CEG 3136 Summary Sheet

Fall 2025
University of Ottawa
Owen Daigle

Contents

1 Data Representation

One **Byte** is defined as 8 bits. In 32 bit architecture, one **Word** is 32 bits, and therefore a half word is 16 bits, and a double word is 64 bits.

We have unsigned integers and signed integers. Unsigned is simple, signed can be represented in either signed magnitude, twos complement, or ones complement.

Ex. Show -5 in 4 bits in all 3 forms

Positive 5: `0101`

Signed Magnitude: `1101`

Twos Complement: `1011`

Ones Complement: `1010`

In an adder/subtractor, we generate a carry flag (C) (carry out of most significant bit [MSB]) and an overflow flag (V). If we are doing unsigned, the carry flag signifies something is wrong (C=1 for addition, C=0 for subtraction). For signed, we ignore C, and use V. If V=1, something went wrong.

Overflow is the XOR of the carry out of the MSB and the carry in to the MSB.

We typically use twos complement since it makes addition and subtraction able to use the same logic.

1.1 Strings

Strings are represented using **ASCII** codes. Each string is compared using its ASCII value.

$$CAT < Cat < DOG < Dog < cat < dog$$

Letter	A	B	C	...	X	Y	Z	...	a	b	c	...	x	y	z
ASCII Code	41	42	43		58	59	5A		61	62	63		78	79	7A

1.2 Fixed Point

The **Q Notation** represents the type of fixed point using `UQm.n`.

U	Unsigned. Remove if we are doing signed
Q	Means Q notation
m	Number of integer bits
n	number of fractional bits

Ex. Approximate $-\pi$ using `Q3.12`

This means we have the sign bit, then 3 integer bits and 12 fractional bits.

We can represent pi as:

$$0011.001001000100$$

So after taking the twos complement $-\pi$ becomes:

$$1100.110110111100$$

If it was `UQ4.12` we could only represent π not $-\pi$ but it would be:

$$0011.001001000100$$

To add fixed point numbers, it is very simple. We just treat it as if the radix is non existent.

1.3 Floating Point

Floating point is similar to scientific notation in decimal. We first need to normalize it. So we make it in the form of $1.xxxxxxx \times 2^{exp}$. We then hide the `1` as it is implied.

Using single precision, we have 1 bit for the sign, 8 for exponent, and 23 for the fractional.

To represent negative and positive exponents, we add 127 to the exponent.

Ex. Express $(36.5625)_{10}$ as a 32 bit floating point number using the IEEE standard.

The value in binary is 100100.1001 and normalized is 1.001001001×2^5.

1. Sign: 0
2. Exponent: 5+127 = 132 = 10000100
3. Mantissa: 00100100100000000000000

So the 32 bit number is: 0 10000100 00100100100000000000000 (without the spaces ofc)

We reserve some values for special cases:

Sign	Exponent	Fraction	Meaning
1 / 0	0000 0000	0000 ... 0000	0
1 / 0	1111 1111	0000 ... 0000	+ / - infinity
x	1111 1111	any non zero value	NaN (Not a Number)

Floating point has big dynamic range, but is less precise and more complicated than fixed point.

2 ARM Instructions

ARM has 3 main processor families;

CORTEX-A	High performance Application processors
CORTEX-R	Reliable Real time processors for mission critical purposes
CORTEX-M	Low cost, Low power Microcontroller

ARM has a few different instruction sets. The CORTEX-M series supports the T32 instruction set which includes both space saving 16 bit instructions, and high performance 32 bit instructions.

Other architectures such as CORTEX-A support T32 and A32 instructions with some supporting A64 as well.

ARM is a RISC architecture, so it cannot directly access memory through instructions. It must first **load** from memory into registers, then **modify**, and finally **store** back into memory.

We have 16 **core registers** and some **special purpose registers** as well. Since we are 32 bit, the registers are all 32 bits wide.

Register	General or Special	Purpose
R0	C	General Purpose
R1	C	General Purpose
R2	C	General Purpose
R3	C	General Purpose
R4	C	General Purpose
R5	C	General Purpose
R6	C	General Purpose
R7	C	General Purpose
R8	C	General Purpose
R9	C	General Purpose
R10	C	General Purpose
R11	C	General Purpose
R12	C	Intra Procedure Call Register (IR)
R13	C	Stack Pointer (SP) - Often there are two: MSP (Main) and PSP (Process)
R14	C	Link Register (LR)
R15	C	Program Counter (PC)
xPSR	S	Program Status Register
BASEPRI	S	Interrupt Priorities
PRIMASK	S	Enabling and Disabling Interrupts
FAULTMASK	S	Fault Handling
CONTROL	S	

Often we map some hardware device to a memory to make it easier to work with. For example, we may have it set up so bit 7 of `R0` is 1 if an LED is on, or 0 if the LED is off.

Assembly has 4 main classes of instructions:

- Arithmetic and Logic
- Data Movement
- Compare and Branching
- Miscellaneous

Each instruction has 4 parts:

General From	label	mnemonic	operand(s)	comments
Ex		BX	LR	; branch to LR
Ex	LOOP	CMP	R1, R2	; start of loop, compares R1 and R2
Ex		STR	R1, R2	
Ex		ADD	R3, R5, R8	

We also have assembly directives which are just information for the assembler such as `ALIGN`, `EXPORT`, and `ENDP`.

Directive	**Meaning**
AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

2.1 Arithmetic and Logic

Here are just a few of the arithmetic instructions for T32.

Mnemonic	Syntax	Meaning	Operation
ADD	{Rd,} Rn, Op2	Add	Rd $\leftarrow$ Rn + Op2
ADC	{Rd,} Rn, Op2	Add w/ carry	Rd $\leftarrow$ Rn + Op2 + Carry
SUB	{Rd,} Rn, Op2	Subtract	Rd $\leftarrow$ Rn $-$ Op2
SBC	{Rd,} Rn, Op2	Subtract w/ carry	Rd $\leftarrow$ Rn $-$ Op2 + Carry $-$ 1
RSB	{Rd,} Rn, Op2	Reverse subtract	Rd $\leftarrow$ Op2 $-$ Rn
MUL	{Rd,} Rn, Rm	Multiply	Rd $\leftarrow$ (Rn $\times$ Rm)[31 : 0]
MLA	Rd, Rn, Rm, Ra	**Multiply and accumulate**	
		Rd $\leftarrow$ (Ra + (Rn $\times$ Rm))[31 : 0]	
MLS	Rd, Rn, Rm, Ra	**Multiply and subtract**	
		Rd $\leftarrow$ (Ra $-$ (Rn $\times$ Rm))[31 : 0]	
SDIV	{Rd,} Rn, Rm	**Signed divide**	Rd $\leftarrow$ Rn $\div$ Rm
UDIV	{Rd,} Rn, Rm	**Unsigned divide**	Rd $\leftarrow$ Rn $\div$ Rm

There are also lots of logic ones such as `AND`, `ORR`, `EOR` (XOR), `ORN` (NOR) and so on.

There are many other instructions, the ARM T32 instruction set has a lot.

2.1.1 NZCV Flags

These flags are stored in bits 28 to 31 of the PSR.

Flag	Meaning
N	Negative - Result is Negative
Z	Zero - Result is Zero
C	Carry - Unsigned Arithmetic out of range
V	Overflow - Signed Arithmetic out of range

To update these flags, we add an S to the end of the instruction.

Ex.

Does not update NZCV flags: ADD, SUB, MUL, etc

Does update NZCV flags: ADDS, SUBS, MULS, etc

Always updates NZCV flags: CMP, CMN, TST, TEQ

2.1.2 Saturation

Saturation is a logical operation that deals with the case where overflow occurs.

Normally, it will wrap back around to the lowest value. However, sometimes we want to cap the highest value.

