

# ARM-Thumb Machine Code Reference

Michael Lyle

October 2021

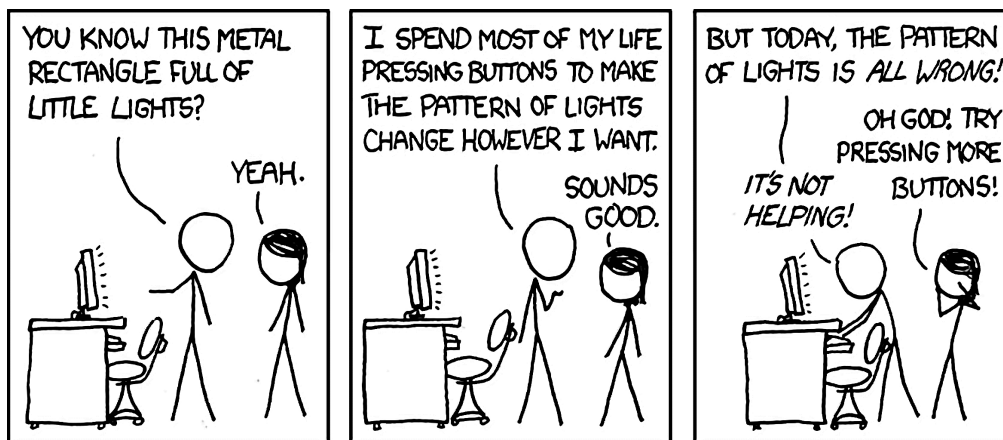
## Contents

<b>1</b>	<b>Architecture Summary</b>	<b>3</b>
1.1	Machine Language vs. Assembly Language . . . . .	3
<b>2</b>	<b>ARM-Thumb Basic Instructions</b>	<b>4</b>
	ADD (add immediate) . . . . .	4
	ADD (add registers) . . . . .	4
	B (branch) . . . . .	5
	CMP (compare registers) . . . . .	5
	MOV (move immediate) . . . . .	6
	MOV (move register) . . . . .	6
	MUL (multiply registers) . . . . .	7
	SUB (subtract immediate) . . . . .	7
	SUB (subtract registers) . . . . .	8
<b>3</b>	<b>ARM-Thumb Conditional Branches</b>	<b>9</b>
	BZ (branch if zero/equal) . . . . .	9
	BNZ (branch if not zero/not equal) . . . . .	10
	BMI (branch if negative) . . . . .	10
<b>4</b>	<b>ARM-Thumb Miscellaneous Instructions</b>	<b>11</b>
	LSL (logical shift left, immediate) . . . . .	11
	ORR (logical or, register) . . . . .	11
	AND (logical and, register) . . . . .	12
	LDR (load register) . . . . .	12
	STR (store register) . . . . .	13
4.1	Logical Shifting (Illustrated) . . . . .	14
4.2	Putting a Large Value in a Register . . . . .	14
<b>5</b>	<b>Special (SuperVisor Call) Instructions</b>	<b>15</b>
	SVC00 (toggle LED) . . . . .	15
	SVC01 (turn LED off) . . . . .	15
	SVC02 (turn LED on) . . . . .	15
	SVC03 (blink the LED) . . . . .	15
	SVC11 (sleep, tenths) . . . . .	15

SVC12 (sleep, seconds) . . . . .	16
SVC20 (clear screen) . . . . .	16
SVC21 (output number, denary) . . . . .	16
SVC22 (output number, hex) . . . . .	16
SVC23 (output ASCII character) . . . . .	16
SVC24 (draw dot) . . . . .	17
SVC25 (draw icon) . . . . .	17
5.1 Screen Colors (8-bit) . . . . .	17
5.2 Screen Coordinates . . . . .	17

## 6 Miscellaneous Reference 18

6.1 ASCII Character Map . . . . .	18
6.2 ARM Thumb Instruction Encoding . . . . .	19
6.3 Hexadecimal and Decimal Conversion (Nybbles) . . . . .	20
6.4 Branch Offsets (as used in B, BNZ, etc.) . . . . .	20
ARM Thumb-16 Quick Reference Card . . . . .	21



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

# 1 Architecture Summary

In this course, we will be learning ARM Thumb machine language. ARM is one of the most common instruction set architectures in the world.

