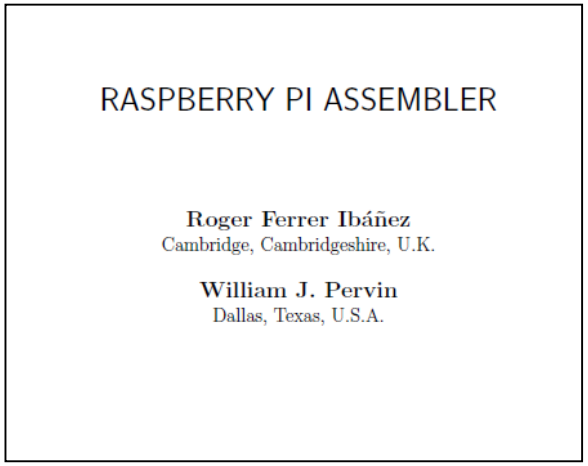
Raspberry Pi Assembler
Arrays and Structures

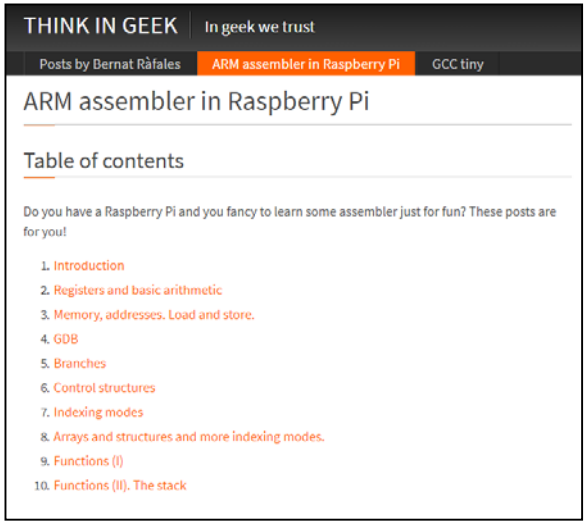

RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

William J. Pervin
Dallas, Texas, U.S.A.

Chapter 8: Raspberry Pi Assembler
"Raspberry Pi Assembler" by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/27/arm-assembler-raspberry-pi-chapter-8/


THINK IN GEEK In geek we trust

Posts by Bernat Ràfales | ARM assembler in Raspberry Pi | GCC tiny

ARM assembler in Raspberry Pi

Table of contents

Do you have a Raspberry Pi and you fancy to learn some assembler just for fun? These posts are for you!

1. Introduction
2. Registers and basic arithmetic
3. Memory, addresses. Load and store.
4. GDB
5. Branches
6. Control structures
7. Indexing modes
8. Arrays and structures and more indexing modes.
9. Functions (I)
10. Functions (II). The stack

Embedded Systems II – EEE3096S
Y. Abdul Gaffar
S. Winberg

Raspberry Pi Assembler

Arrays and Structures

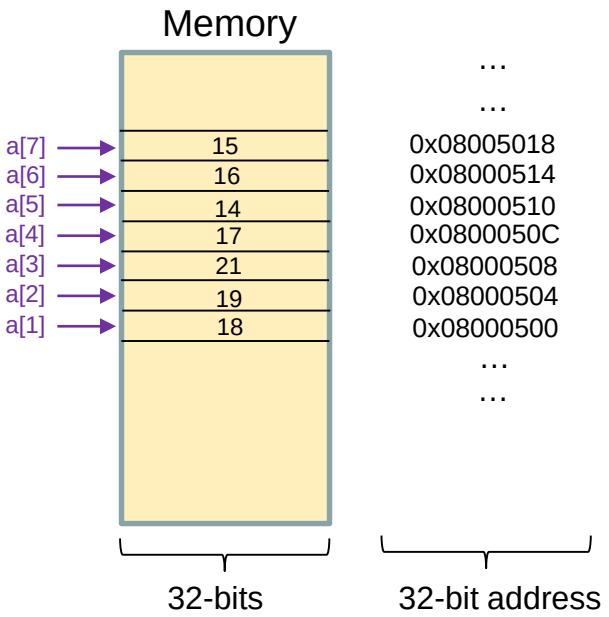
- So far, we have only looked at scalar 32-bit variables
- The next step, is to work with arrays and structures

Arrays

Raspberry Pi Assembler

Arrays

- What is an array?
  - An array is a collection of data of the same type. Examples of types are char, int, double and word. Each type reserves a different amount of space in memory.
  - Example: array **a** of average temperature values during the week. The name of the array is a and it has 7 values. Let **a** = [18 19 21 17 14 16 15]

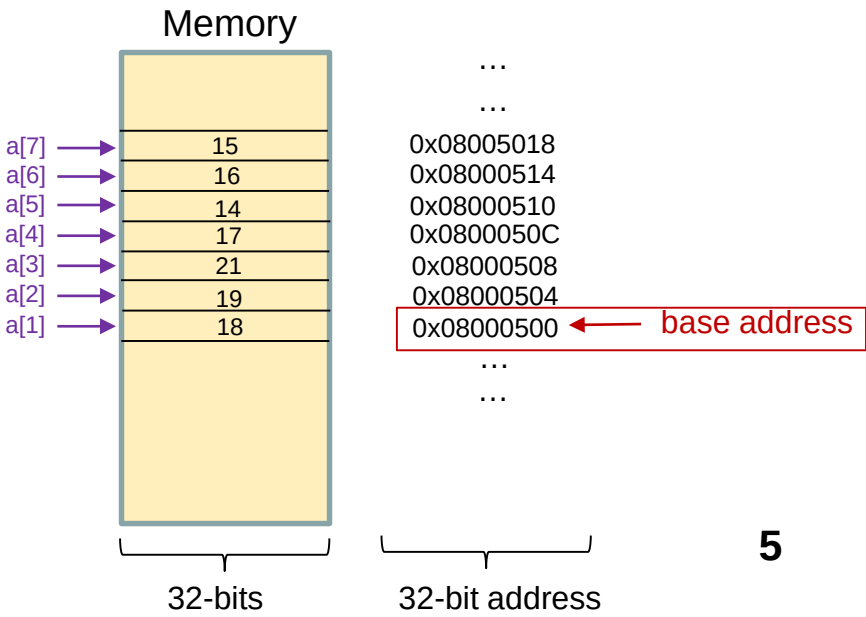


Raspberry Pi Assembler

Arrays

- What is an array?
  - An array is a collection of data of the same type. Examples of types are char, int, double and word. Each type reserves a different amount of space in memory.
  - Example: array **a** of average temperature values during the week. The name of the array is a and it has 7 values. Let **a** = [18 19 21 17 14 16 15]

- What are the properties of an array
  - Base address: address of the first element of the array (eg. 0x080000500)
  - Name of the array: represents the base address

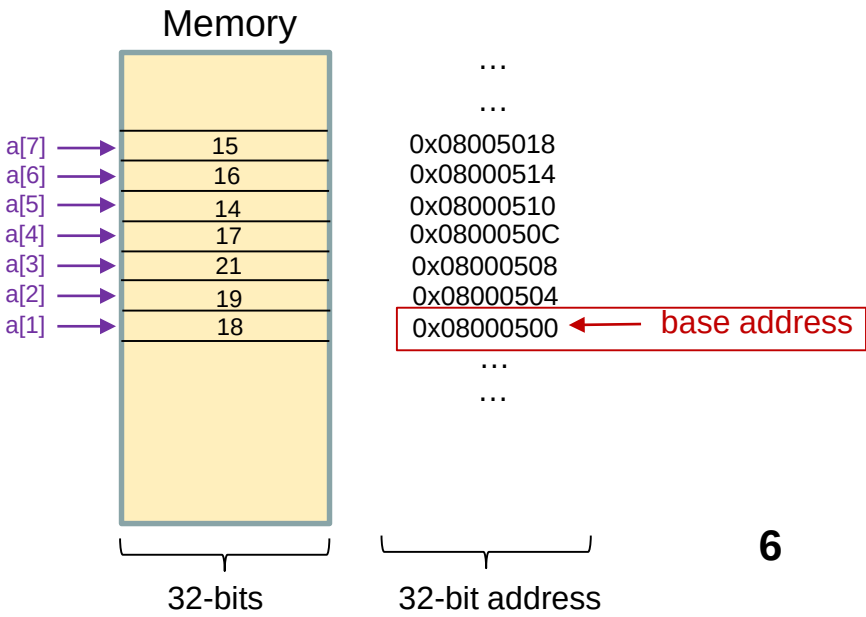


Raspberry Pi Assembler

Arrays

- What is an array?
  - An array is a collection of data of the same type. Examples of types are char, int, double and word. Each type reserves a different amount of space in memory.
  - Example: array **a** of average temperature values during the week. The name of the array is a and it has 7 values. Let **a** = [18 19 21 17 14 16 15]

- What are the properties of an array
  - Base address: address of the first element of the array (eg. 0x080000500)
  - Name of the array: represents the base address



The name of the array is a pointer to the first value of the array

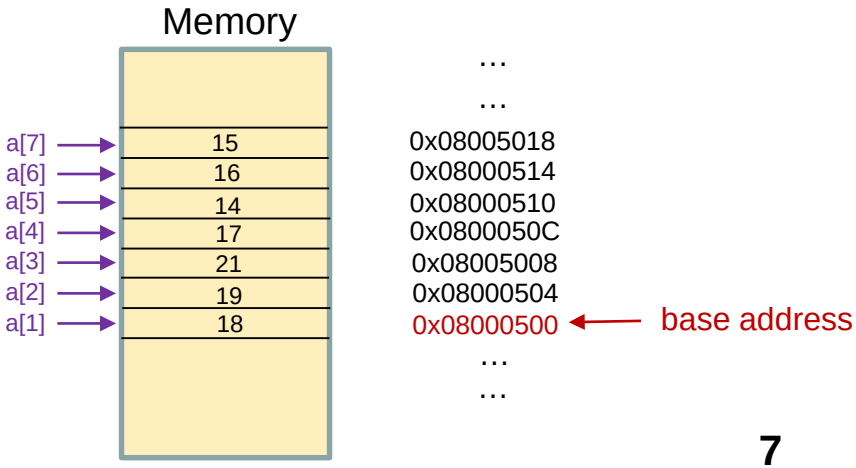
Raspberry Pi Assembler

Arrays: accessing an index of an array

- How do you access the 6th index of an array?
  - In assembly language: In order to access the contents of the 6th index, we first need to calculate the address of the 6th index.

The address of the 6th index is calculated using the following steps:

- Obtain the base address of the array



Raspberry Pi Assembler

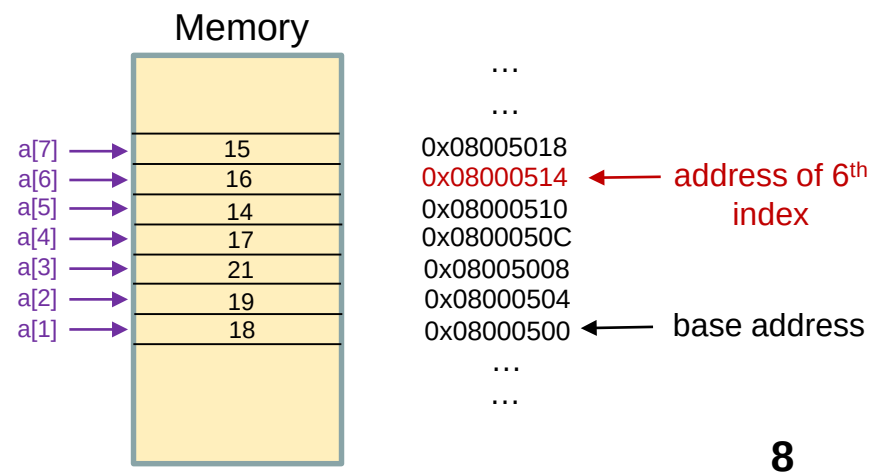
Arrays: accessing an index of an array

- How do you access the 6th index of an array?
  - In assembly language: In order to access the contents of the 6th index, we first need to calculate the address of the 6th index.

The address of the 6th index is calculated using the following steps:

- Obtain the base address of the array
- Calculate the address of the 6th index using the formula:

Address = base address + (6-1) x 4



Raspberry Pi Assembler

Defining arrays

- Declare an array **a** with 100 elements. Each element of the array should be an integer. Perform this declaration in the C language and assembly.

Raspberry Pi Assembler

Defining arrays

- Declare an array **a** with 100 elements. Each element of the array should be an integer. Perform this declaration in the C language and assembly.
  - C language

```
int a[100];
```

Defines an array of 100 elements of type integer. Each element occupies 4 bytes, since one integer is a 32-bit number.

  - Assembly

```
/* -- array01.s */
.data
a: .skip 400
```

- Defines a symbol **a**
- The directive **.skip** tells the assembler to reserve 400 bytes of data for the symbol **a** [Note: 4 x 100 = 400]

Structures

Raspberry Pi Assembler

Structures

- What is a structure?
  - A structure is a collection of data of **different** types that is grouped under a single name in a block of memory
  - Example: structure **b** for a student

```
struct  my_struct
{
        char  Name[50]
        int  ClassMark
        int  ExamMark
}   b;
```

Raspberry Pi Assembler

Structures

- What is a structure?
  - A structure is a collection of data of **different** types that is grouped under a single name in a block of memory
  - Example: structure **b** for a student
- What are the properties of a structure?
  - A structure is made up of many fields

```
struct  my_struct
{
        char  Name[50]
        int  ClassMark
        int  ExamMark
}  b;
```

field 1

field 2

field 3

Define a variable **b** of type my_struct

Raspberry Pi Assembler

Structures

- What is a structure?
  - A structure is a collection of data of **different** types that is grouped under a single name in a block of memory
  - Example: structure **b** for a student

```
struct  my_struct
{
        char  Name[50]
        int  ClassMark
        int  ExamMark
}  b;
```

The name of the structure **b** represents the address of the first field of the structure

- What are the properties of a structure?
  - A structure is made up of many fields
  - The name of the structure represents the the base address

Raspberry Pi Assembler

Defining a structure

- Declare a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. Perform this declaration in the C language and assembly

Raspberry Pi Assembler

Defining a structure

- Declare a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. Perform this declaration in the C language and assembly
  - C language

```
struct my_struct
{
  char f0;
  int f1;
} b;
```

field 1

field 2

Define variable b of type my_struct

Raspberry Pi Assembler

Defining a structure

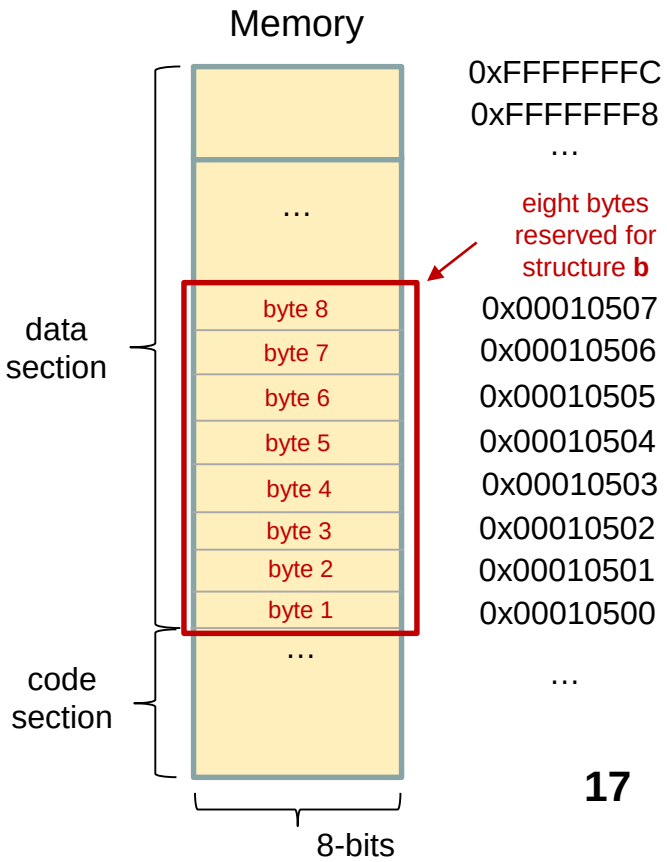
- Declare a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. Perform this declaration in the C language and assembly
  - C language

```
struct my_struct
{
  char f0;
  int f1;
} b;
```

  - Assembly

```
b:  .skip 8
```

- Defines a symbol **b**
- The directive **.skip** tells the assembler to reserve 8 bytes of data for the symbol **b**



