Branch and Return Methods

Young Won Lim
12/26/24

Based on

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

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

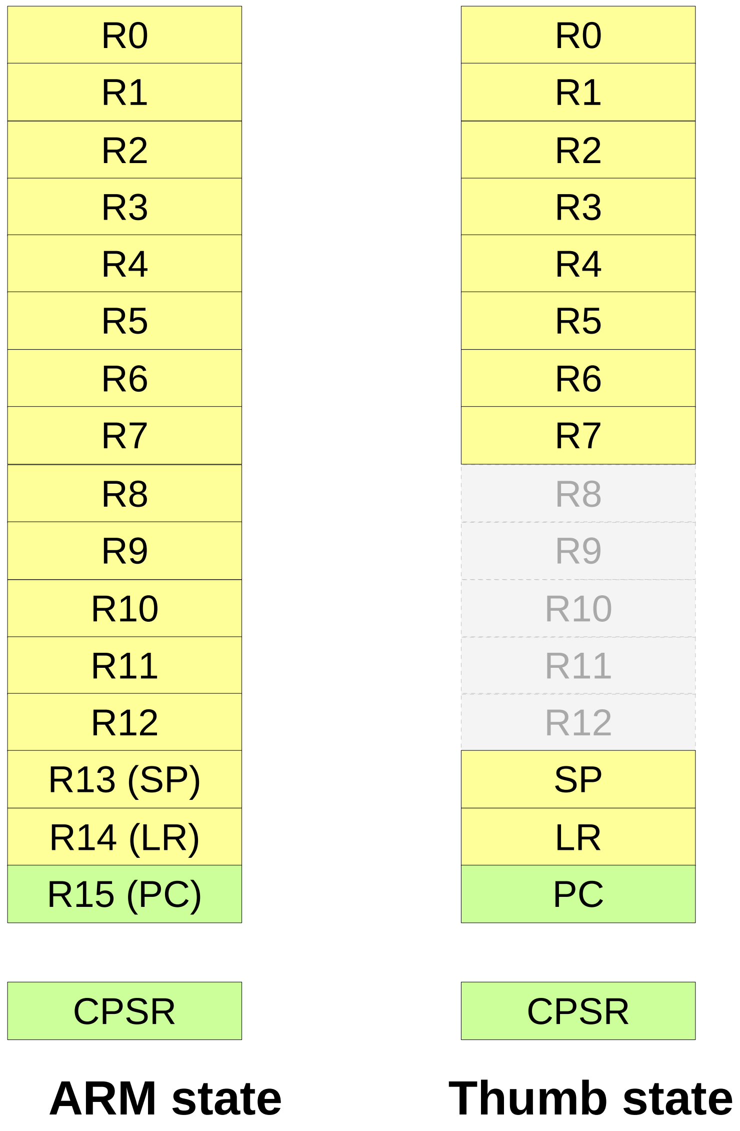
Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

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

Branch and Return Instructions

ARM vs. Thumb programmer’s models



ARM state

- 16 + 1 = 17 normal registers

Thumb state

- 11 + 1 = 12 normal registers

ARM Register Sets (2-1)

- The biggest register difference involves the **SP** register.
  - the Thumb state
    unique stack mnemonics (**PUSH**, **POP**)
  - the ARM state.
    no such stack mnemonics (**PUSH**, **POP**)

- **PUSH**, **POP** instructions assume
  the existence of a stack pointer (**R13**)

- **PUSH**, **POP** instructions translate
  into **load** and **store** instructions
  in the ARM state.

https://www.embedded.com/introduction-to-arm-thumb/

PUSH and **POP** Thumb instructions (1)

PUSH stores registers on the stack,

with the lowest numbered register
using the lowest memory address

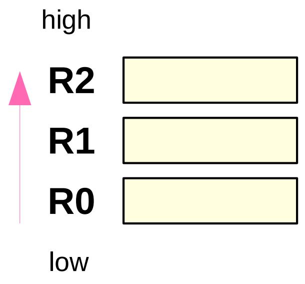
and the highest numbered register
using the highest memory address.

POP loads registers from the stack,

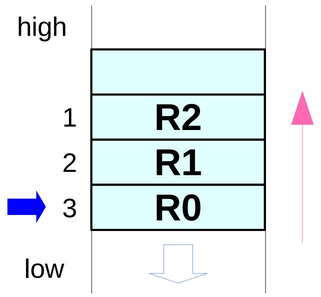
with the lowest numbered register
using the lowest memory address

and the highest numbered register
using the highest memory address.

the registers in the **{ }** can be specified in any order,
but the order in which they appear on the stack is fixed



Full Descending Stack



https://stackoverflow.com/questions/63304428/ordering-of-registers-in-push-and-pop-brackets

PUSH and **POP** Thumb instructions (2)

If you want these in a different order then you have to write them in separate instructions in the assembly language.

The registers are stored in sequence,
the lowest-numbered register
to the lowest memory address (start_address),

through to the highest-numbered register
to the highest memory address (end_address)

the start_address is the value of the SP
minus 4 times the number of registers to be stored.

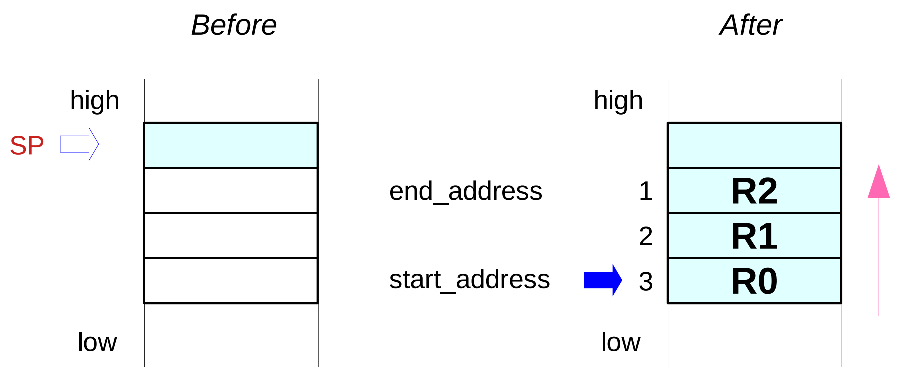
Subsequent addresses are formed
by incrementing the previous address by four.

One address is produced for each register
that is specified in .

The end_address value is four less
than the original value of SP.

The SP register is decremented
by four times the numbers of registers in .

Full Descending Stack



start_address = SP – 4 * 3 new SP

end_address = SP – 4

https://stackoverflow.com/questions/63304428/ordering-of-registers-in-push-and-pop-brackets

PUSH and **POP** Thumb instructions (3)

So according to the above explanations,
the ordering of registers in one PUSH bracket
doesn't matter.

PUSH {R0, R1, R2}
PUSH {R2, R1, R0}
PUSH {R1, R2, R0}

all would result in the some ordering in the stack

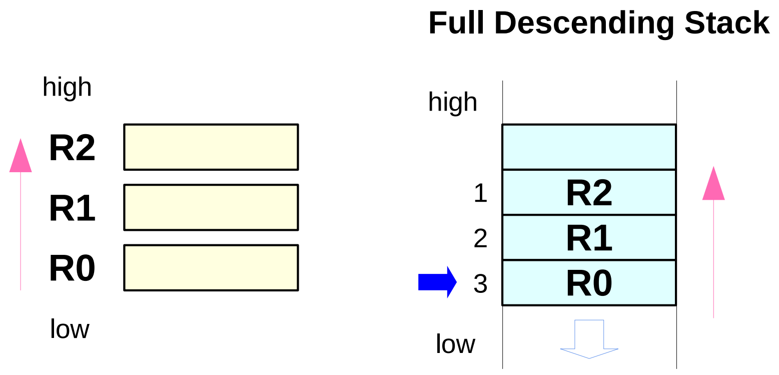
the lowest / highest numbered register (**R0**/**R2**)
uses the lowest / highest (stack) memory address

if a single **PUSH** instruction has
multiple registers in the bracket,

looks like sorted pushing actions

first, **PUSH R2** to take the highest address,
followed by **PUSH R1** and
ended with **PUSH R0** to take the lowest address

but **bit mask** is actually used in implementation



PUSH {R0, R1, R2}

PUSH {R2, R1, R0}

PUSH {R1, R2, R0}

1	**PUSH R2**
2	**PUSH R1**
3	**PUSH R0**

all equivalent instructions

https://stackoverflow.com/questions/63304428/ordering-of-registers-in-push-and-pop-brackets

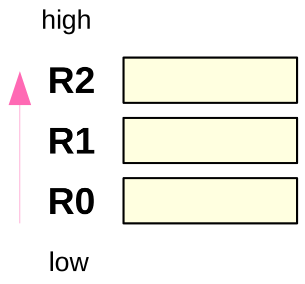
PUSH and **POP** Thumb instructions (4)

if **R2** get pushed last and popped first in a LIFO stack
  or **SP** pointing **R2**
  or **R2** takes the lowest stack address,

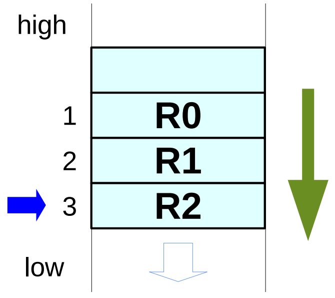
there is no single **PUSH** bracket statement

but three separate PUSH instructions must be used

```
PUSH R0
PUSH R1
PUSH R2
```



Full Descending Stack



If you look at assembled hex/binary,
you'll find that **push** with same registers
but different order encode to the same instruction.

That will be related to instruction encoding,
because it's pretty much a **bitmask** of registers

No single PUSH Instruction is possible

1	**PUSH R0**
2	**PUSH R1**
3	**PUSH R2**

https://stackoverflow.com/questions/63304428/ordering-of-registers-in-push-and-pop-brackets

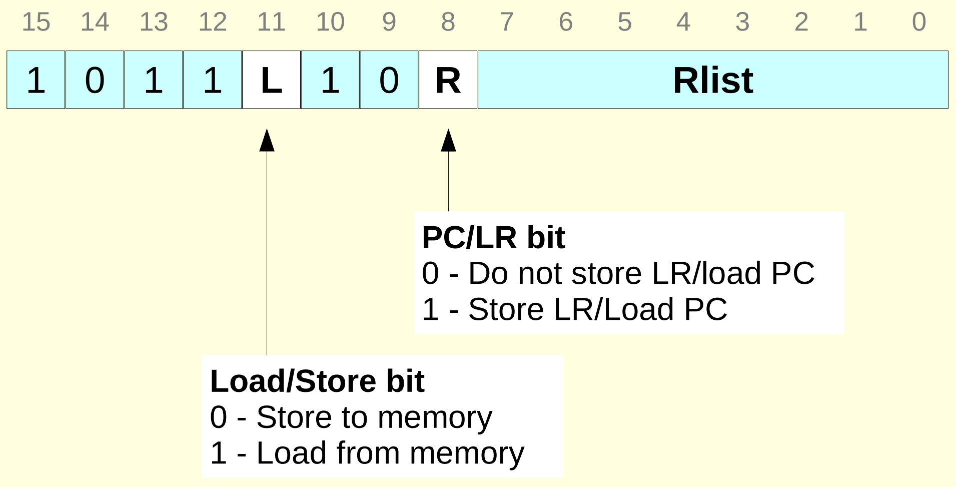
PUSH and **POP** Thumb instructions (5)

The instructions in this group allow registers 0-7
and optionally **LR** to be pushed onto the stack,

and registers 0-7 and optionally **PC**
to be popped off the stack.

The stack is always assumed to be Full Descending.

