Computer Systems II

Li Lu
Room 605, CaoGuangbiao Building
li.lu@zju.edu.cn
https://person.zju.edu.cn/lynnluli

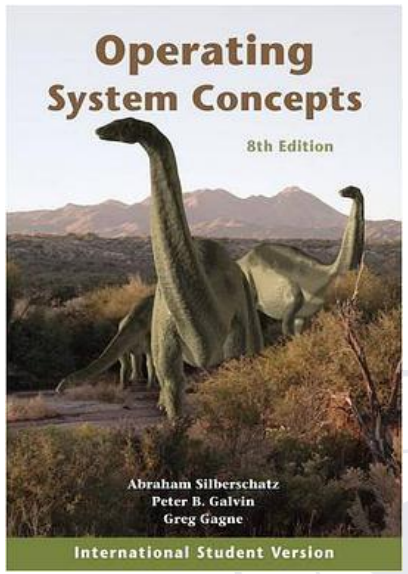
About the Class

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 ~ 2 weeks
- Hazard of Pipeline CPU ~ 2 weeks
- Software/Hardware Interfaces ~ 1 week
- Introduction of OS ~ 2 weeks
- Interrupt ~ 1 week
- Process and Thread ~ 2 weeks
- Scheduling, Synchronization and Deadlock ~ 3 weeks
- Final Review ~ 1 week

Instructor & TA

- Instructors
  - Li Lu 卢立 (li.lu@zju.edu.cn)
  - Yajin Zhou 周亚金 (yajin_zhou@zju.edu.cn)

- TAs
  - Yangye Zhou 周扬叶 (1076192792@qq.com)
  - Jingjing Wang 王晶晶 (3200104880@zju.edu.cn)

Course Organization

- Lectures
  - Monday (Zijingang West 1-504)
    - 2:15 PM – 3:50 PM
  - Thursday (Zijingang West 1-504)
    - 2:15 PM – 3:50 PM

- Labs
  - Monday (Zijingang lab room)
    - 4:15 PM – 5:50 PM (Only ODD weeks)
  - Thursday (Zijingang lab room)
    - 4:15 PM – 5:50 PM (Every week)
  - No group, please work alone

Course Grading

- Homework Assignment 10%
  - TBD
- Projects 60%
  - Lab 0 – CPU Design Review
  - Lab 1 – Pipeline CPU Design with Stall
  - Lab 2 – Hazard and Forwarding
  - Lab 3 – Kernel Boot
  - Lab 4 – Interrupt
  - Lab 5 – Simple Scheduling
  - Lab 6 – Running OS on CPU
- Final Exam 30%

Course Policy

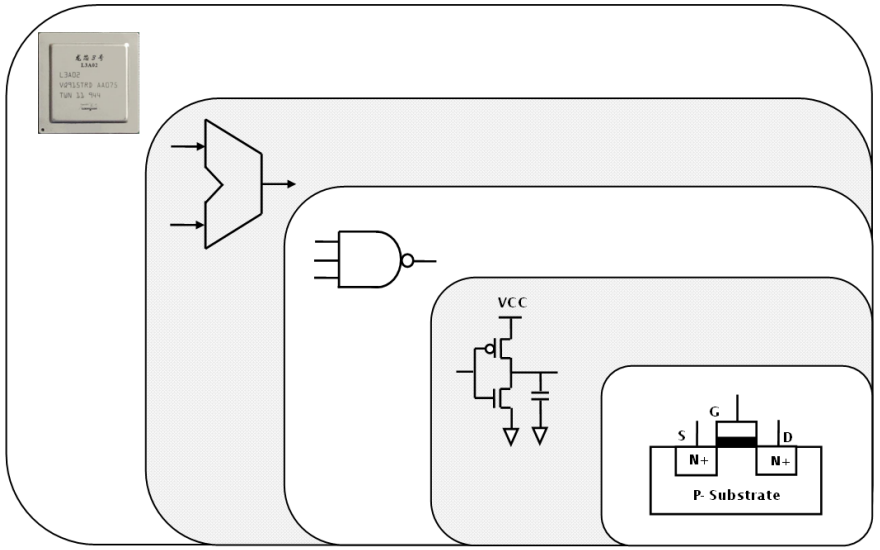
- Academic integrity
  - We will strictly enforce the university, college, and department policies against academic dishonesty
  - Plagiarism in any form will not be tolerated!
- Unless otherwise noted, work turned in should reflect your independent capabilities
  - If unsure, note / cite sources and help
- Late work penalized 5%/day
  - No penalty for documented emergency or by prior arrangement in special circumstances

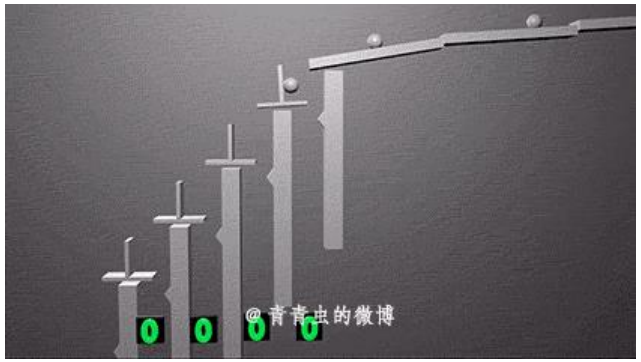
Systems I Review

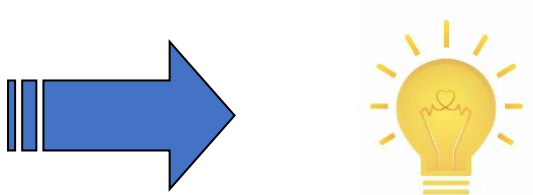
Begin with Answering the Questions

- Why use binary in the computers?
- What is the architecture of computers?
- What is the inside of processor (CPU)?
- Why do? Why using the CPU to solve the problem?
- What are the basic principles of CPU/ISA design?
- How to design a CPU? And how to improve it?

Why use binary in the computers?



