COS10004 Computer Systems

Lecture 9.1 – Functions in ARM Assembly - Function basics

Chris McCarthy

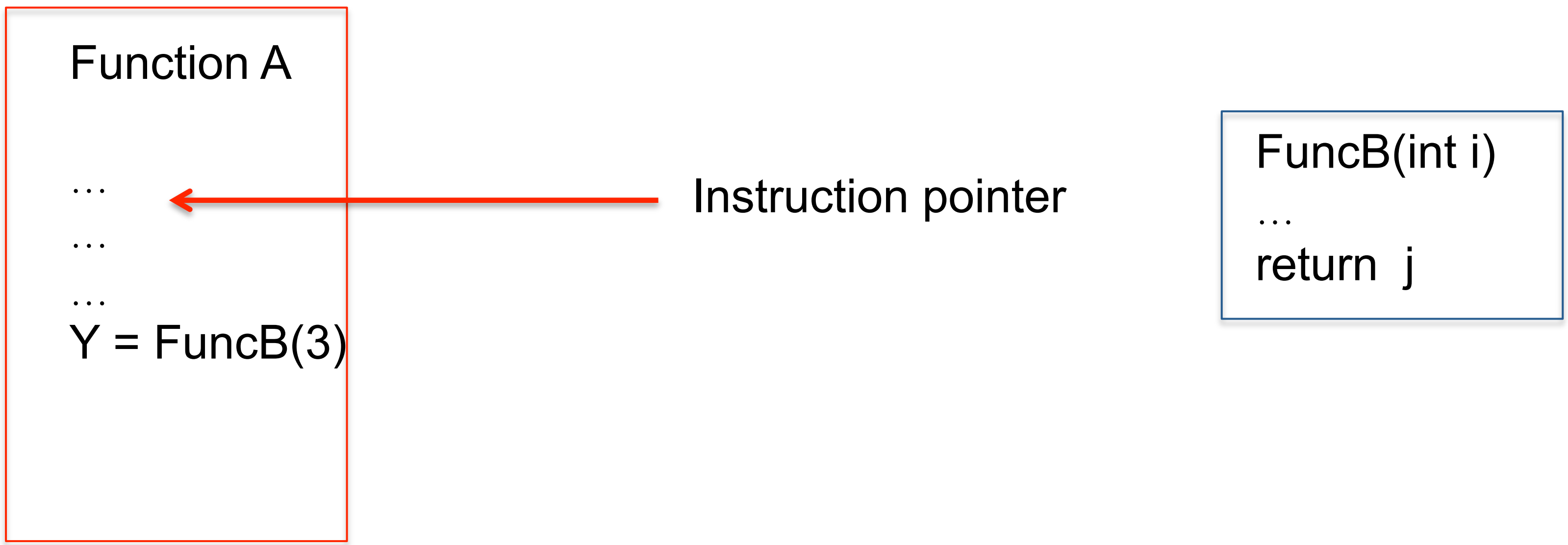
ARM ASM

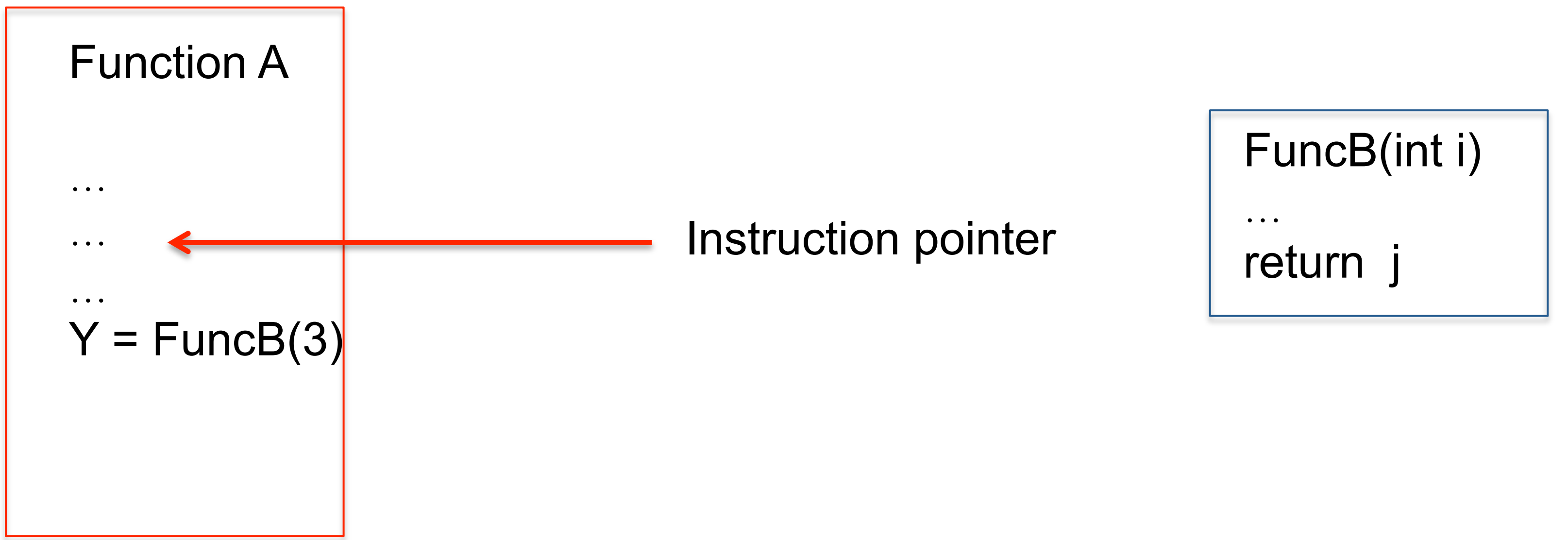
- *assignment (mov, ldr, str)*
- *arithmetic (add, sub, mul, ~~div~~)*
- *labels, branch (b)*
- *registers, GPIO*
- *selection (cmp, tst)*
- *functions, parameters (bl)*
- *stack (push, pop)*
- *aliases/variables (.req, .unreq)*
- *ARM timer*
- *Turn on|off GPIO (gpio.s) (OK01, OK02)*