L	**R**		
0	0	**PUSH**	**{ Rlist }**
0	1	**PUSH**	**{ Rlist, LR }**
1	0	**POP**	**{ Rlist }**
1	1	**POP**	**{ Rlist, PC }**



PUSH {R0-R4,LR}
; Store **R0**,**R1**,**R2**,**R3**,**R4** and **R14** (**LR**) at the stack
; pointed to by **R13** (**SP**) and update **R13**.
; Useful at start of a sub-routine to
; save workspace and return address.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	**0**	1	0	**1**	0	0	0	1	1	1	1	1

POP {R2,R6,PC}
; Load **R2**,**R6** and **R15** (**PC**) from the stack
; pointed to by **R13** (**SP**) and update **R13**.
; Useful to restore workspace and return from sub-routine.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	**1**	1	0	**1**	0	1	0	0	0	1	0	0

http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf

PUSH and POP Thumb instructions (6)

```
.thumb

push  {r0,r1,r2}
push  {r2,r1,r0}
push  {r0}
push  {r1}
push  {r2}
```

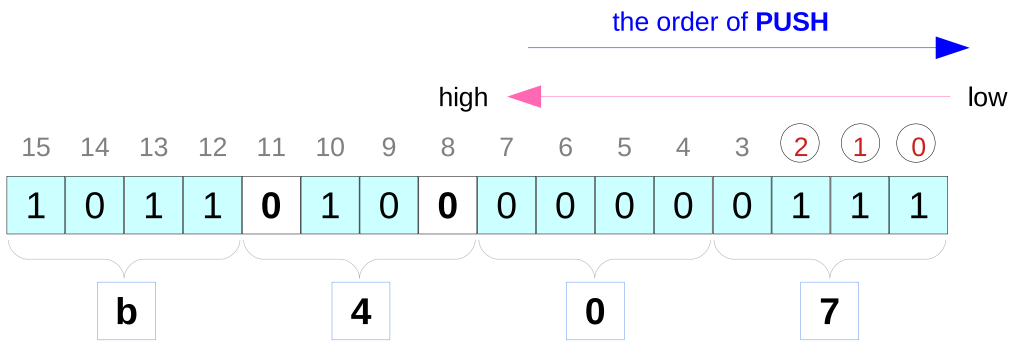
Disassembly of section .text:

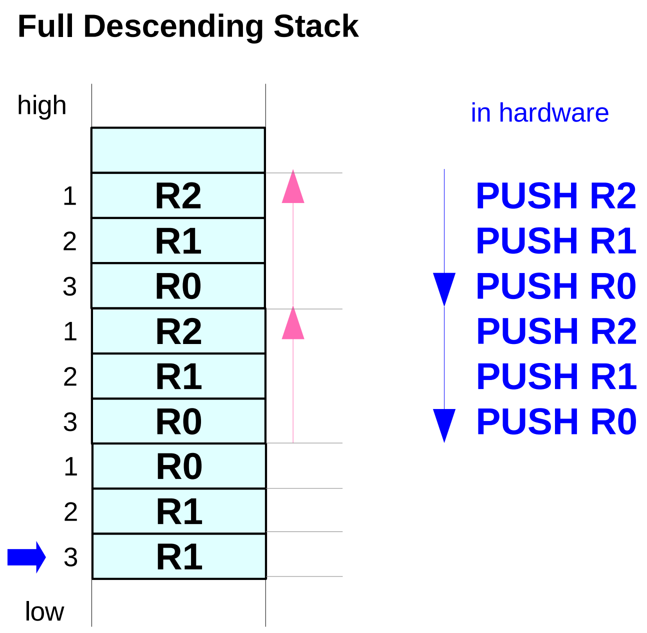
```
00000000 <.text>:
   0:   b407       push    {r0, r1, r2}
   2:   b407       push    {r0, r1, r2}
   4:   b401       push    {r0}
   6:   b402       push    {r1}
   8:   b404       push    {r2}
```

in the push instruction the lower 8 bits are a register list/mask.
So the **7** in **b407** indicates the three registers **r0**,**r1**,**r2**.

the hardware goes from bit 7 to bit 0 if set then push that register

If you want these in a different order then you have to write them in separate instructions in the assembly language.





https://stackoverflow.com/questions/63304428/ordering-of-registers-in-push-and-pop-brackets

ARM Register Sets (2-2)

- The CPSR register holds
  - processor mode bits (user or exception flag)
  - interrupt mask bits
  - condition codes and
  - Thumb status bit

- The Thumb status bit (**T**) indicates the processor's current state:
  - 0 for ARM state (default)
  - 1 for Thumb.

- Although other bits in the CPSR may be modified in software, it's dangerous to write to **T** directly;
  - the results of an improper state change are *unpredictable*.

N Negative flag
Z Zero flag
C Carry flag
V Overflow flag

To disable Interrupt (**IRQ**), set **I**
To disable Fast Interrupt (**FIQ**), set **F**

USR	User mode
FIQ	Fast Interrupt mode
SVC	Supervisor mode
ABT	Abort mode
UND	Undefined mode
SYS	System mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	**Z**	**C**	**V**																					**I**	**F**	**T**			**mode**		

https://www.embedded.com/introduction-to-arm-thumb/

Branch instructions

B,	BL,
BX,	BLX

BL and **BLX** copy the return address into **LR** (**R14**)

B,	BL,
BX,	BLX

BX and **BLX** can change the processor state

https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/branch-and-control-instructions/b--bl--bx--blx--and-bxj

Branch instructions and operand types

• B {cond} label • ~~B {cond} Rm~~	• BL {cond} label • ~~BL {cond} Rm~~
• ~~BX {cond} label~~ • BX {cond} Rm	• BLX {cond} label • BLX {cond} Rm

Branch
Branch with **L**ink
Brand and e**X**change
Brand with **L**ink and e**X**change

• B {cond} label	• BL {cond} label
• ~~BX {cond} label~~	• BLX {cond} label

- B {cond} label
- BL {cond} label
- BLX {cond} label

• ~~B {cond} Rm~~	• ~~BL {cond} Rm~~
• BX {cond} Rm	• BLX {cond} Rm

- BX {cond} Rm
- BLX {cond} Rm

https://www.embedded.com/introduction-to-arm-thumb/

B and **BL** instructions (1)

- **B** {cond} label
- ~~B {cond} Rm~~

- **BL** {cond} label
- ~~BL {cond} Rm~~

Branch
Branch with **L**ink
Brand and e**X**change
Brand with **L**ink and e**X**change

- cond is an optional condition code
- label is a program-relative expression

- The **B** instruction
  - causes a branch to label.

- The **BL** instruction
  - copies the address of the next instruction into **r14** (**lr**, the link register)
  - causes a branch to label.

https://www.embedded.com/introduction-to-arm-thumb/

B and **BL** instructions (2)

- machine-level **B** and **BL** instructions
  have a range of ±32Mb
  from the address of the current instruction.

  - However, you can use these instructions
    even if label is out of range.

  - Often you do not know
    where label is placed by the linker.

  - When necessary, the ARM linker
    adds veneer code to allow longer branches

2^{24} Byte = 2^4 MB = 16 MB

➡ +/- 8 MB (forward, backward)

➡ +/- 32 MB (2 lsb's : 4 bytes alignment)

https://www.embedded.com/introduction-to-arm-thumb/

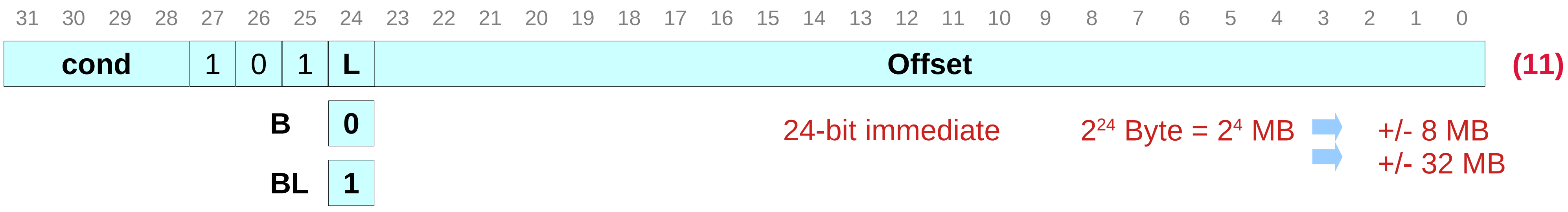
B and **BL** instructions (3)

- The ARM **BL** instruction has a 24-bit immediate for encoding the branch offset
- this would give you a range of 2^{24} bytes, or +/-8MB (given that the immediate allows forwards or backwards).
- all ARM instructions are 4 bytes long, and must be size aligned.
- no need to consider the *two* least significant bits of the address
- taking our branch range from +/-8MB to +/-32MB.

2^{24} Byte = 2^4 MB = 16 MB

➡ +/- 8 MB (forward, backward)

➡ +/- 32 MB (2 lsb's : 4 bytes alignment)



24-bit immediate

2^{24} Byte = 2^4 MB ➡ +/- 8 MB
➡ +/- 32 MB

https://community.arm.com/support-forums/f/architectures-and-processors-forum/3061/range-of-bl-instruction-in-arm-state

BX and BLX instructions (1)

- ~~BX {cond} label~~
- BX {cond} Rm

- BLX {cond} label
- BLX {cond} Rm

- cond is an optional condition code
- label is a program-relative expression
- Rm is a register containing an address to branch to

- The **BX** instruction
  - causes a branch to the address contained in Rm
  - changes the instruction set, if required:

- The **BLX** instruction
  - copies the address of the next instruction
    into **r14** (**lr**, the link register)
  - causes a branch to label.
  - can change the instruction set

Branch
Branch with **L**ink
Brand and e**X**change
Brand with **L**ink and e**X**change

https://www.embedded.com/introduction-to-arm-thumb/

BX and **BLX** instructions (2)

• **B** {cond} label • ~~**B** {cond} Rm~~	• **BL** {cond} label • ~~**BL** {cond} Rm~~
• ~~**BX** {cond} label~~ • **BX** {cond} Rm	• **BLX** {cond} label • **BLX** {cond} Rm

Branch
Branch with **L**ink
Brand and e**X**change
Brand with **L**ink and e**X**change

with label
always changes the state.
ARM state → Thumb state
Thumb state → ARM state

with Rm
Rm[0] = **0** → to ARM state
Rm[0] = **1** → to Thumb state

Both ARM state
and Thumb state provide
B, **BL**, **BX**, **BLX**

https://www.embedded.com/introduction-to-arm-thumb/

B, **BL**, **BX**, and **BLX** instructions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	**L**	**Offset**																							

Branch	**0**	**B{<cond>} <address>**	PC := Offset
Branch with Link	**1**	**BL{<cond>} <address>**	R14 := PC+8; PC := Offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	**0**	1	**Rn**			

Branch and Exchange **BX{<cond>} Rn** PC := Rn;

Rn[0] = **0** → to ARM state
Rn[0] = **1** → to Thumb state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	**1**	1	**Rn**			

Branch with Link and Exchange **BLX{<cond>} Rn** R14 := PC+8; PC := Rn;

Rn[0] = **0** → to ARM state
Rn[0] = **1** → to Thumb state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	**1**	**1**	**1**	1	0	1	**H**	**Offset**																							

Branch with Link and Exchange **BLX{<cond>} <address>** PC := Offset

always changes the state.
ARM state → Thumb state
Thumb state → ARM state

Branch instructions – changing the state

BX Rn
BLX Rn

changes the state depending on bit[0] of Rn:

Rn[0] = **0** → ARM state
Rn[0] = **1** → Thumb state

cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	**0**	1	**Rn**	

Branch and Exchange **BX{<cond>} Rn** **PC := Rn;**

cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	**1**	1	**Rn**	

Branch with Link and Exchange **BLX{<cond>} Rn** **R14 := PC+8;** **PC := Rn;**

BLX *label* always changes the state.

ARM state → Thumb state
Thumb state → ARM state

1	**1**	**1**	**1**	1	0	1	**H**	**Offset**

Branch with Link and Exchange **BLX{<cond>} <address>** **PC := Offset**

https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/branch-and-control-instructions/b--bl--bx--blx--and-bxj

BLX in ARM Architecture v5

In ARM Architecture v5
both ARM and Thumb state

provide a **BLX** instruction

that will call a subroutine addressed by a register

and correctly sets the return address
to the sequentially next value of the program counter.

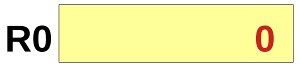
/IHI0042E_aapcs.pdf

Switching the state (1) **BX** or **BLX**

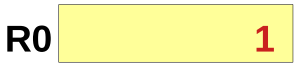
- There are several ways to enter or leave the Thumb state properly.

- The usual method is via the Branch and Exchange (**BX**) instruction.
- also Branch, Link, and Exchange (**BLX**) if you're using an ARM with version 5 architecture.

- During the branch, the CPU examines the least significant bit (lsb) of the destination address to determine the *new* state.



BX R0 ; to ARM state
BLX R0 ; to ARM state



BX R0 ; to Thumb state
BLX R0 ; to Thumb state

https://www.embedded.com/introduction-to-arm-thumb/

Switching the state (2) Exception Handler

- When an **exception** occurs, the processor automatically begins executing in ARM state at the address of the exception vector.
- So another way to change state is to place your 32-bit code in an **exception handler**.
- If the CPU is running in Thumb state when that exception occurs, you can count on it being in ARM state within the handler.
- If desired, you can have the exception handler put the CPU into Thumb state via a branch.

https://www.embedded.com/introduction-to-arm-thumb/

Switching the state (3) **T** bit in the **SPSR**

The final way to change the state is
via a **return** from **exception**.

- When returning from the processor's exception mode,
  the saved value of T in the **SPSR** register is used
  to restore the state.

- This T bit can be used, for example,
  by an operating system
  to manually restart a task in the Thumb state –
  if that's how it was running previously.

https://www.embedded.com/introduction-to-arm-thumb/

Branch and Exchange (1)

- the Branch and Exchange (**BX**) instruction.
- also Branch, Link, and Exchange (**BLX**)
  if you're using an ARM with version 5 architecture.

- During the branch, the CPU examines
  the least significant bit (lsb) of the destination address
  to determine the *new state*.

