INF564 – Compilation

Jean-Christophe Filliâtre

x86-64 assembly
a little bit of computer arithmetic

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

\[
\begin{array}{cccc}
  b_{n-1} & b_{n-2} & \ldots & b_1 & b_0 \\
\end{array}
\]

typically, $n$ is 8, 16, 32, or 64
unsigned integer

bits \ = \ b_{n-1} b_{n-2} \ldots b_1 b_0  \\
value \ = \ \sum_{i=0}^{n-1} b_i 2^i

<table>
<thead>
<tr>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000\ldots000</td>
<td>0</td>
</tr>
<tr>
<td>000\ldots001</td>
<td>1</td>
</tr>
<tr>
<td>000\ldots010</td>
<td>2</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>111\ldots110</td>
<td>2^{n-2}</td>
</tr>
<tr>
<td>111\ldots111</td>
<td>2^{n-1}</td>
</tr>
</tbody>
</table>

example: 00101010_2 = 42
signed integer: two’s complement

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

bits = $b_{n-1}b_{n-2} \ldots b_1b_0$

value = $-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i2^i$

data table:

<table>
<thead>
<tr>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100...000</td>
<td>$-2^{n-1}$</td>
</tr>
<tr>
<td>100...001</td>
<td>$-2^{n-1} + 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>111...110</td>
<td>$-2$</td>
</tr>
<tr>
<td>111...111</td>
<td>$-1$</td>
</tr>
<tr>
<td>000...000</td>
<td>0</td>
</tr>
<tr>
<td>000...001</td>
<td>1</td>
</tr>
<tr>
<td>000...010</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>011...110</td>
<td>$2^{n-1} - 2$</td>
</tr>
<tr>
<td>011...111</td>
<td>$2^{n-1} - 1$</td>
</tr>
</tbody>
</table>

example:

$11010110_2 = -128 + 86 = -42$
according to the context, the same bits are interpreted either as a signed or unsigned integer

every:

- $11010110_2 = -42$ (signed 8-bit integer)
- $11010110_2 = 214$ (unsigned 8-bit integer)
The machine provides several operations:

- Logical, or bitwise, operations (AND, OR, XOR, NOT)
- Shift operations
- Arithmetic operations (addition, subtraction, multiplication, etc.)
<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>x 00101001</td>
</tr>
<tr>
<td>NOT x</td>
<td>11010110</td>
</tr>
<tr>
<td>AND</td>
<td>x 00101001</td>
</tr>
<tr>
<td>y 01101100</td>
<td>x AND y 00101000</td>
</tr>
<tr>
<td>OR</td>
<td>x 00101001</td>
</tr>
<tr>
<td>y 01101100</td>
<td>x OR y 01101101</td>
</tr>
<tr>
<td>XOR</td>
<td>x 00101001</td>
</tr>
<tr>
<td>y 01101100</td>
<td>x XOR y 01000101</td>
</tr>
</tbody>
</table>
• logical shift left (inserts least significant zeros)

$$\leftarrow \begin{array}{cccc}
& b_{n-3} & \ldots & b_1 \\
& b_0 & 0 & 0
\end{array} \leftarrow$$

($$\ll$$ in Java, $$\text{lsl}$$ in OCaml)

• logical shift right (inserts most significant zeros)

$$\rightarrow \begin{array}{cccc}
& 0 & 0 & b_{n-1} \\
& \ldots & b_3 & b_2
\end{array} \rightarrow$$

($$\gg$$ in Java, $$\text{lsr}$$ in OCaml)

• arithmetic shift right (duplicates the sign bit)

$$\rightarrow \begin{array}{cccc}
& b_{n-1} & b_{n-1} & b_{n-1} \\
& \ldots & b_3 & b_2
\end{array} \rightarrow$$

($$\gg$$ in Java, $$\text{asr}$$ in OCaml)
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 Gb = $2^{30}$ 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!)
reality is more complex:

- several (co)processors, some dedicated to floating-point
- one or several memory caches
- virtual memory (MMU)
- etc.
The execution principle of a register-based processor is as follows:

1. **Register**: A register, such as `%rip` (Instruction Pointer), contains the address of the next instruction to execute.
2. **Fetch**: We read one or several bytes at this address.
3. **Decode**: We interpret these bytes as an instruction.
4. **Execute**: We execute the instruction.
5. **Update**: We modify the register `%rip` to move to the next instruction (typically the one immediately after, unless we jump).

This process is repeated as the program runs through its instructions.
execution principle

CPU

%rip 0000056
%rax 0000012 %rbx 0000040
%rcx 0000022 %rdx 0000000
%rsi 0000000 ...

