# 11 Raspberry Pi Assembly Language

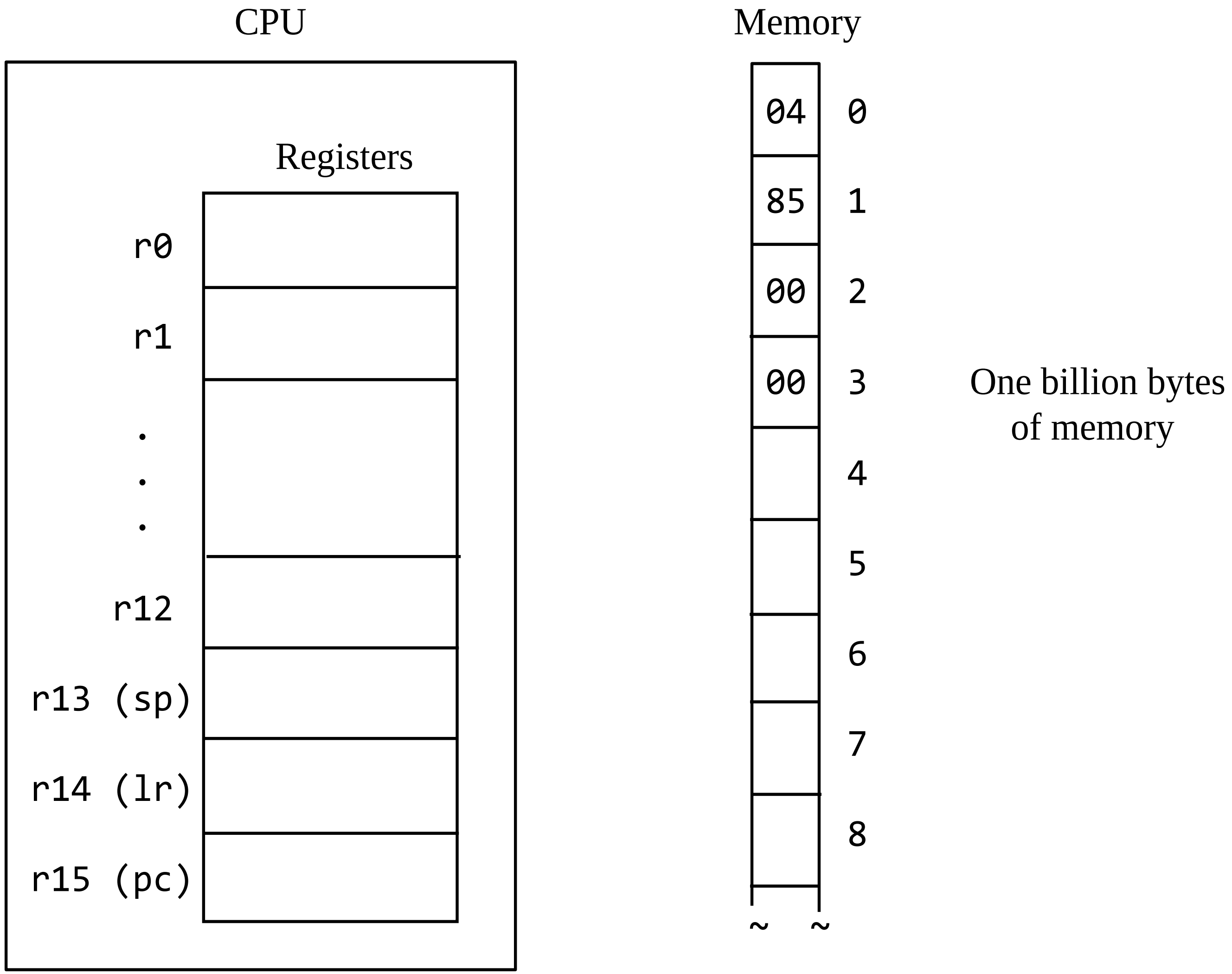
## Introduction

Assembly language          Machine language

```
mov r7, #1          11100011101000000111000000000001
```

