J Compiler Overview

CIS 2400 Recitation 10

Recitation Outline

- HW10/11 Overview & the J language
- HW10/11 Structure
  - token.h
  - Program Structure
- LC4 Details
  - The Call Stack
  - Calling conventions
  - LC4 assembler Directives
- Tips
  - Testing
  - Error Handling

Overview

Overview

In this assignment, you will read in a .j file and create an equivalent LC4 .asm file

THAT'S IT

You don't have to worry about simulating anything, setting up the PC to run main first, etc.

The J Language: Basics

Stack-based language, similar to the RPN calculator from HW07

Literals:

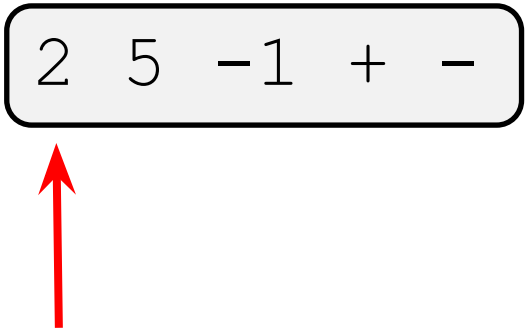
- All values can be represented as 16-bit 2C
- Positive or negative
- Decimal (digits and - sign) or hexadecimal (preceded by `0x`) formats

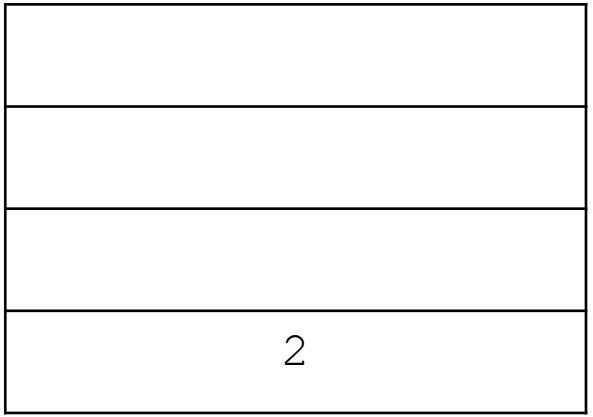
Example:

```
2 5 -1 + -
```

The J Language: Basics

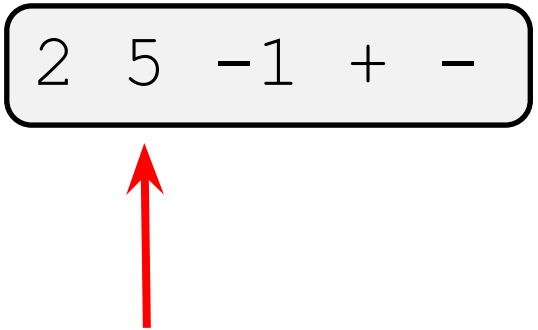
Example:

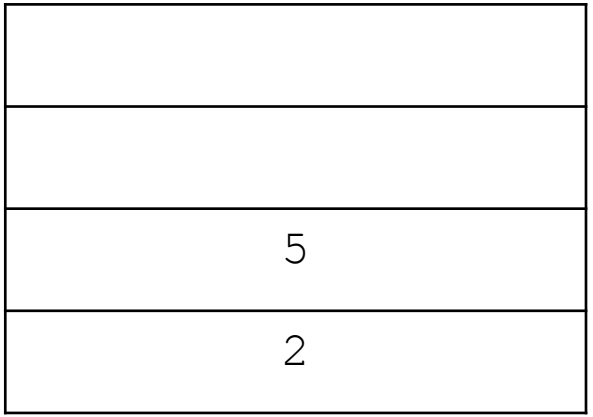




The J Language: Basics

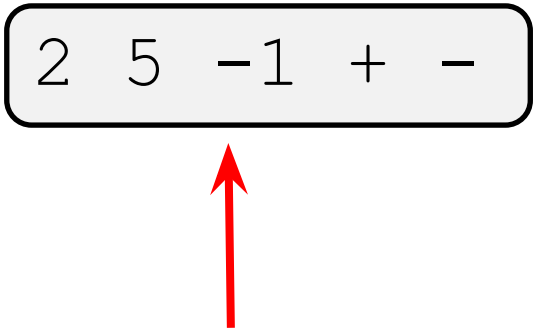
Example:

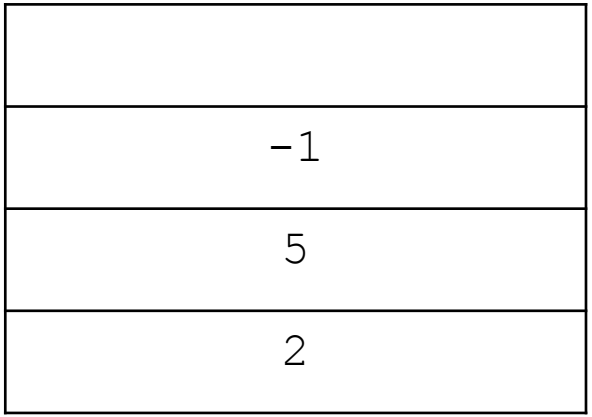




The J Language: Basics

Example:





The J Language: Basics

Example:

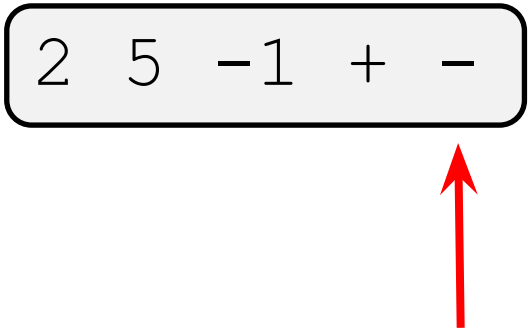
```
2 5 -1 + -
```

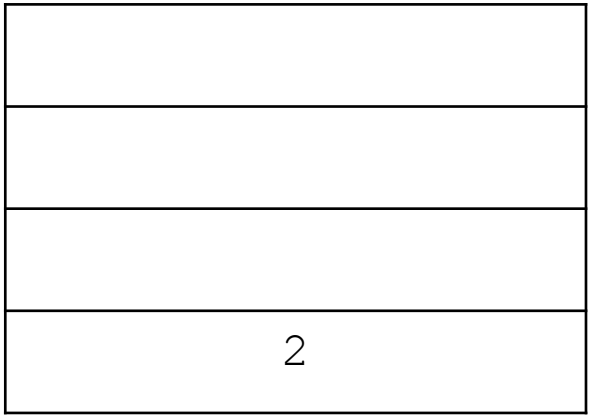
4
2

Note that the top value of the stack is the first operand!

