

DB 2

07 – Expression Evaluation

Summer 2018

Torsten Grust
Universität Tübingen, Germany

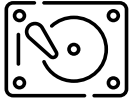
1 | Q_6 — Expression Evaluation

For a large class of queries, the **CPU effort to evaluate (complex) expressions** may easily match the time spent for I/O and data access:

```
SELECT t.a * 3 - t.a * 2 AS a,  
       t.a - power(10, t.c) AS diff,  
       ceil(t.c / log(2)) AS bits  
FROM   ternary AS t;
```

Iterate over rows t , access required fields (here: $t.a$, $t.c$), evaluate (multiple) expressions per row, construct resulting row.

Using **EXPLAIN** on Q_6 : **INSERT**



EXPLAIN VERBOSE

```
SELECT t.a * 3 - t.a * 2      AS a,  
        t.a - power(10, t.c) AS diff,  
        ceil(t.c / log(2))   AS bits  
FROM   ternary AS t;
```

QUERY PLAN

Seq Scan on public.ternary t (cost=0.00..40.00 rows=1000 width=20)

Output: ((a * 3) - (a * 2)),

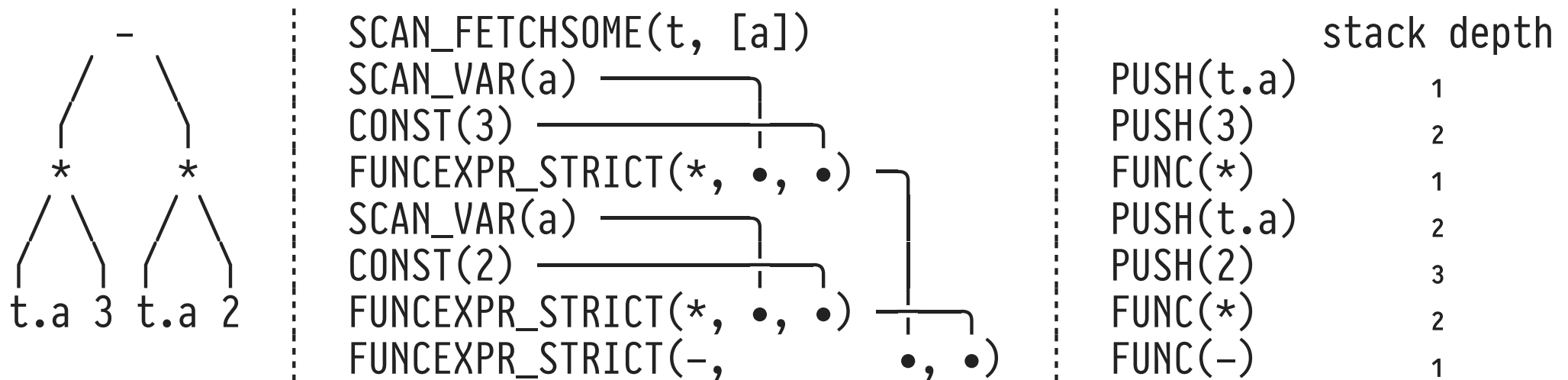
((a)::double precision - power('10'::double precision, c)),

ceil((c / '0.301029995663981'::double precision))

- Expressions have been parenthesized, simplified, and annotated with type casts as required by SQL semantics.

Internal Representations of $t.a * 3 - t.a * 2$

- DBMSs—just like interpreters and compilers—**transform expressions into internal representations** that facilitate simplification and evaluation:



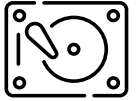
- Postorder traversal of expression tree to obtain a linearized “program”. Arg slots (\bullet) or stack push/pop.

