EEE3096S: Embedded Systems II

LECTURE 14:

ARM ASSEMBLY PROGRAMMING (PART 3)

Presented by:

Dr Yaaseen Martin

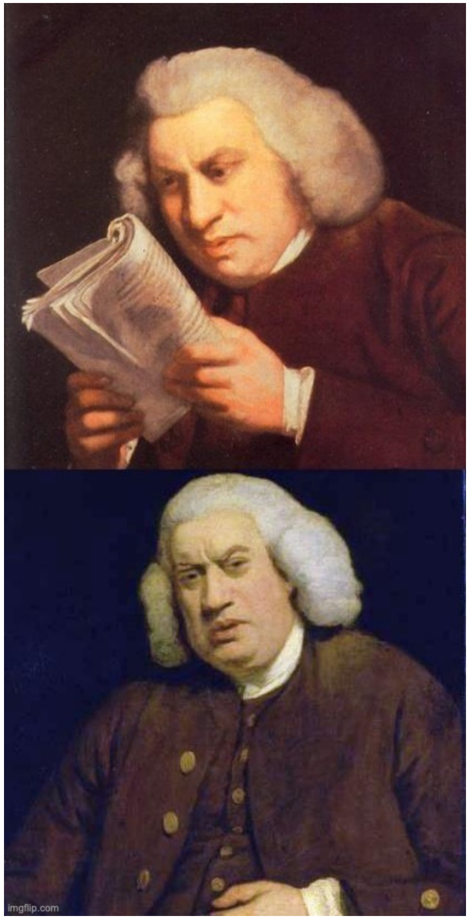
Electrical Engineering
University of Cape Town

SIGNED VS UNSIGNED: QUICK RECAP

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4	=	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

A WILD READING ASSIGNMENT APPEARS!

- Read up on object and executable file formats
- Think about how computers store *programs* on disk and in memory
- Read over file on Amathuba: **Reading on Object Files.pdf**



COMMON (BAD) ASSUMPTIONS IN ASSEMBLY

- Avoid making these common assumptions…
  - “The hardware probably works – so I'll write the code, then test the board and code together”
  - “I can skip commenting this as it’s super obvious”
  - “Changing these two lines won't affect the rest of my program”
- Incorrect assumptions can prove costly and time consuming.

OTHER COMMON ASSUMPTIONS

- "Everything is plug-and-play… surely this is too 🤓"
- "This peripheral will surely work fine when I connect it 🙂"
- "My board must be broken because all my connections are perfect 😎"
- "My system must be bad because my code is flawless and yet it still doesn't work. Life is nothing but pain 🥲"

STRATEGIES FOR C & ASSEMBLY DEVELOPMENT

- When programming in C, take **small** steps
  - Think after every step
  - A bit of thinking is cheaper and faster than debugging and rewriting
- When programming in Assembly, take ***tiny*** steps
  - Assembly language programs are subtle and quick to anger

MIXING C AND ASSEMBLY

WHY MIX C AND ASSEMBLY?

- Interfacing an Assembly program with C **libraries** (or vice-versa)
- **Performance** – writing speed-critical key parts of a program in assembly, and leaving the rest in C, may significantly boost overall performance
- **Size** – human-written Assembly can be smaller than compiler-generated code
- *Main reason:* because nowadays, if you write any Assembly then it's usually a tiny piece of your (larger C) program that does something very **fast** and **specialised**

FUNCTIONS IN ASSEMBLY

- You can't specify a **function prototype** (function declaration specifying its parameters and return type) in the assembler
- You need to ensure that the Assembly implementation matches the prototype the C compiler is using
- The interface between a C function and an Assembly function can be a good hiding place for bugs – some care is required
- You need to ensure you are using the same **Application Binary Interface (ABI)** as is being used by the C compiler

WHAT IS AN ABI?

- An Application Binary Interface (ABI) is an **interface between two binary program modules*** (usually functions)
- However, 'module' is used in this case because it might not *always* be functions; could be communicating in some way with a control element (e.g. mutex, I/O or data structure)
- Adhering to an ABI is usually the responsibility of the **compiler**, but the application programmer may have to deal with this directly if writing Assembly language to connect to compiled high-level code (e.g. C)

* Further details of this concept at https://en.wikipedia.org/wiki/Application_binary_interface

CLARIFICATION: API VS ABI

- **API = Application Programming Interface**
- **API** is **higher-level** and defines the interface between software components at the source code level; usually a list of functions and data structures available in the application library
- **ABI** is **lower-level**, defining the binary or machine code connection to modules in the application; think of it as the “compiled” version of an API, or an API on the machine-code level

DEVELOPING AN ASSEMBLY FUNCTION

FUNCTIONS IN ASSEMBLY

The assembler has some commands which are used to support interfacing:

.global *name* e.g. .global testfunc	@ Indicates that *name* is a **global symbol** (accessible from other modules)
.type *name*, function e.g. .type testfunc, function	@ Indicates that the symbol *name* is a **function**
***name*:** e.g. testfunc:	@ Tells assembler to create a symbol with *name* at this position (i.e. a **label**)

FUNCTIONS IN ASSEMBLY

In a C header file (e.g. *asm_module.h*):

char *strcpy(char * a, char* b); *// Function prototype*

In the Assembly language listing:

.global strcpy

.type strcpy, function

strcpy:

…Assembly implementation of the function…

DEALING WITH PARAMETERS

• The first 13 input parameters are passed to the function using registers **r0 to r12**:

```
char *strcpy(char *out, char *in);
```

r0 r1

DEALING WITH RESULTS

- The return result is returned in **r0**:

```
char *strcpy(char *out, char *in);
```

r0

CALLING CONVENTIONS

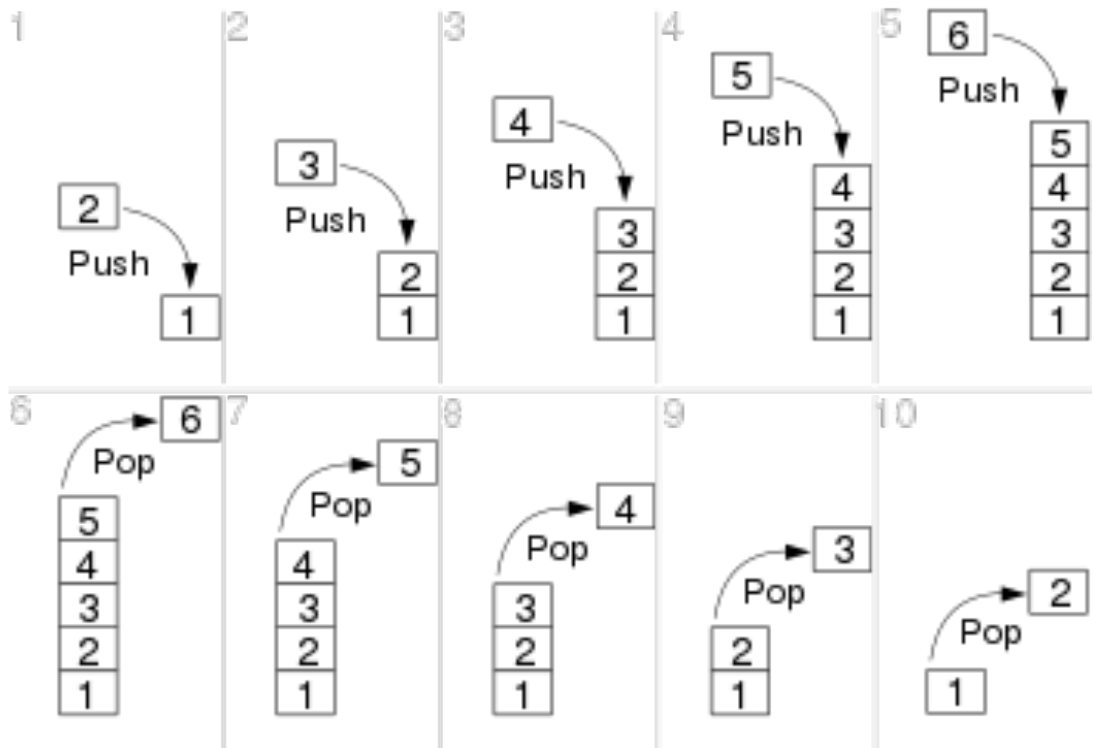
Calling Convention*:

The way that parameters, return values, local variables and return addresses are handled.

- Different platforms and operating systems use different conventions
- The calling convention is part of the ABI being adhered to
- We will describe the calling convention GCC uses on ARM
- The calling convention is broken up into two parts:
  - Function prologue: happens at the start of the function and sets up the stack for local variables
  - Function epilogue: happens at the end of the function and frees up local variables before returning to the calling function

* The calling convention is just one of potentially many things defined in an ABI

THE STACK

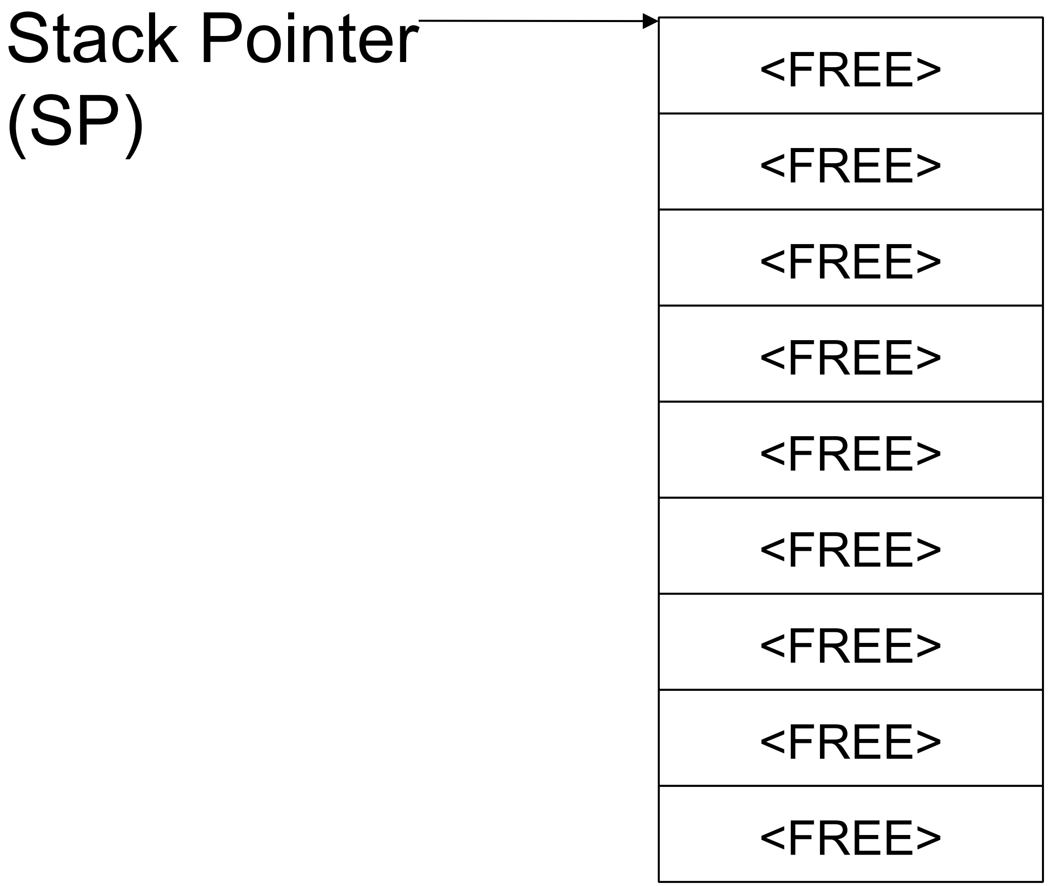


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

STACK FRAMES

- The stack space allocated to a specific function is called a **stack frame**; each function builds "its own stack" on top of the main stack
- The stack frame contains parameters, local variables and the return address
- At the start of the function, **Frame Pointer** (FP) = SP
- For ARM, the **stack** grows <u>down</u> and the **heap** (for dynamic memory) grows <u>upwards</u>
- SP points to the next free location in the stack

