Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

01 – Addressing Modes

CS1022 – Introduction to Computing II

Dr Jonathan Dukes / jdukes@scss.tcd.ie
School of Computer Science and Statistics

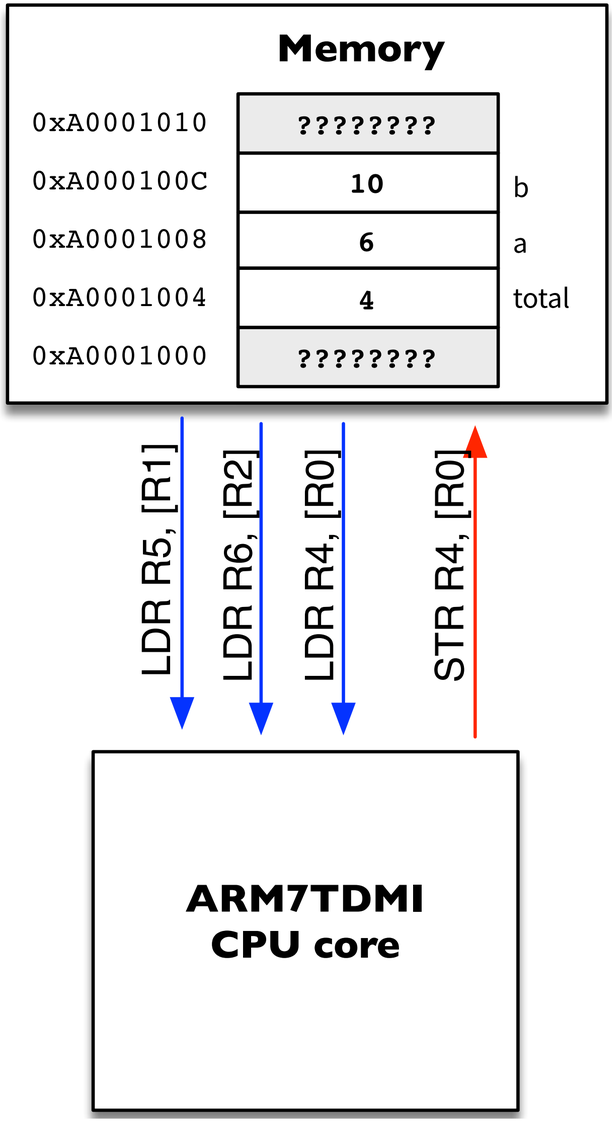
LDR and STR Reviewed

e.g. Compute **total = total + (a × b)**, where **total**, **a** and **b** are stored in memory at the addresses contained in **R0**, **R1** and **R2** respectively

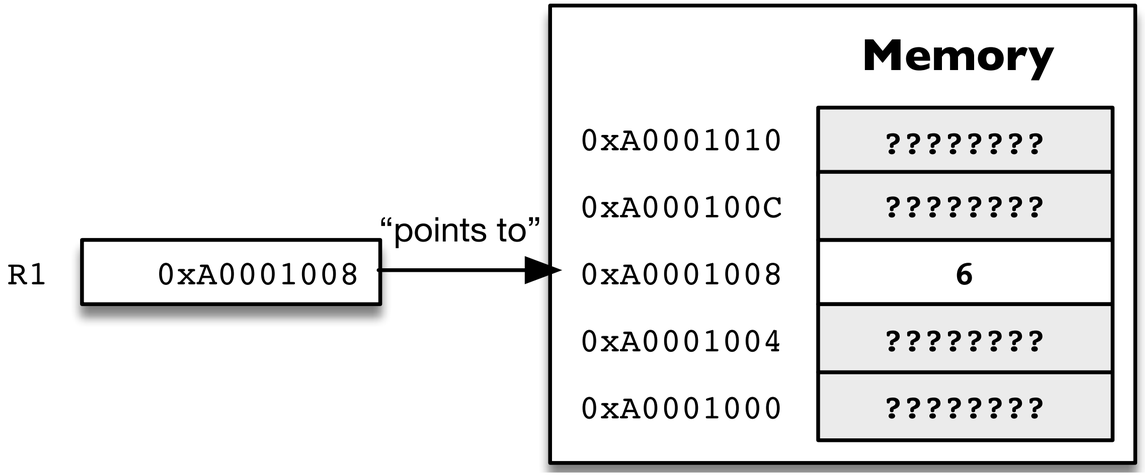
```
LDR   R5, [R1]      ; Load a
LDR   R6, [R2]      ; Load b
MUL   R5, R6, R5    ; tmp = a * b

LDR   R4, [R0]      ; Load total
ADD   R4, R4, R5    ; total = total + (tmp)

STR   R4, [R0]      ; Store total back to memory
```