With Saturation (4 bits)	7+1=-8
Without Saturation (4 bits)	7+1=7

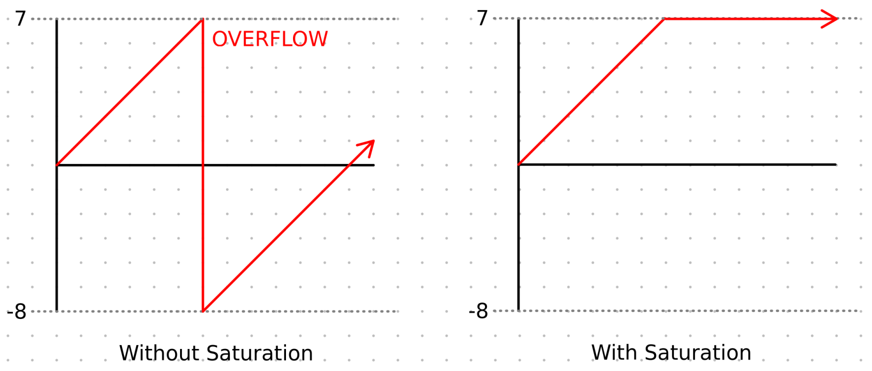


Figure 1: Saturation

2.1.3 Other Instructions

Instruction	Description	Similar Instructions
RBIT Rd, Rn	Reverses bit order in word	REV (byte order), REV16 (For half words), REVSH (Sign Extend)
SXTB {Rd,} Rm	Sign Extension (Byte)	SXTH (Half word), UXTB/UXTH (Zero extend)
MOV Rd, Rx	Move from Rx to Rd	MVN (MV and NOT), MRS (From special reg), MSR (From gen to special)
LSL Rd, Rn, #	Move Rn to Rd and left shift	LSR (right logical), ASR (Right arithmetic), ROR (rotate right)

2.2 Memory

Memory is byte addressable, but we typically only start a 32 bit word at a multiple of 4, a 16 bit half word at a multiple of 2, and a byte at any point.

LDRxx R0, [R1]	Load from memory at R1 into R0
STRxx R0, [R1]	Store contents of R0 into memory at R1

If we are storing something smaller than the memory width (byte, or halfword) we need to differentiate between signed (add `S`) [`LDRSB`, `LDRSH`] and unsigned (do not add `S`) [`LDRB`, `LDRH`].

When loading and storing, we can also address bits after the location we specify. This is useful for arrays. We have a few modes:

Register Offset	LDR r0, [r1, r2]	Target: r1+r2
Immidiate Offset	LDR r0, [r1, #8]	Target: r1 + 8
Pre-Index	LDR r0, [r1, #4]!	Target: r1+4, update r1 to r1+4 after read
PostIndex	LDR r0, [r1], #4	Target: r1, increase r1 by 4 after read

We can also load just a byte, or halfword using the suffix `B` or `H` to get `LDRB` and `LDRH`. Note that `LDRSB` extends the sign, and `LDRB` extends with zeros.

2.2.1 LD and STR Multiple Registers

We use the instructions `LDM` and `STM` to store or load multiple registers. We append this with a list of registers such as `LDM__␣{R1,R2,R5,R3}`. We also have an identifier as shown in the below table. Loads occur in ascending register order, and stores occur in descending register order.

The modes are:

IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

Ex. We have the following memory, and then `STMIB␣R4␣{R5,␣R6,␣R7}` is run. What are the three resulting memory stores?

```
R4: 0x20000080
R5: 0x55555555
R6: 0x66666666
R7: 0x77777777
```

We are storing multiple registers, increasing before mode. Since we are storing, we store in descending register order.

It will therefore be storing starting in `0x20000080+4`, then `0x20000080+8`, and finally `0x20000080+C`. The values will be `R7`, `R6`, and then `R5`.

```
0x2000 0084: 0x7777 7777
0x2000 0088: 0x6666 6666
0x2000 008C: 0x5555 5555
```

2.3 Endianess

Endianess means within a 32 bit word (or any multi byte data structure) do we start the LSB at the low address (little endian) or high address (big endian)?

Ex. If we have the memory and instructions below with a *little endian system*, where does `0x33` end up?

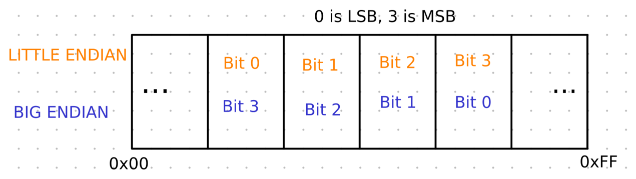


Figure 2: Endianess

```
R0: 0x20004400
R1: 0x11223344
STR R1, [R0, #4]!
```

Running the `STR` instruction, we take `R1` and put it in memory at `R0+4`. In terms of how it gets put, this depends on the endianess.

Since it is little endian, then the lowest memory address has the LSB (`0x44`).

	...	...
LO	0x20004404	0x44
	0x20004405	0x33
	0x20004406	0x22
HI	0x20004407	0x11
	...	...

2.4 Control Flow Instructions

We have 4 variations of the branch command. These will branch to either a label, or an address.

Instruction	Operands	Description
B	label	Branch
BL	label	Branch with link
BLX	Rm	Branch and Exchange with link
BX	Rm	Branch and Exchange

We have condition codes. These are appended to almost any instruction and they will only execute the instruction if the condition is true. It uses the status flags NZCV. These are typically performed after a `CMP` operation.

Suffix	Description	Flags tested
EQ	Equal	$Z = 1$
NE	Not Equal	$Z = 0$
CS/HS	Unsigned Higher or Same	$C = 1$
CC/LO	Unsigned Lower	$C = 0$
MI	Minus (Negative)	$N = 1$
PL	Plus (Positive or Zero)	$N = 0$
VS	Overflow Set	$V = 1$
VC	Overflow Cleared	$V = 0$
HI	Unsigned Higher	$C = 1 \ \& \ Z = 0$
LS	Unsigned Lower or Same	$C = 0 \ \| \ Z = 1$
GE	Signed Greater or Equal	$N = V$
LT	Signed Less Than	$N \neq V$
GT	Signed Greater Than	$Z = 0 \ \& \ N = V$
LE	Signed Less than or Equal	$Z = 1 \ \| \ N \neq V$
AL	Always	None

Ex. Branch to `FOO` if `r0` is less than `0`.

```
CMP r0, #0                ;compare r0 with 0
BLE FOO                   ;branch to FOO if LE (Z=1)
```

This is similar to some c code doing:

```
if (a < 0) { //assuming a is in r0
        foo(); //or something else, whatever code is located at FOO
}
```

ARM assembly lets us use the IT (If Then) syntax as well.

Ex.

```
ITTE NE                   ; Two commands will follow with NE
ANDNE r0, r0, r1          ; Then one command with the opposite
ANDNE r2, r2, #1          ; of NE which is EQ
MOVEQ r2, r3              ;

ITT EQ                    ; The IT can be ommitted from code
MOVEQ ...                 ; and the assembler will add it.
ADDEQ ...
```

3 Subroutines

The link register `LR` contains the return address of the subroutine. This is copied back to the PC when the subroutine is finished.

In ARM, we store any parameters in registers R0 through R3. Any additional parameters need to be put on the stack. Also, if the parameters are larger than 32 bits, they can take up more than one register (a 128 bit parameter would take up R0, R1, R2, R3, and a 64 bit parameter would take up either R0 and R1, or R2 and R3).

It returns the return value in R0.

Registers R0 through R3 can be freely changed by the subroutine, as well as R12 and R14 (LR). In other words, the calling function cannot expect them to keep the same data when the subroutine returns. The opposite is true with registers R4 to R11 where they must be preserved. If the subroutine changes anything in those registers, it must return them to the previous value before returning.

We use `BL` or `BLX` to call a subroutine.

Ex. We have the following function signature:

`uint64_t dog(uint16_t cat, uint64_t horse, uint32_t pig);`

These parameters will go into:

```
   R0   : cat
   R2   : horse pt 1    -- doubles must be either R0 R1, or R2 R3
   R3   : horse pt 2    -- 2 parts since double
  [SP]  : pig           -- since no space left in R0-R3
```

The return parameter will be in `R0` and `R1` since it is a double.