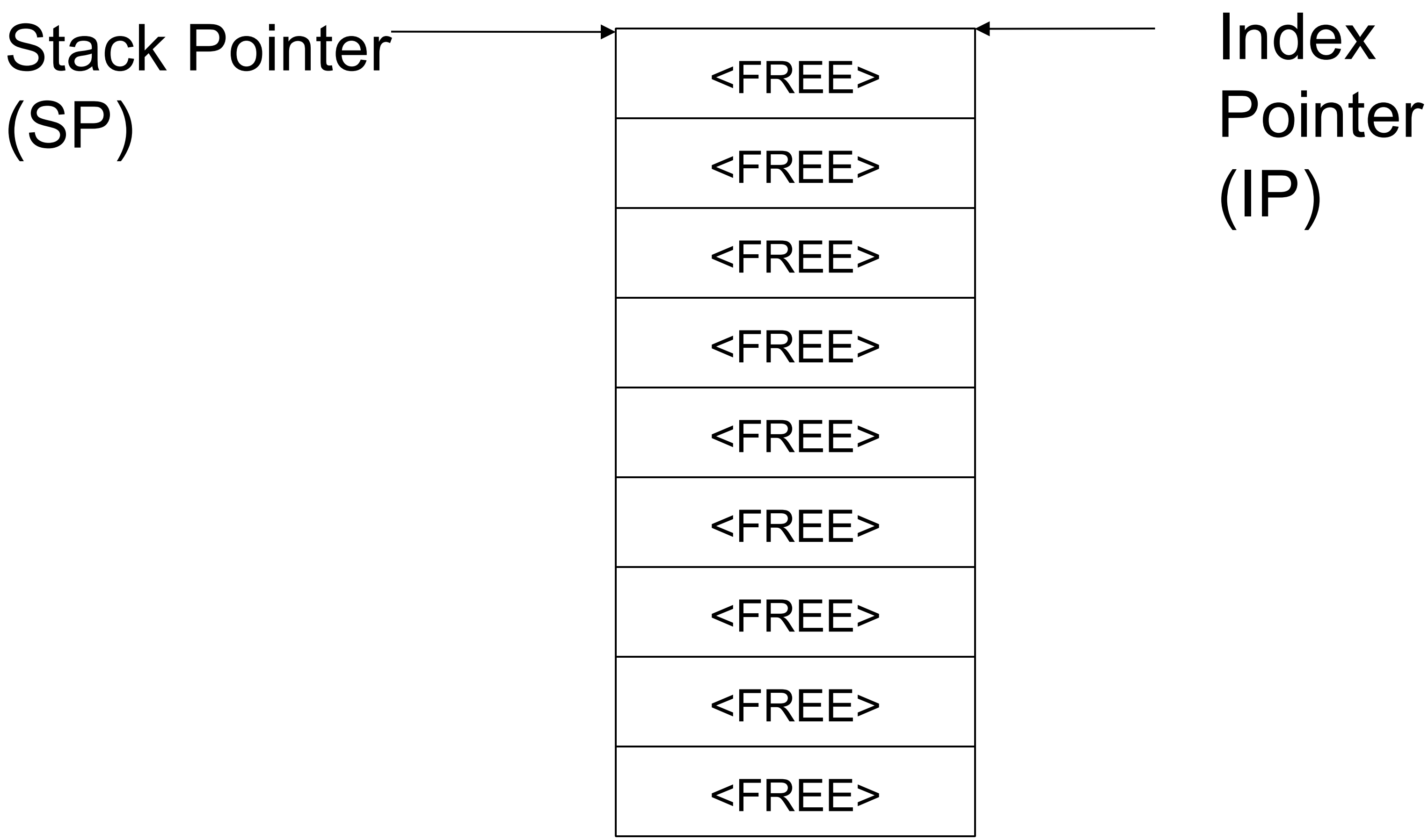
STACK FRAME EXAMPLE: PROLOGUE



Instruction: **bl** strcpy

The above may translate to bl =strcpy, loading the 32-bit value from a near-by memory location into the pc register