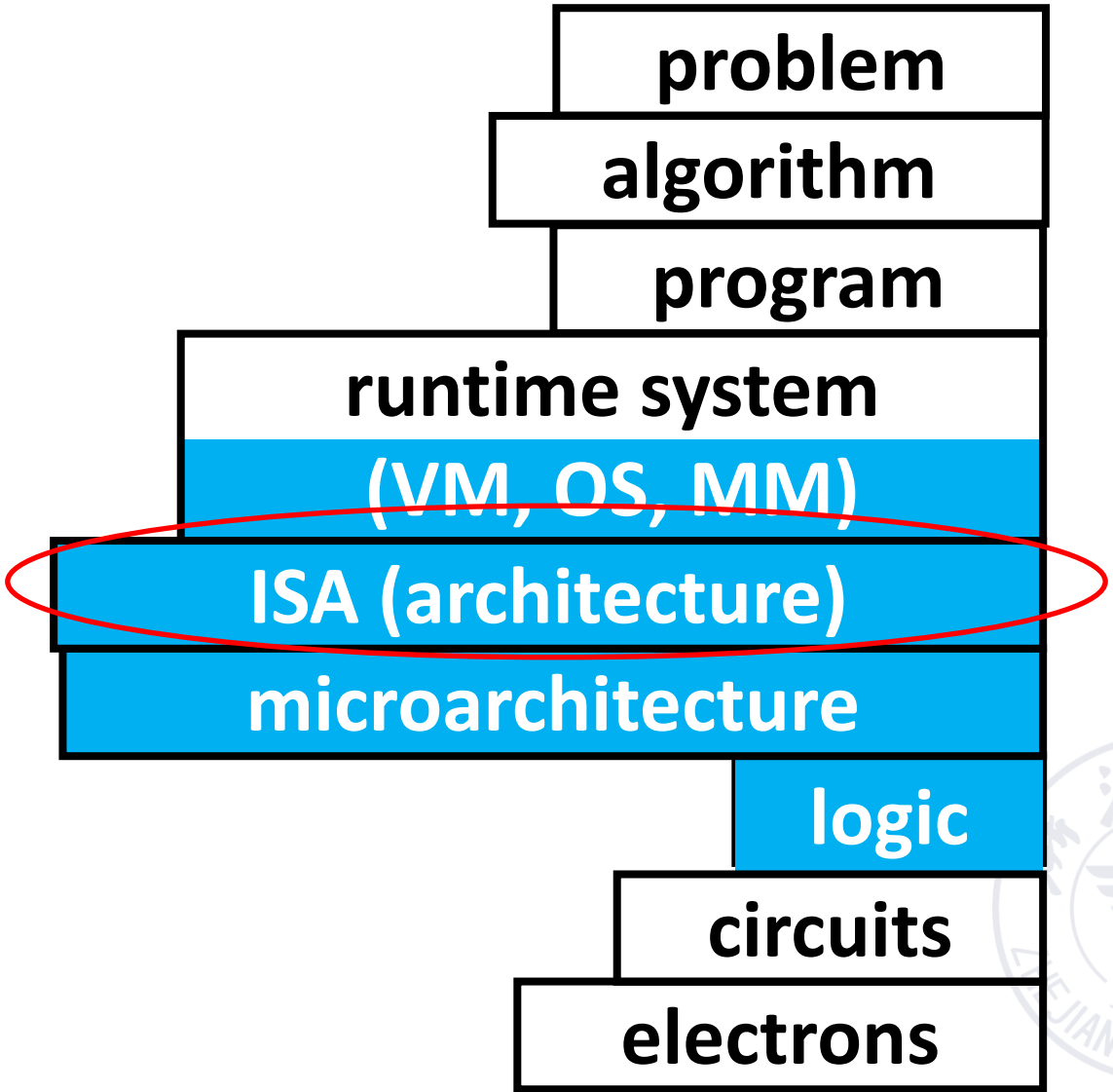




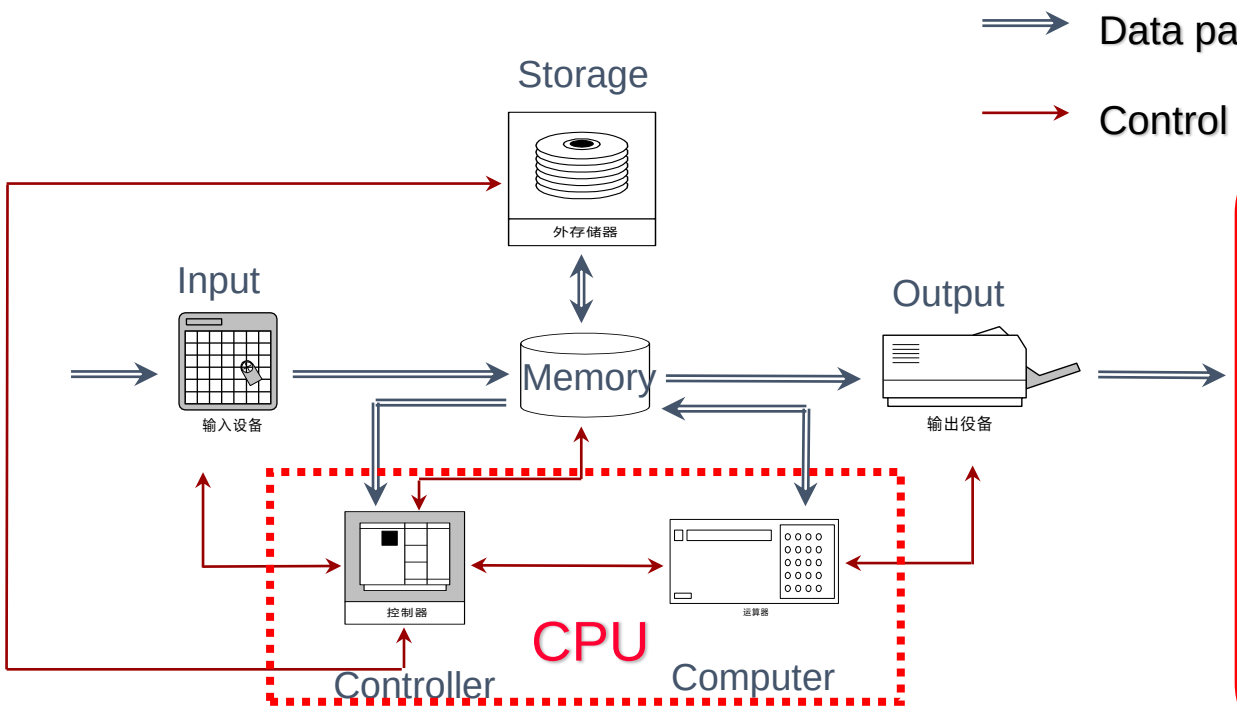
- The computer is composed of electronic components, and binary is the easiest to realize.

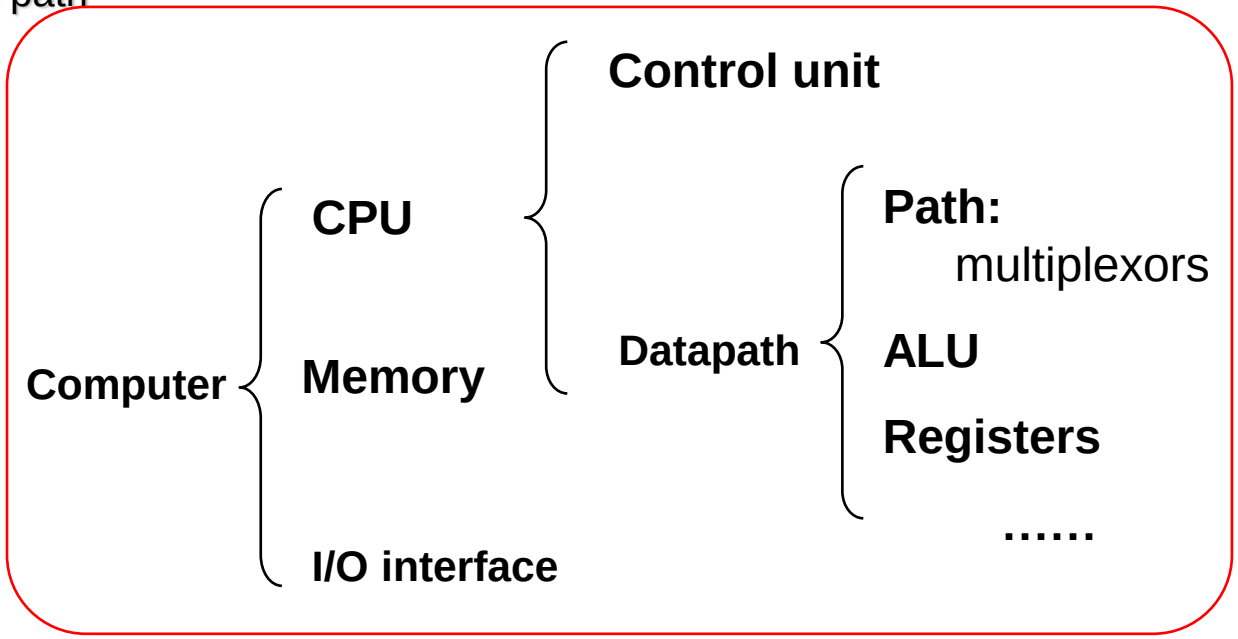
What is the architecture of computers?

Computer Architecture I



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.

