Control Flow and Loops

Steven R. Bagley

Introduction

- Started to look at writing ARM Assembly Language
- Saw the structure of various commands
- Load (`LDR`), Store (`STR`) for accessing memory
- `SWI`s for OS access
- Data Processing Instructions
- Assembler Directives

Writing Assembly Programs

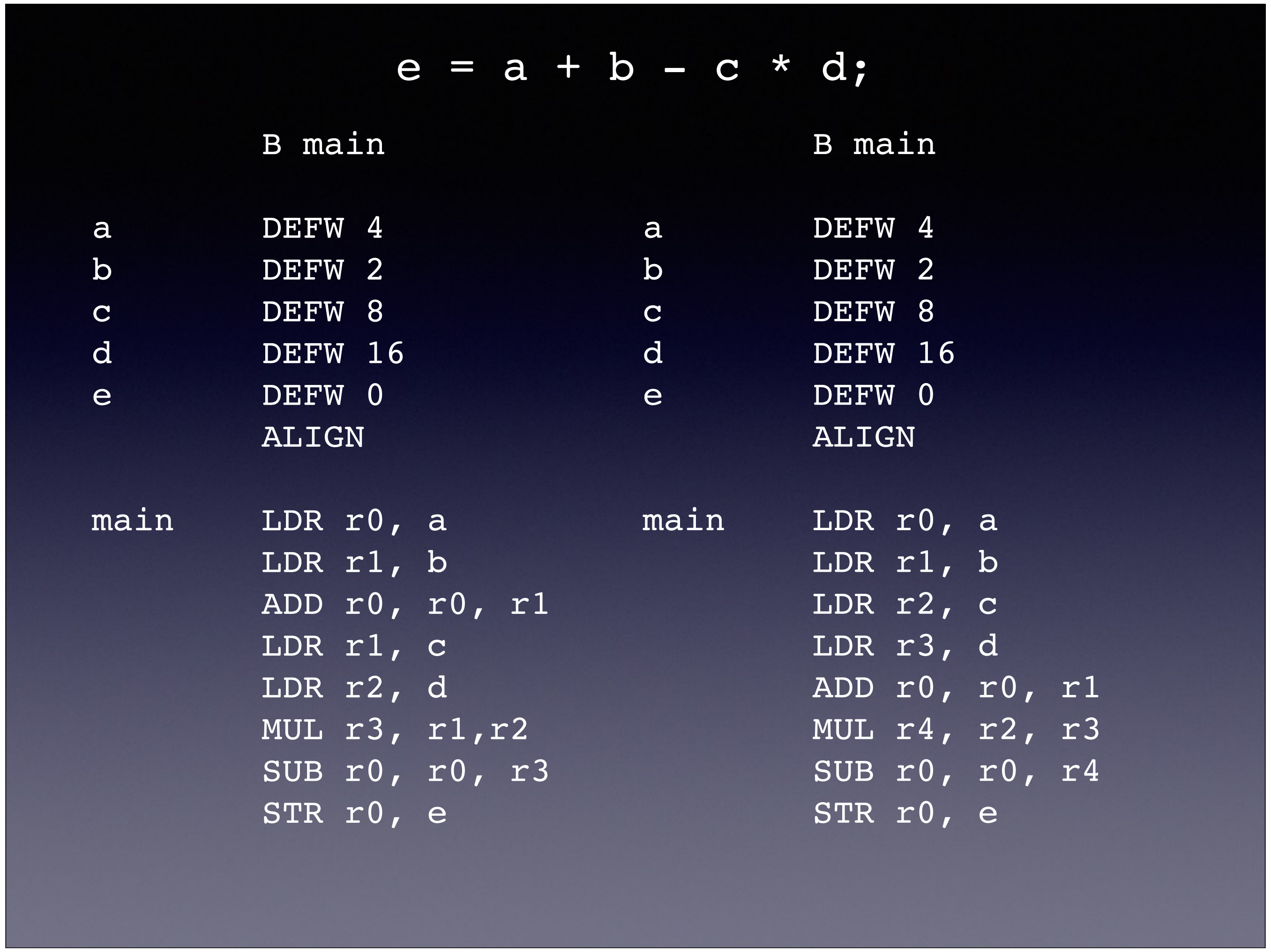
- One way to start writing assembly programs is to start by thinking about the C version
- And then converting it to assembler in stages
- First remove all the high-level features that assembly doesn’t have
- Then start assign variables into registers
- But you may also need to use memory to store some variables too (can use `DEFW` to create space for this)

e = a + b - c * d;

```
          B main

a         DEFW 4
b         DEFW 2
c         DEFW 8
d         DEFW 16
e         DEFW 0
          ALIGN

main      LDR r0, a
          LDR r1, b
          ADD r0, r0, r1
          LDR r1, c
          LDR r2, d
          MUL r3, r1,r2
          SUB r0, r0, r3
          STR r0, e
```

One on the right uses more registers…

But if the next thing also uses the value of b, say, it might be the better choice

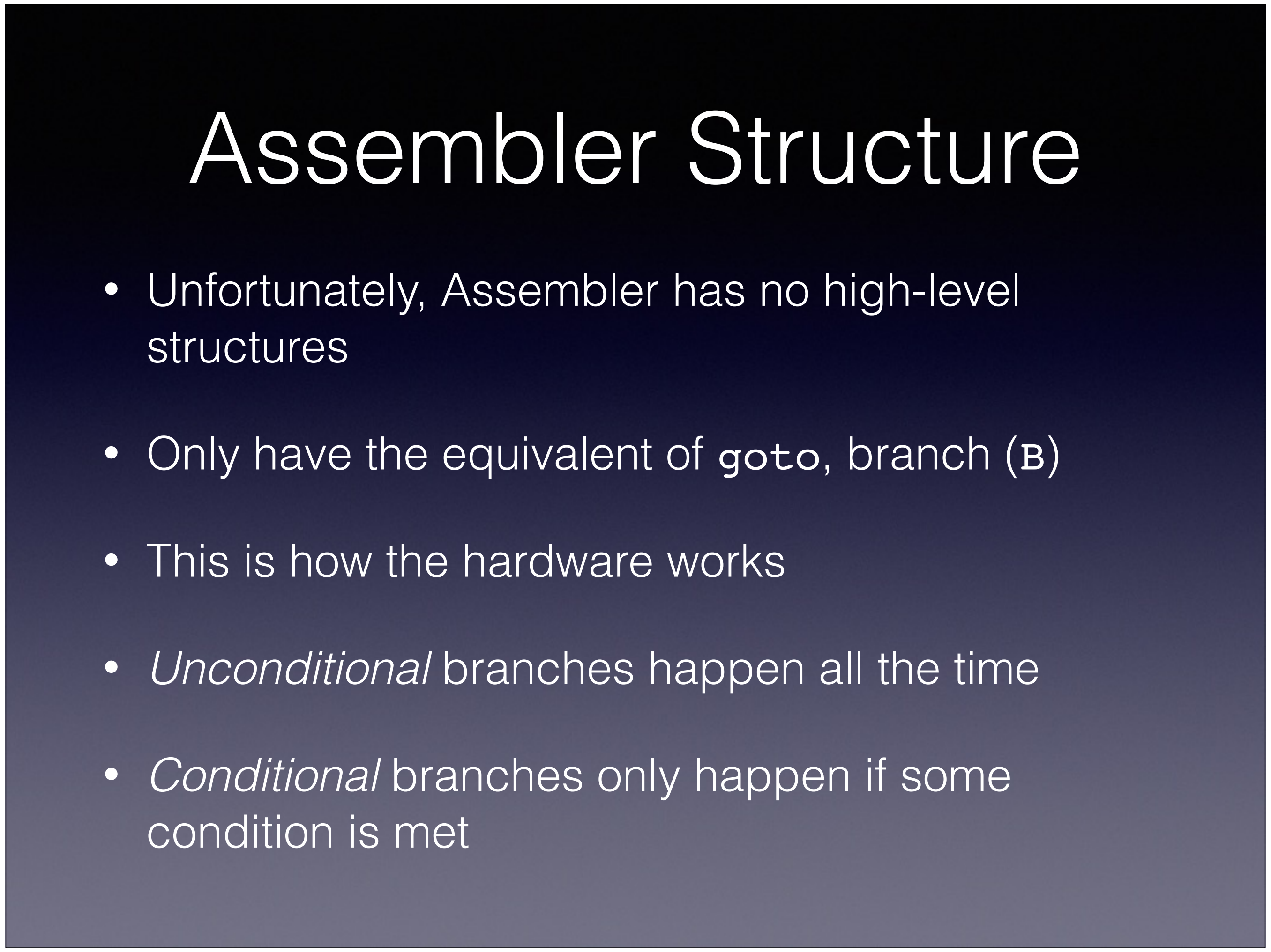


e = a + b – c * d;

```
        B main                        B main

a       DEFW 4                a       DEFW 4
b       DEFW 2                b       DEFW 2
c       DEFW 8                c       DEFW 8
d       DEFW 16               d       DEFW 16
e       DEFW 0                e       DEFW 0
        ALIGN                         ALIGN

main    LDR r0, a             main    LDR r0, a
        LDR r1, b                     LDR r1, b
        ADD r0, r0, r1                LDR r2, c
        LDR r1, c                     LDR r3, d
        LDR r2, d                     ADD r0, r0, r1
        MUL r3, r1,r2                 MUL r4, r2, r3
        SUB r0, r0, r3                SUB r0, r0, r4
        STR r0, e                     STR r0, e
```

