optimization 3 / SIMD

Changelog

29 October 2020: aliasing exercise: change options, worked-through example slide

last time

instruction dispatch

data flow model

- latency bound: look for sequence of instructions one-after-another
- reassociation (order of operations change) + multiple accumulators

compiler optimizer limitations:

- can't guess speed v size tradeoffs
- mostly method/file at a time

aliasing: two variables might be the same

- usually with pointers — might go to same array
- prevents compiler from keeping things in registers
- maye prevent other optimizations, too
- `for (i = 0; i < N; ++i) result[j] += M[j*N + i]`
  - what if result and matrix point to parts of same array?
  - then result[j] and matrix[j*N + 1] could be same element
  - would affect what's added to result[j] when i = 1!
  - compiler needs to handle!

compiler limitations

needs to generate code that does the same thing…

- …even in corner cases that “obviously don’t matter”

often doesn’t ‘look into’ a method

- needs to assume it might do anything

can’t predict what inputs/values will be

- e.g. lots of loop iterations or few?

can’t understand code size versus speed tradeoffs

aliasing exercise

```
void add(int *s1, int *s2, int *d) {
    for (int i = 0; i < 1000; ++i)
        d[i] = s1[i] + s2[i];
}
```

The compiler **cannot** generate code equivalent to this:

```
void add(int *s1, int *s2, int *d) {
    for (int i = 0; i < 1000; i += 2) {
        int temp1 = s1[i] + s2[i];
        int temp2 = s1[i+1] + s2[i+1];
        d[i] = temp1; d[i+1] = temp2;
    }
}
```

Which is an example of a call where the results could disagree:

A. `add(&A[0], &A[1], &B[0])` B. `add(&A[0], &A[0], &A[1])`

C. `add(&B[0], &A[10], &A[0])` D. `add(&A[1000], &A[1001], &A[0])`

(assume A, B are distinct, large arrays)

aliasing exercise

recall: s1=s2=A+0; d=A+1

```
for (int i = 0; i < 1000; ++i)
    d[i] = s1[i] + s2[i];

/* i = 0: */ A[1] = A[0] + A[0];
/* i = 1: */ A[2] = A[1] + A[1];
```

```
for (int i = 0; i < 1000; i += 2) {
    temp1 = s1[i] + s2[i];
    temp2 = s1[i] + s2[i];
    d[i] = temp1;
    d[i] = temp2;

/* i = 0: */ temp1 = A[0] + A[0];
             temp2 = A[1] + A[1];
             A[1] = temp1;
             A[2] = temp2;
```

aliasing and cache optimizations

```
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];
```

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

`C = A`? `C = &A[10]`?

compiler can't generate same code for both

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

loop with a function call

```
int addWithLimit(int x, int y) {
    int total = x + y;
    if (total > 10000)
        return 10000;
    else
        return total;
}
...
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = addWithLimit(sum, array[i]);
    return sum;
}
```

function call assembly

```
    movl (%rbx), %esi // mov array[i]
    movl %eax, %edi   // mov sum
    call addWithLimit
    ...
    ...
addWithLimit:
    ... /* code here */
    ret
```

extra instructions executed: two moves, a call, and a ret

function call assembly

```
    movl (%rbx), %esi // mov array[i]
    movl %eax, %edi   // mov sum
    call addWithLimit
    ...
    ...
addWithLimit:
    ... /* code here */
    ret
```

extra instructions executed: two moves, a call, and a ret

alternative: *inline* the call

```
    ... /* code here (+ small changes for arguments
                        being in different places) */
    ret
```

manual inlining

```
int sum(int *array, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum = sum + array[i];
        if (sum > 10000)
            sum = 10000;
    }
    return sum;
}
```

inlining pro/con

avoids call, ret, extra move instructions

allows compiler to use more registers

- no caller-saved register problems

but not always faster:

worse for instruction cache

- (more copies of function body code)

compiler inlining

compilers will inline, but...

will usually avoid making code much bigger

- heuristic: inline if function is small enough
- heuristic: inline if called exactly once

will usually not inline across .o files

some compilers allow hints to say "please inline/do not inline this function"

compiler limitations

needs to generate code that does the same thing…

…even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

remove redundant operations (1)

```
int number_of_As(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (1, fix)

