PC Relative Addressing

Copyright (c) 2024 - 2014 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

https://thinkingeek.com/arm-assembler-raspberry-pi/

PC Relative Addressing

PC (Program Count) **R15** Register

The Program Counter (or PC) is
a register inside the microprocessor
that stores the memory address
of the next instruction to be executed.

In ARM processors, the Program Counter is
a 32-bit register which is also known as R15.

The processor first fetches the instruction
from the address stored in the PC.

fetch

The fetched instruction is then decoded
so that it can be interpreted by the microprocessor.

decode

Once decoded, the instruction can then be executed
and the PC incremented so that it contains
the address of the next instruction.

execute

the **fetch**-**decode**-**execute** cycle.

http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

PC (Program Count) R15 Register

memory addresses are given in bytes (byte addresses)

memory is usually accessed by a word and
aligned on word boundaries. (word addresses)
for a high performance

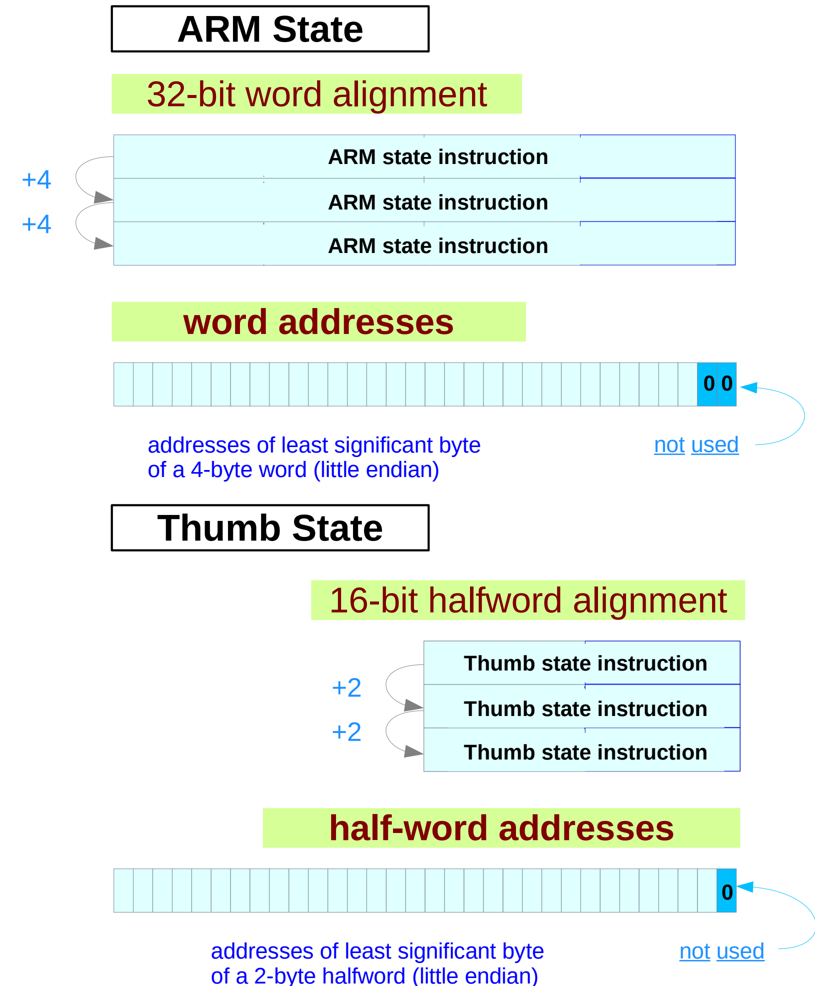
but also can be accessed by a byte or a halfword
with a performance loss

in ARM processors,
all ARM instructions take up one word (4 bytes).
all Thumb instructions take up one halfword (2 bytes).

incrementing the PC for the next instruction corresponds to

PC + 4 in the ARM state

PC + 2 in the Thumb state



http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

PC (Program Count) R15 Register

memory addresses are given in bytes (byte addresses)

memory is usually accessed by a word and
aligned on word boundaries. (word addresses)
for a high performance

