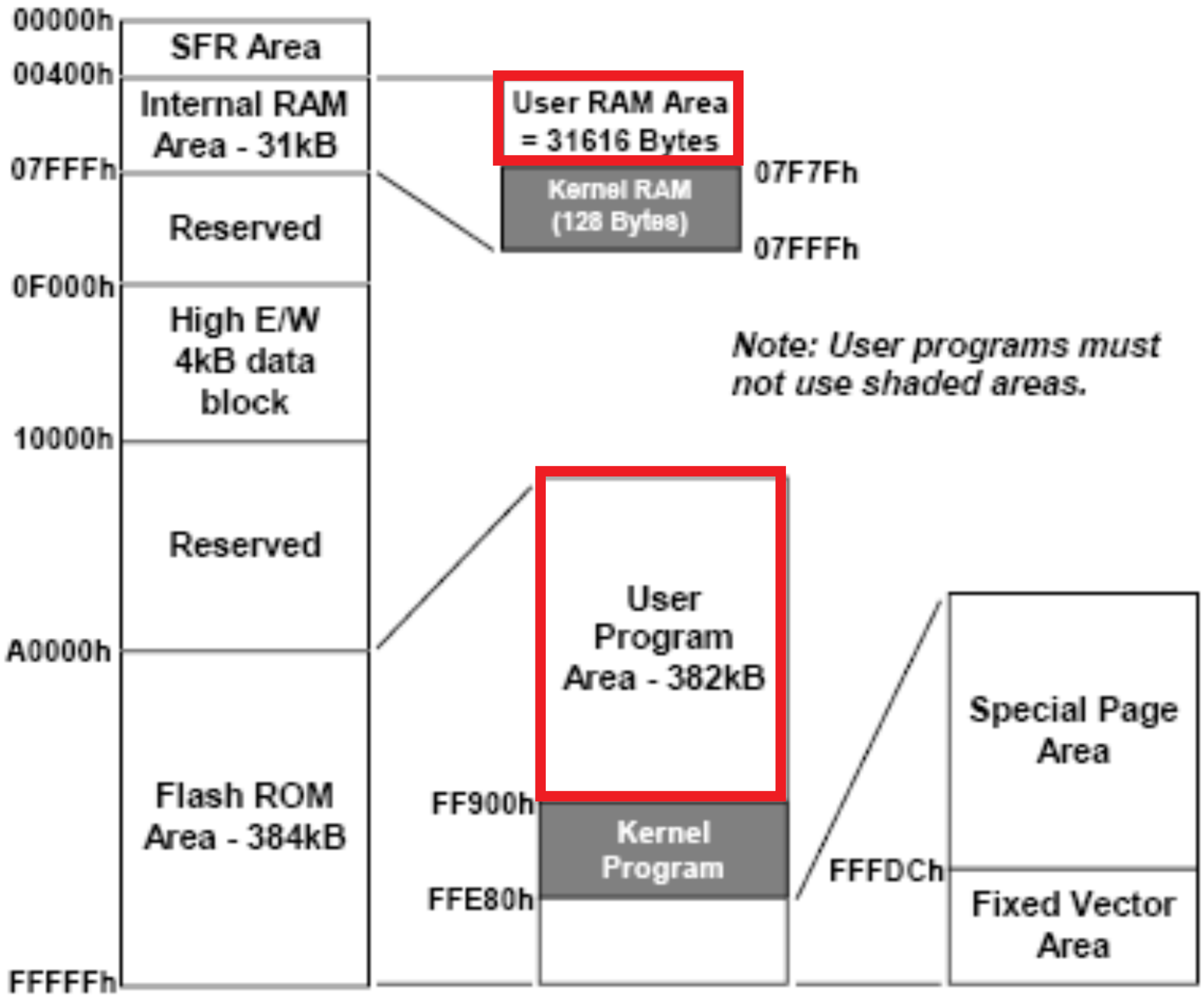
Stacks and Function Calls

Remember the Memory Map for Our MCU



Classifying Data

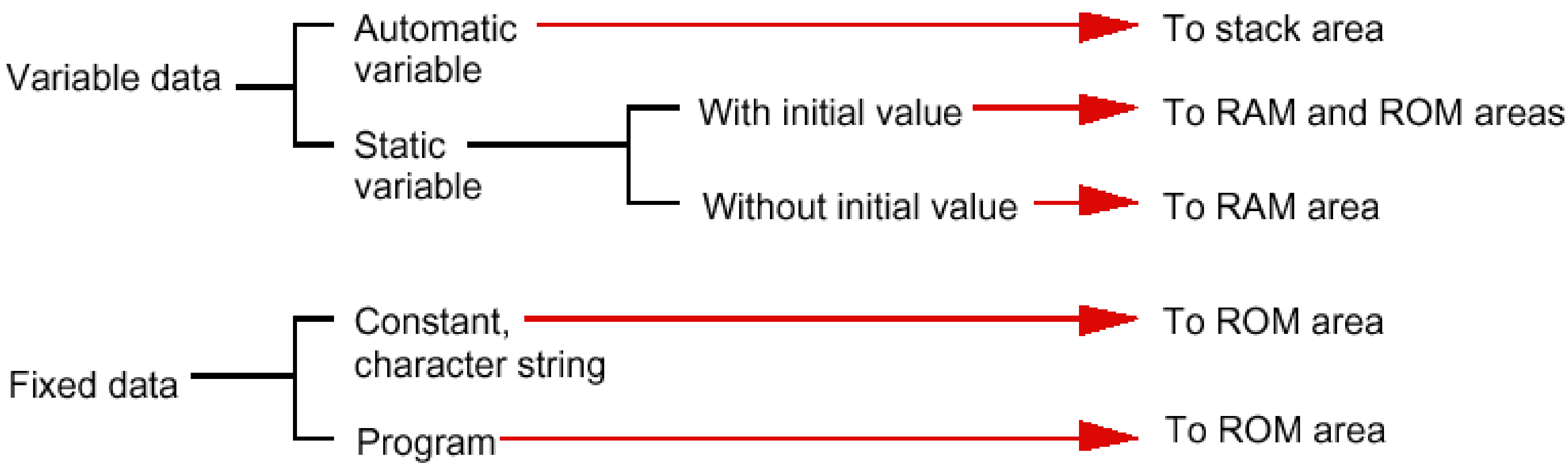


Figure 2.1.1 Types of data and code generated by NC30 and their mapped areas

Section Names and Contents

Table 2.1.1 Sections types Managed by NC30

Section base name	Content
data	Contains static variables with initial values.
bss	Contains static variables without initial values.
rom	Contains character strings and constants.
program	Contains programs.
vector	Variable vector area (compiler does not generate)
fvector	Fixed vector area (compiler does not generate)
stack	Stack area (compiler does not generate)
heap	Heap area (compiler does not generate)

"Block started by symbol"

