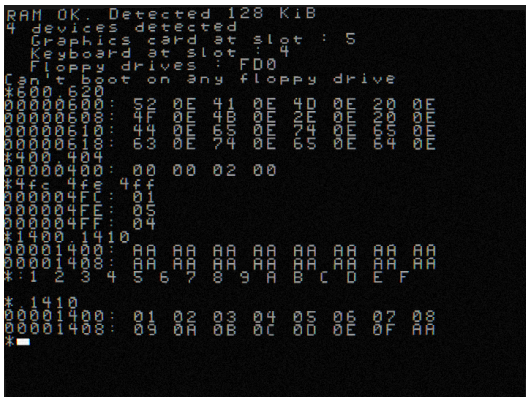
INTRO TO ASSEMBLY LANGUAGE



Machine code is binary-encoded instructions that can be executed directly by a computer. It is the lowest-level representation of a computer program. When we run a program, the machine code (typically a series of hexadecimal numbers, or binary) is loaded into RAM. The processor then takes a sequence of one or more bytes - which correspond to an instruction defined by the hardware's Instruction Set Architecture (ISA) - and executes it. It continues reading a sequence of bytes at a time in this fashion until the program terminates.

Assembly language was created as a shorthand for machine code; it is (somewhat) easier for humans to write/understand, while (nearly) maintaining a 1:1 correspondence to machine code. It accomplishes this by using *mnemonics*: symbolic labels that replace the hexadecimal/binary sequences corresponding to instructions in the machine code.

For example, the binary sequence below is described using mnemonics with ADD Rd, Rn, Rm. Each symbolic label is a synonym for some sub-sequence of bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

This makes it easy for an **assembler** to translate in the opposite direction, from assembly language into machine code, when the program is executed by the machine.

High-level programming languages, which provide *source code*, can run independently of processor type, while machine code and assembly language are *hardware dependent*. These languages (Java, C, Python, etc.) are much easier to write/understand, but they still have inefficiencies compared to assembly. Nowadays, processors have gotten incredibly fast at compiling/interpreting high-level languages, so this performance decrease is negligible.

Assembly language is really only useful for:

- Hardware Control
    - Embedded programming on a small microcontroller with limited memory. Assembly programs occupy less memory, and they allow you

to accurately and directly control I/O ports, RAM, and take advantage of specialized features of the hardware.
    - Multi-tasking Operating Systems must pass control from one task to the next by saving and restoring the Stack Pointer register. This can only be done in assembly!
- Understanding Computing
    - Seeing how compilers work so that you can write efficient high-level code.
    - Learning how your CPU works. High-level code makes use of a lot of abstraction, while assembly forces the programmer to understand exactly what the hardware is doing during the execution of a program.
- Select cases where timing is critical and marginal improvements in performance really count.

Instruction Set Architecture (ISA) is the interface between hardware and software. It consists of a functional definition of operations, modes, and storage locations supported by the hardware, along with a precise definition of how to invoke and access them. Basically, it is a detailed guide for anyone who wishes to write a program that can be executed by this hardware (translated into a functional assembly program).

It does <u>not</u> determine:
- How operations are implemented
- Which operations are fast/slow
- Which operations take more/less power

These characteristics will depend on the hardware design and implementation.

Our Hardware

We are using a microcontroller with a Cortex-M4 processor, which is based on the ARMv7-M architecture.

A Bit About ARM

- ARM is **fabless**:
    - They design the processor, architecture and license it to anyone who wants to use it
    - They do not build it themselves, but instead leave implementation up to the customer
    - This is why we are using NXP semiconductors with ARM cores / architectures!
- Some ARM architectures use the ARM ISA
    - Fixed-length 32-bit instructions
    - Used in applications requiring the highest level of performance

- For many applications of embedded devices, memory efficiency is much more important than the processor speed
    - ARM added support for the Thumb ISA to reduce memory cost in their processors

Thumb ISA

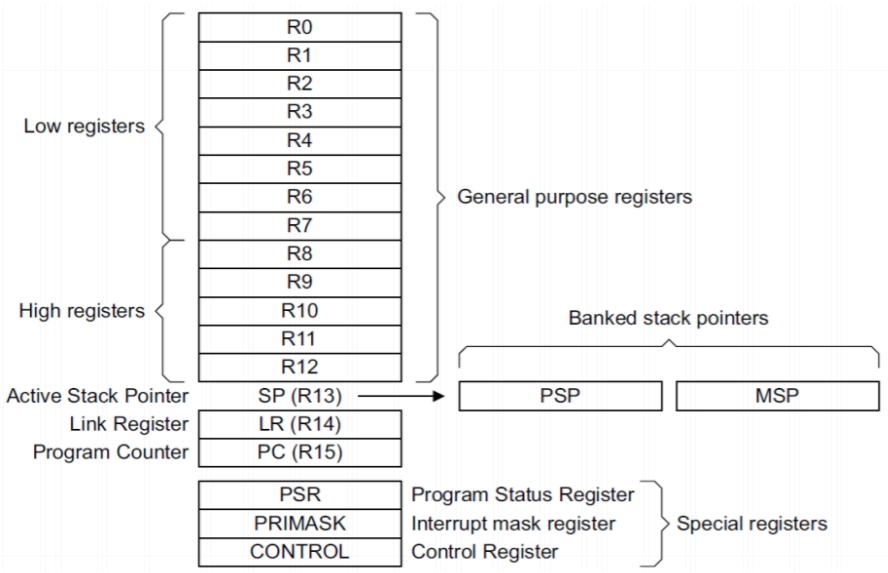
- 16-bit instructions (plus some 32-bit instructions) which are a "shorthand" for a subset of the most commonly used instructions in the 32-bit ARM ISA
    - Every instruction in Thumb has a functionally equivalent 32-bit instruction in ARM instruction set, however not all ARM instructions are available in Thumb instruction set
    - *Reduces the amount of memory used* when we use 16-bit instructions instead of 32-bit!
    - ARM architectures allow for interworking - an execution state bit allows you to switch between Thumb state execution and ARM state execution to maximize performance

NOTE: The ARMv7-M architecture only supports Thumb instructions, so we must always be in Thumb state execution.

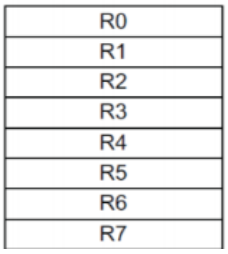
The Cortex-M4 processor has a:

- 32-bit datapath
    - **Datapath** refers to the components of the processor that perform arithmetic operations and hold data (recall datapath + control unit = CPU)
    - 32-bit ALU - takes two 32-bit operands and produces a 32-bit result
    - 32-bit Register File - registers are 32 bits wide

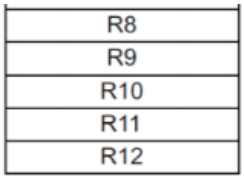
Core Registers for ARM Processor



The General Purpose Registers, **R0**-**R12**, are used to store temporary data within the processor.

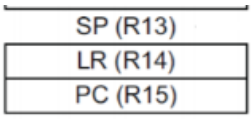


The Low registers, **R0**-**R7**
are accessible by all instructions



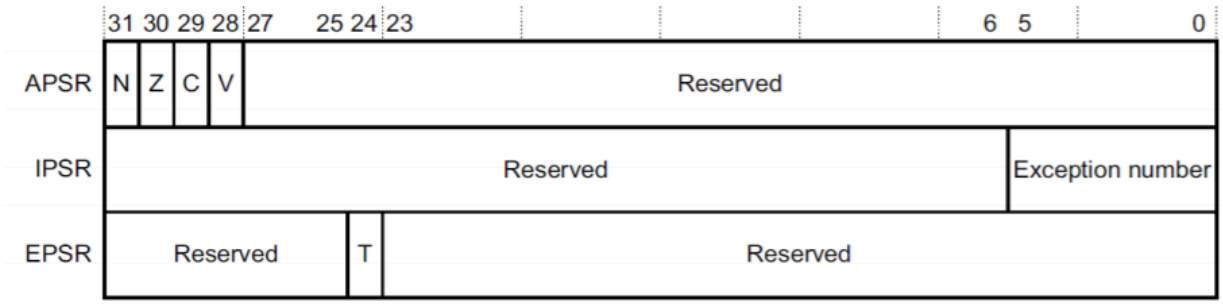
The High Registers, **R8**-**R12**
are accessible by 32-bit instructions,
but are not accessible by most 16-bit instructions

The registers **R13**, **R14**, **R15** are special-purpose registers



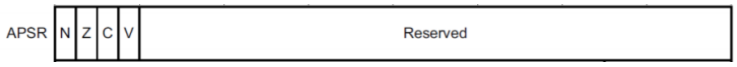
SP (R13) is the Stack Pointer
LR (R14) is the Link Register
PC (R15) is the Program Counter

Program Status Register (PSR)



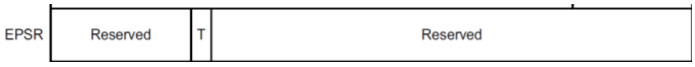
The Program Status Register comprises 3 sub-registers:

- Application PSR (APSR)
  - Flag setting instructions modify the flag bits **N**, **Z**, **C**, **V**
    N=Negative, Z=Zero, C=Carry, V=Overflow
  - Processor uses these flags to evaluate conditional execution in conditional branch instructions
  - Using the "S" suffix on an ALU operation will update the APSR
    - Ex. SUBS updates APSR flag bits, SUB does not
  - CMP, CMN always update the APSR



- Interrupt PSR (IPSR)
  - Processor writes to IPSR on exception entry and exit
  - NOTE: MSR may be a useful function to use here, it moves the contents of a special register to a general purpose register

- Execution PSR (EPSR)
  - Contains the T bit corresponding to the Thumb state
  - Since this particular architecture only supports Thumb instructions, the processor can only execute instruction if T = 1



Thumb Instruction Encoding

- In a complete cycle of instruction execution, a flow of instructions from memory to the CPU is established, called an **instruction stream**
- Thumb instruction stream is a sequence of halfword-aligned halfwords
  - If 16-bit instruction, then it is a single halfword in the stream
  - If 32-bit instruction, then it is two consecutive halfwords in the stream

Memory

The purpose of memory is to allow the computer to store information. The simplest way that we can imagine memory for our processor is a binary sequence.

10110011111110000101000O...

To access and store information, we have to maintain some pointer to our data. We can do this by organizing memory as an array, a contiguous chunk of space divided into blocks of equal size. We can refer to a piece of data in memory using its *address*, which tells us where the data is located in memory. Now, we have created an address space, so our memory looks like this:

Data	1	0	1	1	0	0	1	1	1	1
Address	0	1	2	3	4	5	6	7	8	9

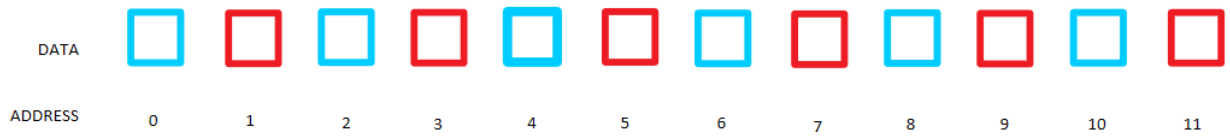
The address space simply provides a number of discrete addresses at which a chunk of data is stored. The question is, how large do we want the chunk of data stored at a particular address to be? In the address space above, each address contains 1 bit. The purpose of memory is to store values that are useful to us. If we are only able to access one bit at a time from memory, it does not really help us when we must perform computations.

The smallest chunk size that is actually useful to us is 8 bits (1 byte). A single byte can represent 256 values. ASCII, a standard for encoding characters for digital communication, assigns 256 total characters (the first 128 are really all that you would need) to binary values. This means that we can store all of the characters necessary for text and punctuation in just 1 byte!

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Since the byte is the smallest chunk of data that can represent meaningful values to us in their entirety, we will make our memory byte-addressable: each address contains 1 byte. Our memory now looks something like this, where each colored block represents 1 byte of data.

