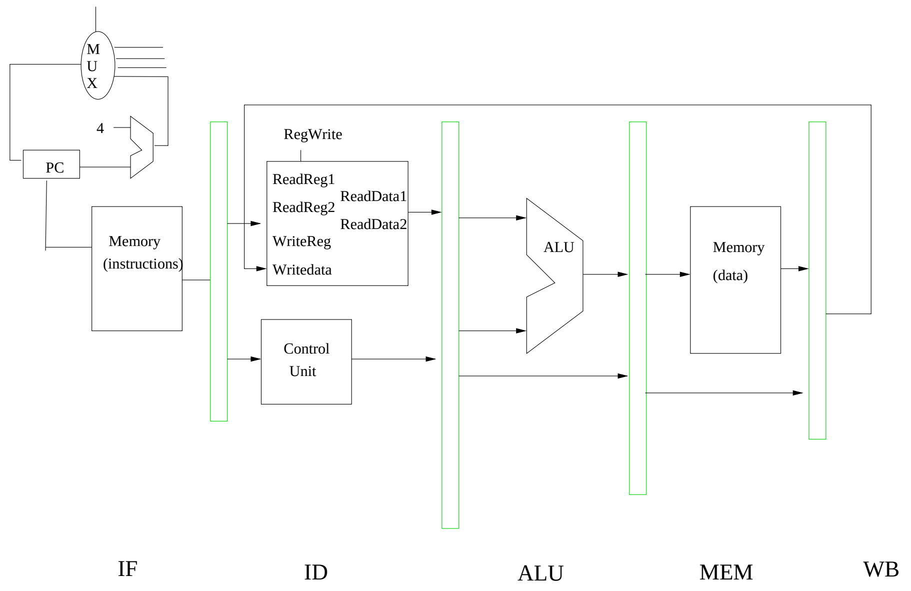
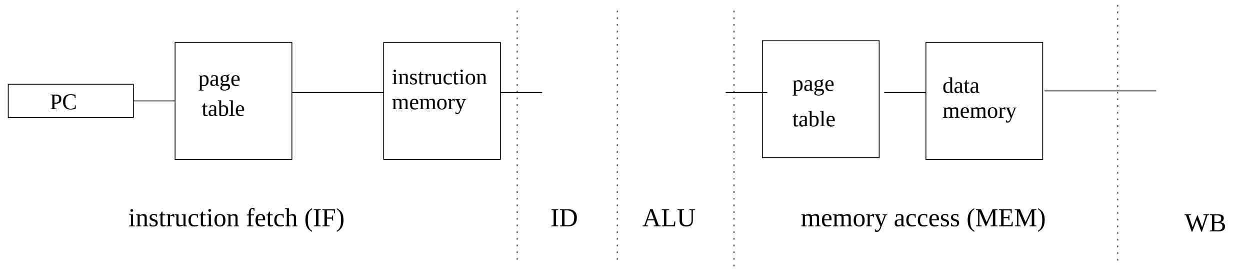
Cache: motivation

Last lecture we discussed the different types of physical memory and the relationship between program (virtual) memory addresses and physical addresses. We also discussed the memory hierarchy, and how we can't have quick access to all physical memory. Instead, we need to use a combination of small fast expensive memory (SRAM), slower and bigger less expensive memory (DRAM), and much slower and much bigger and much cheaper memory (disk).

How do these ideas about memory back to the datapaths that we learned about earlier? Recall the basic pipeline scheme:



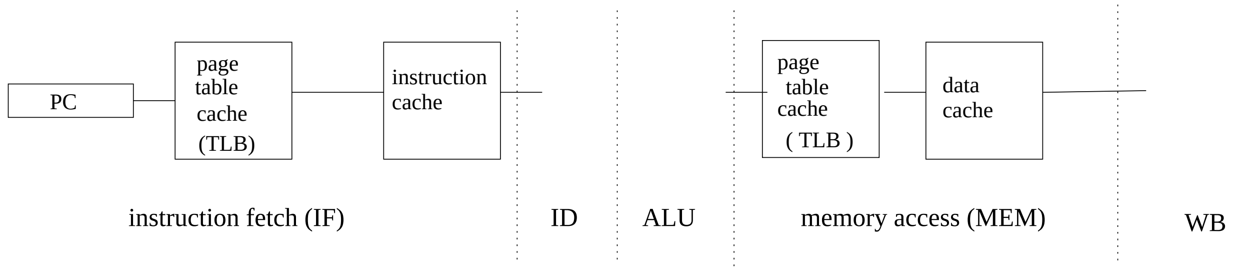
There, "Memory" box was a placeholder for an unspecified mechanism. Then, last lecture we learned that the memory accesses involve two steps: first, translate the 32 bit virtual address to a physical address (using a page table lookup); second, use the physical address to index into the physical memory. So we now think of the IF and MEM stages as shown below (two stages each). This is still an abstract representation, since a "page table" is not a circuit, not is it clear yet what we mean by the "instruction memory" and "data memory" boxes. Do we mean RAM or disk, for example? It will take a few more lectures to unpack this.