3.1 Stack

The ARM stack uses a full descending stack. This means that the stack pointer points to the top piece of data on the stack. The stack also grows down to memory address 0 as items are pushed to it.

We have the instructions `PUSH␣<reg␣list>` and `POP␣<reg␣list>`.

When we push or pop multiple registers, the highest number register is pushed first, and popped last.

Ex. `PUSH␣{r6,␣r8,␣r7}`

This instruction will first push `r8`, followed by `r7` and then finally `r6`.

It would be equivalent to `PUSH␣{r6,␣r7,␣r8}` and to `PUSH␣{r8,␣r6,␣r7}` and so on.

Ex. `POP␣{r6,␣r8,␣r7}`

This instruction will first pop `r6`, followed by `r7` and then finally `r8`.

It would be equivalent to `POP␣{r6,␣r7,␣r8}` and to `POP␣{r8,␣r6,␣r7}` and so on.

Ex.

```
...                         ;main program
        BL foo
...

foo PROC
        PUSH {r4}           ;we are using r4 which must be preserved
         ...
        MOV r4, #1          ;this changes r4, good thing we saved it
         ...
        POP {r4}            ;this restores r4 for the caller function
        BX LR               ;goes back to caller
ENDP
```

We also need to preserve the LR on the stack if we call a subroutine from inside a subroutine since it could be overridden. Then we would have no way to return to the main program.

Often we have two stack pointers, the `MSP` (main) and `PSP` (process). This is toggled by a bit in the CONTROL register.

4 C and Assembly

When we have C code, it goes through a lot of steps to get into ARM assembly to be loaded onto the microcontroller.

$$Proprocessor \rightarrow Compiler \rightarrow Assembler \rightarrow Linker \rightarrow Loader \rightarrow MCU(thru\ programmer) \rightarrow Debugger$$

Typically, C will set up data so every word starts at an even multiple of 4 bits address. This is for efficiency. Same idea with half words, but every 2 bits. C does this by padding extra space. We can use the `__packed` keyword in c to not pad the extra space. But this can cause weird behavior.

If we want to mix C and assembly, we can do this. In assembly, we call a C function using the `import` keyword, and export a function to C using the `export` keyword. Similarly, in C we can import something from assembly using the `extern` keyword. This can work with data as well as functions.

By default, every variable in C is exported to the whole system by the linker. If we use the `static` keyword, then it will only be accessible in the entirety of the one C file.

Ex. What would the memory look like with something like this if we use packed, or unpacked?

```
typedef struct {
 uint8_t   id;
 char     *name;
 uint16_t  x_loc;
 uint16_t  y_loc;
 uint16_t  z_loc;
 uint32_t  score;
 uint8_t   color;
 uint8_t   icon;
} Player_t;
```

Packed	**Unpacked**
id	id
char	0
char	0
char	0
char	char
x_loc	char
x_loc	char
y_loc	char
y_loc	x_loc
z_loc	x_loc
z_loc	y_loc
score	y_loc
score	z_loc
score	z_loc
score	0
colour	0
icon	score
	score
	score
	score
	colour
	icon
	0
	0

4.1 Volatile Datatypes

The `volatile` keyword means each time we use a variable, we need to import it from memory into a register. This is useful when an external event may change memory at any point. In this case, we need to ensure that we don't use an old version of the variable.

4.2 Interrupts

An interrupt is a signal that occurs that tells the controller that it needs to stop whatever it is currently doing, save the state using the stack, and then move on to the interrupt service routine (ISR). After it then restores the stack, and goes back to the user program.

It needs to save `xPSR`, `PC`, `LR`, `R12`, `R3`, `R2`, `R1`, and `R0` . Therefore it can use any of those registers to store data. Any other registers that are used must be returned to their original state.

Note that we use the `LR` to indicate the stack pointer, so we need to preserve it to `BX␣LR` at the end of the ISR.

To enable the interrupt, we need to do:

1. Program peripheral control register to allow it to generate interrupts
2. Program NVIC (Nested Vector Interrupt Controller) to accept interrupts

NVIC lets us enable or disable interrupts to the peripherals. Each interrupt in the NVIC has its own priority set. There is the subpriority (order in which the interrupts get run), and preemption priority (higher preemption priorities can interrupt other interrupts).

We can clear an interrupt by writing 1 to the correct register.

5 FPU

The Floating Point Unit (FPU) greatly improves efficiency when dealing with floats over software floating point calculations. These FPUs are however quite large and expensive to implement, so they are not found on all systems.

By default, this FPU is disabled due to power usage. It can easily be enabled.

The FPU coprocessor has its own bank of general purpose registers, and special purpose registers labeled as:

s1	s0
s3	s2
s5	s4
...	...
s31	s30
FPCAR	
FPSCR	
FPCCR	

If we are operating with doubles instead of floats, then we use two of those registers to hold a double. So `d0` would use `s1` and `s0`. We can copy to and from the `s` registers to the `r` registers.

d0[0]	d0[1]
d1[0]	d1[1]
d2[0]	d2[1]
...	...
d14[0]	d14[1]
d15[0]	d15[1]

We have similar instructions to the regular CPU, but for the FPU they are prefixed with `V`. So `LDR` becomes `VLDR`. There are even a few advanced functions not present in the standard CPU such as `VSQRT`.

Standard	**FPU**
LDR	VLDR
STR	VSTR
MOV	VMOV
ADD	VADD
MLA	VMLA
CMP	VCMP
N/A	VABS
N/A	VSQRT

We often suffix the commands with something like `F32/F64` or `U32/S32`. These signify float/double, and unsigned/signed. They are appended to the instruction like `VLDR.F32.S32`. These are used to convert from one format to another.

6 GPIO

To access our GPIO devices, we use memory mapped IO. This is where a section of the memory addresses are mapped to the IO device such as LED and speaker. All the bits we access through software are found in these registers such as the clock enable, GPIO port enable, GPIO mode, and so on.

The GPIO speed can be configured where a higher speed is faster, but uses more power and has more noise.

The GPIO pin can be set up in different modes such as input, output, analog, or alternate function.

The GPIO pin can be either in open drain, or pull push mode. This changes how the physical transistors are set up. Open drain disables the PMOS so it cannot drive HI at all. It can only drive LOW or floating. We also have a pull up or pull down resistor that can also be enabled/disabled.

We have a certain number of GPIO devices, each device has a certain number of pins. So we could have 8 devices with 16 pins each for a total of 128 inputs/outputs.

6.1 Pin Output/Input modes

For output pins, we can either have the pin in PUSH/PULL mode, or OPEN DRAIN mode. If it is in PUSH/PULL, the pin can either be driven HI, or LO. If it is in OPEN DRAIN mode, then the PMOS is disabled, and we can only drive LO. If we do not drive LO, then the pin will be floating.

For input (and output) pins, we can either use a pull up or pull down resistor. If we use pull up, then the line is by default at HI. If we use pull down, then the line is by default at LO.

6.2 Debouncing

Debouncing is needed for any sort of input switch. If we do not have debouncing, then the hardware will detect multiple presses per single button press, which is not what we want. We can either debounce using hardware with an RC circuit (or an SR latch and dual throw switch if we are fancy), or through software with a delay.

6.3 Interrupts

An interrupt is an external signal sent into the CPU for processing. So when a button is pressed, or motion is detected on a motion detector, an interrupt will be generated.

7 Timers

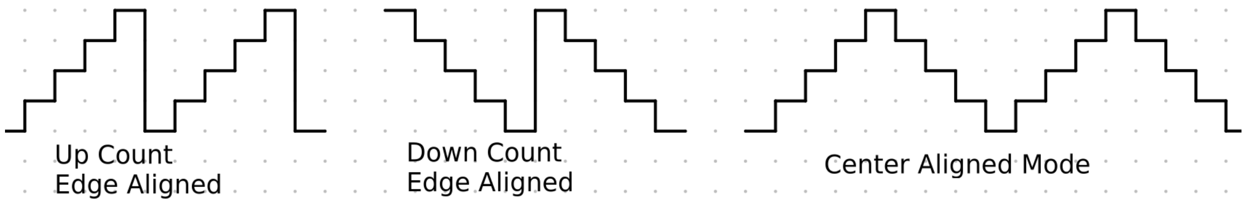
Timers generate interrupts at a fixed interval. This is what the SYSTICK driver does, it generates an interrupt every say 1ms. SYSTICK is a hardware component in ARM CORTEX M.

