x86-64 Integer Registers, 16 Registers

	64 bits	32 bits	16 bits	8 bits	
1st argument	%rdi	%edi	%di	%dl	Caller Saved
2nd argument	%rsi	%esi	%si	%sl	Caller Saved
3rd argument	%rdx	%edx	%dx	%dl	Caller Saved
4th argument	%rcx	%ecx	%cx	%cl	Caller Saved
5th argument	%r8	%r8d	%r8w	%r8b	Caller Saved
6th argument	%r9	%r9d	%r9w	%r9b	Caller Saved
Return value	%rax	%eax	%ax	%al	Caller Saved

C Declaration	`char`	`short`	`int`	`float`	`double`	`long int`	`char*`	`long double`
Size	1	2	4	4	8	8	8	10 / 12

Operand Types

Immediate	Register	Memory Reference
$Imm	%rax, %eax, …	Addr
$-536, $0x1F	R[r_b] for referenced value	M[Addr] for referenced value

Memory Addressing Modes

Absolute	Imm	Mem[Imm]
Indirect	(r_b)	Mem[R[r_b]]
Displacement, R[%rdi] start of memory region, Imm Offset	Imm(r_b)	Mem[R[r_b] + Imm]
Base Register + Index Register	(r_b, r_i)	Mem[R[r_b] + R[r_i]]
Base Register + Index Register + Offset	Imm(r_b, r_i)	Mem[R[r_b] + R[r_i] + Imm]
Scaled Index	(, r_i, s)	Mem[s * R[r_i]]
Scaled Index + Offset	Imm(, r_i, s)	Mem[s * R[r_i] + Imm]
Scaled Index	(r_b, r_i, s)	Mem[R[r_b] + s * R[r_i]]
Most General Form	Imm(r_b, r_i, s)	Mem[R[r_b] + s * R[r_i] + Imm]

Moving Data Instruction, *mov{x} Src, Dst, copy data from Src to Dst (Register / Memory Address)*
suffix has to match register *Src, Dst can't both refer to memory location*

movb	Move byte
movw	Move word (2 bytes)
movl	Move double word (4 bytes)
movq	Move quad word (8 bytes)
movabsq	Move absolute quad word

movq vs moveabsq

movq's immediate *src operand 32-bit two's complement number*, *sign-extended to 64 bits* for the destination.

movabsq's immediate *src operand 64-bit value*, but *Dst must be Register*.

```
movabsq $0x0011223344556677, %rax     %rax = 0011223344556677
movb    $-1, %al                       %rax = 00112233445566FF
movw    $-1, %ax                       %rax = 001122334455FFFF
movl    $-1, %eax                      %rax = 00000000FFFFFFFF
movq    $-1, %rax                      %rax = FFFFFFFFFFFFFFFF
```

Recall that 1 byte can represent 256 numbers, FF in Hexadecimal
When movl has Register as Dst, will set high-order 4 bytes of the Register to 0
For any operation that fills in 32 bits, by convention, the upper higher bits will be filled with 0. (This is why there is no need for ***movzlq***.)

movl $0x4050, %eax	Immediate → Register	4 Bytes
movw (%rdi, %rcx), %ax	Memory → Register	2 Bytes
movb $-17, (%esp)	Immediate → Memory	1 Byte
movq %rax, -12(%rbp)	Register → Memory	8 Bytes

Moving Data Instruction, *movz{b/w}{w/l/q} Src, Dst, copying zero-extended source value to destination*

movzbw	Move zero-extended byte to word
mozbl	Move zero-extended byte to double word
movzwl	Move zero-extended word to double word
movzbq	Move zero-extended word to quad word

movzwq	Move zero-extended word to quad word

Moving Data Instruction, *movs{b/w}{w/l/q} Src, Dst, copying sign-extended source value to destination*

movsbw	Move sign-extended byte to word
mosbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax (no operands)

```
movabsq $0x00007FFFC61FA4E8, %rax      %rax = $0x00007FFFC61FA4E8
                                       Mem[0x00007FFFC61FA4E8] = 0x80
movsbl (%rax), %ebx                    %rbx = 00000000FFFFFF80
```

Remember that 0x80 is 0b10000000. Sign-extend causes higher-order bytes to be set to FF.

To perform movslq, first move the data and sign-extend on it.

Arithmetic and Logical Operations

Address computation instruction, *lea src, dest, Load Effective Address,* where src is memory address mode expression, and dest register to address computed by expression of the form *x + k * y*. *For computing addresses without a memory reference. i.e. no need for memory access, as well as common arithmetic operation.* leaq 7(%rdx, %rdx, 4) %rax **#%rax = 5x + 7**

```
short scale3(short x, short y, short z) {
        short t = 10y + z + x * y;
        return t;
}
```

```
scale3:
    leaq (%rsi, %rsi, 9), %rbx
    leaq (%rbx, %rdx), %rbx
    leaq (%rbx, %rdi, %rsi), %rbx
    ret
```

x86 Arithmetic Operations

Instruction	Operation	Notes
addl src, dst	dst = dst + src	Addition
subl src, dst	dst = dst - src	Subtraction
imull src, dst	dst = dst * src	Multiplication
sall src, dst	dst = dst << src	Shift Arithmetic Left
sarl src, dst	dst = dst >> src	Shift Arithmetic Right
xorl src, dst	dst = dst ^ src	Bitwise XOR
andl src, dst	dst = dst & src	Bitwise AND
orl src, dst	dst = dst | src	Bitwise OR
incl dest	dst = dst + 1	Increment by 1
decl dest	dst = dst - 1	Decrement by 1
negl dest	dst = -dst	Negate
notl dest	dst = ~dst	Bitwise not

Special Arithmetic Operations, providing 128 bits with registers %rdx and %rax

Instruction	Operation	Notes
imulq src	R[%rdx]:R[%rax] ←src X R[%rax]	Signed Multiplication
mulq src	R[%rdx]:R[%rax] ←src X R[%rax]	Unsigned Multiplication
idivq src	R[%rdx] ←R[%rdx]:R[%rax] mod src; R[%rax]←R[%rdx]:R[%rax] / src	Signed Divide
divq src	R[%rdx] ←R[%rdx]:R[%rax] mod src; R[%rax]←R[%rdx]:R[%rax] / src	Unsigned Divide
cqto	R[%rdx]:R[%rax] ←signExtend(R[%rax])	Convert quad word to octo word

```
int arithmetic (int x, int y, int z) {
        int t1 = x + y;
        int t2 = z + t1;
        int t3 = x + 4;
        int t4 = y * 48;
        int t5 = t3 + t4;
        int ret = t2 * t5;
        return ret;
}
```

```
arithmetic:
        leal (%rdi, %rsi), %eax         #eax = x + y
        addl %edx, %eax                 #eax = edx + eax
        leal (%rsi, %rsi, 2), %edx      #edx = y * 3
        sall $4, %edx                   #edx = edx * 16
        leal 4(%rdi, %rdx), %ecx        #ecx = x + 4 + edx
        imull %ecx, %eax                #eax = eax * ecx
        ret
```

