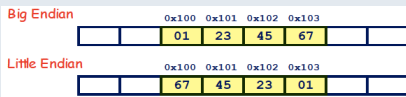
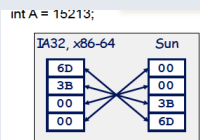


Byte ordering of 0x01234567



Byte representation of ints



Bit operations (integral data type)

```

01101001  01101001  01101001
& 01010101  | 01010101  ^ 01010101  ~ 01010101
01000001  01111101  00111100  10101010
    
```

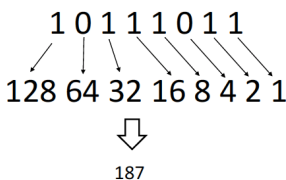
Logical operators

```

- !0x41 = 0x00
- !0x00 = 0x01
- !0x41 = 0x01
- 0x09 && 0x55 = 0x01
- 0x09 || 0x55 = 0x01
- p && *p (avoids null pointer access)
    
```

Unsigned integers

$$B2U(X) = \sum_{i=0}^{W-1} x_i \cdot 2^i$$



2's complement

for each positive number (X), assign value to its negative (-X), such that $X + (-X) = 0$ with "normal" addition, ignoring carry out

```

00101 (5)      01001 (9)
+ 11011 (-5)   + 10111 (-9)
00000 (0)      00000 (0)
    
```

2's complement

$$\text{Two'sComp}(x) + x = 0$$

$$\text{Two'sComp}(x) = \sim x + 1$$

Converting 2's C to decimal

Converting Binary (2's C) to Decimal

- If MS bit is one, take two's complement to get a positive number.
- Get the decimal as if the number is unsigned (using power of 2s).
- If original number was negative, add a minus sign.

n	2 ⁿ
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Floating Point Rep

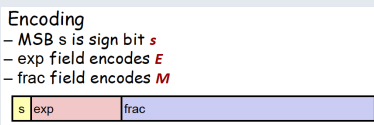
$(-1)^s M 2^E$

Sign bit **s** determines whether number is negative or positive

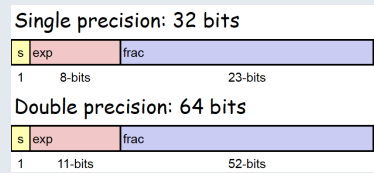
Significand **M** a fractional value

Exponent **E** weights value by power of two

Encoding



Precision



Normalized encoding

Condition: $\text{exp} \neq 000...0$ and $\text{exp} \neq 111...1$

referred to as Bias

Exponent is: $E = \text{Exp} - (2^{k-1} - 1)$, k is the # of exponent bits

- Single precision: $E = \text{exp} - 127$
- Double precision: $E = \text{exp} - 1023$

Significand is: $M = 1.xxx...x_2$

- Range(M) = [1.0, 2.0-ε)
- Get extra leading bit for free

Normalized encoding example

Value: Float F = 15213.0;

$$15213_{10} = 11101101101_2 = 1.1101101101_2 \times 2^{13}$$

Significand

M = 1.1101101101₂

frac = 11011011010000000000₂

Exponent

E = exp - Bias = exp - 127 = 13

→ exp = 140 = 10001100₂

Result:

0	10001100	110110110110100000000000
s	exp	frac

Denormalized encoding

Condition: $\text{exp} = 000...0$

Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - \text{Bias}$)

Significand is: $M = 0.xxx...x_2$ (instead of $M = 1.xxx_2$)

Cases

- exp = 000...0, frac = 000...0
 - Represents zero
 - Note distinct values: +0 and -0
- exp = 000...0, frac ≠ 000...0
 - Numbers very close to 0.0

Specialized encoding

Condition: $\text{exp} = 111...1$

Case: $\text{exp} = 111...1, \text{frac} = 000...0$

- Represents value ∞ (infinity)
- Operation that overflows
- E.g., 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -∞

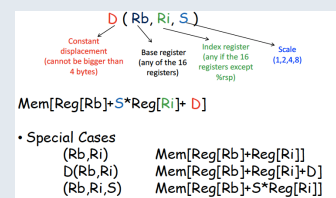
Case: $\text{exp} = 111...1, \text{frac} \neq 000...0$

- Not-a-Number (NaN)
- Represents case when no numeric value can be determined
- E.g., sqrt(-1), ∞ - ∞, ∞ × 0

movq operand combo

	Source	Dest	Src, Dest	C Analog
movl	Imm	Reg	movq %0x4, %rax	temp = 0x4;
		Mem	movq %-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Address computation



