

optimization 2

last time

loop unrolling

- reduce loop overhead (maintaining indices, check bounds)
- less instructions, so should be faster
- limit to benefit???

out-of-order, multiple issue processors (*superscalar*)

need for multiple versions of registers

- e.g. write-after-write hazards (writeback stages reordered)
- track versions to know what to forward

register renaming

- architectural (assembly) to physical (internal) register mapping
- new physical register after each write
- translate instructions to use physical registers in preprocessing step
- 'commit' step identifies when physical registers become free

register renaming: missing pieces

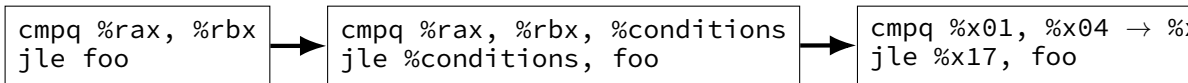
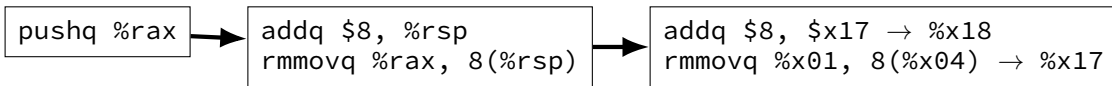
what about “hidden” inputs like `%rsp`, condition codes?

one solution: translate to instructions with additional register parameters

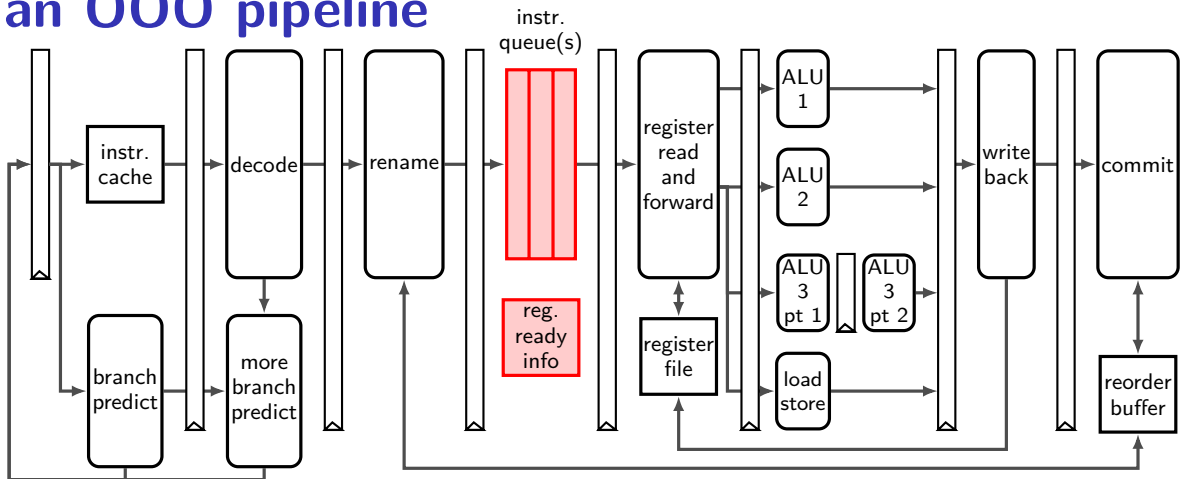
- making `%rsp` explicit parameter

- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones



an OOO pipeline



instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit

ALU 1

ALU 2

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit cycle# 1
 ALU 1 1
 ALU 2

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit cycle# 1
 ALU 1 1
 ALU 2

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

execution unit cycle# 1
ALU 1 1
ALU 2 —

...

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

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

scoreboard

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

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

instruction queue and dispatch

instruction queue

#	instruction
1	<code>mrmovq (%x04) → %x06</code>
2	<code>mrmovq (%x05) → %x07</code>
3	<code>addq %x01, %x02 → %x08</code>
4	<code>addq %x01, %x06 → %x09</code>
5	<code>addq %x01, %x07 → %x10</code>

... ..

scoreboard

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

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

ALU

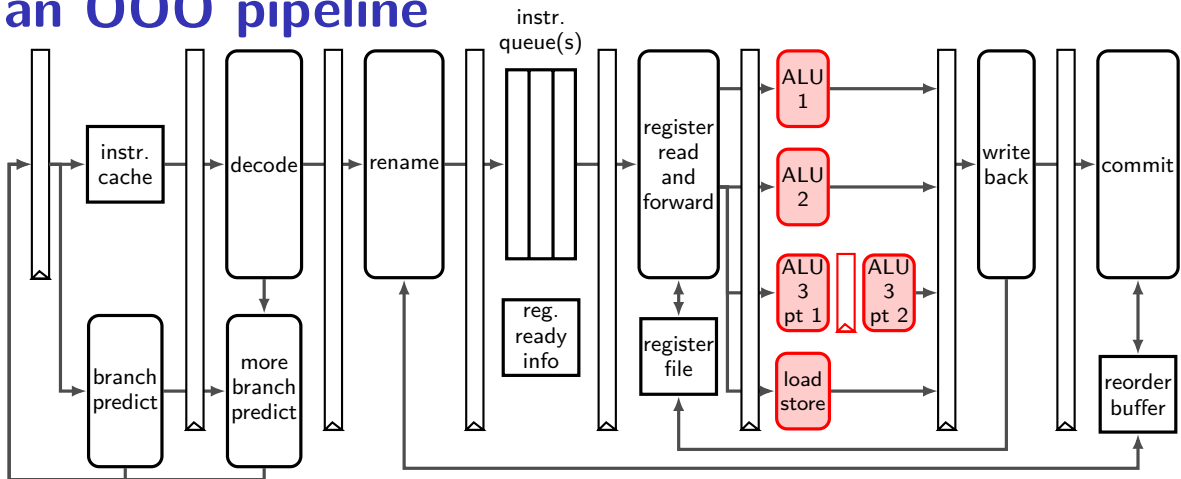
data cache



assume

1 cycle/access

an OOO pipeline



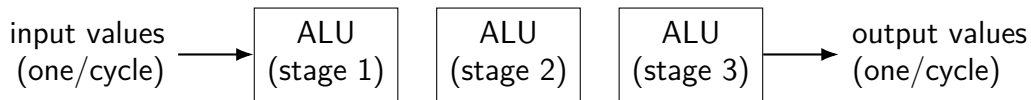
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



