Queen Mary
University of London
Science and Engineering

School of Electronic Engineering and Computer Science
QMUL-BUPT Joint Programme

EBU6475 Microprocessor System Design
EBU5476 Microprocessors for Embedded Computing

C as Implemented in Assembly Language

References:
Chapter 5.6, The Definitive Guide to ARM®;
Chapter 5, Embedded Systems Fundamentals

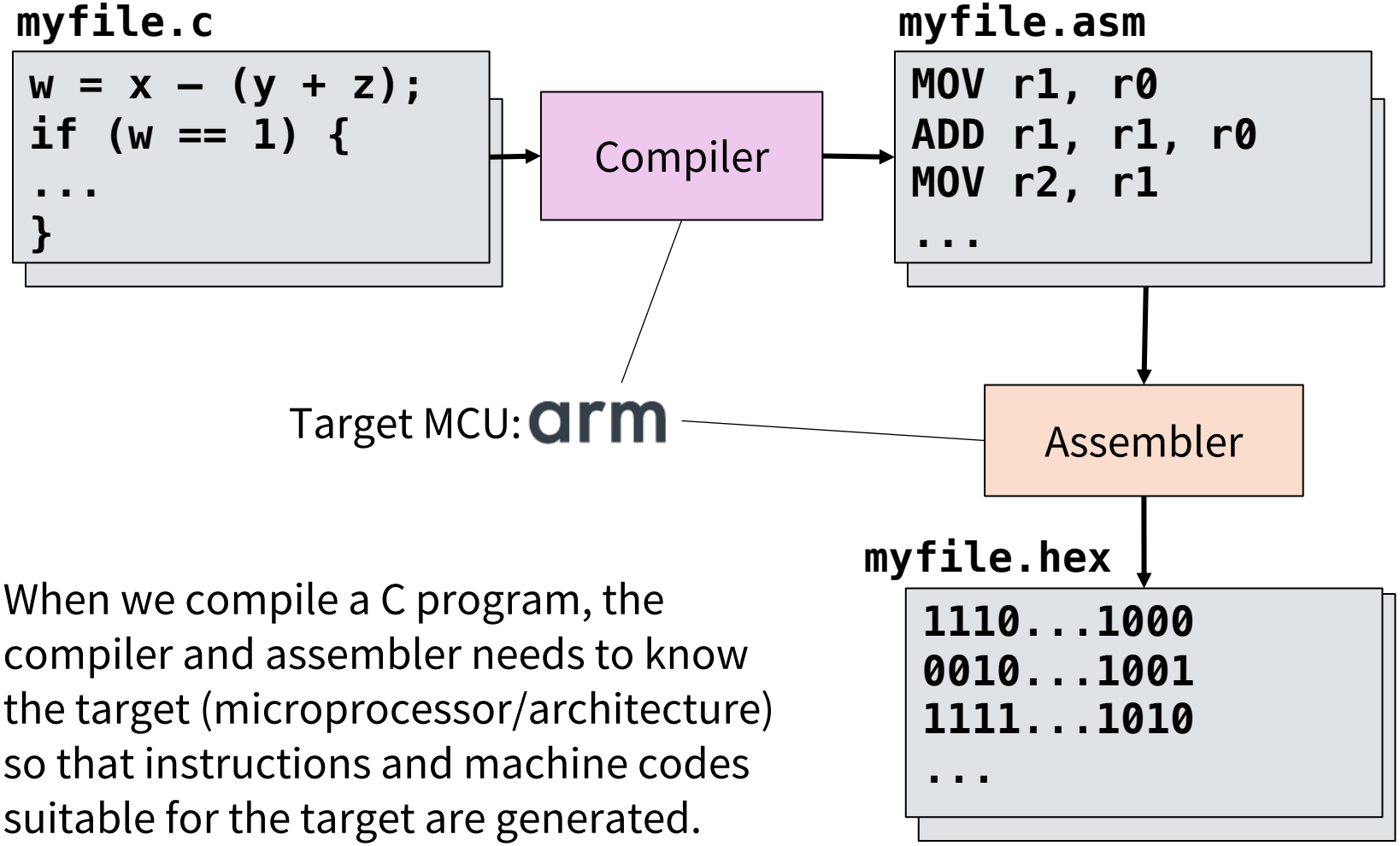
Last updated: 13 March 2022

arm
University Program Education Kits

Overview

- We program in C for convenience.
- There are no MCUs which execute C, only machine code.
- So we compile the C to assembly code, a human-readable representation of machine code.
- We need to know what the assembly code implementing the C looks like …
  - To use the processor efficiently
  - To analyze the code with precision
  - To find performance and other problems
- An overview of what C gets compiled to
  - control flow, C start-up module, subroutines calls, stacks, data classes and layout, pointers, ets.

Compilation and Assembling



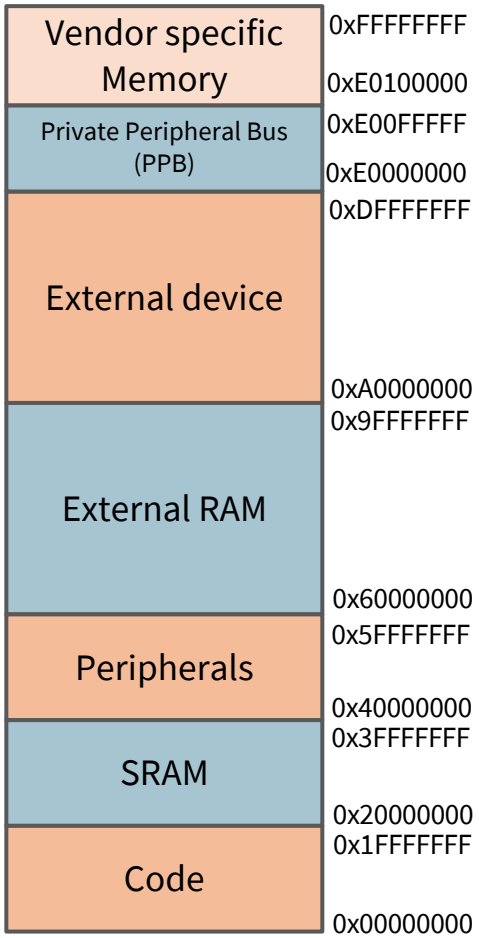
When we compile a C program, the compiler and assembler needs to know the target (microprocessor/architecture) so that instructions and machine codes suitable for the target are generated.

ARM Memory Map

32-bit memory address gives a 4 GB memory space, separated into logical regions for different purposes.
e.g. 0x00000000 – 0x1FFFFFFF (512 MB) is recommended for internal code.

r15 | PC

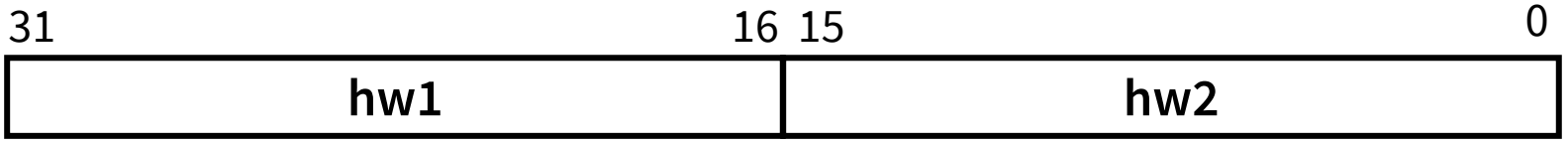
r15, the Program counter (PC) stores the beginning address of the current instruction.



ARM Thumb-2 Instruction

In Thumb-2, most instructions are encoded as two bytes, to save (code) memory space. But there are cases where instructions cannot be fitted in 2 bytes, then 1 word = 4 bytes will be used.

Most cases: 2 bytes (half word); otherwise: 4 bytes (a word)



The *first halfword* (hw1) determines the instruction length and functionality. If the processor decodes the instruction as 32-bit long, then the processor fetches the *second halfword* (hw2) of the instruction from the instruction address plus two.

先fetch一个hw(hw1, instruction长度, 功能)，如果长度是32bits，那就再fetch一个hw(hw2).

