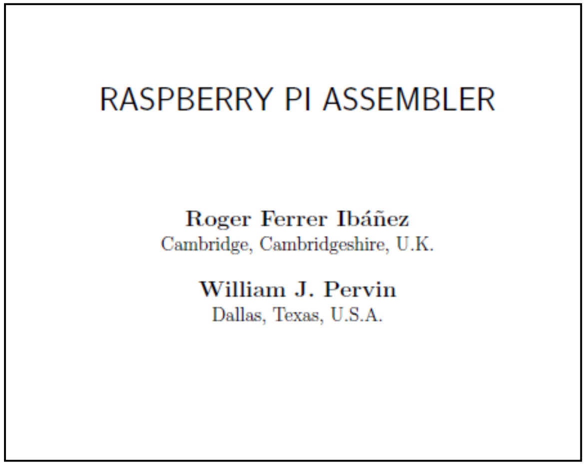
Raspberry Pi Assembler
Debugging



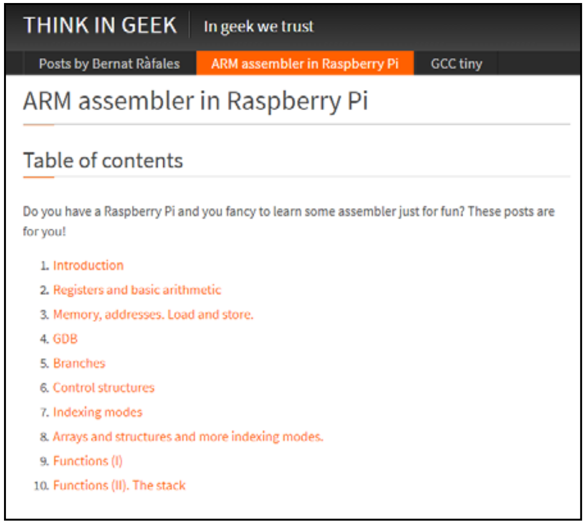
RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 4: Raspberry Pi Assembler
"Raspberry Pi Assembler" by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/12/arm-assembler-raspberry-pi-chapter-4/



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
Debugging

- What is debugging?
  - Debugging refers to the process used to identify errors in your program, since it is rare to write bug-free code all at once.

- Example to explain the concept of debugging
  - You were tasked to develop a program called **SortVector** to sort a sequence of numbers in descending order
  - Let **VectorUnsorted** = [5 3 2 15 8]
  - After developing the program, the sorted vector **VectorSorted** was obtained

    VectorSorted = SortVector(**VectorUnsorted**)

    VectorSorted = [15 8 5 2 3]

Raspberry Pi Assembler
Debugging

- What is debugging?
  - Debugging refers to the process used to identify errors in your program, since it is rare to write bug-free code all at once.

- Example to explain the concept of debugging
  - You were tasked to develop a program called **SortVector** to sort a sequence of numbers in descending order
  - Let **VectorUnsorted** = [5 3 2 15 8]
  - After developing the program, the sorted vector **VectorSorted** was obtained

    VectorSorted = SortVector(**VectorUnsorted**)

    VectorSorted = [15 8 5 2 3] error in the final result

  - Thus, the program needs to be debugged. Thereafter, the error in the program should be corrected.

Raspberry Pi Assembler
Debugging

- What is debugging?
  - Debugging refers to the process used to identify errors in your program, since it is rare to write bug-free code all at once.

- Example to explain the concept of debugging
  - You were tasked to develop a program called **SortVector** to sort a sequence of numbers in descending order
  - Let **VectorUnsorted** = [5 3 2 15 8]
  - After developing the program, the sorted vector **VectorSorted** was obtained

    VectorSorted = SortVector(**VectorUnsorted**)

    VectorSorted = [15 8 5 2 3] error in the final result

  - Thus, the program needs to be debugged. Thereafter, the error in the program should be corrected.
  - What does the debugging process involve? First, we predict the values of variables in the program. Then we step through the code and identify errors by comparing the difference between the predicted and the observed values.

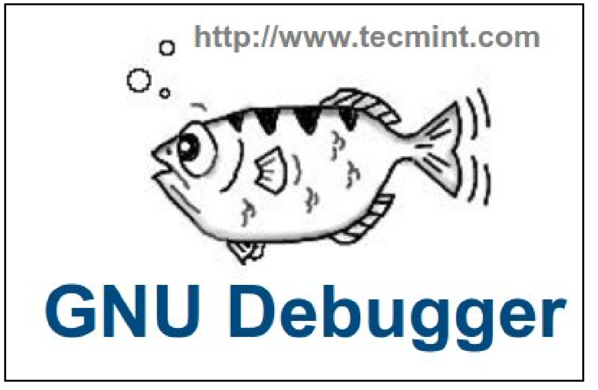
GNU Debugger

Raspberry Pi Assembler
GNU Debugger

- What is GNU Debugger?
  - GNU Debugger (GDB) is an open-source software used to debug other programs

- Basic functionality offered by GDB
  - User can step through the code
  - Print the contents of CPU registers and variables stored in the .data section, to the terminal
  - Disassemble machine instructions, ie. go from machine instructions to assembly code
  - Insert breakpoints in the code
  - … many more

- Let's work with GDB and learn commands
  as we go along



GNU Debugger and store2 program

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3
  - Why are we debugging a program that works correctly?

```
/* -- 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
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3
  - Why are we debugging a program that works correctly?
  - To demonstration how to use GDB. Let's check the contents of CPU registers and variables in memory after certain lines of code have executed

After line 9 has executed

Check contents of r3

After line 10 has executed

- Check address of myvar1 and contents of r1. These two should be the same.
- Check the contents of myvar1

```
1 /* -- store02.s */
2 .data
3 myvar1:   .word 0
4 myvar2:   .word 0
5 .text
6 .global main
7 main:
8  ldr r1, =myvar1  @ r1 <- &myvar1
9  mov r3, #3       @ r3 <- 3
10  str r3, [r1]     @ *r1 <- r3
11  ldr r2, =myvar2  @ r2 <- &myvar2
12  mov r3, #4       @ r3 <- 4
13  str r3, [r2]     @ *r2 <- r3
14  ldr r1, =myvar1  @ r1 <- &myvar1
15  ldr r1, [r1]     @ r1 <- *r1
16  ldr r2, =myvar2  @ r2 <- &myvar2
17  ldr r2, [r2]     @ r2 <- *r2
18  add r0, r1, r2
19  bx  lr
```

Raspberry Pi Assembler
GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug

the argument is the file to be debugged

```
$ gdb --args ./store02
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/> ...
Reading symbols from /home/RPiA/Chapter03/store02 ... done.
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main

```
(gdb) start
Temporary breakpoint 1 at 0x8394 : file store02.s, line 14.
Starting program: /home/RPiA/Chapter03/store02
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed

Type **disassemble** to convert the code from machine instructions to assembly instructions

The next instruction to be executed is denoted by the => symbol

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
=> 0x00008394 <+4>:  mov r3, #3
   0x00008398 <+8>:  str r3, [r1]
   0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>: andeq r0,r1,r4,ror #10
   0x000083C4 <+52>: andeq r0,r1,r8,ror #10
End of assembler dump.
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed

Let’s make a few observations of the assembler code

Address of myvar1 has been assigned to 0x83c0, which is 48 bytes from where **main** begins

The first instruction of **main** can be found at address 0x00008390

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
=> 0x00008394 <+4>:  mov r3, #3
   0x00008398 <+8>:  str r3, [r1]
   0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>: andeq r0,r1,r4,ror #10
   0x000083C4 <+52>: andeq r0,r1,r8,ror #10
End of assembler dump.
```

- The assembler has written code to store the real addresses of myvar1 and myvar2.
- However, the disassembler has simply translated these machine instructions into presumed assembler code

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction

The next instruction to be executed is displayed to the user

```
(gdb) stepi
10 str r3, [r1]    @ *r1 <- r3
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed

The last instruction that was executed was mov r3, #3 because it is the line of code before the => symbol

The next instruction to be executed

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
=> 0x00008398 <+8>:  str r3, [r1]
   0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3

Type **info registers r0 r1 r2 r3** to view the contents of these CPU registers

```
(gdb) info registers r0 r1 r2 r3
r0             0x2          2
r1             0x10564 66916
r2             0xbefff86c      3204445692
r3             0x3          3
```

Confirms that r3 has the value 0x3 in hex or 3 in decimal

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3
7. Type **stepi** to step through the code, ie. only execute the next instruction

If we disassembled the code, it would look like …

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
   0x00008398 <+8>:  str r3, [r1]
=> 0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

last instruction executed

```
(gdb) stepi
11      ldr r2, =myvar2      @ r2 <- &myvar2
```

The next instruction to be executed is displayed to the user

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3
7. Type **stepi** to step through the code, ie. only execute the next instruction
8. Check that r1 has the address of myvar1
    - View contents of r1
    - View address of myvar1

If we disassembled the code, it would look like …

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
   0x00008398 <+8>:  str r3, [r1]
=> 0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

last instruction executed

```
(gdb) info register r1
r1            0x10564    66919
```

contents of r1 is 0x10564 in hex and 66919 in decimal

Great, it has changed. In fact this is the address of `myvar1`. Let's check that using its symbolic name and C syntax.

