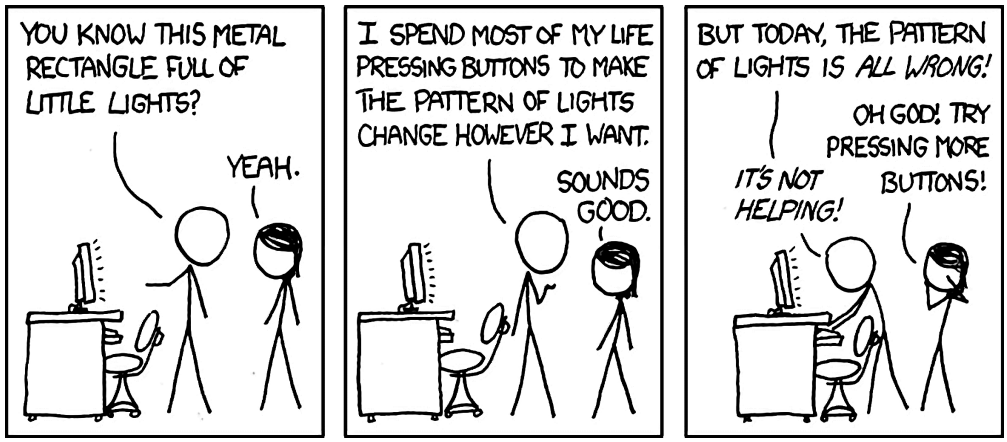
ARM-Thumb Machine Code Reference

Michael Lyle

2024 Edition

Contents



XKCD #722, "Computer Problems" (by Randall Munroe)

1 Architecture Summary

In this course, we will be learning ARM Thumb machine language.

Specifically, we are writing code for the ARMv7 Thumb architecture. Its characteristics are:

- 32-bit (4 byte) register (word) size
- 32-bit (4 byte) address size
- 16-bit (2 byte) instruction length
- Addresses point to individual bytes. Instructions and words should be aligned in memory.
  - An address pointing aligned with the beginning of a word will end in 0, 4, 8, or C in base 16.
  - An address pointing to an instruction will end in 0, 2, 4, 6, 8, A, C, or E in base 16.
- Instructions can access memory, or perform arithmetic, but not both (RISC / load-store machine).
- Eight general-purpose registers are easily accessed ($r0$-$r7$).
- Five additional general-purpose registers are more difficult to access ($r8$-$r12$)
- Three special purpose registers (stack pointer, link register, and program counter)
- Capable ALU with hardware multiplier and standard flags:
  - N (negative),
  - Z (zero),
  - C (carry),
  - and V (overflow)

1.1 Machine Language vs. Assembly Language

In this course, we are writing programs in machine language. This is a raw sequence of numbers that tells the computer what to do.

One step up from machine language is to use assembly language, where each line of code corresponds to a single machine instruction, which the assembler then translates into the binary format of machine language. Assemblers make the process of programming slightly easier by handling the details of instruction encoding and calculating addresses for the programmer.

Most contemporary programmers work with high-level languages, such as *C*, *Java*, or *Python*. These languages require conversion to machine code for execution. This is typically done through compilation, where the high-level code is translated into machine code, or through interpretation, where another program executes the high-level code step-by-step.

2 Using the ARM Microprocessor Trainer

You will be receiving an Microprocessor Trainer in this course. Before home computers were readily available, this was one way that many people learned to program. The Trainer has a keypad where you can inspect and manipulate the contents of memory, as well as run your program.

Unlike earlier trainers, the system designed for this class uses a modern instruction set, **ARM-Thumb**, which is still actively used on microcontrollers. It is closely related to the **ARM** instruction set used in modern phones, tablets, and Apple computers.

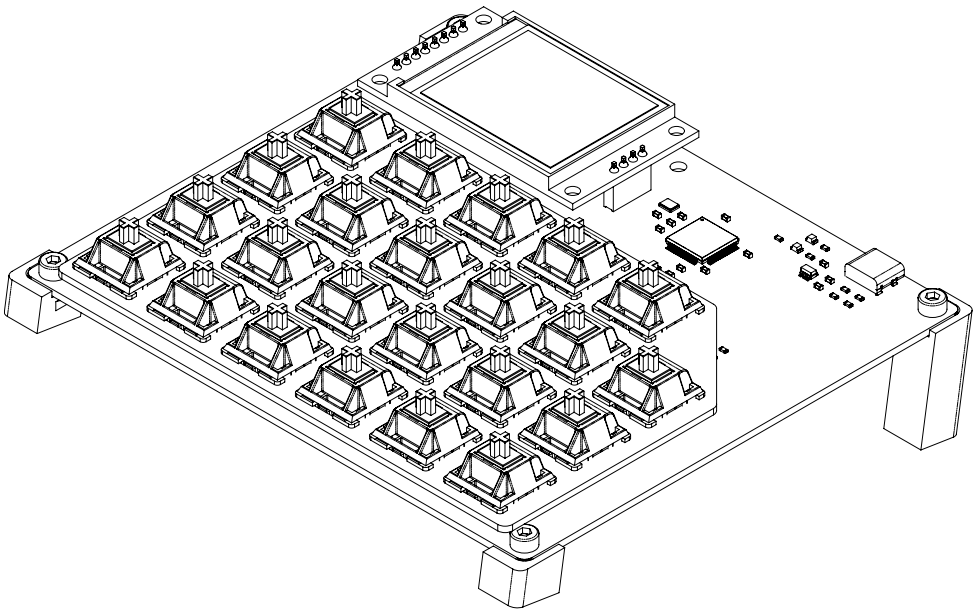


Figure 1: ARM Microprocessor Trainer, Revision B

If you've programmed before, this will be more cumbersome than you're used to! It's important to pre-plan your programs and think about what each instruction will do.

2.1 Description of Keys

- **0-9, A-F**: Represent hexadecimal digits, which are used to enter numbers into the trainer.
- **CLR**: Functions as a backspace key, allowing you to correct any mistakes made during input.
- **ADDR**: Toggles the cursor between editing the memory address and the value at that address.
- **LOAD**: Loads the value located at the specified address into the display. Pressing this key repeatedly will advance the address sequentially, allowing you to view subsequent values.

- **STOR**: Saves the currently displayed value into the specified memory address, effectively writing data to memory. This will also advance the address.
- **STEP**: Executes the instruction at the current memory address and then automatically advances to the next address, facilitating step-by-step execution of your code.
- **RUN**: Starts a continuous execution of the program beginning at the current address. The execution will proceed without interruption until you press **STEP**.

2.2 Screen Contents

The screen provides an overview of the current status of the microprocessor trainer.

