

Changelog

Changes made in this version not seen in first lecture:

11 April 2018: loop unrolling v cache blocking (2): corrected second example which just did no loop unrolling or cache blocking before correction instead of loop unrolling

loop unrolling (ASM)

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

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

```
loop:
    cml      %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

performance labs

this week — loop optimizations

next week — vector instructions (AKA SIMD)

performance HWs

partners or individual (your choice)

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

loop unrolling v cache blocking (1)

(assuming we started with kij loop order...:)

loop unrolling *and* cache blocking:

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

loop unrolling

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

loop unrolling v cache blocking (2)

(assuming we started with kij loop order...:)

loop unrolling *and* cache blocking:

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

pretty useless loop unrolling

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

interlude: real CPUs

modern CPUs:

execute **multiple instructions at once**

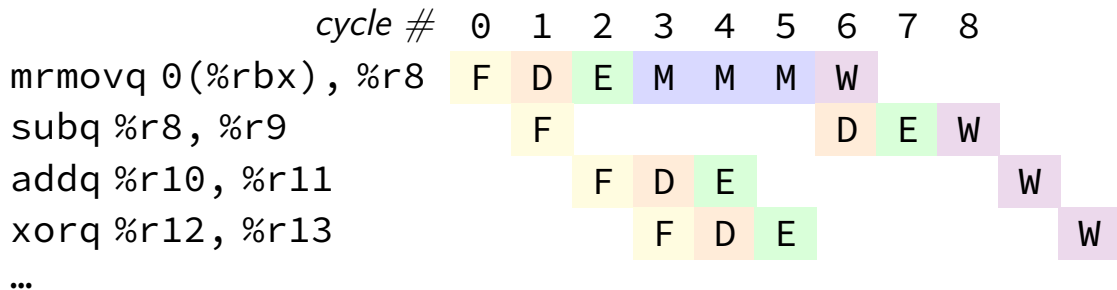
execute instructions **out of order** — whenever **values available**

beyond pipelining: out-of-order

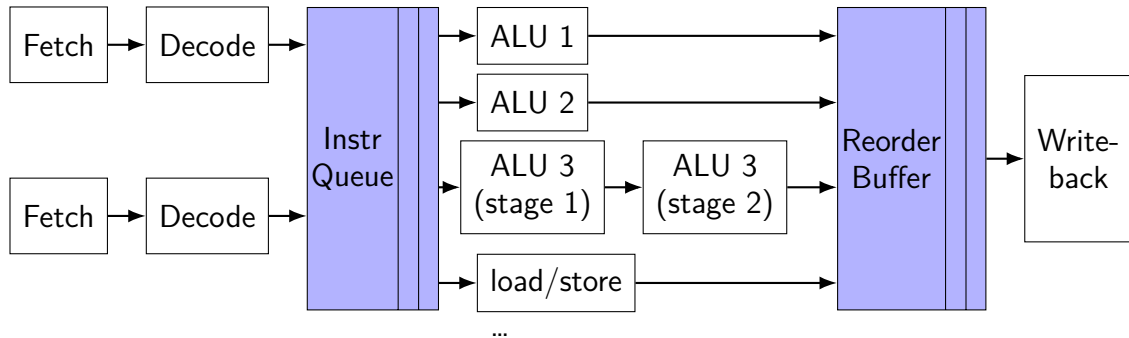
find **later instructions to do** instead of stalling

lists of available instructions in pipeline registers
take any instruction with available values

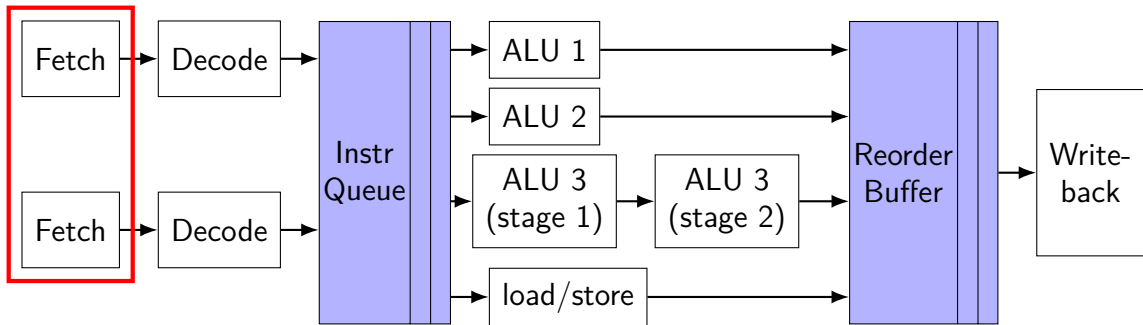
provide **illusion that work is still done in order**
much more complicated hazard handling logic



modern CPU design (instruction flow)

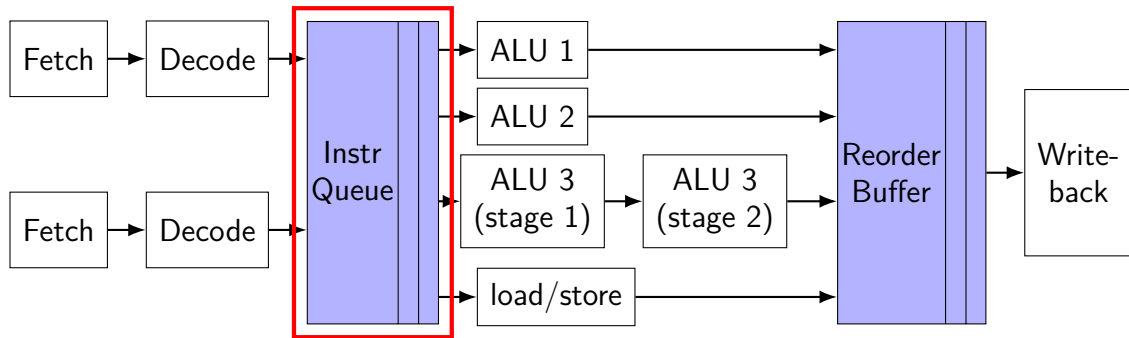


modern CPU design (instruction flow)



fetch multiple instructions/cycle

modern CPU design (instruction flow)

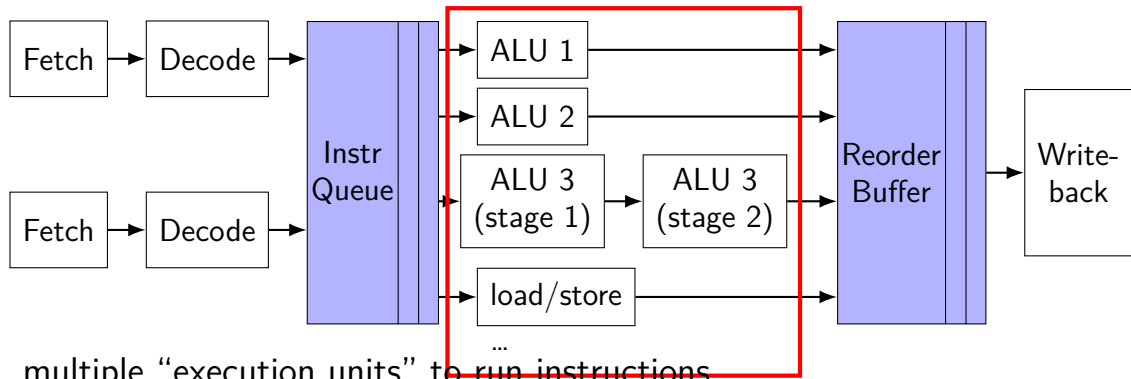


keep list of pending instructions

run instructions from list when operands available

forwarding handled here

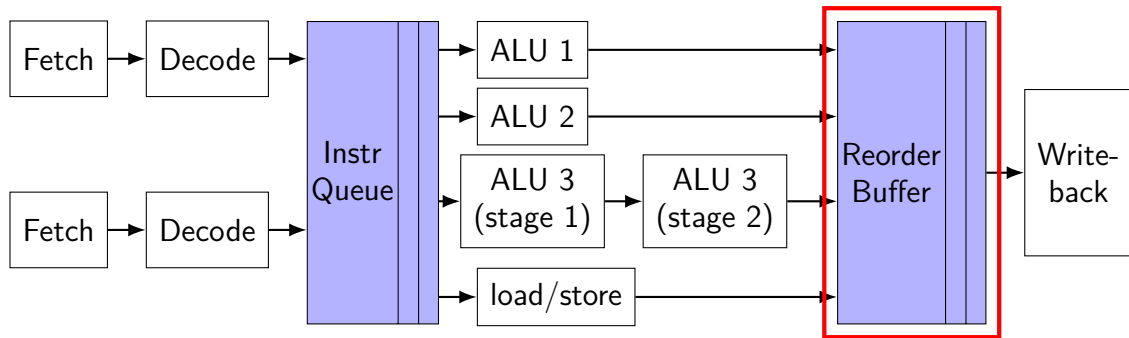
modern CPU design (instruction flow)



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 | <i>running</i> |
| 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 | <code>addq %rax, %rdx</code> | done |
| 2 | <code>addq %rbx, %rdx</code> | <i>running</i> |
