At the end of the last lecture, I discussed briefly why caches are needed. Data and instructions can take up alot of physical memory and so in practice the majority of data and instructions must be kept in larger, cheaper, slower memory. We uses caches in order to keep in a very fast (small, expensive) memory those data and instructions that are frequently used. Caches can be accessed within a single clock cycle.

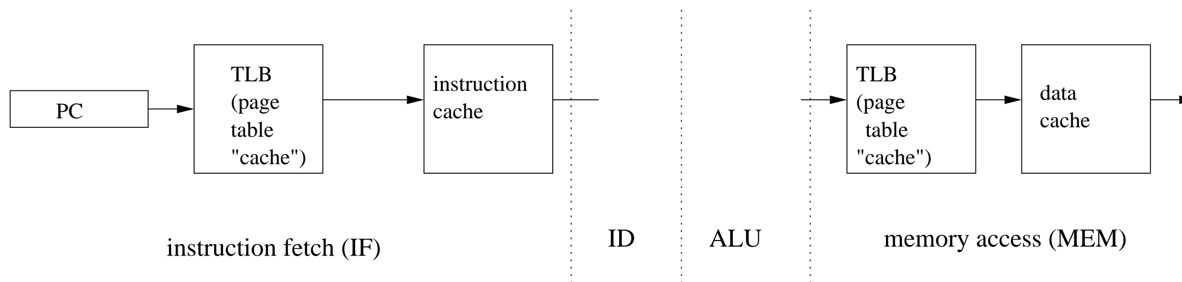
Translation lookaside buffer (TLB)

We begin today by considering a specialized cache for the page table cache which is called the *translation lookaside buffer* (or TLB). This cache is very important and is, in a sense, more fundamental than the data and instruction caches. All memory accesses require that a virtual address is translated into a physical address. Since these translations are done so commonly, the translations themselves need a cache (the TLB).

Let's relate this idea to the five stages of the MIPS datapath which we have recently discussed. Recall the first was instruction fetch, where the current instruction was read from memory. The figure below shows what really happens there. We have replaced the (virtual) Memory unit with two (physical) SRAM units, namely caches. The first cache is the TLB. It translates/maps the 32 bit virtual address to a physical address. (Details below.) The physical address is then used to access the instruction from the instruction cache, assuming the instruction we want is indeed in the instruction cache (called a "hit"). If both the translation and the instruction are in the caches just mentioned, then the instruction indeed can be fetched/read in one clock cycle.

Similarly, the data memory access (MEM stage) requires that we use the TLB to translate from 32 bit MIPS address to a physical address, and then access the data from the data cache.

[ASIDE: in a pipelined system, we would split the TLB in two – one for instructions and one for data. The reason is the same reason as the reason we have a "split cache" for instructions and data, namely so that two different pipeline stages can both have access to a TLB.]



Direct mapping

How is the TLB organized? Recall that the 32 bit MIPS address is partitioned into a virtual page number and a page offset. Let's continue with the example from earlier in which these two fields are 20 and 12 bits, respectively. Suppose that the TLB has 512 (2^9) entries. Then, we can partition the 20 bit virtual page number into two components:

- a *TLB index* (lower 9 bits of VPN). The index is used as an address into the TLB.

- a *tag* (upper 11 bits of VPN). The tag is used to disambiguate the 2^{11} virtual page numbers that have that 9 bit index.

[ASIDE: Those of you who know how hash tables work will recognize this idea. The TLB index is the hash key, and the tag is the hash value.]

For example, consider the following example. Suppose we have two (32 bit) virtual addresses in some program:

```
  (tag, 11)     (TLB index,9)  (page offset, 12)
 01010100100      001001011      010101111111
 01010100100      001001011      001001001001
```

Because these two virtual addresses have the same values in their TLB index fields, their translations would be mapped to the same entry of the TLB. Notice that they happen to have the same tag as well, and so their 20 bit virtual page numbers (VPN) are the same. Thus these addresses lie on the same page.

Now consider a slightly different example:

```
  (tag, 11)     (TLB index,9)  (page offset, 12)
 01000111011      001001011      010101111111
 01010100100      001001011      001001001001
```

Here the TLB index is the same, but now the tags differ (and so the virtual page numbers differ as well.) Since the virtual page numbers differ, the pages differ. Hence the physical page numbers must also differ. Only one of these VPN's can be represented in the TLB at any one time.