Raspberry Pi Assembler

Defining a structure

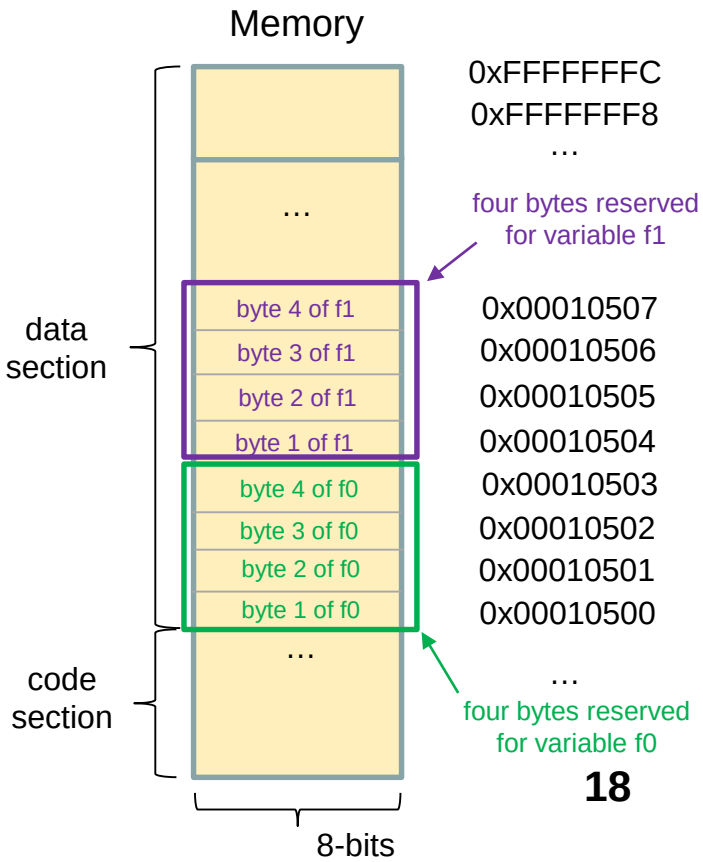
- Declare a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. Perform this declaration in the C language and assembly
  - C language

```
struct my_struct
{
  char f0;
  int f1;
} b;
```

  - Assembly

```
b:  .skip 8
```

- Defines a symbol **b**
- The directive **.skip** tells the assembler to reserve 8 bytes of data for the symbol **b**
- **Note**: each field needs to be start at a 4 byte boundary. So four bytes are reserved so both f0 and f1



Raspberry Pi Assembler

Defining a structure

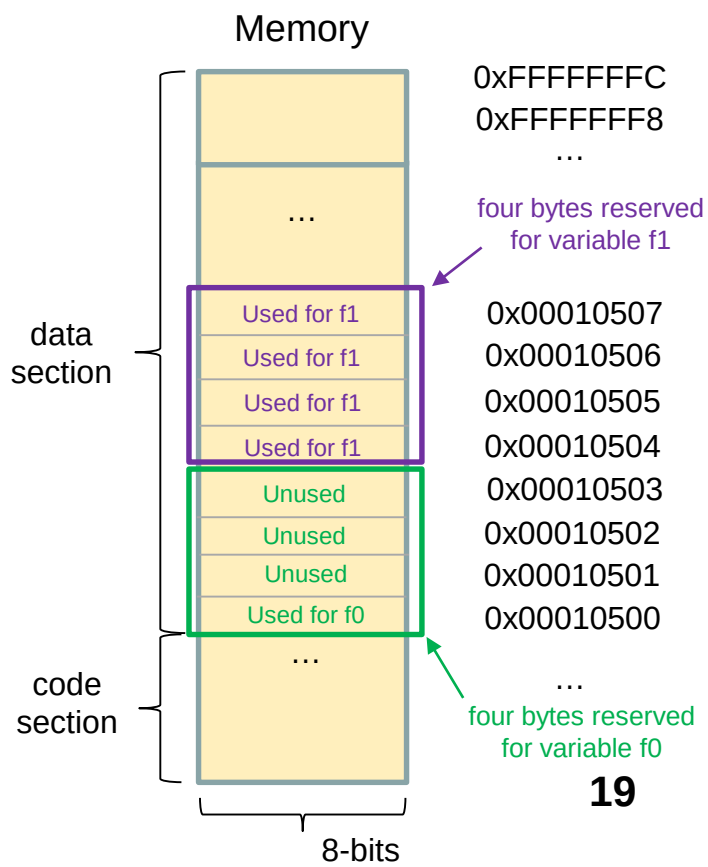
- Declare a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. Perform this declaration in the C language and assembly
  - C language

```
struct my_struct
{
  char f0;
  int f1;
} b;
```

  - Assembly

```
b:  .skip 8
```

- Defines a symbol **b**
- The directive **.skip** tells the assembler to reserve 8 bytes of data for the symbol **b**
- **Note**: each field needs to be start at a 4 byte boundary. So four bytes are reserved so both f0 and f1
- Since f0 is of type char, only one byte is used. The other three bytes are unused.



Addressing Modes and Indexing Modes for Arrays

Raspberry Pi Assembler

Addressing Modes and Indexing Modes for Arrays

- What is meant by addressing modes and indexing modes?
  - Addressing modes refers to the different ways that an address in Memory can be computed.

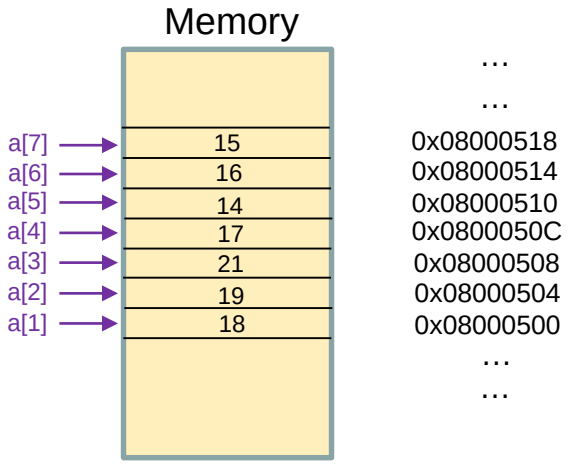


Illustration of array **a** in Memory

Raspberry Pi Assembler

Addressing Modes and Indexing Modes for Arrays

- What is meant by addressing modes and indexing modes?
  - Addressing modes refers to the different ways that an address in Memory can be computed.
    - Let's take one example: the address of the 4th index of **a** can be obtained as:

Offset mode: base address + $\underbrace{(4-1) \times 4}_{\text{offset}}$

offset → offset either specified as an immediate value or the contents of a CPU register

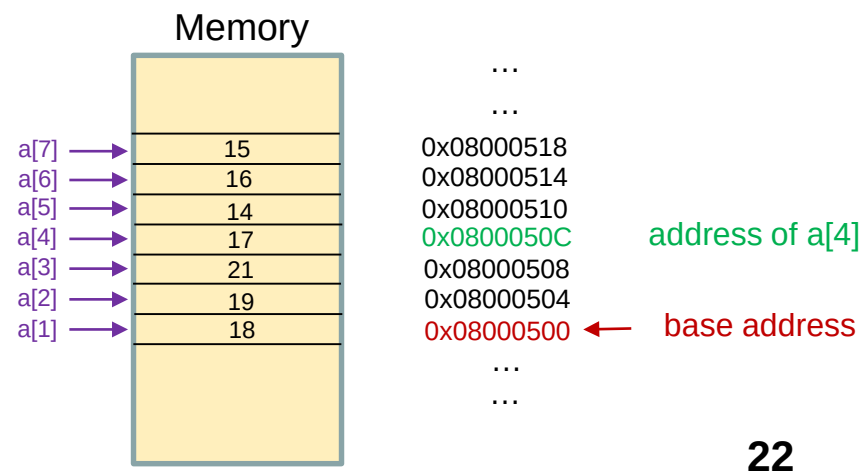


Illustration of array **a** in Memory

Raspberry Pi Assembler

Addressing Modes and Indexing Modes for Arrays

- What is meant by addressing modes and indexing modes?
  - Addressing modes refers to the different ways that an address in Memory can be computed.
    - Let's take one example: the address of the 4th index of **a** can be obtained as:

Offset mode: base address + $\underbrace{(4-1) \times 4}_{\text{offset}}$

offset → offset either specified as an immediate value or the contents of a CPU register

  - Indexing modes refers to assembly instructions that use addressing modes to compute the memory address and performs an assembly operation.

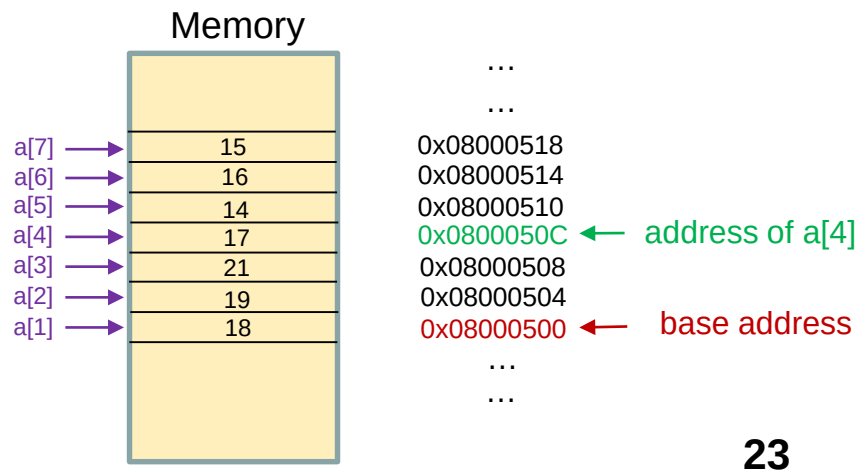


Illustration of array **a** in Memory

Raspberry Pi Assembler

Addressing Modes and Indexing Modes for Arrays

- Let's consider the following program:
  - Define an array of 100 elements. Thereafter, write the index value into each index of the array, ie. a[0] = 0, a[1] = 1, a[2] = 2. Let the first index equal to 0.

```
for (i = 0; i < 100; i++)
    {  a[i] = i;
    }
```

Equivalent high level C code of the program

- We will look at two approaches of implementing this program:
  - Non-indexing modes: the memory address is directly specified either as an immediate value of the contents of a CPU register. The address does not need to be computed in the same assembly instruction that the operation is specified
  - Indexing modes: in one assembly instruction, both the memory address needs to be computed and an assembly instruction needs to be performed. Typically the memory address is obtained by adding a base address to an offset.

Non-indexing mode

Raspberry Pi Assembler
Non-indexing mode

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

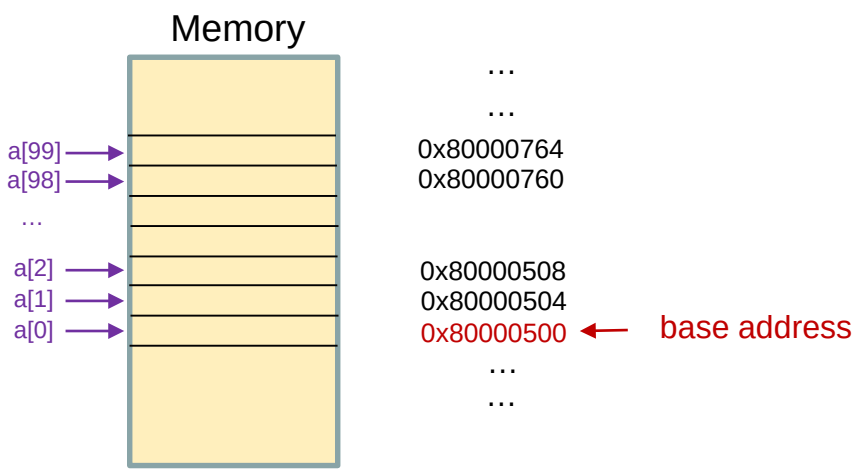
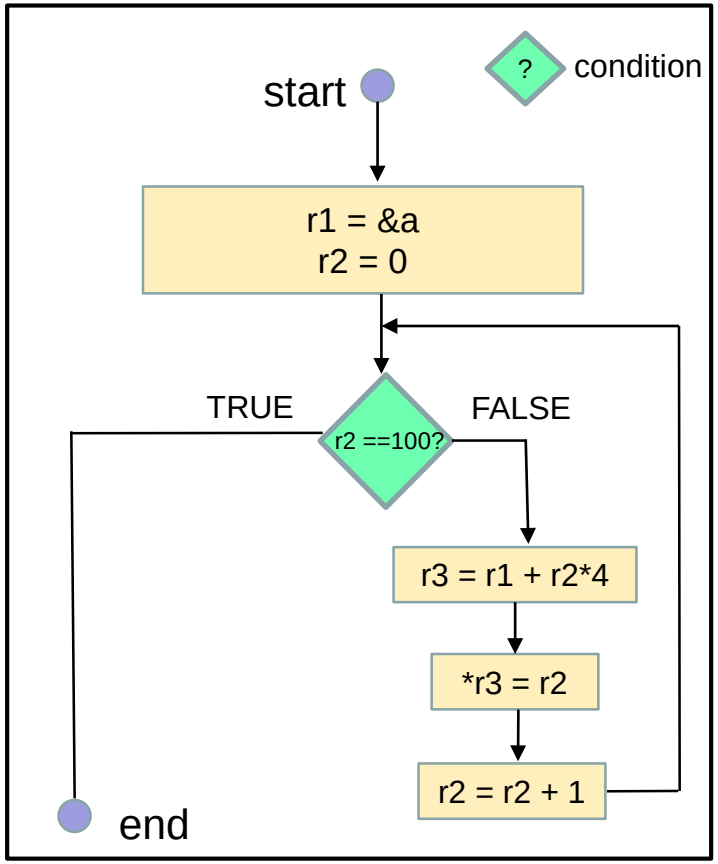