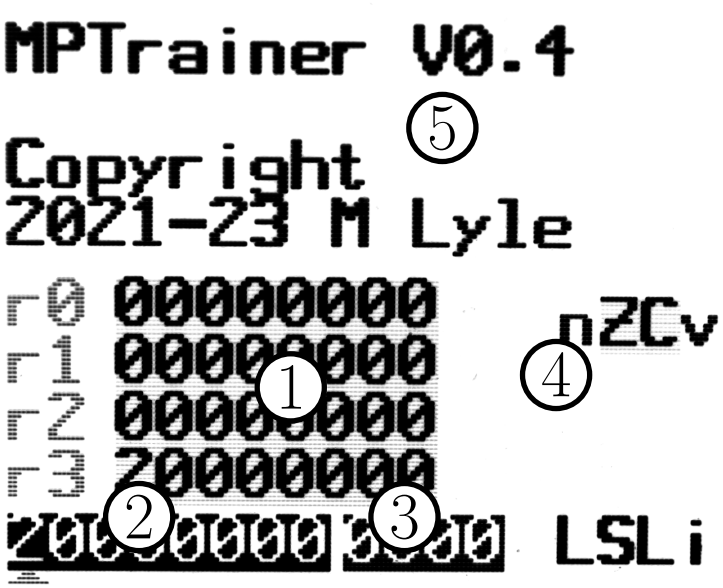


Figure 2: Screen Contents at Startup

Various types of information are displayed on the screen:

1. The values of registers are displayed centrally.
2. The current address is displayed on the left. A red arrow indicates we are currently editing the first digit of the address.
3. The value (that may be loaded or stored) is displayed to the right of the address.
4. The current flags (produced by the ALU and used by conditional branches) are displayed on the right side of the screen.
5. At the top of the screen is where a program's input/output is displayed.

2.3 Entering Instructions

Once you have written your program and translated it into hexadecimal format, you will need to enter these instructions into the Microprocessor Trainer. Follow these steps to input each instruction:

1. Turn on the Microprocessor Trainer and locate the keypad.
2. Begin by setting the starting address where your program will be stored. Enter `20000000` using the numeric keys (0-9, A-F).
3. Press the **ADDR** key to switch the focus from entering an address to entering a value.
4. Now, enter the first instruction from your program. Remember, each instruction should be a 4-digit hexadecimal number (consisting of the characters 0-9 and A-F).
5. After typing the hexadecimal instruction, press the **STOR** key. This will store your instruction at the current address. The Trainer will automatically advance to the next memory address.
6. Repeat steps 4 and 5 for each subsequent instruction in your program:
   - Enter the next 4-digit hexadecimal instruction.
   - Press the **STOR** key to store the instruction and move to the next address.
7. Continue this process until you have entered all the instructions of your program.

Make sure you enter each instruction correctly and in the proper order. If you make a mistake, you can always go back to any address by entering it and pressing **ADDR** to correct the value.

Tip: Always double-check each instruction after entering it to make sure it has been input correctly before hitting **STOR**. This will save you time troubleshooting later!

Predicting Outcomes Before Execution: Before running your program, take a moment to predict the expected results and behavior of your code. For example, think about what values you expect to see in registers, and when. By anticipating the values in registers and the effects of each instruction, you can better identify when things might go wrong and understand why.

2.4 Running a Program

Once you have successfully entered all of your program's instructions into the Microprocessor Trainer, the next step is to run your program to see how it works.

Setting the Starting Point: First, you need to set the Trainer to the beginning of your program. Ensure the address field is active by pressing **ADDR** if necessary. Using the keypad, enter the address `20000000`, which is where your program starts. Finally, press the **ADDR** key to confirm the start address.

Choosing the Mode of Execution: With the starting address set, you have two options for running your program:

- **Step-by-Step Execution:** If you want to see how each instruction is executed one at a time, press the **STEP** key. This allows you to observe the changes and understand the flow of your program in detail. After pressing **STEP**, the Trainer executes the current instruction and then pauses, waiting for you to press **STEP** again for the next instruction.
- **Continuous Execution:** If you prefer to run your program continuously from the start to the end, press the **RUN** key. The Trainer will execute all the instructions continuously until the end of the program is reached, or until you press the **STEP** key to halt execution.

Monitoring and Troubleshooting: As your program runs, watch the display on the Trainer to see the outputs and any changes in the values stored in the memory. If the program does not behave as expected, you may need to stop the execution, review and correct the instructions you entered. Remember, debugging is a normal part of learning to program.

Tip: Start with the step-by-step execution mode when you're testing a new program. It gives you a better understanding of how each part of your program works and helps in identifying and fixing any errors.

2.5 Inspecting Memory

After you run your program, or even while it is running, you might want to check what is happening inside the memory of the Microprocessor Trainer. This process is called "inspecting memory," and it lets you see the values stored in different memory locations. Here's how you can do it:

Access the Memory: To look at the contents of a specific memory address, you need to first stop your program if it is running. You can do this by pressing the **STEP** key to pause execution.

Choosing the Address: Use the **ADDR** key to select the address field. Enter the desired memory address using the numeric keys (0-9, A-F). Remember that your program starts at `20000000`, so any addresses from this point onward may contain data or instructions from your program.

Loading the Value from Memory: Once you have entered the desired address, press the **LOAD** key. This will load the value at the address you entered, and switch the cursor to the value field.

2.6 Function Keys

The **LOAD** key on the Microprocessor Trainer is not just for loading the value at the current address. When held down and used in combination with other keys, it activates a variety of special functions. Here's how you can use these combinations effectively:

- **Keys 0 - 2**: Display different sets of registers on the screen. Pressing **LOAD** + **0** will show registers `r0` to `r3`, which is the default display.
- **Keys 4 - 6**: Save the currently entered program into internal flash memory. Each key represents a different storage slot, allowing for multiple programs to be stored simultaneously.
- **Keys 8 - A**: Load previously saved programs from the corresponding slots in the internal flash memory. This is useful for retrieving and running stored code without re-entering it.
- **Key D**: Toggle the display mode of the registers between hexadecimal and decimal. This can be helpful for easier reading or specific debugging needs.
- **Key F**: Activate a fast run mode, which executes the program without updating the display between each instruction. This mode allows for quicker execution but is generally not recommended for debugging, as changes in registers and memory are not visible in real time.

2.7 The Easter Egg

In your explorations with the Microprocessor Trainer, there's a special feature hidden for you to discover. This hidden feature, often called an "Easter egg," can be activated under specific conditions.

Hint: If you manage to load the ASCII values for the characters 'S', 'N', 'A', and 'K' into registers `r0` through `r3` respectively, and then execute a Supervisor Call (SVC) instruction code $\texttt{45}_{16}$ (which stands for 'E' in ASCII), something unexpected will happen!

Discover on Your Own: You might want to figure out how to achieve this setup and see what happens for yourself. It's a fun way to apply what you've learned about ASCII values, registers, and supervisor calls.

3 ARM-Thumb Basic Instructions

The following are just a few of the many instructions in the ARM-Thumb instruction set. There are enough instructions in this manual to be able to write any program. In other words, this set of instructions are Turing-complete. But other instructions, not listed here, are often necessary to write the shortest and fastest possible program.