FUNCTIONS

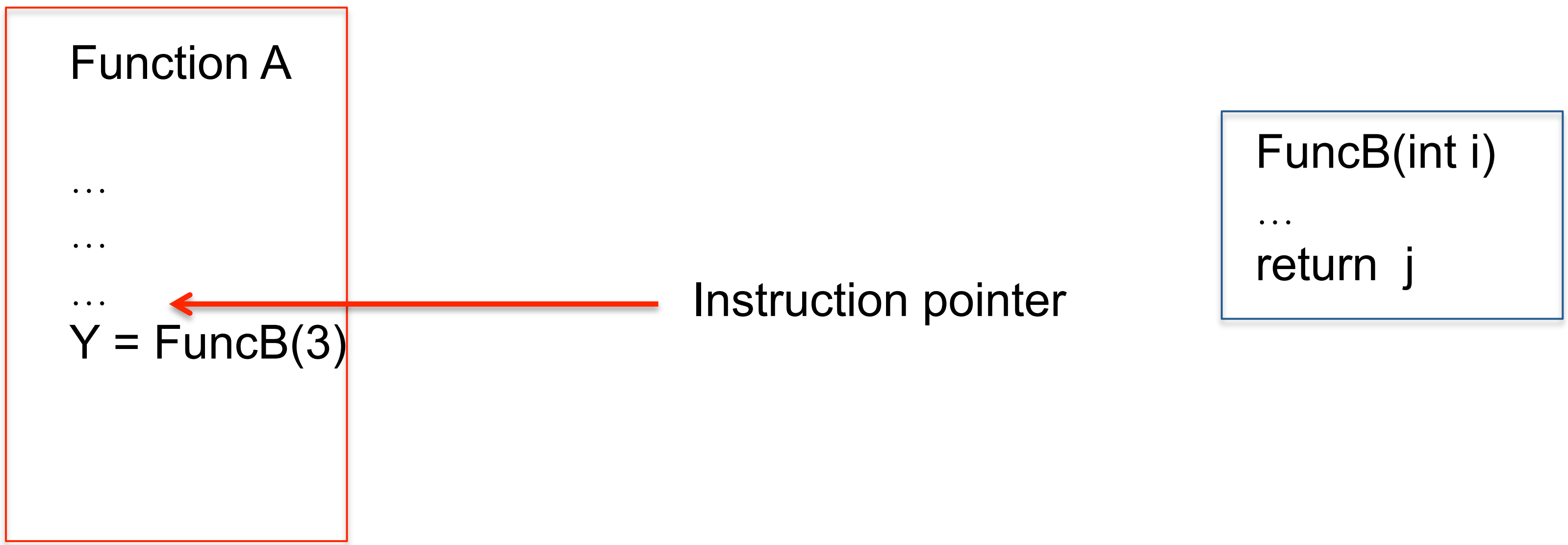
- Functions/methods/procedures/sub-routines:
  – A callable block of organised, re-usable code
  – Typically single action
  – accepts arguments (ie parameters)
- Eg in C:

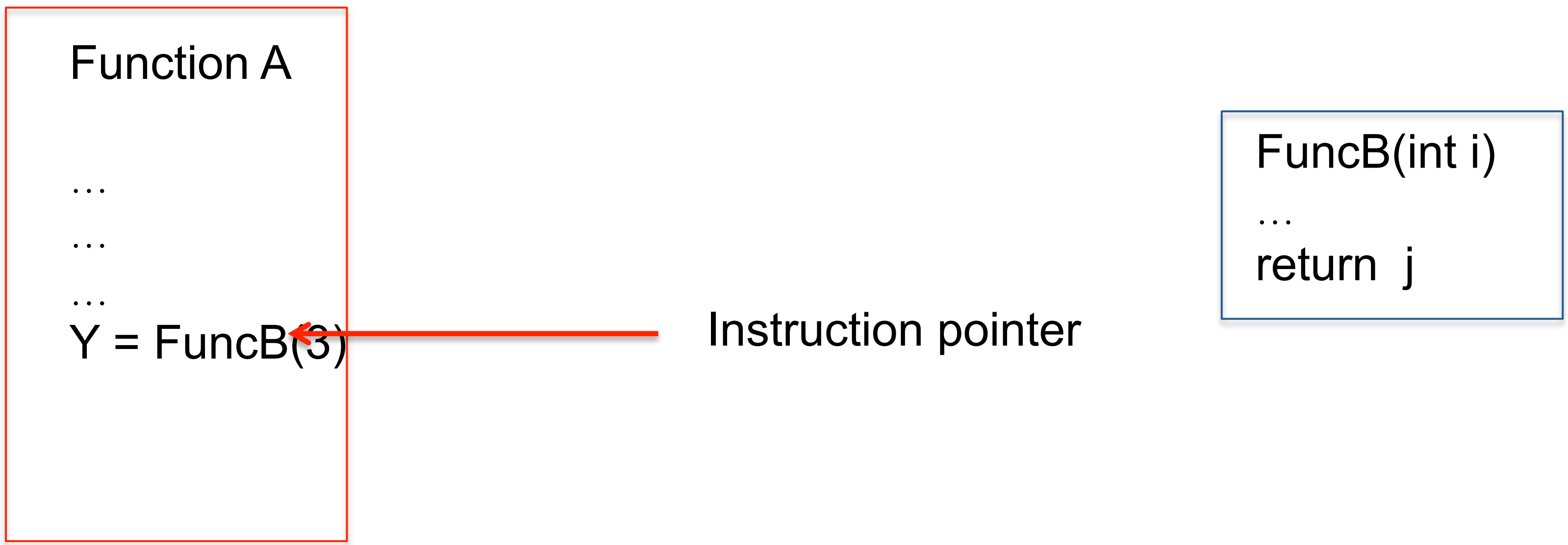
```
int add(int x, inty)
{
    int sum = x + y;
    return(sum);
}
```

