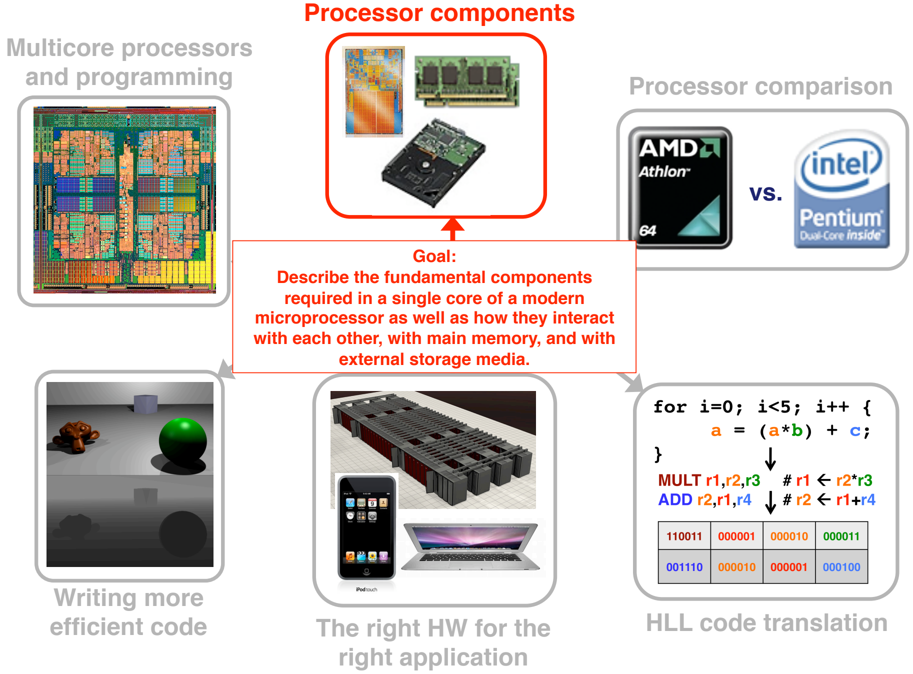
Lecture 19
Cache Organizations

Suggested Readings

- Readings
  - H&P: Chapter 5.2 and 5.3



Processor components

Multicore processors and programming

Processor comparison

Goal:
Describe the fundamental components required in a single core of a modern microprocessor as well as how they interact with each other, with main memory, and with external storage media.

