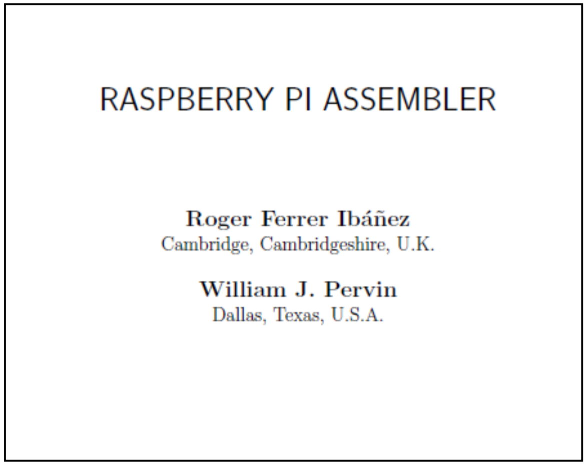
Raspberry Pi Assembler

Writing and running an assembly program



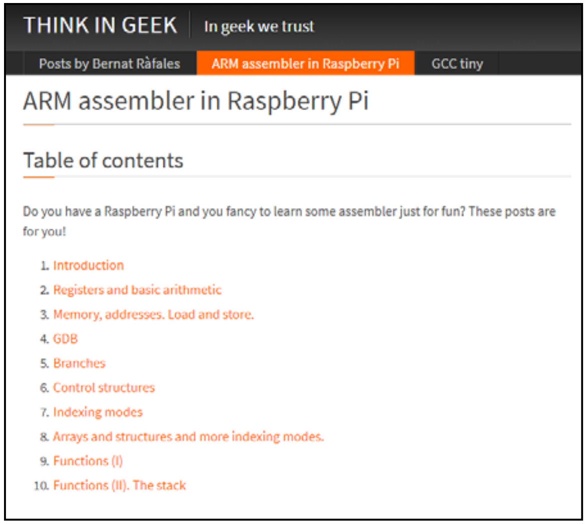
RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 1: Raspberry Pi Assembler
“Raspberry Pi Assembler” by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/



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!

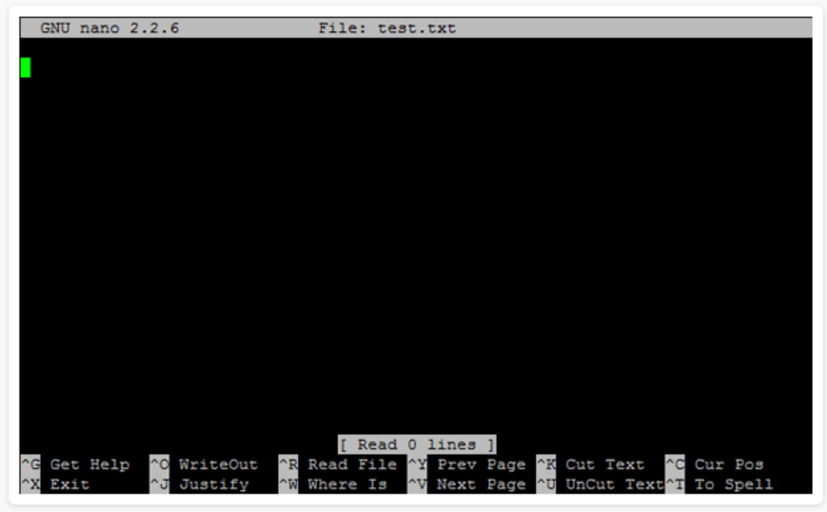
1. Introduction
2. Registers and basic arithmetic
3. Memory, addresses. Load and store.
4. GDB
5. Branches
6. Control structures
7. Indexing modes
8. Arrays and structures and more indexing modes.
9. Functions (I)
10. Functions (II). The stack

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

Raspberry Pi Assembler

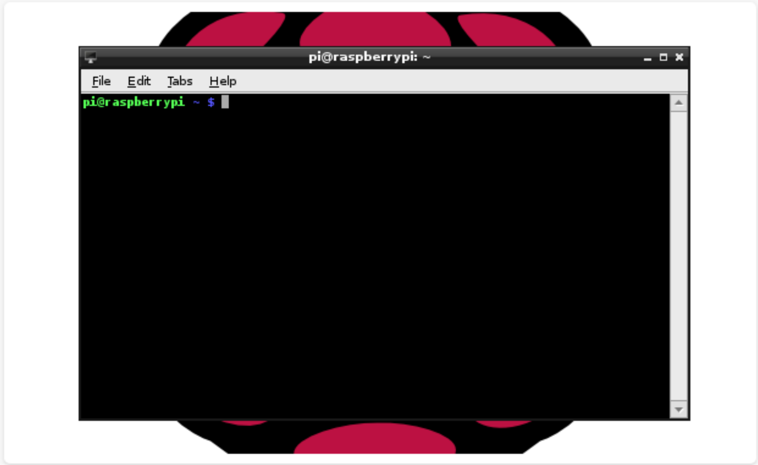
Writing and running an assembly program

- After understanding some fundamentals in computer architecture, we are now ready to write our first assembly program
- A two-step process is used to write and run an assembly program
  - **Step 1**: Write assembly code using the GNU nano text editor
  - **Step 2**: Assemble, link and run the executable file using command prompt or Linux shell



```
[ Read 0 lines ]
```

GNU Nano text editor



```
pi@raspberrypi ~ $
```

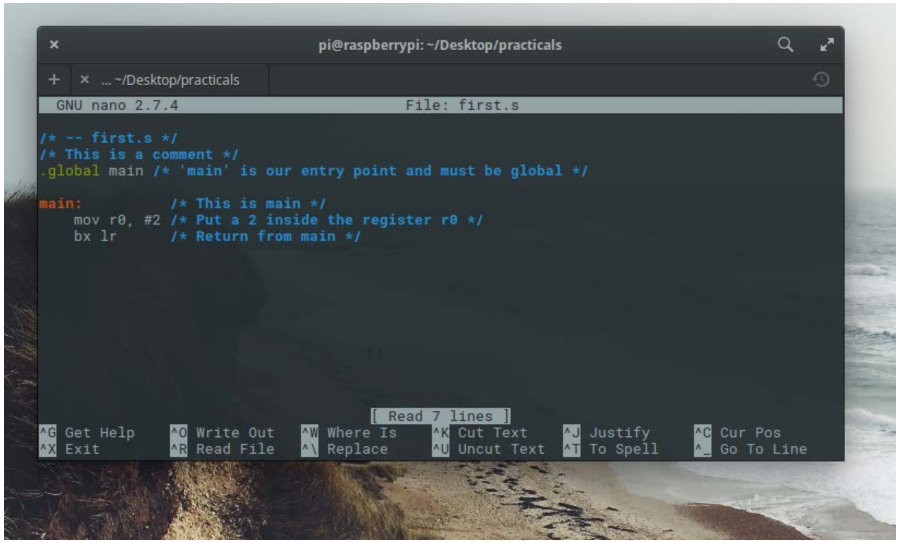
Command prompt or Linux shell

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file

```
/* -- first.s */
/* This is a comment */
.global main      /* entry point must be global */
.func main        /* ’main’ is a function */

main:             /* This is main */
    mov r0, #2    /* Put a 2 into register r0 */
    bx  lr        /* Return from main */
```

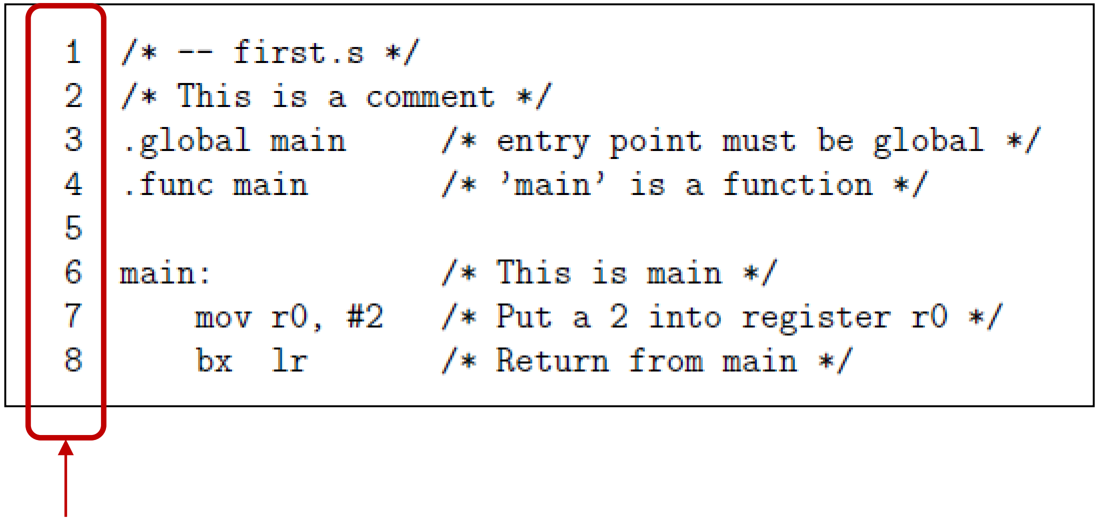


```
  GNU nano 2.7.4                          File: first.s

/* -- first.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global */

main:           /* This is main */
    mov r0, #2 /* Put a 2 inside the register r0 */
    bx lr       /* Return from main */

                                      [ Read 7 lines ]
```

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



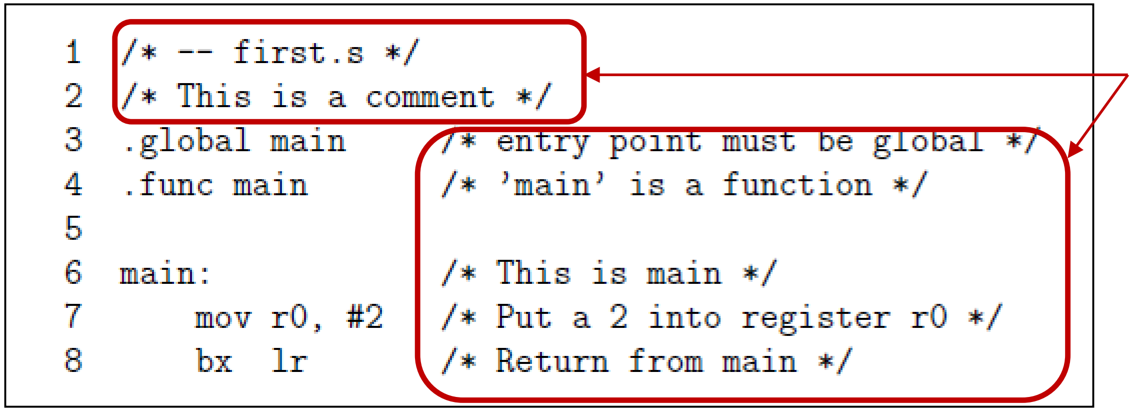
```
/* -- first.s */
/* This is a comment */
.global main        /* entry point must be global */
.func main          /* 'main' is a function */

main:               /* This is main */
    mov r0, #2      /* Put a 2 into register r0 */
    bx  lr          /* Return from main */
```

Line numbers used to refer to a line of code later in the slides. They are not part of the program

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



```
/* -- first.s */
/* This is a comment */
.global main        /* entry point must be global */
.func main          /* ’main’ is a function */

main:               /* This is main */
    mov r0, #2      /* Put a 2 into register r0 */
    bx  lr          /* Return from main */
```

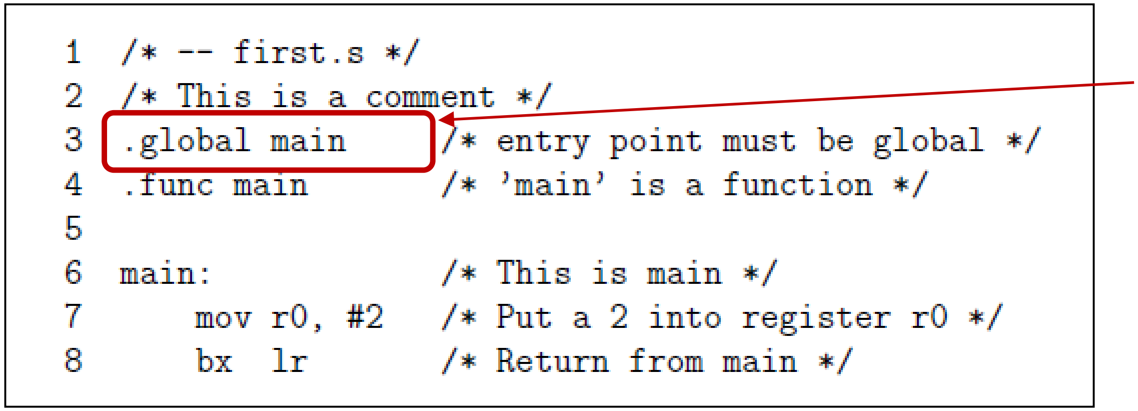
These are comments.
/* denotes the start of comment
*/ denotes the end of a comment

- The assembler ignores text between the /* and */
- Assembler code is difficult to understand, so comments are extremely useful to document what the code is doing

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



```
/* -- first.s */
/* This is a comment */
.global main        /* entry point must be global */
.func main          /* ’main’ is a function */

main:               /* This is main */
     mov r0, #2     /* Put a 2 into register r0 */
     bx  lr         /* Return from main */
```

.global main is an example of a directive for the GNU assembler

Directives tell the GNU Assembler to do something special other than emit a binary code. They start with a period denoted by a (.) followed by the name of the directive and possibly some arguments

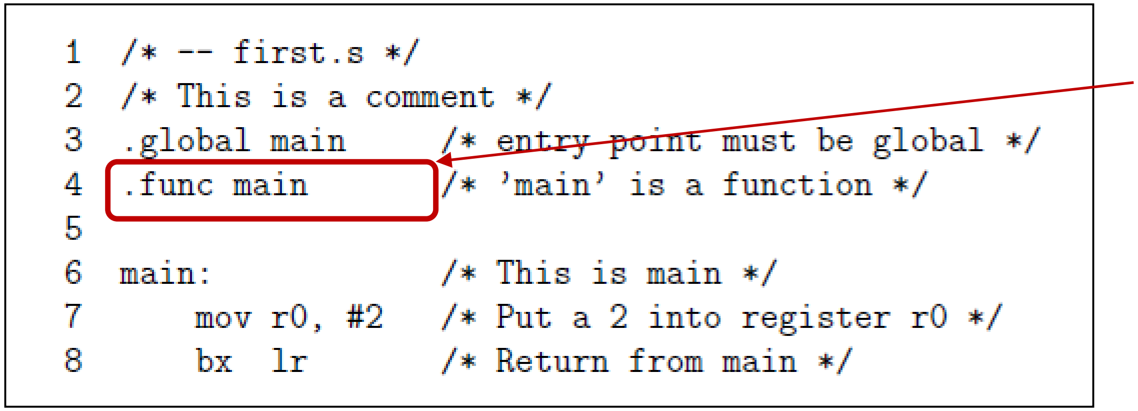
.global main is a directive to make ***main*** a global scope, so that it is recognisable outside the program.

This is needed because the C linker will call ***main*** at runtime. If its is not global, it will not be callable and the linking phase will fail.

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



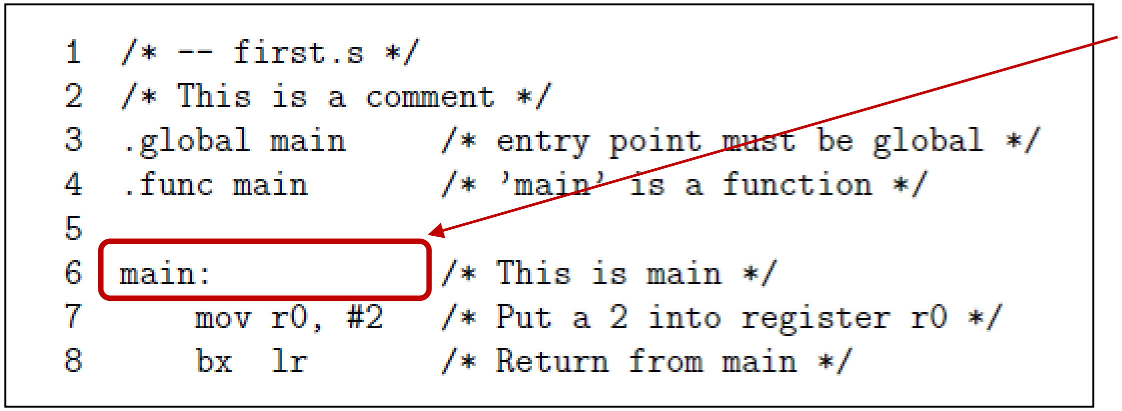
```
/* -- first.s */
/* This is a comment */
.global main          /* entry point must be global */
.func main            /* ’main’ is a function */

main:                 /* This is main */
    mov r0, #2        /* Put a 2 into register r0 */
    bx  lr            /* Return from main */
```

- A GNU assembler directive to declare ***main*** to be a function, which consists of code, ie. instructions of a program

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



```
1  /* -- first.s */
2  /* This is a comment */
3  .global main       /* entry point must be global */
4  .func main         /* 'main' is a function */
5
6  main:              /* This is main */
7       mov r0, #2    /* Put a 2 into register r0 */
8       bx  lr        /* Return from main */
```

Defining the label main

A line of GNU Assembler code that is not a directive, will be of the form

label: **instruction parameters comments**

- Blank lines are ignored by the Assembler
- A line with only a **label** applies that label to the next line

- The **instruction** part is the ARM assembler language

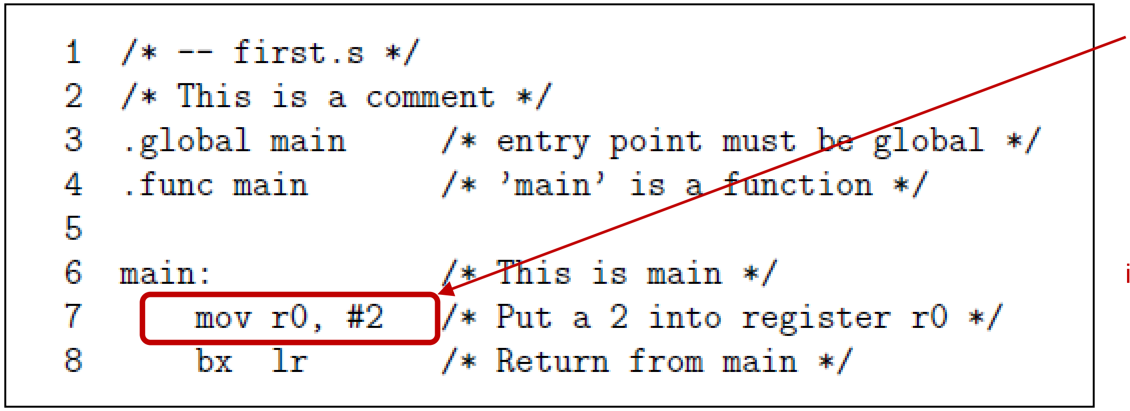
∴ main: on line 6 is just defining the label that applies to the instruction on line 7

We could have written line 6 and 7 as:
main: mov r0, #2

Raspberry Pi Assembler

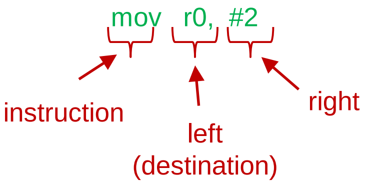
Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



```
/* -- first.s */
/* This is a comment */
.global main        /* entry point must be global */
.func main          /* 'main' is a function */

main:               /* This is main */
    mov r0, #2      /* Put a 2 into register r0 */
    bx  lr          /* Return from main */
```

Moves the decimal value of 2 into CPU register r0. In ARM syntax, the destination is mainly on the left (exception is the STR instruction):

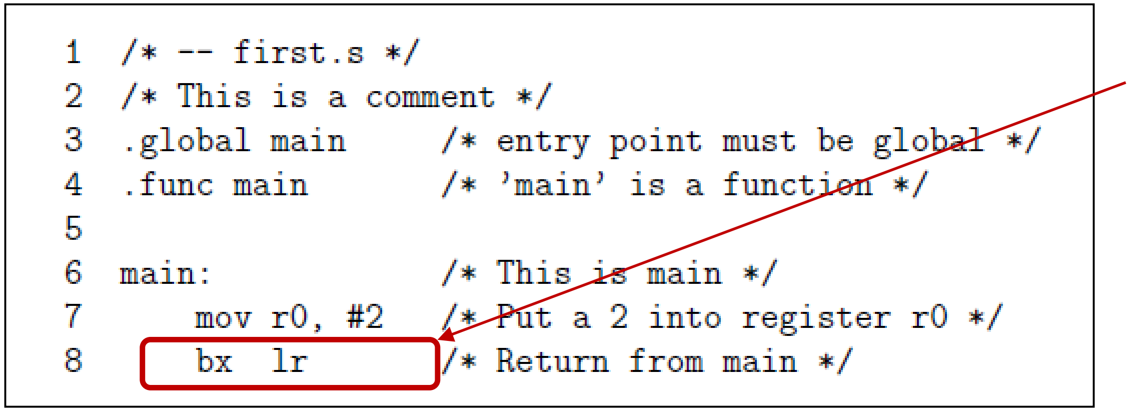


In a high level programming language, this would translate to:
r0 = 2

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file



```
/* -- first.s */
/* This is a comment */
.global main       /* entry point must be global */
.func main         /* ’main’ is a function */

main:              /* This is main */
    mov r0, #2     /* Put a 2 into register r0 */
    bx  lr         /* Return from main */
```

The instruction **bx** means *Branch and eXchange*

A branch instruction is used to change the sequential execution of instructions of a program.

