Introduction to Computer Organization

DIS 1A – Week 6

Slides modified from Eric Kim

Agenda

- Program Performance
- Memory Hierarchy
- Cache
- Stack Exploits
- Attack Lab

Program Performance

- So far, we have reasoned about code performance at a high-level
  - Example: binary search of a sorted array of length N has O(log N) behavior

In this class: reason at the compiler-level *and* processor-level!

Program Performance

- Processor-level considerations
  - Instruction pipelining
  - Exploiting parallelism
  - Out-of-Order execution (OoO)
- Skill: How to write C code to "encourage" compiler to generate assembly code that fully-utilizes processor?

Program Example

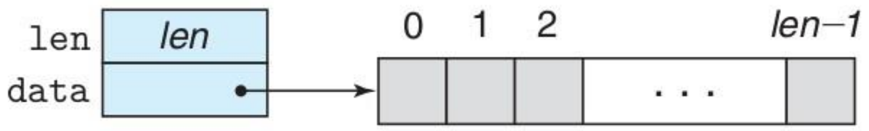


Figure 5.3 **Vector abstract data type.** A vector is represented by header information plus array of designated length.

code/opt/vec.h

```
/* Create abstract data type for vector */
typedef struct {
    long int len;
    data_t *data;
} vec_rec, *vec_ptr;
```

code/opt/vec.h

data_t can be an int, float, or double

Program Example

```c
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Figure 5.5 **Initial implementation of combining operation.** Using different declarations of identity element *IDENT* and combining operation *OP*, we can measure the routine for different operations.

IDENT is either 0 (add), or 1 (mult).
OP is either + or *.

Loop Unrolling

- <u>Reason 1</u>*:* Less overhead due to loop bookkeeping (ie "i<n", "i++).
- <u>Reason 2</u>: Exposes structure in code, allowing compiler to perform additional optimizations

Loop Unrolling

```
void combine5(vec_ptr v, data_t *dest) {
  long int i; long int length =
  vec_length(v);  long int limit = length-1;
  data_t *data =
  get_vec_start(v);  data_t acc =
  IDENT;
  /* Combine 2 elements at a time
  */  for (i = 0; i < limit; i+=2)
    acc = (acc OP data[i]) OP data[i+1];
  /* Finish any remaining elements
  */  for (; i < length; i++)
    acc = acc OP data[i];
  *dest = acc;
}
```

Can unroll loop further (ie k > 2)

Loop Unrolling

- Speedup is due to reduced overhead relating to loop maintenance/bookkeeping

Multiple Accumulators

- Modern processors have fully pipelined add/mult
- Critical bottleneck in combine: we have to write to acc after each mult.
    - Why is this a problem?

      *Constrains each mult to have to occur **sequentially**.*
    - How can we overcome?

      *Remove this constraint! Write to **separate** accumulators.*

Multiple Accumulators

```
/* Unroll loop by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Instruction Level Parallelism

- The optimizations presented 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 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 to do 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

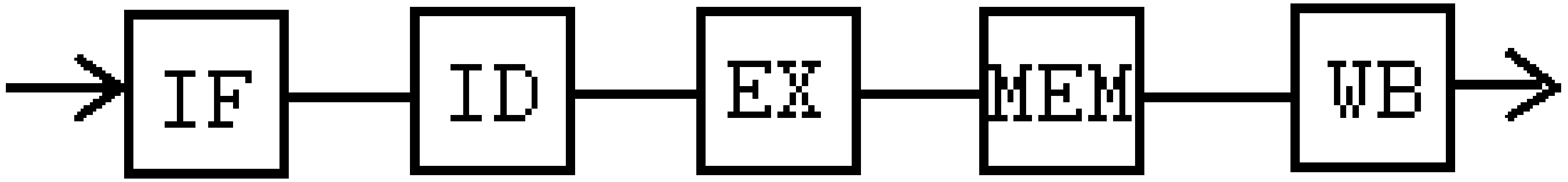
- Power Wall:
  - As the clock rate increases, the power needed to operate everything increases. In addition to requiring a lot of power, much of it is leaked/dissipated as heat.
- Memory Wall:
  - The processor speeds have increased significantly since the 80's.
  - Also recall the RAM access time has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

Instruction Level Parallelism

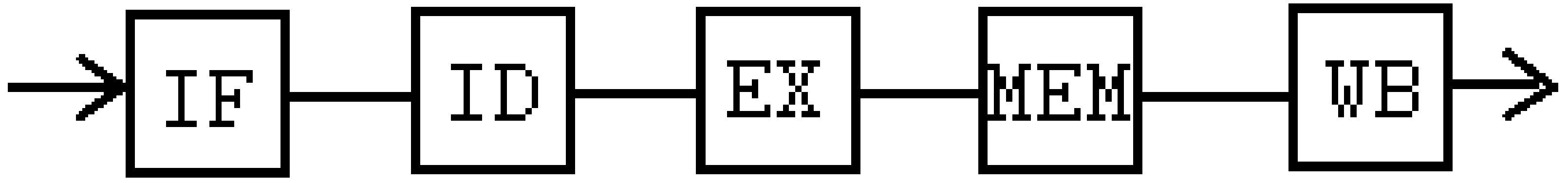
- Memory Wall:
  - The bottleneck is still in RAM access and it is the thing we'd need to reduce in order to get a significant improvement.

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.

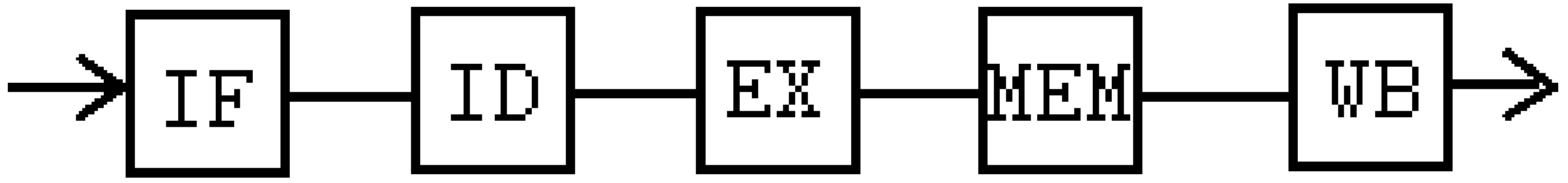


Instruction Level Parallelism



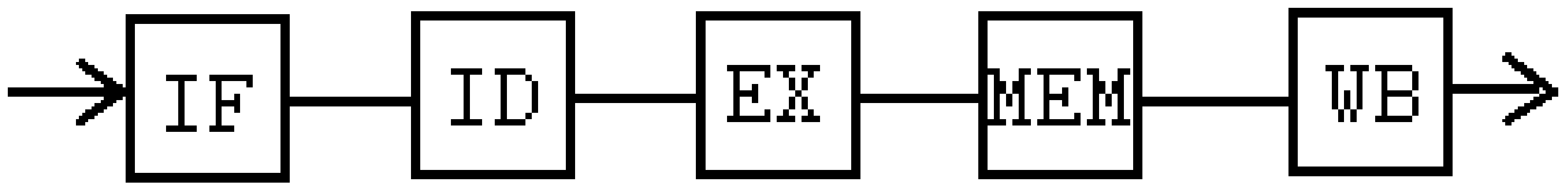
- Each stage corresponds to a physical component in the processor.
- IF (Instruction Fetch): Takes as input the address of the instruction (%rip). Then it finds the actual instruction in memory.
- ID (Instruction Decode/Register Read): Given the instruction bytes, get the appropriate values from memory.