One on the right uses more registers…

But if the next thing also uses the value of b, say, it might be the better choice



Assembler Structure

- Unfortunately, Assembler has no high-level structures
- Only have the equivalent of `goto`, branch (`B`)
- This is how the hardware works
- *Unconditional* branches happen all the time
- *Conditional* branches only happen if some condition is met

Saw the different types last lecture…

`ifs` in Assembler

- Do a `CMP` to test for the condition
- Branch if the condition is *not* met to skip over the code
- Code is then executed only if the condition is met…
- Otherwise we skip over it…

`if` Examples

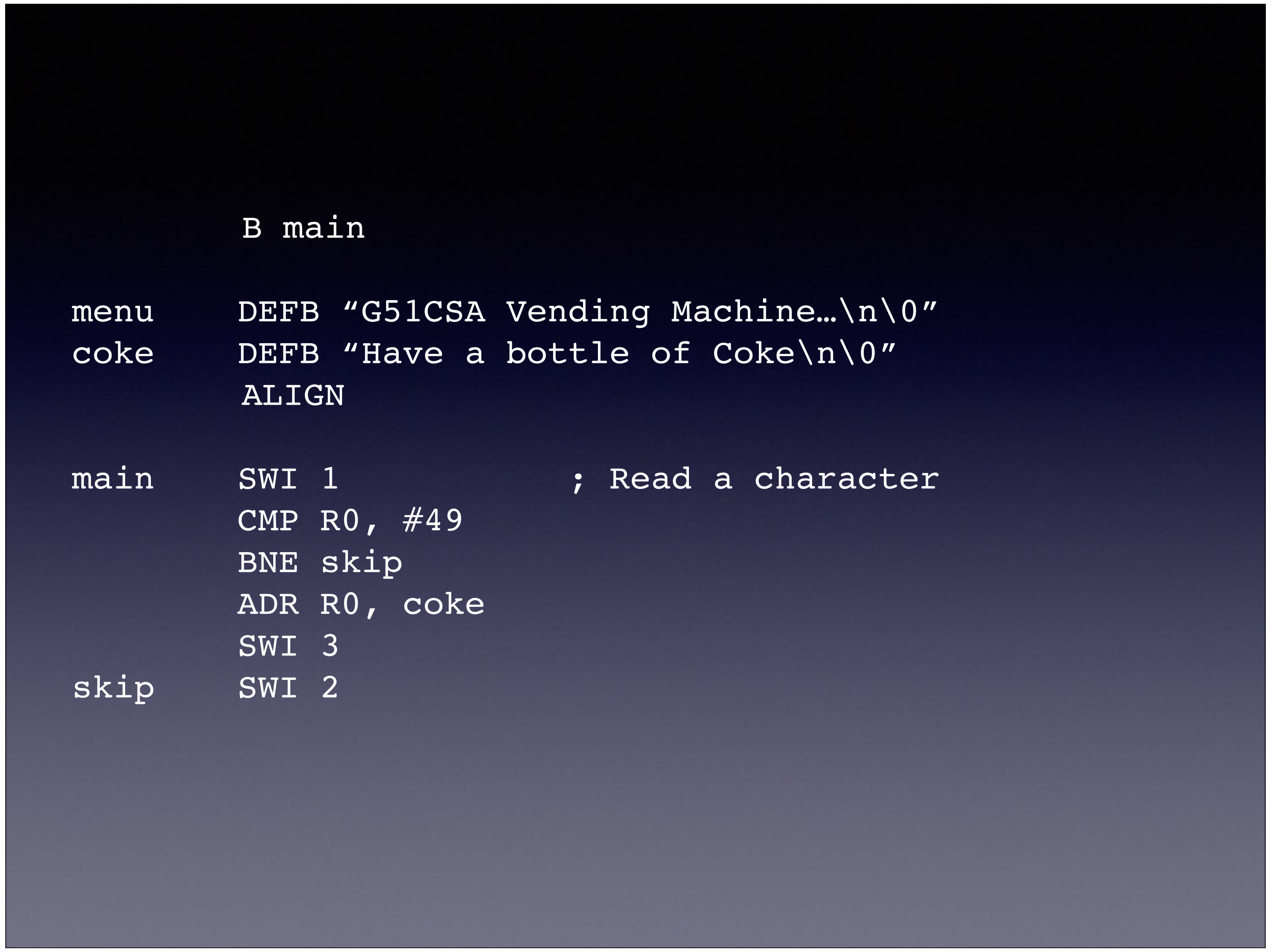
C	Assembler

`if(R0 == 10)`
`{`
`…`
`}`
	`CMP R0, #10`
	`BNE skip`
	`…`
	`skip`

`if(R0 < R1)`
`{`
`…`
`}`
	`CMP R0, R1`
	`BGE skip`
	`…`
	`skip`

`if(R0 >= 10)`
`{`
`…`
`}`
	`CMP R0, #10`
	`BLT skip`
	`…`
	`skip`

