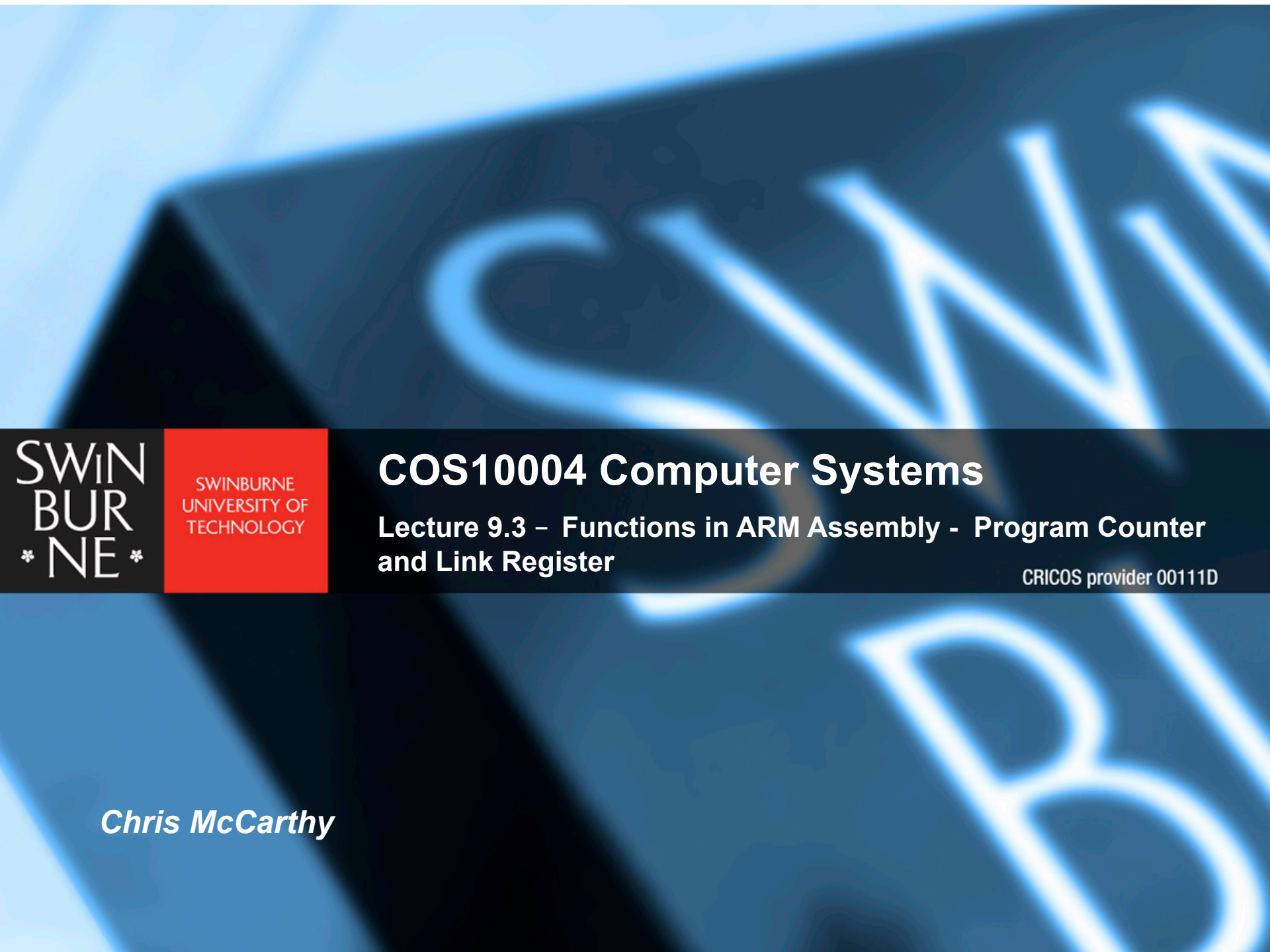


COS10004 Computer Systems

Lecture 9.3 – 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 THE ABI

- **Application Binary Interface** (ABI) sets standard way of using ARM registers.
  - r0-r3 used for function arguments and return values
  - r4-r12 promised not to be altered by functions
  - **lr** and **sp** used for stack management
  - **pc** is the next instruction – we can use it to exit a function call

EXAMPLE CODE FRAGMENT

```
loop$:
  str r1,[r0,#32] ;on
  push {r0,r1}          ;save a backup copy of r0
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
  pop {r0,r1} ;restore the backup copy of r0
  str r1,[r0,#44] ;off
  push {r0,r1}
 mov r0,BASE
 mov r1,$80000
   bl Delay  ;call Delay
pop {r0,r1}
b loop$
```

EXAMPLE CODE FRAGMENT

```
loop$:
  str r1,[r0,#32] ;on
  push {r0,r1}          ;save a backup copy of r0
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
  pop {r0,r1} ;restore the backup copy of r0
  str r1,[r0,#44] ;off
  push {r0,r1}
 mov r0,BASE
 mov r1,$80000
   bl Delay  ;call Delay
pop {r0,r1}
b loop$
```

Calling function “Delay”

Program control jumps to Instruction address represented by the label Delay

EXAMPLE CODE FRAGMENT

```
loop$:
  str r1,[r0,#32] ;on
  push {r0,r1}          ;save a backup copy of r0
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
  pop {r0,r1} ;restore the backup copy of r0
  str r1,[r0,#44] ;off
  push {r0,r1}
 mov r0,BASE
 mov r1,$80000
   bl Delay  ;call Delay
pop {r0,r1}
b loop$
```

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

How does it know how to get back ???

