Compiler

x86-64 Assembly

Jyun-Ao Lin

iFIRST & CSIE, NTUT

Credits

A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Overview of the course

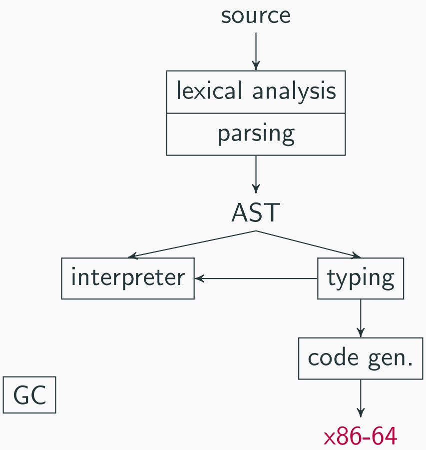


Table of contents

A little bit of computer arithmetic

Reminder of computer arithmetic

An integer is represented using n bits,
written from right (least significant) to left (most significant)

b_{n-1}	b_{n-2}	$\cdots$	b_1	b_0

The bits b_{n-1}, b_{n-2}, etc. are said to heavy weight
and the bits b_0, b_1, etc. are said to low weight

The number n of bits is typically equal to 8, 16, 32 or 64
When $n = 8$ we are talking about a byte

unsigned integer

$$
\begin{aligned}
\text{bits} &= b_{n-1} b_{n-2} \ldots b_1 b_0 \\
\text{value} &= \sum_{i=0}^{n-1} b_i 2^i
\end{aligned}
$$

bits	value
$000\ldots000$	0
$000\ldots001$	1
$000\ldots010$	2
$\vdots$	$\vdots$
$111\ldots110$	$2^n - 2$
$111\ldots111$	$2^n - 1$

Example

$$00101010_2 = 42$$

Signed integer: two's complement

the most significant bit b_{n-1} is the sign bit

$$\begin{aligned}\text{bits} &= b_{n-1} b_{n-2} \ldots b_1 b_0 \\ \text{value} &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\end{aligned}$$

Example

$$\begin{aligned}11010110_2 &= -128 + 86 \\ &= -42\end{aligned}$$

bits	value
$100 \ldots 000$	-2^{n-1}
$100 \ldots 001$	$-2^{n-1} + 1$
$\vdots$	$\vdots$
$111 \ldots 110$	-2
$111 \ldots 111$	-1
$000 \ldots 000$	0
$000 \ldots 001$	1
$000 \ldots 010$	1
$\vdots$	$\vdots$
$011 \ldots 110$	$2^{n-1} - 2$
$011 \ldots 111$	$2^{n-1} - 1$

Beware!

According to the context, the same bits are interpreted either as a signed or unsigned integer

Example

- $11010110_2 = -42$ (signed 8-bit integer)
- $11010110_2 = 214$ (unsigned 8-bit integer)

Operations

the machine provide operations such as

- logical (aka bitwise) operations: and, or, xor, not
- shift operations
- arithmetic operations: addition, subtraction, multiplication, etc.

Logical operations

operation		example
negation	x	00101001
	not x	11010110
and	x	00101001
	y	01101100
	x and y	00101000
or	x	00101001
	y	01101100
	x or y	01101101
xor	x	00101001
	y	01101100
	x xor y	01000101

Shift operation

- logical shift left (inserts least significant zeros)

$$\leftarrow \begin{array}{|c|c|c|c|c|c|}\hline b_{n-3} & \cdots & b_1 & b_0 & 0 & 0 \\ \hline\end{array} \leftarrow$$

  (<< in Java, `lsl` in Ocaml)

- logical shift right (inserts most significant zeros)

$$\rightarrow \begin{array}{|c|c|c|c|c|c|}\hline 0 & 0 & b_{n-1} & \cdots & b_3 & b_2 \\ \hline\end{array} \rightarrow$$

  (>>> in Java, `lsr` in Ocaml)

- arithmetic shift right (duplicates the sign bit)

$$\rightarrow \begin{array}{|c|c|c|c|c|c|}\hline b_{n-1} & b_{n-1} & b_{n-1} & \cdots & b_3 & b_2 \\ \hline\end{array} \rightarrow$$

