last time

a transformation

```
for (int kk = 0; kk < N; kk += 2)
  for (int k = kk; k < kk + 2; ++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];
```

split the loop over k — should be exactly the same

(assuming even N)

a transformation

```
for (int kk = 0; kk < N; kk += 2)
  for (int k = kk; k < kk + 2; ++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];
```

split the loop over k — should be exactly the same

(assuming even N)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now reorder split loop — same calculations

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking

```
for (int kk = 0; kk < N; kk += 2)
  /* was here: for (int k = kk; k < kk + 2; ++k) */
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        /* load Aik, Aik+1 into cache and process: */
        for (int k = kk; k < kk + 2; ++k)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
```

now reorder split loop — same calculations

now handle B_{ij} for $k+1$ right after B_{ij} for k

(previously: $B_{i,j+1}$ for k right after B_{ij} for k)

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
  }
}
```

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
  }
}
```

Temporal locality in C_{ij}s

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
  }
}
```

More spatial locality in A_{ik}

simple blocking – expanded

```
for (int kk = 0; kk < N; kk += 2) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block" of 2 k values: */
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
  }
}
```

Still have good spatial locality in B_{kj}, C_{ij}

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)

...

...

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)

...

A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] ...

A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] ...

...

counting misses for A (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for A:

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)

A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)

...

A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] ...

A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] ...

...

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)
...

...

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)
...
A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] ...
A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] ...
...

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

A[0*N+0] and A[0*N+1] usually in same cache block

counting misses for A (2)

A[0*N+0], A[0*N+1], A[0*N+0], A[0*N+1] ...(repeats N times)
A[1*N+0], A[0*N+1], A[0*N+0], A[1*N+1] ...(repeats N times)
...
A[(N-1)*N+0], A[(N-1)*N+1], A[(N-1)*N+0], A[(N-1)*N+1] ...
A[0*N+2], A[0*N+3], A[0*N+2], A[0*N+3] ...
...

likely cache misses: only first iterations of j loop

how many cache misses per iteration? usually one

A[0*N+0] and A[0*N+1] usually in same cache block

about $\frac{N}{2} \cdot N$ misses total

counting misses for B (1)

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

access pattern for B:

B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...

counting misses for B (2)

access pattern for B:
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...

counting misses for B (2)

access pattern for B:
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...

likely cache misses: any access, each time

counting misses for B (2)

access pattern for B:
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...
likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

counting misses for B (2)

access pattern for B:
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
B[2*N+0], B[3*N+0], ...B[2*N+(N-1)], B[3*N+(N-1)]
B[4*N+0], B[5*N+0], ...B[4*N+(N-1)], B[5*N+(N-1)]
...
B[0*N+0], B[1*N+0], ...B[0*N+(N-1)], B[1*N+(N-1)]
...
likely cache misses: any access, each time

how many cache misses per iteration? equal to # cache blocks in 2 rows

about $\frac{N}{2} \cdot N \cdot \frac{2N}{\text{block size}} = N^3 \div$ block size misses

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

$N^2/2$ total misses (before blocking: N^2)

about $2N \div$ block size misses from B per j-loop

$N^3 \div$ block size total misses (same as before blocking)

about $N \div$ block size misses from C per j-loop

$N^3 \div (2 \cdot$ block size$)$ total misses (before: $N^3 \div$ block size)

simple blocking – counting misses

```
for (int kk = 0; kk < N; kk += 2)
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+kk+0] * B[(kk+0)*N+j];
      C[i*N+j] += A[i*N+kk+1] * B[(kk+1)*N+j];
    }
```

$\frac{N}{2} \cdot N$ j-loop executions and (assuming N large):

about 1 misses from A per j-loop

$N^2/2$ total misses (before blocking: N^2)

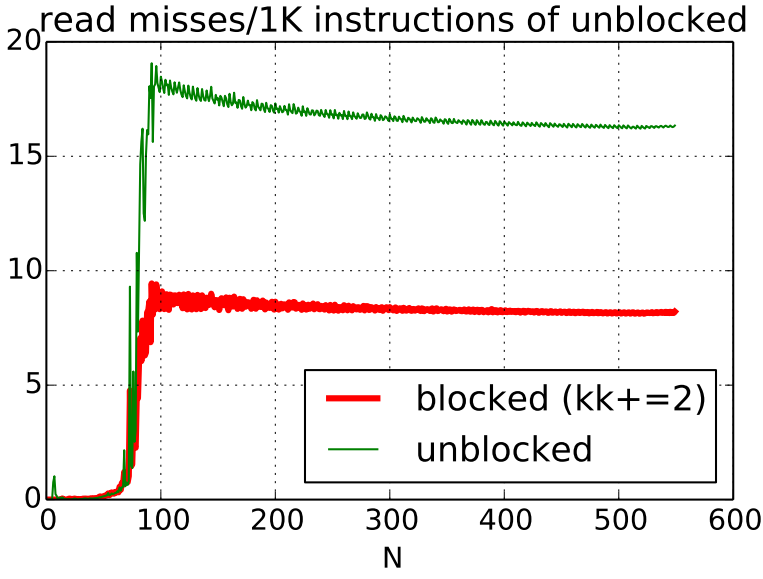
about $2N \div$ block size misses from B per j-loop

$N^3 \div$ block size total misses (same as before blocking)

about $N \div$ block size misses from C per j-loop

$N^3 \div (2 \cdot \text{block size})$ total misses (before: $N^3 \div$ block size)

improvement in read misses



simple blocking (2)

same thing for i in addition to k?

```
for (int kk = 0; kk < N; kk += 2) {
  for (int ii = 0; ii < N; ii += 2) {
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      for (int k = kk; k < kk + 2; ++k)
        for (int i = 0; i < ii + 2; ++i)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
  }
}
```

simple blocking — locality

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
  }
}
```

simple blocking — locality

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; i += 2) {
    /* load a block around Aik */
    for (int j = 0; j < N; ++j) {
      /* process a "block": */
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
  }
}
```

now: more temporal locality in B

previously: access B_{kj}, then don't use it again for a long time

simple blocking — counting misses for A

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely 2 misses per loop with A (2 cache blocks)

total misses: $\frac{N^2}{2}$ (same as only blocking in K)

simple blocking — counting misses for B

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely $2 \div$ block size misses per iteration with B

total misses: $\frac{N^3}{2 \cdot \text{block size}}$ (before: $\frac{N^3}{\text{block size}}$)

simple blocking — counting misses for C

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
```

$\frac{N}{2} \cdot \frac{N}{2}$ iterations of j loop

likely $\frac{2}{\text{block size}}$ misses per iteration with C

total misses: $\frac{N^3}{2 \cdot \text{block size}}$ (same as blocking only in K)

simple blocking — counting misses (total)

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      C_{i+0,j} += A_{i+0,k+0} * B_{k+0,j}
      C_{i+0,j} += A_{i+0,k+1} * B_{k+1,j}
      C_{i+1,j} += A_{i+1,k+0} * B_{k+0,j}
      C_{i+1,j} += A_{i+1,k+1} * B_{k+1,j}
    }
```

before:

A: $\frac{N^2}{2}$; B: $\frac{N^3}{1 \cdot \text{block size}}$; C $\frac{N^3}{1 \cdot \text{block size}}$

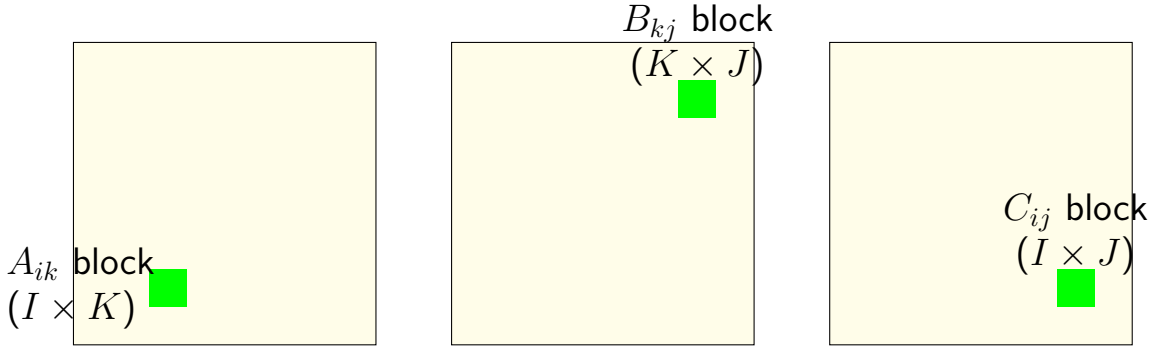
after:

A: $\frac{N^2}{2}$; B: $\frac{N^3}{2 \cdot \text{block size}}$; C $\frac{N^3}{2 \cdot \text{block size}}$

generalizing: divide and conquer

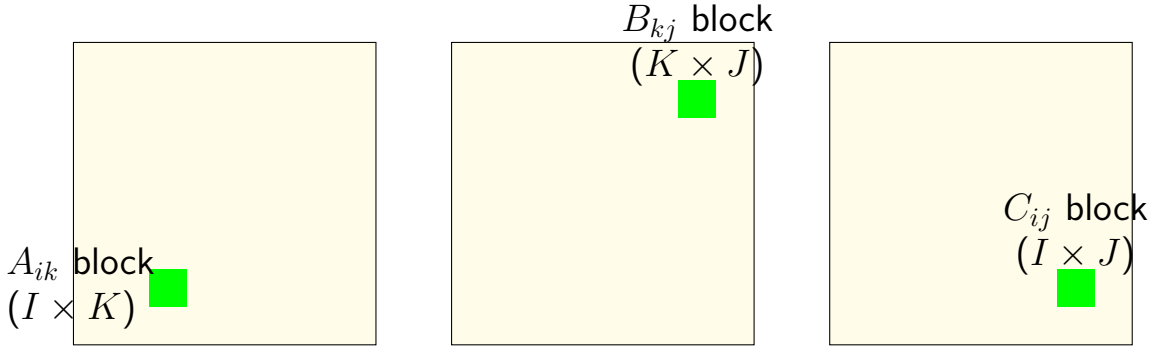
```
partial_matrixmultiply(float *A, float *B, float *C
                int startI, int endI, ...) {
  for (int i = startI; i < endI; ++i) {
    for (int j = startJ; j < endJ; ++j) {
      for (int k = startK; k < endK; ++k) {
        ...
}
matrix_multiply(float *A, float *B, float *C, int N) {
  for (int ii = 0; ii < N; ii += BLOCK_I)
    for (int jj = 0; jj < N; jj += BLOCK_J)
      for (int kk = 0; kk < N; kk += BLOCK_K)
        ...
        /* do everything for segment of A, B, C
           that fits in cache! */
        partial_matmul(A, B, C,
              ii, ii + BLOCK_I, jj, jj + BLOCK_J,
              kk, kk + BLOCK_K)
}
```

array usage: matrix block $\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



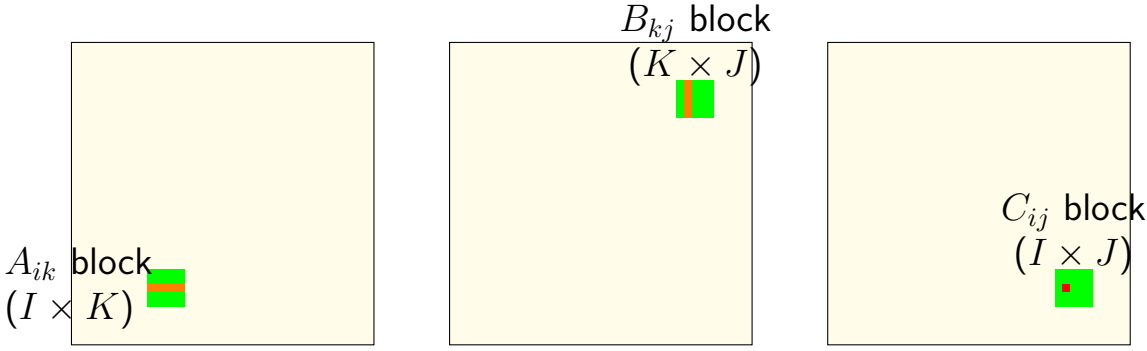
inner loops work on "matrix block" of A, B, C
rather than rows of some, little blocks of others
blocks fit into cache (b/c we choose I, K, J)
where previous rows might not

array usage: matrix block $\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



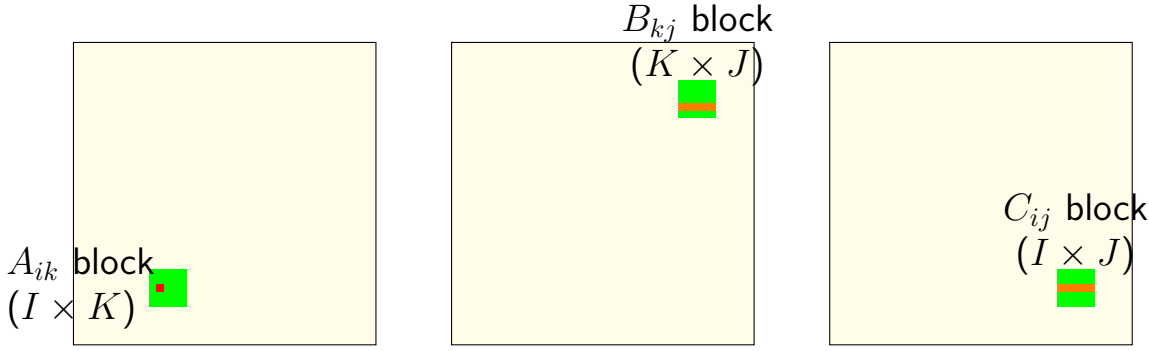
now (versus loop ordering example)
some spatial locality in A, B, and C
some temporal locality in A, B, and C

array usage: matrix block $\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



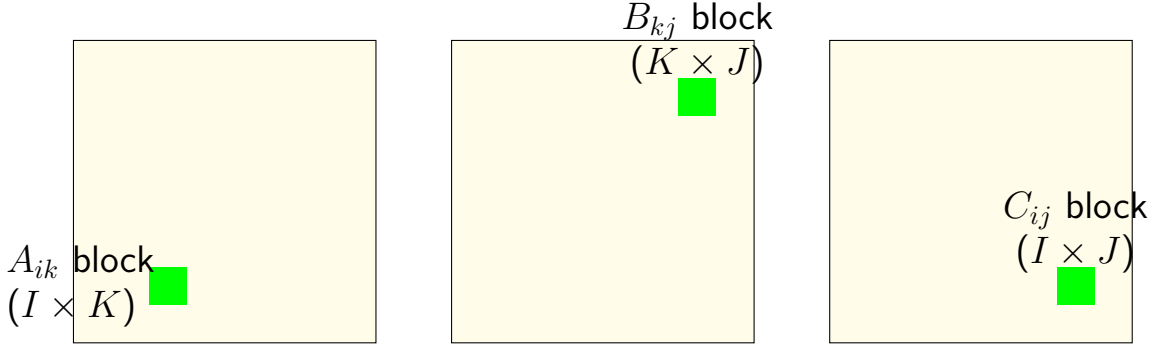
C_{ij} calculation uses strips from A, B
K calculations for one cache miss
good temporal locality!

array usage: matrix block $\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



A_{ik} used with entire strip of B J calculations for one cache miss
good temporal locality!

array usage: matrix block $\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ values need to be lodaed per "matrix block"
(assuming everything stays in cache)

cache blocking efficiency

for each of N^3/IJK matrix blocks:

load $I \times K$ elements of A_{ik}:

$\approx IK \div$ block size misses per matrix block

$\approx N^3/(J \cdot \text{blocksize})$ misses total

load $K \times J$ elements of A_{kj}:

$\approx N^3/(I \cdot \text{blocksize})$ misses total

load $I \times J$ elements of B_{ij}:

$\approx N^3/(K \cdot \text{blocksize})$ misses total

bigger blocks — more work per load!

catch: $IK + KJ + IJ$ elements must fit in cache

otherwise estimates above don't work

cache blocking rule of thumb

fill the most of the cache with useful data

and do as much work as possible from that

example: my desktop 32KB L1 cache

$I = J = K = 48$ uses $48^2 \times 3$ elements, or 27KB.

assumption: conflict misses aren't important

systematic approach

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; ++i) {
    A_ik loaded once in this loop:
    for (int j = 0; j < N; ++j)
      C_ij, B_kj loaded each iteration (if N big):
      B[i*N+j] += A[i*N+k] * A[k*N+j];
```

