

CS 33: Introduction to Computer Organization

Week 7

Admin

- Lab 3: Smashing lab “due” tonight, November 13th
- Midterm II: The Quickening, November 16th

Agenda

- Midterm discussion
- Instruction Level Parallelism (ILP)
- Out-of-Order Processing + Optimization
- Caching
- Task Level Parallelism: Processes/Threads

Midterm II: The Quickening

- What should you prepare for?

Midterm II: The Quickening

- What should you prepare for?
 - Everything

Midterm II: The Quickening

- What should you prepare for?
 - The worst

Midterm II: The Quickening

- What should you prepare for?
 - The worst
- General pieces of advice:
 - Heavily emphasize what the professor has covered in class.
 - Be careful with how much value you attribute to the book exercises.
 - For one thing, the professor thinks book questions are either too easy, too boring, or too long.

Midterm II: The Quickening

- General pieces of advice:
 - He wants difficult questions that are short to answer (if you know the answer). Elegant, but non-obvious.
 - Approximately $\frac{3}{4}$ of midterm II will be post-midterm I material.
 - Also, if you're called upon to write code, don't labor over: semi-colons, proper includes, etc.

Midterm II: The Reckoning

- What was post-midterm I material?
- Buffer overflow
- Floating point
- Optimization
- Instruction Level Parallelism
- Memory Hierarchy/Caching
- Task/Thread-level parallelism
- Homeworks 3/4 + lab 2/3

Instruction Level Parallelism

- The optimizations presented in chapter 5 are all serial optimizations that can marginally improve execution time.
- Marginal improvement can still be significant depending on how much time is spent on the section that was improved.

Instruction Level Parallelism

- However, anything a traditional program does is limited by the fact that a single program or a single thread is expected to do the entire work of a program.
- If we have to iterate over a length 100 array and assign a value to each element, there's not a whole lot a single thread can do to get around this.
- But what if we have two processors dedicated to the same task?

Instruction Level Parallelism

- If instead, we have two processors each working on half of the data, we can finish the array assignment in half of the time, which provides a speedup of 2, which was already faster than the best loop unrolling case.
- Plus, the performance improvement provided by splitting the task among independent agents doesn't degrade nearly as quickly. If we had 100 processors, we could complete the task in the time it takes a processor to execute a single instruction.

Instruction Level Parallelism

- It's this observation that has driven the industry towards parallelism rather than simply increasing clock speed when it comes to improving performance.
- Why can't we just keep increasing the clock speed?

Instruction Level Parallelism

- Power Wall:
 - As the clock rate increases, the power needed to operate everything increases. In addition to requiring a lot of power, much of it is leaked/dissipated as heat.
- Memory Wall:
 - Recall from class that processor speeds have increased significantly since the 80's.
 - Also recall the RAM access time has remained essentially the same.

Instruction Level Parallelism

- Memory Wall:
 - It doesn't help to reduce the processor's speed from 1 ns per instruction to .5 ns per instruction if accessing RAM takes 200 ns and you have a reasonable proportion of memory accesses.
 - The bottleneck is still in RAM access and via Amdahl's Law, the bottleneck is the thing we'd need to reduce in order to get a significant improvement.

Instruction Level Parallelism

- Parallelism is such a powerful method of achieving speed-up that we parallelize everything:
 - Multiple computers running in parallel to form server clusters
 - Multiple processors running in parallel to run different programs simultaneously
 - Multiple threads/processes running in parallel.
 - Multiple instructions within the same thread being executed in parallel.

Instruction Level Parallelism

- Instruction Level Parallelism (or ILP) is that last bullet point.
- To help understand the ways that this is done, it's easier to think in terms of the nebulous “micro-operations”.
- But what are they?

Instruction Level Parallelism

- Recall that the x86 ISA follows the CISC school of thought (Complex Instruction Set Computing).
- This means that the x86 ISA defines many different complicated instructions and is thus a “complex instruction set”.
- Hence: all of the different addressing modes, a million different conditional operations, billions of different operand combinations for “muls”
- Why is something like this a good thing?

Instruction Level Parallelism

- When each instruction is powerful and can do a lot of things, this means that code is more compact. This was useful back in the day when memory constraints were a real problem.
- Each instruction is also closer to the high level languages that we, the users, know and love.
- Ultimately, the ISA defines many instructions so that we only need to execute a few instructions (a “few”).
- Why might CISC be undesirable?

Instruction Level Parallelism

- Having instructions that are more complicated comes at the cost of being slower to execute.
- As the machine is expected to execute more and more complicated instructions, the complexity of the architecture design goes up.
- Having extremely complicated instructions also made it difficult to implement ILP.
- As a result, everything started moving towards RISC or Reduced Instruction Set Computing.

Instruction Level Parallelism

- But what about poor x86?
- ...and for that matter, if CISC isn't used any more, why is x86 so prominent?

Instruction Level Parallelism

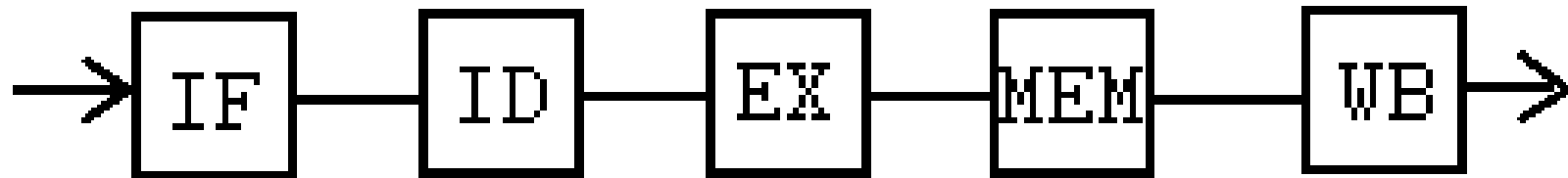
- Mostly historical. After everything x86 has done for us, it'd be cruel to leave it out in the cold.
- But then, how can we do ILP with the CISC x86?
- Secretly have another compilation stage, one that compiles normal x86 into the micro operations
- Essentially, it compiles CISC x86 into RISC-like x86 micro-operations.

Instruction Level Parallelism

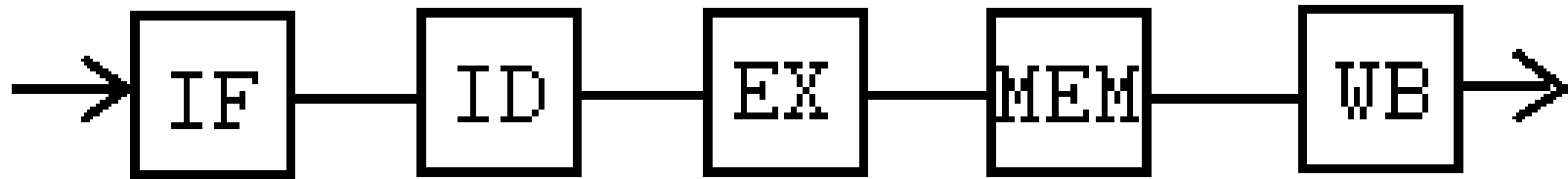
- RISC-like instructions are simple and generally follow a few principles:
 - Operation come in a few basic forms. None of that suffix nonsense.
 - In order to load from memory, we have to directly load it with a load operation.
 - We can't directly add to or from a memory address.

Instruction Level Parallelism

- In order to execute a single simple micro operation, it goes through several steps or “stages” to complete. In the common academic example, there are 5 stages.
- This is known as the pipeline.

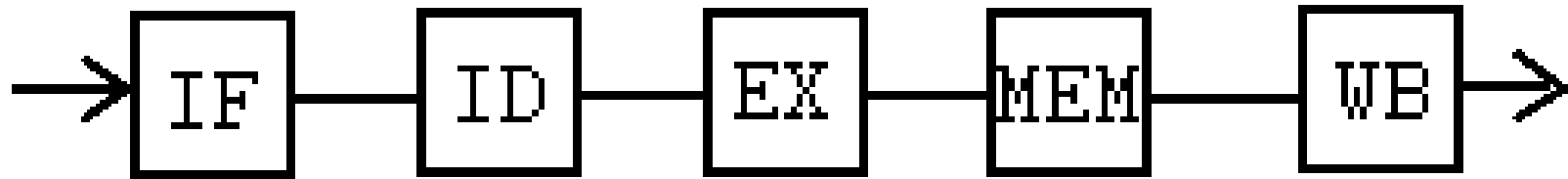


Instruction Level Parallelism



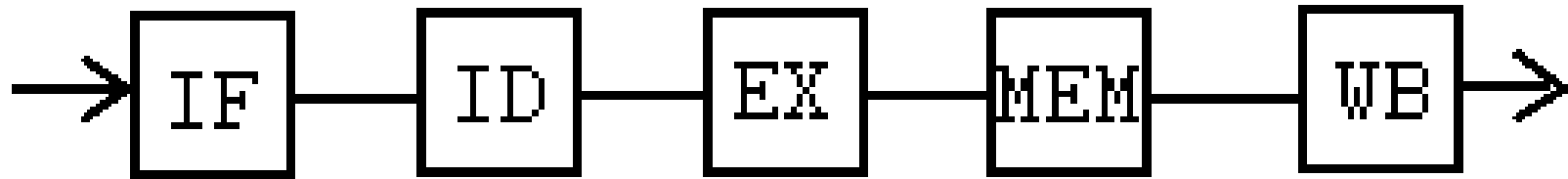
- Each stage corresponds to a physical component in the processor.
- IF (Instruction Fetch): Takes as input the address of the instruction (%rip). Then it finds the actual instruction in memory.
- ID (Instruction Decode/Register Read): Given the instruction bytes, get the appropriate values from memory.

Instruction Level Parallelism



- EX (Execution): Execute any arithmetic operation that the instruction needs to do.
- MEM (Memory): Read/write from memory if necessary.
- WB (Write back): Write to register if necessary.

Instruction Level Parallelism

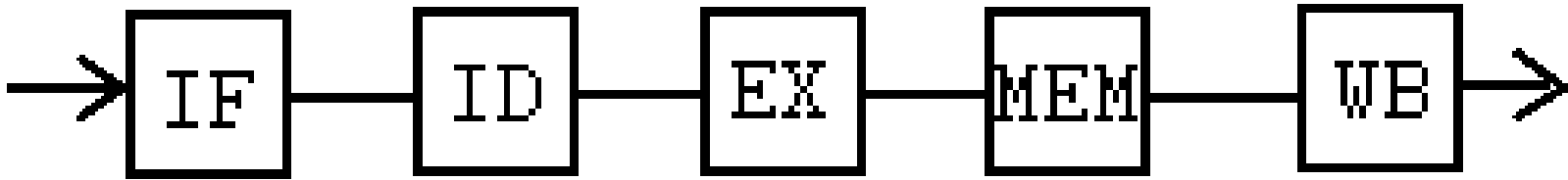


- Assume that each block takes a single cycle to execute. A single instruction would take 5 cycles to complete.
- The latency of each instruction is 5 cycles.
- Our throughput of this system is 1 instruction per every 5 cycles.

Instruction Level Parallelism

- Consider the execution of: `add %eax, %ebx`
- Step 1. The instruction address is used to fetch the instruction
- Step 2. The values from `%eax` and `%ebx` are loaded
- Step 3. `%eax` is added to `%ebx`.
- Step 4. The `add` instruction doesn't use memory
- Step 5. Save the value back into `%ebx`

Instruction Level Parallelism



- At any given point in time, we are only using one of five available components.
- This is an opportunity to execute multiple instructions at the same time.

Instruction Level Parallelism

- Say we have the following instruction sequence:
 - 1. add %eax, %ebx
 - 2. sub %ecx, %edx
 - 3. xor %esi, %edi
- Say the execution of this snippet begins at time 0.
- At time 0: the add would be in the IF stage.
- At time 1: the add would be in the ID stage and the sub would be in the IF stage.
- At time 2: the add would be in the EX stage...

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?
 - It still takes five cycles for any given instruction to complete. The latency has not changed.
- However, how many instructions are completed per cycle?

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?
 - It still takes five cycles for any given instruction to complete. The latency has not changed.
- However, how many instructions are completed per cycle?
 - 1 instruction is completed per 1 cycle rather than 5 cycles. The throughput has increased from $1/5$ instructions per cycle to 1 instruction per cycle. THAT is a huge gain.

Instruction Level Parallelism

- This example of ILP can simultaneously execute 5 instructions at any given time.
- ...or is this perhaps a little too idealized?
- What are some cases where we wouldn't be able to achieve this optimal case?

Instruction Level Parallelism

- Data Hazard: Without pipelining, when each instruction begins, it can safely assume that the previous instruction has completed. With pipelining, that assumption is no longer true.
- Consider:
 - `add %eax, %ebx`
 - `add %ebx, %ecx`
- The first add will not have saved the new value to %ebx by the time the second add needs to read from it.

Instruction Level Parallelism

- Control Hazard:
- When a conditional operation enters the pipeline, it's expected that another operation will enter the pipeline in the next cycle.
- But what instruction should that be if the first operation was conditional?
- We guess using branch prediction, but if we guess wrong, we wasted time doing unnecessary work.

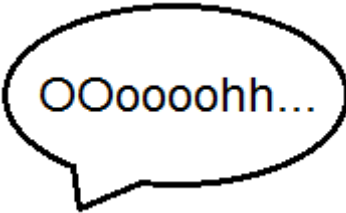

Current State of ILP

- The pipeline processor covered in these slide are a basic overview as to how a single processor or a single execution unit can be used in parallel.
- From here on out, the book assumes multi-issue and out-of-order processors.
- This assumption is key to understanding the dataflow diagrams in 5.7

Post-Pipelining

- Some advanced techniques included:
 - Multi-issue processors: pipelined as usual, but each stage can accommodate multiple instructions at once.
 - Deeper pipelines (modern processors have 12-19 stages)
 - Out of Order (OoO), that is avoiding data dependencies by executing the instructions out of order if possible.