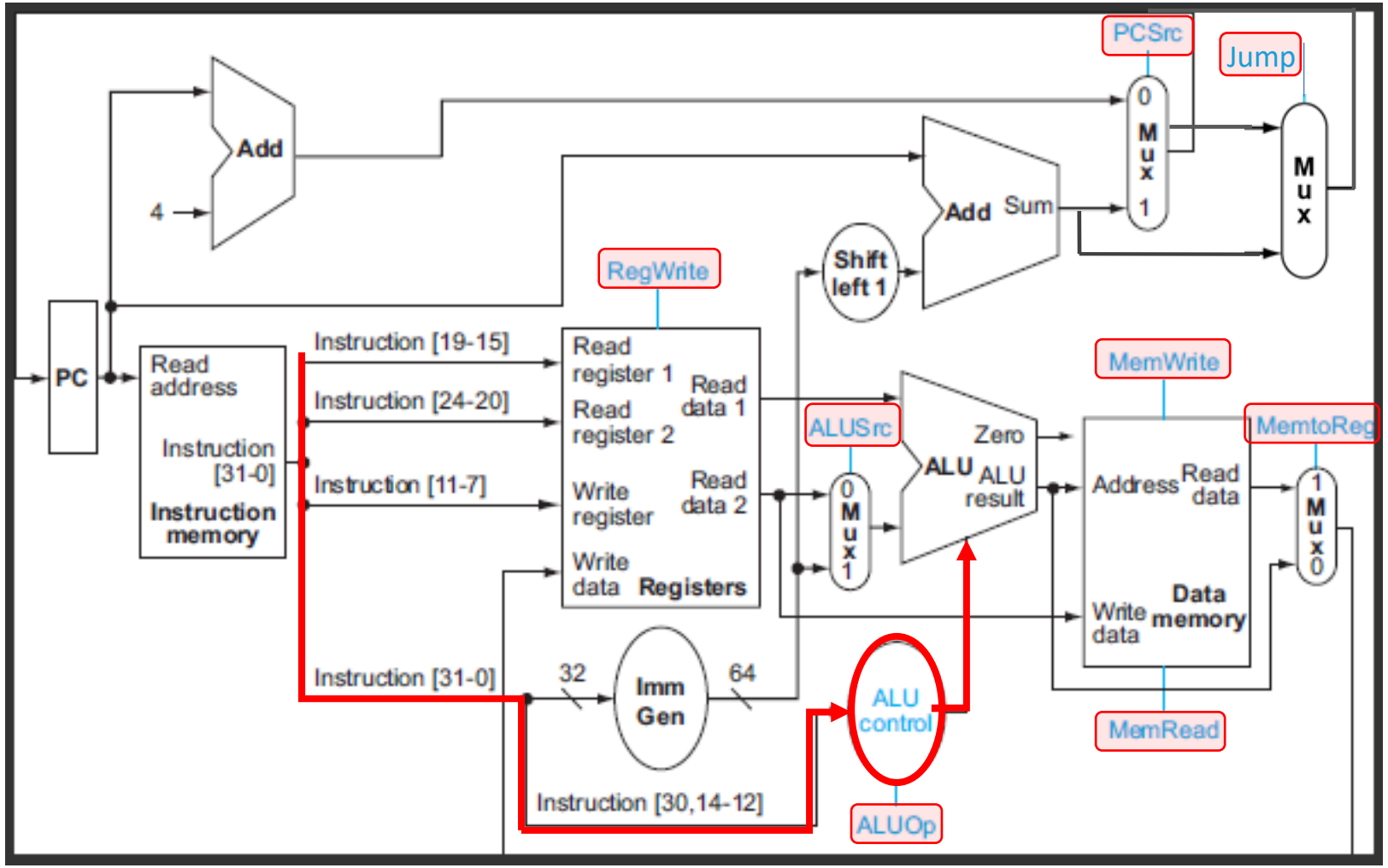
Von Neumann Structure

What is the inside of Processor (CPU)?

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data

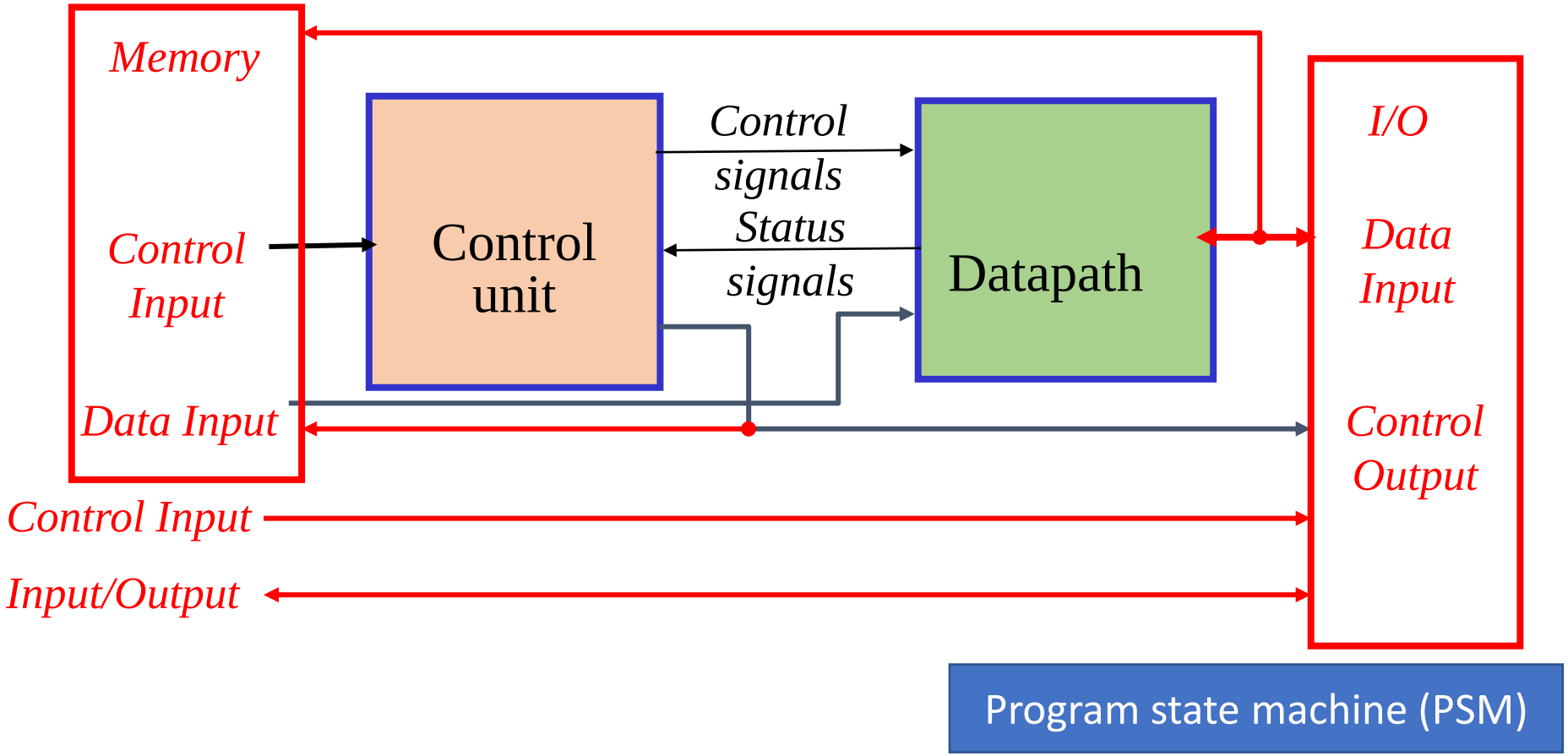
What is the inside of Processor (CPU)?

Output signals



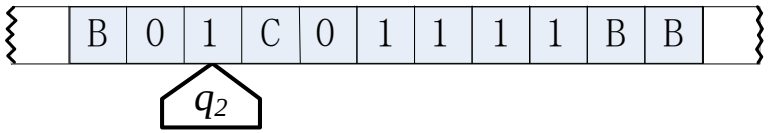
Why do?
Why using the CPU to solve the problem?

- Design methodology : General design



Why do?
Why using the CPU to solve the problem?

