last time

out-of-order execution and instruction queues

the data flow model idea

- graph of operations linked by depedencies

latency bound — need to finish longest dependency chain

multiple accumulators — expose more parallelism

divide by constant

reusing address calculations in loops

vector instructions

modern processors have registers that hold "vector" of values

example: X86-64 has 128-bit registers

- 4 ints or 4 floats or 2 doubles or …

128-bit registers named `%xmm0` through `%xmm15`

instructions that act on all values in register

- vector instructions or SIMD (single instruction, multiple data) instructions

extra copies of ALUs only accessed by vector instructions

example vector instruction

`paddd %xmm0, %xmm1` (packed add dword (32-bit))

Suppose registers contain (interpreted as 4 ints)

%xmm0: [1, 2, 3, 4]
%xmm1: [5, 6, 7, 8]

Result will be:

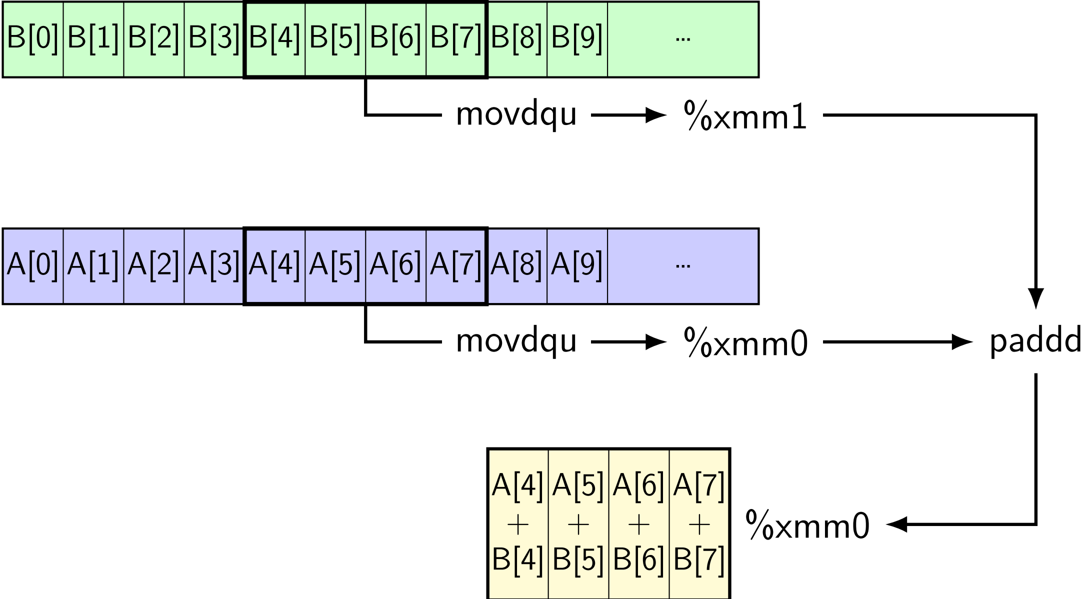
%xmm1: [6, 8, 10, 12]

vector instructions

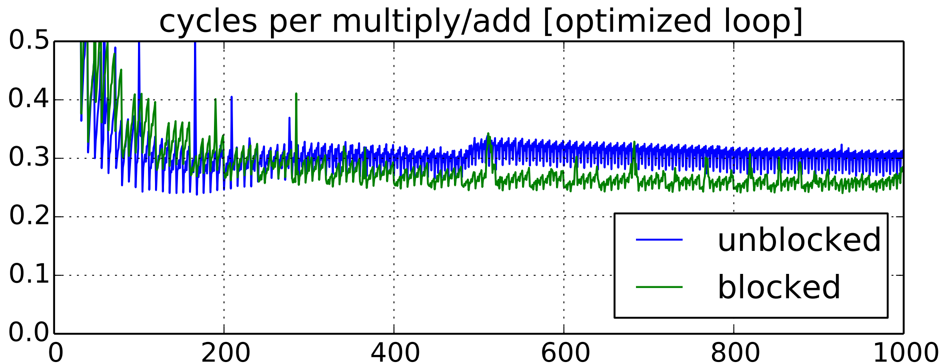
```
void add(int * restrict a, int * restrict b) {
    for (int i = 0; i < 128; ++i)
        a[i] += b[i];
}
```

```
add:
        xorl    %eax, %eax              // init. loop counter
the_loop:
        movdqu  (%rdi,%rax), %xmm0      // load 4 from A
        movdqu  (%rsi,%rax), %xmm1      // load 4 from B
        paddd   %xmm1, %xmm0            // add 4 elements!
        movups  %xmm0, (%rdi,%rax)      // store 4 in A
        addq    $16, %rax               // +4 ints = +16
        cmpq    $512, %rax              // 512 = 4 * 128
        jne     the_loop
        rep ret
```

