Thumb Instruction Programming

Young Won Lim
11/12/24

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

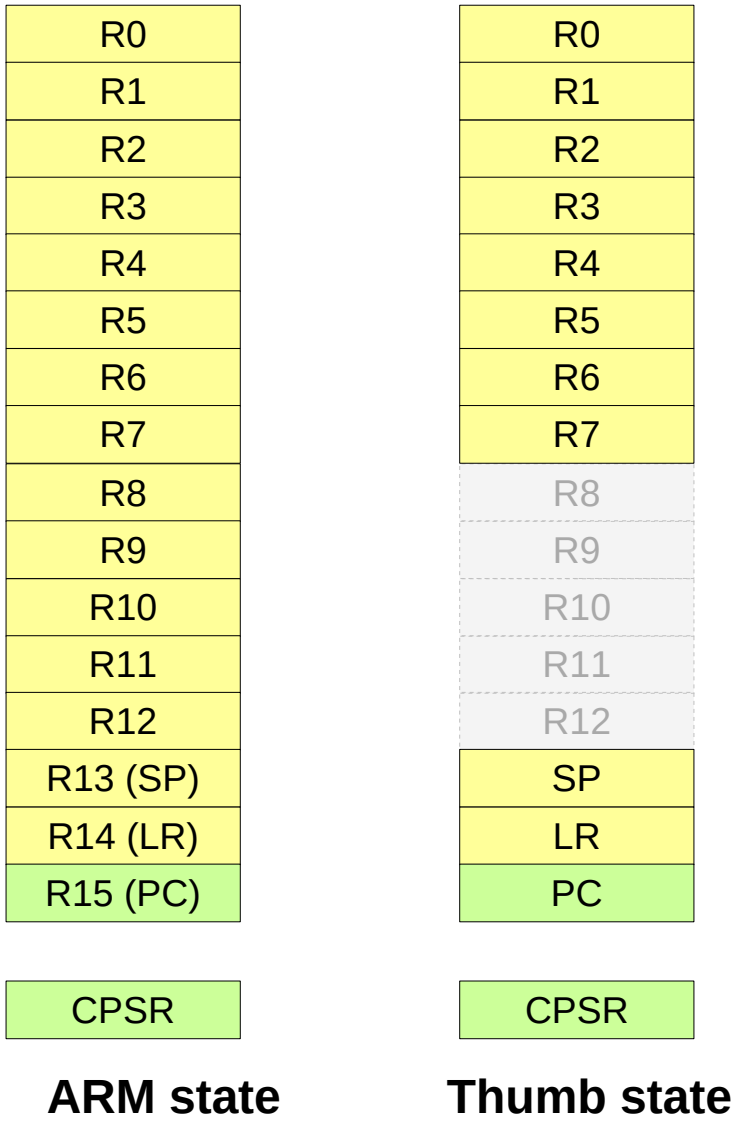
Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

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

Thumb Instruction Programming

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/

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

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

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

• ~~**B** {cond} Rm~~ • **BX** {cond} Rm	• ~~**BL** {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/

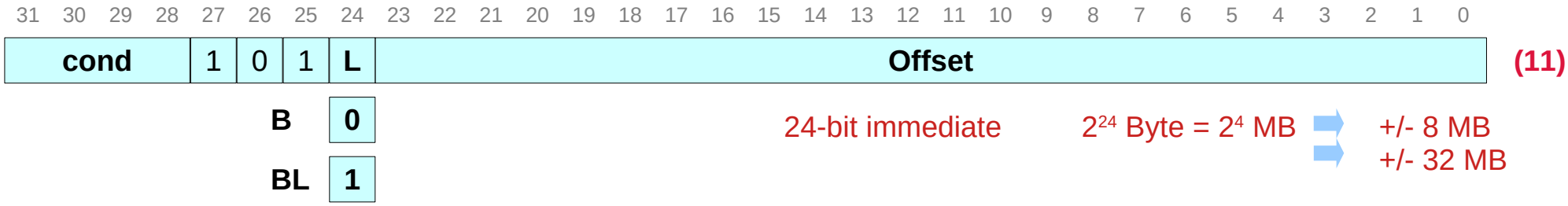
B and **BL** instructions (3)

- 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)



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

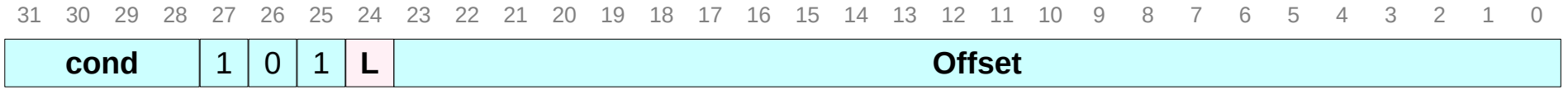
with label
always changes the state.
ARM state → Thumb state
Thumb state → ARM state

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**

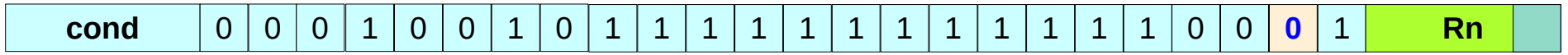
https://www.embedded.com/introduction-to-arm-thumb/

B, BL, BX, and BLX instructions



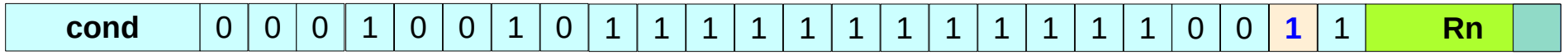
Branch — 0 — B{<cond>} <address> — PC := Offset

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



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

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



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

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

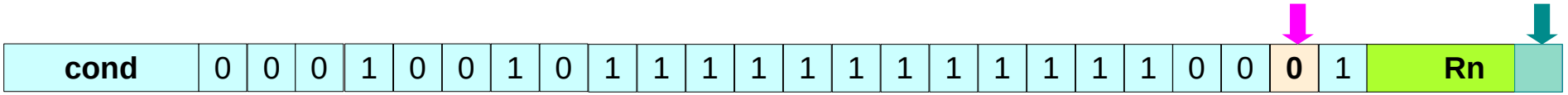
always changes the state.
ARM state → Thumb state
Thumb state → ARM state

Branch instructions – changing the state

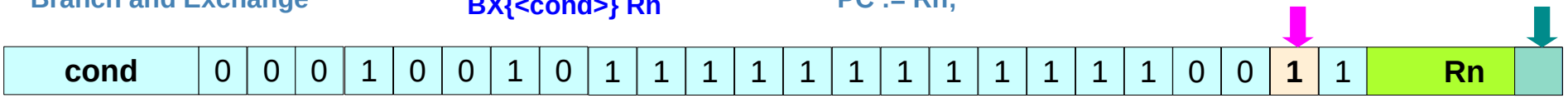
BX Rn
BLX Rn

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

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



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

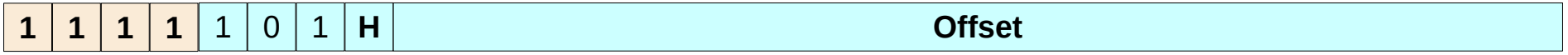


Branch with Link and Exchange — BLX{<cond>} Rn — R14 := PC+8; PC := Rn;

BLX *label*

always changes the state.

ARM state → Thumb state
Thumb state → ARM state



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*.

