Subroutines

- A subroutine or subprogram or procedure or function is a sequence of instructions that performs a well-defined task.

  The instructions are identified by a name. When the name of the subroutine is mentioned, the sequence of instructions associated with the name is executed once.

- ALL high level programming languages have the subroutine construct.

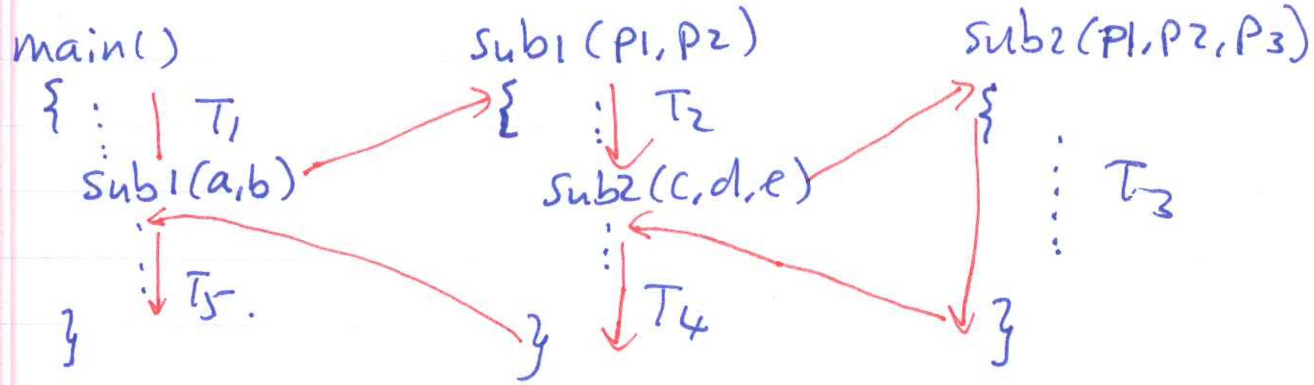
eg: in C:

```
main()
{ int A[100]; B[1000]
  int N1; N2 ;

  Sort(A, N1);        /* sorts N elements
                         in array A */
  Sort(B, N2);
}

~~Sort(A, N)~~
Sort ( int A[]; int N)
 {
   ...
 }
```

- When sort(A, N1) is called, the function body (instructions in Sort-function) is executed once.

  When Sort() is done, the program proceeds with the instruction following the Sort(A, N1) call.

- In general, subroutine calls can be nested arbitrarily:

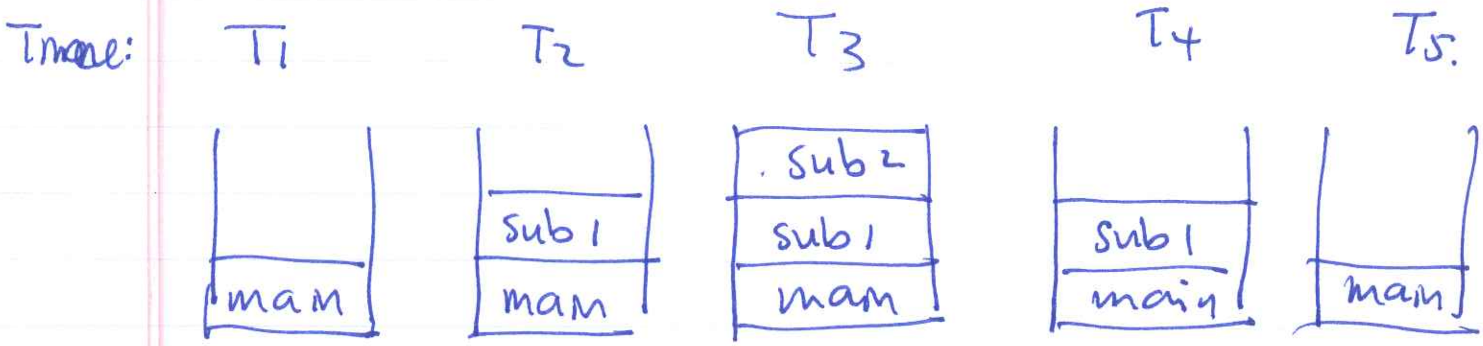


Execution is as follows:

(see arrow)

Time	Active
T_1	main
T_2	Sub 1, main pending
T_3	Sub 2, sub1 & main pending
T_4	sub 1, main pending.
T_5	main.

