C Review and Dissection III:

Function Calls and Array Access

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

Today

- Activation Record
  - Arguments and Automatics
  - Return value
- Arrays
  - Layout in memory
  - Code for accessing elements

Activation Record

- Each time a function is activated (run), space is needed to store data – this is called an activation record
  - arguments – data passed to function
  - local variables
  - return value
  - other bookkeeping information
- Calling a function B from function A involves
  1. Possibly placing **arguments** in a mutually-agreed location
  2. **Transfering control** from function A to the function B
  3. Allocating space for B's local data
  4. **Executing** the function B
  5. Possibly placing **return value** in a mutually-agreed location
  6. De-allocating space for B's
  7. **Returning control** to the function A

Register Conventions

- The register conventions dictate how the compiler uses registers and how values are preserved across function calls.
- Table 6-3 shows the types of registers affected by these conventions.
- Table 6-4 summarizes how the compiler uses registers and whether their values are preserved across calls.

How Register Types Are Affected by the Conventions

Table 6-3 MSP430 Optimizing C/C++ Compiler v 4.1

Register	Type Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Argument pointer	Used as a base value from which a function's parameters (incoming arguments) are accessed
Stack pointer	Holds the address of the top of the software stack
Program counter	Contains the current address of code being executed

Register Usage and Preservation Conventions

Table 6-4 MSP430 Optimizing C/C++ Compiler v 4.1

Register	Alias	Usage	Preserved by Function (1)
R0	PC	Program counter	N/A
R1	SP	Stack pointer	N/A (2)
R2	SR	Status register	N/A
R3		Constant generator	N/A
R4-R10		Expression register	Child
R11		Expression register	Parent
R12		Expression register, argument pointer, return register	Parent
R13		Expression register, argument pointer, return register	Parent
R14		Expression register, argument pointer, return register	Parent
R15		Expression register, argument pointer, return register	Parent

(1) The parent function refers to the function making the function call. The child function refers to the function being called.

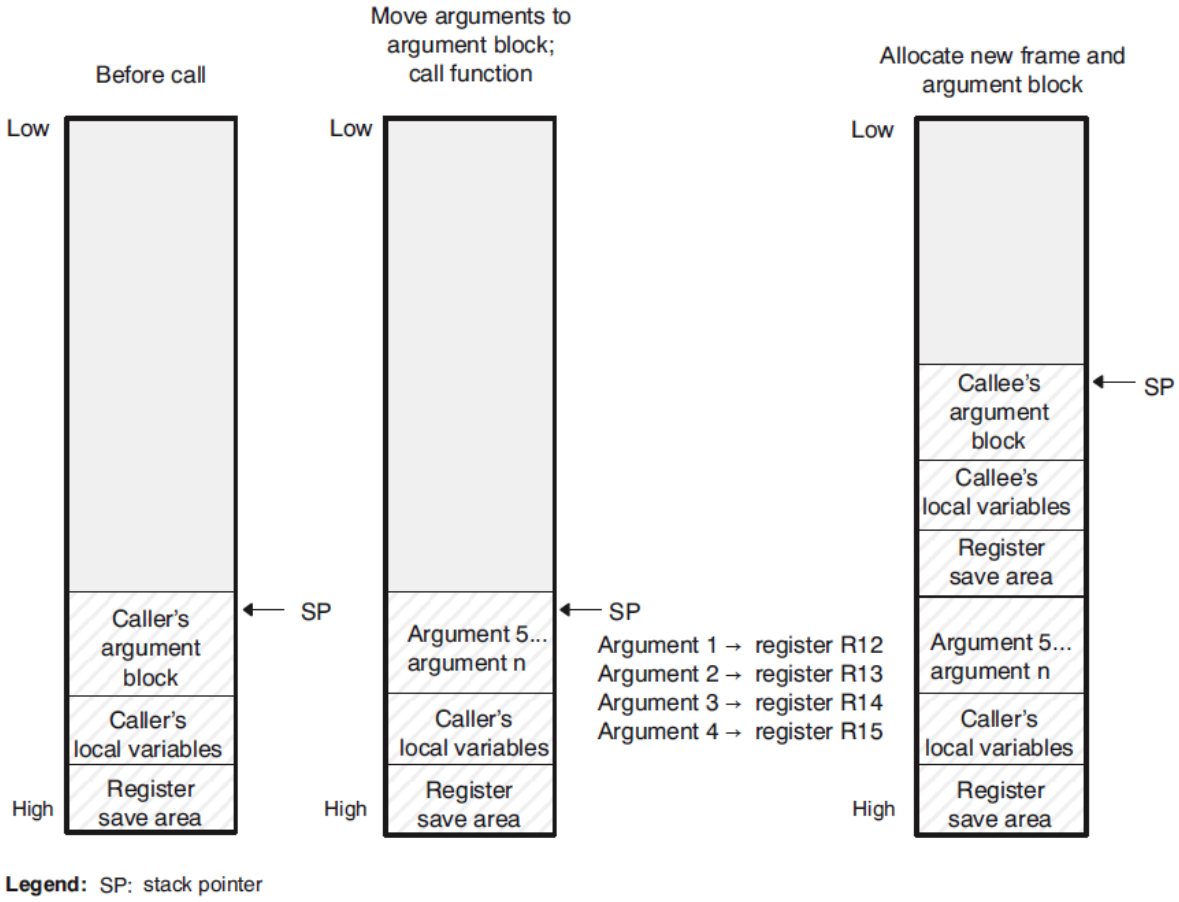
(2) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

Function Structure and Calling Conventions

- The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.
- The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:
  - **Argument block.** The part of the local frame used to pass arguments to other functions. Arguments are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.
  - **Register save area.** The part of the local frame that is used to save the registers when the program calls the function and restore them when the program exits the function.
  - **Save-on-call registers.** Registers R11-R15. The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.
  - **Save-on-entry registers.** Registers R4-R10. It is the called function's responsibility to preserve the values in these registers. If the called function modifies these registers, it saves them when it gains control and preserves them when it returns control to the calling function.

Function Structure and Calling Conventions

- In this example, arguments are passed to the function, and the function uses local variables and calls another function. The first four arguments are passed to registers R12-R15. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.



How a Function Makes a Call

- A function (parent function) performs the following tasks when it calls another function (child function).
  - 1. The calling function (parent) is responsible for preserving any save-on-call registers across the call that are live across the call. (The save-on-call registers are R11-R15.)
  - 2. If the called function (child) returns a structure, the caller allocates space for the structure and passes the address of that space to the called function as the first argument.
  - 3. The caller places the first arguments in registers R12-R15, in that order. The caller moves the remaining arguments to the argument block in reverse order, placing the leftmost remaining argument at the lowest address. Thus, the leftmost remaining argument is placed at the top of the stack.
  - 4. The caller calls the function.
  - Functions defined in C++ that must be called in asm must be defined extern "C", and functions defined in asm that must be called in C++ must be prototyped extern "C" in the C++ file.

