

The New Assembler-Simulator (nas)

30.10.2015

<http://www.cs.hku.hk/~fcmlau/nas.zip>

```
$ flex nas.l
$ bison -d nas.y
$ gcc -o nas lex.yy.c nas.tab.c
$ nas fact.as
$ ...
```

nas

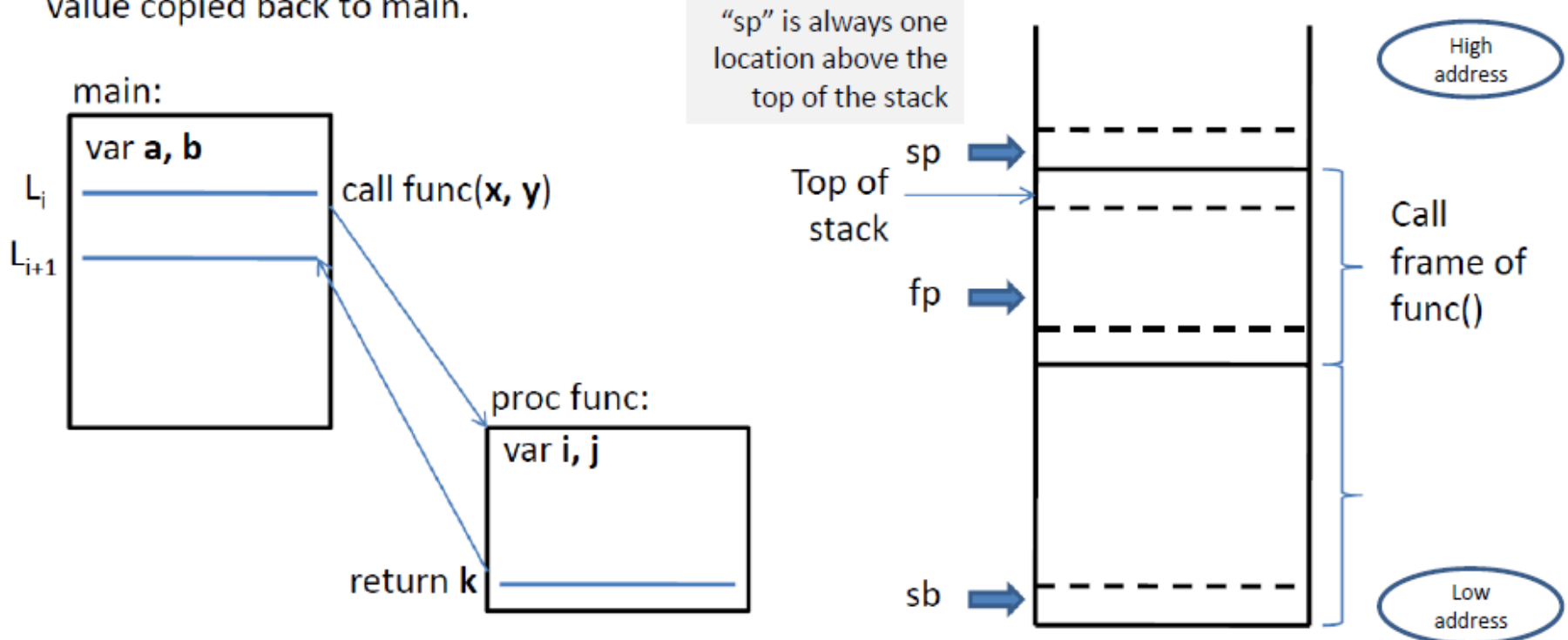
- A stack machine: all operations use push/pop
- Variables
 - In sas (Tutorial 2), there are 26 of them.
 - Can't have a..z in nas because nas has functions, and functions can have local variables whose names may clash with the global ones
 - In nas, variables are unnamed, stored inside the stack, and there can be as many as you want (or as the stack can hold)

```
push "Enter 5 numbers: "; puts_  
geti // = fp[0]  
geti  
geti  
geti  
geti // = fp[4]  
push 4; pop in // in = 4
```

Treat these as variables

The "index register" (registers are named)

Note: This “main” is different from C’s main() which is a function; here it is simply the outermost scope. Hence, **a**, **b** are global variables; **i**, **j** are local variables of func(). Assuming pass-by-value, **x**, **y** are copied to func() and treated as local variables. **k** is the return value copied back to main.