Example of Sections

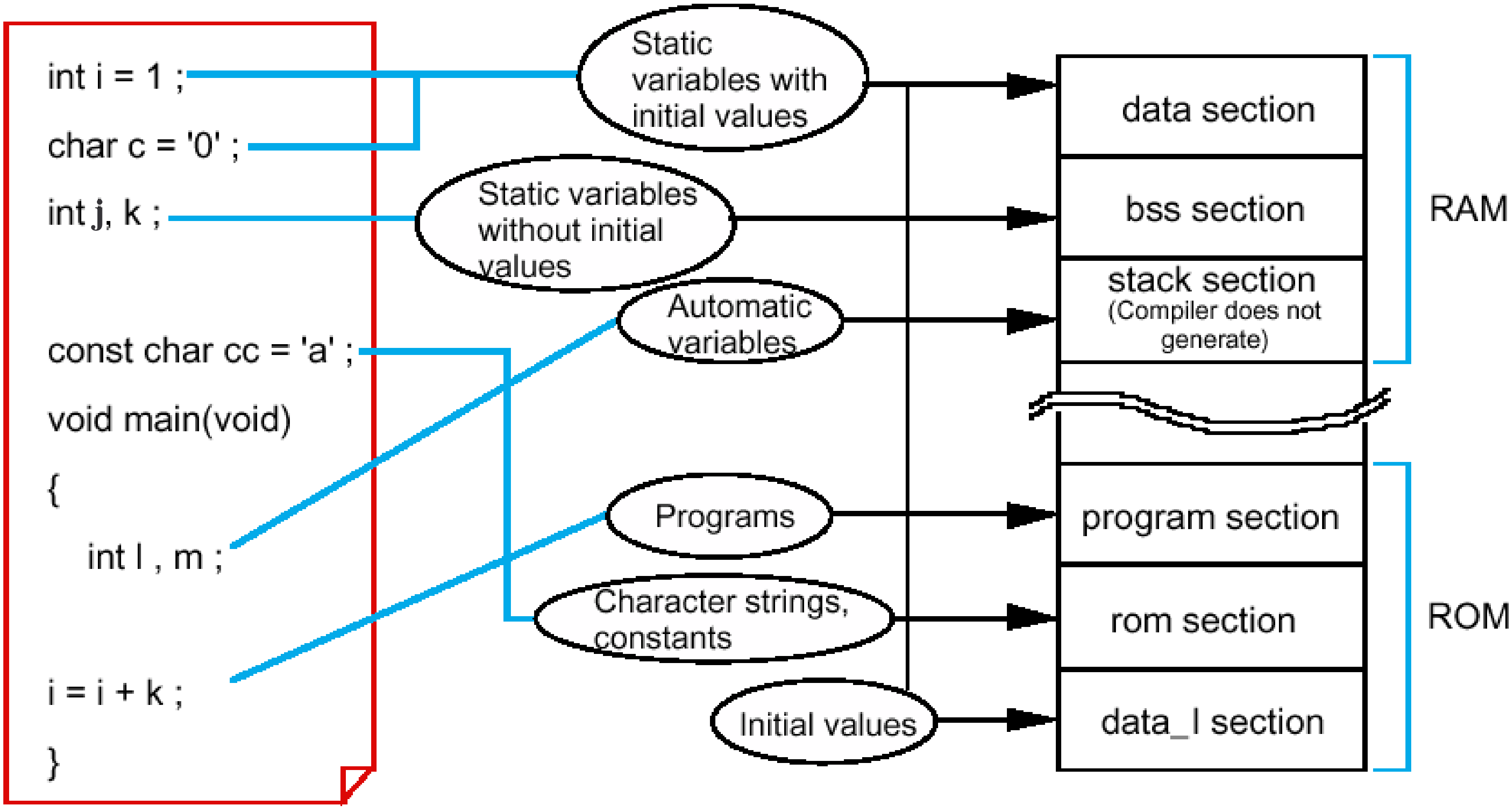


Figure 2.1.3 Mapping data into sections by type

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

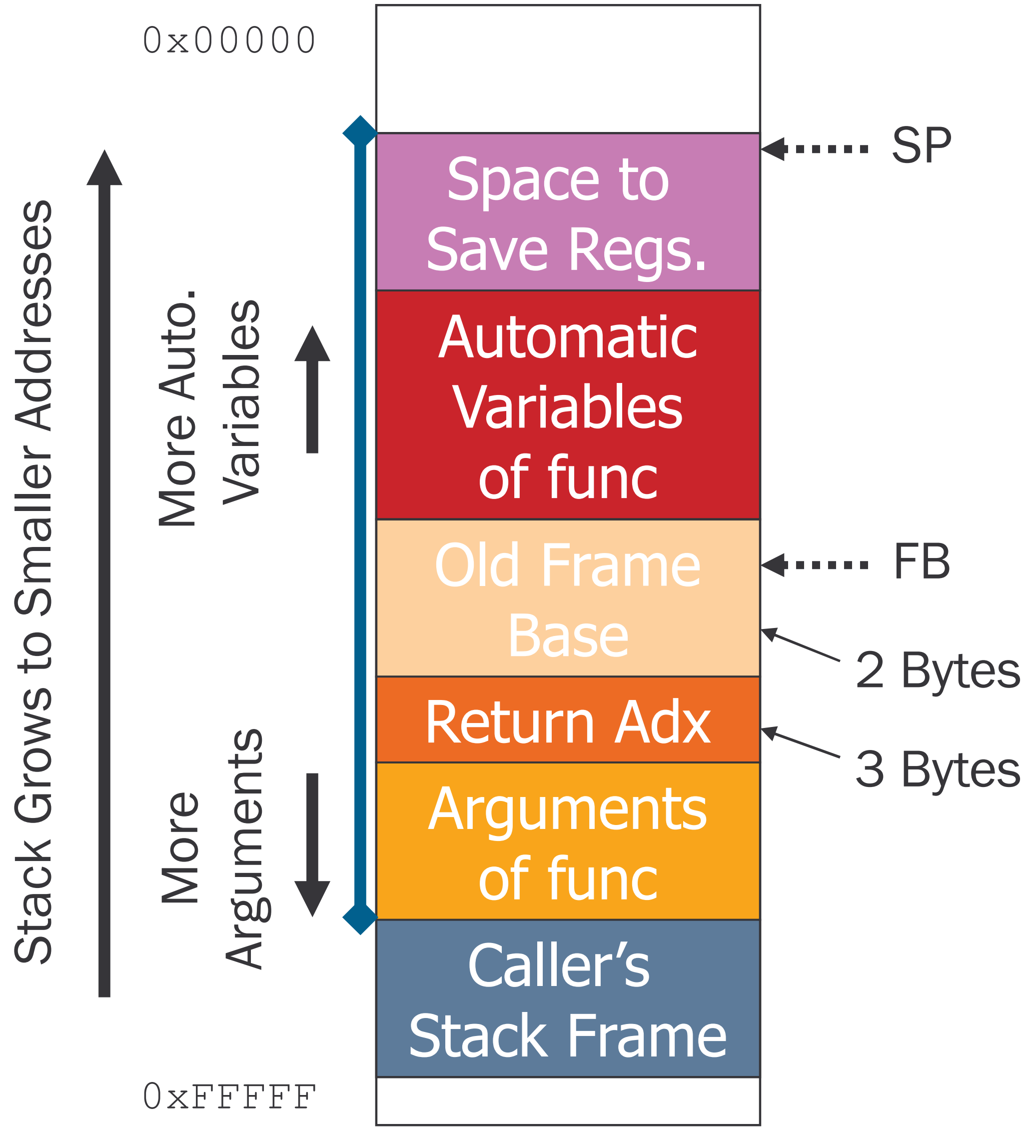
- arguments – data passed to function
- local variables
- return value
- other bookkeeping information

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

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);
}
```

Step 1 - Passing Arguments

Two methods for passing an argument to a function -- *through a register* or *on the stack*

- registers are preferred due to speed

Requirements for register passing

- Argument types are prototype declared
- At least one argument is of a type which can be assigned to a register
- Prototype declaration is complete

Arguments passed by register are allocated space in the stack for temporary use

Compiler keeps track of where arguments are located, so it knows which registers or memory the code should access to find the arguments

Table 2.4.1 Rules for Passing Arguments

Type of argument	First argument	Second argument	Third and following arguments
char type	R1L	Stack	Stack
short, int types near pointer type	R1	R2	Stack
Other types	Stack	Stack	Stack

Example of Step 1

```
void main() {
    int a, b, c;      // These variable are local to main,
    and will be
    a = 10;           // stored in main's frame
    b = 16;
    c = compute(a,b);
}

_main:
    enter         #06H
    mov.w         #000aH,-2[FB]  ;  a
    mov.w         #0010H,-4[FB]  ;  b
    mov.w         -4[FB],R2      ;  b
    mov.w         -2[FB],R1      ;  a
    jsr           $compute
    mov.w         R0,-6[FB]      ;  c
    exitd
```

Move b (arg 2) into R2
Move a (arg 1) into R1

Example of Step 2

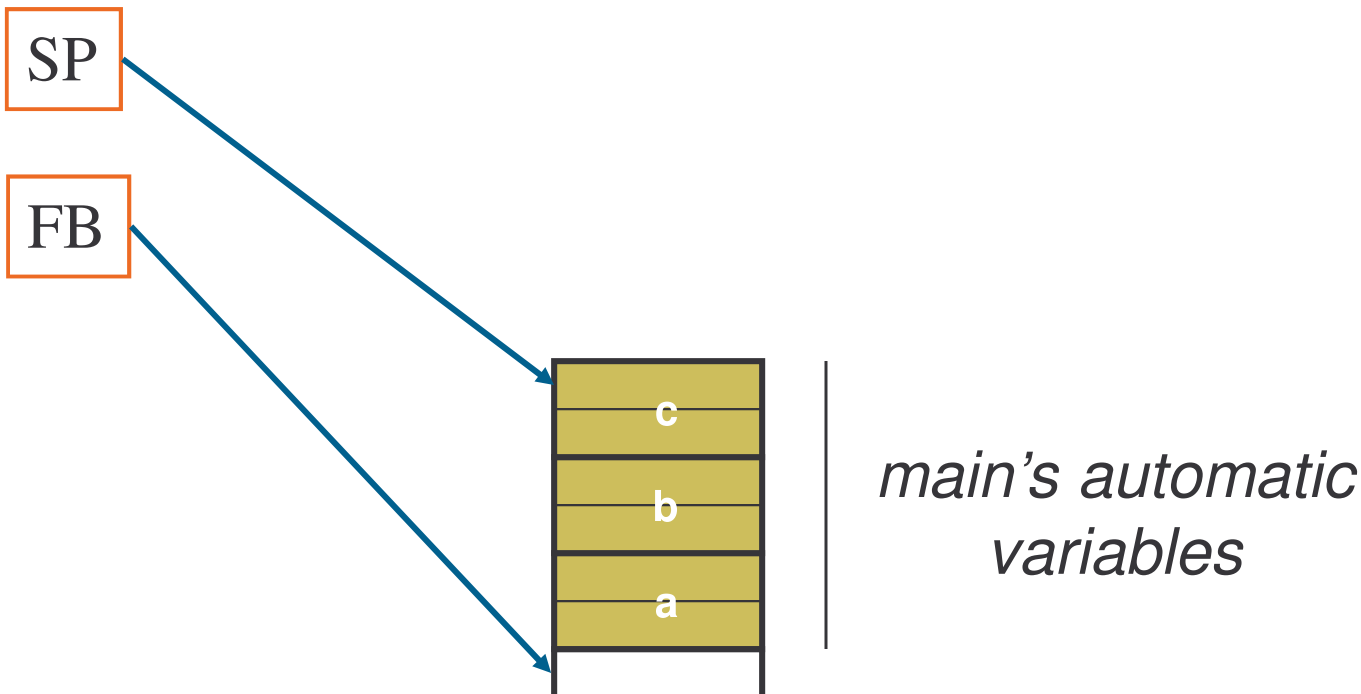
```
void main() {
    int a, b, c;    // These variable are local to
    main, and will be
    a = 10;         // stored in main's frame
    b = 16;
    c = compute(a,b);
}