Post-Pipelining

- Some advanced techniques included:
 - Multi-issue processors: pipelined as usual, but each stage can accommodate multiple instructions at once.
 - Deeper pipeline (processors have 12-19 stages)
 - Out of Order () that is avoiding data dependencies by executing the instructions out of order if possible.

Out of Order Processing

- The traditional pipeline allows us to make use of the fact that some instructions can be safely executed at the same time while maintaining correctness.
 - `movl $1, %eax`
 - `movl $2, %ebx`
- There are no dependencies between the two, so as a result, there's no reason not to execute them simultaneously.

Out of Order Processing

- However, consider the following case:
 - `xorl %eax, %eax`
 - `movl (%edx), %esi`
 - `movl (%esi), %edi`
 - `movl (%edi), %edx`
 - `movl $1, %ebx`
 - ...
 - `movl $2, %ecx`
- Why might we have this sequence of loads?

Out of Order Processing

- Maybe you have:

```
struct list{  
    struct list* head;  
};
```

- ...and you're iterating through three elements of the linked list. So maybe it's not totally ridiculous.

Out of Order Processing

- In any case, once you hit that sequence of `movl`'s, the traditional pipeline is out of luck (ignoring forwarding).
 - `movl (%edx), %esi`
 - `movl (%esi), %edi`
 - `movl (%edi), %edx`
- While the pipeline is handling the memory read from `(%edx)`, we can't go ahead with the next instruction. We have to wait.

Out of Order Processing

- It's understood that sometimes you won't be able to parallelize, even with a pipeline.
- Sometimes there just isn't any parallelism to exploit.
- However, is that really what happened here?
There's simply no parallelism in the code?

Out of Order Processing

- `xorl %eax, %eax`
 - `movl (%edx), %esi`
 - `movl (%esi), %edi`
 - `movl (%edi), %edx`
 - `movl $1, %ebx`
 - ...
 - `movl $2, %ecx`
- In fact, the instructions `%eax`, `%ebx`, and `%ecx` can all be done in parallel with each other and the memory references.

Out of Order Processing

- Therefore, we have a program which *has* parallelism, but the traditional pipeline cannot make use of.
- Thus, OoO, which allows the hardware to find as much parallelism as possible in a block of code by reordering as it can see fit.
- How does this help?

Out of Order Processing

- We can't change the order of the memory references and we still have to wait in between the the dependent memory accesses...
- However, instead of waiting, we can interleave the other independent instructions within, thereby getting other stuff done while having to wait.

Out of Order Processing

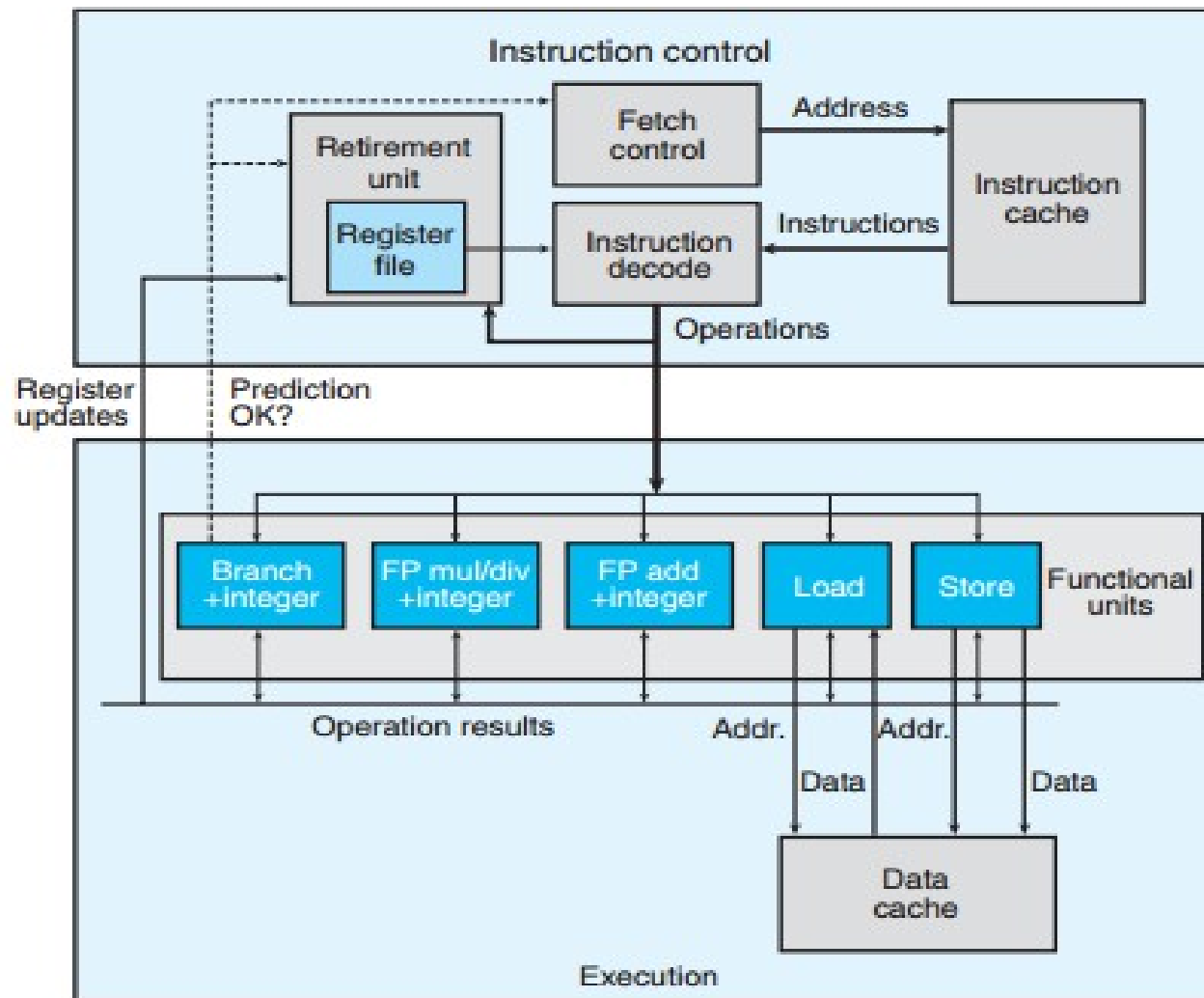
- Keep this idea of executing instructions out of order in mind since the following assumes this information.
- The next sections about data-flow only make sense when you can accept that the order in which instructions are executed doesn't matter, only behavioral correctness.

Out of Order Processing

- Warning/Disclaimer: The following examples are from the 2nd edition of the textbook.
- It's not really like the examples are fundamentally any different, but please note that some of the notation/numbers are going to be slightly different.
- This means that the page numbers listed are 2nd ed. page numbers.

Processor Overview

Section 5.7 Understanding Modern Processors 497



Processor Overview

- The important thing to consider for now are the parts listed in the lower “Execution” block.
- More specifically: there are different specialized functional units which handle a different operations.
- When servicing each instruction, the data will flow to a different functional unit depending on the instruction.
- As a result, performance will differ between different instruction types.

Processor Overview

- How exactly do we measure performance?
- The book (and no one else) uses a lower bound on CPE (cycles per element).
- Which is to say, given a repeated sequence of a given instruction, what's the min no. of cycles it takes to complete a single iteration.

Processor Overview

Bound	Integer		Floating point		
	+	*	+	F *	D *
Latency	1.00	3.00	3.00	4.00	5.00
Throughput	1.00	1.00	1.00	1.00	1.00

- Consider floating point multiplication.
- This chart implies that each floating point mult will take 4 cycles for each iteration if each iteration must be done serially (no overlapping/pipelining).
- However, if it can be pipelined, then a floating point multiplication can be completed every cycle

Characterizing Performance

- Consider combine4 (pg. 493), modified to apply to floating point multiplication.

```
float acc = 1;
```

```
float *data = [INITIALIZE ARRAY];
```

```
for (i = 0; i < length; i++)
```

```
{
```

```
    acc = acc * data[i];
```

```
}
```

```
*dest = acc;
```

- How can we characterize performance?

Characterizing Performance

- There's a floating point multiplication. Should we use the latency bound (4 cycles) or the throughput bound (1 cycle).
- To answer this question, we have to answer the question: can this sequence of operations be pipelined?
- To answer *that* question, we have to answer the question: for each operation, is there a dependency that prevents us from pipelining?
- To answer *that* question...

Characterizing Performance

- The assembly for this

```
combine4: data_t = float, OP = *
i in %rdx, data in %rax, limit in %rbp, acc in %xmm0
1  .L488:                                loop:
2      mulss    (%rax,%rdx,4), %xmm0      Multiply acc by data[i]
3      addq     $1, %rdx                  Increment i
4      cmpq     %rdx, %rbp                Compare limit:i
5      jg       .L488                     If >, goto loop
```

- Which instruction dependencies do we have?

Characterizing Performance

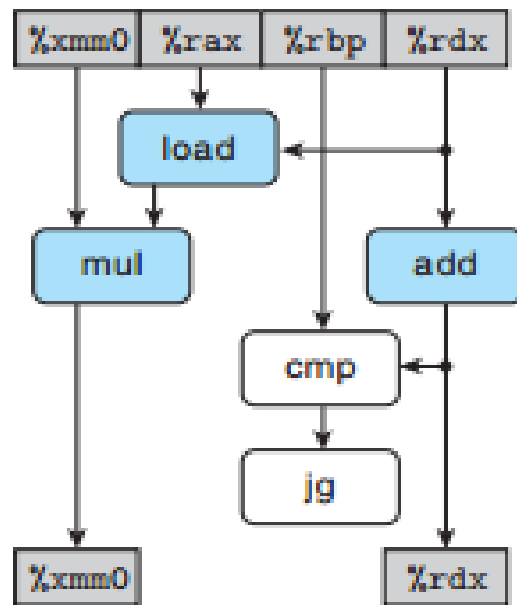
- The assembly for this

```
combined: data_t = float, OP = *
i in %rdx, data in %rax, limit in %rbp, acc in %xmm0
1  .L488:                                loop:
2      mulss    (%rax,%rdx,4), %xmm0      Multiply acc by data[i]
3      addq     $1, %rdx                  Increment i
4      cmpq     %rdx, %rbp                Compare limit:i
5      jg       .L488                     If >, goto loop
```

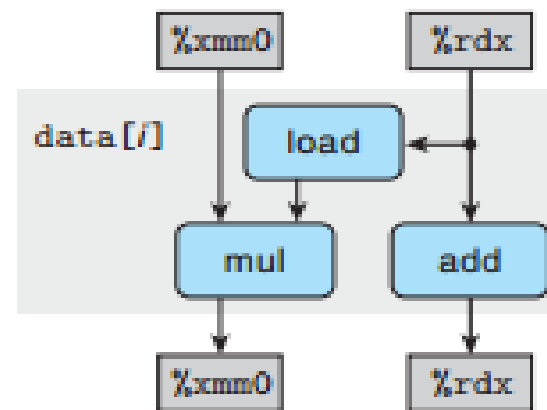
- Which instructions dependencies do we have?
 - 4 relies on 3 (for %rdx).
 - 2 must complete before 3 is completed
- Before we continue, mulss is split into two micro-operations. One to load (%rax, %rdx, 4) and one to multiply the loaded value with %xmm0

Characterizing Performance

- Now... we draw pictures:



(a)

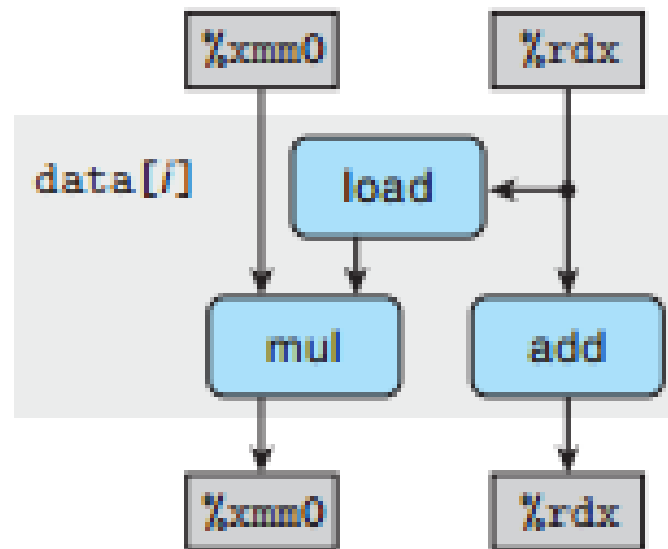


(b)

- A. links up all registers to their instructions.
- B. simplifies this by removing registers that are unchanged and instructions that are not “part of some chain of dependencies”, that is, don't affect registers.

Characterizing Performance

- In particular now, what is the “critical path” or the longest path (in terms of time) through this one iteration?



