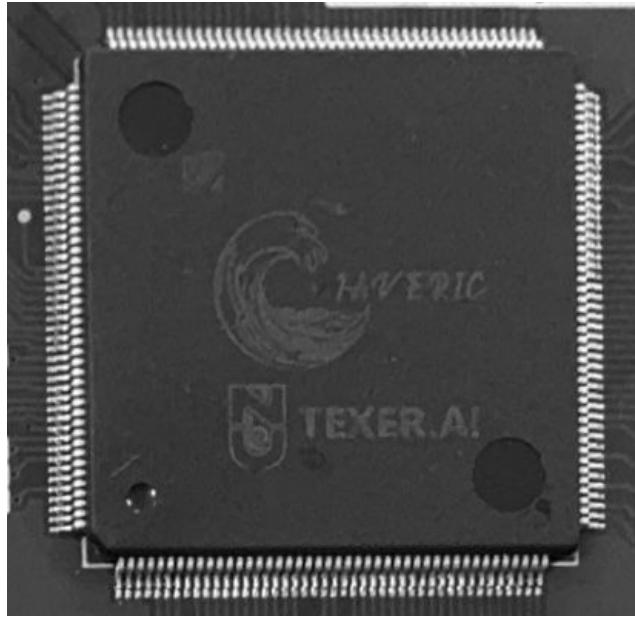


# Processor, Instruction & Program

Xiaoke Su

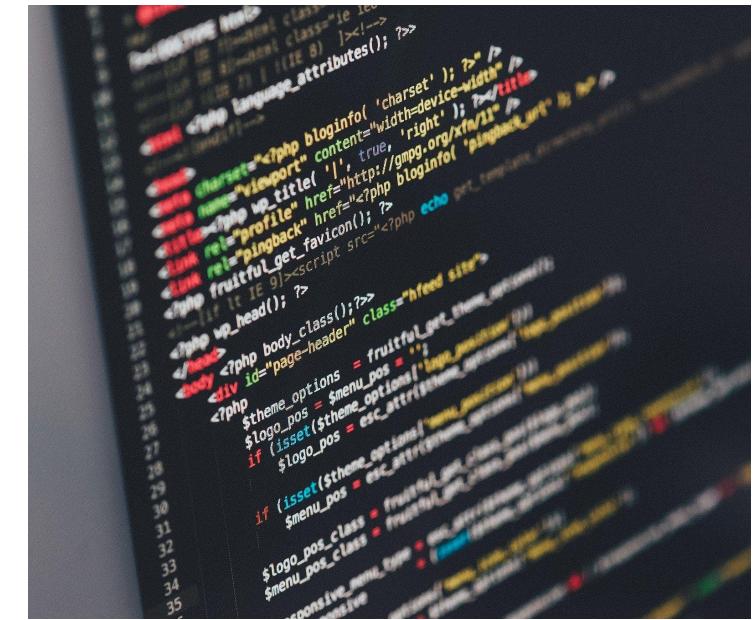
Dec 17, 2025



Processor



Instruction



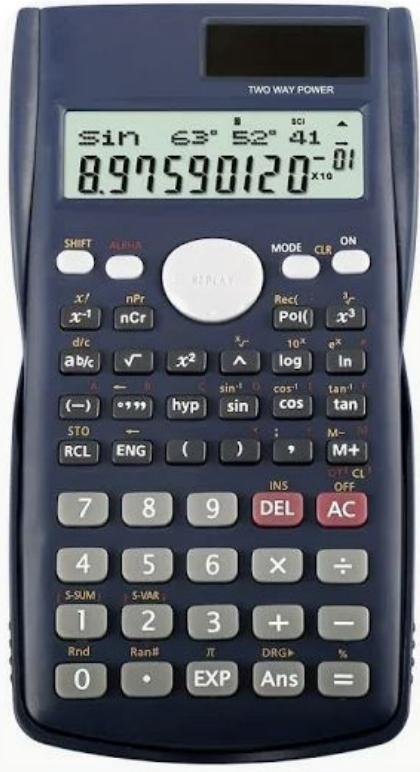
Program

# Contents

- Instruction
- Case Study – simple ISA(sISA)
- State Machine
- C Programming Language
- Conclusion: Relationship among Programs, ISAs, and CPUs

