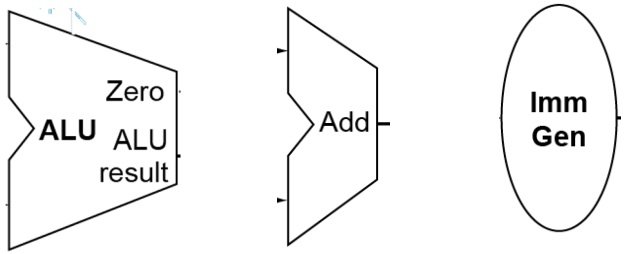


Memories



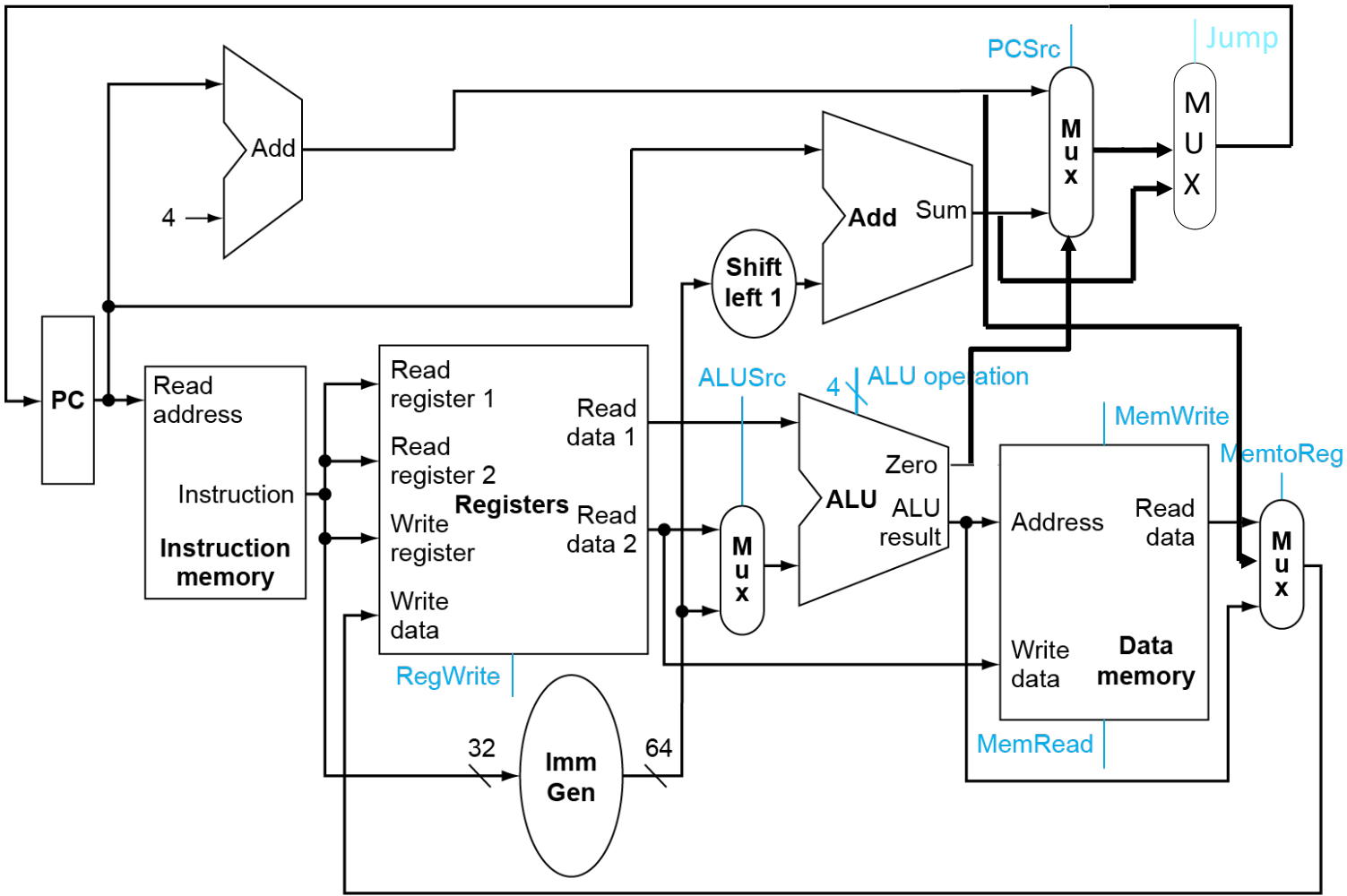
Multiplexer

Arithmetic and Logic



How to Design a CPU?

- Construct the datapath



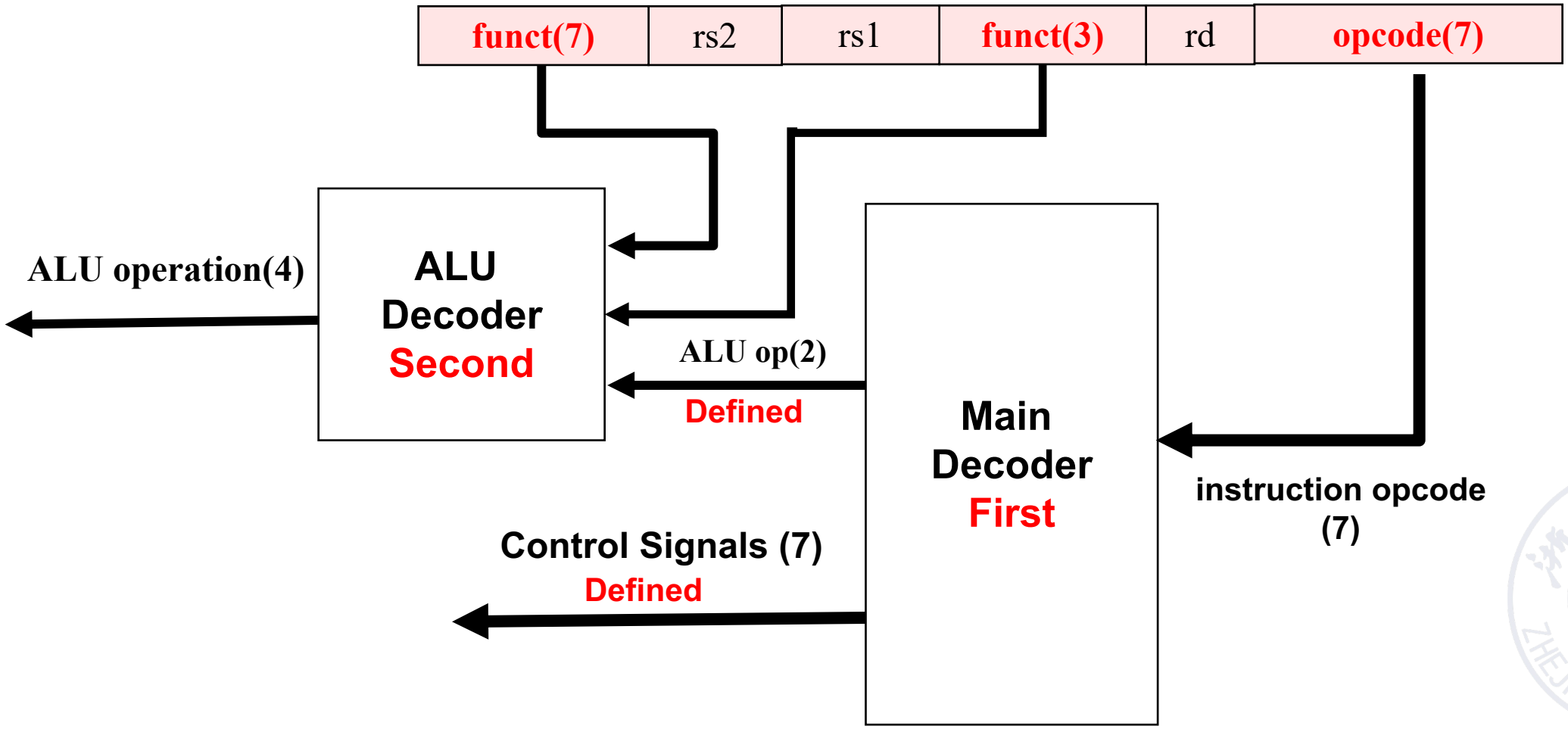
How to Design a CPU?

- Construct the controller
  - Information comes from the 32 bits of the instruction
  - Selecting the operations to perform (ALU, read/write, etc.)
  - Controlling the flow of data (multiplexor inputs)
  - ALU's operation based on instruction type and function code

Input		Output								
Instruction	OPCode	ALUSrcB	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld(I-Type)	0000011	1	01	1	1	0	0	0	0	0
sd(S-Type)	0100011	1	x	0	0	1	0	0	0	0
beq(B-Type)	1100111	0	x	0	0	0	1	0	0	1
Jal(J-Type)	1101111	x	10	1	0	0	0	1	x	x

How to Design a CPU?

- Construct the controller



Why not Single-Cycle?

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Waste of area. If the instruction needs to use some functional unit multiple times.
  - E.g., the instruction ‘mult’needs to use the ALU repeatedly. So, the CPU will be very large
- Any solution for performance improvement?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-type	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

How to Improve the CPU?

- Reduce the number of instructions inner a program
  - Make instructions that *do* more (CISC)
  - User better compilers
- Use less cycles to perform an instruction
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
- Increase the clock frequency
  - Find a *newer* technology to manufacture
  - Redesign time critical components
  - Adopt multi-cycle

Multi-Cycle CPU Design

- Single-Cycle microarchitecture
  - + Simple
  - - Clock cycle time limited by longest instruction (lw)
  - - Two adders/ALUs and two memories
- Multi-Cycle microarchitecture
  - + Shorter clock cycle period
  - + Simpler instructions run faster
  - + Reuse expensive hardware on multiple cycles
  - - Sequencing overhead paid many times

Why not Multi-Cycle?