Details of Assembling

```
MOV r0, #10
LDR r2, [r1]
ADD r0, r0, r2
```

```
08000190: F0 4F 00 0A
08000194: 68 11
08000196: 44 10
```

First instruction requires 4 bytes to encode the immediate data 10 (0x0A) within.
The second and third instructions can be encoded using just 2 bytes (half words).
The machine code is just a list of assembled bytes representing the list of instructions.

```
… F0 4F 00 0A 68 11 44 10 …
```

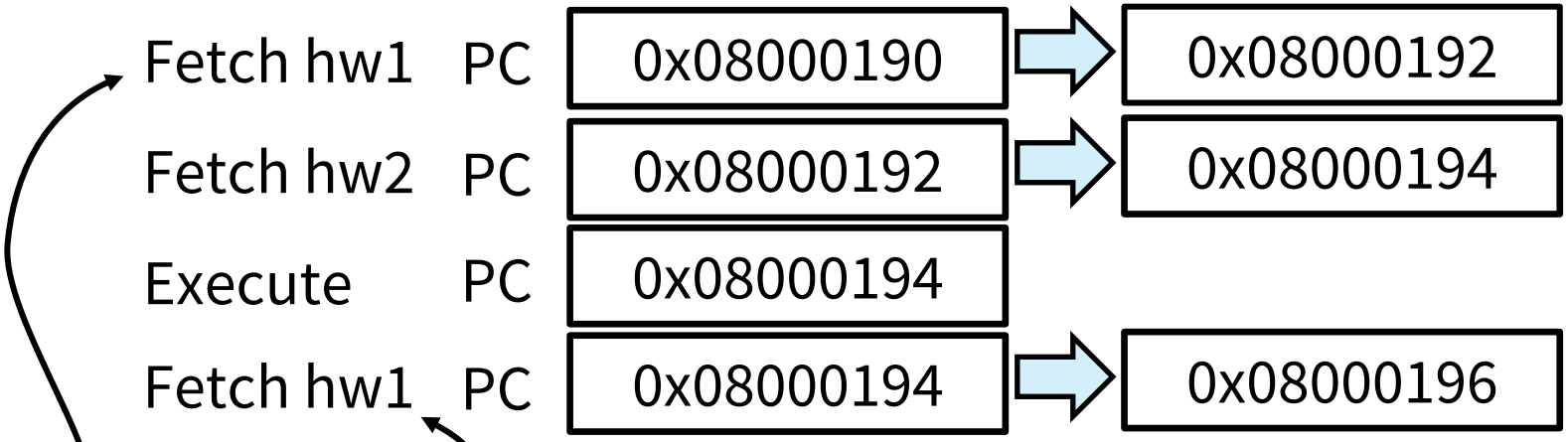
Part of the Assembled program (*.hex)

Program Counter and Fetch

PC contains the address of the current instruction that is going to be fetched.

And the value of PC is incremented by 2 whenever a half word has been fetched during execution.

PC+2+2... sequential execution



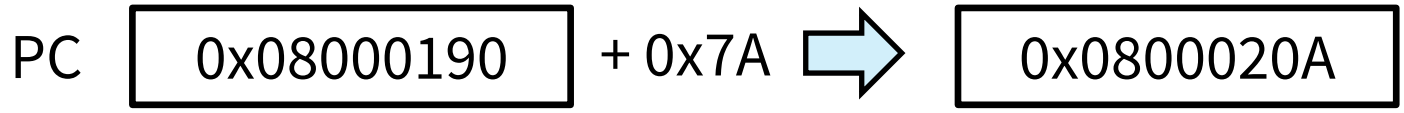

```
MOV r0, #10
LDR r2, [r1]
ADD r0, r0, r2
```

```
08000190: F0 4F 00 0A
08000194: 68 11
08000196: 44 10
```

Offset and Branch Instruction

- By changing the value of PC, we branch(jump) to another instruction.
- The difference (in bytes) of the new address in PC from the current address is called the offset of the jump.

offset可以是 positive or negtive




```
08000190: B label1
08000194:
...

label1
0800020A: LDR r0, [r1]
0800020C:
```

How to store the offset?

- Offset is encoded within the instruction.
- When the offset is too large, then 32-bit instruction has to be used.

Branch Instructions

conditional

unconditional

Unconditional jumps are always taken.
Programmer can choose the suffix for conditional branches.

Mnemonic	Operation
B <label>	unconditional branch
B<cc> <label>	conditional branch on suffix <cc>
BL <label>	branch with return address stored in link register (LR, r14)
BX Rm	branch indirect using register Rm
BLX Rm	branch indirect using register Rm with return addressed stored in LR

BL, BX and BLX are important for calling functions and returning, and will be discussed in great details later.

Condition Code Suffixes

Instruction **CMP** can be used to compare two operands and set the flags in APSR for branches.

```
CMP R0, R1      ; compare R0 and R1,  update APSR
CMP R2, #100    ; compare R2 and 100, update APSR
```

Suffix	Meaning	Suffix	Meaning
EQ	Equal	NE	Not equal
MI	Negative	PL	Positive or zero
VS	Overflow	VC	No overflow
HS	Higher or same, unsigned	LO	Lower, unsigned
HI	Higher, unsigned	LS	Lower or same, unsiged
GE	Greater than or equal, signed	LT	Less than, signed
GT	Greater than, signed	LE	Less than or equal, signed

data movement: MOV, LDR, STR
arithmetic: ADD, SUB, MUL, DIV
branch instruction: B, CMP

Control Flow

Creating if-else, switch-case and loops in Assembly

Control Flow: Conditionals and Loops

- How does the compiler implement conditionals and loops?

```
if (x){
    y++;
} else {
    y--;
}
```

```
switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
}
```

```
for (i = 0; i < 10; i++){
    x += i;
}
```

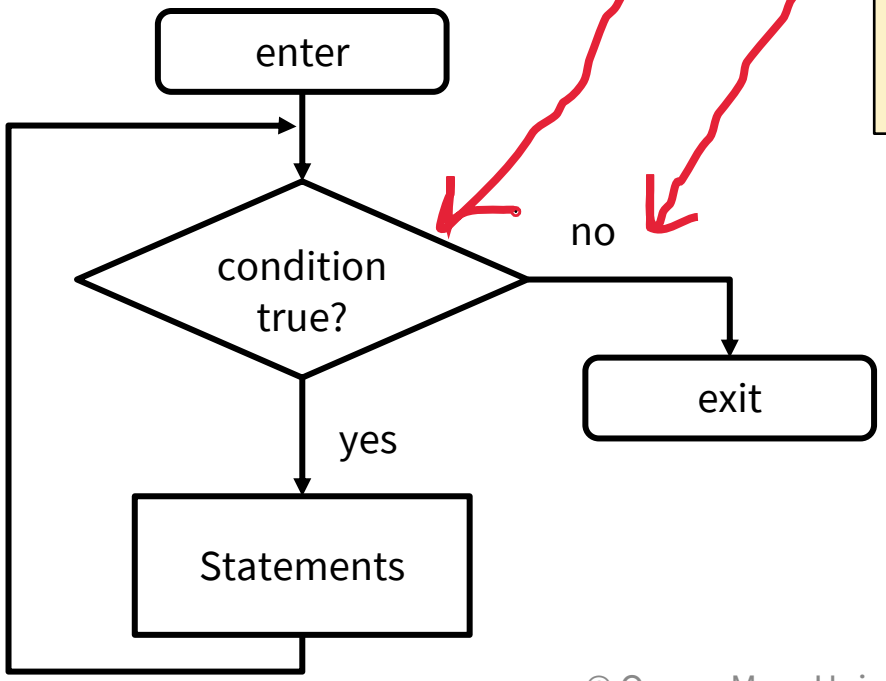
```
while (x<10) {
    x = x + 1;
}
```

```
do {
    x += 2;
} while (x < 20);
```

While Loop: Template

```
// C or Java style
while (condition) {
    stat1;
    stat2;
}
```

```
    ; ARM asm
    {COND_SETUP …}
COND
    {COND_EVAL …}
    B<cond> EXIT
    {INSTS1 …}
    {INSTS2 …}
    B COND
EXIT
```



A single branch can effectively form a while loop. In assembly, several instructions may be required to setup and test the condition.

While Loop: Example

```
x = n - 1; // x@0x20000000, n@0x20000004
while (n % x != 0) {
   x--;
}
```

What does this program trying to do?

```
    LDR  R2, =0x20000004
    LDR  R0, [R2]              ; R0 = n
    SUBS R1, R0, #1            ; R1 = x
WHILE_BEGIN
    UDIV R2, R0, R1            ; R2 = n / x
    MUL  R3, R2, R1            ; R3 = R2 * x
    CMP  R0, R3                ; n == (n / x) * x
    BEQ  WHILE_END
    SUBS R1, R1, #1            ; x--
    B    WHILE_BEGIN           ; loop back
WHILE_END
    LDR  R2, =0x20000000       ; write back to mem
    STR  R1, [R2]
```

modulus operation

If-else: Template

```
// C or Java style
if (condition) {
    stat1;
}
else {
    stat2;
}
```

