optimization 1

Changelog

Changes made in this version not seen in first lecture:

- 24 October: fixed locality exercise version 3 to start innermost loop at jj, not 0
- 24 October: instruction queue and dispatch (multicycle): correct some incorrect entries in scoreboard; adjust animation of cycle schedule
- 28 October: register renaming example (2): correct %x13 (old version of r8) that should have been %x18 (new version of r8)
- 29 October: register renaming example (2): correct %x04 (rax) that should have been %x18 (new version of r8)

on last time

we had problems putting together a good lecture on cache performance optimization yesterday

please the lecture video that goes over loop ordering more completely + more cautiously, and which covers which covers **cache blocking**

I'm going to review/introduce that now...— and we'll talk again at next exam review

locality exercise (1)

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

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

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

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?

loop ordering

matrix-like math: multiple options to order loops

- for all i, for all j, for all k: f(i,j,k)
- for all k, for all i, for all j: f(i,j,k)
- ...

analysis trick: look at innermost loop (k and j above)

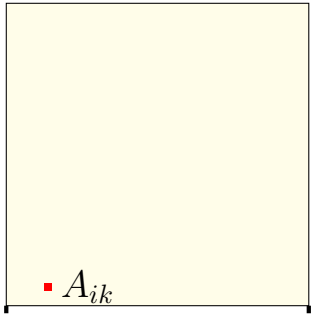
...and spatial locality:

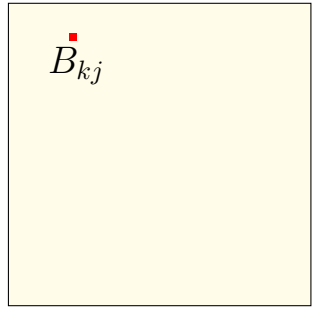
- row-major: iterate through rows: spatial locality
- less cache misses because row accessed together

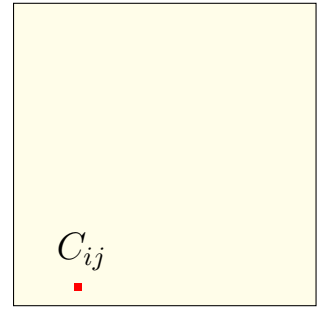
...and temporal locality:

- reusing value in innermost loop

array usage: ijk order



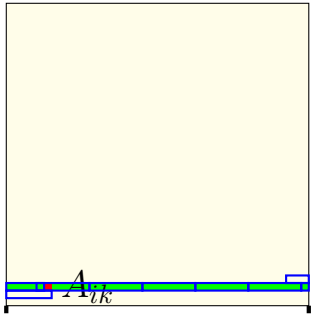


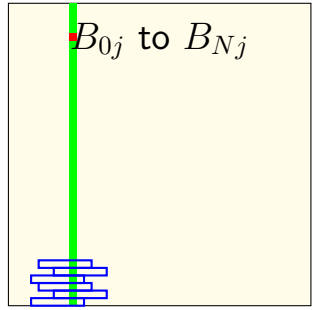


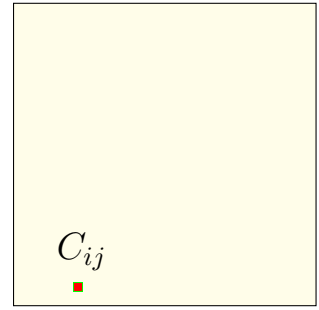
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

if N large:
using C_{ij} many times per load into cache
using A_{ik} once per load-into-cache
(but using $A_{i,k+1}$ right after)
using B_{kj} once per load into cache

array usage: ijk order



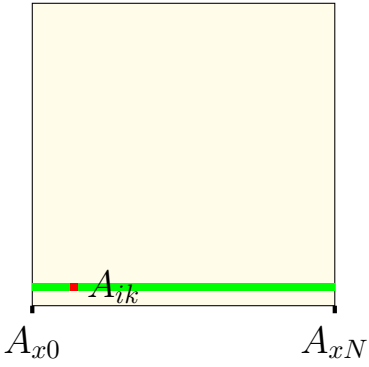


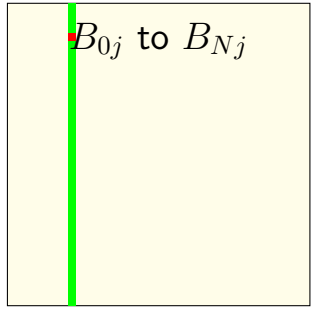


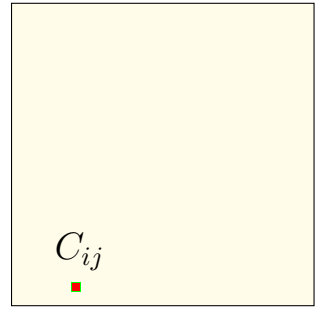
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
good spatial locality in A
(rows stored together = reuse cache blocks)
bad spatial locality in B
(use each cache block once)
no useful spatial locality in C

array usage: ijk order



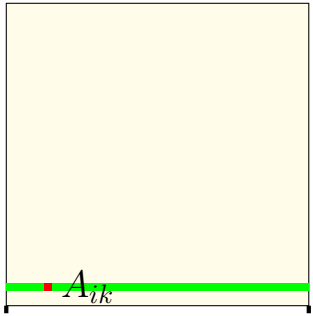


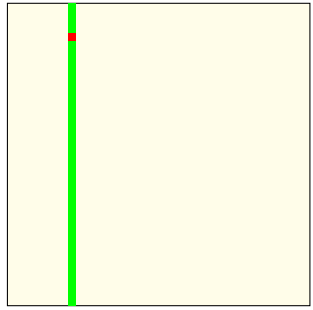


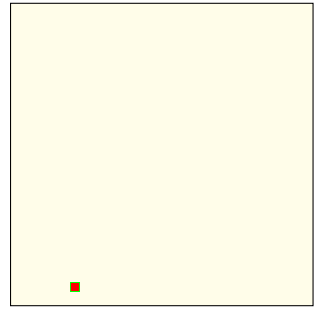
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
temporal locality in C
bad temporal locality in everything else
(everything accessed exactly once)

array usage: ijk order



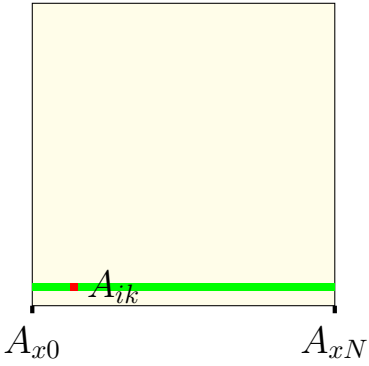


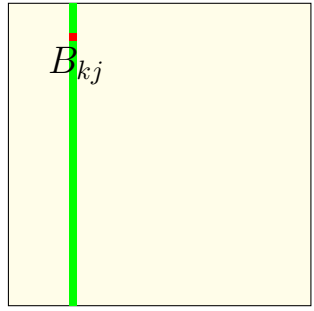


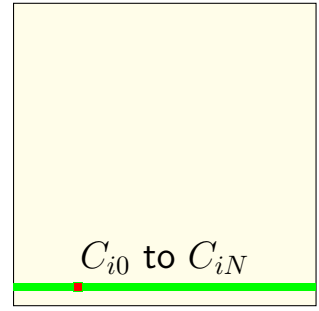
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
row of A (elements used once)
column of B (elements used once)
single element of C (used many times)

array usage: ijk order



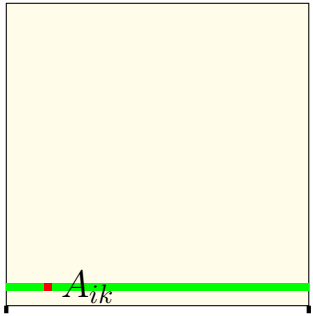


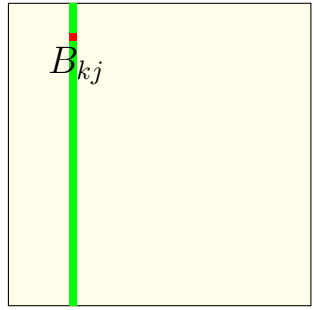


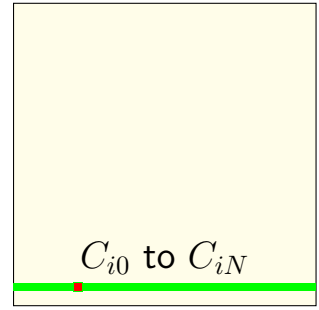
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at two innermost loops together:
some temporal locality in A (column reused)
some temporal locality in B (row reused)
some temporal locality in C (row reused)

array usage: ijk order



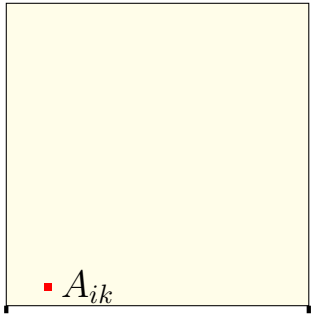


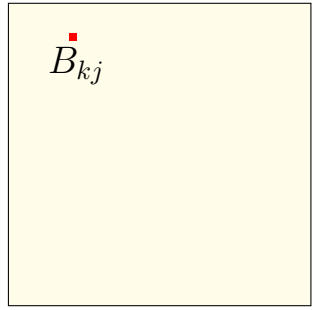


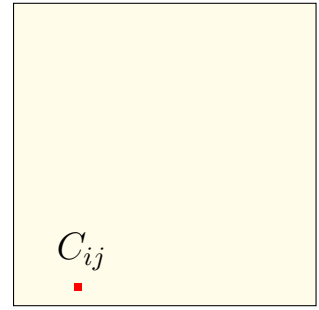
for all i:
  for all j:
    for all k:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at two innermost loops together:
good spatial locality in A
poor spatial locality in B
good spatial locality in C

array usage: kij order



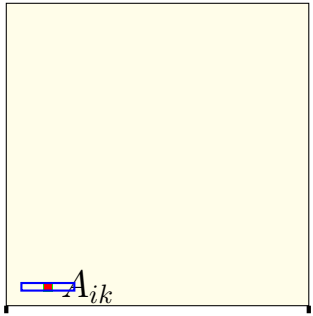


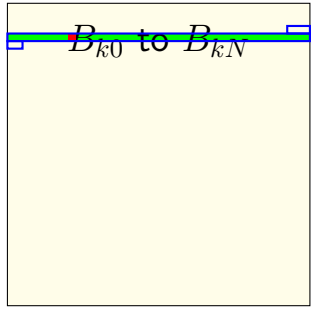


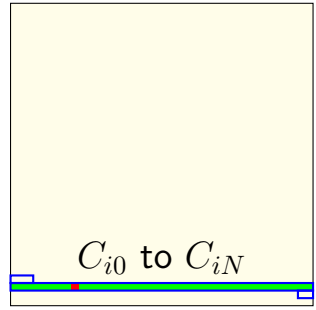
for all k:
  for all i:
    for all j:
      $C_{ij} += A_{ik} \times B_{kj}$

if N large:
using C_{ij} once per load into cache
(but using $C_{i,j+1}$ right after)
using A_{ik} many times per load-into-cache
using B_{kj} once per load into cache
(but using $B_{k,j+1}$ right after)

array usage: kij order



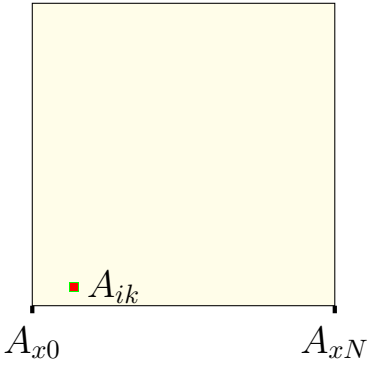


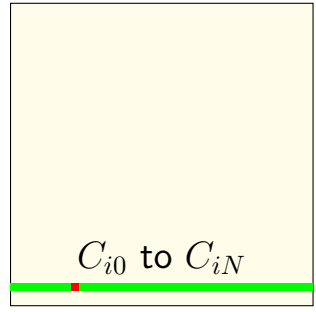


for all k:
  for all i:
    for all j:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
spatial locality in B, C
(use most of loaded B, C cache blocks)
no useful spatial locality in A
(rest of A's cache block wasted)