# Instruction

# Instruction



- Calculator could calculate the results by pressing buttons, like  $1+2=$
- CPU is much further...
- CPU is controlled by **instructions**
  - *add* could ask CPU to add two data

# Instruction

- Instructions provide all the information the CPU needs, to get the actions done:
  - data(to be processed): operand
  - how to process the data(add? load? store?): opcode



# Register

- Some complex processing needs several steps
  - E.g. calculating  $1+2+3+\dots+10$ , we need to calculate  $1+2$  first, and then plus 3 with the last result
- Store temporary result: register
- We need more than 1 registers, called General Purpose Register(GPR)
  - Register Set, Register File
- Instructions should let CPU know:
  - Read data from which GPR(source operand)
  - Store result in which GPR(destination operand)

# Instruction

- Instruction = menu encoded by binary numbers
- Now we define a simple CPU(sCPU)
  - 4 GPRs, supporting 3 instructions,
  - of which 2 instructions are:

7	6	5	4	3	2	1	0			
+	-	-	-	-	-	-	-			
	00		rd		rs1		rs2		R[rd]=R[rs1]+R[rs2]	add instruction, add two registers together
+	-	-	-	-	-	-	-			
	10		rd		imm			R[rd]=imm	li instruction, load immediate value, fill high bits with zeros	
+	-	-	-	-	-	-	-			

7	6	5	4	3	2	1	0
+---+---+---+---+							
00   rd   rs1   rs2   R[rd]=R[rs1]+R[rs2]	add instruction, add two registers together						
+---+---+---+---+							
10   rd   imm   R[rd]=imm	li instruction, load immediate value, fill high bits with zeros						
+---+---+---+---+							

- 3 instructions ---> at least 2 bits opcode
- 4 registers ---> 2 bits for locating GPR (GPR index)
- R[rd] means the value of register rd
- imm: immediate number/value

# Computer: Stored-Program

- In computer perspective: program = instruction list
- The biggest difference between computer and calculator
  - program will automatically control the execution of computer
- To let computer know what is the next instruction, we need a Program Counter(PC)
- Then, Computer will just repeat the following steps:
  - Fetch the instruction from the memory location indicated by PC
  - Execute the instruction
  - Update the PC

# Branch Instruction: to modify PC

- As PC stores the location of current instruction, PC should also be a register (but not belong to GPR)
- Design some instructions to modify PC, in order to improve program flexibility
  - bner0 (branch if NOT equal r0)

7	6	5	2	1	0
+----+	-----+----+				
11	addr	rs2	if (R[0] != R[rs2])	PC=addr	bner0 instruction, if not equal to R[0], jump
+----+	-----+----+				

# Instruction Set Architecture(ISA)

- **GPRs, PC, memory, instructions, and their executions** all belong to **Instruction Set Architecture (ISA)**
- **x86, ARM, and RISC-V** are **different ISAs**
- **ISA: a set of specifications**(written in manuals), which defines the **functionality and behavior of a model machine**
- A **model machine** is a machine that exists only in abstraction. It only focuses on **what the machine does and how it behaves**, without discussing **how it is implemented**.
- When model machine is **implemented using digital circuits**, we can get a **real computer**.

# Now we have a simple ISA (sISA)

# Example

- 0: li r0, 10 # 10 in decimal
- 1: li r1, 0
- 2: li r2, 0
- 3: li r3, 1
- 4: add r1, r1, r3
- 5: add r2, r2, r1
- 6: bner0 r1, 4
- 7: bner0 r3, 7

Instruction	Opcode	Meaning
add rd, rs1, rs2	00	add rs1+rs2 together, and write to rd
li rd, imm	10	load the immediate value imm to rd, fill high bits with zeros
bner0 rs2, addr	11	if rs2 not equal to R0, PC jumps to addr

**Q: Give values of PC and 4 registers(PC, R0, R1, R2, R3) after EACH instruction**

PC	R0	R1	R2	R3	Notes
0	0	0	0	0	initial state
1	10	0	0	0	After executing PC = 0
?	?	?	?	?	?

