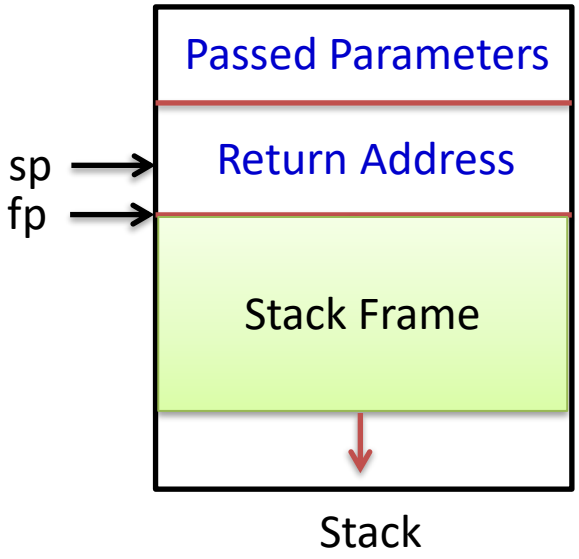
Last Time

- Stack-based languages, like C, make use of stacks for the following purposes
  - Call-return (pointer) mechanism
  - Parameters
    - Pass-by-value
    - Pass-by-reference
  - Local variables
  - Nested function calls
  - Recursive function calls
  - Re-entrant Code

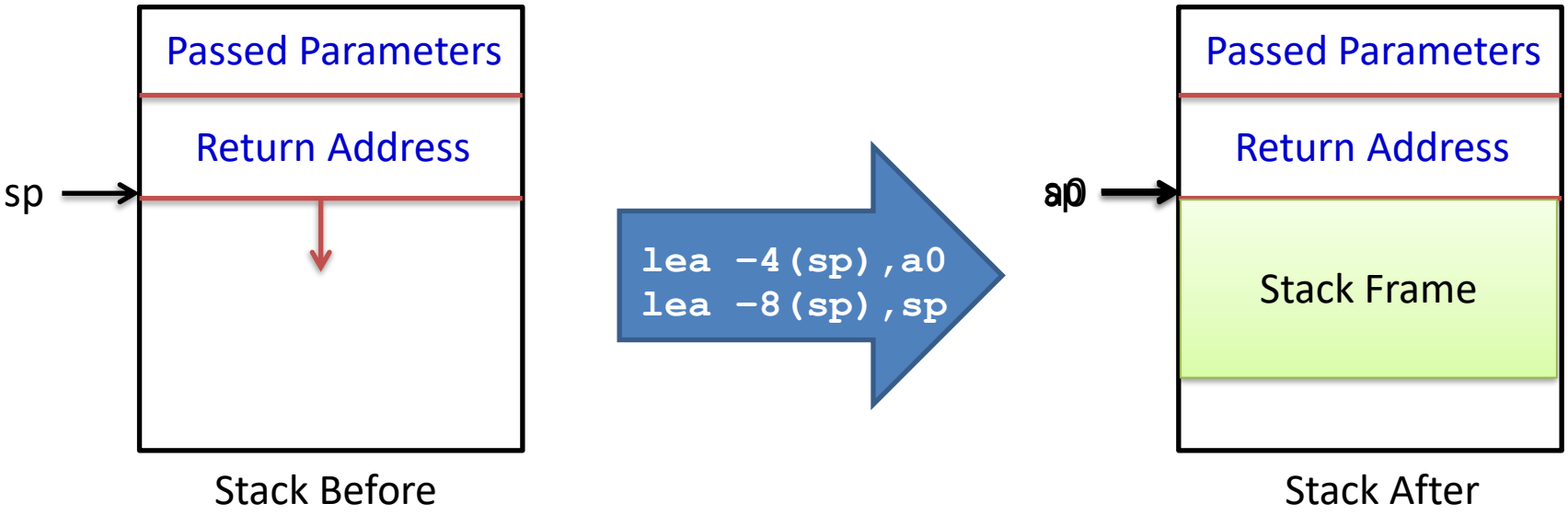
Local Variables and Stack Frames

- A subroutine frequently requires local variables
  - Local variables are temporary, intermediate values required only by the subroutine
- Two items closely associated with dynamic storage for local variables are
  - stack frame
  - frame pointer (fp)



