[ASIDE: The ordering of slides in actual lecture was a bit confusing, since I put a brief discussion of the TLB in the middle. That discussion really belonged at the end of last lecture, but there was no time then, because it was a quiz day and there alot of interaction in that class. I have rearranged the slides to try to avoid this confusion, and the notes below corresponds to this new arrangement. The notes below contain much more information than I gave in the lecture. I will go over that information a few lectures from now, when I return to page faults and disk accesses.]

TLB miss and TLB refill (and page fault)

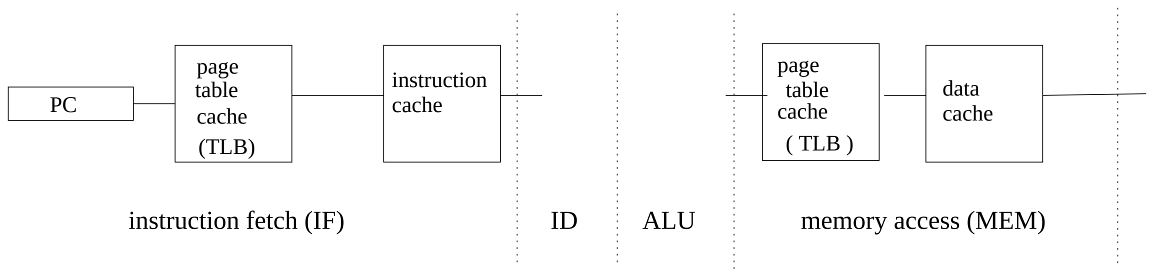
Last lecture I discussed the TLB and how virtual addresses are translated to physical addresses. I only discussed the cases of TLB hits. Here I will briefly discuss TLB misses, namely what happens if the TLB doesn't contain a desired virtual-to-physical translation. If a TLB miss occurs, an exception results. The program jumps to the exception handler which analyzes what the exception was, and then jumps to a special kernel program which handles TLB misses – namely the *TLB miss handler*. This kernel program consults the page table (in main memory) of the current process, to see if the desired word is in main memory i.e. if that page table entry's valid bit is 1.

If the page valid bit is 1 then this physical page containing the address we want is in main memory. (Indeed, the page valid bit could be called instead the 'physical page is in main memory' bit.) The TLB miss handler retrieves the physical page number from the page table and fills that entry in the TLB (called a *TLB refill*), and sets the valid bit for that TLB entry to 1. The kernel then returns control to the program so that it can continue its execution, namely it can perform the virtual-to-physical translation that had caused the TLB miss.

If, however, the page valid bit is 0, then the page that the program wants to access is not in main memory. Rather it is on disk, and so a page fault occurs: the TLB miss handler calls the *page fault handler*, and the page fault handler arranges for the desired page to be copied from the disk to main memory. (We'll see how in a few lectures from now when we talk about disk accesses.) The page fault handler then updates the page table appropriately, namely it changes the page valid bits and physical page numbers. If a page swap occured (so a page was also transferred from main memory to disk), then two entries of the page table must be changed, namely the entries corresponding to incoming and outgoing pages. The page fault handler then returns to the TLB miss handler. Now, the requested page is in main memory and the page valid bit in that entry of the page table is 1. So the TLB miss handler can copy the page table entry into the TLB. The TLB miss handler then returns control to the original process.

Data and instruction caches

The TLB provides a quick translation from a virtual address to a physical address in main memory. However, main memory is still a bit too slow to use everytime we want an instruction or we want to access a data word. To speed up memory accesses, this is why we also use a cache for instructions and data.



Suppose the cache contains 128 KB (2^{17} bytes).[1] [**ADDED March 20: Here I am only counting the bytes for the instructions (or data); I am not counting the extra bits that are needed for the tag, and the dirty and valid bits.] These extra bits can be significant (see Exercises 6 Q2).**] Also, I am not distinguishing instruction and data cache. Also, again suppose our main memory contains 1 GB (2^{30} bytes). How is a cache organized and accessed?

case 1: each cache entry has one word (plus tag, etc)

Since MIPS instructions and data are often one word (4 bytes), it is natural to access 4-tuples of bytes at a time. Since the number of bytes in the instruction cache (or data cache) is 2^{17} and we have one word per entry, the cache would have 2^{15} entries holding one word (2^2 bytes) each.