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



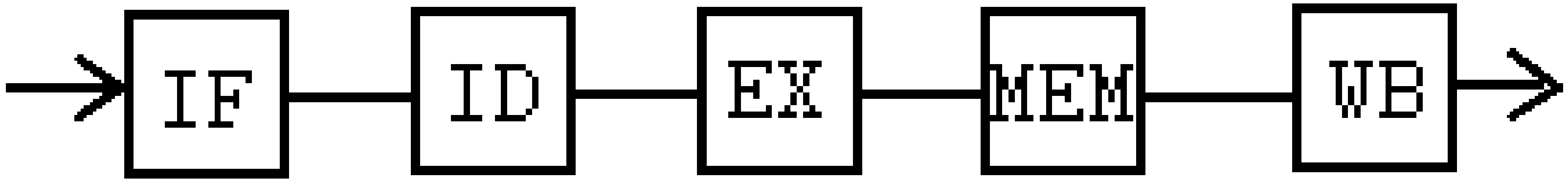
- Assume that each block takes a single cycle to execute. A single instruction would take 5 cycles to complete.
- The latency of each instruction is 5 cycles.
- Our throughput of this system is 1 instruction per every 5 cycles.

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.

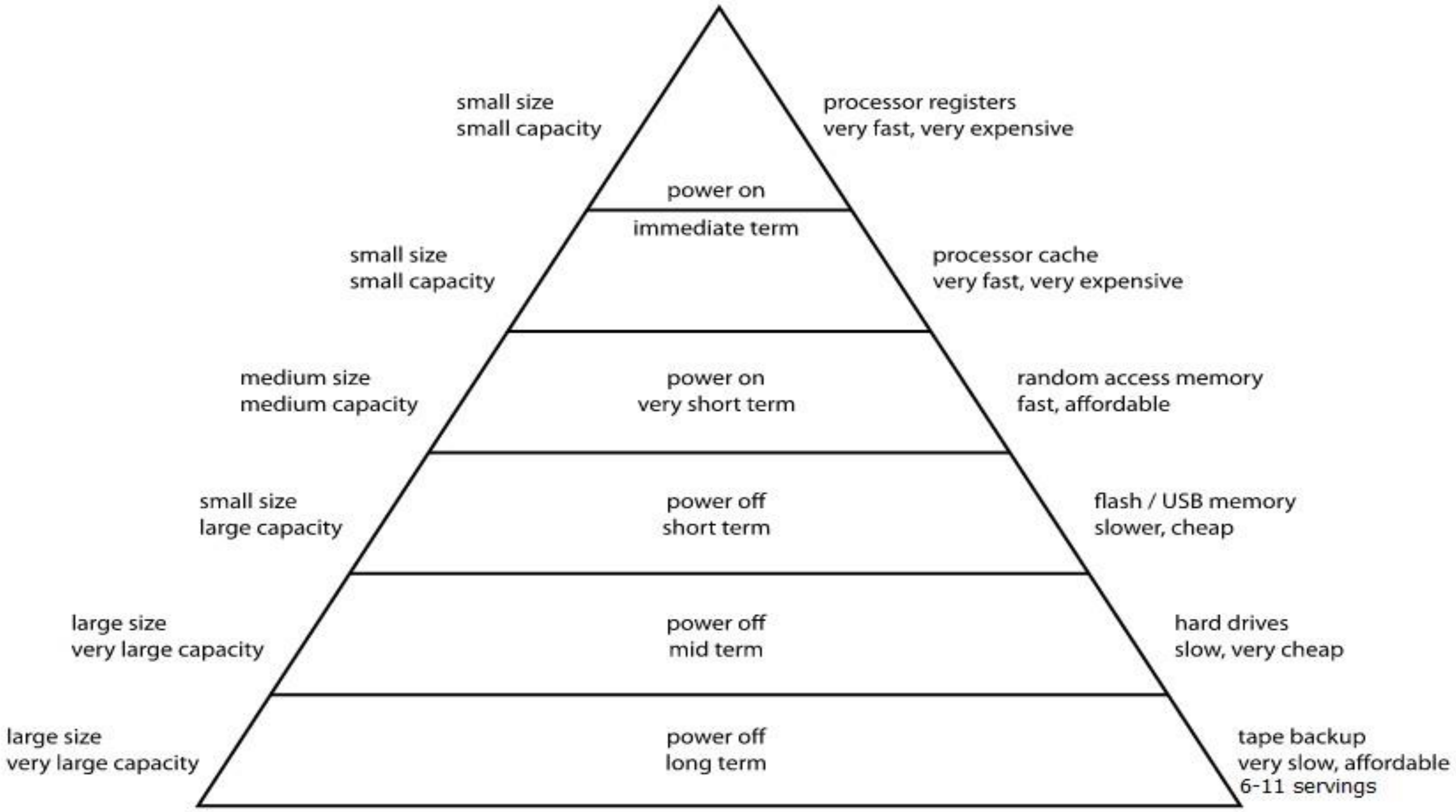
Instruction Level Parallelism

- 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.

Branch misprediction

- Modern processors employ *speculative execution* to fully utilize CPU for branches
  - Fancy term for: guess which branch to take
- If we guess wrong, then we need to **undo** the instructions we executed!
  - Flush the pipeline, and start again at mispredicted instruction

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.
- 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

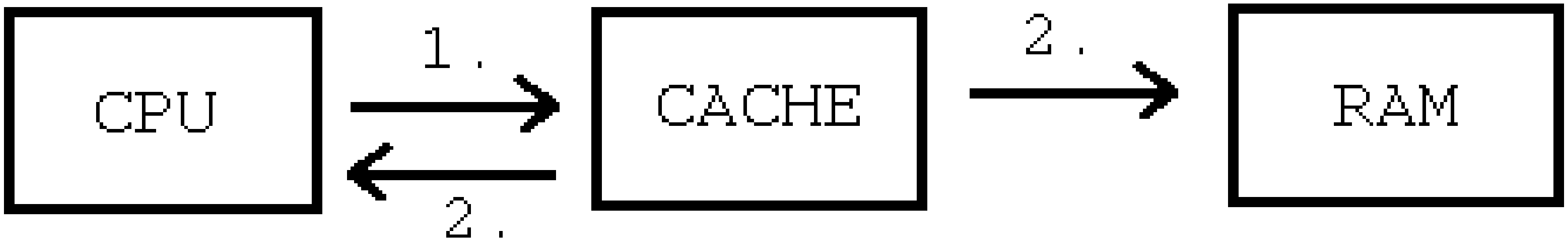
- As a result, we introduce a layer of memory that is hidden from assembly/machine instructions.
- This layer operates on the same principle:
  - When you need to operate on data, copy it from DRAM into a faster (but smaller) memory.
  - Once you need it to be committed to actual main memory, copy it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