_main:
    enter       #06H
    mov.w       #000aH,-2[FB]  ;  a
    mov.w       #0010H,-4[FB]  ;  b
    mov.w       -4[FB],R2      ;  b
    mov.w       -2[FB],R1      ;  a
    jsr         $compute
    mov.w       R0,-6[FB]      ;  c
    exitd
```

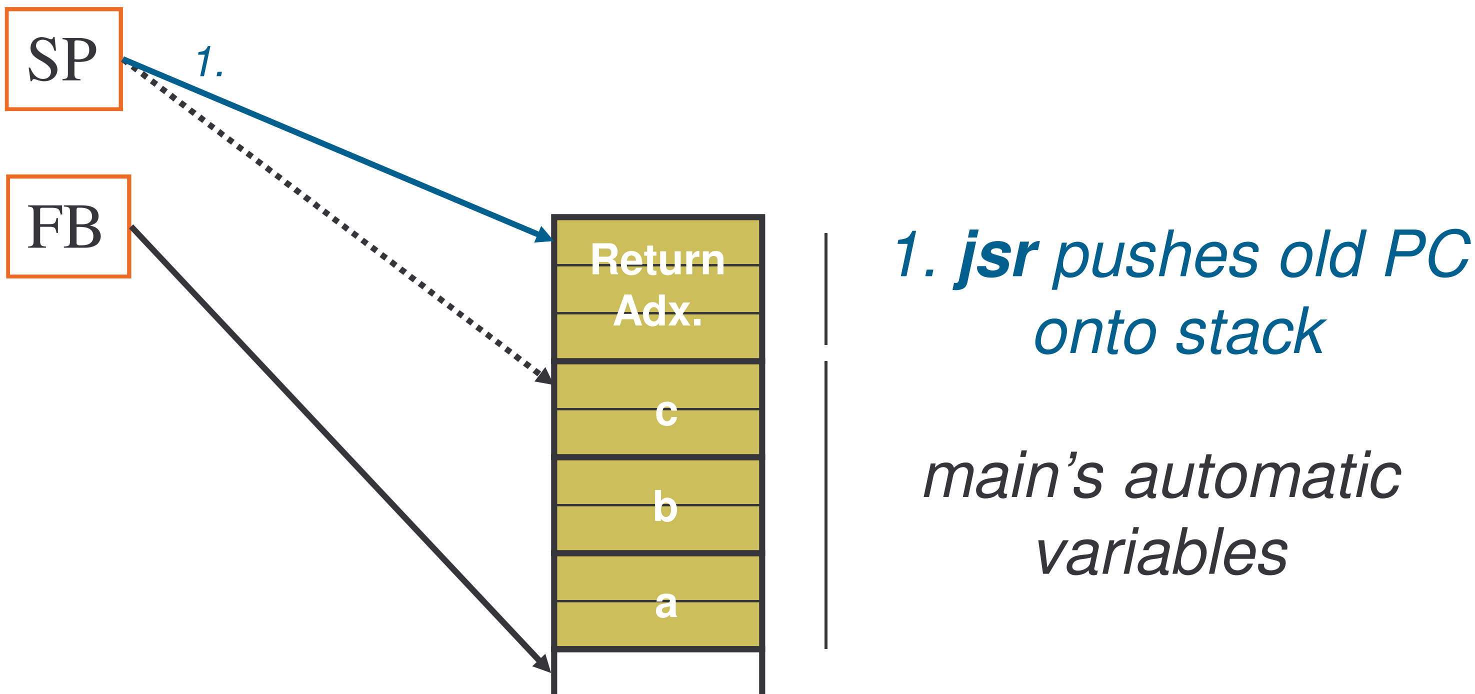
Jump to subroutine

1. *push address of next instruction (mov.w R0,-6[FB]) onto call stack*
2. *load PC with address $compute*

Call Stack before main executes jsr compute



Call Stack arriving at compute



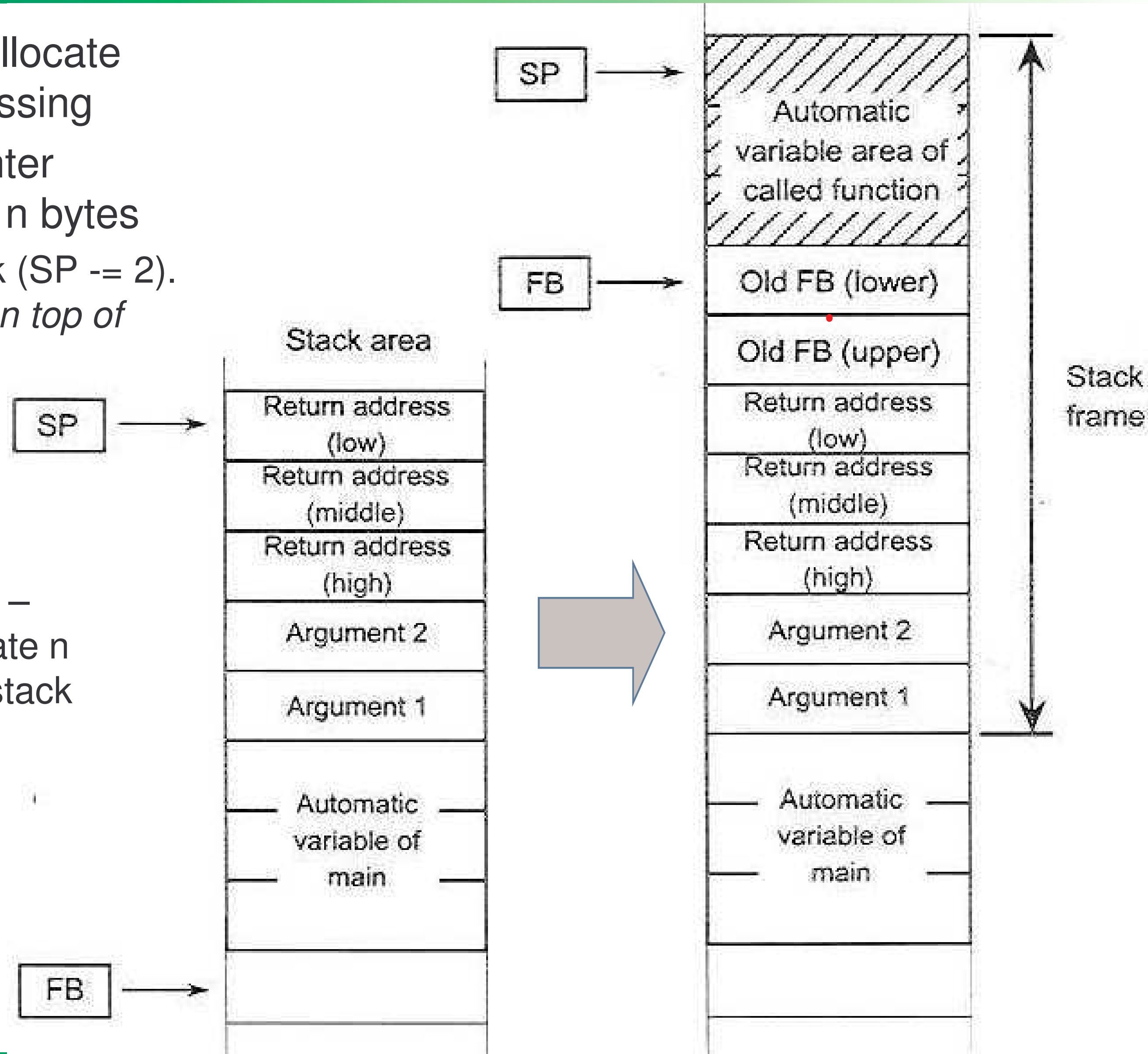
Step 3 – Allocate space on stack

Save dynamic link and allocate space for local processing

Enter #n instruction – Enter function and allocate n bytes

- Push FB onto stack (SP -= 2). *SP points to data on top of stack.*
- Copy SP to FB (make FB point to where SP currently points)
- Subtract n from SP – automatically allocate n bytes of space on stack

MALPM, pp. 81-82



Example of Steps 3 & 4

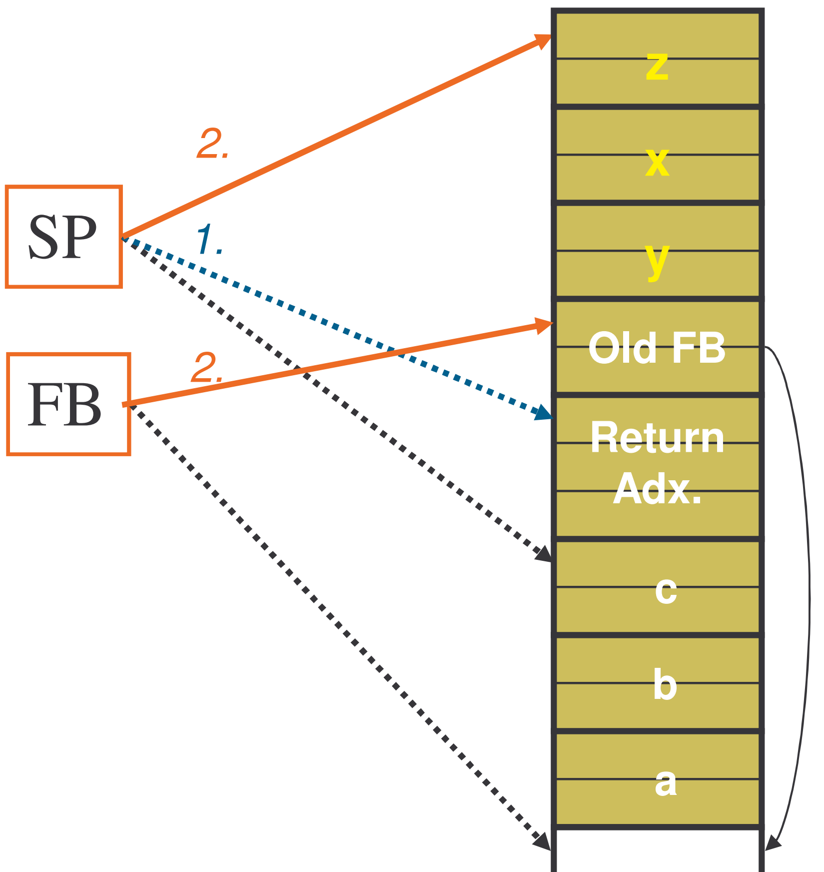
```
E1:
;## #     FUNCTION compute
;## #     FRAME  AUTO     (z)      size   2,  offset -6
;## #     FRAME  AUTO     (y)      size   2,  offset -2
;## #     FRAME  AUTO     (x)      size   2,  offset -4
;## #     REGISTER ARG (x)         size   2,  REGISTER R1
;## #     REGISTER ARG (y)         size   2,  REGISTER R2