We will cover branching in more depth in Chapter 5

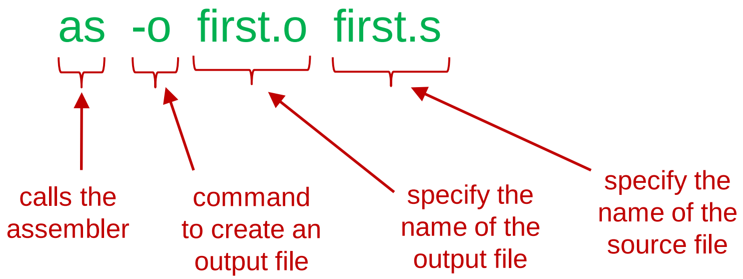
After the **bx** instruction executes, the code leaves the main function and program ends

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file
- **Step 2**: Assemble, link and run the executable file using the command prompt or Linux shell
  - 2A. Assemble code:

Convert from assembly to machine code



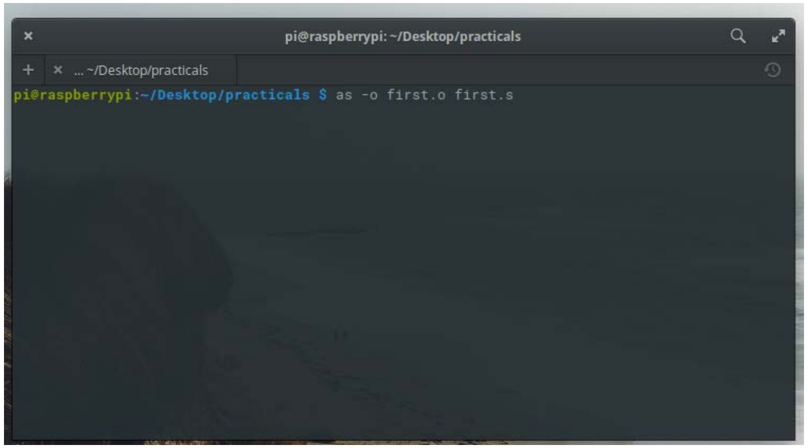
as -o first.o first.s

calls the assembler

command to create an output file

specify the name of the output file

specify the name of the source file



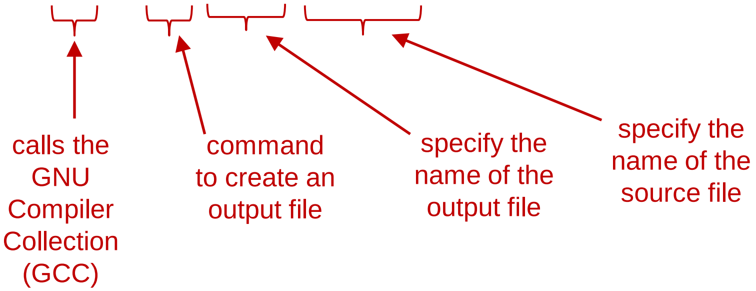
```
pi@raspberrypi:~/Desktop/practicals $ as -o first.o first.s
```

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file
- **Step 2**: Assemble, link and run the executable file using the command prompt or Linux shell
  - 2A. Assemble code: as -o first.o first.s
  - 2B. Link file: gcc -o first first.o

Create an executable

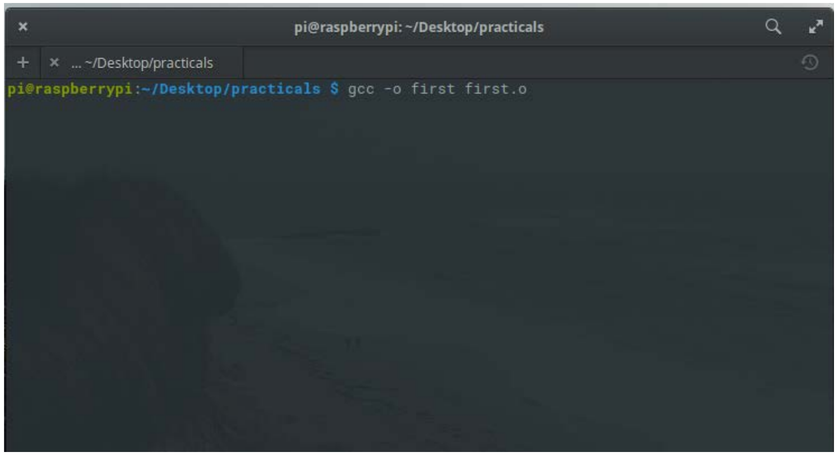


calls the GNU Compiler Collection (GCC)

command to create an output file

specify the name of the output file

specify the name of the source file

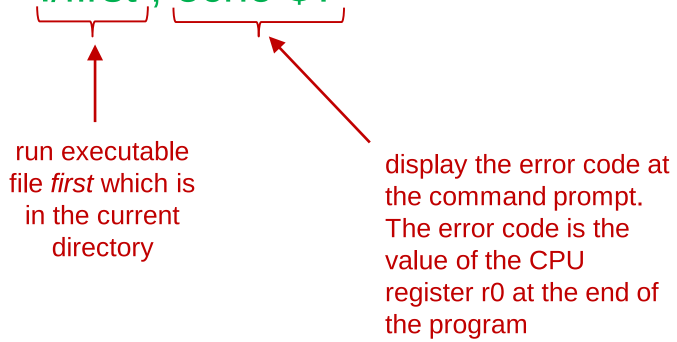


```
pi@raspberrypi:~/Desktop/practicals $ gcc -o first first.o
```

Raspberry Pi Assembler

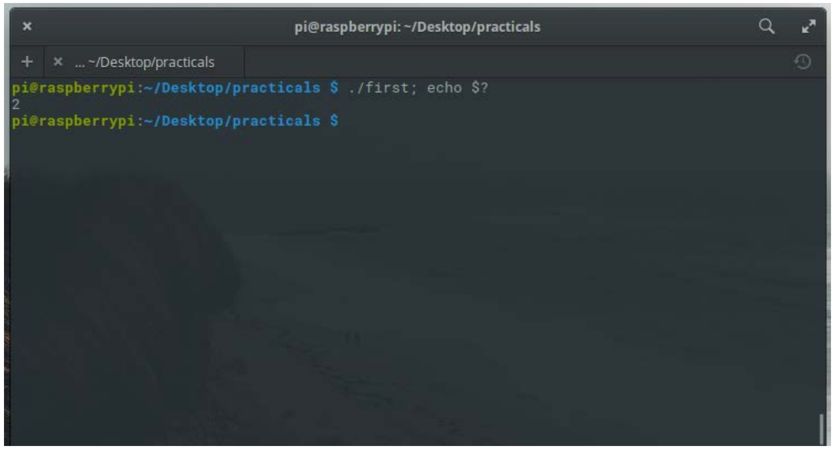
Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file
- **Step 2**: Assemble, link and run the executable file using the command prompt or Linux shell
  - 2A. Assemble code: as -o first.o first.s
  - 2B. Link file: gcc -o first first.o
  - 2C. Run executable: ./first ; echo $?



run executable file *first* which is in the current directory

display the error code at the command prompt. The error code is the value of the CPU register r0 at the end of the program



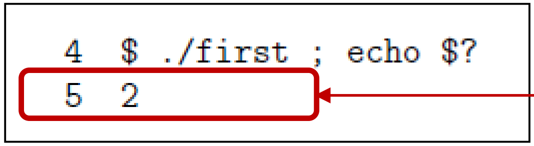
```
pi@raspberrypi:~/Desktop/practicals $ ./first; echo $?
2
pi@raspberrypi:~/Desktop/practicals $
```

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file
- **Step 2**: Assemble, link and run the executable file using the command prompt or Linux shell
  - 2A. Assemble code: as -o first.o first.s
  - 2B. Link file: gcc -o first first.o
  - 2C. Run executable: ./first ; echo $?

- What happens when the code runs?



```
$ ./first ; echo $?
2
```

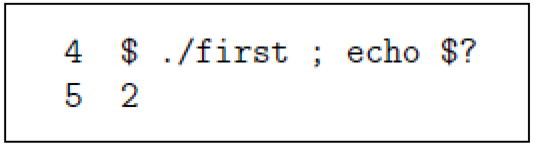
The value 2 is displayed to the user, which is exactly the value of the contents of CPU register r0

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click 'Yes' to keep the file
- **Step 2**: Assemble, link and run the executable file using the command prompt or Linux shell
  - 2A. Assemble code: `as -o first.o first.s`
  - 2B. Link file: `gcc -o first first.o`
  - 2C. Run executable: `./first ; echo $?`
- What happens when the code runs?

A makefile can be written to ease the task of assembling, linking and running the file.

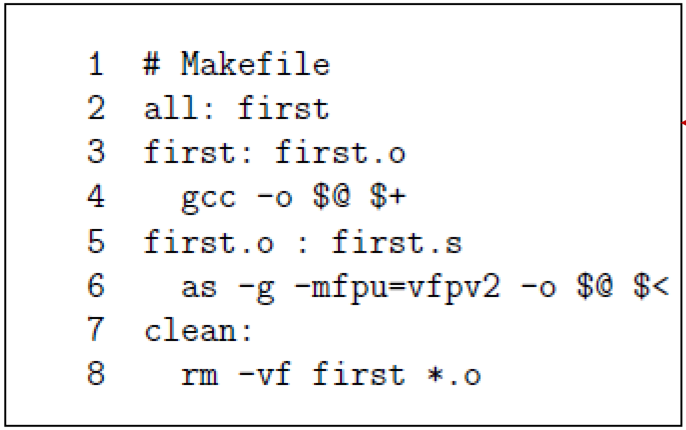


```
$ ./first ; echo $?
2
```

Raspberry Pi Assembler

Writing and running an assembly program

- **Step 1**: writing assembly code using the GNU nano text editor
  - 1A. In the terminal type nano first.s to launch Nano with a text file named first.s
  - 1B. Write out the assembly program in the GNU Nano text editor and press Ctrl-X to exit. Click ‘Yes’ to keep the file
- **Step 2**: ~~Assemble, link and run the executable file using the command prompt or Linux shell~~ Write a makefile to assemble, link and run the program



```
# Makefile
all: first
first: first.o
  gcc -o $@ $+
first.o : first.s
  as -g -mfpu=vfpv2 -o $@ $<
clean:
  rm -vf first *.o
```

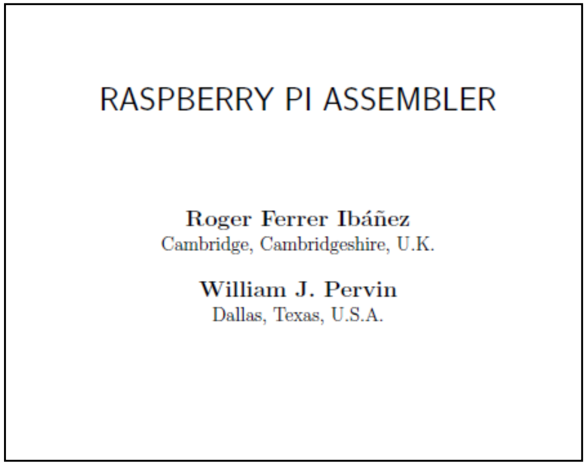
- Example of a makefile to assemble, link and run the program first.s
- Save this program into a file named *makefile*

Read about GNU make

https://www.gnu.org/software/make/manual/make.html

Raspberry Pi Assembler

An assembly program: adding two numbers



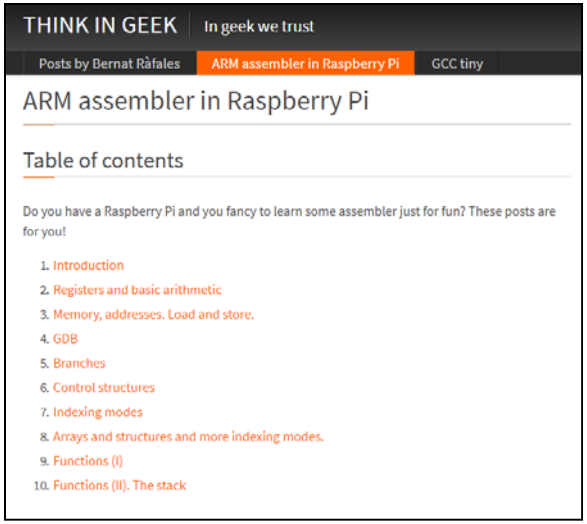
RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 2: Raspberry Pi Assembler
“Raspberry Pi Assembler” by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/10/arm-assembler-raspberry-pi-chapter-2/



ARM assembler in Raspberry Pi

Table of contents

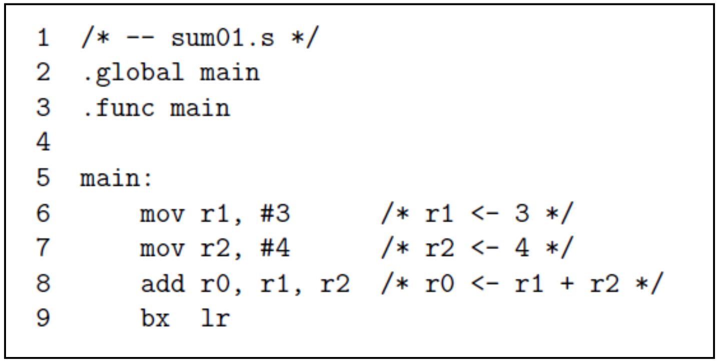
Do you have a Raspberry Pi and you fancy to learn some assembler just for fun? These posts are for you!

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

Raspberry Pi Assembler

An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2

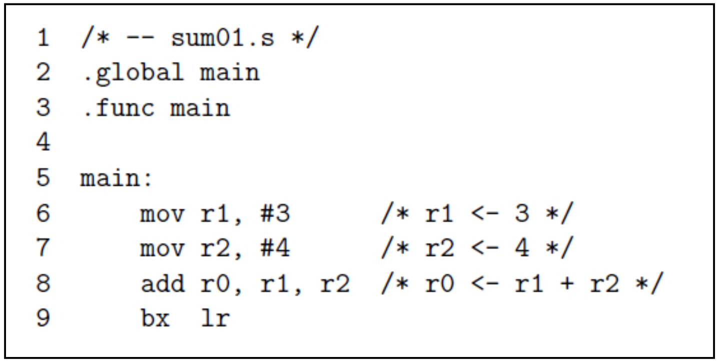


```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3      /* r1 <- 3 */
    mov r2, #4      /* r2 <- 4 */
    add r0, r1, r2  /* r0 <- r1 + r2 */
    bx  lr
```

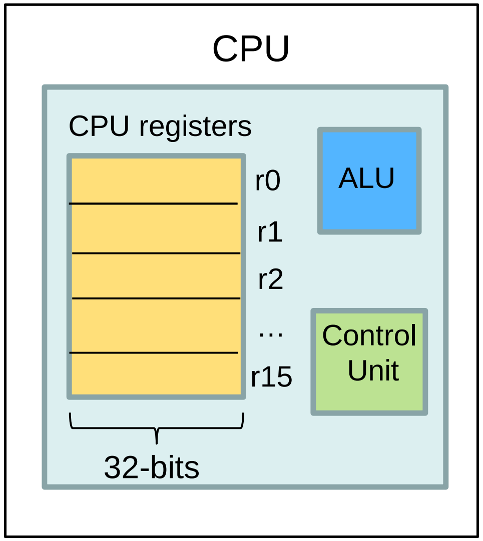
Raspberry Pi Assembler

An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



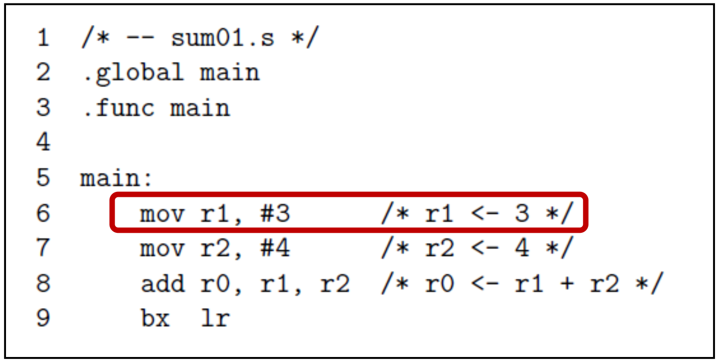
```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3       /* r1 <- 3 */
    mov r2, #4       /* r2 <- 4 */
    add r0, r1, r2   /* r0 <- r1 + r2 */
    bx  lr
```



Raspberry Pi Assembler

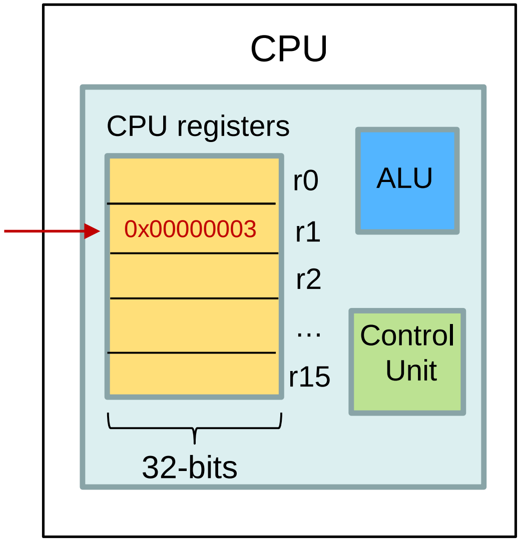
An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3         /* r1 <- 3 */
    mov r2, #4         /* r2 <- 4 */
    add r0, r1, r2     /* r0 <- r1 + r2 */
    bx  lr
```

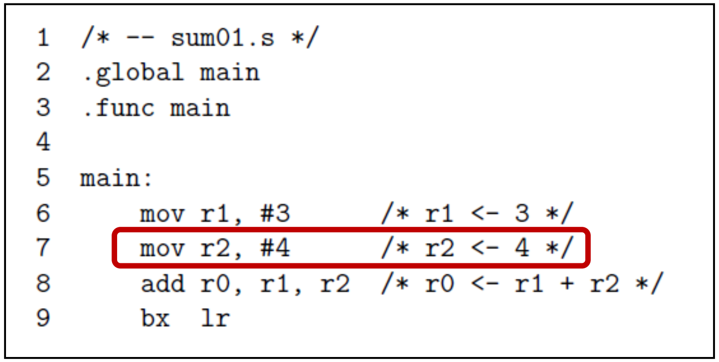
After mov r1, #3 has executed



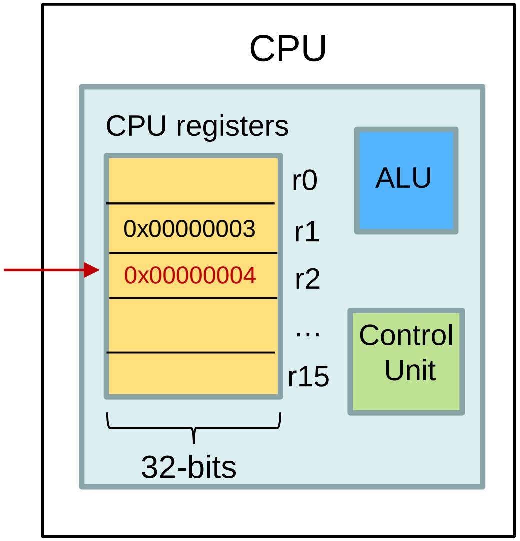
Raspberry Pi Assembler

An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
    - r1 = 3
    - r2 = 4
    - r0 = r1 + r2



```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3          /* r1 <- 3 */
    mov r2, #4          /* r2 <- 4 */
    add r0, r1, r2      /* r0 <- r1 + r2 */
    bx  lr
```

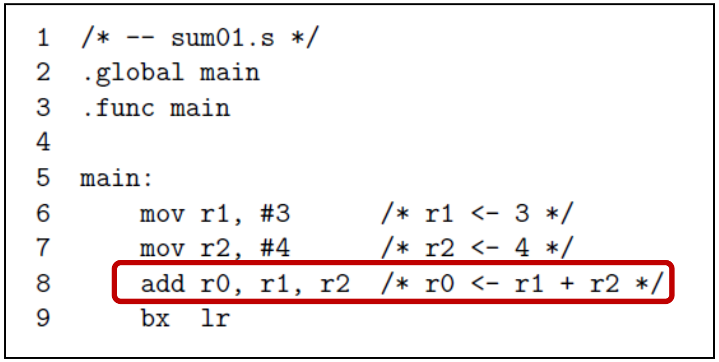


After mov r2, #4 has executed

Raspberry Pi Assembler

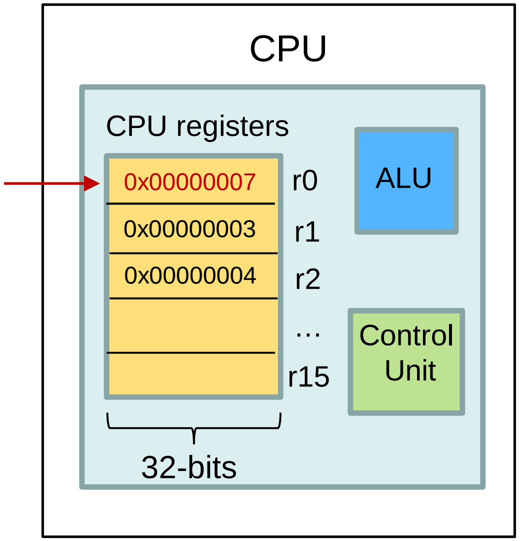
An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3       /* r1 <- 3 */
    mov r2, #4       /* r2 <- 4 */
    add r0, r1, r2   /* r0 <- r1 + r2 */
    bx  lr
```

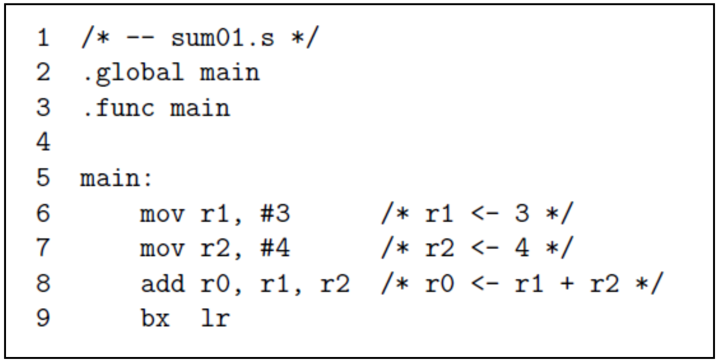
After add r0, r1, r2 has executed



Raspberry Pi Assembler

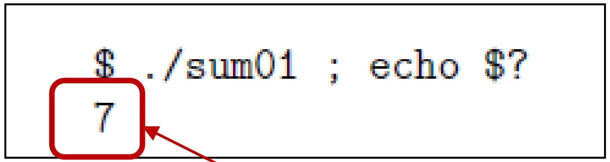
An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



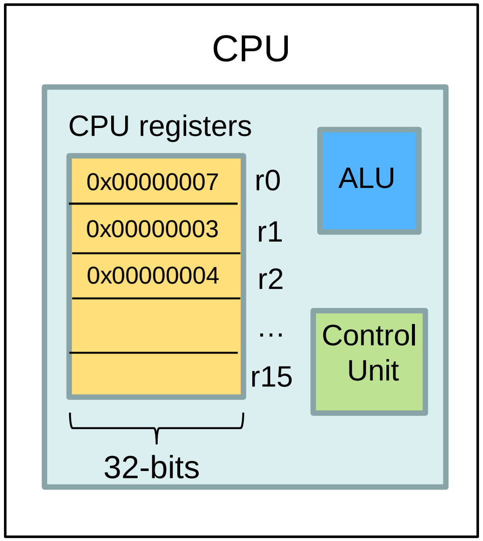
```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3      /* r1 <- 3 */
    mov r2, #4      /* r2 <- 4 */
    add r0, r1, r2  /* r0 <- r1 + r2 */
    bx  lr
```

- After the program has executed …



```
$ ./sum01 ; echo $?
7
```

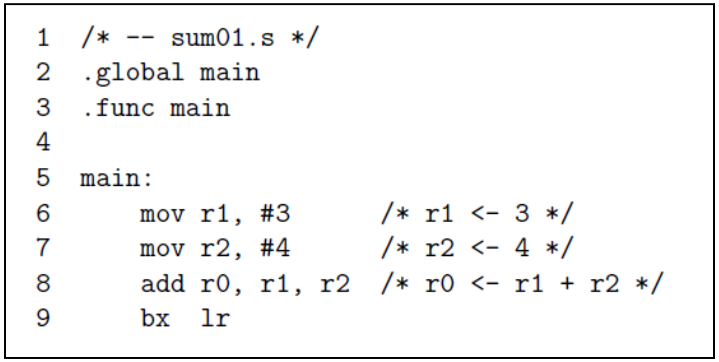
the value of the CPU register r0 is displayed to the user



Raspberry Pi Assembler

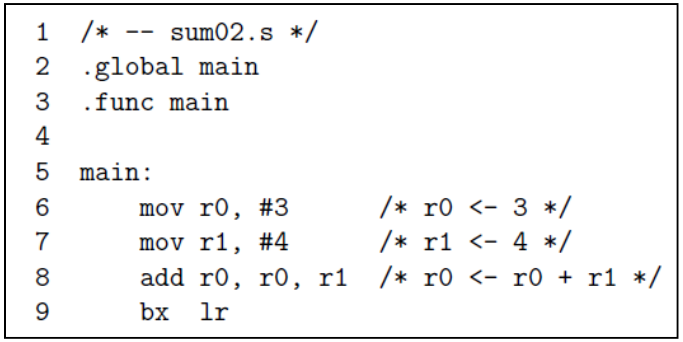
An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3      /* r1 <- 3 */
    mov r2, #4      /* r2 <- 4 */
    add r0, r1, r2  /* r0 <- r1 + r2 */
    bx  lr
```

- **Program 2, sum02.s:** perform the sum of two numbers using two CPU registers. The equivalent code using a high level language is:
  - r0 = 3
  - r1 = 4
  - r0 = r0 + r1

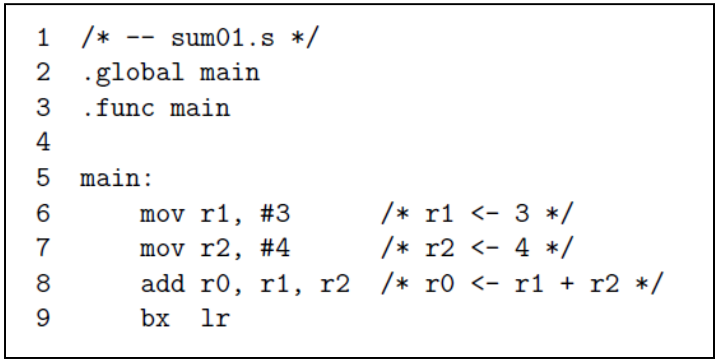