Multiplication

Unsigned

- Form 1: `imulq s, d`
 - $d = s * d$
 - multiply two 64-bit operands and put the result in 64-bit register
- Form 2: `mulq s`
 - one operand is `rax`
 - The other operand given in the instruction
 - product is stored in `rdx` (high-order part) and `rax` (low order part)
 - Full 128-bit result

Signed

- Form 1: `imulq s, d`
 - $d = s * d$
 - multiply two 64-bit operands and put the result in 64-bit register
- Form 2: `imulq s`
 - one operand is `rax`
 - The other operand given in the instruction
 - product is stored in `rdx` (high-order part) and `rax` (low order part)
 - Full 128-bit result

Division

Unsigned

- `divq s`
 - Dividend given in `rdx` (high order) and `rax` (low order)
 - Divisor is `s`
 - Quotient stored in `rax`
 - Remainder stored in `rdx`

Signed

- `idivq s`
 - Dividend given in `rdx` (high order) and `rax` (low order)
 - Divisor is `s`
 - Quotient stored in `rax`
 - Remainder stored in `rdx`

SetX dest: only set lower 1 byte of register

setx	Condition	Description
<code>sete</code>	ZF	Equal / Zero
<code>setne</code>	\neg ZF	Not Equal / Not Zero
<code>seta</code>	SF	Negative
<code>setna</code>	\neg SF	Nonnegative
<code>setg</code>	\neg (SF<OF) && (S<OF)	Greater (Signed)
<code>setge</code>	\neg (SF<OF)	Greater or Equal (Signed)
<code>setl</code>	(SF<OF)	Less (Signed)
<code>setle</code>	(SF<OF) ZF	Less or Equal (Signed)
<code>seta</code>	\neg CF && \neg SF	Above (unsigned)
<code>setb</code>	CF	Below (unsigned)

Jumping

JX	Condition	Description
<code>jmp</code>	<code>i</code>	Unconditional
<code>je</code>	ZF	Equal / Zero
<code>jne</code>	\neg ZF	Not Equal / Not Zero
<code>js</code>	SF	Negative
<code>jns</code>	\neg SF	Nonnegative
<code>jg</code>	\neg (SF<OF) && (S<OF)	Greater (Signed)
<code>jge</code>	\neg (SF<OF)	Greater or Equal (Signed)
<code>jl</code>	(SF<OF)	Less (Signed)
<code>jle</code>	(SF<OF) ZF	Less or Equal (Signed)
<code>ja</code>	\neg CF && \neg SF	Above (unsigned)
<code>jb</code>	CF	Below (unsigned)

2 operand instructions

Format	Computation
<code>addq Src, Dest</code>	<code>Dest = Dest + Src</code>
<code>subq Src, Dest</code>	<code>Dest = Dest - Src</code>
<code>imulq Src, Dest</code>	<code>Dest = Dest * Src</code>
<code>salq Src, Dest</code>	<code>Dest = Dest << Src</code> ← Also called <code>shlq</code>
<code>sarq Src, Dest</code>	<code>Dest = Dest >> Src</code> ← Arithmetic
<code>shrq Src, Dest</code>	<code>Dest = Dest >> Src</code> ← Logical
<code>xorq Src, Dest</code>	<code>Dest = Dest ^ Src</code>
<code>andq Src, Dest</code>	<code>Dest = Dest & Src</code>
<code>orq Src, Dest</code>	<code>Dest = Dest Src</code>

one operand instructions

<code>incq Dest</code>	<code>Dest = Dest + 1</code>
<code>decq Dest</code>	<code>Dest = Dest - 1</code>
<code>negq Dest</code>	<code>Dest = - Dest</code>
<code>notq Dest</code>	<code>Dest = ~ Dest</code>

useful instruction for division

cqto

- No operands
- Takes the sign bit from `rax` and replicates it in `rdx`

Setting condition codes

The processor does not know if you are using signed or unsigned integers. OF and CF are set for every arithmetic operation.