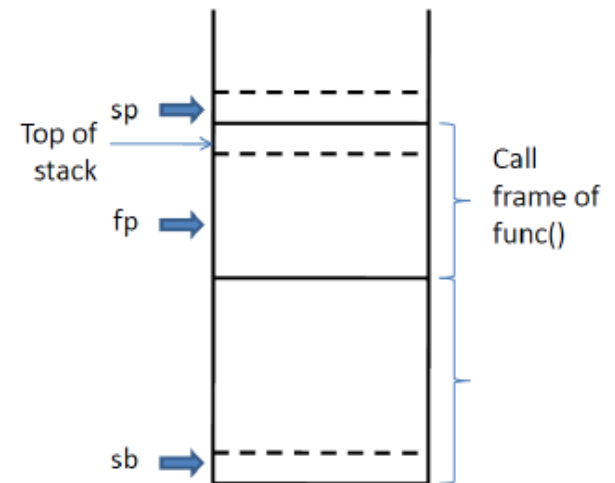


Special registers of the machine: **sp** (stack pointer), **fp** (frame pointer) which points *near* the bottom of the current frame, and **sb** (stack base) ... and **in** (index register) for implementing arrays

Also called **ac** (accumulator), and can be used as a general register, if you need one.

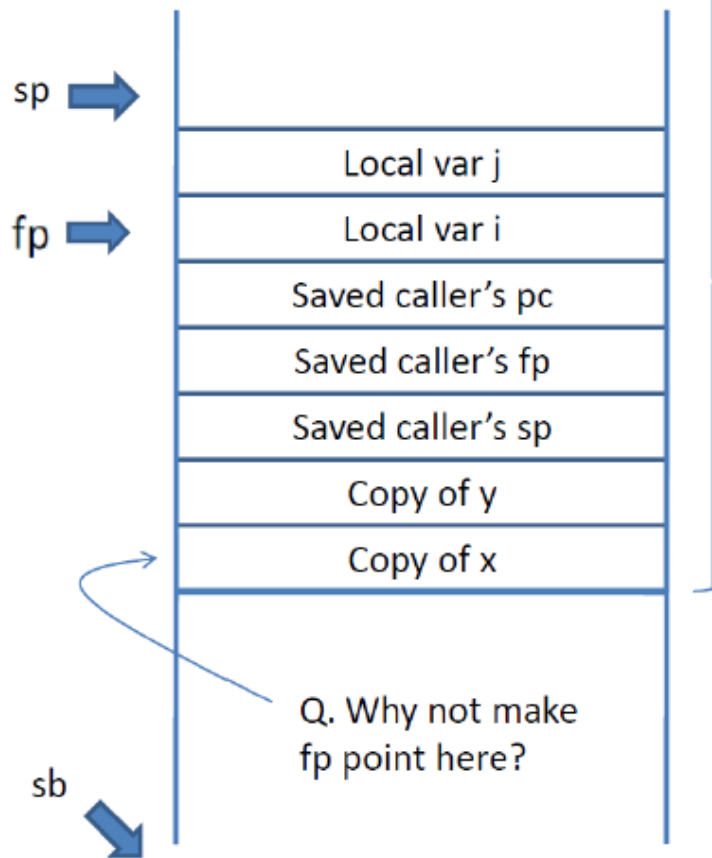
Using Variables

- Everything is on the stack, except strings (their addresses are pushed instead)
- To access local variables inside a function:
 - Relative to fp → e.g. “fp[-1]”
- To access global variables:
 - Relative to sb → e.g. “sb[3]”
- Only can access own frame and main, but not other frames in between



A Call Frame

"call func(x, y) "



Call frame of function/callee

i is referred to as fp[0]

j ... fp[1]

x ... fp[-5]

y ... fp[-4]

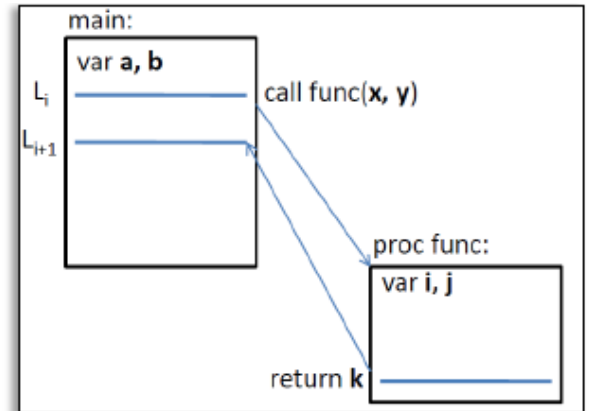
...