- We can "visualize" the Active & pending subroutine by means of a stack:



Only the subroutine at the top of the stack is active, while all others are pending.

- We can clearly see that subroutine call sequence is Last-In-First-Out: the subroutine called last will finish first, i.e. a stack-datastructure

- ~~To implement the subroutine construct of high level progr. languages, assembler language provides 2 instructions that~~

- Most assembler languages provide 2 instructions to implement the subroutine construct.

  ALL computers use a stack to implement ~~procedu~~ subroutines (the 2 instructions manipulate the stack

How does a subroutine look like in assembler code?

- C's functions (or Pascal's procedures & functions) look very structured:

output type: `float` — name: `SquareRoot` — input: `(float x)`

```
float SquareRoot (float x)
{
    -- ;
    -- ;
    -- ;
}
```

body: the part between `{` and `}`.

- In contrast, subroutines in assembler code looks just like "nothing".

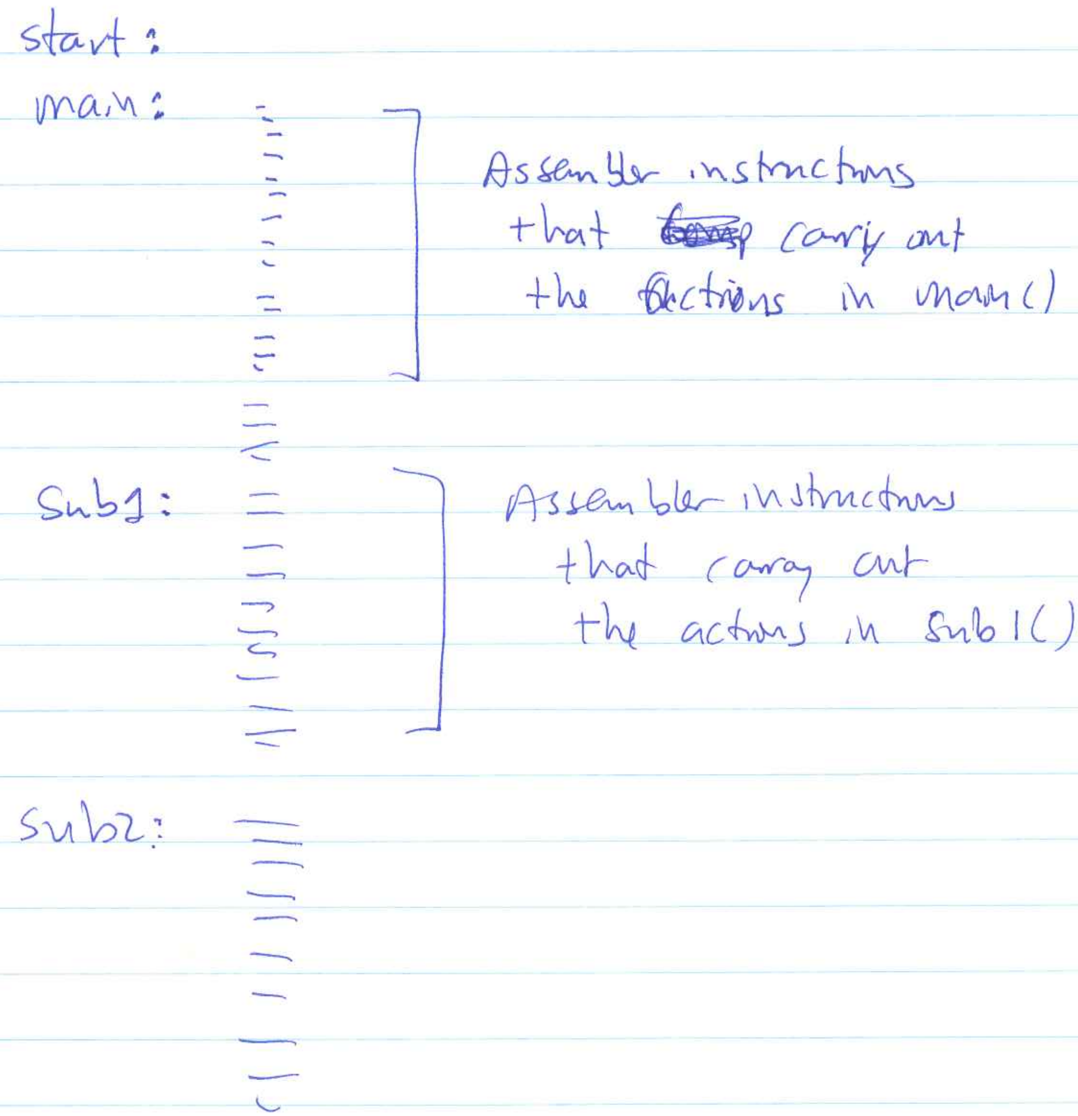
  You can't even find ~~a subroutine~~ the start of a subroutine easily.