```
int number_of_As(const char *str) {
    int count = 0;
    int length = strlen(str);
    for (int i = 0; i < length; ++i) {
        if (str[i] == 'a')
            count++;
    }
    return count;
}
```

call strlen once, not once per character!

Big-Oh improvement!

remove redundant operations (2)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    for (int i = 0; i < N; ++i) {
        if (i + amount < N)
            dest[i] = source[i + amount];
        else
            dest[i] = source[N - 1];
    }
}
```

compare i + amount to N many times

remove redundant operations (2, fix)

```
int shiftArray(int *source, int *dest, int N, int amount) {
    int i;
    for (i = 0; i + amount < N; ++i) {
        dest[i] = source[i + amount];
    }
    for (; i < N; ++i) {
        dest[i] = source[N - 1];
    }
}
```

eliminate comparisons

compiler limitations

needs to generate code that does the same thing…

…even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

needs to assume it might do anything

can't predict what inputs/values will be

e.g. lots of loop iterations or few?

can't understand code size versus speed tradeoffs

exercise: when optimizations backfire...

Which of these optimizations are likely to **increase** machine code size? (**Select all that apply.**)

Which of these optimizations are likely to **increase** number of instructions executed? (**Select all that apply.**)

A. cache blocking B. function inlining C. loop unrolling
D. moving a calculation outside a loop
E. multiple accumulators (after loop unrolling)

looplab speeds on my desktop

original assembly: 2.0 cycles/element

unrolled x2: 1.0 cycles element

unrolled x4: 1.0 cycles element

unrolled x8: 1.0 cycles element

unrolled x8, 4 accumulators: 0.5 cycles element

looplab speeds on my desktop

original assembly: 2.0 cycles/element

unrolled x2: 1.0 cycles element

unrolled x4: 1.0 cycles element

unrolled x8: 1.0 cycles element

unrolled x8, 4 accumulators: 0.5 cycles element

Clang 6 optimized code: 0.13 cycles/element

GCC optimized code: 0.14 cycles/element

looplab speeds on my desktop

original assembly: 2.0 cycles/element

unrolled x2: 1.0 cycles element

unrolled x4: 1.0 cycles element

unrolled x8: 1.0 cycles element

unrolled x8, 4 accumulators: 0.5 cycles element

Clang 6 optimized code: 0.13 cycles/element

GCC optimized code: 0.14 cycles/element

how? instructions that add *16 pairs of shorts* at once!

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)

