Addressing, Subroutines and Stacks

Steven R. Bagley

Endianness

- Computer memory is addressed in bytes
- But often have to deal with things that are bigger than a byte
- How do the bits of these bigger entities get laid out into the bytes of memory?
- Two approaches: Big Endian, and Little Endian

Endianness

Split into 8bit chunks

MSB = most significant byte

LSB = Least Significant byte

Endianness

0x12345678

Split into 8bit chunks

MSB = most significant byte

LSB = Least Significant byte

Endianness

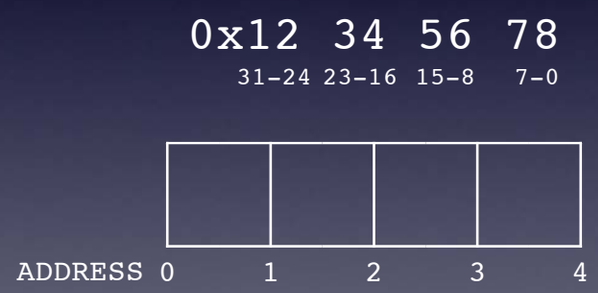
0x12 34 56 78

Split into 8bit chunks

MSB = most significant byte

LSB = Least Significant byte

Endianness

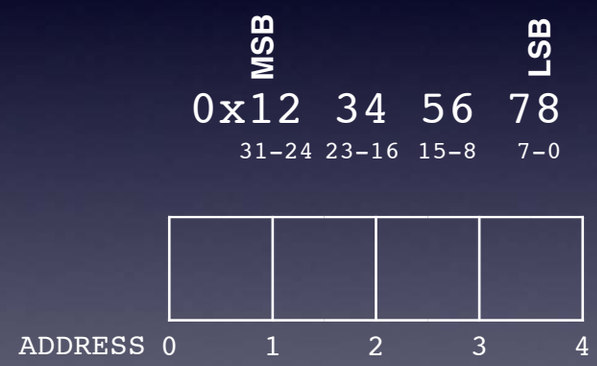


Split into 8bit chunks

MSB = most significant byte

LSB = Least Significant byte

Endianness

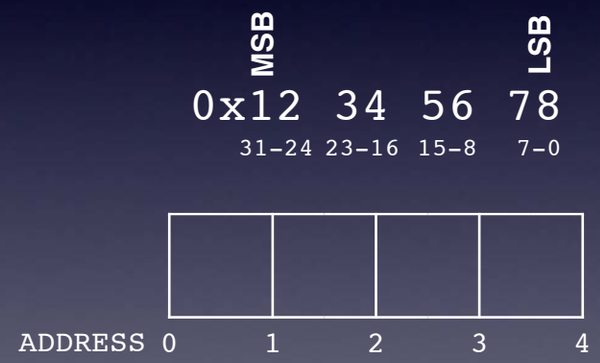


Split into 8bit chunks

MSB = most significant byte

LSB = Least Significant byte

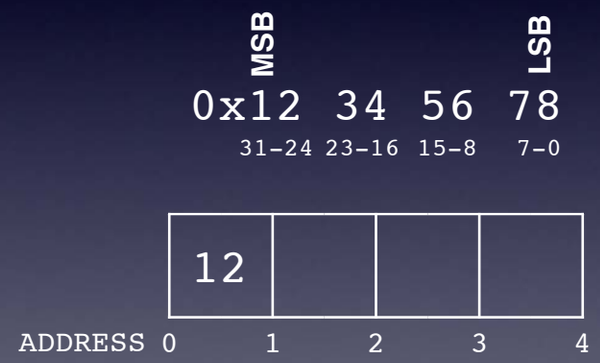
Big Endian



MSB = most significant byte

LSB = Least Significant byte

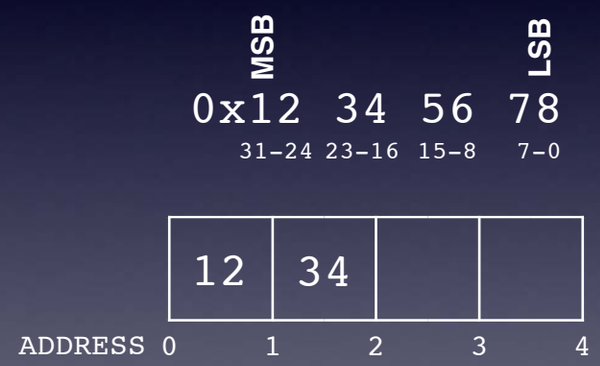
Big Endian



MSB = most significant byte

LSB = Least Significant byte

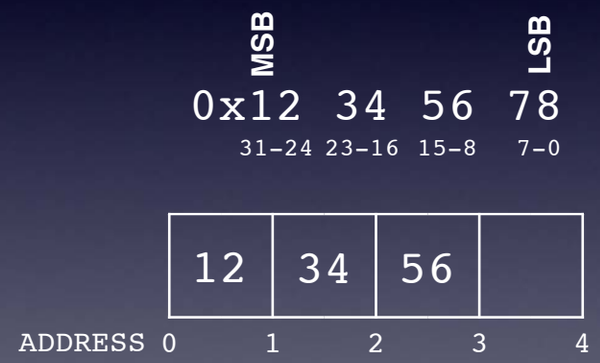
Big Endian



MSB = most significant byte

LSB = Least Significant byte

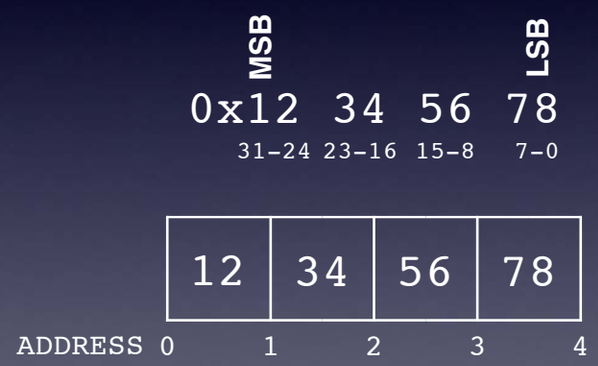
Big Endian



MSB = most significant byte

LSB = Least Significant byte

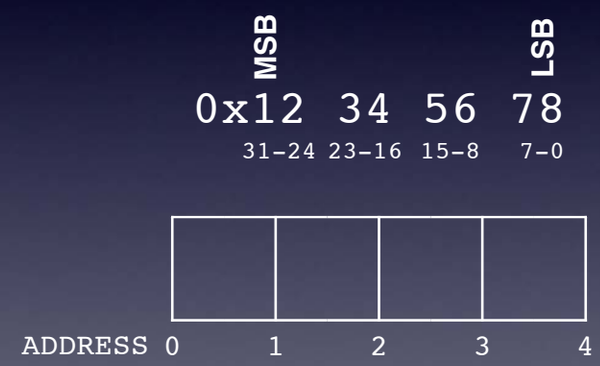
Big Endian



MSB = most significant byte

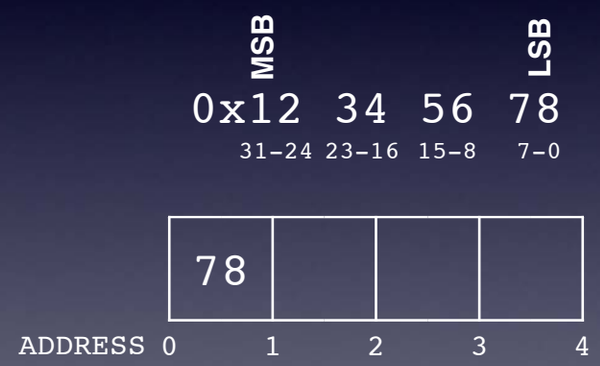
LSB = Least Significant byte

Little Endian



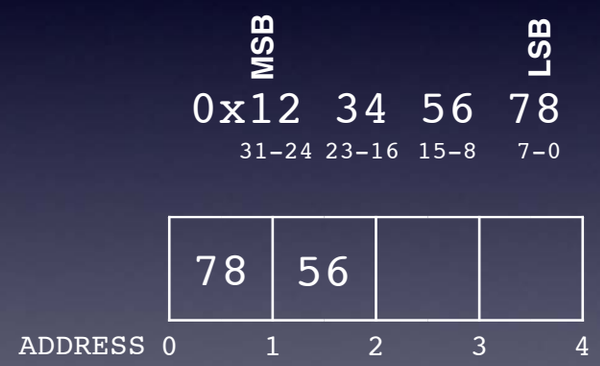
Yes, it’s all backwards, but it can make the chip design easier…

Little Endian



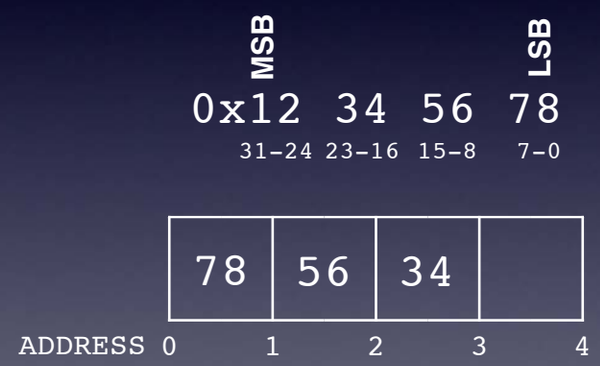
Yes, it’s all backwards, but it can make the chip design easier…

Little Endian



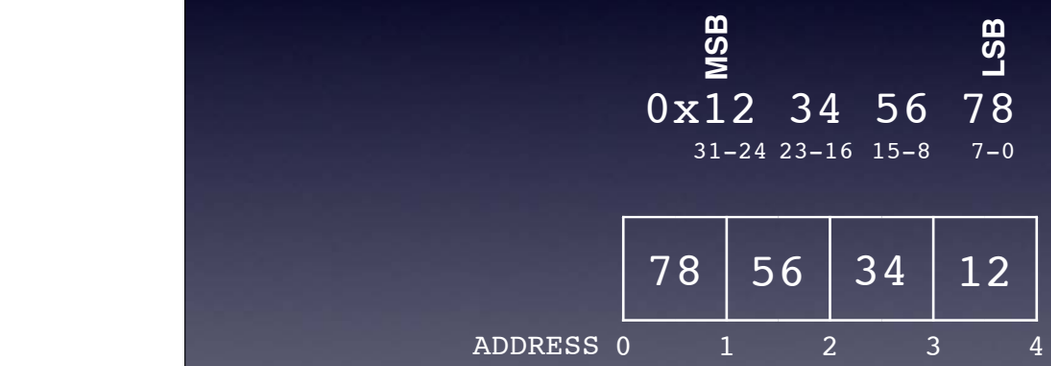
Yes, it’s all backwards, but it can make the chip design easier…

Little Endian



Yes, it’s all backwards, but it can make the chip design easier…

Little Endian



Yes, it’s all backwards, but it can make the chip design easier…

Writeback

- ARM's `LDR` and `STR` instructions also allow you to write the calculated address back into the base register
- This is called *writeback*
- Means that we can auto update the address register for free

Writeback

- Allows you to implement the equivalent of the C `*p++` and `*++p`
- Already seen similar to **`LDRB R2, [R1]`**
- Means “Load the low-order byte, at the memory location at address held in **`R1`** into in **`R2`**”
- We then did an **`ADD R1, R1, #1`** to update the address

Writeback

- But we can also do `LDRB R2, [R1], #1` and use write back to get the same effect
- This will copy the contents of the place in memory denoted by `[R1]` into `R2`
- But afterwards update the pointer in `R1` to point to the next byte (i.e. add one to the value in `R1`)
- This is done for free!

No extra cycles spent on doing this

Go and show how we can alter strlen.s to use write back and compare the different addresses.

Writeback

- Note the value added on is in bytes
- It is *not* converted to the size of the type as C does
- Need to do this manually (i.e. `[R1],#4`)
- Can also use a register here (with optional shift)

Pre-index Writeback

- Can also do the same with a pre-index operation
- But the syntax is different
- Here we just put an `!` after the square brackets, e.g. `LDR R1, [R0, #4]!`
- This loads `R1` with the value at memory location `R0+4`
- And updates `R0` to contain `R0 + 4`

ARM Indirect Addressing Summary

- Two possibilities for each of:
  - Address used: Register or Register + offset
  - Final register value: Unchanged or +offset

`Rn` is base register	`Address = Rn`	`Address = Rn + offset`
Rn unchaged	`[Rn]` **indirect** e.g. `[R1]`	`[Rn, offset]` **pre-indexed with offset**, e.g.
Rn = Rn + offset	`[Rn], offset` **post-indexed** e.g. `[R1] , #1`	`[Rn, offset]!` **pre-indexed with write back** e.g. `[R1, #12]!`

ARM Indirect Addressing Summary

- Offset can be a `number`, `-number`, Register or -Register
- Register offsets can also be shifted…

