CSO-1

Stack and Functions

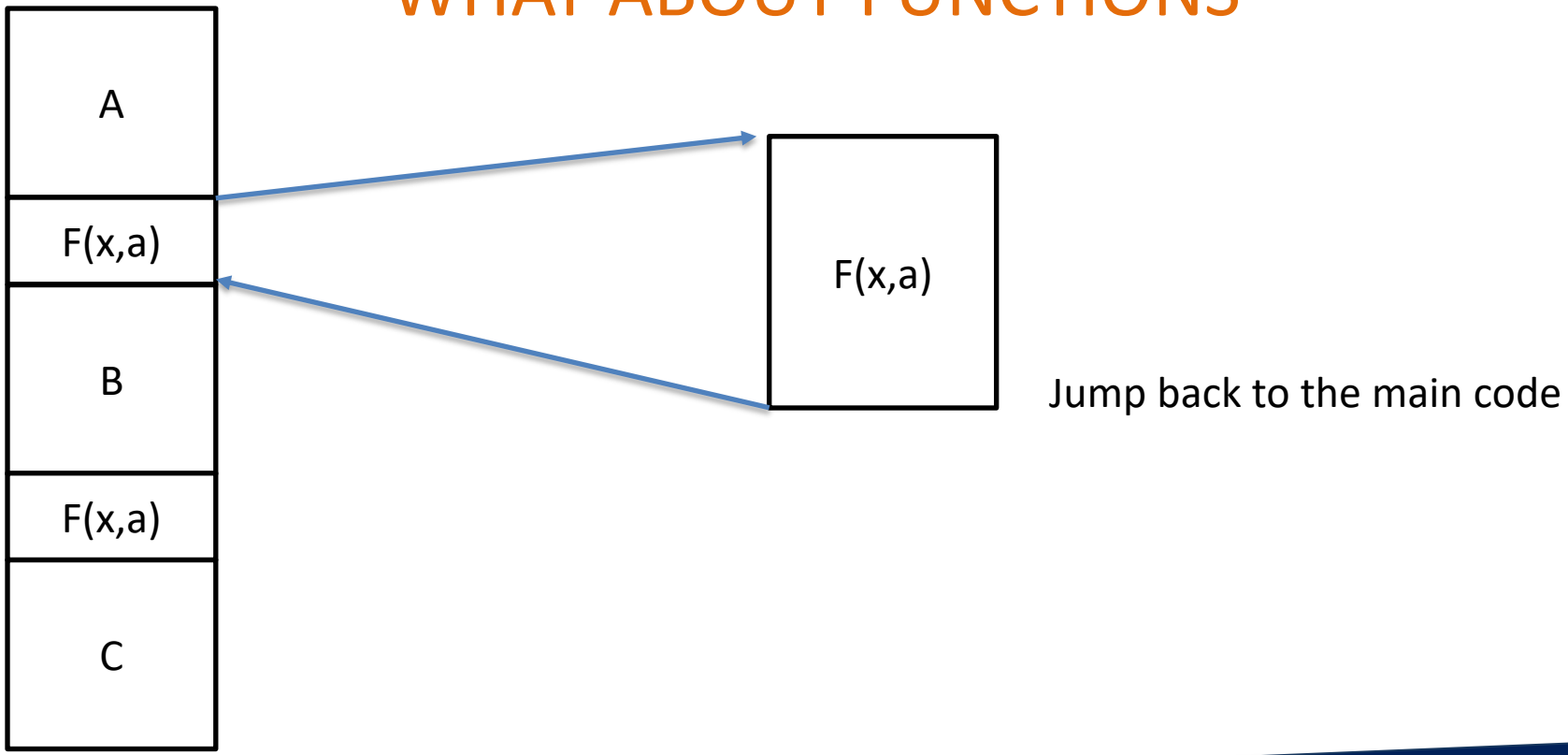
Daniel G. Graham PhD

THOUGHTS ON THE EXAM

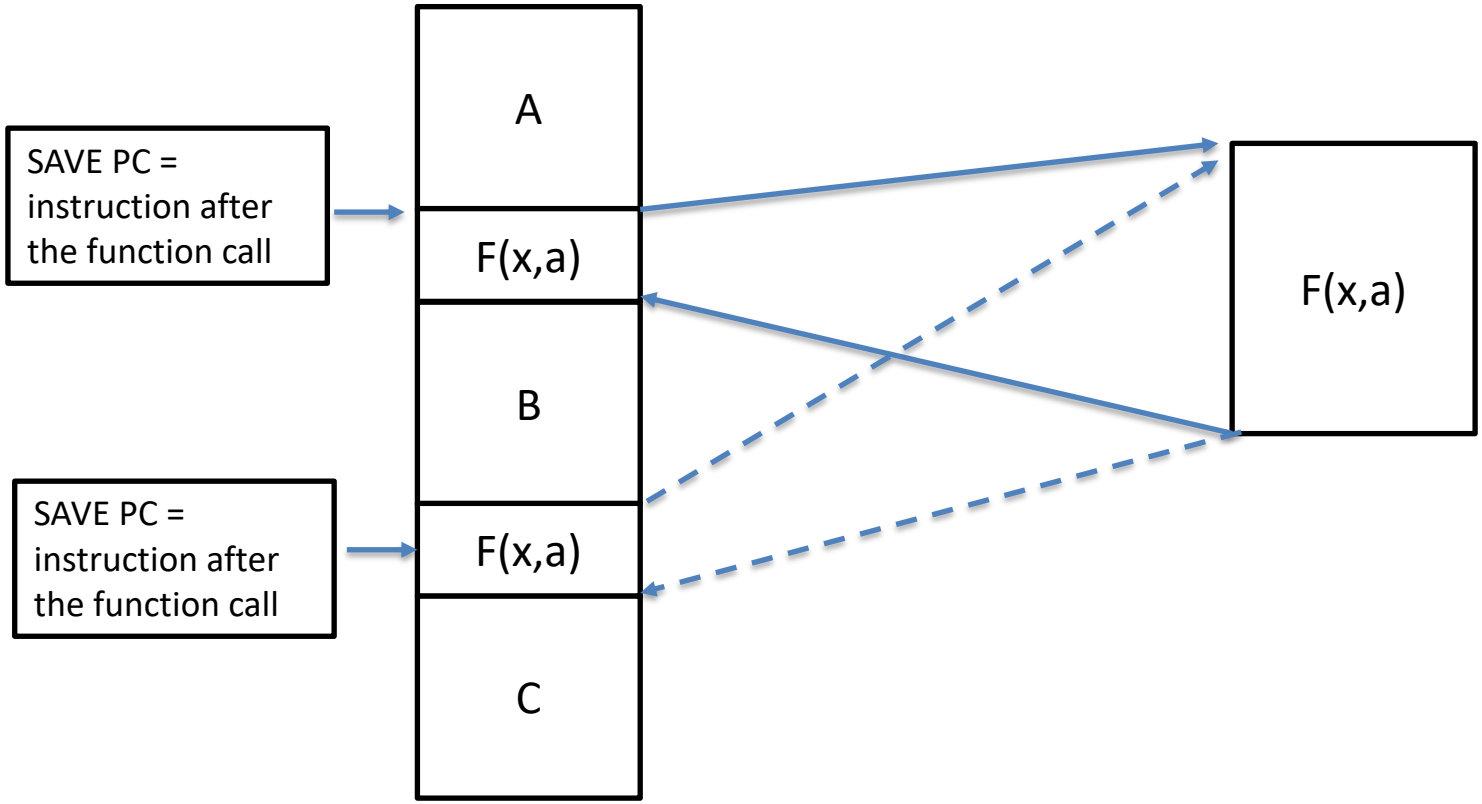
- I am so proud of you all.
    - It was challenging exam and you all faced it with grit
- I have read all the very thoughtful feedback on Piazza and other sites.
- The TA has always advocated for you. I should have paid closer attention to their feedback.
- The TA team and I will meet on Monday. Hopefully, you will have your grade back before lab on Tuesday.
- Hope that you all restful fall break.

1. Review of ISA
2. Review of ISA function instruction
3. Detailed Example Function call
4. Start Introduction to x86 assembly

WHAT ABOUT FUNCTIONS



Jump back to the main code


SAVE PC = instruction after the function call
A
F(x,a)
B
SAVE PC = instruction after the function call
F(x,a)
C
F(x,a)
Jump back to saved value

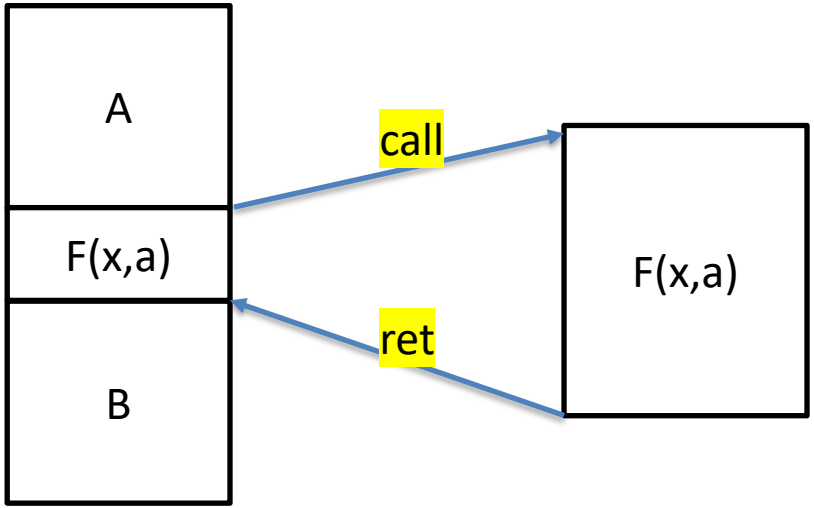
DEFINING A NEW INSTRUCTION

Let's create a new instruction that will both save the location to return and jump to the beginning of the function. We'll name this our **call** instruction

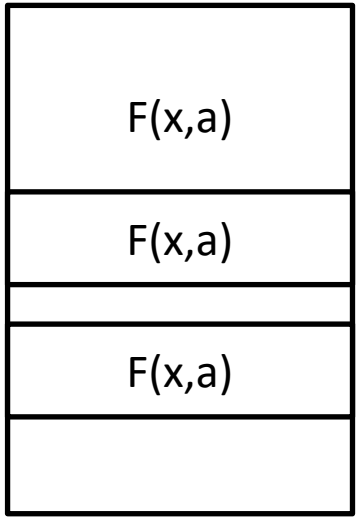
Save pc+2 , set pc = M[pc+1]