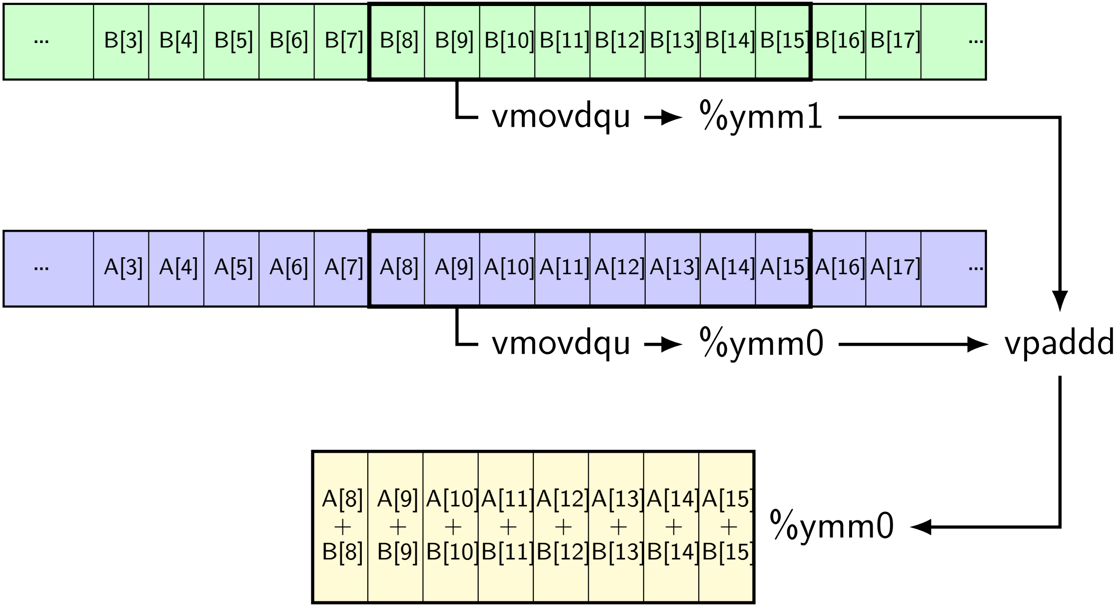
```
unsigned int A[512], B[512];
...
for (int i = 0; i < 512; i += 8) {
    A[i+0] = A[i+0] + B[i+0];
    A[i+1] = A[i+1] + B[i+1];
    A[i+2] = A[i+2] + B[i+2];
    A[i+3] = A[i+3] + B[i+3];
    A[i+4] = A[i+4] + B[i+4];
    A[i+5] = A[i+5] + B[i+5];
    A[i+6] = A[i+6] + B[i+6];
    A[i+7] = A[i+7]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

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

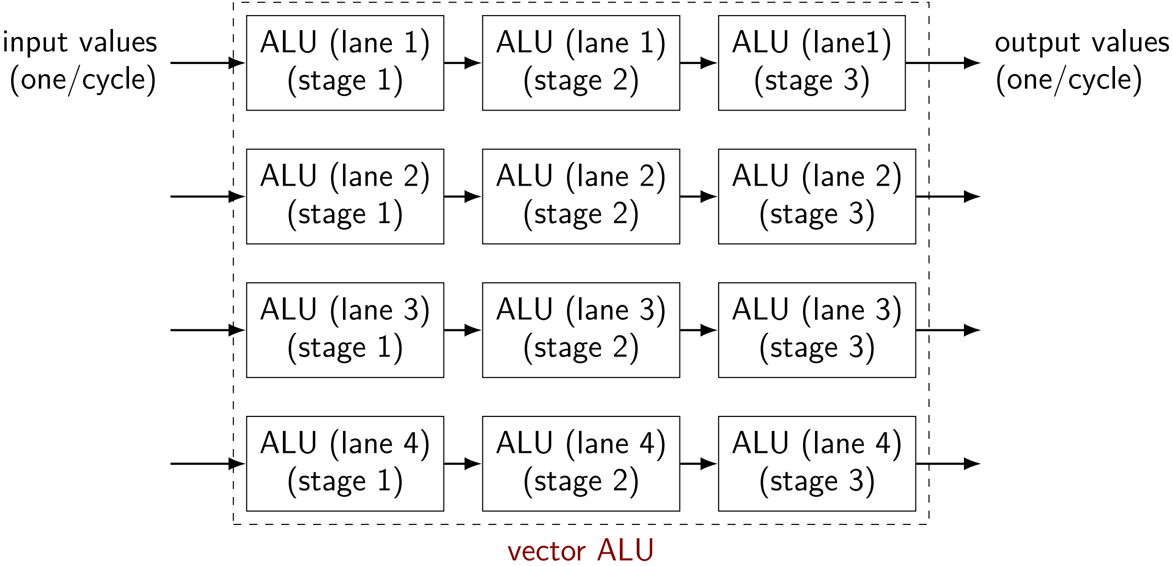
desired assembly

```
    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



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];
}
```

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

```
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
f
  // "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; i < 512; i += 8) {
  // "si256" -->
  // a_values =                                     (8 x 32 bits)
  __m256i a_valu                                   56i*) &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"