Another issue to be aware of is that, at any time, the TLB may contain translations from several different processes. One might think that this should not be allowed and that the TLB should be flushed (all values set to 0) every time one process pauses (or ends) and another continues (or starts). However, this policy would be unnecessary. Often TLBs are designed so that they can contain translations from multiple processes at any time, (that is, a TLB can contain entries from different page tables). In order to distinguish which page table an entry belongs to, one add another field to the table called a *Process ID (PID)*. It is a number identifying the process. [1] A translation in the TLB can only be used if the ProcessID in the TLB matches the ProcessID for the current process (which itself is held in a special register).

Finally, each entry of the TLB has a valid bit which says whether the entry correspond to a valid translation or not. For example, when a process finishes running, the TLB entries that were used by that process are no longer valid and the so the valid bits for those entries are set to 0.

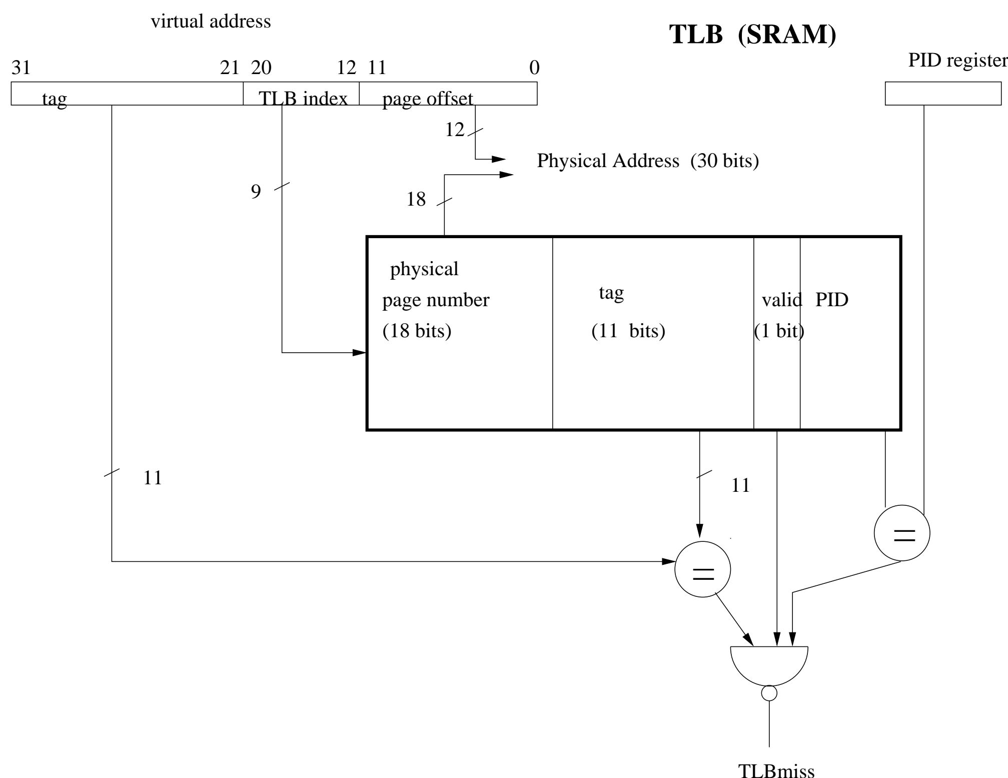
The tag field is there to ensure that the VPN of the instruction (or data) that we are accessing is the same as the VPN of the entry stored in the TLB. The row of the TLB is specified by the TLB index, and the tag specifies the remaining VPN bits.

Have a look again at the datapath on page 1. What circuit is inside the little box that says "TLB"? We see that there is alot of detail hidden there. Consider a 32 bit address (either the PC, or an address leaving the ALU stage). This 32 bit virtual address is partitioned in fields mentioned

[1] The concept here should be familiar to you. Consider telephone numbers and area codes. Seven digit phone numbers are virtual addresses. Area codes are like PIDs. Area codes are used to distinguish 7-digit phone numbers in different cities.

above. To ensure that the TLB contains the translation for this virtual address, a row from the TLB must be retrieved (using the TLB index). The TLB valid bit must be checked and must have the value 1. The tag field must match the upper bits of the virtual address. The Process ID field (PID) must match the Process ID of the current process. If all these conditions are met, then the physical page number stored at that entry of the TLB is concatenated with the page offset bits of that virtual address (see figure below) and the result defines the physical address i.e. the translated address. Since SRAM is fast, all this happens within one clock pulse.