• **B** {cond} label • ~~B {cond} Rm~~	• **BL** {cond} label • ~~BL {cond} Rm~~
• ~~BX {cond} label~~ • **BX** {cond} Rm	• **BLX** {cond} label • **BLX** {cond} Rm

BX **R0** ; to ARM state
BLX R0 ; to ARM state

R0 0

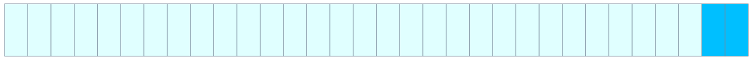
BX **R0** ; to Thumb state
BLX R0 ; to Thumb state

R0 1

with label
always changes the state.
ARM state → Thumb state
Thumb state → ARM state

with Rm
Rm[0] = **0** → to ARM state
Rm[0] = **1** → to Thumb state

address of a 32-bit word in **Rm**



not used

https://www.embedded.com/introduction-to-arm-thumb/

Branch and Exchange (2)

- Since all ARM instructions will align themselves on either a 32- or 16-bit boundary, the lsb of the address is not used in the branch directly.
- if the lsb is 1 when branching from ARM state, the processor switches to Thumb state before it begins executing from the new address;
- if the lsb is 0 when branching from Thumb state, the processor switches back to ARM state it goes.

• **B** {cond} label • ~~B {cond} Rm~~	• **BL** {cond} label • ~~BL {cond} Rm~~
• ~~BX {cond} label~~ • **BX** {cond} Rm ⬅	• **BLX** {cond} label ⬅ • **BLX** {cond} Rm ⬅

BX Rm ⬅
BLX Rm ⬅
; destination address in the regsiter Rm
If **Rm**[0] is 0, to ARM state.
If **Rm**[0] is 1, to Thumb state.

BLX *lable* ⬅
; destination address is the PC-relative *lable* expression
always change: (ARM → Thumb, Thumb → ARM)

with label ⬅
always changes the state.
ARM state → Thumb state
Thumb state → ARM state

with Rm ⬅
Rm[0] = **0** → to ARM state
Rm[0] = **1** → to Thumb state

https://www.embedded.com/introduction-to-arm-thumb/

Entering and leaving the Thumb state (1)

- several ways to enter or leave the Thumb state properly.
- the usual method is via the **BX** (**B**ranch and E**X**change) instruction.
- also **BLX** (**B**ranch, **L**ink, and E**X**change) with version 5 architecture.
- during the branch, the CPU examines
  the lsb of the destination address in a register operand
  to determine the new state.
-

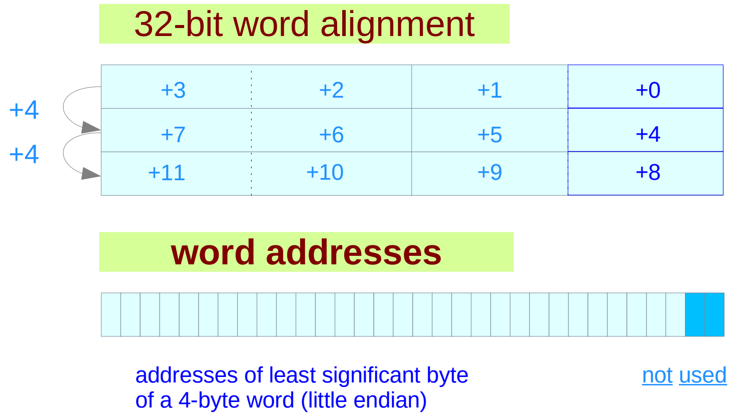
- **BX** {cond} Rm
- **BLX** {cond} Rm

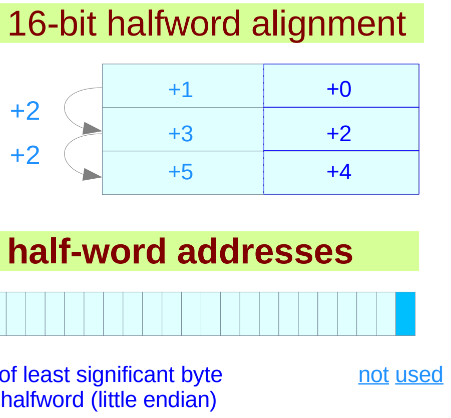
with Rm
Rm[0] = **0** → to ARM state
Rm[0] = **1** → to Thumb state

https://community.arm.com/developer/ip-products/processors/f/cortex-a-forum/5655/question-about-a-code-snippet-on-arm-thumb-state-change

Entering and leaving the Thumb state (2)

- all ARM instructions will align themselves on either a 32- or 16-bit boundary →
- the lsb of the destination address is not used in the branch directly.
- if the lsb is 1 when branching from ARM state, the processor switches to Thumb state before it begins executing from the new address;
- if the lsb is 0 when branching from Thumb state, back to ARM state it goes.




16-bit halfword alignment
+2
+2
+1
+0
+3
+2
+5
+4
half-word addresses
addresses of least significant byte
of a 2-byte halfword (little endian)
not used

https://community.arm.com/developer/ip-products/processors/f/cortex-a-forum/5655/question-about-a-code-snippet-on-arm-thumb-state-change

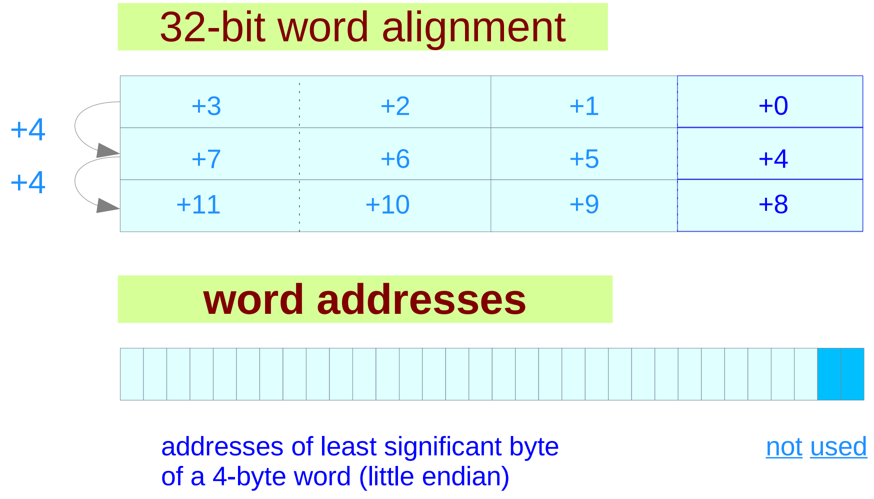
32-bit / 16-bit alignment

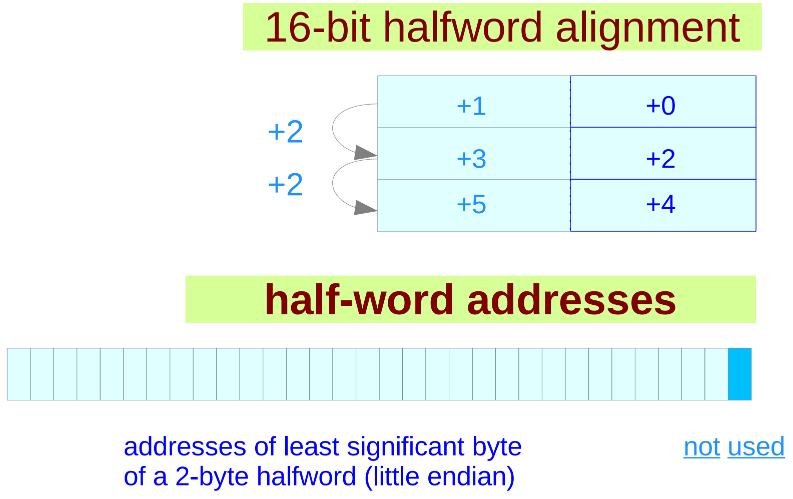
Since all ARM instructions have
either a 32- or 16-bit alignment

the LSB of the address is not used in the branch directly.

32-bit (4 bytes) word - the least significant 2 bits of the target address are not used
16-bit (2 bytes) word - the least significatn 1 bit of the target address is not used

can use the the least significant bit is used to change the state (ARM ↔ Thumb)




16-bit halfword alignment
+2
+2
+1
+0
+3
+2
+5
+4
half-word addresses
addresses of least significant byte of a 2-byte halfword (little endian)
not used

https://www.cs.princeton.edu/courses/archive/fall13/cos375/ARMthumb.pdf

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

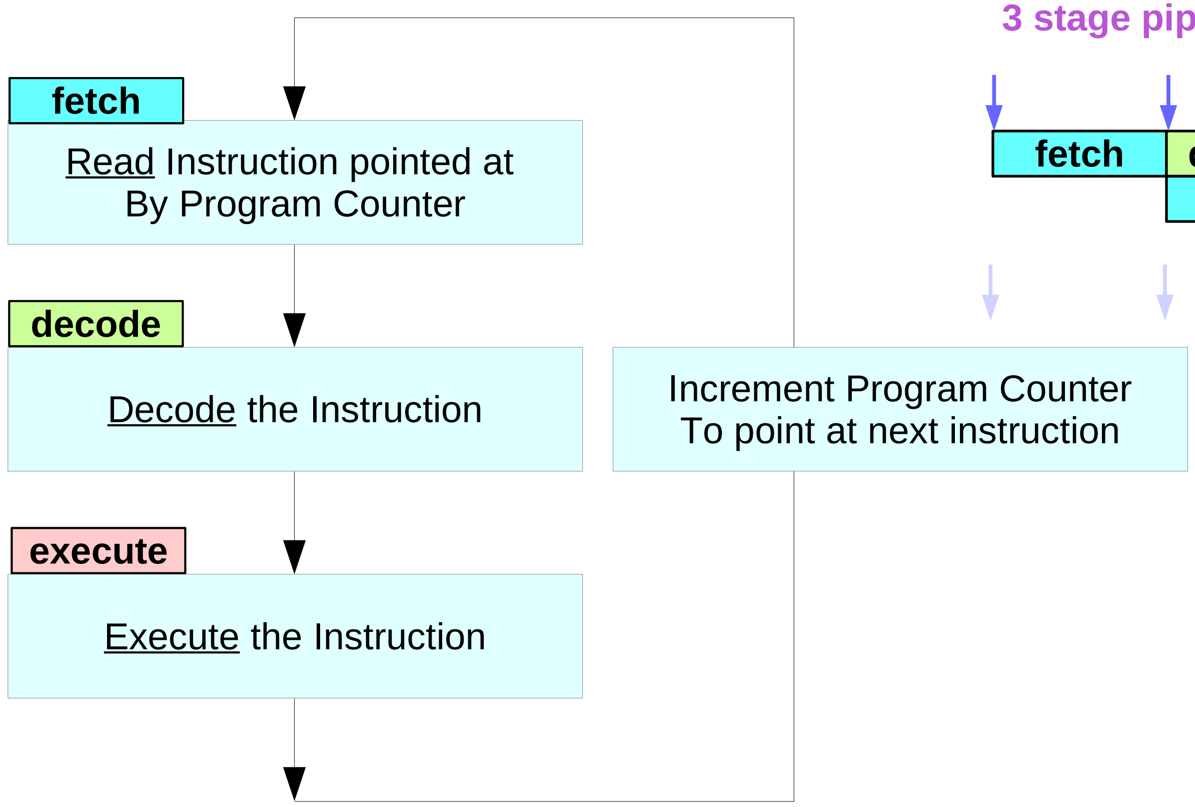
Once decoded, the instruction can then be executed
and the PC incremented so that it contains
the address of the next instruction.

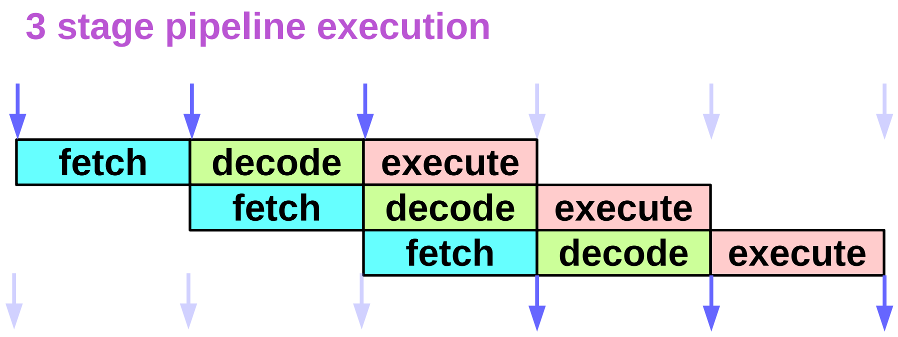
execute

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

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

PC and 3-stage pipeline (1)


fetch
Read Instruction pointed at By Program Counter
decode
Decode the Instruction
execute
Execute the Instruction
Increment Program Counter To point at next instruction

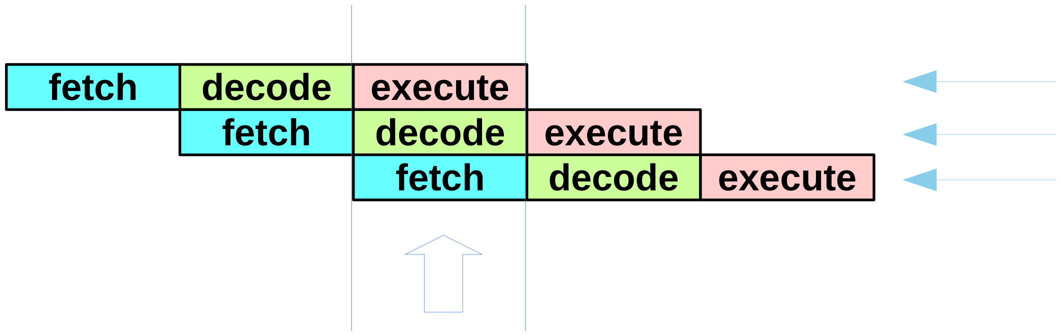


Execute a current instruction
Decode the next instruction
Fetch the next's next instruction

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

PC and 3-stage pipeline (2)

3 stage pipeline execution



the current instruction is being executed
the next instruction is being decoded
the next's next instruction is being fetched

the current
time slot

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

Incrementing **PC** (1)

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

- 4 byte size in ARM state
- 2 byte size in Thumb state.