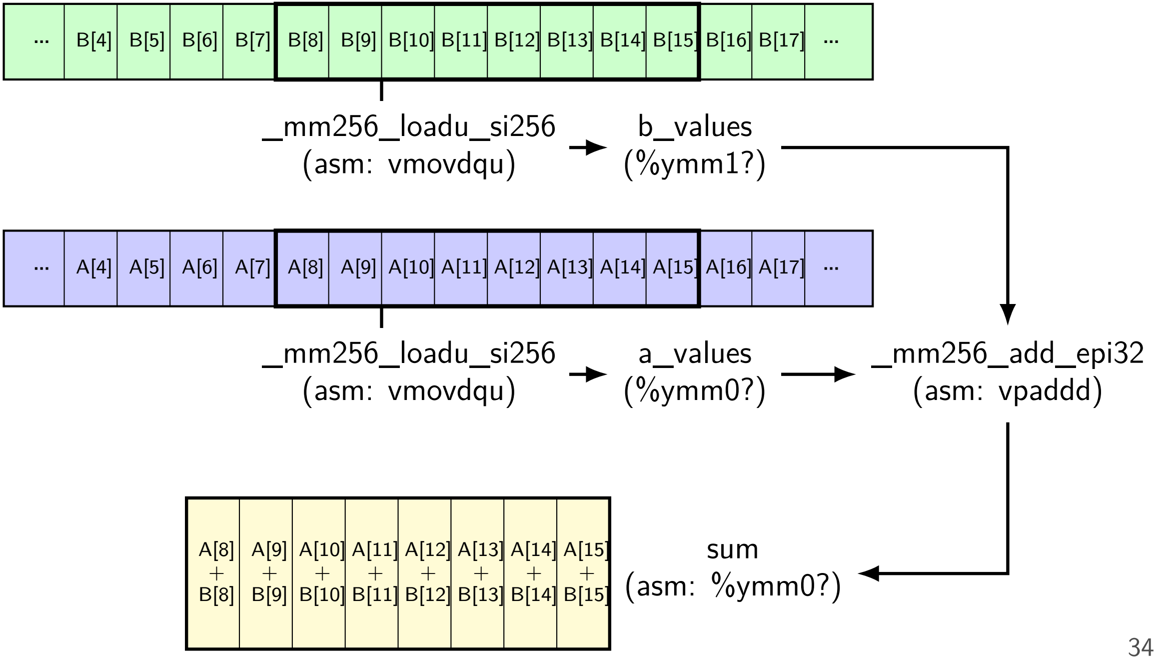
vector intrinsics: different size

```
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);
}
```

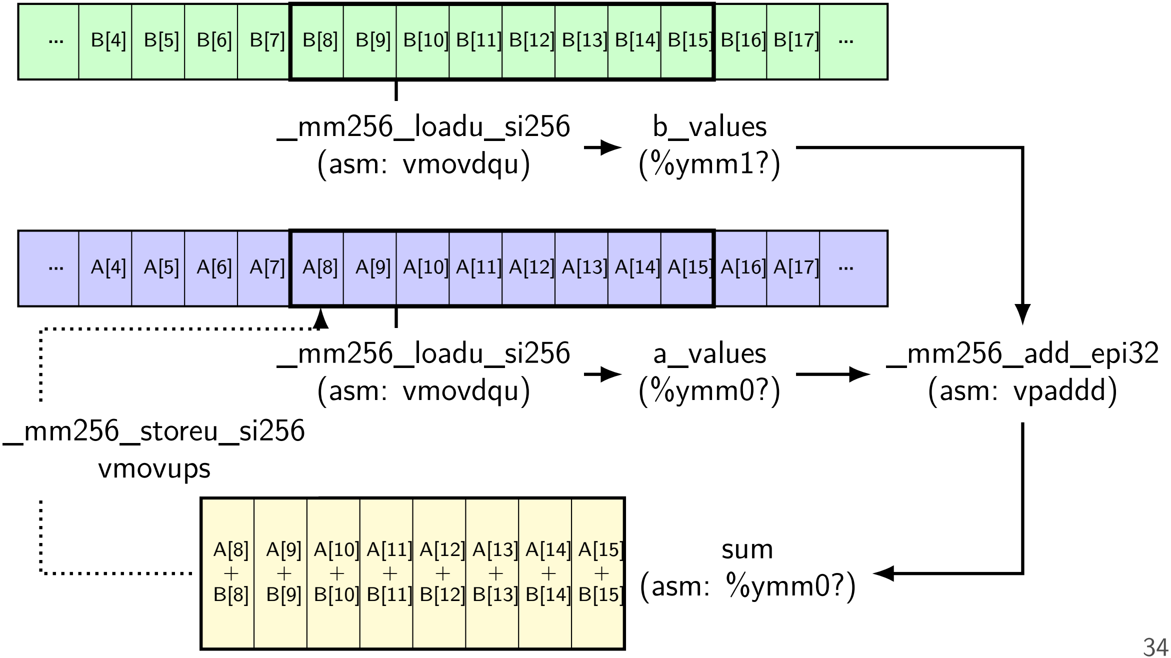
vector intrinsics: different size

```
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)



exercise