    .glb   $compute
$compute:
    enter #06H
    mov.w          R1,-4[FB]       ;  x  x
    mov.w          R2,-2[FB]       ;  y  y
    mov.w          -4[FB],R1       ;  x
    jsr    $squared
    mov.w          R0,-6[FB]       ;  z
    mov.w          -2[FB],R1       ;  y
    jsr    $squared
    add.w          -6[FB],R0       ;  z
    add.w          _globalD,R0
    mov.w          R0,-6[FB]       ;  z
    mov.w          -6[FB],R0       ;  z
    exitd
```

Step 3. Save dynamic link, Allocate 6 bytes of space on stack for 3 local variables

Step 4. Execute body of function

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

Step 5 – Place Return Value in Proper Location

Some functions return a value – integer, pointer, structure, etc.

If not a struct or union, pass back in register for speed

Table 2.4.2 Rules for Passing Return Value

Data type	Returning method
char	R0L
int short	R0
long float	R2R0
double	R3R2R1R0
near pointer	R0
far pointer	R2R0
struct union	Store address is passed via a stack

If a struct or union, pass back on stack. Pointer to space on stack is provided (and space is allocated) when function is called

Example of Step 5

```
E1:
;## #     FUNCTION compute
;## #     FRAME  AUTO     (z)      size  2,  offset -6
;## #     FRAME  AUTO     (y)      size  2,  offset -2
;## #     FRAME  AUTO     (x)      size  2,  offset -4
;## #     REGISTER ARG  (x)      size  2,  REGISTER R1
;## #     REGISTER ARG  (y)      size  2,  REGISTER R2