Figure 11.1

# Architecture of the Raspberry Pi (A Simplified View)

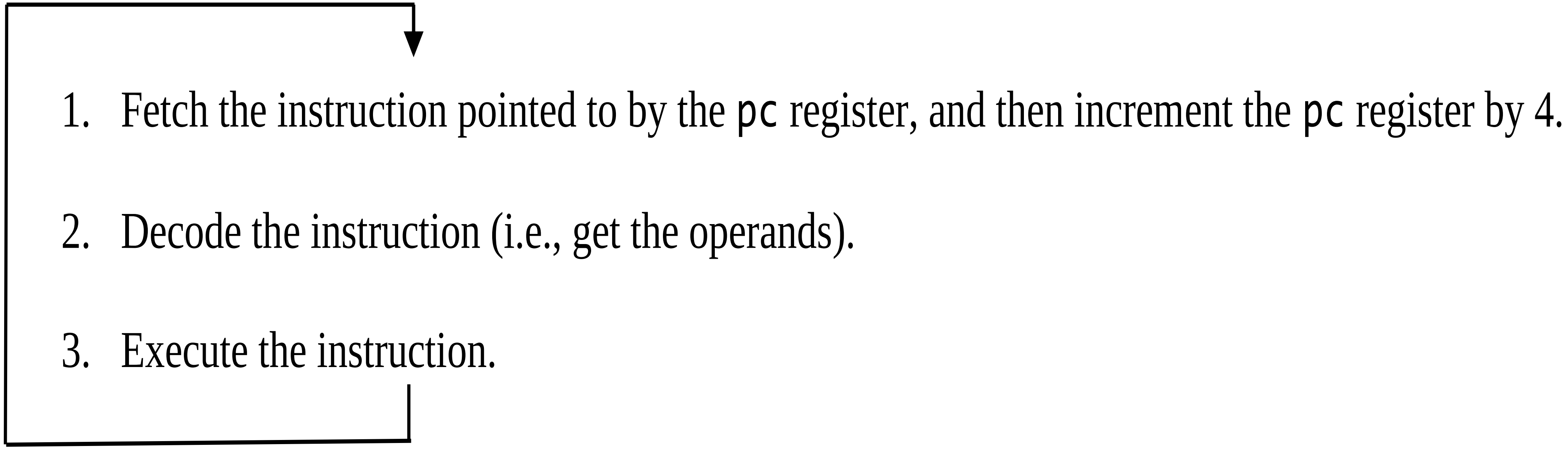


```
add r0, r1, r2
```

destination register

# How Instructions are Executed

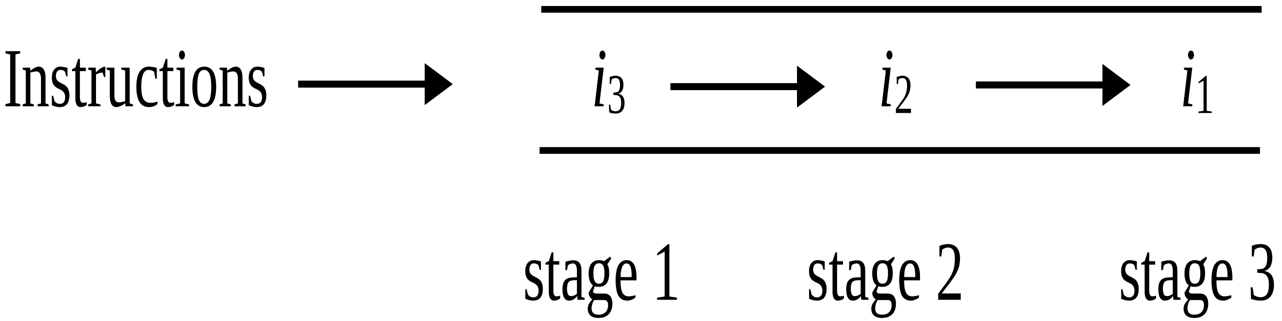
Fig. 11.3 shows the three-step loop that the CPU performs as it processes instructions:



1. Fetch the instruction pointed to by the `pc` register, and then increment the `pc` register by 4.
2. Decode the instruction (i.e., get the operands).
3. Execute the instruction.