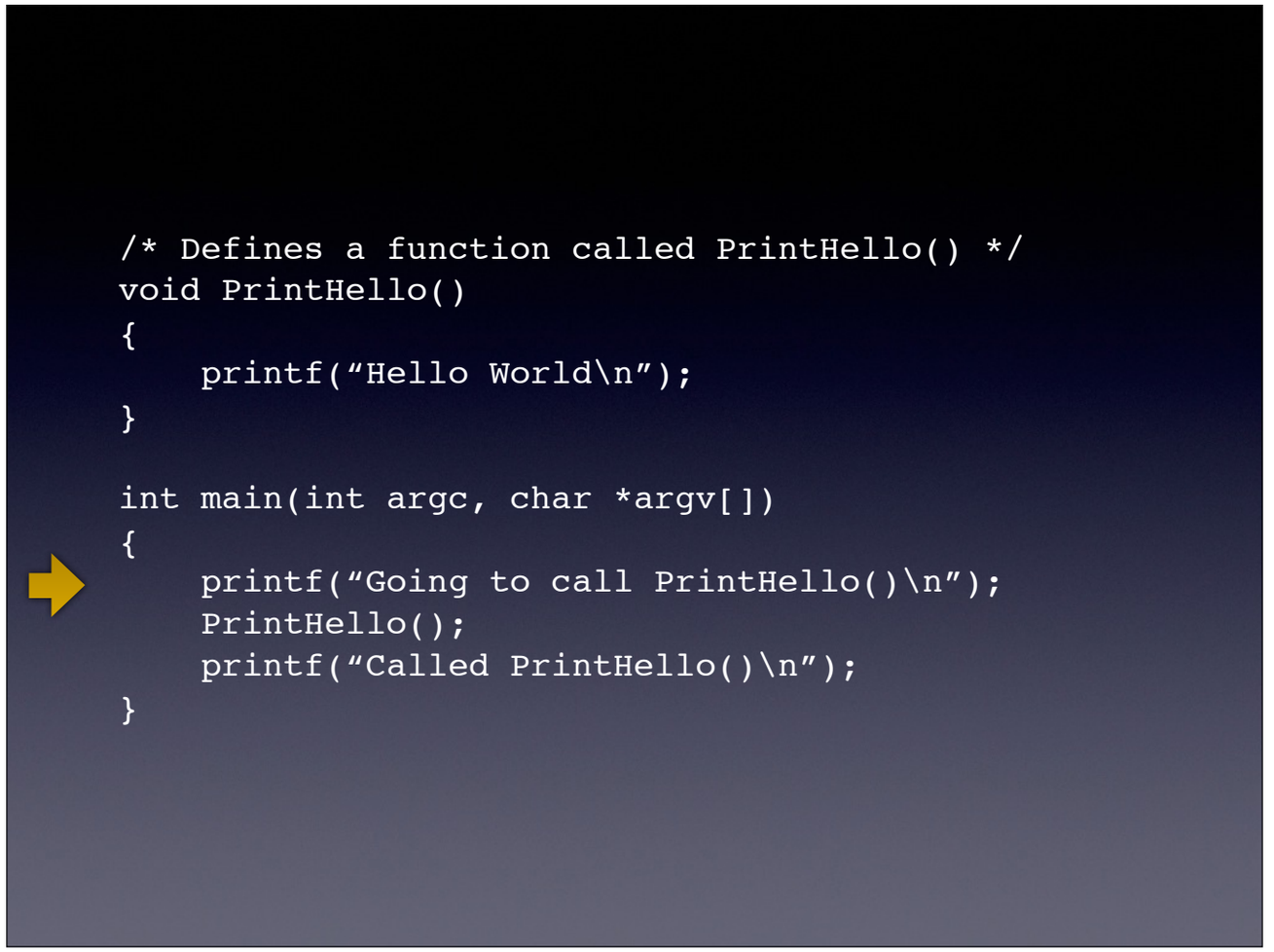
Subroutines and Stacks

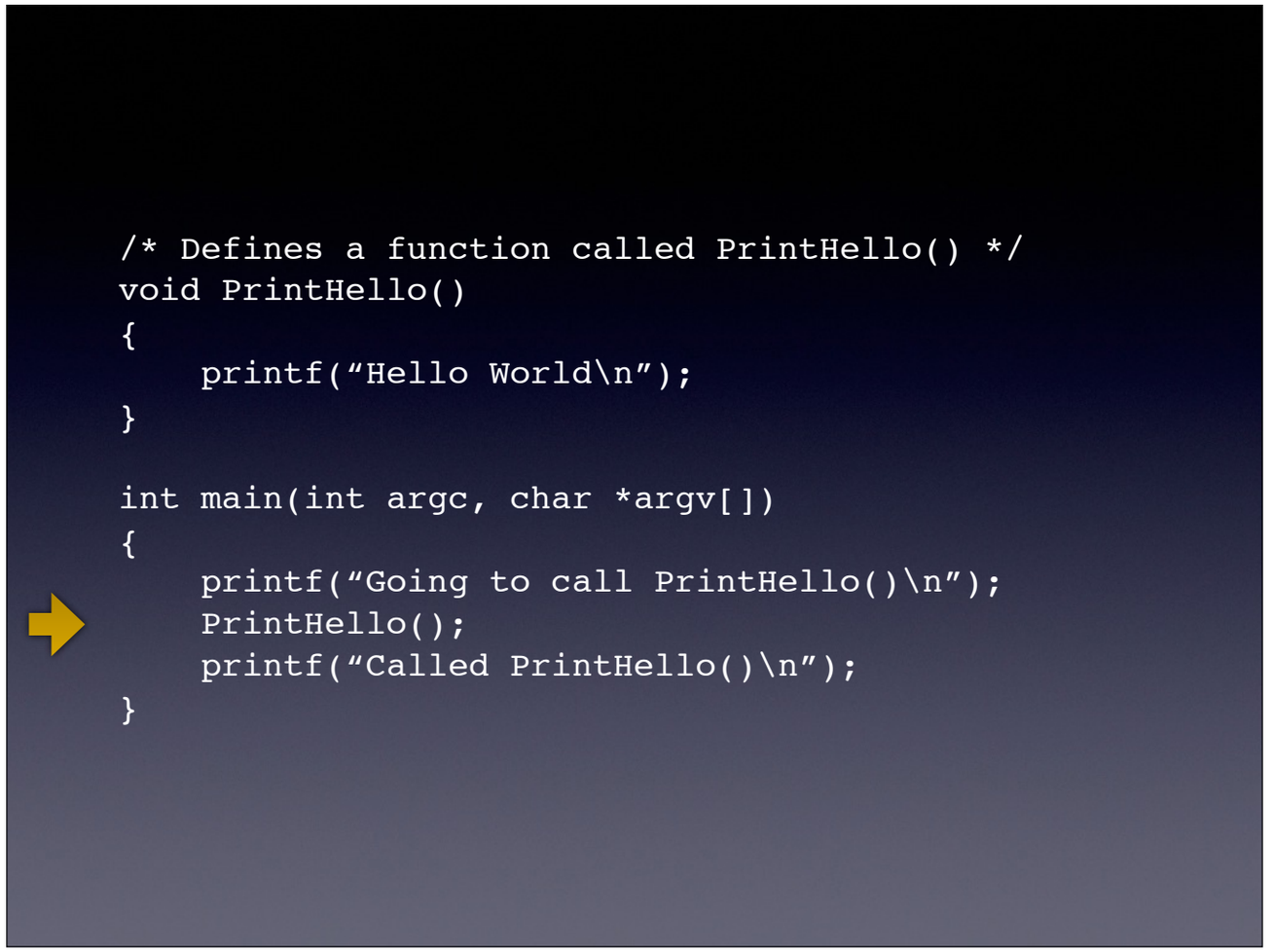
What is a Function?

- In C, we've written functions
  - Small blocks of code that can be called and reused
  - Also called *Sub-Routines*, *Procedures*, or *Methods*
- Technically, a function should return a value
- Some purists would argue that a C function returning **`void`** is actually a procedure

Why Procedures?

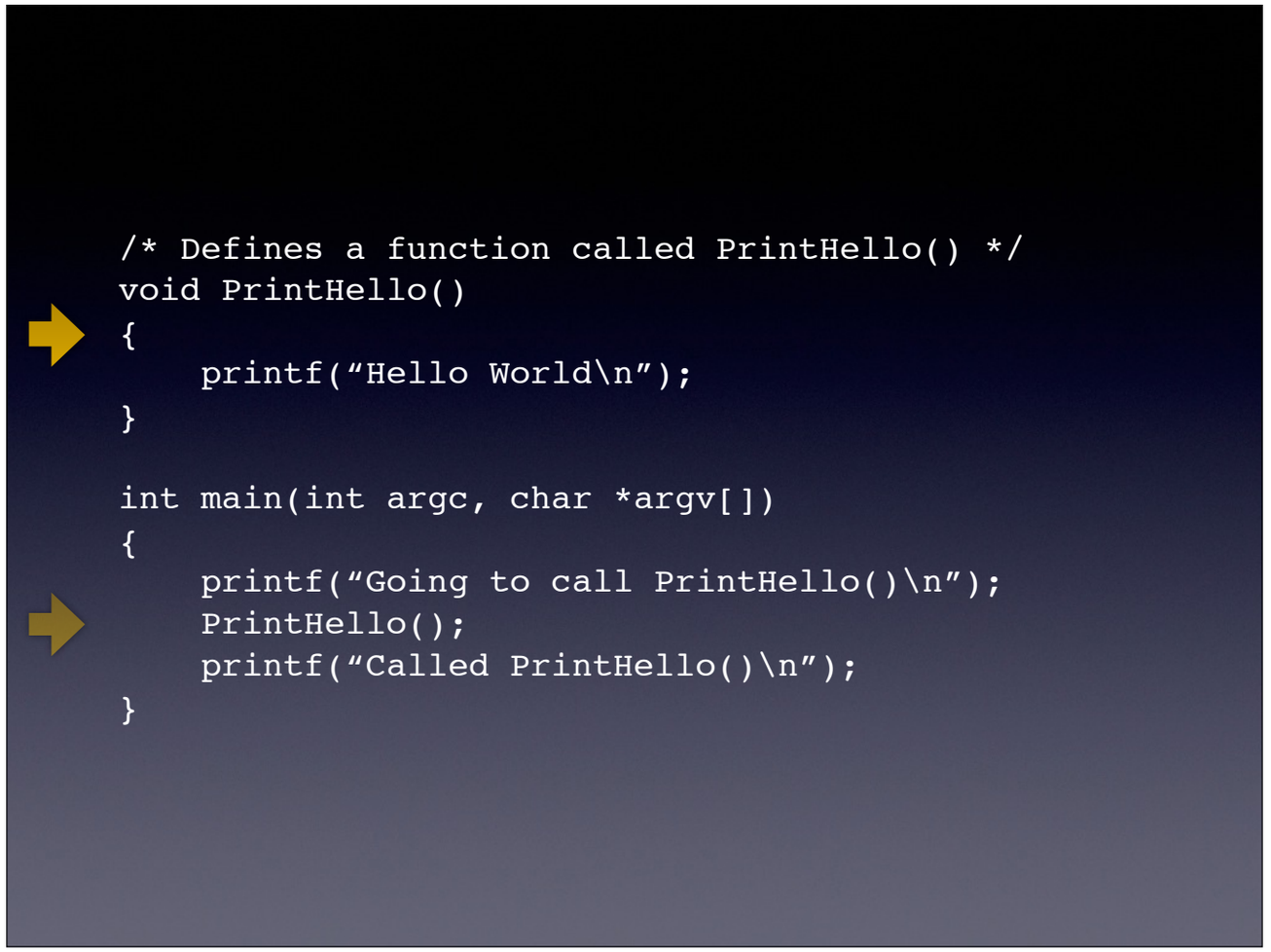
- Procedures reduce duplication of code and enable re-use
- Decompose programs into smaller, manageable parts
- Procedures call other procedures, possibly even themselves
- What happens when we call a procedure?