values from A_{ik} used N times per load

values from B_{kj} used 1 times per load

but good spatial locality, so cache block of B_{kj} together

values from C_{ij} used 1 times per load

but good spatial locality, so cache block of C_{ij} together

exercise: miss estimating (3)

```
for (int kk = 0; kk < 1000; kk += 10)
    for (int jj = 0; jj < 1000; jj += 10)
        for (int i = 0; i < 1000; i += 1)
            for (int j = jj; j < jj+10; j += 1)
                for (int k = kk; k < kk + 10; k += 1)
                    A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

assuming: cache not close to big enough to hold 1K elements, but big enough to hold 500 or so

estimate: *approximately* how many misses for A, B?

hint 1: part of A, B loaded in two inner-most loops only needs to be loaded once

loop ordering compromises

loop ordering forces compromises:

for k: for i: for j: c[i,j] += a[i,k] * b[j,k]

perfect temporal locality in a[i,k]

bad temporal locality for c[i,j], b[j,k]

perfect spatial locality in c[i,j]

bad spatial locality in b[j,k], a[i,k]

loop ordering compromises

loop ordering forces compromises:

for k: for i: for j: c[i,j] += a[i,k] * b[j,k]

perfect temporal locality in a[i,k]

bad temporal locality for c[i,j], b[j,k]

perfect spatial locality in c[i,j]

bad spatial locality in b[j,k], a[i,k]

cache blocking: work on blocks rather than rows/columns

- have some temporal, spatial locality in everything

cache blocking pattern

no perfect loop order? work on rectangular matrix blocks

size amount used in inner loops based on cache size

in practice:

- test performance to determine 'size' of blocks

sum array ASM (gcc 8.3 -Os)

```
long sum_array(long *values, int size) {
    long sum = 0;
    for (int i = 0; i < size; ++i) {
        sum += values[i];
    }
    return sum;
}
```

```
sum_array:
        xorl    %edx, %edx                      // i = 0
        xorl    %eax, %eax                      // sum = 0
loop:
        cmpq    %edx, %esi
        jle     endOfLoop                       // if (i < size) break
        addq    (%rsi,%rdx,8), %rax             // sum += values[i]
        incq    %rdx                            // i += 1
        jmp     loop
endOfLoop:
        ret
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop                   // if (i < size) break
        addq    (%rdi,%rdx,8), %rax         // sum += values[i]
        incq    %rdx                        // i += 1
        jmp     loop
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop                 // if (i < size) break
        addq    (%rdi,%rdx,8), %rax        // sum += values[i]
        addq    8(%rdi,%rdx,8), %rax       // sum += values[i+1]
        addq    $2, %rdx                  // i += 2
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop                   // if (i < size) break
        addq    (%rdi,%rdx,8), %rax         // sum += values[i]
        incq    %rdx                        // i += 1
        jmp     loop
endOfLoop:
```

`size` iterations $\times$ 5 instructions

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop                 // if (i < size) break
        addq    (%rdi,%rdx,8), %rax        // sum += values[i]
        addq    8(%rdi,%rdx,8), %rax       // sum += values[i+1]
        addq    $2, %rdx                  // i += 2
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

`size` $\div 2$ iterations $\times$ 6 instructions

loop unrolling (C)

```
for (int i = 0; i < N; ++i)
    sum += A[i];
```

```
int i;
for (i = 0; i + 1 < N; i += 2) {
    sum += A[i];
    sum += A[i+1];
}
// handle leftover, if needed
if (i < N)
    sum += A[i];
```

more loop unrolling (C)

```
int i;
for (i = 0; i + 4 <= N; i += 4) {
    sum += A[i];
    sum += A[i+1];
    sum += A[i+2];
    sum += A[i+3];
}
// handle leftover, if needed
for (; i < N; i += 1)
    sum += A[i];
```

loop unrolling performance

on my laptop with 992 elements (fits in L1 cache)

work/loop iteration	cycles/element	instructions/element
1	1.33	4.02
2	1.03	2.52
4	1.02	1.77
8	1.01	1.39
16	1.01	1.21
32	1.01	1.15

1.01 cycles/element — latency bound

loop unrolling on MM

original code:

```
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];
    }
```

---

loop unrolling in j loop (not cache blocking)

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; j += 2) {
      C[i*N+j] += A[i*N+k+0] * B[k*N+j];
      C[i*N+j+1] += A[i*N+k+1] * B[k*N+j+1];
    }
```

loop unrolling on MM

original code:

```
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];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
k=i=0, j=1: C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
k=i=0, j=3: C[0*N+3], A[0*N+0], B[0*N+3]
...

---

loop unrolling in j loop (not cache blocking)

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; j += 2) {
      C[i*N+j] += A[i*N+k+0] * B[k*N+j];
      C[i*N+j+1] += A[i*N+k+1] * B[k*N+j+1];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
C[0*N+3], A[0*N+0], B[0*N+3]
...

loop unrolling on MM

original code:

```
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];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
k=i=0, j=1: C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
k=i=0, j=3: C[0*N+3], A[0*N+0], B[0*N+3]
...

---

loop unrolling in j loop (not cache blocking)

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; j += 2) {
      C[i*N+j] += A[i*N+k+0] * B[k*N+j];
      C[i*N+j+1] += A[i*N+k+1] * B[k*N+j+1];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
C[0*N+3], A[0*N+0], B[0*N+3]
...

loop unrolling on MM

original code:

```
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];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
k=i=0, j=1: C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
k=i=0, j=3: C[0*N+3], A[0*N+0], B[0*N+3]
...

---

loop unrolling in j loop (not cache blocking)

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; j += 2) {
      C[i*N+j] += A[i*N+k+0] * B[k*N+j];
      C[i*N+j+1] += A[i*N+k+1] * B[k*N+j+1];
    }
```

access order:
k=i=j=0: C[0*N+0], A[0*N+0], B[0*N+0]
C[0*N+1], A[0*N+0], B[0*N+1]
k=i=0, j=2: C[0*N+2], A[0*N+0], B[0*N+2]
C[0*N+3], A[0*N+0], B[0*N+3]
...

partial cache blocking in MM

original code:

```
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];
    }
```

(incomplete) cache blocking with only k:

changes locality v. original (order of A, B, C accesses)

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

loop unrolling v cache blocking (0)

cache blocking for k only: (with teeny 1 by 1 by 2 matrix blocks)

changes locality v. original (order of A, B, C accesses)

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

---

with loop unrolling added afterwards:

same order of A, B, C accesses as above

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+k+0] * B[(k+0)*N+j];
      C[i*N+j] += A[i*N+k+1] * B[(k+1)*N+j];
    }
```

loop unrolling v cache blocking (0)

cache blocking for k only: (with teeny 1 by 1 by 2 matrix blocks)

changes locality v. original (order of A, B, C accesses)

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

---

with loop unrolling added afterwards:

same order of A, B, C accesses as above

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+k+0] * B[(k+0)*N+j];
      C[i*N+j] += A[i*N+k+1] * B[(k+1)*N+j];
    }
```

loop unrolling v cache blocking

cache blocking for k only (1x1x2 blocks) *and* then loop unrolling

same order of A, B, C accesses as original

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j) {
      C[i*N+j] += A[i*N+k+0] * B[(k+0)*N+j];
      C[i*N+j] += A[i*N+k+1] * B[(k+1)*N+j];
    }
```

---

versus pretty useless loop unrolling in k-loop

same order of A, B, C accesses as original

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      C[i*N+j] += A[i*N+k+0] * B[(k+0)*N+j];
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      C[i*N+j] += A[i*N+k+1] * B[(k+1)*N+j];
}
```

loop unrolling v cache blocking (1)

cache blocking for k,i only: (1 by 2 by 2 matrix blocks)

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

---

cache blocking for k,i and loop unrolling for i:

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j)
      for(int kk = k; kk < k + 2; ++kk) {
        C[(i+0)*N+j] += A[(i+0)*N+kk] * B[(kk)*N+j];
        C[(i+1)*N+j] += A[(i+1)*N+kk] * B[(kk)*N+j];
      }
```

exercise

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

Which of the following suggests changing order of memory accesses?

```
/* version A */
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; j += 2) {
    A[i*N+j] += B[i] + C[j]
    A[i*N+j+1] += B[i] + C[j+1]
  }
```