Figure 11.3

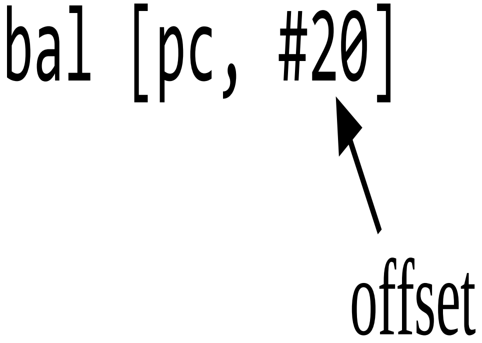
# Instruction pipeline



Instructions

$i_3$ $i_2$ $i_1$

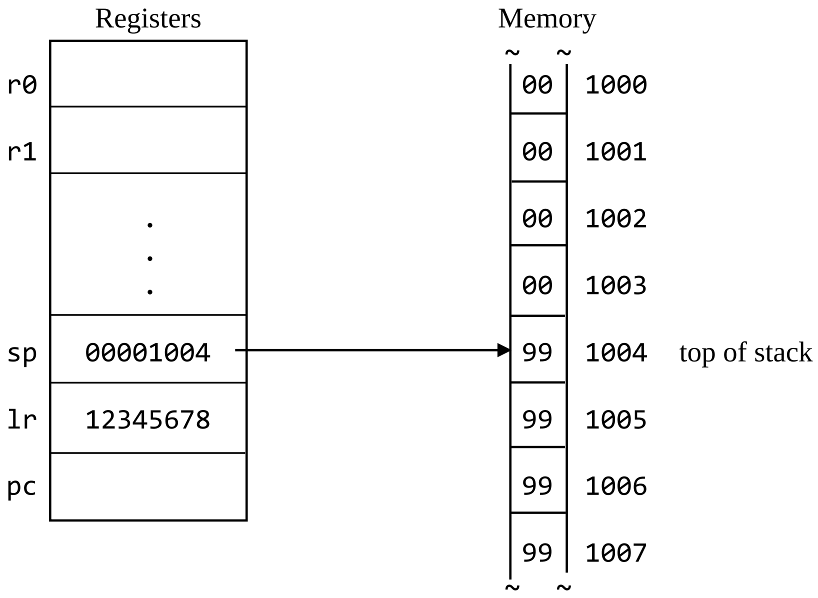
stage 1 stage 2 stage 3



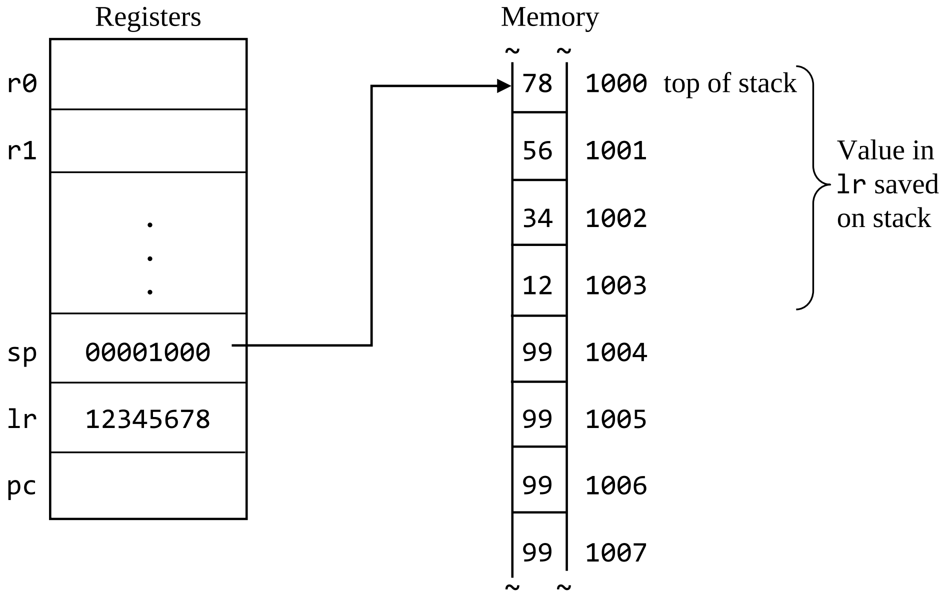
```
bal [pc, #20]
```

offset

1) Before a push of the value in `lr`:

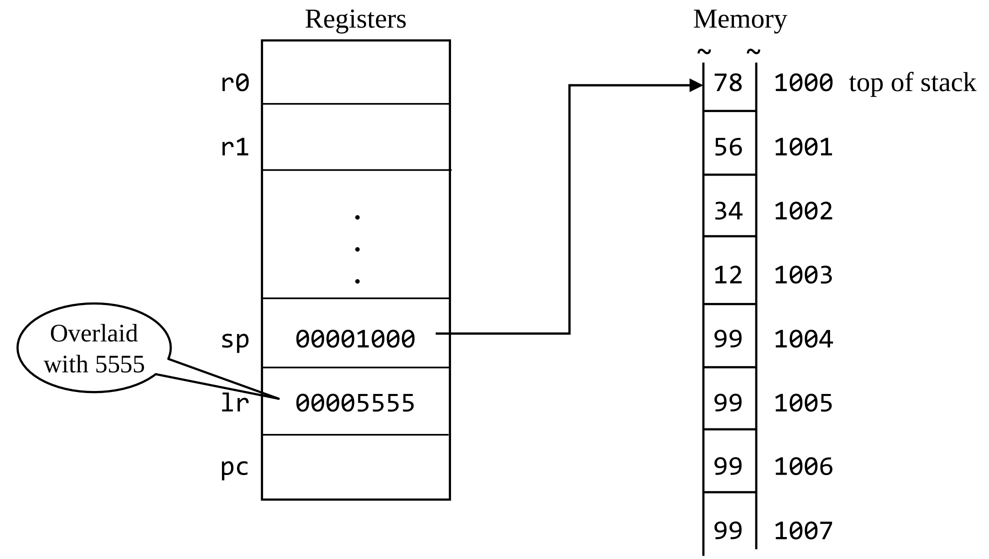


2) After the push of the value in `lr`:

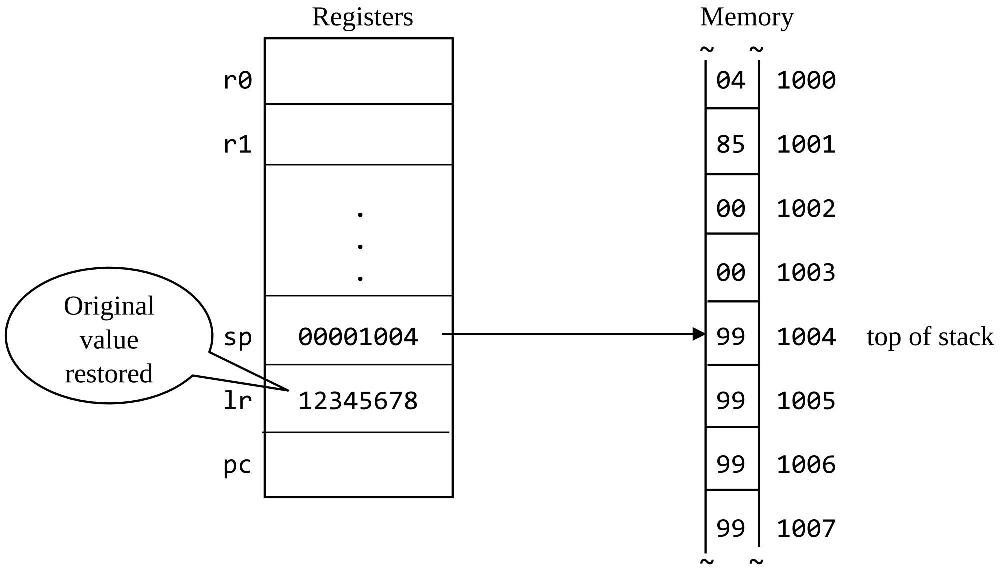


Now suppose the value in `lr` is overlaid with 5555.