| 3 | <code>addq %rcx, %rdx</code> | waiting for 2 |
| 4 | <code>cmpq %r8, %rdx</code> | waiting for 3 |
| 5 | <code>jne ...</code> | waiting for 4 |
| 6 | <code>addq %rax, %rdx</code> | waiting for 3 |
| 7 | <code>addq %rbx, %rdx</code> | waiting for 6 |
| 8 | <code>addq %rcx, %rdx</code> | waiting for 7 |
| 9 | <code>cmpq %r8, %rdx</code> | waiting for 8 |

... ..

| execution unit | cycle# 1 | 2 | ... |
|----------------|----------|---|-----|
| ALU 1 | 1 | 2 | |
| ALU 2 | — | — | |

instruction queue operation

| # | instruction | status |
|---|------------------------------|----------------|
| 1 | <code>addq %rax, %rdx</code> | done |
| 2 | <code>addq %rbx, %rdx</code> | done |
| 3 | <code>addq %rcx, %rdx</code> | <i>running</i> |
| 4 | <code>cmpq %r8, %rdx</code> | waiting for 3 |
| 5 | <code>jne ...</code> | waiting for 4 |
| 6 | <code>addq %rax, %rdx</code> | waiting for 3 |
| 7 | <code>addq %rbx, %rdx</code> | waiting for 6 |
| 8 | <code>addq %rcx, %rdx</code> | waiting for 7 |
| 9 | <code>cmpq %r8, %rdx</code> | 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 | <i>running</i> |
| 5 | jne ... | waiting for 4 |
| 6 | addq %rax, %rdx | <i>running</i> |