```
long foo[8] = {1,1,2,2,3,3,4,4};
long bar[8] = {2,2,2,3,3,3,4,4};
__mm256i foo0_as_vector = _mm256_loadu_si256((__m256i*)&foo[0])
__mm256i foo4_as_vector = _mm256_loadu_si256((__m256i*)&foo[4])
__mm256i bar0_as_vector = _mm256_loadu_si256((__m256i*)&bar[0])

__mm256i result = _mm256_add_epi64(foo0_as_vector, foo4_as_vector);
result = _mm256_mullo_epi64(result, bar0_as_vector);
_mm256_storeu_si256((__mm256i*) &bar[4], result);
```

Final value of bar array?

```
A. {2,2,2,3,12,12,24,24}  B. {2,2,2,3,15,15,28,28}
C. {2,2,2,3,10,10,20,20}  D. {12,12,24,24,3,3,4,4}
E. {14,14,26,27,3,3,4,4}  F. {14,14,26,27,12,12,24,24}
G. something else
```

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

// Cij= {C_{i,j} + A_{i,k} × B_{k,j}, C_{i,j+1} + A_{i,k} × 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

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"

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

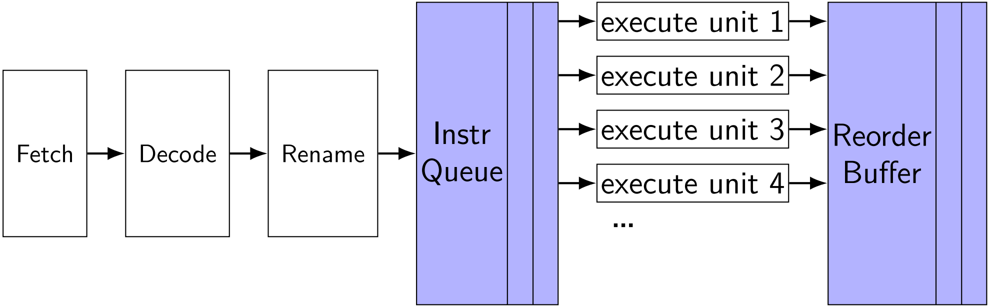