Convention

Register	Uses
%rdi	x, 1st arg
%rsi	y, 2nd arg
%rdx	z, 3rd arg, t4
%rax	t1, t2, rval
%rcx	t5

```
int logical (int x, int y) {
        int t1 = x ^ y;
        int t2 = t1 >> 17
        int t3 = (1 << 13) - 7;
        int ret = t2 & t3;
        return ret;
}
```

```
logical:
        xorl %esi, %edi        #edi = x ^ y
        sarl $17, %edi         #edi = edi >> 17
        movl %edi, %eax        #
        andl $8185, %eax       #eax = t2  & 8185
        ret
```

Mask t3 = $2^{13} - 7$

A mask defines which bits you want to keep, and which bits you want to clear.

Control Flow

* (:) in Assembly gives memory location

Conditional branch / jump	Unconditional branch / jump
Jump to somewhere else if some condition is true, otherwise execute next instruction	Always jump when you get `break`, `continue, ...`

Condition Codes, single bit registers

SF, Sign Flag / Negative (signed)	ZF, Zero Flag	OF, Overflow Flag (signed)	CF, Carry Flag (unsigned)

Implicit setting, side effects by arithmetic operations (not lea) *e.g. addl / addq src, dst* ***#t = a + b***

<table>
<tr><th>SF SET</th><th>ZF SET</th><th>OF SET</th><th>CF SET</th></tr>
<tr><td>If t < 0 (signed)</td><td>If t == 0</td><td>If two’s complement overflow (signed)
<code>(a > 0 && b > 0 && t < 0) || (a > 0 && b > 0 && t < 0)</code></td><td>If carry out from the most significant bit (unsigned), <u>carry out</u> written to CF</td></tr>
</table>

Explicit Setting, by `compare` instruction

cmpl / cmpq b, a **#Compute a - b without specifying destination**

<table>
<tr><th>SF SET</th><th>ZF SET</th><th>OF SET</th><th>CF SET</th></tr>
<tr><td>If (a - b) < 0 (signed)</td><td>If a == b</td><td>If two’s complement overflow (signed)
<code>(a > 0 && b > 0 && (a - b) < 0) || (a > 0 && b > 0 && (a - b) < 0)</code></td><td>If carry out from the most significant bit (unsigned), <u>carry out</u> written to CF</td></tr>
</table>

Explicit Setting, by `test` instruction

testl b, a **#Compute a & b without specifying destination**

SF SET	ZF SET	OF SET	CF SET
If a & b < 0	If a & b == 0	NaN	NaN

testl %eax, %eax **#Sets SF and ZF, checks if %eax > 0 OR %eax == 0 OR %eax < 0**

Reading Condition Codes

set{} instructions* **#Set low order byte to 0 or 1 based on computation of CC**

<table>
<tr><th>set{*} instruction</th><th>Condition</th><th>Description</th></tr>
<tr><td>sete dst</td><td>ZF</td><td>Equal / Zero</td></tr>
<tr><td>setne dst</td><td>~ZF</td><td>Not Equal / Not Zero</td></tr>
<tr><td>sets dst</td><td>SF</td><td>Negative</td></tr>
<tr><td>setns dst</td><td>~SF</td><td>Nonnegative</td></tr>
<tr><td>setg dst</td><td>~(SF^OF)&~ZF</td><td>Greater (Signed)</td></tr>
<tr><td>setge dst</td><td>~(SF^OF)</td><td>Greater or Equal (Signed)</td></tr>
<tr><td>setl dst</td><td>(SF^OF)</td><td>Less (Signed)</td></tr>
<tr><td>setle dst</td><td>(SF^OF) | ZF</td><td>Less or Equal (Signed)</td></tr>
<tr><td>seta dst</td><td>~CF&~ZF</td><td>Above (Unsigned)</td></tr>
<tr><td>setb dst</td><td>CF</td><td>Below (Unsigned)</td></tr>
<tr><td>setbe dst</td><td>CF | ZF</td><td>Below or Equal (Unsigned)</td></tr>
</table>

Jumping

j{} instructions* **#Jump to target. Conditional Jump depends on CC Registers.**

j{}* instruction	Condition	Description

jmp target	1	Unconditional (Direct Jump)
jmp *Operand	1	Unconditional (Indirect Jump)
je target	ZF	Equal / Zero
jne target	~ZF	Not Equal / Not Zero
js target	SF	Negative
jns target	~SF	Nonnegative
jg target	~(SF^OF)&~ZF	Greater (Signed)
jge target	~(SF^OF)	Greater or Equal (Signed)
jl target	(SF^OF)	Less (Signed)
jle target	(SF^OF) | ZF	Less or Equal (Signed)
ja target	~CF&~ZF	Above (Unsigned)
jb target	CF	Below (Unsigned)

Choosing Instructions for Conditional

		cmp b, a	***test b, a***
je	Equal	a == b	a & b == 0
jne	Not Equal	a != b	a & b != 0
js	Negative		a & b < 0
jns	Non-negative		a & b >= 0
jg	Greater	a > b	a & b > 0
jge	Greater or Equal	a >= b	a & b >= 0
jl	Less	a < b	a & b < 0
jle	Less or Equal	a <= b	a & b <= 0
ja	Above (Unsigned)	a > b	
jb	Below (Unsigned)	a < b	

	cmp 5, (%rax)		***test %rdi, %rdi***		***test %rax, %rdi***
je:	*(%rax) == 5*	***je:***	*%rdi == 0*	***je:***	*%rdi == 0*
jne:	*(%rax) != 5*	***jne:***	*%rdi != 0*	***jne:***	*%rdi != 0*
jg:	*(%rax) > 5*	***jg:***	*%rdi > 0*	***jg:***	*%rdi > 0*
jl:	*(%rax) < 5*	***jl:***	*%rdi < 0*	***jl:***	*%rdi > 0*

```
long abs_diff (long x, long y) {
        long result;
        if (x > y) {
                result = x - y;
        } else {
                result = y - x;
        }
        return result;
}
```

```
abs_diff:
        cmpq   %rsi, rdi        # x : y
        jle .L4
        movq %rdi, %rax
        subq %rsi, %rax
        jmp .L5
    .L4:                        # x <= y
        movq %rsi, %rax
        subq %rdi, %rax
    .L5:
        ret                     # Take memory location from stack and go there
```