Implicitly setting condition code: addq src, dest

CF (Carry flag) set if carry out from most significant (31st) bit (unsigned overflow)

ZF (Zero flag) set if `t == 0`

SF (Sign flag) set if `t < 0` (as signed)

OF (Overflow flag) set if signed overflow (`a > 0 && b > 0 && t < 0` || `a < 0 && b < 0 && t > 0`)

Bad cases for conditional move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Effect of operations

Logical Operations CF=0, OF=0

Operations

shift CF=value of last bit shifted out; OF=0

INC, DEC OF and ZF may change, CF unchanged

Explicitly setting condition codes

`cmpl b, a` a-b result not stored anywhere

`testq b, a` a&b result not stored anywhere

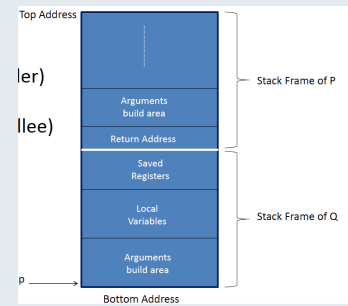
When are local variables in stack?

Enough registers

No reference to & so no need to go to memory

No arrays, structures

When P(caller) calls Q



Structure representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

Structure represented as block of memory

– Big enough to hold all of the fields

Fields ordered according to declaration

– Even if another ordering could yield a more compact representation

Compiler determines overall size + positions of fields

– Machine-level program has no understanding of the structures in the source code

Procedure data flow

Registers

• First 6 arguments

```

%rdi
%rsi
%rdx
%rcx
%r8
%r9
```

• Return value

```
%rax
```

Stack

```

***
Arg n
***
Arg 8
Arg 7
```

• Only allocate stack space when needed

• When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.

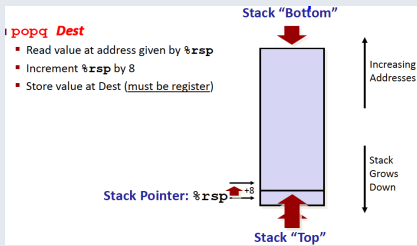
Register usage

%rax	Return value	%rax
- Return value		%rdi
- Also caller-saved		%rsi
- Can be modified by procedure		%rdx
%rdi, ..., %r9	Arguments	%rcx
- Arguments		%r8
- Also caller-saved		%r9
- Can be modified by procedure		%r10
%r10, %r11	Caller-saved temporaries	%r11
- Caller-saved		
- Can be modified by procedure		

Register usage contd

%rbx, %r12, %r13, %r14	Caller-saved	%rbx
- Caller-saved		%r12
- Caller must save & restore		%r13
%rbp	Caller-saved	%r14
- Caller must save & restore		%rbp
- May be used as frame pointer	Special	%rsp
- Can mix & match		
%rsp	Special form of callee save	
- Restored to original value upon exit from procedure		

Popq dest (for stack)



Array access

Basic Principle

$T A[L]$;
 - Array of data type T and length L
 - Identifier A used as a pointer to array element 0: Type T^*

`int val[5];`

0	1	2	3	4
x	$x+4$	$x+8$	$x+12$	$x+16$

Reference	Type	Value
<code>val[4]</code>	int	3
<code>val</code>	int *	x
<code>val+1</code>	int *	$x+4$
<code>&val[2]</code>	int *	$x+8$
<code>val[5]</code>	int	??
<code>*(val+1)</code>	int	5
<code>val + i</code>	int *	$x+4i$

Cache structure



Cache calculation

$S=2^s$ number of sets	$g = \log_2 M$ # set index bits
$E = \#$ of elements	$l = \log_2 B$ # block offset bits
$B = 2^b$ block size (bytes)	$i = \log_2 B$ # tag bits
$N = 2^n$ max number of unique addresses	$n = \log_2 N$ # of address bits
$C = B \times S$ cache size in bytes	

Cache miss

3 cases compulsory, capacity, conflict

How to reduce miss
 block size++, associativity++, cache size++

Reducing miss penalty
 write through (update all) vs write back (update when needed)

multilevel cache (optimize hit rate L1, miss rate L2)

Replacement policies
 LRU, LFU, FIFO, rand

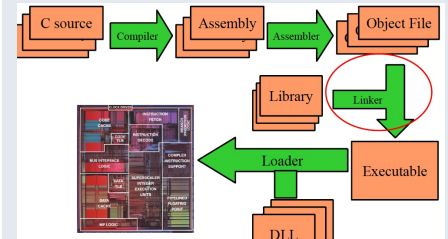
Cache access time

Let m = cache access time
 M = access time memory ($m < M$)
 P = probability that we find data in cache
 Average access time = $Pm + (1 - P)M = m + (1 - P)(M - m)$

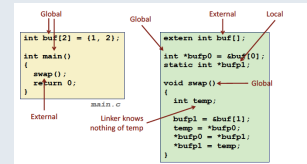
Why Linkers

- Modularity**
- Write program as a set of smaller source files, rather than one giant file
 - Allow for libraries of common functions (more on this later)
 - e.g., math library, standard C library
- Efficiency**
- Separate compilation saves time
 - Change one source file, compile that file only, and then relink.
 - Libraries save memory space
 - Common functions can be aggregated into a single file...
 - Yet executable files contain only code for the functions they actually use.

