Assembly Language Lecture Series: X86-64 Control Transfer Instructions

Sensei RL Uy, College of Computer Studies, De La Salle University, Manila, Philippines

Copyright Notice

This lecture contains copyrighted materials and is use solely for instructional purposes only, and not for redistribution.

Do not edit, alter, transform, republish or distribute the contents without obtaining express written permission from the author.

x86-64 Arithmetic Instructions

- 1. CMP
- 2. JMP
- 3. Jcc
- 4. LOOP
- 5. JCXZ/JECXZ

- 6. PUSH
- 7. **POP**
- 8. CALL
- 9. **RET**

CMP (compare)

Syntax: CMP src1, src2

src1 – src2

result discarded

src1: reg/mem

src2: reg/mem/imm8_16_32

Flags affected:

*all status flags

Note:

- 1. Immediate value up to **32-bit** only
- When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format
- Negative number in hex has to be sign-extended to 64-bit

CMP (compare)

Syntax: CMP src1, src2

src1 – src2

result discarded

src1: reg/mem

src2: reg/mem/imm8_16_32

Flags affected:

*all status flags

```
section .text
MOV RAX, 0x0000_0000_0000_0006
MOV RBX, 0x0000_0000_0000_0006
CMP RAX, RBX
```

- 1. What will RAX contain after execution?
- 2. What will be the value of the status flags after execution?

CMP (compare)

Syntax: CMP src1, src2

src1-src2

result discarded

src1: reg/mem

src2: reg/mem/imm8_16_32

Flags affected:

*all status flags

Example:

```
section .text

MOV RAX, 0x0000_0000_0000_0006

MOV RBX, 0x0000_0000_0000_0006

CMP RAX, RBX
```

- What will RAX contain after execution?
- 2. What will be the value of the status flags after execution?

```
RAX = 0000000000000000 OF = 0

CF = 0 PF = 1

SF = 0 AF = 0

7F = 1
```

For readability: 0000_0000_00006

JMP (Jump instruction)

Unconditional branch

JMP < label>

Flags affected: none

JMP (Jump instruction)

Unconditional branch

JMP < label>

Flags affected: none

Example:

```
section .text
MOV CL, -2
JMP L1
MOV CL, 1
L1: NOP
```

JMP (Jump instruction)

Unconditional branch

JMP < label>

Flags affected: none

Example:

```
section .text
MOV CL, -2
JMP L1
MOV CL, 1
L1: NOP
```

What will CL contain after execution?

CL = FE

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 0xFE
MOV BL, 0xFF
CMP AL, BL
JL L1
MOV CL, 0x00
JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < BL then goto L1 else next line

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 0xFE
MOV BL, 0xFF
CMP AL, BL
JL L1
MOV CL, 0x00
JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < BL then goto L1 else next line

1. What will CL contain after execution?

CL = FF

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 56
MOV BL, 45
CMP AL, BL
JL L1
MOV CL, 0x00
JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < BL then goto L1 else next line

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 56
MOV BL, 45
CMP AL, BL
JL L1
MOV CL, 0x00
JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < BL then goto L1 else next line

1. What will CL contain after execution?

CL = 00

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text MOV AL, 0x45 MOV BL, 0xFE CMP AL, BL JG L1 MOV CL, 0x00 JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL > BL then goto L1 else next line

Jcc < label>

Conditional branch (Signed):

JL: jump if less than

JG: jump if greater than

JE: jump if equal

JNE: jump if not equal

JGE: jump if >=

JLE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 0x45
MOV BL, 0xFE
CMP AL, BL
JG L1
MOV CL, 0x00
JMP FINTS

L1: MOV CL, 0xFF

FINIS: NOP

If AL > BL then goto L1 else next line

1. What will CL contain after execution?

CL = FF

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 0xFE
MOV BL, 0xFF
CMP AL, BL
JB L1
MOV CL, 0x00
JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < unsigned BL then goto L1 else next line

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text
MOV AL, 0xFE
MOV BL, 0xFF
CMP AL, BL
JB L1
MOV CL, 0x00
JMP FINIS

If AL < unsigned BL

then goto L1

else next line

L1: MOV CL, 0xFF

FINIS: NOP

1. What will CL contain after execution?

CL = FF

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text MOV AL, 0x56 MOV BL, 0x45 CMP AL, BL JB L1 MOV CL, 0x00 JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL < unsigned BL then goto L1 else next line

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text MOV AL, 0x56 MOV BL, 0x45 CMP AL, BL JB L1 MOV CL, 0x00 JMP FINIS

L1: MOV CL, ØxFF

FINIS: NOP

If AL < unsigned BL then goto L1 else next line

1. What will CL contain after execution?

CL = 00

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text MOV AL, 0xFE MOV BL, 0x45 CMP AL, BL JA L1 MOV CL, 0x00 JMP FINTS

JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

If AL > unsigned BL then goto L1 else next line

Jcc < label>

Conditional branch (Unsigned):

JB: jump if less than

JA: jump if greater than

JAE: jump if >=

JBE: jump if <=

Flags affected: none

Example:

section .text MOV AL, 0xFE MOV BL, 0x45 CMP AL, BL JA L1 MOV CL, 0x00 JMP FINTS

L1: MOV CL, 0xFF

FINIS: NOP

If AL > unsigned BL then goto L1 else next line

1. What will CL contain after execution?

CL = FF

Jcc < label>

Conditional branch (Using Flag):

JC: jump if CF = 1

JNC: jump if CF = 0

JZ: jump if ZF = 1

JNZ: jump if ZF = 0

JS: jump if SF = 1

JNS: jump if SF = 0

JP: jump if PF = 1

JNP: jump if PF = 0

JO: jump if OF = 1

JNO: jump if OF = 0

Flags affected: none

Jcc < label>

Conditional branch (Using Flag):

JC: jump if CF = 1

JNC: jump if CF = 0

JZ: jump if ZF = 1

JNZ: jump if ZF = 0

JS: jump if SF = 1

JNS: jump if SF = 0

JP: jump if PF = 1

JNP: jump if PF = 0

JO: jump if OF = 1

JNO: jump if OF = 0

Flags affected: none

Example:

section .text MOV AL, 0xFF INC AL JZ L1 MOV CL, 0x00

If AL == 0 then goto L1 else next line

JMP FINIS

L1: MOV CL, 0xFF FINIS: NOP

Jcc < label>

Conditional branch (Using Flag):

JC: jump if CF = 1

JNC: jump if CF = 0

JZ: jump if ZF = 1

JNZ: jump if ZF = 0

JS: jump if SF = 1

JNS: jump if SF = 0

JP: jump if PF = 1

JNP: jump if PF = 0

JO: jump if OF = 1

JNO: jump if OF = 0

Flags affected: none

Example:

section .text MOV AL, 0xFF INC AL

MOV CL, 0x00

JMP FINIS

L1: MOV CL, 0xFF

FINIS: NOP

What will CL contain after execution?

CL = FF

If AL == 0then goto L1 else next line

LOOP (Loop according to RCX counter)

DEC RCX if RCX ≠ 0 then loop to label else next instruction

*can also be register ECX or CX

Flags affected: none

LOOP (Loop according to RCX counter)

DEC RCX
if RCX ≠ 0 then loop to label
else next instruction

*can also be register ECX or CX

Flags affected: none

```
section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP
```

- 1. What will RCX contain after execution?
- 2. What will AL contain after execution?

LOOP (Loop according to RCX counter)

DEC RCX
if RCX ≠ 0 then loop to label
else next instruction

*can also be register ECX or CX

Flags affected: none

```
section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP
```

- 1. What will RCX contain after execution?
- 2. What will AL contain after execution?

Example:

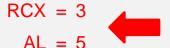
section .text MOV RCX, 03

MOV AL, 05

L1: DEC AL

L00P L1

FINIS: NOP



Example:

MOV RCX, 03
MOV AL, 05
L1: DEC AL
LOOP L1
FINIS: NOP

RCX = 3 AL = 4

```
section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP
```



```
section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1 Is RCX equal to 0?

FINTS: NOP
```

```
RCX = 2

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1  Is RCX equal to 0?

FINIS: NOP  No, then proceed.
```

Example:

section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP

RCX = 2

AL = 4

Continue until RCX is equal to 0

Example:

section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP

RCX = 1

AL = 3

Continue until RCX is equal to 0

x86-64 Control Transfer Instructions: LOOP

Example:

section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP

RCX = 0

AL = 2

Continue until RCX is equal to 0

x86-64 Control Transfer Instructions: LOOP

Example:

section .text

MOV RCX, 03

MOV AL, 05

L1: DEC AL

LOOP L1

FINIS: NOP

RCX = 0

AL = 2

Stop at NOP

x86-64 Control Transfer Instructions: JRCXZ, JECXZ, JCXZ

JRCXZ

Syntax: JRCXZ <label>

if RCX == 0 then goto label

Flags affected: none

JECXZ

Syntax: JECXZ <label>

if ECX == 0 then goto label

Flags affected: none

JCXZ

Syntax: CXZ <label>

if CX == 0 then goto label

Flags affected: none

x86-64 Control Transfer Instructions: JRCXZ, JECXZ, JCXZ

JRCXZ

Syntax: JRCXZ <label>

if RCX == 0 then goto label

Flags affected: none

JECXZ

Syntax: JECXZ <label>

if ECX == 0 then goto label

Flags affected: none

JCXZ

Syntax: CXZ < label>

if CX == 0 then goto label

Flags affected: none

Example:

section .text

MOV RCX, 0x0000000000000000

MOV RBX, 0x00000000000000005

JRCXZ FINIS

L1: DEC RBX

LOOP L1

FINIS: NOP

1. What will RBX contain after execution?

x86-64 Control Transfer Instructions: JRCXZ, JECXZ, JCXZ

JRCXZ

Syntax: JRCXZ <label>

if RCX == 0 then goto label

Flags affected: none

JECXZ

Syntax: JECXZ <label>

if ECX == 0 then goto label

Flags affected: none

JCXZ

Syntax: CXZ < label>

if CX == 0 then goto label

Flags affected: none

Example:

section .text

JRCXZ FINIS

L1: DEC RBX

LOOP L1

FINIS: NOP

1. What will RBX contain after execution?

RBX = 000000000000005

Exercise

Write an x86-64 program to count the number of elements in the list found in memory location ALPHA. The list is terminated by 0. Store the count in memory location COUNT. Output the count as well.