We have a reload value stored in a register (`ARR`). This is the starting value that the counter starts up, and then counts down to 0 when it creates the interrupt.

We also have a current value register to get the current value of the counter. This needs to be cleared on startup before running a timer since it has a random value.

We also often have some other timers that are independent of the processor. These are useful for peripheral devices. They can either operate in **capture** mode, or **compare** mode (useful for PWM, 1 if below Comp value, 0 if above, [or opposite depending on PWM mode]). Capture will record the time when events occur, and compare will trigger events at specific times.

When counting, we can either use down counting, up counting, or center counting.



We can also change the repetitions. This allows us to determine the number of reloads between events. This uses the repetition counter register (`RCR`). This is done using an input prescaler which is basically another timer.

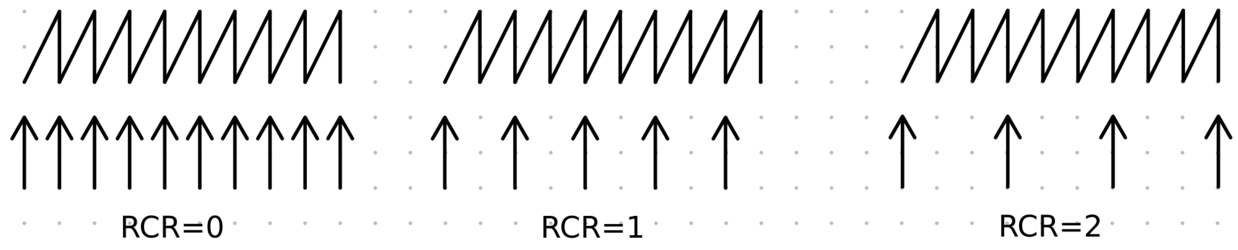


Figure 3:

7.1 Systick

This is the build in timer that generates interrupts at a fixed interval. We can enable it by enabling the systick interrupt (found in `SysTick->CTRL`), enabling the systick clock source, and enabling the actual systick. We also have to convigure the NVIC to actually make the interrupt.

Ex. If we have `SysTick_LOAD=499999`, and CPU Freq is 25 MHz, what is the period between two systick interrupts?

To calculate this, I need to know that the SYSTICK counts down from 499999 to 0, and it takes that many clock cycles to do so. So the 25MHz clock cycles becomes 50.

Now we divide the 1s (Hz is cycles per 1s) by 50 to get 20ms.

8 Direct Memory Access (DMA)

By default, when we copy from a peripheral to memory, we need to go through the CPU. This is inefficient, the DMA lets us directly go from the peripheral to the memory. This is better since it uses much less CPU. We have a seperate DMA controller that can initialise the transfers. The data can either flow through the DMA, or directly fly by to the memory (does not directly go through DMA).

DMA has less CPU overhead compared to traditional memory access, and it is faster.

9 DAC and ADC (Analog Interfacing)

When interfacing with analogue devices, we need to either convert a digital signal to analogue, or convert an analogue signal to digital, depending on whether or not it is an output or input.

9.1 DAC

For the digital to analog converter, we have a certain number of bits which gives the range of analog values.

The DAC can be implemented in a few ways such as a string of resisters with switches acting as a voltage divider, or as a combination of resistors with switches (same idea, but we use combinations of resistors for more possible outputs with less resistors), and we can also use PWM with a filter.

Ex. How many resistors are required for a simple 6 bit string DAC?

This is just a list of resistors that act as a voltage divider, and we can turn on any one resistor. We need 63 resistors to give 64 options since $64 = 2^6$.

9.2 ADC

When converting from analog to digital, we need to sample at a certain frequency, and each sample again has a bit depth (resolution). The sampling rate that we need to reconstruct the analog signal is called the Nyquist rate which is twice the max frequency of the signal.

One way to do this is to use Successive Approximations (SAR). This basically uses a comparator to test all the digital values compared to the raw analog signal using a binary search.

Ex. We have a 12 bit SAR ADC, where it takes 4 cycles to sample the signal. What is the total sampling time?

For the total sampling time, we need to account for the time to sample the signal, and then the time to determine the actual signal using the SAR. Since this is a binary search, it will take at most 12 cycles.

$$T = 4 + 12 = 16 \text{ cycles}$$

10 Serial Communications

10.1 I2C

I2C is a 2 wire communication standard. It has a data line (SDA), and a clock line (SCL). It is relatively low speed. Both wires use open drain drivers with a pull up resistor (so are logic 1 by default).

- START is when SDA has falling edge when SCL is HI

- STOP is when SDA has rising edge when SCL is HI
- DATA can only change when SCL is LO

We have a target, which can be 7 or 10 bits, and then we have the data. If we have more than 1 controller, then the arbitration works by each controller checking SDA. If SDA is LO when it is driving SDA HI, then that controller backs off and waits. This works since SDA is pulled up to HI.

10.2 SPI

SPI (Serial Peripheral Interface) is a full duplex connection. It has the following wires:

- SCLK - Clock
- $\overline{SS}$ - Peripheral Select (n SS lines for n slaves) [Active LO]
- MOSI - Master Out, Slave In
- MISO - Master In, Slave Out

We have some parameters for the SPI clock such as CPHA [phase] (0 uses first edge of clock, 1 uses second edge) and CPOL [polarity] (0 has idle low clk, 1 has idle high clk).

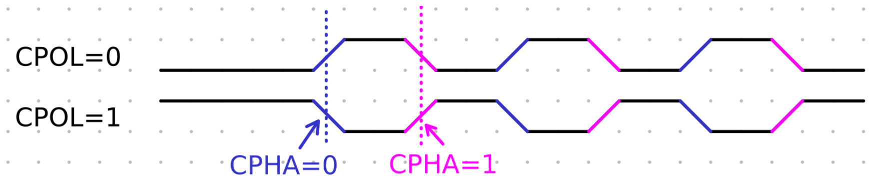


Figure 4: CPOL vs CPHA

10.3 UART

UART is a very widely used protocol to transfer data asynchronously (without a clock wire). It uses only two wires, Tx (transmit) and Rx (receive).

The data frame is idle HI, and contains a start bit followed by a certain number of data bits (usually 8), optionally a parity bit, and then a stop bit.

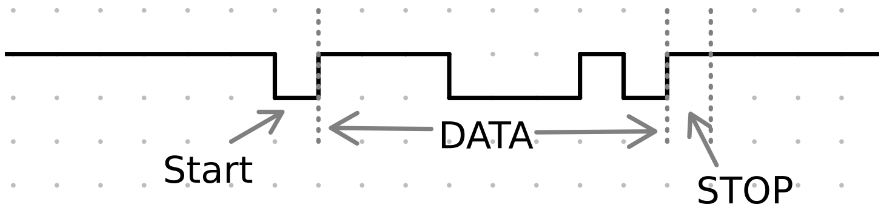


Figure 5: UART Bit Stream

Since UART is asynchronous, the receiver and transmitter must each know the clock rate (to within 10% error). This is known as the BAUD rate (bits per second).

The receiver typically oversamples the signal to get each pulse in the middle (to account for error). It is often 8x or 16x oversampling.

When the receiver receives a byte, and puts it in the register, it sends an interrupt. Then upon reading from that register, the interrupt is cleared and it can be populated with the next received byte.

UART is transmitted LSB first.

There are a few protocols to send this bit stream such as RS-232 where Logic 1 is +5V to +15V, and Logic 0 is -5V to -15V.