| 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 ... | done |
| 6 | addq %rax, %rdx | done |
| 7 | addq %rbx, %rdx | <i>running</i> |
| 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 | <i>running</i> |
| 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 | <i>running</i> |

... ..

| 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 |

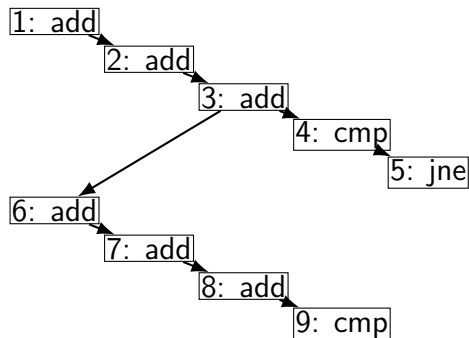
... ..

| 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 |

... ..

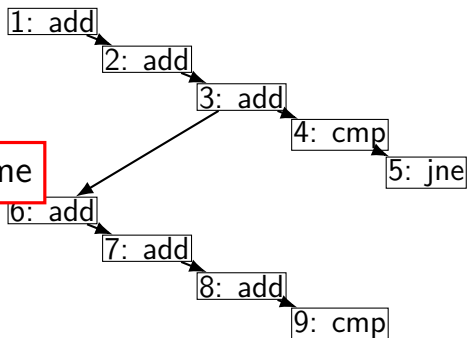


| execution unit | cycle# | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|----------------|--------|---|---|---|---|---|---|-----|-----|
| ALU 1 | | 1 | 2 | 3 | 4 | 5 | 8 | 9 | |
| ALU 2 | | — | — | — | 6 | 7 | — | ... | |

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 | done |
| 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



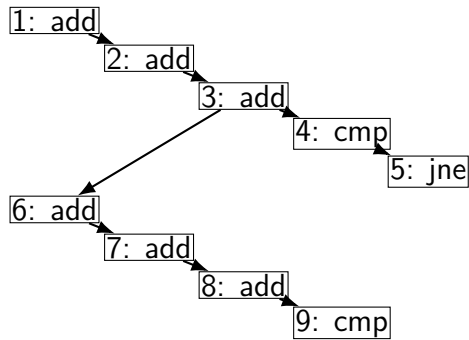
... ..

| 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

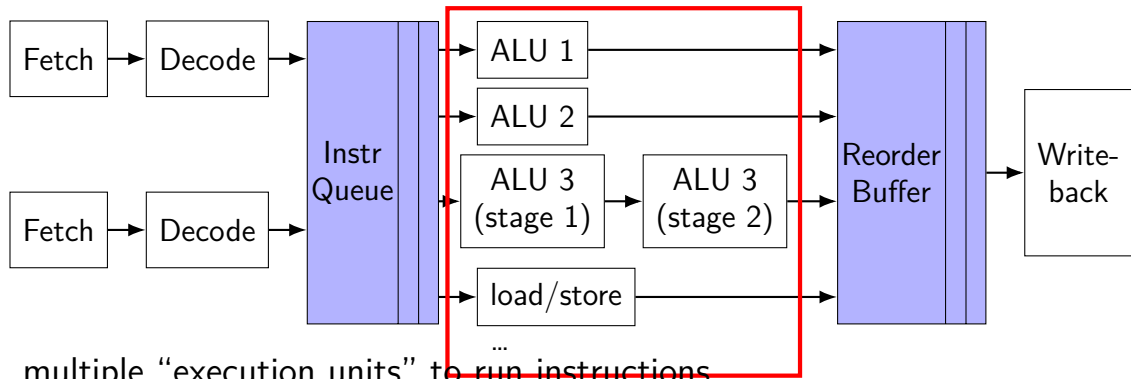
| # | instruction | status |
|---|-----------------|--------|
| 1 | addq %rax, %rdx | done |
| 2 | addq %rbx, %rdx | done |
| 3 | addq %rcx, %rdx | done |
| 4 | cmpq %r8, %rdx | done |