- Even Longer Time for Instruction Execution
  - E.g., 92.5s (Single-cycle) < 133.9s (Multi-cycle)
  - Cannot achieve expected performance compared with single-cycle
- But provide a new perspective for CPU design
  - Finer-grained execution in one clock cycle
- We will improve performance by ***pipelining***

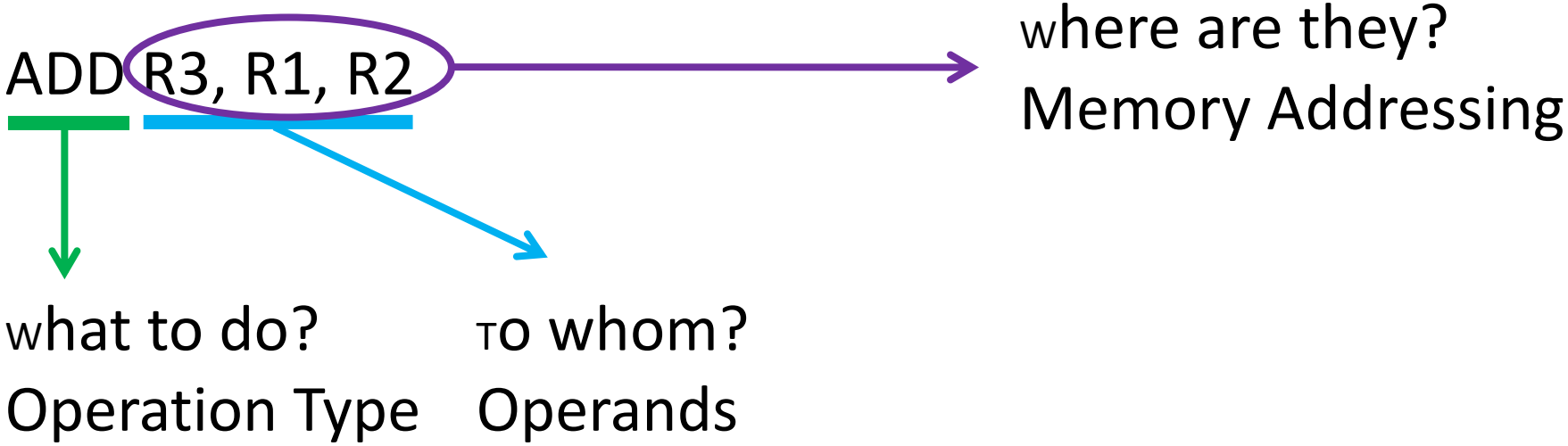
Instruction Set Principles

Goals of ISA design

- Compatibility
- Versatility
- High efficiency
- Security

Basic principles of ISA design

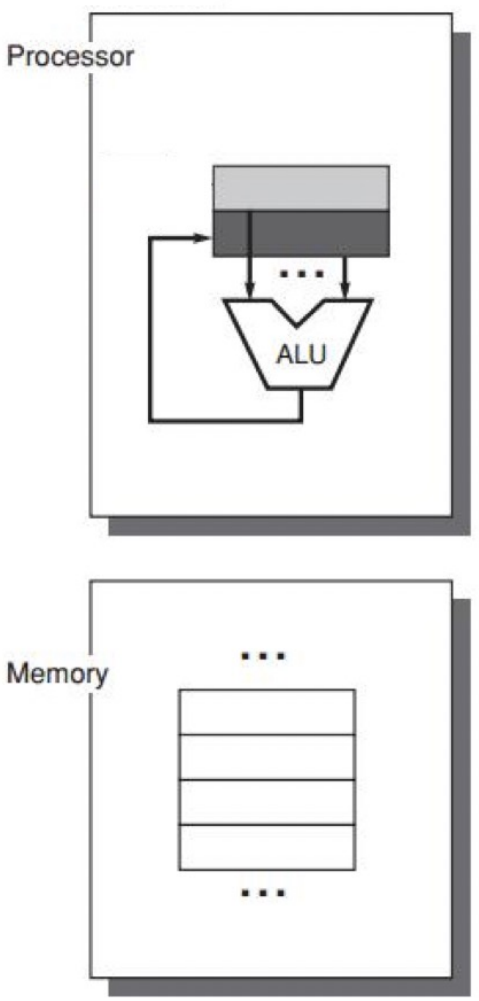
- Instruction Set **Architecture**



ISA Classification Basis

- The types of internal storage:
  - Stack
  - Accumulator
  - Register

In processor, stores data fetched from memory or cache



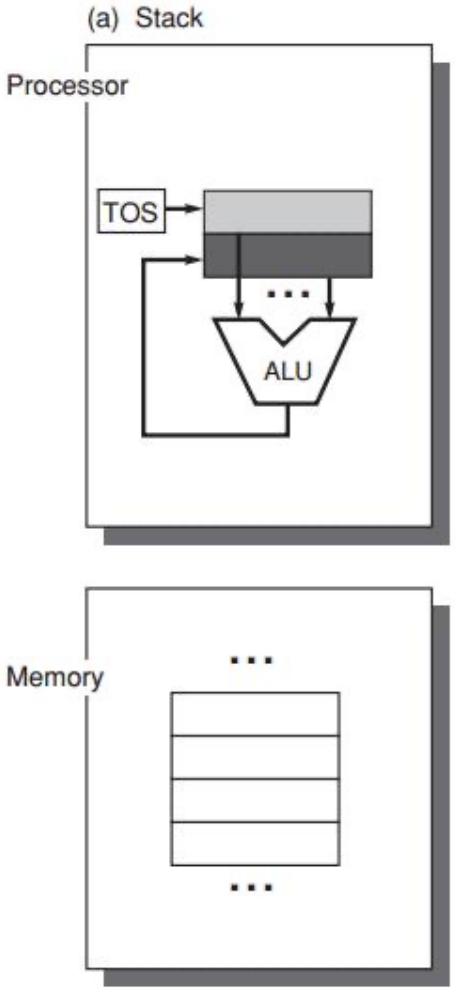
ISA Classes

- Stack architecture
- Accumulator architecture
- General-purpose register architecture (GPR)

ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

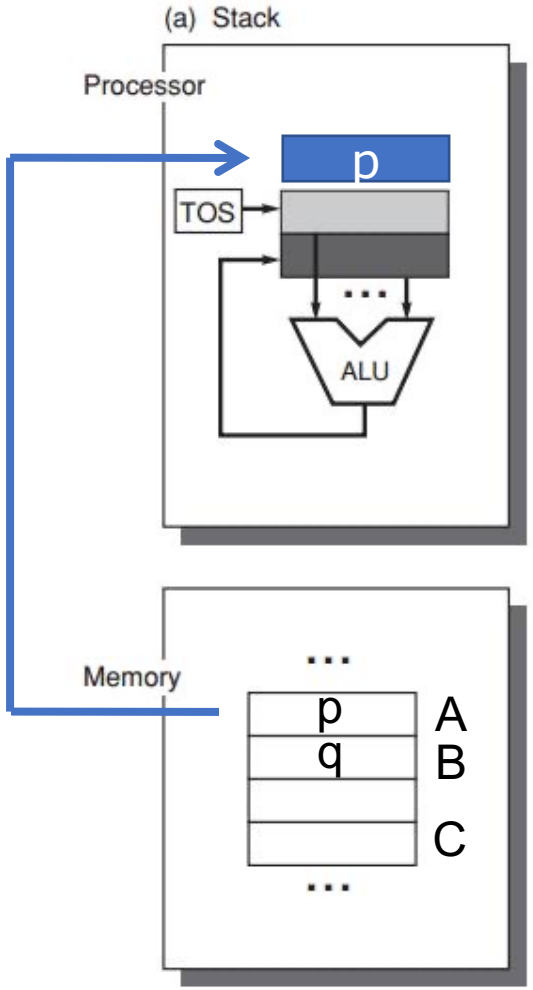
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

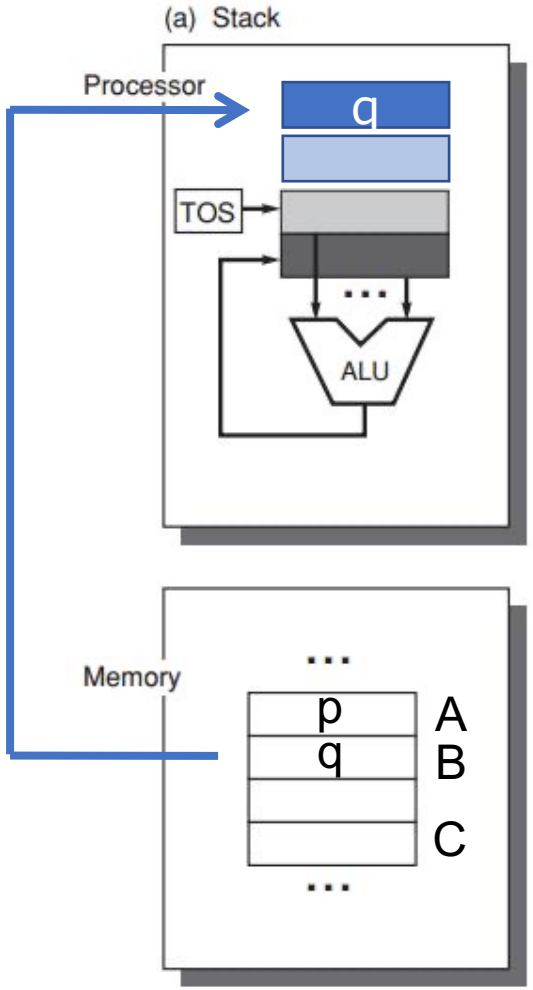
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