```
/* version B */
for (int i = 0; i < N; i += 2)
  for (int j = 0; j < N; j += 2) {
    A[i*N+j] += B[i] + C[j];
    A[i*N+j+1] += B[i] + C[j+1];
    A[(i+1)*N+j] += B[i+1] + C[j];
    A[(i+1)*N+j+1] += B[i+1] + C[j+1];
  }
```

interlude: real CPUs

modern CPUs:

execute multiple instructions at once

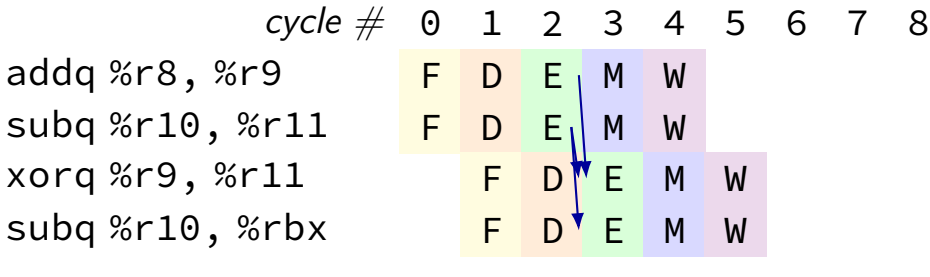
execute instructions out of order — whenever values available

beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex



beyond pipelining: out-of-order

find later instructions to do instead of stalling

lists of available instructions in pipeline registers

- take any instruction with available values

provide illusion that work is still done in order

- much more complicated hazard handling logic

cycle #	0	1	2	3	4	5	6	7	8	9	10	11
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	W	C				
`subq %r8, %r9`		F					D	E	W	C		
`addq %r10, %r11`			F	D	E	W					C	
`xorq %r12, %r13`				F	D	E	W					C

…

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- value in last stage may not be most up-to-date
- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- value in last stage may not be most up-to-date
- older value may be written back before newer value?

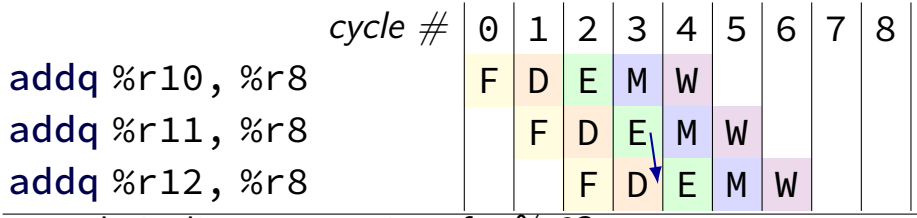
problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

read-after-write examples (1)



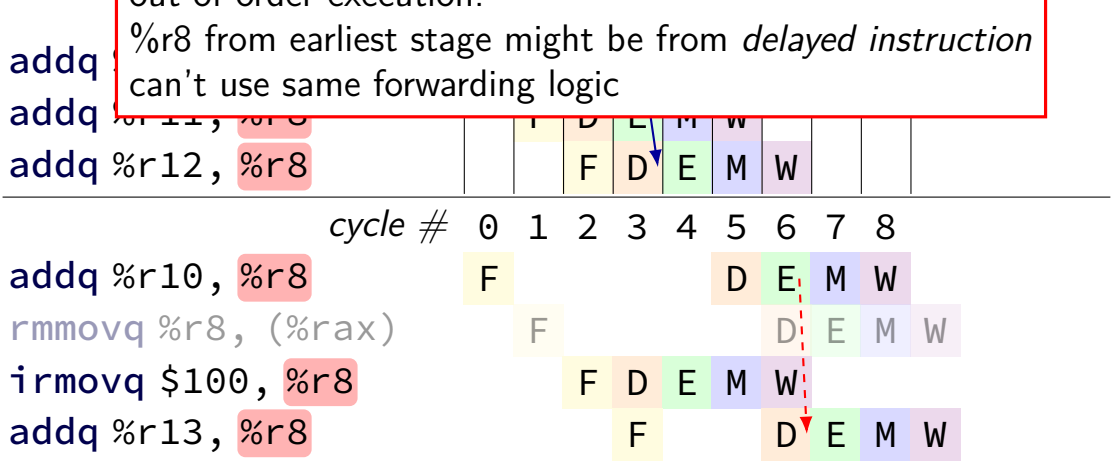
normal pipeline: two options for %r8?

choose the one from *earliest stage*

because it's from the most recent instruction

read-after-write examples (1)

out-of-order execution:
%r8 from earliest stage might be from *delayed instruction*
can't use same forwarding logic



register version tracking

goal: track different versions of registers

out-of-order execution: may compute versions at different times

only forward the correct version

strategy for doing this: preprocess instructions represent version info

makes forwarding, etc. lookup easier

rewriting hazard examples (1)

`addq %r10, %r8`	`addq %r10, %r8`$_{v1}$ $\rightarrow$ `%r8`$_{v2}$
`addq %r11, %r8`	`addq %r11, %r8`$_{v2}$ $\rightarrow$ `%r8`$_{v3}$
`addq %r12, %r8`	`addq %r12, %r8`$_{v3}$ $\rightarrow$ `%r8`$_{v4}$

read different version than the one written

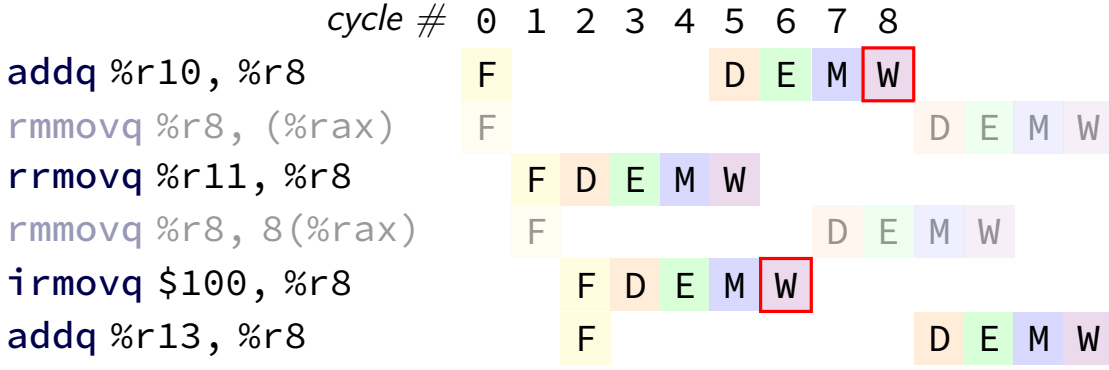
- represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

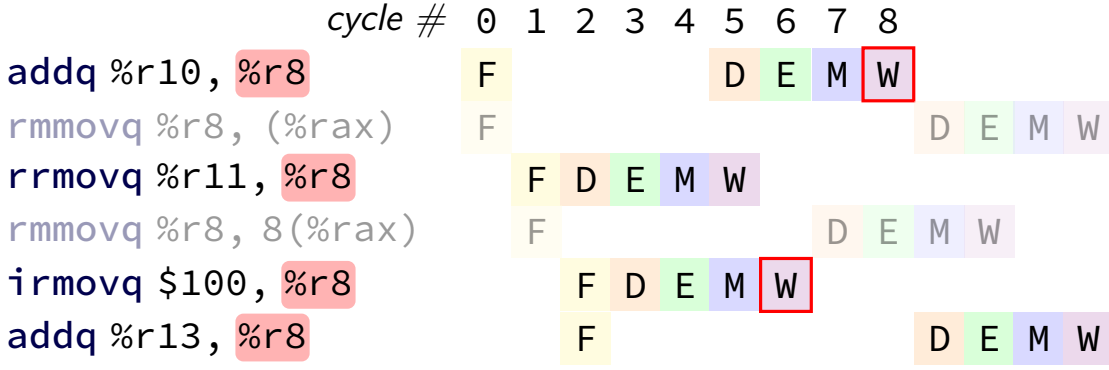
for now: version numbers

later: something simpler to implement

write-after-write example

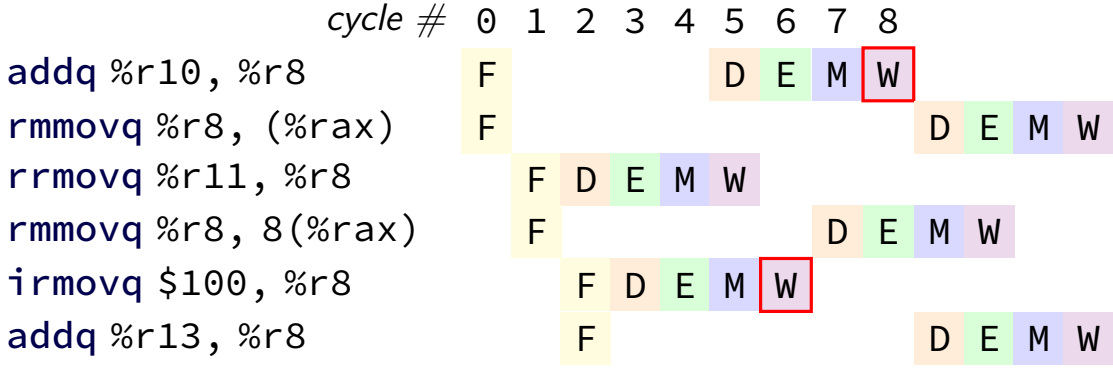


write-after-write example



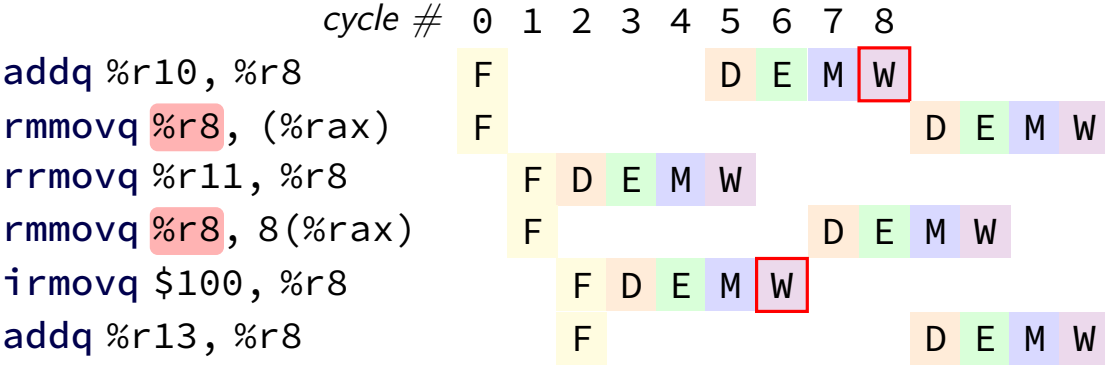
out-of-order execution:
if we don't do something, newest value could be overwritten!

write-after-write example



two instructions that haven't been started could need *different versions* of %r8!

write-after-write example



keeping multiple versions

for write-after-write problem: need to keep copies of multiple versions

both the new version and the old version needed by delayed instructions

for read-after-write problem: need to distinguish different versions

solution: have lots of extra registers

...and assign each version a new 'real' register

called register renaming

register renaming

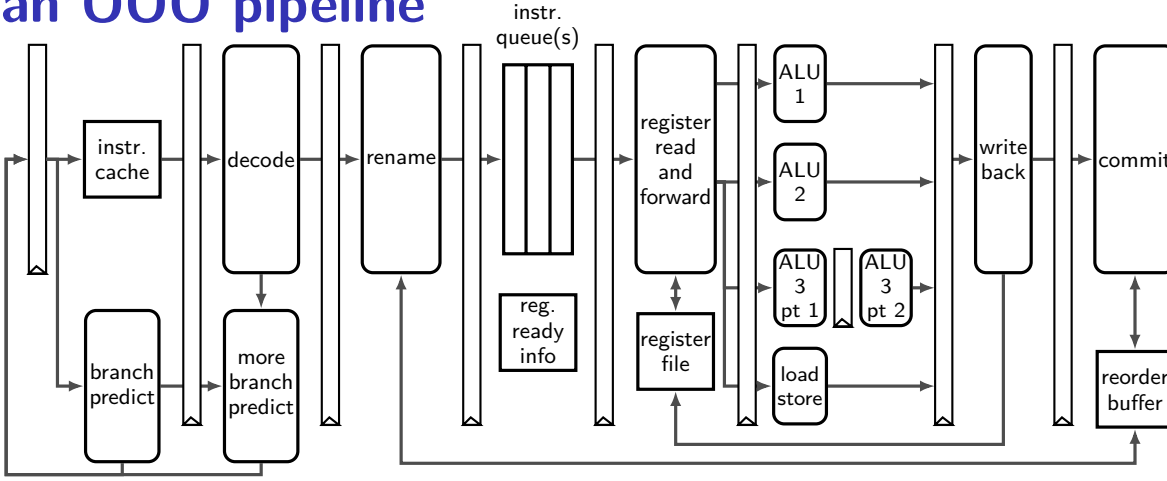
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

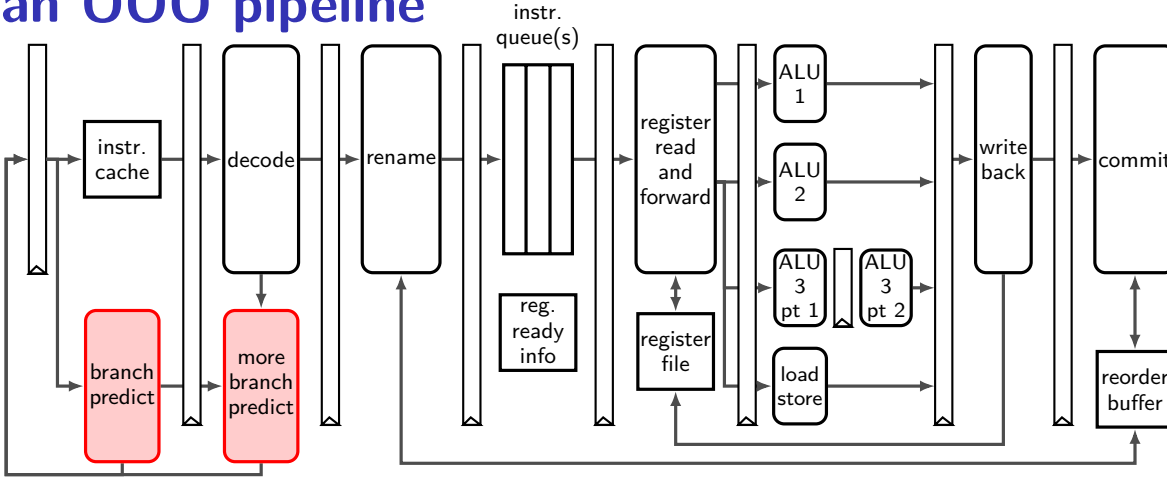
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

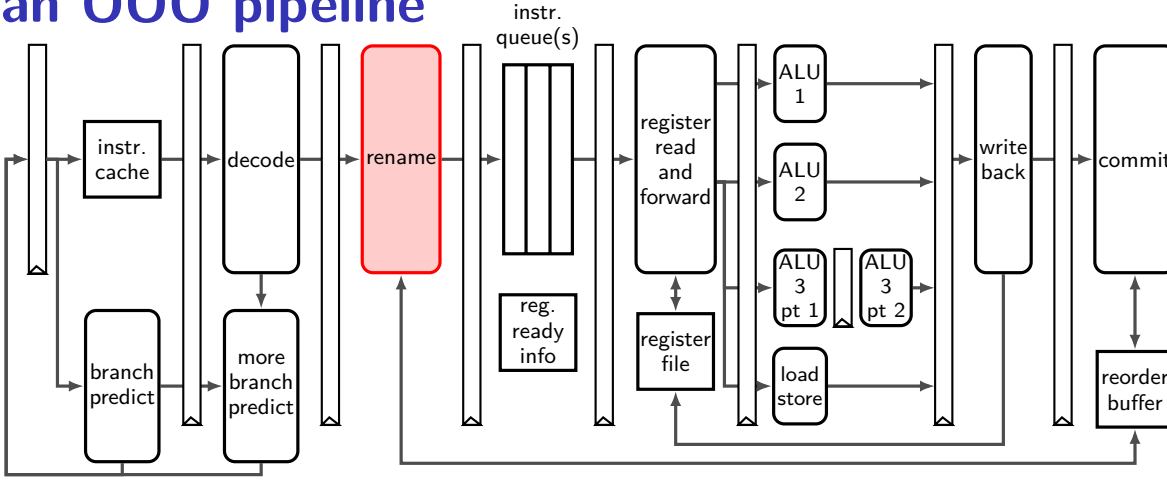


an OOO pipeline



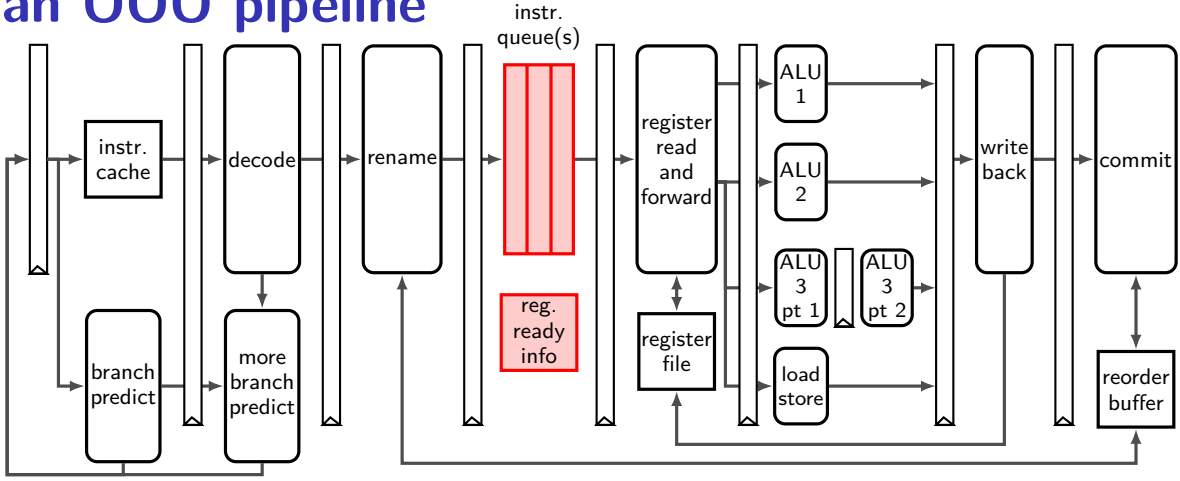
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



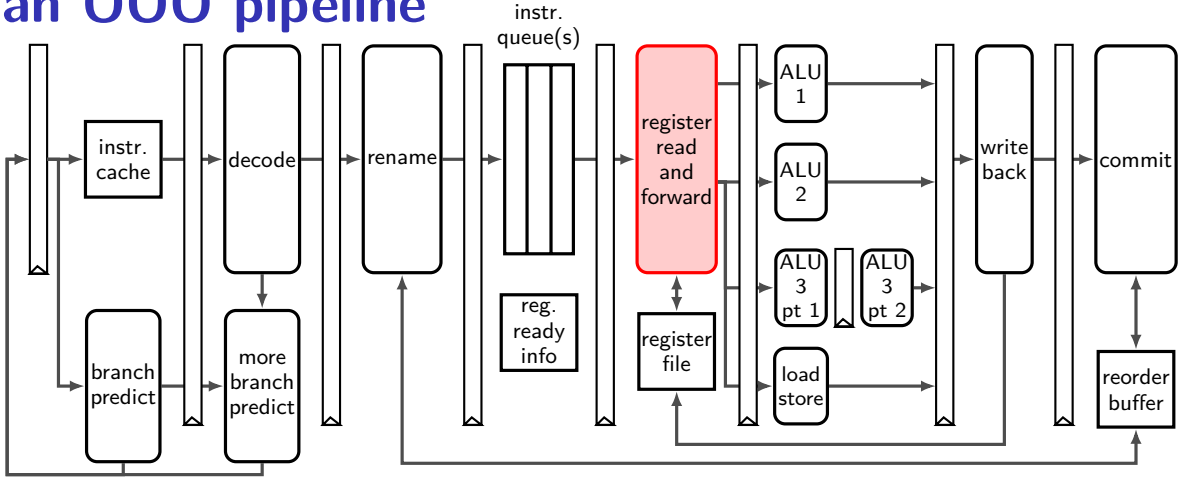
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



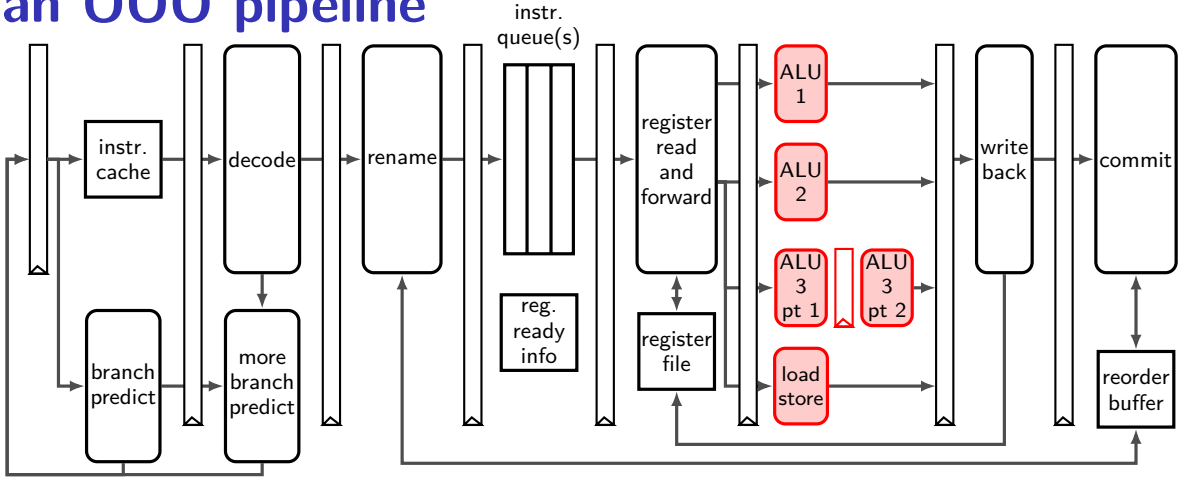
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



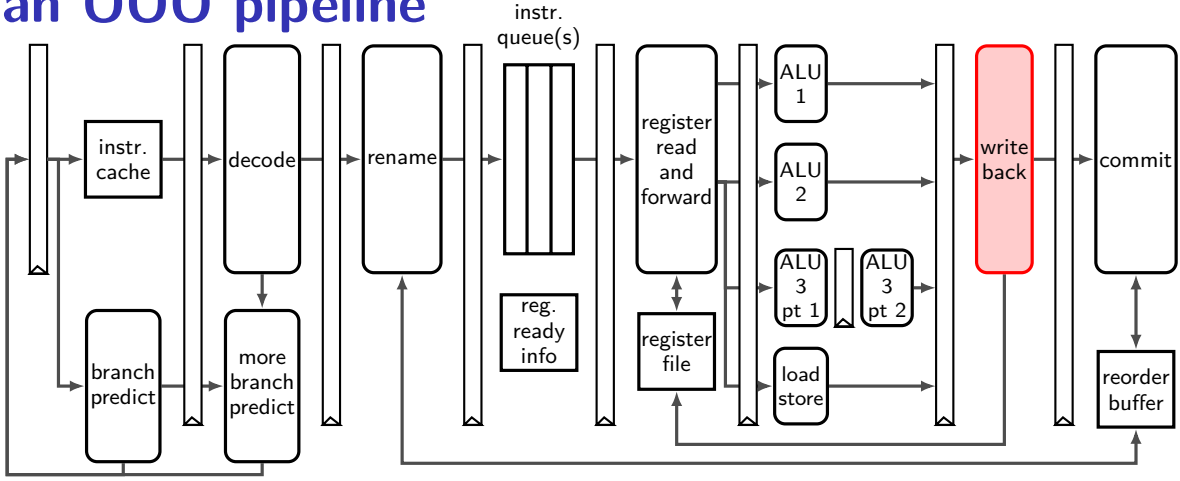
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



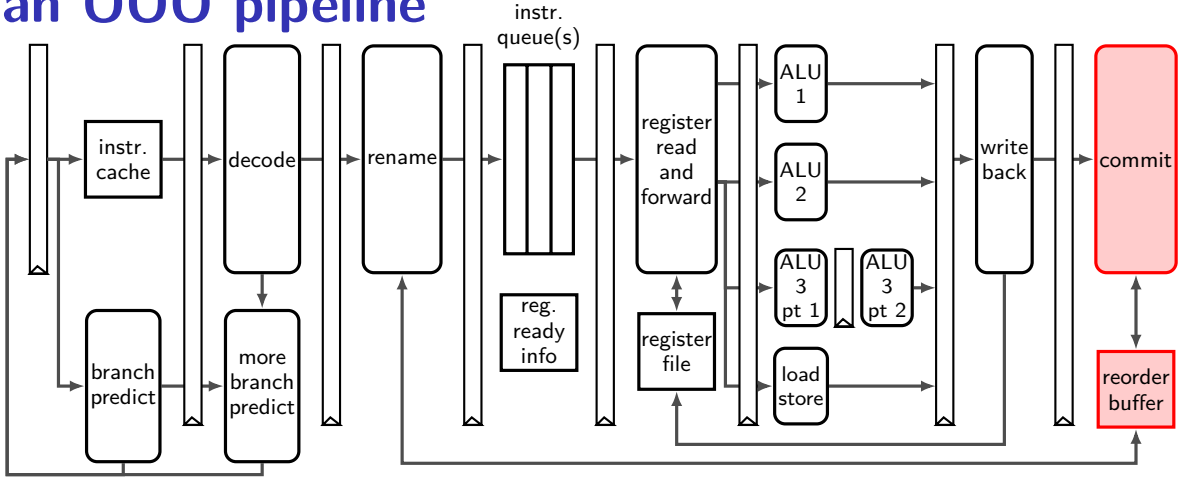
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



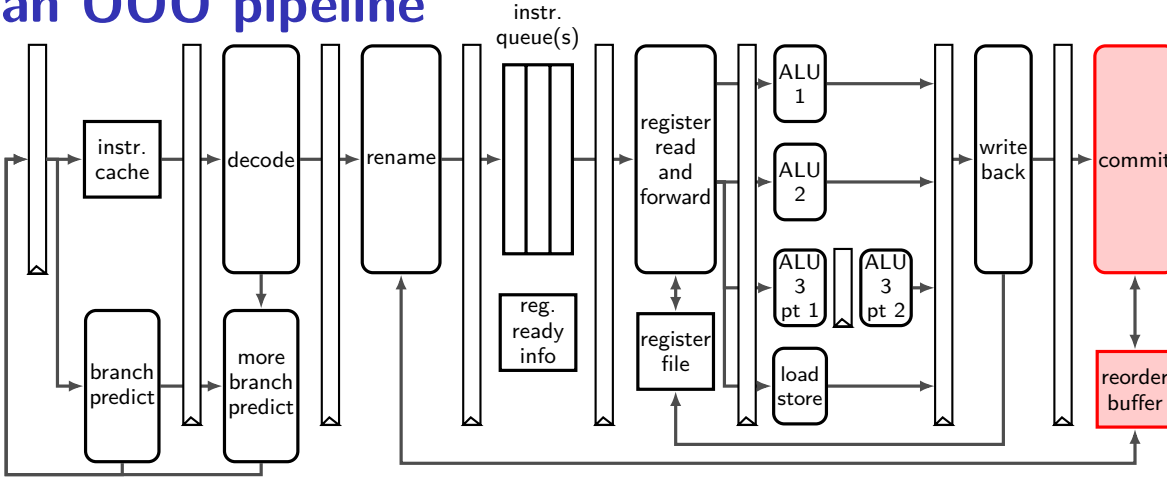
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



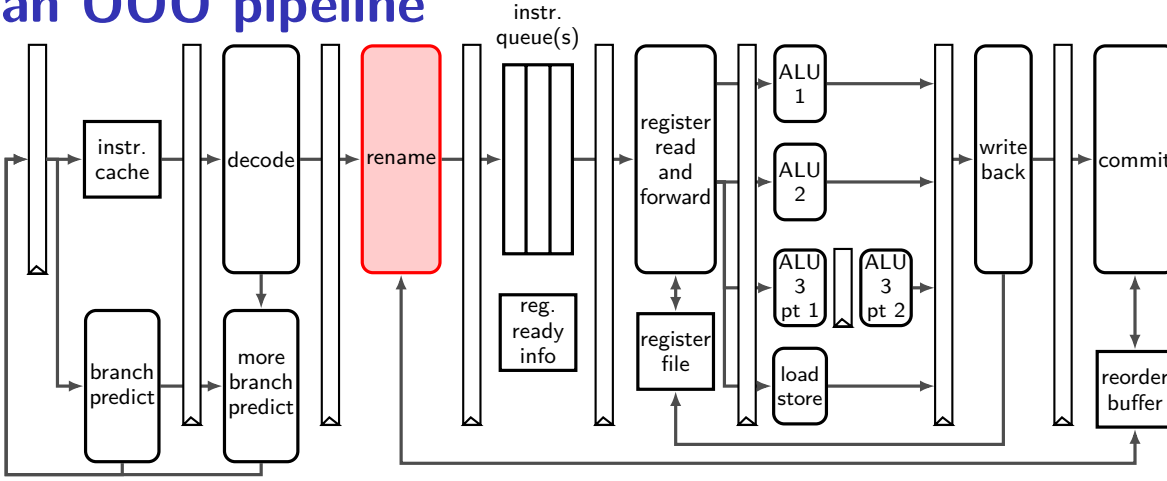
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

an OOO pipeline



commit stage also handles branch misprediction
reorder buffer tracks enough information to undo mispredicted instrs.

an OOO pipeline



register renaming

rename *architectural registers* to *physical registers*

architectural = part of instruction set architecture

different name for each version of architectural register

register renaming state

original

renamed