vector add picture



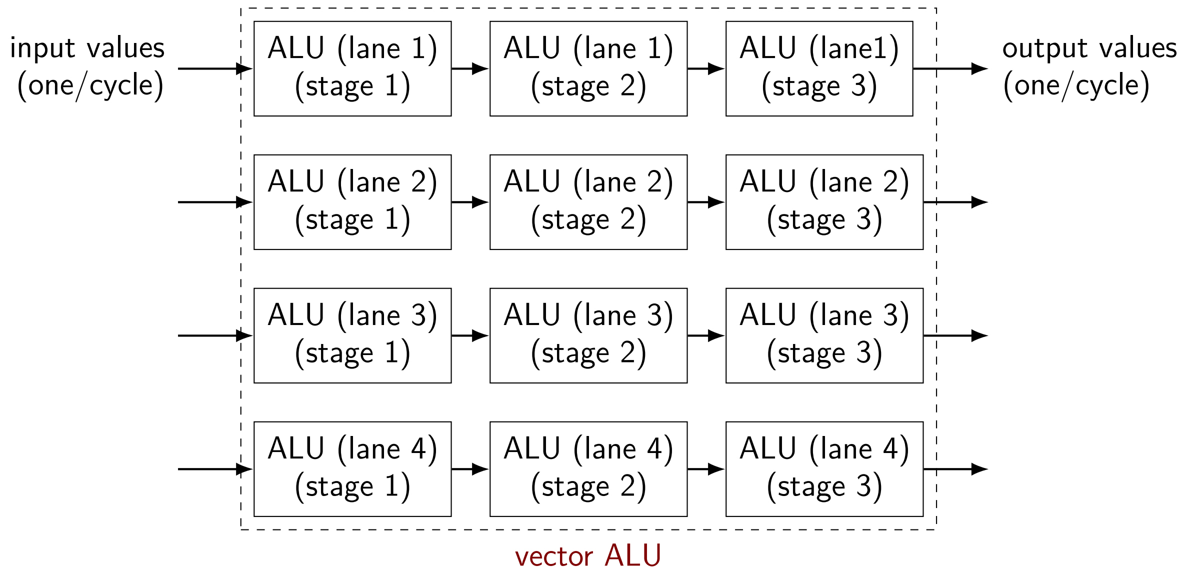
wiggles on prior graphs



variance from this optimization

8 elements in vector, so multiples of 8 easier

one view of vector functional units



why vector instructions?

lots of logic not dedicated to computation

- instruction queue
- reorder buffer
- instruction fetch
- branch prediction
- …

adding vector instructions — little extra control logic

…but a lot more computational capacity

vector instructions and compilers

compilers can sometimes figure out how to use vector instructions
    (and have gotten much, much better at it over the past decade)

but easily messsed up:
- by aliasing
- by conditionals
- by some operation with no vector instruction
- ...

fickle compiler vectorization (1)

GCC 7.2 and Clang 5.0 generate vector instructions for this:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

but not:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                B[i * N + j] += A[i * N + k] * A[j * N + k];
}
```

fickle compiler vectorization (2)

Clang 5.0.0 generates vector instructions for this:

```
void foo(int N, unsigned int *A, unsigned int *B) {
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

but not: (probably bug?)

```
void foo(long N, unsigned int *A, unsigned int *B) {
    for (long k = 0; k < N; ++k)
        for (long i = 0; i < N; ++i)
            for (long j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

vector intrinsics

if compiler doesn't work...

could write vector instruction assembly by hand

second option: "intrinsic functions"

C functions that compile to particular instructions

vector intrinsics: add example

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 4) {
     // "si128" --> 128 bit integer
     // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
     __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
     // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
     __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

     // add four 32-bit integers
     // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
     __m128i sums = _mm_add_epi32(a_values, b_values);

     // {a[i], a[i+1], a[i+2], a[i+3]} = sums
     _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

special type `__m128i` — "128 bits of integers"
other types: `__m128` (floats), `__m128d` (doubles)