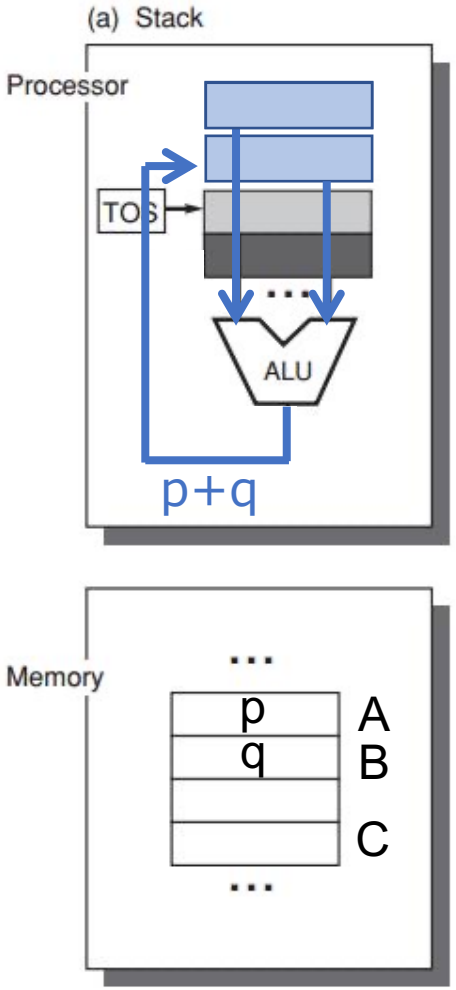
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



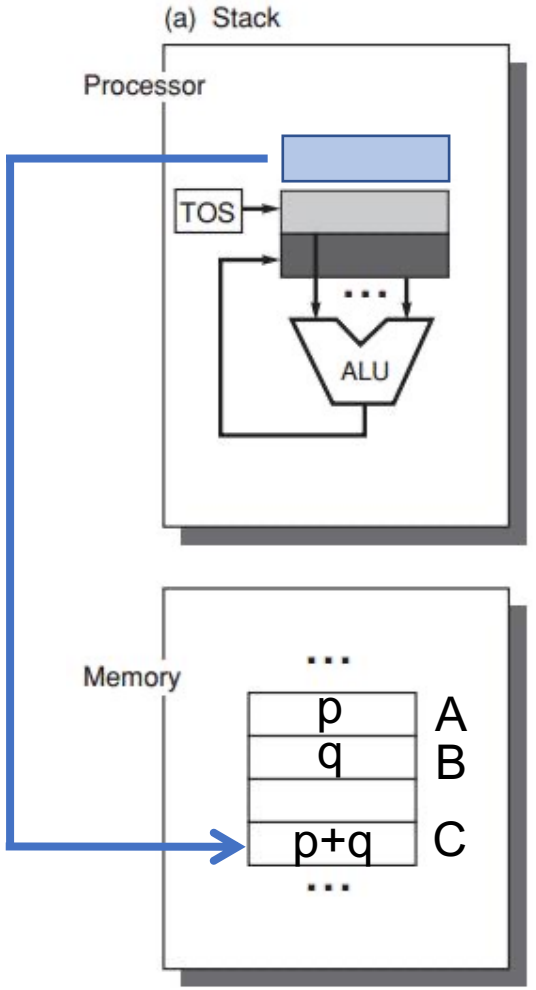
ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)
- First operand removed from
  second op replaced by the result
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

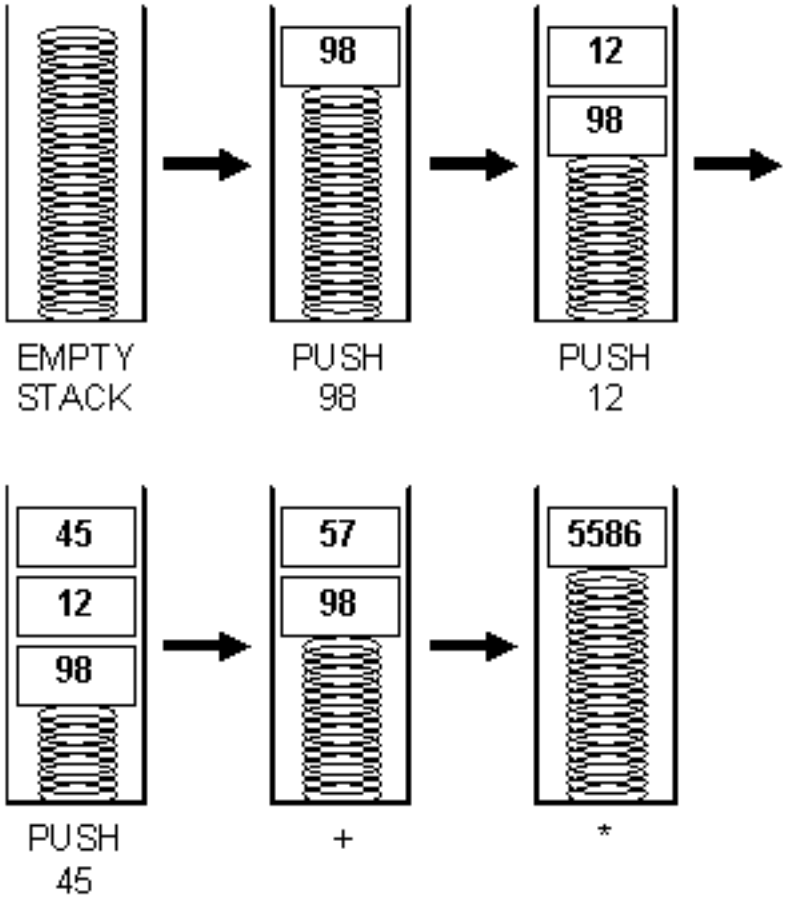
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)
- First operand removed from
  second op replaced by the result
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

- Example:

98 * (12 + 45)



ISA Classes: Accumulator Architecture

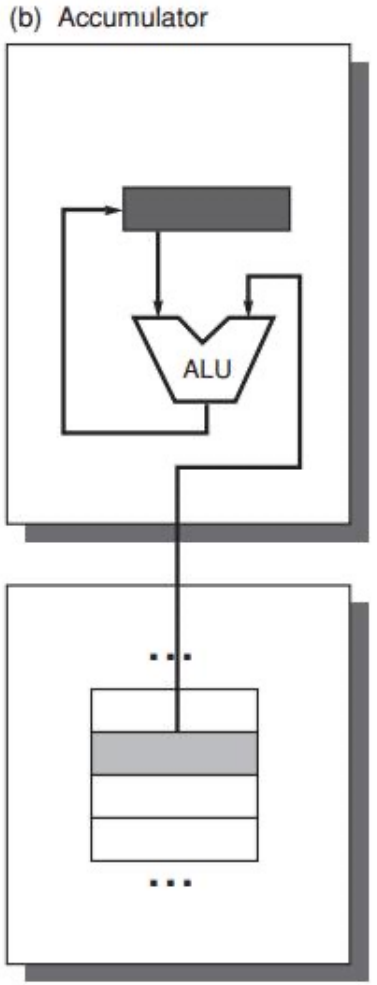
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: Accumulator Architecture

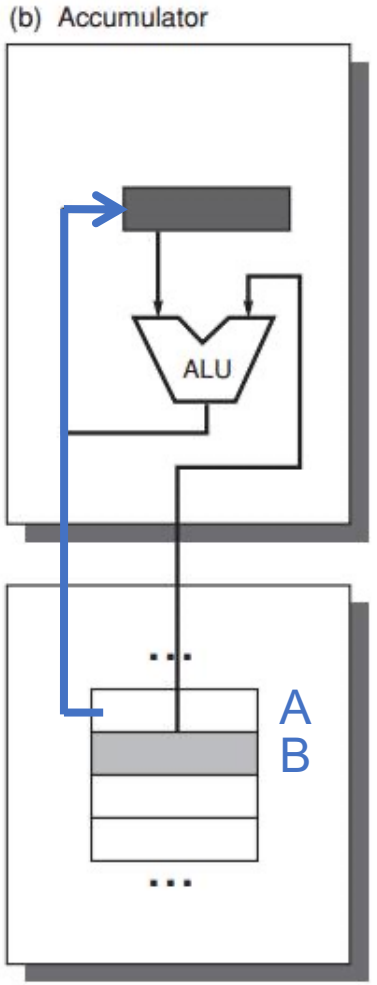
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



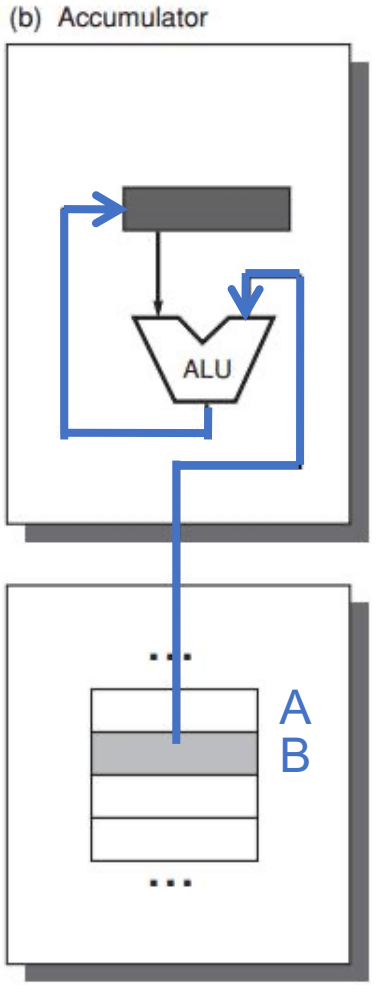
ISA Classes: Accumulator Architecture

- 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



ISA Classes: Accumulator Architecture

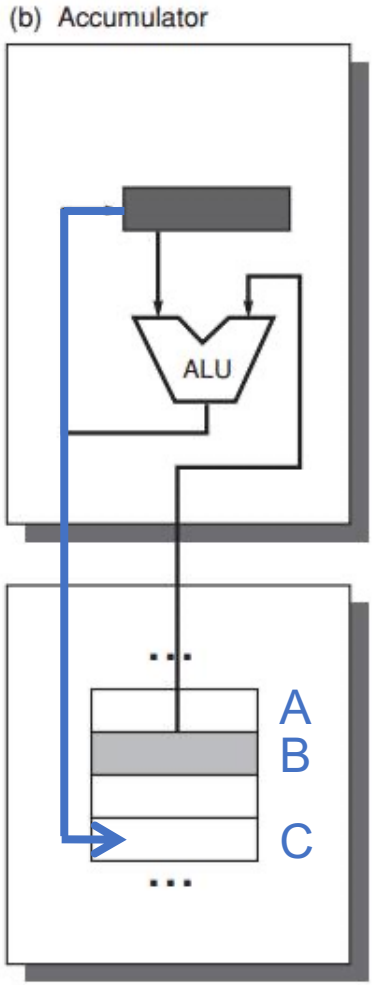
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: General-Purpose Register Arch