section .data ALPHA db 0x12, 0x34, 0x56, 0x99, 0x00 COUNT db 0x00

section .text
global CMAIN
CMAIN:
;write your code here
xor rax, rax
ret

address	Memory data (byte)	
COUNT	00	
	00	
	99	
	56	
	34	
ALPHA	12	

x86-64 Control Transfer Instructions: PUSH

PUSH(Push onto Stack)

```
PUSH src
src: r16_64
src: m16 64
src: imm8_16_32
8-bit immediate is sign-extended
RSP ← RSP – 8 |
RSP \leftarrow RSP - 2
[RSP] ← source
*No flags affected
```

Example:

```
section .text
MOV RAX, 0x123456789ABCDEF0
PUSH RAX
```

NOTE: **Does not support 32-bit operands** immediate is treated as 64-bit sign-extended

x86-64 Control Transfer Instructions: PUSH

PUSH(Push onto Stack)

Example:

section .text

MOV RAX, 0x123456789ABCDEF0

PUSH RAX



address	Stack Memory
0060FE39	
0060FE38	12
0060FE37	34
0060FE36	56
0060FE35	78
0060FE34	9A
0060FE33	ВС
0060FE32	DE
0060FE31	F0

x86-64 Control Transfer Instructions: POP

POP (pop a value from the stack)

```
POP dst
dst: r16 64
dst: m16 64
dst←[rsp]
RSP ← RSP + 8 |
RSP \leftarrow RSP + 2
*No flags affected
```

Example:

```
section .text POP RAX
```

NOTE: Does not support 32-bit operands

x86-64 Control Transfer Instructions: POP

POP (pop a value from the stack)

Example:

section .text
POP RAX

address	Stack Memory
0060FE39	
0060FE38	12
0060FE37	34
0060FE36	56
0060FE35	78
0060FE34	9A
0060FE33	ВС
0060FE32	DE
0060FE31	F0 O

RAX
(register)
72
34
3 6
78
<i>97</i> X
BC
ØE
FO

RSP

x86-64 Control Transfer Instructions: CALL/RET

CALL RET

The address of the next instruction (RIP) is push to the stack

The 8-byte data is POP'd to RIP

Example:

assume return address

Example:

Test:

call Test

ret

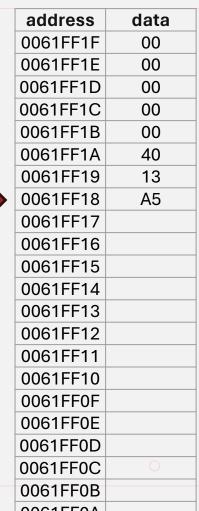
ret

of this 0x0000 0000 0040 14E5

xor rax, rax

PRINT STRING

"Stack"



RSP

0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

assume return address

Example:

Test:

call Test

ret

ret

return address

of this 0x0000 0000 0040 14E5

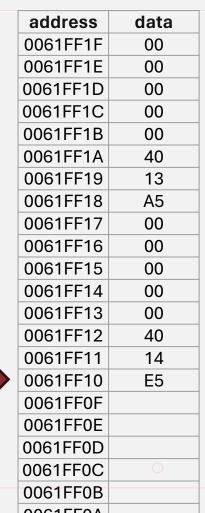
xor rax, rax

PRINT STRING

call Test pushes the

"Stack"

RSP



0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

assume return address

Example:

Test:

call Test

ret

ret

Then jumps to

Test function

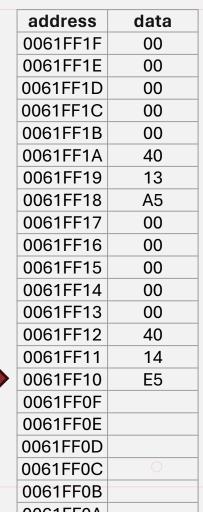
of this 0x0000 0000 0040 14E5

xor rax, rax

PRINT STRING

"Stack"

RSP



0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

assume return address

Example:

Test:

call Test

ret

ret

command

Execute function

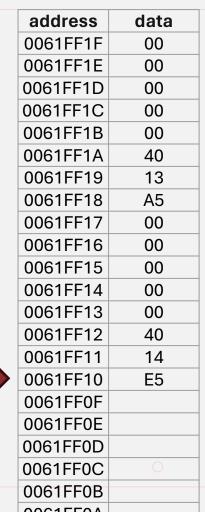
of this 0x0000 0000 0040 14E5

xor rax, rax

PRINT STRING

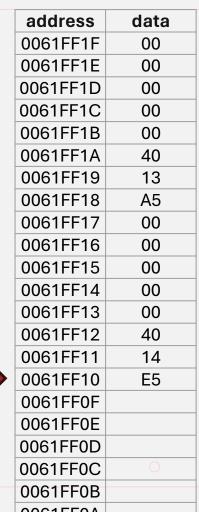
"Stack"

RSP



0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory



0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

RSP 0x061FF10

assume return address

Example:

Test:

RIP

call Test

ret

ret

of this 0x0000 0000 0040 14E5

xor rax, rax

PRINT STRING

ret pops the return

address and give to

"Stack"

RSP

assume return address

Example:

Test:

call Test

ret

ret

of this 0x0000 0000 0040 14E5

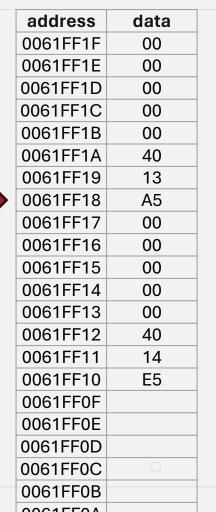
xor rax, rax

PRINT STRING

Then jumps to return

address based on RIP

"Stack"



RSP

0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

address	data
0061FF1F	00
0061FF1E	00
0061FF1D	00
0061FF1C	00
0061FF1B	00
0061FF1A	40
0061FF19	13
0061FF18	A5
0061FF17	00
0061FF16	00
0061FF15	00
0061FF14	00
0061FF13	00
0061FF12	40
0061FF11	14
0061FF10	E5
0061FF0F	
0061FF0E	
0061FF0D	
0061FF0C	0
0061FF0B	

0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

RSP 0x061FF20

Test:

Lastly, return properly to 0x0000_0000_0040_13A5

Example:

call Test

ret

assume return address

of this 0x0000 0000 0040 14E5

xor rax, rax

PRINT STRING "Stack"

What's Wrong Here?

Higher memory address

Lower memory address

section .text rax, 0x1111 1111 1111 1111 mov rbx, 0x2222 2222 2222 2222 push rax push rbx call func1 xor rax, rax ret func1: pop rcx PRINT HEX 8, rcx rdx pop PRINT HEX 8, rdx ret

STACK MEMORY

address	data	
061FF1F	00	
061FF1E	00	
061FF1D	00	
061FF1C	00	
061FF1B	00	
061FF1A	40	
061FF19	13	
061FF18	A5	

0061FF17

0061FF16

0061FF15

0061FF14

0061FF13

0061FF12

0061FF11

0061FF10

0061FF0F

0061FF0E 0061FF0D 0061FF0C

0061FF0B

0x0000_0000_0040_13A5 is the return address of SASM What's Wrong Here? **Higher memory address** section .text rax, 0x1111 1111 1111 1111 mov rbx, 0x2222 2222 2222 2222 push rax push rbx **RSP** call func1 xor rax, rax ret func1: pop rcx PRINT HEX 8, rcx 0061FF11 rdxpop 0061FF10 PRINT HEX 8, rdx 0061FF0F ret 0061FF0E 0061FF0D 0061FF0C Lower memory address 0061FF0B

STACK MEMORY

address	data
0061FF1F	00
0061FF1E	00
0061FF1D	00
0061FF1C	00
0061FF1B	00
0061FF1A	40
0061FF19	13
0061FF18	A5
0061FF17	
0061FF16	
0061FF15	
0061FF14	
0061FF13	
0061FF12	