Let's now run through the sequence of steps by which an instruction accesses a word in the cache. Suppose the processor is *reading* from the cache. In the case of an instruction cache, it is doing an instruction fetch. (In the case of a data cache, it is executing say a `lw` instruction.) The starting address of the word to be loaded is translated from virtual (32 bits) to physical (30 bits). We saw last class how the TLB does this. Then, the 30 bit physical address is used to try to find the word in the cache. How?

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

The next fifteen bits (2-16) are used to index into the 2^{15} entries in the cache. We are assuming a read rather than write, so the entry is then read out of 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 it would be helpful to refresh the basic idea that you can feed 15 bits into a circuit and read out a row of data from a 2D array of flipflops.)

The upper 13 bits of the physical address (the tag) are compared to the 13 bit tag field at that cache entry. If the two are the same, and if the valid bit of that entry is on, then the word sitting at that entry of the cache is the correct one. If the upper 13 bits don't match the tag or if the valid bit is off, however, then the word 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

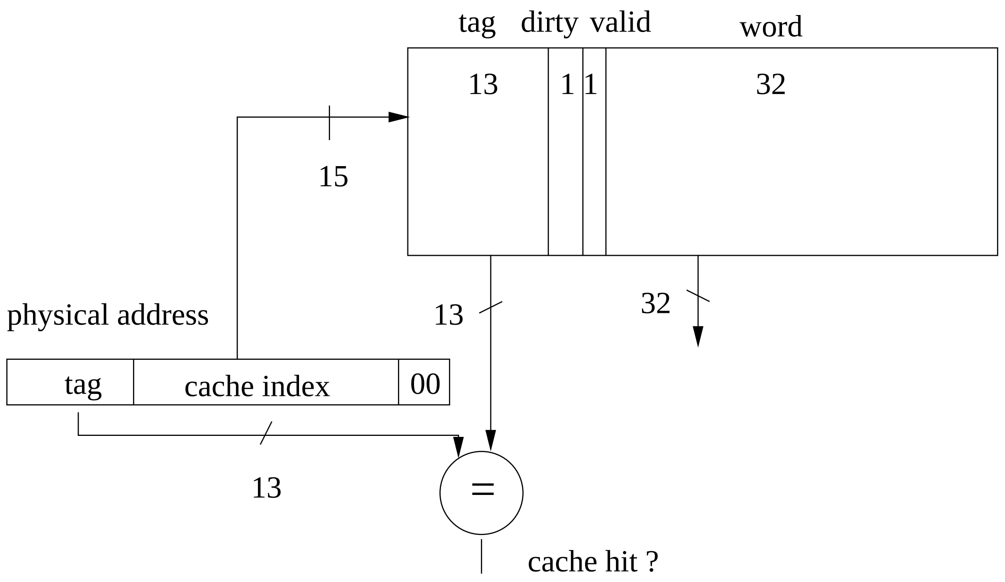
[1] Note that this is significantly larger than the TLB size from last lecture. In principle, the TLB and cache sizes could be similar. There are technical reasons why the TLB is typically smaller, which I am omitting. (It has to do with the mechanism for indexing. I am only presenting one method for indexing, called "direct mapping", but there are other schemes that use more complicated circuits. These are called "set associative" and "fully associative" caches.

and so a kernel program known as the *cache miss handler* takes over. (We will return to this later in the lecture.)

Using this approach, the cache memory would be organized as shown in the circuit below. For each of the 2^{15} word entries in the cache, we store four fields:

- the word, namely a copy of the word that starts at the 30 bit physical (RAM) address represented by that cache entry
- the upper 13 bits of that 30 bit physical address, called the *tag*. The idea of the tag is the same as we saw for the TLB. The tag is needed to distinguish all entries whose physical addresses have the same bits 2-16.
- a *valid* 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, leftover by a previous process that has terminated.
- a *dirty* bit that says whether the byte has been written to since it was brought into memory. We will discuss this bit later in the lecture.

The case of a write to the data cache e.g. `sw` uses a similar circuit. But there are some subtleties with writes. I will discuss these later.



