CS61C Notes

Leo Villani

May 2024

1 Pipeline

Iron Law of Processor Performance $\frac{time}{program} = \frac{instructions}{program} * \frac{cycles}{instructions} * \frac{time}{cycle}$

$\frac{instructions}{program}$ up to the algorithm really $O(n^2) vs O(n)$
$\frac{cycles}{instructions}$ - for single cycle datapath only 1 but then can pipeline
$\frac{time}{cycle}$ - min clock period which is $\frac{1}{clockfreq}$
Processor throughput represents the $\frac{instructions}{time}$ which is just the inverse of $\frac{cycles}{instructions} * \frac{time}{cycle}$
Thus for the RISC-V datapath we can just pipeline each stage via pipeline registers.

1.1 Structural Hazards

Definition 1.1 *A hazard is a situation in which a planned instruction cannot execute in the proper clock cycle.*

1. Structural Hazard: Hardware does not support access across multiple instructions in the same cycle
2. Data Hazard: Instruction have data dependency, need to wait for previous instruction to complete its data read/write
3. Control Hazard: Flow of execution depends on previous instruction

Our RISC-V datapath avoids structural hazards via better hardware, i.e. Reg File block simultaneously supports two reads, one write.
If the same register is written and read in one cycle we may have a data hazard. We need to write then read. But if there are multiple instructions that need to change the value in a row, add and sub, then we just stall using NOPs.
Another solution to data hazards is forwarding which instead of waiting for regfile to update you can just use the ALU result for the next instruction as that will be whats stored in the regfile.

Control Hazards occur when the instruction fetched may not be the one needed. For example if the beq branch is taken. We can just flush the pipeline by converting incorrect instructions to NOP. This is only a problem when you actually take a branch.

2 OS

A program is a sequence of instructions to run (such as an executable)
A process is the actual execution of a program. Each process is a largely separate entity, with its own memory space.
Each process is composed of threads, which are independently running instruction sequences that share most memory.
So far, we've discussed programs that use only 1 thread, on 1 process. However, a single program may spawn multiple processes, each of which use multiple threads.
The datapath we've discussed so far is a CPU core. A single core can run one thread at any given time.
The CPU is composed of multiple cores, which thus allows multiple threads to be run simultaneously.

3 Parallelism

3.1 Amdahls Law

AKA the bane of performance programming
"The maximum speedup we can attain with our code is limited by the fraction that cannot be sped up"
If we speed up 95 percent of our code, we can get at most a 20x speedup
Almost any optimization we discuss from now on will only affect one portion of your code. If you only focus on one part, you'll eventually be unable to continue optimizing.
Formal equation: Speedup $= \frac{1}{(1-F)+\frac{F}{S}}$, where F is percent of code that you speed up and S is how many times faster you make that part.
That's annoying to work with; much easier to treat these problems as word problems instead

3.2 Operation Time Costs

1. File operations: Extremely slow (retrieving from disk, so it takes a long time)
   Also includes prints and other I/O, so it's a good idea to avoid printing lots of data when testing runtime.

2. Memory Operations: 100x faster than file operations (accessing RAM)
   Can be improved through caching, but memory operations still take 3-100 clock cycles

3. Branches and Jumps: Slower than most operations due to hazards (potentially 5 clock cycles)
   Can be improved through loop unrolling

4. Arithmetic Operations: Fastest operations (1 cycle ideally sometimes more)

Definition 3.1 (Function Inlining) *Replace a function call with the body of that function*

Definition 3.2 (Loop Unrolling) *Increase the number of steps done per iteration of a loop*

Definition 3.3 (Variable Caching) *Save commonly used values in variables, instead of forcing recomputations, additionally save commonly used variables in registers (i.e. register int i = 0)*

Definition 3.4 (Data Caching) *Save commonly used data "closer" to the CPU, so we can access it in fewer cycles (10 cycles), note accessing adjacent memory addresses will be on average faster than accessing nonadjacent memory addresses (i.e. transpose the matrix before multiplying)*

3.3 SIMD (DLP, data level parallelism)

Instead of doing math on one number at a time, we can instead do math on several numbers at a timed in a single clock cycle
We use SIMD instructions which use specialized vector registers, the benefit is not really just doing 4 times less additions but doing less load/stores at a time.