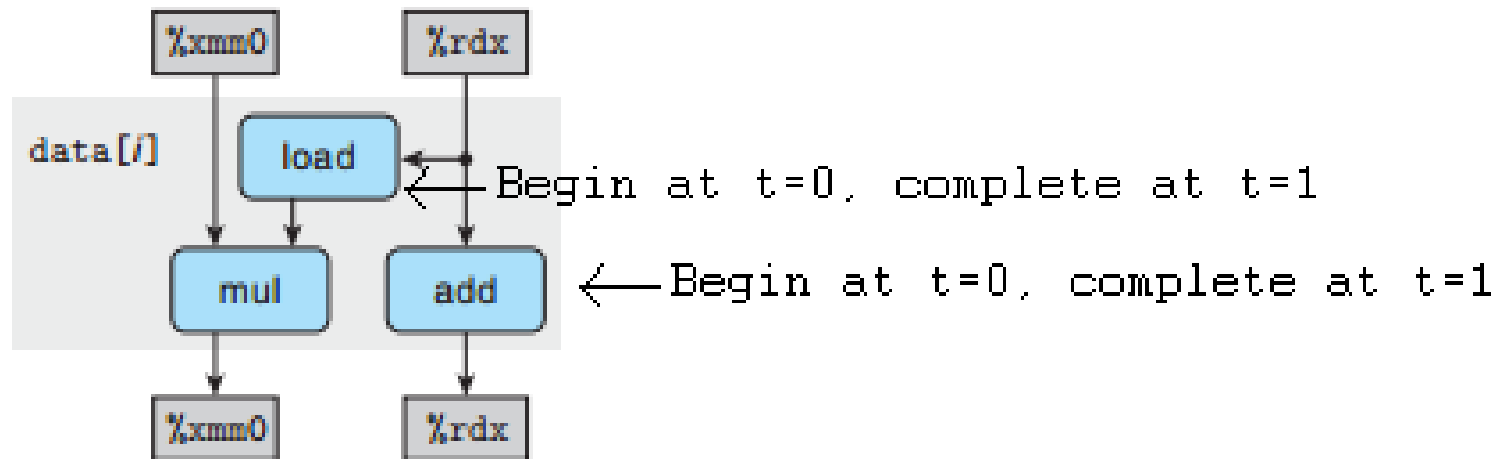
(b)

Characterizing Performance

- As stated in 5.12 (which would be nice info to have here in section 5.7), the peak CPE of a load is 1.
- Let's assume that now.
- Also, latency of floating point multiplication is 4, integer add is 1.
- Consider iteration 0:

Characterizing Performance

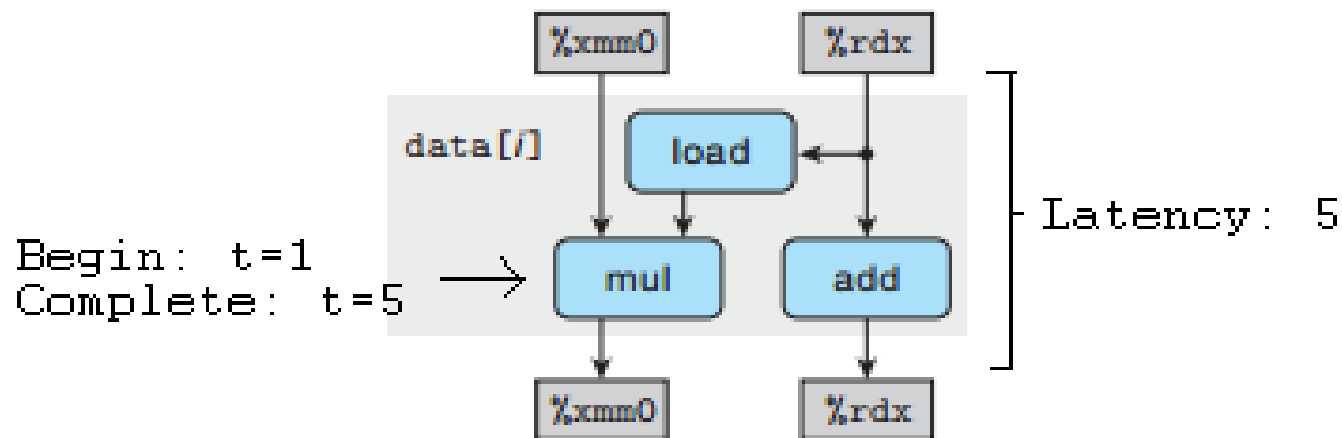
- Iteration 0:



- In the beginning, the mul cannot occur until the load has completed.
- Thus, the load and the add can occur concurrently while the mul must wait.

Characterizing Performance

- Iteration 0:



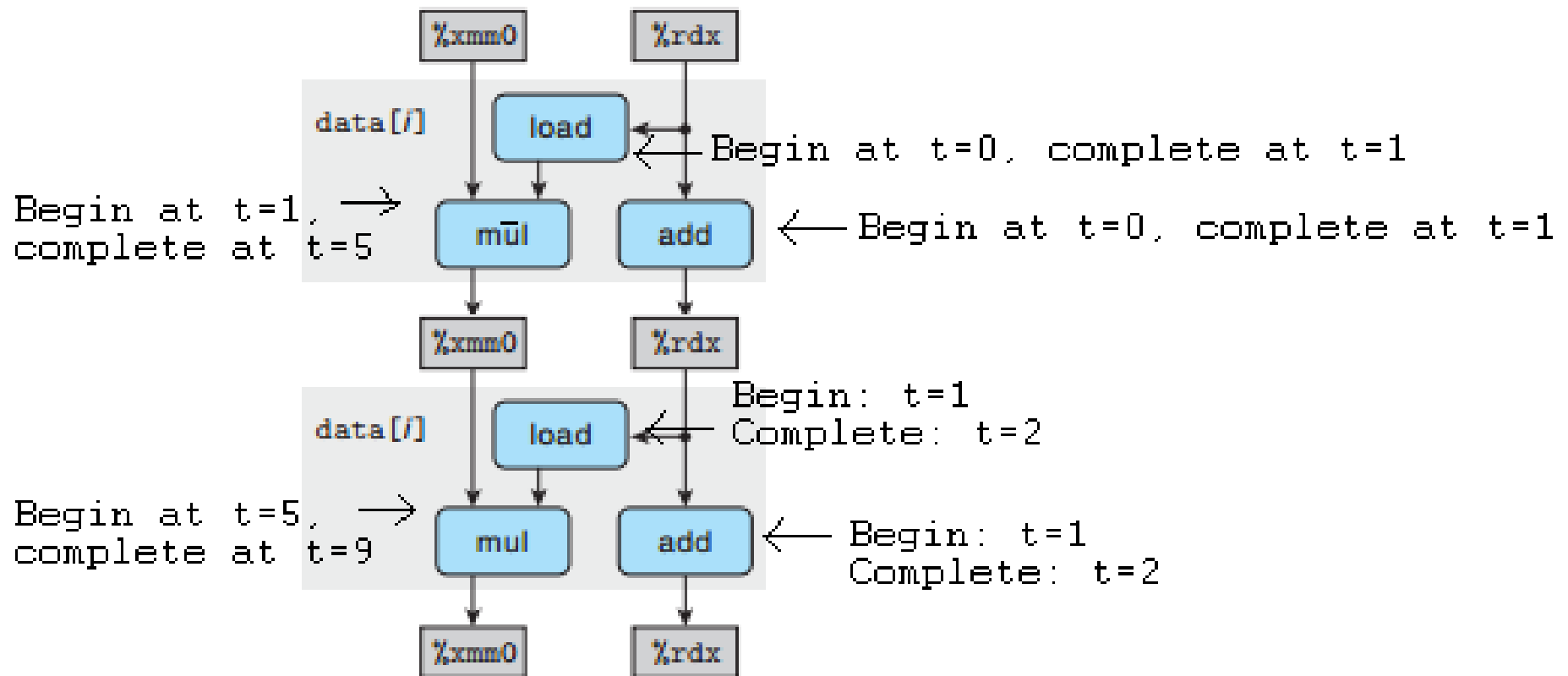
- Thus, the mul occurs after one cycle and completes 4 cycles later.
- As a result, the latency through the first iteration is 5

Characterizing Performance

- Note however, that we are assuming OoO and parallelism.
- When the load of iteration 0 completes, the next load (iteration 1) will rely on the new value of %rdx.
- The new value of %rdx is assigned as a result of the add, which also only takes one cycle.
- As a result, the load/add of iteration 1 can start while the multiplication of iteration 0 is occurring.

Characterizing Performance

- Iteration 0 + 1



Characterizing Performance

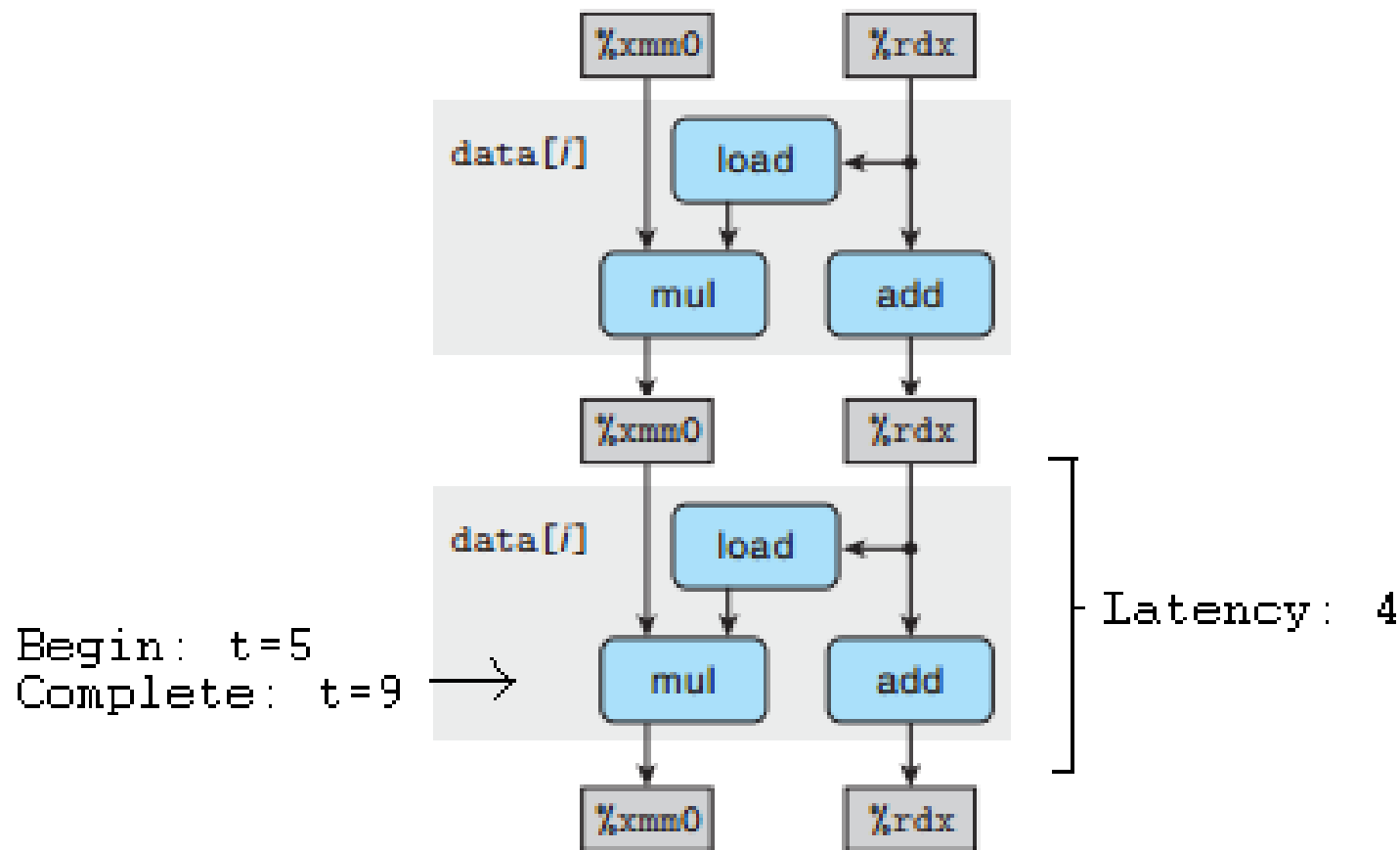
- The mul of iteration 1 must wait until the previous multiplication is complete because the multiplication reads from %xmm0 and writes to %xmm0.
- However, it doesn't have to wait any extra time for the load, because the load of iteration 1 is completing while the mul of iteration 0 is occurring.
- Just as long as we make sure that the add of iteration 1 doesn't complete before we do the load and mul of iteration 0, this can be done in parallel

Characterizing Performance

- The easy way to identify the critical path in a data flow diagram is to find the the longest path between register A of the previous iteration and register A of the next iteration, not the longest path from the the previous iteration to the next iteration.

Characterizing Performance

- Iteration 0 + 1:



Loop Unrolling II: The Revenge

- Previously, we considered the cost reduction of in-order loop unrolling.
- It wasn't promising.
- Now let's look at it from this CPE perspective in a system that allows for out of order execution.

Loop Unrolling II: The Revenge

- This is the code:

```
float acc = 1;
float *data = [INITIALIZE ARRAY];
for (i = 0; i < length; i+=2)
{
    acc = (acc * data[i]) * data[i+1];
}
...the rest of it
```

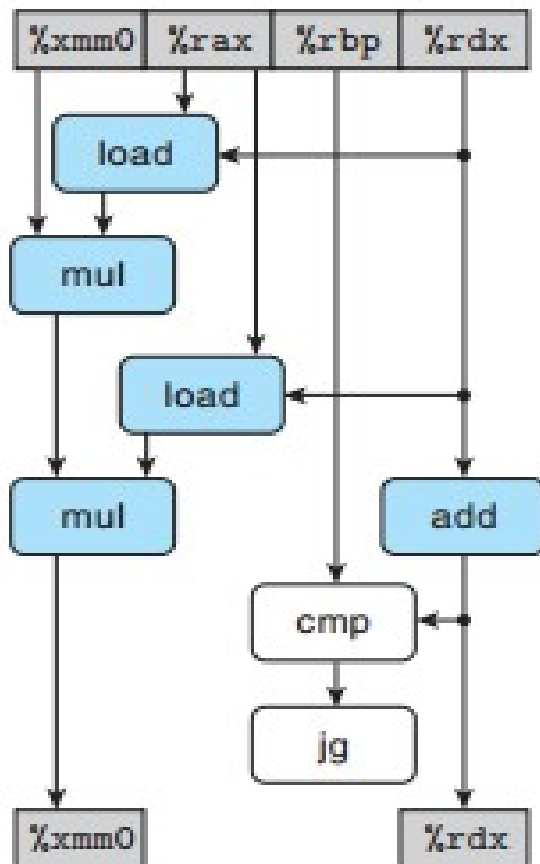
Loop Unrolling II: The Revenge

- This is the assembly:

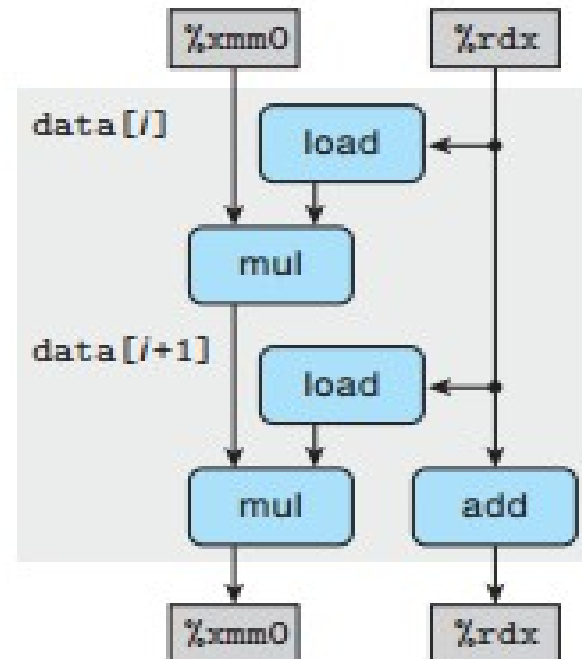
```
mulss (%rax, %rdx, 4), %xmm0  
mulss 4(%rax, %rdx, 4), %xmm0  
addq 2, %edx  
cmpq %rdx, %rbp  
jg loop
```

Loop Unrolling II: The Revenge

- This is the picture:



(a)



(b)

Loop Unrolling II: The Revenge

- The two loads can be done simultaneously.
- As before, the loads/add of iteration $i+1$ can be done while doing the multiplication of iteration i .
- However, now for `%xmm0`, we have to do two multiplications. Neither of these can be done out of order and neither can be overlapped with each other or with the other loops.
- `%xmm0` \rightarrow `mul` \rightarrow `mul` \rightarrow `%xmm0` is the critical path.