PC	R0	R1	R2	R3	Notes
0	0	0	0	0	initial state
1	10	0	0	0	After executing the instruction with PC = 0, r0 is updated to 10, and PC is updated to the location of the next instruction.
2	10	0	0	0	After executing the instruction with PC = 1, r1 is updated to 0, and PC is updated to the location of the next instruction.
3	10	0	0	0	After executing the instruction with PC = 2, r2 is updated to 0, and PC is updated to the location of the next instruction.
4	10	0	0	1	After executing the instruction with PC = 3, r3 is updated to 1, and PC is updated to the location of the next instruction.
5	10	1	0	1	After executing the instruction with PC = 4, r1 is updated to r1 + r3, and PC is updated to the location of the next instruction.
6	10	1	1	1	After executing the instruction with PC = 5, r2 is updated to r2 + r1, and PC is updated to the location of the next instruction.
4	10	1	1	1	After executing the instruction with PC = 6, since r1 does not equal r0, PC is updated to 4.
5	10	2	1	1	After executing the instruction with PC = 4, r1 is updated to r1 + r3, and PC is updated to the location of the next instruction.
...	...	...	...	...	...

## Example 2

- Write a program with the 3 instructions(add, li, bne r0), to calculate the sum of all **odd** numbers from 1 to 10. Give all the states of PC and registers.

# Example 2

- 0: li r0, 11 # 11 in decimal
- 1: li r1, 0
- 2: li r2, 1
- 3: li r3, 2
- 4: add r1, r1, r2
- 5: add r2, r2, r3
- 6: bner0 r2, 4
- 7: bner0 r3, 7

PC	R0	R1	R2	R3
0	0	0	0	0
1	9	0	0	0
2	9	0	0	0
3	9	0	1	0
4	9	0	1	2
5	9	1	1	2
6	9	1	3	2
4	9	1	3	2
5	9	4	3	2
...	...	...	...	...

# Review the program calculating sum of 1-10

```
10001010      # 0: li r0, 10
10010000      # 1: li r1, 0
10100000      # 2: li r2, 0
10110001      # 3: li r3, 1
00010111      # 4: add r1, r1, r3
00101001      # 5: add r2, r2, r1
11010001      # 6: bne r0 r1, 4
11011111      # 7: bne r0 r3, 7
```

# li rd, imm

7	6	5	4	3	2	1	0
+-----+-----+-----+-----+							
10		rd		imm		R[rd]=imm	
+-----+-----+-----+-----+							

li instruction, load immediate number, fill high bits with zeros

## opcod

10001010	# 0: li r0, 10
10010000	# 1: li r1, 0
10100000	# 2: li r2, 0
10110001	# 3: li r3, 1

# li rd, imm

7	6	5	4	3	2	1	0
+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+
10   <b>rd</b>        imm        R[rd]=imm							
+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+	+-----+-----+-----+

li instruction, load immediate number, fill high bits with zeros

address of

rd

10001010	# 0: li r0, 10
10010000	# 1: li r1, 0
10100000	# 2: li r2, 0
10110001	# 3: li r3, 1

# li rd, imm

7	6	5	4	3	2	1	0
+	-	-	-	-	-	-	+
	10		rd		imm		R[rd]=imm
+	-	-	-	-	-	-	+

li instruction, load immediate number, fill high bits with zeros

imm

value

10001010	# 0: li r0, 10
10010000	# 1: li r1, 0
10100000	# 2: li r2, 0
10110001	# 3: li r3, 1