0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

RSP 0000_0000_0061_FF18

RAX

RCX

RBX

RDX

STACK MEMORY What's Wrong Here? address data **Higher memory address** 0061FF1F 00 0061FF1E 00 section .text 0061FF1D 00 rax, 0x1111 1111 1111 1111 mov 0061FF1C 00 rbx, 0x2222 2222 2222 2222 0061FF1B 00 address of the 0061FF1A 40 push rax 0061FF19 13 push rbx **RSP** 0061FF18 **A5** call func1 **RSP** 0061FF17 xor rax, rax 0061FF16 ret 0061FF15 **RAX** func1: 0061FF14 pop rcx 0061FF13 **RBX** PRINT HEX 8, rcx 0061FF12 rdx0061FF11 pop **RCX** 0061FF10 PRINT HEX 8, rdx 0061FF0F ret **RDX** 0061FF0E 0061FF0D 0061FF0C Lower memory address 0061FF0B

0x0000 0000 0040 13A5 is the return address of SASM

RSP contains the

stack memory 0000 0000 0061 FF18

1111 1111 1111 1111

What's Wrong Here? address data **Higher memory address** 0061FF1F 00 0061FF1E 00 section .text 0061FF1D 00 rax, 0x1111 1111 1111 1111 mov 0061FF1C 00 rbx, 0x2222 2222 2222 0061FF1B 00 push 0061FF1A 40 rax 0061FF19 13 push rbx **RSP** 0061FF18 **A5** call func1 0061FF17 xor rax, rax 0061FF16 ret 0061FF15 func1: 0061FF14 pop rcx 0061FF13 PRINT HEX 8, rcx 0061FF12 rdx0061FF11 pop 0061FF10 PRINT HEX 8, rdx 0061FF0F ret 0061FF0E 0061FF0D 0061FF0C Lower memory address 0061FF0B