Illustration of array **a** in Memory

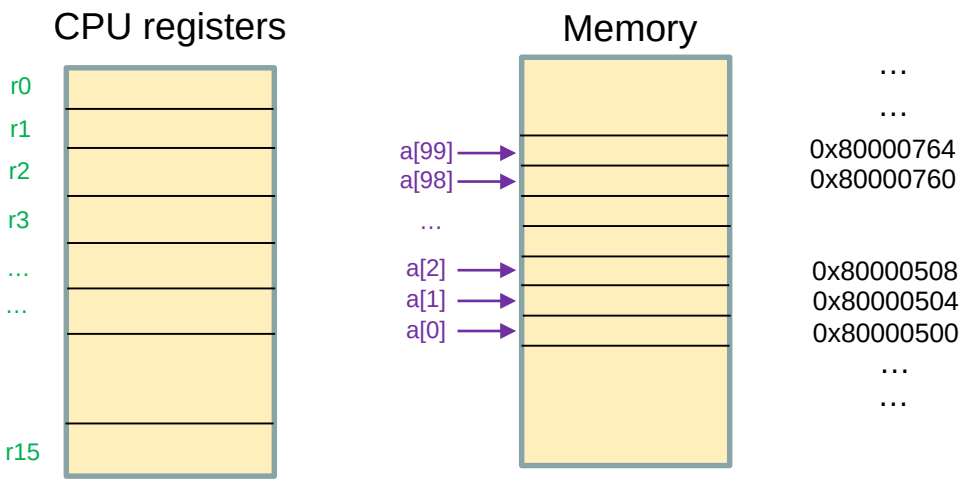


Flow chart of the program

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index




```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

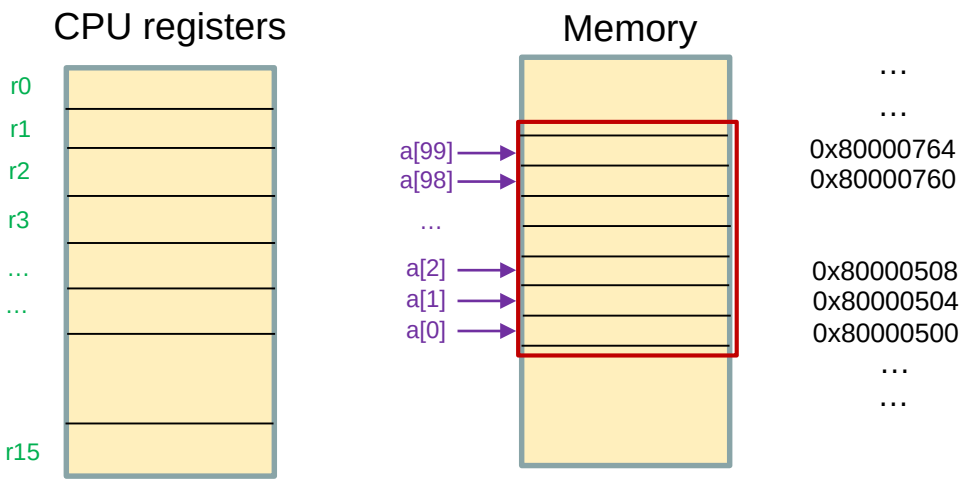
Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Reserves 400 bytes in memory for array **a**

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

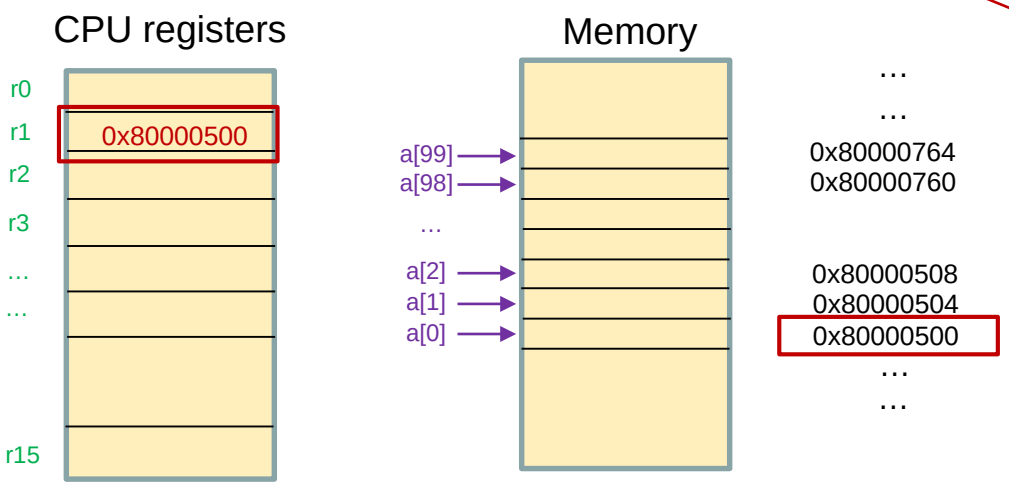


Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Loads the address of the first index of **a** into r1



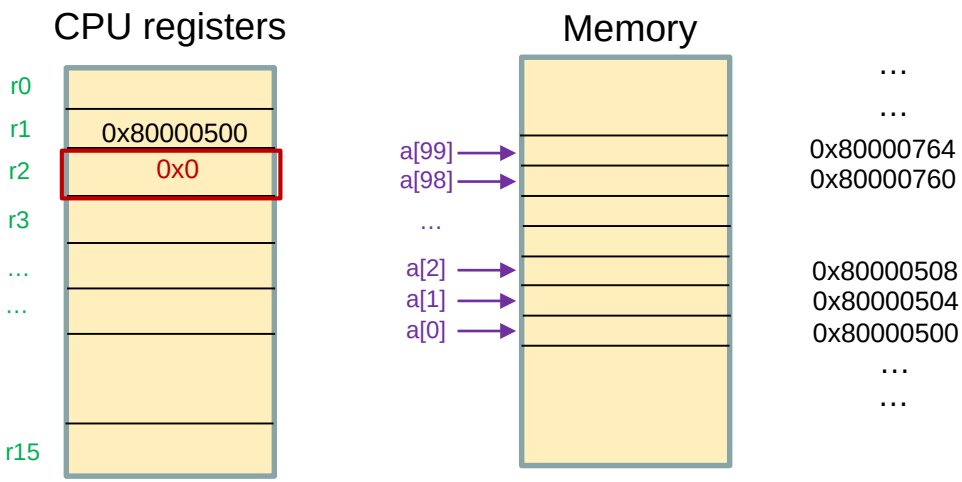

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

moves the value 0 into r2



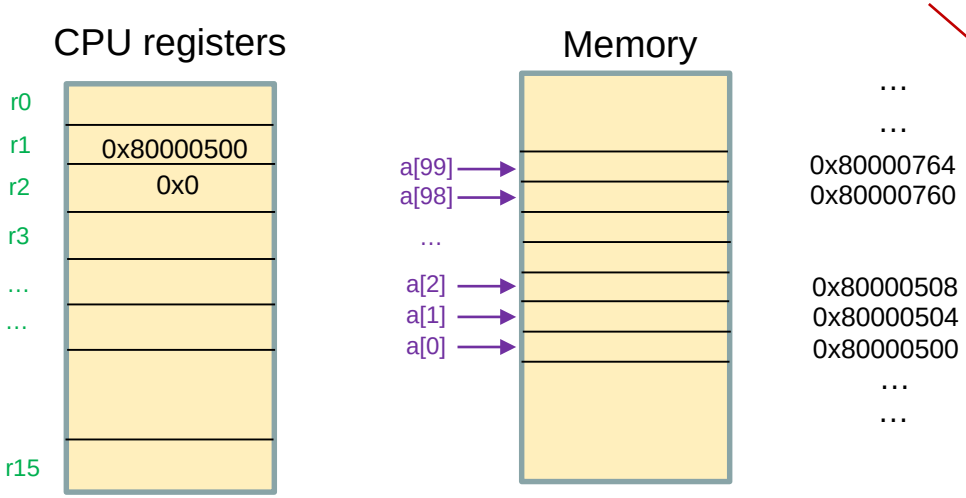

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

computes r2 – 100 and updates the CPSR. Since (r2-100) ≠ 0, the Z flag is cleared to 0



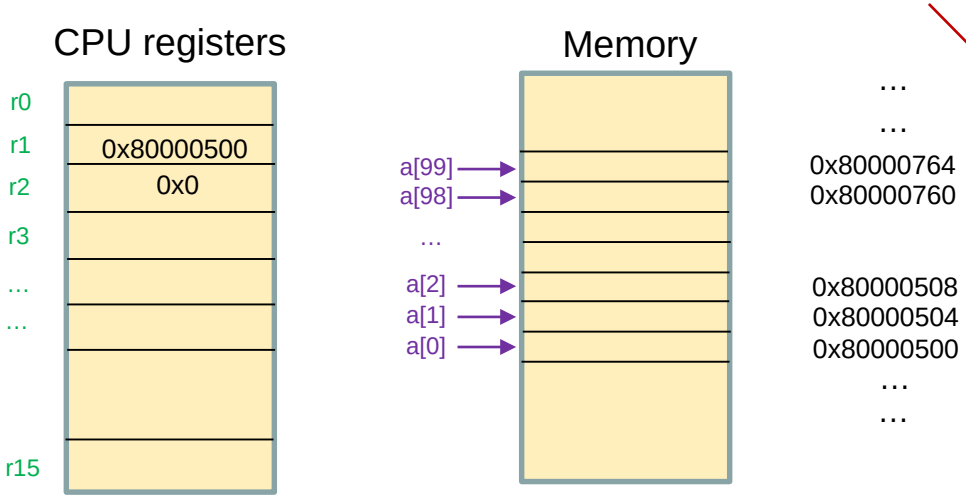

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Since Z ≠ 1, the branch is not executed

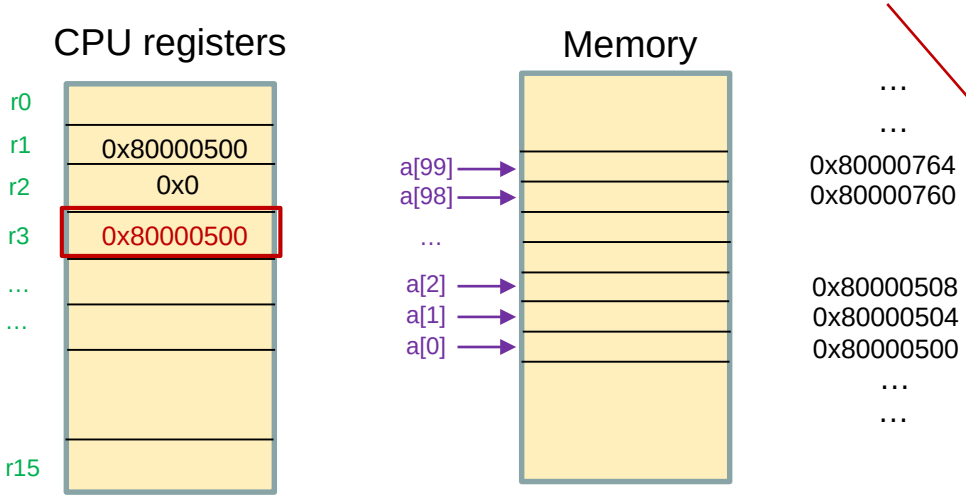



```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index



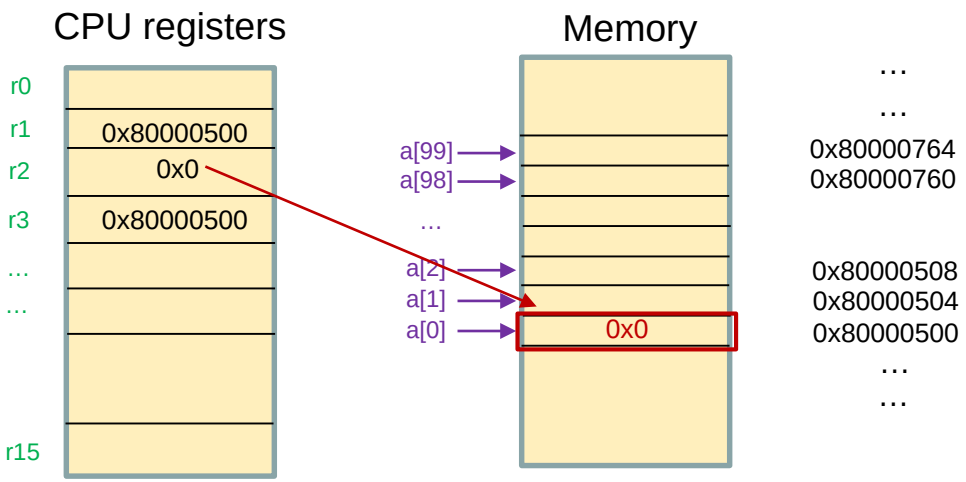
r3 = r1 + 0

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

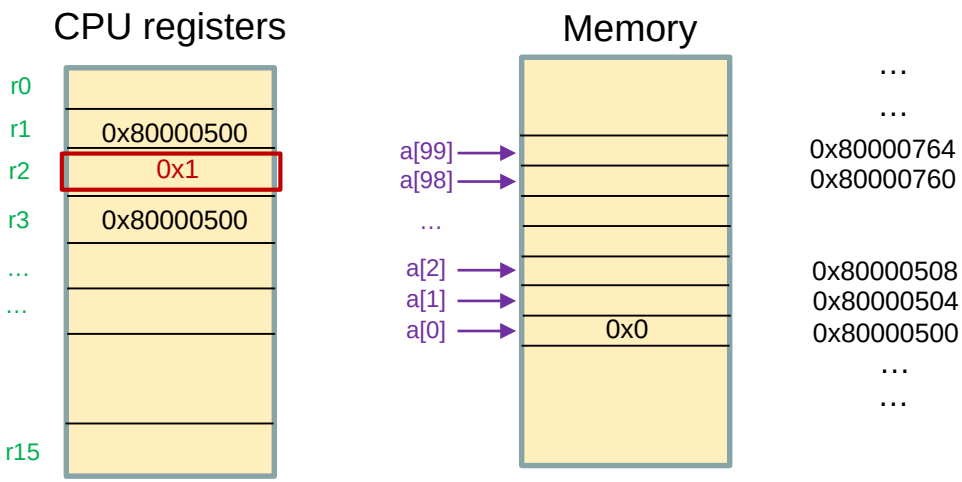



```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a              @ r1 <- &a
    mov r2, #0              @ r2 <- 0

Loop:
    cmp r2, #100            @ Have we reached 100 yet?
    beq end                 @ If so, leave the loop
    add r3, r1, r2, LSL #2  @ r3 <- r1 + (r2*4)
    str r2, [r3]            @ *r3 <- r2
    add r2, r2, #1          @ r2 <- r2 + 1
    b   Loop                @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index



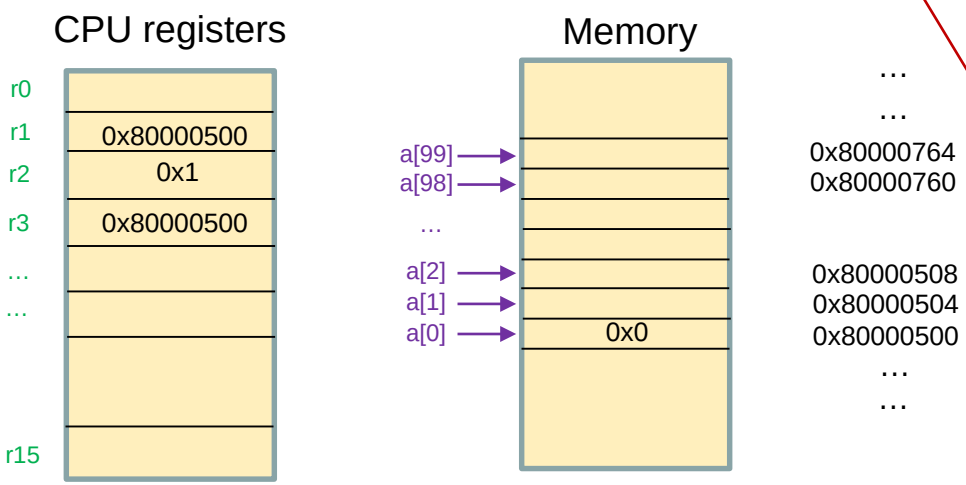

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler
Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Unconditional branch to label Loop

```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

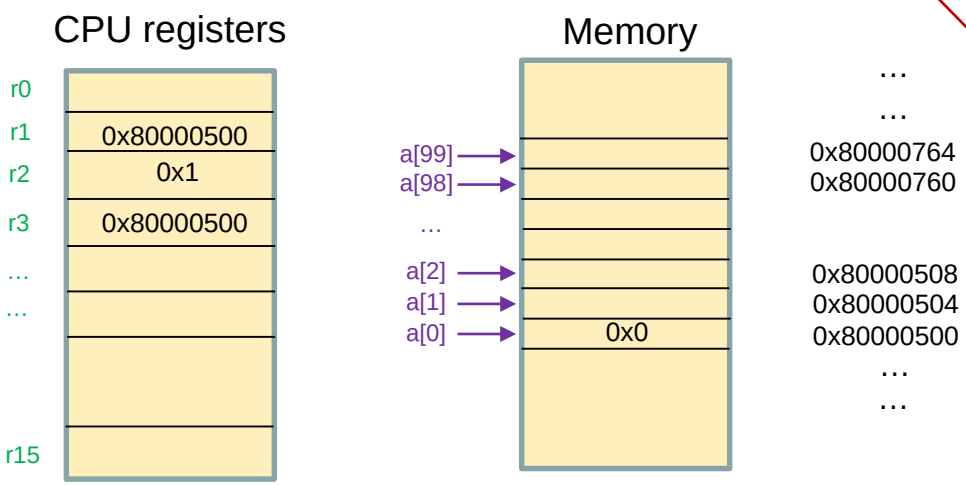


Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

computes r2 – 100 and updates the CPSR. Since (r2-100) ≠ 0, the Z flag is cleared to 0

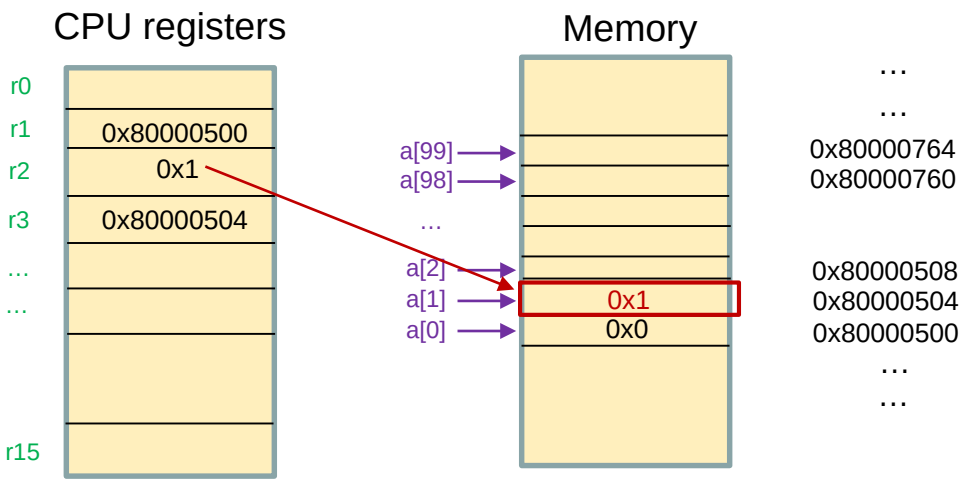



```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

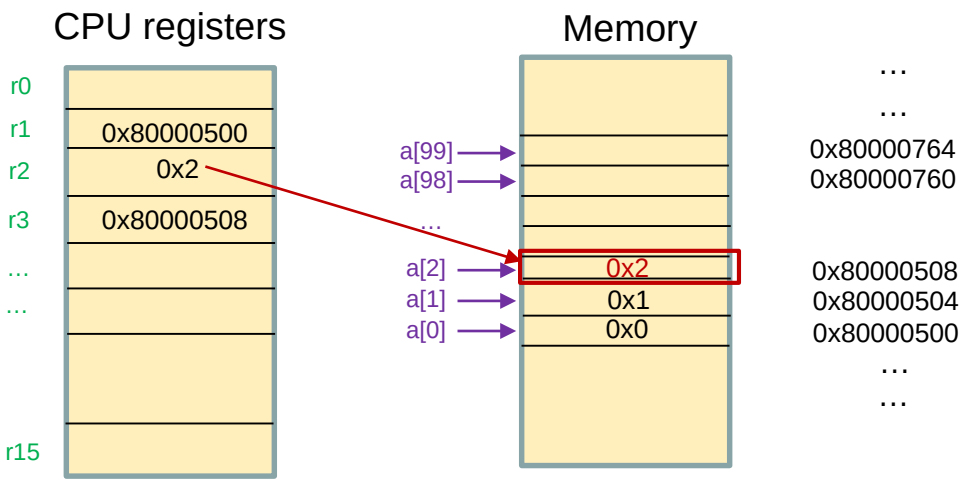