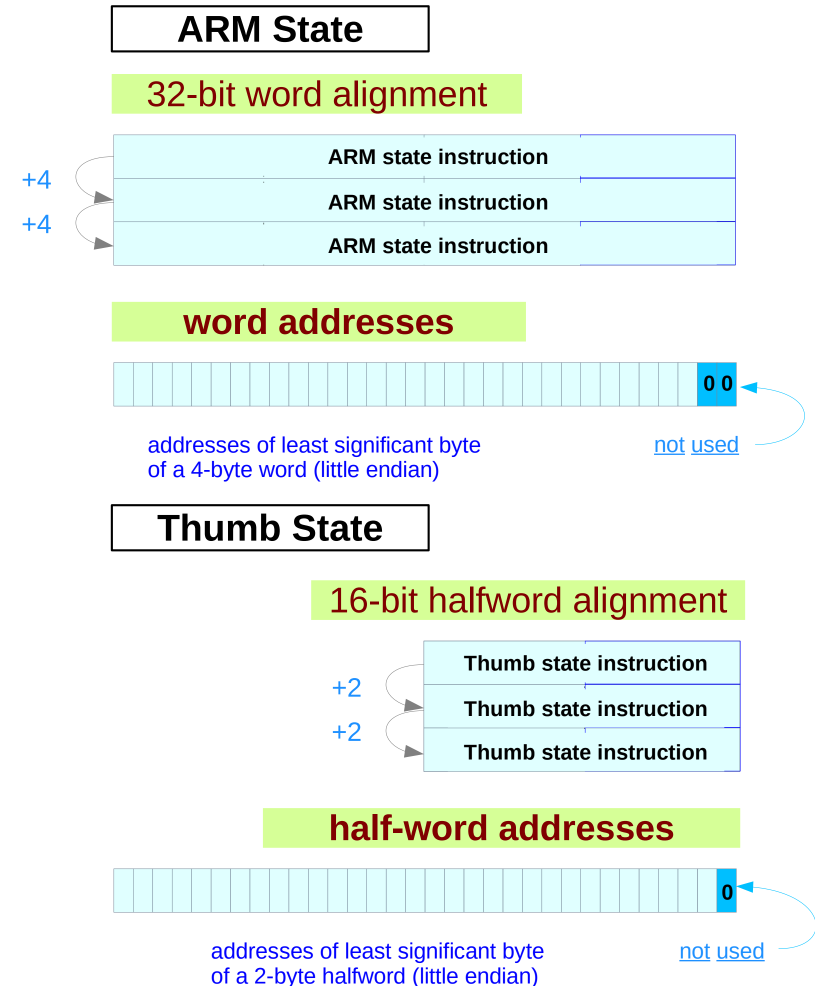
but also can be accessed by a byte or a halfword
with a performance loss

in ARM processors,
all ARM instructions take up one word (4 bytes).
all Thumb instructions take up one halfword (2 bytes).

incrementing the PC for the next instruction corresponds to

in the ARM state
PC + 4 (in a word address)

in the Thumb state
PC + 2 (in a halfword address)



http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

PC (Program Count) **R15** Register

The Program Counter is automatically incremented
by the size of the instruction executed.

This size is always 4 bytes in ARM state
and 2 bytes in THUMB mode.

When a branch instruction is being executed,
the PC holds the destination address.

During execution, PC stores

the address of the current instruction plus

- 8 (two ARM instructions) in ARM state,
- 4 (two Thumb instructions) in Thumb(v1) state.

