Computer Systems
Subroutines and Stacks

Lecture Objectives

To introduce the fundamentals concepts of subroutines and how they are implemented using stacks.

Lecture Outline

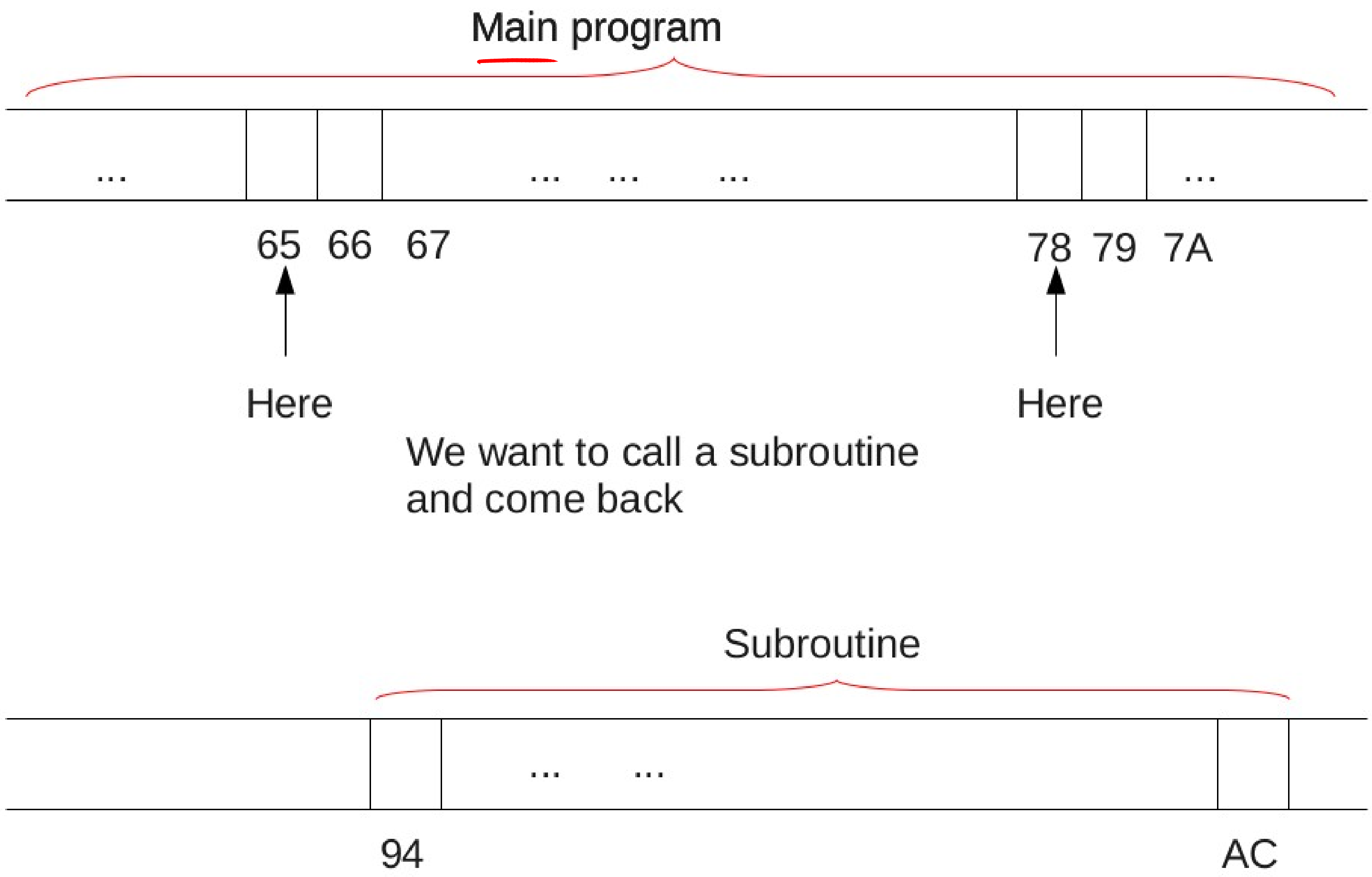
- How a Subroutine Works?
  - Call / Return Instructions
- Introduction to Stacks
  - Saving Registers
  - Stacks for Calculation
- Reverse Polish Notation
- Bitwise Boolean Operations
- Conditional & Unconditional Jumps
- Summary