Loop Unrolling II: The Revenge

- As a result, the latency through one iteration is 8 and we've gotten even less benefit from loop unrolling than in the in-order case.
- This was an even more confusing way to reach a even lamer result.
- What happened?

Loop Unrolling II: The Revenge

- The whole point of out-of-order (and this section) is the exploit parallelism.
- The unadorned assembly that was written was inherently serial.
- It wasn't so much the fault of unrolling as it was that we weren't looking hard enough for parallelism.
- So how can we parallelize this?
 - `acc = (acc * data[i]) * data[i+1];`

Loop Unrolling II: The Revenge

- The dependency is based on the fact that both operations act upon %xmm0, which leads to serialization.
 - In particular, it means that for:
$$\text{acc} = (\text{acc} * \text{data}[i]) * \text{data}[i+1];$$
 - The first thing that's done is $\text{acc} * \text{data}[i] = \text{temp}$, and THEN $\text{temp} * \text{data}[i+1]$.
- How can we do this in parallel?

Loop Unrolling II: The Revenge

- One idea: use two accumulators.
- The problem is that for each loop, we rely on one accumulator to hold the result of two computations.
- Let's use two accumulators.

Loop Unrolling II: The Revenge

- Here's the code:

```
for(i = 0; i < length; i+2)
{
    acc0 = acc0 * data[i];
    acc1 = acc1 * data[i+1];
}
acc0 = acc0 * acc1
...and the rest.
```

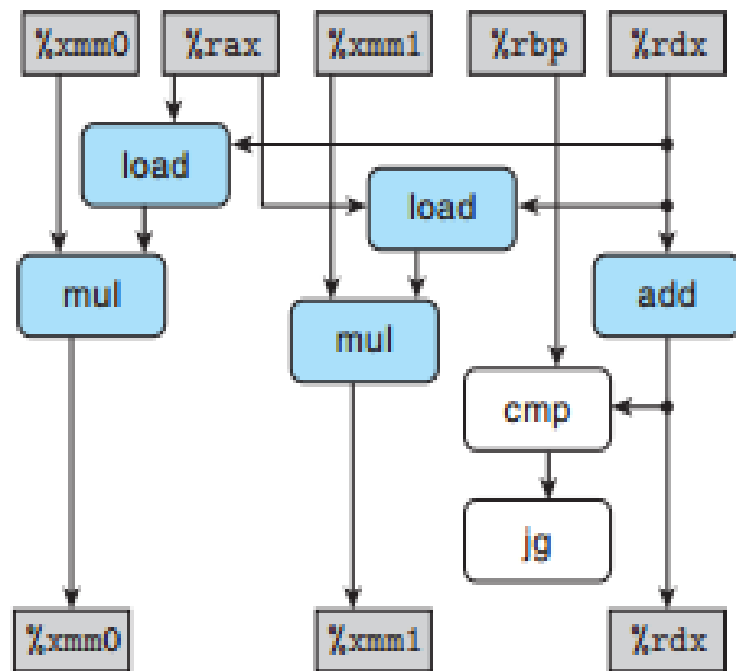
- Note that this optimization... is technically not correct because floating point is not generally associative, but we'll allow some imprecision for now.

Loop Unrolling II: The Revenge

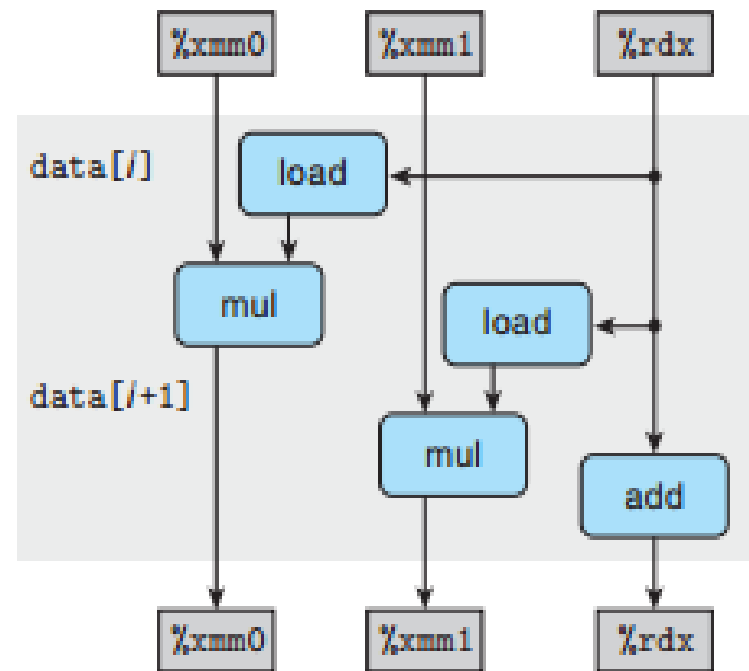
- Here's the assembly:
 - `mulss (%rax, %rdx, 4), %xmm0`
 - `mulss 4(%rax, %rdx, 4), %xmm1`
 - `addq 2, %rdx`
 - `cmpq %rdx, %rbp`
 - `jg loop`

Loop Unrolling II: The Revenge

- ...and here's the picture:



(a)



(b)

Loop Unrolling II: The Revenge

- Now, because there are xmm0 and xmm1 and only one multiplication operates on each register, the multiplication that handles xmm0 and xmm1 can be done in parallel without any concern for serial dependency.
- Also assume that load/add can be done out of order.
- The new CPE is 4. We've cut down the amount of cycles by $\frac{1}{2}$.
- That's more like it.

Loop Unrolling II: The Revenge

- Is there another way to “fix” this?

`acc = (acc * data[i]) * data[i+1];`

Loop Unrolling II: The Revenge

- Surprisingly perhaps, let's consider the “()” part of all of this.
- $\text{acc} = (\text{acc} * \text{data}[i]) * \text{data}[i+1];$
- What's the order of operations?
 - 1. $\text{acc} * \text{data}[i]$ (call this term temp), depends on acc from previous iteration
 - 2. $\text{temp} * \text{data}[i+1]$, depends on completion of temp, which depends on acc.
- This ordering seems to be a problem.
- Let's try $\text{acc} = \text{acc} * (\text{data}[i] * \text{data}[i+1]);$

Loop Unrolling II: The Revenge

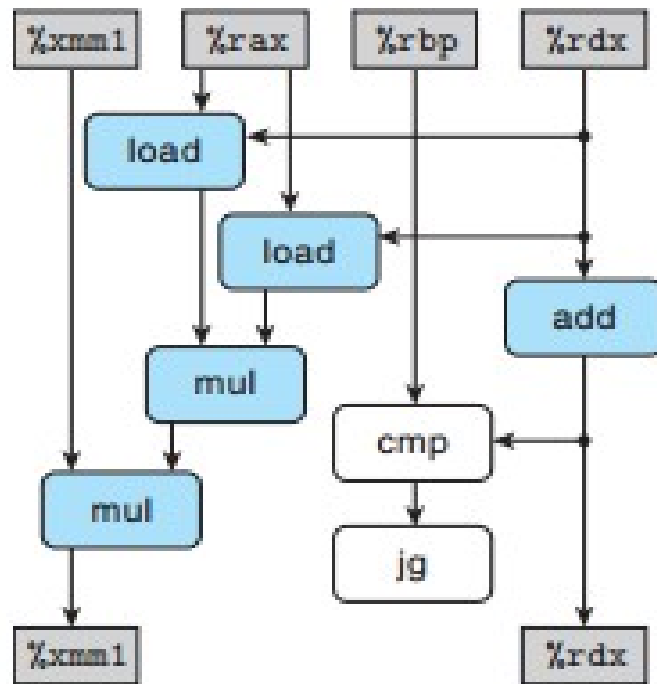
- Here's the assembly:

```
movss (%rax, %rdx, 4), %xmm0
mulss 4(%rax, %rdx, 4), %xmm0
mulss %xmm0, %xmm1
add 2, %rdx
cmpq %rdx, %rbp
jg loop
```

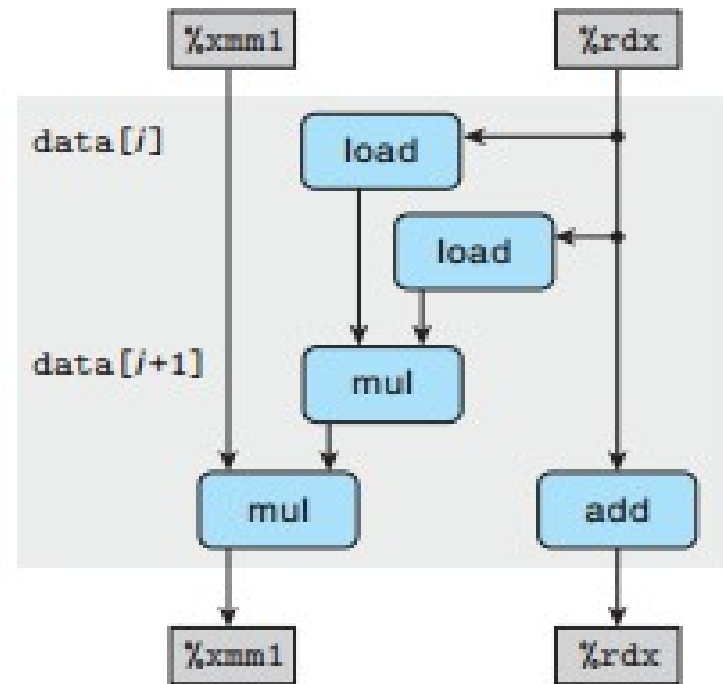
- The result for $\text{data}[i] * \text{data}[i+1]$ is in %xmm0. The accumulator is in %xmm1
- Again, not technically correct due to floating point not being associative, but whatever.

Loop Unrolling II: The Revenge

- Here's the picture:



(a)



(b)

- Well, it looks like we still have two muls in serial... game over...

Loop Unrolling II: The Revenge

- But wait, let's think about this.
- Consider iteration 0. The two loads and the adds occur in parallel. Say they start at $t=0$ and end at $t=1$.
- Then, the first muls ($\text{data}[i] * \text{data}[i+1]$) can start at $t=1$ and end at $t=5$
- Then, the second muls ($\text{acc} * (\dots)$) can start at $t=5$ and end at $t=9$.
- Thus, we complete the first iteration with a nine cycle latency.

Loop Unrolling II: The Revenge

- But wait, the loads and the add only read from %rdx, which means that the loads/add of iteration 1 can begin at $t=1$ and end at $t=2$.
- But wait, if you think about it, $\text{data}[i]$ does not depend on acc , therefore $\text{data}[i] * \text{data}[i+1]$ of iteration 1 can begin at $t=2$ and end at $t=6$.
- But wait, by the time we get to the second muls of the next iteration, the first muls will have already been completed.
- But wait, that means that the critical path is only 1 muls so the latency for each subsequent iteration is 4.
- But wait, this technique is called re-association.

Practice Midterm: Q9

- 9. Section 5.9.2 of the book shows how to apply the associative law to improve performance while unrolling a loop. Can we use a similar idea to improve performance by applying the distributive law $A*(B + C) == A*B + A*C$? Why or why not?

Practice Midterm: Q9

- What does that mean? What's A? What's B?
- Consider the two reasonable cases:
- 1. A is accumulator, b and c are arrays:
 - $A = A * (b[i] + c[i])$ vs. $A = A * b[i] + A * c[i]$
- 2. B is accumulator/single variable, a and c are arrays:
 - $B = a[i] * (B + c[i])$ vs. $B = a[i] * B + a[i] * c[i]$

Segue

- How realistic was that assumption that load (ie memory accesses) take only one cycle?
- According to your book (pg 603, 3rd ed), a realistic access time of DRAM is 20 ns.
- The clock rate is listed as being 3 GHz.
- GHz = cycles per seconds
- This means $1/(3 \times 10^9)$ seconds per cycle = .3333 ns.

Segue

- This means that a single clock cycle is .333333 ns.
- A single instruction basic arithmetic instruction (integer add, floating point multiply) takes 1 to 4 cycles.
- The 20 ns DRAM access time takes 60 cycles.