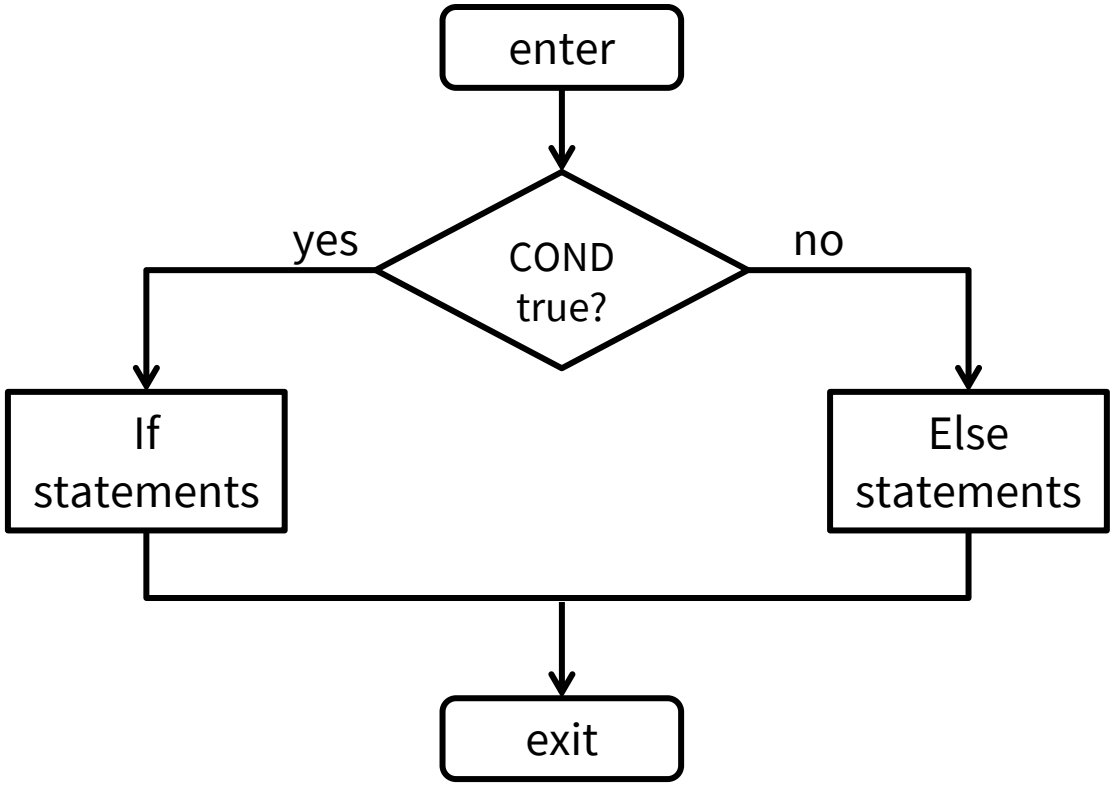
```
    ; ARM asm
IFBEGIN
    {COND_SETUP …}
    {COND_EVAL …}
IFPART
    B<cond> ELSEPART
    {INSTS1 …}
    B IFEND *
ELSEPART
    {INSTS2 …}
IFEND
```

Remember branch unconditionally over the else-part by the end of the if-part.

You can choose to swap the if-part and else-part by inverting the conditions (e.g. EQ to NE, or vice versa)

If-else: Template (Cont')

```
     ; ARM asm
IFBEGIN
     {COND_SETUP …}
     {COND_EVAL …}
IFPART
     B<cond> ELSEPART
     {INSTS1 …}
     B IFEND
ELSEPART
     {INSTS2 …}
IFEND
```



If-else: Example

```
// x, y, z @ 0x20000000, 04, 08
if (x >= y) {
    z = x;
} else {
    z = y;
}
```

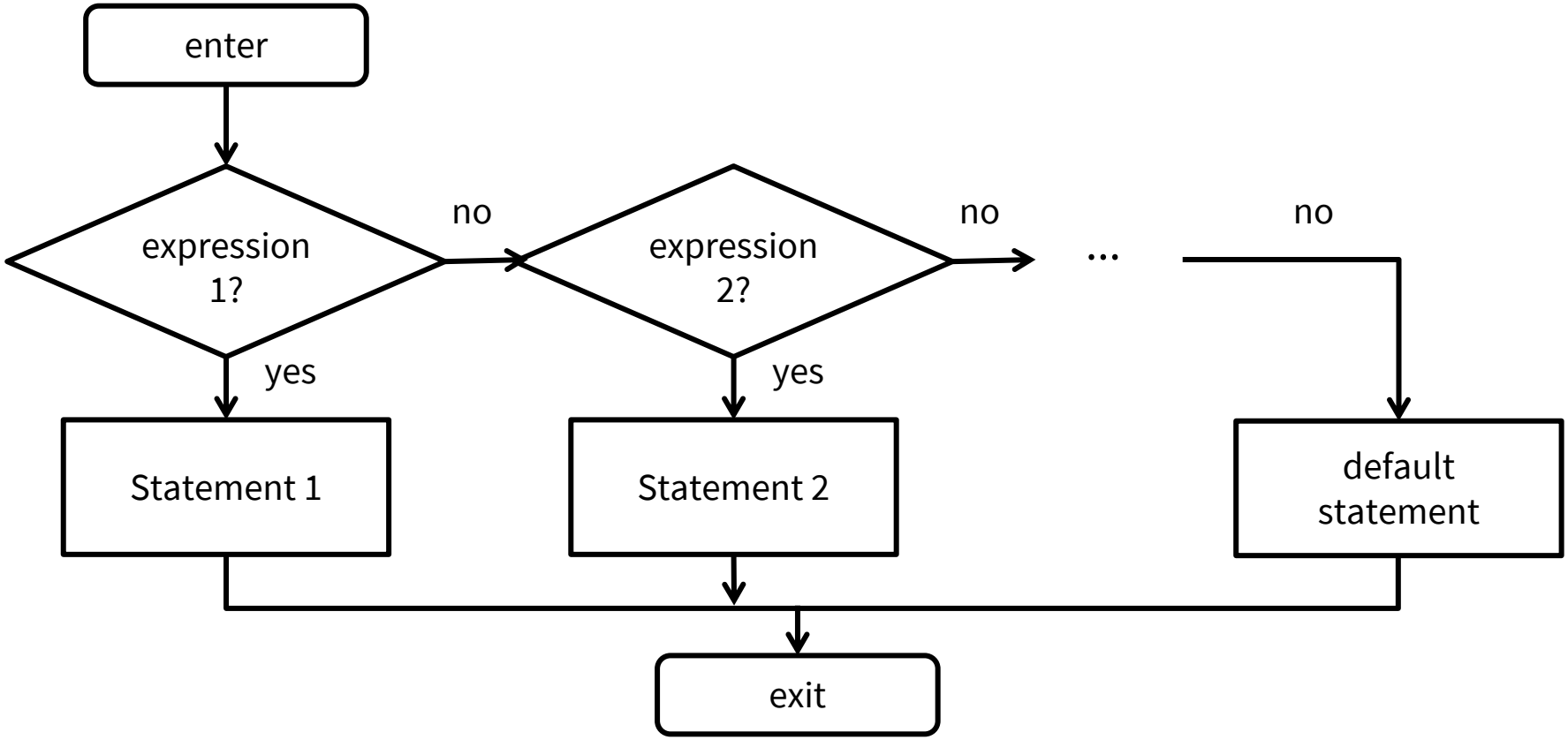
What does this program trying to do?

```
    LDR R3, =0x20000000
    LDR R0, [R3], #4
    LDR R1, [R3], #4
IFBEGIN
    CMP R0, R1
    BLT ELSEPART ; BLT: Less Than
IFPART
    STR R0, [R3]
    B IFEND
ELSEPART
    STR R1, [R3]
IFEND
```

Sometimes you need to insert more jumps when the condition checking is more complicated. But the principle is the same.

Case: Template

Case structure chooses one statement from many, which is effectively an iterated if-else structures, as shown.



Case: Template (Cont')

```
// C or Java style
switch (expression){
  case (choice1): stat1; break;
  case (choice2): stat2; break;
  ...
  default: statN; break;
}
```

```
     ; ARM asm
SWITCH_BEGIN
     {COND_SETUP …}
     {COND_EVAL …}
CHOICE1
     B<cond> CHOICE2
     {INST1 …}
     B SWITCH_END
CHOICE2
     B<cond> CHOICE3
     {INST2 …}
     B SWITCH_END
…
SWITCH_END
…
```

A case can be easily translated into assembly.

Control Flow: Exercises

Try to translate any of following code snippets into ARM assembly.

```
if (x){
    y++;
} else {
    y--;
}
```

```
switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
}
```

```
for (i = 0; i < 10; i++){
    x += i;
}
```

```
while (x < 10) {
    x = x + 1;
}
```

```
do {
    x += 2;
} while (x < 20);
```

Stack and Functions

Stack Memory in ARM

Cortex-M processors use a stack memory model called **full-descending stack**: stack向下生长

- When started, SP is set to the end of stack memory space.
- PUSH operation: SP = SP – 4, then store the value @SP
- POP operation: read the value @SP then SP = SP + 4