| 5 | jne ... | done |
| 6 | addq %rax, %rdx | done |
| 7 | addq %rbx, %rdx | done |
| 8 | addq %rcx, %rdx | done |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

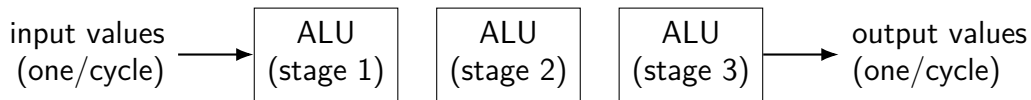
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



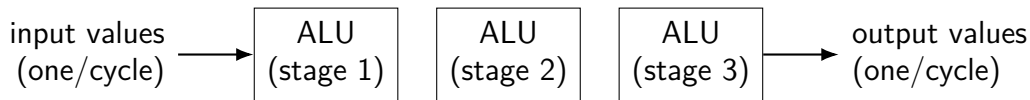
execution units AKA functional units (1)

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

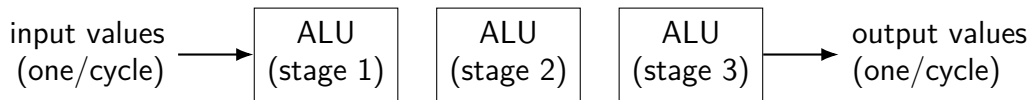
execution units AKA functional units (1)

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

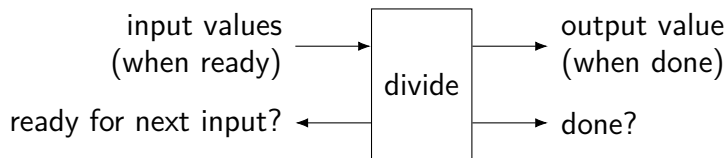
no parallelism!

execution units AKA functional units (2)

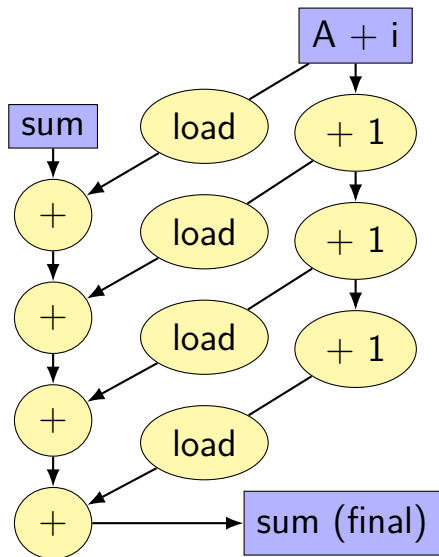
where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:

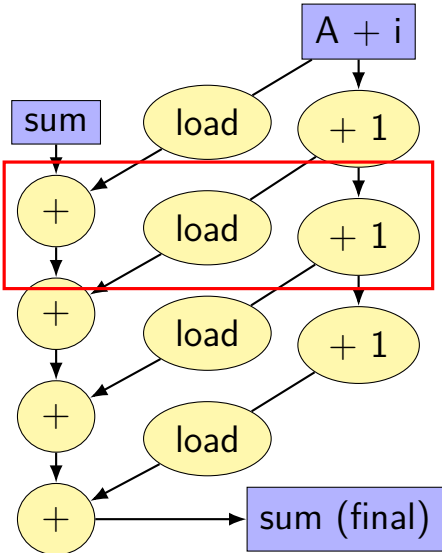


data flow model and limits



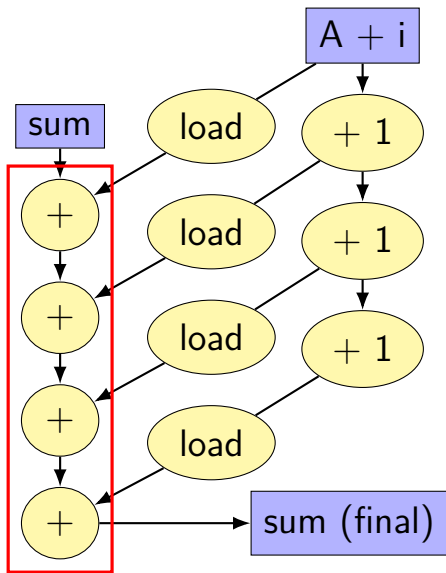
```
for (int i = 0; i < N; i += K) {  
    sum += A[i];  
    sum += A[i+1];  
    ...  
}
```

data flow model and limits



three ops/cycle (if each one cycle)

data flow model and limits



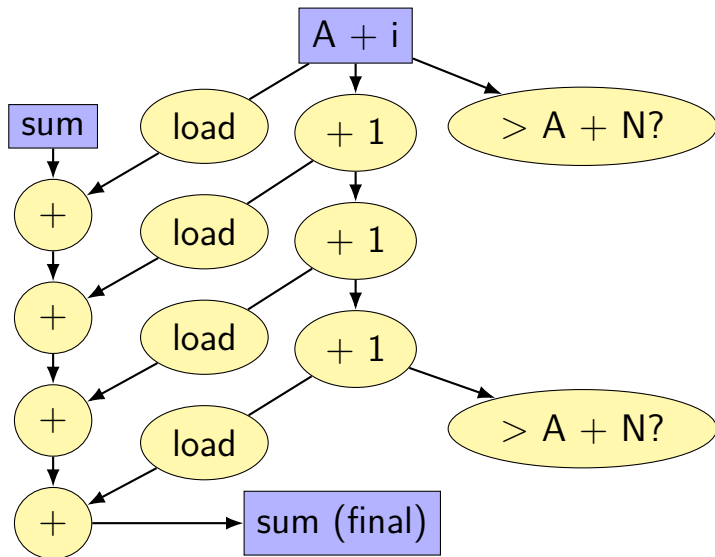
need to do additions

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
```