```
    /* -- array01.s */
    .data
    a: .skip 400
    .text
    .global main
    main:
        ldr r1, =a                  @ r1 <- &a
        mov r2, #0                  @ r2 <- 0

 Loop:
continue cmp r2, #100               @ Have we reached 100 yet?
        beq end                     @ If so, leave the loop
        add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
        str r2, [r3]                @ *r3 <- r2
        add r2, r2, #1              @ r2 <- r2 + 1
        b   Loop                    @ Goto beginning of the Loop
    end:
        bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index



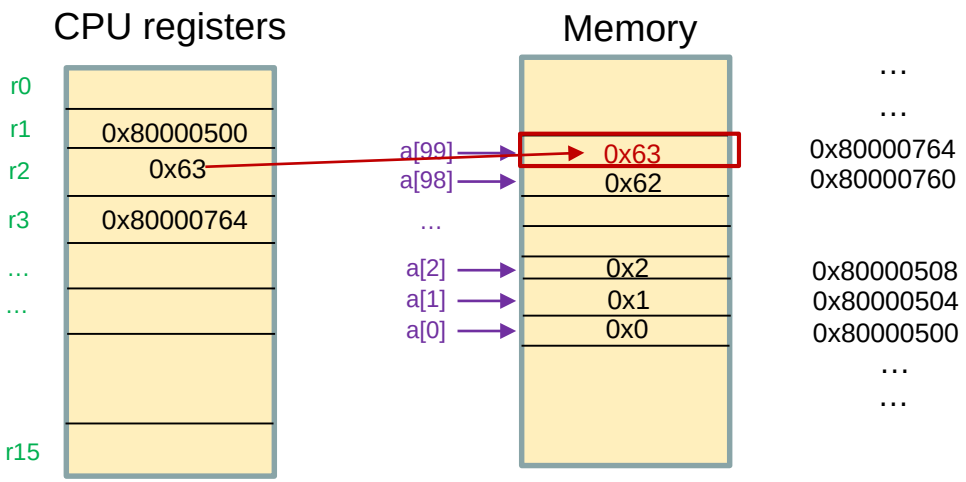

```
    /* -- array01.s */
    .data
    a: .skip 400
    .text
    .global main
    main:
        ldr r1, =a                  @ r1 <- &a
        mov r2, #0                  @ r2 <- 0

 Loop:
continue cmp r2, #100               @ Have we reached 100 yet?
        beq end                     @ If so, leave the loop
        add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
        str r2, [r3]                @ *r3 <- r2
        add r2, r2, #1              @ r2 <- r2 + 1
        b   Loop                    @ Goto beginning of the Loop
    end:
        bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Note: the decimal value 99 is equivalent to the hex value 0x63



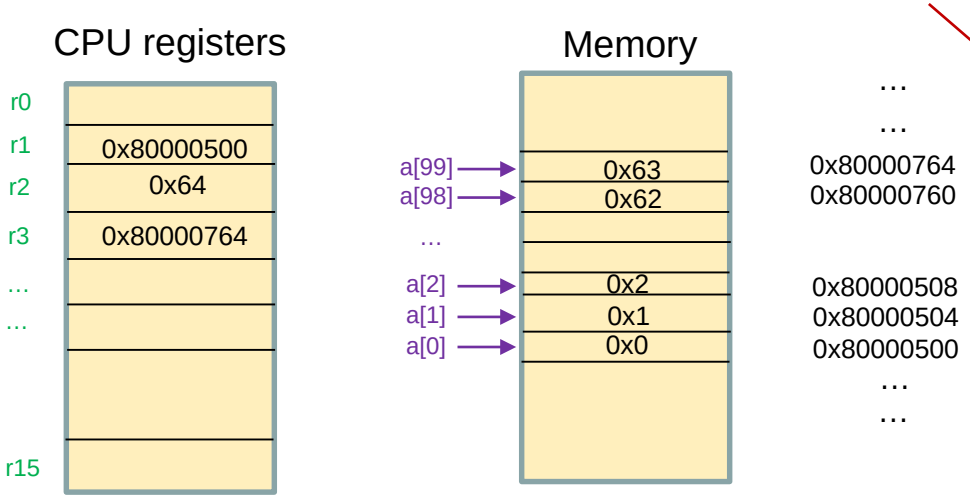

```
    /* -- array01.s */
    .data
    a: .skip 400
    .text
    .global main
    main:
        ldr r1, =a                  @ r1 <- &a
        mov r2, #0                  @ r2 <- 0

 Loop:
continue cmp r2, #100               @ Have we reached 100 yet?
        beq end                     @ If so, leave the loop
        add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
        str r2, [r3]                @ *r3 <- r2
        add r2, r2, #1              @ r2 <- r2 + 1
        b   Loop                    @ Goto beginning of the Loop
    end:
        bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

computes r2 – 100 and updates the CPSR. Since (r2-100) == 0, the Z flag is set to 1




```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

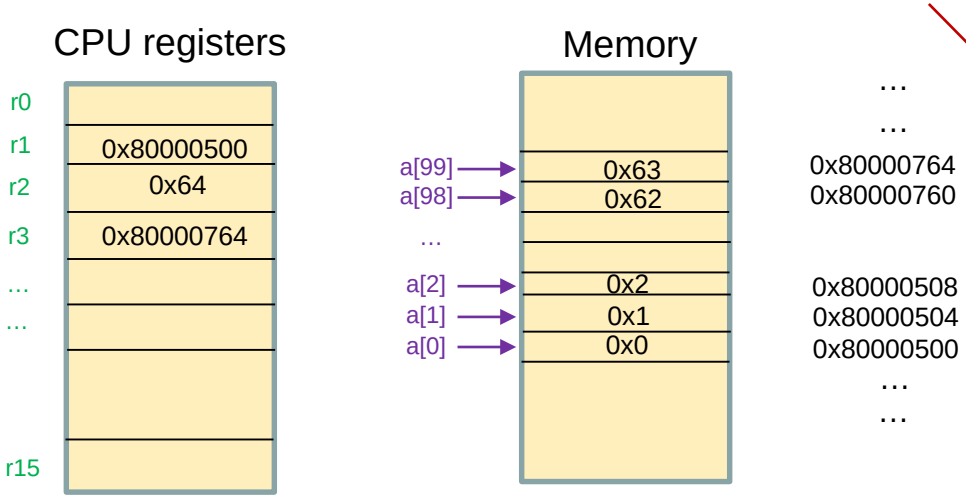
continue

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Since Z == 1, the branch is executed

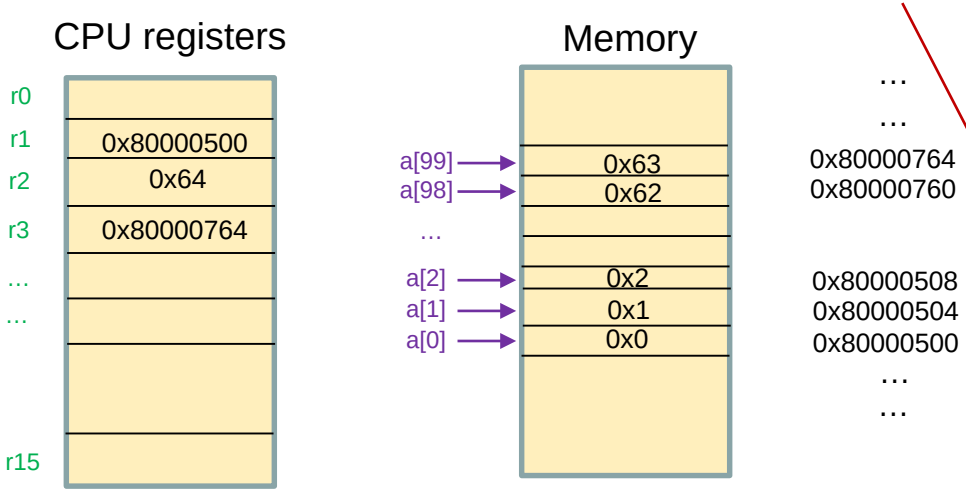



```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index



program terminates

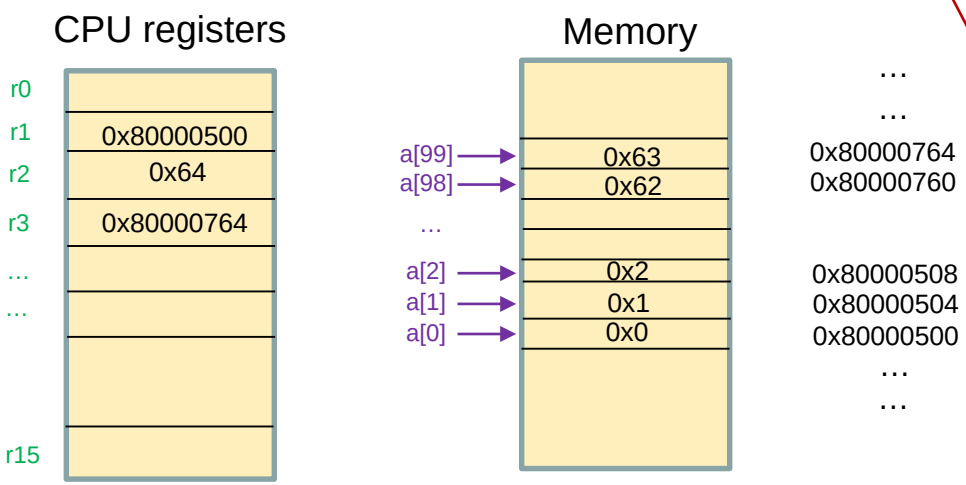
```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Raspberry Pi Assembler

Non-indexing mode

- Assembly program:
  - Variables used:
    - r1 : address of the first index of array **a**
    - r2 : index of the current value
    - r3 : memory address of the current index

Using ARM instructions with indexing modes, we can replace these three lines of code with one assembly instruction




```
/* -- array01.s */
.data
a: .skip 400
.text
.global main
main:
    ldr r1, =a                  @ r1 <- &a
    mov r2, #0                  @ r2 <- 0

Loop:
    cmp r2, #100                @ Have we reached 100 yet?
    beq end                     @ If so, leave the loop
    add r3, r1, r2, LSL #2      @ r3 <- r1 + (r2*4)
    str r2, [r3]                @ *r3 <- r2
    add r2, r2, #1              @ r2 <- r2 + 1
    b   Loop                    @ Goto beginning of the Loop
end:
    bx  lr
```

Indexing modes

Raspberry Pi Assembler

Indexing modes

- ARM provides different indexing modes
  - Non-updating indexing modes
  - Updating indexing modes
  - Post-indexing modes
  - Pre-indexing modes

Non-updating indexing mode

Raspberry Pi Assembler

Non-updating indexing modes

- In non-updating indexing modes, the base address and an offset are used to compute the final memory address. After the instruction has executed, the contents of the CPU register that contains the base address does not change value.

- Recap: syntax of ARM instructions

```
instruction Rdest, Rsource1
```

- Syntax of non-updating indexing mode 1

```
instruction Rdest, [Resource1, #±immediate]
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 1**

```
instruction Rdest, [Resource1, #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
str r2, [r1, #+12]  @ *(r1 + 12) <- r2
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 1**

```
instruction Rdest, [Resource1, #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

base address

offset (maximum value is 4096)

```
mov r2, #3          @ r2 <- 3
str r2, [r1, #+12]  @ *(r1 + 12) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + 12
- Then perform the store operation using the computed memory address

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 1**

```
instruction Rdest, [Resource1, #±immediate]
```

- Example: set a[3] to 3

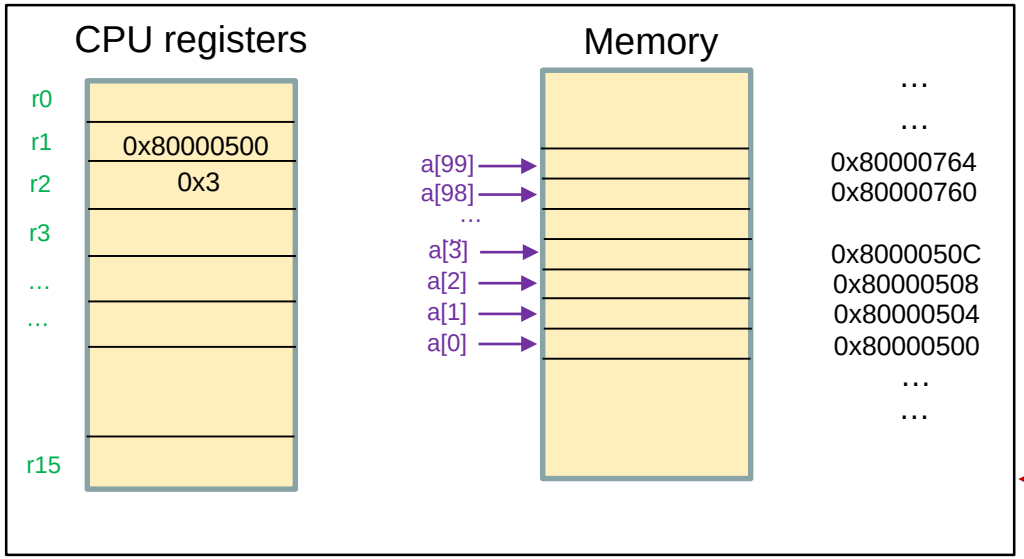
Assume: r1 already contains the address of the first field of array **a**

base address

offset (maximum value is 4096)

```
mov r2, #3          @ r2 <- 3
str r2, [r1, #+12]  @ *(r1 + 12) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + 12
- Then perform the store operation using the computed memory address



Before str r2, [r1, #12] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 1**

```
instruction Rdest, [Resource1, #±immediate]
```

- Example: set a[3] to 3

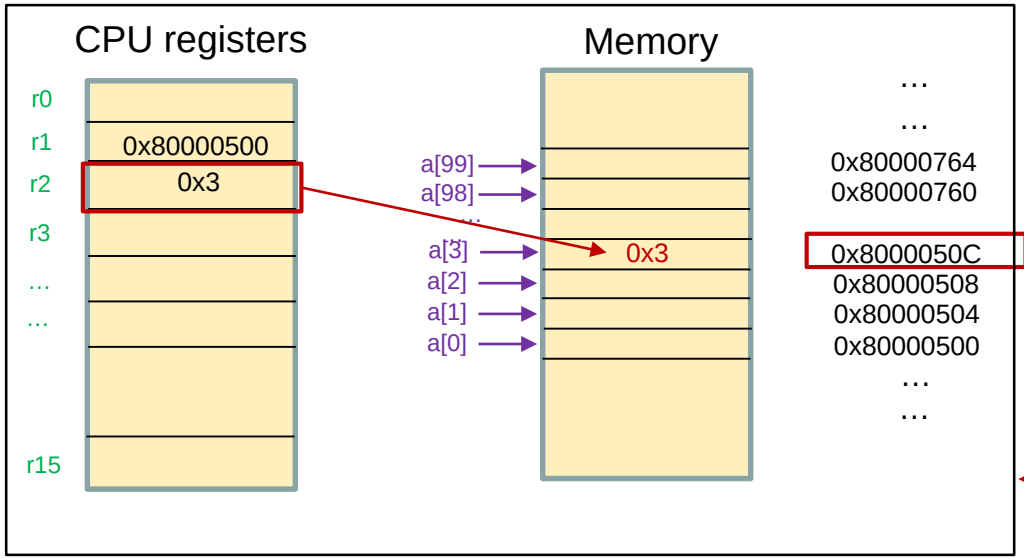
Assume: r1 already contains the address of the first field of array **a**

base address

offset (maximum value is 4096)

```
mov r2, #3          @ r2 <- 3
str r2, [r1, #+12]  @ *(r1 + 12) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + 12
- Then perform the store operation using the computed memory address



r1 + 12 = 0x8000050C

After str r2, [r1, #12] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 1**