10.4 USB

USB is a much more complicated protocol which is not covered in depth in this course. USB has a Data + and Data -. These form a differential pair. This is the only wire we care about.

We use a Non Return to Zero Inverted scheme for the bitstream. This is where Logic 0 is represented by a change in voltage, and Logic 1 by the same voltage.

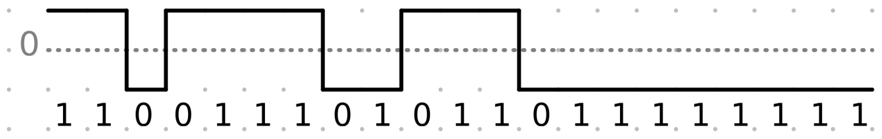


Figure 6: USB NZRI Bit Stream

In terms of actually framing the data, it is very complicated, where there are multiple layers.

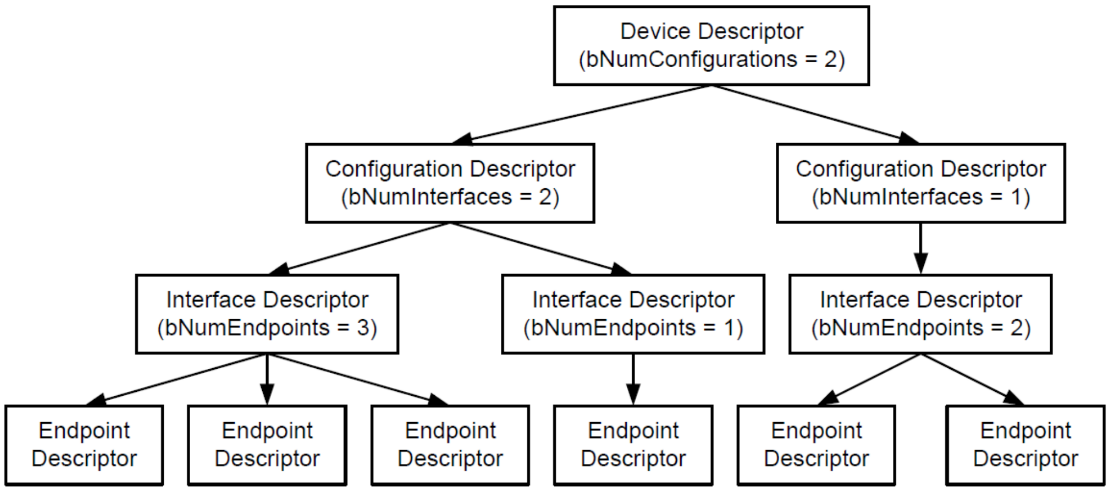

Device Descriptor (bNumConfigurations = 2)
Configuration Descriptor (bNumInterfaces = 2)
Configuration Descriptor (bNumInterfaces = 1)
Interface Descriptor (bNumEndpoints = 3)
Interface Descriptor (bNumEndpoints = 1)
Interface Descriptor (bNumEndpoints = 2)
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor

Figure 7: USB Layers Overview

11 The Bus (Bus Interfaces)

The BUS is basically a huge set of wires that everything is connected to. This include the CPU, MEMORY, GPIO, SPI, I2C, TIMER, ADC, DAC, RTC, and so on.

We have two main architectures for the bus and how it connects to the rest of the system. Von-Neumann means that the instructions and data share memory, and therefore share a bus interface. In a Harvard architecture, the instruction memory is seperate from the data memory, and there are two seperate bus interfaces.

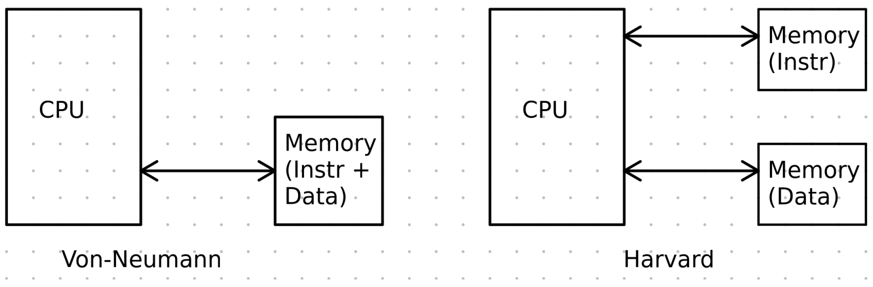


Figure 8: Von Neumann vs Harvard Architecture

We usually use a matrix to connect all the master devices to the slave devices. This matrix is good because multiple devices can communicate at once if they are different masters and different slaves.

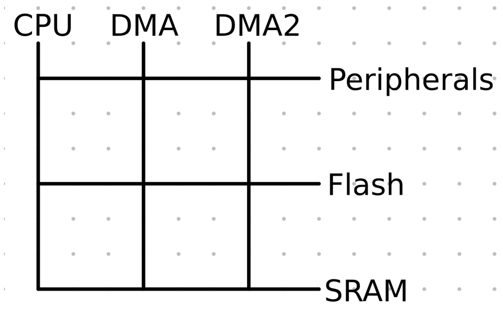


Figure 9: Bus Matrix Example

We have a bunch of different bus architectures used in ARM such as:

- Advanced Peripheral Bus (APB)
- Advanced System Bus (ASB)
- Advanced High-performance Bus (AHB)
- Advanced Extensible Interface (AXI)

Typically we use APB or AHB. APB is lower power, and simpler, while AHB is higher performance partially because it is pipelined.

12 Appendix

12.1 Bitfield Manipulation

Some useful information is how to manipulate bits in a bitfield.

What	**C like Code**	**Logic Operation**
Set masked Bits	`data |= mask`	OR
Clear masked Bits	`data &= ~mask`	Negated AND
Toggle masked Bits	`data ^= mask`	XOR
Test any bits for Non Zero	`data & mask`	AND

12.2 Reference Sheets

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Function Call/Return		**Operation**	**Notes**	**Clock Cycles**
BL	*label*	LR ← *return address;* PC ← *address of label*	BL is used to call a function	2-4
BLX	R_n	LR ← *return address;* PC ← R_n		
BX	R_n	PC ← R_n	BX LR is used as function return	
B	*label*	PC ← *address of label*		

Load Integer Constant		**Operation**	**Flags**	**Notes**	**Clock Cycles**
ADR	R_d,*label*	R_d ← *address of label*		PC-4095 ≤ *address* ≤ PC+4095	1
MOV{S}	R_d, *constant*	R_d ← *constant*	NZ	0≤*constant*≤255 (FF_{16}) & a few others	
MVN{S}	R_d, *constant*	R_d ← ~*constant*	NZ	0≤*constant*≤255 (FF_{16}) & a few others	
MOVW	R_d, *constant*	R_d ← *constant*		0≤*constant*≤65535 ($FFFF_{16}$)	
MOVT	R_d, *constant*	R_d<31..16> ← *constant*		0≤*constant*≤65535 ($FFFF_{16}$)	

Load/Store Memory		**Operation**	**Bits**	**Notes**	**Clock Cycles**
LDRB	R_d,[*address mode*]	R_d ← *memory*<7..0> *(zero extended)*	8	R_d<31..8> ← 24 0's	2
LDRSB	R_d,[*address mode*]	R_d ← *memory*<7..0> *(sign extended)*	8	R_d<31..8> ← 24 copies of R_d<7>	
LDRH	R_d,[*address mode*]	R_d ← *memory*<15..0> *(zero extended)*	16	R_d<31..16> ← 16 0's	
LDRSH	R_d,[*address mode*]	R_d ← *memory*<15..0> *(sign extended)*	16	R_d<31..16> ← 16 copies of R_d<16>	
LDR	R_d,[*address mode*]	R_d ← *memory*<31..0>	32		
LDRD	R_t,R_{t2},[*address mode*]	R_{t2}.R_t ← *memory*<63..0>	64	Can't use register offset adrs mode	3
STRB	R_d,[*address mode*]	R_d → *memory*<7..0>	8		2
STRH	R_d,[*address mode*]	R_d → *memory*<15..0>	16		
STR	R_d,[*address mode*]	R_d → *memory*<31..0>	32		
STRD	R_t,R_{t2},[*address mode*]	R_{t2}.R_t → *memory*<63..0>	64	Can't use register offset adrs mode	3