It is used in almost all cellular phones, most tablets, and many laptop computers, in addition to a large share of the microcontrollers in cars and everyday items.

ARM Thumb is a portion of the instruction set consisting of 16 bit long instructions. Normal ARM instructions are 32 bits long, but we will not use them in this class.

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 we write one line of code for each instruction, and the assembler translates this into machine language. Assemblers make this process slightly easier by handling the details of instruction encoding and calculating addresses for the programmer.

## 2 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, <u>and</u> 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 $12_{16}$ ( $18_{10}$ ) to $r3$
0011 0111 1010 0101	37A5	Adds the number $A5_{16}$ ( $165_{10}$ ) 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)

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

iiiiiii... (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 <b>THIS</b> 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 $r_0$ from register $r_1$ and updates flags
0100 0010 1011 1110	42BE	Subtracts $r_7$ from register $r_6$ 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 $FF_{16}$ ( $255_{10}$ )
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, <u>and</u> 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 $32_{16}$ ( $50_{10}$ ) from $r3$
0011 1111 1010 0101	3FA5	Subtracts the number $A5_{16}$ ( $165_{10}$ ) 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.

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

<b>nnn</b>	(3 bits)	This is the register number of the operand to subtract.
<b>nnn</b>	(3 bits)	This is the register number of the operand to subtract from.
<b>ddd</b>	(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 $r3$



### 3 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.

IF Z:  $pc = pc + 2 * (i + 2)$

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

Examples:

1101 0000 <i>iiii</i> <i>iiii</i>	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 <b>THIS</b> instruction (infinite loop)
1101 0000 1111 1111	D0FF	IF Z: Goes to the instruction after this one (does nothing)
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.

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

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

Examples:

1101 0001 <i>iiii</i> <i>iiii</i>	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)
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.

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

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

Examples:

1101 0100 <i>iiii</i> <i>iiii</i>	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

## 4 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  $d$ .

$$r[d] = r[m] \ll \text{immediate}$$

iiiiii	(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  $t$ .

$$r[t] = \text{memory}(r[m] + r[n])$$

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.

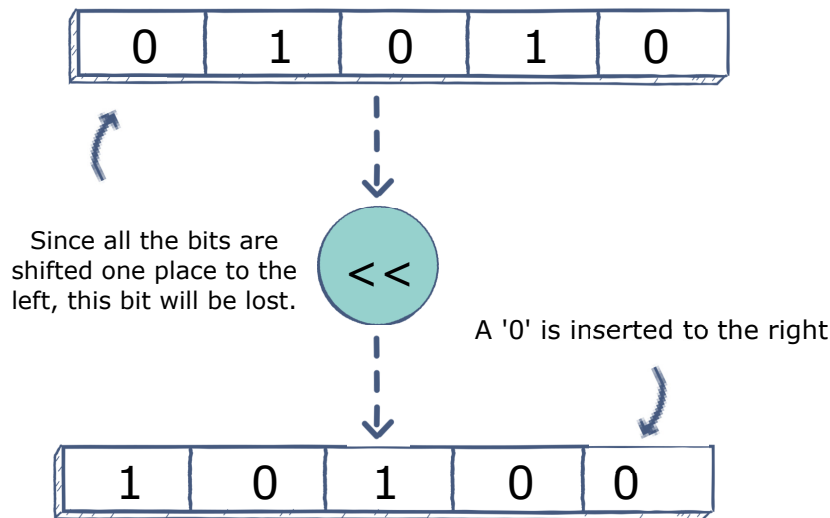
$$\text{memory}(r[m] + r[n]) = r[t]$$

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 $r7 + r0$

## 4.1 Logical Shifting (Illustrated)



## 4.2 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 $r0$ , $r1$	Ors $r0$ 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 $r0$ , $r1$	Ors $r0$ 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 $r0$ , $r1$	Ors $r0$ 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.