Let's also create an instruction that sets the PC back to the saved. We'll name this our return instruction or ret for short

pc = Saved Value

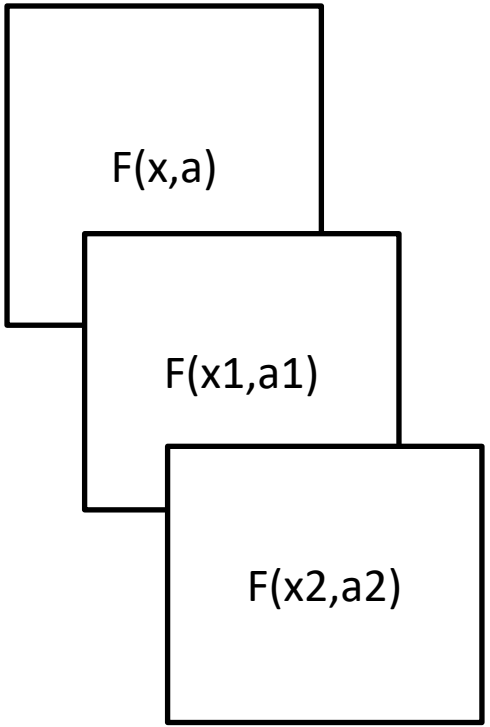


WHAT ABOUT FUNCTIONS



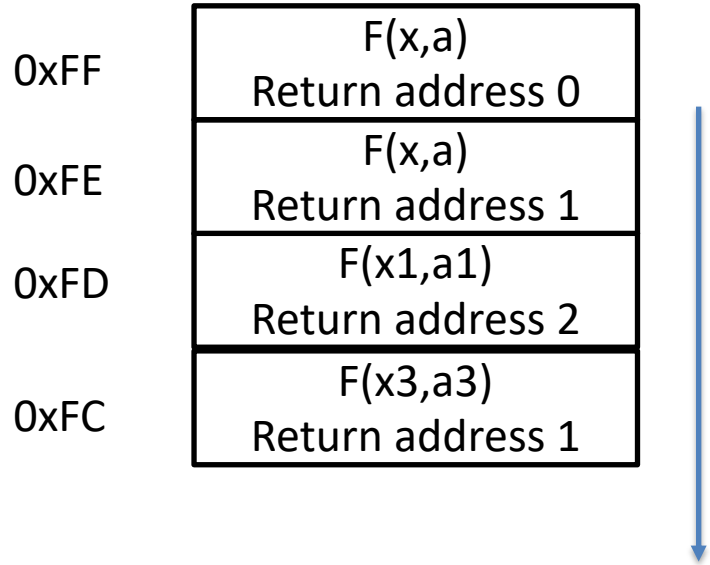
What about recursive functions? Functions that call themselves

Now we need to keep track of both the location return to (multiple function calls and the register state of function before the call)



THE STACK

We are going to a region of memory that will hold the stack of function states and their associated return addresses.

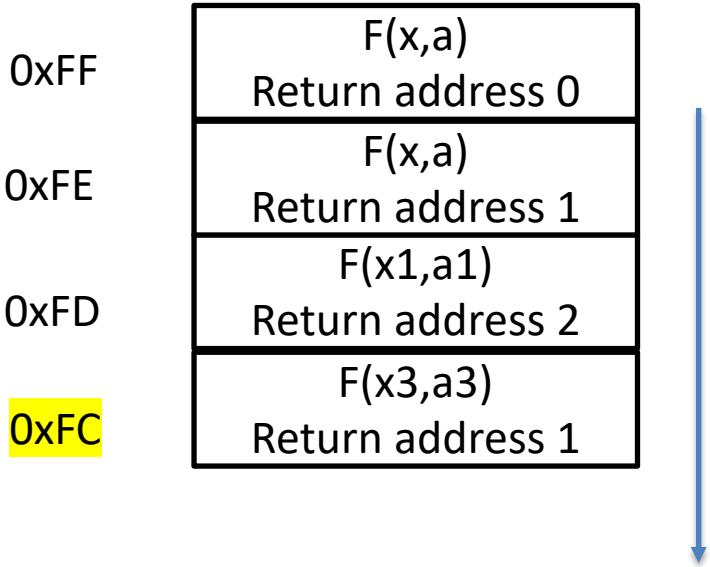


By convention keep adding new things to the stack by growing it to lower addresses

THE STACK

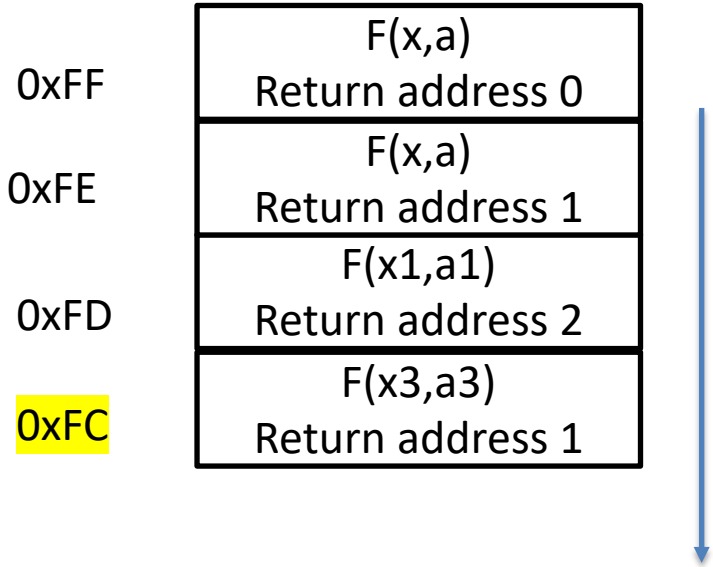
RSP | 0xFC

We also define a new register that holds the location of the TOP of the stack in memory. We'll name this register RSP



PUSH AND POP INSTRUCTIONS

RSP | 0xFC



We'll also create two instructions that will add and remove values from the stack.

The push instruction will decrement the RSP and to the top of the stack

Example push(0x04)

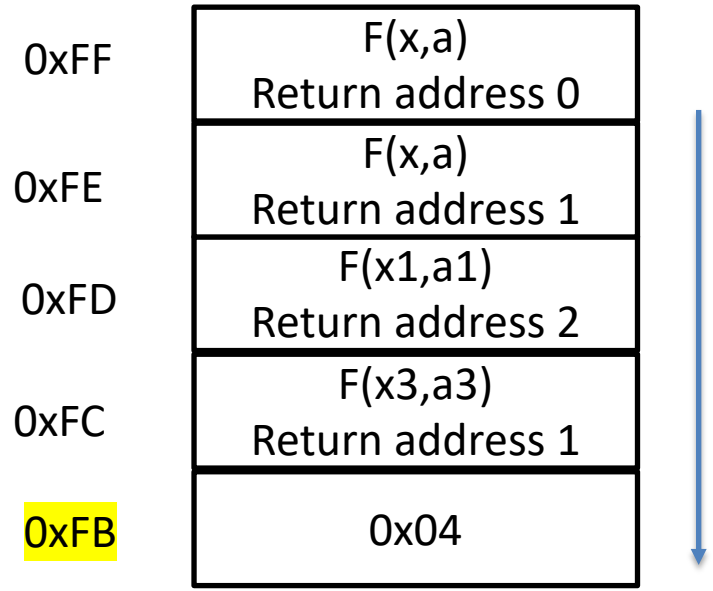
PUSH AND POP INSTRUCTIONS

RSP | 0xFB

We'll also create two instructions that will add and remove values from the stack.

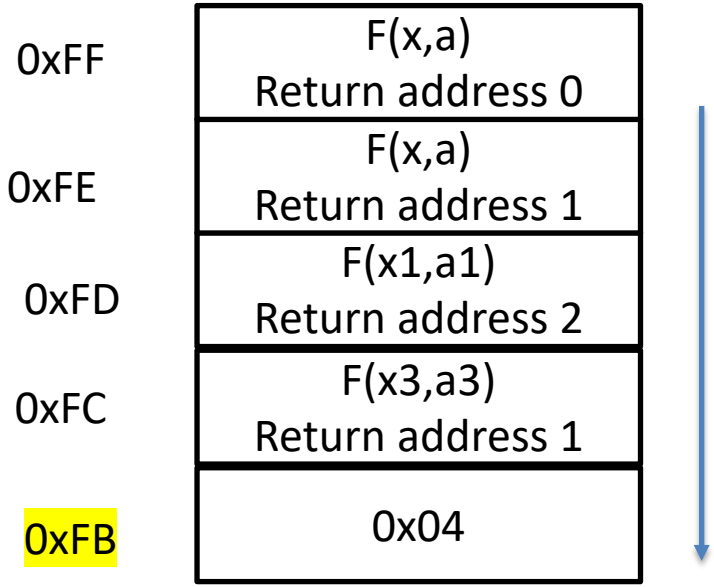
The push instruction will decrement the RSP and to the top of the stack

Example push(0x04)



PUSH AND POP INSTRUCTIONS

RSP | 0xFB



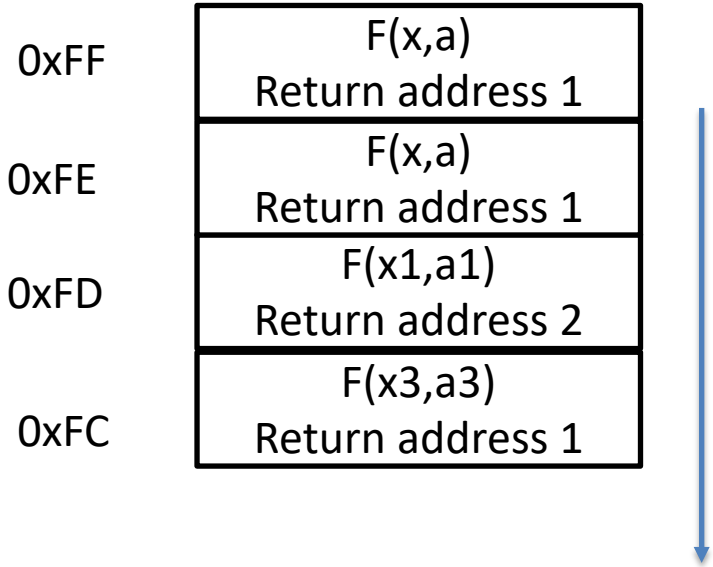
We'll also create two instructions that will add and remove values from the stack.