RAM

instruction: 48 c7 c0 2a 00 00 00 00

decoding: movq %rax 42

i.e. store 42 into register %rax
again, reality is more complex:

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

- **CISC** (*Complex Instruction Set*)
  - many d’instructions
  - many addressing modes
  - many instructions read / write memory
  - few registers
  - examples: VAX, PDP-11, Motorola 68xxx, 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 labs and the project)
x86-64 architecture
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
• 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
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; it is a compiler
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
.text  # instructions follow
.globl main  # make main visible for ld
main:
movq $message, %rdi  # argument of puts
    call puts
movq $0, %rax  # return code 0
ret

.data  # data follow
message:
    .string "Hello, world!"  # 0-terminated string
assembling

> as hello.s -o hello.o

linking (gcc appelle ld)

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

(note: no need for -no-pie in salles info)

execution

> ./hello
Hello, world!
we can **disassemble** using `objdump`

```
> objdump -d hello.o
0000000000000000 <main>:
  0:  48 c7 c7 00 00 00 00  mov $0x0,%rdi
  7:  e8 00 00 00 00     callq c <main+0xc>
  c:  48 c7 c0 00 00 00 00 mov $0x0,%rax
 13:  c3               retq
```

we note that
- addresses for the string and `puts` are not yet known
- the code is located at address 0
we can also disassemble the executable

```
> objdump -d hello
0000000000004004e7 <main>:
    4004e7: 48 c7 c7 30 10 60 00 mov $0x601030,%rdi
    4004ee: e8 fd fe ff ff callq 4003f0 <puts@plt>
    4004f3: 48 c7 c0 00 00 00 00 mov $0x0,%rax
    4004fa: c3 retq
```

we now see

- an effective address for the string ($0x601030)
- an effective address for function puts ($0x4003f0)
- a program location at $0x4004e7
we note that the bytes of 0x00601030 are stored in memory in the order 30, 10, 60, 00

we say that the machine is **little-endian**

other architectures are **big-endian** or **bi-endian**

(reference: Jonathan Swift’s *Gulliver’s Travels*)
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 0x4004e7: 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
...
```
an alternative is Nemiver (installed in salles infos)

```
> nemiver hello
```
instruction set
<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%spl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• loading a constant into a register

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

• loading the address of a label into a register

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

• copying a register into another register

```assembly
movq %rax, %rbx  # rbx <- rax
```
• addition of two registers
  \[\text{addq} \ %rax, \ %rbx \quad \# \ rbx \leftarrow rbx + rax\]
  (similarly, \text{subq}, \text{imulq})

• addition of a register and a constant
  \[\text{addq} \ %rcx, \ %rbx \quad \# \ rcx \leftarrow rcx + 2\]

• particular case
  \[\text{incq} \ %rbx \quad \# \ rbx \leftarrow rbx+1\]
  (similarly, \text{decq})

• négation
  \[\text{negq} \ %rbx \quad \# \ rbx \leftarrow -rbx\]
• logical NOT

\[
\text{notq} \quad \%rax \quad \# \text{ rax } \leftarrow \text{not}(\text{rax})
\]

• AND, OR, XOR

\[
\begin{align*}
\text{orq} & \quad \%rbx, \%rcx \quad \# \text{ rcx } \leftarrow \text{or}(\text{rcx}, \text{rbx}) \\
\text{andq} & \quad \$0\text{xff}, \%rcx \quad \# \text{ efface les bits } \geq 8 \\
\text{xorq} & \quad \%rax, \%rax \quad \# \text{ met à zéro}
\end{align*}
\]
- **shift left (inserting zeros)**

  ```assembly
  salq $3, %rax    # 3 fois
  salq %cl, %rbx   # cl fois
  ```

- **arithmetic shift right (duplicating the sign bit)**

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

- **logical shift right (inserting zeros)**

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

- **rotation**

  ```assembly
  rolq $2, %rdi
  rorq $3, %rsi
  ```
the suffix `q` means a 64-bit operand (*quad words*)

other suffixes are allowed

<table>
<thead>
<tr>
<th>suffix</th>
<th>#bytes</th>
<th>#bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
<td>(byte)</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>(word)</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
<td>(long)</td>
</tr>
<tr>
<td>q</td>
<td>8</td>
<td>(quad)</td>
</tr>
</tbody>
</table>

`movb $42, %ah`
when operand sizes differ, one must indicate the extension mode

\[
\begin{align*}
\text{movzbq} & \ %al, \ %rdi \quad \# \text{with zeros extension} \\
\text{movswl} & \ %ax, \ %edi \quad \# \text{with sign extension}
\end{align*}
\]
an operand between parentheses means an **indirect addressing**
i.e. the data in memory at this address

\[
\text{movq} \quad $42, (%rax) \quad \# \text{mem}[\text{rax}] \leftarrow 42 \\
\text{incq} \quad (%rbx) \quad \# \text{mem}[\text{rbx}] \leftarrow \text{mem}[\text{rbx}] + 1
\]

note: the address may be a label

\[
\text{movq} \quad \%rbx, (x)
\]
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]
```
operation `lea` computes the effective address of the operand

\[ A(B, I, S) \]