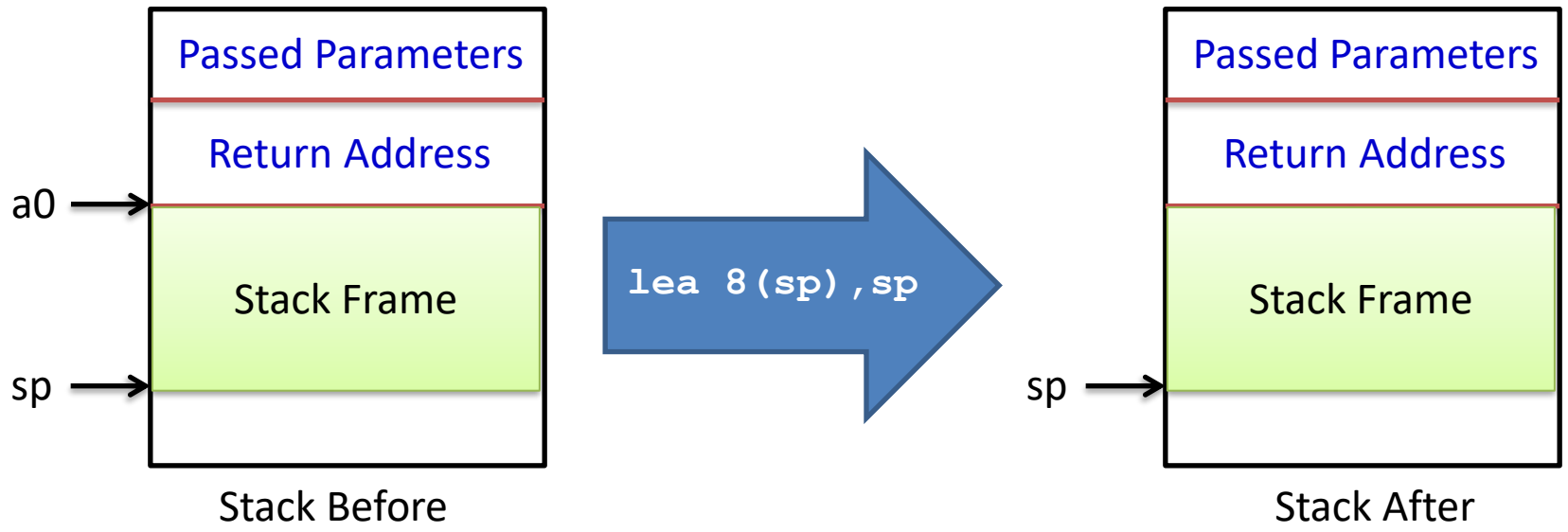
Manually Creating a Stack Frame

- Create a stack frame large enough to hold 8 bytes of local data and use a0 (point to first longword) as the frame pointer



Manually Destroy a Stack Frame

- Destroy the previous stack frame thus “freeing” the memory allocated to local variables