    .glb   $compute
$compute:
    enter #06H
    mov.w          R1,-4[FB]       ;  x  x
    mov.w          R2,-2[FB]       ;  y  y
    mov.w          -4[FB],R1       ;  x
    jsr     $squared
    mov.w          R0,-6[FB]       ;  z
    mov.w          -2[FB],R1       ;  y
    jsr     $squared
    add.w          -6[FB],R0       ;  z
    add.w          _globalD,R0
    mov.w          R0,-6[FB]       ;  z
    mov.w          -6[FB],R0       ;  z
    exitd
```

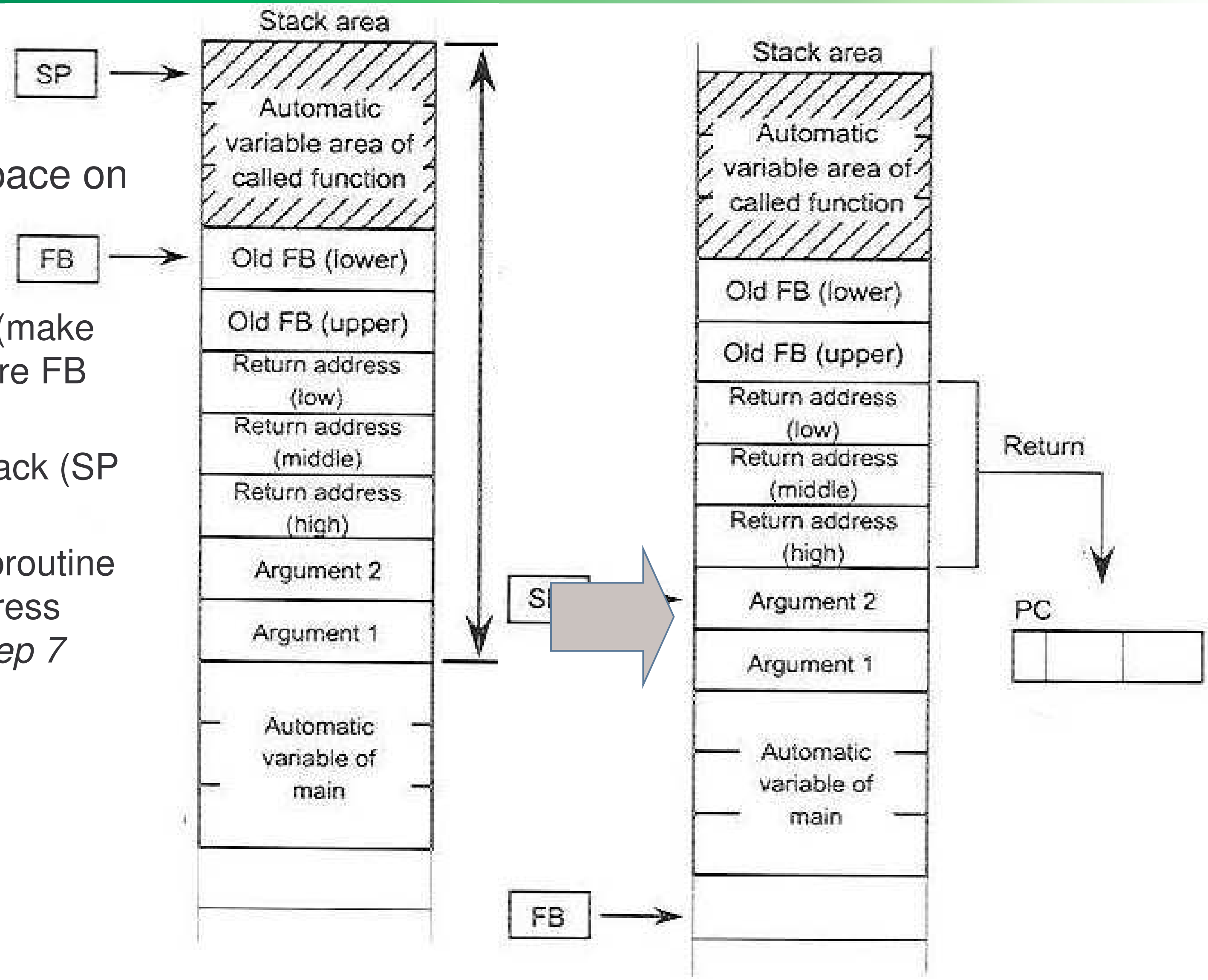
Step 5. Compute returns an int, so copy result to R0

Steps 6 and 7 – Deallocate Space and Return

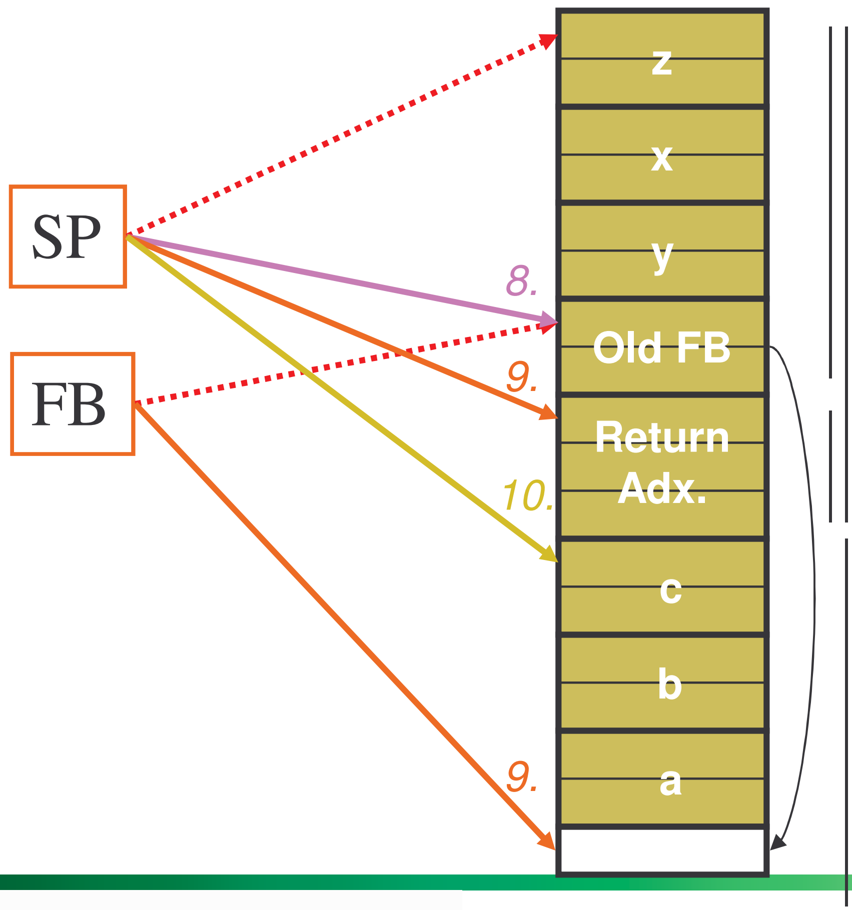
Exitd – Deallocate space on stack and exit function

- Copy FB to SP (make SP point to where FB currently points)
- Pop FB from Stack (SP += 2)
- Return from subroutine (pop return address from stack) – *step 7*

MALPM, pp. 81-82



Call Stack as compute executes exitd



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

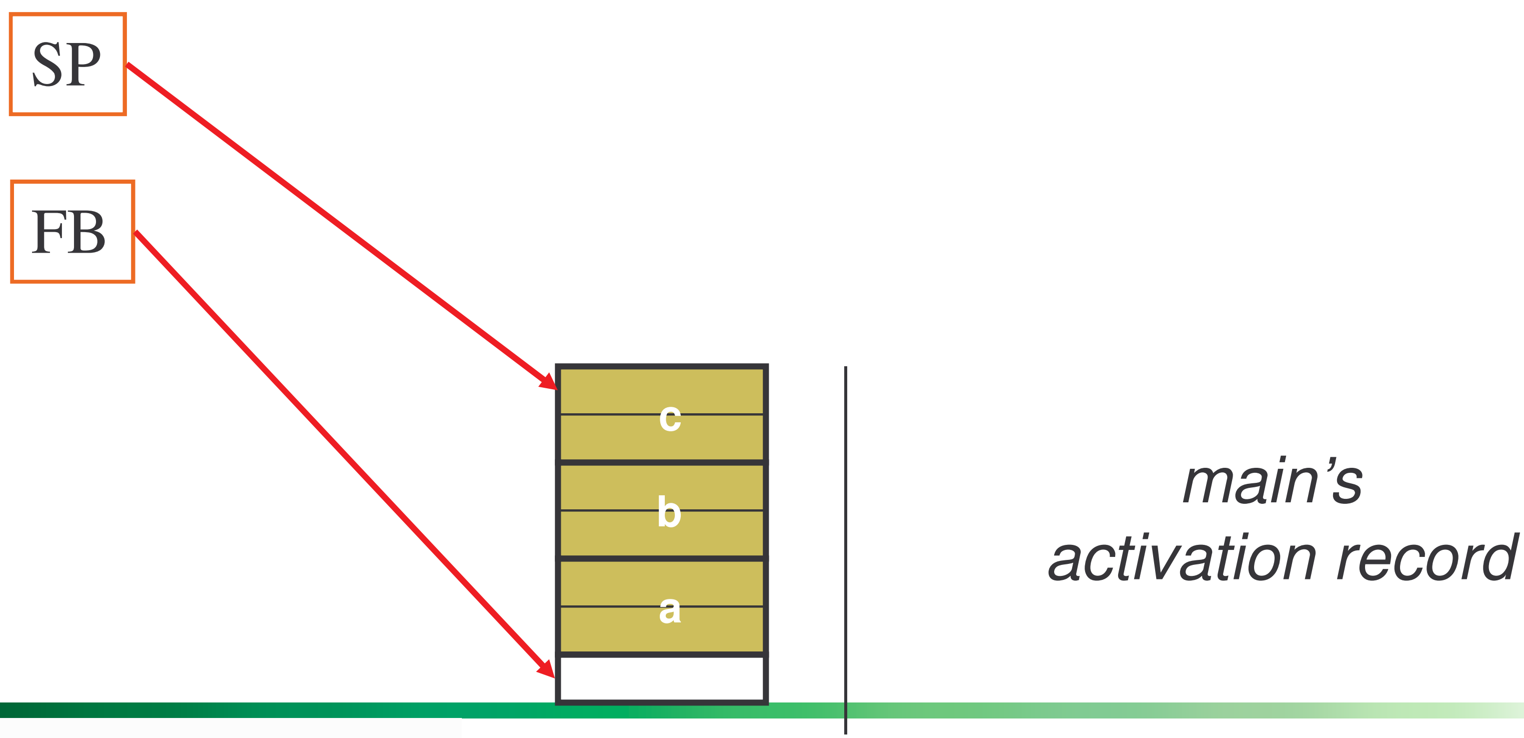
9. ***exitd*** *then pops FB*

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



Activation Record – Auto Vars and Arguments

```
int init2(long ss, char * a, int n, char j, int lala) {
  int i, i2, i3, sum = 0;
  char mc = 'r', nc = 'j';

;## #   FRAME   AUTO    (       i2)     size    2,      offset -12
;## #   FRAME   AUTO    (       i3)     size    2,      offset -10
;## #   FRAME   AUTO    (        i)     size    2,      offset -8
;## #   FRAME   AUTO    (      sum)     size    2,      offset -6
;## #   FRAME   AUTO    (        a)     size    2,      offset -4
;## #   FRAME   AUTO    (       nc)     size    1,      offset -2
;## #   FRAME   AUTO    (       mc)     size    1,      offset -1
;## #   FRAME   ARG     (       ss)     size    4,      offset 5
;## #   FRAME   ARG     (        n)     size    2,      offset 9
;## #   FRAME   ARG     (        j)     size    1,      offset 11
;## #   FRAME   ARG     (     lala)     size    2,      offset 12
;## #   REGISTER ARG    (        a)     size    2,      REGISTER R2
;## #   ARG Size(9)     Auto Size(12)   Context Size(5)
```

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*

Main() Function

```
;## #      FUNCTION main
;## #      FRAME  AUTO    (c)        size   2,  offset -6
;## #      FRAME  AUTO    (b)        size   2,  offset -4
;## #      FRAME  AUTO    (a)        size   2,  offset -2

