SWINBURNE UNIVERSITY OF TECHNOLOGY

COS10004 Computer Systems

Lecture 9.2 ARM Assembly – The Software Stack

CRICOS provider 00111D

Chris McCarthy

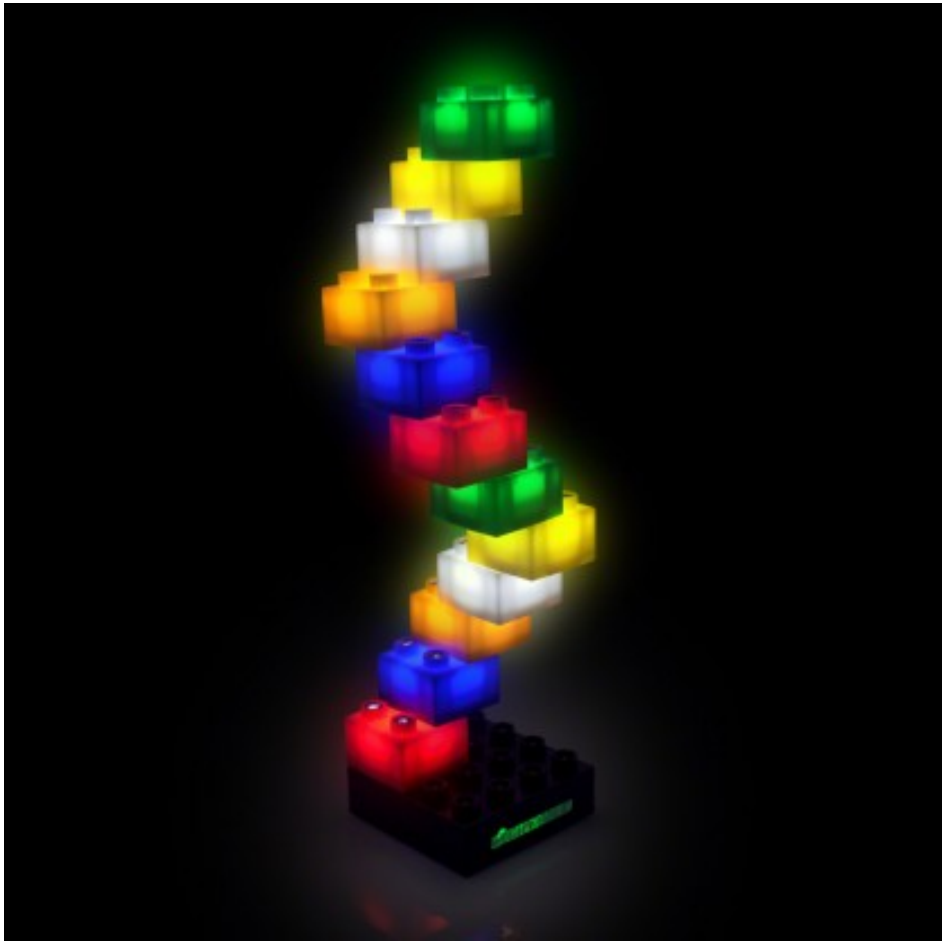
FUNCTIONS IN ASM

- 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 !

STACKS



www.glow.co.uk

PUSH, POP AND THE STACK

- ARM computers have a software stack*.
- A separate area of RAM is available for temporary values.
- A value in a register can be pushed onto the stack to preserve it for later.
- It can be popped off later (in LIFO order).
- We can get the memory location (a pointer to it) by *checking the SP* (R13) register.