While the pop instruction increments RSP and returns the value at the top of the stack

Example x = pop()

PUSH AND POP INSTRUCTIONS

RSP | 0xFC



We'll also create two instructions that will add and remove values from the stack.

While the pop instruction returns the value at the top of the stack and **then** increments RSP

Example x = pop() returns 0x04

WHAT ABOUT THE FUNCTION PARAMETERS

We need to define a calling convention. The rules that we'll follow when we call a function.

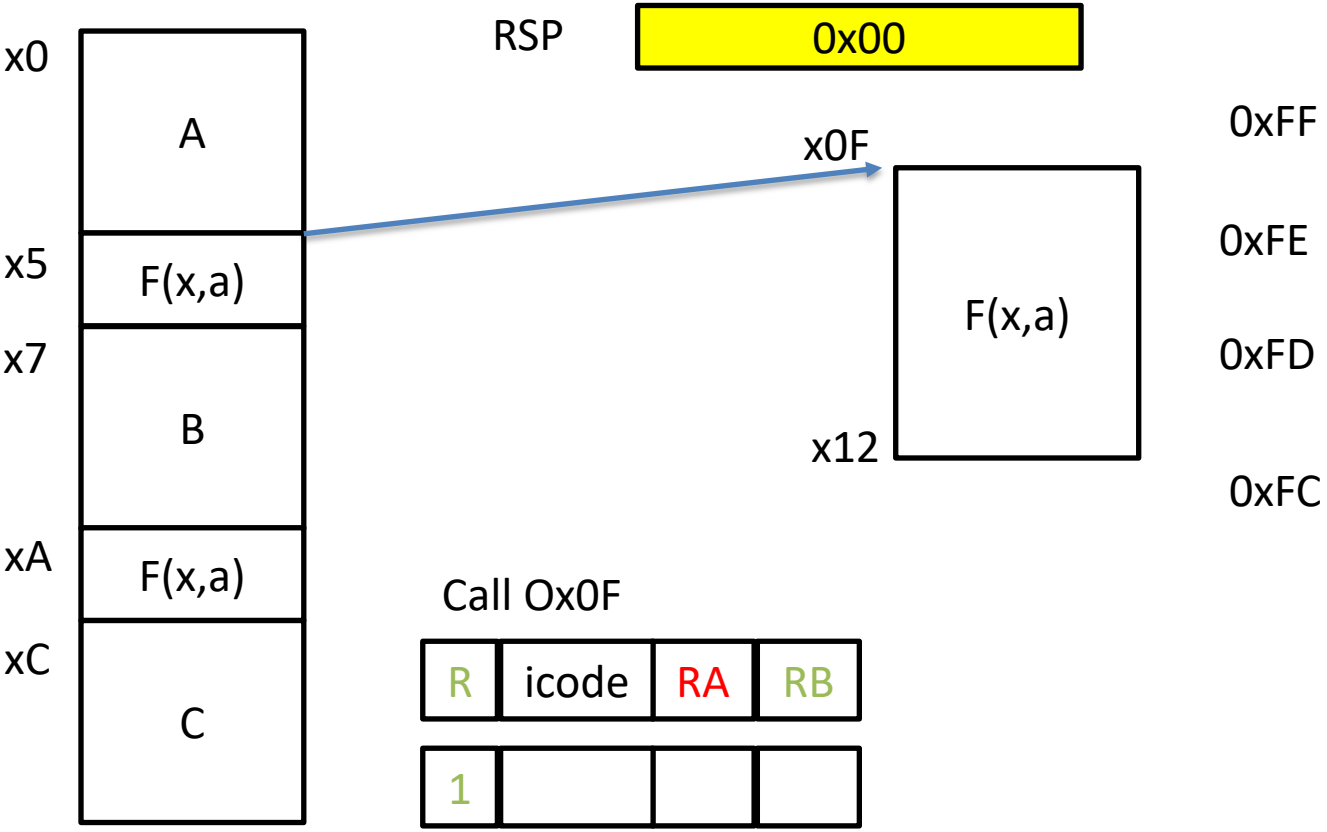
1. For our simple processor functions are limited to 2 parameters.
2. The first parameter will be stored in R2
3. The second parameter will be stored in R3
4. The return value of the function will be stored in R0
5. If the function uses any other registers save them before modifying them and restore them before returning.

```
input = 0xFF
shiftAmount = 0x02
output = left_shift(input, shiftAmount)
```

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

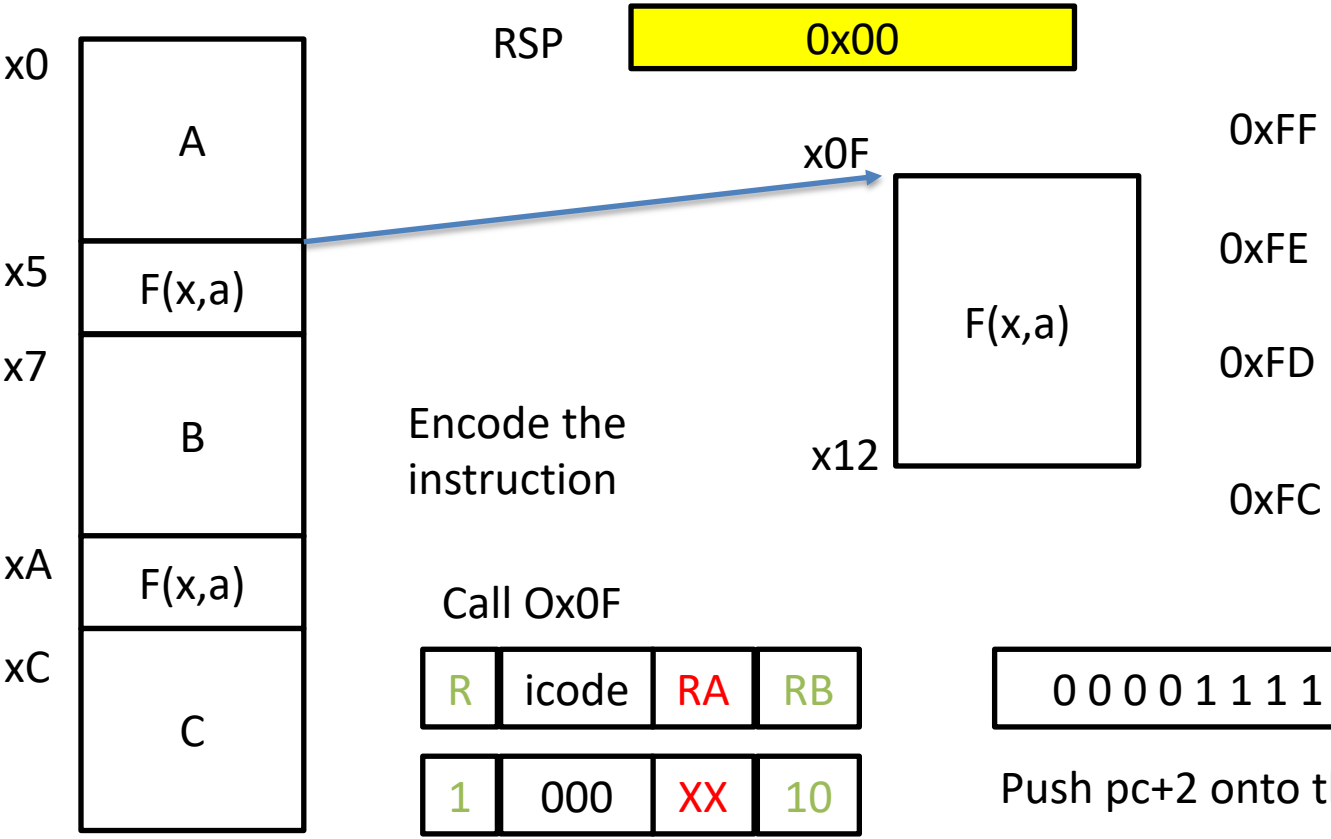
ISA EXTENDED BY SETTING R BIT TO 1

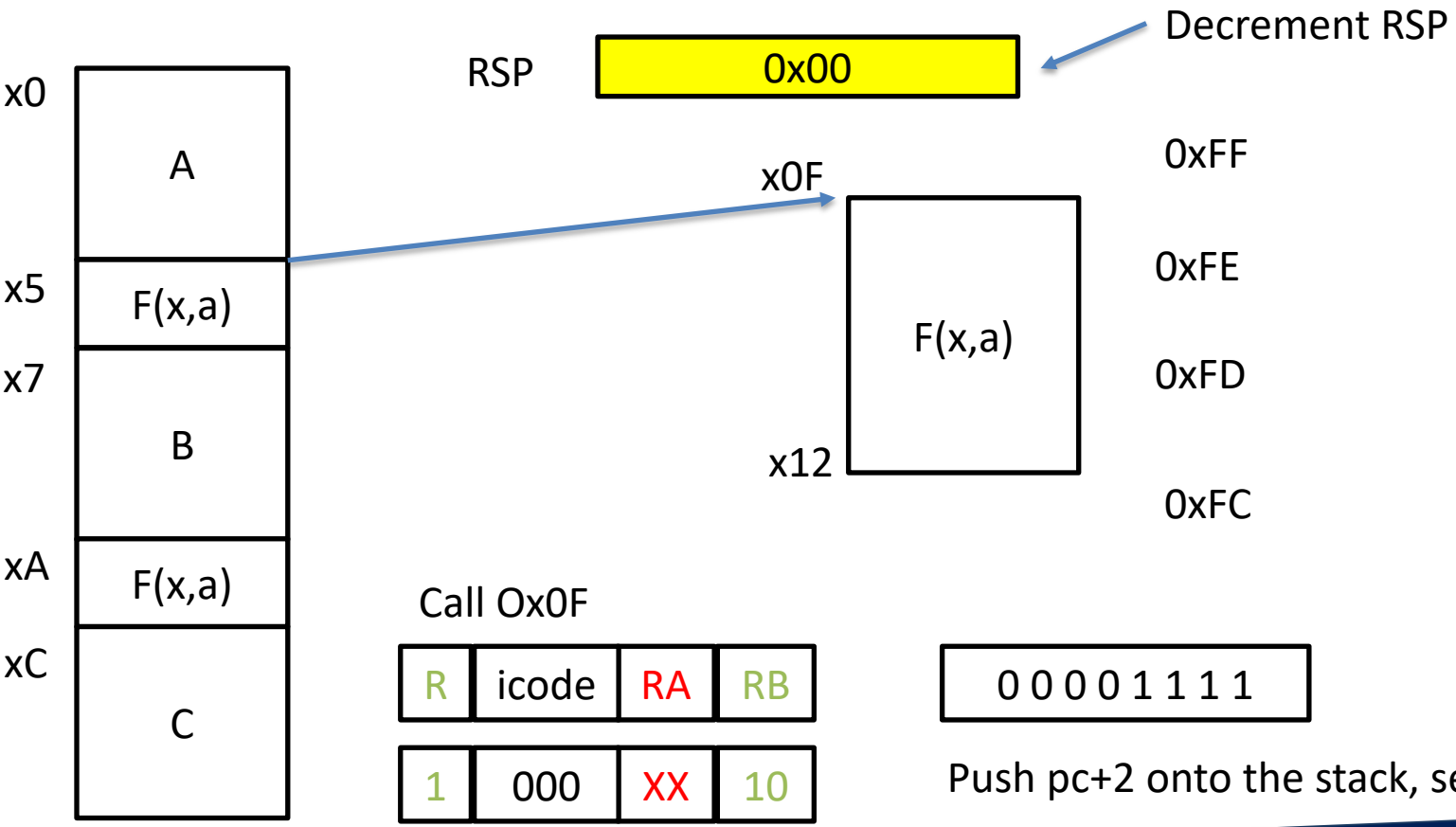
icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.