- Better way? Turing Machine
  - 1. An infinitely long paper tape TAPE
  - 2. A read/write head HEAD
  - 3. A set of control rules TABLE
  - 4. A state register



Turing Model：Marvin • Minsky (1967)

- Ideal vs. Universal
  - Ideal: cannot implement, because no unlimited tape exists
  - Universal: Long enough tape with head and tail connected, to replace the infinite long paper tape

Why do?
Why using the CPU to solve the problem?

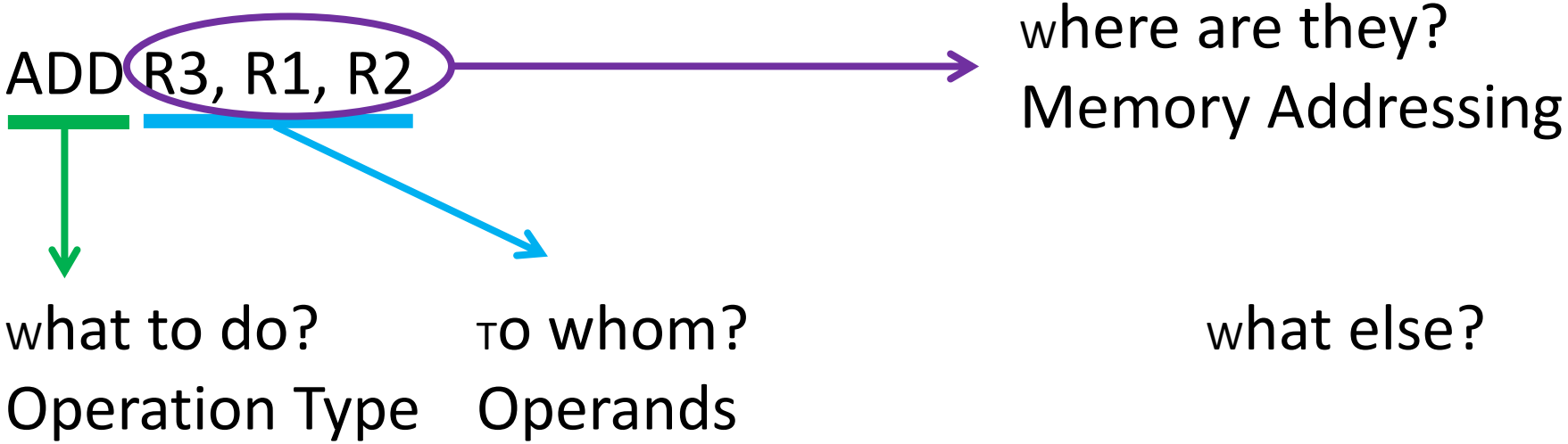
- CPU
  - General digital system
  - A Turing Machine
    - Implemented by Register Transmission Control Technology
      - Datapath
        - The component of processor that performs arithmetic operations
      - Control
        - The component of processor that commands the datapath, memory, and I/O device according to the instructions of the program

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**



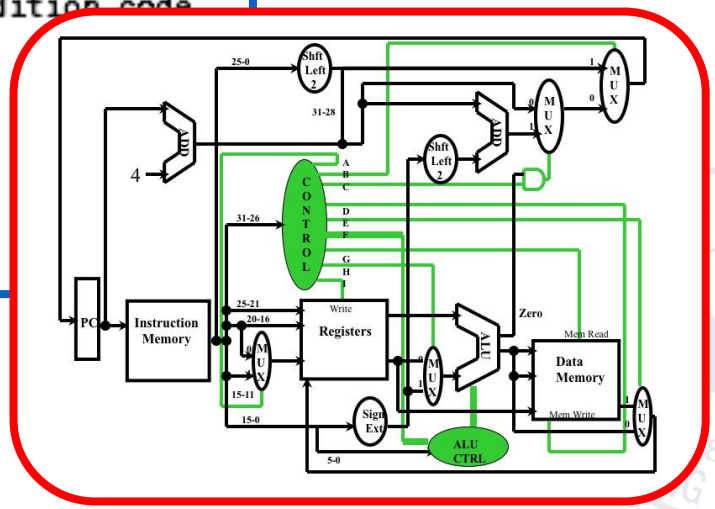
What are the basic principles of CPU/ISA design?

- 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 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)

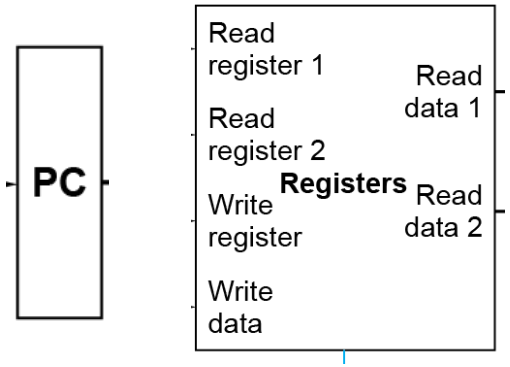
How to Design a CPU?

- 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

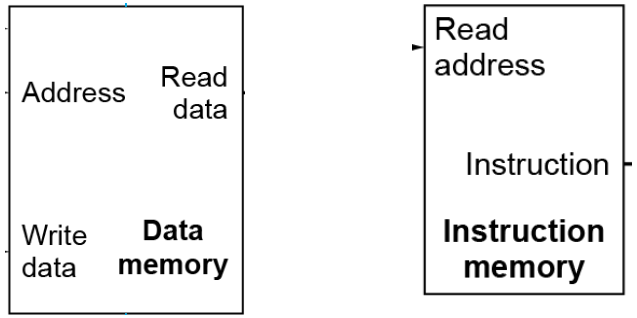
How to Design a CPU?

