CS61 Section Notes
Week 7 (Fall 2010)

Virtual Memory

We are given a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 4-byte words
- Physical addresses are 16 bits wide.
- Virtual addresses are 20 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal.** The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

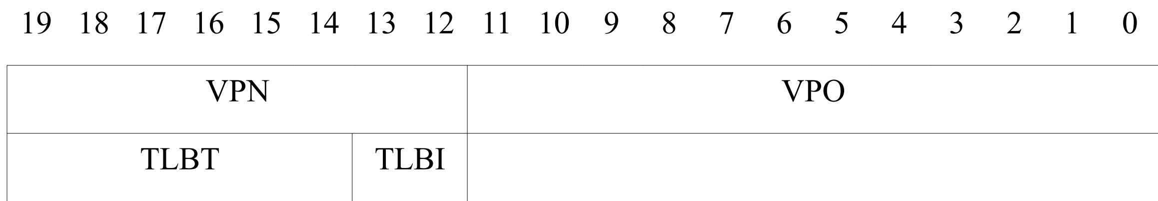
Question 1a: The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

VPO The virtual page offset

VPN The virtual page number

TLBI The TLB index

TLBT The TLB tag



Question 1b: The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO The physical page offset

PPN The physical page number

For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `0x7E37C`

Question 2a: Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	0	1	1	0	1	1	1	1	1	0	0

Question 2b: Address translation

Parameter	Value
VPN	0x7E
TLB Index	0x2
TLB Tag	0x1F
TLB Hit? (Y/N)	yes
Page Fault? (Y/N)	no
PPN	0x8

Question 2c: Physical address format (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1	0	1	1	1	1	1	0	0

Virtual address: `0x16A48`

Question 3a: Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0

Question 3b: Address translation

Parameter	Value
VPN	0x16
TLB Index	0x2
TLB Tag	0x5
TLB Hit? (Y/N)	no
Page Fault? (Y/N)	yes
PPN	0x-

Question 3c: Physical address format (one bit per box)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

4. Doesn’t virtual memory ruin locality, since now blocks of memory that are contiguous in virtual memory may be spread far apart in physical memory? Why is locality still important?

Pages (~4K) tend to be larger than cache blocks (~128 bytes), so consecutive blocks of virtual memory will still likely be consecutive in physical memory.

More about Caching

Questions:

5. Why don't we include the tags bits when measuring a cache's size?

The tag bits (and the valid bit) do not hold any cache data so they are not counted in the size of the cache.

6. Why are the *s* bits chosen from the middle of the address? Why not just use the *s* high bits?

If the high-order bits are used as an index, than some contiguous memory blocks will map to the same cache set. If the middle bits are used, adjacent blocks always map to different cache lines.

Set Associative Caches

The code we saw last week causes the cache to *thrash*. A cache *thrashes* when it repeatedly loads and then evicts the same set of cache blocks. We can alleviate this issue with a cache that has more than one cache line per set. A cache with $1 < E < C/B$ is called an E-way set associative cache.

Example:
Consider a 128 byte, 2-way set associative cache where each cache line contains 16 bytes on a 16-bit processor with 2^{16} addressable bytes of memory, and assume the cache is initially empty (the valid bits are all zero). Let's walk through how accessing the memory locations 0x10, 0x20, 0x53 and 0x98, in that order, interacts with the cache.