Conditional Move Instruction, *cmov{*}, more efficient than conditional branching but both branches are evaluated causing an overhead*

if (Test) Dst <- Src ***# Move value from Src to Dst if Test holds***

int a = (b < c) ? b : c;

```
cmpl %esi, %edi        # Compare the value of esi to edi and set flags
movl %esi, %eax        # Move the value of esi to eax
cmovge %edi, %eax      # If flag indicates greater than or equal to then move value of edi to eax
```

Bad cases:

- Expensive Computations — val = Test(x) ? **Hard1(x)** : **Hard2(x)**;
- Risky Computations — val = p ? *p : 0;
- Computations with side-effects — <u>val = x > 0 ? x *= 7 : x += 3;</u> Both values get computed and changes the value

```
long abs_diff (long x, long y) {
        long result;
        if (x > y) {
                result = x - y;
        } else {
                result = y - x;
        }
        return result;
}
```

```
abs_diff:
        movq %rdi, %rdx        # x
        subq %rsi, %rdx        # res = x - y
        movq %rsi, %rax        # y
        subq %rdi, %rax        # eval = y - x
        cmpq %rsi, %rdi        # x : y
        cmovle %rdx, %rax
        ret
```

```
if (j > 1 || j < i) {
        …
} else {
        …
}
```

```
cmpq $1, %rsi
setg %dl
cmpq %rdi, rsi
setl %al
orb %al, %dl
je .else    condition code set by side effect
```

do-while

C, factorial

```
int fact_do (int x) {
        int result = 1;
        do {
                result *= x;
                x = x - 1;
        } while (x > 1);
        return result;
}
```

C goto, factorial

```
int fact_goto (int x) {
        int result = 1;
Loop:
        result *= x;
        x = x - 1;
        if (x > 1)
                goto Loop;
        return result;
```

Assembly

```
fact_do:
        movl $1, %eax
    .L2:
        imull %edi, %eax
        subl $1, %edi
        cmpl $1, %edi
        jg .L2        # A directive is an instruction used by the assembler
        rep ret       # Some processors behave badly when ret follows after cond.
                      jump, solution is to add rep
```

while vs do-while (C and C goto comparison)

do-while

```
do                  loop:
    Body                Body
    while (Test);       if (Test)
                            goto loop;
```

while

```
while (Test)        goto test;
    Body            loop:
                        Body
                    test:
                        if (Test)
                            goto loop;
                    done:
```

`do-while` can be more optimized than `while`

gcc -Og will perform jump to the middle for `while`

while

C, factorial

```
int fact_while(int x) {
        int result = 1;
        while (x > 1) {
                result *= x;
                x = x - 1;
        }
        return result;
}
```

C goto, factorial

```
int fact_while(int x) {
        int result = 1;
        goto test;
    loop:
        result *= x;
        x = x - 1;
    test:
        if (x > 1) {
                goto loop;
        }
        return result;
```

Assembly, factorial

```
fact_while:
        movl $1 %eax
        jmp .L4
    .L3:
        imull %edi, %eax
        decl %edi
    .L4:
        cmpl $1, %edi
        jg .L3
        rep ret
```

With **-O1**, `while` translated to `do-while`

while

```
while (Test)
        Body
```

do-while

```
if (!Test)
    goto done;
do
    Body
    while (Test);
done:
```

goto-middle

```
    if (!Test)
        goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```

C, factorial

```
int fact_while(int x) {
        int result = 1;
        while (x > 1) {
                result *= x;
                x = x - 1;
        }
        return result;
}
```

C goto, factorial

```
int fact_while(int x) {
        int result = 1;
        if (!(x>1))
            goto done;
    loop:
        result *= x;
        x = x - 1;
        if (x > 1) {
                goto loop;
        }
        return result;
```

Assembly, factorial

```
fact_while:
        movl $1 %eax
        cmpl $1, %edi
        jle .L4
    .L3:
        imull %edi, %eax
        decl %edi
        cmpl $1, %edi
        jg .L3
    .L4:
        ret
```

for

```
int fact_for (int n) {
        int i;
        int result = 1;
        for (i = 2; i < n; i++) {
                result *= i;
        }
        return result;
}
```

```
for (Init; Test; Update)
    Body
```

We can convert this to a `while`

```
Init;
while (Test) {
    Body
    Update;
}
```

Machine Code generated by GCC

1. `do-while` strategy
2. “jump-to-middle” strategy

`do-while` GOTO

```
    Init;
    if (!Test) {
        goto done;
    }
loop:
    Body
    Update;
    if (Test) {
        goto loop;
    }
done:
```

`jump-to-middle` GOTO

```
    Init;
    goto test; #unfavored do to unconditional jump
loop:
    Body
    Update
test:
    if (Test)
        goto loop;
done:
```

`for`

```
for (i = 2; i <= n; i++) {
    result *= i;
}
```

`do-while` GOTO

```
    i = 2;
    if (!(i <= n)) {
        goto done;
loop:
    result *= i;
    i++;
    if (i <= n) {
        goto loop;
    }
done:
```

`jump-to-middle` GOTO

```
    i = 2;
    goto test;
loop:
    result *= i;
    i++;
test:
    if (i <= n) {
        goto loop;
    }
done:
```

`break`

`break` *statement terminates execution of the immediately enclosing* `while, do, for`, or `switch` statement. *Control passes to the statement following the loop body (or the compound statement of a* `switch` *statement).*

`continue`

`continue` statement passes control to the end of the immediately enclosing `while, do, for` statement.