Load/Store Multiple		**Operation**	**Notes**	**Clock Cycles**
POP	{*register list*}	*registers* ← *memory*[SP]; SP+=4×#*registers*	*regs*: Not SP; PC/LR, but not both	1 + #*registers*
PUSH	{*register list*}	SP−=4×#*registers*; *registers* → *memory*[SP]	*regs:* Neither SP or PC.	
LDMIA	R_n!,{*register list*}	*registers* ← *memory*[R_n]	if "!" is appended, then R_n += 4×#*registers*	
STMIA	R_n!,{*register list*}	*registers* → *memory*[R_n]		
LDMDB	R_n!,{*register list*}	*registers* ← *memory*[R_n − 4×#*registers*]	if "!" is appended, then R_n −= 4×#*registers*	
STMDB	R_n!,{ *register list*}	*registers* → *memory*[R_n − 4×#*registers*]		

Move / Add / Subtract		**Operation**	**Flags**	**operand2 options:**	**Clock Cycles**
MOV{S}	R_d,R_n	R_d ← R_n	NZ		1
ADD{S}	R_d,R_n,*operand2*	R_d ← R_n + *operand2*	NZCV	1. *constant* 2. R_m *(a register)* 3. R_m,*shift* *(Any kind of shift)*	
ADC{S}	R_d,R_n,*operand2*	R_d ← R_n + *operand2* + C	NZCV		
SUB{S}	R_d,R_n,*operand2*	R_d ← R_n − *operand2*	NZCV		
SBC{S}	R_d,R_n,*operand2*	R_d ← R_n − *operand2* + C − 1	NZCV		
RSB{S}	R_d,R_n,*operand2*	R_d ← *operand2* − R_n	NZCV		

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Multiply / Divide		***Operation***	***Flags***	***Notes***	***Clock Cycles***
MUL{S}	R_d,R_n,R_m	$R_d \leftarrow (R_n \times R_m)<31..0>$	NZC	32 ← 32×32; C←undefined	
MLA	R_d,R_n,R_m,R_a	$R_d \leftarrow R_a + (R_n \times R_m)<31..0>$		32 ← 32 + 32×32	
MLS	R_d,R_n,R_m,R_a	$R_d \leftarrow R_a - (R_n \times R_m)<31..0>$		32 ← 32 - 32×32	
SMMUL{R}	R_d,R_n,R_m	$R_d \leftarrow (R_n \times R_m)<63..32>$		Upper half of signed 64-bit product;	
SMMLA{R}	R_d,R_n,R_m,R_a	$R_d \leftarrow R_a + (R_n \times R_m)<63..32>$		Append R: Round towards +∞ (Adds	1
SMMLS{R}	R_d,R_n,R_m,R_a	$R_d \leftarrow R_a - (R_n \times R_m)<63..32>$		0x80000000 to the 64-bit product)	
$\begin{bmatrix}S\\U\end{bmatrix}$MULL	R_{dlo},R_{dhi},R_n,R_m	$R_{dhi}R_{dlo} \leftarrow R_n \times R_m$		**S**igned/**U**nsigned: 64 ← 32x32	
$\begin{bmatrix}S\\U\end{bmatrix}$MLAL	R_{dlo},R_{dhi},R_n,R_m	$R_{dhi}R_{dlo} \leftarrow R_{dhi}R_{dlo} + R_n \times R_m$		**S**igned/**U**nsigned: 64 ← 64 + 32×32	
$\begin{bmatrix}S\\U\end{bmatrix}$DIV	R_d,R_n,R_m	$R_d \leftarrow R_n / R_m$		**S**igned/**U**nsigned: 32 ← 32÷32	2-12

Saturating Instructions		***Operation***	***Min***	***Max***	***operand2 options***	***Clock Cycles***
SSAT	R_d,n,*operand2*	$R_d \leftarrow$ *operand2*	-2^{n-1}	$2^{n-1}-1$	1. R_m *(a register)* 2. R_m,ASR *constant*	
USAT	R_d,n, *operand2*	$R_d \leftarrow$ *operand2*	0	2^n-1	3. R_m,LSL *constant*	1
QADD	R_d,R_n,R_m	$R_d \leftarrow R_n + R_m$	-2^{31}	$2^{31}-1$	(Q ← 1 if saturates)	
QSUB	R_d,R_n,R_m	$R_d \leftarrow R_n - R_m$				

SIMD Signed Saturating ADD/SUB		***Operation***	***Min to Max***	***Notes***	***Clock Cycles***
QADD$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] + R_m[bits]	8: -2^7 to $+2^7-1$	For bytes 0-3: bits 7..0, 15..8, 23..16, & 31..24	1
QSUB$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] - R_m[bits]	16: -2^{15} to $+2^{15}-1$	(No flags affected)	

SIMD Unsigned Saturating ADD/SUB		***Operation***	***Min to Max***	***Notes***	***Clock Cycles***
UQADD$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] + R_m[bits]	8: 0 to 2^8-1	For halfwords 0 and 1: bits 15..0 & 31..16	1
UQSUB$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] - R_m[bits]	16: 0 to $2^{16}-1$	(No flags affected)	

SIMD Signed Non-Saturating ADD/SUB		***Operation***	***GE Flags***	***Notes***	***Clock Cycles***
SADD$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] + R_m[bits]	sum ≥ 0 ? 1 : 0	Parallel operations: Four 8-bit operations,	1
SSUB$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] - R_m[bits]	diff ≥ 0 ? 1 : 0	or two 16-bit operations	

SIMD Unsigned Non-Saturating ADD/SUB		***Operation***	***GE Flags***	***Notes***	***Clock Cycles***
UADD$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] + R_m[bits]	overflow ? 1 : 0	Parallel operations: Four 8-bit operations,	1
USUB$\left[\frac{8}{16}\right]$	R_d,R_n,R_m	R_d[bits] ← R_n[bits] - R_m[bits]	diff ≥ 0 ? 1 : 0	or two 16-bit operations	

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Q and GE Flag Instructions		***Operation***	***Notes***	***Clock Cycles***
SEL	R_d,R_n,R_m	R_d[bits] ← (GE[byte] = 1) ? R_n[bits] : R_m[bits]	For bytes 0-3: bits 7..0, 15..8, 23..16, & 31..24	1
MRS	R_d,APSR	R_d<31..27> ← NZCVQ R_d<19..16> ← GE flags	All other bots of R_d are filled with zeroes.	
MSR	APSR_nzcvq,R_n	NZCVQ ← R_n<31..27>	Other flags in the PSR are not affected.	
MSR	APSR_g,R_n	GE flags ← R_n<19..16>		

SIMD Multiply Instructions		***Operation***	***Notes***	***Clock Cycles***
SMUAD	R_d,R_n,R_m	R_d ← R_n<15..00>×R_m<15..00> + R_n<31..16>×R_m<31..16>	Sets Q flag if an addition or subtraction overflows; does <u>not</u> saturate.	1
SMUSD	R_d,R_n,R_m	R_d ← R_n<15..00>×R_m<15..00> − R_n<31..16>×R_m<31..16>		
SMLAD	R_d,R_n,R_m,R_a	R_d ← R_a + R_n<15..00>×R_m<15..00> + R_n<31..16>×R_m<31..16>		
SMLSD	R_d,R_n,R_m,R_a	R_d ← R_a + R_n<15..00>×R_m<15..00> − R_n<31..16>×R_m<31..16>		
SMLALD	R_{dlo},R_{dhi},R_n,R_m	R_{dhi}.R_{dlo} += Rn<15..00>×R_m<15..00> + Rn<31..16>×R_m<31..16>		
SMLSLD	R_{dlo},R_{dhi},R_n,R_m	R_{dhi}.R_{dlo} += Rn<15..00>×R_m<15..00> − Rn<31..16>×R_m<31..16>		

Appending "X" to instruction mnemonic changes operand2s to Rn<15..00>×Rm<31..16> and Rn<31..16> × Rm<15..00>.