PostgreSQL implements a **threaded interpreter** over linearized expressions (middle column of previous slide):

program[]	dispatch[]	opcode implementations
<pre>ip->[[opc=1,res=>•,...]] [[opc=3,res=>•,const=42]] ⋮</pre>	<pre>1 → SCAN_VAR: 2 → ... 3 ↘ ↘ → CONST: ⋮</pre>	<pre>...; ip = ip + 1; goto dispatch[ip->opc]; ip->res = ip->const; ip = ip + 1; goto dispatch[ip->opc];</pre>
linearized expression	dispatch table	

- Note: `ip`: instruction pointer, `opc`: operation code.
- Relies on support for *computed goto* (e.g., common in C).

Expression Interpretation Overhead



Overhead of expression interpretation has been found to be **massive** in DBMS (cf. the threaded interpretation vs. machine code for `t.a * 2`).

- Field access and interpretation in *hot query code path*, rediscovers same row structure and follows same opcode pointers for every row processed. Wasteful.
- 💡 Invest in **just-in-time (JIT) compilation** of expression program into machine code once, benefit for all subsequent rows.
 - **N.B.:** LLVM-based support for JIT compilation of expressions being added to PostgreSQL v11 as we speak.

2 : Q_6 — Expression Evaluation

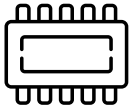


```
SELECT t.a * 3 - t.a * 2    AS a,  
        t.a - power(10, t.c) AS diff,  
        ceil(t.c / log(2))   AS bits  
FROM   ternary AS t;
```

MonetDB compiles expressions into sequences of MAL operations. Like data processing, expression evaluation is column-oriented (as opposed to row-by-row).

- We will find that this vector-based evaluation mode fits modern CPU architecture particularly well.

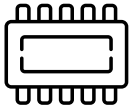
Using **EXPLAIN** on Q_6 : **DELETE**



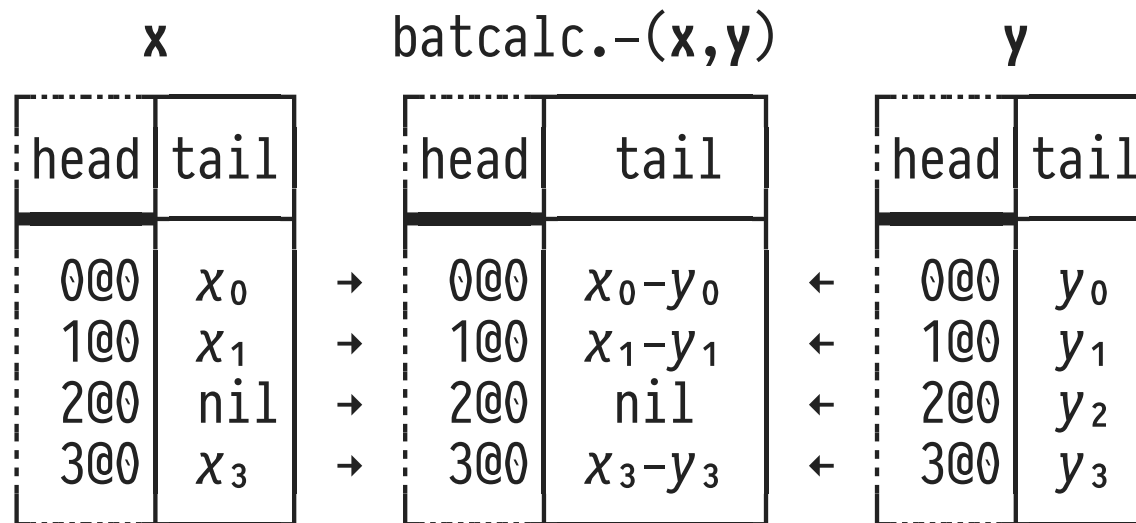
```
sql> EXPLAIN SELECT t.a * 3 - t.a * 2 AS a,  
      ceil(t.c / log(2)) AS bits  
      FROM ternary AS t;  
  
:  
ternary :bat[:oid] := sql.tid(sql, "sys", "ternary");  
c0      :bat[:dbl] := sql.bind(sql, "sys", "ternary", "c", 0:int);  
c       := algebra.projection(ternary, c0);  
e1      :bat[:dbl] := batcalc./(c, 0.6931471805599453:dbl);  
e2      :bat[:dbl] := batmath.ceil(e1);           ◀ result column bits  
a0      :bat[:int] := sql.bind(sql, "sys", "ternary", "a", 0:int);  
a       := algebra.projection(ternary, a0);  
e3      :bat[:lng] := batcalc.lng(a);             ◀ cast to type lng  
e4      :bat[:lng] := batcalc.*(e3, 3:bte);  
e5      :bat[:lng] := batcalc.*(e3, 2:bte);  
e6      :bat[:lng] := batcalc.-(e4, e5);          ◀ result column a  
:
```