first step — tool to determine where you spend time

tools exist to do this for programs

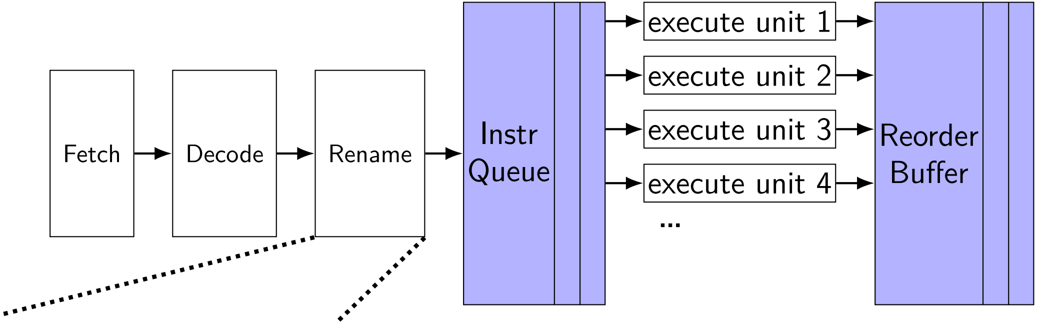
example on Linux: `perf`

backup slides

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



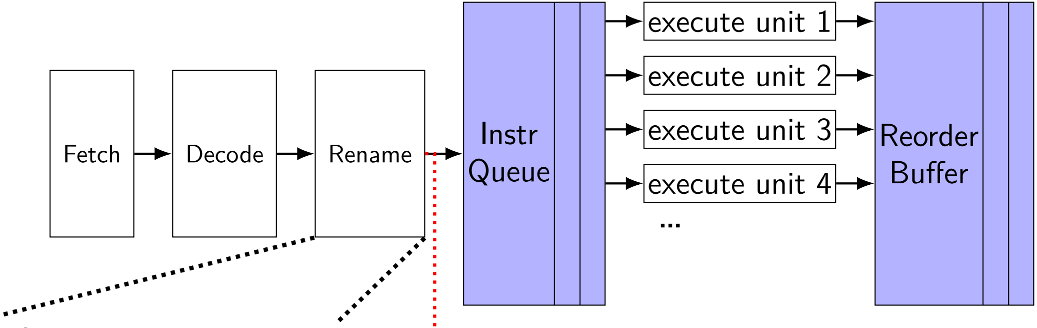
free regs

T19
T23
...

for new instrs

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

exceptions and OOO (one strategy)



free regs

T19
T23
...

for new instrs

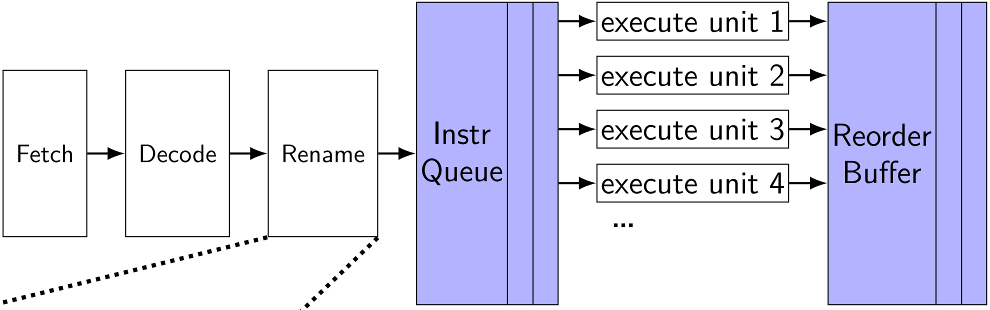
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

done instrs committed in order

new instrs added

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

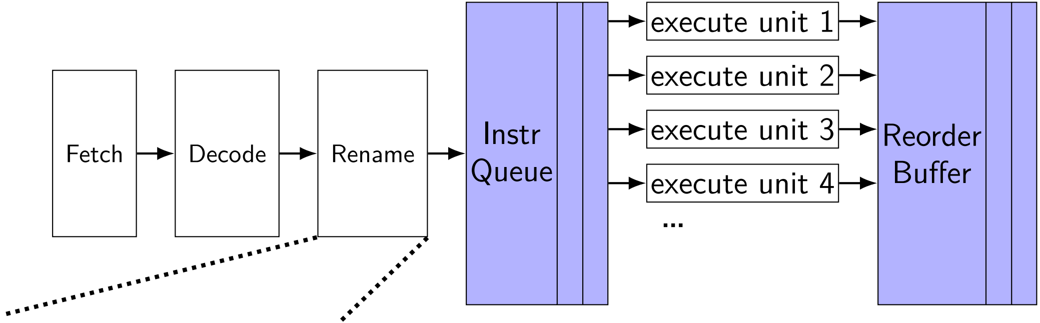
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

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

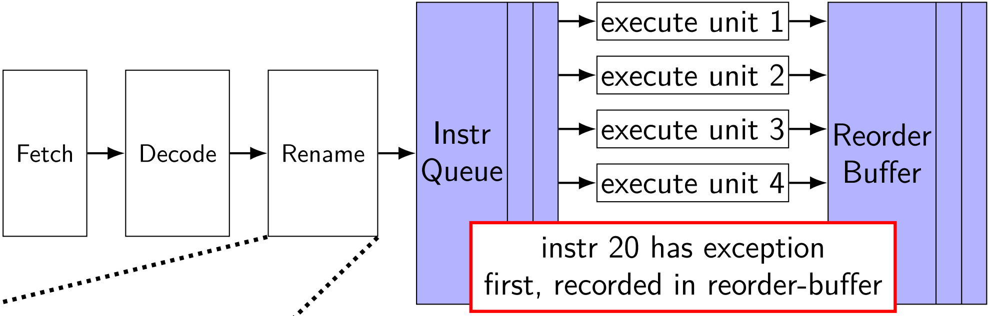
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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)



free regs

T19