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.

Thus the content of PC

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

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

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

Incrementing **PC** (2)

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

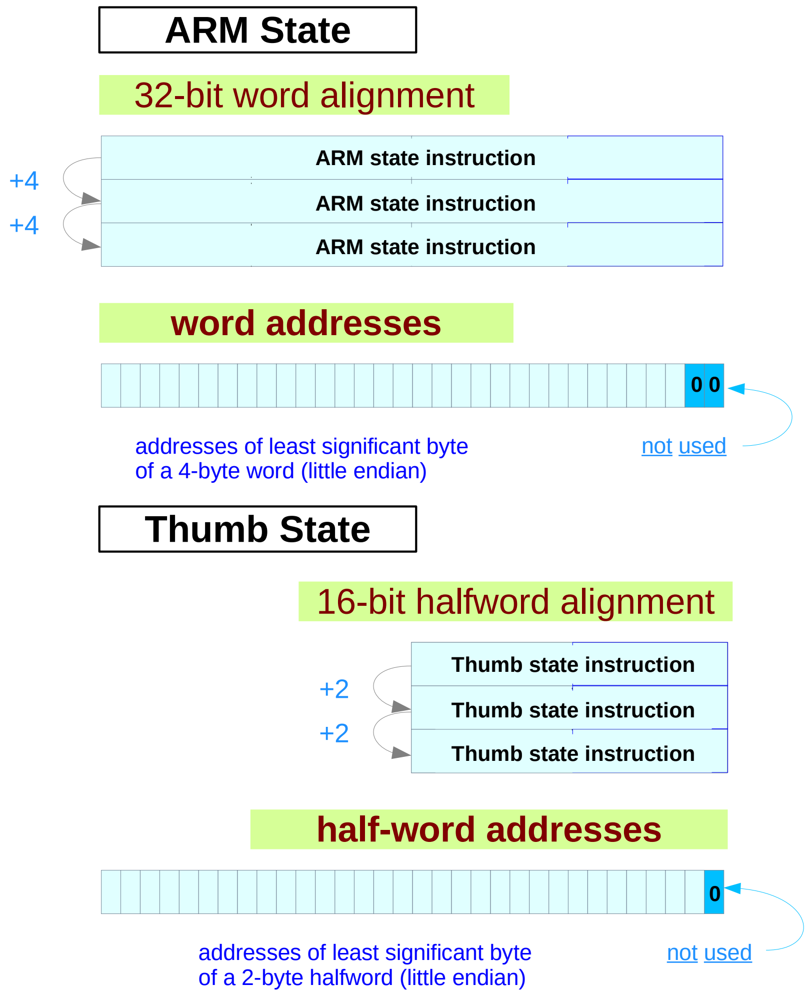
but also can be accessed by a byte or a halfword
with a performance loss

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

The **PC** values in ARM and Thumb states

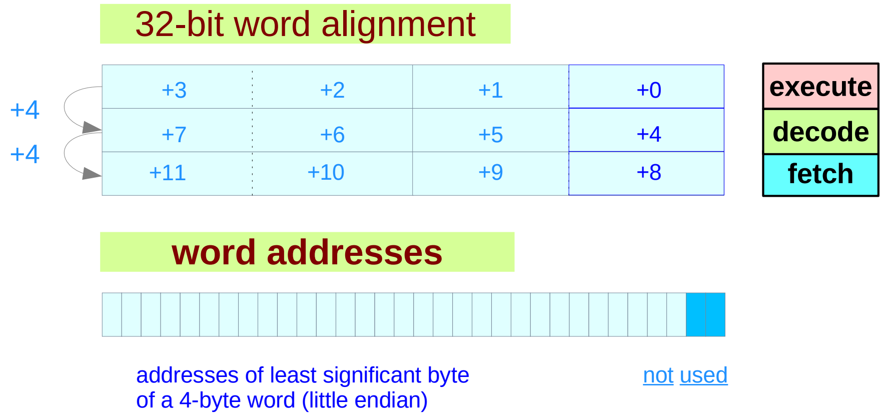
In **A32** code, PC + 8
the value of the PC is
the address of the current instruction plus 8 bytes.

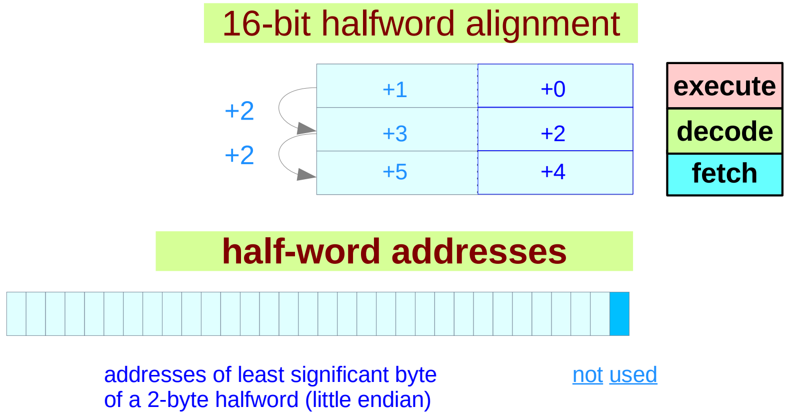
In **T32** code: PC + 4
the value of the PC is

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.





16-bit halfword alignment
+1
+0
+3
+2
+5
+4
+2
+2
execute
decode
fetch
half-word addresses
addresses of least significant byte
of a 2-byte halfword (little endian)
not used

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

The **PC** values in Thumb states (1)

In hardware, PC in Thumb state can point any halfword

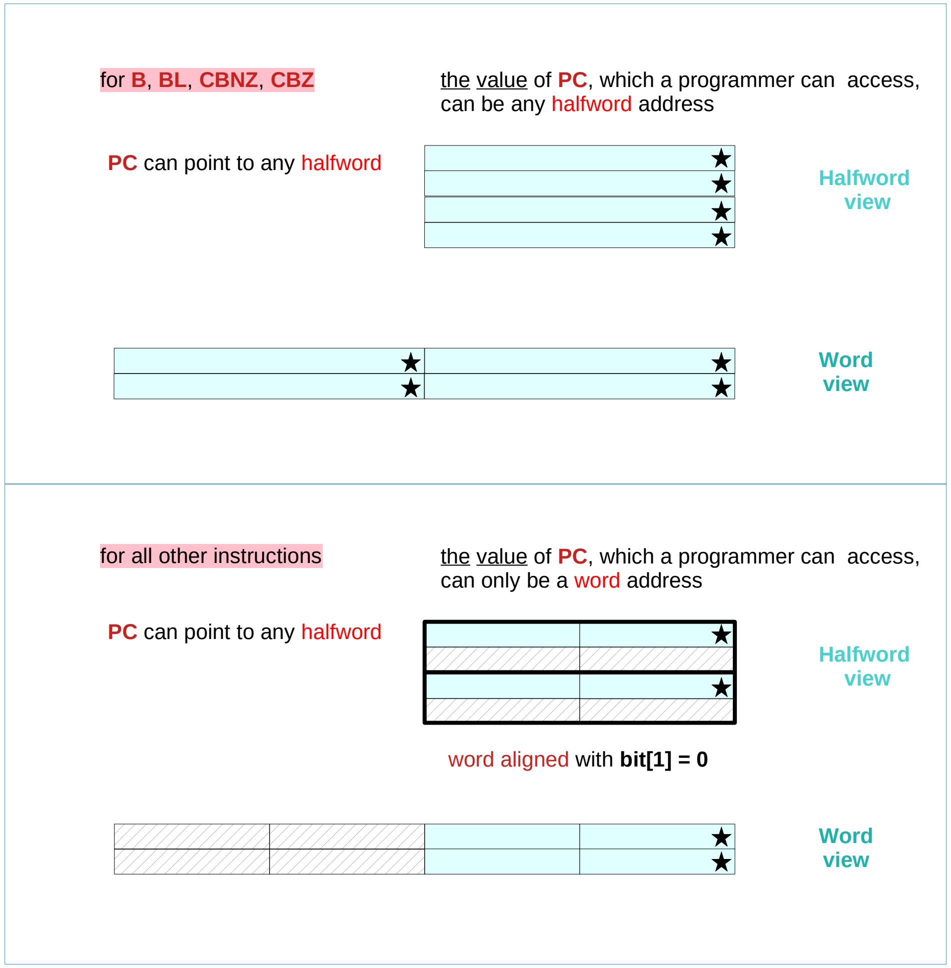
when a programmer access PC in Thumb state,
during the execution stage of the pipeline,
its value can be

- (PC + 4)

  any halfword address of the next's next instruction
  for **B**, **BL**, **CBNZ**, **CBZ** instructions

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

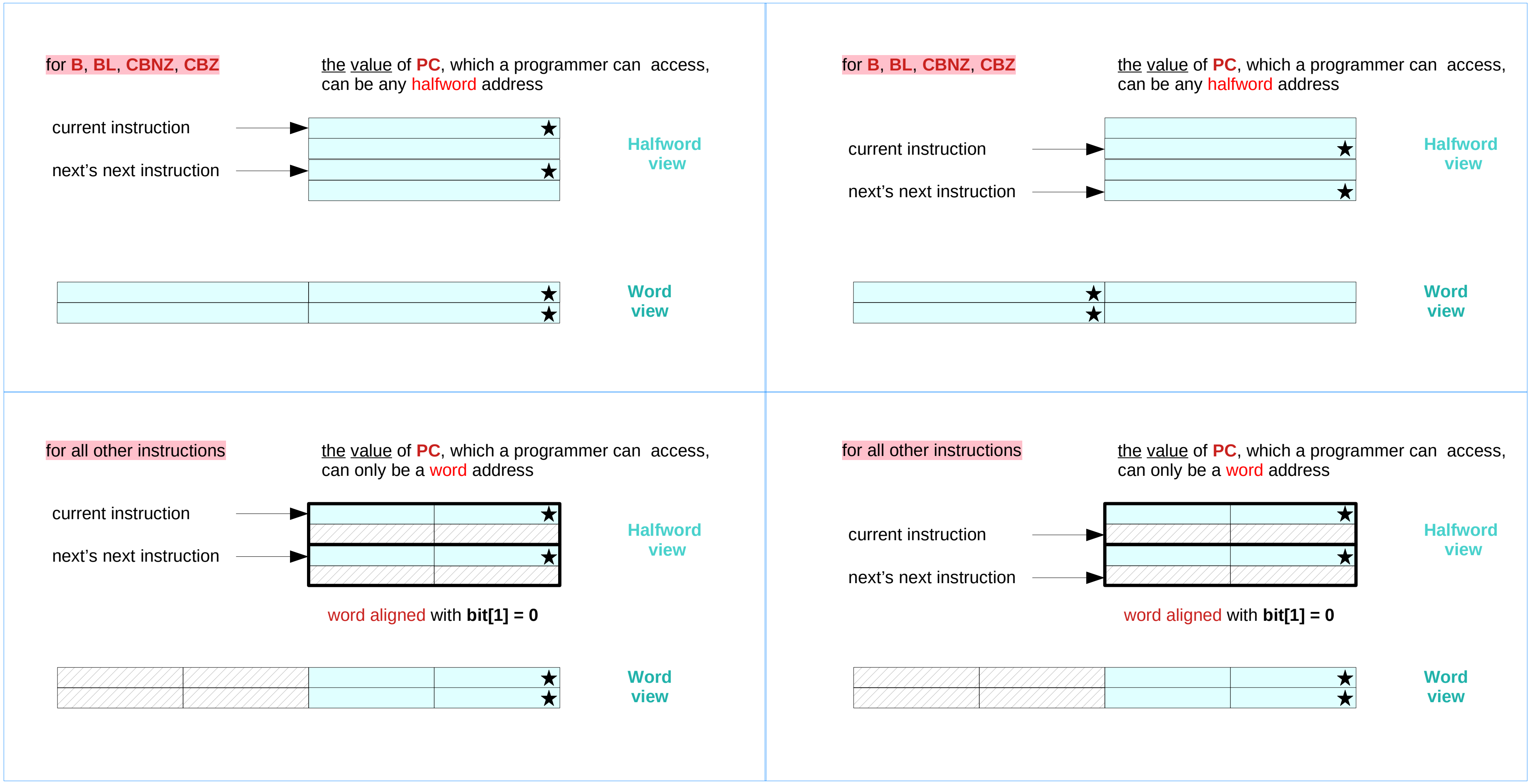
  only a word address of the next's next instruction
  for all other instructions



for **B**, **BL**, **CBNZ**, **CBZ**
the value of **PC**, which a programmer can access, can be any halfword address
PC can point to any halfword
Halfword view
Word view
for all other instructions
the value of **PC**, which a programmer can access, can only be a word address
PC can point to any halfword
Halfword view
word aligned with **bit[1] = 0**
Word view