3) Before a pop into `lr` register:



4) After the pop into `lr` (which restores `lr` with its original value).



# Some Simple Assembly Language Programs

```
                            @ ap1.s
          .text             @ start of read-only segment
          .global _start
_start:
          ldr r0, x         @ load r0 from x
          mov r7, #1        @ mov 1 into r7
          svc 0             @ supervisor call to terminate program

x:        .word 14          @ the variable x
```

```
6            mov r7, #1        @ mov 1 into r7
```

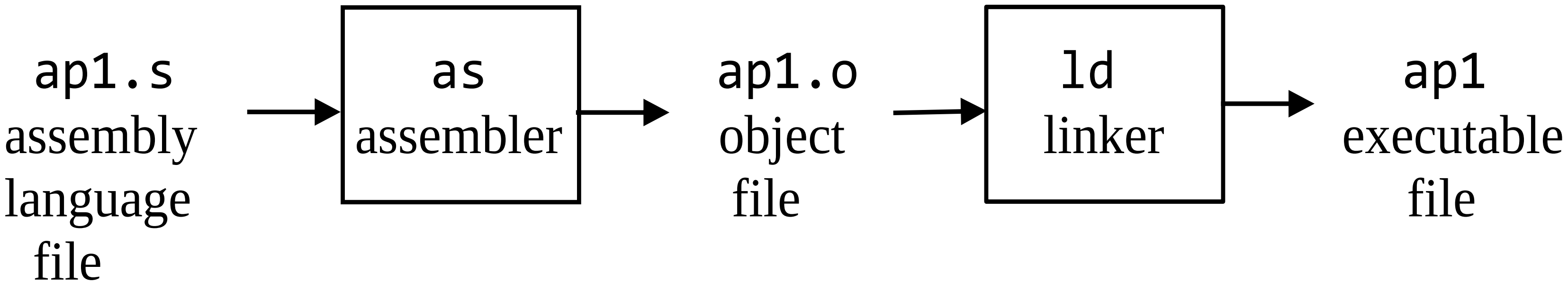
$$1110\ 0011\ 1010\ 0000\ 0111\ \underbrace{0000\ 0000\ 0001}_{\text{immediate data equal to 1}}$$

```
as ap1.s -o ap1.o

ld ap1.o -o ap1
```