[ASIDE: Unlike in the TLB, there is no process id field PID in the cache. The reason is that 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. Or they might be sharing data.]

case 2: each cache entry has a block of words (plus tag, etc)

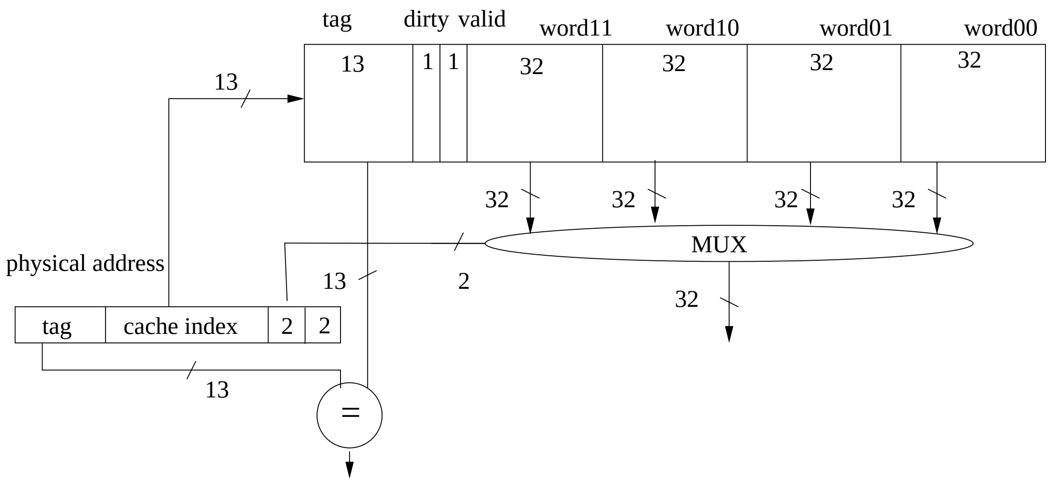
Case 1 above took advantage of a certain type of "spatial locality" of memory access, namely that bytes are usually accessed from memory in four-tuples (words). This is true both for instructions and data. Another type of spatial locality is that instructions are typically executed in sequence. (Branches 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 all executed. 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 word accesses. For example, arrays are stored in consecutive program addresses. Because pages are relatively large, neighboring program addresses tend to correspond to neighboring physical addresses. Similarly, words that are nearby on the stack tend to be accessed at similar times during the process execution.

A simple way to implement this idea of spatial locality is as follows: Rather than making the lines (rows) in the cache hold one word each, we let them hold (say) four words each. We take four words that are consecutive in memory (namely consecutive in both virtual memory and in physical memory). These consecutive words are called a *block*. If our cache holds $2^{17} = 128$ KB and each block holds 4 words or $4 \times 4 = 2^4$ bytes, then the cache can hold up to 2^{13} blocks, so we would use 13 bits to index a line in the cache.

To address a word within a block, we need two bits, namely bits 2 and 3 of the physical address. (Bits 0 and 1 are the "byte offset" within a word.) Bits 2,3 are the block offset which 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. This might always work best, but it simplifies the circuitry.

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. One could draw a similar circuit for writing a word (or block) to the cache, although the multiplexor part would need to be changed.



Hits and misses

The above discussion was about *indexing* mechanisms, namely how we read from a cache assuming that the cache has the word we want (a hit). We next consider two other aspects of caches. The first is what happens when the address we are indexing is *not* in the cache: a cache *miss*. The second aspect is what happens when we write to a cache, either from a register to the cache (in the case of a store word), or when we copy a block from main memory to the cache in the case of a miss. Specifically, we are concerned here with consistency between the cache and main memory.

Instruction Cache

The instruction and data caches have a subtle difference: instructions are only fetched (read) from memory, but data can be read from or written to memory. For the instruction cache, blocks are copied from main memory to the cache. For the data cache, blocks can be copied either from main memory to the cache, or vice-versa. There can also be writes from registers to the data cache e.g. by `sw` (or `swc1`) instructions.

We begin with the instruction cache, which is simpler. If we are fetching and the instruction is in the cache, i.e. a *hit*, then we have the case discussed earlier in the lecture. If the instruction is not in the cache, however, we have a *miss*. This causes an exception – a branch to the exception handler which then branches to the *cache miss handler*. This kernel program arranges for the appropriate block in main memory (the one that contains the instruction) to be brought into the cache. The valid bit for that cache entry is then set, indicating that the block now in the cache indeed represents a valid block in main memory. The cache miss handler can then return control to the process and we try again to fetch the instruction. Of course, this time there is a hit.

