SWINBURNE UNIVERSITY OF TECHNOLOGY

COS10004 Computer Systems

Lecture 10.4 – Functions in ARM Assembly - Program Counter and Link Register

CRICOS provider 00111D

Chris McCarthy

FUNCTIONS IN ASM

- Not 'native' to assembly
  - We need to do a lot of the management ourselves
- Argument passing:
  - How do we pass arguments from one function to another
- Storing and recalling register values
  - each function we call will want to use the same registers (only 13 general purpose registers !)
  - How do we manage this ?
- Managing the program control
  - Jumping from one function to another, and then returning back !

FUNCTIONS IN ASM

- Not 'native' to assembly
  - We need to do a lot of the management ourselves
- Argument passing:
  - How do we pass arguments from one function to another
- Storing and recalling register values
  - each function we call will want to use the same registers (only 13 general purpose registers !)
  - How do we manage this ?
- Managing the program control
  - Jumping from one function to another, and then returning back !

RECALL OUR "FLASHING LED" PROGRAM

```
1        mov r0, #.green
2        mov r1, #.white
3        mov r2, #1          ; 1sec delay time
4  flash:
5        str r0,.Pixel367    ; flash on
6        LDR r3,.Time        ; start time
7  timer1:
8        LDR r4, .Time       ; current time
9        sub r5,r4,r3        ; elapsed time = current time - start time
10       CMP r5,r2           ; compare elapsed to delay time
11       BLT timer1
12       str r1,.Pixel367    ; flash off
13       LDR r3,.Time        ; start time
14 timer2:
15       LDR r4,.Time        ; current time
16       sub r5,r4,r3        ; elapsed time = current time - start time
17       CMP r5,r2           ; compare elapsed to delay time
18       BLT timer2
19       B flash
20       halt
```

RECALL OUR "FLASHING LED" PROGRAM

```
1        mov r0, #.green
2        mov r1, #.white
3        mov r2, #1          ; 1sec delay time
4  flash:
5        str r0,.Pixel367    ; flash on
6        LDR r3,.Time        ; start time
7  timer1:
8        LDR r4, .Time       ; current time
9        sub r5,r4,r3        ; elapsed time = current time - start time
10       CMP r5,r2           ; compare elapsed to delay time
11       BLT timer1
12       str r1,.Pixel367    ; flash off
13       LDR r3,.Time        ; start time
14 timer2:
15       LDR r4,.Time        ; current time
16       sub r5,r4,r3        ; elapsed time = current time - start time
17       CMP r5,r2           ; compare elapsed to delay time
18       BLT timer2
19       B flash
20       halt
```

RECALL OUR "FLASHING LED" PROGRAM

```
1        mov r0, #.green
2        mov r1, #.white
3        mov r2, #1          ; 1sec delay time
4  flash:
5        str r0,.Pixel367    ; flash on
6        LDR r3,.Time        ; start time
7  timer1:
8        LDR r4, .Time       ; current time
9        sub r5,r4,r3        ; elapsed time = current time - start time
10       CMP r5,r2           ; compare elapsed to delay time
11       BLT timer1
12       str r1,.Pixel367    ; flash off
13       LDR r3,.Time        ; start time
14 timer2:
15       LDR r4,.Time        ; current time
16       sub r5,r4,r3        ; elapsed time = current time - start time
17       CMP r5,r2           ; compare elapsed to delay time
18       BLT timer2
19       B flash
20       halt
```

DEFINING FUNCTIONS

- We’re obviously duplicating code in this example, which in general we want to avoid
- Why not write a function to do this !

DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:

```
 1|       mov r0, #.green
 2|       mov r1, #.white
 3|       mov r2, #1          ; 1sec delay time
 4|flash:
 5|       str r0,.Pixel367   ; flash on
 6|       LDR r3,.Time       ; start time
 7|       BL delay           ; call delay function
 8|       str r1,.Pixel367   ; flash off
 9|       LDR r3,.Time       ; start time
10|       BL delay           ; call delay function
11|       B flash
12|       halt
```

DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:

```
 1|       mov r0, #.green
 2|       mov r1, #.white
 3|       mov r2, #1          ; 1sec delay time
 4|flash:
 5|       str r0,.Pixel367   ; flash on
 6|       LDR r3,.Time       ; start time
 7|       BL delay           ; call delay function
 8|       str r1,.Pixel367   ; flash off
 9|       LDR r3,.Time       ; start time
10|       BL delay           ; call delay function
11|       B flash
12|       halt
```

DEFINING FUNCTIONS

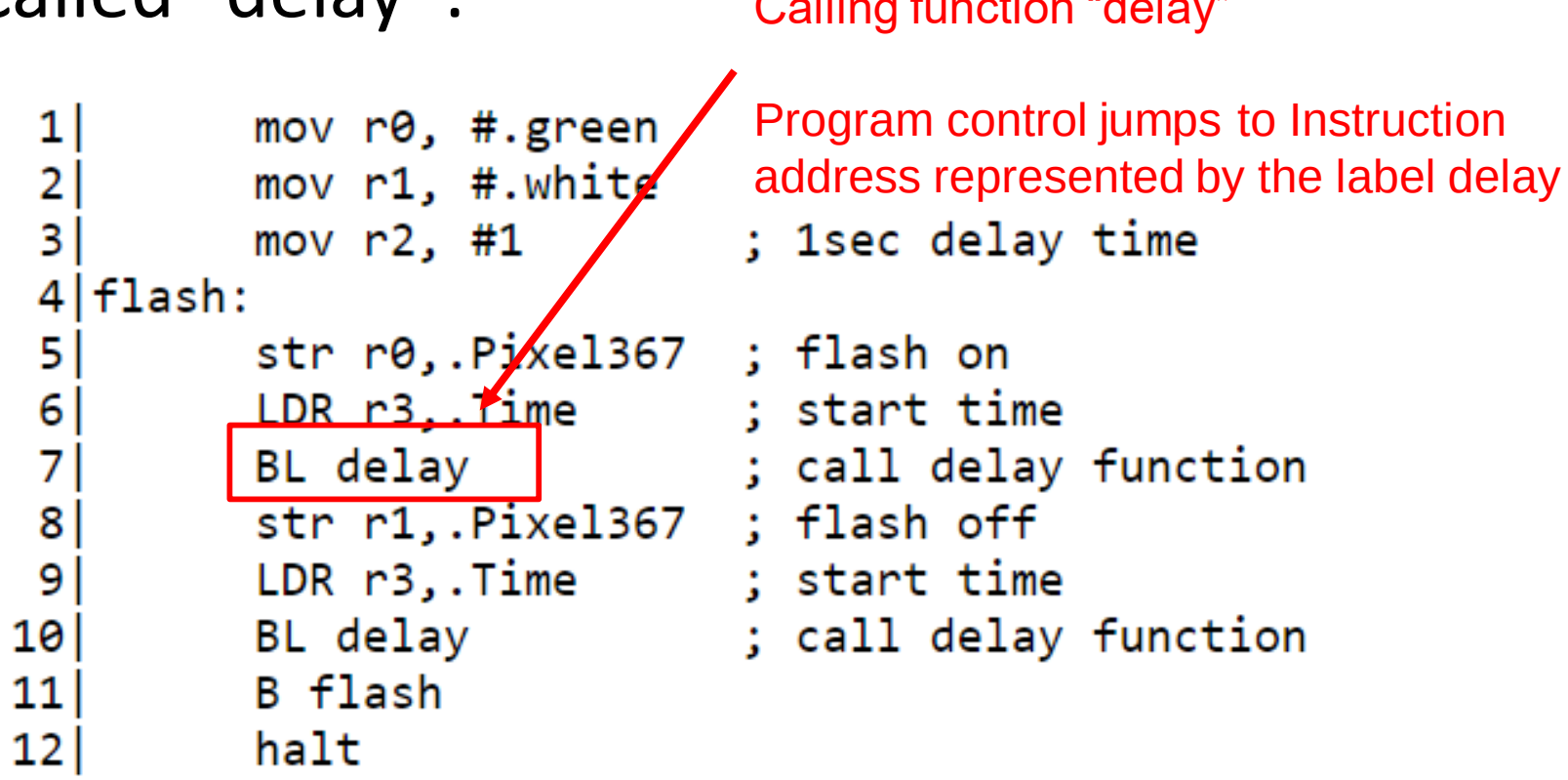
- Imagine we had already written a function called “delay”:

```
 1|       mov r0, #.green
 2|       mov r1, #.white
 3|       mov r2, #1          ; 1sec delay time
 4|flash:
 5|       str r0,.Pixel367   ; flash on
 6|       LDR r3,.Time       ; start time
 7|       BL delay           ; call delay function
 8|       str r1,.Pixel367   ; flash off
 9|       LDR r3,.Time       ; start time
10|       BL delay           ; call delay function
11|       B flash
12|       halt
```