The J Language: Basics

Example:





The J Language: Basics

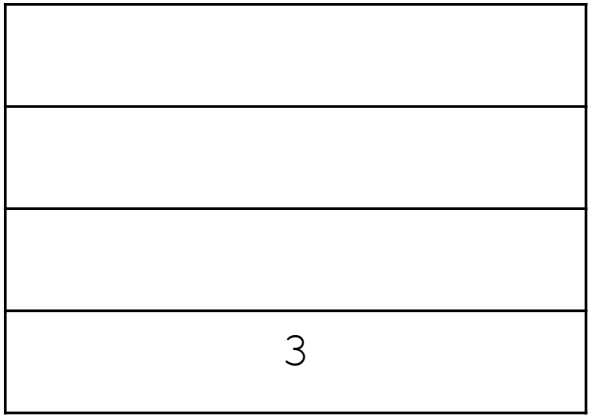
Example:

```
1 3 2 0x7 + / *
```

The J Language: Basics

Example:

```
1 3 2 0x7 + / *
```



The J Language: Advanced

Not as simple as HW07:

- Other operators
- Comparison, rotations, etc.
- If/else/endif
- While loops
- Functions

If/Else/Endif

We can also have if/else/endif → conditional is met if value is non-zero

- Example

```
7 2 % if 1 else 0 endif
```

- Same example formatted differently (look familiar?)

```
7 2 %
if
    1
else
    0
endif
```

If/Else/Endif Example

```
7 2 % if 1 else 0 endif
```

If/Else/Endif Example

```
7 2 % if 1 else 0 endif
```

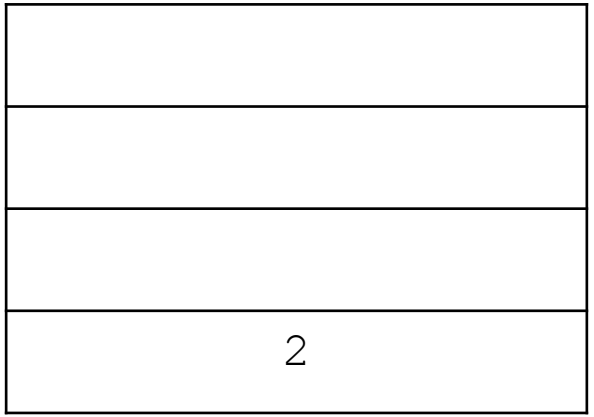
2
7

If/Else/Endif Example

```
7 2 % if 1 else 0 endif
```

Note that the Top value of the stack is the first operand!

2 % 7 = 2



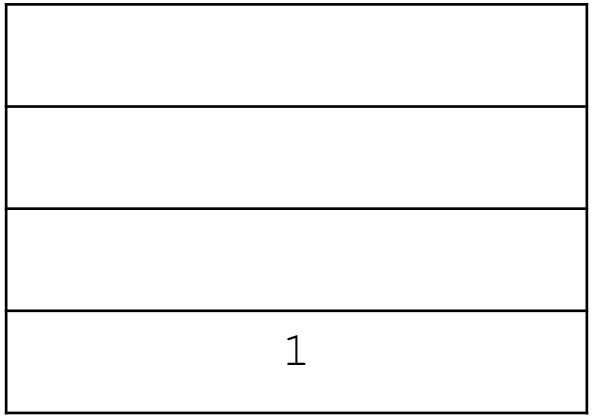
If/Else/Endif Example

```
7 2 % if 1 else 0 endif
```

If/Else/Endif Example

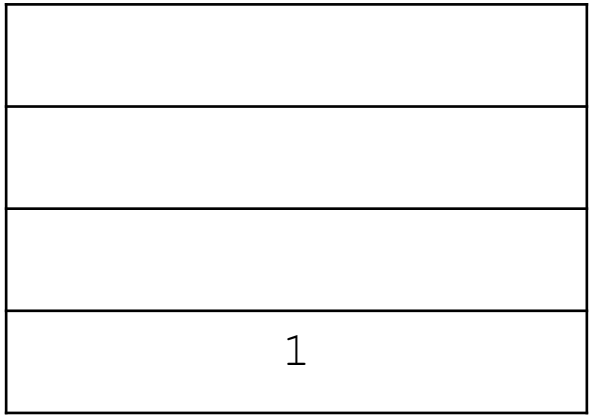
```
7 2 % if 1 else 0 endif
```

2 is nonzero, so we push 1 to the stack



If/Else/Endif Example

```
7 2 % if 1 else 0 endif
```



The else branch
is skipped over

If Statements – Some Caveats

- Some if statements may not have an else statement

```
7 2 % if 1 endif
```

- There can be programs that have many if statements
  - If you are using labels in LC4 (which you should do), you must have unique labels for the different IF/ELSE/ENDIF statements
- There may be some programs that have nested if/else/endif statements:

```
2 3 4 - if - if 2 else 1 endif else 0 endif
```

Stack Operations

If we had a stack (bottom) 3 4 5 (top)

drop	3 4
dup	3 4 5 5
swap	3 5 4
rot	4 5 3

J Functions

J can define functions with the token `defun`

Example function:

- First token after `defun` should be an identifier naming the function
- `argN` gets the nth value from top of the stack starting before the function is invoked.
- `return` returns from the function, placing the top value of the stack as the new top value of the stack for the caller.

```
defun square
arg1
dup * dup
return
```

J Functions: Example

Say we have the function

```
defun square
arg1
dup * dup
return
```

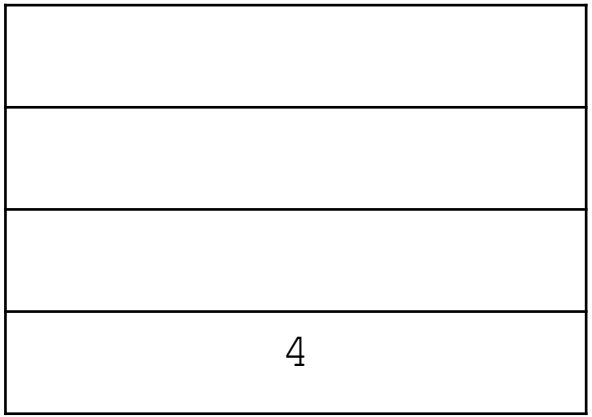
and call it with

```
4 square
```

J Functions: Example

```
defun square
arg1
dup * dup
return
```

```
4 square
```



J Functions: Example

```
defun square
arg1
dup * dup
return
```

```
4 square
```

Caller's portion of the stack (e.g. caller's stack frame)

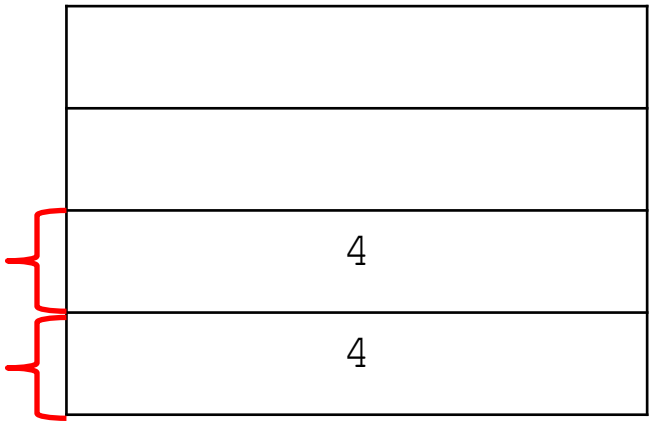
J Functions: Example

```
defun square
arg1
dup * dup
return
```

```
4 square
```

