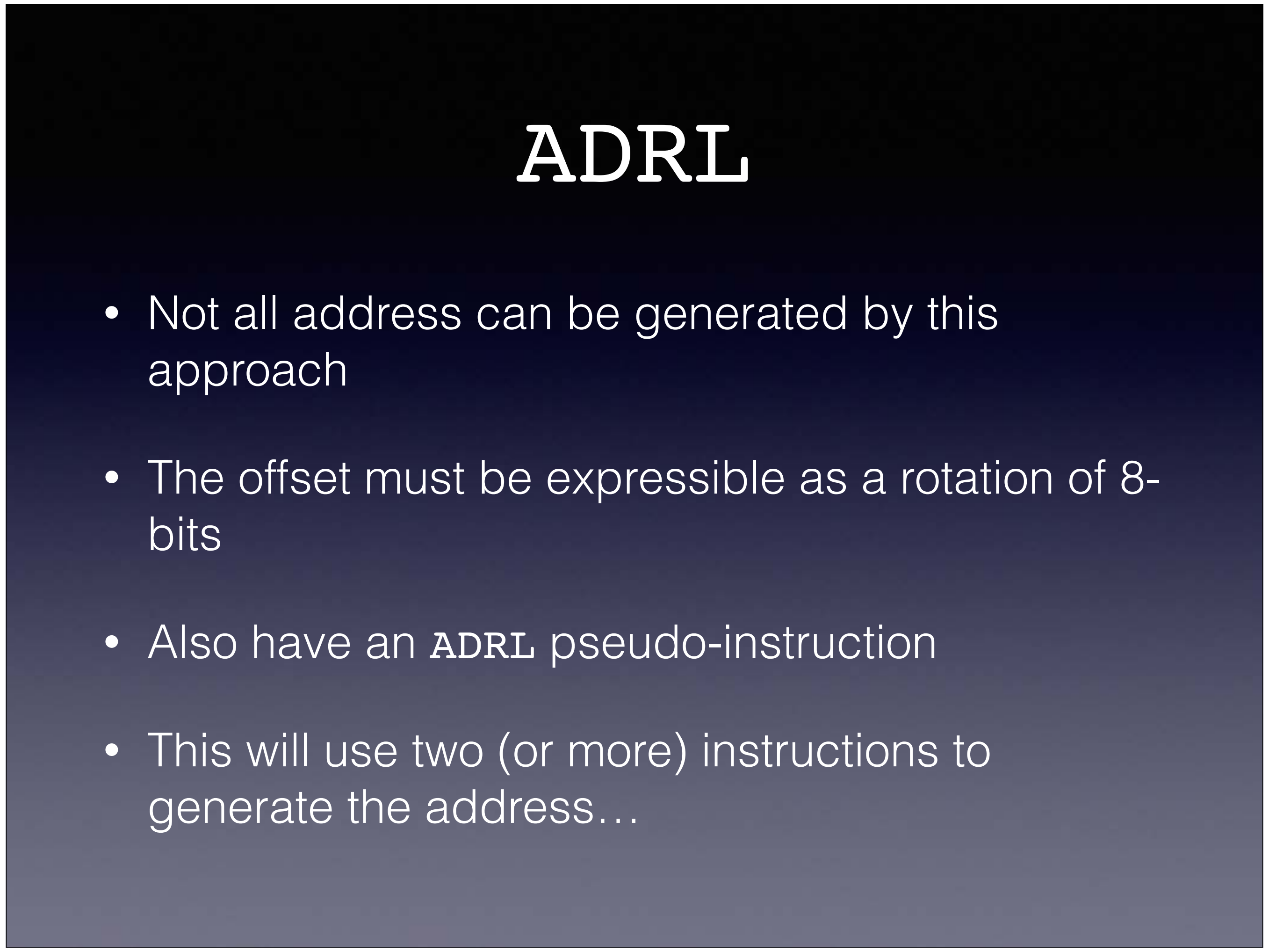
Addressing

Steven R. Bagley

`ADR` instruction

- If address is ‘in range’, then destination register can be set using a single-instruction by adding/subtracting an offset to the value in `PC`
- Offset must be expressible in 8-bits with rotation
- So `ADR R0,one` might become `SUB R0, PC, #&18` or `ADD R0, PC, #&80`
- Depends on the relationship between the label and the instruction’s location

