data level parallelism / exceptions 1

last time (1)

PRIME+PROBE

- attacker fill cache set(s) with attacker data
- let victim run, use cache set(s)
- measure speed of accessing attacker data → which cache sets used

cache coherency

- multiple cores with own caches → inconsistent versions?
- solution: invalidate or update *all other caches* on write
- glossed over details: who has a copy? always need to send invalidate? etc.

last time (2)

FLUSH+RELOAD

- Intel CLFLUSH instruction: invalidate address in all caches (on all CPUs)
- attacker does CLFLUSH(part of shared array)
- let victim run, possible use part of shared array
- measure speed of accessing shared array → was it used after flush

data level parallelism

- one instruction: do multiple copies of same thing (SIMD)
- hardware support: wide ('vector') registers holding array of values
- hardware support: multi-*lane* ALUs
- do more operations/cycle without much extra control logic
- sometimes compilers use these instructions automatically
- otherwise... *intrinsics* to help compiler use new instructions

unvectorized add (original)

```
unsigned int A[512], B[512];
...
for (int i = 0; i < N; i += 1) {
    A[i] = A[i] + B[i];
}
```

unvectorized add (unrolled)

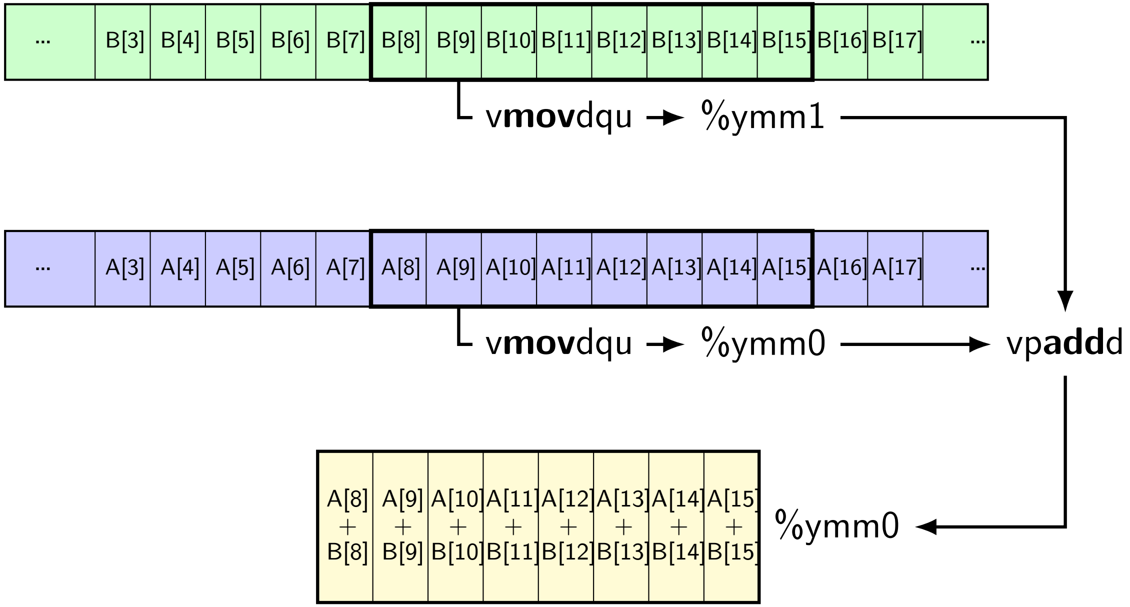
```
unsigned int A[512], B[512];
...
for (int i = 0; i < 512; i += 8) {
    A[i+0] = A[i+0] + B[i+0];
    A[i+1] = A[i+1] + B[i+1];
    A[i+2] = A[i+2] + B[i+2];
    A[i+3] = A[i+3] + B[i+3];
    A[i+4] = A[i+4] + B[i+4];
    A[i+5] = A[i+5] + B[i+5];
    A[i+6] = A[i+6] + B[i+6];
    A[i+7] = A[i+7] + B[i+7];
}
```

goal: use SIMD add instruction to do all 8 adds above

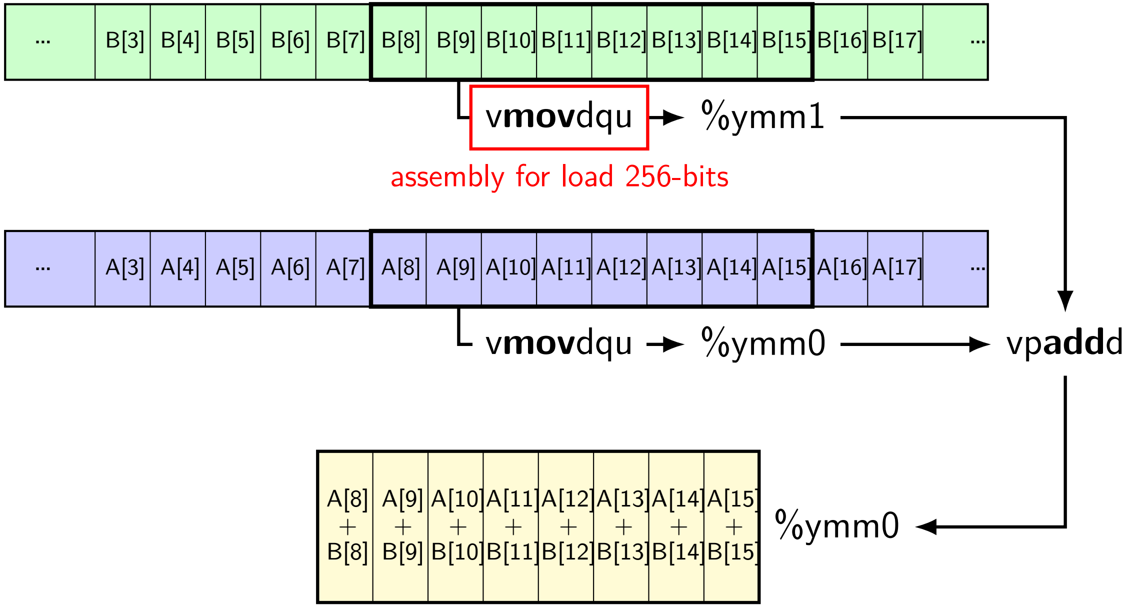
desired assembly

```
  xor %rax, %rax
the_loop:
  vmovdqu A(%rax), %ymm0        /* load 256 bits of A into ymm0
  vmovdqu B(%rax), %ymm1        /* load 256 bits of B into ymm1
  vpaddd %ymm1, %ymm0, %ymm0    /* ymm1 + ymm0 -> ymm0 */
  vmovdqu %ymm0, A(%rax)  /* store ymm0 into A */
  addq $32, %rax                /* increment index by 32 bytes *
  cmpq $2048, %rax              /* offset < 2048 (= 512 * 4) byt
  jne the_loop
```

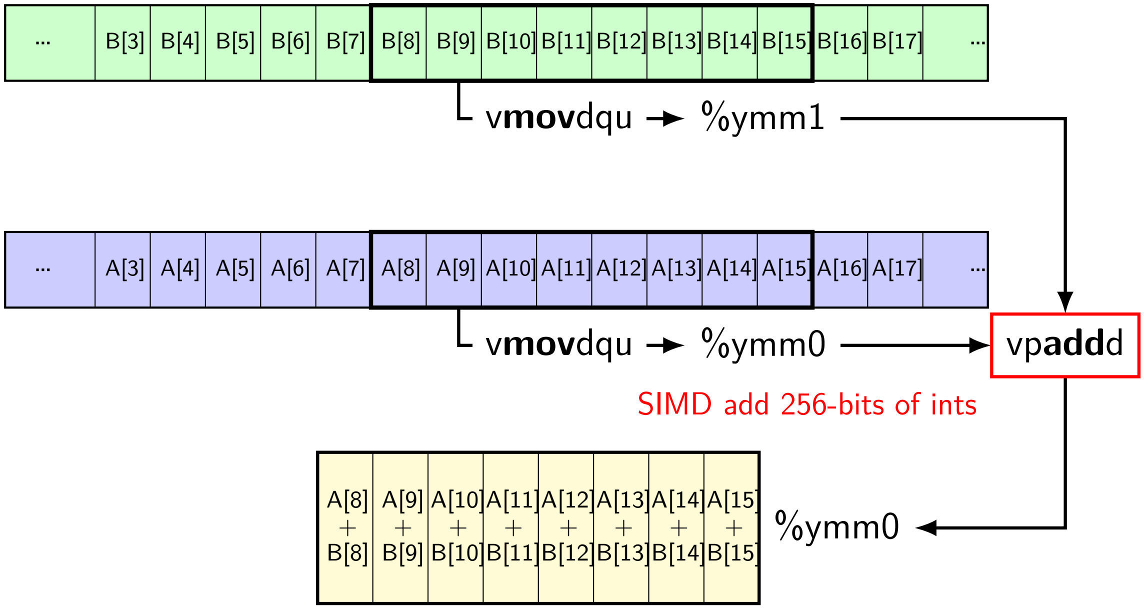
vector add picture (A[x] = A[x] + B[x])



vector add picture (A[x] = A[x] + B[x])