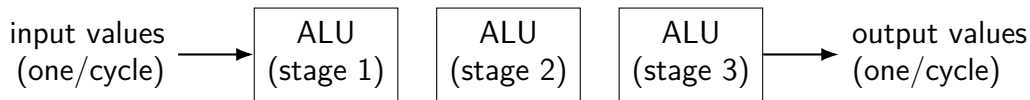
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



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

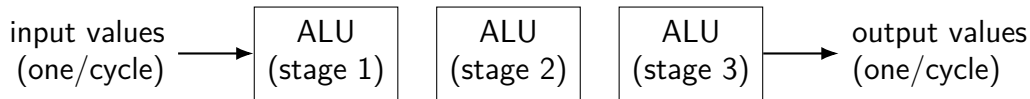
execution units AKA functional units (1)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

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



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

3×3 cycles + any time to forward values

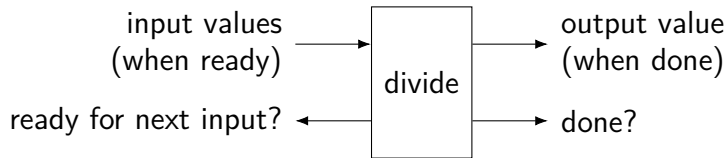
no parallelism!

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue

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

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

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
...	...

...

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

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
...	...

...

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#	
ALU 1 (add, cmp, jxx)	1	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	3
	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	4	
ALU 2 (add, cmp, jxx)		—	—	—	—	—	
ALU 3 (mul) start		2	3	7	8	8	
ALU 3 (mul) end			2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

scoreboard

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

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

instruction queue and dispatch (multicycle)

instruction queue

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

... ..

scoreboard

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

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

OOO limitations

can't always find instructions to run

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

need to track all uncommitted instructions

- can only go so far ahead

- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

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

OOO limitations

can't always find instructions to run

plenty of instructions, but all depend on unfinished ones

programmer can adjust program to help this

need to track all uncommitted instructions

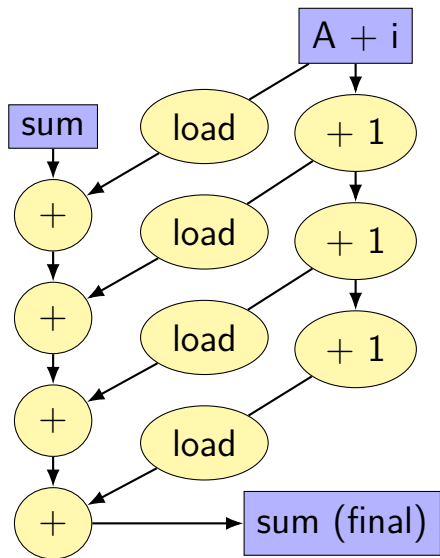
can only go so far ahead

e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

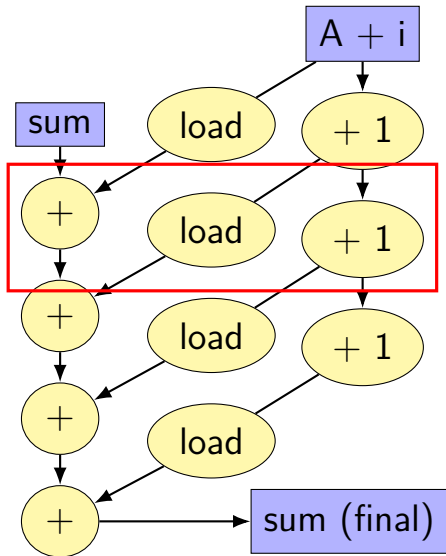
e.g. Intel Skylake: approx 16 cycles (v. 2 for pipehw2 CPU)

data flow model and limits



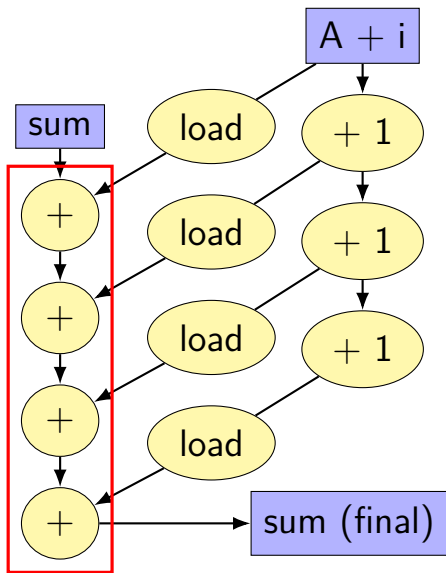
```
for (int i = 0; i < N; i += K) {  
    sum += A[i];  
    sum += A[i+1];  
    ...  
}
```

data flow model and limits



three ops/cycle (if each one cycle)