T23
…

for new instrs

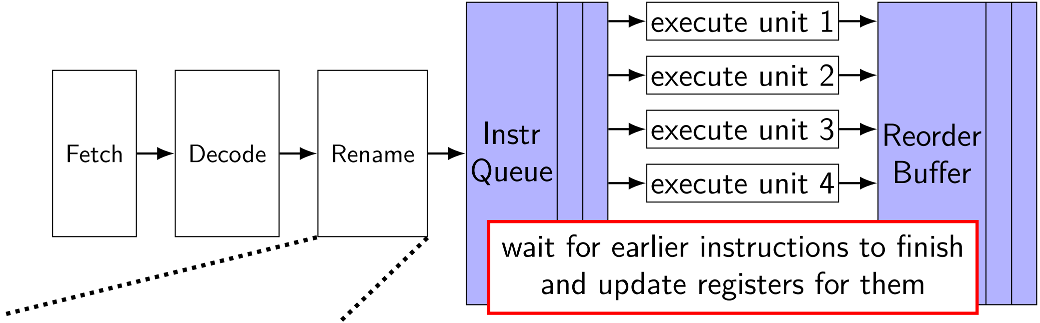
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

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

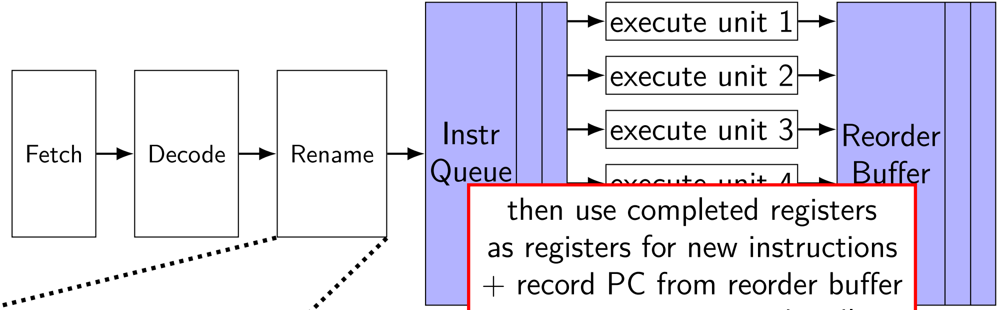
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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

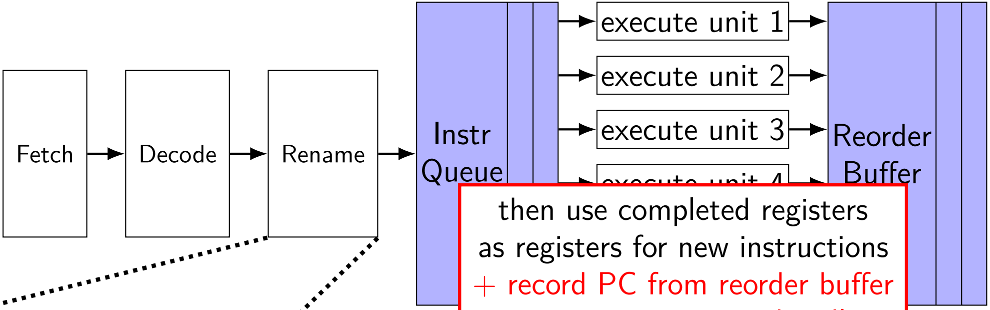
arch. reg	phys. reg
RAX	T38
RCX	T32
RBX	T48
RBX	T34
…	…

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

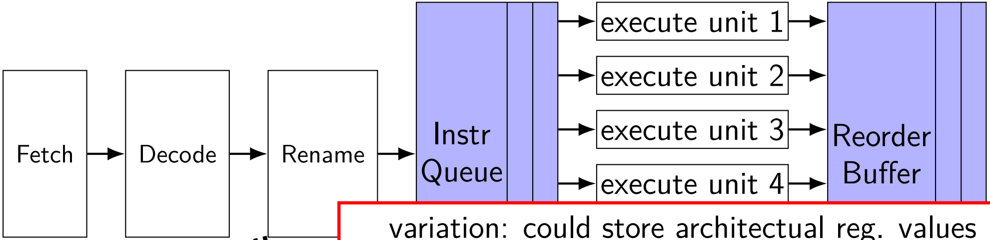
arch. reg	phys. reg
RAX	T38
RCX	T32
RBX	T48
RBX	T34
…	…

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

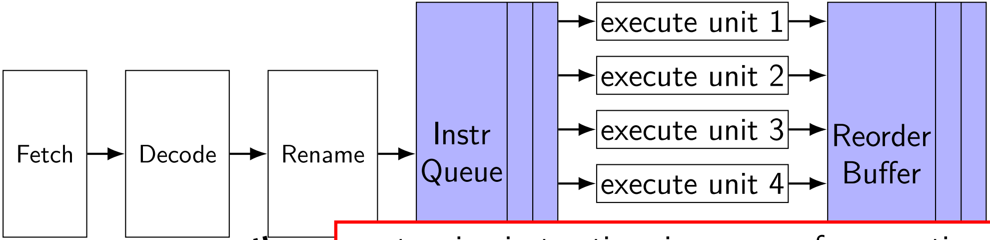
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
…	…

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		
…	…	…	…	…

addressing efficiency

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        Cij += A[i * N + k] * B[k * N + j];
      }
      C[i * N + j] = Cij;
    }
  }
}
```