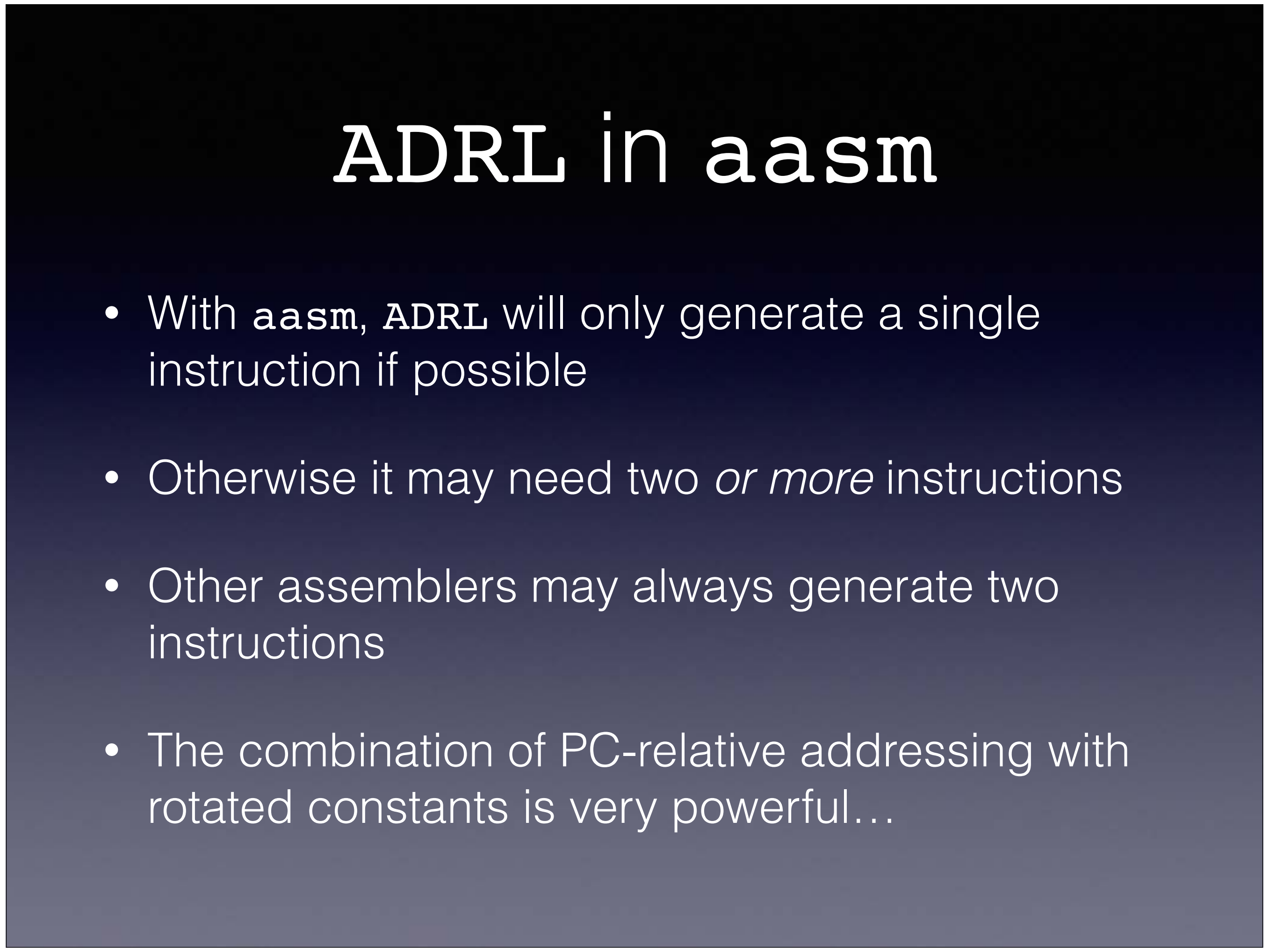


ADRL

- Not all address can be generated by this approach
- The offset must be expressible as a rotation of 8-bits
- Also have an **`ADRL`** pseudo-instruction
- This will use two (or more) instructions to generate the address…

Again usually ADD or SUB instructions

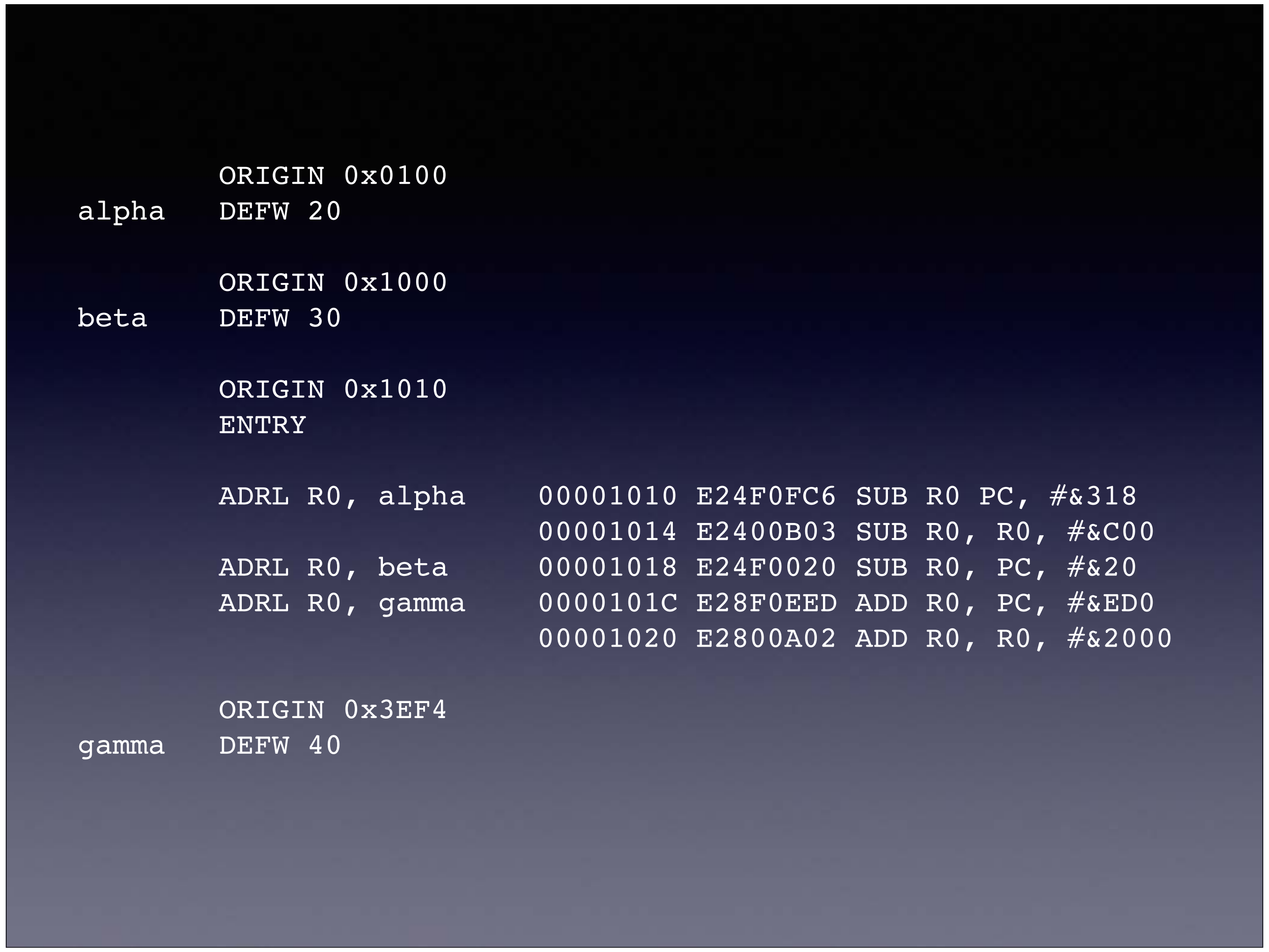
Given an example — use origin to demo it