a ... sb[0]

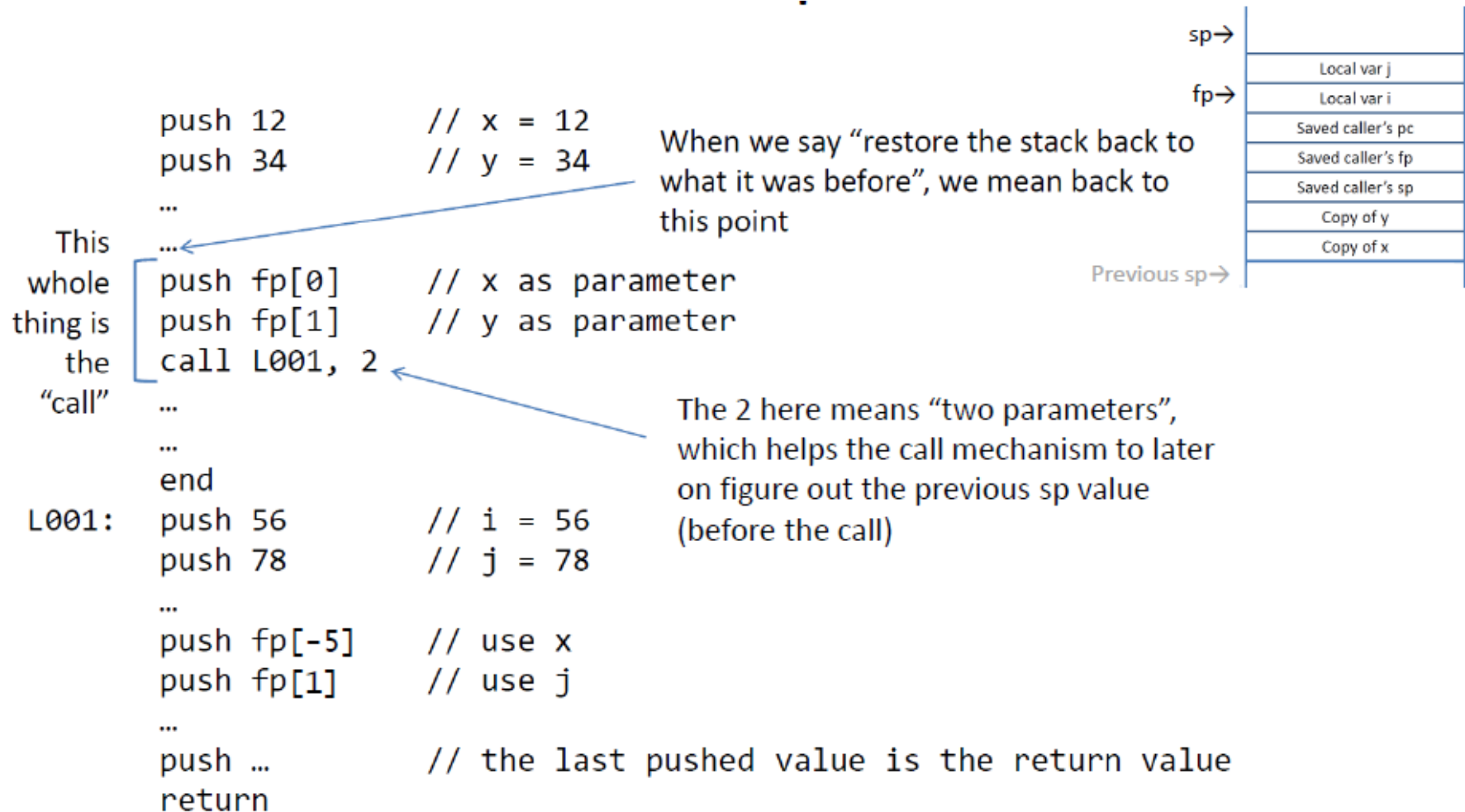
b ... sb[1]

...