# add rd, rs1, rs2

7	6	5	4	3	2	1	0	
00		rd		rs1		rs2		R[rd]=R[rs1]+R[rs2]

add instruction, add two registers together

## opcod

e

00010111	# 4: add r1, r1, r3
00101001	# 5: add r2, r2, r1

# add rd, rs1, rs2

7	6	5	4	3	2	1	0
+----+----+----+----+							
00   rd   rs1   rs2   R[rd]=R[rs1]+R[rs2]	add instruction, add two registers together						
+----+----+----+----+							

address of  
**rd**

00010111	# 4: add r1, r1, r3
00101001	# 5: add r2, r2, r1

# add rd, rs1, rs2

7	6	5	4	3	2	1	0
+	-	-	-	-	-	-	-
	00		rd		rs1		rs2
	R[rd]=R[rs1]+R[rs2]						

add instruction, add two registers together

address of  
rs1

00010111	# 4: add r1, r1, r3
00101001	# 5: add r2, r2, r1

# add rd, rs1, rs2

7	6	5	4	3	2	1	0				
+	-	-	-	-	-	-	-				
	0	0		rd		rs1		rs2		R[rd]=R[rs1]+R[rs2]	add instruction, add two registers together
+	-	-	-	-	-	-	-	-	-	-	-

address of  
rs2

00010111	# 4: add r1, r1, r3
00101001	# 5: add r2, r2, r1

# bner0 rs2, addr

7	6	5	4	3	2	1	0
+-----+-----+-----+-----+							
11							
+-----+-----+-----+	addr		rs2		if (R[0]!=R[rs2]) PC=addr	bner0	instruction, if not equal to R[0], then jump
+-----+-----+-----+							

**opcod**

e

11010001	# 6: bner0 r1, 4
11011111	# 7: bner0 r3, 7

# bner0 rs2, addr

7	6	5	4	3	2	1	0
11	addr		rs2	if (R[0]!=R[rs2]) PC=addr bner0 instruction, if not equal to R[0], then jump			
+-----+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+

**add**

**r**

11010001	# 6: bner0 r1, 4
11011111	# 7: bner0 r3, 7

# bner0 rs2, addr

7	6	5	4	3	2	1	0	
11	addr		rs2		if (R[0]!=R[rs2]) PC=addr bner0 instruction, if not equal to R[0], then jump			
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	

address of  
rs2

11010001	# 6: bner0 r1, 4
11011111	# 7: bner0 r3, 7

# CPU: continuously executes instructions

- The “stupidity” of a computer
  - A computer mechanically updates register states **strictly according to the meaning of instructions**
- The “intelligence” of a computer
  - Its extremely **high execution speed**

# CPU: continuously executes instructions

- Example: a 2 GHz CPU
  - A 2 GHz CPU can perform **2,000,000,000 operations per second.**
  - The program that computes  **$1 + 2 + \dots + 10$  requires fewer than 40 instructions.**
  - The CPU therefore spends **less than 0.00000002 seconds**, that is, **20 ns**.
- A human calculating with a calculator needs **22 key presses**.
- Even at **10 key presses per second**, this takes **about 2 seconds**.

# Review the program

- *add, li*: assembly language (symbolic representation of instructions)
- CPU reads instructions by machine language: they are in binary forms