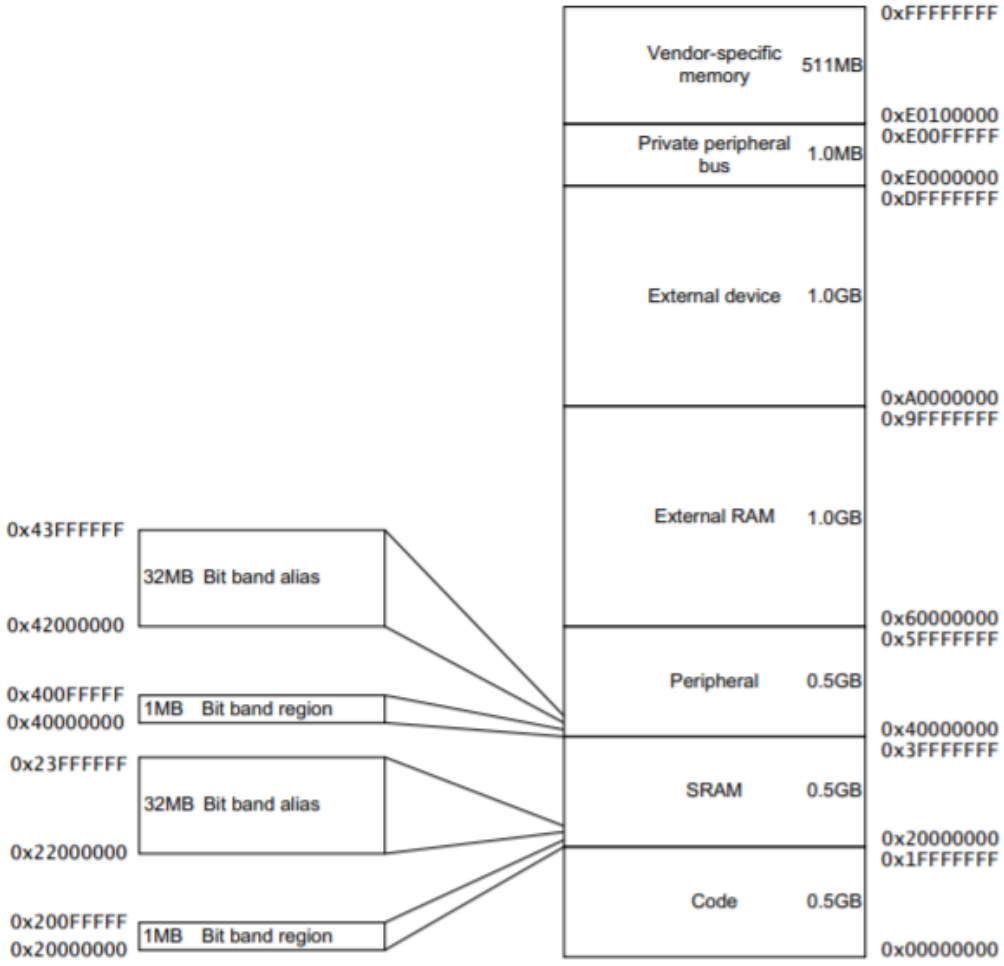


Memory Model for Cortex-M4 Processor

The Cortex-M4 has a 32-bit addressing space. What this *means* is that:

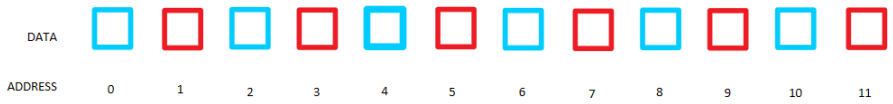
- Addresses in memory are 32 bits wide
- Since an address has 32 bits, and each bit can take the values 0 or 1, there are 2^{32} = 4294967296 possible unique 32-bit addresses
- Memory is *byte-addressable*. Each address points to a location in memory where 1 byte is stored.

- Altogether, if we have 4294967296 possible addresses, and each can refer to 1 byte of data, this means that a 32-bit processor can access at most 4294967296 * 1B = 4GB of RAM

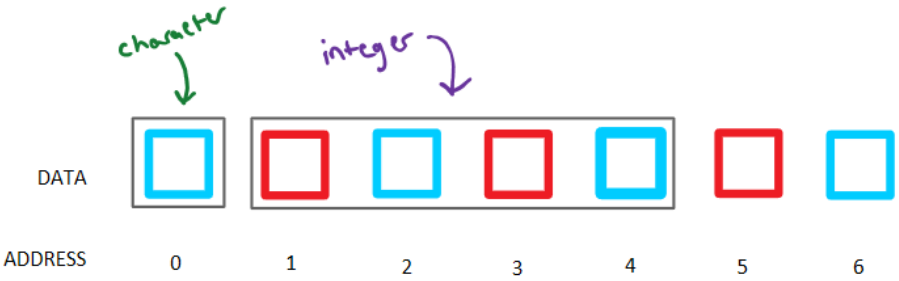


Alignment

We imagined our memory space looking like the picture below, where each address contains 1 byte.



The smallest chunk of data relevant to us is 1 byte, but what is the largest chunk of data we might want to store? We will often want to use integers in our programs, which are 32 bits, or 4 bytes. Suppose we want to store a character and an integer in memory.



Accessing the value of the character is pretty easy - we can just load the value stored at address 0. For the integer, it is a bit more complicated. We would have to load the value stored at address 1, address 2, address 3, and address 4 and combine them together in some way to finally retrieve our integer. This complicates the retrieval of the value and takes a lot of time.

Unfortunately, all of the primitive data types in C other than char are greater than 1 byte in size, so accessing/storing values of these types would require multiple load/store instructions.

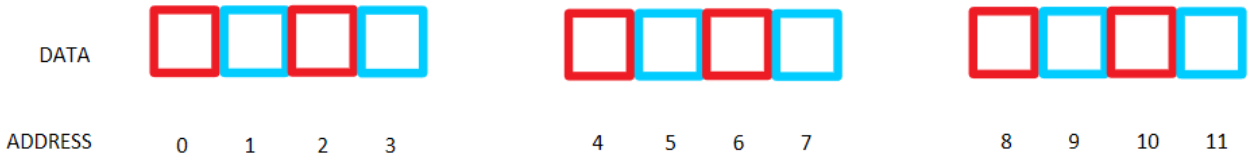
Type	Size (bits)	Description
char	8	ASCII character
short	16	integer value
int	32	integer value
long	32	integer value
float(double)	32(64)	Not available in all –M CPUs

A *word* is the natural unit of data used by a particular processor design. Since our primitive values are all at most 32 bits, or 4 bytes in size, we could choose 4 bytes to be the size of a word. Any single value in one of these primitive types fits in a single word.

The ARM architecture supports the data types:

Byte 8 bits
Half-Word 16 bits
Word 32 bits

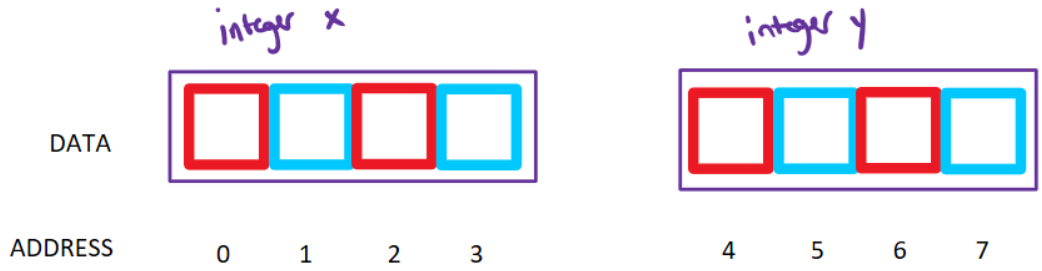
Instead of load/store instructions accessing a single byte in memory, they will access a single word.



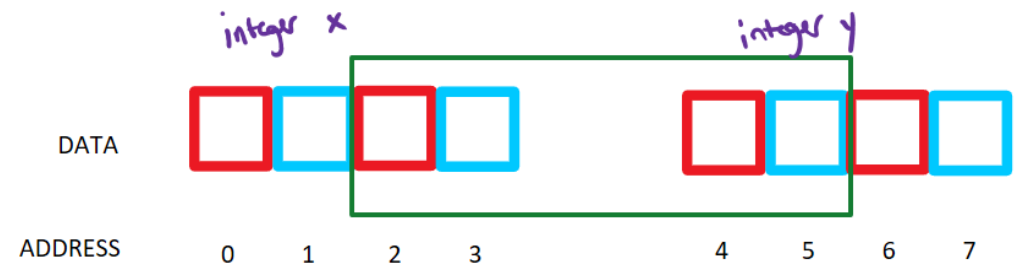
In memory, we will access the *word* at address 0 (through 3), or the word at address 4 (through 7), and so on.

- ARMv7-M architecture uses a single address space of 2^{32} 8-bit bytes.
    - Byte addresses run from 0 -> 2^{32} - 1 (unsigned number rep.)
- 2^{32} 8-bit bytes = 2^{30} 32-bit words
    - Each word's address is *word-aligned*, meaning it is divisible by 4
- 2^{32} 8-bit bytes = 2^{31} 16-bit halfwords
    - Each halfword's address is halfword-aligned, meaning it is divisible by 2
- What is memory alignment??

We know that when the computer reads from or writes to a memory address, it will operate on a word-sized chunk. Memory alignment means that if we are accessing a word, the starting address should be a multiple of the word size. Suppose we had two integers x and y stored in memory.



If we tried to access the value at address 2 (which is not a multiple of the word size 4), we would retrieve the following:



This does not correspond to any meaningful value because it spans two separate words.

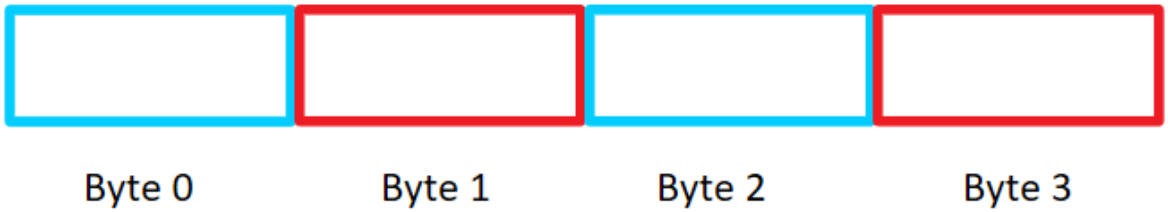
Endianness

How does our computer store values that are bigger than 1 byte in memory?

Let's consider a value that is larger than 1 byte that we wish to store in memory:

0x00abcdef

This is a 32-bit, or 4-byte, hexadecimal number. As we know, memory is byte-addressable: each address refers to 1 byte of data. So, let's consider a chunk of 4 addresses that we can use to store our 4 bytes of data.

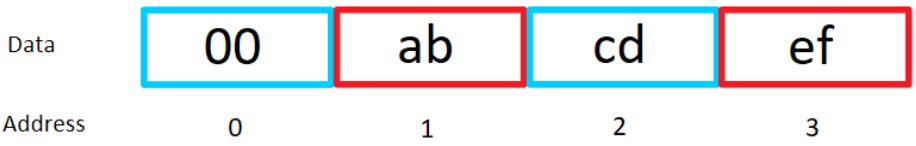


Now, we can break up our data into its separate bytes.