```
while (1) {
    …
    goto label_1;
    …
    label_1;
}
```

```
while (1) {
    …
    continue;
    …
}
```

`switch`

Jump Tables are only used when you have a small range of values, close together

```
switch (x) {
    case val_0:
        block 0;
    case val_1:
        block 1
    …
    case val_n-1:
        block n-1;
}
```

`Jump Table`

```
JTab:
    Targ0
    Targ1
    …
    Targn-1
```

`Jump Targets`

```
Targ0:
    Code Block 0
Targ1:
    Code Block 1
…
Targn-1:
    Code Block n-1

target = JTab[x];
goto *target;
```

```
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        …
    }
    return w;
}
```

Jump Table
#usually at the end of ASM file

```
.section .rodata   #read only
    .align 8   #byte addressable
.L4
    .quad .L8   # x = 0
    .quad .L3   # x = 1
    .quad .L5   # x = 2
    .quad .L9   # x = 3
```

```
                                        .quad .L8   # x = 4
                                        .quad .L7   # x = 5
                                        .quad .L7   # x = 6
switch_ex:
    movq %rdx, %rcx
    cmpq $6, %rdi
    ja .L8          # Unsigned, catches negative default cases
    jle *.L4(, %rdi, 8)     # Jump Table .L4, Indirect Jump by providing memory location
                            # Must scale by factor of 8, addresses are 8 bytes
```

sparse `switch` statement

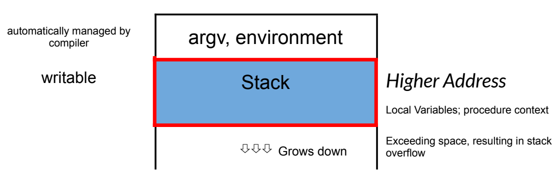
Organize the cases as a binary tree.
Translation into if-then-else requires a maximum of 9 tests, and therefore inefficient.

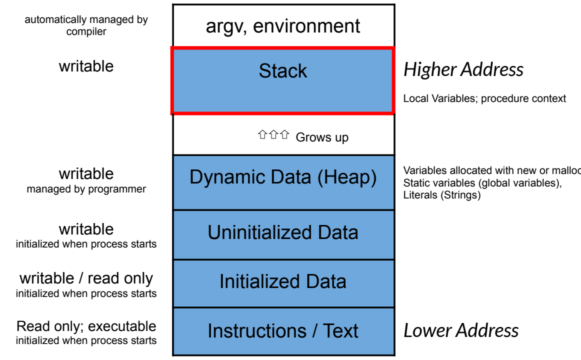
```c
int div111 (int x) {
    switch (x) {
        case 0: return 0;
        case 111: return 1;
        case 999: return 9;
        default: return -1;
    }
}
```

```
div111:
    cmpl $111, %edi
    je .L2
    jl .L3
    cmpl $999, %edi
    je .L4
    jne .L5
.L2:
    movl $1, %eax
    ret
.L3:
    cmpl $0, %eax
    je .L6
    jne .L5
.L4:
    movl $9, %eax
    ret
.L5:
    movl $-1, %eax
    ret
.L6:
    movl $0, %eax
    ret
```

Stack Procedure

Memory Layout



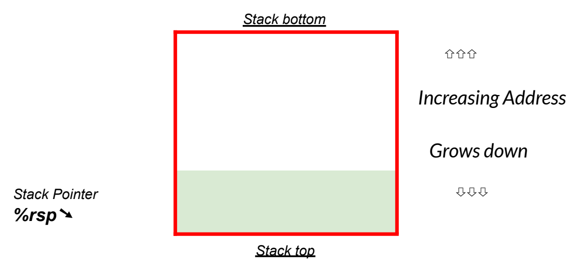


Stack

%rbp
Callee-saved Register, can optionally be used a Frame Pointer

%rsp
Stack Pointer

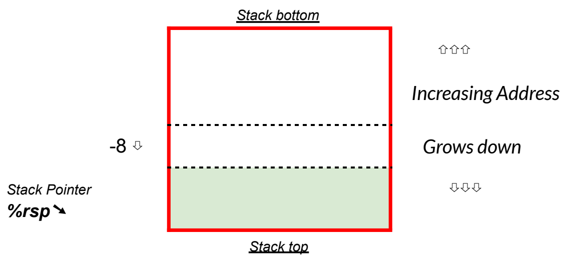
Stack Pointer
Content of *Stack Pointer* is a Memory Address from the *Stack*
Initially points to *<u>Stack top</u>*



Push

Writing data in memory

pushq Src	*Read operand at Src* *Decrement %rsp by 8* *Write operand at address given by %rsp*

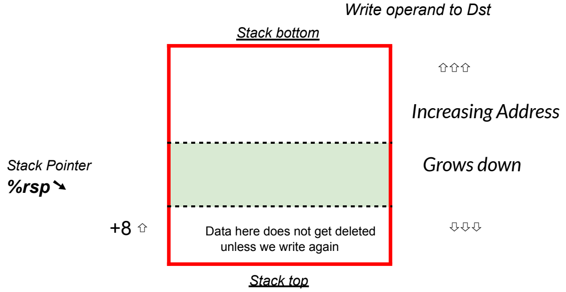


Pop

Reading data in memory

popq Dst

Read operand at address %rsp
Increment %rsp by 8
Write operand to Dst



Procedure call mechanisms

Memory Location 1

void addresult(int x, int y, int* dst) {
    int result = add(x, y);   → ***Jump to address***
    *dst = result;
}   ← **Come back**

Memory Location 2

int add(int a, int b) {
    int result = a + b;
    return result;
}

Control Flow

Caller

…
<set up arg> *(set up args add function will need)*
call *(call add function)* →
(clear all args in stack memory) **< clean up args >**
(retrieve result) **< find return val >** ↖
…
…

Callee

< create localvars > *(if any local variables in function, allocate in memory)*
…
…
< set upreturn val >
< destroy local vars > *(if memory allocated for local variables, delete)*
return

Callee *needs to know where to find args*
Callee *needs to know where to find return address*
Caller *needs to know where to find return value*
Caller *and* ***Callee*** ***uses to same registers*** *(make sure data doesn't get overwritten)*

Procedure Call

`call` *label*

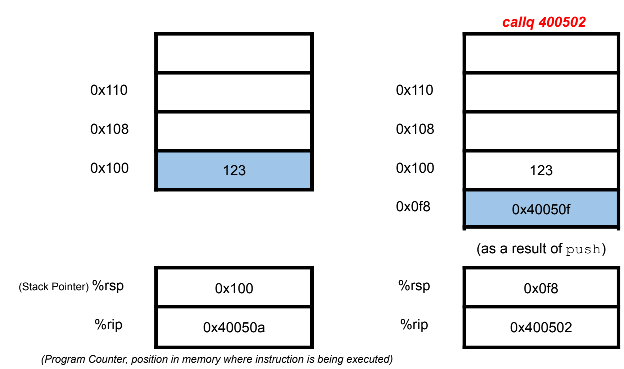
`push` return address on stack
`jmp` to `label`

Procedure return

`ret`

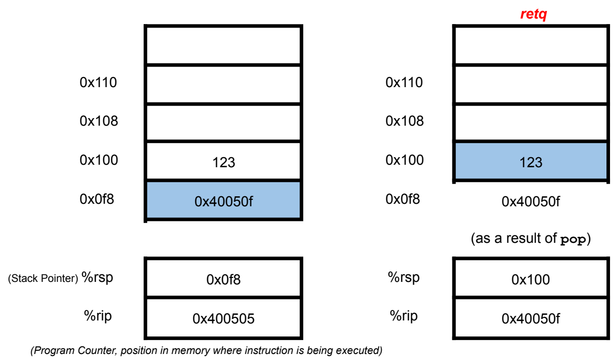
`pop` address from stack
`jmp` to address

40050a: e8 f3 ff ff ff   *`callq 400502 <add>`*   ***# return address = 0x40050f***
40050f: 89 03   *`mov %eax, (%rbx)`*
(5 byte and 2 byte instructions)



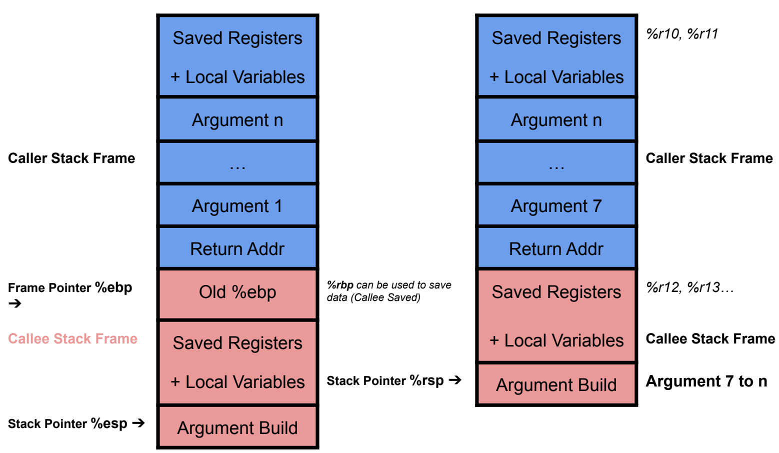
add:

```
400502    lea (%rdi, %rsi, 1), %eax
400505    retq
```



Memory Management

IA32 / Linux Stack Frame **X86-64 / Linux Stack Frame**



In the previous architecture, the frame pointer was used to move around the stack.
In X86-64, the *first 6 arguments are stored in the registers* (faster to get values), hence the caller argument starts from 7 in the caller stack frame.

Register Saving Conventions

Caller	***Callee***

`yoo:`
`movl $12345, %edx`
`call who`
`addl %edx, %eax`
`ret`
	`who:`
	`movl 8(%ebp), %edx`
	`addl $98195, %edx`
	`ret`

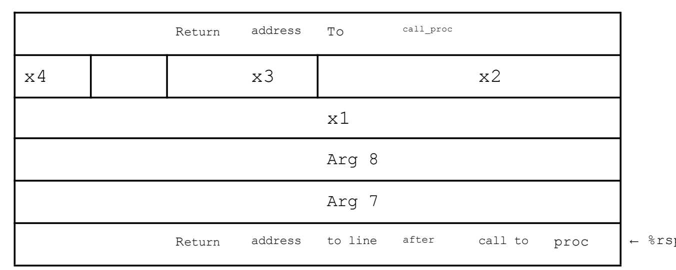
Contents of register `%edx` overwritten by `who`
Value needs to get saved

Caller Save
Caller saves temporary values in its frame before calling

Callee Save `%rbx, %rbp, %r12~%r15`
Callee saves temporary values in it frame before using

```
long int call_proc() {
     long x1 = 1;
     int x2 = 2;
     short x3 = 3;
     char x4 = 4;
     proc(x1, &x1, x2, &x2,
          x3, &x3, x4, &x4);
     Return (x1+x2) * (x3-x4);
}
```

```
call_proc:
     subq $32, %rsp
     movq $1, 16(%rsp)
     movl $2, 24(%rsp)
     movw $3, 28(%rsp)
     movb $4, 31(%rsp)
     …
```



Callee must preserve the callee-saved register's **values by** <u>**not changing the value at all**</u> **or** <u>**by pushing original value to the stack**</u>**.**

<u>**Caller-saved registers can be modified by any function.**</u>
Caller needs to <u>***save the values in the stack***</u> **to preserve the old value as the** <u>**callee is free to alter the values.**</u>

Upon pushing to stack and popping, the value of the register is restored.
Stack used when…
Too many local variables to hold in registers
Uses address-of operator (&) to compute the address of a local variable
Calls another function that takes more than 6 arguments
Need to save the state of callee-save registers before modifying them

Arrays and Structures

Basic Data Types

Type	*Size*	*C / Java*
byte	1 byte	*char*
word	2 bytes	*short*
double word	4 bytes	*int, float*
quad word	8 bytes	*long, double*
paragraph	16 bytes	*long double*

Array allocation

T **A[***L***]**;

*Allocated region of L * sizeof(T) bytes in memory*

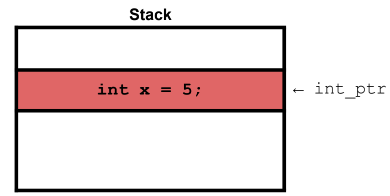
`int val[5], 20 bytes`

↓x **(BASE ADDRESS in memory, pointer)**
x+20↓

9	8	1	9	5

Reference	Type	Value
val[4]	int	5
val	int*	x
val+i	int*	x+4*i
&val[2]	int*	x+8
val[5]	int	Whatever is at [x+20]
*(val+1)	int	8

Pointer




```
int x = 5;
int* int_ptr = &x;

print(*int_ptr) prints 5
x = *int_ptr;
(*Address gives value in Address)

val[i] = *(val + i);
```

Array in C is like a block of memory you locate and a pointer pointing to the first address in memory.