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

Note: we can make use of it to perform arithmetic

```assembly
leaq (%rax,%rax,2), %rbx  # rbx <- 3*%rax
```
most operations set the **processor flags**, according to their outcome

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>the result is 0</td>
</tr>
<tr>
<td>CF</td>
<td>a carry was propagated beyond the most significant bit</td>
</tr>
<tr>
<td>SF</td>
<td>the result is negative</td>
</tr>
<tr>
<td>OF</td>
<td>arithmetic overflow (signed arith.)</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

(notable exception: **lea**)
three instructions can test the flags

- **conditional jump (jcc)**
  
  \[
  \text{jne \ label}
  \]

- **computes 1 (true) or 0 (false) (setcc)**
  
  \[
  \text{setge \ %bl}
  \]

- **conditional mov (cmovcc)**
  
  \[
  \text{cmovl \ %rax, \ %rbx}
  \]

<table>
<thead>
<tr>
<th>suffix</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>( z = 0 )</td>
</tr>
<tr>
<td>ne</td>
<td>( nz \neq 0 )</td>
</tr>
<tr>
<td>s</td>
<td>( &lt; 0 )</td>
</tr>
<tr>
<td>ns</td>
<td>( \geq 0 )</td>
</tr>
<tr>
<td>g</td>
<td>( &gt; ) signed</td>
</tr>
<tr>
<td>ge</td>
<td>( \geq ) signed</td>
</tr>
<tr>
<td>l</td>
<td>( &lt; ) signed</td>
</tr>
<tr>
<td>le</td>
<td>( \leq ) signed</td>
</tr>
<tr>
<td>a</td>
<td>( &gt; ) unsigned</td>
</tr>
<tr>
<td>ae</td>
<td>( \geq ) unsigned</td>
</tr>
<tr>
<td>b</td>
<td>( &lt; ) unsigned</td>
</tr>
<tr>
<td>be</td>
<td>( \leq ) unsigned</td>
</tr>
</tbody>
</table>
one can set the flags without storing the result anywhere, as if doing a subtraction or a logical AND

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

(beware of the direction!)

```assembly
testq  %rbx, %rax  # flags of AND(rax, rbx)
```
unconditional jump

• to a label

\[
\text{jmp} \quad \text{label}
\]

• to a computed address

\[
\text{jmp} \quad *\%rax
\]
the challenge of compilation

this is translating 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
⇒ registers cannot hold all the local variables
⇒ we need to allocate memory

yet function calls obey a *last-in first-out* mode, so we can use a **stack**
<table>
<thead>
<tr>
<th>stack</th>
<th>the <strong>stack</strong> is allocated at the top of the memory, and increases downwards; <code>%rsp</code> points to the top of the stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>dynamic data (which need to survive function calls) are allocated on the <strong>heap</strong> (possibly by a GC), at the bottom of the data zone, above static data</td>
</tr>
<tr>
<td>↑</td>
<td>this way, no collision between the stack and the heap (unless we run out of memory)</td>
</tr>
<tr>
<td>dynamic data (heap)</td>
<td></td>
</tr>
<tr>
<td>static data</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
</tr>
</tbody>
</table>

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

• pushing

\[
\begin{align*}
\text{pushq} &\quad \$42 \\
\text{pushq} &\quad \%rax
\end{align*}
\]

• popping

\[
\begin{align*}
\text{popq} &\quad \%rdi \\
\text{popq} &\quad (\%rbx)
\end{align*}
\]

element:

\[
\begin{align*}
\text{pushq} &\quad \$1 \\
\text{pushq} &\quad \$2 \\
\text{pushq} &\quad \$3 \\
\text{popq} &\quad \%rax
\end{align*}
\]

\[
\%rsp \rightarrow
\]

\[
\begin{align*}
\vdots \\
1 \\
2 \\
3 \\
\downarrow
\end{align*}
\]
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
two instructions for this purpose

instruction

\[
\text{call } g
\]

1. pushes the address of the next instruction on the stack
2. transfers control to address \text{g}

and instruction

\[
\text{ret}
\]

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

problem: any register used by $g$ is lost for $f$
	here are many solutions, but we typically resort to 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`, `%14` and `%r15` are callee-saved i.e. the callee must save them; 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
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

(for `hello.s` earlier, we skipped that; we were lucky!)
alignment

stack alignment may be performed explicitly

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

or indirectly

```assembly
f:   pushq %rbx  # we save %rbx
     ...
     ...
     ...  # because we use it here
     ...
     popq %rbx  # and we restore it
     ret
```
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, however, we must obey the calling conventions
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

```assembly
  call   callee
```
1. saves %rbp and set it, for instance with
   