In the example below, main memory has 1 GB (30 address bits), the TLB has $2^9 = 512$ entries. Pages are 4 KB (12 bits for page offset), so there are 18 bits used to address pages in main memory.[2] (i.e. $18 + 12 = 30$).



One final point: last lecture we said that pages can be either in RAM or on disk, and we saw that there are two lengths of physical addresses – one for RAM and one for disk. The TLB does not store translations for pages on disk, however. If the program is trying to address a word that only lies on the hard disk, then (by design) it is impossible that the TLB will have the translation for that address; the TLB only holds translations for addresses that are in RAM. In order to have access to that word, a page fault will need to occur and the page will need to be copied from the hard disk to main memory. We will discuss how this is done in the next lecture.

[2]Here we are ignoring the fact that some parts of main memory are not broken into pages. This was discussed last class. For example, the design is simpler if the page tables themselves are not stored in paged part of memory.

Data and instruction caches

Recall the sketch from page 1. Two types of SRAM are used to quickly access instructions or data, namely the TLB and the cache. We just explained how the TLB is used to quickly translate virtual addresses to physical addresses. Now let's look at the instruction and data caches.

Suppose our cache contains 128 KB (2^{17} bytes) and again suppose our main memory contains 1 GB (2^{30} bytes). How should the cache be organized and accessed?

case 1: each entry has a single byte

Suppose the cache entries were single bytes. Then 17 bits would be needed to specify a cache address. The simplest way to map the 30 bit main memory address to the 17 bit cache address is to use the lower 17 bits of the physical (main memory) address. This implies that any two main memory addresses that had the same lower 17 bits would index to the same cache address. (This is just the same *direct mapping* method that we used in the TLB.)

Using this approach, the cache memory would be organized as follows. For each of the 2^{17} entries in the cache, we store four fields:[3]

- the byte of data
- the upper 13 bits of the 30 bit physical address of that byte, called the *tag*. The idea of the tag is the same as we saw for the TLB. In this case, we need to distinguish entries whose physical addresses have the same lower 17 bits.
- a single bit that specifies whether there is indeed something stored in the cache entry, or whether the tag and byte of data are junk bits, for example, left over by a previous process that has terminated. This single bit is called the *valid* bit.
- a *dirty* bit. This says whether the byte has been written to since it was brought into memory. (We will discuss next lecture how this bit is used.)

Let's now run through the sequence of steps by which an instruction accesses a byte in the cache. Suppose a MIPS processor is executing a `lb` instruction (load byte). First, the address of the byte to be loaded is translated from virtual (32 bits) to physical (30 bits). We saw above how the TLB does this. Then, the 30 bit physical address is used to try to find the byte in the cache. How is this done? There are two steps.

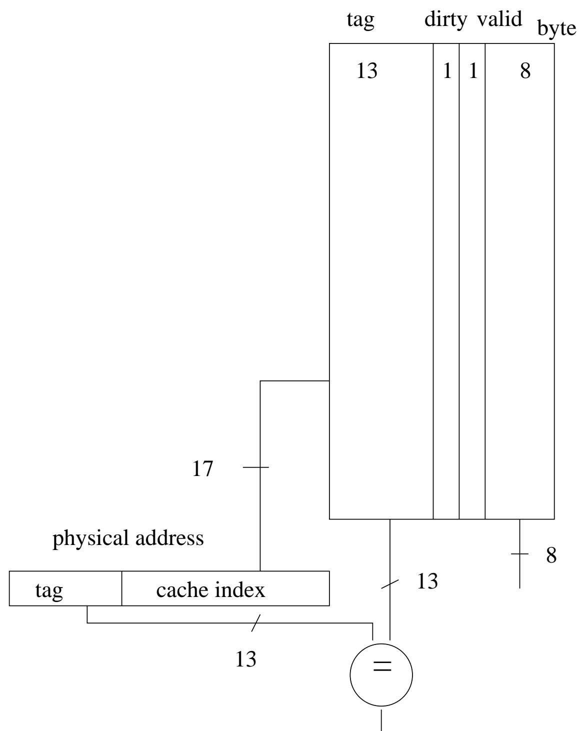
First, the lower 17 bits of the physical address are used as an index into the cache. (Back in lecture 6, we looked at basic designs for retrieving bits from memory. You do not need to refresh all the details here, but certainly you should be comfortable with the idea that you can feed 17 bit address into a circuit and read out a byte of data!) A line of the cache is read out, containing all the fields mentioned above.