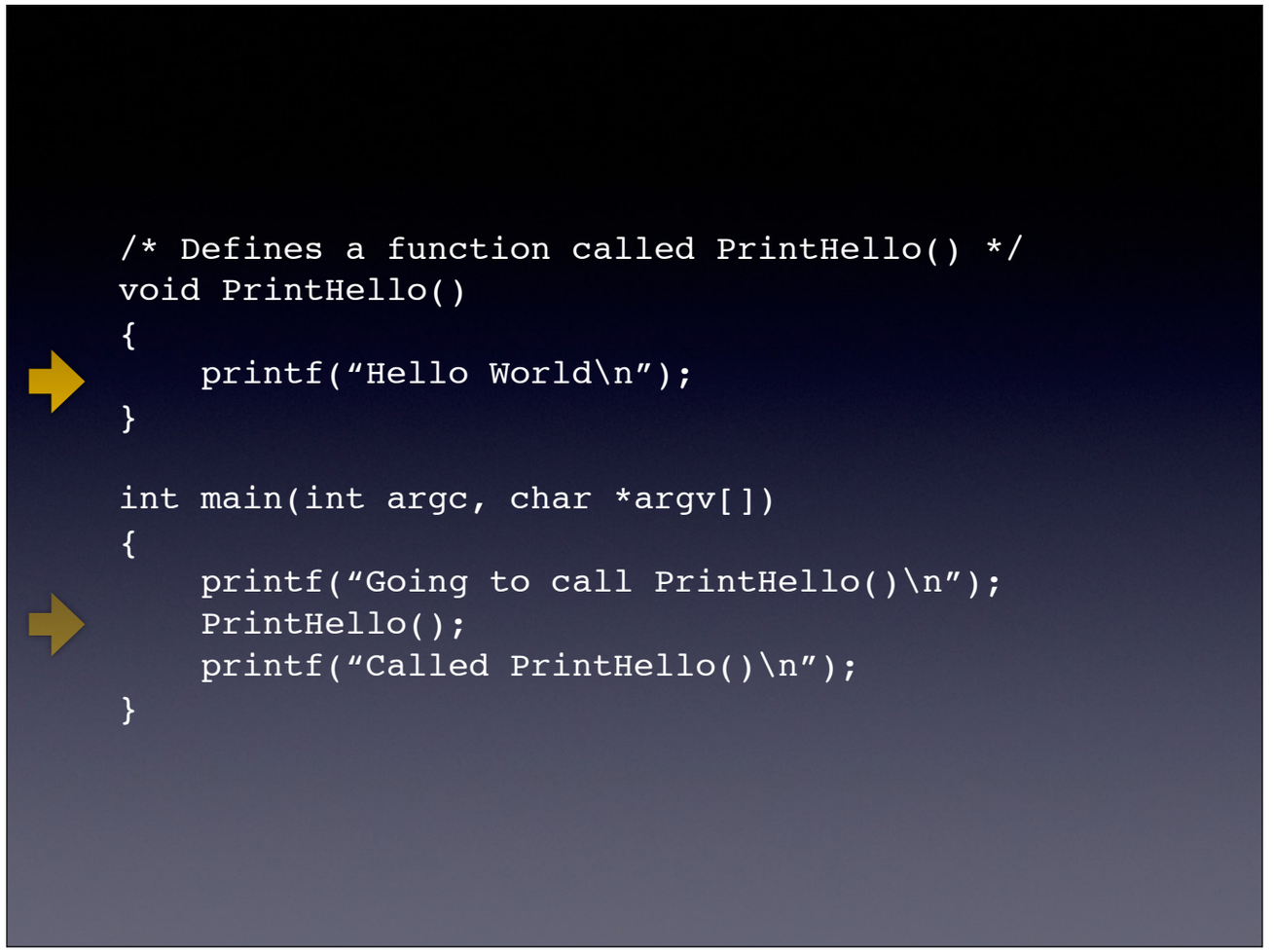

/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}

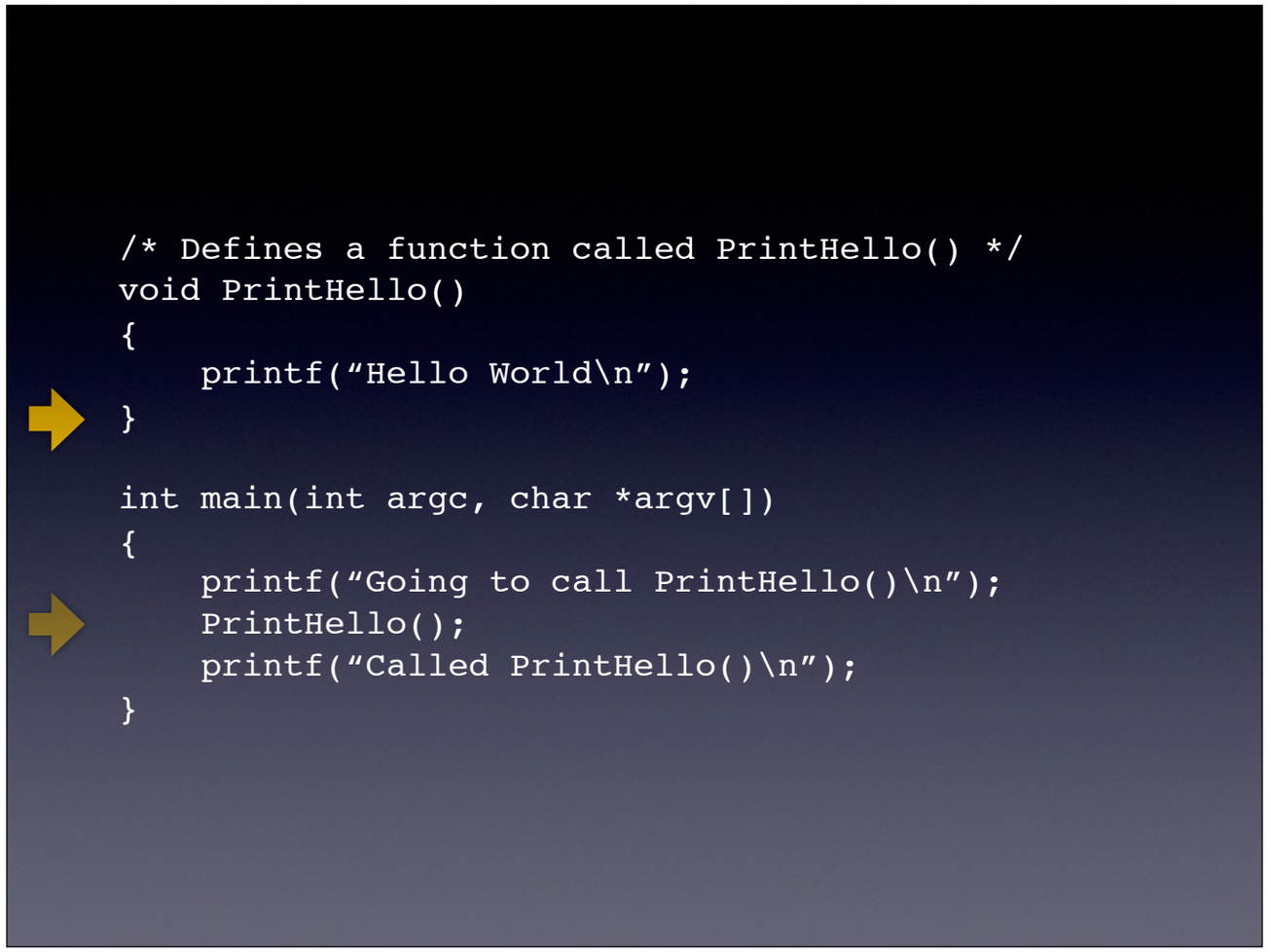


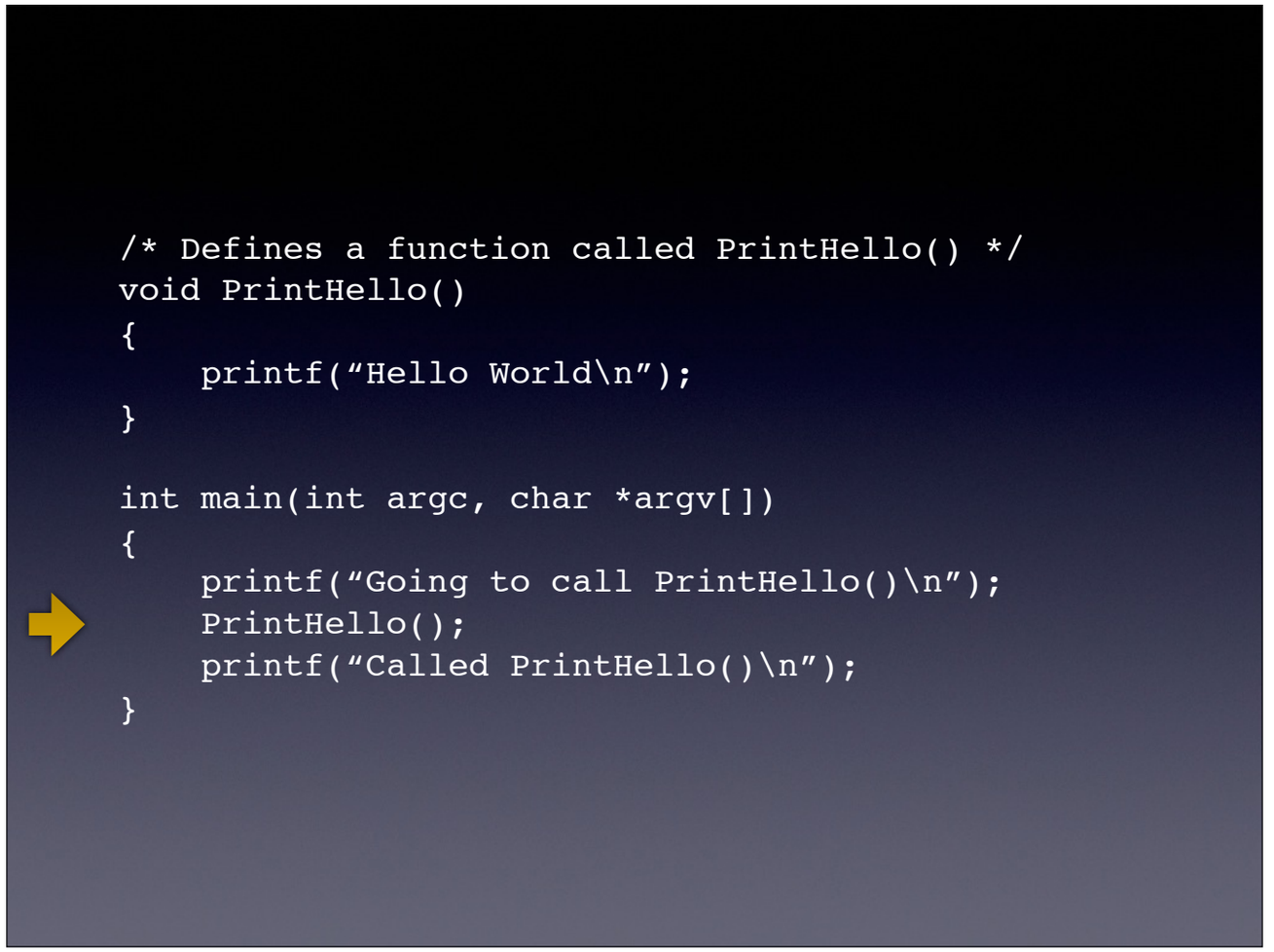
```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```


/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}

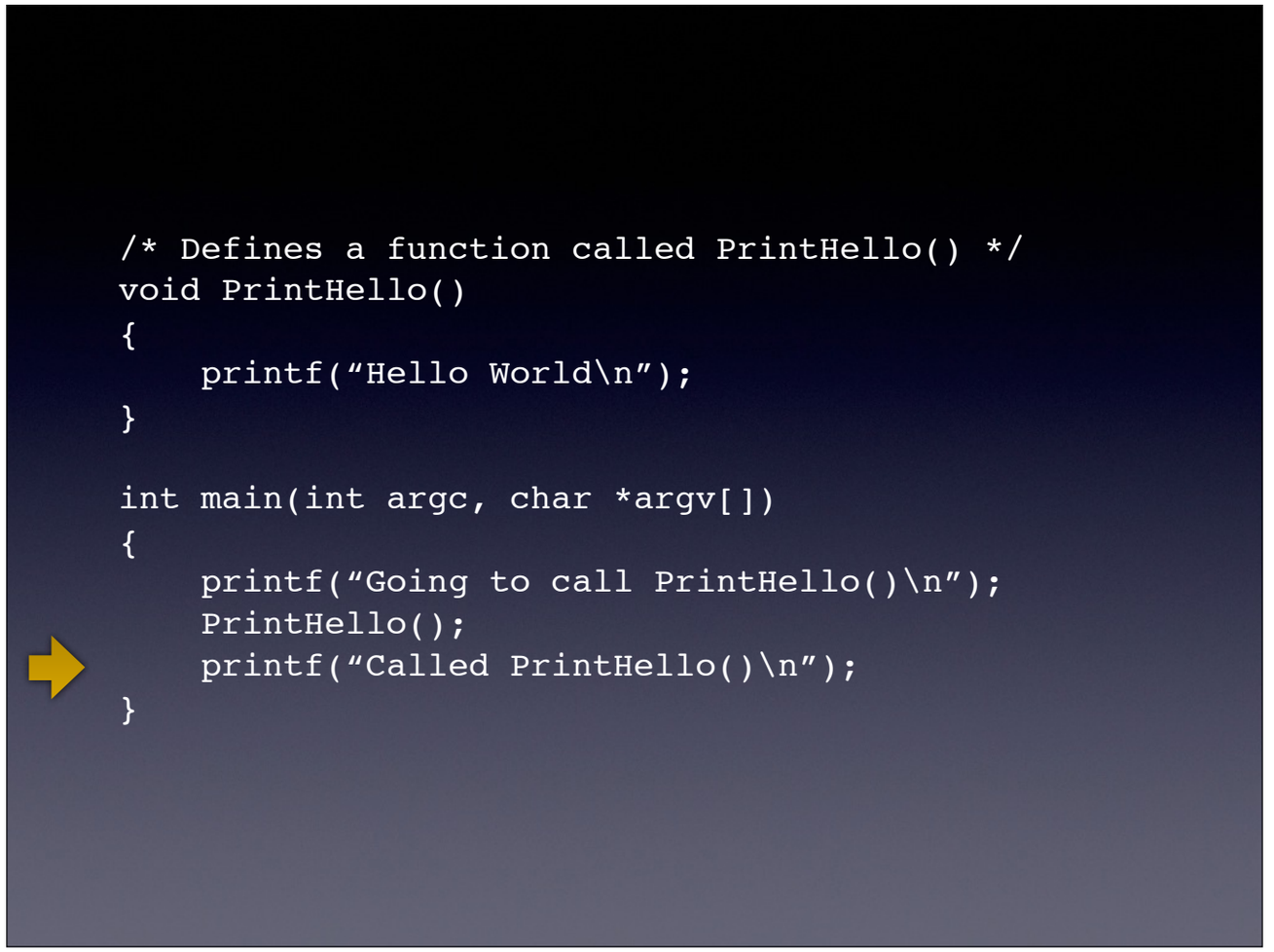

/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}


/* Defines a function called PrintHello() */
void PrintHello()
{
printf(“Hello World\n”);
}
int main(int argc, char *argv[])
{
printf(“Going to call PrintHello()\n”);
PrintHello();
printf(“Called PrintHello()\n”);
}



```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```



```
/* Defines a function called PrintHello() */
void PrintHello()
{
    printf("Hello World\n");
}

int main(int argc, char *argv[])
{
    printf("Going to call PrintHello()\n");
    PrintHello();
    printf("Called PrintHello()\n");
}
```

Calling a Procedure

- When we call a procedure
  - The *caller* is suspended
  - Control is passed to the *callee*
  - Callee performs the requested task
  - Callee returns control to the caller
- How do we implement this in assembler?

Implementing Subroutines

- Have already seen how we can jump to a block of code
- Use the branch (`B`) instruction…
- Can branch to the start of the callee…
- But how do we get back into the caller?
- How about another branch?

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        ; ??? What goes here to go back to caller?

main    ADR R0, string
        B strlen
        SWI 4
        SWI 2
```

Branch to the instruction after the call

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        B next

main    ADR R0, string
        B strlen
next    SWI 4
        SWI 2
```

How do we get back?

Implementing Subroutines

- This works…
- Caller (**`main`**) calls the subroutine (**`strlen`**)
- Callee (**`strlen`**) can then calculate the length of the string
- Before branching back into the caller (**`main`**)
- But what if we want to call **`strlen`** twice?

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        B next

main    ADR R0, string
        B strlen
next    SWI 4
        ADR R0, secondString
        B strlen
        SWI 2
```

Ooops, now branches back to next again — which is the wrong place!

Implementing Subroutines

- We've hard-coded the 'return' address which means we can only call it from one place
- The callee has no way of knowing how where it should jump back to
- Only the caller knows this
- Need a mechanism for the caller to let the callee know where to branch back to…

Self-modifying code

- Recall that a program is just a series of opcodes in memory
- This memory can be modified like any other
- Program as it is running replaces the bytes making up the branch instruction with an instruction that branches to the right place

Calculate the correct offset and formulate it as an instruction…

Then store it in memory at the right point

Self-modifying Code

- This is messy!
- But it works
- Was used on some of the first computers to implement sub-routines

EDSAC and the Wheeler jump

Program Counter

- There's another way we can change which instruction the CPU executes next
- Remember that the Program Counter is just another register (`R15` on the ARM)
- If we move a value into `R15` then the CPU will execute the instruction at that new address next

Program Counter

- The caller knows which is the next instruction that should execute
- So it could pass the *address* of that function to the callee
- When the callee has finished it can just move that address into the `PC` to return to the caller

Link Register

- On ARM, this is done by passing the address in register `R14`
- Which is called the *Link Register*
- Callee can then return to the caller with a simple `MOV PC, R14`

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        MOV PC, R14

main    ADR R0, string
        ADR R14, next
        B strlen
next    SWI 4
        ADR R0, secondString
        ADR R14, nextnext
        B strlen
nextnextSWI 2
```

Now each time strlen is called, R14 has a different address

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        MOV PC, R14

main    ADR R0, string
        MOV R14, PC
        B strlen
        SWI 4
        ADR R0, secondString
        MOV R14, PC
        B strlen
        SWI 2
```

If we remember the Fetch-decode-execute cycle then we can remember that the PC will already be pointing 2 instructions ahead so we could just replace it with a MOV R14,PC rather than having to get the address of a label…

```
        B main

        …

strlen  MOV R1, R0
loop    LDRB R2, [r0], #1
        CMP R2, #0
        BNE loop
        SUB R0, R0, R1
        MOV PC, R14