Caching

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

Caching

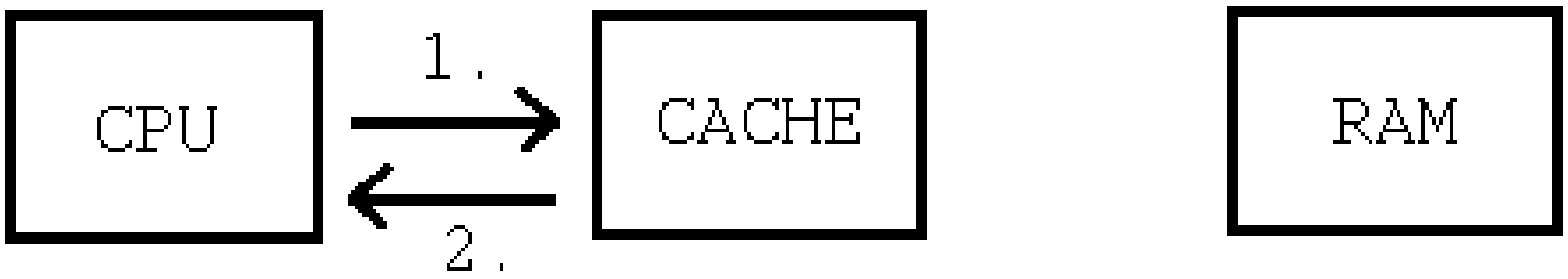
- 1.Issue the address to the cache. Does the cache contain the data at address 0x10?
- 2.If the cache has it, then give the data back to the CPU. Else fetch it from RAM



- Implicitly, EVERY memory access goes through the cache.

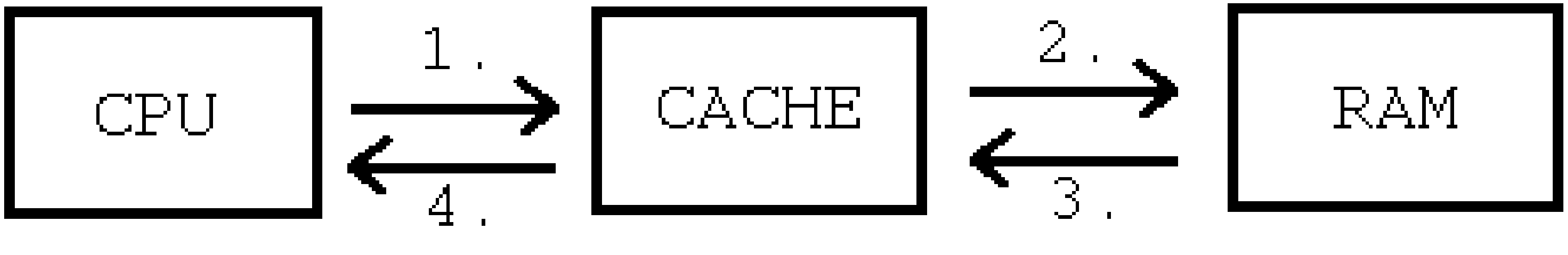
Caching

- 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

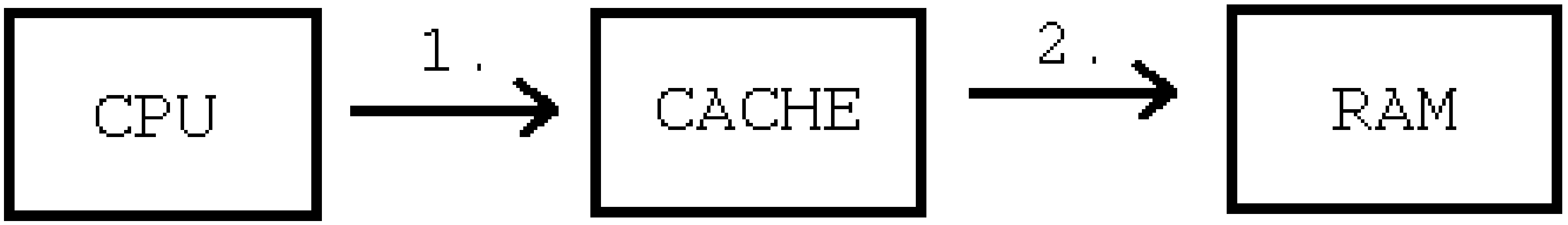
- How much data is brought into the cache at any given access is a function of the cache itself and the defined “cache block” size, which is the granularity in which cache operations are dealt with.

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

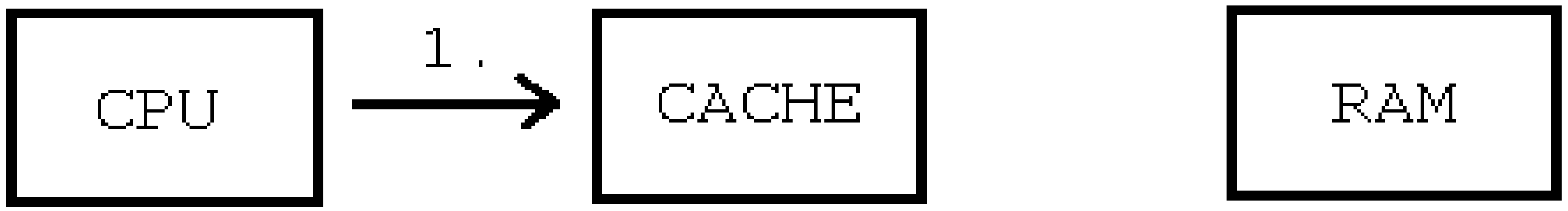
Caching

- 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



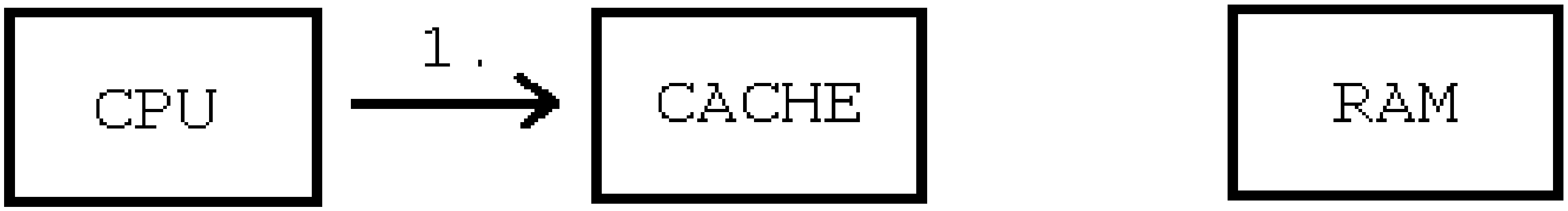
Caching

- 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-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.

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.