Segue

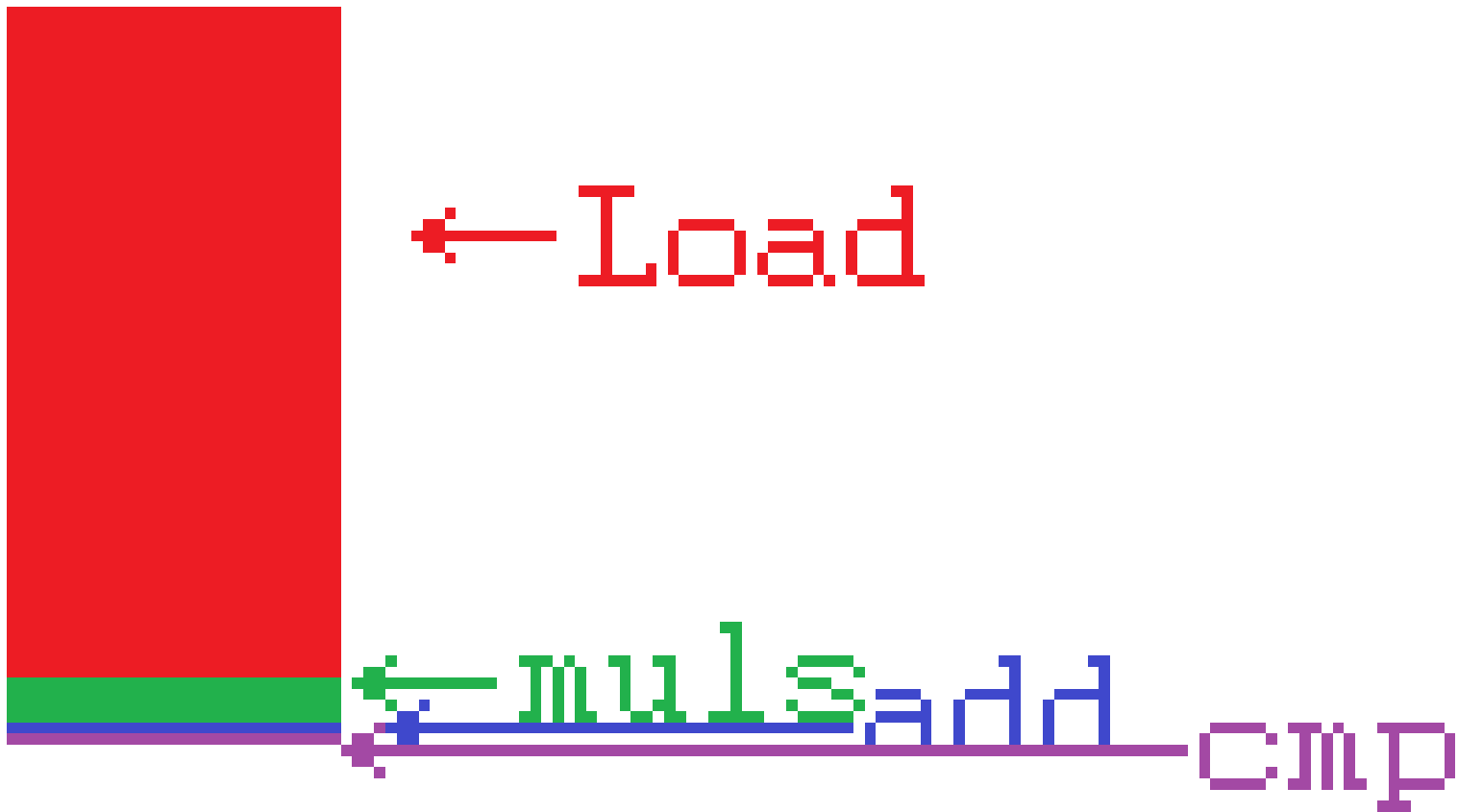
- That suggests of these four instructions:

mulss		load
addq	=>	mul
cmpq		add
		cmp

- The proportion of time spent doing each instruction is this:

Segue

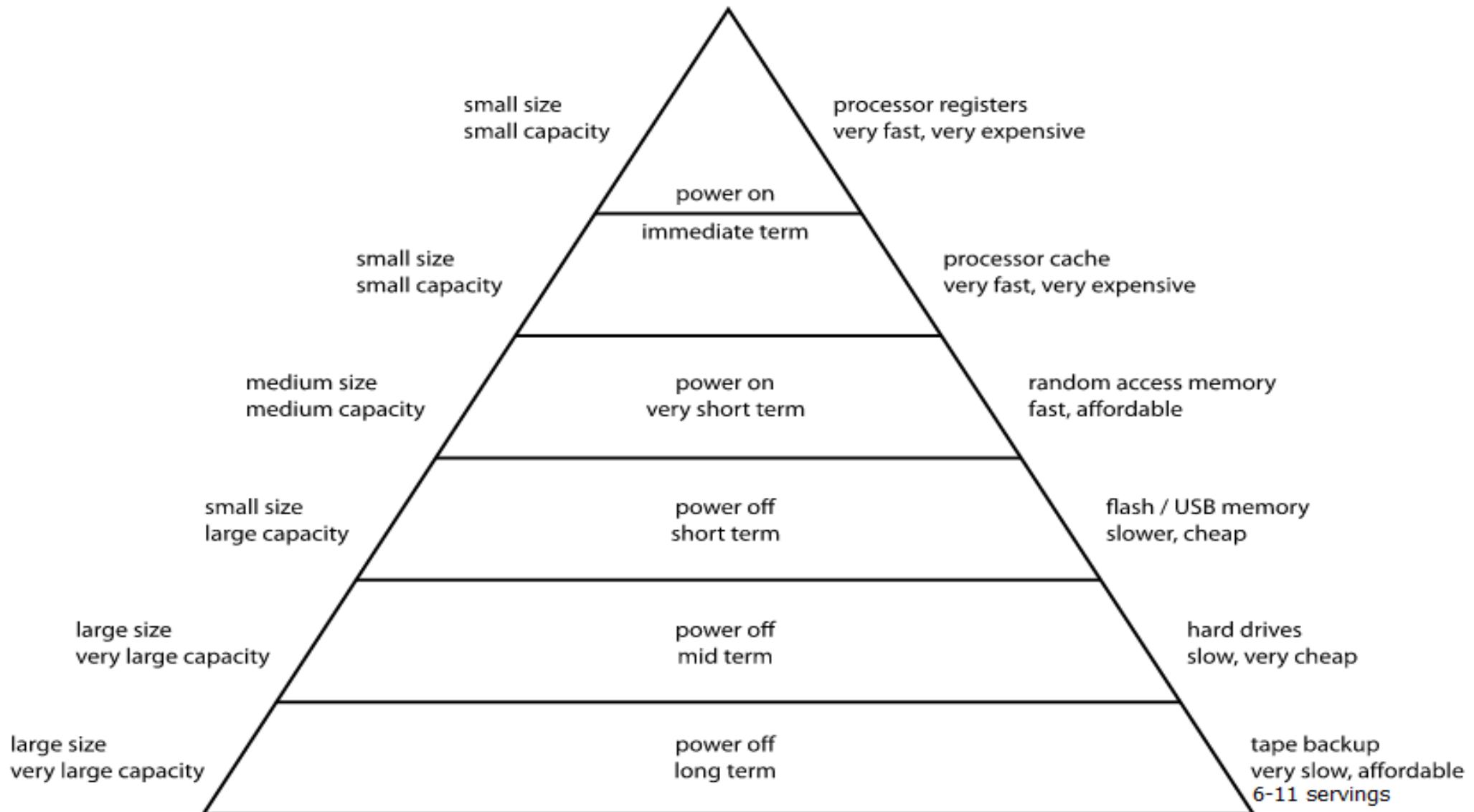
- This:



Segue

- Obviously, this is unacceptable.
- Also... this is not what we describe as being accurate.
- So given how slow main memory is, how do we actually create a usable system?
- Consider the memory hierarchy pyramid

Memory Hierarchy



Courtesy of Wikipedia.org

Memory Hierarchy

- The higher up on the pyramid, the faster the access, but the lower the capacity.
- Because memory is relatively slow, we'd like to spend as much time as possible dealing with the upper levels of the pyramid.
- The very top layer is are the registers.

Memory Hierarchy

- With the RISC-like micro-operations, we have the rule that in order to do anything with data, we must first operate on it in a register.
- Hence, no `movl (%ebx), (%ecx)`
- The rationale is that if we have a working variable, we'd like to operate on the highest level of the hierarchy as possible to speed things up.
- In a way, this means that registers are like quick temporary copies of the values in memory.

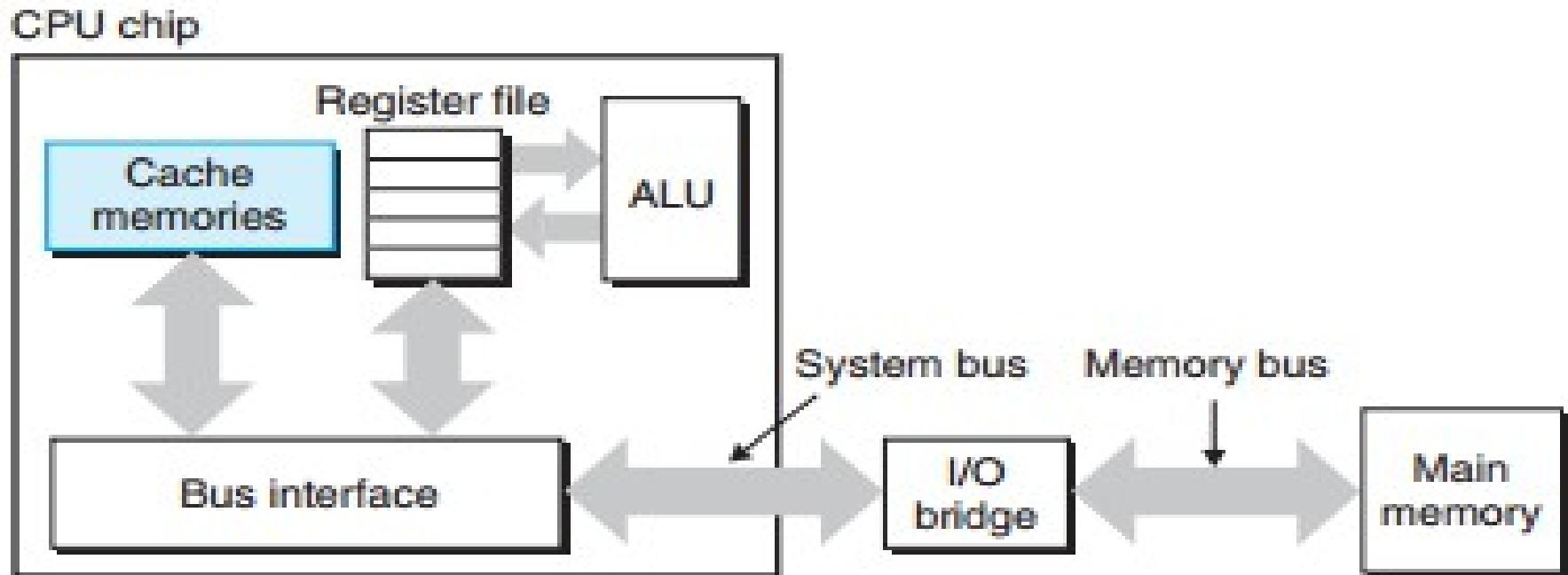
Memory Hierarchy

- When dealing with a value (for example, consider the accumulator), you do all computations on registers.
- Then you commit to memory. That way, you can avoid going to memory altogether.
- However, being at the very tip of the hierarchy means you only have a limited amount of registers.
- What if you wanted to deal with an array?

Memory Hierarchy

- As a result, we introduce a layer of memory that is hidden from assembly/machine instructions.
- This layer operates on the same principle:
 - When you need to operate on data, copy it from DRAM into a faster (but smaller) memory.
 - Once you need it to be committed to actual main memory, copy it back.
- Unlike registers however, it does so implicitly.
- This layer is the cache.

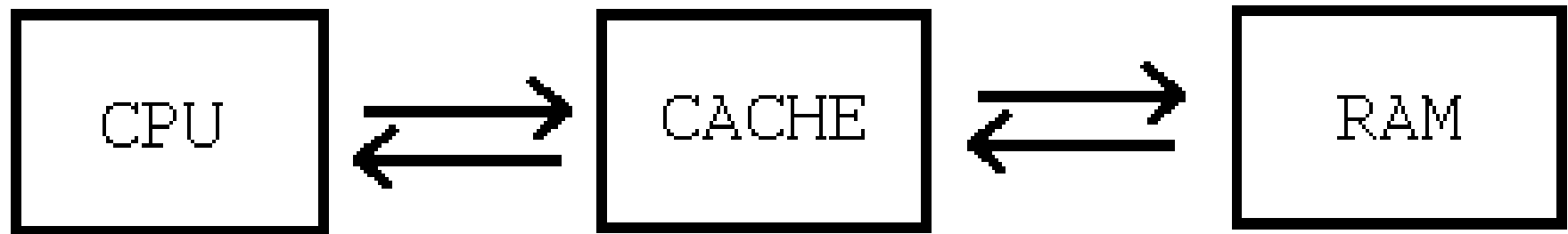
Revised CPU



- Some diagrams will show this. The cache is a separate unit that sits in the CPU.

Revised CPU

- For our purposes, however, we can simplify the cache as follows.
- This diagram suggests that any interaction with the RAM must first “go through” the cache.



Caching

- Extremely broad overview
- Consider:
 - `movl (%ebx), %eax`
 - `%ebx = 0x10`

Caching

- 1. Issue the address to the cache. Does the cache contain the data at address 0x10?
- 2. If the cache has it, then give the data back to the CPU. Else fetch it from RAM



- Implicitly, EVERY memory access goes through the cache. What happens next is where the magic lies.