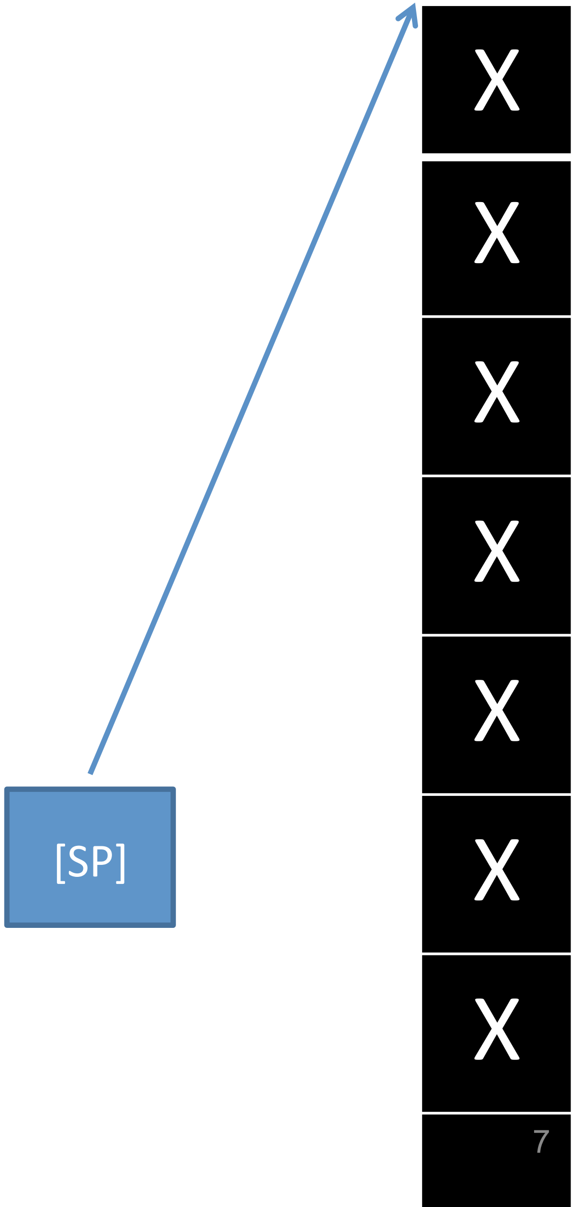
* SOFTWARE STACK?

- A section of RAM managed by the SP (stack pointer) register.
- A sort of 32-bit (64-bit in ARM8) wide array which starts (element 0) high in RAM and grows down as values are added to it.
- The stack pointer stores the memory location of the last value added (pushed) to the stack.
- Each push decrements SP by 4 (4 bytes per word).
- A pop operation removes the last value in the stack and increments the SP by 4 (4 bytes per word)

Software stack (depth only limited by RAM)

Example:

x = don’t care. [SP] points to start of stack.