STACK MEMORY

0x0000 0000 0040 13A5 is the return address of SASM

RSP contains the address of the

stack memory 0000 0000 0061 FF18 **RSP**

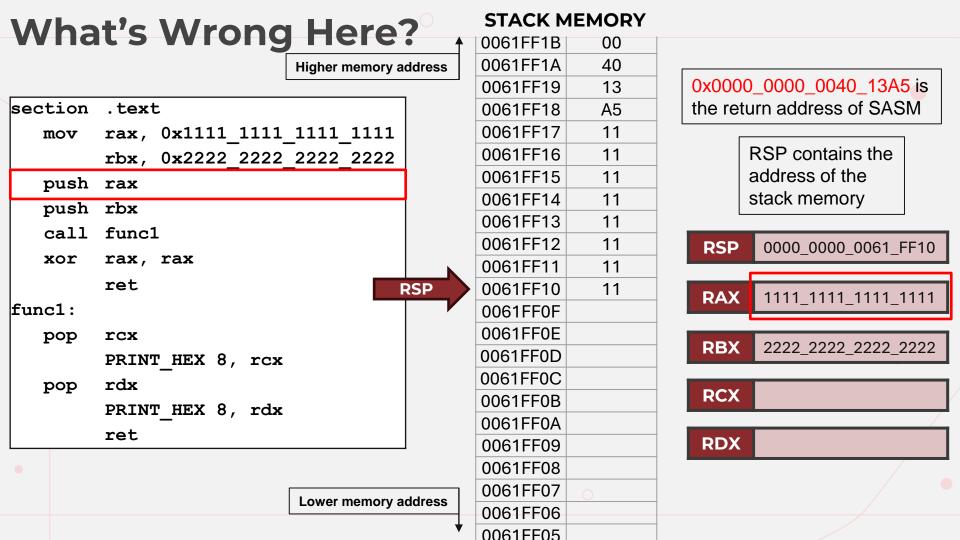
RAX 1111 1111 1111 1111

2222_2222_2222

RCX

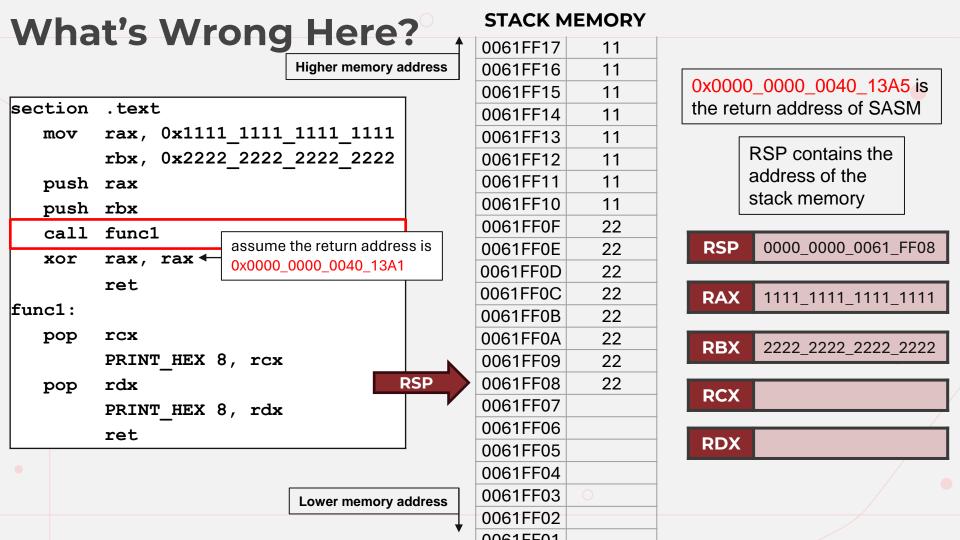
RBX

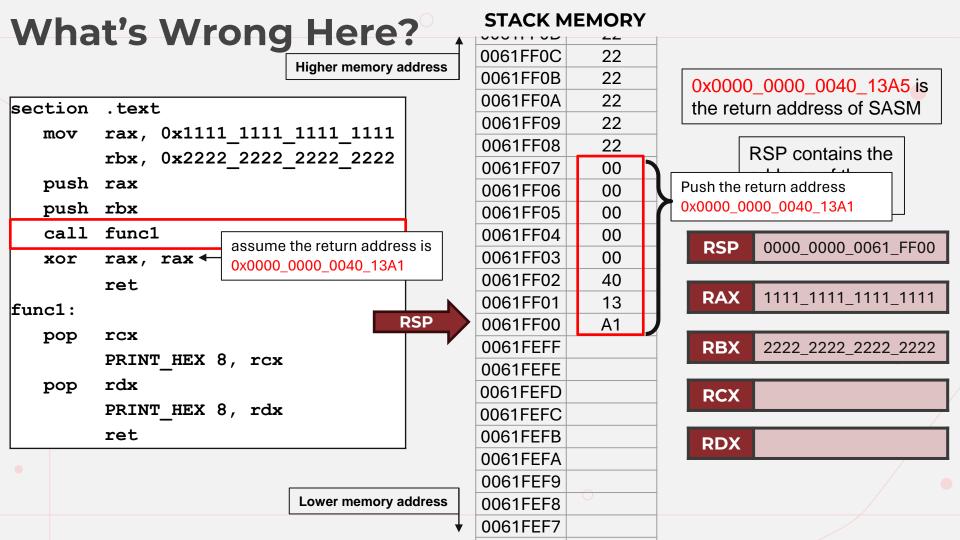
RDX

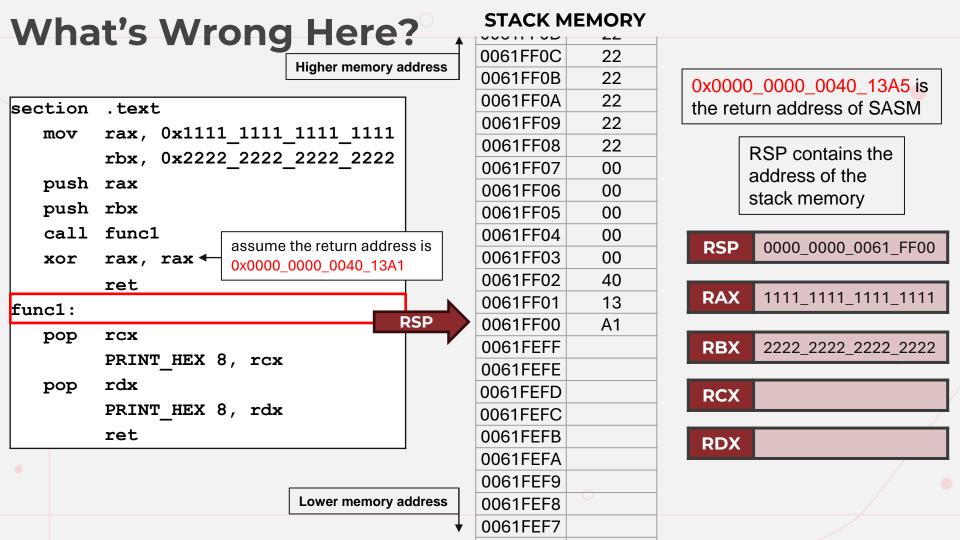


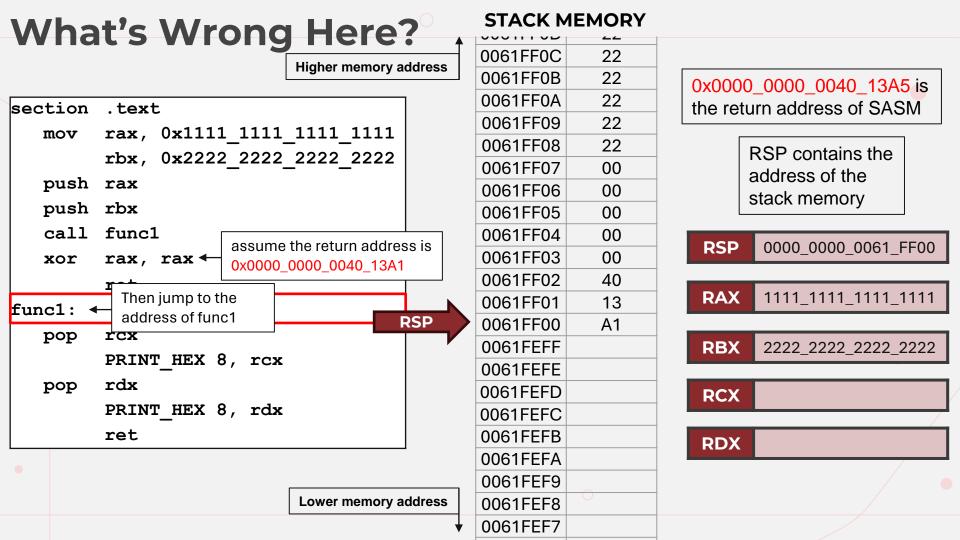
STACK MEMORY What's Wrong Here? 0061FF17 11 **Higher memory address** 0061FF16 11 0x0000 0000 0040 13A5 is 0061FF15 11 section the return address of SASM .text 0061FF14 11 rax, 0x1111 1111 1111 1111 mov 11 0061FF13 RSP contains the rbx, 0x2222 2222 2222 2222 11 0061FF12 address of the 11 0061FF11 push rax stack memory 0061FF10 11 push rbx 22 0061FF0F call func1 0000 0000 0061 FF08 **RSP** 0061FF0E 22 xor rax, rax 22 0061FF0D ret 22 0061FF0C **RAX** 1111 1111 1111 1111 func1: 0061FF0B 22 pop rcx 22 0061FF0A **RBX** 2222 2222 2222 2222 PRINT HEX 8, rcx 0061FF09 22 **RSP** 0061FF08 22 rdxpop **RCX** 0061FF07 PRINT HEX 8, rdx 0061FF06 ret **RDX** 0061FF05 0061FF04 0061FF03 Lower memory address 0061FF02