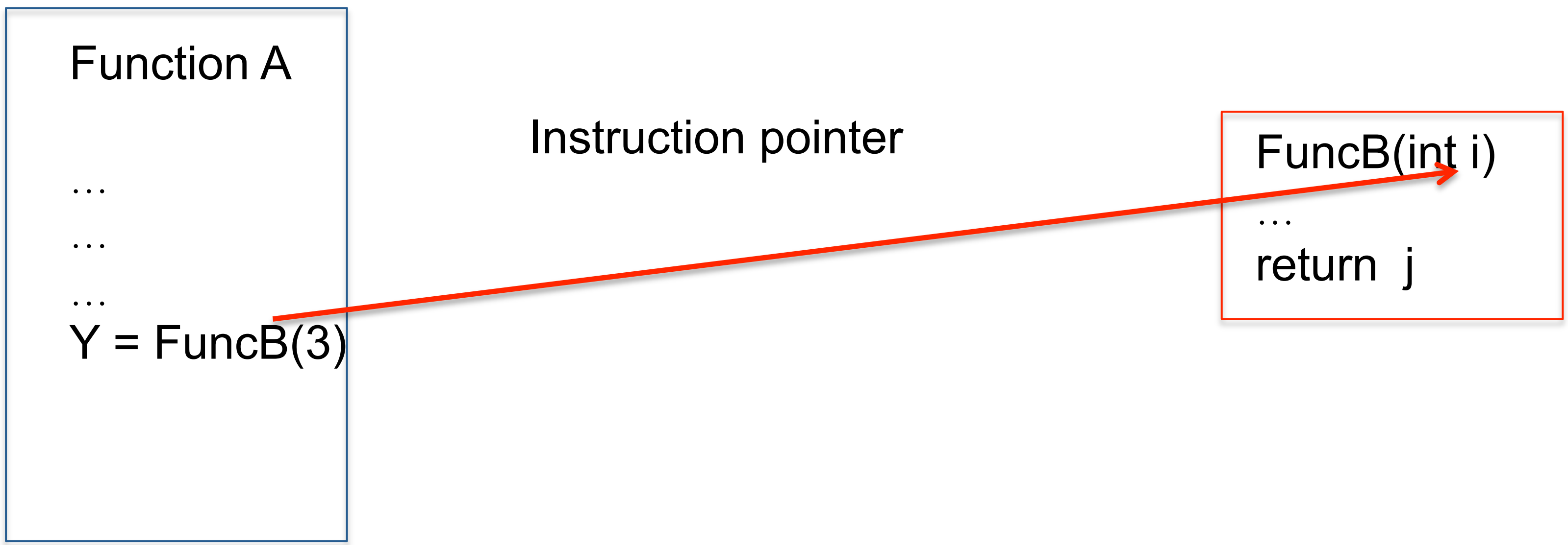

Function A
...
...
...
Y = FuncB(3)
Instruction pointer
FuncB(int i)
...
return j

