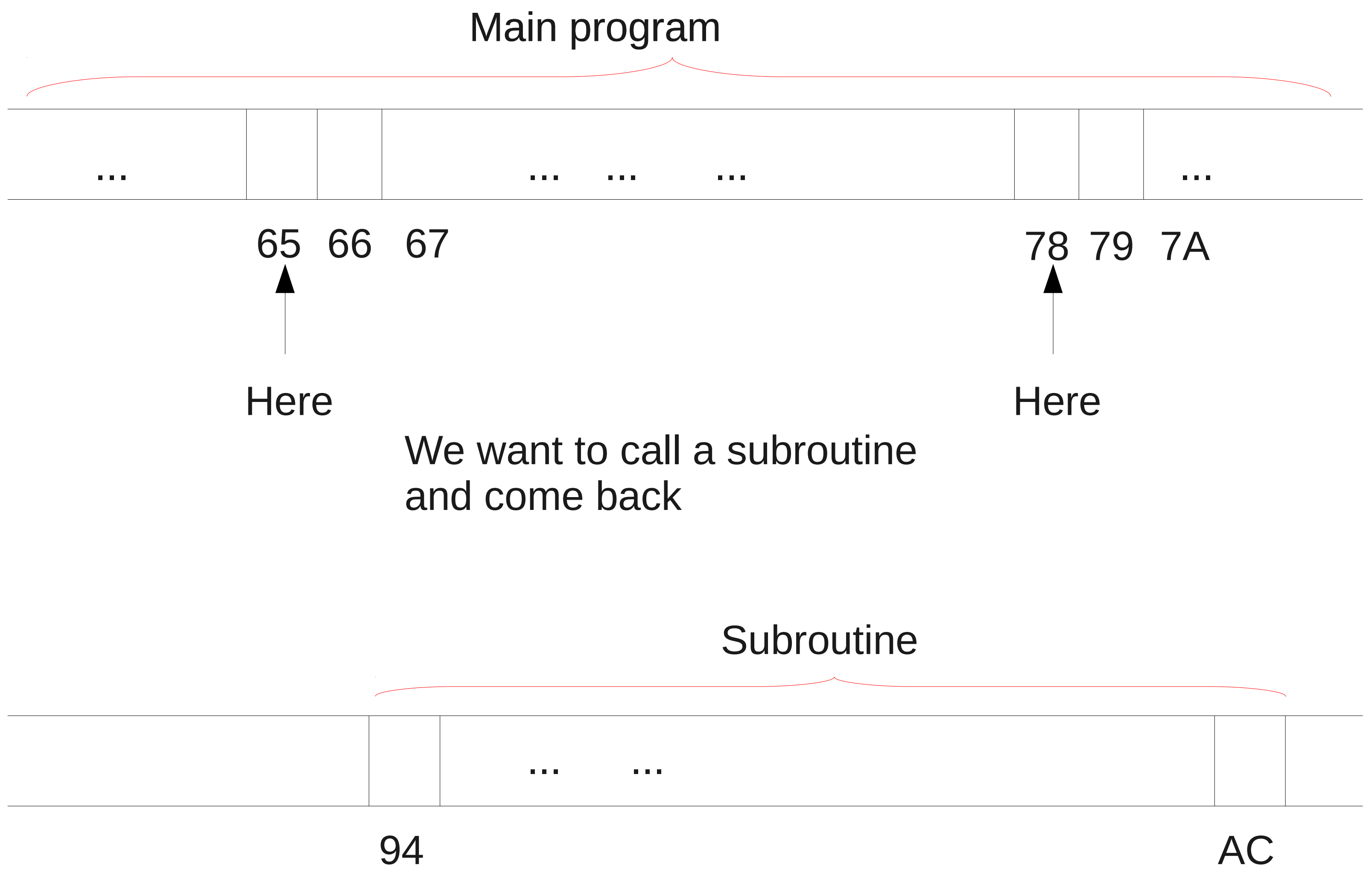
Subroutines and Stacks

Ata Kaban
University of Birmingham