Subroutines (Methods) – Example

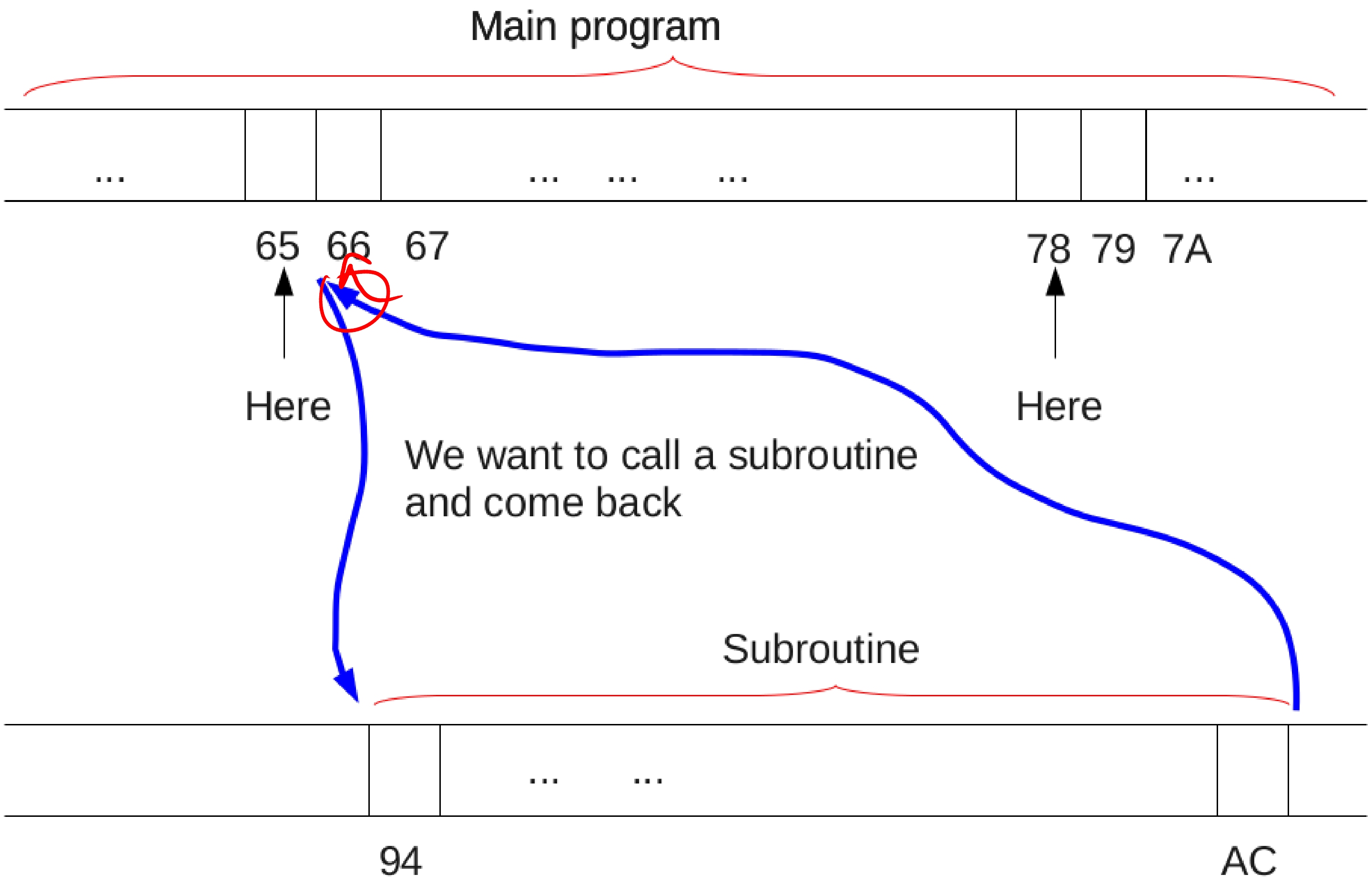
```
class Main {
    public static void main(String[] args) {
        ... .. ...
        myFunction();
        ... .. ...
    }

    private static void myFunction() {
        // function body
        ... .. ...
        ... .. ...
    }
}
```

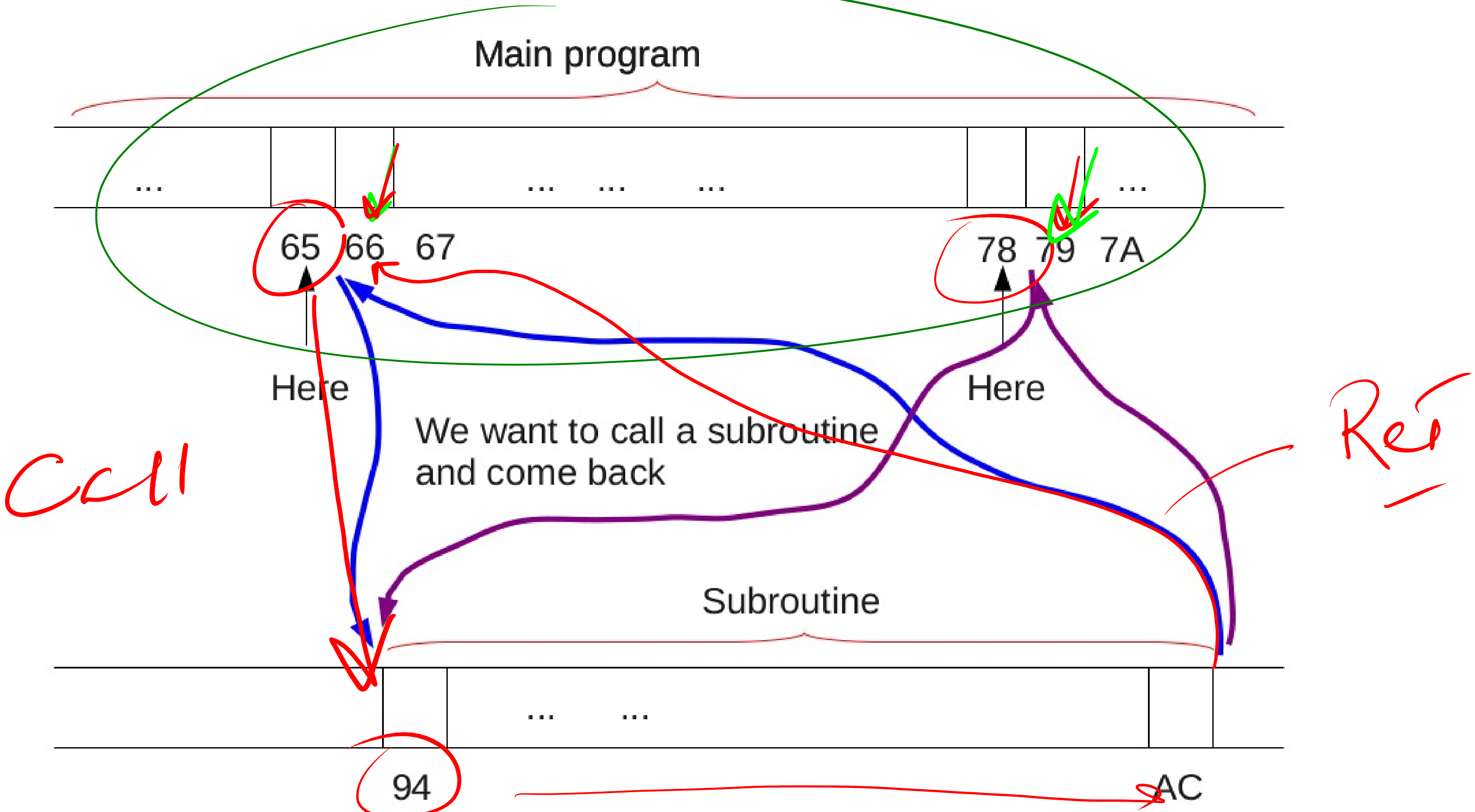
Subroutines (Methods)