ADD (add immediate) `0011 0ddd iiii iiii`

Add an 8 bit value (included in the instruction) to a register. Flags are updated based on the result value.

$$r[d] = r[d] + i$$

`ddd`	(3 bits)	This is the register number of the operand to add to, and the destination register where the result should be placed.
`iiii iiii`	(8 bits)	This is the value that should be added, included as an immediate operand in the instruction.

Examples:

`0011 0ddd iiii iiii`	3???	Basic instruction encoding
`0011 0000 0000 0001`	3001	Adds the number `1` to $r0$
`0011 0011 0001 0010`	3312	Adds the number $\texttt{12}_{16}$ ($\texttt{18}_{10}$) to $r3$
`0011 0111 1010 0101`	37A5	Adds the number $\texttt{A5}_{16}$ ($\texttt{165}_{10}$) to $r7$

ADD (add registers) `0001 100m mmnn nddd`

Add two registers together, storing the result in a (optionally different) third register. Flags are updated.

$$r[d] = r[m] + r[n]$$

`mmm`	(3 bits)	This is the register number of the first operand to add.
`nnn`	(3 bits)	This is the register number of the second operand to add.
`ddd`	(3 bits)	This is the register number to store the result in.

Examples:

`0001 100m mmnn nddd`	1???	Basic instruction encoding
`0001 1000 0000 0000`	1800	Adds $r0$ to register $r0$ and stores the result in $r0$ (doubles $r0$)
`0001 1000 0101 0011`	1853	Adds $r1$ to $r2$ and stores the result in $r3$
`0001 1000 1000 1011`	188B	Adds $r2$ to $r1$ and stores the result in $r3$ (same as above)

B (branch) `1110 0iii iiii iiii`

Branches the program counter to a new location. (Goto a different part of your program)

$$pc = pc + 2 * (i + 2)$$

`iiiiiiii...`	(11 bits)	Number of instructions to jump forward (or backwards, if negative...)

Examples:

`1110 0iii iiii iiii`	E???	Basic instruction encoding
`1110 0111 1111 0000`	E7F0	Goes back to 14 instructions before this one.
`1110 0111 1111 0001`	E7F1	Goes back to 13 instructions before this one.
`1110 0111 1111 0010`	E7F2	Goes back to 12 instructions before this one.
`1110 0111 1111 1100`	E7FC	Goes back to 2 instructions before this one.
`1110 0111 1111 1101`	E7FD	Goes back to 1 instructions before this one.
`1110 0111 1111 1110`	E7FE	Goes to **THIS** instruction (infinite loop)
`1110 0111 1111 1111`	E7FF	Goes to the instruction after this one (does nothing)
`1110 0000 0000 0000`	E000	Skips 1 instruction after this one
`1110 0000 0000 0001`	E001	Skips 2 instructions after this one
`1110 0000 0000 0010`	E002	Skips 3 instructions after this one
`1110 0000 0000 1111`	E00F	Skips 16 instructions after this one

CMP (compare registers) `0100 0010 10mm mnnn`

Subtracts one register from another and updates flags. The result of the subtraction is thrown away.

$$tmp = r[n] - r[m]$$

`nnn`	(3 bits)	This is the register number of the operand to subtract from.
`mmm`	(3 bits)	This is the register number of the subtrahend

Examples:

`0100 0010 10mm mnnn`	42??	Basic instruction encoding
`0100 0010 1000 0001`	4281	Subtracts $r0$ from register $r1$ and updates flags
`0100 0010 1011 1110`	42BE	Subtracts $r7$ from register $r6$ and updates flags

MOV (move immediate) `0010 0ddd iiii iiii`

Moves a value, provided in the instruction, into a chosen register.

$$r[d] = i$$

`ddd`	(3 bits)	This is the register number where the result should be placed.
`iiii iiii`	(8 bits)	This is the value that should be placed into the destination register. The high 24 bits of the register will be 0 and the remaining 8 bits will be the immediate value.

Examples:

`0010 0ddd iiii iiii`	2???	Basic instruction encoding
`0010 0000 0000 0000`	2000	Sets $r0$ to the value `0`
`0010 0111 1111 1111`	27FF	Sets $r7$ to the value $\texttt{FF}_{16}$ (255_{10})
`0010 0010 0000 0101`	2205	Puts the number `5` in $r2$

MOV (move register) `0100 0110 00mm mddd`

Copies a value from one register to another.

$$r[d] = r[m]$$

`mmm`	(3 bits)	This is the register number from which the value should be copied.
`ddd`	(3 bits)	This is the register number where the value should be placed.

Examples:

`0100 0110 00mm mddd`	46??	Basic instruction encoding
`0100 0110 0000 1000`	4608	Sets $r0$ to the value in $r1$
`0100 0110 0000 0001`	4601	Sets $r1$ to the value in $r0$
`0100 0110 0011 1110`	463E	Sets $r6$ to the value of $r7$

MUL (multiply registers) `0100 0011 01nn nddd`

Multiplies register n by register d, and stores the result in register d.

$$r[d] = r[d] * r[n]$$

`nnn`	(3 bits)	This is the register number that contains the first number to be multiplied.
`ddd`	(3 bits)	This is the register number with the second multiplicand, and where the value should be placed.

Examples:

`0100 0011 01nn nddd`	43??	Basic instruction encoding
`0100 0011 0100 0000`	4340	Squares $r0$ and puts the result back in $r0$
`0100 0011 0100 1000`	4348	Sets $r0$ to $r1$ times $r0$
`0100 0011 0100 0001`	4341	Sets $r1$ to $r1$ times $r0$
`0100 0011 0110 0011`	4363	Sets $r3$ to $r3$ times $r4$

SUB (subtract immediate) `0011 1ddd iiii iiii`

Subtract an 8 bit value (included in the instruction) from a register. Flags are updated based on the result value.

$$r[d] = r[d] - i$$

`ddd`	(3 bits)	This is the register number of the operand to subtract from, and the destination register where the result should be placed.
`iiii iiii`	(8 bits)	This is the value that should be subtracted, included as an immediate operand in the instruction.

Examples:

`0011 1ddd iiii iiii`	3???	Basic instruction encoding
`0011 1000 0000 0001`	3801	Subtracts the number `1` to $r0$
`0011 1011 0011 0001`	3B32	Subtracts the number $\texttt{32}_{16}$ ($\texttt{50}_{10}$) from $r3$
`0011 1111 1010 0101`	3FA5	Subtracts the number $\texttt{A5}_{16}$ ($\texttt{165}_{10}$) from $r7$

SUB (subtract registers) `0001 101m mmnn nddd`

Subtract one register from another, storing the result in a (optionally different) third register. Flags are updated.

$$r[d] = r[n] - r[m]$$

`nnn`	(3 bits)	This is the register number of the operand to subtract.
`nnn`	(3 bits)	This is the register number of the operand to subtract from.
`ddd`	(3 bits)	This is the register number to store the result in.