```
/* -- sum02.s */
.global main
.func main

main:
    mov r0, #3      /* r0 <- 3 */
    mov r1, #4      /* r1 <- 4 */
    add r0, r0, r1  /* r0 <- r0 + r1 */
    bx  lr
```

Raspberry Pi Assembler

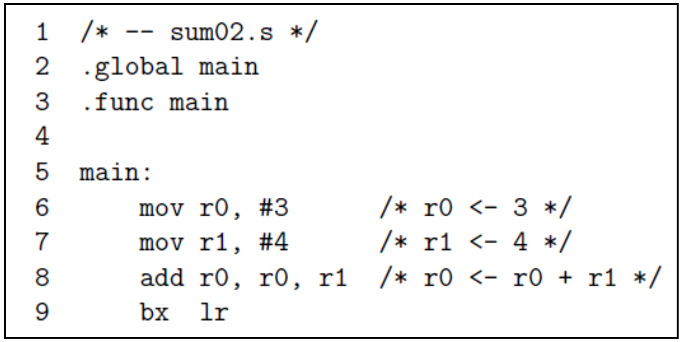
An assembly program: adding two numbers

- **Program 1, sum01.s:** perform the sum of two numbers using three CPU registers. The equivalent code using a high level language is:
  - r1 = 3
  - r2 = 4
  - r0 = r1 + r2



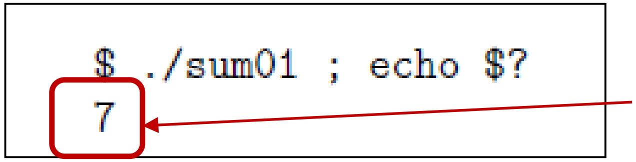
```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3       /* r1 <- 3 */
    mov r2, #4       /* r2 <- 4 */
    add r0, r1, r2   /* r0 <- r1 + r2 */
    bx  lr
```

- **Program 2, sum02.s:** perform the sum of two numbers using two CPU registers. The equivalent code using a high level language is:
  - r0 = 3
  - r1 = 4
  - r0 = r0 + r1



```
/* -- sum02.s */
.global main
.func main

main:
    mov r0, #3       /* r0 <- 3 */
    mov r1, #4       /* r1 <- 4 */
    add r0, r0, r1   /* r0 <- r0 + r1 */
    bx  lr
```

After the program has executed …

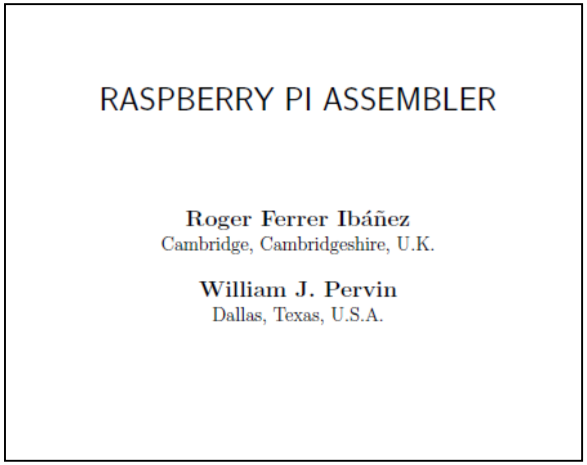


```
$ ./sum01 ; echo $?
7
```

Exactly the same result as program 1. Program2 is more efficient because it only uses two CPU registers

Raspberry Pi Assembler

Memory



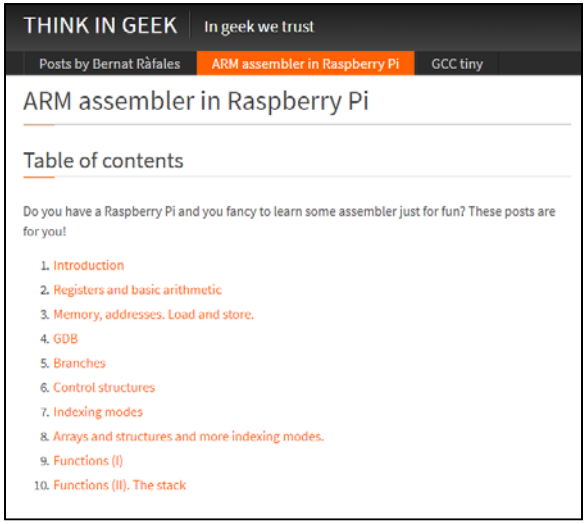
RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 3: Raspberry Pi Assembler
“Raspberry Pi Assembler” by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/11/arm-assembler-raspberry-pi-chapter-3/



ARM assembler in Raspberry Pi

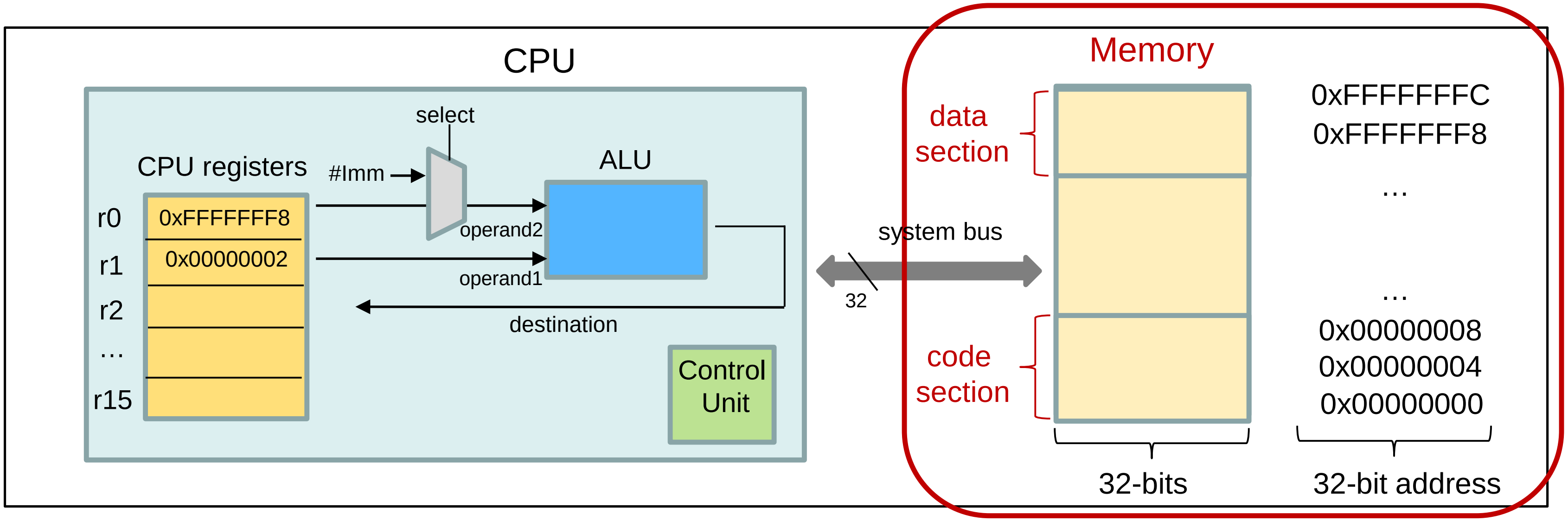
Table of contents

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

Raspberry Pi Assembler

Recap: memory and ARM load-store architecture

- A computer has memory where both code and data are stored

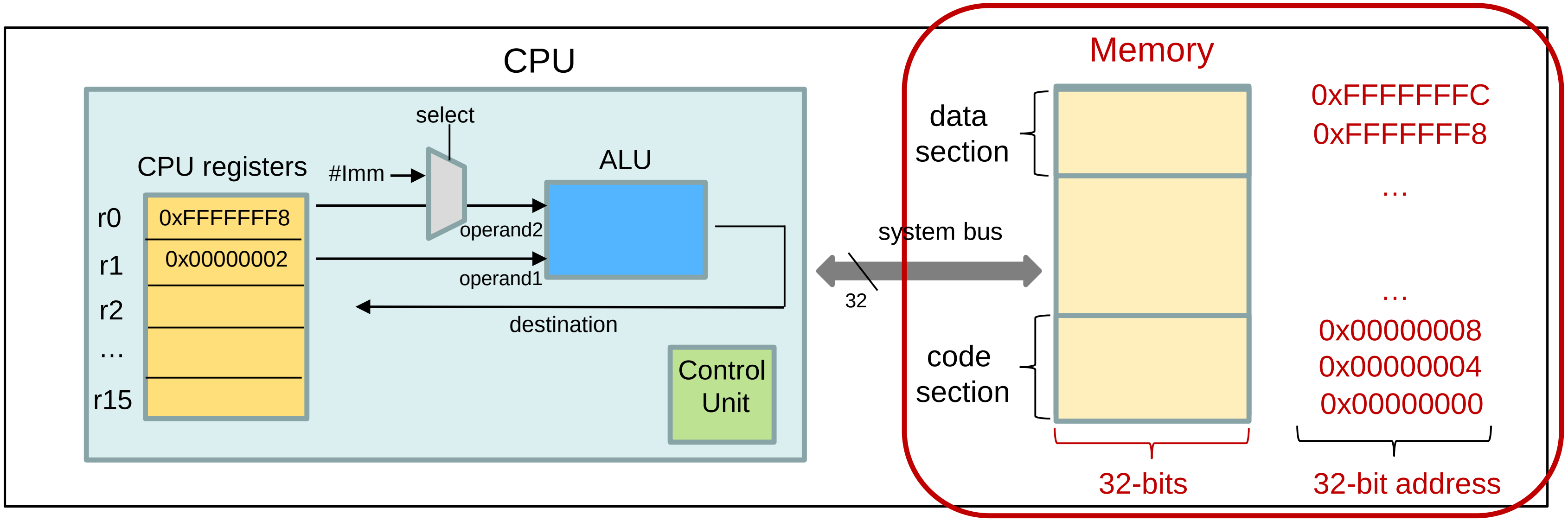


Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

Recap: memory and ARM load-store architecture

- A computer has memory where both code and data are stored
  - Each 8-bit register in memory has a unique address
  - By stacking four 8-bits registers together to form 32-bits, the address of every 4th register increments by the value 4

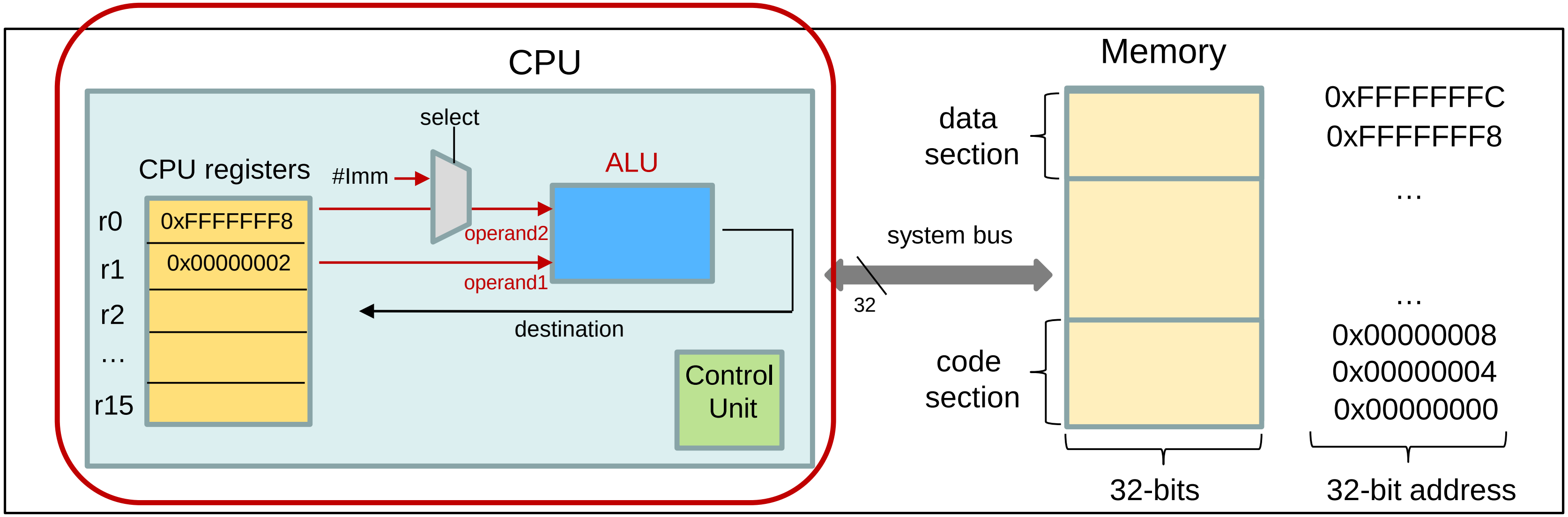


Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

Recap: memory and ARM load-store architecture

- A computer has memory where both code and data are stored
- The ARM CPU is a load-store architecture
  - Data from memory must be loaded into the CPU registers in order for the ALU to operate on them
  - When operations need to be done on data in memory, then a Store (STR) operation must be performed to copy this data into a CPU register



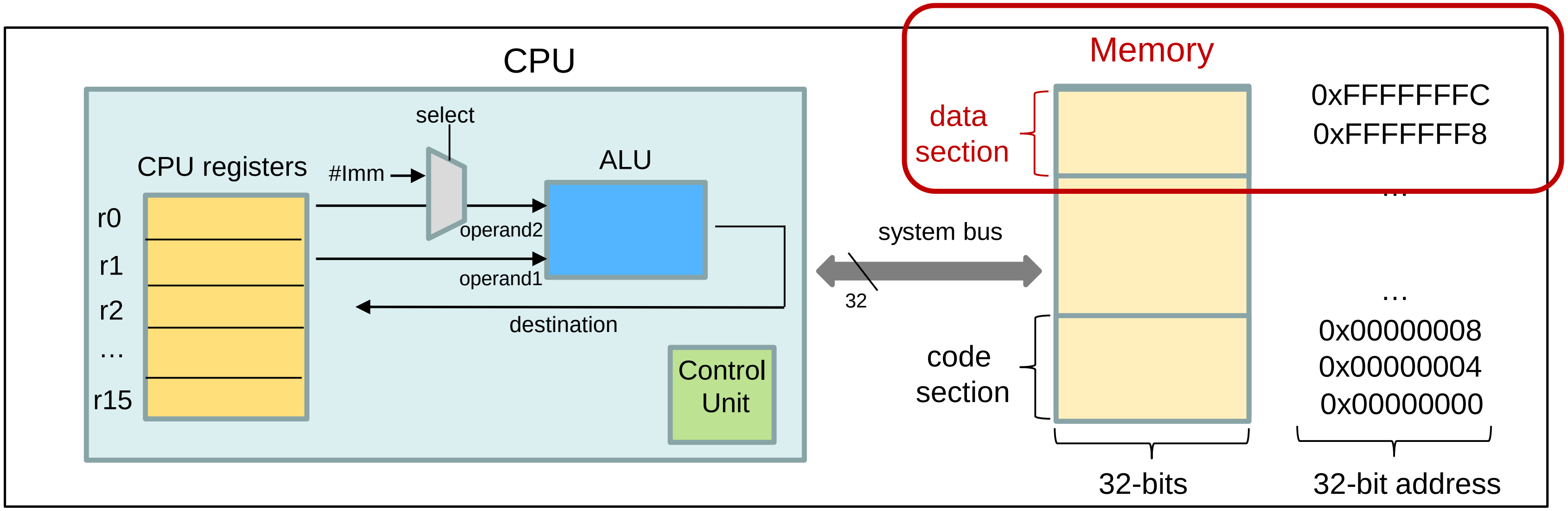
Simplified, conceptual block diagram of a computer

Defining variables in Memory

Raspberry Pi Assembler

Defining variables in Memory

- In assembly, we can define variables in the data section of Memory

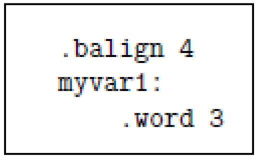


Simplified, conceptual block diagram of a computer

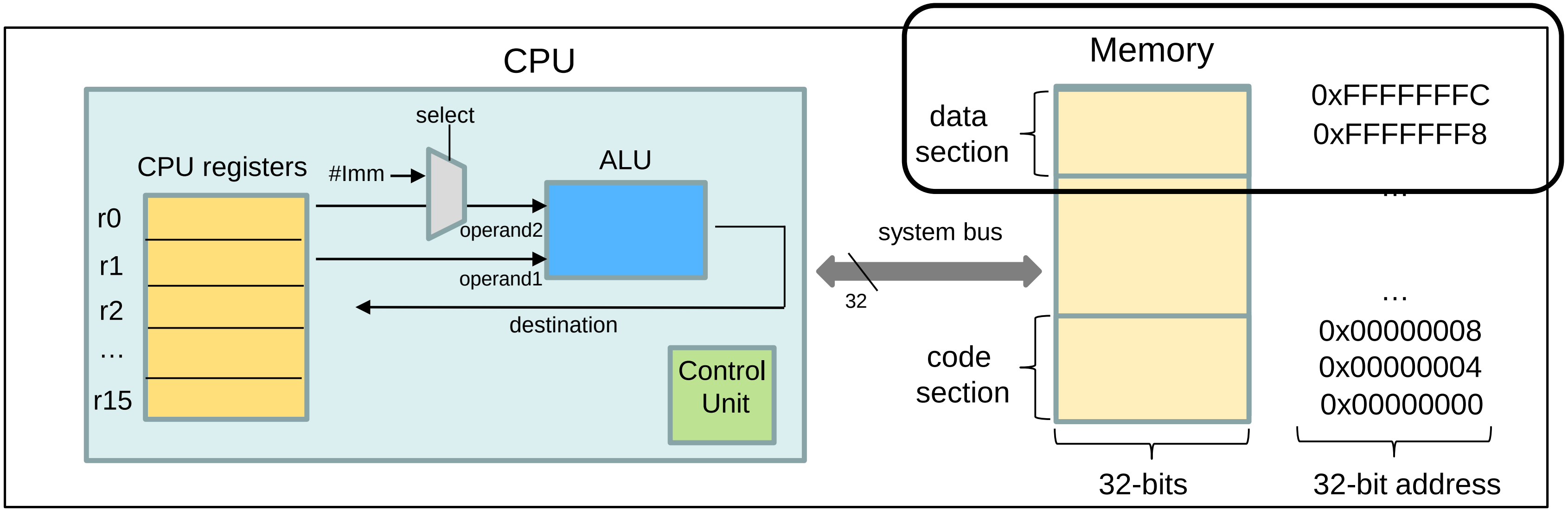
Raspberry Pi Assembler

Defining variables in Memory

- In assembly, we can define variables in the data section of Memory
- Let’s look at an example:
  - Define a 4-byte variable named *myvar1* and initialise it to the value 3



```
 .balign 4
myvar1:
     .word 3
```

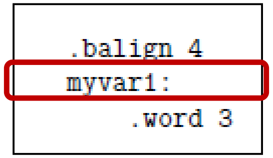


Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

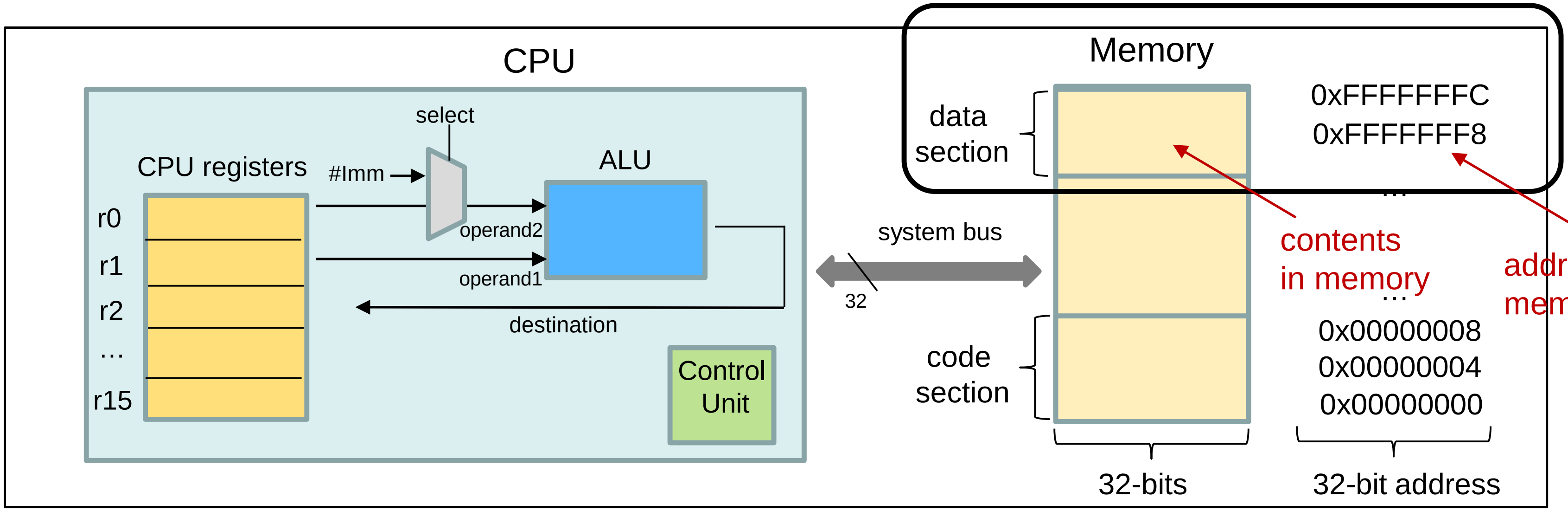
Defining variables in Memory

- In assembly, we can define variables in the data section of Memory
- Let’s look at an example:
  - Define a 4-byte variable named *myvar1* and initialise it to the value 3



```
    .balign 4
   myvar1:
        .word 3
```

- The assembly label myvar1 corresponds to the name of the variable *myvar1* and the label represents the address of the variable in memory
- Later, the assembler tool (as) will assign an 32-bit address to this label. Example, myvar could be assigned the value 0xFFFF0000
- Thus, a label is a **symbolic name** to an address in memory and not the contents in memory



contents in memory

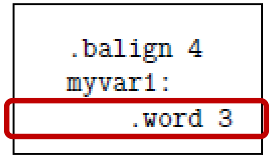
address in memory

Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

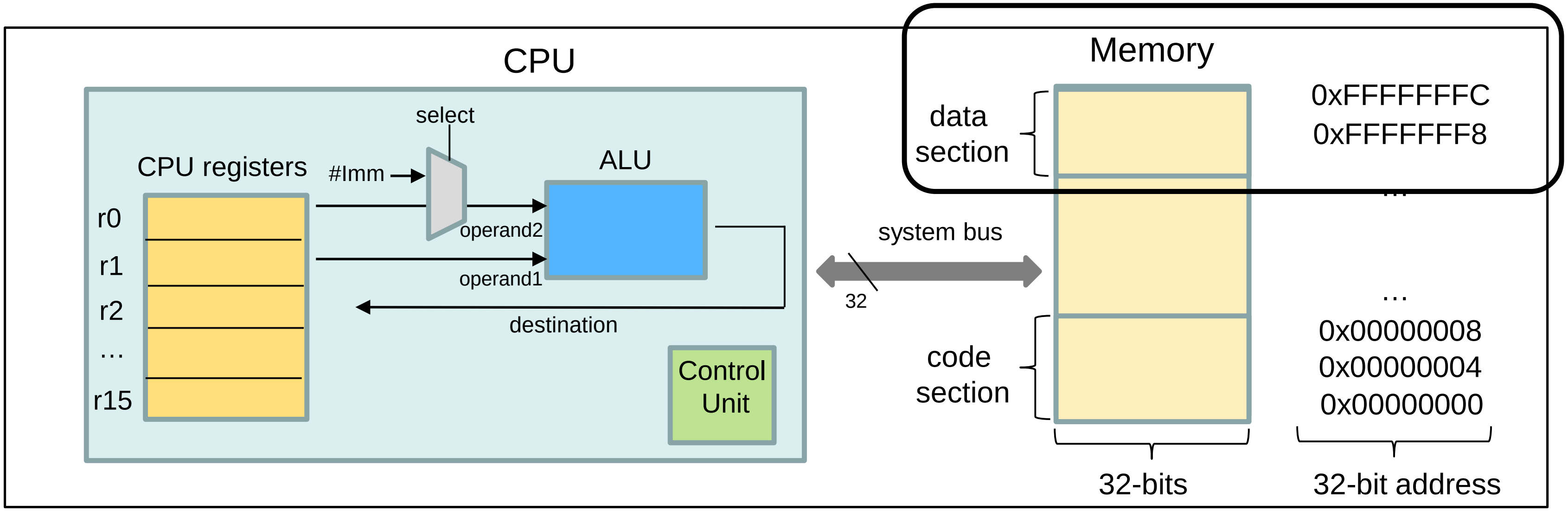
Defining variables in Memory

- In assembly, we can define variables in the data section of Memory
- Let’s look at an example:
  - Define a 4-byte variable named *myvar1* and initialise it to the value 3