Subroutines (Methods)



Subroutines (Methods)



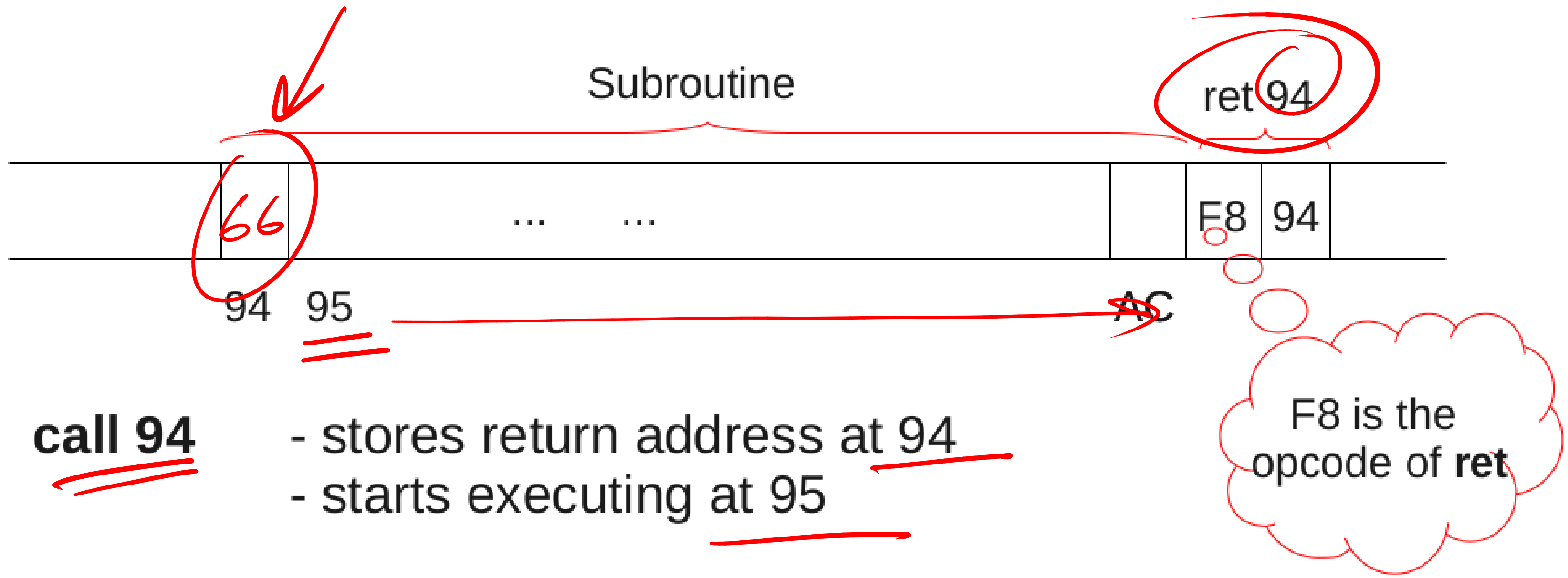
Could jump to 94 for subroutine, but how to know where to jump back afterwards?
Must store the return address somewhere.

Call / Return Instructions

- Two new operations:
  - **call** operand : Like jump, but stores current PC value (the return address) somewhere suitable.
  - **ret** : Read return address from where it was stored, and load it into PC.

Storing the Return Address

- **Idea #1**: In each subroutine, its first byte is used to store the return address