- Assembly Language and Machine Language could be translated into each other

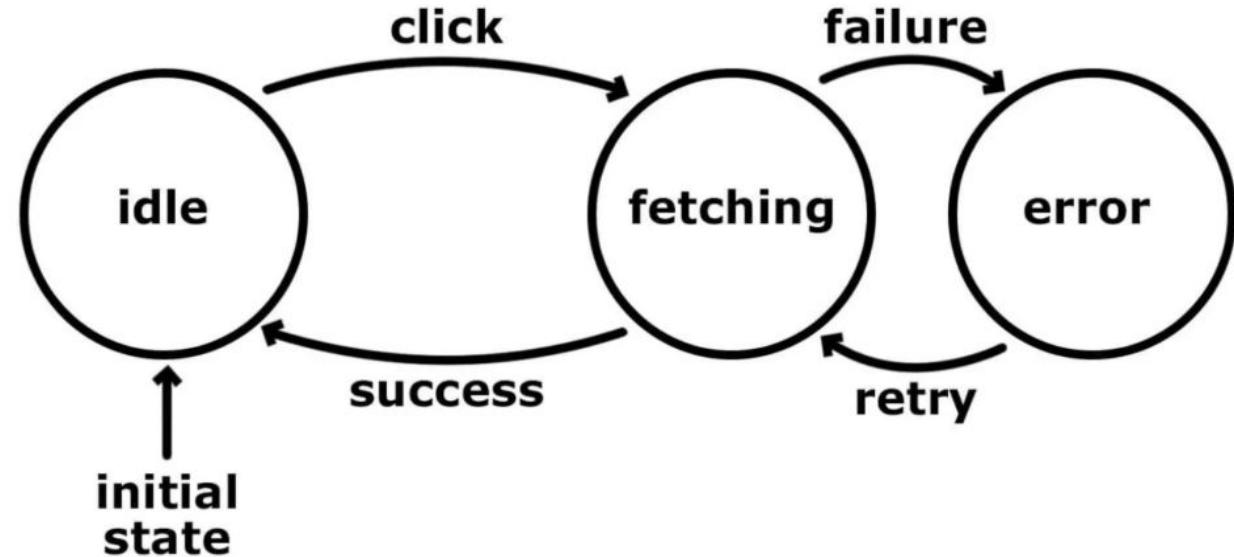
- Machine language: **difficult to understand**
- Assembly language: **more readable, explicitly represents the opcode and operands of each instruction**

```
10001010    # 0: li r0, 10
10010000    # 1: li r1, 0
10100000    # 2: li r2, 0
10110001    # 3: li r3, 1
00010111    # 4: add r1, r1, r3
00101001    # 5: add r2, r2, r1
11010001    # 6: bner0 r1, 4
11011111    # 7: bner0 r3, 7
```

# State Machine

# State Machine

- The definition of a state machine includes the following parts:
- State set  $S = \{S_1, S_2, \dots\}$
- Event set  $E$
- State transition rules  $\text{next}: S \times E \rightarrow S$ 
  - the binary function  $\text{next}(S, E)$  specifies the next state after receiving stimulus event  $E$  while in state  $S$
- Initial state  $S_0 \in S$



# The State Machine Model of the ISA

- State set  $S = \{ (PC, R, M) \}$ 
  - $R = GPR$
  - $M = \text{Memory}$
- Set of events  $E = \{ \text{Instructions} \}$ 
  - Execute the instruction pointed to by the PC and update the state
- State transition rules  $\text{next}: S \times E \rightarrow S$ 
  - the semantics of the instruction
  - specify how the state should change after executing a certain instruction, thereby transitioning from one state to another
- Initial state  $S_0 = (PC_0, R_0, M_0)$

# Example: State Machine of sISA

- State set  $S = \{ (PC, R, M) \} = \{ (PC, r0, r1, r2, r3, M) \}$
- For example, in the sequence summation example, a certain state  $S_k = (5, 10, 1, 0, 1, [10001010, 10010000, 10100000, 10110001, 00010111, 00101001, 11010001, 11011111])$ 
  - [] denotes the information stored in memory, which is the encoding of the instruction sequence mentioned earlier
  - since instructions in sISA do not modify the information stored in memory, to simplify the representation, we can omit the description of memory in the state:  $S_k = (5, 10, 1, 0, 1)$

# Example: State Machine of sISA

- State transition rules:

7	6	5	4	3	2	1	0
+-----+-----+-----+-----+							
00   rd   rs1   rs2   R[rd]=R[rs1]+R[rs2]							
+-----+-----+-----+-----+							
10   rd   imm   R[rd]=imm							
+-----+-----+-----+-----+							
11   addr   rs2   if (R[0]!=R[rs2]) PC=addr							
+-----+-----+-----+-----+							

- $S_{k+1} = \text{next}(S_k, 00101001)$ 
  - when the state is  $S_k = (5, 10, 1, 0, 1)$ , executing the instruction  $00101001$  results in the next state
    - the behaviour of this instruction is  $R[2] = R[2] + R[1]$ , so  $S_{k+1} = (6, 10, 1, 1, 1)$
  - Initial state  $S_0 = (0, 0, 0, 0, 0)$