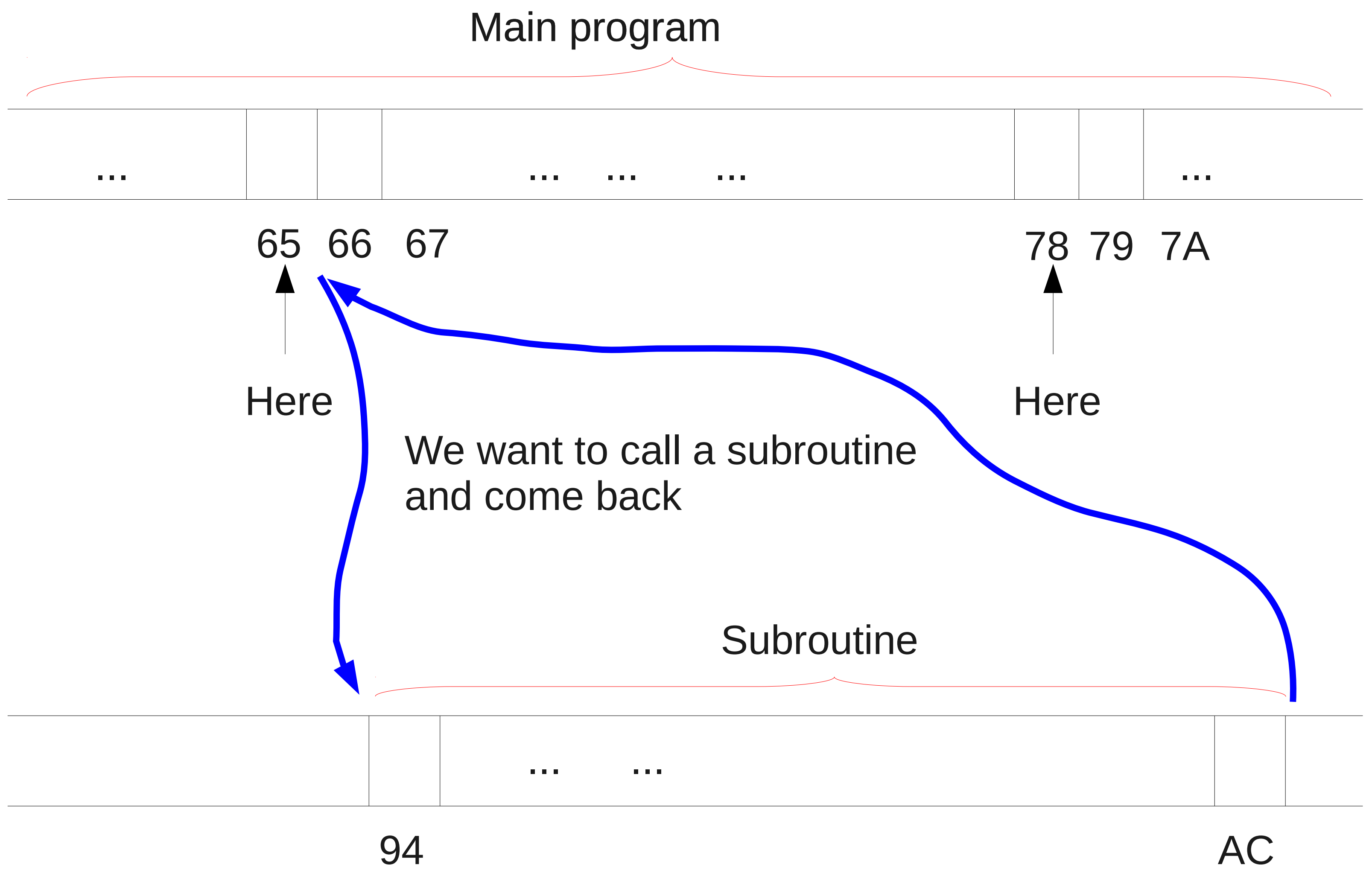
Credits to: Steve Vickers

Subroutines (methods)