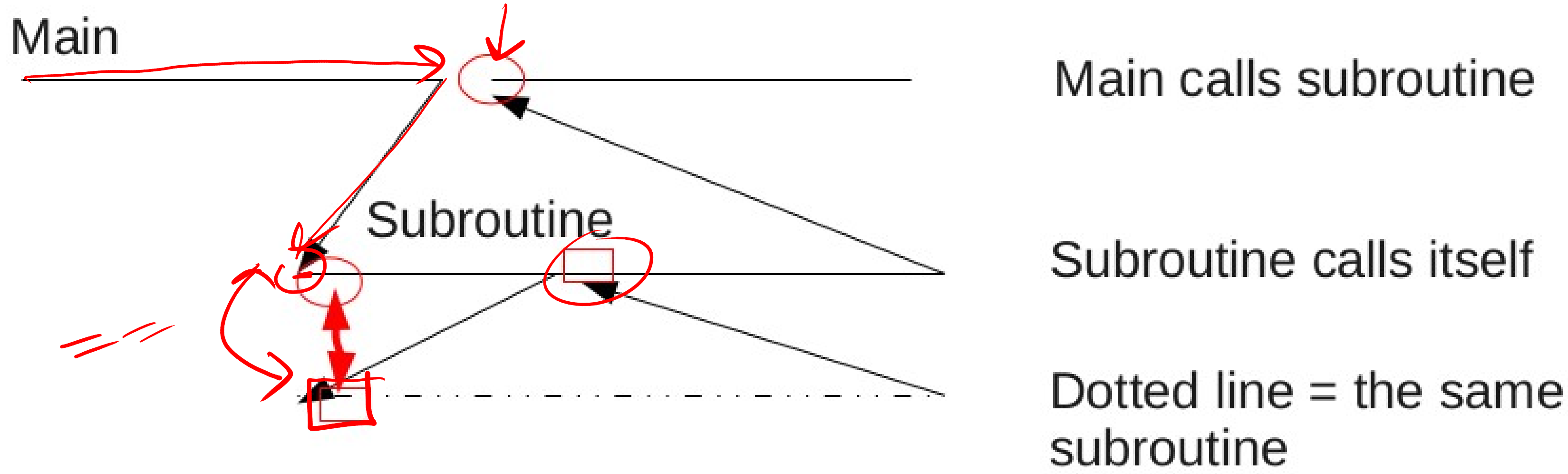


call 94 - stores return address at 94
- starts executing at 95

ret 94 - loads pc with return address at 94

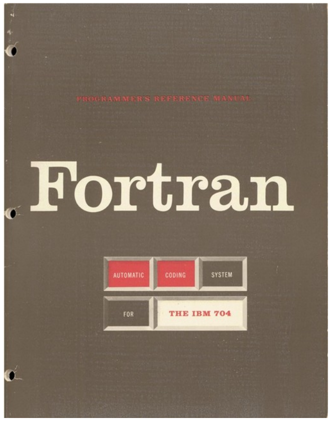
Disadvantage of Idea #1

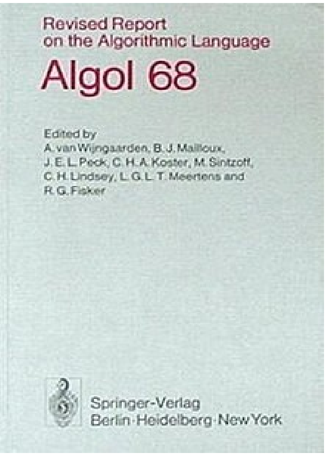
- Can only store one return address
- Consequence: subroutine cannot call itself
  - If it were to call itself it would need to store 2 return addresses



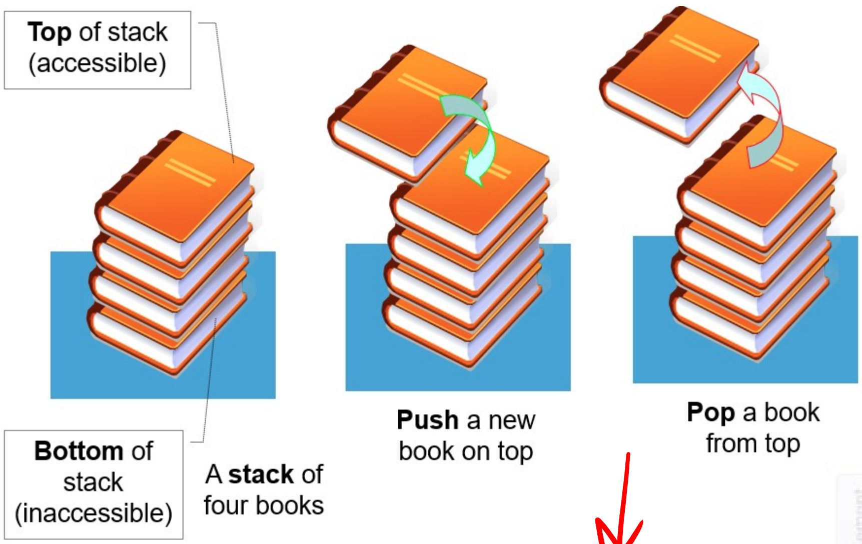
History: 2 Early Programming Languages

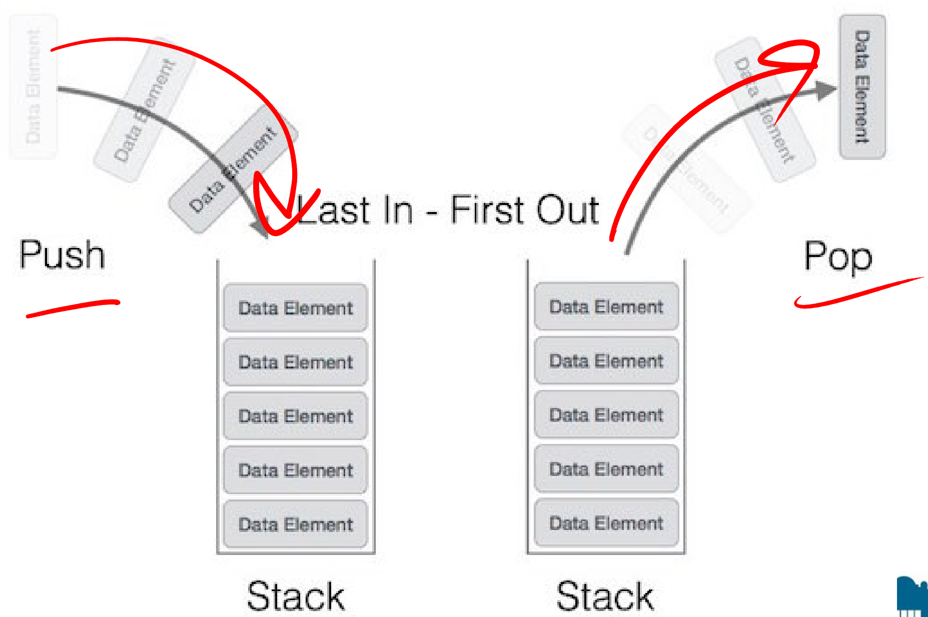
- FORTRAN ("FORmula TRANslation")
  - Used jumps (GOTO), conditional jumps
  - Banned recursion (i.e. for a method to call itself) to allow idea 1.
- Algol ("Algorithmic language")
  - structured programming (if .. then .. else, etc.)
  - Allowed recursion
- Initially, FORTRAN was more successful, but modern languages (e.g. C, Java) took forward the ideas from Algol.





What is a Stack?





How Stacks Work?

- A stack can flexibly store a variable number of bytes
- LIFO = Last In, First Out (aka FILO)

a	b	c	d
42	54	13	27

9
-6
10
2

push a

42
9
-6
10
2

How Stacks Work?

- A stack can flexibly store a variable number of bytes
- LIFO = Last In, First Out (aka FILO)

a	b	c	d
42	54	13	27

42
9
-6
10
2

push c

13
42
9
-6
10
2

b = pop();

How Stacks Work?