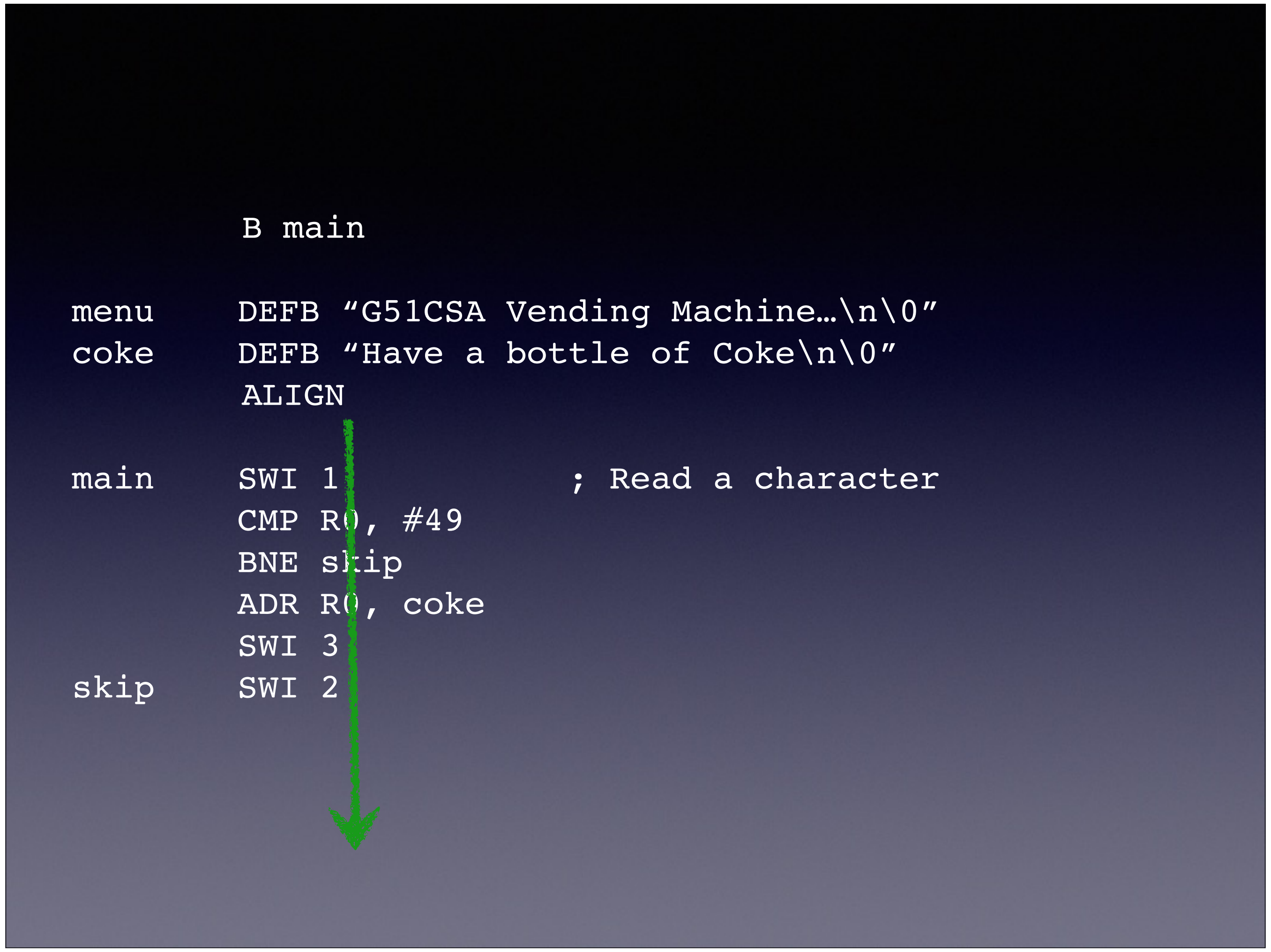
`if(R0 != 0)`
`{`
`…`
`}`
	`CMP R0, #10`
	`BEQ skip`
	`…`
	`skip`



```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```

Beginning of the Coke vending machine from G51PRG

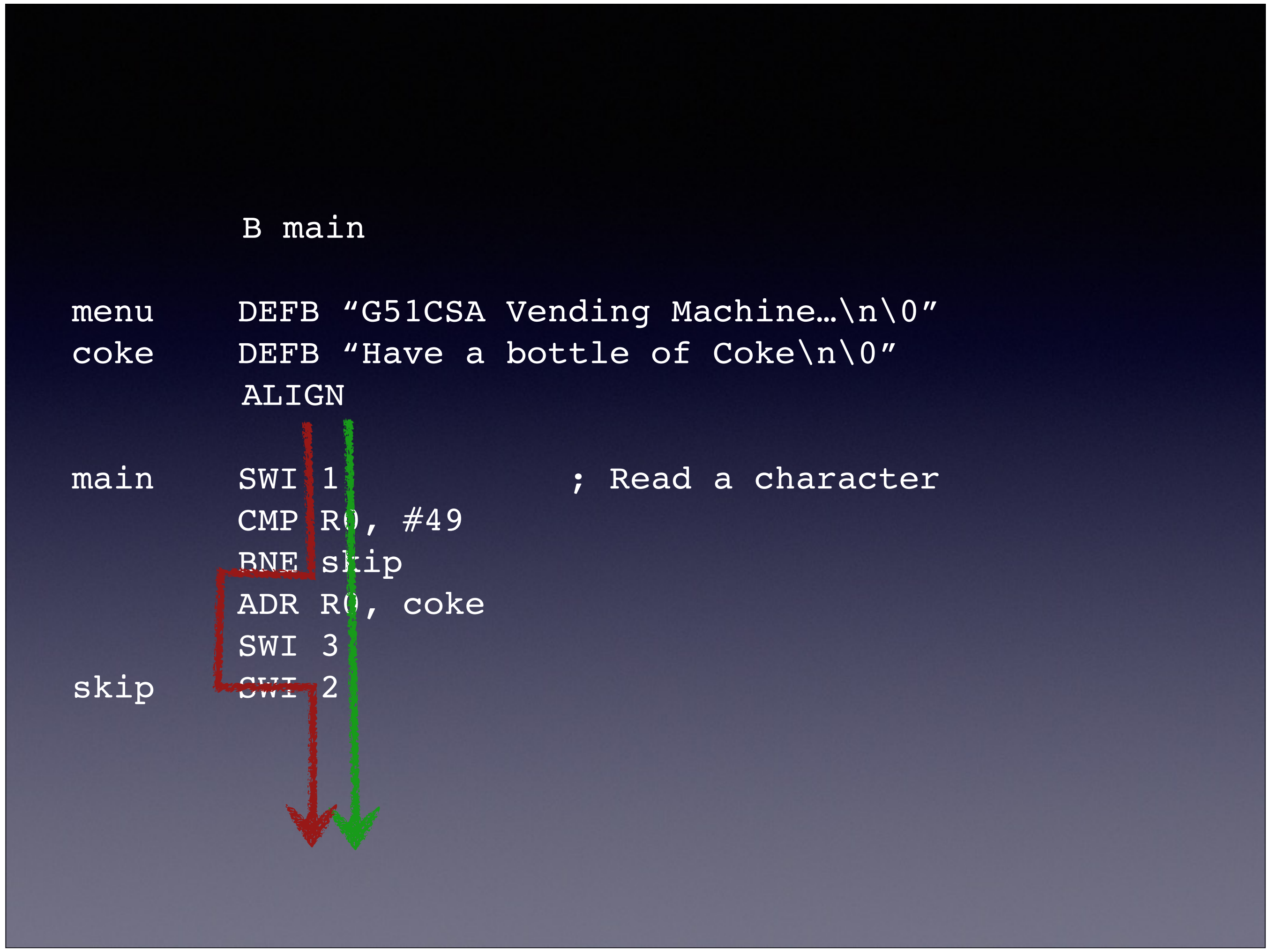
label name for skip unimportant — but it needs to be unique



```
        B main

menu    DEFB "G51CSA Vending Machine…\n\0"
coke    DEFB "Have a bottle of Coke\n\0"
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```

Beginning of the Coke vending machine from G51PRG

label name for skip unimportant — but it needs to be unique



```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
        ALIGN

main    SWI 1               ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    SWI 2
```

Beginning of the Coke vending machine from G51PRG

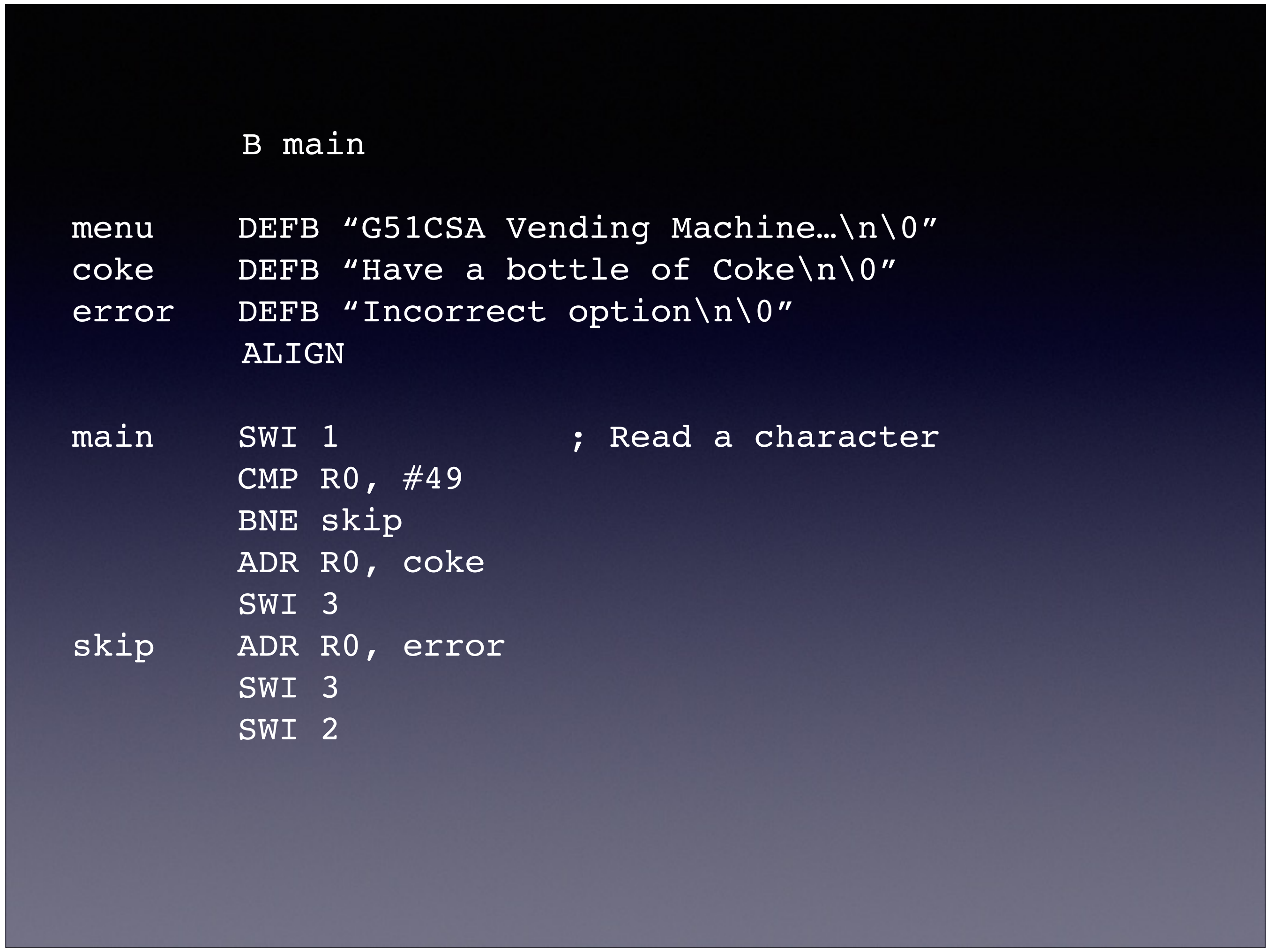
label name for skip unimportant — but it needs to be unique

