ENGN2219/COMP6719
Computer Systems & Organization

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian National University

Recap: Array Sum

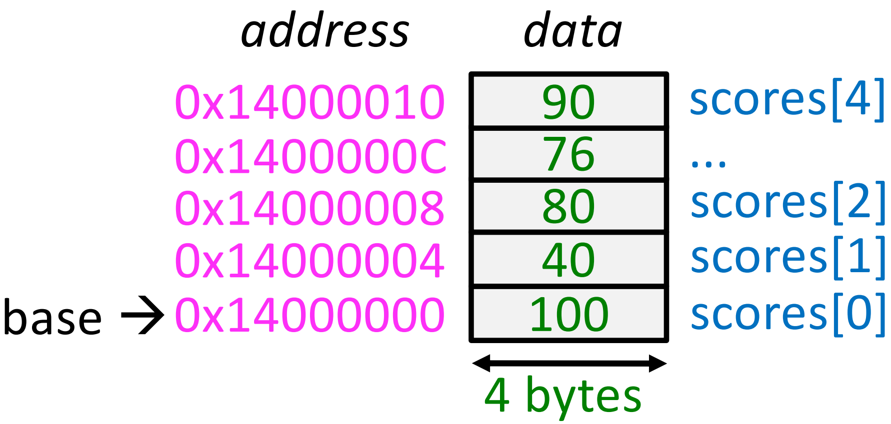
Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```



Showing the scores array in memory

Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

address	data	
0x14000010	90	scores[4]
0x1400000C	76	...
0x14000008	80	scores[2]
0x14000004	40	scores[1]
base → 0x14000000	100	scores[0]
	4 bytes	

Showing the scores array in memory

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV   R0,  #0x14000000
   MOV   R1,  0
LOOP
   CMP   R1,  #200
   BGE   L3
   LSL   R2,  R1,  #2
   LDR   R3,  [R0, R2]
   ADD   R3,  R3,  #10
   STR   R3,  [R0, R2]
   ADD   R1,  R1,  #1
   B     LOOP
L3
```

- R0 = base addr
- i = 0
- i < 200?
- no? exit loop
- i = i + 1
- R3 = scores[i]
- R3 = R3 + 10
- scores[i] += 10
- i = i + 1
- repeat

Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

	address	*data*	
	0x14000010	90	scores[4]
	0x1400000C	76	…
	0x14000008	80	scores[2]
	0x14000004	40	scores[1]
base →	0x14000000	100	scores[0]

4 bytes

Showing the scores array in memory

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000      ▪ R0 = base addr
   MOV  R1,  0                ▪ i = 0
LOOP
   CMP  R1,  #200             ▪ i < 200?
   BGE  L3                    ▪ no? exit loop
   LSL  R2,  R1,  #2          ▪ i = i + 1
   LDR  R3,  [R0, R2]         ▪ R3 = scores[i]
   ADD  R3,  R3,  #10         ▪ R3 = R3 + 10
   STR  R3,  [R0, R2]         ▪ scores[i] += 10
   ADD  R1,  R1,  #1          ▪ i = i + 1
   B    LOOP                  ▪ repeat
L3
```

Another LDR Variant

- We have seen two ways of specifying the offset so far

```
LDR   R3,   [R0, #16]
```

dest   base   offset

```
LDR   R3,   [R0, R2]
```

dest   base   offset in index/offset register

- LSL + LDR combo often used in tandem in array traversals
- ISA supports eliminating the extra LSL instruction

```
LDR   R3,   [R0, R1, LSL #2]
```

Left shift is the same as multiplying by 2

- Memory address
  - Left shift R1 by 2 (scale R1)
  - Add to R0
  - Address = R0 + (R1 * 4)

