Introduction to Computer and Programming

Lecture 13

Yue Zhang
Westlake University

August 1, 2023

Chapter 13.

Computer and Programs

Computers and Programs

- Computers are programmable finite state machines.
- Each computer has a specific instruction set.
- We can directly write machine code (bytecode).

ADD	R3	R1	R2	constant 0
000001	00011	00001	00010	0000000000

- but this is time consuming.
- and not general to all machines.

A Bytecode Program

- Adding numbers

instructions

input data

output

A Bytecode Program

- Adding numbers

0	REG[R0]+32 ⇒ REG[R1]
4	REG[R0]+36 ⇒ REG[R2]
8	REG[R0]+40 ⇒ REG[R3]
12	MEM[REG[R1]] ⇒ REG[R4]
16	MEM[REG[R2]] ⇒ REG[R5]
20	REG[R4]+ REG[R5] ⇒ REG[R6]
24	REG[R6] ⇒ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

- Since registers start with random values, set R0 to constant 0 in hardware.
- HALT = 0 is a special instruction to stop the machine.
- The first three instructions contain addresses for data (32, 36, 40) which can be decided only after the rest of the program is designed.

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⇒ REG[R1]
4	REG[R0]+36 ⇒ REG[R2]
8	REG[R0]+40 ⇒ REG[R3]
12	MEM[REG[R1]] ⇒ REG[R4]
16	MEM[REG[R2]] ⇒ REG[R5]
20	REG[R4]+ REG[R5] ⇒ REG[R6]
24	REG[R6] ⇒ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⇒ REG[R1]
4	REG[R0]+36 ⇒ REG[R2]
8	REG[R0]+40 ⇒ REG[R3]
12	MEM[REG[R1]] ⇒ REG[R4]
16	MEM[REG[R2]] ⇒ REG[R5]
20	REG[R4]+ REG[R5] ⇒ REG[R6]
24	REG[R6] ⇒ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000

	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	00000000000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

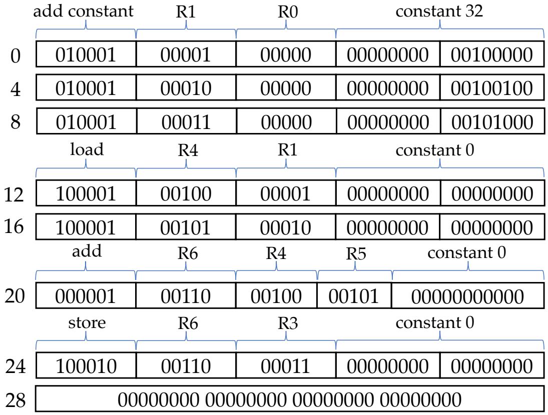
⟹

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	000000000000
	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

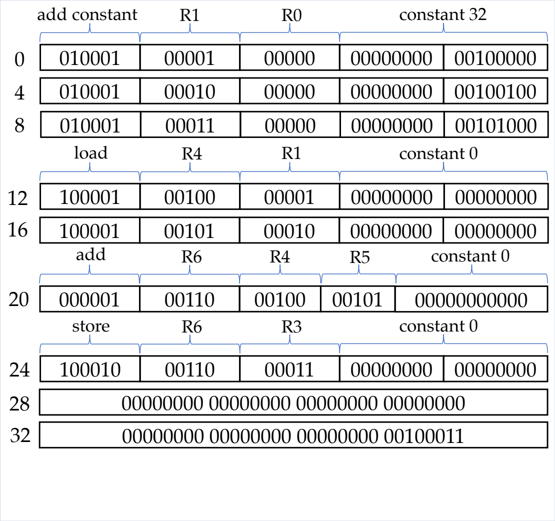


	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	000000000000
	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000
28	00000000 00000000 00000000 00000000				

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0



	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000

	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	00000000000

	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000

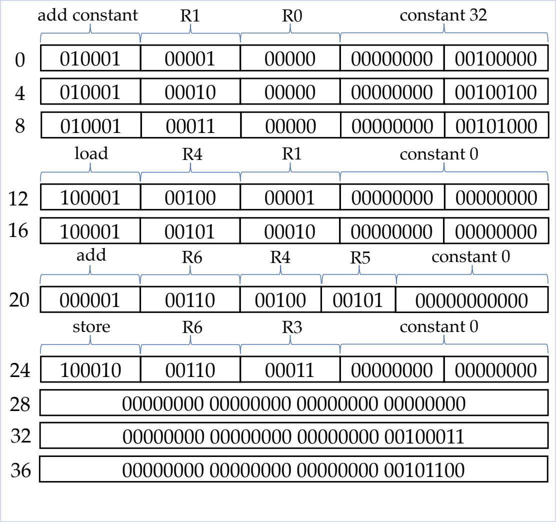
28	00000000 00000000 00000000 00000000
32	00000000 00000000 00000000 00100011