# More about ISA

- Computer working process is like a mathematical game.
- The state machine defines the rules of the game.
- Once you understand the state machine,  
you are ready to understand how computers work.
- Commercial, real-world ISAs such as x86, ARM, and RISC-V
  - have more general-purpose registers
  - support many more instructions
  - and define more complex instruction behaviors
- However, their core is still a state machine.
- **BTM — Read The Friendly Manual**

# C Programming Language

# Limitations of Assembly Language

- Assembly language **can also be used for programming**, but it becomes **inconvenient for developing large programs**.
  - When programming in assembly, one must explicitly manage **how data flows between GPRs**
  - Moreover, **each instruction typically performs only a very limited operation**
  - As a result, **even simple program logic may require dozens of instructions to express.**
- Modern software development is typically done using **high-level programming languages**.
  - We choose C because it allows us to **easily establish a clear connection between programs and how the computer system works.**

# C Programming Language

```
/* 1 */ int main() {  
/* 2 */     int x = 1;  
/* 3 */     int y = 2;  
/* 4 */     int z = x + y;  
/* 5 */     printf("z = %d\n", z);  
/* 6 */     return 0;  
/* 7 */ }
```

- The content between /\* and \*/ is a comment
  - Comments **do not affect the program logic** and are mainly **for human readers**
- **int** denotes an **integer type**

# C Programming Language

```
/* 1 */ int main() {  
/* 2 */     int x = 1;  
/* 3 */     int y = 2;  
/* 4 */     int z = x + y;  
/* 5 */     printf("z = %d\n", z);  
/* 6 */     return 0;  
/* 7 */ }
```

- `int main() { ... }` defines a function named **main**.
  - A **function** is a basic building block in C
  - Executing a C program means **executing the code inside its functions**
  - **main** function is the **entry point** of a C program
  - That is, a C program **starts execution from the main function**

# C Programming Language

```
/* 1 */ int main() {  
/* 2 */     int x = 1;  
/* 3 */     int y = 2;  
/* 4 */     int z = x + y;  
/* 5 */     printf("z = %d\n", z);  
/* 6 */     return 0;  
/* 7 */ }
```

- The content enclosed in braces { ... } is the **function body**, which consists of **statements**.
- Each statement specifies **one operation to be performed by the program**.

# C Programming Language

```
/* 1 */ int main() {  
/* 2 */     int x = 1;  
/* 3 */     int y = 2;  
/* 4 */     int z = x + y;  
/* 5 */     printf("z = %d\n", z);  
/* 6 */     return 0;  
/* 7 */ }
```

- Statements inside the braces are **executed sequentially by default.**
  - `int x = 1;` defines a variable named **x** and initializes it with the value **1**
  - A **variable** is an object used to **store information in a program**
  - In C, **each statement ends with a semicolon ;**
  - `int y = 2;` is defined in the same way
  - `int z = x + y;` defines a variable **z** and initializes it with the value **x + y**

# C Programming Language

```
/* 1 */ int main() {
/* 2 */     int x = 1;
/* 3 */     int y = 2;
/* 4 */     int z = x + y;
/* 5 */     printf("z = %d\n", z);
/* 6 */     return 0;
/* 7 */ }
```

- `printf("z = %d\n", z);` is a **function call statement**, which calls the function **printf**.
- It is similar to a **function in mathematics**. Given  $f(x) = x + 1$ , computing  $f(3)$  means **substituting 3 into the definition of f and evaluating it**.
- Likewise, the statement above **passes two arguments**, “`z = %d\n`” and `z`, into the definition of `printf` and executes it.
- In a C program, **printf** is a special function used to **output information to the terminal**.
- In the first argument, `%d` indicates that the second argument `z` should be printed in **decimal format**.
- As a result, this function call **prints the current value of z to the terminal in decimal form**.

# C Programming Language

```
/* 1 */ int main() {  
/* 2 */     int x = 1;  
/* 3 */     int y = 2;  
/* 4 */     int z = x + y;  
/* 5 */     printf("z = %d\n", z);  
/* 6 */     return 0;  
/* 7 */ }
```

- **return 0;** is a **function return statement**.
- It indicates that the function has **finished execution** and returns **0 as the result** to the statement that called this function.

