DESN2000: Engineering Design & Professional Practice (EE&T)

Week 5

Functions, subroutines and procedural call standard

David Tsai

School of Electrical Engineering & Telecommunications

Graduate School of Biomedical Engineering

d.tsai@unsw.edu.au

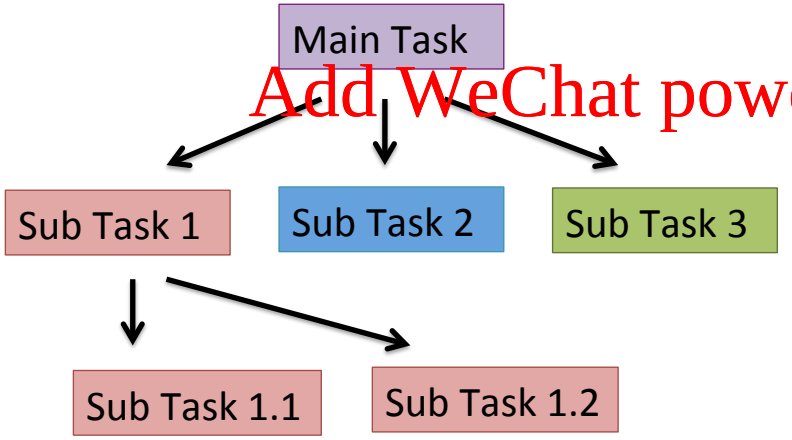
Biomedical Microsystems Lab

This week

- Stack
- Function call
- Stack operations
- ARM architecture procedure call standard (AAPCS)
- Function call examples

Functions & subroutines

- Advantages:
  - Modularise the system.
  - Divide-and-conquer.
- A subroutine that **returns a value** is called a function.




```
Main    xxxxx
        xxxxx
        BL sub1
        BL sub2
        BL sub3

sub1    xxxxx
        BL sub1.1
        BL sub1.2

sub2    xxxxx

sub3    xxxxx

sub1.1  xxxxx
sub1.2  xxxxxx
```

Functions and subroutines

- Need to:
  1. Save and restore information between function / subroutine calls.
  2. Pass information (arguments and return values) to / from subroutines.
- This is achieved via **stacks**.
- Need to follow rules when writing subroutines – e.g. how to handle register conflicts?

- **ARM Architecture Procedure Call Standard** (AAPCS) specifies these rules so that subroutines developed by different programmers can talk to each other.

- Functions/subroutines you develop should be **AAPCS compliant**.

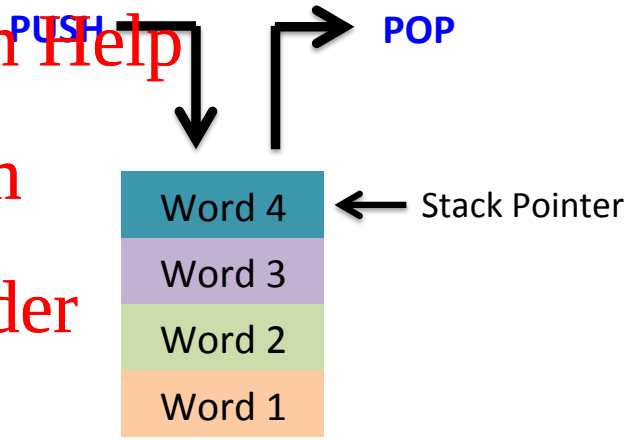
Stack

- Special area in the memory with:
  - Variable length
  - Fixed starting address
- Has **Last-In-First-Out** (**LIFO**) data structure, with two operations:
  - PUSH
  - POP
- These operations typically happen at word level.
- **Stack pointer** (SP or R13) holds the address of either the top-most empty entry or top-most last filled entry in the stack.
- Each processor mode has its own stack pointer.

i.e. each mode has a different stack in memory

Stack

- Data is PUSHED on to the stack using **STR** or **STM** (store multiple) instructions with stack pointer as the base register.
- Data is POPPED from the stack using **LDR** and **LDM** (load multiple) instructions with stack pointer as the base register.
- Upon each PUSH or POP, the stack pointer is updated to point to
  - either the next empty entry
  - or the last filled entry