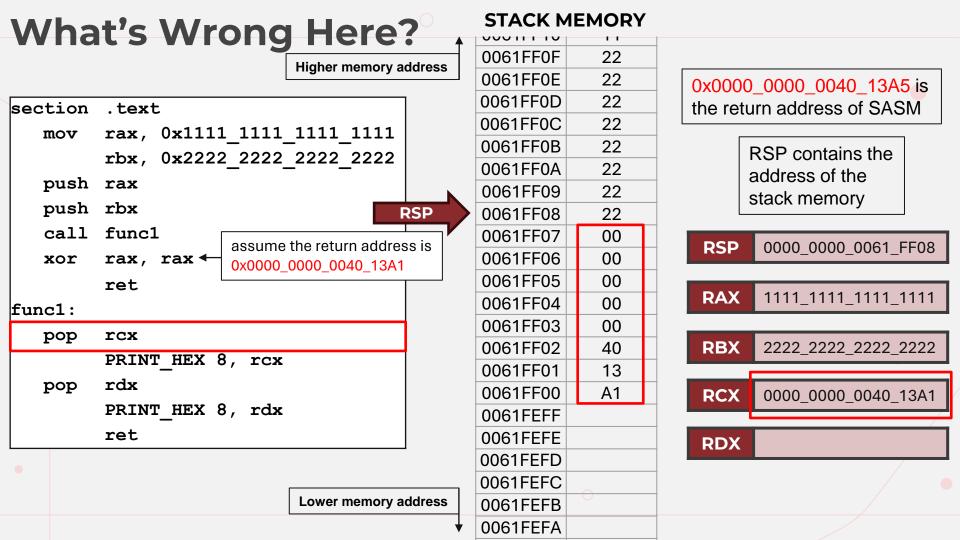
0061 FF01

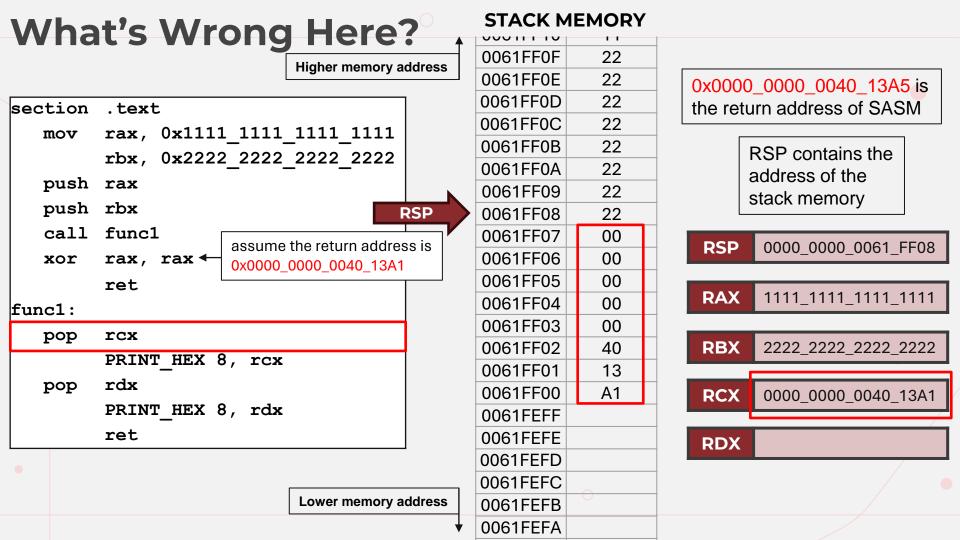


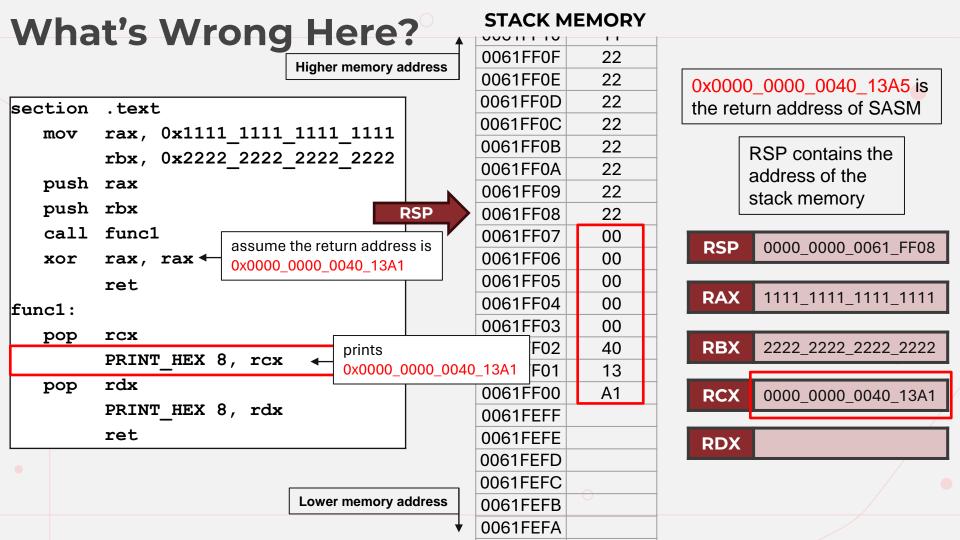




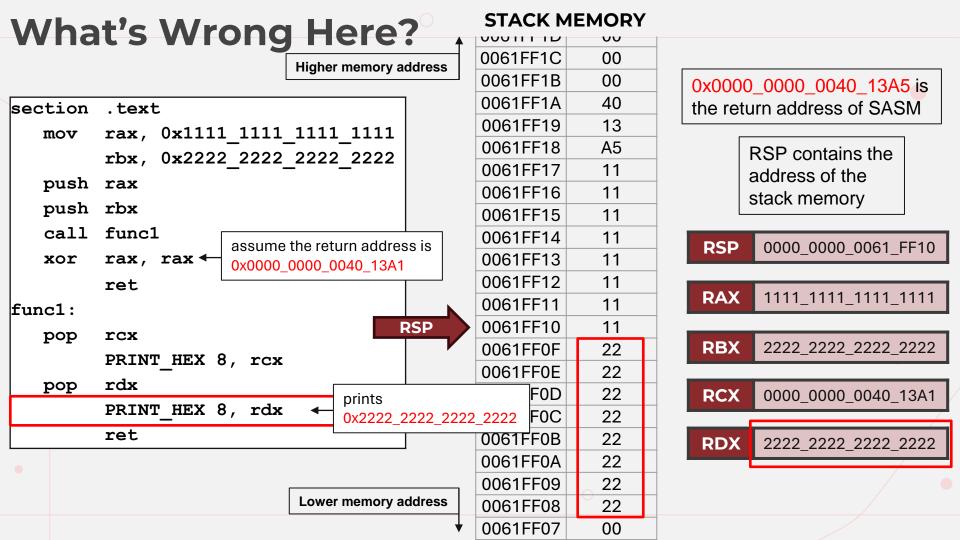


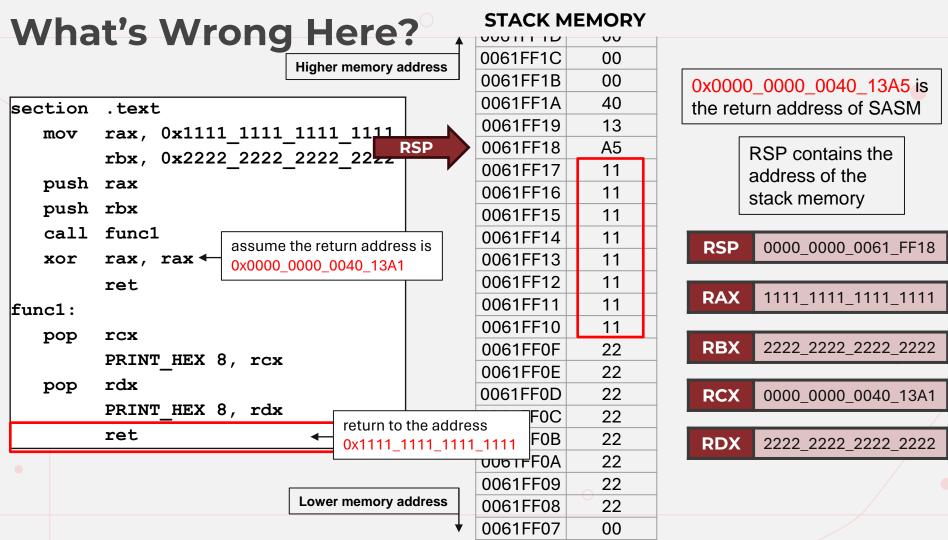


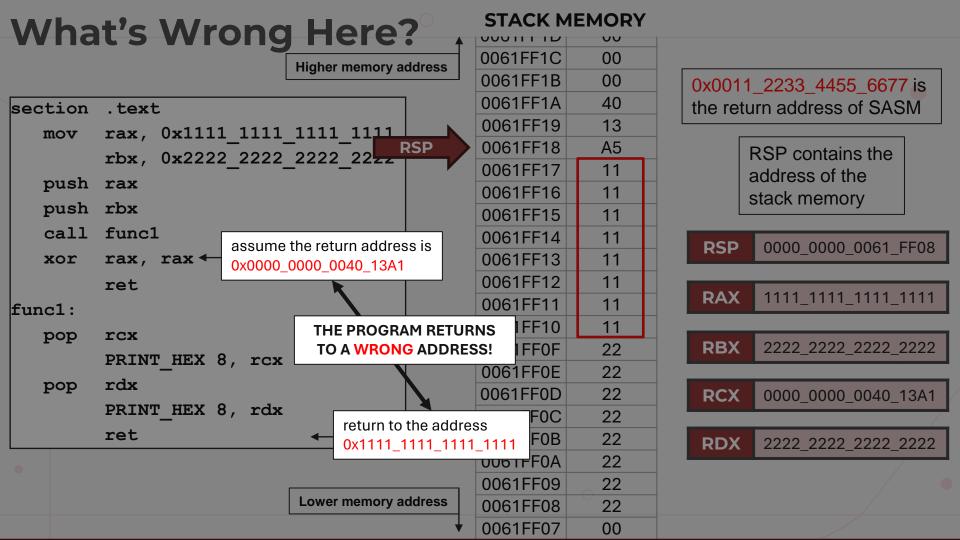




STACK MEMORY What's Wrong Here? טווווטטט 0061FF1C 00 **Higher memory address** 0061FF1B 00 0x0000 0000 0040 13A5 is 0061FF1A 40 section the return address of SASM .text 0061FF19 13 rax, 0x1111 1111 1111 1111 mov 0061FF18 **A5** RSP contains the rbx, 0x2222 2222 2222 2222 0061FF17 11 address of the push rax 0061FF16 11 stack memory push rbx 11 0061FF15 call func1 11 0061FF14 assume the return address is 0000 0000 0061 FF10 **RSP** 0061FF13 11 xor rax, rax ← 0x0000 0000 0040 13A1 0061FF12 11 ret **RAX** 1111 1111 1111 1111 0061FF11 11 func1: **RSP** 0061FF10 11 rcx pop **RBX** 2222 2222 2222 2222 0061FF0F 22 PRINT HEX 8, rcx 0061FF0E 22 rdxpop 0061FF0D 22 **RCX** 0000 0000 0040 13A1 PRINT HEX 8, rdx 0061FF0C 22 ret 22 0061FF0B **RDX** 2222_2222_2222 0061FF0A 22 0061FF09 22 Lower memory address 0061FF08 22 00 0061FF07







What's Wrong Here?

Higher memory address

Lower memory address

section	.text
mov	rax, 0x1111_1111_1111
	rbx, 0x2222_2222_2222
push	rax
push	rbx
call	func1
xor	rax, rax
	ret
func1:	
mov	rbp, rsp
mov	rcx, [rbp+8]
mov	rdx, [rbp+16]
	PRINT_HEX 8, rcx
	NEWLINE
	PRINT_HEX 8, rdx
	ret

STACK MEMORY

address	data
0061FF1F	00
0061FF1E	00
0061FF1D	00
0061FF1C	00
0061FF1B	00
0061FF1A	40
0061FF19	13
0061FF18	A5

0061FF17 0061FF16 0061FF15 0061FF14 0061FF13 0061FF12 0061FF10 0061FF0F 0061FF0E 0061FF0D 0061FF0D

0061FF0B

0x0000_0000_0040_13A5 is the return address of SASM

What's Wrong Here? **Higher memory address** section .text rax, 0x1111 1111 1111 1111 mov rbx, 0x2222 2222 2222 2222 push rax push rbx call func1 assume the return address is rax, rax ← xor 0x0000 0000 0040 13A1 ret func1: rbp, rsp mov rcx, [rbp+8] mov rdx, [rbp+16] mov PRINT HEX 8, rcx NEWLINE PRINT HEX 8, rdx ret Lower memory address

STACK MEMORY

	_
address	data
0061FF1F	00
0061FF1E	00
0061FF1D	00
0061FF1C	00
0061FF1B	00
0061FF1A	40
0061FF19	13
0061FF18	A5
0061FF17	
0061FF16	
0061FF15	
0061FF14	
0061FF13	
0061FF12	
0061FF11	
0061FF10	
0061FF0F	
0061FF0E	
0061FF0D	
0061FF0C	0
0061FF0B	