How a Called Function Responds

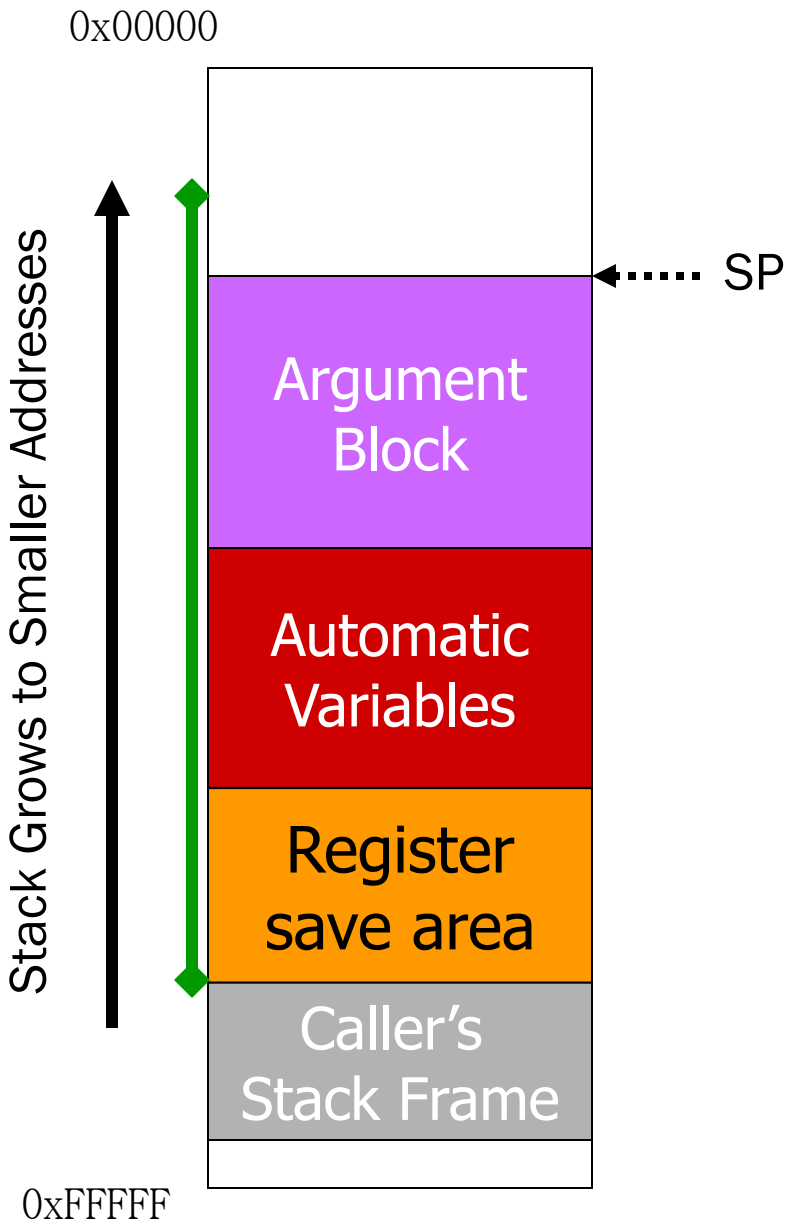
- A called function (child function) must perform the following tasks:
  - 1. If the function is declared with an ellipsis, it can be called with a variable number of arguments. The called function pushes these arguments on the stack if they meet both of these criteria:
    - The argument includes or follows the last explicitly declared argument.
    - The argument is passed in a register.
  - 2. The called function pushes register values of all the registers that are modified by the function and that must be preserved upon exit of the function onto the stack. Normally, these registers are the save-on-entry registers (R4-R10) if the function contains calls. If the function is an interrupt, additional registers may need to be preserved.
  - 3. The called function allocates memory for the local variables and argument block by subtracting a constant from the SP. This constant is computed with the following formula:

$$size\ of\ all\ local\ variables + max = constant$$

  - The *max* argument specifies the size of all parameters placed in the argument block for each call.
  - 4. The called function executes the code for the function.
  - 5. If the called function returns a value, it places the value in R12, R12 and R13, or R12 through R15, depending on the size of the return type.
  - 6. If the called function returns a structure, it copies the structure to the memory block that the first argument, R12, points to. If the caller does not use the return value, R12 is set to 0. This directs the called function not to copy the return structure.
  - For EABI, structures and unions with size 32 bits or less are passed by value, either in registers or on the stack. Structures and unions larger than 32 bits are passed by reference.
  - In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = func(x), where s is a structure and f is a function that returns a structure, the caller can simply pass the address of s as the first argument and call f. The function f then copies the return structure directly into s, performing the assignment automatically.
  - You must be careful to properly declare functions that return structures, both at the point where they are called (so the caller properly sets up the first argument) and at the point where they are declared (so the function knows to copy the result).
  - 7. The called function deallocates the frame and argument block by adding the constant computed in .
  - 8. The called function restores all registers saved in .
  - 9. The called function returns.