Applying DLP to Matrix Multiply

```
__m128 v3 = _mm128_set1_ps(0);
int i;
for(i = 0; i < arrlen/4*4;i+=4) {
    __m128 v1 = _mm128_load_ps(arr+i);
    __m128 v2 = _mm128_load_ps(arrtwo+i);
    v3 = _mm256_fmadd_ps(v1, v2, v3);
}
float mem[4] __attribute__ ((aligned (32)));
_mm128_store(mem, v3);
for(;i<arrlen;i++) {
    mem[0]+=arr[i]*arrtwo[i];
}
return mem[0]+mem[1]+mem[2]+mem[3];
```

3.4 TLP (thread level parallelism

3.4.1 MultiThreading vs ¡ Multiprocess Code

Threads: Different instruction sequences on the same process
Threads on the same process share memory
"Easy" to communicate
Limited to a set of cores wired to the same memory block (1 node)

Processes: Largely independent from each other
Different processes can't easily share memory
"Difficult" and time consuming to communicate
Can expand to as many cores as you have available, over as many nodes as you want

3.4.2 Multithreading Framework Overview

Each thread has its own registers
Each thread has its own PC
Each thread has its own stack
Each thread shares the same heap
Communication is done through shared memory
Threads run simultaneously, so we have no control over the order in which threads do their work

3.4.3 Open MP

OpenMP is an extension of C used for multithreaded code (shared memory so no multi-node computation)
Each thread has its own stack for private variables, but otherwise shares memory with other threads
In order to create a parallel section:
pragma omp parallel

//parallel code

In a parallel segment

1. $\mathbf{omp}_g et_n um_t hreads()returnsthenumberofthreadsrunning$

3.4.4 Shared vs Private Variables

By default, any variable declared outside the parallel segment is shared: all threads write/read from the same variable
Any variable declared inside the parallel segment is private: Each thread has

its own version of the variable.
Can overrule this with the private and shared keywords:
pragma omp parallel private(var) shared(var2)
Note that heap memory is always shared, though we can have private pointers and private mallocs (if we do the malloc in the parallel segment, and free them within the parallel segment)

3.4.5 Data Races

Recall the OS can choose whichever threads it wants to run, and change threads at any time
This is one of the biggest downsides to multithreading: A multithreaded program is no longer deterministic, and will have a random execution order every time we run the program.
Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

3.4.6 Locks

A lock is an object which helps with synchronization.
Essentially, each thread can try to "acquire" a lock, but only one thread can have the lock at a given time.
Code surrounded by a lock is called a "critical section", because only one thread is allowed to run that section at a time.
This leads to critical sections:
OpenMP gives you some commands that let you use critical segments completely safely
#pragma omp barrier
Forces all threads to wait until all threads have hit the barrier
#pragma omp critical
Creates a critical segment in parallel code; only one thread can run a critical segment at a time.

3.5 PLP (process level parallelism)

3.5.1 Multithreading Framework Overview

While a multithreaded program is fundamentally one program, individual processes are essentially distinct program instances entirely
Different processes don't share memory, but generally can share the same file system
In a multithreaded program, the entire thing crashes if any single thread crashes.
In a multiprocess program, each process runs independently, so if one process crashes/terminates, the others still keep going.

Because you don't share memory, you can't use locks or concurrency primitives

3.5.2 Open MPI

2. int MPI_Init(int* argc), char*** argv)
   Initializes the MPI framework, connects everything together, etc.
   Should be done at the start of an MPI program (Technically a few things are allowed before MPI_Init, but best practice is to do this first.)
   Send in the addresses to argc and argv, though for Open MPI, it doesn't actually use them; this is for compatibility with other MPI systems

2. int MPI,Finalize()
   Finalizes the program, cleaning up the MPI framework
   Should be the lat thing done in an MPI program. Must be done by all processes before termination

3. MPI_Comm_size(MPI,Comm comm, int *size)

4. MPI_Comm_rank(MPI_Comm comm, int *rank)
   returns the number of MPI nodes in the group and process ID, respectively
   the first argument can be used if you split up your processes into groups
   Can alxways use MPI_COMM_WORLD to get the size of the entire program/process ID relative to all processes

Note that all of these functions receive as input a pointer which will be used to store the return value. The actual return value is used to specify if the operation worked (0 if success, an error code if failure), so don't conflate the two!