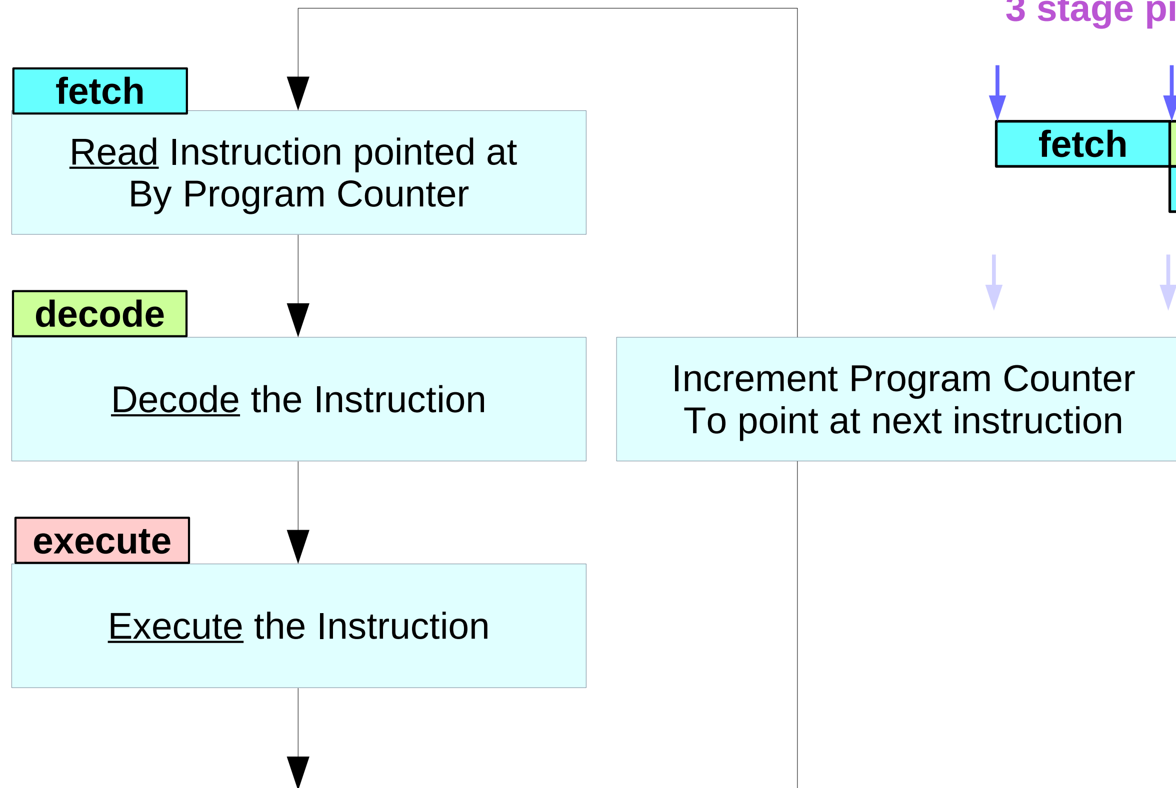
This is different from x86 where PC always points
to the next instruction to be executed.

Thus the content of PC

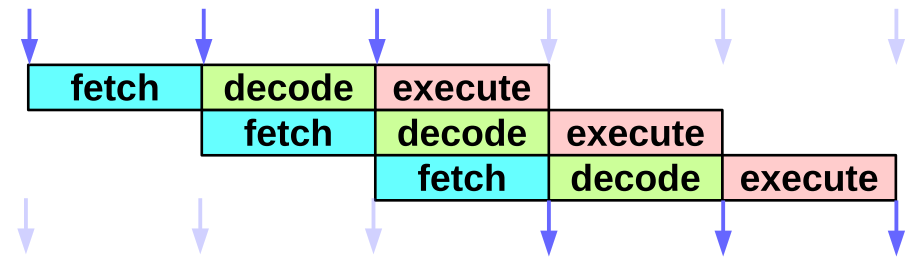
- a word address in ARM state
- a halfword address in Thumb state