Addressing Modes



The syntax `[R1]` is just one of many ways that we can specify the the memory address that we want to access using LDR or STR

Remember: `[R1]` tells the processor to access the value in memory at the address contained in register R1

The syntax `[R1]` corresponds to an **Addressing Mode**

`[R1]` is an abbreviated form of `[R1, #0]` (the #0 is implied if omitted)

The address accessed by LDR or STR is called the **Effective Address (EA)**

Immediate Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, #offset]`	EA = Rn + offset	`LDR R0, [R1, #4]`
`[Rn]`	EA = Rn + #0 *(#0 is assumed)*	`LDR R0, [R1]`

Effective Address is calculated by adding ***offset*** to the address in the base register ***Rn*** (note: offset may be negative)

Base register ***Rn*** is not changed

Example: load three consecutive word-size values from memory into registers R4, R5 and R6, beginning at the address contained in R0

```
LDR   R0, =label    ; Initialise base register
LDR   R4, [R0]      ; R4 = Memory.word[R0 + 0] (default = 0)
LDR   R5, [R0, #4]  ; R5 = Memory.word[R0 + 4]
LDR   R6, [R0, #8]  ; R6 = Memory.word[R0 + 8]
```

Register Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, Rm]`	EA = Rn + Rm	`LDR R0, [R1, R2]`

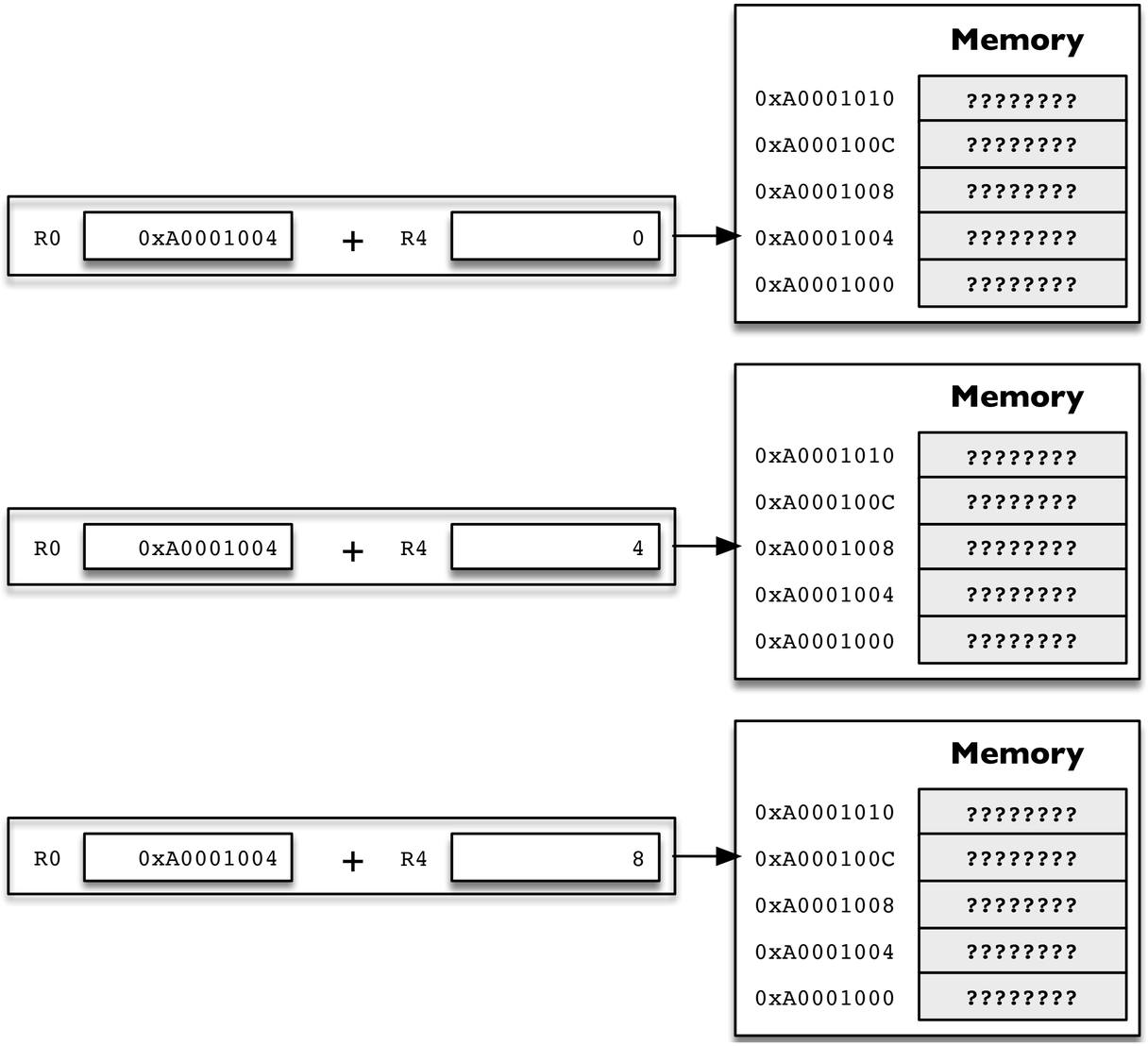
Effective Address is calculated by adding offset in ***Rm*** to the address in the base register ***Rn***