Mnemonic	Operation
PUSH reglist	Push register(s) to stack
POP reglist	Pop from stack to register(s)

to protect register data

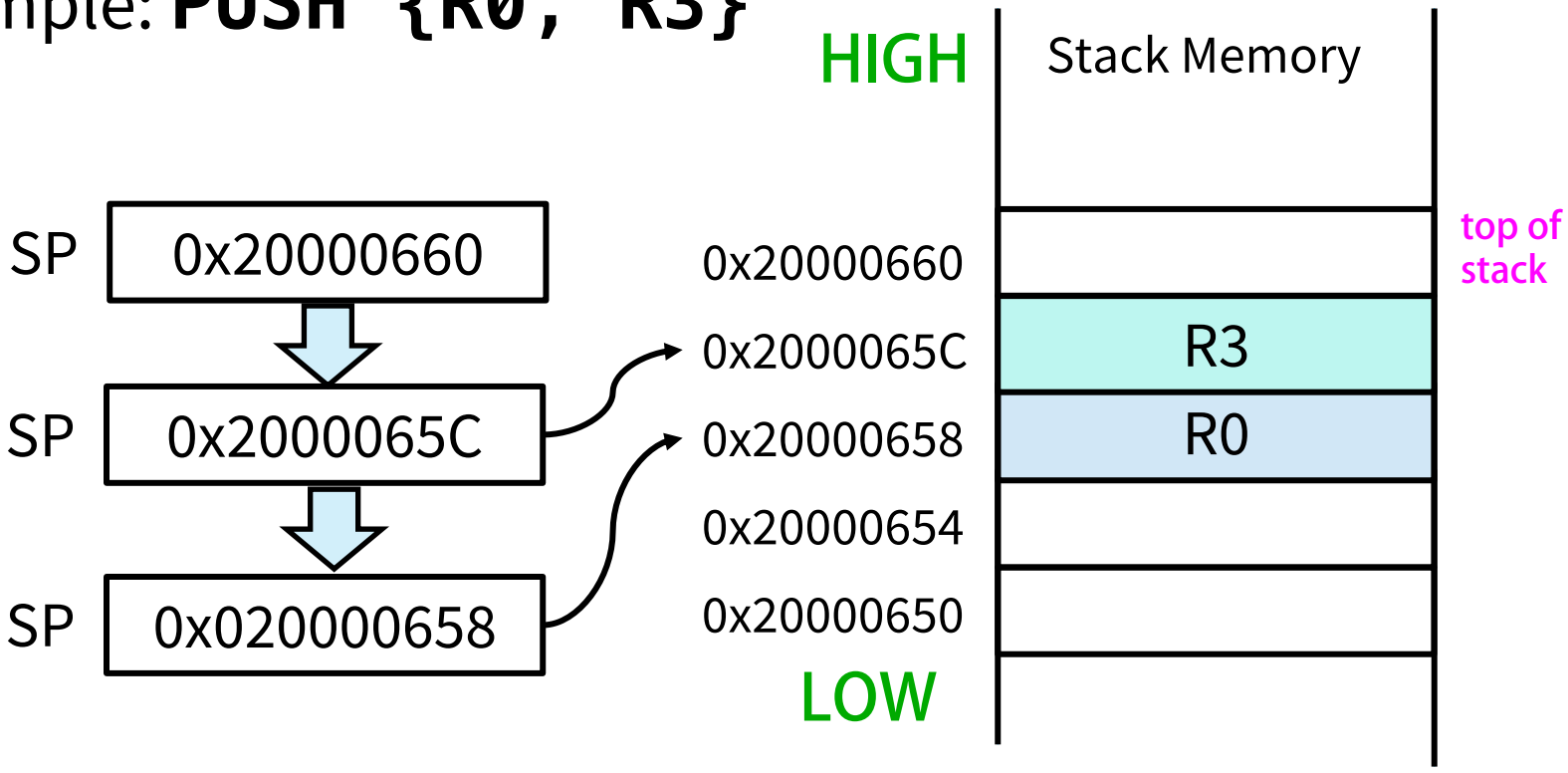
The list of registers (reglist) is specified with braces ({ }) in UAL.

```
PUSH {R0, R4–R7} ; Push r0, r4, r5, r6, r7
POP  {R2–R3, R5} ; Pop to r2, r3, r5
```

Push and pop transfers are at least 1 word of data.

Stack: Push - Example

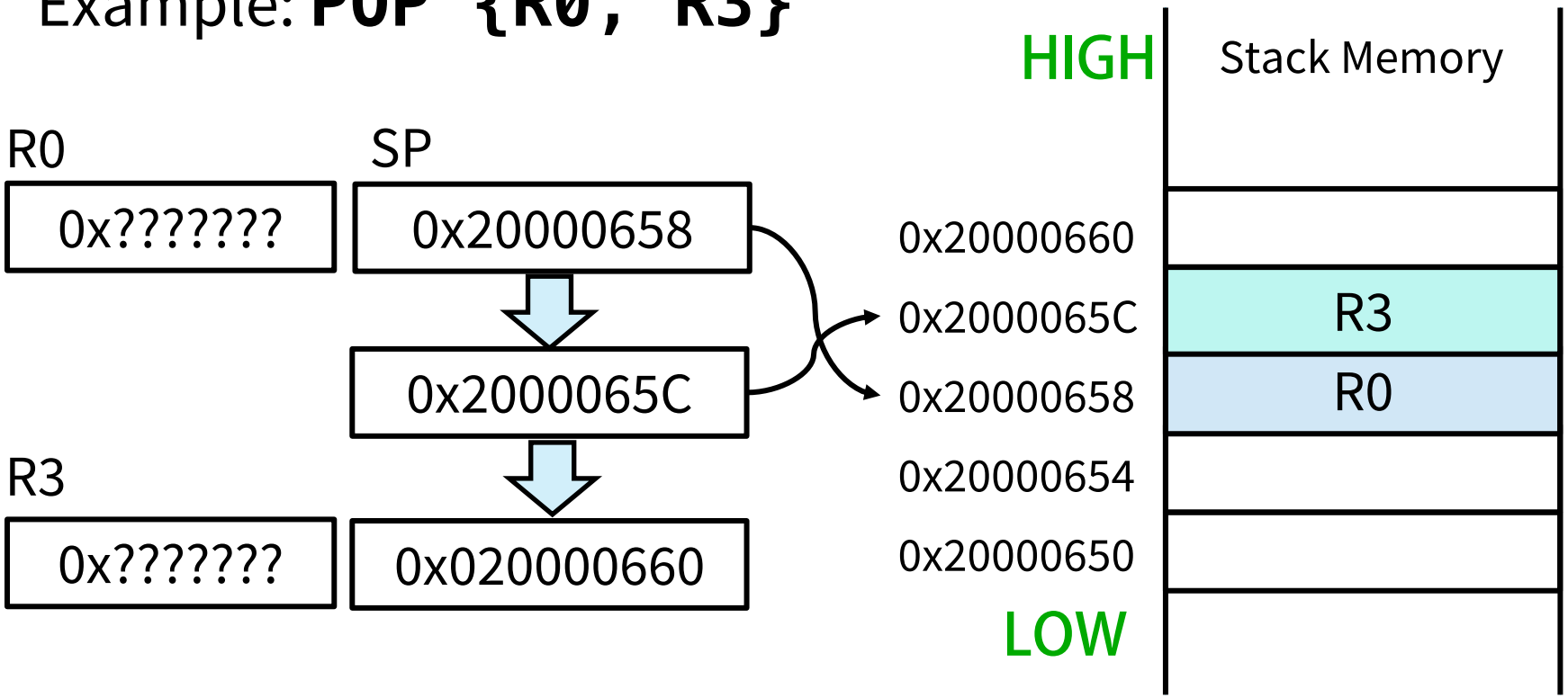
Example: **`PUSH {R0, R3}`**



The lower numbered the register is, the lower memory address in the stack. （我们这样规定）

Stack: Pop - Example

Example: **`POP {R0, R3}`**



The lower numbered the register is, the lower memory address in the stack.