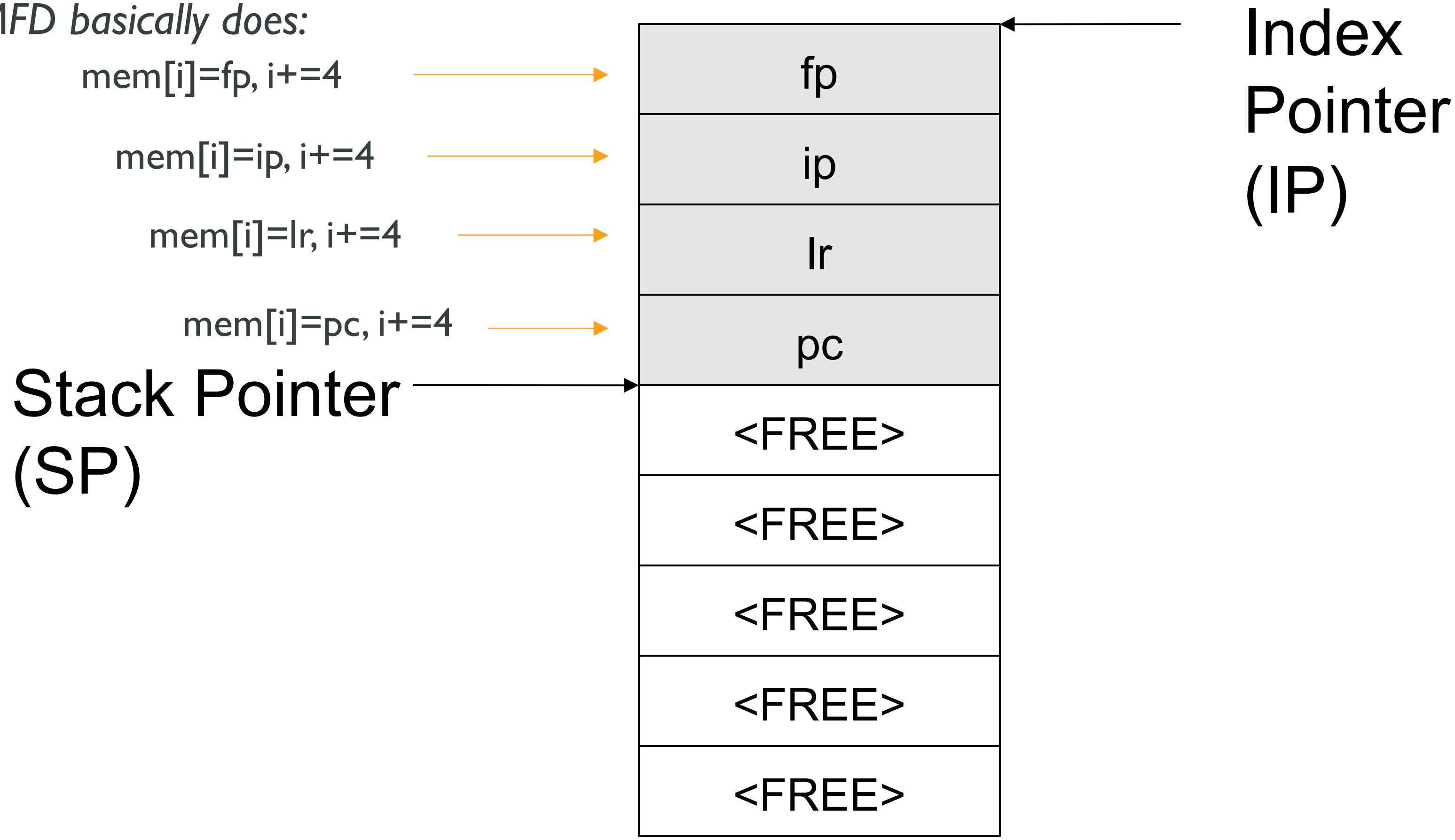
STACK FRAME EXAMPLE: PROLOGUE



Instruction: **mov** ip, sp

STACK FRAME EXAMPLE: PROLOGUE

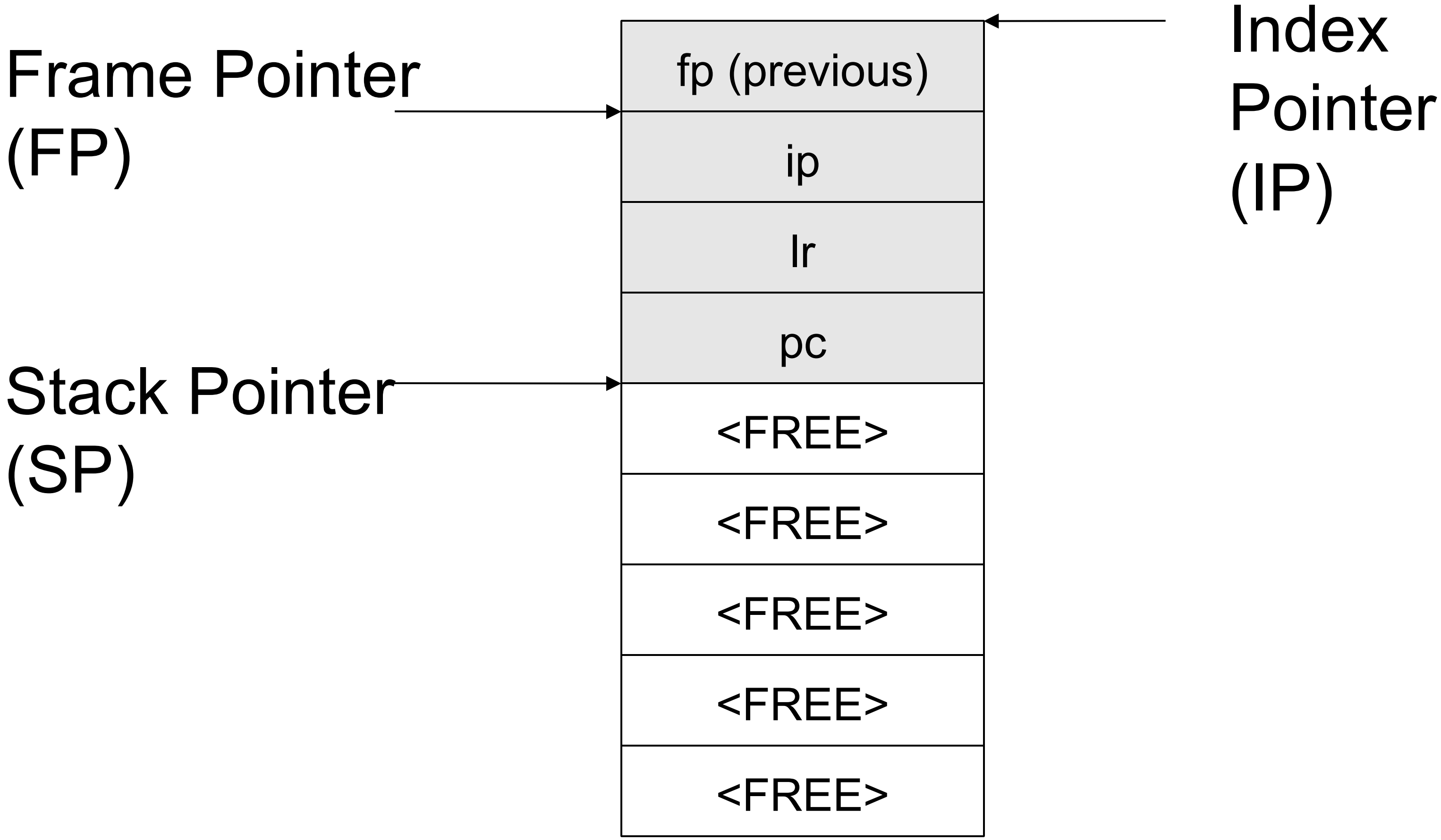
STMFD basically does:



Instruction: **stmfd** sp!, {fp, ip, lr, pc}

i.e. stmfd is the special ARM instruction 'Store Multiple Decrement before Full Descending'.… This is essentially a PUSH instruction and in some assemblers, you can use push. This is a special instruction to help with setting the stack frame.