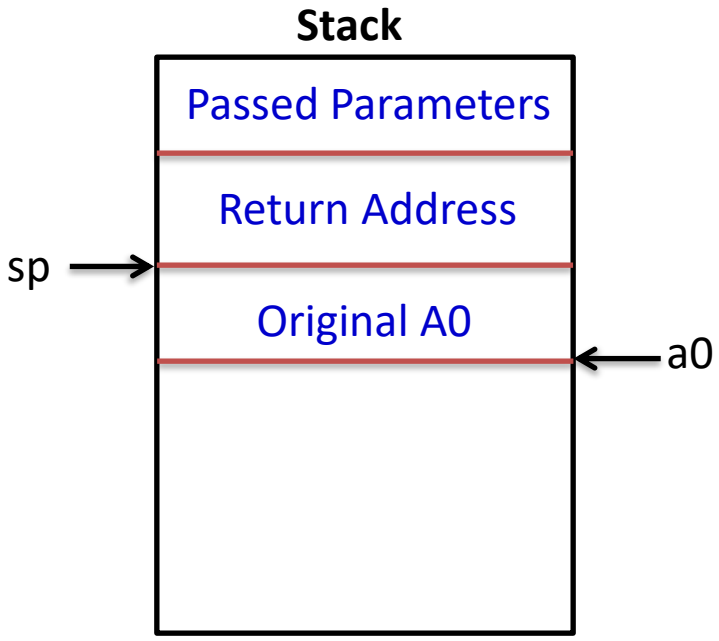


LINK Instruction

LINK Link and Allocate

Syntax: `link An,#-d`

Operation:
SP = SP – 4
Memory[SP] = A0
A0 = SP
SP = SP - d



- Consider the following code

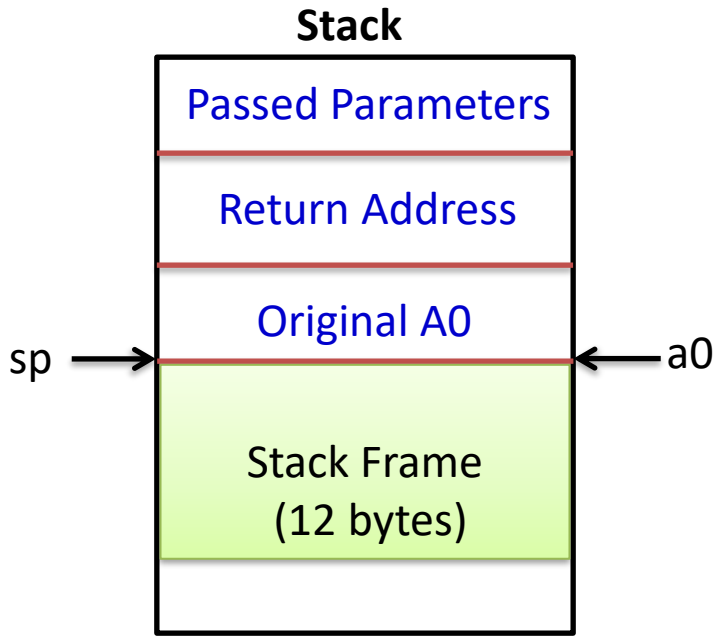
```
link a0,#-12
```

LINK Instruction

LINK Link and Allocate

Syntax: `link An,#-d`

Operation:
SP = SP – 4
Memory[SP] = A0
A0 = SP
SP = SP - d

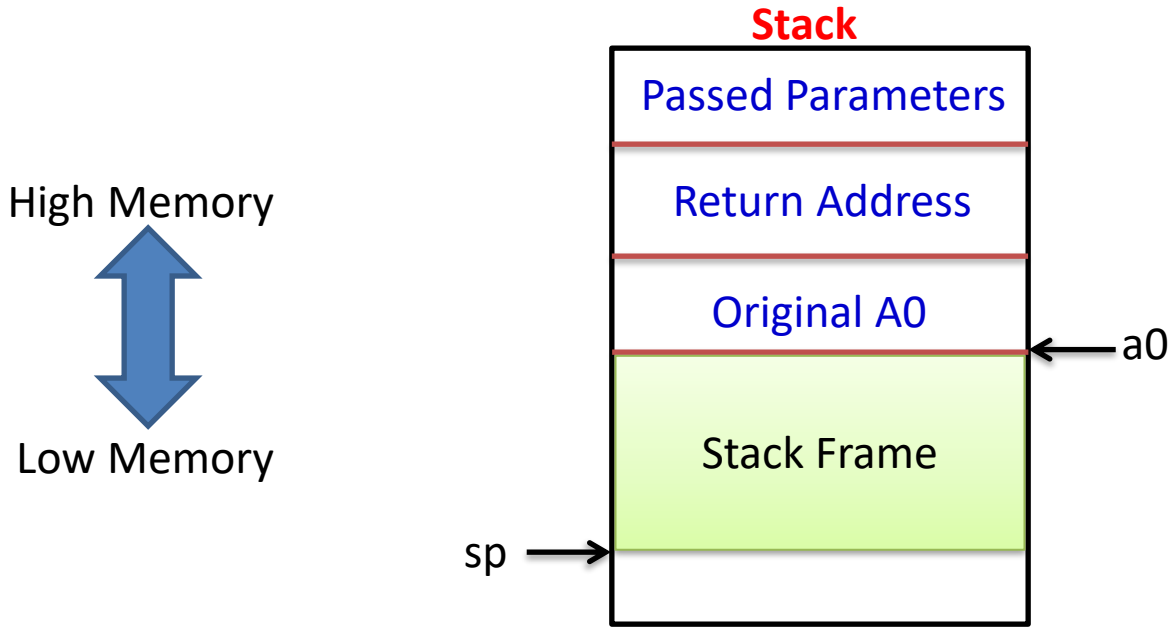


- Consider the following code

```
link a0,#-12
```

After Creating the Stack Frame

- Passed parameters are accessed with a positive displacement with respect to the frame pointer, A0
  - move +d(A0),Dn
- Local variables or temporary variables are accessed with a negative displacement with respect to the frame pointer, A0
  - move -d(A0),Dn