Stack: operations

- Recall
  - Load multiple:
    LDM<addr_mode>{cond} <Rn> {!}, <register_list> {^}
  - Store multiple:
    STM<addr_mode>{cond} <Rn> {!}, <register_list> {^}

opcode: IA, IB, DA, DB

EQ, NE, etc

Reg, usually R13

Write back

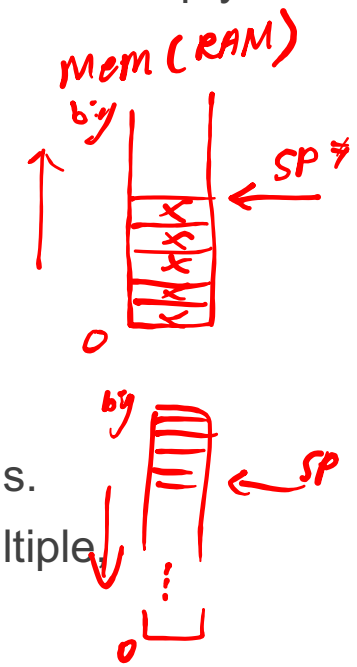
e.g. R0-R5

- Advantages of using LDM / STM over LDR / STR for multiple data transfers:
  - Reduced code size.
  - Faster execution – only one instruction fetched from the memory.
- With LDM / STM:
  - Stack pointer (R13) should not be used in *register_list*.
  - Cannot have LR and PC in the *register_list* at the same time.

Stack: different types

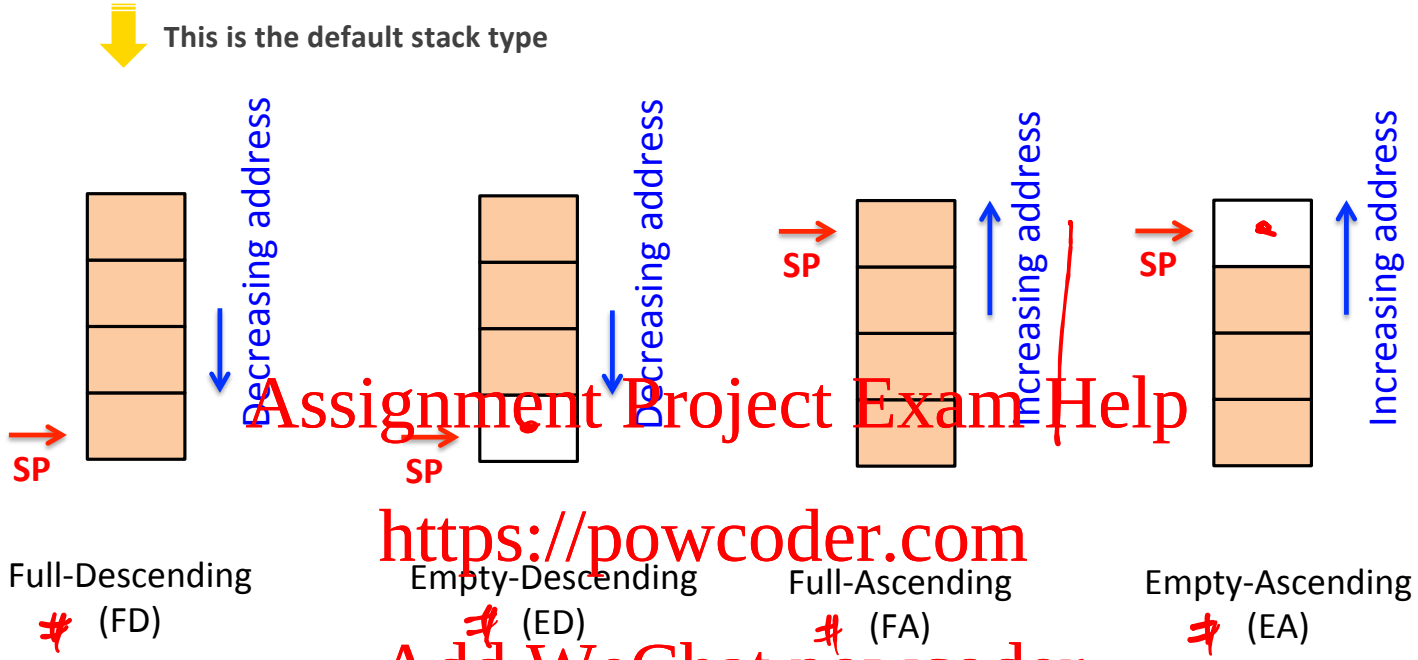
- Stack types:
  - Descending / Ascending: whether the stack grows downwards (address decreases as stack grows) or upwards (address increases as stack grows).
  - Full / Empty: whether the stack pointer points to the last filled element or next empty space on the stack.
- Four stack types:
  1. Full Descending (FD)
  2. Full Ascending (FA)
  3. Empty Descending (ED)
  4. Empty Ascending (EA)

This is the default stack type



- These suffixes can be used as addressing modes in LDM / STM instructions.
  - **LDMFD** (load multiple, full descending) is equivalent to **LDMIA** (load multiple, increment after).