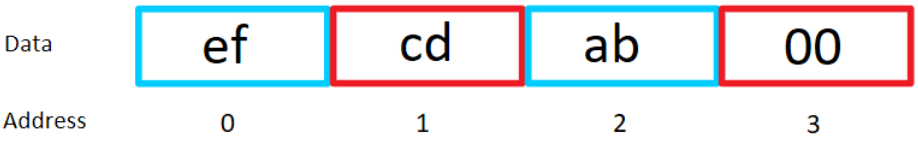
0x00abcdef → 0x 00 ab cd ef

How do we assign the bytes that make up our data to the addresses in memory? There are two different ways:

1)

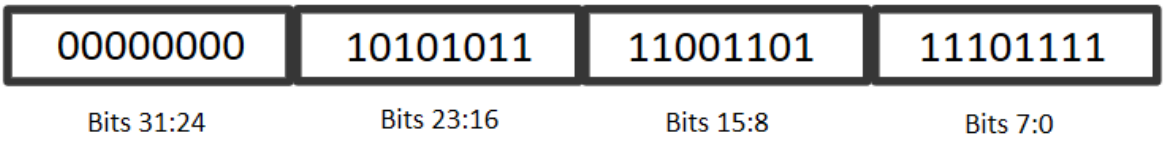


2)



The first seems like the most intuitive way to do this, it places the bytes into addresses "in order."

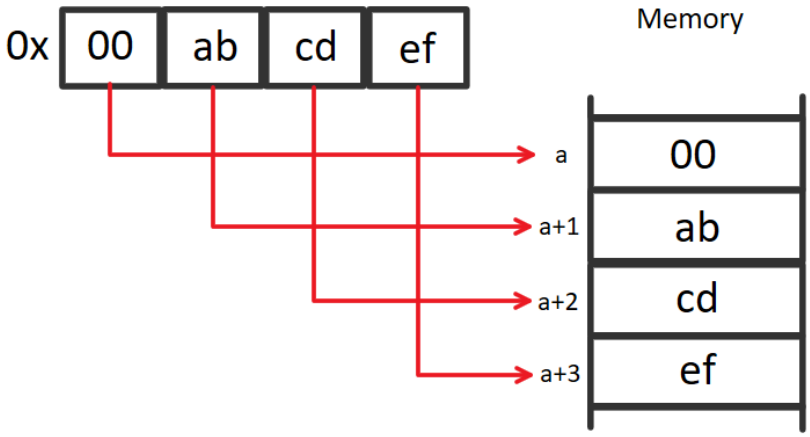
To see why the second way makes sense, let's write our data, 0x00abcdef in binary representation:



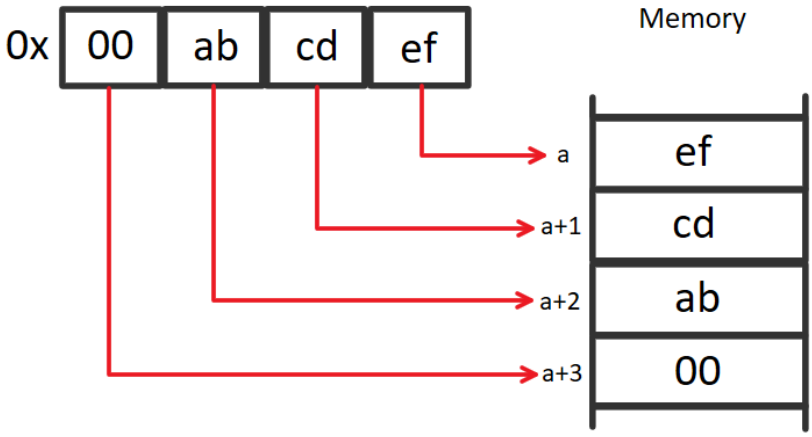
With this representation, we can see very clearly that **[ef]** is the *least significant* byte in our data (in bits 7:0). Method 2) places the least significant byte in the lowest address, and then continues in this order. Byte 0 goes in address 0, byte 1, goes in address 1, byte 2 goes in address 2, byte 3 goes in address 3.

In practice, some computers use the first method of storing multiple bytes of data, called *Big Endian*, and some use the second, called *Little Endian*.

Big Endian - store most significant byte in lowest address



Little Endian - store least significant byte in lowest address



Machines that use different byte-order conventions can still communicate with each other as long as the convention they are using is specified.

The reason why there is no standard byte-order convention is that there are some advantages/disadvantages to both methods:

With little-endian, the way we access a value that is 1-byte, 2-bytes, 4-bytes (or more) is consistent. We begin by selecting the least significant byte, at offset 0, and continue incrementing the offset as necessary. We will be able to see this clearly in our assembly instructions.

With big-endian, it is easier for humans to read/understand when examining memory values, especially for debugging.

The Cortex-M4 uses *little-endian*!

Instruction Format

An instruction consists of two parts:

- Operation Code - specifies the operation to be performed
- Operand(s) - specify the data it will be performed on

Consider a 32-bit instruction. Since there are only 32 bits to allocate in the entire instruction, a longer OP code means there are fewer bits dedicated to Operand(s), and vice versa.

If we have a greater Opcode size, then we can have a larger number of operations available to us - a larger instruction set. But then, we must have a smaller Operand size, so fewer memory locations can be accessed directly. To make efficient use of the bits in an instruction, we have different *addressing modes*.

Addressing Modes

There are different ways in which you can specify the address of an operand in an instruction. These different ways are called addressing modes. They allow us to strategically choose how to provide the operand(s) of an instruction so that we can fit all instructions in 32 bits.

Three addressing modes in ARMv7-M architecture:

1) Immediate Addressing
    - ❏ In this mode, the operand is an immediate value. In the instruction, we use this immediate value as the operand.
2) Register
    - ❏ In this mode, the operand is stored in a register on the CPU. In the instruction, we use the address of this register to specify the operand.
3) Scaled Register

For memory access operations, there are different methods of calculating the *effective address*, the one we use to access memory.

An address has two components:

- A base register, which could be any of the general-purpose registers
- An offset

Load and Store operations have 3 primary addressing modes:

1) Offset
   The offset is added to or subtracted from the base register to form the effective address.
2) Pre-indexed
   The offset is added to or subtracted from the base register to form the effective address. The base register is then updated to store this new address.
3) Post-indexed
   The value of the base register is used as the effective address. The offset is then added to or subtracted from the base register, and this new address is stored in the base register.

The advantage of having fewer addressing modes is a simpler ISA and implementation.

Load/Store

ARM is a *Load/Store* architecture. This means that instructions are divided into two categories:

1) Memory accesses
    - A Load operation moves data from memory into a register.
    - A Store operation moves data from a register into memory.

2) ALU operations
    - Arithmetic/logic operations are performed on data stored only in registers

There are a mix of register-memory, memory-register, and register-register instructions in the instruction set. In a load/store architecture, memory accesses (memory-register and register-memory) are factored out into just two instructions. In our case, these are LDR and STR. We only need different addressing modes for these two instructions. All other instructions are register-register, so they can use the Register addressing mode.

IMPLEMENTING C LANGUAGE CONSTRUCTS

Subroutines

A C program consists of one or more functions. A function may take arguments (parameters), include a series of statements, and may return a result.

Subroutines are important for *modularity*. The idea of modularity is that we can split up our program into smaller parts which can be considered in isolation. For example, in Lab1, we must implement the function morseDigit(int n). This is a great example of the benefits of modularity in practice:

- Smaller parts can be worked on separately.
    - When approaching this task, we naturally divide it into smaller tasks so that we can work on it one piece at a time. One way to begin working on this program is by creating a series of conditional branches to check if n=0, n=1, n=2, and so on. In pseudocode, something like:

```
morseDigit

checkIf0

            if 0, BL ZERO
            else, go to checkIf1

checkIf1
```

    - For this part, we can assume we have some function ZERO that signals a zero in morse code; we do not need to have ZERO already implemented. This way, our partner could work on the ZERO function separately.
- Understand a part of the code in isolation.

    - In order to combine our conditional branches with our partner's ZERO function, we do not need to know how ZERO was implemented. We can work under the assumption that ZERO will signal a zero in morse code. Our partner does not need to understand how the value of n is checked in order to create the ZERO function. They must simply understand the ZERO function in isolation.
- Reuse code to make debugging easier.
    - To signal different numbers in morse code, the LED must signal different combinations of dots and dashes. We may have functions for EIGHT and NINE like this:

```
EIGHT
                BL      DASH
                BL      DELAY
                BL      DASH
                BL      DELAY
                BL      DASH
                BL      DELAY
                BL      DOT
                BL      DELAY
                BL      DOT
NINE
                BL      DASH
                BL      DELAY
                BL      DASH
                BL      DELAY
                BL      DASH
                BL      DELAY
                BL      DASH
                BL      DELAY
                BL      DOT
```

    - This leads us to create the functions DOT and DASH, which turn the LED on for different amounts of time. Of course, we would never want to write out all of the code within the DOT function every time we wanted the LED to signal a DOT, since we know this is a function we will be reusing over and over again. If DOT has a bug, we can fix it within the function body for DOT, and this will fix the bug everywhere we have called it. Obviously, if we wrote out the code for DOT every time instead, we would have to go through and fix the bug in each instance.

A subroutine must support:

- Calls from multiple program locations
- Nested and recursive calls

ARM Calling Convention

There are two participants in a function call - the caller and the callee. If routine A calls the routine B, then routine A is the caller and routine B is the callee. In practice, this looks something like:

```
__main
        MOV     R0, #3
        BL      Fibonacci
```

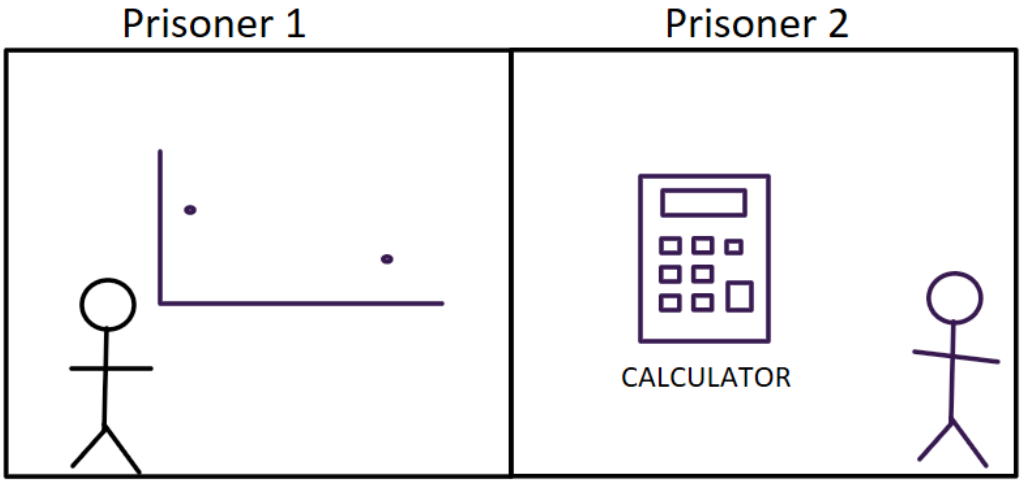
```
Fibonacci
                CMP     R0, #0
                BLE     eqzero
                CMP     R0, #1
                BEQ     eqone
                B       recurrence

eqzero          MOV     R0, #0
                BX      LR

eqone           BX      LR
```

Here, __main is the caller of the function Fibonacci. On the right, Fibonacci is the callee.

To maintain modularity, the caller and callee are separate routines. Still, they must be interfaced in some way to work together. *Calling convention* defines a contract between the caller and the callee.