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 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!)

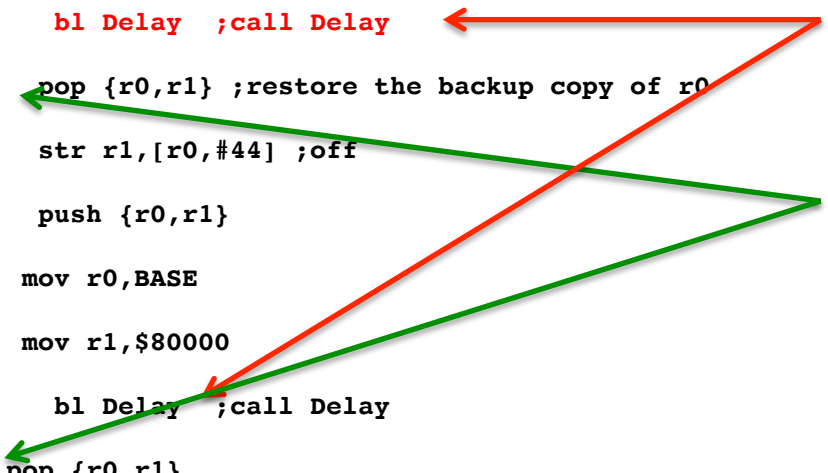
HELPFUL INSTRUCTION - BL

- `bl label$:`
  - causes program control to jump to label$,

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

EXAMPLE CODE FRAGMENT

```
loop$:
  str r1,[r0,#32] ;on
  push {r0,r1}          ;save a backup copy of r0
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
 pop {r0,r1} ;restore the backup copy of r0
  str r1,[r0,#44] ;off
  push {r0,r1}
 mov r0,BASE
 mov r1,$80000
   bl Delay  ;call Delay
pop {r0,r1}
b loop$
```



`bl Delay` sets:

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

INSIDE DELAY

```
Delay:  ;params: r0 = BASE, r1 = $800000
mov r3,r0    ; move base address to local register
orr r3,$00003000
mov r4,r1    ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:  ;label still has to be different from all the others
  ldrd r6,r7,[r3,#4]
  sub r8,r6,r5
  cmp  r8,r4
  bls loopt1 ;branch if lower or same (<=)
mov pc,lr  ;move contents of lr into pc
```

INSIDE DELAY

```
Delay:  ;params: r0 = BASE, r1 = $800000
mov r3,r0    ; move base address to local register
orr r3,$00003000
mov r4,r1    ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:  ;label still has to be different from all the others
  ldrd r6,r7,[r3,#4]
  sub r8,r6,r5
  cmp  r8,r4
  bls loopt1 ;branch if lower or same (<=)
mov pc,lr  ;move contents of lr into pc
```

This ensures the next instruction executed is the one immediately after the function call !

DELAY FUNCTION (ALTERNATIVE)

```
Delay:   ;params: r0 = BASE, r1 = $800000
push {lr}
mov r3,r0
orr r3,$00003000
mov r4,r1   ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:   ;label still has to be different from one in _start
   ldrd r6,r7,[r3,#4]
   sub r8,r6,r5
   cmp  r8,r4
   bls loopt1
pop {pc}   ;return
```

We can also use the stack to store and recall the lr

The pop operation writes the address to the pc register

DELAY FUNCTION (ALTERNATIVE)

```
Delay:   ;params: r0 = BASE, r1 = $800000
push {lr}
mov r3,r0
orr r3,$00003000
mov r4,r1   ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:   ;label still has to be different from one in _start
   ldrd r6,r7,[r3,#4]
   sub r8,r6,r5
   cmp  r8,r4
   bls loopt1
pop {pc}   ;return
```

We can also use the stack to store and recall the lr

The pop operation writes the address to the pc register

Using the stack to store lr is needed when there are nested function calls !

Why ? … think about what happens to lr in this case.

DELAY FUNCTION (BETTER)(2)

```
Delay:  ;params: r0 = BASE, r1 = $800000
mov r3,r0
orr r3,$00003000
mov r4,r1    ;~0.5s
ldrd r6,r7,[r3,#4]
mov r5,r6
loopt1:  ;label still has to be different from all the others
  ldrd r6,r7,[r3,#4]
  sub r8,r6,r5
  cmp  r8,r4
  bls loopt1 ;branch if lower or same (<=)
bx lr  ;return to lr - no need to update pc ourselves
```

TIMER3.asm

This way works best with the FASMARM compiler

A BIT MORE TO BX (JUST AN FYI)

- bx stands for “branch exchange”
- It exchanges the ARM instruction set for the “thumb” instruction set.
- ARM instructions don’t support stack operations (push, pop), so we need to use thumb mode instructions.
  - Thumb mode has fewer registers (r0-r7) but it runs faster- it’s 16-bit.
  - Recursive functions MUST be in thumb state because they use the stack.
  - Any function which calls another function (and pushes things onto the stack) must run in thumb state.

- More details here:
- http://www.embedded.com/electronics-blogs/beginner-s-corner/4024632/Introduction-to-ARM-thumb

SUMMARY

- Function calls require branching to a different instruction address
  - Use bl to branch
  - Use bx lr to return
- 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