data flow model and limits



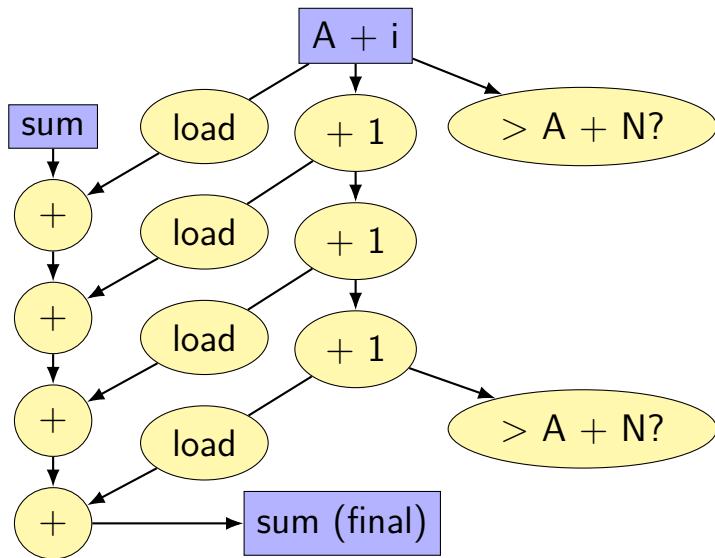
need to do additions

one-at-a-time

book's name: critical path

time needed: sum of latencies

data flow model and limits



reassociation

assume a single pipelined, 5-cycle latency multiplier

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

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

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

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

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


reassociation

assume a single pipelined, 5-cycle latency multiplier

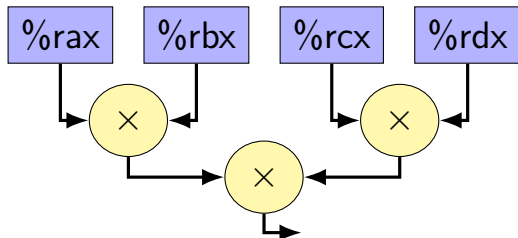
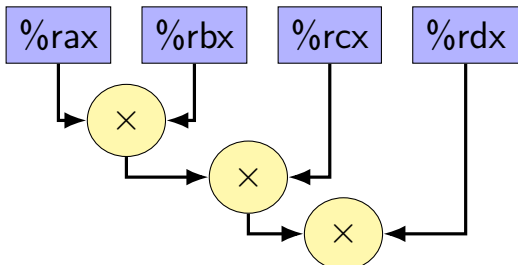
exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

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

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

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

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



reassociation

assume a single pipelined, 5-cycle latency multiplier

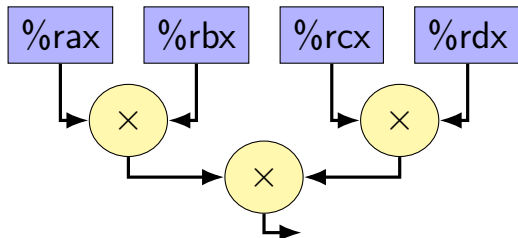
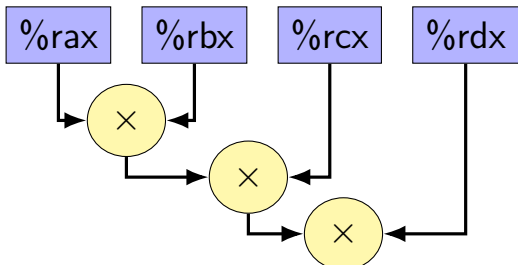
exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

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

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

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

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



reassociation

assume a single pipelined, 5-cycle latency multiplier

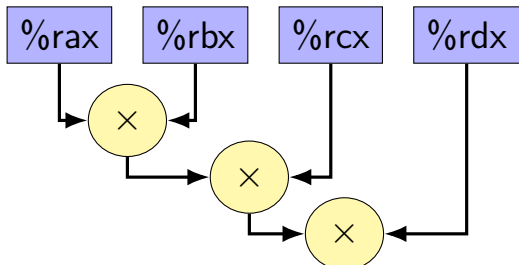
exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