Registers

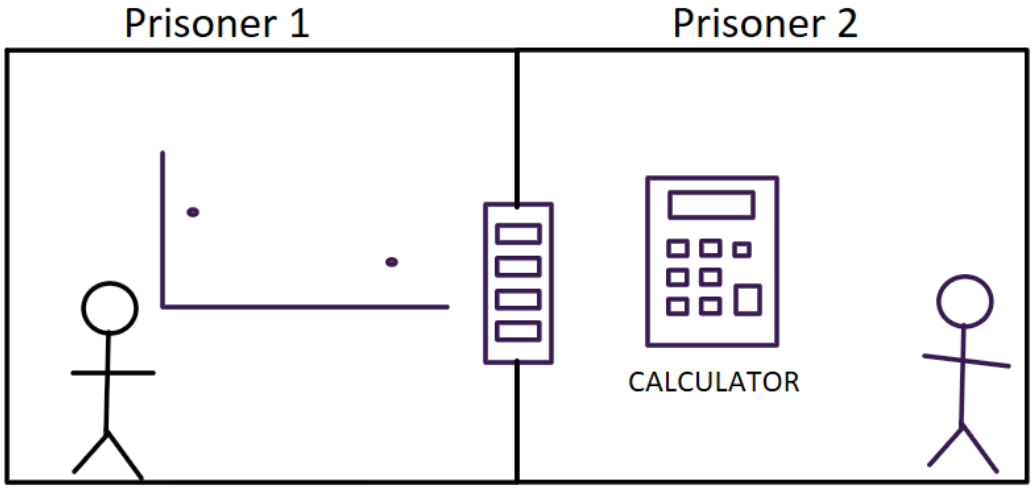
Two prisoners are offered a chance to earn their freedom if they can complete this challenge. They will be placed in two adjacent rooms but must use teamwork to solve a problem.



Their captor will release them if Prisoner 1 can tell him the distance between 2 points drawn on a graph on the wall in Prisoner 1's room. Prisoner 1 sees the two points plotted on the graph so he knows their exact coordinates, but he doesn't know basic math! He has no idea how to compute the distance between the points. Prisoner 2, on the other hand, has a calculator in his room, and a slip of paper with a formula for the distance between two points. The paper says that, for two points, (x_1,y_1) and (x_2,y_2), the distance between them is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

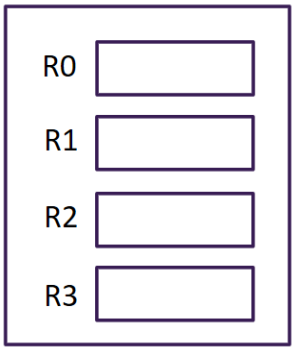
Unfortunately, the coordinates of the two points are only visible in the other room! The prisoners will not be able to communicate verbally, but they know that there is a small slot in the wall between the two rooms. In both rooms, the prisoner will be able to open this slot to reveal a small tray between them.



Additionally, in each room, the prisoner has access to a box with blocks numbered 0-9. We will assume that there are unlimited blocks, so the prisoners can stick blocks together to form any number.

The prisoners have full knowledge of this setup, and are given 5 minutes to create a plan before they are placed in their respective rooms. They come up with this plan:

> First, the prisoners decide on a standard way of numbering the trays for consistency. From top to bottom, the trays will be R0, R1, R2, R3.



Prisoner 1 will select one point, (x_1,y_1). He will stick numbered blocks together to form a block with the value x_1, and place it into R0. Then, he will use the blocks to form y_1 and place it into R1. Following this procedure, for the next point on the graph, (x_2,y_2), he will place the x_2 block into R2 and the y_2 block into R3.

Prisoner 2 will then open the slot to look at the tray. He will plug the value in R0 in for x_1, R1 in for y_1, R2 in for x_2, and R3 in for y_2 in his distance equation, and use the calculator to solve it.

Then, Prisoner 2 will stick together blocks to form the value of the result his calculator gives, and he will place it in R0.

Finally, Prisoner 1 will open the slot and retrieve the block in R0, telling the captor its value.

This game is similar to how subroutine calls work in accordance with the calling convention. Prisoner 1, like the caller, has the responsibility of supplying the correct arguments, the two points on the graph. He assumes that Prisoner 2 will hold up his end of the deal, that is, given the right arguments, Prisoner 2 will correctly compute the distance between them. Then, it is up to Prisoner 1 to determine how to use this result Prisoner 2 computes. In this case, it's Prisoner 1's responsibility to give this value to the captor as his solution.

The caller and callee are expected to hold up their end of the contract in isolation, but they must be able to pass information between them: the arguments and the return value. The calling convention is a scheme for how subroutines receive parameters from their caller and return a result.

Note that there is no physical difference between the registers R0-R3 and R4-R11. The calling convention is just a *convention*, a somewhat arbitrary agreement that allows the caller and callee to work together because they can agree on it - just like how the two prisoners decided to label/use the tray.

We use R0-R3 to pass the arguments to the callee because these registers act as the shared storage space between different subroutines. They are sort of like the *placenta* of registers, allowing values to pass through from one distinct subroutine to another. If the caller places the arguments in R0-R3, they are accessible to the callee. When the callee returns a value, it is in R0, so that it is accessible to the caller.

In contrast, registers R4-R11 are not shared between subroutines. Suppose Prisoner 2 has to perform some scratch work to determine the distance between the points (1,2) and (3,5). Perhaps he computes $(1-3)^2$ to be 4 on his calculator and decides to store

this value. Then, he can use his calculator to compute $(2-5)^2$, and it evaluates to 9. He adds 9 + 4 to get 13, and finally takes the square root of 13. These intermediary values like 4, 9, and 13 are like local variables in a subroutine. They are stored in R4-R11 because they are needed within this subroutine, but are not needed by other subroutines. Prisoner 1 does not need to know any of these intermediary values, he only cares about the final result.

Suppose, additionally, that the captor tells Prisoner 1 some word "x." For the solution, he must repeat the word "x" aloud a number of times equal to the distance between the two points. Prisoner 2 does not need to know the word "x," but Prisoner 1 needs to be able to store it somewhere because he will need it later. In fact, he wouldn't want to store "x" in the tray both prisoners have access to because then Prisoner 2 might replace it and he'd lose it forever. So, it may be important for our prisoners to have not only a shared storage space, but also their own private storage locations that are unaffected by the other prisoner.

This brings us to the idea of callee-saved and caller-saved registers.

Caller-Saved Registers

- R0-R3
- The caller must save the values in registers R0-R3 if it wants to be able to access them after the subroutine call completes.
    - R0-R3 is a shared space, so the callee may change the values in these registers.
    - Suppose the caller needs to value n + f(n). The caller first moves the value n into R0 so it is passed as the argument to f. But then, f moves the value of f(n) into R0 as the return value, overwriting the value n. If the caller needs the value of n, it must save it before calling f!

Callee-Saved Registers

- R4-R11
- If the callee needs to use any of the registers R4-R11, it must first save the values stored in them. Then, after its completion, it must return registers R4-R11 to their original values.
    - Remember, R4-R11 are like the private storage space of a subroutine. The caller has its own set of values in these registers, and it expects these values to be unchanged after the subroutine call completes.

How do we save registers?

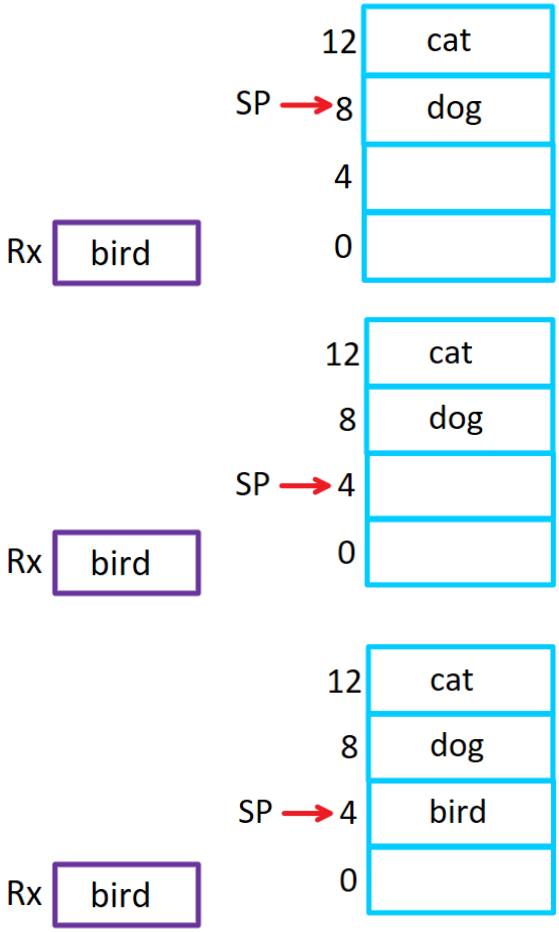
The Stack

We use R0-R3 to pass arguments to the callee. But what if there are more than 4 arguments? These can be passed to the callee via another space accessible by all subroutines - the stack. A *stack* is a data structure that is typically used to store temporary data for functions.

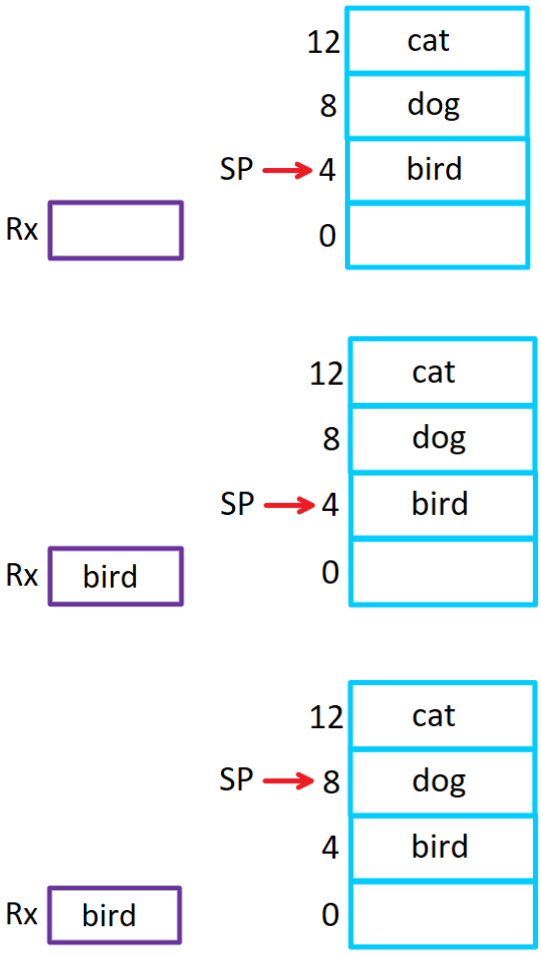
Specifically, we use a full descending stack, so the stack grows downward, toward lower addresses, as we add values to it. The stack pointer, SP, stores the memory address of the stack's last (most recent) element.

It has the operations:

- PUSH Rx
    - Decrement SP to create a new "top" of the stack
    - Store the value Rx where SP is pointing



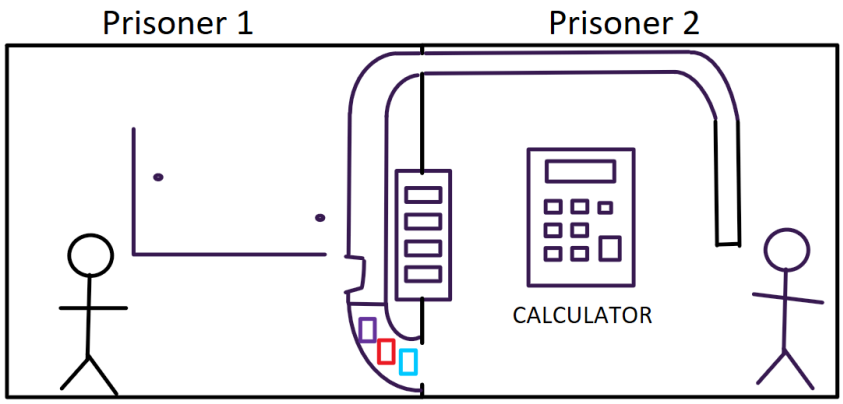
- POP Rx
    - Stores the "top" of the stack in Rx
    - Increments SP to remove "top" from stack



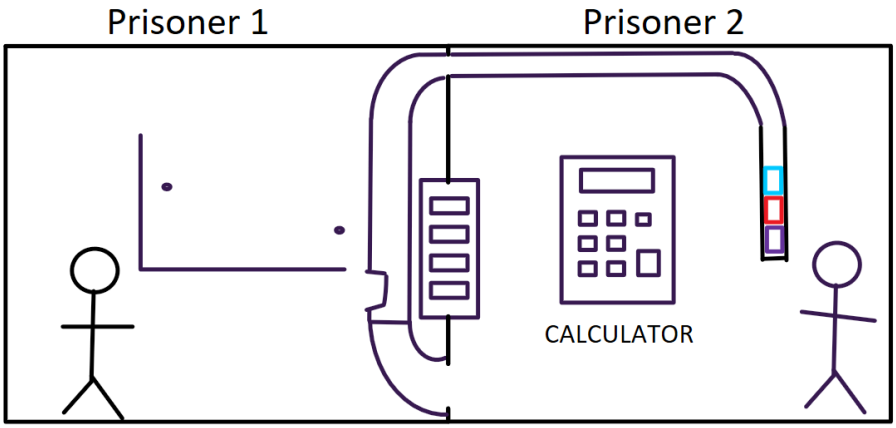
Note also that the stack is word-aligned (4 bytes). This means that addresses on the stack are always incremented (or decremented) from the SP by a multiple of 4.

