AZERIA

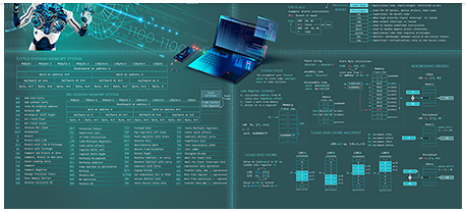
ARM & THUMB

ARM processors have two main states they can operate in (let's not count Jazelle here), ARM and Thumb. These states have nothing to do with privilege levels. For example, code running in SVC mode can be either ARM or Thumb. The main difference between these two states is the instruction set, where instructions in ARM state are always 32-bit, and instructions in Thumb state are 16-bit (but can be 32-bit). Knowing when and how to use Thumb is especially important for our ARM exploit development purposes. When writing ARM shellcode, we need to get rid of NULL bytes and using 16-bit Thumb instructions instead of 32-bit ARM instructions reduces the chance of having them.

The calling conventions of ARM versions is more than confusing and not all ARM versions support the same Thumb instruction sets. At some point, ARM introduced an enhanced Thumb instruction set (pseudo name: Thumbv2) which allows 32-bit Thumb instructions and even conditional execution, which was not possible in the versions prior to that. In order to use conditional execution in Thumb state, the "it" instruction was introduced. However, this instruction got then removed in a later version and exchanged with something that was supposed to make things less complicated, but achieved the opposite. I don't know all the different variations of ARM/Thumb instruction sets across all the different ARM versions, and I honestly don't care. Neither should you. The only thing that you need to know is the ARM version of your target device and its specific Thumb support so that you can adjust your code. The ARM Infocenter should help you figure out the specifics of your ARM version (http://infocenter.arm.com/help/index.jsp).

As mentioned before, there are different Thumb versions. The different naming is just for the sake of differentiating them from each other (the processor itself will always refer to it as Thumb).

- Thumb-1 (16-bit instructions): was used in ARMv6 and earlier architectures.
- Thumb-2 (16-bit and 32-bit instructions): extents Thumb-1 by adding more instructions and allowing them to be either 16-bit or 32-bit wide (ARMv6T2, ARMv7).
- ThumbEE: includes some changes and additions aimed for dynamically generated code (code compiled on the device either shortly before or during execution).



- Conditional execution: All instructions in ARM state support conditional execution. Some ARM processor versions allow conditional execution in Thumb by using the IT instruction. Conditional execution leads to higher code density because it reduces the number of instructions to be executed and reduces the number of expensive branch instructions.
- 32-bit ARM and Thumb instructions: 32-bit Thumb instructions have a .w suffix.
- The barrel shifter is another unique ARM mode feature. It can be used to shrink multiple instructions into one. For example, instead of using two instructions for a multiply (multiplying register by 2 and using MOV to store result into another register), you can include the multiply inside a MOV instruction by using shift left by 1 -> Mov R1, R0, LSL #1 ; R1 = R0 * 2

To switch the state in which the processor executes in, one of two conditions have to be met:

- We can use the branch instruction BX (branch and exchange) or BLX (branch, link, and exchange) and set the destination register's least significant bit to 1. This can be achieved by adding 1 to an offset, like 0x5530 + 1. You might think that this would cause alignment issues, since instructions are either 2- or 4-byte aligned. This is not a problem because the processor will ignore the least significant bit. More details in Part 6: Conditional Execution and Branching.
- We know that we are in Thumb mode if the T bit in the current program status register is set.

INTRODUCTION TO ARM INSTRUCTIONS

The purpose of this part is to briefly introduce into the ARM's instruction set and it's general use. It is crucial for us to understand how the smallest piece of the Assembly language operates, how they connect to each other, and what can be achieved by combining them.

As mentioned earlier, Assembly language is composed of instructions which are the main building blocks. ARM instructions are usually followed by one or two operands and generally use the following template:

```
MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
```

Due to flexibility of the ARM instruction set, not all instructions use all of the fields provided in the template. Nevertheless, the purpose of fields in the template are described as follows:

```
MNEMONIC     - Short name (mnemonic) of the instruction
{S}          - An optional suffix. If S is specified, the condition flags are updated on the result of
{condition}  - Condition that is needed to be met in order for the instruction to be executed
```

```
Operand2     - Second (flexible) operand. Can be an immediate value (number) or a register with an opt
```

While the MNEMONIC, S, Rd and Operand1 fields are straight forward, the condition and Operand2 fields require a bit more clarification. The condition field is closely tied to the CPSR register's value, or to be precise, values of specific bits within the register. Operand2 is called a flexible operand, because we can use it in various forms – as immediate value (with limited set of values), register or register with a shift. For example, we can use these expressions as the Operand2:

```
#123                - Immediate value (with limited set of values).
Rx                  - Register x (like R1, R2, R3 ...)
Rx, ASR n           - Register x with arithmetic shift right by n bits (1 = n = 32)
Rx, LSL n           - Register x with logical shift left by n bits (0 = n = 31)
Rx, LSR n           - Register x with logical shift right by n bits (1 = n = 32)
Rx, ROR n           - Register x with rotate right by n bits (1 = n = 31)
Rx, RRX             - Register x with rotate right by one bit, with extend
```

As a quick example of how different kind of instructions look like, let's take a look at the following list.

```
ADD   R0, R1, R2         - Adds contents of R1 (Operand1) and R2 (Operand2 in a form of register) and
ADD   R0, R1, #2         - Adds contents of R1 (Operand1) and the value 2 (Operand2 in a form of an im
MOVLE R0, #5             - Moves number 5 (Operand2, because the compiler treats it as MOVLE R0, R0, #
MOV   R0, R1, LSL #1     - Moves the contents of R1 (Operand2 in a form of register with logical shift
```

As a quick summary, let's take a look at the most common instructions which we will use in future examples.

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load

ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

‹ PART 2: DATA TYPES AND REGISTERS

PART 4: LOAD AND STORE ›