CCNA 200-301 :
Day 5: Ethernet LAN Switching Pt.1 :-

IP addresses are Layer3 not Layer2!

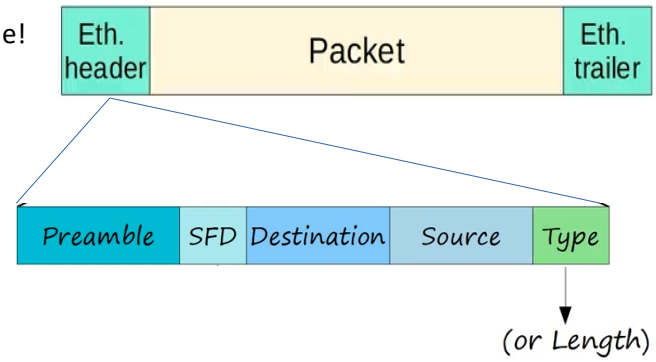
Ethernet involves Layer1 and Layer2 of the **OSI** Model.

LAN : is a network contained within a relatively small area, like an office floor or your home network!
Routers are used to connect separate LANs.
Switches do not separate LANs, but adding more can be used to expand an existing LAN.
If there are two switches connected to different router interfaces, they make two different LANs!

In this lesson: we'll look at how traffic is sent and received within a LAN:

As the Layer2 **PDU** is called a **frame**, we're going to focus on how switches receive and forward frames, specifically Ethernet frames, since it's the Layer2 protocol used in virtually every LAN in existence today!!

Encapsulating the Packet with a header and trailer, We got an Ethernet frame!



→ **Let's look at the header:** there are **5** fields in the header

SFD: Start Frame Delimiter

The **Preamble** and **SFD** are used for synchronization and to allow the receiving device to be prepared to receive the rest of the data in the frame.

Destination: the Layer2 address to which the frame is being sent.
Source: the Layer2 address of the device which sent the frame.

Type: indicates the Layer3 protocol used in the encapsulated Packet, which is almost always **Internet Protocol, IPv4** or **IPv6**. However, sometimes this is a **Length** field, indicating the length of the encapsulated data, depending on the version of Ethernet.

The **Ethernet trailer** has only 1 field, the **FCS** which stands for **Frame Check Sequence**.
FCS is used by the receiving device to detect any errors that might have occurred in transmission.

→ Let's talk in more detail:
Let's talk about the Preamble and the SFD first, which you can think of them as a **set**!

Preamble: is **7** bytes long (56 bits), and a series of **Alternating 0**'s and **1**'s like this **10101010 * 7**.
The purpose of this is that it allows devices to synchronize their receiver clocks, to make sure they're ready to receive the rest of the frame and the data inside

SFD: it's length is **1** byte (8 bits), its bit pattern is **10101011**, similar to each byte of the preamble but the last bit is a **1**.
It indicates or Mark the end of the Preamble and the beginning of the rest of the frame!

Destination and **Source**: indicate the devices sending and receiving the frame, like when you send an e-mail, both the source and destination are included in the e-mail.
The addresses are used in Ethernet are the destination and source **'MAC addresses'**, **MAC** stands for Media Access Control.
MAC is **6** bytes (48 bits), address of the **Physical device**, and is actually assigned to the device when it is made!!
This is separate from a logical address like an **IP** address.

Type or **Length**:
The last field of the Ethernet frame, It is **2** bytes (16 bits) length.
It can be used to represent either the type of the encapsulated packet, or the length of the encapsulated packet.
→ If the value in this field is **1500 or less**, this means it is indicating the **LENGTH** of the encapsulated packet (in bytes)!
For example, if the value in this field is **1400** it means that the encapsulated packet is 1400 bytes in length!
→ If the value was **1536 or greater**, this means it indicates the **TYPE** of the encapsulated packet (usually **IPv4** or **IPv6**).
And the length is determined via other methods.
For example, a value of **0x0800** which is written in hexadecimal and is equal to 2048 in decimal, indicates the version of the **IP**.
And the 2048 is greater than 1536, and identify that the version of the **IP** is version 4 (**IPv4**).
For example, a value of **0x86DD** which is written in hexadecimal and is equal to 34525 in decimal, indicates the version of the **IP**.
And the 34525 is greater than 1536, and identify that the version of the **IP** is version 6 (**IPv6**).

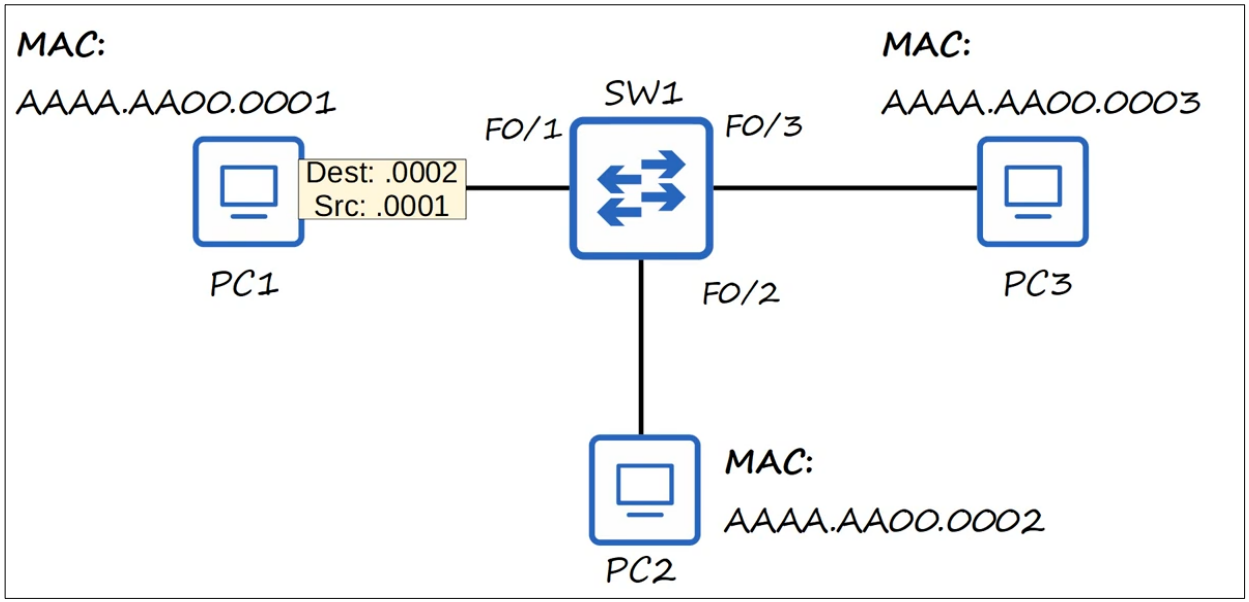
Try to remember the length of each field in the Ethernet frame!

The only field of the Ethernet trailer is the **FCS**, which stands for **Frame Check Sequence**.
It is **4** bytes (32 bits) in length. Its purpose is to detect corrupted data by running a '**CRC**' algorithm over the received data.
CRC means **Cyclic Redundancy Check**.
Cyclic refers to something called 'Cyclic Codes', **Redundancy** refers to the fact that these **4** bytes at the end of the message, enlarge the message without adding any new information, so they are **redundant**. **Check** refers to the fact that it checks, or verifies, the data for errors.

→ **Now**, the total size including ethernet header and trailer, to **26** bytes (208 bits).

→ MAC Address:-

MAC stands for Media Access Control.
MAC is **6** bytes (48 bits), address of the **Physical device**, and is actually assigned to the device when it is made!!
This is separate from a logical address like an **IP** address
A.K.A '**Burned-In Address**' (**BIA**), This is because the address is burned-in to the device as it is made!
The **MAC** address is *globally* **unique**! You'll never see two devices in the world have the same MAC address.
Although, there are MAC addresses known as '*locally-unique*' addresses, which don't have to be globally unique through out the world, However, In almost all cases MAC addresses are *globally unique*.
The first **3** bytes of the MAC address are the **OUI** '***Organizationally Unique Identifier***', which is assigned to the company making the device. So **Cisco** will have various **OUI**s which only Cisco can use!
The last **3** bytes are unique to the device itself.
MAC addresses are written as a series of **12 hexadecimal** characters.



This kind of frame is called **Unicast frame**: a frame destined for a single target.

So, **PC1** sends the frame through it's network interface card, which is connected to **SW1**, after the **SW1** receives the frame, it looks at the source MAC address field of the frame and then uses that information to ***learn*** where **PC1** is!

MAC Address Table

MAC	Interface
.0001	F0/1

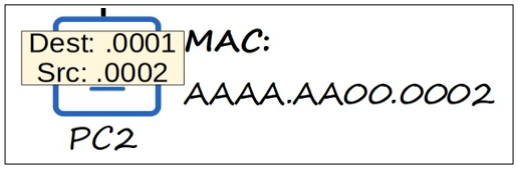
As you can see here, it adds the MAC address to it's MAC address table, and it associates that MAC address with its **F0/1** interface!
This is known as a ***Dynamically Learned MAC address***.
Or Just ***Dynamic MAC address***.
Because it wasn't manually configured on the Switch. The Switch learned it itself.
Every Switch will keep a MAC address table like this and fills it dynamically by looking at the source MAC address of frames it receives!

Since **SW1** received a frame from source MAC address AAAA.AA00.0001 on it's **F0/1** interface, It knows that it can ***reach*** that MAC address on **that** (**F0/1**) interface! And adds it to the MAC address table! *→ An Important Concept! ←*
→ This is how Switches dynamically learn where each device on the network is!, by looking at the source MAC address of the frame.

NOW, there is one problem:
The destination of the frame is AAAA.AA00.0002, but **SW1** doesn't know where that is!
This by the way, called an ***Unknown Unicast Frame***, a frame for which the Switch doesn't have an entry in its MAC address table.

This has only one option, that is to **flood** the frame!
Flood means to forward the frame out of ALL of its interfaces, except the one it received the packet on (**PC1**).
It will happen like this, **SW1** copies the frame and sends it out its **F0/2** and **F0/3** interfaces. (but not the **F0/1**)
After that, **PC3** ignores the packet because the destination MAC address doesn't match its own MAC address, and simply drops out the packet!
PC2, however, receives the packet, and then processes it normally, up the **OSI** stack.
However, unless **PC2** sends a reply of some sort, it stops here!

SW1 never receives a packet from **PC2**, so it can't learn **PC2**'s MAC address and use it to populate the MAC address table!
So, let's say **PC1** sends another frame to **PC2**, the same process happens again, **SW1** will flood the packet, PC3 drops the frame and **PC2** receives it and process it normally.



NOW, Let's say **PC2** wants to send some data to **PC1**, may be a reply to what it sent.
PC2 sends he frame out of its network interface, and **SW1** receives it.
Now **SW1** looks at the source MAC address of the frame, and then adds **PC2** MAC address to its MAC address table, associating it with the **F0/2** interface.
This time, however, it doesn't **flood** the frame. This is called a ***Known Unicast Frame***, because the destination is already in its MAC address table. And then **PC1** process the frame up the **OSI** stack, through the de-encapsulation process.
Whereas UNKNOWN Unicast Frames are *flooded*, KNOWN Unicat Frames are simply *forwarded*.

NOTE : Dynamic MAC addresses are removed from the MAC address table after **5 minutes** of inactivity!

→ Let's Look to an another example, with 2 Switches :

If **PC1** wants to send some data to **PC3**, the source MAC address is AAAA.AA00.0001 and the destination MAC address is AAAA.AA00.0003.
So, **PC1** sends the frame out of its network interface **F0/1** and it arrives at **SW1**, and then **SW1** learns **PC1**'s MAC address from the source address field of the frame, and associates it with the interface on which it was received **F0/1**. Now **SW1** has learned that **PC1** can be reached via it's **F0/1** interface, but still doesn't know where **PC3** is! This is an UNKNOWN Unicast Frame!

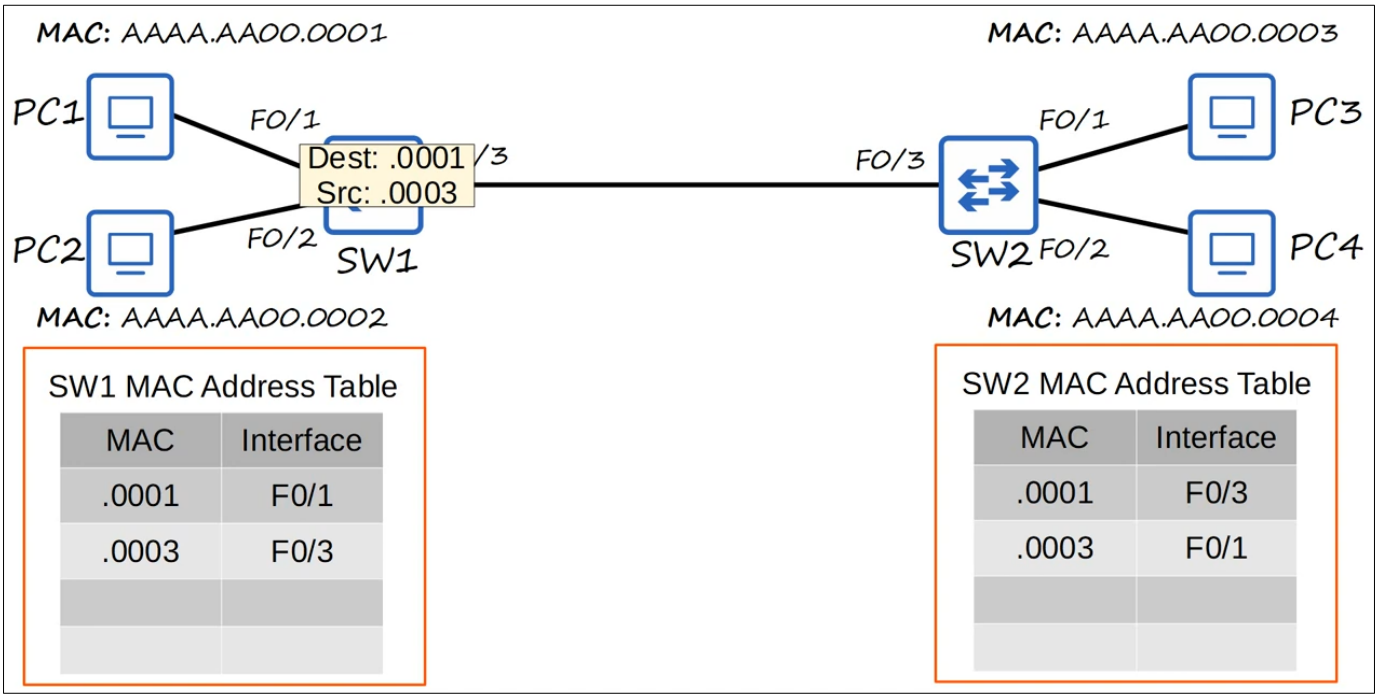
Now **SW1** *floods* out all of it's ports, except the one it was received on!
In this case it will *flood* the frame out of **F0/2** and **F0/3**, but not **F0/1**!
NOW **PC2** drops out the frame, and the exact same rules *apply*.
Just like **SW1** did, **SW2** uses the source MAC address field of the frame to dynamically learn **PC1**'s MAC address and the interface it can reach **PC1**. (associate it with **F0/3**)

NOTE that, unlike on **SW1, PC1** isn't actually directly connected to the interface **SW2** enters in its own MAC address table!
However, this is the interface which **SW2** will use to reach **PC1**.
And that is the meaning of the interface in the MAC address table, it doesn't mean the device is directly connected to this interface.

NOW **SW2** received a Unicast Frame, that is a frame destined for a single device, but still doesn't where the destination MAC address is!? Because it is not on its own MAC address table! So it will *flood*! **SW2** floods the frame out all interfaces, except the one it was received on (**F0/1** and **F0/2** but not **F0/3**)!
And Now **PC4** drops out the frame as the destination MAC address doesn't match its own!
PC3 receives the frame as matches its own MAC address! And let's say **PC3** wants to reply to **PC1**!
NOW the source and the destination MAC addresses will be reversed!
PC3 sends the frame out of its network interface and it is received by **SW2**.
SW2 learns **PC3**'s MAC address and associates its **F0/1** interface in its MAC address table.

The Switch uses the Source MAC address field to fill its MAC address table, because if it receives a frame from that source on the interface, it knows that it can reach that MAC address via that interface.

SW2 already has an entry for the destination MAC address in its table, so there is **no** need to *flood* the frame!
Now it is *forwarded* normally out of the corresponding interface in the MAC address table which is **F0/3**!
The frame is received by **SW1** which adds an entry for **PC3**'s MAC address in it own table with the interface **F0/3**.
Finally, since **SW1** has an entry for the destination MAC address in its own table, it will then *forward* the frame out of the corresponding interface **F0/1**, and it reaches its destination **PC1**.



And here are some information about the HEXADECIMAL

DEC.	HEX.	DEC.	HEX.	DEC.	HEX.	DEC.	HEX.
0	0	8	8	16	10	24	18
1	1	9	9	17	11	25	19
2	2	10	A	18	12	26	1A
3	3	11	B	19	13	27	1B
4	4	12	C	20	14	28	1C
5	5	13	D	21	15	29	1D
6	6	14	E	22	16	30	1E
7	7	15	F	23	17	31	1F

Each hexadecimal digit is **4** bits, so **2** digits equals **8** bits or **1** byte.

QUICK:

SFD signifies the end of the **Preamble**, and not used to provide receiver clock synchronization.
Preamble is an alternating series of **1**'s and **0**'s, allows the receiving device to synchronize its receive clock!
FCS is used to detect errors that occurred during transmission.
OUI is the first half (**24** bits) of a MAC address. It's a unique value assigned to the maker of the device.
Source is the field of the Ethernet frame which the Switch uses it to populate its MAC address table.
Destination field of the Ethernet frame also **specifies** a MAC address, it doesn't help the Switch to populate its MAC address table.
UNKNOWN Unicast frame is the kind of frame that Switch *floods* out of all interfaces except the one it was received on.
KNOWN Unicast frame is a frame for which the destination MAC address is already in the Switch's MAC address table.

Day 6: Ethernet LAN Switching Pt.2 :-

→ We will talk a little bit about the frames:
The **Preamble** and **SFD** are not considered part of the Ethernet header, although they're sent with every Ethernet frame!
So the Ethernet header consists of the three fields (**Destination**, **Source** and **Type**).
Therefore the size of the Ethernet header + trailer will be **18** bytes.
There is a minimum size for an Ethernet frame which is **64** bytes including (header + Payload [Packet] + trailer).
64 bytes – **18** bytes (header + trailer) = **46** bytes for the **Payload**.
The **64** bytes doesn't include the **Preamble** and **SFD** by the way.

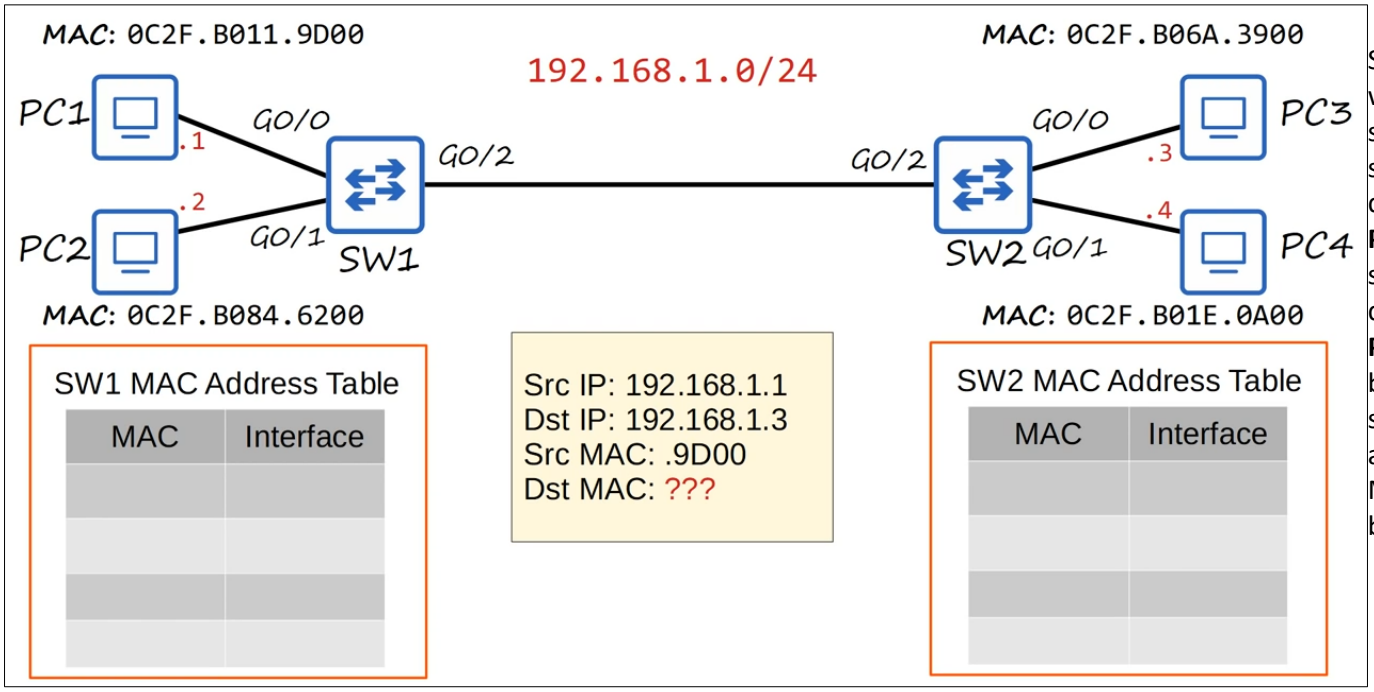
If the **Payload** is less than **46** bytes, *padding* bytes are added to make it equal to **46** bytes, and these bytes are **zeros** by the way!

Remember: that the **Preamble** and **SFD** might not be included as part of the Ethernet header, depending on how you define it.
But they are included in every Ethernet frame.

→ Let's get started:

We made a simple change to the previous network, we made the Interface as **Gigabit Ethernet** like (**G0/0**, **G0/1** and **G0/2**) and we also gave the PCs a more realistic MAC addresses.
And we'll add some IP addresses, **192.168.1.0/24**, this IP address represents the whole network here.
Then **192.168.1.1**, represents the **PC1** IP address and **192.168.1.2** represents the **PC2** IP address, etc.

When a device sends some data to another device, it doesn't just include a **Source** and **Destination** MAC address.
Encapsulated within that frame is an **Internet Protocol**, a.k.a, **IP** packet, and that IP Packet includes a **Source** and **Destination** IP address.



So, If **PC1** wants to send some data to **PC3**, the src and dest IP of **PC1** will be as shown and src MAC will be abbreviated as shown.
However, **PC1** doesn't know **PC3** MAC address.
When you send data to another computer, you enter the IP not the MAC address.
So the user enter the destination IP, but **PC1** has to discover **PC3**'s MAC by itself!

Remember: These 2 Switches operate at Layer2 not Layer3, so they need to use MAC addresses not IP addresses.

So, If **PC1** wants to send the frame to **PC3**, but first it has to learn **PC3**'s MAC address.
To do so, it uses something called **ARP**, **Address Resolution Protocol.**

Let's take a look at ARP:

→ It is used to discover Layer2 address (MAC) of a known Layer3 address (IP).

→ **ARP** process consists of 2 messages:

- **ARP Request:** sends by the device wants to know the MAC address of the other device.
- **ARP Reply:** which is sent to inform the requesting device of the MAC address.

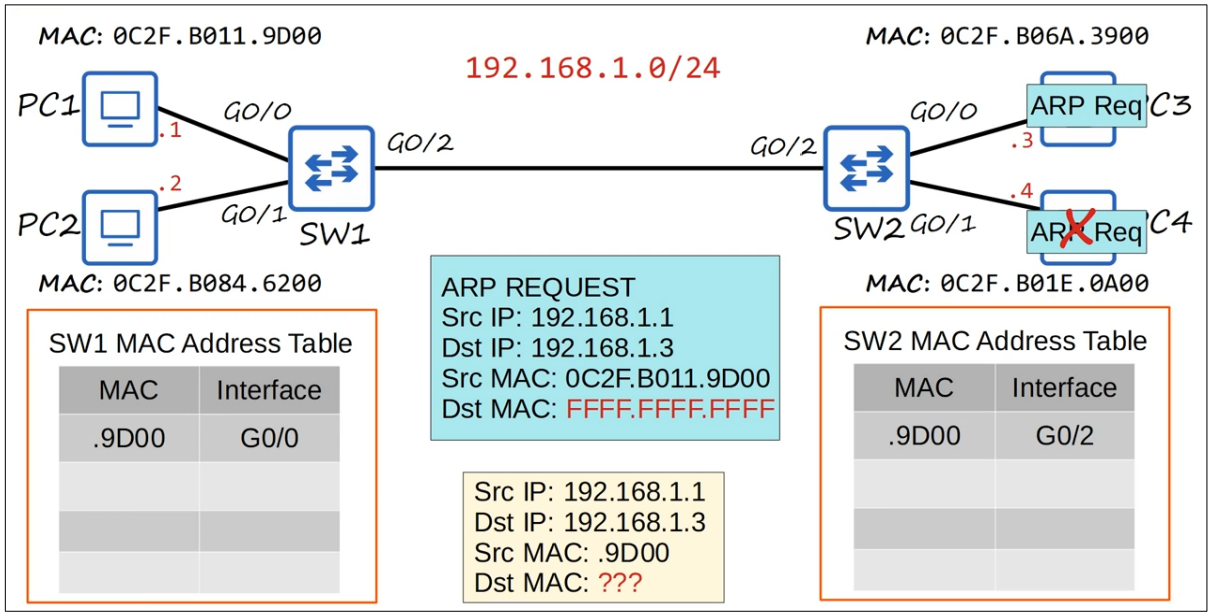
In our example, **PC1** would sent the **ARP Request** and **PC3** would sent the **ARP Reply**.

→ **ARP Request** is **broadcast**, which means it is sent to all hosts on the network.

Because the Layer2 address of the destination is unknown, it **broadcasts** the request and waits for a reply from the correct device.

→ **ARP Reply** is **unicast**, which means it is sent only to one host (the host that sent the ARP request).

→ **FFFF.FFFF.FFFF** is the broadcast MAC address that is sent as the destination MAC address in the **ARP Request**.



Now, **PC1** sends the frame out of its network interface and it is received by **SW1**, which is adding **PC1**'s MAC address to its table and associates it with the **G0/0** interface.