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨



	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	000000000000

	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000

28 00000000 00000000 00000000 00000000

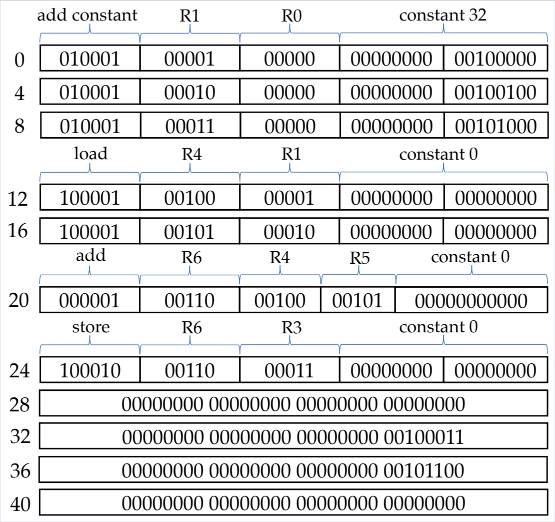
32 00000000 00000000 00000000 00100011

36 00000000 00000000 00000000 00101100

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0



	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	000000000000
	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000
28	00000000 00000000 00000000 00000000				
32	00000000 00000000 00000000 00100011				
36	00000000 00000000 00000000 00101100				
40	00000000 00000000 00000000 00000000				

A Bytecode Program

- Adding numbers — the code

0	REG[R0]+32 ⟹ REG[R1]
4	REG[R0]+36 ⟹ REG[R2]
8	REG[R0]+40 ⟹ REG[R3]
12	MEM[REG[R1]] ⟹ REG[R4]
16	MEM[REG[R2]] ⟹ REG[R5]
20	REG[R4]+ REG[R5] ⟹ REG[R6]
24	REG[R6] ⟹ MEM[REG[R3]]
28	HALT
32	35
36	44
40	0

⇨

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000

	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

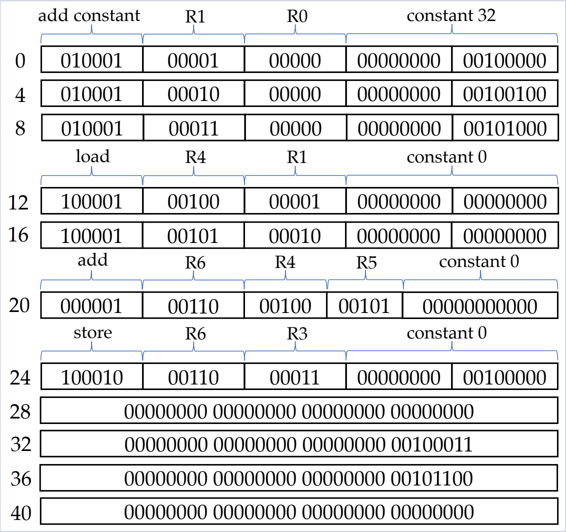
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	00000000000

	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000

28	00000000 00000000 00000000 00000000
32	00000000 00000000 00000000 00100011
36	00000000 00000000 00000000 00101100
40	00000000 00000000 00000000 00000000

- Change the last three rows if you want to add different numbers.

The Assembly Version



	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	000000000000
	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00100000
28	00000000 00000000 00000000 00000000				
32	00000000 00000000 00000000 00100011				
36	00000000 00000000 00000000 00101100				
40	00000000 00000000 00000000 00000000				

- Bytecode is time-consuming.
- Easy to make mistakes.
- Different machines have different instruction coding.
- Solution: use macros to represent numbers.

The Assembly Version

- Assembly Language

  - symbols —— represent numbers

    e.g., $R0 = 0, R1 = 1, \cdots, R31 = 31$

  - macros —— assemble instructions, etc.

    $WORD(x)$ —— a word (binary number) with decimal value x.

    $ADD(X_1, X_2, Y)$ —— e.g., $ADD(R1, R2, R3)$, $REG[R1] + REG[R2] \Longrightarrow REG[R3]$

    $ADDC(X, x, Y)$ —— e.g., $ADDC(R1{,}5, R2)$, $REG[R1] + 5 \Longrightarrow REG[R2]$

    $SUB(X_1, X_2, Y)$ —— e.g., SUB$(R1, R2, R3)$, $REG[R1] - REG[R2] \Longrightarrow REG[R3]$

    $MUL(X_1, X_2, Y)$ —— e.g., $MUL(R1, R2, R3)$, $REG[R1] * REG[R2] \Longrightarrow REG[R3]$

    … … … …

    $LOAD(ADDR, Y)$ —— e.g., $LOAD(R3, R1)$, $MEM[REG[R3]] \Longrightarrow REG[R1]$

    $STORE(Y, ADDR)$ —— e.g., $STORE(R1, R3)$, $REG[R1] \Longrightarrow MEM[REG[R3]]$

    $JMP(X, ADDR, Y)$ —— e.g., $JMP(R1, R3, R4)$, if $REG[R1] = 0, PC + 4 \Longrightarrow REG[R4]$, $REG[R3] \Longrightarrow PC$

    $HALT()$

The Assembly Version

0	REG[R0]+32 ⇒ REG[R1]
4	REG[R0]+36 ⇒ REG[R2]
8	REG[R0]+40 ⇒ REG[R3]
12	MEM[REG[R1]] ⇒ REG[R4]
16	MEM[REG[R2]] ⇒ REG[R5]
20	REG[R4]+ REG[R5] ⇒ REG[R6]
24	REG[R6] ⇒ MEM[REG[R3]]
28	HALT
x: 32	35
36	44
40	0

- Assembly Language

  - symbols —— represent numbers

  - macros —— assemble instructions, etc.
    alias can be made.