```
(gdb) p &myvar1
$3 = (<data variable, no debug info>  *) 0x10564
```

That again agrees with our expectations! In addition we can see what is in that variable:

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let’s debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3
7. Type **stepi** to step through the code, ie. only execute the next instruction
8. Check that r1 has the address of myvar1
    - View contents of r1
    - View address of myvar1

If we disassembled the code, it would look like …

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
   0x00008398 <+8>:  str r3, [r1]
=> 0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

last instruction executed

```
(gdb) info register r1
r1             0x10564    66919
```

contents of r1 is 0x10564 in hex and 66919 in decimal

Great, it has changed. In fact this is the address of myvar1. Let’s check that using its symbolic name and C syntax.

Type p to print to terminal

& denotes address

```
(gdb) p &myvar1
$3 = (<data variable, no debug info> *) 0x10564
```

address of myvar1 is 0x10564 in hex

Observation: counter for the printed result

That again agrees with our expectations! In addition we can see what is in that variable:

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3
7. Type **stepi** to step through the code, ie. only execute the next instruction
8. Check that r1 has the address of myvar1
   - View contents of r1
   - View address of myvar1

∴ **r1 is equal to the address of myvar1**

If we disassembled the code, it would look like …

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
   0x00008398 <+8>:  str r3, [r1]
=> 0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

last instruction executed

```
(gdb) info register r1
r1            0x10564    66919
```

contents of r1 is 0x10564 in hex and 66919 in decimal

Great, it has changed. In fact this is the address of myvar1. Let's check that using its symbolic name and C syntax.

Type p to print to terminal

& denotes address

```
(gdb) p &myvar1
$3 = (<data variable, no debug info> *) 0x10564
```

Observation: counter for the printed result

address of myvar1 is 0x10564 in hex

That again agrees with our expectations! In addition we can see what is in that variable:

Raspberry Pi Assembler

GNU Debugger and the store02 program

- Let's debug the **store02** program developed at the end of Chapter 3

1. Start GDB and specify the program to debug
2. Type **start** to run the program up to and including the first instruction of main
3. Disassemble the machine instructions into human readable assembly instructions and to observe the next instruction to be executed
4. Type **stepi** to step through the code, ie. only execute the next instruction
5. Disassemble to confirm the last instruction that was executed
6. View the contents of registers r0, r1, r2, r3 to confirm that r3 was changed to the value of 3
7. Type **stepi** to step through the code, ie. only execute the next instruction
8. Check that r1 has the address of myvar1
    - View contents of r1
    - View address of myvar1
9. Lastly, check that the variable myvar1 has been assigned the value of r3

If we disassembled the code, it would look like …

```
(gdb) disassemble
Dump of assembler code for function main:
   0x00008390 <+0>:  ldr r1, [pc, #40] ; 0x83c0 <main+48>
   0x00008394 <+4>:  mov r3. #3
   0x00008398 <+8>:  str r3, [r1]
=> 0x0000839c <+12>: ldr r2, [pc, #32] ; 0x83c4 <main+52>
   0x000083a0 <+16>: mov r3, #4
   0x000083a4 <+20>: str r3, [r2]
   0x000083a8 <+24>: ldr r1, [pc, #16] ; 0x83c0 <main+48>
   0x000083ac <+28>: ldr r1, [r1]
   0x000083b0 <+32>: ldr r2, [pc, #12] ; 0x83c4 <main+52>
   0x000083b4 <+36>: ldr r2, [r2]
   0x000083b8 <+40>: add r0, r1, r2
   0x000083bc <+44>: bx lr
   0x000083c0 <+48>:    andeq r0,r1,r4,ror #10
   0x000083c4 <+52>:    andeq r0,r1,r8,ror #10
End of assembler dump.
```

last instruction executed