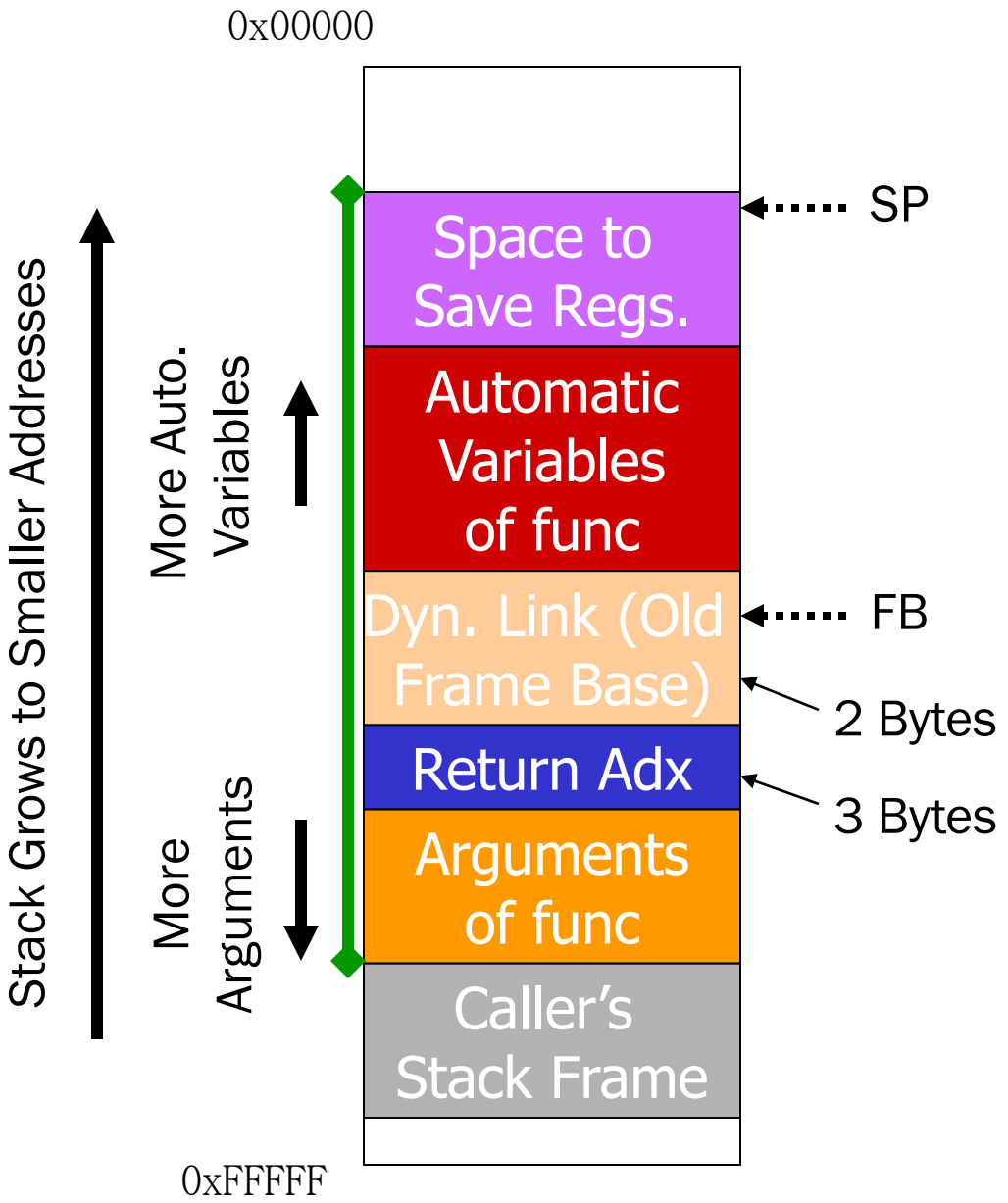
Activation Record / Stack Frame

- Register Save Area
- Called Functions Automatic Variables
- Called Functions Argument Block



Activation Record / Stack Frame

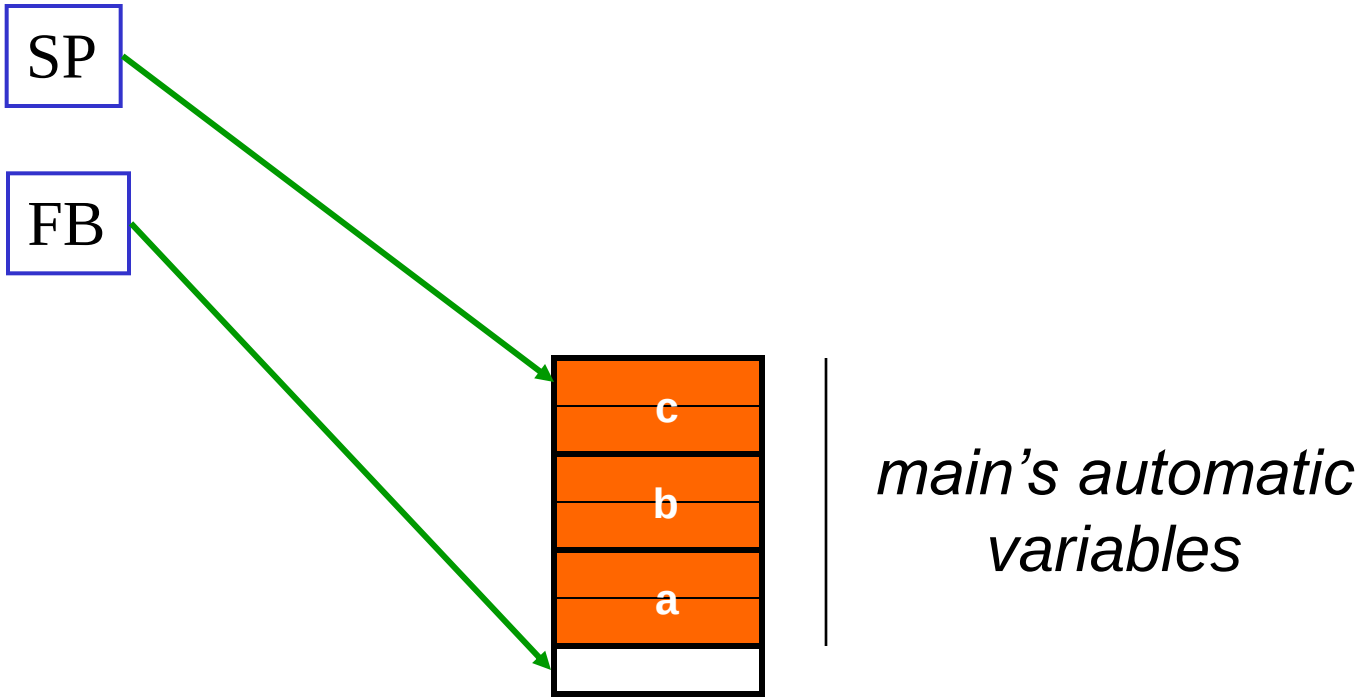
- Frame base == *dynamic link*
- 5 bytes used for
  - old frame base (0[FB], 1[FB])
  - return address (2[FB], 3[FB], 4[FB])
- *enter* and *exitd* instructions used to
  - modify, save and restore SP and FB
  - return from subroutine



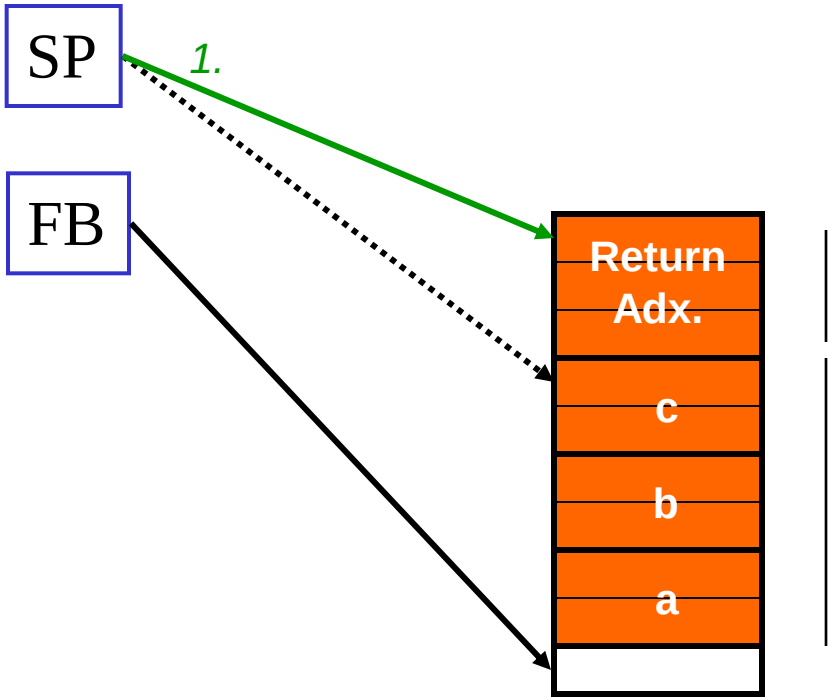
Example Program with Function Calls

```
const int globalD=6;
int compute(int x, int y);
int squared(int r);

void main() {
    int a, b, c;    // These are main's automatic variables, and will be
    a = 10;         // stored in main's frame
    b = 16;
    c = compute(a,b);
}
int compute(int x, int y) {
    int z;
    z = squared(x);
    z = z + squared(y) + globalD;
    return(z);
}

int squared(int r) {
    return (r*r);
}
```