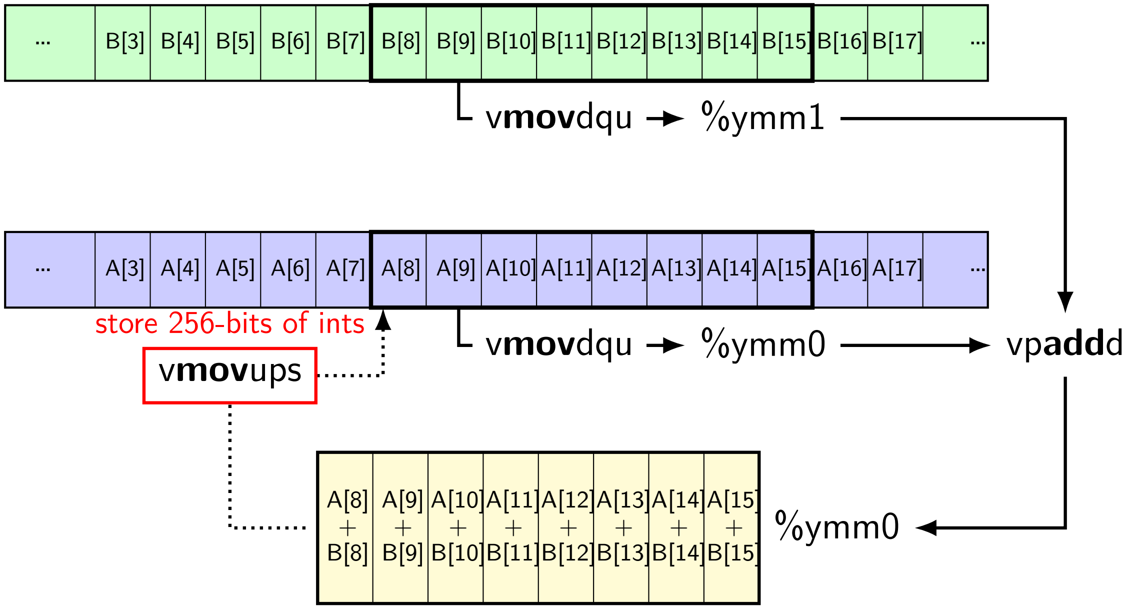


vector add picture (A[x] = A[x] + B[x])



...
B[3] B[4] B[5] B[6] B[7] B[8] B[9] B[10] B[11] B[12] B[13] B[14] B[15] B[16] B[17]
...
vmovdqu
%ymm1
...
A[3] A[4] A[5] A[6] A[7] A[8] A[9] A[10] A[11] A[12] A[13] A[14] A[15] A[16] A[17]
...
vmovdqu
%ymm0
vpaddd
SIMD add 256-bits of ints
A[8] + B[8]
A[9] + B[9]
A[10] + B[10]
A[11] + B[11]
A[12] + B[12]
A[13] + B[13]
A[14] + B[14]
A[15] + B[15]
%ymm0

vector add picture (A[x] = A[x] + B[x])



vector intrinsics: add example

```
int A[512], B[512];

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

special type `__m256i` — “256 bits of integers”
other types: `__m256` (floats), `__m128d` (doubles)

```
int A[512

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

functions to store/load

si256 means “256-bit integer value”

u for “unaligned” (otherwise, pointer address must be multiple of 32)

```
i

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

```
int A[512], B[512];

for (int i = 0;
  // "si256" -->
  // a_values =                                   8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

function to add

epi32 means “8 32-bit integers”

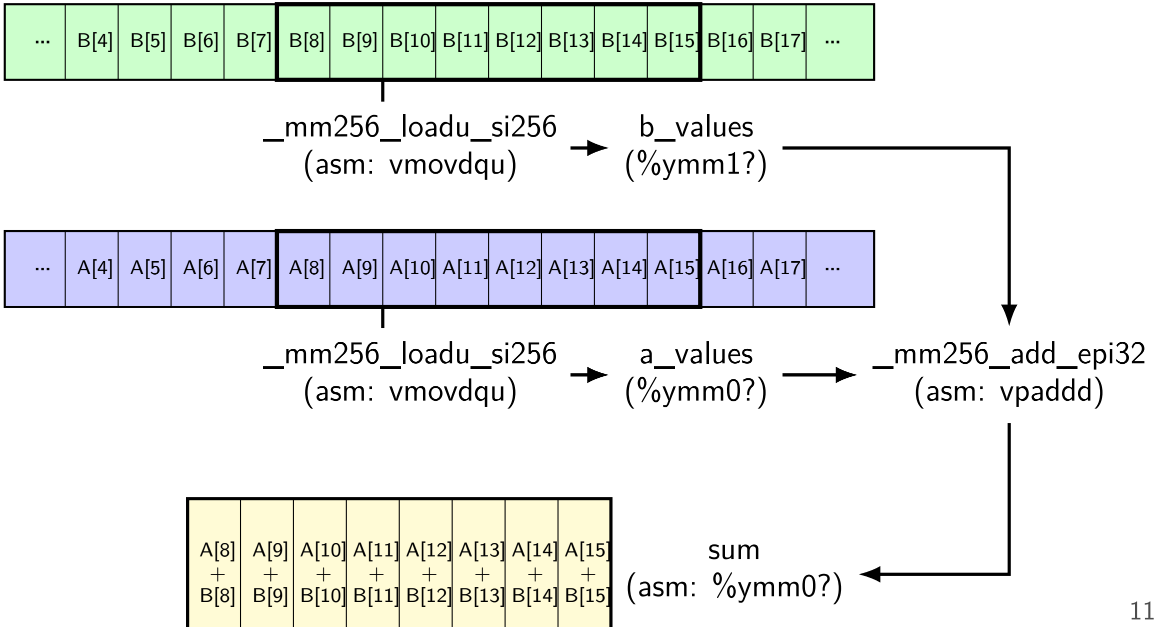
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (4 x 64 bits)
   __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);
   // add four 64-bit integers: vpaddq %ymm0, %ymm1
   // sums = {A[i] + B[i], A[i+1] + B[i+1], ...}
   __m256i sums = _mm256_add_epi64(a_values, b_values);
   // {A[i], A[i+1], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

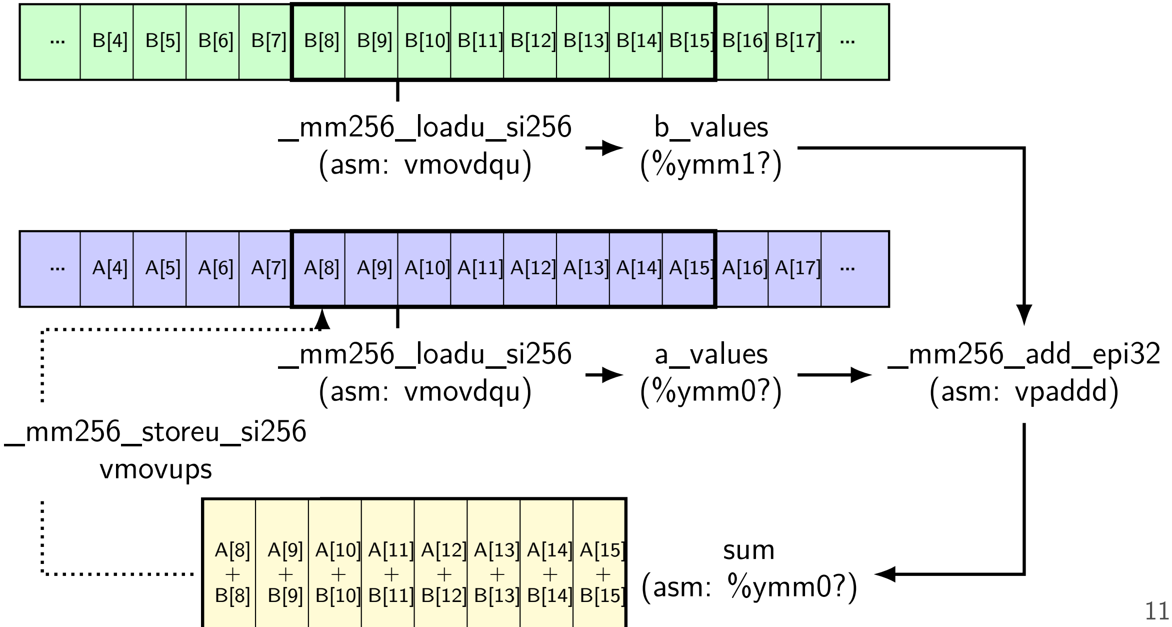
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (4 x 64 bits)
   __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);
   // add four 64-bit integers: vpaddq %ymm0, %ymm1
   // sums = {A[i] + B[i], A[i+1] + B[i+1], ...}
   __m256i sums = _mm256_add_epi64(a_values, b_values);
   // {A[i], A[i+1], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector add picture (intrinsics)



vector add picture (intrinsics)