- Only explicit operands

  registers

  memory locations

- Operand access:

  direct memory access

  loaded into temporary storage first

ISA Classes: General-Purpose Register Arch

Two Classes:

- Register-memory architecture

  any instruction can access memory

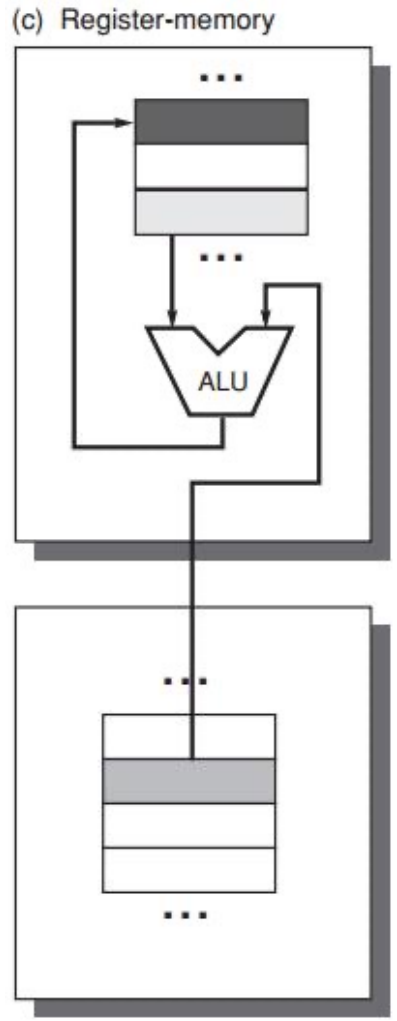
- Load-store architecture

  only load and store instructions can access memory

GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

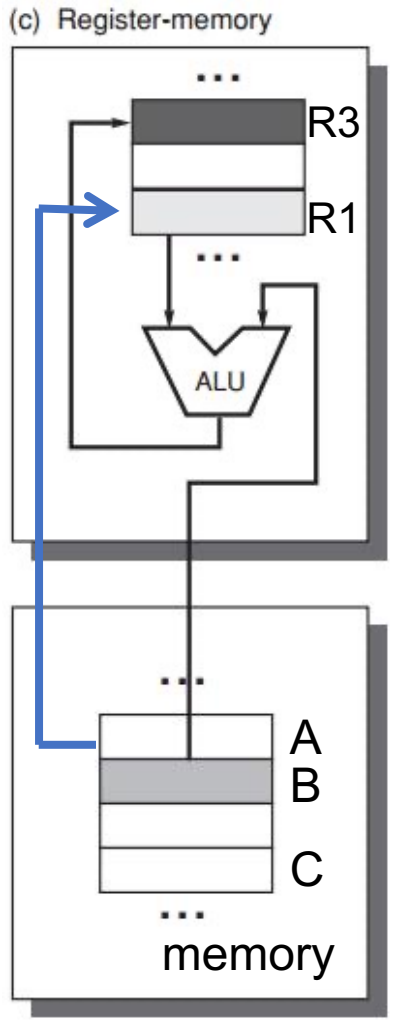
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

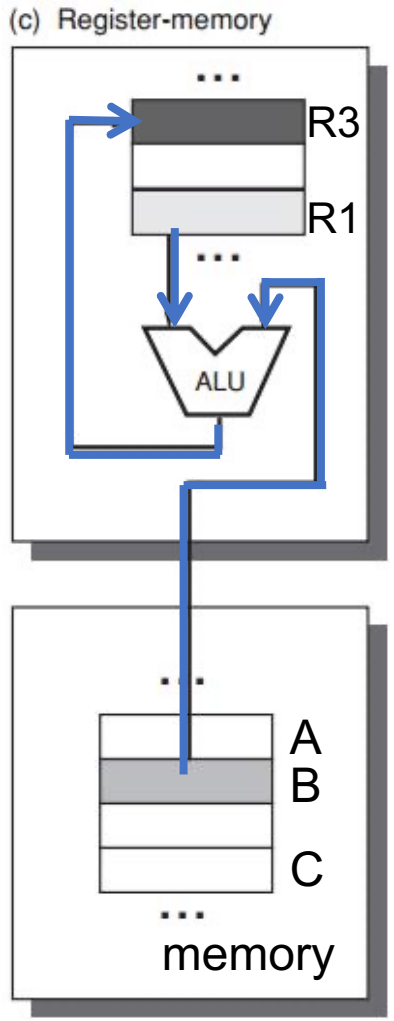
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

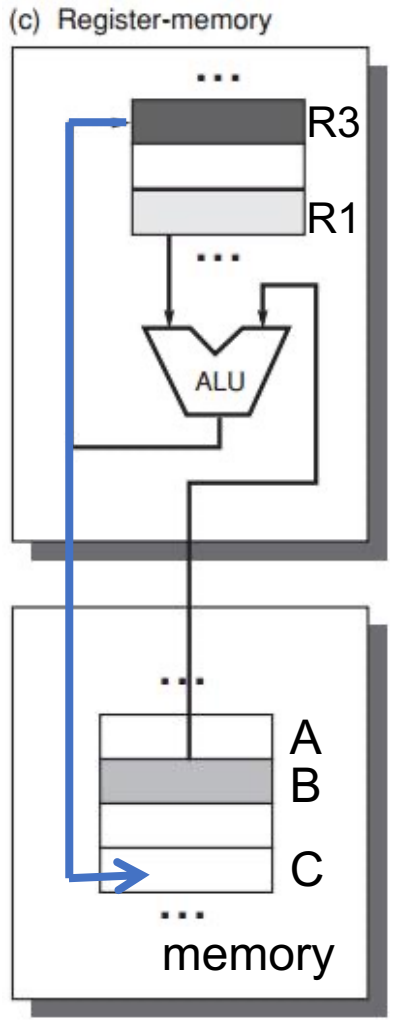
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: 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

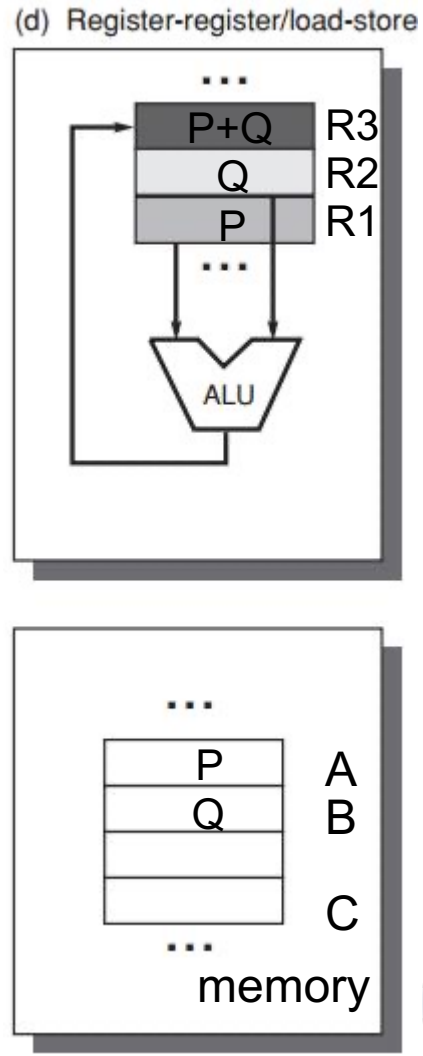
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

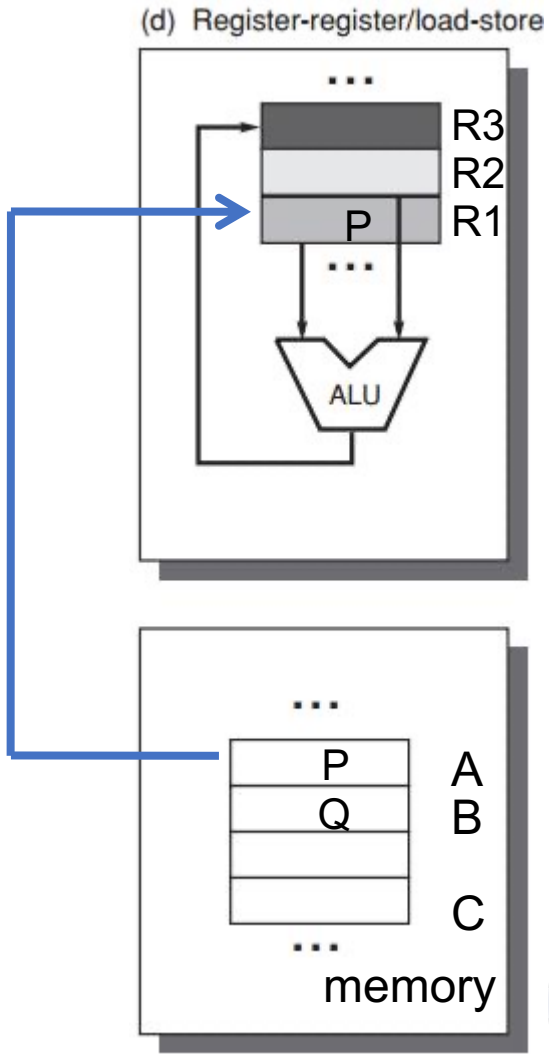
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