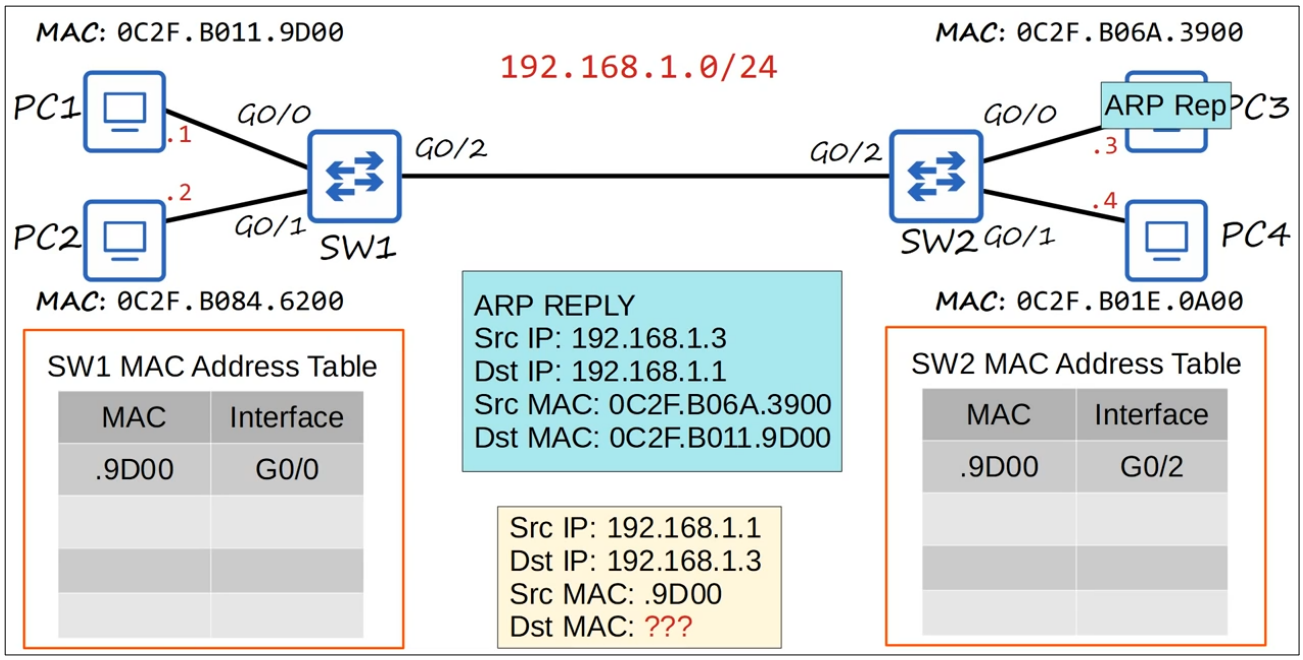
Since the destination MAC address is all **FFFF**'s, **SW1** broadcasts the frame out of all of its interfaces except the one it was received on. This is very like what a Switch does with an UNKNOWN Unicast frame. **PC2** receives it, but ignores the frame.

Then **SW2** learns **PC1**'s MAC address and adds it to its table associating it with **G0/2** interface.

Since the destination MAC address is all **FFFF**'s, SW2 also broadcasts the frame out of all of its interfaces except the one it was received on. Now **PC4** ignores the frame as the destination IP address doesn't match its own IP address.

However, **PC3** recognizes that he dest IP address does match its own, so it doesn't ignore the **ARP request**.

What **PC3** really does is send the other message, the **ARP reply**. Now we can see the **ARP reply** packet reversed to the previous **ARP request**.

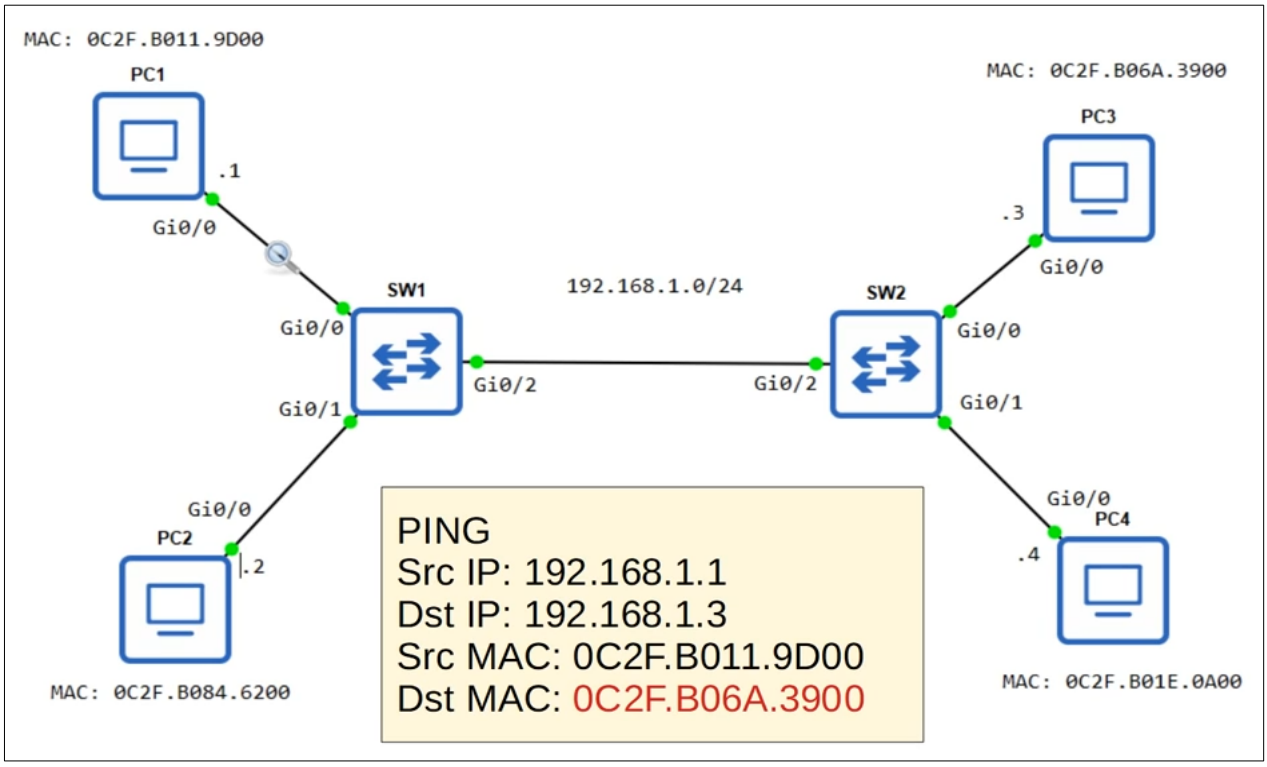


Now the destination MAC address matches the own MAC of **PC1**.
Now **PC3** sends the frame out of its interface to **SW2** directly, as it has an entry for the destination of the MAC address of **PC1**.
This kind of frame is a KNOWN Unicast frame and **SW2** *floods* it directly.
SW2 learns the **PC3**'s MAC address and adds it to its own table associating it with its **G0/0** interface.
SW2 sends the frame towards **SW1** as it learns the interface of **PC1**, **SW1** receives the frame and since it is already learned **PC1**'s MAC address on the **G0/0** interface, it simply forwards the frame out of the interface and **PC1** finally receives the frame!
PC1 will then use that information to add an entry for **PC3** to its **ARP Table**. Which is used to store these IP address to MAC address associations.

→ Let's take a look at the ARP Table:
we use the command (`arp -a`) to view the ARP Table, On Windows, Linux or MacOS.
This screenshot is from my pc:

```
C:\Users\hp>arp -a

Interface: 192.168.242.1 --- 0x3
  Internet Address      Physical Address      Type
  192.168.242.255       ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static

Interface: 192.168.56.1 --- 0x8
  Internet Address      Physical Address      Type
  192.168.56.255        ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static

Interface: 192.168.221.1 --- 0xf
  Internet Address      Physical Address      Type
  192.168.221.255       ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

The **Internet Address** column displays IP addresses (Layer3 addresses).
The **Physical Address** column displays MAC addresses that correspond to the IP addresses (Layer2 addresses).
If the **Type** column displays ***static***, it means that it is a default entry, it wasn't actually learned by sending an ARP request.
However, If the **Type** column displays ***dynamic***, it means that the entry was learned by sending an ARP request and receiving an ARP reply.
NOTE: Switch only floods the UNKNOWN Unicast frame and the Broadcast frame.
→ Packet tracer & GNS3 :
Packet tracer is a network simulator, it's a piece of software designed to simulate the operation of the real network.
GNS3, however, runs actual Cisco IOS software, so these are real Cisco switches running virtually.
GNS3 requires you to purchase your own copies of Cisco IOS, so although GNS3 itself is free, using Cisco IOS with GNS3 is not!



If **PC1** wants to *ping* **PC3**, the same previous operation will occur.

→ Ping:

- A network utility that is used to test reachability.
  For example, to test if two computers can reach each other
- Measures round-trip time.
  For example, time from **PC1** to **PC3**, then back to **PC1**.
- Uses two messages: somehow similar to **ARP**
  1. **ICMP Echo Request**, but the PC won't broadcast the request, it is sent to a *specific* host. It has to know the MAC address of the destination host, which is why **ARP** must be used first.
  2. **ICMP Echo Reply**

To use *ping*, we use the command (**`ping + ip`**)

```
PC1#
PC1#ping 192.168.1.3
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.1.3, timeout is 2 seconds:
.!!!!
Success rate is 80 percent (4/5), round-trip min/avg/max = 20/20/22 ms
PC1#
```

By default, a *ping* in Cisco IOS sends **5 ICMP echo requests**, and then you should get **5 ICMP echo replies** back. And the default size of each ping is **100**-bytes.

The period or the (**.**) indicates a failed ping, and the exclamation points (**!**) indicate a successful ping.

The first ping fails because of the **ARP**.

PC1 didn't know the destination MAC address, so it had to use **ARP**, and in that time the first ping failed.

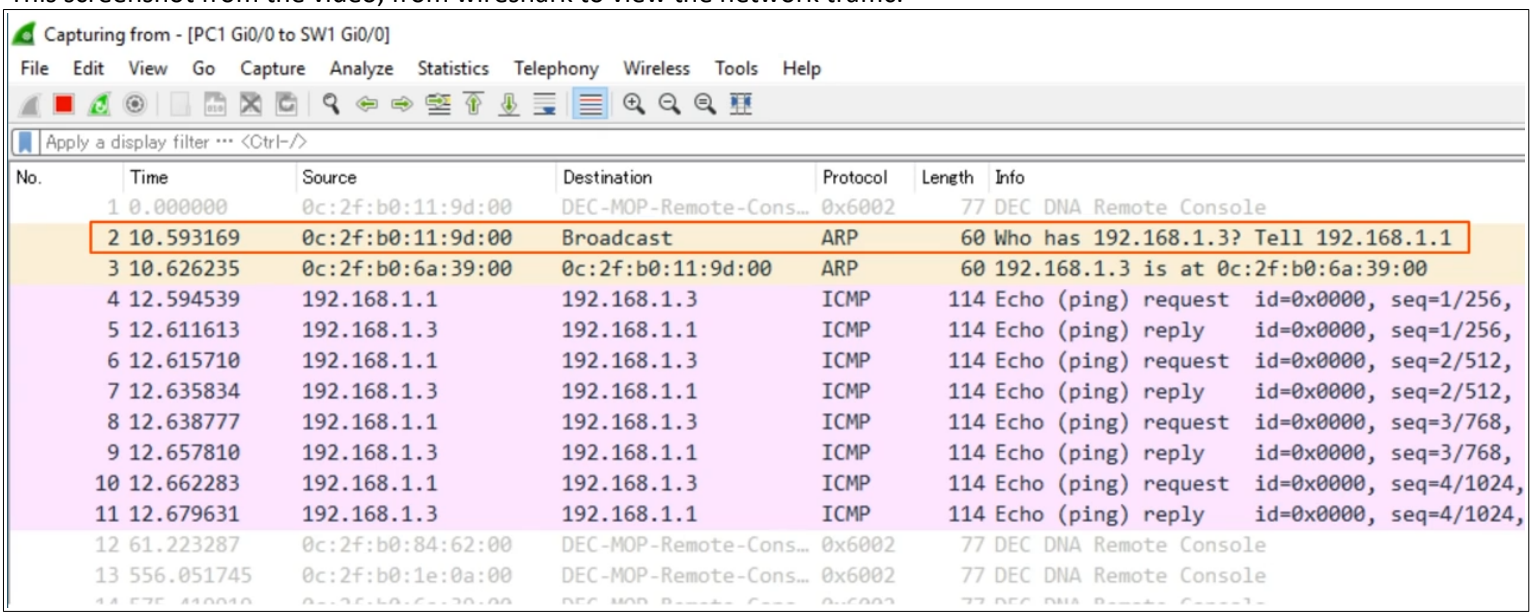
After **PC1** learned **PC3**'s MAC address, however, the ping succeeded !

Let's take a look at the **ARP** table, in Cisco IOS, like the previous screenshot:

The command only in Cisco IOS, is (**`show arp`**)

```
PC1#show arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  192.168.1.1            -   0c2f.b011.9d00  ARPA   GigabitEthernet0/0
Internet  192.168.1.3           34   0c2f.b06a.3900  ARPA   GigabitEthernet0/0
PC1#
```

Wireshark allows you to perform '**packet captures**', to analyze the contents of network traffic.
This screenshot from the video, from wireshark to view the network traffic:



The **ARP request** is asking, which MAC address has an IP address of **192.168.1.3**?, and to send the reply to itself **192.168.1.1**.
It is a **broadcast**, all are **FFFF**'s.
Next is the **ARP reply**, from **PC3** with the destination to **PC1**'s MAC address, in the Info section, telling **PC1** its MAC address!
After that there are **4 ICMP echo requests** and **4 ICMP echo replies**!
Note that: the **ICMP echo requests** have a source IP of **PC1** and destination of **PC3**, and the **ICMP echo replies** have a source of **PC3** and destination of **PC1**.

Basically If device A wants to send some traffic to device B, which is on the same network, device A first has to use ARP to learn device B's MAC address, and then it can send traffic to device B.

→ Let's take a look at the MAC address table on a Cisco Switch:
to view it we use the command (**`show mac address-table`**)

```
SW1#show mac address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type        Ports
----    -----------       --------    -----
   1    0c2f.b011.9d00    DYNAMIC     Gi0/0
   1    0c2f.b06a.3900    DYNAMIC     Gi0/2
Total Mac Addresses for this criterion: 2
SW1#
```

Vlan means Virtual Local Area Network.
The default value of **VLAN** is **1**.
Dynamic MAC addresses display in the MAC address table as well. As the Switch learned them dynamically.
And we have **Ports**, which is another word for **Interfaces**.
Remember that these dynamic MAC address are removed from the MAC address table after **5** minutes of inactivity.
This is known as **Aging**.
However, you can also *manually* remove MAC addresses from the table by using the command;
(**`clear mac address-table`**).
If you don't want to clear all the MAC addresses from the table you can add some additional options to the command, like:

- **`clear mac address-table dynamic address 0c2f.b011.9d00`**
- **`clear mac address-table dynamic interface Gi0/0`**
  - This clears all MAC address table entries for a specific interface.

Some **Wireshark**:
Minute **24:10** to Minute **26:00**

NOTE: The **ARP** Ethernet type is (**0x0806**).
This indicates that an **ARP** packet is inside of this Ethernet frame.

QUIZ:

Both the **ICMP request and reply** are unicast messages.
The **ARP request** message is used to learn the Layer2 address of a host.

Day 7: IPv4 addressing Pt.1 :-

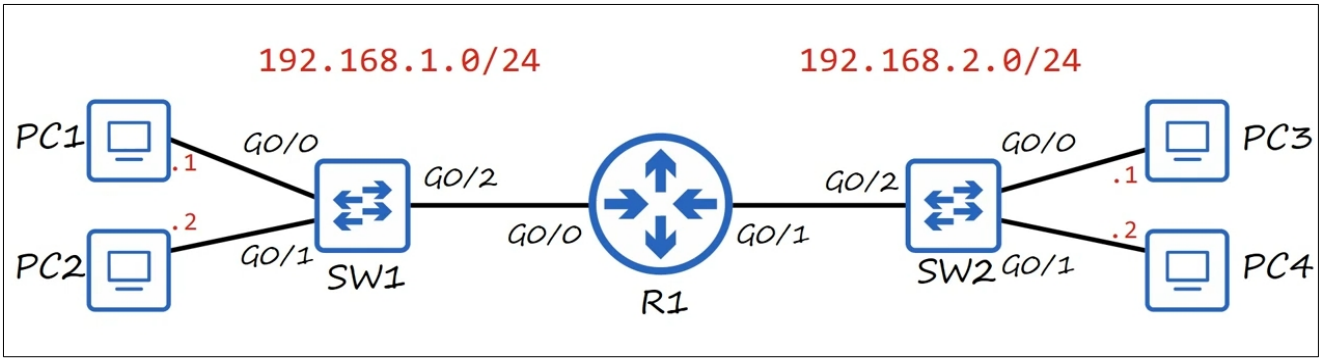
→ <u>How the traffic forwarded between different LANs</u>
→ Let's get started:

We are going up the **OSI** model, from Layer2, Data Link layer, to Layer 3, the Network Layer.

Some review on Layer3:
→ The network Layer provide connectivity between end hosts on *different* networks (outside of the LAN).
→ Provides logical addressing (**IP** addresses).
→ Provides path selection between source and destination.
→ Routers operate at Layer3.

IP addresses are logical addresses you assign when you configure the device.
Over larger or complex networks, like the Internet, there can be many different possible paths to a destination. Selecting the best path is part of Layer3's functionality.

→ Our focus will be specifically on the Logical Layer3 addresses (*IP Addresses*):



Now If we put a Router between the previous network with 2 Switches, The **R1** will split them into two separate networks!
<u>*NOTE*</u>: the first 3 groups of numbers of theses network addresses represent the network itself (**192.168.1** or **192.168.2**).
and only (**.0**) changes to represent the end hosts on the network.
(**/24**) at the end means, they are used to tell what part of the address represents the network and which part represents the end hosts, the PCs. (**/24**) means that the first **3** groups of numbers represent the network.

There is one thing is missing in this network diagram, the Router needs an IP address!
And not only one IP address, it needs an IP address for each network it is connected to.
So, let's give the **R1**'s **G0/0** interface an IP address of **192.168.1.254** and it's **G0/1** interface an IP address of **192.168.2.254**.

<u>This time:</u> If **PC1** sends a frame to the broadcast MAC address of all **FFFF**'s, **SW1** will receive the frame and it floods it out of all its interfaces except the one it was received on. So, it sends the frame out of **G0/1** and **G0/2**, and **PC2** and **R1** receive the frame. However, that's where it ends! The broadcast is *limited* to the Local network, it doesn't cross the Router and go to **SW2**, **PC3** and **PC4**.

IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags				Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

This is a chart from Wikipedia showing the **IPv4** header. **IP**, or, **Internet Protocol**, is the primary Layer3 protocol in use today and version 4 is the version in use in most networks.

→ In this Day, we'll focus on 2 fields, the Source and Destination IP address:

→ These fields are both **4 bytes** (**32** bits) in length.

Let's take a look at this IP address (**192.168.1.254**):
An IP address is 32 bits long, so each of these four groups of numbers represents **8** bits.
192 represents **8** bits, **168** represents **8** bits, **1** represents **8** bits, **254** represents **8** bits.
If we write these **8** bits out as **1**'s and **0**'s, **192** → **11000000**, **168** → **10101000**, **1** → **00000001**, **254** → **11111110**.
This way of writing with just **0** and **1** is called ***binary***, and it is difficult to read and understand!
So, IP address are written with what's called *dotted decimal*, because there are for decimal number, **192**, **168**, **1** and **254** separated by dots or *periods.*

→ Let's learn about *binary:*
A quick review about *Decimal* and *Hexadecimal* from Minute **8:26** to Minute **10:20**.

Binary System: Binary is (***base 2***), meaning each digit increases by a factor of **2**, it doubles.
So, that means that **192** which is equal in binary **11000000** is really:

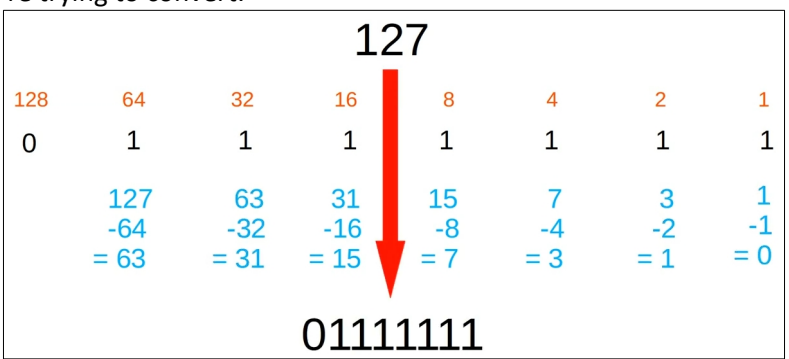
1	1	0	0	0	0	0	0
1 * 128	**1 * 64**	0 * 32	0 * 16	0 * 8	0 * 4	0 * 2	0 * 1

128 + 64 = 192

And so on for the rest of the 3 group numbers, **168**, **1** and **254.**
This is the preferred way to convert the binary ***octets*** to *decimal*.

Note: You'll often hear each of these **8** bits groups referred to as '***octets***'.

→ Let's try converting in the opposite, from *decimal* to *binary*:
To do so, it is recommended, to write out the values of each bit in a binary octet, like the picture.
Then try to subtract each number from the decimal number you're trying to convert.
And if it subtracts well, then we add a **1** under that unit,
f we can't subtract we write a **0** under that unit.

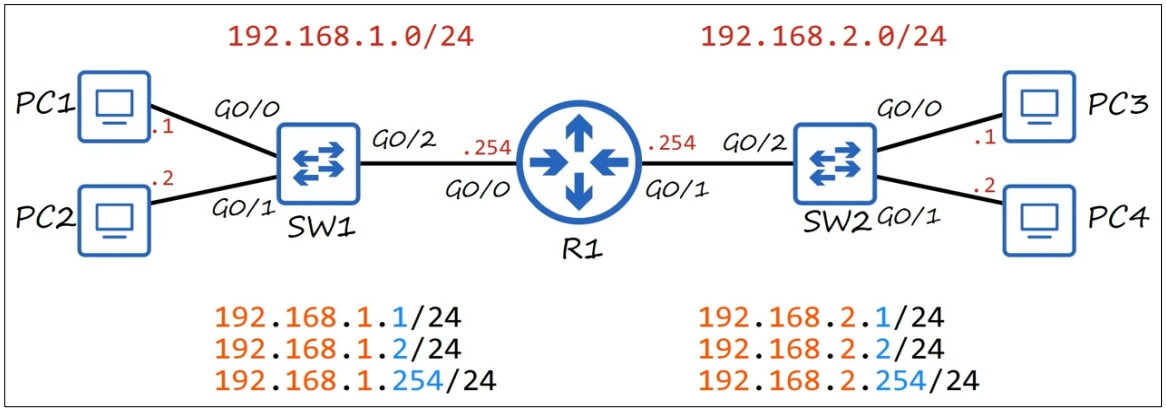


NOTE: The example of converting **221** from decimal to binary in the video is ***incorrect***!! The right value is: **11011101**.

The range of possible numbers that can be represented with **8** binary bits, ranges from **0**, if all bits are zero, to **255**, if all bits are ones. Because **128** + **64** + **32** + **16** + **8** + **4** + **2** + **1** = **255**.

So, and **IPv4** address is really a series of **32** bits. It is split up into ***4 octets***, and then written in ***dotted decimal*** format to make it simpler for us to read and understand.
You also remember there was **/24**, that is used to *identify* which part of the IP address represents the network and which represents the end host.
Since an IP address is **32** bits, you can guess what this **/24** means?
It means that the first **24** bits of this IP address represents the network portion of the address, and the remaining **8** bits represent the end host.
So, the first **24** bits is equal to = the first **3** octets, because 8 + 8 + 8 = 24.
So, **192.168.1** is the network portion of the address, and **254** is the host portion.

If we look at the network from the previous slide, we will see that **192.168.1** and **192.168.2** are the network portion of both the two separate LANs, and the host portion is what changed. You will notice that the host portion changed to the hosts connected together in the LAN.



Another example: **154.78.111.32/16**, now which 16 bits is the network portion and which is the host portion?
Well, what comes after the Slash **/**, we start it from the beginning, which means here the first half of the IP address.
So, **154.78** is the network potion and **111.32** is the host portion.

Another example: **12.128.251.23/8**, now the first **8** bits (**12**) is the network portion, and the last **24** is the host portion (**128.251.23**).

→ IPv4 Address Classes:

→ **IPv4** addresses are split up into **5** different '***classes***'.
→ The class of an **IPv4** address is determined by the ***first octet*** of the address.
→ However the slide, the classes of address we will be focusing on are **A**, **B** and **C**.
→ Addresses in class **D** are reserved for '***multicast***' addresses.
→ Multicast is another type of address, separate from unicast and broadcast
→ Class **E** addresses are reserved for experimental uses.
→ The end of the class **A** range is usually considered to be **126**, not **127**.

Class	First octet	First octet numeric range
A	0xxxxxxx	0-127
B	10xxxxxx	128-191
C	110xxxxx	192-223
D	1110xxxx	224-239
E	1111xxxx	240-255

→ Loopback Addresses:
→ The **127** range is reserved for '***loopback addresses***'.
→ The range is anything with a first octet of **127**, so **127.0.0.0** to **127.255.255.255**.
→ These addresses are used to test the '***network stack***' (think **OSI**, **TCP/IP model**) on the local device.
→ If a device sends any network traffic to an address in this range, it is simply processed back up the **TCP/IP stack** as if there were traffic received from another device.

In this image, I pinged **127.0.0.1** on my Windows PC, and you can see that your own PC is responding to its own pings:

```
C:\Users\hp>ping 127.0.0.1

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

In this image, I sent a ping to a random IP address in the **127** range, **127.214.15.10**, and again My PC responds back to its pings:

```
C:\Users\hp>ping 127.214.15.10

Pinging 127.214.15.10 with 32 bytes of data:
Reply from 127.214.15.10: bytes=32 time<1ms TTL=128
Reply from 127.214.15.10: bytes=32 time<1ms TTL=128
Reply from 127.214.15.10: bytes=32 time<1ms TTL=128
Reply from 127.214.15.10: bytes=32 time<1ms TTL=128

Ping statistics for 127.214.15.10:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Notice the round trip times, all **0ms** milliseconds.
That's because the traffic isn't going anywhere, my PC just sending and receiving these pings to and from itself.

Now, here's the chart again, with a column added, **Prefix Length**.
Prefix Length is used to identify the length of the network portion of the address.

Class	First octet	First octet numeric range	Prefix Length
A	0xxxxxxx	0-127	/8
B	10xxxxxx	128-191	/16
C	110xxxxx	192-223	/24

Actually, if you look back at these three addresses we used in our examples, you can see that there Prefix Length matches the table:

Class A: 12.128.251.23/8

Class B: 154.78.111.32/16

Class C: 192.168.1.254/24

→ In class **A** there fewer possible network addresses, however, because the host portion is very long, there can be many hosts on each network.
→ Class **C** is the opposite, there are many more possible networks, but because the host portion is smaller, there are fewer hosts on each network.

Class	Leading bits	Size of *network number* bit field	Size of *rest* bit field	Number of networks	Addresses per network
Class A	0	8	24	128 (2^7)	16,777,216 (2^{24})
Class B	10	16	16	16,384 (2^{14})	65,536 (2^{16})
Class C	110	24	8	2,097,152 (2^{21})	256 (2^8)

The 'Leading bits' column refers to the first bits of the first octet.
The 'Size of network number bit' field displays the number of bits in the network portion of the IP address.
'Number of networks' displays the number of possible networks in each class.

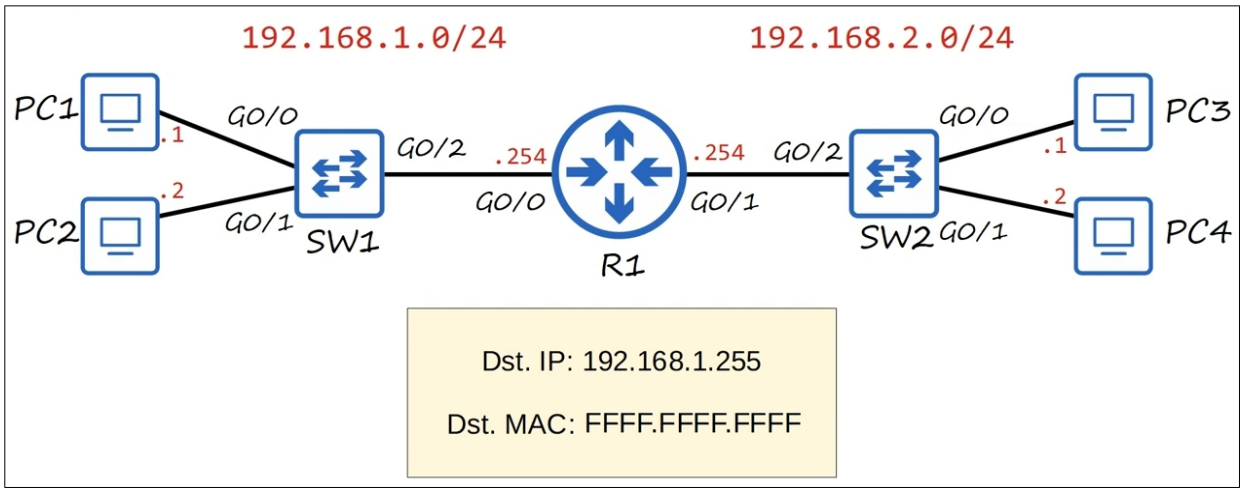
However, because the first address in each network is the network address, it can't be assigned to hosts!
Also, the last address of the network is the ***broadcast*** address, the Layer3 used when you want to send traffic to all hosts, and it also can't be assigned to the hosts.
So, really the host count is two less!

→ Netmask:

There is another way of writing these prefix length. Using a slash, followed by the length of the prefix, is a newer and easier way of writing the prefix length.
On *Juniper* network devices, you write prefix length using this slash notation.
However, *Cisco* devices still use an older, slightly more complicated way of writing the prefix length.
That is using a *dotted decimal netmask*.
A ***netmask*** is written in dotted decimal like an IP address, where the network portion is all **1**s and the host portion is all **0**s
For example, the ***netmask*** of class **A** address is **255.0.0.0**. That is the dotted decimal version of **11111111** followed by **24** zeros.
So, the ***netmask*** of class **B** address is **255.255.0.0**, it's the dotted decimal version of **11111111 11111111** followed by **16** zeros.
Netmask of class **C** address is **255.255.255.0**, it's the dotted decimal version of **11111111 11111111 11111111** followed by **8** zeros.
So these ***prefix length*** and the ***netmasks*** are the same thing, just written in different ways.

→ Two more types of IP addresses: The *network* and *broadcast* addresses:

→ If the host portion of an IP address is all **0**s, it means it is the ***Network address***.
→ The ***Network address*** is the *identifier* of the network itself.
→ In the last network diagram, you can see **192.168.1.0/24**. We know that **/24** means the first three octets are the network portion, and the last octet is the host portion.
→ Since the host portion is ***zero***, it means the last octet, the host portion, is all ***zero***s. Therefore, this address **192.168.1.0/24**, is the network address.
→ *Keep in mind*, The ***Network address*** can not be assigned to a host!
→ The ***Network address*** is the first address of the network, but the first ***usable*** address is *one* above the ***Network address***.
In this case, it is **192.168.1.1**, which is assigned to **PC1**.
→ However the last address in the network, with a host portion of all **1**s, is the ***broadcast address*** for the network.
→ Like the *network address*, the ***broadcast address*** can not be assigned to a host.
→ Although it is the last address in the network, the last ***usable*** address is actually *one* under the broadcast address.
In this case, that's **192.168.1.254**, the address assigned to **R1**'s **G0/0** interface.
→ So, the ***broadcast address*** is the Layer3 address used to send a packet to all hosts on the local network.
→ If a packet is sent with this destination IP address, **192.168.1.255**, what do you think will be the destination MAC address of the frame it is encapsulated in? → My answer is, the destination MAC address will be **FFFF.FFFF.FFFF**.
→ I am right! It will be all **FFFF**'s, the ***broadcast MAC address.***
→ If **PC1**,for example, sent a ping to **192.168.1.255**, It would be received by **PC2** and **R1**'s **G0/0** interface!


192.168.1.0/24
192.168.2.0/24
PC1
.1
Gi0/0
.2
PC2
Gi0/1
SW1
Gi0/2
.254
Gi0/0
R1
.254
Gi0/1
Gi0/2
SW2
Gi0/0
.1
Gi0/1
.2
PC3
PC4
Dst. IP: 192.168.1.255
Dst. MAC: FFFF.FFFF.FFFF

Day 8: IPv4 addressing Pt.2 :-

→ Some Review and Clarification:

→ We know that **127** range is reserved for the loopback, so it not considered a part of Class **A** range.
→ However, the **0** range is also reserved in Class **A**, so some might say class **A** really begins at **1**, making the range **1 – 126**. Different sources say different things, so it is recommended to remember the Class **A** as **0 – 127**.

→ **How to calculate the maximum number of *usable* addresses:-** that can be assigned to hosts:
Let's take this Class **C** network **192.168.1.0/24**, it uses a **/24** prefix length as it's Class **C**. and therefore the last octet, last **8** bits, are the host portion.
→ That means that the host portion can be **0** to **255**. Which gives **256** addresses as a total.

Host portion = **8** bits = $\mathbf{2^8}$ = **256**.
There are 2 types of addresses, the first is when the host portion is all *zero*s, it will be the *network* address (*Network ID*).
The second is when all are *one*s, it will be the *broadcast* address. Both of them can not be assigned to the host.

→ The maximum number of hosts per network is = $2^8 - 2 = 254$ hosts.

Let's take a look at this Class **B** network, **172.16.0.0/16**
→ The *broadcast* address is **172.16.255.255**, and the *network* address is **172.16.0.0**
→ The host portion is = $\mathbf{2^{16}}$ = **65**,**536**
→ Maximum hosts per network = $\mathbf{2^{16} - 2}$ = **65**,**534** hosts.

Let's take one more example on this Class **A** network, **10.0.0.0/8**:
→ The *broadcast* address is **10.255.255.255**, and the *network* address is **10.0.0.0**.
→ The host portion is = $\mathbf{2^{24}}$ **= 16**,**777**,**216**
→ Maximum hosts per network = $\mathbf{2^{24} - 2}$ = **16**,**777**,**214**.

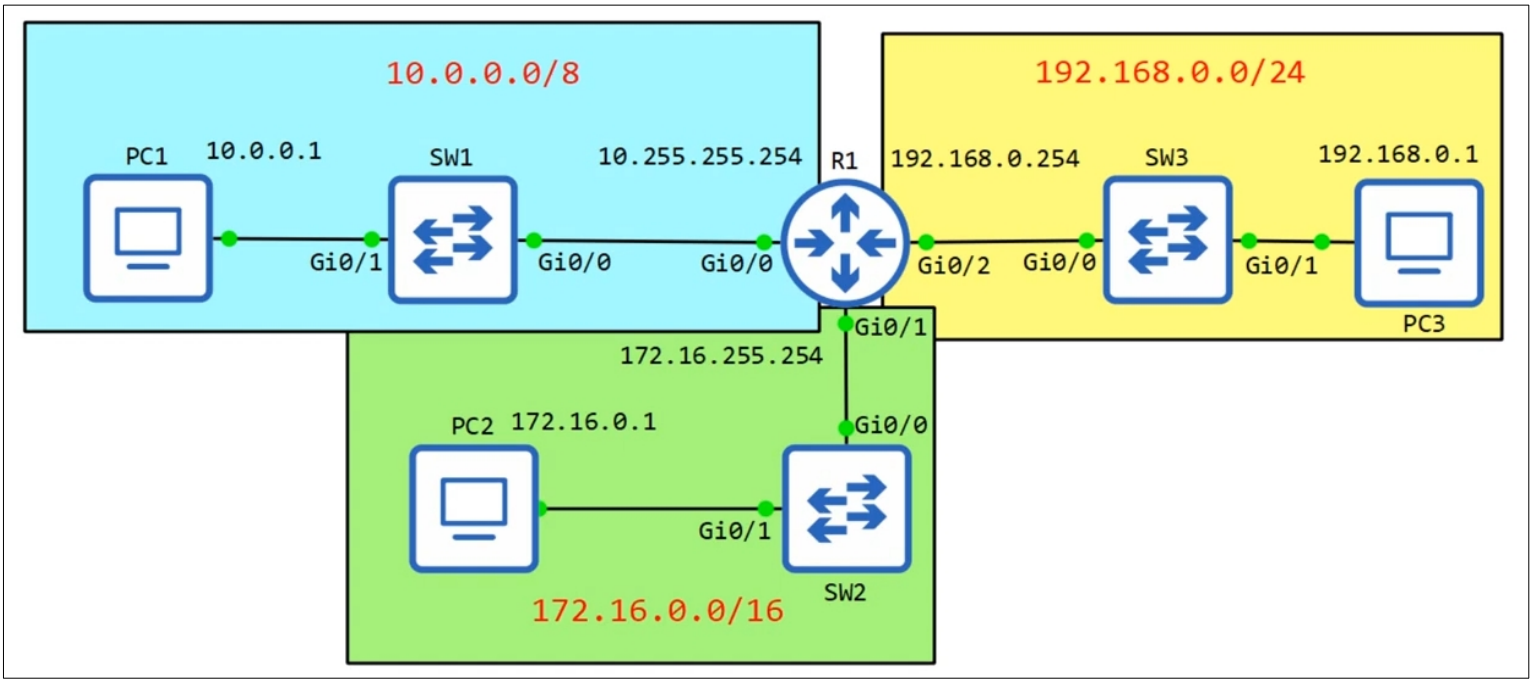
→ The formula for determining the number of hosts on a network is :
$\mathbf{2^n - 2}$, while the **n** = number of host bits.

→ **First/Last Usable Address:**
First Usable address, is the one after the *network* address!
Last Usable address, is the one before the *broadcast* address!
→ So, let's get them to the network, **192.168.1.0/24**:
The *network* address is the address with the host portion of all *zeros*, which is **192.168.1.0**, and the *broadcast* address is the address with the host portion of all *ones*, which is **192.168.1.255**,
As the previous, ***first usable*** address is **192.168.1.1**, and ***last usable*** address is **192.168.1.254**.

→ So, let's get them to the network, **172.16.0.0/16:**
The *network* address is the address with the host portion of all *zeros*, which is **172.16.0.0**, and the *broadcast* address is the address with the host portion of all *ones*, which is **172.16.255.255**,
As the previous, ***first usable*** address is **172.16.0.1**, and ***last usable*** address is **172.16.255.254**.

→ So, let's get them to the network, **10.0.0.0/8:**
The *network* address is the address with the host portion of all *zeros*, which is **10.0.0.0**, and the *broadcast* address is the address with the host portion of all *ones*, which is **10.255.255.255**,
As the previous, ***first usable*** address is **10.0.0.1**, and ***last usable*** address is **10.255.255.254**.

→ IPv4 Addressing:



→ Let's go into the CLI of **R1** and make the configurations:

```
R1>en
R1#show ip interface brief
Interface                  IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0         unassigned      YES unset  administratively down down
GigabitEthernet0/1         unassigned      YES unset  administratively down down
GigabitEthernet0/2         unassigned      YES unset  administratively down down
GigabitEthernet0/3         unassigned      YES unset  administratively down down
R1#
```

→ We used **en**, the shortcut of the **`enable`** command, to enter privileged exec mode.
The second command (**`show ip interface brief`**), we use it to confirm the status of each interface on the devices, as well as their IP addresses.
→ The **Interface** column, lists the interfaces on the device.
→ The **IP-Address** column, lists the IP of each interface. They're all unassigned yet.
→ **OK?** column, a *legacy* feature of the command, It's not relevant any more. Basically, It says whether or not the IP address is valid or not. On modern devices, the device won't let you assign invalid IP addresses, So, You should never see **NO** in the column.
→ The **Method** column, indicates the method by which the interface was assigned an IP address. Currently it Is *unset*.
→ The **Status** column, You can consider this the Layer1 status of the interface. If the interface is enabled, there is a cable connected, and the other end of the cable is properly connected to another device, you should see **up** here, not **down**. However, here it displays **administratively down**. This means the interface has been disabled with the '**shutdown**' command. AS we haven't done any configurations yet, this is the *default status* of Cisco router interfaces!
NOTE1: Cisco Switch interfaces are *not* ***administratively down*** by default.
They will be **up** if connected to another device or **down** if not.
NOTE2: Even though **GigabitEthernet0/0**, **0/1** and **0/2** on this router are connected to Switches, the Interfaces remain ***administratively down***, because the '**shutdown**' command is applied to them by default.
→ The final field is '***Protocol***', while the **Status** column refers to the Layer1 status of the interface, this '***Protocol***', is the Layer2 status. Because the interfaces are down at Layer1, Layer2 can not operate, So all of these interfaces are down at Layer2.
You' ll never see an interface are *down* at the **Status** column and *up* in the **Protocol** column, although the reverse is *possible*.
-- Remember: the **Status** refers to Layer1 Status, for example, is the interface is shutdown, is there a cable attached, etc.
Protocol refers to Layer2 Status, for example, is Ethernet functioning properly between this device and the one connected to it.

→ Configurations:

→ Let's configure the **GigabitEthernet0/0** first:

```
R1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#interface gigabitethernet 0/0
R1(config-if)#
```

We use the command (**`conf t`**), the shortcut of the 'configure terminal' to enter *Global Config Mode*.
Next, to configure the interface itself, We have to enter *Interface Config Mode.* So we use the command (**interface**) followed by the name of interface, *gigabitethernet 0/0.*
And now, as you can see, it displays (**config-if**) beside the hostname of the device.
→
Some ways to enter the **Configuration Mode:**
Minute **13:55** to Minute **15:32**.
→
→ Let's set the IP Address:
That is done with the command (**`ip address`**) and followed by the IP address you wanna set.

```
R1(config-if)#ip address 10.255.255.254 ?
  A.B.C.D  IP subnet mask

R1(config-if)#ip address 10.255.255.254 255.0.0.0
R1(config-if)#no shutdown
R1(config-if)#
*Dec  7 08:29:08.937: %LINK-3-UPDOWN: Interface GigabitEthernet0/0, changed state to up
*Dec  7 08:29:09.938: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet0/
0, changed state to up
R1(config-if)#
```

We first used the *context-sensitive* help, the question mark **?**, to display the next option, and it is the '***subnet mask***'.
This is another name for the '***netmask***'.
As opposed to writing **/8** for this class **A** address, we will have to write out the *subnet mask* in dotted decimal.
-- Remember: The **/8** is equivalent to **255.0.0.0**.
Next, We entered the command (**`no shutdown`**), The command we use to *enable* the interface. As they are being shutdown by the default!
Now, that we enter the command to enable the interface, we got 2 messages on the device.
The first one says '*Interface GigabitEthernet0/0, changed state to up*'. This refers to the Physical Layer status, the '**Status**' column of the (**`show ip interface brief`**).
The second message says '*Line protocol on Interface GigabitEthernet0/0, changed state to up*'. This is the Layer2 status of the interface, the '**Protocol**' column of the (**`show ip interface brief`**) command.
Now, if we gave a look at the (**`show ip interface brief`**) command, both of those columns should display up.

```
R1(config-if)#do sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0         10.255.255.254  YES manual up                    up
GigabitEthernet0/1         unassigned      YES unset  administratively down down
GigabitEthernet0/2         unassigned      YES unset  administratively down down
GigabitEthernet0/3         unassigned      YES unset  administratively down down
R1(config-if)#
```

We used (**`do`**) to let us execute this privileged exec mode command from interface config mode.
We also used the shortcut instead of using the full command.

We can now see the IP address, the method is displayed as **manual** instead of **unset**, and both the **Status** and **Protocol** display up.
Seems Like our interface configuration was a success!

→ Configurations:

→ Let's configure the **GigabitEthernet0/1** second:
We will give it an Address of **172.16.0.0** and the prefix length will be **/16**, and the ***subnet mask*** will be **255.255.0.0**.
First we used the (**`int g0/1`**) command to enter the *Interface Configuration Mode* for the interface.
Followed by the command (**`ip add 172.16.0.0 255.255.0.0`**) and then with the command (**`no shutdown`**) or just (**`no shut`**).
And then we (**`do sh ip int br`**) to show the IP addresses of the Interfaces to be sure that every thing works fine!

```
R1(config-if)#int g0/1
R1(config-if)#ip add 172.16.255.254 255.255.0.0
R1(config-if)#no shut
R1(config-if)#
*Dec  7 08:51:42.648: %LINK-3-UPDOWN: Interface GigabitEthernet0/1, changed state to up
*Dec  7 08:51:43.649: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet0/
1, changed state to up
R1(config-if)#do sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0         10.255.255.254  YES manual up                    up
GigabitEthernet0/1         172.16.255.254  YES manual up                    up
GigabitEthernet0/2         unassigned      YES unset  administratively down down
GigabitEthernet0/3         unassigned      YES unset  administratively down down
R1(config-if)#
```

→ You can directly switch from one interface to the other by the command (**`int`** + interface name).

→ Configurations:

→ Let's configure the **GigabitEthernet0/2** finally:
Same Like the others!

```
R1(config-if)#int g0/2
R1(config-if)#ip add 192.168.0.254 255.255.255.0
R1(config-if)#no shut
R1(config-if)#
*Dec  7 09:05:41.505: %LINK-3-UPDOWN: Interface GigabitEthernet0/2, changed state to up
R1(config-if)#
*Dec  7 09:05:42.505: %LINEPROTO-5-UPDOWN: Line protocol on Interface GigabitEthernet0/
2, changed state to up
R1(config-if)#do sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0         10.255.255.254  YES manual up                    up
GigabitEthernet0/1         172.16.255.254  YES manual up                    up
GigabitEthernet0/2         192.168.0.254   YES manual up                    up
GigabitEthernet0/3         unassigned      YES unset  administratively down down
R1(config-if)#
```

→ Some more '`show`' commands:

They can be used to check out the interfaces on a Cisco device:

→ (**`show interfaces`** + interface): It shows a lot of information for each interface, This command show primarily Layer1 and Layer2 information about the interface, but also some Layer3.

```
R1#show interfaces g0/0
GigabitEthernet0/0 is up, line protocol is up
  Hardware is iGbE, address is 0c1b.8444.f000 (bia 0c1b.8444.f000)
  Internet address is 10.255.255.254/8
  MTU 1500 bytes, BW 1000000 Kbit/sec, DLY 10 usec,
     reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, loopback not set
  Keepalive set (10 sec)
  Auto Duplex, Auto Speed, link type is auto, media type is RJ45
  output flow-control is unsupported, input flow-control is unsupported
  ARP type: ARPA, ARP Timeout 04:00:00
  Last input 00:00:06, output 00:00:05, output hang never
  Last clearing of "show interface" counters never
  Input queue: 0/75/0/0 (size/max/drops/flushes); Total output drops: 0
  Queueing strategy: fifo
  Output queue: 0/40 (size/max)
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute output rate 0 bits/sec, 0 packets/sec
     167 packets input, 30159 bytes, 0 no buffer
     Received 0 broadcasts (0 IP multicasts)
     0 runts, 0 giants, 0 throttles
     0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored
     0 watchdog, 0 multicast, 0 pause input
     350 packets output, 39097 bytes, 0 underruns
     0 output errors, 0 collisions, 2 interface resets
     105 unknown protocol drops
     0 babbles, 0 late collision, 0 deferred
     1 lost carrier, 0 no carrier, 0 pause output
     0 output buffer failures, 0 output buffers swapped out
```

--NOTE: That the MAC Address is mentioned twice, that's because the **B.I.A** is the actual Physical Address of the device, but you can also configure a different MAC Address in the **CLI**, although usually you won't configure a different MAC Address.

→ (**`show interfaces description`**): like the (**`do sh ip int br`**) command, But it also has this '**Description**' column. Interface description are *optional*, but can be very helpful in identifying the purpose of each interface.

→ Let's configure descriptions on each of these interfaces:

From *Global Configuration Mode*, We enter the *Interface Configuration Mode* for **G0/0**.

Now we use the command (**`description`** + description)! Or just (**`desc`** + description)

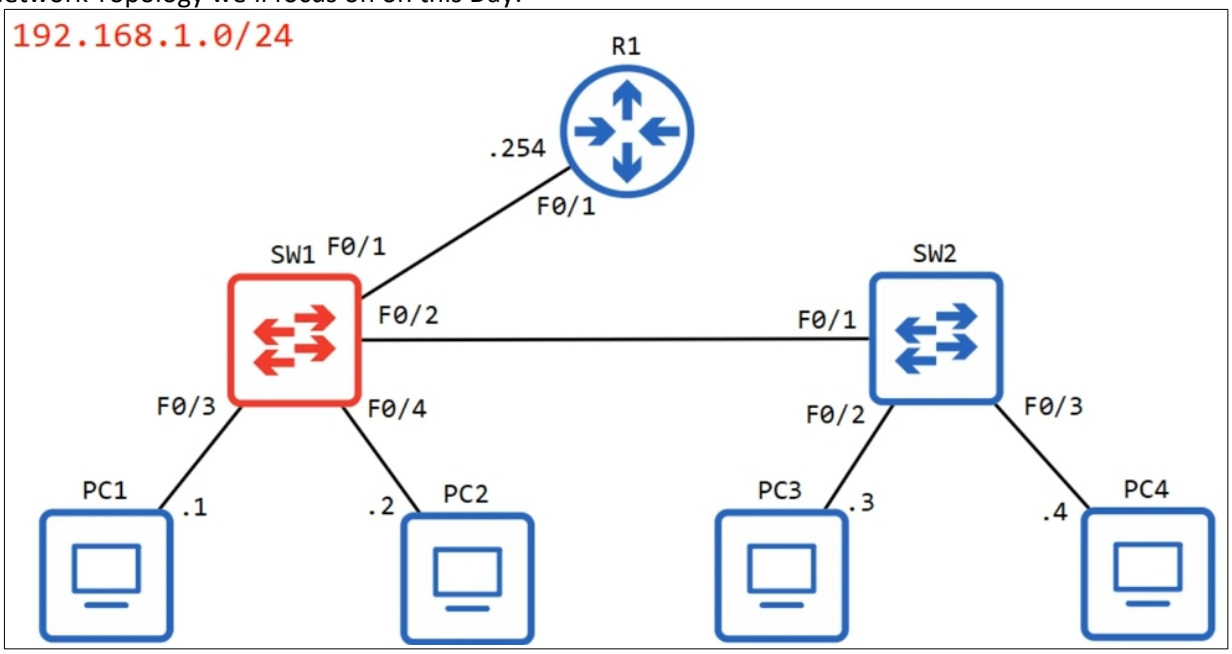
```
R1(config)#int g0/0
R1(config-if)#description ## to SW1 ##
R1(config-if)#int g0/1
R1(config-if)#desc ## to SW2 ##
R1(config-if)#int g0/2
R1(config-if)#desc ## to SW3 ##
R1(config-if)#do sh int desc
Interface                      Status         Protocol Description
Gi0/0                          up             up       ## to SW1 ##
Gi0/1                          up             up       ## to SW2 ##
Gi0/2                          up             up       ## to SW3 ##
Gi0/3                          admin down     down
```

-- Remember: There are many **show** commands that can be used, but for now just remember these three:

- **`show ip interface brief`**
- **`show interfaces`**
- **`show interfaces description`**

Day 9: Switches Interfaces :-

This is the Network Topology we'll focus on on this Day:



This is a single LAN, **192.168.1.0/24**, We are going to focus on **Switch1**, configuring its network interfaces.

→ CLI of SW1:

First, We enter the Privileged Exec Mode with the (**en**) command, and then (**sh ip int br**), we will note that there are some differences here:

```
SW1>en
SW1#sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
Vlan 1                 unassigned      YES unset  up                    up
FastEthernet0/1        unassigned      YES unset  up                    up
FastEthernet0/2        unassigned      YES unset  up                    up
FastEthernet0/3        unassigned      YES unset  up                    up
FastEthernet0/4        unassigned      YES unset  up                    up
FastEthernet0/5        unassigned      YES unset  down                  down
FastEthernet0/6        unassigned      YES unset  down                  down
FastEthernet0/7        unassigned      YES unset  down                  down
FastEthernet0/8        unassigned      YES unset  down                  down
FastEthernet0/9        unassigned      YES unset  down                  down
FastEthernet0/10       unassigned      YES unset  down                  down
FastEthernet0/11       unassigned      YES unset  down                  down
FastEthernet0/12       unassigned      YES unset  down                  down
```

As you can see, the Four Interfaces which are connected to devices have a **Status** column, which is the Layer1 status, and a **Protocol** column, which is the Layer2 status of **up**/**up**.

Without any done configurations on **SW1**, Now we can see a difference between Cisco Routers and Switches!

-- Router interfaces are in an administratively disabled state by default meaning they have the '**shutdown**' command applied.

-- Switch interfaces, however, don't have the shutdown command applied, so if you connect them to another device they will usually be in the **up**/**up** state with no configurations required.

Now, the IP address is *unassigned*, and it will remain that way because these are *Layer2 Switch-Port*, they don't need an IP address.

** → The concept of MultiLayer Switching, where you actually *do* assign IP addresses to Switches, will be for a future Lesson ← **

** → Also the VLAN interface, will be a topic for another Lesson ← **

The other interfaces aren't connected to any other devices, so their status is **down/down**.
-- The difference between **administratively down** and **down**, is because of the ***shutdown*** command!

To summarize:

> **Router** interfaces have the `shutdown` command applied by default
> =will be in the `administratively down/down` state by default
>
> **Switch** interfaces do NOT have the '`shutdown`' command applied by default
> =will be in the `up/up` state if connected to another device
> OR
> in the `down/down` state if not connected to another device

Let's Look an another useful command to check on the Switch interfaces:
That is '**`show interfaces status`**':

```
SW1#show interfaces status
Port      Name               Status       Vlan       Duplex  Speed Type
Fa0/1                        connected    1          a-full  a-100 10/100BaseTX
Fa0/2                        connected    trunk      a-full  a-100 10/100BaseTX
Fa0/3                        connected    1          a-full  a-100 10/100BaseTX
Fa0/4                        connected    1          a-full  a-100 10/100BaseTX
Fa0/5                        notconnect   1            auto   auto 10/100BaseTX
Fa0/6                        notconnect   1            auto   auto 10/100BaseTX
Fa0/7                        notconnect   1            auto   auto 10/100BaseTX
Fa0/8                        notconnect   1            auto   auto 10/100BaseTX
Fa0/9                        notconnect   1            auto   auto 10/100BaseTX
Fa0/10                       notconnect   1            auto   auto 10/100BaseTX
Fa0/11                       notconnect   1            auto   auto 10/100BaseTX
Fa0/12                       notconnect   1            auto   auto 10/100BaseTX
```

→ Let's Check every field:

- The **Port** field, simply lists each interfaces.
- The **Name** field, it's the description of the interface.
- The **Status** field, different than the '**`sh ip int br`**', The connected interfaces show a status of *connected* and the unconnected interfaces show a status of *notconnect*.
- The **Vlan** field, it can be used to divide LANs into smaller LANs. The *default* Vlan is **1**, The only one that doesn't show a **Vlan** of **1** is **F0/2**, which shows ***trunk***. Just take note of the fact that the interface connected to the other Switch, **SW2**, is a ***trunk*** interface.
- The **Duplex** field, indicates whether the device is capable of both sending and receiving data at the same time, which is known as *full-duplex*, or it is not which is called *half-duplex*. **Duplex** is ***auto*** be default on Cisco Switches, meaning it will negotiate with the neighboring device and use *full-duplex* if possible. Note that all of the unconnected interfaces have a duplex of **auto**, and the connected interfaces have a duplex of '**a-full**', **a** stands for auto, and it means that it automatically negotiated a duplex of *full* with the neighboring device
- The **Speed** field, which is also **auto** by default. These are **FastEthernet** interfaces, so they are capable of speeds up to **100** megabits per second. However, they are also capable of operating at **10** megabits per second. **Auto** means they're able to negotiate with the device they are connected to and use the fastest speed both devices are capable of. '**a-100**' means a speed of **100** megabits per second was '*auto-negotiated*' with the neighboring device.
- The **Type** field, These are all **RJ45** interfaces for copper **UTP** cables, but if they were small form-factor pluggable, or **SFP** modules, you'd see that here instead. In this case, we see **10/100BaseTX**, the **10/100** referring to the speeds at which these interfaces can operate.

→ Configuring Interface Speed and Duplex:
Auto-negotiation works well, so usually you'll leave it be, but let's go and manually configure the **Speed** and **Duplex** of an interface, **F0/1** which is connected to **R1**:

```
SW1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
SW1(config)#int f0/1
SW1(config-if)#speed ?
10                      Force 10 Mbps operation
100                     Force 100 Mbps operation
auto                    Enable AUTO speed configuration
SW1(config-if)#speed 100
SW1(config-if)#duplex ?
auto                    Enable AUTO duplex configuration
full                    Force full duplex operation
half                    Force half-duplex operation
SW1(config-if)#duplex full
SW1(config-if)#description ## to R1 ##
```

-- First, we entered the *Configuration Mode* and then used the command (**`speed ?`**) to allow options and then set it to **100** since **R1**'s interface is a FastEthernet also.
-- Then the command (**`duplex ?`**) to get the options, both **SW1** and **R1** support *full-duplex*, so we choose the *full* option.
-- Finally, we configured the description of the interface.

Okay, so when we use the command (**`show interfaces status`**) you can see both the configured **Speed** and **Duplex**:

```
SW1#sh int status
Port      Name               Status       Vlan       Duplex  Speed Type
Fa0/1     ## to R1 ##        connected    1            full    100 10/100BaseTX
Fa0/2                        connected    trunk      a-full a-100 10/100BaseTX
Fa0/3                        connected    1          a-full a-100 10/100BaseTX
```

The **Speed** is **100** rather than **a-100** and the **Duplex** is **full** rather than **a-full**, because they are not being *auto-negotiated* any more. Normally, You'll keep *auto-negotiation* **on**, but if perhaps there is some problem and it's not working, you should know how to manually configure the speed and the duplex of an interface.

Okay, Now What about the *unconnected* interfaces??
-- Although the fact that Switch interfaces are enabled by default is convenient, as you can just plug a device in and use it straight away, it can be a *Security Concern*! Really, You should disable the interfaces!
-- Fortunately, instead of having to configure each of the **8** interfaces one by one, there is a way to configure **8** interfaces at once. Here's a command that can save you a lot of time (**`interface range f0/5 – 12`**) and we then (**`shutdown`**) the interfaces.