128-bit version, too

history: 256-bit vectors added in extension called AVX (c. 2011)

before: 128-bit vectors added in extension called SSE (c. 1999)

128-bit intrinsics exist, too:

- `__m256i` becomes `__m128i`
- `_mm256_add_epi32` becomes `_mm_add_epi32`
- `_mm256_loadu_si256` becomes `_mm_loadu_si128`

matrix multiply

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

(simple version, no cache blocking, no avoiding aliasing beteeen C, B, A,…)

matmul unrolled

```
void matmul(unsigned int *A, unsigned int *B, unsigned int *C) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; j += 8) {
        /* goal: vectorize this */
        C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
        C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
        C[i * N + j + 2] += A[i * N + k] * B[k * N + j + 2];
        C[i * N + j + 3] += A[i * N + k] * B[k * N + j + 3];
        C[i * N + j + 4] += A[i * N + k] * B[k * N + j + 4];
        C[i * N + j + 5] += A[i * N + k] * B[k * N + j + 5];
        C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
        C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
      }
}
```

(NB: would probably also want to do cache blocking…)

handy intrinsic functions for matmul

`_mm256_set1_epi32` — load eight copies of a 32-bit value into a 256-bit value

- instructions generated vary; one example: `vmovd` + `vpbroadcastd`

`_mm256_mullo_epi32` — multiply eight pairs of 32-bit values, give lowest 32-bits of results

- generates `vpmulld`

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from C
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j + 0]);
... // manipulate vector here
// store eight elements into C
_mm_storeu_si256((__m256i*) &C[i * N + j + 0], Cij);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from B
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j + 0]);
... // multiply each by B[i * N + k] here
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements starting with B[k * n + j]
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j + 0]);
// load eight copies of A[i * N + k]
Aik = _mm256_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm256_mullo_epi32(Aik, Bkj);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
Cij = _mm256_add_epi32(Cij, multiply_results);
// store back results
_mm256_storeu_si256(..., Cij);
```

matmul vectorized

```
__m256i Cij, Bkj, Aik, Aik_times_Bkj;

// Cij = {$C_{i,j}$, $C_{i,j+1}$, $C_{i,j+2}$, ..., $C_{i,j+7}$}
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j]);
// Bkj = {$B_{k,j}$, $B_{k,j+1}$, $B_{k,j+2}$, ..., $B_{k,j+7}$}
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j]);

// Aik = {$A_{i,k}$, $A_{i,k}$, ..., $A_{i,k}$}
Aik = _mm256_set1_epi32(A[i * N + k]);

// Aik_times_Bkj = {$A_{i,k} \times B_{k,j}$, $A_{i,k} \times B_{k,j+1}$, $A_{i,k} \times B_{k,j+2}$, ..., $A_{i,k} \times B_{k,j+7}$}
Aik_times_Bkj = _mm256_mullo_epi32(Aij, Bkj);

// Cij= {$C_{i,j} + A_{i,k} \times B_{k,j}$, $C_{i,j+1} + A_{i,k} \times B_{k,j+1}$, ...}
Cij = _mm256_add_epi32(Cij, Aik_times_Bkj);

// store Cij into C
_mm256_storeu_si256((__m256i*) &C[i * N + j], Cij);
```

moving values in vectors?

sometimes values aren't in the right place in vector

example:

have: [1, 2, 3, 4]

want: [3, 4, 1, 2]

there are instructions/intrinsics for doing this

called shuffling/swizzling/permute/...

sometimes might need combination of them

worst-case: could rearrange on stack..., I guess

example shuffling operation (1)

goal: [1, 2, 3, 4] to [3, 4, 1, 2] (64-bit values)

```
/* x = {1, 2, 3, 4} */
__m256i x = _mm256_setr_epi64x(1, 2, 3, 4);
__m256i result = _mm256_permute4x64_epi64(
        x,
        /* index 2, then 3, then 0, then 1 */
        2 | (3 << 2) | (0 << 4) | (1 << 6)
        /* could also write _MM_SHUFFLE(1, 0, 3, 2) */
    );
/* result = {3, 4, 1, 2} */
```

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

early versions (128-bit vectors): SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

- 128-bit vectors

this class (256-bit): AVX, AVX2

not this class (512+-bit): AVX-512

- 512-bit vectors

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

GPUs are essentially vector-instruction-specialized CPUs

other vector interfaces

intrinsics (our assignments) one way

some alternate programming interfaces

- have compiler do more work than intrinsics

e.g. CUDA, OpenCL, GCC's vector instructions

other vector instructions features

more flexible vector instruction features:

- invented in the 1990s
- often present in GPUs and being rediscovered by modern ISAs

reasonable conditional handling

better variable-length vectors

ability to load/store non-contiguous values

some of these features in AVX2/AVX512

an infinite loop

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

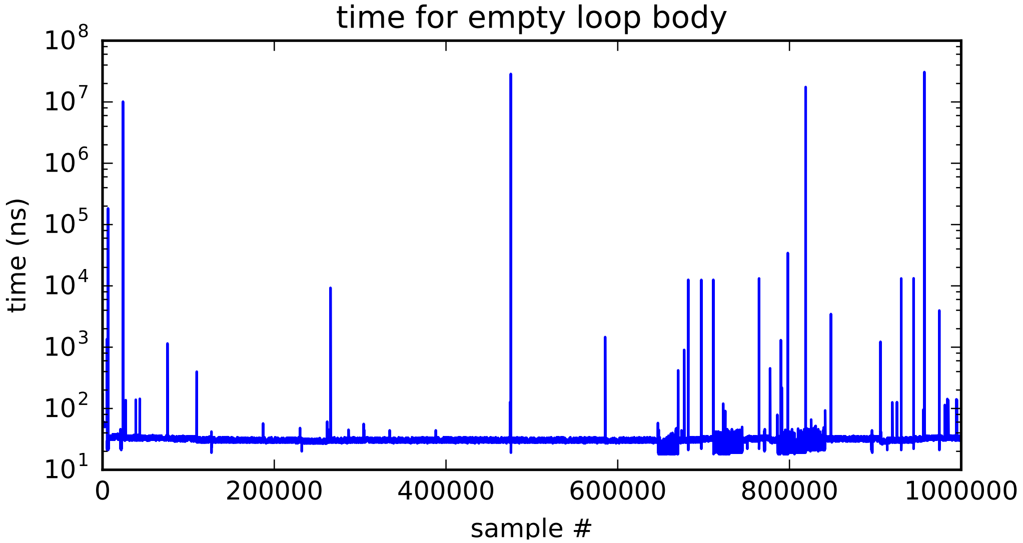
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

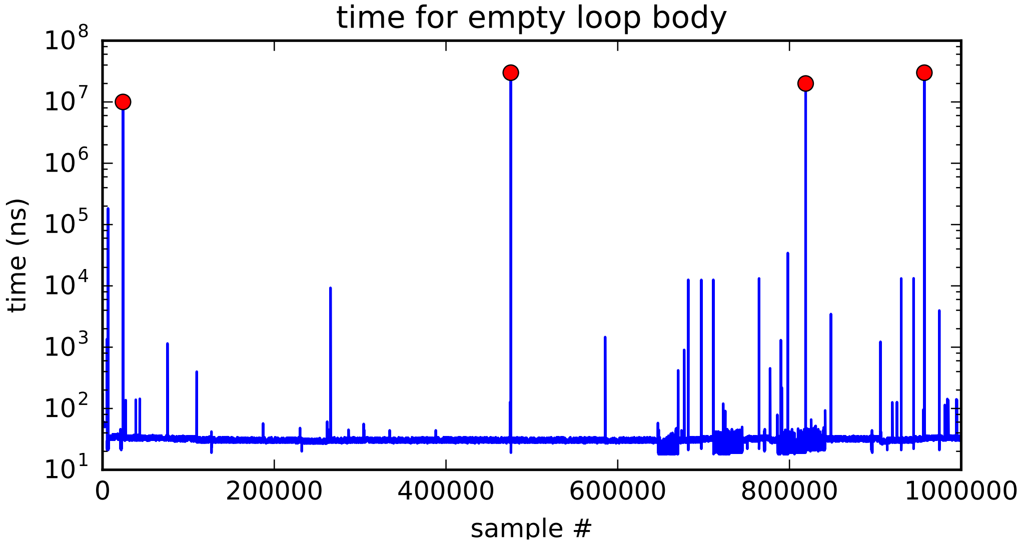
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

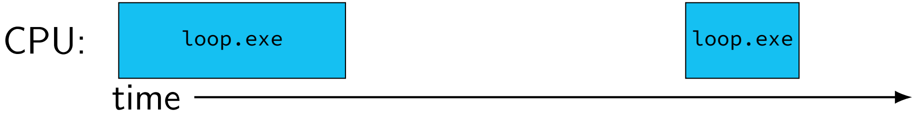
doing nothing on a busy system



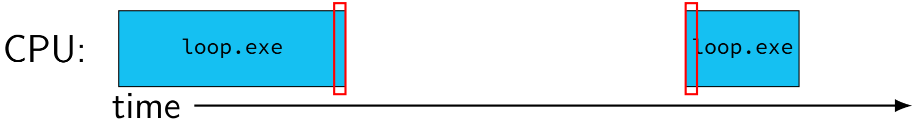
doing nothing on a busy system



time multiplexing



time multiplexing

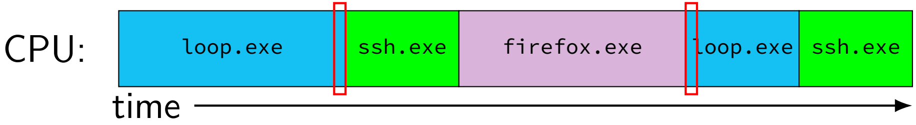


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

——— million cycle delay ———

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

time multiplexing

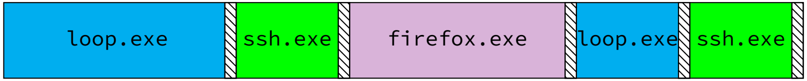


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

—— million cycle delay ——

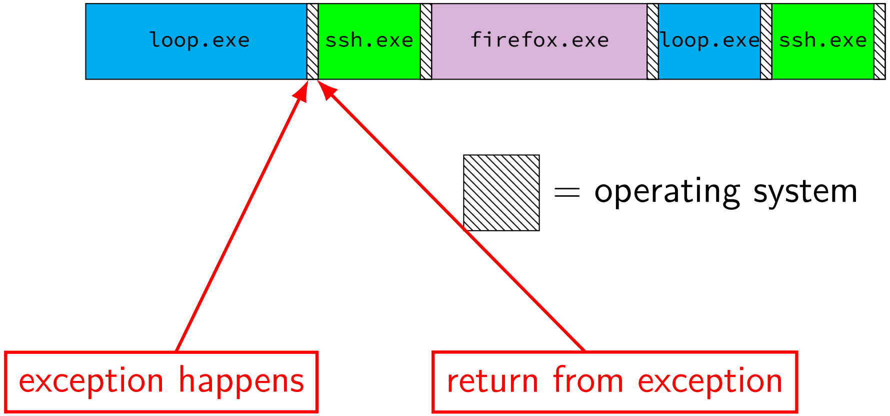
```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really