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

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

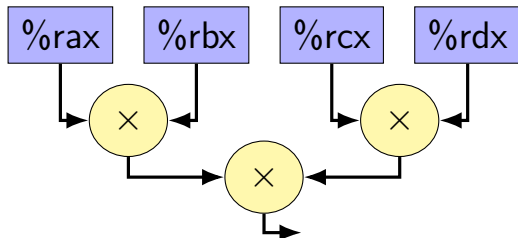
```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```

15
cycles



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

11
cycles



reassociation

assume a single pipelined, 5-cycle latency multiplier

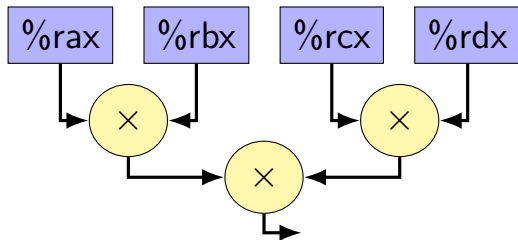
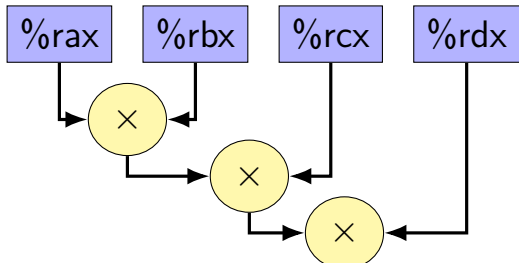
exercise: how long does each take? assume instant forwarding. (hint: think about data-flow graph)

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

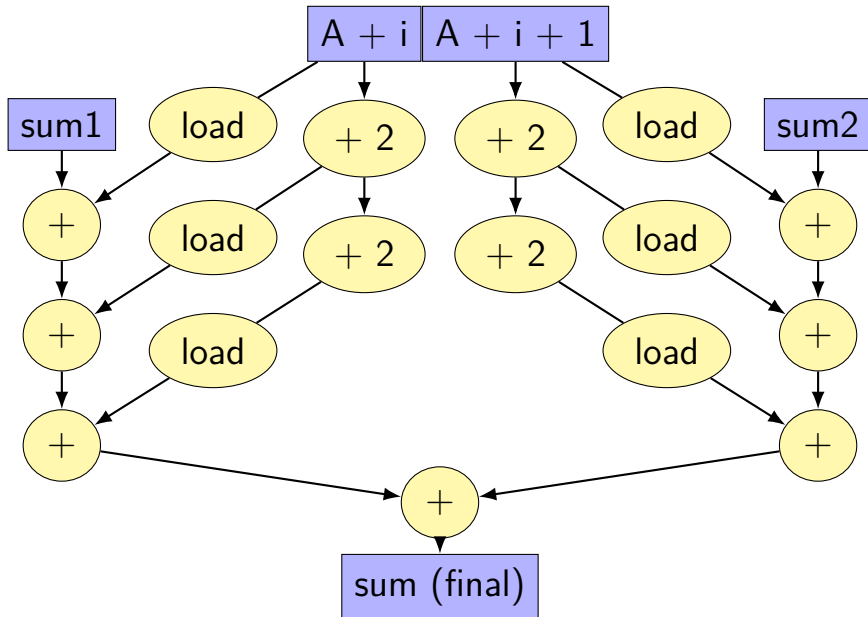
$$(a \times b) \times (c \times d)$$

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

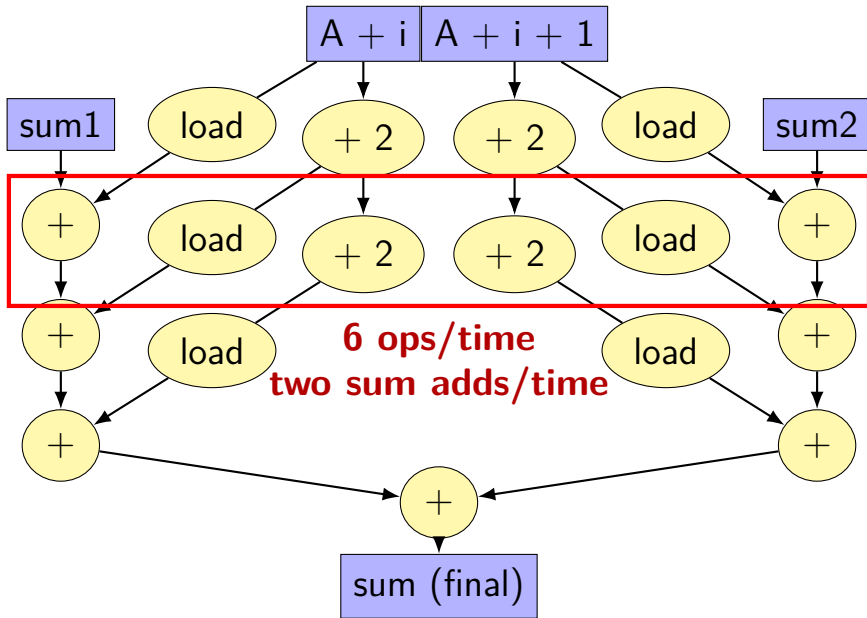
```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



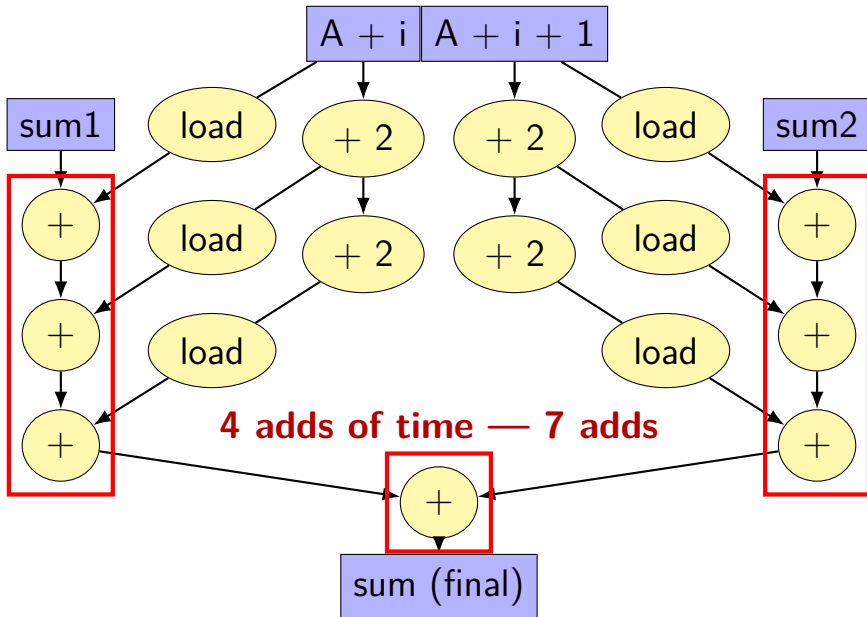
better data-flow



better data-flow



better data-flow



multiple accumulators

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


multiple accumulators performance

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

16x unrolling, variable number of accumulators

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

starts hurting after too many accumulators

why?

multiple accumulators performance

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

16x unrolling, variable number of accumulators

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

starts hurting after too many accumulators

why?