Examples:

`0001 101m mmnn nddd`	1???	Basic instruction encoding
`0001 1010 0000 0000`	1A00	Subtracts $r0$ from register $r0$ and stores the result in $r0$ (sets $r0$ to `0`)
`0001 1010 0101 0011`	1A53	Subtracts $r1$ from $r2$ and stores the result in $r3$
`0001 1000 1000 1011`	1A8B	Subtracts $r2$ from $r1$ and stores the result in $r3$

4 ARM-Thumb Conditional Branches

BZ (branch if zero/equal) `1101 0000 iiii iiii`

Branches the program counter to a new location, IF the Z flag is set. This will branch if the result of the previous ALU operation was zero, or after using the CMP instruction on two equal values. If the Z flag is not set, this instruction does nothing and execution continues at the next instruction.

$$\text{IF } Z\text{: } pc = pc + 2 * (i + 2)$$

`iiiiiiii`	(8 bits)	Number of instructions to jump forward (or backwards, if negative...)

Examples:

`1101 0000 iiii iiii`	D0??	Basic instruction encoding
`1101 0000 1111 0000`	D0F0	IF Z: Goes back to 14 instructions before this one.
`1101 0000 1111 0001`	D0F1	IF Z: Goes back to 13 instructions before this one.
`1101 0000 1111 0010`	D0F2	IF Z: Goes back to 12 instructions before this one.
`1101 0000 1111 1100`	D0FC	IF Z: Goes back to 2 instructions before this one.
`1101 0000 1111 1101`	D0FD	IF Z: Goes back to 1 instructions before this one.
`1101 0000 1111 1110`	D0FE	IF Z: Goes to **THIS** instruction (infinite loop if taken)
`1101 0000 1111 1111`	D0FF	IF Z: Goes to the instruction after this one (does nothing, either way)
`1101 0000 0000 0000`	D000	IF Z: Skips 1 instruction after this one
`1101 0000 0000 0001`	D001	IF Z: Skips 2 instructions after this one
`1101 0000 0000 0010`	D002	IF Z: Skips 3 instructions after this one
`1101 0000 0000 1111`	D00F	IF Z: Skips 16 instructions after this one

BNZ (branch if not zero/not equal) `1101 0001 iiii iiii`

Branches the program counter to a new location, IF the Z flag is **not** set. This will branch if the result of the previous ALU operation wasn't zero, or after using the CMP instruction on two inequal values.

$$\text{IF NOT } Z\text{: } pc = pc + 2 * (i + 2)$$

`iiiiiiii`	(8 bits)	Number of instructions to jump forward (or backwards, if negative...)

Examples:

`1101 0001 iiii iiii`	D1??	Basic instruction encoding
`1101 0001 1111 0000`	D1F0	IF NOT Z: Goes back to 14 instructions before this one.
`1101 0001 1111 0001`	D1F1	IF NOT Z: Goes back to 13 instructions before this one.
`1101 0001 1111 1101`	D1FD	IF NOT Z: Goes back to 1 instructions before this one.
`1101 0001 1111 1111`	D1FF	IF NOT Z: Goes to the instruction after this one (does nothing, either way)
`1101 0001 0000 0000`	D100	IF NOT Z: Skips 1 instruction after this one
`1101 0001 0000 0001`	D101	IF NOT Z: Skips 2 instructions after this one

BMI (branch if negative) `1101 0100 iiii iiii`

Branches the program counter to a new location, IF the if the result of the previous ALU operation was less than zero, or after using the CMP instruction where the comparand is larger than the other value.

$$\text{IF } N\text{: } pc = pc + 2 * (i + 2)$$

`iiiiiiii`	(8 bits)	Number of instructions to jump forward (or backwards, if negative...)

Examples:

`1101 0100 iiii iiii`	D4??	Basic instruction encoding
`1101 0100 1111 0000`	D4F0	IF N: Goes back to 14 instructions before this one.
`1101 0100 1111 1101`	D4FD	IF N: Goes back to 1 instructions before this one.
`1101 0100 1111 1111`	D4FF	IF N: Goes to the instruction after this one (does nothing)
`1101 0100 0000 0000`	D400	IF N: Skips 1 instruction after this one

There are more conditional branches. You can find information on how they are encoded in sections 7.2 and 7.2.1.

5 ARM-Thumb Miscellaneous Instructions

LSL (logical shift left, immediate) `0000 0iii iimm mddd`

Shifts register n left by a fixed number of bits, and stores the result in register d.

$$r[d] = r[m] << immediate$$

`iiiii`	(5 bits)	This is the number of bits to shift the number left.
`mmm`	(3 bits)	This is the register number that contains the number to be shifted.
`ddd`	(3 bits)	This is the register number to store the result in.

Examples:

`0000 0iii iimm mddd`	0???	Basic instruction encoding
`0000 0000 0100 0000`	0040	Shifts $r0$ one bit left (doubling it) and puts the result back in $r0$
`0000 0010 0010 0111`	0227	Shifts $r4$ eight bits left (multiplying by 256) and puts the result in $r7$

ORR (logical or, register) `0100 0011 00mm mddd`

Computes the logical or operation of registers m and d, and stores the result back in register d.

$$r[d] = r[m] \text{ logical-or } r[d]$$

`mmm`	(3 bits)	This is the register number that contains the first operand for logical-or.
`ddd`	(3 bits)	This is the register number that contains the second operand for logical-or, and the destination register to store the result within.

Examples:

`0100 0011 00mm mddd`	43??	Basic instruction encoding
`0100 0011 0000 1000`	4308	Ors $r1$ with $r0$ and stores the result in $r0$
`0100 0011 0010 1110`	432E	Ors $r5$ with $r6$ and stores the result in $r6$

AND (logical and, register) `0100 0000 00mm mddd`

Computes the logical and operation of registers m and d, and puts the result back in register d.

$$r[d] = r[m] \text{ logical-and } r[d]$$

`mmm`	(3 bits)	This is the register number that contains the first operand for logical-and.
`ddd`	(3 bits)	This is the register number that contains the second operand for logical-and, and the destination register to put the result within.

Examples:

`0100 0000 00mm mddd`	40??	Basic instruction encoding
`0100 0000 0000 1000`	4008	Ands $r1$ with $r0$ and stores the result in $r0$
`0100 0000 0010 1110`	402E	Ands $r5$ with $r6$ and stores the result in $r6$

LDR (load register) `0101 100m mmnn nttt`

Retrieves the word at the memory address formed from the sum of registers m and n, and puts the result in register t.

$$r[t] = memory(r[m] + r[n])$$

`mmm`	(3 bits)	This is the first register that is combined to form the memory address.
`nnn`	(3 bits)	This is the second register that is combined to form the memory address.
`ttt`	(3 bits)	This is the target, where the word retrieved from memory is placed.

Examples:

`0101 100m mmnn nttt`	5???	Basic instruction encoding
`0101 1001 1100 0001`	59C1	Loads the value from address $r7 + r0$ into $r1$

STR (store register) `0101 000m mmnn nttt`

Forms a memory address by adding registers m and n, and stores the contents of register t at that location.