`ADRL` in `aasm`

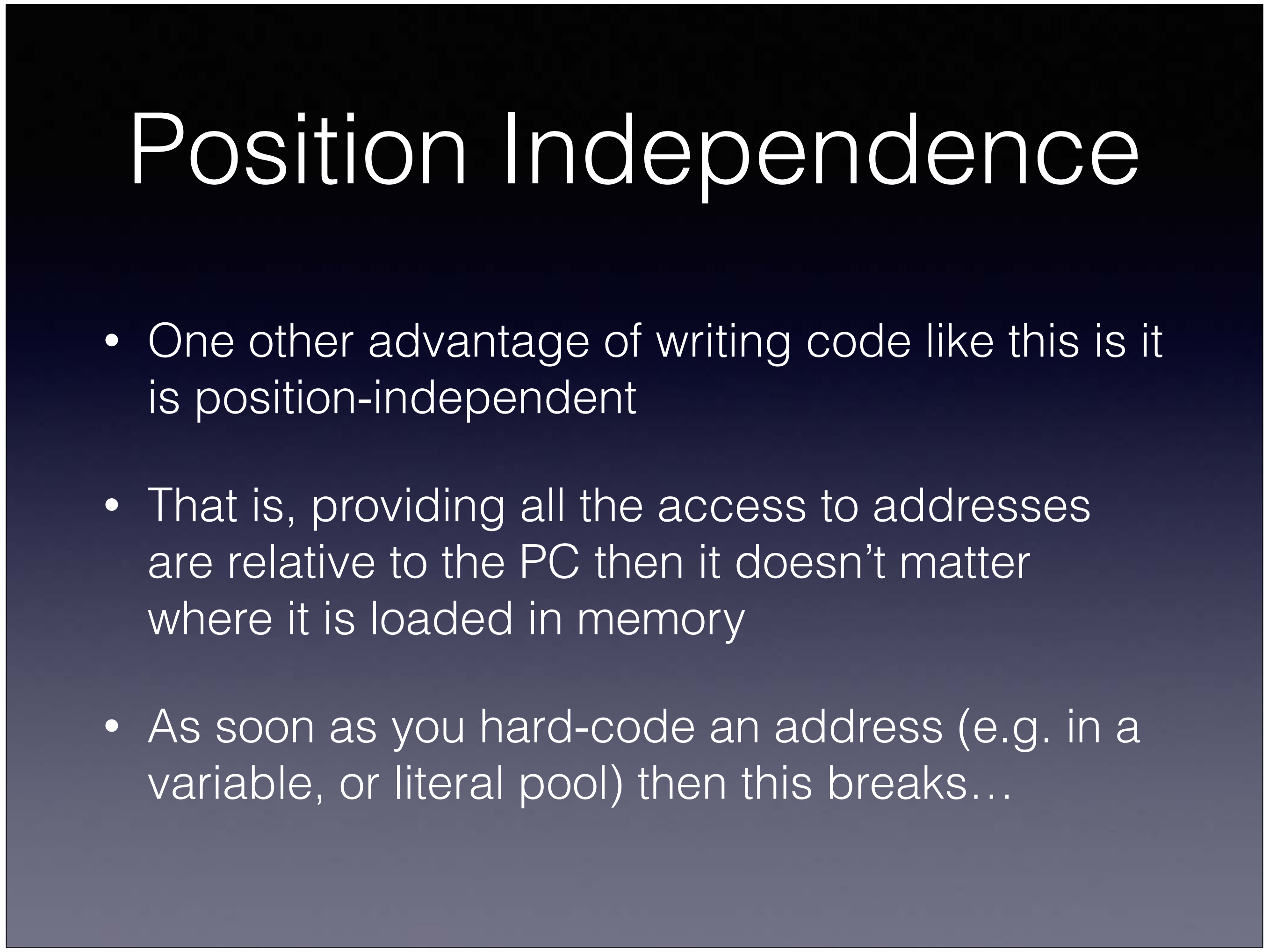
- With `aasm`, `ADRL` will only generate a single instruction if possible
- Otherwise it may need two *or more* instructions
- Other assemblers may always generate two instructions
- The combination of PC-relative addressing with rotated constants is very powerful…