Or else…

- How do we handle the `else` clause?
- Already doing it…
- If it's not true we are branching to `skip`
- Can put the `else` section there
- But the code that is executed when the condition is true will also get executed

Or else…

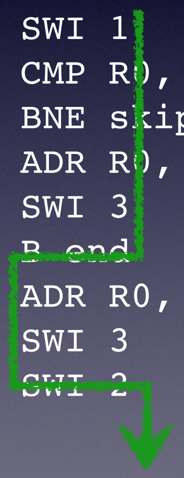
- But we can easily get around that…
- Put a branch at the end of the true block to skip the `else` section…
- This would be an unconditional branch…



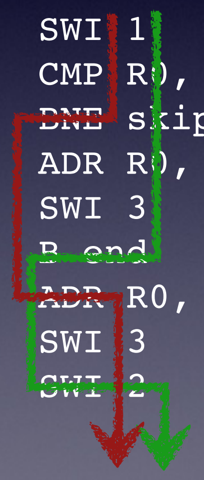
```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
error   DEFB “Incorrect option\n\0”
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
skip    ADR R0, error
        SWI 3
        SWI 2
```

This is problematic, because the error message will always be printed

```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
error   DEFB “Incorrect option\n\0”
        ALIGN

main    SWI 1           ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2
```



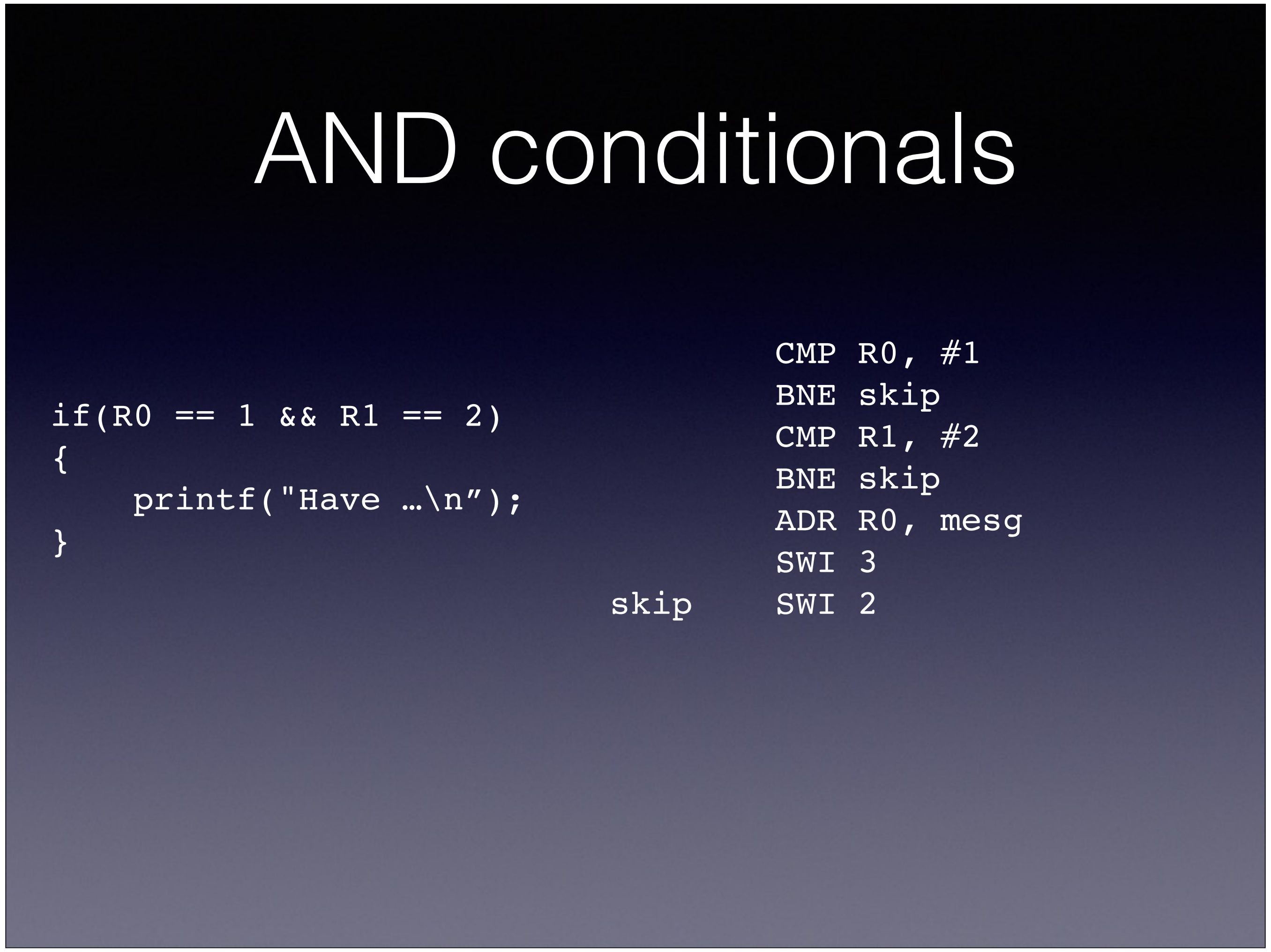
```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
error   DEFB “Incorrect option\n\0”
        ALIGN

main    SWI 1               ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2
```



```
        B main

menu    DEFB “G51CSA Vending Machine…\n\0”
coke    DEFB “Have a bottle of Coke\n\0”
error   DEFB “Incorrect option\n\0”
        ALIGN

main    SWI 1               ; Read a character
        CMP R0, #49
        BNE skip
        ADR R0, coke
        SWI 3
        B end
skip    ADR R0, error
        SWI 3
end     SWI 2
```

Combined Conditionals

- Should be possible to see how we can do combined conditionals too
- Just use multiple `CMP`/`Bxx` pairs
- Can also do lazy evaluation as with C…
- But need to watch our branch conditions…



AND conditionals

```
if(R0 == 1 && R1 == 2)
{
    printf("Have …\n”);
}
```

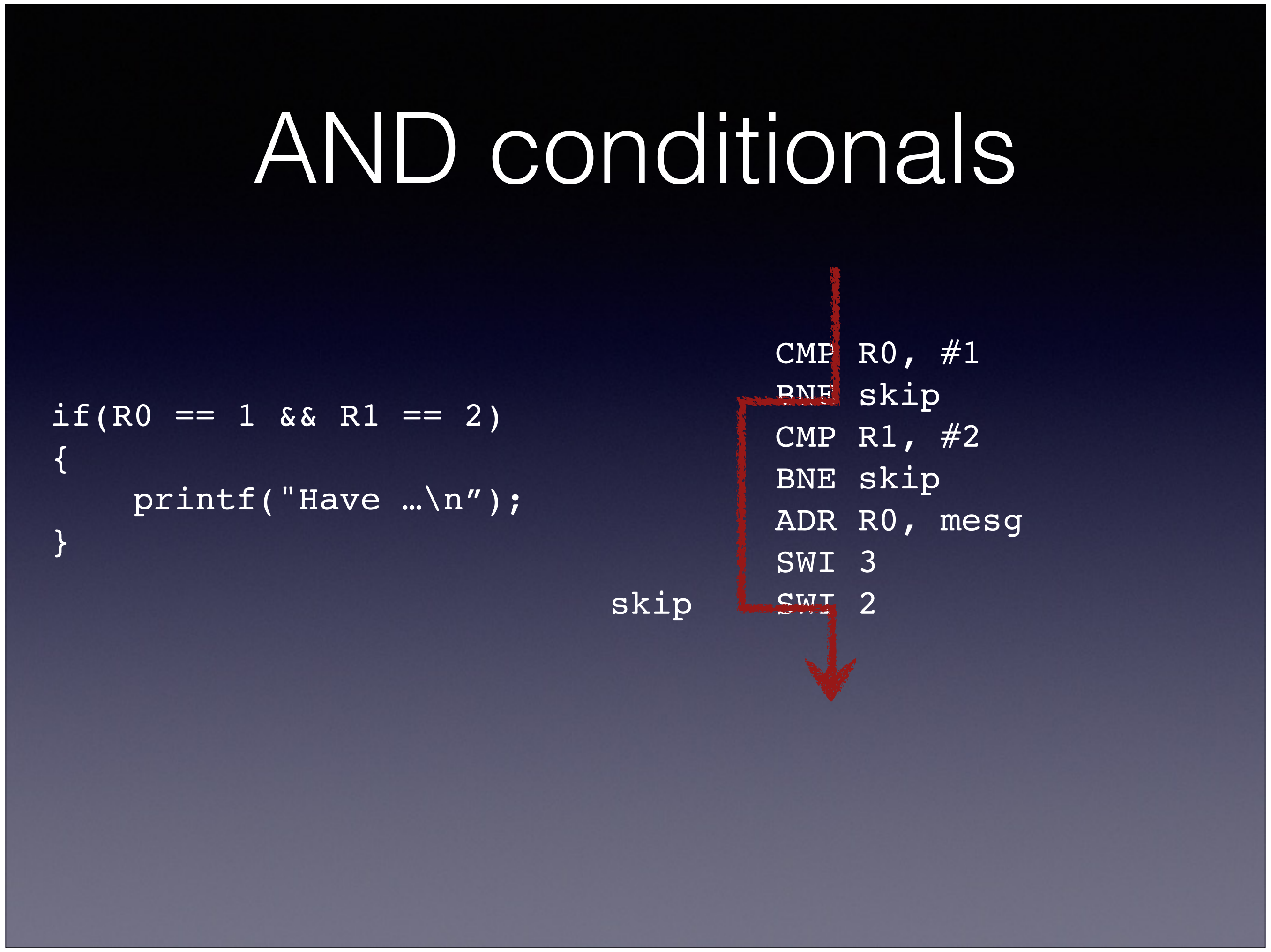
```
        CMP R0, #1
        BNE skip
        CMP R1, #2
        BNE skip
        ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 and r1 == 2

Note how both branches use the same condition code

If it’s ever false skip it



AND conditionals

