Knowledge Dump

Dinesh Thogulua
dinesh.thogulua@gmail.com

July 31, 2018

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

System Architecture

1 A GENERIC APPLICATION PROCESSOR

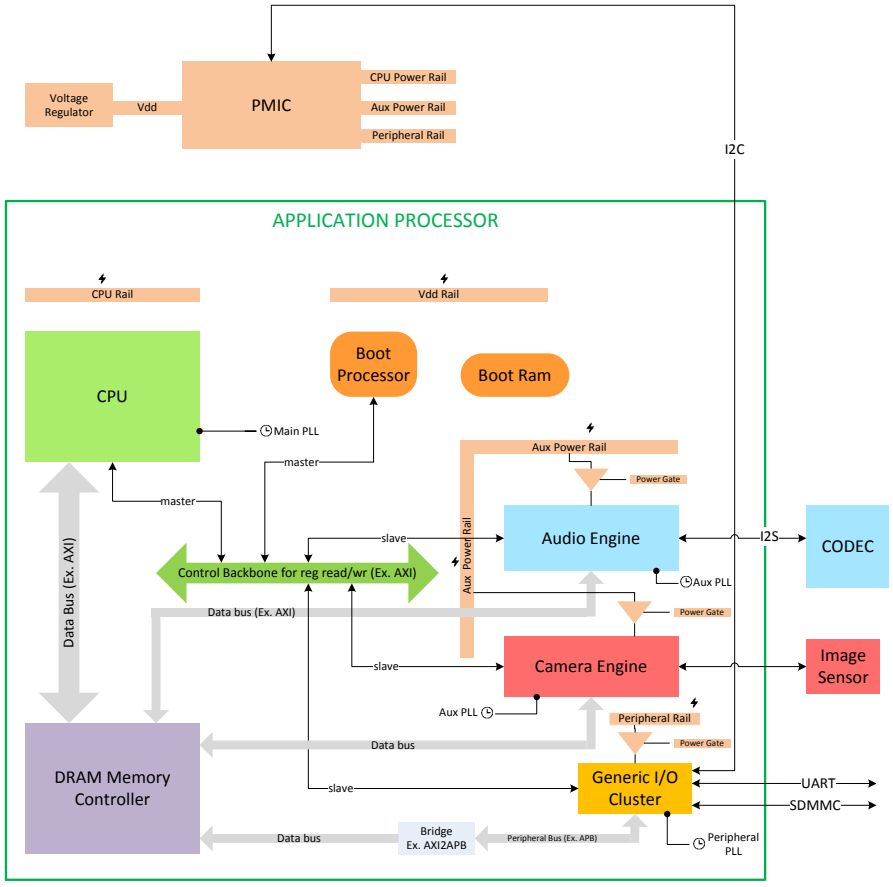


Figure 1.1: A typical Application Processor

A generic application processor (AP) consists of a CPU, Memory Controller (MC), I/Os, and, typically, many auxiliary engines. There could be multiple power rails supplying power at different voltage levels to different parts of an AP. Similarly

there could be multiple clocks supplying clocks of different frequencies to different parts of the AP.

The power rails are all connected to outputs of a Power Management IC (PMIC), which regulates the mains/battery power (a.k.a. Vdd) and has internal mechanisms to divide the regulated mains power to achieve the different power levels required on different power rails. A single power rail could be powering multiple blocks, each containing a "power gate" that can essentially turn that block on or off. The power rails and power gates together give the voltage control granularity required to reduce the power consumption of the AP to the most optimal value. The PMIC itself is controlled via an I2C from the AP.

Power rail levels, power gates can be used to control static power consumption

The PLLs take reference clock from a crystal oscillator and generate integer/non-integer multiples of that clock. Typically, one PLL output is used by several modules, each having a clock divider. These dividers are usually integer dividers or, integer.5 dividers (2, 2.5, 3, 3.5 etc.). Together, the PLLs and the dividers form the "clock tree" of the chip. The clocks going modules can be gated at many levels, including shutting down the entire PLL, primary gating, which gates one output of a PLL, second level clock gating, which is based on certain internal variables of a module indicating functional inactivity. All these can be used to optimize the power consumtion of the AP as a whole.

PLL output frequency, clock dividers, gates can be used to control dynamic power consumption

The fabric (bus, like AXI) that the CPU uses to transact (read/write) data with the hardware registers of I/Os, hardware accelerators, PLLs etc., is called the "Control Backbone". This is usually different from the fabrics used by the CPU, auxiliary engines to transact data with the DRAM (via MC), which can be called the "data paths". The hardware register address space is colloquially referred to as, "Memory Mapped I/O space" or "MMIO space". So the Control Backbone is for accessing the MMIO space, while the data paths are used to access the DRAM address space.

1.1 A NOTE ON OPTIMIZING POWER CONSUMPTION

Optimizing power consumption is an important goal of a system architecture, especially if the system is expected to run out of a battery instead of the mains. There are two components of power consumption in an IC: Dynamic and Static power. Dynamic power consumption is because of loss when transistors toggle and Static power consumption is because of leakage from Vcc to ground of a transistor. If a module needs to run at a higher clock frequency, it would require a higher voltage. The higher the clock frequency, the higher the dynamic power consumption for the same voltage. Similarly, the higher the voltage, the higher the static (a.k.a. leakage) power consumption for the same clock frequency. The OS kernel in CPU usually does dynamic voltage and frequency scaling (DVFS) wherein it uses all the power, clock knobs described in the previous paragraphs to achieve the most optimal power consumption possible for a given use case.

DVFS in kernel is responsible for optimizing power consmption

2

VIRTUALIZATION

In general purpose systems, it is advisable to ensure that different processes or threads running on the CPU do not have visibility into each others stack, heap, global variables etc. This is to prevent malicious code from compromising other processes. Besides this, it is also advisable for each process to use virtual addresses instead of physical addresses for all memory accesses. This is to ensure that processes can be loaded into different physical addresses from one time to another and also to give processes a continguous address map while in reality the various addresses accessed by processes may be fragmented in the physical memory. The OS acts as an abstraction layer to achieve the above mentioned objectives, the umbrella term for which is *virtualization* or *software isolation*. One can extrapolate the concept of virtualization to running multiple OSs on the CPU at the same time (with each running their own pool of processes), wherein, another abstraction layer called a *hypervisor* acts like a OS for OSs - One can think of a hypervisor as an operating system which schedules, virtualizes processes like any other operating system, just that for a hypervisor, a process is actually an OS with a pool of processes. People generally refer to this as many "guest" OSs running on a hypervisor. The necessity for having software isolation in a hypervisor case is exactly the same as that in the case of an OS. From here on, we will just talk about systems without hypervisor. One can easily extrapolate concepts below for the case of a hypervisor.

OS gives virtual addresses to processes; does virtual to physical address mapping

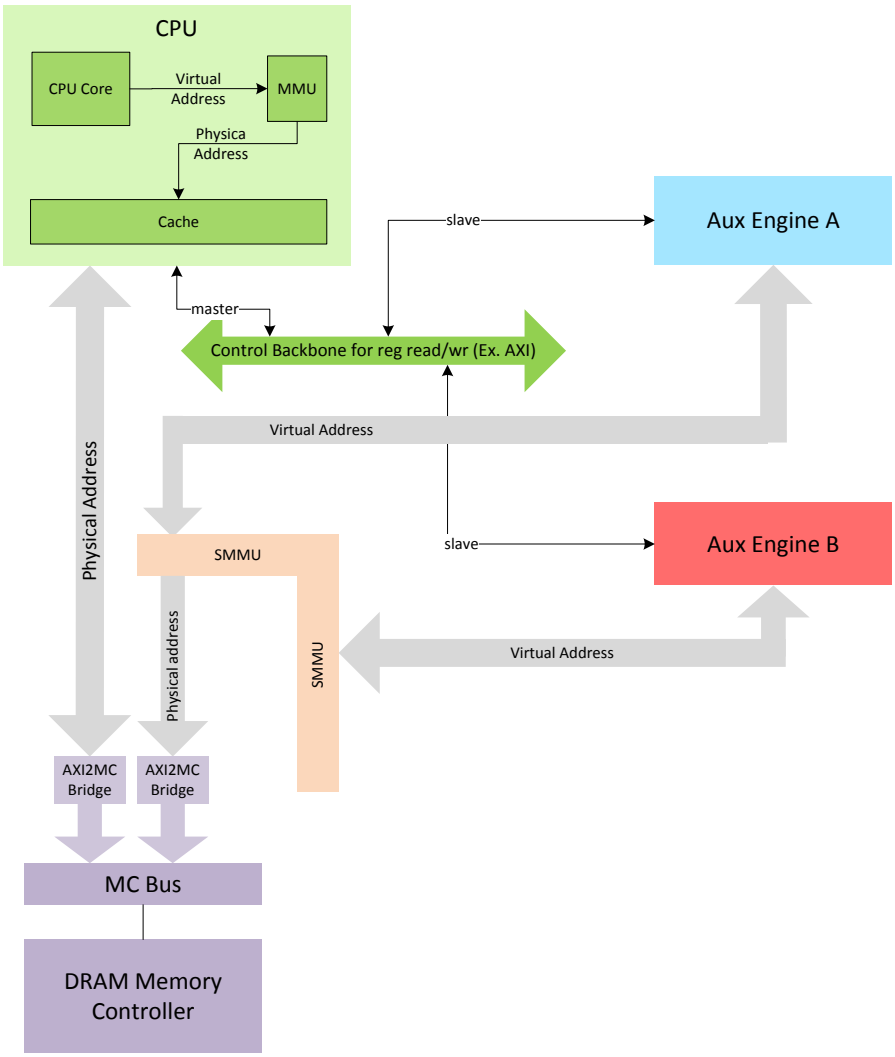


Figure 2.1: Virtualizaton in an Application Processor

2.1 PAGE TABLES

The way an OS achives virtualization is by using *page tables*. When a program is compiled and linked, the resulting *elf*, *bin* or *hex* file contains, besides the code itself, some information about the program embedded in it. This information pertains to the size of code, size of the stack, global variables etc. When the program is requested to be executed, the OS looks into this information, allocates the require amount of memory (which aren't necessarily contiguous). Then the OS creates a look-up table, called the "page table" that has the physical addresses of

Page table contains virtual to physical address mapping

the just allocated blocks in one column and a corresponding set of virtual addresses in another column . Even though the physical addresses may not be contiguous, the virtual addresses always are.

Kernel writes page table to MMU. MMU does actual translation

The page tables themselves are usually stored somewhere in the RAM by the OS. Before a process is run, its page table is loaded into a hardware module called "Memory Management Unit (MMU)". MMU's job is to take the virtual address generated by the CPU core at its input, and converts it into the corresponding physical address at its output. In other words, thought the OS creates the page table, the actual run-time translation is carried out by the MMU. The MMU generates a page fault if a process (possibly malicious) tries to access a memory outside its allocated virtual address range. When a context switch happens from one process to another, the OS, after halting the first process and before starting the second process, changes the page table in the MMU from that of the first process to that of the second. This is an important part of the security architecture provided by virtualization. The page table switch in MMU can be only achieved by a task running with a high level of security privilege. The OS thus runs in this priveleged mode and other processes cannot even access the MMU registers required for changing page tables.

The virtualization architecture involving MMU as described above is a watered-down version of real-word virtualization architectures. Should the reader require in-depth information, please refer to the Cache & MMU section in Microprocessor Architecture.

2.2 HARDWARE VIRTUALIZATION

So far, we have talked about virtualization only from the point of view of software codes accessing the memory. But in an application processor, many hardware modules also access the memory. Some examples are DMA engines managing data traffic of I/Os and auxiliary CPUs. If a malicious code accesses, say, a DMA engine, and uses the engine to access memory that it otherwise cannot (because MMU will prevent it), then security will be compromised. This may sound unlikely at first - Afterall, a user mode program cannot directly access any hardware, including a DMA engine and it has to go through the hardware's driver which is part of the OS. Furthermore, any program would do a malloc to get a buffer for the DMA. The malloc itself goes through the kernel and the kernel will ensure that the address allocated is within the range of that program's legal range. The base address returned would of course be a virtual address. When the DMA is setup by the program, it uses the DMA driver which translates the virtual address to the physical address and gives it to the DMA engine. So all in all, it may sound like security cannot be compromised by a malicious code by misusing a hardware module. However, while the driver may not be checking whether the size of the DMA buffer exceeds the legitimate address range of the program. Many other complex scenarios like this are possible. Hence it is importnat to have a degree

of control over the addresses accessible by hardware modules as well. This is where an SMMU (System MMU) comes into picture. SMMU is very similar to an MMU - just like MMU does a virtual to physical address mapping using page tables that are different for different user mode processes, the SMMU maintains a page table for every hardware MC bus master. Thus, the DMA driver will, instead of translating the virtual base address from the user mode program to a physical address, it will translate it to a different virtual address and give this to the DMA engine.

SMMU is for hw what MMU is for SW

3

DVFS

Dynamic Voltage and Frequency Scaling is essentially the mechanism used by a system to ensure different components of the system (and hence the entire system itself) are consuming power optimally. The data transaction capacity of buses and I/Os and the processing capacity of other hardware modules will depend on the clock frequency at which the module is operating. For instance, a 16-bit bus, if clocked at 1 MHz, cannot carry bits at a rate faster than 16 Mbps. Similarly, a filter module requiring 5 clocks to process one sample would require the clock frequency to be at a minimum of $fs * 5$, where fs is the sampling frequency of the signal. While this is said about the clock frequency, transistors cannot be asked to toggle at any clock frequency at a fixed $V_c c$. Because a trasistor's transfer function has decreasing magnitude response with increasing frequency, there would be a point in the clock frequency after which the transistor voltage will have to be increased so that its ouput can be differentiated as a 1 or a 0 by the following transistor. Thus, one can think of a series of voltages and an associated range of clock frequencies supported by each voltage level. DVFS is all about fixing the clock frequency of a module, and hence, the voltage of that module, for a given use case.

It is normal for a hardware module to share a voltage rail with many other hardware modules. So, for a use case, the DVFS mechanism will choose a voltage level for a voltage rail to be the maximum of the voltage levels required for the different modules connected to the rail. The case for clock frequency is somewhat similar - one PLL may be supplying clock to many hardware modules and its output clock is chosen to be the maximum of all the clock frequencies required for different modules sourcing clock from it. There is, however, a small difference - usually every hardware module has a clock divider that divides the PLL clock to obtain the required clock. This is because clock dividers are cheap. Thus the DVFS has the choice of changing the clock divisor for a particular module instead changing the PLL output. Power dividers, on the other hand, aren't cheap - PMICs are generally costly and their cost increases with the no. of rails they can support. So it is normal that DVFS has a fine degree of control over the clock frequency, but a much coarser degree of control over the voltage of a module.