R0 | 0

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

R0 | 1

```
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/

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

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 ⬅

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

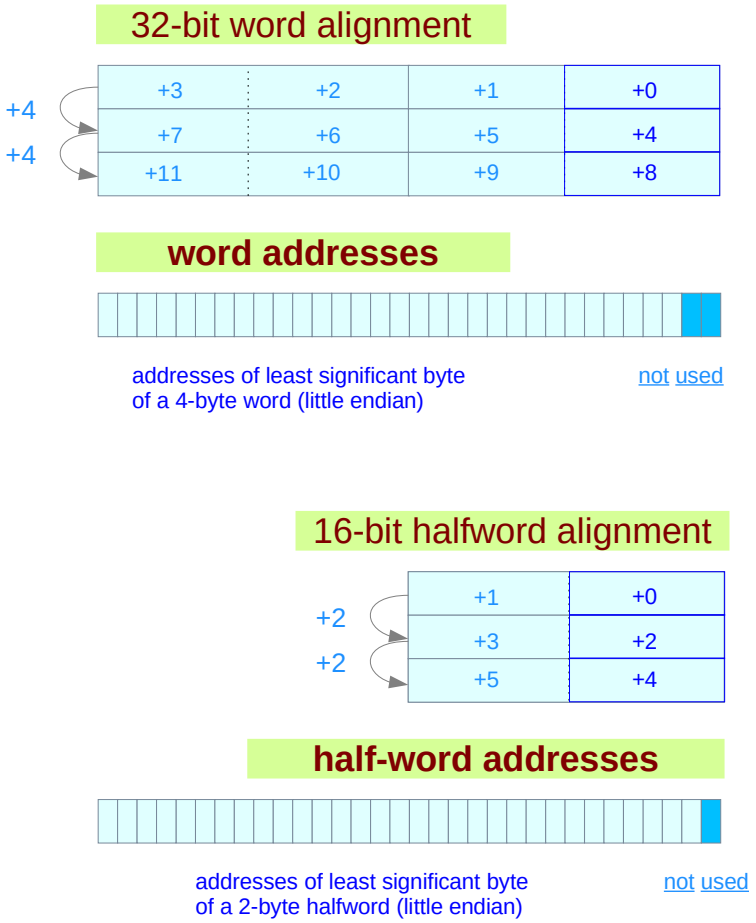
BX Rm ⬅
BLX Rm ⬅
; destination address in the regsiter Rm
If **Rm**[0] is 0, to ARM state.
If **Rm**[0] is 1, to Thumb state.

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

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

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.



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

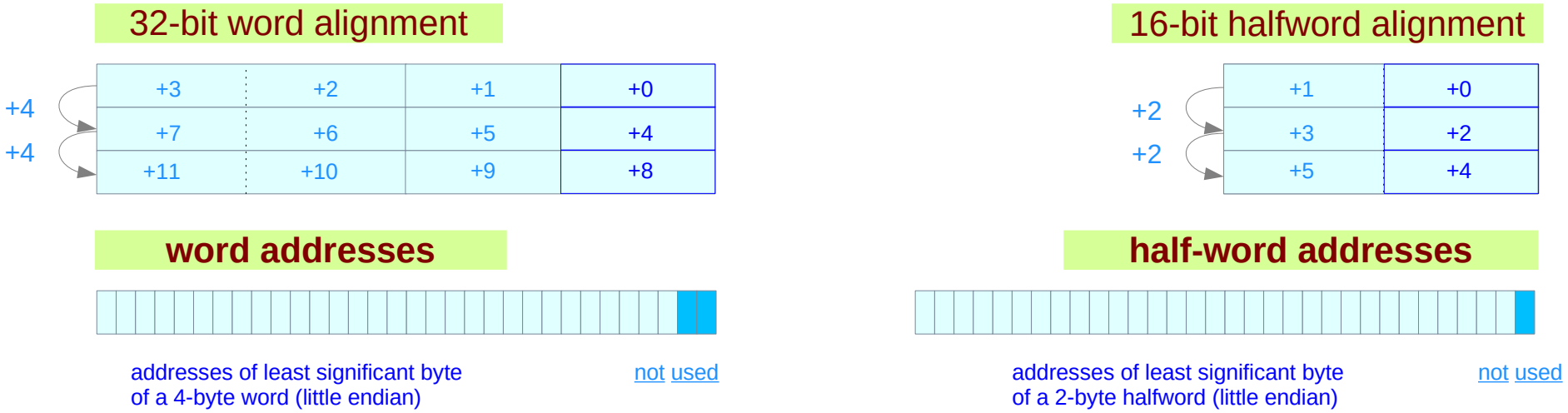
32-bit / 16-bit alignment

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)



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 (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

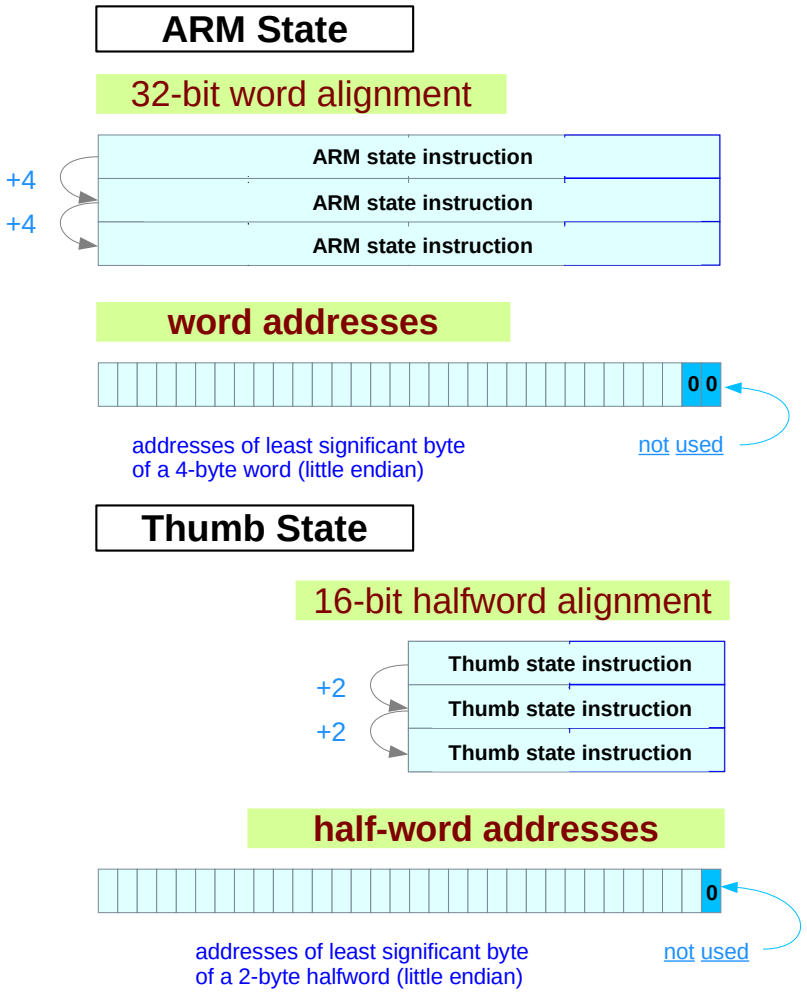
in ARM processors,
all ARM instructions take up one word (4 bytes).
all Thumb instructions take up one halfword (2 bytes).

incrementing the PC in the ARM state

PC + 4

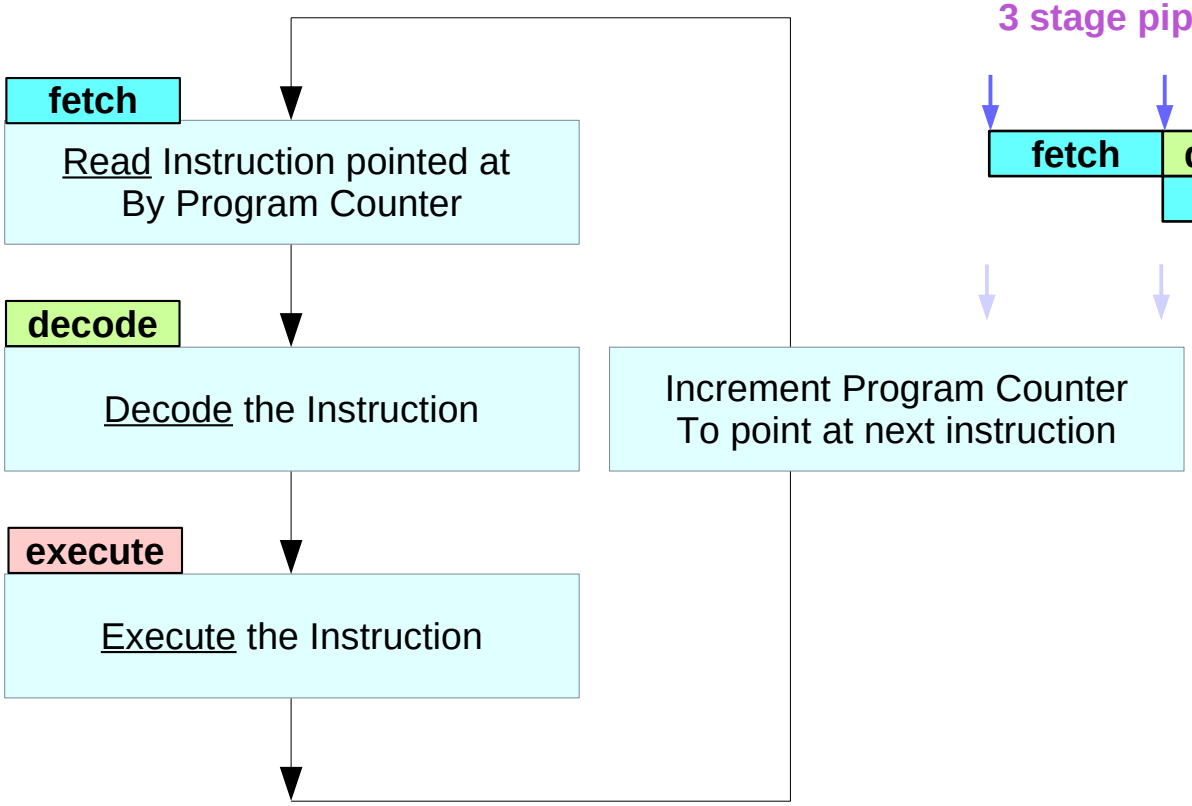
incrementing the PC in the Thumb state

PC + 2

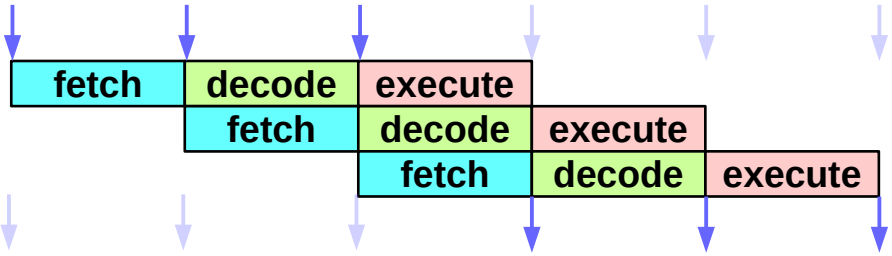


http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture1/lecture1-4-2.html

PC (Program Counter) **R15** Register



3 stage pipeline execution

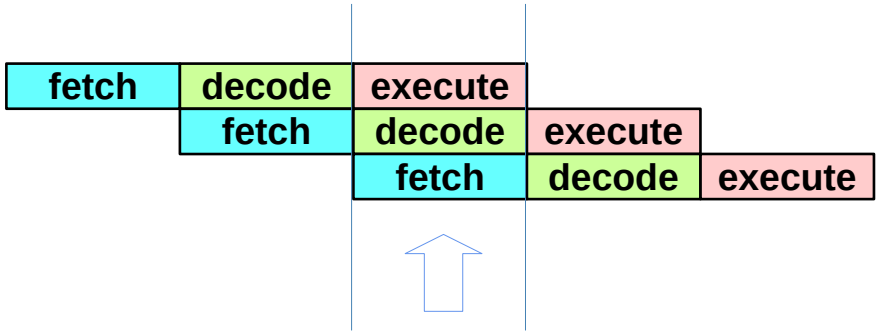


Execute a current instruction
Decode the next instruction
Fetch the next's next instruction

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



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

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

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.

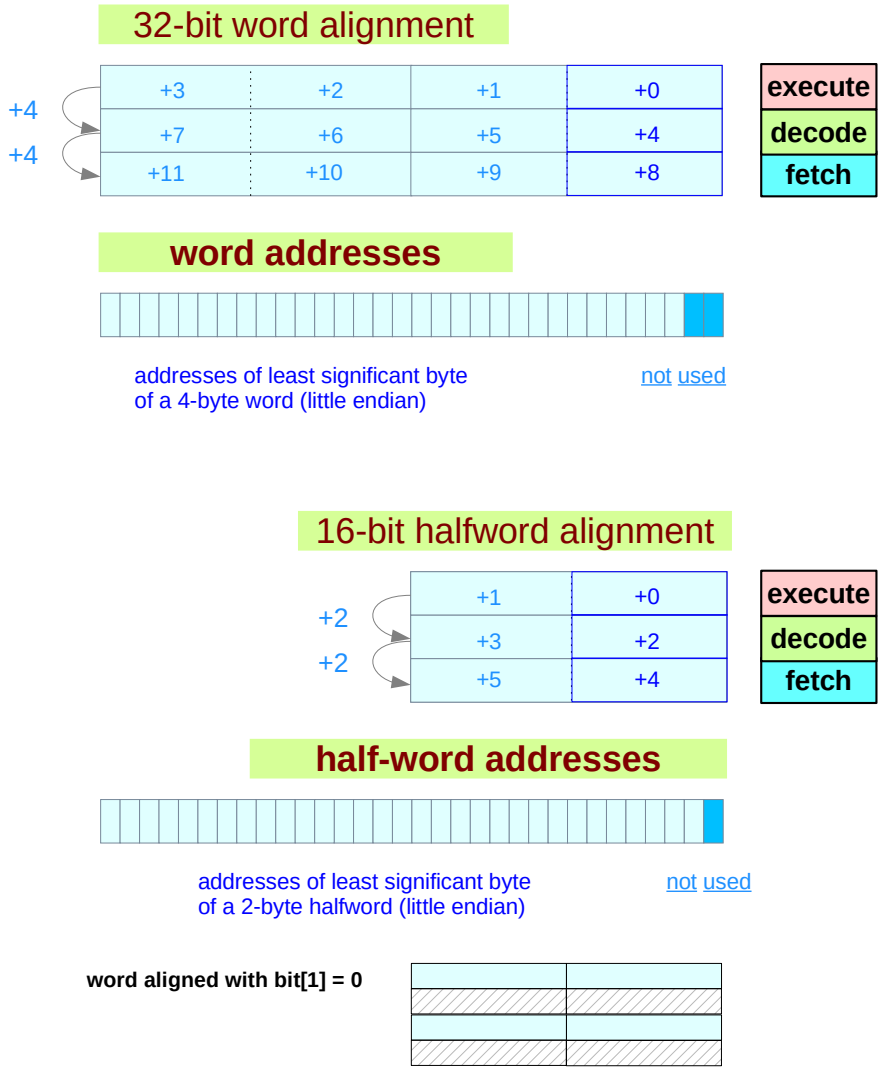
In **T32** code: PC + 4

For **B**, **BL**, **CBNZ**, and **CBZ** instructions,
the value of the PC is
the address of the current instruction plus 4 bytes.

For all other instructions that use labels,
the value of the PC is
the address of the current instruction plus 4 bytes,
with bit[1] of the result cleared to 0
to make it word-aligned.

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

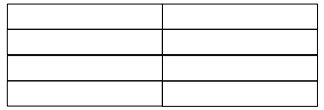
In **A32** code, PC + 8
the value of the PC is
the address of the current instruction plus 8 bytes.

In **T32** code: PC + 4

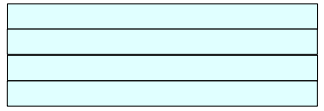