```
if(R0 == 1 && R1 == 2)
{
    printf("Have …\n”);
}
```

```
        CMP R0, #1
        BNE skip
        CMP R1, #2
        BNE skip
        ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

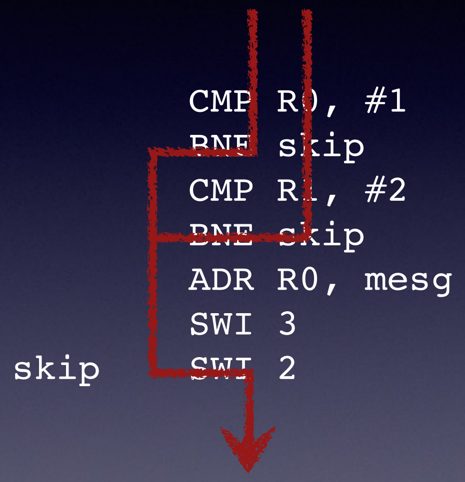
Highlighted code is only executed when r0 == 1 and r1 == 2

Note how both branches use the same condition code

If it’s ever false skip it

AND conditionals

```
if(R0 == 1 && R1 == 2)
{
    printf("Have …\n”);
}
```



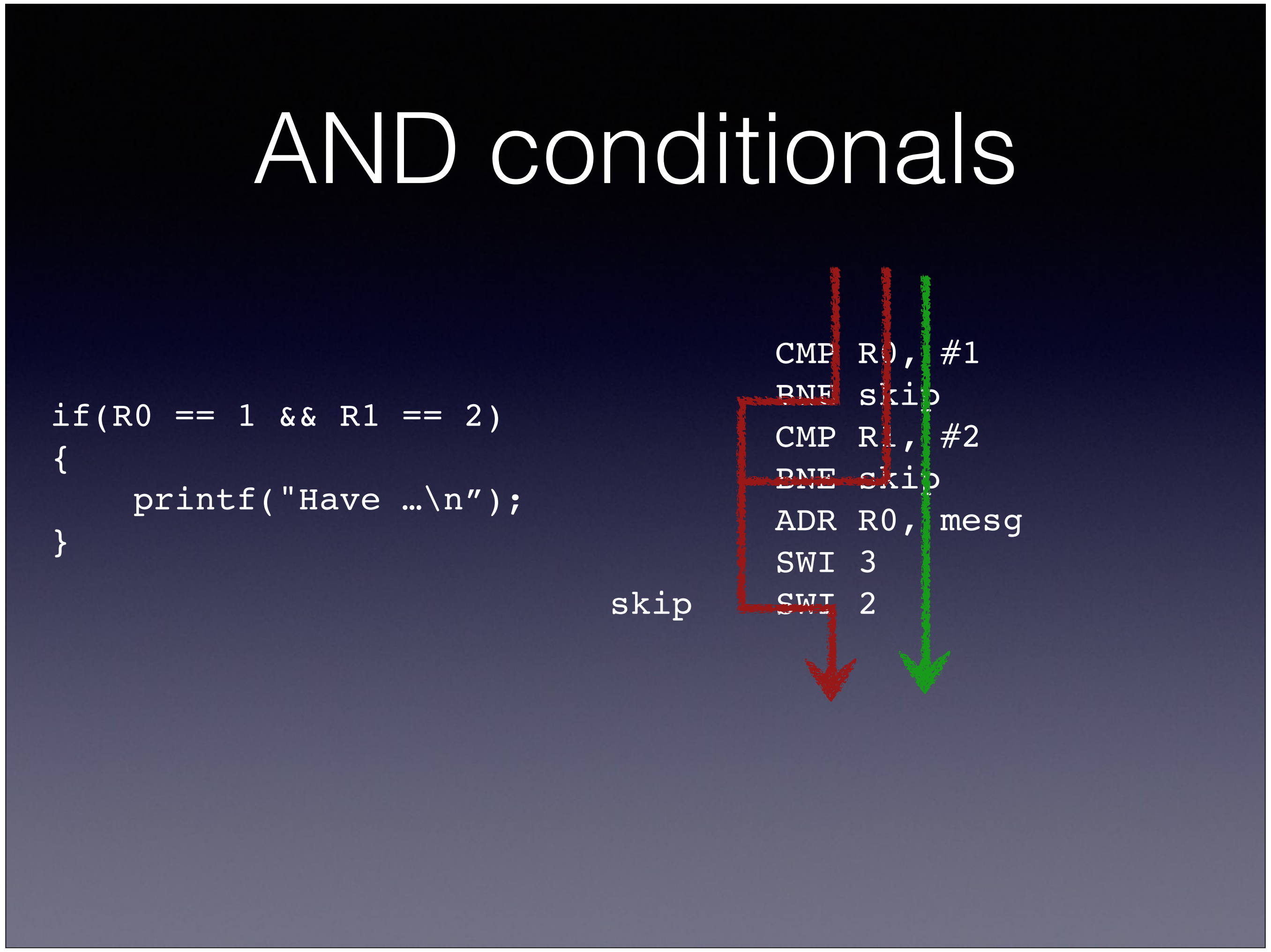
```
        CMP R0, #1
        BNE skip
        CMP R1, #2
        BNE skip
        ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 and r1 == 2

Note how both branches use the same condition code

If it’s ever false skip it



AND conditionals

