1. Cache addressing (continued from last time)
    a. What type of addresses does the cache use?
    b. Cache addresses can be *physical*, using actual memory addresses
        i. Slower, as the cache must wait for the MMU
        ii. Memory management unit (MMU) converts virtual to physical addresses
        iii. However, physically addressed caches don't have to deal with the aliasing problem below
    c. Can also be *logical*, using the same virtual addresses that the process uses
        i. Faster, as the cache doesn't need to wait for MMU
            1. Can reuse the same address that the process was using
        ii. However, these caches encounter the *aliasing* problem
            1. Same virtual address can be used in multiple user processes
            2. Cache would say there's a hit
            3. However, that data doesn't correspond to process currently running
            4. Need to flush cache every time we switch processes
                a. Also known as a *context switch*, which you will hear constantly in ECS 150
            5. Incur a performance penalty every time we flush cache
                a. Lose all our built-up cache data
                b. Must go back to RAM to fetch new values

2. Cache design
    a. Multiple-level caches
        i. See these in many modern CPUs
        ii. L1 cache is the smallest and closest to the CPU
            1. Usually logically addressed
        iii. L2 cache is larger than the L1, but is further away from the CPU
            1. Takes more time to access the L2 than the L1
            2. Can be either physically or logically addressed
        iv. Same applies for L3
            1. Even larger than L2, but even slower
            2. Usually physically addressed
        v. Will even see L4 caches on occasion
            1. Much rarer than the other levels, though
    b. Unified or split
        i. Split – have separate caches for data or instructions
        ii. Unified – combine the two
    c. Cache size
        i. Larger cache means slower cache
            1. Spend more time checking addresses
            2. Worst case paths are worse due to larger cache area
        ii. There is no optimum cache size
            1. Exact specifications will always depend on machine being run
    d. Many other cache specifications
        i. Will talk about write and replacement policies in more detail later
        ii. Inclusive versus exclusive is another important design parameter for multiple-level caches
            1. Will not go into this one in this course

3. Cache address layout
    a. Going to use slightly different terminology than the book
        i. Same idea, but I find the book's variable names less than intuitive
    b. Terminology
        i. *C* = cache size
            1. Almost always given in terms of KB or MB
            2. KB = 2^{10}, MB = 2^{20}, GB = 2^{30}
                a. Following typical computer science convention
                    i. Kilo-, mega-, giga-, etc. are powers of 2
                b. Will probably want to memorize these
                    i. Will be using powers of 2 a lot for these problems
        ii. *LS* = line size
            1. Size of the lines of cache we discussed earlier
            2. Typical line size ranges from 4 to 64 bytes, but could be larger
        iii. *LIC* = lines in cache
            1. The total number of lines we have
            2. LIC = C / LS
        iv. *S* = sets
            1. Cache lines divided into groups called sets
            2. Exactly how many sets depends on the cache mapping
                a. Will discuss this further in a bit
        v. *W* = ways
            1. The number of places that a line of cache could potentially go in each set
            2. Will discuss this further in a bit
            3. S * W = LIC
    c. Address layout (in bits)

Tag	**Set**	**Offset**		**Address Bits**
Address bits – set bits – offset bits	$\log_2 S$	$\log_2 LS$	=	Usually given

        i. Split up an entire physical address according to the above
        ii. Set and offset bits determine where exactly a block of memory from RAM goes in cache
        iii. Tag uniquely identifies memory addresses that map to same set
            1. Tag bits are always "whatever's left"
    d. Problems of this type
        i. Usually given some of the variables above
        ii. Asked to calculate the rest

4. Direct mapped (DM) caches
    a. How does an address from main memory map to the cache?
    b. Direct mapped (DM)
        i. Each block of main memory gets directly mapped to a single cache line
        ii. W = 1, S = LIC
            1. Only one line in each set
            2. Number of sets is equal to number of lines
                a. S = LIC / 1 = LIC
        iii. Easy and cheap to implement
        iv. What happens if different memory accesses map to same line of cache?
            1. Line of cache keeps getting evicted for the other one
            2. Performance penalty since we must keep going to main memory for line we just evicted
        v. Example
            1. 8-byte DM cache with line size of 2, and 4-bit address
            2. LIC = C / LS = 8 / 2 = 4
            3. For a DM cache, S = LIC = 4
            4. Address format below

Tag	Set	Offset		Address Bits
4 – 2 – 1 = 1 bit	$\log_2 S = \log_2 4 = 2$ bits	$\log_2 LS = \log_2 2 = 1$ bit	=	4 bits

5. DM cache mapping example *(not covered in lecture)*
    a. Use the same cache specifications and address format we calculated in the previous problem
    b. Memory values for addresses below