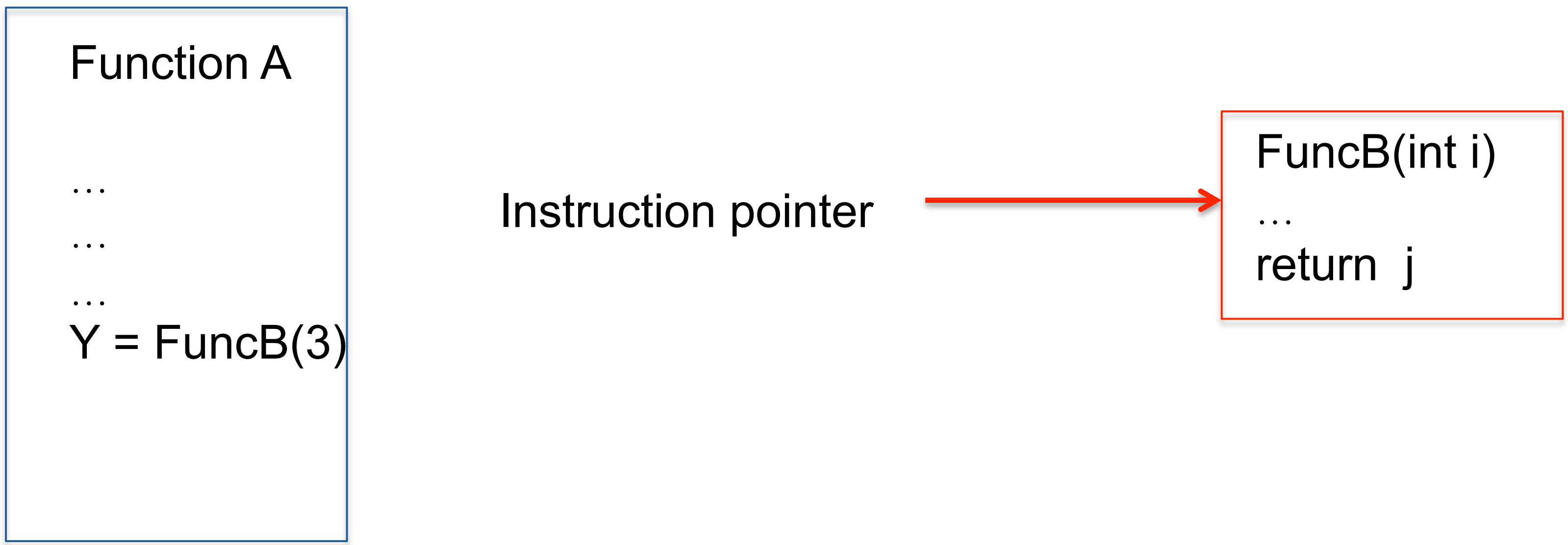


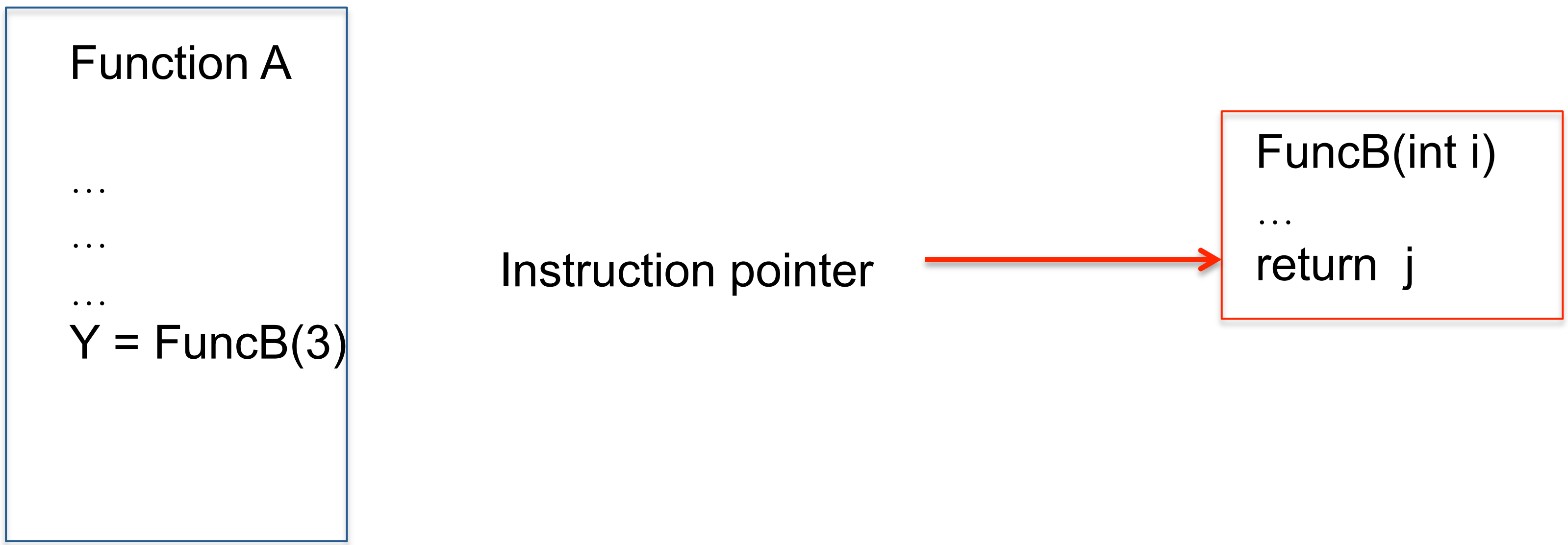
Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j