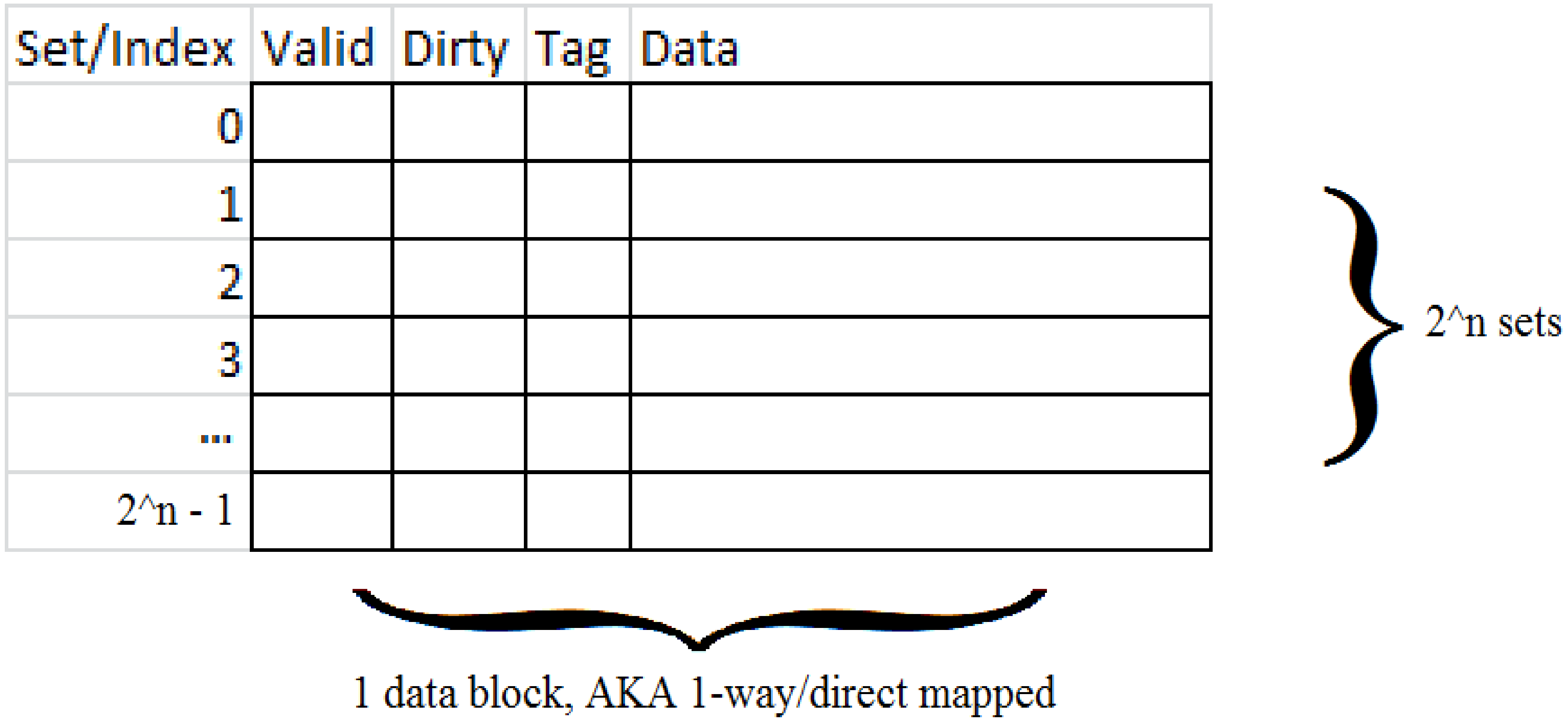
Cache

- 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.

Direct Mapped Cache



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

- 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

- 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]
- XXXX XXXX XXXXX
- ...you're dealing with the block that starts at:
- XXXX XXXX 00000

Direct Mapped Cache

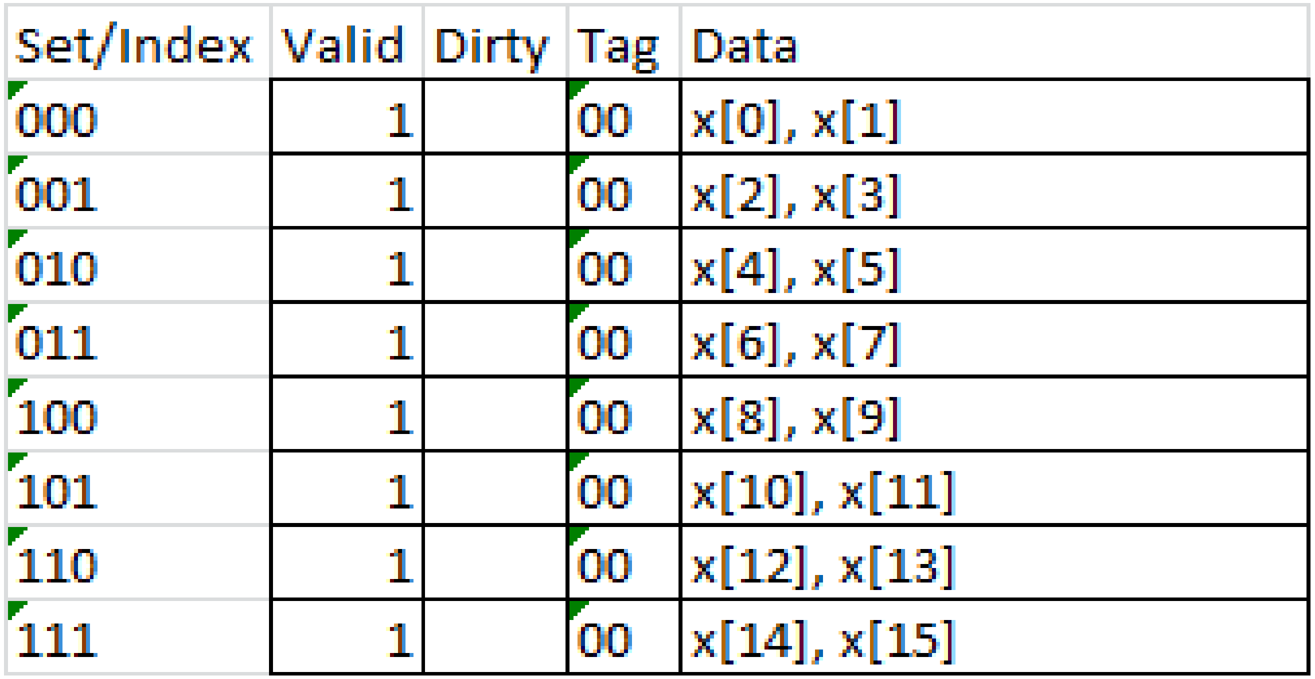
- Access order:
- &x[0] = 00 000 000 : Set 0, cold miss, bring in x[0] and
- x[1] &x[1] = 00 000 100 : Set 0, hit.
- &x[2] = 00 001 000 : Set 1, cold miss, bring in x[2] and
- x[3] &x[3] = 00 001 100 : ...
- …
- &x[15]= 00 111 100

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.

Stack Exploits

- Consider a server that hosts publicly accessible server code that performs tasks A, B, and C and an attacker that has access to the server and wants to cause trouble.
- It wouldn't be terribly impressive (or convincing) if the attacker claimed that he/she took over the server and coerced it to perform tasks A, B, or C.
- However, if the attacker could get the code to perform task D (something the was not intended to be performed by the server)...

Stack Exploits

- To do this, an attacker would want to get the server code to execute code of the attacker's choice.
- The attacker cannot directly influence the operations of the CPU or swap out the code contents in memory. It'd be pretty much game over if they could.
- However, very commonly, a user of a program will have the power to provide input to the server, which can affect on the stack.