Call Stack before main executes jsr compute



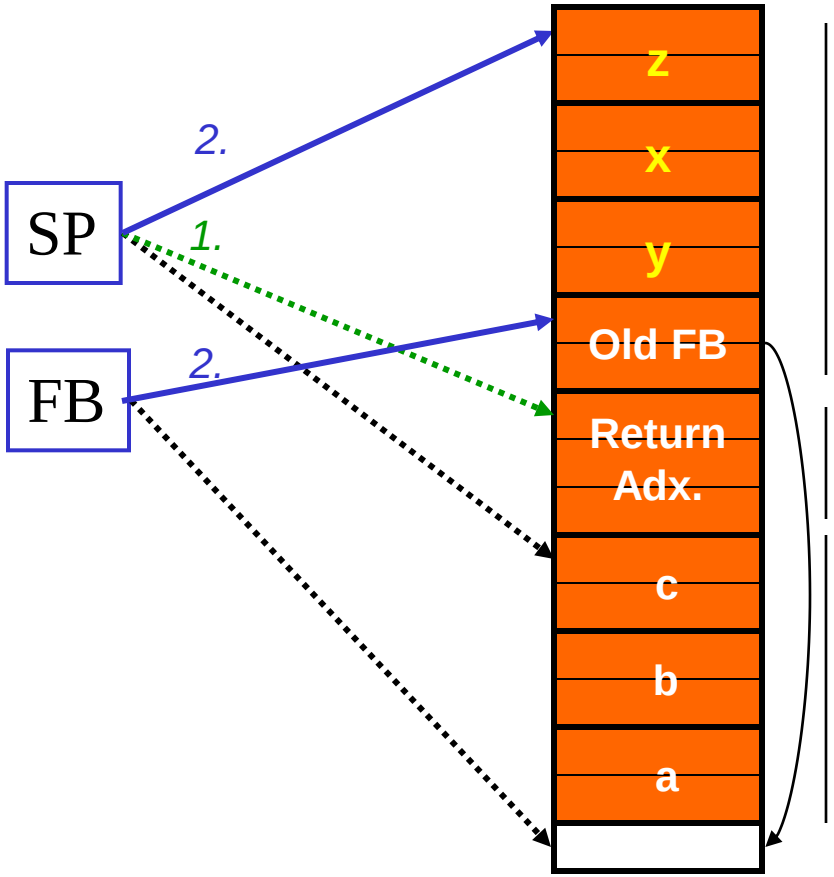
Call Stack arriving at compute



*1. **jsr** pushes old PC onto stack*

main's automatic variables

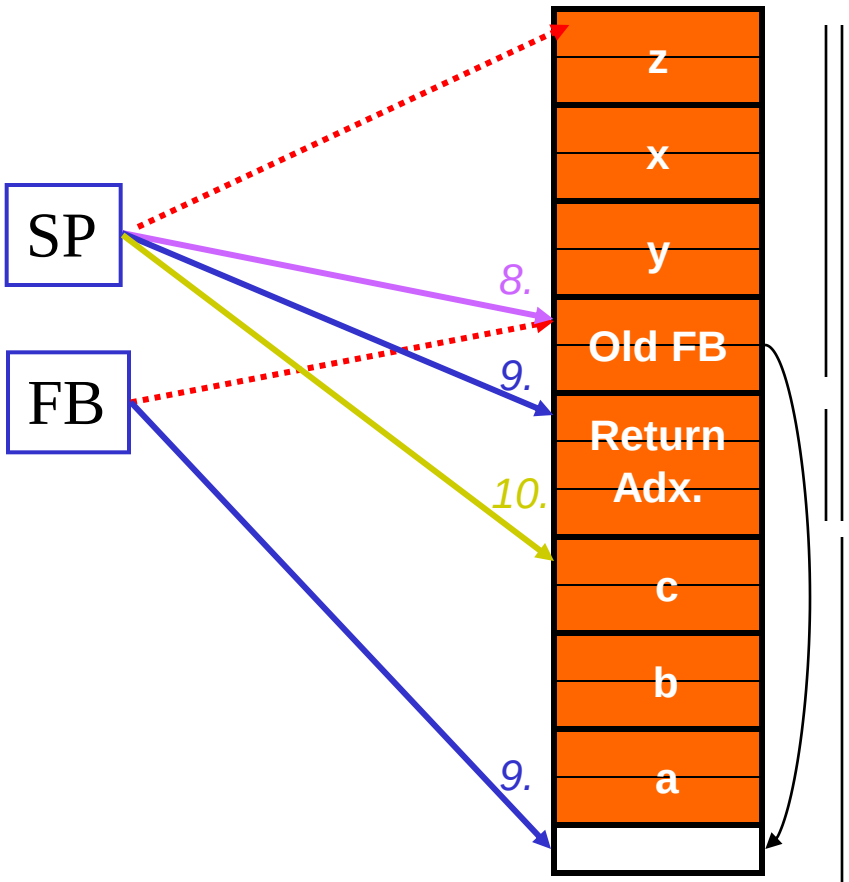
Call Stack executing enter #6 in compute



*2. **enter #6** pushes old FB value onto stack, copies SP to FB, and allocates 6 more bytes (for automatic variables x, y, z)*