array usage: kij order

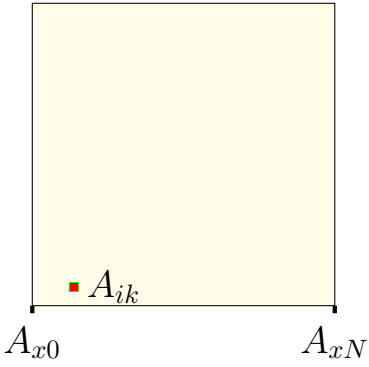


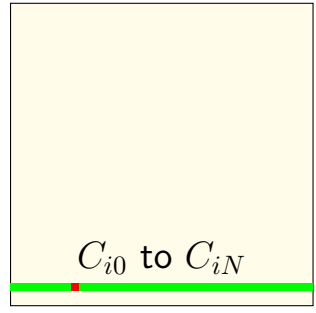


for all k:
  for all i:
    for all j:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
temporal locality in A
no temporal locality in B, C
(B, C values used exactly once)

array usage: kij order

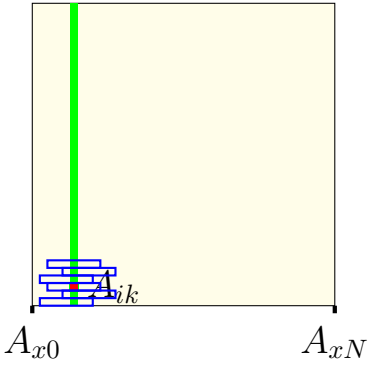


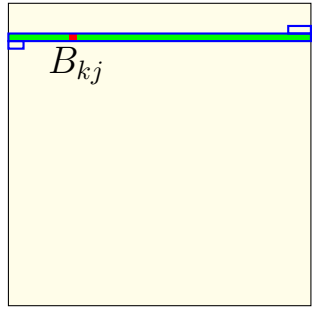


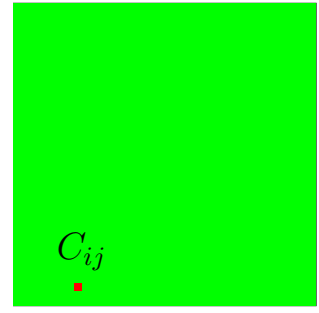
for all k:
  for all i:
    for all j:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at innermost loop:
processing one element of A (use many times)
row of B (each element used once)
column of C (each element used once)

array usage: kij order







for all k:
  for all i:
    for all j:
      $C_{ij} += A_{ik} \times B_{kj}$

looking only at two innermost loops together:
good temporal locality in A (column reused)
good temporal locality in B (row reused)
bad temporal locality in C (nothing reused)

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

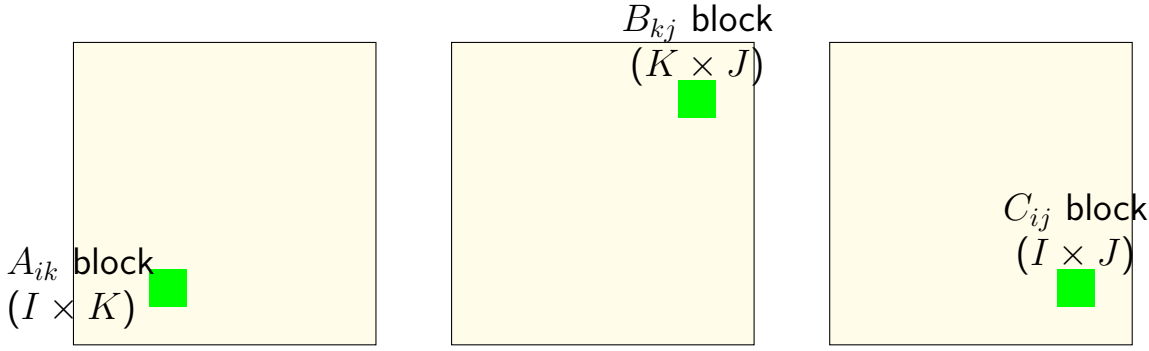
- have some temporal, spatial locality in everything

view 2: 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: block

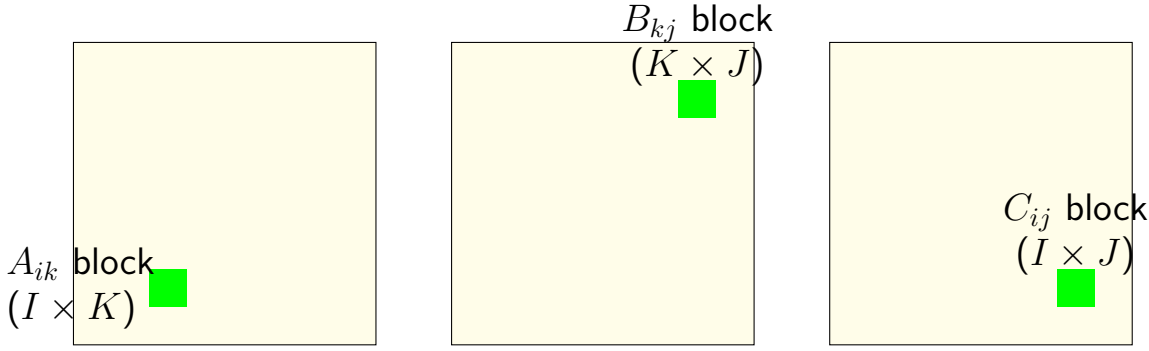
$$\mathbf{C_{ij}} \mathbf{+=} \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$$



inner loops work on "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: block

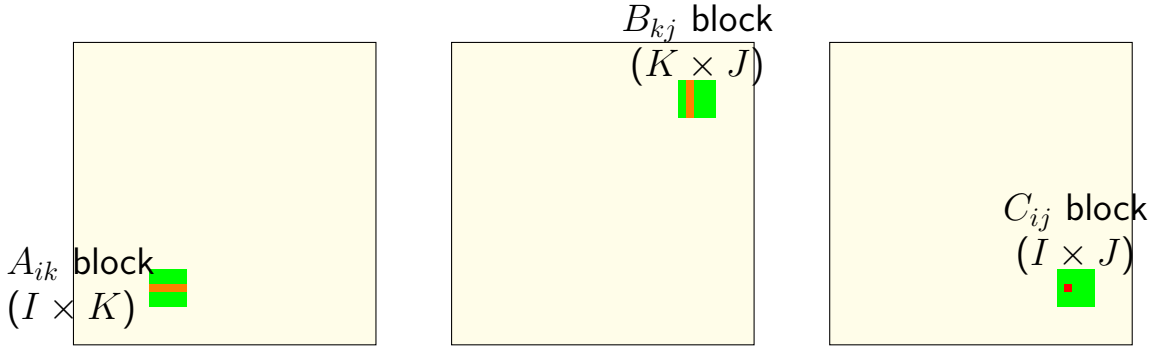
$$\mathbf{C_{ij}} \mathbf{+=} \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$$



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

array usage: block

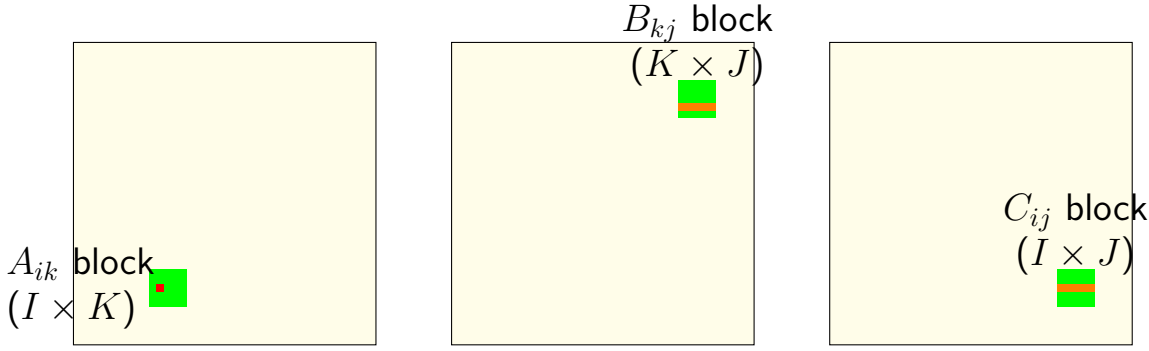
$$\mathbf{C_{ij}} \mathbf{+=} \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$$



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

array usage: block

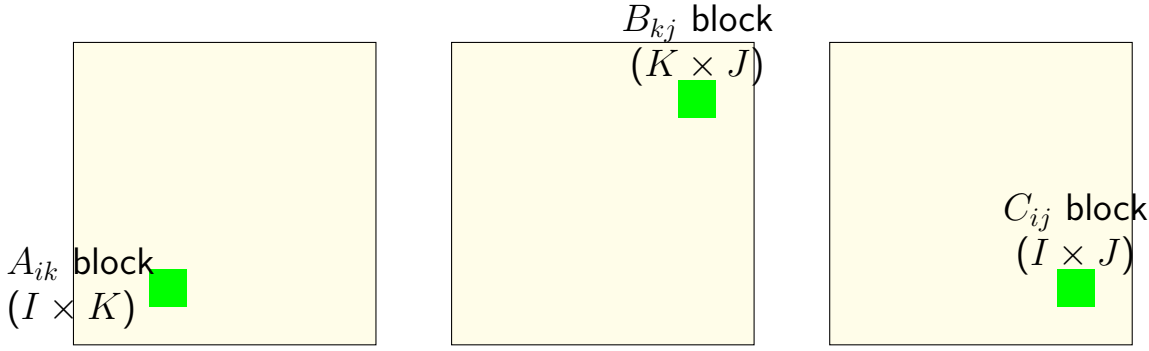
$\mathbf{C_{ij}} \mathbf{+=} \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$



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

array usage: block

$$\mathbf{C_{ij}} \mathrel{+}= \mathbf{A_{ik}} \cdot \mathbf{B_{kj}}$$



(approx.) KIJ fully cached calculations
for $KI + IJ + KJ$ loads
(assuming everything stays in cache)

cache blocking efficiency

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

do $> J$ multiplies with each

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

do I multiplies with each

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

do K adds with each

bigger blocks — more work per load!

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

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

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
        xorl    %eax, %eax
loop:
        cmpq    %edx, %esi
        jle     endOfLoop
        addq    (%rsi,%rdx,8), %rax
        incq    %rdx
        jmp     loop
endOfLoop:
        ret
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp     loop
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

loop unrolling (ASM)

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        incq    %rdx
        jmp     loop
endOfLoop:
```

```
loop:
        cmpl    %edx, %esi
        jle     endOfLoop
        addq    (%rdi,%rdx,8), %rax
        addq    8(%rdi,%rdx,8), %rax
        addq    $2, %rdx
        jmp     loop
        // plus handle leftover?
endOfLoop:
```

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)

times unrolled	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

instruction cache/etc. overhead

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+0)*N+j];
      C[i*N+j+1] += A[i*N+k+1] * B[(k+1)*N+j+1];
    }
```