3.5.3 The Manager Worker Framework

Very common framework for MPI programs, fairly simple to implement while being versatile enough to adapt to new processes.
Assumes that the problem youre solving can be reduced to a set of independent tasks that can be done in any order, indepenedent of each other
Main Idea: have two roles:
-manager, whose job is to assign work and inform the user of progress
-worker, who receives work from the manager and does the work
This involves writing two versions of the code, one for process 0, and for all other processes

Performance Programming Overview

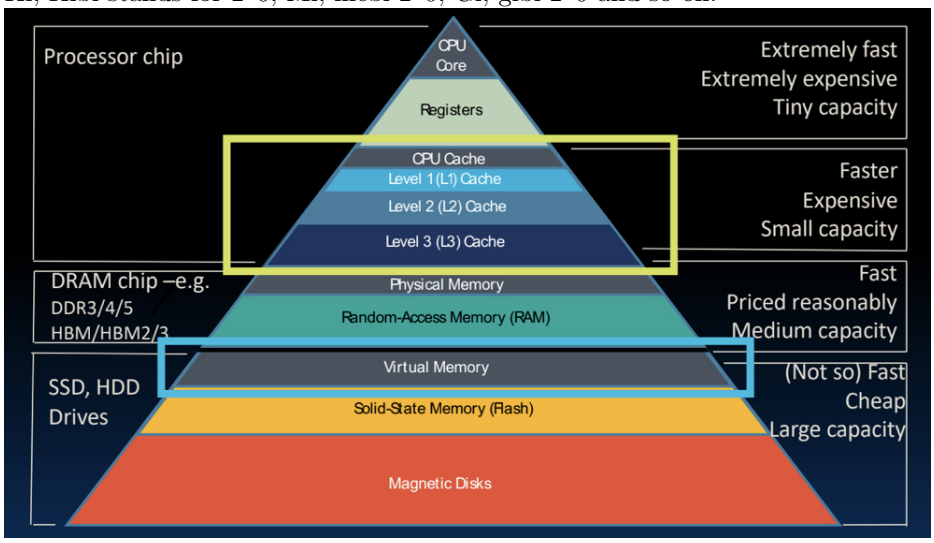
CS 61C Spring 2024

Optimization	Max Speedup	Pros	Cons
Register/Function Inlining	<2x	Easy change, reduces memory accesses	Minimal effect, optimizing compiler might do this already
Loop Unrolling	<2x	Reduces Branching	Minimal effect, significant penalty to maintainability
Cache Optimizations	~10x	Surprisingly good	Often requires algorithmic changes
SIMD	~8x	Fairly applicable, minimal overhead	Limited by hardware, often hit hard by Amdahl's Law
Multithreading/OpenMP	#cores/node	More flexible than SIMD and MPI, generally	Concurrency issues, high overhead
Multiprocess/Open MPI	#cores	Can be extended arbitrarily large	Expensive communication, high overhead

25

4 Caches

Main Memory is DRAM.
Ki, Kibi stands for 2^10, Mi, mebi 2^20, Gi, gibi 2^30 and so on.



As we get closer to the processor, hardware is smaller, faster and more expensive. Also data is smaller and a subset of lower levels.
A cache contains copies of data that are being used. A cache works on the principles of temporal and spatial locality.

Definition 4.1 (Temporal Locality) *If we use it now chances are that well*

want to use it again soon. So keep most recently accessed data items closer to the processor.