```
if(R0 == 1 && R1 == 2)
{
    printf("Have …\n”);
}
```

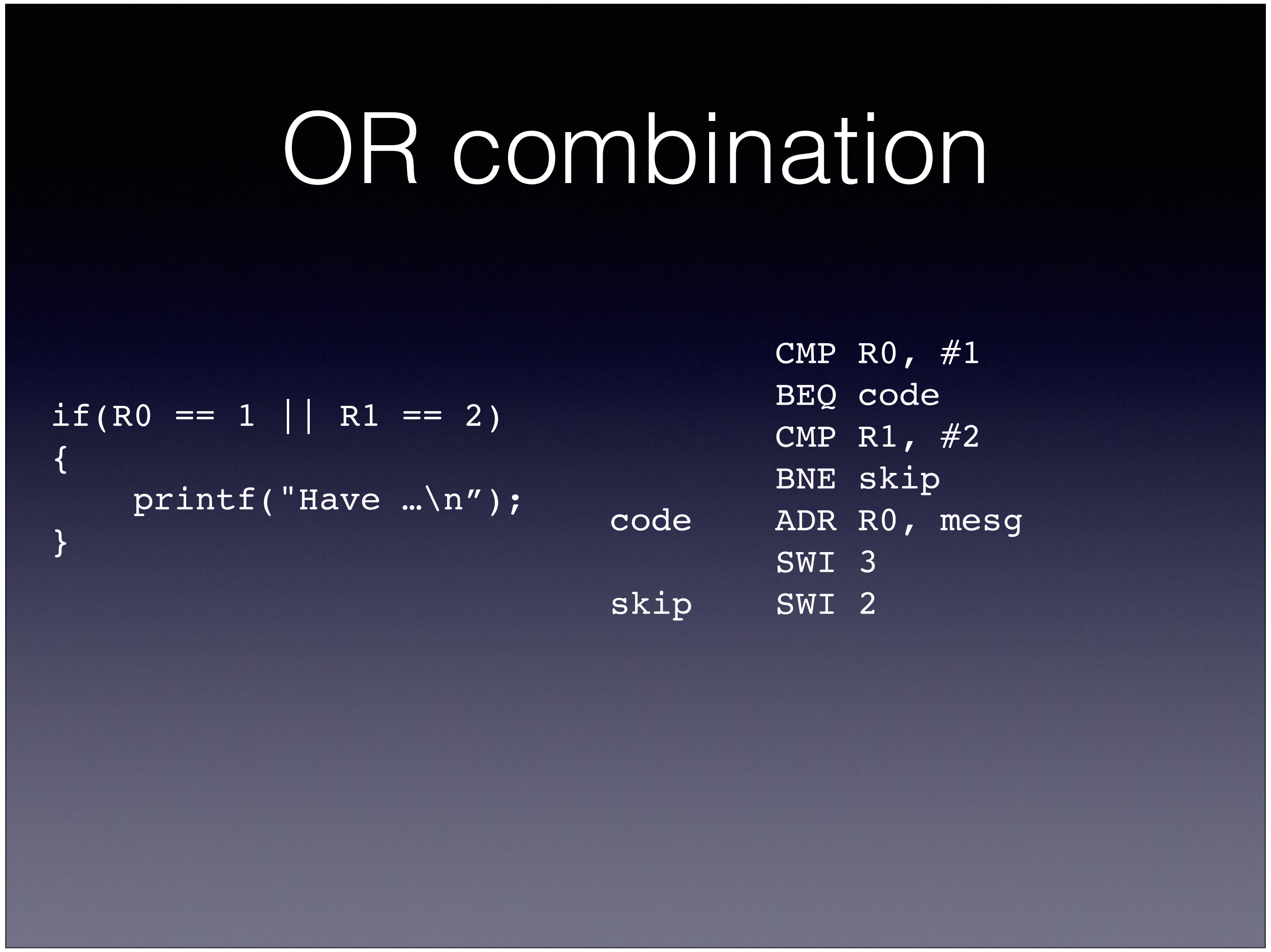
```
        CMP R0, #1
        BNE skip
        CMP R1, #2
        BNE skip
        ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 and r1 == 2

Note how both branches use the same condition code

If it’s ever false skip it



OR combination

```
if(R0 == 1 || R1 == 2)
{
    printf("Have …\n”);
}
```

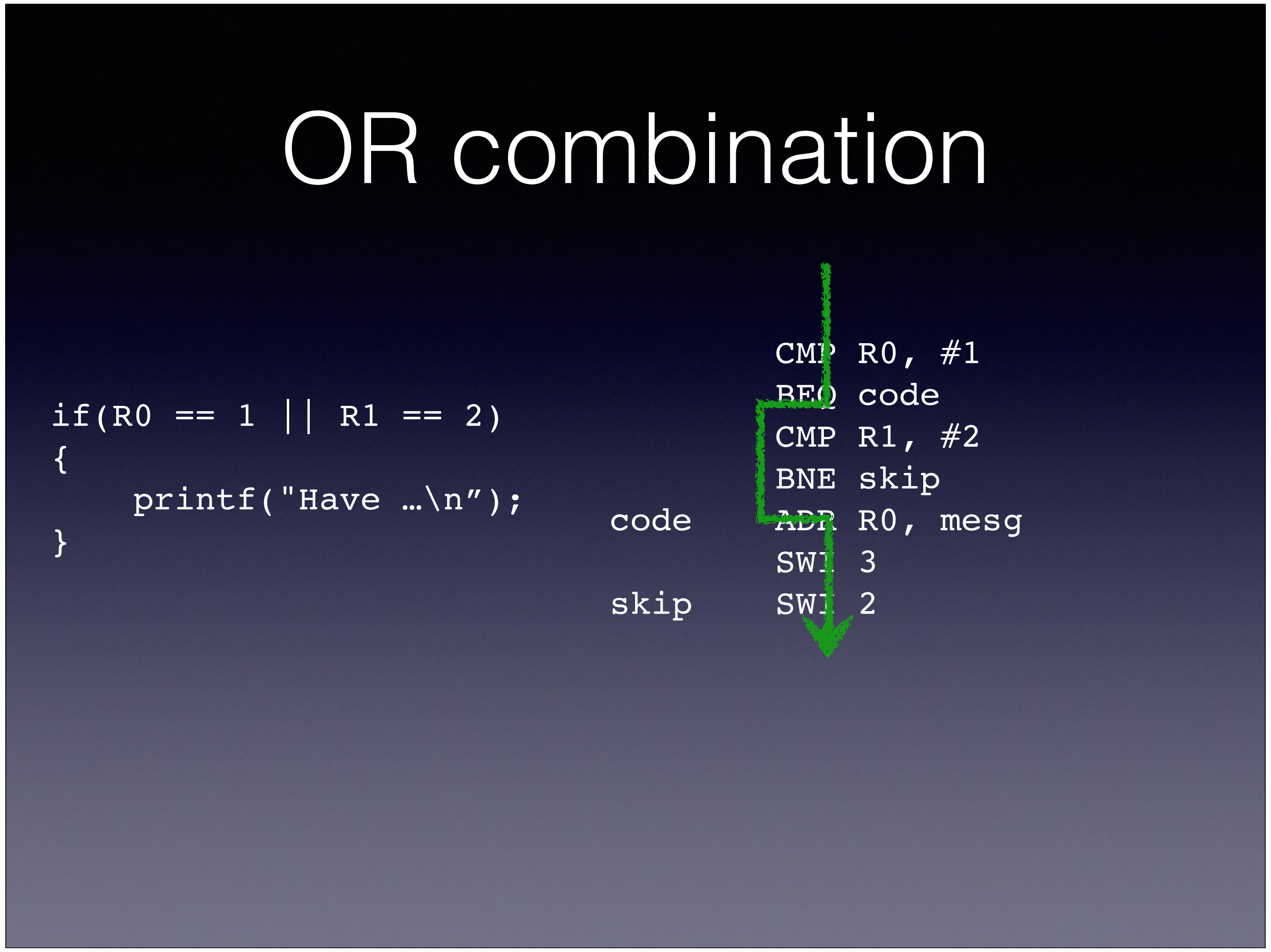
```
        CMP R0, #1
        BEQ code
        CMP R1, #2
        BNE skip
code    ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 or r1 == 2

Note how the first branch branches if equal, the second if not equal

Again it’s lazy evaluation — if the first test is true, then we don’t need to do the second test so can just jump into the code



OR combination