Signed Multiply Halfwords		***Operation***	***Notes***	***Clock Cycles***
SMULBB	R_d,R_n,R_m	R_d ← R_n<15..00> × R_m<15..00>	32 ← 16×16	1
SMULBT	R_d,R_n,R_m	R_d ← R_n<15..00> × R_m<31..16>		
SMULTB	R_d,R_n,R_m	R_d ← R_n<31..16> × R_m<15..00>		
SMULTT	R_d,R_n,R_m	R_d ← R_n<31..16> × R_m<31..16>		

Pack Halfwords		***Operation***	***operand2 options:***	***Notes***	***Clock Cycles***
PKHBT	R_d,R_n,*operand2*	**B**tm: R_d<15..00> ← R_n<15..00> **T**op: R_d<31..16> ← *operand2*<31..16>	1. R_m *(a register)* 2. R_m,LSL *constant* 3. R_m,ASR *constant*	*Shift constants:* LSL: 1-31 ASR: 1-32	1
PKHTB	R_d,R_n,*operand2*	**T**op: R_d<31..16> ← R_n<31..16> **B**tm: R_d<15..00> ← *operand2*<15..00>			

Compare Instructions		***Operation***	***operand2 options:***	***Notes***	***Clock Cycles***
CMP	R_n,*operand2*	R_n − *operand2*	1. *constant* 2. R_m *(a register)* 3. R_m,*shift* *(any kind of shift)*	Updates: NZCV	1
CMN	R_n,*operand2*	R_n + *operand2*		Updates: NZCV	
TST	R_n,*operand2*	R_n & *operand2*		Updates: NZC	
TEQ	R_n,*operand2*	R_n ^ *operand2*		Updates: NZC	

Zero/Sign-Extend Instructions		***Operation***	***operand2 options:***	***Clock Cycles***
$\left[\begin{smallmatrix}S\\U\end{smallmatrix}\right]$XTB	R_d, *operand2*	R_d ← Sign (S) extend or Unsigned (U) extend operand2<7..0>	1. R_m *(a register)* 2. R_m,ROR *constant* *(constant=8, 16 or 24)*	1
$\left[\begin{smallmatrix}S\\U\end{smallmatrix}\right]$XTH	R_d, *operand2*	R_d ← Sign (S) extend or Unsigned (U) extend operand2<15..0>		

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Conditional Branch Instructions		***Operation***	***Notes***	***Clock Cycles***
Bcc	*label*	Branch to *label* if "cc" is true	"cc" is a condition code	1 (Fail) or 2-4
CBZ	R_n, *label*	Branch to *label* if R_n=0	Can't use in an IT block	
CBNZ	R_n, *label*	Branch to *label* if $R_n \neq 0$	Can't use in an IT block	
IT$c_1c_2c_3$	*condition code*	Each c_i is one of T, E, or *empty*	Controls 1-4 instructions	1

Shift Instructions			***Operation***	***Flags***	***operand2 options***	***Notes***	***Clock Cycles***
ASR{S}	R_d,R_n,*operand2*	*// 1-32 bits*	$R_d \leftarrow R_n$ >> *operand2* (arithmetic shift right)	NZC	1. *constant*	Sign extends	1
LSL{S}	R_d,R_n,*operand2*	*// 1-31 bits*	$R_d \leftarrow R_n$ << *operand2* (logical shift left)	NZC	2. R_m *(a register)*	Zero fills	
LSR{S}	R_d,R_n,*operand2*	*// 1-32 bits*	$R_d \leftarrow R_n$ >> *operand2* (logical shift right)	NZC	When operand2 is a constant: LSL: shifts 0-31 bits;		
ROR{S}	R_d,R_n,*operand2*	*// 1-31 bits*	$R_d \leftarrow R_n$ >> *operand2* (rotate right)	NZC	ASR,LSR,ROR: 1-32 bits	right rotate	
RRX{S}	R_d,R_n	*// 1 bit*	$R_d \leftarrow R_n$ >> 1; R_d<31> $\leftarrow$ C; C $\leftarrow R_n$<0>	NZC	RRX shifts only by 1 bit.	33-bit rotate w/C	

Bitwise Instructions		***Operation***	***Flags***	***operand2 options***	***Notes***	***Clock Cycles***
AND{S}	R_d,R_n,*operand2*	$R_d \leftarrow R_n$ & *operand2*	NZC	1. *constant*		1
ORR{S}	R_d,R_n,*operand2*	$R_d \leftarrow R_n$ | *operand2*	NZC	2. R_m *(a register)*		
EOR{S}	R_d,R_n,*operand2*	$R_d \leftarrow R_n$ ^ *operand2*	NZC	3. R_m,*shift*		
BIC{S}	R_d,R_n,*operand2*	$R_d \leftarrow R_n$ & ~*operand2*	NZC	*(Any kind of shift)*		
ORN{S}	R_d,R_n,*operand2*	$R_d \leftarrow R_n$ | ~*operand2*	NZC			
MVN{S}	R_d,*operand2*	$R_d \leftarrow$ ~*operand2*	NZC			

Bitfield Instructions		***Operation***	***Notes***	***Clock Cycles***
BFC	R_d,*lsb*,*width*	*SelectedBitfieldOf*(R_d) $\leftarrow$ 0		1
BFI	R_d,R_n,*lsb*,*width*	*SelectedBitfieldOf*(R_d) $\leftarrow$ *LSBitsOf*(R_n)		
SBFX	R_d,R_n,*lsb*,*width*	$R_d \leftarrow$ *SelectedBitfieldOf*(R_n)	Sign extends	
UBFX	R_d,R_n,*lsb*,*width*	$R_d \leftarrow$ *SelectedBitfieldOf*(R_n)	Zero extends	

Bits / Bytes / Words		***Operation***	***Notes***	***Clock Cycles***
CLZ	R_d,R_n	$R_d \leftarrow$ *CountLeadingZeroesOf*(R_n)	#leading 0's = 0-32	1
RBIT	R_d,R_n	$R_d \leftarrow$ *ReverseBitOrderOf*(R_n)		
REV	R_d,R_n	$R_d \leftarrow$ *ReverseByteOrderOf*(R_n)		

Pseudo-Instructions		***Operation***	***Flags***	***Replaced by***	***Clock Cycles***
LDR	R_d,=*constant*	$R_d \leftarrow$ *constant*		MOV, MVN, MOVW, or LDR	1
NEG	R_d,R_n	$R_d \leftarrow -R_n$	NZCV	RSBS R_d,R_n,0	
CPY	R_d,R_n	$R_d \leftarrow R_n$		MOV R_d,R_n	

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Floating-Point PUSH/POP		***Operation***	***Clock Cycles***
VPUSH	{*FP register list*}	SP −= 4×#*registers*, copy *registers* to *memory*[SP]	1 + #*registers*
VPOP	{*FP register list*}	Copy *memory*[SP] to *registers*, SP += 4×# *registers*	

Floating-Point Load Constant			***Clock Cycles***
VMOV	S_d,*fpconstant*	*fpconstant* must be $\pm m \times 2^{-n}$, $(16 \le m \le 31;\ 0 \le n \le 7)$	1

Floating-Point Copy Registers		***Operation***	***Clock Cycles***
VMOV	S_d,S_m	$S_d \leftarrow S_m$	1
VMOV	R_d,S_m	$R_d \leftarrow S_m$	
VMOV	S_d,R_m	$S_d \leftarrow R_m$	
VMOV	R_t,R_{t2},S_m,S_{m+1}	$R_t \leftarrow S_m$; $R_{t2} \leftarrow S_{m+1}$ (S_m, S_{m+1} adjacent regs)	2
VMOV	S_m,S_{m+1},R_t,R_{t2}	$S_m \leftarrow R_t$; $S_{m+1} \leftarrow R_{t2}$ (S_m, S_{m+1} adjacent regs)	