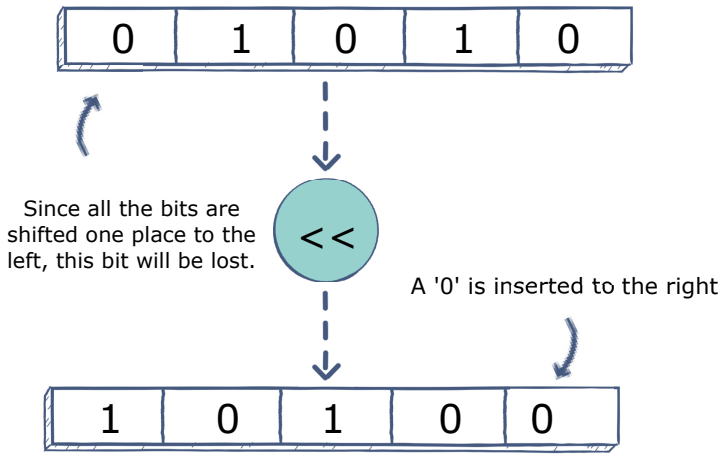
$$memory(r[m] + r[n]) = r[t]$$

`mmm`	(3 bits)	This is the first register that is combined to form the memory address.
`nnn`	(3 bits)	This is the second register that is combined to form the memory address.
`ttt`	(3 bits)	The contents of this register are stored to the chosen location in memory.

Examples:

`0101 000m mmnn nttt`	5???	Basic instruction encoding
`0101 0001 1100 0001`	51C1	Stores the value in $r1$ to the address $r7$ + $r0$

5.1 Logical Shifting (Illustrated)



5.2 Putting a Large Value in a Register

Often, it will be desirable to put a number larger than we can have in a single immediate into a register. The immediate move instruction available to us only can place an 8 bit value in a register, but registers are 32 bits wide.

There are many ways to accomplish this, including using load instructions. However, it can be complicated to calculate addresses. Below is a simple method to place the value 12345678_{16} into $r0$, while using $r1$ as a temporary space.

2012	`MOV` $r0$, 12_{16}	Places the value 12_{16} into $r0$.
0200	`LSL` $r0$, $r0$, `#8`	Shifts $r0$ left 8 bits; it now contains 1200_{16}.
2134	`MOV` $r1$, 34_{16}	Puts 34_{16} into $r1$.
4309	`ORR` $r0$, $r1$	Logical OR of $r0$ and $r1$, putting the result in $r0$, which now contains 1234_{16}.
0200	`LSL` $r0$, $r0$, `#8`	Shifts $r0$ left 8 bits; it now contains 123400_{16}.
2156	`MOV` $r1$, 56_{16}	Puts 56_{16} into $r1$.
4309	`ORR` $r0$, $r1$	Logical OR of $r0$ and $r1$, putting the result in $r0$, which now contains 123456_{16}.
0200	`LSL` $r0$, $r0$, `#8`	Shifts register 0 left 8 bits; it now contains 12345600_{16}.
2178	`MOV` $r1$, 78_{16}	Puts 78_{16} into $r1$.
4309	`ORR` $r0$, $r1$	Logical OR of $r0$ and $r1$, putting the result in $r0$, which now contains 12345678_{16}!

In this way, we can put large numbers into any of our registers. This approach isn't terribly efficient: 10 instructions and 20 bytes are used to load a 4 byte word into a register, but it is simple and it works.

6 Special (SuperVisor Call) Instructions

Supervisor Call (SVC) instructions are special system-level operations that provide input/output (I/O) functionality and other system utilities. These instructions have been specifically configured for the computing environment used in this course.

Many of these expect for an appropriate value to be loaded to $r0$ (or other registers) before executing them.

SVC00 (toggle LED) `1101 1111 0000 0000`

If the LED is off, this turns it on. Otherwise, it turns the LED off.

`1101 1111 0000 0000`	DF00	Basic instruction encoding

SVC01 (turn LED off) `1101 1111 0000 0001`

Turns the LED off.

`1101 1111 0000 0001`	DF01	Basic instruction encoding

SVC02 (turn LED on) `1101 1111 0000 0010`

Turns the LED on.

`1101 1111 0000 0010`	DF02	Basic instruction encoding

SVC03 (blink the LED) `1101 1111 0000 0011`

This slowly blinks the LED. The value in $r0$ determines how many times the LED blinks.

`1101 1111 0000 0011`	DF03	Basic instruction encoding

SVC11 (sleep, tenths) `1101 1111 0001 0001`

This freezes the program for a short amount of time. The length of time, in tenths of a second, is specified in $r0$. For instance, if $r0$ is 15, this will wait 1.5 seconds before the program resumes running.

`1101 1111 0001 0001`	DF11	Basic instruction encoding

SVC12 (sleep, seconds) `1101 1111 0001 0010`

This freezes the program for a short amount of time. The length of time, in seconds, is specified in $r0$. For instance, if $r0$ is 15, this will wait 15 seconds before the program resumes running.

`1101 1111 0001 0010`	DF12	Basic instruction encoding

SVC20 (clear screen) `1101 1111 0010 0000`

This clears the top half of screen and positions the cursor to write out text at the upper left.

`1101 1111 0010 0010`	DF20	Basic instruction encoding

SVC21 (output number, denary) `1101 1111 0010 0001`

This prints the number in $r0$ on the screen, as a base 10 (denary) number. For instance, if $r0$ is F_{16} (or 15_{10}), this will write the number 15 on the screen.

`1101 1111 0010 0001`	DF21	Basic instruction encoding

SVC22 (output number, hex) `1101 1111 0010 0010`

This prints the number in $r0$ on the screen, as a base 16 (hexadecimal) number. For instance, if $r0$ is F_{16} (or 15_{10}), this will write F on the screen.

`1101 1111 0010 0010`	DF22	Basic instruction encoding

SVC23 (output ASCII character) `1101 1111 0010 0011`

This prints the character in $r0$ on the screen. Only the last 8 bits of $r0$ are used. For instance, if $r0$ is $4D_{16}$ (or 77_{10}), this will write the character 'M' on the screen. See the ASCII chart later in this guide for reference.

`1101 1111 0010 0011`	DF23	Basic instruction encoding

SVC24 (draw dot) `1101 1111 0010 0100`

This changes the color of one pixel on the screen. The color to put in the pixel is stored as an 8 bit value in $r0$. The coordinates to change are ($r1$, $r2$).

`1101 1111 0010 0100`	DF24	Basic instruction encoding

SVC25 (draw icon) `1101 1111 0010 0101`

This draws a 32 byte (16 halfword) icon on the screen. The icon is expected to immediately follow this instruction, and when the draw is complete the instruction pointer will be increased by 22_{16} to skip the icon data. The color to draw the icon is stored as an 8 bit value in $r0$. The coordinates to draw the icon at are ($r1$, $r2$).