For **B**, **BL**, **CBNZ**, and **CBZ** instructions,
the value of the PC is
the address of the current instruction plus 4 bytes.

For all other instructions that use labels,
the value of the PC is
the address of the current instruction plus 4 bytes,
with bit[1] of the result cleared to 0
to make it word-aligned.

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

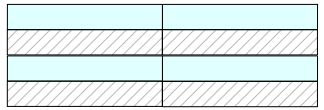


PC can point any halfword



PC can point only a word

word aligned with bit[1] = 0



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

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**

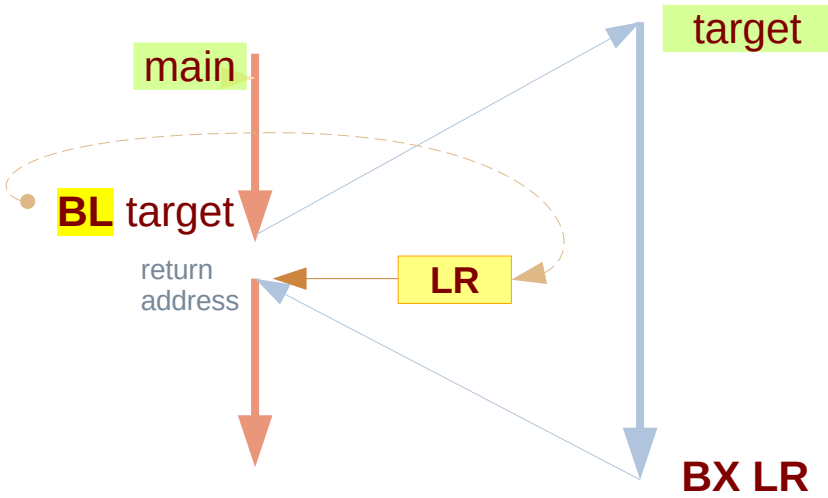
PC ← the destination address target

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

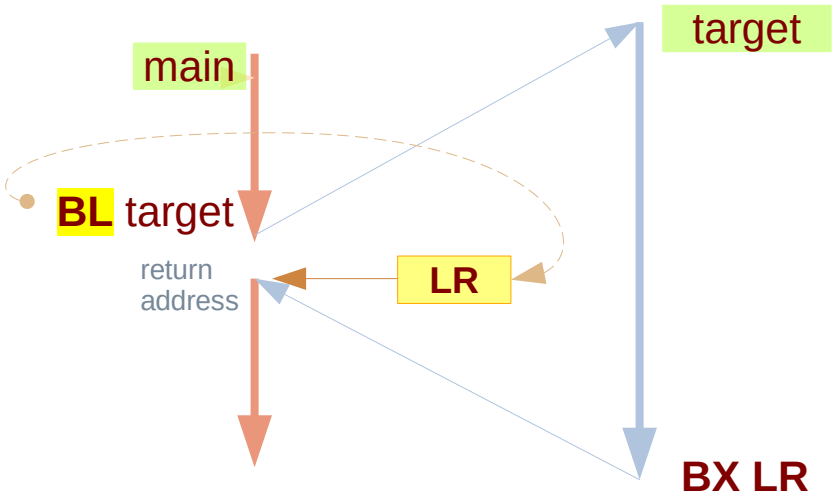
/IHI0042E_aapcs.pdf

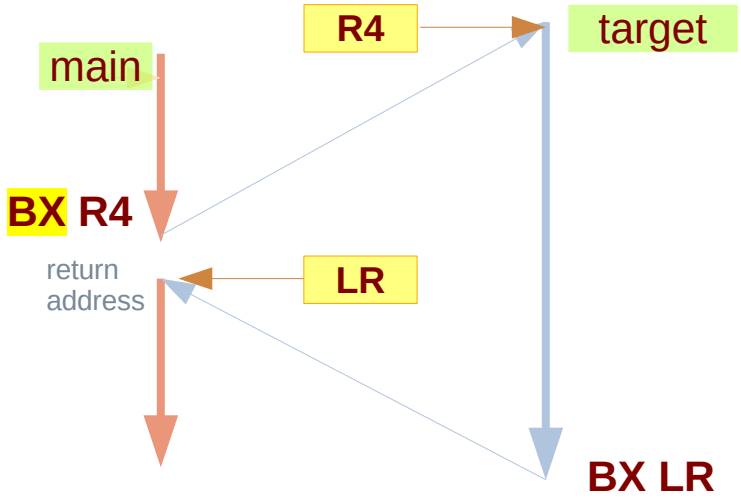


"BL target" in Thumb state
then assign **LR[0] = 1**
(to return to Thumb code)

"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**

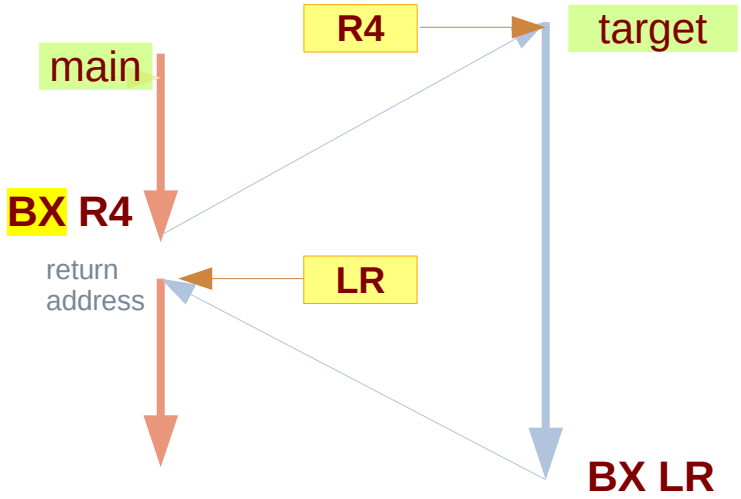
Subroutine call (3) **BX** (Branch and eXchange) operation

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, **R4 := target+1;**
to call a subroutine addressed by **R4**
with control returning to the following instruction,

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

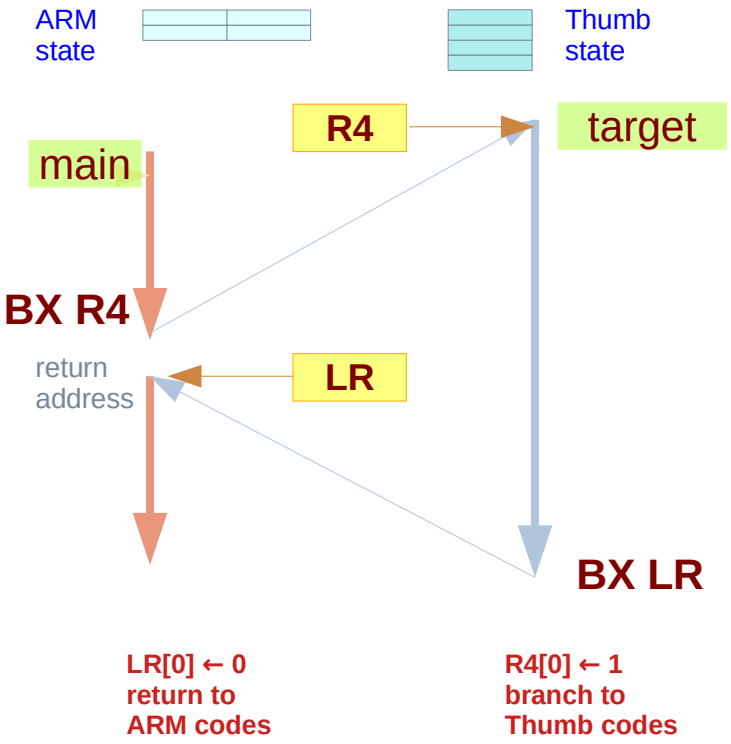


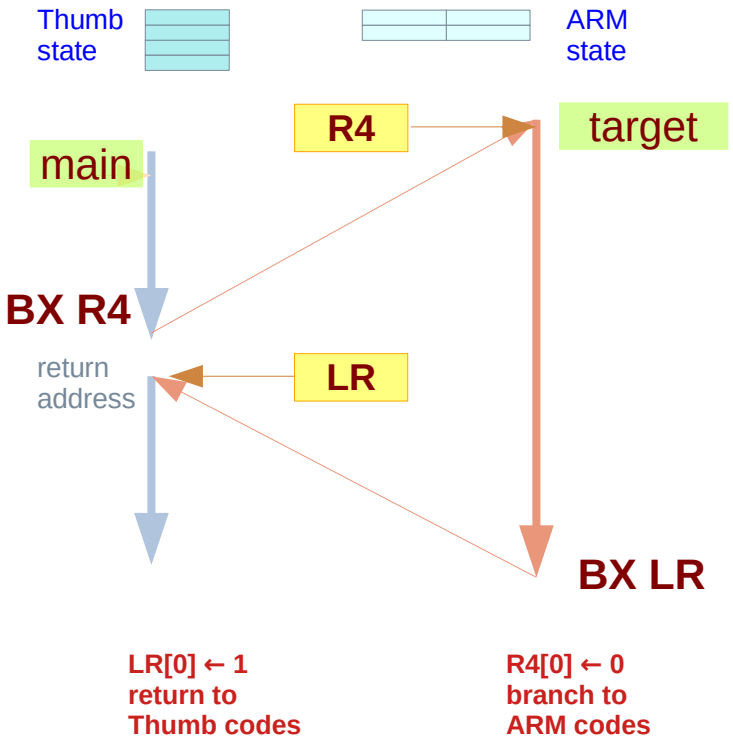
LR[31:1] ← the return address

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

/IHI0042E_aapcs.pdf

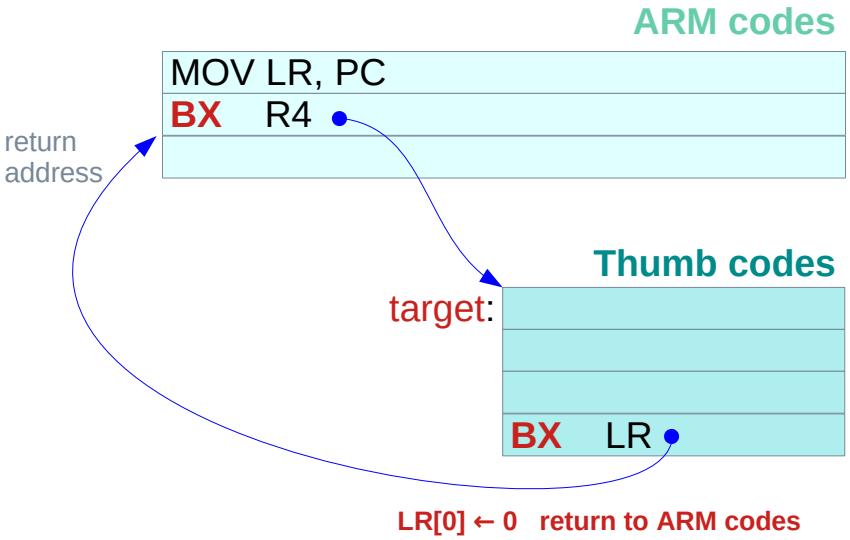
Subroutine call (4) ARM vs. Thumb state

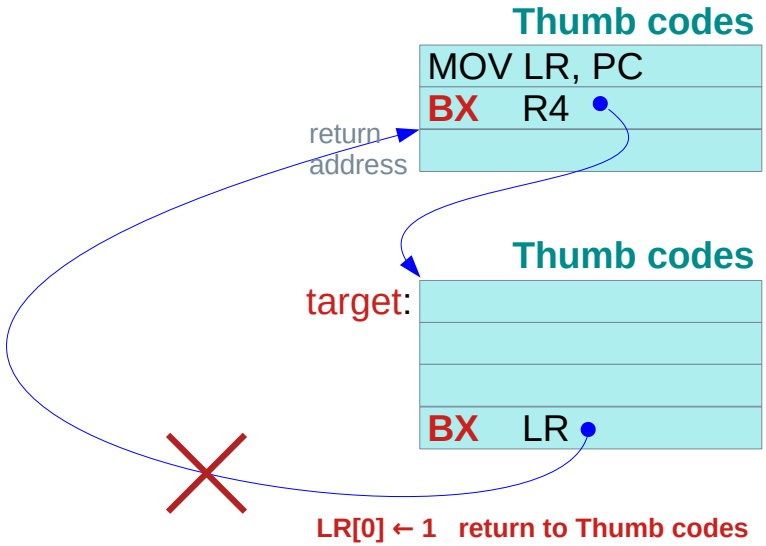




/IHI0042E_aapcs.pdf

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

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

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

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

LR[0] ← 1 return to Thumb codes

Return from a procedure (1-1)

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 (1-2)

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**.

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

Return from a procedure (2-1)

When writing non-leaf functions, i.e.
functions that calls other functions,
the value of **lr** must be preserved,
since calling another 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 preserve
the registers we modify.

For example, if we want
to preserve **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.
```