square's stack frame

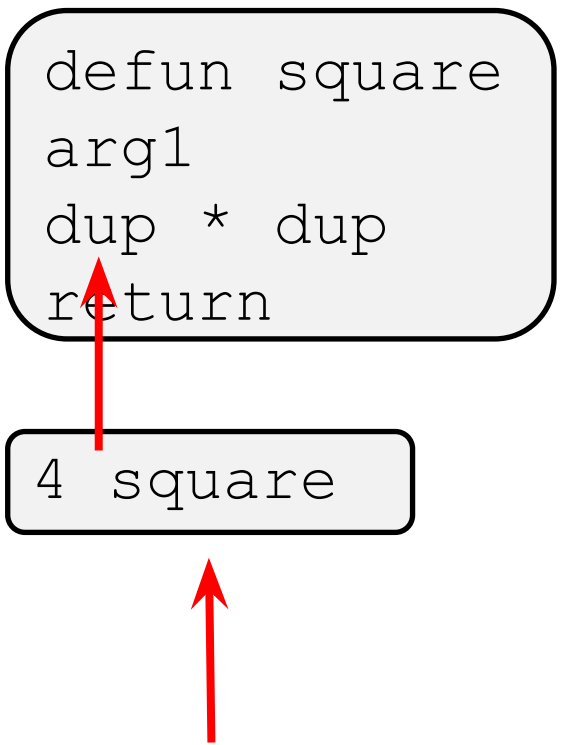
caller's stack frame



J Functions: Example

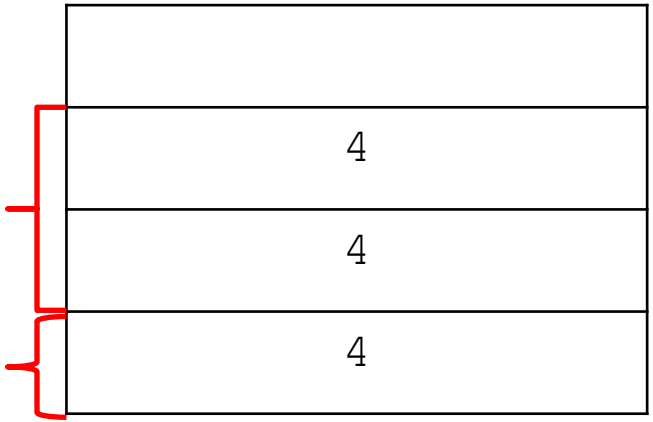
```
defun square
arg1
dup * dup
return
```

```
4 square
```



square's stack frame

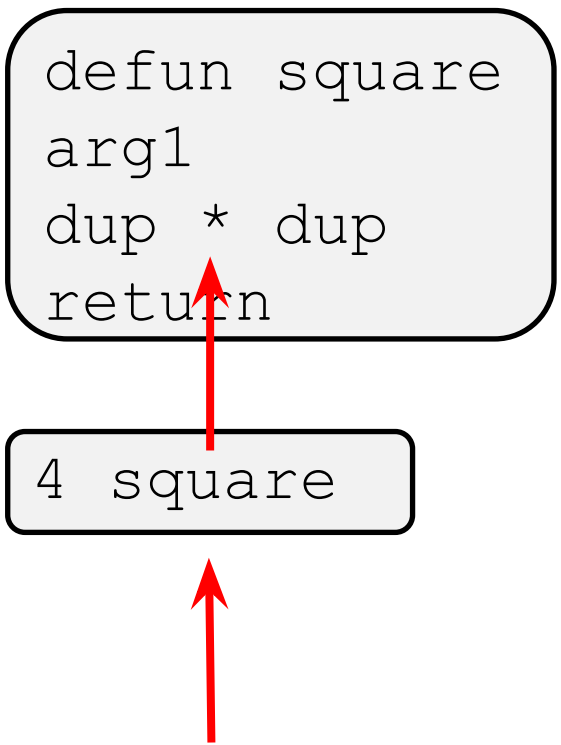
caller's stack frame



J Functions: Example

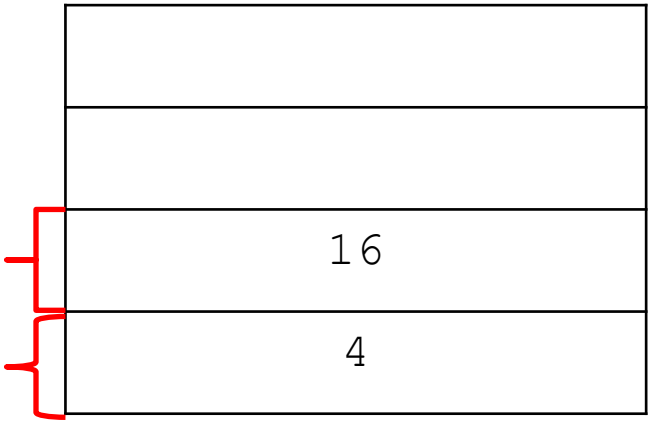
```
defun square
arg1
dup * dup
return
```

```
4 square
```



square's stack frame

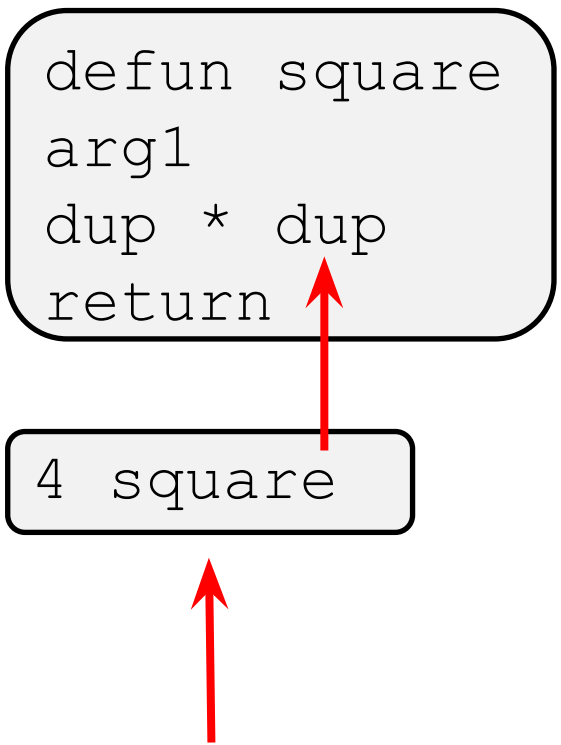
caller's stack frame



J Functions: Example

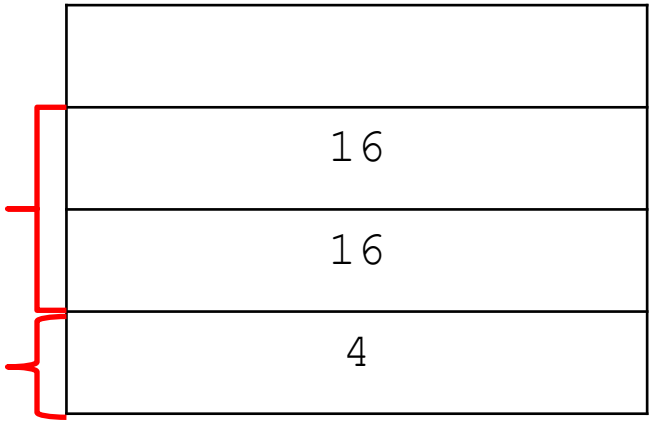
```
defun square
arg1
dup * dup
return
```

```
4 square
```