Stack Exploits

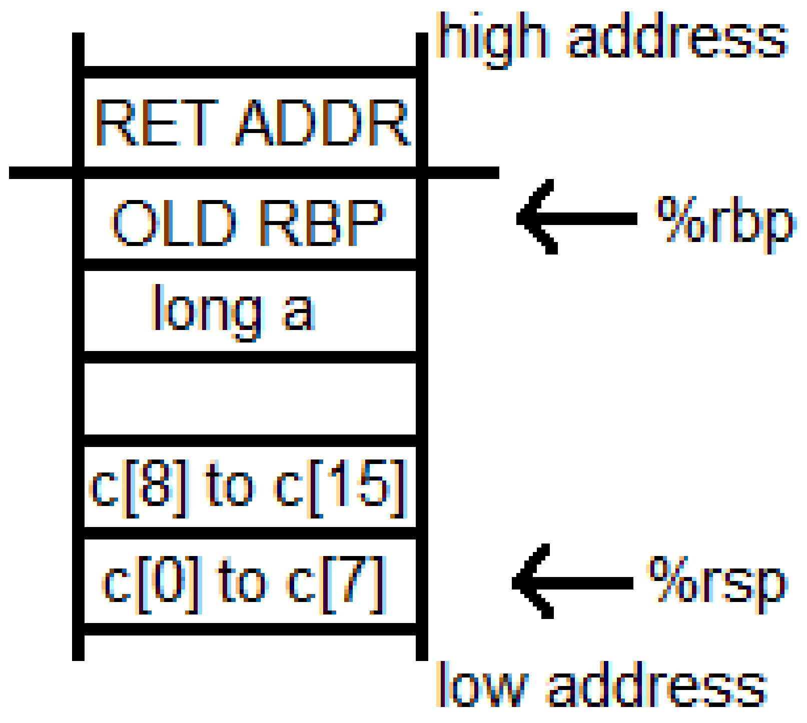
- Consider the following function:

```
int terrible()
{
  long a = 0x7766554433221100;
  char c[16];
  gets(c);
}
```

- When compiled, the stack looked like this:

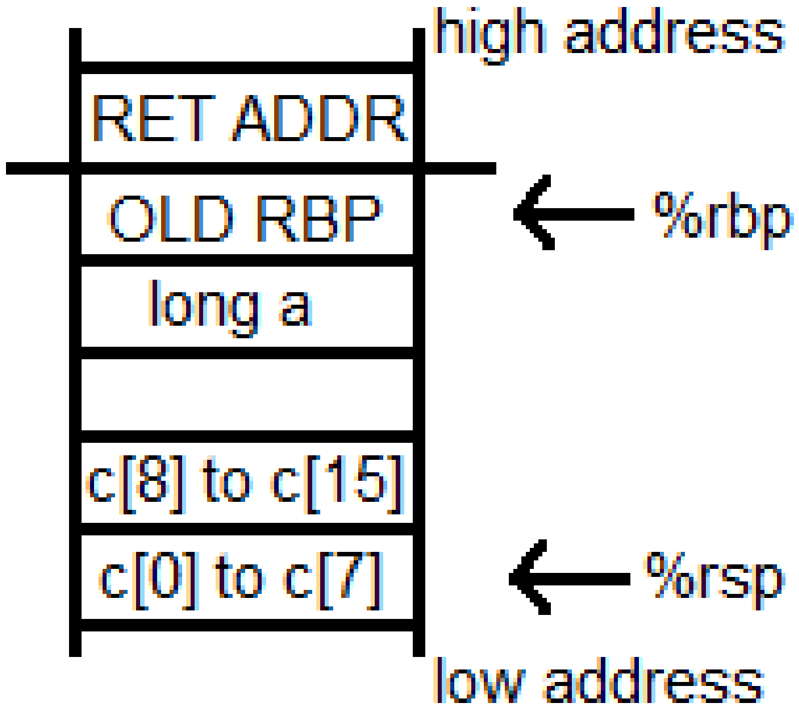
Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?



Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?
  - Alignment purposes. Recall that %rsp must be 16-byte aligned. The assumption is that %rsp was aligned when calling this function.
- The code looks like:



Stack Exploits

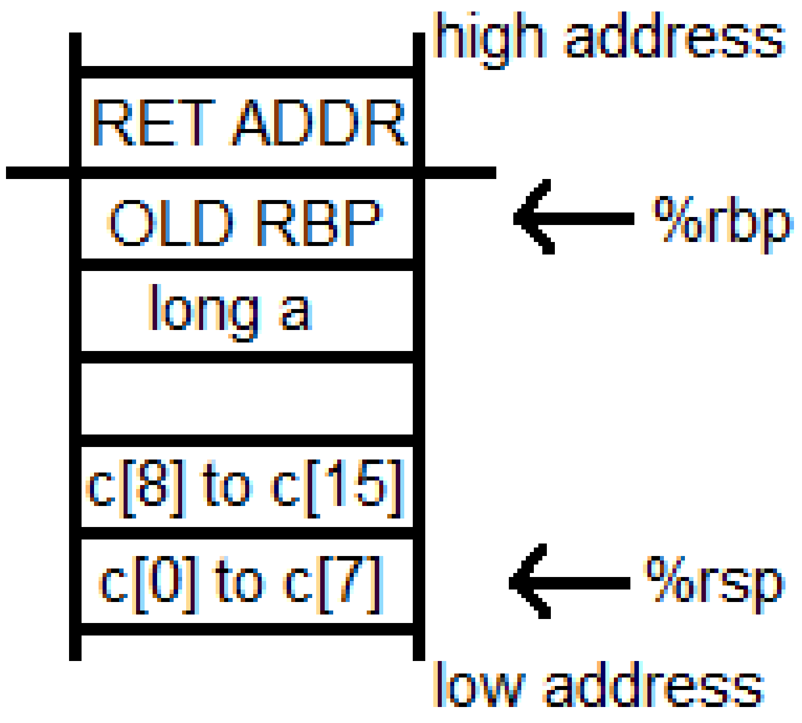
```
Dump of assembler code for function blah (gcc with
no options):
 0x400528 <+0>:     push   %rbp
 0x400529 <+1>:     mov    %rsp,%rbp
 0x40052c <+4>:     sub    $0x20,%rsp
 0x400530 <+8>:     movabs $0x7766554433221100,%rax
 0x40053a <+18>:    mov    %rax,-0x8(%rbp)
 0x40053e <+22>:    lea    -0x20(%rbp),%rax
 0x400542 <+26>:    mov    %rax,%rdi
 0x400545 <+29>:    callq  0x4003b0 <gets@plt>
 0x40054a <+34>:    leaveq
 0x40054b <+35>:    retq
End of assembler dump.
```

Stack Exploits

- The “gets” function takes as an argument, a character pointer. Then, it asks the user to input a character string where it then copies the string into the specified character pointer.
- For example, if you called “get(c)” and the user input provided “ABCD”, then the bytes for “ABCD” would be copied from *c to *c + 3.
- The pointer in this example is -0x20(%rbp) or -32(%rbp).