```
void vect
  for (int i = 0; i < 128; i += 4) {
    // "si128" --> 128 bit integer
    // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
    __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

functions to store/load

si128 means "128-bit integer value"

u for "unaligned" (otherwise, pointer address must be multiple of 16)

```
    // "si128" --> 128 bit integer
    // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
    __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 4) {
    // "si128"
    // a_values                                    +3]}
    __m128i a_v                                   28i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m128i sums = _mm_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

function to add

`epi32` means “4 32-bit integers”

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 2) {
    // a_values = {a[i], a[i+1]} (2 x 64 bits)
     __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1]} (2 x 64 bits)
     __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);
     // add two 64-bit integers: paddq %xmm0, %xmm1
     // sums = {a[i] + b[i], a[i+1] + b[i+1]}
     __m128i sums = _mm_add_epi64(a_values, b_values);
     // {a[i], a[i+1]} = sums
     _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 2) {
    // a_values = {a[i], a[i+1]} (2 x 64 bits)
     __m128i a_values = _mm_loadu_si128((__m128i*) &a[i]);
    // b_values = {b[i], b[i+1]} (2 x 64 bits)
     __m128i b_values = _mm_loadu_si128((__m128i*) &b[i]);
     // add two 64-bit integers: paddq %xmm0, %xmm1
     // sums = {a[i] + b[i], a[i+1] + b[i+1]}
     __m128i sums = _mm_add_epi64(a_values, b_values);
     // {a[i], a[i+1]} = sums
     _mm_storeu_si128((__m128i*) &a[i], sums);
  }
}
```

recall: square

```
void square(unsigned int *A, unsigned int *B) {
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

square unrolled

```
void square(unsigned int *A, unsigned int *B) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; j += 4) {
        /* goal: vectorize this */
        B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
        B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
        B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
        B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
      }
}
```

handy intrinsic functions for square

`_mm_set1_epi32` — load four copies of a 32-bit value into a 128-bit value

- instructions generated vary; one example: `movq` + `pshufd`

`_mm_mullo_epi32` — multiply four pairs of 32-bit values, give lowest 32-bits of results

- generates `pmulld`

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

---

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements from B
Bij = _mm_loadu_si128(&B[i * N + j + 0]);
... // manipulate vector here
// store four elements into B
_mm_storeu_si128((__m128i*) &B[i * N + j + 0], Bij);
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements from A
Akj = _mm_loadu_si128(&A[k * N + j + 0]);
... // multiply each by A[i * N + k] here
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements starting with A[k * n + j]
Akj = _mm_loadu_si128(&A[k * N + j + 0]);
// load four copies of A[i * N + k]
Aik = _mm_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm_mullo_epi32(Aik, Akj);
```

vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
Bij = _mm_add_epi32(Bij, multiply_results);
// store back results
_mm_storeu_si128(..., Bij);
```

square vectorized

```
__m128i Bij, Akj, Aik, Aik_times_Akj;

// Bij = {$B_{i,j}$, $B_{i,j+1}$, $B_{i,j+2}$, $B_{i,j+3}$}
Bij = _mm_loadu_si128((__m128i*) &B[i * N + j]);
// Akj = {$A_{k,j}$, $A_{k,j+1}$, $A_{k,j+2}$, $A_{k,j+3}$}
Akj = _mm_loadu_si128((__m128i*) &A[k * N + j]);

// Aik = {$A_{i,k}$, $A_{i,k}$, $A_{i,k}$, $A_{i,k}$}
Aik = _mm_set1_epi32(A[i * N + k]);

// Aik_times_Akj = {$A_{i,k} \times A_{k,j}$, $A_{i,k} \times A_{k,j+1}$, $A_{i,k} \times A_{k,j+2}$, $A_{i,k} \times A_{k,j+3}$}
Aik_times_Akj = _mm_mullo_epi32(Aij, Akj);

// Bij= {$B_{i,j} + A_{i,k} \times A_{k,j}$, $B_{i,j+1} + A_{i,k} \times A_{k,j+1}$, ...}
Bij = _mm_add_epi32(Bij, Aik_times_Akj);

// store Bij into B
_mm_storeu_si128((__m128i*) &B[i * N + j], Bij);
```

shuffles/swizzles

```
/* x = 32-bit values: {10, 20, 30, 40} */
__m128i x = _mm_setr_epi32(10, 20, 30, 40);
/* y = {20, 10, 40, 30} */
/* __MM_SHUFFLE macro lists indices to select in reverse order
__m128i y = _mm_shuffle_epi32(x, _MM_SHUFFLE(2, 3, 0, 1));
```