```
instruction Rdest, [Resource1, #±immediate]
```

- Example: set a[3] to 3

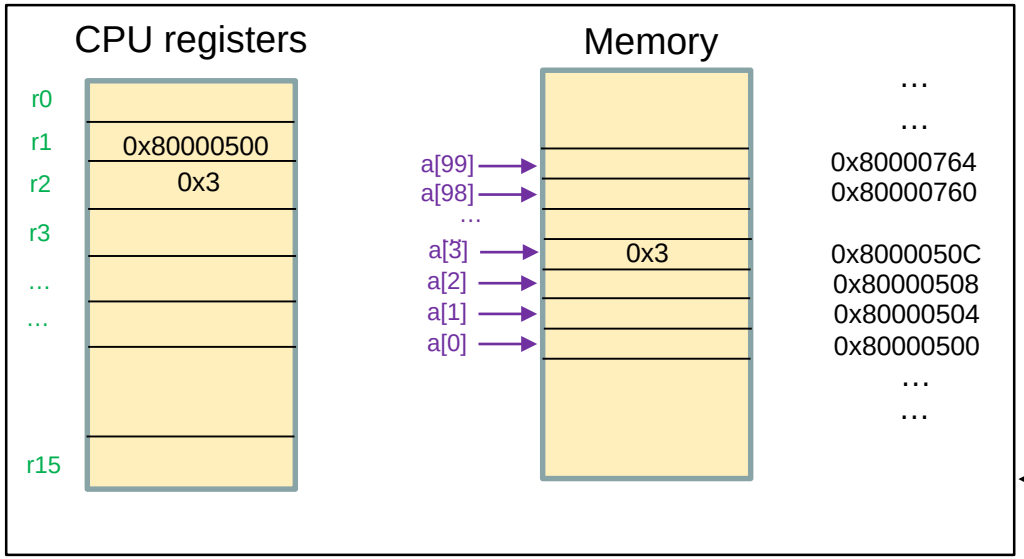
Assume: r1 already contains the address of the first field of array **a**

base address

offset (maximum value is 4096)

```
mov r2, #3          @ r2 <- 3
str r2, [r1, #+12]  @ *(r1 + 12) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + 12
- Then perform the store operation using the computed memory address



After str r2, [r1, #12] has executed

Feature of non-updating indexing mode

After the instruction has executed, the contents of r1 does not change

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

offset value contained in a CPU register. Not limited to an offset value of 4095 like in mode 1

```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

offset value contained in a CPU register. Not limited to an offset value of 4095 like in mode 1

```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + r3
- Then perform the store operation using the computed memory address

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

offset value contained in a CPU register. Not limited to an offset value of 4095 like in mode 1

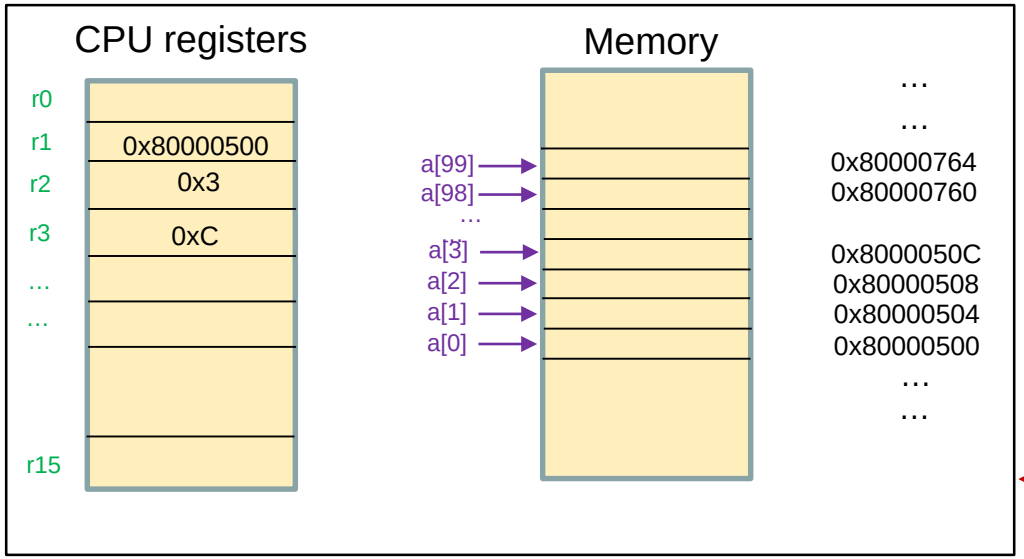
```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + r3
- Then perform the store operation using the computed memory address



Before str r2, [r1, +r3] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

offset value contained in a CPU register. Not limited to an offset value of 4095 like in mode 1

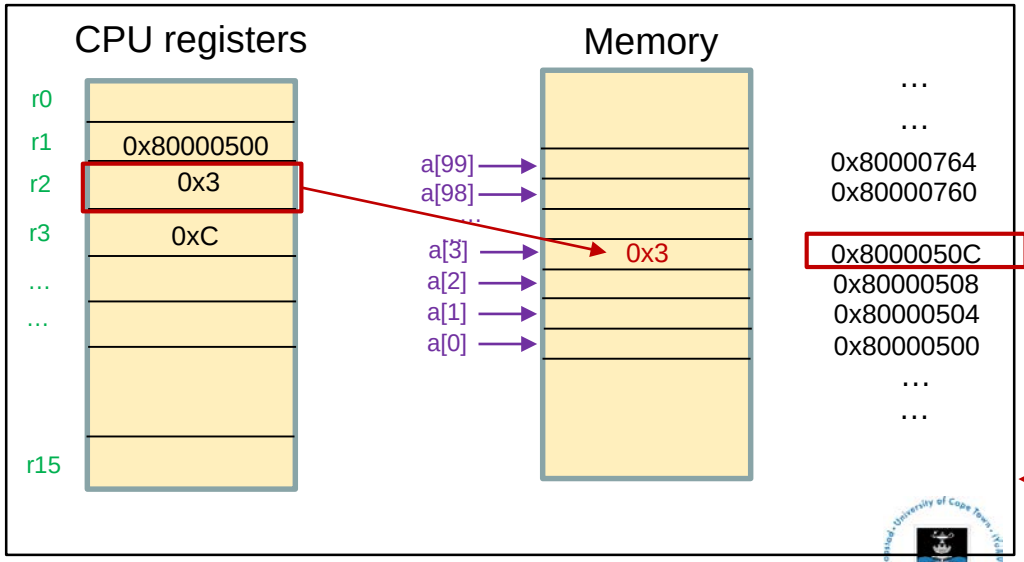
```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + r3
- Then perform the store operation using the computed memory address



r1 + r3 = 0x8000050C

After str r2, [r1, +r3] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 2**

offset value contained in a CPU register. Not limited to an offset value of 4095 like in mode 1

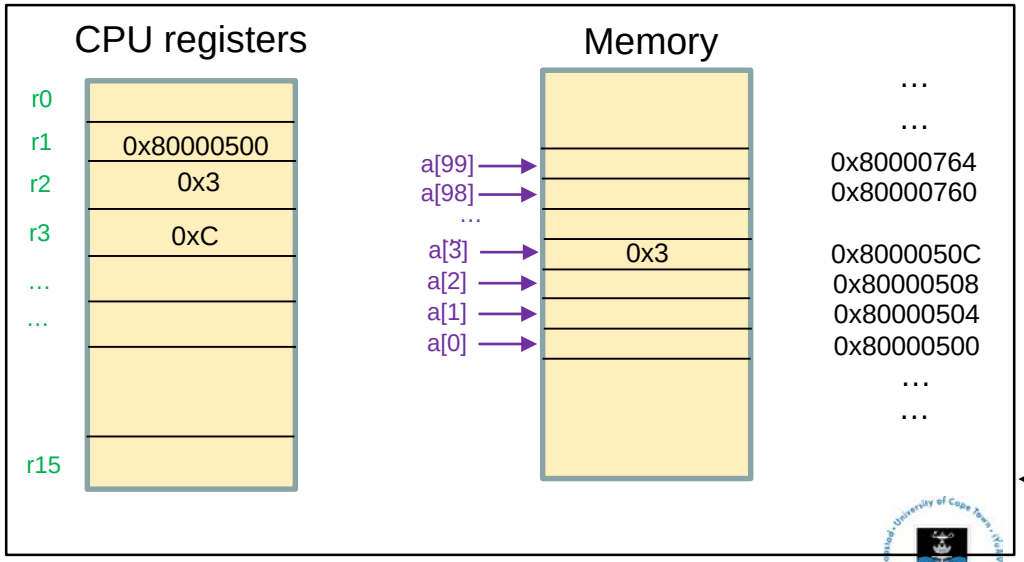
```
instruction Rdest, [Rsource1, ±Rsource2]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**

```
mov r2, #3          @ r2 <- 3
mov r3, #12         @ r3 <-12
str r2, [r1, +r3]   @ *(r1 + r3) <- r2
```

- First compute address: sum of base address and offset. This is given by r1 + r3
- Then perform the store operation using the computed memory address



After str r2, [r1, +r3] has executed

Feature of non-updating indexing mode

After the instruction has executed, the contents of r1 and r3 do not change

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]  @ *(r1 + r2*4) <- r2
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]  @ *(r1 + r2*4) <- r2
```

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]   @ *(r1 + r2*4) <- r2
```

- First compute the offset: LSL #2 of r2
- Then, compute the address: sum of base address and offset
- Then perform the store operation using the computed memory address

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]   @ *(r1 + r2*4) <- r2
```

- First compute the offset: LSL #2 of r2
- Then, compute the address: sum of base address and offset
- Then perform the store operation using the computed memory address

For non-updating indexing mode 3: don't need to use r3

- r1 : address of the first element of array **a**
- r2 : index of the current value
- ~~r3 : memory address of the current index~~

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

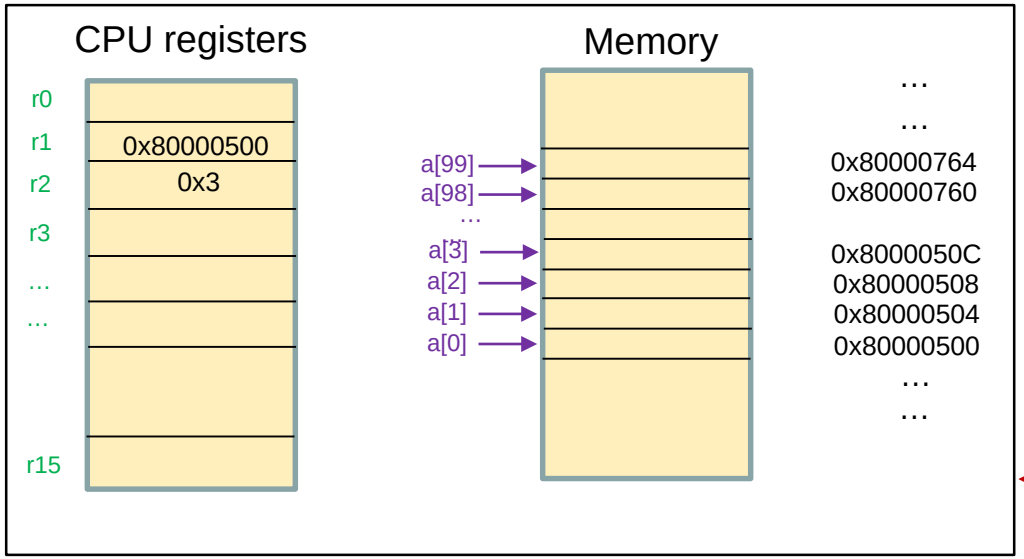
```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]   @ *(r1 + r2*4) <- r2
```

- First compute the offset: LSL #2 of r2
- Then, compute the address: sum of base address and offset
- Then perform the store operation using the computed memory address



Before str r2, [r1, +r2, LSL #2] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

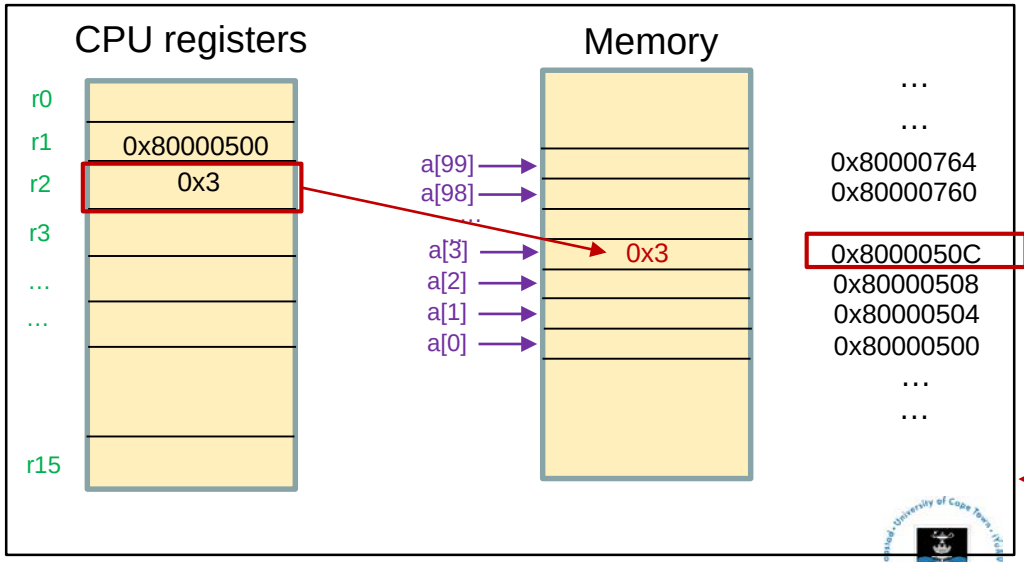
```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]   @ *(r1 + r2*4) <- r2
```

- First compute the offset: LSL #2 of r2
- Then, compute the address: sum of base address and offset
- Then perform the store operation using the computed memory address



r1 + 4 x r2 = 0x8000050C

After str r2, [r1, +r2, LSL #2] has executed

Raspberry Pi Assembler

Non-updating indexing modes

- **Non-updating indexing mode 3**

offset value needs to be computed first

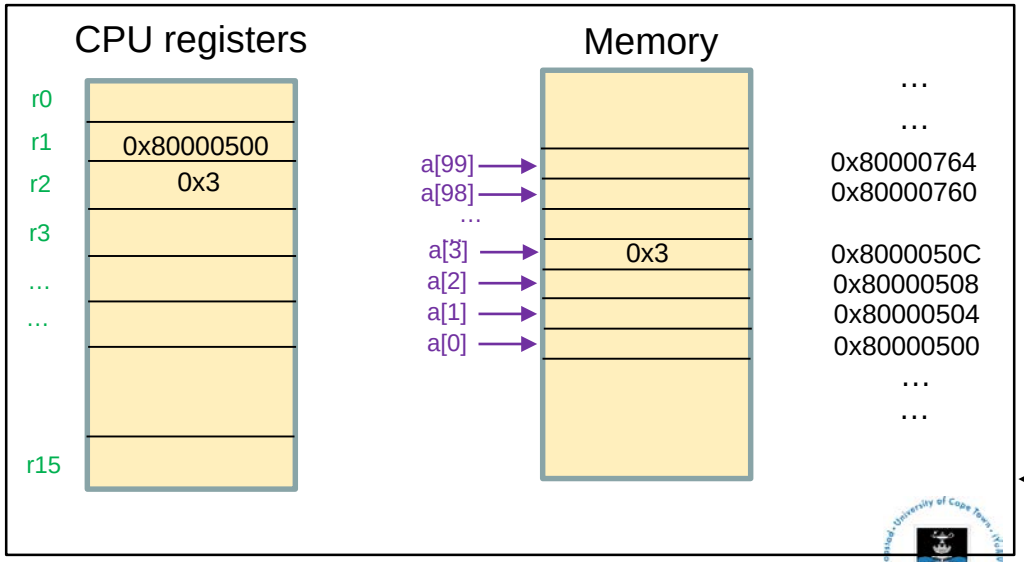
```
instruction Rdest, [Rsource1, ±Rsource2, shift_operation #±immediate]
```

- Example: set a[3] to 3

Assume: r1 already contains the address of the first field of array **a**, and r2 has the value 3

```
str r2, [r1, +r2, LSL #2]   @ *(r1 + r2*4) <- r2
```

- First compute the offset: LSL #2 of r2
- Then, compute the address: sum of base address and offset
- Then perform the store operation using the computed memory address



Feature of non-updating indexing mode

After the instruction has executed, the contents of r1 and r2 do not change

After str r2, [r1, +r2, LSL #2] has executed

Updating indexing mode

Raspberry Pi Assembler

Updating Indexing Modes

- Recap: we looked at the following program earlier :
  - Define an array of 100 elements. Thereafter, write the index value into each index of the array, ie. a[0] = 0, a[1] = 1, a[2] = 2. Let the first index equal to 0.

```
for (i = 0; i < 100; i++)
    {  a[i] = i;
    }
```

Equivalent high level C code of the program

- Let's look at another approach to implement the program using assembly instructions that use updating indexing modes:

Raspberry Pi Assembler

Updating Indexing Modes

- In updating indexing modes, the base address and an offset are used to compute the final memory address. After the instruction has executed, the contents of the CPU register that contains the base address does change value.