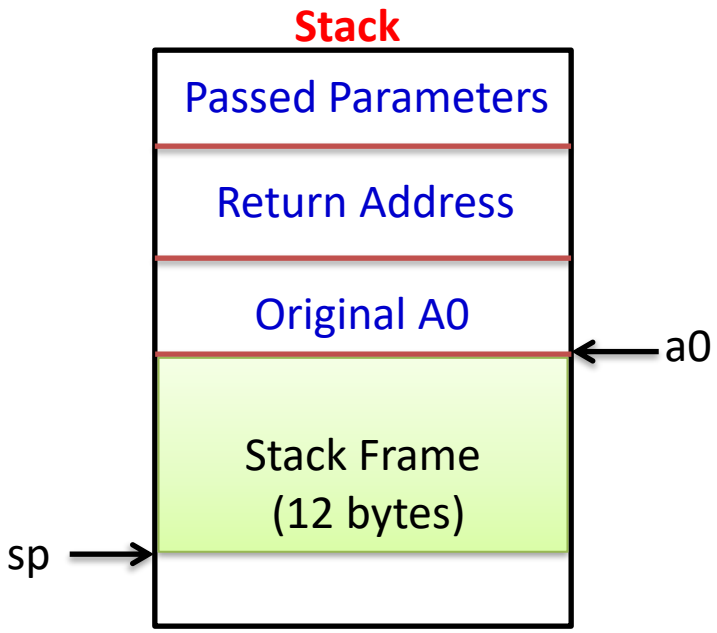


UNLK Instruction

UNLK Unlink

Syntax: unlk `An`

Operation: SP = An
An=Memory[SP]
SP = SP + 4



- Consider the following code

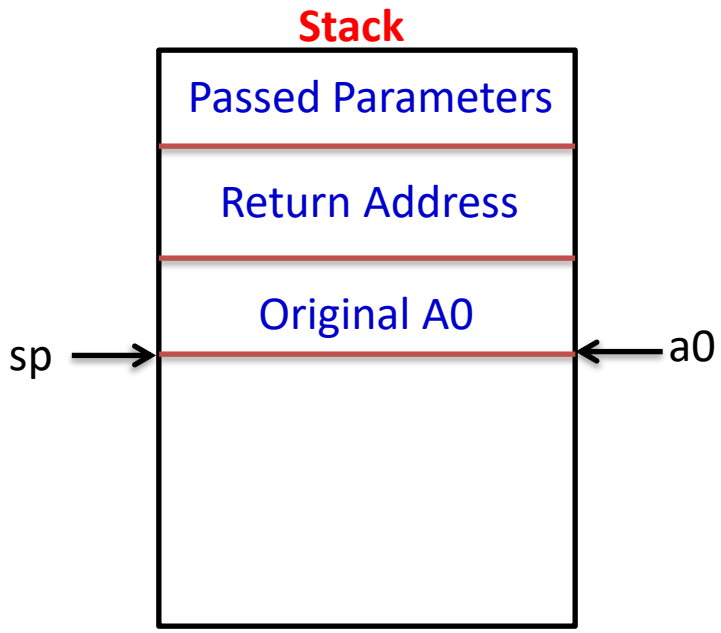
```
unlk a0
```

UNLK Instruction

UNLK Unlink

Syntax: unlk `An`

Operation: SP = An
An=Memory[SP]
SP = SP + 4



- Consider the following code

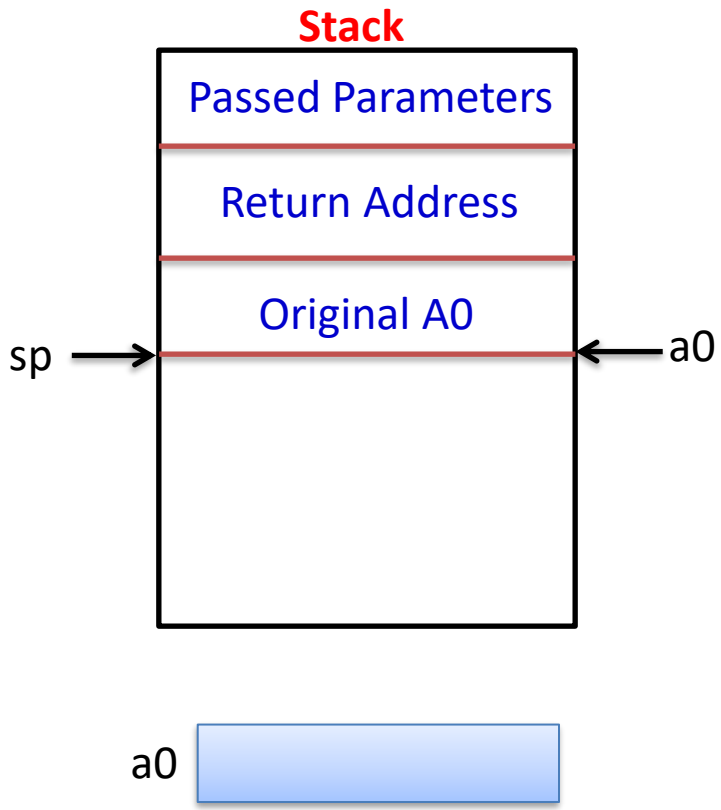
```
unlk a0
```

UNLK Instruction

UNLK Unlink

Syntax: unlk `An`

Operation: SP = An
An=Memory[SP]
SP = SP + 4



- Consider the following code

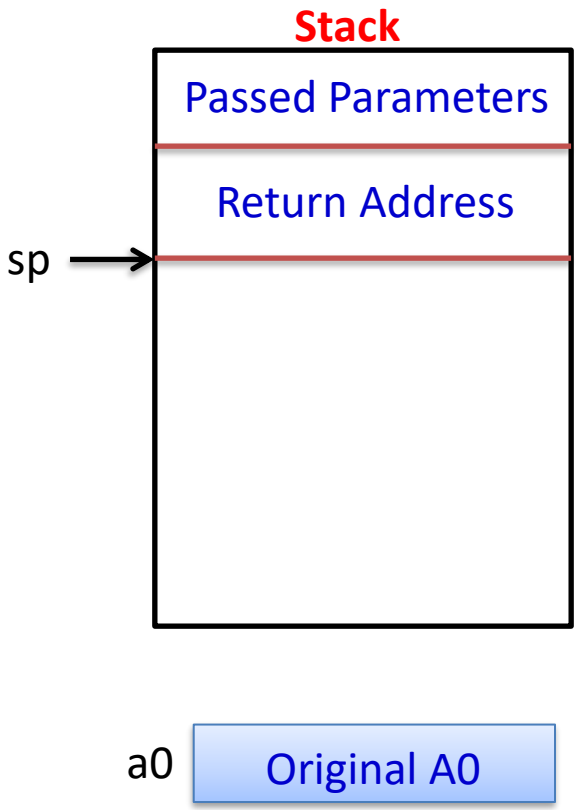
```
unlk a0
```

UNLK Instruction

UNLK Unlink

Syntax: unlk `An`

Operation: SP = An
An=Memory[SP]
SP = SP + 4



- Consider the following code

```
unlk a0
```

Call and Return Conventions with Stack Frames

- Caller
  - Push arguments onto stack (`MOVE Dn,-(sp)` or `PEA`)
  - Call subroutine by executing `JSR` instruction
  - Remove arguments from stack upon return (`LEA d(sp),sp`)
- Callee Setup
  - Allocate memory for new frame (`LINK An,#-d`)
  - Save working registers (`MOVEM reg.lis,-(sp)`), as needed
- Callee Running
  - Use frame pointer (`An`) and indirect addressing with offset (`±d(An)`) to access parameters and local variables
- Callee Return
  - Put return value in data register (`D0`), if required
  - Restore working registers (`MOVEM (sp)+,reg.lis`), as needed
  - De-allocate memory for stack frame (`UNLK An`)
  - Return by `RTS`