ISA EXTENDED BY SETTING R BIT TO 1

icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.



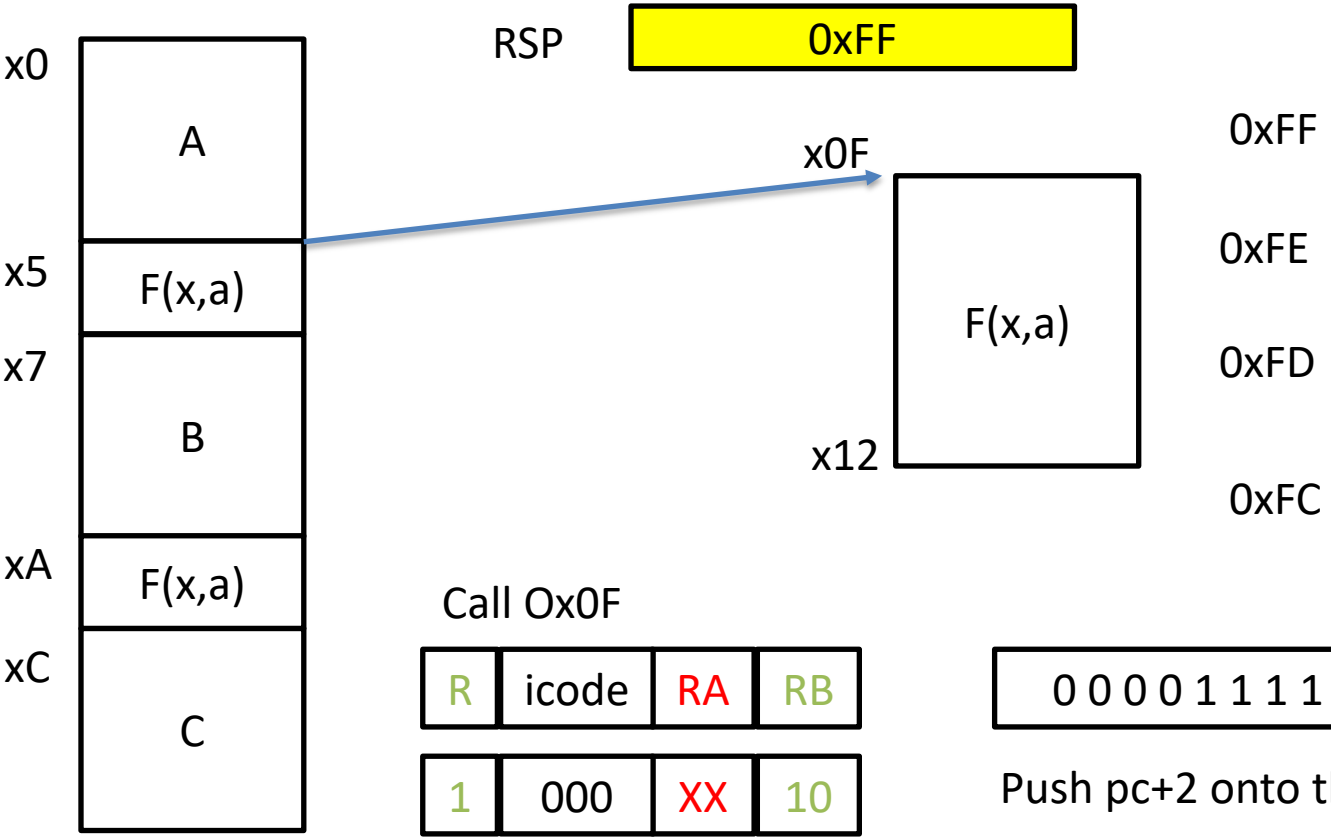


Call Ox0F

R	icode	RA	RB
1	000	XX	10

0 0 0 0 1 1 1 1

Push pc+2 onto the stack, set pc = M[pc+1]