Condensing Array Sum – 1

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000
   MOV  R1,  0
LOOP
   CMP  R1,  #200
   BGE  L3
   LDR  R3,  [R0, R1, LSL, #2]
   ADD  R3,  R3,  #10
   STR  R3,  [R0, R2]
   ADD  R1,  R1,  #1
   B    LOOP
L3
```

	address	*data*	
	0x14000010	90	scores[4]
	0x1400000C	76	…
	0x14000008	80	scores[2]
	0x14000004	40	scores[1]
base →	0x14000000	100	scores[0]

4 bytes

<u>*Showing the scores array in memory*</u>

ARM Indexing Modes

- Offset Addressing `LDR R0, [R1, R2]`
  - *Address is the sum of base register + offset (#20, #–20, –R2)*
  - *Base register is unchanged*
- Pre-indexed Addressing `LDR R0, [R1, R2]!`
  - *Address is the sum of base register + offset*
  - *Base register is updated with the address*
- Post-index Addressing `LDR R0, [R1], R2`
  - *Address is the base register*
  - *Base register is updated with the new address only after the memory access*

ARM Indexing Modes

- Offset Addressing — LDR R0, [R1, R2]
  - *Address = R1 + R2*
  - *R1 = Unchanged*
- Pre-indexed Addressing — LDR R0, [R1, R2]!
  - *Address = R1 + R2*
  - *R1 = R1 + R2*
- Post-index Addressing — LDR R0, [R1], R2
  - *Address = R1*
  - *R1 = R1 + R2*
- *In all cases, offset can be an immediate*

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000        ▪ R0 = base addr
   MOV  R1,  R0,  #800          ▪ R1 = base + 800
LOOP
   CMP  R0,  R1                 ▪ end of array?
   BGE  L3                      ▪ yes? exit loop
   LDR  R2,  [R0]               ▪ R2 = scores[i]
   ADD  R2,  R2,  #10           ▪ scores[i] + 10
   STR  R2,  [R0], #4           ▪ store scores[i]
                                ▪ and R0 = R0 + 4
   B    LOOP                    ▪ repeat loop
L3
```

	address	*data*	
	0x14000010	90	scores[4]
	0x1400000C	76	...
	0x14000008	80	scores[2]
	0x14000004	40	scores[1]
base →	0x14000000	100	scores[0]

4 bytes

Showing the scores array in memory

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

```
Assembly code:
; R0 = array base address
; R1 = i
    MOV   R0,   #0x14000000
    MOV   R1,   R0,   #800
LOOP
    CMP   R0,   R1
    BGE   L3
    LDR   R2,   [R0]
    ADD   R2,   R2,   #10
    STR   R2,   [R0], #4

    B     LOOP
L3
```

- This version of `Array Sum` first computes the address of the last byte of the array (`#0x14000800`)
- Each iteration of `LOOP` checks if `R0` is `greater than or equal to` `#0x14000800`
- If so, we are done, so step out of `LOOP`
- `STR R2, [R0], #4`
  - Stores `R2` at `[R0],` and after that, adds 4 to `R0`

Explaining

```
1. CMP   R0,   R1
2. BGE   LOOP
```

- When CPU encounters: `CMP  R0,  R1`
  - It subtracts `R1` from `R0`
  - It sets the `flags` in the CSPR register
  - No register is updated with the result (no side-effects)
- When CPU encounters: `BGE LOOP`
  - CPU checks the `flags` to establish if `R0 is greater than or equal to R1`
- Some conditions are easy to establish, and others harder
  - `EQ` is easily established by looking at the *zero* flag
  - If the *zero* flag is 1, it means (`R0 – R1`) is 0, meaning `R0` and `R1` are equal

Conditional Execution

- Week 5, part 2 lecture
  - Two ways of setting the flags in the CPSR register
  - Conditional execution in general

Bytes and Characters

- Characters on the English keyboard can be encoded in a single byte
- **char** in C is an 8-bit integer under the hood
  - C operators close to the hardware (basic types)
  - No string, list, or other composite types
- ASCII standard is for mapping characters to integer codes
  - Other standards such as Unicode are ASCII supersets
- Need instructions to manipulate bytes!

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Loading/Storing Bytes

- LDRB
  - *Load byte in register, and zero extend to fill the 32 bits*
- LDRSB
  - *Load byte in register, and sign-extend to fill the 32 bits*
- STRB
  - *Store the LSB of the 32-bit integer into the specified byte in memory*
  - *More significant bits of the register are ignored*

Loading/Storing Bytes

- What is in R1, R2, and memory after each of the instruction has executed?
- Assume R4 = 0

Byte Address	*Data*
4	...
3	F7
2	8C
1	42
0	03

Registers

xx	xx	xx	xx	LDRB R1, [R4, #2]
xx	xx	xx	xx	LDRSB R2, [R4, #2]
11	10	A1	9B	STRB R3, [R4, #3]

Loading/Storing Bytes

- What is in `R1`, `R2`, and `memory` after each of the instruction has executed?
- Assume R4 = 0

Byte Address	*Data*
4	...
3	9B
2	8C
1	42
0	03

Registers

00	00	00	8C	LDRB R1, [R4, #2]
FF	FF	FF	8C	LDRSB R2, [R4, #2]
xx	xx	xx	9B	STRB R3, [R4, #3]

Strings in C

- A series of characters is a string
  - char welcome[6] = {'H', 'E', 'L', 'L', 'O', '\0'};
  - char welcome[]   = "HELLO";

- Compiler figures out the length
- 5 + 1 for '\0'
- Manually track length (unlike Python)