Example

- Use a subroutine to calculate the following:

$$\text{Result} = (P^2 + Q^2) / (P^2 - Q^2)$$

- Assumptions:
  - Parameters P and Q are 16-bit unsigned values passed to the subroutine by *value*
  - The result is a 32-bit long word that is passed by *reference*
  - A minimum number of registers must be used (i.e., 1 data register and 1 address register not including the frame pointer)
  - All working registers should be saved on the stack
  - A stack frame should be used to hold the temporary values P^2, Q^2 P^2+Q^2, and P^2-Q^2

Assembler Code – Part 1

```
* Reserve storage for Result
        org             $2000
result  ds.l            1

* Setup dummy values for working registers
        lea             $a6a6a6a6,a6            ;pointer
        lea             $a0a0a0a0,a0            ;frame pointer
        move.l          #$d6d6d6d6,d6           ;working register

* push arguments onto stack and call subroutine
        move.w          #7,-(sp)        ;push value of P
        move.w          #6,-(sp)        ;push value of Q
        pea             result          ;push address of result
        jsr             calc            ;evaluate expression
        lea             8(sp),sp        ;clean up stack
        trap            #14             ;return to 68KMB
```

Assembler Code – Part 2

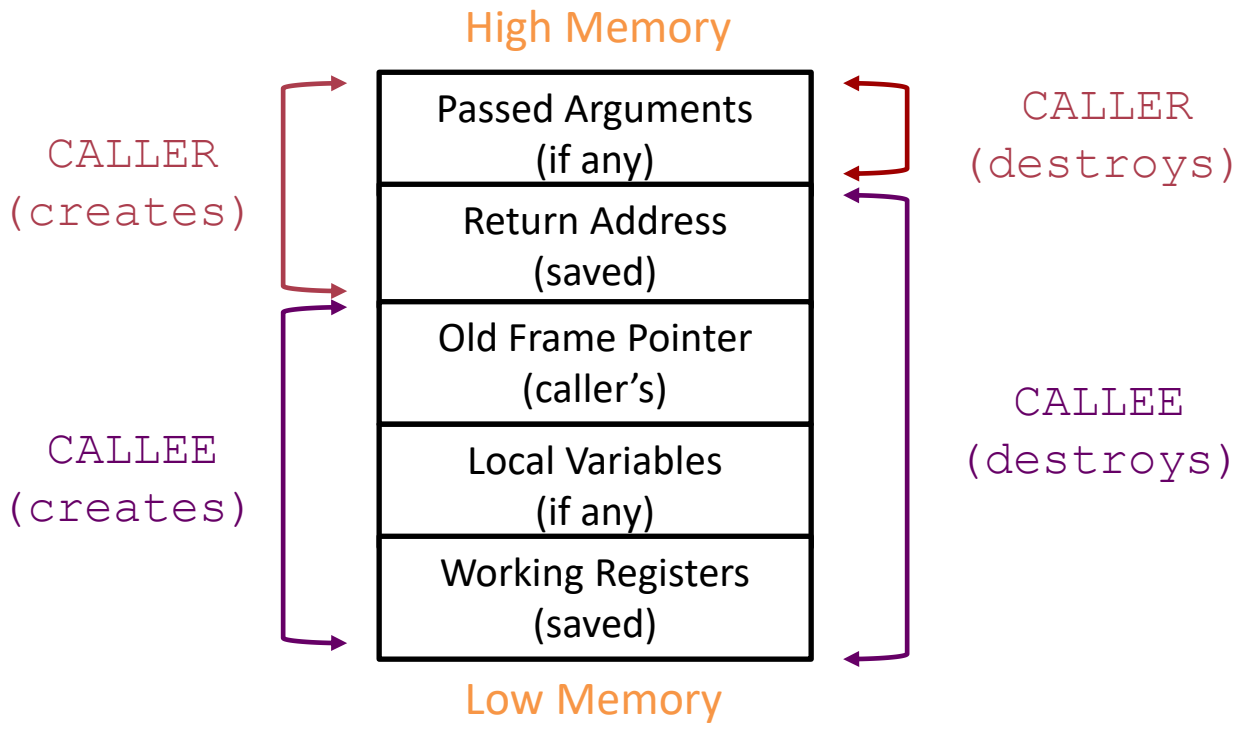
```
* Subroutine calc

calc    link      a0,#-14        ;allocate 14-byte stack frame
        movem.l   d6/a6,-(sp)    ;save working registers

        move.w    14(a0),d6      ;get P from stack
        mulu      d6,d6          ;calculate P2
        move.w    d6,-4(a0)      ;save p2 stack

        move.w    12(a0),d6      ;get Q from stack
        mulu      d6,d6          ;calculate Q2
        move.w    d6,-8(a0)      ;save Q2 stack

        move.l    -4(a0),d6      ;get P2 from stack
        add.l     -8(a0),d6      ;calculate P2
        move.l    d6,-12(a0)     ;save P2 + Q2 on stack frame
```