```
add %r10, %r8  ...
add %r11, %r8  ...
add %r12, %r8  ...
```

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original

```
add %r10, %r8  ...
add %r11, %r8  ...
add %r12, %r8  ...
```

renamed

table for architectural (external) and physical (internal) name (for next instr. to process)

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original

renamed

```
add %r10, %r8  ...
add %r11, %r8  ...
add %r12, %r8  ...
```

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

list of available physical registers
added to as instructions finish

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

%x18
%x20
%x21
%x23
%x24
…

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
add %x07, %x18 → %x20
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~ ~~%x18~~ ~~%x20~~
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~%x20~~
%x21
%x23
%x24
…

register renaming example (1)

original	renamed
`add %r10, %r8`	`add %x19, %x13` → `%x18`
`add %r11, %r8`	`add %x07, %x18` → `%x20`
`add %r12, %r8`	`add %x05, %x20` → `%x21`

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~~~%x18~~%x20~~%x21~~
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (1)

original	renamed
`add %r10, %r8`	`add %x19, %x13` → `%x18`
`add %r11, %r8`	`add %x07, %x18` → `%x20`
`add %r12, %r8`	`add %x05, %x20` → `%x21`

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13%x18%x20~~%x21
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free regs

%x18
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
rmmovq %x18, (%x04) → (memory)
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13 → %x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04) →` (memory)
`subq %r8, %r11`	
`mrmovq 8(%r11), %r11`	
`irmovq $100, %r8`	
`addq %r11, %r8`	

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02

%x21
%x23
%x24
…

could be that %rax = 8+%r11
could load before value written!
possible data hazard!
not handled via register renaming
option 1: run load+stores in order
option 2: compare load/store addresses

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
rmmovq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	~~%x07~~%x20
%r12	%x05
%r13	%x02

free regs

~~%x18~~
~~%x20~~
%x21
%x23
%x24
…

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13` → `%x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04)` → (memory)
`subq %r8, %r11`	`subq %x18, %x07` → `%x20`
`mrmovq 8(%r11), %r11`	`mrmovq 8(%x20),` (memory) → `%x21`
`irmovq $100, %r8`	
`addq %r11, %r8`	

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	~~%x07~~~~%x20~~~~%x21~~
%r12	%x05
%r13	%x02

free regs

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
rmmovq %r8, (%rax)
subq %r8, %r11
mrmovq 8(%r11), %r11
irmovq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
rmmovq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
mrmovq 8(%x20), (memory) → %x21
irmovq $100 → %x23
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~ ~~%x18~~ %x23
%r9	%x17
%r10	%x19
%r11	~~%x07~~ ~~%x20~~ %x21
%r12	%x05
%r13	%x02

free regs

~~%x18~~
~~%x20~~
~~%x21~~
~~%x23~~
%x24
…

register renaming example (2)

original	renamed
`addq %r10, %r8`	`addq %x19, %x13` → `%x18`
`rmmovq %r8, (%rax)`	`rmmovq %x18, (%x04)` → (memory)
`subq %r8, %r11`	`subq %x18, %x07` → `%x20`
`mrmovq 8(%r11), %r11`	`mrmovq 8(%x20),` (memory) → `%x21`
`irmovq $100, %r8`	`irmovq $100` → `%x23`
`addq %r11, %r8`	`addq %x21, %x23` → `%x24`

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~~~%x18~~~~%x23~~%x24
%r9	%x17
%r10	%x19
%r11	~~%x07~~~~%x20~~%x21
%r12	%x05
%r13	%x02

free regs

~~%x18~~
~~%x20~~
~~%x21~~
~~%x23~~
~~%x24~~
…

register renaming exercise

original

```
addq %r8, %r9
movq $100, %r10
subq %r10, %r8
xorq %r8, %r9
andq %rax, %r9
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x21
%r12	%x05
%r13	%x02
…	…

free regs

%x18
%x20
%x21
%x23
%x24
…

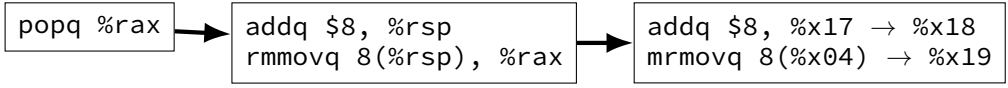
register renaming: missing pieces

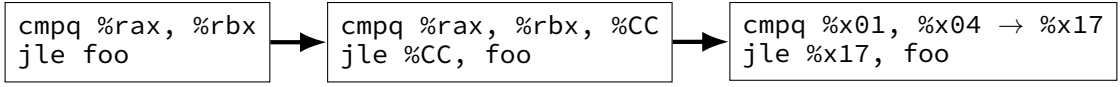
what about "hidden" inputs like `%rsp`, condition codes?

one solution: translate to intructions with additional register parameters

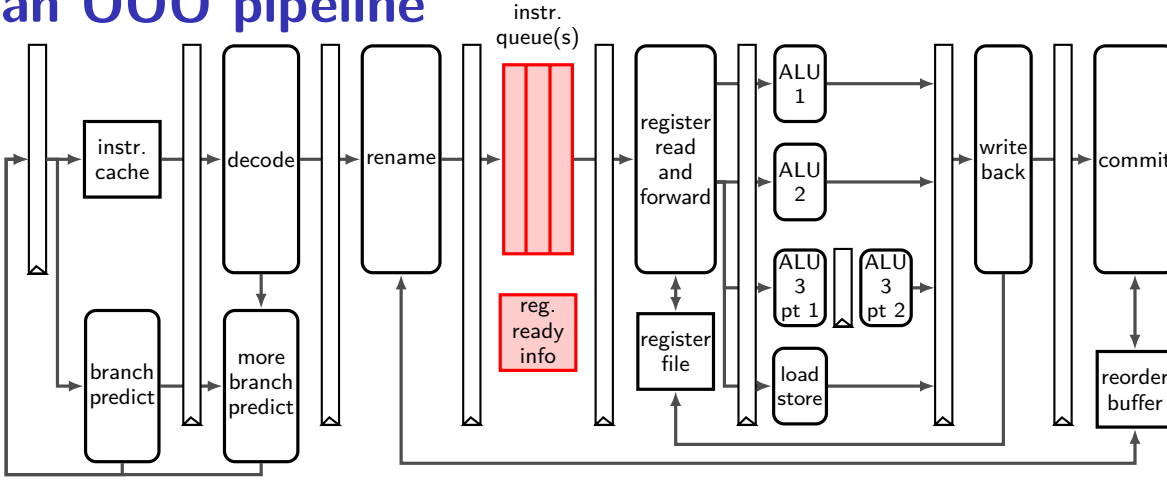
- making %rsp explicit parameter
- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones





an OOO pipeline



instruction queue and dispatch

instruction queue

#	*instruction*
1	`addq %x01, %x05 → %x06`
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit

ALU 1

ALU 2

...

instruction queue and dispatch

instruction queue

#	*instruction*
1	`addq %x01, %x05 → %x06`
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*
ALU 1	**1**
ALU 2	

...

instruction queue and dispatch

instruction queue

#	*instruction*
1	`addq %x01, %x05 → %x06`
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	...
ALU 1	1	
ALU 2		

instruction queue and dispatch

instruction queue

#	*instruction*
1	`addq %x01, %x05 → %x06`
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*
ALU 1	1
ALU 2	—

...

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle#* 1	2
ALU 1	1	**2**
ALU 2	—	—

...

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle#* 1	2	3
ALU 1	1	2	**3**
ALU 2	—	—	—

...

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle#* 1	2	3	...
ALU 1	1	2	3	
ALU 2	—	—	—	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x09 → %x11`
8	`addq %x03, %x10 → %x12`
9	`cmpq %x04, %x11 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle#* 1	2	3	4	...
ALU 1	1	2	3	**4**	
ALU 2	—	—	—	**6**	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~addq %x01, %x05 → %x06~~
~~2~~	~~addq %x02, %x06 → %x07~~
~~3~~	~~addq %x03, %x07 → %x08~~
~~4~~	~~cmpq %x04, %x08 → %x09.cc~~
5	jne %x09.cc, ...
~~6~~	~~addq %x01, %x08 → %x10~~
7	addq %x02, %x09 → %x11
8	addq %x03, %x10 → %x12
9	cmpq %x04, %x11 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle#* 1	2	3	4	...
ALU 1	1	2	3	4	
ALU 2	—	—	—	6	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~addq %x01, %x05 → %x06~~
~~2~~	~~addq %x02, %x06 → %x07~~
~~3~~	~~addq %x03, %x07 → %x08~~
~~4~~	~~cmpq %x04, %x08 → %x09.cc~~
~~5~~	~~jne %x09.cc, ...~~
~~6~~	~~addq %x01, %x08 → %x10~~
~~7~~	~~addq %x02, %x09 → %x11~~
8	addq %x03, %x10 → %x12
9	cmpq %x04, %x11 → %x13.cc
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	pending
%x12	pending
%x13	pending
…	…

execution unit	*cycle#* 1	2	3	4	5	…
ALU 1	1	2	3	4	**5**	
ALU 2	—	—	—	6	**7**	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
~~4~~	~~`cmpq %x04, %x08 → %x09.cc`~~
~~5~~	~~`jne %x09.cc, ...`~~
~~6~~	~~`addq %x01, %x08 → %x10`~~
~~7~~	~~`addq %x02, %x09 → %x11`~~
~~8~~	~~`addq %x03, %x10 → %x12`~~
9	`cmpq %x04, %x11 → %x13.cc`
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	~~pending~~ ready
%x12	pending
%x13	pending
…	…

execution unit	*cycle#* 1	2	3	4	5	6	…
ALU 1	1	2	3	4	5	**8**	
ALU 2	—	—	—	6	7	—	

instruction queue and dispatch

instruction queue

#	*instruction*
1	~~addq %x01, %x05 → %x06~~
2	~~addq %x02, %x06 → %x07~~
3	~~addq %x03, %x07 → %x08~~
4	~~cmpq %x04, %x08 → %x09.cc~~
5	~~jne %x09.cc, ...~~
6	~~addq %x01, %x08 → %x10~~
7	~~addq %x02, %x09 → %x11~~
8	~~addq %x03, %x10 → %x12~~
9	~~cmpq %x04, %x11 → %x13.cc~~
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	pending
…	…

execution unit	*cycle#* 1	2	3	4	5	6	7	…
ALU 1	1	2	3	4	5	8	**9**	
ALU 2	—	—	—	6	7	—	**…**	

instruction queue and dispatch

instruction queue

#	*instruction*
1	~~`addq %x01, %x05 → %x06`~~
2	~~`addq %x02, %x06 → %x07`~~
3	~~`addq %x03, %x07 → %x08`~~
4	~~`cmpq %x04, %x08 → %x09.cc`~~
5	~~`jne %x09.cc, ...`~~
6	~~`addq %x01, %x08 → %x10`~~
7	~~`addq %x02, %x09 → %x11`~~
8	~~`addq %x03, %x10 → %x12`~~
9	~~`cmpq %x04, %x11 → %x13.cc`~~
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ ready
…	…

execution unit	*cycle#* 1	2	3	4	5	6	7	…
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	…	

instruction queue and dispatch

instruction queue

#	*instruction*
1	`mrmovq (%x04) → %x06`
2	`mrmovq (%x05) → %x07`
3	`addq %x01, %x02 → %x08`
4	`addq %x01, %x06 → %x09`
5	`addq %x01, %x07 → %x10`
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	
%x07	
%x08	
%x09	
%x10	
…	…

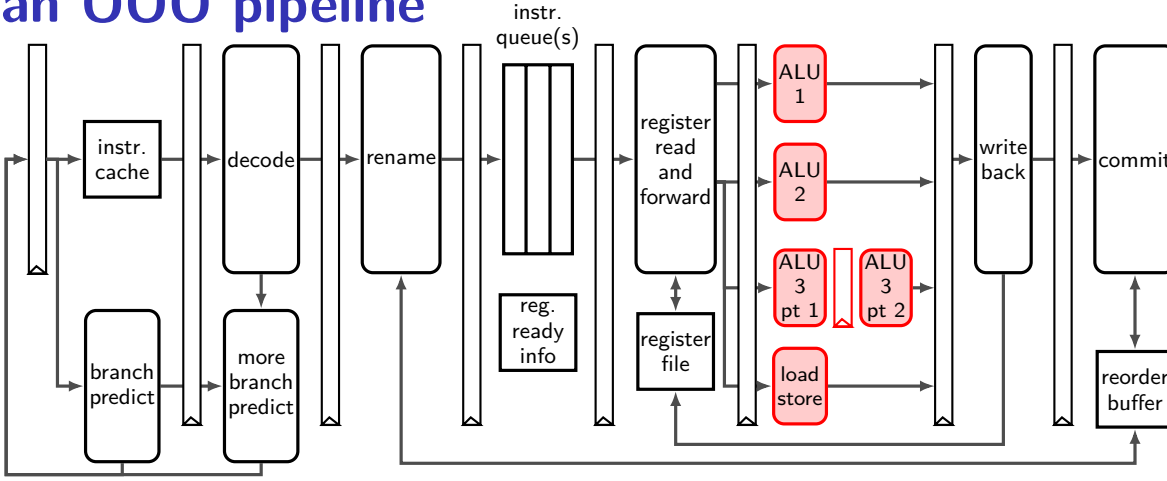
execution unit *cycle#* 1 2 3 4 5 6 7 …

ALU

data cache

assume 1 cycle/access

an OOO pipeline



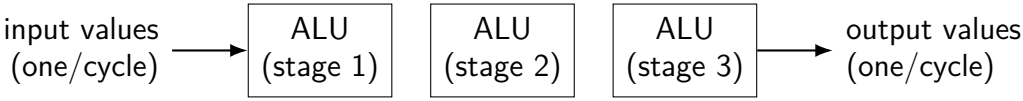
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



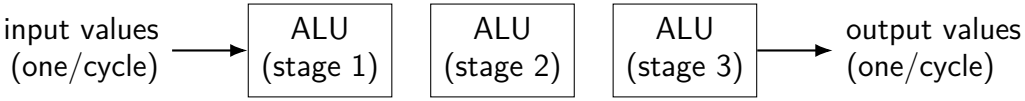
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

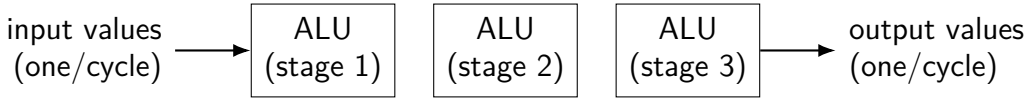
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

3×3 cycles + any time to forward values

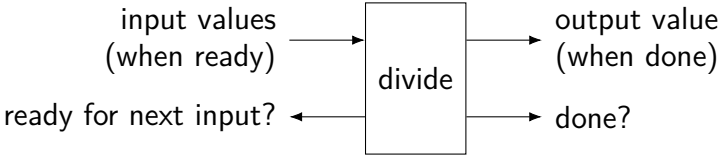
no parallelism!