Stack: different types

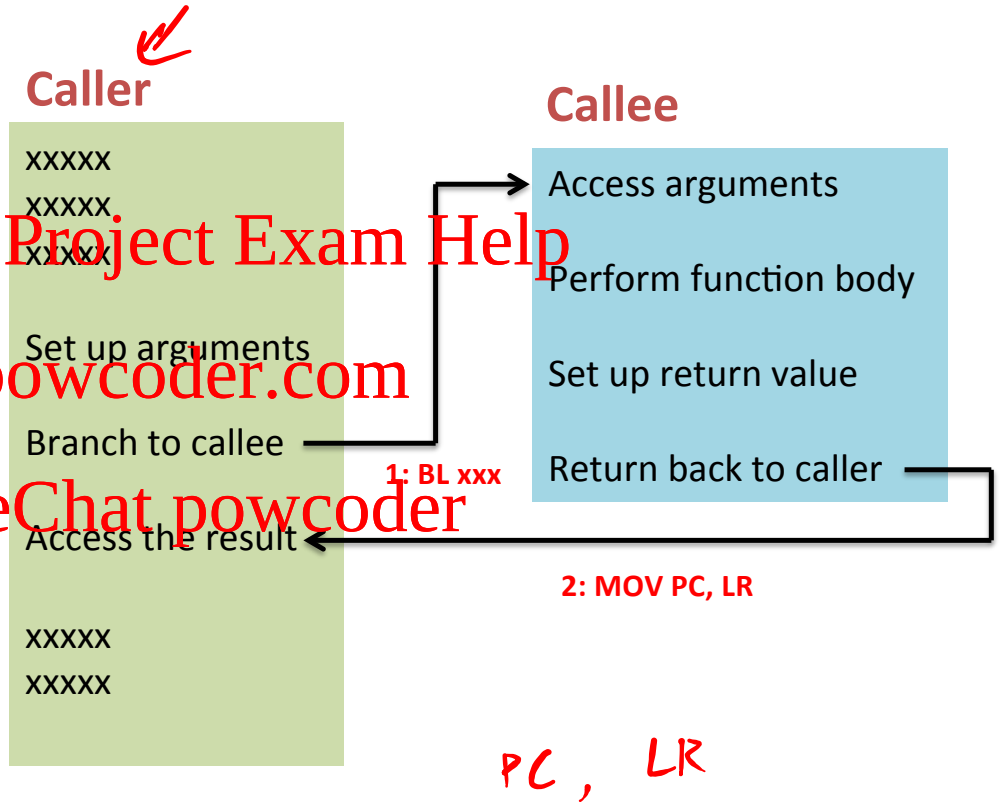


Stack Type	Push	Pop
Full descending	STMFD (STMDB)	LDMFD (LDMIA)
Full ascending	STMFA (STMIB)	LDMFA (LDMDA)
Empty descending	STMED (STMDA)	LDMED (LDMIB)
Empty ascending	STMEA (STMIA)	LDMEA (LDMDB)

- Pick one type and use (push / pop) operations consistently.