```
(gdb) p myvar1
$4 = 3
```

Observation: counter for the printed result

confirms that the contents of the variable myvar1 is 3, which is equal to register r3

GNU Debugger: commands

Raspberry Pi Assembler

GNU Debugger: commands

GDB cheatsheet - page 1

Running

 # gdb <program> [core dump]
Start GDB (with optional core dump).

 # gdb --args <program> <args…>
Start GDB and pass arguments

 # gdb --pid <pid>
Start GDB and attach to process.

set args <args...>
Set arguments to pass to program to be debugged.

run
Run the program to be debugged.

kill
Kill the running program.

Breakpoints

break <where>
Set a new breakpoint.

delete <breakpoint#>
Remove a breakpoint.

clear
Delete all breakpoints.

enable <breakpoint#>
Enable a disabled breakpoint.

disable <breakpoint#>
Disable a breakpoint.

Watchpoints

watch <where>
Set a new watchpoint.

delete/enable/disable <watchpoint#>
Like breakpoints.

<where>

function_name
Break/watch the named function.

line_number
Break/watch the line number in the current source file.

file:line_number
Break/watch the line number in the named source file.

Conditions

break/watch <where> if <condition>
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

condition <breakpoint#> <condition>
Set/change the condition of an existing break- or watchpoint.

Examining the stack

backtrace
where
Show call stack.

backtrace full
where full
Show call stack, also print the local variables in each frame.

frame <frame#>
Select the stack frame to operate on.

Stepping

step
Go to next instruction (source line), diving into function.

next
Go to next instruction (source line) but don't dive into functions.

finish
Continue until the current function returns.

continue
Continue normal execution.

Variables and memory

print/format <what>
Print content of variable/memory location/register.

display/format <what>
Like „print", but print the information after each stepping instruction.

undisplay <display#>
Remove the „display" with the given number.

enable display <display#>
disable display <display#>
En- or disable the „display" with the given number.

x/nfu <address>
Print memory.
n: How many units to print (default 1).
f: Format character (like „print").
u: Unit.

Unit is one of:

- *b*: Byte,
- *h*: Half-word (two bytes)
- *w*: Word (four bytes)
- *g*: Giant word (eight bytes)).

Raspberry Pi Assembler
GNU Debugger: commands

GDB cheatsheet - page 2

Format

a	Pointer.
c	Read as integer, print as character.
d	Integer, signed decimal.
f	Floating point number.
o	Integer, print as octal.
s	Try to treat as C string.
t	Integer, print as binary (*t* = „two“).
u	Integer, unsigned decimal.
x	Integer, print as hexadecimal.

<what>

`expression`
: Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

`file_name::variable_name`
: Content of the variable defined in the named file (static variables).

`function::variable_name`
: Content of the variable defined in the named function (if on the stack).

`{type}address`
: Content at *address*, interpreted as being of the C type *type*.

`$register`
: Content of named register. Interesting registers are $esp (stack pointer), $ebp (frame pointer) and $eip (instruction pointer).

Threads

`thread <thread#>`
: Chose thread to operate on.

Manipulating the program

`set var <variable_name>=<value>`
: Change the content of a variable to the given value.

`return <expression>`
: Force the current function to return immediately, passing the given value.

Sources

`directory <directory>`
: Add *directory* to the list of directories that is searched for sources.

`list`
`list <filename>:<function>`
`list <filename>:<line_number>`
`list <first>,<last>`
: Shows the current or given source context. The *filename* may be omitted. If *last* is omitted the context starting at *start* is printed instead of centered around it.

`set listsize <count>`
: Set how many lines to show in „list“.

Signals

`handle <signal> <options>`
: Set how to handle signles. Options are:

(no)print: (Don't) print a message when signals occurs.

(no)stop: (Don't) stop the program when signals occurs.

(no)pass: (Don't) pass the signal to the program.

Informations

`disassemble`
`disassemble <where>`
: Disassemble the current function or given location.

`info args`
: Print the arguments to the function of the current stack frame.

`info breakpoints`
: Print informations about the break- and watchpoints.

`info display`
: Print informations about the „displays“.

`info locals`
: Print the local variables in the currently selected stack frame.

`info sharedlibrary`
: List loaded shared libraries.

`info signals`
: List all signals and how they are currently handled.

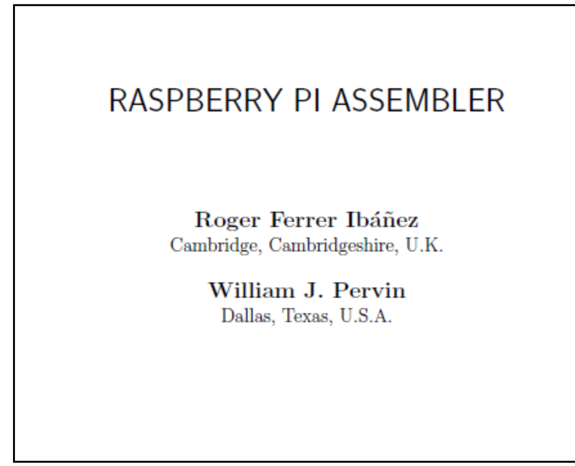
`info threads`
: List all threads.

`show directories`
: Print all directories in which GDB searches for source files.

`show listsize`
: Print how many are shown in the „list“ command.

`whatis variable_name`
: Print type of named variable.

Raspberry Pi Assembler
Branching



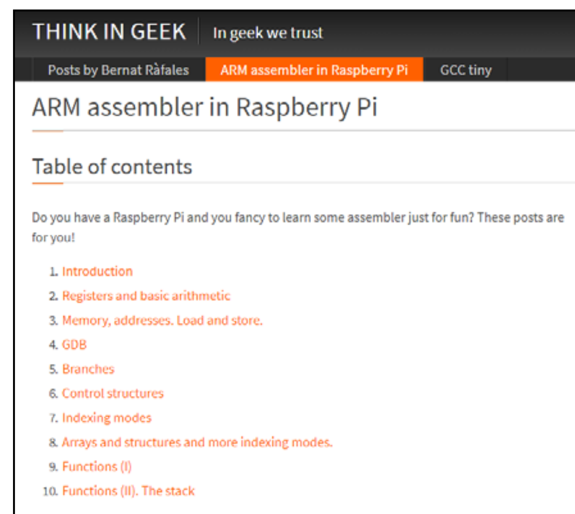
RASPBERRY PI ASSEMBLER

Roger Ferrer Ibáñez
Cambridge, Cambridgeshire, U.K.

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

Chapter 5: Raspberry Pi Assembler
“Raspberry Pi Assembler” by R. Ferrer and W. Pervin

https://thinkingeek.com/2013/01/19/arm-assembler-raspberry-pi-chapter-5/



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

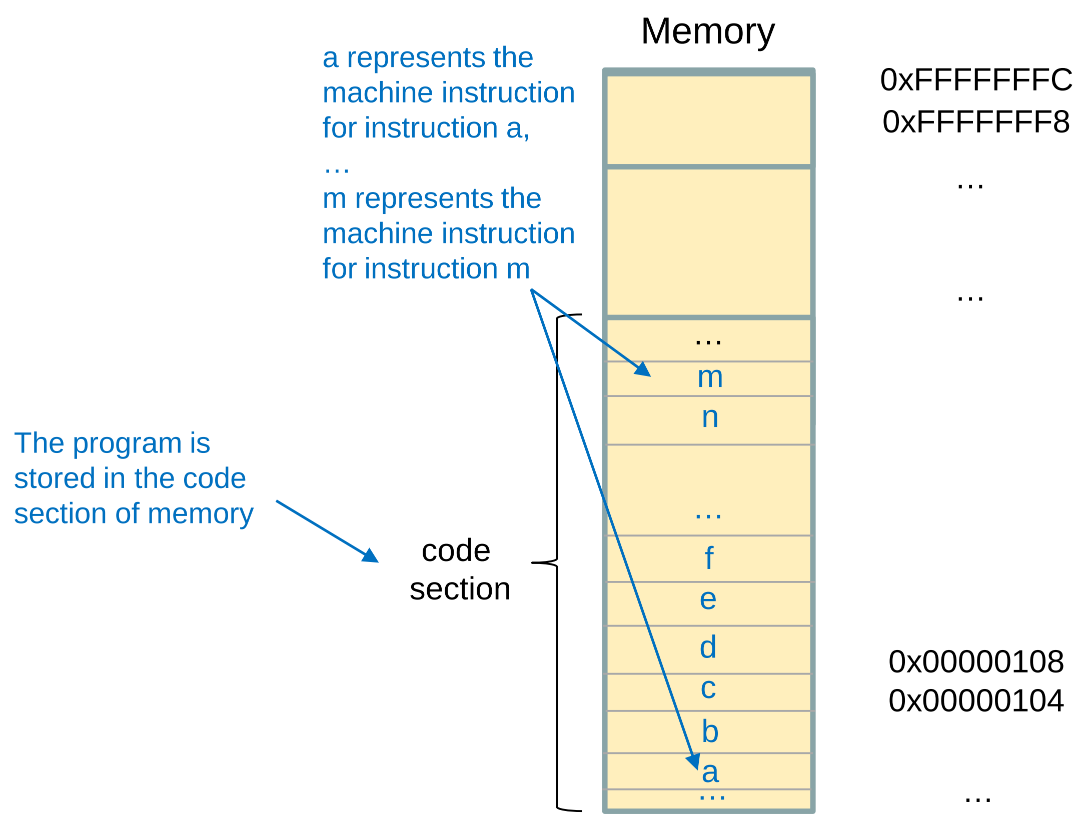
Embedded Systems II – EEE3096S
Y. Abdul Gaffar
S. Winberg

Raspberry Pi Assembler

Introduction: programs

- Consider a program with many operations.
  - The first instruction is denoted by ‘instruction a’. The second instruction is denoted by ‘instruction b’, …
  - Furthermore, let the letter ‘a’ denote the 32-bit machine instruction related to instruction a, ‘b’ denote the machine instruction related to instruction b, …

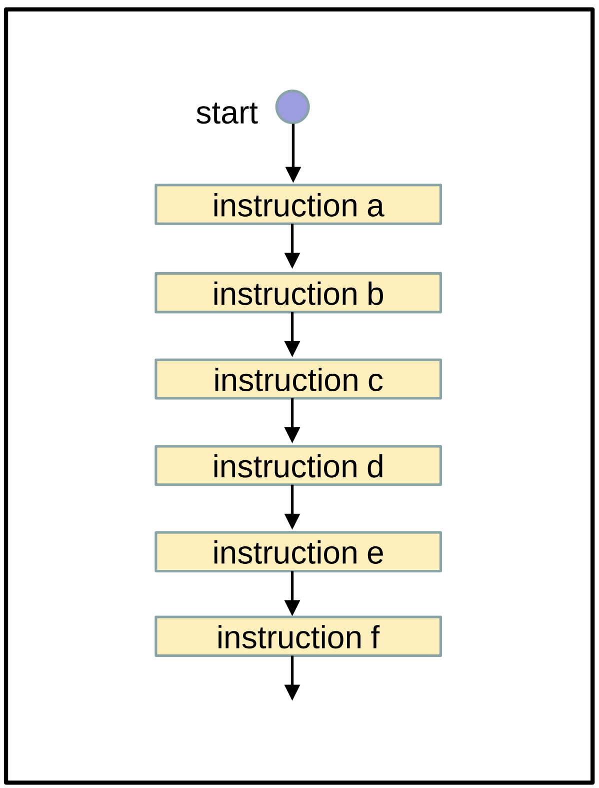
```
*/ Example program */

Instruction a
Instruction b
Instruction c
….