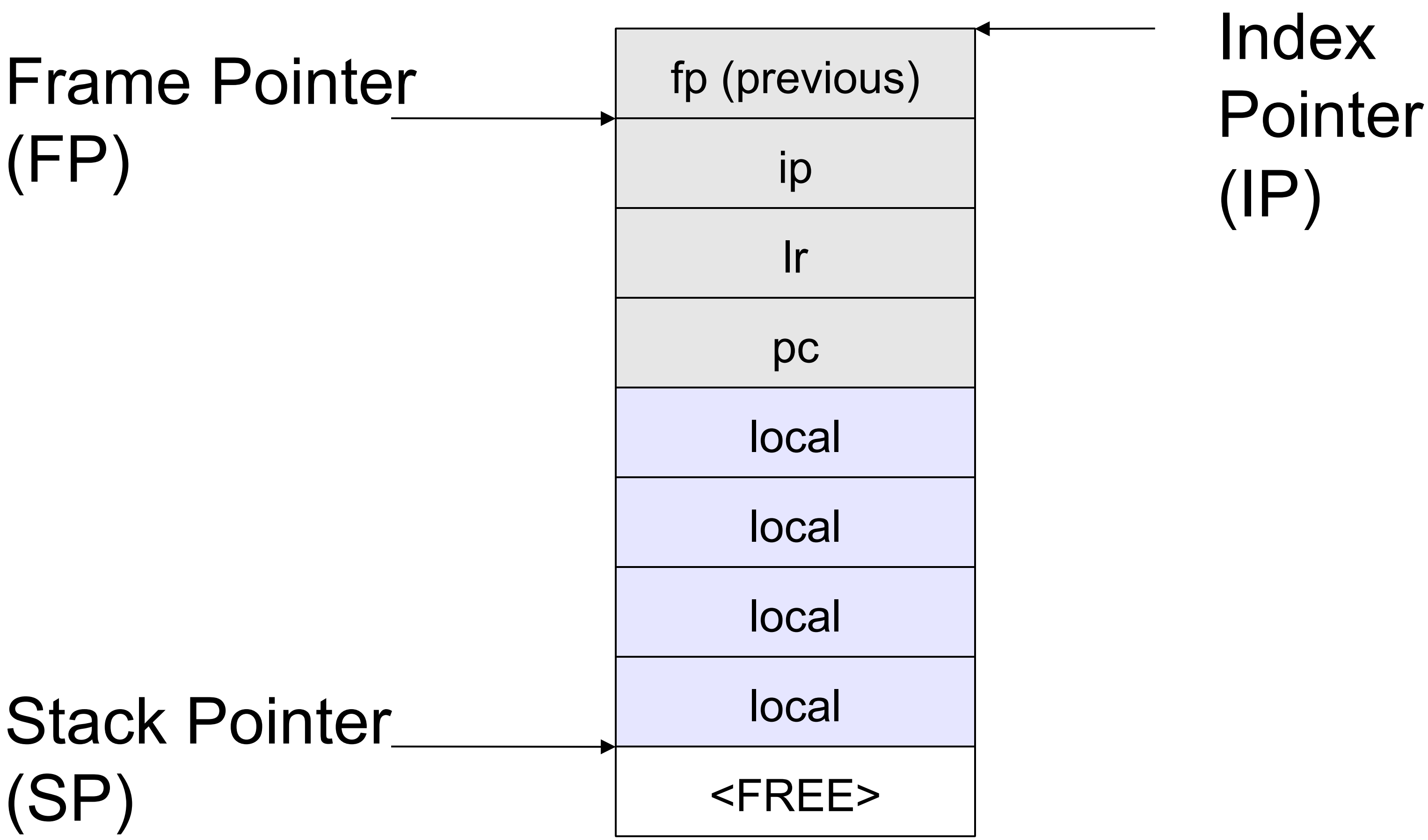
STACK FRAME EXAMPLE: PROLOGUE



Instruction: **sub** fp, ip, #4

*i.e. the new frame pointer is set to the address word after the index pointer; this is done as an LDR instruction would read the 32-bit data **above** FP, which would belong to the previous function's stack frame. Also note that ARM uses Little Endian by default, so the LSB is stored at the lowest address.*

STACK FRAME EXAMPLE: PROLOGUE



Instruction: **sub** sp, sp, #16

i.e. space for local variables are declared by moving sp ahead by the amount needed

ASSEMBLY FUNCTION PROLOGUE

strcpy:

@Save sp

index pointer

mov ip, sp

@stmfd: store multiple

stmfd sp!, {fp, ip, lr, pc}

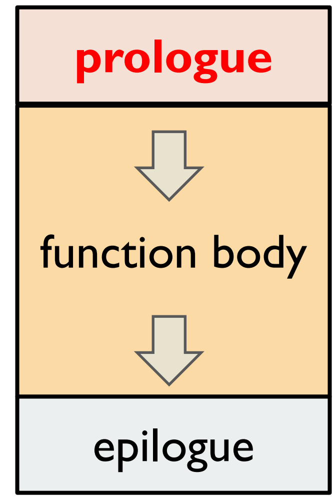
@fp points to old fp on stack

sub fp, ip, #4

@allocate 4 more words on stack

sub sp, sp, #16

i.e. starting the function



The ARM instruction:

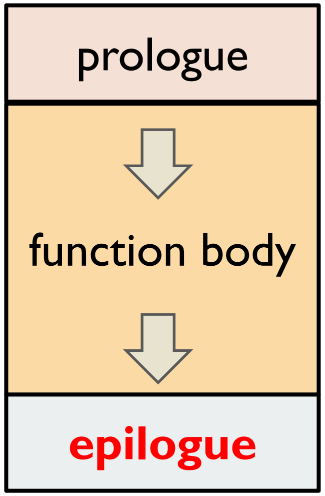
```
stmfd sp!, {r1, r2}
```

corresponds to C code:

```
unsigned int* sp;
unsigned int r1,r2;
*sp = r1; sp++;
*sp = r2; sp++;
```

ASSEMBLY FUNCTION EPILOGUE

i.e. ending the function



strcpy:

(…previous code…)

@pop registers stored on stack

@starting at fp (which we set in prologue)