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];
      }
```

interlude: real CPUs

modern CPUs:

execute multiple instructions at once

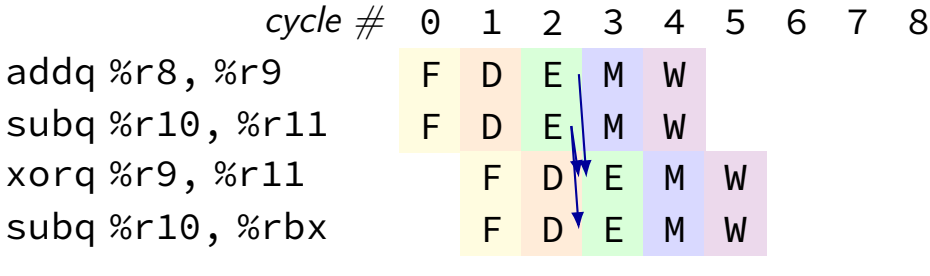
execute instructions out of order — whenever values available

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?

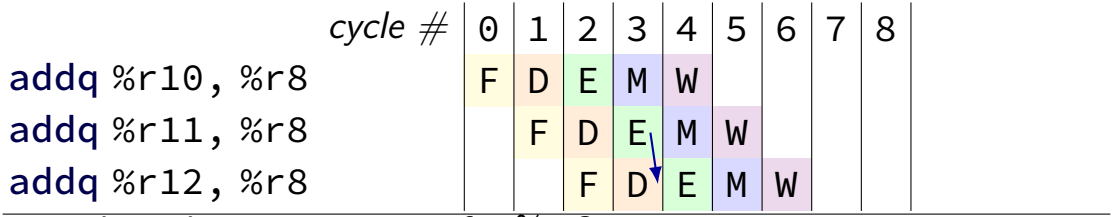
problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

read-after-write examples (1)



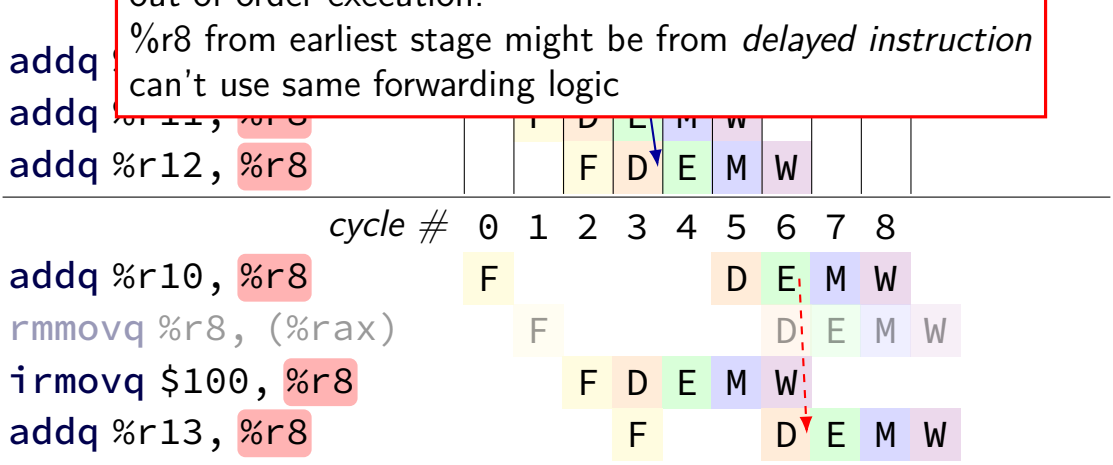
normal pipeline: two options for %r8?

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

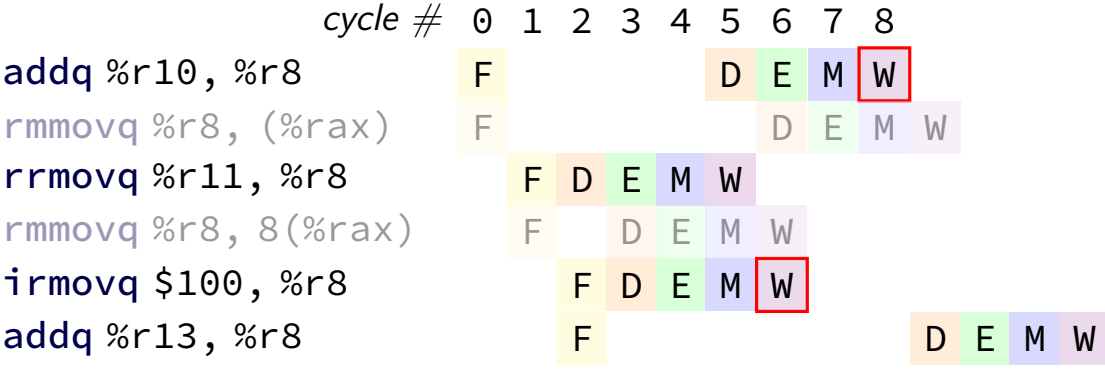
- represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

for now: version numbers

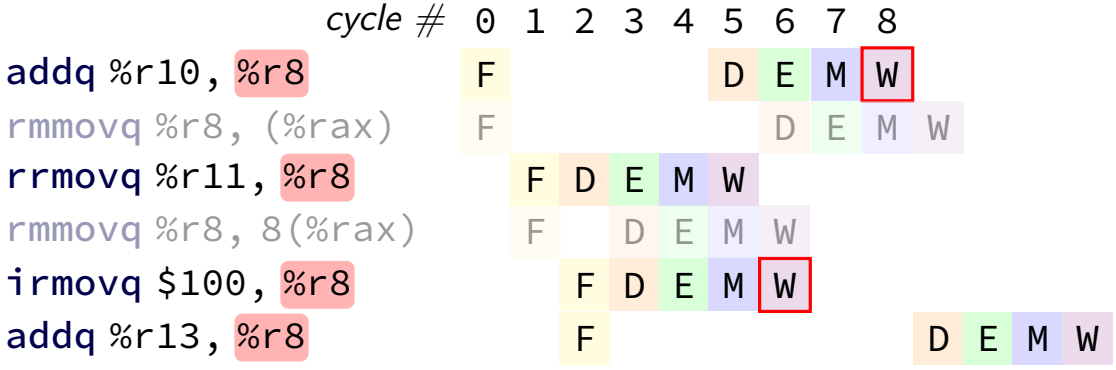
later: something simpler to implement

write-after-write example



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

write-after-write example



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

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

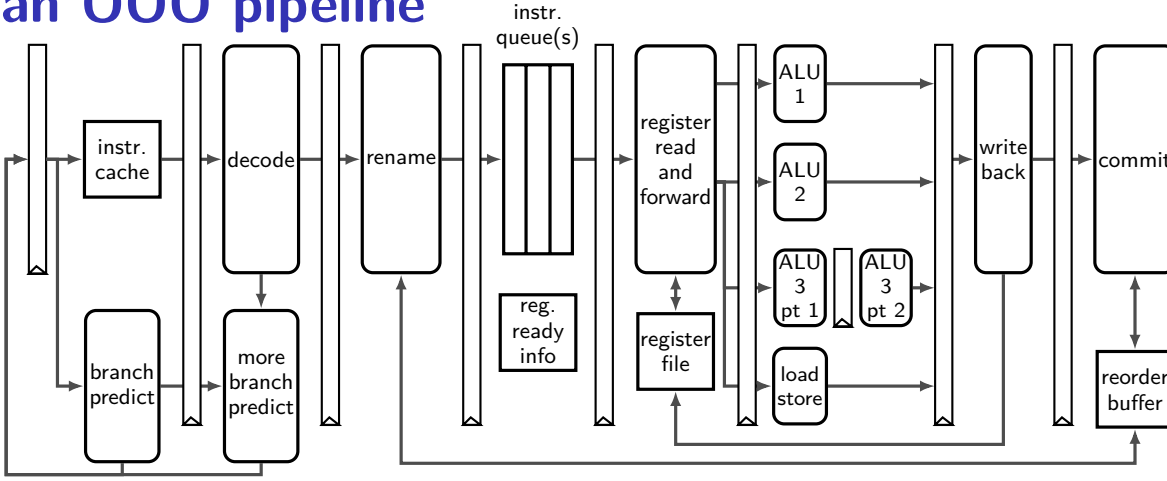
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

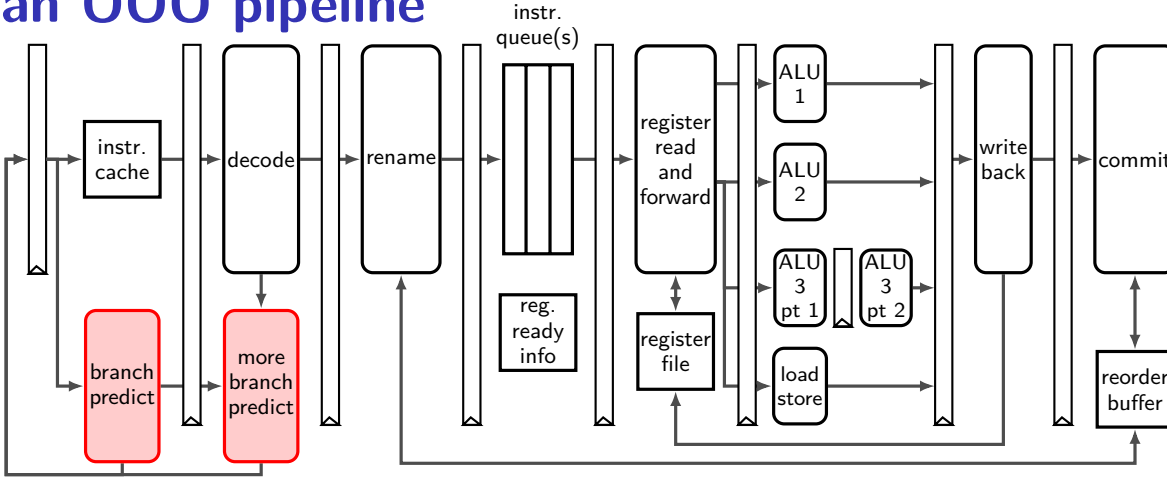
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

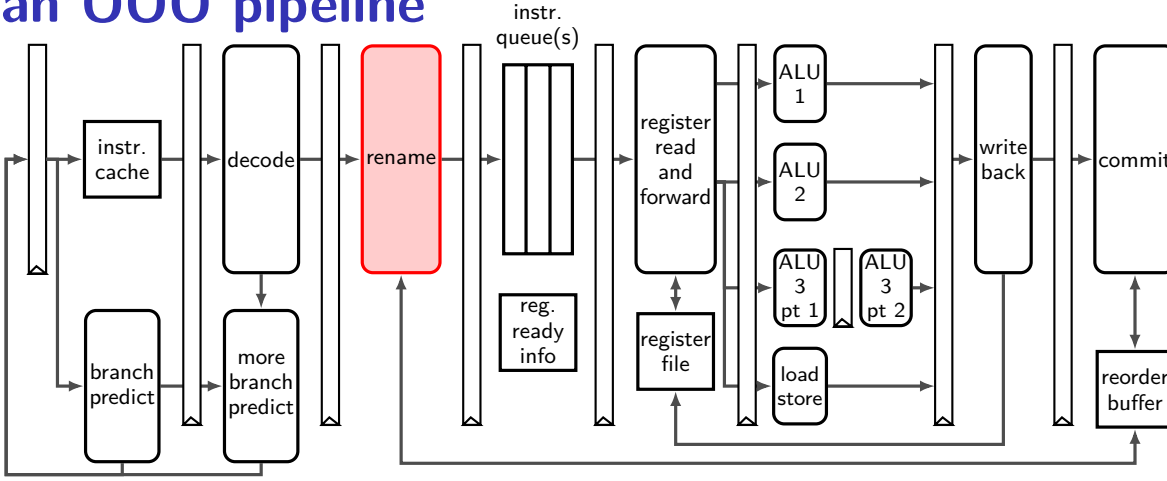


an OOO pipeline



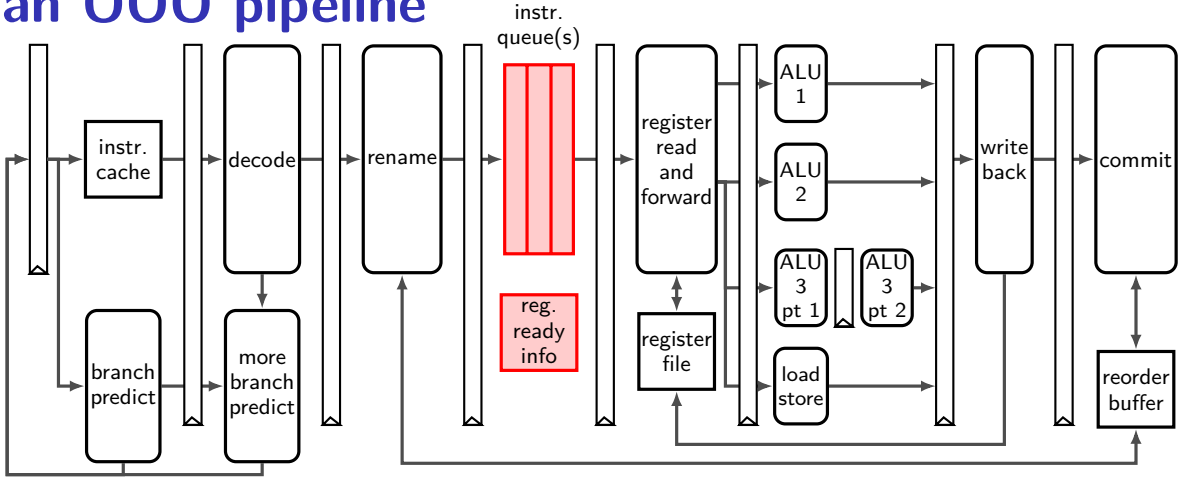
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



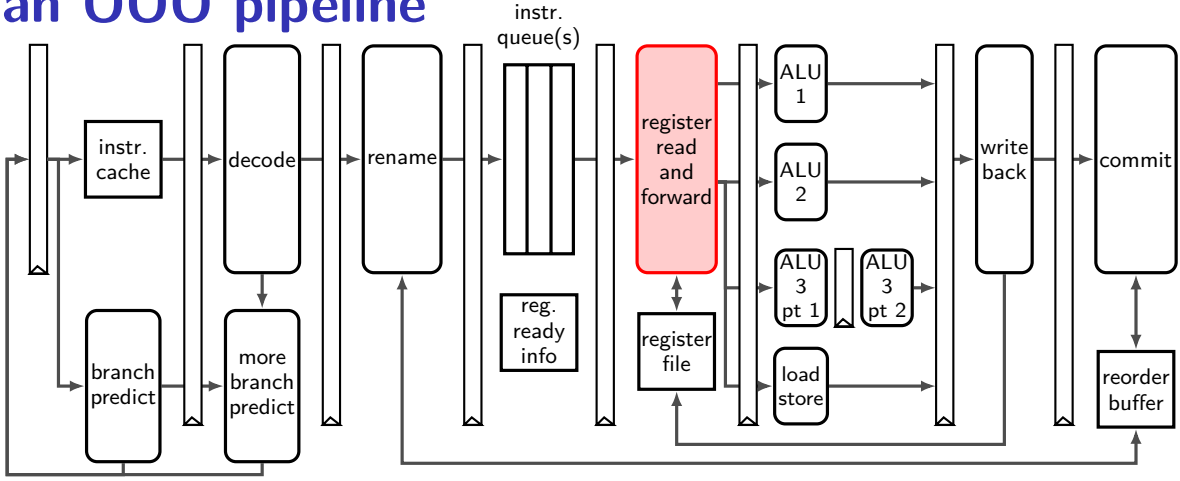
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



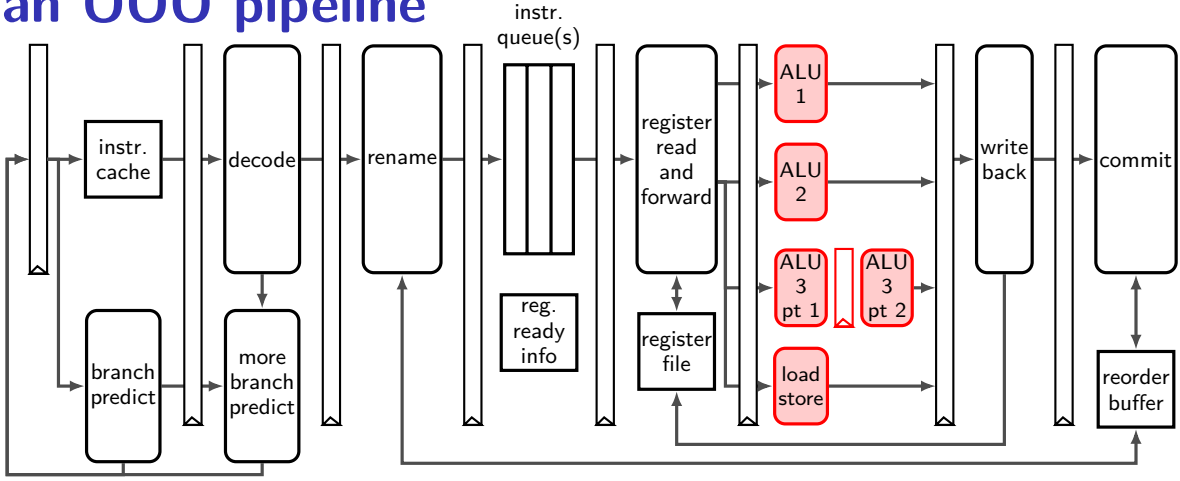
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



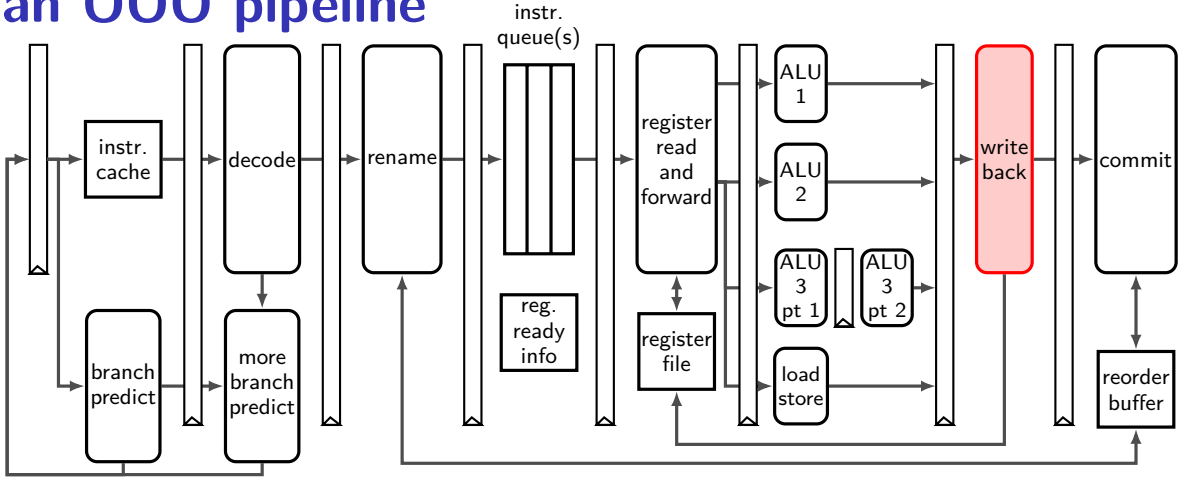
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



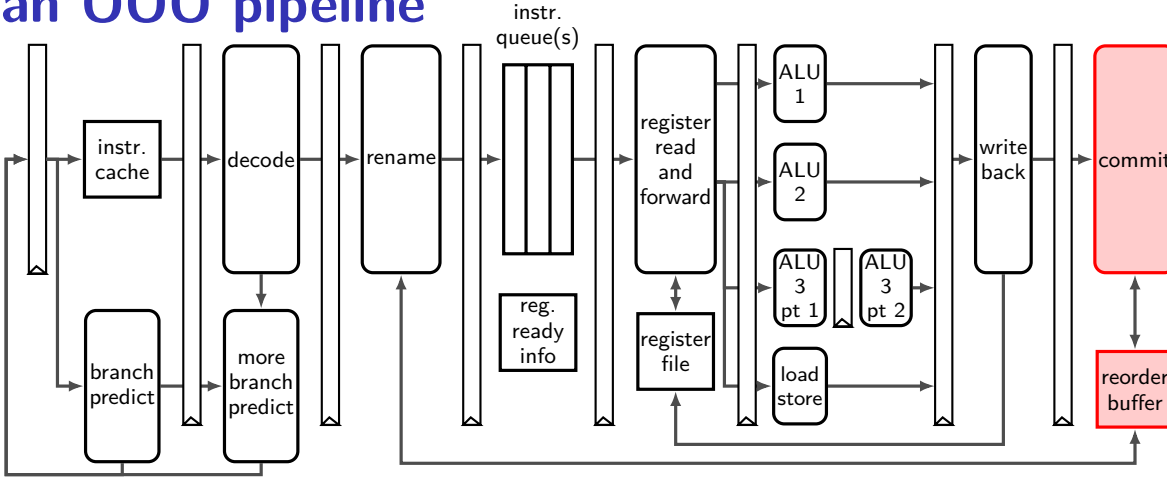
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



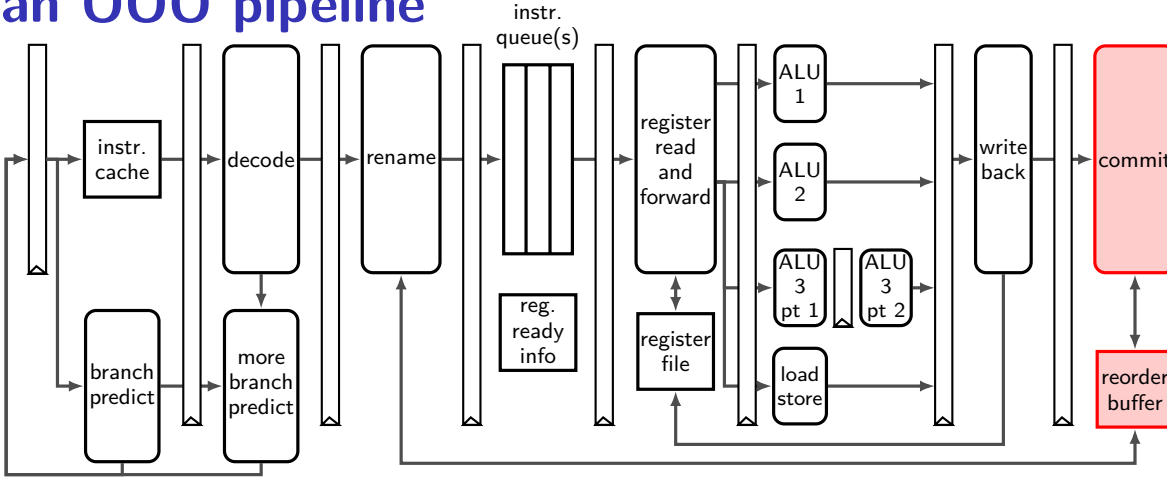
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



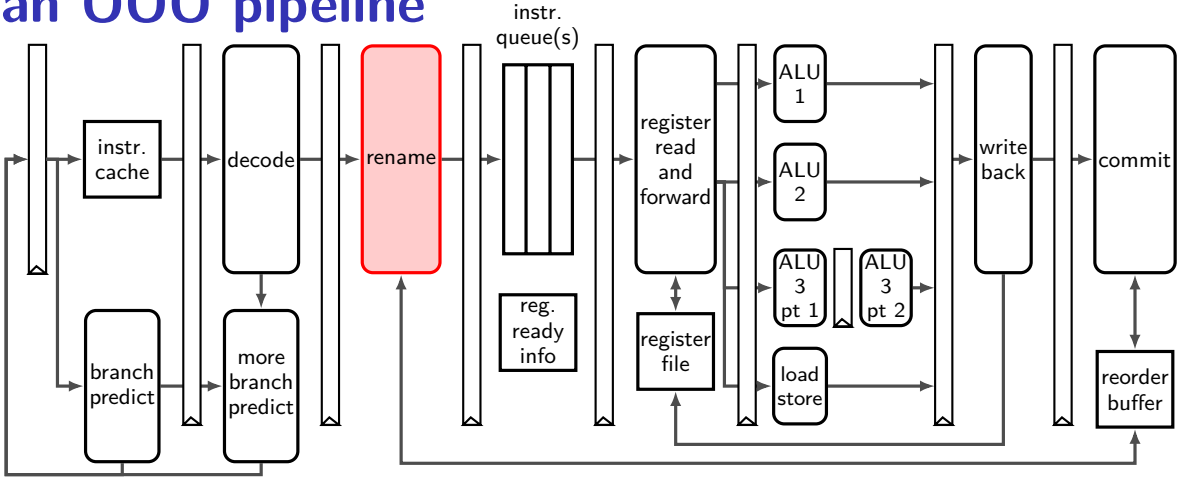
new commit (sometimes *retire*) stage finalizes instruction
figures out when physical registers can be reused again

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 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 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
%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
%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	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`	

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~~
…

renaming with microcode

original