- MAL ops `batcalc.⊗` accept two BATs or one BAT + one scalar (like `2:bte`, `3:bte`, `0.693...:dbl` \equiv `log(2)`).

Column-Based “Zip” Semantics

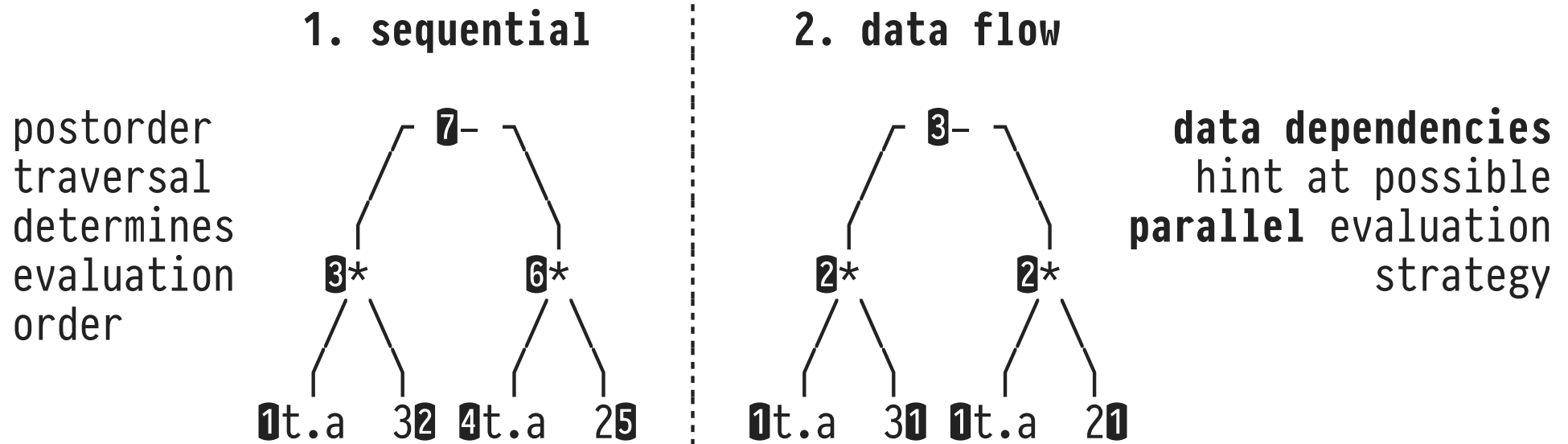
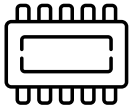


Operators `batcalc.⊗` merge the tails of two synchronized BATs using binary operator \otimes , yields a new BAT:



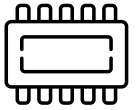
- `batcalc.⊗` contains checks for arithmetic exceptions (overflow, divide by 0). Also: `nil ⊗ x = x ⊗ nil = nil`.