@ldmea = pre-decrement load

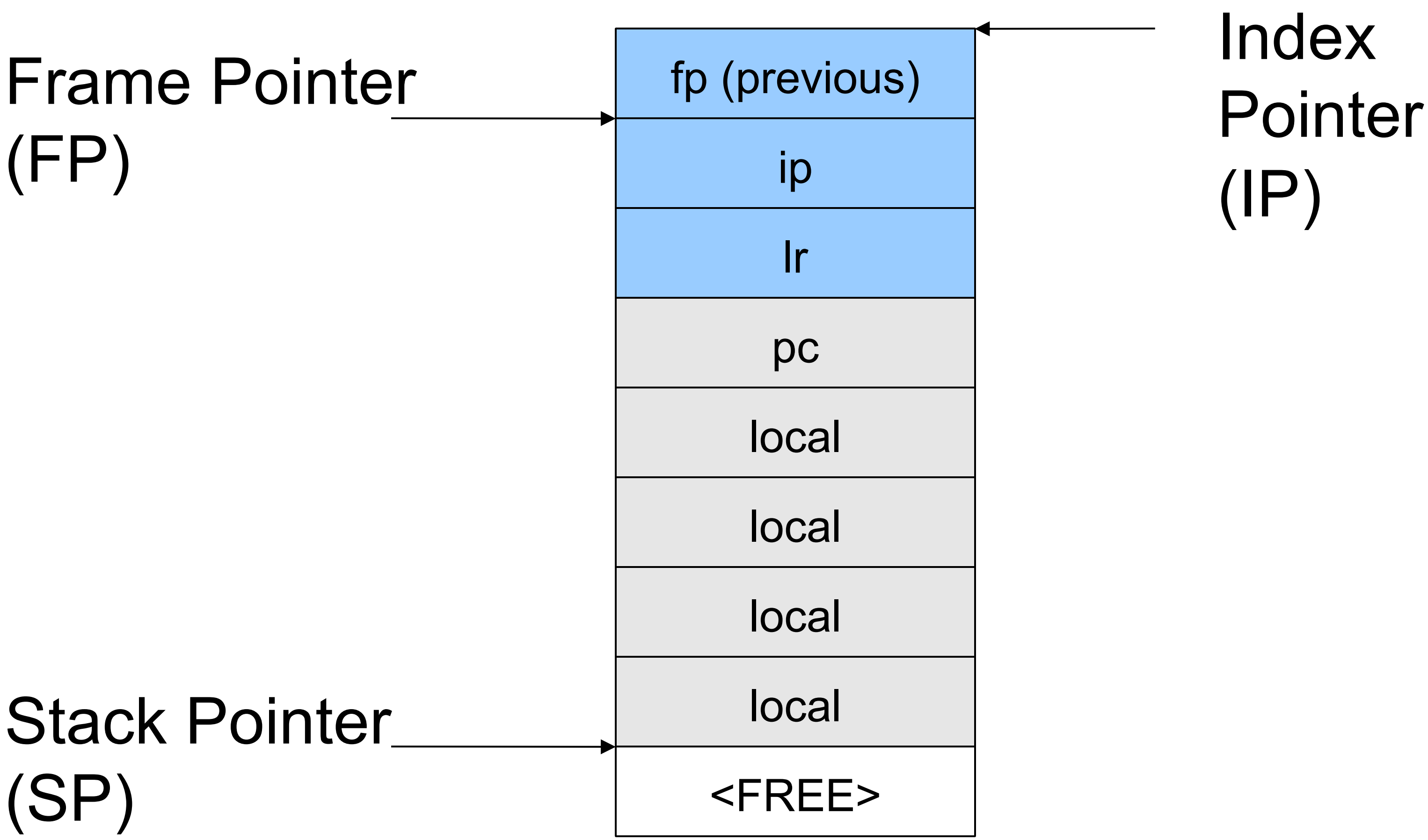
ldmea fp, {fp, sp, pc}

@ note how value of lr gets loaded into pc, undoing the effects of **BL** and returning from the function

BL = Branch and Link, the instruction used to call a function

BL rx does: lr = pc+4, pc = rx, and continues execution from address rx, where rx is a register value. You can also branch relative to the pc (pc = pc+*x*)

STACK FRAME EXAMPLE: EPILOGUE



Instruction: **ldmea** fp, {fp, sp, pc}
Restores the frame pointer as it was in the calling function

SAVING LOCAL VARIABLES

There are **two** basic ways in which you can save **local** variables so that a given function does not change local variables of the calling function:

- **Store Before Call (SBC)**
  (possibly more optimal)

OR

- **Call Before Store (CBS)**, a.k.a.
  Store Before Use (SBU)
  (possibly less optimal
  but more reliable)

(not to be confused with BCS:)

SAVING LOCAL VARIABLES

Store Before Call (SBC):

- In this case, the code *before* a function call is designed so that **all** local variables are **saved** to memory **before** the function call is made
- This used in some implementations of the -02 compiler optimisation level

E.g.: Use registers… (e.g., use of r0-r7)
… might have finished with r0-r3
Store registers (currently in use) into memory (str r4-r7)
call_function () // uses 8 registers, r0-r7
Restore registers that were in use (ldr r4-r7)
FUNCTION INVOCATION OVERHEAD = 4 reg * 2 str/ldr = 8 memory accesses

Hint: take note of what the function invocation overhead is and how it is calculated, as this is a useful thing for considering the optimality of your code, and also a favourite type of test question!