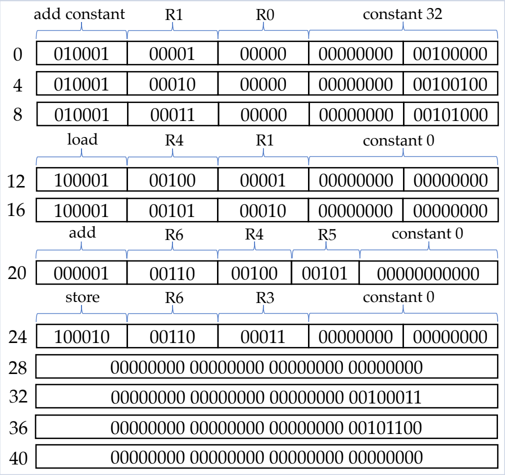
    $MOVE(X, Y)$ —— $ADD(X, R0, Y)$

    $ASSIGN(x, Y)$ —— $ADDC(R0, x, Y)$

    $GOTO(ADDR, Y)$ —— $JMP(R0, ADDR, Y)$

  - labels —— mark memory address
    x: $WORD(35)$, x is the address of the word.

The Assembly Version



add constant | R1 | R0 | constant 32

0 | 010001 | 00001 | 00000 | 00000000 | 00100000

4 | 010001 | 00010 | 00000 | 00000000 | 00100100

8 | 010001 | 00011 | 00000 | 00000000 | 00101000

load | R4 | R1 | constant 0

12 | 100001 | 00100 | 00001 | 00000000 | 00000000

16 | 100001 | 00101 | 00010 | 00000000 | 00000000

add | R6 | R4 | R5 | constant 0

20 | 000001 | 00110 | 00100 | 00101 | 00000000000

store | R6 | R3 | constant 0

24 | 100010 | 00110 | 00011 | 00000000 | 00000000

28 | 00000000 00000000 00000000 00000000

32 | 00000000 00000000 00000000 00100011

36 | 00000000 00000000 00000000 00101100

40 | 00000000 00000000 00000000 00000000

Bytecode

v.s.

$ADDC(R0, x, R1)$

$ADDC(R0, y, R2)$

$ADDC(R0, z, R3)$

$LOAD(R1, R4)$

$LOAD(R2, R5)$

$ADD(R4, R5, R6)$

$STORE(R6, R3)$

$HALT$

$x: WORD(35)$

$y: WORD(44)$

$z: WORD(0)$

Assembly

The Assembly Version

	add constant	R1	R0	constant 32		
0	010001	00001	00000	00000000	00100000	$ADDC(R0, x, R1)$
4	010001	00010	00000	00000000	00100100	$ADDC(R0, y, R2)$
8	010001	00011	00000	00000000	00101000	$ADDC(R0, z, R3)$
	load	R4	R1	constant 0		
12	100001	00100	00001	00000000	00000000	$LOAD(R1, R4)$
16	100001	00101	00010	00000000	00000000	$LOAD(R2, R5)$
	add	R6	R4	R5	constant 0	
20	000001	00110	00100	00101	000000000000	$ADD(R4, R5, R6)$
	store	R6	R3	constant 0		
24	100010	00110	00011	00000000	00000000	$STORE(R6, R3)$
28	00000000 00000000 00000000 00000000					$HALT$
32	00000000 00000000 00000000 00100011					$x: WORD(35)$
36	00000000 00000000 00000000 00101100					$y: WORD(44)$
40	00000000 00000000 00000000 00000000					$z: WORD(0)$

Bytecode

v.s.

Assembly

- do not need to remember byte coding.
- use x, y, z to replace 32, 36, 40, which allows the first three instructions to be easily written.

The Assembly Language

- Much easier to program.
- Need to use **assemble** to translate **assembly code** into **bytecode**.
- Still time-consuming.
  - Need to worry about registers.
  - Need to allocate memory manually.
  - Need to manage jumping manually.

The C Version

- We want to program without worrying about (as much as possible):
  - The hardware structure such as registers and word size.
  - The memory structure of the bytecode.
  - The control of program counter.
  - Input, output devices, OS calls.
- C addresses these concerns by:
  - Defining abstract data types (e.g., int x)
  - Defining abstract control flow (e.g., if, while, function)
- The C **compilers** translates C code into bytecode.
  - Design a compiler for each machine architecture.

The C Version

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000
	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000
	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	00000000000
	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000
28	00000000 00000000 00000000 00000000				
32	00000000 00000000 00000000 00100011				
36	00000000 00000000 00000000 00101100				
40	00000000 00000000 00000000 00000000				

Bytecode

v.s.

```
ADDC(R0, x, R1)
ADDC(R0, y, R2)
ADDC(R0, 2, R3)

LOAD(R1, R4)
LOAD(R2, R5)

ADD(R4, R5, R6)

STORE(R6, R3)
HALT
x: WORD(35)
y: WORD(44)
z: WORD(0)
```

Assembly

```
int main(){
    int x, y, z;
    x = 35;
    y = 44;
    z = x + y;
    printf("%d", z);
    return 0;
}
```

C

The C Version

	add constant	R1	R0	constant 32	
0	010001	00001	00000	00000000	00100000
4	010001	00010	00000	00000000	00100100
8	010001	00011	00000	00000000	00101000

	load	R4	R1	constant 0	
12	100001	00100	00001	00000000	00000000
16	100001	00101	00010	00000000	00000000

	add	R6	R4	R5	constant 0
20	000001	00110	00100	00101	00000000000

	store	R6	R3	constant 0	
24	100010	00110	00011	00000000	00000000

28	00000000 00000000 00000000 00000000
32	00000000 00000000 00000000 00100011
36	00000000 00000000 00000000 00101100
40	00000000 00000000 00000000 00000000

Bytecode

v.s.