`ap1` (or `./ap1`)

```
echo $?
```



`ap1.s` assembly language file

`as` assembler

`ap1.o` object file

`ld` linker

`ap1` executable file

Figure 11.7

```
rpi ap1.s -r
```

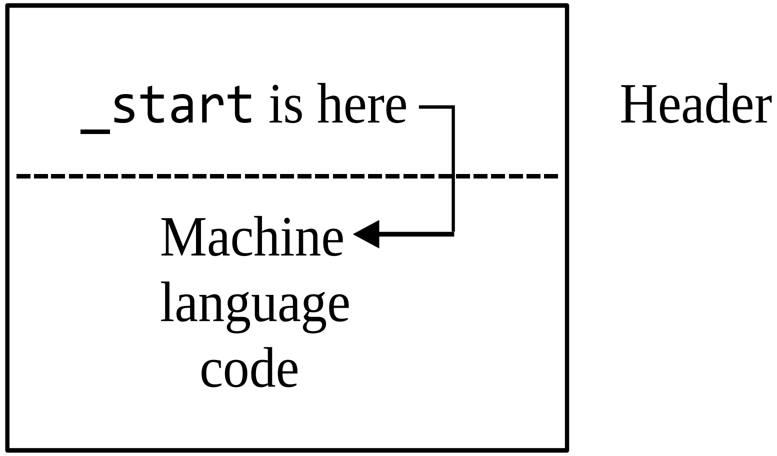


Figure 11.8

1. The OS searches for the `ap1` file.
2. The OS loads the `ap1` file into memory, adjusting addresses as necessary
3. The OS determines from the header in the `ap1` file which instruction should be executed first. It branches to that instruction.
4. The `ap1` program executes.
5. The `svc` instruction causes a branch back to the OS.
6. The OS displays the command line prompt, indicating it is ready to accept another command.

```
                            @ ap2.s
          .text             @ start of read-only segment
          .global _start
_start:
          ldr r0, x         @ load r0 from x
          str r0, y         @ store r0 in y (does not work)
          mov r7, #1        @ move 1 into r7
          svc 0             @ terminate program

x:        .word 2           @ the variable x
y:        .word 0           @ the variable y
```

illegal

Figure 11.9

```
segmentation error
```

# User and Supervisor Modes

```
                            @ ap3.s
          .text             @ start of read-only segment
          .global _start
                            @                    address
f:        mov r0, #3        @ mov 3 into r0       8000
          mov pc, lr        @ return to caller    8004

_start:   bl  f             @ call f              8008
          mov r7, #1        @ move 1 into r7      8012
          svc 0             @ terminate program   8016
```

# How an Address Fits into a Machine Instruction

```
ldr r0, x
```

```
ldr r0, [pc, #4]
```

```
                                 @ ap4.s
          .text                  @ start of read-only segment
          .global _start
_start:                          @ address
          ldr r0, [pc, #4]       @ 8000
          mov r7, #1             @ 8004
          svc 0                  @ 8008

x:        .word 14               @ 8012
```

# Using the .text and .data Directives

```
                            @ ap5.s
          .text             @ start of read only segment
          .global _start
_start:                 illegal
          ldr r0, x         @ does not work
          mov r7, #1
          svc 0

          .data             @ start of read/write segment
x:        .word 5           @ the variable x
```

Figure 11.12

```
                            @ ap6.s
          .text             @ start of read only segment
          .global _start
_start:
          ldr r0, ax        @ load address of x
          ldr r0, [r0]      @ load r0 from address in r0
          mov r7, #1
          svc 0
ax:       .word x           @ label x is a symbolic address

          .data             @ start of read/write segment
x:        .word 67          @ the variable x
```

Figure 11.13

```
                            @ ap7.s
          .text             @ read only segment
          .global _start
_start:
          ldr r0, =x        @ load address of x
          ldr r0, [r0]      @ load r0 from x
          mov r7, #1
          svc 0
                            @ literal pool is here

          .data             @ read/write segment
x:        .word 67          @ the variable x
```

# Linking Separately Assembled Modules

Consider the two files in Fig. 11.15. The two files, `m1.s` and `m2.s`, together make up one program.

m1.s

```
         .global _start
_start:
         bl  f
         mov r7, #1
         svc 0
```

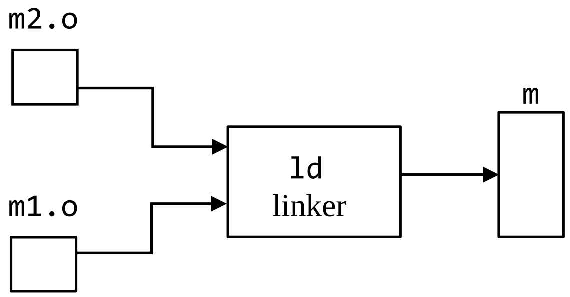
m2.s

```
         .global f
x:       .word 7
f:       ldr r0, x
         mov pc, lr
```