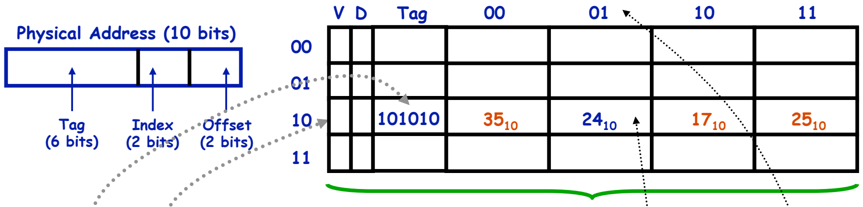
```
for i=0; i<5; i++ {
      a = (a*b) + c;
}
MULT r1,r2,r3    # r1 ← r2*r3
ADD r2,r1,r4     # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

Writing more efficient code

The right HW for the right application

HLL code translation



	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

A 4-entry direct mapped cache with 4 data words/block

1 Assume we want to read the following data words:

Tag		Index		Offset	Address Holds Data
101010	|	10	|	00	35_{10}
101010	|	10	|	01	24_{10}
101010	|	10	|	10	17_{10}
101010	|	10	|	11	25_{10}

All of these physical addresses would have the same tag

All of these physical addresses map to the same cache entry

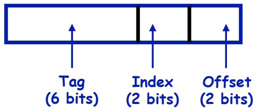
2 If we read 101010 10 01 we want to bring data word 24_{10} into the cache.

Where would this data go? Well, the index is 10. Therefore, the data word will go somewhere into the 3rd block of the cache. (make sure you understand terminology)

More specifically, the data word would go into the 2nd position within the block – because the offset is '01'

3 The principle of spatial locality says that if we use one data word, we'll probably use some data words that are close to it – that's why our block size is bigger than one data word. So we fill in the data word entries surrounding 101010 10 01 as well.

Physical Address (10 bits)



	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

A 4-entry direct mapped cache with 4 data words/block

4 Therefore, if we get this pattern of accesses when we start a new program:

1.) 101010 10 00
2.) 101010 10 01
3.) 101010 10 10
4.) 101010 10 11

After we do the read for 101010 10 00 (word #1), we will automatically get the data for words #2, 3 and 4.

What does this mean? Accesses (2), (3), and (4) ARE NOT COMPULSORY MISSES

5 What happens if we get an access to location:
100011 | 10 | 11 (holding data: 12_{10})

Index bits tell us we need to look at cache block 10.

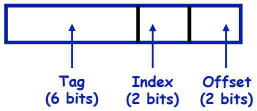
So, we need to compare the tag of this address – 100011 – to the tag that associated with the current entry in the cache block – 101010

These DO NOT match. Therefore, the data associated with address 100011 10 11 IS NOT VALID

What we have here could be:

- A compulsory miss
  - (if this is the 1st time the data was accessed)
- A conflict miss:
  - (if the data for address 100011 10 11 was present, but kicked out by 101010 10 00 – for example)

Physical Address (10 bits)



	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

This cache can hold 16 data words…

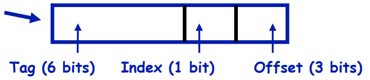
6 What if we change the way our cache is laid out – but so that it still has 16 data words? One way we could do this would be as follows:

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1											

1 cache block entry

All of the following are true:

- This cache still holds 16 words
- Our block size is bigger – therefore this should help with compulsory misses
- Our physical address will now be divided as follows:
- The number of cache blocks has DECREASED
  - This will INCREASE the # of conflict misses



7 What if we get the same pattern of accesses we had before?

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

Note that there is now more data associated with a given cache block.

Pattern of accesses:
(note different # of bits for offset and index now)

1.) 101010 1 000
2.) 101010 1 001
3.) 101010 1 010
4.) 101010 1 011

However, now we have only 1 bit of index.
Therefore, any address that comes along that has a tag that is different than '101010' and has 1 in the index position is going to result in a conflict miss.

7 But, we could also make our cache look like this…

	V	D	Tag	0	1
000					
001					
010					
011					
100			101010	35_{10}	24_{10}
101			101010	17_{10}	25_{10}
110					
111					

There are now just 2 words associated with each cache block.

Again, let's assume we want to read the following data words:

	Tag	Index	Offset	Address Holds Data
1.)	101010	100	0	35_{10}
2.)	101010	100	1	24_{10}
3.)	101010	101	0	17_{10}
4.)	101010	101	1	25_{10}

Assuming that all of these accesses were occurring for the 1st time (and would occur sequentially), accesses (1) and (3) would result in compulsory misses, and accesses would result in hits because of spatial locality. (The final state of the cache is shown after all 4 memory accesses).

Note that by organizing a cache in this way, conflict misses will be reduced.
There are now more addresses in the cache that the 10-bit physical address can map too.

	V	D	Tag	0	1
00					
01					
10					
11					
00			101010	35_{10}	24_{10}
01			101010	17_{10}	25_{10}
10					
11					

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

All of these caches hold exactly the same amount of data – 16 different word entries

	V	D	Tag	000	001	010	011	100	101	110	111
0											
1			101010	35_{10}	24_{10}	17_{10}	25_{10}	A_{10}	B_{10}	C_{10}	D_{10}

As a general rule of thumb, "long and skinny" caches help to reduce conflict misses, "short and fat" caches help to reduce compulsory misses, but a cross between the two is *probably* what will give you the best (i.e. lowest) overall miss rate.

But what about capacity misses?

What's a capacity miss?

- The cache is only so big. We won't be able to store every block accessed in a program – must them swap out!
- Can avoid capacity misses by making cache bigger

	V	D	Tag	00	01	10	11
00							
01							
10			101010	35_{10}	24_{10}	17_{10}	25_{10}
11							

Thus, to avoid capacity misses, we'd need to make our cache physically bigger – i.e. there are now 32 word entries for it instead of 16.

FYI, this will change the way the physical address is divided. Given our original pattern of accesses, we'd have:

	V	D	Tag	00	01	10	11
000							
001							
010			10101	35_{10}	24_{10}	17_{10}	25_{10}
011							
100							
101							
110							
111							

Pattern of accesses:

1.) 10101 | 010 | 00 = 35_{10}
2.) 10101 | 010 | 01 = 24_{10}
3.) 10101 | 010 | 10 = 17_{10}
4.) 10101 | 010 | 11 = 25_{10}

(note smaller tag, bigger index)