Assembly Language
Part 4

Stack-relative addressing
• With stack-relative addressing, the stack pointer acts sort of like the base pointer of an array, with the instruction’s operand specifier as the index
• In other words, the operand value is found at the address, which is the sum of the stack pointer’s value and the value of the operand specifier, as shown below:

\[
\text{operand value} = \text{value at SP[op specifier]}
\]

Stack-relative addressing
• A stack is not quite the same thing as an array
  – If you were putting several items in an array, you would probably put the first item at index 0, the second at 1, etc., until the nth item is placed at address \(n-1\)
  – Think of a stack as a backwards array—the nth item is placed at 0, and the first item at \(n-1\)
• Another (and perhaps more correct) way of thinking of the Pep/8 runtime stack, is that it grows upward in memory – that is, if two 2-byte values are placed on the stack beginning at address 0008, the second value would be at 0006 (instead of 000A)

Example
lda 'h', i ; push 'h' on stack
stbytea -1, s
lda 'e', i ; push 'e'
stbytea -2, s
lda 'y', i ; push 'y'
stbytea -3, s
lda '!', i ; push '!' stbytea -4, s
subsp 4, i ; announce 4 bytes are taken
charo 3, s ; output 'h'
charo 2, s ; output 'e'
charo 1, s ; output 'y'
charo 0, s ; output '!' addsp 4, i ; let go of stack storage
stop .end

Stack-oriented instructions
• The ADDSP and SUBSP instructions are used to add and subtract values to (or from) the stack pointer
• Remember, the stack grows upward – so SUBSP allocates stack storage and ADDSP deallocates it
• The example on the next slide illustrates (not especially efficient) use of the runtime stack to store data

All that for …?
• The previous example was just an illustration; the runtime stack is more commonly used for allocation of temporary memory during the run of a program, specifically for:
  – parameter & local variable allocation
  – procedure calls
Procedure (method) calls
Consider the following Java program:

```java
public class Message {
    public static void printMsg() {
        System.out.print("Hello");
        System.out.println();
    }
    public static void main(String[] args) {
        printMsg();
        printMsg();
        printMsg();
    }
}
```

Simplest version of method calling in Java:
- printMsg is void & takes no arguments
- printMsg & main are static, so we don't need to create an object

Equivalent Pep/8 program
```
.File: voidNoParams
BR main :
  ;******* void printMsg ()
printMsg: 
  STRO msg.d
  CHAR "n" ,
  RET0
  msg: ASCII "Hello\000"
  ;******* int main ()
main: CALL printMsg ;
CALL printMsg ;
CALL printMsg ;
STOP
.END
```

CALL is something like an unconditional branch in this instance, since there isn't any memory allocated for parameters

Pep/8 and procedure calls
- Although not immediately obvious from the previous example, both the CALL and RETn instructions use the system stack
- For this simple example, the stack is used to store the return address; when local variables are involved, the stack also stores these
- The next two slides illustrate how CALL and RETn work with respect to the stack

Pep/8 and procedure calls
- CALL instruction:
  `SP = SP - 2 ;` decrement stack pointer – the
  ; stack grows up from address
  ; FBCF, so we're moving 2 bytes
  ; up (size of an address) to FBCD
  Mem[SP] = PC; store current value of program
  ; counter, get ready for branch
  PC = Operand; branch to location specified
  ; by operand

Pep/8 and procedure calls
- RETn instruction:
  `SP = SP + n ;` At this point, we’re returning
  ; from a method – incrementing
  ; stack pointer by n bytes
  ; effectively deallocates memory
  ; for n local variables
  PC = Mem[SP] ; recall stored return address
  SP = SP + 2 ; complete pop from stack (give
  ; back memory address occupied)

Parameter passing
- In general, high level languages allow two types of parameter passing:
  - pass by value: a copy of the argument’s value is assigned to the corresponding parameter
  - pass by reference: the parameter is a pointer to the address of the original argument’s value
- Java uses both, but with less programmer control than is available in C++:
  - primitives are always passed by value
  - objects are always passed by reference
Pass-by-value example

```java
public void Printer(
    public static void line (int n) {
        int c;
        for(c=0; c<n; c++)
            Sop("\*";
        Sopn();
    }
    public static void main (String [] args) {
        line(7);
    }
}
```

What’s going on here

- Calling procedure (main) pushes arg(s) on stack with SUBSP, then pushes return address on stack with CALL
- Called procedure (line) allocates space for local variables with SUBSP, executes its body, deallocates local variables and pops return address with RETn
- Calling procedure pops arg(s) with ADDSP

---

Pep/8 equivalent

```pep8
****main()!
main: STRO msgI,d,DECJ,a,d,STRO msg2,d,DECJ,b,d,LDAt,a,b,STRA-2,s,NDLAb,b,NDSPA-4, x;CALL order: LDA x,STRO msg2.d,DECO J,a,b,STRO msg3,d,DECO b,d,CHARO "n", STOP
```

A program with reference parameters

```c
#include <iostream.h>
int main() {
    cout << "Enter an integer: ";
    cin >> a;
    cout << "Enter an integer: ";
    cin >> b;
    cout << "Ordered they are: ";
    r = a;
    s = temp;
    } // ra1
    void order (int & x, int & y) {
        if (x > y) {
            swap (x, y);
        } // ra2
    }
```

---

Pep/8 equivalent

```pep8
****main()!
main: LDA T, i<begin procedure call
STA-2, s<get ready to push arg
SUBSP 2, i<push arg
CALL line ADDSP 2, i<push return address STOP POP arg
```
Pep/8 equivalent

BR main
  a. BLOCK 2  ; global variable
  b. BLOCK 2  ; global variable

;****** void swap (int& r, int& s)
  r. EQUATE 6  ; formal parameter
  s. EQUATE 4  ; formal parameter
  temp. EQUATE 8; local variable
  swap. SUBSP 2, i ; allocate local
  LDA r, sf
  STA temp, s
  LDA s, sf
  STA r, sf
  LDA temp, s
  STA s, sf
  RET2

; allocate local,
; pop retAddr

int a, b; // global variables
void swap (int& r, int& s) {
  int temp;
  temp = r;
  r = s;
  s = temp;
}