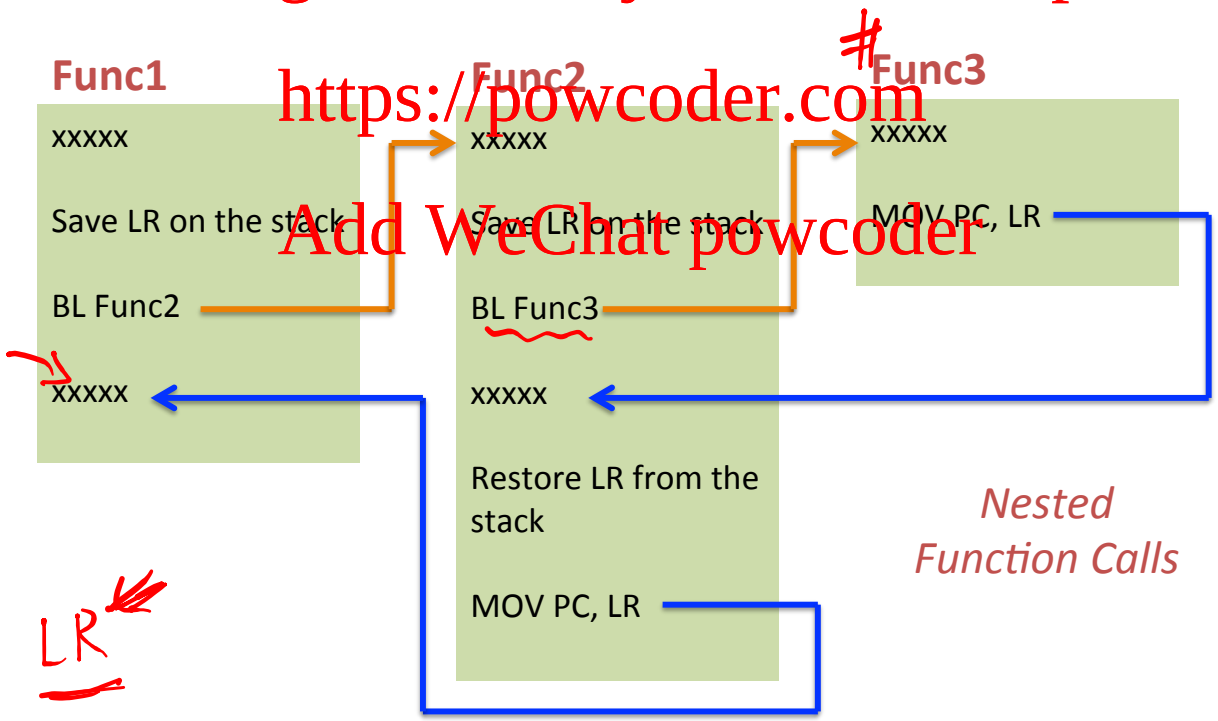
Function call basics

- Two parties: Caller, Callee.
- Functions have return values, subroutines do not.
- Caller has to set up the arguments.
- Callee has to set up the return value.
- Function call:
  - Achieved using BL (branch & link).
  - **BL** automatically saves the current program counter (PC) in the link register (LR).
  - Returning is achieved by **MOV** `MOV PC, LR` which restores PC with the saved LR value.



Nested function calls

- More complicated situation: Func1 → Func2, then Func2 → Func3.
- If function calls another function:
  - Everything before applies.
  - Plus **saving LR on the stack** in Func2, before calling Func3, so that we can return up the calling chain.



Function calls & register conflicts

- Only 15 registers sharable between caller and callee.
- Caller needs registers to pass arguments. Callee needs registers to return values.
- Callee should not corrupt any registers that caller might use after the function call.
- To help resolve these conflicts, register groups are defined with usage rules:
  - Scratch / argument: **A1 – A4**
  - Local variables: **V1 – V8**

Mode					
User/System	Supervisor	Abort	Undefined	Interrupt -IRQ	Fast Interrupt - FIQ
R0 , A1	R0	R0	R0	R0	R0
R1, A2	R1	R1	R1	R1	R1
R2, A3	R2	R2	R2	R2	R2
R3, A4	R3	R3	R3	R3	R3
R4, V1	R4	R4	R4	R4	R4
R5, V2	R5	R5	R5	R5	R5
R6, V3	R6	R6	R6	R6	R6
R7,V4	R7	R7	R7	R7	R7
R8,V5	R8	R8	R8	R8	R8 _FIQ
R9, V6	R9	R9	R9	R9	R9_ FIQ
R10,V7	R10	R10	R10	R10	R10_ FIQ
R11, V8	R11	R11	R11	R11	R11_ FIQ
R12, ip	R12	R12	R12	R12	R12_ FIQ
R13, sp	R13_ SVC	R13_ ABORT	R13 _UNDEF	R13_ IRQ	R13_ FIQ
R14, lr	R14 _SVC	R14_ ABORT	R14 _UNDEF	R14_ IRQ	R14_ FIQ

Function calls & register conflicts

- Caller
  - Optionally save A1 – A4 on stack if it wants to use them after the function call (callee might corrupt them). These are **caller-save registers**.
  - Uses A1 – A4 to pass arguments to callee.
  - Additional arguments are passed via the stack.
  - Saves LR before BL. Because LR holds the **return address,** and BL modifies LR.
- Callee
  - Uses A1 to transfer the **return value**.
  - Save V1 – V8 on stack if it uses them, then restore before returning (Caller assumes these are unchanged on return). These are **callee-save registers**.
  - Return to caller by performing MOV PC, LR.

Function calls & register conflicts

Which registers may have changed after the BL instruction?

```
....
....
BL func2
....
....
```


- A1 – A4 can be changed.
  Callee *func2* can modify A1 – A4 at any time as A1 is used to transfer the return value.
- V1 – V8 remain unchanged.
  If callee *func2* uses V1 – V8, it has to save them on stack and restore the original content prior to returning.
- SP remains unchanged.
  Callee's stack frame is removed when returning.
- LR has changed.
- IP (R12) can be changed.
  This is a scratch register.

AAPCS register conventions

- The foregoing convention is specified by the **ARM Architecture Procedure Call Standard**.

Register name	Software name	Usage
R0 – R3	A1 – A4	First 4 int arguments Scratch registers Function results
R4 – R11	V1 – V8	Local variables
R9	SB	Static variable base
R10	SL	Stack limit
R11	FP	Frame pointer
R12	IP	Intra-procedure call scratch register
R13	SP	Stack pointer
R14	LR	Return address
R15	PC	Program counter

static int x = 0;

Memory is finite, stack size is limited

Pointer to local variables on the stack

Not saved by caller nor restored by callee

Blue: software convention (programmer's responsibility)

Red: Hardware assisted

Setting up function calls

- Generalizing, every function must perform 3 tasks:
  1. Create a **stack frame** to
     - » backup registers that conflict with the caller (i.e. used by both caller and callee)
     - » Save the LR, if the current function calls another function
     - » pass arguments (if A1 – A4 insufficient)
  2. Perform operations (function body).
  3. Remove the stack frame and return to caller.