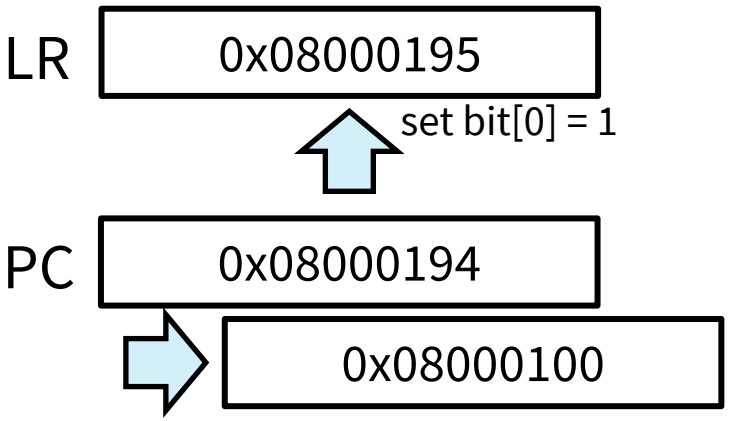
Functions

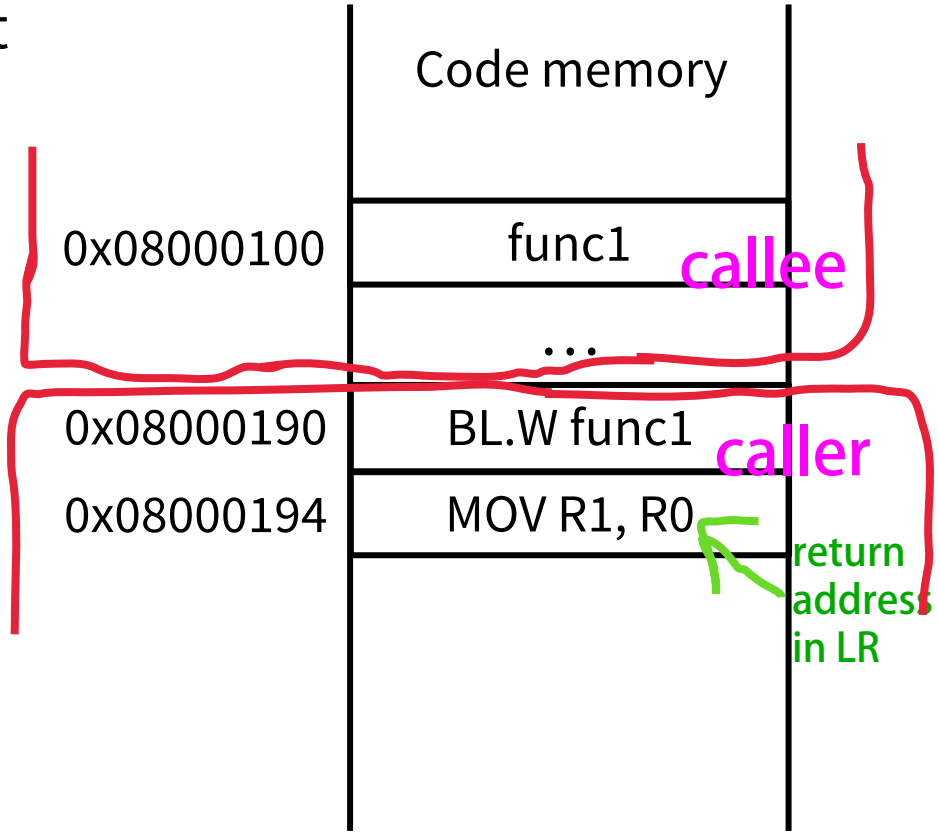
- A function is a subpart of a larger program that is used several times.
  - It saves code memory by reusing the functions. (Otherwise the function is copied all over the memory)
- In C, a function takes arguments from the caller and return a value to the caller.
  - The number of arguments varies: from 0 to a large list (consider printf() as an example).
- Issue #1: What is the mechanism in assembly to call a function and return from a function?
- Issue #2: How to provide arguments to a function and get a return value?

LR: Storing the Return Address

1. **BL label** instruction occurs.
2. The return address of the next instruction is in PC.
3. PC is copied to LR (+ bit[0] set to specify Thumb mode).
4. The branch address is loaded to PC.

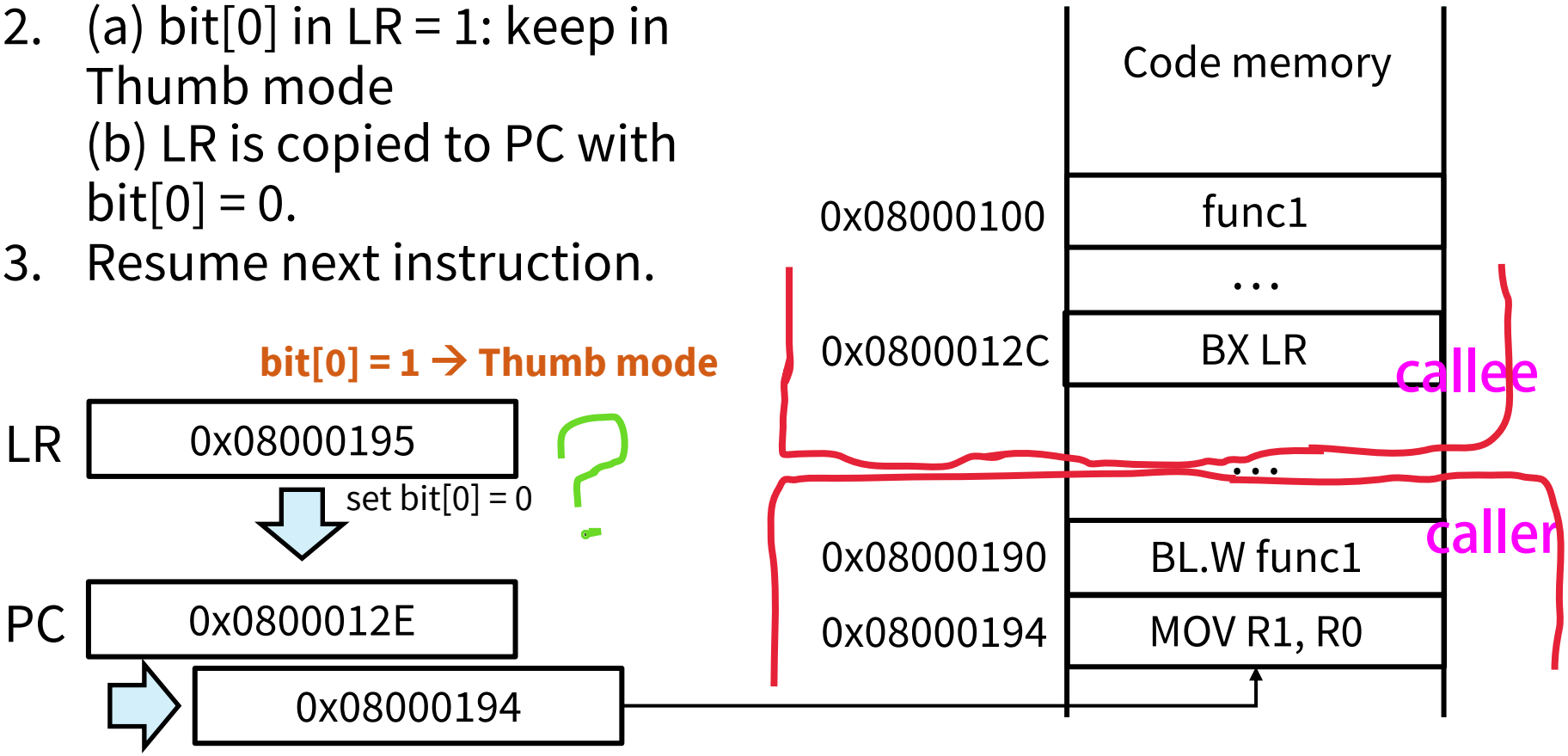
不然就是ARM mode





LR: Restoring the Return Address

1. **BX LR** instruction occurs.
2. (a) bit[0] in LR = 1: keep in Thumb mode
   (b) LR is copied to PC with bit[0] = 0.
3. Resume next instruction.



Application Binary Interface

- Defines rules which allow separately developed functions to work together
- ARM Architecture Procedure Call Standard (AAPCS)
  - Which registers must be saved and restored
  - How to call procedures
  - How to return from procedures
- AAPCS register use conventions:
  - Make it easier to create modular, isolated and integrated code
  - R0 - R3: scratch registers are not expected to be preserved upon returning from a called subroutine r0 - r3 free to use
  - R4 – R8, R10-R11: Preserved (“variable”) registers are expected to have their original values upon returning from a called subroutine

AAPCS Core Register Use

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6,SB,TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Must be saved, restored by callee-procedure it may modify them.

Must be saved, restored by callee-procedure it may modify them. Calling subroutine expects these to retain their value.

Don't need to be saved. May be used for arguments, results, or temporary values.

Function Arguments and Return Values

- How to call a function?
  - With branch link (BL) or branch link and exchange instruction (BLX)
- How to pass the arguments?
  - Much faster to use registers than stack (memory)
  - But quantity of registers is limited.
- Basic rules
  - Process arguments in order they appear in source code
  - Round size up to be a multiple of 4 bytes
  - Copy arguments into core registers (r0-r3), aligning doubles to even registers
  - Copy remaining arguments onto stack, aligning doubles to even addresses

Return Values

- Callee returns value in register(s) or stack.
  - Registers: straight forward
  - Stack
    Caller allocates space for return value, then passes the pointer to the space as an argument to the callee.
    Then the callee stores result at that pointed location.

callee将caller分配的一片空间的地址作为argument，return value放在里面。类似将指针作为一个argument

Return value size	Registers used for passing	
	Fundamental Data Type	Composite Data Type
1-4 bytes	R0	R0
8 bytes	R0 - R1	Stack
16 bytes	R0 - R3	Stack
Indeterminate size	Not available	Stack

Function Call Example

```
int func1(int arg1, int arg2) {
  arg2 += func2(arg1, 4, 5, 6);
  …
}
```

R0 R1

```
__asm int func1(int arg1, int arg2){
    MOVS r4, r1          ; r4 = arg2
                         ; 1st argument already in r0
    MOVS r1, #4          ; 2nd argument for func2
    MOVS r2, #5          ; 3rd argument for func2
    MOVS r3, #6          ; 4th argument for func2

    BL func2             ; call func2

    ADDS r4, r0, r4      ; return value in r0
    …
}
```

设置 argument

Function Call Example (Cont')

```
int func2(int arg1, int arg2, int arg3, int arg4){
    return arg1 * arg2 * arg3 * arg4;
}
```

```
__asm int func2(int arg1, int arg2, int arg3, int arg4){
    MULS r0, r1, r0          ; r0 = arg1 * arg2
    MULS r0, r2, r0          ; r0 = r0 * arg3
    MULS r0, r3, r0          ; r0 = r0 * arg4
    BX   lr                  ; return value in r0
}
```

However, if we execute the func1 as written, it does not work! Why?