- A stack can flexibly store a variable number of bytes
- LIFO = Last In, First Out (aka FILO)

a	b	c	d
42	**13**	13	27

13
42
9
-6
10
2

pop b

42
9
-6
10
2

How Stacks Work?

- A stack can flexibly store a variable number of bytes
- LIFO = Last In, First Out (aka FILO)

a	b	c	d
42	13	13	**42**

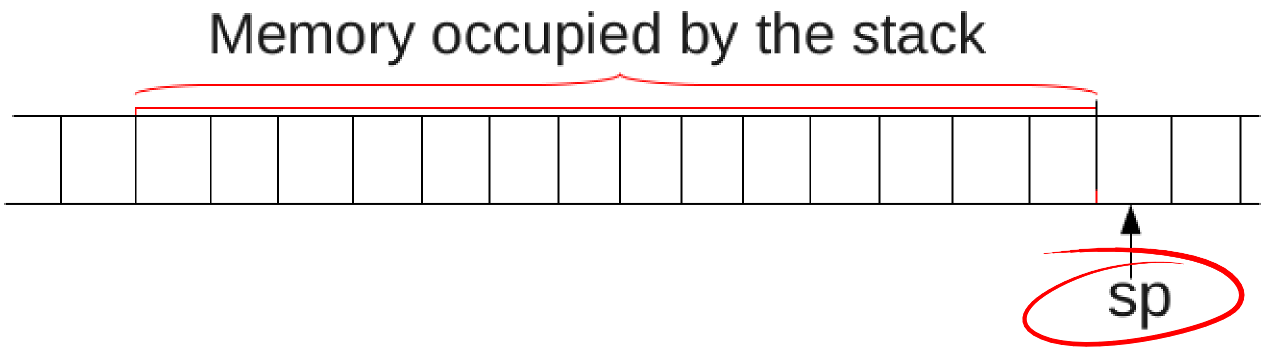
42
9
-6
10
2

pop d

9
-6
10
2

In Memory

- Give CPU another register
  - sp (stack pointer) – shows where top is



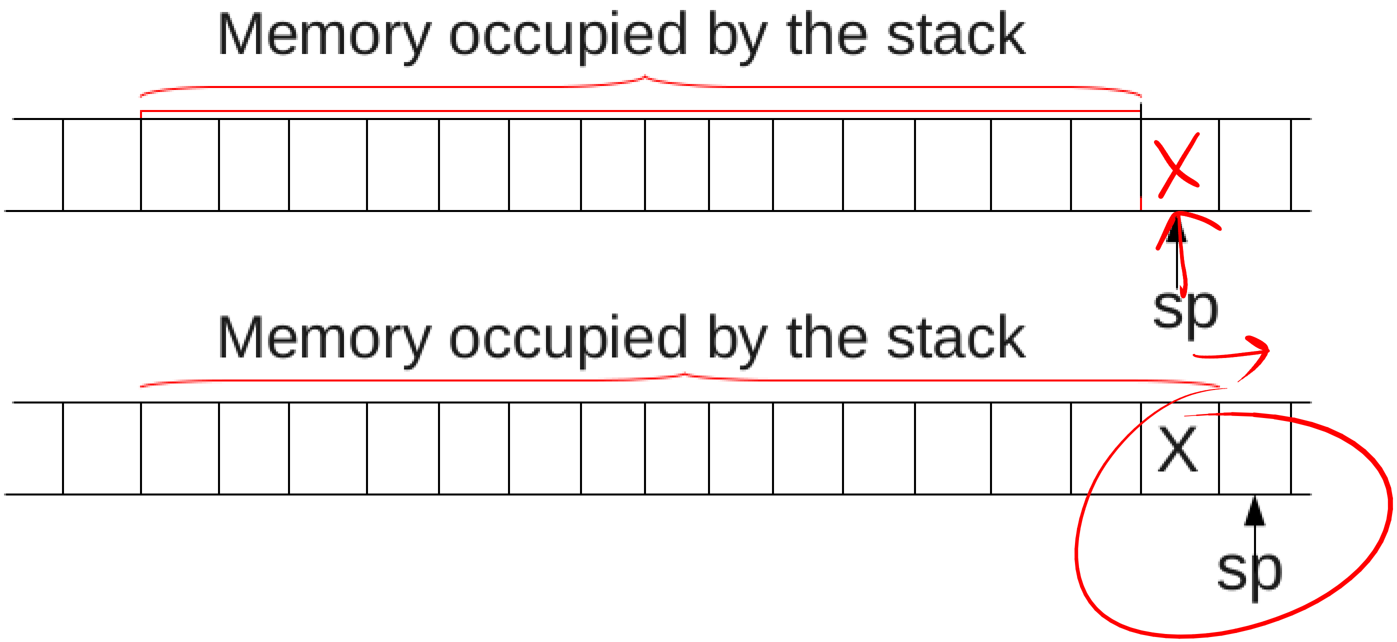
- Don't need to know where bottom is!
  - Provided that we are careful: only pop when you know you've pushed

In Memory

- To push a value X:
  - Write the value to memory at address sp
  - Add 1 to sp
- To pop a value:
  - Subtract 1 from sp
  - Read value from memory at address sp