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

The **PC** values in Thumb states (2)


for B, BL, CBNZ, CBZ
the value of PC, which a programmer can access, can be any halfword address
current instruction
next's next instruction
Halfword view
Word view
for B, BL, CBNZ, CBZ
the value of PC, which a programmer can access, can be any halfword address
current instruction
next's next instruction
Halfword view
Word view
for all other instructions
the value of PC, which a programmer can access, can only be a word address
current instruction
next's next instruction
Halfword view
word aligned with bit[1] = 0
Word view
for all other instructions
the value of PC, which a programmer can access, can only be a word address
current instruction
next's next instruction
Halfword view
word aligned with bit[1] = 0
Word view

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

Return from a procedure (1)

ARM is unusual among the processors
by having the program counter available
as a "general purpose" register.

Most other processors have the program counter hidden,
and its value will only be disclosed as the return address
when calling a function.

If you want to modify it, a jumping instruction is used.

For example, on the **x86**, the program counter
is called the instruction pointer,
and is stored in **eip**,
which is not an accessible register.

After a function call, **eip** is pushed onto the stack,
at which point it could be examined.

Return is done through the **ret** instruction
which pops the return address off the stack,
and jumps there.

Another example: on the **MIPS**,
the program counter is stored into **register 31**
after executing a **JALR** instruction,
which is used for function calling.

The value in there can be examined,
and a return is a register jump **JR** to that register.

https://quantum5.ca/2017/10/19/arm-ways-to-return/

Return from a procedure (2)

ARM's unusual design allows many,
many ways of returning from functions.

But first, we must understand
how function calls work on the ARM.

On ARM, the program counter is register 15,
or **r15**, also called **pc**.

The instruction to call a function is
bl (for immediate offsets, a label operand)
or **blx** (for addresses in registers, a register operand).

These instructions stores the return address
in **r14**, called the link register, or **lr**.

To return, we must put this value back
into **pc**.

B,	**BL,**
BX,	**BLX**

BL and **BLX** copy
the return address
into **LR** (**R14**)

LR (**R14**) ⬅ return address

https://quantum5.ca/2017/10/19/arm-ways-to-return/

Return from a procedure (3)

Method 1

When writing <u>non-leaf</u> functions, i.e.
functions that calls other functions,
the value of lr must be <u>preserved</u>,
since <u>calling</u> <u>another</u> function will overwrite it.

The most common way is
to store it on the stack.

On the ARM, **push** and **pop** instructions

use **push** and **pop** to <u>preserve</u>
the registers we modify.

For example, if we want
to <u>preserve</u> **r3**, **r4**, and **lr**,
we can write **push** {**r3**, **r4**, **lr**}.

A normal function will look like:

```
push      {r3, r4, lr}   ; save registers.

; function body.

pop       {r3, r4, pc}   ; restore registers
                         ; and return (pc←lr)
```

https://quantum5.ca/2017/10/19/arm-ways-to-return/

use **push** and **pop** to restore all the registers,
except putting what was **lr**
when we are doing **pop** into **pc**.

This will overwrite **pc** with the return address,
achieving the return.

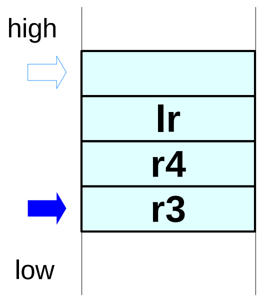
Note that we could instead use **r14**
instead of **lr** and **r15** instead of **pc**,
but this is less clear on the intent.

push {r3, r4, lr}

pop {r3, r4, pc}

equivalent
PUSH
sequence

1 **PUSH LR**
2 **PUSH R4**
3 **PUSH R3**



equivalent
POP
sequence

1 **POP R3**
2 **POP R4**
3 **POP PC**

Return from a procedure (4)

```
;;; Define Procedures

function1   PROC            ; using PROC and ENDP for procedures
        PUSH {R5, LR}       ; save values in the stack

        MOV   R5,#8         ;set initial value for the delay loop

delay
        SUBS R5, R5, #1
        BNE   delay

        POP   {R5, PC}      ;pop out the saved value from the stack,
                            ;check the value in the R5 and
                            ;if it is the saved value

        ENDP


;;   ----------------

MOV   R5, #9                ;; prepare for function call


        BL     function1
```

https://www.labs.cs.uregina.ca/301/ARM-subroutine/lecture.html

Return from a procedure (5)

Method 2

use an unconditional jump to register to return

useful in leaf functions
where **lr** needs not to be stored on the stack.

```
bx        lr
```

this jumps to the address in **lr**,
setting **pc** to **lr**, and completing the return.

Method 3

Similar in rationale to method 2,
ARM lets you manipulate the program counter
as you would any other register.

```
mov       pc, lr
```

This copies **lr** into **pc**, also completing the return.

https://quantum5.ca/2017/10/19/arm-ways-to-return/

Return from a procedure (6)

Method n

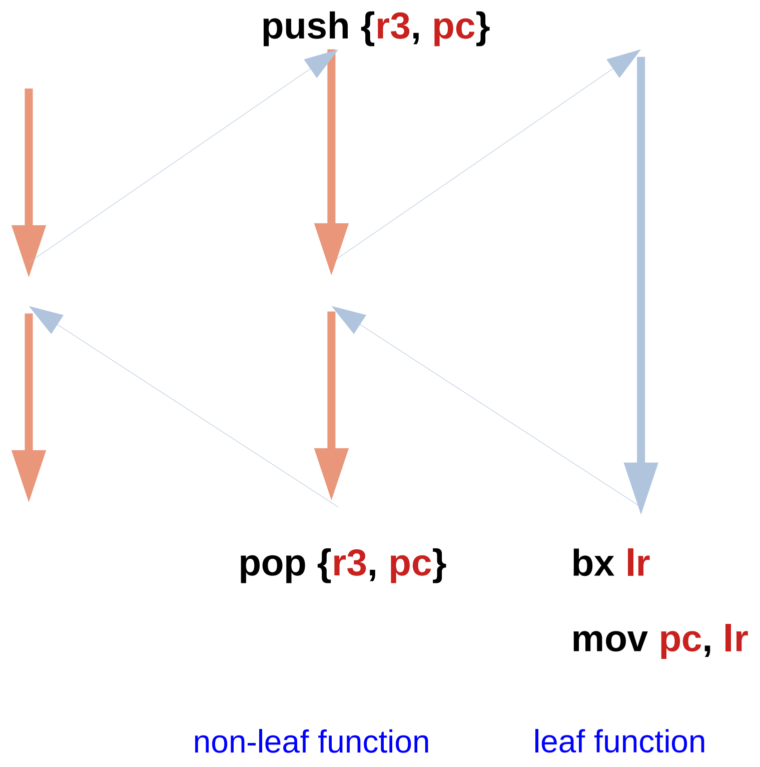
many other ways of copying the value
from one register into another

but as long as **lr** at the beginning of the function call
is goplaced into **pc**, a return is completed.

But please, use the most sensible ways to return.

This means you should prefer the first two,
depending on whether the function is a leaf.

As a distant third, use method 3 (**mov pc, lr**).



https://quantum5.ca/2017/10/19/arm-ways-to-return/

Return from a procedure (6)

R13 (**SP**)
R14 (**LR**)
R15 (**PC**)

subroutines in an ARM assembler code in BASIC.
The subroutines are said to be ended with

MOVS R15,R14 MOVS PC,LR

call for each subroutine starting with the label:

.some_subroutine

But when I want to end the routine
that calls subroutines within it,
if I use:

MOV R15,R14 MOV PC,LR

the program tends to hang rather than
to exit from that routine.

OS_Exit cannot be used
because it brings the entire program
to an abrupt conclusion instead of
continuing to the next BASIC line.

https://www.riscosopen.org/forum/forums/11/topics/3986`

MOVS instruction

MOV{S}	**Rd, Rm**
MOVS	**Rd, #imm**
MVNS	**Rd, Rm**

S	is an optional suffix. If **S** is specified, the condition code flags are updated on the result of the operation
Rd	is the destination register.
Rm	is a register.
Imm	is any value in the range 0-255.

The **MOV** instruction copies the value of **Rm** into **Rd**.

The **MOVS** instruction performs the same operation
as the **MOV** instruction, but also updates the **N** and **Z** flags.

The **MVNS** instruction takes the value of **Rm**,
performs a bitwise logical negate operation on the value,
and places the result into **Rd**.

If **S** is specified, these instructions:

update the **N** and **Z** flags according to the result

do not affect the **C** or **V** flags.

https://developer.arm.com/documentation/ddi0597/2024-12/Base-Instructions/MOV--MOVS--immediate---Move--immediate--

MOVS instruction

MOVS Rd, #imm

Move (immediate) writes an immediate value
to the destination register.

If the destination register is not the **PC**,
the **MOVS** variant of the instruction
updates the condition flags based on the result.

The field descriptions for <**Rd**> identify the encodings
where the **PC** is permitted as the destination register.

Arm *deprecates* any use of these encodings.

However, when the destination register is the **PC**:

1. MOV PC, #imm

The **MOV** variant of the instruction is an interworking branch,

see Pseudocode description of operations
on the AArch32 general-purpose registers and the PC.

2. MOVS PC, #imm

The **MOVS** variant of the instruction
performs an exception return
without the use of the stack.

In this case:

The PE branches to the address written to the **PC**,
and restores PSTATE from SPSR_<current_mode>.

The PE checks SPSR_<current_mode>
for an illegal return event.

The instruction is UNDEFINED in Hyp mode.

The instruction is CONSTRAINED UNPREDICTABLE
in User mode and System mode.

* PE (Processing Element)

https://developer.arm.com/documentation/ddi0597/2024-12/Base-Instructions/MOV--MOVS--immediate---Move--immediate--

Branches and Interworking (1)

When using Thumb, the system will typically have both ARM and Thumb functions.

Even if you compile your application for Thumb, you might still have to think about *such things* as libraries and prebuilt binaries.

The core must know which instruction set is to be used for the code being executed after a branch, procedure call or return.

This interworking between instruction sets

When writing C code, the linker takes care of this for us, but a little more care is required when porting assembly code.

The target instruction set state is determined in different ways depending on the type of branch.

We can consider a number of different instructions:

- **Function Return**

  MOV PC, LR

  BX LR

- **Function Return from the Stack**

  LDMFD SP!, {registers, pc}
  POP {<registers>, pc}

- **Branch**

  B
  BL

- **PC modification**

  MOV PC, register

  BX register

- **Function Call to Register address**

  MOV LR, PC
  MOV PC, register

  BLX <register>

https://developer.arm.com/documentation/den0013/d/Porting/Porting-ARM-code-to-Thumb/Branches-and-interworking

Branches and Interworking (2)

Function return

Legacy code might use the **MOV PC, LR** instruction.

This is unsafe for systems that contain a mix of ARM and Thumb code and must be replaced by **BX LR** for code running on all later architectures.

Function return from the stack

This is done using the **LDMFD SP!, {registers, pc}** instruction that will continue to work correctly in the ARMv7-A architecture, although a newer, equivalent form, **POP {<registers>, pc}** is also available.

This is used when registers that must be preserved by the function are PUSHed at the start of the function.

Branch

A simple **B** instruction will work in the same fashion on all ARM architectures.

If ARM and Thumb instructions are mixed in a single source file (this is unusual), there is no automatic instruction set switch for local symbols.

The assembler might introduce a veneer depending on whether it knows that the destination is in a different instruction set and is definitely a code symbol (such as a .type <symbol>, %function or .thumb_func).

Because a symbol appears in a code section it is not assumed to be a code symbol unless specifically tagged in this way.

If the label is in a different file, the linker will take care of any necessary instruction set change.