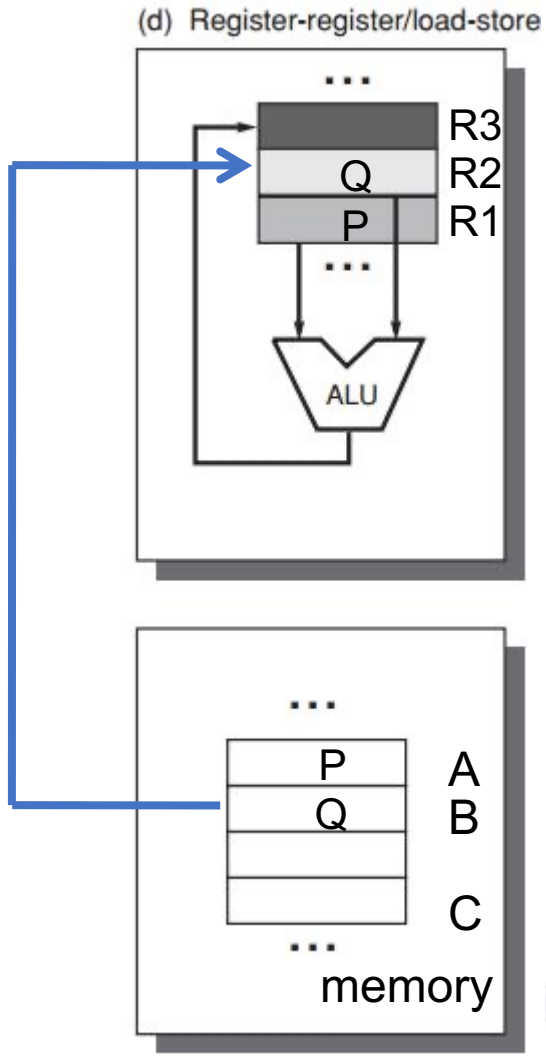
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

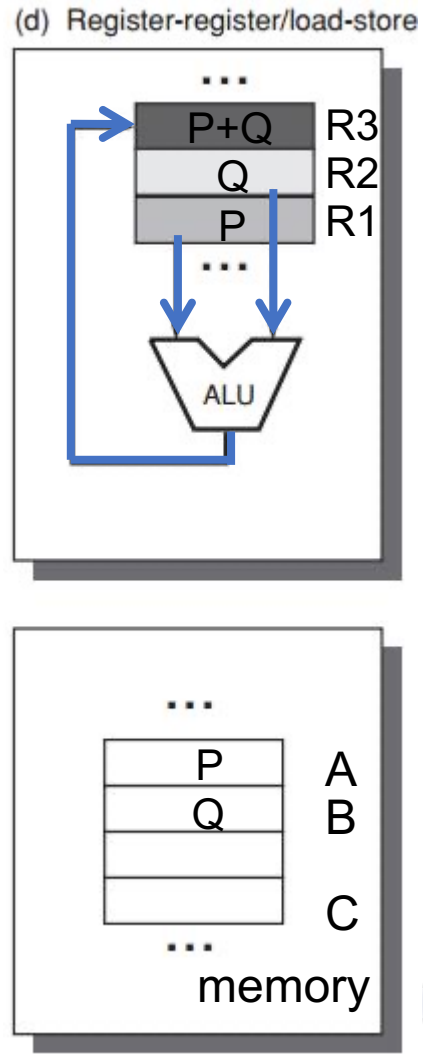
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

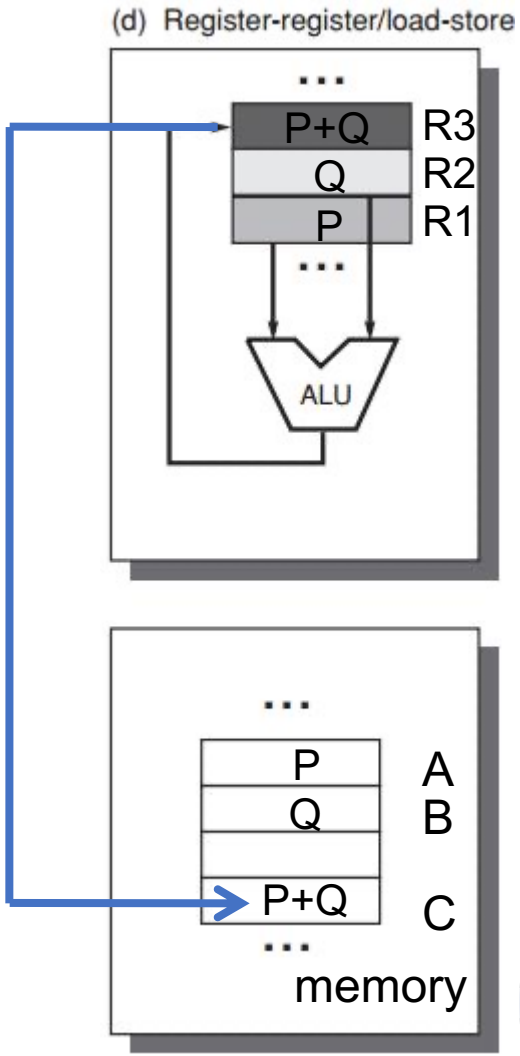
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR Classification

- ALU instruction has 2 or 3 operands?

  2 operands = 1 result & source op + 1 source op

  3 operands = 1 result op + 2 source op

- ALU instruction has 0, 1, 2, or 3 operands of memory address

GPR Classification

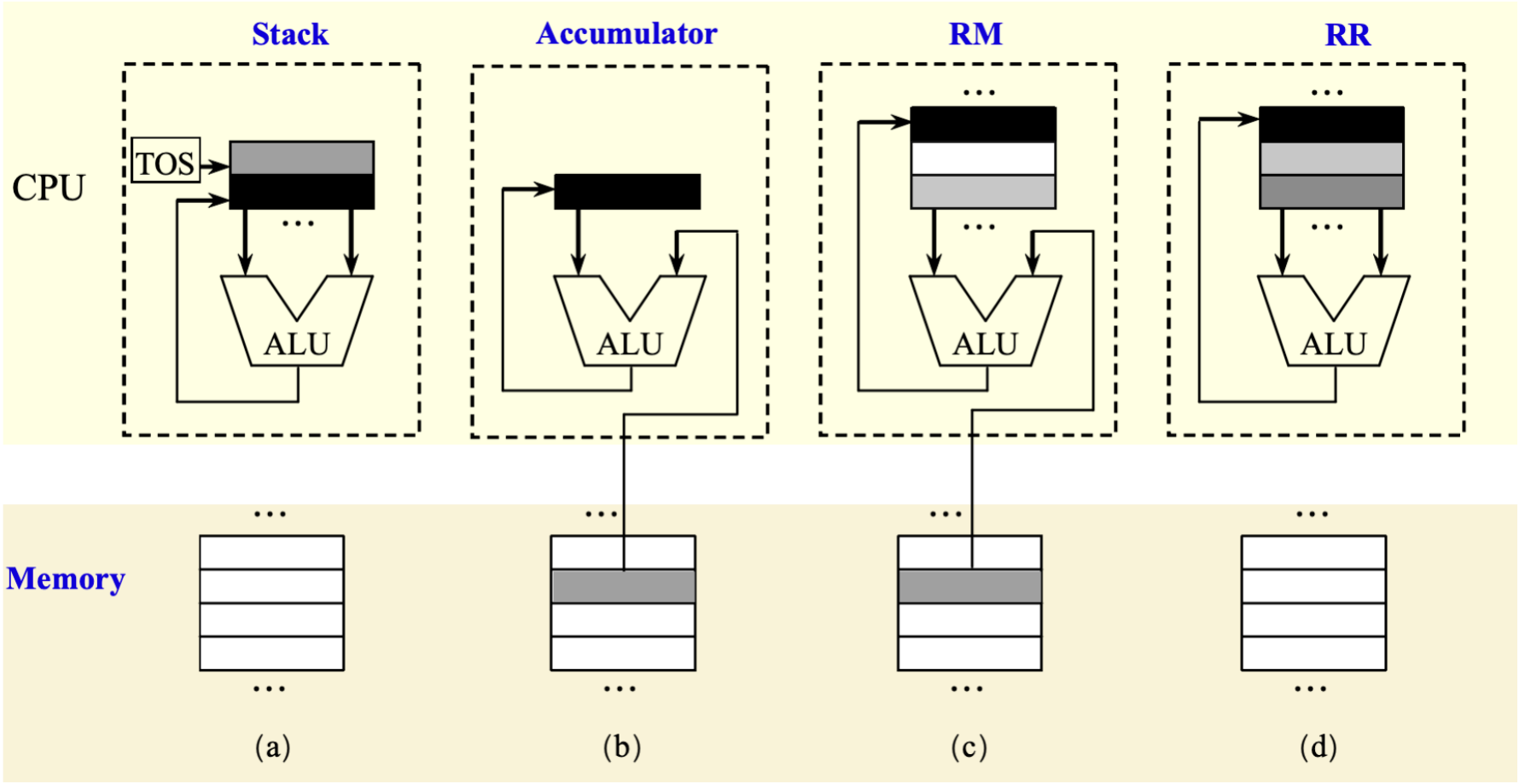
- Three major classes

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

GPR Classification

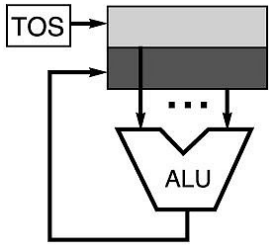
Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

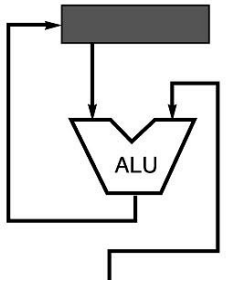
GPR Classification



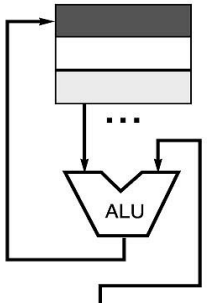
Code Sequence C = A + B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1, A	Load R1,A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

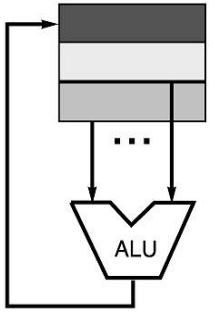




acc = acc + mem[C]



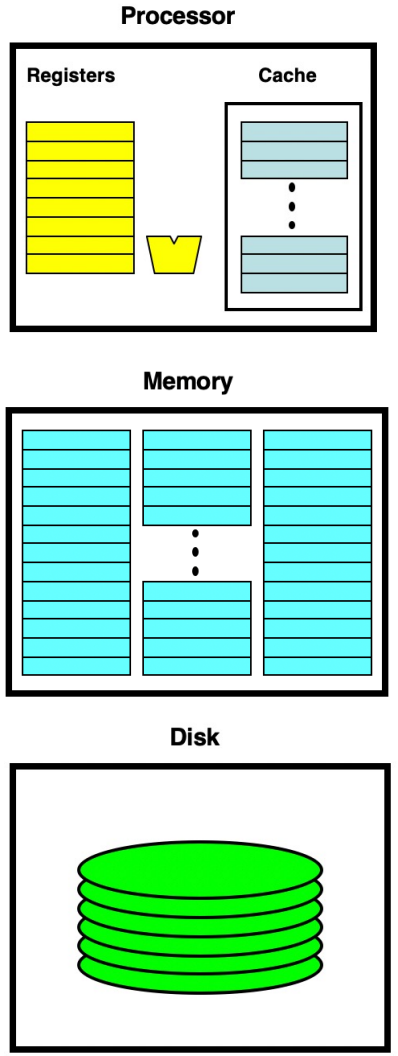
R1 = R1 + mem[C]



R3 = R1 + R2

Why do almost all new architectures use GPRs?