- Compiler inserts a null terminator '\0' automatically

- Need a way to know the end of the string
- C strings are null-terminated

Ex: Manipulating Char Array

C code:

```
char array[10] = "ENGN2219!";
int i;

for (i = 0; i < 10; i = i + 1)
   array[i] = array[i] – 32;
```

Ex: Manipulating Char Array

- Transform the 10-character ASCII string, namely array, from lower case to upper case

```
C code:

   char array[10] = "finalexam";
   int i;

   for (i = 0; i < 10; i = i + 1)
      array[i] = array[i] – 32;
```

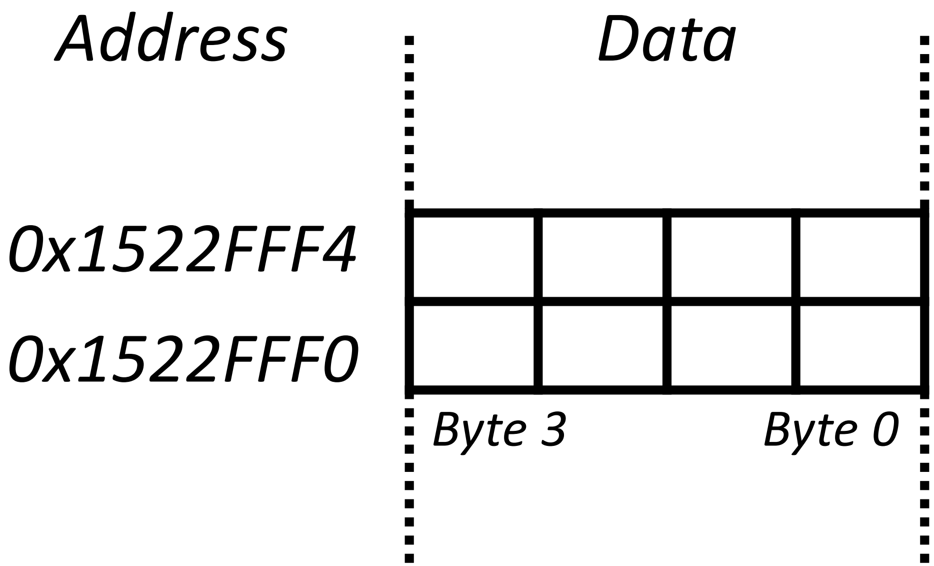
```
Assembly code:
; R0 = base addr, R1 = i
   MOV    R0,  #0              ; i = 0
LOOP
   CMP    R1,  #10             ; i < 10?
   BGE    DONE                 ; if i >= 10, exit
   LDRB   R2,  [R0, R1]        ; R2 = array[i]
   SUB    R2,  R2,  #32        ; subtract 32
   STRB   R2,  [R0, R1]        ; store array[i]
   ADD    R1,  R1,  #1         ; i = i + 1
   B      LOOP                 ; repeat loop
DONE
```

Exercise: Strings in Memory

- Show how “HELLO!” is stored in memory below at address 0x1522FFF0.

ASCII Encoding

H	0x48
E	0x65
L	0x6C
O	0x6F
!	0x21
Null	0x00



Exercise: Strings in Memory

- Show how “HELLO!” is stored in memory below at address 0x1522FFF0.

ASCII Encoding

H	0x48
E	0x65
L	0x6C
O	0x6F
!	0x21
Null	0x00

Address	*Data*			
0x1522FFF4		00	21	6F
0x1522FFF0	6C	6C	65	48
	Byte 3			*Byte 0*

Practice

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000       ; R0 = 0x60000000

  LDR R1, [R0]              ; R1 = array[0]
  LSL R1, R1, #3            ; R1 = R1 << 3 = R1*8
  STR R1, [R0]              ; array[0] = R1

  LDR R1, [R0, #4]          ; R1 = array[1]
  LSL R1, R1, #3            ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]          ; array[1] = R1
```

Exercise

C Code

```
int array[200];
int i;
for (i=199; i >= 0; i = i - 1)
   array[i] = array[i] * 8;
```

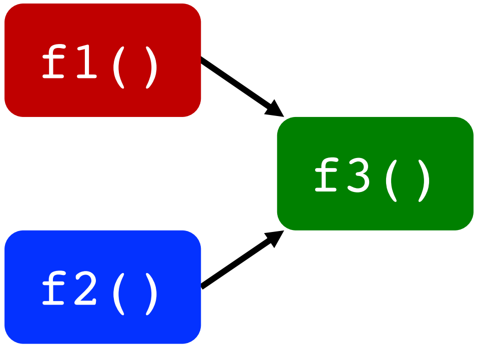
ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]    ; R2 = array(i)
  LSL  R2, R2, #3              ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]    ; array(i) = R2
  SUBS R0, R0, #1              ; i = i - 1
                               ; and set flags
  BPL  FOR                     ; if (i>=0) repeat
loop
```