square's stack frame

caller's stack frame

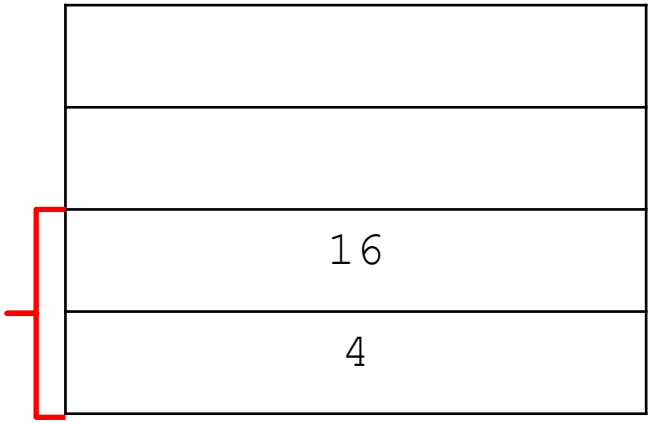


J Functions: Example

```
defun square
arg1
dup * dup
return
```

```
4 square
```

caller's stack frame



This is all J at a high level, many LC4 details left out (for now)

Code Structure

Getting Started

- To get started with the homework, we recommend that you implement token.c to read from a file and output tokens.
- A 'token' in a .j file can be 'arg1', '+', 'defun', etc.
- We represent a token with:

```
typedef struct {
  token_type type;
  int literal_value;
  int arg_no;
  char str[MAX_TOKEN_LENGTH];
} token;
```

Enum similar to INSN_TYPE

The value if the token is a literal (e.g. '5' or '0xCAFE')

Value of N for argN

Str value of a IDENT token (more on IDENT in a moment)

Getting Started

The token types are mostly self explanatory, with two exceptions (IDENT, LITERAL)

- DEFUN -> “defun” in the .j file
- ARG -> “arg1” or som other argN in the.j file
- Etc.

```
typedef enum { DEFUN, IDENT, RETURN,
               PLUS, MINUS, MUL, DIV, MOD,
               AND, OR, NOT,
               LT, LE, EQ, GE, GT,
               IF, ELSE, ENDIF, WHILE,
               DROP, DUP, SWAP, ROT,
               ARG, LITERAL, BAD_TOKEN } token_type;
```

LITERAL Token

- Used to represent an integer “literal” in j
- For example, in the program

```
7 5 4 + -
```

  7, 5 and 4 are literals.

- Can have a leading ‘-’ to mark the number as negative
  - Be careful! There is also a token that is just the ‘-’ symbol
- Can be in hexadecimal e.g.

```
0x7 5 0x0004 + -
```

IDENT Token

- Used to find tokens that identify a function
- In the example program, there are 3 ident tokens
- A token is an IDENT if it is not one of the other token types, starts with a letter and the following characters can be under scores '_', numbers, or other letters
- You can assume that all calls to functions are to defined functions and that there are no duplicate function names

```
defun square
arg1
dup *
return

defun main
4 square
return
```

token.h

- In token.h, we provide the declaration for the function:

```
int read_token(token *theToken, FILE *theFile);
```

  - Takes in a FILE* theFile to read from
  - Returns a token through theToken output parameter
  - Returns whether an error was encountered or not
- Many way to read the file:
  - Read the file line by line (using something like fgets)
  - Read the file string by string (probably using fscanf)
  - Read the file character by character (using fgetc)
  - Some combination of these
- You are also allowed to modify this function

Token Processing

With how J works, almost all tokens can be processed on their own (e.g. you don't have to read future tokens and/or remember pass tokens to process it)

There are two exceptions to this:

- Function definitions
  - Need to read the defun token and the next token which should be an IDENT for the function name
- The if/else/endif tokens
  - Need to know the labels to jump to, and handle nested if/else/endifs

Program Structure

Main program:

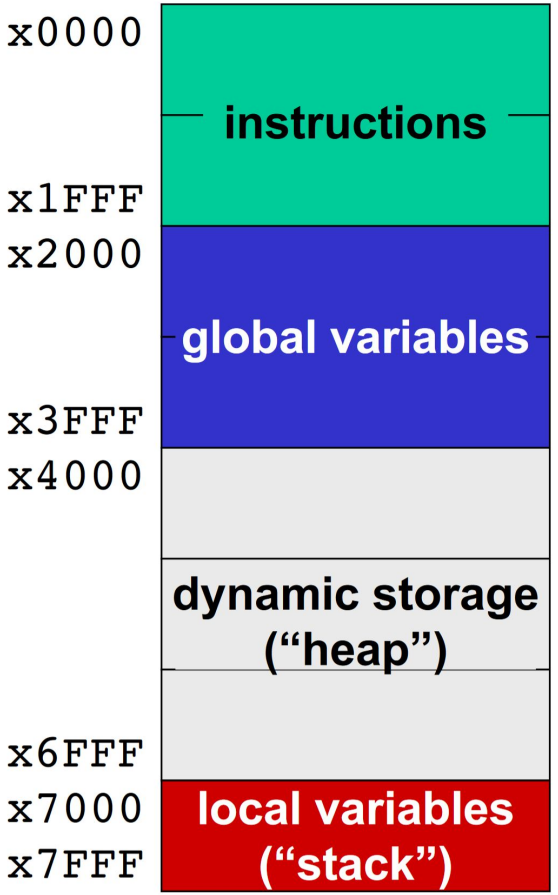
```
in_f = fopen(input_file_name)
out_f = fopen(output_file_name)
token
while readtoken(&token, in_f) not error:
   // write the assembly to out_f
   // based on what token is
```

This part is up to you, could create function(s) to handle this.

LC4 Details

The Call Stack

- A portion of memory used for keeping track of functions
- Each invocation/execution has a stack frame that is pushed onto the stack.
  - The stack frame contains the local variables, base pointer, and return address.