main's automatic variables

Call Stack as compute executes exitd



*8. **exitd** copies FB to SP, deallocating space for x,y,z*

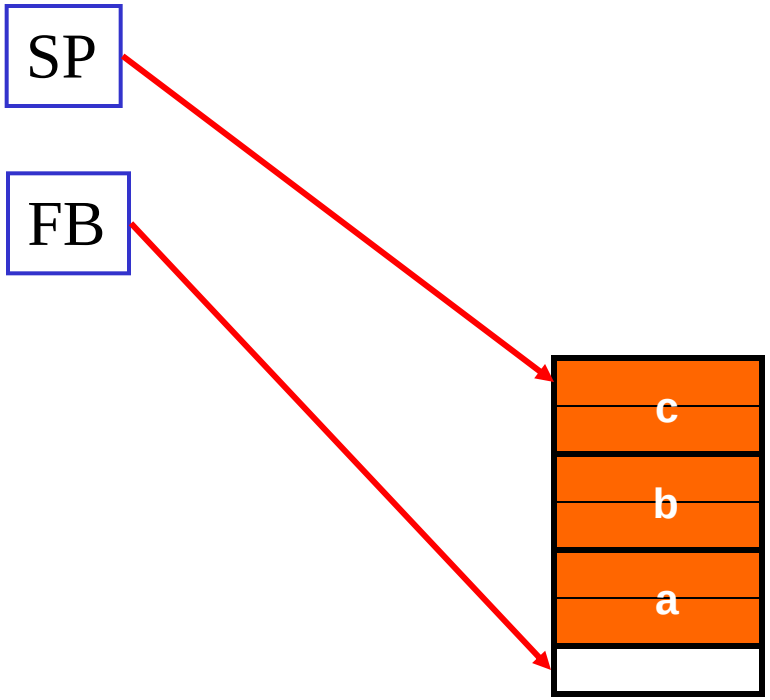
*9. **exitd** then pops FB*

*10. **exitd** pops the return address off the stack into the program counter*

compute's activation record

main's activation record

Call Stack as compute continues in main



main's activation record

Place Return Value in Proper Location

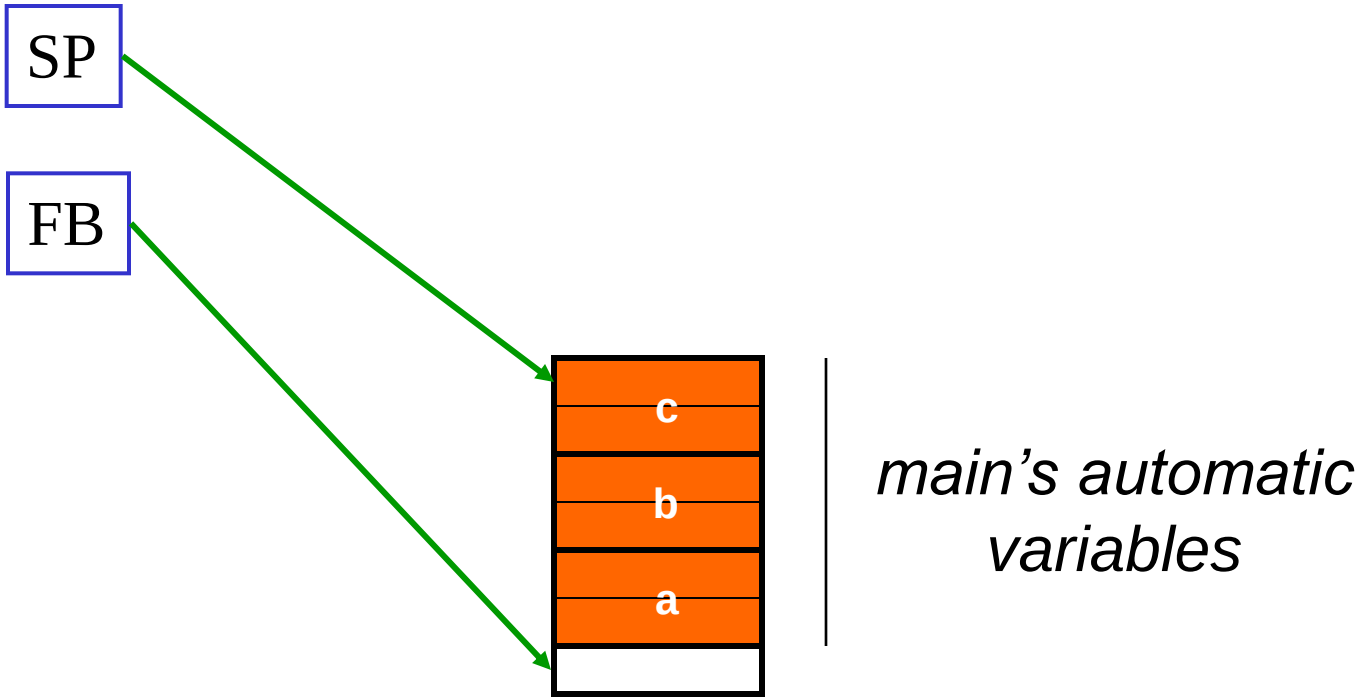
- If the called function returns a value, it places the value in R12, R12 and R13, or R12 through R15, depending on the size of the return type.

- If the called function returns a structure, it copies the structure to the memory block that the first argument, R12, points to. If the caller does not use the return value, R12 is set to 0. This directs the called function not to copy the return structure.

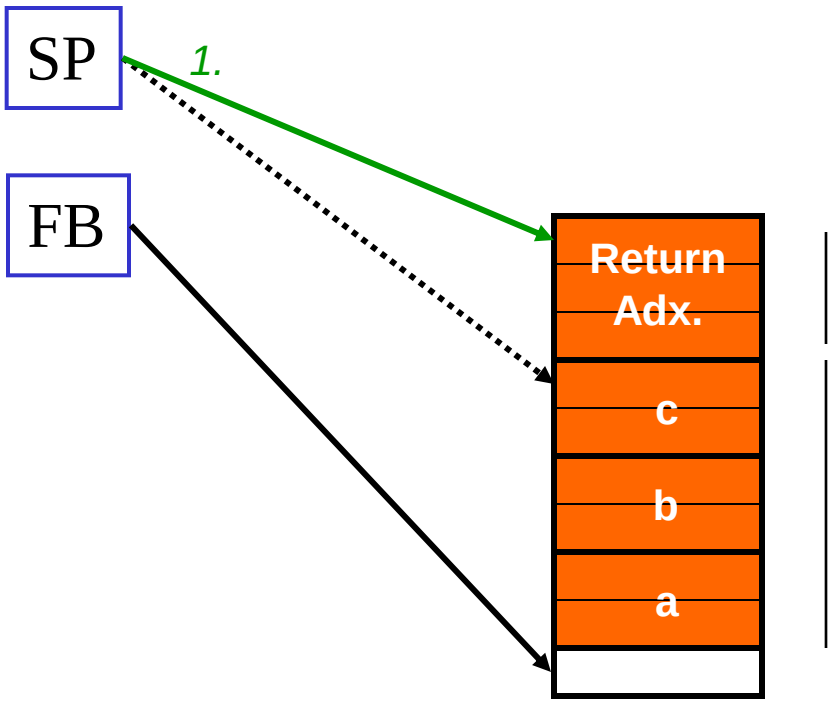
Putting It All Together

- Next we'll see how the stack grows and shrinks
  - *main* calls *compute*
  - *compute* calls *squared*
  - *squared* ends, returning control to *compute*
  - *compute* ends, returning control to *main*