Returning to our prisoner example once more, suppose Prisoner 1 is given two 3-D coordinates, (x_1, y_1, z_1) and (x_2, y_2, z_2). He must determine the distance between these points.

For this more challenging problem, the captor has modified the room to include a tube. Prisoner 1 can drop blocks into the tube and they will fall to the bottom and maintain their order. For example, Prisoner 1 may add the turquoise block, then the red block, and finally the purple block.



When Prisoner 1 is done adding blocks, he presses a button and compressed air shoots the blocks up through the tube and over into Prisoner 2's room.



The prisoners adjust their plan slightly to accommodate the new challenge. Prisoner 1 decides to place x_1, y_1, z_1, and x_2 in the tray in R0, R1, R2, and R3 respectively. Then, he decides to place y_2 and z_2 in the tube, and presses the button to send them over.

Prisoner 2 receives the first 4 arguments, x_1, y_1, z_1, and x_2 in order via the tray. Next, he needs y_2, so he grabs the first block from the tube. However, there's a problem! Prisoner 1 placed y_2 then z_2 in the tube, so the first block Prisoner 2 grabs is z_2!

To make things easier, we can have Prisoner 1 pass in the remaining 2 arguments in reverse order - z_2 then y_2. This way, Prisoner 2 can retrieve all 6 arguments in order.

If there are more than 4 arguments, the caller must push the remaining arguments to the stack in reverse order. Then, the callee can POP the current value on the stack into R4, then POP the next into R5, and so on, in order.

Nested Subroutine Calls

The other issue accounted for by the calling convention is how a value is returned and made available to the caller. Let's examine the example from class:

```
;caller
        ...
        BL func ;call func
ra1
;func(a,b,c) : returns a+b*c
func
    MUL R4,R1,R2 ;b*c
    MOV R1,R4    ;R0=a,R1=b*c
    BL sum       ;call sum
ra2
    BX LR        ;return

;sum(m,n) : returns (m+n)
sum
    ADD R0,R0,R1 ;m+n
    BX LR
```

Recall that a branch instruction causes the processor to jump to a different line of the program and begin executing instructions in order from that point. The branch instruction is perfect for a function call - it causes the processor to jump to the function body and execute the statements within it. But why not use **B func**?

The issue here is that, once the function call has completed, we need the processor to jump back to the next line in the program (the line after the function call). We can accomplish this using the Link Register.

In the caller, if we are currently at the BL func instruction, then the PC contains the address of this instruction. After the call completes, we want our program to execute the line in the program following BL func, at address PC+4. This is the next value of the program counter. If we store this address, PC+4, in the link register, then at the end of the function call we can branch to this address.

The caller calls the function *func* using the instruction **BL func**. The BL instruction performs a branch-with-link operation. It transfers the sequentially next value of the program counter - the return address - to the link register (LR)* and the destination address into the program counter (PC).

*Bit 0 of the link register will be set to 1 if the BL instruction was executed from Thumb state, 0 if executed from ARM state.

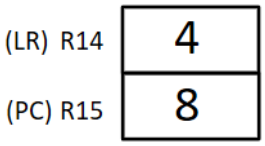
To see how this works, I've added some line numbers in green (counting by 4) to the example from class. We can walk through the execution of this program and keep track of the values in the LR and PC.

```
        ;caller
              ...
→  0          BL func ;call func
   4    ra1
     ;func(a,b,c) : returns a+b*c
   8 func
  12      MUL R4,R1,R2 ;b*c
  16      MOV R1,R4    ;R0=a,R1=b*c
  20      BL sum       ;call sum
  24 ra2
  28      BX LR        ;return

     ;sum(m,n) : returns (m+n)
  32 sum
  36      ADD R0,R0,R1 ;m+n
  40      BX LR
```

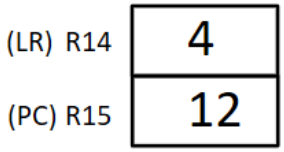
(LR) R14	4
(PC) R15	0

Since this is a function call, we want to be able to branch to line 4 after func completes, so we store 4 in the LR.

```
    ;caller
          ...
 0        BL func ;call func
 4   ra1
  ;func(a,b,c) : returns a+b*c
 8 func
12     MUL R4,R1,R2 ;b*c
16     MOV R1,R4    ;R0=a,R1=b*c
20     BL sum       ;call sum
24 ra2
28     BX LR        ;return

  ;sum(m,n) : returns (m+n)
32 sum
36     ADD R0,R0,R1 ;m+n
40     BX LR
```




```
    ;caller
          ...
 0        BL func ;call func
 4   ra1
  ;func(a,b,c) : returns a+b*c
 8 func
12     MUL R4,R1,R2 ;b*c
16     MOV R1,R4    ;R0=a,R1=b*c
20     BL sum       ;call sum
24 ra2
28     BX LR        ;return

  ;sum(m,n) : returns (m+n)
32 sum
36     ADD R0,R0,R1 ;m+n
40     BX LR
```



```
     ;caller
           ...
 0         BL func ;call func
 4   ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12     MUL R4,R1,R2 ;b*c
16     MOV R1,R4    ;R0=a,R1=b*c
20     BL sum       ;call sum
24 ra2
28     BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36     ADD R0,R0,R1 ;m+n
40     BX LR
```

(LR) R14	4
(PC) R15	16

```
     ;caller
           ...
 0         BL func ;call func
 4   ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12     MUL R4,R1,R2 ;b*c
16     MOV R1,R4    ;R0=a,R1=b*c
20     BL sum       ;call sum
24 ra2
28     BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36     ADD R0,R0,R1 ;m+n
40     BX LR
```

(LR) R14	24
(PC) R15	20

This is a function call, so we want to be able to branch to line 24 after sum completes. When we store 24 in the LR, it overwrites the previous value of 4.

```
    ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
24 ra2
28      BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
40      BX LR
```

(LR) R14 | 24

(PC) R15 | 32

```
    ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
24 ra2
28      BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
40      BX LR
```

(LR) R14 | 24

(PC) R15 | 36

```
    ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
24 ra2
28      BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
→40     BX LR
```

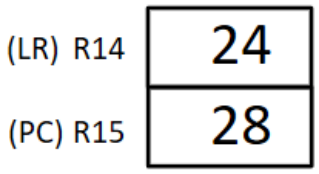
(LR) R14	24
(PC) R15	40

The call to sum completes, and the instruction **BX LR** allows us to correctly branch back to the line following **BL sum**, line 24.

```
    ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
→24 ra2
28      BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
40      BX LR
```

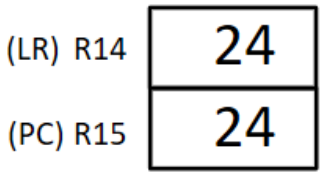
(LR) R14	24
(PC) R15	24

```
     ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
24 ra2
→28     BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
40      BX LR
```



At this point, we have completed the call to func, so we want to branch back to the line following **BL func**, line 4. But 4 is no longer stored in LR and there is no way to retrieve it! **BX LR** brings us right back to line 24.

```
     ;caller
            ...
 0          BL func ;call func
 4    ra1
   ;func(a,b,c) : returns a+b*c
 8 func
12      MUL R4,R1,R2 ;b*c
16      MOV R1,R4    ;R0=a,R1=b*c
20      BL sum       ;call sum
→24 ra2
28      BX LR        ;return

   ;sum(m,n) : returns (m+n)
32 sum
36      ADD R0,R0,R1 ;m+n
40      BX LR
```



The program continues looping in this fashion, unable to exit the function call.

We can solve this problem using the stack! Within the call to func, we can push the address currently stored in LR onto the stack. When the call to func completes, we will want to branch to this address, so we just need to be able to retrieve it at the end of the call.

When we reach the instruction **BL sum** within func, it will overwrite the value in the LR, to the line following **BL sum**. When the call to sum completes, the program will be able to correctly branch to the line following **BL sum**.

At this point, we have our return value and we are ready to complete the call to func. We can POP the address we stored on the stack earlier into the LR. Then, **BX LR** will bring us to the line following **BL func**!

```
; Function Call

      …
      BL func ; call func
      …

func
      PUSH LR
      PUSH R4

      MUL R4,R1,R2 ; b*c

      ; call sum(a,b*c)
      MOV R1,R4
      BL sum

      POP R4
      POP LR
      BX LR
```

Above, you can see that after pushing the value in LR onto the stack, we push R4. The point of showing this is that we push LR onto the stack *first*, and pop it *last*. The callee may need to push any of the registers R4-R11 on the stack if it needs to use them, as these are callee-saved registers. And any subroutines called within this function call may also push values onto the stack. No matter what is pushed onto the stack during this function call, it must be popped off the stack before the end of the call. We can see in this example that we POP R4 before we POP LR. At the very end, when we are ready to return and complete the function call, we need the SP to be pointing at the address we want to branch to, so that we can POP it into the LR.

LR is a caller-saved register, because we typically must push it onto the stack before we call any subroutine. Calling a nested subroutine requires the **BL** instruction because we must save the address it should return to, which overwrites the LR. We

do not need to save LR on the stack in main because when we are executing instructions in main, there is no address stored in LR, since main has no caller.

Local Variables

In many cases, a function may define some local variables within its body. Consider the following example.

```
int factorial (int n) {
      int i;
      int result = 1;

      for (i = n; i > 0; i--) {
            result *= i;
      }

      return result;
}
```

We have good reason to define the local variable result. In each iteration of the for-loop, we need to be able to access the current value of the result in order to update it. We need to be able to keep track of the progress we have made so far, because the result of each iteration builds off of the one before it.

In general, we use variables when we need to store a value, allowing it to be accessed later. Local variables are used for values we want to have stored while we are executing the function we declare them in, but do not need outside of this function. If a homeowner hires a plumber to fix a leaky pipe, the plumber may bring over his own tool box that he uses to complete the job. The homeowner does not need to use any of the tools himself, but the plumber couldn't complete the job without them. The homeowner just cares about the end result - getting that pipe fixed - so after that's done, he doesn't really care how the plumber did it or what tools he used.

The calling convention defines how we handle local variables. When a function that declares local variables is called, it allocates space for them on the stack (after pushing callee-saved registers).

```
int function(<parameters>) {
    int x, y, z;
    ...
    return result;
}
```

```
function
        PUSH {R4-R11}
        SUB SP, #12
        ;...
        ADD SP, #12
        POP {R4-R11}
```

You may wonder why we update the stack pointer instead of simply pushing our local variables onto the stack as needed, and then popping them off at the end, like we do with caller and callee saved registers.

The reason we allocate space on the stack is because we need to be able to access and update the values stored in these local variables within the function. With caller and callee saved registers, we only need to retrieve their values after the function call has completed, so that they are preserved. Recall the following instruction:

LDR Rx,[SP,#v]

Unlike POP, this instruction is a non-destructive way to access a value stored on the stack. We use local variables because we want to have them stored during the execution of the function, so that we can access their values later.