tons of multiplies by N??

isn't that slow?

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will often do this

increment/decrement by N ($\times$ sizeof(float))

addressing efficiency

compiler will usually eliminate slow multiplies

doing transformation yourself often slower if so

`i * N; ++i` into `i_times_N; i_times_N += N`

way to check: see if assembly uses lots multiplies in loop

if it doesn't — do it yourself

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
int offset = 0;
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
    offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

another addressing transformation

```
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
        C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
        // ...
```

```
    int offset = 0;
    float *Ai0_base = &A[0*n+k];
    float *Ai1_base = Ai0_base + n;
    float *Ai2_base = Ai1_base + n;
    // ...
    for (int i = 0; i < n; i += 20) {
        C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
        C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
        // ...
        offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

---

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

alternative addressing transformation

instead of:

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

---

could do:

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

addressing efficiency generally

mostly: compiler does very good job itself

- eliminates multiplications, use pointer arithmetic
- often will do better job than if how typically programming would do it manually

sometimes compiler won't take the best option

- if spilling to the stack: can cause weird performance anomalies
- if indexing gets too complicated — might not remove multiply

if compiler doesn't, you can always make addressing simple yourself

- convert to pointer arith. without multiplies

recall: shifts

we mentioned that compilers compile $x/4$ into a shift instruction

they are really good at these types of of transformation...

> "strength reduction": replacing complicated op with simpler one

but can't do without seeing special case (e.g. divide by constant)

loop unrolling downsides

bigger executables $\rightarrow$ instruction cache misses

slower if small number of loop iterations

extra code to handle leftovers, etc.

want to unroll loops that are run a lot and quick to execute

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```

figuring out how to unroll?

exercise: why can the compiler probably not do this transformation?

```
void foo() { int sum = 0;
  for (int i = 0; i < some_global_variable; ++i) {
    sum += some_function();
  }
}
```

```
void foo_transformed() { int sum = 0;
  int i = 0;
  if (some_global_variable % 2 == 1) {
    i += 1;
    sum += some_function();
  }
  for (; i < some_global_variable; i += 2) {
    sum += some_function();
    sum += some_function();
  }
}
```

multiple accumulators downsides

downsides of loop unrolling

  bigger executables, slower if small number of iterations

+ uses extra registers (can't use those regs for something else)

want to use multiple accumulators if latency likely bottleneck

problem: compiler probably can't tell if this meets those criteria

```
for (int i = 0; i < some_variable; ++i) {
  sum += some_function();
}
```