Call and Return Conventions with Stack Frames

- Caller
  - Push arguments onto stack (`MOVE Dn,-(sp)` or `PEA`)
  - Call subroutine by executing `JSR` instruction
  - Remove arguments from stack upon return (`LEA d(sp),sp`)
- Callee Setup
  - Allocate memory for new frame (`LINK An,#-d`)
  - Save working registers (`MOVEM reg.lis,-(sp)`), as needed
- Callee Running
  - Use frame pointer (`An`) and indirect addressing with offset (`±d(An)`) to access parameters and local variables
- Callee Return
  - Put return value in data register (`D0`), if required
  - Restore working registers (`MOVEM (sp)+,reg.lis`), as needed
  - De-allocate memory for stack frame (`UNLK An`)
  - Return by `RTS`

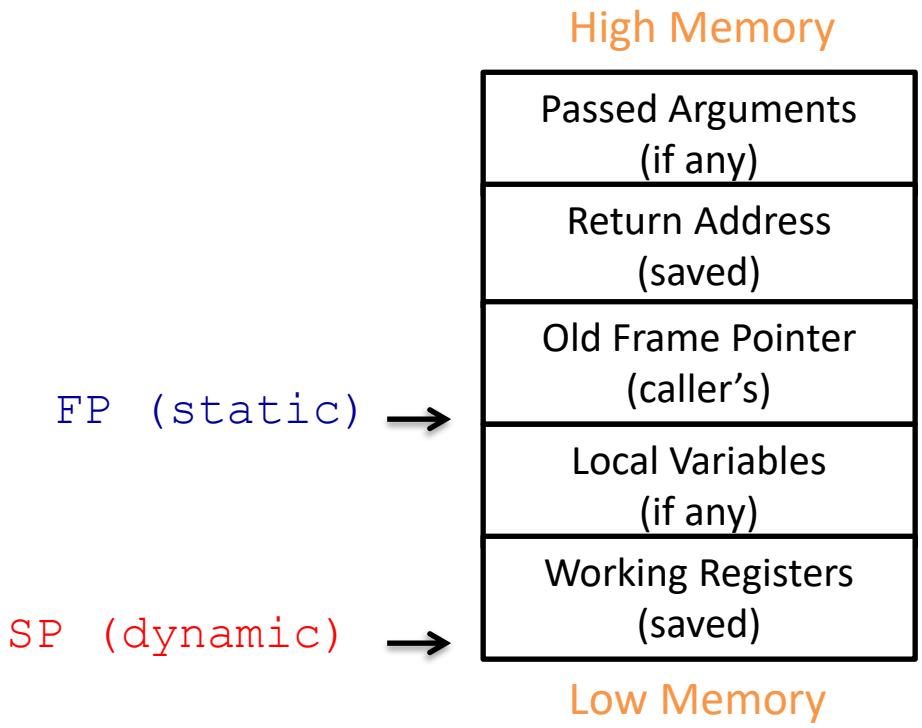
Activation Record

- Each *call* to a C function causes an activation record to be created on the runtime stack
  - Contains all data necessary for function to execute



Activation Record

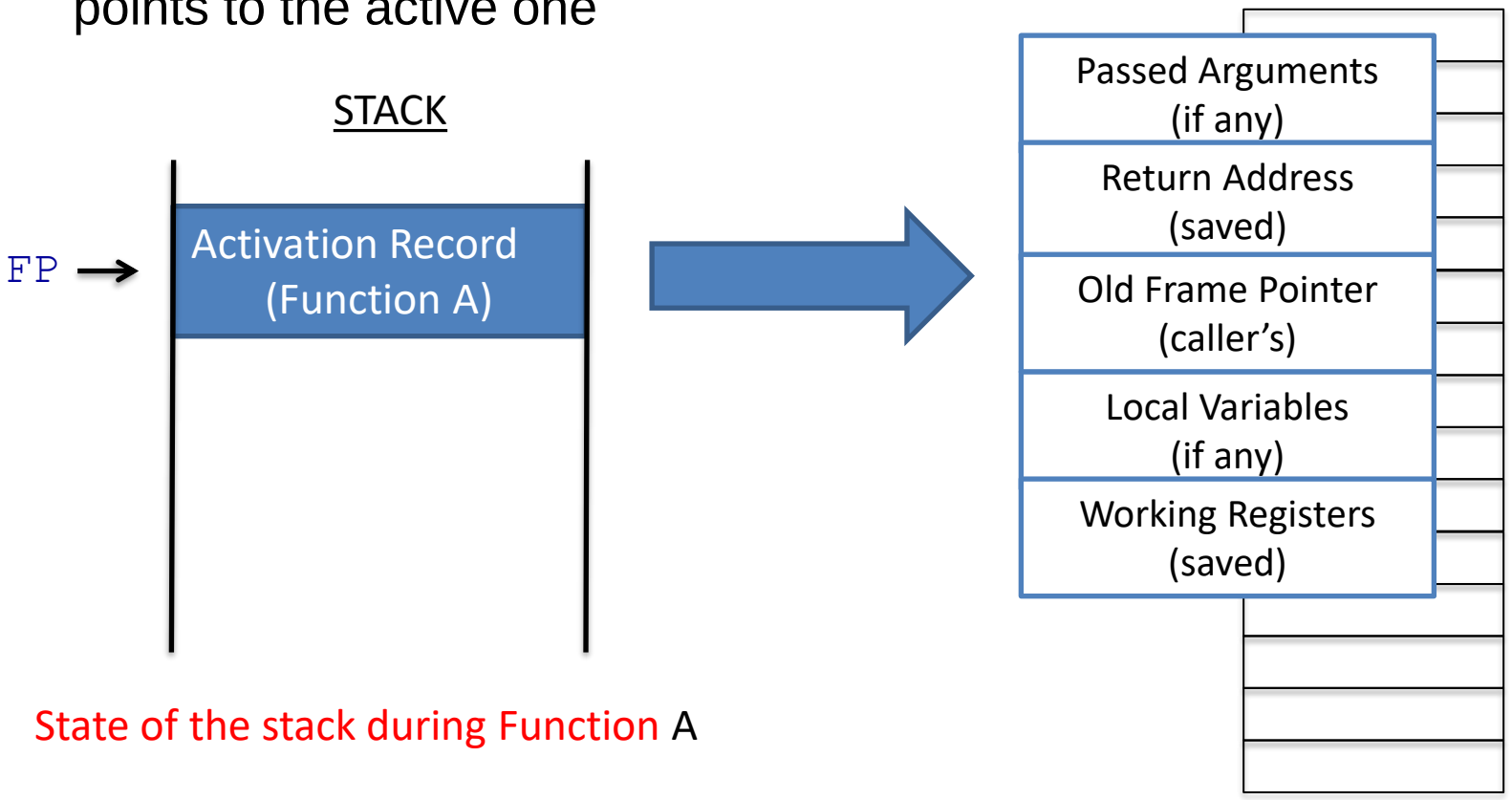
- Each *call* to a C function causes an activation record to be created on the runtime stack
  - Contains all data necessary for function to execute