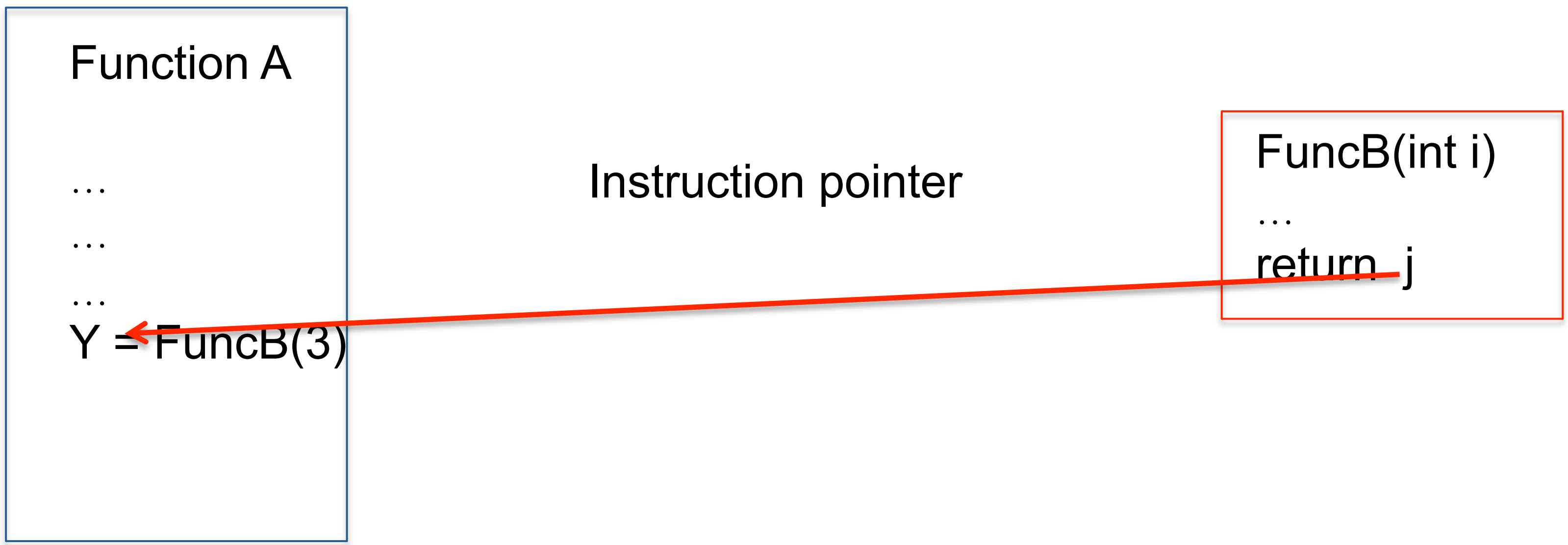

Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j

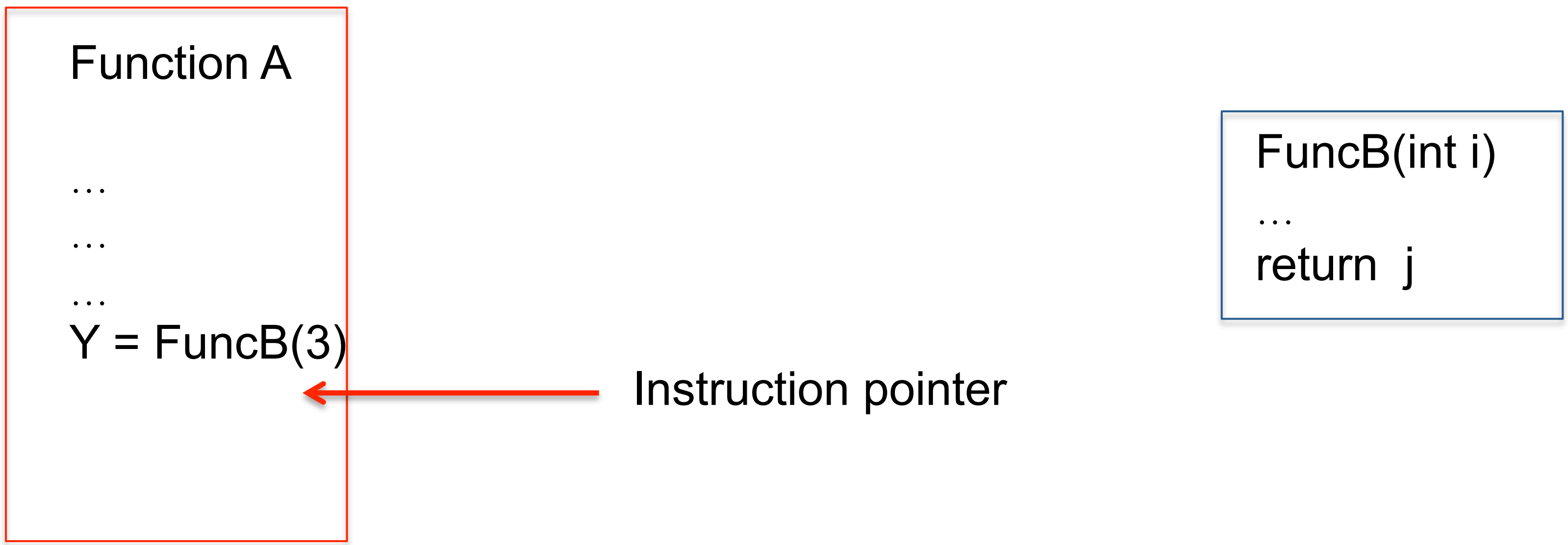

Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j


Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j


Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j


Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j


Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j



Function A
…
…
…
Y = FuncB(3)
Instruction pointer
FuncB(int i)
…
return j

FUNCTION BASICS

- When a function is called:
  - Arguments need to be placed somewhere the function can access
  - program control shifts to the function's instructions
- When a function completes:
  - Return value needs to placed somewhere for the calling function to retrieve
  - program control shifts back to the instruction immediately after where it was called from
- Managing this requires a some *house keeping* needed *!*
  - High level programming languages hide most of this !
  - Not ASM!

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 !

REGISTER MANAGEMENT

- **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

ABI

Register	Brief	Preserved	Rules
r0	Argument and result	No	r0 and r1 are used for passing the first two arguments to functions, and returning the results of functions. If a function does not use them for a return value, they can take any value after a function.
r1	Argument and result	No	
r2	Argument	No	r2 and r3 are used for passing the second two arguments to functions. There values after a function is called can be anything.
r3	Argument	No	

Calling functions

- By convention, the first two function arguments are loaded into r0 and r1.
- The next two are put into r2 and r3.
- The return value of the function is written into r0 and r1 (lowest word in r0).
- The function promises not to alter r4-r12.
- ... but suppose the function needs to use many registers to do calculations...

AND ANOTHER THING...

The RPi 2 has a 4-core ARM7 architecture (and Rpi 3 is ARM8).