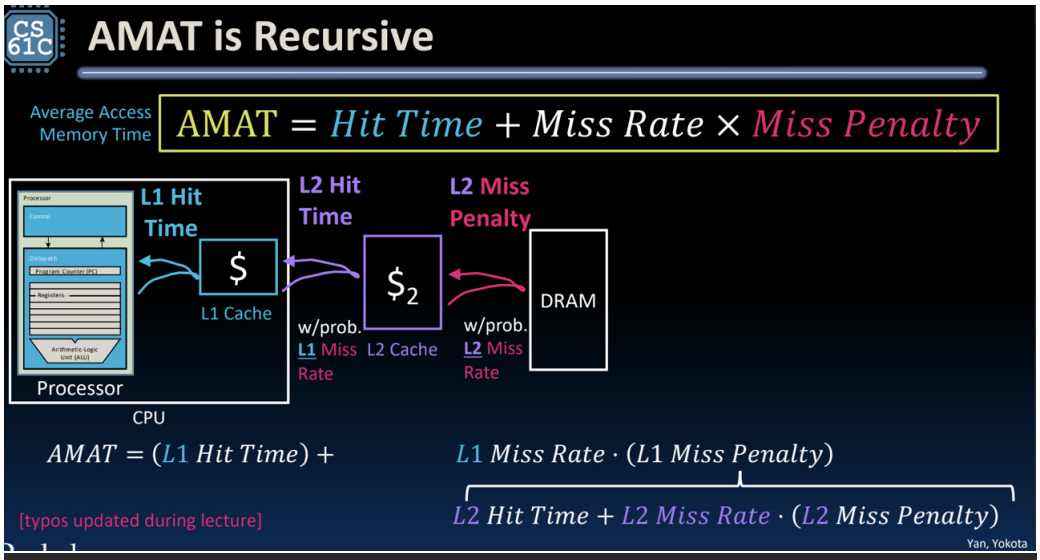
Definition 4.2 (Spatial Locality) *If we use a piece of memory chances are we'll use the neighboring pieces soon. So move blocks consisting of continguous words closer to the processor.*

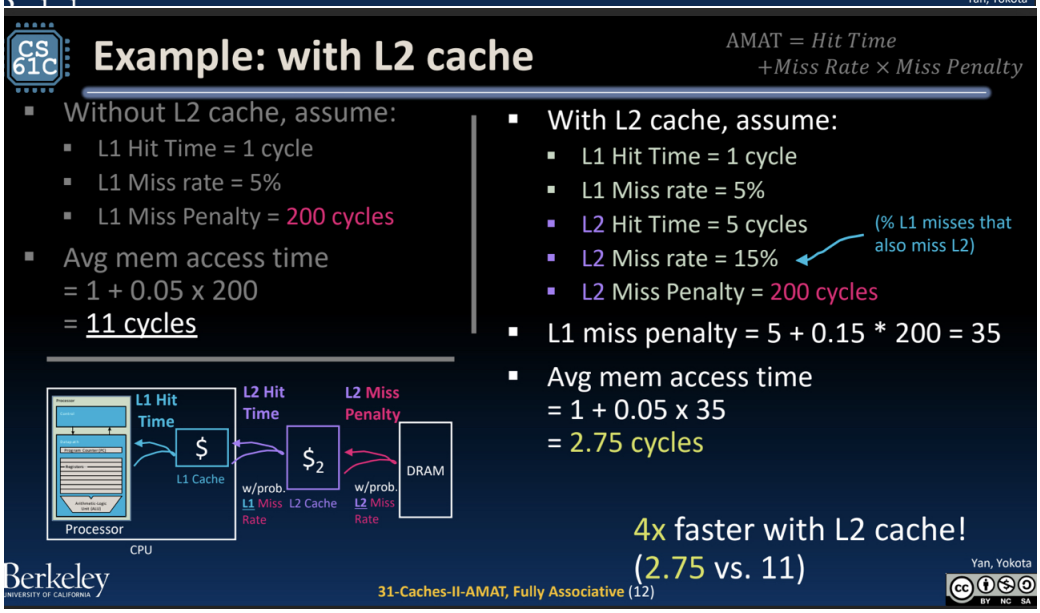
4.1 Memory Access with a Cache

1. Processor issues address X to the cache
2. Cache checks for copy of data at address X
3. If hit (finds match): cache reads Y and send Y to the processor. If miss (no match) cache send X to memory. Then memory reads Y at X and sends block with Y to cache and the cache replaces some block to store the block containing Y. Then the cache sends Y to the processor.
4. processor loads Y into register Z