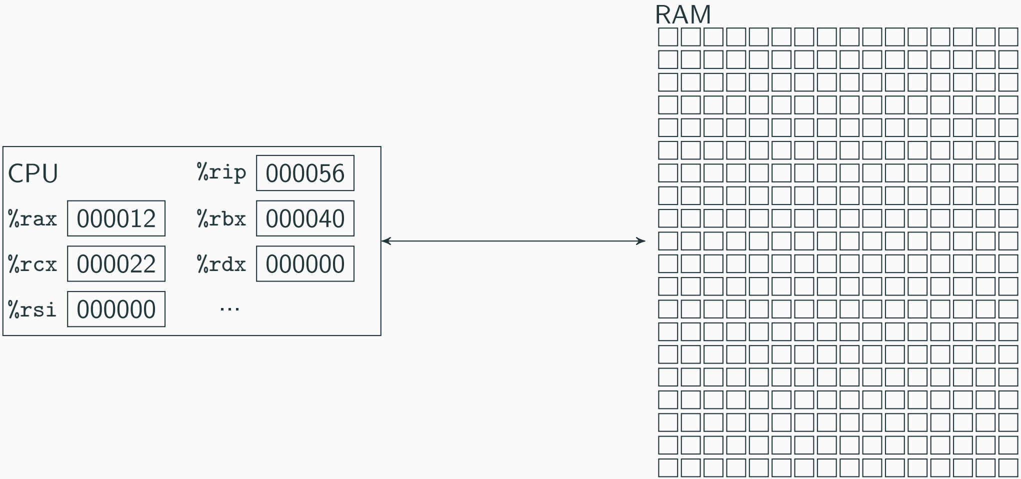
  (>> in Java, `asr` in Ocaml)

A little bit of architecture

roughly speaking, a computer is composed

- of a CPU, containing
  - few integer and floating-point registers
  - some computation power
- memory (RAM)
  - composed of a large number of bytes (8 bits)
    for instance, 1 GiB $= 2^{30}$ bytes $= 2^{33}$bits, that is $2^{2^{33}}$ possible states
  - contains data and instructions

A little bit of architecture



accessing memory is costly (at one billion instructions per second, light only traverses 30 centimeters!)

A little bit of architecture

reality is more complex:

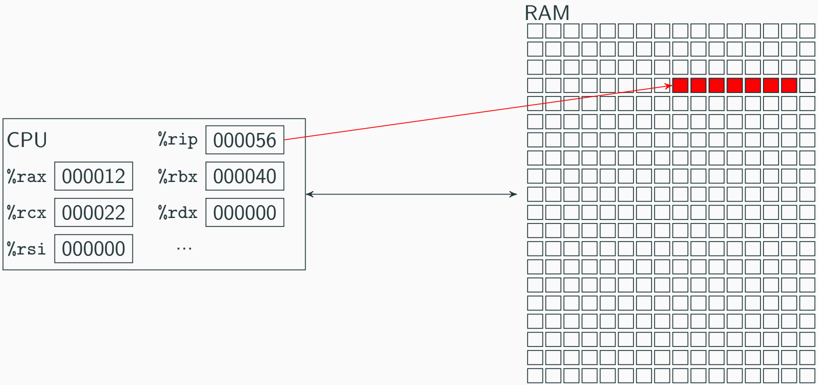
- several (co)processors, some dedicated to floating-point
- one or several memory caches
- virtual memory (MMU)
- etc.

Execution principle

execution proceeds according to the following:

- a register (`%rip`) contains the address of the next instruction to execute
- we read one or several bytes at this address (*fetch*)
- we interpret these bytes as an instruction (*decode*)
- we execute the instruction (*execute*)
- we modify the register `%rip` to move to the next instruction (typically the one immediately after, unless we jump)

Execution principle



instruction:	48	c7	c0	2a	00	00	00
decoding:		`movq`	`%rax`	42			

i.e. store 42 into register `%rax`.

Principle

again, reality is more complex:

- pipelines
  - several instructions are executed in parallel
- branch prediction
  - to optimize the pipeline, we attempt at predicting conditional branches

Which architecture for this course?