Call Stack before main executes jsr compute



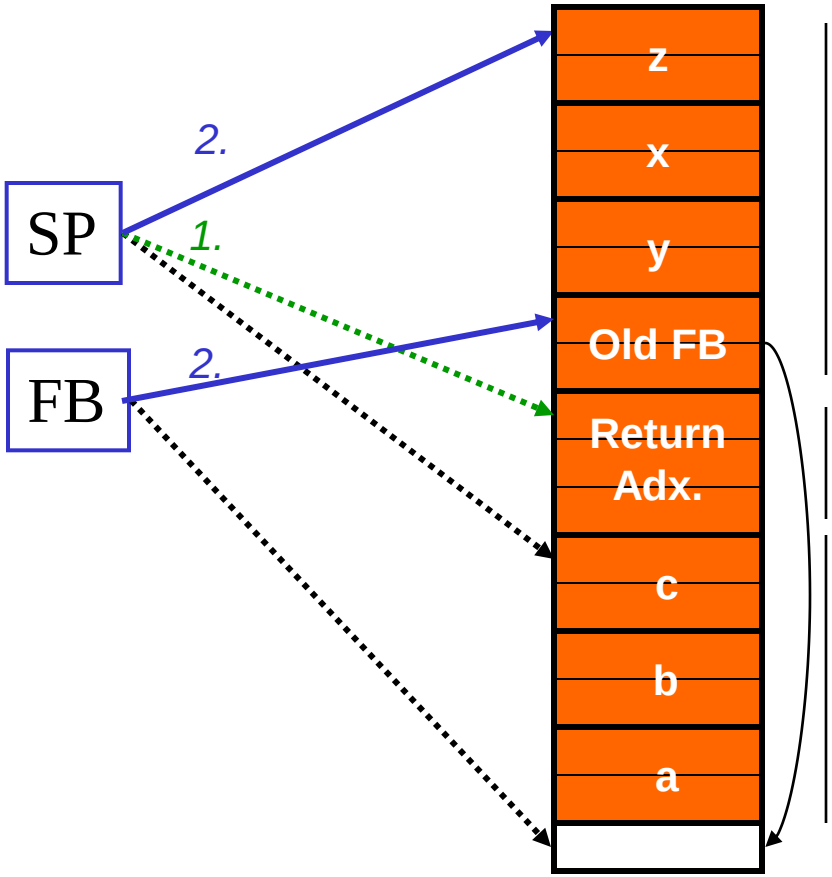
Call Stack arriving at compute



*1. **jsr** pushes old PC onto stack*

main's automatic variables

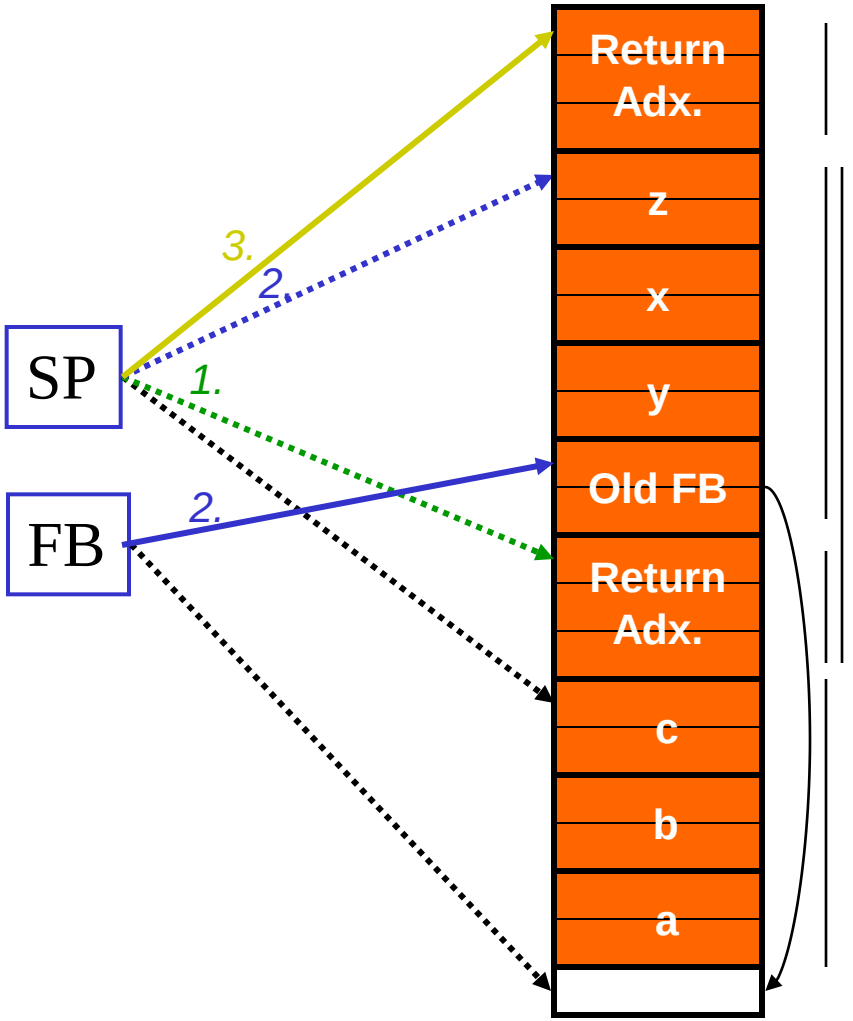
Call Stack executing enter #6 in compute



*2. **enter #6** pushes old FB value onto stack, copies SP to FB, and allocates 6 more bytes (for x, y, z)*