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

Return from a procedure (2-1)

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…

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

Return from a procedure (2-1)

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

I.e. PUSH {R0,R1,R2}, PUSH {R2,R1,R0},
and PUSH {R1,R2,R0}
all would result in the some ordering in the stack

because "...the lowest/highest numbered register (R0/R2)
uses the lowest/highest (stack) memory address...".

Does that mean if a single PUSH instruction
has multiple registers in the bracket,
the assembler automatically sorts the pushing actions out
in the object code, where PUSH R2 goes first into the stack
to take the highest address, followed by PUSH R1
and ended with PUSH R0 taking the lowest address?

So if I want to guarantee R2 get pushed last
and popped first in a LIFO stack
(i.e. SP pointing R2 or for R2
to take the lowest stack address),
I cannot do so in one PUSH bracket statement
but only separately with PUSH R0; PUSH R1; PUSH R2?

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

https: -pop-brackets

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

Return from a procedure (2-1)

```
.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}
```

from the ARM ARM you can see in the push instruction the lower 8 bits are a register list/mask. So r0 is bit 0, r1 is bit 1 and so on. So the 7 in b407 indicates the three registers r0,r1,r2. The logic operates on machine code not assembly language, the machine code goes from bit 7 to bit 0 if set then push that register. All the assembler does is create the machine code it doesn't create extra instructions or anything like that.

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

Return from a procedure (2-1)

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 .

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

Return from a procedure (2-2)

This is our first way of returning:
using push to restore all the registers,
except putting what was lr
when we are doing push 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.