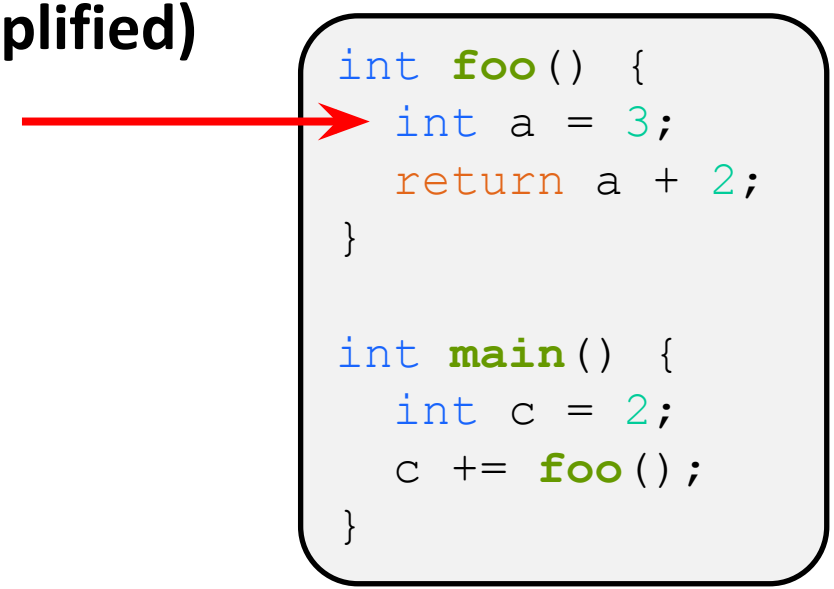


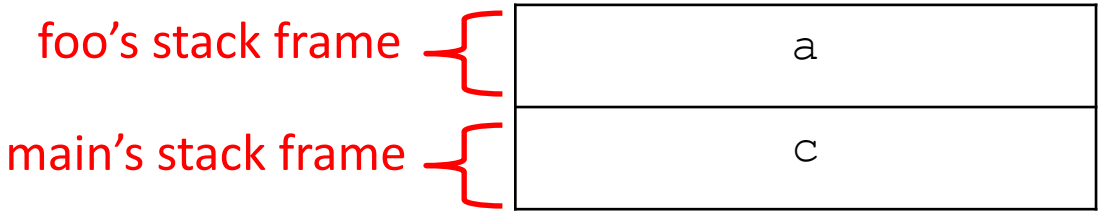
The Call Stack: C Example (simplified)

```
int foo() {
  int a = 3;
  return a + 2;
}

int main() {
  int c = 2;
  c += foo();
}
```

The Call Stack: C Example (simplified)

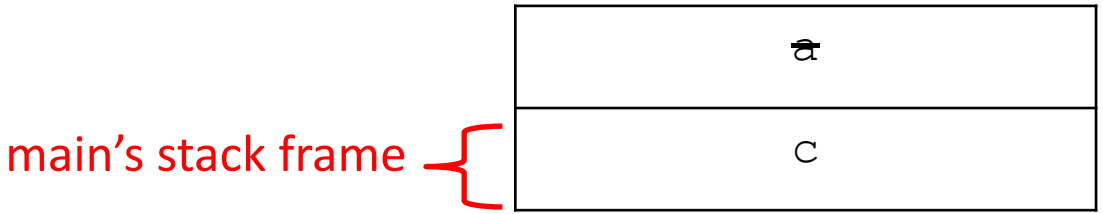




The Call Stack: C Example (simplified)

```
int foo() {
  int a = 3;
  return a + 2;
}

int main() {
  int c = 2;
  c += foo();
}
```

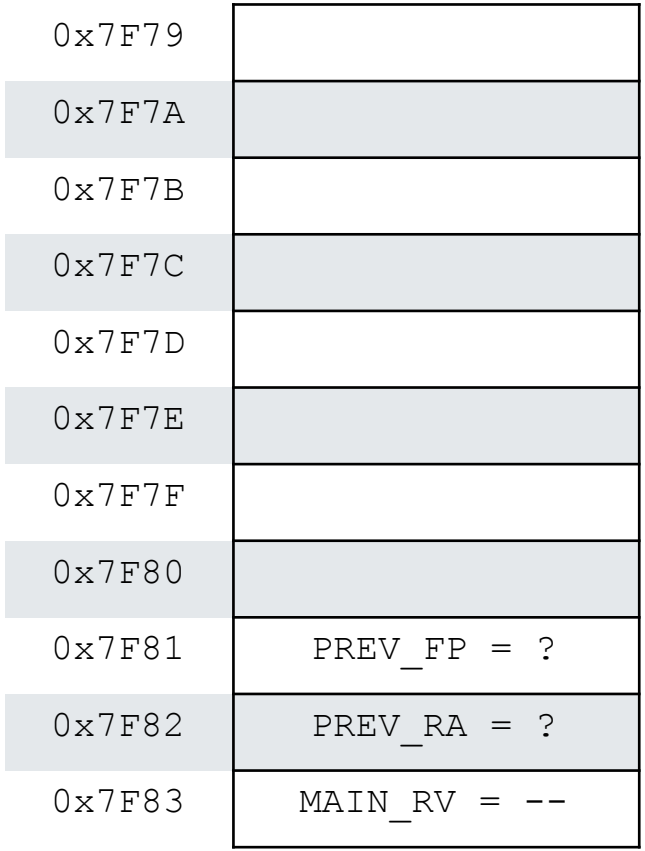


Some Acronyms

There are many things used to maintain the stack

- PC
  - The Program Counter. Keeps track of the next instruction to be executed
- SP
  - The Stack Pointer. Keeps track of the top of the stack (R6 in LC4)
- FP
  - The Frame Pointer. Keeps track of the bottom of the current stack frame. (R5 in LC4)
- RA
  - The Return Address. What to set the PC to when we return from the function so we can resume executing the calling function
- RV
  - The Return Value. The value returned from the function

The Call Stack: C Example (LC4 Details)