main    ADR R0, string
        BL strlen
        SWI 4
        ADR R0, secondString
        BL strlen
        SWI 2
```

Such a common thing to do that it is built into the CPU with the BL instruction

Branch-with-link

Branch-with-Link

- Branch with Link instruction (**`BL`**) automatically puts the address of the following instruction in **`R14`**
- Always **`R14`**, we can't chose the register
- Can also be a conditional branch (e.g. **`BLEQ`**)
- Subroutine can then return to the next instruction with ease…

Branch-with-link if equal…

Basic procedure calls

- Seen that `BL` instruction uses `R14` as the link register to store return address
- In simple cases, at the end of a procedure we just need to do `MOV PC, R14`
- Some routines may be able to do their job solely with registers
- Need conventions for register usage to avoid overwriting and misunderstandings
- Thus we have the *APCS* (*A*RM *P*rocedure *C*all *S*tandard) to guide us…

APCS Register Use Convetion

Register	APCS name	APCS role
R0	a1	Argument 1 / integer result / scratch register
R1	a2	Argument 2 / integer result / scratch register
R2	a3	Argument 3 / scratch register
R3	a4	Argument 4 / scratch register
R4	v1	Register variable 1
R5	v2	Register variable 2
R6	v3	Register variable 3
R7	v4	Register variable 4
R8	v5	Register variable 5
R9	sb/v6	Static Base / Register variable 6
R10	sl/v7	Stack Limit / Register variable 7
R11	fp	Frame Pointer
R12	ip	Scratch register / specialist use by linker
R13	sp	Lower end of current stack frame
R14	lr	Link address / scratch register
R15	pc	Program Counter

Scratch registers do not need to be preserved through a function call, but all other registers should be. As far as the caller is concerned it should be as if the function call never happened

Note if more arguments are needed than registers they are placed on the stack before the procedure call. Each argument must take up a multiple of 4 bytes on the stack. For 8-byte wide values, two registers are used…

Caller Saved Registers

- **`R0`—`R3`** used to pass arguments into a function
- Inside the function they may be used by the function for any purpose.
- R0 often used to return the result
- Caller must expect **`R0`–`R3`** to be trashed (reused) by the procedure…

Caller Saved Registers

- If the Caller doesn't want the values in `R0-R3` trashing,then it must preserve them itself (e.g. by saving them in memory)
- A typical simple *leaf* function (such as `strlen`)
  - Would only use `R0–R3`
  - Be called with `BL`
  - Returns with `MOV PC, R14`

Leaf instruction is one that doesn't call any other function…

If it did call another function then R14 would get trashed…

Callee Saved Registers

- **`R4`**–**`R8`** are registers which the callee is expected to save if it uses them
- In other words, the values of **`R4`**–**`R8`** should be unchanged when control returns to the calling function
- So if the called function needs these register for extra workspace, it must save them
- And then restore them before returning to the caller

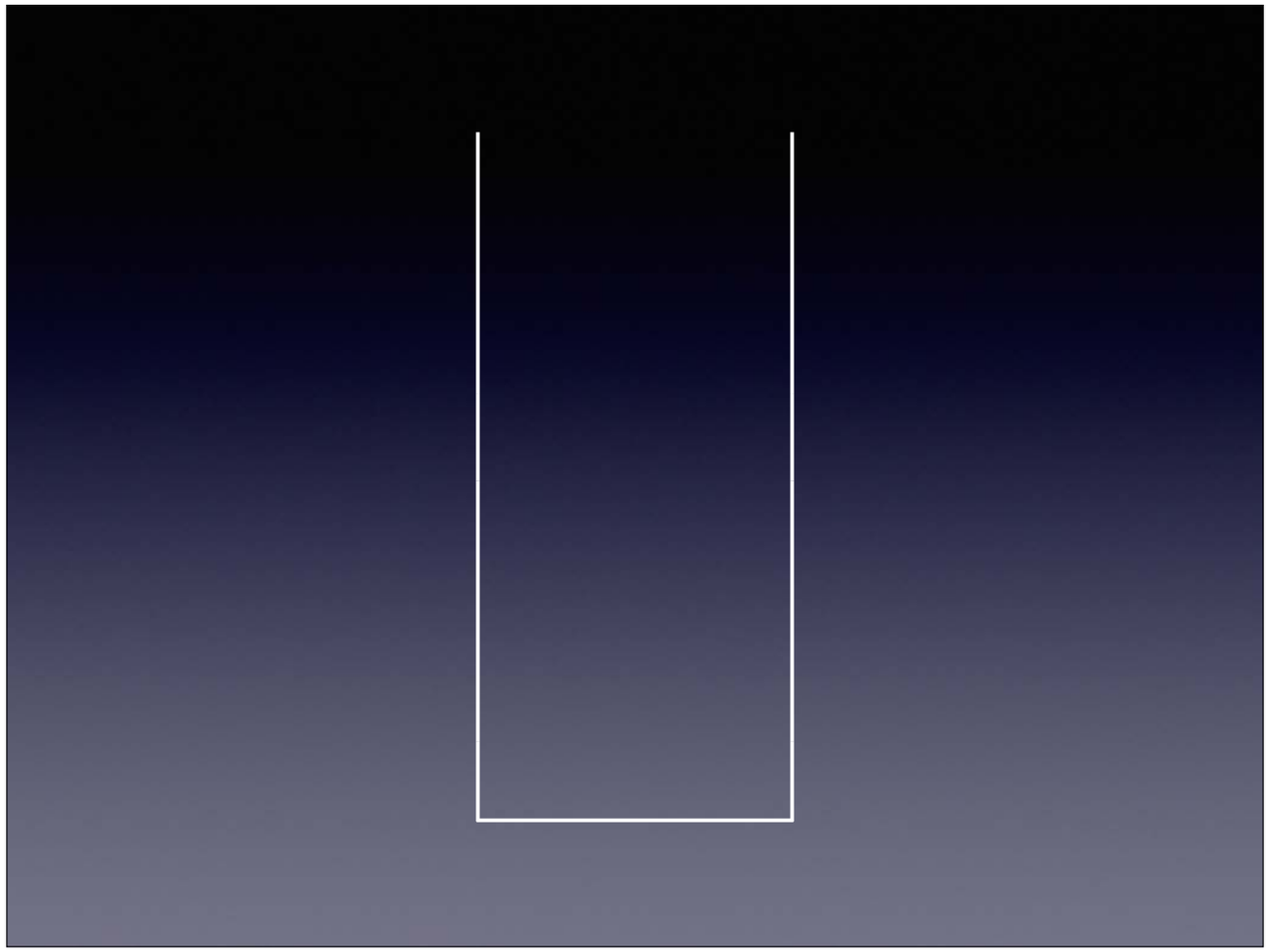
Saving Registers

- We have a limited number of registers
- But lots of memory
- Can use memory to save the values
- But we need a disciplined way to do this
- Most often this is done using a *stack*

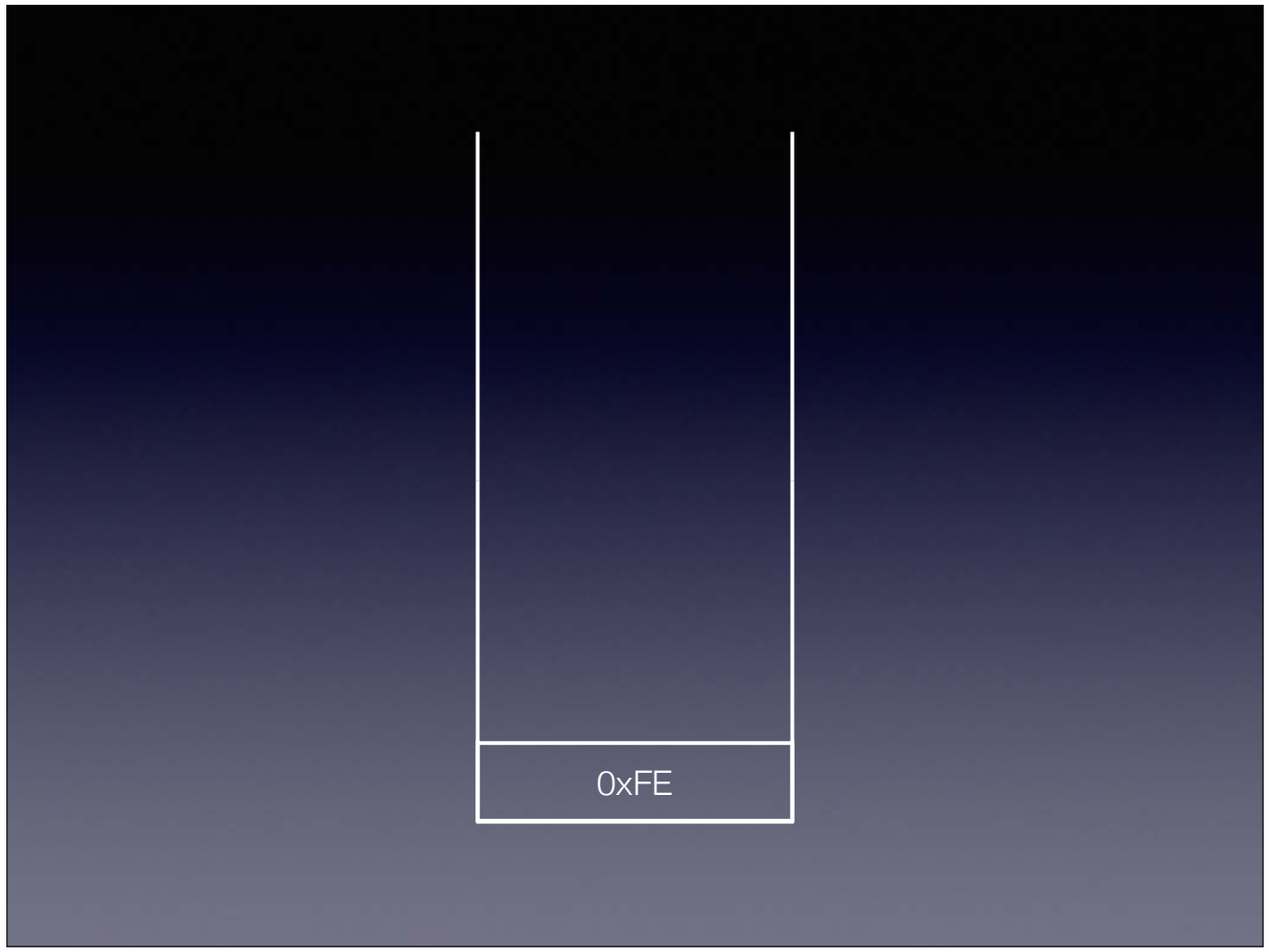
The Stack

- The Stack is a data structure
- Used to store values
- Provides *Last In, First Out* data storage (LIFO)
- Data comes out in reverse order to which it goes in
- Placing words on the stack is termed *pushing*
- Taking words off the stack is called *popping*

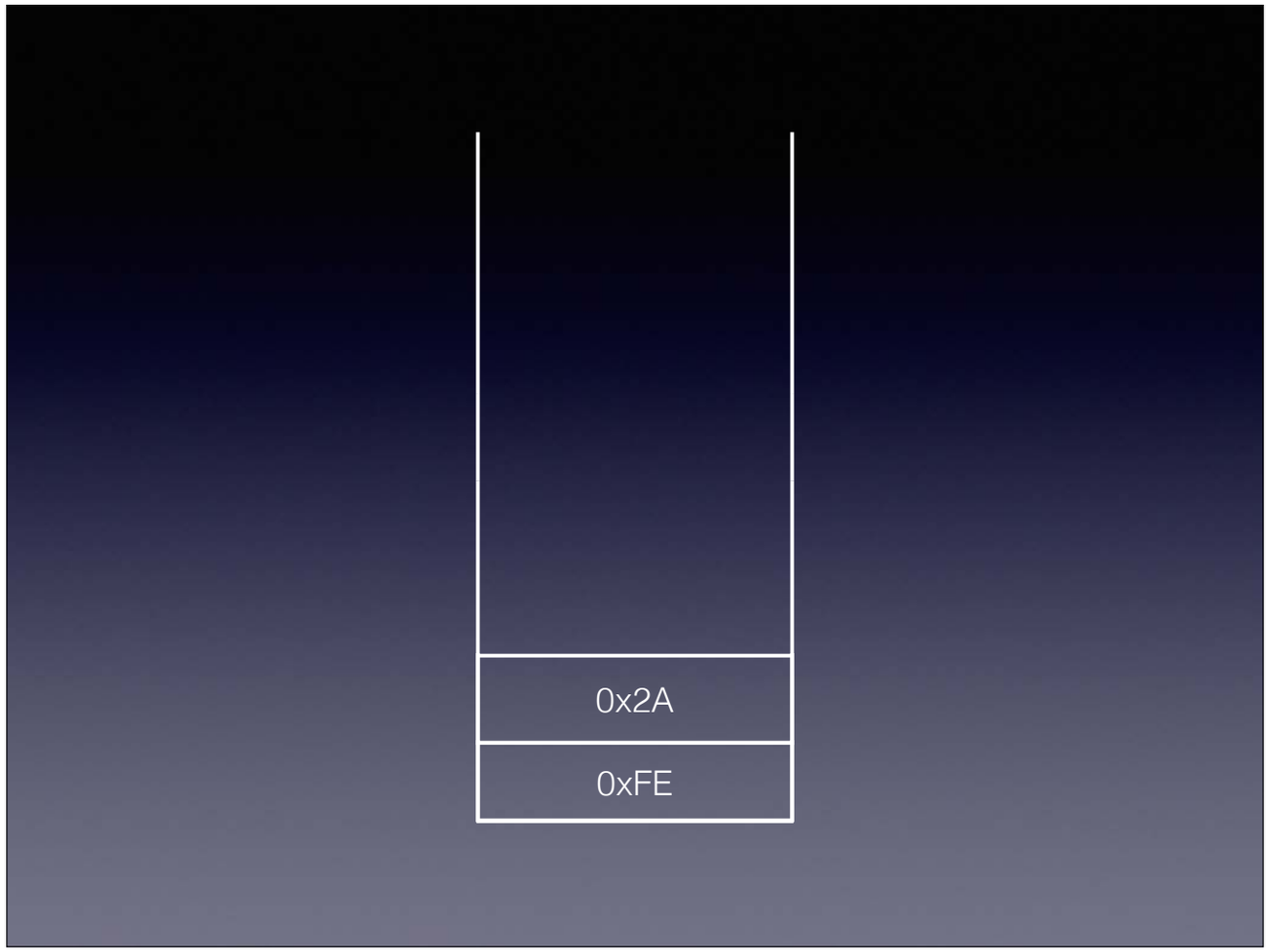
Like a stack of plates, we can only remove the top thing from the stack (although we can peek at the values below it)



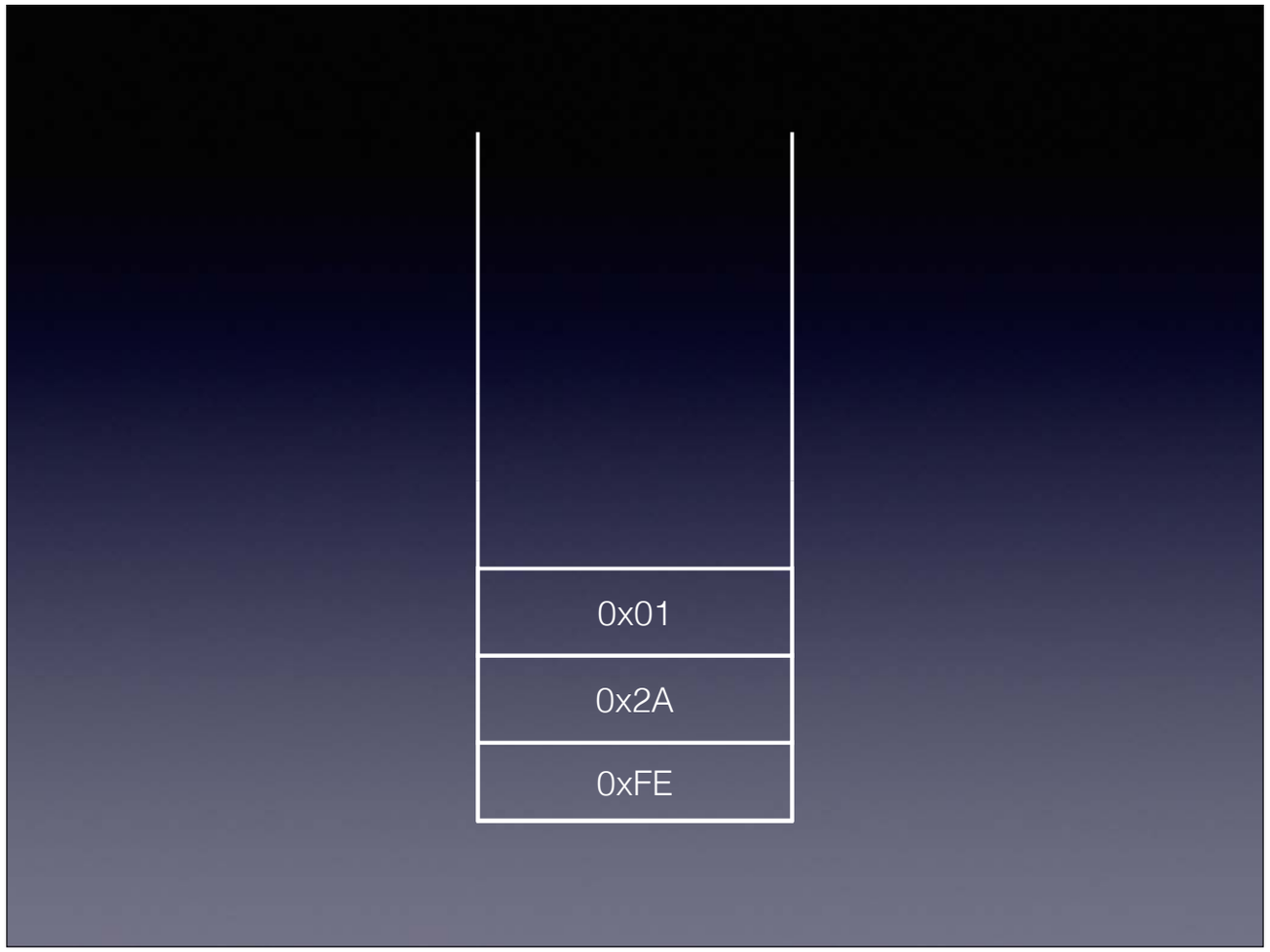
Pushing the values on the stack and then popping them back off in the reverse order



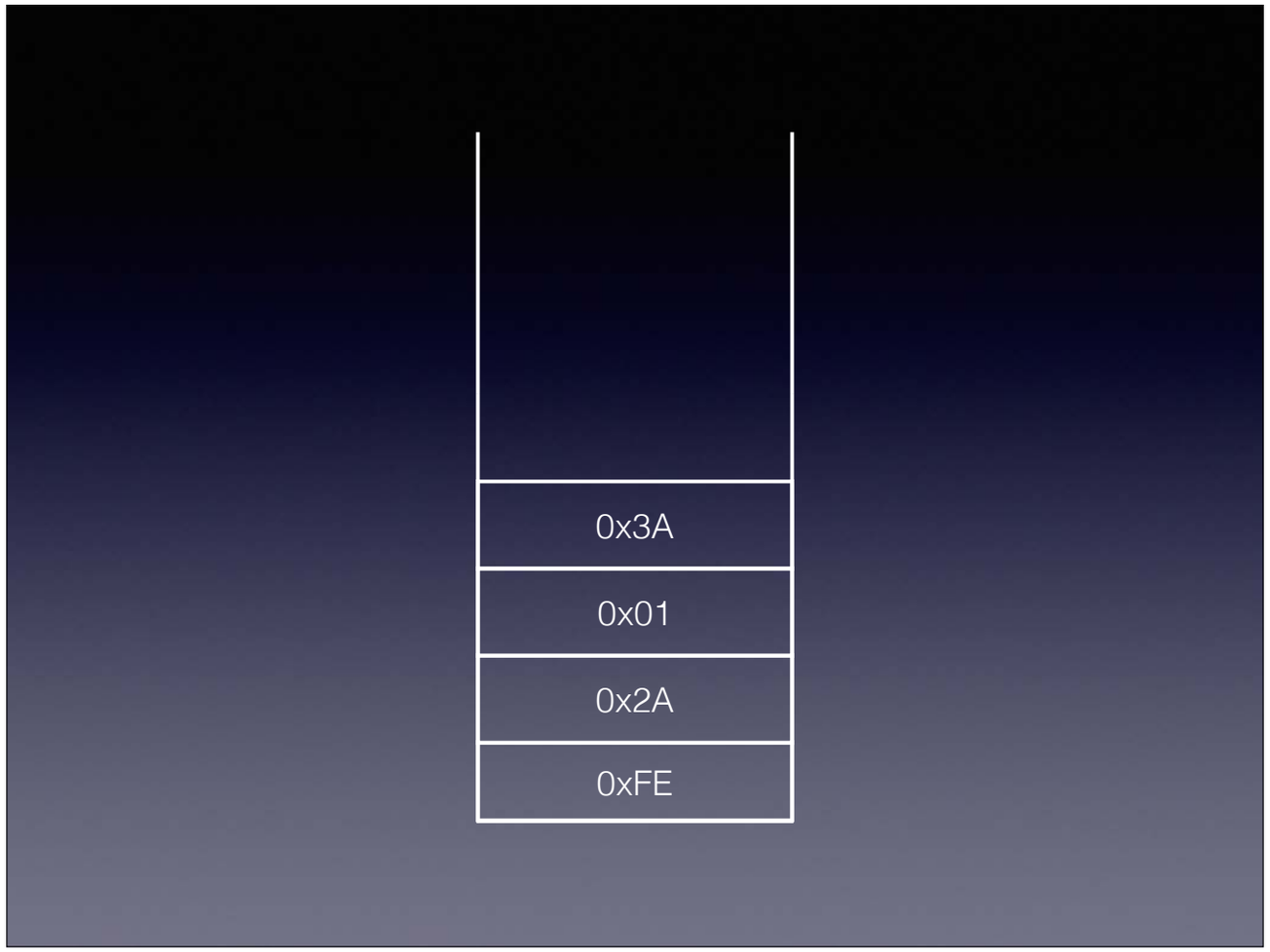
Pushing the values on the stack and then popping them back off in the reverse order



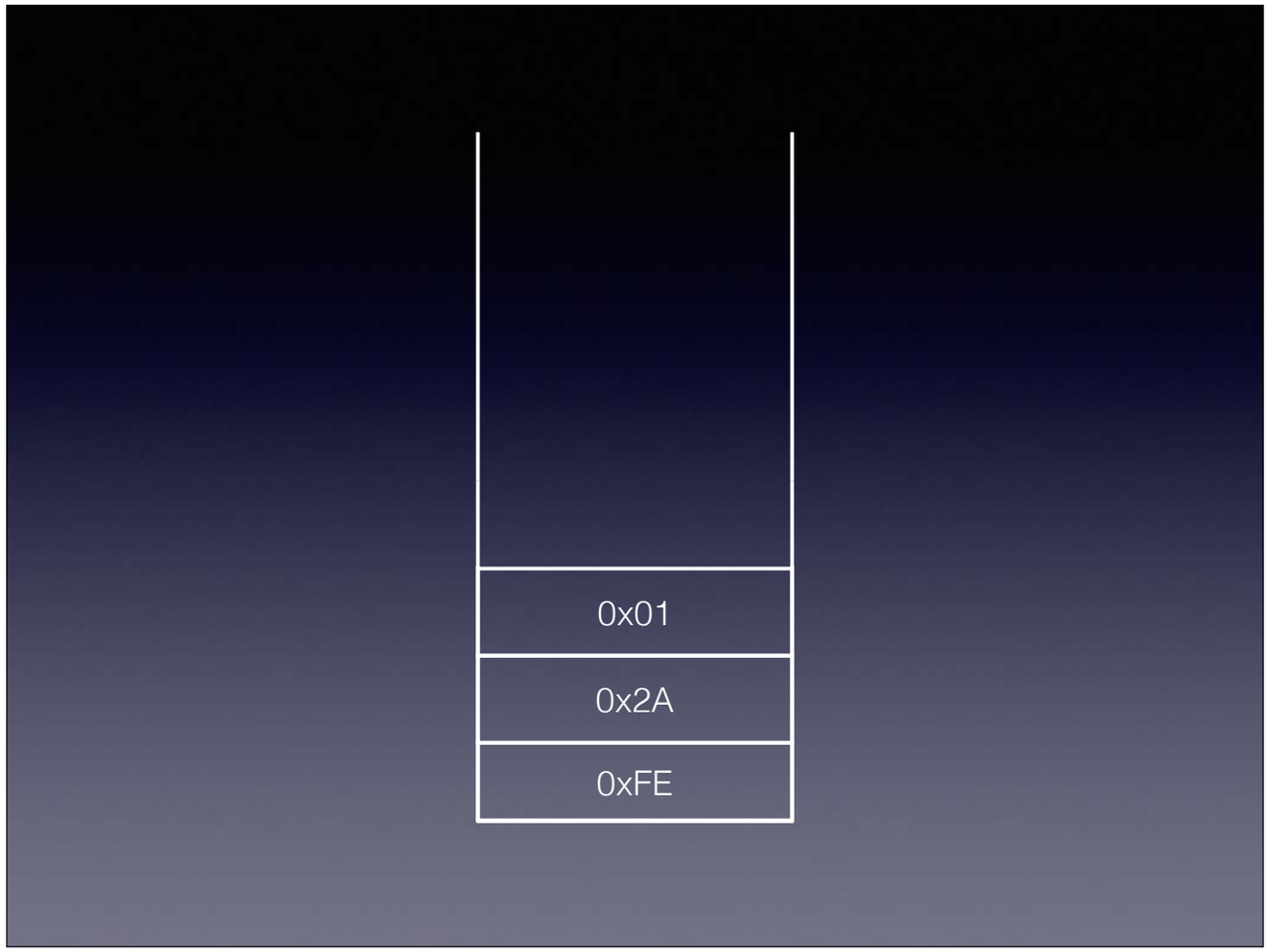
Pushing the values on the stack and then popping them back off in the reverse order



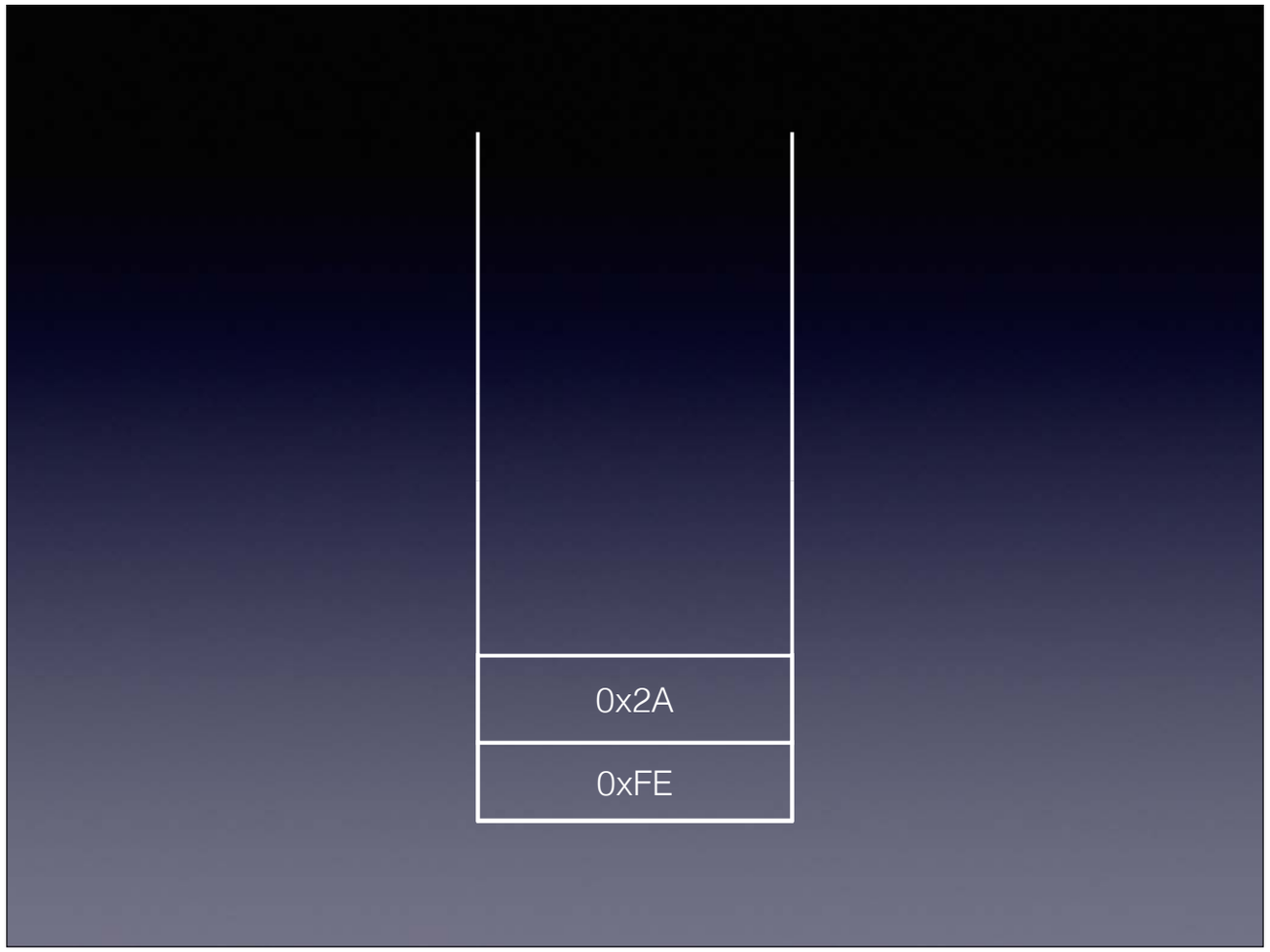
Pushing the values on the stack and then popping them back off in the reverse order



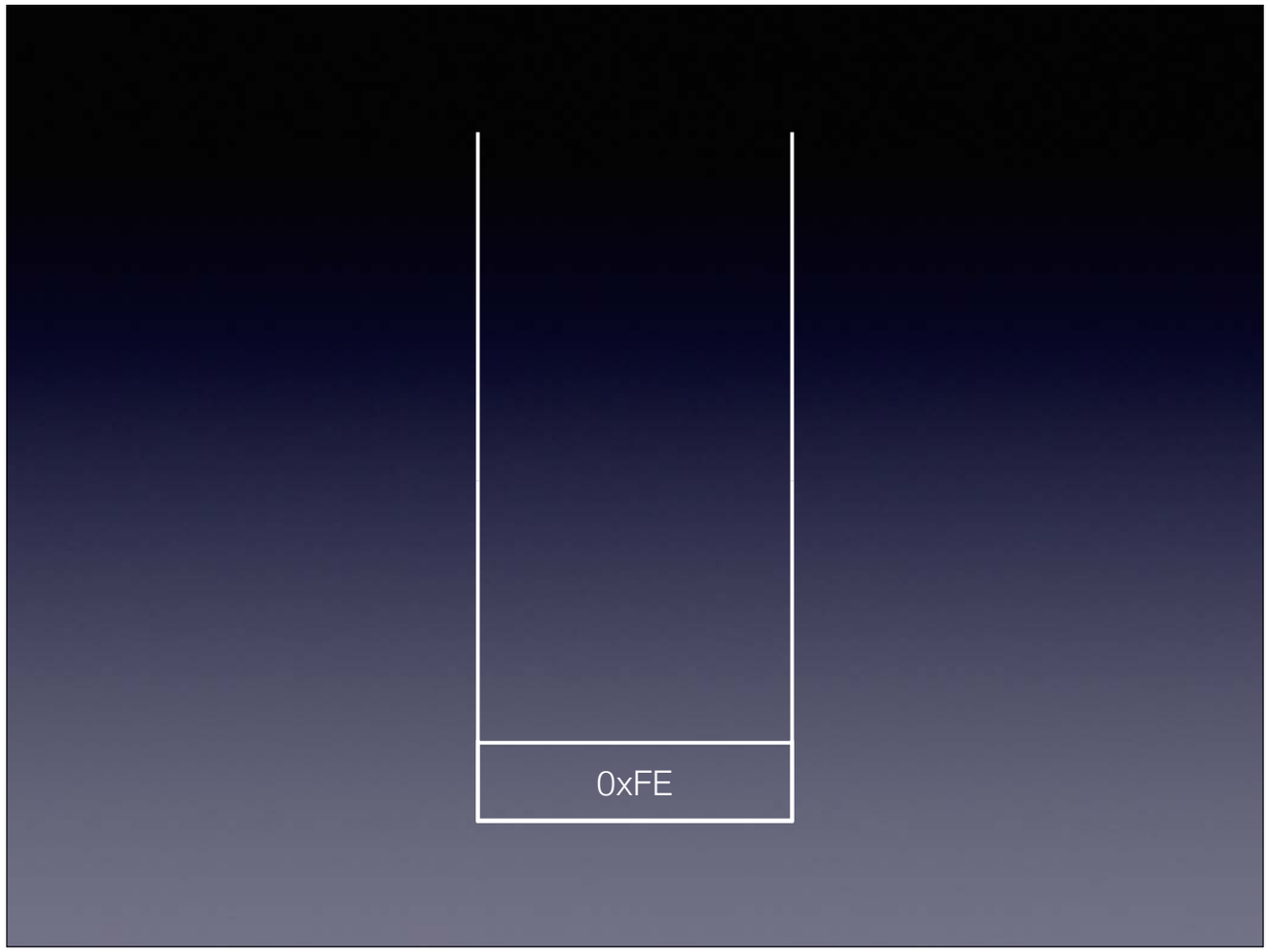
Pushing the values on the stack and then popping them back off in the reverse order



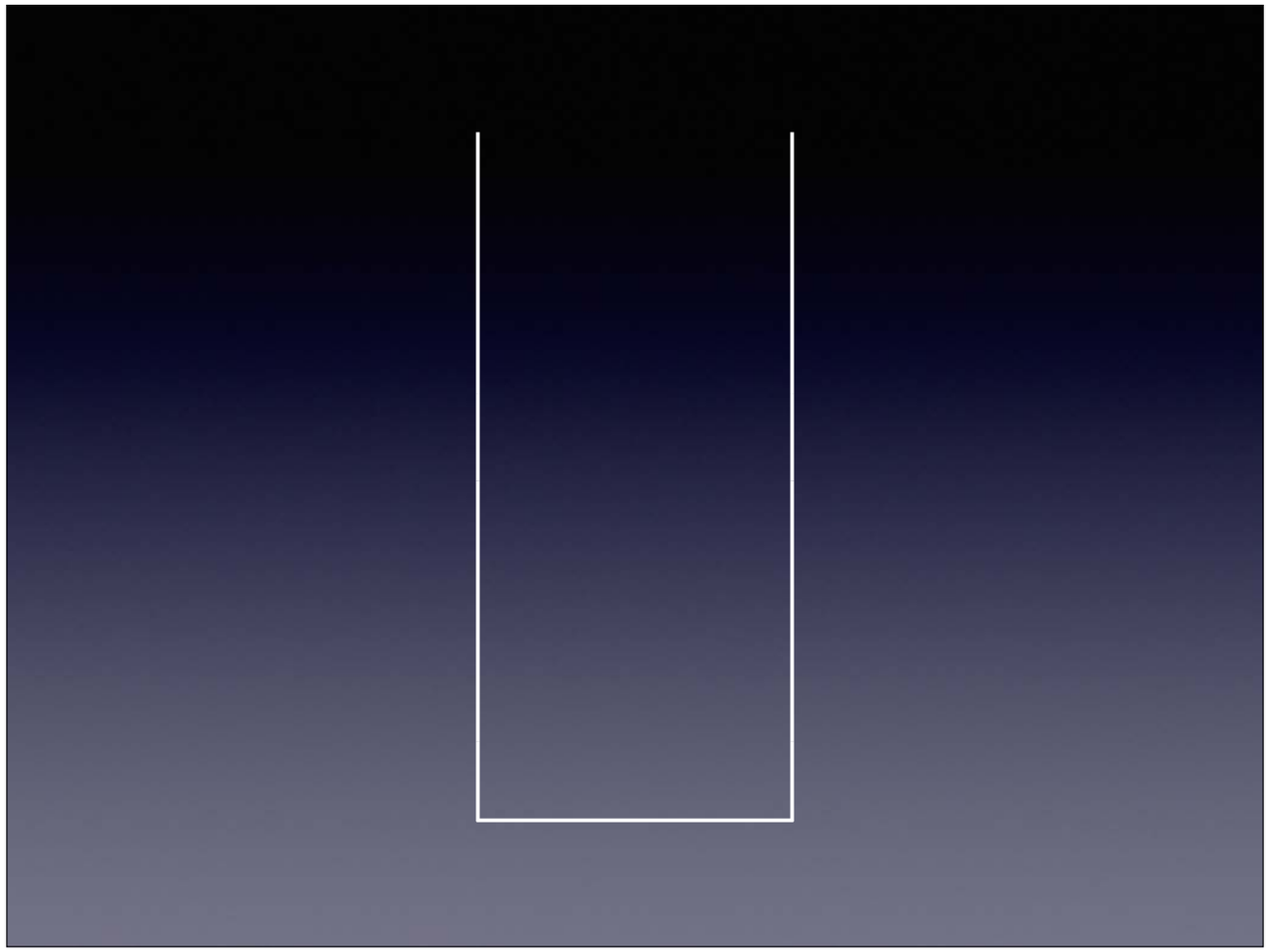
Pushing the values on the stack and then popping them back off in the reverse order



Pushing the values on the stack and then popping them back off in the reverse order



Pushing the values on the stack and then popping them back off in the reverse order



Pushing the values on the stack and then popping them back off in the reverse order

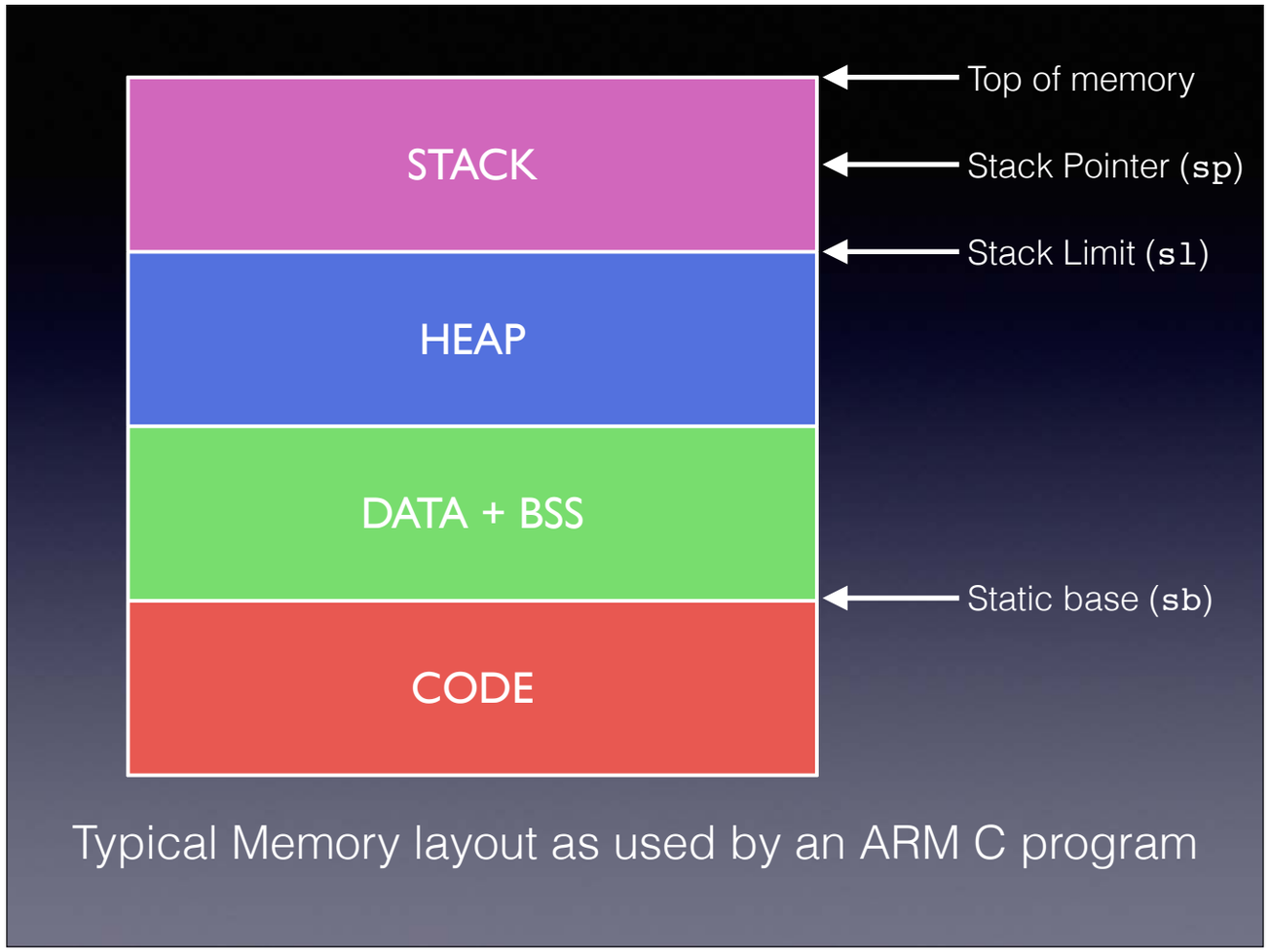
Stack Implementation

- How do we implement a stack in memory?
- Turns out that there are various approaches
- Need a stack pointer, register `R13` is used on ARM
- But what does it point to?

Stack Implementation

- Does the stack grow downwards (descending addresses) or upwards (ascending addresses) in memory?
- Does the stack pointer point to the topmost filled location (full stack)
- Or should it point to the next empty location just beyond the top of the stack (stack empty)?
- No right answer, but most systems including ARM use a “full-descending” stack

That means the stack pointer points to the top-most item on the stack (full) and the memory addresses grow downwards



Typical Memory layout as used by an ARM C program

Saw this in PRG
Stack pointer starts off at the top of memory and moves down and up as things are pushed and popped (respectively) between the top of memory and the stack limit

Using the APCS names for things here — some assemblers (like aasm) will let you use these names interchangeable with R13 (for sp) etc.

Stack Implementation

- Can use **`LDR`** and **`STR`** to push and pop registers from the stack
- Because we are using a full-descending stack, we need to use *pre-decrement* to push something on the stack