k, the return value will be left on top of the stack after the restoration

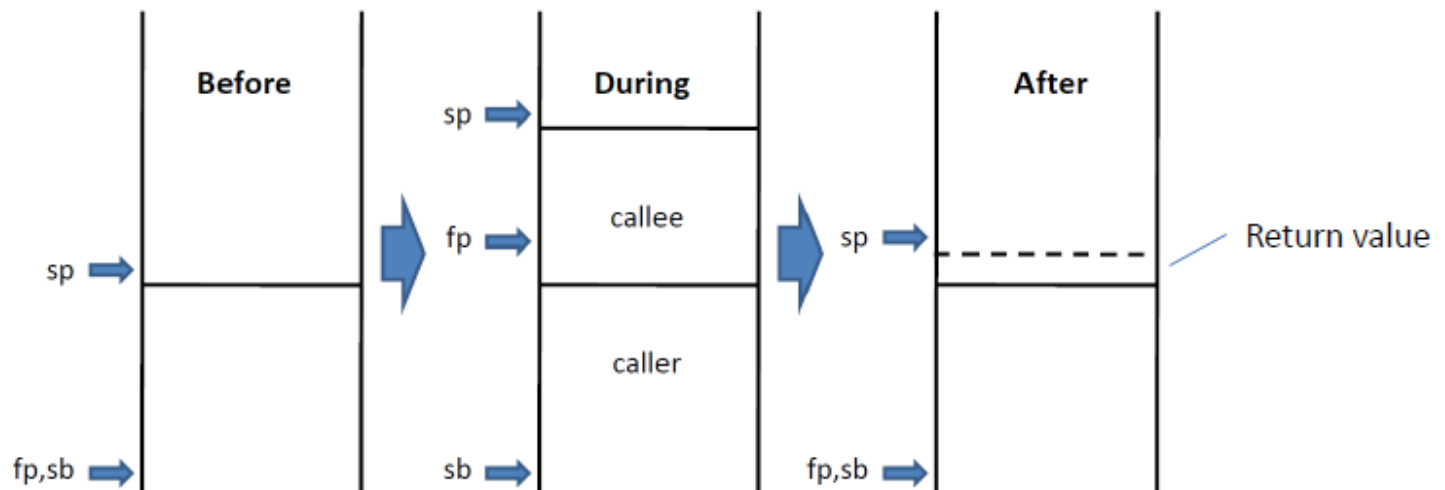


The General Call Pattern



Stack Frames

- When a program begins, $fp = sp = sb = 0$
 - sp rises and drops as actions in Main unfolds
- After a call, the stack must be restored to what it was before the call (+ the return value if there's one)
- At the moment of calling, caller's sp , fp , and pc (program counter) which points at the caller's next instruction (L_{i+1}) are saved in the callee's frame



Push & Pop

push 123	push "123" onto the stack
push -456	push "-456" onto the stack
push fp[2]	push the content of "where fp is pointing + 2"
push fp[-7]	push the "... - 7"
pop sb[4]	pop the stack and store the value in "the stack bottom + 4"
push fp[in]	push ... "where fp is pointing + the value of in"
push fp[-in]	Illegal; instead, you can make the value of in negative
push in	push the value of in
pop in	pop the stack and store the value in in
push fp	push the value of fp
push fp[0]	push the content of where in the stack fp is pointing

fp is used as an address

Example: max.as

```
// max.as
    push "Enter 2 numbers: "; puts_
    geti
    geti } Reads inputs and passes them as arguments to function
    call L001, 2
    puti_ // print the return value
    push " is larger"; puts
    end
L001: push fp[-4]
    push fp[-5] } Retrieves and pushes the two arguments
    compgt
    j1 L002
    push fp[-5]
    ret
L002: push fp[-4]
    ret
```

Do not print \n

Which is at the stack's top

Return value

Example: fact.as

```
// recursive fact.as
    push "Please enter a +ve int < 13: "; puts_
    geti
    call L001, 1
    puti
    end

// factorial():
L001:  push fp[-4]
        j0 L002
        push fp[-4]; push 1; sub
        call L001, 1 // recursive call
        push fp[-4]
        mul
        ret
L002:  push 1
        ret
```

Read n

Call fact(n)

Print return value

n

n = n - 1

Return n x fact(n - 1)

Example: rev-c.as

```
// rev-c.as
    push "Please enter a line:"; puts
    push 0; pop in                                // in = 0
L001:  getc; // NO pop fp[in] here !!              // fp[in] = getc
    push fp[in]; push 10; compeq; j1 L002          // if newline goto L002
    push in; push 1; add; pop in                  // in++
    jmp L001
L002:  push in; push 1; sub; pop in                // in--
    push fp[in]; putc_
    push in; j0 L003; jmp L002
L003:  push ''; putc
    end
```

Print a newline