```
 .balign 4
myvar1:
     .word 3
```

- The assembler directive .word states that the assembler should reserve 4 bytes. In this case, for the label myvar1. Note: the size of a word is 4 bytes

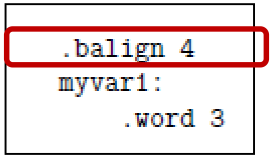


Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

Defining variables in Memory

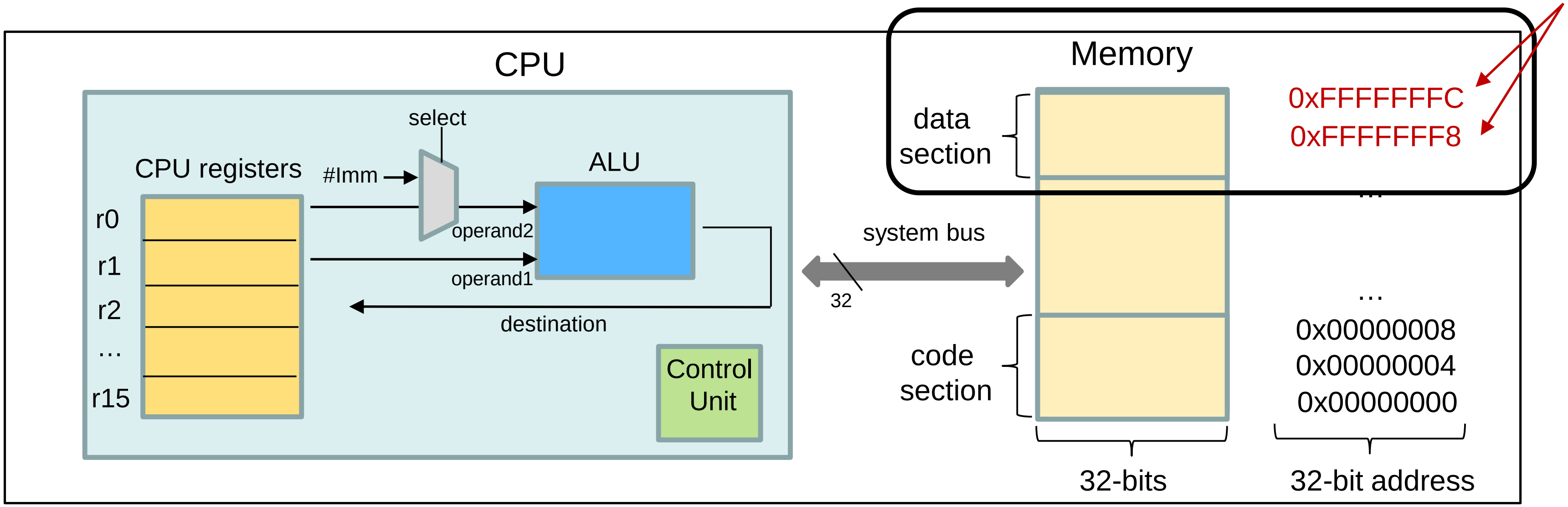
- In assembly, we can define variables in the data section of Memory
- Let’s look at an example:
  - Define a 4-byte variable named *myvar1* and initialise it to the value 3



```
    .balign 4
    myvar1:
         .word 3
```

- The assembler direct .balign ensures that the address of label myvar1 starts at a 4-byte boundary. This means that the address of myvar1 will be allocated to a value that is a multiple of 4

Addresses that are a multiple of 4



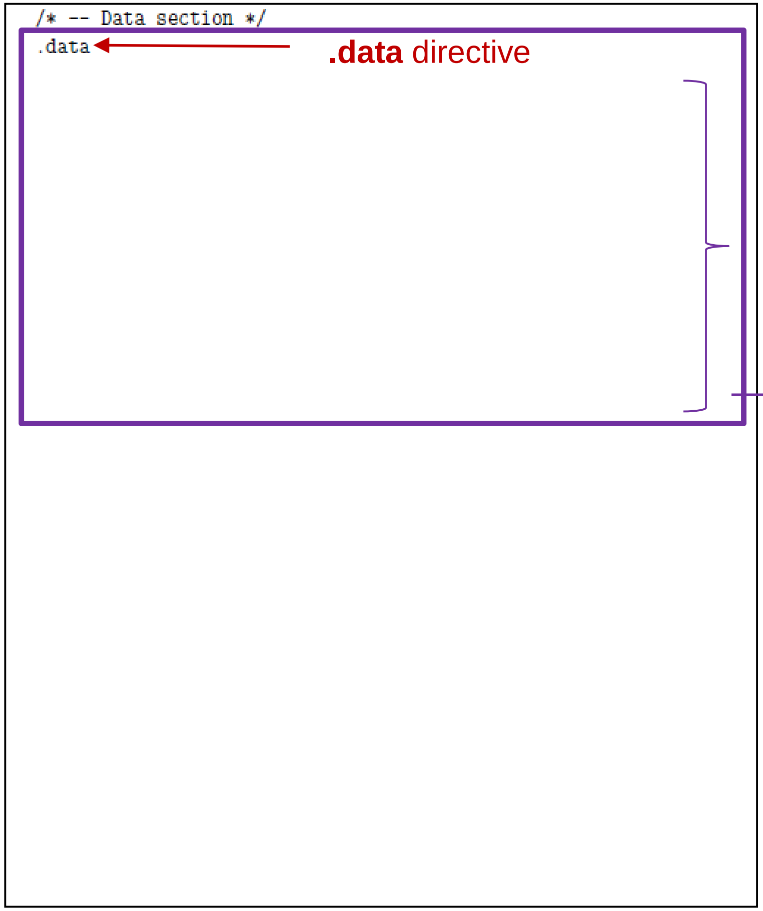
Simplified, conceptual block diagram of a computer

An Assembly Program: .data and .text

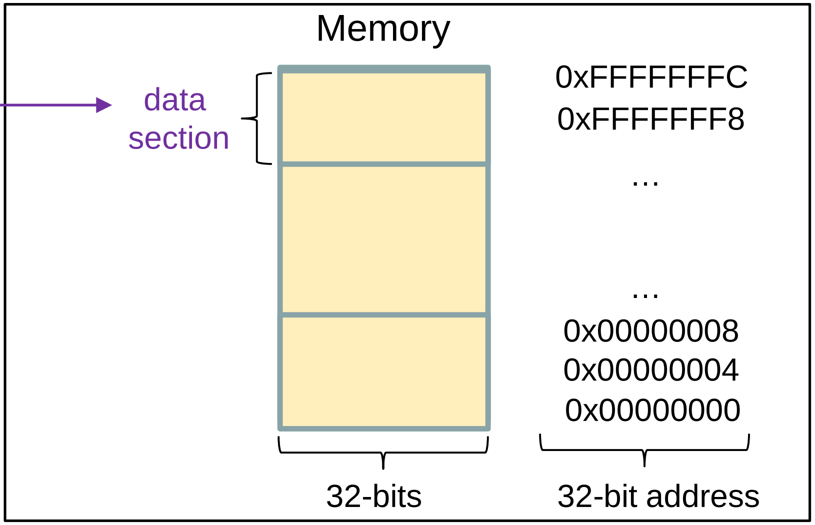
Raspberry Pi Assembler

An Assembly program: .data and .text

- In an assembly program:
  - the **.data** directive tells the assembler to store entities that follow in the data section



```
/* -- Data section */
.data
```

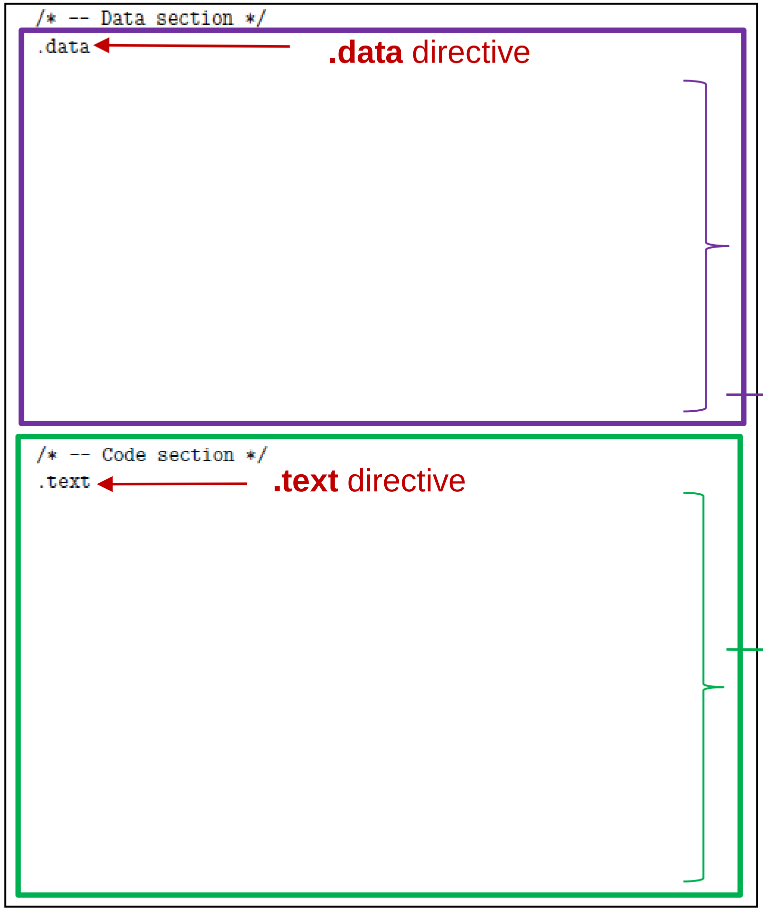


Memory block of a computer

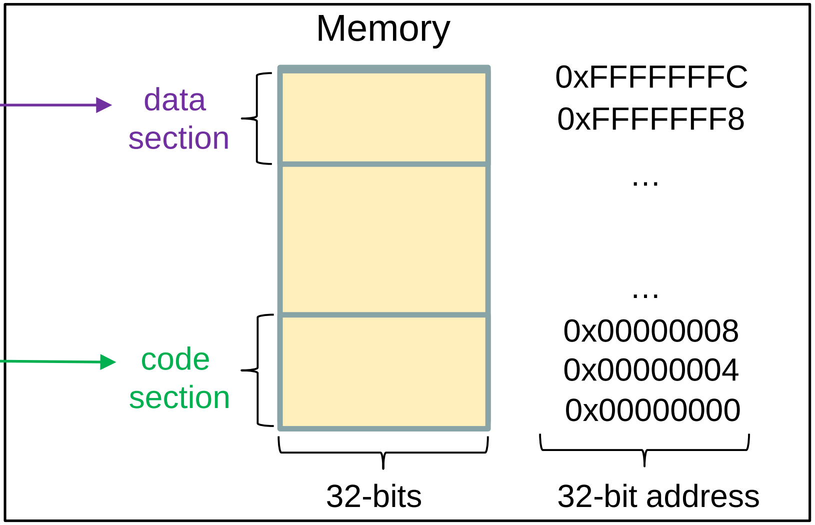
Raspberry Pi Assembler

An Assembly program: .data and .text

- In an assembly program:
  - the **.data** directive tells the assembler to store entities that follow in the data section
  - the **.text** directive tells the assembler to store entities that follow in the code section