```
pushq %r8
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%rsp	%x11
…	…
%r8	%x13
%r9	%x17
%r10	%x19
…	…

free regs

%x18
%x20
%x21
%x23
%x24
…

renaming with microcode

original

```
pushq %r8
```

renamed

```
iaddq $8, %x11 → %x18
rmmovq %x13, 0(%x18) → (memory)
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%rsp	~~%x11~~%x18
…	…
%r8	%x13
%r9	%x17
%r10	%x19
…	…

pushq is really complicated
one implementation option:
split into simpler "microinstructions"
also exposes %rsp to register renmaing

%x20
%x21
%x23
%x24
…

renaming and condition codes

original

```
cmpq %r8, %r9
jle D
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%rsp	%x18
…	…
%r8	%x13
%r9	%x17
%r10	%x19
…	…
CC	%x04.cc

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

renaming and condition codes

original

```
cmpq %r8, %r9
jle D
```

renamed

```
subq %x13, %x17 → %x18.cc
jle %x18.cc, D
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%rsp	%x18
…	…
%r8	%x13
%r9	%x17
%r10	%x19
…	…
CC	~~%x04.cc~~%x18.cc

%x20
%x21
%x23
%x24
…

one option for condition codes:
map to physical registers and track with renaming
not sure if real CPUs do this option
(complicates the commit stage?
more area for regs than alternative?)
alternative 1: entirely separate cond. code registers (with free register list)
alternative 2: handle in 'in-order' part of pipeline?

renaming and condition codes

original