```
if(R0 == 1 || R1 == 2)
{
    printf("Have …\n”);
}
```

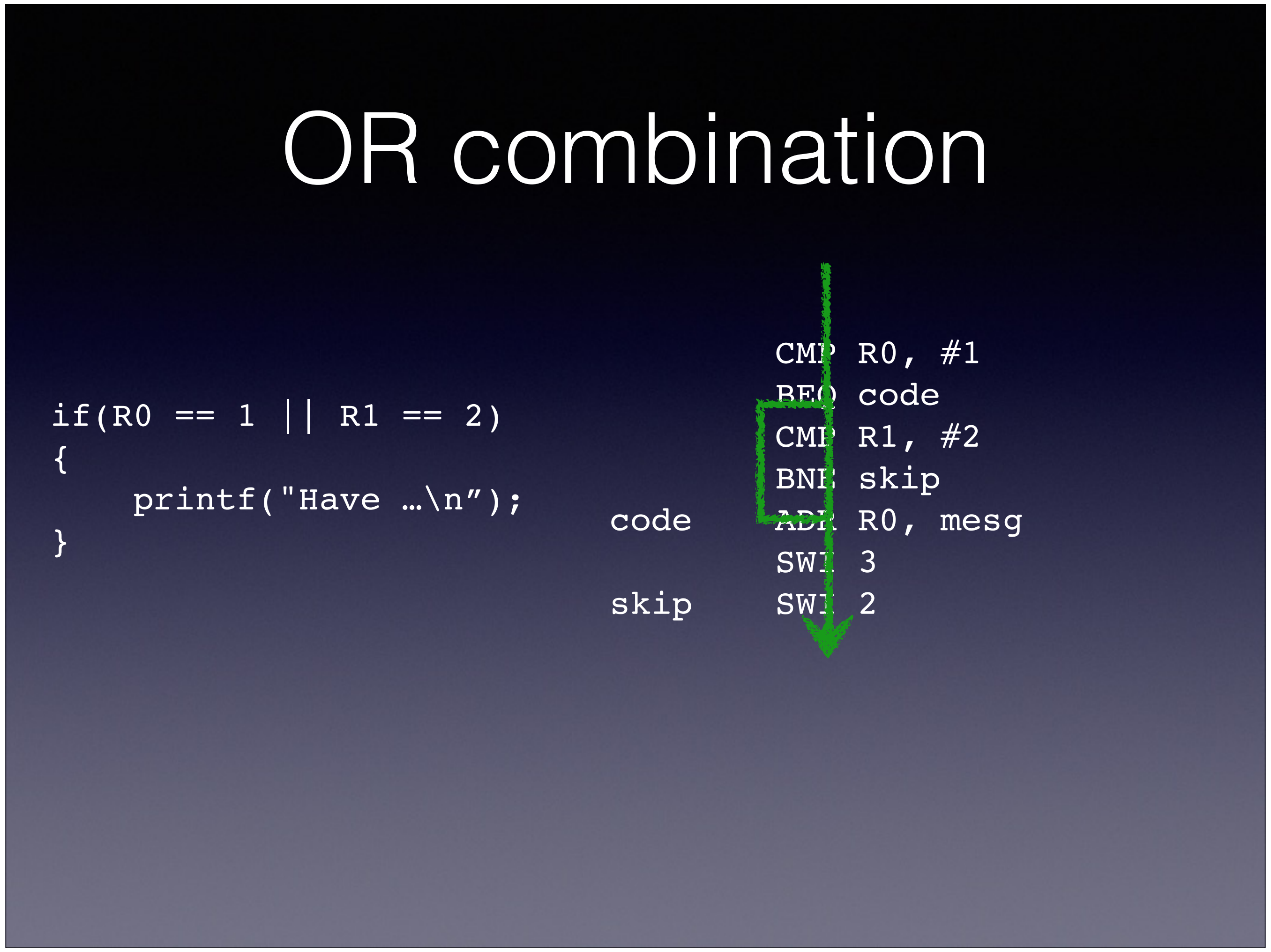
```
        CMP R0, #1
        BEQ code
        CMP R1, #2
        BNE skip
code    ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 or r1 == 2

Note how the first branch branches if equal, the second if not equal

Again it’s lazy evaluation — if the first test is true, then we don’t need to do the second test so can just jump into the code



OR combination

```
if(R0 == 1 || R1 == 2)
{
    printf("Have …\n”);
}
```

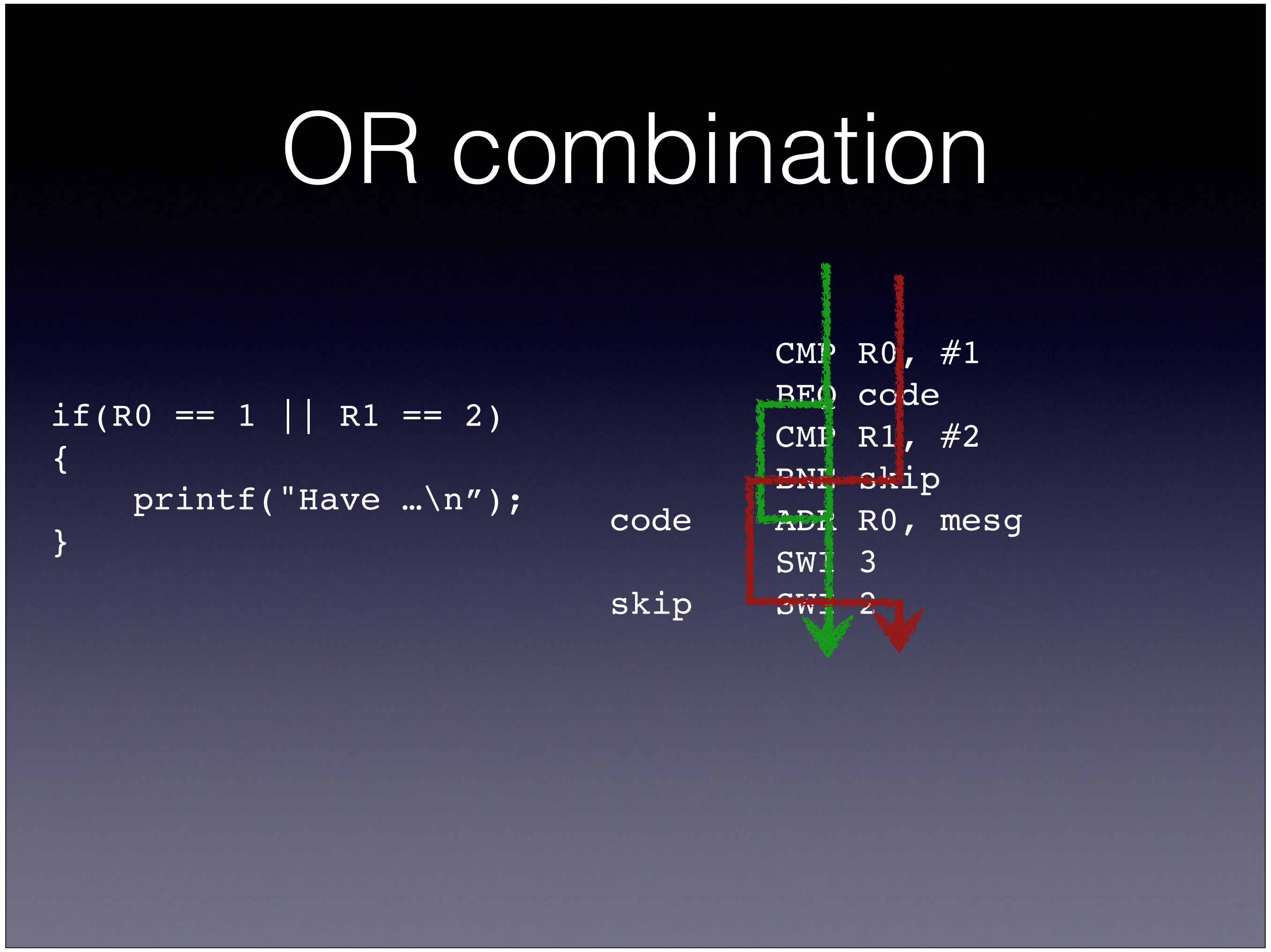
```
        CMP R0, #1
        BEQ code
        CMP R1, #2
        BNE skip
code    ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 or r1 == 2

Note how the first branch branches if equal, the second if not equal

Again it’s lazy evaluation — if the first test is true, then we don’t need to do the second test so can just jump into the code



OR combination