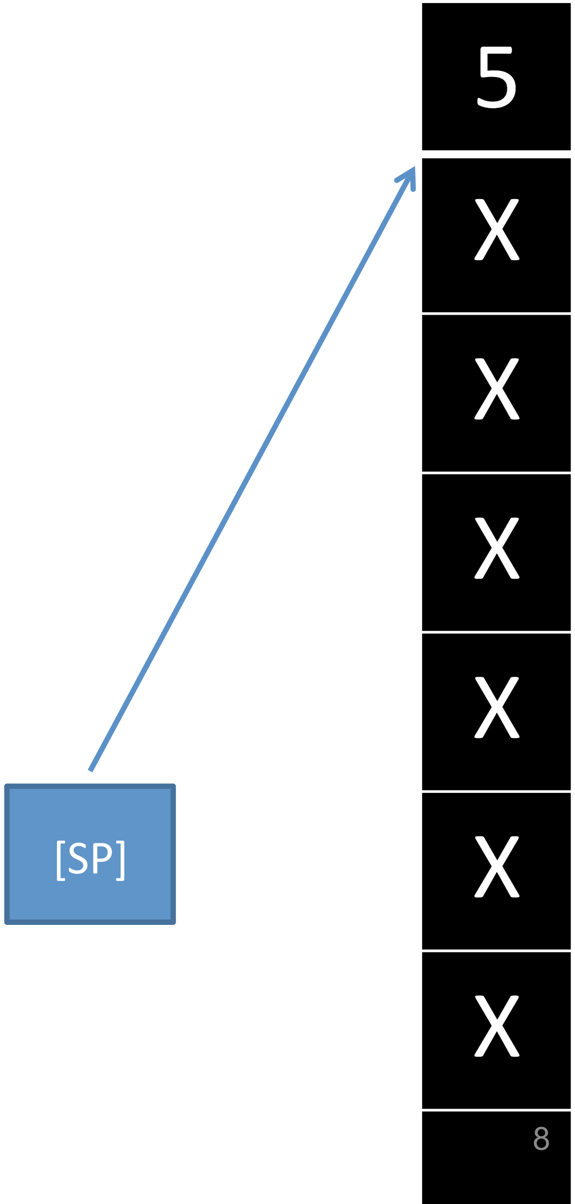


Software stack (depth only limited by RAM)

Example:

x = don’t care.

push 5 – SP decremented by 4. [SP] points to 5



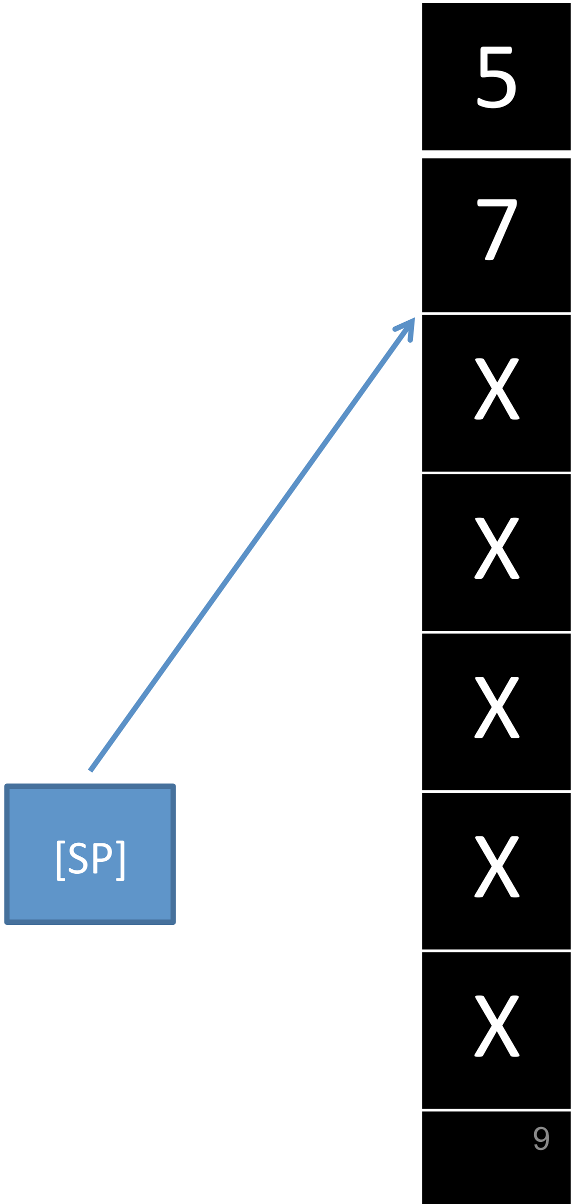
Software stack (32-bit) (depth only limited by RAM)

Example:

x = don’t care.

push 5 – SP decremented by 4

push 7 - SP decremented by 4, [SP] points to 7.



Software stack (depth only limited by RAM)