  A C program with:

```
main()
{
  :
}
sub1( )
{
  :
}
sub2( )
{
  :
}
```

→ in assembler it looks like this:

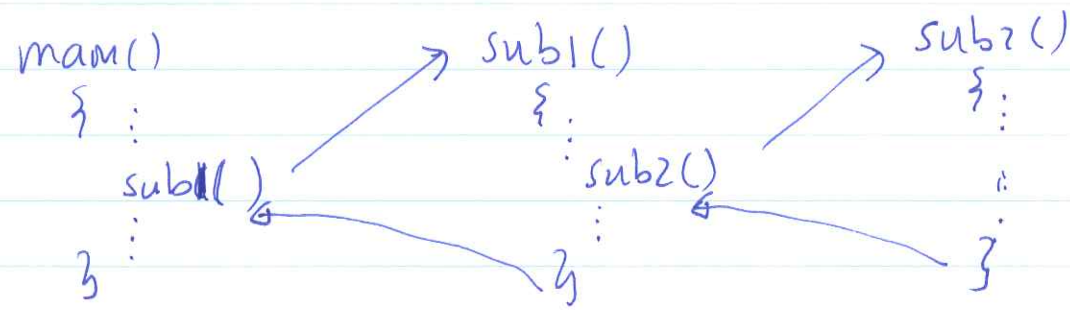
An assembler program that contain subroutines look like this:



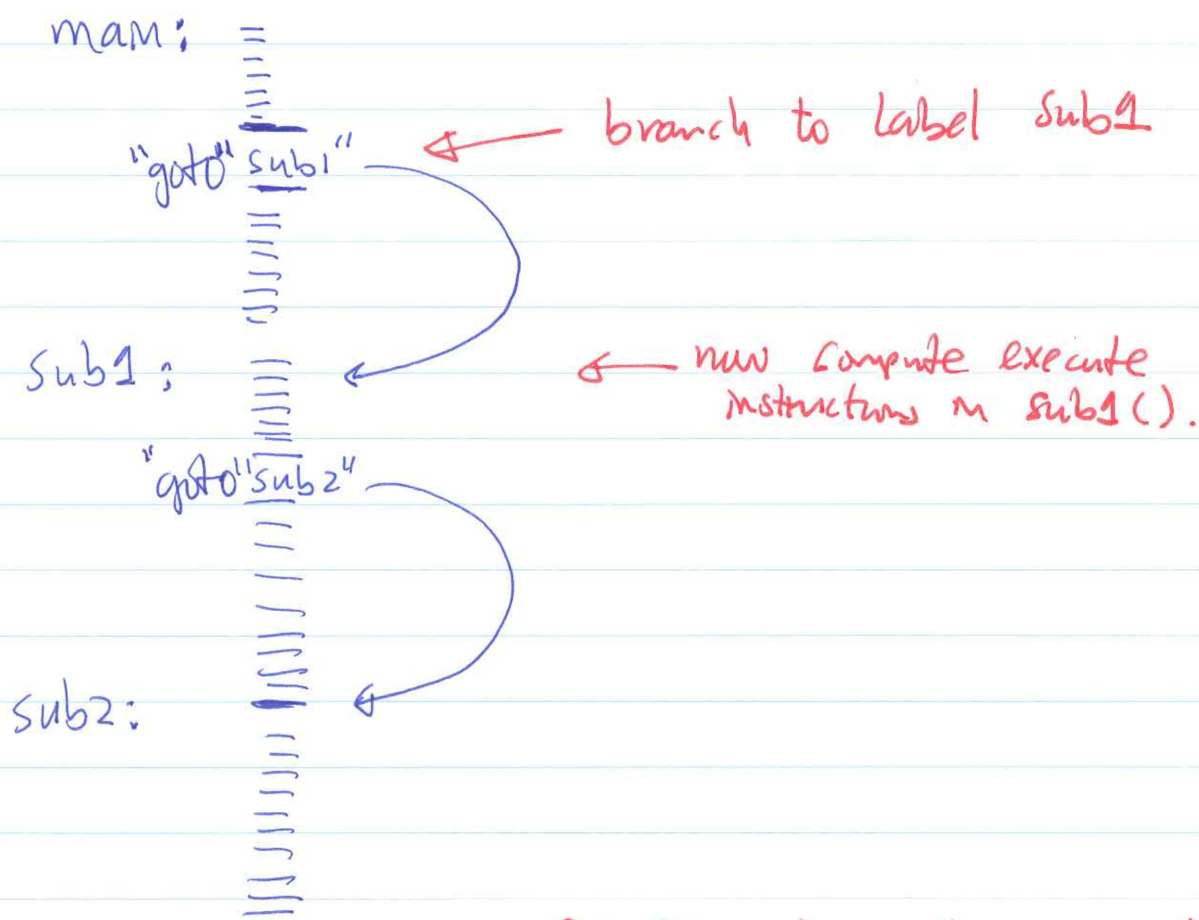
- So the start of a subroutine is just identified by a label! With all other labels you stick in the program for writing loops & if-statements, the start of a subroutine is really hard to find!

Effect of a function/subroutine call:

Recall in C:

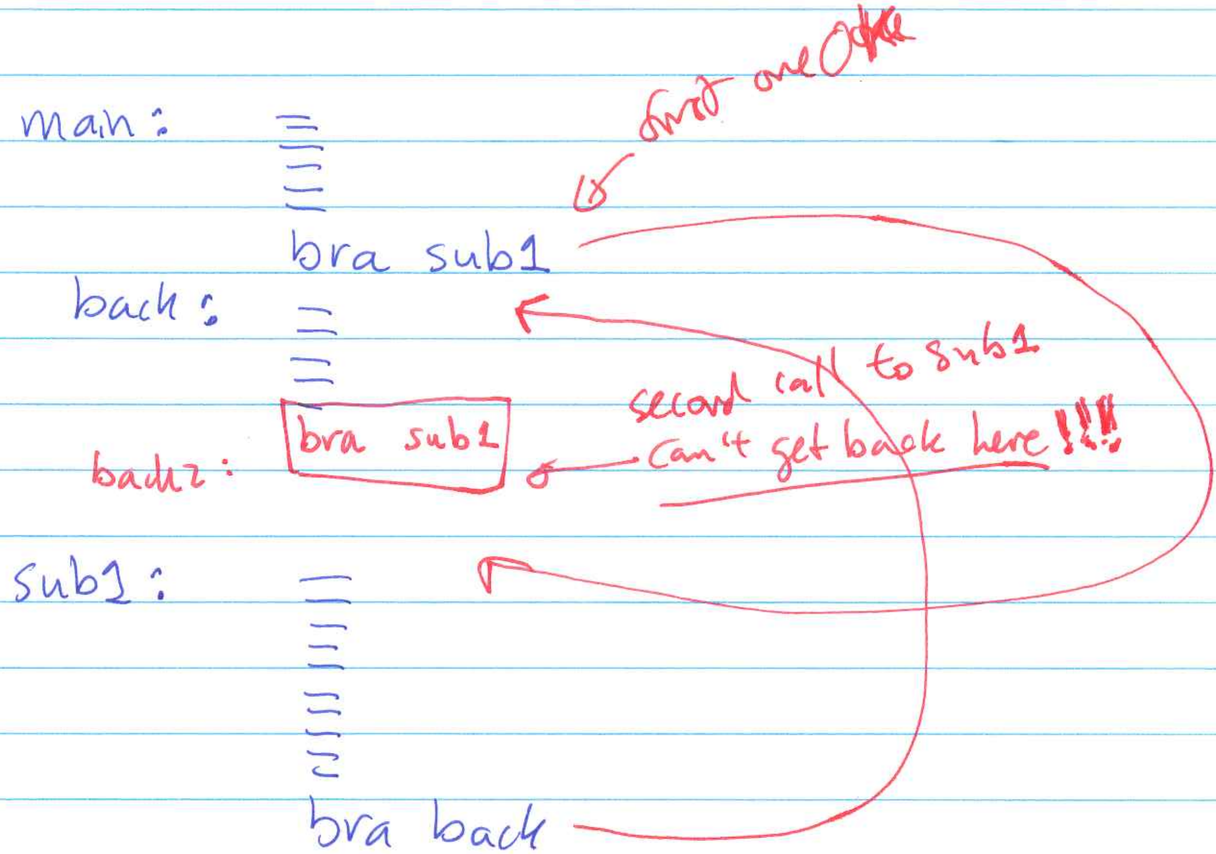


In assembler:



Question: how do we get back to where we left?

Q: What's wrong with this solution:



A: sub1 can't be shared because it always goes back to main.

The Right approach is the following:

main :

"goto sub1" — I need to save the location where I came from.

→ ~~the subroutine~~ A special subroutine jump ~~function~~ instruction is used!

sub1 :

"go back to where I came from".

Q: ~~How can I go~~
What do I need to do so that I can "go back to where I came from" ???

A: Save the location where you left off.

Where do I find this location ???

→ PC !!! (progr. Counter).

(Hansel & Gretel)

Moral of the story:

Before you branch to a subroutine you must save the "return address" (location where you want to go back to).

The data structure that is used to save the return addresses is a **Stack** because the subroutines are

First In Last Out (FILO).

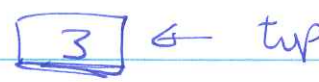
What is a Stack?

A data structure that grows and shrinks dynamically.

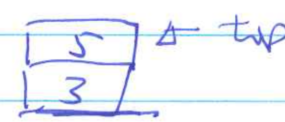
Items are "pushed" on the stack (stack grows) and "popped" off the stack (stack shrinks).

eg: empty stack :

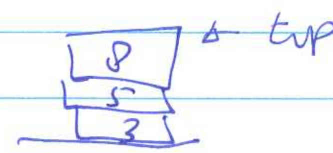
push(3) :



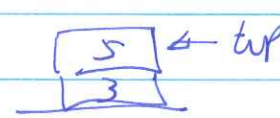
push(5) :



push(8) :