If we think of the two stage memory accesses as being from main memory (RAM) or from disk, then we see immediately that we have a speed problem. Even if the word we want is in main memory rather than on disk, each main memory (DRAM) access requires many clock cycles (say

10). Thus, every instruction would require many clock cycles. We would need about 10 cycles to ge the translation from the page table, and we need about 10 cycles to fetch the instruction, and we would need another 10 clock cycles or load or store the word.

The solution to the problem is to use small very fast memory (SRAM) to hold frequently used page table entries (virtual to physical page number translations), and frequently used instructions and data. These small and very fast SRAM memories are called *caches*. For now, we'll just think of them as boxes which have an input and an output, where each box is a sequential circuit similar to the register array (but much bigger in practice). In the figure below, the input to the page table cache is a virtual address and the output is a physical address. The input to the instruction or data cache is a physical address and the output is an instruction or data word. (For the data cache, it is also possible to write into it (e.g. `sw`). We will get to that next lecture.)



Page table cache (”Translation lookaside buffer” or TLB)

Let's first consider the page table cache which is historically is called the *translation lookaside buffer* (or TLB). This cache is as important as the data and instruction caches, since 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 to be cached.

You can think of the TLB as being split in two parts – one for instruction address translations and one for data address translations. I will often keep the discussion simple by just referring to ”the” TLB.

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

- a *TLB index* (lower 9 bits of VPN). The index specifies which out of $2^9 = 512$ entries in the TLB that we are referring to. You can think of it as an address of an entry in the TLB.
- a *tag* (upper 11 bits of VPN). The tag is used to disambiguate the 2^{11} virtual page numbers that have a given 9 bit index.