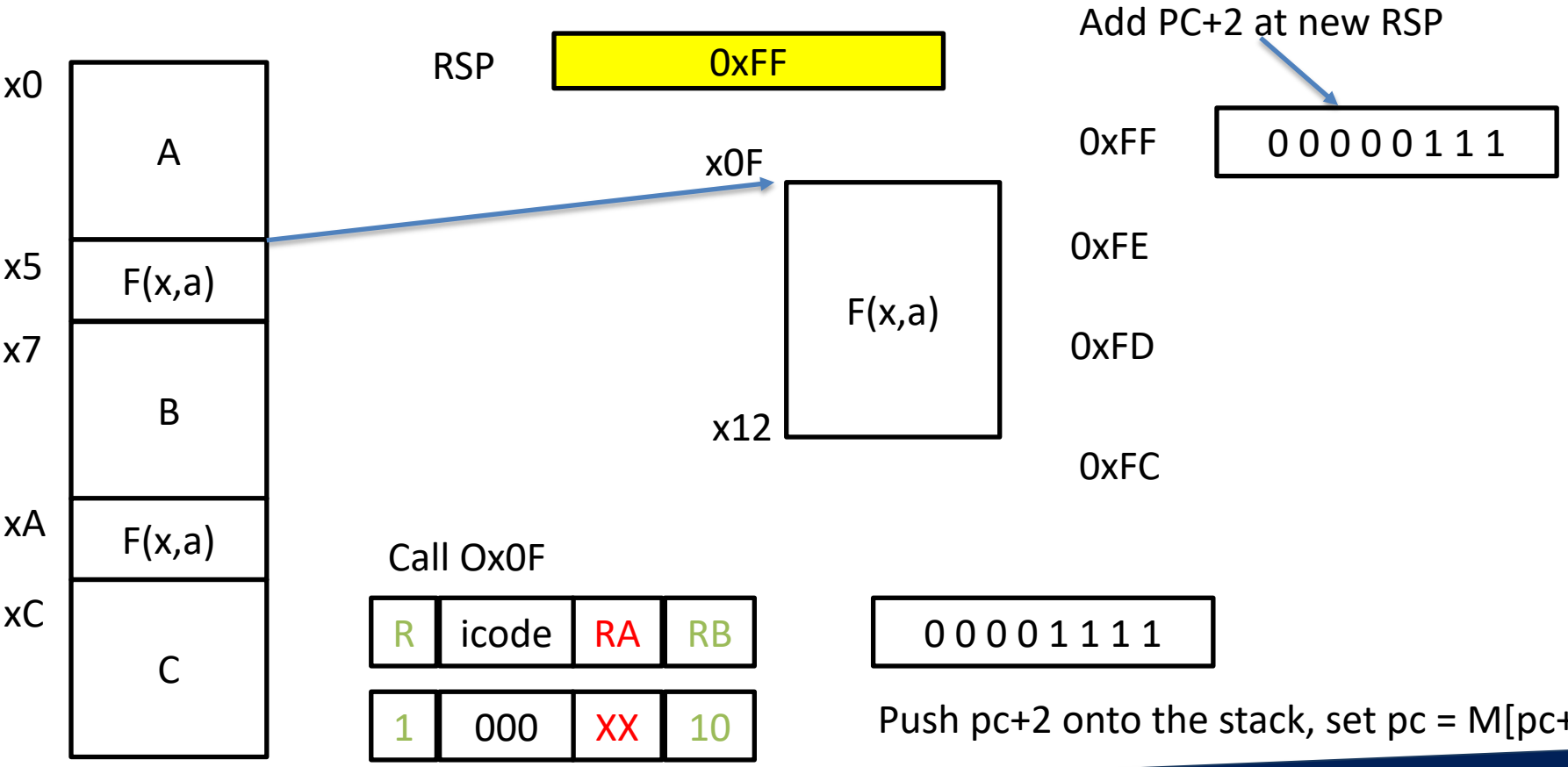


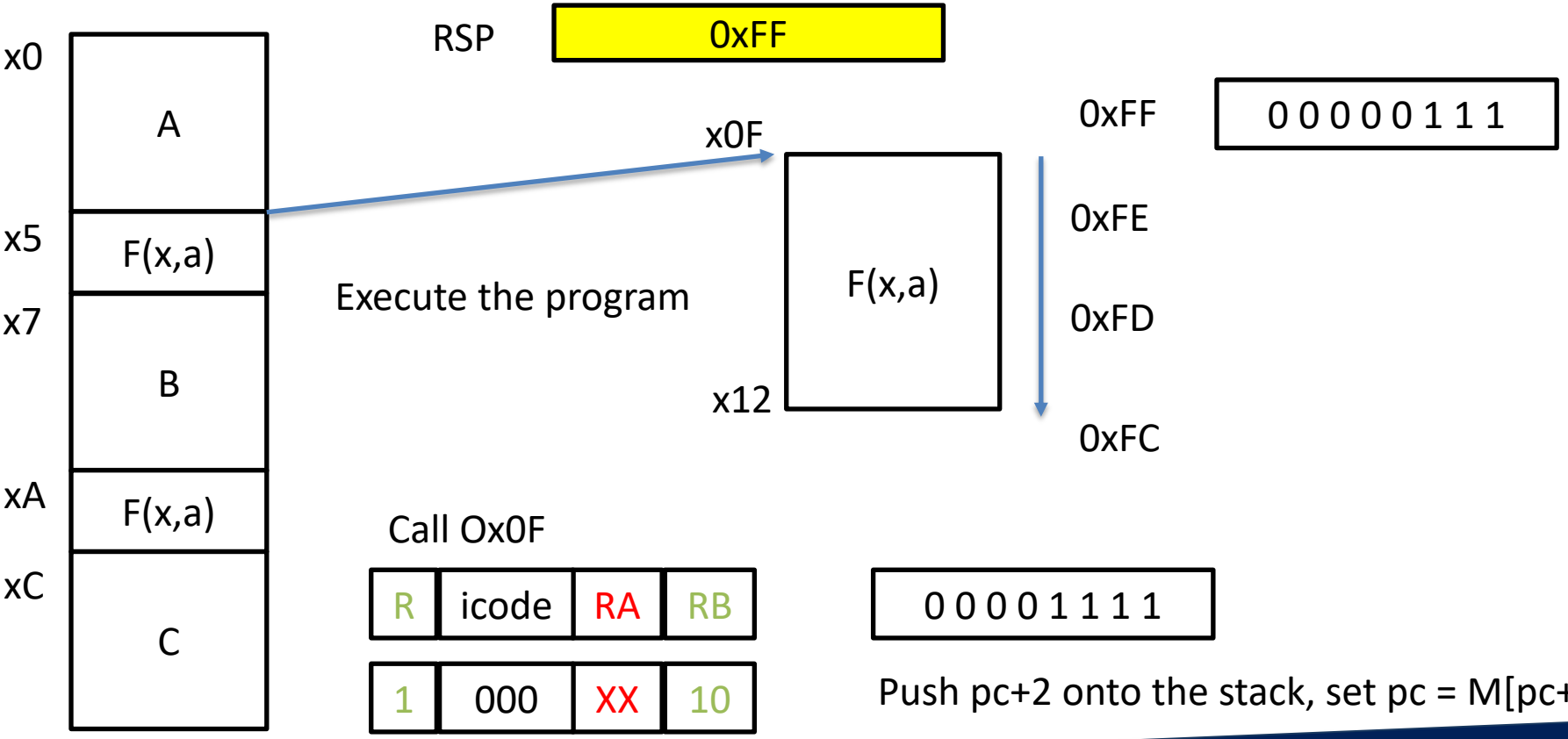
Call Ox0F

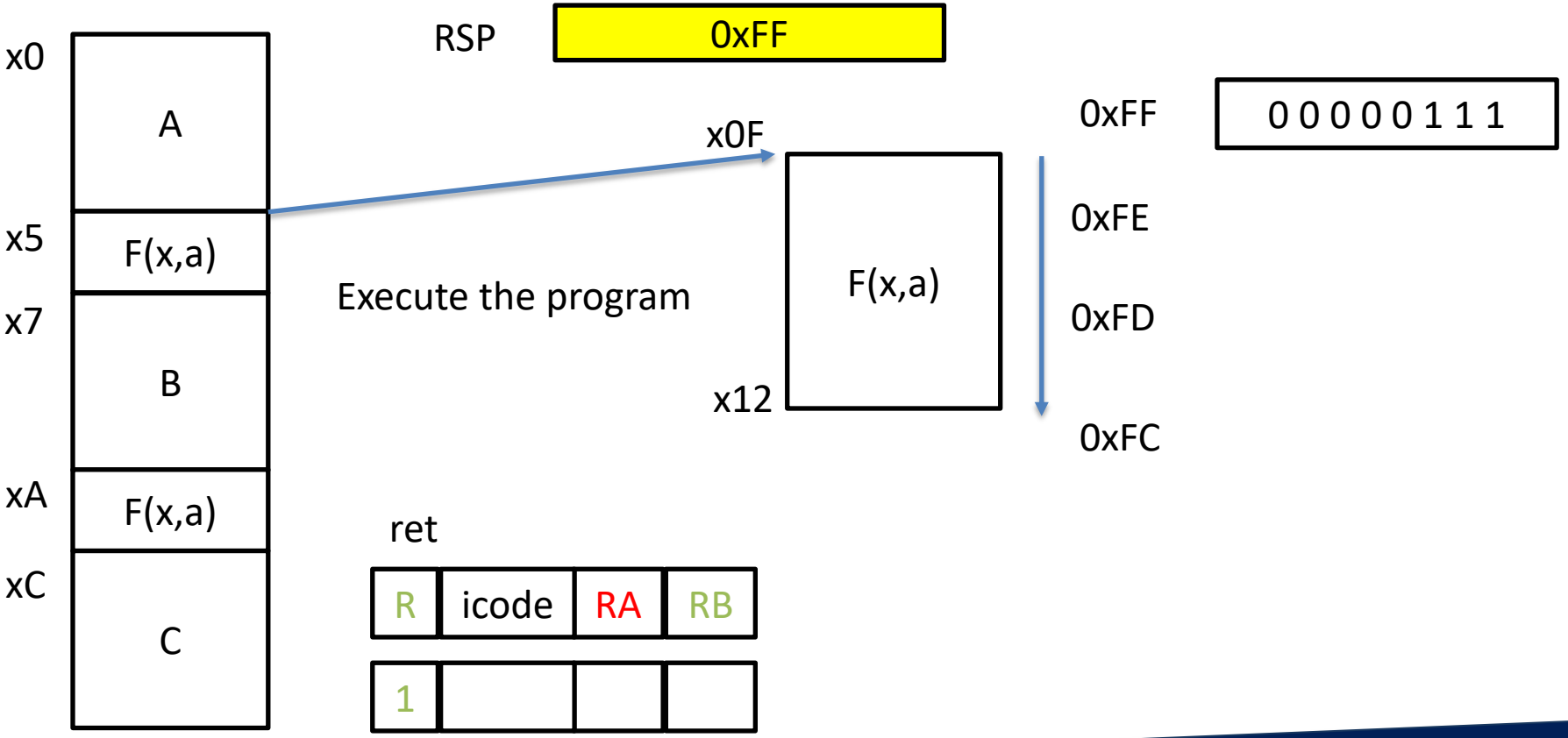
R	icode	RA	RB
1	000	XX	10

0 0 0 0 1 1 1 1

Push pc+2 onto the stack, set pc = M[pc+1]