Base register ***Rn*** and offset register ***Rm*** are not changed

Example: load three consecutive word values from memory into registers R1, R2 and R3 beginning at the address in R0

```
LDR    r0, =label      ; Initialise base register
LDR    r4, =0          ; Initialise offset register = 0
LDR    r1, [r0, r4]    ; r1 = Memory.word[r0 + r4]
ADD    r4, r4, #4      ; r4 = r4 + 4
LDR    r2, [r0, r4]    ; r2 = Memory.word[r0 + r4]
ADD    r4, r4, #4      ; r4 = r4 + 4
LDR    r3, [r0, r4]    ; r3 = Memory.word[r0 + r4]
```

Register Offset Addressing



Reading
1st Word

Reading
2nd Word

Reading
3rd Word

Example: Upper Case String (using Register Offset Addressing)


```
   LDR   r1, =teststr     ; address = teststr
   LDR   r2, =0           ; index = 0

   LDRB  r0, [r1, r2]     ; char = Memory.Byte[address + index]
whStr
   CMP   r0, #0           ; while ( char != 0 )
   BEQ   eWhStr           ; {
   CMP   r0, #'a'         ;  if (char >= 'a'
   BLO   eIfLC            ;   &&
   CMP   r0, #'z'         ;   char <= 'z')
   BHI   eIfLC            ;  {
   BIC   r0, #0x00000020  ;   char = char AND NOT 0x00000020
   STRB  r0, [r1, r2]     ;   Memory.byte[address + index] = char
eIfLC                     ;  }
   ADD   r2, r2, #1       ;  index = index + 1
   LDRB  r0, [r1, r2]     ;  char = Memory.Byte[address + index]
   B     whStr            ; }
eWhStr                    ;
```

Example: Upper Case String (using Register Offset Addressing)


```
   LDR   r1, =teststr     ; address = teststr
   LDR   r2, =0           ; index = 0

whStr
   LDRB  r0, [r1, r2]     ; while ( (char = Memory.Byte[address + index])
   CMP   r0, #0           ;               != 0 )
   BEQ   eWhStr           ; {
   CMP   r0, #'a'         ;  if (char >= 'a'
   BLO   eIfLC            ;    &&
   CMP   r0, #'z'         ;    char <= 'z')
   BHI   eIfLC            ;  {
   BIC   r0, #0x00000020  ;    char = char AND NOT 0x00000020
   STRB  r0, [r1, r2]     ;    Memory.byte[address + index] = char
eIfLC                     ;  }
   ADD   r2, r2, #1       ;  index = index + 1
   LDRB  r0, [r1, r2]     ;  char = Memory.Byte[address + index]
   B     whStr            ; }
eWhStr                    ;
```

By moving the label whStr to include the top LDRB, we can eliminate the bottom LDRB. We will use this construct from now on.

pseudo-code in blue evaluates to the newly loaded value of char (i.e. just like Java, C, etc.!)

no longer needed!