- We will consider two updating indexing modes:
  - post-indexing mode
  - pre-indexing mode

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

```
instruction Rdest, [Rsource1], #±immediate
```

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

```
instruction Rdest, [Rsource1], #±immediate
```

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

```
instruction Rdest, [Rsource1], #±immediate
```

Example: str r2, [r1], #4

- First the store operation is performed using the memory address contained in r1
- Then, r1 is updated: r1 = r1 + 4

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

```
instruction Rdest, [Rsource1], #±immediate
```

Post-indexing performs two 'simple' instructions in one:

```
str   r2,  [r1]
add   r1,  r1,  #4
```

Example: `str r2, [r1], #4`

- First the store operation is performed using the memory address contained in r1
- Then, r1 is updated: r1 = r1 + 4

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

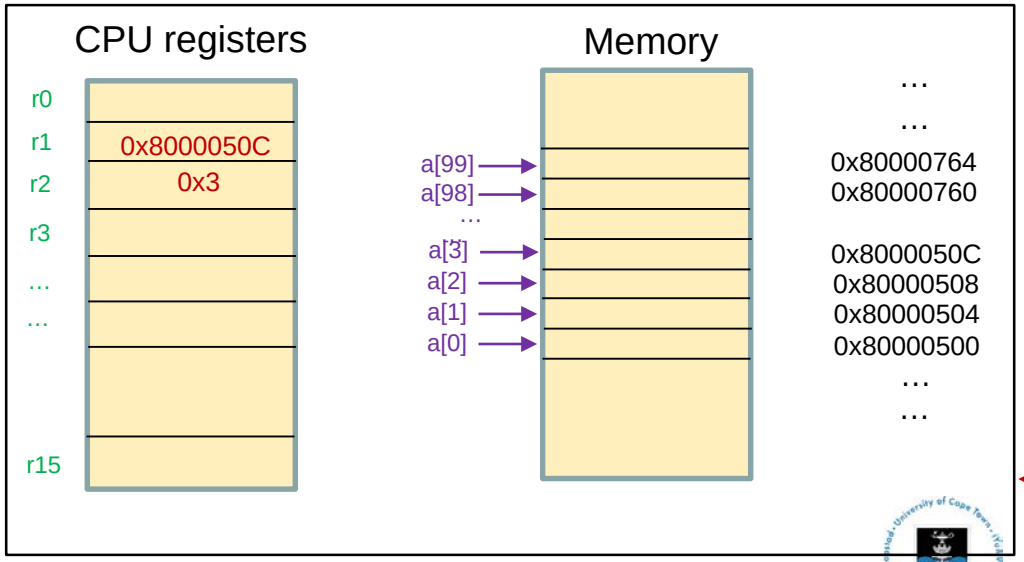
```
instruction Rdest, [Rsource1], #±immediate
```

Post-indexing performs two 'simple' instructions in one:

```
str   r2, [r1]
add   r1, r1, #4
```

Example: `str r2, [r1], #4`

- First the store operation is performed using the memory address contained in r1
- Then, r1 is updated: r1 = r1 + 4



Before `str r2, [r1], #4` has executed

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

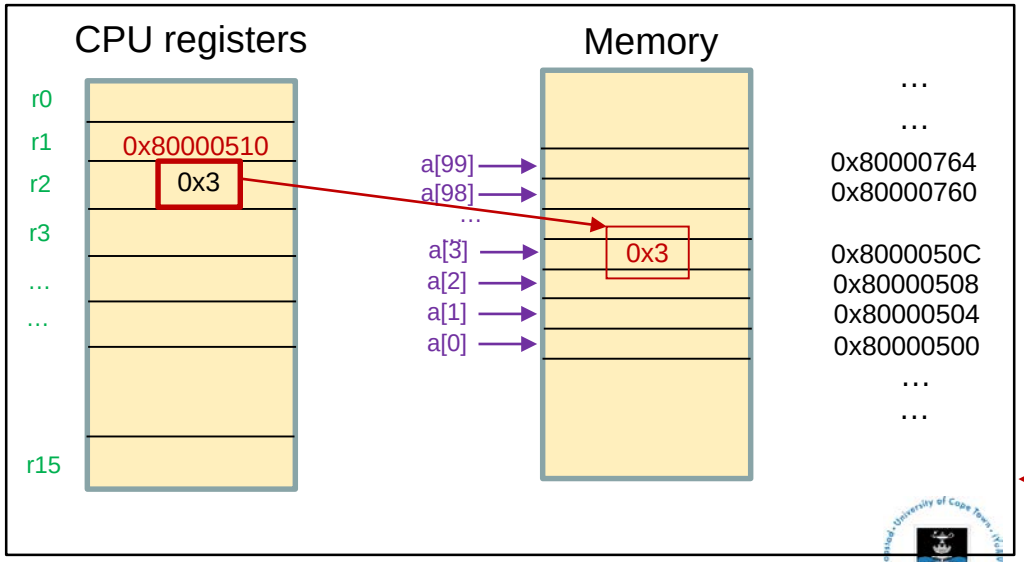
```
instruction Rdest, [Rsource1], #±immediate
```

Post-indexing performs two 'simple' instructions in one:

```
str   r2, [r1]
add   r1, r1, #4
```

Example: `str r2, [r1], #4`

- First the store operation is performed using the memory address contained in r1
- Then, r1 is updated: r1 = r1 + 4



After `str r2, [r1], #4` has executed

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- **Post indexing mode**

offset value specified as an immediate value. Limited to an offset value of 4095

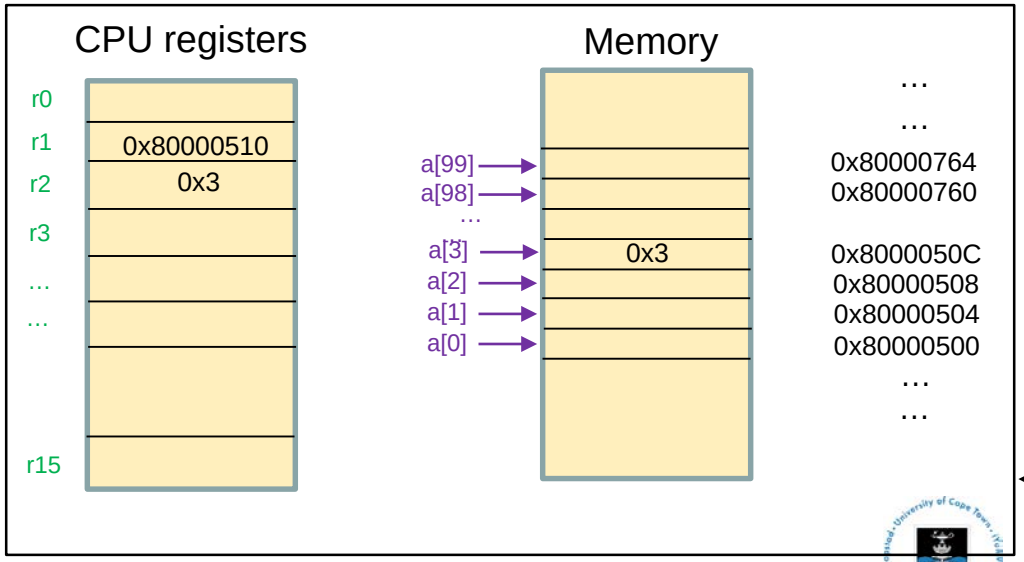
```
instruction Rdest, [Rsource1], #±immediate
```

Post-indexing performs two 'simple' instructions in one:

```
str   r2, [r1]
add   r1, r1, #4
```

Example: `str r2, [r1], #4`

- First the store operation is performed using the memory address contained in r1
- Then, r1 is updated: r1 = r1 + 4



After `str r2, [r1], #4` has executed

Features of post-indexing mode

- The address is updated post, ie. after, the operation is performed
- After the instruction has executed the value of r1 changes

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : memory of the current index of array **a**
    - r2 : index of the current value

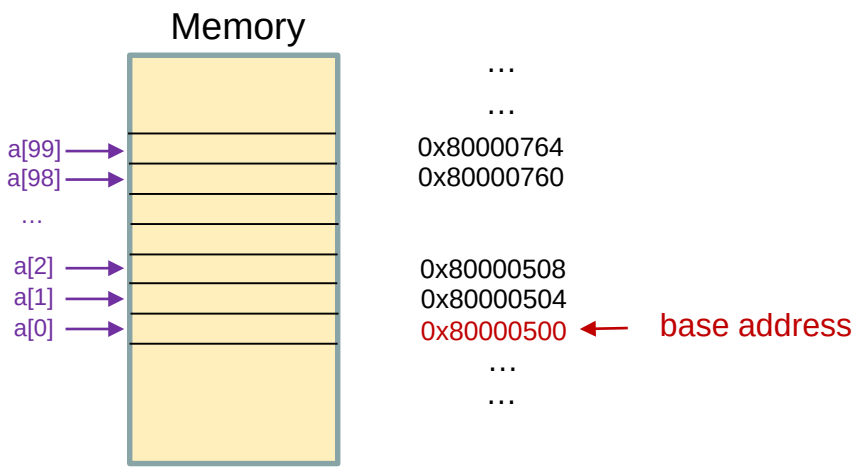
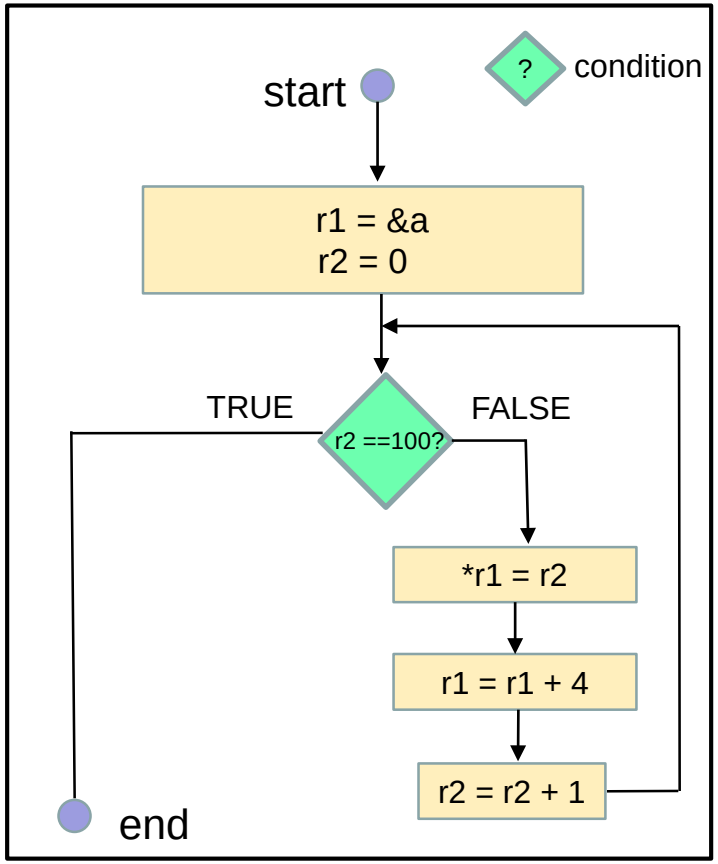


Illustration of array **a** in Memory



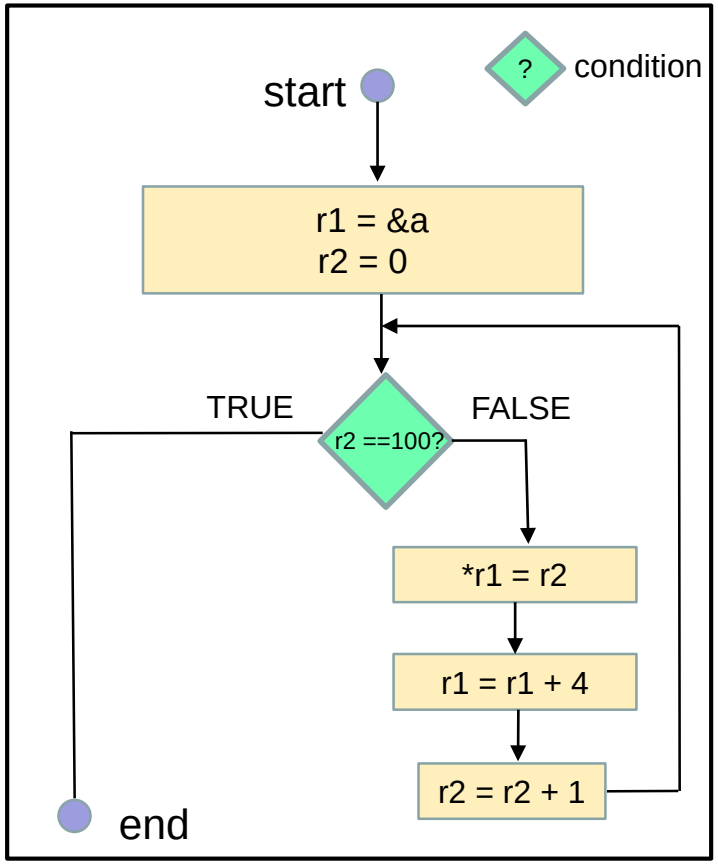
Flow chart of the program

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : memory of the current index of array **a**
    - r2 : index of the current value

```
loop:
    cmp r2, #100        @ Have we reached 100 yet?
    beq end             @ If so, leave the loop, otherwise continue
    str r2, [r1], #+4   @ *r1 <- r2 then r1 <- r1 + 4
    add r2, r2, #1      @ r2 <- r2 + 1
    b   loop            @ Go to the beginning of the loop
end:
```



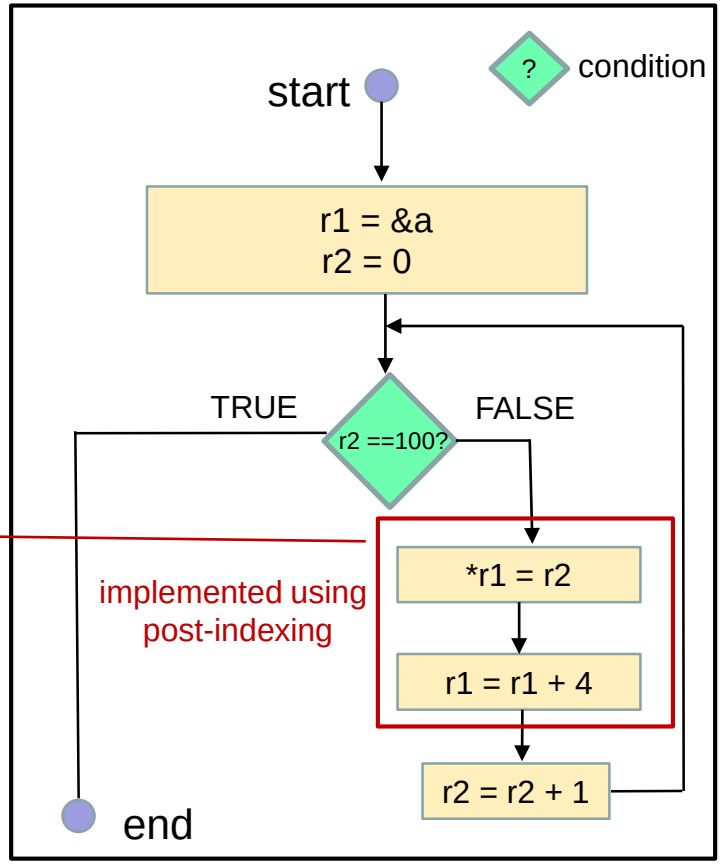
Flow chart of the program

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : memory of the current index of array **a**
    - r2 : index of the current value

```
loop:
    cmp r2, #100        @ Have we reached 100 yet?
    beq end             @ If so, leave the loop, otherwise continue
    str r2, [r1], #+4   @ *r1 <- r2 then r1 <- r1 + 4
    add r2, r2, #1      @ r2 <- r2 + 1
    b   loop            @ Go to the beginning of the loop
end:
```



Flow chart of the program

Raspberry Pi Assembler

Updating Indexing Modes: post-indexing

- Post-indexing mode 1

```
instruction Rdest, [Rsource1], #±immediate
```

```
str    r2,  [r1],  #4
```

Equivalent to the following operations:

```
str    r2,  [r1]
add    r1,  r1,  #4
```

- Post-indexing mode 2

```
instruction Rdest, [Rsource1], ±Rsource2
```

```
str    r2,  [r1],  +r2
```

Equivalent to the following operations:

```
str    r2,  [r1]
add    r1,  r1,  r2
```

- Post-indexing mode 3

```
instruction Rdest, [Rsource1], ±Rsource2, shift_operation #±immediate
```

```
str    r2,  [r1],  +r2, LSL  #4
```

Equivalent to the following operations:

```
str    r2,  [r1]
add    r1,  r1,  r2, LSL  #4
```

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

```
instruction Rdest, [Rsource1, #±immediate]!
```

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

```
instruction Rdest, [Rsource1, #±immediate]!
```

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

```
instruction Rdest, [Rsource1, #±immediate]!
```

Example: str r2, [r1, #4] !