4.2 AMAT: average memory access time

First note the following definitions:
-Hit rate: fraction of access that hit in the cache
-Hit time: time (latency) to access cache memory (including tag comparison)
-Miss Rate: 1 - Hit rate
-Miss penalty: time(latency) to replace a block from lower level in memory hierarchy to cache
Example:
Suppose you have a cahce with hit time of 1 cycle, miss rate of 5 percent and miss penalty of 20 cycles.
What is the average memory access time (AMAT)?
$AMAT = HitRate * (HitTime) + MissRate * (HitTime + MissPenalty) = HitTime + MissRate * (MissPenalty) = 1 + 0.5 * 20 = 2$
Note: AMAT is recursive so for a two layer cache when finding miss penalty you have a subproblem (look at the image below)





4.3 Fully Associative Caches

Placement Policy: the data can be stored anywhere in the cache
For a given memory address, split into two fields the tag and the offset.
To check if a block has our data we just check the tag
Fully associative means that any blovck can potentially match our tag so we need to check all tags for the correct block
We also need a valid bit so that when the cache is warming up it knows if the

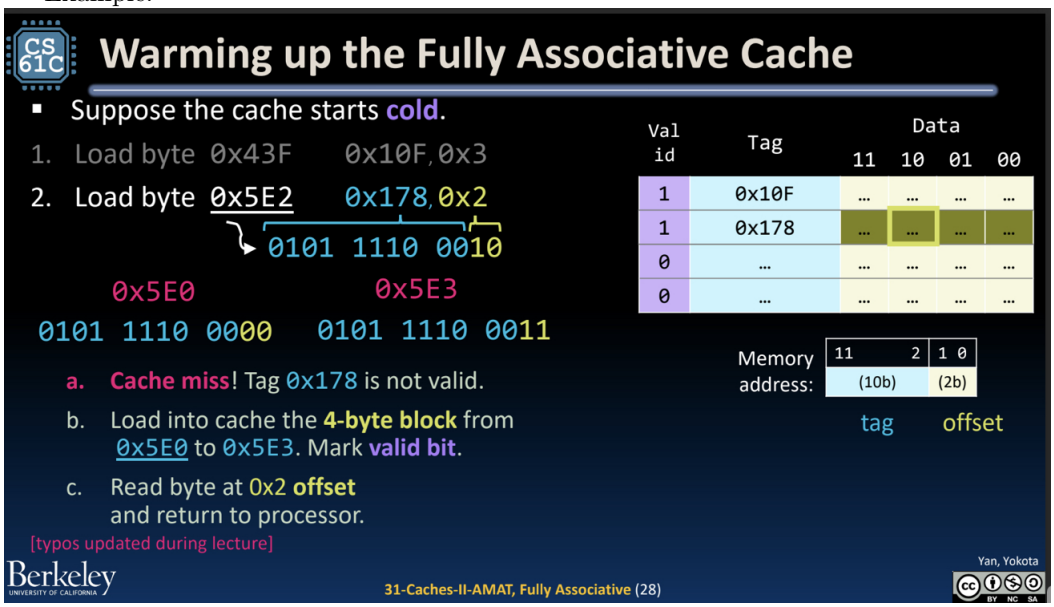
entry in the cache is valid or not

4.3.1 Cache Temperatures

The cache can be one of four things:

1. Cold: Cache is empty
2. Warming: Cache filling with values you'll hopefully be accessing again soon
3. Warm: Cache is doing its job, fair % of hits
4. Hot: Cache is doing very well, high % of hits

Example:



Note the following definitions:
-Cache block/line: a single entry in the cache
-Block size/line size: #bytes per cache block
-Capacity: Total #data bytes that can be stored in a cache
-Tag: Identifies data stored at a given cache block
-Valid bit: indicate if data stored at a cache block is valid
Block size is just amount of data in block so for 2 offset bits block size is $2^2 = 4$
Capcacity is just block size times number of rows in the cache
Offset size can thus be computed as $log_2(blocksize)$
Tag bits are then $\#addressbits - \#offsetbits$

4.4 Block Replacement Policies

When the FA (Fully associative) cache reaches capacity we can still miss. Therefore we need a policy for block replacement!
LRU: replace the entry that has not been used for the longest time
The con is that this is hard to implement with an LRU bit that says how long its been since its been used. (higher number, used further in past)
We can define other block replacement policies analagously:
MRU, FIFO (queue), LIFO (stack), Random
Note: FIFO and LIFO are good approximations for LRU and MRU

