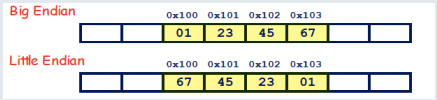
Cheatography

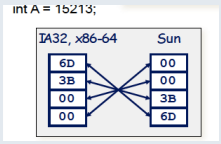
CSO Cheat Sheet

by mathildapurr via cheatography.com/33013/cs/10300/

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

- 0x69 && 0x55  = 0x01
- 0x69 || 0x55  = 0x01
- p && *p       (avoids null pointer access)
```

Unsigned integers

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

1 0 1 1 1 0 1 1

128 64 32 16 8 4 2 1

⇩

187

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

Two'sComp(x) + x = 0

```
Two's Comp(x) = ~x + 1
```

Converting 2's C to decimal

Converting Binary (2's C) to Decimal

1. If MS bit is one, take two's complement to get a positive number.
2. Get the decimal as if the number is unsigned (using power of 2s).
3. 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

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

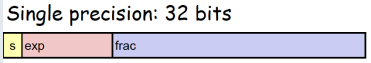
Sign bit ***s*** determines whether number is negative or positive
Significand ***M*** a fractional value
Exponent ***E*** weights value by power of two

Encoding

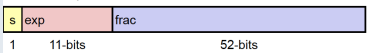
Encoding
- MSB s is sign bit ***s***
- exp field encodes ***E***
- frac field encodes ***M***

Precision

Single precision: 32 bits



Double precision: 64 bits



Normalized encoding

Condition: exp ≠ 000…0 and exp ≠ 111…1

referred to as Bias

Exponent is: ***E*** = ***Exp*** - (2^{k-1} - 1), k is the # of exponent bits
- Single precision: E = exp - 127
- Double precision: E = exp - 1023

frac

Significand is: ***M*** = $1.xxx\ldots x_2$
- Range(*M*) = [1.0, 2.0-ε)
- Get extra leading bit for free

Range(E)=[-126,127]
Range(E)=[-1022,1023]

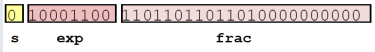
Normalized encoding example

Value: `Float F = 15213.0;`
$15213_{10} = 11101101101101_2$
$= 1.1101101101101_2 \times 2^{13}$

Significand
$M = 1.1101101101101_2$
$frac = 1101101101101\,0000000000_2$

Exponent
E = exp - Bias = exp - 127 = 13
→ exp = 140 = 10001100_2

Result:



Denormalized encoding

Condition: exp = 000…0

Exponent value: ***E*** = 1 - ***Bias*** (instead of ***E*** = 0 - ***Bias***)
Significand is: ***M*** = $0.xxx\ldots x_2$ (instead of ***M=1.xxx***$_2$)

frac

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

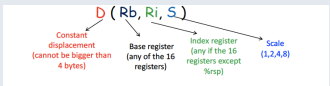
Condition: `exp` = 111…1

Case: `exp` = 111…1, `frac` = 000…0
- Represents value ∞ (infinity)
- Operation that overflows
- E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

Case: `exp` = 111…1, `frac` ≠ 000…0
- Not-a-Number (NaN)
- Represents case when no numeric value can be determined
- E.g., sqrt(-1), ∞ − ∞, ∞ × 0

movq operand combo

Address computation



Mem[Reg[Rb]+S*Reg[Ri]+ D]

• Special Cases

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

By **mathildapurr**
cheatography.com/mathildapurr/

Published 22nd December, 2016.
Last updated 22nd December, 2016.

Cheatography

CSO Cheat Sheet

by mathildapurr via cheatography.com/33013/cs/10300/

Multiplication

Unsigned
- form 1: imulq s, d
  - d = s * d
  - multiply two 64-bit operands and put the result in 64-bit operand