```
STR R0, [R13, #-4]!
```

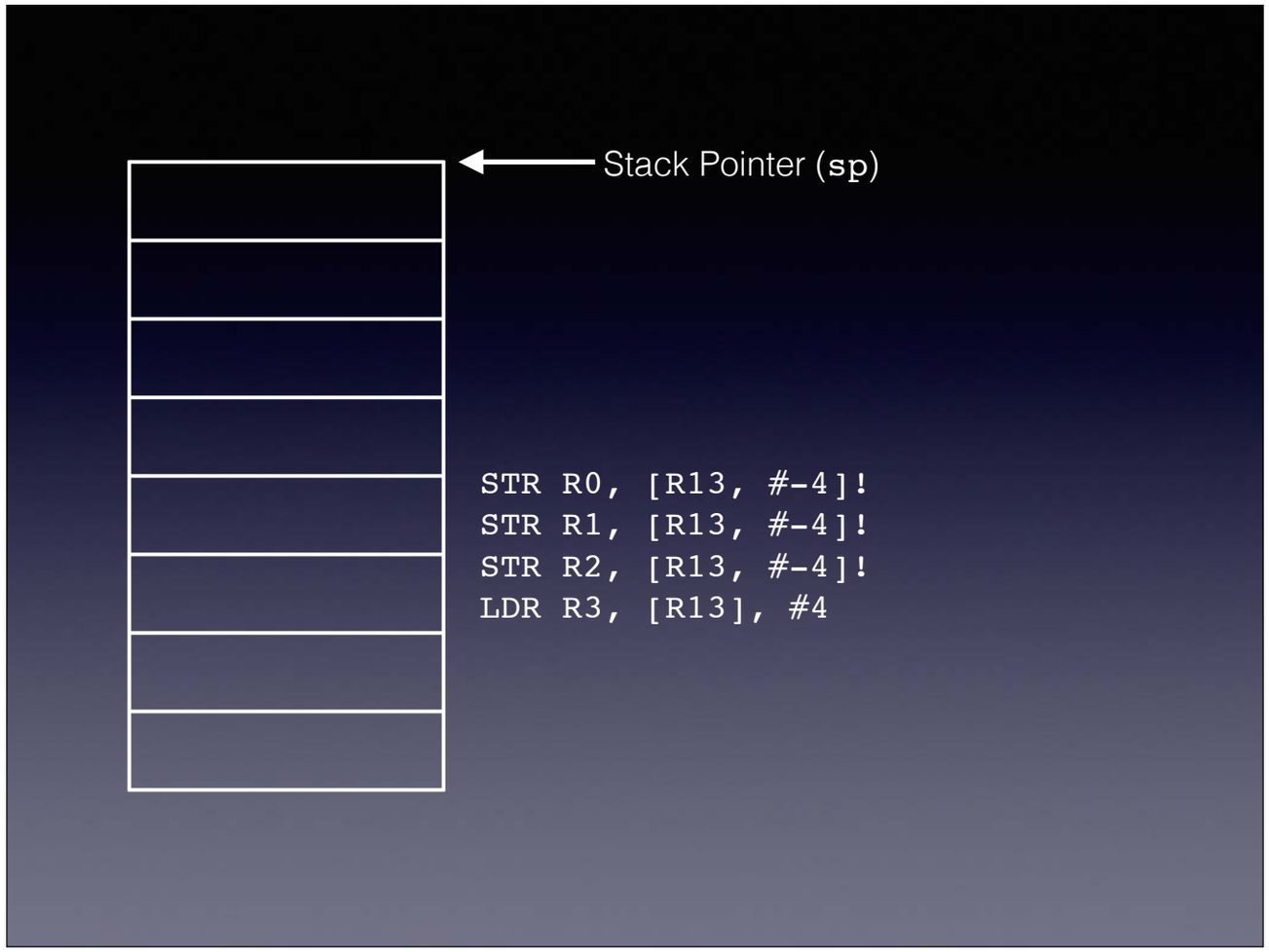
- And post-increment to pop