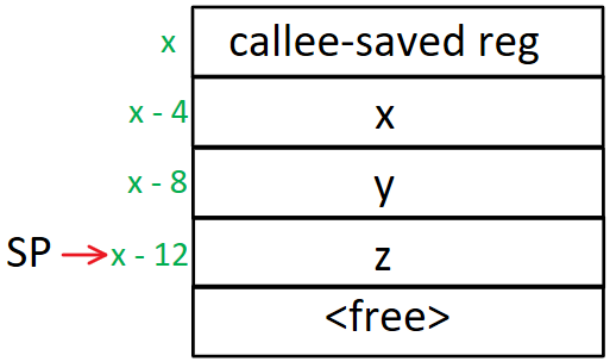
```
int function(int arr[], int n) {
    int v;
    ;....
    v = average(arr[], n);
    for (int i = 0; i < n; i++) {
            if (arr[i] == v)
                return i;
    }
    return -1;
}
```

In this function, for example, we want to return the index of the first element in arr[] of length n which has a value equal to the average value of the elements in arr[]. We store this average value in a local variable v because we need to compare each element in arr[] to it, one by one. We do not want to call the helper function each time since the value isn't going to change, so we store it as a local variable to enable

us to access it repeatedly throughout the function. If we were to pop the value of v off the stack the first time we accessed it, we wouldn't be able to access it again.

The LDR instruction allows us to access a local variable's value without removing it from the stack, as long as we know where it is located relative to SP. Consider what would happen if we stored local variables on the stack any way we wanted. It could get confusing if we tried to keep track of all of the local variables' locations on the stack. Suppose we had recorded each local variable's offset from the current SP. If we updated the SP, we would then have to update every single offset accordingly. Following the convention makes this much simpler.

Suppose that SP currently points at some address x. We decrement SP by 12 because our function declares 3 local variables which are 4B each since they are integers. Note that we allocate space on the stack for these variables even though they have not even been given values yet.



In general, we will subtract the number of bytes required by each local variable declared from SP in order to allocate space on the stack for all local variables. In doing this, SP points to the last local variable declared in our function, in this case z. We can access the value of z, the third variable declared, using LDR Rx, [SP]. Since we know y was the second variable declared, we can access the value of y using the instruction LDR Rx, [SP, #4]. From the third variable, we go back by 1*4B to find the second variable. Lastly, we can access x because we know it was the first variable declared, using the instruction LDR Rx, [SP, #8]. From the third variable, we go back by 2*4B to find the first variable.

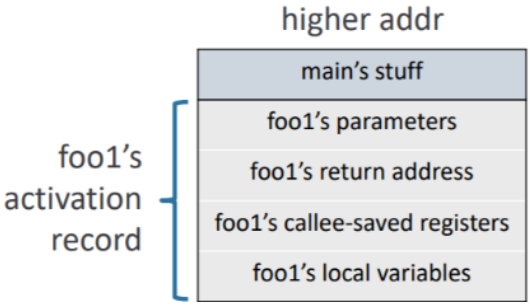
Instead of keeping track of the locations of each individual local variable on the stack, we only have to keep track of one. We establish a singular location at which the last local variable's value is stored. Then, all local variables can be found at addresses relative to this location, based on the order in which they were declared.

Activation Record

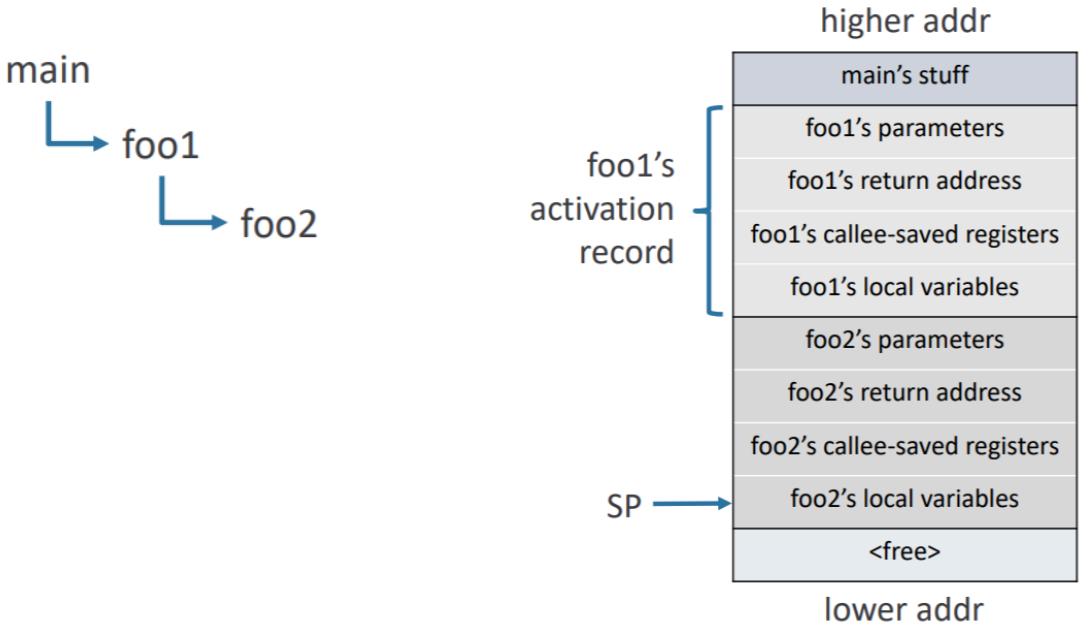
When a subroutine calls another subroutine, the stack may change while the callee is being executed. With nested function calls, the stack may grow incredibly large, extending far from where the SP pointed in the original caller. The *activation record* is the portion of the stack relevant to a particular subroutine during execution.

We have discussed this idea of a “shared” space and a “private” space for different subroutines. The stack is one space that is shared among all subroutines. The activation record shows us how every subroutine is able to use the same stack to store and access the values relevant to them.

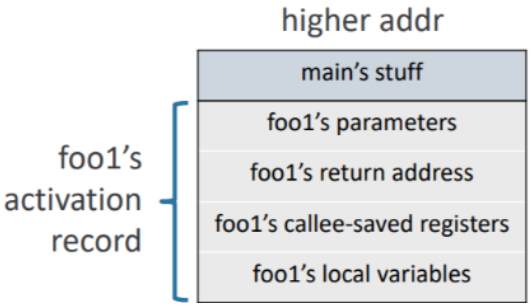
Consider a function foo1 that is currently executing. The SP points to its last local variable.



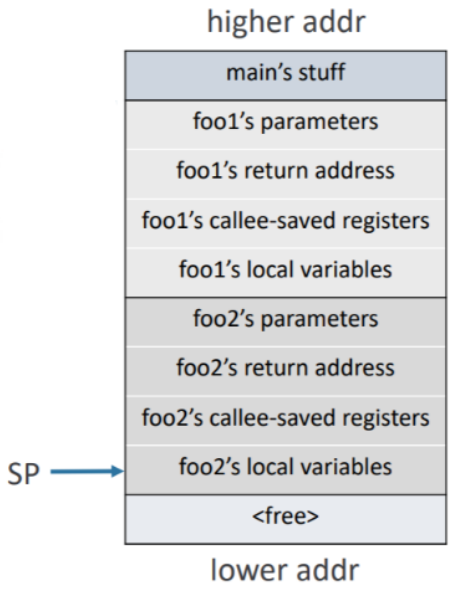
foo1 calls another function, foo2, causing the stack to grow.



Now, the stack has changed, “losing” foo1’s place by moving the SP. When foo2 returns, we need to ensure that the SP is restored to where it was before the function call. foo1 may continue and need to be able to access its local variables. It expects the stack to be unchanged by foo2:



Thanks to the calling convention, it is very easy to restore the stack. Here’s how it works for foo2. If we have followed the calling convention described throughout this section, this is how the stack will look before foo2 has begun its computations.



INPUT/OUTPUT

I/O is the communication between an information processing system and the outside world. Up until now we have focused on how a computer works as an isolated system, but computers are so useful to us because they can interact with the “outside world.” The human brain is quite impressive, but computers have a strong

advantage when it comes to information processing. So, we let computers handle information processing for us. With I/O, we are able to input data for the computer to process, and output the results of these computations.

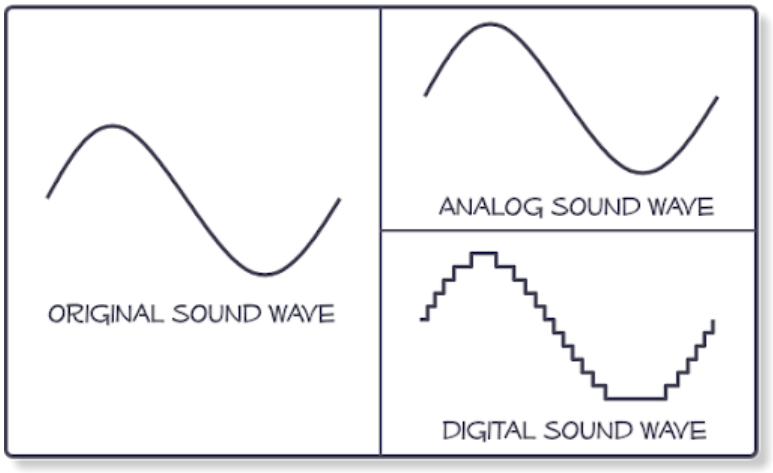
Input devices like a mouse and keyboard allow people to control computers in an intuitive, fundamentally human way. Conversely, output devices like monitors turn the machine code being executed by the processor into something graphical that is much more palatable for humans. I/O devices act as a sort of translator; humans and computers "speak" vastly different languages, and I/O devices allow us to each work in our own language while still being able to communicate with one another. It's easy for most people to use computers every day without having a clue how they work, because I/O allows for *abstraction*. The average user does not need to know how a computer functions in order to make use of it.

There are some major differences in these two systems that are interfaced. First, consider how data is represented in each system.

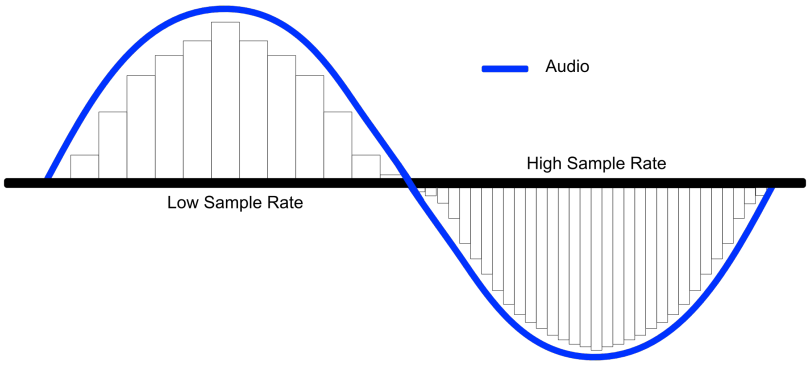
A signal is a way of conveying information. It's a function of time, space, energy - any observable quality which conveys information. For example, sound travels in waves and it can be represented as a signal. An analog signal is a signal which varies with time in some quality, analogously to the variation in the real world phenomenon. In the case of some sound, an analog audio signal (representing some real-world sound) varies in voltage exactly analogous to how the pressure of the sound waves vary. Electrical signals are the representation that is most relevant to our purposes. The current, voltage, or frequency of an electrical signal can be varied to represent the real-world information.

Real world qualities are usually real-valued. Analog signals can be continuous, with theoretically infinite precision, varying exactly as we observe the data to vary in nature. (Analog signals aren't always continuous, though. By definition, they are just true to nature. If the natural event is discrete, then an analog signal representing it would be discrete as well).

However, in a computer, information processing is greatly complicated by continuous, smooth signals. Electrical circuits have a lot of noise, which is captured by analog signals with incredible precision. This can skew the data we want to represent. If the smallest variations are rendered, how can we tell what values are important in computations? Additionally, in order to quantify the data expressed in analog signals, due to the infinite precision, it would require a corresponding infinite number of digits. Computers use *digital* technology, working on discrete-values signals that approximate the real-world data. These digital signals describe real-world phenomena with a series of discrete values, which can be processed because they are now finite.