`1101 1111 0010 0101`	DF25	Basic instruction encoding (32 bytes of icon data follow)

6.1 Screen Colors (8-bit)

SVC25 and **SVC24** take in $r0$ an 8-bit color value. This color value is formatted in the bit pattern `RRGG GBBB`, where `RR` is a two-bit value indicating the intensity of the color red, `GGG` is a three-bit value indicating the intensity of green, and `BBB` is a three-bit value indicating the intensity of the color blue.

For instance, if it's desired to draw an icon with the color purple, that requires bright red and bright blue, with no green. Therefore, we will use the color `1100 0111` for full intensity red, no green, and full intensity blue. This becomes the value $C7_{16}$ which can be placed into $r0$ with the instruction `MOV R0, C7` encoded as `20C7`.

6.2 Screen Coordinates

Screen drawing routines also take in coordinates in ($r1$, $r2$). Unlike graphing coordinates, as y gets bigger this moves down the screen. The upper left corner of the screen is at (0, 0). The upper right of the screen is at (160, 0). The lowest row that you should use on the screen is row 53. Therefore, the bottom left and right corners of the screen are positioned at (0, 53) and (160, 53), respectively.

7 Miscellaneous Reference

7.1 ASCII Character Map

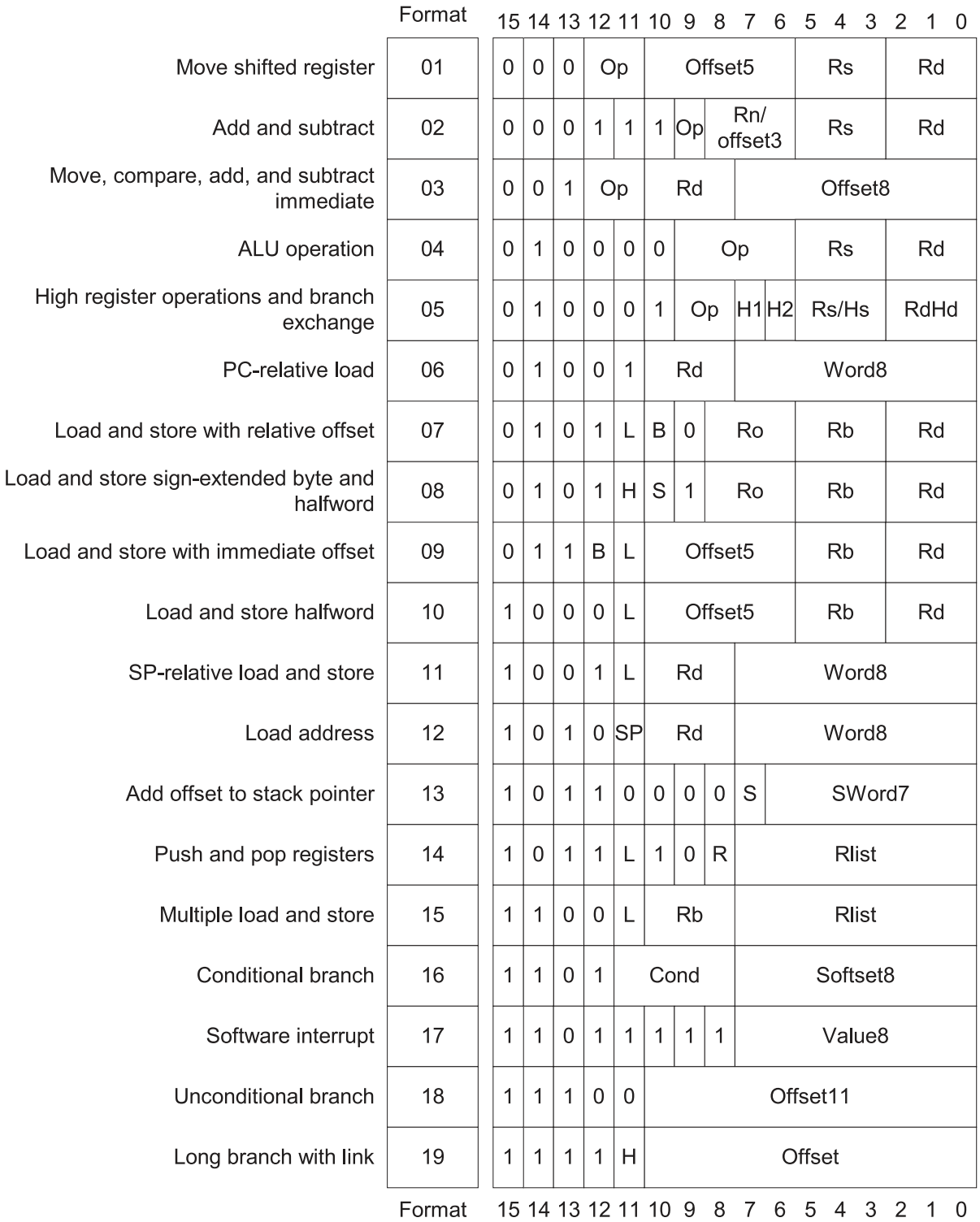
b7 b6 b5 / BITS b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 0 NUL	16 10 DLE	32 20 SP	48 30 0	64 40 @	80 50 P	96 60 ‘	112 70 p
0 0 0 1	1 1 SOH	17 11 DC1	33 21 !	49 31 1	65 41 A	81 51 Q	97 61 a	113 71 q
0 0 1 0	2 2 STX	18 12 DC2	34 22 "	50 32 2	66 42 B	82 52 R	98 62 b	114 72 r
0 0 1 1	3 3 ETX	19 13 DC3	35 23 #	51 33 3	67 43 C	83 53 S	99 63 c	115 73 s
0 1 0 0	4 4 EOT	20 14 DC4	36 24 $	52 34 4	68 44 D	84 54 T	100 64 d	116 74 t
0 1 0 1	5 5 ENQ	21 15 NAK	37 25 %	53 35 5	69 45 E	85 55 U	101 65 e	117 75 u
0 1 1 0	6 6 ACK	22 16 SYN	38 26 &	54 36 6	70 46 F	86 56 V	102 66 f	118 76 v
0 1 1 1	7 7 BEL	23 17 ETB	39 27 '	55 37 7	71 47 G	87 57 W	103 67 g	119 77 w
1 0 0 0	8 8 BS	24 18 CAN	40 28 (	56 38 8	72 48 H	88 58 X	104 68 h	120 78 x
1 0 0 1	9 9 HT	25 19 EM	41 29)	57 39 9	73 49 I	89 59 Y	105 69 i	121 79 y
1 0 1 0	10 A LF	26 1A SUB	42 2A *	58 3A :	74 4A J	90 5A Z	106 6A j	122 7A z
1 0 1 1	11 B VT	27 1B ESC	43 2B +	59 3B ;	75 4B K	91 5B [	107 6B k	123 7B {
1 1 0 0	12 C FF	28 1C FS	44 2C ,	60 3C <	76 4C L	92 5C \	108 6C l	124 7C |
1 1 0 1	13 D CR	29 1D GS	45 2D −	61 3D =	77 4D M	93 5D]	109 6D m	125 7D }
1 1 1 0	14 E SO	30 1E RS	46 2E .	62 3E >	78 4E N	94 5E ^	110 6E n	126 7E ~
1 1 1 1	15 F SI	31 1F US	47 2F /	63 3F ?	79 4F O	95 5F _	111 6F o	127 7F DEL