```
LDR R0, [R13],#4
```

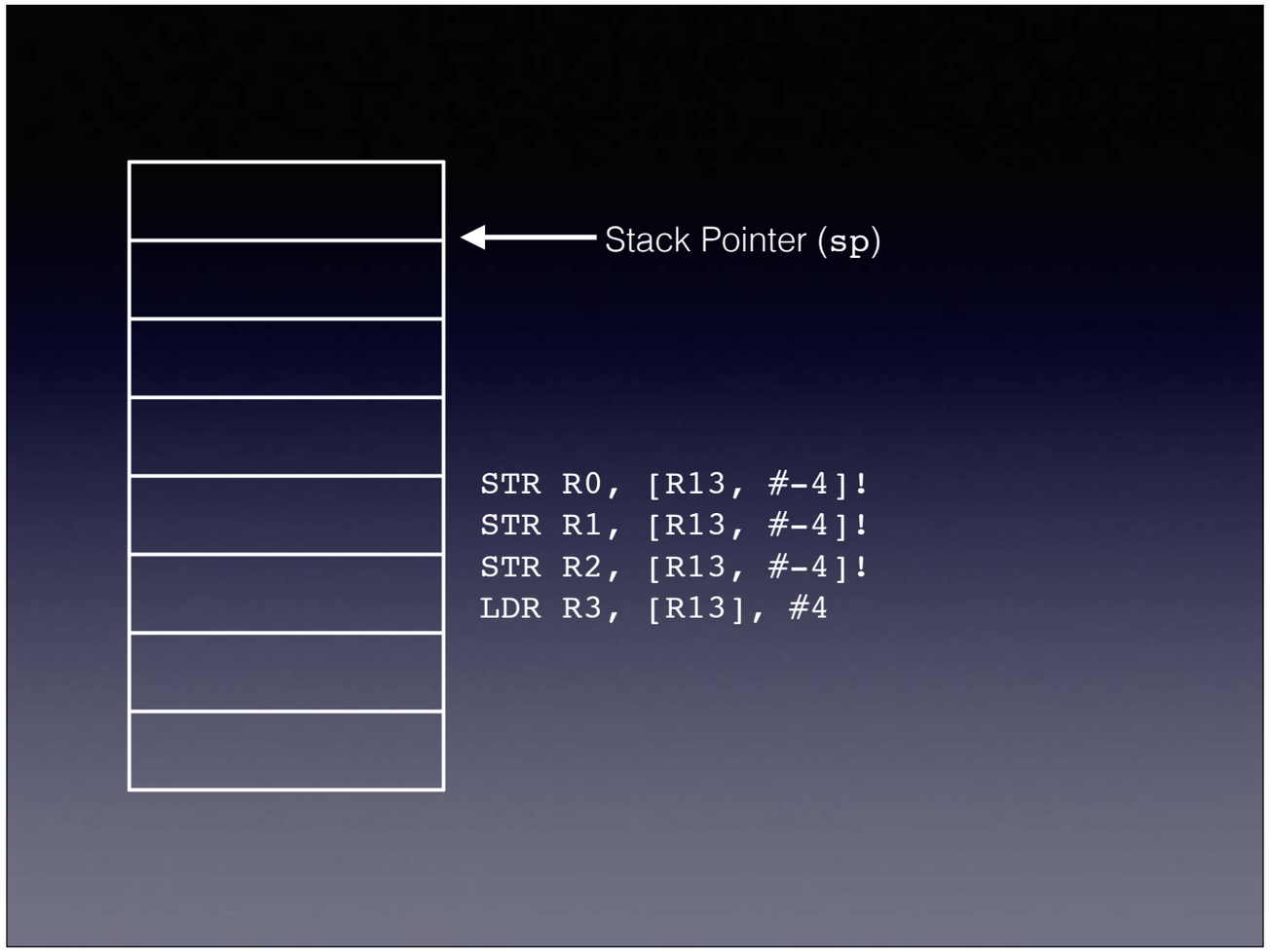
You might need to draw a stack here and show how it works