- Knowing the elements

Registers

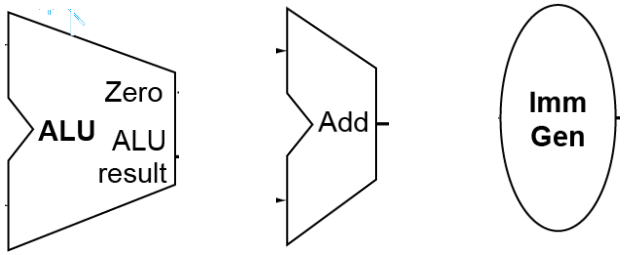


Memories



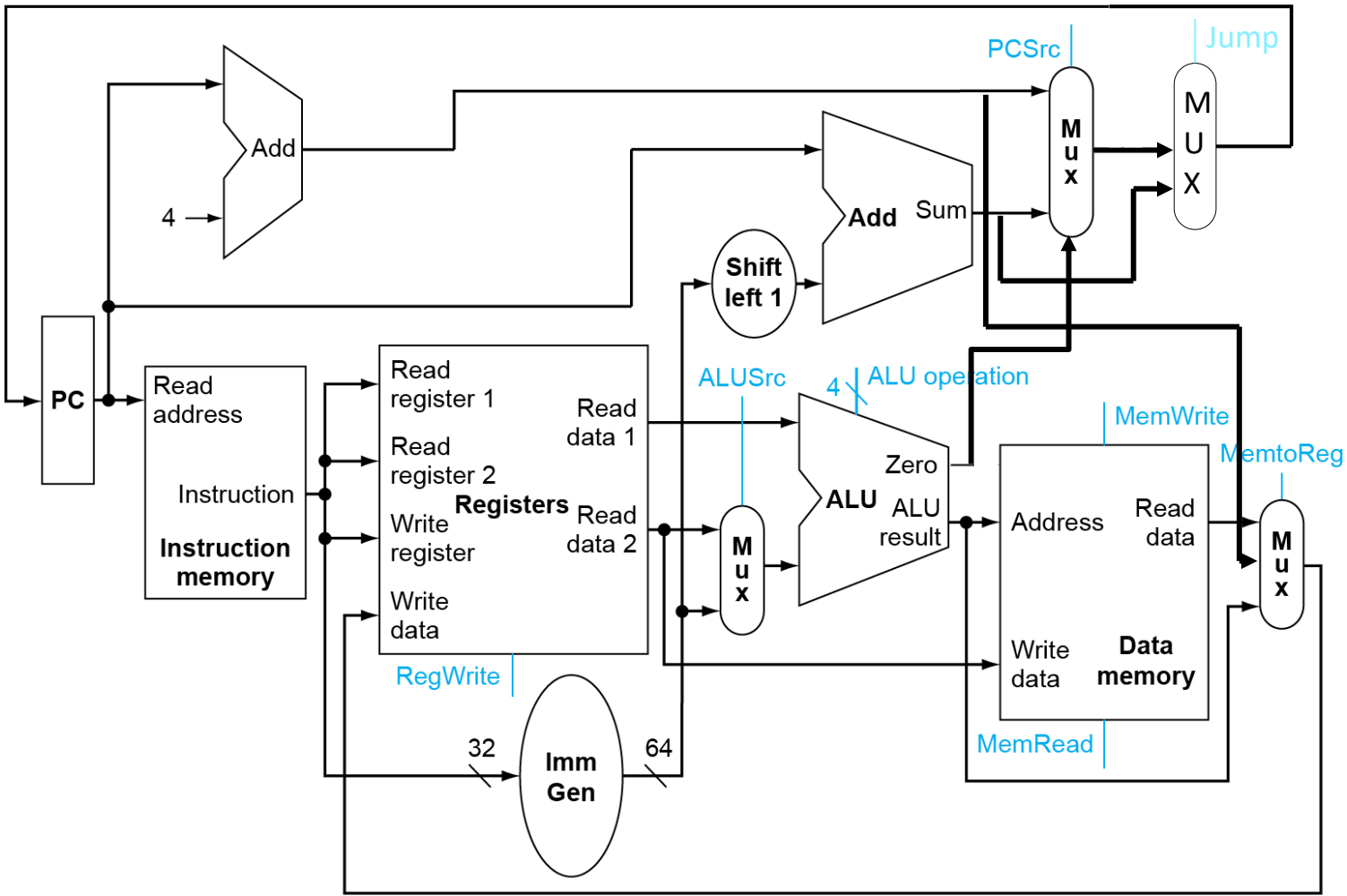
Multiplexer

Arithmetic and Logic



How to Design a CPU?

- Construct the datapath



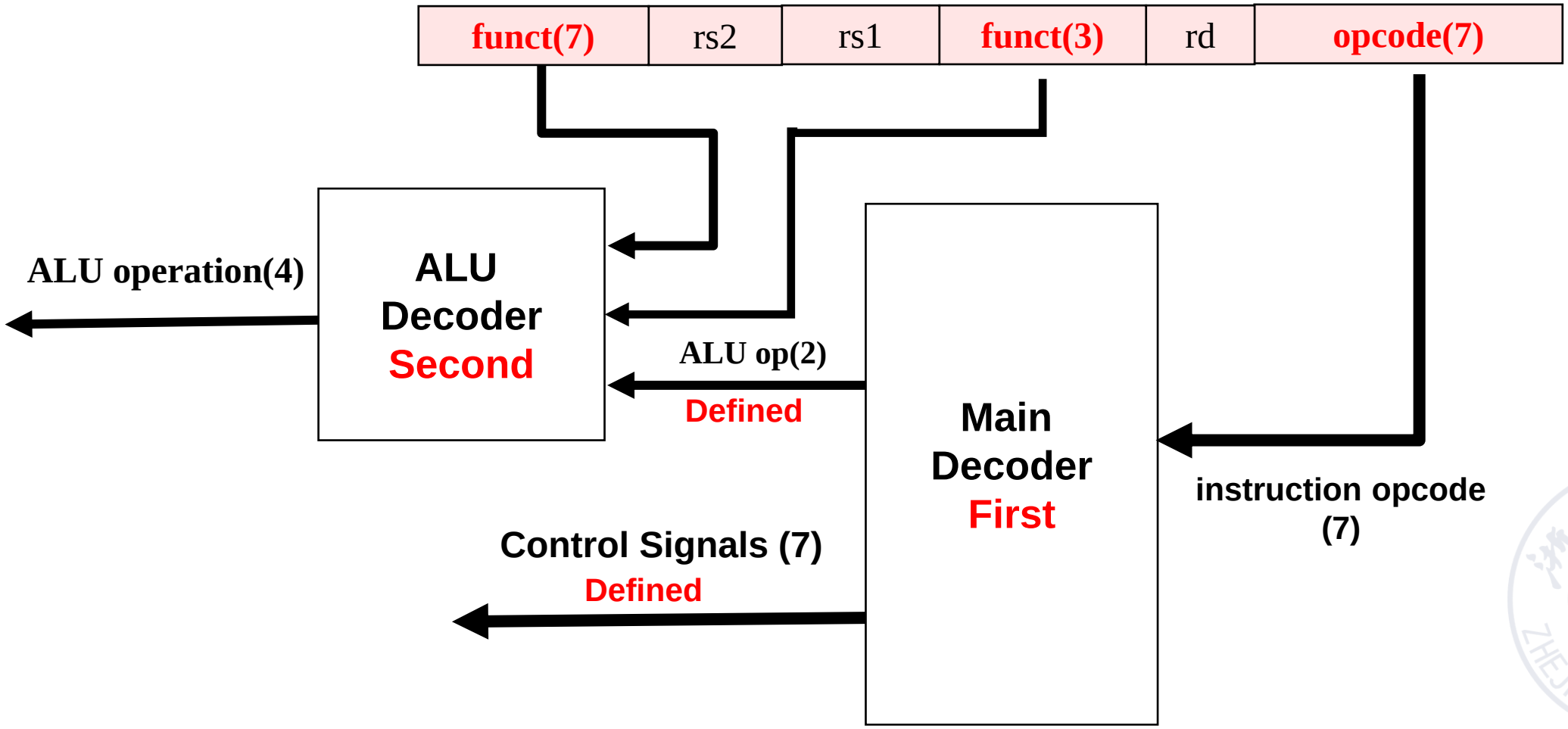
How to Design a CPU?

- 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?

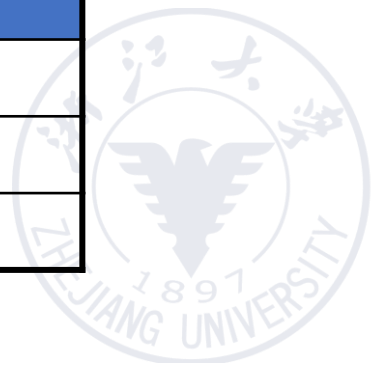
- Construct the controller



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
  - Make instructions that *do* more (CISC)
  - User better compilers
- Use less cycles to perform the 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

Basic Principles of Multi-Cycle CPU Design

- Separate the execution of instructions into several stages with the same period
  - Hard to achieve the same period, thus aims to be *almost the same*
- Each stage occupies one clock cycle
- Each clock cycle at most completes a memory access, register access, or ALU operation
- Execution result of the previous clock cycle needs to be stored in specific sequential logic components
- Clock cycle period depends on the most complex operation