   ```assembly
   pushq %rbp
   movq %rsp, %rbp
   ```

2. allocates its stack frame, for instance with
   
   ```assembly
   subq $48, %rsp
   ```

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

\[
%rbp \rightarrow \quad \text{old } %rbp
\]

\[
%rsp \rightarrow \quad \text{saved registers}
\]

\[
\text{argument 8} \quad \text{argument 7}
\]

\[
\quad \text{ret. addr.}
\]

\[
\quad \quad \downarrow
\]

\[
\quad \quad \text{local variables}
\]

%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

\[
\text{leave}
\]

that is equivalent to

\[
\begin{align*}
\text{movq} & \quad \%rbp, \%rsp \\
\text{popq} & \quad \%rbp
\end{align*}
\]

4. executes

\[
\text{ret}
\]
the caller, after the call

1. pops arguments 7, 8, ..., if any
2. restores the caller-saved registers, if needed
• 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
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));}
int t(int a, int b, int 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;
}

int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
this program computes the number of solutions to the \( N \)-queens problem

```
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 q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```
how does it work?

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

<table>
<thead>
<tr>
<th>integers</th>
<th>sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>a \cap b</td>
</tr>
<tr>
<td>a + b</td>
<td>a \cup b, when a \cap b = \emptyset</td>
</tr>
<tr>
<td>a - b</td>
<td>a \setminus b, when b \subseteq a</td>
</tr>
<tr>
<td>\sim a</td>
<td>\complement a</td>
</tr>
<tr>
<td>a &amp; -a</td>
<td>{min(a)}, when a \neq \emptyset</td>
</tr>
<tr>
<td>\sim(\sim 0 \ll n)</td>
<td>{0, 1, \ldots, n - 1}</td>
</tr>
<tr>
<td>a \times 2</td>
<td>{i + 1</td>
</tr>
<tr>
<td>a / 2</td>
<td>{i - 1</td>
</tr>
</tbody>
</table>
explaining a&-a

in two’s complement: \(-a = \sim a + 1\)

\[
\begin{align*}
a &= b_{n-1}b_{n-2} \ldots b_k10\ldots0 \\
\sim a &= \overline{b_{n-1}b_{n-2} \ldots b_k}01\ldots1 \\
-a &= \overline{b_{n-1}b_{n-2} \ldots b_k}10\ldots0 \\
a&-a &= 0 \ 0\ldots\ 010\ldots0
\end{align*}
\]

example:

\[
\begin{align*}
a &= 00001100 = 12 \\
-a &= 11110100 = -128 + 116 \\
a&-a &= 00000100
\end{align*}
\]
clarification: code with sets

```c
int t(a, b, c)
    f ← 1
    if a ≠ ∅
        e ← (a\b)\c
        f ← 0
    while e ≠ ∅
        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 $c$
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 q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
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
• locale 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

\[
\begin{array}{c}
\vdots \\
\text{ret. adr.} \\
%rax \ (f) \\
%rcx \ (e) \\
%r8 \ (d) \\
%rdx \ (c) \\
%rsi \ (b) \\
%rdi \ (a) \\
\end{array}
\]

%rsp →
allocating/deallocating the stack frame

```
t:                                     
  subq $48, %rsp
  ...                                  
  addq $48, %rsp
  ret
```
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        ...
    }
    return f;
}

movq $1, %rax  # f <- 1
testq %rdi, %rdi  # a = 0 ?
jz t
        ...

    t_return:
        addq $48, %rsp
        ret
when a \neq 0

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

note the use of a temporary register %r9 (not saved)
while (expr) {
    body
}

L1: ...
    compute expr into %rcx
    ...
    testq %rcx, %rcx
    jz L2
    ...
    body
    ...
    jmp L1
L2: ...
there are better options, though

while (expr) {
  body
}

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

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

\begin{verbatim}
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
\end{verbatim}
int main() {
    int q;
    scanf("%d", &q);
    ...
}

main:
    subq $8, %rsp  # alignment
    movq $input, %rdi
    movq $q, %rsi
    xorq %rax, %rax
    call scanf
    movq (q), %rcx
    ...

.data
input:
    .string "%d"
q:
    .quad 0
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
    addq   $8, %rsp
    ret
this code is not optimal

(for instance, we create a stack frame even when \( a = 0 \))

yet it is better than the code produced by gcc -O2 or by clang -O2

no reason to show off: we wrote an assembly code specific to this C program, manually, not a compiler!
• 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 simpler at https://godbolt.org/)

• now we have to automate all this
• *Computer Systems: A Programmer’s Perspective* (R. E. Bryant, D. R. O’Hallaron)
• its PDF appendix *x86-64 Machine-Level Programming*
• TD 1
  • manual compilation of C programs

• next lecture
  • abstract syntax
  • semantics
  • interpreter