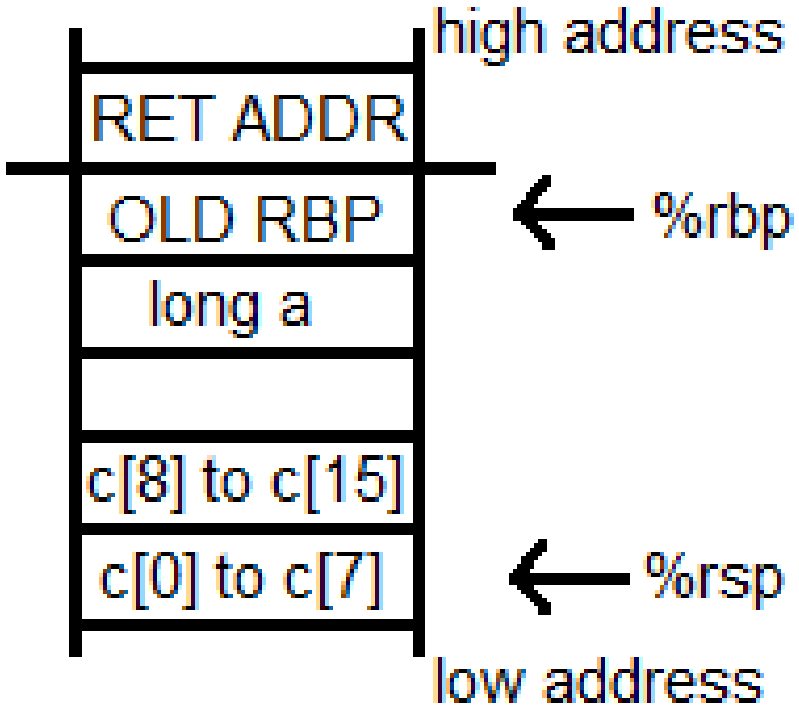
Stack Exploits

- If the user typed “jonathan” (8 characters), these 8 bytes would span from *c to *c + 7
- If the user typed “mr_super_long_name_the _third_esquire”, this would be copied from *c to *c + 35
- However, in the C code, we only specified a character array of size 16.
- “gets” don't care



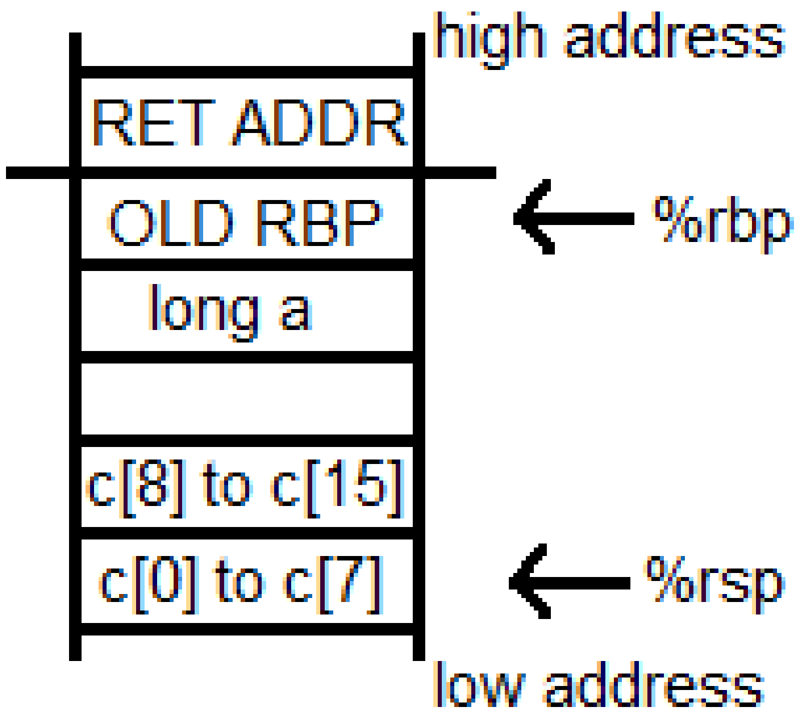
Stack Exploits

- As a result, the user can provide a string to arbitrarily write to *c to *c + x.
- This means, the user could also overwrite long a, the old rbp, and most worryingly, the return address.
- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?



Stack Exploits

- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?
- At least 40 characters (which will wipe out long a and OLD RBP), and then: 0x00, 0x08, 0x40, 0x00 … (assume little endian)
- Suddenly, once the function returns, it will attempt to execute something that it did not intend to.



Stack Exploits: Injecting Code

- Knowing that we are no longer able to rely on existing code, we will somehow need to supply our own.
- Then, we can jump to our own code when we return.
- How exactly do we specify code? It's not exactly like the program is going to be asking us for code to inject. Don't be ridiculous.

Stack Exploits: Injecting Code

- Actually... it sort of it
- “gets” gives us direct access to writing into memory; whatever string of bytes is provided as an input is saved into memory/the stack.
- Sure it's not the code segment, but we can provide a string that contains the code that we want to execute,
- Then in the return address, write the address of the code that we provided.

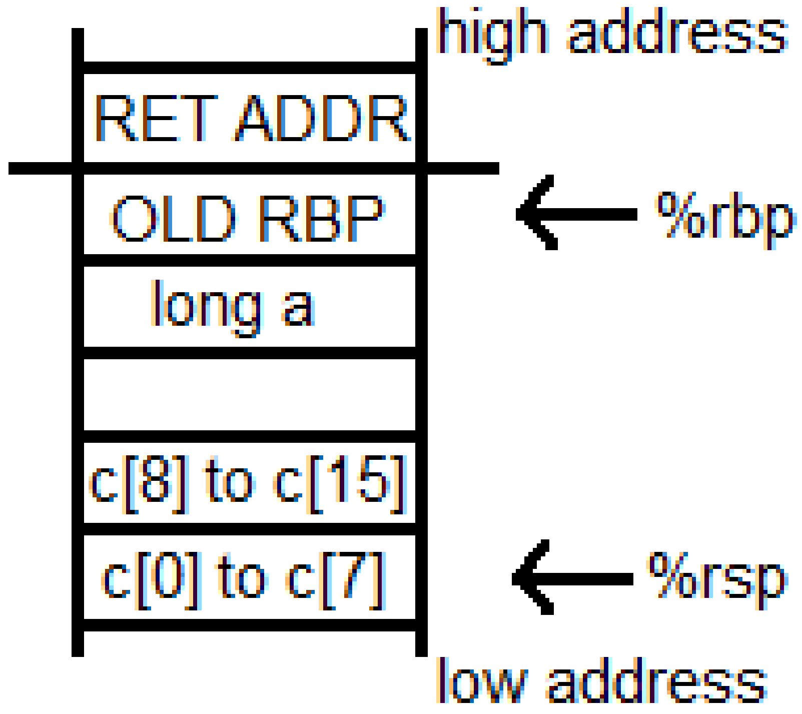
Stack Exploits: Injecting Code

- Let's say as our vicious attack to take over the server is to execute the following instructions:

```
48 89 f8      mov    %rdi,%rax
48 83 c0 01   add    $0x1,%rax
```

- Go big or go home.
- Consider the previous “gets” example:

Stack Exploits: Injecting Code



- The function gets(c) is called.
- Say %rsp = 0x7fffffffe1d0.
- Then the memory before calling “gets” looks like this:

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

```
0x7fffffffe1d0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1e0: 0x00400570  0x00000000  0x33221100  0x77665544
0x7fffffffe1f0: 0xffffe200  0x00007fff  0x00400560  0x00000000
```

- First, some notes about this little endian GDB format.