0004 EE0 4

0x0000_0000_0040_13A5 is the return address of SASM

RSP contains the address of the stack memory

RSP 0000_0000_0061_FF18

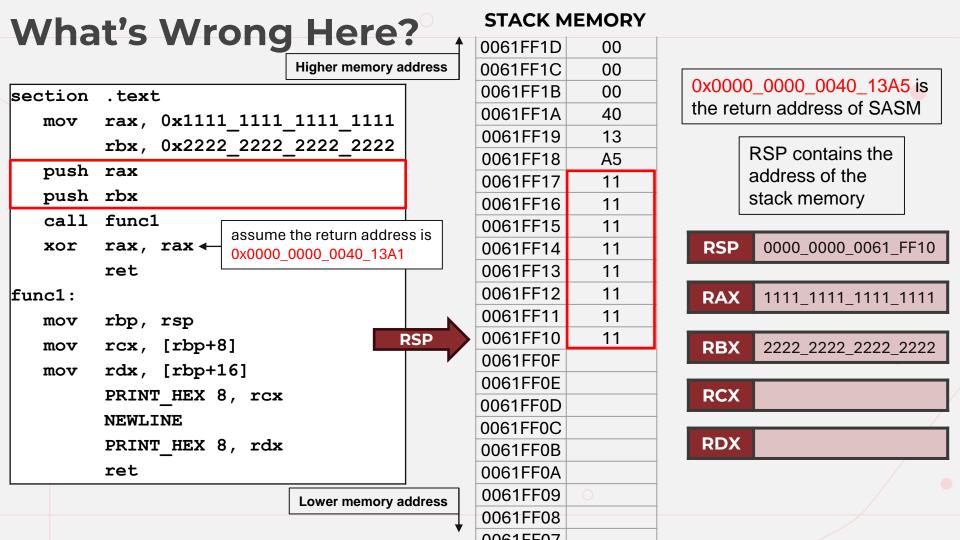
RAX 1111_1111_1111

2222 2222 2222 2222

RCX

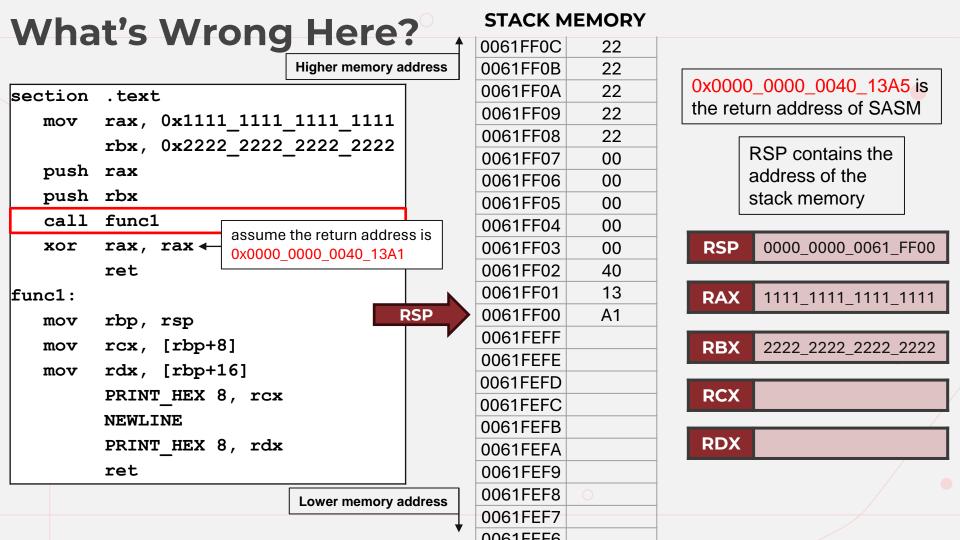
RBX

RDX



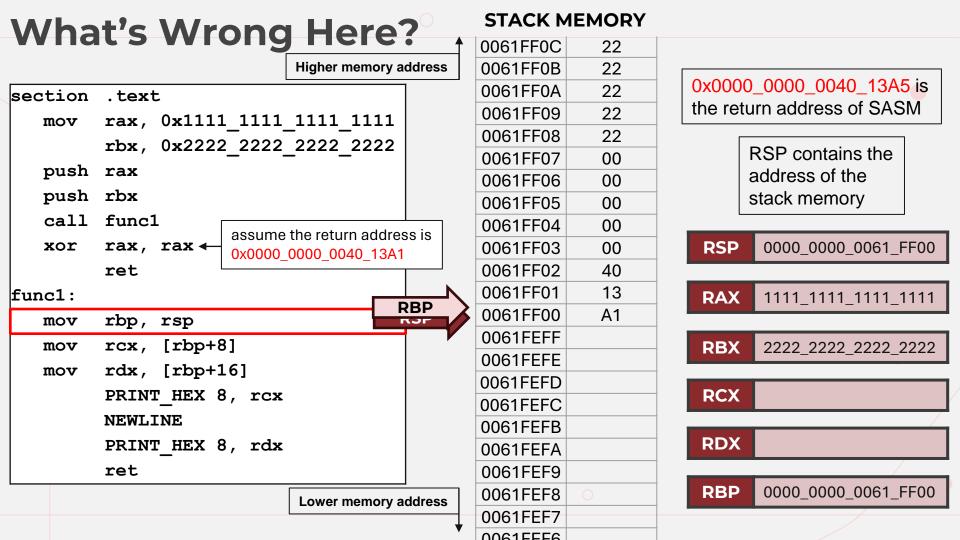
STACK MEMORY What's Wrong Here? 00011113 0061FF18 **A5 Higher memory address** 0061FF17 11 0x0000 0000 0040 13A5 is section .text 11 0061FF16 the return address of SASM rax, 0x1111 1111 1111 1111 mov 0061FF15 11 rbx, 0x2222 2222 2222 2222 0061FF14 11 RSP contains the push rax 0061FF13 11 address of the 0061FF12 11 push rbx stack memory 0061FF11 11 call func1 assume the return address is 0061FF10 11 0000 0000 0061 FF08 rax, **RSP** xor rax ← 0x0000 0000 0040 13A1 0061FF0F 22 ret 0061FF0E 22 func1: RAX 1111 1111 1111 1111 0061FF0D 22 rbp, rsp mov 0061FF0C 22 rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 0061FF0B 22 rdx, [rbp+16] mov 0061FF0A 22 0061FF09 22 PRINT HEX 8, rcx **RCX** 0061FF08 22 **RSP** NEWLINE 0061FF07 **RDX** PRINT HEX 8, rdx 0061FF06 ret 0061FF05 Lower memory address 0061FF04 0061FF03

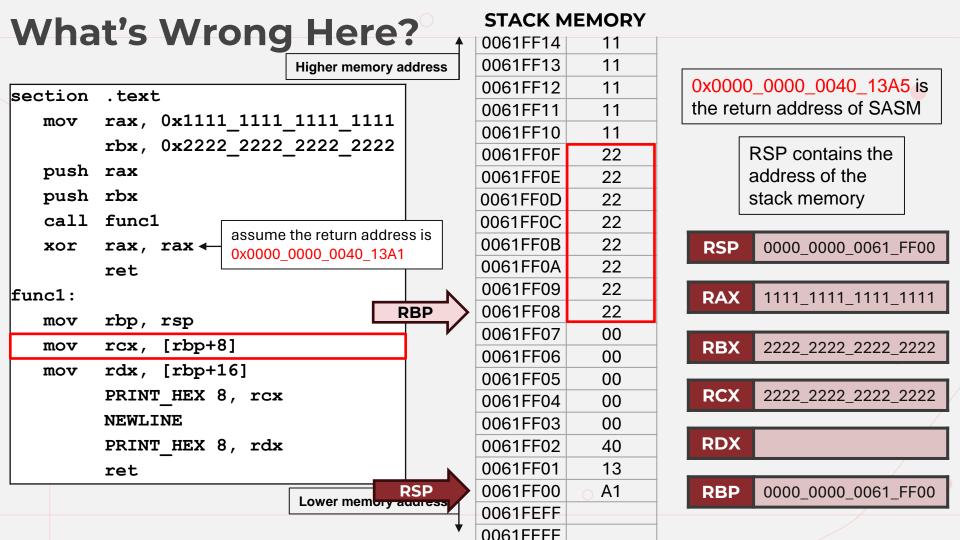
STACK MEMORY What's Wrong Here? 00011113 0061FF18 **A5 Higher memory address** 0061FF17 11 0x0000 0000 0040 13A5 is section .text 11 0061FF16 the return address of SASM rax, 0x1111 1111 1111 1111 mov 0061FF15 11 rbx, 0x2222 2222 2222 2222 0061FF14 11 RSP contains the 0061FF13 11 push rax address of the 11 0061FF12 push rbx stack memory 0061FF11 11 call func1 assume the return address is 0061FF10 11 0000 0000 0061 FF08 xor rax, rax ← **RSP** 0x0000 0000 0040 13A1 0061FF0F 22 ret 0061FF0E 22 func1: RAX 1111 1111 1111 1111 0061FF0D 22 rbp, rsp mov 0061FF0C 22 rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 22 0061FF0B rdx, [rbp+16] mov 0061FF0A 22 0061FF09 22 PRINT HEX 8, rcx **RCX** 22 **RSP** 0061FF08 NEWLINE 0061FF07 **RDX** PRINT HEX 8, rdx 0061FF06 ret 0061FF05 Lower memory address 0061FF04 0061FF03



STACK MEMORY What's Wrong Here? 0061FF0C 22 **Higher memory address** 0061FF0B 22 0x0000 0000 0040 13A5 is 0061FF0A 22 section .text the return address of SASM 22 0061FF09 rax, 0x1111 1111 1111 1111 mov 0061FF08 22 rbx, 0x2222 2222 2222 RSP contains the 0061FF07 00 push rax address of the 0061FF06 00 push rbx stack memory 0061FF05 00 call func1 0061FF04 00 assume the return address is 0000 0000 0061 FF00 rax, rax ← **RSP** xor 0061FF03 00 0x0000 0000 0040 13A1 ret 0061FF02 40 13 0061FF01 func1: RAX 1111 1111 1111 1111 **RSP** 0061FF00 A1 mov rbp, rsp 0061FEFF rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 0061FEFE rdx, [rbp+16] mov 0061FEFD PRINT HEX 8, rcx **RCX** 0061FEFC NEWLINE 0061FEFB **RDX** PRINT HEX 8, rdx 0061FEFA ret 0061FEF9 0061FEF8 Lower memory address 0061FEF7