Can be important when writing code to know exactly how many instructions are executed…



```
        ORIGIN 0x0100
alpha   DEFW 20

        ORIGIN 0x1000
beta    DEFW 30

        ORIGIN 0x1010
        ENTRY

        ADRL R0, alpha    00001010 E24F0FC6 SUB R0 PC, #&318
                          00001014 E2400B03 SUB R0, R0, #&C00
        ADRL R0, beta     00001018 E24F0020 SUB R0, PC, #&20
        ADRL R0, gamma    0000101C E28F0EED ADD R0, PC, #&ED0
                          00001020 E2800A02 ADD R0, R0, #&2000

        ORIGIN 0x3EF4
gamma   DEFW 40
```

Various examples of ADRL in action in aasm



Position Independence

- One other advantage of writing code like this is it is position-independent
- That is, providing all the access to addresses are relative to the PC then it doesn't matter where it is loaded in memory
- As soon as you hard-code an address (e.g. in a variable, or literal pool) then this breaks…

Doesn't mean the code cannot be loaded anywhere but the thing loading it will need to 'relocate' it to the new memory address…

Loads and Stores

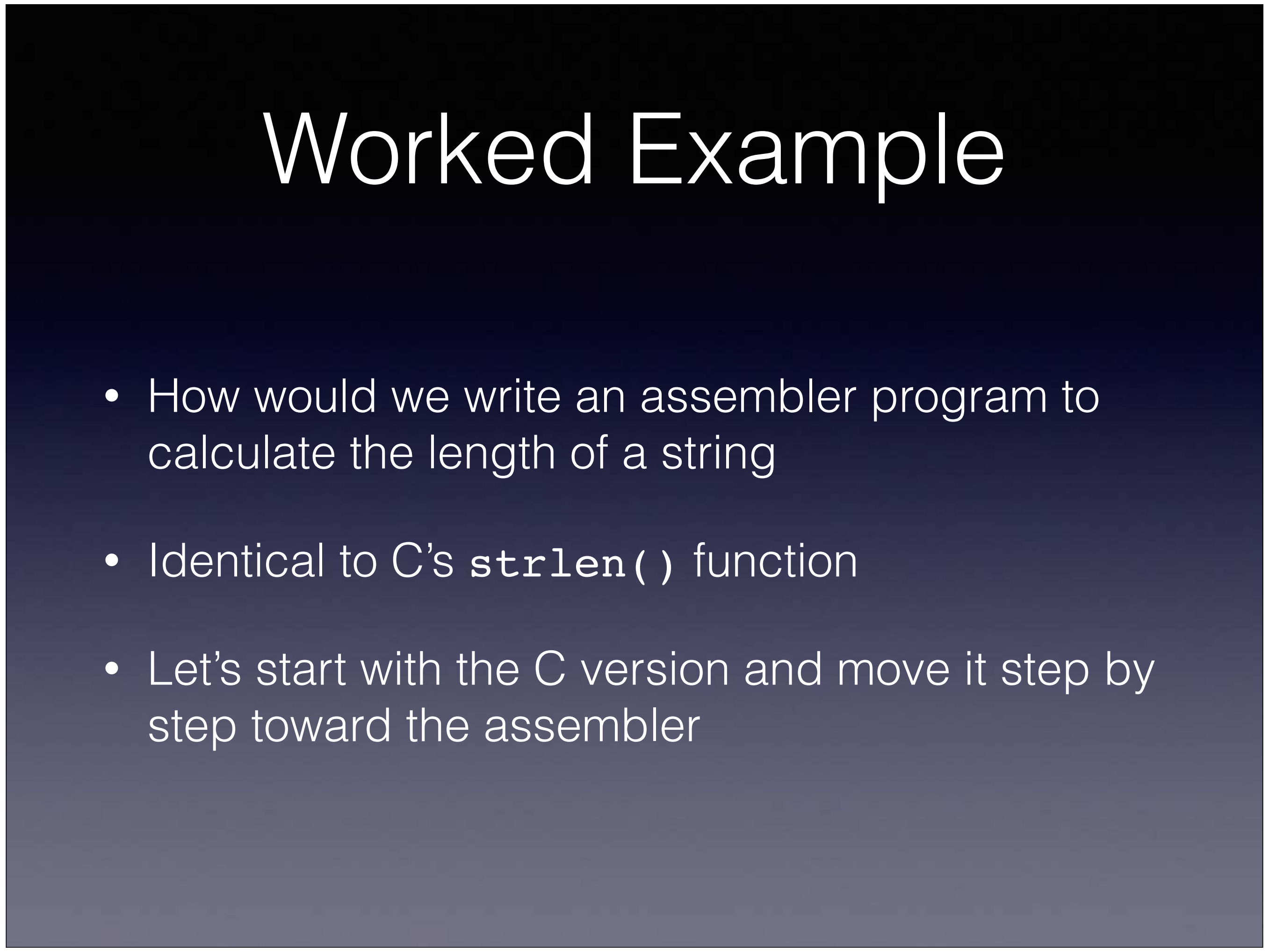
- So far we've seen `LDR` and `STR` in use to access a specific address, e.g.
  `LDR R0, one`
- On other systems, this is knows as *absolute addressing*
- But ARM doesn't support absolute addressing…
- Again the assembler is helping us out here…

Indirect Addressing

- ARM only supports *Indirect Addressing*
- Works just like a pointer in C
- A register contains the address to look the value up in
- CPU gets the value in memory at the address in the register