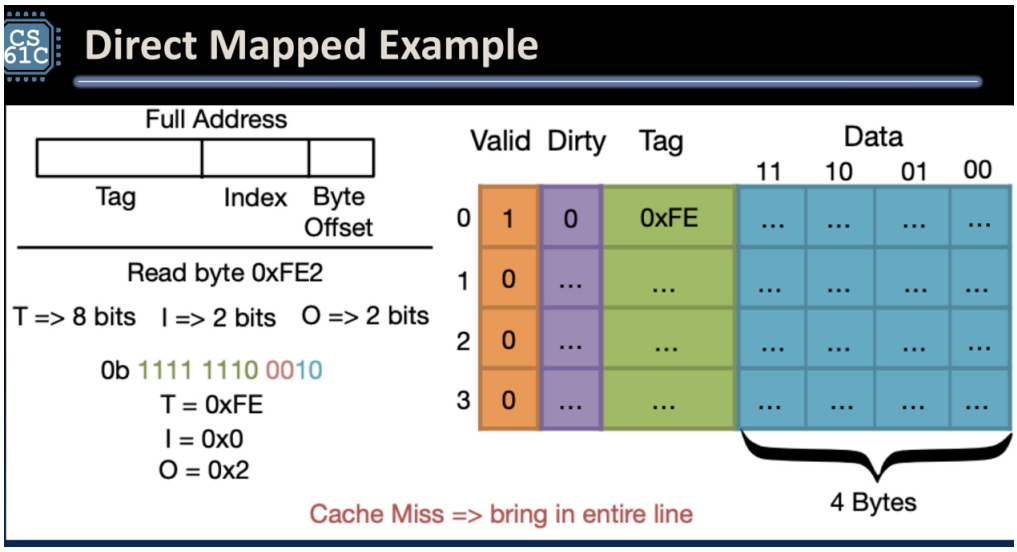
4.5 Write Policies

4.5.1 Write Through vs Write Back

-store instructions write to memory which changes values
-hardware needs to ensure that cache and memory have consistent information
-**write-through**:
write to cache and memory at the same time
-**write-back** (requires additional dirty bit field):
write data in cache and set a dirty bit to 1
when this block gets evicted from the cache (and "back" to memory), write to memory

4.6 Direct Mapped Cache

Put a new block in one specific place
Placement policy: Each memory address is associated with exactly one possible block in the cache. Thus to check for existence in the cache we only need to look in a single location in the cache.
In direct mapped caches split address to also get index of cache block to directly map to.
Full address = [tag + index + Byte Offset]
Example:



Benefit of simpler hardware than a fully associative cache
Note: In direct mapped caches there is only ever one block to evict-the existing block with matching index so no block replacement policies from FA.
Still can use both write policies.

4.7 Types of Misses

1. Compulsory Miss: Caused by the first access to a block that has never been in the cache
2. Capacity Miss: caused when the cache cannot contain all the blocks needed during the execution of a program. Occur when blocks were in the cache, replaced, and later retrieved.
3. Conflict Miss: Multiple blocks compete for the same location in the cache, even when the cache has not reached full capacity

Conflict misses occur in direct mapped caches and set associative caches, but would not occur in a fully associative cache.

4.8 Summary

Fully Associative Caches:
- A specific line of data can be stored in any line of the cache
- No index
- Need to choose block replacement policy
- Need to choose write policy

- Hardware: expensive

Direct Mapped Caches:
- A specific line of data can only be stored in one index of the cache
- Has index
- If the line you want to store the data in is occupied, you kick out that line
- Need to choose write policy
- Hardware: cheap

Set Associative Caches:
- A specific line of data can be stored at only one index of the cache but multiple lines can be in each index
- Has index
- Need to choose replacement policy
- Need to choose write policy

Note: this method mostly balance good parts of FA (i.e. better performance by reducing conflict misses) with good parts of DM (simpler hardware)