Similar considerations apply for a function call (**BL**).

https://developer.arm.com/documentation/den0013/d/Porting/Porting-ARM-code-to-Thumb/Branches-and-interworking

Branches and Interworking (3)

PC modification

Care might be required with other instructions
that modify the PC
and produce a branch effect.

For example, **MOV PC, register**
must be replaced with **BX register**
in systems that contain both ARM and Thumb code.

Function call to register address

If code contains a sequence like **MOV LR, PC**
followed by **MOV PC, register**,

this will not work in a system that has both ARM and Thumb code.
You must replace it with the single instruction **BLX <register>**.

https://developer.arm.com/documentation/den0013/d/Porting/Porting-ARM-code-to-Thumb/Branches-and-interworking

Branches and Interworking (4)

When a destination or return address is variable
or calculated at run-time, take care to appropriately set
the Thumb bit (bit [0]) in the address correctly

and to use the correct type of branch,
to make sure that the call (and return, if applicable)
switches instruction set appropriately.

If an external label or function defined in another object
is referenced, the linker will produce an address
with the Thumb bit (bit [0]) set appropriately.

However, if you reference a symbol internal to the object,
things are more complicated.

For C functions, or code tagged as Thumb, bit [0] will be
set appropriately, but it will not be set appropriately
for other symbols.

In particular, GNU Assembler local labels will not
have the Thumb bit set appropriately,
nor will the GNU current assembly location symbol ".".

Therefore, when coding in assembler,
if an address will be passed
to any other function or object,

for example,
as a return address, method address or callback,
you must handle the Thumb bit setting yourself,
setting bit [0] of the address where required.

https://developer.arm.com/documentation/den0013/d/Porting/Porting-ARM-code-to-Thumb/Branches-and-interworking

Interworking (1)

When the core executes ARM instructions,
it is said to be operating in ARM state.

When it is operating in Thumb state,
it is executing Thumb instructions.

A core in a particular state can only execute instructions
from that instruction set.

We must make sure that the core does not receive instructions
of the wrong instruction set.

Each instruction set includes instructions
to change processor state.

ARM and Thumb code can be mixed,
if the code conforms to the requirements of the ARM
and Thumb Procedure Call Standards.

Compiler generated code will always do so,
but assembly language programmers
must take care to follow the specified rules.

https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/Interworking

Interworking (2)

Selection of processor state is controlled
by the T bit in the CPSR.

When T is 1, the processor is in Thumb state.
When T is 0, the processor is in ARM state.

However, when the T bit is modified,
it is also necessary to flush the instruction pipeline
(to avoid problems with instructions
being decoded in one state
and then executed in another).

Special instructions are used to accomplish this.

These are BX (Branch with eXchange) and
BLX (Branch and Link with eXchange).
LDR of PC and POP/LDM of PC also have this behavior.

In addition to changing the processor state
with these instructions, assembly programmers must
also use the appropriate directive
to tell the assembler to generate code
for the appropriate state.

https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/Interworking

Interworking (3)

The BX or BLX instruction branches to an address
contained in the specified register,
or an offset specified in the opcode.

The value of bit [0] of the branch target address determines
whether execution continues in ARM state or Thumb state.

Both ARM (aligned to a word boundary)
and Thumb (aligned to a halfword boundary) instructions
do not use bit [0] to form an instruction address.

This bit can therefore safely be used
to provide the additional information
about whether the BX or BLX instruction
should change the state to ARM (address bit [0] = 0)
or Thumb (address bit [0] = 1).

The BL label will be turned into a BLX label as appropriate
at link time if the instruction set of the caller
is different from the instruction set of label,
assuming that it is unconditional.

https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/Interworking

Interworking (4)

A typical use of these instructions is
when a call from one function to another is made
using the BL or BLX instruction,
and a return from that function is made
using the BX LR instruction.

Alternatively, we can have a non-leaf function
that pushes the link register onto the stack
on entry and pops the stored link register
from the stack into the program counter, on exit.

Here, instead of using the BX LR instruction to return,
we have a memory load.

Memory load instructions that modify the PC
might also change the processor state
depending on the value of bit [0] of the loaded address.

https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/Interworking

Return from a procedure (7)

```
.entry
  BL          myfunction
  MOV          PC, R14

.myfunction
  ; does nothing
  MOV         PC, R14
```

This will fail because the exit address is in **R14** on entry,
and the **BL** call trashes that, so your program cannot ever exit
as the return address is gone.

https://www.riscosopen.org/forum/forums/11/topics/3986

Return from a procedure (8)

Consider:

```
.entry                  ; entry point, return address in R14
  BL   myfunction       ; call subroutine (puts return address in R14)
  MOV  PC, R14          ; return to BASIC (R14 will come back here)

.myfunction
  ; does nothing
  MOV  PC, R14          ; exit subroutine by jumping back to R14
```

If you follow through this code, you'll see that the line
that it supposed to return to BASIC is the instruction following the **BL**,
which means **R14** will point to it,
so it'll just keep jumping to itself <cue spooky voice>forever!!!!!

https://www.riscosopen.org/forum/forums/11/topics/3986

Return from a procedure (9)

How to fix this? You need to preserve **R14** prior to it being used again.
Like this (assuming **R13** is a valid stack, it is from BASIC):

```
.entry
  STR  R14, [R13, #-4]!        ; stack R14
  BL   myfunction
  LDR  PC, [R13], #4           ; unstack R14 directly into PC to exit

.myfunction
  ; does nothing
  MOV  PC, R14
```

The weird looking offsets are to write-back **R13**
to support a full descending stack.

The STR’s “#-4]!” performs a decrement before action
(akin to the behaviour of STMFD/STMDB),
while the LDR’s “], #4” performs an increment after action
(akin to LDMFD/LDMIA).

This supports the type of stack used within RISC OS.

https://www.riscosopen.org/forum/forums/11/topics/3986

Return from a procedure (10)

You might have come across it like this,
but these days it is inefficient to use a multiple register instruction
to store and load single registers (and indeed, ARM64 doesn't support STM/LDM at all!).

This is for information purposes as you will probably come across code
that does this. It's inefficient, so try to remember the STR/LDR version given above…

```
.entry
  STMFD R13!, {R14}
  BL    myfunction
  LDMFD R13!, {PC}

.myfunction
  ; does nothing
  MOV   PC, R14
```

https://www.riscosopen.org/forum/forums/11/topics/3986

Return from a procedure (6)

Certainly, PUSH and POP,
which are the preferred forms now I think,
are used with single registers.

If your assembler supports PUSH and POP
then it should automatically switch
between LDM/STM and LDR/STR
depending on whether one or many registers
need to be transferred
(they are basically just aliases
for the corresponding load/store instruction).

Of course the downside is you can’t specify
the base register or whether writeback is used,
so they’re only good for basic stack interactions.

I seem to remember at one time
being told that STM is decomposed into separate STR instructions
in more recent ARM versions.

Return from a procedure (6)

Possibly you’re thinking of AArch64 – there is no LDM/STM,
only the single register LDR/STR and double register LDP/STP.

Or maybe you’re thinking about the fact
that modern ARMs allow interrupts to occur
in the middle of an LDM/STM –
if this happens then the ARM will point the exception return address
at the LDM/STM so that the entire instruction is restarted
once the interrupt has been dealt with
(it also makes sure the base address register hasn’t been updated,
but the state of the other registers/memory locations
used by the instruction are undefined
– so LDM/STM is no longer atomic from a single-CPU perspective)

Return from a procedure (11)

That’s what Entry and EXIT (and friends) are for:

Entry → push a stack frame for procedure entry
(implicitly adds lr to the register list),
optionally reserving a block of local workspace on the stack

EXIT → return from a procedure by popping the workspace + register list
from the most recent Entry
(i.e. the one located directly before it in the assembler listing)

EntryS/EXITS → variants which save and restore some or all of the PSR

EXITV/EXITVC/EXITVS → return with **V** flag in a specified state

PullEnv/PullEnvS → pop the stack frame without returning from the procedure

ALTENTRY → generate an Entry/EntryS equivalent to the most recent
(used when shared code can have multiple entry points)

FRAMLDR/FRAMSTR → load/store specific registers from the stack frame
(calculates the correct offset, assuming you haven’t used Push/Pull
or adjusted SP manually)

If you’re observant you’ll also spot that there’s an ENTRY macro which is equivalent to Entry,
but that one isn’t used any more because objasm confuses it with the ENTRY directive.

https://www.riscosopen.org/forum/forums/11/topics/3986

Subroutine call (1) **BL** (Branch and link) operation

Both the ARM and Thumb instruction sets contain
a primitive subroutine call instruction, **BL** target,
which performs a branch-with-link operation.

LR ← the return address
the next value of the **PC**

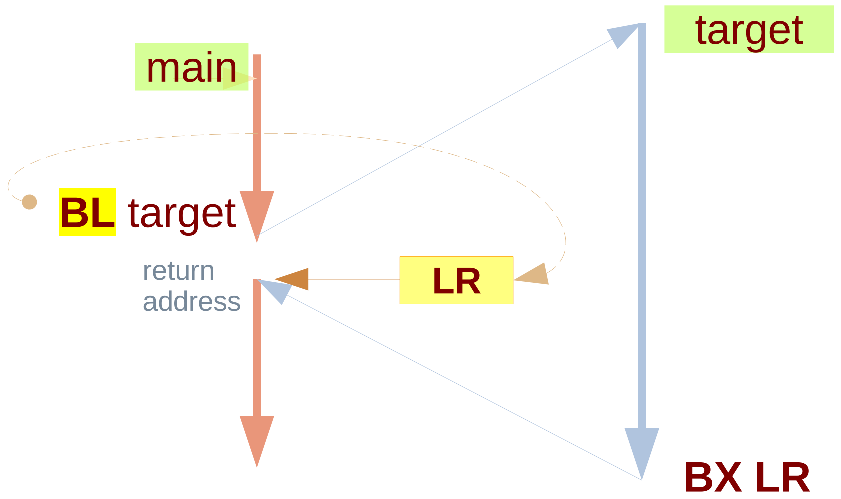
PC ← the destination address target

LR[0] ← **1** if **BL** target was executed from Thumb state
LR[0] ← **0** if **BL** target was executed from ARM state

The result is to transfer control
to the destination address,
passing the return address in **LR**
as an additional parameter
to the called subroutine

Control is returned to the instruction following the **BL**
when the return address is loaded back into the PC

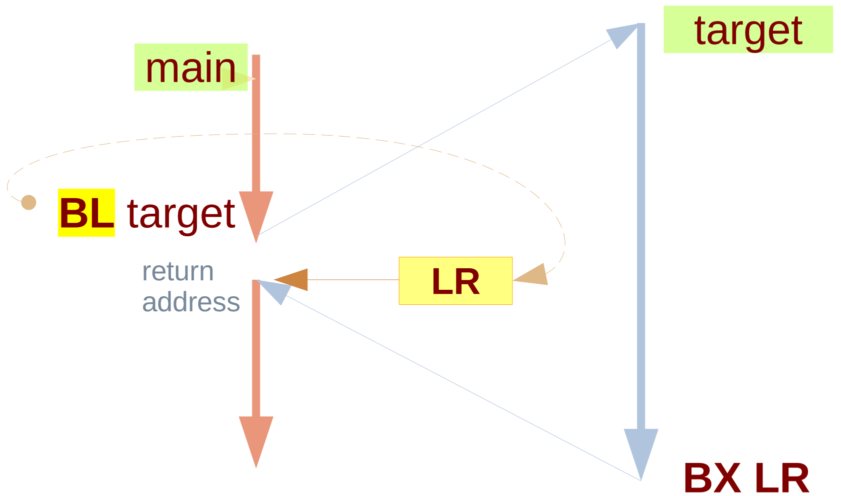
/IHI0042E_aapcs.pdf

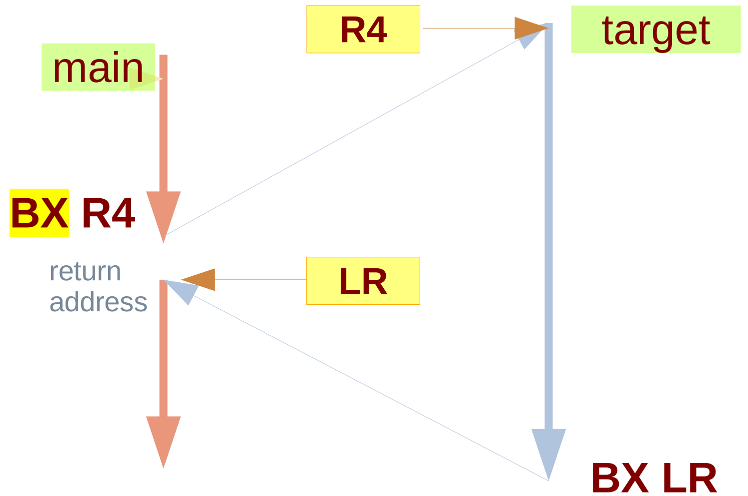


"BL target" in Thumb state
then assign **LR[0] = 1**
(to return to Thumb code)