DVFS has fine degree of clock control, coarse voltage control

3.1 SYNTHESIS OPTIONS FOR A MODULE

In the first paragraph of this section, it was mentioned that a transistor has a transfer function that has decreasing magnitude response for increasing frequency. While this is true, it is not necessary for the entire chip to be made of the same type of transistor with the same transfer functiion - one can choose to synthesize one module with a particular type of transistor and another module with a different type of transistor. Usually transistors are broadly classified into different "cell types", such as HVT, SVT, LVT cells. The SVT cells can run at higher clock frequencies for a given Vcc than the HVT cells and the LVT cells can run at an even higher clock frequency for that $V_c c$. However, the penalty for choosing to synthesize a module at SVT, rather than HVT, is leakage power - SVT cells leak more than HVT cells (typically 25 - 65 times more). One may ask, "what is the point of going to an SVT instead of an HVT"? SVT may require a smaller voltage for a given clock frequency as compared to a HVT, but it doesn't necessaily give us any power benefits as the leakage of SVT is higher than that of HVT. The answer for this question could be simple - it may be the case that the module in question is small in area and that we care more about the dynamic power consumption, which is directly proportional to the clock frequency, than its static power consumption, which is proportional to its area. However the answer could also be complex - the module in question may be connected to the same power rail as 10 other modules and, for a use case, the module in question may need a much higher clock frequency than all other modules under that power rail. Were we to synthesize the module in question at HVT cells, that would increase the rail voltage for 9 other modules. The collective power consumption (static+dynamic) in this case may be much higher than the case where the module in question is synthesized at SVT, and hence its higher clock frequency required doesn't push the rail voltage requirement up. Of course, these calculation would vary wildly from one module to another, one use case to another. Beyond this, one needs to also consider the likelihood of the use case in question. Basically there is no single formula for choosing the cell type for a module. It is the job of the hardware architect to do the required analysis and prescribe the cell type of a module.

Choosing right transistor for a module may involve module level, system level, use case level analysis

Besides choosing a cell type, one gets to also choose a finer granularity in transistor type. In other words, when a cell type is chosen, we would have only selected a class of transistors. Thus, even if we choose HVT as the cell type for a particular module, we still will have to choose, among all the transistors in the HVT class, which exact transistor to choose - within the HVT class, one can find one transistor to be capable of running at a slightly higher frequency at the same Vcc than a different transistor. Usually, it is very straightforward to choose the cell type for a module - normally CPUs would required faster cell types than, say, I/Os. The real job of the hardware architect then is to choose the correct transistor type within the selected cell type.

3.2 REAL LIFE METHODOLOGY USED FOR SYNTHESIS

The above paragraphs may lead you to believe that the HW architect chooses the transistor type for a module directly. However the reality is different. Usually the hardware engineer first determines, among a set of use cases, which ones should be “low voltage(LV)” use cases, which ones should be “medium voltage(MV)” and which ones should be “high voltage(HV)”. For example, a 16KHz stereo music playback (without video) may be determined as a low voltage use case, while a movie playback with 5.1 audio at 96 KHz may be chosen as a high voltage use case. After determining the “voltage type” of the use case, say LV, for every module that is run for that use case, the clock frequency at which it needs to run for that use case is noted as the “LV corner” frequency for that module. Thus, after classifying all the use cases into different voltage types and devolving the voltage type on the modules run during those use cases, we would have a “LV corner”, “SV corner” and an “HV corner” (frequency values) for every module. This then is fed to the synthesis tool. The synthesis tool takes the preferred cell type as an input and selects the appropriate transistor type within that cell type for a module based on it LV, SV, HV corners - it basically chooses the transistor that will give the best power consumption based on these corner frequencies. If the synthesis tool cannot find a transistor type that satisfies the given corners for the preferred cell type, it will ask the user to choose a different cell type. So one starts with HVT cells and if the synthesis tool gives up, moves to SVT cells and so on. Remember that, with changing IC process (nm technology), the leakage characteristics of different cells and different transistors within the cell change - so it is humanly impossible for a hardware architect to manuall choose the transistor type for a module. Hence, synthesis tools are utilized to do the actual selection.

4 MICROPROCESSOR ARCHITECTURE

4.1 CACHE AND MMU

Cache is a RAM that CPU has dedicated access to and which is used to temporarily store copies of data in main memory. The basic idea of a Cache is described in Miscellaneous chapter (part: Hardware), section "Memory". Here we assume the reader knows the idea and purpose of caches and focus on topics such as the working mechanism of cache, architectural options etc.

4.1.1 Basic Cache Operation

When a CPU issues an address for a read operation, the *cache controller* checks if a copy of the data in that address (in the main memory) is available in the cache. If it is, then we have a *cache hit* and the cache controller simply gives that data to the CPU without the need to actually fetch the data all the way form the main memory. If a copy of that data isn't available in the cache, then we have a *cache miss* and the cache controller fetches the data from the main memory, gives it to the CPU and also store the data in the cache, so that, the next time CPU accesses the same address, we will have a cache hit. This is how data frequently accessed by the CPU (within the context of a function or a bunch of functions) build up in the cache resulting in less and less cache misses, and proportionally less accesses to the main memory.

Compilers usually allocate memory contiguously for variables inside a function. And a function that is accessing a local variable one cycle is most likely to access another local variable within the next few cycles. So it make sense that, if a cache miss happens when the first variable is asked for, we fetch not just that variable, but also data from nearby addresses which likely belong to other variables that are soon to be accessed as well. This will reduce the likelihood of cache misses overall. A key idea here is that, while CPU is consuming the first variable, the cache controller is fetching the nearby address in parallel. And just as the CPU finishes consuming the first varaible and issues an access for nearby variables, we already have them in the cache. The amount of data fetched by a cache when a cache miss happens constitute what is called a *cache line*.

Let us take an example cache of 4KB size that serves a main memory of 4GB and see how the operations described above would work in this case. Mapping

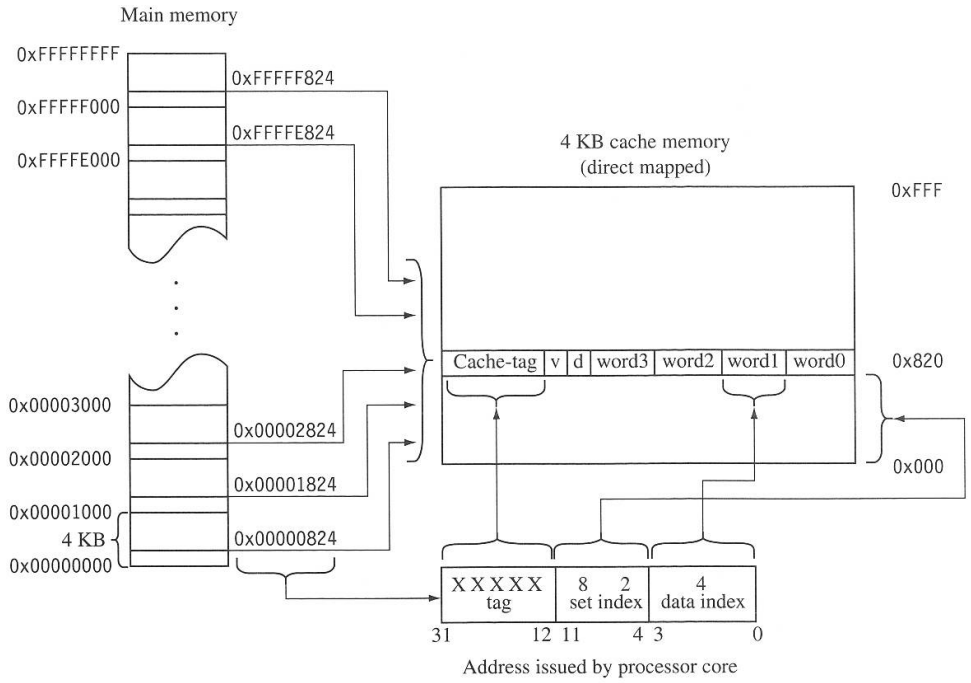


Figure 4.1: Basic Cache Operation

the 4GB address of the main memory to the 4KB address of the cache means that several addresses in the main memory will inevitably overlap in the cache. One way to do this is to simply divide the main memory into chunks of 4KB each and let them all overlap in the cache - this is equivalent to using only the least-significant 12 bits of the address issues by the CPU. Our example here uses this method. When the CPU issues a memory access, the cache controller initially ignores the most-significant 10 bits and uses the remaining 12 bits to look inside the cache (12 bits address equals 4kB, which is the cache size. So we are good). But when it reaches the cache line in that 12-bit address, it checks its *cache-tag* which contains the most-significant 10-bits of the main memory address to which the data in that cache line belongs to. So the cache controller checks the cache-tag and if the cache-tag does match the most-significant 10-bits of the address issued by the CPU, we have a cache hit. Otherwise we have a cache miss. If a cache miss happens, the cache controller will feltch, in our example case, 4-words of data within the same 4-word-boundary as the address issues by the CPU from the main memory[1].

[1]Note that the cache controller will fetch the data in the address issued by the CPU first and only then fetch the rest of the data within the same 4-word-boundary, no matter where the CPU issued address is within that 4-word-boudnary

The data thus fetched will replace the data in the cache line and the cache-tag will be promptly updated to contain the 10 MSBs of the address issued by the CPU.

Note that, in our example, instead of the cache controller looking for the LSB 12 bits of the address in one shot, it uses an 8-bit *set index* first, looks up the cache line associated with it, check the cache-tag and only if there is a cache hit, will go look into the exact word within that line (or the 4 LSBs of the address). The idea of a set index is a natural consequence of the size of the cache line - when you know that the cache always picks up 4 words of data in one shot, then there is no meaning in checking if the exact word being asked is available - for it will be available as long as we have a cache hit for the entire 4-word-boundary.

So far, we have talked about how cache works for a CPU read. But a cache is also used for a CPU write - as much as we want to reduce the read traffic to the main memory from the CPU, so much we want to reduce the write traffic to the main memory as well. Imagine a scenario where a function running in the CPU reads a variable - we have seen how the copy of the data associated iwth this variable's address in the main memory will be made available in the cache. But what happens when the function manipulates that variable - there will be a CPU write and the data in the cache will be changed. Now the data value of the variable in the cache and in the data memory are different! Does that present us with a problem? Not necessarily and not right-away. As long as no other component of the system (a hardware accelerator or a DMA engine) is not accessing the same address location (of the variable in question), we are OK. If it is only the CPU that is going to continue to use that variable, further read accesses to that variable will only use the data in the cache, which is anyway fresh. However if the scenario is about to change - that a hardware accelerator now needs access to the new value of that variable, the CPU will have to issue a *cache flush*. A cache flush is meant to write back the updated values in the cache to the main memory. But how does the cache controller know which of the values in the cache were updated by the CPU and which ones weren't? - it doesn't make sense to write back all the values in the cache to the main memory because that would be such a waste of bus bandwidth. This situation is tackled using the *dirty bit* indicated by the letter "d" in the above diagram. This bit is set everytime a word associated with the respective cache line is written to by the CPU. So when a cache flush is issued, the cache controller only writes back the cache lines that have this bit set.

The *validity bit* is used for the exact opposite purpose as the dirty bit - Imagine a case when some component in the system, say a DMA controller associated with an I/O, writes into the main memory. Let us assume that the cache controller had acquired a copy of the data in that address location into the cache some time back. This means that the data in the cache is now stale and needs to be updated with the fresh data from the main memory. In this case, the CPU invalidates the cache [2], i.e.,

[2] let us assume CPU came to know about the DMA writing into the main memory via a DMA completion interrupt

sets the "v" bit to zero. And when the CPU access the invalidated address locations the next time, cache controller fetches data from the main memory instead of giving the stale data to the CPU. Again, notice that, just as in the case of the CPU read, we are note interested in updating all the invalidated cache locations with fresh data - that would be wasteful of bus bandwidth - but only those which are later accessed by the CPU.

This section is under construction. Key words: Cache line (with valid and dirty bit), set association, cache, TLB architecture - http://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/22-vm3-i.pdf

4.2 MEMORY MANAGEMENT UNIT

This section is under construction. It will contain information on TLBs, what happens in MMU during context switching etc.

4.3 FABRIC

Even the most basic application processor will at least contain a CPU, a DMA engine, a Memory Controller and some I/O Controllers (if not any hardware accelerators). The way these talk to each other is through a fabric: Essentially AXI, APB, AHB, PCIE interfaces from these modules go to a interconnect switch which arbitrates, routes communication between pairs of interfaces. CPU and the DMA engine, in our example, will be Masters on the interconnect, whereas the Memory Controller and I/O Controllers will be slaves. Only the masters can initiate a communication (mem read/write, reg read/write) and the slaves can only respond. The traffic in the fabric can be broadly classified into MMIO access (basically register read/write) and Memory access (basically DRAM or some SRAM). The masters have different priorities. In case two masters try to access the same slave at the same time, based on their priority level and possibly the traffic type (and may be even some other parameters), the inconnect switch will decide which master will get the link to the slave.

4.3.1 Switch

Let us take a simple Application Processor as shown below. Let us use this as an example to explore the functionality of a switch in detail.

Basically the CPU can access the system memory via an AXI and configure the rest of the hardware (MMIO space) in the system using an APB multi-port Bridge [3]. The DMA engine (DMAe) supplies data from the system memory to I/Os and

[3] An APB Multi-port, uses a single APB bus connecting all the hardware (all the hardware module's basically evesdrop on the bus's signals) and the APB Bridge drives the signals on the line (seen by all hardware), but raises the chip select of only one of them. This way we can avoid having each hardware hooked up to an independent APB bus which will increase the wire count of the

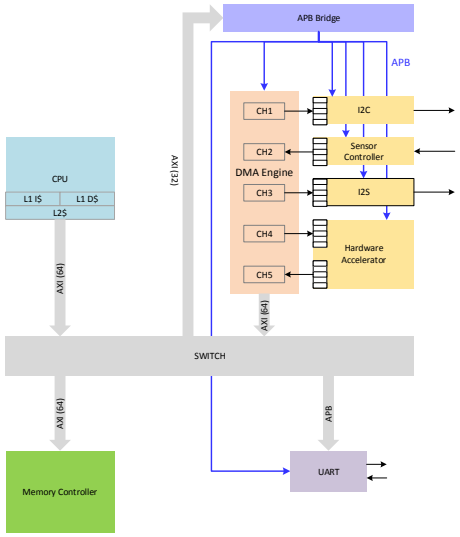


Figure 4.2: A Simple Application Processor

hardware accelerators without CPU having to read data and write to the hardware one-by-one (with data FIFOs of hw sitting in the MMIO space).

Now the switch has two functionalities: Arbitration and routing. Let us assume that CPU has higher priority that DMAe - so the arbitration in our simple AP case is...well simple. Next, to know which slave to route a master's request, we need to deal with the important concept of the *Address Map*. But before we go into the Address Map, we need to remember a key point: The masters have no idea in which slave a memory address is located. So their read/write requests only contain the destination address and the destination slave - It is entirely the job of the switch to figure out which slave a request should be routed to.

Address Map is simply a division or allocation of available addresses in the system to various slaves. Let us say that we decide that the system address will be 32-bits wide [4]. This means that the AXI buses coming out of masters and going to slaves will have 32-bit wide address signal. Anyway, this 32-bit address width implies a 4GB address space. Let as assume that (for whatever reason), we decide on the following address map.

Thus DRAM gets two address *apertures*[5], while MMIO and UART get one aperture each. This address map is known to all the masters as well as the switch itself. So, for instance, the CPU (OS) knows that data memory is either in DRAM space 0 or 1 and hence allocates stack and heap only in those areas. And the switch

system.