Floating-Point Load Registers		***Operation***	***Clock Cycles***
VLDR	S_d,[R_n]	$S_d \leftarrow$ *memory32*[R_n]	2
VLDR	S_d,[R_n,*constant*]	$S_d \leftarrow$ *memory32*[R_n + *constant*]	
VLDR	S_d,*label*	$S_d \leftarrow$ *memory32*[Address of *label*]	
VLDR	D_d,[R_n]	$D_d \leftarrow$ *memory64*[R_n]	3
VLDR	D_d,[R_n,*constant*]	$D_d \leftarrow$ *memory64*[R_n + *constant*]	
VLDR	D_d,*label*	$D_d \leftarrow$ *memory64*[Address of *label*]	
VLDMIA	R_n!,{*FP register list*}	*FP registers* ← *memory*, R_n = lowest address; Updates R_n if write-back flag (!) is included.	1 + #*registers*
VLDMDB	R_n!,{*FP register list*}	*FP registers* ← *memory*, R_n-4 = highest address; Must append (!) and always updates R_n	

Floating-Point Store Registers		***Operation***	***Clock Cycles***
VSTR	S_d,[R_n]	$S_d \rightarrow$ *memory32*[R_n]	2
VSTR	S_d,[R_n,*constant*]	$S_d \rightarrow$ *memory32*[R_n + *constant*]	
VSTR	D_d,[R_n]	$D_d \rightarrow$ *memory64*[R_n]	3
VSTR	D_d,[R_n,*constant*]	$D_d \rightarrow$ *memory64*[R_n + *constant*]	
VSTMIA	R_n!,{*FP register list*}	*FP registers* → *memory*, R_n = lowest address; Updates R_n if write-back flag (!) is included.	1 + #*registers*
VSTMDB	R_n!,{*FP register list*}	*FP registers* → *memory*, R_n-4 = highest address; Must append (!) and always updates R_n	

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Floating-Point Convert Representation		***Operation***		***Clock Cycles***
VCVT.F32.U32	S_d,S_m	S_d ← (float) S_m, *where S_m is an unsigned integer*		1
VCVT.F32.S32	S_d,S_m	S_d ← (float) S_m, *where S_m is a 2's comp integer*		
VCVT{R}.U32.F32	S_d,S_m	S_d ← (uint32_t) S_m	*Rounded if suffix "R" is appended* using current rounding mode (FPSCR bits 23 and 22, default is nearest even)	
VCVT{R}.S32.F32	S_d,S_m	S_d ← (int32_t) S_m		

Floating-Point Arithmetic		***Operation***	***Clock Cycles***
VADD.F32	S_d,S_n,S_m	$S_d \leftarrow S_n + S_m$	1
VSUB.F32	S_d,S_n,S_m	$S_d \leftarrow S_n - S_m$	
VNEG.F32	S_d,S_m	$S_d \leftarrow -S_m$	
VABS.F32	S_d,S_m	$S_d \leftarrow \| S_m \|$; *(clears FPU sign bit, N)*	
VMUL.F32	S_d,S_n,S_m	$S_d \leftarrow S_n \times S_m$	
VDIV.F32	S_d,S_n,S_m	$S_d \leftarrow S_n \div S_m$	14
VSQRT.F32	S_d,S_m	$S_d \leftarrow \sqrt{S_m}$	
VMLA.F32	S_d,S_n,S_m	$S_d \leftarrow S_d + S_n \times S_m$	3
VMLS.F32	S_d,S_n,S_m	$S_d \leftarrow S_d - S_n \times S_m$	

Floating-Point Compare		***Operation***	***Clock Cycles***
VCMP.F32	S_d,S_m	Computes S_d - S_m and updates *FPU Flags* in FPSCR	1
VCMP.F32	S_d,0.0	Computes S_d - 0 and updates *FPU Flags* in FPSCR	
VMRS	APSR_nzcv,FPSCR	*Core CPU Flags* ← *FPU Flags* (Needed between VCMP.F32 and conditional branch)	

Addressing Modes for ***floating-point*** load and store instructions (VLDR & VSTR):

Addressing Mode	***Syntax***	***Meaning***	***Example***
Immediate Offset	[R_n]	*address* = R_n	[R5]
	[R_n,*constant*]	*address* = R_n + *constant*	[R5,100]

Shift Codes:

Any of these may be applied as the "*shift*" option of "*operand2*" in Move / Add / Subtract, Compare, and Bitwise Groups.

Shift Code	***Meaning***	***Notes***
LSL *constant*	**L**ogical **S**hift **L**eft by *constant* bits	Zero fills; 0 ≤ *constant* ≤ 31
LSR *constant*	**L**ogical **S**hift **R**ight by *constant* bits	Zero fills; 1 ≤ *constant* ≤ 32
ASR *constant*	**A**rithmetic **S**hift **R**ight by *constant* bits	Sign extends; 1 ≤ *constant* ≤ 32
ROR *constant*	**RO**tate **R**ight by *constant* bits	1 ≤ *constant* ≤ 32
RRX	**R**otate **R**ight e**X**tended (with carry) by 1 bit	

Cortex-M4F Instructions used in *ARM Assembly for Embedded Applications* (ISBN 978-1-09254-223-4)

Addressing Modes for ***integer*** load and store instructions (LDR, STR, etc.):

Any of these may be used with all variations of LDR/STR except LDRD/STRD, which may not use Register Offset Mode.

Addressing Mode	*Syntax*	*Meaning*	*Example*
Immediate Offset	[R_n]	*address* = R_n	[R5]
	[R_n,*constant*]	*address* = R_n + *constant*	[R5,100]
Register Offset	[R_n,R_m]	*address* = R_n + R_m	[R4,R5]
	[R_n,R_m,LSL *constant*]	*address* = R_n + (R_m << *constant*)	[R4,R5,LSL 3]
Pre-Indexed	[R_n,*constant*]!	$R_n \leftarrow R_n$ + *constant*; *address* = R_n	[R5,100]!
Post-Indexed	[R_n],*constant*	*address* = R_n; $R_n \leftarrow R_n$ + *constant*	[R5],100

Condition Codes:

If appended to an FPU instruction within an IT block, the condition code precedes any extension. (E.g., VADD**GT**.F32)

Condition Code	*CMP Meaning*	*VCMP Meaning*	*Requirements*
EQ (Equal)	==	==	Z = 1
NE (Not Equal)	!=	!= or unordered	Z = 0
HS (Higher or Same)	unsigned ≥	≥ or unordered	C = 1 *Note: Synonym for "CS" (Carry Set)*
LO (Lower)	unsigned <	<	C = 0 *Note: Synonym for "CC" (Carry Clear)*
HI (Higher)	unsigned >	> or unordered	C = 1 && Z = 0
LS (Lower or Same)	unsigned ≤	≤	C = 0 || Z = 1
GE (Greater Than or Equal)	signed ≥	≥	N = V
LT (Less Than)	signed <	< or unordered	N ≠ V
GT (Greater Than)	signed >	>	Z = 0 && N = V
LE (Less Than or Equal)	signed ≤	≤ or unordered	Z = 1 || N ≠ V
CS (Carry Set)	unsigned ≥	≥ or unordered	C = 1 *Note: Synonym for "HS" (Higher or Same)*
CC (Carry Clear)	unsigned <	<	C = 0 *Note: Synonym for "LO" (Lower)*
MI (Minus)	negative	<	N = 1
PL (Plus)	non-negative	≥ or unordered	N = 0
VS (Overflow Set)	overflow	unordered	V = 1
VC (Overflow Clear)	no overflow	not unordered	V = 0
AL (Always)	unconditional	unconditional	Always true

Notes:
1. This is only a partial list of the most commonly-used ARM Cortex-M4 instructions.
2. Clock Cycle counts do not include delays due to stalls when an instruction must wait for the previous instruction to complete.
3. There are magnitude restrictions on immediate constants; see ARM documentation for more information.