MAL: Sequential Execution vs. Data Flow



1. Order of assignment to temporary result BATs e_i follows postorder traversal of expression tree.
2. Spawn CPU threads to evaluate data-independent subexpressions in // (see MonetDB's [dataflow](#) optimizer).

batcalc.⊗: Column-Based Operator Implementations (1)

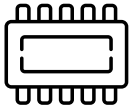


MonetDB supplies type- and \otimes -specific implementations of MAL operations (code generation via C preprocessor macros):

```
/* batcalc.-(left:bat[:lng], right:bat[:lng]):bat[:lng] */
                                     ▲
int i, j, k;
int nils = 0;

for (i = start, j = start*1, k = start; k < end; i += 1, j += 1, k += 1) {
    /* nil checking */
    if (is_lng_nil(left[i]) || is_lng_nil(right[j])) {
        result[k] = lng_nil;
        nils++;
    } else {
        /* omitted: overflow checking (abort on error or emit nil) */
        result[k] = left[i] - right[j];
    }
}
```

batcalc.⊗: Column-Based Operator Implementations (2)



MonetDB supplies type- and \otimes -specific implementations of MAL operations (code generation via C preprocessor macros):

```
/* batcalc.-(left:bat[:lng], right:lng):bat[:lng] */
int i, j, k;
int nils = 0;

for (i = start, j = start*0, k = start; k < end; i += 1, j += 0, k += 1) {
    /* nil checking */
    if (is_lng_nil(left[i]) || is_lng_nil(right[j])) {
        result[k] = lng_nil;
        nils++;
    } else {
        /* omitted: overflow checking (abort on error or emit nil) */
        result[k] = left[i] - right[j];
    }
}
```

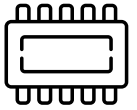
3 | Column-Based Operators vs. Expression Interpretation

Expression evaluation through column-based operator and row-wise interpretation compared:

Column-Based (MonetDB)	Row-Wise (PostgreSQL)
zero degrees of freedom instruction locality optimizable tight loops <ul style="list-style-type: none">• loop pipelining• blocking• loop unrolling data parallelism full materialization	variable-width rows w/ fields of various types computed goto, long code paths complex control flow, code in many functions <ul style="list-style-type: none">• unpredictable branches focus on single row row-by-row result generation

- Compilers **optimize tight code loops** inside MAL operators.
- CPUs offer wide registers and instructions to exploit **data //ism** (SIMD: *single instruction, multiple data*).

Compiling Tight Loops (cf. MAL Operators)



Inspect Intel® x86 code generated by LLVM's C compiler `clang` for MonetDB's routine `BATcalcsb` (`batcalc.-`), simplified:

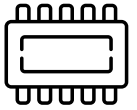
```
#define SIZE 1024

void BATcalcsb(int *left, int *right, int *result)
{
    int i, j, k;

    for (i = j = k = 0; k < SIZE; i += 1, j += 1, k += 1) {
        result[k] = left[i] - right[j];
    }
}
```

- Arrays `left`, `right`/`result` represent input/output BATs.

Assembly Code for Simple Tight Loop



Uses `clang` (options `-O2 -fno-vectorize -fno-unroll-loops`).

- Register assignment:

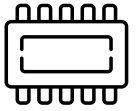
`left: %rdi, right: %rsi, result: %rdx, i/j/k: %rax`

BATcalcsb:

```
    movq $-4096, %rax          # 4096 = 1024 * 4 (≡ size of int)
loop:
    movl 4096(%rdi,%rax), %ecx  # %ecx ←32 mem[4096 + %rdi + %rax]
    subl 4096(%rsi,%rax), %ecx  # %ecx ←32 %ecx -32 mem[4096 + %rsi + %rax]
    movl %ecx, 4096(%rdx,%rax)  # mem[4096 + %rdx + %rax] ←32 %ecx
    addq $4, %rax              # 4096 / 4 = 1024 loop iterations
    jne loop                   # exit if %rax = 0
    retq
```

- **N.B.:** One loop exit test per array element computed.