[4] Address width is determined at architecture time. It doesn't have to be a power of two. 37-bit, 26-bit address widths are not unusual

[5] An aperture in an AMAP refers to a contiguous range of addresses

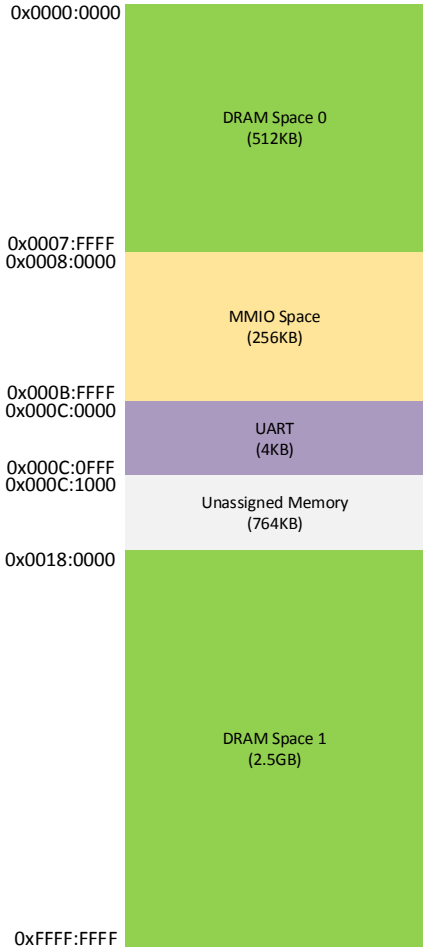


Figure 4.3: An Example Address Map

uses the address map to figure out to which slave it should forward a master's request to based on the destination address bits in that request.

The switch considers all the addresses within the address map to be legitimate unless the address map explicitly mentions an area as unassigned (as in our example). In the case of the master's request containing an invalid (unassigned) address, the switch will respond with an address decode error. Otherwise it will forward the request to the next level, which can be a bridge (like the APB Bridge) or a end client itself. Now it is the responsibility of the bridge or the client to tell the master if an address is illegitimate: Even if the address map shows a contiguous MMIO space, it may have large unassigned address ranges within it. In fact, even going further, a module that occupies a set of addresses within the MMIO region may have empty space it reserved for future expansion. In the former case, the

APB Bridge (in our example, that is the only MMIO master on the MMIO APB bus) will respond with decode error. In the latter case, the end module will.

One may wonder why the DRAM space is split into two apertures. Though it was done just to let the reader know that one slave can have two apertures assigned to it, it isn't uncommon to find such an arrangement in real-life systems. The reason is that, some CPU, for security reasons, don't allow accesses beyond 1GB (or some arbitrary value) of address space during boot time. So it becomes necessary to put all the addresses one may need during boot time within the 1GB.

The address map tells the switch to which slave a master's request should be forwarded to. However, it doesn't tell the switch whether to forward that request at all or deny access. Often it is useful to have switches restrict access of certain masters to certain slaves for security reasons. For instance, the system architect knows that the DMA Engine has no business accessing the MMIO space. So s/he may want to switch to not allow a DMAe request to be forwarded to the APB bridge even thought that request contained a legitimate address according to the address map. This is because hackers could exploit an unrestricted access and use DMAe to make a chip to hang.

These address restrictions are encoded in a *Connectivity Matrix* as shown below

Connectivity (MxS)	Memory Controller	APB Slave	UART
CPU	Yes	Yes	Yes
DMAe	Yes	No	No

Table 4.1: Connectivity Matrix

4.4 ISSUES RELEVANT TO FABRICS

4.4.1 Access Restrictions

Access restrictions don't stop with the connectivity matrix of a switch in a Fabric. It only gives an extremely coarse level of restriction. It is possible that we don't want a slave to be accessed by master at some times, but not at other times. For ex., we don't want hackers to overwrite some program memory area - In this case unless the CPU master's access originated in the SW from the OS (and not user's code, which could be hacker's code), we should not grant it access to certain memory areas. Going even further, we may want another level of security, whether, even if the access was generated by OS, if it was not in a secure mode (like ARM's Trust Zone mode), we may not want to grant access to, say, the region in DRAM that stores digital signatures, private keys etc. These additional levels are restrictions are implemented not in the switch, but in the slaves themselves. AXI protocol has three *PROT* bits that are set by the master depending on which mode the SW

that generated the request was in. In DRAM, there could be a bunch of address (sub)apertures that have different levels of access restrictions implemented: DRAM may grant access to some apertures only if the PROT bits indicate a secure access, and some to secure or privileged access etc.

While the PROT bits are native to the AXI protocol, it is possible that the system architect may want a different type of security. For ex., s/he may want to DRAM to differentiate between a privileged access coming from a certain master from the privileged access coming from certain other master. In this case, though AXI won't support it natively, s/he can use AXI's generic *user bits* (AWUSER). This can be made to carry, say a 4-bit custom master ID - In hardware, at the AXI interface between a master and the switch, the wires associated with user bits can be pulled up or pulled down to create a master ID pattern that is essentially hardwired into the system (and hence impossible to defeat).

4.4.2 Latency

Latency is the delay seen by a master from when it puts out a request to when it gets back a response. Imagine a human interaction situation in which a news anchor is talking, over a satellite connection to a field reporter. We often see that it takes several seconds for the news anchor's questions to reach the field reporter. And the field reporter processes the question in her brain and replies. Similary in a network, when a master puts out a request, it may be delayed due to arbitration at the switch(es), arbitration at the slave and also slave request processing time. In the human interaction example, to tackle the latency, the best strategy is for the news anchor to ask many questions in a row - so while it takes some time for the first question to reach the field reporter, she can process several questions in a row and offer a lenghthy response. This strategy works as long as the news anchor's one question doesn't depend on the field reporter's answer to his previous question. This plays out the same way in a network. Thus, as long as requests are independent from one another, the master may choose to send many of them back to back instead of sending each request only after the previous request received a response.

In our example AP, the DMAe has to provide the I2S samples isochronously so that the I2S never runs out of samples to send out to the external world. Imagine that the data rate of the I2S is 48KHz and its samples' size is 32-bit. Let us also assume that DRAM word size is 32-bits as well to make this illustration simpler. Let us assume that the bus protocol or DRAM restriction allows a single request to upto 16 words and no more (i.e., a single, say, AXI request can indicate that it wants 16 words of data). And let us assume that the network max latency is 1ms. For simplicity, let us further assume that the DMAe has a big FIFO in each of its channels and that the I2S has no FIFO of its own. In this case, a good strategy for the DMAe would be to first make a 1ms worth of requests every time its internal memory is touches a lower limit of 1ms worth of samples. In other

words, at the very beginning, the DMAe would make three requests, each for 16 words to the DRAM (3*16 = 48 samples = 1ms worth of samples). Now when it receives back-to-back responses from the DRAM, it fills its internal buffer with these responses, i.e., 48 samples. But immediately this value is at the lower limit level of 48 samples. So as soon as it receives all the responses from the DRAM, it makes another three 16-word requests. Suppose this second set of requests get their response within 0.5 seconds (since 1ms is only max latency), its buffer will now contain 1.5ms worth of data (I2S drained 0.5ms of data in the mean time..). This time, the DMAe doesn't have to make its third set of three 16-word requests right after it got the back-to-back responses from the DRAM. It can wait for 0.5ms (at which point, its FIFO will have 48 samples left) and make the next set of requests.

Remember that, in AXI, the request channels and response channels are completely independent. So the fabric can absorb requests coming from a master (by raising 'ready' signal) into an internal FIFO somewhere (and then forward the request to the slave when its time comes), but not respond. Think of it as packets in a computer network: When the home computer makes a request to the google server, those packets are accepted right away by the switch and then by routers, but only the google server responds - the switch or router don't. And while the packets to google server are in transit, the home user can make a separate request to some other site without waiting for google server's response.

Coming back to our DMAe's strategy of sending out 3 requests in a row, this means that somewhere in the fabric, the DMAe must have for it 3 requests worth of buffer available. This could be at the point where the DMAe connects to the switch or the point where the switch interface to the DRAM originates. In the former case, we could simply have a 3-request buffer. In the latter case, since other masters would also be accessing DRAM, the buffer needs to account for DRAM as well as other masters. This concept of multiple back-to-back requests is called, "maximum outstanding requests"

4.4.3 Ordering OOO replies

Memory Controller, as a rule of thumb, never responds in the same order in which it received requests causing the phenomenon of *Out of Order replies*. This is because, it sees that the earliest arrived request requires it to open a new memory bank that the one that is already open, whereas a later arrived request is to the already-open memory bank. Frequently switching between memory banks is wasteful of power and also has latency implications.

For some masters, in some cases, it is OK to receive OOO replies. For example, a DMAe requires all requests originating from a particular channel be responded to in order - but it doesn't require, say, two requests coming from two different channels be responded to in order. Thus we need some way of differentiating between the set of requests that require in-order responses and those which don't

care. In our example AP, the fabric uses AXI IDs (AWID field) - requests that have the same AXI ID need responses in order. But requests that have different AXI IDs can be responded to OOO.

At this point, it is a good idea to understand the idea of AXI IDs a little more.

- AXI IDs are not associated with masters and are associated with transactions, but are not necessarily associated with slaves. So two requests going to the same slave could have two different AXI IDs.
- One master can use multiple AXI IDs. But the method used by a master that determines what AXI ID to use for a particular request is completely up to the master and is implementation specific
  - For ex., a DMAe can be implemented to use one AXI ID for each of its channels. A CPU can decide to use one AXI ID per thread.
  - A master may choose to use different AXI IDs for different slaves. But this is matter of strategy and hence doens't negate the top point in this list. An example is to avoid head-of-line blocking of requests associated with fast slaves by requests associated with slow slaves. More about this can be found in later subsections.
- Two masters cannot have use the same AXI ID. AXI IDs are thus allocated by the system architect to different masters - Some masters may get many. Some may get few. Some may even get just one AXI ID.

Coming back to OOO replies, it becomes the responsibility of the fabric to re-order OOO replies from the Memory Controller for requests that require in-order responses. So the fabric has to do some sort of book keeping of (AXI ID, destination address) pairs (near the point where the AXI to slave goes from the switch) and when the replies come from the MC, re-order as necessary.

4.4.4 Head-of-Line Blocking

Head-of-line blocking is a generic term which means that a head item in a FIFO is in someway restricing an item behind it in the queue. A human interaction example would be when a driver is waiting at the intersection in a single lane (one lane on both sides) road to make a right turn (i.e. in India). He needs to wait for the opposite side traffic to have a moment of pause before he makes a turn. But because it is a single lane road, all the drivers behind him have to wait for him to make the move, even though the others are thru traffic - In other words, even though the way is clear for most drivers, they still can't move because the way isn't clear for the head driver in the queue.

The following scenerios of head-of-line blocking issue are common in fabrics.

- A master's request to a readily available slave is waiting because another to non-available slave is blocking it at the head of the queue
- A CPU thread is blocked at the head of the queue by a non-available slave, but other threads accessing other slaves get their requests blocked behind the blocked thread's
- CPU requests to fast responding slaves is blocked by requests to slow responding slaves (with all the switch buffers for outstanding requests filled by requests to slow slaves)

The strategy for tackling head-of-line blocking involves the clever use of AXI IDs (as we did with OOO replies earlier). At the master side of the switch, one could have different sets of outstanding request FIFOs for different AXI IDs. [6]. Basically our strategy of unblocking the queue is not to have a single queue in the first place! i.e., have different queues for different AXI IDs. So the afore mentined scenarios can be resolved using the following AXI ID allocation strategy respectively.

- Use different AXI IDs for different slaves
- Use different AXI IDs for different threads
- Use one AXI ID for fast slaves and one for slow slaves

Of course, it is not possible to give one set of FIFOs for each AXI ID if the no. of AXI IDs used by a master is too many. So the system architect judiciously decides on how many sets of FIFOs to give to a certain master, and which AXI IDs go to which FIFO (so several AXI IDs could go to the same FIFO).

[6] "sets" of FIFOs, because AXI has 5 channel and supporting, say 4 outstanding requests, means having not one 4-deep FIFO, but 5 4-deep FIFOs - i.e., one FIFO for each channel

5 MISCELLANEOUS

5.1 COHERENCY

When there are many processors in a system (think of a CPU cluster with 4 CPUs), each CPU will have its own (L1) cache containing independent sets of cache lines. But if one CPU updates a cache line and then a second CPU accesses the physical memory corresponding to it, the coherency mechanism (hardware module) detecs that the second CPU is accessing stale memory. This coherency mechanism then does a cache flush of that cache line from the first CPU and also invalidates that cache line in the second CPU (so that it's cache DMA will automatically go fetch the freshly updated memory content). Note that, while each of the CPU cores will have its own L1 cache, there is usually an L2 cache shared by all the cores. So whena coherency needs to be done between two CPU cores, a L1 cache flush of one core doesn't necessarily have to update the physical memory in the DRAM, but could simply update the L2 cache. Exactly which method is used will depend on the architecture of the coherency mechanism

I/O coherency reference to something similar when a DMA is involved. If someone sets up a DMA to transfer data to an I/O from a mem location, if the DMA is I/O coherent with the CPU, it means the coherency mechanism will come to know that DMA is going to fetch from a memory location - and if that location is stale, it carries out a cache flush of the corresponding cache lines (which were recently updated) of the appropriate CPU. It works the other way around as well (in case DMA is sending samples from I/O to mem).

5.2 BOOTING

Booting is about setting up an application processor for loading the OS (kernel). In an application processor, either the main CPU or an auxiliary CPU can be used for booting - For now, let us assume that there will be a separate boot processor. At power up, the PMIC has the only the Vdd rail ON and other rails will be OFF. The PLL that supplies clock to the boot processor must be on the Vdd rail so that it can start supplying clock to the boot processor right after power up. The boot processor and the boot ROM that the processor executes data from need to be on Vdd as well. The first thing the boot code does is to power up (and enable clocks to) some local boot RAM, copy the rest of the boot ROM code on to that RAM and

then execute the rest of the code from the RAM - Usually ROM r/w is pretty slow and this is why this step is done. After this point, the boot code turns on power rails, clocks, configure pinmuxes and do whatever is required to read from the bootloader medium. This may be an SD card. Thus the boot ROM only contains minimal code required to load the bootloader, which then takes care of rest of the boot loading. The rest of the boot loading is again about turning of power rails and clocks and configuring hardware registers in the appropriate sequence. The idea of the boot loader is to run some tests to ensure the hardware is OK and then to load the Linux Kernel on to the CPU. Eventually the Linux Kernel will load Android (or GNU).

Part II

Hardware

6 HARDWARE ARCHITECTURE

6.1 STATIC VS. DYNAMIC POWER

A hardware architect will almost always face a tradeoff between area of a module and its clock frequency. For instance, an FIR filter with 10 taps with a max throughput of 48 KHz can either be implemented with 10 single-cycle multipliers or with a single single-cycle multiplier. The former choice would mean that the filter would have to be clocked at 48 KHz and the latter would imply a clocking of 10*48 KHz. This area vs. clock tradeoff is essentially a static vs. dynamic power consumption tradeoff. Though there are no fixed formula on how to decide one over the other, the thumbrule is to conserve area and increase clock frequency up to 500 MHz.