    .section        program,align
_main:
    enter #06H
    mov.w           #000aH,-2[FB]     ;  a
    mov.w           #0010H,-4[FB]     ;  b
    mov.w           -4[FB],R2         ;  b
    mov.w           -2[FB],R1         ;  a
    jsr             $compute
    mov.w           R0,-6[FB]         ;  c
    exitd
```

To get the assembly language output file:
-In HEW, select **Options->Renesas M16C standard toolchain …**
-Exand the **C source file** option under your project name
-Highlight the name of the C file (or default)
-On the **C** tab, set **Category** to **Object**
-Set **Output file type** to **[-S] Assembly language source file (*.a30)**
-Select **OK**
-Build the project (Build all)

Compute() Function

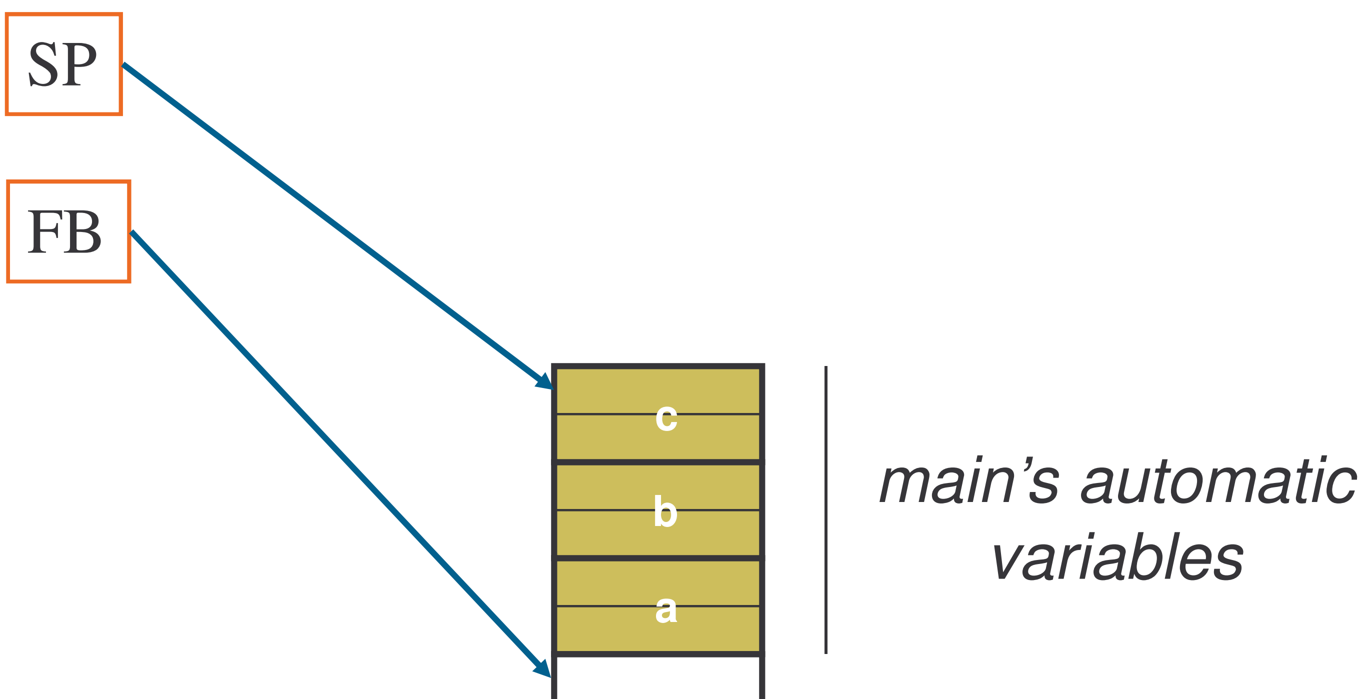
```
;## #     FUNCTION compute
;## #     FRAME  AUTO     (z)        size   2,  offset -6
;## #     FRAME  AUTO     (y)        size   2,  offset -2
;## #     FRAME  AUTO     (x)        size   2,  offset -4
;## #     REGISTER ARG  (x)        size   2,  REGISTER R1
;## #     REGISTER ARG  (y)        size   2,  REGISTER R2
    .glb   $compute
$compute:
    enter #06H
    mov.w          R1,-4[FB]          ;  x  x
    mov.w          R2,-2[FB]          ;  y  y
    mov.w          -4[FB],R1          ;  x
    jsr     $squared
    mov.w          R0,-6[FB]          ;  z
    mov.w          -2[FB],R1          ;  y
    jsr     $squared
    add.w          -6[FB],R0          ;  z
    add.w          _globalD,R0
    mov.w          R0,-6[FB]          ;  z
    mov.w          -6[FB],R0          ;  z
    exitd
```

Squared() Function

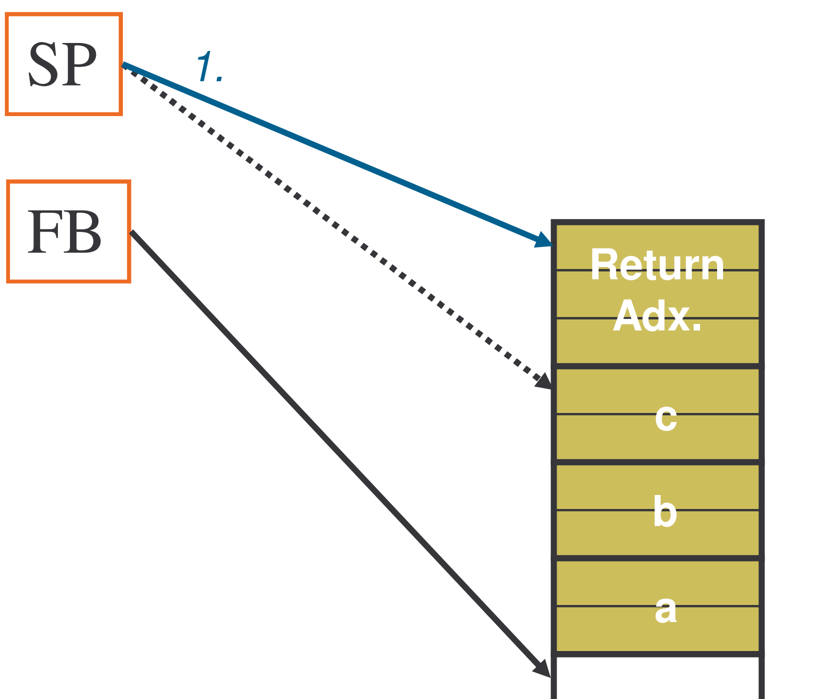
```
;## #     FUNCTION squared
;## #     FRAME  AUTO     (r)         size   2,  offset -2
;## #     REGISTER ARG  (r)         size   2,  REGISTER R1
;## #     ARG Size(0)       Auto Size(2)        Context Size(5)