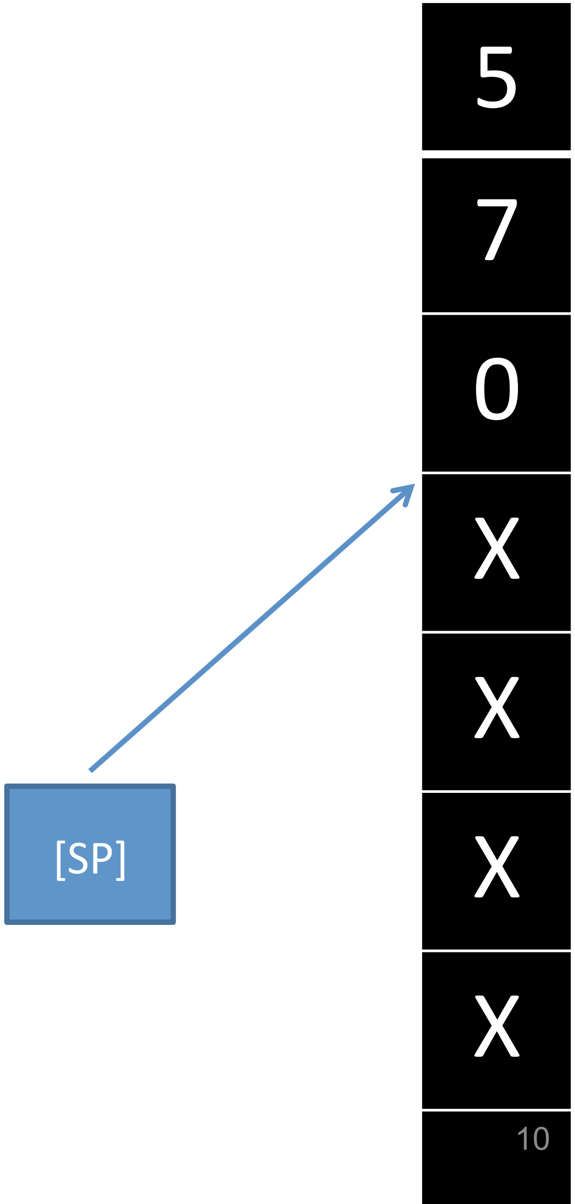
Example:

x = don’t care.

push 5 – SP decremented by 4

push 7 - SP decremented by 4

push 0 – SP decremented by 4. [SP] points to 0



Software stack (depth only limited by RAM)

Example:

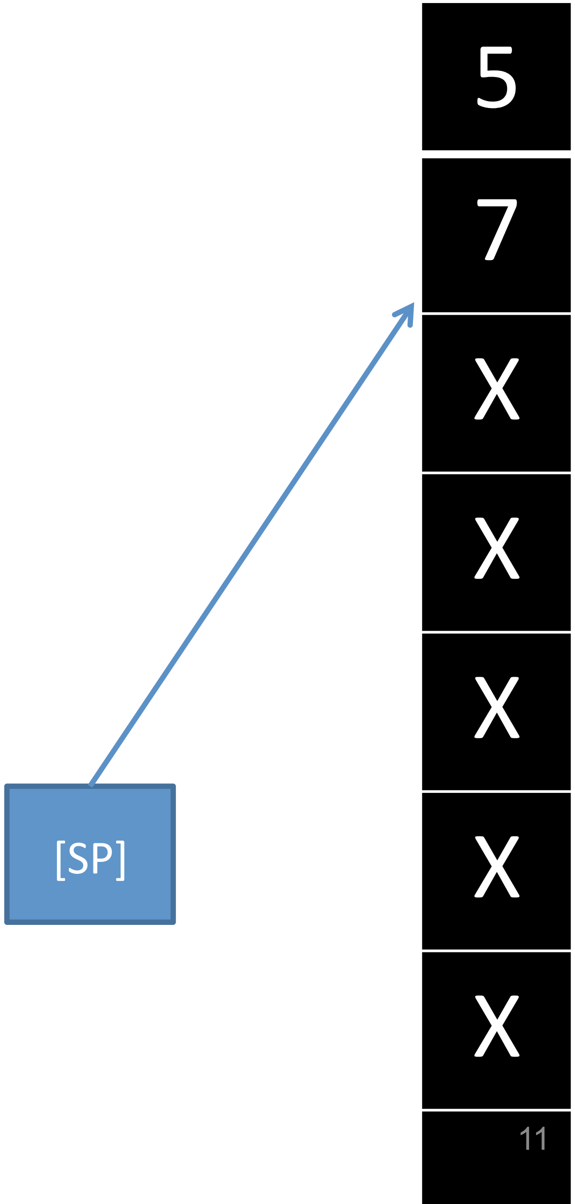
x = don’t care.

push 5 – SP decremented by 4

push 7 - SP decremented by 4

push 0 – SP decremented by 4

pop 0 – SP incremented by 4. [SP] points to 7



Software stack (depth only limited by RAM)

Example:

x = don't care.