Two main families of microprocessors

- CISC (*Complex Instruction Set*)
  - many instructions
  - many addressing modes
  - many instructions read / write memory
  - few registers
  - examples: VAX, PDP-11, Motorola 68xxx, AMD/Intel x86
- RISC (*Reduced Instruction Set*)
  - few instructions
  - few instructions read / write memory
  - many registers
  - examples: Alpha, Sparc, MIPS, ARM

we choose x86-64 for this course (and the project)

x86-64 architecture

History sketch

x86 a family of compatible architectures

- **1974** Intel 8080 (8 bits)
- **1978** Intel 8086 (16 bits)
- **1985** Intel 80386 (32 bits)

x86-64 a 64-bit extension

- **2000** introduced by AMD
- **2004** adopted by Intel

x86-64 architecture

- 64 bits
  - arithmetic, logical, and transfer operations over 64 bits
- 16 registers
  - `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rbp`, `%rsp`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`
- addresses memory over at least 48 bits ($\geq$ 256 TB)
- many addressing modes

x86-64 assembly

we do not code in machine language, but using the assembly language

the assembly language provides several facilities:

- symbolic names
- allocation of global data

assembly language is turned into machine code by a program called an assembler (a compiler)

Environment

in this lecture, I'm using Linux and GNU tools

in particular, I'm using GNU assembly, with AT&T syntax

in other environments, the tools may differ

in particular, the assembly language may use Intel syntax, which is different

hello world

```
        .text                       # instructions follow
        .globl main                 # make main visible for ld
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $message, %rdi      # argument of puts
        call    puts
        movq    $0, %rax            # return code 0
        popq    %rbp
        ret
        .data                       # data follow
message:
        .string "hello, world"      # 0-terminated string
```

Execution

assembling

```
> as hello.s -o hello.o
```

linking (`gcc` calls `ld`)

```
> gcc -no-pie hello.o -o hello
```

(note: no need for `-no-pie` if you use older version of `gcc`)

execution

```
> ./hello
Hello, world!
```

disassembling

we can disassemble using `objdump`

```
> objdump -d hello.o
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp, %rbp
   4:   48 c7 c7 00 00 00 00    mov    $0x0, %rdi
   b:   e8 00 00 00 00          call   10 <main+0x10>
  10:   48 c7 c0 00 00 00 00    mov    $0x0, %rax
  17:   5d                      pop    %rbp
  18:   c3                      ret
```

we note that

- addresses for the string and puts are not yet known
- the code is located at address 0

disassembling

we can also disassemble the executable

```
> objdump -d hello
0000000000401126 <main>:
   401126:   55                      push   %rbp
   401127:   48 89 e5                mov    %rsp, %rbp
   40112a:   48 c7 c7 30 40 40 00    mov    $0x404030, %rdi
   401131:   e8 fa fe ff ff          call   401030 <puts@plt>
   401136:   48 c7 c0 00 00 00 00    mov    $0x0, %rax
   40113d:   5d                      pop    %rbp
   40113e:   c3                      ret
```

we now see

- an effective address for the string ($0x404030)
- an effective address for function puts ($0x401030)
- a program location at $0x401126

endianness

Note that the bytes of `0x00404030` are stored in memory in the order `30, 40, 40, 00`

We say that the machine is little-endian

Other architectures are big-endian or bi-endian

(reference: Jonathan Swift's *Gulliver's Travels*)

gdb

a step-by-step execution is possible using gdb (the GNU debugger)

```
> gcc -g -no-pie hello.s -o hello
> gdb hello
GNU gdb (GDB) 7.1-ubuntu
...
(gdb) break main
Breakpoint 1 at 0x401126: file hello.s, line 4.
(gdb) run
Starting program: .../hello
Breakpoint 1, main () at hello.s:4
4 movq $message, %rdi
(gdb) step
5 call puts
(gdb) info registers
...
```

Nemiver

We can also use Nemiver (if you've installed it)

```
> nemiver hello
```

Instruction set

Registers

63	31	15	8 7	0
%rax	%eax	%ax	%ah	%al
%rbx	%ebx	%bx	%bh	%bl
%rcx	%ecx	%cx	%ch	%cl
%rdx	%edx	%dx	%dh	%dl
%rsi	%esi	%si		%sil
%rdi	%edi	%di		%dil
%rbp	%ebp	%bp		%bpl
%rsp	%esp	%sp		%spl

63	31	15	8 7	0
%r8	%r8d	%r8w		%r8b
%r9	%r9d	%r9w		%r9b
%r10	%r10d	%r10w		%r10b
%r11	%r11d	%r11w		%r11b
%r12	%r12d	%r12w		%r12b
%r13	%r13d	%r13w		%r13b
%r14	%r14d	%r14w		%r14b
%r15	%r15d	%r15w		%r15b

constants, addresses, copies

- loading a constant into a register

```
movq    $0x2a, %rax   # rax <- 42
movq    $-12,  %rdi
```