(Explicit) Loop Unrolling



- Manually perform **loop unrolling** to
 1. improve the ratio (*useful work*) / (*loop exit test*),
 2. expose independent work that may be executed in `//`:

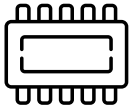
```
void BATcalcsb(int *left, int *right, int *result)
{
    int i, j, k;

    for (i = j = k = 0; k < SIZE; i += 4, j += 4, k += 4) {
        result[k] = left[i] - right[j];
        result[k+1] = left[i+1] - right[j+1];
        result[k+2] = left[i+2] - right[j+2];
        result[k+3] = left[i+3] - right[j+3];
    }
}
```

Diagram illustrating loop unrolling. The code shows a loop where `i`, `j`, and `k` are incremented by 4 in each iteration. The four assignments inside the loop are grouped by a large right-facing curly brace, with the text "independent, execute in any order or even in `//`" to its right. Three downward-pointing arrows are positioned above the loop header, pointing to the increments `i += 4`, `j += 4`, and `k += 4` respectively.

- **N.B.:** Needs code to handle the case `SIZE mod 4 \neq 0`.

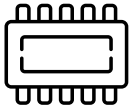
Loop Unrolling



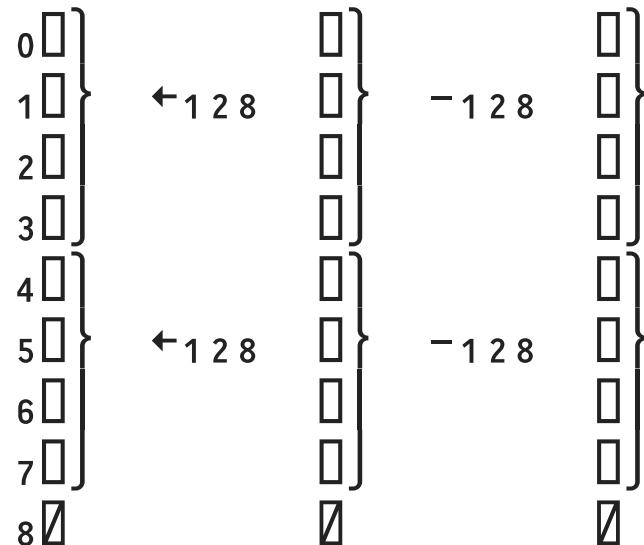
Compiler `clang` (options `-O2 -fno-vectorize -funroll-loops`)
unrolls four loop iterations (easy for CPU to //ize):

```
BATcalcsb:
    movq $-1024, %rax                # i/j/k
loop:
    movl 4096(%rdi,%rax,4), %ecx      # %ecx ←32 left[i]
    subl 4096(%rsi,%rax,4), %ecx      # %ecx ←32 %ecx -32 right[j]
    movl %ecx, 4096(%rdx,%rax,4)      # result[k] ←32 %ecx
    movl 4100(%rdi,%rax,4), %ecx      # %ecx ←32 left[i+1]
    subl 4100(%rsi,%rax,4), %ecx      # %ecx ←32 %ecx -32 right[j+1]
    movl %ecx, 4100(%rdx,%rax,4)      # result[k+1] ←32 %ecx
    movl 4104(%rdi,%rax,4), %ecx      # :
    subl 4104(%rsi,%rax,4), %ecx
    movl %ecx, 4104(%rdx,%rax,4)
    movl 4108(%rdi,%rax,4), %ecx
    subl 4108(%rsi,%rax,4), %ecx
    movl %ecx, 4108(%rdx,%rax,4)
    addq $4, %rax                    # 1024 / 4 = 256 loop iterations
    jne loop                         # exit if %rax = 0
    retq
```

Data-Parallelism Through SIMD



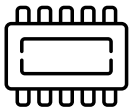
result[] left[] right[]



- Read/compute/write four array elements (of width 4×32 bits = 128 bits) at a time in **data-parallel** fashion.
- Relies on SIMD register and instructions (e.g., Intel® SSE registers `%xmmi` and instruction `move double quad word`)

- ⚠ Requires care if
 - arrays `result[]` and `left[]/right[]` overlap in memory,
 - residual array elements (see `∅`) are to be processed.