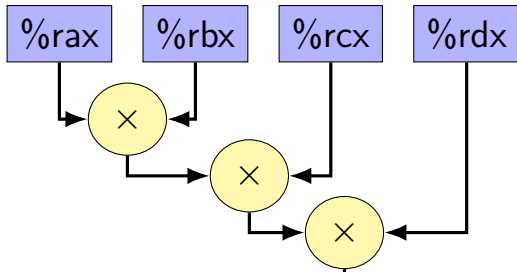
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

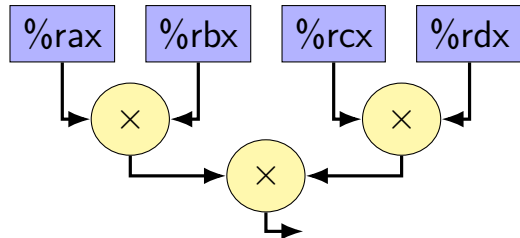
$$((a \times b) \times c) \times d$$

```
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
```



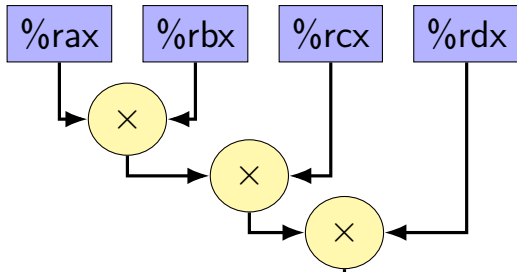
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

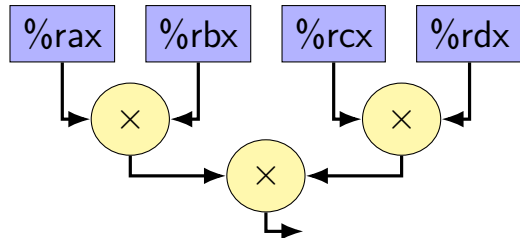
$$((a \times b) \times c) \times d$$

```
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

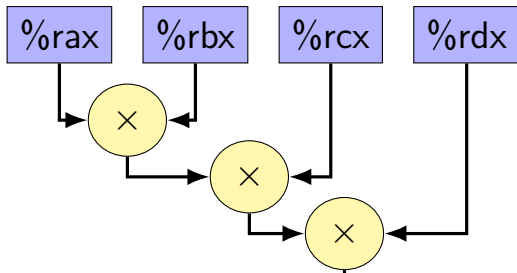
assume a single pipelined, 5-cycle latency multiplier

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
```