https://azeria-labs.com/arm-data-types-and-registers-part-2/

PC (Program Counter) **R15** Register



3 stage pipeline execution

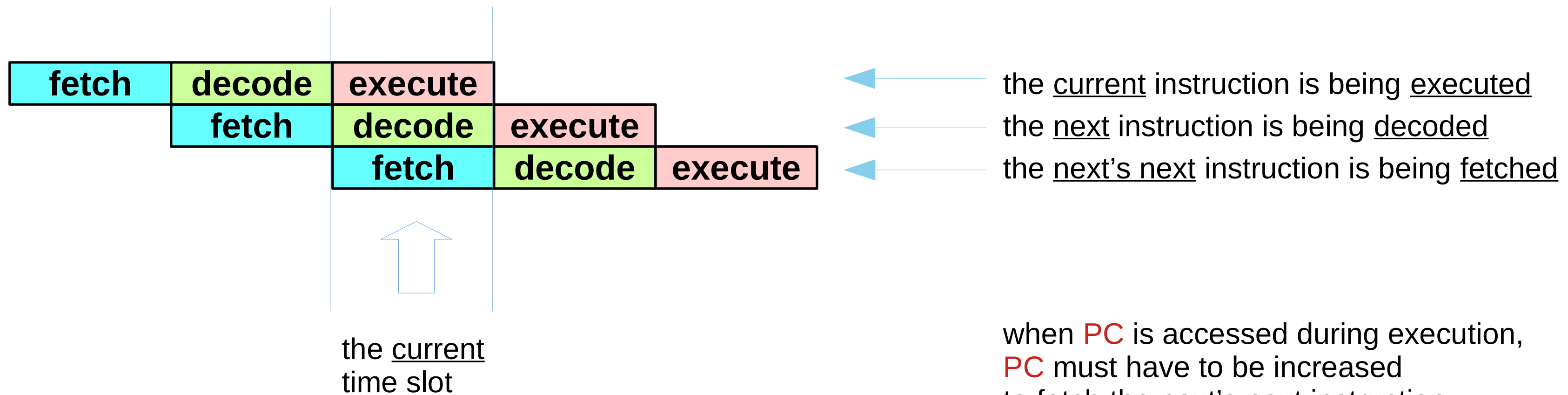


Execute a current instruction
Decode the next instruction
Fetch the next's next instruction

http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

PC (Program Counter) **R15** Register

3 stage pipeline execution



when PC is accessed during execution,
PC must have to be increased
to fetch the next's next instruction

PC + 8 for ARM instructions

PC + 4 for Thumb instructions

Execute a current instruction
Decode the next instruction
Fetch the next's next instruction

http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

Register relative and PC relative expressions (1)

armasm supports

PC-relative and
register-relative expressions.

a register-relative expression evaluates
to a named register combined with a numeric expression.

a PC-relative expression as a label or the PC,
optionally combined with a numeric expression.

1. using label
2. using PC

3. [PC, #number] for some instructions

https://developer.arm.com/documentation/dui0801/b/Cacdbfji

Register relative and PC relative expressions (2)

If you specify a label, the assembler calculates

the offset from the PC value
of the current instruction
to the address of the label.

the assembler encodes the offset
in the instruction.

If the offset is too large,
the assembler produces an error.

The offset is either added to or subtracted
from the PC value to form the required address.

ARM recommends you write

PC-relative expressions using labels

rather than PC
because the value of PC depends on the instruction set.

https://developer.arm.com/documentation/dui0801/b/Cacdbfji

PC relative addressing (1)

1 What is PC-relative addressing?

PC-relative addressing is a way of
specifying the address of an operand
by adding or subtracting a signed offset
to the PC register.

The offset is usually encoded
as an immediate value in the instruction,
and it represents the number of bytes
from the current instruction to the target location.

For example, if the PC register holds
the address of the current instruction,
and the offset is 4, then the operand address is PC + 4,
which is the address of the next instruction.

https://www.linkedin.com/advice/1/how-do-you-calculate-offset-pc-relative-addressing

PC relative addressing (2)

2 Why use PC-relative addressing?

PC-relative addressing offers several advantages
compared to other addressing modes,
such as absolute, register, or base-plus-offset.

It reduces the size of code
due to the offset usually being smaller
than the full address of the operand,
which also makes the code more portable
since it does not depend on the absolute memory location.

Additionally, PC-relative addressing
simplifies relocation of the code
since the offset does not need to be adjusted.