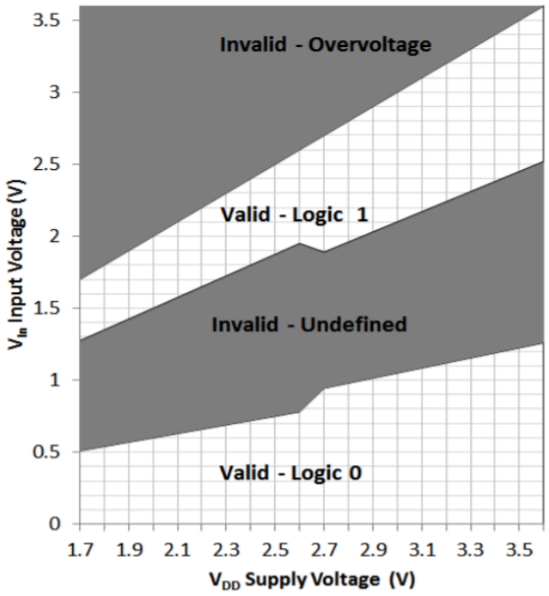


Digital signals are an approximation of the real-valued data. This is why, for example, we can download a high quality version of a song that takes up more space than a lower quality version. We convert an analog signal to digital by repeatedly taking samples of it at some fixed intervals of time. When we have a high sample rate, the digital signal more closely approximates the real-world analog signal, so we observe it to be higher quality. Analog signals are more suited to humans because we evolved to perceive natural phenomena and analog signals mimic nature. On the other hand, digital signals are much more compatible with computers because they are easier to process.



In our processor, we work with binary values. This means we must convert analog signals to digital signals that can take on only the values 0 and 1. To accomplish this, we will call any voltage above some threshold a 1, and any voltage below some threshold a 0. If we were to choose one value as this cutoff point for either direction, assigning 0s and 1s to intermediate values close to this threshold would not be consistent. To get around this, we will say that values in some intermediate range are

undefined. Values above and below this range are valid and can be treated as 0s and 1s, as shown in the diagram below. This approach is robust against noise.



I/O Hardware

The hardware components in the processor that are used for I/O are called *pins*. Pins allow the processor to electrically connect with external components. We refer to a group of pins as a *port*. Microcontrollers typically have many pins on them that each have some function(s). Some of these pins are General Purpose Input/Output pins (GPIOs), meaning they can function as a digital input or a digital output.

When a GPIO is configured as an input, it reads the voltage on the pin and returns the appropriate binary value of 0 or 1. We might configure a GPIO as an input if we had some sensor and we wanted to use the data it collected from the world in a program. For example, using the data sensed by an accelerometer to direct the motion of an autonomous robot.

When a GPIO is configured as an output, it can be set to output a 0 or a 1 (0 V or V_{DD}). We might configure a GPIO as an output if we have some data in our program and we need our hardware to transform it into a physical, digital signal that we could present externally. For example, turning on an LED.

GPIOs can translate physical signals into 0s and 1s (or vice versa) but we need a way to access them from within a program executing on the processor. There are different ways we could do this:

1. Special Instructions
   Some ISAs include instructions specifically for I/O. For example,

IN R4, PORTB could be an instruction that reads the value from PortB as an input, and stores it in register R4.
OUT PORTB, R4 could be an instruction that outputs the value stored in register R4 to PortB.
With this method, the ports are special operands in the instructions.

2. Special Registers
   Some ISAs designate a specific register for output/input operations.
3. Memory Mapped I/O
   With this method, we refer to ports using addresses in memory.

We will be using memory mapped I/O. In our usual address space, certain addresses will be used for I/O operations. We will be using our usual load and store operations to access I/O devices.

Input vs. Output

Handling Outputs
There are on-chip registers connected to each port. These registers each have 32-bits, with each bit corresponding to one of the 32 pins in the port. Writing a 1 or 0 to a particular bit will cause the state to change accordingly on the corresponding pin, after a brief delay.

Handling Inputs
Inputs are more complicated to handle than outputs. By nature, when producing an output, the program itself is causing the change. The program writes to a register, which causes a physical signal to be output. Inputs, however, are caused by external forces. We need to have some way for the program to read the input value, which can change at any time. We can do this using *polling* or *interrupts*.

Polling
- Use software to periodically check the value.
- Within the program, we have a loop. In each iteration of the loop, we read the value.

Interrupt
- Use external hardware to notify the processor when there is a new input.
- Run the Interrupt Service Routine
- Return to the original program after the interrupt has been handled

Polling vs. Interrupts

Polling	**Interrupts**

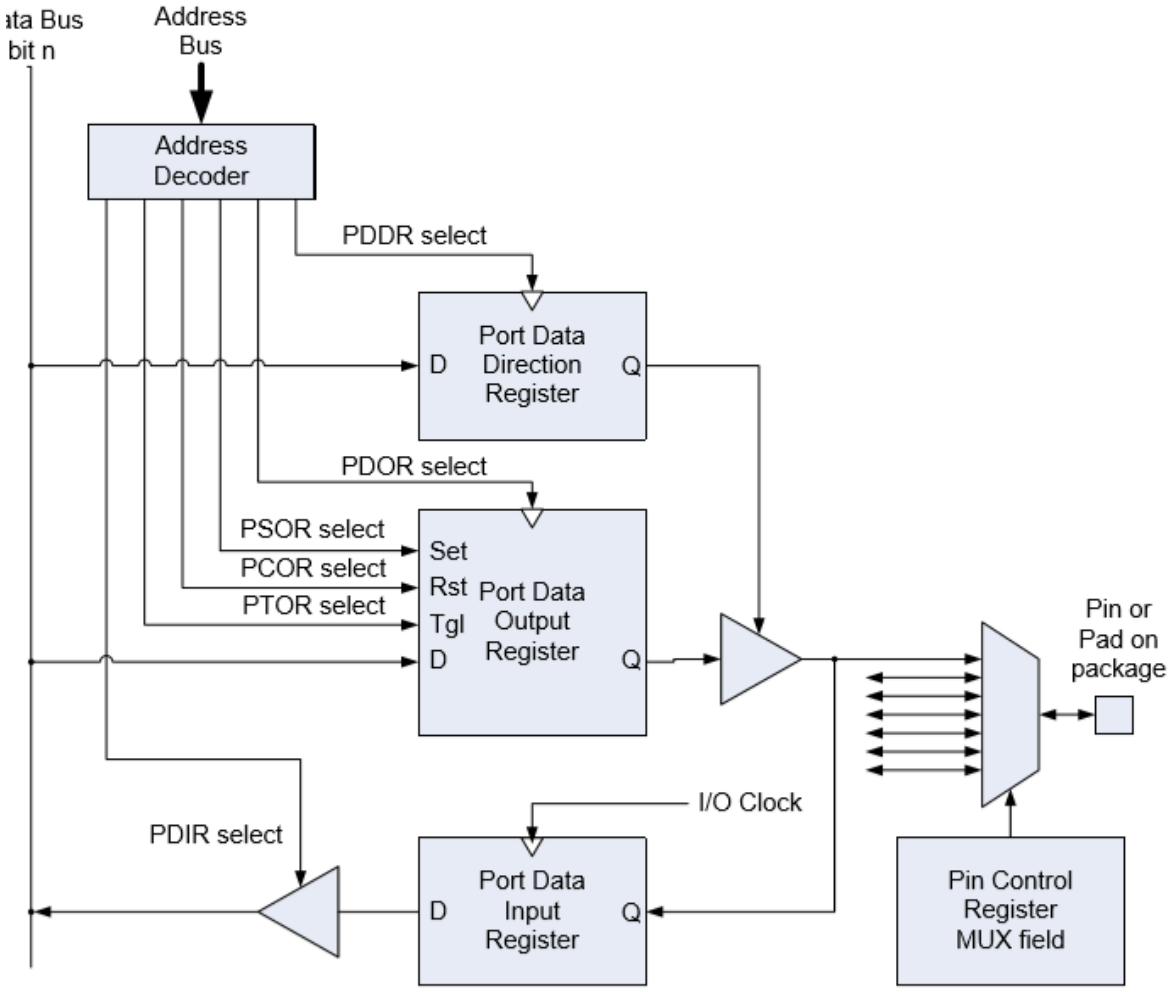
Simple and intuitive to implement.	More complex to implement.
Slower to respond to new inputs, since it checks the value periodically. The input can arrive in between checks.	Quicker to respond to new inputs. External hardware will trigger the interrupt as soon as a new input is received.
Scales poorly with the number of inputs we need to receive. The program needs to check the value of every single input. The time it takes increases proportionally to the number of inputs we must check.	Scales well.
Wastes CPU cycles. We expect that in the majority of checks we make, there is no new input. We only care about the cases in which there is a new input, so the time spent checking an unchanged value is effectively a waste.	Efficient. No time is wasted because the program executes normally without taking any additional time as long as no new input is received. Only executes the extra code when necessary (when there is a new input).

How do we access (read or write) I/O port from software?

- Option 1: Special Instruction in the ISA for input/output
    - Ports are special operands
- Option 2: Special Register in the ISA for input/output
    - I/O operation determined by writing specific values to this register
- Option 3: Memory Mapped I/O
    - Reads and writes to specific memory locations correspond to different I/O operations
    - The same address space is used for memory and I/O devices. When the CPU accesses the data at an address, it may refer to the I/O device.
    - Areas of the address space must be set aside specifically for I/O devices, not available for normal memory.
    - Requires more complex hardware to deal with addresses?
    - Benefit of making programming easier since the same addressing modes and instructions used for memory are available for I/O access, as opposed to having limited instructions you can use (remember load/store architecture)

Input vs Output

- Control
  - Direction Registers and Data Registers
  - can set the direction to configure a pin as an input or output
- output
  - typically easier to handle than input
  - there is a slight delay before the state changes on the pin / physical signal is output by the device
- input
  - how does the software know if the input is *valid* (in the 0 or 1 ranges, not intermediate or too high)?
    - valid bit = 8 bits of data, 9th bit is toggled to indicate new data
    - encoded data - 01 = false, 10 = true, 00 = no data
  - how does software know *when* the input is ready?
    - Polling
    - Interrupt



- triangles are tri-state buffers
    - passes input to output when enable is 1
- PDDR, PDOR, PDIR each have 32 bits because each port has 32 pins
    - each are flip flops ?
    - each port has its own PDDR, PDOR, PDIR, and PCR
- PDDR
    - PDDR select signal controls the direction - if we are configuring as input or output
    - 1 is output, 0 is input
- PDOR
    - has 32 bits, corresponding to the 32 pins. if a pin is configured as an output and its corresponding bit in the PDOR is 1, it outputs a 1 (for LED, lights up)
    - PSOR select (32 bits)

        - for 0, corresponding bits in PDOR are unchanged
        - for 1, corresponding bits in PDOR are set to 1
    - PCOR select (32 bits)
        - for 0, corresponding bits in PDOR are unchanged
        - for 1, corresponding bits in PDOR are cleared to 0
    - PTOR select (32 bits)
        - for 0, corresponding bits in PDOR are unchanged
        - for 1, corresponding bits in PDOR are toggled
    -
- PDIR
    - has 32 bits corresponding to 32 pins. if a pin is not configured as an input, the corresponding bit can have any value. if a pin is configured as an input, its corresponding bit will have the value of the input signal to this pin.
    - has a different clock, not sure why, but probably has to do with input coming at any time
- PCR (Pin Control Register MUX field)
    - pins can have up to 8 uses
    - in the lab we used (001) to set up a pin as a GPIO

- MUX/Decoder
    - for an output, its a mux which will transfer Q from PDOR to the port
    - for an input, it's a decoder which will transfer the pin values on the port which are configured as GPIOs to Q in PDIR

Interrupt Handling

The sequence:

- processor is executing the main code
- peripheral device sends an interrupt signal to the processor
- finishes current instruction (except for lengthy instructions)
- processor saves the current state
    - xPSR, PC, LR, R12, R3, R2, R1, R0
- HW puts interrupt/exception number in a register
- switches to privileged mode
- looks up the address for the appropriate ISR for this interrupt/exception (in the vector table)
- updates PC to the beginning of ISR
- updates LR with the return address (next instruction in main code to be executed after the interrupt is handled)
- execute the exception handler program

Questions

- why exactly do we save the xPSR?
    -
- why do we save both the PC and LR
    - because we are putting EXC_RETURN in LR
- why is R12 saved
    - R12 is the scratch register
    - apparently R12 is caller saved
- two stack pointers??
- where exactly is this vector table? what does it look like?
    - the vector table is just some space in our normal memory address space reserved for holding these values
    - the vector table helps us map an Interrupt Request to an appropriate ISR
    - for example, we know that we have IRQ #31, so we can use the appropriate pointer given by the formula (0x40 + 4(31))
    - this pointer is an address in memory that points to the vector corresponding to this interrupt request
    - this vector itself is an address for the interrupt service routine that handles IRQ#31

- non-maskable vs maskable interrupts
    - maskable - can disable them
    - non-maskable - can't disable them
    - volatile variables can be changed
        - ISR can change its value

- EXC_RETURN
    - 3 different values
    - tells us the return mode (handler or thread), the return stack (PSP or MSP)
    - MSP = 0, PSP = 1
    - Handler = 0, Thread = 1
    - suppose it is MSP and handler mode. we might do this if another exception handler was running when the current exception was requested.
    - suppose it is PSP and thread. we might do this if we want to continue executing the main program

- Exceptions include anything that interrupts the normal execution of the program
    - really, "exceptions" refers to the subcategory of things like Arithmetic Overflow. these exceptions occur entirely within the program, which is

why they are synchronous (on the same clock as the program is running on). they are generated by the program
    - there are also interrupts, which refer to the external interruptions of the program. they are caused by something external to the main program, which is why they are asynchronous.
    - finally system calls are triggered intentionally in the program to make use of the benefits of privileged mode. (OS)

- timing
    - critical path determines our limiting factor blah blah
    - we have to push PSR, PC, LR, R12, R3, R2, R1, R0 onto the stack
    - this requires 8 load instructions
    - load/store instructions take 2 cycles
    - so it takes about 16 cycles before we begin executing the ISR

- Stack Pointers
    - privileged mode (like ISR) uses MSP
    - unprivileged mode (like main program) uses PSP

- starting the exception handler
    - starts running unless there is a higher priority exception requested
    - can save registers on the stack
        - for example, in our ISR, we could've had a helper function to create a delay
- exiting the exception handler
    - we load the value in EXC_RETURN into PC
    - the is hardwired into the processor
    - it will restore the state so we can exit the handler
        - EXC_RETURN tells us which stack to consider (with MSP or PSP) so we know where to pop the context saved earlier from
        - PSR, PC, LR, R12, R3, R2, R1, R0

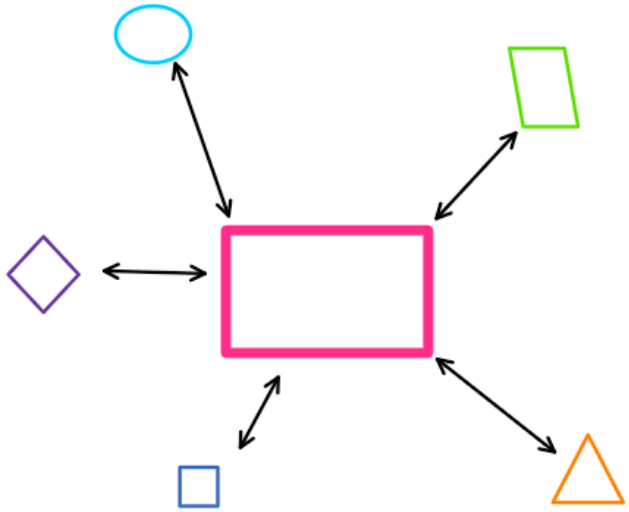
••••••
- in C, all variables are volatile by default
    - volatile = values can be changed at any time
    - non-volatile = preserves value in between function calls
        - initialization takes place only once
        - connection to write-back and write-through
        - can fail in the case of polling because you would load the initial value and keep checking that same value instead of checking the new input each time
            - in general, because we aren't explicitly changing the value of the variable within the program

Interrupt Handling	**Function Calls**
Caller: interrupt software Callee: Interrupt Service Routine	Caller: some function, like main, which calls a subroutine Callee: the function called

Buses

At a high level, a bus is a network which connects various components in a computer, or multiple computers together. It's analogous to the network of nerves connected in a nervous system to our brains that allow us to make full use of our entire bodies. We can imagine all of our body parts as components connected to a computer. When we see a baseball flying towards us, we need to be able to interpret that visual information sensed in our eyes to direct the movement of our hand to catch it.

But the way in which we connect these external components is important. In our computer, we could have each external component interface solely with the CPU. The CPU would receive signals, and send them out to the appropriate external components they are relevant to.

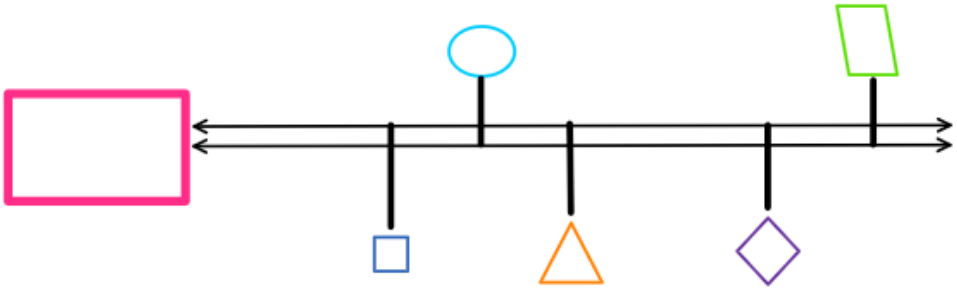


Imagine how complex the system above becomes as we add more components. In a car, for example, we have many, many services provided by specialized electronic components - mirror adjustment, seat adjustment, window control, sound systems, airbags, power steering, cruise control, and more. Connecting all these modules to a central computer individually is complex - requiring the computer to allow for a high number of wired connections, necessitating long lengths of wire in large systems.

CAN Bus

The Controller Area Network (CAN) Bus Protocol was developed specifically to solve this problem in the automotive industry. CAN is one protocol - set of rules - defining how data should be transferred in a network between computers, or between components inside a computer.

CAN is efficient, low cost solution which is flexible and robust.
The CAN Bus Protocol requires no singular host.
It is a *broadcast* type of bus.
It is a *message-based protocol*.
It is easy to add and remove nodes from the network, since CAN doesn't require identifying/referencing nodes in the network.
In the CAN Bus Protocol, all of the components in the network communicate along just two wires.



~~The two wires are CAN High (CANH) and CAN Low (CANL). The reason for this is that CAN was developed to interface computers in relatively large systems: cars. To make these long wires robust against noise, the CAN protocol says that communication must be done by *differential signalling*. With traditional signalling, the signal is sent along one wire and its value is relative to GND. With two wires, however, we can send the original signal along the CANH wire, and the inverted signal along the CANL wire.~~

CAN Bus

In computing, a bus is a communication system that transfers data between components inside a computer or between computers. Each component connected to a bus is called a node. Buses use a set of rules for the transfer of data. This set of rules is called a protocol. We will learn about one such bus protocol - CAN (Controller Area Network) Bus Protocol.

CAN bus is a serial data bus which means that it transfers one bit of data (0 or 1) at a time. It has multiple masters i.e., any node that is connected to the node can initiate the transmission of data. It is a broadcast bus so the data that is sent by a node on the bus can be read by every other node connected to the bus.

CAN bus uses a wired-AND method for arbitration. Arbitration is a situation in which two or more nodes are trying to send data to the bus but since it is a serial data bus, it can accept data from only one of them. Remember that AND returns 1 only if all the operands are 1. Similarly, the bus accepts a value of 1 only if all the nodes that are trying to send data are sending 1, otherwise it only accepts 0. In this sense, 0 is the **'dominant'** value and 1 is the **'recessive'** value since in a competition between 1 and 0, 0 wins.

Instead of a single wire for the transmission of data, CAN bus uses a pair of wires and the voltage difference between these two wires signals the value that is currently on the bus. If a node is sending 0, it will pull one of the wires to high voltage, say 5 V and the other wire to low voltage, say 0 V. This creates a voltage difference of 5 V between

the wires and the value is interpreted as 0 in the digital sense. Conversely, if all nodes are sending 1, then the wires are pulled together to a middle voltage, say 2.5 V so the voltage difference between them is 0 V and this is interpreted as a 1 in the digital sense.

Each node on the CAN bus reads the bus in each cycle even if it is not sending a value. If the value that it reads is not the same as the one it is sending, it is likely that it is trying to send a 1 that is being 'dominated' by a 0 being sent by another node.

A single bit 0 at the beginning of the frame indicates a **start-of-frame**. When a frame is done, it ends with a bunch of 1's. So this 0 will indicate the start of a new frame. This is followed by an 11-bit message ID that indicates what kind of message is being sent. Again, 0 is dominant and 1 is recessive.

Example: Consider 3 devices. Device 1 is trying to send data with message ID 101....., device 2 is trying to send data with message ID 011..... and device 3 is trying to send data with message ID 001..... . Remember that these message IDs are 11 bits long. All of the three devices will send the first bit of the ID to the bus. Because of the AND arbitration, 0 will be accepted. When device 1 reads the bus, it will realize that the value on the bus does not match what it is sending so it will back off and try again in the next frame i.e., it will not try to send any data for the remainder of this frame. This is why messages that are of higher priority should have lower message IDs so that there are more 0's and it goes to the bus faster.

The next bit of the frame is the **RTR** (Remote Transmission Request) bit. If this bit is 0, it means that the node that is initiating the transmission is sending the data. If it is 1, it means that the node is requesting another remote node to send the data. In this sense, 0 is a write and 1 is a read request. The next bit is **IDE** (Identifier Extension Bit) which indicates if more bits are required for the message ID.

The next 4 bits represent the **length of the data** being sent to the bus. However, the actual maximum length of data that can be sent is 8 so any number greater than 8 will be interpreted as 8. The next 8 bits (1 byte) are reserved for the **actual data** to be sent. If this was a Remote Transmission Request, then you would have no data in this field.

The rest of the CAN bus frame is used to make sure that the data is sent correctly. Each node has hardware to compute **CRC** (Cyclic Redundancy Code) from a given data. Once a node receives data, it computes the CRC and compares it with the 15-bit CRC on the bus. If there is a mismatch, it is likely that there was an error in sending the data. The node sort of tells all the other nodes that there was an error by sending an error frame that is just 6 consecutive 0's. If this happens, the transmission of data will be restarted. The **CRC delimiter** bit is always set to 1. If a node receives the data

correctly (CRC matches), it will write a 0 to the **ACK** (acknowledgement) bit. This is to ensure that at least one node received the message correctly. The **ACK delimiter** is always set to 1. The end of the frame (**EOF**) is indicated by 10 consecutive 1's. After this, the frame can be reinitiated.

CAN bus has no explicit clock signal so there is a need to synchronize the internal clocks of the nodes. The SOF is always 0 and EOF is 1. So the start of a new frame is always indicated by 10 1's followed by a 0. Whenever this happens, all the nodes synchronize their clocks to the clock of the node that initiated the transmission. At other times, the nodes resynchronize their clocks whenever the value on the bus changes. But there is a problem if the data on the bus does not change for a lot of cycles. In order to give the clocks an opportunity to resynchronize, the CAN bus uses a **stuffing bit**. The hardware inserts a bit of the opposite value after every 5 consecutive identical bits. The receiver knows that after every 5 identical bits, the 6th bit is a stuff bit so it discards this bit while decoding the message. This is why it makes sense for an error frame to be indicated by 6 consecutive 0's.