Caching

- May seem complicated, but if it's in the cache, access time is ~ 1 cycle, completely masking the ludicrous 60 cycle delay of memory.
- There are four cases to consider in terms of behavior:
 - Read hit: Read from memory, the cache has the data you're looking for.
 - Read miss: Read from memory, the cache doesn't have the data you're looking for.
 - Write hit: Write to memory, the cache has the data you're looking for.
 - Write miss: You can probably figure this one out

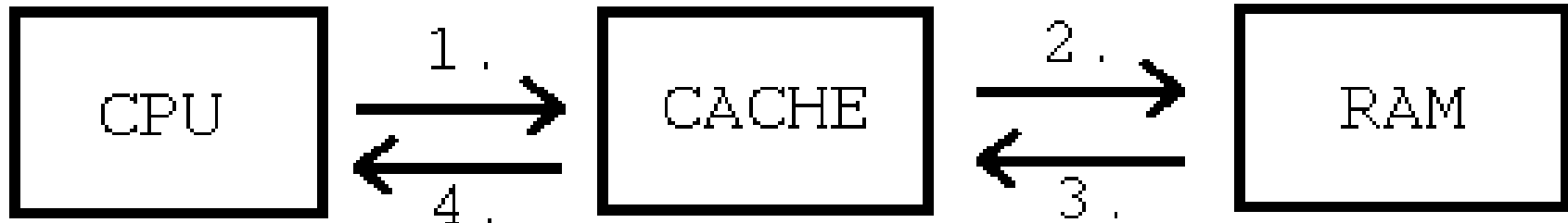
Caching

- Read Hit:
 - 1. Issue request to cache. The data we're looking for is in the cache.
 - 2. Respond with the data to the CPU



Caching

- Read Miss:
 - 1. Issue request to cache. The data we're looking for is NOT in the cache.
 - 2. Issue request to RAM. As far as you know, the data MUST be in RAM.
 - 3. Respond with data first to cache. Store the data in the cache.
 - 4. Respond with data back to CPU.



Caching

- When we have a cache miss, why do we go through the trouble of pulling the data into the cache?
- Well, we have to have some way of filling the cache.
- Either we populate the cache based on cache misses or we try to pre-populate it based on other information from compilation or running time, which is extremely tricky.

Caching

- First of all, how do we decide what to pull into the cache?
- Let's say, we have `movl (%ebx), %eax`.
- We need 1 word from memory.
- So we pull in one word from memory into the cache?

Caching

- Caching makes use of a key principle called locality:
 - Temporal Locality: Data that you access is very likely to be accessed again in the near future.
 - Spatial Locality: If you access a piece of data, it's likely that you'll access data that is in a nearby address in the near future.

Caching

- Pulling in a single word would make use of temporal locality, but if we're going to memory, it doesn't make sense to just pull in a value especially since pulling two words from memory is not twice as slow as pulling one word from memory.
- Consider the case where you're iterating through a size 100 array of ints. If we only pull in one word at a time, the cache doesn't help us at all unless we iterate through the array repeatedly.
- We will make 100 requests and we will miss every single time.

Caching

- However, if we pulled in the entire array into the cache when we accessed the first variable, the entire array is set up for us in the cache.
- We will have 1 miss and 99 hits.
- As another example, let's say we access a variable on the stack.
- Chances are, we're in a function and it would make sense to pull in that entire function's stack frame into the cache at once.

Caching

- Also, I said that the cache is much smaller than main memory. That's how it can physically afford to be fast.
- In that case, that means we can't fit everything in the cache at once, just as with registers.
- With registers, the compiler can figure it out, but the compiler isn't able to figure out the caching rules.
- When the time comes, we'll have to have some policy to prioritize what to throw out of the cache.

Caching

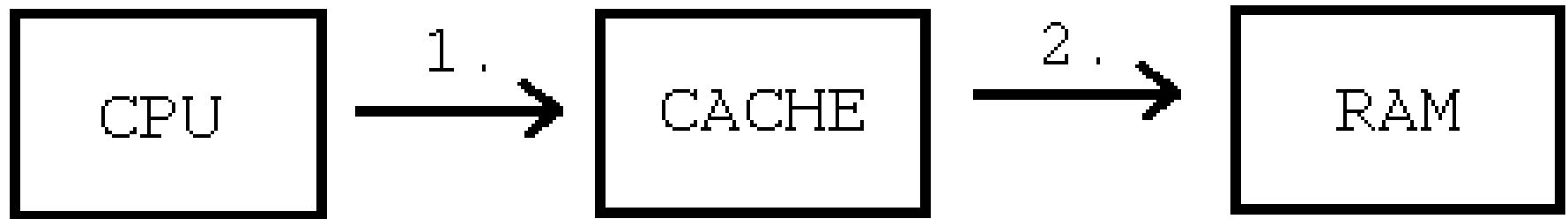
- How much data is brought into the cache at any given access is a function of the cache itself and the defined “cache block” size, which is the granularity in which cache operations are dealt with.

Caching

- Write Hit:
- When you want to write and you find the data is in the cache, you have a choice to make.
- Do you write just to the cache?
 - This is known as the write-back policy.
- Or do I write to the cache AND also write to memory.
 - This is known as the write-through policy

Caching

- Write Hit (Write-through)
- 1. Issue the write to the cache. The data is in the cache. Write to cache
- 2. Issue the write to memory. Write to memory as well



Caching

- Write Hit (Write-back)
- 1. Issue the write to the cache. The data is in the cache. Write to cache.
- Now you have to remember that the cache is inconsistent with memory. Mark the data in the cache with a “dirty bit”.



Caching

- Write Hit (Write-back)
- When that data must be thrown out of the cache to satisfy another request, write the “dirty” data back into memory to make it consistent.



Write-Hit Policy (Write Policy)

- Benefits of Write-Through?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
 - Memory and cache are always synchronized and consistent.
- Downsides?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
 - Memory and cache are always synchronized and consistent.
- Downsides?
 - Have to incur that crazy memory penalty every single time you write.
 - Or maybe not? A write buffer can be used in conjunction with a write-through cache. Write to the write buffer after which the processor is allowed to continue while the buffer writes to main memory.

Write-Hit Policy (Write Policy)

- Benefits of Write-Back?

Write-Hit Policy (Write Policy)

- Benefits of Write-Back?
 - Write-through requires 1 write to memory for every store instruction. Write-back effectively collects all of the writes to a particular block and needs to perform one write to memory. This is much better compared to write through if you're frequently writing to the same block.
- Downsides?

Write-Hit Policy (Write Policy)

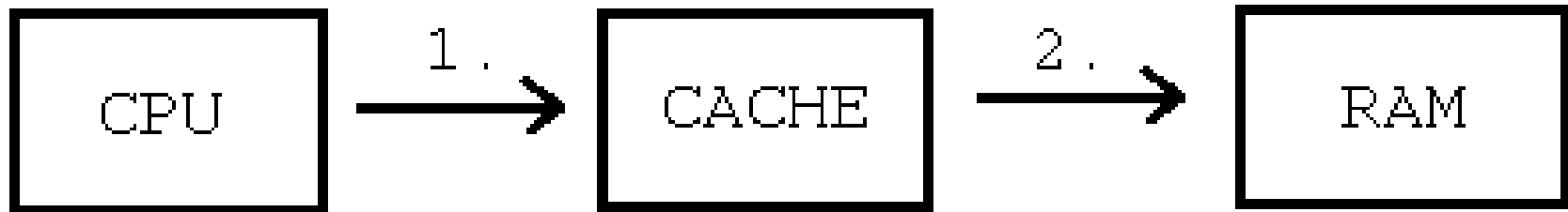
- Downsides?
 - Write-through pays the cost of writing to memory for stores, but write-back may mean paying the cost of writing to memory for loads.
 - Load performance may be more critical.
 - Write-back is going to be a bit more complicated to manage.
- Realistically however, overall performance is going to be much better with a write-back cache.

Write-Miss Policy (Write Policy)

- Write Miss
- When you look for data to write in the cache and you find that it's not there, you again have a choice to make.
- Do you first pull the data into the cache, then write?
 - This is known as write-allocate
- Do you just write the data into memory
 - This is known as no-write-allocate.
 - They sure phoned that name in.

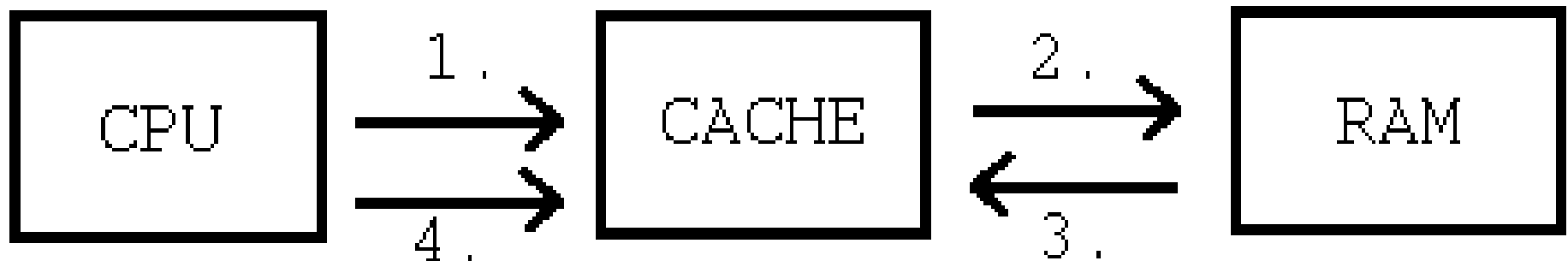
Write-Miss Policy (Write Policy)

- Write miss (no-write-allocate)
- 1. Issue the write to the cache. The data is NOT there.
- 2. Issue the write to RAM. Write to RAM



Write-Miss Policy (Write Policy)

- Write miss (write allocate)
- 1. Issue the write to the cache. The data is NOT there.
- 2. Request the data from RAM (don't write just yet).
- 3. Pull the data from RAM back to cache.
- 4. CPU re-issue write request. Now the write is a hit. Allow write-hit policy to take over



Write-Miss Policy

- Benefits of using write-allocate:

Write-Miss Policy

- Benefits of using write-allocate:
 - Pulling the block into the cache could potentially help future accesses to memory (ex. storing into sequential elements of an array, repeatedly storing and loading due to register spilling)
- For example, if you're writing to every element of a size 100 array that isn't currently in the cache.
- Without write-allocate, you miss and go to memory every time.
- On the other hand...

Write-Miss Policy

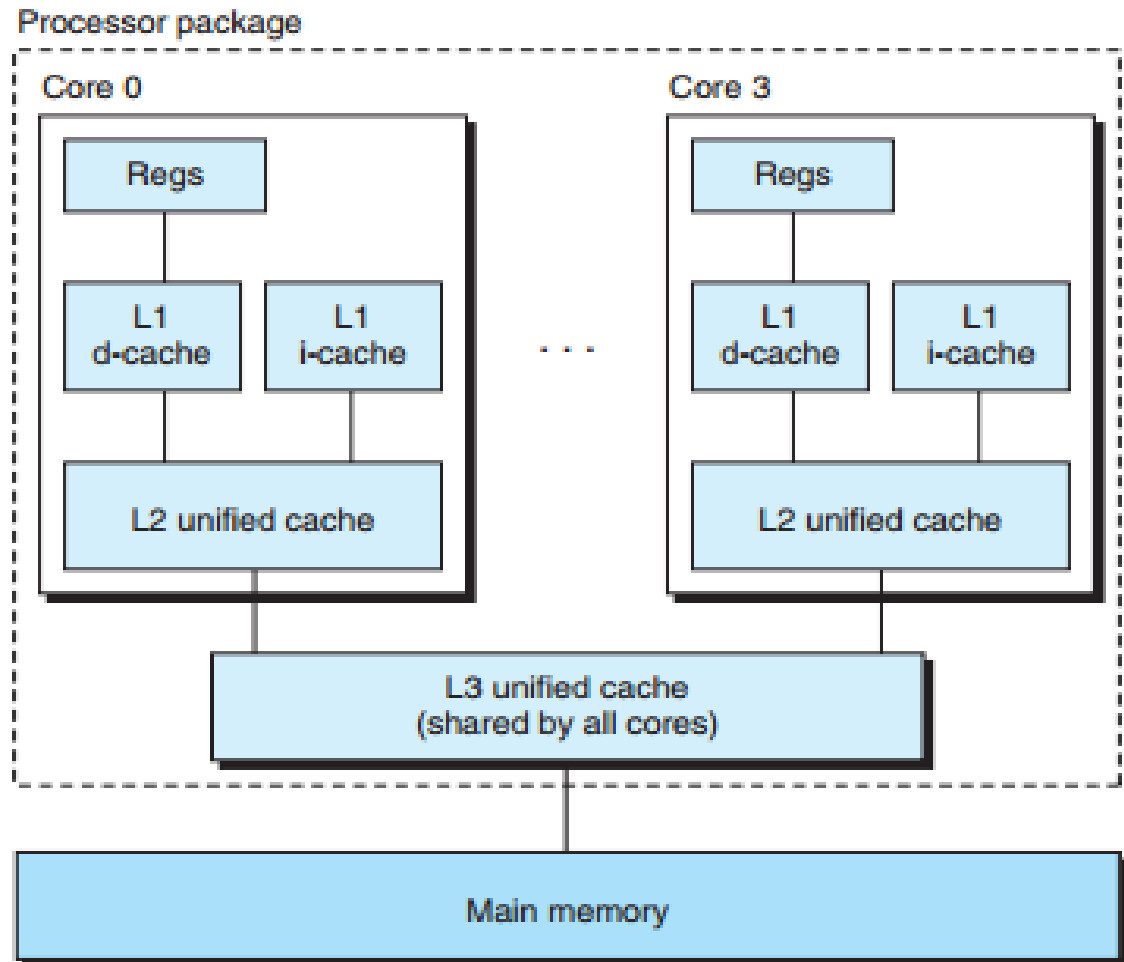
- On the other hand...
 - This is only beneficial if there is “store” locality.
 - Consider the case of sparsely assigning a value to variables that is scattered in an isolated position.
 - If you use write-allocate, all you're doing is needlessly polluting the cache with a block of memory that you may not access again anytime soon.
 - The act of allocating takes time.

Write-Miss Policy

- These are the basic caching operations, but it became clear that a single cache was not enough.
- In order to further exploit locality, processors developed with a hierarchy of caches.

Write-Miss Policy

Figure 6.40
Intel Core i7 cache
hierarchy.



Write-Miss Policy

- If you miss in the $L_{<X>}$ cache, instead of going to memory, you consult the $L_{<X+1>}$ cache, which is larger, but slower, but still way faster than RAM (up until L3).
- Also note that there is a split L1 cache, which is to say that there is a cache that is dedicated for holding Data and a separate cache for holding Instructions. L2 and L3 are unified, that is instructions and data both belong to them.
- Why?