It also enables creation of position-independent code (PIC)
that can be executed from any address without modification
and facilitates implementation of control structures,
like jump tables, switch statements, and loops,
by using the PC register as a base
for accessing a table of offsets.

https://www.linkedin.com/advice/1/how-do-you-calculate-offset-pc-relative-addressing

PC relative addressing (3)

3 How to calculate the offset for PC-relative addressing?

The offset for PC-relative addressing
depends on the instruction set architecture (ISA)
and the assembler syntax of the assembly language.

Different ISAs may have different rules for encoding
and interpreting the offset, and different assemblers
may have different ways of expressing
the offset in the source code.

However, a general formula for calculating the offset is:

offset = target_address - (current_address + instruction_size)

where target_address is the address of the operand,
current_address is the address of the current instruction,
and instruction_size is the size of the current instruction in bytes.

The offset may be positive or negative,
depending on whether the target_address is ahead or
behind the current_address.

https://www.linkedin.com/advice/1/how-do-you-calculate-offset-pc-relative-addressing

PC relative addressing (4)

4 How to use PC-relative addressing in assembly?

To use PC-relative addressing in assembly,
you need to know the syntax of the assembler
and the format of the instructions
that support this addressing mode.

For example, in ARM assembly,
you can use PC-relative addressing
with instructions such as
LDR (load register),
STR (store register),
B (branch), and
BL (branch with link).

The syntax for these instructions is:

```
LDR Rd, [PC, #offset]
STR Rd, [PC, #offset]
B label
BL label
```

where Rd is the destination or source register,
offset is the immediate value of the offset,
and label is a symbolic name for the target address.

The assembler will calculate the offset
based on the formula given above,
and encode it in the instruction.

Note that some instructions may have
limitations on the range
or alignment of the offset, and some
may require a special syntax
for PC-relative addressing.

https://www.linkedin.com/advice/1/how-do-you-calculate-offset-pc-relative-addressing

PC relative addressing (5)

5 Examples of PC-relative addressing in assembly

To demonstrate the use of
PC-relative addressing in assembly,
here are some examples
of code snippets using this technique.

For instance,
loading a constant value from a literal pool
may involve the command 'LDR R0, [PC, #8]',
while accessing a global variable requires
'LDR R0, var(PC)'.

Additionally, implementing a jump table requires
'LDR R0, [PC, R1, LSL #2]'.

Each of these commands is designed to load
the address of the case at PC + (R1 * 4),
followed by a 'BX R0' branch to the case.

Finally, each case is labeled and contains code
for the respective action before branching to the end.

https://www.linkedin.com/advice/1/how-do-you-calculate-offset-pc-relative-addressing

PC relative addressing (6)

You will be using the PC as the base address,
and you need to put the address within reach
of a pc-relative LDR.

So you'll do:

LDR PC, test_addr

...

test_addr .word 0x0803FF00

You just need to make sure there's not too much code
between the LDR and the '.word'.

This will translate to LDR PC, [PC, #offset].

I don't remember the reach of the offset off hand
but it's not that big.
For a short though this shouldn't be an issue.

You can google 'arm literal pool'
and find out more about this method,
we do not however support
the '=' syntax in the TI assembler
so you have to put the .word in yourself.

https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/331018/jump-to-address-store

PC relative addressing (7-1)

I think you probably need to do this in 2 steps then.

First jump to some addresss CLOSE to 0x0803FF00,
using the same method talked about in the last post.
(i.e. change the .word to something like 0x0803FE00
.. close to 0x0803FF00).

At this address you need to make sure
there is another LDR PC, [PC,#idx]
that points to 0x0803FE00
so that the contents of 0x0803FE00 gets loaded into the PC.
I don't think you can do this in one step
because the LDR we're using is limited to an imm12 offset...
and that's probably not going to get you from flash to RAM in one step.

https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/331018/jump-to-address-store

PC relative addressing (7-2)

I skimmed through the instruction set
to see if there's another way and I don't see it,
but I might be missing something.