OOG1EEEG

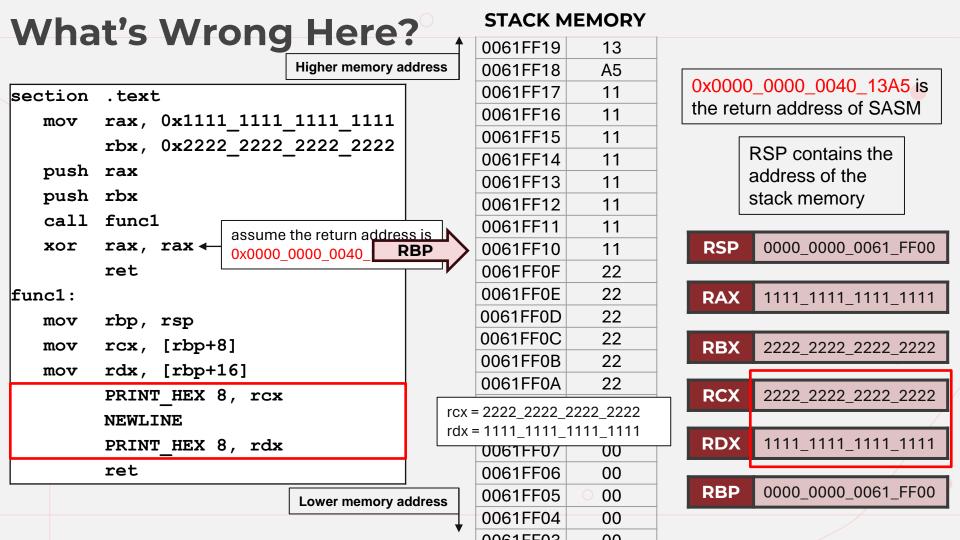


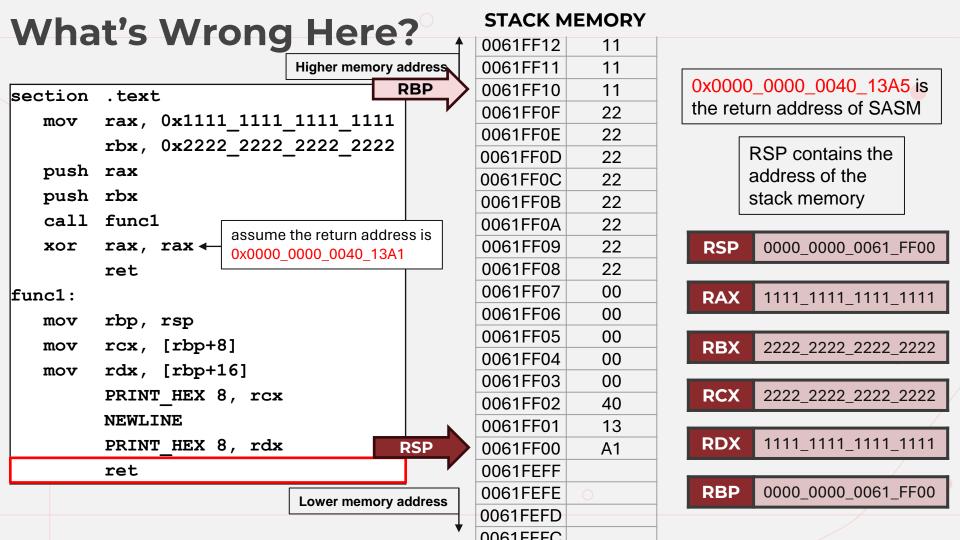


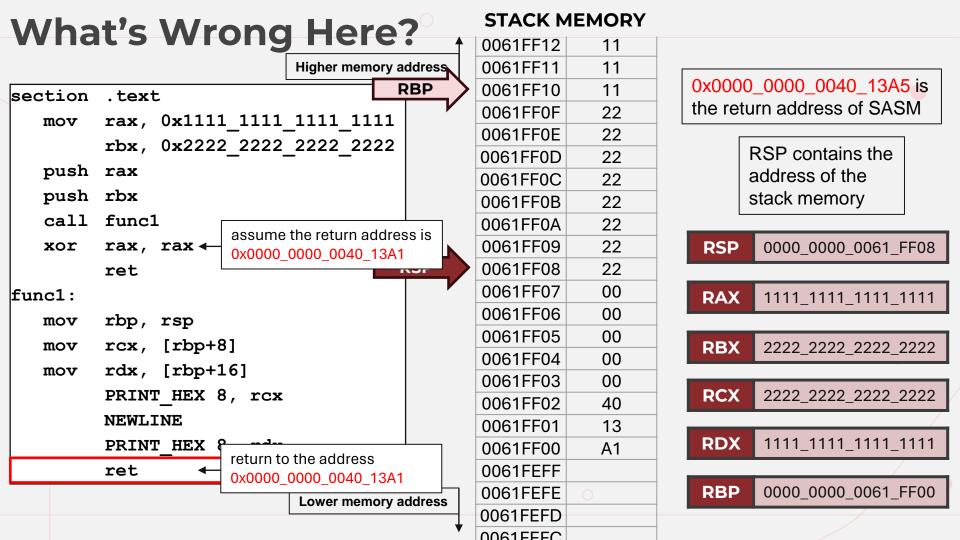
STACK MEMORY What's Wrong Here? 0061FF19 13 **Higher memory address** 0061FF18 **A5** 0x0000 0000 0040 13A5 is 11 0061FF17 section .text the return address of SASM 0061FF16 11 rax, 0x1111 1111 1111 1111 mov 11 0061FF15 rbx, 0x2222 2222 2222 2222 RSP contains the 0061FF14 11 push rax address of the 0061FF13 11 push rbx stack memory 0061FF12 11 call func1 0061FF11 11 assume the return address is 0000 0000 0061 FF00 xor rax, **RSP** rax ← 0061FF10 11 **RBP** 0x0000 0000 0040 ret 0061FF0F 22 0061FF0E 22 func1: RAX 1111 1111 1111 1111 0061FF0D 22 rbp, rsp mov 22 0061FF0C rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 0061FF0B 22 rdx, [rbp+16] mov 0061FF0A 22 PRINT HEX 8, rcx **RCX** 2222 2222 2222 2222 0061FF09 22 NEWLINE 0061FF08 22 **RDX** 1111 1111 1111 1111 PRINT HEX 8, rdx 0061FF07 00 ret 0061FF06 00 0000_0000_0061_FF00 **RBP** 0061FF05 00 Lower memory address 0061FF04 00