Instruction n
Instruction m
…
```



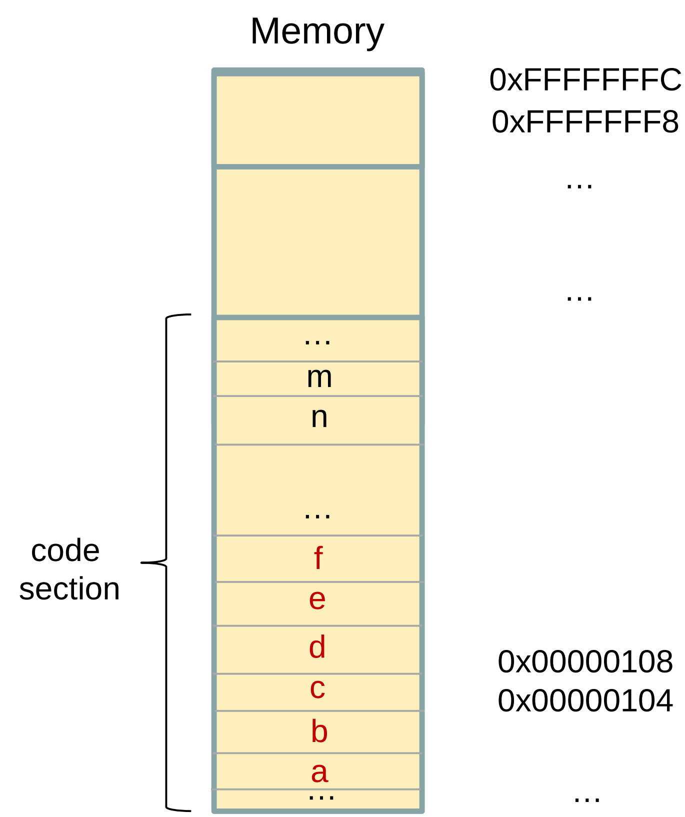
Raspberry Pi Assembler

Programs without branches

- Typically, in a simple program, after the next instruction is fetched from memory, the PC (R15) is incremented by 4. Thus, instructions are processed sequentially.



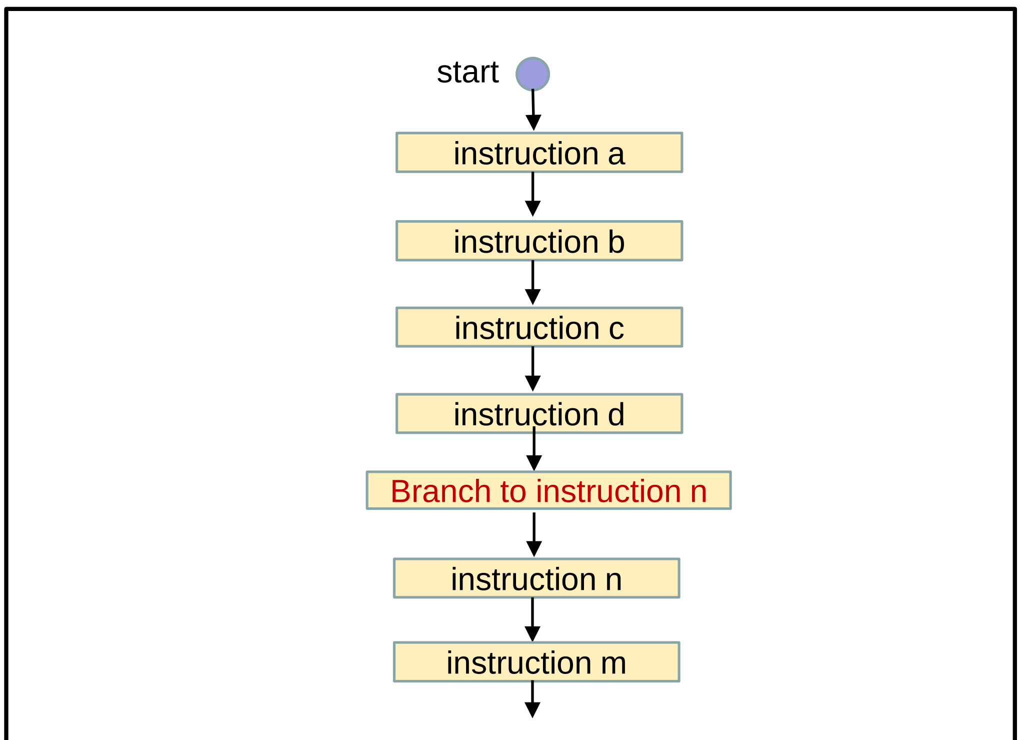
Flow chart of a simple program without branching



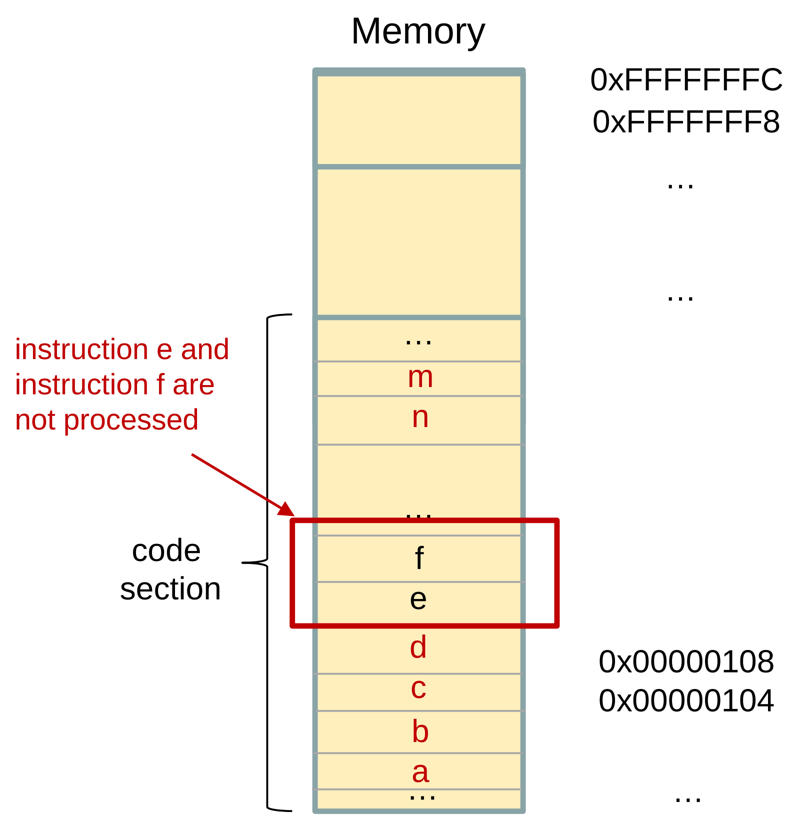
Raspberry Pi Assembler

Programs with branches

- What is branching?
  - Branching refers to changing the sequential processing of instructions
  - When an instruction modifies the value of the Program Counter (PC), then the new PC is used to address the next instruction. In this way, branching is implemented. There is unconditional branching



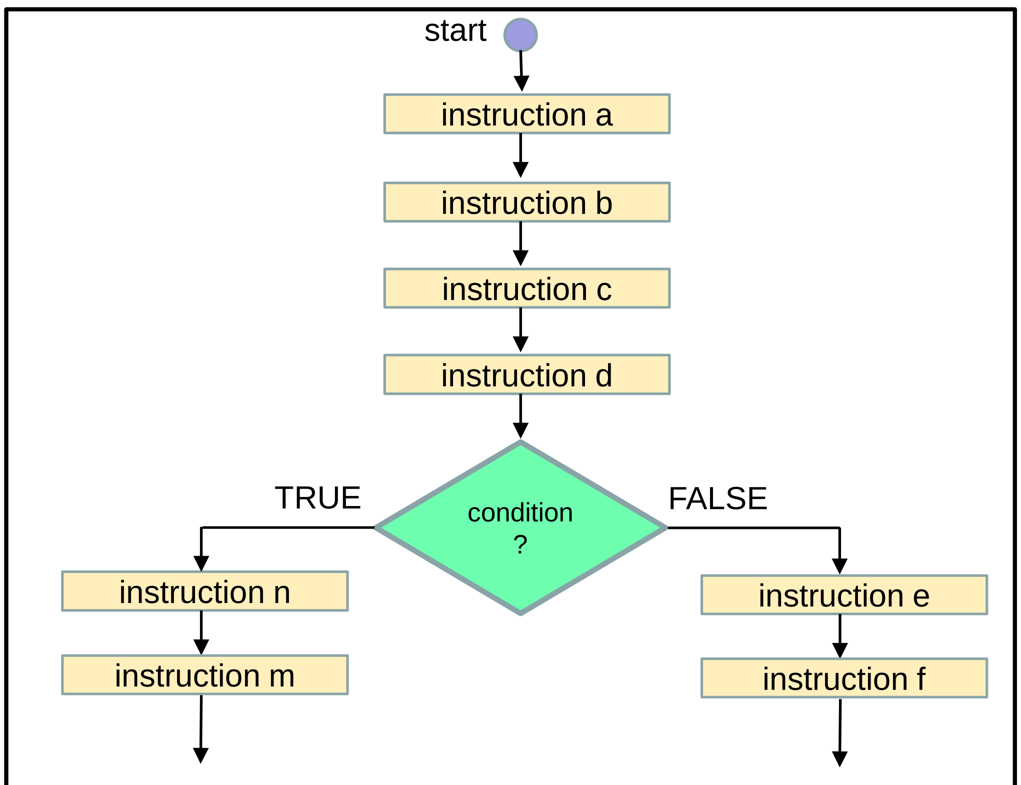
Flow chart of a program with unconditional branching



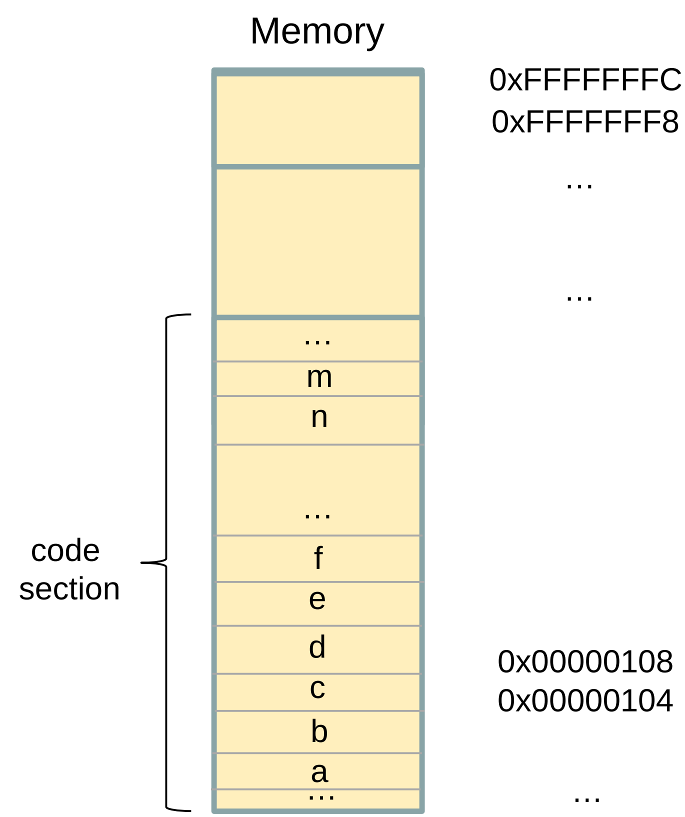
Raspberry Pi Assembler

Programs with branches

- What is branching?
  - Branching refers to changing the sequential processing of instructions
  - When an instruction modifies the value of the Program Counter (PC), then the new PC is used to address the next instruction. In this way, branching is implemented. There is unconditional branching and conditional branching