Indirect Addressing

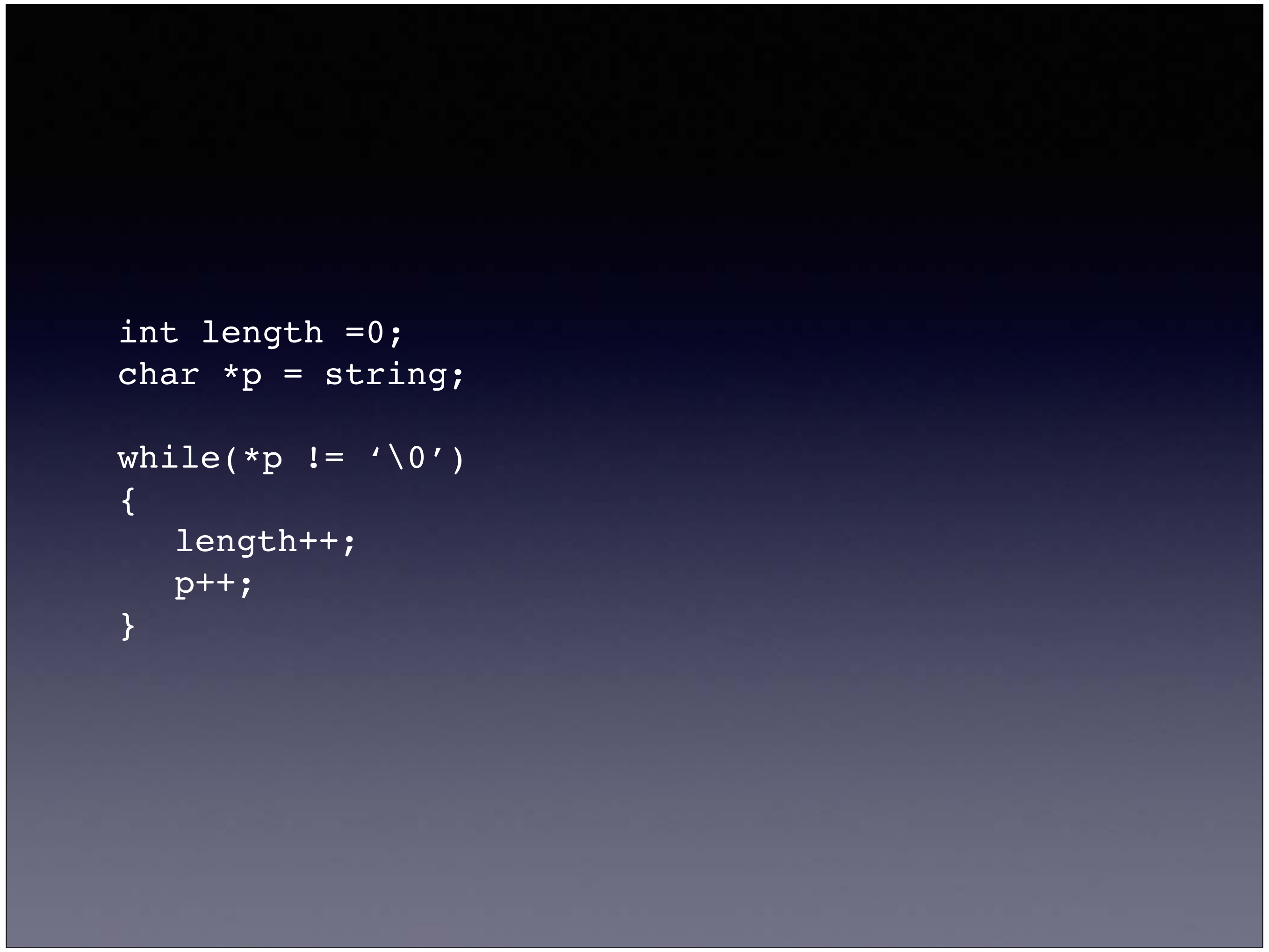
- Load the register with the address
```
ADR R1, mesg
```
- Then use that register in square brackets to access the value
```
LDRB R0, [R1]
STR R4, [R2]
```
- If we change the value in the register we change the address looked up



Worked Example

- How would we write an assembler program to calculate the length of a string
- Identical to C's `strlen()` function
- Let's start with the C version and move it step by step toward the assembler

Go demo…



```
int length =0;
char *p = string;

while(*p != ‘\0’)
{
   length++;
   p++;
}
```