$ADDC(R0, x, R1)$
$ADDC(R0, y, R2)$
$ADDC(R0, 2, R3)$

$LOAD(R1, R4)$
$LOAD(R2, R5)$

$ADD(R4, R5, R6)$

$STORE(R6, R3)$
$HALT$
$x: WORD(35)$
$y: WORD(44)$
$z: WORD(0)$

Assembly

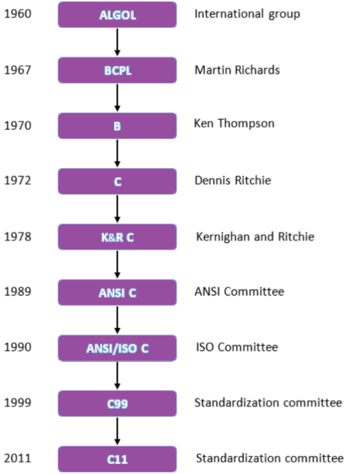
```
int main(){
    int x, y, z;
    x = 35;
    y = 44;
    z = x + y;
    printf("%d", z);
    return 0;
}
```

C

- No registers !
- No manual memory allocation !
- Easy access to monitor !

The C Language

- C was invented in the early 1970s for developing UNIX, a famous operating system.
- C and its variations are among the most widely used programming languages.
- Since C is directly compiled into bytecode, the speed of code can be highly optimized.



1960	ALGOL	International group
1967	BCPL	Martin Richards
1970	B	Ken Thompson
1972	C	Dennis Ritchie
1978	K&R C	Kernighan and Ritchie
1989	ANSI C	ANSI Committee
1990	ANSI/ISO C	ISO Committee
1999	C99	Standardization committee
2011	C11	Standardization committee

The C Language

- I will briefly introduce:
  - how abstract data types are achieved.
  - how abstract control flow is achieved.
- I will **not** introduce C in detail.

The C Language

Abstract Data Types

- Simple types
  - int a; —— integer, a word.
  - unsigned int ua; —— unsigned integer, a word.
  - char c; —— a byte.
- Literals
  - 132, -55, 'A', "abc", similar to the assembly, these (char array) are translated to binary by the compiler.
- Structured types
  - int a[100]; —— array of integers.
  - char s[7]; —— array of chars, strings.

WestlakeNLP

The C Language

Control Structures

- branching

```
if (<integer expression>){
    <statement 1>
} else {
    <statement 2>
}
```

- looping

```
while (<integer expression>){
    <statement 1>
}
```

➢ A statement can be a block, in {…}.

➢ There are also *do … while* and *for* statements.

The C Language

Control Structures

- functions

```
<return type> <function name>(<parameters>){
    <statement blocks>
}
```

e.g.,

```
int f (int x){
    return x+1;
}
```

The C Language

Control Structures

- functions

```
<return type> <function name>(<parameters>){
    <statement blocks>
}
```

- C is a procedural programming language, where code is organized in functions, and there is a *main* function.

```
int main (/*required arguments*/){
    … …
    return 0; /*indicating status*/
}
```

The C Language

How is C compiled into bytecode?

- A Simple Case

C code

```
int main(){
    static int x = 35;
    static int y = 44;
    static int z;
    z = x + y;
    return 0;
}
```

Compiled code (equivalent assembly)

```
ADDC(R0, x, R1)
ADDC(R0, y, R2)
ADDC(R0, z, R3)
LOAD(R1, R4)
LOAD(R2, R5)
ADD(R4, R5, R6)
STORE(R6, R3)
HALT
x: WORD(35)
y: WORD(44)
z: WORD(0)
```

- What does “static” mean? We will talk about it later.

The C Language

How is C compiled into bytecode?

- ➢ Control Structure —— branching

C code

```
if (<expression>){
    <statements 1>
} else {
    <statements 2>
}
```

Compiled code (equivalent assembly)

$$\begin{gathered}
evaluate < expression > \\
storing\ the\ result\ to R1 \\
ADDC(R0, else\_001, R2) \\
ADDC(R0, endif\_001, R3) \\
JMP(R1, R2, R4) \\
translate < statements1 > \\
GOTO(R3) \\
else\_001: \\
translate < statements2 > \\
endif\_001:
\end{gathered}$$

The C Language

How is C compiled into bytecode?

- Control Structure —— looping

C code

```
while (<expression>){
    <statements>
}
```

Compiled code (equivalent assembly)

$$ADDC(R0, while\_001, R1)$$
$$ADDC(R0, endwhile\_001, R2)$$
$$while\_001:$$
$$evalute < expression >$$
$$storing\ the\ results\ to\ R3$$
$$JMP(R3, R2, R4)$$
$$translate < statements >$$
$$GOTO\ R1$$
$$endwhile\_001:$$

WestlakeNLP

The C Language

How is C compiled into bytecode?

- How functions are compiled?
  - Functions can call functions.
  - Each function call has arguments and return values.
  - How to store them?

The C Language

How is C compiled into bytecode?

- Now back to "static"

```
int main(){
    static int x = 35;
    static int y = 44;
    static int z;
    z = x + y;
    return 0;
}
```

```
int main(){
    int x = 35;
    int y = 44;
    int z;
    z = x + y;
    return 0;
}
```

- What if the "static" signs are removed?

The C Language

How is C compiled into bytecode?