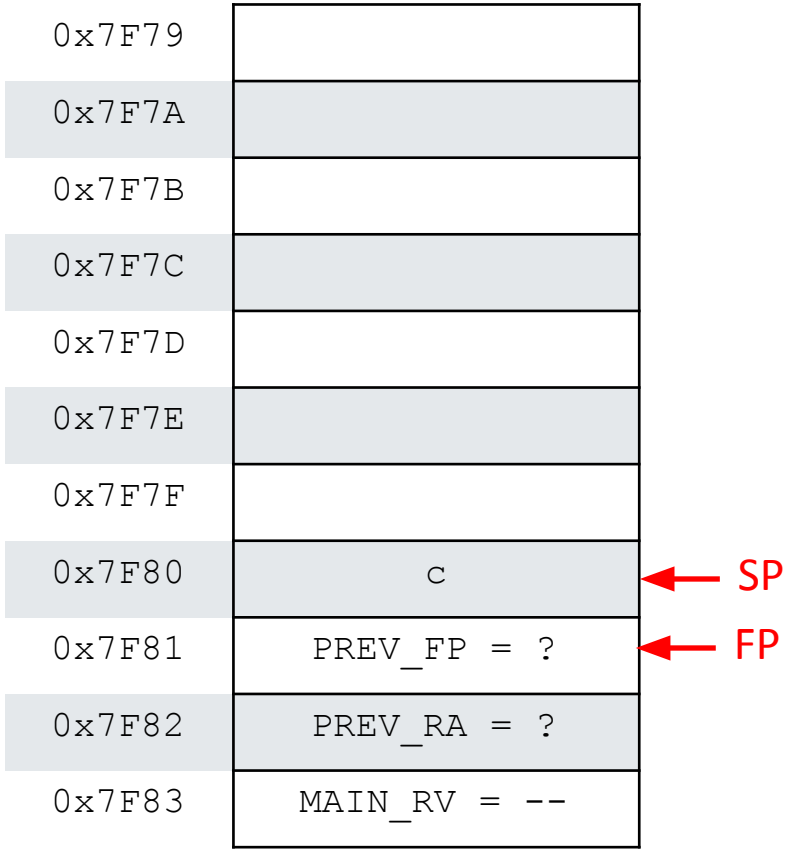



```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

PC →

The Call Stack: C Example (LC4 Details)

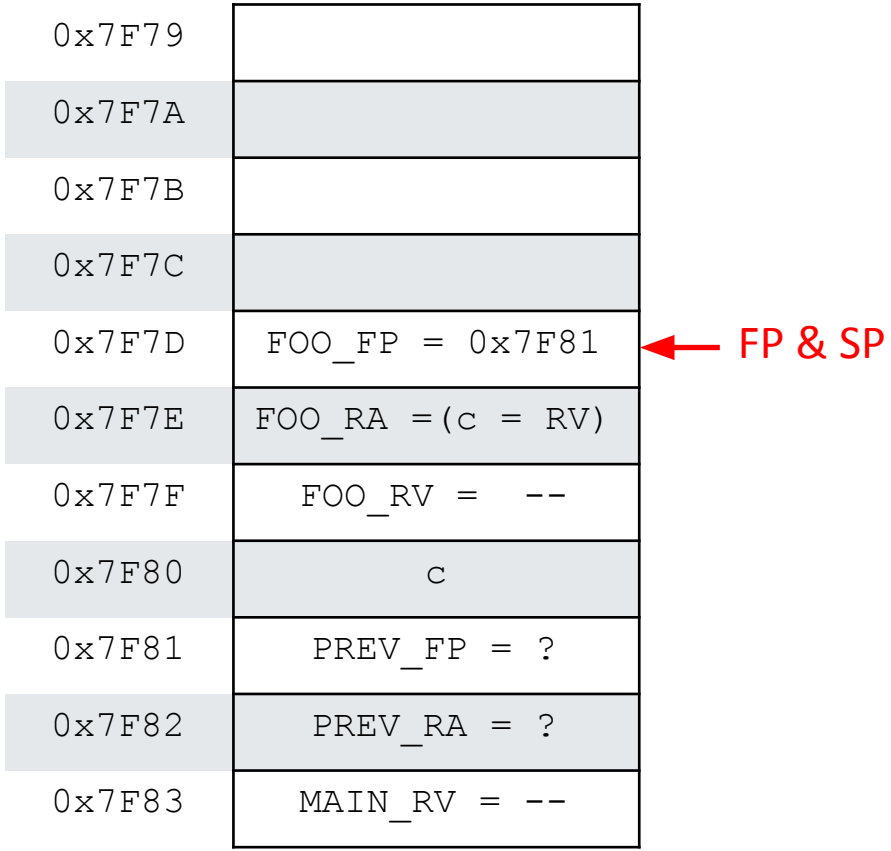



```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

PC →

The Call Stack: C Example (LC4 Details)

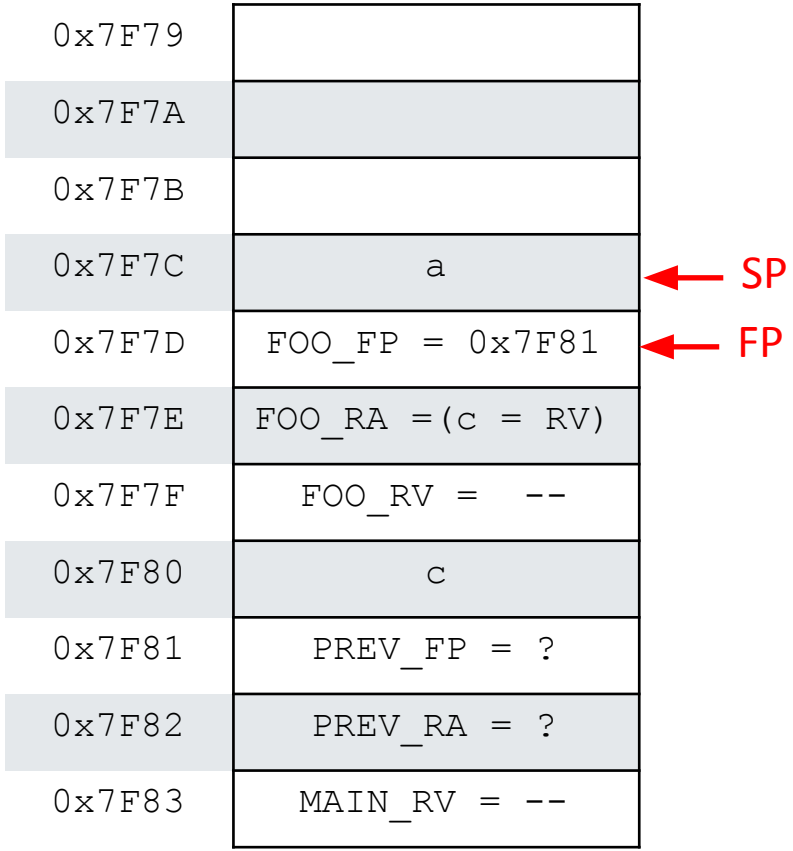


PC →

```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

The Call Stack: C Example (LC4 Details)



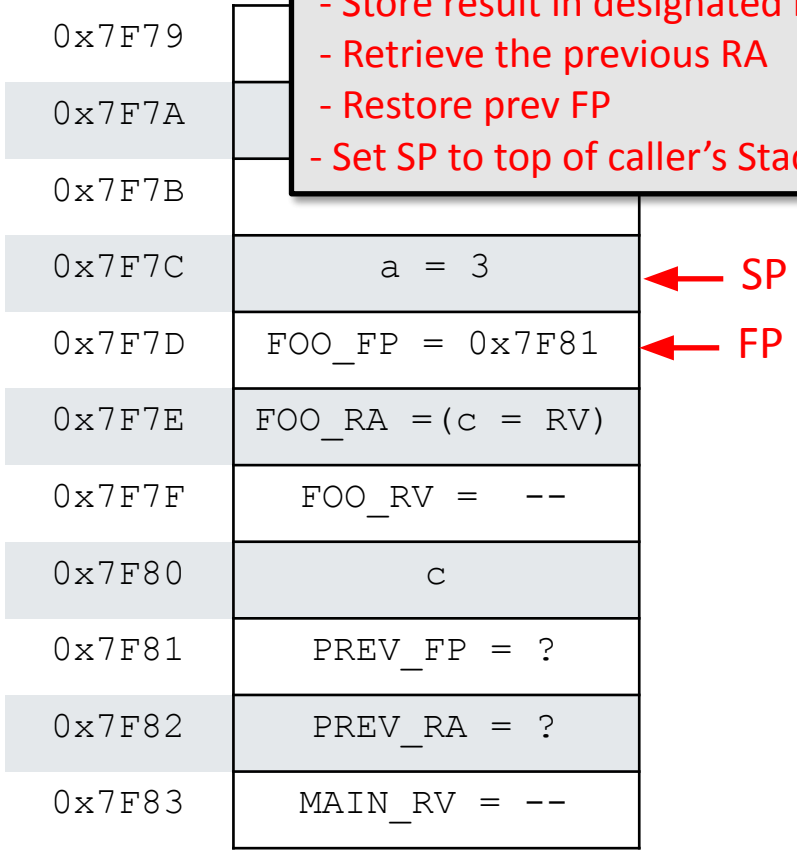
PC →

```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

The Call Sta s)

To return, we need to:
- Store result in designated RV slot
- Retrieve the previous RA
- Restore prev FP
- Set SP to top of caller's Stack frame

