There are three architectural parameters that characterize a virtual memory system and hence the architecture of the MMU.

P is the number of address bits used for the page offset in both virtual and physical addresses.

V is the number of address bits used for the virtual page number.

And M is the number of address bits used for the physical page number.

All the other parameters, listed on the right, are derived from these three parameters.

As mentioned earlier, the typical page size is between 4KB and 16KB, the sweet spot in the tradeoff between the downside of using physical memory to hold unwanted locations and the upside of reading as much as possible from secondary storage so as to amortize the high cost of accessing the initial word over as many words as possible.

The size of the virtual address is determined by the ISA.

We're now making the transition from 32-bit architectures, which support a 4 gigabyte virtual address space, to 64-bit architectures, which support a 16 exabyte virtual address space.

"Exa" is the SI prefix for 10^18 - a 64-bit address can access a *lot* of memory!

The limitations of a small virtual address have been the main cause for the extinction of many ISAs.

Of course, each generation of engineers thinks that the transition they make will be the final one!

I can remember when we all thought that 32 bits was an unimaginably large address.

Back then we're buying memory by the megabyte and only in our fantasies did we think one could have a system with several thousand megabytes.

Today's CPU architects are feeling pretty smug about 64 bits - we'll see how they feel in a couple of decades!

The size of physical addresses is currently between 30 bits (for embedded processors with modest memory needs) and 40+ bits (for servers that handle large data sets).

Since CPU implementations are expected to change every couple of years, the choice of physical memory size can be adjusted to match current technologies.

Since programmers use virtual addresses, they're insulated from this implementation choice.

The MMU ensures that existing software will continue to function correctly with different sizes of physical memory.

The programmer may notice differences in performance, but not in basic functionality.

For example, suppose our system supported a 32-bit virtual address, a 30-bit physical address and a 4KB page size.

So p = 12, v = 32-12 = 20, and m = 30 - 12 = 18.

There are 2^m physical pages, which is 2^18 in our example.

There are 2^v virtual pages, which is 2^20 in our example.

And since there is one entry in the page map for each virtual page, there are 2^20 (approximately one million) page map entries.

Each page map entry contains a PPN, an R bit and a D bit, for a total of m+2 bits, which is 20 bits in our example.

So there are approximately 20 million bits in the page map.

If we were thinking of using a large special-purpose static RAM to hold the page map, this would get pretty expensive!

But why use a special-purpose memory for the page map?

Why not use a portion of main memory, which we have a lot of and have already bought and paid for?

We could use a register, called the page map pointer, to hold the address of the page map array in main memory.

In other words, the page map would occupy some number of dedicated physical pages.

Using the desired virtual page number as an index, the hardware could perform the usual array access calculation to fetch the needed page map entry from main memory.

The downside of this proposed implementation is that it now takes two accesses to physical memory to perform one virtual access: the first to retrieve the page table entry needed for the virtual-to-physical address translation, and the second to actually access the requested location.

Once again, caches to the rescue.

Most systems incorporate a special-purpose cache, called a translation look-aside buffer (TLB), that maps virtual page numbers to physical page numbers.

The TLB is usually small and quite fast.

It's usually fully-associative to ensure the best possible hit ratio by avoiding collisions.

If the PPN is found by using the TLB, the access to main memory for the page table entry can be avoided, and we're back to a single physical access for each virtual access.

The hit ratio of a TLB is quite high, usually better than 99%.

This isn't too surprising since locality and the notion of a working set suggest that only a small number of pages are in active use over short periods of time.

As we'll see in a few slides, there are interesting variations to this simple TLB page-map-in-main-memory architecture.

But the basic strategy will remain the same.

Putting it all together: the virtual address generated by the CPU is first processed by the TLB to see if the appropriate translation from VPN to PPN has been cached.

If so, the main memory access can proceed directly.

If the desired mapping is not in the TLB, the appropriate entry in the page map is accessed in main memory.

If the page is resident, the PPN field of the page map entry is used to complete the address translation.

And, of course, the translation is cached in the TLB so that subsequent accesses to this page can avoid the access to the page map.

If the desired page is not resident, the MMU triggers a page fault exception and the page fault handler code will deal with the problem.

Here's a final example showing all the pieces in action.

In this example, p = 10, v = 22, and m = 14.

How many pages can reside in physical memory at one time?

There are 2^m physical pages, so 2^14.

How many entries are there in the page table?

There's one entry for each virtual page and there are 2^v virtual pages, so there are 2^22 entries in the page table.

How many bits per entry in the page table?

Assume each entry holds the PPN, the resident bit, and the dirty bit.

Since the PPN is m bits, there are m+2 bits in each entry, so 16 bits.

How many pages does the page table occupy?

There are 2^v page table entries, each occupying (m+2)/8 bytes, so the total size of the page table in this example is 2^23 bytes.

Each page holds 2^p = 2^10 bytes, so the page table occupies 2^23/2^10 = 2^13 pages.

What fraction of virtual memory can be resident at any given time?

There are 2^v virtual pages, of which 2^m can be resident.

So the fraction of resident pages is 2^m/2^v = 2^14/2^22 = 1/2^8.

What is the physical address for virtual address 0x1804?

Which MMU components are involved in the translation?

First we have to decompose the virtual address into VPN and offset.

The offset is the low-order 10 bits, so is 0x004 in this example.

The VPN is the remaining address bits, so the VPN is 0x6.

Looking first in the TLB, we that the VPN-to-PPN mapping for VPN 0x6 is cached, so we can construct the physical address by concatenating the PPN (0x2) with the 10-bit offset (0x4) to get a physical address of 0x804.

You're right!

It's a bit of pain to do all the bit manipulations when p is not a multiple of 4.

How about virtual address 0x1080?

For this address the VPN is 0x4 and the offset is 0x80.

The translation for VPN 0x4 is not cached in the TLB, so we have to check the page map, which tells us that the page is resident in physical page 5.

Concatenating the PPN and offset, we get 0x1480 as the physical address.

Finally, how about virtual address 0x0FC?

Here the VPN is 0 and the offset 0xFC.

The mapping for VPN 0 is not found in the TLB and checking the page map reveals that VPN 0 is not resident in main memory, so a page fault exception is triggered.

There are a few things to note about the example TLB and page map contents.

Note that a TLB entry can be invalid (it's R bit is 0).

This can happen when a virtual page is replaced, so when we change the R bit to 0 in the page map, we have to do the same in the TLB.

And should we be concerned that PPN 0x5 appears twice in the page table?

Note that the entry for VPN 0x3 doesn't matter since it's R bit is 0.

Typically when marking a page not resident, we don't bother to clear out the other fields in the entry since they won't be used when R=0.

So there's only one *valid* mapping to PPN 5.