For example, suppose we have two (32 bit) virtual addresses:

```
 (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 VPN's would be mapped to the same entry of the TLB. Notice that these addresses happen to have the same tag as well, since their 20 bit virtual page numbers (VPN) are the same, that is, these addresses lie on the same page. Only the page offset of the addresses differ.

Now consider a slightly different example:

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

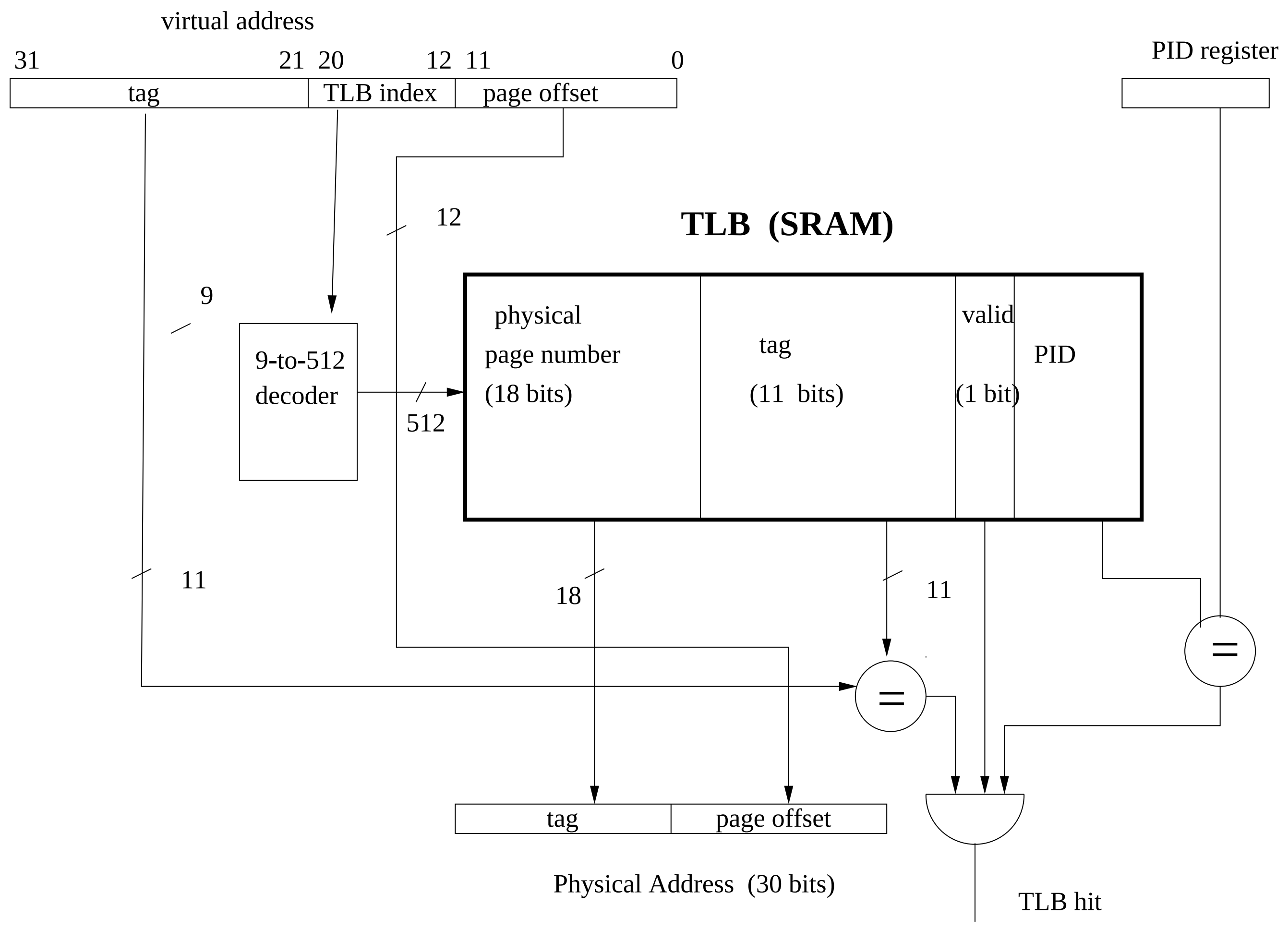
Here the TLB index and page offsets are the same, but the tags differ (and so the virtual page numbers differ as well.) The VPN's differ (since the tags differ). Since two different virtual pages must be represented physically by different physical pages, the physical page numbers must also differ. But since both VPNs map to the same TLB entry, it follows that only one of these two translations can be represented in the TLB at any one time. More details on how this works will be given below.

Another issue to be aware of is that the TLB can 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 harsh flushing policy is unnecessary: there is an easy way to keep track of different processes within the TLB. To distinguish which process (and hence, which page table) an entry belongs to, one can add another field to each entry of the TLB, namely a *Process ID (PID)* field. This is simply a number that identifies the process. The concept here should be familiar to you. PID's are like area codes for telephone numbers. They are used to distinguish possibly the same 7-digit phone numbers in two different cities.

Finally, each entry of the TLB has a valid bit which says whether the entry corresponds 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 considered valid and the so the valid bits for all entries of that process are set to 0.

Have a look again at the datapath on page 2. What circuit is inside the little boxes that say "page table cache (TLB)"? The answer is shown below. Consider a 32 bit virtual address which must be translated into a physical address. This virtual address is partitioned into tag/index/offset fields. To ensure that the TLB contains the translation for this virtual address, the indexed row from the TLB is retrieved. (Specifically, a decoder is used to identify which row to retrieve from. Recall the RowSelect control mechanism from lecture 6.) 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 that entry in the TLB can used for the translation, so 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 the IF stage i.e. within the clock cycle.

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 the TLB will not have the translation for that address; the TLB only holds translations for addresses that are in main memory (RAM). In order to have access to that word on disk, a page fault would need to occur and the page would need to be copied from the hard disk to main memory. We will discuss how this is done later in the course.


virtual address
31
21 20
12 11
0
tag
TLB index
page offset
PID register
TLB (SRAM)
12
9
9-to-512 decoder
512
physical page number (18 bits)
tag (11 bits)
valid (1 bit)
PID
11
18
11
=
=
tag
page offset
Physical Address (30 bits)
TLB hit