```
/* -- Data section */
.data

/* -- Code section */
.text
```

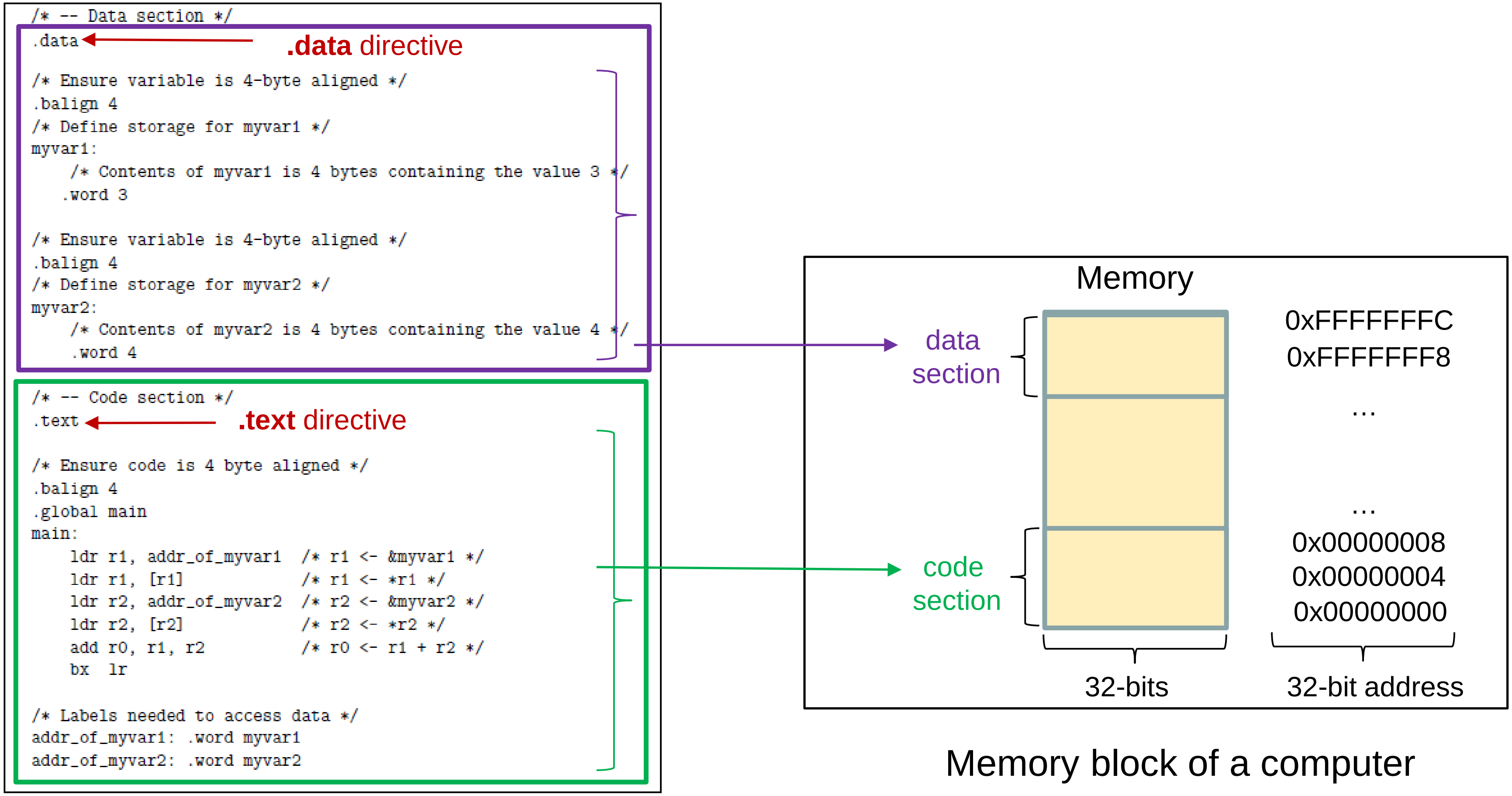


Memory block of a computer

Raspberry Pi Assembler

An Assembly program: .data and .text

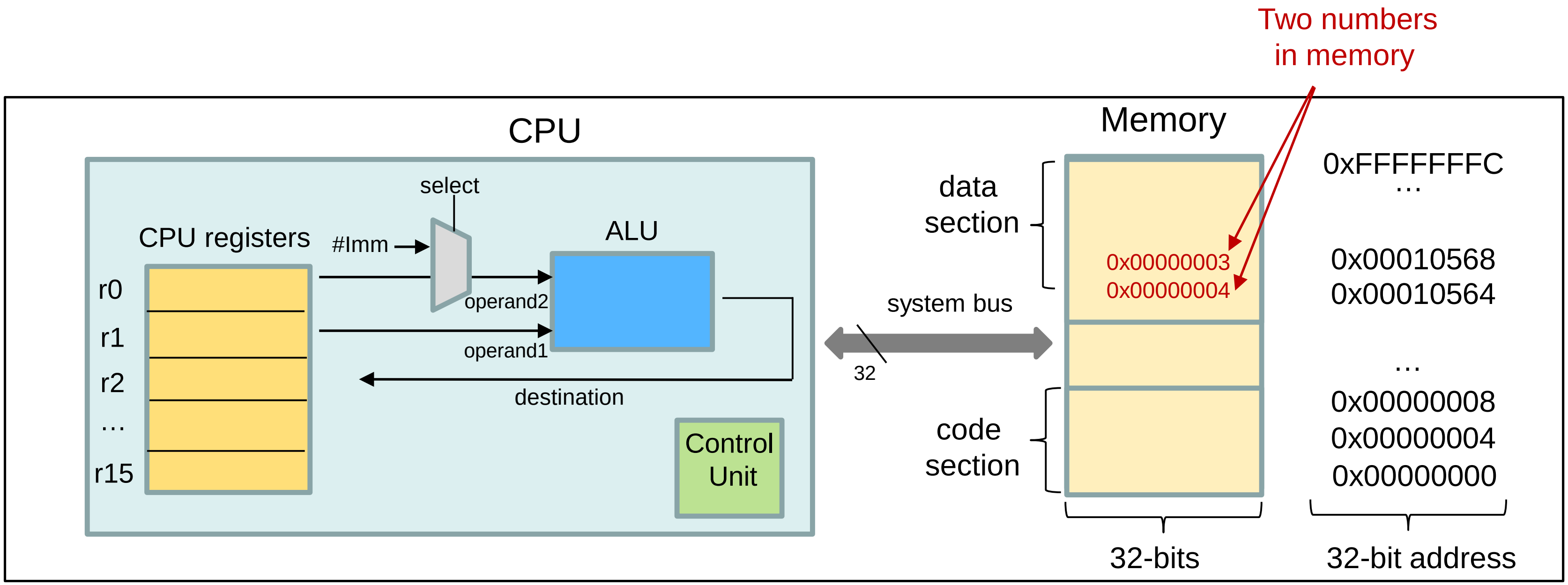
- In an assembly program:
  - the **.data** directive tells the assembler to store entities that follow in the data section
  - the **.text** directive tells the assembler to store entities that follow in the code section



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

Memory block of a computer

Assembly Program 1: Adding two numbers in memory

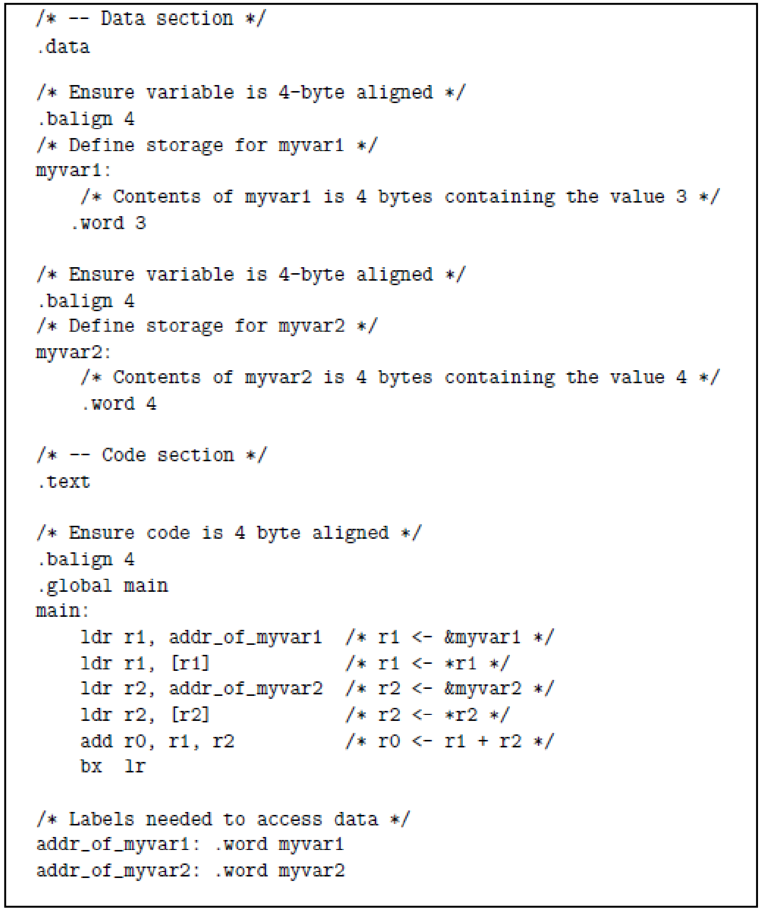


Simplified, conceptual block diagram of a computer

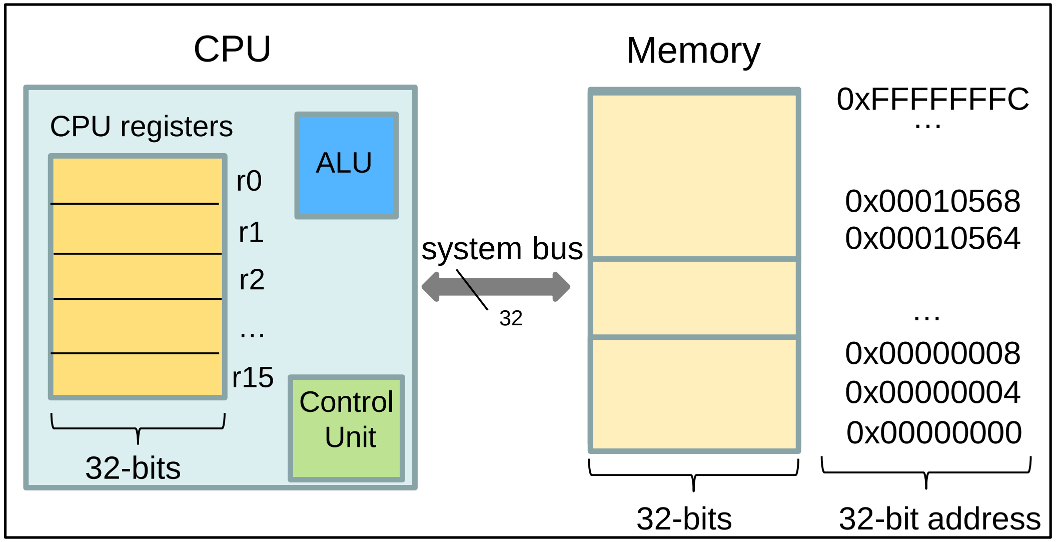
Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

- Let’s review assembly program 1 to add two numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```



Simplified block diagram of a modern computer

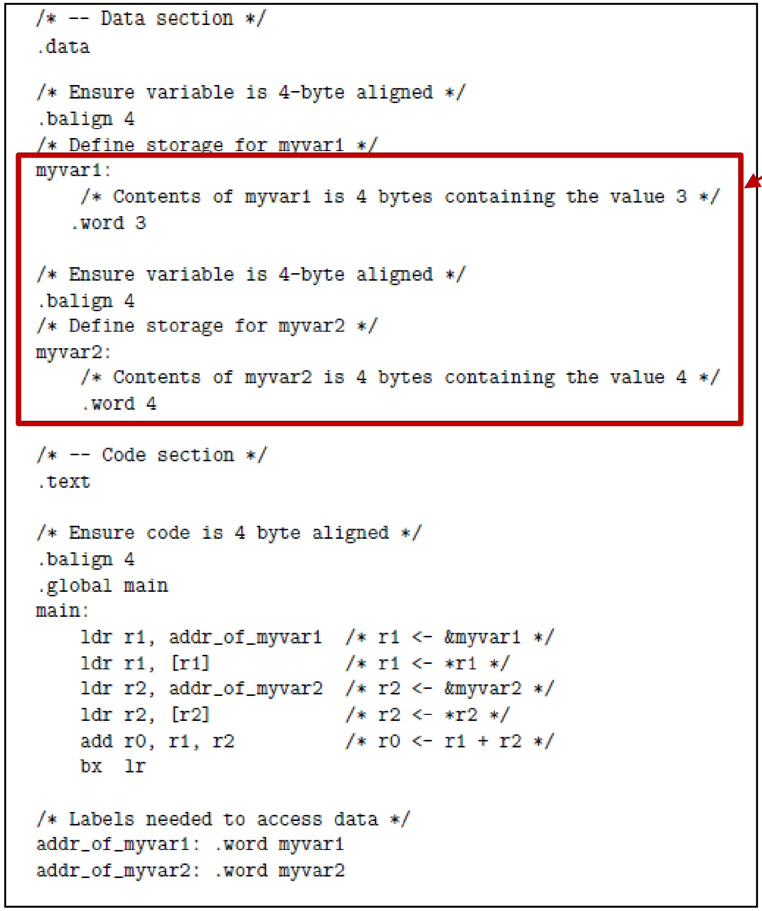
Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

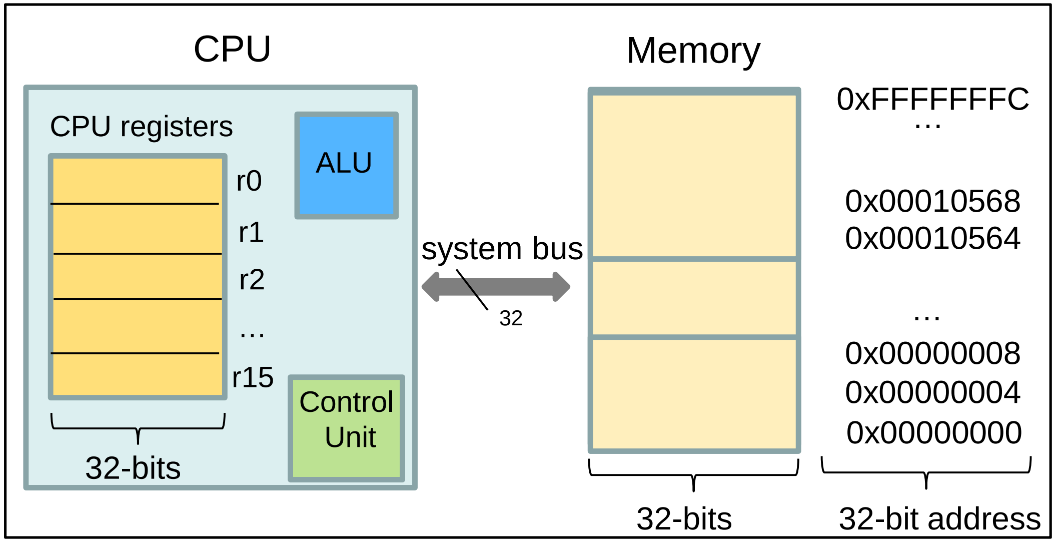
- Let’s review assembly program 1 to add two numbers in memory

- Define two 4 byte variables **myvar1** and **myvar2** and initialise them to the values 3 and 4 respectively.
- Ensure that this is done in the .data section of the program and not the .text section

- **Note:** in assembly, we cannot directly access a label in the .data section from the .text section and vice versa. This is because a program cannot modify instructions in the .text section, however a program can modify variables in the .data section.



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

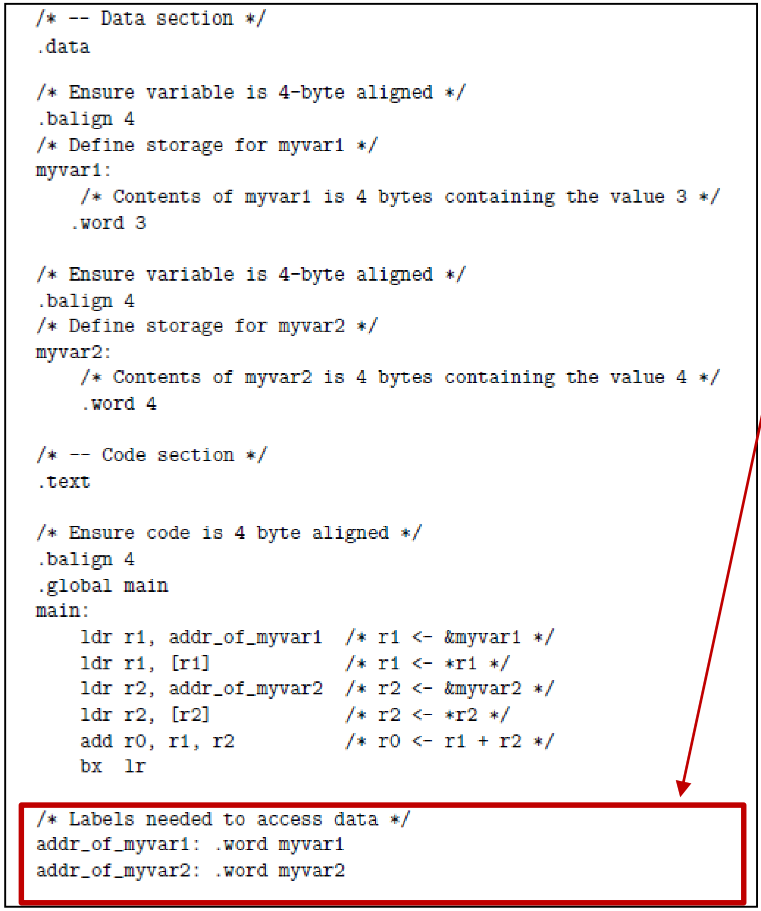


Simplified block diagram of a modern computer

Raspberry Pi Assembler

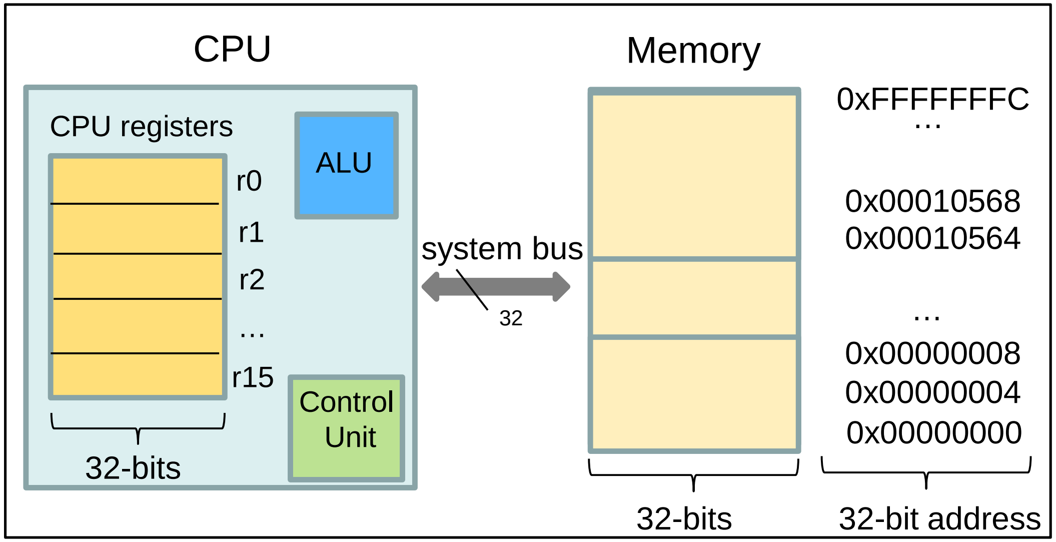
Assembly program 1: adding 2 numbers in memory

- Let’s review assembly program 1 to add two numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

- We define a label in the .text section of the program to refer to the labels in the .data section
- In this case, we defined **addr_of_myvar1** to refer to the address of **myvar1**, and **addr_of_myvar2** to refer to the address of **myvar2**.
- **Note:** the final address of **addr_of_myvar1** and **addr_of_myvar2** will be done by the Linker and not the assembler.

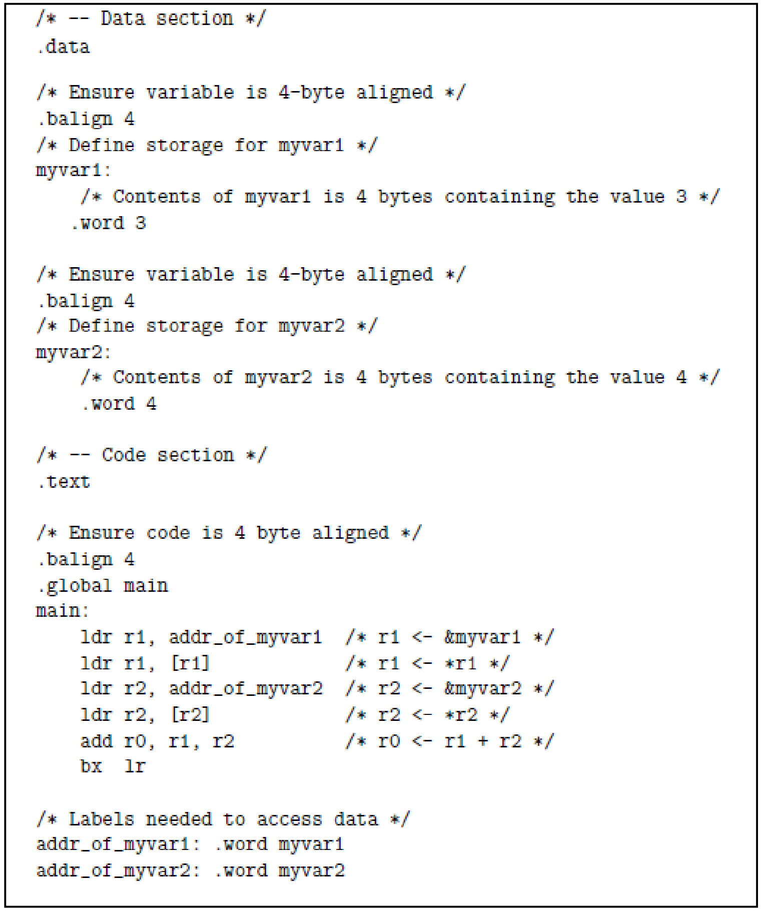


Simplified block diagram of a modern computer

Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

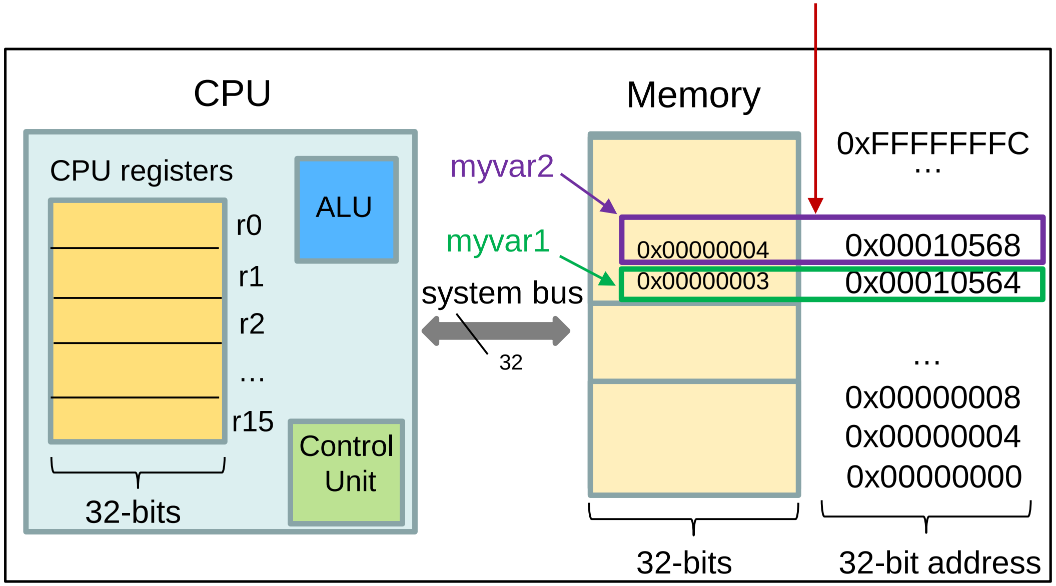
- Let’s review assembly program 1 to add two numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

Assume that after the linking step:

- the address of myvar1 is the value 0x00010564
- the address of myvar2 is the value 0x00010568



Simplified block diagram of a modern computer

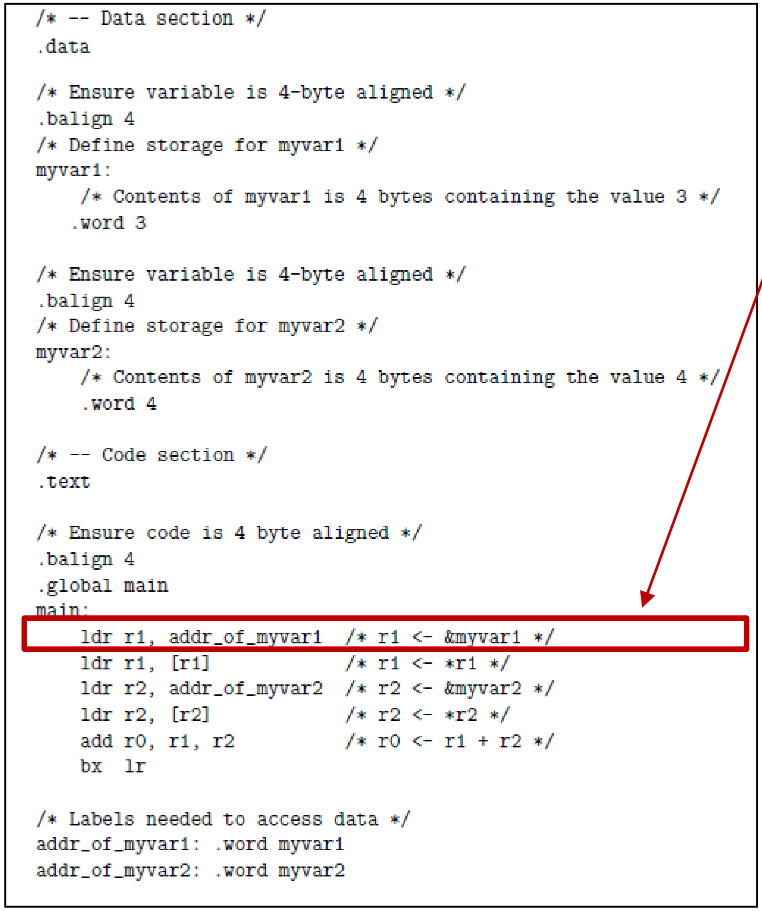
Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

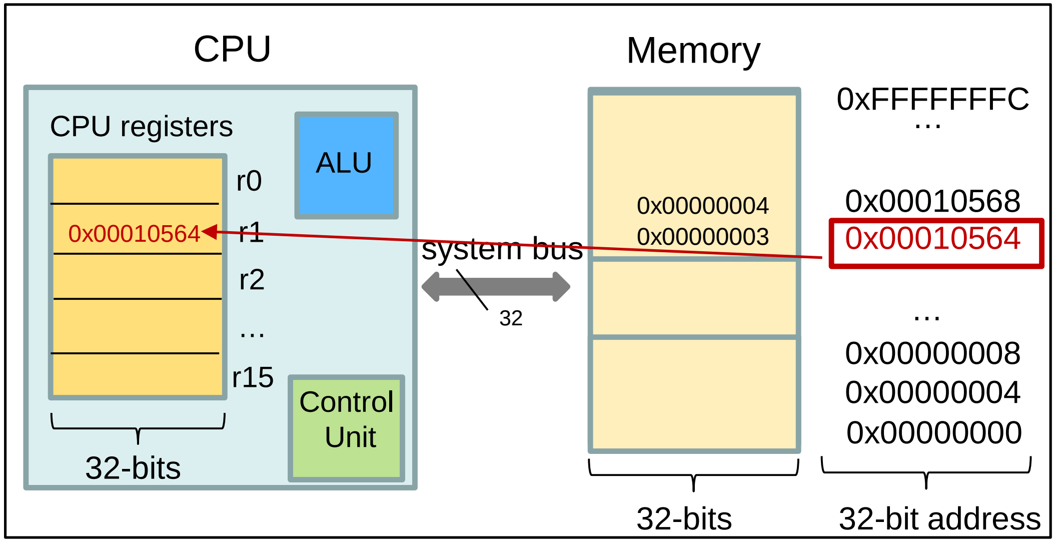
- Let’s review assembly program 1 to add two numbers in memory

After this line of code has executed:

- the address of myvar1 is loaded into CPU register r1



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

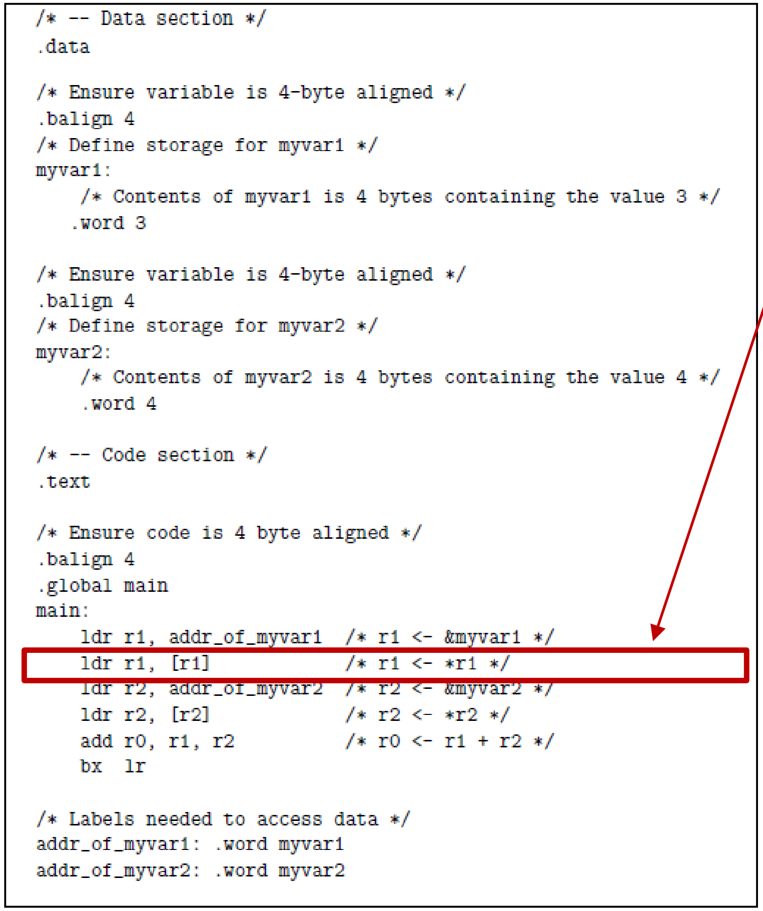


Simplified block diagram of a modern computer

Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

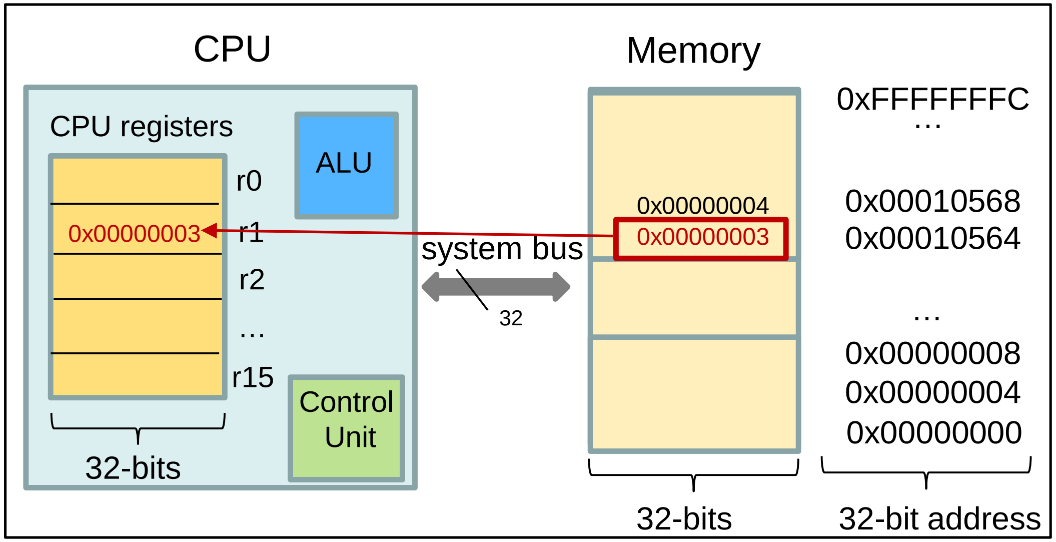
- Let’s review assembly program 1 to add two numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

- The memory address specified by CPU register r1 (ie. 0x00010564) is loaded into CPU register r1

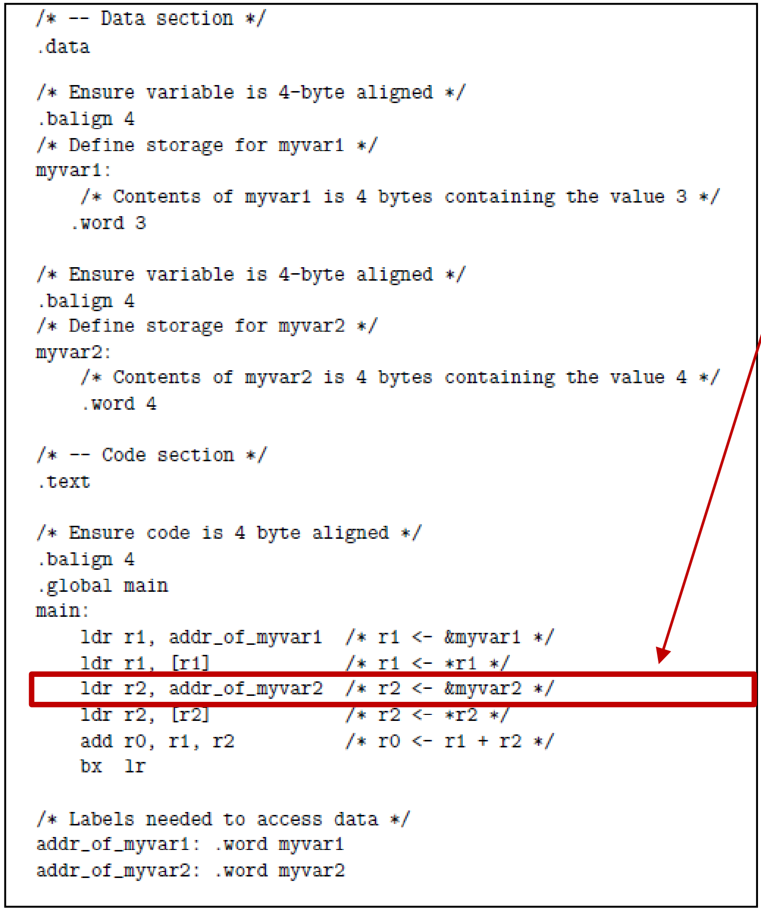


Simplified block diagram of a modern computer

Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

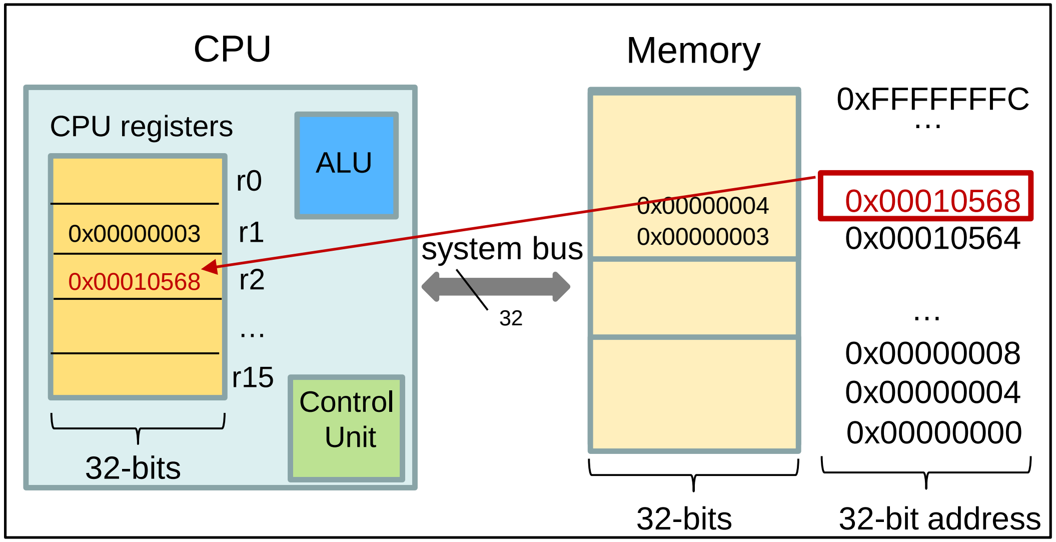
- Let’s review assembly program 1 to add two numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

- the address of myvar2 is loaded into CPU register r2



Simplified block diagram of a modern computer

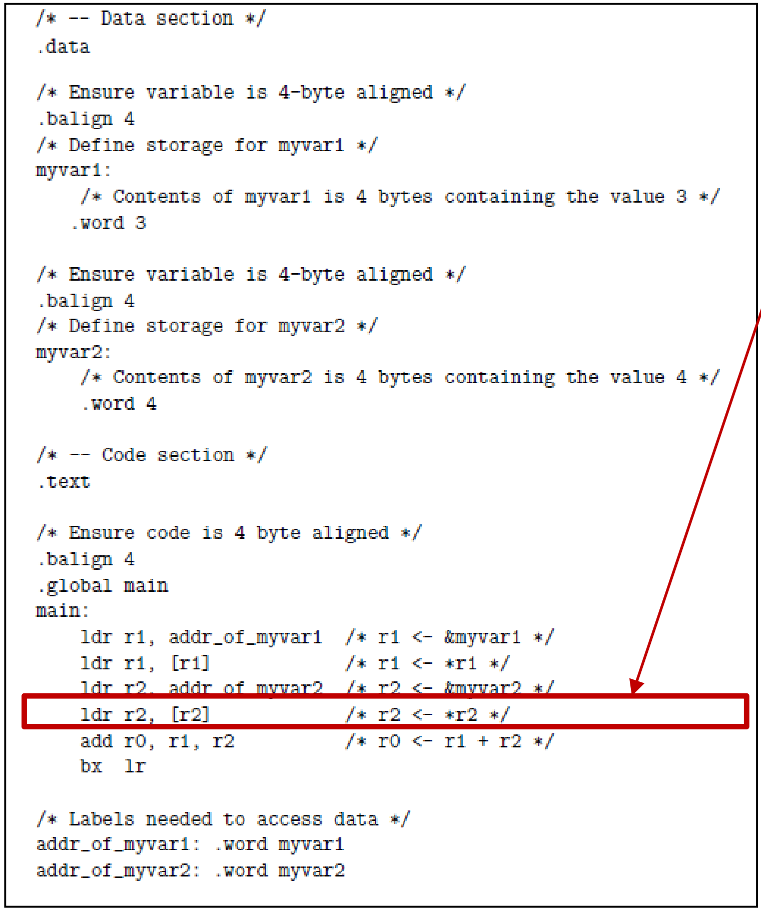
Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

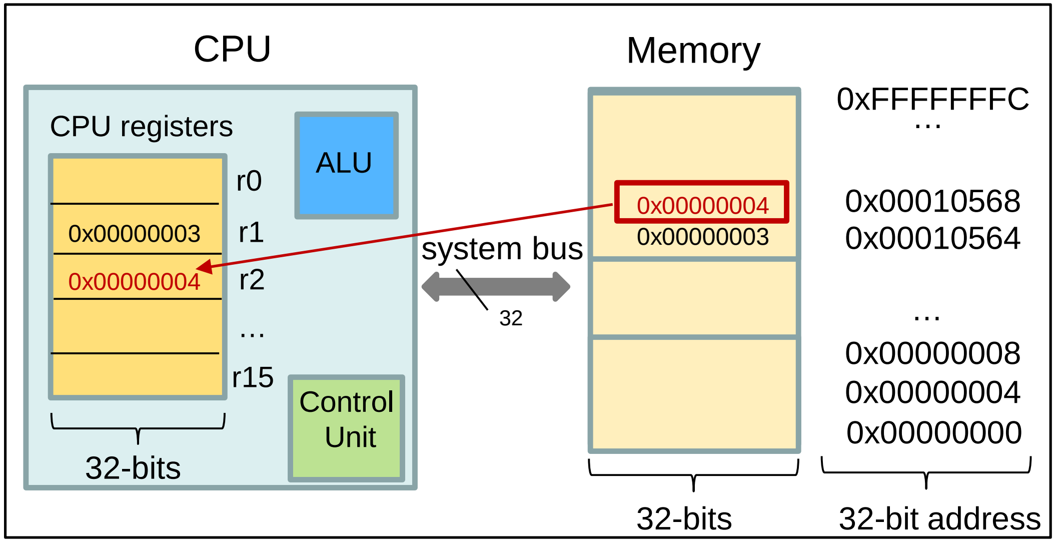
- Let’s review assembly program 1 to add two numbers in memory

After this line of code has executed:

- The memory address specified by CPU register r2 (ie. 0x00010568) is loaded into CPU register r2



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```



Simplified block diagram of a modern computer

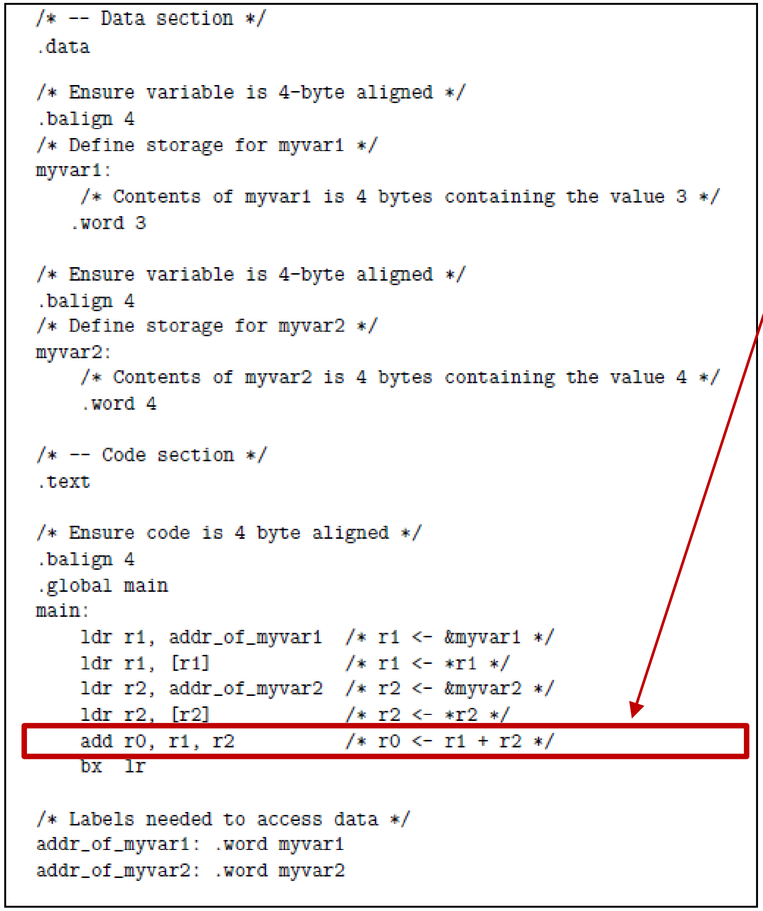
Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

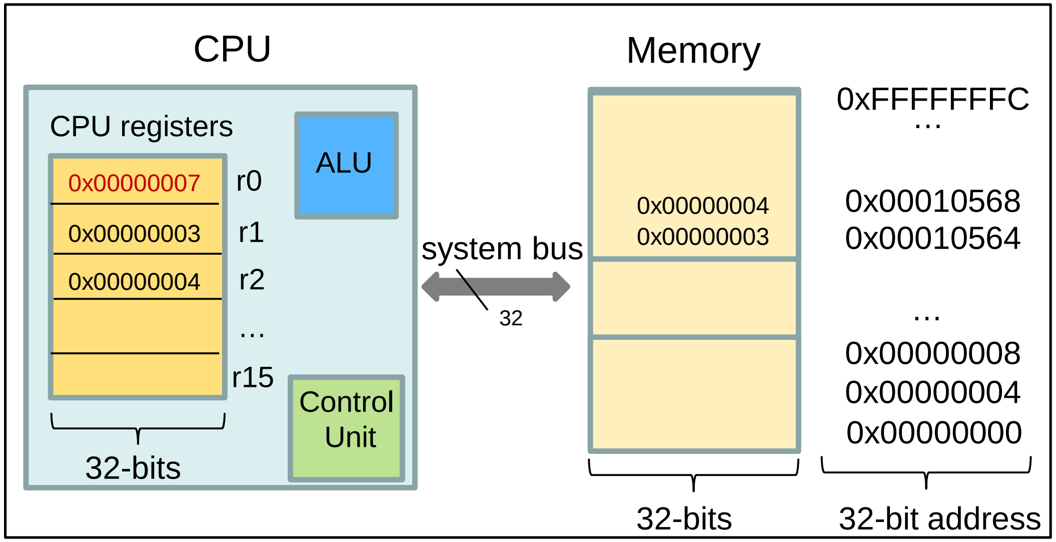
- Let’s review assembly program 1 to add two numbers in memory

After this line of code has executed:

- The values of CPU registers r1 and r2 are added up and put into r0: r0 = r1 + r2



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

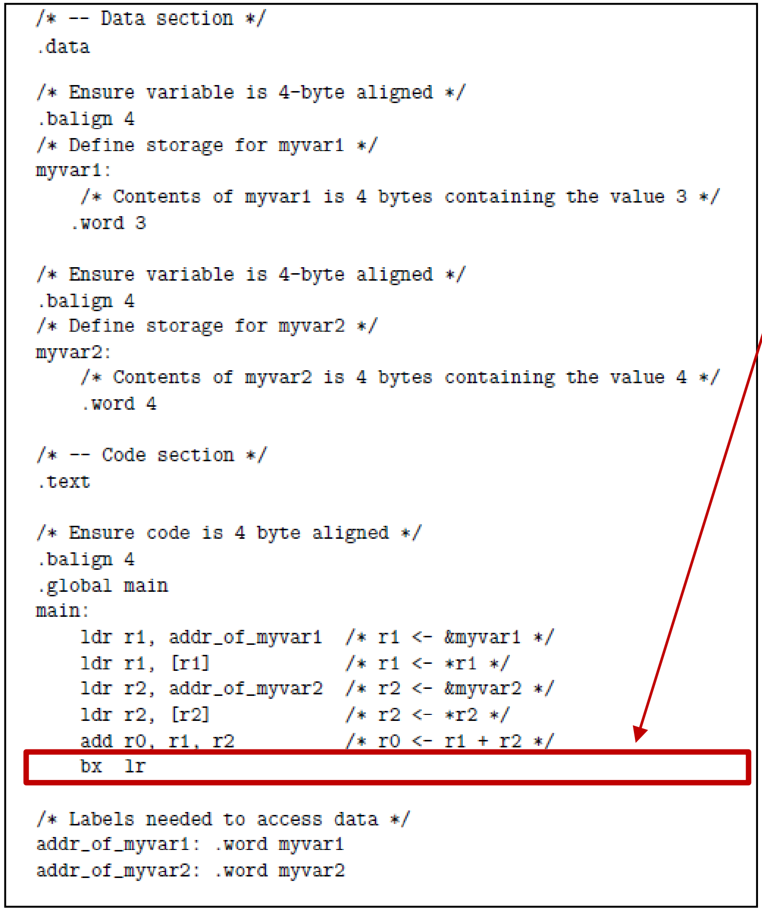


Simplified block diagram of a modern computer

Raspberry Pi Assembler

Assembly program 1: adding 2 numbers in memory

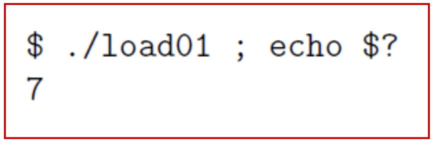
- Let’s review assembly program 1 to add two numbers in memory



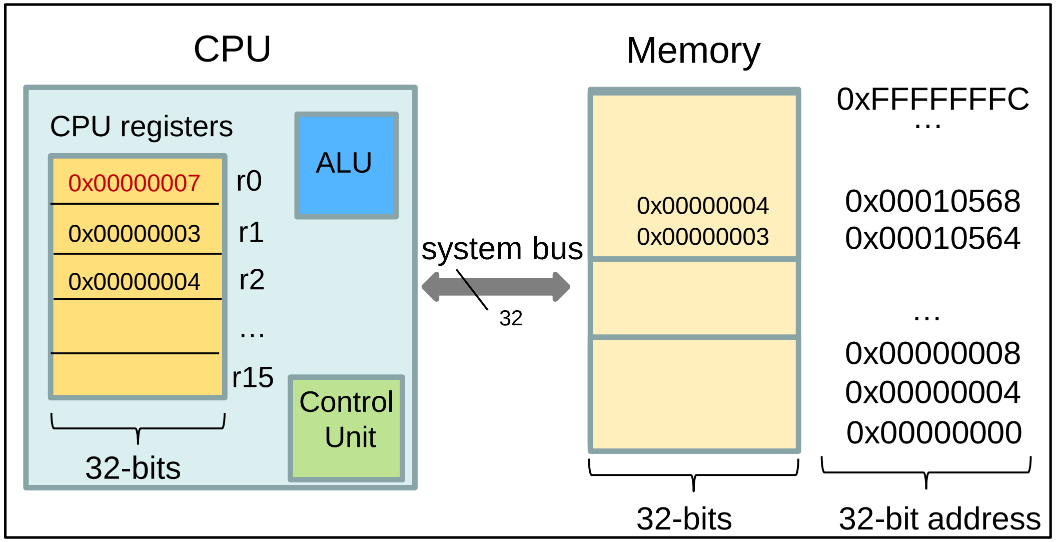
```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is 4 bytes containing the value 3 */
   .word 3