▨ = operating system

time multiplexing really



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

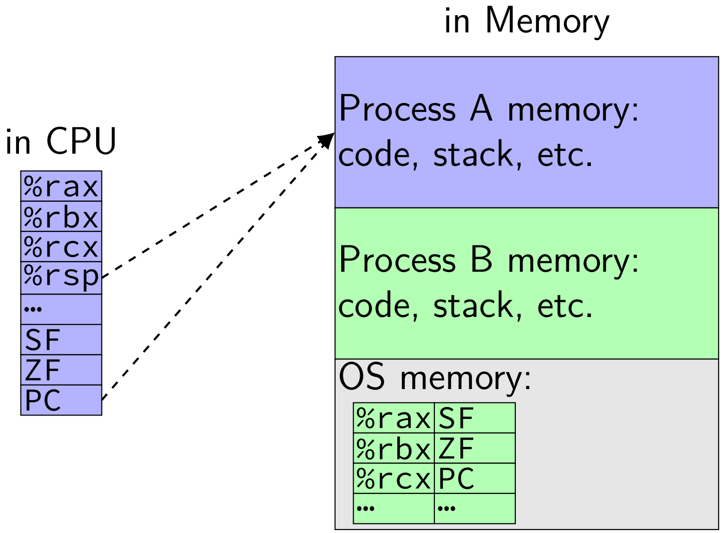
program counter

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

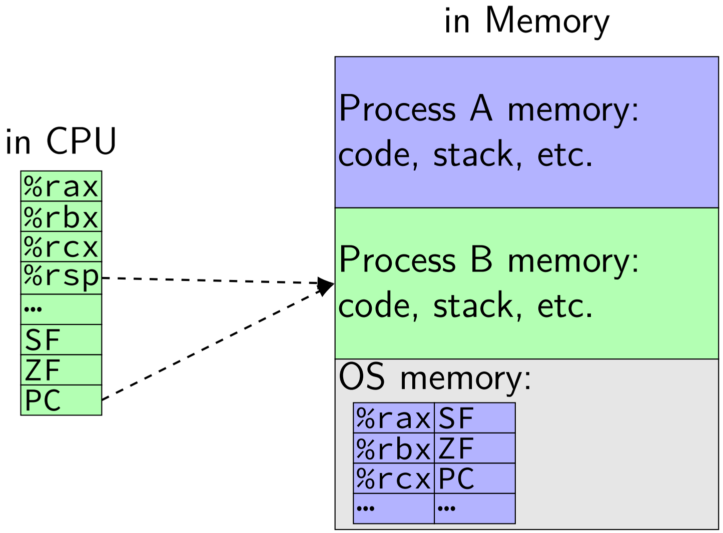
context switch pseudocode

```
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)



memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`      // ...`
`      // do work`
`      // ...`
`      movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

memory protection

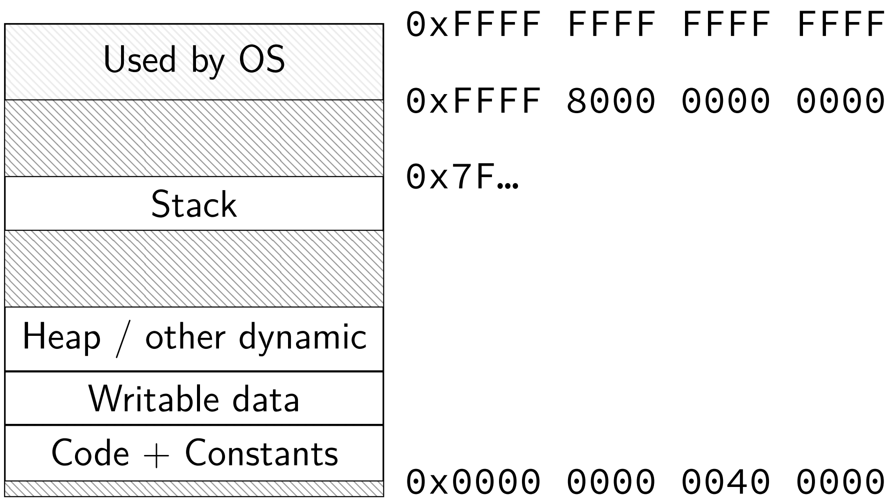
reading from another program's memory?

Program A	Program B

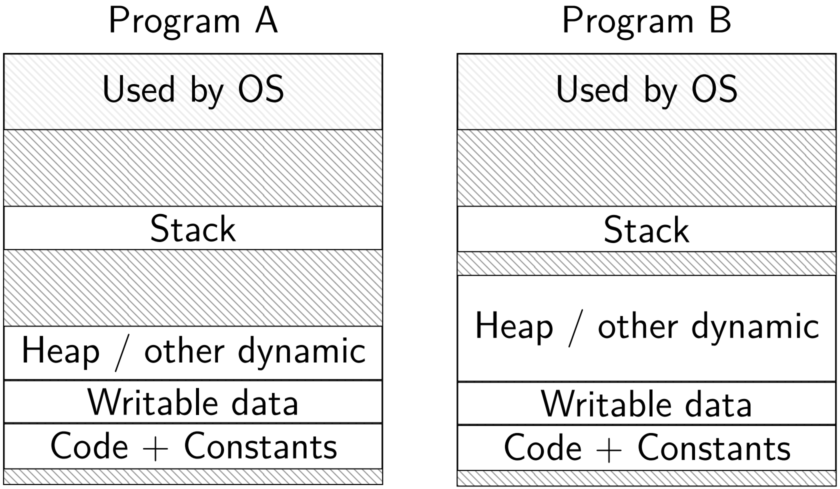
`0x10000: .word 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: `%rax` is 42 (always)	result: might crash

program memory



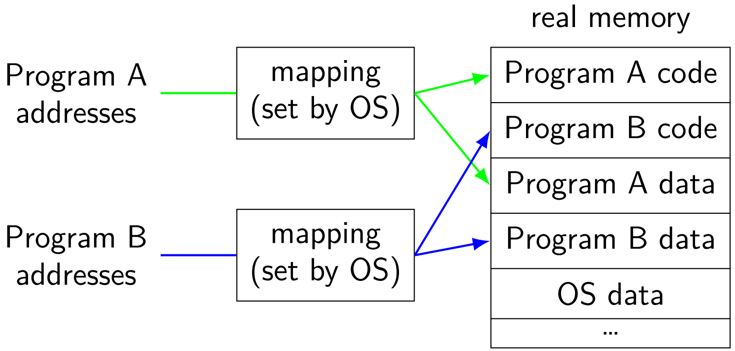
program memory (two programs)



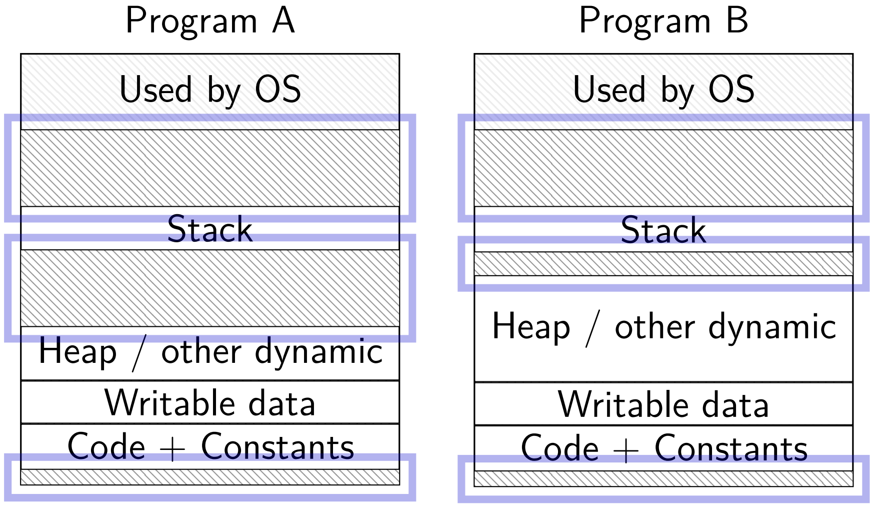
address space

programs have illusion of own memory

called a program's address space



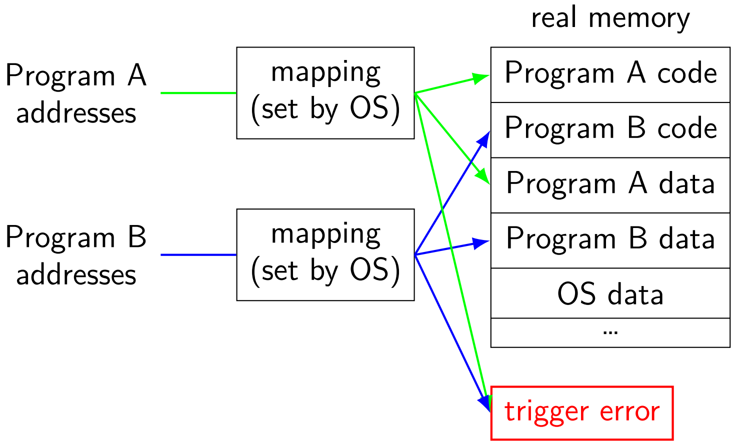
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

next topic

called virtual memory

mapping called page tables

mapping part of what is changed in context switch

context

all registers values

    `%rax` `%rbx`, …, `%rsp`, …

condition codes

program counter

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

address space: map from program to real addresses

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

types of exceptions

$$\left.\begin{array}{l}\text{interrupts — externally-triggered}\\ \quad\text{timer — keep program from hogging CPU}\\ \quad\text{I/O devices — key presses, hard drives, networks, ...}\\ \text{aborts — hardware is broken}\end{array}\right\}\begin{array}{l}\text{asynchronous}\\ \text{not triggered by}\\ \text{running program}\end{array}$$

$$\left.\begin{array}{l}\text{traps — intentionally triggered exceptions}\\ \quad\text{system calls — ask OS to do something}\\ \text{faults — errors/events in programs}\\ \quad\text{memory not in address space (“Segmentation fault”)}\\ \quad\text{privileged instruction}\\ \quad\text{divide by zero}\\ \quad\text{invalid instruction}\end{array}\right\}\begin{array}{l}\text{synchronous}\\ \text{triggered by}\\ \text{current program}\end{array}$$

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