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

```
0x7fffffffe1d0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1e0: 0x00400570  0x00000000  0x33221100  0x77665544
0x7fffffffe1f0: 0xffffe200  0x00007fff  0x00400560  0x00000000
```

- Each column is a 4-byte word.
- 0x00400570 is the 4-byte word at address 0x7fffffffe1e0.
- 0x00000000 is the word at address 0x7fffffffe1e4
- 0x33221100 is the word at address 0x7fffffffe1e8
- 0x77665544 is the word at address 0x7fffffffe1ec

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

```
0x7fffffffe1d0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1e0: 0x00400570  0x00000000  0x33221100  0x77665544
0x7fffffffe1f0: 0xffffe200  0x00007fff  0x00400560  0x00000000
```

- Each word is presented as the actual value. In a little endian machine.
- 0x70 is the byte at address 0x7fffffffe1e0.
- 0x05 is the byte at address 0x7fffffffe1e1
- 0x40 is the byte at address 0x7fffffffe1e2
- 0x00 is the byte at address 0x7fffffffe1e3

Stack Exploits: Injecting Code

- `(gdb) x/40xw $rsp`
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

```
0x7fffffffe1d0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1e0: 0x00400570  0x00000000  0x33221100  0x77665544
0x7fffffffe1f0: 0xffffe200  0x00007fff  0x00400560  0x00000000
```

- Remember, we want to execute:

```
   48 89 f8      mov    %rdi,%rax
   48 83 c0 01   add    $0x1,%rax
```

- Because of gets(c), we have the power to write bytes into the 'c' buffer.

Stack Exploits: Injecting Code

```
Dump of assembler code for function blah (gcc with
no options):
 0x400528 <+0>:     push   %rbp
 0x400529 <+1>:     mov    %rsp,%rbp
 0x40052c <+4>:     sub    $0x20,%rsp
 0x400530 <+8>:     movabs $0x7766554433221100,%rax
 0x40053a <+18>:    mov    %rax,-0x8(%rbp)
 0x40053e <+22>:    lea    -0x20(%rbp),%rax
 0x400542 <+26>:    mov    %rax,%rdi
 0x400545 <+29>:    callq  0x4003b0 <gets@plt>
 0x40054a <+34>:    leaveq
 0x40054b <+35>:    retq
End of assembler dump.
```

Stack Exploits: Injecting Code

- We need to do two things:
  - Inject the code.
  - Overwrite the return address to point to the injected code.
- One way of doing this is to inject the code with “gets” at the beginning of “c” and then have the return address point to address of buffer “c”.

Stack Exploits: Protection From

- How do we handle/prevent such an exploit?
- There are two aspects that allow an exploit like this:
    1. The ability to write beyond an allocated buffer (ie, the buffer overrun) to overwrite return addresses.
    2. The ability to execute code of the attacker's specification.
- We can try to protect our system by preventing both aspects.

Stack Exploits: Protection From

- How do you protect against:
  - The ability to write beyond an allocated buffer (ie, the buffer overrun)

Stack Exploits: Protection From

- Method 1: Don't use C.
- The issue of overrunning a buffer is archaic and well-documented. Modern languages automate and abstract away many of C's low level behavior such as pointer manipulation and maintaining proper array access.
- Modern languages tend to maintain the length of buffers and will crash if you attempt to access an area out of range.
- What's a weakness of this approach?

Stack Exploits: Protection From

- What's an issue of the “don't use C” approach?
- Depending on your preference of programming languages, this could be a blessing.
- However, there's a reason that C is still the predominant language for low level OS code. You have more direct control over memory, I/O, and etc. and more control over things that other languages don't trust the user with.
- C is fast.
- Please don't stop using C.

Stack Exploits: Protection From

- Method 2: Write better code. Use safer functions.
- If you try to compile code using “gets”, it's either unrecognized or you get this:
  - “warning: the `gets' function is dangerous and should not be used.”
  - The compiler just called a function “dangerous”. Time to sleep with one eye open.
- Use functions that allow you to specify a maximum length of bytes such as fgets() and read().
- If you write a program with no potential for buffer overrun, problem solved.

Stack Exploits: Return Oriented Programming

- The bytes that comprise the program's code can be interpreted differently from their original intent.
- For example, recall that ret has byte code 'C3'
- Say we have the code snippet:

```
   0x...fe8 <+120>:   8d 04 39          lea    (%rcx,%rdi,1),%eax
   0x...feb <+123>:   48 39 c3          cmp    %rax,%rbx
   0x...fee <+126>:   0f 82 c4 00 00 00       jb
0x7ffff7dee0b8 <add_to_global+328>
```

Stack Exploits: Return Oriented Programming

- Say we have the code snippet:

```
   0x...fe8 <+120>:   8d 04 39          lea    (%rcx,%rdi,1),%eax
   0x...feb <+123>:   48 39 c3          cmp    %rax,%rbx
   0x...fee <+126>:   0f 82 c4 00 00 00       jb
0x7ffff7dee0b8 <add_to_global+328>
```

- Hey look, it's the return instruction at 0x...fed!
- If we interpreted 0x...fed as the starting point of an instruction, we would instead be calling the return function where there wasn't originally a ret.

Stack Exploits: Return Oriented Programming

- This idea is the crux of Return Oriented Programming.
- If we can't execute the stack, we're limited to the functions provided by the source code + linked libraries that are in the executable segments of memory.

Stack Exploits: Return Oriented Programming

- Find bytes in the code that form meaningful instructions that end in the ret instruction.
- These snippets will be our new “functions”.
- Then, write the addresses of the sequence of functions you want to execute to the stack.
- If you can find enough meaningful instructions to do what you want, you've just created arbitrary code out of a limited set of code.

Stack Exploits: Return Oriented Programming

- For example, pretend that:
  - The ret instruction is represented by byte “RR”
  - You want to execute the following code:
  - CC AA BB   movq %rdi, %rax
  - CC DD EE FF GG HH addq $1, %rax
  - No, this is not the mythical “octodecimal” representation. These are just fake placeholder bytes.
  - You find a snippet of the provided library code that does this:

Stack Exploits: Return Oriented Programming

```
...
0x4006af <+16>:    JJ EE             mov %eax,%ecx
0x4006b1 <+18>:    BB VV             sar %cl,%edx
0x4006b3 <+20>:    JJ QQ             mov %edx,%eax
0x4006b5 <+22>:    JJ RR TT          mov %eax,-0x4(%rbp)
0x4006b8 <+25>:    LL AA VV CC AA    mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG          sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB          mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD          mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR    mov $0xffffffff,%edx
0x4006cb <+44>:    JJ CC             mov %eax,%ecx
…
```

- Well, I don't see the instructions we're looking for...