"BL target" in ARM state
then assign **LR[0] = 0**
(to return to ARM code)

Subroutine call (2) **BL** vs. **BX**





BL target
~~**BL R4**~~

BL has
no register operand

BL sets the return
address in **LR**

~~**BX** target~~
BX R4

BX has
no label operand

a programmer must
explicitly set the return
address in **LR**

Subroutine call (3) **BX** (Branch and eXchange) operation

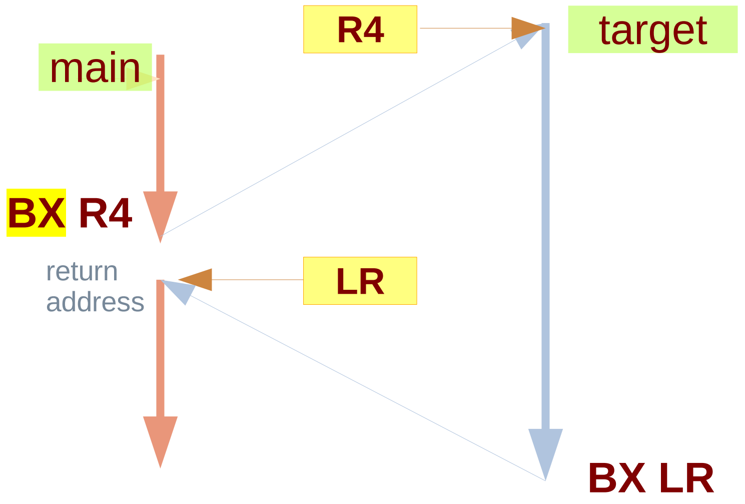
A subroutine call can be synthesized
by any instruction sequence that has the effect:

LR[31:1]	← return address	**R14 := PC+8;**
LR[0]	← code type *at* return address (0 ARM, 1 Thumb)	
PC	← subroutine address	**PC := R4;**

in ARM-state,
to call a subroutine addressed by **R4**
with control returning to the following instruction,

R4 := target+1;

```
        MOV LR, PC    ⟵   R14 := PC+8;
        BX    R4      ⟵
return:               ⟵   R14[0] = 0;
```

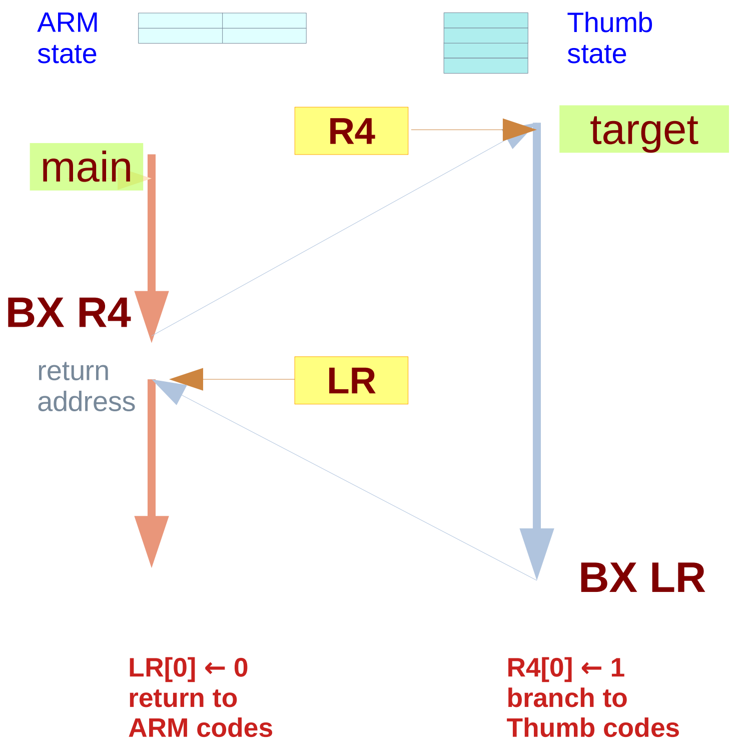


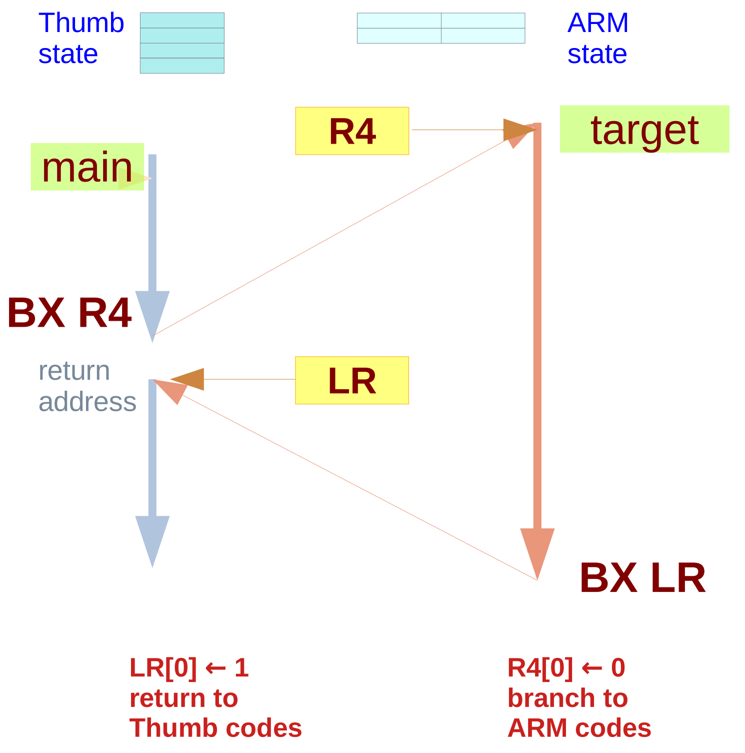
LR[31:1] ← the return address

LR[0] ← 0   return to ARM codes
LR[0] ← 1   return to Thumb codes

/IHI0042E_aapcs.pdf

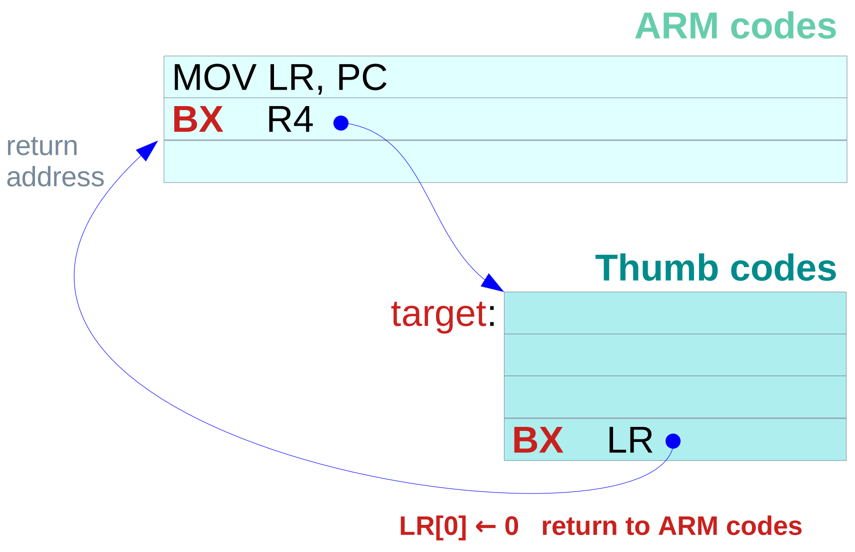
Subroutine call (4) ARM vs. Thumb state

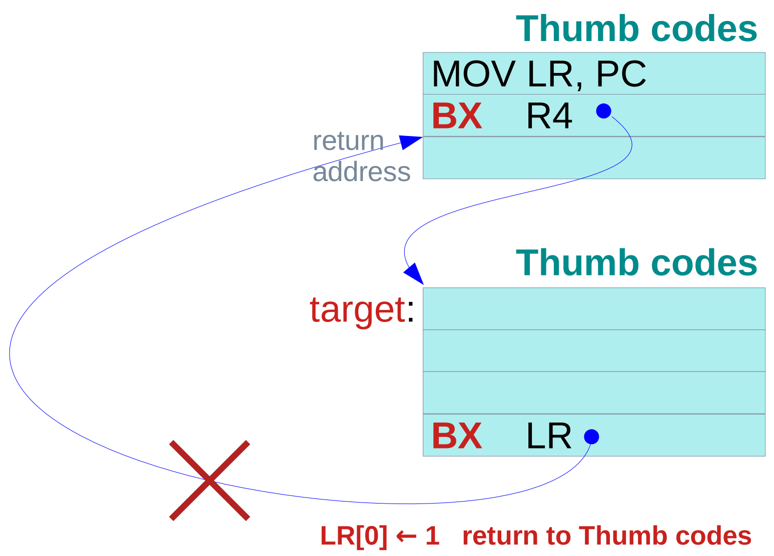




/IHI0042E_aapcs.pdf

Subroutine call (5) the lsb of a destination address





LR ← the return address
  LR[31:1] ← the return address
  LR[0] ← 0 return to ARM codes

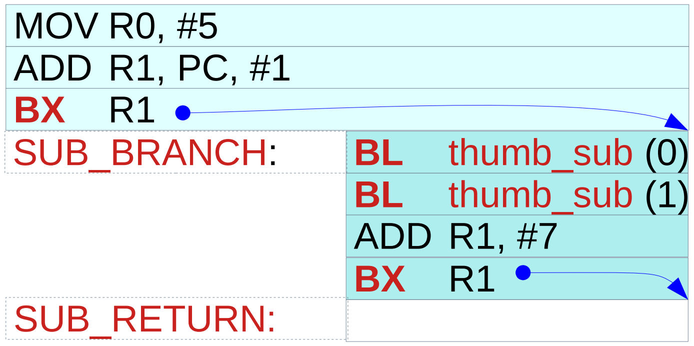
this will not work from Thumb state
because the instruction that sets **LR**
does not copy the Thumb-state bit to **LR[0]**

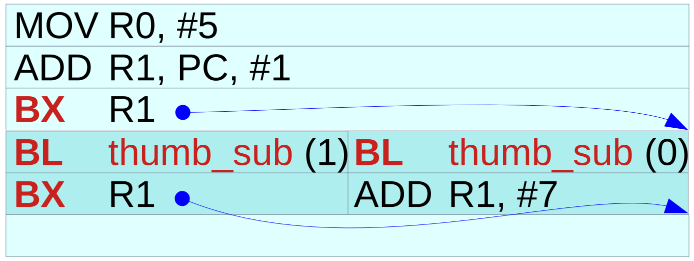
(**LR[0]** must be set to **1**)

```
        MOV LR, PC    ⬅   R14 := PC+8;
        BX  R4        ⬅
return:               ⬅   R14[0] = 0;
```

LR[0] ← 1 return to Thumb codes

State changing example (1)





In ARM mode, **PC** indicates 2 instructions ahead

PC of '**ADD R1,PC,#1**' is
the address of SUB_BRANCH

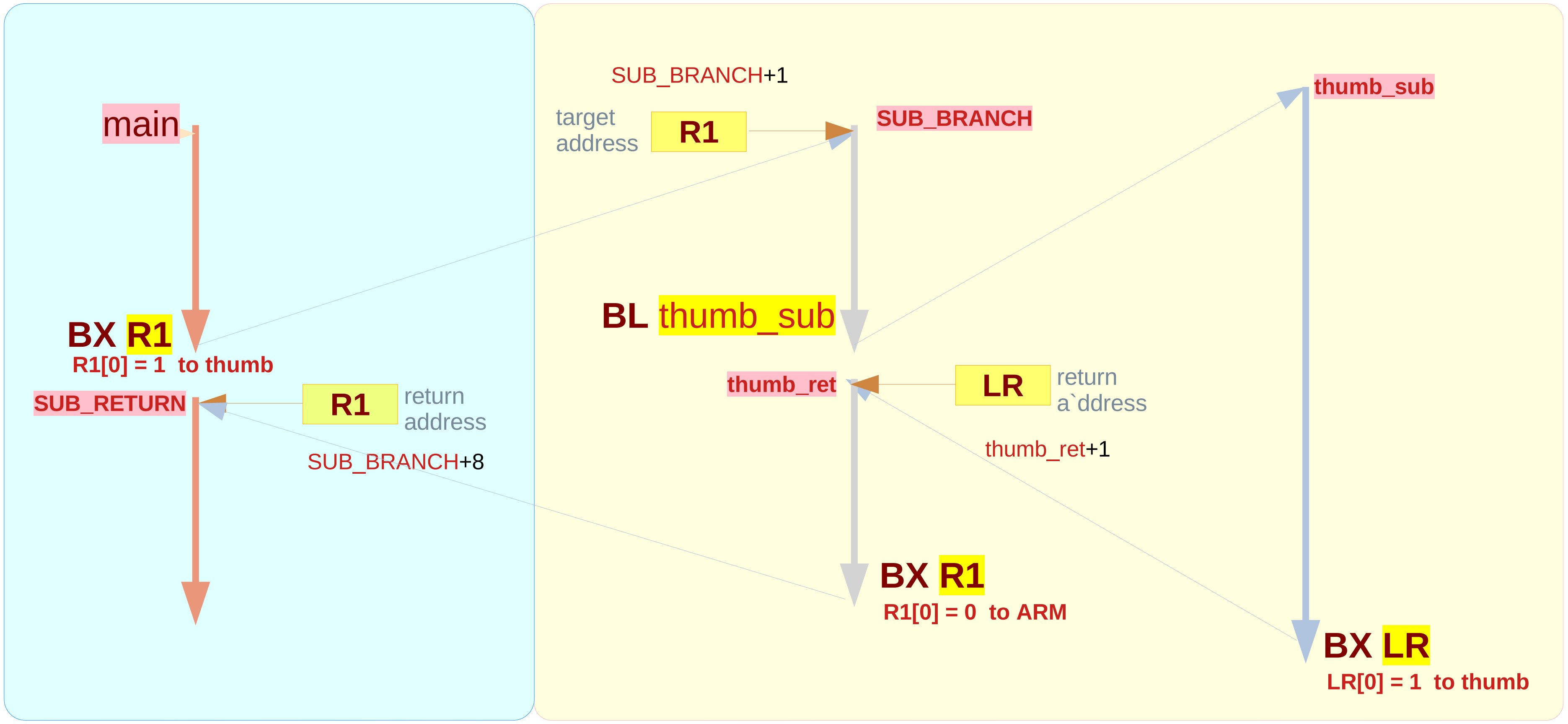
execution mode switch from **ARM** to **Thumb**
at the SUB_BRANCH and
the program will execute in **Thumb** mode.