Static power consumption is because of leakage; Dynamic because of transistor toggling

6.2 CHOOSING TO GO THE HARDWARE WAY

Any functionality can be implemented in software or hardware. Each has its own advantages. Software is very flexible in that something broken can easily be fixed, whereas hardware is pretty much written in stone. However, software uses the CPU's generic functionalities, and hence, will most likely consume higher power compared to a hardware tailored for one particular functionality. However, when the use case requiring the functionality in question is over, CPU's MHz can be utilized for some other functionality, while the hardware will be sitting idle leaking power - So if the functionality in question isn't alive most of the time, implementing it in hardware may be a bad idea. Hence, hardware accelerators are usually the way to go in ASICs, such an ADSL system where the same kind of functionality needs to be repeated for every received symbol. And they are usually *not* the way to go in general purpose computing devices such as application processors, where it is hard to predict when a certain functionality (as part of a use case) will be alive and when it won't be alive.

SW is flexible; HW is efficient

Thumbrule is to go HW for ASICs, go SW for general purpose ICs

These above described scenario, of course, *usually* the case in general purpose ICs, which means there are unusual cases where hardware may be the way to go even when the functionality in question occurs rarely. An example would be a system where we don't know what are all the software that can run on the CPU . In this case, if we implement all the known functionalities in hardware, then CPU MHz will be freed up to run the unknown functionalities. Another example would

All thumbrules have caveats!

be a CPU that would run at SV voltage corner at, say 500 MHz, and would be pushed to the HV corner if a 50 MHz functionality is added to it - If the power rail the CPU is connected to has all of its other components asking mostly for SV, then the power rail would now have to run at HV just because of the CPU. This may cause higher leakage in all other components - let us call this unnecessary extra leakage in other components as $\Delta P_a * T_f$, where ΔP_a is the extra leakage power and T_f is the time for which the functionality in question will be alive. Keeping this in one hand, suppose we use a hardware module to implement that 50 MHz functionality in question and if the unnecessary leakage power of the module when the functionality is not alive is $\Delta P_b * (T_b - T_f)$, where T_b is some sort of a total time for all use cases. If $\Delta P_b * (T_b - T_f) < \Delta P_a * T_f$, then it would make sense to implement that 50 MHz functioanlity in hardware.

HW can operate sample-by-sample; SW only on chunks

Besides the power angle, there is also a latency angle that needs to be considered - software will always work on chunks of data at a time, while hardware can work on a sample-by-sample basis. This is because the OS allocates time slices to different software codes in chunks and within that chunk, it is economical to process data in chunks as well. This automatically means higher running latency. So hardware usually has lower running latency than software. This, of course, doesn't apply to hardware modules that cannot operate on data on a sample-by-sample basis. An example is an FFT accelerator.

7 MISCELLANEOUS

7.1 MEMORY

There are two types of memory namely volatile and non-volatile memory. As a thumbrule, non-volatile memory is slower (in reading, writing) than volatile memory. Within volatile memory, there are two sub-types namely Static RAM (SRAM) and Dynamic Ram (DRAM). DRAM uses a trasistor and a capacitor to store a bit, while an SRAM uses a flip-flop containing 4 to 6 transistors to do the same job. Thus a DRAM has a higher memory density (more bits per sq. mm) than SRAM. The downside of DRAM is that it is slower to read, write than SRAM and also needs to be constantly "refreshed" because the capacitor slowly leaks its charge. Usually on-chip RAMs, small in size, such as scratch pad RAMs and Caches are SRAMs, while the "main memory", relatively much larger in size, is a DRAM and is off-chip. The SoC, however, provide a "Memory Controller" that can periodically refresh (and do many other things) the off-chip DRAM.

SRAM for speed; DRAM for memory density

The term "Tightly Coupled Memory" is used for scratch RAMs that CPUs have dedicated access to - i.e., RAMs that aren't shared with other modules. Thus RAMs sitting on a bus (say AXI) as a slave serving many bus masters (say, a CPU, a DMA engine etc.) cannot be termed as a TCM. By design, CPU access to TCMs is faster than access to other internal RAMs as the CPU will have to deal with bus protocol overheads and has to compete with other modules while trying to access a non-TCM RAM. The downside of this is that TCMs may be under-utilized (for ex., when CPU is in WFI), and hence it should be economically sized to have the smallest memory capacity acceptable.

TCMs are SRAMs dedicated to CPU

Cache, like TCM, is a memory that is dedicated to the CPU. It is used to store copies of frequency accessed data that is present in some iRAM (on or off-chip). When a CPU starts afresh (say, after a reset), and tries to read a memory location, instead of reading the value in the location, a DMA is started to copy an entire block of memory in the vicinity of the generated address from the RAM to the Cache. The idea is that the CPU is running a code that is likely to access nearby address locations in that block sometime in the near future. Any further CPU access to addresses inside the block will imply reading data from the Cache instead of going all the way to the RAM. An importnat point to note is that, when the first DMA happens, the DMA first fetches the address CPU asked, no matter whether

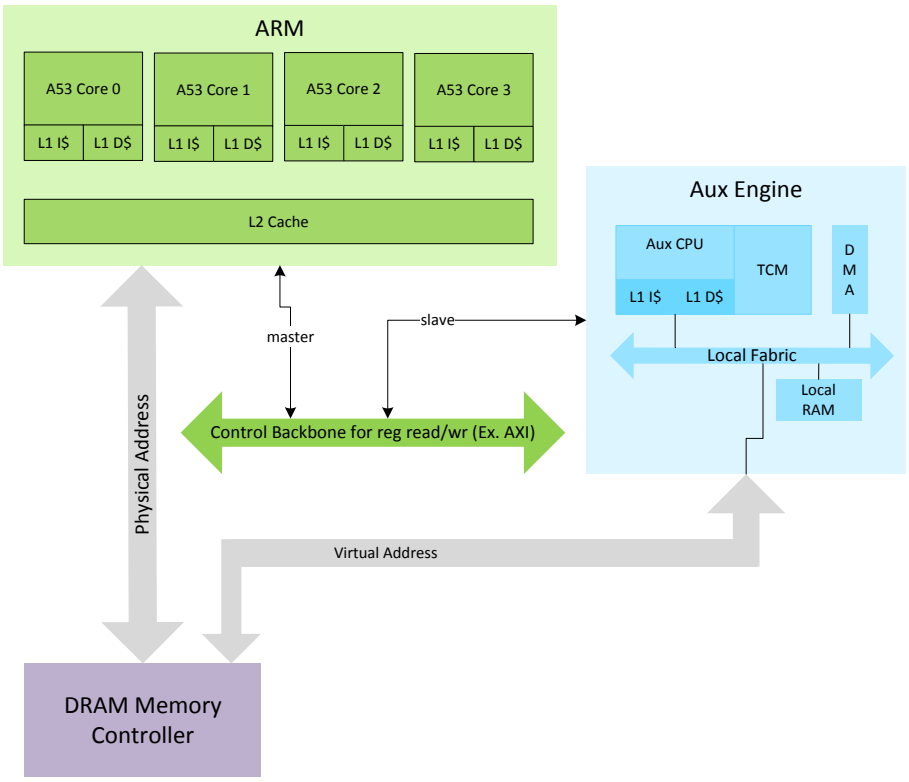


Figure 7.1: Different Memory Types

the address is at the beginning of the block to be fetched, or at the end of it or somewhere in the middle. This means that CPU will quickly get the data it wanted and will start working on it, while, in parallel, the DMA will continue fetching the remainder of the block.

Just like the OS loads a program from a flash memory into RAM before executing it, - because flash access is slower than RAM - one can think of Cache as a kind of a TCM into which contents of the RAM are loaded into so that access to the contents could be faster. However the similarities end there. While OS deliberately loads a code from flash into RAM, is managed completely by hardware and done dynamically (per memory access) rather than just once before a context switch. For instance, the first time CPU accesses a location, it results in what is called a *cache miss*, which means that the data CPU is trying to access is not in cache, but need to be copied into cache from RAM. After the first DMA is over, when the next addresses CPU accesses are found to be in the cache, it is called a *cache hit*. Since cache is limited, a future cache miss may mean that current contents of the cache may have to be flushed in order to make way for new contents. All of these tasks

are part of the cache hardware and are performed dynamically. More details about how cache works, various cache architectures can be found in the Cache & MMU section in Microprocessor Architecture

Memory Size :	Dynamic RAM > Internal RAM > L2 Cache > L1 Cache
Speed :	Dynamic RAM < Internal RAM < L2 Cache < L1 Cache

7.2 HARD RESET AND SOFT RESET

Hard reset is used to reset a modules registers back to their "Power-on Reset (POR)" values and reset internal state machine to the initial state. Soft reset does only the latter part of resetting the state machine while the register values (had they been modified by the software) are not reset back to their default values. Generally the hardware architect defines what the POR values should be for every register field. The POR value is then converted into necessary pull-ups and pull-downs[1] sitting behind a mux that either forces these values (on hard reset) or allows user writes through. The pull-ups and pull-downs do occupy space and one may be tempted not to define the POR value (leave it at X): After all, the POR values are prescribed so as to reduce SW configuration burden, and if the HW architect chooses, s/he could make it mandatory for the software to always configure all the registers of a module before using that module, and hence save area by eliminating pull-ups and pull-downs. However this is a bad idea as the possibility of having a bit as X, at any time, implies X propagation and verification engineers now have to identify all the conditions when this X could happen and all the paths in the module this X could be propagated to and manually waive them in the testing tool. This process is cumbersome. Besides that, not having POR values is just a bad idea - it is akin to having variables in a software that come up with garbage values: if the code accidentally utilizes such a variable without assigning a value, imagine the bugs that could spawn.

Hard reset affects reg values. Soft reset doesn't

A note on defining POR values: POR values must be chosen with the big picture of the scenarios in which the module in question will be used. One cannot frivolously make the POR values to be all zeros. The idea is that, if we know the most likely configuration of the registers, then the POR values should match that configuration - so that, when SW wants to use the module after power-on or a hard reset, they can simply configure one or two registers (possibly just write to the enable register) and get on to using the module.

It is idiotic to make all POR values zeros

Hard reset is asynchronous in nature. So a module need not have its clock running when a hard reset is applied. But de-asserting a hard reset requires clock. So generally one turns on the power and clock of a module before hard-resetting it. Soft reset is completely synchronous. So pretty much every operation on a module requires the clock to be on first.

[1] "Pullups" and "pulldowns" are resistors connected to Vcc and Gnd respectively

8

TRANSISTOR AMPLIFIER

8.1 TRANSISTOR CHARACTERISTICS

In a Bi-Junction Transistor (BJT), Collector is physically the largest region, followed by the Emitter. Base is the smallest region. In terms of doping, Emitter is the most heavily doped of the three, followed by the Collector and the Base is the least doped. The following picture captures this info.

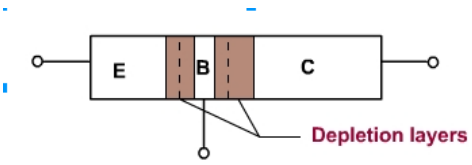
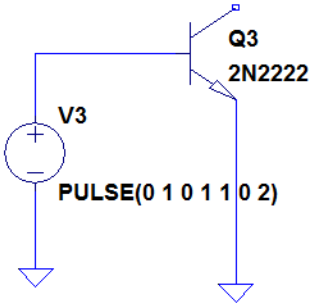


Figure 8.1: A BJT

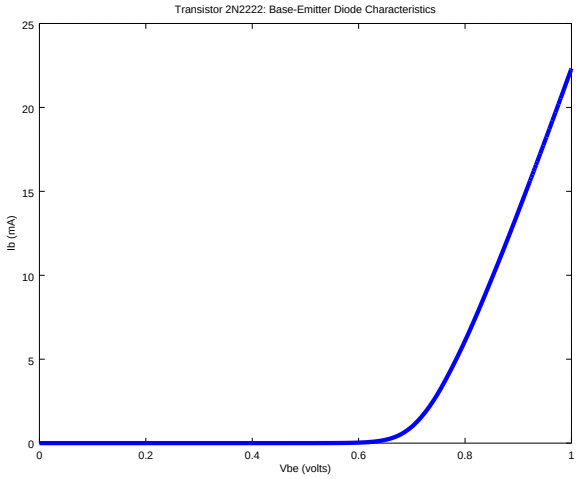
Let us look at how BJT acts as an electrically controlled switch in the CE configuration. We can use an abstract analogy for it: Imagine we are trying to turn right into Aakruti homes. But the gate is closed. We are waiting for the gate to be opened. There are lot of cars waiting behind us that need to continue going down the street - although the street is empty, they cannot go because we are blocking them. The moment the gate is opened and we make the turn, cars behind us can continue down the street. We can think of the Aakruti homes gate as the terminal connecting Base to the battery - as soon as it is positively biased, it is like opening the gate. This will allow electrons to smoothly from the Collector to the Emitter.

In BJT, say, in an npn one, the way this analogy works is that, as soon as BE is positively biased, electrons in the depletion layer get dislodged and go out of the Base terminal thereby allowing electrons to flow from Emitter to the Collector [1]. The reason why not all electrons will simply exit out of the Base terminal is that the Base is very thin and the path out of the Base is like a narrow gate - not many cars can turn into it - and since the wide road is now unblocked, most go from the Emitter to the Collector.

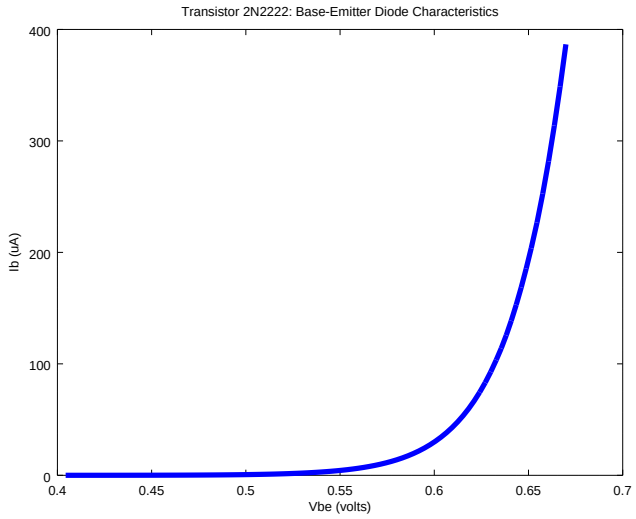
Let us look at some SPICE simulations and try to understand the transistor characteristics better. First let us understand the Base-Emitter diode's voltage and current characteristics by using the circuit in Figure 8.2a. The Collector is intentionally left to float as we are only interested in the other two terminals.



(a) BE Diode Circuit



(b) V_{BE} Vs. I_B



(c) V_{BE} Vs. I_B Zoomed in

Figure 8.2: The Base-Emitter Diode

When we run the simulations and check the relationship between voltage and current at the Base-Emitter terminals, we get a graph as shown in Figure 8.2b. A zoomed in version is shown in Figure 8.2c

It is clear that the diode opens at around a V_{BE} of 0.65V or so and from there on, it seems that V_{BE} and I_B are linearly related to each other.

We made a small mistake in the circuit depicted in Figure 8.2b - We gave a fixed voltage to the BE terminal. However, as we know, as the voltage is increased, the resistance across BE decreases. This is not captured in the circuit and the associated graphs. We can introduce a Base resistor so that, when the resistance across BE changes, V_{BE} changes. Figure 8.3a shows this circuit and Figure 8.3b, Figure 8.3c depict how V_{BE} and I_B change with V_{in}

Now, with the new figures, we can clearly see that V_{BE} doesn't keep on increasing with V_{in}. After reaching about 0.66V (approx), V_{BE} saturates. We can also see that I_B is practically zero until V_{BE} is around 0.6V and then increases almost linearly w.r.to V_{in} after that.