LEGEND: dec / CHAR / hex

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

(modified by Michael Lyle, 2021)

7.2 ARM Thumb Instruction Encoding



Source: DDI0210, Arm Limited, 2004

7.2.1 CondCodes and Suffixes for Conditional Branches

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Source: DDI0084D, Arm Limited, 1998

7.3 Hexadecimal and Decimal Conversion (Nybbles)

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Decimal	Binary	Hexadecimal
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

7.4 Branch Offsets (as used in B, BNZ, etc.)

Number of Instructions	Address Offset	8 Bit Conditional Branch Offset	11 Bit Branch Offset
Back 30	$-3C_{16}$	$E0_{16}$	$7E0_{16}$
Back 29	$-3A_{16}$	$E1_{16}$	$7E1_{16}$
...			
Back 16	-20_{16}	EE_{16}	$7EE_{16}$
Back 15	$-1E_{16}$	EF_{16}	$7EF_{16}$
Back 14	$-1C_{16}$	$F0_{16}$	$7F0_{16}$
Back 13	$-1A_{16}$	$F1_{16}$	$7F1_{16}$
Back 12	-18_{16}	$F2_{16}$	$7F2_{16}$
...			
Back 2	-4_{16}	FC_{16}	$7FC_{16}$
Back 1	-2_{16}	FD_{16}	$7FD_{16}$
Infinite Loop	0	FE_{16}	$7FE_{16}$
Forward 1 (Do Nothing)	$+2_{16}$	FF_{16}	$7FF_{16}$
Forward 2 (Skip 1)	$+4_{16}$	00_{16}	000_{16}
Forward 3 (Skip 2)	$+6_{16}$	01_{16}	001_{16}
...			
Forward 15 (Skip 14)	$+1E_{16}$	$0E_{16}$	$00E_{16}$
Forward 16 (Skip 15)	$+20_{16}$	$0F_{16}$	$00F_{16}$
Forward 17 (Skip 16)	$+22_{16}$	10_{16}	010_{16}
Forward 18 (Skip 17)	$+24_{16}$	11_{16}	011_{16}

General Equation for Branch Calculation: $\text{Offset} = \frac{\text{Target}-\text{CurAddr}}{2} + 2$

Backwards Unconditional Branches: $\text{Offset} = 800_{16} + \frac{\text{Target}-\text{CurAddr}}{2} - 2$

Backwards 8-bit Conditional Branches: $\text{Offset} = 100_{16} + \frac{\text{Target}-\text{CurAddr}}{2} - 2$

Thumb® 16-bit Instruction Set
Quick Reference Card

This card lists all Thumb instructions available on Thumb-capable processors earlier than ARM®v6T2. In addition, it lists all Thumb-2 16-bit instructions.
The instructions shown on this card are all 16-bit in Thumb-2, except where noted otherwise.
All registers are Lo (R0-R7) except where specified. Hi registers are R8-R15.

Key to Tables				
§	See Table **ARM architecture versions**.		<loreglist+LR>	A comma-separated list of Lo registers. plus the LR, enclosed in braces, { and }.
<loreglist>	A comma-separated list of Lo registers, enclosed in braces, { and }.		<loreglist+PC>	A comma-separated list of Lo registers. plus the PC, enclosed in braces, { and }.

Operation		§	Assembler	Updates	Action	Notes
Move	Immediate		MOVS Rd, #<imm>	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		MOVS Rd, Rm	N Z	Rd := Rm	Synonym of LSLS Rd, Rm, #0
	Hi to Lo, Lo to Hi, Hi to Hi		MOV Rd, Rm		Rd := Rm	Not Lo to Lo.
	Any to Any	6	MOV Rd, Rm		Rd := Rm	Any register to any register.
Add	Immediate 3		ADDS Rd, Rn, #<imm>	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		ADDS Rd, Rn, Rm	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		ADD Rd, Rd, Rm		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any	T2	ADD Rd, Rd, Rm		Rd := Rd + Rm	Any register to any register.
	Immediate 8		ADDS Rd, Rd, #<imm>	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		ADCS Rd, Rd, Rm	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		ADD SP, SP, #<imm>		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		ADD Rd, SP, #<imm>		Rd := SP + imm	imm range 0-1020 (word-aligned).
	Form address from PC		ADR Rd, <label>		Rd := label	label range PC to PC+1020 (word-aligned).
Subtract	Lo and Lo		SUBS Rd, Rn, Rm	N Z C V	Rd := Rn – Rm	
	Immediate 3		SUBS Rd, Rn, #<imm>	N Z C V	Rd := Rn – imm	imm range 0-7.
	Immediate 8		SUBS Rd, Rd, #<imm>	N Z C V	Rd := Rd – imm	imm range 0-255.
	With carry		SBCS Rd, Rd, Rm	N Z C V	Rd := Rd – Rm – NOT C-bit	
	Value from SP		SUB SP, SP, #<imm>		SP := SP – imm	imm range 0-508 (word-aligned).
	Negate		RSBS Rd, Rn, #0	N Z C V	Rd := – Rn	Synonym: NEGS Rd, Rn
Multiply	Multiply		MULS Rd, Rm, Rd	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare			CMP Rn, Rm	N Z C V	update APSR flags on Rn – Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Negative		CMN Rn, Rm	N Z C V	update APSR flags on Rn + Rm	
	Immediate		CMP Rn, #<imm>	N Z C V	update APSR flags on Rn – imm	imm range 0-255.
Logical	AND		ANDS Rd, Rd, Rm	N Z	Rd := Rd AND Rm	
	Exclusive OR		EORS Rd, Rd, Rm	N Z	Rd := Rd EOR Rm	
	OR		ORRS Rd, Rd, Rm	N Z	Rd := Rd OR Rm	
	Bit clear		BICS Rd, Rd, Rm	N Z	Rd := Rd AND NOT Rm	
	Move NOT		MVNS Rd, Rd, Rm	N Z	Rd := NOT Rm	
	Test bits		TST Rn, Rm	N Z	update APSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		LSLS Rd, Rm, #<shift>	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			LSLS Rd, Rd, Rs	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		LSRS Rd, Rm, #<shift>	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			LSRS Rd, Rd, Rs	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		ASRS Rd, Rm, #<shift>	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			ASRS Rd, Rd, Rs	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		RORS Rd, Rd, Rs	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Thumb 16-bit Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		`LDR Rd, [Rn, #<imm>]`	Rd := [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		`LDRH Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][15:0])	Clears bits 31:16. imm range 0-62, even.
	byte		`LDRB Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][7:0])	Clears bits 31:8. imm range 0-31.
	with register offset, word		`LDR Rd, [Rn, Rm]`	Rd := [Rn + Rm]	
	halfword		`LDRH Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		`LDRSH Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		`LDRB Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		`LDRSB Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		`LDR Rd, <label>`	Rd := [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		`LDR Rd, [SP, #<imm>]`	Rd := [SP + imm]	imm range 0-1020, multiple of 4.
	Multiple, not including base		`LDM Rn!, <loreglist>`	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		`LDM Rn, <loreglist>`	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		`STR Rd, [Rn, #<imm>]`	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		`STRH Rd, [Rn, #<imm>]`	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16]. imm range 0-62, even.
	byte		`STRB Rd, [Rn, #<imm>]`	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8]. imm range 0-31.
	with register offset, word		`STR Rd, [Rn, Rm]`	[Rn + Rm] := Rd	
	halfword		`STRH Rd, [Rn, Rm]`	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		`STRB Rd, [Rn, Rm]`	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		`STR Rd, [SP, #<imm>]`	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
	Multiple		`STM Rn!, <loreglist>`	Stores list of registers	Always updates base register, Increment After.
Push	Push		`PUSH <loreglist>`	Push registers onto full descending stack	
	Push with link		`PUSH <loreglist+LR>`	Push LR and registers onto full descending stack	
Pop	Pop		`POP <loreglist>`	Pop registers from full descending stack	
	Pop and return	4T	`POP <loreglist+PC>`	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	`POP <loreglist+PC>`	Pop, branch, and change to ARM state if address[0] = 0	
If-Then	If-Then	T2	`IT{pattern} {cond}`	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table **Condition Field**.
Branch	Conditional branch		`B{cond} <label>`	If {cond} then PC := label	label must be within – 252 to + 258 bytes of current instruction. See Table **Condition Field**.
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== | !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		`B <label>`	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		`BL <label>`	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		`BX Rm`	PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	`BLX <label>`	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	`BLX Rm`	LR := address of next instruction, PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
Extend	Signed, halfword to word	6	`SXTH Rd, Rm`	Rd[31:0] := SignExtend(Rm[15:0])	
	Signed, byte to word	6	`SXTB Rd, Rm`	Rd[31:0] := SignExtend(Rm[7:0])	
	Unsigned, halfword to word	6	`UXTH Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[15:0])	
	Unsigned, byte to word	6	`UXTB Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[7:0])	
Reverse	Bytes in word	6	`REV Rd, Rm`	Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24]	
	Bytes in both halfwords	6	`REV16 Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24]	
	Bytes in low halfword, sign extend	6	`REVSH Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF	