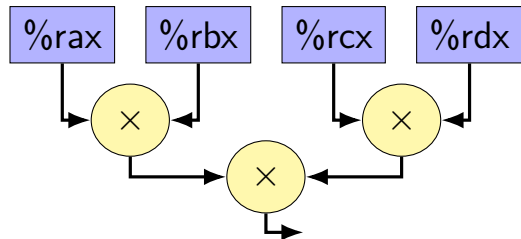
15 cycles



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

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

11 cycles



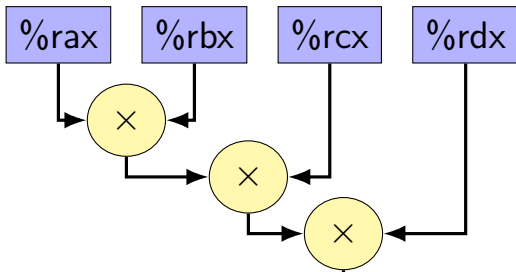
reassociation

assume a single pipelined, 5-cycle latency multiplier

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

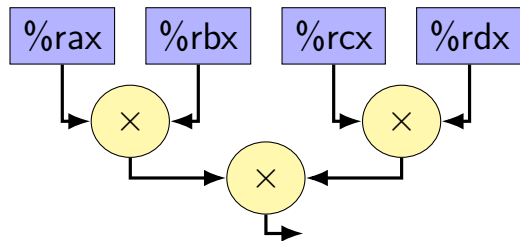
$$((a \times b) \times c) \times d$$

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

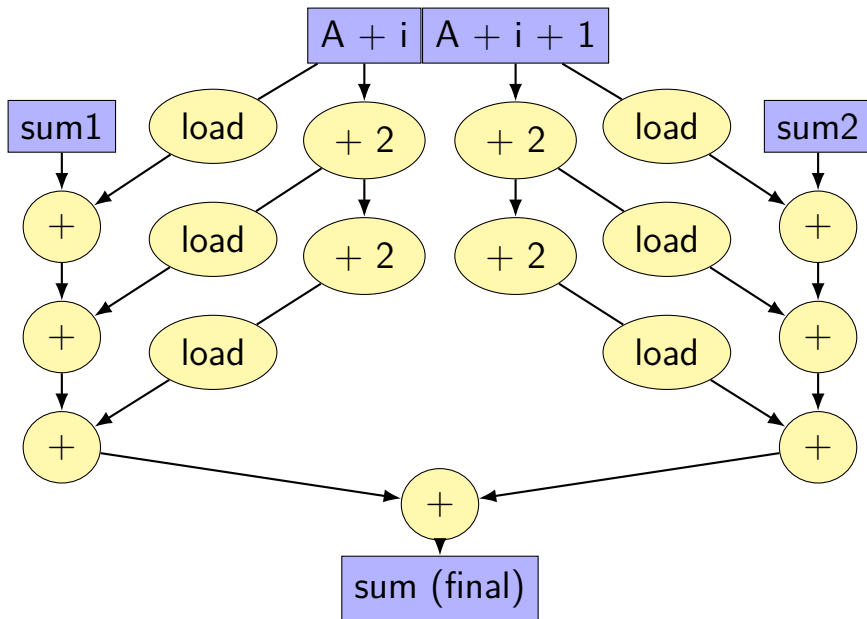


$$(a \times b) \times (c \times d)$$

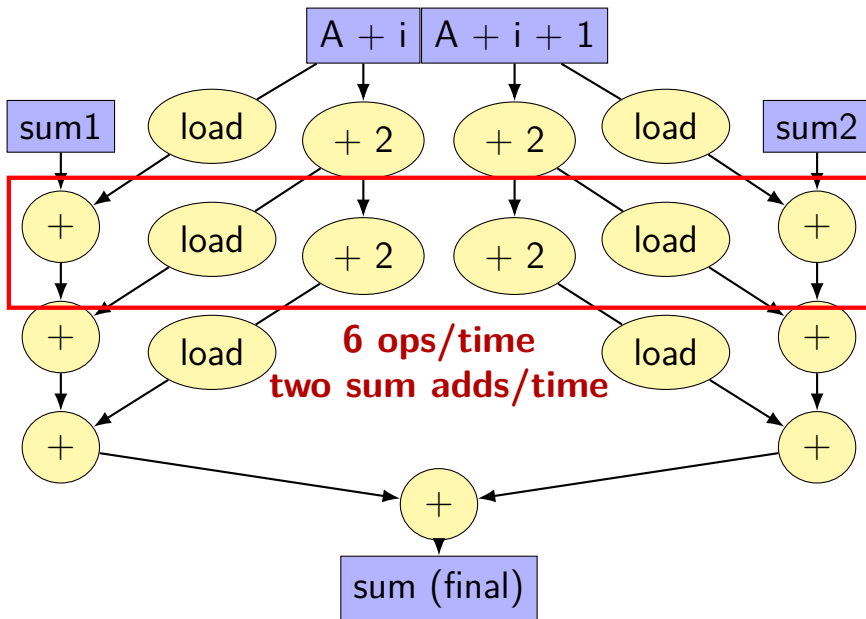
```
imulq %rbx, %rax  
imulq %rcx, %rdx  
imulq %rdx, %rax
```