8 accumulator assembly

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

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

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

16 accumulator assembly

compiler runs out of registers

starts to use the stack instead:

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

code does **extra cache accesses**

also — already using all the adders available all the time

so performance increase not possible

multiple accumulators performance

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

16x unrolling, variable number of accumulators

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

starts hurting after too many accumulators

why?

maximum performance

2 additions per element:

- one to add to sum

- one to compute address (part of mov)

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

- over 16 because loop unrolled 16 times

- loop overhead

- compiler not as efficient as it could have been

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

probably $2 + 1/4$ microinstructions, too

- cmp+jXX apparently becomes 1 microinstruction (on this Intel CPU)

- probably extra microinstruction for load in add

hardware limits on my machine

4(?) register renamings per cycle

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

4-6 instructions decoded/cycle

(depending on instructions)

4(?) microinstructions committed/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 committed/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

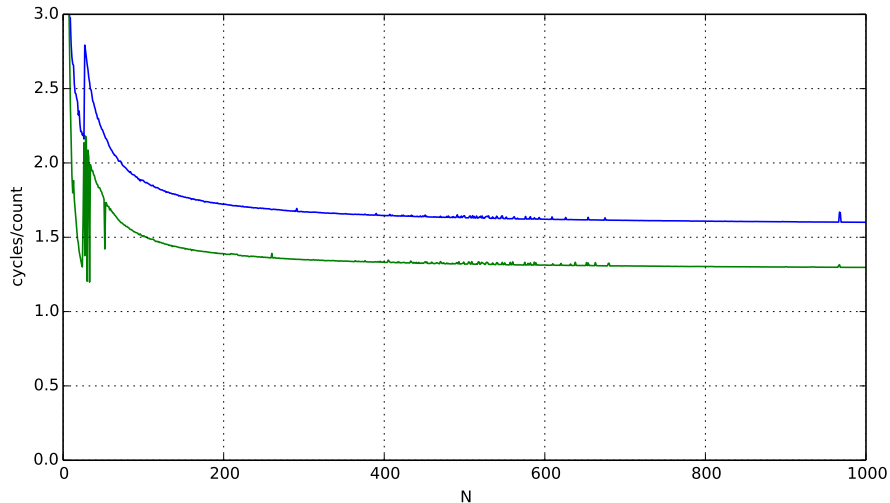
```
void sumRows1(int *result, int *matrix, int N) {  
    for (int row = 0; row < N; ++row) {  
        result[row] = 0;  
        for (int col = 0; col < N; ++col)  
            result[row] += matrix[row * N + col];  
    }  
}
```

non-contrived aliasing

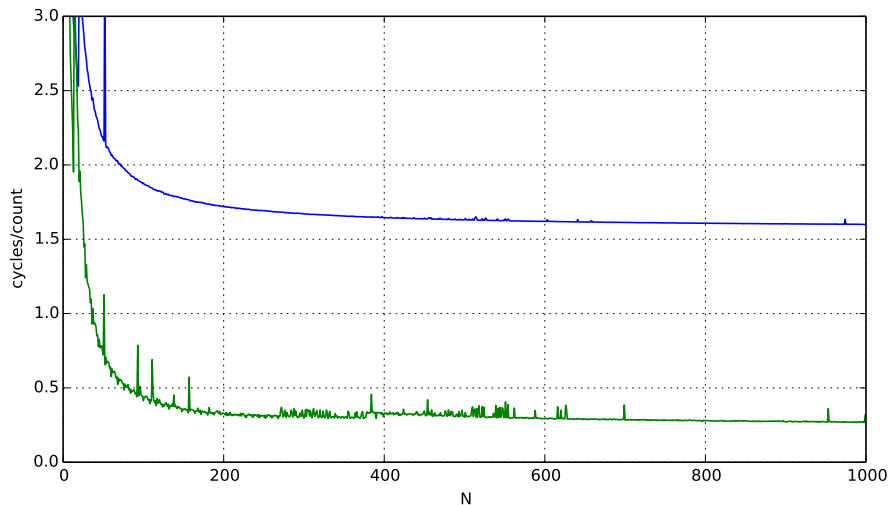
```
void sumRows1(int *result, int *matrix, int N) {  
    for (int row = 0; row < N; ++row) {  
        result[row] = 0;  
        for (int col = 0; col < N; ++col)  
            result[row] += matrix[row * N + col];  
    }  
}
```

```
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];
    }
}
```

(if time) addressing efficiency

backup slides

instruction queue and dispatch

instruction queue

#	instruction
1	<code>mrmovq (%x04) → %x06</code>
2	<code>mrmovq (%x05) → %x07</code>
3	<code>addq %x01, %x02 → %x08</code>
4	<code>addq %x01, %x06 → %x09</code>
5	<code>addq %x01, %x07 → %x10</code>

... ..

scoreboard

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

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

ALU

data cache (stage 1)

data cache (stage 2)

data cache (stage 3)

addressing efficiency

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

tons of multiplies by N??

isn't that slow?

addressing transformation

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

transforms loop to **iterate with pointer**

compiler will often do this

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

addressing transformation

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

transforms loop to **iterate with pointer**

compiler will often do this

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

addressing efficiency

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

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

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

if it doesn't — do it yourself

another addressing transformation

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

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

compiler will sometimes do this, too

another addressing transformation

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

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

compiler will sometimes do this, too

another addressing transformation

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

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

storing 20 A_{iX_base} ? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

another addressing transformation

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

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

storing 20 A_{iX_base} ? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

alternative addressing transformation

instead of:

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

could do:

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

avoids spilling on the stack, but more dependencies

alternative addressing transformation

instead of:

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

could do:

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

avoids spilling on the stack, but more dependencies

addressing efficiency generally

mostly: compiler does very good job itself

- eliminates multiplications, use pointer arithmetic

- often will do better job than if how typically programming would do it manually

sometimes compiler won't take the best option

- if spilling to the stack: can cause weird performance anomalies

- if indexing gets too complicated — might not remove multiply

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

- convert to pointer arith. without multiplies

recall: shifts

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

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

“strength reduction”: replacing complicated op with simpler one

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

Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

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

2 load functional units

but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

determines how far ahead branch mispredictions, etc. can happen

example assembly (unoptimized)

```
long sum(long *A, int N) {  
    long result = 0;  
    for (int i = 0; i < N; ++i)  
        result += A[i];  
    return result;  
}
```

```
sum:    ...
```

```
the_loop:
```

```
    ...
```

```
    leaq    0(,%rax,8), %rdx // offset ← i * 8  
    movq   -24(%rbp), %rax // get A from stack  
    addq   %rdx, %rax      // add offset  
    movq   (%rax), %rax    // get *(A+offset)  
    addq   %rax, -8(%rbp) // add to sum, on stack  
    addl   $1, -12(%rbp)  // increment i
```

```
condition:
```

```
    movl   -12(%rbp), %eax  
    cmpl  -28(%rbp), %eax  
    jl    the_loop
```

```
    ...
```

example assembly (gcc 5.4 -Os)

```
long sum(long *A, int N) {  
    long result = 0;  
    for (int i = 0; i < N; ++i)  
        result += A[i];  
    return result;  
}
```

```
sum:  
    xorl    %edx, %edx  
    xorl    %eax, %eax  
the_loop:  
    cmpl   %edx, %esi  
    jle    done  
    addq   (%rdi,%rdx,8), %rax  
    incq   %rdx  
    jmp    the_loop  
done:  
    ret
```

example assembly (gcc 5.4 -O2)

```
long sum(long *A, int N) {  
    long result = 0;  
    for (int i = 0; i < N; ++i)  
        result += A[i];  
    return result;  
}
```

```
sum:  
    testl    %esi, %esi  
    jle     return_zero  
    leal   -1(%rsi), %eax  
    leaq   8(%rdi,%rax,8), %rdx // rdx=end of A  
    xorl   %eax, %eax  
the_loop:  
    addq   (%rdi), %rax // add to sum  
    addq   $8, %rdi    // advance pointer  
    cmpq   %rdx, %rdi  
    jne    the_loop  
    rep   ret  
return_zero:    ...
```

example assembly (gcc 9.2 -O3)

sum:

```
    testl    %esi, %esi  
    ... /* approx 10 lines omitted */
```

the_loop:

```
    movdqu  (%rax), %xmm2 /* ←- load 16 bytes from array */  
    addq    $16, %rax  
    paddq   %xmm2, %xmm0 /* ←- add 2 pairs of longs */  
    cmpq    %rdx, %rax  
    jne     the_loop  
    ... /* approx 20 lines omitted */  
    ret
```

example assembly (gcc 9.2 -O3 -march=skylake)

sum:

```
testl    %esi, %esi  
... /* approx 10 lines omitted */
```

the_loop:

```
vpaddq   (%rax), %ymm0, %ymm0 /* ← add 4 pairs of longs */  
addq     $32, %rax  
cmpq     %rdx, %rax  
jne      the_loop  
... /* approx 20 lines omitted */  
ret
```

gcc 9.2 -O3 -funroll-loops -march=skylake

sum:

```
testl    %esi, %esi
```

```
... /* approx 60 lines omitted */
```

the_loop: /* loop unrolled 8 times + instrs that add 4 pairs at a t

```
vpaddq   (%r8), %ymm0, %ymm1 /* ←- add 4 pairs of longs */
```

```
addq     $256, %r8
```

```
vpaddq   -224(%r8), %ymm1, %ymm2
```

```
vpaddq   -192(%r8), %ymm2, %ymm3
```

```
vpaddq   -160(%r8), %ymm3, %ymm4
```

```
vpaddq   -128(%r8), %ymm4, %ymm5
```

```
vpaddq   -96(%r8), %ymm5, %ymm6
```

```
vpaddq   -64(%r8), %ymm6, %ymm7
```

```
vpaddq   -32(%r8), %ymm7, %ymm0
```

```
cmpq     %rcx, %r8
```

```
jne      .L4
```

```
... /* approx 20 lines omitted */
```

```
ret
```


example assembly (clang 9.0 -O -march=skylake)

sum:

```
testl    %esi, %esi
```

```
... /* approx 35 lines omitted */
```

```
the_loop: /* loop unrolled + multiple accumulators + instrs that 4 pairs at a time */
```

```
vpaddq   (%rdi,%rsi,8), %ymm0, %ymm0
```

```
vpaddq   32(%rdi,%rsi,8), %ymm1, %ymm1
```

```
vpaddq   64(%rdi,%rsi,8), %ymm2, %ymm2
```

```
vpaddq   96(%rdi,%rsi,8), %ymm3, %ymm3
```

```
vpaddq   128(%rdi,%rsi,8), %ymm0, %ymm0
```

```
vpaddq   160(%rdi,%rsi,8), %ymm1, %ymm1
```

```
vpaddq   192(%rdi,%rsi,8), %ymm2, %ymm2
```

```
vpaddq   224(%rdi,%rsi,8), %ymm3, %ymm3
```

```
vpaddq   256(%rdi,%rsi,8), %ymm0, %ymm0
```

```
vpaddq   288(%rdi,%rsi,8), %ymm1, %ymm1
```

```
vpaddq   320(%rdi,%rsi,8), %ymm2, %ymm2
```

```
vpaddq   352(%rdi,%rsi,8), %ymm3, %ymm3
```

```
vpaddq   384(%rdi,%rsi,8), %ymm0, %ymm0
```

```
vpaddq   416(%rdi,%rsi,8), %ymm1, %ymm1
```

```
vpaddq   448(%rdi,%rsi,8), %ymm2, %ymm2
```

```
vpaddq   480(%rdi,%rsi,8), %ymm3, %ymm3
```

```
addq     $64, %rsi
```

```
addq     $4, %rax
```

```
jne the_loop
```