Stack operations for function calls

- Example: using a full-descending (FD) stack, save V1, V2 and LR to memory.
- Creating stack frame
  - Version 1

```
SUB SP, SP, #12     ; reserve space for 3 registers
STR LR, [SP, #8]    ; save LR
STR V2, [SP, #4]    ; save V2
STR V1, [SP, #0]    ; save V1
```

  - Version 2 – using store multiple decrement-before

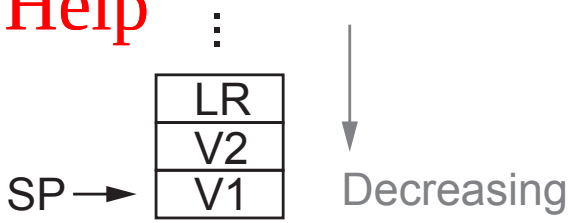
```
STMDB SP!, {V1,V2,LR}
```

- Removing stack frame
  - Version 1

```
LDR V1, [SP, #0]    ; restore V1
LDR V2, [SP, #4]    ; restore V2
LDR LR, [SP, #8]    ; restore LR
ADD SP, SP, #12     ; remove space
```

  - Version 2 – using load multiple increment-after

```
LDMIA SP!, {V1, V2, LR}
```



Stack operations for function calls

- Example: using a full-descending (FD) stack, save V1, V2 and LR to memory.
- Creating stack frame
  - Version 3

```
STR LR, [SP, #-4]!  ; save LR
STR V2, [SP, #-4]!  ; save V2
STR V1, [SP, #-4]!  ; save V1
```

Pre-indexed decrement of SP…

Subtract address by 4 then write them

- Removing stack frame
  - Version 3

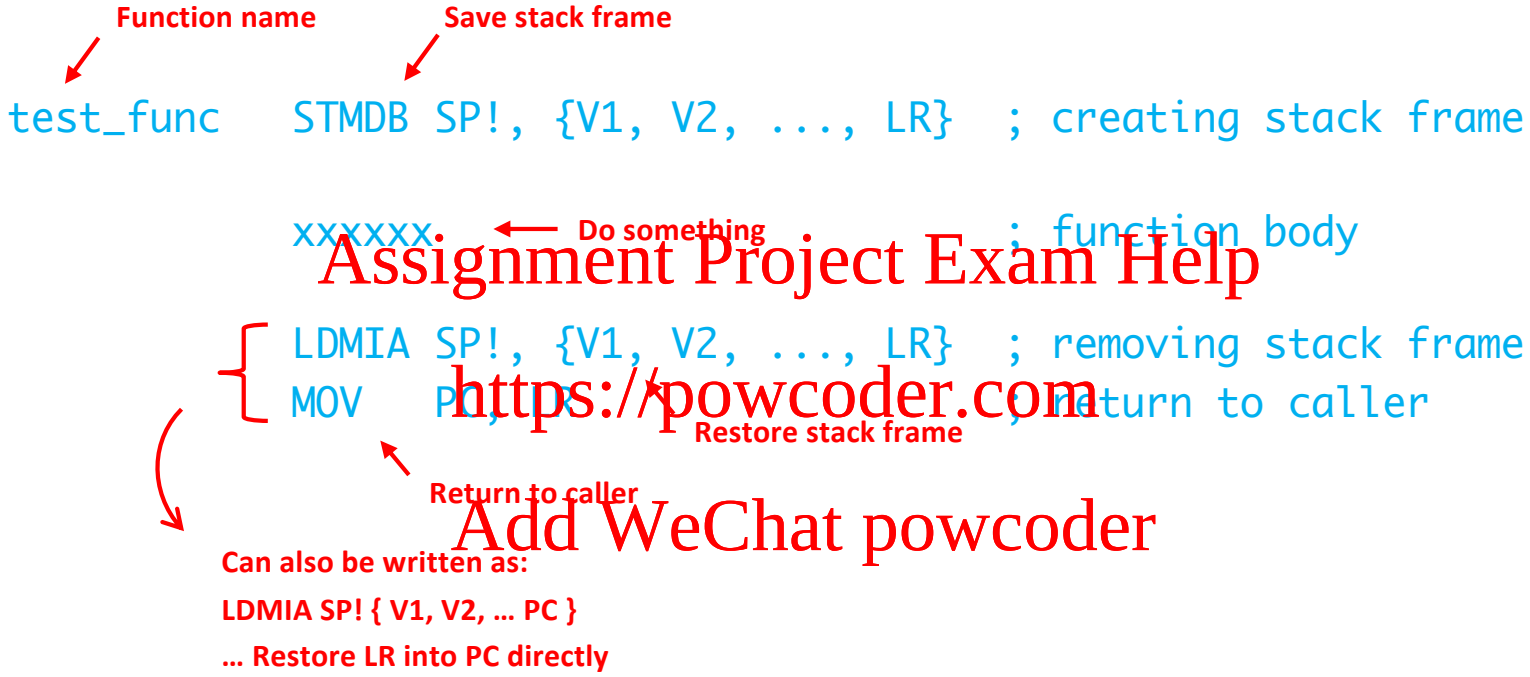
```
LDR V1, [SP], #4    ; restore V1
LDR V2, [SP], #4    ; restore V2
LDR LR, [SP], #4    ; restore LR
```