- Both compatible with 32-bit and 64-bit software (like Windows 10).
- Sometimes need to use registers in pairs (remember the timer registers?) to get 64-bit wide values.
- r0+r1, r2+r3, r4+r5, r6+r7, r8+r9, r10+r11
- Only 6 64-bit registers available on RPi2 (more on version 3). We need to be smarter
  - *by using the stack!*

SUMMARY

- Functions are the building blocks of programs:
  - Organised, re-usable blocks of code
- Higher level programming languages have built in support for functions:
  - Not ASM!
- One thing we need to manage is register use
- Application Binary Interface (ABI) defines conventions for the use of registers
- Next lecture:
  - How do we store and recall register values ? With a stack of course !

PRESERVING VALUES WITH THE STACK

- The solution is simple.
- Push any registers we want to preserve (e.g. r0-r3) onto the stack before setting their values (as function arguments).
  – Push other registers (r4-12) on to the stack before re-using them.
- Pop them off the stack when the function returns. MUST BE DONE IN REVERSE ORDER
- Process: mov the return value (from r0,r1) and then pop r0 and r1 off the stack.

EXAMPLE SYNTAX

- Push and pop accept multiple registers if in a { , , ,...} list

```
push {r4,r5}  ;back them up onto the stack
;use r4 and r5  for something else
pop {r4,r5} ;restore them from the stack
```

Correct order is preserved for {lists}

- Alternatively, do one at a time (but pop in reverse order)

```
push {r4}
push {r5}
; do something
pop {r5}
pop {r4}
```

CALLING AND RETURNING

- ARM Assembly does not have call and return operations.
  - Simulate them with branch operations.
- **lr** (link register) contains the address of the next instruction after a function call.
  - We use this to tell the code what to run after a function finishes.
  - The current address of code to be run is stored in the program counter (**pc**). Setting this to the value in **lr** makes the program resume after a function has finished.

2 WAYS OF MANAGING LR, PC

```
FunctionLabel:
;do something
mov pc,lr  ;set pc to the
 next line of the caller
```

- Alternatively (better),

```
FunctionLabel:
push {lr}
;do something
pop {pc}
```

- Calling function:

```
bl FunctionLabel
```

Better because now we can call functions from within functions

bl means *branch with link* - updates **lr** to contain the next address after the branch

Each push in here must be matched with a pop so that the value popped into **pc** is the value that was in **lr**

Delay function (2)

```
Delay:  ;this function has no parameters
mov r3,$3F000000 ;RPi2 and 3
orr r3,$00003000
mov r4,$80000 ;~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  ;return
```

Delay function (alternate)

```
Delay:  ;this function has no parameters
push {lr}
mov r3,$3F000000
orr r3,$00003000
mov r4,$800000 ;~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
```

DELAY FUNCTION (BETTER)(2)

```
Delay:  ;this function has no parameters
mov r3,$3F000000
orr r3,$00003000
mov r4,$80000 ;~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  ;branch to lr without updating pc (return)
```

TIMER3.asm

This way works best with the FASMARM compiler

THE REST OF THE CODE (2)

```
;OK4 with functions (LED connected to GPIO18)
mov r0,$3F000000
orr r0,$00200000
mov r1,#1
lsl r1,#24   ;GPIO18
str r1,[r0,#4]
mov r1,#1
lsl r1,#18
loop$:
  str r1,[r0,#32] ;on
  bl Delay   ;call Delay
  str r1,[r0,#44] ;off
  bl Delay   ;call Delay
b loop$
include "TIMER3.asm"
```

OK3F.zip

FORGETTING WHICH REGISTER DOES WHAT?

- We can use labels to associate constant values with names (memory addresses in the code), and MOV those names into registers.

```
BASE = $3F000000
GPIO_OFFSET=$20000
TIMER_OFFSET=$3000   ;sets up hard-coded constants
Mov r0, BASE
ORR r0,GPIO_OFFSET   ;puts values into registers
```

…

- Just like pointers and values in C, can't perform operations on constant names.

```
      ADD BASE,#1 ;won't work
               MOV r0,BASE
               ADD r0,#1    ;works
```

MAIN FUNCTION (NAMED CONSTANTS)

```
BASE       = $3F000000
GPIO_OFFSET=$00200000
mov r0,BASE
orr r0,GPIO_OFFSET
mov r1,#1
lsl r1,#24   ;GPIO18
str r1,[r0,#4]
mov r1,#1
lsl r1,#18
loop$:
  str r1,[r0,#28] ;on
  bl Delay   ;call Delay
  str r1,[r0,#40] ;off
  bl Delay   ;call Delay
b loop$
include "TIMER2.asm"
```

Delay function (named constants)

```
Delay:  ;this function has no parameters
TIMER_OFFSET=$3000
mov r3,BASE
orr r3,TIMER_OFFSET
mov r4,$80000 ;~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  ;branch to lr without updating lr (return)
;Note: main code (prev. slide) is loaded before include,
so compiler knows what BASE is
```

TIMER2.asm

push

Break time

How many beans are in my cup of coffee? In decimal, Hex, Octal, any radix?

Break time

Counting in software...

- Counting requires two operations
- Increment
- Carry

- We can do this in hardware (Flip-flops, gates) but it's cheaper to build a universal machine which executes code.
- Then we only have to program it to count.

- Harder than you think!