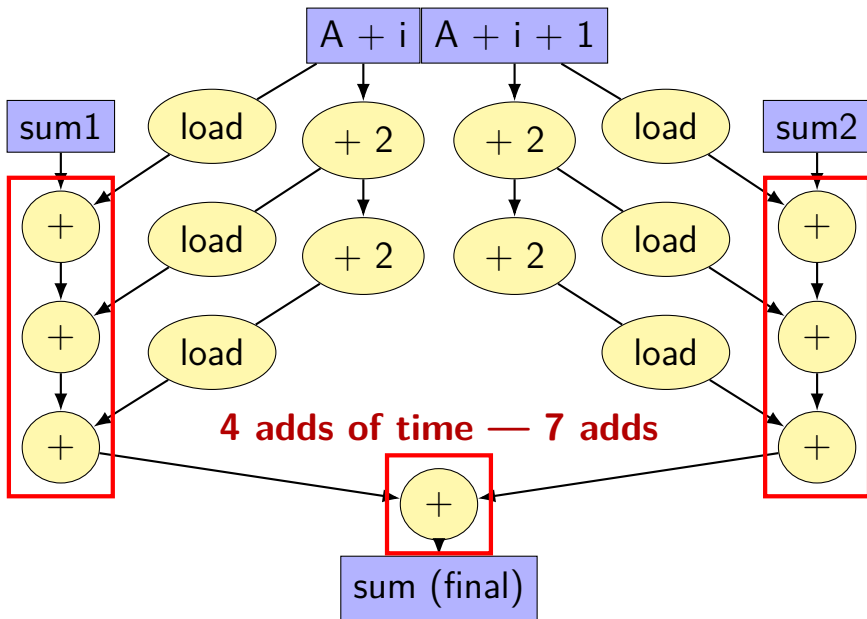
better data-flow



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

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

loop overhead

compiler not as efficient as it could have been

my machine: 4 add/etc. or branches/cycle

4 copies of ALU (effectively)

$$(2 + 2/16 + 1/16 + 1/16) \div 4 \approx 0.57 \text{ cycles/element}$$

constant multiplies/divides (1)

```
unsigned int fiveEights(unsigned int x) {  
    return x * 5 / 8;  
}
```

```
fiveEights:  
    leal    (%rdi,%rdi,4), %eax  
    shrl   $3, %eax  
    ret
```

constant multiplies/divides (2)

```
int oneHundredth(int x) { return x / 100; }
```

oneHundredth:

```
    movl    %edi, %eax
    movl    $1374389535, %edx
    sarl    $31, %edi
    imull   %edx
    sarl    $5, %edx
    movl    %edx, %eax
    subl    %edi, %eax
    ret
```

$$\frac{1374389535}{2^{37}} \approx \frac{1}{100}$$

constant multiplies/divides

compiler is very good at handling

...but need to actually use constants

addressing efficiency

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

tons of multiplies by N??

isn't that slow?

addressing transformation

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

transforms loop to **iterate with pointer**

compiler will usually do this!

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

addressing transformation

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

transforms loop to **iterate with pointer**

compiler will usually do this!

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

addressing efficiency

compiler will **usually** eliminate slow multiplies
doing transformation yourself often slower if so

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

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

if it doesn't — do it yourself

another addressing transformation

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

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

compiler will do this, too

another addressing transformation

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

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

compiler will do this, too

another addressing transformation

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

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

storing 20 AiX_base ? — need the stack

another addressing transformation

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

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

storing 20 AiX_base ? — need the stack

alternative addressing transformation

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

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

more dependencies (latency bound?), more additions?, less registers
might need multiple accumulator transformation?

alternative addressing transformation

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

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

more dependencies (latency bound?), more additions?, less registers
might need multiple accumulator transformation?

vector instructions

modern processors have registers that hold “vector” of values

example: X86-64 has 128-bit registers

4 ints or 4 floats or 2 doubles or ...

128-bit registers named %xmm0 through %xmm15

instructions that act on **all values in register**

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

extra copies of ALUs only accessed by vector instructions

example vector instruction

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

Suppose registers contain (interpreted as 4 ints)

`%xmm0`: [1, 2, 3, 4]

`%xmm1`: [5, 6, 7, 8]

Result will be:

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

vector instructions

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

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