Functions

- High-level languages offer functions to enable
  - Abstraction & Modularity
  - Code reuse
  - Readability
  - Testability & validation
  - Maintainability
- Functions are also called *procedures* or *subroutines*
- Functions are ubiquitous, encouraging `ISA` support
  - Special jump instructions
  - Special scratch space to store temporary variables
  - Ways to reduce interference b/w functions



Functions: Our Goal

- Architectural support for functions
  - Branch and Link instruction (BL)
  - Stack Pointer (SP)
  - Link Register (LR)
- Microarchitecture-level impacts of programming styles (Iteration vs. Recursion)
- Provides a deeper understanding of hardware/software interaction and tradeoffs

Functions in C

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}
```

42 and 7 are function arguments provided by caller, i.e., `main()`

```
int sum(int a, int b)
{
  return (a + b);
}
```

- `main()` is caller (calling someone else)
  - Returns nothing (void)
  - No input arguments

- `sum()` is *callee* (being called by someone)
  - Two input arguments of type int: a and b
  - Return type: integer
  - Returns the sum of a and b

Leaf and Non-Leaf Functions

- `sum()` is a leaf function
  - It does not call another function
- `main()` is a non-leaf function
  - It calls another function
- Non-leaf functions are more complicated especially at the assembly level
- `sum()` can be called from many different functions
  - Code reuse

Functions as Detectives

- Secret mission
- Acquire necessary resources
- Perform the mission
- Leave no trace
- Return safely

Functions as Detectives

- Caller stores *arguments* in specific registers
- Caller transfers *flow control* to the callee (`call`)
- Callee acquires/allocates memory for doing work
- Callee executes the function body
- Callee stores the result in a specific register
- Callee *returns* control to the caller (`return`)

ARM Function Calls

- Instruction for calling the function
  - **`BL` (B**ranch and **L**ink**)**
  - CPU branches to the label specified by `BL`
  - CPU stores the *return address* in the link register (`LR`)
- Return address is the address of the next instruction after the function call
- Returning from function
  - Move the link register into `PC`
  - `MOV PC, LR`
- Passing arguments (convention)
  - `R0, R1, R2, R3`
- Returning value (convention)
  - `R0`

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN      BL   SIMPLE
0x00000204           ADD  R4, R5, R6
...

0x00401020 SIMPLE    MOV  PC, LR
```

- `BL` — branches to `SIMPLE`
  `LR` = `PC` + 4 = **0x00000204**
- `MOV PC, LR` — makes `PC` = `LR`
  (the next instruction executed is at **0x00000200**)

- `MAIN` and `SIMPLE` are labels (memory addresses) in assembly
- `BL` transfers flow to `SIMPLE` and stores the *return address* in `LR`
- The function returns after `MOV`, and the next instruction (`ADD`) is executed

Example: Difference of Sums

```
C code:
int main() {
   int y;
   ...
   y = diffofsums(2, 3, 4, 5);
   ...
}

int diffofsums(int f, int g, int h, int i) {
   int result;
   result = (f + g) – (h + i);
   return result;
}
```

ARM Assembly Code

```
; R4 = y

MAIN
   ...
   MOV R0, #2          ; argument 0 = 2
   MOV R1, #3          ; argument 1 = 3
   MOV R2, #4          ; argument 2 = 4
   MOV R3, #5          ; argument 3 = 5
   BL  DIFFOFSUMS      ; call function
   MOV R4, R0          ; y = returned value
   ...

; R4 = result
DIFFOFSUMS
   ADD R8, R0, R1      ; R8 = f + g
   ADD R9, R2, R3      ; R9 = h + i
   SUB R4, R8, R9      ; result = (f + g) - (h + i)
   MOV R0, R4          ; put return value in R0
   MOV PC, LR          ; return to caller
```

Questions

- How can we pass more than 4 function arguments?
- How can we ensure that registers in use by the caller are not corrupted?
  - `DIFFOFSUMS` overwrites `R4, R8, R9`
  - `MAIN` may need these registers after return
- The Stack
  - A special area in memory used across function calls
  - Preserving registers, passing arguments, scratch space

The Stack

- Abstract view
  - Last In First Out (LIFO) Queue
- Stored in memory at some arbitrary address in memory
- Caller and callee can *push* things onto the stack and *pop* things off the stack
- Stack expands and contracts over time as function call and return