main's automatic variables

Call Stack as compute executes jsr squared

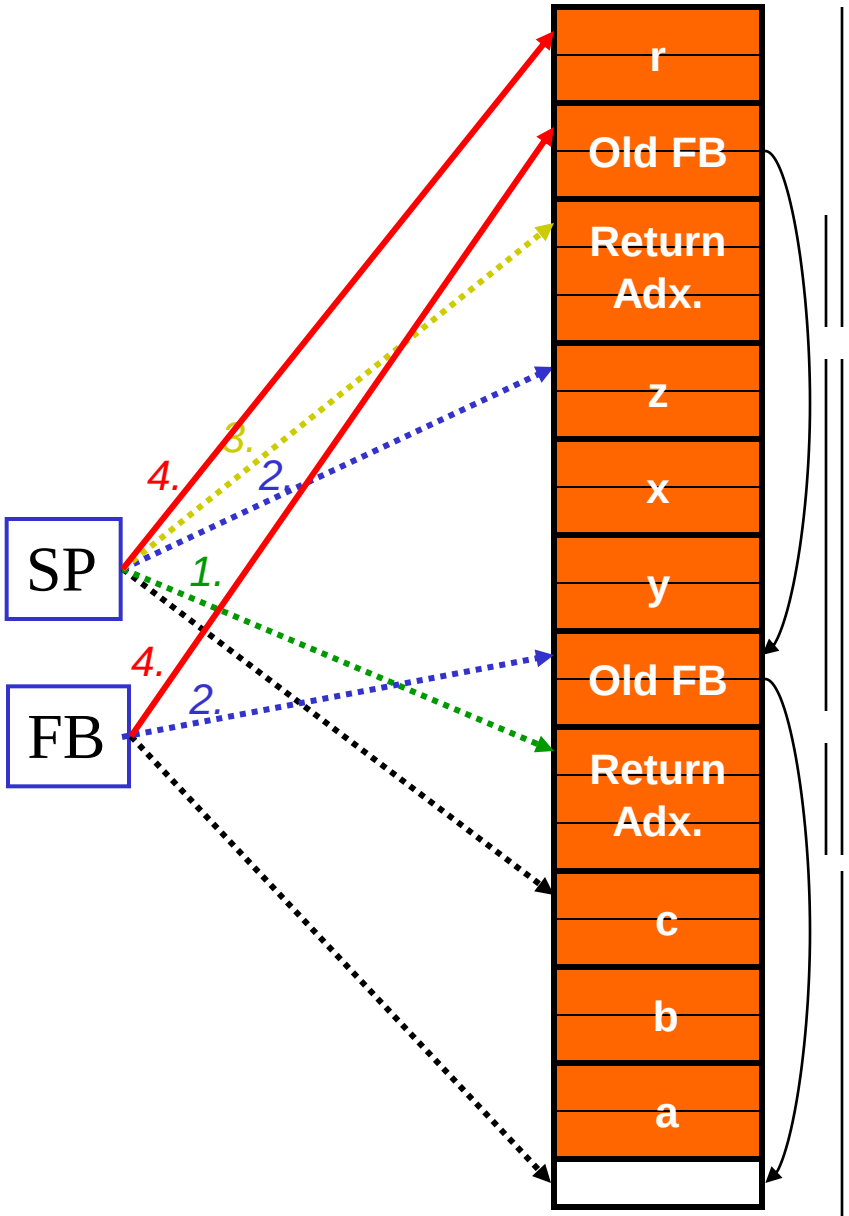


*3. **jsr** pushes old PC onto stack*

compute's activation record

main's activation record

Call Stack as squared executes enter #2

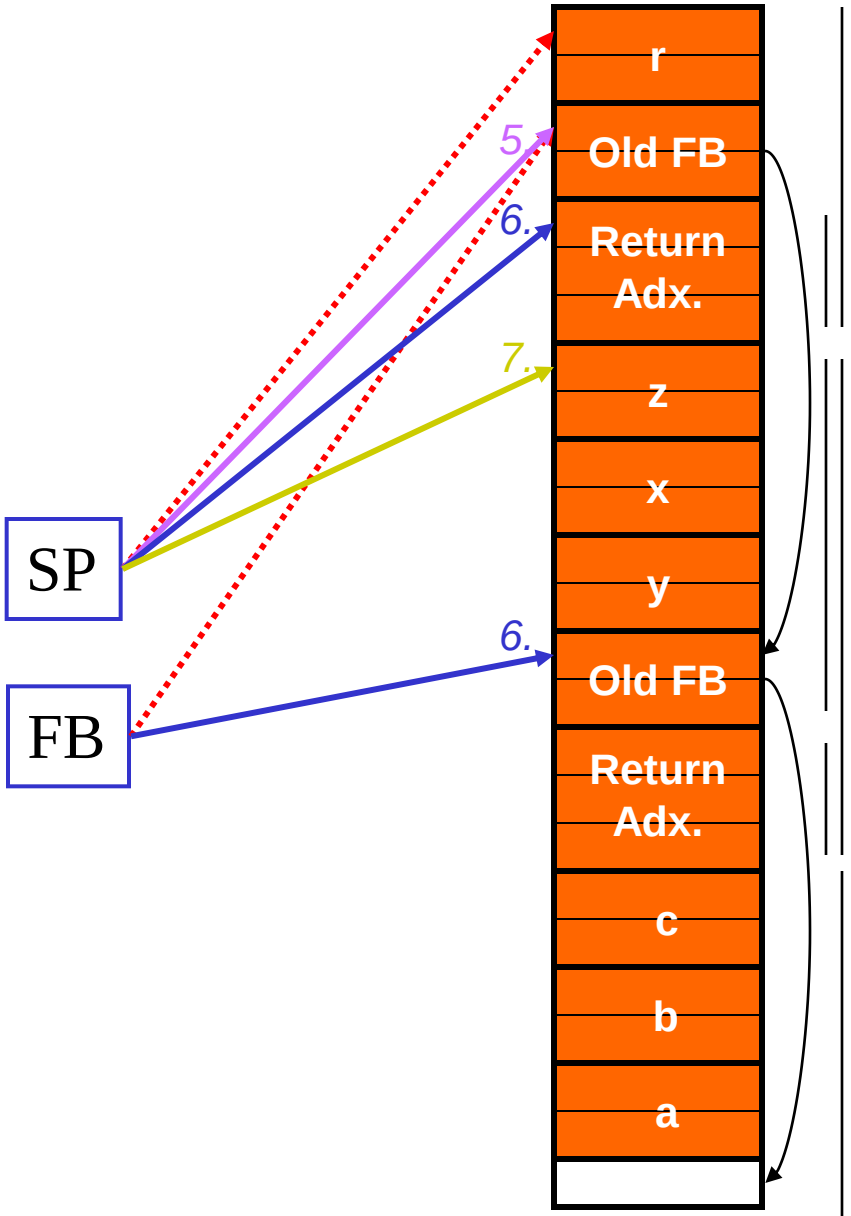


*4. **enter #2** pushes old FB value onto stack, copies SP to FB and allocates 2 more bytes (for r)*