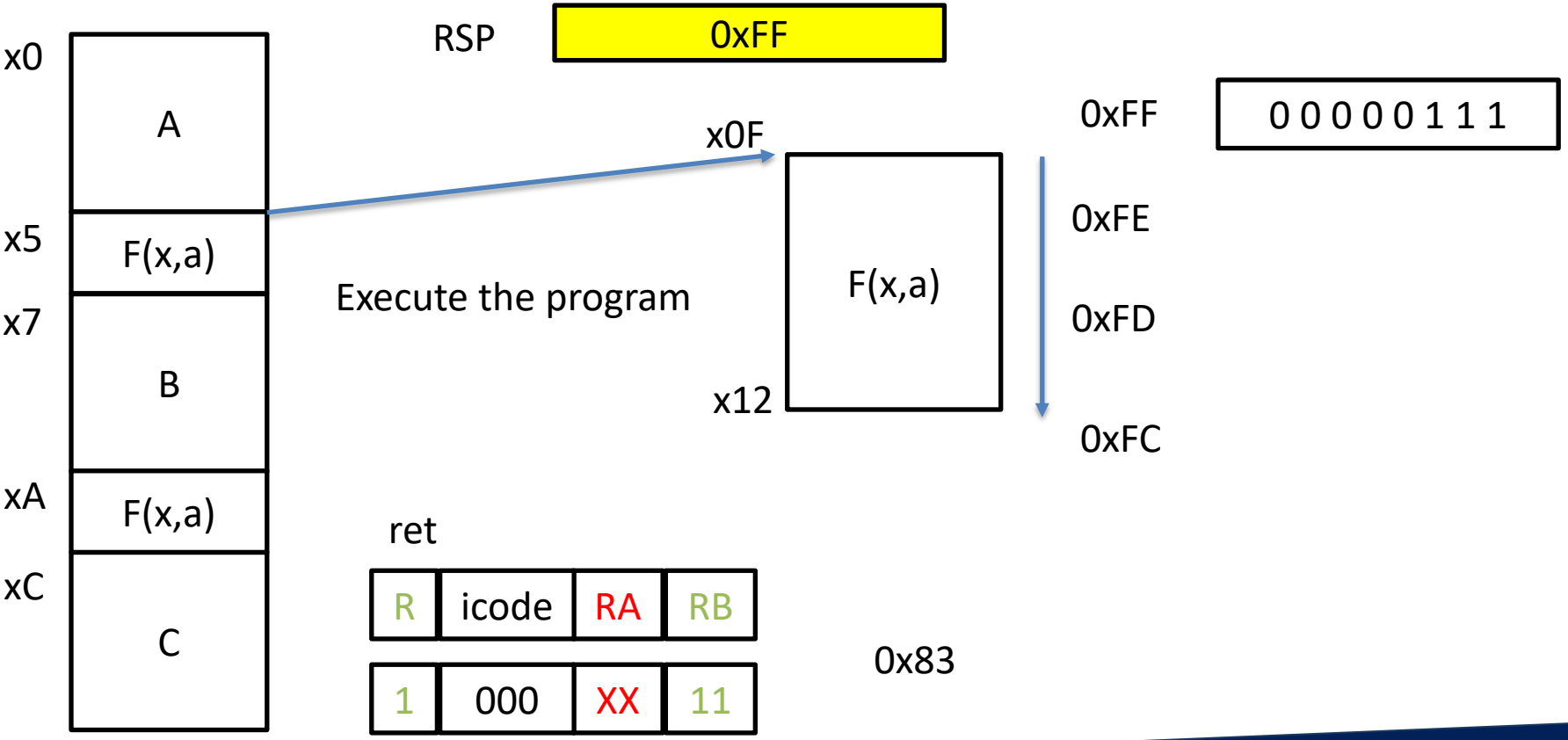


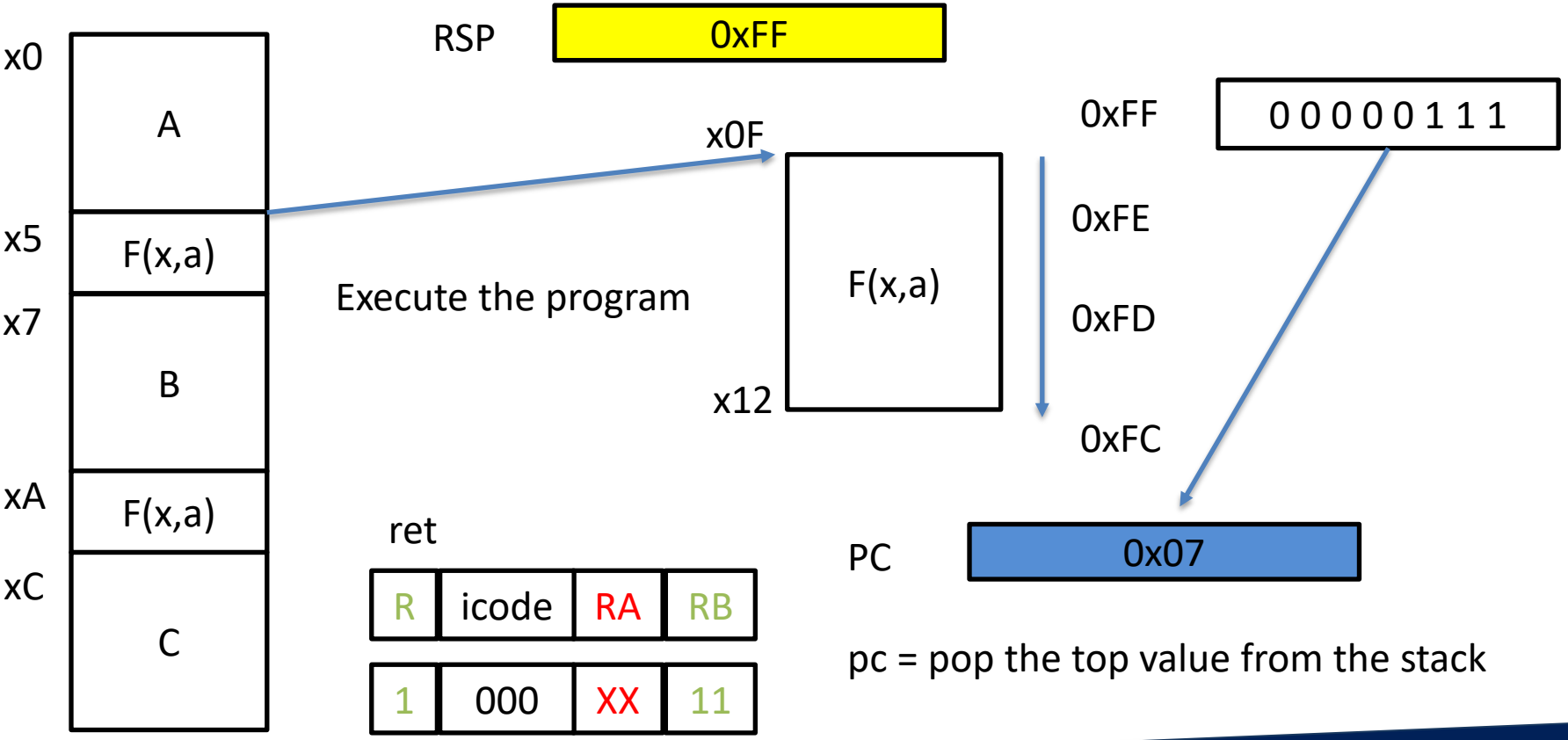


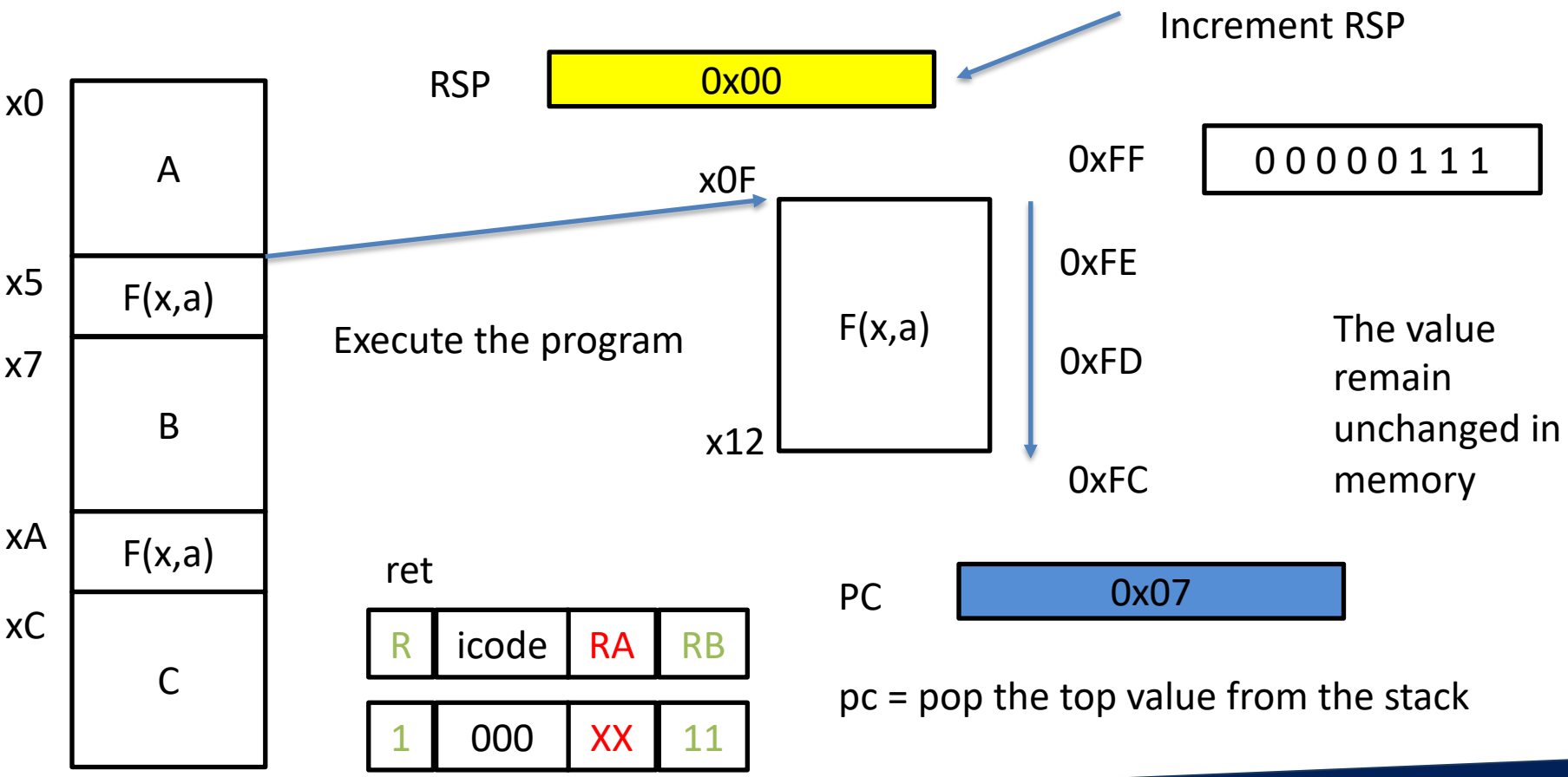


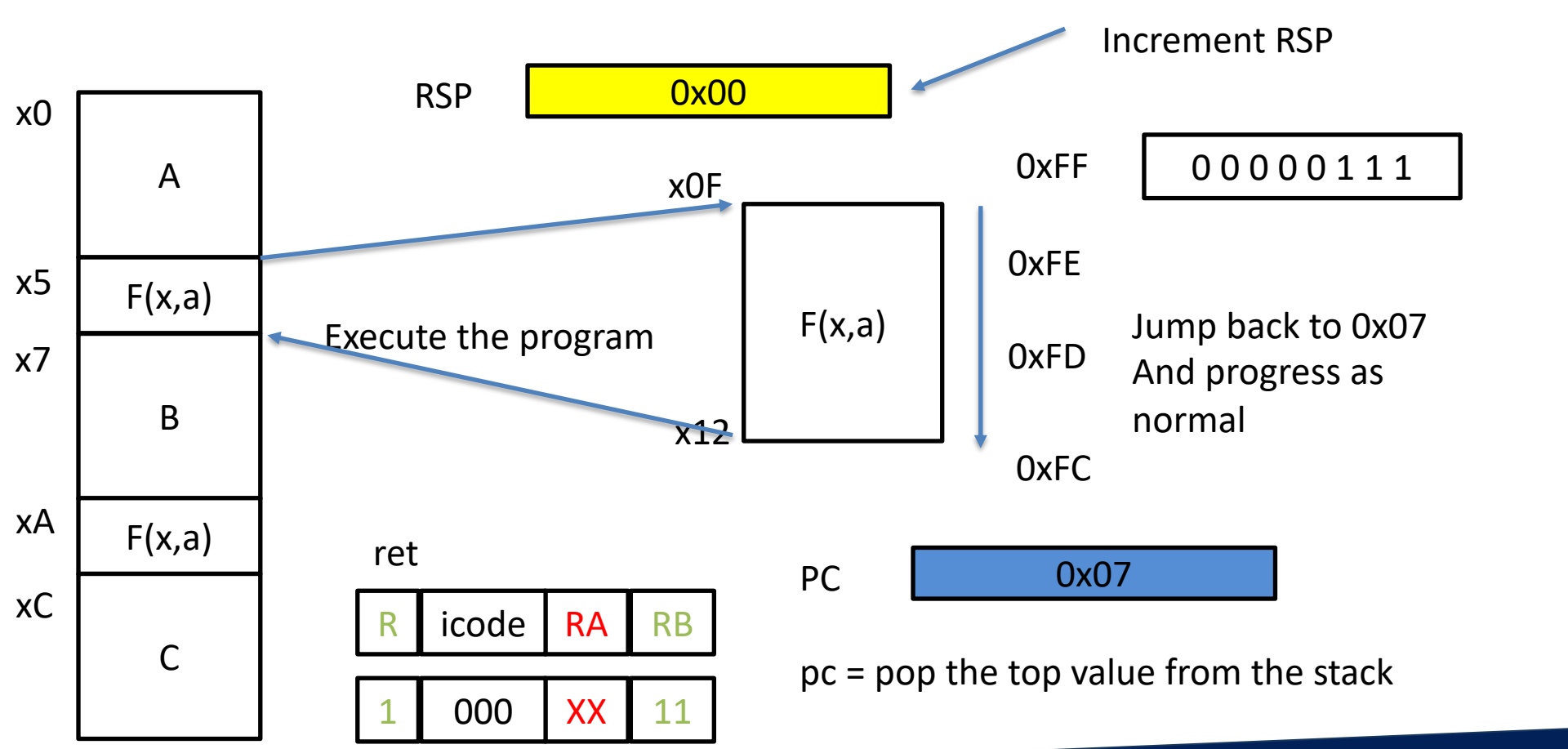
ISA EXTENDED BY SETTING R BIT TO 1

icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.