} **asynchronous**
not triggered by running program

traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

} **synchronous**
triggered by current program

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

$\Big\}$ **asynchronous**
not triggered by running program

traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

$\Big\}$ **synchronous**
triggered by current program

timer interrupt

(conceptually) external timer device

(usually on same chip as processor)

OS configures before starting program

sends signal to CPU after a fixed interval

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

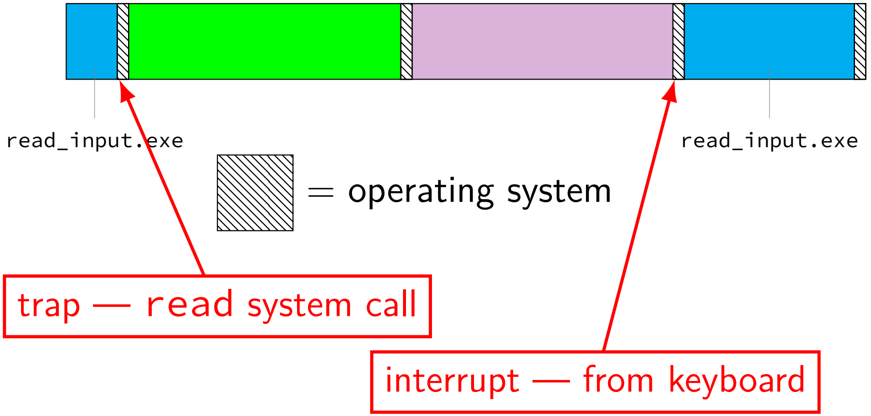
} asynchronous
not triggered by running program

traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

} synchronous
triggered by current program

keyboard input timeline



types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

} **asynchronous**
not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

} **synchronous**
triggered by current program

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

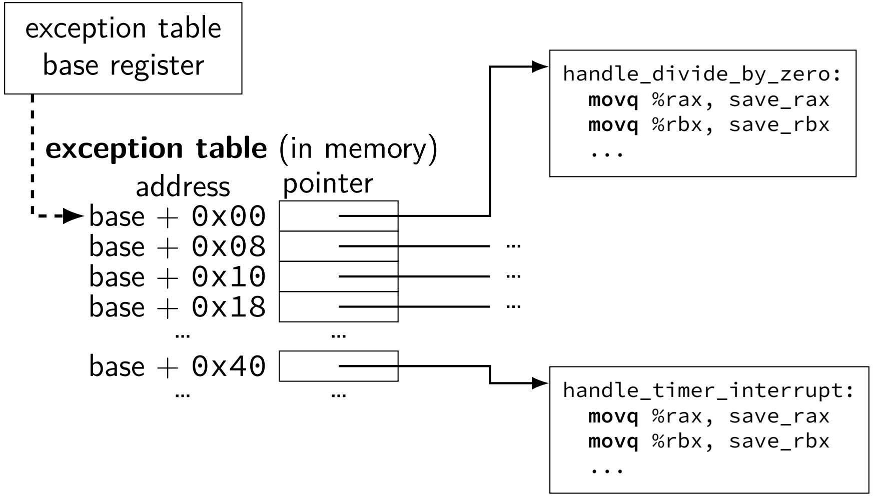
- jump done without program instruction to do so

exception implementation: notes

I/textbook describe a simplified version

real x86/x86-64 is a bit more complicated
(mostly for historical reasons)

locating exception handlers



```
handle_divide_by_zero:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

```
handle_timer_interrupt:
  movq %rax, save_rax
  movq %rbx, save_rbx
  ...
```

running the exception handler

hardware saves the old program counter (and maybe more)

identifies location of exception handler via table

then jumps to that location

OS code can save anything else it wants to , etc.

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC (and maybe more)
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC (and maybe more)
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

new logic: save the old PC (and maybe more)
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

added to CPU for exceptions

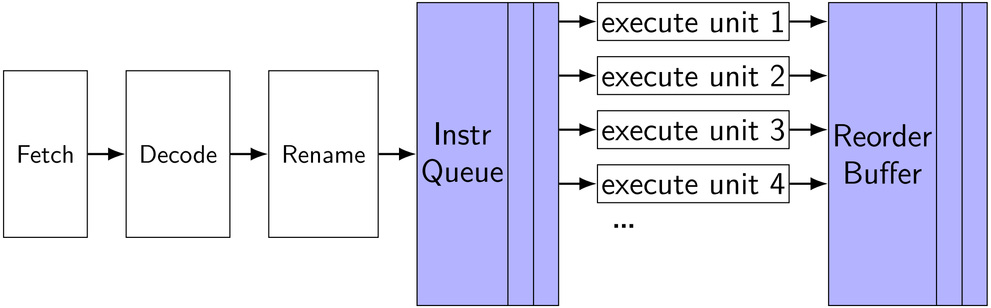
new instruction: set exception table base

new logic: jump based on exception table

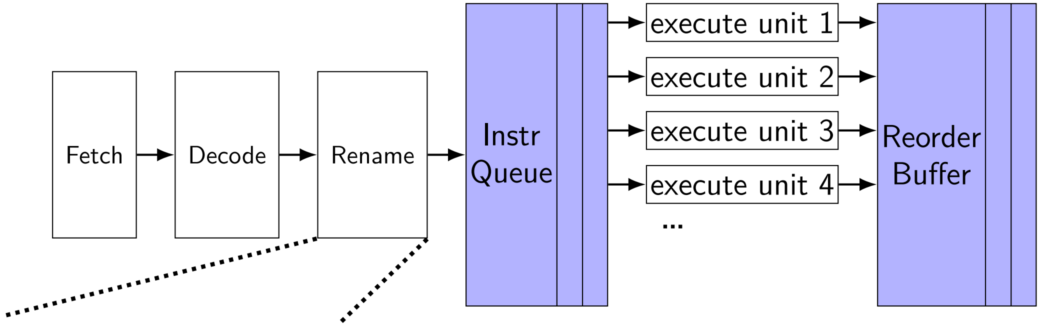
new logic: save the old PC (and maybe more)
- to special register or to memory