Now that we understand the BE diode well, we can move on to exploring the Collector side of things. We can do this by connecting the Collector to a voltage source as shown in Figure 8.4a. A series resistor at the Collector is added for the same reason we added a series resistor at the Base.

Figure 8.4b and Figure 8.4c shows how V_{CE} and I_C change w.r.to I_B and V_{BE} respectively. There is a striking revelation in these figures. V_{CE} and I_C vary linearly with I_B, but non-linearly with V_{BE}. Besides that, we remember from Figure 8.3b that, while V_{BE} eventually settled to a value of about 0.82V, I_B kept changing linearly with V_{in}. Both these characteristics mean that studying Collector behaviour against I_B is better than studying them against V_{BE}. They also encourage us to think of a CE amplifier as a current amplifier rather than a voltage amplifier.

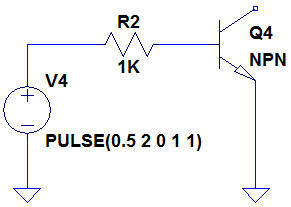
Another stiking thing is that the open loop current gain, i.e., $A_{OL} = \frac{I_C}{I_B}$, while averaged over the period when the transistor is active, is very close to 200. In fact, if we exclude the small range at the end when I_C is seen flattening out, it is exactly 200. This is specific to 2N2222, our example transistor, which makes it a very attractive choice as an A_{OL} of 200 will make many mathematical equations simple.

All the information we need to build a CE amplifier is available to us in the form of Figure 8.4b. However, the information in that figure is usually depicted in a different way in most of the literature - they show I_C against V_{CE} for a certain I_B value. Typically many curves, each one representing one I_B value are superimposed as shown in Figure 8.5. That figure was obtained by using a constant current source to the Base and a varying V_{CC}.

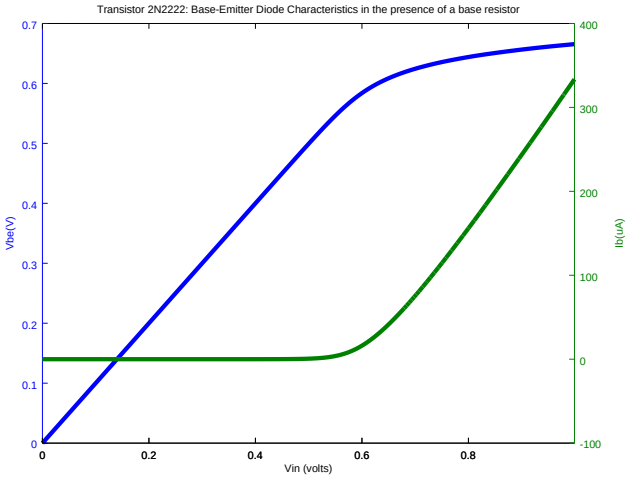
8.1.1 Effect of V_{CC} and I_C on Current Gain

We found out earlier that the open loop gain, A_{OL} for 2N2222 transistor is 200. But that was in a specific context - we were using a circuit that had a $1K\Omega$ resistor at the Base, a $1K\Omega$ resistor at the Collector and a V_{CC} of 5V. But what happens when this context changes? Will the current gain remain the same? Let us find out.

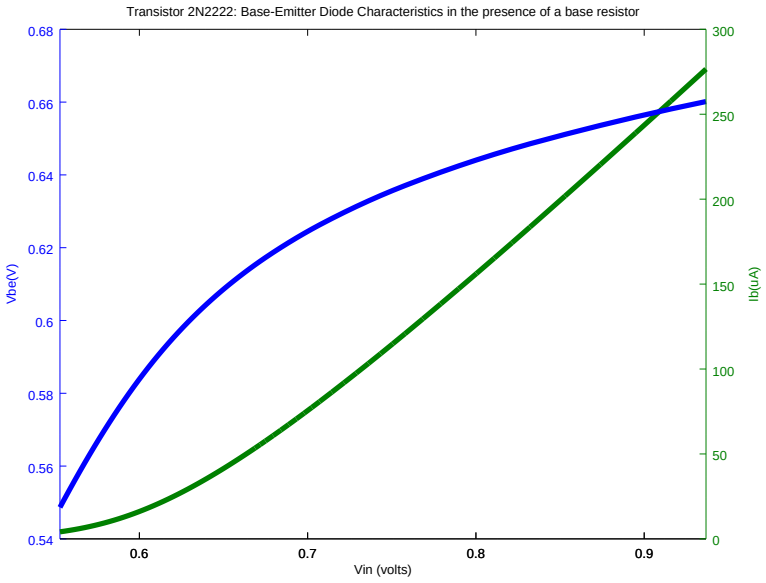
Figure 8.7 shows three similar circuits with changing V_{CC} value while every-



(a) BE Diode Circuit



(b) V_{BE} Vs. I_B

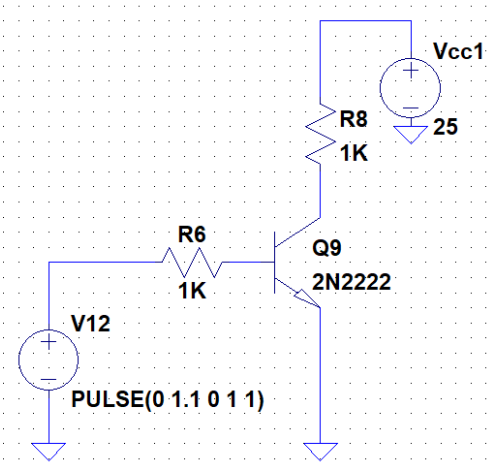


(c) V_{BE} Vs. I_B Zoomed in

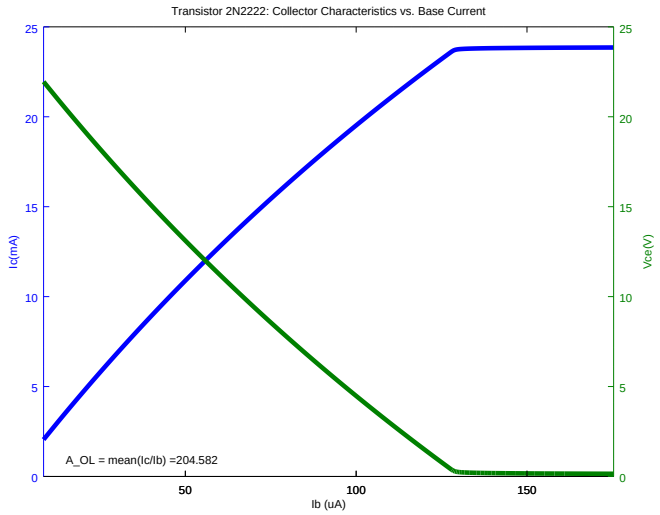
Figure 8.3: The Base-Emitter Diode with a Base Resistor

thing else is the same. It also shows what happens to I_C when V_{CC} is changed. Just to be sure, values of I_B are also plotted. There are several interesting observations from the results:

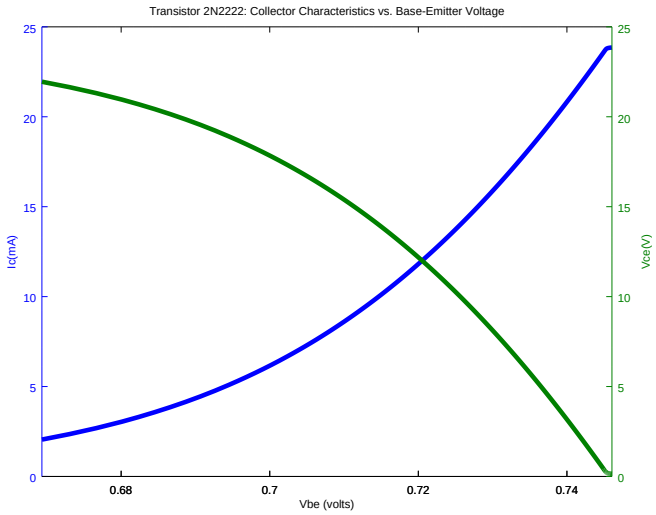
- Changing V_{CC} had an effect on I_B. The effect was small: A 60% and 140%



(a) Base-Emitter Vs Collector



(b) V_{CE}, I_C as functions of I_B



(c) V_{CE}, I_C as functions of V_BE

Figure 8.4: Effect of Base Parameters on Collector Parameters

change in V_{CC} respectively changes I_B by merely -0.04% and -0.1% respectively.

- The effect of V_{CC} on I_C was also small. A 60% and 140% change in V_{CC} changed I_C by 2.8% and 6.8% only.
- Consequently, A_{OL} changes by 3% and 7% respectively for V_{CC} changes of 60% and 140% respectively. In other words, changing V_{CC} had an insignificant effect on the current gain.

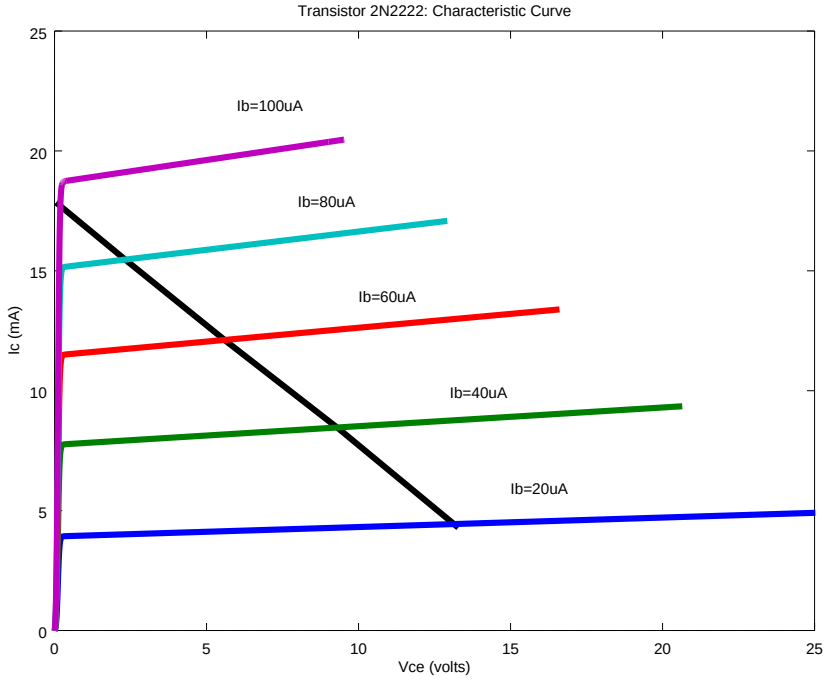


Figure 8.5: Transistor Characteristics: NPN 2N2222

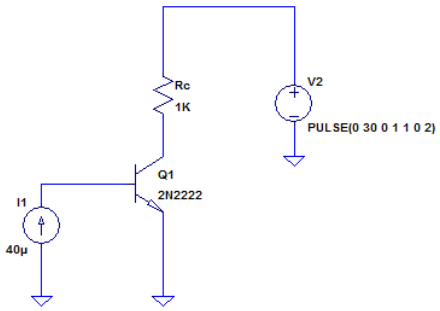


Figure 8.6: CE Amplifier with Constant Current Source and a Varying V_{CC}

- The implication of the previous point is that, when V_{CC} changes, V_{CE} is the only quantity affected.

Figure 8.7 shows three similar circuits with changing R_C value while everything else is the same. It also shows what happens to I_C when R_C is changed. Just to be sure, values of I_B are also plotted. There are several interesting observations from the results:

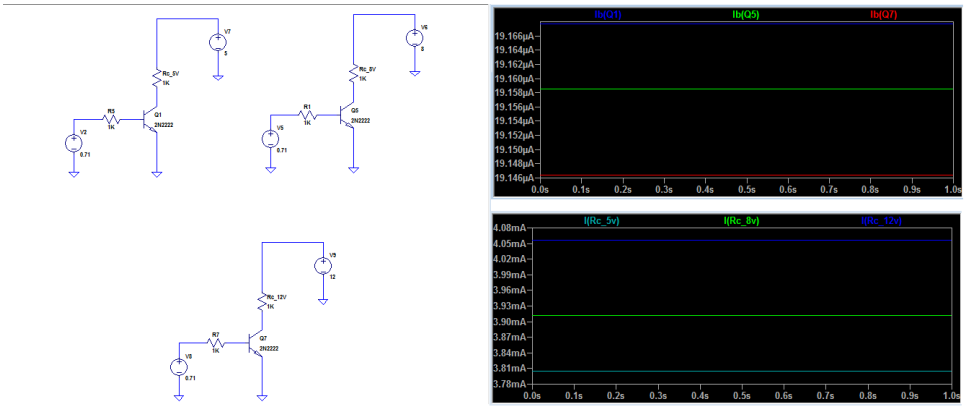


Figure 8.7: Effect of V_{CC} value on Current Gain

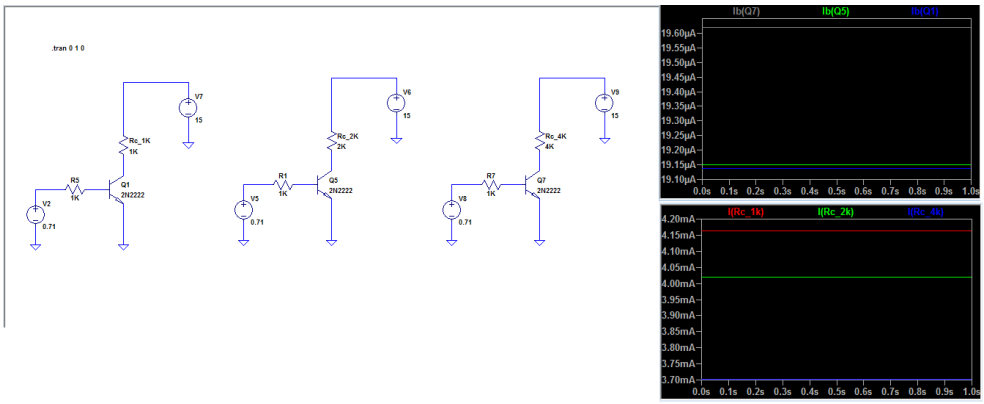


Figure 8.8: Effect on R_C value on Current Gain

- Changing R_C has negligible effect on I_B. A 100%, 400% change in R_C values caused a 0.005% and 2.5% change in I_B respectively
- Same was the case with I_C. A 100% and 400% change in R_C changed I_C by -3.3% and -11% only.
- Consequently the effect of changing R_C had an insignificant effect on the A_{OL}. A 100%, 400% change in R_C values caused a -3.2% and -13.3% change in A_{OL} values respectively.
- An interesting implication is that, it is important to choose an appropriate V_{CC} for this experiment. For instance, choosing a V_{CC} of 8V instead of 15V would have misled us with the results: Remembering from the previous experiment that changing V_{CC} has practically no effet on A_{OL}, and interpreting the result of emphthis experiment, the value of I_C would have remained at somewhere close to 4mA, resulting in a voltage drop of 8V and 16V across R_C when

its values are $2K\Omega$ and $4K\Omega$ respectively. Since the supply itself is 8V, this would have meant that V_{CE} would have been 0V and the drop across R_C in both cases would have saturated to 8V. In other words, we would have driven the transistor into its saturation region.