So you can always also see
if ARM's support system has any better hints/ideas.

-Anthony

EDIT: you might also look at "SVC" but this is an exception
so it's not exactly what you want.
But its' another way to make a jump table call
without messing up the user mode registers.

https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/331018/jump-to-address-store

PC relative addressing (8)

PC register holds pointer to next instruction LDR instruction
is loading the value of second operand
into first operand (for example)

LDR r0, [pc, 0x5678]

is equivalent to this "C code"

r0 = *(pc + 0x5678)

It's pointer dereferencing with base offset.

And my question:
I found this code

LDR PC, [PC,-4]

It's commented like monkey patching, etc..

https://stackoverflow.com/questions/24115899/arm-ldr-instruction-on-pc-register

PC relative addressing (9)

And my question:
I found this code

LDR PC, [PC,-4]

It's commented like monkey patching, etc..
How I understand this code

pc = *(pc - 4)

I this case "pc" register will dereference
the address of previous instruction
and will contain the "machine code" of instruction
(not the address of instruction),
and program will jump to that invalid address
to continue execution,
and probably we will get "Segmentation Fault".

So what I'm missing or not understanding?

https://stackoverflow.com/questions/24115899/arm-ldr-instruction-on-pc-register

PC relative addressing (10)

The thing that makes me to think is the brackets of second operand in LDR instruction.
As I know on x86 architecture brackets are already dereferencing the pointer,
but I can't understand the meaning in ARM architecture.

```
mov r1, 0x5678
add r1, pc
mov r0, [r1]
```

is this code equivalent to?

```
LDR r0, [pc, 0x5678]
```

https://stackoverflow.com/questions/24115899/arm-ldr-instruction-on-pc-register

PC relative addressing (11)

The thing that makes me to think is the brackets of second operand in LDR instruction.
As I know on x86 architecture brackets are already dereferencing the pointer,
but I can't understand the meaning in ARM architecture.

```
mov r1, 0x5678
add r1, pc
mov r0, [r1]
```

is this code equivalent to?

```
LDR r0, [pc, 0x5678]
```

e the edit: mov cannot take a memory operand (ARM is a load-store architecture),
so that code is invalid as is - if the third instruction was ldr r0, [r1] it would be equivalent.
ldr r0, [pc, 0x5678] can't be encoded as a single instruction as the immediate is too big
(i.e. it can't be represented by an 8-bit value rotated by an even number of bits).

https://stackoverflow.com/questions/24115899/arm-ldr-instruction-on-pc-register

PC relative addressing (12)

LDR PC,[PC, -4] means load a word from the address formed by the current PC (R15) minus 4 and put that value in PC. Since PC is 8 bytes ahead of the current instruction you'll be loading from current_instruction_address+8-4 == current_instruction_address+4 –
Michael
Commented Jun 9, 2014 at 8:09
Thanks, but one more thing [PC, -4] what does it do ? "pc - 4" or *(pc - 4) ? –
l0gg3r
Commented Jun 9, 2014 at 8:12
That would be the latter. –
Michael
Commented Jun 9, 2014 at 8:15
in that case LDR PC, [PC,-4] will equivalent to this code pc = **((pc - 4)),
or my misunderstanding is that LDR is not dereferencing the pointer? –

https://stackoverflow.com/questions/24115899/arm-ldr-instruction-on-pc-register

The values of PC in ARM and Thumb states

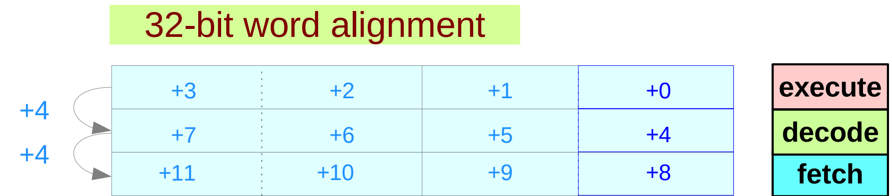
In **A32** code, PC + 8
the value of the PC is
the address of the current instruction plus 8 bytes.