Post-indexed increment of SP…

Read memory then add 4 to address

Basic assembly structure of a function

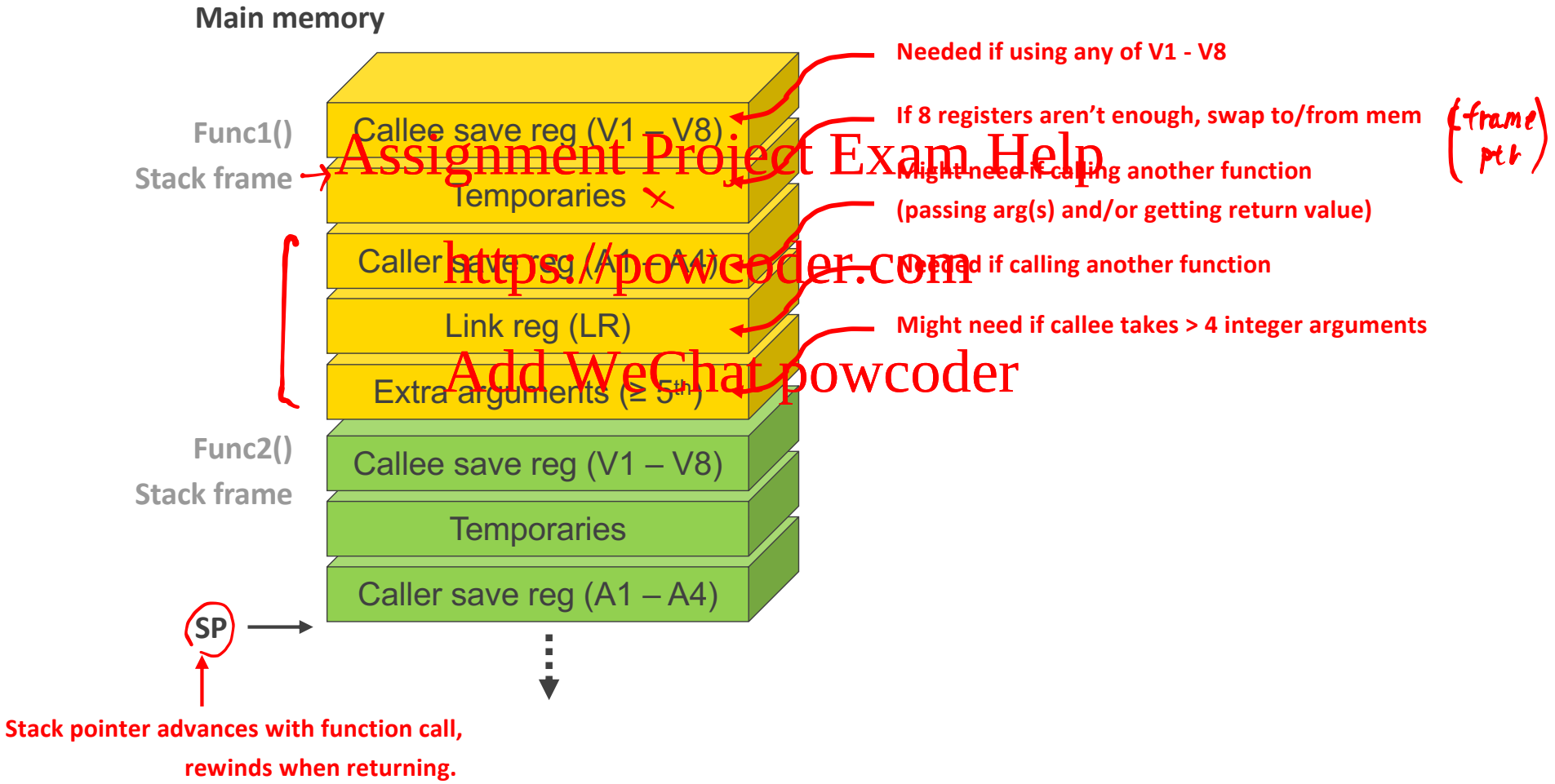
- Template for function:



- Expand this structure for more complicated cases
  - When calling additional functions (perhaps restoring A1 ~ A4 after callee returns).
  - Passing more than 4 integer arguments (put args 5th onwards on the stack).