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***

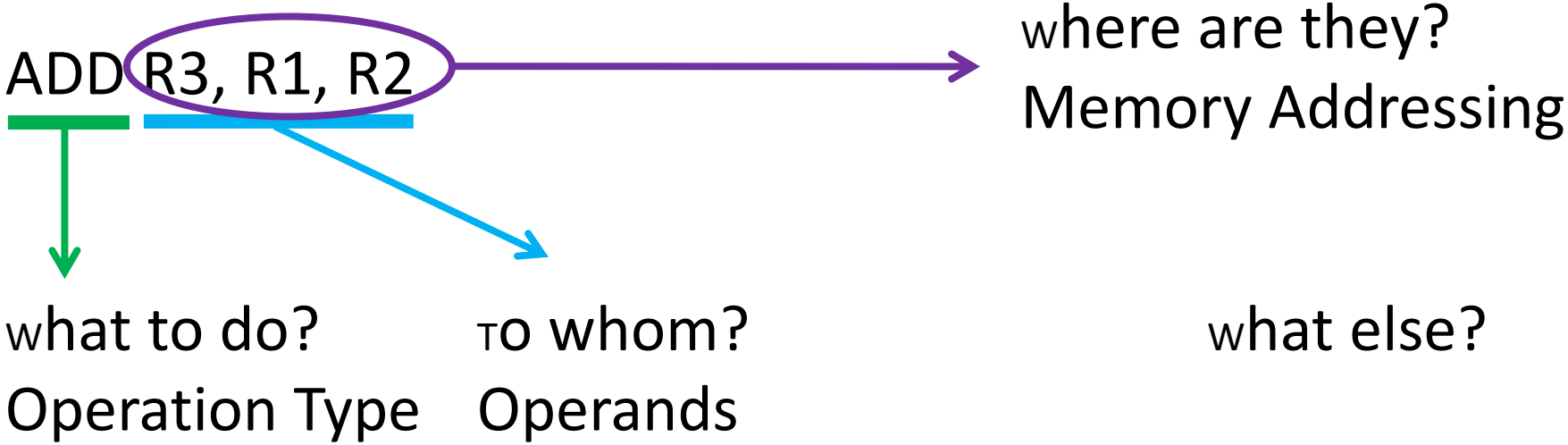
Instruction Set Principles

What are the basic principles of 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 ISA design?

- Instruction Set **Architecture**



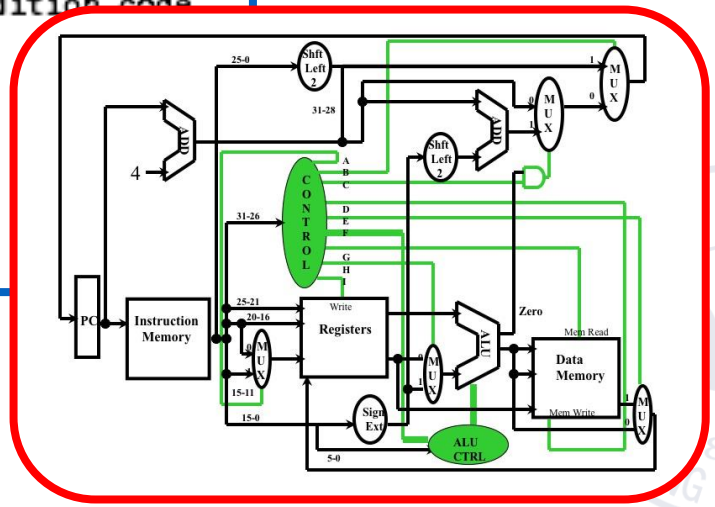
What are the basic principles of ISA design?

- 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

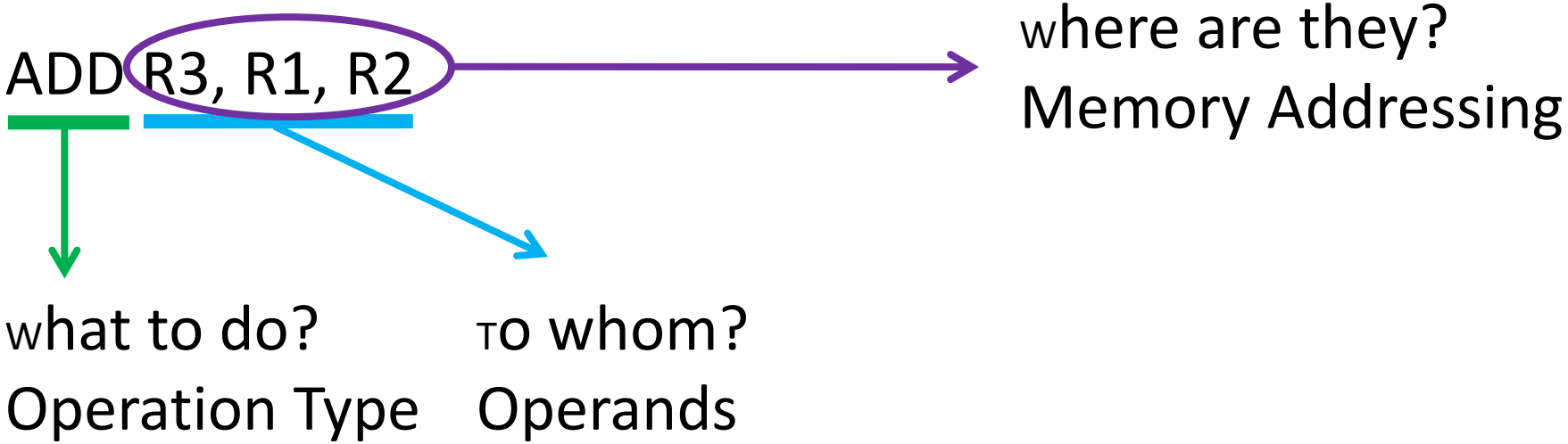
```
        LOAD    R1,&a       ; R1 <-- contents of 'a'
        LOAD    R2,&a       ; R2 <-- contents of 'a'
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JNE     @L1         ; goto L1 if not equal
        ADD     R1,R2       ; R1 <-- R1 + R2
        TEST    R1,R2       ; 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
```



What are the basic principles of ISA design?

- Compatibility
- Versatility
- High efficiency
- Security

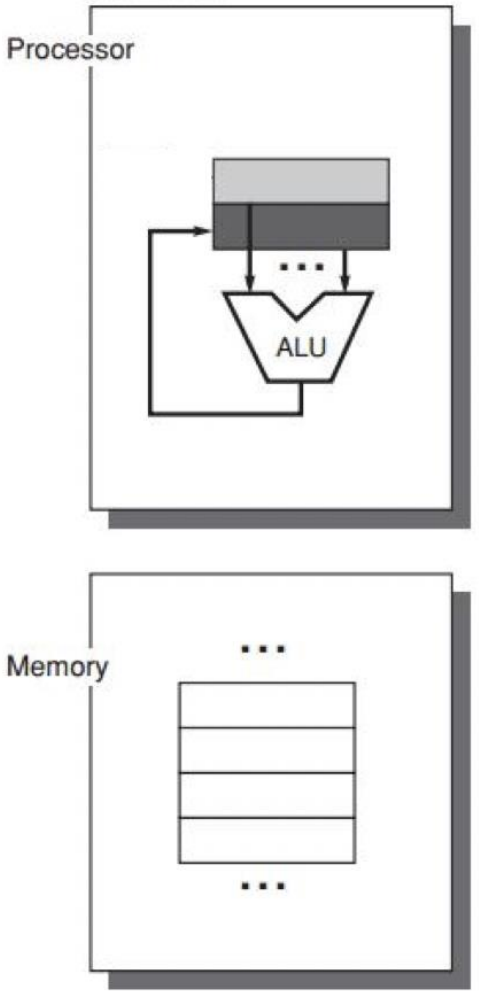
ISA Classification Basis



ISA Classification Basis

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

In processor, stores data fetched from memory or cache



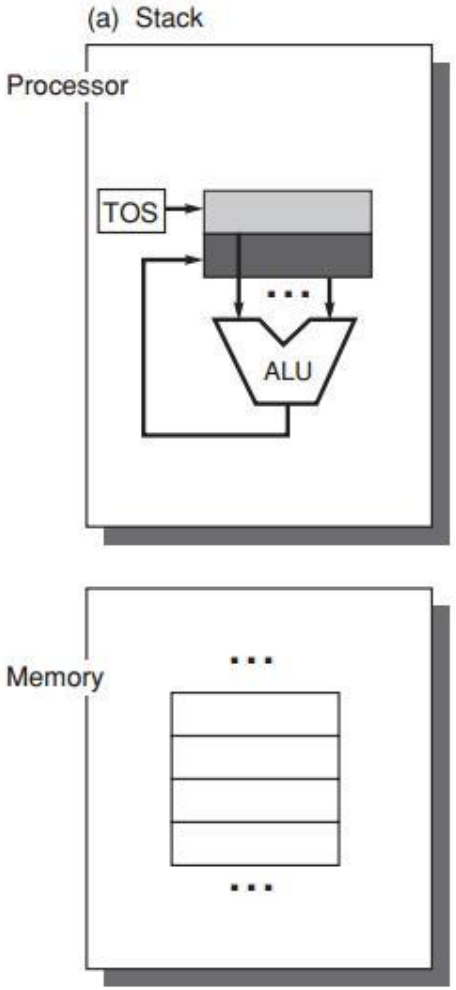
ISA Classes

- 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)