RECURSION ACTUALLY MAKES THIS EASIER

```
// radix = base of number (global)
void Increment(counter[], digitIdx)
{
   if (digitIdx <= maxDigitIdx)
   {
      if (counter[digitIdx] == radix-1)  //carry
      {
          counter[digitIdx]=0;
          Increment(counter[], digitIdx+1)
      } else {
          counter[digitIdx]++;               //increment
      }
   }
}
```

In ASM*

```
Increment:
;r0 = counter_address
;r1 = digit
;r2 = maxDigit
;r3 = radix-1
mov r4,r1  ;copy for later to a temp variable
cmp r1,r3  ;if digit == maxDigit return
beq end
cmp r0[r1], r3  ;if this digit != radix-1 (e.g. 9)
bne continue    ;just add 1 (increment)
;carry
mov r0[r1], #0  ;reset this counter
add r4,#1       ;add 1 to copy of digit
push {lr}       ;backup lr (we'll need it later when the next line returns)
 bl Increment
pop {lr}
b end  ;all done
continue:
add r0[r1], #1 ;increment
end:
;call display function here
bx lr
```

*not tested on computers

pop

PASSING ARGUMENTS TO FUNCTIONS

- So far we have made our code a bit neater, but we have still used all of the registers.
- To re-use the registers we need to:
  - Back up registers we need to re-use in a function
  - Store arguments for the function in r0-r3
  - Call the function
  - Read the return values from r0-r1 (optional)
  - Restore the registers we backed up.

PASSING ARGUMENTS TO FUNCTIONS

- We could store the arguments in registers r4-r12, but the ABI says put them in r0-r3.
  - We'll send r0 (BASE) and r1 (the time - $80000).
  - The main loop will look like this:

```
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,$80000
   bl Delay   ;call Delay
pop {r0,r1}
b loop$
```

Really common to do this because r0-r3 are input params – need to set them every time we call a function.

r0-r1 contain the return value if there is one.

Delay function (receives base, time)

```
Delay:  ;this function has 2 parameter
TIMER_OFFSET=$3000
mov r3,r0   ;BASE passed in r0
orr r3,TIMER_OFFSET
mov r4,r1   ;$80000 passed in r1
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
bx lr  ;return
```

timer2_2Param.asm

SOFTWARE STACK

- With the RPi we need to initialise the stack pointer (sp) before doing pushes and pops.

```
MOV SP, $1000
;should be enough room (4096 bytes)
```

MAIN PROGRAM CODE

```
format binary as 'img'
mov sp,$1000  ;make room on stack
BASE       =$3F000000
GPIO_OFFSET=$00200000
mov r0,BASE
orr r0,GPIO_OFFSET
mov r1,#1
lsl r1,#21  ;B+,2 GRN
str r1,[r0,#16]
mov r1,#1
lsl r1,#15
loop$:
 str r1,[r0,#32] ;on
  push {r0,r1}  ;save a backup copy of r0,r1
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
  pop {r0,r1} ;restore the backup copy of r0,r1
 str r1,[r0,#44] ;off
  push {r0,r1}
  mov r0,BASE
  mov r1,$80000
   bl Delay  ;call Delay
 pop {r0,r1}
b loop$
include "timer2_2Param.asm"
```

OK4_2Param.asm

RE-USE

- Our TIMER code will work with any model of Pi, because it gets the BASE address as a parameter.
- We can have *n* versions of the main program (e.g., B+ version, 2B version, 3B version?) that all use the same <u>timer</u> code.
- This is good design.

OK4_2Param.zip

IF REGISTERS ARE ALREADY IN USE...

- push r0,r1,r2,r3 (just to be safe) onto the stack before setting argument values:

```
loop$:
  str r2,[r0,#32] ;on
  push {r0,r1,r2,r3}
  mov r0,$80000
  bl Delay  ;call Delay
  pop {r0,r1,r2,r3}
  str r2,[r0,#44] ;off
  push {r0,r1,r2,r3}
  mov r0,$80000
  bl Delay  ;call Delay
  pop {r0,r1,r2,r3}
b loop$
```

ANOTHER TECHNIQUE

- In other versions of ASM, it is common practice to push the params onto the stack in reverse order,
  - [inside the function] increment the stack pointer (ESP / SP / whatever) by the number of bytes of params,
  - mov each param from RAM relative to the SP value. `movl 8(%ebp), %eax`
  - Good when passing arrays or lots of params.
  - Lets you get at the stack without popping.

ANOTHER TECHNIQUE

- In other versions of ASM, it is common practice to push the params onto the stack in reverse order,
  - [inside the function] i … k pointer (ESP / … bytes of param…
  - mov e… the SP value.
  - Good … rams.
  - Lets yo… ck without popping.

This is messy.
We will NOT do things this way.
We will use the ABI standard

Multiple arguments

```
loop$:
  str r2,[r0,#32] ;on
  push {r0,r1,r2,r3}
  mov r0,$80000
  mov r1,$3F000000
  mov r2,$00003000
bl Delay  ;call Delay
  pop {r0,r1,r2,r3}
  str r2,[r0,#44] ;/off
  push {r0,r1,r2,r3}
  mov r0,$80000
  mov r1,$3F000000
  mov r2,$00003000
  bl Delay  ;call Delay
  pop {r0,r1,r2,r3}
b loop$
```

Best to back these all up, and restore them all later. Some operations can change values in r0-r3 without your knowledge.

RECEIVING TWO ARGUMENTS

```
Delay:  ; has three parameters
mov r3,r2
orr r3,r1 ;r1+r2 is the address
mov r4,r0 ;r0 is the time
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
bx lr  ;return
```

OTHER COMPILERS…VARIABLES?

```
Delay:  //r0 has two parameters
push {lr}
baseaddress .req r3
delay .req r4
now .req r6
start .req r5
elaspsed .req r8
mov baseaddress,r1 //r1 is the address
mov delay,r0 //r0 is the time
ldrd now,r7,[baseaddress,#4]
mov start,now
loopt1:
ldrd now,r7,[r3,#4]
  sub elapsed,now,start
  cmp  elapsed,delay
  bls loopt1
...
pop {pc} //return
```

replacing the register names with 'meaningful' names

But don't forget to .unreq the variables before you leave

OTHER COMPILERS...VARIABLES?