- First, r1 is updated: r1 = r1 + 4
- Then, the store operation is performed using the memory address contained in r1

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

```
instruction Rdest, [Rsource1, #±immediate]!
```

Pre-indexing performs two 'simple' instructions in one:

```
add   r1,  r1,  #4
str   r2,  [r1]
```

Example: `str r2, [r1, #4] !`

- First, r1 is updated: r1 = r1 + 4
- Then, the store operation is performed using the memory address contained in r1

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

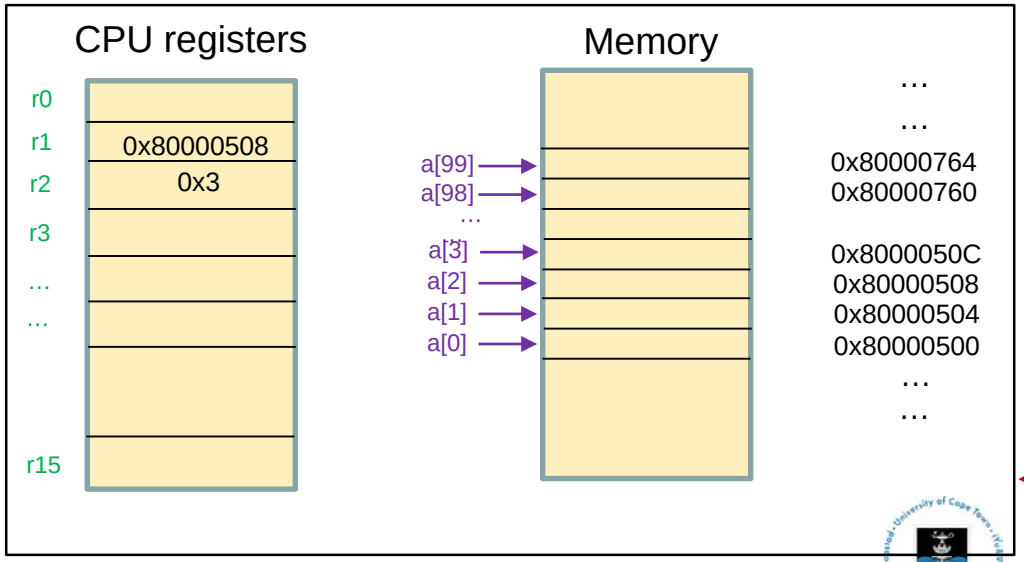
```
instruction Rdest, [Rsource1, #±immediate]!
```

Pre-indexing performs two 'simple' instructions in one:

```
add   r1,  r1,  #4
str   r2,  [r1]
```

Example: `str r2, [r1, #4] !`

- First, r1 is updated: r1 = r1 + 4
- Then, the store operation is performed using the memory address contained in r1



Before `str r2, [r1, #4]!` has executed

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

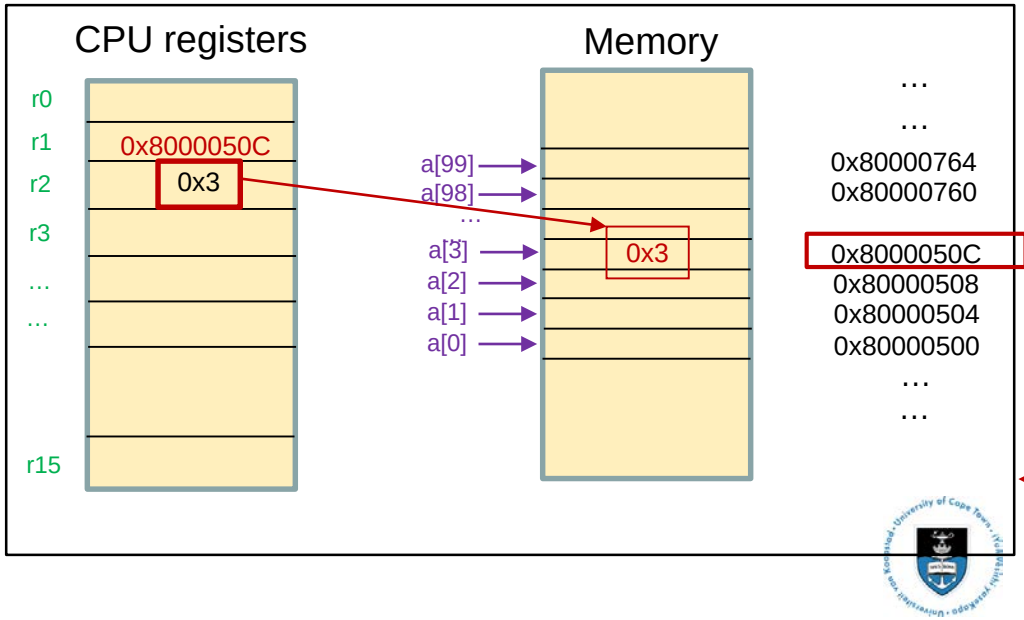
```
instruction Rdest, [Rsource1, #±immediate]!
```

Pre-indexing performs two 'simple' instructions in one:

```
add   r1,  r1,  #4
str   r2,  [r1]
```

Example: `str r2, [r1, #4] !`

- First, r1 is updated: r1 = r1 + 4
- Then, the store operation is performed using the memory address contained in r1



0x80000508 + 4 = 0x8000050C

After `str r2, [r1, #4]!` has executed

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- **Pre-indexing mode**

offset value specified using Rsource1 and an immediate value. The immediate value is limited to an offset value of 4095

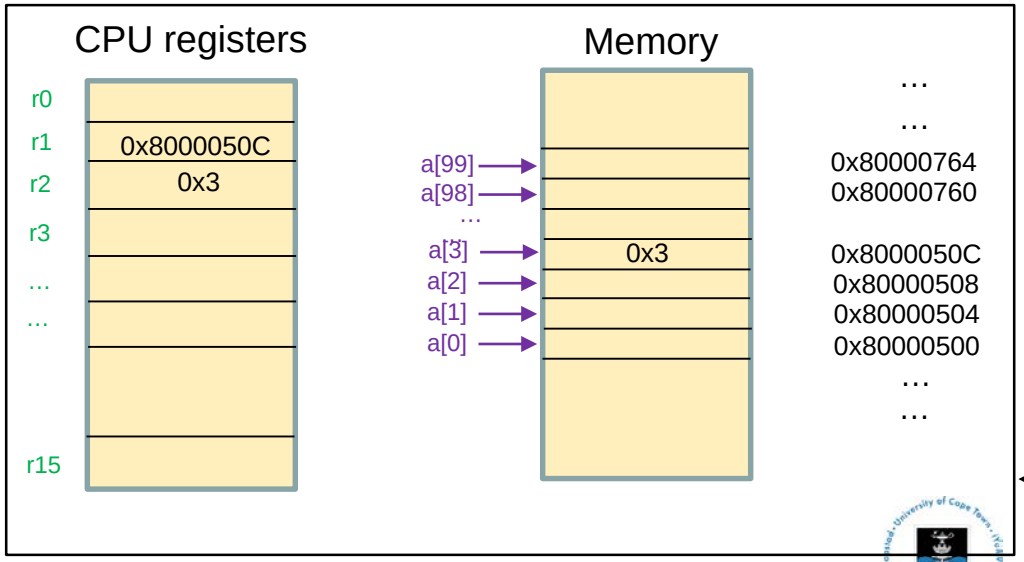
```
instruction Rdest, [Rsource1, #±immediate]!
```

Pre-indexing performs two 'simple' instructions in one:

```
add   r1,  r1,  #4
str   r2,  [r1]
```

Example: `str r2, [r1, #4] !`

- First, r1 is updated: r1 = r1 + 4
- Then, the store operation is performed using the memory address contained in r1



After `str r2, [r1, #4]!` has executed

Features of pre-indexing mode

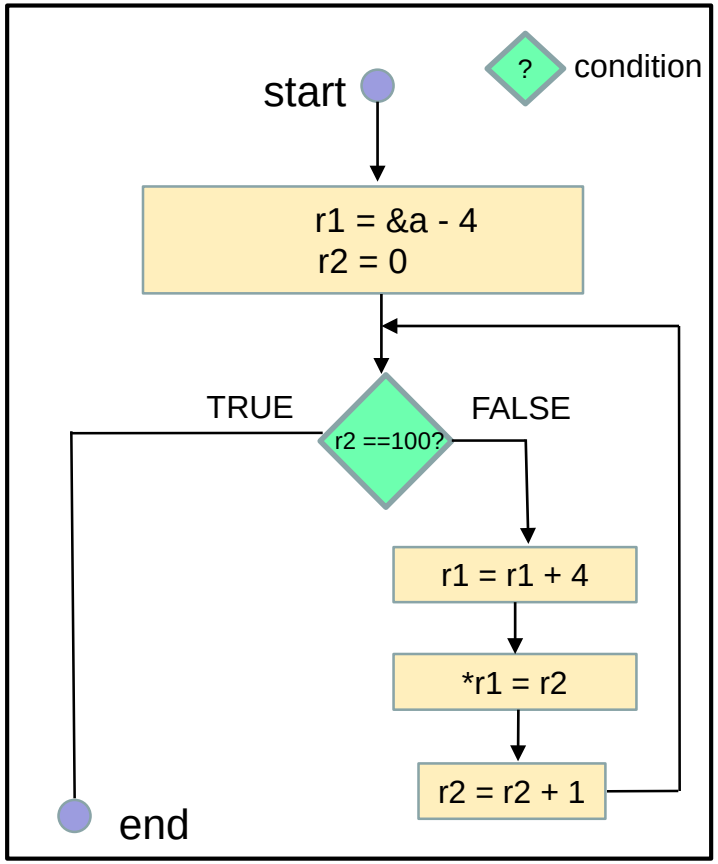
- The address is updated pre, ie. before, the operation is performed
- After the instruction has executed, the value of r1 changes

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : memory of the previous index of array **a**
    - r2 : index of the current value

```
loop:
    cmp r2, #100        @ Have we reached 100 yet?
    beq end             @ If so, leave the loop, otherwise continue
    str r2, [r1, #+4]!
    add r2, r2, #1      @ r2 <- r2 + 1
    b   loop            @ Go to the beginning of the loop
end:
```



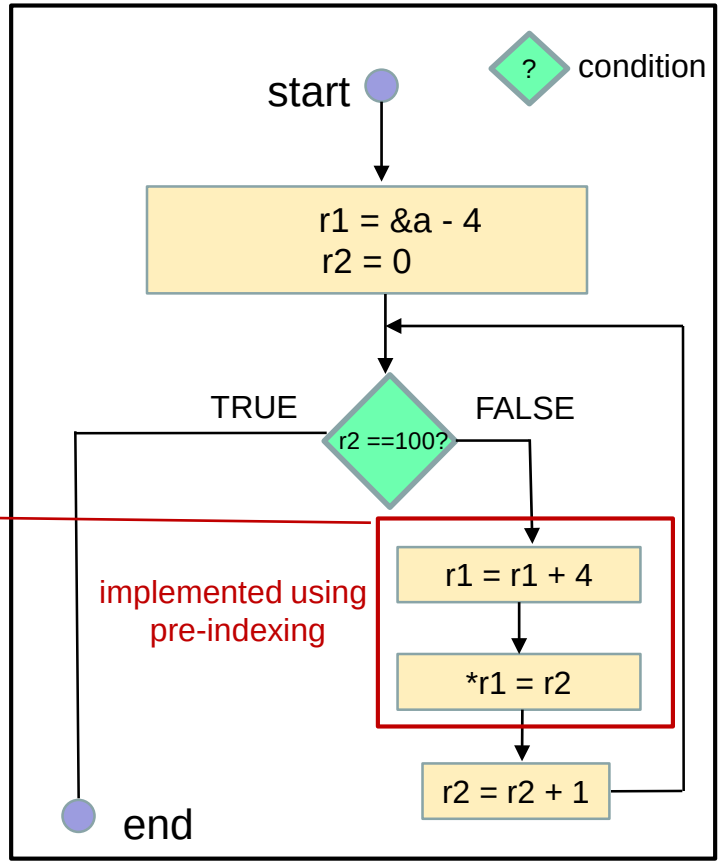
Flow chart of the program

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- First, let's plan out the flow chart for the program:
  - Variables used:
    - r1 : memory of the previous index of array **a**
    - r2 : index of the current value

```
loop:
    cmp r2, #100        @ Have we reached 100 yet?
    beq end             @ If so, leave the loop, otherwise continue
    str r2, [r1, #+4]!  @ r2 <- r2 + 1
    add r2, r2, #1      @ r2 <- r2 + 1
    b   loop            @ Go to the beginning of the loop
end:
```



Flow chart of the program

Raspberry Pi Assembler

Updating Indexing Modes: pre-indexing

- Pre-indexing mode 1

```
instruction Rdest, [Rsource1, #±immediate]!
```

```
str    r2,  [r1,  #4 ] !
```

Equivalent to the following operations:

```
add   r1,  r1,  #4
str   r2,  [r1]
```

- Pre-indexing mode 2

```
instruction Rdest, [Rsource1, ±Rsource2]!
```

```
str    r2,  [r1,  +r2 ] !
```

Equivalent to the following operations:

```
add   r1,  r1,  r2
str   r2,  [r1]
```

- Pre-indexing mode 3

```
instruction Rdest, [Rsource1, ±Rsource2, shift_operator #±immediate]!
```

```
str    r2,  [r1,  +r2, LSL  #4 ] !
```

Equivalent to the following operations:

```
add   r1,  r1,  r2, LSL  #4
str   r2,  [r1]
```

Updating the value of a field of a structure

Raspberry Pi Assembler

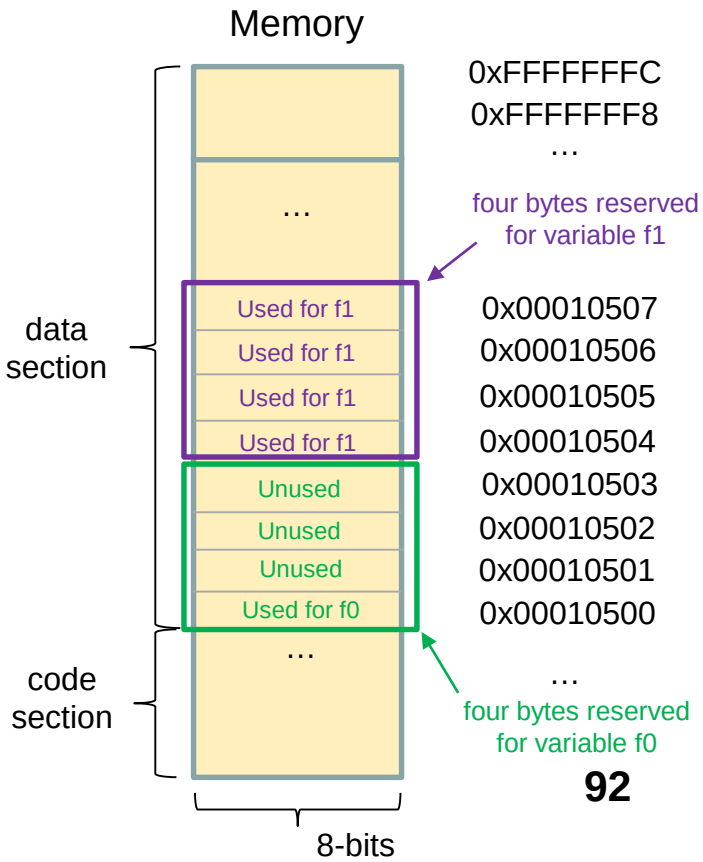
Structures: updating the value of a field

- Recap: earlier, we declared a structure **b** with two fields. The name of the first field is f0 and is type char and the name of the second field is f1 and is of type int. The declaration in the C language and assembly are:
  - C language

```
struct my_struct
{
  char f0;
  int f1;
} b;
```