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

Return from a procedure (3)

Method 2

We can use an unconditional jump to register to return,
which is useful in leaf functions where lr is never stored on the stack.
This is simply:

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, but as stated in the beginning,
ARM lets you manipulate the program counter as you would any other register.
So… we have:

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 (4)

Method n

Of course, there are many other ways of copying the value in one register into another, and to list that would be fairly silly. But as long as lr at the beginning of the function call is placed 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 (5)

```
.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 (6)

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 (7)

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 fully 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 (8)

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 (9)

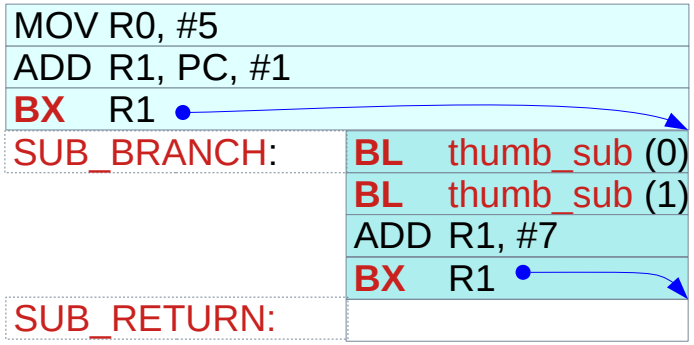
That's what Entry and EXIT (and friends) are for:

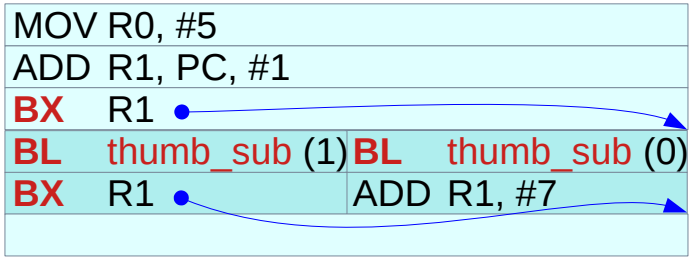
Entry → push a stack frame for procedure entry
(implicitly adds lr to the register list),
optionally reserving a block of local workspace on the stack
EXIT → return from a procedure by popping the workspace + register list
from the most recent Entry
(i.e. the one located directly before it in the assembler 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

State changing example (1)





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

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

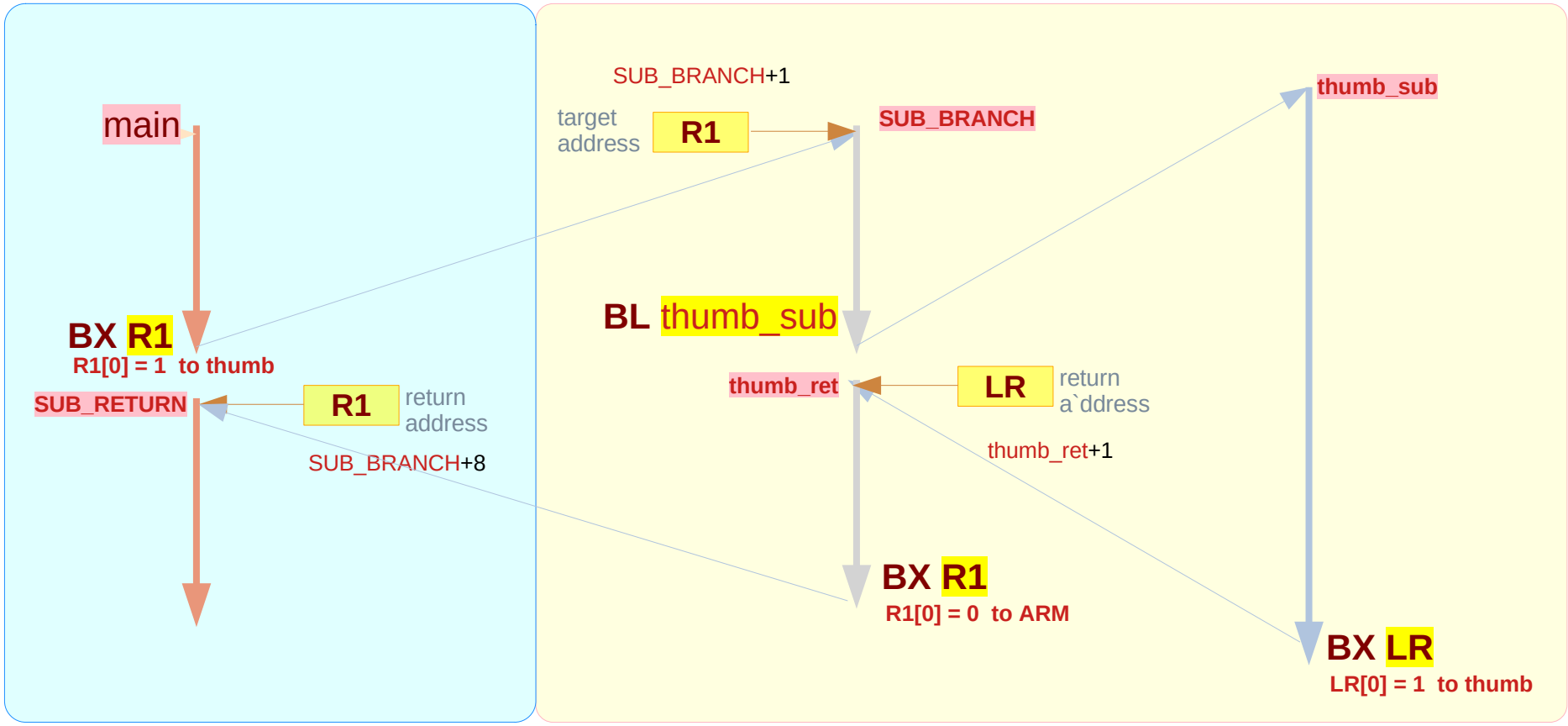
execution mode switch from **ARM** to **Thumb**
at the SUB_BRANCH and
the program will execute in **Thumb** mode.

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-at

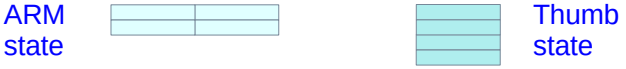
Branch and link operation (2)



/IHI0042E_aapcs.pdf

Branch and Exchange (2)

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)

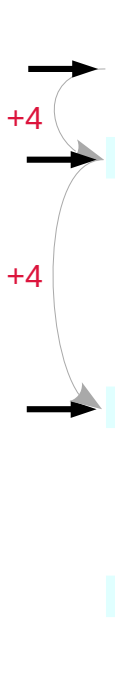
```
mov  R0, #5       ; argument to function is in R0
add  R1, PC,#1    ; load address of SUB_BRANCH,
                  ; set for THUMB by adding 1
BX   R1           ; R1 contains address
                  ; of SUB_BRANCH+1
                  ; to switch to Thumb

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,
                        ; 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/

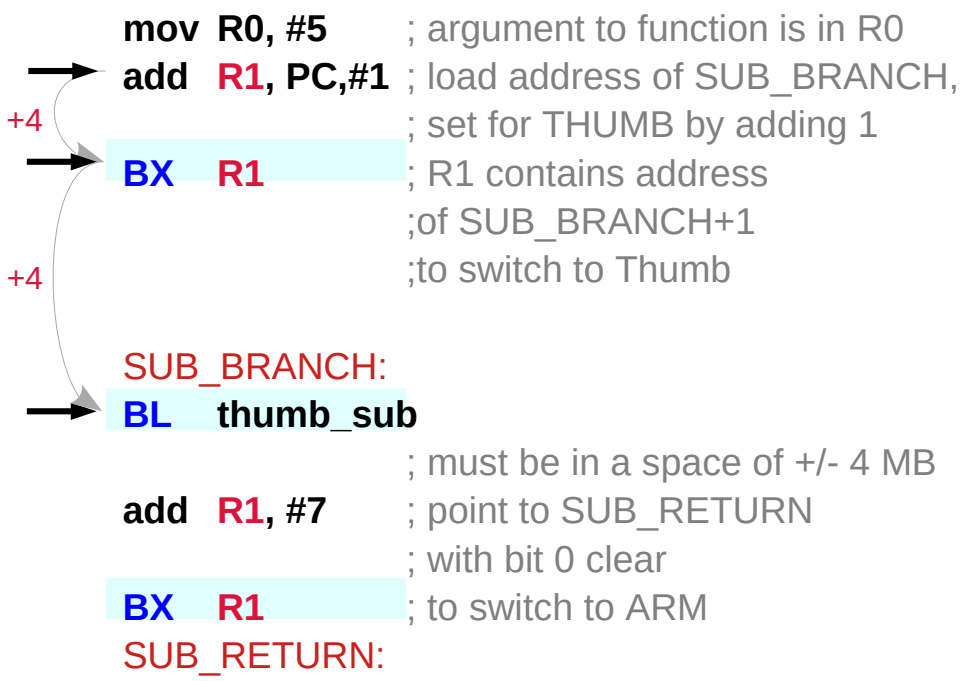
Branch and Exchange (5)

- 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)

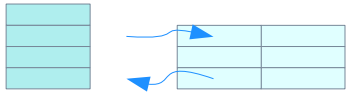
  - **BL** (**H=0**) Offset_high (2 bytes)
  - **BL** (**H=1**) Offset_low (2 bytes)

- **add** **R1**, **#7** (2 bytes)
- **BX** **R1** (2 bytes)



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.