LDR and STR Addressing Modes

Mode	Pseudo-code Operation *	Example
Offset modes – the base address register *is not* modified		
[Rn]	EA = Rn	`LDR R0, [R1]`
[Rn, #offset]	EA = Rn + offset	`LDR R0, [R1, #4]`
[Rn, Rm]	EA = Rn + Rm	`LDR R0, [R1, R2]`
[Rn, Rm, LSL #shift]	EA = Rn + (Rm << shift)	`LDR R0, [R1, R2, LSL #2]`
[Rn, Rm, LSR #shift]	EA = Rn + (Rm >> shift)	`LDR R0, [R1, R2, LSR #2]`
Pre-indexed modes – the base address register *is* modified *before* accessing memory		
[Rn, #offset]!	Rn = Rn + offset EA = Rn	`LDR R0, [R1, #4]!`
[Rn, Rm]!	Rn = Rn + Rm EA = Rn	`LDR R0, [R1, R2]!`
[Rn, Rm, LSL #shift]!	Rn = Rn + (Rm << shift) EA = Rn	`LDR R0, [R1, R2, LSL #2]!`
[Rn, Rm, LSR #shift]!	Rn = Rn + (Rm >> shift) EA = Rn	`LDR R0, [R1, R2, LSR #2]!`
Post-indexed modes – the base address register *is* modified *after* accessing memory		
[Rn], #offset	EA = Rn Rn = Rn + offset	`LDR R0, [R1], #4`
[Rn], Rm	EA = Rn Rn = Rn + Rm	`LDR R0, [R1], R2`
[Rn], Rm, LSL #shift	EA = Rn Rn = Rn + (Rm << shift)	`LDR R0, [R1], R2, LSL #2`
[Rn], Rm, LSR #shift	EA = Rn Rn = Rn + (Rm >> shift)	`LDR R0, [R1], R2, LSR #2`

***** EA is the *Effective Address*, which is the memory address to which the load or store operation is applied.

Note: only a subset of the above addressing modes can be used to load or store halfwords, signed-halfwords or signed-bytes. See ARM Architecture Reference Manual section A5.3.

STM and LDM – STore and LoaD Multiple

Instruction	Example	
Base register Rn is not modified (no !)		
STM*mode* Rn, {list}	`STMIA R12, {R0-R3}`	Store the contents of R0-R3 in memory at the address contained in R12
LDM*mode* Rn, {list}	`LDMIA R12, {R5,R7,R10}`	Load R5, R7 and R10 with the contents of memory at the address contained in R12
Base register Rn is modified (Rn!)		
STMmode Rn!, {list}	`STMFD SP!, {R4-R12,R14}`	Push R4-R12 and R14 on to the system stack, updating the system stack pointer
LDM*mode* Rn!, {list}	`LDMFD SP!, {R4-R12,R14}`	Pop 10 words off the top of the system stack into R4-R12 and R14, updating the system stack pointer

mode

STM – STore Multiple	
Instruction	Stack-Oriented Synonym
STMDB (decrement before)	**STMFD** (full descending)
STMIB (increment before)	**STMFA** (full ascending)
STMDA (decrement after)	**STMED** (empty decending)
STMIA (increment after)	**STMEA** (empty ascending)

LDM – LoaD Multiple	
Instruction	Stack-Oriented Synonym
LDMIA (increment after)	**LDMFD** (full descending)
LDMDA (decrement after)	**LDMFA** (full ascending)
LDMIB (increment before)	**LDMED** (empty decending)
LDMDB (decrement before)	**LDMEA** (empty ascending)

Example: Sum (using Register Offset Addressing)

Design and write an assembly language program that will calculate the sum of 10 word-size values stored in memory

```
   LDR    R1, =testdata  ; address = address of first word-wize value
   LDR    R0, =0         ; sum = 0
   LDR    R4, =0         ; count = 0
   LDR    R5, =0         ; offset = 0

whSum
   CMP    R4, #10        ; while (count < 10)
   BHS    eWhSum         ; {
   LDR    R6, [R1, R5]   ;  num = Memory.Word[address + offset]
   ADD    R0, R0, R6     ;  sum = sum + num
   ADD    R5, R5, #4     ;  offset = offset + 4
   ADD    R4, R4, #1     ;  count = count + 1
   B      whSum          ; }
eWhSum                   ;
```

Scaled Register Offset Addressing

Addressing Mode Syntax	Operation	Example
`[Rn, Rm, LSL #shift]`	EA = Rn + (Rm x 2^{shift})	`LDR R0, [R1, R2, LSL #2]`

Effective Address is calculated by adding offset in ***Rm***, shifted left by ***shift*** bits, to the address in the base register ***Rn***