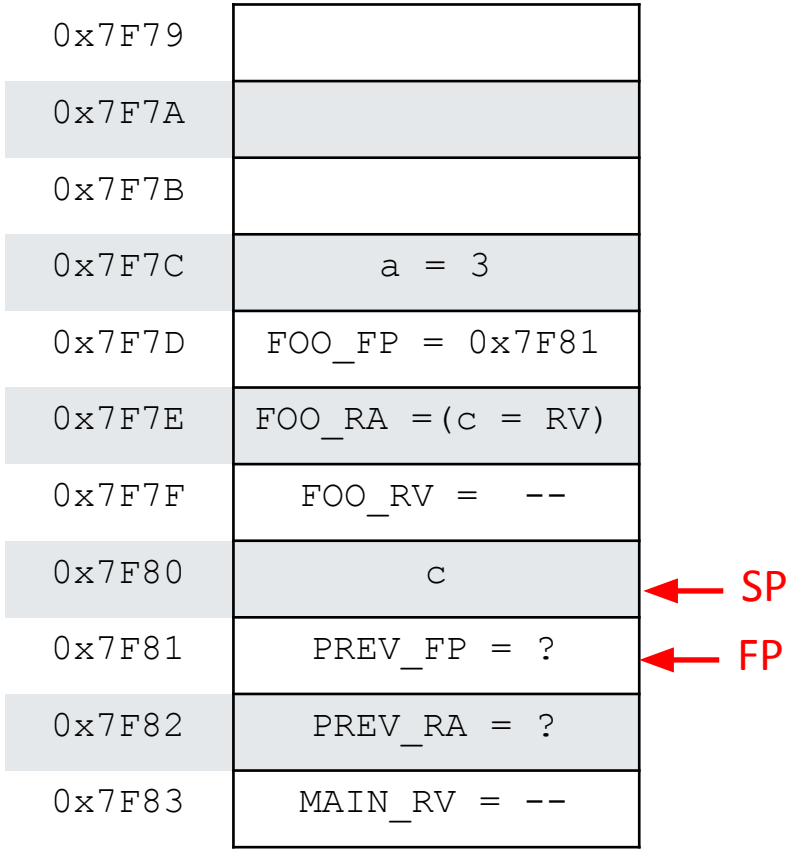


PC →

```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

The Call Stack: C Example (LC4 Details)

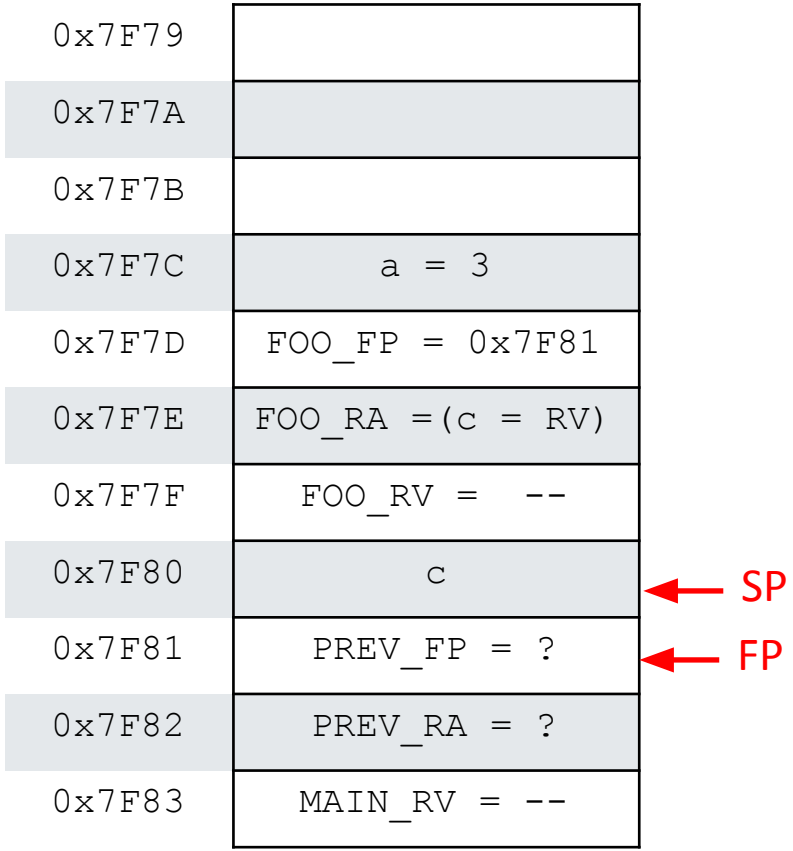



```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

PC →

The Call Stack: C Example (LC4 Details)



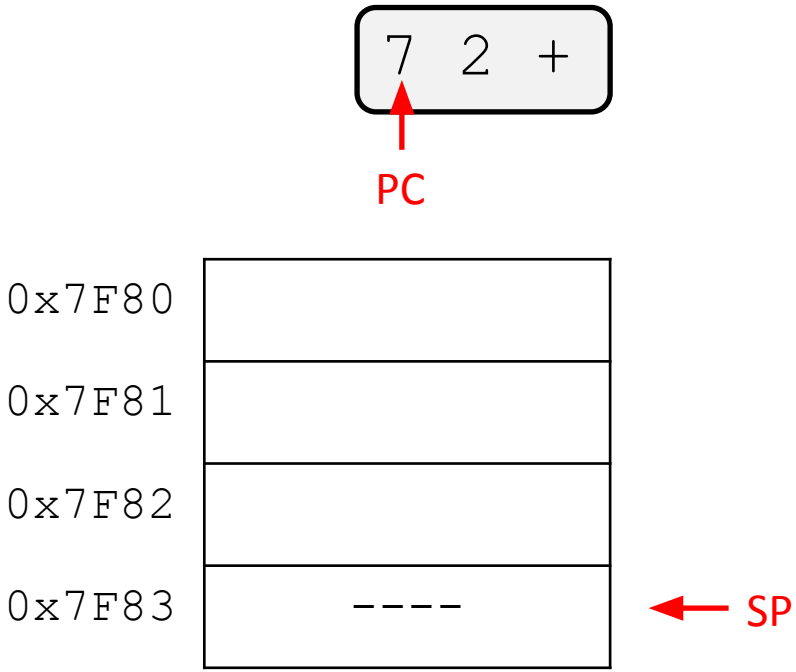

```
int foo() {
  int a;
  a = 3;
  return a;
}

int main() {
  int c;
  c = foo();
  return 0;
}
```

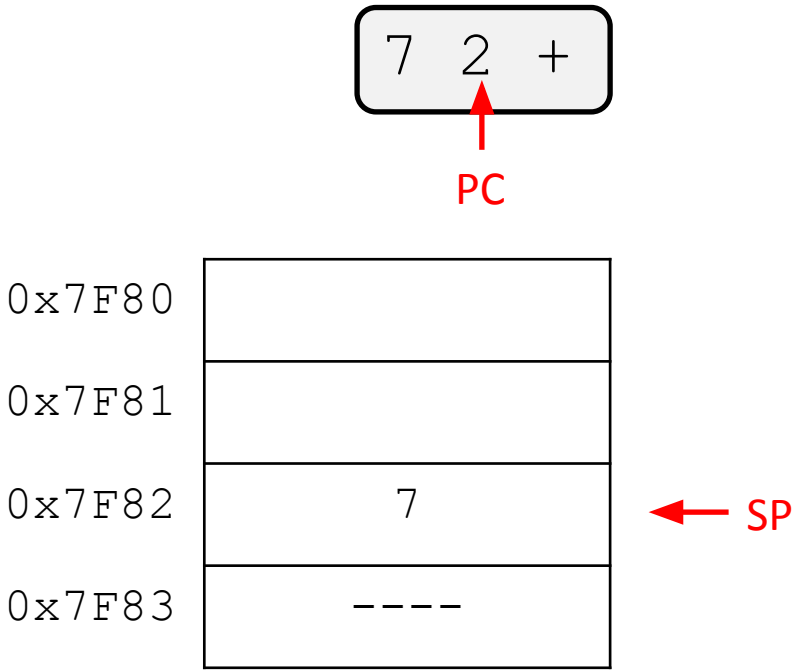
PC →

How Does This Apply To J?