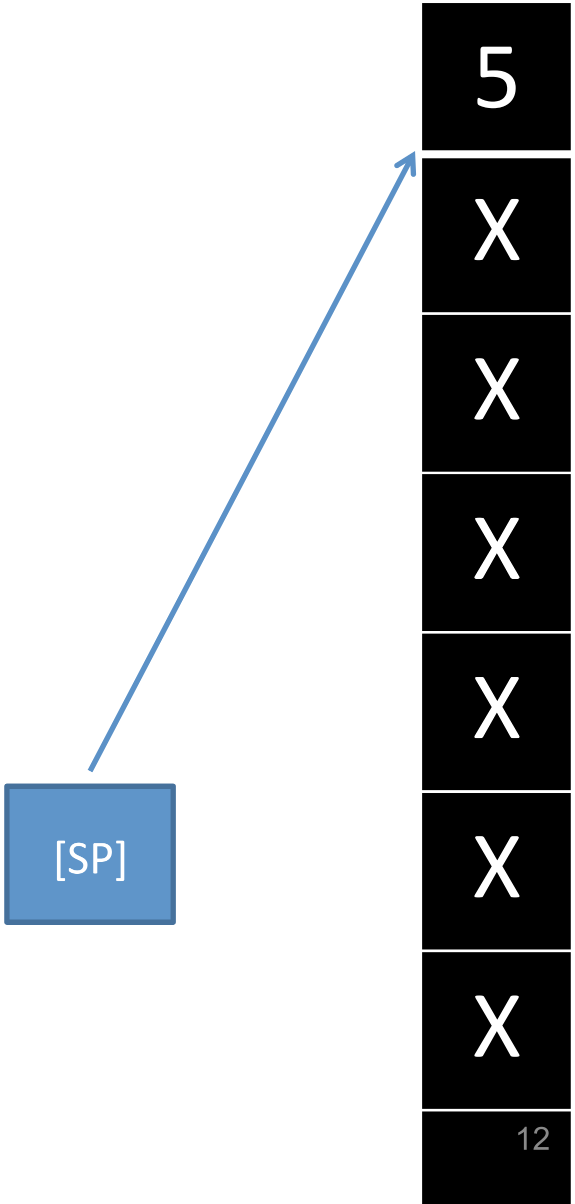
push 5 – SP decremented by 4

push 7 - SP decremented by 4

push 0 – SP decremented by 4

pop 0 – SP incremented by 4

pop 7 – SP incremented by 4. [SP] points to 5.



Software stack (depth only limited by RAM)

Example:

x = don't care.