Now we have replaced the duplicated code with a function call

DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:

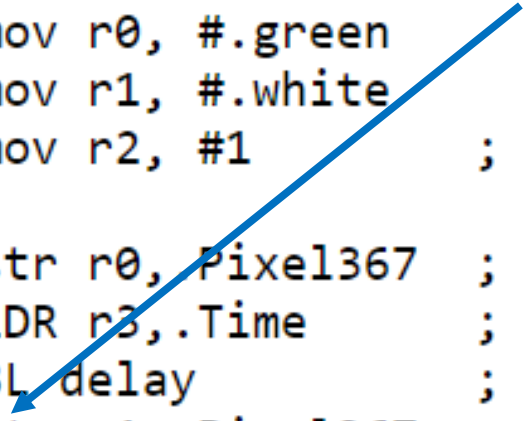


DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:

Once the function is complete, program control returns to instruction after function call.

How does it know how to get back ???



```
 1|       mov r0, #.green
 2|       mov r1, #.white
 3|       mov r2, #1          ; 1sec delay time
 4|flash:
 5|       str r0, Pixel367   ; flash on
 6|       LDR r3,.Time       ; start time
 7|       BL delay           ; call delay function
 8|       str r1,.Pixel367   ; flash off
 9|       LDR r3,.Time       ; start time
10|       BL delay           ; call delay function
11|       B flash
12|       halt
```

KEY REGISTERS

- Program counter (pc, also r15):
  - Holds the address of the next instruction to execute
- Link Register (lr, also r14):
  - Holds the address of instruction to return to after a function is complete

How are they used for function calls?

- Program counter (pc):
  - Is updated when a branch to label (**BL**) is encountered
- Link Register (lr):
  - holds what was in pc register before it was changed
  - i.e., address of the next instruction after the function call
  - brings us back to where we came from (we'd be lost otherwise!)

HELPFUL INSTRUCTION - BL

- **BL** `label$:`
  - causes program control to jump to label$,

  but also ….
  - copies next instruction to lr so we know how to get back!

DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:

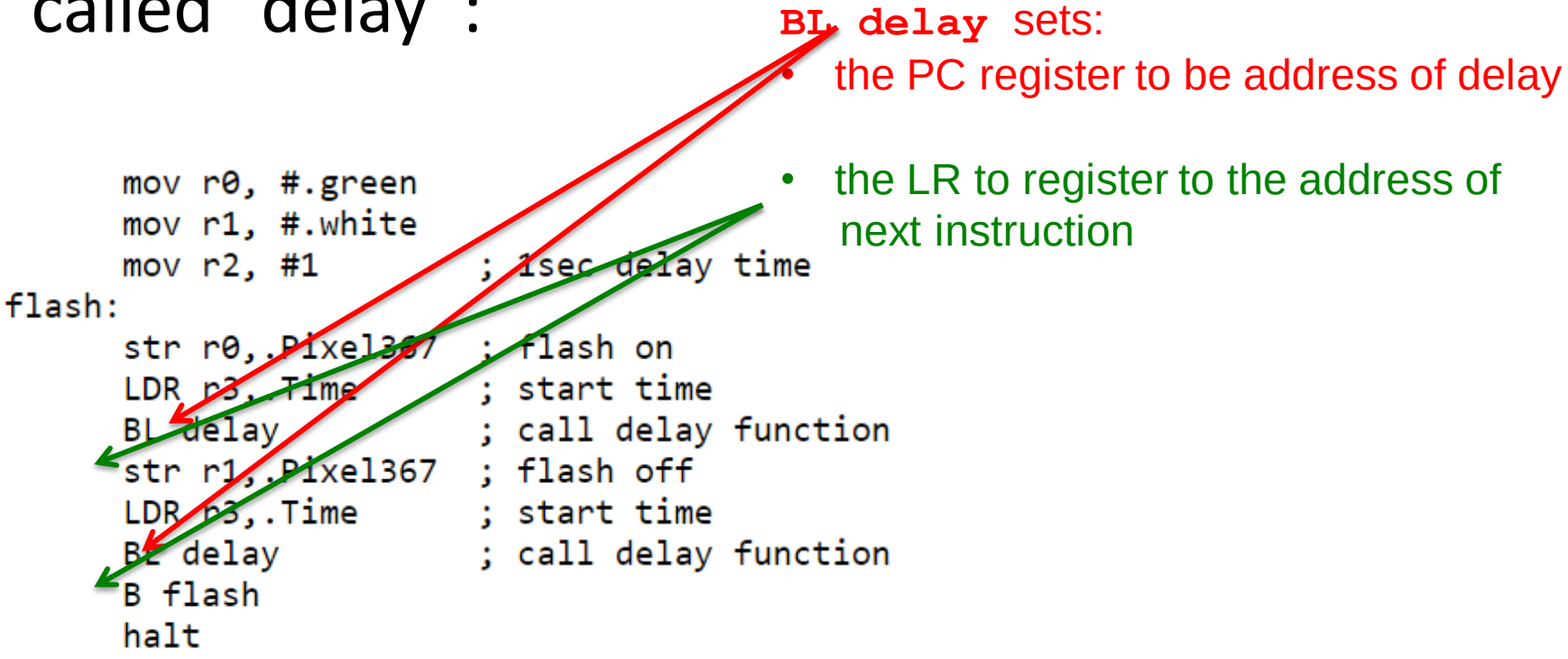
```
 1|      mov r0, #.green
 2|      mov r1, #.white
 3|      mov r2, #1          ; 1sec delay time
 4|flash:
 5|      str r0,.Pixel367   ; flash on
 6|      LDR r3,.Time       ; start time
 7|      BL delay           ; call delay function
 8|      str r1,.Pixel367   ; flash off
 9|      LDR r3,.Time       ; start time
10|      BL delay           ; call delay function
11|      B flash
12|      halt
```

`BL delay` sets:

- the PC register to be address of delay
- the LR to register to the address of next instruction

DEFINING FUNCTIONS

- Imagine we had already written a function called “delay”:



Defining delay function

- So far we have only talked about how we might call a delay function
- Lets now think about writing the actual function
- Lets the write the function so that it takes a single argument:
  - The number of seconds to delay

DELAY FUNCTION

```
delay:
 push {R3,R4,R5,R6}
 MOV R3, R0        ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
            LDR R5, .Time     ; update time
            SUB R6, R5, R4    ; calc elapsed time
            CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

DELAY FUNCTION

```
delay:                                  Label defining start of function in memory
 push {R3,R4,R5,R6}
 MOV R3, R0        ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
          LDR R5, .Time     ; update time
          SUB R6, R5, R4    ; calc elapsed time
          CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