optimizing compilers

these usually make your code fast

often not done by default

compilers and humans are good at **different kinds** of optimizations

aliasing problems with cache blocking

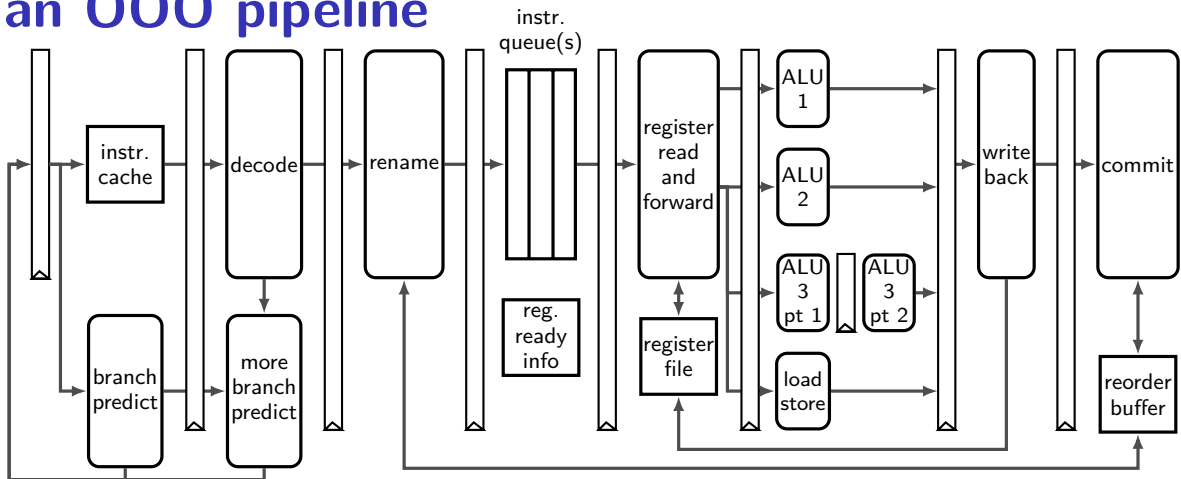
```
for (int k = 0; k < N; k++) {  
  for (int i = 0; i < N; i += 2) {  
    for (int j = 0; j < N; j += 2) {  
      C[(i+0)*N + j+0] += A[i*N+k] * B[k*N+j];  
      C[(i+1)*N + j+0] += A[(i+1)*N+k] * B[k*N+j];  
      C[(i+0)*N + j+1] += A[i*N+k] * B[k*N+j+1];  
      C[(i+1)*N + j+1] += A[(i+1)*N+k]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

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

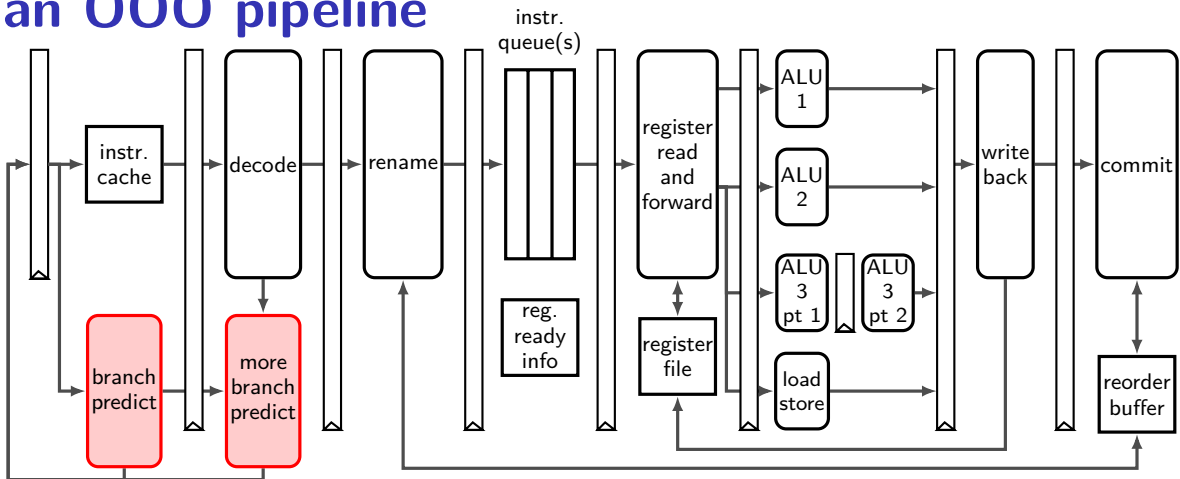
“register blocking”