# C Programming Language

```
/* 1 */ int main() {
/* 2 */     int x = 1;
/* 3 */     int y = 2;
/* 4 */     int z = x + y;
/* 5 */     printf("z = %d\n", z);
/* 6 */     return 0;
/* 7 */ }
```

- **The components of the C language**
  - **Variables** — the objects being processed
  - **Statements** — the flow of operations that process the data
  - **Input and output functions** — allow variables to interact with the outside world

# State Machine Model of C Program

- **Intuition**
  - Variables ≈ GPRs
  - Statements ≈ Instructions
- Set of states:  $S = \{ (PC, V) \}$ 
  - $V = \{v_1, v_2, v_3, \dots\}$  represents the values of **all variables in the program**
- PC = **Program Counter**, indicating the **currently executed statement**
  - Statements inside braces are **executed sequentially by default**, which implicitly assumes the existence of a PC

# State Machine Model of C Program

- The set of events is:
  - $E = \{ \text{statements} \}$
  - The execution process is: execute the statement pointed to by the PC and update the program state
- **State transition rule**  $\text{next}: S \times E \rightarrow S$ 
  - The semantics of a statement define how the program state should change after executing that statement
  - Who defines the semantics? **the C language standard manual**
- **Initial state**  $S_0 = (\text{the first statement of main}, V_0)$ , where  $V_0$  denotes the initial values of program variables

# State Machine Model of C Program

```
/* 1 */ int main() {
/* 2 */     int x = 1;
/* 3 */     int y = 2;
/* 4 */     int z = x + y;
/* 5 */     printf("z = %d\n", z);
/* 6 */     return 0;
/* 7 */ }
```

# State Machine Model of C Program

```
/* 1 */ int main() {
/* 2 */     int x = 1;
/* 3 */     int y = 2;
/* 4 */     int z = x + y;
/* 5 */     printf("z = %d\n", z);
/* 6 */     return 0;
/* 7 */ }
```

PC	x	y	z	Notes
2	?	?	?	initial state
3	1	?	?	After executing the statement with PC = 2, x is updated to 1, and PC is updated to the position of the next statement.
4	1	2	?	After executing the statement with PC = 3, y is updated to 2, and PC is updated to the position of the next statement.
5	1	2	3	After executing the statement with PC = 4, z is updated to x + y, and PC is updated to the position of the next statement.
6	1	2	3	After executing the statement with PC = 5, the terminal outputs 'z = 3' and PC is updated to the position of the next statement.
END	1	2	3	After executing the statement with PC = 6, the program returns from the main function and execution ends.

# Implement a Sequence Sum in C

- This the Sequence Sum program in C
- Please list the states of PC, sum, i

```
/* 1 */ int main() {  
/* 2 */     int sum = 0;  
/* 3 */     int i = 1;  
/* 4 */     do {  
/* 5 */         sum = sum + i;  
/* 6 */         i = i + 1;  
/* 7 */     } while (i <= 10);  
/* 8 */     printf("sum = %d\n", sum);  
/* 9 */     return 0;  
/* 10 */ }
```

```

/* 1 */ int main() {
/* 2 */     int sum = 0;
/* 3 */     int i = 1;
/* 4 */     do {
/* 5 */         sum = sum + i;
/* 6 */         i = i + 1;
/* 7 */     } while (i <= 10);
/* 8 */     printf("sum = %d\n", sum);
/* 9 */     return 0;
/* 10*/ }

```

PC	sum	i	Notes
2	?	?	initial state
3	0	?	After executing the statement with PC = 2, sum is updated to 0, and PC is updated to the position of the next statement.
5	0	1	After executing the statement with PC = 3, i is updated to 1, and PC is updated to the position of the next statement (there is no valid operation on line 4, so it is skipped).
6	1	1	After executing the statement with PC = 5, sum is updated to sum + i, and PC is updated to the position of the next statement.
7	1	2	After executing the statement with PC = 6, i is updated to i + 1, and PC is updated to the position of the next statement.
5	1	2	After executing the statement with PC = 7, since the loop condition i <= 10 is true, the loop body is re-entered.
...	...	...	...

