École Polytechnique

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

\[
\begin{align*}
\text{bits} &= b_{n-1} b_{n-2} \ldots b_1 b_0 \\
\text{value} &= \sum_{i=0}^{n-1} b_i 2^i
\end{align*}
\]

<table>
<thead>
<tr>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<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>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>111...110</td>
<td>2^n - 2</td>
</tr>
<tr>
<td>111...111</td>
<td>2^n - 1</td>
</tr>
</tbody>
</table>

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

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

\[
\text{bits} = b_{n-1}b_{n-2}\ldots b_1b_0 \\
\text{value} = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i2^i
\]

example:
\[
\begin{align*}
11010110_2 &= -128 + 86 \\
&= -42
\end{align*}
\]

<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>
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)
the machine provide several operations

- logical, or bitwise, operations (AND, OR, XOR, NOT)
- shift operations
- arithmetic operations (addition, subtraction, multiplication, etc.)
### logical operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>00101001</td>
</tr>
<tr>
<td>NOT x</td>
<td>11010110</td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>00101001</td>
</tr>
<tr>
<td>y</td>
<td>01101100</td>
</tr>
<tr>
<td>x AND y</td>
<td>00101000</td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>00101001</td>
</tr>
<tr>
<td>y</td>
<td>01101100</td>
</tr>
<tr>
<td>x OR y</td>
<td>01101101</td>
</tr>
<tr>
<td>XOR</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>00101001</td>
</tr>
<tr>
<td>y</td>
<td>01101100</td>
</tr>
<tr>
<td>x XOR y</td>
<td>01000101</td>
</tr>
</tbody>
</table>
shift operation

• logical shift left (inserts least significant zeros)

\[
\begin{array}{c}
| b_{n-3} | \ldots | b_1 | b_0 | 0 | 0 |
\end{array}
\leftarrow
\begin{array}{c}
| b_{n-3} | \ldots | b_0 | 0 | 0 |
\end{array}
\]

(<< in Java, lsl in OCaml)

• logical shift right (inserts most significant zeros)

\[
\begin{array}{c}
| 0 | 0 | b_{n-1} | \ldots | b_3 | b_2 |
\end{array}
\rightarrow
\begin{array}{c}
| 0 | 0 | b_{n-1} | \ldots | b_3 | b_2 |
\end{array}
\]

(>>> in Java, lsr in OCaml)

• arithmetic shift right (duplicates the sign bit)

\[
\begin{array}{c}
| b_{n-1} | b_{n-1} | b_{n-1} | \ldots | b_3 | b_2 |
\end{array}
\rightarrow
\begin{array}{c}
| b_{n-1} | b_{n-1} | b_{n-1} | \ldots | b_3 | b_2 |
\end{array}
\]

(>> 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 \(2^{233}\) possible states
  • contains data and instructions
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 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

**CPU**

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

**RAM**

**Instruction:**

```
48 c7 c0 2a 00 00 00
```

**Decoding:**

```
<table>
<thead>
<tr>
<th>movq</th>
<th>%rax</th>
</tr>
</thead>
</table>

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
which architecture for this course?

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
\textbf{x86} a family of compatible architectures

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

\textbf{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 (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:
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
assembling

> as hello.s -o hello.o

linking (gcc calls 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**

```bash
> 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
we can also disassemble the executable

```
objdump -d hello
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Function</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>401126</td>
<td>push %rbp</td>
<td></td>
<td>0x401126</td>
</tr>
<tr>
<td>401127</td>
<td>mov %rsp,%rbp</td>
<td></td>
<td>0x401127</td>
</tr>
<tr>
<td>40112a</td>
<td>mov $0x404030,%rdi</td>
<td></td>
<td>0x40112a</td>
</tr>
<tr>
<td>401131</td>
<td>call 401030 <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
<td>0x401131</td>
</tr>
<tr>
<td>401136</td>
<td>mov $0x0,%rax</td>
<td></td>
<td>0x401136</td>
</tr>
<tr>
<td>40113d</td>
<td>pop %rbp</td>
<td></td>
<td>0x40113d</td>
</tr>
<tr>
<td>40113e</td>
<td>ret</td>
<td></td>
<td>0x40113e</td>
</tr>
</tbody>
</table>

we now see

- an effective address for the string ($0x404030)
- an effective address for function `puts` ($0x401030)
- a program location at $0x401126
we 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*)
a step-by-step execution is possible using gdb (the GNU debugger)