Data Cache - write-through policy

The data cache is more complicated. Since we can write to the data cache (`sw`), it can easily happen that a cache line does not have the same data as the corresponding block in main memory. There are two policies for dealing with this issue: "write-through" and "write-back". The write-through policy ensures that the cache block is consistent with its corresponding main memory block. We describe it first.

Consider reading from the data cache (as in `lw or lwc1`). If the word is in the cache, then we have a hit and this case was covered earlier. If there is a miss, however, then an exception occurs and we replace that cache entry (namely, the entire block). The previous entry of the cache is erased in the process. This is no problem for the write-through policy since this policy ensures that the cache line (just erased) has the same data as its corresponding block in main memory, and so the erased data is not lost.

Consider happens when we write a word from a register to the cache (`sw` or `swc1`). First suppose that there is a hit: the cache has the correct entry. The word is copied from the register to the cache *and also* the word is copied back to the appropriate block in main memory, so that main memory and cache are consistent (i.e. write through).

If the desired block is not in the cache, then an exception occurs – a *cache miss*. The *cache miss handler* arranges that the appropriate block is transferred from main memory to the cache. The handler then returns control to the program which tries to write again (and succeeds this time – a cache hit). Here is a summary of the"write through (data cache)" policy:

	hit	miss
read (`lw`)	copy word cache $\rightarrow$ reg	copy block main mem $\rightarrow$ cache (and set valid bit) copy word cache $\rightarrow$ register
write (`sw`)	copy word reg $\rightarrow$ cache copy word cache $\rightarrow$ main mem	copy block main memory $\rightarrow$ cache (and set valid bit) copy word register $\rightarrow$ cache copy word cache $\rightarrow$ main memory

Data cache: "write back" policy

The second policy avoids copying the updated cache block back to main memory unless it is absolutely necessary. Instead, by design, each entry in the cache holds the most recent version of a block in main memory. The processor can write to and read from a block in the cache as many times as it likes without updating main memory – as long as there are hits. The only special care that must be taken is when there is a cache miss. In this case, the entry in the cache must be replaced by a new block. But before this new block can be read into the cache, the old block must be written back into main memory so that inconsistencies between the block in the cache (more recent version) and the block in main memory (older version) are not lost. This is how the "write back" scheme delays the writing of the block to memory until it is really necessary.

To keep track of which lines in the cache are consistent with their corresponding blocks in main memory, , a *dirty bit* is used – one per cache line. When a block is first brought into the cache, the dirty bit is set to 0. When a word is written from a register to a cache block (e.g. `sw`), the dirty bit is set to 1 to indicate that at least one word in the block no longer corresponds to the word in main memory.

Later, when a cache miss occurs at the cache line, and when the dirty bit is 1, the data block at that cache line needs to be written back to main memory, before the new (desired) block can be brought into the cache. That is, we *"write-back"*. This policy only writes a block back to main memory when it is really necessary, i.e. when the block needs to be replaced by another block. Note that there is just one dirty bit for the whole block. This bit doesn't indicate which word(s) is dirty. So the whole block is written back to main memory and a whole new block is brought in.

This "write-back" policy helps performance if there are several writes to individual words in a block in the cache before that block is replaced by another.

The following table summarizes the steps of the data cache write-back policy.

	hit	miss
read	copy word: cache → reg	copy block: cache → main mem *(only if valid and dirty bits are 1)* copy block: main memory → cache and set dirty bit = 0, valid bit = 1 copy word: cache → register
write	copy word: reg → cache (and set dirty bit = 1)	copy block: cache → main mem *(only if valid and dirty bits are 1)* copy block: main mem → cache (and set valid bit = 1) copy word: reg → cache (and set dirty bit = 1)

Note that the misses take more time than the hits. So this policy only makes sense when the hits are much more frequent than the misses.

Notice that in the write back scheme, a "write hit" is cheaper and a "read miss" is more expensive. For large caches, the hit rate is typically over 95 per cent. For this reason, a write back policy tends to give better performance than the write through policy for large caches.