WHAT ABOUT THE FUNCTION PARAMETERS

We need to define a calling convention. The rules that we'll follow when we call a function.

1. For our simple processor functions are limited to 2 parameters.
2. The first parameter will be stored in R2
3. The second parameter will be stored in R3
4. The return value of the function will be stored in R0
5. If the function uses any other registers save them before modifying them and restore them before returning.

```
input = 0xFF
shiftAmount = 0x02
output = left_shift(input, shiftAmount)
```

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

THOUGHT EXPERIMENTS

Could you implement the left_shift function using our toy ISA?

a = 1

input = -1

shiftAmount = 2

output = left_shift(input, shiftAmount)

a+= output

```
R1  = 1
R2 = 0xFF
R3 = 0x02
call left_shift
//R0 Contains result
R1+=R0
```

Hint: Left shifts by 1 is equivalent to multiplying the number by 2. Let's Implement left shift function

```
Push R1
Push R2
Push R3
R3=-R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

```
a = 1
input = -1
shiftAmount = 2
output = left_shift(input, shiftAmount)
a+= output
```

```
R1  = 1
R2 = 0xFF
R3 = 0x02
call left_shift
//R0 Contains result
R1+=R0
```

Function

```
Push R1
Push R2
Push R3
R3=-R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

But wait when we change R1 in our function. Will take give use the wrong result?

No because we save and Restore R1 and the beginning and end of our function

Registers

R0	X
R1	X
R2	X
R3	X
PC	00
RSP	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	X
PC	00
RSP	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	02
PC	02
RSP	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	02
PC	02
RSP	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	02
PC	04
RSP	00

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	02
PC	10
RSP	FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																06

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

Registers

R0	X
R1	X
R2	FF
R3	02
PC	10
RSP	FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10																
…																
0F																06

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

```
68 FF
6C  02
82 10
R0 //Contains result
```

THOUGHT EXPERIMENTS

Could you implement the left_shift function using our toy ISA?

output = left_shift(input, shiftArmount)

```
R2 = 0xFF
R3 = 0x02
call left_shift
R0 //Contains result
```

Hint: Left shifts by 1 is equivalent to multiplying the number by 2. Let's Implement left shift function

```
Push R1
Push R2
Push R3
R3= -R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

THOUGHT EXPERIMENTS

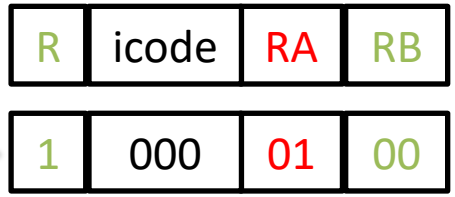
```
Push R1
Push R2
Push R3
R3= -R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.

THOUGHT EXPERIMENTS

```
Push R1
Push R2
Push R3
R3= -R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

```
0x84
0x88
0x8C
0x5D
0x6D 0x01
0x57
1A
0x6D 0x01
0x7D
0x03
R3 = POP
R2 = POP
R1 = POP
RET
```



THOUGHT EXPERIMENTS

```
Push R1
Push R2
Push R3
R3= -R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.

THOUGHT EXPERIMENTS

Push R1	0x84
Push R2	0x88
Push R3	0x8C
R3 =-R3	0x5D
R3+= 1	0x6D 0x01
R1 = PC	0x57
R2+=R2	1A
R3+=1	0x6D 0x01
IF R3 <=0 PC = R1	0x7D
R0 = R2	0x03
R3 = POP	0x8D
R2 = POP	0x89
R1 = POP	0x85
RET	RET

R	icode	RA	RB
1	000	11	01

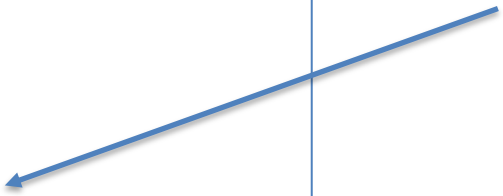
THOUGHT EXPERIMENTS

```
Push R1
Push R2
Push R3
R3= -R3
R3+= 1
R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2
R3 = POP
R2 = POP
R1 = POP
RET
```

icode	b	operation
0		
	0	Decrement rsp and push the contents of rA to the stack
	1	Pop the top value from the stack into rA and increment rsp
	2	Push pc+2 onto the stack, set pc = M[pc+1]
	3	pc = pop the top value from the stack If b is not 2, update the pc as normal.

THOUGHT EXPERIMENTS

Push R1	0x84
Push R2	0x88
Push R3	0x8C
R3=-R3	0x5D
R3+= 1	0x6D 0x01
R1 = PC	0x57
R2+=R2	1A
R3+=1	0x6D x01
IF R3 <=0 PC = R1	0x7D
R0 = R2	0x03
R3 = POP	0x8D
R2 = POP	0x89
R1 = POP	0x85
RET	0x83



R	icode	RA	RB
1	000	XX	11

Registers

R0	X
R1	X
R2	FF
R3	02
PC	10
RSP	FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F																06

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Top of the stack. Contains the address to return after the function executes

Registers

R0	X
R1	X
R2	FF
R3	02
PC	10
RSP	FE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F															X	06

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Top of the stack

Registers

R0	X
R1	X
R2	FF
R3	02
PC	11
RSP	FD

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F														FF	X	06

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Top of the stack

Registers

R0	X
R1	X
R2	FF
R3	02
PC	12
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

Top of the stack

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Store a copy of registers so that we can retore them when we are done. Now we can use the registers

Registers

R0	X
R1	X
R2	FF
R3	-02
PC	13
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Registers

R0	X
R1	X
R2	FF
R3	-01
PC	14
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

Push R1
Push R2
Push R3
R3=-R3
R3+= 1

Registers

R0	X
R1	16
R2	FF
R3	-01
PC	16
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FE
R3	-01
PC	17
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FE
R3	00
PC	18
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FE
R3	00
PC	16
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Take the loop. Jump back to here

Registers

R0	X
R1	16
R2	FE
R3	00
PC	16
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FD
R3	00
PC	17
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FD
R3	01
PC	18
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Registers

R0	X
R1	16
R2	FD
R3	01
PC	1A
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Loop not taken this time

Registers

R0	X
R1	16
R2	FD
R3	01
PC	1A
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Loop not taken this time

Registers

R0	FD
R1	16
R2	FD
R3	01
PC	1B
RSP	FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R1 = PC
R2+=R2
R3+=1
IF R3 <=0 PC = R1
R0 = R2

Loop not taken this time
Return value is stored in R0

Registers

R0	FD
R1	16
R2	FD
R3	02
PC	1C
RSP	FD

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R3 = POP
R2 = POP
R1 = POP
RET

Restore registers except R0 back original state

Registers

R0	FD
R1	16
R2	FD
R3	02
PC	1C
RSP	FD

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

Pop then increment

R3 = POP
R2 = POP
R1 = POP
RET

New top of the stack

Registers

R0	FD
R1	16
R2	FF
R3	02
PC	1C
RSP	FE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

Pop then increment

R3 = POP
R2 = POP
R1 = POP
RET

Registers

R0	FD
R1	X
R2	FF
R3	02
PC	1C
RSP	FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R3 = POP
R2 = POP
R1 = POP
RET

R1 is not restored to what it was before the function was called

Registers

R0	FD
R1	X
R2	FF
R3	02
PC	06
RSP	00

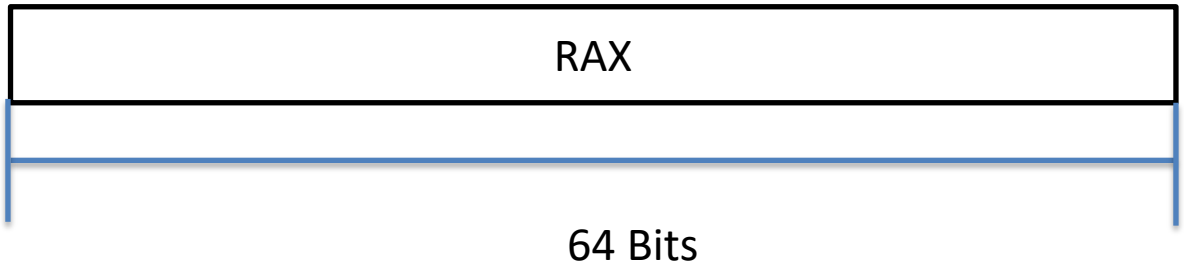
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	68	FF	6C	02	82	10										
10	84	88	8C	5D	6D	01	57	1A	6D	01	7D	03	8D	89	85	83
…																
0F													02	FF	X	06

R3 = POP
R2 = POP
R1 = POP
RET

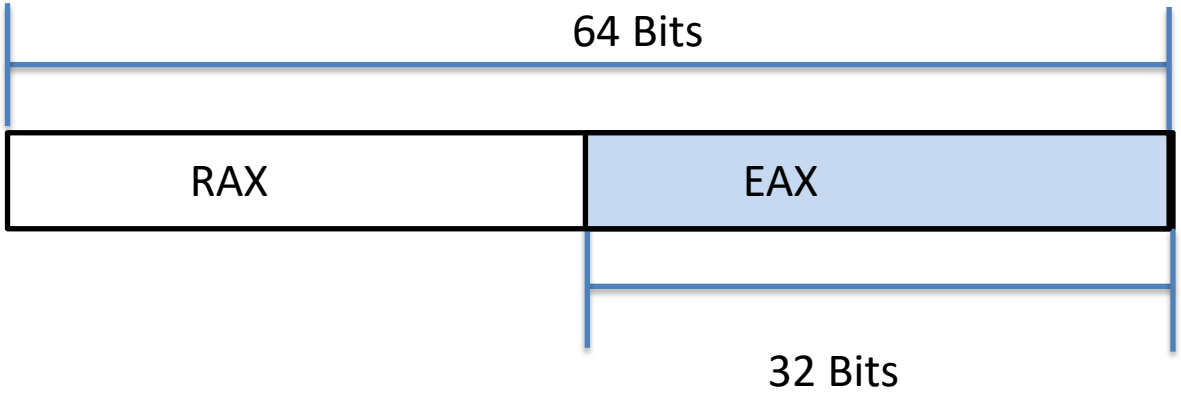
PC gets up dated and we return address after our call instruction