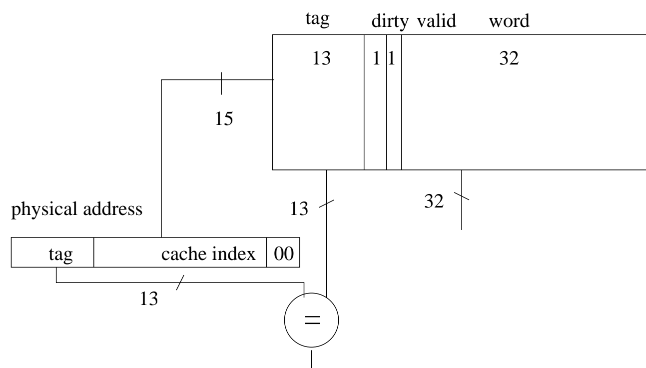
Second, the upper 13 bits are compared to the 13 bit tag field at that cache line. If the two are the same, and if the valid bit is on, then the byte of data stored at that entry of the cache is the correct byte. And so the byte can be read from the cache. If the upper 13 bits don't match the

[3]Unlike in the TLB, we do not use a PID field here. (We could, but we don't.) It is possible for two processes to use the same physical addresses, for example, two processes might be using the same instructions if they are two copies of the same program running or if they are sharing a library.

tag or if the valid bit is off, then the byte cannot be loaded from the cache and that cache entry needs to be refilled from main memory. In this case, we say that a *cache miss* occurs. This is an exception and so a kernel program known as the *cache miss handler* takes over. (More on this next lecture.)

case 1 (one byte per entry) **case 2 (one word per entry)**





case 2: each entry has one word

Since MIPS instructions and data are often one word (4 bytes), it is natural to access 4 bytes at a time from memory. Suppose we keep the number of bytes fixed at 128 KB, as in case 1, but we reorganize so we have one word per entry. Now the cache would now have 2^{15} lines, holding one word each. Again we would need 13 bits for the tag.

The lowest two bits of the address are called the *byte offset* since they specify one of the four bytes within a word. We are assuming the cache words are word aligned. That is, the four bytes within the cache have LSBs of 11,10,01,00.

case 3: blocks (exploiting spatial locality)

Case 2 took advantage of a certain type of "spatial locality" of memory access, namely that bytes are usually accessed from memory in four-tuples (words). A second type of spatial locality is that instructions are typically executed in sequence. (Branches and jumps occur only occasionally). At any time during the execution of a program we would like the instructions that follow the current

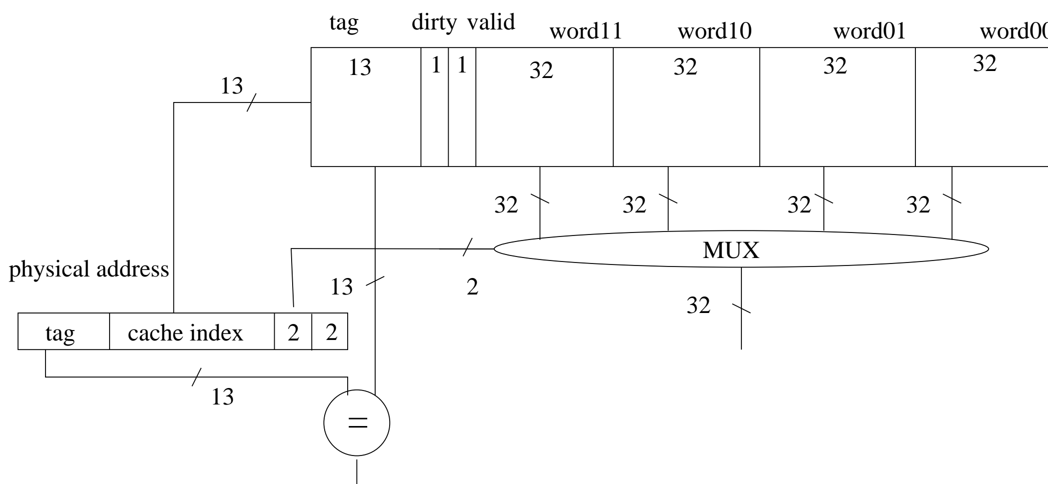
one to be in the cache, since such instructions are likely to be executed shortly. Thus, whenever an instruction is copied into the cache, it would make sense to copy the neighboring instructions into the cache as well.