- 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 operand
- 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
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Jumping

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

2 operand instructions

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

one operand instructions

incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest – 1
negq	Dest	Dest = – Dest
notq	Dest	Dest = ~Dest

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 (31-st) 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
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

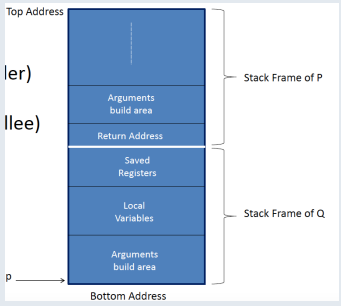
When are local variables in stack?

Enough registers

No reference to & so no need to go to memory

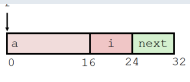
No arrays, structures

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.

By **mathildapurr**
cheatography.com/mathildapurr/

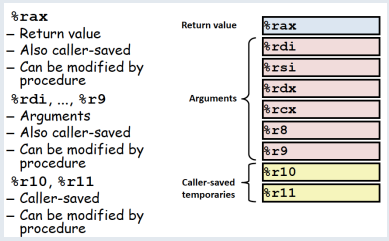
Published 22nd December, 2016.
Last updated 22nd December, 2016.

Cheatography

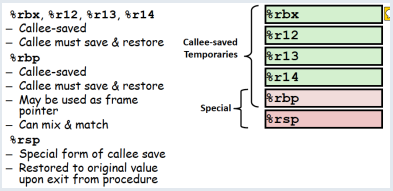
CSO Cheat Sheet

by mathildapurr via cheatography.com/33013/cs/10300/

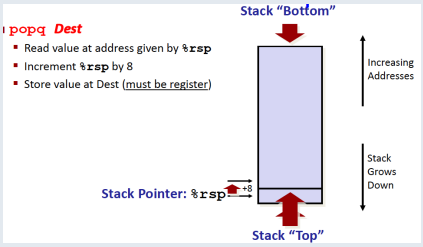
Register usage



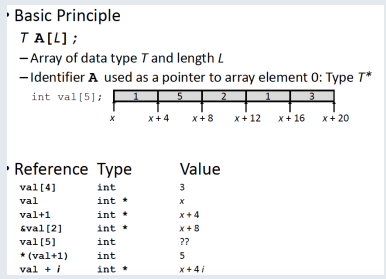
Register usage contd



Popq dest (for stack)

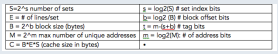


Array access



Cache structure

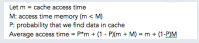
Cache calculation



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

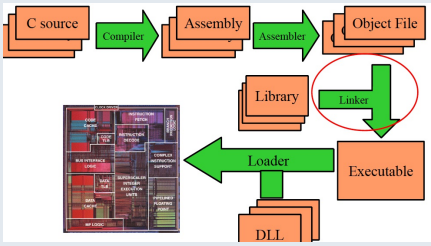


Why Linkers

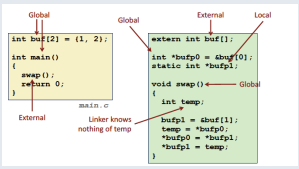
Modularity
- Write program as a set of smaller source files, rather than one giant file
- Allow for libraries of common functions (more on this later)
  - e.g., math library, standard C library

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



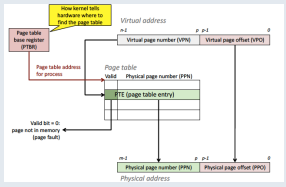
Why VM

memory management and protection	permission bits; uses main efficiently(send unneeded to disk)
Process isolation/memory protection	own add 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



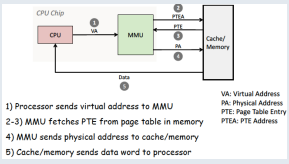
By **mathildapurr**

cheatography.com/mathildapurr/

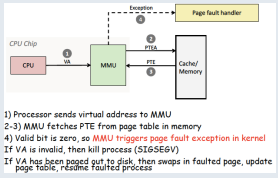
Published 22nd December, 2016.

Last updated 22nd December, 2016.

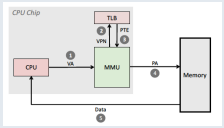
Page hit



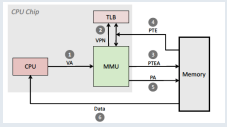
page fault



Speed: TLB hit, mem access-1



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



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.

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

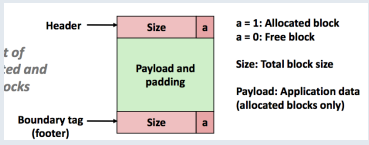
mem alloc challenges

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

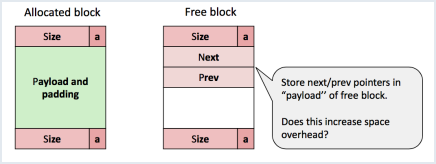
good performance (malloc/free calls return quick)

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

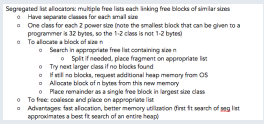
implicit free list with footer and header



explicit free list (pointer don't space+)



Seg list



classes of exceptions

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

exception examples