1. r4 is not preserved by func1(). 只有R0-R3可以modify
2. LR for func1() is over-written when func2() is called!

func2回func1能回去，
但是func1回它的caller的时候，lr已经被改变了，回不去了。怎么办？请看下页

Return Address on the Stack

- Return address is stored in LR by BL or BLX instructions
- But consider a case where a() calls b() which calls c()
  - On entry to b(), LR holds return address in a().
  - When b() calls c(), LR will be overwritten with return address in b().
  - After c() returns, b() will have lost its return address.
- Key question: does this function call a subroutine?
  - Yes: must save and restore LR on stack just like other preserved registers, but LR value is popped into PC rather than LR
  - No: no need to save or restore LR, as it will not be modified

Fixing func1()

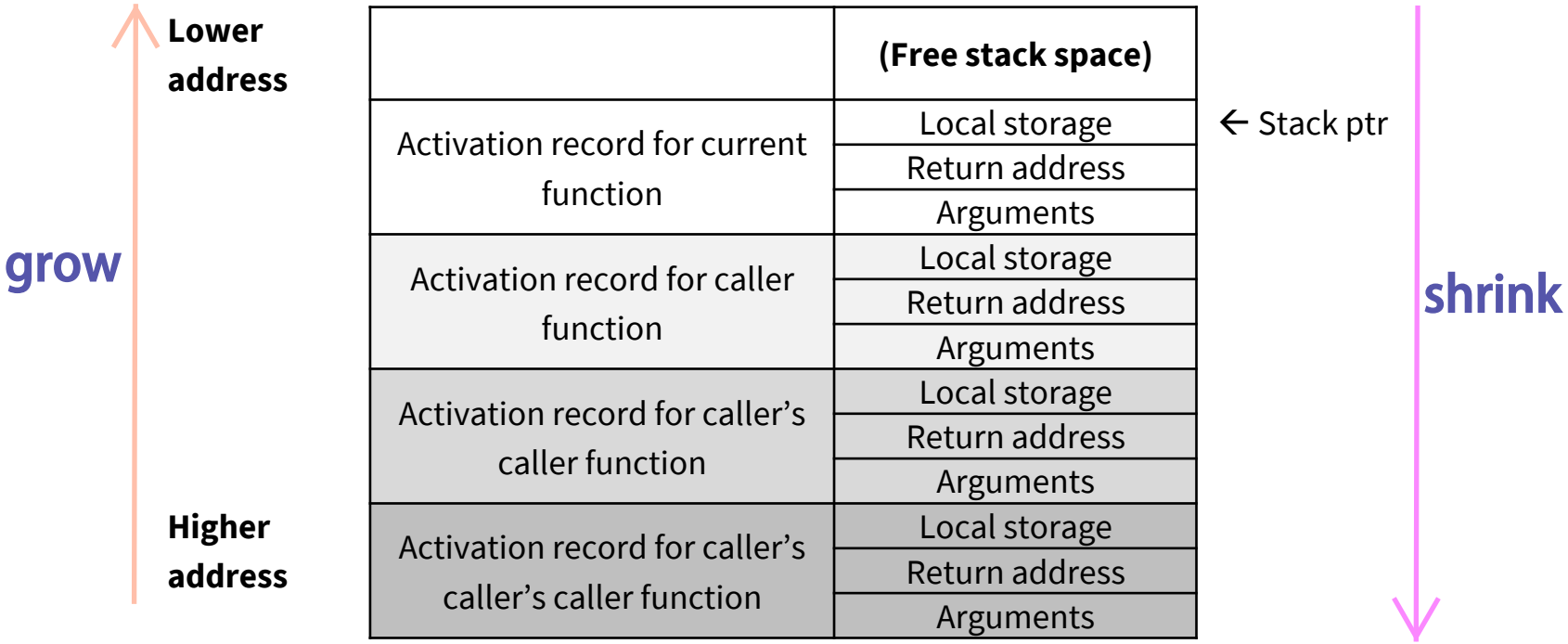
```
int func1(int arg1, int arg2) {
  arg2 += func2(arg1, 4, 5, 6);
  …
}
```

```
__asm int func1(int arg1, int arg2){
    PUSH {r4, lr}       ; store r4 and lr to stack
    MOVS r4, r1         ; r4 = arg1
                        ; 1st argument already in r0
    MOVS r1, #4         ; 2nd argument for func2
    MOVS r2, #5         ; 3rd argument for func2
    MOVS r3, #6         ; 4th argument for func2

    BL func2            ; call func2

    ADDS r4, r0, r4     ; return value in r0
    …
    POP {r4, pc}        ; load lr back to pc, then return
}
```

save and restore

直接变PC，不用再通过LR

Activation Record



- Calling a function creates an activation record on the stack.
- Returning from a function deletes the activation record.
- The record contains possibly return address, arguments, automatic variables (local storage).

EXTRA

Coding Challenge: Bubblesort

Challenge yourself to code the following classic sorting algorithm in ARM assembly.

```
void bubble_asm(int n, int *a){
    int i, j, tmp;    // use low registers
    for (i = n - 1; i > 0; i--){
        for (j = 0; j < i; j++){
            if (a[j] > a[j + 1]){
                // swap a[i] & a[j]
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
        }
    }
}
```

Hint 1: code inside out – prepare if-else, then j-loop and finally i-loop.
Hint 2: use pre-index addressing mode for array access a[i].

Addressing: https://developer.arm.com/documentation/102374/0101/Loads-and-stores---addressing

use memory wisely

Memory requirements and Accessing data in memories

Programmer's World: The Land of Chocolate!

- As many functions and variables as you want.
- All the memory you could ask for.
- Many data types: integers, floating point, etc.
- Many data structures: arrays, lists, trees, sets, dictionaries, etc.
- Many control structures: subroutines, if/then/else, loops, etc.
- OO programming: iterators, polymorphism.

Processor's World

how the compiler treats data, and put data into memory.

- Data types
  - Integers
  - More if you're lucky.
- Instructions
  - Math: +, -, *
  - Logic: AND, OR
  - Shift, rotate
  - Move, swap
  - Compare
  - Branch

Contents of a processor's memory

23	251	151	11	3	1	1	1
213	6	234	2	u	1	1	1
2	33	72	1	a	1	1	a
a	4	h	e	l	l	o	1
67	96	a	0	9	9	9	1
6	11	d	72	7	0	0	0
28	289	37	54	42	0	0	0
213	6	234	2	31	1	1	1

What Memory Does a Program Need?

- Five possible types
  - Code
  - Read-only static data
  - Writable static data
    - Initialized
    - Zero-initialized
    - Uninitialized
  - Heap
  - Stack
- What goes where?
  - Code is obvious.
  - And the others?

```
int a, b;
const char c = 123;
int d = 31;
void main(void) {
    int e;
    char f[32];
    e = d + 7;
    a = e + 29999;
    strcpy(f,“Hello!”);
}
```

zero-initialized

initialized

code region

What Memory Does a Program Need?

- No? Put it in read-only, non-volatile memory
  - Instructions
  - Constant strings
  - Constant operands
  - Initialization values
- Yes? Put it in read/write memory SRAM
  - Variables
  - Intermediate computations
  - Return address

Key Question 1:
Can the information change?

```
int a, b;
const char c = 123;
int d = 31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

What Memory Does a Program Need?

- Statically allocated globally
  - Exists from program start to end
  - Each variable has its own fixed location
  - Space is not reused
- Automatically allocated
  - Exists from function start to end
  - Space can be reused
- Dynamically allocated
  - Exists from explicit allocation to explicit deallocation
  - Space can be reused

malloc()
calloc();
free()

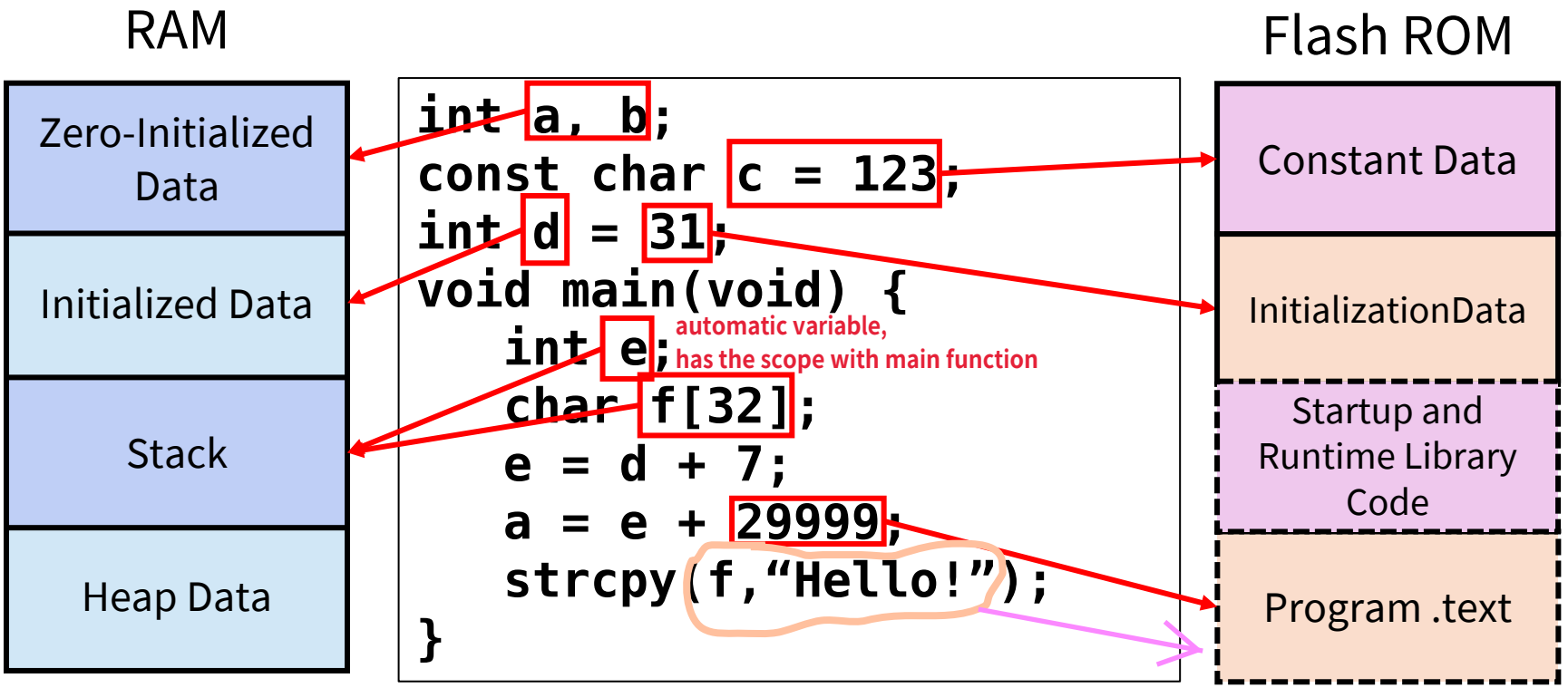
Key Question 2:

How long does the data need to exist?