Stack Exploits: Return Oriented Programming

```
...
0x4006af <+16>:    JJ EE              mov %eax,%ecx
0x4006b1 <+18>:    BB VV              sar %cl,%edx
0x4006b3 <+20>:    JJ QQ              mov %edx,%eax
0x4006b5 <+22>:    JJ RR TT           mov %eax,-0x4(%rbp)
0x4006b8 <+25>:    LL AA VV CC AA     mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG           sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB           mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD           mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR     mov $0xffffffff,%edx
0x4006cb <+44>:    JJ CC              mov %eax,%ecx
…
```

- Well, I don't see the instructions we're looking for...
- But this looks promising

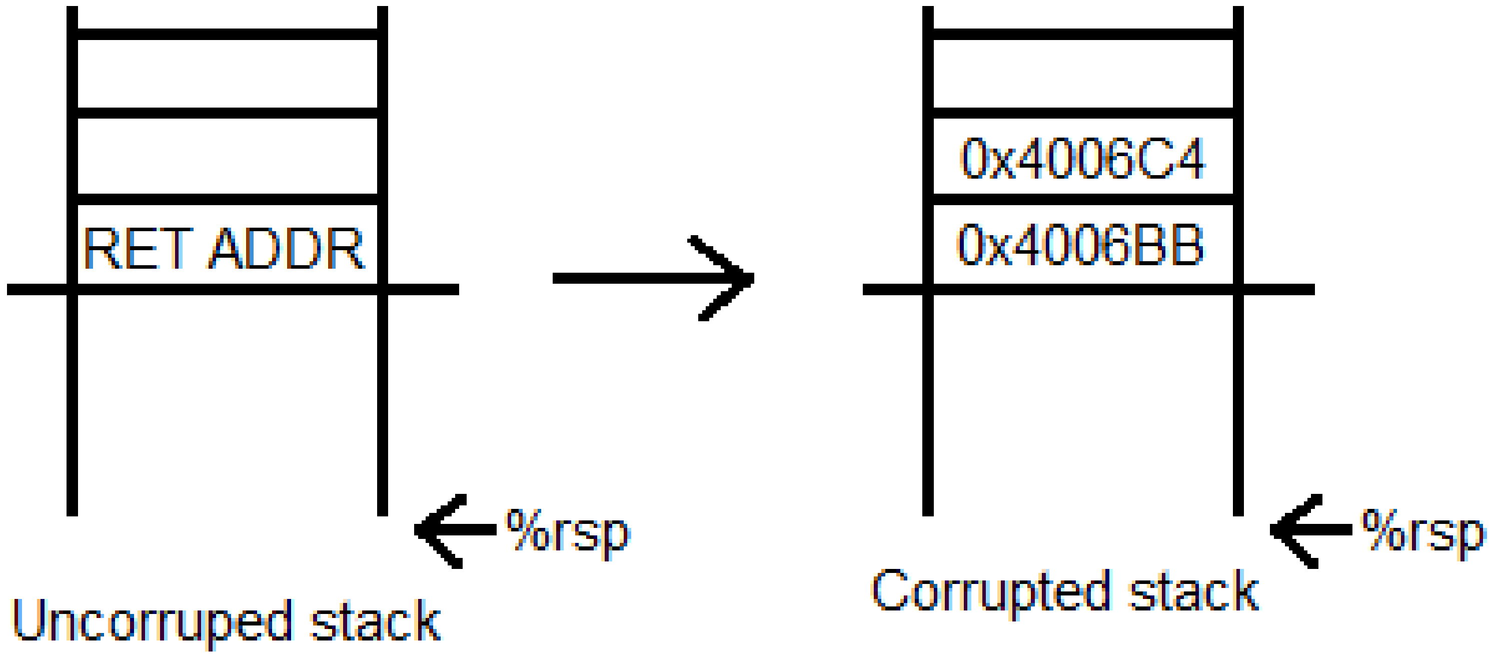
Stack Exploits: Return Oriented Programming

```
...
0x4006b8 <+25>:    LL AA VV CC AA  mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG        sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB        mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD        mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR  mov $0xffffffff,%edx
…
```

- If we treated 0x4006bb as the starting point for an instruction, we'd have:
  - `0x4006bb:    CC AA BB       movq %rdi, %rax`
  - `0x4006be:    RR             ret`
- If we treated 0x4006c4 as the starting point for an instruction, we'd have:
  - `0x4006c4:    CC DD EE FF GG HH   addq $1, %rax`
  - `0x4006ca:    RR                  ret`

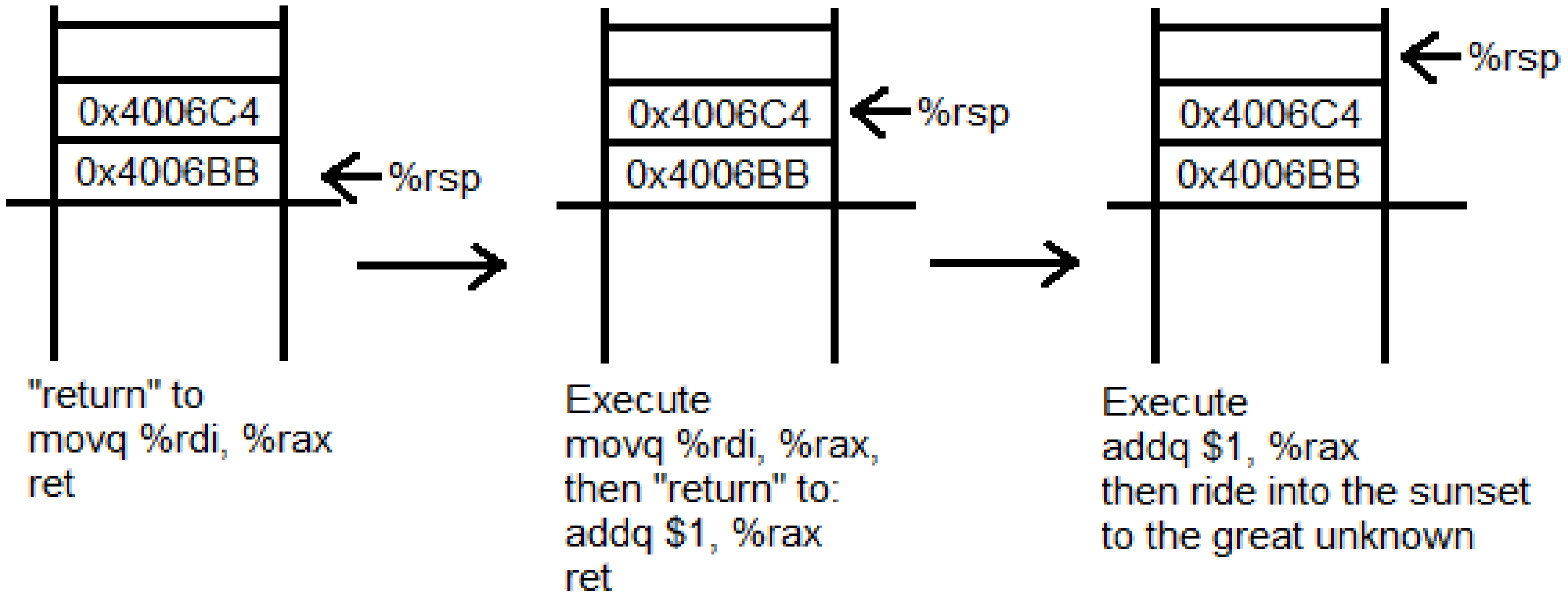
Stack Exploits: Return Oriented Programming

- Assuming we have a buffer overrun exploit to... exploit, we can prepare the stack like this:



- Now, when we return from the function, we jump to 0x4006BB

Stack Exploits: Return Oriented Programming



- Hence: “return oriented”.
- Your “code” now consists of the sequence of return addresses that you wrote to the stack.

Lab 3: Attack Lab

- hex2raw:
  - If your hexadecimal code is in hex.txt and you want to convert to a file called raw.txt, invoke the command as follows:

    "cat hex.txt | ./hex2raw > raw.txt"
  - cat will print out contents of hex.txt

    ...which will be “piped” in as the input to hex2raw.