8.2 AMPLIFIER OPERATING POINT AND DC LOAD LINE

When we design an amplifier, it is prudent for us to not expect the input signal to have a DC value of about 0.7V (in the case of a 2N2222 transistor). Firstly it is normal for the person using the amplifier to assume that it will amplify a small signal with a DC value of 0V into a large signal, again centered around 0V DC. Secondly, different transistors will have different operating voltages - It is unfair to impose the requirement of setting the correct DC value for V_{in} on the user. Hence it is advisable to have internal arrangement that will keep the transistor at the so called quescient point or DC operating point that will allow the maximum range of input values a transistor can handle without sacrificing linearity [1].

To start with, we need a way to supply some Base current even when there is no input. The only other power source for the amplifier is V_{CC} and so we obviously have to derive our I_B from it. Besides that, we also need to have a mechanism by which the input signal's voltage gets added to the quiescent voltage at the Base. This can be achieved by using an input capacitor. Finally, the output of the amplifier needs to have a signal with a DC value of 0. This can also be achieved by using a capacitor. So the circuit in Figure 8.9 represents the starting point for setting up our CE amplifier's quiescent point.

From this circuit, one can draw two equations as shown below:

$$V_{CC} - I_B * R_B - V_{BE} = 0 \tag{8.1}$$
$$V_{CC} - I_C * R_C - V_{CE} = 0 \tag{8.2}$$

With these two equations, we can solve two unknowns - often, they are R_B and R_C. In other words, we will have to come up with some values for the other variables in the equations and then solve the equations to find out R_B and R_C. Among the variables we need to fix, V_{CC} is often a system-level constraint. For example, it may be battery operated system using two AA batteries, in which case,

[1] It is important to understand that, although biasing a transistor is useful as explained in this paragraph, it is detrimental in circuits where power consumption is the biggest issue. After all, when the transistor is biased. it will have some Base current, and more importantly, some Collector current flowing even when there is no input signal, amounting to wastage of power. Imagine a battery powered device which remains in stand-by state (no input) for a long time, such as a walkie-talkie. Biasing is obviously a bad idea in those cases.One should instead spec out the input signal requirement and meticulously ensure that the circuitry to the left of the transistor is engineered to produce the required type of input signal. For the sake of continuity of the discussions, let us save further discussions on these cases to a later time

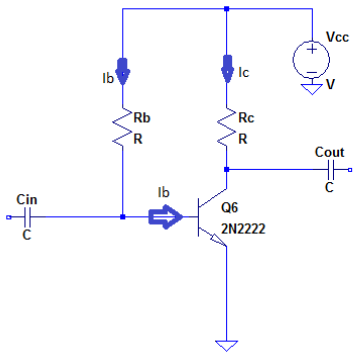


Figure 8.9: CE Amplifier with Constant Current Source and a Varying V_{CC}

the V_{CC} will be <6V. Once V_{CC} is available, next step is to decide how much the max value of I_C should be. Power consumption constraints, load's ability to handle current if R_C itself is the load, the max current capacity of the power source, circuit heat requirements etc. will determine I_{Cmax}. We can then draw the DC load line in Figure 8.5. Note that, since $I_C = I_{Cmax}$ when $V_{CE} = 0$, using Equation 8.2, one can directly find R_C value. Or, one can also determine quiescent point V_{CE} and I_C values first and then substitute them into Equation 8.2 to find R_C. Both methods will give the same R_C value. Deciding what quiescent V_{CE} should be will depend on standby power requirements and/or expected input signal range (and hence the expected signal output range). If range is the concern, then it makes sense to choose quiescent V_{CE} to be $\frac{V_{CC}}{2}$. Since the transistor always operates along the DC load line, once quiescent V_{CE} is chosen, quiescent I_C and I_B assume consequential values. Besides these, from Figure 8.3b, V_{BE} can be found out for a given I_B value. Thus, we will have all the information available with us to find R_B using Equation 8.2.

8.2.1 Example

Let us assume that V_{CC} is 6V and I_{Cmax} is 6mA. That would meant that R_C should be $1K\Omega$. Now the DC load line equation can be obtained by observing that the end points are (V_{CC}, 0) and (0, I_{Cmax}):

$$\frac{y-0}{I_{Cmax}} = \frac{x-V_{CC}}{-V_{CC}} \tag{8.3}$$

$$Rearranging., \tag{8.4}$$

$$y = \frac{-I_{Cmax}}{V_{CC}} * x + I_{Cmax} \tag{8.5}$$

$$= \frac{-1}{R_C} * x + I_{Cmax} \tag{8.6}$$

$$\tag{8.7}$$

So, if we choose quiescent V_{CE} to be $\frac{V_{CC}}{2}$, then using equation Equation 8.5, we find out that quiescent I_C will be 3mA. Since 2N2222 has a current gain, A_{OL} of 200(approx), that would mean that $I_B = 15\mu A$. For this I_B, V_{BE}, from **??** is approximately 0.685V. Substituting these in equation Equation 8.1,

$$R_B = \frac{V_{CC} - V_{BE}}{I_B} \tag{8.8}$$

$$= \frac{6-0.685}{15\mu} \tag{8.9}$$

$$= \frac{5.315}{15\mu} \tag{8.10}$$

$$= 354.33K\Omega \tag{8.11}$$

We can substitute the above values in a SPICE model and verify using simulations that we are indeed achieving the quiescent point values for I_C and V_{CE} as desired. But in the real world, all resistor values aren't available[2]. So we will have to choose a popular resistor whose value comes the closest to the R_B value of $354.33K\Omega$.This happens to be $330K\Omega$. Using this R_B value, we can either reverse engineer V_{BE}, I_B combination using **??** - In a way, we used the above calculations to get an idea of the R_B value we are looking for, and then work our way backwards to find out what I_C is going to be. We can then, if desired, tweak the R_C value to come up with something that will get us closer to the quiescent $V_{CE} = \frac{V_{CC}}{2}$ point. We can use maths for all this or simply run simulaitons with $330K\Omega$ of R_B and play with R_C values. I used simulations to find out that I_C will be 3.25mA. So if we desire a VCE of $\frac{V_{CC}}{2}$, then our R_C should be $\frac{3V}{3.25mA} = 923\Omega$. So if we can find a resistor value closer to 923Ω than $1K\Omega$ is, then we can use that resistor instead of a $1K\Omega$ resistor for R_C.

A note on the input dynamic range: For a moment, let us assume that we don't get a resistor value closer to 923Ω than $1K\Omega$ is. At $R_C = 1K\Omega$, our quiescent V_{CE}

[2]Unless we are talking about a custom designed LSI circuit

is already at 2.747V. We can only go down to 0V[3]. This means that, on the lower side, we have a room of 2.747V instead of a desired swing of 3V with which we set out our design. Since our input signal is very likely to have an equal swing on the positive and negative side, this implies that the V_{CE} value cannot swing more than 2.747V, not only on the negative side but also on the positive side as well. So the max value of V_{CE} will be $2.747V + 2.747V = 5.494V$. To achieve this V_{CE}, we need a drop of $6V - 5.494V = 0.506V$, in other words, an I_C of $0.506mA$. Since our A_{OL} is approximately 200, this would mean an I_B of $2.53\mu A$, or, referring to table:VbeVsIb, a V_{BE} of 0.637V. Our quiescent V_{BE} is 0.686V (found from simulation). This leaves us a room of $0.686V - 0.637V = 0.049V$. In other words our input signal's maximum allowed swing is 100mV (approx).

It is always important to check in simulation if our theoretical calculations are right. This is because we have made some simplifications related to linearlity between V_{BE}, I_B and more that are not a 100% true in reality. Furthermore, we also have deviated from theoretical values (ex. R_B value). So Figure 8.10 shows the simulation results of our actual circuit.

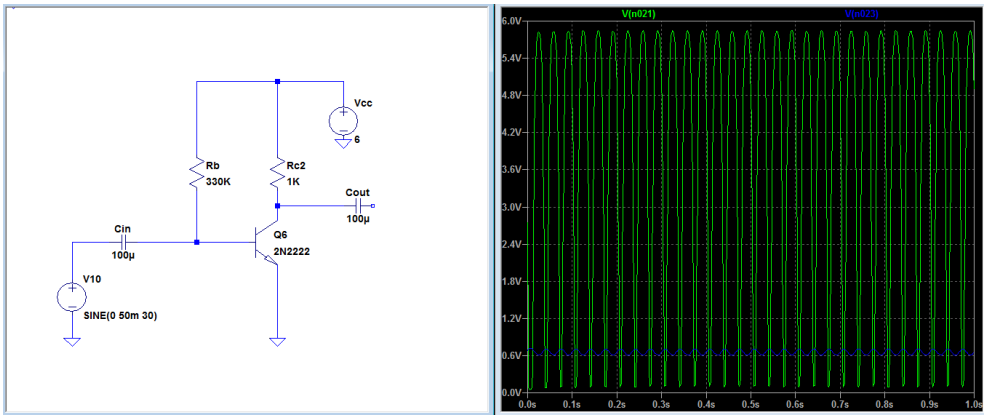


Figure 8.10: CE Amplifier with DC biasing derviced from $V_C C$

We can clearly see that the V_{CE} signal is covering the entire voltage range possible - so our prediction that the input signal swing can't be more than a 100mV was right.

[3]Had our calculations gave us an I_C such that quiescent V_{CE} ended up being higher (instead of lower) than $V_{CC}/2$, then the upper limit of 6V (i.e., V_{CC} would have been more restrictive on the V_{CE} swing than the ground level of 0V

BIBLIOGRAPHY

[1] Sangho Kim, *Transistors, How to they Work?*, Learn Engineering, Massachusetts, 2nd edition, 2016.

Part III

Software

9 OPERATING SYSTEM BASICS

Any OS is comprised of a kernel and some software services built on top of the kernel that give higher level functionalities to the user. For ex., In Linux based operating systems, Linux forms the Kernel and, on top of that, whole lot of GNU services are available, such as the bash shell. The kernel interacts with hardware, loads and schedules processes, does context switching etc. Theoretically the user can directly call the functions provided by the kernel. But it would be cumbersome and not straightforward. Software services like GNU tools provide more intuitive functions to the user and inside these functions call necessary kernel functions in necessary order to accomplish the user desired task.

OS is kernel plus some software services built on top of the kernel

There is no limit to know many such layers of abstraction can be built - An OS may have very high level software services, that then use lower level software services which eventually call kernel functions. In fact, modern OSs like Android, easily have 5 layers of software services built on top of the kernel. This is precisely what makes the term “OS” vague. While all the services running in the privileged mode in the CPU can be labeled as part of the Kernel, there is no such a clear definition for “OS” as it could have however many layers of software running in the non-privileged mode. Some people call the kernel as the operating system and call, what other people traditionally call, OSs as “packages” or other terms.

10 GNU-LINUX

The GNU project was envisioned to provide a lot of free software services for "Unix like" kernels - basically kernels that have function names (along with intended functionalities) the same as Unix function names. This meant that no matter which kernel is used, the user can install and run GNU tools on top of that kernel as long as it is a "Unix like" kernel. Unix OS (kernel + software services) was expensive at that time and GNU founders wanted benefits of the computer revolution available to everyone, not just those who have the money to buy expensive OSs like Unix. GNU project initially focused on creating software services hoping that one day someone will write a free Unix-like kernel. When Linus Trovalds wrote the Linux kernel[1] and made it free, it perfectly fit into GNU's equations. The GNU-Linux combo meant that now users had an entire OS available to them for free [2]. Though today there are other free Unix-like kernels available which can be used with GNU tools, Linux is still the most preferred kernel.

GNU can be used with any Unix-like kernel, not just Linux

10.1 LINUX DISTRIBUTIONS

While it is up to a user to choose what GNU tools to use on top of the Linux kernel, the choices of GNU tools are too many. Few users meticulously want to choose GNU tools to use for their display, audio, video, internet and other needs. Hence the idea of linux distributions (actually GNU-Linux distributions) or "distros" was born - a group of enthusiasts meticulously choose GNU tools to satisfy each one of user's conceivable requirements, test if the tools are interacting with the Linux kernel or with each other without issues and publish packages that contain all the (tested sw versions) of chosen GNU tools and the (tested version of) Linux kernel. Users can simply download these packages and can jump start using a computer right after installing the package, which are called "distros". Some examples of distros are Ubuntu, Red Hat and CentOS.

"Distros" are packages containing select set of GNU tools and Linux kernel

Over the years, the idea of packages has undergone major evolutions. At some point, corporations, like Red Hat, made it a commercial venture to provide packages. The idea is that, the free packages put together by enthusiasts offer no

[1] written completely in C

[2] Free software and open source software are independent concepts. GNU is free as well as open source. But there are commercial open source tools as well. Open source is meant to allow users to examine the quality of the code they obtained free of cost or for a fee

guarantee in terms of performance, reliability etc., and big firms hence cannot use GNU-Linux as compared to, say, Windows OS, wherein Microsoft delivers the OS with some guarantees, and in case of failures, accepts some liabilities and provides continuous support. So companies like Red Hat make packages with rigorously tested GNU-Linux components and give certain guarantees so that GNU-Linux becomes an acceptable alternative to other OSs like Windows and Unix to big firms. Along the same lines, some companies started making commercial packages that had a similar, but slightly different rationale - their customer firms need packages that are focused on specific functionalities, like networking, because their products focus only on such services - ex., web servers. Such firms do not want the clutter of GNU tools unrelated to networking. Of course, like all other firms, additionally they want guarantees and liabilities accompanying the packages. An example of such a package is the SUSE Linux.

There are commercial and free distros

Adding to the free/not-free distro flavours mentioned in the above paragraphs, the companies providing commercial distros realized that workers of the customer firms may be unfamiliar with their packages. So they made free versions of their non-free packages with the idea of grooming common people who, in the future, will form the work force of their customer firms. So Red Hat provides Fedora, SUSE provides "open SUSE" etc., which are all free. While this is the scene with commercial firms, the enthusiasts have gone berserk - there are now many major distros (ex. Debian, Arch, Alpine etc.), minor variations of these distros (Ubuntu, Tiny etc.) to subtler variations such as (Kubuntu, Mint etc.) and so on and on. There are/have been hundreds of distros. But many either have a very small following are dead in the water. As of when this was written, Ubuntu is, by a wide margin, the most preferred free package - but this may change in the future.

There are hundreds of distros. Ubuntu is most popular.

10.2 MISCELLANEOUS

10.2.1 Kernal Prints

The components of a Linux kernel, namely system calls, device drivers etc., output many "kernel prints" - these are basically *printk* statements, which are the kernel equivalent of *printf* statements. These prints go into the kernel log ring buffer. This buffer is then processed by syslog daemon for user visibility. One can use the command *dmesg* to directly parse the log ring buffer and display all the kernel prints. These prints are very useful while debugging if anything is not working in Linux.

11 LINUX DEVICE DRIVERS

Drivers abstract out hw registers, expose meaningful functions

Linux device drivers, as with device drivers of any OS, are abstraction layers built over hardware devices' register configuration interface. Device drivers are part of the kernel and hence run in a privileged mode called the "kernel mode". All the software outside the kernel (part of the OS or applications) run in a less privileged mode called the "user mode". User mode programs call device drivers functions, which then configure the registers of the hardware device in question.