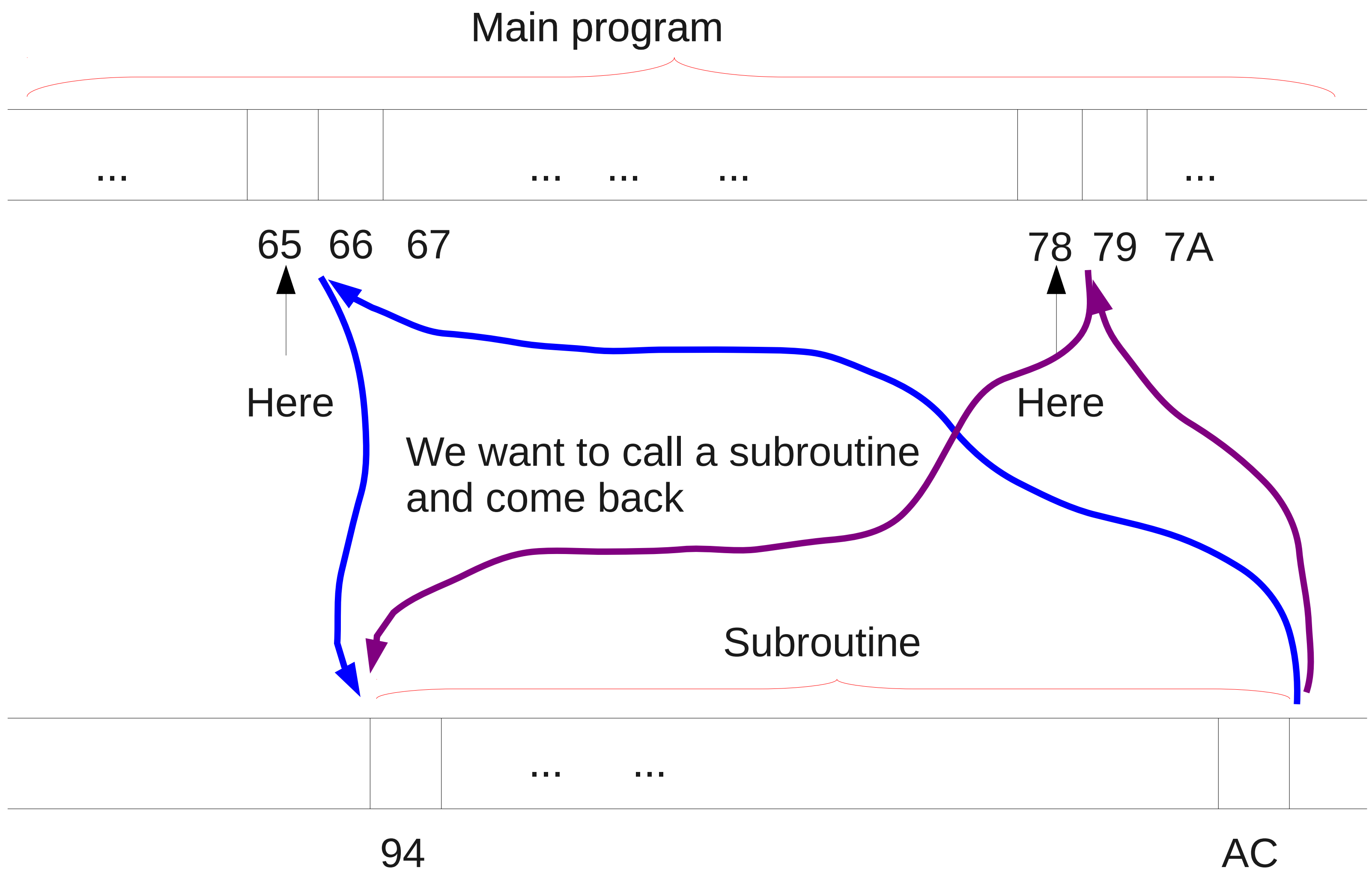


We want to call a subroutine
and come back

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

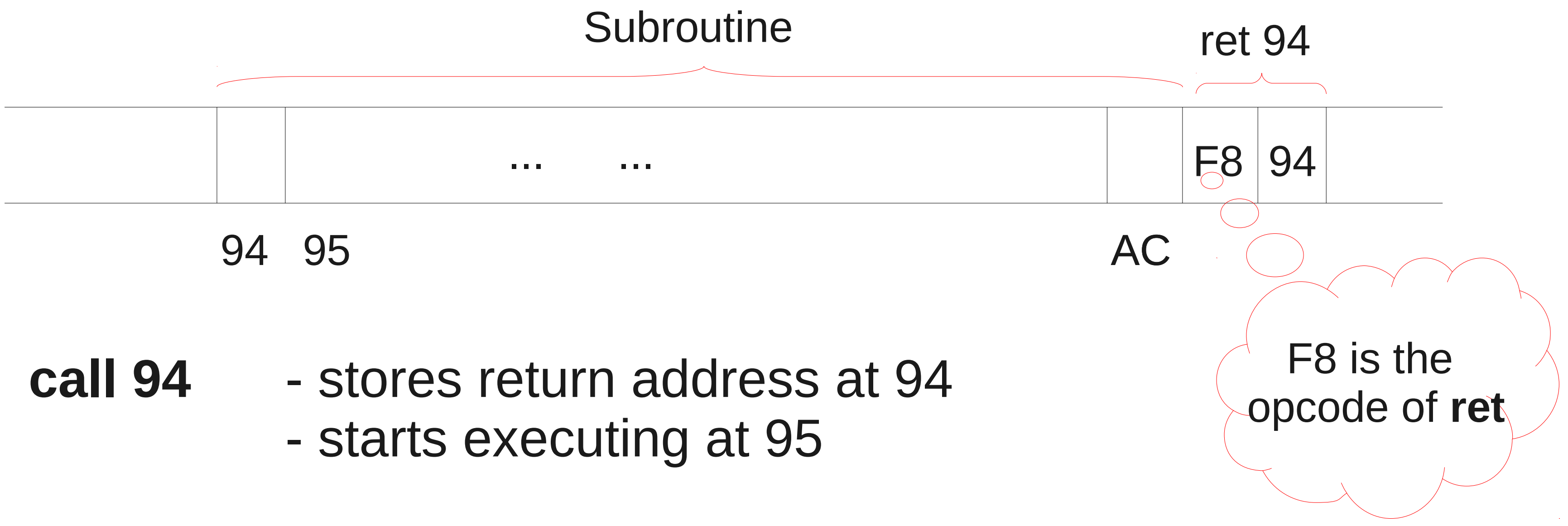
Two new operations:

call operand	Like jp, 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

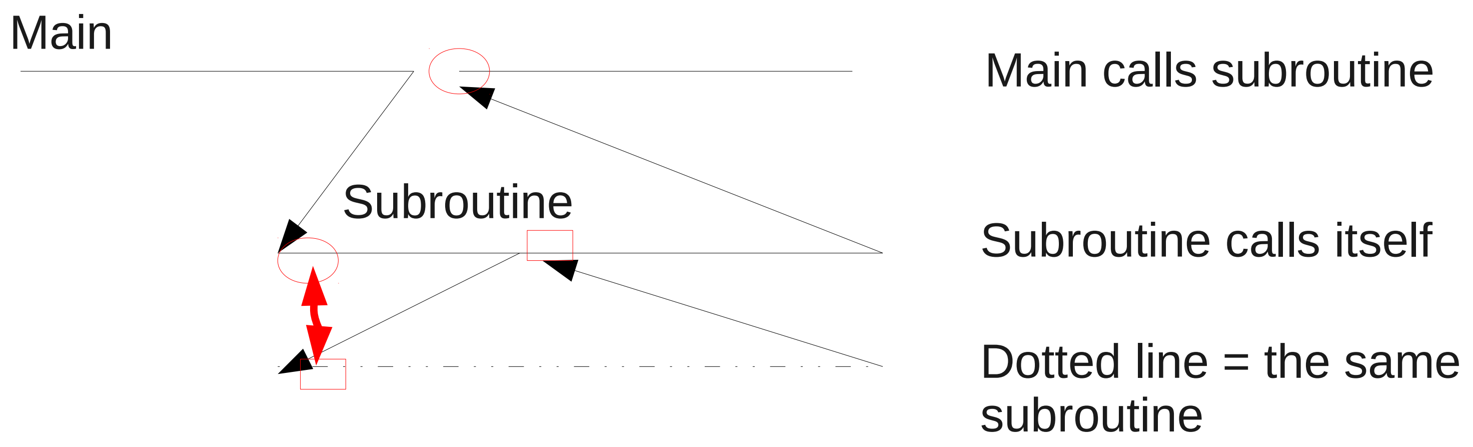


call 94 - stores return address at 94
- starts executing at 95

ret 94 - loads pc with return address at 94

Disadvantage of Idea 1

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

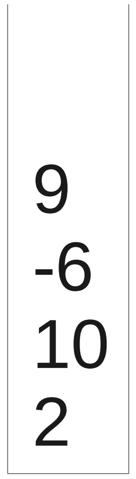


History: 2 early programming languages

- 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.