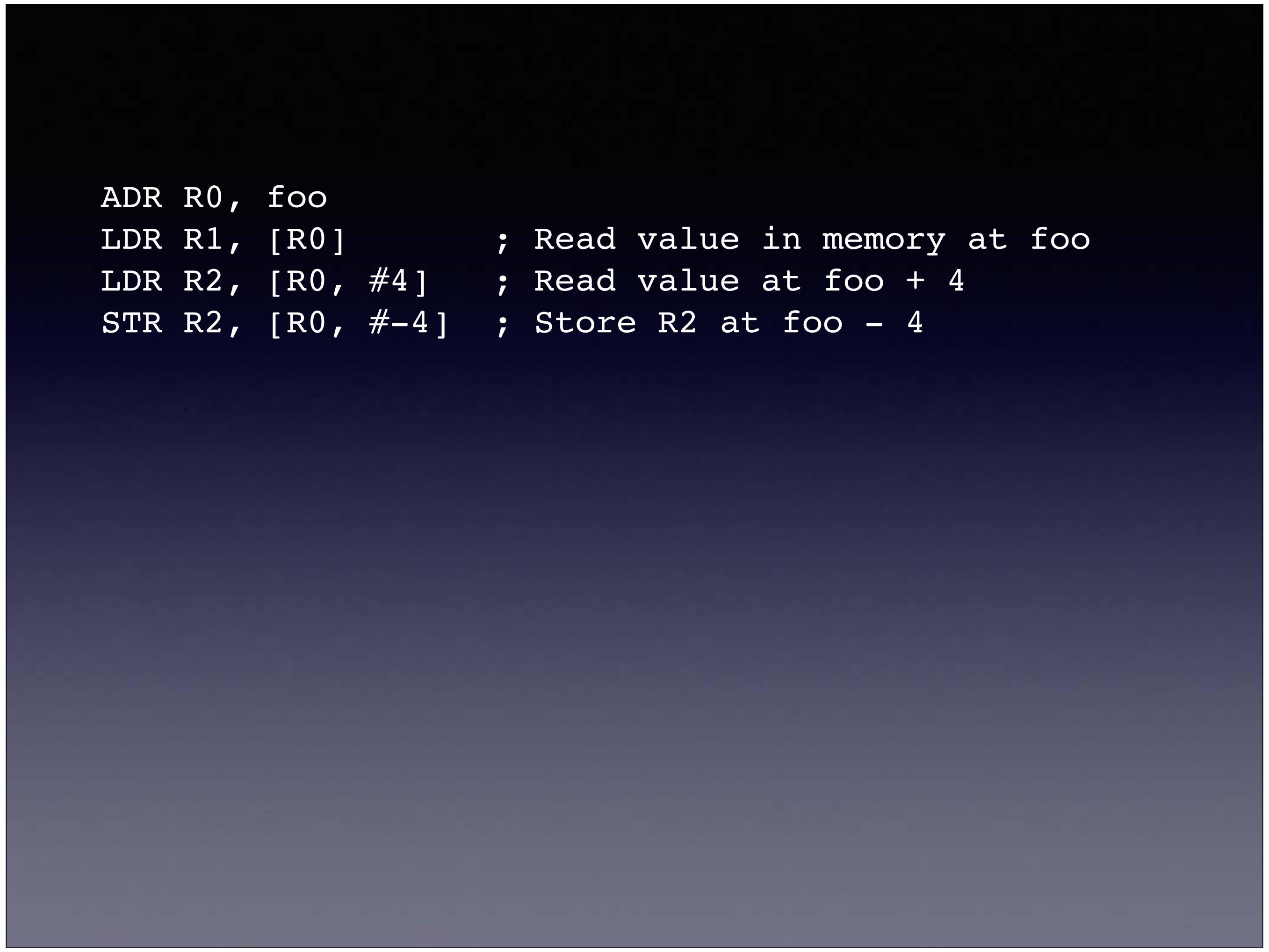
A string length routine…

Absolute Addressing

- So how does the assembler handle ‘absolute addressing’
- Does it generate the address in a register and do an indirect lookup on that?
- Not quite…
- Uses the `PC` again…
- With an index offset