- Now back to “static”

```
int main(){
    static int x = 35;
    static int y = 44;
    static int z;
    z = x + y;
    return 0;
}
```

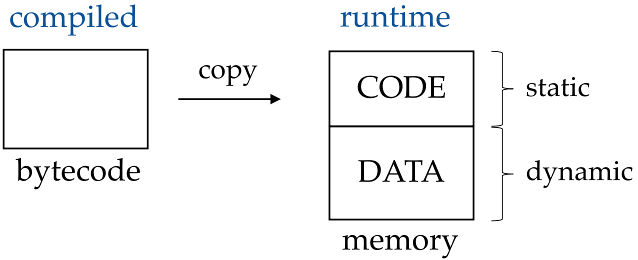
```
int main(){
    int x = 35;
    int y = 44;
    int z;
    z = x + y;
    return 0;
}
```

- What if the “static” signs are removed?
  - The compiled bytecode will differ.
  - The variables x, y and z will no longer be in the bytecode.
  - Where are they now?

The C Language

How is C compiled into bytecode?

- C has a dynamic memory allocation mechanism.

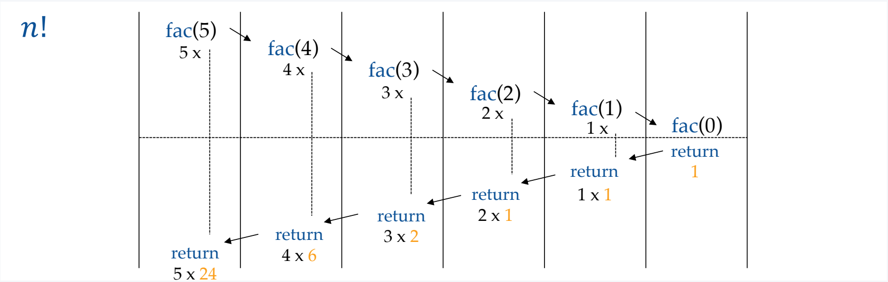


- Dynamic memory allocation is achieved by executing code in the bytecode (CODE) section, and down to the DATA section during the runtime.

How is C compiled into bytecode?

- Runtime memory.
- A type of memory allocation that suits function calls.

Stack



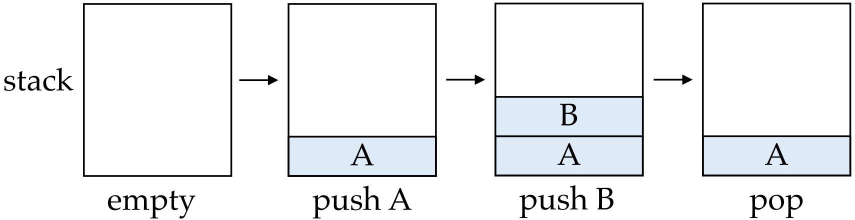
The C Language

How is C compiled into bytecode?

- Runtime memory.
- A type of memory allocation that suits function calls.

Stack

The stack works in last-in-first-out order.

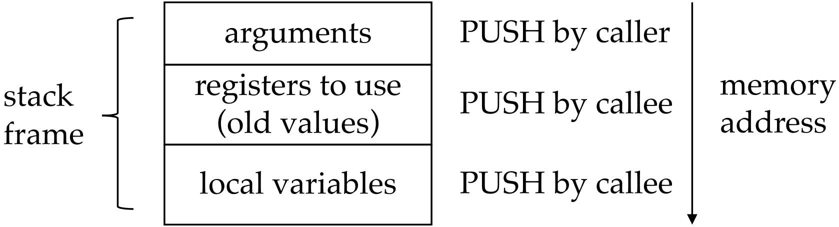


The C Language

How is C compiled into bytecode?

- Functions —— runtime memory

```
int f (int x){
    int y = 1;
    return x + y;
}
```



stack frame

arguments	PUSH by caller
registers to use (old values)	PUSH by callee
local variables	PUSH by callee

memory address

The C Language

How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack.	④
PUSH a word 1 onto the stack.	⑤
LOAD arguments a, b into R1 and R2, respectively.	⑥
ADD R1, R2 and the value 1, storing the result into a return value register.	⑦
RESTORE old values of R1, R2.	⑧
POP all values pushed in this call.	⑨
JMP back.	⑩
PUSH two words 2 and 3 onto the stack in reverse order.	①
JMP *f*.	②
LOAD return value into register.	③
POP two slots.	(3a)

The C Language

How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

code section

data section

The C Language

How is C compiled into bytecode?

C code

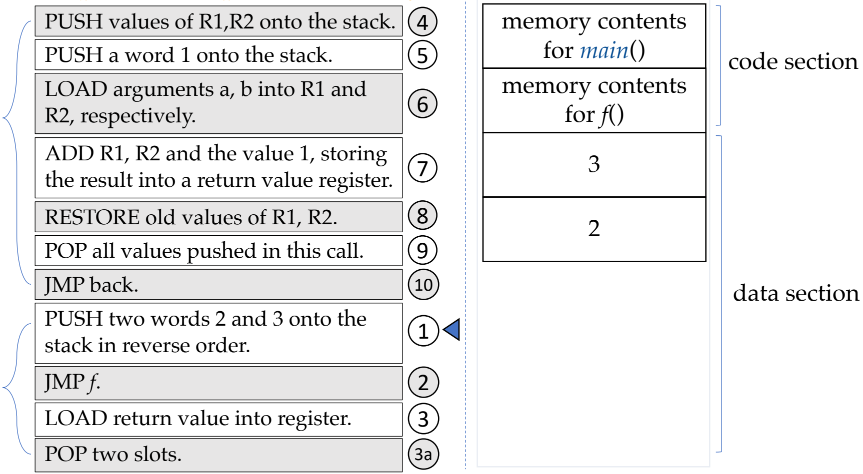
- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