    .glb   $squared
$squared:
    enter #02H
    mov.w           R1,-2[FB]          ;  r  r
    mov.w           -2[FB],R0          ;  r
    mul.w           -2[FB],R0          ;  r
    exitd
```

Call Stack before main executes jsr compute



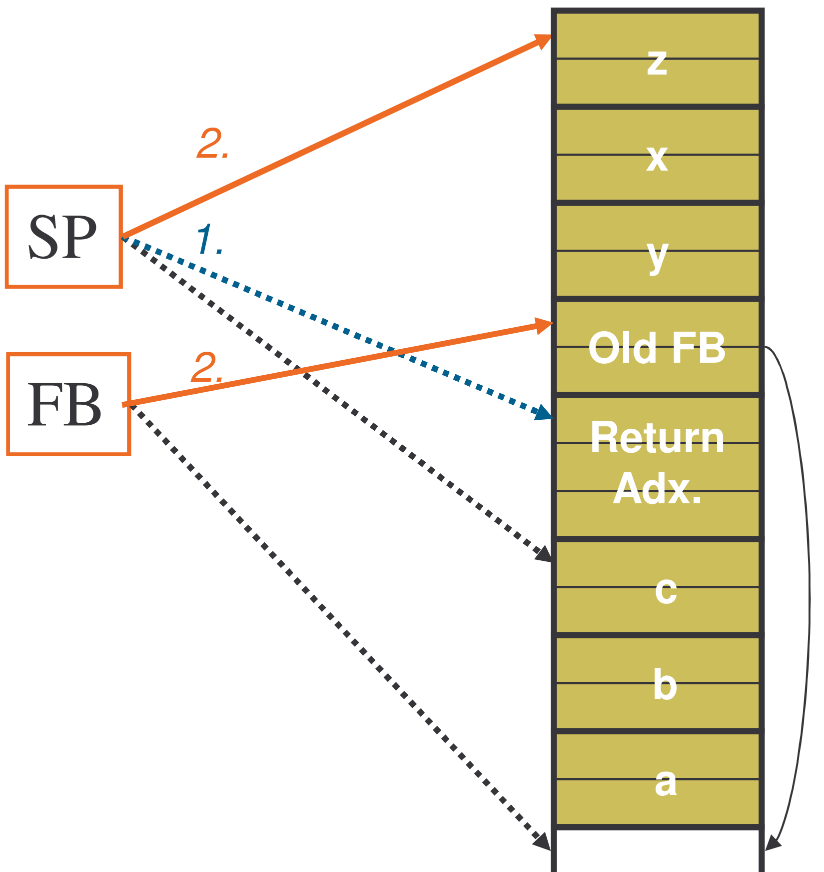
Call Stack arriving at compute



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

main's automatic variables

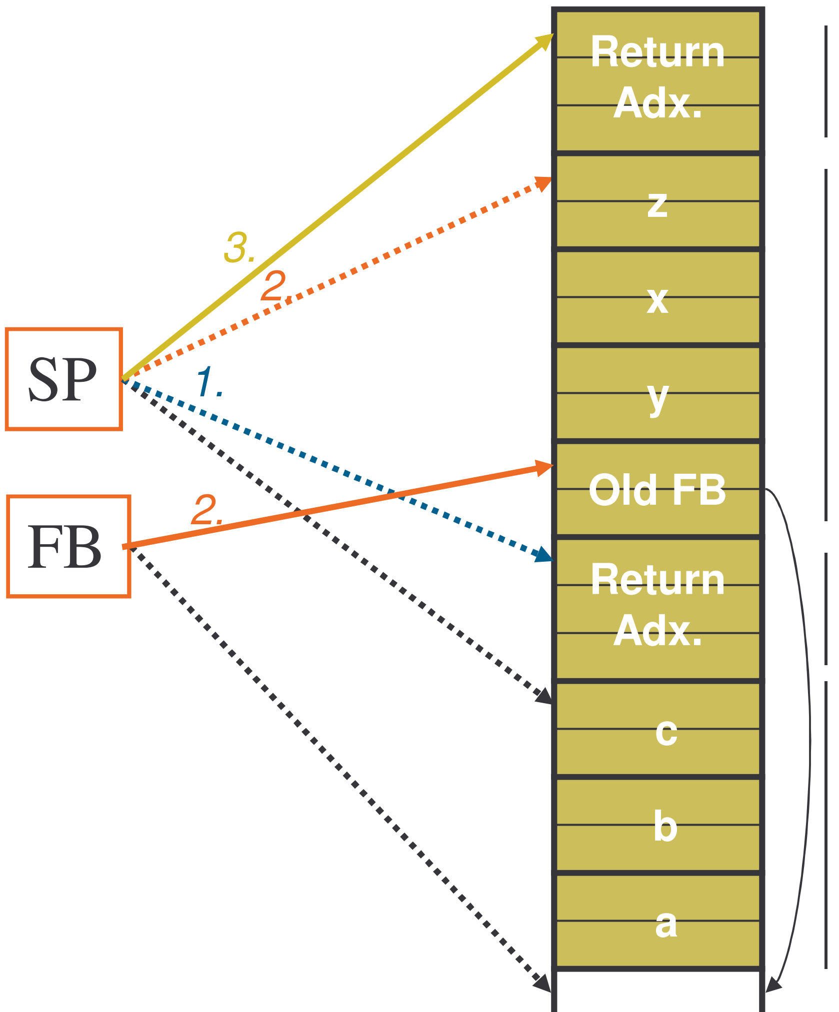
Call Stack executing enter #6 in compute



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

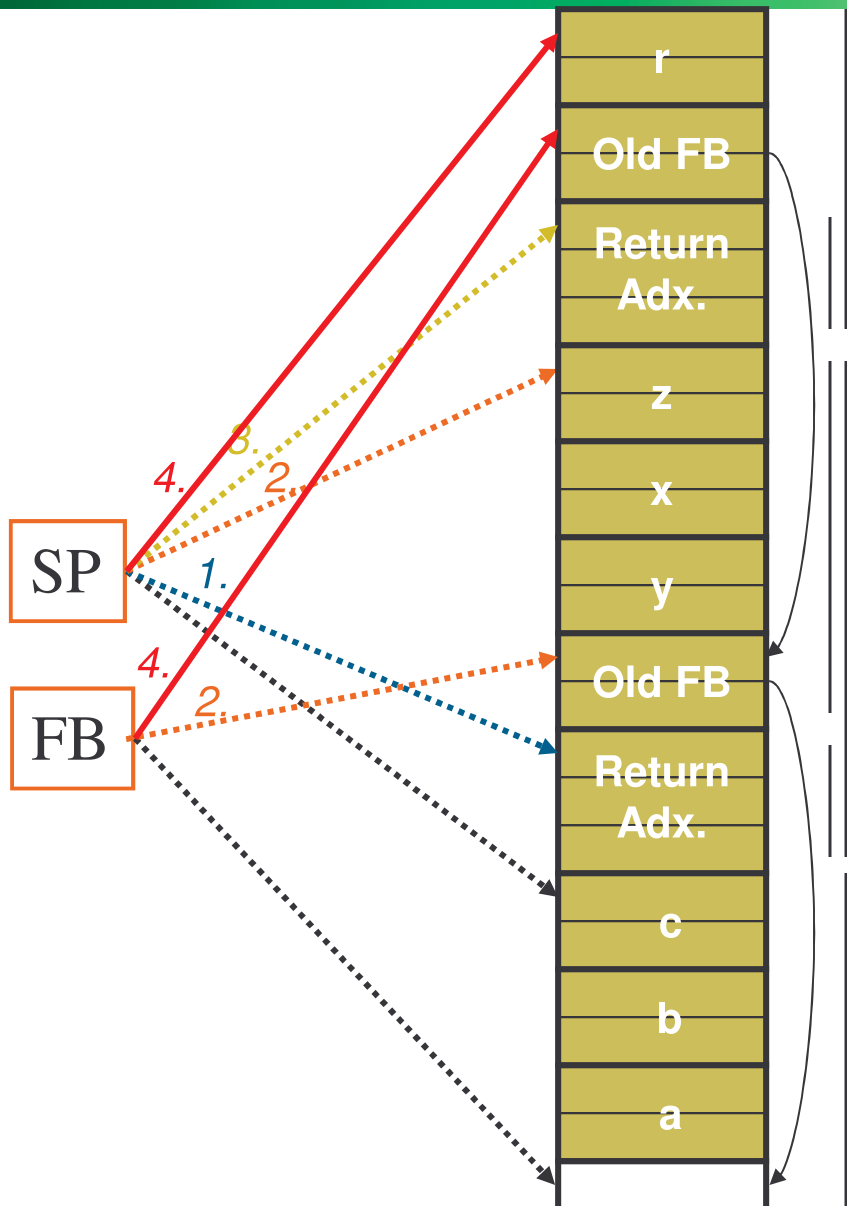


*3. **jsr** pushes old PC onto stack*

compute's activation record

main's activation record

Call Stack as squared executes enter #2

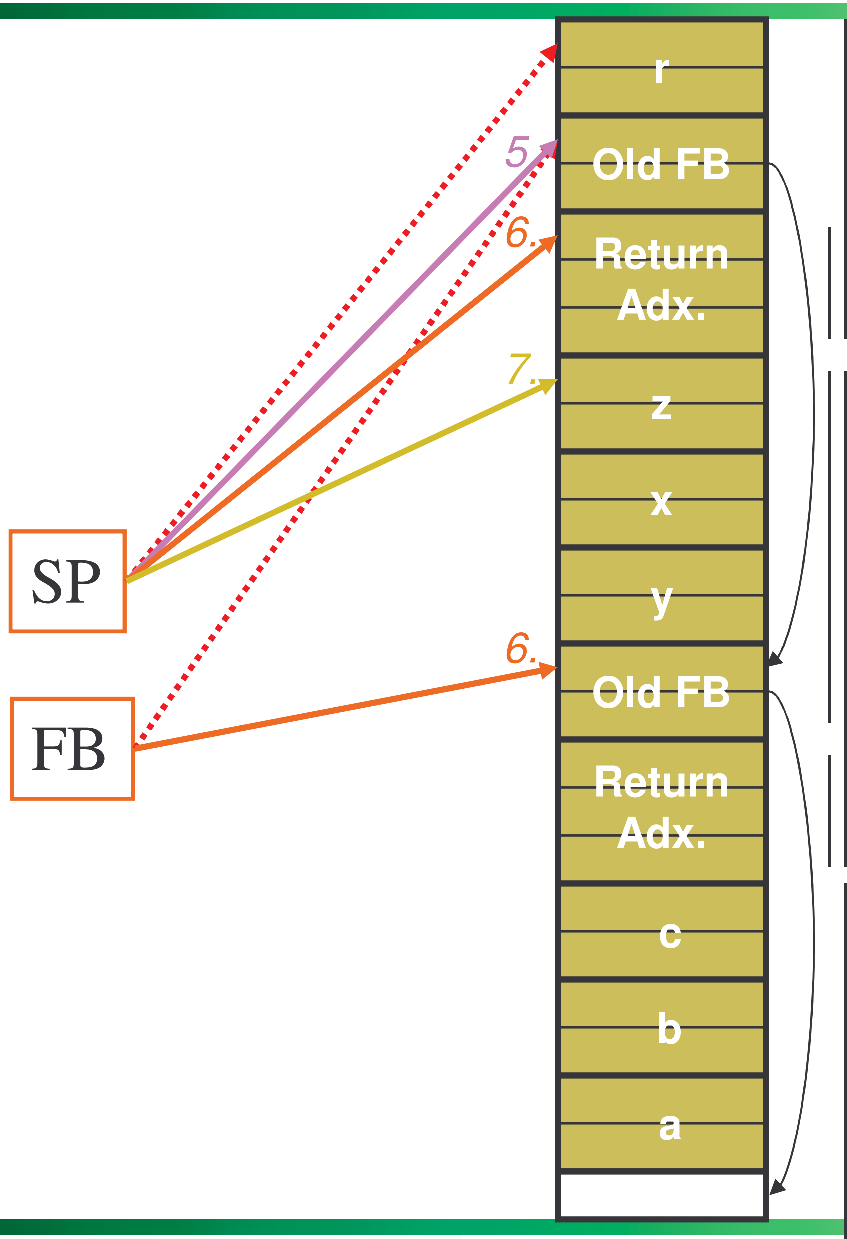


*4. **enter #2** pushes old FB value onto stack, copies SP to FB and allocates 2 more bytes (for r)*

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

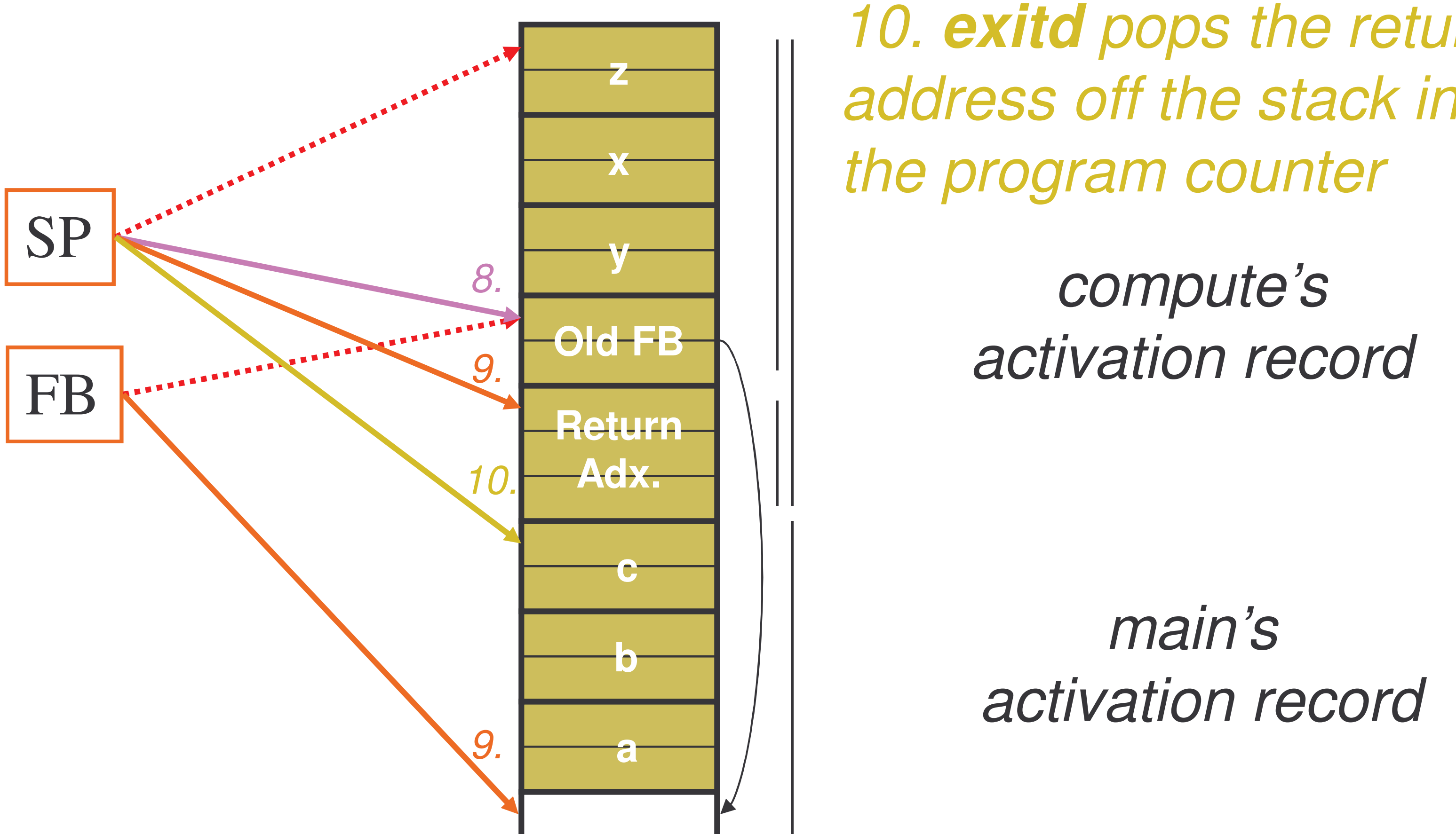
main's activation record

Call Stack as compute executes exitd

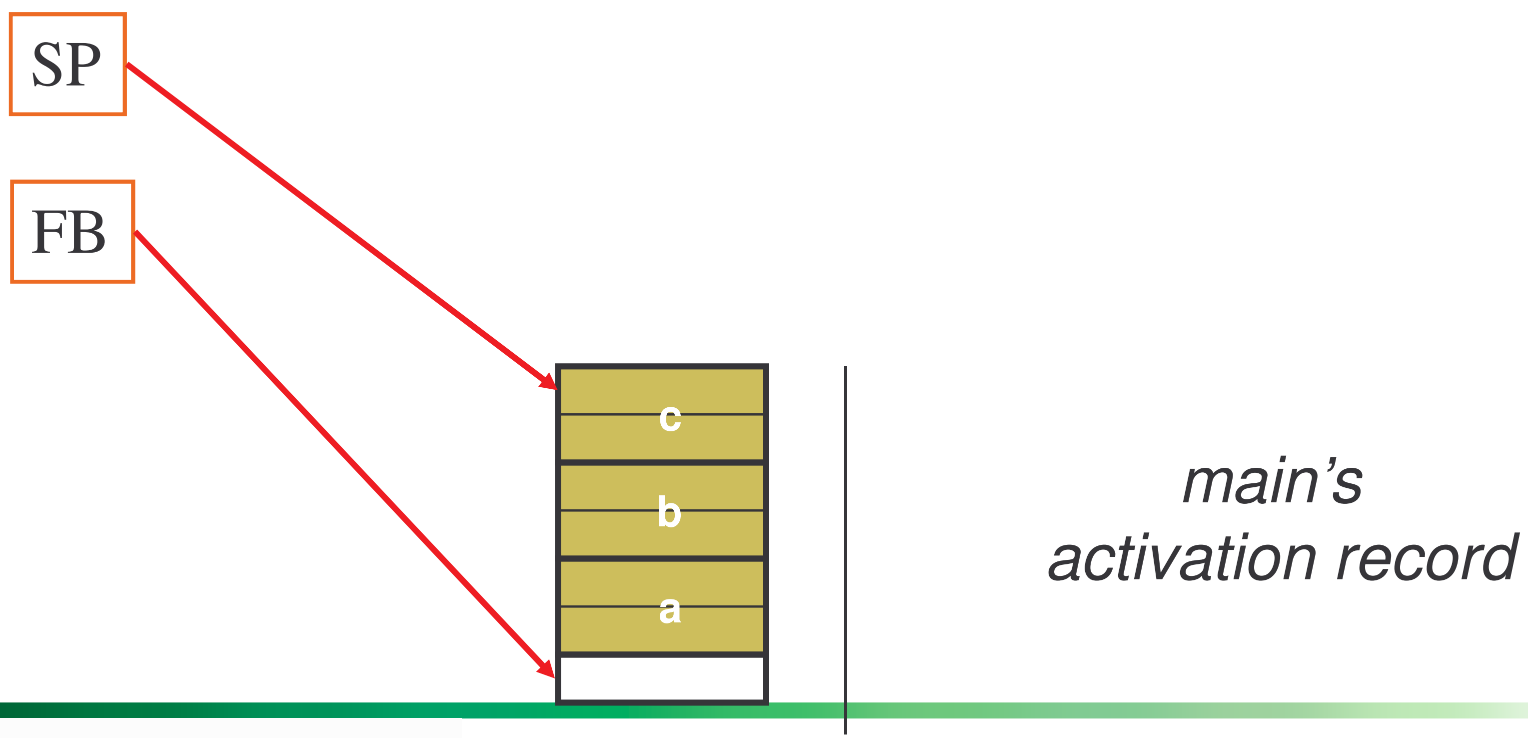
*8. **exitd** copies FB to SP, deallocating space for x,y,z*

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

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



Call Stack as compute continues in main



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)