Address	Data	Address	Data
0110	0x1B	0111	0x59
1000	0xFE	1001	0x3D
1010	0x25	1011	0x98
1110	0x0C	1111	0x3A

    c. How do we map the addresses above to the cache?
        i. Use the address layout above to split the addresses into each portion
        ii. Use those portions below to fill in the cache in the next example
        iii. Next example explains exactly why a specific access is a hit or a miss

Memory Access	Tag	Set	Offset	Access Type	Hit or Miss
0110	0	11	0	Read	Miss
0111	0	11	1	Read	Hit
1000	1	00	0	Read	Miss
1001	1	00	1	Read	Hit
1110	1	11	0	Read	Miss
1111	1	11	1	Read	Hit
1010	1	01	0	Read	Miss
1011	1	01	1	Read	Hit

d. Use the memory access table from the previous problem to fill in the cache below
    i. Make accesses in order that the table specifies
e. First two accesses
    i. 0110 reads from the cache
        1. Valid bit for that line was 0 to start
            a. Line in cache isn't valid, so we have a miss
        2. We grab the line from memory
            a. When making accesses to the cache, need to pull entire line
            b. We also grab 0111 from memory, because line size is two
        3. Read from cache, CPU pulls value 0x1B from the cache
            a. Set valid bit now that this line is valid
            b. Dirty bit remains 0
                i. Line can't be dirty if we just pulled it from the cache
                ii. Will only need to change dirty bit upon writes
    ii. 0111 reads from the cache
        1. Compare tags, tag for 0110 matches tag currently in set 11
            a. Line in cache is valid, so we have a hit
        2. Since tags match, CPU pulls value 0x59 from the cache
            a. No change to valid or dirty bits

Set Number	Line Number	Tag	Byte 0	Byte 1	Valid	Dirty
00	0				0	0
01	1				0	0
10	2				0	0
11	3	0	1B	59	~~0~~ 1	~~0~~ 0

f. Next two accesses
    i. Same idea as previous two
    ii. 1000 reads from the cache
        1. Line in cache isn't valid, so we have a miss
        2. We grab the line from memory
            a. We also grab 1001 from memory
        3. Read from cache, CPU pulls value 0xFE from the cache
            a. Set valid and dirty bits appropriately
    iii. 1001 reads from the cache
        1. Compare tags, tag for 1001 matches tag currently in set 00
        2. Since tags match, CPU pulls value 0x3D from the cache
            a. No change to valid or dirty bits

Set Number	Line Number	Tag	Byte 0	Byte 1	Valid	Dirty
00	0	1	FE	3D	~~0~~ 1	~~0~~ 0
01	1				0	0
10	2				0	0
11	3	0	1B	59	~~0~~ 1	~~0~~ 0

g. Next two accesses
   i. 1110 reads from the cache
      1. There is a valid line in the cache, but the tags don't match!
      2. Need to evict the line currently in cache and place new line inside
         a. Dirty bit is 0 so don't need to worry about writing new values back to memory
      3. We grab the line from memory
         a. We also grab 1111 from memory
         b. Evict line currently in cache, replace with new line and new tag
      4. Read from cache, CPU pulls value 0x0C from the cache
         a. Set valid and dirty bits appropriately
   ii. 1111 reads from the cache
      1. Compare tags, tag for 1111 matches tag currently in set 11
      2. Since tags match, CPU pulls value 0x3A from the cache
         a. No change to valid or dirty bits

Set Number	Line Number	Tag	Byte 0	Byte 1	Valid	Dirty
00	0	1	FE	3D	~~0~~ 1	~~0~~ 0
01	1				0	0
10	2				0	0
11	3	~~0~~ 1	~~1B~~ 0C	~~59~~ 3A	~~0~~ ~~1~~ 1	~~0~~ ~~0~~ 0

h. Final two accesses
   i. 1010 reads from the cache
      1. Line in cache isn't valid, so we have a miss
      2. We grab the line from memory
         a. We also grab 1011 from memory
      3. Read from cache, CPU pulls value 0x25 from the cache
         a. Set valid and dirty bits appropriately
   ii. 1011 reads from the cache
      1. Compare tags, tag for 1011 matches tag currently in set 01
      2. Since tags match, CPU pulls value 0x98 from the cache
         a. No change to valid or dirty bits

Set Number	Line Number	Tag	Byte 0	Byte 1	Valid	Dirty
00	0	1	FE	3D	~~0~~ 1	~~0~~ 0
01	1	1	25	98	~~0~~ 1	~~0~~ 0
10	2				0	0
11	3	~~0~~ 1	~~1B~~ 0C	~~59~~ 3A	~~0~~ ~~1~~ 1	~~0~~ ~~0~~ 0