- loading the address of a label into a register

```
movq    $label, %rdi
```

- copying a register into another register

```
movq    %rax, %rbx   # rbx <- rax
```

arithmetic

- addition of two registers

```
    addq    %rax, %rbx      # rbx <- rbx + rax
```

(similarly, `subq`, `imulq`)

- addition of a register and a constant

```
    addq    $2, %rcx        # rcx <- rcx + 2
```

- particular case

```
    incq    %rbx            # rbx <- rbx+1
```

(similarly, `decq`)

- negation

```
    negq    %rbx            # rbx <- -rbx
```

logical operations

- logical not

```
notq    %rax            # rax <- not(rax)
```

- and, or, exclusive or

```
orq     %rbx, %rcx      # rcx <- or(rcx, rbx)
andq    $0xff, %rcx     # erases bits >= 8
xorq    %rax, %rax      # zeroes %rax
```

shift

- shift left (inserting zeros)

```
salq    $3, %rax    # 3 times
salq    %cl, %rbx   # cl times
```

- arithmetic shift right (duplicating the sign bit)

```
sarq    $2, %rcx
```

- logical shift right (inserting zeros)

```
shrq    $4, %rdx
```

- rotation

```
rolq    $2, %rdi
rorq    $3, %rsi
```

operand size

the suffix q means a 64-bit operand (*quad words*)

other suffixes are allowed

suffix	#bytes	
`b`	1	*byte*
`w`	2	*word*
`l`	4	*long*
`q`	8	*quad*

```
    movb    $42, %ah
```

when operand sizes differ, one must indicate the extension mode

```
    movzbq  %al, %rdi        # with zeros extension
    movswl  %ax, %edi        # with sign  extension
```

memory access

an operand between parentheses means an indirect addressing
i.e. the data in memory at this address

```
movq    $42, (%rax)     # mem[rax] <- 42
incq    (%rbx)          # mem[rbx] <- mem[rbx] + 1
```

note: the address may be a label

```
movq    %rbx, (x)
```

limitation

operations do not allow several memory accesses

```
addq    (%rax), (%rbx)
```

```
Error: too many memory references for `add'
```

one has to use a temporary register

```
movq    (%rax), %rcx
addq    %rcx, (%rbx)
```

indirect addressing

the general form of the operand is

$$A(B, I, S)$$

and it stands for address $A + B + I \times S$ where

- A is a 32-bit signed constant
- I is 0 when omitted
- $S \in \{1, 2, 4, 8\}$ (is 1 when omitted)

```
    movq    -8(%rax,%rdi,4), %rbx  # rbx <- mem[-8+rax+4*rdi]
```

effective address

operation `lea` computes the effective address of the operand

$$A(B, I, S)$$

```
    leaq    -8(%rax,%rdi,4), %rbx    # rbx <- -8+rax+4*rdi
```

note: we can make use of it to perform arithmetic

```
    leaq  (%rax,%rax,2), %rbx  # rbx <- 3*%rax
```

flags

most operations set the processor flags, according to their outcome

flag	meaning
`ZF`	the result is 0
`CF`	a carry was propagated beyond the most significant bit
`SF`	the result is negative
`OF`	arithmetic overflow (signed arith.)
etc	

(notable exception: `lea`)

using the flags

three instructions can test the flags

- conditional jump (j*cc*)

```
jne label
```

- computes 1 (true) or 0 (false) (set*cc*)

```
setge %bl
```

- conditional mov (cmov*cc*)

```
cmovl %rax, %rbx
```

suffix		meaning
e	z	$= 0$
ne	nz	$\neq 0$
s		< 0
ns		≥ 0
g		signed $>$
ge		signed $\geq$
l		signed $<$
nl		signed $\leq$
a		unsigned $>$
ae		unsigned $\geq$
b		unsigned $<$
be		unsigned $\leq$

comparisons

one can set the flags without storing the result anywhere, as if doing a subtraction or a logical and

```
    cmpq    %rbx, %rax         # flags of rax - rbx
```

(beware of the direction!)

```
    testq   %rbx, %rax       # flags of rax & rbx
```

unconditional jump

- to a label

```
    jmp     label
```

- to a computed address

```
    jmp     *%rax
```

but also

many, many other instructions
[*Enumerating x86-64 – It's Not as Easy as Counting*]

including SSE instructions operating on large registers containing several integers or floating-point numbers