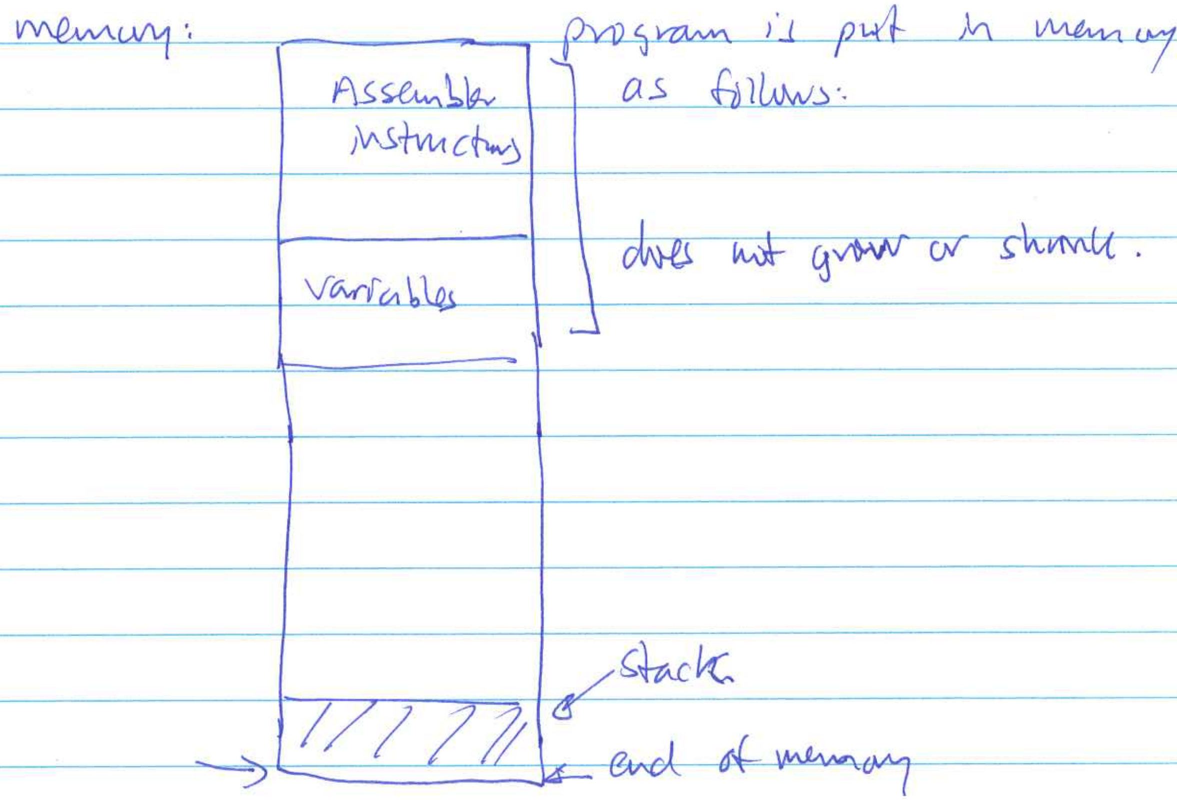
pop()



(pop() returns 8)
- top element of the stack.

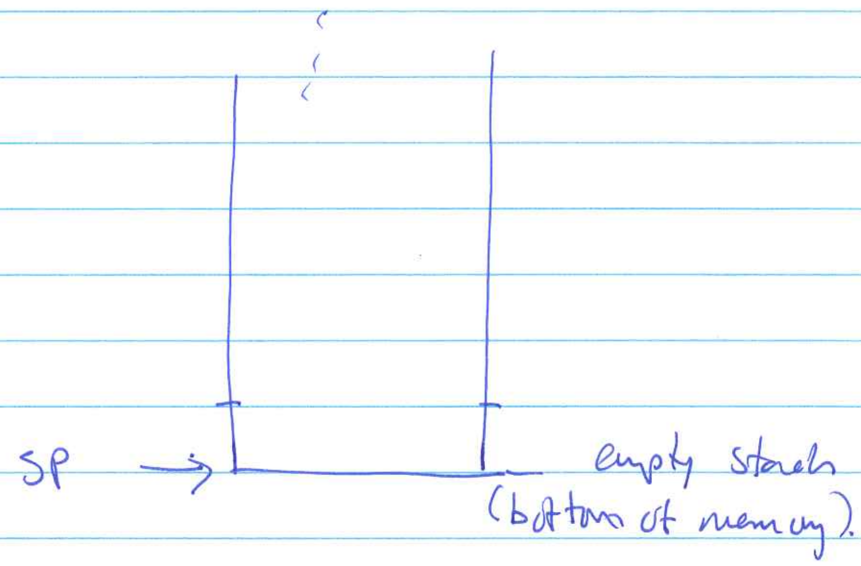
How are stacks implemented?

Each program has one "system stack" and it is organized as follows:

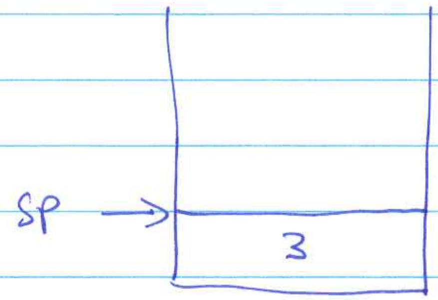


- ~~You can "push" or "pop" physically by physically putting things in memory.~~
- Push & Pop operations are implemented by "moving" a stack (top) pointer.