```
/* x = 8-bit values: {10, 20, 30, 40, 50, ..., 160} */
__m128i x = _mm_setr_epi8(10, 20, 30, 40, 50, 60, 70, 80,
                          90, 100, 110, 120, 130, 140, 150, 16
/* y = {30, 30, 30, 30, 40, 40, 40, 40, 10, 10, 10, 10, 20, 20
__m128i y = _mm_shuffle_epi8(x,
    __mm_setr_epi8(2, 2, 2, 2, 3, 3, 3, 3, 0, 0, 0, 0, 1, 1, 1
```

more misc operations

many more variations/special cases of shuffles

combining values from different vectors into one vector

arithmetic *within* a vector

extracting parts of vectors

...

alternate vector interfaces

intrinsics functions/assembly aren't the only way to write vector code

e.g. GCC vector extensions: more like normal C code

- types for each kind of vector
- write + instead of `_mm_add_epi32`

e.g. CUDA (GPUs): looks like writing multithreaded code, but each thread is vector "lane"

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

this class: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

- supported on lab machines
- 128-bit vectors

latest X86 processors: AVX, AVX2, AVX-512

- 256-bit and 512-bit vectors

also other ISAs have these: e.g. NEON on ARM, MSA on MIPS, AltiVec/VMX on POWER, …

other vector instructions features

SSE pretty limiting

other vector instruction sets often more featureful:
(and require more sophisticated HW support)

better conditional handling

better variable-length vectors

ability to load/store non-contiguous values

some of these features in some versions of AVX

smooth preview

smooth — take average of every 3x3 block of pixels in an image

pixels: 4 one-byte values

...but we need shorts to do computation of average

intermediate values > 255

vectors of 16 bit values $\rightarrow$ two pixels/vector regiser

adding pixels

```
red = pixel[0].red + pixel[1].red + ...;
green = pixel[0].green + pixel[1].green + ...;
blue = pixel[0].blue + pixel[1].blue + ...;
alpha = pixel[0].blue + pixel[1].blue + ...;
```

```
/* vector of 16-bit values, last 64-bits unused? */
combined_parts = _mm_add_epi16(
    _mm_add_epi16(pixel_zero_parts, pixel_one_parts),
    ...);
```

optimizing real programs

spend effort where it matters

e.g. 90% of program time spent reading files, but optimize computation?

e.g. 90% of program time spent in routine A, but optimize B?

profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: `perf`

perf usage

sampling profiler

- stops periodically, takes a look at what's running

`perf record OPTIONS program`

- example OPTIONS:
- `-F 200` — record 200/second
- `--call-graph=dwarf` — record stack traces

`perf report` or `perf annotate`

children/self

“children” — samples in function or things it called

“self” — samples in function alone

demo

other profiling techniques

count number of times each function is called

not sampling — exact counts, but higher overhead

might give less insight into amount of time

tuning optimizations

biggest factor: how fast is it actually

setup a benchmark

- make sure it's realistic (right size? uses answer? etc.)

compare the alternatives

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

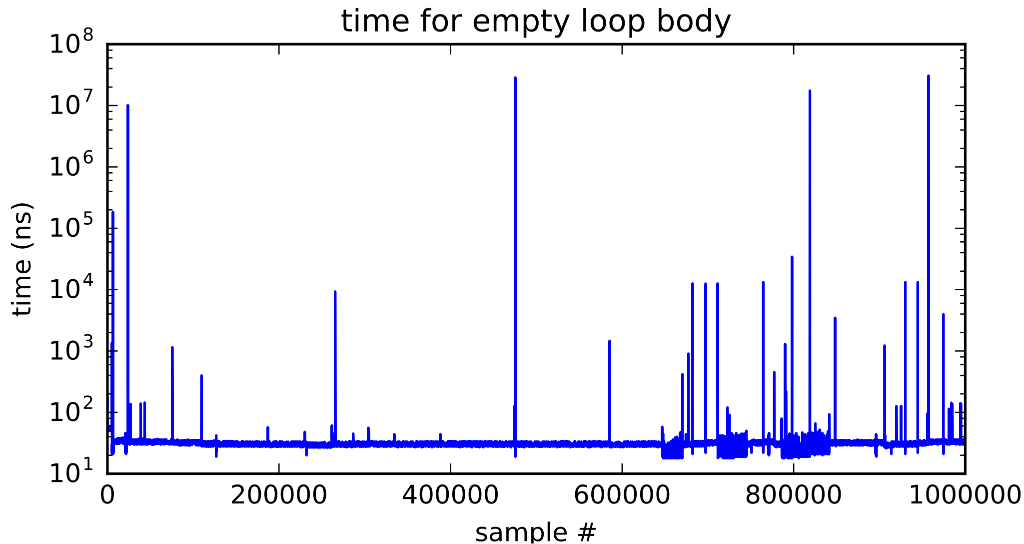
If I run this on a lab machine, can you still use it?
...if the machine only has one core?

timing nothing

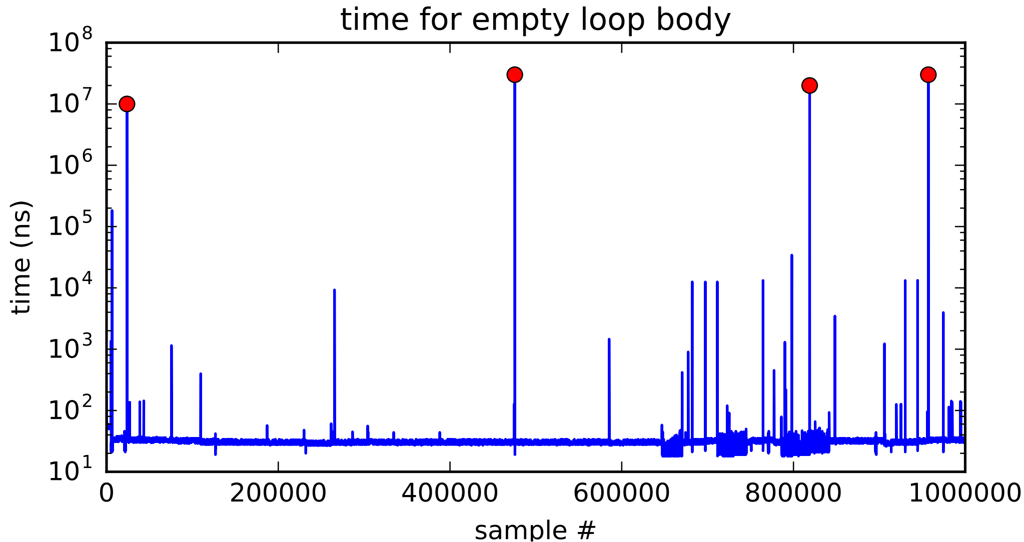
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

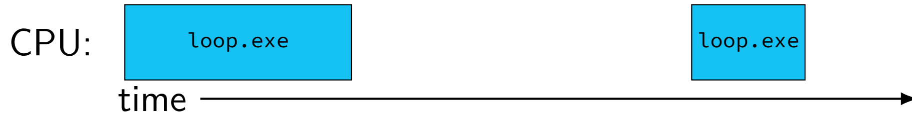
doing nothing on a busy system



doing nothing on a busy system



time multiplexing



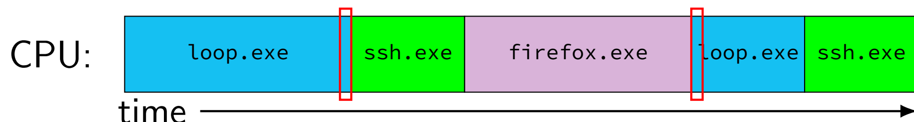
time multiplexing