```bash
> gcc -g -no-pie hello.s -o hello 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
...
an alternative is Nemiver (installed in salles infos)

> nemiver hello
instruction set
constants, addresses, copies

• loading a constant into a register

\[
\begin{align*}
\text{movq} & \quad $0x2a, \%rax \quad \# \ rax \leftarrow 42 \\
\text{movq} & \quad $-12, \%rdi
\end{align*}
\]

• loading the address of a label into a register

\[
\text{movq} \quad $\text{label}, \%rdi
\]

• copying a register into another register

\[
\text{movq} \quad \%rax, \%rbx \quad \# \ rbx \leftarrow rax
\]
• 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

```
egq    %rbx    # rbx <- -rbx
```
• logical NOT

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

• AND, OR, XOR

```
orq %rbx, %rcx # rcx <- or(rcx, rbx)
andq $0xff, %rcx # erases bits >= 8
xorq %rax, %rax # zeroes %rax
```
- **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
  ```
the suffix \texttt{q} means a 64-bit operand (\emph{quad words})

other suffixes are allowed

\begin{center}
\begin{tabular}{llc}
suffix & \#bytes & \texttt{(}\textit{bytes})\\
\texttt{b} & 1 & \texttt{(byte)} \\
\texttt{w} & 2 & \texttt{(word)} \\
\texttt{l} & 4 & \texttt{(long)} \\
\texttt{q} & 8 & \texttt{(quad)} \\
\end{tabular}
\end{center}

\texttt{movb} $42$, \%ah
when operand sizes differ, one must indicate the \textit{extension mode}

\begin{itemize}
\item \texttt{movzbq \%al, \%rdi} # with zeros extension
\item \texttt{movswl \%ax, \%edi} # with sign extension
\end{itemize}
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)
```
operations do not allow several memory accesses

\begin{verbatim}
addq (%rax), (%rbx)
\end{verbatim}

Error: too many memory references for ‘add’

one has to use a temporary register

\begin{verbatim}
movq (%rax), %rcx
addq %rcx, (%rbx)
\end{verbatim}
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)

\[ \text{movq } -8(\%rax,\%rdi,4), \%rbx \quad \# \text{ rbx <- mem[-8+rax+4*rdi]} \]
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
```
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)**
  
  ```
  jne label
  ```

- **computes 1 (true) or 0 (false) (setcc)**
  
  ```
  setge %bl
  ```

- **conditional mov (cmovcc)**
  
  ```
  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 ≠ 0</td>
</tr>
<tr>
<td>s</td>
<td>s &lt; 0</td>
</tr>
<tr>
<td>ns</td>
<td>s ≥ 0</td>
</tr>
<tr>
<td>g</td>
<td>s &gt; signed</td>
</tr>
<tr>
<td>ge</td>
<td>s ≥ signed</td>
</tr>
<tr>
<td>l</td>
<td>s &lt; signed</td>
</tr>
<tr>
<td>le</td>
<td>s ≤ signed</td>
</tr>
<tr>
<td>a</td>
<td>a &gt; unsigned</td>
</tr>
<tr>
<td>ae</td>
<td>a ≥ unsigned</td>
</tr>
<tr>
<td>b</td>
<td>b &lt; unsigned</td>
</tr>
<tr>
<td>be</td>
<td>b ≤ unsigned</td>
</tr>
</tbody>
</table>
one can set the flags without storing the result anywhere, as if doing a subtraction or a logical AND

\[
\text{cmpq} \quad \%rbx, \%rax \quad \# \text{flags of } \text{rax} - \text{rbx}
\]