Runtime memory



PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

The C Language

How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

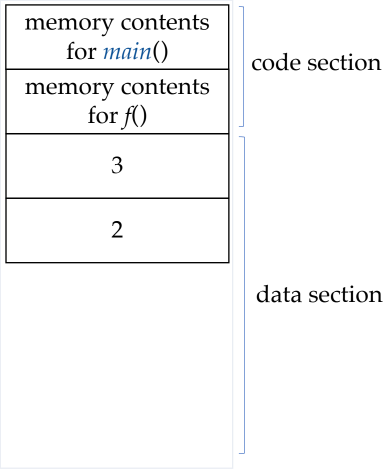
PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ② ◀

LOAD return value into register. ③

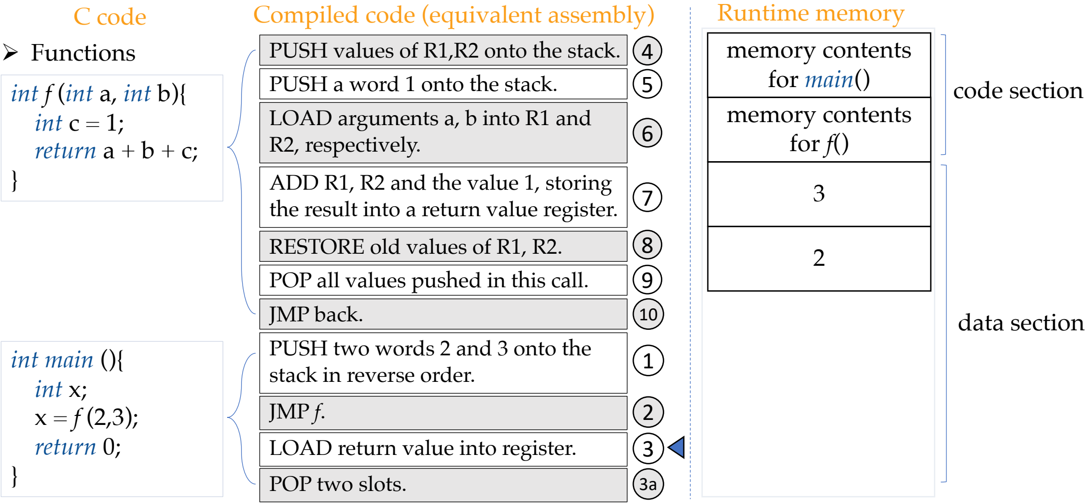
POP two slots. (3a)

Runtime memory



The C Language

How is C compiled into bytecode?



C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

memory contents for *main*()

memory contents for *f*()

3

2

code section

data section

The C Language

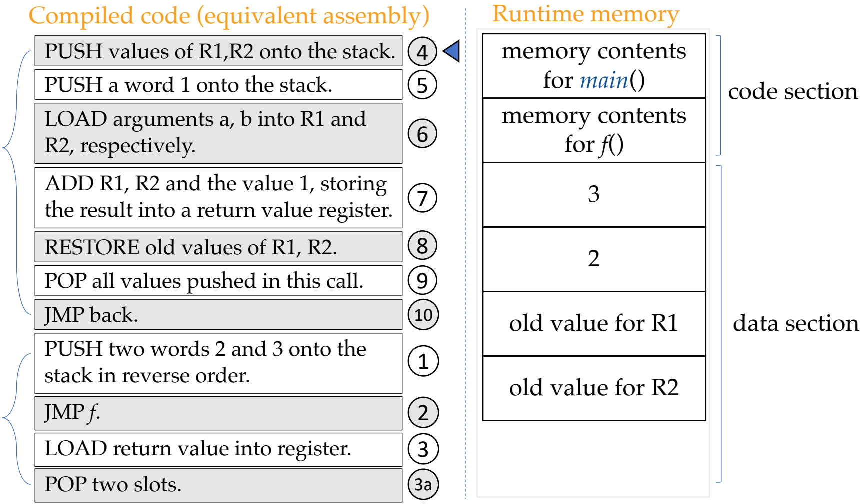
How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```



Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

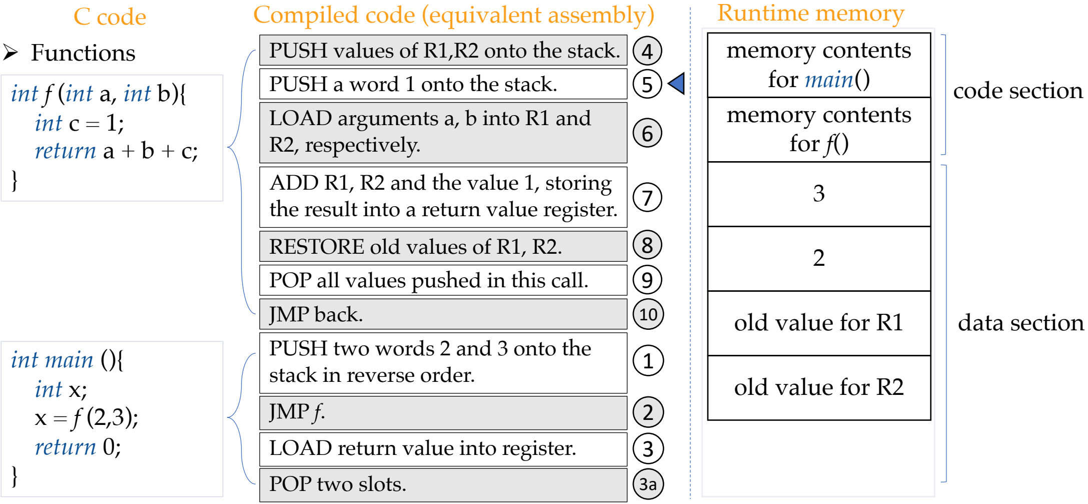
memory contents for *main*()
memory contents for *f*()
3
2
old value for R1
old value for R2

code section

data section

The C Language

How is C compiled into bytecode?