Data-Parallelism Through SIMD (Prelude)

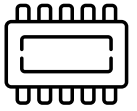


Compiler `clang` (options `-O2 -fvectorize`) uses SIMD registers and instructions. Here: prelude, checking for array overlap:

BATcalcsb:

```
leaq 4096(%rdi), %rax # left: %rdi-□□□□□□-4096+%rdi ≡ %rax
cmpq %rdx, %rax      # result: %rdx-■■■■■■■■ } □□□□□□→ → → →
seta %r9b             # %r9b ← true, if %rax > %rdx, i.e. { ■■■■■■■■
leaq 4096(%rdx), %rcx # result: %rdx-■■■■■■■■-4096+%rdx ≡ %rcx
cmpq %rdi, %rcx      # left: %rdi-□□□□□□ } □□□□□□
seta %r10b            # %r10b ← true, if %rcx > %rdi, i.e. { ■■■■■■■■→ → → →
leaq 4096(%rsi), %rax # :
cmpq %rdx, %rax
seta %al
cmpq %rsi, %rcx
seta %r8b
testb %r10b, %r9b     # %r9b ∧ %r10b = true, if left[] and result[] overlap
jne slow              # if so, choose "slow" non-SIMD unrolled code variant
andb %r8b, %al         # %r8b ∧ %al = true, if right[] and result[] overlap
jne slow              # if so, choose "slow" variant
:
```

Data-Parallelism Through SIMD (Main Loop)



Process 16 elements per iteration (SIMD + 2 loops unrolled):

```

:
movq $-1024, %rax
loop:
movdqu 4096(%rdi,%rax,4), %xmm0 # %xmm0 ←128 left[i+0...i+3]
movdqu 4112(%rdi,%rax,4), %xmm1 # %xmm1 ←128 left[i+4...i+7]
movdqu 4096(%rsi,%rax,4), %xmm2 # %xmm2 ←128 right[i+0...i+3]
psubd %xmm2, %xmm0             # %xmm0 ←128 %xmm0 -128 %xmm2
movdqu 4112(%rsi,%rax,4), %xmm2 # %xmm2 ←128 right[i+4...i+7]
psubd %xmm2, %xmm1             # %xmm1 ←128 %xmm1 -128 %xmm2
movdqu %xmm0, 4096(%rdx,%rax,4) # result[i+0...i+3] ←128 %xmm0
movdqu %xmm1, 4112(%rdx,%rax,4) # result[i+4...i+7] ←128 %xmm1
movdqu 4128(%rdi,%rax,4), %xmm0 # %xmm0 ←128 left[i+8 ...i+11]
movdqu 4144(%rdi,%rax,4), %xmm1 # %xmm1 ←128 left[i+12...i+15]
movdqu 4128(%rsi,%rax,4), %xmm2 # %xmm2 ←128 right[i+8...i+11]
psubd %xmm2, %xmm0             # %xmm0 ←128 %xmm0 -128 %xmm2
movdqu 4144(%rsi,%rax,4), %xmm2 # %xmm2 ←128 right[i+12...i+15]
movdqu %xmm0, 4128(%rdx,%rax,4) # %xmm1 ←128 %xmm1 -128 %xmm2
psubd %xmm2, %xmm1             # result[i+8 ...i+11] ←128 %xmm0
movdqu %xmm1, 4144(%rdx,%rax,4) # result[i+12...i+15] ←128 %xmm1
addq $16, %rax                 # 1024 / 16 = 64 iterations
jne loop                       # exit if %rax = 0
:
```

4 × 32 bits = 128 bits wide

.....

loop #n

loop #n+1