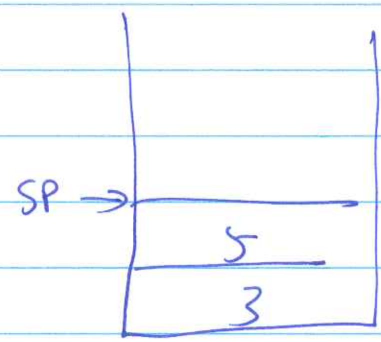
Schematically:



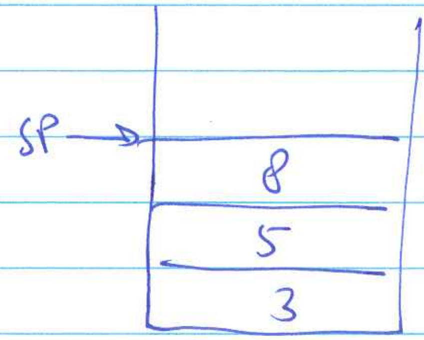
push (3):



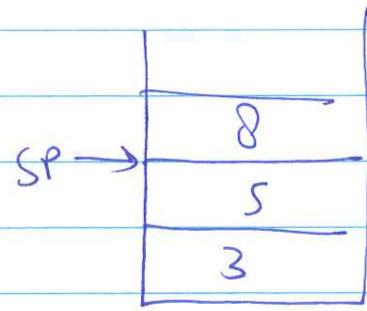
push (5)



push (8)



pop()



Note: 8 is still in memory, but if you follow the rule that pop always access the element pointed to by SP, you can't access 8 by pop().

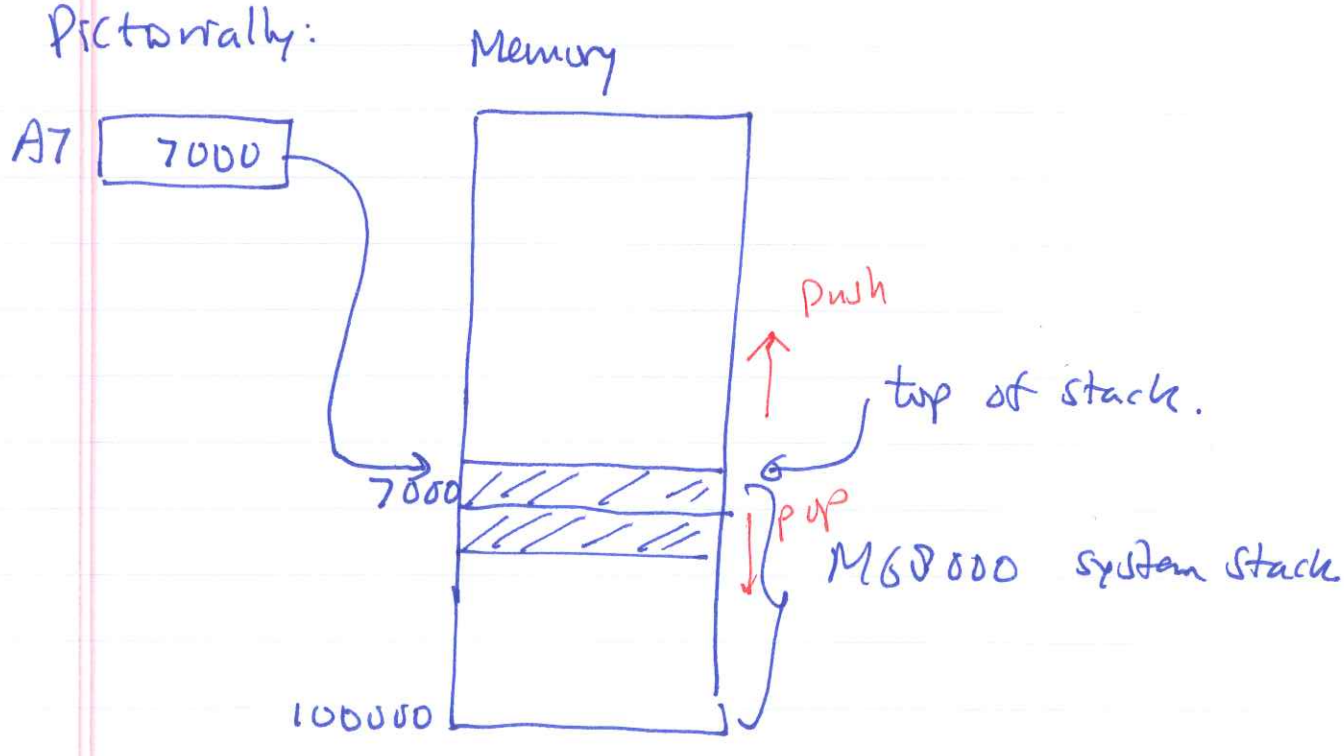
(next push will overwrite the value 8!).