compute's activation record

main's activation record

Call Stack as squared executes exitd



*5. **exitd** copies FB to SP, deallocating space for r*

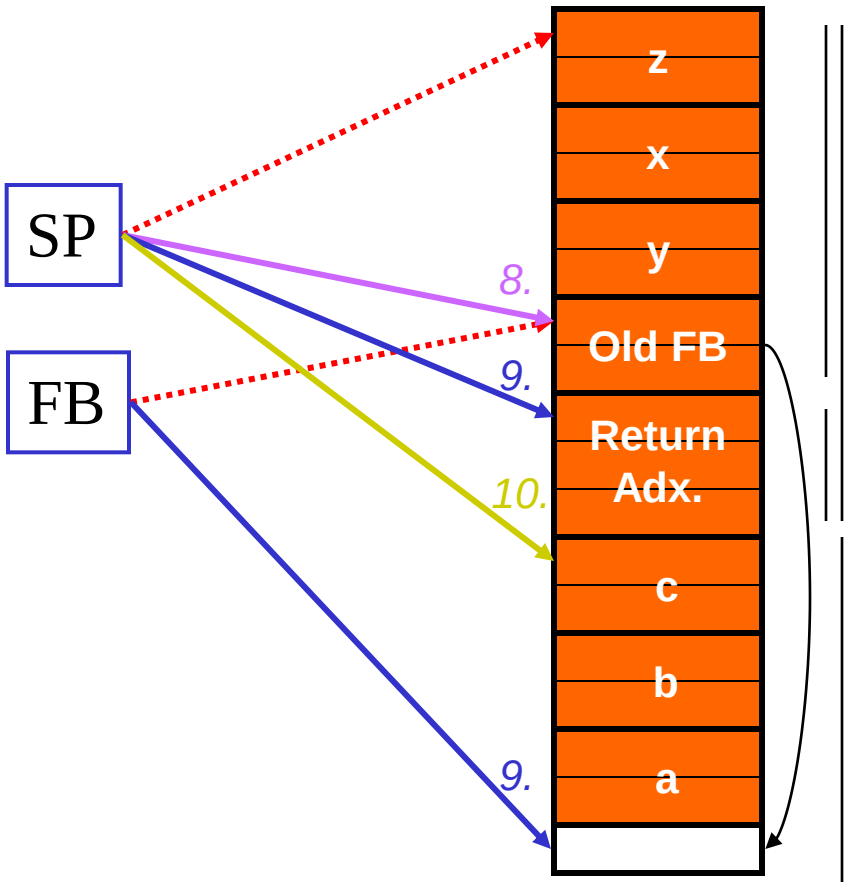
*6. **exitd** then pops FB*

*7. **exitd** pops the return address off the stack into the program counter*

compute's activation record

main's activation record

Call Stack as compute executes exitd



*8. **exitd** copies FB to SP, deallocating space for x,y,z*

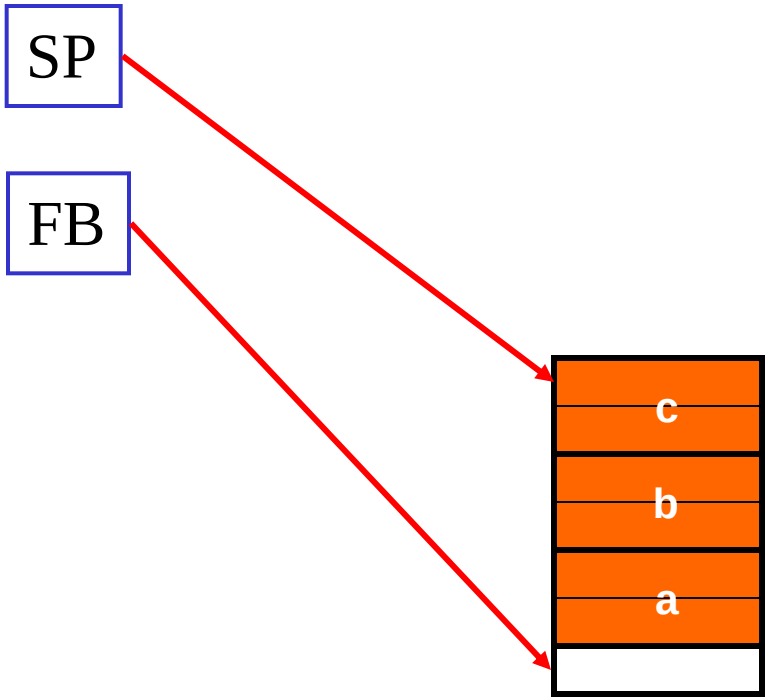
*9. **exitd** then pops FB*

*10. **exitd** pops the return address off the stack into the program counter*

compute's activation record

main's activation record

Call Stack as compute continues in main



main's activation record

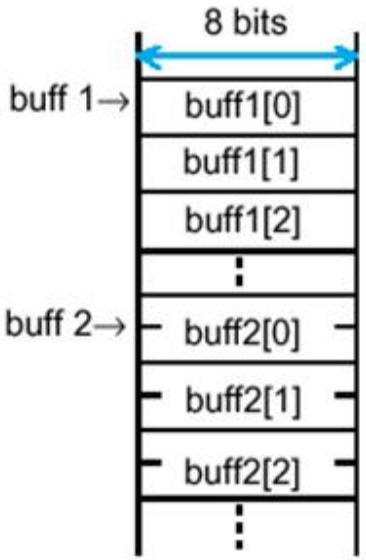
Notes for Actual Implementation by Compiler

- Desired Format:
  - offset[FB] variable_or_item_name
- Order in which local variables are located in activation record is cryptic
- Instead assume items are added the following order (with decreasing address)
  - Automatic variables, in order of declaration
    - One byte items
    - Two byte items
  - Space for arguments which have been passed by register, in order of declaration. Why? For local temporary storage, in case this function calls another function (it may need to use the registers for arguments)
    - One byte items
    - Two byte items
- Don’t forget the old frame pointer and return address

1D Arrays

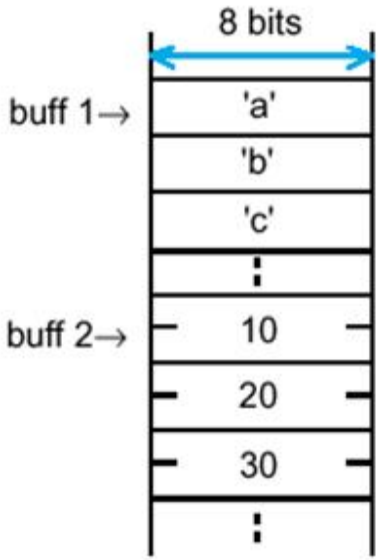
- Declaration of one-dimensional array

```
char  buff1[3] ;
int  buff2[3] ;
```

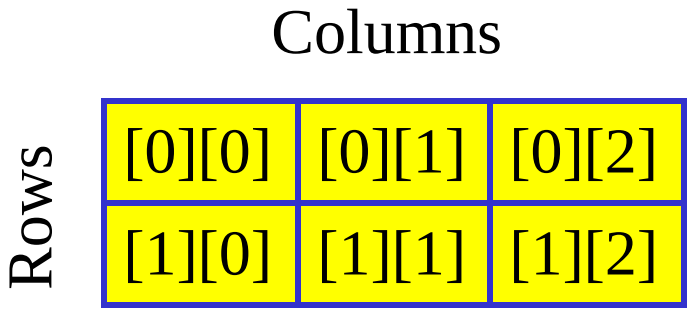


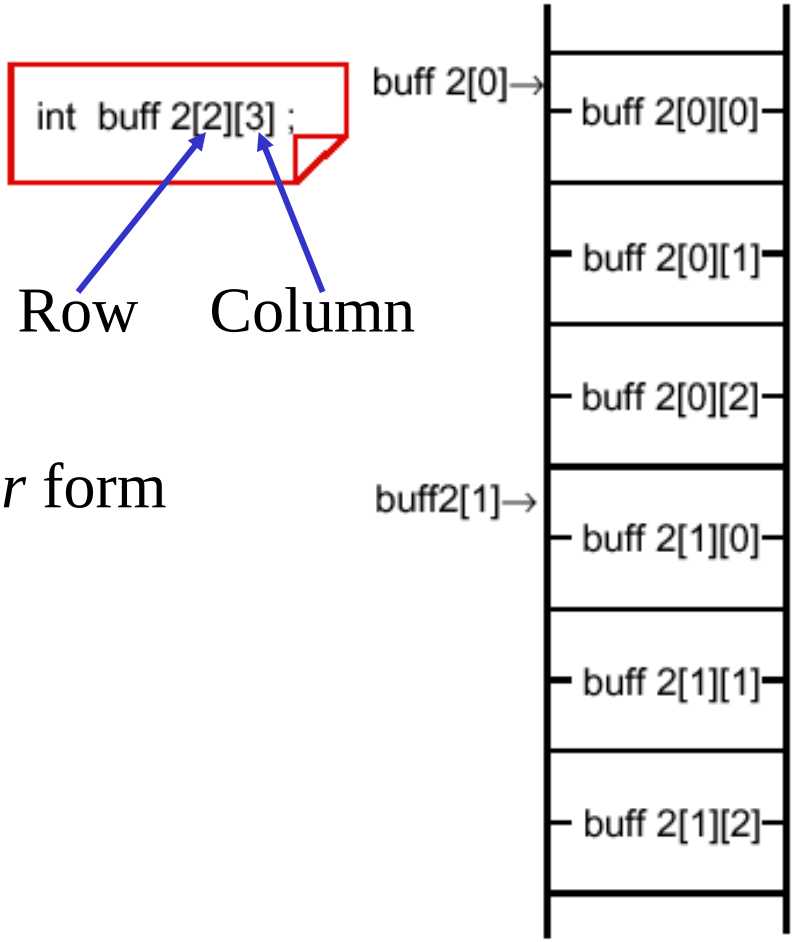
- Declaration and initialization of one-dimensional array

```
char  buff1[ ] = {
   'a' , 'b' , 'c'
} ;

int  buff2[ ] = {
   10 , 20 , 30
} ;
```



2D Arrays





- C arrays are stored in a *row-major* form (a row at a time)

Row-Major Array Layout