Frame pointer is used to access all *arguments* and *local* values

Multiple Activation Records and the FP

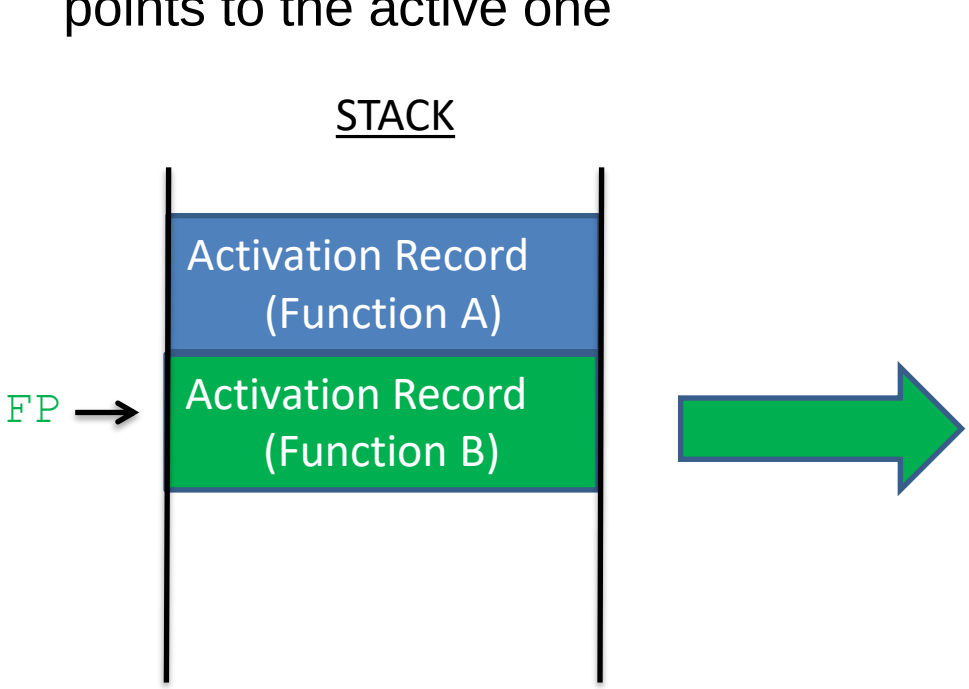
- Multiple Activation Records build on top of one another, and the FP points to the active one

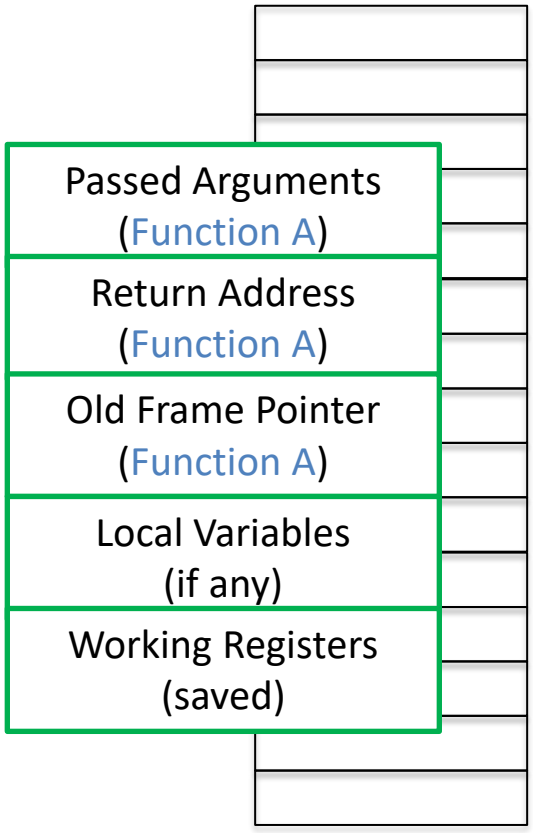


State of the stack during Function A

Multiple Activation Records and the FP

- Multiple Activation Records build on top of one another, and the FP points to the active one