The challenge of compilation

the challenge of compilation

The challenge of compilation is to translate a high-level program into this instruction set

in particular, we have to

- translate control structures (tests, loops, exceptions, etc.)
- translate function calls
- translate complex data structures (arrays, structures, objects, closures, etc.)
- allocate dynamic memory

Function calls

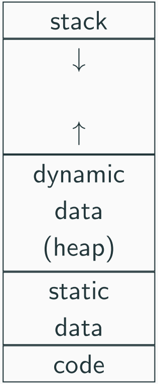
observation: function calls can be arbitrarily nested

$\Rightarrow$ registers cannot hold all the local variables

$\Rightarrow$ we need to allocate memory

yet function calls obey a *last-in first-out* mode, so we can use a stack

The stack



the stack is allocated at the top of the memory, and increases downwards; `%rsp` points to the top of the stack

dynamic data (which needs to survive function calls) is allocated on the heap (possibly by a GC), above static data, and increases upwards

this way, no collision between the stack and the heap (unless we run out of memory)

note: each program has the illusion of using the whole memory; the OS creates this illusion, using the MMU

stack handling

- pushing

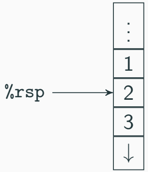
```
pushq   $42
pushq   %rax
```

- popping

```
popq    %rdi
popq    (%rbx)
```

Example

```
pushq   $1
pushq   $2
pushq   $3
popq    %rax
```



function call

when a function f (the caller)
needs to call a function g (the callee),
it cannot simply do

```
    jmp g
```

since we need to come back to the code of f when g terminates

the solution is to make use of the stack

function call

two instructions for this purpose

instruction

```
    call    g
```

1. pushes the address of the next instruction on the stack
2. transfers control to address g

and instruction

```
    ret
```

1. pops an address from the stack
2. transfers control to that address

function call

problem: any register used by g is lost for f

there are many solutions,
but we typically resort to calling conventions

calling conventions

- up to six arguments are passed via registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- other arguments are passed on the stack, if any
- the returned value is put in `%rax`

- registers `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14` and `%r15` are `callee-saved`, i.e. the callee must save them if needed; typically used for long-term data, which must survive function calls
- the other registers are caller-saved i.e. the caller must save them if needed; typically used for short-term data, with no need to survive calls

- `%rsp` is the stack pointer, `%rbp` the frame pointer

alignment

on function entry, `%rsp + 8` must be a multiple of `16`

library functions (such as `scanf` for instance) may fail if this is not ensured

alignment

stack alignment may be performed explicitly

```
f:   subq   $8, %rsp    # align the stack
     ...
     ...                # since we make calls to extern functions
     ...
     addq   $8, %rsp
     ret
```

or indirectly

```
f:   pushq  %rbx        # we save %rbx
     ...
     ...                # because we use it here
     ...
     popq   %rbx        # and we restore it
     ret
```

calling conventions

calling conventions are nothing more than conventions

in particular, we are free not to use them as long we stay within the perimeter of our own code

when linking to external code (e.g. `puts` earlier), however,
we must obey the calling conventions

function calls, in four steps

there are four steps in a function call

1. for the caller, before the call
2. for the callee, at the beginning of the call
3. for the callee, at the end of the call
4. for the caller, after the call

they interact using the top of the stack, called the stack frame and located between `%rsp` and `%rbp`

the caller, before the call

1. passes arguments in `%rdi,... ,%r9`, and others on the stack, if more than 6
2. saves caller-saved registers, in its own stack frame, if they are needed after the call
3. executes

```
    call callee
```

the callee, at the beginning of the call

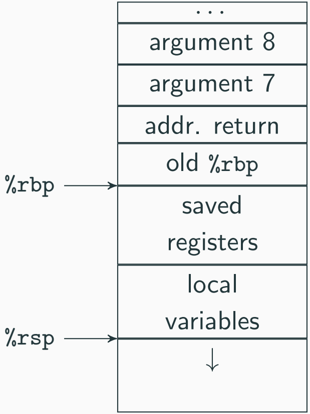
1. saves `%rbp` and set it, for instance with

```
    pushq    %rbp
    movq     %rsp, %rbp
```

2. allocates its stack frame, for instance with

```
    subq     $48, %rsp
```

3. saves callee-saved registers that it intends to use



`%rbp` eases access to arguments and local variables, with a fixed offset (whatever the top of the stack)

the callee, at the end of the call

1. stores the result into `%rax`
2. restores the callee-saved registers, if needed
3. destroys its stack frame and restores `%rbp` with