SAVING LOCAL VARIABLES

Call Before Store (CBS):
In this case, the code at the start of a function stores all the registers to be used in the function, whether or not the registers were actually used by the calling function

E.g. Use registers… (e.g., use of r0-r7)
… might have finished with r0-r3
Only some registers currently in use (i.e. r4-r7)
call_function () // uses 8 registers, r0-r7
  store r0-r7
   … do body of function
  load r0-r7
  return
Continue using r4-r7 in calling function
FUNCTION INVOCATION OVERHEAD = 8 reg * 2 str/ldr = 16 memory accesses

i.e., in this example, '*call before store*' used 8 more memory accesses than *store before call*; i.e. less efficient

INLINE ASSEMBLY

READ

- Assembler code can be included in C code
- Useful for accessing hardware or CPU features that are not exposed by the C language
- GCC uses the ***asm*** keyword, i.e.:
  asm("instruction": outputs: inputs);
- Other compilers use other keywords and syntax
- Code with inline asm is **not** ANSI C, so many people do not like using it at all

GCC INLINE ASSEMBLY

READ

Example – Rotate a value right by one bit

int rotRight(int val) {

int result;

asm(“mov %0, %1, ror #1”

: “=r” (result)

: “r” (val)); // i.e. result = val ror #1

return result;

}

Great document on inline ASM can be found here:

http://www.ethernut.de/en/documents/arm-inline-asm.html

PROGRAMMING EXAMPLE (FOR READING)

READ

CONCATENATE STRINGS

- We are going to write a function which adds one string to the end of another
- Similar to one in the C library called strcat

A REAL-WORLD EXAMPLE: CONCATENATING STRINGS

```
// Concatenate "in" to the end of "out"
char *strcat(char *out, char *in) {
       int i = 0, j = 0;
       while (out[i])
              // Go to end of out
              i++;
       while (in[j])
              // Copy characters from in to out
              out[i++] = in[j++];
       out[i] = 0; // Terminate out string
       return out; // Return the out pointer
}
```

DEFINE THE C PROTOTYPE FOR THE ASSEMBLY CODE

- You can tell C about the Assembly module by putting a prototype of the function into a *.h* file (C header file). We are effectively creating a module (strcatx.h) and the Assembly file is strcatx.s (*.s* being the extension for ARM Assembly)

```
// File strcatx.h:
// Prototype declaration of strcat
   char *strcat(char *out, char *in);
```

- Also, since we want to concatenate strings using ASCII, we only need to consider **1 byte** at a time since each letter only needs 8 bits; e.g. the letter “A” is 01000001 in binary

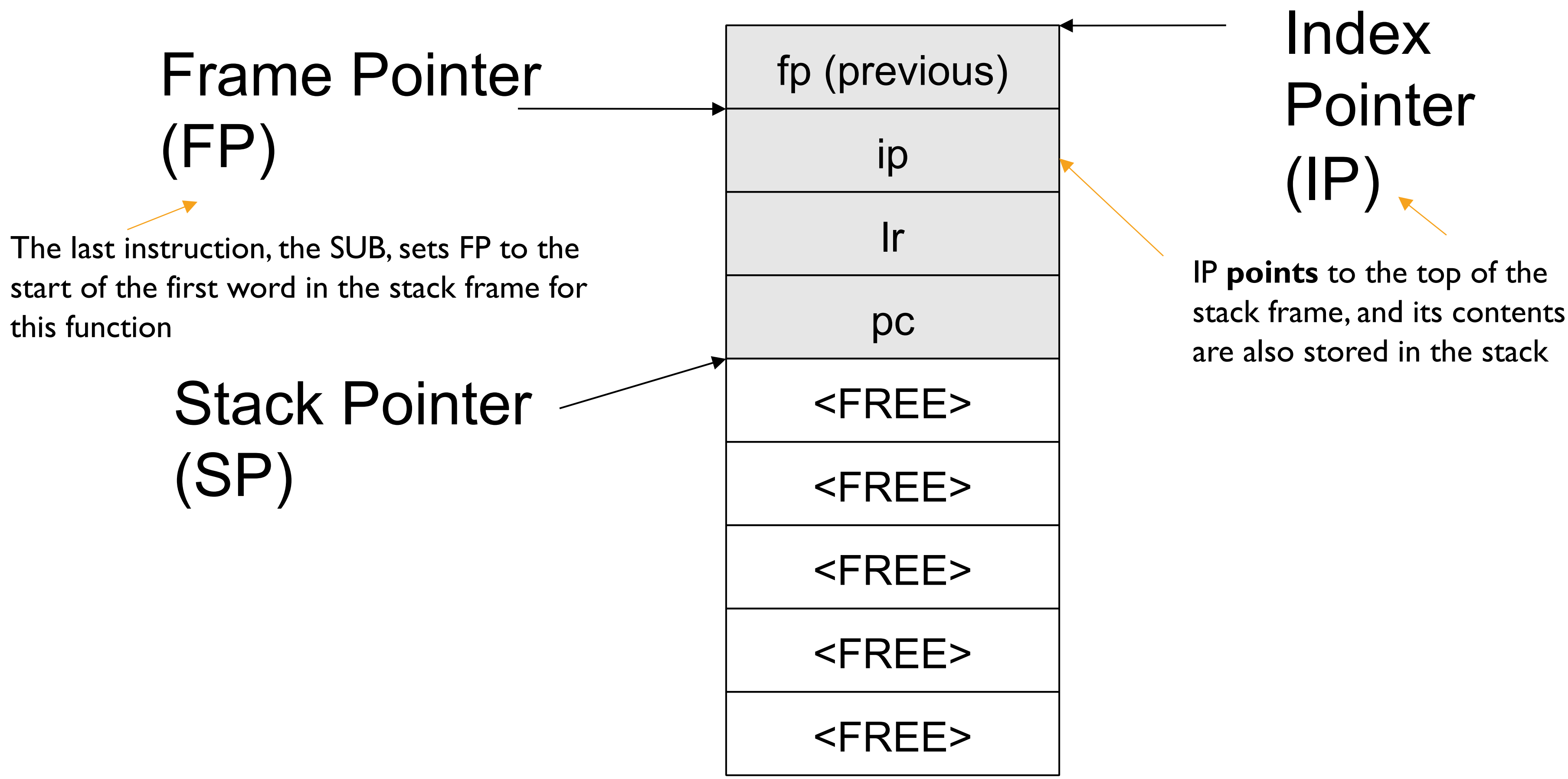
FUNCTION PROLOGUE

C	char *strcat(char *out, char *in) {
ASM	.text .align 2 .global strcat .type strcat, function strcat: **mov** ip, sp **stmfd** sp!, {fp, ip, lr, pc} **sub** fp, ip, #4

<u>Note</u>: *out* is mapped to r0, *in* is mapped to r1; since each of these is a char*, the **pointer/address** is saved into the registers — **not** the actual value of the parameter

We are clearly expecting 'store before call' method to be used… the 'call before store' would simply be adding r0-r4 into the register list in the stmfd instruction.