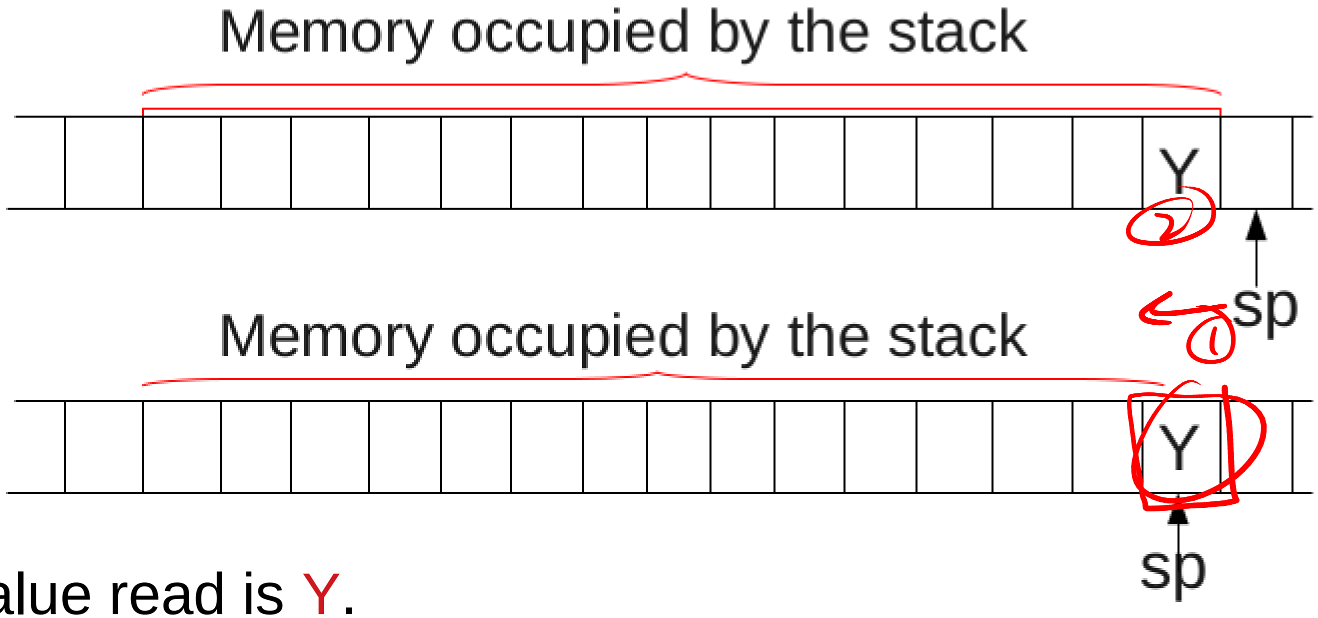
In Memory – Push a Value

- To push a value X:
  - Write the value to memory at address sp
  - Add 1 to sp



In Memory – Pop a Value

- To pop a value:
  - Subtract 1 from sp
  - Read value from memory at address sp



The value read is Y.
Still in memory but will be overwritten by a later push operation.

Storing the Return Address

- **Idea #2:** Store the return address on a stack.

call N	*works as if:*
	push pc // push program counter
	ld pc N // jump to N
ret	*works as if:*
	pop pc // pop return address
	// & jump to it

(none of these are actual operations)

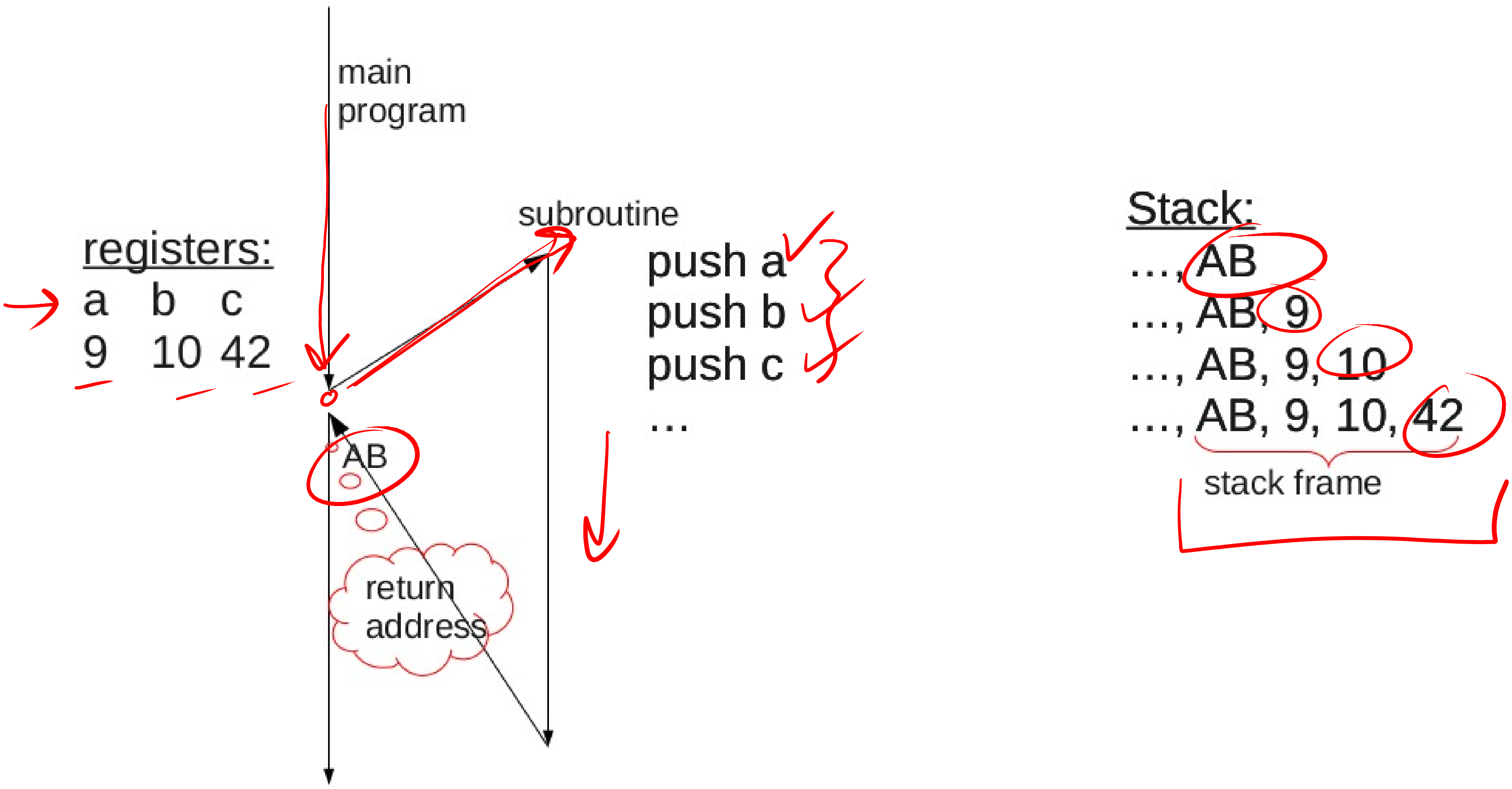
Why should we Save Registers?