```
cmpq %r8, %r9
jle D
```

renamed

```
subq %x13, %x17 → %x18.cc
jle %x18.cc, D
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%rsp	%x18
…	…
%r8	%x13
%r9	%x17
%r10	%x19
…	…
CC	~~%x04.cc~~%x18.cc

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

renaming trickiness (1)

need to expose input + outputs

hidden dependencies on stack pointer, condition codes, memory, etc.

stack pointer + condition codes

- turn into visible register somehow
- alternative: force to execute in-order

memory: complex techniques we won't discuss

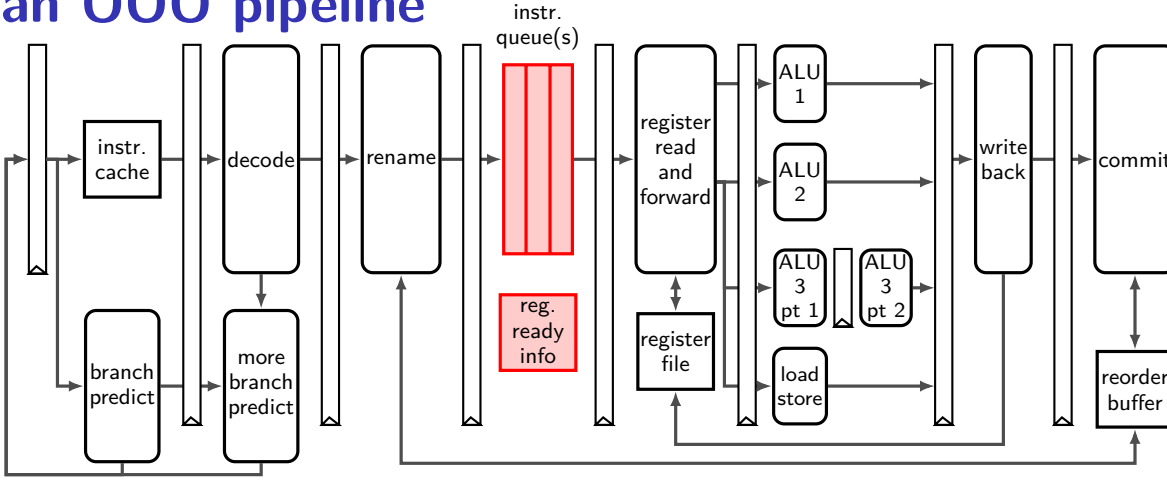
renaming trickiness (2)

opportunity to translate complex instructions into simpler

commonly used for memory operands, probably some stack instructions

- `popq %rcx` → addq, store
- `addq %rax, (%rbx)` → load, addq, store
- ...

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	
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		
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
%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
%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

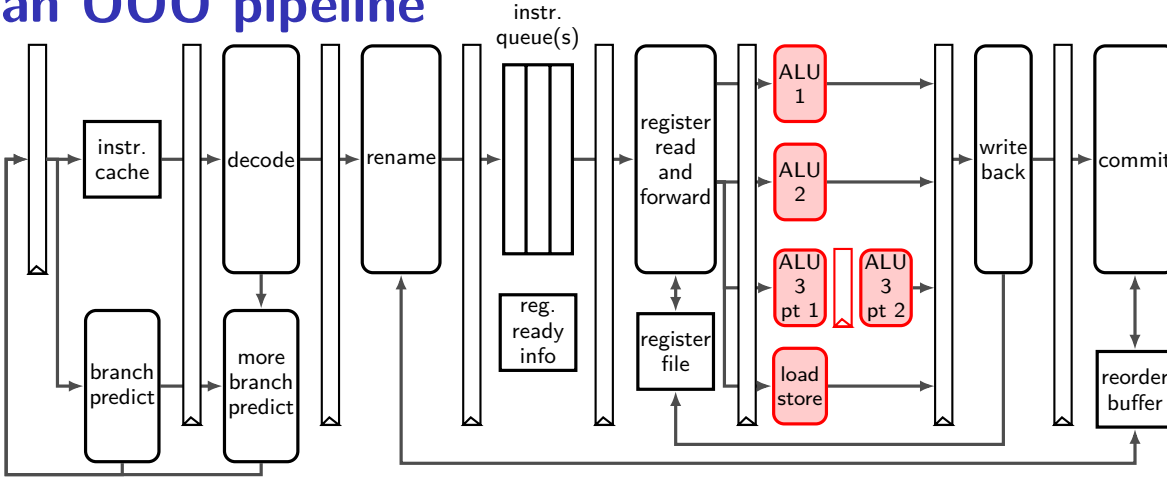
#	*instruction*
1	~~`addq %x01, %x05 → %x06`~~
2	~~`addq %x02, %x06 → %x07`~~
3	~~`addq %x03, %x07 → %x08`~~
4	~~`cmpq %x04, %x08 → %x09.cc`~~
5	~~`jne %x09.cc, ...`~~
6	~~`addq %x01, %x08 → %x10`~~
7	~~`addq %x02, %x09 → %x11`~~
8	~~`addq %x03, %x10 → %x12`~~
9	~~`cmpq %x04, %x11 → %x13.cc`~~
…	…

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	—	…	

an OOO pipeline



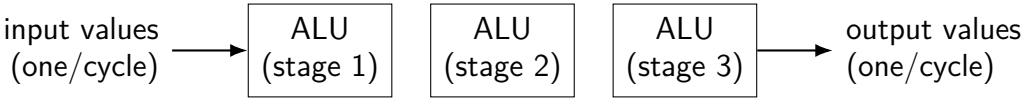
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



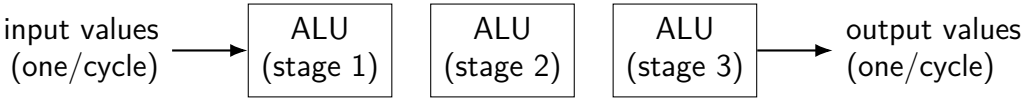
execution units AKA functional units (1)

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 $A \times (B \times (C \times D))$?

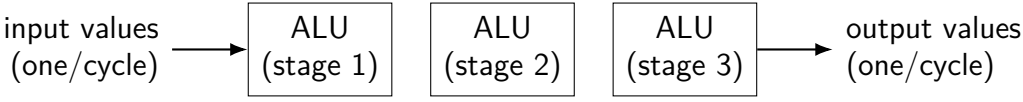
execution units AKA functional units (1)

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

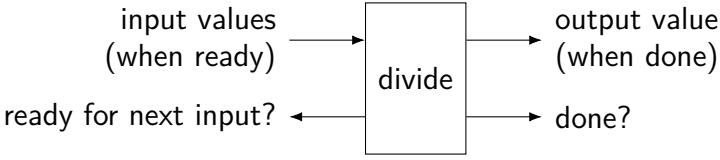
no parallelism!

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

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

branch misprediction has a big cost (relative to pipelined)

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

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

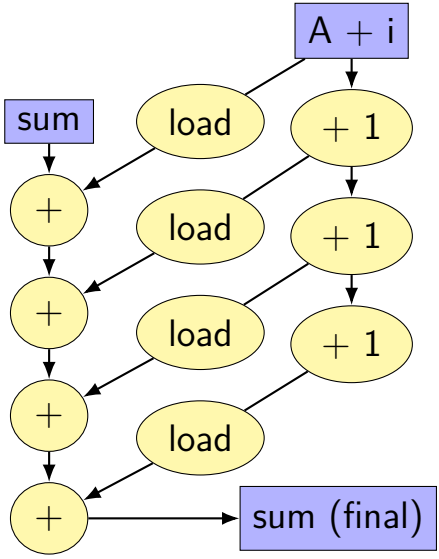
- but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

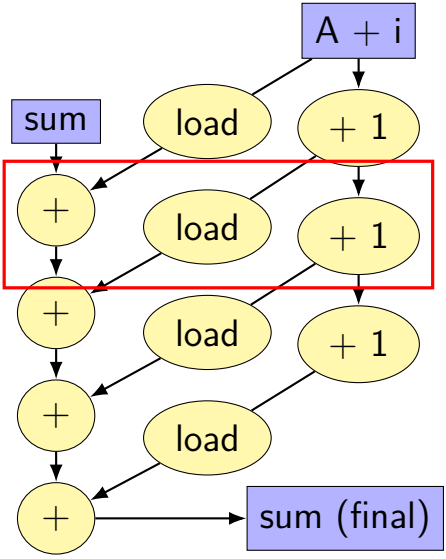
- determines how far ahead branch mispredictions, etc. can happen

data flow model and limits



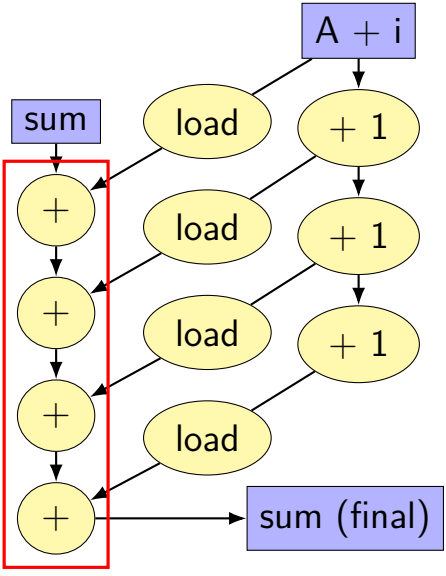

```
for (int i = 0; i < N; i += K) {
    sum += A[i];
    sum += A[i+1];
    ...
}
```

data flow model and limits



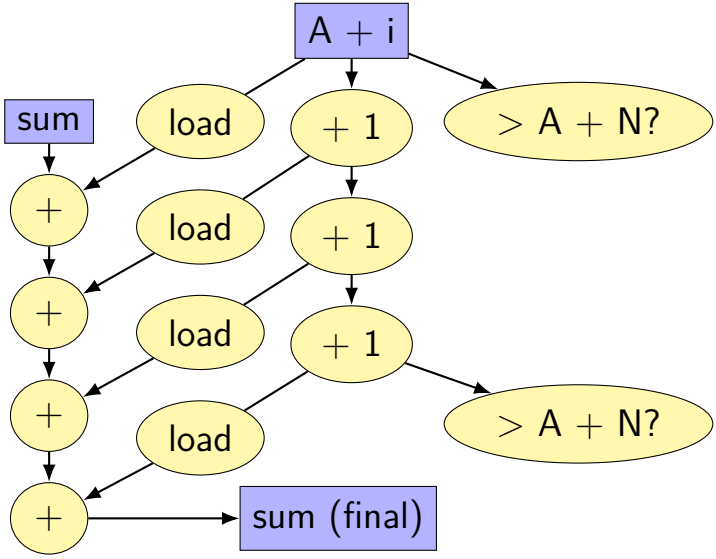
three ops/cycle (if each one cycle)

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
```

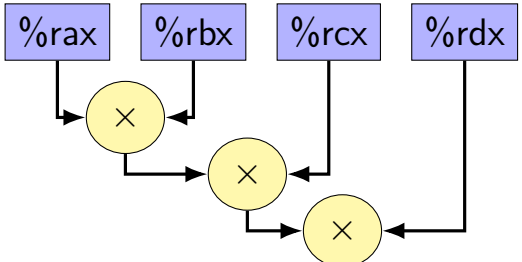
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

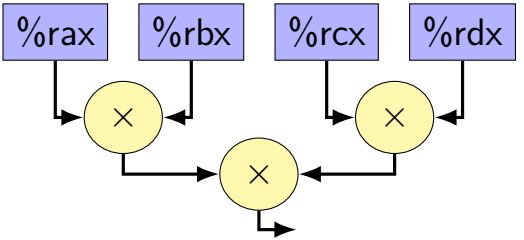
$((a \times b) \times c) \times d$

```
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
```



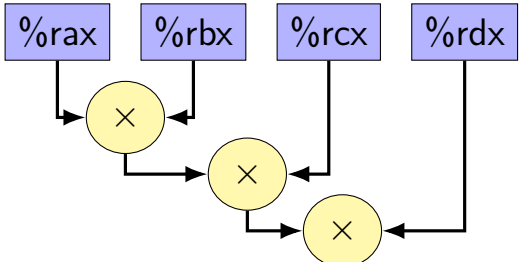
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

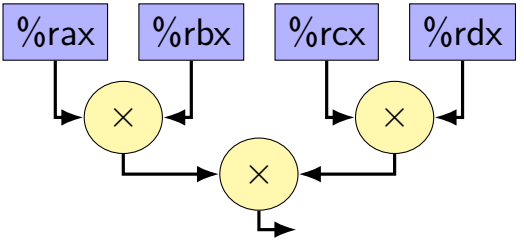
$((a \times b) \times c) \times d$