Base register ***Rn*** and offset register ***Rm*** are not changed

Example: load three consecutive word values from memory into registers R1, R2 and R3 beginning at the address in R0

```
LDR    r0, =label              ; Initialise base register
LDR    r4, =0                  ; Initialise index register = 0
LDR    r1, [r0, r4, LSL #2]    ; r1 = Memory.Word[r0 + r4 * 4]
ADD    r4, r4, #1              ; r4 = r4 + 1
LDR    r2, [r0, r4, LSL #2]    ; r2 = Memory.Word[r0 + r4 * 4]
ADD    r4, r4, #1              ; r4 = r4 + 1
LDR    r3, [r0, r4, LSL #2]    ; r3 = Memory.Word[r0 + r4 * 4]
```

Example: Sum (using Scaled Register Offset Addressing)

Re-write our program to calculate the sum of 10 word-size values stored in memory, this time using scaled register offset addressing

Allows *count* to be used for *offset*!!

```
   LDR   R1, =testdata          ; address = address of first word-wize value
   LDR   R0, =0                 ; sum = 0
   LDR   R4, =0                 ; count = 0

whSum
   CMP   R4, #10                ; while (count < 10)
   BHS   eWhSum                 ; {
   LDR   R6, [R1, R4, LSL #2]   ;  num = Memory.Word[address + count * 4]
   ADD   R0, R0, R6             ;  sum = sum + num
   ADD   R4, R4, #1             ;  count = count + 1
   B     whSum                  ; }
eWhSum                          ;
```

Pre- and Post-Indexed Addressing

Many programs iterate sequentially through memory (examples?)

Often manifested as an LDR/STR, followed by an ADD

```
LDR   R4, [R1]      ; load
ADD   R1, R1, #4    ; increment base address register R1
```

ARM architecture provides a set of addressing modes that incorporate the increment/decrement into the execution of the LDR/STR instruction

Pre-Indexed Addressing	Post-Indexed Addressing
1. Increment / Decrement base address register (Rn) 2. Compute Effective Address	1. Compute Effective Address 2. Increment / Decrement base address register (Rn)

Post-Indexed Addressing Modes


```
LDR   R4, [R1]
ADD   R1, R1, #4
```

⟺

```
LDR   R4, [R1], #4
```

Immediate Offset Addressing

Immediate Post-Indexed Addressing

Syntax: post-indexed addressing modes place the immediate/register/scale addressing mode operands **after** the [ ]

Behaviour: (i) the LDR/STR is performed first using the original base register value, (ii) the base register is updated by applying the post-index operation

Modes: Immediate Post-Indexed, Register Post-Indexed, Scaled Register Post-Indexed (LSR/LSL)

Pre-Indexed Addressing Modes


```
ADD   R1, R1, #4
LDR   R4, [R1]
```

Immediate Offset Addressing

```
LDR   R4, [R1, #4]!
```

Immediate Pre-Indexed Addressing

Syntax: pre-indexed addressing modes place the immediate/register/scale addressing mode operands **inside** the [ ]

Behaviour: (i) the base register is updated by applying the pre-index operation, (ii) the LDR/STR is performed using the updated base register value

Modes: Immediate Pre-Indexed, Register Post-Indexed, Scaled Register Post-Indexed (LSR/LSL)

Example: Upper Case String (using Immediate Post-Indexed Addressing)


```
   LDR   r1, =teststr        ; address = teststr

whStr
   LDRB  r0, [r1], #1        ; while ( (char = Memory.Byte[address++])
   CMP   r0, #0              ;       != 0 )
   BEQ   eWhStr              ; {
   CMP   r0, #'a'            ;  if (char >= 'a'
   BCC   endIfLC             ;   AND
   CMP   r0, #'z'            ;   char <= 'z')
   BHI   endIfLC             ;  {
   BIC   r0, #0x00000020     ;   char = char AND NOT 0x00000020
   STRB  r0, [r1, #-1]       ;   Memory.Byte[address - 1] = char
endIfLC                      ;  }
   B     whStr               ;
eWhStr                       ; }
```

Immediate Post-Indexed Addressing

Note the use of Immediate Offset Addressing to (temporarily) compensate for the earlier post-increment

Example: Sum (using Immediate Post-Indexed Addressing)


```
   LDR   R1, =testdata ; address = address of first word-wize value
   LDR   R0, =0        ; sum = 0
   LDR   R4, =0        ; count = 0

whSum
   CMP   R4, #10       ; while (count < 10)
   BHS   eWhSum        ; {
   LDR   R6, [R1], #4  ;  num = Memory.Word[address]; address=address+4
   ADD   R0, R0, R6    ;  sum = sum + num
   ADD   R4, R4, #1    ;  count = count + 1
   B     whSum         ; }
eWhSum                 ;
```