STACK FRAME VIEW (END OF PROLOGUE)



No space for local variables yet

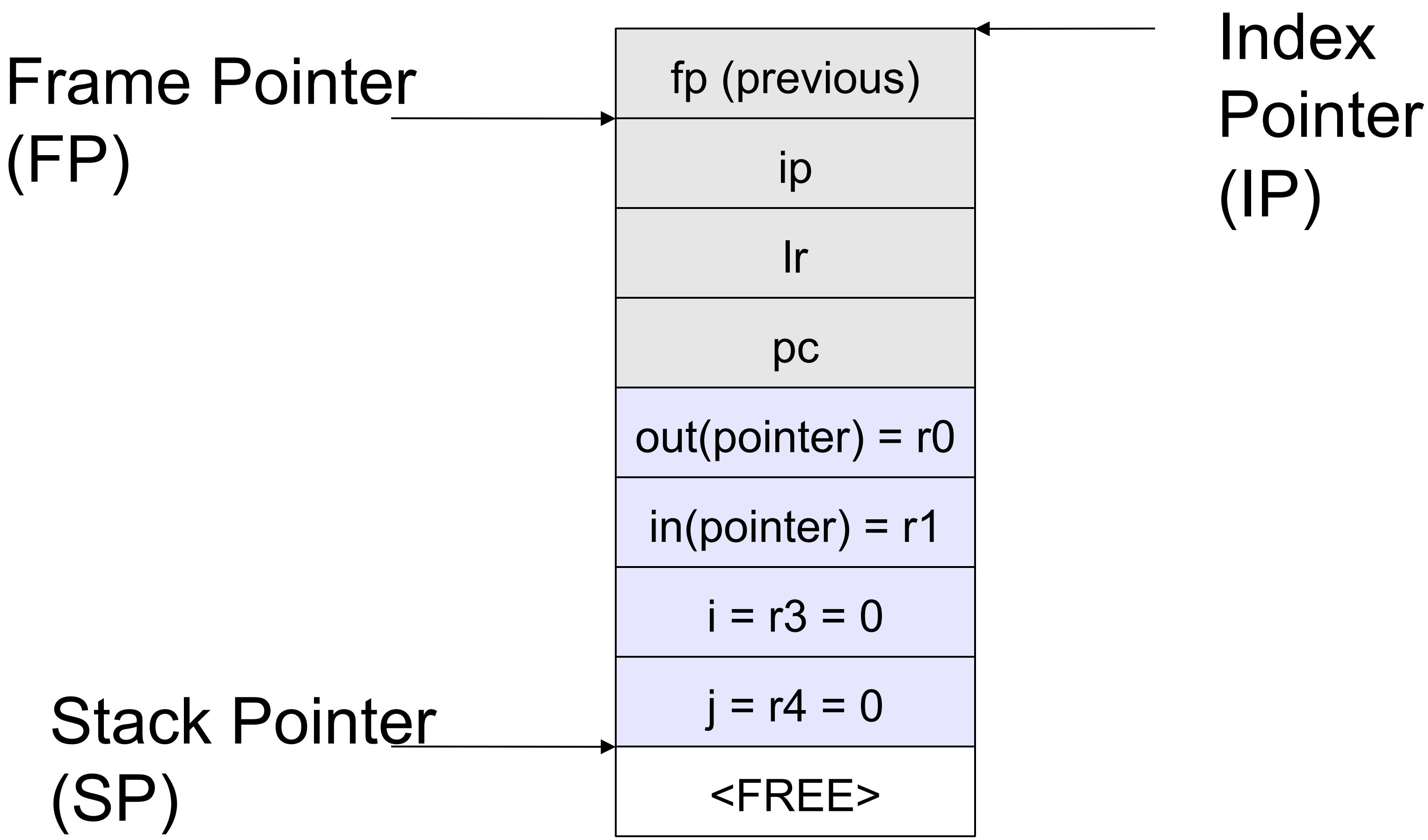
DECLARE LOCAL VARIABLES

C	int i = 0, j = 0;
ASM	

```
sub sp, sp, #16        @ Space for 4 locals below
str r0, [fp, #-16]     @ [fp - 16] = out (pointer)
str r1, [fp, #-20]     @ [fp - 20] = in (pointer)
mov r3, #0
str r3, [fp, #-24]     @ [fp – 24] = i
mov r4, #0
str r4, [fp, #-28]     @ [fp – 28] = j
```

We directly assign values to the registers (we're assuming 'store before call') and could copy them through to the stack memory, but this isn't so optimal unless these registers are needed for other things before being used in the body of the function.

STACK FRAME VIEW (LOCALS DECLARED)



Space and values declared for local variables;
values are in the registers and also copied to stack memory

START PROCESSING

C	while (out[i]) i++;
ASM	ldr r0, [fp, #-16] @ r0 = out (pointer) ldr r3, [fp, #-24] @ r3 = i loop1: ldrb r2, [r0, r3] @ load byte (i.e. out[i]) in r2 cmp r2, 0 @ r2 == 0? beq next @ yes → go to next add r3, r3, #1 @ no → i++; strb r3, [fp, #-24] @ Update mem[fp – 24] = i b loop1 @ loop back around next: @end of loop

COPY STRING DATA

C	while (in[j]) out[i++] = in[j++];
ASM	

Exercise: Write this part of the code and discuss possible solutions with your peers :)

Tips:

- Load j into a register
- Use **ldrb** and **strb** to copy byte data
- Use **cmp** to compare
- Use **add** to increment i and j
- Use **b** to loop again

FINAL PART AND EPILOGUE

C	out[i] = 0; return out;
ASM	ret_out: **add** r0, r0, r3 @r0 = out (address) + i (bytes) **mov** r3, #0 @ r3 = 0 **strb** r3, [r0] @ out[i] = 0 **ldr** r0, [fp, #-16] @ return out pointer **ldmea** fp, {fp, sp, pc} @ do call return

Put a '\0' at the end of the concatenated string (the final part of the function body) and the function returns the length of this concatenated string in r0. The last instruction, ldmea, restores the stack frame and returns to the calling function. Notice that again we are assuming the 'store before call' method, we have not bothered to replace registers used.