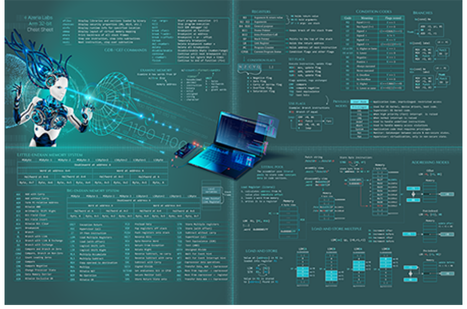
CONDITIONAL EXECUTION

We already briefly touched the conditions' topic while discussing the CPSR register. We use conditions for controlling the program's flow during it's runtime usually by making jumps (branches) or executing some instruction only when a condition is met. The condition is described as the state of a specific bit in the CPSR register. Those bits change from time to time based on the outcome of some instructions. For example, when we compare two numbers and they turn out to be equal, we trigger the Zero bit (Z = 1), because under the hood the following happens: a – b = 0. In this case we have **EQ**ual condition. If the first number was bigger, we would have a **G**reater **T**han condition and in the opposite case – **L**ower **T**han. There are more conditions, like **L**ower or **E**qual (LE), **G**reater or **E**qual (GE) and so on.

The following table lists the available condition codes, their meanings, and the status of the flags that are tested.

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) || (N!=V)

Condition Code	Meaning (for cmp or subs)	Status of Flags
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0
MI	Negative (or Minus)	N==1
PL	Positive (or Plus)	N==0
AL	Always executed	–
NV	Never executed	–
VS	Signed Overflow	V==1
VC	No signed Overflow	V==0
HI	Unsigned Higher	(C==1) && (Z==0)
LS	Unsigned Lower or same	(C==0) | | (Z==0)



We can use the following piece of code to look into a practical use case of conditions where we perform **conditional addition**.

```
.global main

main:
        mov     r0, #2     /* setting up initial variable */
        cmp     r0, #3     /* comparing r0 to number 3. Negative bit get's set to 1 */
        addlt   r0, r0, #1 /* increasing r0 IF it was determined that it is smaller (lower than) numbe
        cmp     r0, #3     /* comparing r0 to number 3 again. Zero bit gets set to 1. Negative bit is 
        addlt   r0, r0, #1 /* increasing r0 IF it was determined that it is smaller (lower than) numbe
        bx      lr
```

The first CMP instruction in the code above triggers **N**egative bit to be set (2 – 3 = -1) indicating that the value in r0 is **L**ower **T**han number 3. Subsequently, the ADDLT instruction is executed because LT condition is full filled when V != N (values of overflow and negative bits in the CPSR are different). Before we execute second CMP, our r0 = 3. That's why second CMP clears out Negative bit (because 3 – 3 = 0, no need to set the negative flag) and sets the **Z**ero flag (Z = 1). Now we have V = 0 and N = 0 which results in LT condition to fail. As a result, the second ADDLT is not executed and r0 remains unmodified. The program exits with the result 3.

CONDITIONAL EXECUTION IN THUMB

In the Instruction Set chapter we talked about the fact that there are different Thumb versions. Specifically, the Thumb version which allows conditional execution (Thumb-2). Some ARM processor versions support the "IT" instruction that allows up to 4 instructions to be executed conditionally in Thumb state.

Reference: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABIJDIC.html

Syntax: IT{*x*{*y*{*z*}}} *cond*

- *cond* specifies the condition for the **first** instruction in the IT block
- *x* specifies the condition switch for the **second** instruction in the IT block
- *y* specifies the condition switch for the **third** instruction in the IT block
- *z* specifies the condition switch for the **fourth** instruction in the IT block

The structure of the IT instruction is "IF-Then-(Else)" and the syntax is a construct of the two letters T and E:

- IT refers to If-Then (next instruction is conditional)
- ITT refers to If-Then-Then (next 2 instructions are conditional)
- ITE refers to If-Then-Else (next 2 instructions are conditional)
- ITTE refers to If-Then-Then-Else (next 3 instructions are conditional)
- ITTEE refers to If-Then-Then-Else-Else (next 4 instructions are conditional)

Each instruction inside the IT block must specify a condition suffix that is either the same or logical inverse. This means that if you use ITE, the first and second instruction (If-Then) must have the same condition suffix and the third (Else) must have the logical inverse of the first two. Here are some examples from the ARM reference manual which illustrates this logic:

```
ITTE   NE           ; Next 3 instructions are conditional
ANDNE  R0, R0, R1   ; ANDNE does not update condition flags
ADDSNE R2, R2, #1   ; ADDSNE updates condition flags
MOVEQ  R2, R3       ; Conditional move

ITE    GT           ; Next 2 instructions are conditional
```

```
ADDGT  R1, R0, #55  ; Conditional addition in case the GT is true
ADDLE  R1, R0, #48  ; Conditional addition in case the GT is not true

ITTEE  EQ           ; Next 4 instructions are conditional
MOVEQ  R0, R1       ; Conditional MOV
ADDEQ  R2, R2, #10  ; Conditional ADD
ANDNE  R3, R3, #1   ; Conditional AND
BNE.W  dloop        ; Branch instruction can only be used in the last instruction of an IT block
```

Wrong syntax:

```
IT     NE           ; Next instruction is conditional
ADD    R0, R0, R1   ; Syntax error: no condition code used in IT block.
```

Here are the conditional codes and their opposite:

Condition Code		**Opposite**	
Code	**Meaning**	**Code**	**Meaning**
EQ	Equal	NE	Not Equal
HS (or CS)	Unsigned higher or same (or carry set)	LO (or CC)	Unsigned lower (or carry clear)
MI	Negative	PL	Positive or Zero
VS	Signed Overflow	VC	No Signed Overflow
HI	Unsigned Higher	LS	Unsigned Lower or Same
GE	Signed Greater Than or Equal	LT	Signed Less Than
GT	Signed Greater Than	LE	Signed Less Than or Equal
AL (or omitted)	Always Executed	There is no opposite to AL	

Let's try this out with the following example code:

```
.syntax unified    @ this is important!
.text
.global _start

_start:
    .code 32
    add r3, pc, #1   @ increase value of PC by 1 and add it to R3
    bx r3            @ branch + exchange to the address in R3 -> switch to Thumb state because LSB = 1

    .code 16         @ Thumb state
    cmp r0, #10
    ite eq           @ if R0 is equal 10...
    addeq r1, #2     @ ... then R1 = R1 + 2
    addne r1, #3     @ ... else R1 = R1 + 3
    bkpt
```

.code 32

This example code starts in ARM state. The first instruction adds the address specified in PC plus 1 to R3 and then branches to the address in R3. This will cause a switch to Thumb state, because the LSB (least significant bit) is 1 and therefore not 4 byte aligned. It's important to use bx (branch + exchange) for this purpose. After the branch the T (Thumb) flag is set and we are in Thumb state.

.code 16

In Thumb state we first compare R0 with #10, which will set the Negative flag (0 – 10 = – 10). Then we use an If-Then-Else block. This block will skip the ADDEQ instruction because the Z (Zero) flag is not set and will execute the ADDNE instruction because the result was NE (not equal) to 10.

Stepping through this code in GDB will mess up the result, because you would execute both instructions in the ITE block. However running the code in GDB without setting a breakpoint and stepping through each instruction will yield to the correct result setting R1 = 3.

BRANCHES

Branches (aka Jumps) allow us to jump to another code segment. This is useful when we need to skip (or repeat) blocks of codes or jump to a specific function. Best examples of such a use case are IFs and Loops. So let's look into the IF case first.

```
.global main

main:
        mov     r1, #2     /* setting up initial variable a */
        mov     r2, #3     /* setting up initial variable b */
        cmp     r1, r2     /* comparing variables to determine which is bigger */
        blt     r1_lower   /* jump to r1_lower in case r2 is bigger (N==1) */
        mov     r0, r1     /* if branching/jumping did not occur, r1 is bigger (or the same) so store
        b       end        /* proceed to the end */
r1_lower:
        mov r0, r2         /* We ended up here because r1 was smaller than r2, so move r2 into r0 */
        b end              /* proceed to the end */
end:
        bx lr              /* THE END */
```

The code above simply checks which of the initial numbers is bigger and returns it as an exit code. A C-like pseudo-code would look like this:

```
int main() {
   int max = 0;
   int a = 2;
   int b = 3;
   if(a < b) {
    max = b;
   }
   else {
    max = a;
   }
   return max;
}
```

Now here is how we can use conditional and unconditional branches to create a loop.

```
.global main

main:
        mov     r0, #0     /* setting up initial variable a */
loop:
        cmp     r0, #4     /* checking if a==4 */
        beq     end        /* proceeding to the end if a==4 */
        add     r0, r0, #1 /* increasing a by 1 if the jump to the end did not occur */
        b loop             /* repeating the loop */
end:
        bx lr              /* THE END */
```

A C-like pseudo-code of such a loop would look like this:

```
int main() {
   int a = 0;
   while(a < 4) {
   a= a+1;
   }
   return a;
}
```

B / BX / BLX

There are three types of branching instructions:

- Branch (**B**)
  - Simple jump to a function
- Branch link (**BL**)
  - Saves (PC+4) in LR and jumps to function
- Branch exchange (**BX**) and Branch link exchange (**BLX**)
  - Same as B/BL + exchange instruction set (ARM <-> Thumb)

- Needs a register as first operand: BX/BLX reg

BX/BLX is used to exchange the instruction set from ARM to Thumb.

```
.text
.global _start

_start:
     .code 32         @ ARM mode
     add r2, pc, #1   @ put PC+1 into R2
     bx r2            @ branch + exchange to R2

    .code 16          @ Thumb mode
     mov r0, #1
```

The trick here is to take the current value of the actual PC, increase it by 1, store the result to a register, and branch (+exchange) to that register. We see that the addition (add r2, pc, #1) will simply take the effective PC address (which is the current PC register's value + 8 -> 0x805C) and add 1 to it (0x805C + 1 = 0x805D). Then, the exchange happens if the Least Significant Bit (LSB) of the address we branch to is 1 (which is the case, because 0x805D = 10000000 0101110**1**), meaning the address is not 4 byte aligned. Branching to such an address won't cause any misalignment issues. This is how it would look like in GDB (with GEF extension):

Please note that the GIF above was created using the older version of GEF so it’s very likely that you see a slightly different UI and different offsets. Nevertheless, the logic is the same.

Conditional Branches

Branches can also be executed conditionally and used for branching to a function if a specific condition is met. Let’s look at a very simple example of a conditional branch suing BEQ. This piece of assembly does nothing interesting other than moving values into registers and branching to another function if a register is equal to a specified value.

```
.text
.global _start
```

```
_start:
   mov r0, #2
   mov r1, #2
   add r0, r0, r1
   cmp r0, #4
   beq func1
   add r1, #5
   b func2
func1:
   mov r1, r0
   bx  lr
func2:
   mov r0, r1
   bx  lr
```

‹ PART 5: LOAD/STORE MULTIPLE

PART 7: STACK AND FUNCTIONS ›