Spatial locality also arises for data accesses. For example, arrays are stored in consecutive program (virtual) addresses. These tend to correspond to consecutive physical addresses. Similarly, words that are nearby on the stack tend to be accessed at similar times during the process execution.

What is a simple way to implement this idea of spatial locality? Rather than making the lines (rows) in the cache hold one word each, we let them hold (say) 4 words each. By design, we take four words that are consecutive in memory, that is, consecutive in both virtual memory, and in physical memory. These consecutive words are called a *block*. If our cache holds 128 KB, then it can hold up to 2^{13} blocks, so we would use 13 bits to index a line in the cache.

To address a byte within the four words (16 bytes) of each block, we need four bits, namely the four lowest bits of the physical address. Bits 0 and 1 are the "byte offset." The next two LSBs (bits 2,3) are the block offset. They specify one of four words within the block. Note that each block is 16 bytes (4×4) and the blocks are block aligned – they always start with the physical memory address that has 0000 in the least significant four bits.

As shown in the figure below, to access a word from the cache, we read an entire block out of the cache. Then, we select one of the four words in that block.



Fully associative cache

While the direct mapping method used above is simple, it turns out that it doesn't give the best performance in practice. The main problem is that there are many different physical addresses that can map to one cache line. Suppose there are two physical addresses that belong to different blocks but they map to the same cache line, and that these two physical addresses are accessed often. For example, these could be two instructions in a program that are executed frequently. If we use direct mapping then only one of the two blocks can be in the cache at any one time. This means that we will get many cache misses.

A radically different scheme, known as the *fully associative cache*, is to allow blocks to occur in any line of the cache. When checking if a word is in the cache, we don't index only one cache line. Rather, examine *all* cache lines! To do so, we let the tag be the entire block number. For example, if our physical addresses (RAM) were 30 bits and there were 16 bytes per block (4 words $\times$ 4 bytes/word), then the block number would be 26 bits. For *each row* of the cache, we would need a circuit that takes the block number (26 bits) of the block containing the desired word and compares it to the 26 bit tag of the block stored at that line. If the tags match, then the blocks match and that cache row contains the desired word.

Is everything wonderful now? Not really. The above scheme has its good points, namely maximum flexibilty in where a block can be stored in the cache. However, it has bad points too. In order to compare the tags at each line, we would need special circuits to do so at each cache line. Such circuits take space. Ultimately, the space they use means that we have less space for other things – like blocks! So, in order to fit this design on a given physical area (a silicon chip), we would need to use fewer blocks that we could have used in the direct mapping case. And using fewer blocks means that we store less in the cache. Another cost is that we need to decide which cache line to remove, and this could be complicated since it would depend on factors such as which cache line was least recently used. So, there are tradeoffs to consider.

[ASIDE: I am not drawing the circuit in detail, since the idea here is not the hardware implementation but rather the idea is what problem is being solved. In the slides, I sketched out loosely how you can think of this circuit though.]

(N way) set associative cache (Advanced Topic)

A compromise between a direct mapped and fully associative cache is the *set associative cache.* The idea here is to use N direct mapped caches, such that any memory block can be put into any of the N direct mapped caches.

Equivalently, you can think of this design as consisting of *sets* of fully associative caches, each consisting of N blocks. Then,

$$\text{number of sets} = \frac{\text{number blocks in cache}}{N}$$

For example, if our cache has 2^{13} blocks in total (like in our example above), and we used $N = 2$ way associativity, then we would have 2^{12} sets, each set consisting of $N = 2$ blocks. We sometimes refer to N as the *set size.*

To index the cache, we take the block number and compute the set index using direct mapping:

$$\text{set index} = (\text{block number}) \mod (\text{number of sets})$$

We then check each of the N lines of the cache that correspond to this set index. (See figures in slides.) We match the tags to see if any of the N physical addresses match. If so, then we have a cache hit.

How many bits are there in the tag field? Suppose again we have 2^{13} blocks in the cache, and suppose $N = 2$. Then each of the direct mapping caches has 2^{12} blocks, i.e. we have 2^{12} sets of 2 blocks each. Our physical memory has $2^{26} = 2^{30}/2^4$ blocks, so we need 26 bits for a block number. The direct mapping has 2^{12} sets (see above), and so our tag would be $14 = 26 - 12$ bits.