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

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

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

BL __call_via_r4
BX r4

Stop
BX r4

LR[0] = 0 → ARM state

BX LR

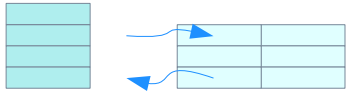
```
        CODE16
ThumbProg
        MOV     r0, #2
        MOV     r1, #3
        ADR     r4, ARMSubroutine

        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.

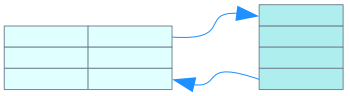
You must use a **BX LR** instruction
at the end of the ARM subroutine to return to the caller.

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
ARMSubroutine
        ADD     r0, r0, r1
        BX      LR

        END
```

```
; Subsequent instructions are ARM.

; Add the numbers together
; and return to Thumb caller
; (bit 0 of LR set by Thumb BL).
; 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

Thumb Instruction

Thumb instruction set benefits

- The biggest reason to look for an ARM processor with the Thumb instruction set is if you need to reduce code density.
- In addition to reducing the total amount of memory required, you may also be able to narrow the data bus to just 16 bits.
- With the narrower bus, it will take two bus cycles to fetch a single 32-bit instruction;
- but you'll only pay that penalty in the parts of your code that can't be implemented with the Thumb instructions.
- And you'll still have the benefits of a powerful 32-bit RISC processor. A nifty trick indeed.

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

Thumb instructions (1)

- The Thumb instructions
  - 16-bit instructions
  - a compact *shorthand* for a subset of the 32-bit ARM instructions

- every Thumb instruction has the *equivalent* 32-bit ARM instruction.

- not every ARM instructions has the *equivalent* Thumb subset;
  - for example, there's no way to *access* status or coprocessor registers.

- a single ARM instruction can only be simulated with a sequence of Thumb instructions
  - a long branch with link (**BL**)
  - the assembler splits
    Instruction 1 (**H = 0**)
    Instruction 2 (**H = 1**)

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

Thumb instructions (2)

- the ARM contains only one instruction set:
  the 32-bit set.

- When it's operating in the Thumb **state**,

  the processor simply *expands*
  the smaller *shorthand instructions*
  fetched from memory

  into their 32-bit equivalents.

- The difference between two equivalent instructions
  (the ARM and Thumb instructions) lies in

  how the *instructions* are
  fetched and *interpreted*
  prior to execution,
  not in how they *function*.

- dedicated hardware expands
  the 16-bit instruction into 32-bit

  it doesn't slow execution even a bit.

- the narrower 16-bit instructions
  do offer memory advantages.

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

Thumb instructions (3)

- Roughly speaking, a CPU instruction is
  a particular *sequence of bits*

- to the CPU, a particular *sequence of bits* could mean
  "**add** two 32-bit values and carry"

- The exact value of *bits in this sequence*
  has nothing to do with values being added.

- In the ARM mode, this *sequence of bits* has 32 bits.
- In the thumb mode, it only has 16 bits.

- apparently, the thumb mode has less number
  of encoded instructions than the ARM mode
  (less bits to encode them),

- for a same function,
  most instructions are encoded differently
  for the ARM and the thumb modes, respectively,

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

Thumb instructions (4)

- for example, the x86 uses 8-bit instructions
  but is also able to work on 32 bit values.

- For ARM, the *instruction length* is what changes
  when you switch to/from ARM and thumb modes.

- For example, the instruction **MOV R0, R1**
  copy the contents of the 32-bit **R1** register
  to the **R0** register
  is encoded in the following way:

  - *E1A00001* for ARM (32 bit : 4 bytes)
  - *4608* for Thumb (16-bit : 2 bytes)

- But the processor will perform
  exactly the same operation, and
  it will do it on 32-bit wide data,
  whatever the mode.

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

Thumb instructions (5)

- The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions.

- Thumb instructions are 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.

- Thumb instructions operate with the standard ARM register configuration, enabling excellent interoperability between ARM and Thumb states.

- Thumb has all the advantages of a 32-bit core:

  - 32-bit address space
  - 32-bit registers
  - 32-bit shifter and Arithmetic Logic Unit (ALU)
  - 32-bit memory transfer

https://developer.arm.com/documentation/ddi0333/h/introduction/arm1176jz-s-architecture-with-jazelle-technology/the-thumb-instruction-set

Thumb instructions (6)

- The ARM processor
  can *manipulate* 32 bit values
  because it is a 32-bit processor,
  whatever mode it is running in (Thumb or ARM).

- thus, registers are 32 bits wide

- register width doesn't change
  when you switch mode (state)

- the data bus width of the processor
  has nothing to do with
  the length of the instructions.

- The instructions could be encoded in any length.

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

Thumb instructions (7)

- The Thumb instruction set provides
  most of the *functionality* of a typical application.

  - arithmetic and logical operations
  - load/store data movements
  - conditional and unconditional branches

- any code written in C could be
  executed successfully in Thumb state.

- However, device drivers and exception handlers
  must often be written at least partly in ARM state

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

Thumb instructions (8)

- Switching modes allows programmers
  to decide on the compromise
  between **code density** and **flexibility**

- can *pack more instructions*
  in a kB of code with 16-bit instructions,

- but the 32 bit instructions are more *flexible*
  - they offer more features and
  - you can do more with a single instruction

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

Thumb instructions (9)

- All Thumb instructions are 16 bits in length.
- Thumb provides approximately 30% better code density over ARM code.
- Most code written for Thumb is in a high-level language such as C and C++.
- ATPCS (ARM Thumb Procedure Call Standard) defines how ARM and Thumb code call each other, called ARM-Thumb interworking.
- Interworking uses the branch exchange (**BX**) instruction and branch exchange with link (**BLX**) instruction to *change state* and *jump* to a specific routine.

https://www.sciencedirect.com/topics/computer-science/thumb-instruction-set

Thumb instructions (10)

- In Thumb, *only* the branch instructions
  are conditionally executed.

- The barrel shift operations are separate instructions
  - **ASR**
  - **LSL**
  - **LSR**
  - **ROR**

- The multiple-register load-store instructions
  only support the increment after (**IA**) addressing mode.

- The Thumb instruction set includes **POP** and **PUSH**
  instructions as stack operations.

- **POP** and **PUSH** instructions only support
  a full descending stack.

- There are no Thumb instructions
  to access the coprocessors, cpsr, and spsr.

https://www.sciencedirect.com/topics/computer-science/thumb-instruction-set

Thumb instructions (11)

	ARM (**CPSR T**=0)	**Thumb** (**CPSR T**=1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	most	only branch instruction
Data Processing Instructions	access to barrel shifter and ALU	*separate* barrel shifter and ALU instructions
Program Status Reg	R/W in privileged mode	*no* direct access
Register usage	15 general purpose reg + PC	8 general purpose reg + 7 high reg + PC

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

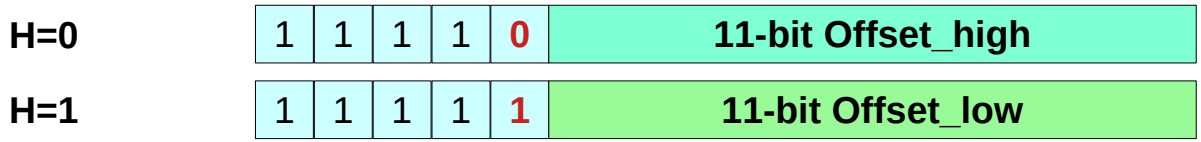
Thumb long branch with link **BL** instruction (1)

THUMB assembler : **BL label**

H=0
LR := PC + OffsetHigh << 12

H=1
temp := next instruction address
PC := LR + OffsetLow << 1
LR := temp | 1

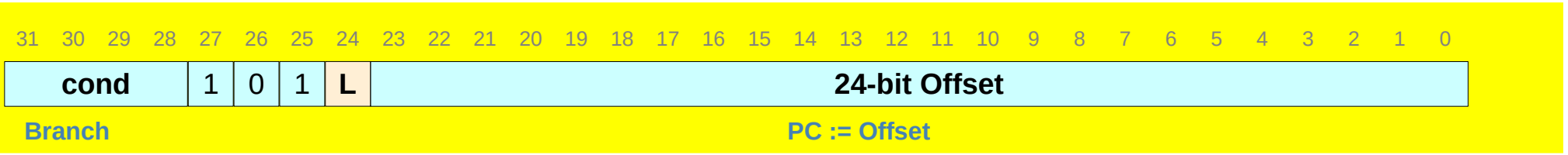
PC := PC + (OffsetHigh << 12) + (OffsetLow << 1)



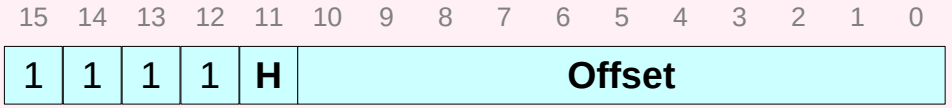
http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf?ref=zdimension.fr

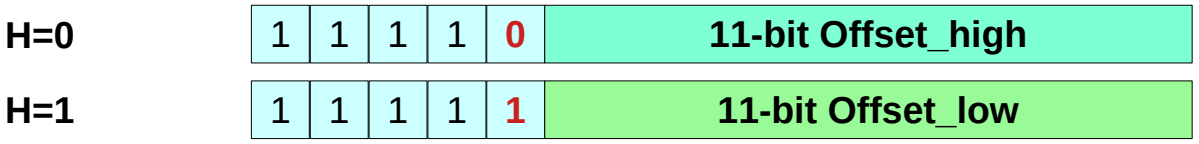
Thumb long branch with link **BL** instruction (2)

ARM B or BL instruction



Thumb BL instruction





http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf?ref=zdimension.fr

Thumb long branch with link **BL** instruction (3)

Examples

```
        BL faraway      ; Unconditionally Branch to 'faraway'
next ...                ; and place following instruction address,
                        ; ie 'next', in R14, the Link Register (LR)
                        ; and set bit 0 of LR high (1)
                        ; Note that the THUMB opcodes will contain
                        ; the number of halfwords to offset.

faraway ...             ; Must be Half-word aligned.
```

H=0
LR := PC + OffsetHigh << 12

H=1
temp := next instruction address
PC := LR + OffsetLow << 1
LR := temp | 1

PC := PC + (OffsetHigh << 12) + (OffsetLow << 1)

http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf?ref=zdimension.fr

Thumb long branch with link **BL** instruction (4)

- This format specifies a long branch with link.

- The assembler splits
  the 23-bit two's complement half-word offset
  specifed by the label into *two* 11-bit halves,
  ignoring bit 0 (which must be 0),
  and creates *two* THUMB instructions.

- **Instruction 1 (H = 0)**
  - In the *first* instruction
    the Offset field contains
  - the upper 11 bits of the target address.
  - this is shifted left by 12 bits and
  - added to the current **PC** address.
  - The resulting address is placed in **LR**.

- **Instruction 2 (H =1)**
  - In the *second* instruction
    the Offset field contains
  - the lower 11-bit of the target address.
  - this is shifted left by 1 bit and
  - added to **LR**.
  - **LR**, which now contains the full 23-bit address,
    is placed in **PC**,
    the address of the instruction following the **BL**
  - is placed in **LR** and bit 0 of **LR** is set.
  - the branch offset must take account of
    the prefetch operation,
  - which causes the **PC** to be 1 word (4 bytes)
    ahead of the current instruction

http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt3.pdf?ref=zdimension.fr

Thumb-2 Instruction

Thumb-2 Instructions (1)

- Thumb-1 only does 16 bit instructions
- Thumb-2 can do both 16 bit & 32 bit instructions

- Thumb-1 and Thumb-2
  - share same architecture for 32 bit data.
  - share the same data bus
    since only the instruction registers are *different*.

- for 64 bit processors,
  Thumb (**T32**) can support
    both 16 & 32 bit instructions
    with some different in each set
  in order to conserve code space for some applications
  but at the expense of duplicate libraries.

Thumb-1
16-bit
instructions
32-bit GP regs

Thumb-2
Mixed 16- and 32-bit
instructions
32-bit GP regs

T32
Mixed 16- and 32-bit
instructions
32-bit GP regs

A32
32-bit instructions
32-bit GP regs

A64
32-bit instructions
32- and 64-bit GP regs

https://electronics.stackexchange.com/questions/353192/how-does-an-arm-processor-in-thumb-state-execute-32-bit-values

Thumb-2 Instructions (2)

- Thumb-2 is an enhancement
  to the 16-bit Thumb instruction set.

- Thumb-2 adds 32-bit instructions
  that can be *freely intermixed*
  with 16-bit instructions in a program.

- the additional 32-bit instructions
  enable Thumb-2

  - to cover the functionality of the ARM instruction set.

  - to combine the code density of earlier versions of
    Thumb, with performance of the ARM instruction.

ARM		32-bit
Thumb	16-bit	
Thumb-2	16-bit	32-bit

added 32-bit Thumb-2 instruction

https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

Thumb-2 Instructions (3)

- The most important difference
  between the Thumb-2 instruction set
  and the ARM instruction set is

  that most 32-bit Thumb instructions are unconditional,
  whereas most ARM instructions can be conditional.

ARM		32-bit (conditional)
Thumb	16-bit (unconditional)	
Thumb-2	16-bit (unconditional)	32-bit (unconditional)

- Thumb-2 introduces a conditional execution instruction,
  IT, that is a *logical if-then-else function*
  that you can apply to following instructions
  to make them conditional.

- **I**f cond **T**hen … **E**lse …

```
ITTET EQ
ADD r0,r0,r0
ADD r1,r0,r0
ADD r2,r0,r0
ADD r3,r0,r0
```

```
ITTET EQ
T EQ + ADD r0,r0,r0
T EQ + ADD r1,r0,r0
E EQ + ADD r2,r0,r0
T EQ + ADD r3,r0,r0
```