DELAY FUNCTION

```
delay:
 push {R3,R4,R5,R6}
 MOV R3, R0        ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
          LDR R5, .Time     ; update time
          SUB R6, R5, R4    ; calc elapsed time
          CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

Push all registers we are about to use onto stack so we can restore them at the completion of function

DELAY FUNCTION

This is where we accept the parameter passed in via R0. We move it into a register to work with locally

```
delay:
 push {R3,R4,R5,R6}
 MOV R3, R0      ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
          LDR R5, .Time     ; update time
          SUB R6, R5, R4    ; calc elapsed time
          CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

DELAY FUNCTION

```
delay:
 push {R3,R4,R5,R6}
 MOV R3, R0        ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
          LDR R5, .Time     ; update time
          SUB R6, R5, R4    ; calc elapsed time
          CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

This implements the Delay just like we Saw in Week 8 (the "better dumb timer")

DELAY FUNCTION

```
delay:
 push {R3,R4,R5,R6}
 MOV R3, R0        ; move delay time param into R3

 LDR R4, .Time     ; get start time
timer:
          LDR R5, .Time     ; update time
          SUB R6, R5, R4    ; calc elapsed time
          CMP R6, R3        ; check elapsed time
 BLT timer

 pop {R3,R4,R5,R6}
 RET
```

An ARMlite instruction specifically for returning from Functions. It makes sure the Program Counter register is updated using the address in Link Register

BL AND RET

- BL and RET instructions work as a pair
- BL (Branch to Label) essentially does this:

```
MOV LR,PC  ; copy current next instruction to LR
MOV PC, #delay    ; set PC to make “delay” the next instruction
```

BL AND RET

- BL and RET instructions work as a pair
- BL (Branch to Label) essentially does this:

```
MOV LR,PC  ; copy current next instruction to LR
MOV PC, #delay    ; set PC to make “delay” the next instruction
```

We do this so we know where to return to after the function is finished

BL AND RET

- BL and RET instructions work as a pair
- BL (Branch to Label) essentially does this:

```
MOV LR,PC  ; copy current next instruction to LR
MOV PC, #delay    ; set PC to make “delay” the next instruction
```

We do this so we know where to jump to next to execute the function.

You could do this of using BL, but note that the order here matters (think about why ?)

BL AND RET

- RET (ie., RETurn), is how we exit functions, and return where we came from.
- RET essentially does this:
  - MOV PC, LR
    - It copies the original next instruction (ie., the one we first copied into LR before we called the function) back to PC so that it is the next instruction executed.

WHOLE PROGRAM

```
 1|      mov R0, #.green
 2|      mov R1, #.white
 3|      mov R2, #1         ; 1sec delay time
 4|flash:
 5|      str R0,.Pixel367   ; flash on
 6|      LDR R3,.Time       ; start time
 7|      push {R0}
 8|      MOV R0, R2
 9|      BL delay           ; call delay function
10|      Pop {R0}
11|      str R1,.Pixel367   ; flash off
12|      LDR R3,.Time       ; start time
13|      push {R0}
14|      MOV R0, R2
15|      BL delay           ; call delay function
16|      pop {R0}
17|      B flash
18|      halt
19|delay:
20|      push {R3,R4,R5,R6}
21|      MOV R3, R0         ; move delay time param into R3
22|      LDR R4, .Time      ; get start time
23|timer:
24|      LDR R5, .Time      ; update time
25|      SUB R6, R5, R4     ; calc elapsed time
26|      CMP R6, R3         ; compare elapsed to delay time
27|      BLT timer
28|      pop {R3,R4,R5,R6}
29|      RET
```

SUMMARY

- Function calls require branching to a different instruction address
  - Use bl to branch to a label
  - Use RET to return back to calling code
- Program counter (pc) and Link Registers:
  - pc: address of the next instruction to execute
  - lr: address of instruction to return to after a function is complete