Summary of stack frame

- Func1() called Func2(), and processor is executing Func2(). Stack frame might look like the following, in the most general case:



C to assembly example

- Translate the following C code to assembly

```
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```

- ARM assembly

```
sumSquare       STR LR, [SP, #-4]!  ; save return addr
                STR A2, [SP, #-4]!  ; save A2 (y)... used locally
                MOV A2, A1          ; setting up mult(x,x)
                BL  mult            ; call mult func
                LDR A2, [SP], #4    ; restore y
                ADD A1, A1, A2      ; mult() + y
                LDR LR, [SP], #4    ; get return addr
                MOV PC, LR
```


- Note: arguments passed via registers (A1 and A2).

Assembly example 1

- Translate this C code into assembly:

```
int Doh(int i, int j, int k, int l) {
    return i + j + l;
}
```

Four arguments, A1 – A4

Return in A1

- Assembly:

```
Doh         ADD A1, A1, A2
            ADD A1, A1, A4
            MOV PC, LR
```


- Easiest case. Everything can be done using Ax registers and no further function call.

Assembly example 2

- Translate this C code into assembly:

```
int Doh(int i, int j, int k, int m, char c, int n) {
    return i + j + n;
}
```

Return in A1

Six arguments: A1 – A4, <SP>, <SP+4>

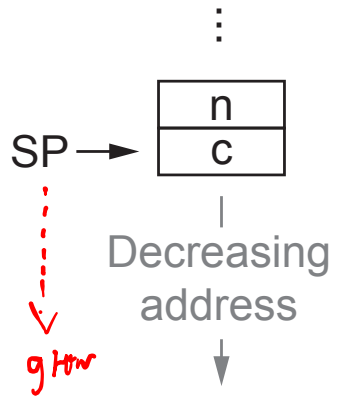
stack

- Assembly:

```
Doh     LDR IP, [SP, #4]    ; load n
        ADD A1, A1, A2      ; i+j
        ADD A1, A1, IP      ; (i+j)+n
        MOV PC, LR
```


Note LDR op. Reading the argument from caller's frame.

- Like example 1 but needs to pass 5th and 6th arguments via stack.



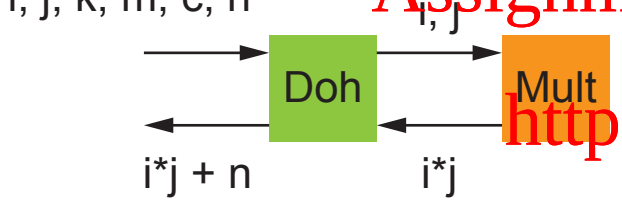
Assembly example 3

- Translate this C code into assembly:

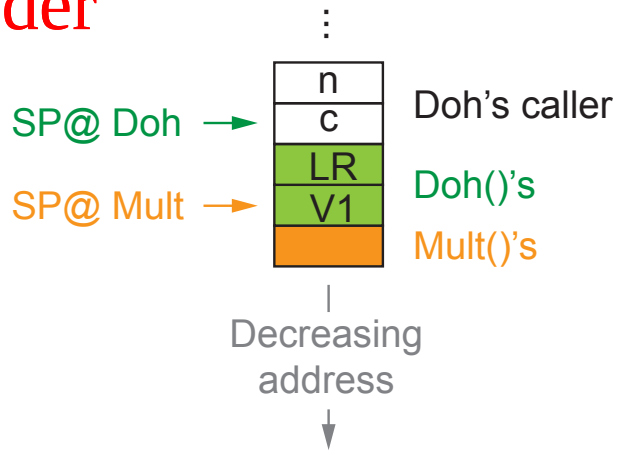
```
int Doh(int i, int j, int k, int m, int c, int n) {
    return Mult(i, j) + n;
}
```

Six arguments: A1 – A4, <SP>, <SP+4>

Return in A1



- Nested function call. *Doh()* needs to:
  1. Save LR on stack
  2. Save register(s) used locally
  3. Load the 6th arg *n* from stack
  4. Do work, including calling *Mult()*
  5. Reverse steps 2 – 1.
  6. Return with result in A1.



Assembly example 3

- Translate this C code into assembly:

```
int Doh(int i, int j, int k, int m, int c, int n) {
    return Mult(i, j) + n;
}
```

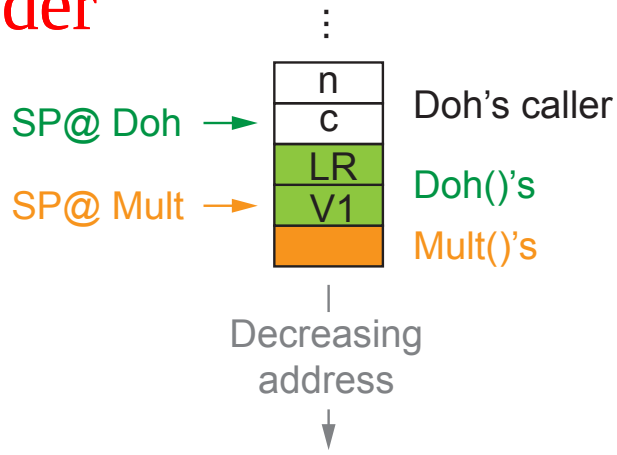
Six arguments: A1 – A4, <SP>, <SP+1>

Return in A1