new instruction: return from exception
- i.e. jump to saved PC

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



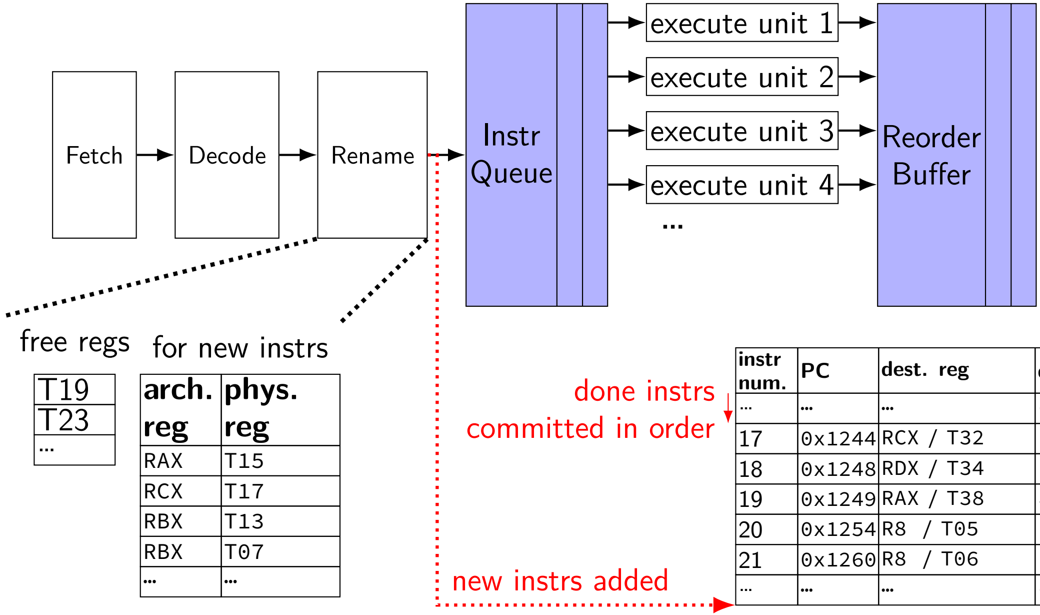
free regs

T19
T23
...

for new instrs

arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

exceptions and OOO (one strategy)

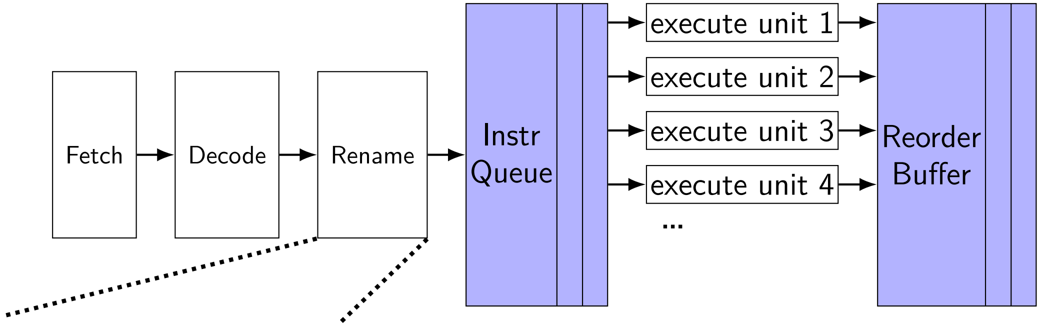


T19
T23
…

arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
17	0x1244	RCX / T32		
18	0x1248	RDX / T34		
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05		
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

T19
T23
...

for new instrs

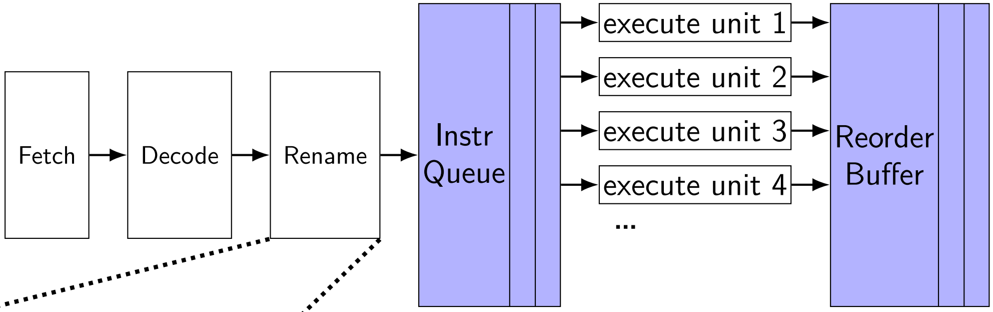
arch. reg	**phys. reg**
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

for complete instrs

arch. reg	**phys. reg**
RAX	T21
RCX	~~T2~~ T32
RBX	T48
RDX	T37
...	...

instr num.	**PC**	**dest. reg**	**done?**	**except?**
...	...	...	...	...
17	0x1244	RCX / T32	✓	
18	0x1248	RDX / T34		
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05		
21	0x1260	R8 / T06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

T19
T23
…

for new instrs

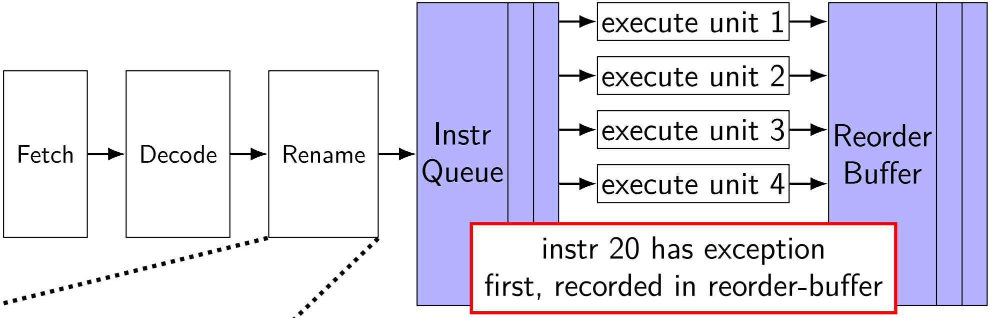
arch. reg	**phys. reg**
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

for complete instrs

arch. reg	**phys. reg**
RAX	T21
RCX	~~T2~~ T32
RBX	T48
RDX	T37
…	…

instr num.	**PC**	**dest. reg**	**done?**	**except?**
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / T32~~	~~✓~~	
18	0x1248	RDX / T34		
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05		
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



instr 20 has exception first, recorded in reorder-buffer

free regs

T19
T23
…

for new instrs

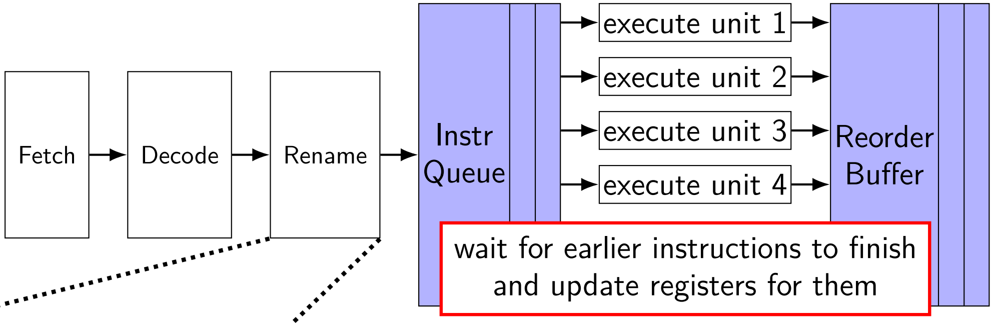
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

for complete instrs

arch. reg	phys. reg
RAX	T21
RCX	~~T2~~ T32
RBX	T48
RDX	T37
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / T32~~	~~✓~~	
18	0x1248	RDX / T34		
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

T19
T23
…

for new instrs

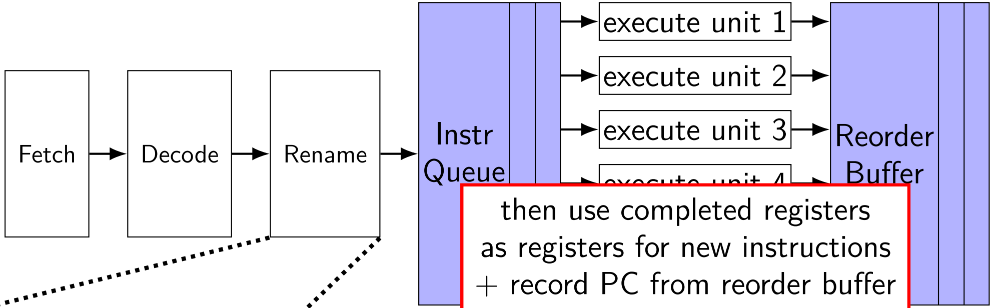
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

for complete instrs

arch. reg	phys. reg
RAX	~~T21~~ T38
RCX	~~T2~~ T32
RBX	T48
RDX	~~T37~~ T34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / T32~~	~~✓~~	
18	0x1248	RDX / T34	✓	
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



then use completed registers as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