```
int a, b;
const char c = 123;
int d = 31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

Reuse memory if possible.

Program Memory Use



C Type and Class Qualifiers

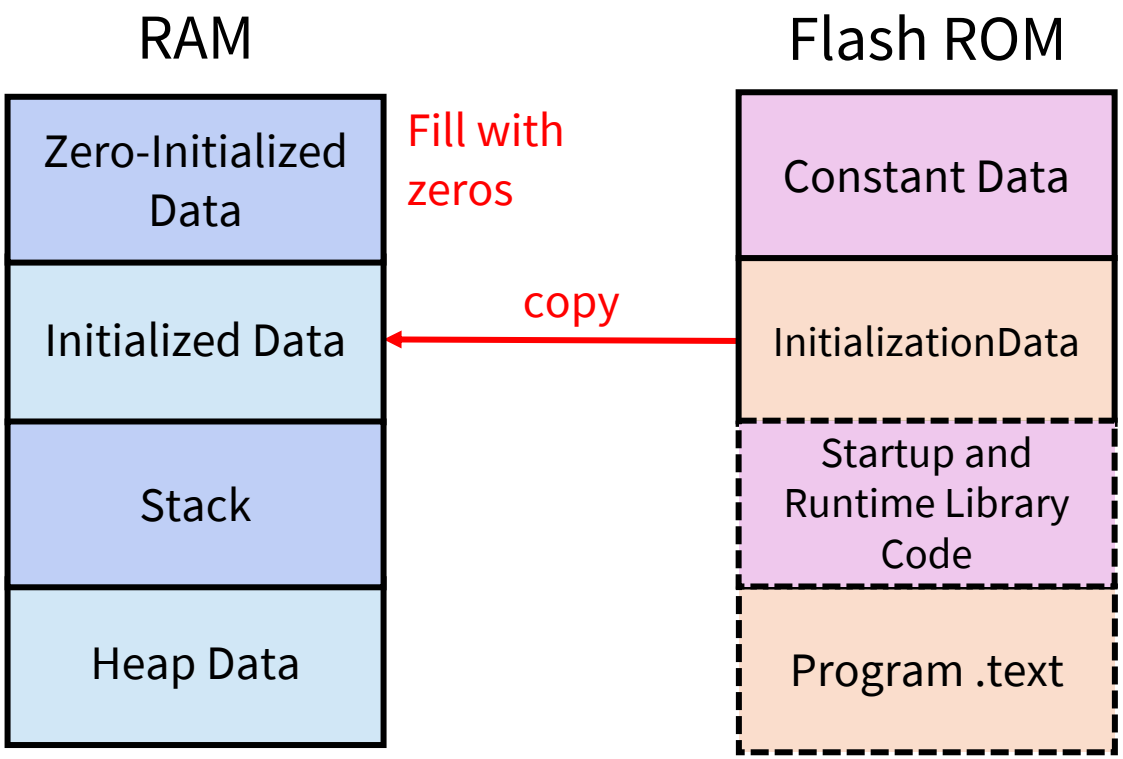
- const
  - Never written by program, can be put in ROM to save RAM
- volatile

to speed up: **register** int a;

  - Can be changed outside of normal program flow: Interrupt Service Routine, hardware-controlled registers
  - Remind the compiler to be careful with optimizations
- static
  - Declared within function, retains value between function invocations
  - Scope is limited to function

C Run-Time Start-Up Module

- After reset, MCU must…
- Initialize hardware
  - Peripherals, etc.
  - Set up stack pointer
- Initialize C or C++ run-time environment
  - Set up heap memory
  - Initialize variables



Accessing Data

- What does it take to get at a variable in memory?
  - Depends on location, which depends on storage type (static, automatic, dynamic)

```
int siA;
void static_auto_local() {
     int aiB;
     static int siC=3;
     int * apD;
     int aiE=4, aiF=5, aiG=6;

     siA = 2;
     aiB = siC + siA;
     apD = & aiB;
     (*apD)++;
     apD = &siC;
     (*apD) += 9;
     apD = &siA;
     apD = &aiE;
     apD = &aiF;
     apD = &aiG;
     (*apD)++;
     aiE+=7;
     *apD = aiE + aiF;
}
```

Accessing Static Variables

- Static variable can be located anywhere in 32-bit memory space, so need a 32-bit pointer
- But we cannot fit a 32-bit pointer into a 16-bit instruction (or a 32-bit instruction), so save the pointer separate from instruction, but nearby so we can access it with a short PC-relative offset.
  - Load the pointer into a register (r0)
  - Can now load variable's value into a register (r1) from memory using that pointer in r0
  - Similarly can store a new value to the variable in memory

Static Variables

将static variable的地址放在离instruction不远处。用offset访问

siA, siC are static

program itself

- Key
  - variable's value
  - variable's address
  - address of copy of variable's address
- Addresses of siA and siC are stored as literals to be loaded into pointers
- Variables siC and siA are located in .data section with initial values.

```
        AREA ||.code||, CODE, ALIGN=2
;;;20          siA = 2;
00000e  2102  MOVS     r1,#2        ; r1 = 2
000010  4a37  LDR      r2,|L1.240|  ; r2 = &siA
000012  6011  STR      r1,[r2,#0]   ; *r2 = r1
;;;21      aiB = siC + siA;
000014  4937  LDR      r1,|L1.244|  ; r1 = &siC
000016  6809  LDR      r1,[r1,#0]   ; r1 = *r1
000018  6812  LDR      r2,[r2,#0]   ; r2 = *r2
00001a  1889  ADDS     r1,r1,r2     ; r1 = r1 + r2
...
                                          ROM
|L1.240|
                DCD       ||siA||
|L1.244|
                DCD       ||siC||
        AREA ||.data||, DATA, ALIGN=2
||siC||
                DCD       0x00000003
                                          RAM
||siA||
                DCD       0x00000000
```

Automatic Variables Stored on Stack

- Variables in C are implicitly automatic.
- Automatic variables are stored in a function's activation record (unless optimised and promoted to register)
- Activation records are located on the stack.
- Calling a function creates an activation record, allocating space on stack.
- Returning from a function deletes the activation record, freeing up space on stack.

```
int main(void) {
    auto vars;
    a();
}

void a(void) {
    auto vars;
    b();
}

void b(void) {
    auto vars;
    c();
}

void c(void) {
    auto vars;
    …
}
```

Automatic Variables

change value of SP

```
int main(void) {
    auto vars;
    a();
}

void a(void) {
    auto vars;
    b();
}

void b(void) {
    auto vars;
    c();
}

void c(void) {
    auto vars;
    …
}
```

Lower address

Higher address

	(Free stack space)
Activation record for current function C	Local storage ← Stack pointer while executing C
	Saved regs
	Arguments (optional)
Activation record for caller function B	Local storage ← Stack pointer while executing B
	Saved regs
	Arguments (optional)
Activation record for caller's caller function A	Local storage ← Stack pointer while executing A
	Saved regs
	Arguments (optional)
Activation record for caller's caller's caller function main	Local storage ← Stack pointer while executing main
	Saved regs
	Arguments (optional)

Addressing Automatic Variables

- Program must allocate space on stack for variables
- Stack addressing uses an offset from the stack pointer:
  `LDR Rm, [SP, #offset]`
- Items on the stack are word aligned
  - In instructions, one byte used for offset, which is multiplied by four
  - Possible offsets: 0, 4, 8, …, 1020
  - Maximum range addressable this way is 1024 bytes

10 bits in instruction

Address	Contents
SP	
SP+0x4	
SP+0x8	
SP+0xC	
SP+0x10	
SP+0x14	
SP+0x18	
SP+0x1C	
SP+0x20	

Automatic Variables

grow

Address	Contents
SP	aiG
SP+4	aiF
SP+8	aiE
SP+0xC	aiB
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

- Initialize aiE
- Initialize aiF
- Initialize aiG

- Store value for aiB

```
;;;14     void static_auto_local( void ) {
000000  b50f PUSH {r0-r3,lr}
000002  4010 SUBS sp, sp #16
;;;15   int aiB;
;;;16   static int siC=3;
;;;18   int aiE=4, aiF=5, aiG=6;
000004  2104 MOVS  r1,#4
000006  9102 STR   r1,[sp,#8]
000008  2105 MOVS  r1,#5
00000a  9101 STR   r1,[sp,#4]
00000c  2106 MOVS  r1,#6
00000e  9100 STR   r1,[sp,#0]
…
;;;21       aiB = siC + siA;
…
00001c  9103 STR   r1,[sp,#0xc]
```

Array Access

This is an advanced topic – not be covered in final exam.