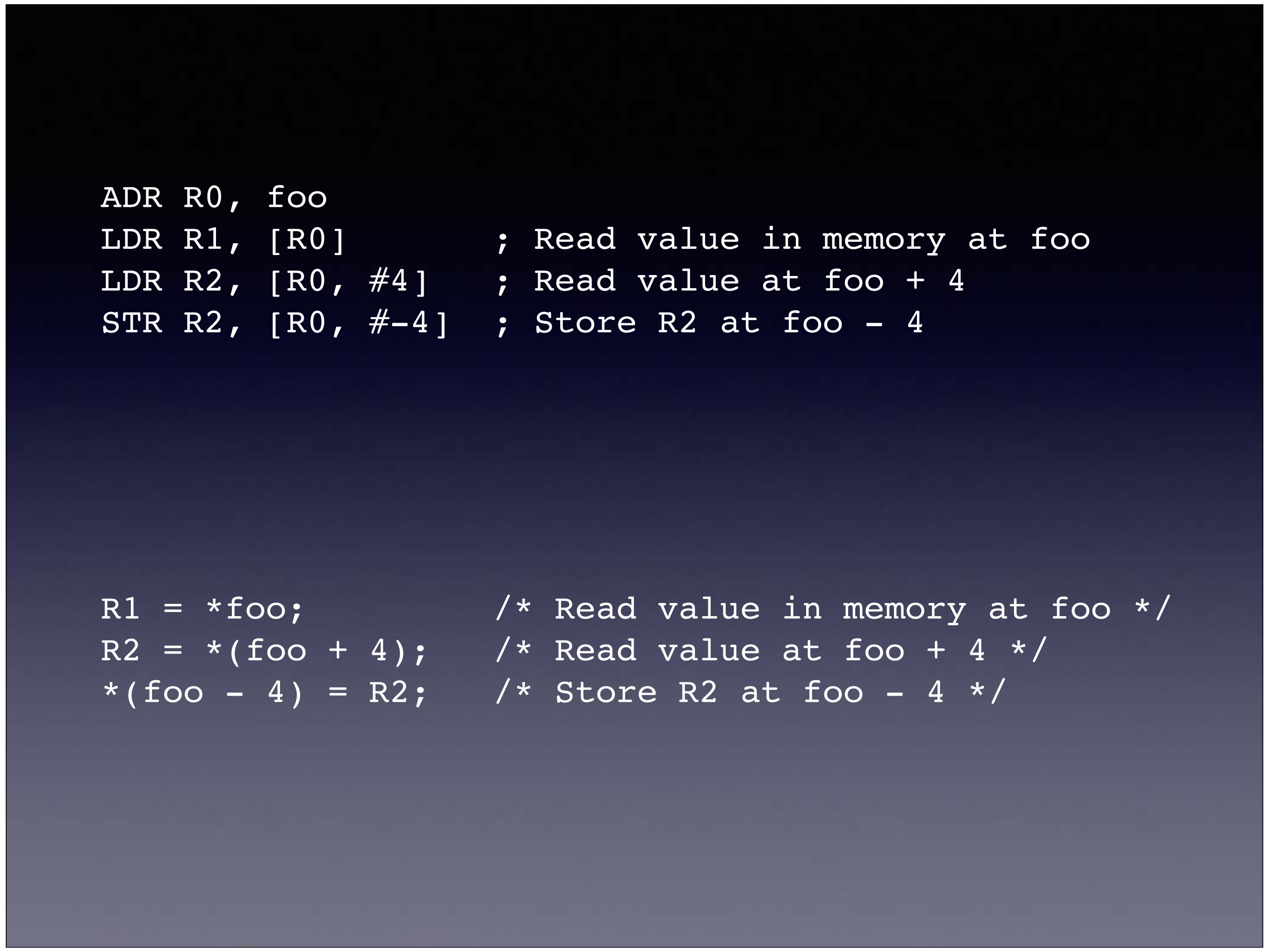
Indirect With Index

- The instruction layout for `LDR` and `STR` includes space for a 12-bit unsigned offset
- A further bit controls whether this is added or subtracted from the address in the register
- Yet another bit controls whether this happens before (pre-index) or after (post-index) the address is used to access memory



```
ADR R0, foo
LDR R1, [R0]       ; Read value in memory at foo
LDR R2, [R0, #4]   ; Read value at foo + 4
STR R2, [R0, #-4]  ; Store R2 at foo - 4
```

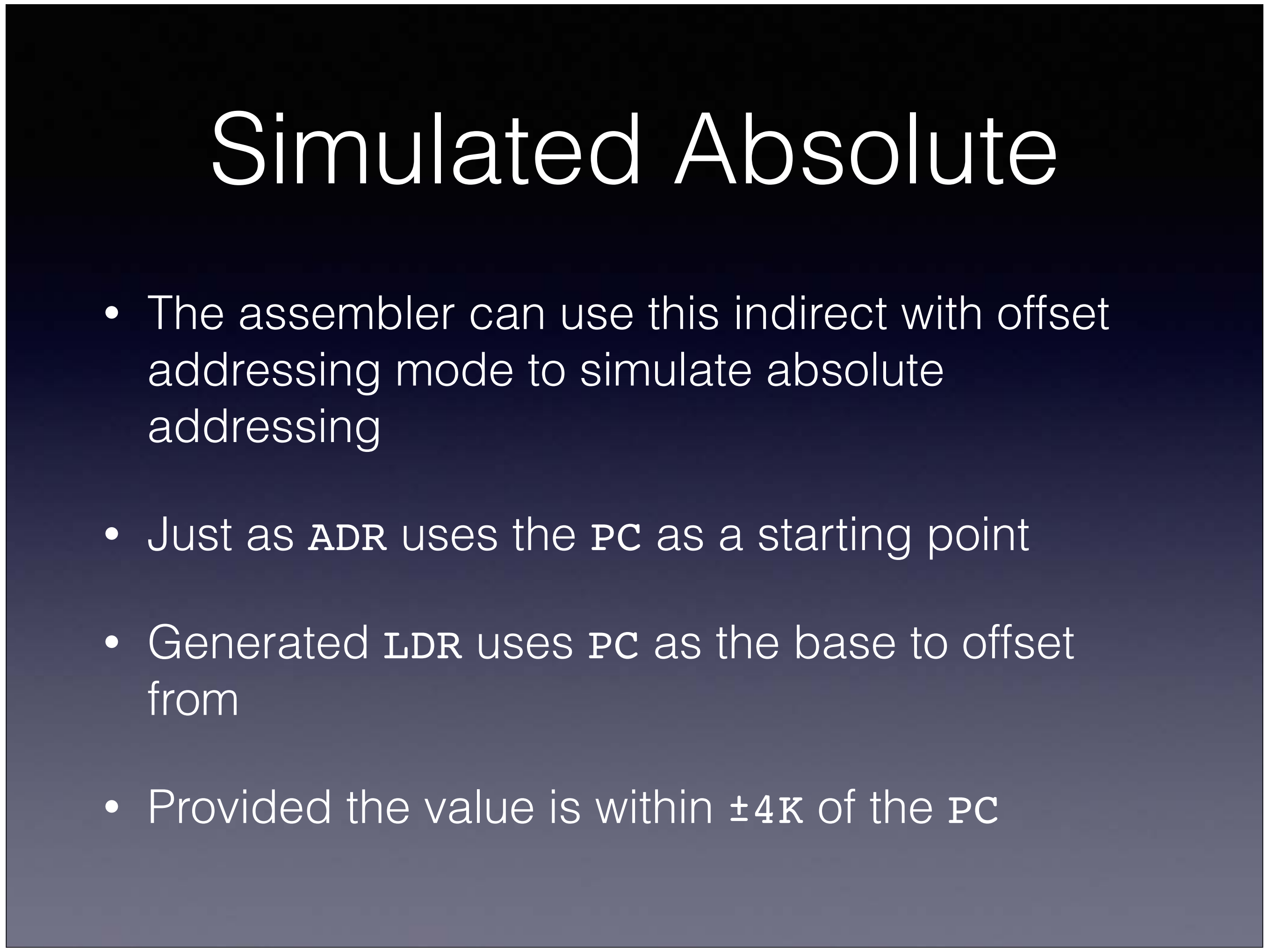
Good for accessing arrays…



```
ADR R0, foo
LDR R1, [R0]        ; Read value in memory at foo
LDR R2, [R0, #4]    ; Read value at foo + 4
STR R2, [R0, #-4]   ; Store R2 at foo - 4
```

```
R1 = *foo;          /* Read value in memory at foo */
R2 = *(foo + 4);    /* Read value at foo + 4 */
*(foo - 4) = R2;    /* Store R2 at foo - 4 */
```