```
STR R0, [R13, #-4]!
STR R1, [R13, #-4]!
STR R2, [R13, #-4]!
LDR R3, [R13], #4
```

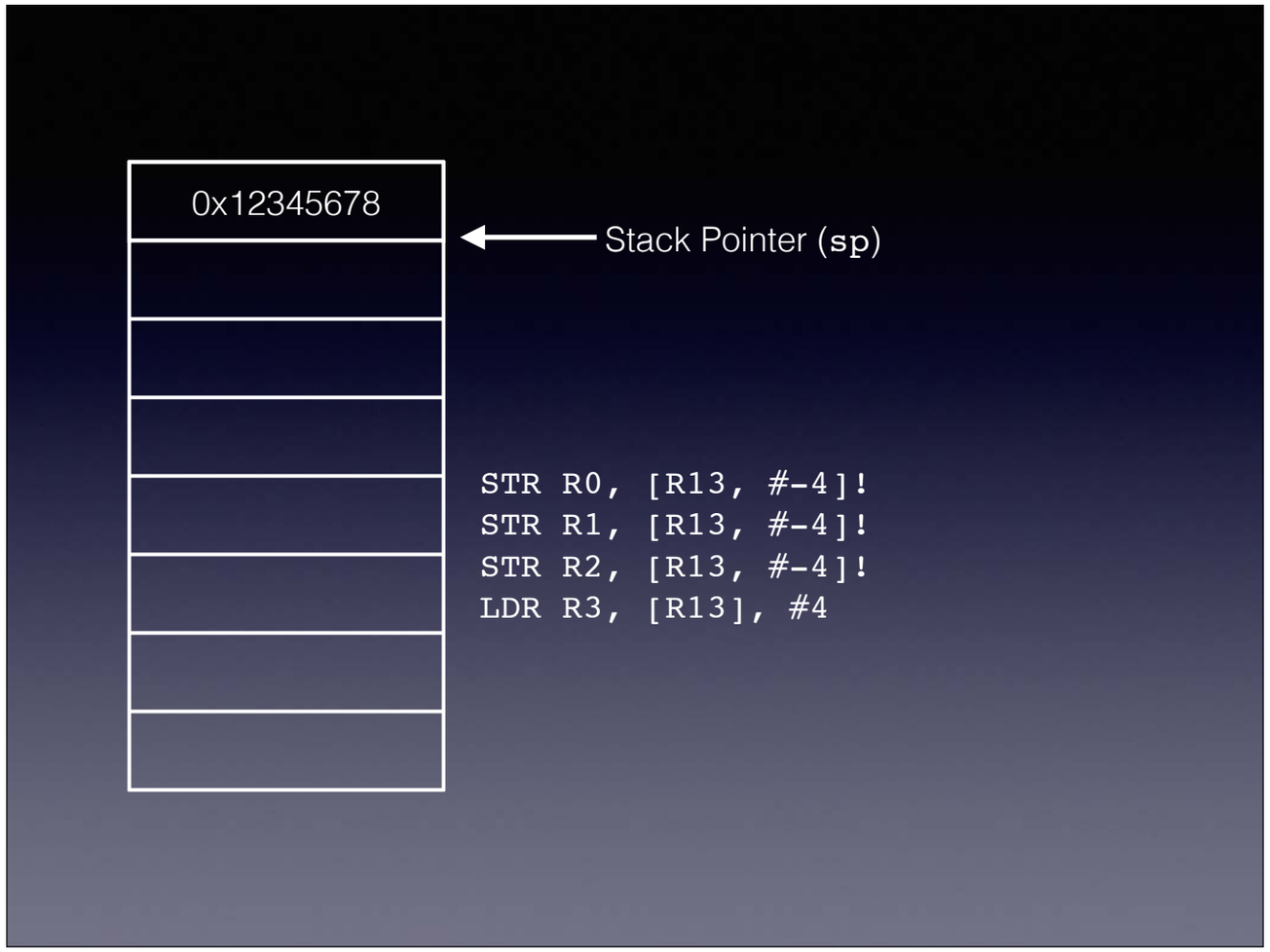
Note that after a value has been ‘popped’ off the stack its still left in memory



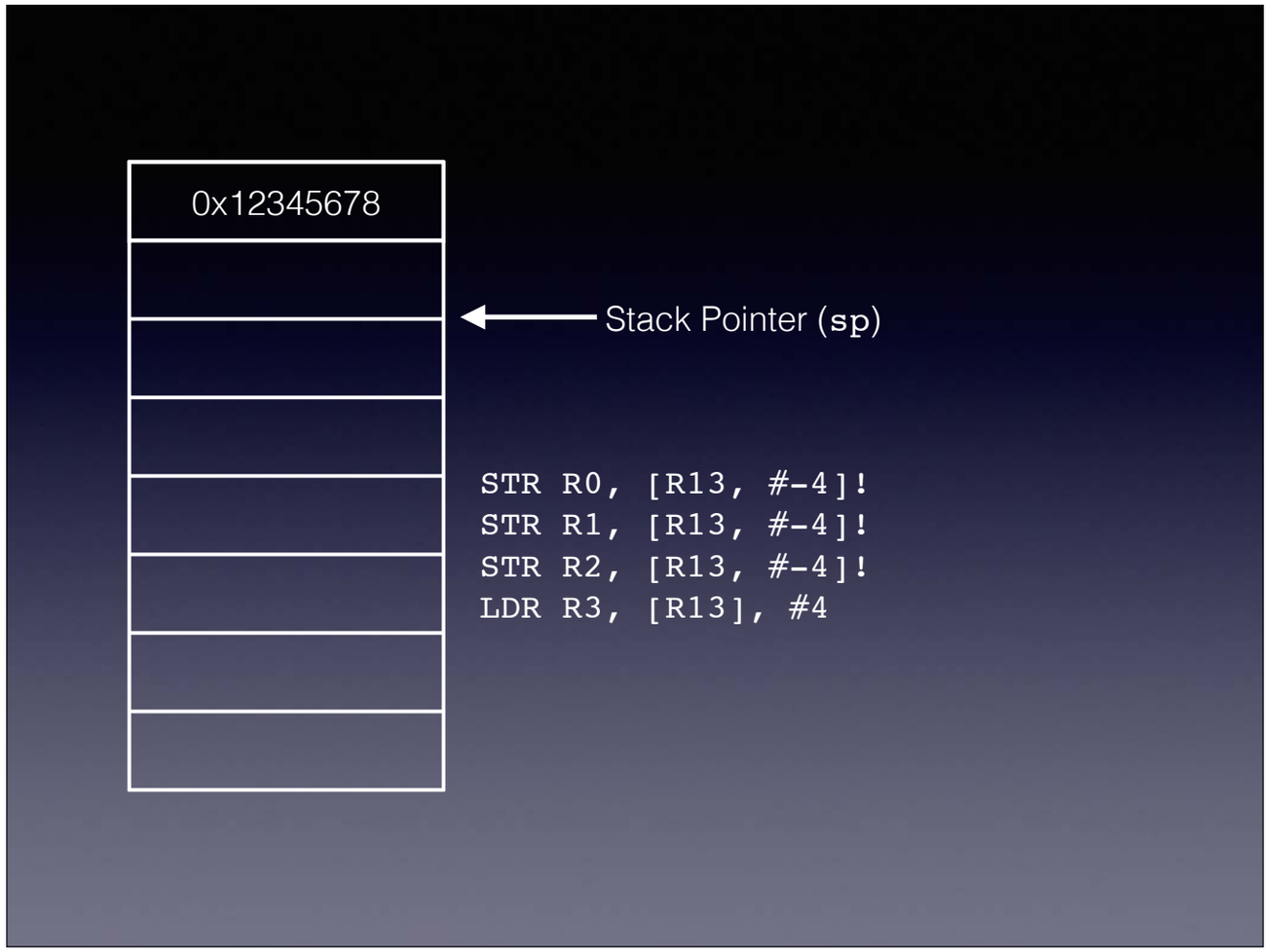
Note that after a value has been ‘popped’ off the stack its still left in memory



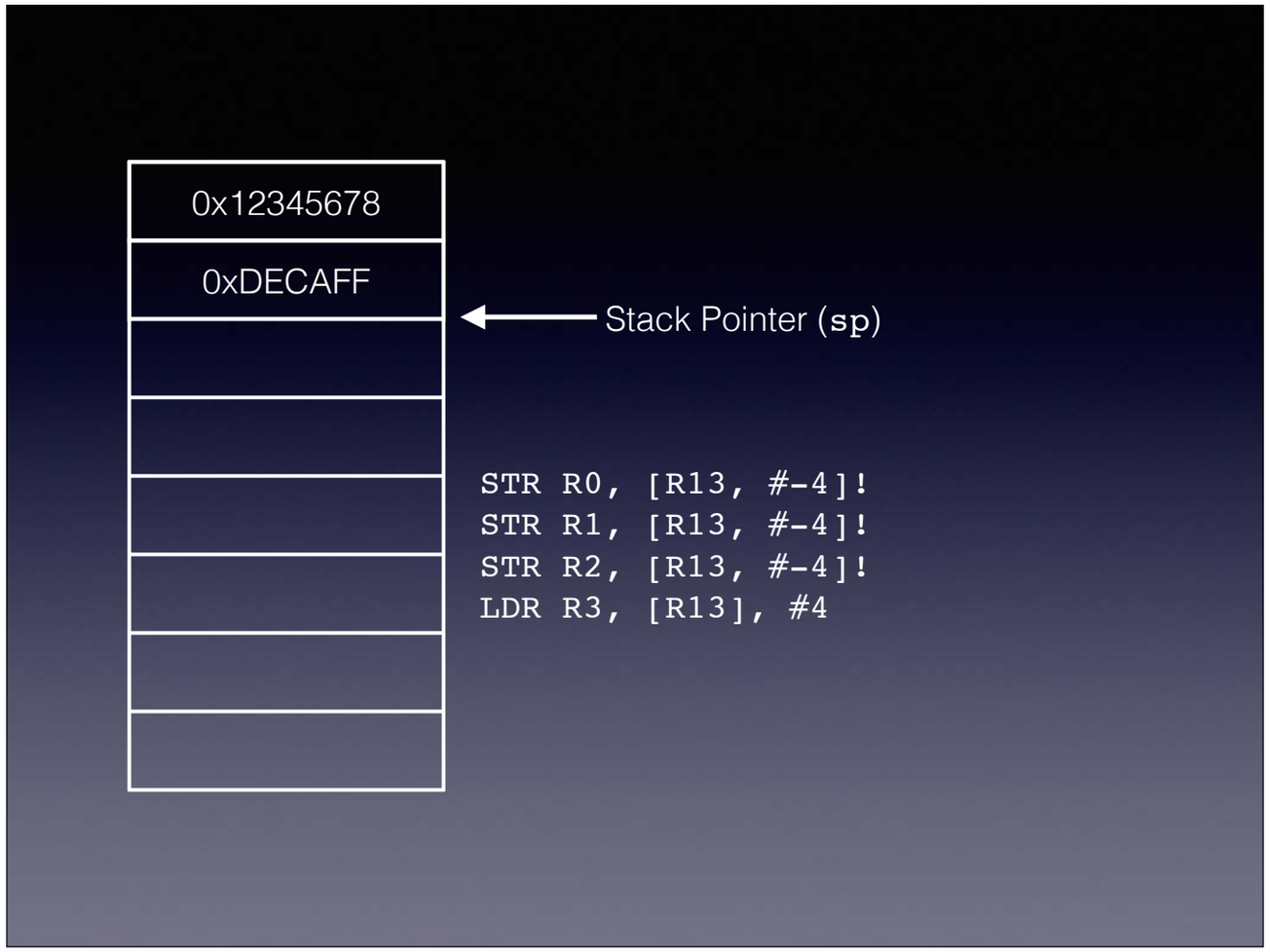
Note that after a value has been ‘popped’ off the stack its still left in memory



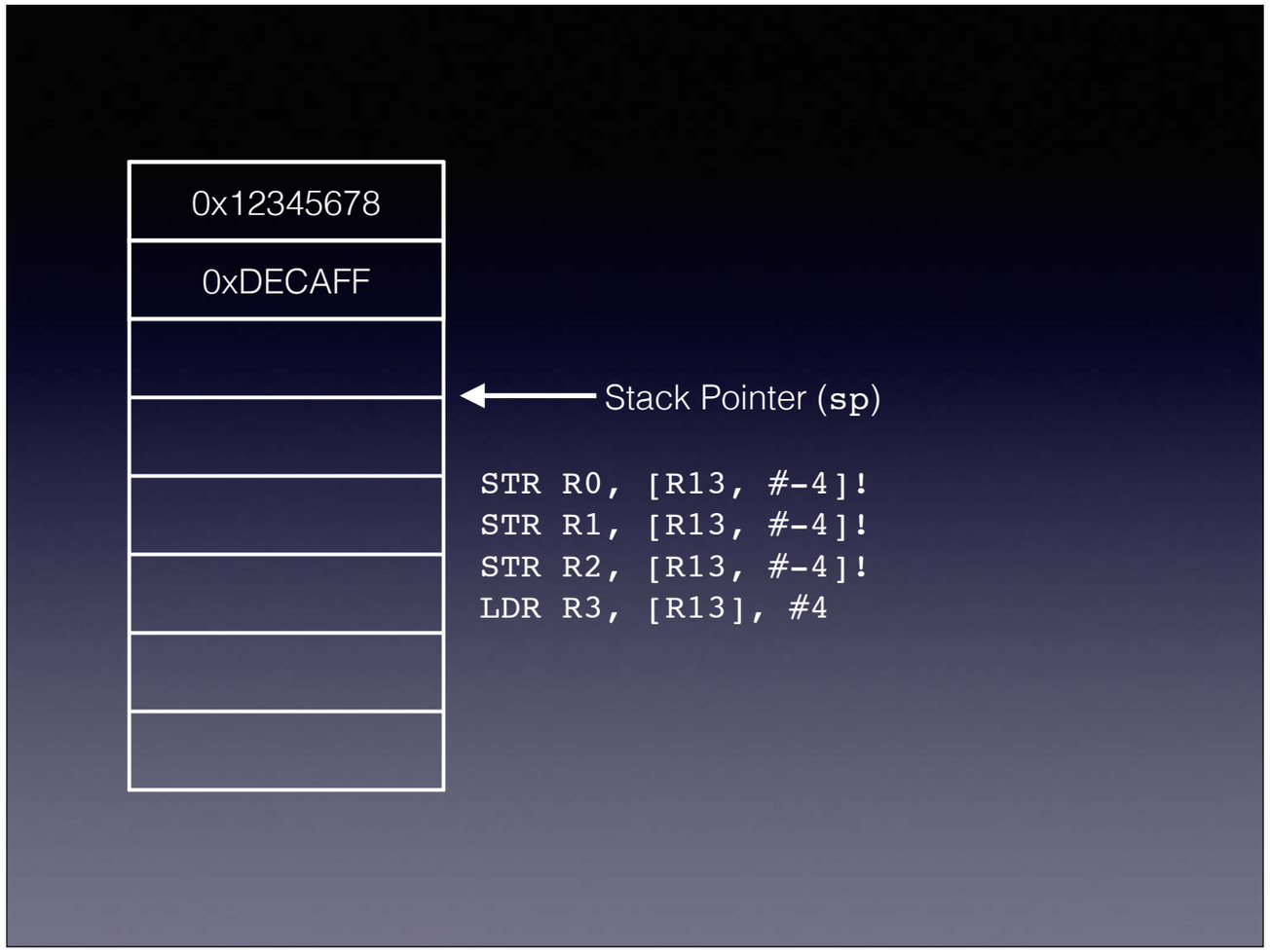
Note that after a value has been ‘popped’ off the stack its still left in memory



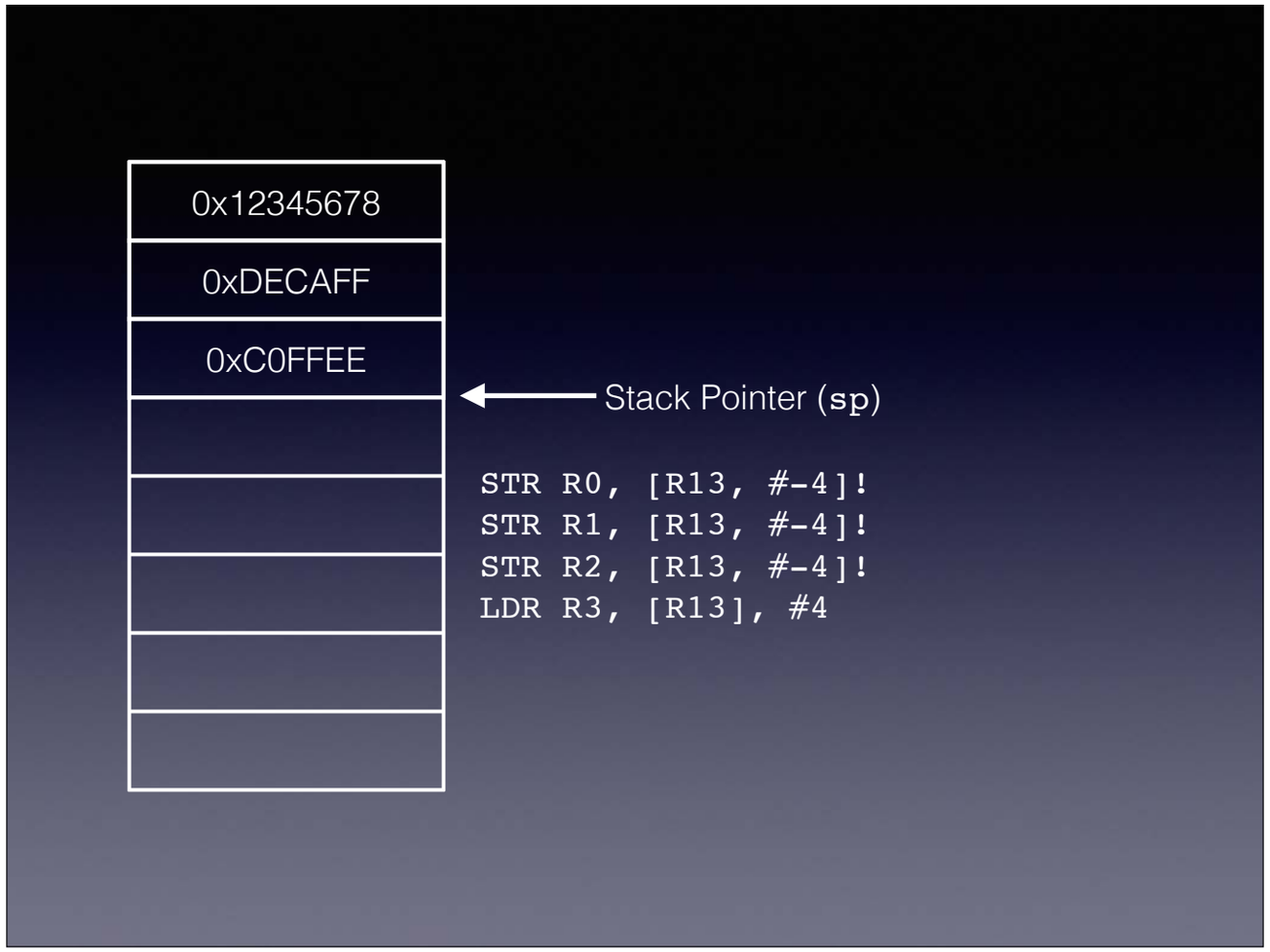
Note that after a value has been ‘popped’ off the stack its still left in memory



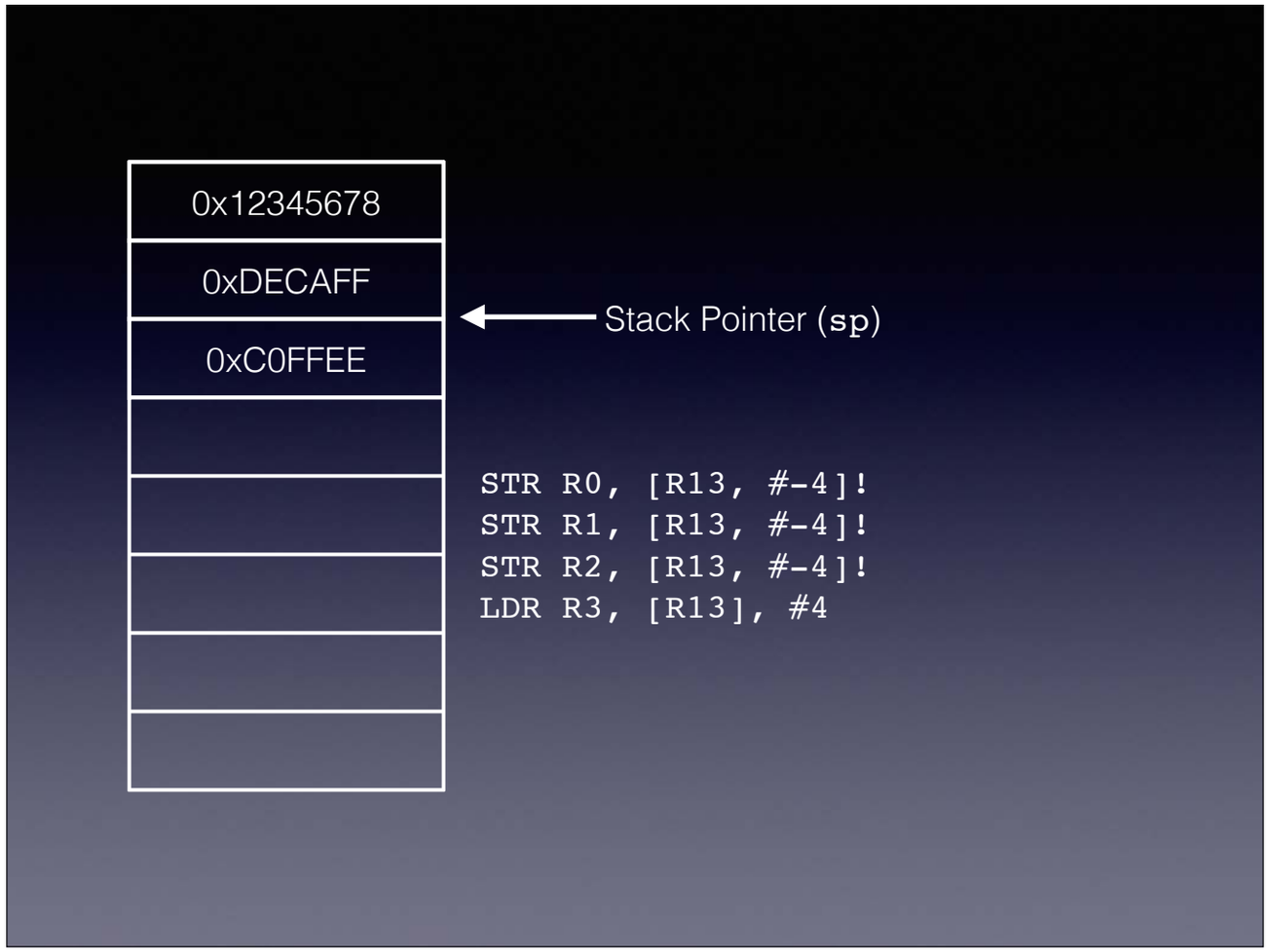
Note that after a value has been ‘popped’ off the stack its still left in memory



Note that after a value has been ‘popped’ off the stack its still left in memory



Note that after a value has been ‘popped’ off the stack its still left in memory



Note that after a value has been ‘popped’ off the stack its still left in memory

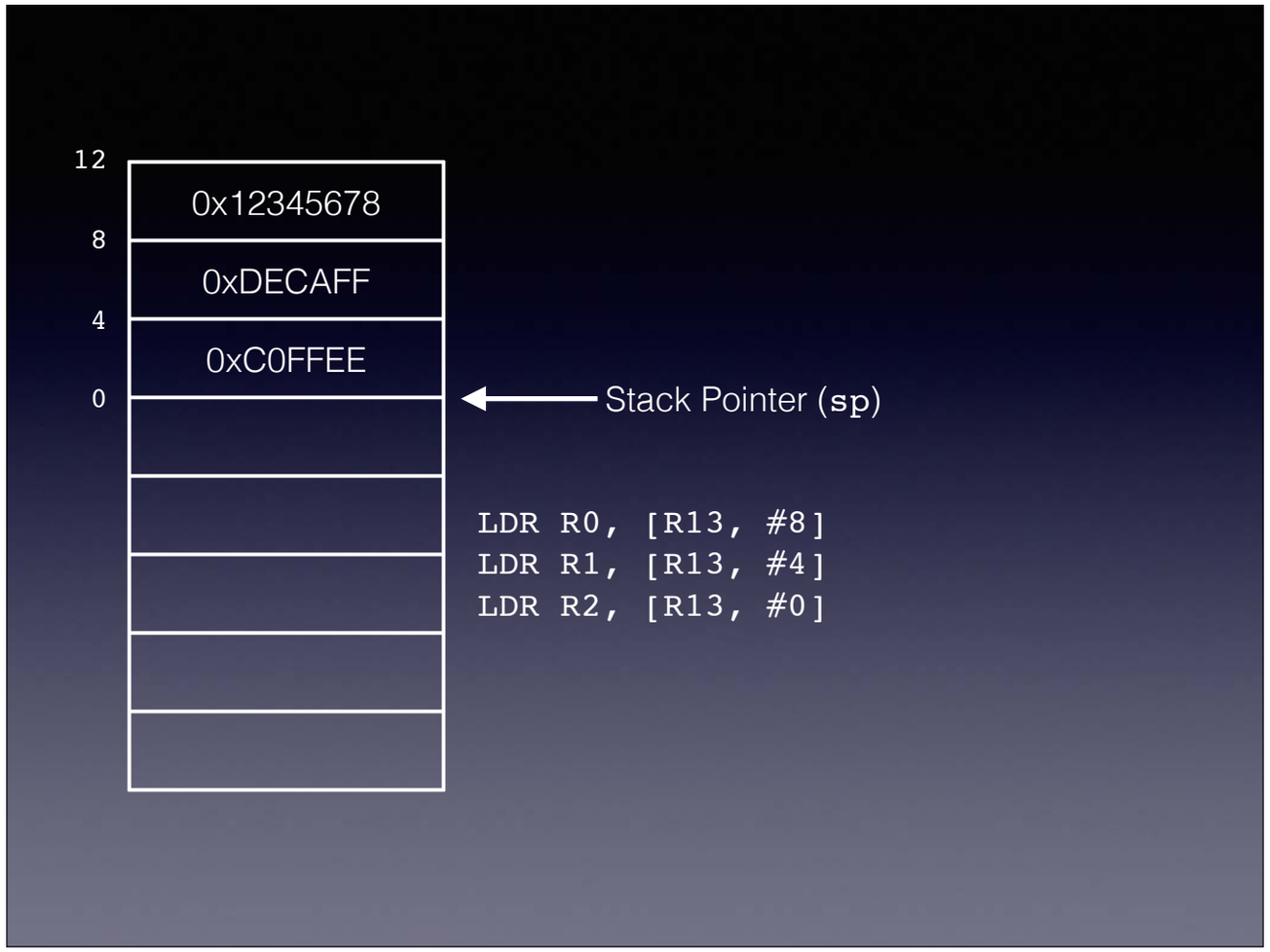
Stack Implementation

- Can also access values pushed on the stack without popping them
- Just use an offset from the stack point `R13`
- For example, to access the third thing on the stack