T19
T23
...

for new instrs

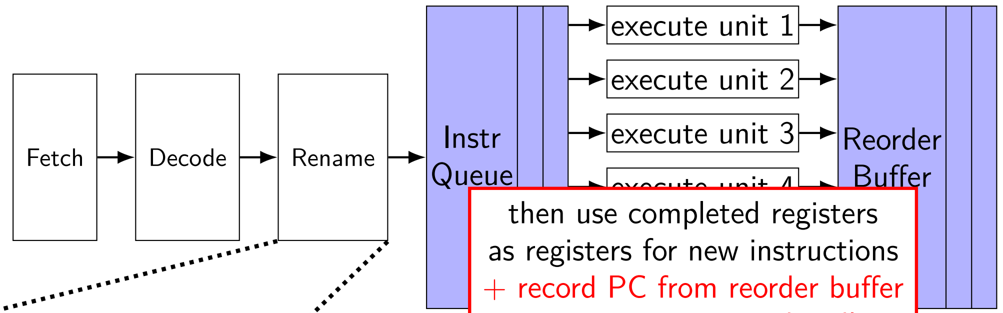
arch. reg	phys. reg
RAX	T38
RCX	T32
RBX	T48
RBX	T34
...	...

for complet

arch. reg	phys. reg
RAX	~~T21~~ T38
RCX	~~T2~~ T32
RBX	T48
RDX	~~T37~~ T34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / T32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / T34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / T38~~	~~✓~~	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
...	...	...	...	...

exceptions and OOO (one strategy)



then use completed registers
as registers for new instructions
+ record PC from reorder buffer
+ jump to exception handler

free regs

T19
T23
…

for new instrs

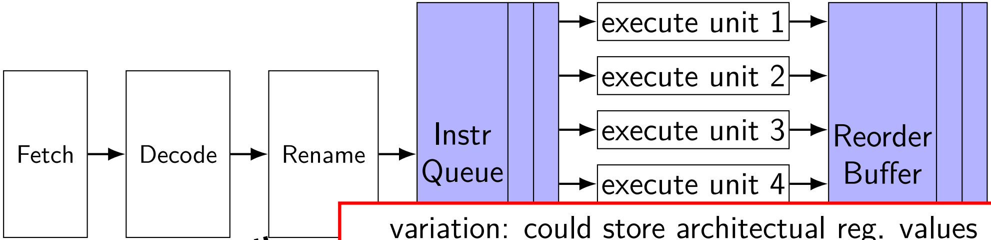
arch. reg	phys. reg
RAX	T38
RCX	T32
RBX	T48
RBX	T34
…	…

for complet

arch. reg	phys. reg
RAX	~~T21~~ T38
RCX	~~T2~~ T32
RBX	T48
RDX	~~T37~~ T34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / T32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / T34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / T38~~	~~✓~~	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



variation: could store architectual reg. values instead of mapping for completed instrs.
(and copy values instead of mapping on exception)

free regs

T19
T23
…

for new instrs

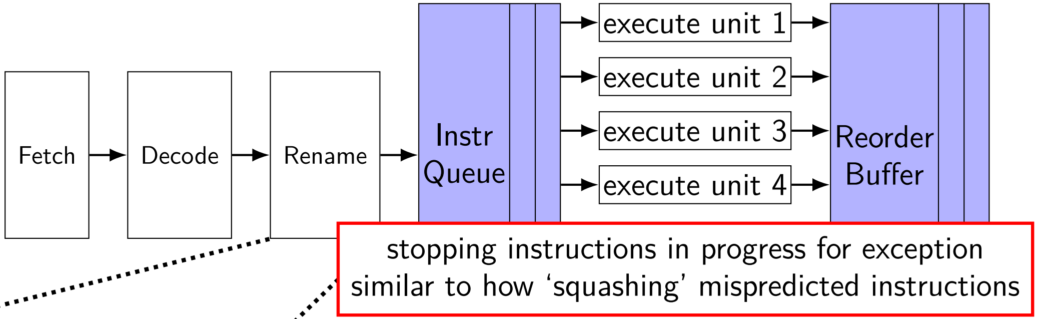
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

for complete instrs

arch. reg	value
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / T32~~	✓	
18	0x1248	RDX / T34	✓	
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
…	…	…	…	…

exceptions and OOO (one strategy)



stopping instructions in progress for exception
similar to how 'squashing' mispredicted instructions

free regs

T19
T23
...

for new instrs

arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~T21~~ T38
RCX	~~T2~~ T32
RBX	T48
RDX	~~T37~~ T34
...	...

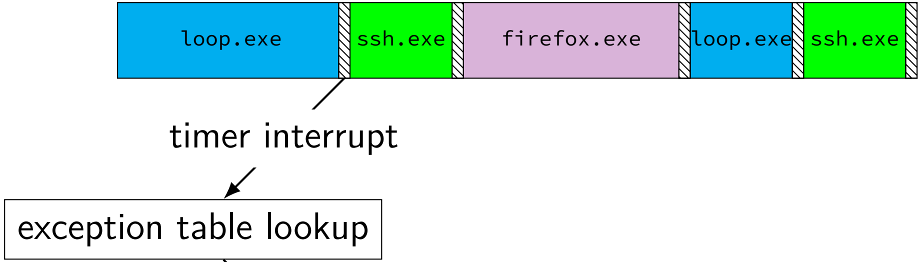
instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / T32~~	✓	
18	0x1248	RDX / T34	✓	
19	0x1249	RAX / T38	✓	
20	0x1254	R8 / T05	✓	✓
21	0x1260	R8 / T06		
...	...	...	...	...

exception handler structure

1. save process's state somewhere
2. do work to handle exception
3. restore a process's state (maybe a different one)
4. jump back to program

```
handle_timer_interrupt:
  mov_from_saved_pc save_pc_loc
  movq %rax, save_rax_loc
  ... // choose new process to run here
  movq new_rax_loc, %rax
  mov_to_saved_pc new_pc
  return_from_exception
```

exceptions and time slicing



```
handle_timer_interrupt:
    ...
    ...
    set_address_space ssh_address_space
    mov_to_saved_pc saved_ssh_pc
    return_from_exception
```

defeating time slices?

```
my_exception_table:
    ...
my_handle_timer_interrupt:
    // HA! Keep running me!
    return_from_exception

main:
    set_exception_table_base my_exception_table
loop:
    jmp loop
```

defeating time slices?

wrote a program that tries to set the exception table:

```
my_exception_table:
    ...

main:
    // "Load Interrupt
    //  Descriptor Table"
    // x86 instruction to set exception table
    lidt my_exception_table
    ret
```

result: `Segmentation fault` (exception!)

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

$\Bigg\}$ asynchronous
not triggered by running program

traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space (“Segmentation fault”)
- privileged instruction
- divide by zero
- invalid instruction

$\Bigg\}$ synchronous
triggered by current program

privileged instructions

can't let any program run some instructions

allows machines to be shared between users (e.g. lab servers)

examples:

- set exception table
- set address space
- talk to I/O device (hard drive, keyboard, display, …)
- …

processor has two modes:

- kernel mode — privileged instructions work
- user mode — privileged instructions cause exception instead

kernel mode

extra one-bit register: “are we in kernel mode”

exceptions enter kernel mode

return from exception instruction leaves kernel mode

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

} asynchronous
not triggered by running program

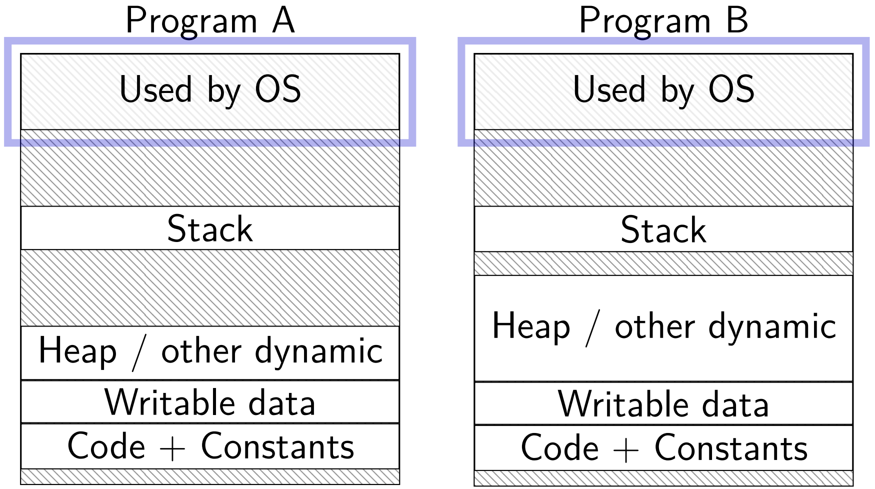
traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

} synchronous
triggered by current program

what about editing interrupt table?