- Subroutine may need to use registers for its calculations.
  - But previous register values are needed on return!

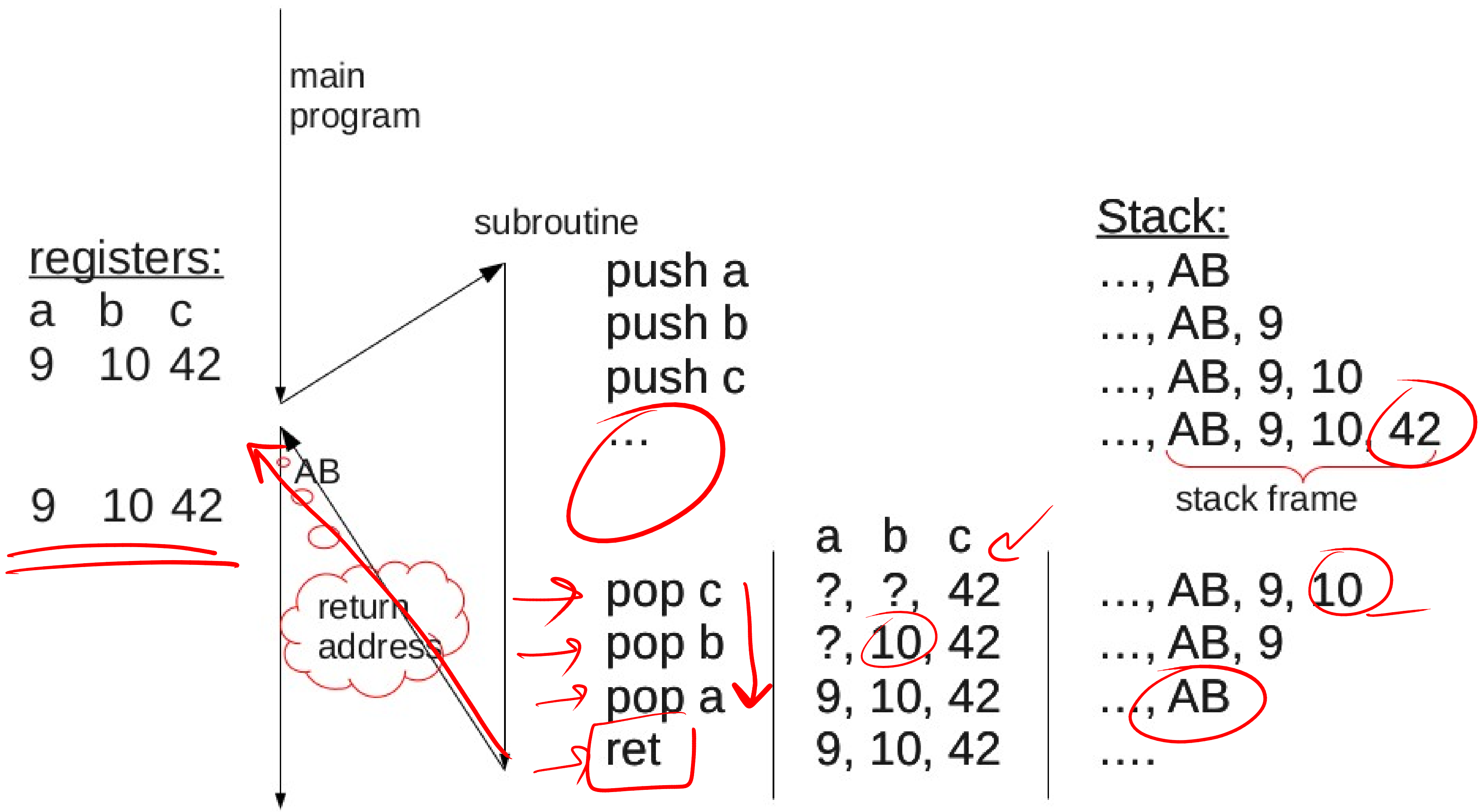
- Common pattern for subroutines:
  - Start by pushing all registers
  - Pop them back before return

- Return address & saved registers = **stack frame**
  Java method-calls develop this idea.

Saving Registers – Stack Frame



Saving Registers – Stack Frame



Stacks for Calculation

- Example:

$$(5+2)*\sqrt{x*x+y*y}+8$$

What order are operations applied in?

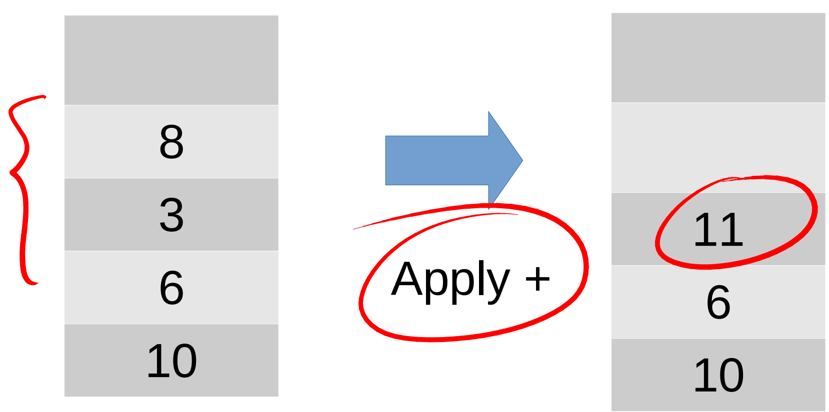
$$(5+2)*\sqrt{x*x+y*y}+8$$

1 6 5 2 4 3 7

$\sqrt{}$ means "square root of" (SQRT)
e.g.
SQRT(x*x + y*y)

Reverse Polish Notation (RPN)

- Order of operations is as written
- No brackets needed
- Powerful use of stack (= operand stack) to store intermediate results
- If its a Number or Variable: push it on the stack
- If its an Operation: apply to the top elements on stack & push result back on the stack



RPN for the Example

$$(5+2)*\sqrt{x*x+y*y}+8$$

1 6 5 2 4 3 7

Reverse Polish: push operands, then operate.