//replacing the register names with 'meaningful' names.

```
Delay:  //r0 has two parameters
push {lr}
baseaddress .req r3
delay .req r4
now .req r6
start .req r5
elaspsed .req r8
...
//the code using the 'variables'
...
.unreq elaspsed
.unreq start
.unreq now
.unreq delay
.unreq baseaddress
pop {pc} //return
```

order doesn't matter

Forgetting to do this will cause headaches later

ONE MORE THING...

- We promised not to let the function alter r4-r12.
- Solution:

```
push {r4,r5,r6,r7,r8,r9,r10,r11,r12}
```

- ;at start of function (after push{lr})

```
pop {r4,r5,r6,r7,r8,r9,r10,r11,r12}
```

- ;at end of function (before pop{pc})

ARM ASM

- *assignment (mov, ldr)*
- *arithmetic (add, sub, mul, ~~div~~)*
- *labels, branch (b)*
- *registers, GPIO*
- *selection (cmp, tst)*
- *functions, parameters (bl)*
- *stack (push, pop)*
- *aliases/variables (.req, .unreq)*
- *ARM timer*
- *Turn on|off GPIO (gpio.s) (OK01, OK02)*

INCLUDES

- Easy.
- Put each function (or group of related functions) in a dedicated source file.
- The *include* command will combine them with your main.asm (the one you compile) and assemble as one source file.

DIVIDING THE WORK…

- gpio.asm - code for accessing the LED controlled by a GPIO register
- timer.asm - code for using the system timer
- flash.asm - code for flashing the LED *n* times (calls gpio and wait)
- kernel7.asm – startup code for calling flash

Recursion

- Using the stack there is almost no limit to how many processes we can launch.
- We can get a function to call itself over and over again because the stack will hold all the temporary values (inputs, outputs) and play them back in the right order.
- We can program an algorithm to keep going (refining the answer) until it reaches a required level of accuracy.
  - e.g. PI to n decimal places

THINGS WE NEED FOR RECURSION

1. A function must call itself.
2. A function must pass a parameter to itself.
3. The parameter must change in a systematic way.
4. A function must have an exit condition (so that it will stop calling itself.
   – In ASM, we need to push the LR onto the stack for each function call so that all of the calls return a value.
   – Each return will use a different value of LR.

Following code adapted from: http://www.slideshare.net/StephanCadene/arm-procedure-calling-conventions-and-recursion

FACTORIAL

factorialj.asm

- Factorial(n) – n*n-1*n-2*n-3*…*1
- e.g. 4! = 4*3*2*1

```
FACTORIAL:
sub r1,r1,#1    ;3. r1 approaches 1
cmp r1,#1       ;4. exit if 1
beq EXIT
mul r0,r0,r1    ;total=total*param
push {r1,lr}    ;2. push onto the stack,
                ;preserving the PC.
bl FACTORIAL    ;1. call FACTORIAL
EXIT:
pop {r1,lr}    ;pop off the stack
bx lr          ;RETURN
```

CALLING FACTORIALJ.ASM

```
format binary as 'img'  ;must be first
;kernel7.asm
;r0 = current param (changes 4,3,2,1)
;r1 = current answer(changes 4,12,24)
include "factorialj.asm"

mov r1,#4        ;input
mov sp,$1000     ;make room on the stack
mov r0,r1        ;4,4;12,3; 24,2; 24,1
(stops)
bl FACTORIAL
mov r7,r0        ;store answer
```

GETTING THE ANSWER OUT

- We can flash the LED ANSWER times.

```
loop$:        ;r7 contains the ANSWER
  mov r1,#1
  lsl r1,#15
  str r1,[r0,#32]  ;turn LED on

  mov r2,$0F0000
  bl TIMER  ;just a dumb timer here

  mov r1,#1
  lsl r1,#15
  str r1,[r0,#44]  ;turn LED off

  mov r2,$0F0000
  bl TIMER

  sub r7,#1
  cmp r7,#0
bne  loop$
```

The Dumb Timer function

```
;TIMER.asm - dumb timer
;r2=number of loops
TIMER:
  wait1$:
    sub r2,#1
    cmp r2,#0
    bne wait1$
bx lr
```

PUTTING IT TOGETHER…

```
;Calculate
mov r1,#4 ;input
mov sp,$1000  ;make room on the stack
mov r0,r1
bl FACTORIAL
mov r7,r0 ;store answer
BASE = $3F000000 ;RP2 ;GPIO_SETUP
GPIO_OFFSET = $200000
mov r0,BASE
orr r0,GPIO_OFFSET
mov r1,#1
lsl r1,#21
str r1,[r0,#16]     ;set GPIO47 to output
loop$:
  mov r1,#1
  lsl r1,#15
  str r1,[r0,#32]  ;turn LED on
  mov r2,$0F0000  ;not using r2 for anything else so no need to push/pop
  bl TIMER
  mov r1,#1
  lsl r1,#15
  str r1,[r0,#44]  ;turn LED off
  mov r2,$0F0000
  bl TIMER
sub r7,#1
cmp r7,#0
bne  loop$  ;end of outer loop. Runs r7 times
wait:
b wait
include "TIMER.asm"
include "factorialj.asm"
```

kernel7.asm
recursion.zip

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

THE LAB

- You're going to implement functions and change over the timer in a factorial calculation program.
- Next week: GPIO input handling and screen writing (strap yourselves in!)