```
LDR R0, [R13, #8]
```

- Offsets are positive because we use a full-descending stack

If we'd have used an ascending stack we would have needed to use negative offsets…



Positive offsets in memory allow us to access values on the stack (although there’s no change to the stack pointer

Multiple Loads and Stores

- Stack is a good place to preserve register values
- ARM provides instructions to load and store registers en bloc
- Rather than having to use multiple `STR` and `LDR` operations
- Using the `LDM` and `STM` instructions (Load/Store Multiple)

Multiple Loads and Stores

- These instructions again use a base register, with an option for write-back
- For the main stack, we use `R13` as the base register
- `LDM`/`STM` both require a suffix depending on the stack regime
- For APCS, we use `LDMFD` and `STMFD`

FD = full-descending

`LDM/STM` Addressing Modes

- ARM instruction set provides four addressing modes for `LDM/STM`
- Addressing mode is provided as a suffix to the instruction, e.g. `LDMIA`, `STMDB`
- Describes whether the addresses are incremented or decremented
- And whether it happens before or after the values are loaded/stored

`LDM/STM` Addressing Modes

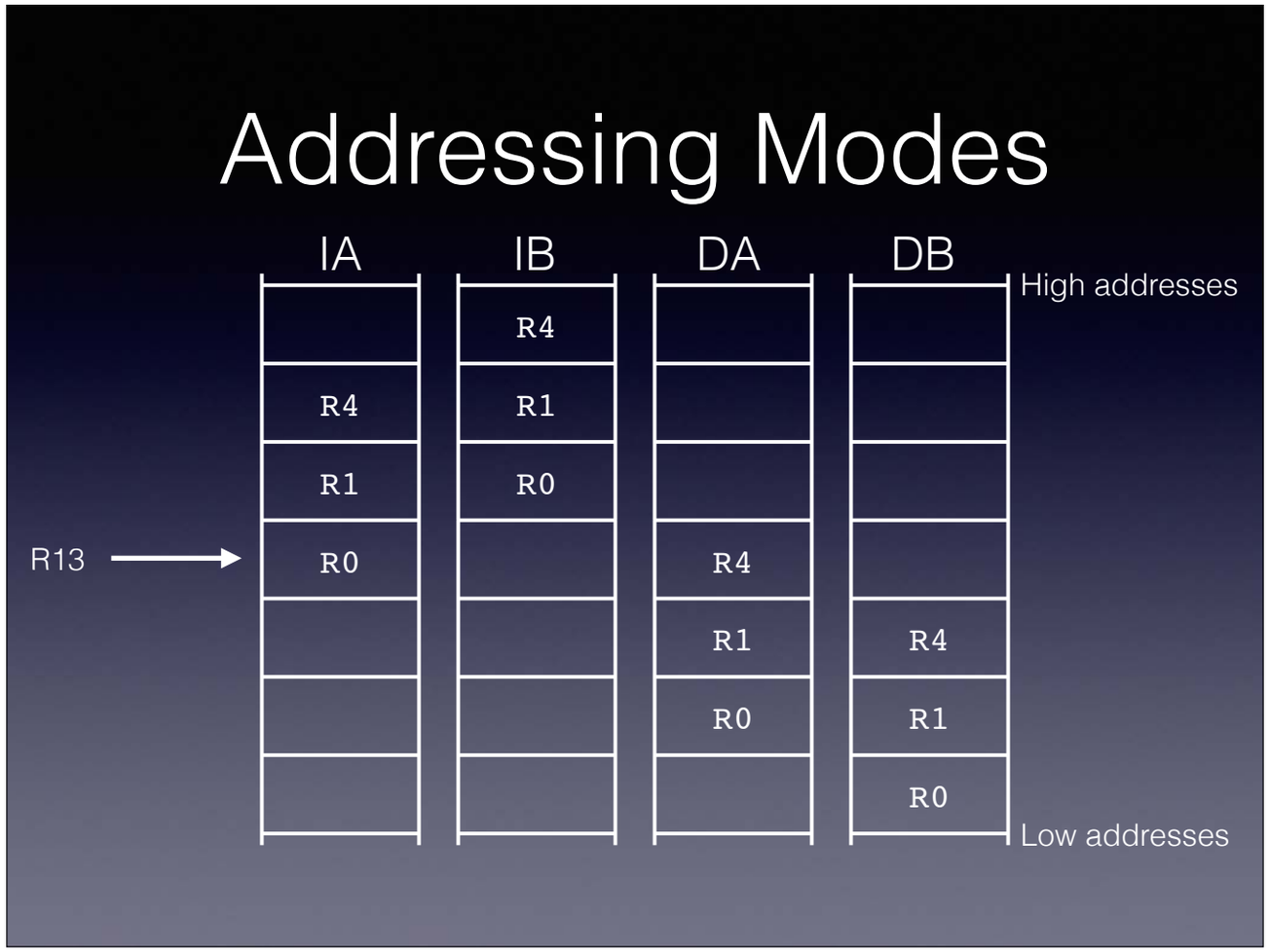
Suffix	Meaning
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

Stack Oriented Suffixes

Stack Type	Push	Pop
Full, Descending	`STMFD (STMDB)`	`LDMFD (LDMIA)`
Full, Ascending	`STMFA (STMIB)`	`LDMFA (LDMDA)`
Empty, Descending	`STMED (STMDA)`	`LDMED (STMIB)`
Empty, Ascending	`STMEA (STMIA)`	`LDMEA (STMDB)`

Assemblers also provide stack-oriented suffixes, where you tell it what stack regime you are using. Assembler then maps this automatically to the correct addressing mode (shown in brackets). We'll only worry about the first type

Addressing Modes



The effect of the different addressing modes for STM/LDM starting at R13

Full Descending Stack

- With a Full Descending stack, a multiple store (`STMFD`) corresponds to *push*ing registers onto the stack
- Conversely, a multiple load (`LDMFD`) corresponds to a *pop* from the stack
- Could use `STMDB` and `LDMIA` as well

Full Descending Stack

- Consider `LDMFD R13, {R0-R3}`
- Registers specified between curly braces
- This is equivalent to:

```
LDR R0, [R13]
LDR R1, [R13, #4]
LDR R2, [R13, #8]
LDR R3, [R13, #12]
```

- But notice that `R13` isn't updated

Stack Writeback

- If we want the stack pointer, `R13`, to be updated then we need to specify that we want write back
- Done by placing a ! after the base register
- For example:
  `LDMFD R13!, {R0-R3}`

LDMxx/STMxx

- Register list is specified between curly braces
- Use commas to separate them and a hyphen to specify a range, e.g.
  `LDMFD R13!, {R0-R5, R8, R14}`
- Pops `R0`, `R1`, `R2`, `R3`, `R4`, `R5`, `R8` and `R14`
- Remember the item in the lowest address goes to the lowest register number

Stack Frames

- Data stored on the stack as part of a function call forms part of the *stack frame* for that invocation
- Stack frames are used to store register values, but also to create space for local variables used within the function
- Also used to preserve the link register (`R14`)

Local variables go on the stack because you can then be sure of a unique instance of them for that invocation. Functions may be called while the function is already running (e.g. recursion, multi-threaded code, etc.) Can’t always just be stored in registers (Even if we have enough of them, since we might need to pass the address to something)

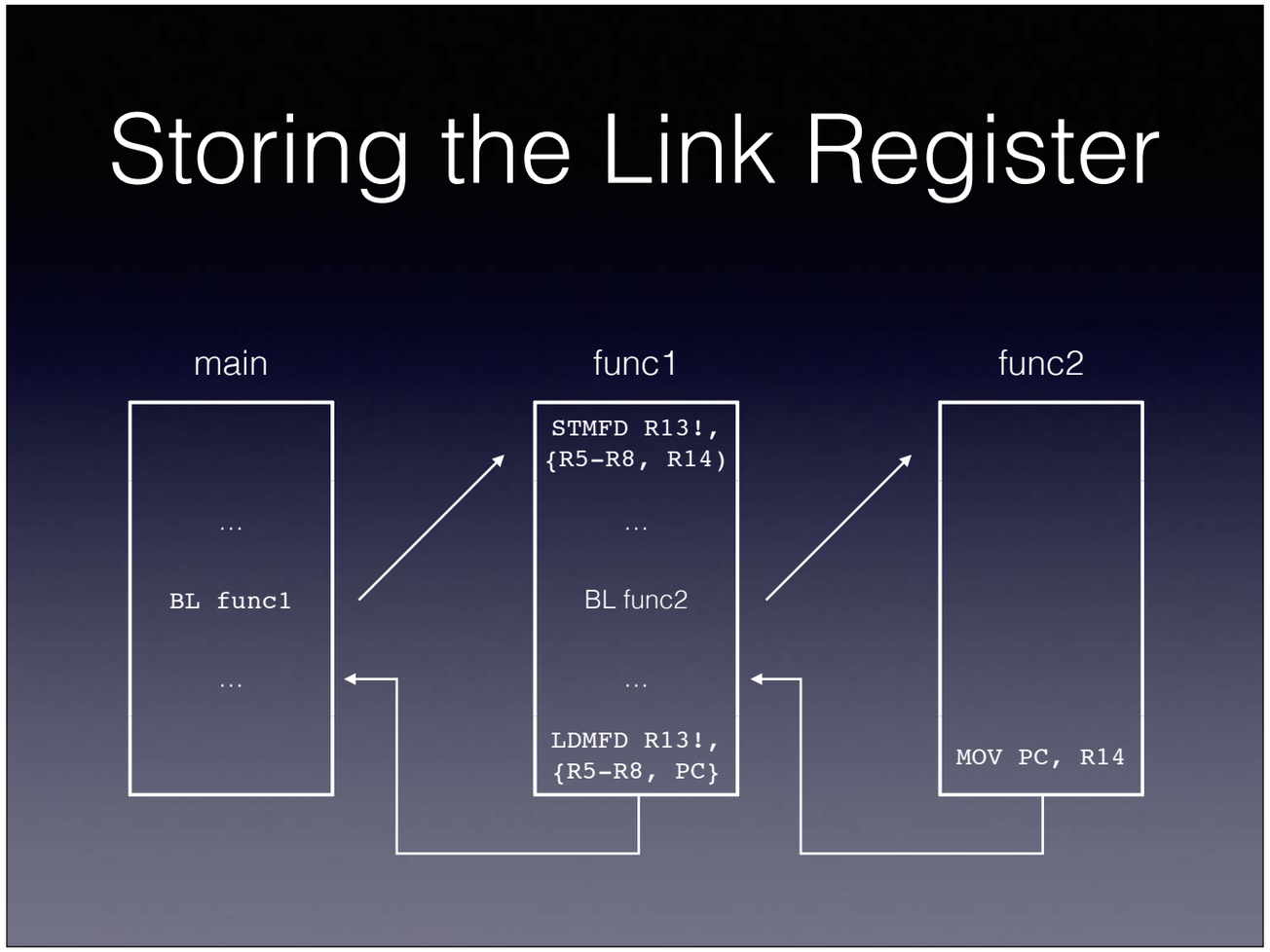
Stack Frames

- When a procedure exits and return to the caller, everything it put on the stack must be popped
- This is why local variables vanish once a function exits
- The caller expects the stack to be exactly as it left it

Storing the Link Register

- If we are a leaf function (i.e. we don't call anything else) then there is no need to store the link register
- If we do call a function, then it is necessary to preserve the link register
- Why?

Storing the Link Register



Note how we restore to PC, not R14 — saves us an instruction

Storing the Link Register

- The `BL func1` in main stores the return address in `R14`
- But then the `BL func2` inside `func1` overwrites it
- So `func2` returns correctly to `func1`
- But if `func1` were to return using `MOV PC, R14` then `R14` would have the wrong value

Storing the Link Register

- Non-leaf functions definitely need to stack the link register value
- But note the stack frame push/pop in **`func1`**
- The **`LDMFD`** restores the stored return address directly into the **`PC`**
- This causes an instantaneous return to **`main`**

Local Variables

- Local variables are stored on the stack
- Guarantees a unique instance of the variables for each function invocation
- Can easily create space for them on the stack by using a **`SUB`** instruction after we preserve the registers
- Use an **`ADD`** later to remove them all quickly…
- Can then use an offset from `R13` to access them
- But remember the offset will change as you push and pop more values…