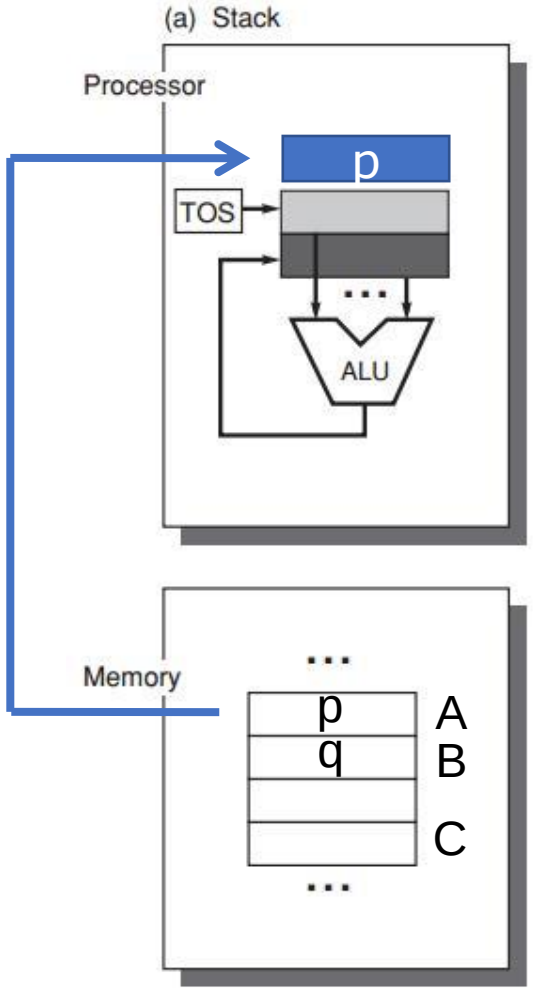
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

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

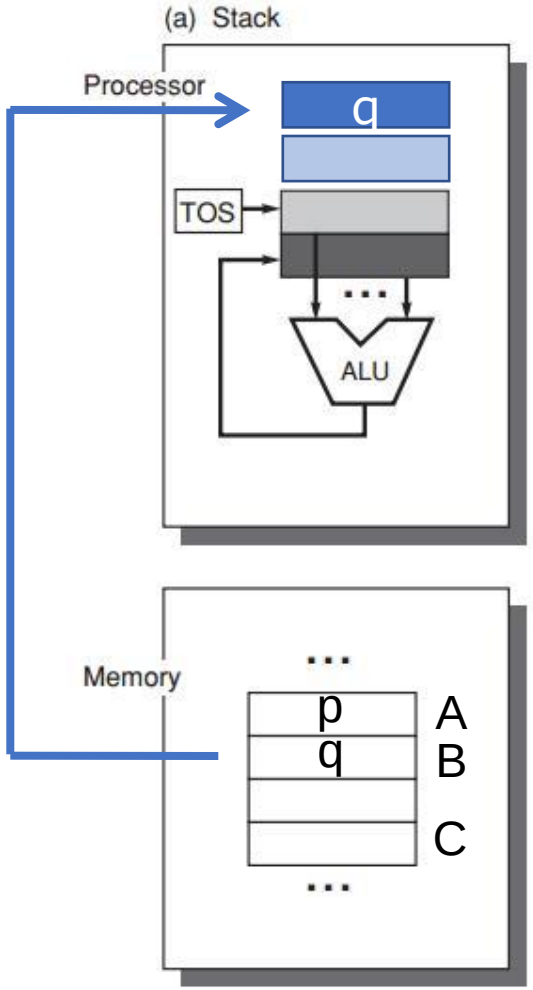
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

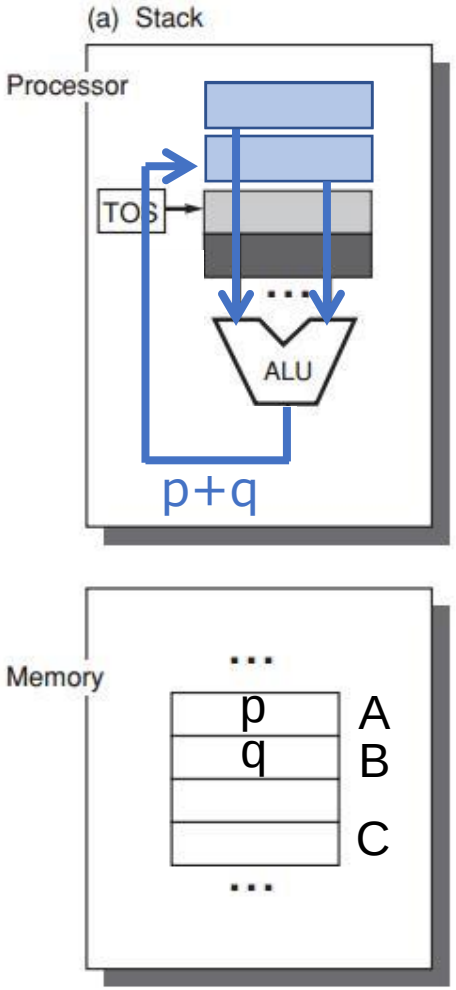
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