   ```
       leave
   ```

   that is equivalent to

   ```
       movq    %rbp, %rsp
       popq    %rbp
   ```

4. executes

   ```
       ret
   ```

the caller, after the call

1. pops arguments 7, 8, ..., if any
2. restores the caller-saved registers, if needed

Exercise

Program the following function

$\texttt{isqrt}(n) \equiv$

$c \leftarrow 0$

$s \leftarrow 1$

while $s \leq n$ **do**

  $c \leftarrow c + 1$

  $s \leftarrow s + 2c + 1$

return c

and print the value of $\texttt{isqrt}(17)$

recap

- a machine provides
  - a limited instruction set
  - efficient registers, costly access to the memory
- the memory is split into
  - code / static data / dynamic data (heap) / stack
- function calls make use of
  - a notion of stack frame
  - calling conventions

An example of compilation

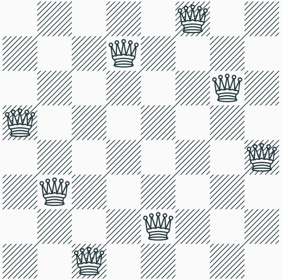
An example of compilation

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(c+d)/2));return f
    ;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

clarification

```
int t(int a, int b, int c) {
  int f=1;
  if (a) {
    int d, e = a & ~b & ~c;
    f = 0;
    while (d = e & -e) {
      f += t(a-d, (b+d)*2, (c+d)/2)
        e -= d;
    }
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("q(%d) = %d\n", n, t(~(~0<<n), 0, 0));
}
```

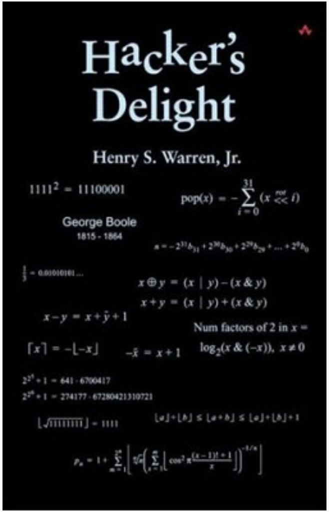
this program computes the number of solutions to the N-queens problem



how does it work?

- brute force search (backtracking)
- integers used as sets:
  e.g. $13 = 0 \cdots 01101_2 = \{0, 2, 3\}$

integers	sets
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, when $a \cap b = \emptyset$
`a-b`	$a \setminus b$, when $b \subseteq a$
`~a`	a^c
`a&-a`	$\{\min(a)\}$, when $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i+1 \mid i \in a\}$, written $S(a)$
`a/2`	$\{i-1 \mid i \in a \wedge i \neq 0\}$, written $P(a)$



explaining `a&-a`

in two's complement: `-a` $=$ `~a+1`

$$
\begin{array}{rcl}
\texttt{a} & = & b_{n-1}b_{n-2}\ldots b_k 10\ldots 0 \\
\texttt{\~{}a} & = & \overline{b_{n-1}b_{n-2}}\ldots \overline{b_k}01\ldots 1 \\
\texttt{-a} & = & \overline{b_{n-1}b_{n-2}}\ldots \overline{b_k}10\ldots 0 \\
\texttt{a\&-a} & = & 0 \quad 0\ldots\ 010\ldots 0
\end{array}
$$

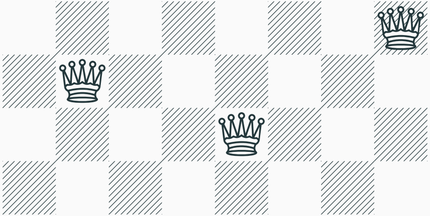
Example

$$
\begin{array}{rcccl}
\texttt{a} & = & 00001100 & = & 12 \\
\texttt{-a} & = & 11110100 & = & -12 = -128 + 116 \\
\texttt{a\&-a} & = & 00000100 & &
\end{array}
$$