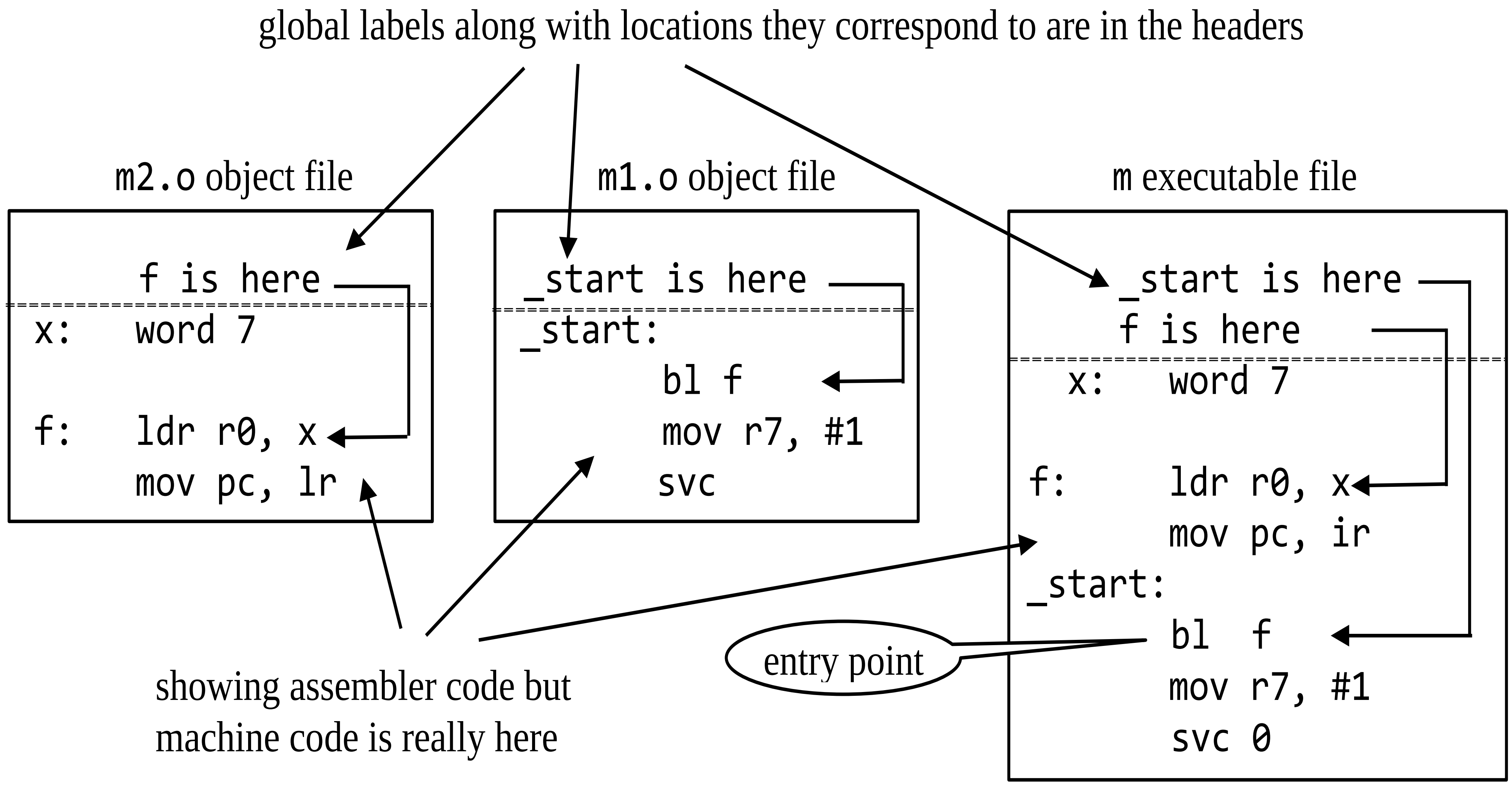
Figure 11.15

```
as m1.s -o m1.o
as m2.s -o m2.o

ld m2.o m1.o -o m
```



# Object and Executable File Format (simplified)



global labels along with locations they correspond to are in the headers

m2.o object file

```
     f is here
x:   word 7

f:   ldr r0, x
     mov pc, lr
```

m1.o object file

```
_start is here
_start:
       bl f
       mov r7, #1
       svc
```

m executable file

```
    _start is here
    f is here
 x:   word 7

f:    ldr r0, x
      mov pc, ir
_start:
      bl  f
      mov r7, #1
      svc 0
```

showing assembler code but
machine code is really here

Fig. 11.17