```
int val[0];
val = &val[0];
(&Expr gives pointer to Expr)
```

Array access

`int val[4]`

↓16	↓20	↓24	↓28 32↓
1	5	2	1

```
int get_digit(int val[], int dig) {
     return val[dig];
}

get_digit:
     movslq %esi, %rsi
     movl (%rdi, %rsi, 4), %eax #each elements are 4 bytes
     ret
```

Array loop

```
void digit_int(int val[], int len) {
     if (len > 0) {
          int *ptr = val;  # val = &val[0]
          int *vend = val + len;
          do {
```

```
                *ptr = *ptr + 1;
                ptr++;
            } while (ptr != vend);
        }
}

digit_int:
        testl %esi, %esi
        jle   .L1
        movq %rdi, %rax
        leal -1(%rsi), %edx #len-1 in %edx
        leaq 4(%rdi, %rdx, 4), %rdx # %rdi + %rdx x 4 + 4 in %rdx
.L3:
        addl $1, (%rax)
        addq $4, %rax
        cmpq %rdx, %rax
        jne   .L3
.L1:
        rep ret
```

↓ptr vend↓

val[0]	….	….	val[len-1]

Nested Arrays

In C, arrays are stored as row major

```
typedef int zip_deg[5]; #typedef, user defined variable
                        array of 5 ints
zip_dig pgh[4] = {{1, 5, 2, 0, 6},
                  {1, 5, 2, 1, 3},
                  {1, 5, 2, 1, 7},
                  {1, 5, 2, 2, 1}} # zip_dig pgh[4]
                                   = int pgh[4][5]
array of 4, elements type zip_dig, array of 5 ints
```

pgh[0][0] … … pgh[3][4]

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

Row 0 Row 1 Row 2 Row 3

T A[**R**][**C**]

2D array of type T
Array size = R * C * K bytes
(sizeof(T) = K bytes)

Starting Address for A[i]

A+i*(C*K)

A[0]

A[0][0]	…	A[0][C-1]

↑A

A[i]

A[i][0]	…	A[i][C-1]

↑A+i * (C * K)

```
int *get_pgh_zip(int index) {
        return pgh[index];
}
        # returns pointer to integer
        # pgh[index] array of 5 ints
```

```
zip_dig pgh[4] = {
          {1, 5, 2, 0, 6},
          {1, 5, 2, 1, 3},
          {1, 5, 2, 1, 7},
          {1, 5, 2, 2, 1}}
```

leaq (%rdi, %rdi, 4), %rax # 5 * index
leaq pgh(, %rax, 4), %rax # pgh + 4 * 5 * index

Element Access A[i][j]

A[i]

…	A[i][j]	…

↑**A + i * (C * K)** + j * K

A+i*(C*K)+ j*K
= A+(i*C+j)*K

```
int get_pgh_zip(int index, int digit) {
        return pgh[index][digit];
}
        # returns integer value
```

```
leaq (%rdi, %rdi, 4), %rax # 5 * index
addq %rax, %rsi  #          # digit + 5 * index
movl pgh(, %rsi, 4), %eax  # *(pgh + 4 * (digit + 5 * index)), movl performs memory reference
```

Multilevel Array, uses pointers rather than contiguous allocation in memory, does not have to be row major

```
int cmu[5] = {1, 5, 2, 1, 3};
int cmu[3] = {1, 5, 2};
int cmu[2] = {1, 5};

int *univ[3] = {mit, cmu, ucb};
```

#array of pointers, each pointer pointing to another array
#pointer is 8 bytes

```
int get_univ_digit (int index, int dig) {
      return univ[index][dig];
}
```

```
salq $2, %rsi                # rsi = dig * 4
addq univ(, %rdi, 8), %rsi   # rsi = Mem[univ + index * 8] + dig * 4
movl (%rsi), %eax            # return univ[index][dig] element value
```

Two memory reads, **Mem[Mem[univ + 8 * index] + 4 * dig]**

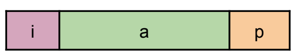
1. First to get pointer to row array
2. Access element within array

Multilevel Array vs Nested Array

Multilevel Array	*Nested Array*
`int get_univ_digit (int index, int dig) { return univ[index][dig]; }`	`int get_pgh_digit(int index, int digit) { return pgh[index][digit]; }`
Mem[Mem[univ + index * 8] + 4 * dig]	Mem[pgh + index * 20 + 4 * dig]

Only use Multilevel Arrays if arrays differ in size

Structs, structured group of variables possibly including other structs, contiguously allocated




```
struct rec {
       int i;
       int a[3];
       int *p;
};
```

Accessing a structure member is done by using a . operator

```
struct song song1;
song1.lengthInSeconds = 213;
song1.yearRecorded = 1994;
```

```
struct song song2;
song2.lengthInSeconds = 214;
song2.yearRecorded = 1988;
```

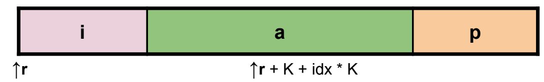
Pointer to a struct

```
struct rec r1;
struct rec *r = &r1;
(*r).i = val;      # dereference
r->i = val;        # alternatively
```

```
void set_i(struct rec *r, int val)          # %eax = val
{                                           # %edx = r
      r->i = val;                           movl %eax, (%edx)  # Mem[r] = val
}
```

Pointer to structure member

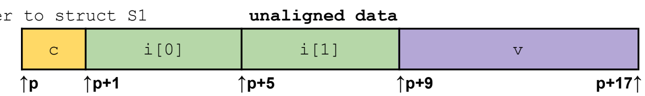
```
int *find_a(struct rec *r, int idx)         # %ecx = idx
{                                           # %edx = r
      return &r->a[idx];                    leal (,%ecx,4), %eax # 4*idx
}                                           leal 4(%eax, %edx), %eax
```

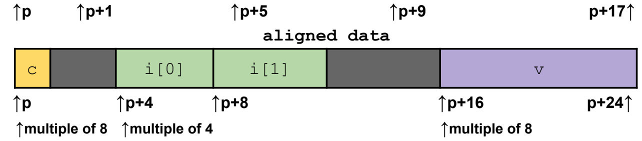


Structures and Alignment

```
typedef struct S1 { // Will be able to use p to define other vars
      char c;
      int i[2];
      double v;
} *p; // pointer to struct S1
```