Stacks

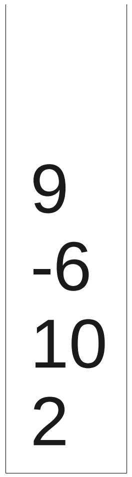
- A stack can flexibly store a variable number of bytes
- LIFO = last in, first out



Stacks

a	b	c
42	54	13

push a



Stacks

a	b	c
42	54	13

push a

42
9
-6
10
2

Stacks

a	b	c
42	54	13

pop b

42
9
-6
10
2

Stacks

a	b	c
42	**42**	13

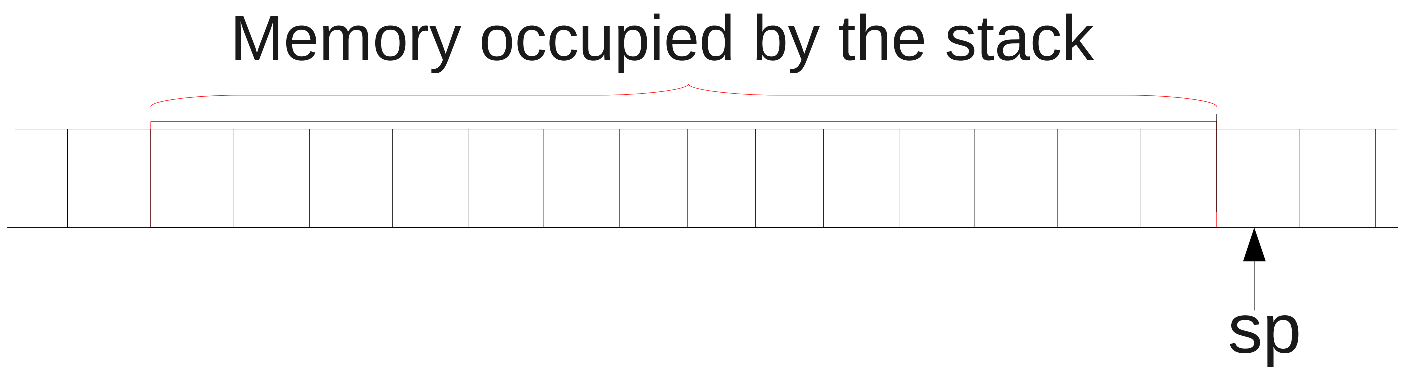
pop b

```
9
-6
10
2
```

In memory

- Give CPU another register

  sp (stack pointer) – shows where top is



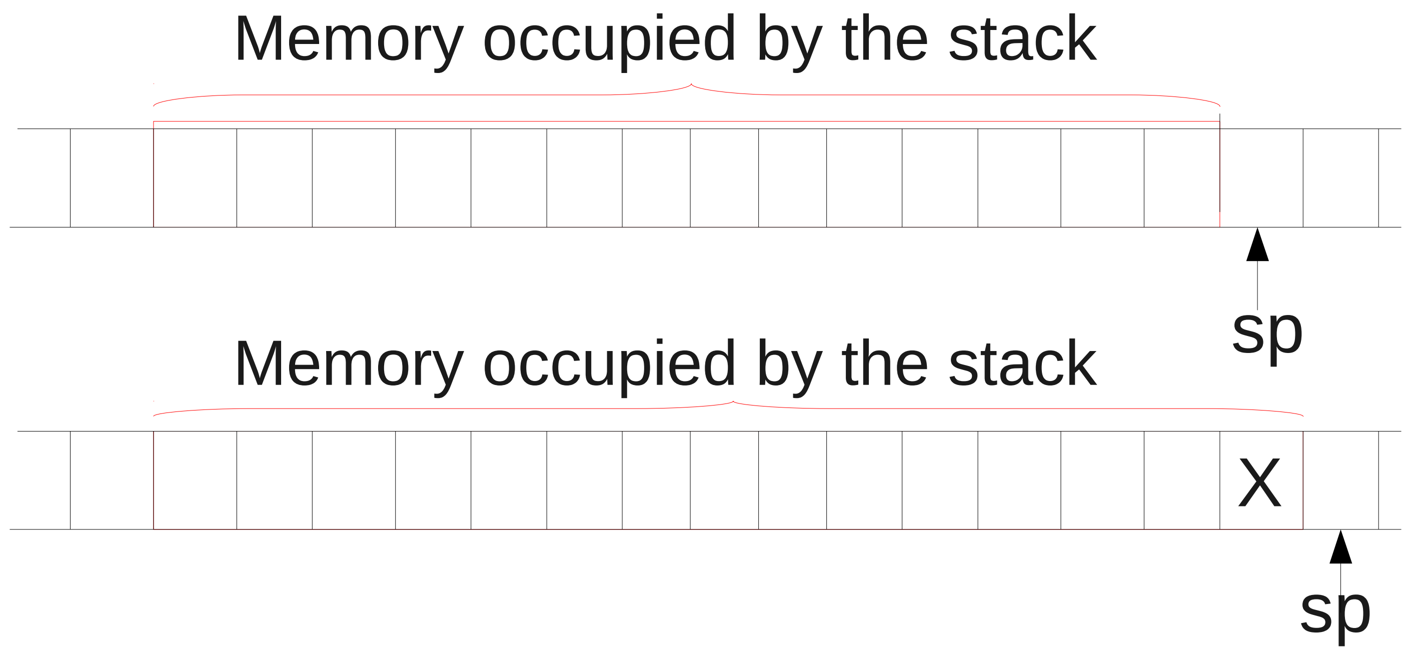
- Don't need to know where bottom is!
  - Provided that we are careful: only pop when you know you've pushed