- One byte items
  - Automatic variables, in order of declaration
  - Arguments which have been passed by register, in order of declaration
    - Why? For local temporary storage
- Two byte items
  - Automatic variables, in order of declaration
  - Arguments which have been passed by register, in order of declaration
    - Why? For local temporary storage
- etc.

Don't forget the old frame pointer and return address

Example of Passing Mixed Arguments

```
;## # FUNCTION compute2
;## # FRAME    AUTO (      z)        size  2,   offset -2
;## # FRAME    AUTO (      x)        size  2,   offset -2
;## #  FRAME   ARG (      f)  size   4,    offset 5
;## #  FRAME   ARG (      y)  size   2,    offset 9
;## #  REGISTER ARG    (      x)   size   2,   REGISTER R2
;## #     ARG Size(6)        Auto Size(2)        Context Size(5)
$compute2:
    enter #02H
    mov.w        R2,-2[FB]        ; x x
    mov.w        -2[FB],R1        ; x
    jsr    $squared
    mov.w        R0,-2[FB]        ; z
    mov.w        9[FB],R1         ; y
    jsr    $squared
    add.w        -2[FB],R0        ; z
    lde.w  _globalD,R1
    add.w        R1,R0
    mov.w        R0,-2[FB]        ; z
    mov.w        -2[FB],R0        ; z
    exitd
```

```
int compute2(float f, int x, int y) {
…
}
```

*Load argument **y** from stack*