/* Ensure variable is 4-byte aligned */
.balign 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is 4 bytes containing the value 4 */
    .word 4

/* -- Code section */
.text

/* Ensure code is 4 byte aligned */
.balign 4
.global main
main:
    ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
    ldr r1, [r1]            /* r1 <- *r1 */
    ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
    ldr r2, [r2]            /* r2 <- *r2 */
    add r0, r1, r2          /* r0 <- r1 + r2 */
    bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

- The program ends and the contents of the CPU register r0 is displayed to the user



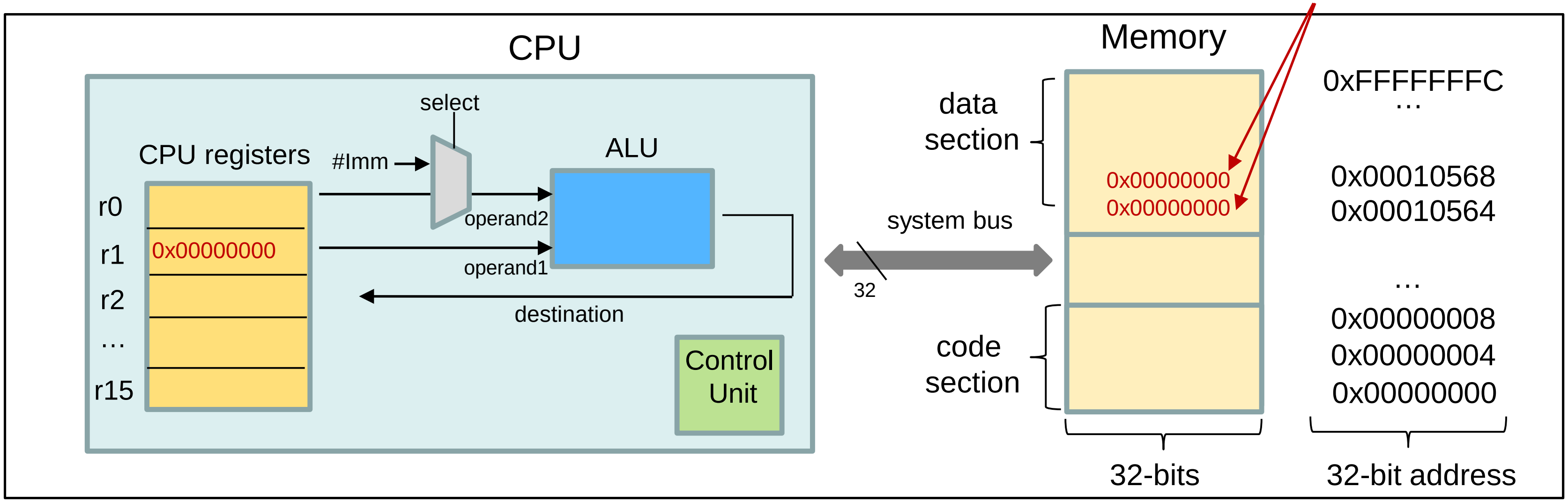
```
$ ./load01 ; echo $?
7
```



Simplified block diagram of a modern computer

Assembly Program 2:
Adding two numbers in memory

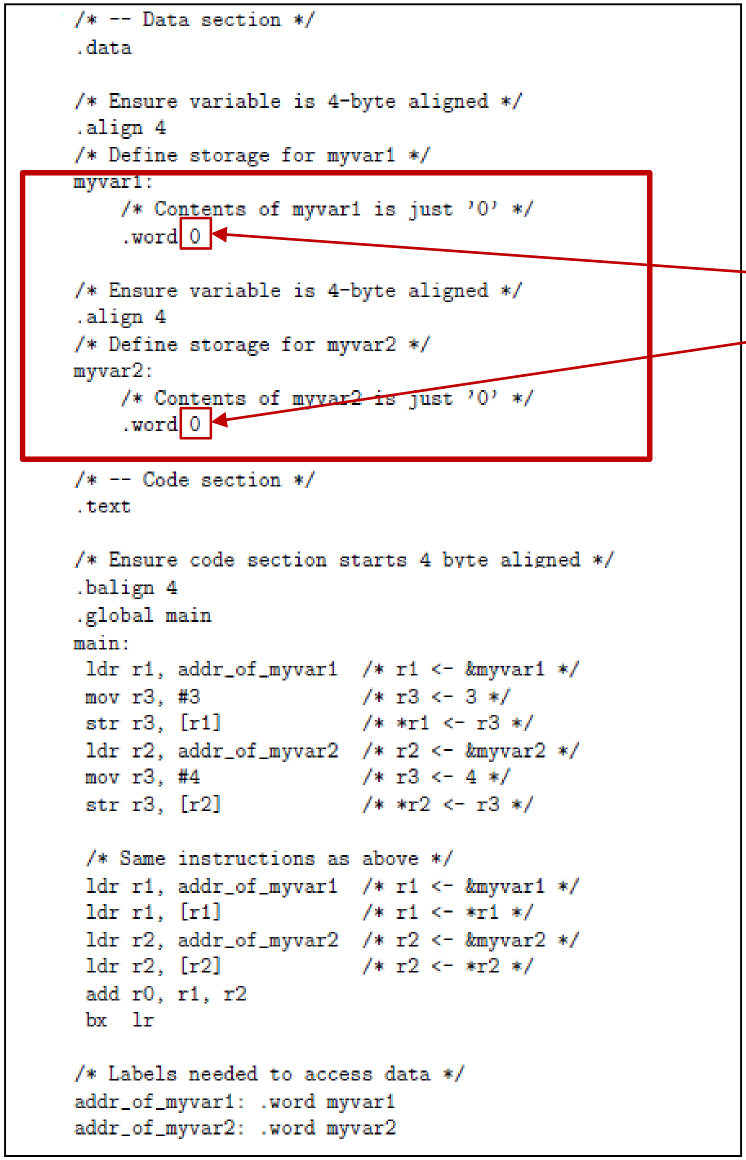
Two numbers in memory initiliased to zero. Later we write the value 3 and 4 into these memory addresses using the STR instruction



Simplified, conceptual block diagram of a computer

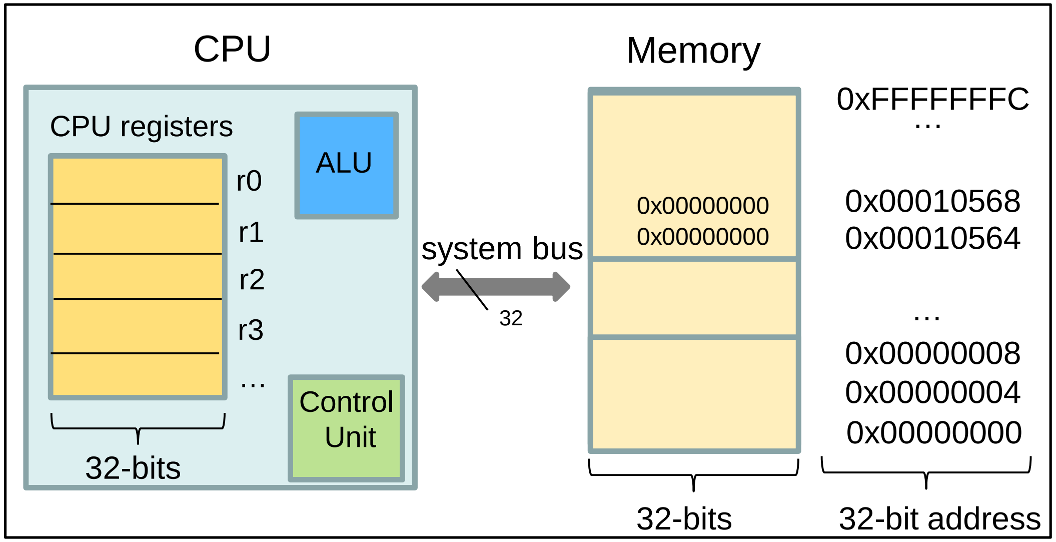
Raspberry Pi Assembler

Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '0' */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '0' */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