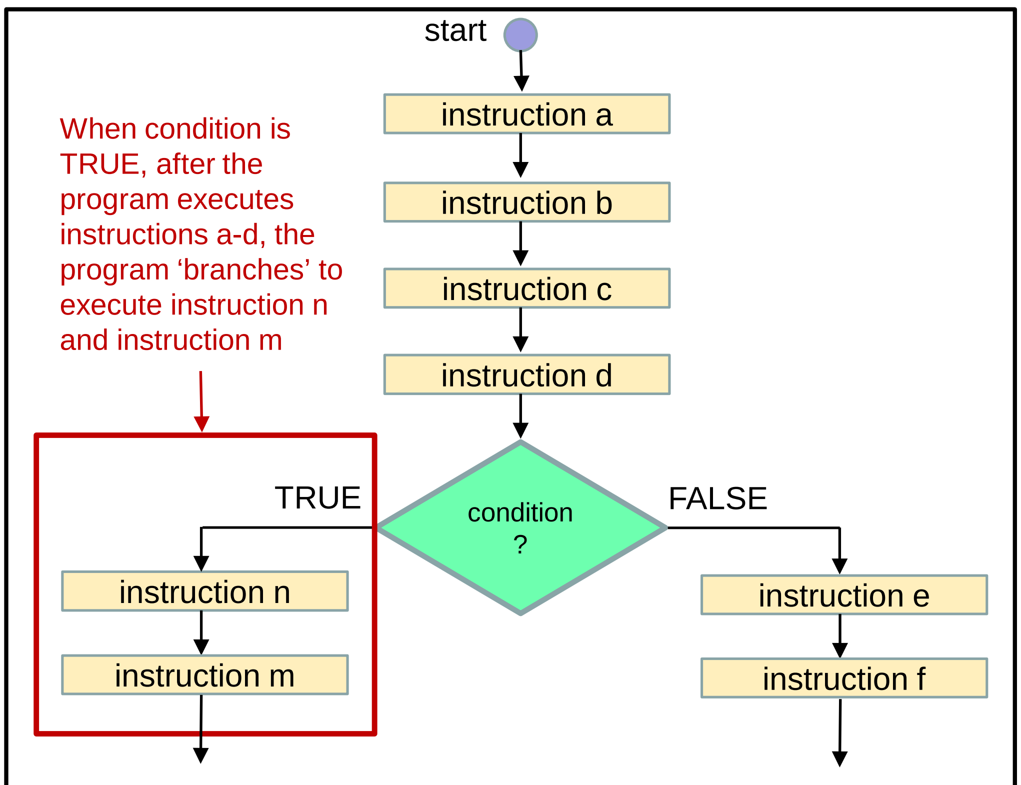
Flow chart of a program with conditional branching



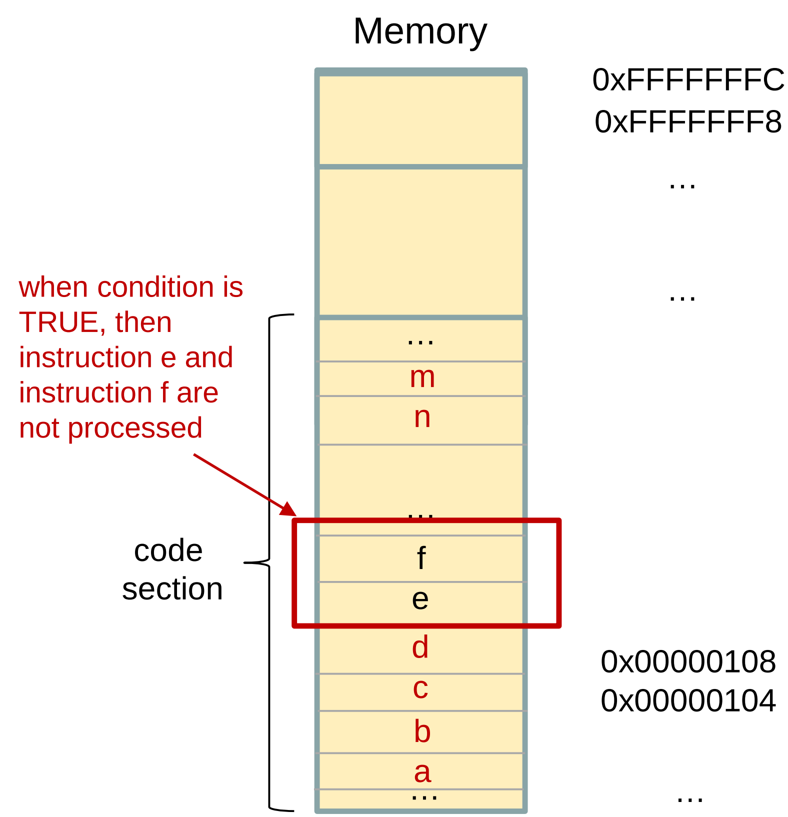
Raspberry Pi Assembler

Programs with branches

- What is branching?
  - Branching refers to changing the sequential processing of instructions
  - When an instruction modifies the value of the Program Counter (PC), then the new PC is used to address the next instruction. In this way, branching is implemented. There is unconditional branching and conditional branching



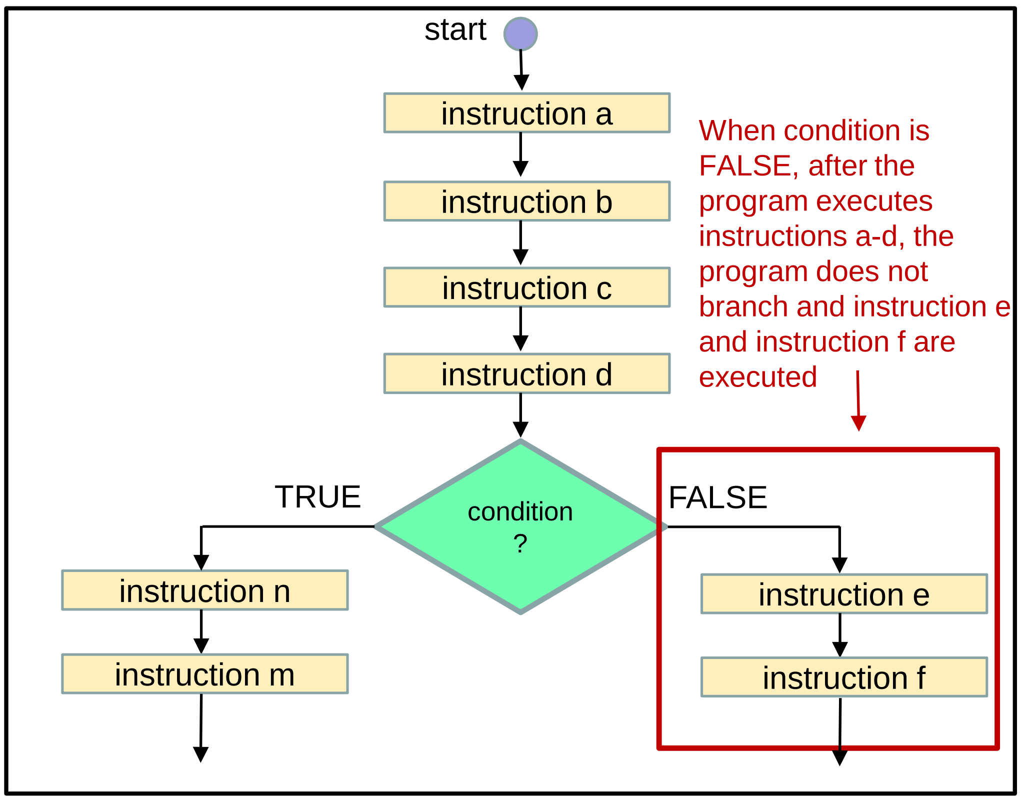
Flow chart of a program with conditional branching



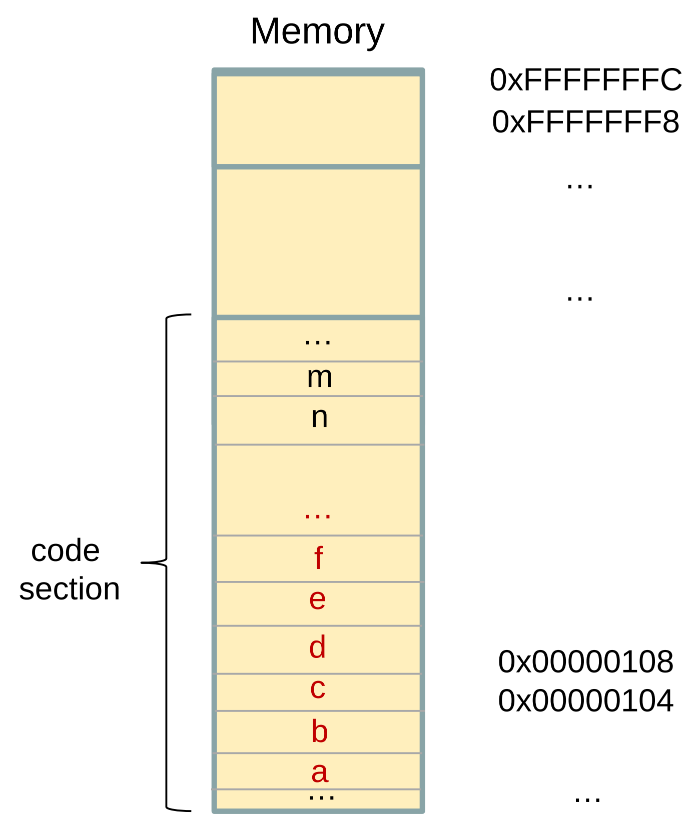
Raspberry Pi Assembler

Programs with branches

- What is branching?
  - Branching refers to changing the sequential processing of instructions
  - When an instruction modifies the value of the Program Counter (PC), then the new PC is used to address the next instruction. In this way, branching is implemented. There is unconditional branching and conditional branching



Flow chart of a program with conditional branching



Unconditional branching

Raspberry Pi Assembler
Unconditional branches

- Let’s look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2   @ r0 <- 2
    b   end      @ branch to 'end'
    mov r0, #3   @ r0 <- 3
end:
    bx  lr
```