```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing




```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really

time multiplexing really

exception happens

return from exception

OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

saves old program counter, registers somewhere

sets new registers, jumps to new program counter

called context switch

- saved information called context

context

all registers values

`%rax` `%rbx`, ..., `%rsp`, ...

condition codes

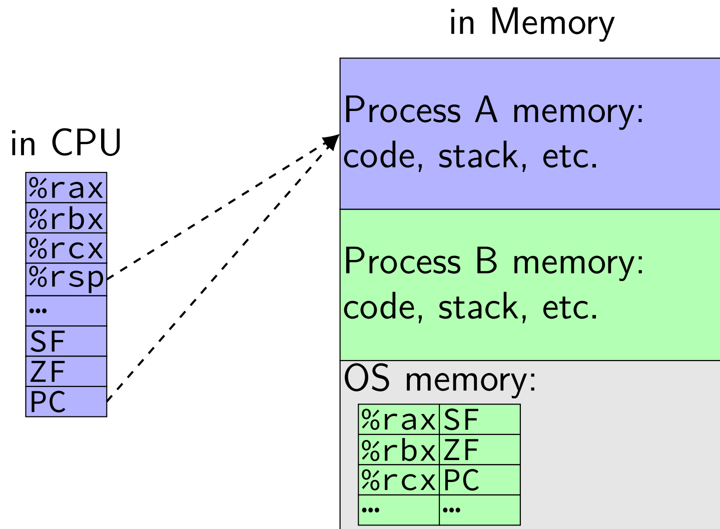
program counter

i.e. all visible state in your CPU except memory

context switch pseudocode

```
context_switch(last, next):
  copy_preexception_pc last->pc
  mov rax,last->rax
  mov rcx, last->rcx
  mov rdx, last->rdx
  ...
  mov next->rdx, rdx
  mov next->rcx, rcx
  mov next->rax, rax
  jmp next->pc
```

contexts (A running)



contexts (B running)