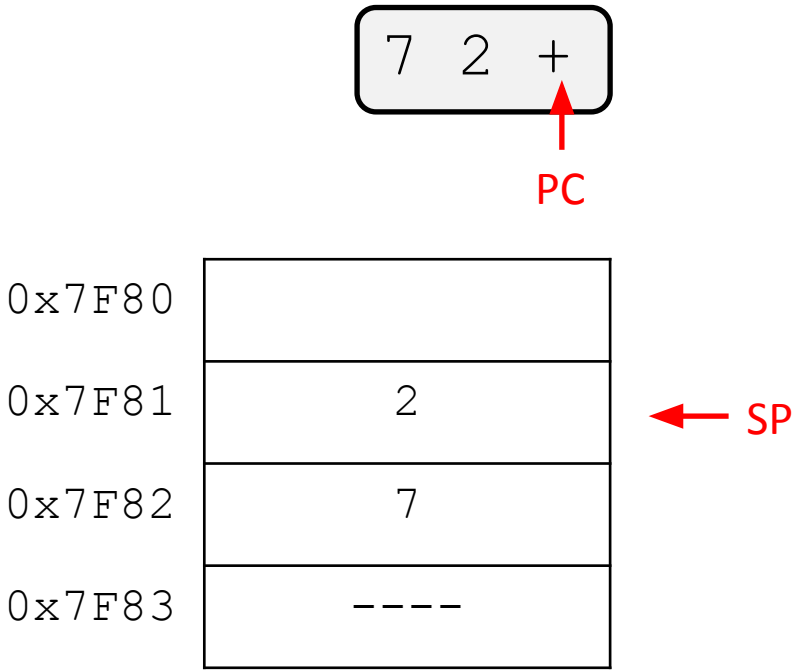
We can't give all the details, but consider the following program:



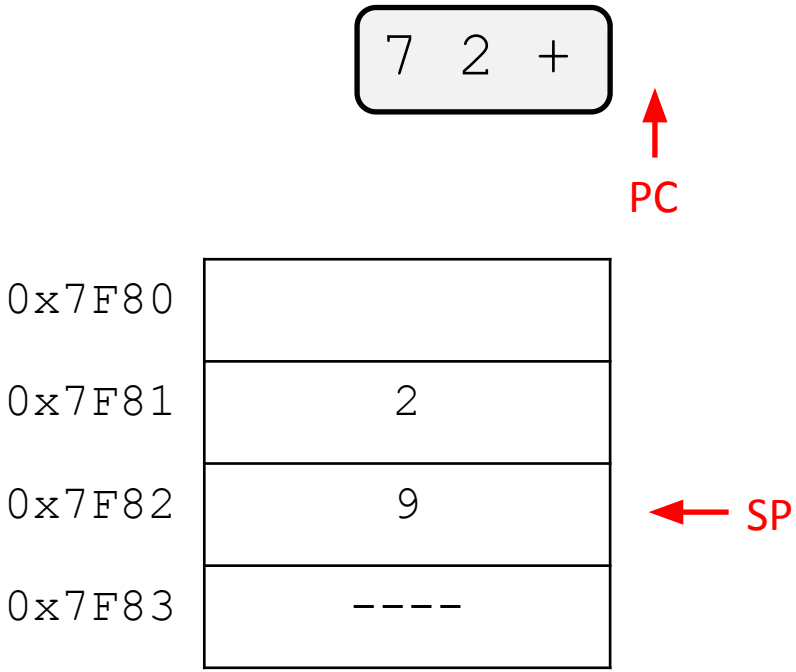
How Does This Apply To J?



How Does This Apply To J?



How Does This Apply To J?



Assembler Directives

There are two assembler directives you will need for this assignment

- .CODE
  - Tells the assembler that we are about to start a new section of instruction code. Store this in the appropriate memory segment
- .FALIGN
  - Align the current address to the next multiple of 16.
  - Necessary for functions since functions must start at an offset that is a multiple of 16 for JSR to work

Tips

This didn't cover everything

This presentation is already probably too long…

Some difficulties you may need to figure out

- We didn't actually show any LC4 instructions, just high-level ideas
  - Look to lectures and old recitations for this
- Unique labels, nested if/else/endif, literals that need const & hiconst, etc.

Unique Labels

Remember, you cannot use offsets for JMP or BR instructions when writing LC4. You must use labels

You will need to use labels to implement IF/ELSE/ENDIF and WHILE

- What happens if you have many IF/ELSE/ENDIF blocks?
- Will need unique labels for each block
- Solution: Keep a counter and have the labels be variations of IF_1, ELSE_1, ENDIF_1, etc.

Nested If/Else/Endif

- What happens if there are many nested if/else/endifs?
- What if some of them are if/endifs?

```
2 3 4 - if - if 2 endif 1 else 0 endif
```

- There are two main approaches to handling this.
  - Using recursion
  - Using a stack data structure similar to HW6
    - **THIS IS NOT THE SAME AS THE CALL STACK IN LC4. THIS IS A C DATA STRUCTURE THAT WOULD BE USED IN YOUR CODE TO GENERATE THE LC4.**

Testing

- It is probably worth testing your `read_token` implementation (but not required)
- Write a short program that continuously reads tokens from a file and prints them out
- Test them on ALL provided test cases to make sure that it works

Testing

To test the program the final program, do the following

- Run your program on a .j file to create the corresponding .asm file
- Use PennSim to run the test case. Make sure you use the provided script and have the necessary files (e.g. os.asm)
- Check to see if the output is the same

Error Checking

- We will largely testing correct .j files
- It is still a good idea to add error checking to make sure you are handling things correctly
  - There shouldn't be a `defun` or `return` in the middle of an IF/ELSE/ENDIF block.
  - You shouldn't run into any BAD_TOKENS
  - The token after a `defun` *should* be an `ident` used for the function name

That's all we have for today!

Reminders:

- TA-lead recitations will take place on
  - Tuesdays 6:30-8:00pm in Moore 100A
  - Wednesday 12:00-1:30pm in Moore 100C
- HW10 is due 12/2 at 11:59pm (checkpoint for assignment)
- HW11 is due 12/9 at 11:59pm