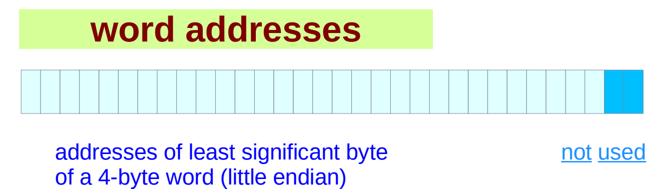
In **T32** code: PC + 4
the value of the PC is

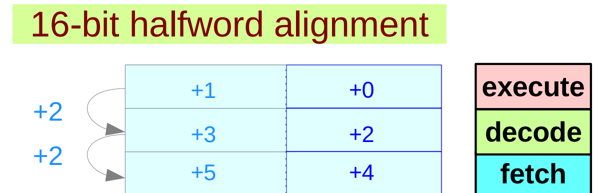
the address of the current instruction plus 4 bytes.
for **B**, **BL**, **CBNZ**, and **CBZ** instructions,

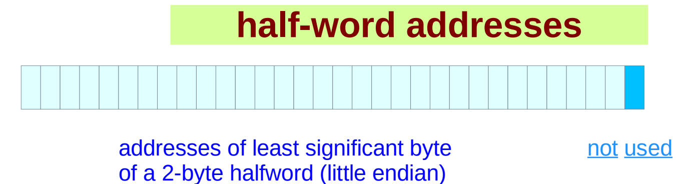
the address of the current instruction plus 4 bytes,
with bit[1] of the result cleared to 0
for all other instructions that use labels,

In **A64** code, PC
the value of the PC is
the address of the current instruction.









https://developer.arm.com/documentation/dui0801/b/Cacdbfji

The values of PC in ARM and Thumb states

In hardware, PC in Thumb state can point any halfword

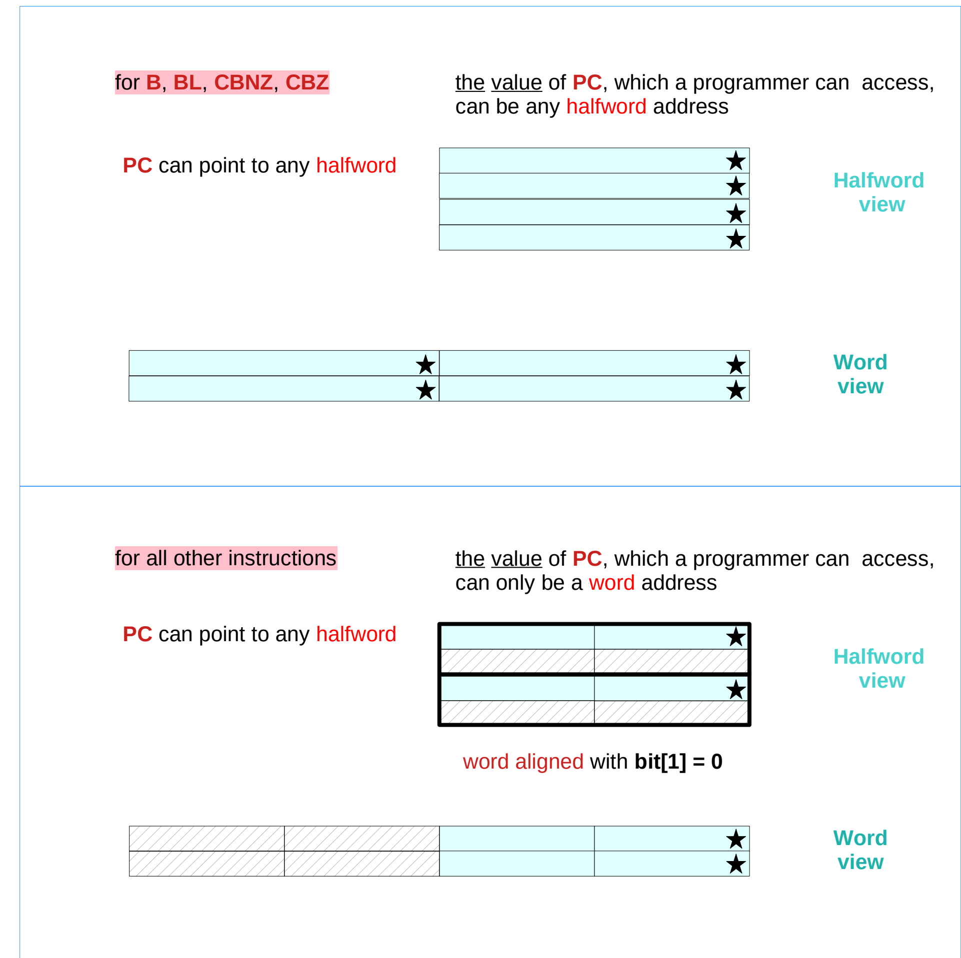
when a programmer access PC,
it value can be