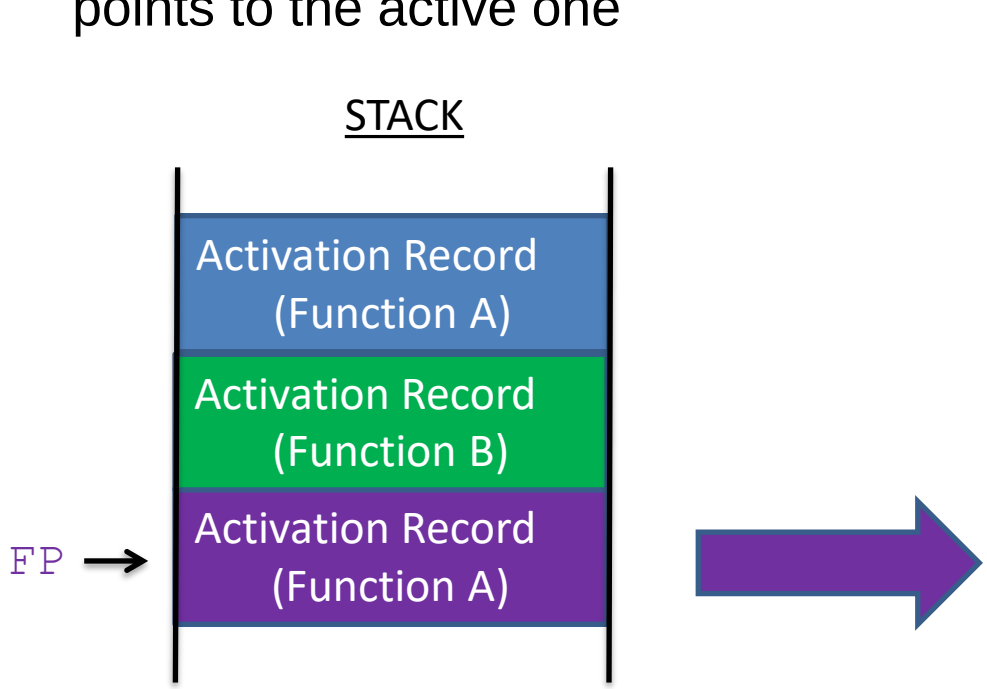


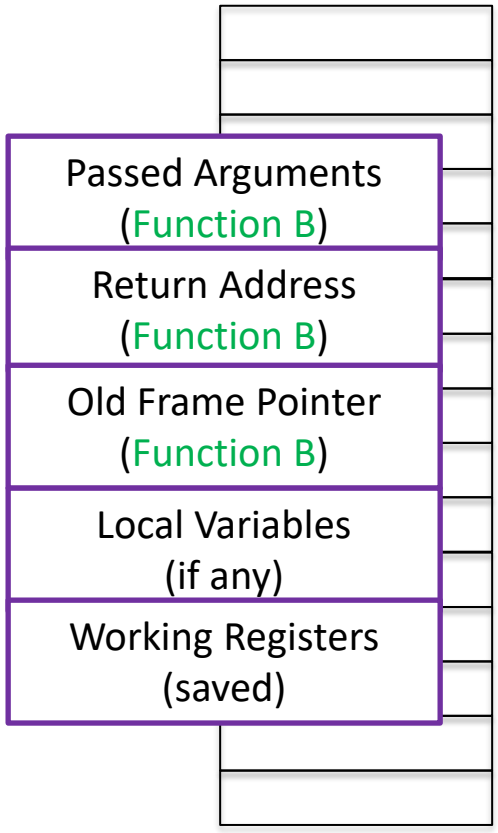


State of the stack during Function B
after a call from Function A

Multiple Activation Records and the FP

- Multiple Activation Records build on top of one another, and the FP points to the active one

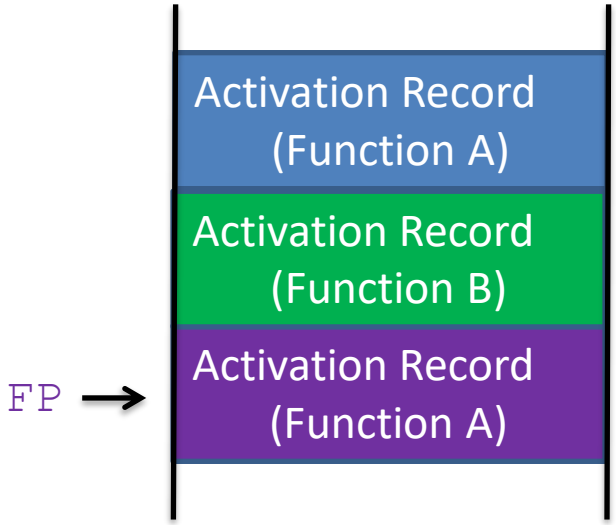




State of the stack after second call to Function A is completed

Multiple Activation Records and the FP

- Multiple Activation Records build on top of one another, and the FP points to the active one



Chain (linked-list) of FPs allow us to trace backwards through preceding calls until we reach the original call