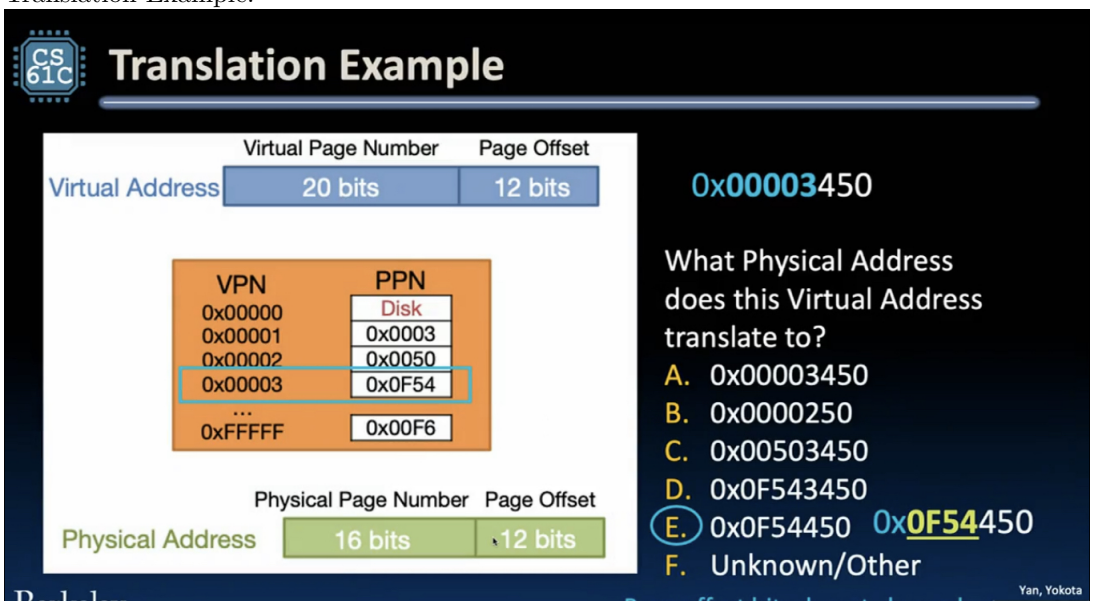
Misses simplified for this class:
If a miss occurs:
- block never evicted before? then compulsory
- cache currently full (no empty spaces)? then capacity
- otherwise? conflict

5 VM

What if main memory is smaller than the program address space?
RV32I provides a 32-bit addresss space, or $2^32 = 4GiB$ but RAM is $1GiB$, which is just 2^30 bytes of addressable memory.
Virtual memory gives each process the illusion of a full memory address space that it has completely for itself.
- Virtual Address Space: set of addresses that the user program knows about
- Physical address space: set of addresses that map to actual physical locations in memory, hidden from user applications

OS handles the virtual address and physical address
Physical memory is broken into pages. A disk access loads an entire page into memory. So if each page is $4KiB = 2^12$ we would need 12 bits of page offset.
Page table translates virtual addresses to physical addresses.
Extract VPN, top x bits, where x is number of bits in virtual address space - offset bits. Then we find the PPN in the page table and attach on the offset.
If page number isnt on page table for the VPN we get a page fault, then the OS asks to load in the page into memory so we update the page table and then find the actual physical address and return the information in the page.
Example:

32-bit virtual address space, 4-KiB pages
Then 2^32 virtual addresses / 2^12 bytes per page os 2^20 different VPN
Note there is one page table per process, each process gets a dedicated page table. This lead to isolation between the processes.
A page table is not a cache, as it has no data in it, that data is still in memory.
Translation Example:



Note: Byte addressable memory even though we are using bits.
Number of Bits Example:
32-bit machine with $8GiB$ of RAM and $16KiB$ pages.
Page offset bits = $log_2(16KiB) = log_2(2^14) = 14$
VPN bits = $32 - 14 = 18$
PPN = $log_2(8GiB) - 14 = log_2(2^33) - 14 = 19$
Note: the virtual page size and the physical page size are the same
The number of page table entries = the number of virtual page numbers
Page table size = size of each page entry (PPN + status bits) times number of page table entries
Page tables are stored in memory.
Thus assuming no caches each load or store requires two memory accesses, page table access and then the memory access.

5.1 Page Faults

Page table entries store status to indicate if the page is in memory or only on disk.
Valid implies on DRAM, and not valid means on disk which is what would trigger a page fault.

Note: this then begs the question what should virtual memory's write policy be We use write-back as disk accesses take way too long.
Thus the status bits for a page table are a valid bit, a dirty bit for the write back policy and a write protection bit (if page is program code...).

5.2 OS: Trap Handler