We get:

$$5 \quad 2 \quad + \quad x \quad x \quad * \quad y \quad y \quad * \quad + \quad \sqrt{} \quad * \quad 8 \quad +$$

1 2 3 4 5 6 7

How to use RPN along with a Stack?

- Suppose that x has value 3, and y has value 4
- Now, evaluate the expression. (Top of stack is on right.)

5, 2

Operation	Stack
	empty
5	5
2	5,2
+	7
x	7,3
x	7,3,3
*	7,9
y	7,9,4
y	7,9,4,4
*	7,9,16
+	7,25
SQRT	7,5
*	35
8	35,8
+	43

Notation for Operand Stack

- To show what an operation does to the stack:
- e.g. subtraction

$$\overbrace{\ldots, \text{val1}, \text{val2}}^{\text{stack before}} \rightarrow \overbrace{\ldots, \text{val1-val2}}^{\text{stack after}}$$

pop these two,
do subtraction,
push result

the rest is unchanged

More on Reverse Polish Notation

- Any expression can be converted to Reverse Polish Notation and then its easy to execute with a stack.
- Applications
  - Humans use reverse Polish directly (See Details)
  - e.g. some pocket calculators – HP in 1970s (HP-41C)

  https://www.theregister.co.uk/Print/2014/01/03/ten_classic_calcutors/

- Forth programming language has two stacks:
  - Operand stacks for calculations
  - Return stack for module calls

  More details:
  https://en.wikipedia.org/wiki/Forth_(programming_language))

Applications of Reverse Polish Notation

- Compile to a reverse Polish form that is then executed.
  - e.g. Postscript format, for printable files
    - executed by printers
  - e.g. Java byte code
    - uses operand stacks for calculations

- In Java, each method call has its own operand stack.

Stack instead of Registers (Stack Machines)

- Use 2 stacks
  - **return stack** for subroutine return
  - **operand stack** for Reverse Polish calculations
- Don't need a,b,c registers
  - **Advantages**
    - More space for calculations
    - Opcodes don't need to specify registers
  - **Disadvantages**
    - Harder to know where things are on the stack

What is an Operand?

- Underlying meaning:
  - Whatever an operator operates on?
- Two meanings here (don't confuse them):

1) Extra bytes after the instruction opcode in memory,

e.g.

ld a **42**

2) Entries in the operand stack.

Machine Instructions as Stack Operators

- Arithmetic:

  e.g. **add** - adds top 2 stack entries
  …, val1, val2 → …, val1+val2

  e.g. **sub** - subtracts top 2 stack entries
  …, val1, val2 → …, val1-val2

  e.g. **neg** - negates top stack entry
  …, val → …, -val

- Similarity: **mul**, **div**, **rem**

remainder

Example of a Stack Machine (JVM mnemonics)

$$5\ 2\ +\ x\ x\ *\ y\ y\ *\ +\ \sqrt{\ }\ *\ 8\ +$$

```
push 5
push 2
add
load x
load x
mul
load y
load y
mul
add
call √
mul
push 8
add
```

~~more next week~~

Operation	Stack
	empty
5	5
2	5,2
+	7
x	7,3
x	7,3,3
*	7,9
y	7,9,4
y	7,9,4,4
*	7,9,16
+	7,25
SQRT	7,5
*	35
8	35,8
+	43

Bitwise Boolean Operations

- Boolean operations on one bit
  0=false, 1=true

“eXclusive OR”

OR	0	1
0	0	1
1	1	1

AND	0	1
0	0	0
1	0	1

XOR	0	1
0	0	1
1	1	0

- Can do these bit-wise on binary values. Example for XOR:

```
0 0 1 1 1 0 1 1 1 1 ...............0 1 1 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 1 ...............1 1 1 1 0 1 1 0 0
------------------------------------------------------------------
0 0 1 0 1 0 0 1 1 0 ...............1 0 0 1 0 0 1 0 0
```

XOR: done on top 2 stack entries (similar to: or, and):
…, val1, val2 → …, val1 XOR val2

Jumps

1
- Unconditional jumps
  - Operand stack is not used!

2
- Conditional jumps

0 x 1000

ifeq: default if is equals to 0

ifeq N // jumps to N if val=0

…, val → …

if_cmpeq N // jumps to N if val1=val2

…, val1, val2 → …

compare

Conditional Jumps with Other Comparisons

jump if	val	=	0	eq
		<		lt
		≤		le
		≠		ne
		>		gt
		≥		ge

≠

6 operators: ifeq, iflt, ifle, etc.

also: if_cmpeq, if_cmplt, etc.

Summary

We have now seen:

- What is a subroutine and how it is implemented.
- What is a stack and how it is used for implementing subroutines.
- What is the importance of saving registers and how these can be used to perform computations.
- What is reverse polish notation and how it is used to to do calculations using an operand stack.
- What is a stack machine and how it operates using a stack instead of registers.
- What are Bitwise Boolean operations and Jump instructions.

Notes on Exercises

- You will need to use an algorithm to convert math expressions from the usual “infix” notation to reverse-Polish. This algo is called Dijkstra's Shunting-Yard Algorithm.

- Have a look at the following links:
  - https://brilliant.org/wiki/shunting-yard-algorithm/
  - https://en.wikipedia.org/wiki/Shunting-yard_algorithm