C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

memory contents for *main*()

memory contents for *f*()

code section

3

2

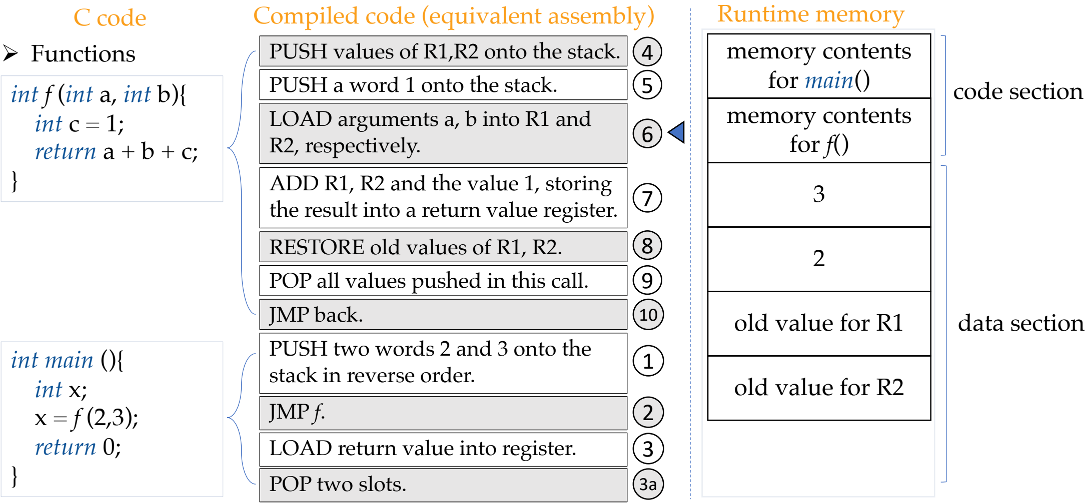
old value for R1

old value for R2

data section

The C Language

How is C compiled into bytecode?



C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥ ◀

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

memory contents for *main*()	code section
memory contents for *f*()	
3	data section
2	
old value for R1	
old value for R2	

The C Language

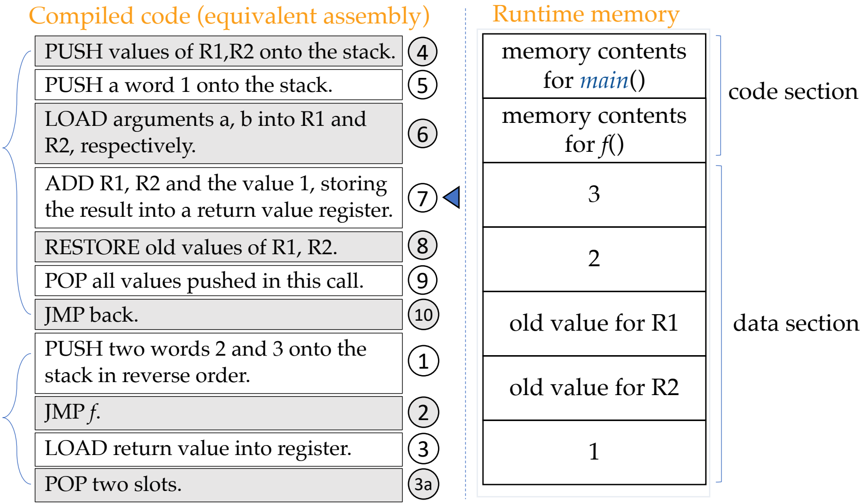
How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```



Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦ ◀

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

memory contents for *main*()
memory contents for *f*()
3
2
old value for R1
old value for R2
1

The C Language

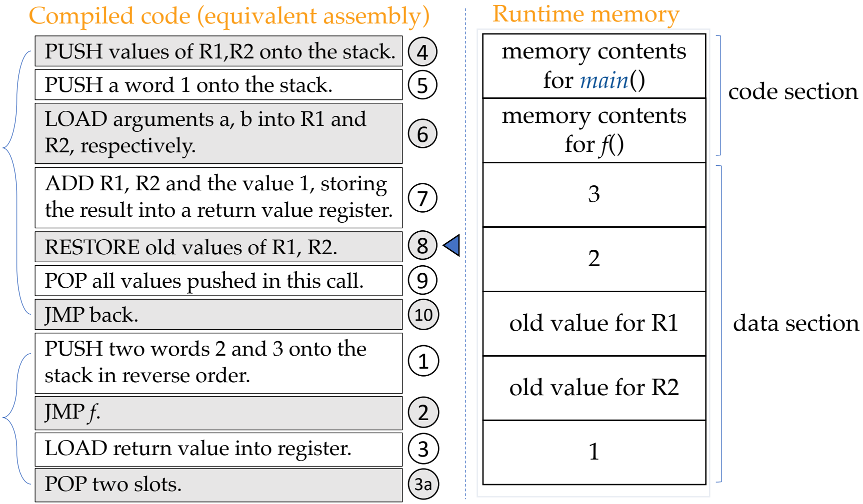
How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```



Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧ ◀

POP all values pushed in this call. ⑨

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

memory contents for *main*()	code section
memory contents for *f*()	
3	data section
2	
old value for R1	
old value for R2	
1	

The C Language

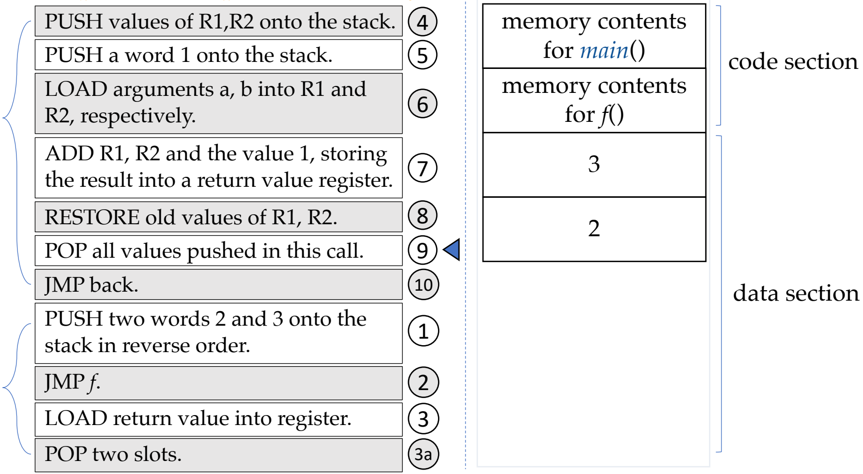
How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```



Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨ ◀

JMP back. ⑩

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

The C Language

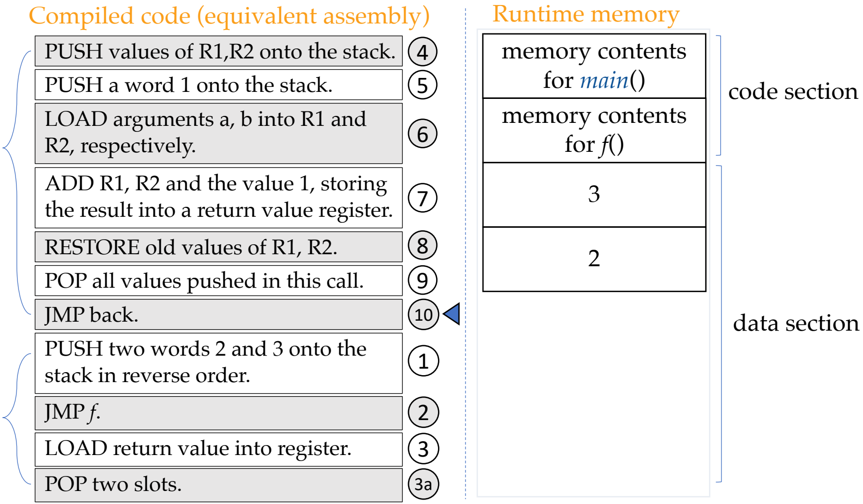
How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```



Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩ ◀

PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a)

Runtime memory

The C Language

How is C compiled into bytecode?

C code

- Functions

```
int f (int a, int b){
    int c = 1;
    return a + b + c;
}
```

```
int main (){
    int x;
    x = f (2,3);
    return 0;
}
```

Compiled code (equivalent assembly)

PUSH values of R1,R2 onto the stack. ④

PUSH a word 1 onto the stack. ⑤

LOAD arguments a, b into R1 and R2, respectively. ⑥

ADD R1, R2 and the value 1, storing the result into a return value register. ⑦

RESTORE old values of R1, R2. ⑧

POP all values pushed in this call. ⑨

JMP back. ⑩

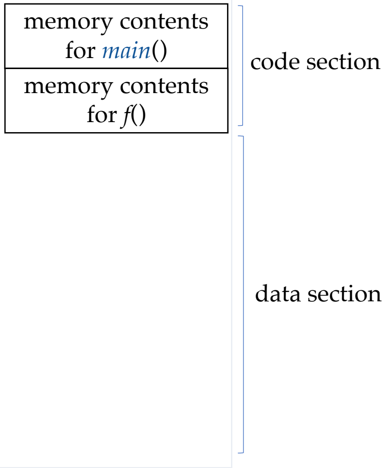
PUSH two words 2 and 3 onto the stack in reverse order. ①

JMP *f*. ②

LOAD return value into register. ③

POP two slots. (3a) ◀

Runtime memory



memory contents for *main*()

memory contents for *f*()

code section

data section

The C Language

- Summary
  - abstract data types
  - abstract control flow
  - procedural language
  - automatic dynamic memory allocation
    - in function calls (stack)
    - manually (heap)
- Limitations
  - still need to have a memory model (can have out-of-memory / segdefault errors)
  - data types not rich

The Python Language

- Not compiled into bytecode, but executed over a **Python interpreter**.
- Python is a bytecode.
- Advantages
    - rich abstract data types
    - object oriented programming
    - more robust

The Pseudocode

- We have already seen pseudocode earlier.
- easy to describe algorithms.
- ChatGPT can deal with pseudocode and do some human language programming!