Raspberry Pi Assembler
Unconditional branches

- Let's look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

Observations

```
/* -- branch01.s */
.text
.global main
main:
     mov r0, #2     @ r0 <- 2
     b   end        @ branch to 'end'
     mov r0, #3     @ r0 <- 3
end:
     bx  lr
```

Labels represent the address of the machine instructions in memory

Raspberry Pi Assembler
Unconditional branches

- Let's look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

Stepping through code to understand the program flow

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2    @ r0 <- 2
    b   end       @ branch to 'end'
    mov r0, #3    @ r0 <- 3
end:
    bx  lr
```

After this instruction has executed, the register r0 will have the value 2

Raspberry Pi Assembler
Unconditional branches

- Let's look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

Stepping through code to understand the program flow

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2    @ r0 <- 2
    b   end       @ branch to 'end'
    mov r0, #3    @ r0 <- 3
end:
    bx  lr
```

After this instruction has executed, the Program Counter (PC) will be loaded with the value of the memory address of label **end**

Raspberry Pi Assembler

Unconditional branches

- Let's look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

Stepping through code to understand the program flow

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2    @ r0 <- 2
    b   end       @ branch to 'end'
    mov r0, #3    @ r0 <- 3
end:
    bx  lr
```

program branches to the **end** label and program terminates

Raspberry Pi Assembler

Unconditional branches

- Let's look at an assembly program that uses the branch instruction
  - The unconditional branch command is denoted by **b**