In memory

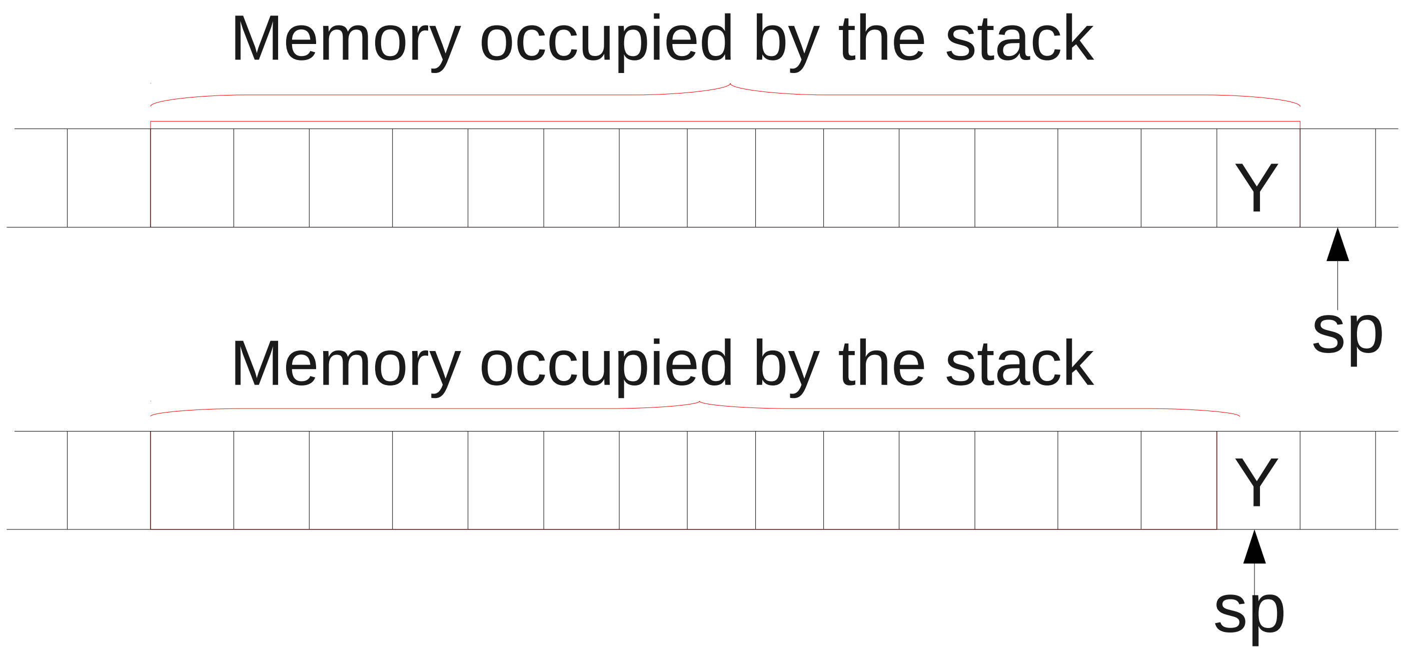
- To push a value X:
  - Write the value to memory at address sp
  - Add 1 to sp

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

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



- To pop a value:
  - Subtract a value 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.