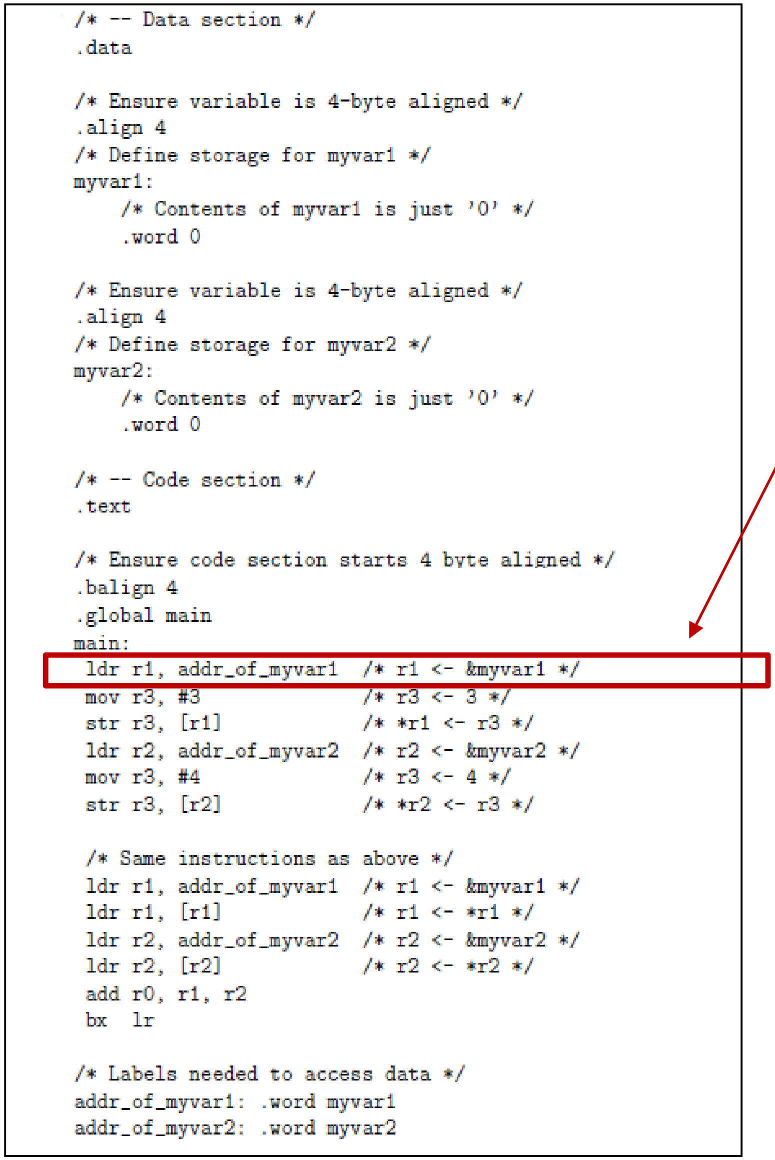
Initialise myvar1 and myvar2 to 0x00000000



Simplified block diagram of a modern computer

Raspberry Pi Assembler

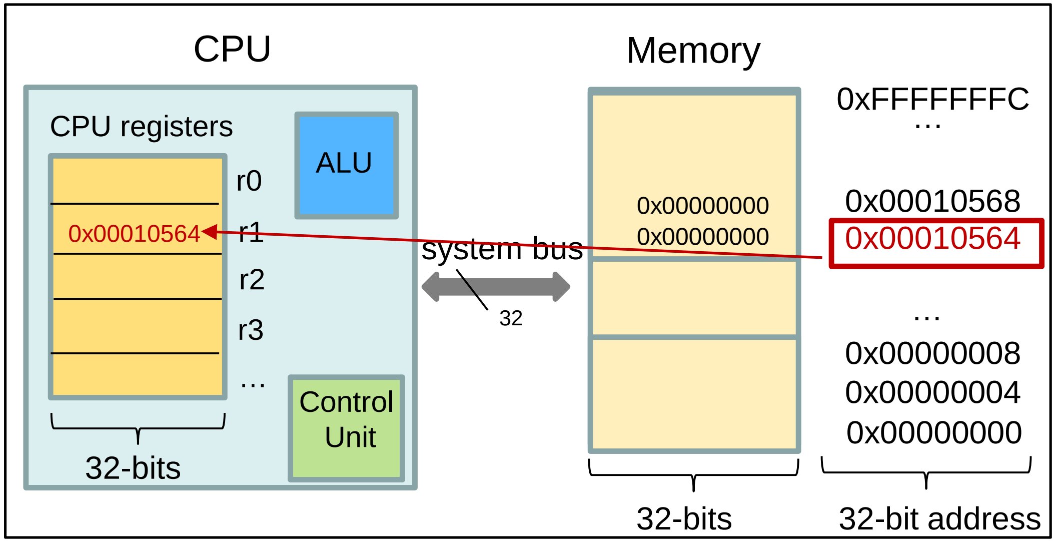
Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '0' */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '0' */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

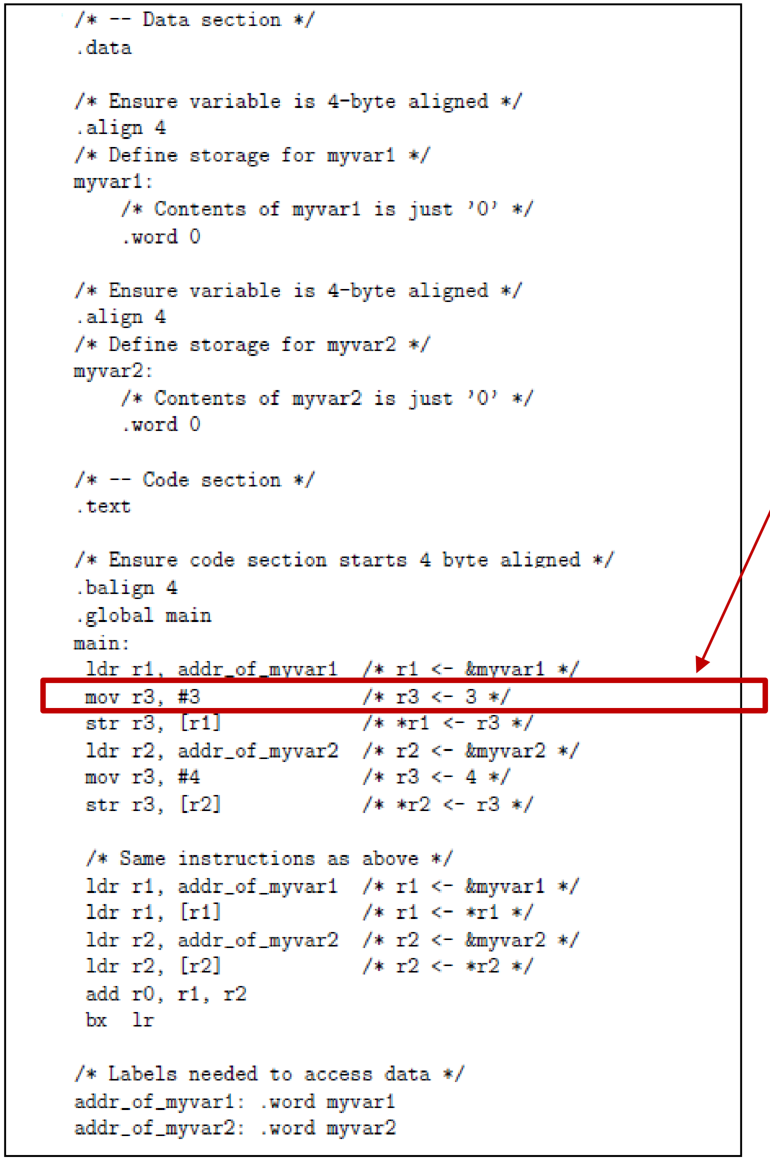
- the address of myvar1 is loaded into CPU register r1



Simplified block diagram of a modern computer

Raspberry Pi Assembler

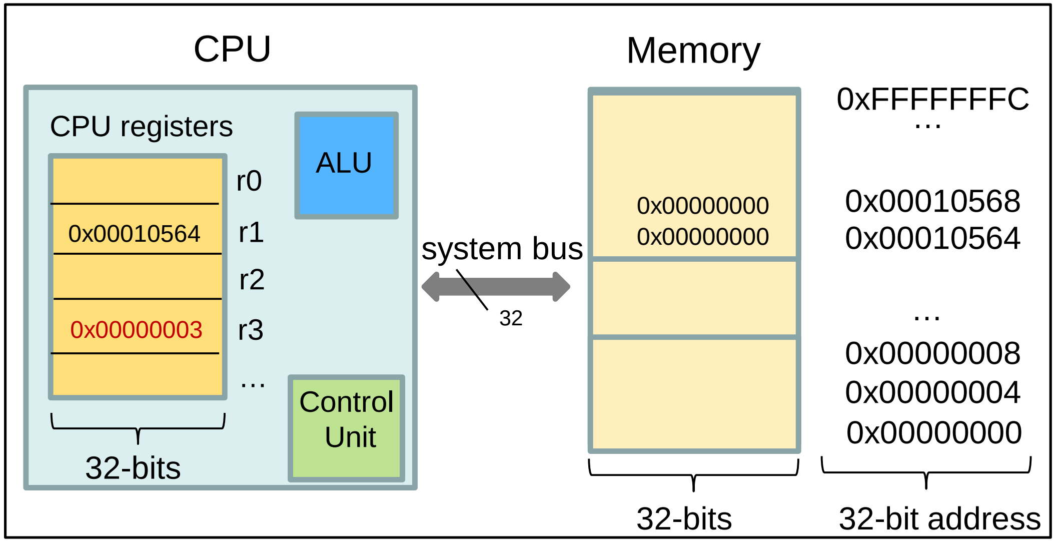
Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '0' */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '0' */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

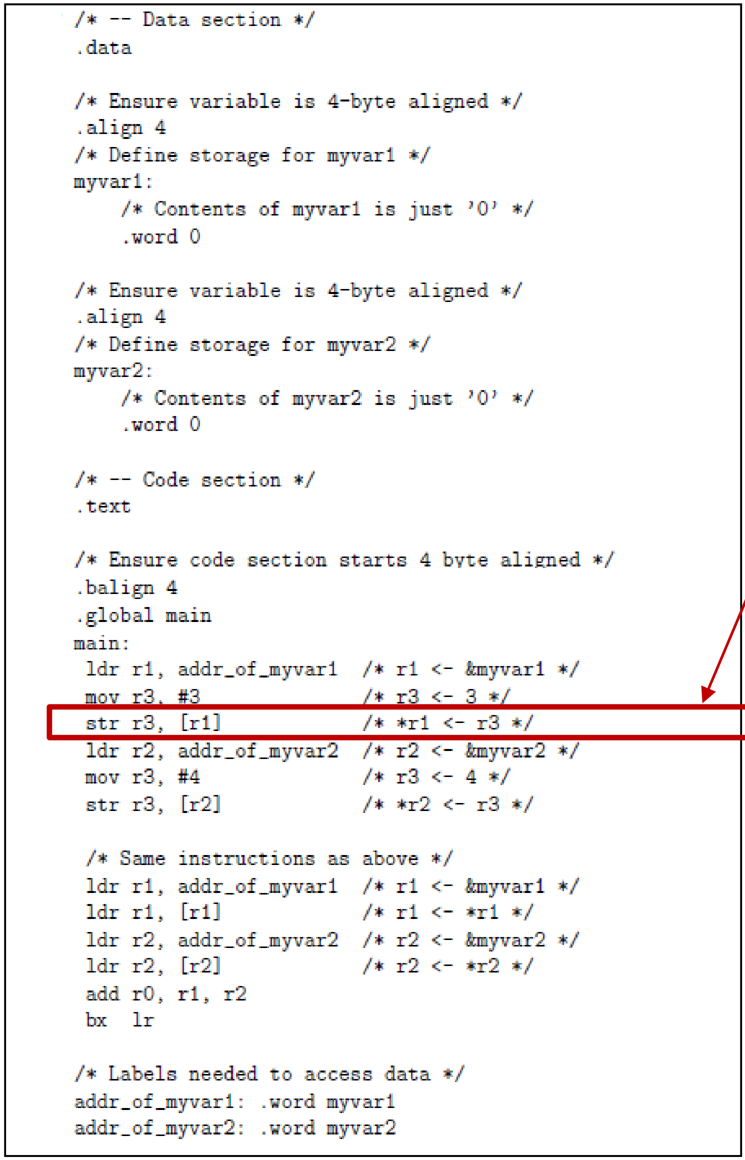
- The value of 3 is moved into CPU register r3



Simplified block diagram of a modern computer

Raspberry Pi Assembler

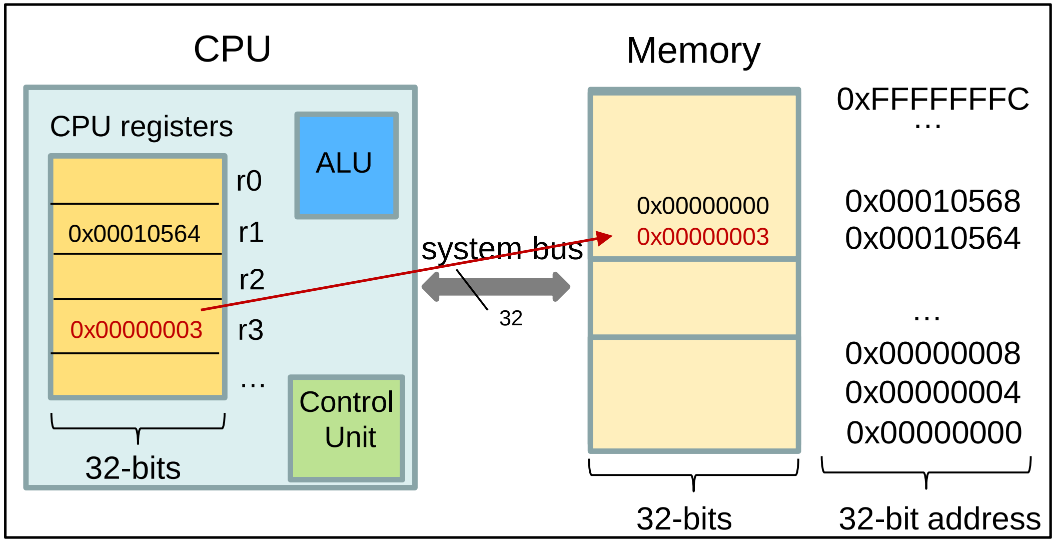
Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just ’0’ */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just ’0’ */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

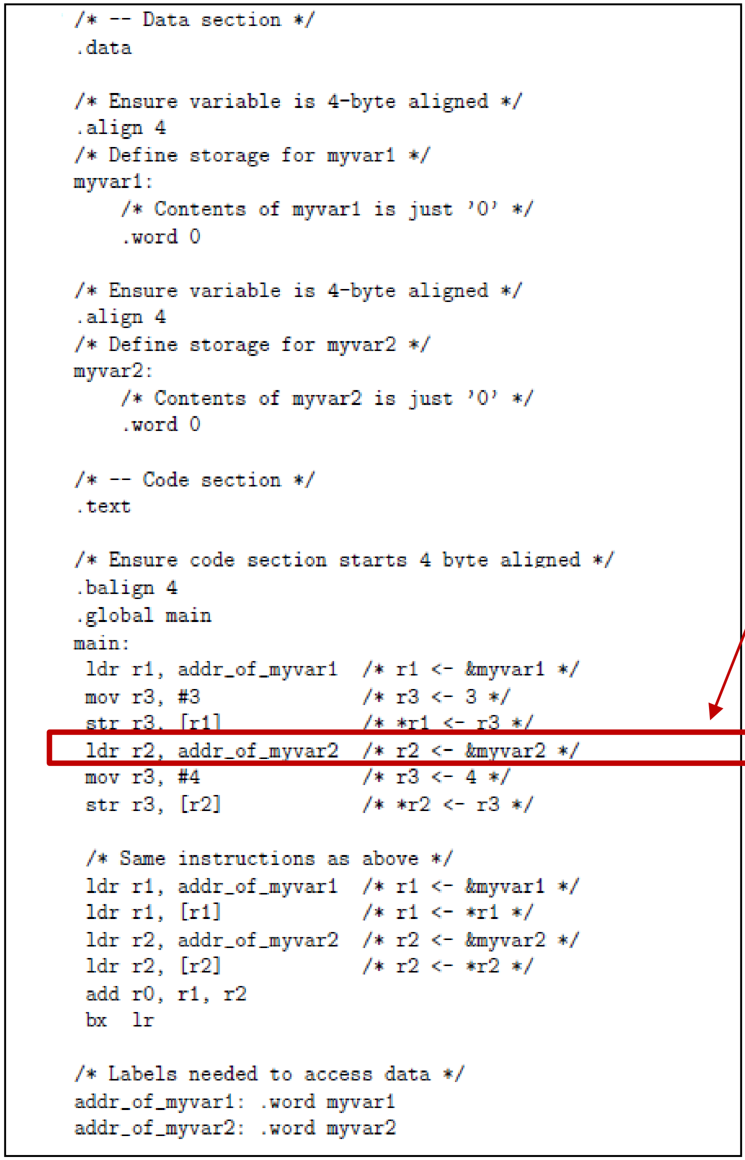
- The contents of the CPU register r3 is transferred to the memory address specified by CPU register r1 (ie. 0x00010564)



Simplified block diagram of a modern computer

Raspberry Pi Assembler

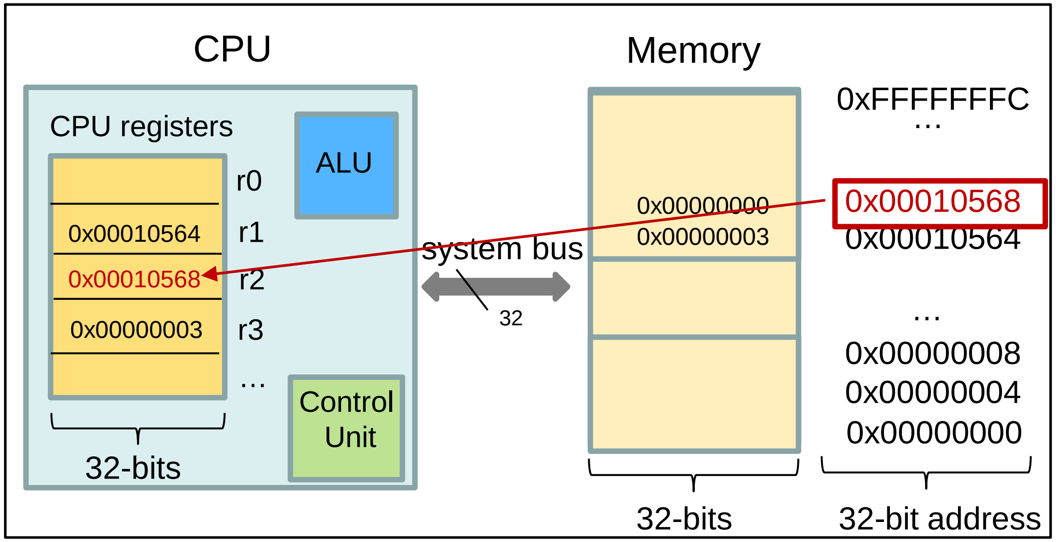
Assembly program 2: adding 2 numbers in memory



```
 /* -- Data section */
 .data

 /* Ensure variable is 4-byte aligned */
 .align 4
 /* Define storage for myvar1 */
 myvar1:
     /* Contents of myvar1 is just '0' */
     .word 0

 /* Ensure variable is 4-byte aligned */
 .align 4
 /* Define storage for myvar2 */
 myvar2:
     /* Contents of myvar2 is just '0' */
     .word 0

 /* -- Code section */
 .text

 /* Ensure code section starts 4 byte aligned */
 .balign 4
 .global main
 main:
  ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
  mov r3, #3              /* r3 <- 3 */
  str r3, [r1]            /* *r1 <- r3 */
  ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
  mov r3, #4              /* r3 <- 4 */
  str r3, [r2]            /* *r2 <- r3 */

  /* Same instructions as above */
  ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
  ldr r1, [r1]            /* r1 <- *r1 */
  ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
  ldr r2, [r2]            /* r2 <- *r2 */
  add r0, r1, r2
  bx  lr

 /* Labels needed to access data */
 addr_of_myvar1: .word myvar1
 addr_of_myvar2: .word myvar2
```

After this line of code has executed:

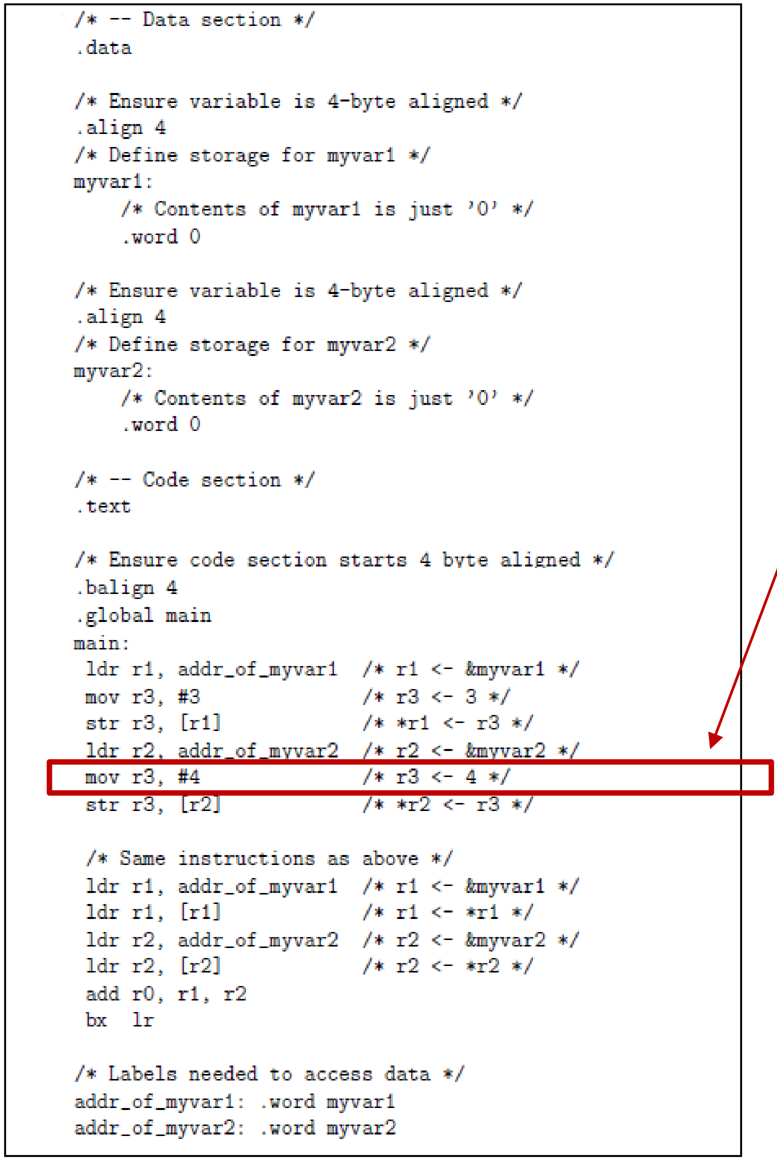
- the address of myvar1 is loaded into CPU register r2



Simplified block diagram of a modern computer

Raspberry Pi Assembler

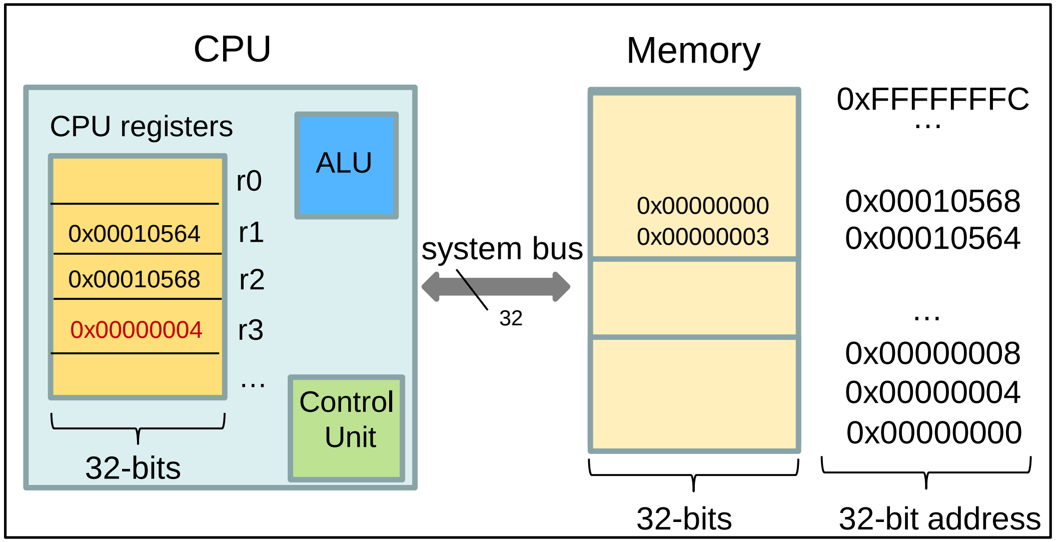
Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '0' */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '0' */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