Source code to execution



Resolving symbols



Why VM

memory management and protection
 permission bits; uses main efficiently (send unneeded to disk)

Process isolation/memory protection
 own address spaces; can't interfere with another's memory

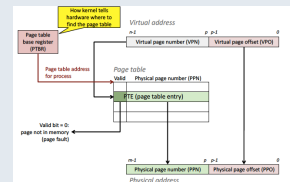
loading linking simplified

VP partitioned to 3 subsets

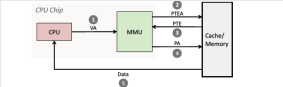
Unallocated
 not yet created, no data, no space

Uncached/cached
 currently cached/not cached

Address translation w page table



Page hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

VA: Virtual Address
PA: Physical Address
PTE: Page Table Entry
PTEA: PTE Address

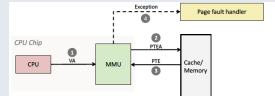
mem alloc challenges

memory utilization (sum of malloc'd data/heap size)

good performance (malloc/free calls return quick)

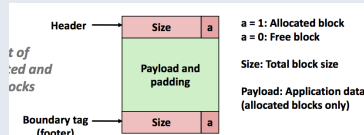
constraints: can't modify malloc'd memory; can't move malloc'd block

page fault

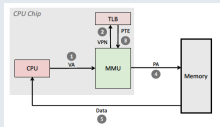


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception in kernel
- IF VA is invalid, then kill process (SIGSEGV)
- IF VA has been passed out to disk, then swaps in faulted page, update page table, resume faulted process

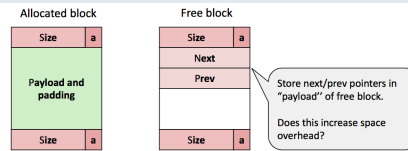
implicit free list with footer and header



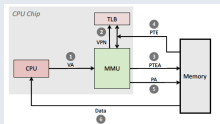
Speed: TLB hit, mem access-1



explicit free list (pointer don't space+)



TLB miss (rare with high assoc): 3 mem accesses



Seg list

- Segregated list allocation: multiple free lists each linking free blocks of similar sizes
- Have separate classes for each small size
 - One class for each 2 power size (the smallest block that can be given to a programmer is 32 bytes, so the 1st class is not 16 bytes)
 - To allocate a block of size n
 - Search in appropriate free list containing size n
 - Split if needed, place fragment on appropriate list
 - Try next larger class (no blocks found)
 - If still no blocks, request additional heap memory from OS
 - Allocate block of n bytes from the new memory
 - Place remainder as a single free block in largest size class
 - To free: coalesce and place on appropriate list
 - Advantages: fast allocation, better memory utilization (first fit search of seg list approximates a best fit search of an entire heap)

size-- multilevel page table

If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.

Only the level 1 table needs to be in main memory at all times.

The level 2 page tables can be created and paged in and out by the VM system as they are needed.

classes of exceptions

Class	Cause	Asynch	Return behavior
Interrupt	Signal from I/O device	MMIO	Always to next instruction
Trap	Intentional exception	Sync	Always to next instruction
Fault	Potential recoverable error	ASPC	Might return to current instruction
Abort	Non-recoverable	ASPC	Never returns

cache and VM

cache uses PA, since with VA, although can be accessed asap, aliasing, 2 VA may map to same block, would not know which one

exception examples

```

Interrupts
Trap: Sys calls (only in kernel mode)
  System calls
    Read, write, open, close, stat, mmap, bio, dup2, pause, alarm, getpid, fork,
    execv, _exit, wait4, kill
Faults:
  Page fault, exception—an instruction references a VA whose page is not in memory and
  must be retrieved from disk
  Divide error
  Invalid memory reference: Segmentation fault
Aborts: machine check (fatal hardware error)
    
```



By **mathildapurr**

cheatography.com/mathildapurr/

Published 22nd December, 2016.

Last updated 22nd December, 2016.

Page 4 of 4.

Sponsored by **ApolloPad.com**

Everyone has a novel in them. Finish Yours!

<https://apollopad.com>