## 5 Special (SuperVisor Call) Instructions

These are special instructions that I have set up to be available on the computing environment you will use. They provide input and output (I/O) functionality and other convenient features. Many of these expect for an appropriate value to be loaded to *r0* (or other registers) before executing them.

<b>SVC00 (toggle LED)</b>	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
---------------------	------	----------------------------

<b>SVC01 (turn LED off)</b>	1101 1111 0000 0001
-----------------------------	---------------------

---

Turns the LED off.

1101 1111 0000 0001	DF01	Basic instruction encoding
---------------------	------	----------------------------

<b>SVC02 (turn LED on)</b>	1101 1111 0000 0010
----------------------------	---------------------

---

Turns the LED on.

1101 1111 0000 0010	DF02	Basic instruction encoding
---------------------	------	----------------------------

<b>SVC03 (blink the LED)</b>	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
---------------------	------	----------------------------

<b>SVC11 (sleep, tenths)</b>	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)

## 5.1 Screen Colors (8-bit)

**SVC25** and **SVC24** take in  $r0$  an 8-bit color value. This is formatted as **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, F8** encoded as **20C7**.

## 5.2 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 bottommost row you should draw on is row 53, which means the other two corners are at  $(0, 53)$  and  $(160, 53)$ .

## 6 Miscellaneous Reference

### 6.1 ASCII Character Map

b7 b6 b5 BITS b4 b3 b2 b1	0	0	0	0	1	1	1	1
	0	0	1	1	0	0	1	1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 ‘	112 p
0 0 0 1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
0 0 1 0	2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
0 0 1 1	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
0 1 0 0	4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
0 1 0 1	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
0 1 1 0	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
0 1 1 1	7 BEL	23 ETB	39 ‘	55 7	71 G	87 W	103 g	119 w
1 0 0 0	8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
1 0 0 1	9 HT	25 EM	41 )	57 9	73 I	89 Y	105 i	121 y
1 0 1 0	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
1 0 1 1	11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
1 1 0 0	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
1 1 0 1	13 CR	29 GS	45 —	61 =	77 M	93 ]	109 m	125 }
1 1 1 0	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
1 1 1 1	15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

LEGEND:

dec
CHAR
hex

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

(modified by Michael Lyle, 2021)

## 6.2 ARM Thumb Instruction Encoding

	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Move shifted register	01	0	0	0	Op		Offset5				Rs			Rd				
Add and subtract	02	0	0	0	1	1	1	Op	Rn/ offset3			Rs			Rd			
Move, compare, add, and subtract immediate	03	0	0	1	Op		Rd		Offset8									
ALU operation	04	0	1	0	0	0	0	Op			Rs			Rd				
High register operations and branch exchange	05	0	1	0	0	0	1	Op	H1	H2	Rs/Hs			RdHd				
PC-relative load	06	0	1	0	0	1	Rd		Word8									
Load and store with relative offset	07	0	1	0	1	L	B	0	Ro			Rb			Rd			
Load and store sign-extended byte and halfword	08	0	1	0	1	H	S	1	Ro			Rb			Rd			
Load and store with immediate offset	09	0	1	1	B	L	Offset5				Rb			Rd				
Load and store halfword	10	1	0	0	0	L	Offset5				Rb			Rd				
SP-relative load and store	11	1	0	0	1	L	Rd		Word8									
Load address	12	1	0	1	0	SP	Rd		Word8									
Add offset to stack pointer	13	1	0	1	1	0	0	0	0	S	SWord7							
Push and pop registers	14	1	0	1	1	L	1	0	R	Rlist								
Multiple load and store	15	1	1	0	0	L	Rb		Rlist									
Conditional branch	16	1	1	0	1	Cond				Softset8								
Software interrupt	17	1	1	0	1	1	1	1	1	Value8								
Unconditional branch	18	1	1	1	0	0	Offset11											
Long branch with link	19	1	1	1	1	H	Offset											
	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Source: DDI0210, Arm Limited, 2004