Thumb 16-bit Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Processor state change	Supervisor Call		`SVC <immed_8>`	Supervisor Call processor exception	8-bit immediate value encoded in instruction. Formerly SWI.
	Change processor state	6	`CPSID <iflags>`	Disable specified interrupts	
		6	`CPSIE <iflags>`	Enable specified interrupts	
	Set endianness	6	`SETEND <endianness>`	Sets endianness for loads and saves.	`<endianness>` can be `BE` (Big Endian) or `LE` (Little Endian).
	Breakpoint	5T	`BKPT <immed_8>`	Prefetch abort *or* enter debug state	8-bit immediate value encoded in instruction.
No Op	No operation		`NOP`	None, might not even consume any time.	Real NOP available in ARM v6K and above.
Hint	Set event	T2	`SEV`	Signal event in multiprocessor system.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for event	T2	`WFE`	Wait for event, IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for interrupt	T2	`WFI`	Wait for IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Yield	T2	`YIELD`	Yield control to alternative thread.	Executes as NOP in Thumb-2. Functionally available in ARM v7.

Condition Field	
Mnemonic	**Description**
`EQ`	Equal
`NE`	Not equal
`CS / HS`	Carry Set / Unsigned higher or same
`CC / LO`	Carry Clear / Unsigned lower
`MI`	Negative
`PL`	Positive or zero
`VS`	Overflow
`VC`	No 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`	Always. Do not use in `B{cond}`

In Thumb code for processors earlier than ARMv6T2, cond must not appear anywhere except in Conditional Branch (`B{cond}`) instructions.

In Thumb-2 code, cond can appear in any of these instructions (except `CBZ`, `CBNZ`, `CPSID`, `CPSIE`, `IT`, and `SETEND`).

The condition is encoded in a preceding `IT` instruction (except in the case of `B{cond}` instructions).

If `IT` instructions are explicitly provided in the Assembly language source file, the conditions in the instructions must match the corresponding `IT` instructions.

ARM architecture versions	
4T	All Thumb versions of ARM v4 and above.
5T	All Thumb versions of ARM v5 and above.
6	All Thumb versions of ARM v6 and above.
T2	All Thumb-2 versions of ARM v6 and above.

Document Number

ARM QRC 0006E

Change Log

Issue	Date	Change
A	Nov 2004	First Release
B	May 2005	RVCT 2.2 SP1
C	March 2006	RVCT 3.0
D	March 2007	RVCT 3.1
E	Sept 2008	RVCT 4.0

CO&D Name: _____ **Date:** _____

Program Name: _____

$Address_{16}$	$Instruction_{16}$	Mnemonic	$Value_{16}$ and/or Source-Regs	Dest-Reg
	(notes)			
20000000				
20000002				
20000004				
20000006				
20000008				
2000000A				
2000000C				
2000000E				
20000010				
20000012				
20000014				

CO&D Name: ______________________________ **Date:** ______________

Program Name: ____

Address$_{16}$	Instruction$_{16}$	Mnemonic	Value$_{16}$ and/or Source-Regs	Dest-Reg
	(notes)			

CO&D Name: _____ **Date:** _____

Program Name: _____

Execution Trace

$Address_{16}$	$R0_{16}$	$R1_{16}$	$R2_{16}$	$R3_{16}$	Notes

Index

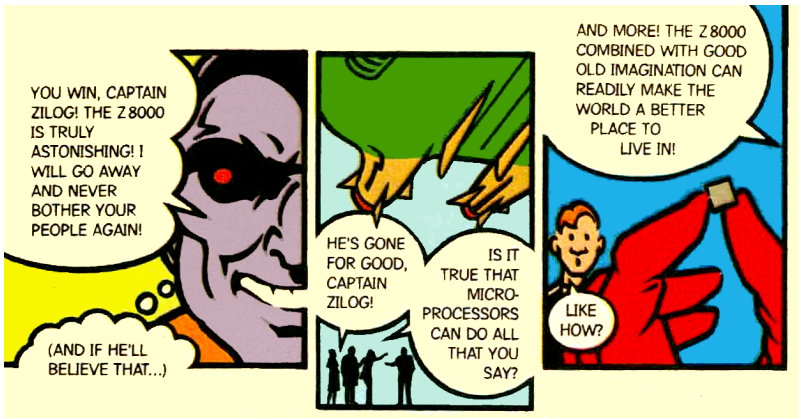


Figure 3: From *Captain Zilog*, 1979, Zilog, Inc.