```
SW1(config)#interface range f0/5 - 12
SW1(config-if-range)#description ## not in use ##
SW1(config-if-range)#shutdown
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/5, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/6, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/7, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/8, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/9, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/10, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/11, changed state to administratively down
00:42:36: %LINK-5-CHANGED: Interface FastEthernet0/12, changed state to administratively down
SW1(config-if-range)#
```

From *Global Config Mode*, type the command and then you entered the range of the interfaces range config mode and set them a uni-description that they are all not working and then make all of them shutdown!

The interfaces in the range don't all have to be consecutive, you can choose your own like this:

```
SW1(config)#int range f0/5 - 6, f0/9 - 12
SW1(config-if-range)#no shut
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/5, changed state to up
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/6, changed state to up
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/9, changed state to up
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/10, changed state to up
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/11, changed state to up
00:57:07: %LINK-3-UPDOWN: Interface FastEthernet0/12, changed state to up
```

Now you can see that only the interfaces in the range are enabled.

So, we shutdown the interfaces from **f0/5** to **f0/12** again, and here's the output of the show command:
We can see the description and the status is now *disabled* rather than *notconnect*.

```
SW1(config-if-range)#do sh int status
Port      Name               Status       Vlan       Duplex  Speed Type
Fa0/1     ## to R1 ##        connected    1            full    100 10/100BaseTX
Fa0/2     ## to SW2 ##       connected    trunk      a-full  a-100 10/100BaseTX
Fa0/3     ## to end hosts ## connected    1          a-full  a-100 10/100BaseTX
Fa0/4     ## to end hosts ## connected    1          a-full  a-100 10/100BaseTX
Fa0/5     ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/6     ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/7     ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/8     ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/9     ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/10    ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/11    ## not in use ##   disabled     1            auto   auto 10/100BaseTX
Fa0/12    ## not in use ##   disabled     1            auto   auto 10/100BaseTX
```

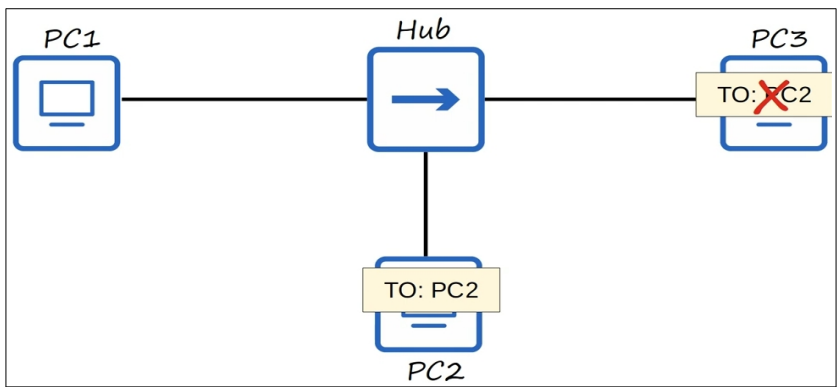
→ Full/Half Duplex:

→ **Half Duplex:** means that The device can not send and receive data at the same time. If it is receiving a frame, it must wait before sending a frame.
→ **Full Duplex:** The device can send and receive data at the same time. It doesn't have to wait. It is the *preferred* way to go!
In modern networks that use Switches, all devices can use *full-duplex* on their interfaces.

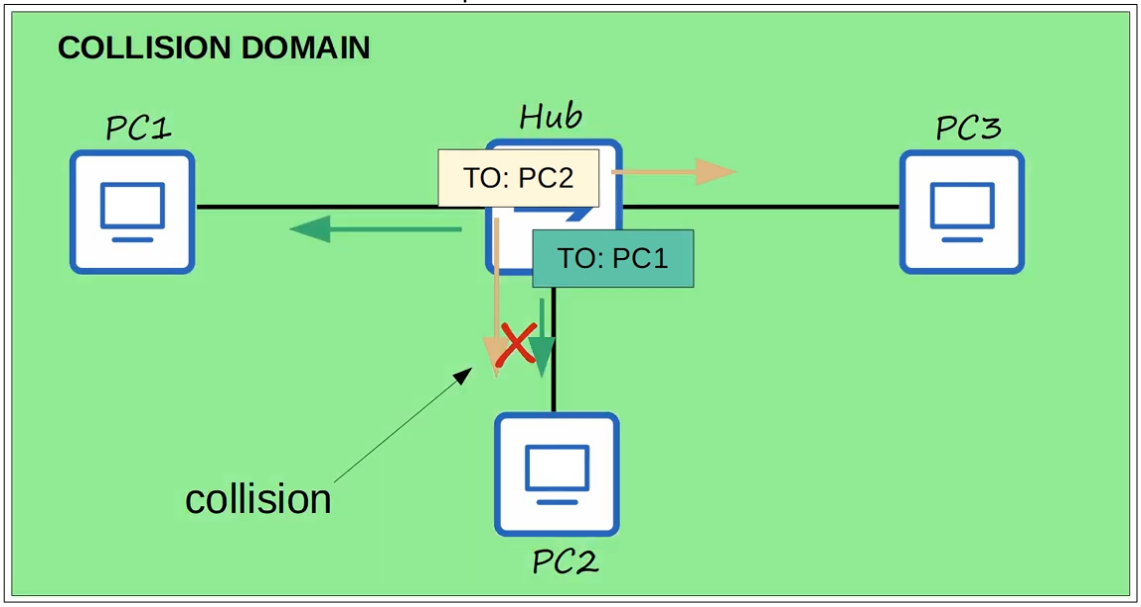
→ LAN Hubs:

So, where is *half-duplex* used? Well, in modern day networks, almost nowhere!
To understand, let's introduce an old type of network device which was around before the network Switch:

The **Hub** is much simpler than a Switch, in fact it is simply a *repeater*. Any frame it receives it *floods* like a Switch does with a *broadcast* or *unknown unicast* frame.
For example, If **PC1** wants to send a frame to **PC2**, it will send the frame out of its network interface, after the **Hub** receives it, it will *repeat* the frame out of its other interfaces, to **PC2** and **PC3**. **PC3** will recognize that the destination MAC address doesn't match with its own and ignore the frame, and **PC2** will receive it normally.



-- Now, what if two PCs tried to send a frame at the same time? In this case, **PC1** is trying to send a frame to **PC2**, and **PC3** trying to send a frame to **PC1**, They both send the frame out of their network interfaces, and this is where the problem occurs!
-- The **Hub** won't send one first and the other after, it simply tries to *flood* both at the same time, and this will result a ***collision*** on the interface, and **PC2** won't receive either frame intact.
-- All devices connected to a **Hub** are part of what's called a ***Collision Domain***.

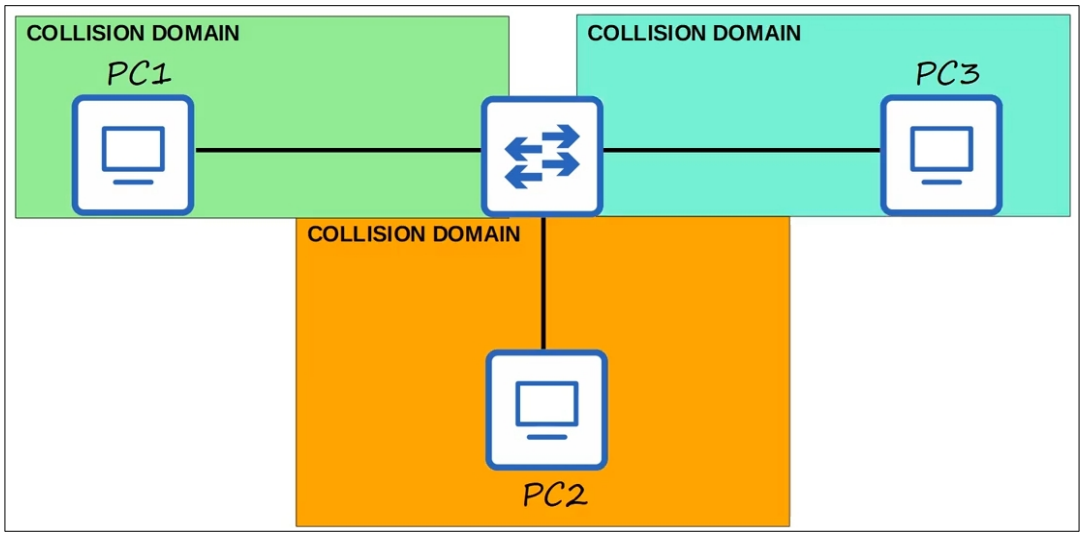


-- The frame they sent could *collide* with frames any of the other devices connected to the **Hub** send.
-- To deal with *collision* in a *half-duplex* situation like this, Ethernet devices use a mechanism called '**CSMA/CD**'.

→ CSMA/CD
-- stands for **Carrier Sense Multiple Access with Collision Detection**.
-- Describes how devices avoid collisions in a *half-duplex* situation, and how they react if collisions do occur.
-- Describes how deices using *half-duplex* listen for activity on a network segment, and then send data only when other devices aren't sending
-- Also describes how a device will react when a collision does occur.
-- How it works: Before sending frames, devices ***listen*** to the collision domain until they detect that other devices are not sending.
If a collision does occur, which can still happen because of bad timing and such, the device sends a jamming signal to inform the other devices that a collision happened!
Each device then waits a random period of time before sending frames again.
The process then *repeats*, with each device listening to check if other devices are sending frames before sending their own frames.
-- That Process works, and it was how networks operated for a long time.

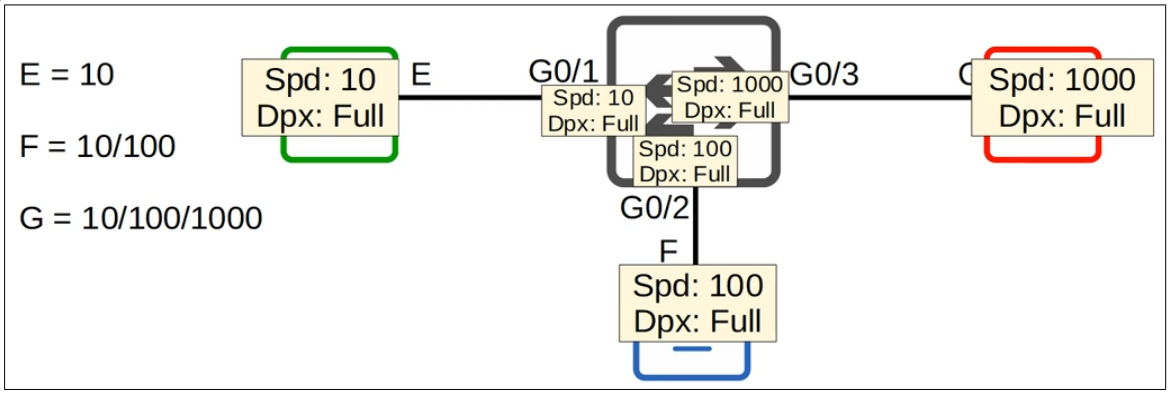
→ Switches and Hubs:
-- But Switches are more *sophisticated* than Hubs.
-- Hubs are simple repeaters which operate at Layer1, repeating whatever signals they receive. Switches operate at Layer2, using Layer2 addressing, MAC addresses, to send frames to specific hosts. They also won't send 2 frames to the host at once.
-- Devices attached to a Hub must operate in *half-duplex*.
-- Devices attached to a Switch can operate in *full-duplex*.

So the previous network, which was *one* collision domain when connected to a Hub, is now *three* collision domains.



-- Because of the improved functionality of Switches over Hubs, these devices can now operate in *full-duplex*, meaning they don't have to worry about whether or not other devices are sending data at the same time, they can send data freely.
-- Although problems like collision still **do** occur, they are usually rare and usually are a sign of a problem, like a *misconfiguration*, rather than a regular occurrence like in a *half-duplex* network.

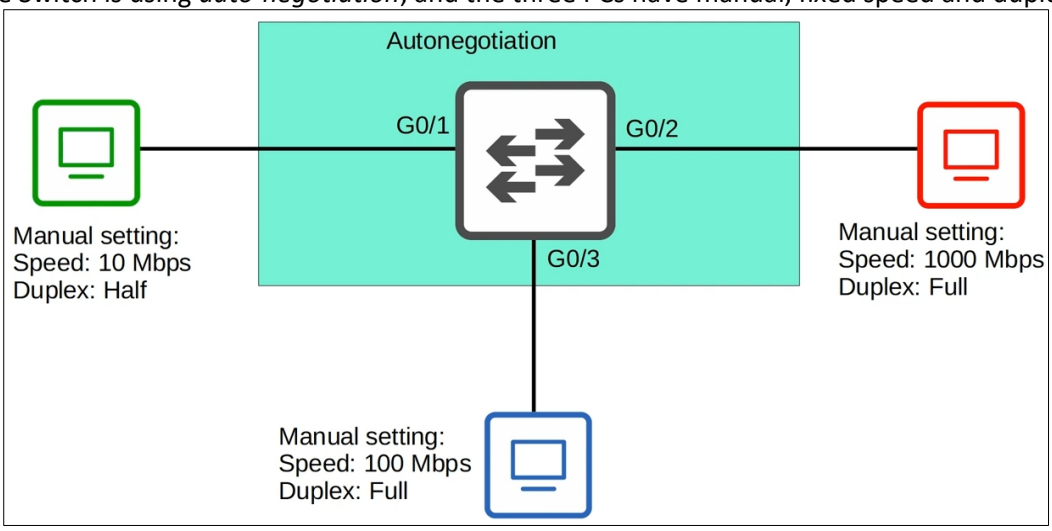
→ **Speed/Duplex *Auto-negotiation:*** This apply to both Routers and Switches:
-- Interfaces that can run at different speeds (**10**/**100** or **10**/**100**/**1000**), have default settings of **Speed** *auto* and **Duplex** *auto*.
-- Interfaces '*advertise*' their capabilities to the neighboring device, and they neogtiate the best **speed** and **duplex** settings they are both capable of.
-- <u>For example</u>:



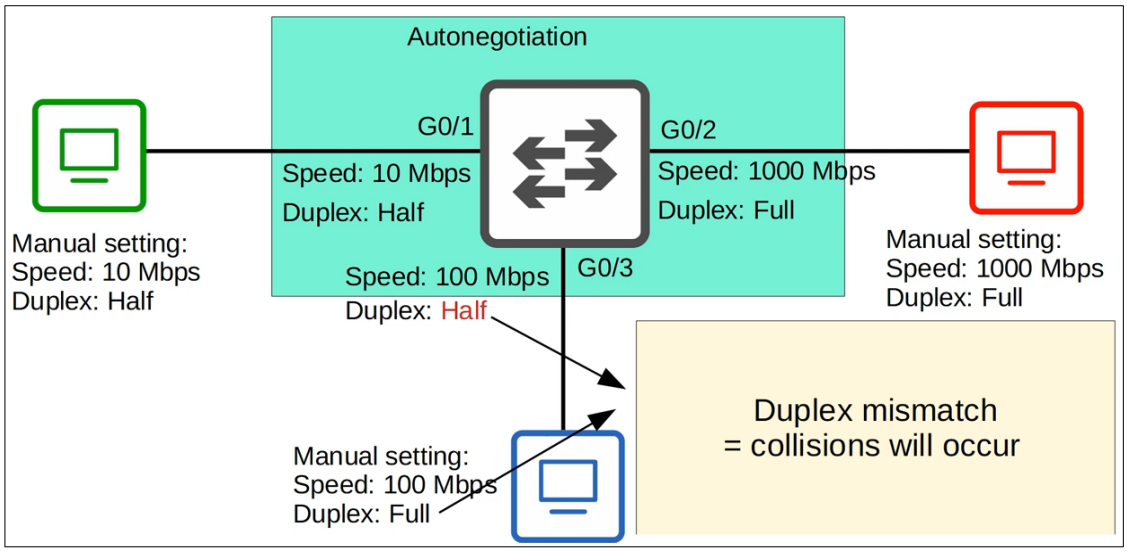
G0/1 and the PC will negotiate to a speed of **10** megabits per second and *full-duplex*.
G0/2 and the PC will negotiate to a speed of **100** megabits per second and *full-duplex*.
G0/3 and the PC will negotiate to a speed of **1000** megabits per second and *full-duplex*.
-- The PCs are all able to use the Max speed of the network interfaces, and the Switch adjusts the speed of its interfaces to match.
-- In a network like this with all PCs and Switches, there is no reason to use *half-duplex*, so they all negotiate to use *full-duplex*.

-- <u>Another situation</u>: What if *auto-negotiation* is disabled on the device connected to the Switch?
So, the Switch is trying to auto-negotiate, but the other devices don't respond. This is how the Switch reply:-
For **Speed**: The Switch will try the sense the speed that the other device is operating at. If it fails to sense, it will use the slowest supported speed (ie. **10Mbps** on a **10**/**100**/**1000** interface)
For **Duplex**: If the speed is **10** or **100 Mbps**, the Switch will use *half-duplex*.
If the speed is **1000 Mbps** or grater, it will use the *full-duplex*.

-- Let's see how this works:
In this case, only the Switch is using *auto-negotiation*, and the three PCs have manual, fixed speed and duplex settings.



We'll also assume that the Switch successfully detects the speed that the PCs are using.
-- The Switch detects the speed of Green PC and set speed to the same **10 Mbps**, and because of that it sets duplex to *half-duplex*.
-- The Switch senses the Red PC using **1000 Mbps**, so it uses the same. And because of that it uses *full-duplex*.-
-- How about the Blue PC connected to **G0/3**? The Switch senses the speed of **100 Mbps**, but then what about the duplex??
The Blue PC is using *full-duplex*, but without *auto-negotiation* the Switch can't sense that!
So, because the speed is **100 Mbps**, the Switch uses *half-duplex*.
And this results in a *duplex mismatch*!, which will cause collisions to occur, resulting in poor network performance.
So, really you should be using *auto-negotiation* on all devices in the network.



Quiz-Note: The *full-duplex* is unaware of that the *half-duplex* is unable to send and receive data ate the same time, and will send data even if the *half-duplex* side isn't ready to receive it, causing *collisions*!

→ Interface Errors:

Let's take a look at some of the errors that can show up on interfaces that otherwise seem to be working:
The Switch will take count of some of these things and you can view them with the command (**show interfaces** + interface).
We took a look at the command when we were talking about Routers, so these things aren't specific of Switch interfaces!
-- This time, let's focus on some of these statistics at the bottom:

```
SW1#show interfaces f0/2
FastEthernet0/2 is up, line protocol is up
  Hardware is Fast Ethernet, address is 000C.3168.8461 (bia 000C.3168.8461)
  Description: ## to SW2 ##
  MTU 1500 bytes, BW 100000 Kbit, DLY 100 usec,
     reliability 255/255, txload 1/255, rxload 1/255
  Auto-duplex, Auto-speed
  Encapsulation ARPA, loopback not set
  ARP type: ARPA, ARP Timeout 04:00:00
  Last input 02:29:44, output never, output hang never
  Last clearing of "show interface" counters never
  Input queue: 0/75/0/0 (size/max/drops/flushes); Total output drops: 0
  Queuing strategy: fifo
  Output queue :0/40 (size/max)
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute output rate 0 bits/sec, 0 packets/sec
     269 packets input, 71059 bytes, 0 no buffer
     Received 6 broadcasts, 0 runts, 0 giants, 0 throttles
     0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored
     7290 packets output, 429075 bytes, 0 underruns
     0 output errors, 3 interface resets
     0 output buffer failures, 0 output buffers swapped out
```

There is a lot of different kinds of counters shown here and you don't have to know all of them at this point.
So, let's just focus on some:

- First up, not errors, but you can see the total number of packets received on the interface and the total number of bytes in those packets.
- **Runts**, Which is frames that are smaller than the minimum Ethernet frame size (**64** bytes).
- **Giants**, Frames that are larger than the maximum Ethernet frame size (**1518** bytes).
- **CRC**, which counts frames that failed their CRC Check (in the Ethernet **FCS trailer**).
- **Frame**, Which counts frames that have an incorrect or illegal format (due to an error).
- **Input errors**, Total of various counters, including the above four counters.
- **Output errors**, counts frames the Switch tried to send, but failed due to an error.