(beware of the direction!)

\[
\text{testq} \quad \%rbx, \%rax \quad \# \text{flags of } \text{AND(rax, rbx)}
\]
• 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

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
**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**
the stack

| stack       |
| ↓           |
| ↑           |
| dynamic data (heap) |
| static data |
| code        |

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

dynamic data (which need to survive function calls) are allocated on the heap (possibly by a GC), at the bottom of the data zone, above static data

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

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

• popping

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

deep copy of `%rdi` and `%rax`

example:

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

%rsp → ```assembly
   ↓
0 1 2 3
```
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, `%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
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
```
... 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
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`
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
```
the callee, at the beginning of the call

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

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

2. allocates its stack frame, for instance with

\[
\text{subq} \quad \$48, \%rsp
\]

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

<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>({\text{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*2</td>
<td>({i + 1 \mid i \in a}), written (S(a))</td>
</tr>
<tr>
<td>a/2</td>
<td>({i - 1 \mid i \in a \land i \neq 0}), written (P(a))</td>
</tr>
</tbody>
</table>
explaining \texttt{a} \& \texttt{-a}

in two’s complement: \(-a = \overline{a} + 1\)

\[
\begin{align*}
a &= b_{n-1}b_{n-2} \ldots b_k10 \ldots 0 \\
\overline{a} &= \overline{b_{n-1}b_{n-2} \ldots \overline{b_k}01} \ldots 1 \\
-a &= \overline{b_{n-1}b_{n-2} \ldots \overline{b_k}10} \ldots 0 \\
a \&\& a = 00001100 = 12 \\
-a &= 11110100 = -128 + 116 \\
a \&\& a = 00000100
\end{align*}
\]
int $t(a, b, c)$
\[
f \leftarrow 1
\]
\[
\text{if } a \neq \emptyset
\]
\[
e \leftarrow (a \setminus b) \setminus c
\]
\[
f \leftarrow 0
\]
\[
\text{while } e \neq \emptyset
\]
\[
d \leftarrow \min(e)
\]
\[
f \leftarrow f + t(a \setminus \{d\}, S(b \cup \{d\}), P(c \cup \{d\}))
\]
\[
e \leftarrow e \setminus \{d\}
\]
\[
\text{return } f
\]

int $queens(n)$
\[
\text{return } t(\{0, 1, \ldots, n - 1\}, \emptyset, \emptyset)$
meaning of \( a, b, \) and \( c \)

\( a \) = columns to be filled = 11100101

\( b \) = forbidden positions because of ascending diagonals = 01101000

\( c \) = forbidden positions because of descending diagonals = 00001001

\( a \& \sim b \& \sim c \) = positions to consider = 10000100
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
- 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

<table>
<thead>
<tr>
<th>%rbp</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret. adr.</td>
<td>%rbp</td>
</tr>
<tr>
<td>saved %rbp</td>
<td>%rax (f)</td>
</tr>
<tr>
<td></td>
<td>%rcx (e)</td>
</tr>
<tr>
<td></td>
<td>%r8 (d)</td>
</tr>
<tr>
<td></td>
<td>%rdx (c)</td>
</tr>
<tr>
<td></td>
<td>%rsi (b)</td>
</tr>
<tr>
<td></td>
<td>%rdi (a)</td>
</tr>
</tbody>
</table>

%rsp →
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        ...
    }
    return f;
}
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 $a \neq 0$

```c
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: ...

compiling the loop
there are better options, though

```assembly
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)
while (d=e&−e) {
    ...
}
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

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
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
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!
• 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*

• *Notes on x86-64 programming* by Andrew Tolmach (available on the course website)
• lab 1
  • manual compilation of C programs

• next lecture **Thursday 19**
  • abstract syntax
  • semantics
  • interpreter