# Advantages of C

- **Compared with assembly language**, programming in C has at least the following advantages:
- **Variable names** can more intuitively reflect their intended purpose. In assembly language, the role of a GPR can only be inferred from the surrounding context.
- **Loops are expressed more clearly**, with the **loop condition** and the **loop body** explicitly distinguished. In assembly language, both the loop condition and the loop body are expressed as instructions and must be inferred from context.

# State Machine Model of Digital Circuits

- Digital logic circuits = combinational logic + sequential logic
- **State S** = { values of sequential elements } which include **registers, memories, flip-flops**, and similar components
- **Events E** = { combinational logic } The signals produced by the **combinational logic** drive the **sequential elements** to change their state

# State Machine Model of Digital Circuits

- State transition rule next:  $S \times E \rightarrow S$
- The state transition function is **determined by the combinational logic in the design**
- Initial state  $S_0 = (\text{values of sequential elements at reset})$

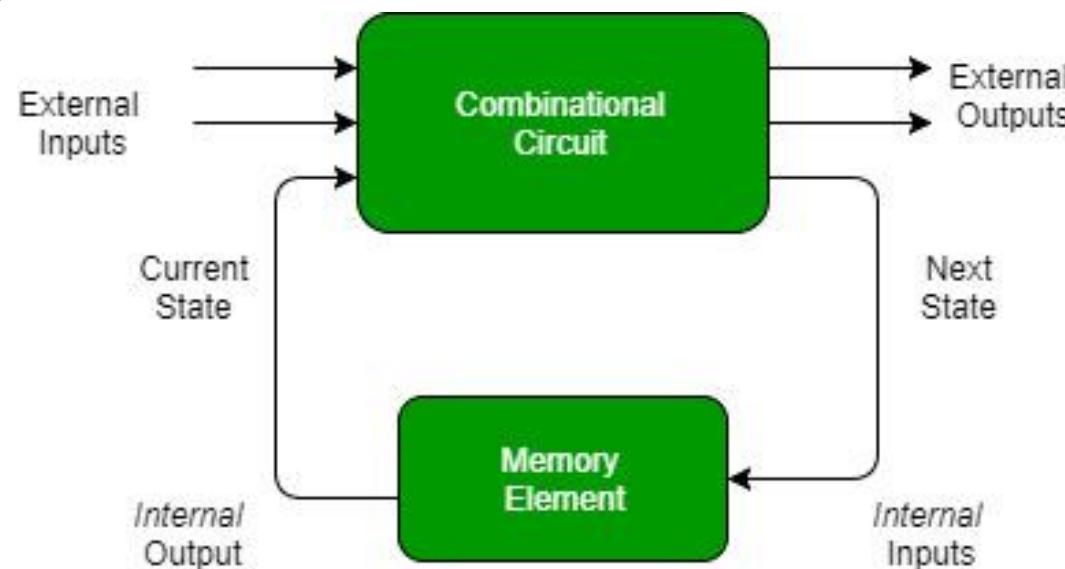


Figure: Sequential Circuit

# The Relationship between Programs, ISAs, and CPUs

- Based on the functional description in the ISA manual, draw a diagram of the CPU structure -> processor microarchitecture design
- Based on the structural design, design the specific circuit -> logic design
- Develop the program -> Software programming
- Translate the program into the instruction sequence described in the ISA manual -> Compilation
- Execute the program on the CPU = Use the instruction sequence compiled from the program to control the CPU circuits to perform state transitions
  - At this point, the three state machines are interconnected:  $S_C \sim S_{ISA} \sim S_{CPU}$

# Try: Design your own ISA

- You could try to design some instructions and make your own ISA.
- Remember:
  - give the bit width of your ISA
  - how many registers
  - define instructions with its opcode and assembly
- Then, use your ISA to design a program
  - To compute something?

# Q&A

Reference: <https://ysyx.oscc.cc/docs/en/2407> (Stage F Chapter 4)