```
Doh     SUB SP, SP, #8
        STR LR, [SP, #4]  ; store LR
        STR V1, [SP, #0]  ; store V1

        LDR V1, [SP, #12] ; V1 := arg #6 (n)
        BL  Mult
        ADD A1, A1, V1

        LDR V1, [SP, #0]
        LDR LR, [SP, #4]
        ADD SP, SP, #8
        MOV PC, LR
```



Assembly example 3

- Translate this C code into assembly:

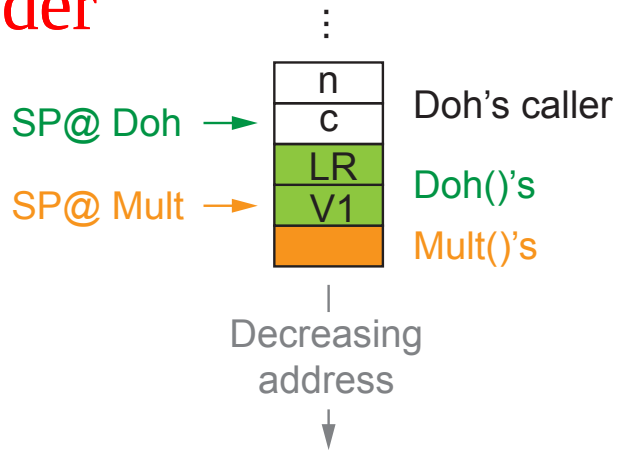
```
int Doh(int i, int j, int k, int m, int c, int n) {
    return Mult(i, j) + n;
}
```

Six arguments: A1 – A4, <SP>, <SP+1>

Return in A1

- Same thing but shorter

```
Doh     STMFD SP!, {LR, V1}
        LDR V1, [SP, #12]  ; V1 := arg #6 (n)
        BL  Mult
        ADD A1, A1, V1
        LDMFD SP!, {LR, V1}
        MOV PC, LR
```

Assembly example 4

- Translate this C code into assembly:

```
int main() {
    int i, j, k, m;
    i = mult(j, k);
    m = mult(i, i);
    ...
    return 0;
}

int mult(int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}
```

Assembly example 4

- Translate this C code into assembly:

```
int main() {
    int i, j, k, m;
    i = mult(j, k);
    m = mult(i, i);
    ...
    return 0;
}

int mult(int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}
```

- For *mult()*:
  - Does not call another function. Do not need to save LR.
  - A1–A4 suffices for body of *mult()*… no register needs to be saved.
  - Return value placed in A1.
- For *main()*:
  - Returns to the operating system. So LR needs to be saved before calling *mult()*.
  - Set up arguments for two *mult()* calls.

Assembly example 4

- Assembly code for *mult()*:

```
mult        MOV  A3, #0          ; prod=0
mult_loop   CMP  A2, #0          ; mlier > 0?
            BEQ  mult_fin
            ADD  A3, A3, A1      ; prod += mcand
            SUB  A2, A2, #1      ; mlier -= 1
            B    mult_loop
mult_fin    MOV  A1, A3          ; a1 = prod
            MOV  PC, LR
```

Assembly example 4

- Assembly code for *main()*:

```
main    STR LR, [SP, #-4]!  ; store ret addr

        ; i=V1 j=V2 k=V3 m=V4
        MOV A1, V2          ; arg1 = j
        MOV A2, V3          ; arg2 = k
        BL  mult
        MOV V1, A1          ; i = mult()
        MOV A1, V1          ; arg1 = i
        MOV A2, V1          ; arg2 = i
        BL  mult
        MOV V4, A1          ; m = mult()
        ...
        MOV A1, #0          ; main() ret value

        LDR LR, [SP], #4    ; restore ret addr
        MOV PC, LR
```

Assembly example 5

- Translate this C code into assembly:

```
int main() {
    int a = 42;
    printf("The meaning of life is %d\n", a);
    return 0;
}
```

... Two args

A1

A2 (pointer to string)

A2 (integer)

- Assembly code:


```
main        STR  LR, [SP, #-4]!  ; save LR

            LDR  A1, =str        ; set up args
            MOV  A2, #42
            BL   printf
            MOV  A1, #0          ; main() ret value

            LDR  LR, [SP], #4    ; restore LR
            MOV  PC, LR

str DCB "The meaning of life is %d\n", 0
```

null term.

This week

- Stack
- Function call
- Stack operations
- ARM architecture procedure call standard (AAPCS)
- Function call examples

In Moodle:

- **Start working on Lab**
  (Due: end of your 3-hr lab)
- **Start doing Week 5 exercise**

References

[1] William Hohl, ARM Assembly Language: Fundamentals and Techniques, CRC Press, 2015 (2nd Edition).

[2] ARM Architecture Reference Manual.