- The M68000 system stack is implemented with the A7 address register.

- A7 is the "stack pointer" (SP) and always contains the address of the top of the stack in memory.

- The M68000 grows from high memory to low memory. In other words: when you push things (data) on the stack, the stack pointer decrements.

Pictorially:



e.

Subroutines in M68000

- Instructions used by caller to call the callee:

(1) BSR <label> (branch subroutine)

(2) JSR <label> (jump subroutine)

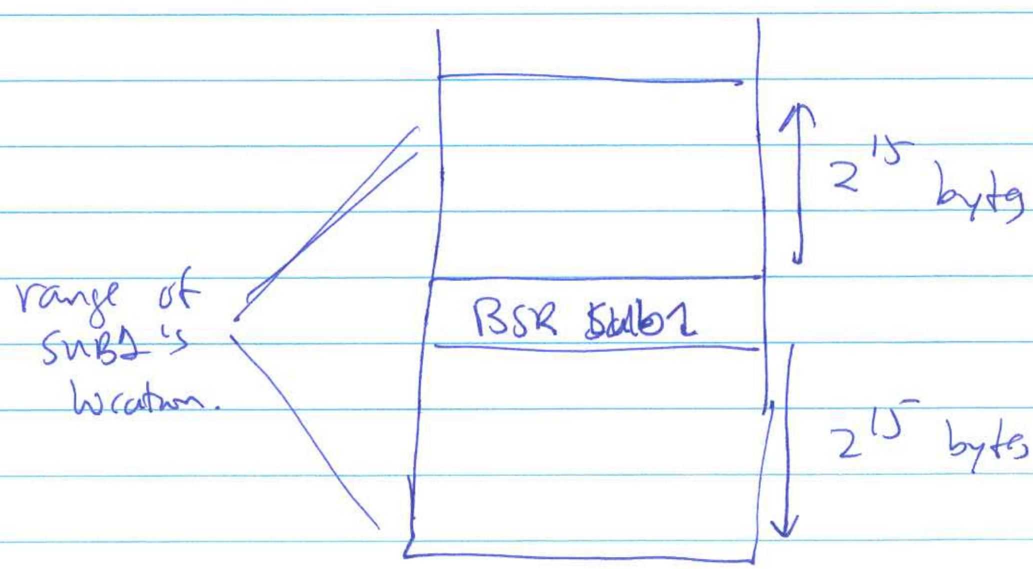
effect: (1) push PC on system stack
(2) branch / jump to <label>
BRA <label>

← Ask them what BRA <label> does!

Difference:

JSR can jump to a subroutine anywhere in memory.

BSR can only reach a "close" subroutine:



BSR is "shorter" (take less bytes to encode the instruction, so faster). But limited range.

- Instruction used by callee to return to the located where callee was called:

RTS (return from subroutine)

effect: pop address (a long word) off the top of the system stack and put ~~the~~ it into the Prog. Counter (PC).

Question: what happens then?

Answer: CPU fetch next instruction from PC, now at the return address!!

- RTS:

Syntax: RTS

Effect: Pop a long word off the stack and put it into the PC.

(i.e. execution now goes to the address value given by the top of the stack prior to the popping).

~~Demo subr.s~~

- How to implement subroutine call & return:

main program:

...

bsr subr

subr:

[instructions that are executed by the subroutine.

rts.

DEMO

⇒ Demo sub.s, — Do an example with: main → A → B