Driver runs in "kernel mode", while SW above the kernel run in a less privileged "user mode"

Sometimes, some device drivers themselves call other device drivers - suppose we have a mouse connected to the computer with a USB cable. Here the "peripheral driver", that is meant to interface with the mouse, will have to use the usb controller driver in order to be able to write into the mouse device's hardware registers. In fact, we can extrapolate it to one more step - if our USB controller was sitting at the periphery of a motherboard, perhaps a PCIe bus is used to throw signals all the way from the CPU to the USB controller. In this case, the USB controller driver itself has to invoke the PCIe driver in order to be able to read/write into the controller's hardware registers. All in all, there is no way to say how many layers of drivers are there in the control chain from the user mode program to the hardware registers of a device - but the crux is that any device driver needs to carry out register writes to its device - that may involve a direct CPU write using an internal bus (AXI/APB etc.) or the write may have to go through complex paths involving other drivers.

Linux drivers can be loaded/unloaded without reboot

Linux device drivers have a special quality that made linux a more attractive choice than windows OS for web server applications - unlike in Windows, device drivers in linux can be loaded and unloaded on the flly without requiring a reboot. Since restarting a server would mean taking down the connections of possibly a few hundred clients, linux automatically became attractive.

Linux device drivers have three types of "verticals" namely,

- Character drivers - for mouse, webcam, keyboard etc.
- Block drivers - for storage devices
- Packet drivers - for networking

And each of these verticals have their associated “horizontals”. For instance, the packet driver vertical has WiFi driver, ethernet driver etc. as its “horizontals”. Block drivers maintain the file protocol stack (FAT32, EXT4 etc.,) and have device specific horizontals like magnetic disk drivers, SD card drivers etc. Character drivers are used for pretty much every device that hasn’t anything to do with networking or storage. One can imagine its horizontals. In fact, the horizontals are too many that character drivers are usually subdivided into tty drivers, sound drivers, console drivers and more.

11.1 FRAMEWORK

Device driver framework defines how user mode programs actually go about using the services provided by the drivers. In Linux, the user mode programs perform file operations on “device files” and a kernel daemon called “Virtual File System” or VFS converts these file read/write operations into function calls to driver functions[1]. Drivers are written like normal dynamically loaded C programs. But instead of being built as a .so shared object[2], they are built as .ko objects, with ‘k’ standing for ‘Kernel space’. This .ko file is referred to as “module”. Once the module is available, the following commands could be used with it:

```
lsmod   /* Lists currently loaded modules */
insmod moduleName.ko /* Loads a module */
modprobe moduleName.ko /* Loads a module and its
   dependencies */
rmmod moduleName.ko /* Unloads a module */
```

Inside the driver code, there are special functions that run when the above module operations are performed on them - *insmod* invokes *module_init()* and *rmmod* invokes *module_exit()*. *module_int()* sets the “major number”and “minor number”[3] of the driver. Alternatively the initialization function can request the kernel to assign these numbers to the driver[4]. Once a driver is loaded, we need to create a corresponding device file to interact with it. The command used for this is:

```
    mknod deviceFileName typeValue majorNo minorNo

    /* Example */
    mknod mydevfile c 60 0
```

[1] While other methods of interacting with drivers exist, for pedagogical reasons we will focus on this method

[2] ‘so’ files is the linux equivalent of windows dll files

[3] minor number is like a version no. of the driver

[4] For some reason, this is considered to be safer

Here the typeValue tells the kernel whether the driver is a Character or Block or Packet driver. Once this is done, user mode programs can use this device file to interact with the driver just loaded.

11.2 MISCELLANEOUS

11.2.1 A Bare Minimum Driver

The following code shows a driver that simply outputs some kernel prints when loaded and unloaded.

```
/* ofd.c - Our First Driver */
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
static int __init ofd_init(void) /* Constructor */
{
ăăăprintk(KERN_INFO "Namaskar: ofd registered");
ăăăreturn 0;
}
static void __exit ofd_exit(void) /* Destructor */
{
ăăăprintk(KERN_INFO "Alvida: ofd unregistered");
}
module_init(ofd_init);
module_exit(ofd_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anil Kumar Pugalia <email_at_sarika-
    pugs_dot_com>");
MODULE_DESCRIPTION("Our First Driver");
```

Listing 11.1: A Bare Minimum Driver

The above code was part of a Linux Drivers Tutorial. It guides the reader step-by-step towards writing simple, and, later on, complex drivers.

11.2.2 Kernel Library Functions

Since the driver runs on kernel space, it cannot use user space libraries [5]. This means that, unless the Linux kernel source code is available, one cannot compile a driver code. In Ubuntu, one can use *apt-get install linux-source* to get the source code[6].

[5] available in /usr/src/include
[6] The source code goes to /usr/source/linux

Part IV

Networking

12 BASICS

Networking is all about enabling different devices to talk to one other. Here the term “device” may refer to computers or any other electronic entity. Thus the Internet connects many devices with one other and, in this case, the devices are computers. But when a computer is connected to many different USB devices via a USB hub, that is also networking. The network could be inside an IC where a CPU and a DMA Engine may be connected to a RAM, ROM and many hardware modules. This is also networking - In this case, the CPU, DMA, Memory Controller (RAM/ROM) are all “devices”. [1]

Scenarios where devices talk to one another can be compared to scenarios where people talk to each other. For instance, take the scenario where a teacher is talking to a class. Most of the time, she is talking and all the students in the class are listening. This is a broadcasting scenario. But when a student has a question, he raises his hand - this conveys to the teacher that someone wants to talk and hence the teacher stops talking to let the student talk. Similary scenario may exists in a network, in which case, solutions applied for the human communications problem can be replicated for the devices: Notice that in the human communication problem, we avoid simultaneous talking of people, by using a visual way of communicating that the student wants to talk. Here one can think of two separate “channels” existing: One is the audio channel in which any one person can send information at a given time and a visual channel which is used to figure out who gets to occupy the audio channl. In terms of devices, we could similarly solve using two channels: The visual channel could simply be an interrupt line, where as the audio channel will be the actual data line(s).

12.1 CSMA/CD

Imagine a bunch of individuals in a room. Say 10 people in the room want to have one-to-one conversation with ten others. To avoid chaos we allow only one

[1] Though, traditionally the word “networking” is used to refer to computer networking like the Internet, this distinction is unnecessary - It is like calling linear algebra as “signal processing”. When we do that, we unnecessarily confuse people into thinking it is a separate field and prevent them from applying techniques learned from Signal Processing on to solving non signal processing problems. In fact, concepts in Networking can be applied to not just electronic entities, but also to mechanical entities, like conveyor belts, road and train networks etc.

conversation to happen at a give time. There are many ways to solve this problem. One is to use some sort of a round robin method: Every person in the room has a no. and there is a digital screen that basically changes number every 20 seconds. Only the person with the no. currently displayed on the screen is allowed to talk. This is a simple solution, but has a fundamental problem: Even if the person whose no. is currently displayed has nothing to say, still the next person in line has to wait for 20 seconds. This is such a waste of time. In such real world situations, normally what people do, is that they see if anyone is talking and when no one is, they start speaking. But if, just as they start talking, if someone else also starts talking, they will stop and let the other person talk. It is possible that, when two people start talking simultaneously, both may back off out of courtesy. And again one of them will try to start talking. Both may again start talking simultaneously....this can happen a bunch of times. And eventually one just stays silent for a longer time hinting that the other person may now talk.

What was described in the above paragraph is called *Carrier Sense Multiple Access with Collision Detection* or *CSMA/CD*. The very first wireline networks (and wireless networks even today) followed this. In this case, simply, a bunch of devices will all be connected to the same bus (a bunch of logically related wires) as shown below. When a device in this simple network wants to send data to another device, it will,

1. First observe, for an amount of time, if there is any activity happening on the bus
    - If there is some activity going on, wait until the bus falls silent
    - If or when the bus is silent, it can start sending data
2. Send data for a small quantum of time and simultaneously read back the data from the bus. Match the sent data with the read-back data
    - If they don't match, then it means someone else also tried to talk during that past quanta and there was a data clash. Stop sending data and go back to step 1
    - In they do match, continue sending data

To avoid any one device hogging the bus for a long time, we can set a rule that no one can keep talking beyond a stipulated period of time.

12.2 NETWORK SWITCH

In modern wireline networks, single bus based networking is rarely found. Instead some sort of a switch is employed as shown below. The switch typically contains on dedicated bus between each pair of devices that would ever want to talk to each other - for now, let us assume that every device would like to, at some point,

talk to every other device on the network. Now, if a device, say device A, wants to talk to device B, it will simply raise a flag and the switch, if no one else is talking to device B, will "enable" the connection between A and B and the communication will start. If two devices want to talk to device B at the same time, the switch will allow the higher priority device to talk first. If both devices have the same priority, the switch will randomly allow one device (coin toss). Since the switch has a dedicated path for each device pair, the arbitration mentioned above is required only when two devices want to talk simultaneously to the same third device. But if the two devices want to talk to two other devices, no arbitration is required. So while a A->B, C->B communication cannot happen simultaneously, a A->B, C->D can.

12.3 MASTER AND SLAVES

The idea of masters and slaves in a network is that, while master may initiate conversations, slaves will either only listen or reply when asked by a master - *i.e.*, A slave will never initiate a conversation. In computer networks, one's laptop would be a master while a web server could be a slave. In ICs, typically CPUs, DMA engines etc., will be masters, while Memory Controllers, I/O Controllers will be slaves.

While a slave will never initiate a conversation, it may want to talk to a master. In this case, just like the teacher in a class room example, side channels are used to inform the master of the slaves "wish". The side channel may be elaborate or could be a single wire interrupt. And just as the teacher may either immediately pause and ask the student to go ahead, or wait for a while and then let the student talk (or completely ignore him), the master device may immediately enquire the slave or may wait for a while and then enquire or completely ignore it.Thus while the slave, may in an indirect way, want to talk to the master, granting that wish is up to the master.

13 OSI MODEL

The most popular model used for networking computers, for long, has been the OSI model. It envisions networking using 7 layers of abstraction as shown in the diagram below (Courtesy: Wikipedia, "OSI Layers").

Layer	Data Unit	Function	Examples
7. Application	Data	High-level APIs, including resource sharing, remote file access, directory services and virtual terminals	Mail, Internet Explorer, Firefox, Google Chrome
6. Presentation	Data	Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption	ASCII, EBCDIC, JPEG
5. Session	Data	Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes	RPC, PAP, HTTP, FTP, SMTP, Secure Shell
4. Transport	Segments	Reliable transmission of data segments between points on a network, including segmentation, acknowledgment and multiplexing	TCP, UDP
3. Network	Packet/Datagram	Structuring and managing a multi-node network, including addressing, routing and traffic control	IPv4, IPv6, ARP, IPsec, AppleTalk, ICMP
2. Data Link	Bit/Frame	Reliable transmission of data frames between two nodes connected by a physical layer	PPP, IEEE 802.2, L2TP
1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium	DSL, Ethernet, USB

Table 13.1: OSI Layers

The Physical layer is all about converting bits to signals that can be transmitted over a long distance and recovered back to bits. Hence it is a topic of digital communications that we will not deal with here. The top 3 layers are too high level and have such a wide scope that we cannot cover them here. So we will

focus on the remaining 3 layers: Transport, Network and Data Link layers. We will occasionally say a thing or two about the rest.

Perhaps the best way of introducing networking is to talk about how data transaction and routing happens at every OSI layer. Since networking devices come at different levels of complexity, with each dealing with data transmission and routing up to certain layer in the OSI and being agnostic about layers above that. In fact, if we try to understand how various networking devices work, starting from the simplest and building up to more complex ones, we could get to grips with the basics of networking.

A *Hub* is a "Layer 1 device" that understands only physical layer. Consequently it is the simplest of all networking devices. A Hub is simply a broadcaster/repeater. It has several ports. However, since physical layer is all about sending bits from one end to another, there is no concept of routing data. Hence a Hub broadcasts the bits it receives from any of its port to all other ports. Connecting many nodes to a Hub is the simplest approach to networking - we will simply be enabling a node to talk to more than one other node, with nodes ignoring data the Hub broadcasts to them, if it wasn't meant for them (This happens at the Data Link Layer level). Without a Hub, a network can have a maximum of only two nodes! - one node directly connected to another.

Hub are outdated. Originally they were cheaper compared to more complex network devices and could "get the job done", they are obviously pretty wasteful since they broadcast all the time.

A *Switch* is a "Layer 2 device" that understands up to Data Link Layer level. A Swich routes data from one of its ports to another using a Data Link Layer address called *MAC address*. When several nodes are connected to a Switch for the first time, the Switch doesn't know the MAC address any of the nodes. Consequently, for some time, it behaves like a Hub - When a Switch receives a Data Link Layer frame on one of its ports, say, port 1, for the first time, it looks into the frame header and notes down the source MAC address found in it against the port 1. But, since it has no idea which of its ports is connected to the node indicated by the destination MAC address in the frame, it simply broadcasts the frame to all other ports. The nodes that receive this frame, examine if the frame belongs to them by matching the destination MAC address in the frame to their own MAC address. Only the actual destination node then responds to the source with a frame while other nodes simply ignore it. So on the way back, when the Switch grabs this return frame, and note down the sender's MAC address against the port it received it in - say, port 2. Thus it slowly gains knowledge of the which of its ports is connected to which MAC address. So after a while, there won't be any unnecessary broadcasting.

A *Router* is a "Layer 3 device" that understands up to the Networking layer. We will see how a router works in a while. But first, let us understand more about the Networking layer. As it's name implies, this is the layer primarily responsible

for networking. This layer uses an addressing scheme called the IP address. Even though it may appear like MAC address is used to connect several nodes and nodes communicated with each other using MAC addresses, it is not true. When a node, say, node A, wants to talk to another node, say, node B, it doesn't right away start sending frames with node B's MAC address. Node A initially knows only node B's IP address [1]. It needs to find out the MAC address of node B in order to start communicating with it - This is because MAC address is the lowest level of address in a network and without it data cannot be moved around in a network[2]. To find out node B's MAC address, node A uses a network layer protocol called *Address Resolution Protocol* or *ARP*. Basically, node A sends puts an ARP message on the network which tells a switch to intentionally broadcast to all the nodes connected to it. All the nodes receive the ARP message and the node that sees the destination IP address match with its own will reply, while others ignore. In the reply, node B will stamp its MAC address in the message and send it to node A's. Node A will thus build a table over time with IP addresses of nodes against their MAC addresses. Note that when node B receives the ARP from node A, it would have made a note of Node A's MAC address before sending the reply.

13.1 CONNECTING TWO LOCAL NETWORKS

In the above paragraphs, notice how we didn't need a router for two nodes to talk to each other? Well, we don't need a router as long as the nodes in a network want to talk only to other nodes in the same network. So, what do we mean by "same network" here? How do we say that two nodes are on the same network and not on different networks? In fact, what does a "different network" mean? To answer these questions, imagine two networks shown below. Each consist of a bunch of nodes connected to each other using a hub.

We know from the paragraphs preceding this section how nodes in each network will talk to another node in its network - we will call this its "local network" from here on. We know how each node will build a table of IP addresses against MAC addresses of other nodes in its local network and how the hub builds a table of MAC addresses against its ports. But what if a node in the network N1 wants to talk to a node in network N2? For that we need routers. Routers are essentially what define a local network. The following figure depicts this.

The way it works is that, the user of a node, say 192.168.0.100, in network N1 would enter the network properties in the node as shown below.