```
269 packets input, 71059 bytes, 0 no buffer
Received 6 broadcasts, 0 runts, 0 giants, 0 throttles
0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored
7290 packets output, 429075 bytes, 0 underruns
0 output errors, 3 interface resets
0 output buffer failures, 0 output buffers swapped out
```

Keep in mind, these counters are shown on the Switch, but they are the same on a Router.

Day 10: IPv4 Header :-

-- The IP is used in the Layer3, to help send the data between devices in separate networks, even on other sides of the world over the Internet, this is known as ***Routing***.

We will focus exclusively on the fields in the **IPv4 Header**.
Or just the Layer3 Header, as it contains the information that is needed to route this packet to its destination.

Since we'll be focusing on Layer3, We will usually be using the term ***Packet***, talking about ***routing packets***, rather than ***frames***.

→ IPv4 Header:

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

To read this chart, it's read *left-to-right*, *top-to-bottom*.

→ **Version** field:

- It is **4** bits in length, half of one *octet*.
- It's purpose is *straight forward*. It identifies the version of IP used. There are only **2** versions of IP in use, **IPv4** and **IPv6**.
- **IPv4**, which is simply identified with a value of **4** in this field, or **0100** in binary.
- **IPv6**, which is identified with a value of **6**, or **0110** in binary.
- Since we're focusing on the **IPv4 Header**, this value will be always **4**, since the **IPv6** has a different structure.

→ **Internet Header Length**, or **IHL** field:

- It is also **4** bits in length, half of one *octet*.
- The final field of the **IPv4 Header**, **Options**, is *variable* in length, so this field is necessary to indicate the total length of the header.
- This field specifies the length of the Header ***in 4-byte increments***. For example, if the value in this field is **5**, *Value of **5** = **5 x 4**-byte = **20** bytes*, meaning the length of the Header is ***20 bytes***.
- Minimum value in this field is **5**, which is equal to **= 20** bytes. That's the length of the **IP** Packet without any **IP** options at the end, so an empty options field.
- Maximum value is **15**, which is the maximum value of **4** bits. **15** (**15** x **4**-bytes = **60** bytes).
- As you can see the value of the **4** bits are **1**, **2**, **4** and **8**, adding them up results in **15**.
- That means the maximum length of the **IP Options** field is, **40 bytes**.
- AN **IPv4 Header** with no **Options** field, is **20 bytes** in length, and that's the minimum length of an **IPv4 Header**.
- AN **IPv4 Header** with a maximum length **Options** field, **40 bytes**, has a length of **60 bytes**. And that's the maximum length of an **IPv4 Header**.
- It ***Only*** indicates the length of the **IPv4 Header** itself.

1	1	1	1	
8	+ 4	+ 2	+ 1	= 15

→ **DSCP** field:

- **DSCP**, stands for '***Differentiated Services Code Point***'.
- It's length is **6** bits.
- This field is used for **Qos** (**Quality of Service**).
- It is used to prioritize delay-sensitive data, such as streaming voice and video etc. If you're loading a web page and the Internet is a little slow, it is not a big deal. However, if you're having a Skype call and there's terrible delay, or the audio and video keep freezing, It can totally ruin the experience.
- This field is used to identify which traffic should receive priority treatment.

→ **ECN** field:

- stands for '**Explicit Congestion Notification**'.
- **2** bits in length.
- Provide *end-to-end (*between two endpoints*)* notification of network congestion without dropping **Packets**.
- Normally in a network, if the network is super busy, if there is congestion, this is *signalled* by dropping packets.
- The **ECN** field provides a way to signal a congested network without dropping packets.
- However, this is an *optional* field that requires both endpoints, as well as the underlying network infrastructure, to support it.

→ **Total Length** field:

- Its length is **16** bits, or **2** octets, or **2** bytes.
- Indicates the Total Length of the Packet (**L3 header** + **L4 segment**). → Layer4 segment includes the L4 header + Data.
- Indicates the length in **bytes** (not **4-byte** increments like the **IHL** field). So, a value of **20** in his field simply means **20** bytes.
- Minimum value of this field is **20**, meaning **20** bytes, which is equal to a minimum-size **IPv4** header with no encapsulated data.
- The maximum value is **65**,**535**, which is the maximum value of **16** binary bits, all set to **1**.

→ **Identification** field:

- Length is **16** bits.
- If a packet is fragmented due to being too large, this field is used to identify which *packet* the fragment belongs to. So it can be re-assembled again to make the original packet.
- All fragments of the same packet will have their own **IPv4 header** with the same value in this field. So they can be re-assembled later.
- Packets are fragmented if larger than the **MTU**, which stands for, **Maximum Transmission Unit**.
- **MTU** is usually **1500** bytes.
- The Maximum Payload size of an Ethernet frame is **1500** bytes, so these are related.
- Fragments are re-assembled by the receiving host.

→ **Flags** field:

- Length is **3** bits.
- It is used to control/identify fragments.
- The **3** bits are functioning like this:
    - **Bit 0**: Reserved, always set to **0** or just '**Not set**'.
    - **Bit 1**: Don't fragment (**DF** bit), If it is set to **1,** it is used to indicate a packet that should not be fragmented or you may see it '**Set**' if it is set rather than **1**.
    - **Bit 2**: More fragments (**MF** bit), It is set to **1** if there are fragments in the packet, and then set to **0** for the last fragment.
    - Also, if the packet is a whole, unfragmented packet, the **MF** bit is set to **0**, since there are no fragments.

→ **Fragment Offset** field:

- It is **13** bits in length.
- Indicates the position of the fragment within the original, unfragmented IP packet.
- Allows fragmented packets to be re-assembled even if the fragments arrive out of order. Since this field lets the receiver know the original order of the fragments.

→ **TTL, Time To Live** field:

- Length is **8** bits. You may see it have a value of **255**, the maximum **8**-bit value.
- A Router will drop a packet with a **TTL** of **0**.
- This field is used to *prevent* infinite loops. If a poor routing configuration causes a packet to be continually sent around in a loop, never reaching its intended destination, if enough traffic like that accumulates, it could causes network congestion, and eventually failure.
- This field prevents that from happening, causing looped traffic to be dropped when the **TTL** reaches **0**.
- This field was originally designed to indicate the packet's maximum lifetime in seconds.
- In practice, However, this actually indicates a '**hop count**'. Each time the packet arrives at a Router on the way to its destination, the router decreases the **TTL** by **1**, until the packet reaches its destination, or the **TTL** reaches **0** and the packet is dropped.
- The current recommend default **TTL** is **64**.
- To summarize, It is reduced by **1** at each each Router the packet passes through. If it reaches **0**, the packet is dropped.

→ **Protocol** field:

- It is **8** bits in length.
- Indicates the protocol of the encapsulated Layer4 PDU.
- Typically, this will be one of the following:
  - **TCP**, which is indicated by a value of **6**.
  - **UDP**, which is indicated by a value of **17**.
  - **ICMP**, which is indicated by a value of **1**.
  - **OSPF**, ***Open Shortest Path First***, which is indicated by a value of **89**.
  - **OSPF**, the Dynamic Routing Protocol, which allows routers to learn routes to the destinations from their neighbors, without us having to manually configure the routes.

→ **Header Checksum** field:

- **16** bits in length.
- A calculated checksum, used to check for errors in the **IPv4 Header**.
- When a Router receives a packet, it calculates the checksum of the header and compares it to the one in this field of the header.
- If the newly calculated checksum and the checksum in the **IPv4 header** do not match, it means that an error has occurred in transmission, so the Router drops the packet.
- This is used to check for errors <u>only</u> in the **IPv4 header**, not in the encapsulated data.
- IP relies on the encapsulated protocol to detect errors in the encapsulated data.
- Both **TCP** and **UDP**, the two Layer4 protocols most likely to be encapsulated, have their own checksum fields to check for/detect errors in the encapsulated data.

→ **Source** and **Destination** fields:

- Both of them is **32** bits in length, as that is the length of an **IPv4** address.
- **Source**: indicates **IPv4** address of the sender of the packet.
- **Destination**: indicates **IPv4** address of the intended receiver of the packet.

→ **Options** field:

- Is an optional field, and can be **0** bits in length if not used, or up to **320** bits, **40** bytes, in length.
- This field is rarely used, however, if the **IHL**, **Internet Header Length**, field is greater than **5**, it means that **Options** are present.
- Here is a chart showing the structure of the **Options** field:

Field	Size (bits)	Description
Copied	1	Set to 1 if the options need to be copied into all fragments of a fragmented packet.
Option Class	2	A general options category. 0 is for "control" options, and 2 is for "debugging and measurement". 1 and 3 are reserved.
Option Number	5	Specifies an option.
Option Length	8	Indicates the size of the entire option (including this field). This field may not exist for simple options.
Option Data	Variable	Option-specific data. This field may not exist for simple options.

NOTE: Just Google and read more about the **IPv4 Header.**

→ Wireshark Packet Capture:

From Minute **17:50** to Minute **24:20**.

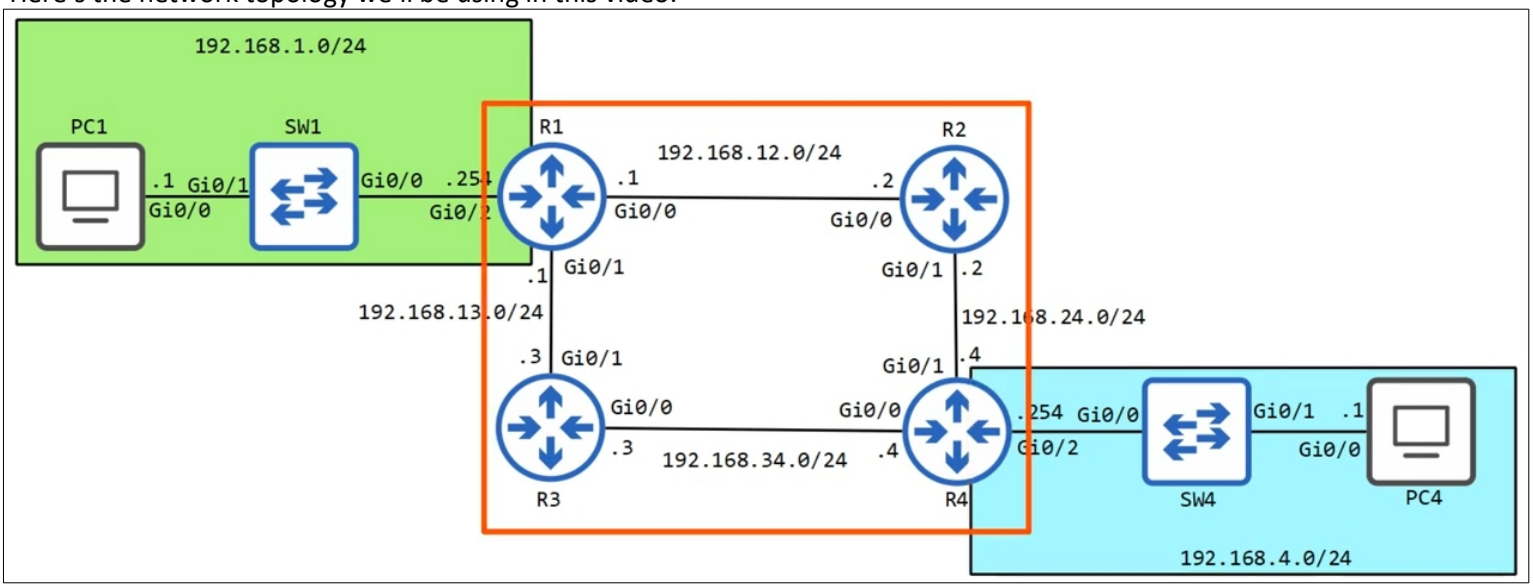
Quick Tips:

→ The **Options** field can *vary* in length from **0** bits to **320** bits.
→ The other field are *fixed-length*.
→ Although the **Total Length** and **IHL** fields are used to represent the variable length of the **IPv4 header** and packet, the fields themselves are *fixed* in length.
→ The **More Fragments** bit, part of the **Flags** field of the **IPv4 header**, is used to indicate that the current fragment is *not* the last fragment of a fragmented packet. It is set to **1** on all fragments except the last, which will set it to **0**.
→ **Fragment Offset**, is a **13**-bit field in the header, not a single bit.
→ **Don't Fragment** bit, is used to *prevent* a packet fro being fragmented.

Day 11: Static Routing :-

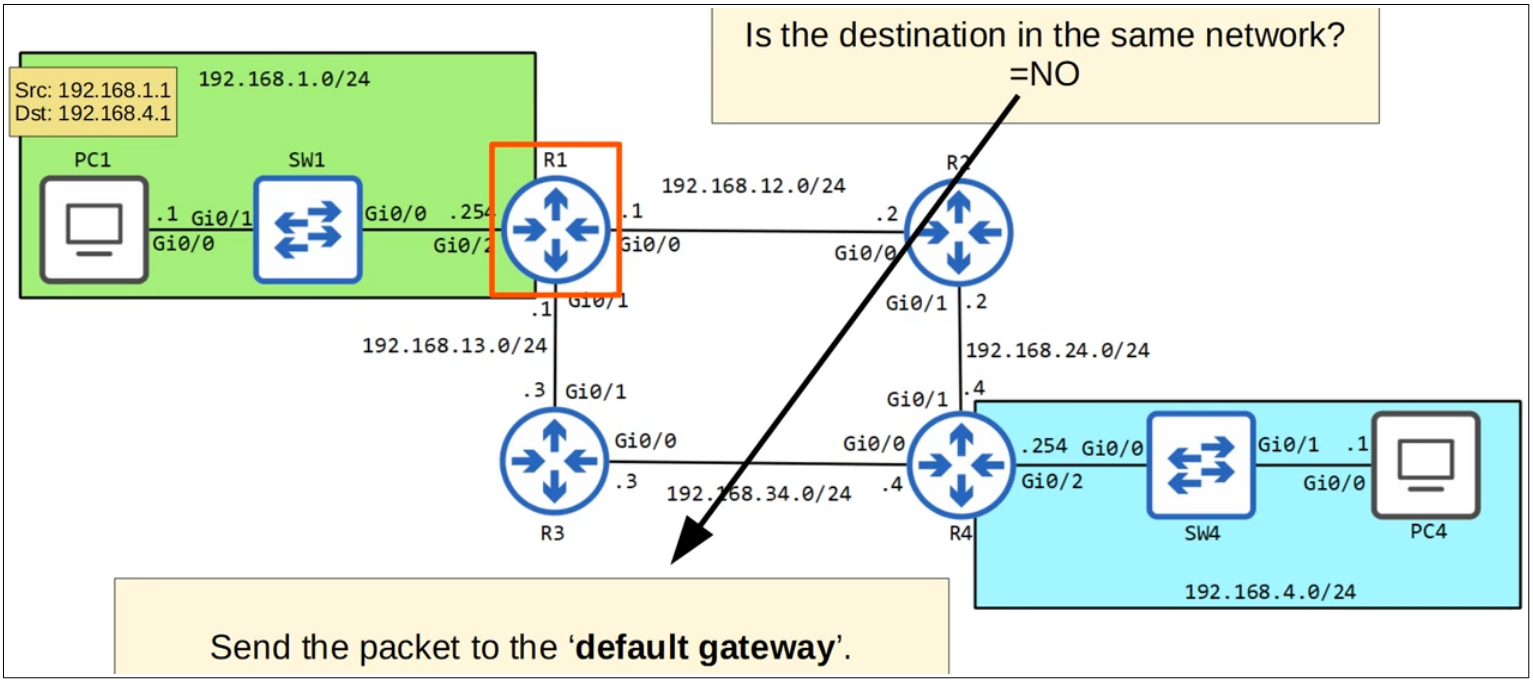
→ IP routing process:

Here's the network topology we'll be using in this video:



Here we have 2 LANs: **192.168.1.0/24** and **192.168.4.0/24**
These 4 Routers represents not a LAN, but rather a **WAN**, which stands for **Wide Area Network**.
→ **WAN** is a network that spreads/extends over a large geographical area.
These Routers could be Kilometers apart, connected by Single-Mode Fiber cable.

So, If **PC1** wants to send a packet to **PC4**, First, **PC1** asks itself "Is the destination IP address in the same network?"
Well, **PC1**'s IP address is **192.168.1.1/24**, and **/24** means that the first **3** octets are the network portion, so **192.168.1** is the network address. In the same boat, **PC4**'s IP address is **192.168.4.1/24** and so on.
The destination is in a different network, a different LAN.
So, the routing logic tells **PC1** to send the packet to the '***default gateway***'.



→ The '***default gateway***' is the device to which the host -**PC1** in this case- will forward data that is destined for another network.
R1 is the *default gateway* for **PC1**.

Routers are the devices used to connect separate networks, for example:
R1 connects the **192.168.1.0/24**, **192.168.12.0/24** and **192.168.13.0/24** networks.
→ So, to reach other networks, an end host will send the packet to its network's Router, the *default gateway*.
Now **R1** has the packet, and it is **R1**'s responsibility to forward it to the next device.
After receiving the packet, **R1** will compare the packet's destination IP address to the ***routing table***.

→ Each Router keeps a *routing table* which stores a list of destinations and how to reach them

Assuming **R1** already has an entry in its *routing table* for the **192.168.4.0/24** network, the entry will look something like this:

```
192.168.4.0/24 via 192.168.12.2, Gi0/0
```

First is the destination and the next is the '***next hop***', the next destination in the path to the **192.168.4.0/24** network, the final destination, and also the interface out of which **R1** will send the packet.
In this case, **R1** could reach the network either by sending it to **R2** or **R3**, but let's say we configured it to send it via **R2**, as you can tell by both the *next-hop* address of **192.168.12.2**, and the exit interface on **R1**, **G0/0**.

→ So, **R1** will forward the packet to the next Router on the route to the destination, which is **R2**.
Now **R2** has the packet, and it is **R2**'s responsibility to forward the packet to the next device.
R2 will follow the same process as **R1**, it will compare the packet's destination IP address to the *routing table* and then, assuming it already has an entry in its *routing table* for the destination, It might look like this:

```
192.168.4.0/24 via 192.168.14.4, Gi0/1
```

The destination is **192.168.4.0/24**, the *next-hop* address is **192.168.24.4** -the **R4**'s IP address- and the exit interface is **G0/1**, which is connected to **R4**.
After looking for the route in the *routing table*, **R2** forwards the packet to **R4**.

→ Now **R4** has the packet, and it is **R4**'s responsibility to forward the packet to the next device.
R4 will follow the same process as both **R1** and **R2**, it will compare the packet's destination IP address to the *routing table*.
In **R4**'s case, the entry in the table will look something like this:

```
192.168.4.0/24 is directly connected, Gi0/2
```

That's because **R4** has an IP address of **192.168.4.254/24** configured on its **G0/2** interface, so it knows the **192.168.4.0/24** network is directly connected to that interface.
It forwards the packet out of that interface, and **SW4** passes it on to **PC4**, the final destination.

→ Notice that we didn't mention MAC addresses, or Layer2 at all. This is just a Layer3 overview of this process.
We'll go in depth later in a video '***Life of a packet***' not just about Layer3, but Layer2 also.

→ The *routing table* on a Cisco Router:

we use the command '**`show ip route`**' from privileged exec mode, and it displays the routing table:

```
PC1#show ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       a - application route
       + - replicated route, % - next hop override, p - overrides from PfR

Gateway of last resort is not set

      192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.1.0/24 is directly connected, GigabitEthernet0/0
L        192.168.1.1/32 is directly connected, GigabitEthernet0/0
PC1#
```

Down here under the codes at the top, two routes are displayed, this may seem like 3 routes, but this line at the top, **192.168.1.0/24 is variably subntted, 2 subnets, 2 masks**, isn't a route.
This lists the classful address, and what class is **192.168.1.0/24**? It's a Class **C** address.
Underneath are the two routes that fit within that class.

Ok, so we've got 2 routes, and these two letters on the left, **C** and **L**, display the type of route. We can find them in the list up top: **L** stands for **Local**, and **C** stands for **Connected**.

→ Let's look for the **Connected** route first:
It's a route to **192.168.1.0/24**.
The IP address configured on **PC1**'s interface is **192.168.1.1/24**, so **192.168.1.0/24** is the network address, the address with a host portion of all **0**'s. **/24** means that the first three octets are the network portion and the last octet is the host portion.
Set the host portion to **0** and that's the network address. There are probably lots of other hosts on this network, maybe with addresses like **192.168.1.2/24**, **192.168.1.3/24**, **192.168.1.4/24**, etc.
All of them are part of the **192.168.1.0/24** network, because the network portion of their address is the same.
So, **PC1** knows it can reach any host on the **192.168.1.0/24** network via its network interface which is **GigabitEthernet0/0** on this Cisco Router which we're using to simulate a PC.

→ Let's look for the **Local** route second:
Look at this one, **192.168.1.1/32**, which is the exact address on **PC1**'s interface, but this time with a **/32** mask.
/32 would mean that all four octets are the network portion, and there is no host portion.
You could say this is a route to the **192.168.1.1/32** network, and there is only one address in the network, **192.168.1.1**.
This is how you identify a single, specific address, by using a **/32** mask.

→ So, to recap, the **Connected** route indicates the network to which the interface is attached.
The **Local** route indicates the actual address configured on the interface, with a **/32** mask.

Connected route = the network the interface is connected to.
Local route = the actual IP address on the interface (with a /32 mask)

So, that means you already know how to configure two types of routes on a Cisco Router. If you configure an IP address on an interface, the **Connected** and **Local** routes of that interface are added!

→ Configuring a *Default Route*:

However, this **PC1** doesn't have a route to the intended destination, **192.168.4.1**.
Actually, this is not a problem, An end-host like a PC doesn't need to know the route to every destination.
All it needs is a **default-gateway** to which it can send any traffic destined for a location outside of the local network.

On a Cisco Router, the *default-gateway* is known as the '***Gateway of last resort***', and as it says here, it is not set yet.

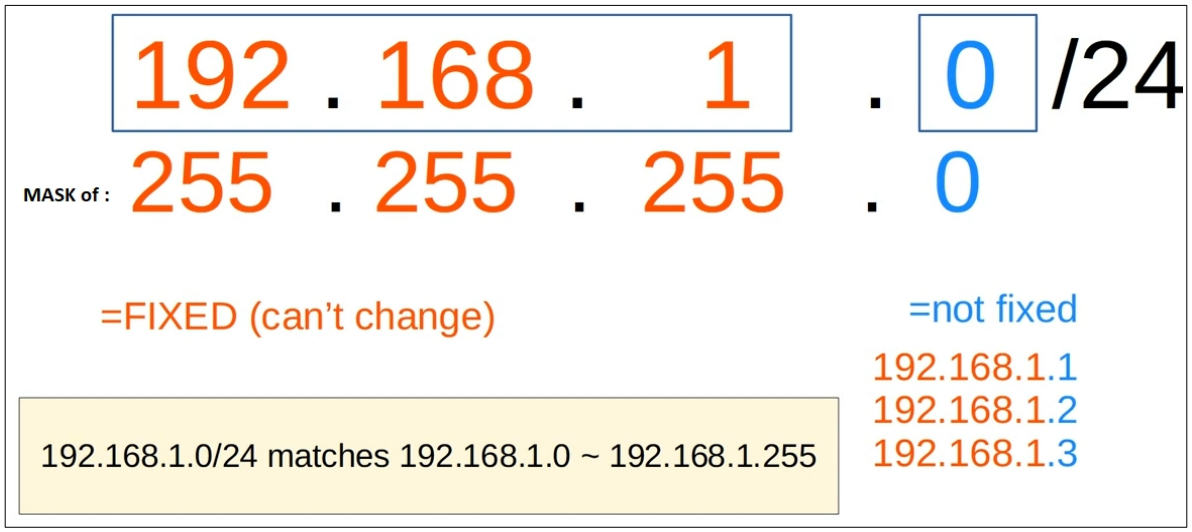
```
Gateway of last resort is not set
```

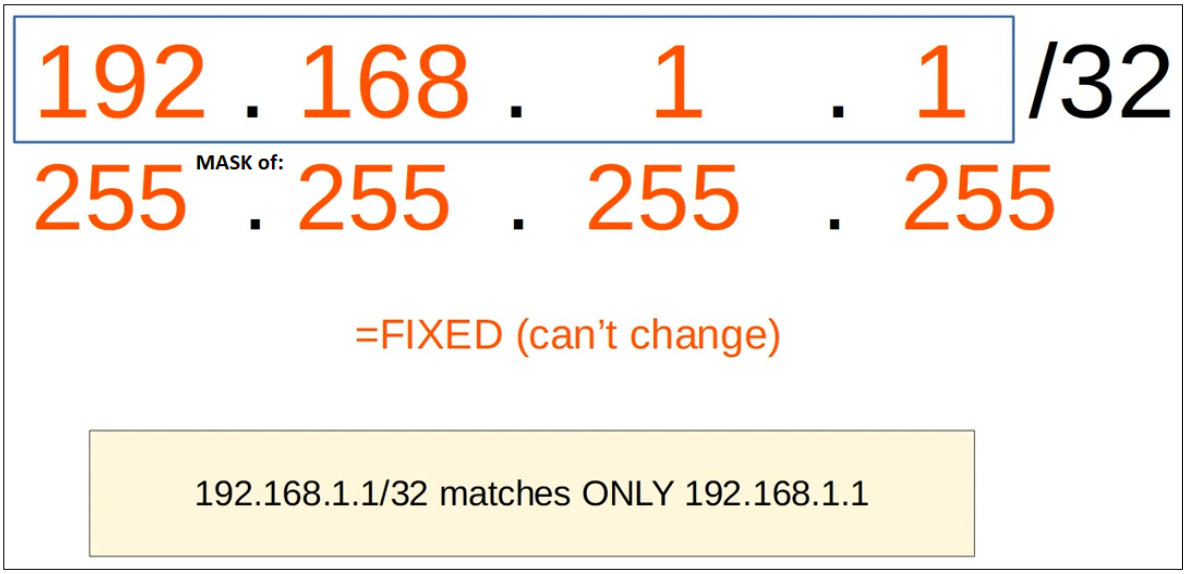
→ So, let's configure the *default-gateway* on our PC, which is actually a Cisco Router.
To configure the G*ateway of last resort*, we must configure a '***default route***'.

→ A '***default route***' is a route that matches **ALL** possible destinations.
It is used only if a more specific route match isn't found in the routing table.
When a Router looks up a route to a destination in its routing table, it looks for the most specific match, and chooses that route.
The *default route* is the LEAST SPECIFIC possible route.
It uses an IP address of: **0.0.0.0**
Mask of: **0.0.0.0**
* To set the *default route/gateway* of **last resort**, configure a route to **0.0.0.0/0**.
* The **/0** means that there is no network portion of the address, no fixed part of the address that can't change.

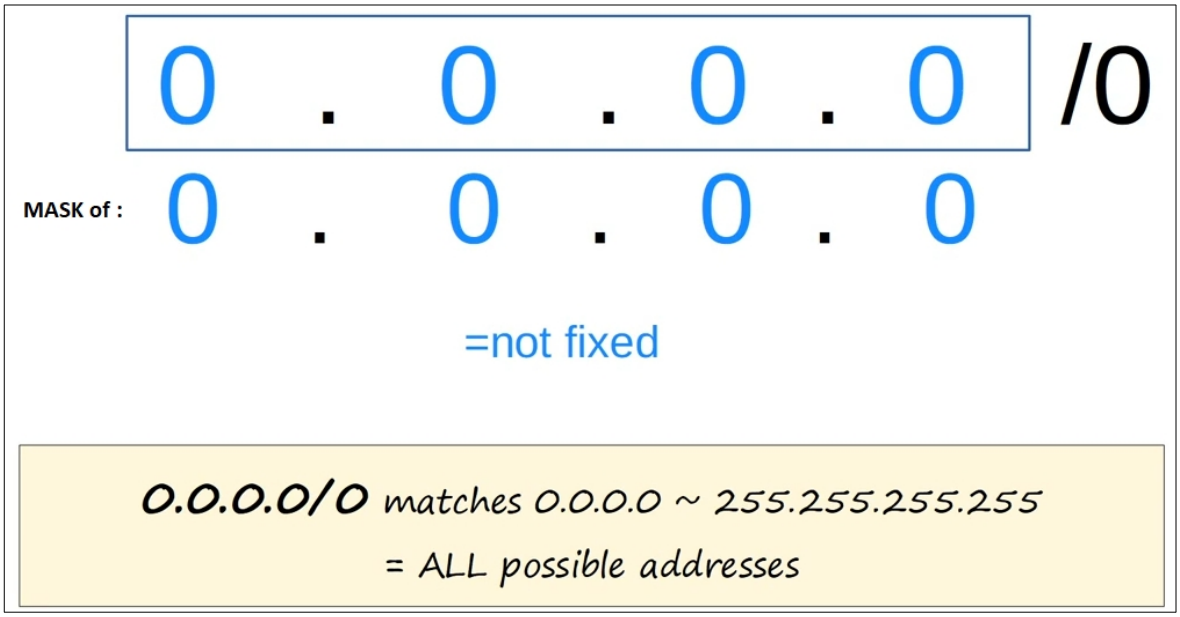
The **0.0.0.0/0** range includes **0.0.0.0** ~ **255.255.255.255 which is = ALL possible addresses**.

→ **Explanation:**





That's why the **/32** mask is used to specify the exact address configured on the interface in the **Local** route.



This is called the LEAST SPECIFIC route. AS It doesn't specify a single address, it includes ALL possible addresses.

→ Configuring a *Static Route*:

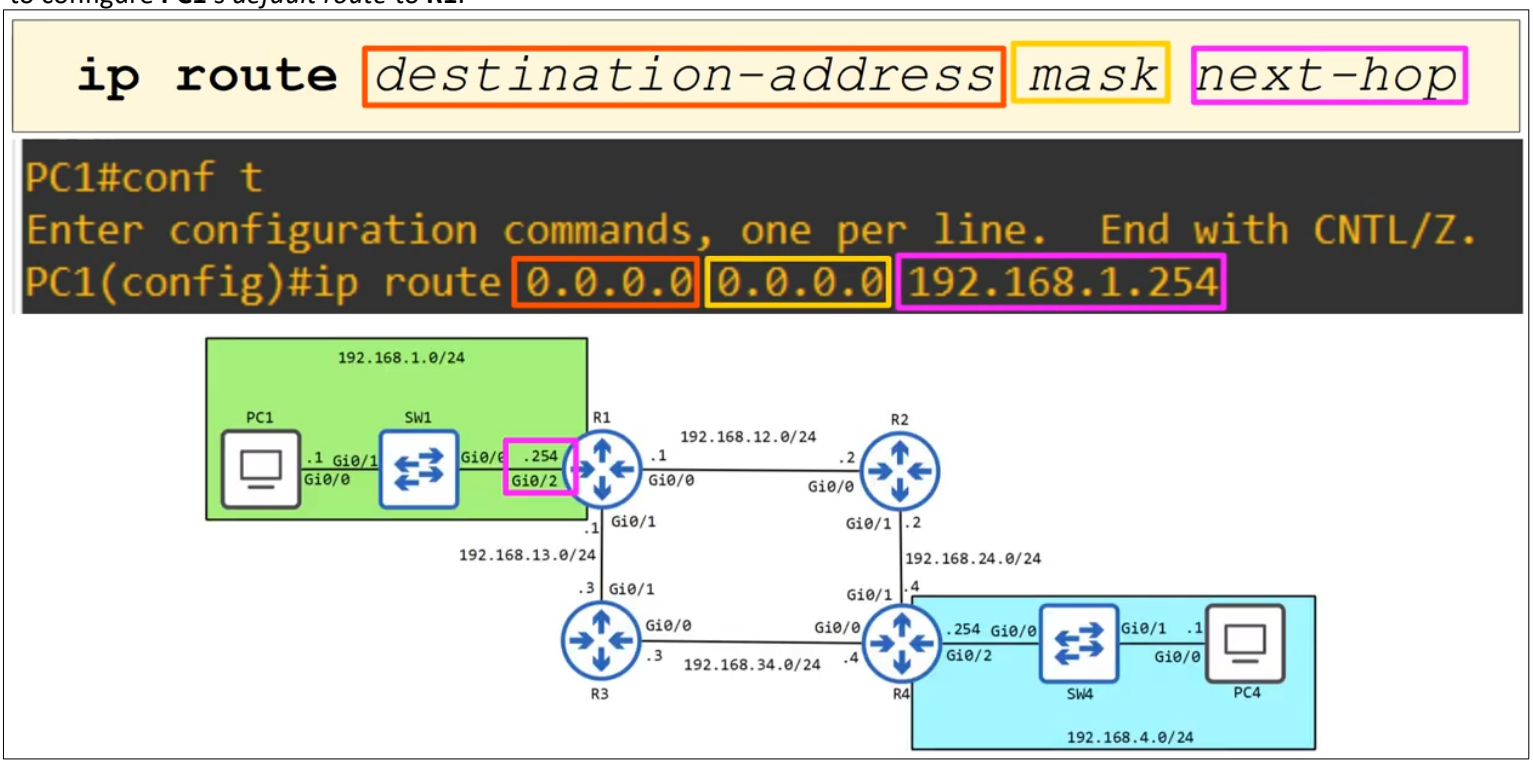
Let's go to actually configuring this *default route*.
To do so, we have to learn how to configure a '***Static Route***'.
A '***Static Route***': is a route you manually configure yourself.
This is different than **Local** and **Connected** routes we saw before, which are automatically added to the routing table when we configure an IP address on an interface and enable it.

Here's the command to configure a *static route:*