0001EE02

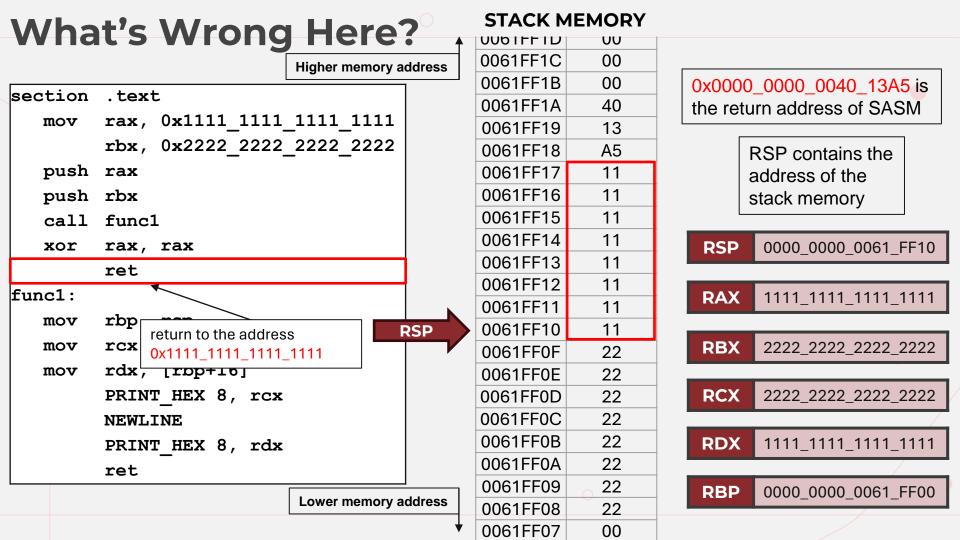
 Δ

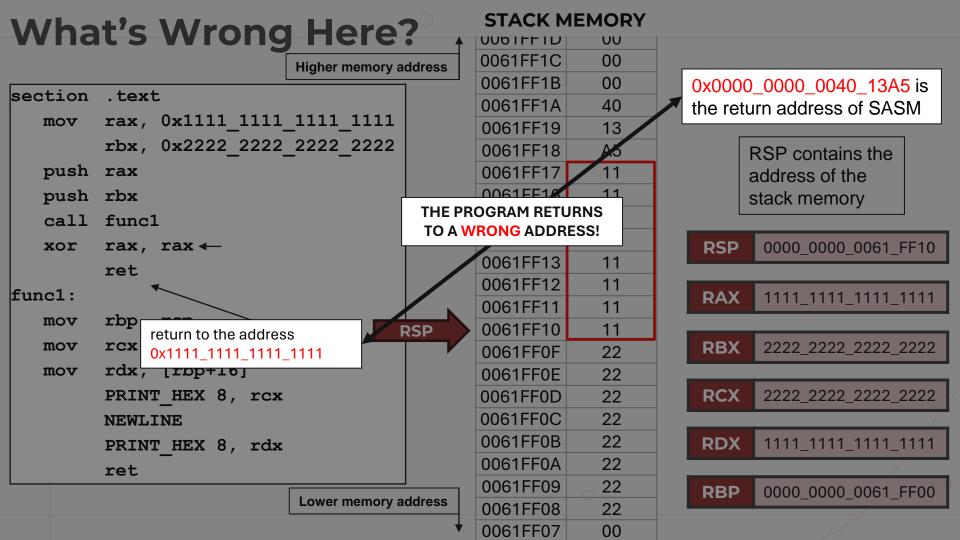


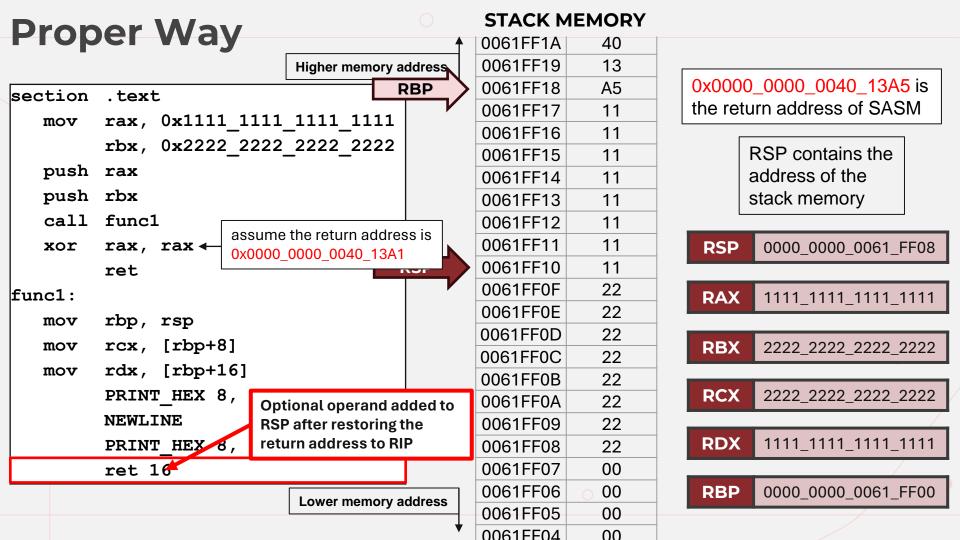




STACK MEMORY What's Wrong Here? 0061FF1A 40 **Higher memory address** 0061FF19 13 0x0000 0000 0040 13A5 is 0061FF18 **A5** section .text the return address of SASM 0061FF17 11 rax, 0x1111 1111 1111 1111 mov 0061FF16 11 rbx, 0x2222 2222 2222 2222 RSP contains the 11 0061FF15 push rax address of the 11 0061FF14 push rbx stack memory 11 0061FF13 call func1 0061FF12 11 0000 0000 0061 FF08 rax, rax 0061FF11 11 **RSP** xor **RSP** ret 0061FF10 11 0061FF0F 22 func1: RAX 1111 1111 1111 1111 0061FF0E 22 mov rbp, rsp 0061FF0D 22 rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 22 0061FF0C rdx, [rbp+16] mov 0061FF0B 22 PRINT HEX 8, rcx **RCX** 2222 2222 2222 2222 22 0061FF0A NEWLINE 0061FF09 22 **RDX** 1111 1111 1111 1111 PRINT HEX 8, rdx 22 0061FF08 ret 0061FF07 00 0000_0000_0061_FF00 **RBP** 0061FF06 00 Lower memory address 00 0061FF05 0061EE04 $\cap \cap$







Proper Way Higher memory address section .text rax, 0x1111 1111 1111 1111 mov rbx, 0x2222 2222 2222 2222 push rax push rbx call func1 xor rax, rax ret func1:

mov rbp, rsp rcx, [rbp+8] mov

rdx, [rbp+16] mov PRINT HEX 8, rcx

> NEWLINE PRINT HEX 8, rdx

ret 16

STACK MEMORY address data

0061FF1F 00 0061FF1E 00 0061FF1D 00 0061FF1C 00

0061FF1B 00 0061FF1A 40

0061FF19 13 0061FF18 **A5** 0061FF17 11

0061FF16 11 0061FF15 11 0061FF14 11 0061FF13 11

0061FF12 11 0061FF11 11

0061FF10 11 0061FF0F 22 0061FF0E 22

0061FF0D

0061FF0C

0061FF0B

22

22

 \sim

22

0x0000 0000 0040 13A5 is the return address of SASM

> RSP contains the address of the stack memory

0000 0000 0061 FF10 **RSP**

RAX 1111 1111 1111 1111

RBX 2222 2222 2222 2222 **RCX** 2222 2222 2222 2222

RDX 1111 1111 1111 1111

0000_0000_0061_FF00 **RBP**

Lower memory address

RSP

STACK MEMORY **Proper Way** address data **Higher memory address** 0061FF1F 00 0x0000 0000 0040 13A5 is 0061FF1E 00 section .text 0061FF1D 00 0x1111 1111 1111 1111 rax, mov 0061FF1C 00 rbx, 0x2222 2222 2222 2222 0061FF1B 00 push rax 0061FF1A 40 push rbx 0061FF19 13 call func1 **RSP** 0061FF18 **A5** 0000 0000 0061 FF18 xor rax, rax **RSP** 0061FF17 11 ret 0061FF16 11 add +16 to bypass func1: 0061FF15 11 RAX parameters 0061FF14 11 rbp, mov rsp 0061FF13 11 [rbp+8] rcx, **RBX** mov 0061FF12 11 rdx, [rbp+16] mov 0061FF11 11 PRINT HEX 8/, **RCX** rcx 0061FF10 11 NEWLINE 0061FF0F 22 **RDX** PRINT HEX 8, rdx 0061FF0E 22 ret 16 0061FF0D 22 **RBP** 0061FF0C 22 Lower memory address 0061FF0B 22

 \sim

the return address of SASM RSP contains the

address of the stack memory

1111 1111 1111 1111

2222 2222 2222 2222 2222 2222 2222 2222

1111 1111 1111 1111

0000_0000_0061_FF00

STACK MEMORY **Proper Way RSP** address data Higher memory address 0061FF1F 00 0x0000 0000 0040 13A5 is 0061FF1E 00 section .text the return address of SASM 0061FF1D 00 rax, 0x1111 1111 1111 1111 mov 00 0061FF1C rbx, 0x2222 2222 2222 RSP contains the 0061FF1B 00 push rax address of the 0061FF1 40 push rbx stack memory 0061FF19 13 call func1 0961FF18 **A5** 0000 0000 0061 FF20 rax, rax **RSP** xor 0061FF17 11 return to the address ret 11 0061FF16 0x0000 0000 0040 13A5 func1: 0061FF15 11 RAX 1111 1111 1111 1111 0061FF14 11 mov rbp, rsp 0061FF13 11 rcx, [rbp+8] **RBX** 2222 2222 2222 2222 mov 0061FF12 11 rdx, [rbp+16] mov 0061FF11 11 PRINT HEX 8, rcx **RCX** 2222 2222 2222 2222 **RBP** 0061FF10 11 NEWLINE 0061FF0F 22 **RDX** 1111 1111 1111 1111 PRINT HEX 8, rdx 0061FF0E 22 ret 16 0061FF0D 22 0000_0000_0061_FF00 **RBP** 22 0061FF0C Lower memory address 0061FF0B 22

 \sim

IF STATEMENT

Pseudo-code:

if (condition) then_block;

Assembly code:

1. Code to set EFLAGS

- 2. Jcc endif; select cc so that the condition is false
- 3.; code for then_block
- 4. endif

IF STATEMENT (Example)

```
if (RSI==RDI)
then EDX = 0x12345678;
```

	CMP RSI, RDI
	JNE endif
	MOV EDX, 0x12345678
endif:	; next instruction

IF-ELSE STATEMENT

Pseudo-code:

if (condition) then_block; else else_block;

- 1. Code to set EFLAGS
- 2. Jcc else_block; select cc so that the condition is false
- 3.; code for then_block
- 4. JMP endif
- 5. else_block:
- 6. endif:

IF-ELSE STATEMENT (Example)

```
if (RSI==RDI) then EDX = 0 \times 12345678;
else EDX = 0 \times 87654321
```

	CMP RSI, RDI
	JNE else
	MOV EDX, 0x12345678
	JMP endif
else:	MOV EDX, 0x87654321
endif:	; next instruction

WHILE LOOP STATEMENT

```
Pseudo-code (Condition Upon Entry):
while (condition) {
    body of loop; }
```

- 1. while:
- 2. Code to set EFLAGS
- 3. Jcc endwhile; select cc so that the condition is false
- 4.; body of loop
- 5. JMP while
- 6. endwhile:

WHILE LOOP STATEMENT (Example)

```
while (RCX<>0)
{EDX = EDX+1
RCX = RCX-1};
```

while:	CMP RCX,0000000000000000
	JE endwhile
	INC EDX
	DEC RCX
	JMP while
endwhile:	; next instruction

DO-WHILE LOOP STATEMENT

Pseudo-code (Condition Upon Exit):
do { body of loop; } while (condition)

- 1. do:
- 2.; body of loop
- 3. ; Code to set EFLAGS
- 4. Jcc do; select cc so that the condition is true

DO-WHILE LOOP STATEMENT (Example)

```
do
  {EDX = EDX+1;
  ECX = RCX-1;}
while RCX <>0
```

do:	INC EDX
	DEC RCX
	CMP ECX,00000000
	JNE do
	; next instruction

SWITCH STATEMENT

```
Pseudo-code:

SWITCH case_var

{{label_list}...:stmt; break};

DEFAULT

statements; break;

END;
```

	CMP case_var, case_list_1
	JNE label_list_2
	; instructions for case_list_1
	JMP end
label_list_2:	CMP case_var, case_list_2
	JNE label_list_n
	; instructions for case_list_2
	JMP end
label_list_n:	CMP case_var, case_list_other
	JNE default
	; instructions for case_list_n
	JMP end
default:	; instructions for case_list_default
end:	; next instructions
	CMP case_var, case_list_1

SWITCH STATEMENT (Example)

```
SWITCH (RSI) {
    1: EBX = EBX +1; break;
    2: EBX = EBX - 1; break;
    3: EBX = EBX + EBX; break;
    DEFAULT
    EBX = EBX - EBX; break;
}
```

	CMP RSI, 0000000000000001	
	JNE next_case_2	
	INC EBX	
	JMP end	
next_case_2:	CMP ESI, 00000000000000002	
	JNE next_case_3	
	DEC EBX	
	JMP end	
next_case_3:	CMP ESI, 0000000000000003	
	JNE default	
	ADD EBX, EBX	
	JMP end	
default:	SUB EBX, EBX	
end:	; next instructions	

FOR-NEXT LOOP STATEMENT

Pseudo-code (Condition Upon Exit):

- 1. Initialized CX/ECX/RCX
- 2. for:
- 3.; body of loop
- 4. LOOP for

FOR-NEXT LOOP STATEMENT (Example)

For ECX = 1 to 5 do EDX = EDX +1

	MOV RCX, 0000000000000005
for:	INC EDX
	LOOP for
	; next instruction