Array Access

- What does it take to get at an array element in memory?
  - Depends on how many dimensions
  - Depends on element size and row width
  - Depends on location, which depends on storage type (static, automatic, dynamic)

```
uint8  buff2[3];
uint16 buff3[5][7];

uint32 arrays(uint8 n, uint8 j) {
  volatile uint32 i;
  i = buff2[0] + buff2[n];
  i += buff3[n][j];
  return i;
}
```

Accessing 1-D Array Elements

- Need to calculate element address: sum of…
  - array start address
  - offset: index * element size
- buff2 is array of unsigned characters
- Move n (argument) from r0 into r2
- Load r3 with pointer to buff2
- Load (byte) r3 with first element of buff2
- Load r4 with pointer to buff2
- Load (byte) r4 with element at address buff2+r2
  - r2 holds argument n
- Add r3 and r4 to form sum

Address	Contents
buff2	buff2[0]
buff2 + 1	buff2[1]
buff2 + 2	buff2[2]

```
00009e  4602 MOV   r2,r0
;;;76     i = buff2[0] + buff2[n];
0000a0  4b1b LDR   r3,|L1.272|
0000a2  781b LDRB  r3,[r3,#0]  ; buff2
0000a4  4c1a LDR   r4,|L1.272|
0000a6  5ca4 LDRB  r4,[r4,r2]
0000a8  1918 ADDS  r0,r3,r4
|L1.272|
   ][7];      DCD   buff2
```

```
uint32 arrays(uint8 n, uint8 j) {
  volatile uint32 i;
  i = buff2[0] + buff2[n];
  i += buff3[n][j];
  return i;
```

Accessing 2-D Array Elements

`uint16 buff3[5][7]`

Address	Contents
buff3	buff3[0][0]
buff3+1	
buff3+2	buff3[0][1]
buff3+3	
(etc.)	
buff3+10	buff3[0][5]
buff3+11	
buff3+12	buff3[0][6]
buff3+13	
buff3+14	buff3[1][0]
buff3+15	
buff3+16	buff3[1][1]
buff3+17	
(etc.)	
buff3+68	buff3[4][6]
buff3+69	

- var[rows][columns]
- Sizes
  - Element: 2 bytes
  - Row: 7*2 bytes = 14 bytes (0xe)
- Offset based on row index and column index
  - column offset = column index * element size
  - row offset = row index * row size

One byte usually has one address although some architectures require memory access to be aligned.

Code to Access 2-D Array

Instruction		r0	r1	r2	r3	r4	Description
;;; i += buff3[n][j];		i	j	n	-	-	
MOVS	r3,#0xe	-	-	-	0xe	-	Load row size
MULS	r3,r2,r3	-	-	n	n*0xe	-	Multiply by row number
LDR	r4,|L1.276|	-	-	-	-	&buff3	Load address of buff3
ADDS	r3,r3,r4	-	-	-	&buff3+n*0xe	-	Add buff3 address to row offset
LSLS	r4,r1,#1	-	j	-	-	j<<1	Multiply column number by 2 (buff3 is uint16 array)
LDRH	r3,[r3,r4]	-	-	-	*(uint16)(&buff3+ n*0xe+j<<1) = buff3[n][j]	j<<1	Load halfword r3 with element at r3+r4 (buff3 + row offset + col offset)
ADDS	r0,r3,r0	i+buff3[n][j]	-	-	buff3[n][j]		Add r3 to r0 (i)

Function Prolog and Epilog

This is another advanced topic – not be covered in final exam.

Prolog and Epilog

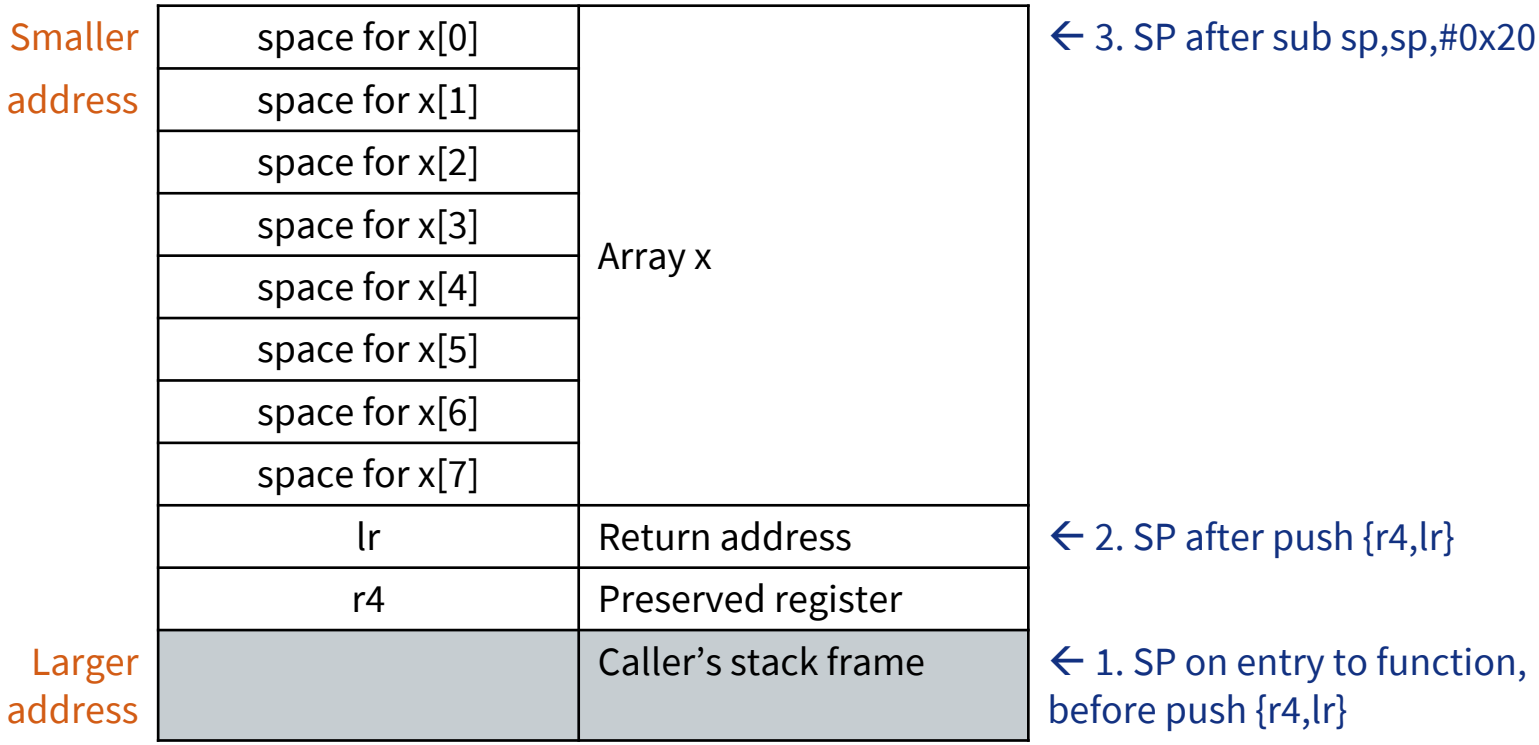
- Before the actual calculation, there are small parts of the program to set up and clean up the registers in function calls.
- In AAPCS
  - Preserved (“variable”) registers r4-r8, r10-r11 must have their original values upon returning.
  - So we must save preserved registers on stack before actual computation (in prolog) and restore them afterwards (in epilog)
- Prolog: instructions/code in the beginning of a function, that
  - Handles function arguments
  - Allocates temporary storage space on stack (subtract from SP)
- Epilog: instructions/code in the end of a function, that
  - may deallocate stack space (add to SP)
  - returns control to calling function

Function Prolog and Epilog

- Prolog: save r4 (preserved register) and link register (return address)
- Prolog: allocate 32 (0x20) bytes on stack for array x by subtracting from SP
- Compute return value, placing in return register r0
- Epilog: deallocate 32 bytes from stack
- Epilog: pop r4 (preserved register) and PC (return address)

```
 fun4 PROC
;;;102  int fun4(char a, int b, char c)
{
;;;103    volatile int x[8];
00010a  b510 PUSH {r4,lr}

00010c  b088 SUB  sp,sp,#0x20
...

;;;106        return a+b+c;
00011c  1858 ADDS r0,r3,r1
00011e  1880 ADDS r0,r0,r2
;;;107    }
000120  b008 ADD  sp,sp,#0x20

000122  bd10 POP  {r4,pc}
              ENDP
```

Activation Record Creation by Prolog



Activation Record Destruction by Epilog