- (PC + 4)

  any halfword address
  for **B**, **BL**, **CBNZ**, **CBZ** instructions

- (PC + 4) with bit[0]=0

  only a word address
  for all other instructions



https://developer.arm.com/documentation/dui0801/b/Cacdbfji

The values of PC in ARM and Thumb states

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value_0
        ; n-1 DCD directives
        DCD     value_n         ; data+4*n points here
        ; more DCD directives
```

https://developer.arm.com/documentation/dui0801/b/Cacdbfji

The values of PC in ARM and Thumb states

```
int f,g,y;//global variables
int sum(int a, int b){
     return (a+b);
}
int main(void){
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}
```

```
    00008390 <sum>:
int sum(int a, int b) {
return (a + b);
}
    8390: e0800001 add r0, r0, r1
    8394: e12fff1e bx lr
    00008398 <main>:
int f, g, y; // global variables
int sum(int a, int b);
int main(void) {
    8398: e92d4008 push {r3, lr}
f = 2;
    839c: e3a00002 mov r0, #2
    83a0: e59f301c ldr r3, [pc, #28] ; 83c4 <main+0x2c>
    83a4: e5830000 str r0, [r3]
g = 3;
    83a8: e3a01003 mov r1, #3
    83ac: e59f3014 ldr r3, [pc, #20] ; 83c8 <main+0x30>
    83b0: e5831000 str r1, [r3]
y = sum(f,g);
    83b4: ebfffff5 bl 8390 <sum>
    83b8: e59f300c ldr r3, [pc, #12] ; 83cc <main+0x34>
    83bc: e5830000 str r0, [r3]
return y;
}
83c0: e8bd8008 pop {r3, pc}
83c4: 00010570 .word 0x00010570
83c8: 00010574 .word 0x00010574
83cc: 00010578 .word 0x00010578
```

https://stackoverflow.com/questions/24091566/why-does-the-arm-pc-register-point-to-the-instruction-after-the-next-one-to-be-e

The values of PC in ARM and Thumb states

see the above LDR's PC value--here
is used to load variable f,g,y's address to r3.

```
83a0: e59f301c ldr r3, [pc, #28];83c4 main+0x2c
PC=0x83c4-28=0x83a8-0x1C = 0x83a8
```

PC's value is just the current executing instruction's next's next instruction.
as ARM uses 32bits instruction, but it's using byte address,
so + 8 means 8bytes, two instructions' length.

so attached ARM archi's 5 stage
pipe line fetch, decode, execute, memory, writeback

ARM's 5 stage pipeline

the PC register is added by 4 each clock,
so when instruction bubbled to execute--the current instruction,
PC register's already 2 clock passed!

now it's + 8. that actually means:
PC points the "fetch" instruction, current instruction
means "execute" instruction, so PC means the next next to be executed.

https://stackoverflow.com/questions/24091566/why-does-the-arm-pc-register-point-to-the-instruction-after-the-next-one-to-be-e

References

[1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
[2] http://blog.bobuhiro11.net/2014/01-13-baremetal.html
[3] http://www.valvers.com/open-software/raspberry-pi/
[4] https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html