```
long get_c(rec_p r) {
      return r->c;
}
```

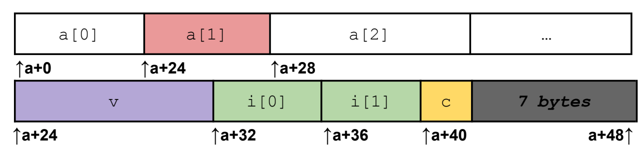




Multiple of 8 because largest alignment of any element is 8 bytes

Address must be multiple of K

```
struct S2 {
      double v;
      int i[2];
      char c;
} a[10];
```



Same size but generally better if large element comes first

Memory Hierarchy

Processor-memory bottleneck

Cache

Computer memory with short access time used for the storage of frequently or recently used instructions or data

Cache	8	9	14	3

smaller, faster, subset of data in memory

↕ Data copied in block-sized transfer units

Memory	0	1	2	3
	4	5	6	7
	8	9	10	11
	12	13	14	15

Block, the unit of transfer

Hit

When a processor requests data in Cache, then it is a HIT.

↕ request 9

Cache	8	**9**	14	3

Miss

When a processor requests data not in Cache, then it is a MISS. Block fetched from Memory, then stored in Cache.

↕ request 1

Cache	8	**1**	14	3

↕ request 1

Memory	0	**1**	2	3

Locality

Tendency of programs using data and instructions with addresses near or equal to those they have used recently.

Temporal Locality	Spatial Locality
Recently referenced items are likely to be referenced again in the near future.	Items with nearby address tend to be referenced close together in time

```
sum = 0;
for(int i = 0; i < n; i++) {
    sum += a[i];
}
return sum;
```

Data

temporal locality, `sum` referenced in each iteration

spatial locality, `a[]` accessed in stride-1 pattern

Instructions

temporal locality, cycle through loop repeatedly, effective if cache stores body of loop

spatial locality, reference instructions one after another