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit	**cycle#** 1
ALU 1 (add, cmp, jxx)	1
ALU 2 (add, cmp, jxx)	–
ALU 3 (mul) start	2
ALU 3 (mul) end	2

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	pending (still)
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit	*cycle#* *1*	**2**		
ALU 1 (add, cmp, jxx)	1	**6**		
ALU 2 (add, cmp, jxx)	–	**–**		
ALU 3 (mul) start	2	**3**		
ALU 3 (mul) end		2	**3**	

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
~~6~~	~~`add %x01, %x03 → %x11`~~
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	pending (still)
%x09	pending
%x10	pending
%x11	~~pending~~ ready
%x12	pending
%x13	pending
%x14	pending
...	...

...

execution unit	*cycle#* *1*	*2*	**3**
ALU 1 (add, cmp, jxx)	1	6	–
ALU 2 (add, cmp, jxx)	–	–	–
ALU 3 (mul) start	2	3	**7**
ALU 3 (mul) end		2	3 **7**

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
5	`jle %x09.cc, ...`
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	pending (still)
%x13	pending
%x14	pending
...	...

execution unit	*cycle#* 1	2	3	**4**	
ALU 1 (add, cmp, jxx)	1	6	–	**4**	
ALU 2 (add, cmp, jxx)	–	–	–	**–**	
ALU 3 (mul) start	2	3	7	**8**	
ALU 3 (mul) end		2	3	7	**8**

...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	pending (still)
%x14	pending
...	...

execution unit	*cycle#*	*1*	*2*	*3*	*4*	**5**	...
ALU 1 (add, cmp, jxx)		1	6	–	4	**5**	
ALU 2 (add, cmp, jxx)		–	–	–	–	–	
ALU 3 (mul) start		2	3	7	8	–	
ALU 3 (mul) end			2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
~~8~~	~~`imul %x03, %x08 → %x13`~~
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ ready
%x14	pending
…	…

execution unit	*cycle#*	*1*	*2*	*3*	*4*	*5*	…
ALU 1 (add, cmp, jxx)		1	6	–	4	5	
ALU 2 (add, cmp, jxx)		–	–	–	–	–	
ALU 3 (mul) start		2	3	7	8	–	
ALU 3 (mul) end			2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

#	instruction
~~1~~	~~add %x01, %x02 → %x03~~
~~2~~	~~imul %x04, %x05 → %x06~~
~~3~~	~~imul %x03, %x07 → %x08~~
~~4~~	~~cmp %x03, %x08 → %x09.cc~~
~~5~~	~~jle %x09.cc, ...~~
~~6~~	~~add %x01, %x03 → %x11~~
~~7~~	~~imul %x04, %x06 → %x12~~
~~8~~	~~imul %x03, %x08 → %x13~~
~~9~~	~~cmp %x11, %x13 → %x14.cc~~
10	jle %x14.cc, ...
...	...

scoreboard

reg	status
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ ready
%x14	~~pending~~ ready
...	...

execution unit	*cycle#* 1	2	3	4	5	**6**	...
ALU 1 (add, cmp, jxx)	1	6	–	4	5	**9**	
ALU 2 (add, cmp, jxx)	–	–	–	–	–	**–**	
ALU 3 (mul) start	2	3	7	8	–		
ALU 3 (mul) end		2	3	7	8		

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~add %x01, %x02 → %x03~~
~~2~~	~~imul %x04, %x05 → %x06~~
~~3~~	~~imul %x03, %x07 → %x08~~
~~4~~	~~cmp %x03, %x08 → %x09.cc~~
~~5~~	~~jle %x09.cc, ...~~
~~6~~	~~add %x01, %x03 → %x11~~
~~7~~	~~imul %x04, %x06 → %x12~~
~~8~~	~~imul %x03, %x08 → %x13~~
~~9~~	~~cmp %x11, %x13 → %x14.cc~~
~~10~~	~~jle %x14.cc, ...~~
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ ready
%x14	~~pending~~ ready
...	...

execution unit	*cycle#* 1	2	3	4	5	6	7	...
ALU 1 (add, cmp, jxx)	1	6	–	4	5	9	**10**	
ALU 2 (add, cmp, jxx)	–	–	–	–	–	–	–	
ALU 3 (mul) start	2	3	7	8	–			
ALU 3 (mul) end		2	3	7	8			

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

- e.g. Intel Skylake: approx 16 cycles (v. 2 for pipehw2 CPU)

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

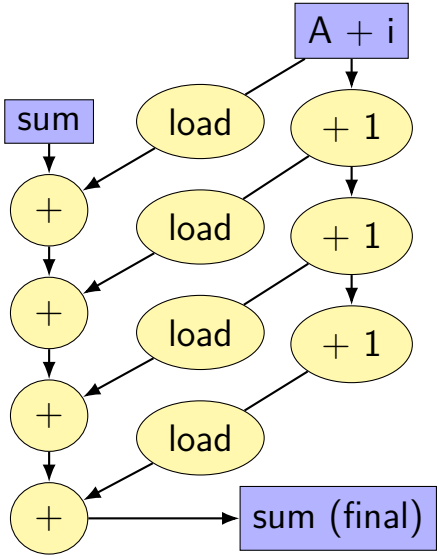
need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

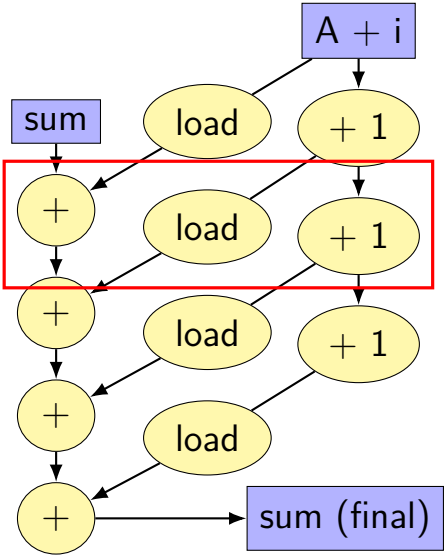
- e.g. Intel Skylake: approx 16 cycles (v. 2 for pipehw2 CPU)

data flow model and limits



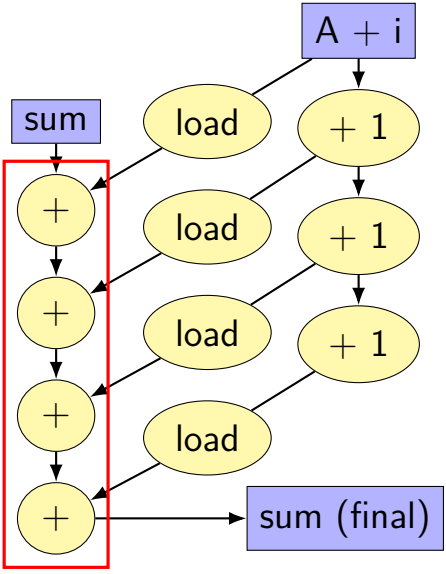

```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



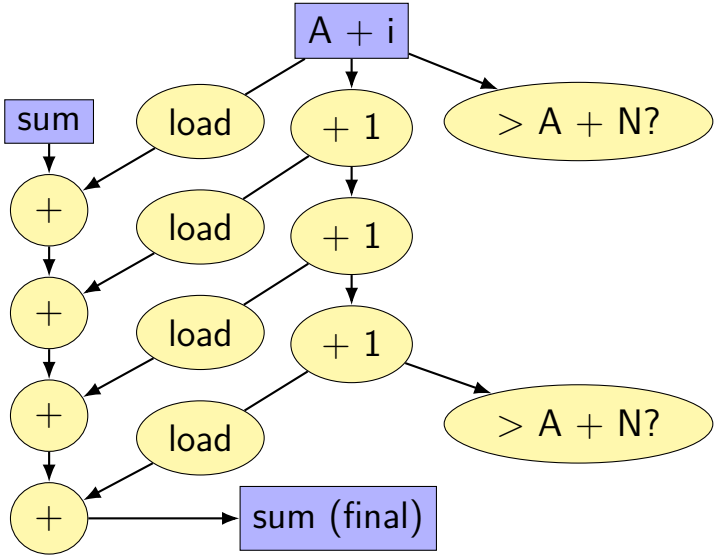
three ops/cycle (if each one cycle)

data flow model and limits



need to do additions
one-at-a-time
book's name: critical path
time needed: sum of latencies

data flow model and limits



reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

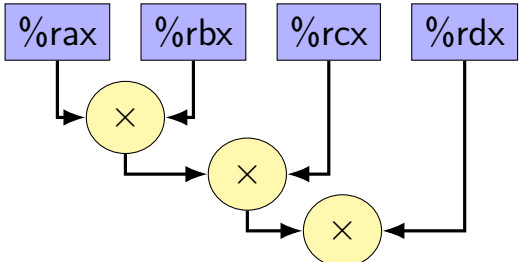
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

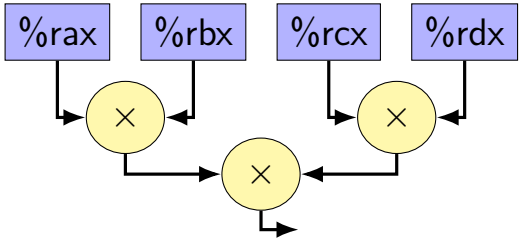
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



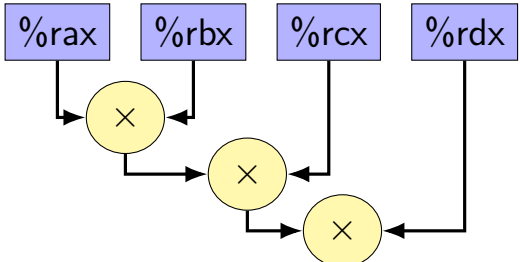
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

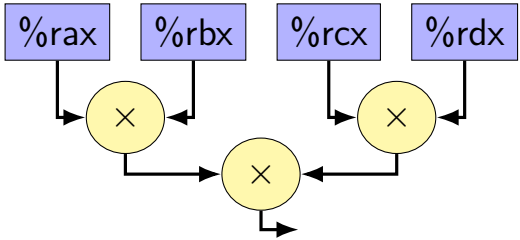
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



reassociation

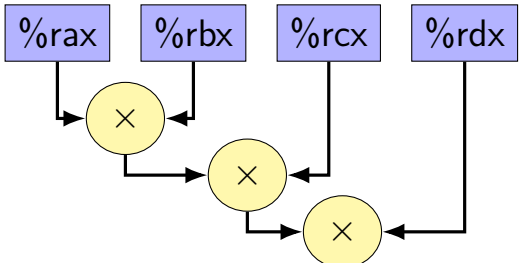
assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

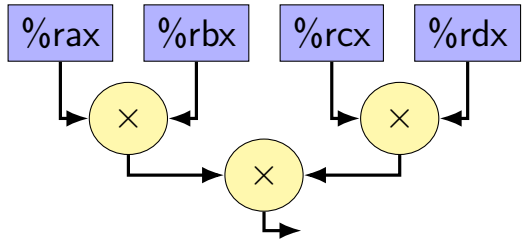
15 cycles



$(a \times b) \times (c \times d)$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```

11 cycles



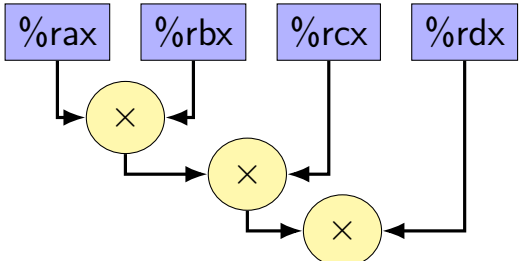
reassociation

assume a single pipelined, 5-cycle latency multiplier

exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

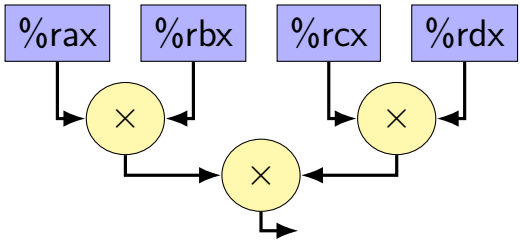
$((a \times b) \times c) \times d$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

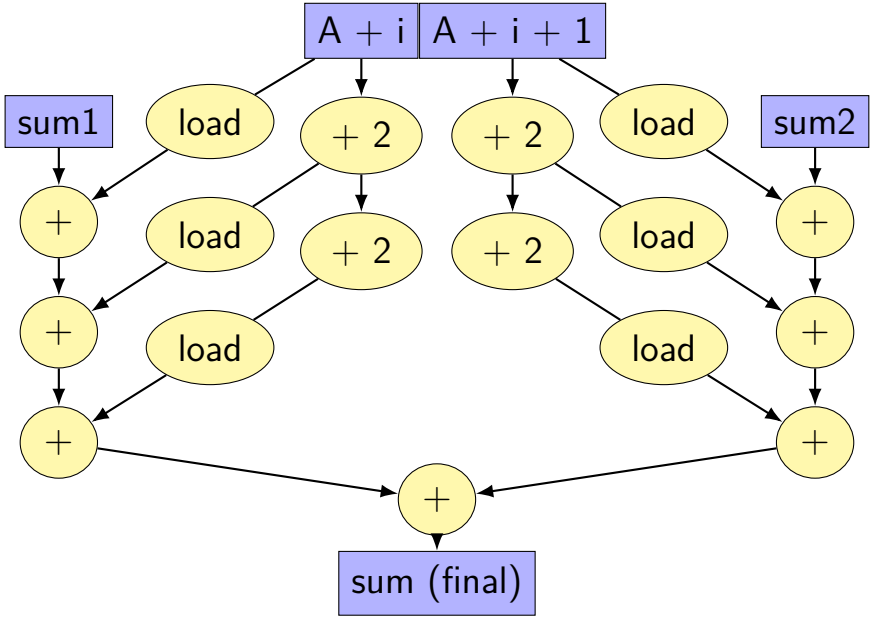


$(a \times b) \times (c \times d)$

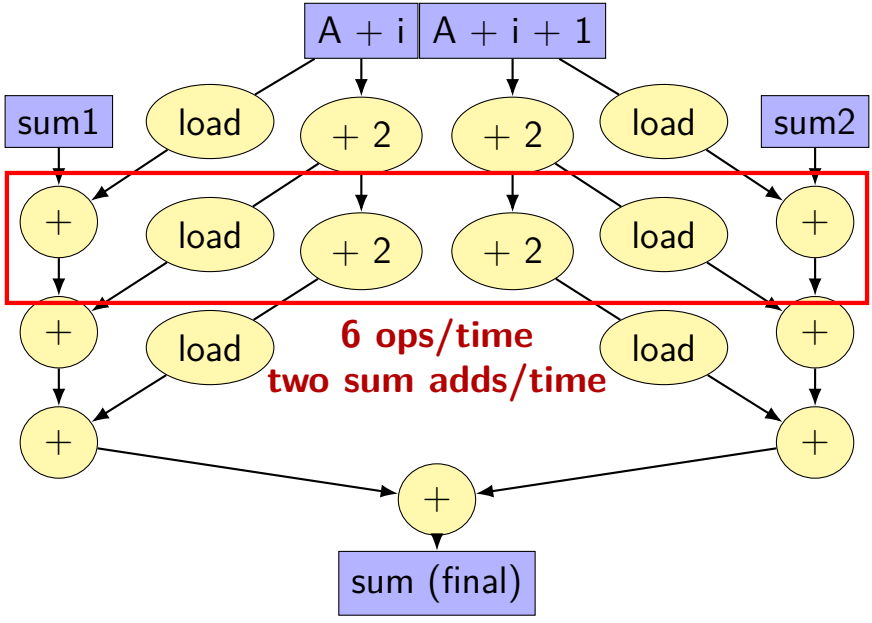
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



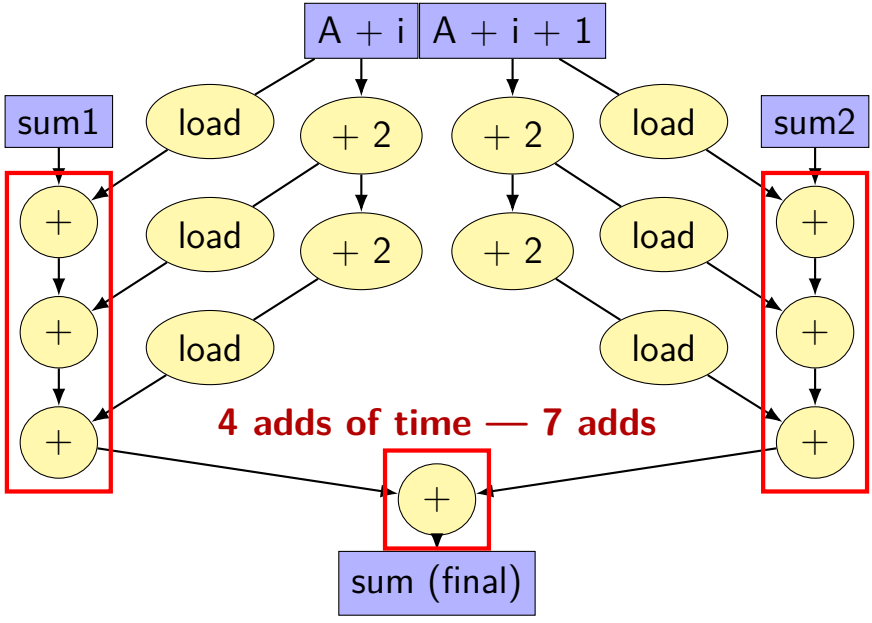
better data-flow



better data-flow



better data-flow



multiple accumulators