  - Assembly

```
b:  .skip 8
```



Raspberry Pi Assembler

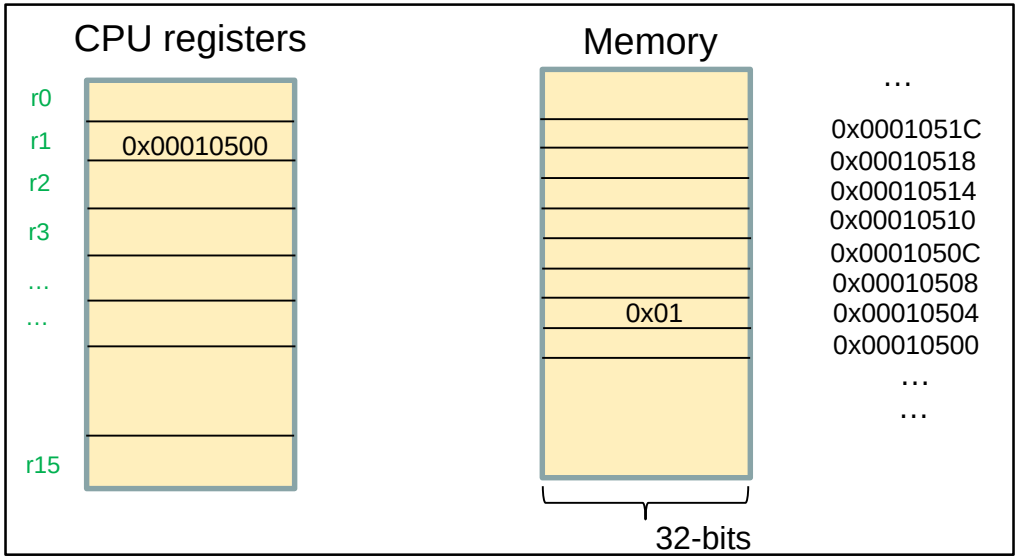
Structures: updating the value of a field

- Task: increment the value of field f1 by 7. Assume r1 contains the base address of the structure.
  - C language

    b.f1 = b.f1 + 7

  - Assembly

```
ldr r2, [r1, #+4]!  @ r1 <- r1 + 4 and then r2 <- *r1
add r2, r2, #7      @ r2 <- r2 + 7
str r2, [r1]        @ *r1 <- r2  where r1 has the correct address
```



Raspberry Pi Assembler

Structures: updating the value of a field

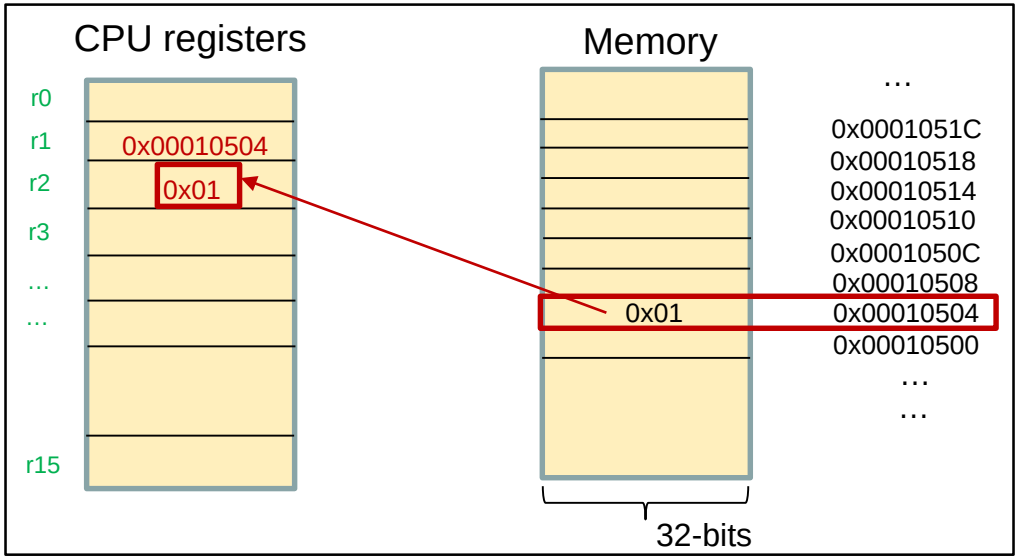
- Task: increment the value of field f1 by 7. Assume r1 contains the base address of the structure.
  - C language

    b.f1 = b.f1 + 7

  - Assembly

load contents of f1 to CPU register r2

```
ldr r2, [r1, #+4]!   @ r1 <- r1 + 4 and then r2 <- *r1
add r2, r2, #7       @ r2 <- r2 + 7
str r2, [r1]         @ *r1 <- r2  where r1 has the correct address
```



Raspberry Pi Assembler

Structures: updating the value of a field

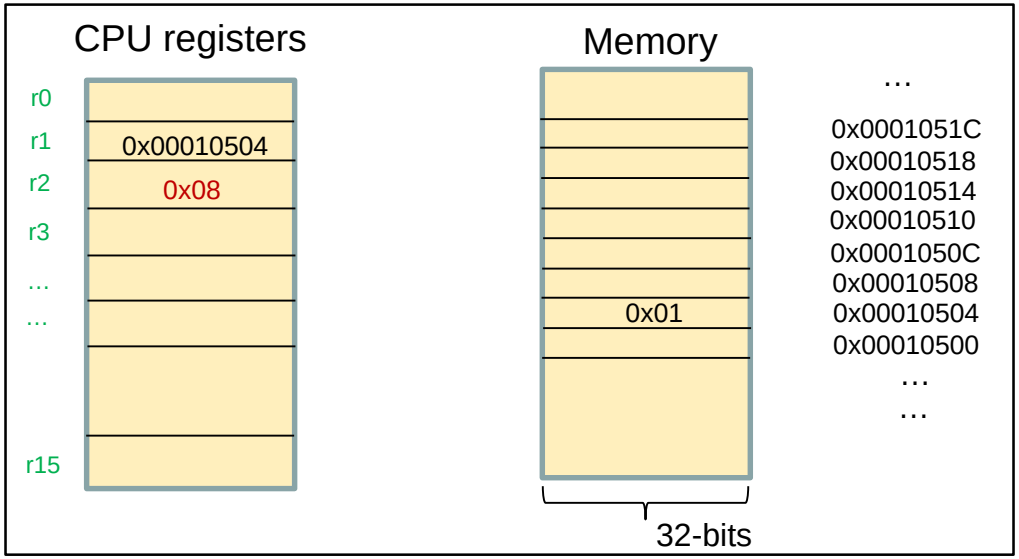
- Task: increment the value of field f1 by 7. Assume r1 contains the base address of the structure.
  - C language

    b.f1 = b.f1 + 7

  - Assembly

Increment the contents of r2 by 7

```
ldr r2, [r1, #+4]!   @ r1 <- r1 + 4 and then r2 <- *r1
add r2, r2, #7       @ r2 <- r2 + 7
str r2, [r1]         @ *r1 <- r2  where r1 has the correct address
```



Raspberry Pi Assembler

Structures: updating the value of a field

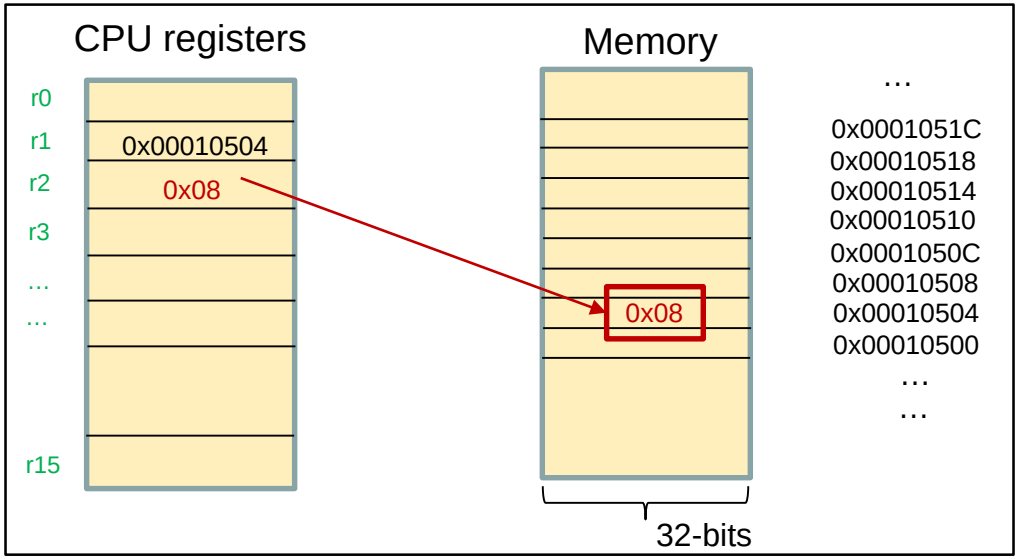
- Task: increment the value of field f1 by 7. Assume r1 contains the base address of the structure.
  - C language

    b.f1 = b.f1 + 7

  - Assembly

store the contents of r2 into field f1

```
ldr r2, [r1, #+4]!   @ r1 <- r1 + 4 and then r2 <- *r1
add r2, r2, #7       @ r2 <- r2 + 7
str r2, [r1]         @ *r1 <- r2  where r1 has the correct address
```



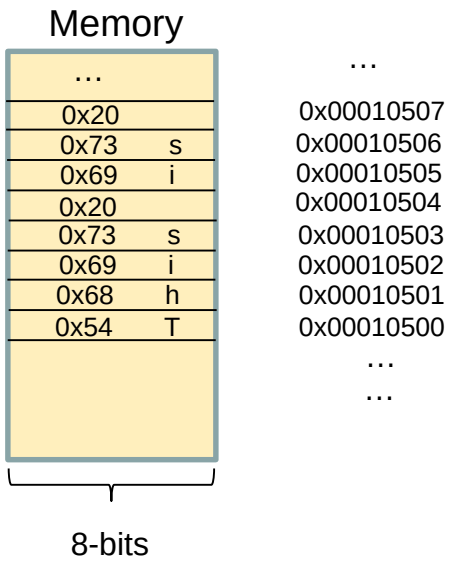
Printing a string to the terminal
(Strings are used in the next chapter)

Raspberry Pi Assembler

Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```



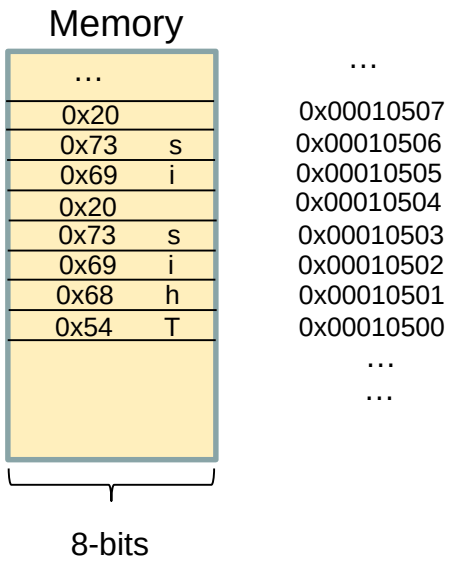
Raspberry Pi Assembler

Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```

S used 17 bytes of memory. One byte per character.
The last byte is a null byte for terminating the string.



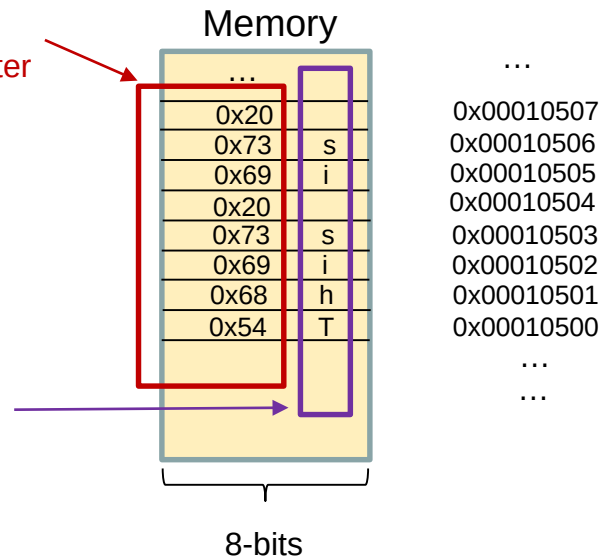
Raspberry Pi Assembler

Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```

ASCII value that represents a character



Character relating to the value in memory.
Note: only a numerical 8-bit number is stored in memory. This value can be used later to display a specific character.

Raspberry Pi Assembler

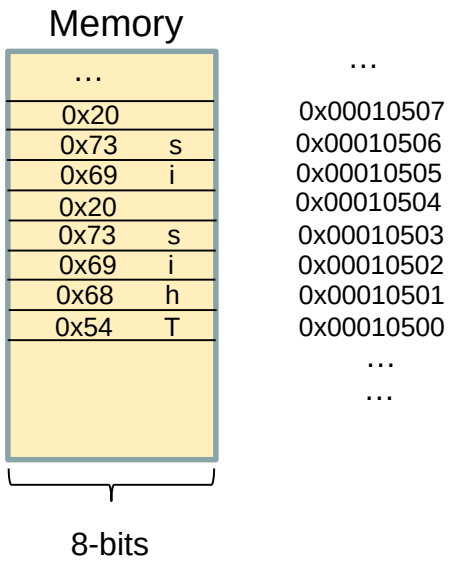
Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```

- Load the address of the string to the CPU register r0

```
ldr  r0,  =S
```



Raspberry Pi Assembler

Printing a string to the terminal

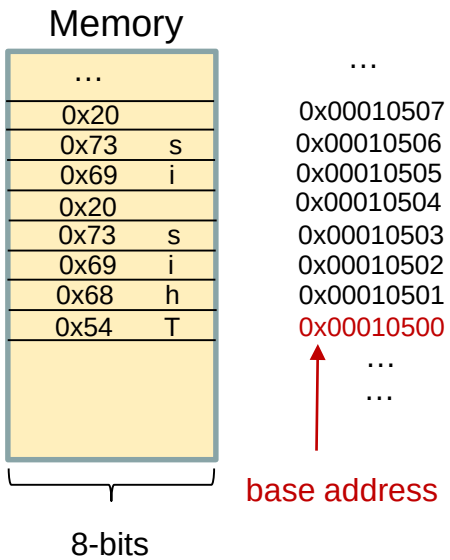
- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```

- Load the address of the string to the CPU register r0

```
ldr  r0, =S
```

The address of the string is the base address of the first index of the array

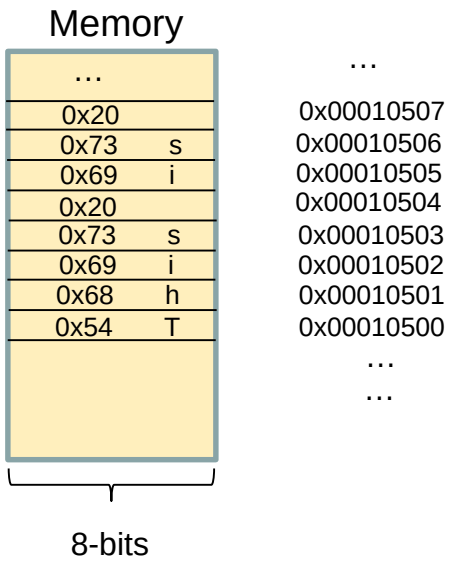


Raspberry Pi Assembler

Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```



- Load the address of the string to the CPU register r0

```
ldr  r0,  =S
```

- Print the string to the terminal

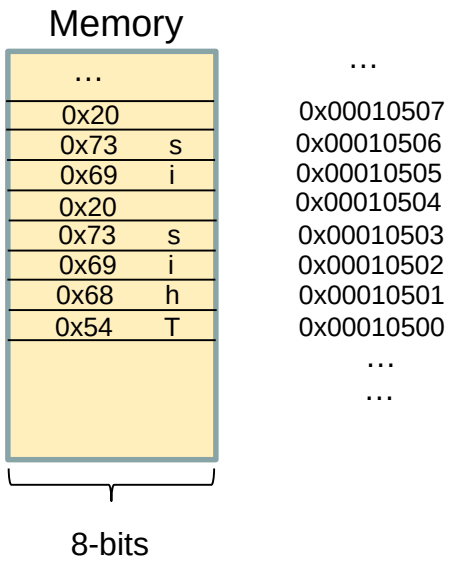
```
bl  puts
```

Raspberry Pi Assembler

Printing a string to the terminal

- A string is an array of characters. Each character occupies one byte of memory and has a unique address. The name of the string represents the address of the first character of the string
- Declaring a string S

```
S :    .asciz   "This is a string"
```



- Load the address of the string to the CPU register r0

```
ldr  r0,  =S
```

- Print the string to the terminal

```
bl  puts
```

- Calls the C function **puts** to print the string to the terminal. The **puts** function requires the address of the string to be in r0
- When compiling the code, the **puts** function will be converted to assembly instructions and then machine instructions