Call and Return

- Store return addresses on stack.

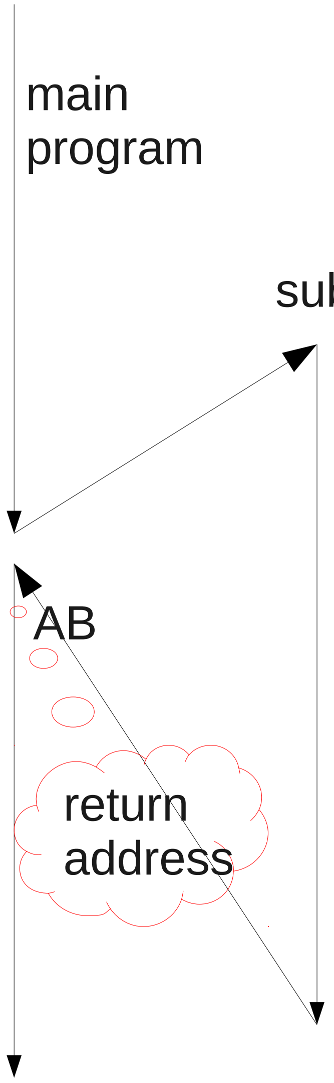
```
call N          works as if:
                push pc   // push program counter
                ld pc N   // jump to N

ret             works as if:
                pop pc    // pop return address
                          // & jump to it
```

(none of these are actual operations)

Saving 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 = <u>stack frame</u>
  Java method-calls develop this idea.



registers:

a	b	c
9	10	42

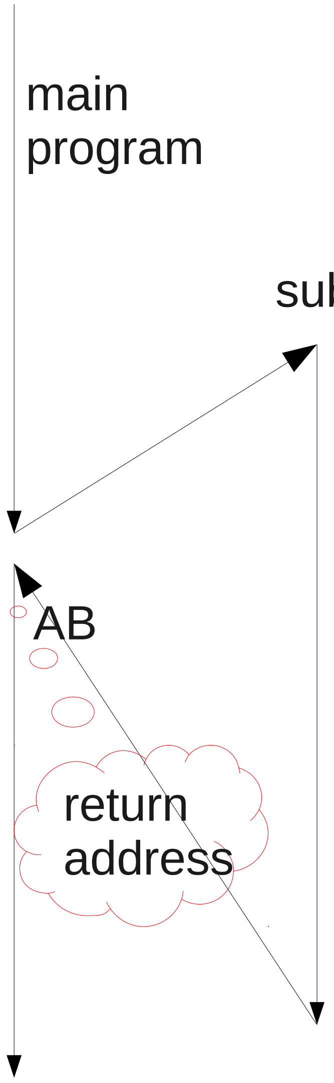
push a
push b
push c
…

Stack:

…, AB
…, AB, 9
…, AB, 9, 10
$\ldots, \underbrace{\text{AB, 9, 10, 42}}_{\text{stack frame}}$

registers:

a	b	c
9	10	42
9	10	42



	Stack:
push a	…, AB
push b	…, AB, 9
push c	…, AB, 9, 10
…	…, AB, 9, 10, 42

$$\ldots, \underbrace{AB, 9, 10, 42}_{\text{stack frame}}$$

	a b c	
pop c	?, ?, 42	…, AB, 9, 10
pop b	?, 10, 42	…, AB, 9
pop a	9, 10, 42	…, AB
ret	9, 10, 42	….