```
/* -- branch01.s */
.text
.global main
main:
    mov r0, #2    @ r0 <- 2
    b   end       @ branch to 'end'
    mov r0, #3    @ r0 <- 3
end:
    bx  lr
```

If you execute this program you will see that it returns an error code of 2.

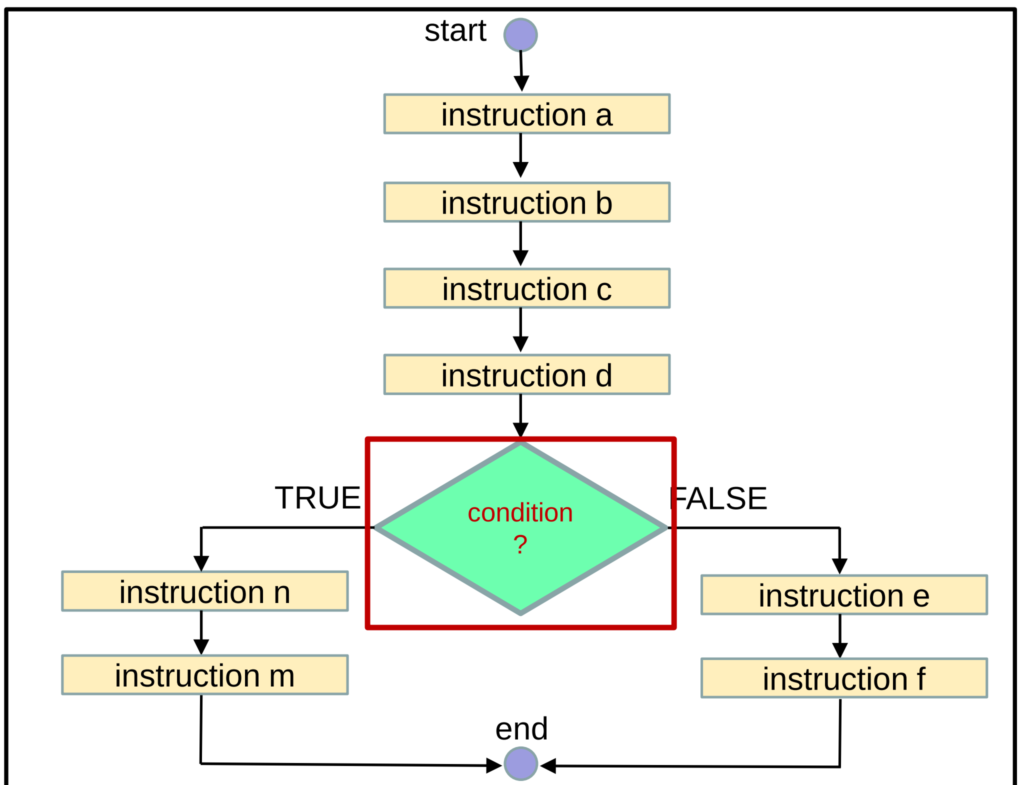
```
$ ./branch01 ; echo $?
2
```

Conditional branching

Raspberry Pi Assembler

Conditional branches

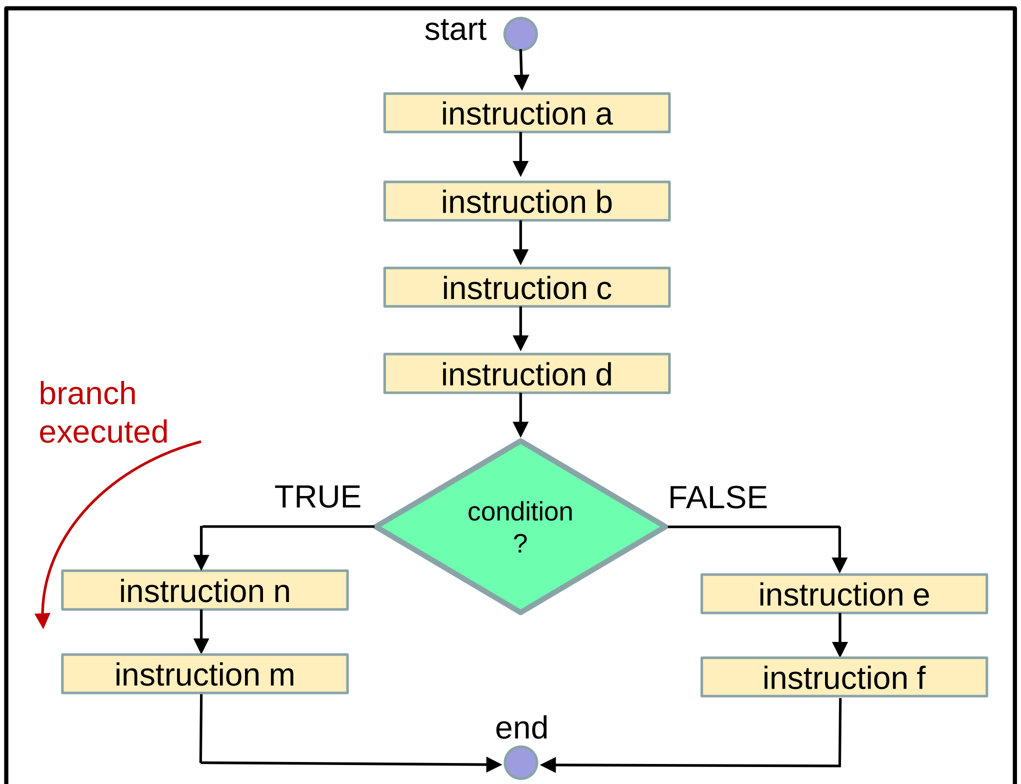
- Conditional branching involves two steps
  1. Evaluate a condition



Flow chart of a program with conditional branching

Raspberry Pi Assembler
Conditional branches

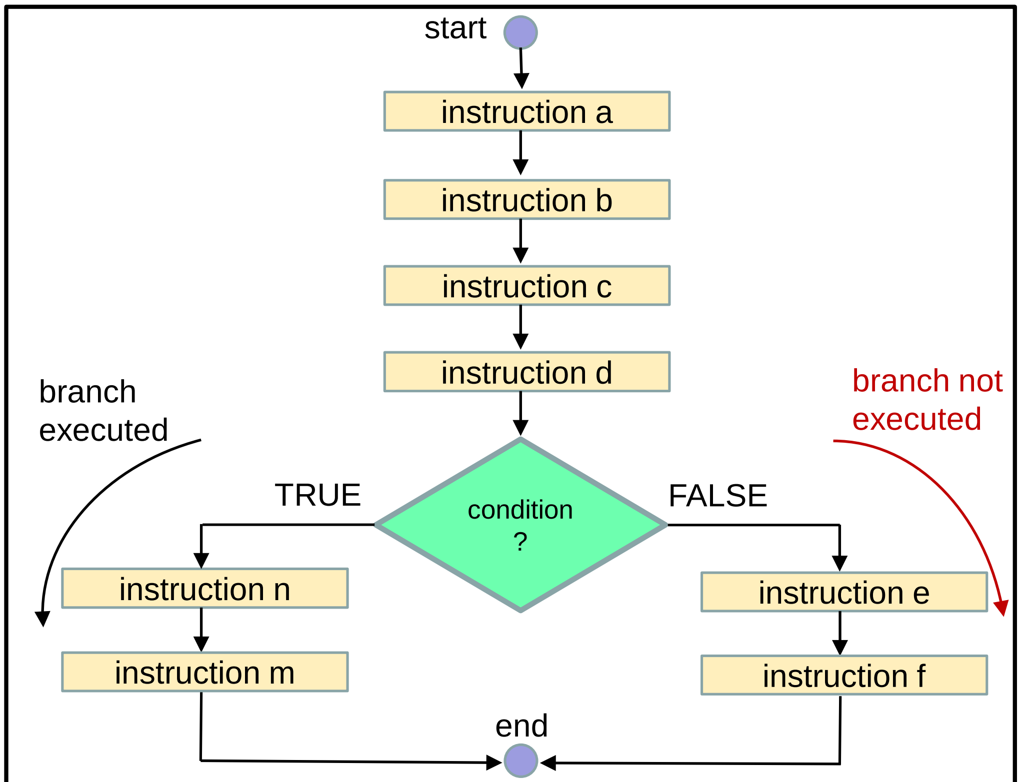
- Conditional branching involves two steps
  1. Evaluate a condition
  2. If the condition is TRUE, then execute the branch instruction



Flow chart of a program with conditional branching

Raspberry Pi Assembler
Conditional branches

- Conditional branching involves two steps
  1. Evaluate a condition
  2. If the condition is TRUE, then execute the branch instruction. If the condition is FALSE, then do not execute the branch instruction

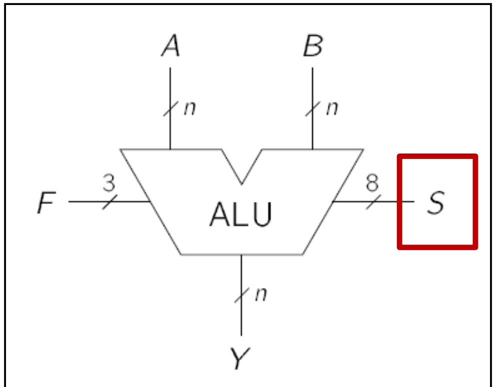


Flow chart of a program with conditional branching

Raspberry Pi Assembler
Conditional branches: condition evaluated

- What condition is evaluated?
  - Typically, one or more of the four flags in the Current Program Status Register (CPSR) is evaluated when a conditional branch instruction is used

- What is the CPSR?
  - The CPSR is a 32-bit register that is found in the CPU
  - This is similar to the status bits **S** from the output of a generic ALU



A n-bit Arithmetic Logic Unit.

Raspberry Pi Assembler

Conditional branches: condition evaluated

- Example: the ARM CPSR
  - There are four condition code flags which may be updated after the previously executed instruction

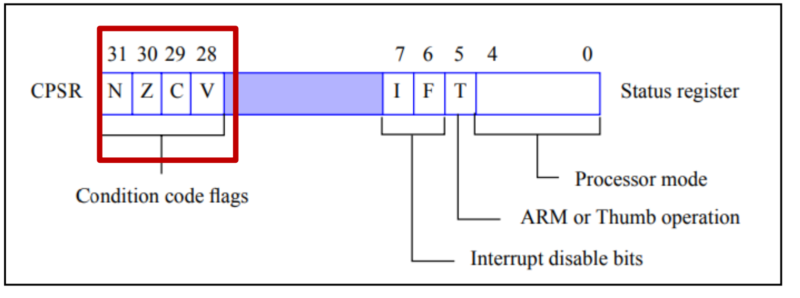
Assumes the operands and the result are both **unsigned** numbers

- Carry (C) : set to 1:
  - If the last operation was addition and there was a unsigned overflow
  - If the last operation was subtraction and a borrow was not needed

  In all other cases, it is cleared to zero
- Zero (Z) : is set to 1 when the last result was zero. Otherwise, it is cleared to 0

Assumes the operands and the result are both **signed** numbers

- Negative (N) : is set to 1 when the last result was negative, ie. the MSB is set to 1. Otherwise, it is cleared to 0.
- Overflow (V) : is set to 1 when the last operation had a signed overflow. Otherwise, it is cleared to 0



The 32-bit ARM CPSR register

Raspberry Pi Assembler

Conditional branches: condition evaluated

- Examples of instructions that update the CPSR
  - add**s** r1, r1, r2
  - sub**s** r2, r1, r0
  - cmp r2, r1
  - …

Raspberry Pi Assembler

Conditional branches: condition evaluated

- Examples of instructions that update the CPSR
    - add**s** r1, r1, r2
    - sub**s** r2, r1, r0
    - cmp r2, r1
    - …

The **s** suffix is used to denote that the CPSR will update after this instruction has executed

Raspberry Pi Assembler
Conditional branches: condition evaluated

- How is branching done based on the four condition code flags?
  - There are many branch instructions that can be used
  - Example: BEQ LABEL
    - The BEQ instruction causes a branch to the location LABEL if the conditional code flag Z is equal to one, when this branch instruction was executed

Raspberry Pi Assembler

Conditional branches: condition evaluated

- How is branching done based on the four condition code flags?
  - There are many branch instructions that can be used
  - Example: BEQ LABEL
    - The BEQ instruction causes a branch to the location LABEL if the conditional code flag Z is equal to one, when this branch instruction was executed

- Examples of branch suffixes and the related conditional code flag that are evaluated
  - Note: place the letter b in front of the suffix to create the full branch instruction.
    Example: NE becomes BNE
    EQ becomes BEQ

Condition field encoding in ARM instructions

Condition suffix	**Condition name**	**Condition Code test**
EQ	Equal (zero)	$Z = 1$
NE	Not equal (nonzero)	$Z = 0$
CS/HS	Carry set/Unsigned higher or same	$C = 1$
CC/LO	Carry clear/Unsigned lower	$C = 0$
MI	Minus (negative)	$N = 1$
PL	Plus (positive or zero)	$N = 0$
VS	Overflow	$V = 1$
VC	No overflow	$V = 0$
HI	Unsigned higher	$\overline{C} \vee Z = 0$
LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
GE	Signed greater than or equal	$N \oplus V = 0$
LT	Signed less than	$N \oplus V = 1$
GT	Signed greater than	$Z \vee (N \oplus V) = 0$
LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
AL	Always	
	not used	

Raspberry Pi Assembler

Conditional branches: example program

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2       @ r1 <- 2
    mov r2, #2       @ r2 <- 2
    cmp r1, r2       @ update cpsr condition codes with r1-r2
    beq case_equal   @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2       @ r0 <- 2
    b   end          @ branch to end
case_equal:
    mov r0, #1       @ r0 <- 1
end:
    bx  lr
```