Write-Miss Policy

- Data and Instructions have different access patterns. Instructions generally have strong locality since instructions are clumped together in the code/text segment of memory.
- As a result, the instructions will likely fill the cache nicely.
- However, if data and instructions use the same cache, the data may be trampling on the instructions and vice versa.

Caching

- Final note:
- Typically, courses that cover caching will talk about the details about how actually keep track of data in the cache and how to map the addresses of memory to locations in the cache.
- That's what the book does.
- This isn't exactly a “typical” class.
- What's probably important for the midterm is to understand the concepts, policies, tradeoffs, etc.

Caching

Caching

- Final note:
- Typically, courses that cover caching will talk about the details about how actually keep track of data in the cache and how to map the addresses of memory to locations in the cache.
- That's what the book does.
- This isn't exactly a “typical” class.
- What's probably important for the midterm is to understand the concepts, policies, tradeoffs, etc.

- This was in the CS 33: Spring '15 Slides. However... this is Fall '15

Caching

- Last quarter, the professor didn't lecture much about the specifics of the cache.
- Last quarter, he didn't mention direct-mapped vs. set-associate or how to index into a cache.
- This quarter... he did.
- I still assert that the focus is going to be more on the high level but just in case (and by popular demand), here's 34 more slides on caching.

Cache

- Definitions
- Cache blocks:
 - A chunk of bytes in memory that are adjacent.
 - Ex: If block size of 2^5 bytes, then each block is 32 bytes long.
 - Block 0: MEM[0] to MEM[31]
 - Block 1: MEM[32] to MEM[63]
 - Etc...
 - Note: MEM[15] belongs to block 0.

Cache

- Definitions
- Cache blocks:
 - A chunk of bytes in memory that are adjacent.
 - Ex: If block size of 2^5 bytes, then each block is 32 bytes long.
 - If the address space is 2^{32} and blocks sizes are 2^5 , then there are 2^{27} blocks in memory.
 - All operations performed on cache (eviction, moving), are done on the granularity of blocks.

Cache

- Definitions
- Each cache consists of an “array” of “sets”.
- Each set consists of one or more cache blocks.
- Each block has a corresponding tag, valid, and dirty (sometimes) bit.
- A tag is like an id that can be used to identify a cache block.
- A valid bit indicates whether the value in the cache is a valid block that corresponds to data.

Cache

- Since the cache is so much smaller than the actual memory space, we need a schema for deciding where to place data into the cache.
- Let's follow the same model as normal memory addresses: if the block is at address A... it's at the one specific location specified by address A.
- A block at address A will belong to one specific spot in the cache specified by address A.

Direct Mapped Cache

Set/Index	Valid	Dirty	Tag	Data
0				
1				
2				
3				
...				
$2^n - 1$				

} 2^n sets



1 data block, AKA 1-way/direct mapped

- How do we decide where each block from memory should go?
- We determine a mapping from the physical address

Direct Mapped Cache

- How do we decide where each block from memory should go?
- We determine a mapping from the physical address.
- Each address is split into three components: Tag, Index, Offset
- For example, the address:
- ...11010110101110
- might be split into:
- ...11010 11010 1110
- Tag Index Offset

Direct Mapped Cache

- Tag: The “id” for the block
- Index: Which set the block belongs to.
 - If our cache has 8 sets, we need a tag that is able to choose 8 different values (3 bits)
- Offset: Which byte within the block that is being accessed.
 - If our block size is 16, we need an offset that is able to choose 16 different values (4 bits).

Direct Mapped Cache

Set/Index	Valid	Dirty	Tag	Data
000				
001				
010				
011				
100				
101				
110				
111				

- As an example, let's consider a Direct Mapped Cache with 8 (2^3) sets, and 8 (2^3) byte blocks.
- Consider the case where the addressable space is 2^8

Direct Mapped Cache

- As an example, let's consider a Direct Mapped Cache with 8 (2^3) sets, and 8 (2^3) byte blocks.
- Consider the case where the addressable space is 2^8 .
- How do we translate an address:
- 0000 0000
- ...into a cache location?

Direct Mapped Cache

- 8 (2^3) sets, 8 (2^3) byte blocks, 2^8 addressable space.
- If each block is 8 bytes, we need to be able to access each byte individually. This means we need 3 offset bits.
- There are 8 sets. We also need 3 bits to index into the 8 different sets.
- This leaves 2 bits to be the tag.
- 00 000 000
- T I O

Direct Mapped Cache

- Say we had a write-allocate cache:
- `int x[16];`
- where `x` begins at address 0. Then we accessed each element in increasing order:
- `&x[0] = 00 000 000`
- `&x[1] = 00 000 100`
- `&x[2] = 00 001 000`
- ...

Direct Mapped Cache

- Note: We have 8-byte block sizes. Therefore when we access, for example, `&x[0]`, we pull in both `x[0]` and `x[1]` into the cache.
- Also, since we operate on blocks, `x[1]` belongs to the same block as `x[0]`. If the cache were empty and we accessed `&x[1]`, we would pull in `x[0]` and `x[1]` into the cache.
- Essentially, if you access memory address:
- [tag] [index] [offset]
- XXXXX XXXX XXXXX
- ...you're dealing with the block that starts at:
- XXXXX XXXX 00000

Direct Mapped Cache

- Access order:
- $\&x[0] = 00\ 000\ 000$
- $\&x[1] = 00\ 000\ 100$
- $\&x[2] = 00\ 001\ 000$
- $\&x[3] = 00\ 001\ 100$
- ...
- $\&x[15] = 00\ 111\ 100$

Set/Index	Valid	Dirty	Tag	Data
000	0			
001	0			
010	0			
011	0			
100	0			
101	0			
110	0			
111	0			

Direct Mapped Cache

- Access order:
- $\&x[0] = 00\ 000\ 000$: Set 0, cold miss, bring in $x[0]$ and $x[1]$
- $\&x[1] = 00\ 000\ 100$: Set 0, hit.
- $\&x[2] = 00\ 001\ 000$: Set 1, cold miss, bring in $x[2]$ and $x[3]$
- $\&x[3] = 00\ 001\ 100$: ...
- ...
- $\&x[15] = 00\ 111\ 100$

Direct Mapped Cache

- When you're comparing a memory address to see if the block in the cache corresponds to that address (ie. if you want to see if there was a hit), you have to compare against the “tag”, which is an id.
- For example, if you were making an access to:
 - 00 000 010
- ...and you found data in your corresponding set, but the tag was “10”. Then this data is not yours. Your tag is “00”

Direct Mapped Cache

- Access order:

- 00 000 000

- 00 000 100

- 00 001 000

- 00 001 100

- ...

- 00 111 100

Set/Index	Valid	Dirty	Tag	Data
000	1		00	x[0], x[1]
001	1		00	x[2], x[3]
010	1		00	x[4], x[5]
011	1		00	x[6], x[7]
100	1		00	x[8], x[9]
101	1		00	x[10], x[11]
110	1		00	x[12], x[13]
111	1		00	x[14], x[15]

Direct Mapped Cache

- For each access, you bring in 8 bytes. This means that in the int array access, for each miss, two elements in the array are brought into the cache. This means a hit rate of 50%. Not bad.

Direct Mapped Cache

- Now consider:
- `int x[16];`
- `int y[16];`
- Where `x` begins at address 0 and `y` begins at address 64.

```
for(int i = 0; i < 16; i++)  
{  
    y[i] = x[i];  
}
```

What is the pattern of access now? Also assume a write-allocate cache.

Direct Mapped Cache

- What is the pattern of access now?
- $\&x[0] = 00\ 000\ 000$
- $\&y[0] = 01\ 000\ 000$
- $\&x[1] = 00\ 000\ 100$
- $\&y[1] = 01\ 000\ 100$
- $\&x[2] = 00\ 001\ 000$
- $\&y[2] = 01\ 001\ 000$
- $\&x[3] = 00\ 001\ 100$
- $\&y[3] = 01\ 001\ 100$
- ...

Direct Mapped Cache

- What is the pattern of access now?
- $\&x[0] = 00\ 000\ 000$: Set 0, cold miss
- $\&y[0] = 01\ 000\ 000$: Set 0, cold miss and replace the tag 00 block
- $\&x[1] = 00\ 000\ 100$: Set 0, conflict miss and replace the tag 01 block
- $\&y[1] = 01\ 000\ 100$: Set 0, conflict miss and replace with tag 00 block
- $\&x[2] = 00\ 001\ 000$: Set 1, cold miss
- $\&y[2] = 01\ 001\ 000$: Set 1, cold miss and replace the tag 00 block
- $\&x[3] = 00\ 001\ 100$: ...
- $\&y[3] = 01\ 001\ 100$
- ...

Direct Mapped Cache

- What is the pattern of access now?
- $\&x[0] = 00\ 000\ 000$
- $\&y[0] = 01\ 000\ 000$
- $\&x[1] = 00\ 000\ 100$
- $\&y[1] = 01\ 000\ 100$
- ...
- Notice that $x[i]$ and $y[i]$ belong to the same set for all i . This means each for each access, the cache is being overwritten. The miss rate is 100%!
- This is thrashing (when alternating accesses happen to map to the same set).

Direct Mapped Cache

- Cold Miss/Compulsory Miss: The spot in the cache is empty or this block has never been accessed before. Therefore it cannot be in the cache.
- Capacity Miss: The cache is full and you access a new block.
- Conflict Miss: The cache ISN'T full, but because of how the cache is organized, the block that was previously brought in had to be evicted.

Direct Mapped Cache

- Consider that the cache size is 2^3 sets * 2^3 bytes/set or 64 bytes total.
- x and y are each 64 bytes. We can't hold the entire arrays in the cache, but we should have a better hit rate than 0.
- Enter...

Set Associative Cache

- What if each set could hold two (or more blocks)? Could we resolve this particular thrashing issue?
- Say we want to repurpose our cache which has a data size of 64 bytes.
- Block size: 8 bytes.
- Now two blocks per set and 4 sets. How do we split up the address now?

Set Associative Cache

- What if each set could hold two (or more blocks)? Could we resolve this particular thrashing issue?
- Say we want to repurpose our cache which has a data size of 64 bytes.
- Block size: 8 bytes.
- Now two blocks per set and 4 sets. How do we split up the address now?
 - 000 00 000
 - T I O

2-Way Associative Cache

Set/Index	Valid	Dirty	Tag	Data	Valid	Dirty	Tag	Data
000	0				0			
001	0				0			
010	0				0			
011	0				0			

Set Associative Cache

- Now, we have two blocks in each set. This means that each set can comfortably seat two blocks without having to evict anything.
- Again, assume a write-allocate cache.

2-Way Associative Cache

- Consider the access pattern again:

- $\&x[0] = 000\ 00\ 000$

- $\&y[0] = 010\ 00\ 000$

- $\&x[1] = 000\ 00\ 100$

- $\&y[1] = 010\ 00\ 100$

- $\&x[2] = 000\ 01\ 000$

- $\&y[2] = 010\ 01\ 000$

- $\&x[3] = 000\ 01\ 100$

- $\&y[3] = 010\ 01\ 100$

- $\&x[4] = 000\ 10\ 000$

- $\&y[4] = 010\ 10\ 000$

- $\&x[5] = 000\ 10\ 100$

- $\&y[5] = 010\ 10\ 100$

- $\&x[6] = 000\ 11\ 000$

- $\&y[6] = 010\ 11\ 000$

- $\&x[7] = 000\ 11\ 100$

- $\&y[7] = 010\ 11\ 100$

After 8 iterations...

2-Way Associative Cache

Set/Index	Valid	Dirty	Tag	Data	Valid	Dirty	Tag	Data
00	1		000	x[0], x[1]	1		010	y[0], y[1]
01	1		000	x[2], x[3]	1		010	y[2], y[3]
10	1		000	x[4], x[5]	1		010	y[4], y[5]
11	1		000	x[6], x[7]	1		010	y[6], y[7]

- 50% hit rate
- The cache is full, but we're only halfway done.
- The next access is x[8] : 001 00 000 and will map to set 0 again.
- How do we choose which to evict? We can do a simple “least-recently-used” policy.

Fully Associative Cache

- Fully associate cache: only 1 set. The associativity determines how many blocks can fit into the cache.
- If fully associative, the address is split into two components, tag and offset.
- Notice that by knowing how the address is split, we can determine block size and the number of sets, but we don't know anything about the associativity.

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?
 - Easy to know exactly where a block should belong and what block should be evicted
- What's the downside?

Associativity Tradeoffs

- What are the benefits of using a Direct Mapped Cache?
 - Easy to know exactly where a block should belong and what block should be evicted
- What's the downside?
 - Thrashing and the consequent poor performance when a cache is not used well

Associativity Tradeoffs

- What's the benefit of having increased associativity?

Associativity Tradeoffs

- What's the benefit of having increased associativity?
 - The problem of thrashing is reduced.
Potentially better use of the cache
- Downside?

Associativity Tradeoffs

- What's the benefit of having increased associativity?
 - The problem of thrashing is reduced.
Potentially better use of the cache
- Downside?
 - Hardware complexity increases.
 - In n-way associative, must check multiple entries to see if a block is present
 - In a fully associative cache, you have to search the entire cache!

Higher Order Parallelism

- With lower level parallelism (as in instruction level parallelism), we've reached something of a limit.
- That's not to say that you won't be able to squeeze more parallelism from the instructions.
- It's just that the work it takes to parallelize even further does not generally yield a proportional amount of performance benefit.

Task Level Parallelism

- Instead, we (more specifically, the book) turns towards task level parallelism which isn't about finding parallelism in the instructions of a single flow of instruction executions, but rather multitasking.
- Multi-tasking means breaking down a program into separate tasks that can be executed concurrently, ideally on different processors (if relevant).