When this node wants to talk to another node in N1, say 192.168.0.102, the node applies the *Subnet Mask* on this IP address (logical AND) and sees if the result matches 192.168.0.0. In our case, it does, and this means this IP address

[1] Either we manually enter node B's IP address into node A's application or DNS may be used. We will talk about this in the next section

[2] For instance, if there is a switch in the network, it cannot understand IP addresses, but only MAC addresses

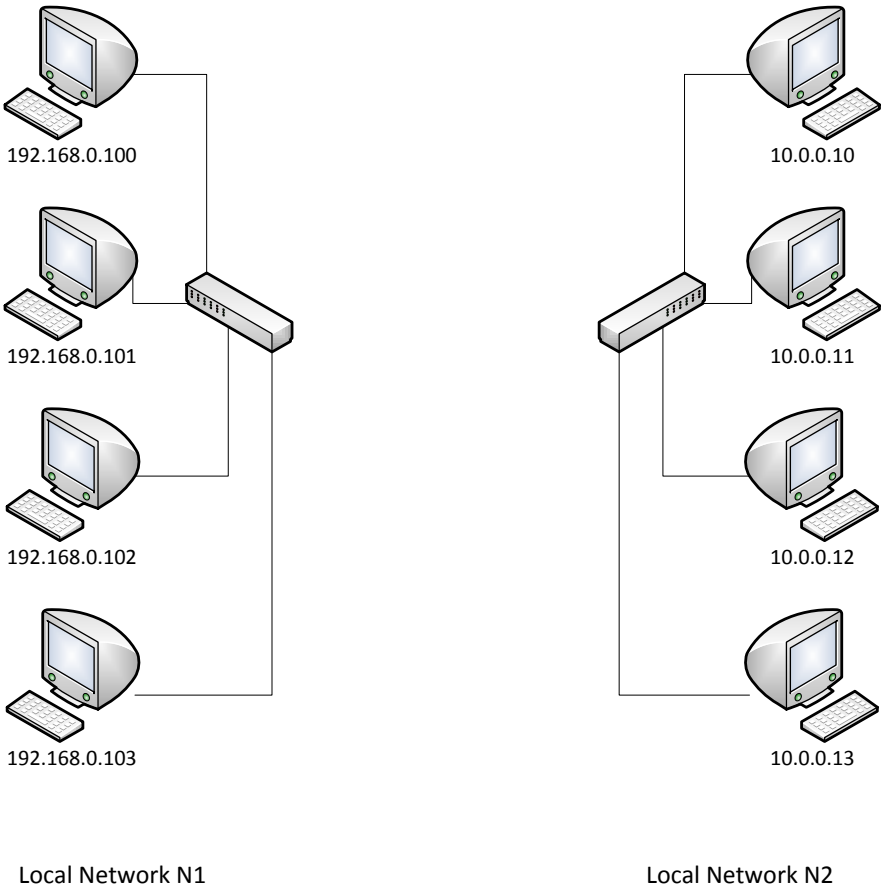


Figure 13.1: Two Local Networks

is in the same network, i.e., N1. At this point, the node 192.168.0.100 proceeds to find out the MAC address of the destination node, 192.168.0.102 (if it already doesn't know), using ARP, so that it can start talking to it. And we know how the rest will work.

Now consider the case when the source node, 192.168.0.100 wants to talk to 10.0.0.10. Again, it applies the subnet mask, but the result, 10.0.0.0 doesn't match its N1's 192.168.0.0. The source node thus understands that the destination node is not in its own network, i.e., N1. In this case, the source node doesn't try to find the destinations MAC address. Instead it simply stamps the message with the router's MAC address and sends the message to the router. Note that the message will still have the destination IP address of 10.0.0.10 - only the MAC address will be that of the router's. How does the source node know the router's IP address? It is because the user enters it in the "Default Gateway" field of the

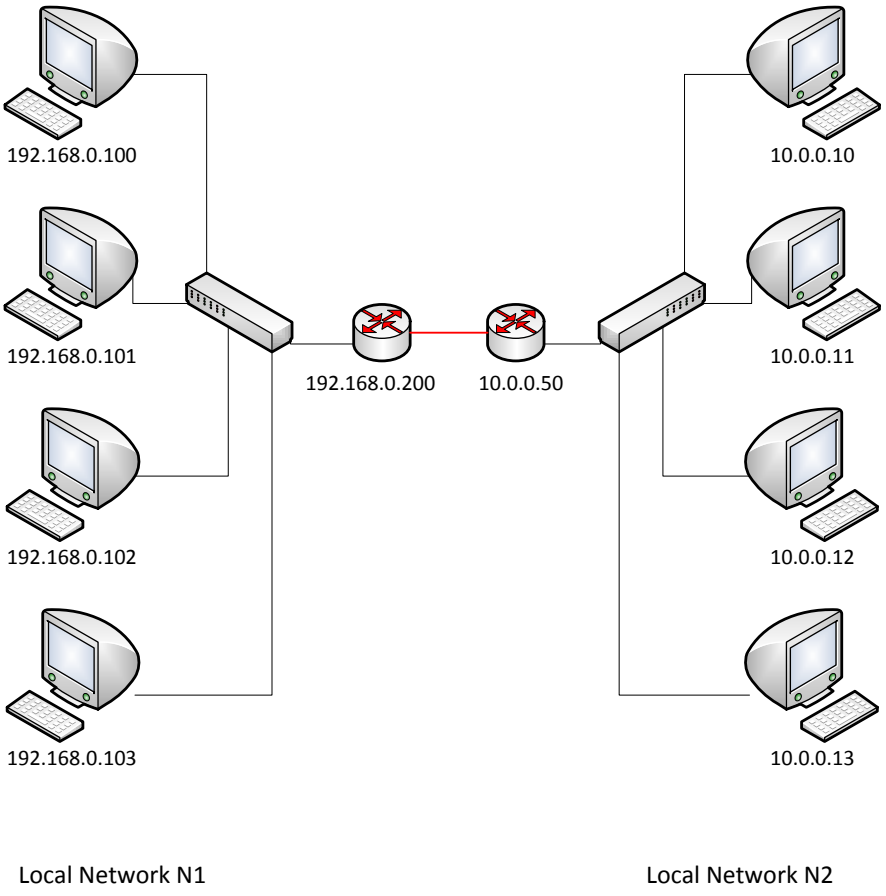


Figure 13.2: Inter Network

network properties dialog box[3]. The router looks at the message it received from the source node, examines the destination address and realizes that it doesn't match with its own IP address - it means the message was actually meant for someone outside network N1. So the message is sent to the router on N2[4]. This is accomplished by the router on N1 changing the destination MAC address of the message (from its own MAC address) to match the MAC address of the router in N2. Again, only the destination MAC address is changed, but the IP address is left intact as 10.0.0.10. Router on N2 receives the message, and just like the router

[3]Note that, if the source node doesn't know the router's MAC address, it will again have to first use ARP to find that out

[4]In our simple case, router on N1 is connected only to the router on N2 and no other router. So whenever it receives a message meant for a node outside its network, it has no other choice but to send to N2. We will talk later about what happens in the case of the router on N1 connected to many other routers - how it will know the destination node is in N2

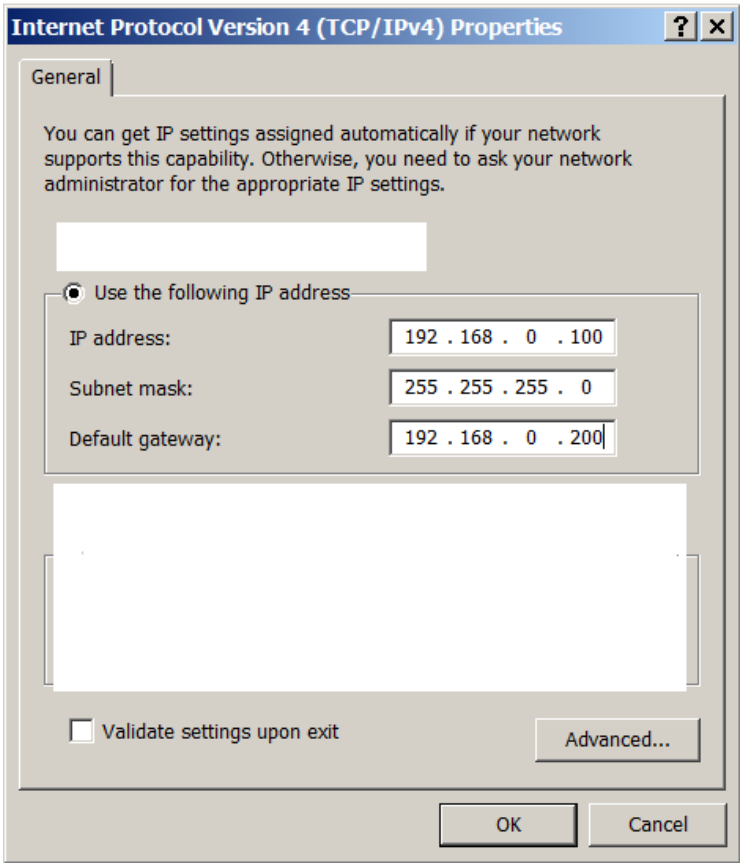


Figure 13.3: Network Properties

on N1, realizes that the message isn't meant for it. However, it understands that the destination IP address is in its own network and forwards the message to the switch in N2 which then sends the message to the destination node.

13.1.1 The Internet

The internet is nothing but a lot of local networks connected together using routers. We just saw how nodes from two local networks can talk to each other with the help of routers. In this case, the routers, which were the "gateways" of these local networks were directly connected to each other. This is an oversimplistic possibility, one which will be impractical to use while connected so many local networks together to form the internet - obviously every local networks gateway cannot be connected to all other gateways. It is actually not required that the gateways of two local networks be directly connected to each other to enable communication between them. In fact, the essence of routers comes clear only when the local networks are *not* connected directly to each other. The figure below shows how two local networks will actually be connected on the internet.

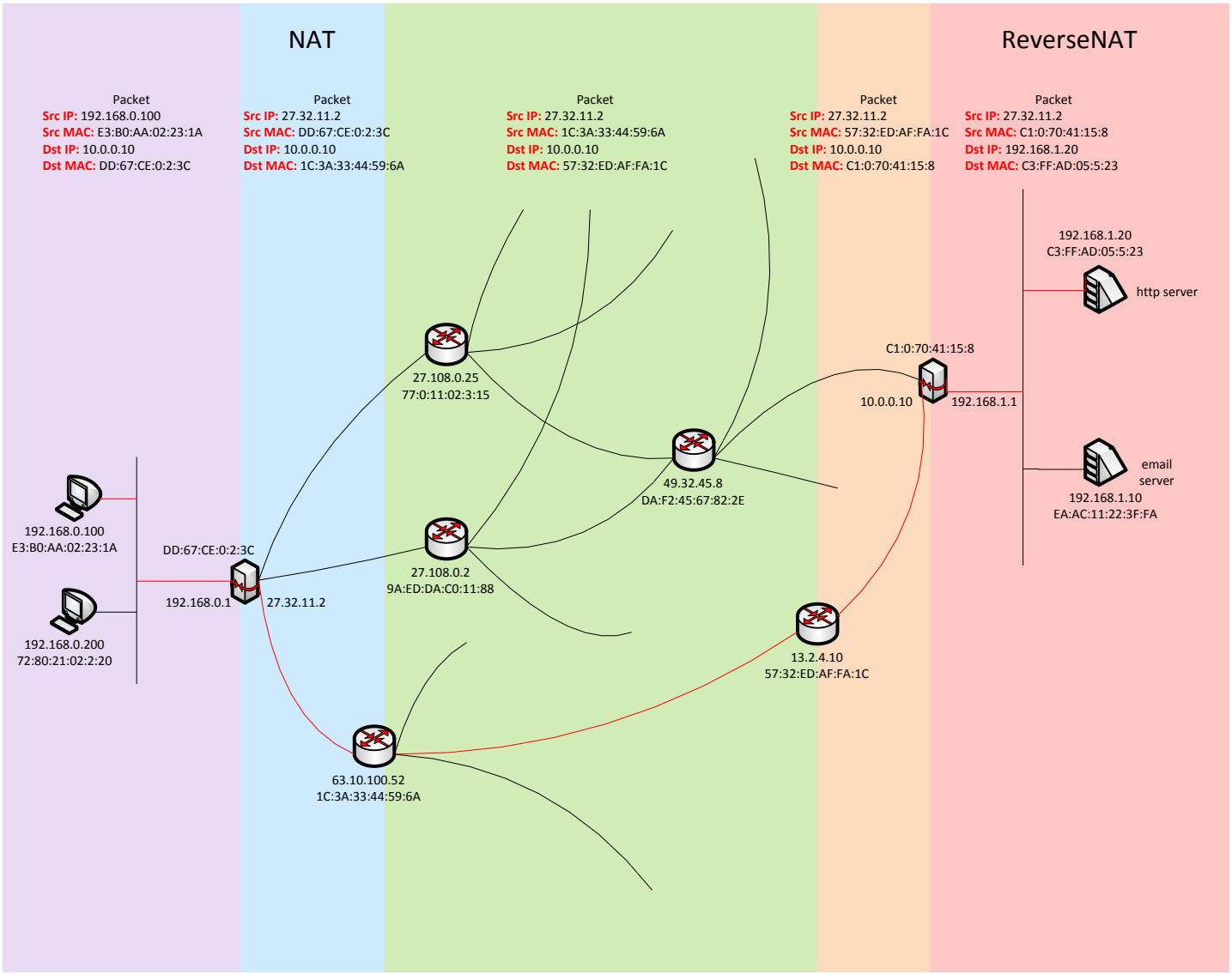


Figure 13.4: The Internet

Compare this with the simplistic version where gateways were directly connected to each other. The difference is that, in the simplistic version when the gateway of N1 realized that the packet sent by 192.168.0.100 wasn't meant for it, as the destination IP address, 10.0.0.10 doesn't match its own IP address, and forwards the packet directly N2. But in the real world, the gateway now has to send it to one of the routers R1, R2, R3. But which one of these will it choose?

The answer is a little complicated, and we will deal with it later, but for now, let us just say that every router keeps a table that indicates which one of the routers it is connected to lies on the shortest path to the destination IP address. This is called the *routing table*. The data in the routing table a best estimate and the router doesn't know whether it is a fact. Actually the routing table data keeps changing from time to time - This is where the whole idea of routing and that of the internet itself becomes interesting: This means that, the gateway of N1, which is also a router, may not choose the same router, that it chose to send the first packet to 10.0.0.10, for the following packets as well. In our case, we show that it chose R3 because its routing table said that R3 was on the shortest path to 10.0.0.10. But, suppose, after sending this packet, its routing table changed,[5] now it may think R1 is on the shortest path to 10.0.0.10 and send the next packet to R1. The same logic applies to how the first level routers R1, R2, R3 will then forward the packet. In other words, every *IP Packet* can potentially follow completely different routes from the source to the destination. This idea is called *packet switching* and this is the genius of the internet.

Network Address Translation

While discussing interconnecting two local networks, it was mentioned that the gateway of N1 changes the source MAC address of the data sent out to its own but leaves the source IP address intact. This is true only if *Network Address Translation*(NAT) is not turned on in the gateway. In the case of internet and most networks, NAT is almost always enabled for all gateways. NAT is all about translating all the IP addresses in a local network to a

[5] We will see later how routing table is built and how it evolves