- Registers are much faster than memory (even cache)
  - Temporal Locality
  - Register values are available immediately
- Registers are convenient for variable storage
  - Compiler assigns some variables just to registers
  - More compact code since small fields specify registers (compared to memory addresses)



More about ISA

Where to find operands?

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`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	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)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

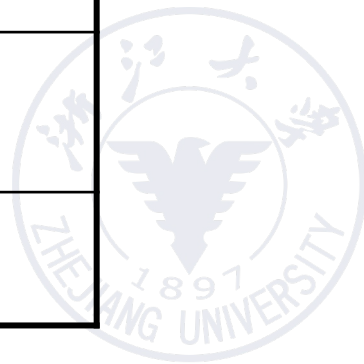
Interpret Memory Address

- 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

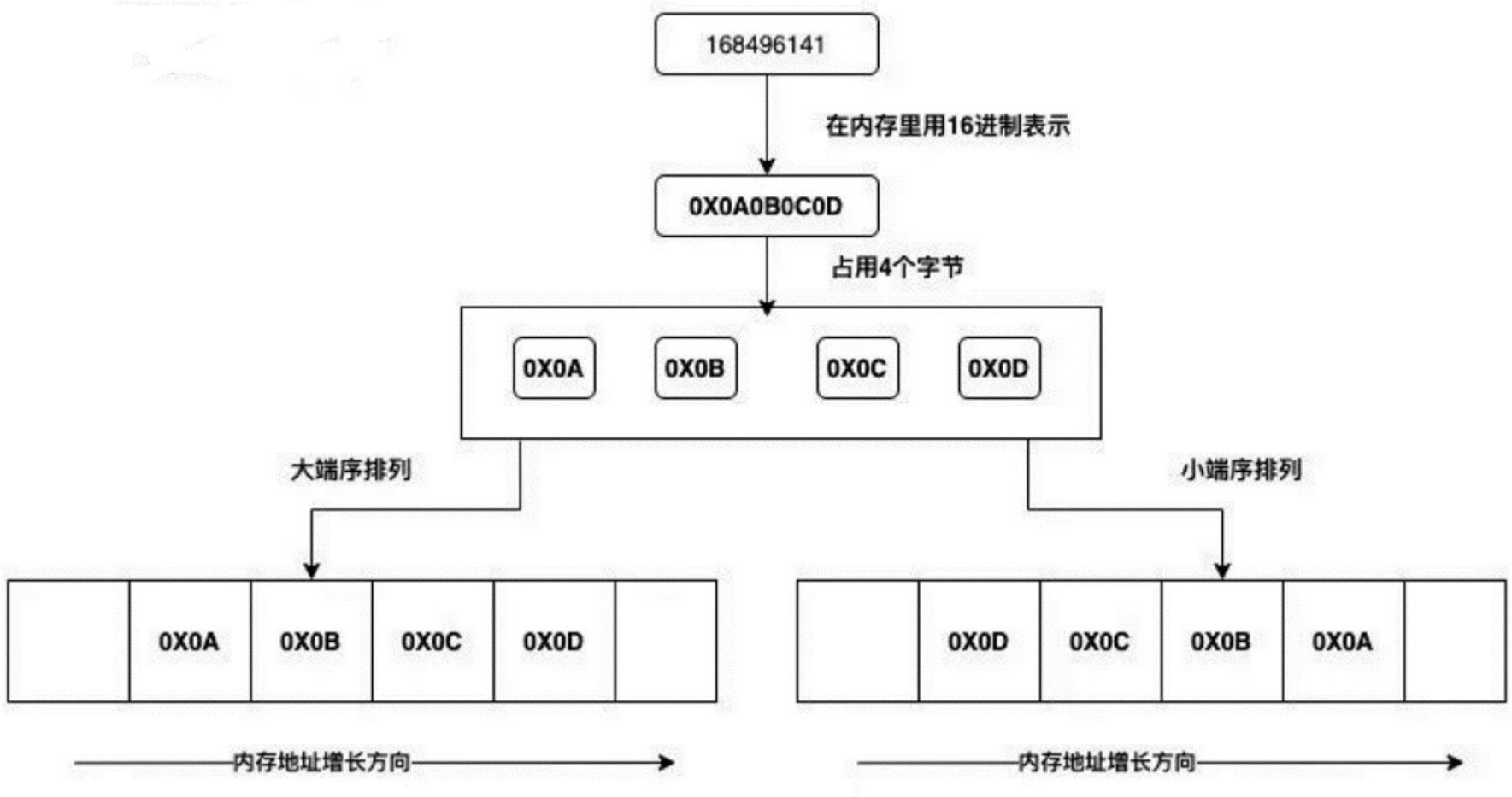
  78 | 56 | 34 | 12

  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* mod *s* = *0*

Interpret Memory Address

- Address alignment

  object width: *s* bytes

  address: *A*

  aligned if *A* mod *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

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:

  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

RISC and RISC-V

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

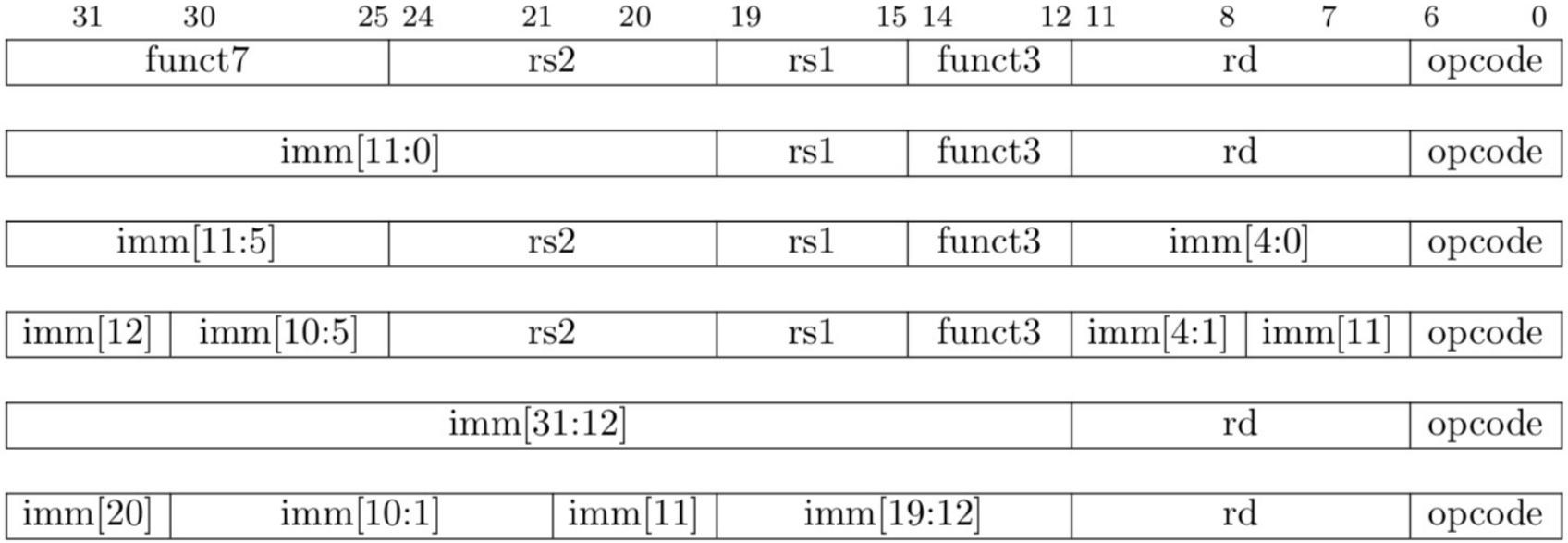
Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**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

- 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



Register-Register

Register Immediate(16bits) + Load

Store

Branch

Register Immediate(20bits)

Jump

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
  - 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