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



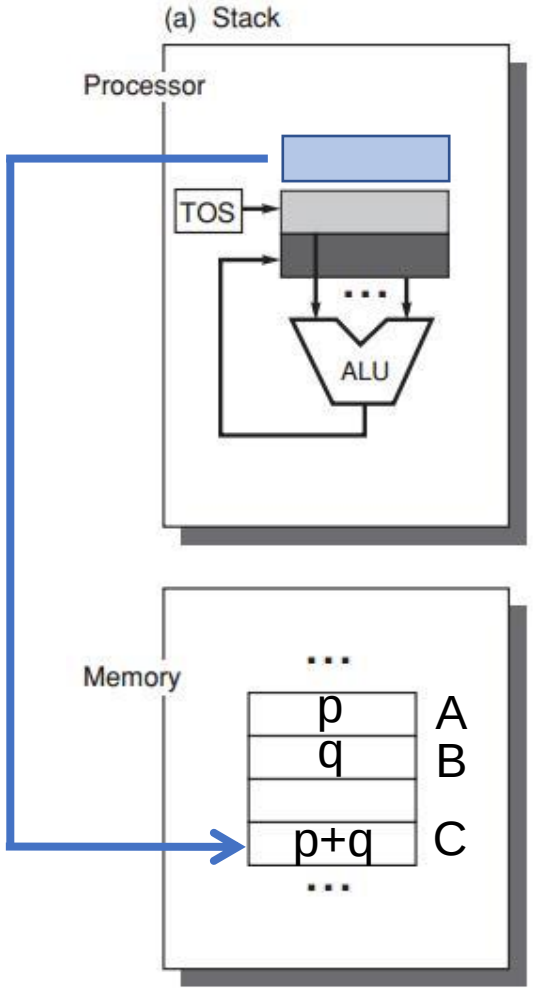
ISA Classes: Stack Architecture

- 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

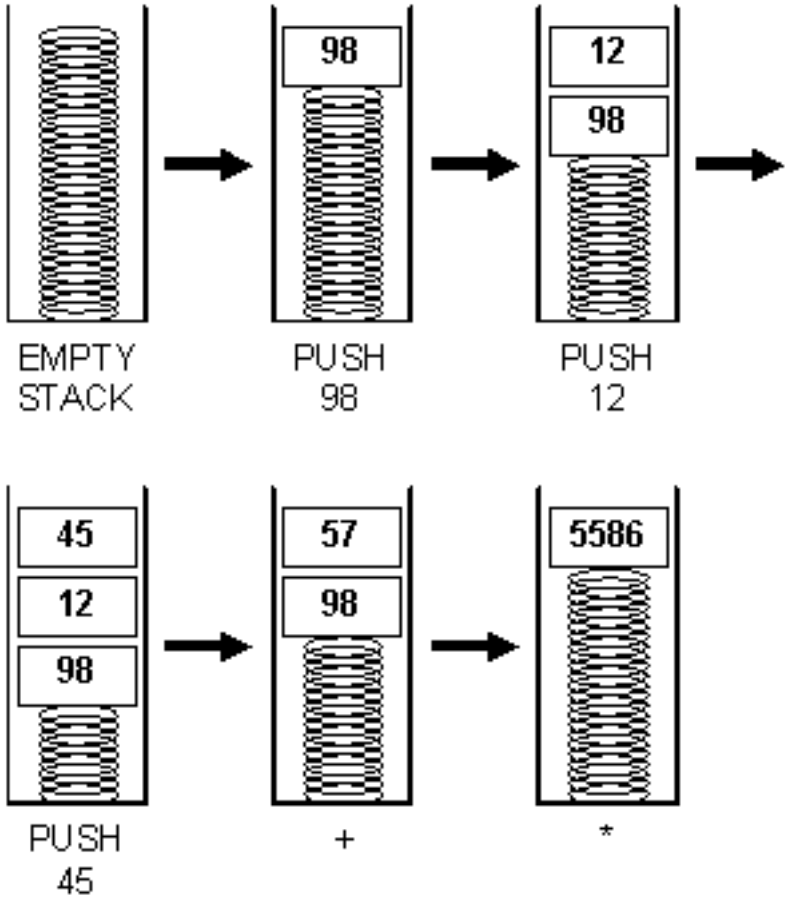
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)
- First operand removed from
  second op replaced by the result
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

- Example:

98 * (12 + 45)



ISA Classes: Accumulator Architecture

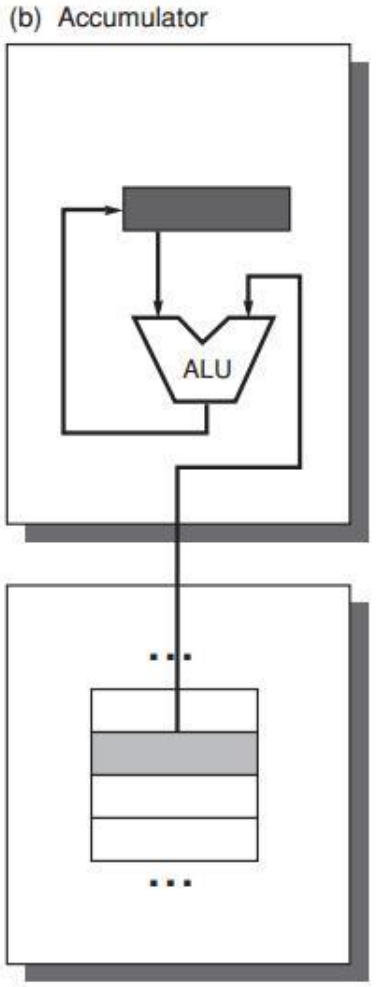
- One implicit operand: the accumulator

  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

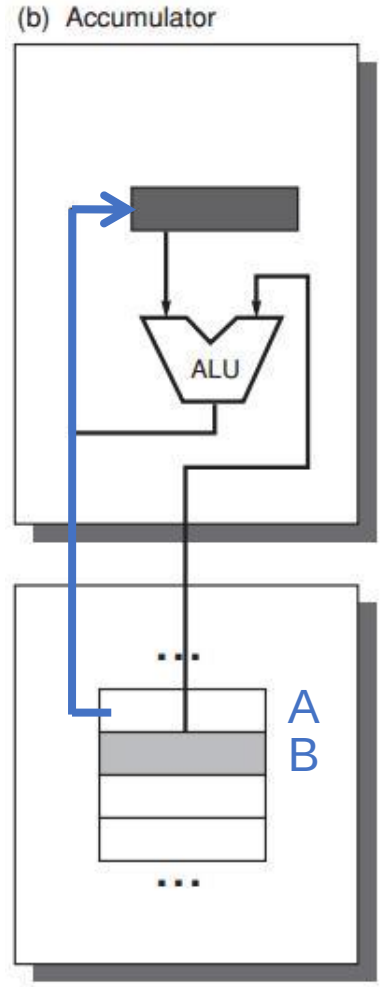
- One implicit operand: the accumulator

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

  Load A

  Add B

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



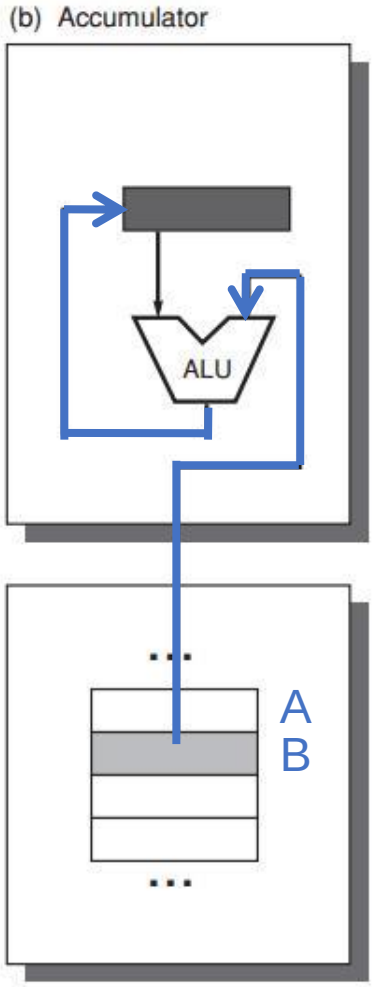
ISA Classes: Accumulator Architecture

- 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

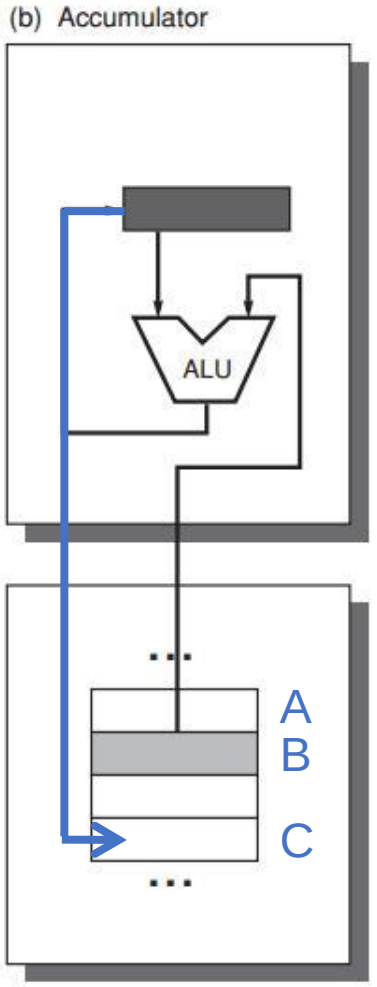
- One implicit operand: the accumulator

  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