```
if(R0 == 1 || R1 == 2)
{
    printf("Have …\n”);
}
```

```
        CMP R0, #1
        BEQ code
        CMP R1, #2
        BNE skip
code    ADR R0, mesg
        SWI 3
skip    SWI 2
```

Could optimise this a lot though

Highlighted code is only executed when r0 == 1 or r1 == 2

Note how the first branch branches if equal, the second if not equal

Again it’s lazy evaluation — if the first test is true, then we don’t need to do the second test so can just jump into the code

From C to Assembler

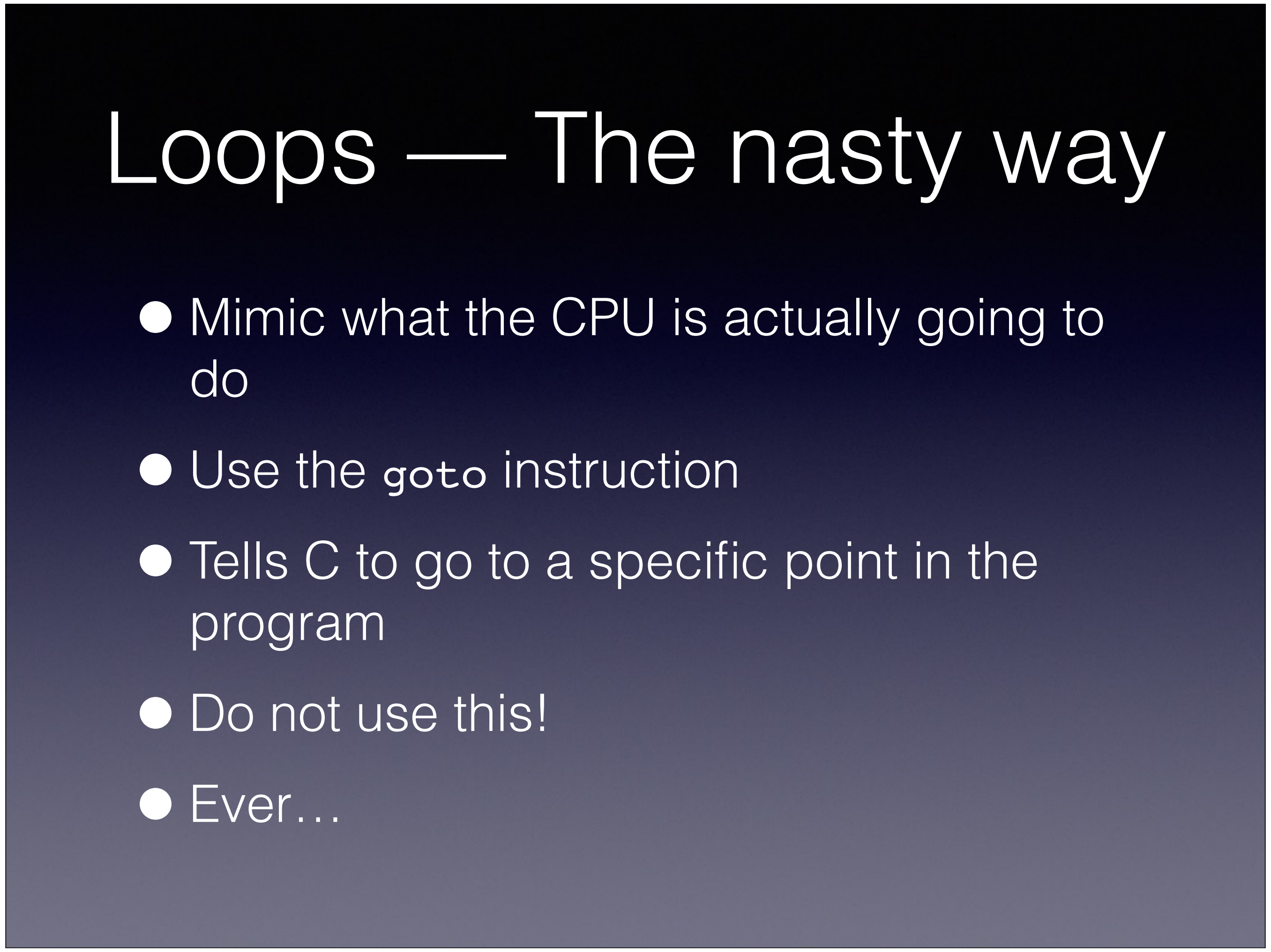
- Plan your assembler programs in C first
- Much easier to get the logic right in C using `ifs`, `whiles` etc.
- Can then convert those high-level structures into machine code
- Already seen how to convert `if` statements…

At least to start with…

Difference between the logic behind a program and the syntax of the language

Loops

- Important building blocks in programs
- `while` loops repeat a block of code whilst the condition holds
- But again, like `if`, ARM assembler does not provide a `while` loop
- Need to manufacture it out of simple components
- Already saw this early on in PRG…



Loops — The nasty way

- Mimic what the CPU is actually going to do
- Use the `goto` instruction
- Tells C to go to a specific point in the program
- Do not use this!
- Ever…

goto Demo…

Loops using goto

- We can translate our `while` loop to using `goto`s
- Bad programming practice normally
- But a good halfway stage between structured programming and assembler
- Remember though that a `while` loop can execute zero or more times…

Run through an example in C

Show that they both run and produce the same result.

`while` loop

```
while(i < 8)
{
    j = j + 3;
    i = i + 1;
}
```

Note how we've turned the structure of the while loop upside down

We've put the test at the end… There's reasons for this madness…

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0

`while` loop

```
while(i < 8)
{
    j = j + 3;
    i = i + 1;
}
```

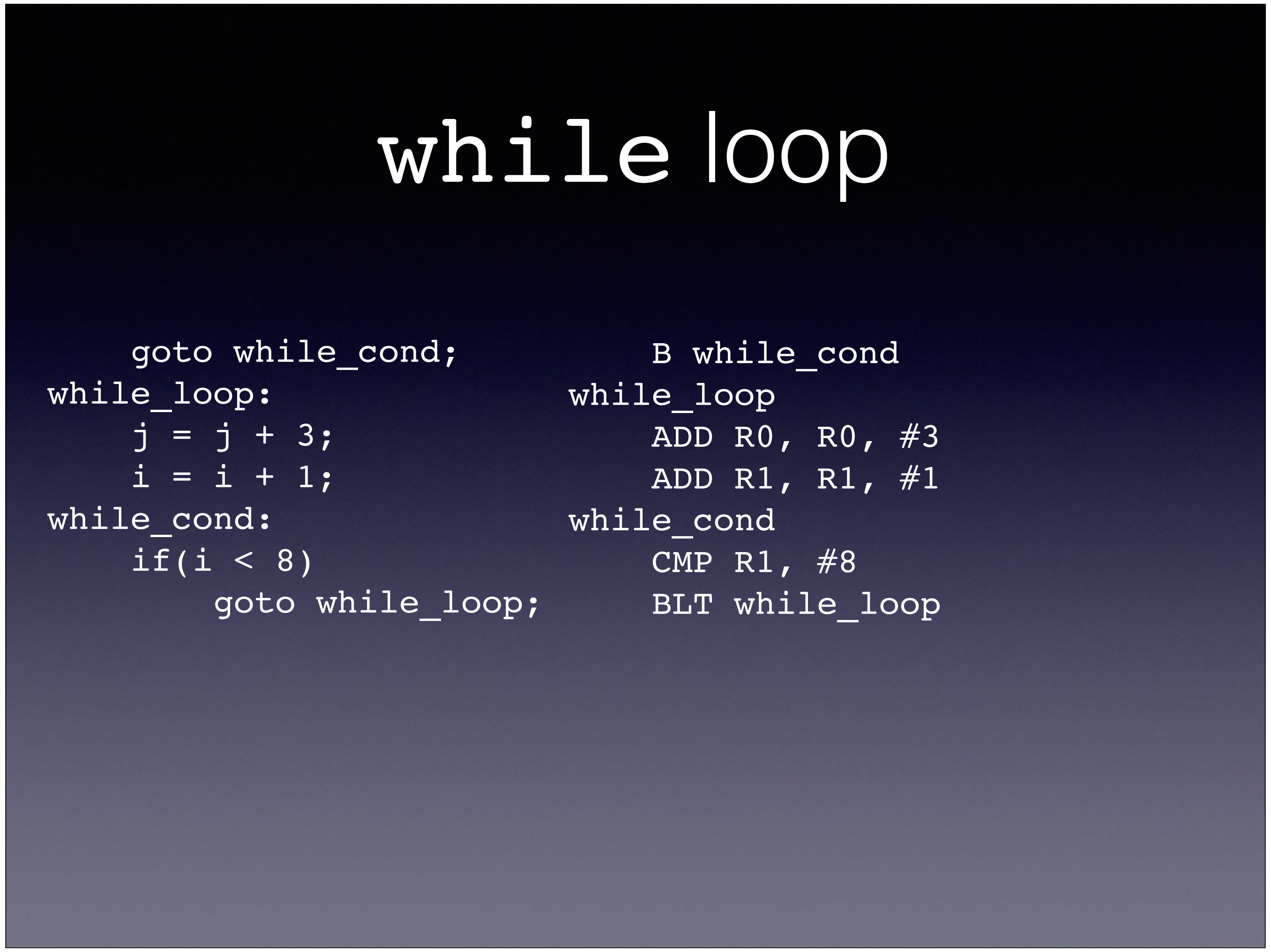
```
    goto while_cond;
while_loop:
    j = j + 3;
    i = i + 1;
while_cond:
    if(i < 8)
        goto while_loop;
```

Note how we’ve turned the structure of the while loop upside down

We’ve put the test at the end… There’s reasons for this madness…

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0



`while` loop

```
    goto while_cond;
while_loop:
    j = j + 3;
    i = i + 1;
while_cond:
    if(i < 8)
        goto while_loop;
```

```
    B while_cond
while_loop
    ADD R0, R0, #3
    ADD R1, R1, #1
while_cond
    CMP R1, #8
    BLT while_loop
```

Note how we’ve turned the structure of the while loop upside down

We’ve put the test at the end… There’s reasons for this madness…

Can then convert this very easily into machine code

ASM version assumes i is in R1 and j is in R0

while loop

- Putting the conditional at the end has several advantages
- Don’t have to invert the conditional (as we did with `if`)
- Will actually execute less instructions…
- Code for a `do…while()` loop is near identical

`while` loop

```
    B while_cond
while_loop
    ADD R0, R0, #3
    ADD R1, R1, #1
while_cond
    CMP R1, #8
    BLT while_loop
```

```
while_cond
    CMP R1, #8
    BGE end
    ADD R0, R0, #3
    ADD R1, R1, #1
    B while_cond
end:
```

ASM version assumes i is in R1 and j is in R0

Uses 4 instructions per loop iteration + 3 to start it off (35)

second method takes 5 instructions per +2 extra (42)

Take a look and see what the compiler does too…