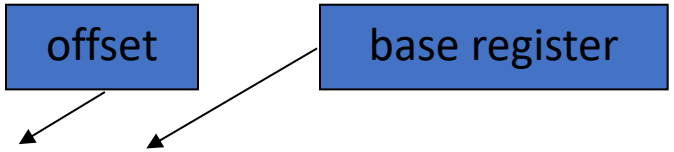
Memory Operand Example 1

- 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




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

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

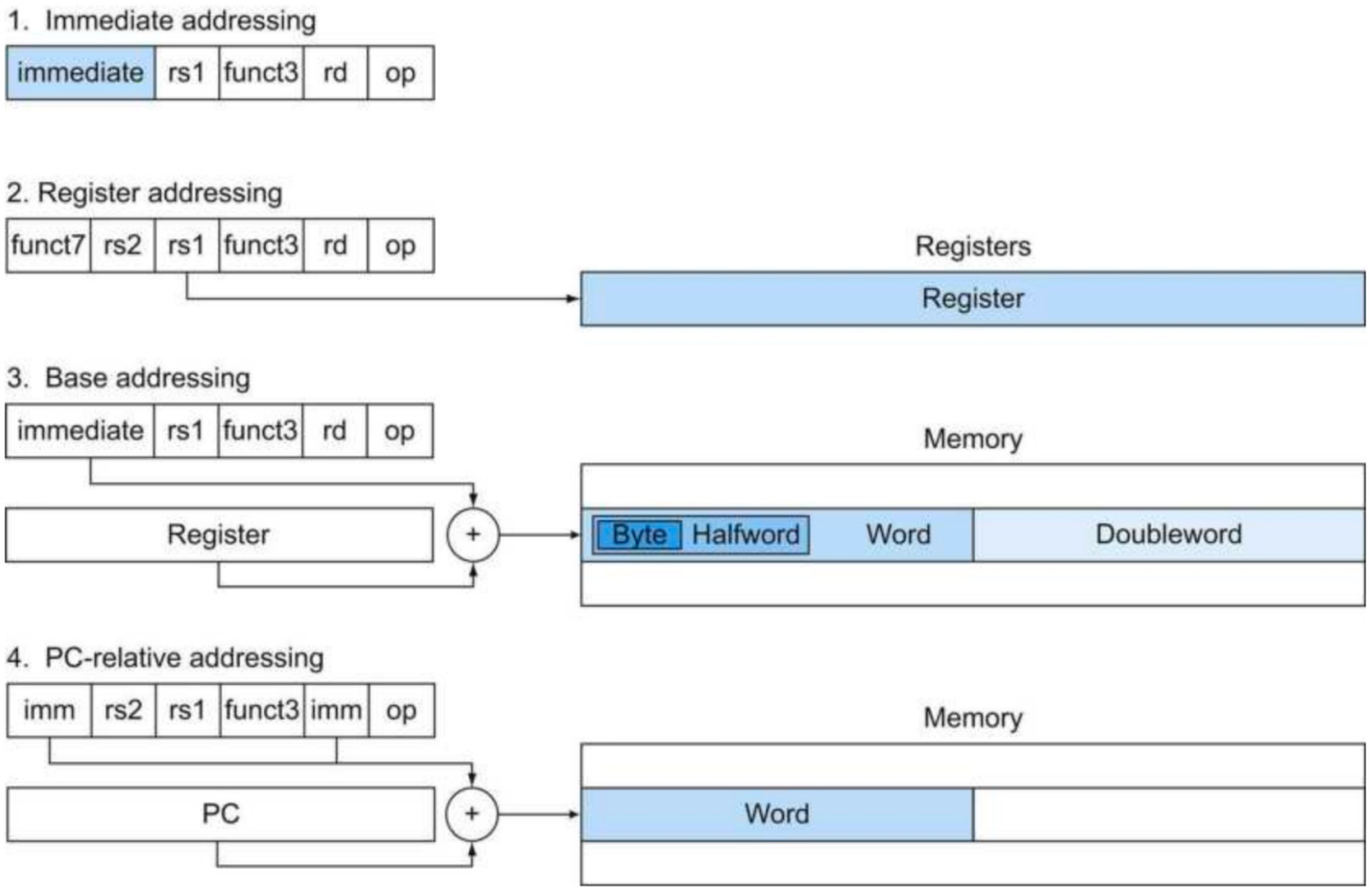
  addi x8, x8, 4

- 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

RISC-V Address Mode



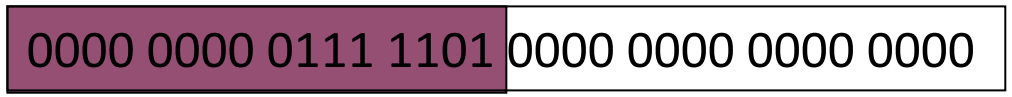
- No Pseudodirect Address (Why?)
- Example: *Jal*
- MIPS: jal offset→ 2^{26}
- RISC-V: jal offset(rd) → 2^{32} → More Address Space + Support for half word address

32-bit Constants

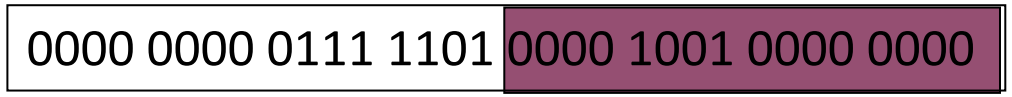
- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  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



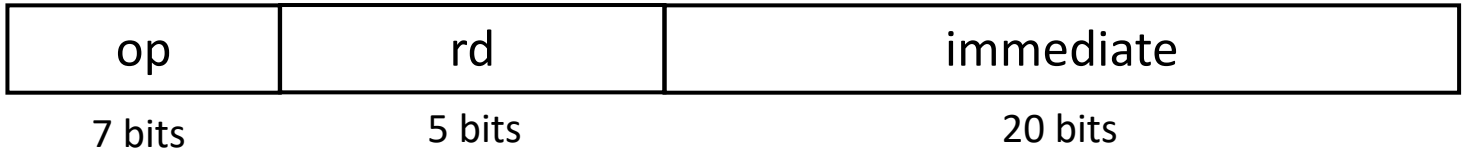
Branch Addressing

- 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 × 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



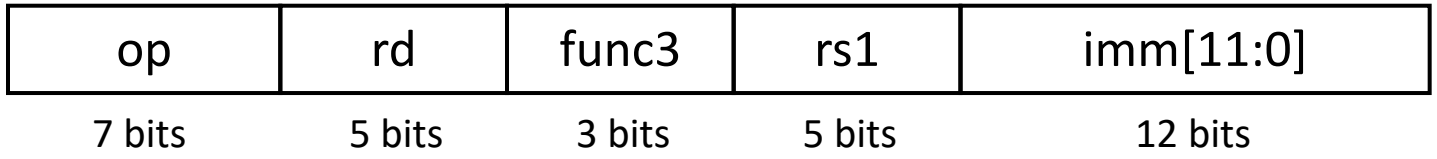
Logical Operations

- 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



- 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

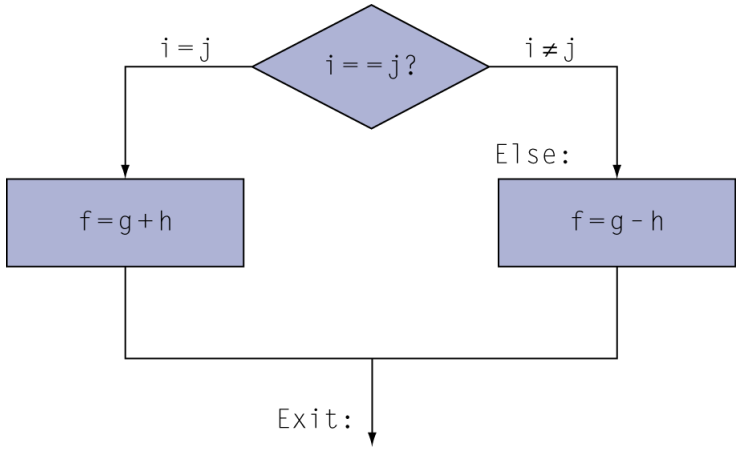
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- ***beq x5, x6, L1***
  - if (rs == rt) branch to PC+L1;
- ***bne x5, x6, L1***
  - if (rs != rt) branch to PC+L1;
- ***jal x1, L1***
  - unconditional jump to PC+L1, and store the previous address PC+4 in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

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 < 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 <, ≥, … slower than =, ≠
  - 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 ⇒ x2 = 1
  - sltu x2, x5, x6 # unsigned
    - +4,294,967,295 > +1 ⇒ 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

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

RISC-V code:

```
leaf_example:
        addi x2, x2, -4
        sw   x7, x2, 0
        add  x5, x10, x11
        add  x6, x12, x13
        sub  x7, x5, x6
        add  x17, x7, x0
        lw   x7, x2, 0
        addi x2, x2, 4
        jalr   x0, x1, 0
```

 # Save x7 on stack

 # Procedure body

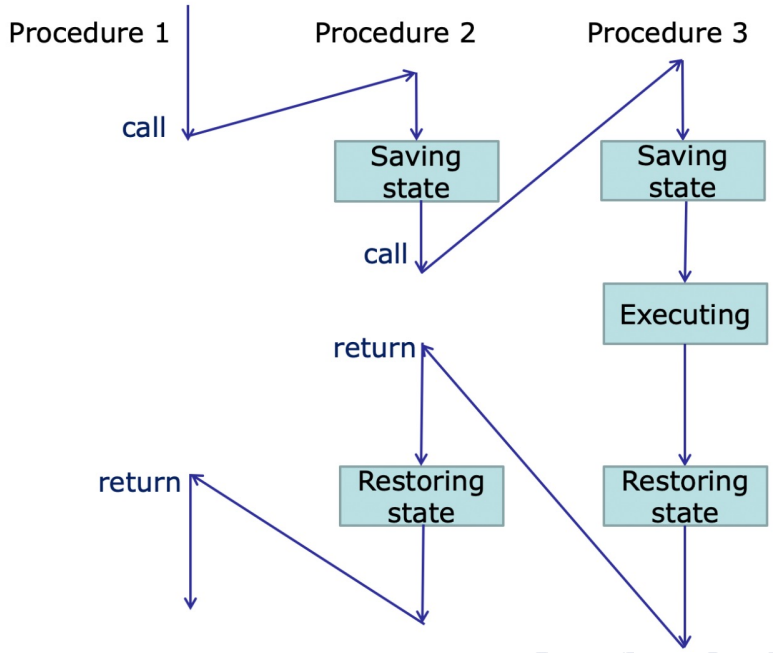
 # Result

 # Restore x7

 # Return

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



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?