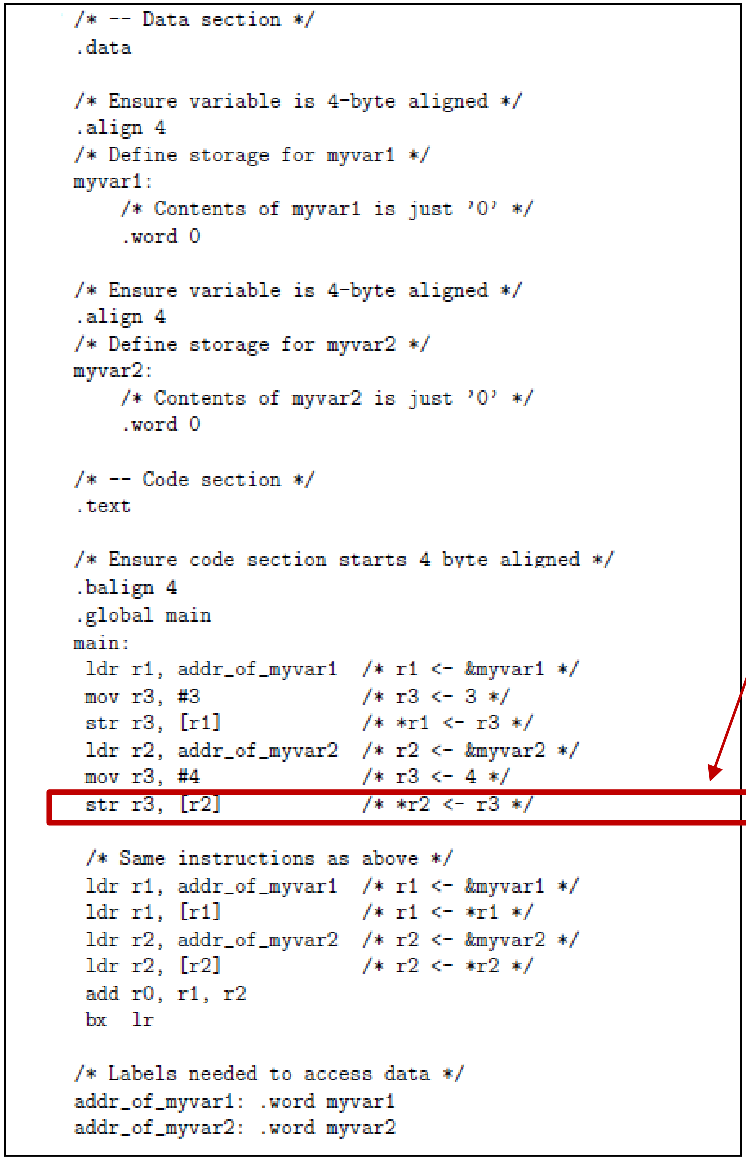
- The value of 4 is moved into CPU register r3



Simplified block diagram of a modern computer

Raspberry Pi Assembler

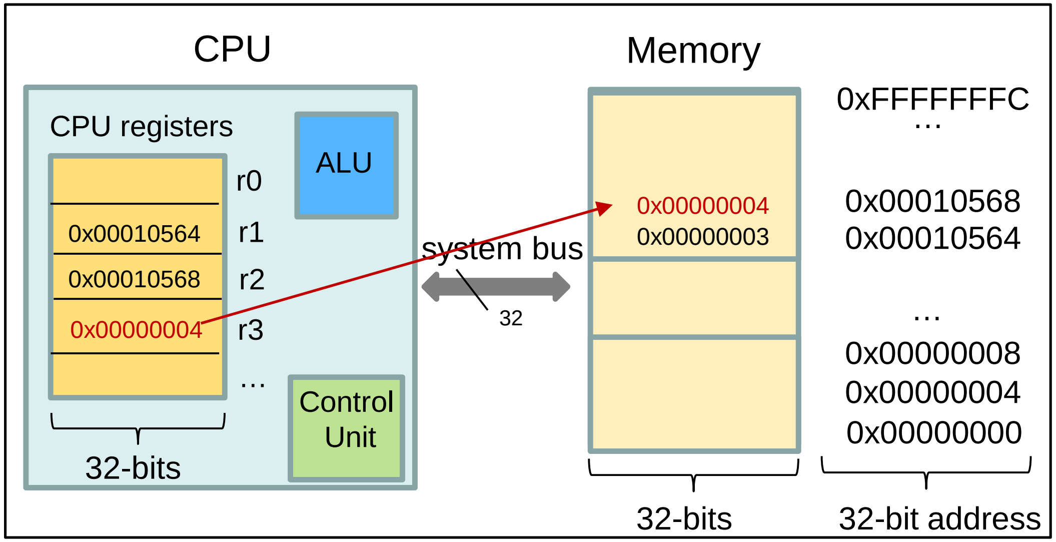
Assembly program 2: adding 2 numbers in memory



```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just ’0’ */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just ’0’ */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

After this line of code has executed:

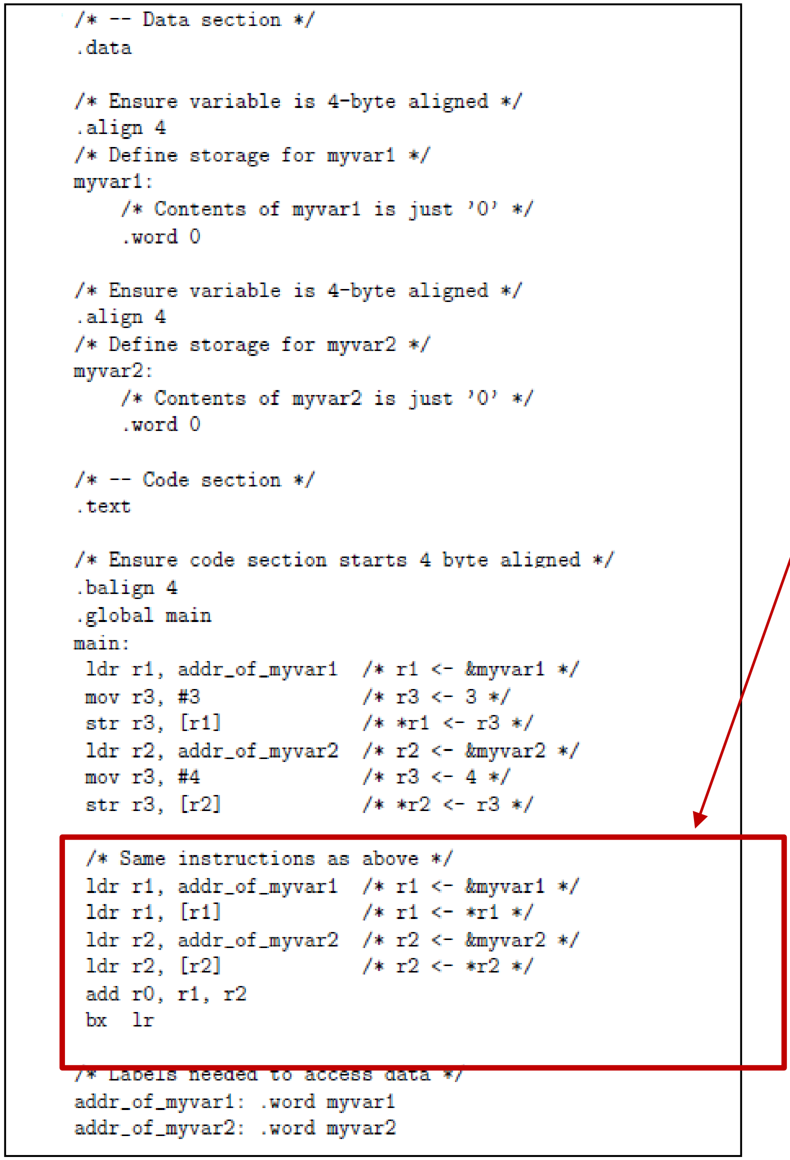
- The contents of the CPU register r3 is transferred to the memory address specified by CPU register r2 (ie. 0x00010568)



Simplified block diagram of a modern computer

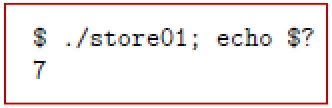
Raspberry Pi Assembler

Assembly program 2: adding 2 numbers in memory

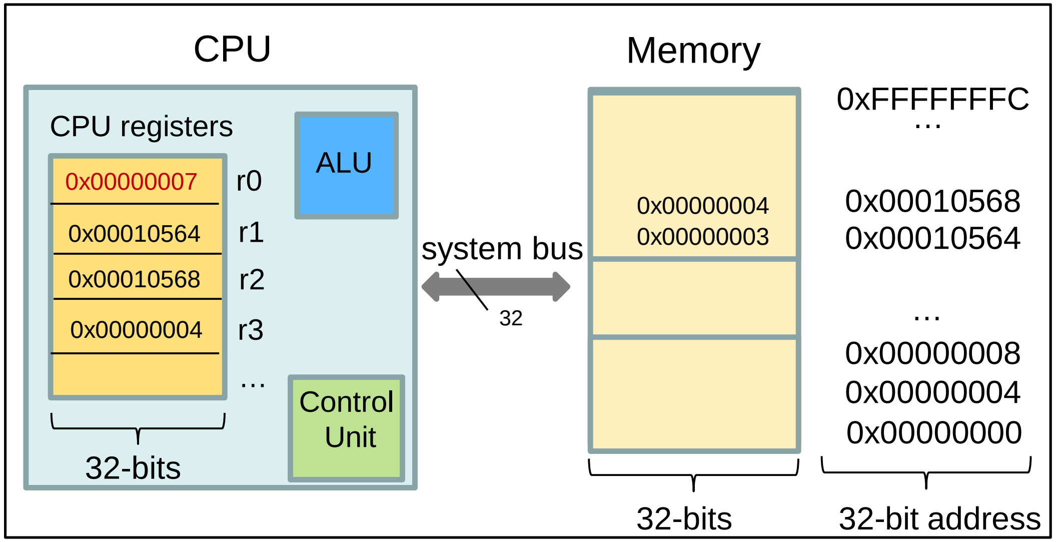


```
/* -- Data section */
.data

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar1 */
myvar1:
    /* Contents of myvar1 is just '0' */
    .word 0

/* Ensure variable is 4-byte aligned */
.align 4
/* Define storage for myvar2 */
myvar2:
    /* Contents of myvar2 is just '0' */
    .word 0

/* -- Code section */
.text

/* Ensure code section starts 4 byte aligned */
.balign 4
.global main
main:
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 mov r3, #3              /* r3 <- 3 */
 str r3, [r1]            /* *r1 <- r3 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 mov r3, #4              /* r3 <- 4 */
 str r3, [r2]            /* *r2 <- r3 */

 /* Same instructions as above */
 ldr r1, addr_of_myvar1  /* r1 <- &myvar1 */
 ldr r1, [r1]            /* r1 <- *r1 */
 ldr r2, addr_of_myvar2  /* r2 <- &myvar2 */
 ldr r2, [r2]            /* r2 <- *r2 */
 add r0, r1, r2
 bx  lr

/* Labels needed to access data */
addr_of_myvar1: .word myvar1
addr_of_myvar2: .word myvar2
```

Same code as program 1



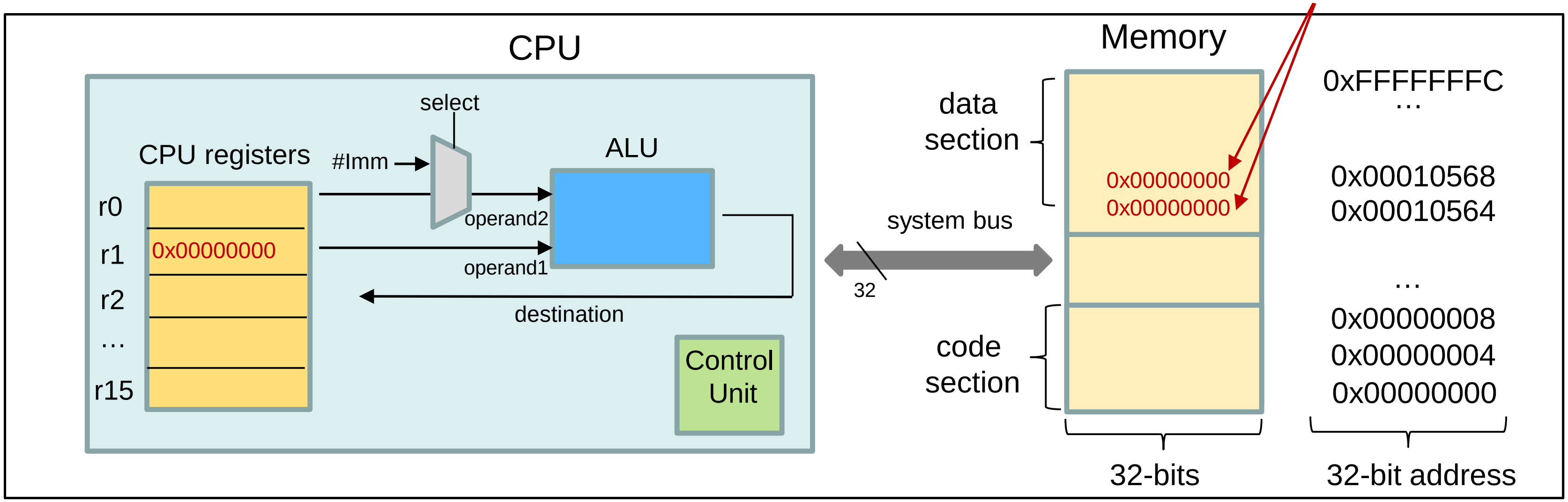
```
$ ./store01; echo $?
7
```



Simplified block diagram of a modern computer

Assembly Program 3:
Simpler version of program 2

Two numbers in memory initiliased to zero. Later we write the value 3 and 4 into these memory addresses using the STR instruction

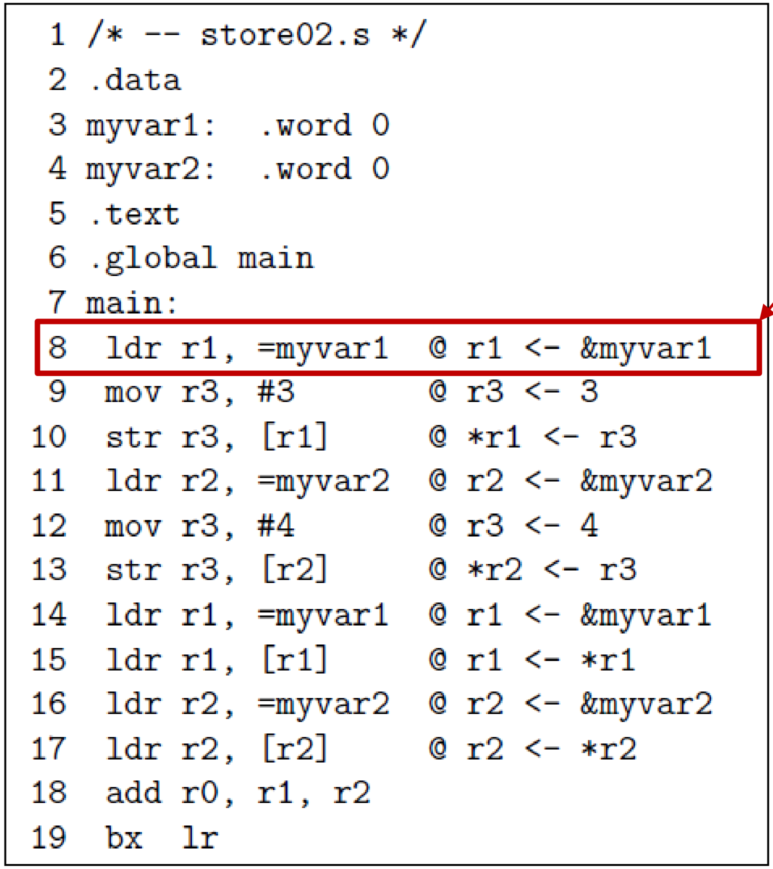


Simplified, conceptual block diagram of a computer

Raspberry Pi Assembler

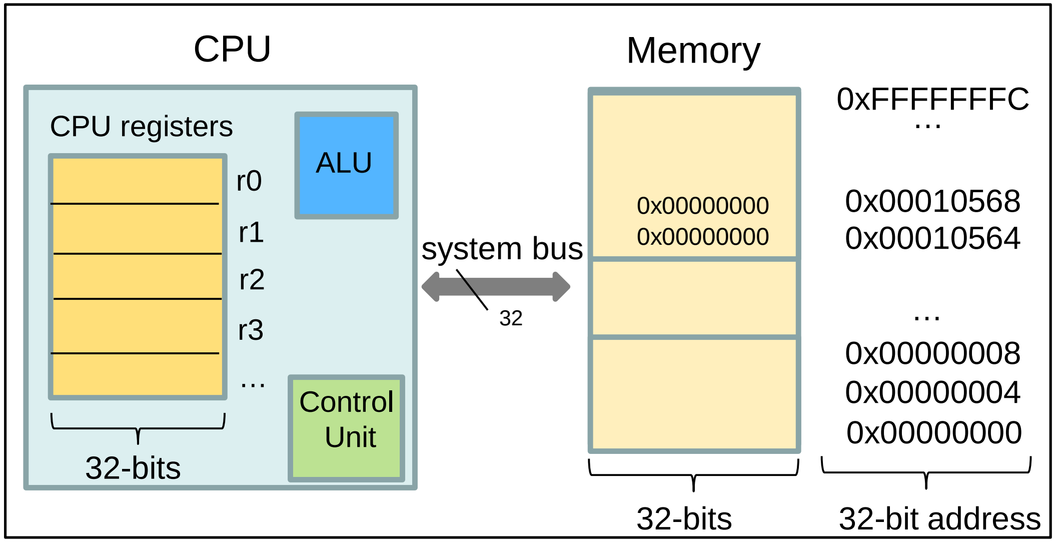
Assembly program 3: adding 2 numbers in memory

- Assembly program 3 shows a simpler way to write assembly code



```
/* -- store02.s */
.data
myvar1:  .word 0
myvar2:  .word 0
.text
.global main
main:
 ldr r1, =myvar1  @ r1 <- &myvar1
 mov r3, #3       @ r3 <- 3
 str r3, [r1]     @ *r1 <- r3
 ldr r2, =myvar2  @ r2 <- &myvar2
 mov r3, #4       @ r3 <- 4
 str r3, [r2]     @ *r2 <- r3
 ldr r1, =myvar1  @ r1 <- &myvar1
 ldr r1, [r1]     @ r1 <- *r1
 ldr r2, =myvar2  @ r2 <- &myvar2
 ldr r2, [r2]     @ r2 <- *r2
 add r0, r1, r2
 bx  lr
```

- We can refer to the address of **myvar1** as =myvar1, then there is no need for extra labels in the .text section to refer to the labels in the .data section

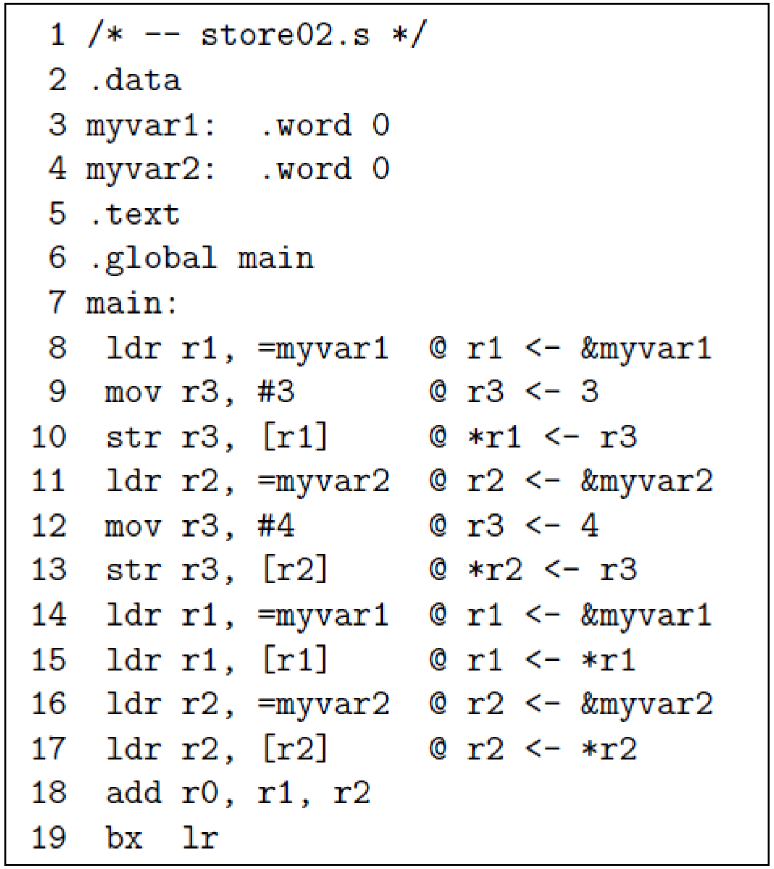


Simplified block diagram of a modern computer

Raspberry Pi Assembler

Assembly program 3: adding 2 numbers in memory

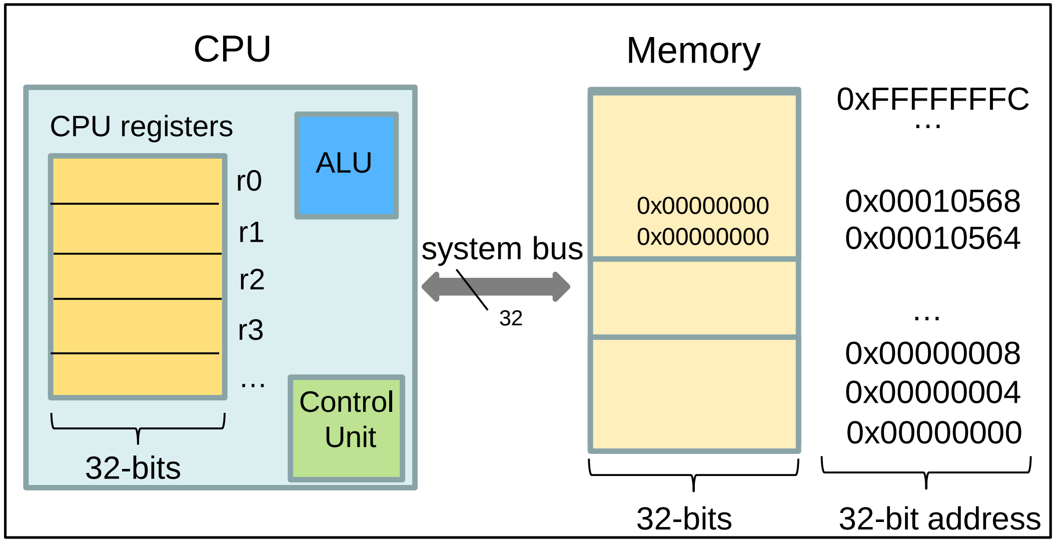
- Assembly program 3 shows a simpler way to write assembly code



```
/* -- store02.s */
.data
myvar1:   .word 0
myvar2:   .word 0
.text
.global main
main:
 ldr r1, =myvar1  @ r1 <- &myvar1
 mov r3, #3       @ r3 <- 3
 str r3, [r1]     @ *r1 <- r3
 ldr r2, =myvar2  @ r2 <- &myvar2
 mov r3, #4       @ r3 <- 4
 str r3, [r2]     @ *r2 <- r3
 ldr r1, =myvar1  @ r1 <- &myvar1
 ldr r1, [r1]     @ r1 <- *r1
 ldr r2, =myvar2  @ r2 <- &myvar2
 ldr r2, [r2]     @ r2 <- *r2
 add r0, r1, r2
 bx  lr
```

- We can refer to the address of **myvar1** as =myvar1, then there is no need for extra labels in the .text section to refer to the labels in the .data section

- Furthermore, if we assume that we only define variables that require 4 bytes of space, then there is no need to use the .balign directive



Simplified block diagram of a modern computer