### 6.3 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
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

### 6.4 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)	$+24_{16}$	$11_{16}$	$011_{16}$

## 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	Assembler	Updates	Action	Notes
<loreglist>	ARM architecture versions.				
	A comma-separated list of Lo registers, enclosed in braces, { and }.			<loreglist>+LR>	A comma-separated list of Lo registers, plus the LR, 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
<b>Move</b>	Immediate	MOV <sub>S</sub> Rd, #<imm>	N Z	Rd := imm	imm range 0-255.
	Lo to Lo	MOV <sub>S</sub> Rd, Rm	N Z	Rd := Rm	Synonym of LSL <sub>S</sub> Rd, Rm, #0
	Hi to Lo, Lo to Hi, Hi to Hi	MOV Rd, Rm		Rd := Rm	Not Lo to Lo.
	Any to Any	MOV Rd, Rm		Rd := Rm	Any register to any register.
<b>Add</b>	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	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).
	Lo and Lo	SUBS Rd, Rn, Rm	N Z C V	Rd := Rn - Rm	
<b>Subtract</b>	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	MULS Rd, Rm, Rd	N Z *	Rd := Rm * Rd	* C and V flags unpredictable in \$4T, unchanged in \$5T and above
<b>Compare</b>	Negative	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.
	Immediate	CMN Rn, Rm	N Z C V	update APSR flags on Rn + Rm	
		CMP Rn, #<imm>	N Z C V	update APSR flags on Rn - imm	imm range 0-255.
<b>Logical</b>	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	
	Logical shift left	LSLS Rd, Rm, #<shift>	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
<b>Shift/rotate</b>	Logical shift right	LSRS Rd, Rd, Rs	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
		LSRS Rd, Rm, #<shift>	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
	Arithmetic shift right	ASRS Rd, Rd, Rs	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
		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	ROBS 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
<b>Load</b>	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.
<b>Store</b>	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.
	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]	Ignore Rd[31:16], imm range 0-62, even.
	byte	STRB Rd, [Rn, #<imm>]	[Rn + imm][7:0] := Rd[7:0]	Ignore 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]	Ignore Rd[31:16]
	byte	STRB Rd, [Rn, Rm]	[Rn + Rm][7:0] := Rd[7:0]	Ignore 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.
<b>Push</b>	Push	PUSH <loreglist>	Push registers onto full descending stack	
<b>Pop</b>	Push with link	PUSH <loreglist+LR>	Push LR and registers onto full descending stack	
	Pop	POP <loreglist>	Pop registers from full descending stack	
	Pop and return	POP <loreglist+PC>	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	POP <loreglist+PC>	Pop, branch, and change to ARM state if address[0] = 0	
	If-Then	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 <b>Condition Field</b> .
<b>Branch</b>	Conditional branch	B{cond} <label>	If {cond} then PC := label	label must be within -252 to +258 bytes of current instruction. See Table <b>Condition Field</b> .
	Compare, branch if (non) zero	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 0xFFFFFFFF	Change to ARM state if Rm[0] = 0.
<b>Extend</b>	Branch with link and exchange	BLX <label>	LR := address of next instruction, PC := label	This is a 32-bit instruction.
	Branch with link and exchange	BLX Rm	LR := address of next instruction, PC := Rm AND 0xFFFFFFFF	label must be within ±4MB of current instruction (T2: ±16MB). Change to ARM state if Rm[0] = 0.
	Signed, halfword to word	SXTB Rd, Rm	Rd[31:0] := SignExtend(Rm[15:0])	
	Signed, byte to word	SXTH Rd, Rm	Rd[31:0] := SignExtend(Rm[7:0])	
	Unsigned, halfword to word	UXTB Rd, Rm	Rd[31:0] := ZeroExtend(Rm[15:0])	
<b>Reverse</b>	Unsigned, byte to word	UXTB Rd, Rm	Rd[31:0] := ZeroExtend(Rm[7:0])	
	Bytes in word	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	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	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	<endianness> can be BE (Big Endian) or LE (Little Endian). 8-bit immediate value encoded in instruction.
	Set endianness	6 SETEND <endianness>	Sets endianness for loads and saves.	
	Breakpoint	5T BKPT <immed_8>	Prefetch abort or enter debug state	
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.

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This reference card is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this reference card, or any error or omission in such information, or any incorrect use of the product.

Document Number

ARM QRC 0006E

Change Log

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