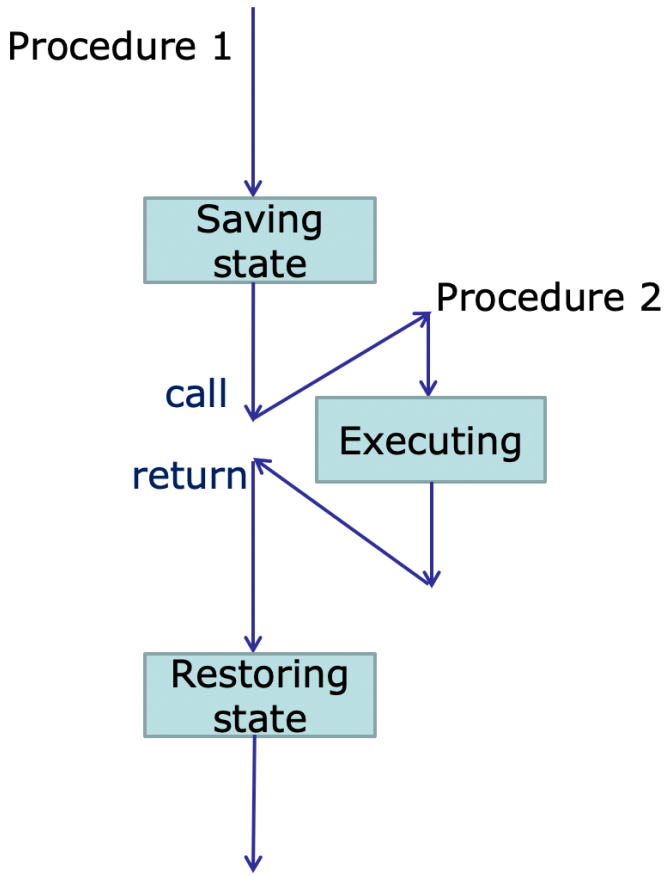
Procedure Invocation Options: Save Registers

- Caller Saving

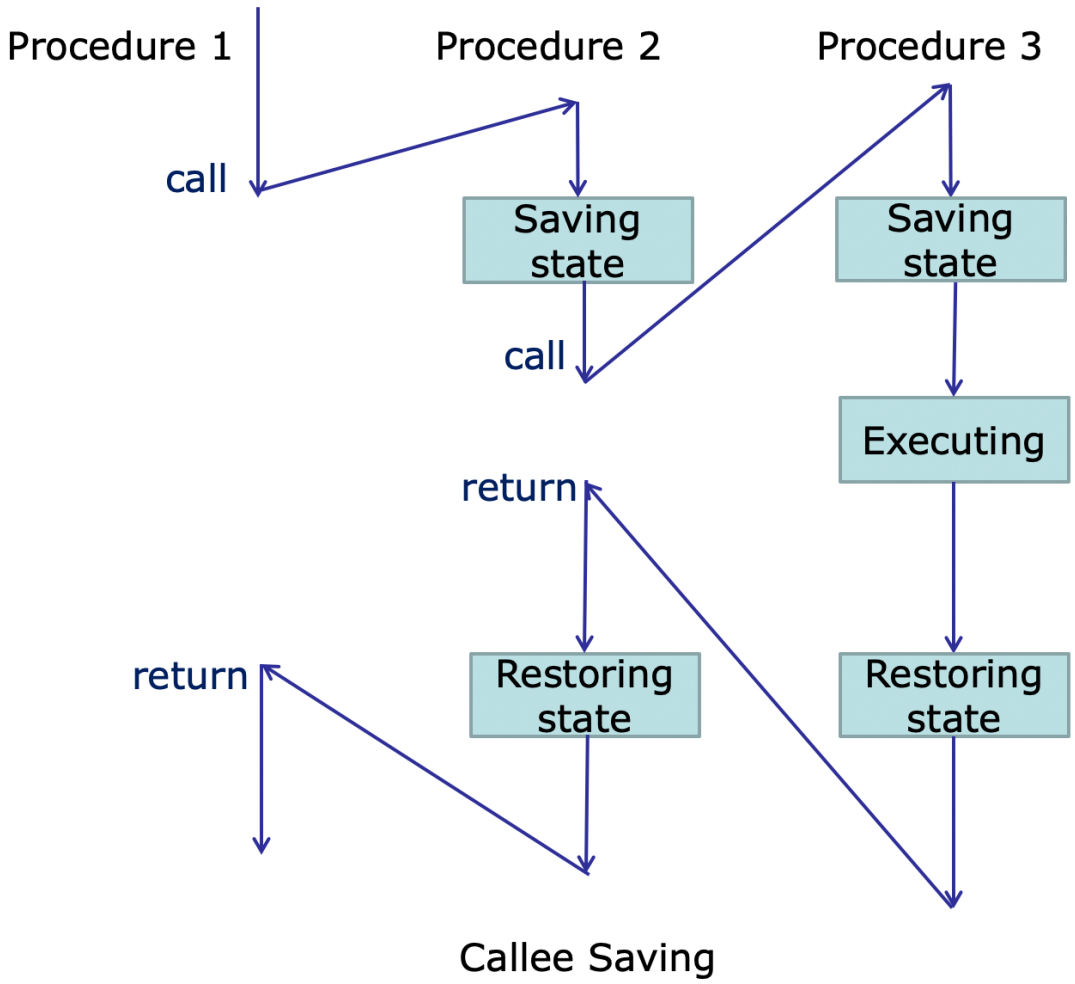
  the calling procedure saves the registers that it wants preserved for access after the call

- Callee Saving

  the called procedure saves the registers it wants to use

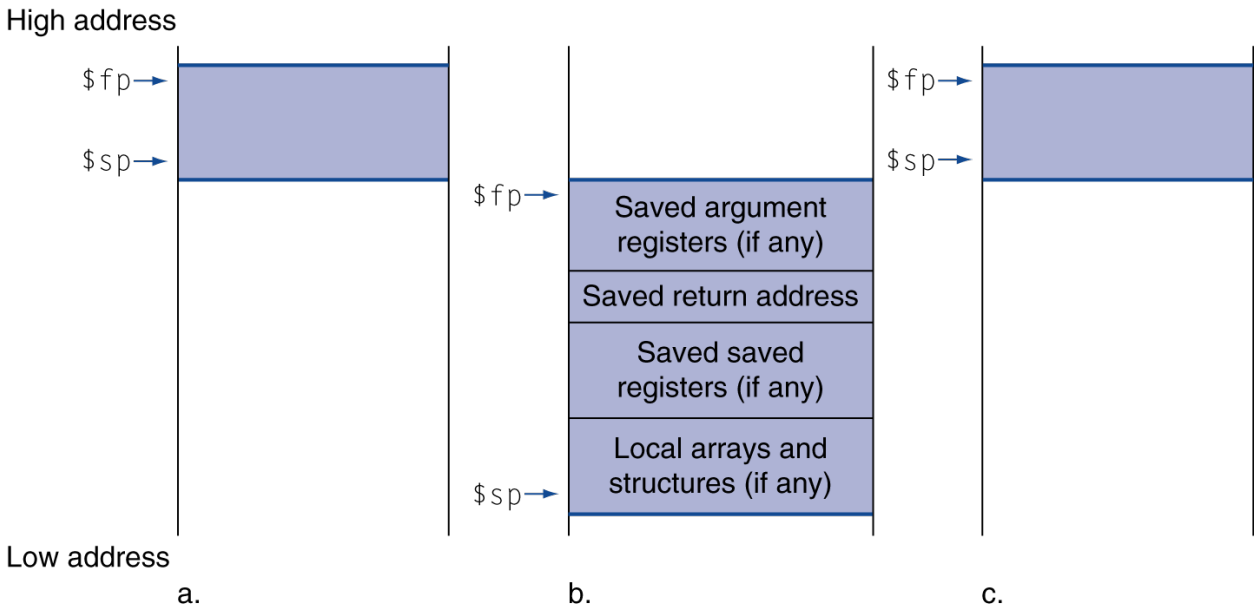


Caller Saving



Callee Saving

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

The Four ISA Design Principles

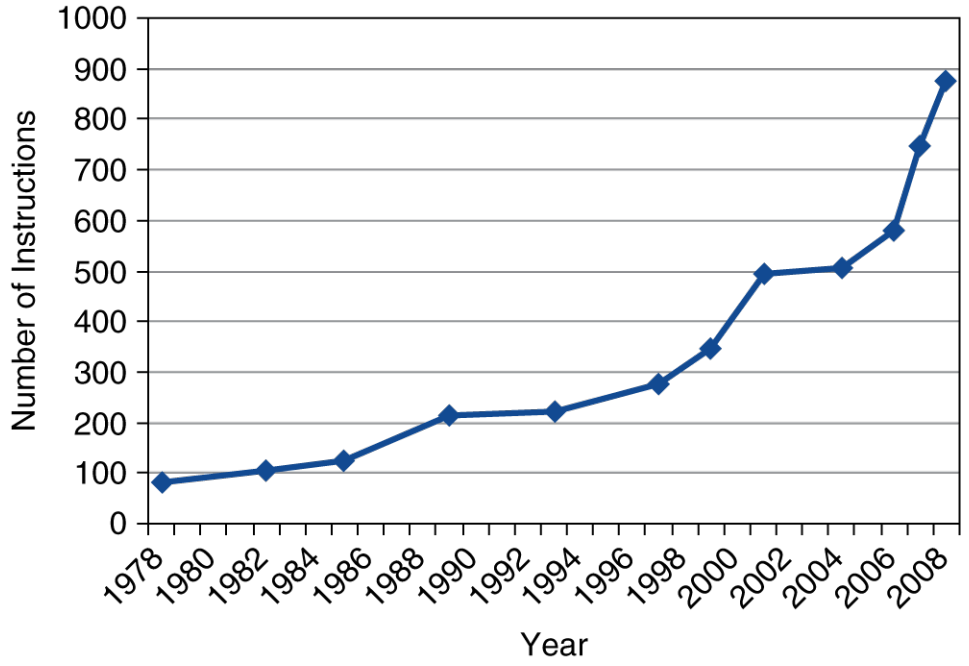
1. Simplicity favors regularity
   - Consistent *instruction size, instruction formats, data formats*
   - Eases implementation by simplifying hardware, leading to higher performance
2. Smaller is faster
   - Fewer bits to access and modify
   - Use the register file instead of slower memory
3. Make 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)

Fallacies

- Powerful instruction ⇒ 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 ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ 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