push 5 – SP decremented by 4

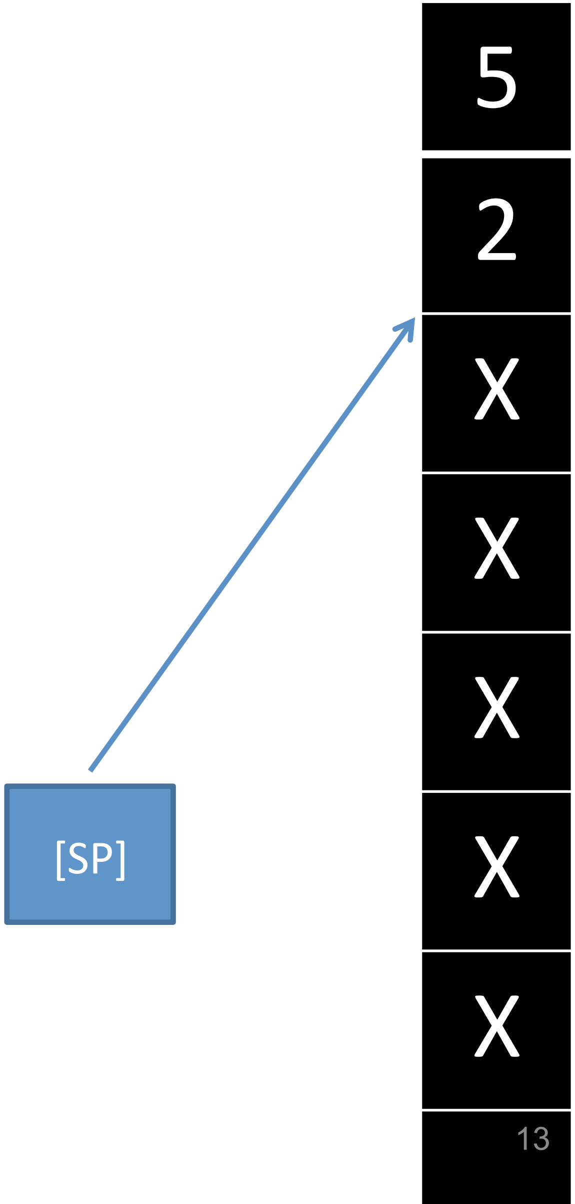
push 7 - SP decremented by 4

push 0 – SP decremented by 4

pop 0 – SP incremented by 4

pop 7 – SP incremented by 4

push 2 – SP decremented by 4. [SP] points to 2.



Software stack (depth only limited by RAM)

Example:

x = don’t care.

push 5 – SP decremented by 4

push 7 - SP decremented by 4

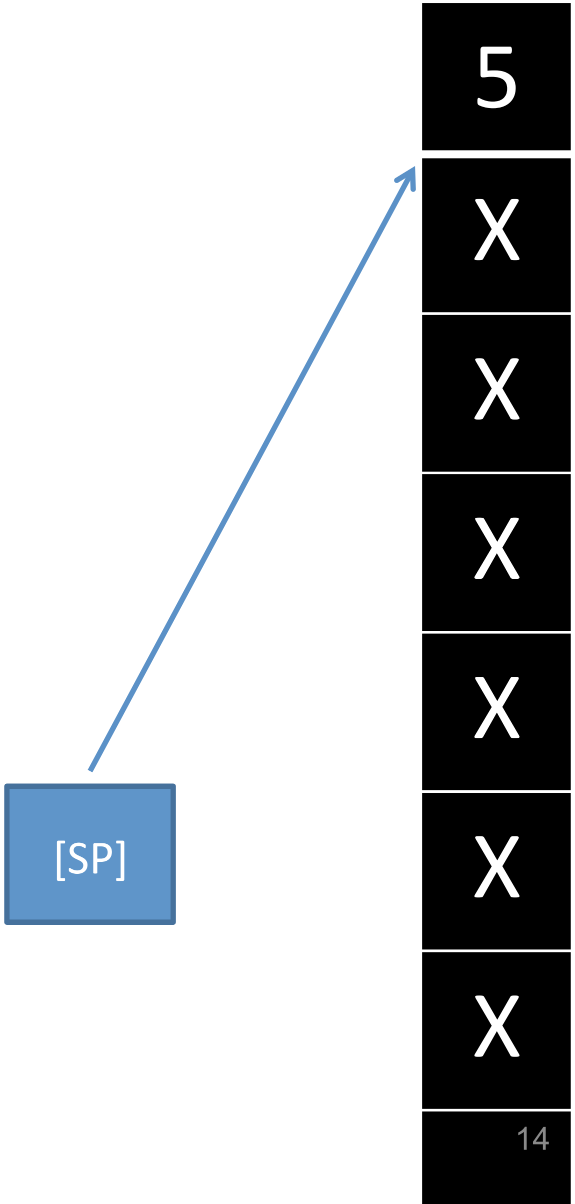
push 0 – SP decremented by 4

pop 0 – SP incremented by 4

pop 7 – SP incremented by 4

push 2 – SP decremented by 4

pop 2 – SP incremented by 4. [SP] points to 5.