```
int i;
long sum1 = 0, sum2 = 0;
for (i = 0; i + 1 < N; i += 2) {
    sum1 += A[i];
    sum2 += A[i+1];
}
// handle leftover, if needed
if (i < N)
    sum1 += A[i];
sum = sum1 + sum2;
```

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

8 accumulator assembly

```
sum1 += A[i + 0];
sum2 += A[i + 1];
...
...
```

```
addq    (%rdx), %rax        // sum1 +=
addq    8(%rdx), %rcx       // sum2 +=
subq    $-128, %rdx         // i +=
addq    -112(%rdx), %rbx    // sum3 +=
addq    -104(%rdx), %r11    // sum4 +=
...
....
cmpq  %r14, %rdx
```

register for each of the `sum1`, `sum2`, ...variables:

16 accumulator assembly

compiler runs out of registers

starts to use the stack instead:

```
movq    32(%rdx), %rax  // get A[i+13]
addq    %rax, -48(%rsp) // add to sum13 on stack
```

code does extra cache accesses

also — already using all the adders available all the time

so performance increase not possible

multiple accumulators performance

on my laptop with 992 elements (fits in L1 cache)

16x unrolling, variable number of accumulators

accumulators	cycles/element	instructions/element
1	1.01	1.21
2	0.57	1.21
4	0.57	1.23
8	0.59	1.24
16	0.76	1.57

starts hurting after too many accumulators

why?

maximum performance

2 additions per element:

- one to add to sum
- one to compute address (part of mov)

3/16 add/sub/cmp + 1/16 branch per element:

- over 16 because loop unrolled 16 times
- loop overhead
- compiler not as efficient as it could have been

2+3/16+1/16 = 2+1/4 instructions per element

probably 2+1/4 microinstructions, too

- cmp+jXX apparently becomes 1 microinstruction (on this Intel CPU)
- probably extra microinstruction for load in add

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

hardware limits on my machine

4(?) register renamings per cycle

(Intel doesn't really publish exact numbers here...)

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions commited/cycle

4 (add or cmp+branch executed)/cycle

$(2 + 1/4) \div 4 \approx 0.57$ cycles/element

getting over this limit

the $+1/4$ was from loop overhead

solution: more loop unrolling!

common theme with optimization:

fix one bottleneck (need to do adds one after the other)

find another bottleneck

loop unrolling performance

on my laptop with 992 elements (fits in L1 cache)

work/loop iteration	cycles/element	instructions/element
1	1.33	4.02
2	1.03	2.52
4	1.02	1.77
8	1.01	1.39
16	1.01	1.21
32	1.01	1.15

1.01 cycles/element — latency bound

backup slides

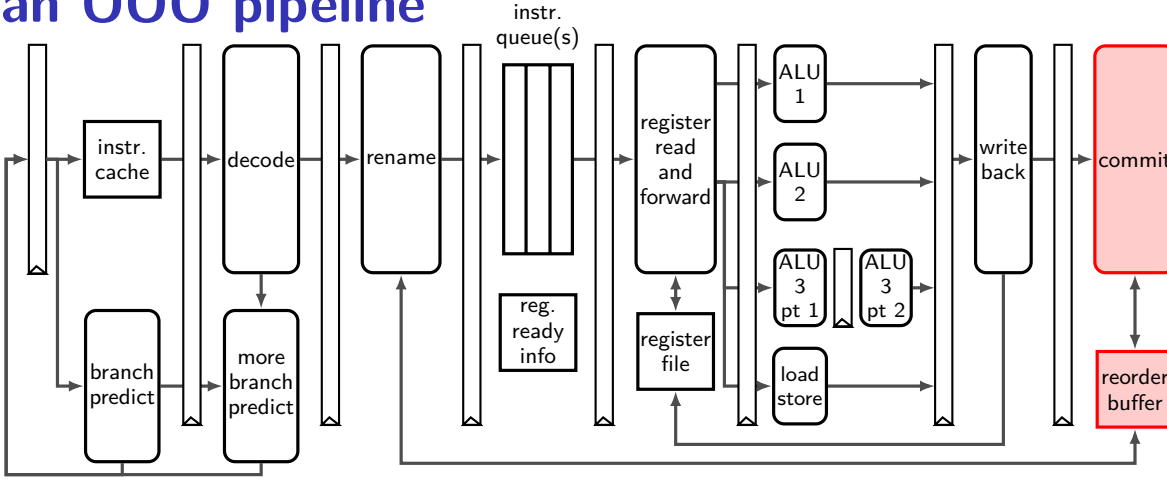
locality exercise (2)

```
/* version 2 */
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        A[i] += B[j] * C[i * N + j]