```
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

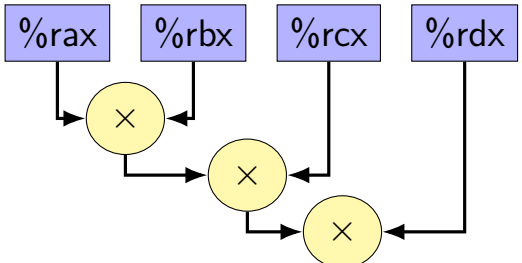
assume a single pipelined, 5-cycle latency multiplier

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
```

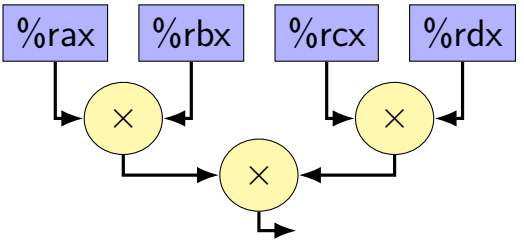
15 cycles



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

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

11 cycles



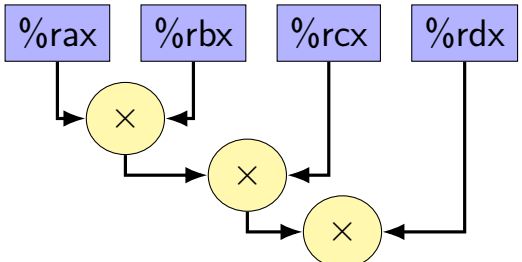
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

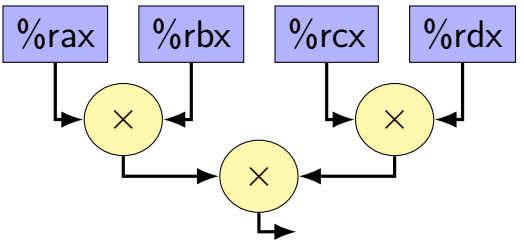
$((a \times b) \times c) \times d$

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

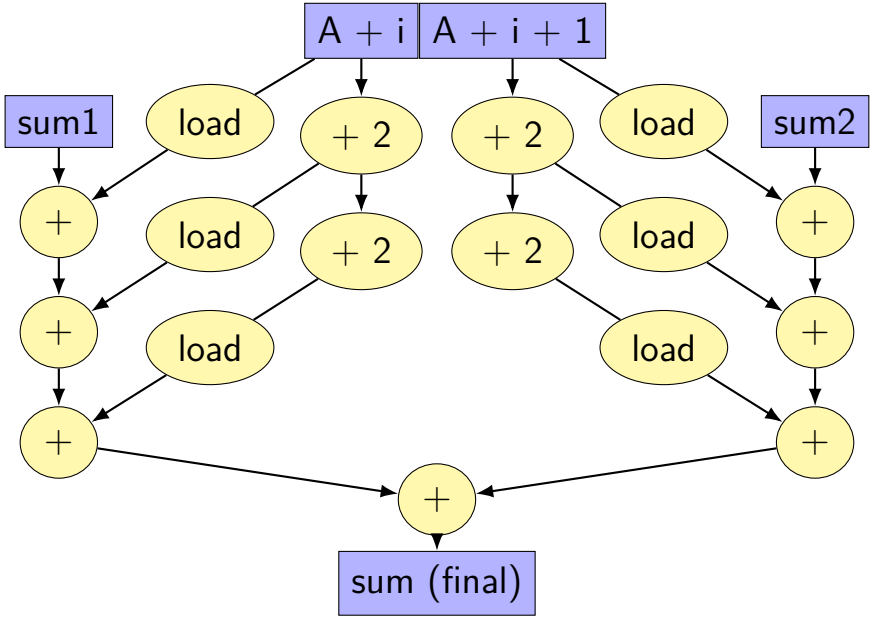


$(a \times b) \times (c \times d)$

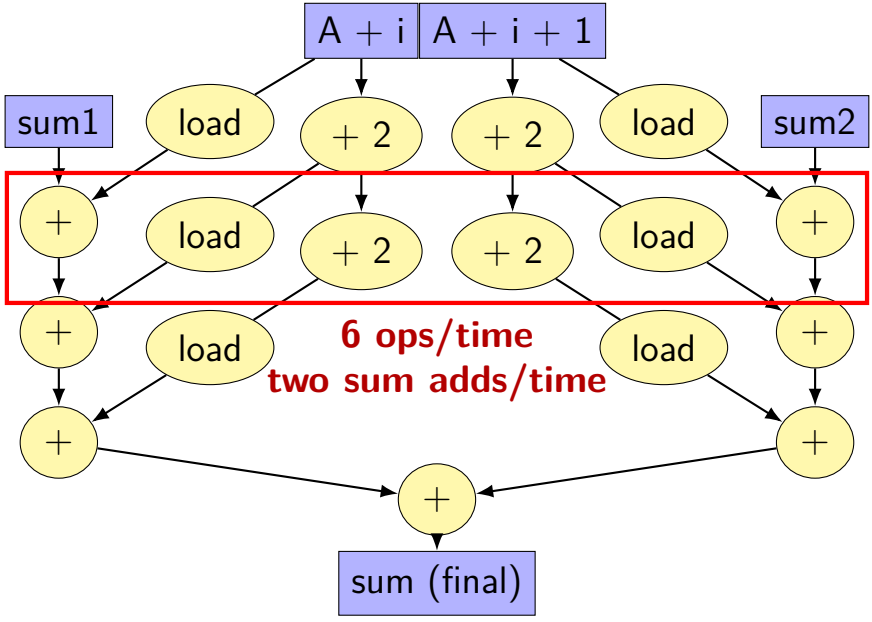
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



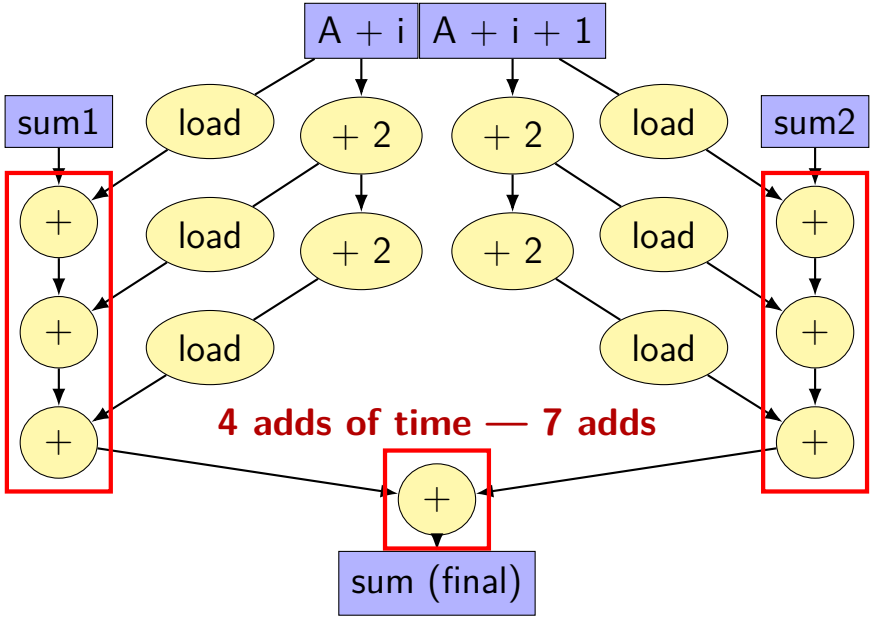
better data-flow



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), %rcx          // 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:

- 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

compiler limitations

needs to generate code that does the same thing...

- ...even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

- needs to assume it might do anything

can't predict what inputs/values will be

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

can't understand code size versus speed tradeoffs

compiler limitations

needs to generate code that does the same thing...

- ...even in corner cases that "obviously don't matter"

often doesn't 'look into' a method

- needs to assume it might do anything

can't predict what inputs/values will be

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

can't understand code size versus speed tradeoffs

aliasing

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
}
```

the compiler **cannot** generate this:

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

aliasing problem

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
    // NOT the same as *px += 2 * *py;
}
...
    long x = 1;
    twiddle(&x, &x);
    // result should be 4, not 3
```

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

non-contrived aliasing

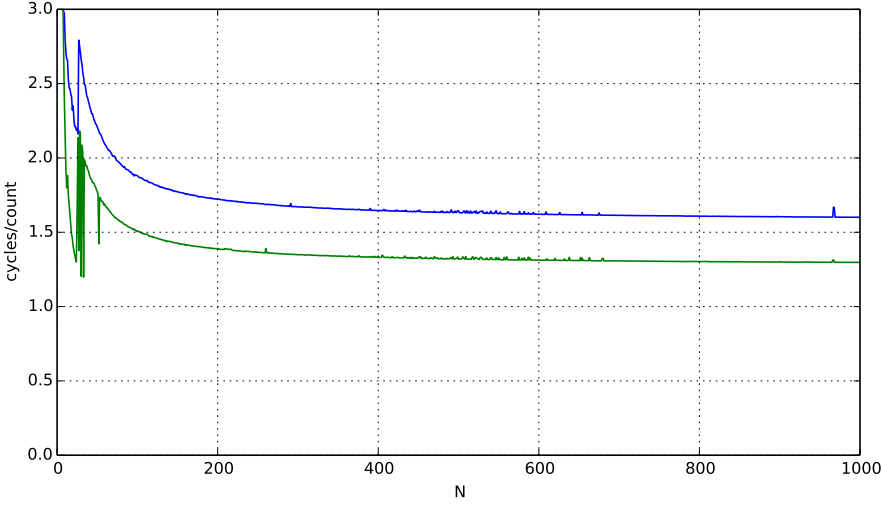
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

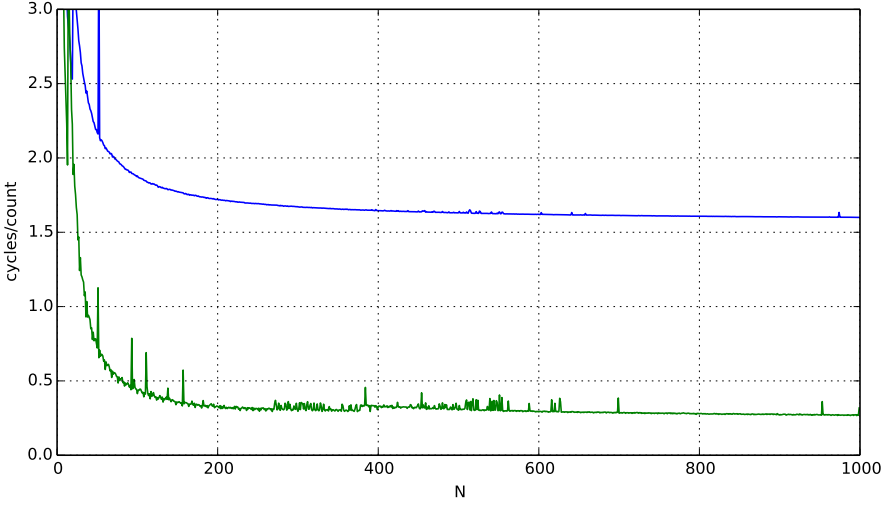
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
void sumRows2(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        int sum = 0;
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
}
```

aliasing and performance (1) / GCC 5.4 -O2



aliasing and performance (2) / GCC 5.4 -O3



automatic register reuse

Compiler would need to generate overlap check:

```
if (result > matrix + N * N || result < matrix) {
    for (int row = 0; row < N; ++row) {
        int sum = 0; /* kept in register */
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
} else {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
        }
    }
}
```

aliasing and cache optimizations

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

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

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

compiler can't generate same code for both

aliasing problems with cache blocking

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

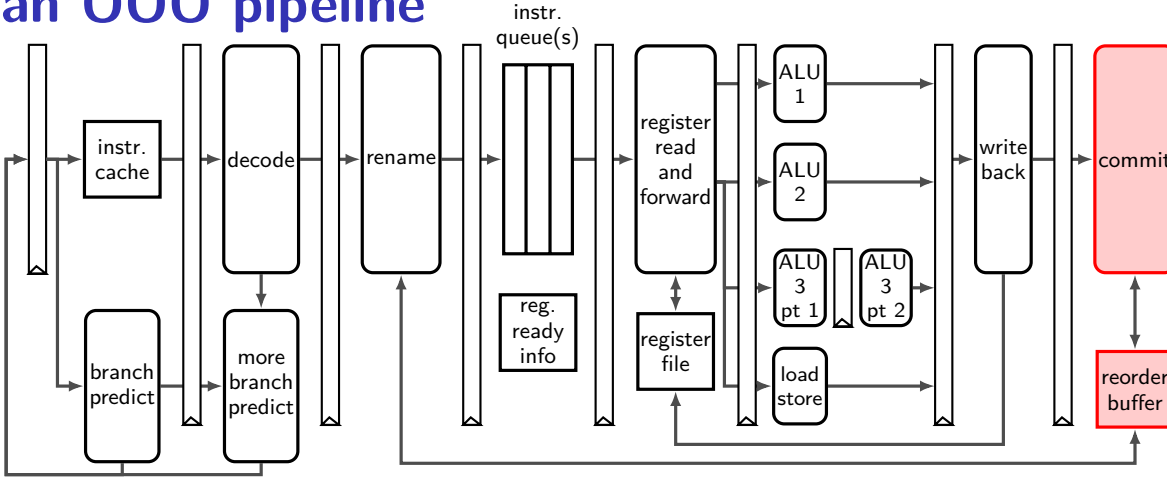
can compiler keep `A[i*N+k]` in a register?