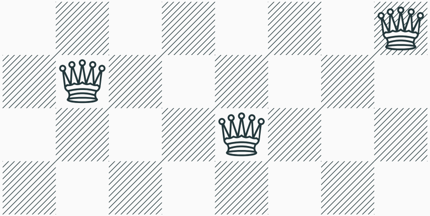
clarification: code with sets

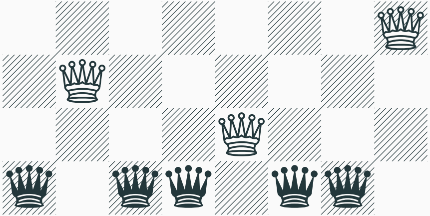
```
int t(a, b, c) =
f ← 1
if a ≠ ∅ then
    e ← (a \ b) \ c
    f ← 0
    while e ≠ ∅ do
        d ← min(e)
        f ← f + t(a \ {d}, S(b ∪ {d}), P(c ∪ {d}))
        e ← e \ {d}
return f

int queens(n) = return t({0, 1, ..., n − 1}, ∅, ∅)
```

meaning of a, b, and c



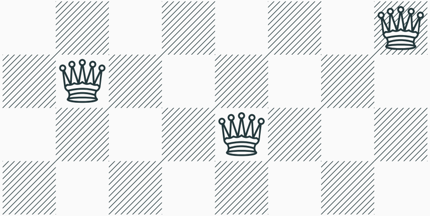
meaning of a, b, and c

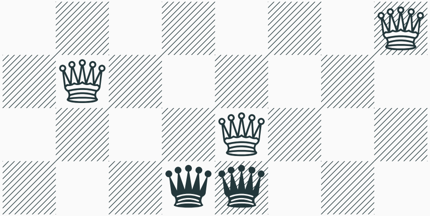




$a =$ columns to be filled $= 10110110_2$.

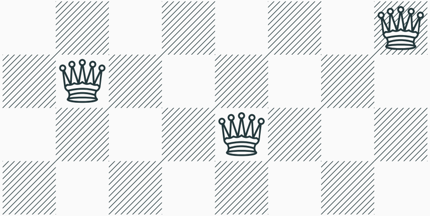
meaning of a, b, and c

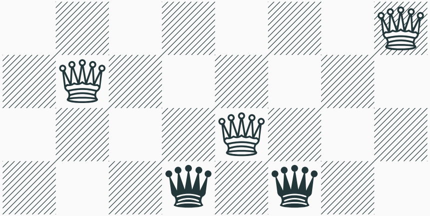




$b =$ unavailable columns by the left diagonals $= 00011000_2$.

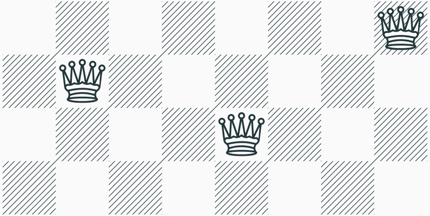
meaning of a, b, and c

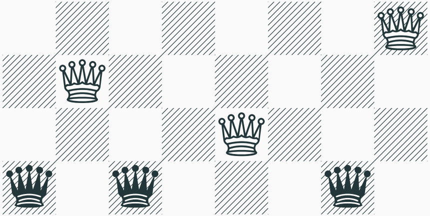




$c =$ unavailable columns by the right diagonals $= 00010100_2$.

meaning of a, b, and c





`a&~b&~c` $=$ available positions $= 10100010_2$.

why using this program?

```c
int t(int a, int b, int c) {
  int f=1;
  if (a) {
    int d, e = a & ~b & ~c;
    f = 0;
    while (d = e & -e) {
      f += t(a-d, (b+d)*2, (c+d)/2)
        e -= d;
    }
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("q(%d) = %d\n", n, t(~(~0<<n), 0, 0));
}
```

short, yet contains

- a test (`if`)
- a loop (`while`)
- a recursive function
- a few computations

and this is an excellent solution to the N-queens problem

compilation

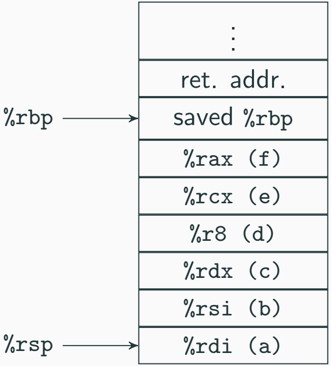
let's start with recursive function `t`; we need