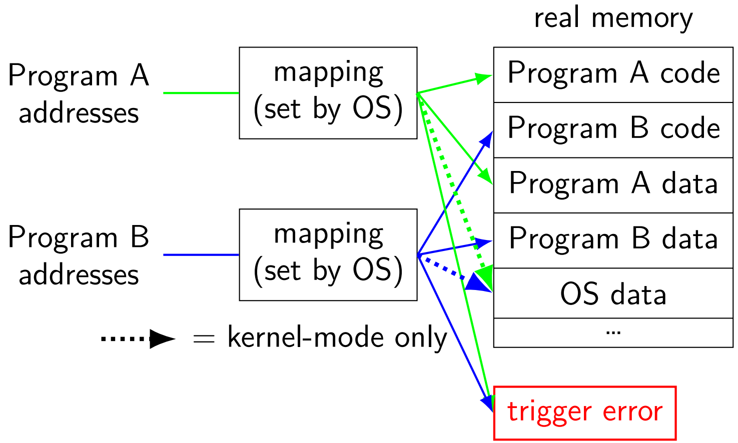
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



protection fault

when program tries to access memory it doesn't own

e.g. trying to write to bad address

when program tries to do other things that are not allowed

e.g. accessing I/O devices directly

e.g. changing exception table base register

OS gets control — can crash the program

- or more interesting things

types of exceptions

interrupts — externally-triggered
- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

$\Big\}$ asynchronous
not triggered by running program

traps — intentionally triggered exceptions
- system calls — ask OS to do something

faults — errors/events in programs
- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

$\Big\}$ synchronous
triggered by current program

kernel services

allocating memory? (change address space)

reading/writing to file? (communicate with hard drive)

read input? (communicate with keyborad)

all need privileged instructions!

need to run code in kernel mode

Linux x86-64 system calls

special instruction: `syscall`

triggers trap (deliberate exception)

Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times “error number”

almost the same as normal function calls

Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello,␣World!\n"
.text
_start:
  movq $1, %rax # 1 = "write"
  movq $1, %rdi # file descriptor 1 = stdout
  movq $hello_str, %rsi
  movq $15, %rdx # 15 = strlen("Hello, World!\n")
  syscall

  movq $60, %rax # 60 = exit
  movq $0, %rdi
  syscall
```

approx. system call handler

```
sys_call_table:
    .quad handle_read_syscall
    .quad handle_write_syscall
    // ...

handle_syscall:
    ... // save old PC, etc.
    pushq %rcx // save registers
    pushq %rdi
    ...
    call *sys_call_table(,%rax,8)
    ...
    popq %rdi
    popq %rcx
    return_from_exception
```

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files

terminals, etc. count as files, too

system call wrappers

can't write C code to generate syscall instruction

solution: call "wrapper" function written in assembly

a note on terminology (1)

real world: inconsistent terms for exceptions

we will follow textbook's terms in this course

the real world won't

you might see:

- 'interrupt' meaning what we call 'exception' (x86)
- 'exception' meaning what we call 'fault'
- 'hard fault' meaning what we call 'abort'
- 'trap' meaning what we call 'fault'
- ... and more

a note on terminology (2)

we use the term “kernel mode”

some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

- different sets of priviliged operations work

backup slides

vector instructions

modern processors have registers that hold "vector" of values

example: current x86-64 processors have 256-bit registers

- 8 ints or 8 floats or 4 doubles or …

256-bit registers named `%ymm0` through `%ymm15`

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

(also 128-bit versions named `%xmm0` through `%xmm15`)

example vector instruction

`vpaddd %ymm0, %ymm1, %ymm2` (packed add dword (32-bit))

Suppose registers contain (interpreted as 4 ints)

%ymm0: [1, 2, 3, 4, 5, 6, 7, 8]
%ymm1: [9, 10, 11, 12, 13, 14, 15, 16]

Result will be:

%ymm2: [10, 12, 14, 16, 18, 20, 22, 24]

vector instructions

```
void add(int * restrict a, int * restrict b) {
    for (int i = 0; i < 512; ++i)
        a[i] += b[i];
}
```

```
add:
  xorl %eax, %eax
the_loop:
  vmovdqu (%rdi,%rax), %ymm0  /* load A into ymm0 */
  vmovdqu (%rsi,%rax), %ymm1  /* load B into ymm1 */
  vpaddd %ymm1, %ymm0, %ymm0  /* ymm1 + ymm0 -> ymm0 */
  vmovdqu %ymm0, (%rdi,%rax)  /* store ymm0 into A */
  addq $32, %rax              /* increment index by 32 bytes */
  cmpq $2048, %rax
  jne the_loop
  vzeroupper        /* ←- for calling convention reasons */
  ret
```

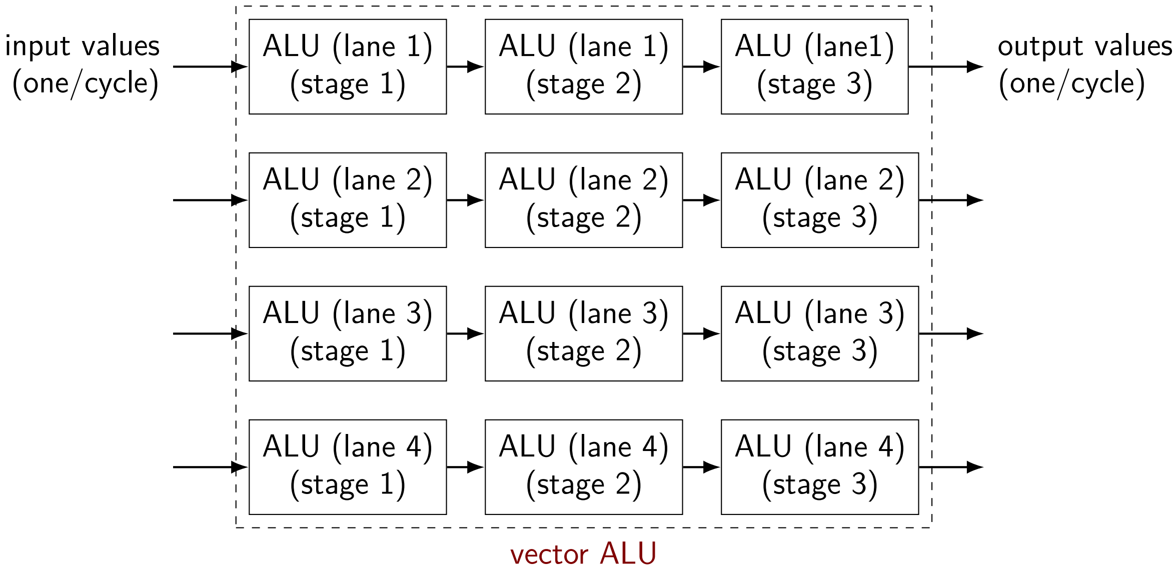
alternate vector interfaces

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"

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 8.2 and Clang 7.0 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: (fixed in later versions)

```
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];
}
```

system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $−1, %rax
    ret
```

system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $−1, %rax
    ret
```

system call wrapper: usage

```
/* unistd.h contains definitions of:
     O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
    int file_descriptor;
    file_descriptor = open("input.txt", O_RDONLY);
    if (file_descriptor < 0) {
        printf("error:␣%s\n", strerror(errno));
        exit(1);
    }
    ...
    result = read(file_descriptor, ...);
    ...
}
```

system call wrapper: usage

```
/* unistd.h contains definitions of:
    O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
  int file_descriptor;
  file_descriptor = open("input.txt", O_RDONLY);
  if (file_descriptor < 0) {
      printf("error:␣%s\n", strerror(errno));
      exit(1);
  }
  ...
  result = read(file_descriptor, ...);
  ...
}
```

protection and sudo

programs always run in user mode

extra permissions from OS do not change this

- sudo, superuser, root, SYSTEM, …

operating system may remember extra privileges

careful exception handlers

```
movq $important_os_address, %rsp
```

can't trust user's stack pointer!

need to have own stack in kernel-mode-only memory

need to check all inputs really carefully

256-bit with 128-bit?

Intel designed 256-bit vector instructions with 128-bit ones in mind

goal: possible to use 128-bit vector ALUs to implement 256-bit instructions

- split 256-bit instruction into two ALU operations

means less instructions move values from top to bottom half of vector

- in particular, complicated to move 16-bit value between halfs

aside on AVX and clock speeds

some processors ran slower when 256-bit ALUs are being used

- includes a lot of notable Intel CPUs

why? they give out heat — can't maintain higher clock speed

- for energy reasons, shut down when not used

still faster *assuming you're using vectors a lot*