```
ADDEQ r0,r0,r0  (Always if for 1st one)
ADDEQ r1,r0,r0  (T for 2nd one)
ADDNE r2,r0,r0  (E for 3rd one)
ADDEQ r3,r0,r0  (T for 4th one)
```

https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

Thumb-2 Instructions (4)

- Thumb-2 instructions are accessible
  as were Thumb instructions
  when the processor is in Thumb state,
  that is, the **T** bit in the CPSR is 1
  and the **J** bit in the CPSR is 0.

TJ = 10

- In addition to the 32-bit Thumb instructions,
  there are several 16-bit Thumb instructions
  and a few 32-bit ARM instructions,
  introduced as part of the Thumb-2 architecture.

https://en.wikipedia.org/wiki/Jazelle#Implementation

New 32-bit Thumb Instructions (1-1)

- The new 32-bit Thumb instructions are added in the space previously occupied by the Thumb **BL** and **BLX** instructions.

TJ = 10

- This is made possible by treating **BL** and **BLX** as 32-bit instructions, instead of treating them as two 16-bit instructions.

- This means that **BL** and **BLX**, and all the other 32-bit Thumb instructions, can only take exceptions on their start address.

- They cannot take exceptions at the boundary between *halfword1* and *halfword2* of the instruction.

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

New 32-bit Thumb Instructions (1-2)

- All implementations must ensure
  that both *halfwords* are fetched and consolidated
  before they are issued and executed
  to *comply* with this exception event restriction.

TJ = 10

- This is a change from Thumb.

- Before Thumb-2,
  the two *halfwords* of **BL** and **BLX** instructions
  execute independently,
  and can take exceptions independently.

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

New 32-bit Thumb Instructions (2-1)

- The new 32-bit Thumb instructions are designed for:

- the existing ARM/Thumb Programmers' Model,
  with as few modifications as possible.

TJ = 10

- Certain changes are essential
  to introduce the 32-bit Thumb instructions, notably
  to the Prefetch abort and Undefined Instruction exceptions.

- There is no increase in the number of registers
  (general purpose or special purpose registers), and
  no increase in register sizes.

- existing compiler code generation techniques,
  as far as possible.

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

New 32-bit Thumb Instructions (2-2)

- New concepts are supplementary rather than obligatory.

- For example, literals can still be loaded
  using PC-relative instructions, or
  use in-line immediate values embedded
  in the **MOV** 16-bit immediate and **MOVT** instructions.

TJ = 10

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

New 32-bit Thumb Instructions (3)

- You may not need to rewrite too much
  depending on what features of the ARM instruction set
  and ARM variant you've used.

TJ = 10

- It's also possible that your ARM code is
  already compatible with Thumb-2.

- ARM created Unified Assembly Language (**UAL**)
  once Thumb-2 was introduced
  in order to increase the portability of code.

- it is not a significant deviation
  from ARM assembly of olden days,
  with the biggest change being the introduction of
  the IT(E) directive for conditional execution.

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

New 32-bit Thumb Instructions (4)

- There are some other constructs that won't port directly, and if you are using features of a more advanced or complex ARM core that the Cortex-M4 doesn't have, then that will require a rewrite of that portion.

TJ = 10

- I think if the code is not already written in **ARM UAL** that, while it would take time, it would be relatively simple to run a script over the code that can flag the usage of features that are not written correctly for **UAL**.

- A simple regular expression could check for conditionals on the end of instructions, and it may even be relatively easy to then convert those constructs to use **IT(E)** <cond>.

  - **I**f cond **T**hen … **E**lse …

https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/New-32-bit-Thumb-instructions

Thumb 2 instruction set (4)

- The main enhancements are:

1. 32-bit instructions added to the Thumb instruction set to:
- provide support for exception handling in Thumb state
- provide access to coprocessors
  - include Digital Signal Processing (DSP)
  - and media instructions

2. improve performance in cases
where a single 16-bit instruction *restricts*
functions available to the compiler.

3. addition of a 16-bit IT instruction
that enables *one* to *four* following Thumb instructions,
the IT block, to be conditional

https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

Thumb 2 instruction set (5)

- The main enhancements are:

4. addition of a 16-bit CZB instruction
- Compare with Zero and Branch (CZB)
  to improve code density by replacing two-instruction sequence with a single instruction.

5. The 32-bit ARM Thumb-2 instructions are added
in the space occupied by the Thumb
BL and BLX instructions

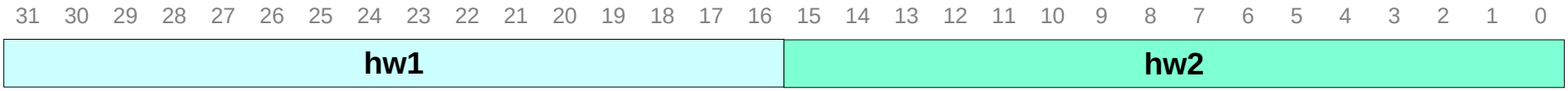
https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

32-bit ARM Thumb-2 Instruction Format (1)

- The first halfword (hw1) determines the instruction length and functionality.

- If the processor decodes the instruction as 32-bit long, then the processor fetches the second halfword (hw2) of the instruction from the instruction address plus two.

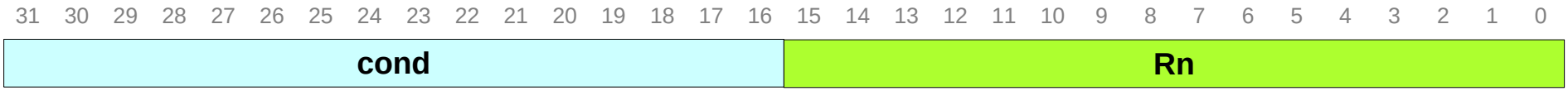
- The availability of both 16-bit Thumb and 32-bit instructions in the Thumb-2 instruction sets, gives you the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications.



https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

32-bit ARM Thumb-2 Instruction Format (2)

- For example, you can code critical loops for applications such as fast interrupts and DSP algorithms using the 32-bit media instructions in Thumb-2 and use the smaller 16-bit classic Thumb instructions for the rest of the application.
  This is for code density and does not require any mode change.



https://developer.arm.com/documentation/ddi0344/c/programmer-s-model/thumb-2-instruction-set

ARM, Thumb, Thumb 2 instruction encodings (1)

- officially there's no "*Thumb-2 instruction set*".

- Ignoring ARMv8
  - where everything is renamed
    and AArch64 complicates things),
- from ARMv4T to ARMv7-A
- there are two instruction sets: ARM and Thumb.

- they are both "32-bit" in the sense that they operate on
  - up-to-32-bit-wide data
  - in 32-bit-wide registers
  - with 32-bit addresses.

- In fact, they represent the exact same instructions

- it is only the instruction encoding which differs

- the CPU has two *different* decode front-ends
  to its pipeline which it can switch between.

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

ARM, Thumb, Thumb 2 instruction encodings (2)

- ARM instructions have
- fixed-width 4-byte encodings
- which require 4-byte alignment.

- Thumb instructions have variable-length
  - 2-byte “narrow” encoding
  - 4-byte "wide" encoding
- requiring 2-byte alignment

- most instructions have 2-byte encodings,
- but **bl** and **blx** have always had 4-byte encodings*.

-

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

ARM, Thumb, Thumb 2 instruction encodings (3)

- The really confusing bit came in ARMv6T2, which introduced "**Thumb-2 Technology**".

- Thumb-2 encompassed not just

  - *adding* a load more instructions to Thumb (mostly with 4-byte encodings) to bring it almost to comparable to ARM,

  - but also *extending* the execution state to allow for conditional execution of most Thumb instructions,

  - and finally introducing a whole new assembly syntax (UAL, "Unified Assembly Language")

    - which *replaced* the previous separate ARM and Thumb syntaxes

    - and allowed *writing* code once and *assembling* it *to either ARM or Thumb instruction set* without modification.

Thumb-2 Technology
4-byte encodings
conditional execution

UAL (Unified Assembly Language)
unify ARM and Thumb syntaxes
assembling to either ARM or Thumb

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

ARM, Thumb, Thumb 2 instruction encodings (4)

- The Cortex-M architectures only implement the Thumb instruction set -

- ARMv7-M (Cortex-M3/M4/M7) supports most of "Thumb-2 Technology", including conditional execution and encodings for VFP instructions,

- whereas ARMv6-M (Cortex-M0/M0+) only uses Thumb-2 in the form of a handful of 4-byte system instructions.

- Thus, the new 4-byte encodings (and those added later in ARMv7 revisions) are still Thumb instructions

- the "Thumb-2" aspect of them is that they can have 4-byte encodings, and that they can (mostly) be conditionally executed via it

  their menmonics are seemed to be only defined in UAL

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

ARM, Thumb, Thumb 2 instruction encodings (7)

- Thumb: 16 bit instruction set
- ARM: 32 bit wide instruction set hence more flexible instructions and less code density
- Thumb2 (mixed 16/32 bit):
  a compromise between ARM and thumb(16) (mixing them), to get both performance/flexibility of ARM and instruction density of Thumb.
- so a Thumb2 instruction can be
  either an ARM (only a subset of) with 32 bit wide instruction
  or a Thumb instruction with 16 bit wide.

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

UAL (Unified Assembly Language) (1-1)

- Unified assembly language (**UAL**) is
  the new assembly syntax introduced by ARM Ltd.
  - to handle the ambiguities introduced
    by the original **Thumb-2** assembly syntax and
  - provide similar syntax for **ARM**, **Thumb** and **Thumb-2**.

- **UAL** is backwards compatible with old **ARM** assembly,
  but incompatible with the **Thumb** assembly syntax.

- **UAL** syntax is the default assembly syntax
  beginning with ARMv7 architectures.

http://downloads.ti.com/docs/esd/SPNU118/unified-assembly-language-syntax-support-spnu1184444.html

UAL (Unified Assembly Language) (1-2)

- When writing assembly code,
  the .arm and .thumb directives are used
  to specify ARM and Thumb UAL syntax, respectively.

- The .state32 and .state16 directives remain
  to specify non-UAL **ARM** and **Thumb** syntax.

- The .arm and .state32 directives are equivalent
  since UAL syntax is backwards compatible in ARM mode.

- Since non-UAL syntax is not supported for **Thumb-2**
  instructions, **Thumb-2** instructions cannot be used
  inside of a .state16 section.

- However, assembly code with .state16 sections that
  contain only non-UAL **Thumb** code can be assembled
  for ARMv7 architectures
  to allow easy porting of older code.

http://downloads.ti.com/docs/esd/SPNU118/unified-assembly-language-syntax-support-spnu1184444.html

UAL (Unified Assembly Language) (2-1)

- the ARM Unified Assembler Language (UAL) syntax provides a canonical form for *all* ARM and Thumb instructions.

- UAL describes the syntax for the mnemonic and the operands of each instruction.

- In addition, it assumes that instructions and data items can be given labels.

- It does not specify the syntax to be used for labels, nor what assembler directives and options are available.

-

https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/Unified-Assembler-Language

UAL (Unified Assembly Language) (2-2)

- Most earlier ARM assembly language mnemonics are still supported as synonyms