Software stack (depth only limited by RAM)

Example:

x = don’t care.

push 5 – SP decremented by 4

push 7 - SP decremented by 4

push 0 – SP decremented by 4

pop 0 – SP incremented by 4

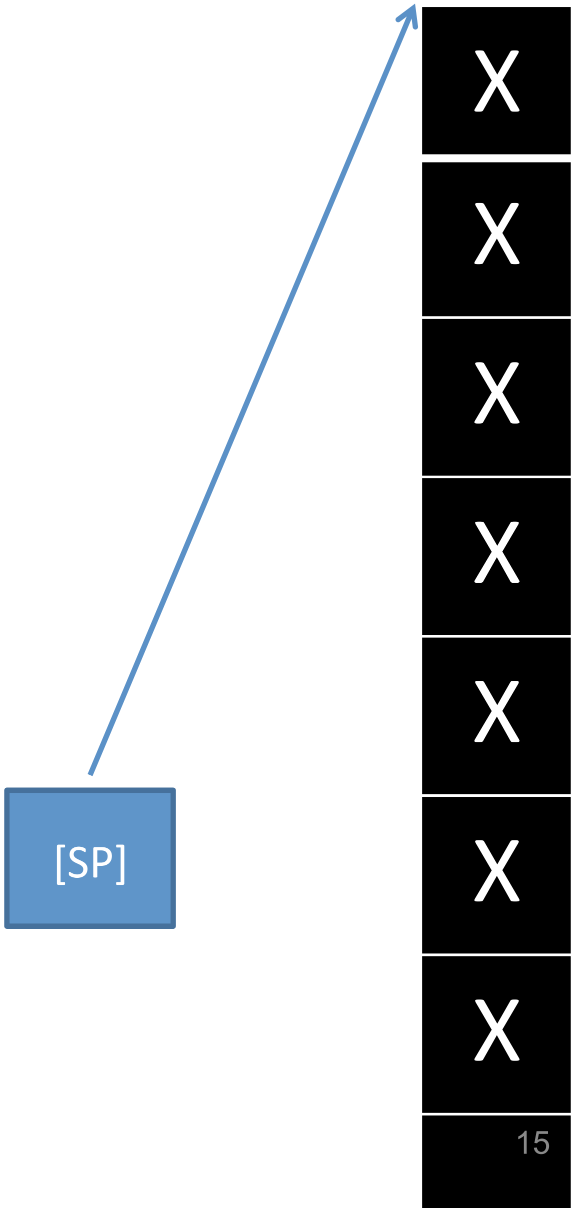
pop 7 – SP incremented by 4

push 2 – SP decremented by 4

pop 2 – SP incremented by 4

pop 5 - SP incremented back to it’s starting value. The stack is now empty.

[SP] points to end of 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}
```

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

ABI CONVENTIONS

- ABI compliant functions:
  - Use r0-r3 for passing and returning values to functions
  - Promise not to alter r4-r12
- … but suppose the function needs to use many registers to do calculations ??
- We can use the stack to store and recall register values !

PASSING ARGUMENTS TO FUNCTIONS

- 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.

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

But before we use r0 and r1 for
passing arguments, we push the
Values they previously held on the stack

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, we bring back
 The original values of r0 and r1 by
“popping” them off the stack

INSIDE DELAY FUNCTION

```
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 timer code.
- This is good design.

OK4_2Param.zip

SUMMARY

- Software Stack:
  - Dedicated RAM used to store values FILO
    - Special register “sp” used to store address of start of the stack
- Stacks allow us to store and recall register values efficiently
- Stacks integral to functions:
  - We need to store and recall register values so we don’t run out of registers to use!
- Next lecture:
  - Managing prigram control with the “lr” and “pc” registers