- to allocate registers
- to compile
  - the test
  - the loop
  - the recursive call
  - the various computations

register allocation

- `a`, `b`, and `c` are passed in `%rdi`, `%rsi`, and `%rdx`
- the result is returned in `%rax`
- local variables `d`, `e`, and `f` will be in `%r8`, `%rcx`, and `%rax`

when making a recursive call, `a`, `b`, `c`, `d`, `e`, and `f` will have to be saved, for they are all used after the call ⇒ saved on the stack



compiling the test

```c
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        ...
    }
    return f;
}
```

```
t:        movq    $1, %rax
          testq   %rdi, %rdi
          jz      t_return
          ...
t_return:
          ret
```

allocating/deallocating the stack frame

```
t:       ...
         pushq   %rbp
         movq    %rsp, %rbp
         subq    $48, %rsp    # allocate 6 words on the stack
         ...
         addq    $48, %rsp
         popq    %rbp
t_return:
         ret
```

when $a \neq 0$

```
if (a) {
    int d, e=a&~b&~c;
    f = 0;
    while ...
}
```

```
xorq    %rax, %rax   # f <- 0
movq    %rdi,%rcx    # e <- a&~b&~c
movq    %rsi, %r9
notq    %r9
andq    %r9, %rcx
movq    %rdx, %r9
notq    %r9
andq    %r9, %rcx
```

note the use of a temporary register `%r9` (not saved)

compiling the loop

```
while (expr) {
    body
}
```

```
        ...
L1:     ...
            compute expr into %rcx
        ...
        testq   %rcx, %rcx
        jz      L2
        ...
            body
        ...
        jmp L1
L2:     ...
```

compiling the loop

there are better options, though

```
while (expr) {
    body
}
```

```
        ...
        jmp L2
L1:     ...
            body
        ...
L2:     ...
            expr
        ...
        testq   %rcx, %rcx
        jnz     L1
```

this way we make a single branching instruction per loop iteration
(apart for the very first iteration)

compiling the loop

```
while (d=e&-e) {
    ...
}
```

```
        jmp loop_test
loop body:
        ...
loop test:
        movq %rcx, %r8
        movq %rcx, %r9
        negq %r9
        andq %r9, %r8
        jnz loop_body
t return:
        ...
```

compiling the loop

```
while (...) {
    f += t(a-d,
           (b+d)*2,
           (c+d)/2);
    e -= d;
}
```

```
loop body:
        movq    %rdi, 0(%rsp)   # a
        movq    %rsi,  8(%rsp)  # b
        movq    %rdx, 16(%rsp)  # c
        movq    %r8,  24(%rsp)  # d
        movq    %rcx, 32(%rsp)  # e
        movq    %rax, 40(%rsp)  # f
        subq    %r8, %rdi
        addq    %r8, %rsi
        salq    $1, %rsi
        addq    %r8, %rdx
        shrq    $1, %rdx
        call    t
        addq    40(%rsp), %rax  #f
        movq    32(%rsp), %rcx  #e
        subq    24(%rsp), %rcx  #-=d
        movq    16(%rsp), %rdx  #c
        movq     8(%rsp), %rsi  #b
        movq     0(%rsp), %rdi  #a
```

main function

```c
int main() {
    int q;
    scanf("%d", &q);
    ...
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $input, %rdi
        movq    $q, %rsi
        xorq    %rax, %rax
        call    scanf
        movq    (q), %rcx
        ...

        .data
input:
        .string "%d"
q:
        .quad   0
```

programme principal

```c
int main() {
    ...
    printf("%d\n",
           t(~(~0<<q), 0, 0));
}
```

```
main:
        ...
        xorq    %rdi, %rdi
        notq    %rdi
        salq    %cl, %rdi
        notq    %rdi
        xorq    %rsi, %rsi
        xorq    %rdx, %rdx
        call    t
        movq    $msg, %rdi
        movq    %rax, %rsi
        xorq    %rax, %rax
        call    printf
        xorq    %rax, %rax
        popq    %rbp
        ret
```

optimization

this code is not optimal

(for instance, we could save only 5 registers)

yet it is more efficient than the output of `gcc -O2` or `clang -O2`

no reason to show off: we wrote an assembly code specific to this C program, manually, not a compiler!

lesson

- producing efficient assembly code is not easy
- observe the code produced by your compiler
  using `gcc -S -fverbose-asm`, or `ocamlopt -S`, etc.

  or even better at `https://godbolt.org/`
- now we have to automate all this

further reading

- [BO16] Computer Systems: A Programmer's Perspective (R. E. Bryant, D. R. O'Hallaron)
- its PDF appendix *x86-64 Machine-Level Programming*

- *Notes on x86-64 programming* by Andrew Tolmach and his webpage x86-64 Resources

References i

R.E. Bryant and D.R. O'Hallaron.
Computer Systems: A Programmer's Perspective.
Always Learning. Pearson, 2016.

Questions?