Here is how the addresses are partitioned into tag bits, set index bits and block offset bits:

Address	`t` Tag bits	`s` Set index bits	`b` Block offset bits
`0x10`	`0000000000`	`01`	`0000`
`0x21`	`0000000000`	`10`	`0001`
`0x53`	`0000000001`	`01`	`0011`
`0x98`	`0000000010`	`01`	`1000`

1. When we access address 0x10, we look in set index 1 for a valid entry with tag 0. Because the cache is empty, we have a cache miss. The processor will load the cache from main memory with the 16 bytes starting at location 0x10, add an entry to set 1 with tag 0, and return byte 0 from the newly-loaded cache block.
2. The processor then looks in set 2 for a valid entry with tag 0. Again, no such entry is found, so the cache is loaded from memory and the byte at offset 1 from the newly-loaded memory block is returned.
3. Address 0x53 maps to set 1, just as address 0x10 did. Unfortunately it cannot be found in the valid cache block belonging to set 1 because its tag bits don't match the entry placed in the cache when 0x10 was loaded. Because this is a two-way set associative cache, there is room for two separate cache lines in set 1, so the 16 bytes of memory starting at address 0x50 are loaded into the one remaining empty cache line in set 1. Set 1 now has two valid cache lines.
4. Address 0x98 also maps to set 1, but its tag bits do not match either of the two valid cache lines belonging to set 1, so one of them must be ejected. Assuming an LRU policy, the cache line with tag 0x0 is ejected because it was loaded first. The processor replaces this entry with one containing the tag bits 0x2, loads 16 bytes from main memory starting at address 0x90 into the cache block, and returns the byte at offset 8.

Another Example

Now suppose we have a cache for a byte-addressable memory where physical addresses are 12 bits wide. The contents of this 4-way set-associative cache consisting of 32 2-byte lines is shown below. Each line in the table shows four cache lines, one set.

4-way Set Associative Cache

	Line 0				Line 1				Line 2				Line 3			
Set	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

7. What happens when the physical address `0x3B6` is looked up in the cache?

Given the attributes of the cache, we can determine that the first 8 bits of the address are used for the tag, the next 3 for the index, and the last 1 for the offset, meaning that our tag, set, and offset are 0x3B, 0x3, and 0x0, respectively. Looking up 0x3B in the third set, we see a valid entry with the value 0x66 in its first byte (zeroth offset).

8. What happens when the physical address `0x1CC` is looked up in the cache?

Similar to the previous exercise, we have tag, set, and offset of 0x1C, 0x6, and 0x0. The entry for 0x1C in set 6 is not valid, so the value would have to be fetched from a higher-level cache.

Questions:

9. A set associative cache has more than one cache line per set, so line matching is more complex. How is the time line matching takes related to the number of lines in a set?

In general, it is hard to say how much extra time line matching takes for a set associative cache, since its highly dependent on how it is implemented in hardware. Most likely, it is slower since it requires more tag bits per line and additional control logic. In particular, it may increase hit time. It can also increase the miss penalty because of increased complexity in choosing a victim line.

10. My 128 bytes cache has a 16 byte cache lines, 4 sets, and 8 cache lines in total. What is the name for this kind of cache?

It is called a 2-way set associative cache

Cache Performance

Analysis

In class we measured the sizes of various caches by writing to larger and larger ranges of sequential memory addresses in a stride-1 pattern. When the delay rose significantly, we knew that we were using a different (slower, but larger) cache.

11. How much of an effect do you think the associativity of a cache has on the test program's measurement? If you were going to take a shortcut and use longer strides, how might associativity affect your results?

If you have stride-1 memory access, the associativity of the cache has little effect. If you have longer strides, increasing your associativity will improve performance.

Tuning

12. If you had a cache with a low associativity (think direct-mapped), would you rather have a program with good spatial locality or good temporal locality (but not both)?

Spatial locality.

13. What if you had high associativity?

The higher the associativity of the cache, the less dependent the cache is on spatial locality.

14. In class, we've mostly thought about caches in terms of a data cache. However, we typically will also have a separate instruction cache. Is loop unrolling good or bad for instruction cache performance? How much would you want to unroll your loops for the best performance?

Loop unrolling is bad for instruction cache performance . In the best case you would unroll your loops to fill the instruction cache but no more.

15. How might you organize your program so that it has good instruction cache behavior if you know certain blocks of code are more likely to be called than others?

You would put them as close together as possible.