```
int sum_array_3d(int a[M][N][N]) {
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; i < M; k++) {
                sum += a[k][i][j];     // Does not take advantage of locality
            }                          // Especially if matrix does not fit in cache
        }       sum += a[i][j][k];
    }
    return sum;
};
```

Cost of Cache Misses

Cache hit time 1 cycle

Miss penalty 100 cycles

97% HIT $1\ cycle\ + 0.03\ \times\ 100\ cycles\ =\ 4\ cycles$

99% HIT $1\ cycle\ + 0.01\ \times\ 100\ cycles\ =\ 2\ cycles$

99% hit **twice as good as** 97% hit

$$Average\ Memory\ Access\ Time\ =\ Hit\ Time\ +\ Miss\ Rate\ \times\ Miss\ Penalty$$

Miss Rate

Fraction of memory references not found in cache (misses / accesses), $1\ -\ hit\ rate$

3%-10% for L1 Cache

Hit Time

Time to deliver a line in the cache to the processor, ***including time to determine if line is in cache***

1-2 clock cycles for L1

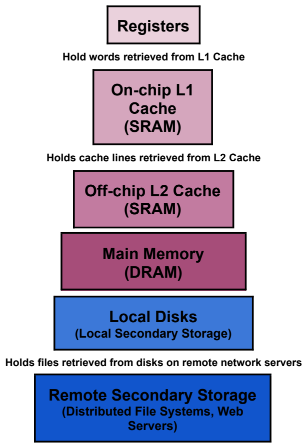
Miss Penalty

Additional time required due to a miss, 50-200 clock cycles usually

Memory Hierarchy

Each level K serves as a ***cache*** for the larger, slower, *level* $K+1$ below

Due to locality, programs tend to access data at *level* K more often than data at *level* $K+1$



Cache Organization

Tag

Together with the Index, tells what address stored in cache

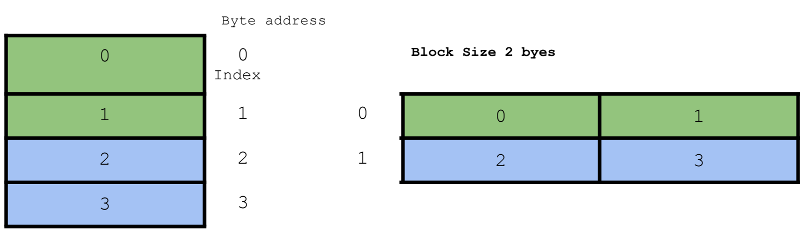
One tag per data position in cache

Spatial Locality

Items with nearby address tend to be referenced close together in time, in **blocks**

Cache Block (Cache Line)

Basic unit for cache storage. Data array for each index in cache has to be as big as the block size



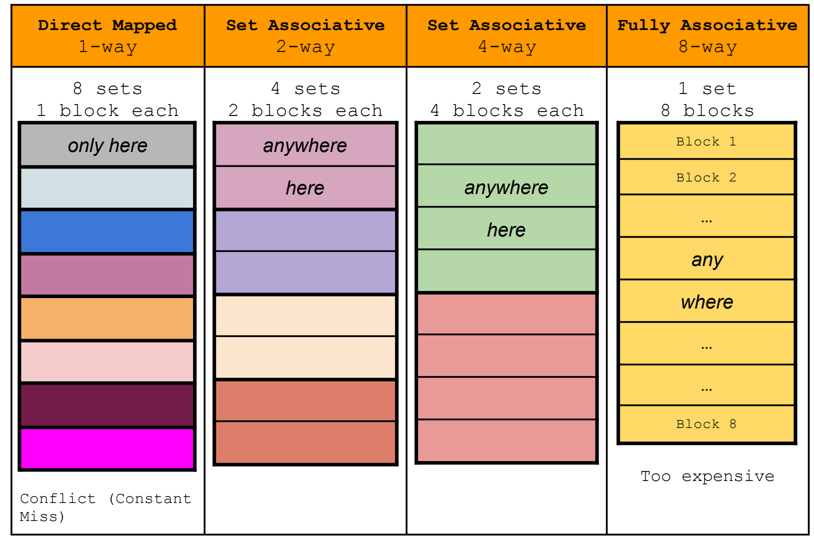
Question

Access 10 **MISS**

Access 11 **HIT**

Access 12 **MISS**

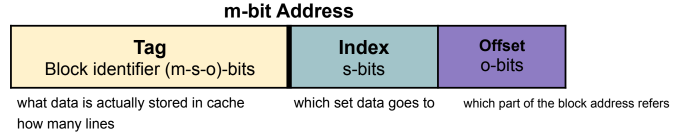
10 and 11 same cache block, block size at least 2 bytes



Direct mapped cache

For every memory location, there is a single location in cache where it can go.

m-bit Address



2^2 *block cache (sets) with* 2^1 *bytes per block (block size), 4 bit address*

Offset 1 bit (only 2^1 bytes within a block) / Index 2 bits (2^2 *block cache*, direct mapped)

0x1833 address in binary 00…0110000110011 offset bit

Block Size 2^4 bytes

Offset bit 4 bit (Can be places in 4 different parts,)

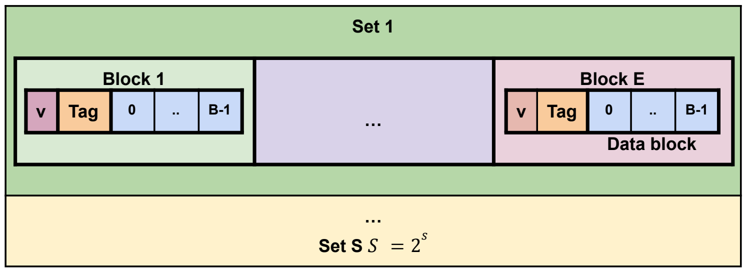
1way 8 sets 1 block then index 3 bits

2 way 4 sets then index 2 bits

Block Replacement

Caches typically replace data that is **almost** least recently used, maximizing temporal locality

Cache



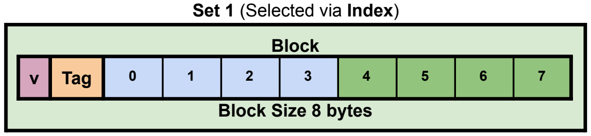
$Cache\ Size\ (C) = Number\ Of\ Sets\ (S) \times Number\ of\ Lines\ /\ Blocks\ (E)\ in\ set \times Block\ Size(B)$

Cache Read

1. Locate set
2. See if any line / block in set has matching tag
3. If match and valid, hit
4. Locate data starting at offset and send to processor

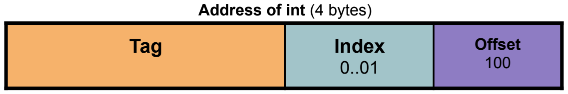
Direct Mapped Cache (E=1)

Set 1 (Selected via **Index**)

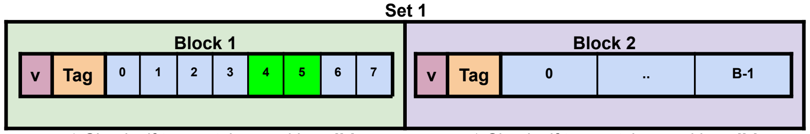


↑ Checks if tag matches and is **valid** ↑ starts reading from **Offset**
If there's a miss, old line evicted and data replaced

Address of int (4 bytes)

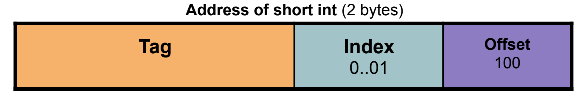


E-way Set Associative Cache (E=2)
Set 1



↑ Checks if tag matches and is **valid** ↑ Checks if tag matches and is **valid**
↑ Read data from **Offset**
<u>If there's no match</u>, ***least recently used*** line becomes a victim

Address of short int (2 bytes)



Cache Misses

Cold (Compulsory) Miss

Occurs on <u>*first access to a block*</u>

Conflict Miss

Occurs when cache is large enough but <u>*multiple data objects all map to same slot*</u>
Lower conflict misses in n-way set-associative cache

Capacity Miss

Amount of data you access repeatedly (*working set*) is *larger than the cache*

Writing Data to Cache

Data can differ down the hierarchy, **Write-Hit**

Solutions
<u>*Write-through*</u>
Write Immediately to memory

<u>*Write-back*</u>
Defer write to memory until line is evicted, write to cache
Need a dirty bit to indicate if line is different from memory or not
If line is evicted, line goes to memory

Write-miss

Solutions
<u>*Write-allocate*</u>
Load into cache, update line in cache

<u>*No-write-allocate*</u>
Immediately written to memory

Caches usually do <u>*write-back*</u> and <u>*write-allocate*</u>

Optimizations

<u>*Write code that has locality*</u>
Spatial, **access data contiguously**
Temporal, ensuring **same data accessed frequently**

Matrix Multiplication

$C_{ij} = A_{ik} \times B_{kj}$

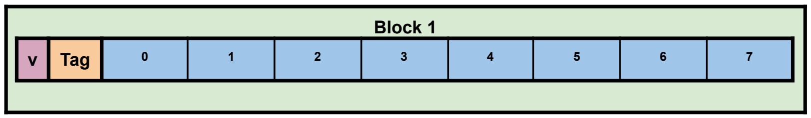
Cache Miss Analysis

Elements are doubles, **8 bytes**
Cache block 64 bytes, can **hold 8 doubles**
Cache Size C << n

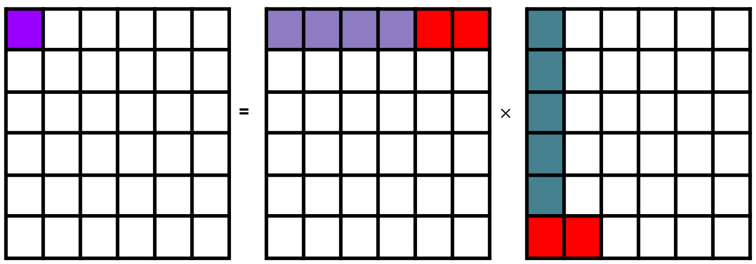
First Iteration

Memory

Row 0	**…**
A[0][0]	…
…	…
A[0][7]	…



n / 8 + n = 9n / 8 misses
(For First Row, if n = 8, after 1 miss, everything hit due to spatial locality)
(For First Column, everything misses)



Same with other iteration

Total Misses

$$9n/8 \times n^2 = 9/8 \times n^3$$

Access

$$2n^3$$

Miss Rate

$$9/16 \times n^3$$

Blocked Matrix Multiplication

Cache Miss Analysis

Elements are doubles, **8 bytes**
Cache block 64 bytes, can **hold 8 doubles**
Cache Size C << n

Three blocks (r x r) fit into cache $3r^2 < C$

First Iteration

$r^2/8$ misses for each block

r^2 elements per block, 8 elements per cache block
$2n/r$ blocks in total hence $nr/4$ misses

Total Misses

$$nr/4 \times (n/r) \times (n/r) = n^3/4r$$