```
ip route destination-address mask next-hop
```

```
ip route destination-address mask exit-interface
```

→ So here, we use the configure terminal command to enter Global configuration mode, and then used **`ip route`** command to configure **PC1**'s *default route* to **R1**.



→ So, from global configuration mode we use '**`do show ip route`**' to check the routing table once more:
And first off, you can see it says '***Gateway of last resort is 192.168.1.254 to network 0.0.0.0***'
The '***default route***' also appears down here with the other routes.
NOTE the code on the left is different, we have an **S** and an asterisk *****, If we look up we will see that
S means **Static**, which we just explained before is a route we manually configure on the device.
***** The asterisk means it is a '***candidate default***' route.
The fact that we can see it is added as the gateway of last resort means it isn't just a ***candidate***, it was selected as the *default route*, but it's still labeled as a ***candidate default route***.

→ Looking back at our Topology, Since **PC1** now has a *default gateway* configured, known as a *Gateway of last resort* on a Cisco Router, it sends the packet to the *default gateway*, **R1**.
Since **R1** has the packet now, let's take a look at its routing table to see what it will do with the packet:

→ R1 routing table:

First we use '**`show ip route`**' command:
Because **R1** is connected to three networks, **192.168.1.0/24**, **192.168.12.0/24** and **192.168.13.0/24**, it has **Connected** routes for each of those networks, as well as the **Local** routes for the IP addresses configured on its interfaces.

```
Gateway of last resort is not set

      192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.1.0/24 is directly connected, GigabitEthernet0/2
L        192.168.1.254/32 is directly connected, GigabitEthernet0/2
      192.168.12.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.12.0/24 is directly connected, GigabitEthernet0/0
L        192.168.12.1/32 is directly connected, GigabitEthernet0/0
      192.168.13.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.13.0/24 is directly connected, GigabitEthernet0/1
L        192.168.13.1/32 is directly connected, GigabitEthernet0/1
```

However, here's a problem: The packet's destination is **192.168.4.1**, in the **192.168.4.0/24** network, but it is not in the routing table!
If this were a Switch receiving a frame with an unknown destination MAC address, It would flood the frame out of all of its interfaces. How about in this case, what will a Router do when it receives a packet with an unknown destination IP address?
The Router will simply ***drop*** the packet!

Switches **flood** frames with unknown destinations (destinations not in the MAC table).

Routers **drop** packets with unknown destinations.

So, unless we configure a route on **R1**, this ping will not succeed.

So, Let's configure a *Static route* on **R1** aimed for **192.168.4.0/24** network:
This time instead of specifying an IP address as the next hop, we just specify the ***exit-interface***, the interface out of which **R1** should send the packet.

```
ip route destination-address mask exit-interface
R1(config)#ip route 192.168.4.0 255.255.255.0 g0/0
```

Then we use '**`do show ip route`**' command, and here's the route we configured:

```
S       192.168.4.0/24 is directly connected, GigabitEthernet0/0
```

Notice how this static route isn't added as the *Gateway of last resort*, because it is for a specific network, not for **0.0.0.0/0** which matches all networks.

→ Now **R1** has a route in its routing table for the destination.
The destination of this packet is **192.168.4.1** which is included in the **192.168.4.0/24** network.
We don't need a route to the exact destination. Because **192.168.4.1** is included in the **192.168.4.0/24** rage, it is fine, it matches.

→ So, **R1** forwards the packet to **R2**.
And there's a problem here, **R2** also doesn't have a route that matches **192.168.4.1**
So we configure a route on **R2**, by the command '`ip route 192.168.4.0 255.255.255.0 192.168.24.4`'
We specified the next-hop IP address of **192.168.24.4** which is **R4**'s IP address.

```
S       192.168.4.0/24 [1/0] via 192.168.24.4
```

So, **R2** receives the packet with destination **192.168.4.1** and looks up the most specific match in its routing table.
It finds this entry for **192.168.4.0/24** and it says via **192.168.24.4**
R2 then looks up for the most specific match for **192.168.24.4**, that is this route here:

```
C       192.168.24.0/24 is directly connected, GigabitEthernet0/1
```

Which is a directly connected route on **GigabitEthernet0/1**, so it will forward the packet out of that interface.
So, **R2** sends the packet to **R4**.

→ R4 routing table:

We haven't configured any static routes on **R4**, will that be a problem?
Let's check:

```
Gateway of last resort is not set

      192.168.4.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.4.0/24 is directly connected, GigabitEthernet0/2
L        192.168.4.254/32 is directly connected, GigabitEthernet0/2
      192.168.24.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.24.0/24 is directly connected, GigabitEthernet0/1
L        192.168.24.4/32 is directly connected, GigabitEthernet0/1
      192.168.34.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.34.0/24 is directly connected, GigabitEthernet0/0
L        192.168.34.4/32 is directly connected, GigabitEthernet0/0
```

Because **R4**'s **G0/2** interface has an IP address of **192.168.4.254** configured, the **192.168.4.0/24** network has automatically been added to its routing table as a **Connected** route.
→ So, **R4** sends the packet out of its **G0/2** interface, and **SW4** forward it to **PC4**.

→ Ping from PC1 to PC4:

we use the command '**ping 192.168.4.1**' on **PC1**:

```
PC1#ping 192.168.4.1
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.4.1, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
PC1#
```

However, as you can see, the ping fails, success rate is 0 percent, 0 out of 5.
PC1 has a *default gateway*, all of the routers along the way know how to reach **PC4**. So, what's the problem??

→ The problem is ***one-way reachability***.
The packet from **PC1** was able to reach **PC4**. **R1**, **R2** and **R4** all have routes to the **192.168.4.0/24** network.
However. To send a reply back, **PC4** sends packets with a source of **192.168.4.1** and a destination of **192.168.1.1**.
The problem is that **R4** and **R2** have no route to reach the **192.168.1.0/24** network.

If we take a look back to **R4** routing table, there is no route to **192.168.1.0/24**. And also **R2**.
Not to mention, we also haven't configured the *default gateway* on **PC4**!

→ So, we configure the *default route* on **PC4** like we did on **PC1**, by using the command
'**`ip route 0.0.0.0 0.0.0.0 192.168.4.254`**'
And it has been selected as the *Gateway of last resort*, the *default gateway*.

```
Gateway of last resort is 192.168.4.254 to network 0.0.0.0

S*    0.0.0.0/0 [1/0] via 192.168.4.254
```

→ So, we also configure a static route to the **192.168.1.0/24** network, and the next-hop is **192.168.24.2**, **R2**'s address.
We use '**`ip route 192.168.1.0 255.255.255.0 192.168.24.2`**'

```
S     192.168.1.0/24 [1/0] via 192.168.24.2
```

→ And finally, **R2**'s configurations:
we use the command '**`ip route 192.168.1.0 255.255.255.0 192.168.12.1`**'

```
S     192.168.1.0/24 [1/0] via 192.168.12.1
```

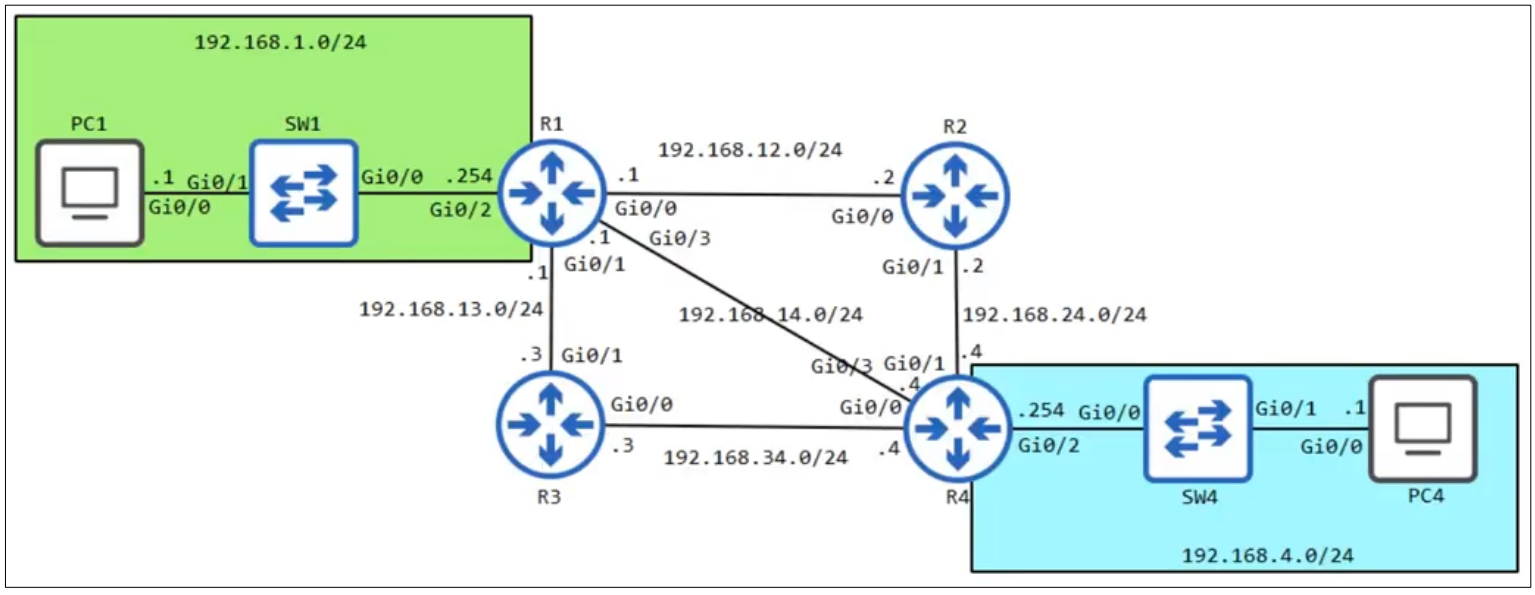
→ We should have ***two-way rachability*** now!

Okay, now we have configured static routes in both directions, you can see that the ping succeeds!

```
PC1#ping 192.168.4.1
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.4.1, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 26/31/41 ms
PC1#
```

→ Most Specific Matching Route:
- When a Router looks up a destination address in its routing table, it looks for the ***most specific matching*** route.
- Most Specific = Longest prefix length (**/32** > **/24** > **/16** > **/8** > **/0**).

→ Here We've added one more link between **R1** and **R4** to our Topology:



It's the **192.168.14.0/24** network, with **R1**'s address being **192.168.14.1** and **R4**'s **192.168.14.4**.
We also configured some extra routes, lets check the route table:

On the **R1**'s routing table, let's say we issue the command '**`ping 192.168.4.1`**' on **R1**.
How many routes actually match **192.168.4.1**?

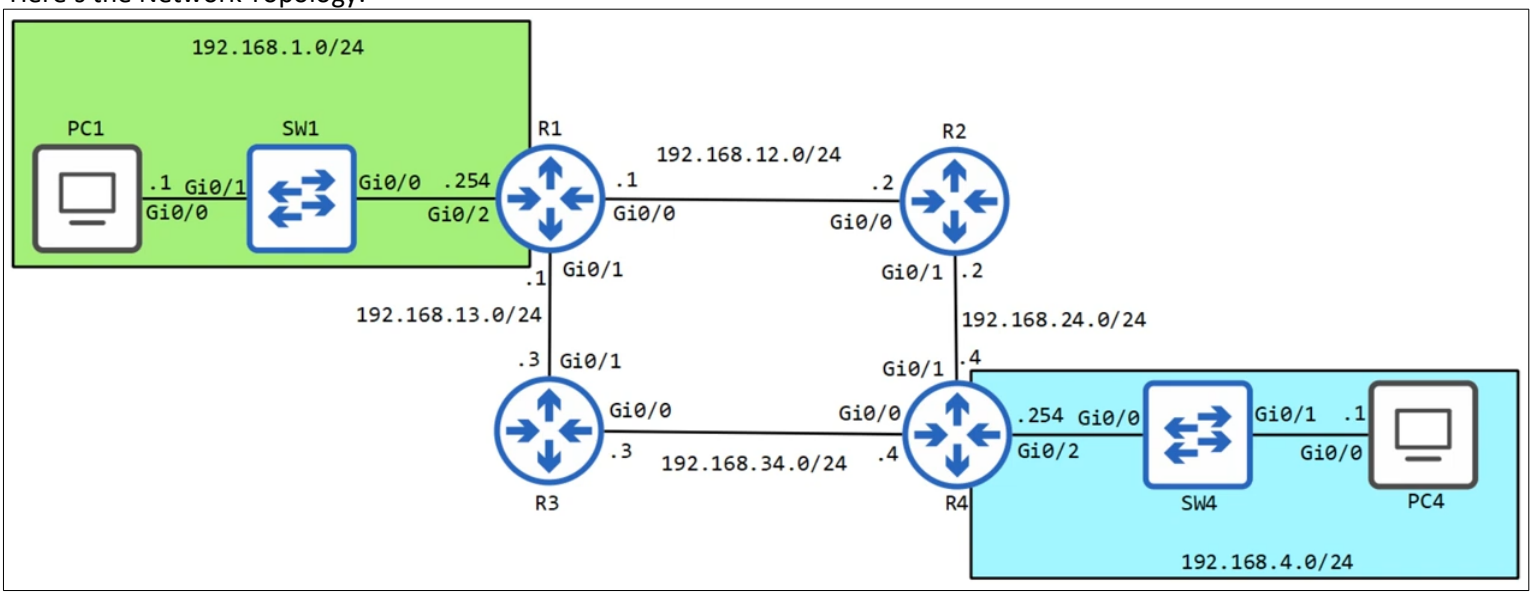
```
S       192.0.0.0/8 [1/0] via 192.168.13.3
        192.168.1.0/24 is variably subnetted, 2 subnets, 2 masks
C          192.168.1.0/24 is directly connected, GigabitEthernet0/2
L          192.168.1.254/32 is directly connected, GigabitEthernet0/2
        192.168.4.0/24 is variably subnetted, 2 subnets, 2 masks
S          192.168.4.0/24 is directly connected, GigabitEthernet0/0
S          192.168.4.1/32 [1/0] via 192.168.14.4
        192.168.12.0/24 is variably subnetted, 2 subnets, 2 masks
C          192.168.12.0/24 is directly connected, GigabitEthernet0/0
L          192.168.12.1/32 is directly connected, GigabitEthernet0/0
        192.168.13.0/24 is variably subnetted, 2 subnets, 2 masks
C          192.168.13.0/24 is directly connected, GigabitEthernet0/1
L          192.168.13.1/32 is directly connected, GigabitEthernet0/1
        192.168.14.0/24 is variably subnetted, 2 subnets, 2 masks
C          192.168.14.0/24 is directly connected, GigabitEthernet0/3
L          192.168.14.1/32 is directly connected, GigabitEthernet0/3
```

We have three routes that matches: **192.0.0.0/8**, **192.168.4.0/24** and **192.168.4.1/32**
and the Most Specific Matching Route is **192.168.4.1/32**.
- We haven't seen a Class C address like **192.0.0.0** with a **/8** mask, but it follows the same logic.
  /8 means that only the first octet is covered by the mask. The other three octets can vary, and they will still match.
  So, **192.168.4.1** does match **192.0.0.0/8**.
- However, As we said, **192.168.4.1** is the most specific route, since it has the longest prefix length, **/32**.
  So, in this case, the Router will choose that route.

Day 12: Life of a Packet:-

→ The Process of Sending a Packet to Remote Networks:-

Here's the Network Topology:

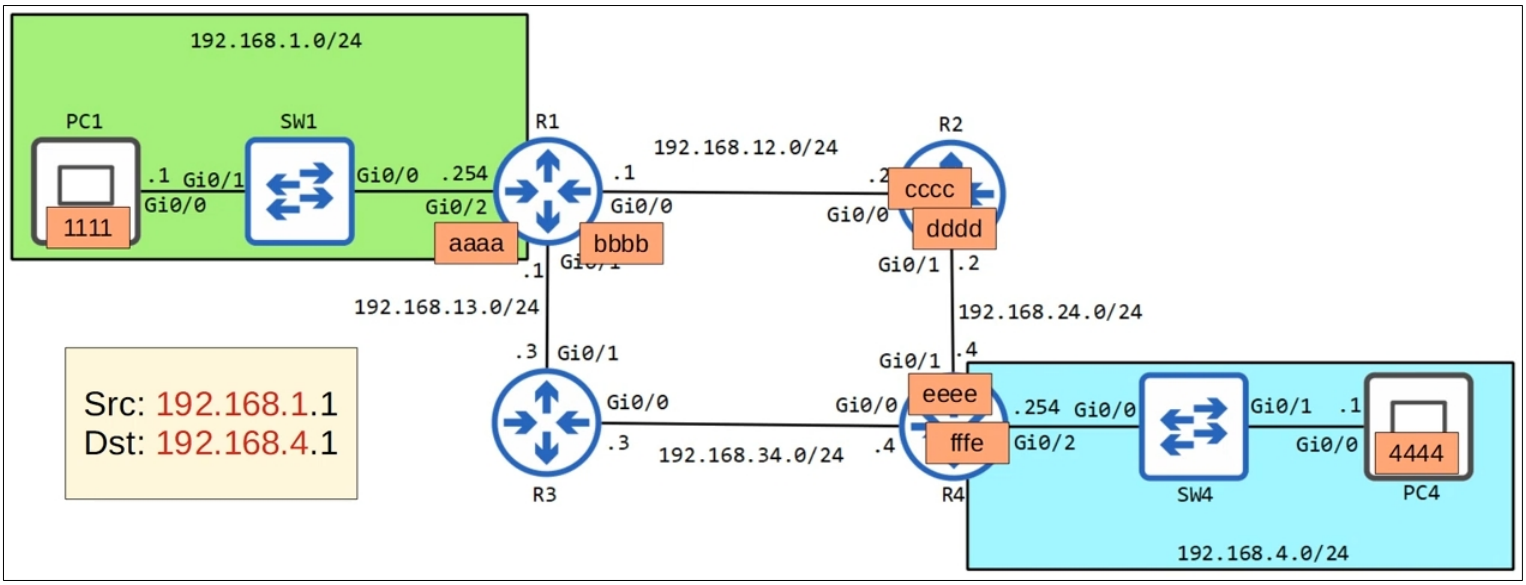


The same Topology we've used to demonstrate Static Routing in Day11 video.

We'll follow a Packet being sent from **PC1** in the **192.168.1.0/24** network, to **PC4** in the **192.168.4.0/24** network.

→ Let's assume we have pre-configured Static Routes on these devices, so that the packet will follow the same path as in the Static Routing video, that is from **PC1** to **R1**, **R2**, **R4**, and then **PC4**.
This doesn't have to be the path the packet takes, the path that goes via **R3** instead of **R2** is valid too, but we'll stick to the last one.

→ Since we're not just looking at Layer3 in this video, let us add MAC addresses for these devices.
As you know MAC address is **12** hexadecimal characters, but just to save space we'll shorten them to **4**:



→ Each interface on a Network device has a ***unique*** MAC address ←

It is not necessary to know the MAC address of the Switches just to avoid clutter.
We didn't make the MAC address of the **G0/2** interface of **R4** with all **FFFF**'s,
because the **FFFF.FFFF.FFFF**, **12 F**s, is the ***Broadcast MAC Address***.
So, just to avoid confusion we added **E** on the end.

Ok, so **PC1** wants to send some data to **PC4**, and it is encapsulated in that IP header. The Source is **192.168.1.1**, **PC1**'s IP address, and the destination is **192.168.4.1**, **PC4**'s IP address.
Now, because **PC1**'s IP address is in the **192.168.1.0/24** network, it notices that the address **192.168.4.1** is in a different network.
→ So it knows that it need to send the packet to its *default gateway*, which is **R1**, something we have already pre-configured.
However, in this example, **PC1** has not sent any traffic yet, so it needs to us **ARP**, Address Resolution Protocol.

→ ARP Process:-

→ So, **PC1** makes this ARP request packet:

ARP Request
Src IP: 192.168.1.1
Dst IP: 192.168.1.254
Dst MAC: ffff.ffff.ffff
Src MAC: 1111

The Source IP address is its own IP address and then The Destination is **R1**'s **G0/2** interface, which is the *default gateway* configured on **PC1**.
The Destination MAC address is the Broadcast MAC address of all **F**s, because it doesn't know the MAC address of **R1**, so it will send the frame to all hosts on the network. Finally, the Source MAC address is its own MAC address.

→ So, it sends the frame, which **SW1** receives and broadcasts out of all its interfaces except the one it received the frame on.
This means that **SW1** will forward the frame out of it's **C0/0** interface.
To translate the meaning of this frame into English, **PC1** is saying **'***Hi 192.168.1.254, What's your MAC address?***'**
NOTE: SW1 learns **PC1**'s MAC address on its **G0/1** interface when the frame arrives on its **G0/1** interface.

→ When this broadcast frame arrives on **R1**, it notices that the destination IP is its own IP, so it creates this ARP reply frame to send back to **PC1**.

ARP Reply
Src IP: 192.168.1.254
Dst IP: 192.168.1.1
Dst MAC: 1111
Src MAC: aaaa

Although, the ARP request message was broadcast, because **R1** learned **PC1**'s IP and MAC addresses from the ARP request message, the ARP reply can be sent unicast directly to **PC1**.
That's what **R1** does.

To translate this ARP reply message into English, basically it means **'***Hi 192.168.1.1, This is 192.168.1.254, My MAC address is AAAA***'**.
NOTE: SW1 will learn **R1**'s MAC address from this message, when the frame arrives on its **G0/0** interface.

→ So, now **PC1** knows the MAC address of its *default gateway*, so it encapsulates the packet with this Ethernet header.
Keep in mind, the original packet is not changed, the destination address remains **PC4**'s IP address, NOT **R1**'s IP address.
Only at Layer2 is the destination set to **R1**'s MAC address.
So, it sends the frame to **R1**.

Src: 192.168.1.1 Dst: 192.168.4.1	Dst: aaaa Src: 1111

→ **R1** receives it, and removes the Ethernet header.
It looks up the destination in its routing table. The most specific match is this entry for the **192.168.4.0/24** network, which specifies a next hop of **192.168.12.2**.
So, **R1** will have to encapsulate this packet with an Ethernet frame with the appropriate MAC address for **192.168.12.2**.
However, **R1** doesn't know **R2**'s MAC address yet.

R1 Routing Table

Destination	Next Hop
192.168.4.0/24	192.168.12.2
...	

→ So, **R1** will send ARP request
The source IP address of this ARP request will be **R1**'s, **192.168.12.1**, and the destination will be **R2**'s, **192.168.12.2**.
The destination MAC address is all **F**s, the broadcast MAC address, because **R1** doesn't know **R2**'s MAC address, and the source is **bbbb**, which is the MAC address of **R1**'s **G0/0** interface.
So, it sends the ARP request, and **R2** receives it

ARP Request
Src IP: 192.168.12.1
Dst IP: 192.168.12.2
Dst MAC: ffff.ffff.ffff
Src MAC: bbbb

ARP Reply
Src IP: 192.168.12.2
Dst IP: 192.168.12.1
Dst MAC: bbbb
Src MAC: cccc

→ **R2** receives the broadcast, and since the destination IP address matches its own IP address, it sends this ARP reply to **R1**.
Because it learned the IP and MAC address of **R1** from the ARP request, It doesn't have to broadcast the frame.
So, **R2** sends this ARP reply back.
Now **R1** knows **R2**'s MAC address, so it can encapsulate the packet with an Ethernet header, inserting **R2**'s MAC address in the destination field, and the MAC address of **R1**'s **G0/0** interface in the source field, and it sends it to **R2**.
After receiving the frame **R2** removes the Ethernet header.
R2 then looks up the destination IP address in its routing table, and the most specific match is this one for **192.168.4.0/24**, with next hop of **192.168.24.4**
Although **192.168.24.0/24** is a connected network to **R2**,
it doesn't know the MAC address of **R4**.

R2 Routing Table

Destination	Next Hop
192.168.4.0/24	192.168.24.4
...	

→ **R2** will now use ARP request to discover **R4**'s MAC address.
R2 prepares this ARP request frame, using its own IP and MAC addresses for the source, **R4**'s IP address as the destination, and the broadcast MAC address, and it forwards it out of its **G0/1** interface.
With this ARP request, **R2** is saying **'***Hi 192.168.24.4, What's your MAC address?***'**

ARP Reply
Src IP: 192.168.24.4
Dst IP: 192.168.24.2
Dst MAC: dddd
Src MAC: eeee

→ **R4** receives the broadcast, and since the destination IP address matches its own IP address, it creates this ARP reply frame to send back to **R2**, once again it already knows **R2**'s IP and MAC addresses because they were used as the source addresses for the ARP request.
It sends the *unicast* frame back to **R2**.
With this reply, **R4** is saying **'***Hi 192.168.24.2, This is 192.168.24.4, My MAC address is EEEE***'**

ARP Reply
Src IP: 192.168.24.4
Dst IP: 192.168.24.2
Dst MAC: dddd
Src MAC: eeee

→ Now that **R2** knows **R4**'s MAC address, it encapsulates **PC1**'s packet with an Ethernet header, with a destination MAC address of *EEEE*, which is **R4**'s **G0/1** interface, and a source MAC address of *DDDD*, which is **R2**'s **G0/1** interface.

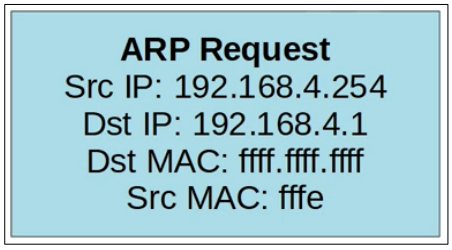
Src: 192.168.1.1 Dst: 192.168.4.1	Dst: eeee Src: dddd

→ **R4** receives the frame and removes the Ethernet header.
It looks up **192.168.4.1** in its routing table, and the most specific match is this entry for **192.168.4.0/24**, which is directly connected via the **G0/2** interface.
But, once again, **R4** doesn't know **PC4**'s MAC address yet.
It will use ARP request to learn **PC4**'s MAC address.
It prepares this ARP request frame, again, the source IP and MAC addresses are its own, the destination IP address is **PC4**'s, and the destination MAC address is the broadcast MAC address, all **F**s.
It sends this message out of its **G0/2** interface

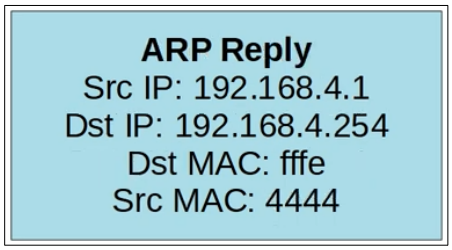
R4 Routing Table

Destination	Next Hop
192.168.4.0/24	directly connected, Gi0/2
...	

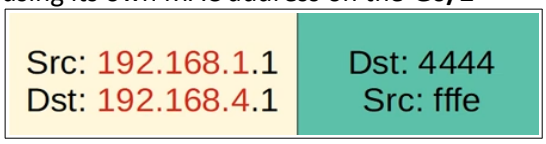
NOTE: SW4 will learn **R4**'s MAC address on its **G0/0** interface from the source MAC address field of his Ethernet frame.



→ After **PC4** receives the frame it checks the destination IP address, since it is its own IP address, it will send and ARP reply.
The ARP reply will be *unicast*, using **PC4**'s IP and MAC addresses for the source and **R4**'s IP and MAC addresses for the destination.
It sends the frame out of its **G0/0** network interface.
NOTE: **SW4** learns **PC4**'s MAC address when it arrives on its **G0/1** interface.



→ Now that **R4** knows **PC4**'s MAC address, it adds an Ethernet header to the packet, using its own MAC address on the **G0/2** interface as the source address, and **PC4**'s MAC address as the destination.
R4 sends the frame to **PC4**, and finally it has reached its destination :)



NOTE: The original Packet hasn't changed throughout the process.
It is always used the same IP header with a source IP address of **192.168.1.1** and a destination IP address of **192.168.4.1**.

NOTE: Switches didn't actually modify the frames at any point. The Switches forwarded the frames and learned the MAC addresses, but they don't actually de-encapsulate and then re-encapsulate the packet with a new Ethernet header.

→ **Minor Point:** In the ARP request , we put the Source IP before the Destination IP, but the Destination MAC before the Source MAC. That's because, in the IPv4 Header the source IP address comes first, but in the Ethernet header the Destination MAC Address comes first.

→ Okay, Now let's say **PC4** sends a reply back to **PC1**, and we've configured static routes on the Routers so that the traffic follows the same path on the way back to **PC1**, going via **SW4**, **R4**, **R2**, **R1**, **SW1**, and then reaching **PC1**, what will be different??
First off, there will be one major difference:
Since these devices have already gone through the ARP process, *there won't be any need for ARP requests and replies.*
The Packet will simply be forwarded from device to device, being de-encapsulated aand then re-encapsulated as it is received by and then forwarded by each Router.

→ Some commands from the Lab video:

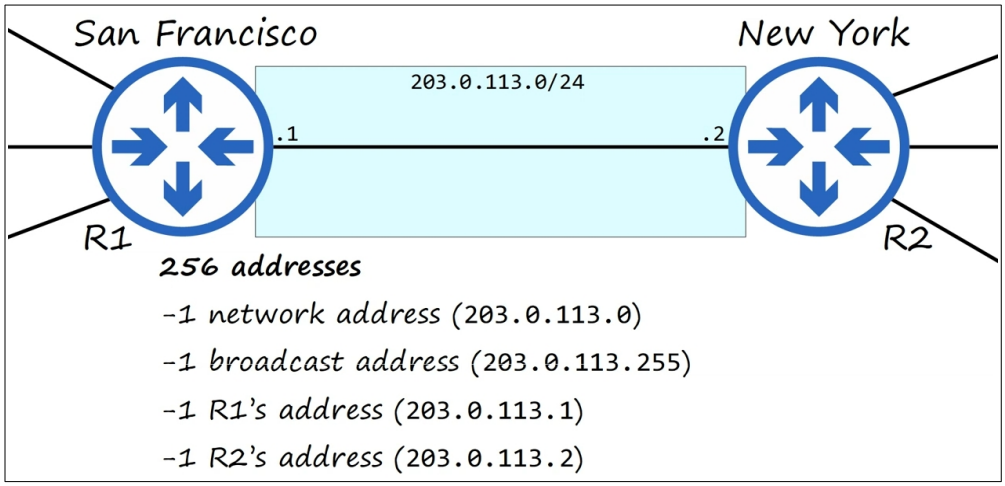
- `ipconfig /all` to know the MAC address and other info on the PC
- `show interfaces [g0/0]` to show the detailed information of the Network interface on a Router
-

→ The **BIA**, Burned-In Address, is the ***actual*** MAC address assigned to the interface by the device Maker, when it was made. You can actually configure a different MAC address in the CLI, and it will use that MAC address!

Day 13: Subnetting pt.1:-

→ IPv4 Address Classes:-

→ How does a company get their own network to use? Well, IP addresses are assigned to companies or organizations by a non-profit American corporation called the **IANA**, the **Internet Assigned Numbers Authority**.
The **IANA** assigns **IPv4** addresses and networks to companies based on their size.
For example, a very large company might receive a **Class A** or **Class B** network,
while a small company might receive a **Class C** network.
This leads to many WASTED IP addresses!
So, multiple methods of improving this system have been created.
An example of how this strict system can waste IP addresses:-



→ We have 2 Routers connected together, **R1** and **R2**. Each Router has *3 separate Layer3 Networks, with different IP networks*.
→ Each of these networks will have a few Switches, with many end-hosts such as PCs and Servers connected to these Switches.
→ There is one more network, The network connecting the two Routers, **R1** and **R2**. It is known as ***point-to-point*** network.
→ Meaning its a network connecting two points.
→ This might be a connection between Offices in different cities, Let's Say San Francisco and New York.
→ As it is a *point-to-point* network, we don't need a large address block, so we can use a Class C network, **203.0.113.0/24**.
→ You will note here that there are only **4** IP addresses in use, and **252** IP addresses are WASTED!
→ This is not an IDEAL system!

Another example:-
→ Company X needs IP addressing for **5000** end hosts. This is a bi problem. Why!?
Because a Class C network doe not provide enough addresses, so a Class B network must be assigned!
Because a Class B network will allows for about **65,000** addresses, this results in about **60,000** addresses being WASTED!

→ CIDR (Classless Inter-Domain Routing):-

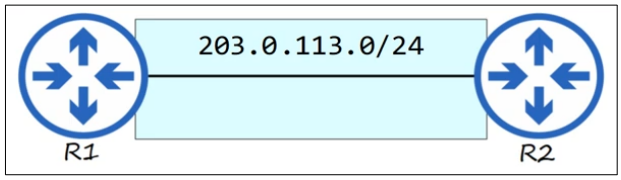
We have introduced **IPv4** adress classes, such as Class A, B and C.
Well, **CIDR** throws all that away and lets us be more flexible with our IPv4 networks.

→ When the Internet was firs being created, the creators did not predict that the Internet would become as large as it is today.
This results in wasted address space like the examples we showed * (And there are many more examples!).

→ The total **IPv4** *address space* includes over **4B billion** of addresses, and that seemed like a huge number of addresses when **IPv4** was created, but now address space exhaustion is a big problem, there's not enough addresses.
→ One way to solve, or remedy this problem is **CIDR**. The **IETF**, (**Internet Engineering Task Force**) introduced **CIDR** in **1993** to replace the '*classful*' addressing system.

→ With **CIDR**, the requirements of (Class **A** addresses must use a **/8** network mask, Class **B /16** mask and Class **C /24** mask) were removed! This allowed larger networks to be split into smaller networks, allowing greater efficiency.
These smaller networks are called '***Subnetworks***' or '***subnets***'.

→ Let's splitt a larger network into a smaller network:



→ Here's the same point-to-point network we looked at before.
Previously, it was assigned the **203.0.113.0/24** network space, but that resulted in lots of wasted addresses!
→ Let's write this out in binary:-
OK, so here's the binary with the dotted decimal underneath:

```
1 1 0 0 1 0 1 1 . 0 0 0 0 0 0 0 0 . 0 1 1 1 0 0 0 1 . 0 0 0 0 0 0 0 0
  203     .     0     .     113     .     0
```

→ The prefix length is **/24**, so here's the network mask, aka, the subnet mask, **255.255.255.0**:

```
1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 0 0 0 0 0 0 0 0
  255     .     255     .     255     .     0
```

REMEMBER: All **1**'s in the subnet mask indicate that the same bit in the address is the network portion
In this case, I made the network portion blue, and the host portion is red.

→ Well, how many host bits are there? **8**, because it's one octet! So, how many potential hosts, or how many usable addresses are there?
Well, the formula is this:

$$2^8 - 2 = 254 \text{ usable addresses}$$

8, the power of **2**, is the number of host bits.
And minus two (**-2**), the network address and the broadcast address.
So, we have **254** usable address, but we only need two, one for **R1** and one for **R2**.
However, **CIDR** allows us to assign different prefix length, it doesn't have to be **/24**.

→ CIDR practice:-
→ Let's get some practice calculating the number of hosts with different prefix length.
203.0.113.0/25. **203.0.113.0/26**. **203.0.113.0/27**, **/28**, **/29**, **/30**, **/31** and finally **/32**.
We put **/31** and **/32** in highlight because they're a little bit special, you'll see when you try to calculate it.
Here the formula to get the usable addresses in the network:

$$2^n - 2 = \text{usable addresses}$$

n = number of host bits

→ So, the solution will be **126**, **62**, **30**, **14**, **6**, **2**, **0**, **0**

→ Video Solution: Here is **203.0.113.0**, but this time with a **/25** mask.
Notice that the network portion of the address has extended into the first bit of the last octet, and the mask in dotted decimal is now written as **255.255.255.128**. We changed the color of the extra bit to purple, but it's part of the network portion, blue part.
→ Now, there are **7** bits in the host portion of the address, so the number of usable addresses is **2** to the power of **7**, minus **2**, which equals **126**. We only need **2** addresses, one for **R1** and one for **R2**, so we'll be wasting **124** addresses!
That's better than wasting **252** addresses with a **/24** prefix length, but still its wasteful.
And so on to the other prefix lengths. And the solution as we did it!
→ If we use a **/30** prefix length, the mask is written as **255.255.255.252** in dotted decimal. There are now only **2** host bits.
That means **2** usable addresses. So, this is perfect! That means **0** wasted addresses!

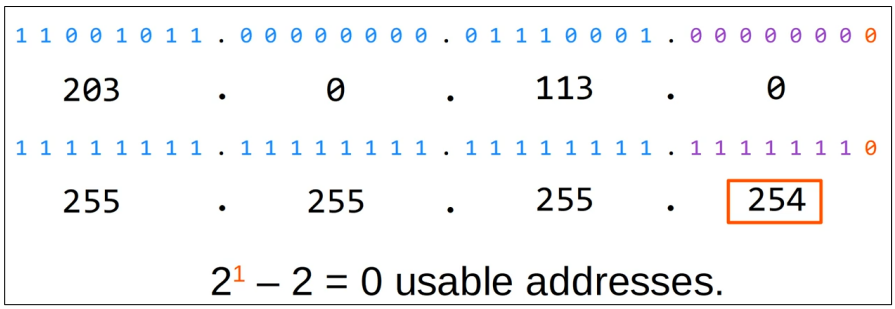
→ Before moving on to **/31** and **/32** let me clarify a little bit:
So, instead of **203.0.113.0/24**, we will use **203.0.113.0/30**, which is a *subnet* of that larger Class **C** network.
203.0.113.0/30 includes the address range of **203.0.113.0** through **203.0.113.3**.
This is how it is converted in binary: These are the **4** addresses in the network which we took up with that subnet.

```
203.0.113.0/30
= 203.0.113.0 - 203.0.113.3
1 1 0 0 1 0 1 1 . 0 0 0 0 0 0 0 0 . 0 1 1 1 0 0 0 1 . 0 0 0 0 0 0 0 0
1 1 0 0 1 0 1 1 . 0 0 0 0 0 0 0 0 . 0 1 1 1 0 0 0 1 . 0 0 0 0 0 0 0 1
1 1 0 0 1 0 1 1 . 0 0 0 0 0 0 0 0 . 0 1 1 1 0 0 0 1 . 0 0 0 0 0 0 1 0
1 1 0 0 1 0 1 1 . 0 0 0 0 0 0 0 0 . 0 1 1 1 0 0 0 1 . 0 0 0 0 0 0 1 1
```

→ So, What about other addresses in the **203.0.113.0/24** range?
The remaining address in the **203.0.113.0/24** address block (**203.0.113.4** – **203.0.113.255**),
Are now available to be used in other *subnets*! That's the magic of subnetting!

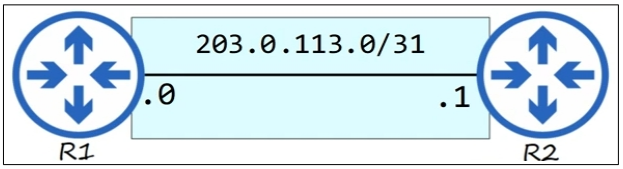
→ Instead of using **203.0.113.0/24** and wasting **252** addresses, we can use **/30** and waste no addresses.
Or, perhaps there is another way to make this more efficient? Let's look into it:

→ CIDR (/31):-
→ If we use a **/31** prefix length, the mask is written as **255.255.255.254** in dotted decimal.
There is now only **1** host bit, so that means '**0 usable addresses**'.
2 to the power of **1** is **2**, minus **2** for the network and broadcast addresses, which equals **0** usable addresses that we can assign.
→ So, you used to NOT be able to use **/31** network prefixes because of this.



HOWEVER, for a point-to-point connection like this it actually is possible to use a **/31** mask!

→ Back to the network, **203.0.113.0/31** network, **R1** is **203.0.113.0**, and **R2** is **203.0.113.1**.
The **203.0.113.0/31** network consists of the addresses from **203.0.113.0** through **203.0.113.1**, which is actually only TWO addresses.



→ Normally, this would be a problem, because it leaves no usable addresses after subtracting the network and broadcast addresses!
→ But for *point-to-point* networks like this, a dedicated connection like this between two Routers, there is actually no need for a network address or a broadcast address.
→ So, we can break the rules in this case and assign the only two addresses in this network to our Routers.

NOTE: If you try this configuration on a Cisco Router, you'll get a warning like this, reminding you to make sure that this is a *point-to-point* link, but it is a totally valid configuration.

```
Router(config-if)#ip address 203.0.113.0 255.255.255.254
% Warning: use /31 mask on non point-to-point interface cautiously
Router(config-if)#
```

→ Once again, The reminding addresses in the **203.0.113.0/24** address block, which are (**203.0.113.2** – **203.0.113.255**) are now available to be used in other networks.
But this time we saved even more addresses, using only **2** addresses instead of **4** for this *point-to-point* connection.
→ People still use **/30** for *point-to-point* connections at times, but **/31** masks are totally valid and more efficient than **/30**.
→ **/31** mask is RECOMMENDED!

→ CIDR (/32):-
→ **/32** mask is written as**255.255.255.255** in dotted decimal, making the entire address the network potion, there are no host bits.
→ If you calculate this using our formula, you will get **-1** usable addresses.
Clearly the formula doesn't work in this case.
You won't be able to use a **/32** mask in the case of *point-to-point* connection, and you will probably never use a **/32** mask to configure an actual interface.

HOWEVER, there are some uses for **/32** mask, for example: When you want to crate a Static Route, not to a network, but just specific host, you can use a **/32** mask to specify that exact host.

CIDR Notation: is the way of writing a prefix with a slash followed by the prefix length, like **/25**, **/26**, etc.
Because it was introduced with the **CIDR** system.
Previously, only the dotted decimal method was used.

NOTE: We have only learned how to *subnet* a Class **C** network, but we will look at Class **A** and Class **B** networks as well!

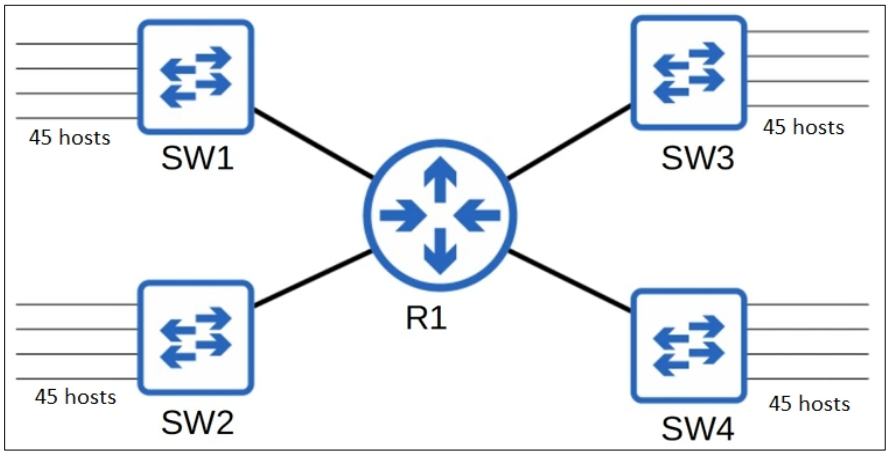
Here is a chart showing the dotted decimal subnet masks, and their equivalent in **CIDR** Notation:

Dotted Decimal	CIDR Notation
255.255.255.128	/25
255.255.255.192	/26
255.255.255.224	/27
255.255.255.240	/28
255.255.255.248	/29
255.255.255.252	/30
255.255.255.254	/31
255.255.255.255	/32

→ The process of Subnetting:-

→ ***Subnetting***: dividing a larger network into smaller networks, called ***subnets***.

Another one example of subnetting:



There are **4** networks connected to **R1**, with many hosts connected to each Switch.

There are **45** hosts per network, **R1** needs an IP address in each network so its address is included in that **45** host number.

You have received the **192.168.1.0/24** network,
And you must divide the network into four subnets that can accommodate the number of hosts required.

→ First off, are there enough addresses in the **192.168.1.0/24** network in he first place?
So, we need **45** hosts per network, including **R1**, but also remember that each network has a network and broadcast address, so that's plus **2**, so we need **47** addresses per subnet. **47** times **4** equals **188**, so there's no problem in terms of the number of hosts.
192.168.1.0/24 is a Class **C** network, so there are **256** addresses, so we will be able to assign **4** subnets to accommodate all hosts, no problem.

→ Let's see how we can calculate the subnets we need to make. We need **4** equal sized subnetw with enough room for at least **45** hosts.
Here, We've written out **192.168.1.0** with a **/30** mask, **255.255.255.252**.
I skipped **/32** and **/31**, since these aren't *point-to-point* links, we can't use **/31**, and definitely can't use **/32**.
Since there are **2** hosts bits, the formula to determine the number of usable addresses is **2** to the power of **2**, minus **2**.
So that means there are **2** usable addresses in a **/30** network. Clearly not enough room to accommodate the **45** hosts we have.

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 0 0 1 . 0 0 0 0 0 0 0 0
   192      .     168     .     1     .     0

1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 0 0
   255      .     255     .     255     .     252
```

→ Ho about if we use a **/29** mask to make these subnets, can we fit the **45** hosts we need? There are three bits, so the formula is **2** to the power of **3**, minus **2**. Therefore there are **6** usable addresses, not enough for **45** hosts.

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 0 0 1 . 0 0 0 0 0 0 0 0
   192      .     168     .     1     .     0

1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 0 0 0
   255      .     255     .     255     .     248
```

→ How about if we use **/28**? There are **4** host bits, so the formula is **2** to the power of **4**, minus **2**. So that means there are **14** usable addresses. And also not enough.

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 0 0 1 . 0 0 0 0 0 0 0 0
   192      .     168     .     1     .     0

1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 0 0 0 0
   255      .     255     .     255     .     240
```

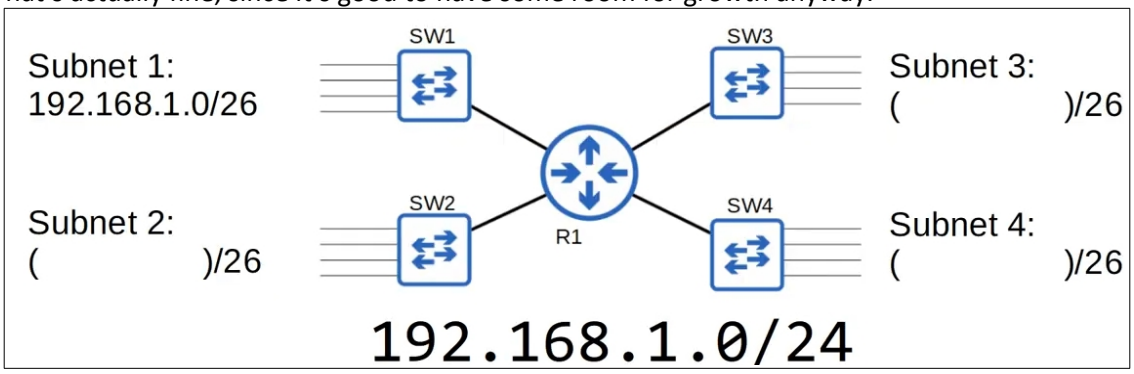
→ How about **/27**? There are **5** host bits, so the formula is **2** to the power of **5**, minus **2**. Equals **30** usable addresses. Not enough.

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 0 0 1 . 0 0 0 0 0 0 0 0
   192          .     168        .       1        .       0

1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 0 0 0 0 0
   255          .     255        .       255      .     224
```

→ How about **/26** mask? There are now **6** host bits, so the formula is **2** to the power of **6**, minus **2**.
That means there are **62** usable address! And finally we found a number!

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 0 0 1 . 0 0 0 0 0 0 0 0
   192          .     168        .       1        .       0

1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 0 0 0 0 0 0
   255          .     255        .       255      .     192
```

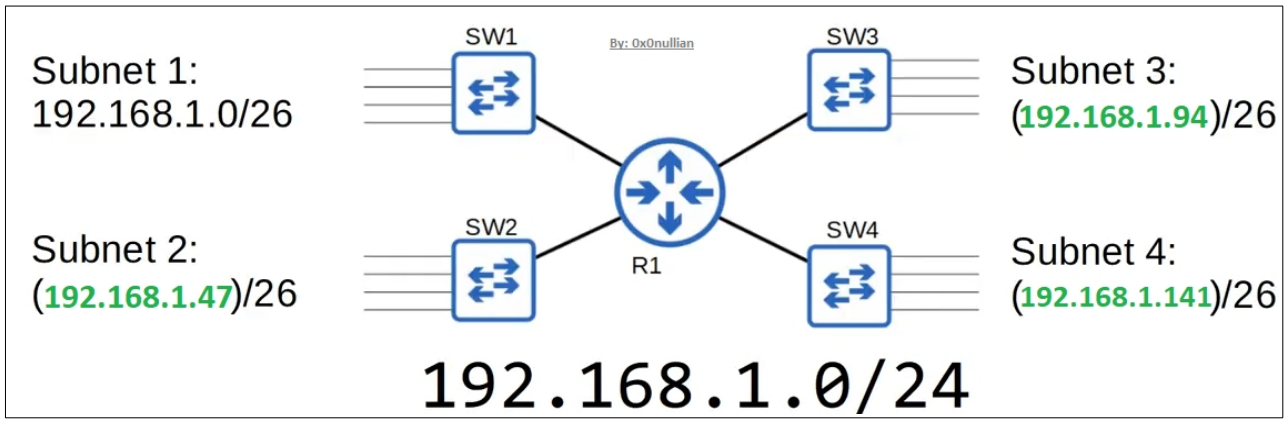
→ **/27** doesn't provide enough address space. **/26** provides more than we need, but we have to go0 with **/26**
Unfortunately we can't always make subnets have exactly the number of addresses we want. There might be some unused address space. That's actually fine, since it's good to have some room for growth anyway.



→ The first subnet (*Subnet1*) is **192.168.1.0/26**. What are the remaining subnets???
This will be a QUIZ to solve!
→ Hint: Find the broadcast address of *Subnet1*. The next address is the network address of *Subnet2*. Repeat the process for *Subnet3* and *Subnet4*.

The QUIZ answer:
→ As the *Subnet1* have **45** hosts, excluding the network address and the broadcast address, the network address is **192.168.1.0/26**, and the first IP address is **192.168.1.1/26** so the broadcast address will be after the last IP address.
The broadcast address is **192.168.1.46**.