And **R1** is now 'SUB_BRANCH+1'
and by adding to 7
it will become 'SUB_BRANCH+8'.

'SUB_BRANCH+8' is
the address of 'SUB_RETURN' and
the program jumps to the address
of which LSB value is 0 and
the execution mode will become
from **Thumb** mode to **ARM** mode.

https://community.arm.com/developer/ip-products/processors/f/cortex-a-forum/5655/question-al

Branch and link operation (2)



/IHI0042E_aapcs.pdf

Branch and Exchange (2)

ARM state ▭ Thumb state ▨

change into Thumb state, then back

```
mov  R0, #5       ; argument to function is in R0
add  R1, PC,#1    ; load address of SUB_BRANCH,
                  ; set for THUMB by adding 1
BX   R1           ; R1 contains address of SUB_BRANCH+1
                  ; assembler-specific instruction
                  ; to switch to Thumb


SUB_BRANCH:
BL   thumb_sub    ; must be in a space of +/- 4 MB
add  R1, #7       ; point to SUB_RETURN with bit 0 clear
BX   R1
; assembler-specific instruction to switch to ARM
SUB_RETURN:
```

https://www.embedded.com/introduction-to-arm-thumb/

Branch and Exchange (3)

- the **BX** instruction example to go from ARM to Thumb state and back.
- first switches to Thumb state (**BX R1**)
- **R1[0] = 1** (because of +1)
- then calls a subroutine written in Thumb code (**BL thumb_sub**)
- upon return from the subroutine (**BX R1**) the system again switches back to ARM state;
- **R1[0] = 0** (because of +1+7= +8)

```
mov  R0, #5      ; argument to function is in R0
add  R1, PC,#1   ; load address of SUB_BRANCH,
                 ; set for THUMB by adding 1
BX   R1          ; R1 contains address
                 ; of SUB_BRANCH+1
                 ; to switch to Thumb

SUB_BRANCH:
BL   thumb_sub
                 ; must be in a space of +/- 4 MB
add  R1, #7      ; point to SUB_RETURN
                 ; with bit 0 clear
BX   R1          ; to switch to ARM
SUB_RETURN:
```

https://www.embedded.com/introduction-to-arm-thumb/

Branch and Exchange (4)

- this example assumes that **R1** is *preserved* by the subroutine.
- The **PC** always contains the address of the current instruction plus 8
  - **add R1, PC,#1**
    - (4 bytes)
  - **BX R1**
    - (4 bytes)
  - SUB_BRANCH
    - (**PC** of **add** inst. + 8 bytes)
  -
  -

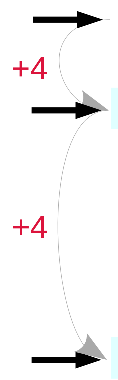
```
        mov  R0, #5        ; argument to function is in R0
  →     add  R1, PC,#1     ; load address of SUB_BRANCH,
  +4                       ; set for THUMB by adding 1
  →     BX   R1            ; R1 contains address
                           ;of SUB_BRANCH+1
  +4                       ;to switch to Thumb

        SUB_BRANCH:
  →     BL   thumb_sub
                           ; must be in a space of +/- 4 MB
        add  R1, #7        ; point to SUB_RETURN
                           ; with bit 0 clear
        BX   R1            ; to switch to ARM
        SUB_RETURN:
```

https://www.embedded.com/introduction-to-arm-thumb/

Branch and Exchange (5)

- The Thumb **BL** instruction actually resolves into two instructions, so *8 bytes* are used between SUB_BRANCH and SUB_RETURN .
- **BL thumb_sub** (4 bytes)
  - **BL (H=0)** Offset_high (2 bytes)
  - **BL (H=1)** Offset_low (2 bytes)
- **add R1, #7** (2 bytes)
- **BX R1** (2 bytes)



```
mov  R0, #5        ; argument to function is in R0
add  R1, PC,#1     ; load address of SUB_BRANCH,
                   ; set for THUMB by adding 1
BX   R1            ; R1 contains address
                   ;of SUB_BRANCH+1
                   ;to switch to Thumb

SUB_BRANCH:
BL   thumb_sub
                   ; must be in a space of +/- 4 MB
add  R1, #7        ; point to SUB_RETURN
                   ; with bit 0 clear
BX   R1            ; to switch to ARM
SUB_RETURN:
```

https://www.embedded.com/introduction-to-arm-thumb/

Thumb → ARM interworking call

to **BL** to an intermediate Thumb code segment that executes the **BX** instruction.

BL __call_via_r4
BX r4

the **BL** instruction loads the **link register** immediately before the **BX** instruction is executed.

Stop
BX r4

In addition, the Thumb instruction set version of **BL** sets **bit 0** when it loads the **link register** with the **return address**.

LR[0] = 0 → ARM state

When a Thumb-to-ARM interworking subroutine call returns using a **BX LR** instruction, it causes the required state change to occur automatically.

BX LR

```
        CODE16
ThumbProg
        MOV     r0, #2
        MOV     r1, #3
        ADR     r4, ARMSubroutine

        BL      __call_via_r4
```

```
Stop
        MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0xAB
__call_via_r4

        BX      r4
```

```
        CODE32
ARMSubroutine
        ADD     r0, r0, r1
        BX      LR

        END
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

Thumb → ARM interworking call

If you always use the same register
to store the address of the ARM subroutine
that is being called from Thumb,
this segment can be used
to send an interworking call to any ARM subroutine.

You must use a **BX LR** instruction
at the end of the ARM subroutine to return to the caller.

You cannot use the **MOV pc,lr** instruction
to return in this situation
because it does not cause
the required change of state.

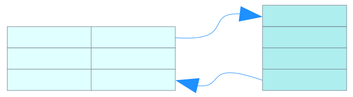
```
ADR   r4, ARMSubroutine

        CODE16
ThumbProg
        ***
        ADR   r4, ARMSubroutine
        BL    __call_via_r4
        ***
__call_via_r4
        BX    r4


        CODE32
ARMSubroutine
        ***
        BX    LR
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

ARM → Thumb interworking call



no need to set bit 0 of the **link register**
because the routine is returning to ARM state.

store the return address by copying **PC** into **LR**
with a **MOV lr,pc** instruction
immediately before the **BX** instruction.

Remember that the address operand to the **BX** instruction
that calls the Thumb subroutine must have bit 0 set
so that the processor executes in Thumb state on arrival.

As with Thumb-to-ARM interworking subroutine calls,
you must use a **BX** instruction to return.

LR[0] = 0 → ARM state

```
ADR   r4, ThumbSub + 1
BX    r4
```

```
        CODE16
        ADR          r4, ThumbSub + 1
        …
        MOV     lr, pc
        BX           r4
```

```
        CODE16
ThumbSub
        ADD     r0, r0, r1
        BX      LR
        END
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

ARM → Thumb interworking call example code (1)

```
        AREA  ArmAdd,CODE,READONLY          ; name this block of code.
        ENTRY                               ; Mark 1st instruction to call.
                                            ; Assembler starts in ARM mode.
main
        ADR     r2, ThumbProg + 1           ; Generate branch target address and set bit 0,
                                            ; hence arrive at target in Thumb state.
        BX      r2                          ; Branch exchange to ThumbProg.
        CODE16                              ; Subsequent instructions are Thumb.
ThumbProg
        MOV     r0, #2                      ; Load r0 with value 2.
        MOV     r1, #3                      ; Load r1 with value 3.
        ADR     r4, ARMSubroutine           ; Generate branch target address, leaving bit 0
                                            ; clear in order to arrive in ARM state.
        BL      __call_via_r4               ; Branch and link to Thumb code segment that will
                                            ; carry out the BX to the ARM subroutine.
                                            ; The BL causes bit 0 of lr to be set.
Stop                                        ; Terminate execution.
        MOV     r0, #0x18                   ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SWI     0xAB                        ; Angel semihosting Thumb SWI
__call_via_r4                               ; This Thumb code segment will
                                            ; BX to the address contained in r4.
        BX      r4                          ; Branch exchange.
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

ARM → Thumb interworking call example code (2)

```
        CODE32                          ; Subsequent instructions are ARM.
ARMSubroutine
        ADD     r0, r0, r1              ; Add the numbers together
        BX      LR                      ; and return to Thumb caller
                                        ; (bit 0 of LR set by Thumb BL).
        END                             ; Mark end of this file.
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

Thumb → ARM interworking call example code (1)

```
        AREA  ThumbAdd,CODE,READONLY                    ; Name this block of code.
        ENTRY                                           ; Mark 1st instruction to call.
                                                        ; Assembler starts in ARM mode.
main
        MOV     r0, #2                                  ; Load r0 with value 2.
        MOV     r1, #3                                  ; Load r1 with value 3.
        ADR     r4, ThumbSub + 1                        ; Generate branch target address and set bit 0,
                                                        ; hence arrive at target in Thumb state.
        MOV     lr, pc                                  ; Store the return address.
        BX      r4                                      ; Branch exchange to subroutine ThumbSub.
Stop                                                    ; Terminate execution.
        MOV     r0, #0x18                               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                            ; ADP_Stopped_ApplicationExit
        SWI     0x123456                                        ; Angel semihosting ARM SWI

        CODE16                                          ; Subsequent instructions are Thumb.
ThumbSub
        ADD     r0, r0, r1                              ; Add the numbers together
        BX      LR                                      ; and return to ARM caller.
        END                                             ; Mark end of this file.
```

https://developer.arm.com/documentation/dui0040/d/Interworking-ARM-and-Thumb/Basic-assembly-language-interworking/Implementing-interworking-assembly-language-

Cortex-M3 : 32-bit processor

- The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions.
- Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.
- The **Cortex-M3 processor** is a high performance 32-bit processor designed for the microcontroller market.
- It offers significant benefits to developers, including: outstanding processing performance combined with
  - fast interrupt handling.
  - enhanced system debug with
  - extensive breakpoint and trace capabilities.

https://developer.arm.com/documentation/dui0552/a/introduction/about-the-cortex-m3-processor-and-core-peripherals

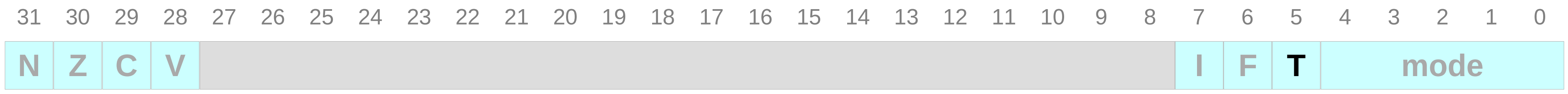
Cortex-M3 : Thumb state only

- The **Cortex-M3** processor only supports execution of instructions in **Thumb state**. (**T** = **1**)

- The following can clear the **T** bit to **0**:
  - instructions **BLX**, **BX** and **POP {PC}**
  - restoration from the stacked **xPSR** value on an exception return
  - bit[0] of the vector value on an exception entry or reset.

- In the **Cortex-M3** processor, attempting to execute instructions when the **T** bit is **0** results in a fault or lockup. See Lockup for more information.

- The Thumb status bit (**T**) indicates the processor's current state:
  - 0 for ARM state (default)
  - 1 for Thumb.



https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers

ARM Exception Handling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	Z	C	V	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R

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