Raspberry Pi Assembler

Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq)

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2        @ r1 <- 2
    mov r2, #2        @ r2 <- 2
    cmp r1, r2        @ update cpsr condition codes with r1-r2
    beq case_equal    @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2        @ r0 <- 2
    b   end           @ branch to end
case_equal:
    mov r0, #1        @ r0 <- 1
end:
    bx  lr
```

This instruction performs the subtraction: r1 - r2 and then updates the CPSR

Raspberry Pi Assembler
Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq). After this instruction has executed, the Z flag of the CPSR register is set to 1, since (r1 – r2) = 0 and the value of the last result was zero

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2       @ r1 <- 2
    mov r2, #2       @ r2 <- 2
    cmp r1, r2       @ update cpsr condition codes with r1-r2
    beq case_equal   @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2       @ r0 <- 2
    b   end          @ branch to end
case_equal:
    mov r0, #1       @ r0 <- 1
end:
    bx  lr
```

This instruction performs the subtraction: r1 - r2 and then updates the CPSR

Raspberry Pi Assembler

Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq). After this instruction has executed, the Z flag of the CPSR register is set to 1, since (r1 – r2) = 0 and the value of the last result was zero
  - When the branch instruction (BEQ) is executed, the Z flag of the CPSR is evaluated and the program will branch to the label ‘case_equal’, since Z has a value of 1

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2        @ r1 <- 2
    mov r2, #2        @ r2 <- 2
    cmp r1, r2        @ update cpsr condition codes with r1-r2
    beq case_equal    @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2        @ r0 <- 2
    b   end           @ branch to end
case_equal:
    mov r0, #1        @ r0 <- 1
end:
    bx  lr
```

Raspberry Pi Assembler
Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq). After this instruction has executed, the Z flag of the CPSR register is set to 1, since (r1 – r2) = 0 and the value of the last result was zero
  - When the branch instruction (BEQ) is executed, the Z flag of the CPSR is evaluated and the program will branch to the label ‘case_equal’, since Z has a value of 1

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2        @ r1 <- 2
    mov r2, #2        @ r2 <- 2
    cmp r1, r2        @ update cpsr condition codes with r1-r2
    beq case_equal    @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2        @ r0 <- 2
    b   end           @ branch to end
case_equal:
    mov r0, #1        @ r0 <- 1
end:
    bx  lr
```

Raspberry Pi Assembler

Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq). After this instruction has executed, the Z flag of the CPSR register is set to 1, since (r1 – r2) = 0 and the value of the last result was zero
  - When the branch instruction (BEQ) is executed, the Z flag of the CPSR is evaluated and the program will branch to the label ‘case_equal’, since Z has a value of 1
  - When the program completes, r0 = 1 and the value 1 is displayed to the terminal

```
/* -- compare01.s */
.text
.global main
main:
    mov r1, #2        @ r1 <- 2
    mov r2, #2        @ r2 <- 2
    cmp r1, r2        @ update cpsr condition codes with r1-r2
    beq case_equal    @ branch to case_equal only if Z = 1
case_different:
    mov r0, #2        @ r0 <- 2
    b   end           @ branch to end
case_equal:
    mov r0, #1        @ r0 <- 1
end:
    bx  lr
```

Raspberry Pi Assembler

Conditional branches: example program

- In the code below
  - The CoMPare (cmp) instruction is executed before the branch instruction (beq). After this instruction has executed, the Z flag of the CPSR register is set to 1, since (r1 – r2) = 0 and the value of the last result was zero
  - When the branch instruction (BEQ) is executed, the Z flag of the CPSR is evaluated and the program will branch to the label ‘case_equal’, since Z has a value of 1
  - When the program completes, r0 = 1 and the value 1 is displayed to the terminal

```
 1 /* -- compare01.s */
 2 .text
 3 .global main
 4 main:
 5     mov r1, #2        @ r1 <- 2
 6     mov r2, #2        @ r2 <- 2
 7     cmp r1, r2        @ update cpsr condition codes with r1-r2
 8     beq case_equal    @ branch to case_equal only if Z = 1
 9 case_different:
10     mov r0, #2        @ r0 <- 2
11     b   end           @ branch to end
12 case_equal:
13     mov r0, #1        @ r0 <- 1
14 end:
15     bx  lr
```

Change line 5 of the code to mov r1, #3

In this case, the branch will not take place and when the program completes, r0 = 2