“register blocking”

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; i += 2) {
    float Ai0k = A[(i+0)*N + k];
    float Ai1k = A[(i+1)*N + k];
    for (int j = 0; j < N; j += 2) {
      float Bkj0 = A[k*N + j+0];
      float Bkj1 = A[k*N + j+1];
      C[(i+0)*N + j+0] += Ai0k * Bkj0;
      C[(i+1)*N + j+0] += Ai1k * Bkj0;
      C[(i+0)*N + j+1] += Ai0k * Bkj1;
      C[(i+1)*N + j+1] += Ai1k * Bkj1;
    }
  }
}
```

Backup slides

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?
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?
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?
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?
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?
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?
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?
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?
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?
~~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?
~~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?
~~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?
~~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?
~~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

caching blocking + loop unroll + no aliasing (1)

blocking for k, i (missing j), plus 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) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      for(int kk = k; kk < k + 2; ++kk) {
        float Bkj = B[kk*N+j];
        Ci0j += A[(i+0)*N+kk] * Bkj;
        Ci1j += A[(i+1)*N+kk] * Bkj;
      }
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

caching blocking + loop unroll + no aliasing (1)

blocking for k, i (missing j), plus 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) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      for(int kk = k; kk < k + 2; ++kk) {
        float Bkj = B[kk*N+j];
        Ci0j += A[(i+0)*N+kk] * Bkj;
        Ci1j += A[(i+1)*N+kk] * Bkj;
      }
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

caching blocking + loop unroll + no aliasing (1)

plus explicitly unroll loop over k values

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      float Bk0j = B[(k+0)*N+j];
      float Bk1j = B[(k+1)*N+j];
      Ci0j += A[(i+0)*N+k] * Bk0j;
      Ci0j += A[(i+0)*N+k+1] * Bk1j;
      Ci1j += A[(i+1)*N+k] * Bk0j;
      Ci1j += A[(i+1)*N+k+1] * Bk1j;
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

caching blocking + loop unroll + no aliasing (1)

plus explicitly unroll loop over k values

```
for (int k = 0; k < N; k += 2)
  for (int i = 0; i < N; i += 2)
    for (int j = 0; j < N; ++j) {
      float Ci0j = C[(i+0)*N+j];
      float Ci1j = C[(i+1)*N+j];
      float Bk0j = B[(k+0)*N+j];
      float Bk1j = B[(k+1)*N+j];
      Ci0j += A[(i+0)*N+k] * Bk0j;
      Ci0j += A[(i+0)*N+k+1] * Bk1j;
      Ci1j += A[(i+1)*N+k] * Bk0j;
      Ci1j += A[(i+1)*N+k+1] * Bk1j;
      C[(i+0)*N+j] += Ci0j;
      C[(i+1)*N+j] += Ci1j;
    }
```

performance labs

this week — loop optimizations

after Thanksgiving — vector instructions (AKA SIMD)

performance HWs

INDIVIDUAL ONLY

assignment 1: rotate an image

assignment 2: smooth (blur) an image

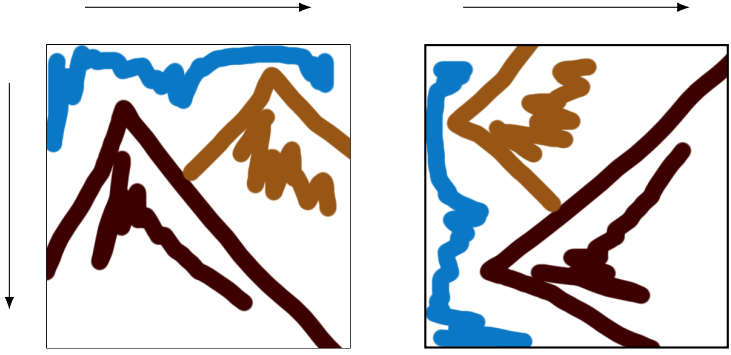
image representation

```
typedef struct {
    unsigned char red, green, blue, alpha;
} pixel;
pixel *image = malloc(dim * dim * sizeof(pixel));

image[0]             // at (x=0, y=0)
image[4 * dim + 5]   // at (x=5, y=4)
...
```

rotate assignment

```
void rotate(pixel *src, pixel *dst, int dim) {
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(dim – 1 – j, i, dim)] =
                src[RIDX(i, j, dim)];
}
```



preprocessor macros

```
#define DOUBLE(x) x*2

int y = DOUBLE(100);
// expands to:
int y = 100*2;
```

macros are text substitution (1)

```
#define BAD_DOUBLE(x) x*2

int y = BAD_DOUBLE(3 + 3);
// expands to:
int y = 3+3*2;
// y == 9, not 12
```

macros are text substitution (2)

```
#define FIXED_DOUBLE(x) (x)*2

int y = DOUBLE(3 + 3);
// expands to:
int y = (3+3)*2;
// y == 9, not 12
```

RIDX?

```
#define RIDX(x, y, n) ((x) * (n) + (y))