NOW LET'S START TALK ABOUT WRITING ASSEMBLY FOR X86 PROCESSORS

X86 REGISTERS

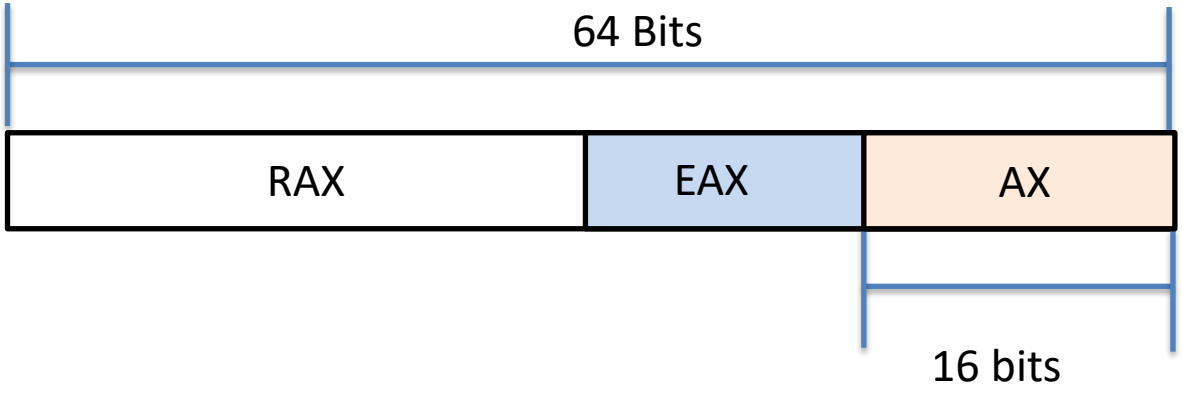


X86 REGISTERS



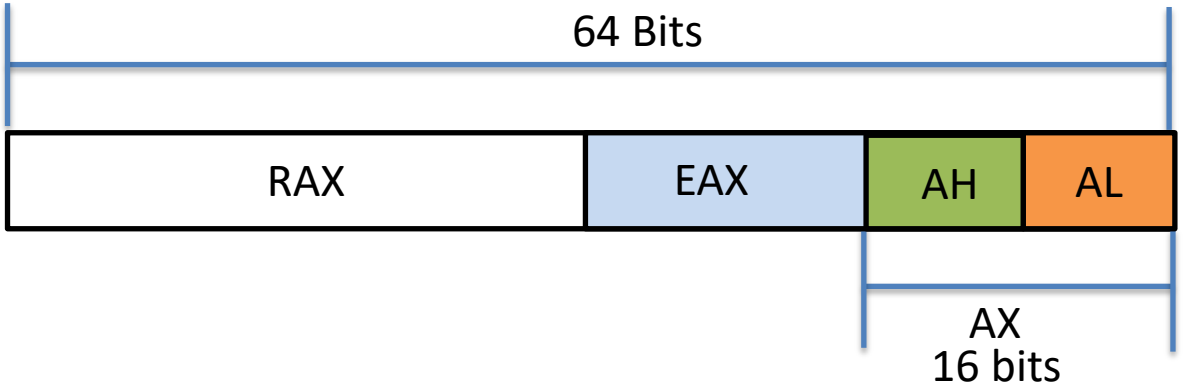
The lowest 32 bits

X86 REGISTERS

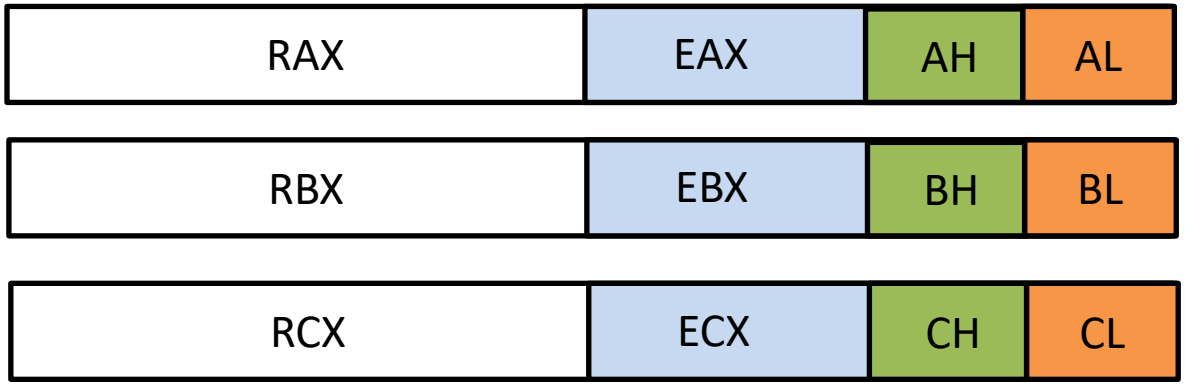


AX can future divided into two registers

X86 REGISTERS



THERE ARE 16 REGISTER



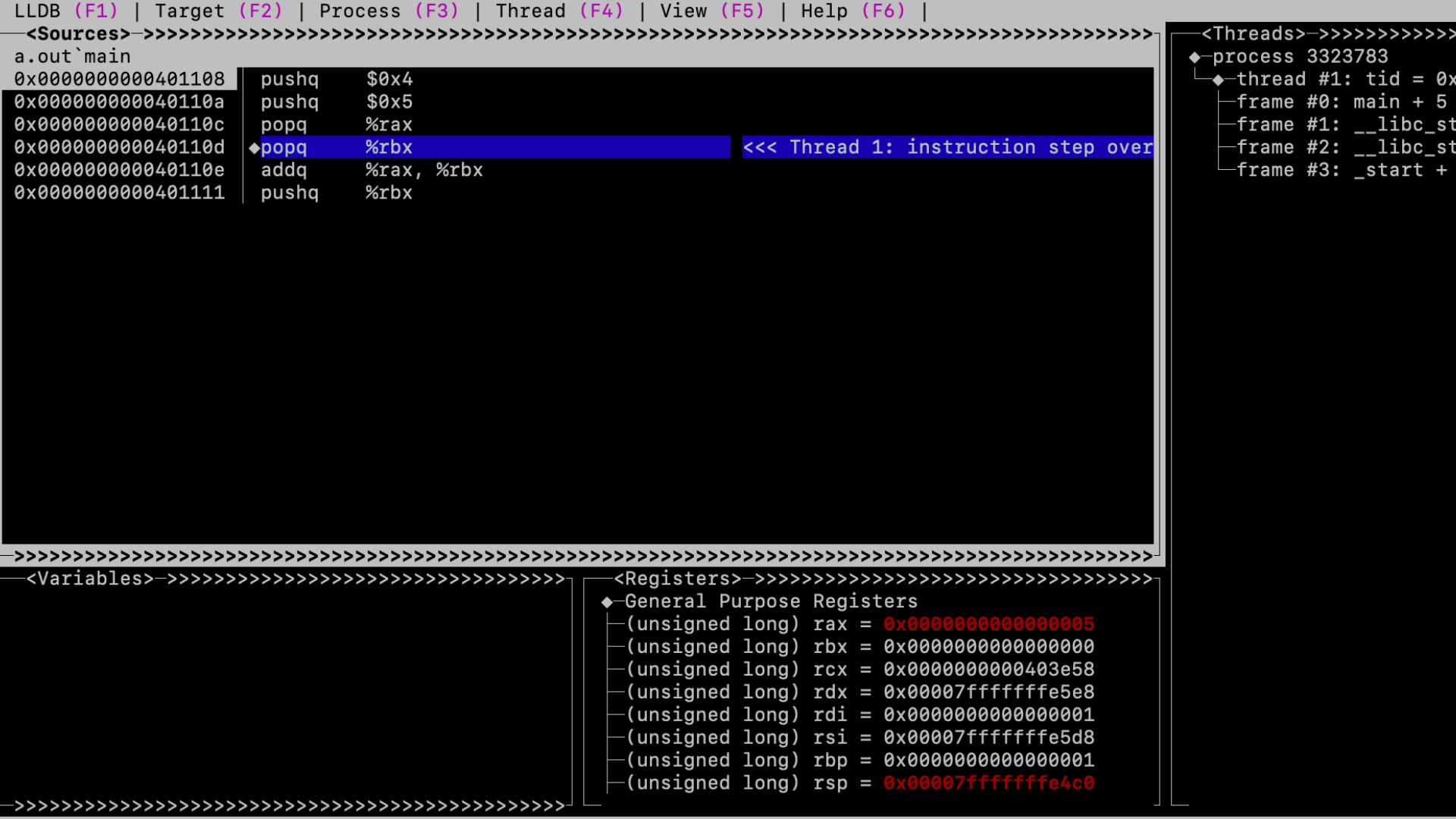
We'll discuss more next time.

A PREVIEW OF WHERE WE ARE GOING

```
0000000000401108 <main>:
  401108:       6a 04                   push   $0x4
  40110a:       6a 05                   push   $0x5
  40110c:       58                      pop    %rax
  40110d:       5b                      pop    %rbx
  40110e:       48 01 c3                add    %rax,%rbx
  401111:       53                      push   %rbx

Disassembly of section .fini:
```

Result of running objdump on Linux executable.


LLDB (F1) | Target (F2) | Process (F3) | Thread (F4) | View (F5) | Help (F6) |
<Sources>
a.out`main
0x0000000000401108 pushq $0x4
0x000000000040110a pushq $0x5
0x000000000040110c popq %rax
0x000000000040110d popq %rbx <<< Thread 1: instruction step over
0x000000000040110e addq %rax, %rbx
0x0000000000401111 pushq %rbx
<Threads>
process 3323783
thread #1: tid = 0x
frame #0: main + 5
frame #1: __libc_st
frame #2: __libc_st
frame #3: _start +
<Variables>
<Registers>
General Purpose Registers
(unsigned long) rax = 0x0000000000000005
(unsigned long) rbx = 0x0000000000000000
(unsigned long) rcx = 0x0000000000403e58
(unsigned long) rdx = 0x00007fffffffe5e8
(unsigned long) rdi = 0x0000000000000001
(unsigned long) rsi = 0x00007fffffffe5d8
(unsigned long) rbp = 0x0000000000000001
(unsigned long) rsp = 0x00007fffffffe4c0