Good for accessing arrays…



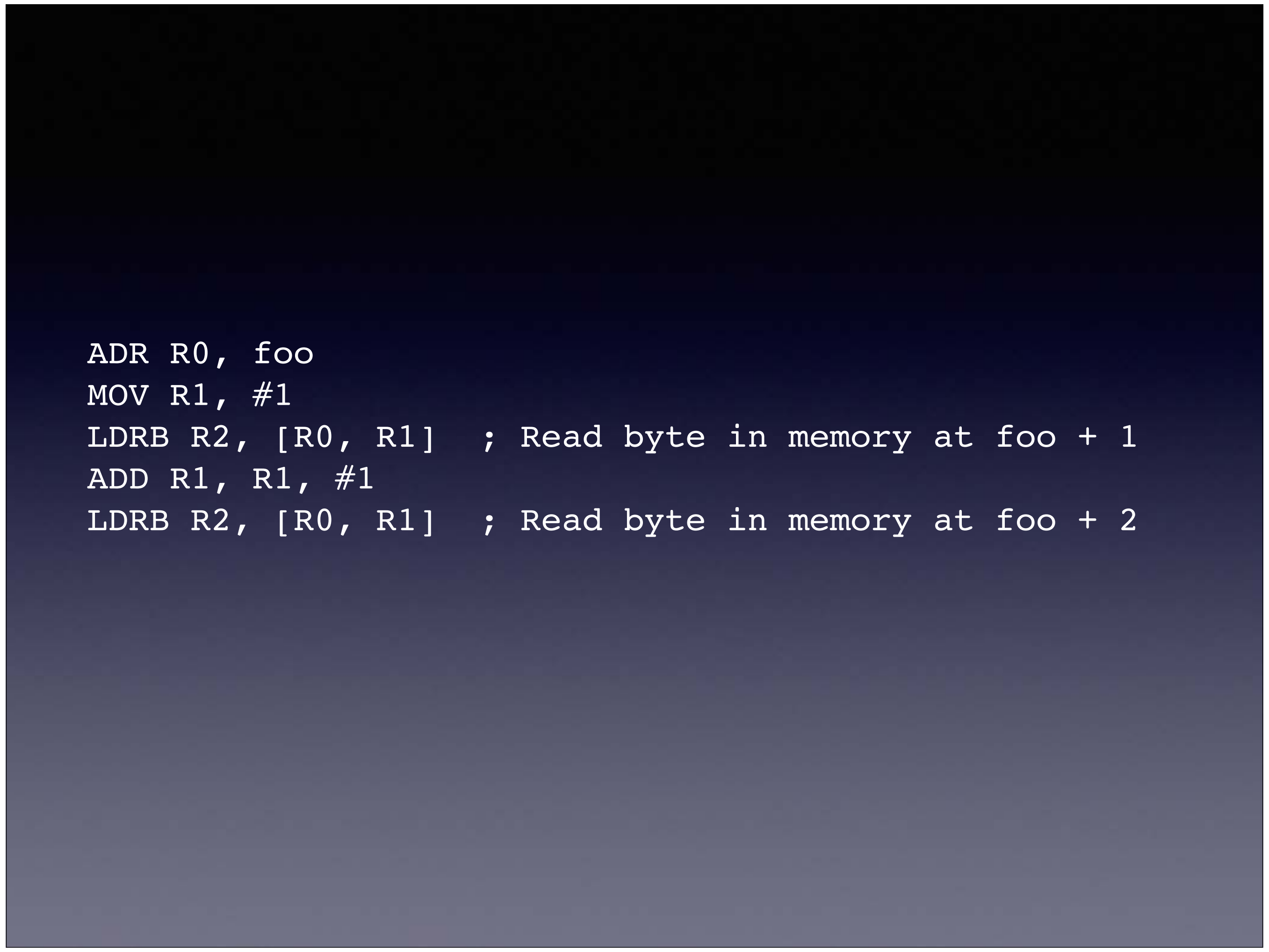
Simulated Absolute

- The assembler can use this indirect with offset addressing mode to simulate absolute addressing
- Just as `ADR` uses the `PC` as a starting point
- Generated `LDR` uses `PC` as the base to offset from
- Provided the value is within `±4K` of the `PC`

Otherwise instruction cannot be generated

Indirect with Register Index

- The index used does not have to be an immediate value
- Can also be another register…
- This gives us array like access to memory
- One register holds the base of the array, the other holds the index of the value we want to access



```
ADR R0, foo
MOV R1, #1
LDRB R2, [R0, R1]  ; Read byte in memory at foo + 1
ADD R1, R1, #1
LDRB R2, [R0, R1]  ; Read byte in memory at foo + 2
```

Again, good for accessing arrays…

Show how we can rewrite the strlen routine using indirect with register index addressing