dst[RIDX(dim - 1 - j, 1, dim)]
// becomes *at compile-time*:
dst[((dim - 1 - j) * (dim) + (1))]
```

performance grading

you can submit multiple variants in one file

- grade: best performance
- don't delete stuff that works!

we will measure speedup on my machine

- web viewer for results (with some delay — has to run)

grade: achieving certain speedup on my machine

- thresholds based on results with certain optimizations

general advice

(for when we don't give specific advice)

try techniques from book/lecture that seem applicable

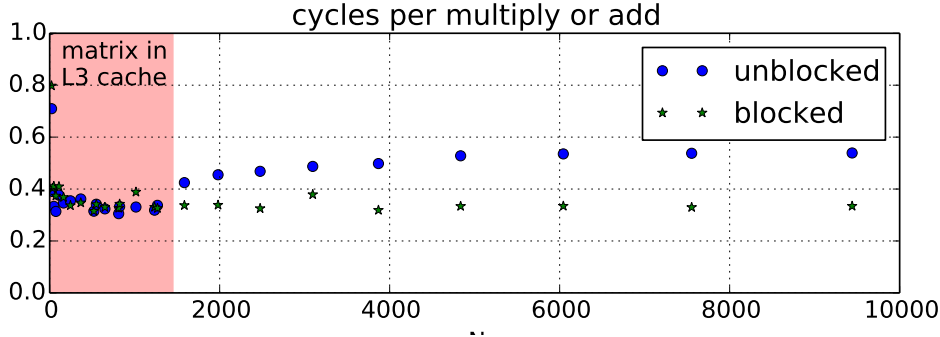
vary numbers (e.g. cache block size)

- often — too big/small is worse

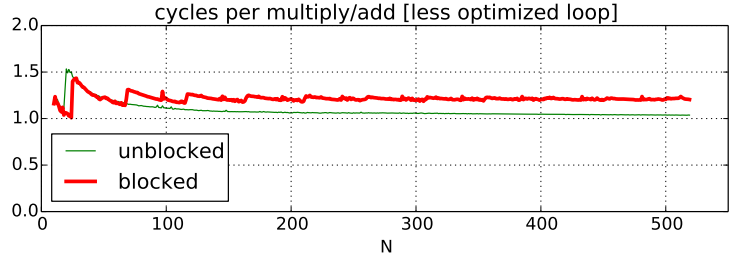
some techniques combine well

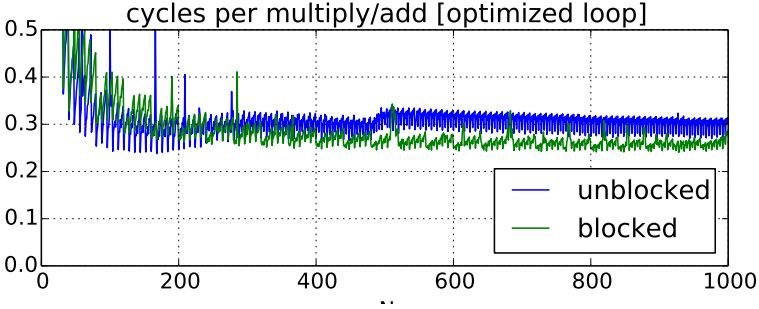
- loop unrolling and cache blocking
- loop unrolling and reassociation/multiple accumulators

cache blocking performance (big sizes)

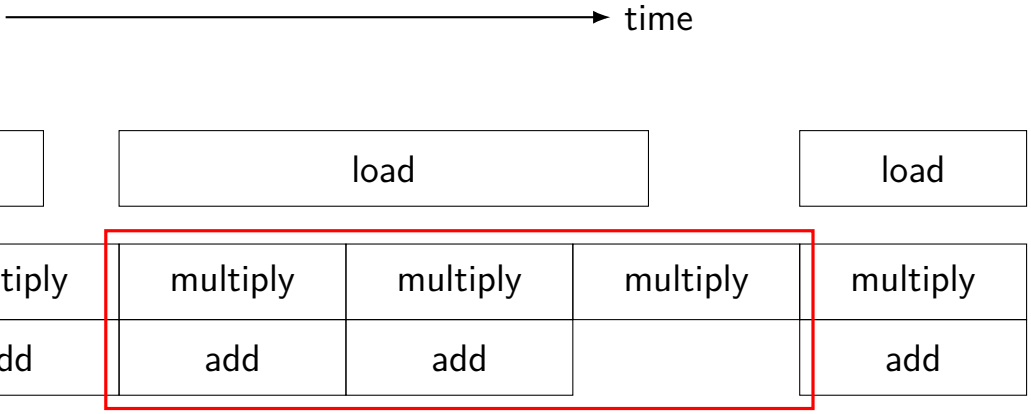


cache blocking performance (small sizes)





overlapping loads and arithmetic



speed of load might not matter if these are slower

optimization and bottlenecks

arithmetic/loop efficiency was the bottleneck

after fixing this, cache performance was the bottleneck

common theme when optimizing:

X may not matter until Y is optimized

the open-source BROOM pipeline

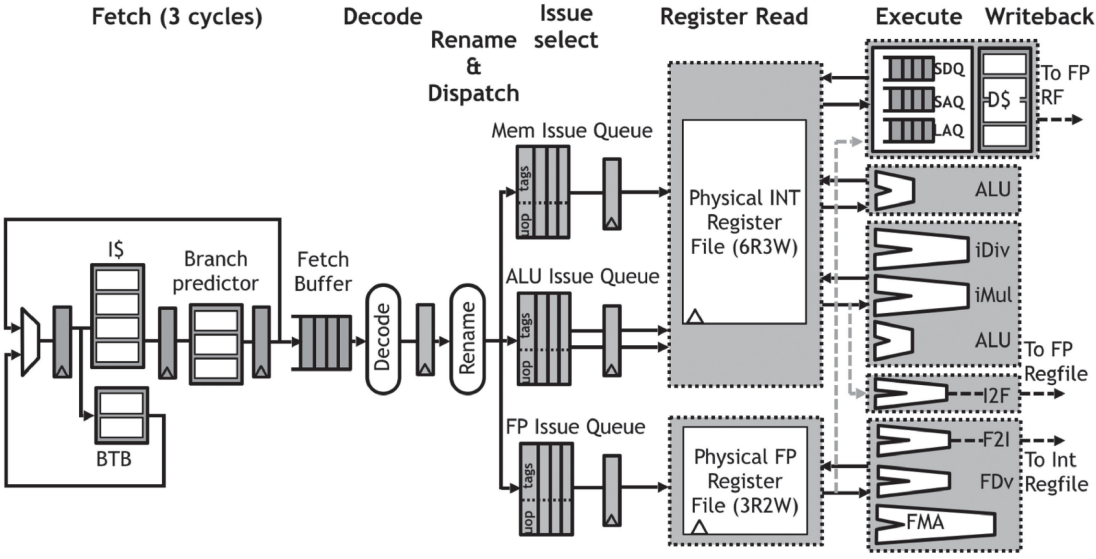
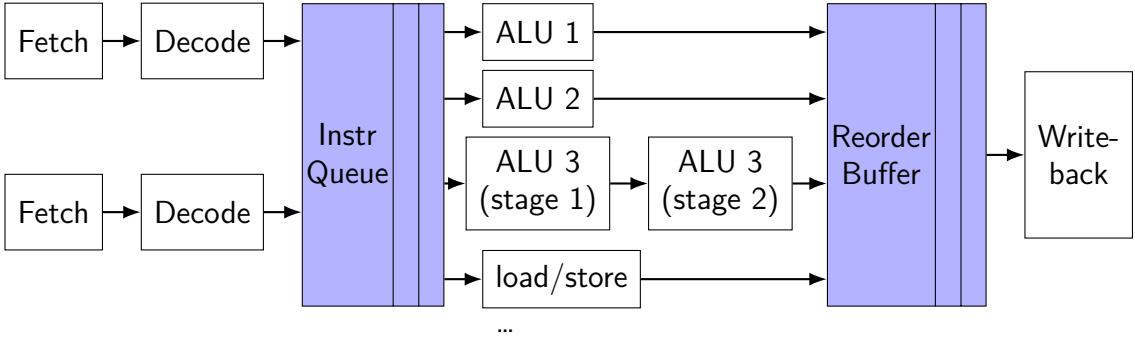
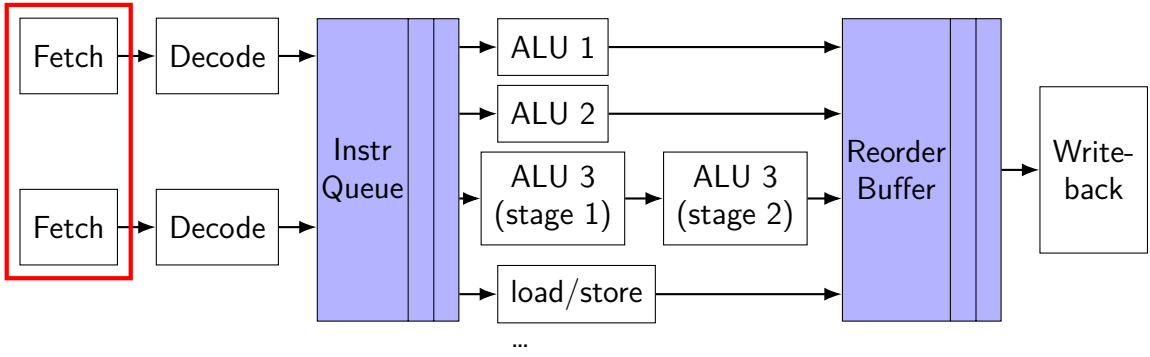


Figure from Celio et al., "BROOM: An Open Source Out-Of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS"

modern CPU design (instruction flow)

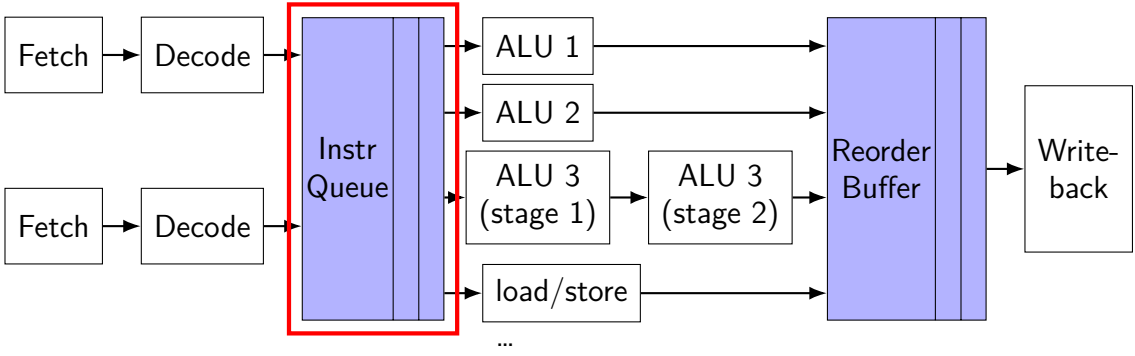


modern CPU design (instruction flow)



fetch multiple instructions/cycle

modern CPU design (instruction flow)

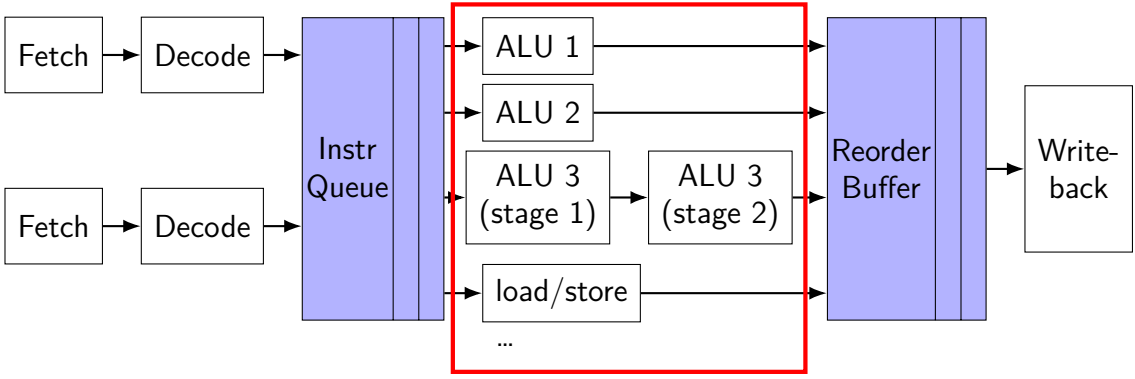


keep list of pending instructions

run instructions from list when operands available

forwarding handled here

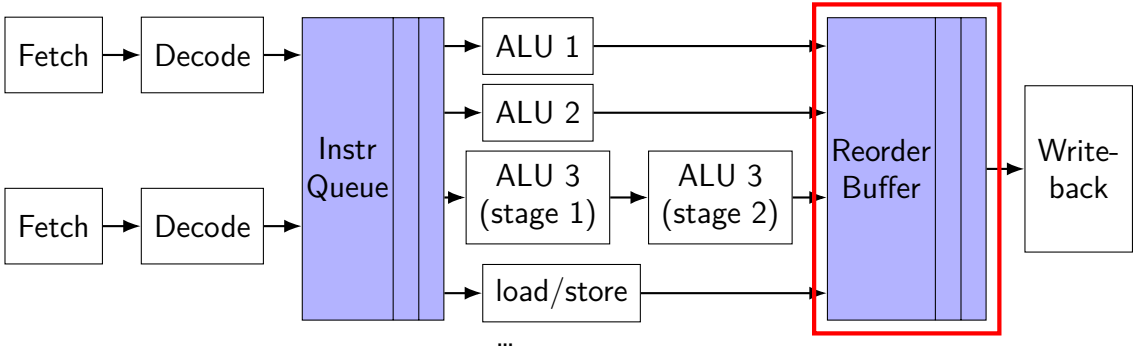
modern CPU design (instruction flow)



multiple "execution units" to run instructions
e.g. possibly many ALUs

sometimes pipelined, sometimes not

modern CPU design (instruction flow)



collect results of finished instructions

helps with forwarding, squashing

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	ready
2	`addq %rbx, %rdx`	waiting for 1
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

execution unit	...
ALU 1	
ALU 2	

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	*running*
2	`addq %rbx, %rdx`	waiting for 1
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	ready
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	*running*
3	`addq %rcx, %rdx`	waiting for 2
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	*running*
4	`cmpq %r8, %rdx`	waiting for 3
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	waiting for 3
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	ready
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	ready
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	*running*
5	`jne ...`	waiting for 4
6	`addq %rax, %rdx`	*running*
7	`addq %rbx, %rdx`	waiting for 6
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	ready
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	ready
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	*running*
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	*running*
8	`addq %rcx, %rdx`	waiting for 7
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	*running*
9	`cmpq %r8, %rdx`	waiting for 8
...	...	

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

instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	*running*
...	...	

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

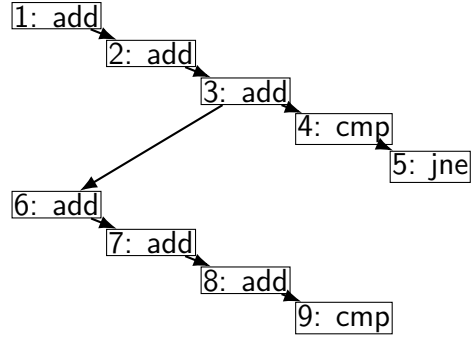
instruction queue operation

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
...	...	

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

data flow

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
...	...	

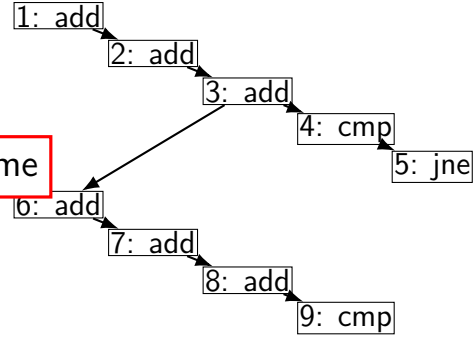


execution unit \ *cycle#*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	...
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	...	

data flow

#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne`	
6	`addq %rax, %rdx`	done
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
…	…	

rule: arrows must go forward in time

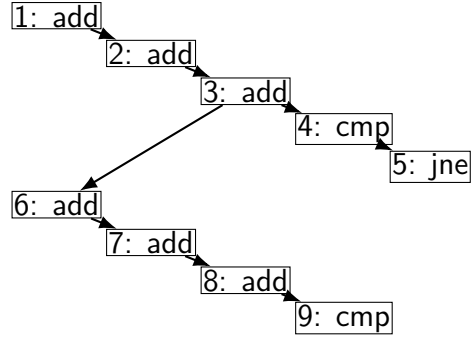


execution unit	*cycle#* 1	2	3	4	5	6	7	…
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	…	

data flow

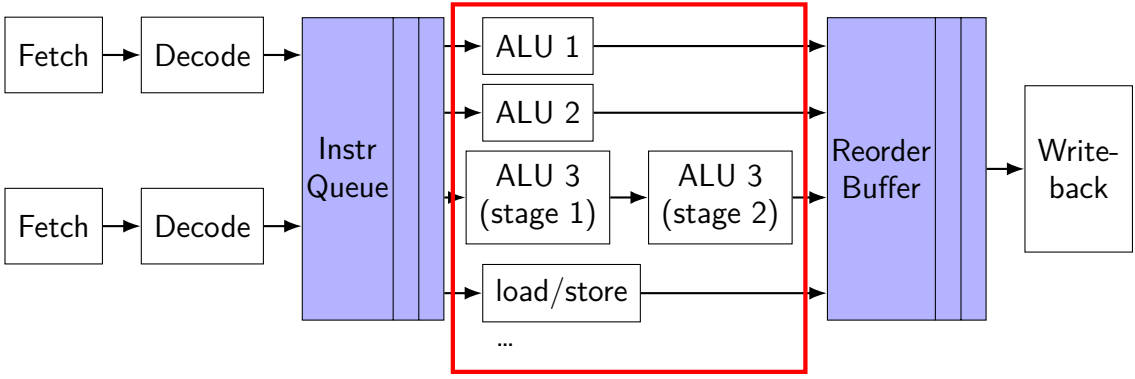
#	*instruction*	*status*
1	`addq %rax, %rdx`	done
2	`addq %rbx, %rdx`	done
3	`addq %rcx, %rdx`	done
4	`cmpq %r8, %rdx`	done
5	`jne ...`	done
6	`ad`	
7	`addq %rbx, %rdx`	done
8	`addq %rcx, %rdx`	done
9	`cmpq %r8, %rdx`	done
...	...	

longest path determines speed



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

modern CPU design (instruction flow)



multiple "execution units" to run instructions
e.g. possibly many ALUs

sometimes pipelined, sometimes not