Processes

- The most difficult thing about processes: describing them in words.
- A “process” encompasses “the act of executing a flow of instructions”.
- So an executable program isn't really a process, but *executing* a program is.
- The OS maintains a list of these executions and these are what are known as “processes”

Processes

- Think of processes as having a similar level of independence as programs.
- When you open a terminal to run a program, you expect it to run in isolation compared to a program running on another terminal.
- This is the primary expectation placed on processes that are running in parallel.

Processes

- When it comes to processes (as with programs that are running concurrently), there is no guarantee as to the order in which the programs will execute.
- This leads to a complication in parallel programming which is synchronization.
- Specifically, you must make sure that your program works correctly, regardless of the order in which everything is executed.
- But more on that later.

Processes

- Each process thinks that it owns everything and that everything revolves around it.
 - The entire addressable memory space
 - [0x000000000000, 0xFFFFFFFFFFFFFFF]
 - All registers
 - The CPU
- In many ways, this means that using processes is simple and safe.
- If I'm running process A (say Chrome), I don't have to worry about it getting in the way of process B that is also running (say Netscape Navigator)

Processes

- Consider two programs A and B.
- At memory address 0x10, program A is storing a variable “int a”.
- If B accesses memory address 0x10, of course it's not going to find A's “int a”.
- Under this restriction, it's essentially impossible for a malicious process to tamper with the memory of another process; it is so self-centered that it can't even comprehend the existence of another entity like itself (insert your own joke here).

Processes

- How to make a process:
 - `fork()`
- The `fork()` function will essentially clone the clone the existing program (memory and all) and after the point at which `fork` is called, two processes will be running.
- The original process that called `fork()` is considered the parent.
- The new process that was created is considered the child();

Processes

- After a process's creation the only difference between the parent and child processes is that the “child” will have 0 as the return value of `fork()`.
- The parent's return value for `fork` will be the pid of the child process
- Now previously, I said that a processes have the same level of autonomy as Chrome and Netscape Navigator.
- However, there is still a hierarchy to the processes that allow for some interaction.

Processes

- A process that creates another process (via fork) is considered the “parent”.
- In general, the parents have some level of control over the children processes.
- All processes maintain this hierarchy.
- A process can have an unbounded number of children but only one parent.

Processes

```
int main()
{
    int dummy = 0;
    if(fork() == 0)
    {
        dummy = 1;
    }
    else
    {
        dummy = 0;
    }
}
```

- When fork() is reached the child's value for fork() is 0
- Thus, the child will execute dummy = 1.
- Meanwhile, the parent's return value for fork is non-zero, therefore it will execute dummy = 0.

Processes

```
int main()
{
    int dummy = 0;
    if(fork() == 0)
    {
        dummy = 1;
    }
    else
    {
        dummy = 0;
    }
}
```

- In both instances the “dummy” variable has the same address in memory.
- However, there is no race condition or possible issue because they have the same addresses but are located in different address spaces.

Processes

- How can they both have totally distinct address spaces?
- Think of `fork()` as having cloned the existing program and now you're running the original program and its clone. These two running processes are as distinct as Chrome and Netscape Navigator.
- So how is it that your computer can run multiple processes/programs simultaneously when each program hogs the entire CPU?

Processes

- Well, if you have multiple cores/CPU's, you can run different processes on each core.
- However, you're probably running more processes than you have CPU's on your computer.
- Essentially, what invariably has to happen is that the processes have to take turn sharing the CPU.

Processes

- This is done via:
 - Operating System Scheduling
 - Context Switches
- When it comes to Processes (and Threads), the OS is the king. It decides which process gets the processor and which one doesn't; it decides who lives and who dies.
- For instance, if Process A has been running for 10 ms. The CPU may be taken away from Process A and given it to Process B if it pleases the OS.

Processes

- This means that once multiple processes are launched, there is no guarantee as to the order in which the processes are executed.
- They are at the mercy of the almighty OS.
- But that's usually okay. You don't usually worry if Netscape Navigator is a few instructions ahead of Chrome.
- The OS will pick and choose whichever to execute.

Processes

- In order to do this, the OS has the responsibility of maintaining the state of each running process, also known as a “context”.
- The “context” of a process essentially consists of all of the stuff that the process believes it owns:
 - Values of registers
 - The stack
 - Page table
 - File table
 - Essentially, the process's identity.

Processes

- When the OS decides that Process A no longer amuses it and would like to perform a context switch to Process B it:
 - Saves Process A's current state/context (ie registers, stack, etc.)
 - Restores Process B's state.
- This idea of taking turns is known as “concurrency”. This is in contrast to parallelism, where things are actually performed simultaneously.
- Am I deliberately missing something? Is this all that's necessary for a process to run correctly?

Processes

- What about memory? Surely the OS doesn't copy/back-up the process's entire memory space?
- Where would it, even?
- My 64-bit computer's RAM is 4.00 GB and my HDD is 370 GB.
- A 48-bit, byte-addressable space is 256 TiB. And that's just one process.
- Stay tuned. The answer to this mystery... next week on “CS 33”!

Processes

- Gee processes sure are swell.
- Because they're isolated and protected, they're simple to manage.
- You can run as many processes as you want and you don't have to worry about them messing each other up.
- However, are there situations in which processes are not ideal?

Processes

- The very thing that makes processes simple to use (isolation) is the thing that makes them inconvenient sometimes.
- What if you specifically want two processes to talk to each other or work together? You have to jump through hoops to get that to happen. For example, what if you just wanted two processes to share one measly int? Processes are completely blind to other process' memory spaces.
- The hoop: special OS managed inter-process communication (IPC)
- Additionally...

Processes

- Process Context Switching is an extremely expensive operation. Because Processes are so self contained, there is a lot that needs to be swapped over in a context switch.
- For these two reasons, if you want to do a computation where many executions are coordinating, processes may be too cumbersome and too slow.
- We need something lighter, faster, more dangerous...

Threads

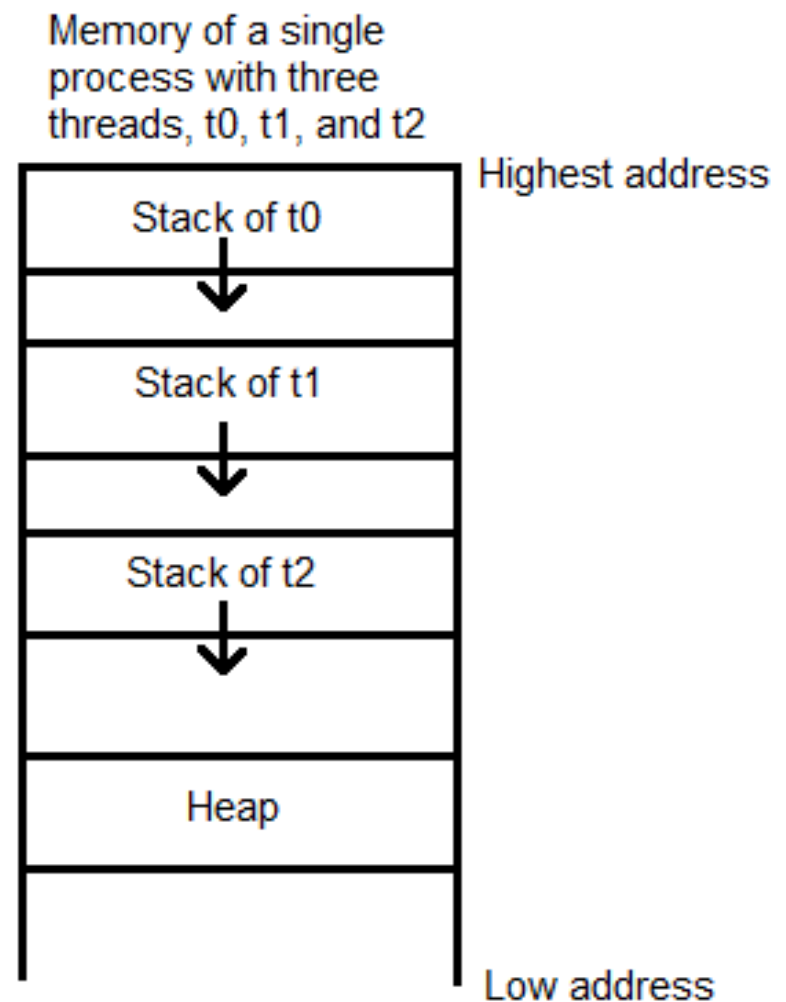
- Enter: Threads
- Threads are Process' leaner, lightweight siblings.
- Like processes, each thread runs its own distinct code in parallel.
- However, unlike processes, threads can easily acknowledge the existence of other threads.
- This is because threads share a single process and thus they share the resources of the process that they exist on.

Threads

- Each thread shares with all other threads on the process the entire addressable memory space
 - [0x000000000000, 0xFFFFFFFFFFFF]
- Each thread has its own space reserved in memory for its own stack.
- However, each thread also believes that it is the sole owner of:
 - CPU
 - All of the registers.

Threads

- Unlike with multiple processes, multiple threads on the same process share the same memory space. However, in an effort to allow for distinct execution, they each have space reserved in memory for their own stacks.



Threads

- Knowing this, what needs to be switched in a thread context switch?

Threads

- Knowing this, what needs to be switched in a thread context switch?
 - Pretty much just the registers.
 - The registers include %rip and %rsp, which means switching registers will account for the different instructions that threads will execute as well as the different stacks that each thread has.

Threads

- Threads have their own stack, but memory is shared. That means that among two threads running on the same process, the address 0x10 WILL point to the same thing.
- This is a major distinction from processes and can both be a major convenience or problem.
- A thread know where its own stack begins and ends, but there's nothing stopping it from accessing another thread's stack.

Threads

- As before, the thread's execution is at the mercy of the OS.
- However, this could present a problem.
- Generally when you use threads, you want them to work together in some way.
- You can't assume any particular order... unless you use special functions to manually maintain order.
- Another job for synchronization.

Threads

- How to create a thread:
- `int pthread_create(pthread_t *tid, pthread_attr_t const *attr, func *f, void *args)`
 - The argument `tid` is a thread id that is a pointer to a `pthread_t` passed in and assigned when the thread is created. A `pthread_t` is essentially a number
 - `attr` specifies options. By default, 0.
 - This creates a thread, assigns the thread id to `tid`. When the thread starts, it will call `f(args)`

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, 0, thread_func, 0);
    pthread_create(&tid2, 0, thread_func, 0);
    pthread_join(tid1, 0);
    pthread_join(tid2, 0);
}
```

- In this example, the main thread will launch two threads with id's saved to tid1 and tid2. Each will run the function “thread_func()” and once they complete, each thread will terminate.
- Using pthread_join, the main thread will wait until tid1 and tid2 complete before finishing (more on that later).

Threads

```
#include <stdio.h>
#include <unistd.h>
```

```
int glob = 10;
```

```
void* proc_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    fork();
    proc_func();
}
```

- This is a “Process” version of the same sort of code. This code will launch a process. Both the parent and child processes will call `proc_func()`.
- What will this print out?

Threads

```
#include <stdio.h>
#include <unistd.h>
```

```
int glob = 10;
```

```
void* proc_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    fork();
    proc_func();
}
```

- Because these are processes, when you call fork, you clone the process and create a new process with a cloned memory space.
- Thus, they will both have “glob”, but they will be accessing different versions of glob.

- Output:

11

11

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, 0, thread_func, 0);
    pthread_create(&tid2, 0, thread_func, 0);
    pthread_join(tid1, 0);
    pthread_join(tid2, 0);
}
```

- What about the thread version?
- What will this print out?

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- There are two threads, t0 and t1 running this code.
- Since these are threads, they share the code and memory. Thus they both have a “glob” and this variable refers to the same variable.
- However, recall that there is no guarantee of the ordering

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- As a result, if the order of instructions was:
 - 1.t0: glob += 1
 - 2.t0: printf
 - 3.t1: glob += 1
 - 4.t1: printf
- The output would be:
11
12

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- However, if it was:
 1. t0: glob += 1
 2. t1: glob += 1
 3. t0: printf
 4. t1: printf
- The output would be:
12
12

Threads

```
#include <stdio.h>
#include <pthread.h>
```

```
int glob = 10;
```

```
void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Could it ever be:

12

11

or

11

11

?

Threads

```
#include <stdio.h>
#include <pthread.h>

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Surely not, right?
- We know from out-of-order processing that the order of the instructions is maintained even if execution doesn't strictly follow that ordering.
- Therefore, if a thread executes “printf”, it must have executed “glob += 1”. Thus:
11
11
- ...doesn't seem possible since when the second thread to reach printf calls printf, both threads must have called glob+=1 (if the first thread already reached printf)

Threads

```
#include <stdio.h>
#include <pthread.h>
```

```
int glob = 10;
```

```
void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Same with:
12
11
- If a thread prints out 12, that means both threads have already executed `glob += 1`
- Thus, if the first thread prints 12, then the next thread must print 12 since it's already executed `glob += 1`.
- Case closed. Right?

Threads

- For one thing, it's a fallacy to think in terms of the C functions.
- The ordering of instructions is a component of the assembly level.
- So what might the assembly code for `thread_func` look like?

Threads

- Assume a simplified version of printf that takes an int argument to print in %edi.

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```

Threads

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```

- If the order of execution is:
 1. t0: insn 1
 2. t1: insn 1
 3. t0: insn 2
 4. t1: insn 2
 5. t0: insn 3
 6. t1: insn 3
 7. t0: insn 4
 8. t1: insn 4
- Because both threads would load glob before either has committed a result to memory, the output would be:
11
11

Threads

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```

- If the order of execution is:
 1. t0: insn 1
 2. t0: insn 2
 3. t0: insn 3
 4. t1: insn 1
 5. t1: insn 2
 6. t1: insn 3
 7. t1: insn 4
 8. t0: insn 4
- t0 increments glob and stores it so glob = 11. Then, t1 runs in its entirety, incrementing glob and printing out 12. Then, t0 prints the value of glob that it had stored in %edi, which was 11.
- Thus:

12
11

Threads

- This was one of the simplest examples of shared memory, and its behavior could be completely wrong, simply because there's no guarantee of the order in which the assembly instructions would be executed.
- In order to properly use threads, we need synchronization.

End of
The Seventh Week
-Three Weeks Remain-