Toy CPU (for the exercises)

- All values & addresses are one byte (=8 bits)
- Registers

a, b, c	: general purpose
[b]	: references a memory location
pc	: program counter
sp	: stack pointer

- Operators

ld	: load
add	: add
sub	: subtract
st	: store
jp	: jump
jnz	: jump if not zero
push, call	: push, method call
pop, ret	: pop, return

- Some syntax & opcodes (don't learn these!)

ld a <val>	06, <val>
ld b <val>	0E, <val>
ld c <val>	16, <val>
add a [b]	24
add b <val>	2E, <val>
sub c <val>	56, <val>
st a [b]	64
jnz c <addr>	B6, <addr>
push a	C0
push b	C8
push c	D0
call <addr>	DE, <addr>
pop a	E0
pop b	E8
pop c	F0
ret	F8

On to Exercise 1 and 2 on the attached Notes.

Exercise 1

Write mnemonics and machine code (for the Toy CPU) for the following tasks:

a) A subroutine, starting at location 10 (hex) that sums memory locations, then stores the result after them. Assume that the calling routine initialised the register b with the address of the memory location of the summands, and the register c with the number of them. Don't forget the return.

b) A main routine, starting at location 00, that calls the subroutine twice: once for summing 4 locations starting at 2A, and once for summing 6 locations starting at 38.

Exercise 2

Your answer to 1 (a) will have used the a, b and c registers in its own calculations. Suppose this is inconvenient for the main program and it would be better not to lose the values that those three registers had when the subroutine was stored. Rewrite the subroutine so that it pushes the three registers at the start, saving their values on the stack, and then pops them back at the end so that the calling routine gets them back unchanged. Make sure you pop them in the right order!

- Next (this week):

  Stacks for calculations – evaluating arithmetic expressions

Stacks for calculation

- Example:

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

What order are operations applied in?

$$(5\underset{1}{+}2)\underset{6}{*}\underset{5}{\sqrt{x\underset{2}{*}x\underset{4}{+}y\underset{3}{*}y}}\underset{7}{+}8$$

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

Reverse Polish notation

- Order of operations is as written
- No brackets needed
- Powerful use of stack (= operand stack) to store intermediate results

Number or variable: push on stack

Operation: apply to top elements on stack

E.g. apply +

Before	After
8	
3	11
6	6
10	10
...	...

$$\left(5\underset{1}{+}2\right)\underset{6}{*}\underset{5}{\sqrt{\vphantom{y}}}\overline{x\underset{2}{*}x\underset{4}{+}y\underset{3}{*}y}\underset{7}{+}8$$

Reverse Polish: push operands, then operate.

We get:

$$5\quad 2\quad \underset{1}{+}\quad x\quad x\quad \underset{2}{*}\quad y\quad y\quad \underset{3}{*}\quad \underset{4}{+}\quad \underset{5}{\sqrt{}}\quad \underset{6}{*}\quad 8\quad \underset{7}{+}$$

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

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 stack:

E.g. subtraction

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

pop these two,
do subtraction,
push result

the rest is unchanged

Any expression can be converted to reverse Polish

- Then easy to execute with a stack

- Applications
  - Humans use reverse Polish directly
    - E.g. some pocket calculator – HP in 1970s, 80s
  - Forth programming language has two stacks:
    - Operand stacks for calculations
    - Return stack for module calls

Applications of reverse Polish

2. 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

- 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 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

[Forget the Toy CPU – remember these mnemonics (JVM)]

- 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

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

Pushing constants on the stack

push N

instruction needs extra byte

… → …, N

- Pretend we also can push values of variables [more on that next week]

load x

… → …, value of x

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

```
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

From slide 30

Jumps

- Unconditional jumps
  - Operand stack not used
- Conditional jumps

```
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

$$\text{jump if} \qquad \text{val} \left\{\begin{array}{c} = \\ < \\ \leq \\ \neq \\ > \\ \geq \end{array}\right\} 0 \qquad\qquad \begin{array}{l} \text{eq} \\ \text{lt} \\ \text{le} \\ \text{ne} \\ \text{gt} \\ \text{ge} \end{array}$$

6 operators: ifeq, iflt, ifle, etc.
also: if_cmpeq, if_cmplt, etc.

Summary

You have now seen:

- Registers for calculating
- Operand stacks for calculating
- Return stacks for saving return addresses and registers

Next week:

JVM puts them together: Bytecode

On to Exercise 3 and Exercise 4 on the 2-page Notes handed out yesterday.

- 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.
  - The attachment explains how it works
  - You can also read the Wikipedia page on this algo https://en.wikipedia.org/wiki/Shunting-yard_algorithm