- Most earlier Thumb assembly language mnemonics are not supported.

-

https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/Unified-Assembler-Language

UAL (Unified Assembly Language) (3)

- UAL includes instruction selection rules
  that specify which instruction encoding is selected
  when more than one can provide the required functionality.

- For example, both 16-bit and 32-bit encodings exist
  for an **ADD R0, R1, R2** instruction.

- The most common instruction selection rule is that
  when both 16-bit and 32-bit encodings are available,
  the 16-bit encoding is selected, to optimize code density.

- Syntax options exist
  to override the normal instruction selection rules and
  ensure that a particular encoding is selected.

- These are useful when disassembling code,
  to ensure that subsequent assembly produces
  the original code, and in some other situations.

https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/Unified-Assembler-Language

NEON and VFP

- For armv7 ISA (and variants)

- The NEON is a SIMD and parallel data processing unit
  for integer and floating point data

- the VFP is a fully IEEE-754 compatible floating point unit

- In particular on the A8,
  the NEON unit is much faster for just about everything,
- even if you don't have highly parallel data,
  since the VFP is non-pipelined.

- So why would you ever use the VFP?!

- The most major difference is
  that the VFP provides double precision floating point.

- Secondly, there are some specialized instructions
  that that VFP offers that there are no equivalent
  implementations for in the NEON unit.

- SQRT comes to mind, perhaps some type conversions.

https://stackoverflow.com/questions/4097034/arm-cortex-a8-whats-the-difference-between-vfp-and-neon

Jezelle DBX (Direct Bytecode Execution)

Jazelle (1)

- **Jazelle DBX** (direct bytecode execution)
  is an extension that allows some ARM processors
  to execute Java bytecode in hardware
  as a third execution state
  alongside the existing ARM and Thumb modes.

TJ = 10

- Jazelle functionality was specified
  in the ARMvTEJ architecture

- the first processor with Jazelle technology
  was the ARM926EJ-S.

- Jazelle is denoted by a "**J**" appended to the CPU name
  except for post-v5 cores where it is required
  (albeit only in trivial form) for architecture conformance.

https://en.wikipedia.org/wiki/Jazelle#Implementation

Jazelle (2)

- The **J bit**
  - The **J bit** in the CPSR indicates when the processor is in Jazelle state.
  - When J = 0
    the processor is in ARM or Thumb state, depending on the T bit.
  - When J = 1
    the processor is in Jazelle state.

TJ = 00	ARM
TJ = 10	Thumb
TJ = 01	Jazelle
TJ = 11	undef

https://developer.arm.com/documentation/ddi0301/h/programmer-s-model/the-program-status-registers/the-j-bit

Jazelle (3)

- The combination of **J = 1** and **T = 1** causes similar effects to setting **T=1** on a non Thumb-aware processor.

- That is, the next instruction executed causes entry to the **Undefined** Instruction exception.

- entry to the exception handler causes the processor to re-enter ARM state, and

- the handler can detect that this was the cause of the exception because **J** and **T** are both set in **SPSR_und**.

- **MSR** cannot be used to change the **J bit** in the CPSR.

TJ = 00	ARM
TJ = 10	Thumb
TJ = 01	Jazelle
TJ = 11	undef

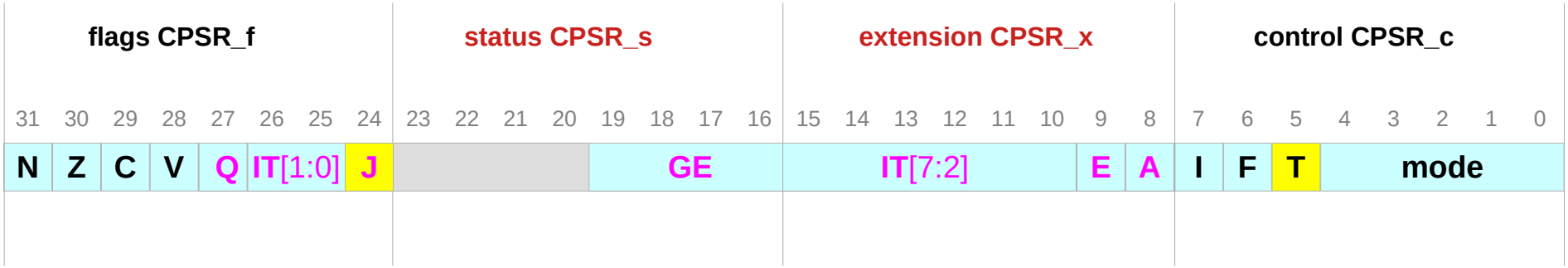
https://developer.arm.com/documentation/ddi0301/h/programmer-s-model/the-program-status-registers/the-j-bit

Jazelle (4)

- The placement of the **J bit** avoids the status or extension bytes in code running on ARMv5TE or earlier processors.

- This ensures that OS code written using the deprecated syntax CPSR, SPSR, CPSR_all, or SPSR_all for the destination of an **MSR** instruction continues to work.

- The **MSR** instruction is used to write
  - to the **CPSR** or
  - to the **SPSR** of the current mode.



Current Program Status Register (CPSR)

https://developer.arm.com/documentation/ddi0301/h/programmer-s-model/the-program-status-registers/the-j-bit

CPSR Bits (1)

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**

the **T bit** shows whether the processor runs in ARM state or in Thumb state.

never set this bit

can be changed only in a privileged mode

To disable Interrupt (**IRQ**), set **I**

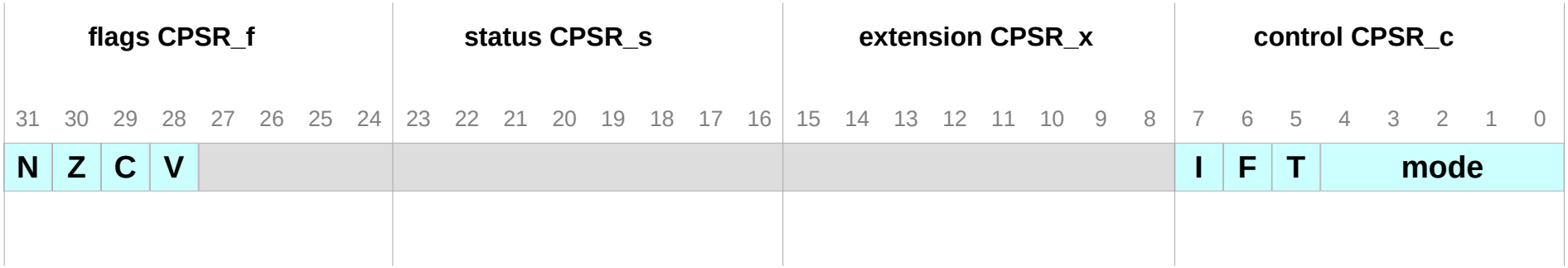
To disable Fast Interrupt (**FIQ**), set **F**

the **T bit** shows whether the processor runs in ARM state or in Thumb state.

never set this bit

can be changed only in a privileged mode

USR	10000
FIQ	10001
IRQ	10010
SVC	10011
ABT	10111
UND	11011
SYS	11111

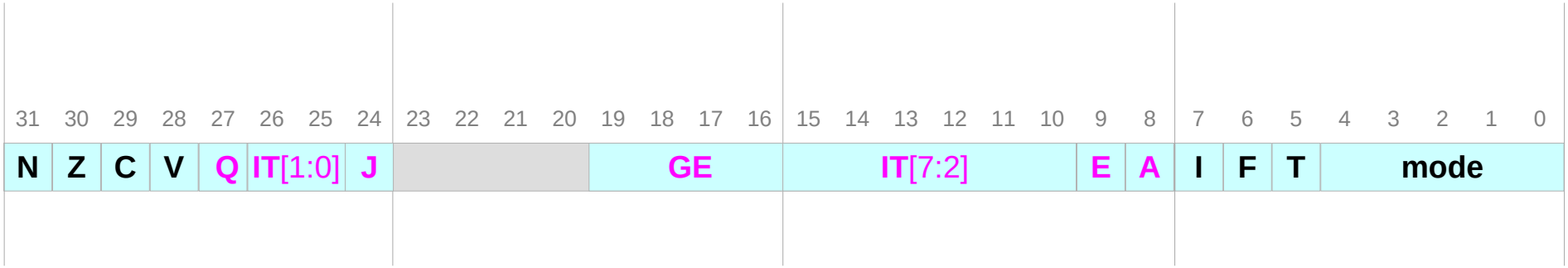


Current Program Status Register (CPSR)

https://developer.arm.com/documentation/ddi0301/h/programmer-s-model/the-program-status-registers/the-j-bit
https://courses.washington.edu/cp105/02_Exceptions/Status_Register_Instructions.html

CPSR Bits (2)

Q	Cumulative saturation bit	**GE**	greater than or equal to flags	**E**	Endianness execution state bit
IT[1:0]	if-Then exectuion state bits	**IT[7:2]**	if-Then exectuion state bits		0 - Little-endian, 1 - Big-endian
	for the Thumb IT (If-Then) instruction		for the Thumb IT (If-Then) instruction	**A**	Asynchronous abort mask bit
J	Jazelle bit				



Current Program Status Register (CPSR)

https://www.keil.com/pack/doc/CMSIS/Core_A/html/group__CMSIS__CPSR.html

MRS – Move to Register from Status

- **MRS** is use to read
  - from the **CPSR** or
  - from the **SPRS** of the current mode

- It move the value from the status register
  into a regular register.

- The **SPSR** that will be read is
  the one that is active for the CPU's current mode.

  MRS R0, CPSR
  MRS R1, SPSR

- Reading the **SPSR** while in user or system mode
  is not valid and yields unpredictable results.

https://courses.washington.edu/cp105/02_Exceptions/Status_Register_Instructions.html

MSR – Move to Status from Register

- The **MSR** instruction is used to write
  - to the **CPSR** or
  - to the **SPSR** of the current mode.

- Writing to the **SPSR** while in the user or system mode is not valid and the results are not predictable.

- Any writes to the **CPSR** in user mode are ignored.
- The **CPSR** can only be written to in a priveleged mode.

- **MSR CPSR, R0**
- **MSR SPSR, R1**

https://courses.washington.edu/cp105/02_Exceptions/Status_Register_Instructions.html

64-bit Processors

A32 + T32 ISA's
A64 ISA

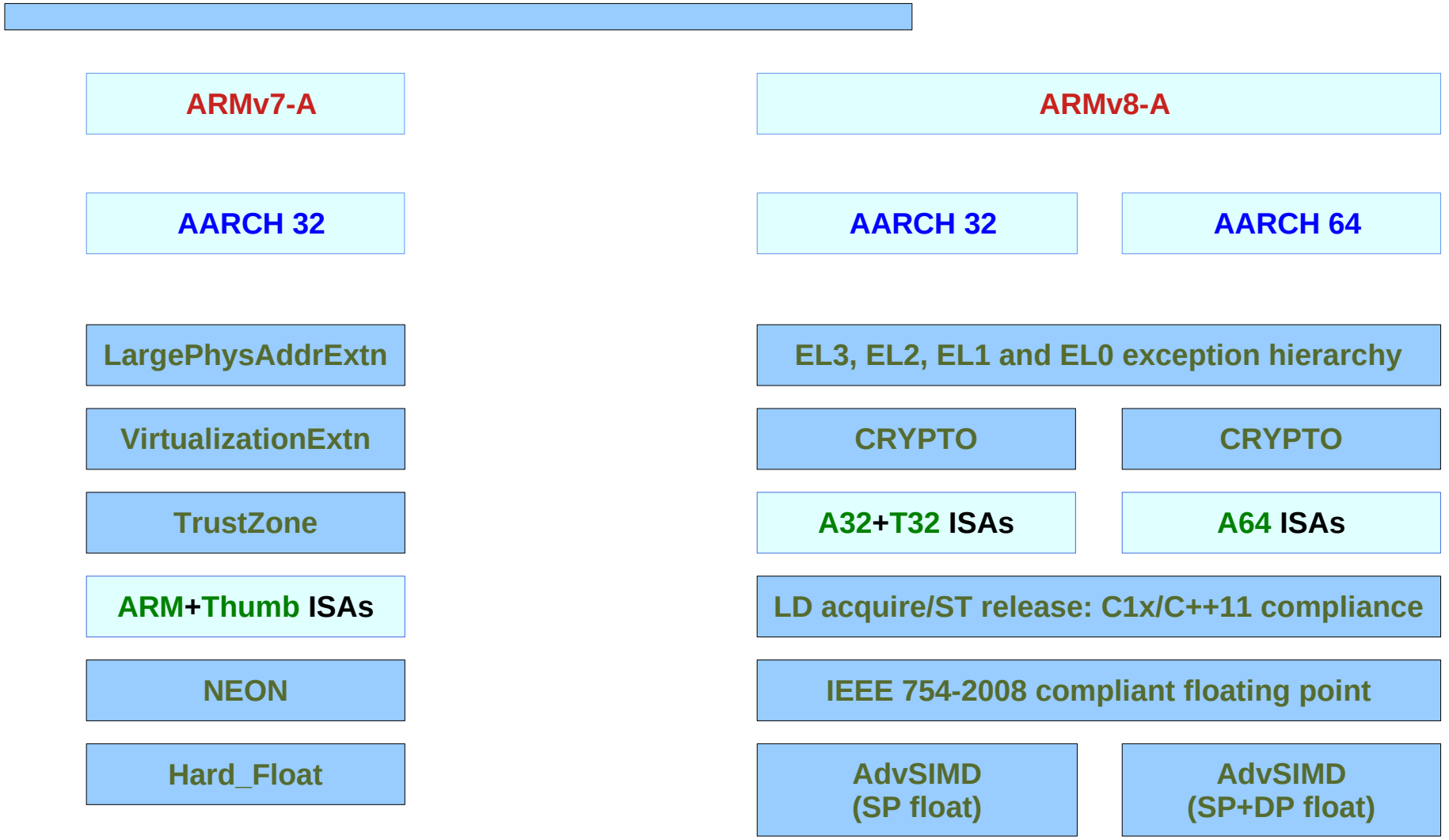
64-bit processor (1)

ARMv7-A	ARMv8-A	
AARCH 32	AARCH 32	AARCH 64
ARM+Thumb ISAs	A32+T32 ISAs	A64 ISAs

ARMv7-A AARCH32 ARM+Thumb ISAs	ARMv8-A AARCH32 A32+T32 ISAs	ARMv8-A AARCH64 A64 ISAs

https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

64-bit processor (1)



https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

ARM, Thumb, Thumb 2 instruction encodings (5)

- there is a 32-bit execution state (AArch32) and a 64-bit execution state (AArch64).

- the 32-bit execution state supports two different instruction sets:
  - T32 ("Thumb") and
  - A32 ("ARM").

- The 64-bit execution state supports only one instruction set - A64.

- All A64, like all A32, instructions are 32-bit (4 byte) in size, requiring 4-byte alignment.

- Many/most A64 instructions can operate on both 32-bit and 64-bit registers (or arguably 32-bit or 64-bit views of the same underlying 64-bit register).

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

ARM, Thumb, Thumb 2 instruction encodings (6)

- All ARMv8 processors (like all ARMv7 processors) that implement AArch32 support Thumb-2 instructions in the T32 instruction set.

- Not all ARMv8-A processors implement AAarch32, and some don't implement AArch64.

- Some Processors support both, but only support AArch32 at lower exception levels.

https://stackoverflow.com/questions/28669905/what-is-the-difference-between-the-arm-thumb-and-thumb-2-instruction-encodings

64-bit processor (1)

- Evolution of the ARM architecture

- The diagram shows how all the features present in **ARMv7-A** have been carried forward into **ARMv8-A**.

- But **ARMv8** supports two execution states:
  - **AArch32**
    the **A32** and **T32** instruction sets
    (**ARM** and **Thumb** in **ARMv7-A**) are supported
  - **AArch64**
    the new **A64** instruction set is introduced.

- Although backwards compatible with **ARMv7-A**,
  the exception, privilege and security model
  has been significantly *extended* and
  is now classified as a set of exception levels,
  EL0 to EL3, in a four-level hierarchy.

ARMv7-A
AARCH32
ARM+Thumb ISAs

ARMv8-A
AARCH32
A32+T32 ISAs,
AARCH64
A64 ISAs

https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

64-bit processor (2)

- In **AArch32**,
  the **ARMv7-A** Large Physical Address Extensions
  are supported, providing
  - 32-bit virtual addressing and
  - 40-bit physical addressing.

- In **AArch64**,
  this is extended, again in a backward compatible way,
  to provide
  - 64-bit virtual addresses and
  - 48-bit physical address

- Other additions include cryptographic support
  at instruction level.

ARMv7-A
AARCH32
ARM+Thumb ISAs

ARMv8-A
AARCH32,
A32+T32 ISAs,
AARCH64
A64 ISAs

https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

64-bit processor (3)

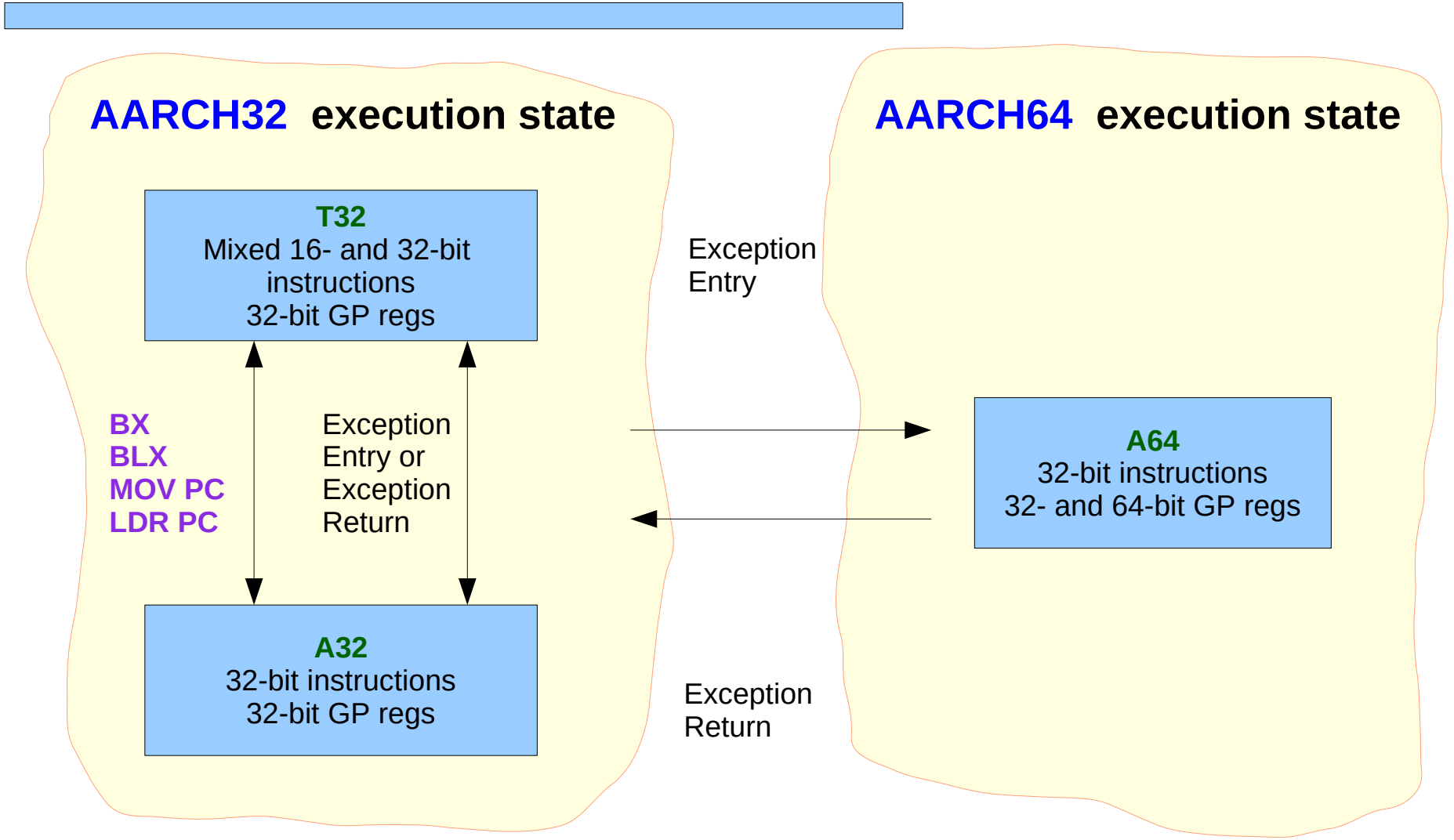
- Overview of **AArch64** in **ARMv8-A**
- The **A64** instruction set, defined in **AArch64**, has been designed from the ground up as a clean, modern instruction set which operates on 64-bit or 32-bit native datatypes or registers.
- **A64** is a fixed-length instruction set in which all instructions are 32 bits in length.
- It does, as you might expect, have many similarities with the **A32** instruction set which you'll be familiar with from earlier ARM architectures.
- There are some things you'll find which are new and some things which you'll go looking for and aren't there!

ARMv7-A
AARCH32
ARM+Thumb ISAs

ARMv8-A
AARCH32,
A32+T32 ISAs,
AARCH64
A64 ISAs

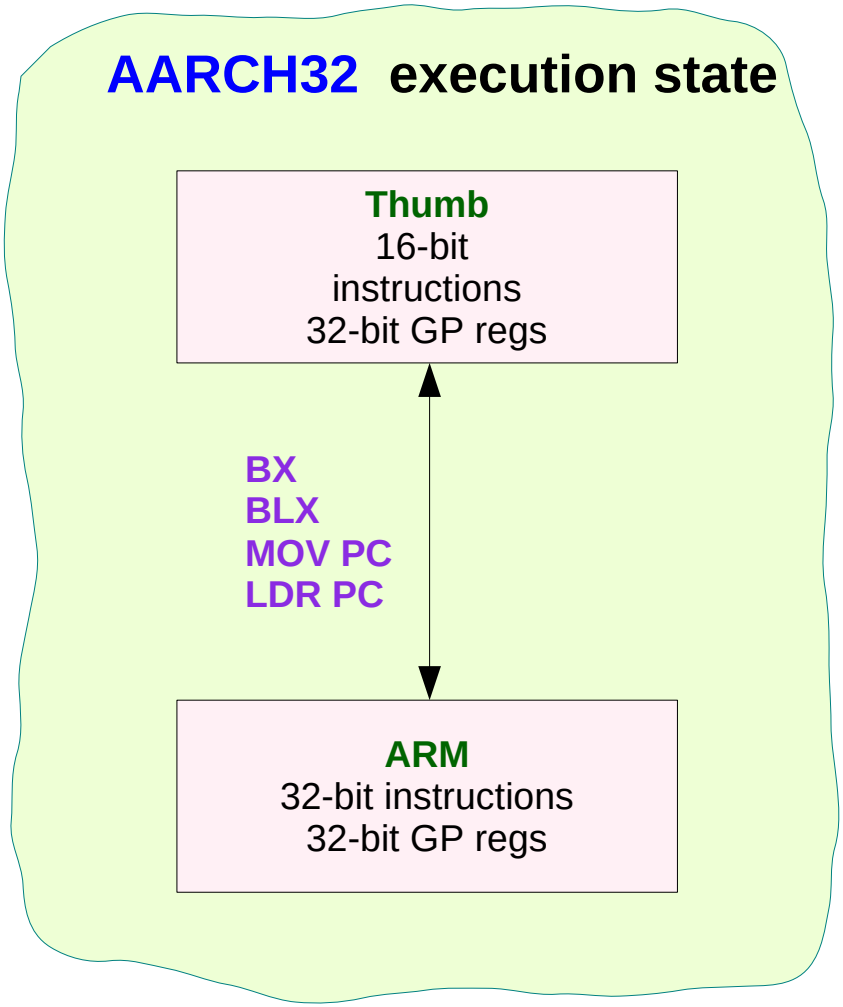
https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

64-bit processor (4)



https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

64-bit processor (5)

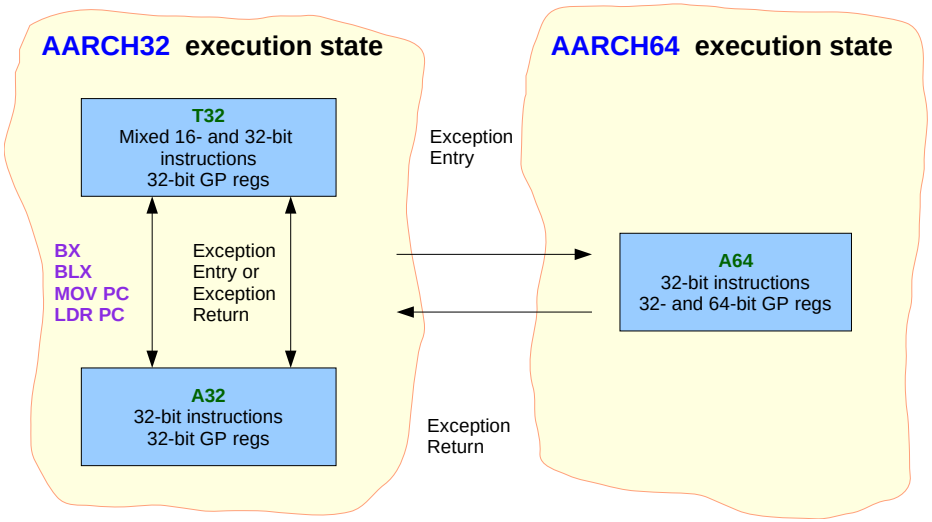


https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

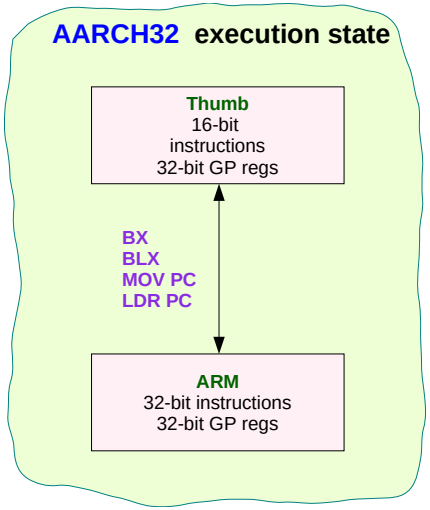
64-bit processor (6)

Changing Execution state and Instruction set

- A fully-populated **ARMv8-A** processor supports both **AArch32** and **Aarch64** execution states.
- Transition between the two is always across an exception boundary.



- This differs from **ARMv7-A** in which a change of instruction set is triggered by an interworking branch (e.g. BLX).



https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

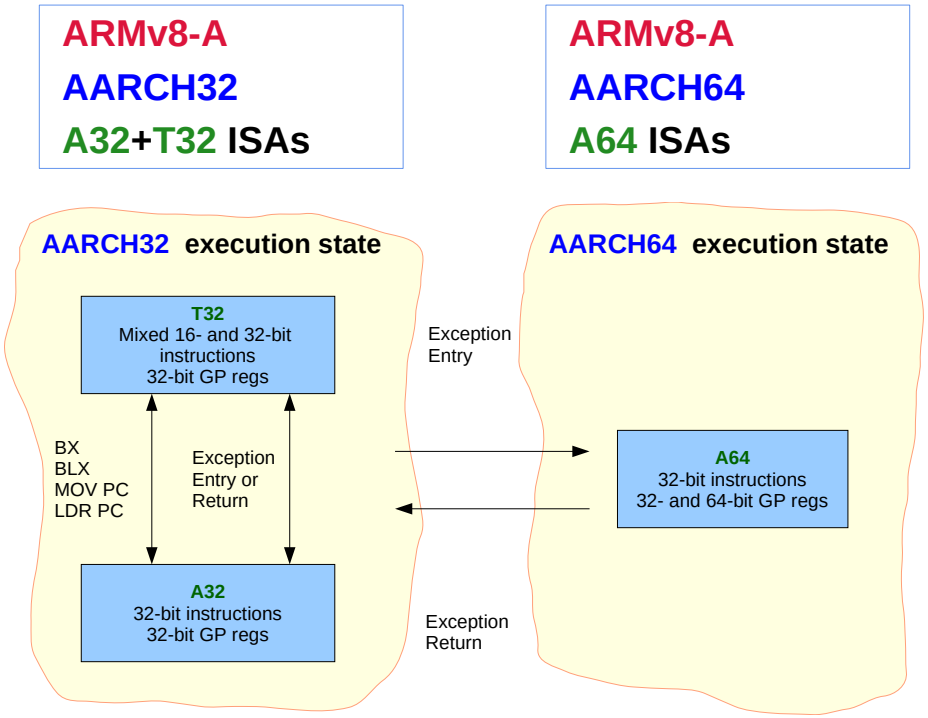
64-bit processor (7)

Changing Execution state and Instruction set

- the relationship between
  the **T32**, **A32** and **A64** instruction sets and
- the events which can cause a switch between them.

- the execution state
  - can stay the same or
  - go from 32-bit to 64-bit
    - when taking an exception, or
    - when returning from an exception

- This introduces a natural hierarchy
  of 64-bit and 32-bit support at each level



https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Porting%20to%20ARM%2064-bit.pdf

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