/* version 3 */
for (int ii = 0; ii < N; ii += 32)
    for (int jj = 0; jj < N; jj += 32)
        for (int i = ii; i < ii + 32; ++i)
            for (int j = jj; j < jj + 32; ++j)
                A[i] += B[j] * C[i * N + j];
```

exercise: which has better temporal locality in A? in B? in C?
how about spatial locality?

an OOO pipeline



reorder buffer: on rename

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

reorder buffer: on rename

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

reorder buffer contains instructions started, but not fully finished new entries created on rename (not enough space? stall rename stage)

reorder buffer: on rename

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

reorder buffer (ROB)

remove here when committed →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

place newly started instruction at end of buffer
remember at least its destination register
(both architectural and physical versions)

reorder buffer: on rename

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x23
…
…

reorder buffer (ROB)

remove here when committed →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

next renamed instruction goes in next slot, etc.

reorder buffer: on rename

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x23
…
…

reorder buffer (ROB)

remove here when committed →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

reorder buffer: on commit

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

reorder buffer: on commit

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

instructions marked done in reorder buffer
when result is computed
but not removed from reorder buffer ('committed') yet

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23
%rdx	%x21
…	…

remove here when committed

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

commit stage tracks architectural to physical register map for committed instructions

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ %x24
%rdx	%x21
…	…

remove here
when committed

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer: on commit

phys → arch. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

phys → arch. reg
for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ %x24
%rdx	%x21
…	…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done
update this register map and free register list
and remove instr. from reorder buffer

reorder buffer: commit mispredict (one way)

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
…	…

phys → arch. reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
~~15~~	~~0x1239~~	~~%rax / %x30~~	✓	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	✓	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	✓	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	✓	
~~19~~	~~0x1249~~	~~%rax / %x38~~	✓	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

reorder buffer: commit mispredict (one way)

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
…	…

phys → arch. reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
~~15~~	~~0x1239~~	~~%rax / %x30~~	✓	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	✓	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	✓	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	✓	
~~19~~	~~0x1249~~	~~%rax / %x38~~	✓	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

when committing a mispredicted instruction…
this is where we undo mispredicted instructions

reorder buffer: commit mispredict (one way)

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

phys → arch. reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
~~15~~	~~0x1239~~	~~%rax / %x30~~	✓	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	✓	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	✓	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	✓	
~~19~~	~~0x1249~~	~~%rax / %x38~~	✓	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

copy commit register map into rename register map so we can start fetching from the correct PC

reorder buffer: commit mispredict (one way)

phys → arch. reg for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

phys → arch. reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
~~15~~	~~0x1239~~	~~%rax / %x30~~	✓	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	✓	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	✓	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	✓	
~~19~~	~~0x1249~~	~~%rax / %x38~~	✓	
20	0x1254	PC	✓	✓
~~21~~	~~0x1260~~	~~%rcx / %x17~~		
…	…	…	…	…
~~31~~	~~0x129f~~	~~%rax / %x12~~	✓	
~~32~~	~~0x1230~~	~~%rdx / %x19~~		

…and discard all the mispredicted instructions (without committing them)

better? alternatives

can take snapshots of register map on each branch

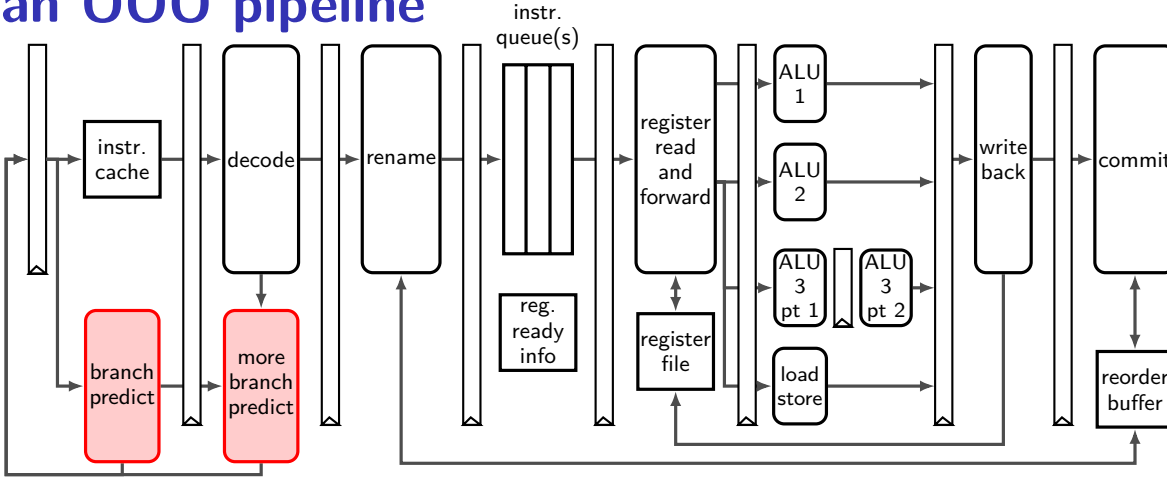
- don't need to reconstruct the table
- (but how to efficiently store them)

can reconstruct register map before we commit the branch instruction

- need to let reorder buffer be accessed even more?

can track more/different information in reorder buffer

an OOO pipeline



branch target buffer

can take several cycles to fetch+decode jumps, calls, returns

still want 1-cycle prediction of next thing to fetch

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)	valid	···
0x00	1	0x400	5	Jxx	0x3FFFF3	···	1	···
0x01	1	0x401	C	JMP	0x401035	---	0	···
0x02	0	---	---	---	---	---	0	···
0x03	1	0x400	9	RET	---	···	0	···
···	···	···	···	···	···	···	···	···
0xFF	1	0x3FF	8	CALL	0x404033	···	0	···

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branches

idx	valid	tag	ofst	type	target	(more info?)	valid	···
0x00	1	0x400	5	Jxx	0x3FFFF3	···	1	···
0x01	1	0x401	C	JMP	0x401035	---	0	···
0x02	0	---	---	---	---	---	0	···
0x03	1	0x400	9	RET	---	···	0	···
···	···	···	···	···	···	···	···	···
0xFF	1	0x3FF	8	CALL	0x404033	···	0	···

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

aside on branch pred. and performance

modern branch predictors are very good

- we might explore how later in semester (if time)

...usually can assume most branches will be predicted

but could be a problem if really no pattern

- e.g. branch based on random number?

generally: measure and see

if branch prediction is bad...

avoiding branches — conditional move, etc.

replace multiple branches with single lookup?

- one misprediction better than K?

instruction queue and dispatch

instruction queue

#	*instruction*
1	`mrmovq (%x04) → %x06`
2	`mrmovq (%x05) → %x07`
3	`addq %x01, %x02 → %x08`
4	`addq %x01, %x06 → %x09`
5	`addq %x01, %x07 → %x10`
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	
%x07	
%x08	
%x09	
%x10	
…	…

execution unit	*cycle#* 1	2	3	4	5	6	7	…
ALU								
data cache (stage 1)								
data cache (stage 2)								
data cache (stage 3)								

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)

Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

but some can handle more/different types of operations than others

2 load functional units

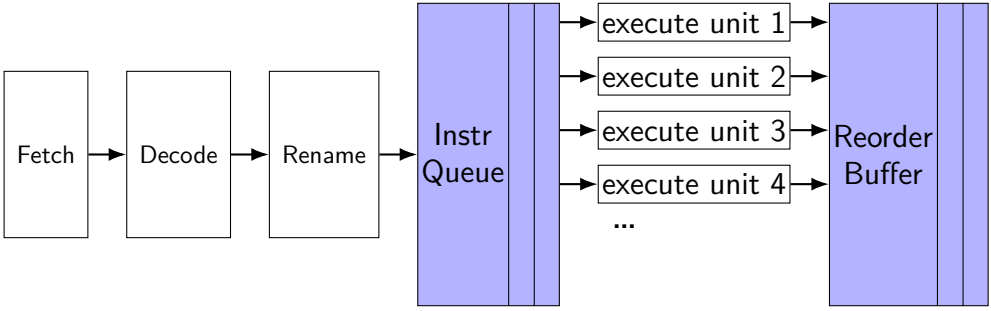
but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

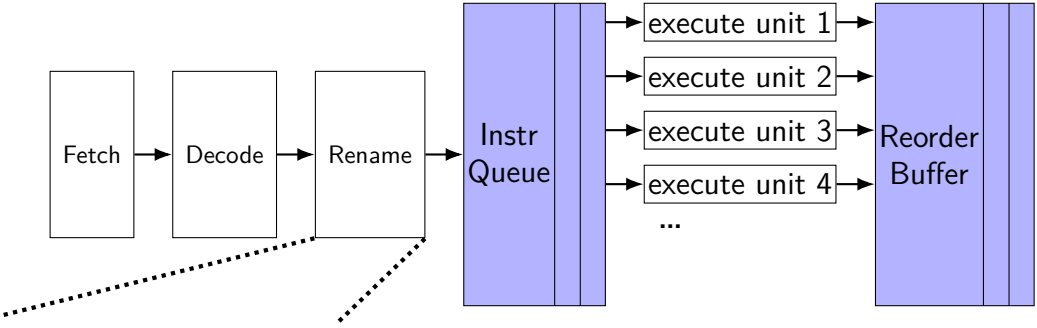
224-entry reorder buffer

determines how far ahead branch mispredictions, etc. can happen

exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



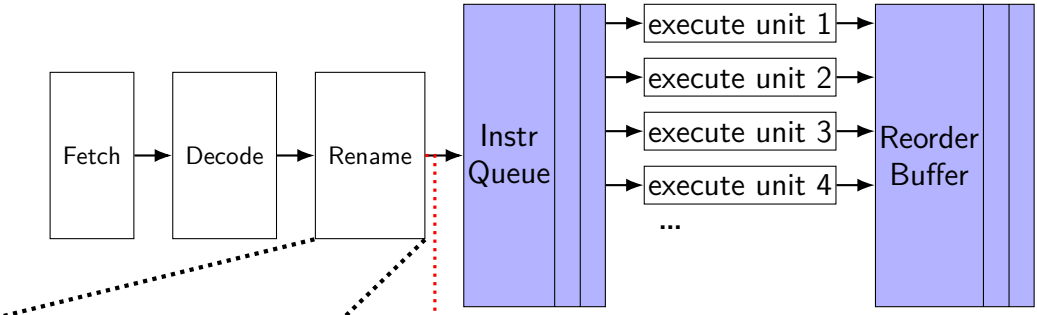
free regs

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

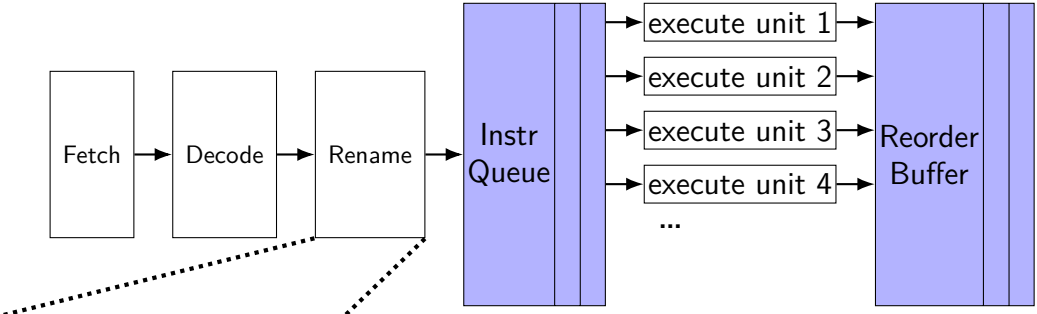
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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

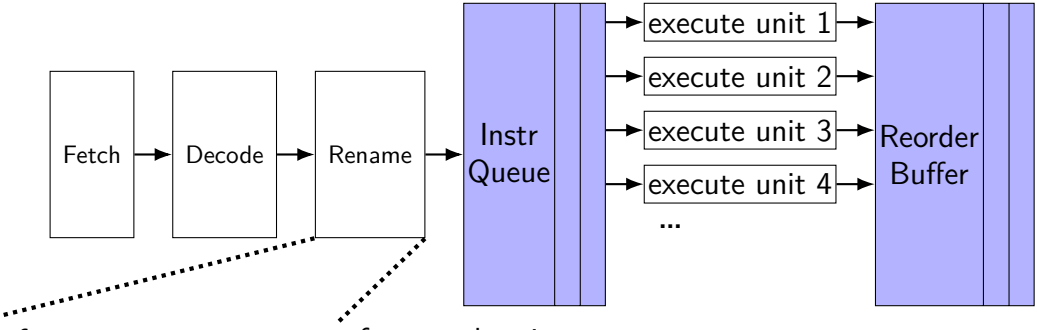
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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

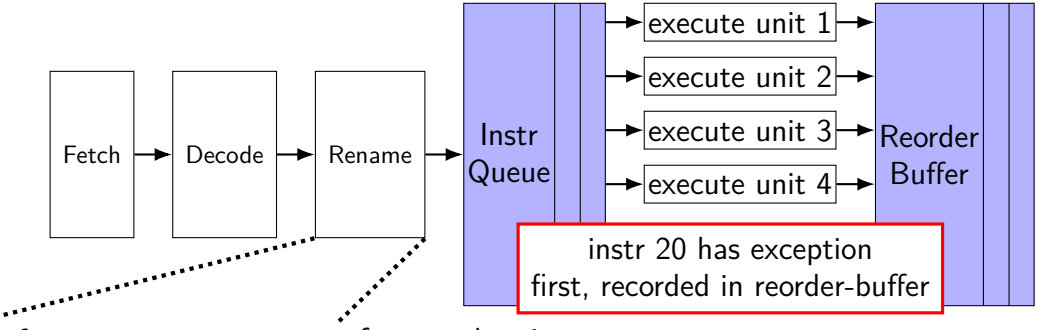
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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

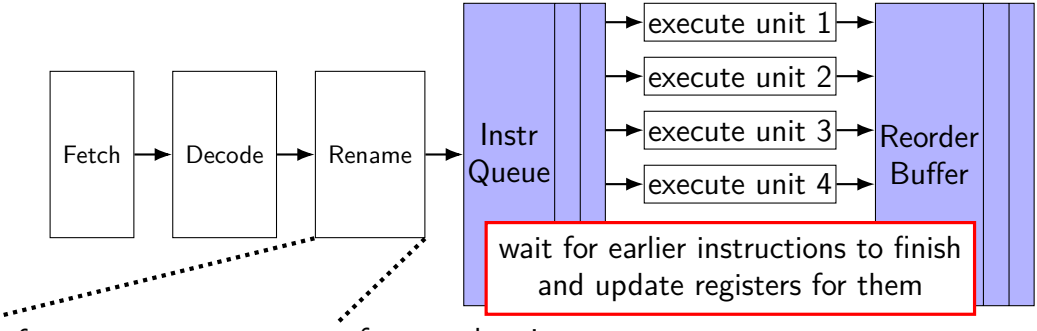
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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

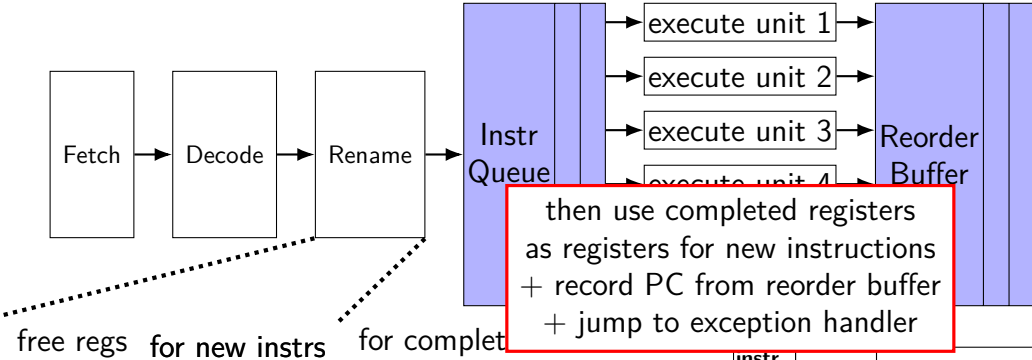
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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)



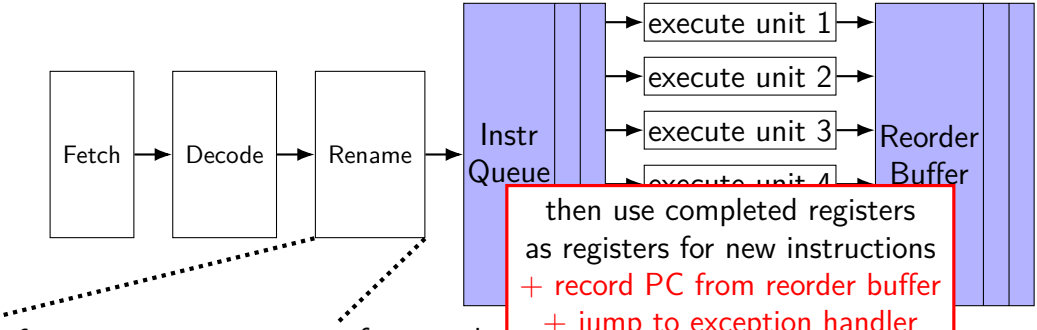
T19
T23
...

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

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)



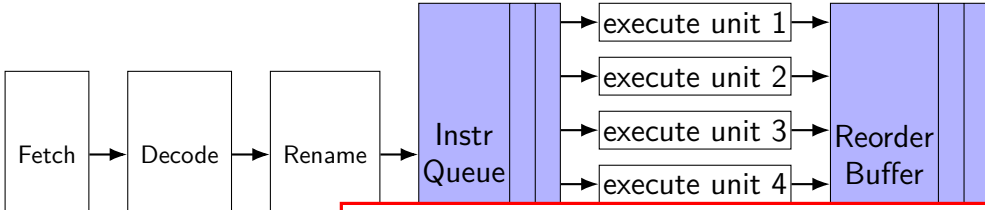
T19
T23
...

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

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

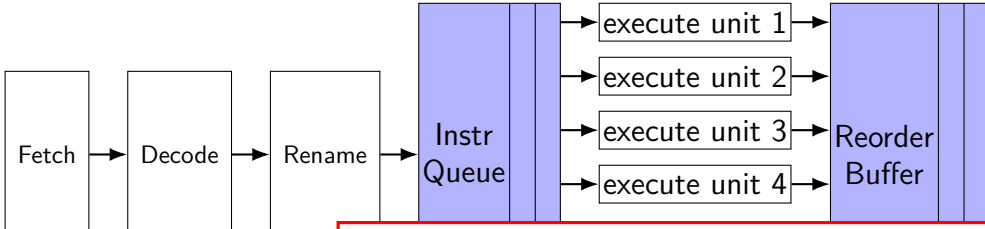
arch. reg	phys. reg
RAX	T15
RCX	T17
RBX	T13
RBX	T07
...	...

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