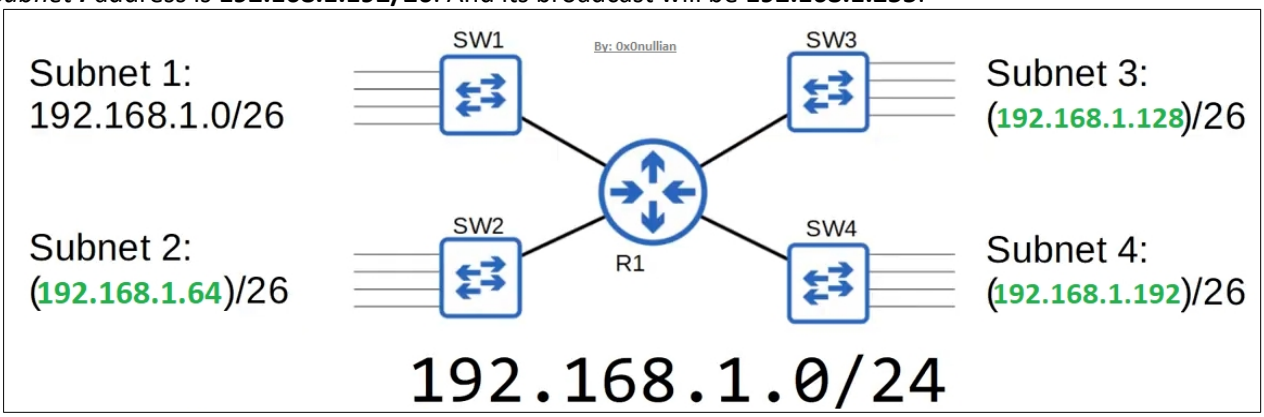
→ So the *Subnet2* address is **192.168.1.47/26**. And its broadcast will be **192.168.1.93**.
→ So the *Subnet3* address is **192.168.1.94/26**. And its broadcast will be **192.168.1.140**.
→ So the *Subnet**4*** address is **192.168.1.141/26**. And its broadcast will be **192.168.1.187**.



My Quiz answer was ***WRONG!***
As I solve it as there are only **47** hosts only, not depending on the prefix length which is **/26** which will be there **64** hosts!

The right ***ANSWER:***
→ As the *Subnet1* have **45** hosts, excluding the network address and the broadcast address, the network address is **192.168.1.0/26**, and the first IP address is **192.168.1.1/26** so the broadcast address will be after the last IP address. The broadcast address is **192.168.1.63**.

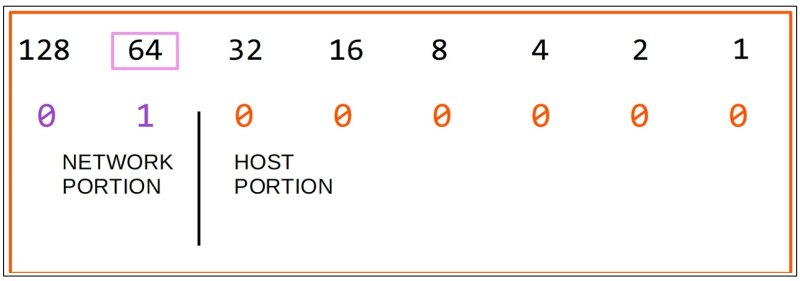
→ So the *Subnet2* address is **192.168.1.64/26**. And its broadcast will be **192.168.1.127**.
→ So the *Subnet3* address is **192.168.1.128/26**. And its broadcast will be **192.168.1.191**.
→ So the *Subnet**4*** address is **192.168.1.192/26**. And its broadcast will be **192.168.1.255**.

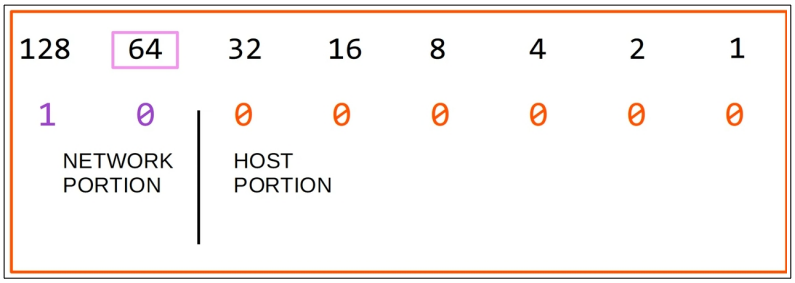


Day 14: Subnetting pt.2:-

→ Subnetting Practice (Class C):-

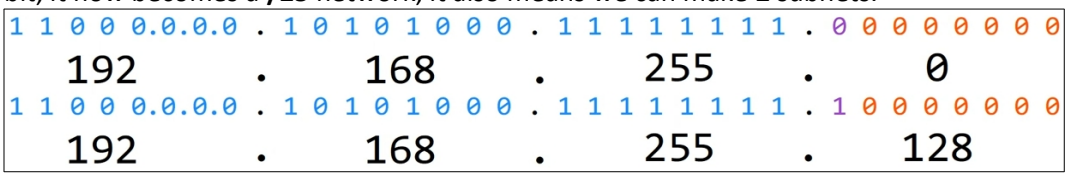
→ **Subnetting Trick:-** On the Last Video QUIZ:
The last bit of the network portion is **64**, this means that to find the next subnet we just have to add **64**.
Add **64** to the first subnet **192.168.1.0/26**, and then we get **192.168.1.64/26**. And then **192.168.1.128/26.** etc.





→ Let's try another similar exercise:
We've been given **192.168.255.0/24** network, and have been asked to divide the network into five subnets of equal size.
In this case, the number of each subnet hasn't been specified, so let's make five subnets that are as large as they can be.
All we have to do is to solve the problem of how many bits we have to *borrow* from the host portion?

- Currently, we don't borrow any bits, so we can't make any subnets, we just still have only one network, **192.168.255.0/24**.
- If we borrow one bit, it now becomes a **/25** network, it also means we can make **2** subnets.

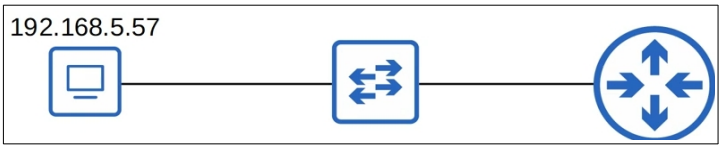


- All of he original network bits, the blue bits, can not be changed. That is the network we received.
- However, the purple bit, the bit we borrowed from the host portion, we can change it! And it can be either **0** or **1**.
- If it's **0**, we have **192.168.255.0/25** network. If it's **1**, we have **192.168.255.128/25** network.
- The formula of the number of subnets is 2^X, and **X** is the number of borrowed bits.

2^x = number of subnets
(x = number of 'borrowed' bits)

- We need **5** subnets, so borrowing one isn't enough.
- If we borrowed **2** bits, **2** to the power of **2** is **4**, and still also isn't enough. The prefix length is **/26**.
- Let's borrow another bit, If we borrow **3** bits, we can make **8** subnets.
- It is more than we need, but if we borrowed **2** bits only and use **/26** mask, we won't have enough.
- So, our first subnet will be **192.168.255.0/27**.
- Using the trick we have learned before, the last bit in the network portion is **32**, So now we should be able to calculate the other subnets, by just adding **32** to the subnet.
- *Subnet2* is **192.168.255.32/27**, *Subnet3* is **192.168.255.64/27**, *Subnet4* is **192.168.255.96/27**.
  and finally *Subnet5* is **192.168.255.128/27**.

→ Another kind of Questions:
Identify the host:
What does host **192.168.5.57/27** belong to?



- So, we have the IP address of the host, but we don't know the network address of the subnet.
- We write the host in binary and dotted decimal, and as it is **/27**, let's show the borrowed bits:

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 1 0 1 . 0 0 1 1 1 0 0 1
   192      .     168     .     5     .     57
```

- The three purple bits are borrowed and added to the network portion.
- Now to find the network address, we just need to change all of host bits to **0**.
- Now the last octet is, **0010000**. Change it back to dotted decimal and we get **192.168.5.32**

```
1 1 0 0 0.0.0.0 . 1 0 1 0 1 0 0 0 . 0 0 0 0 0 1 0 1 . 0 0 1 0 0 0 0 0
   192      .     168     .     5     .     32
```

- So, the host **192.168.5.57/27**, belongs to the subnet **192.168.5.32/27**.

This is a table of the different subnet sizes for Class **C** networks:

Prefix Length	Number of Subnets	Number of Hosts
/25	2	126
/26	4	62
/27	8	30
/28	16	14
/29	32	6
/30	64	2
/31	128	0 (2)
/32	256	0 (1)

→ For the number of Subnets, each additional bit that we borrow doubles the number of subnets.
→ For the number of Hosts, each host bit doubles the amount of hosts, except you have to subtract **2**, for network and broadcast.
→ Take note of **/31**, the number of hosts is **0**. It's because there is only a single host bit, there are only **2** possible addresses ;)
However, for a *point-to-point* connection, you can actually use a **/31** and assign those two addresses to each end of the connection, and have no network or broadcast addresses.
→ Also **/32** technically uses all bits for the network address, allowing no hosts, but you can assign a **/32** mask to identify a specific host when writing routes and such, and in some other special cases!
REMEMBER: The number of hosts indicates the number of *usable hosts*, excluding the network and broadcast addresses.

→ Subnetting Class B Networks:-

→ Looking back to the chart of Network Classes, We can see that there are many more host bits, and therefore many more possible subnets, that can be created with a Class **B** network than with a Class **C**.
HOWEVER, the process of subnetting is EXACTLY the same!

> The process of subnetting Class A, Class B, and Class C networks is EXACTLY THE SAME!

→ Let's start with an example of Subnetting Class **B** network:-
We've been given the **172.16.0.0/16** network. We're asked to create **80** subnets for our company's various LANs.
What prefix length should we use?

→ We can simply use the **2** to the power of **X** formula, where **X** is the number of '*borrowed*' bits.
If we borrowed any bits, we can't make any subnets.
If we borrow **1** bit, we will get **2** subnets, **2** to the power of **1** equals **2**.
This gives us a prefix length of **/17**, and If we write this sunet mask in dotted decimal it is **255.255.128.0**.

> 2^x = number of subnets
> (x = number of 'borrowed' bits)

```
1 0 1 0 1 1 0 0 . 0 0 0 1 0 0 0 0 . 0 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   172      .     16      .     0      .     0
Subnet Mask:
1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   255      .     255     .     128    .     0
```

REMEMBER: When you enter commands in the Cisco CLI, you can't use the **CIDR** Notation like **/17**, you have to enter dotted decimal like **255.255.128.0**.

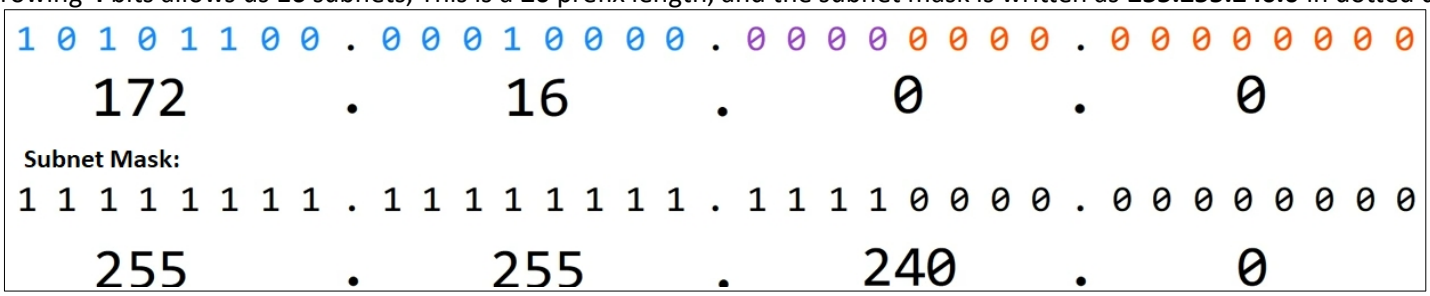
→ **2** subnets isn't enough, Let's borrow another bit:
Borrowing **2** bits allows us **4** subnets. This is a **/18** prefix length, and the subnet mask is written as **255.255.192.0** in dotted decimal.

```
1 0 1 0 1 1 0 0 . 0 0 0 1 0 0 0 0 . 0 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   172      .     16      .     0      .     0
Subnet Mask:
1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   255      .     255     .     192    .     0
```

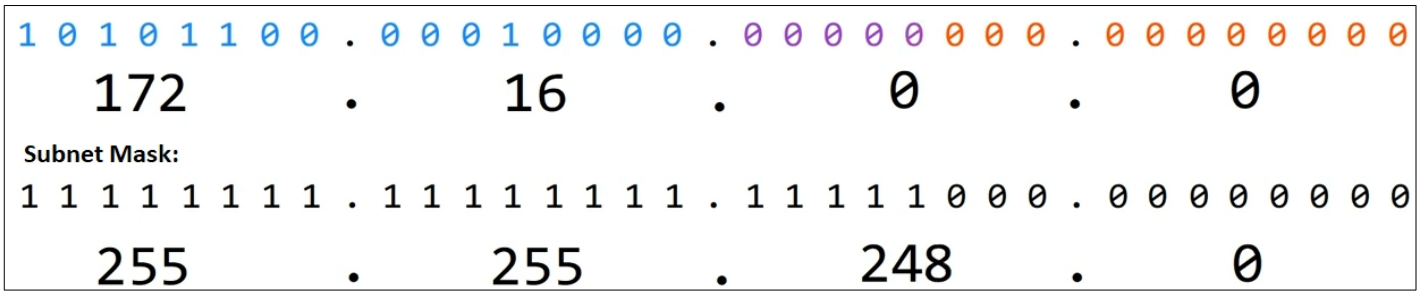
→ Let's borrow another bit:
Borrowing **3** bits allows us **8** subnets, This is a **19** prefix length, and the subnet mask is written as **255.255.224.0** in dotted decimal.

```
1 0 1 0 1 1 0 0 . 0 0 0 1 0 0 0 0 . 0 0 0 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   172      .     16      .     0      .     0
Subnet Mask:
1 1 1 1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 0 0 0 0 0 . 0 0 0 0 0 0 0 0
   255      .     255     .     224    .     0
```

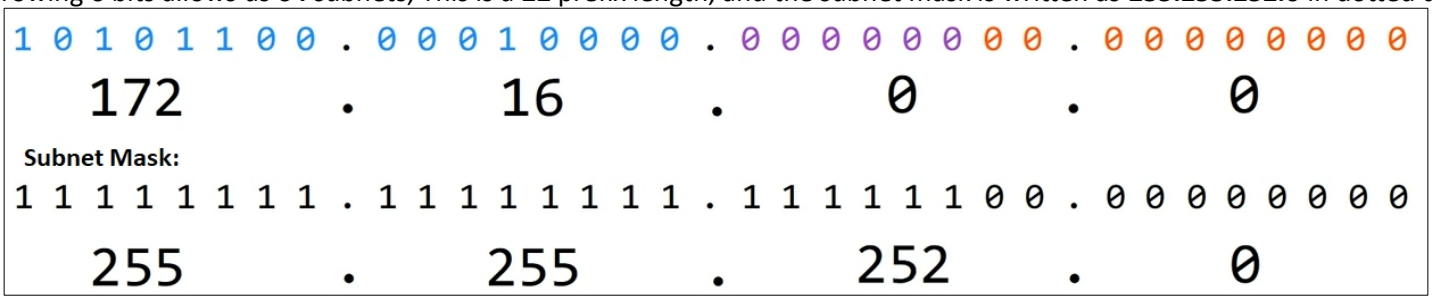
→ Let's borrow another bit:
Borrowing **4** bits allows us **16** subnets, This is a **20** prefix length, and the subnet mask is written as **255.255.240.0** in dotted decimal.



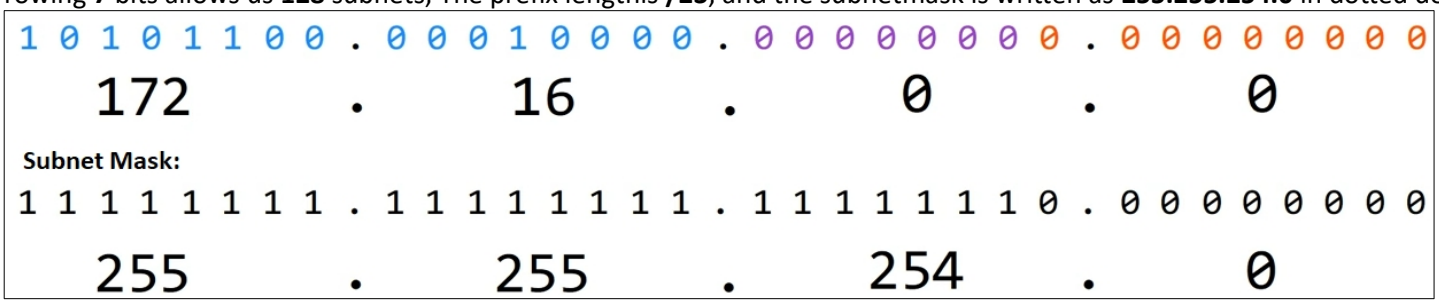
→ Let's borrow another bit:
Borrowing **5** bits allows us **32** subnets, This is a **21** prefix length, and the subnet mask is written as **255.255.248.0** in dotted decimal.



→ Let's borrow another bit:
Borrowing **6** bits allows us **64** subnets, This is a **22** prefix length, and the subnet mask is written as **255.255.252.0** in dotted decimal.



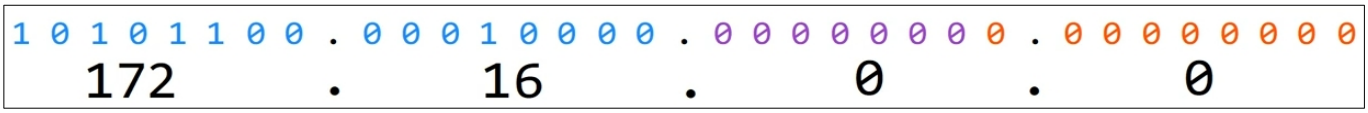
→ Let's borrow one more bit, which should be enough:
Borrowing **7** bits allows us **128** subnets, The prefix lengthis **/23**, and the subnetmask is written as **255.255.254.0** in dotted decimal.



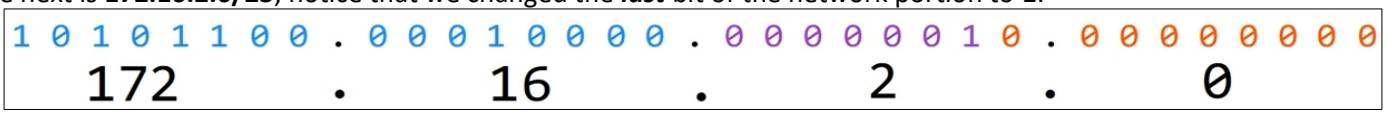
This is the Correct Answer!
We should use **/23** prefix length so we can create the **80** subnets we need.
128 subnets is more than we need, but **/22** only allows for **64**, which is not enough!

→ **Let's look at some of he subnets:**

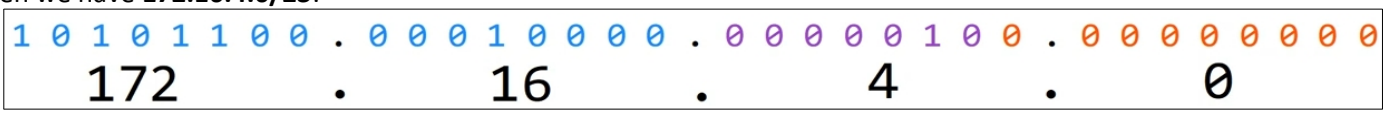
→ The first being **172.16.0.0/23** of course:



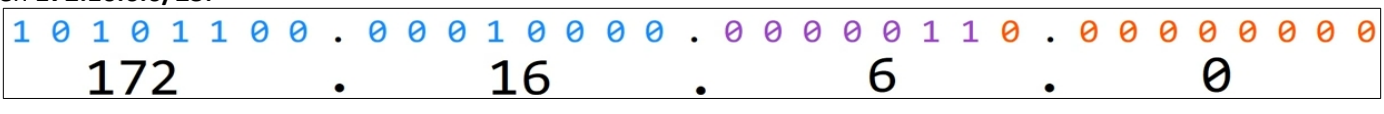
→ The next is **172.16.2.0/23**, notice that we changed the ***last*** bit of the network portion to **1**.



→ Then we have **172.16.4.0/23**:



→ Then **172.16.6.0/23:**

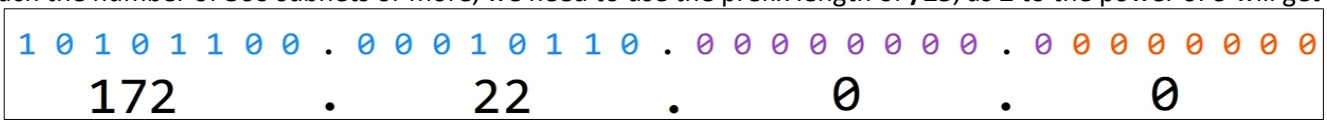


→ Another example:-

We've been given the **172.22.0.0/16** network. We're required to divide the network into **500** separate subnets. What prefix length should we use??

The Answer is:

As the last example, we will borrow a bit every time and calculate the number of subnets we get with the formula **2^x = subnets**. So, to reach the number of **500** subnets or more, we need to use the prefix length of **/25**, as **2** to the power of **9** will get **512**.



NOTE: You can borrow bits even from the last octet, so you can use **/25**, **/26**, **/27**, etc. even with a Class **B** network.

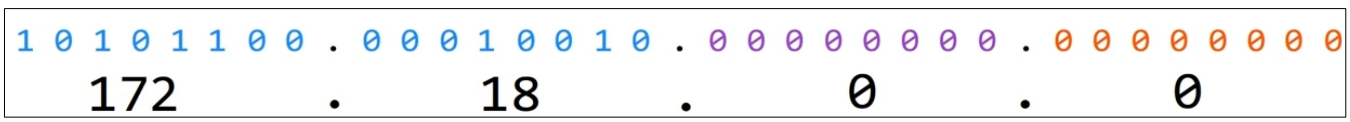
→ Another practice question:-

We've been given the **172.18.0.0/16** network. Our company requires **250** subnets with the same number of *hosts* per subnet. What prefix length should we use??

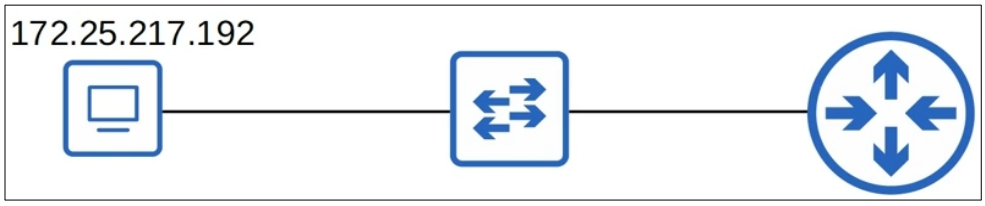
The Answer is:

As the last examples, we will borrow a bit every time and calculate the number of subnets we get with the formula **2^x = subnets**. So, to reach the number of **250** subnets or more, we need to use the prefix length of **/24**, as **2** to the power of **9** will get **256**.

And to calculate the number of hosts we will use the formula of **2^n – 2 = number of hosts**. While **n** is the number of the host bits. So, It will be **2** to the power of **8** minus **2** which equals **254** host per subnet.



→ Identify the subnet:-
What does host **172.25.217.192/21** belong to?



The Answer is:
- Write out the address in binary
- Change all of the host bits to **0**
- And then change the address back to the dotted decimal
- The Answer, by my own is **172.25.216.0/21**.

1 0 1 0 1 1 0 0 . 0 0 0 1 1 0 0 1 . 1 1 0 1 1 0 0 1 . 1 1 0 0 0 0 0 0

172 . 25 . 217 . 192

- Convert the address to binary:

1 0 1 0 1 1 0 0 . 0 0 0 1 1 0 0 1 . 1 1 0 1 1 0 0 0 . 0 0 0 0 0 0 0 0

- Chnage it back to dotted decimal:

172 . 25 . 216 . 0

→ Here's a chart showing the number of available subnets, and the number of available host addresses for each prefix length when sunetting a Class **B** network:

Prefix Length	Number of Subnets	Number of Hosts	Prefix Length	Number of Subnets	Number of Hosts
/17	2	32766	/25	512	126
/18	4	16382	/26	1024	62
/19	8	8190	/27	2048	30
/20	16	4094	/28	4096	14
/21	32	2044	/29	8192	6
/22	64	1022	/30	16384	2
/23	128	510	/31	32768	0 (2)
/24	256	254	/32	65536	0 (1)

- Just know the pattern, for each borrowed bit, the number of subnets doubles, **2**, **4**, **8**, **16**, **32**, etc.
- For each host bit, the number of addresses in each subnet doubles, however you have to subtract **2** to identify the number of usable host addresses.

→ Let's Go to the QUIZ:

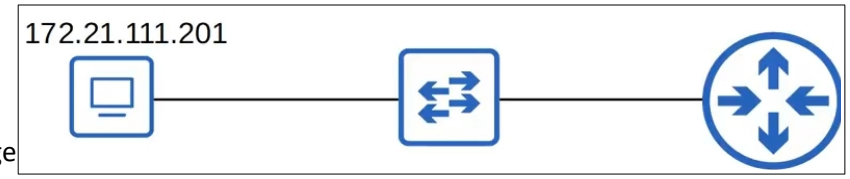
→ Question **1**:
We've been given the **172.30.0.0/16** network. Our company requires **100** subnets with the at least **500** hosts per subnet.
What prefix length should we use??

Answer **1**:

We will borrow a bit every time to reach the number of **100** subnets with the formula 2^x **= subnets**, so we will borrow **7** bits.

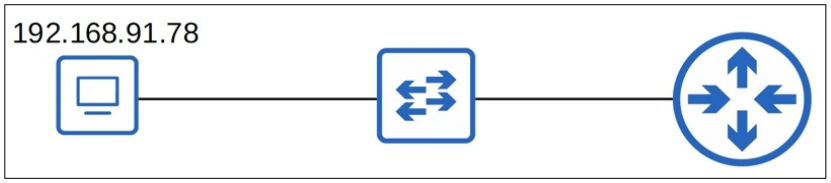
AS **2** to the power of **7** equals **128** bits. And each subnet will have at least **512** subnets due to the formula 2^n **– 2 = hosts**.
So the prefix we should use is **/23**.

→ Question **2**:
What does host **172.21.111.201/20** belong to?



Answer **2**:
We will convert the address to binary, and change the borrowed bits from the binary to **0**, and then we will change it back to dotted decimal. And the host belongs to the network **17.21.96.0/20**.

→ Question **3**:
What is the broadcast address of the network **192.168.91.78/26** belongs to?



Answer **3**:
The subnet address will be **192.168.91.64/26**.
There is also **62** available addresses in this subnet.
And as you know, to get the next subnet, we change the last bit of the network -borrowed- bits to **1**, so the next subnet is **192.168.91.128/26**.
And due to that, the broadcast address is **192.168.91.127**.

→ Question **4**:
You divide the **172.16.0.0/16** network into **4** subnets of equal size. Identify the network and broadcast addresses of the **2nd** subnet.

Answer **4**:
To divide the network to **4** subnets, we need to borrow only **2** bits. So, we will use the prefix length of **/18**.
So to get the network address of the **2nd** subnet, we just change the last bit of the network portion to **1**.
So the network address will be **172.16.0.64/18**. and the broadcast will be **172.16.0.127**.

→ Question **5**:
You divide the **172.30.0.0/16** network into subnets of **1000** hosts each. How many subnets are you able to make?

Answer **5**:
To get the number of **1000** hosts, we have to make the host bits to be **10** bits, As **2** to the power of **10** minus **2** equals **1,022** hosts.
So we will use the prefix length of **/22**.
And to get the number of subnets, as we borrowed **6** bits, so it will be **2** to the power of **6** which equals **64** subnets!

Thanks for everything!
@0x0nullian

Day 15: Subnetting pt.3:-