```
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 = B[k*N + j+0];  
            float Bkj1 = B[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;  
        }  
    }  
}
```

an OOO pipeline

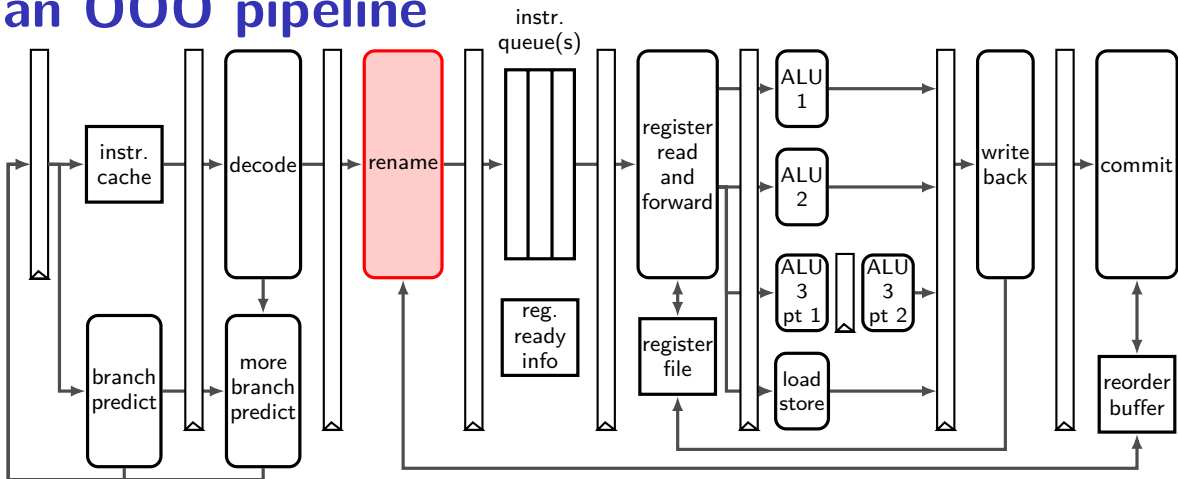


an OOO pipeline



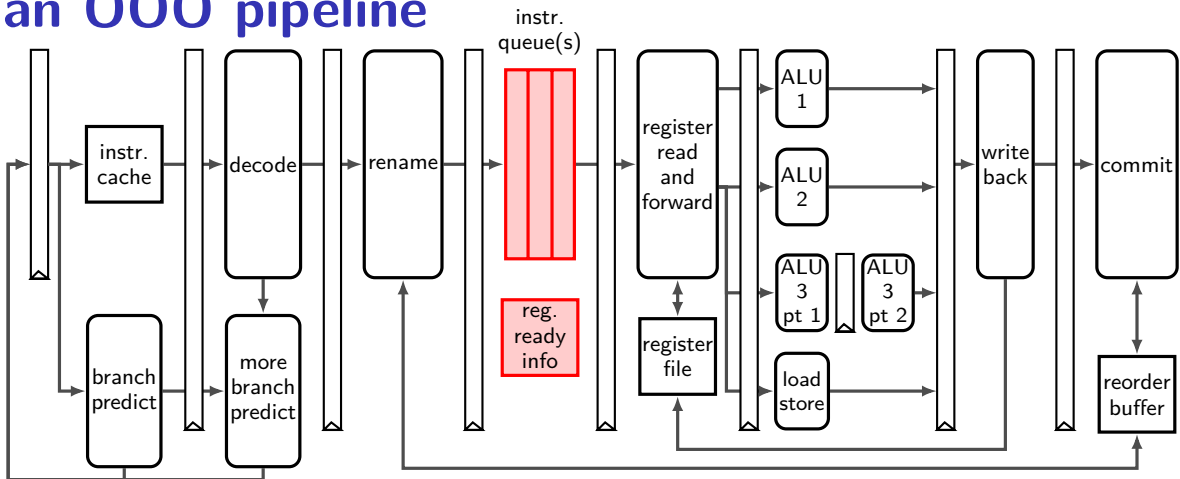
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline



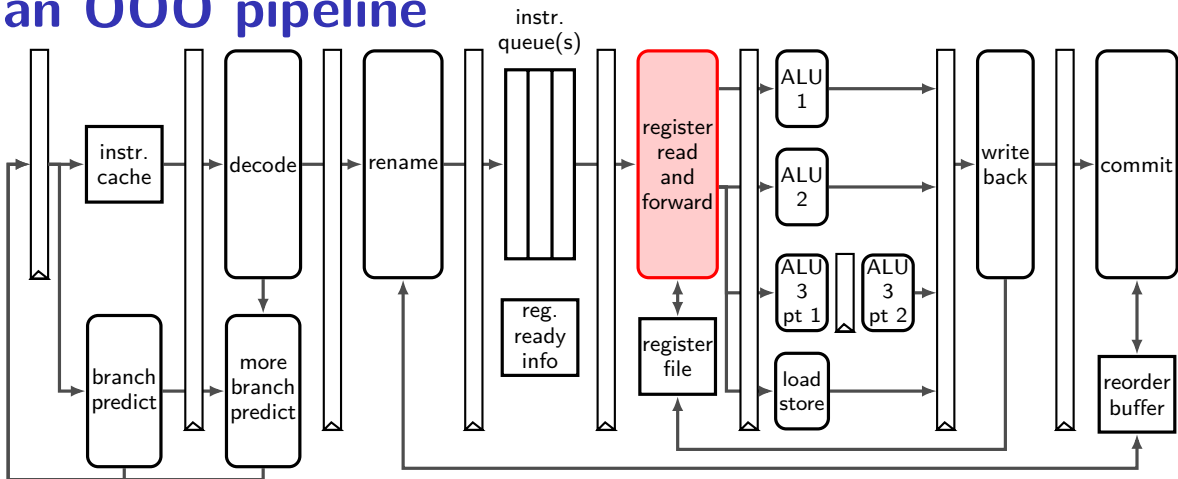
register renaming done here
stage needs to keep mapping from architectural to physical names

an OOO pipeline



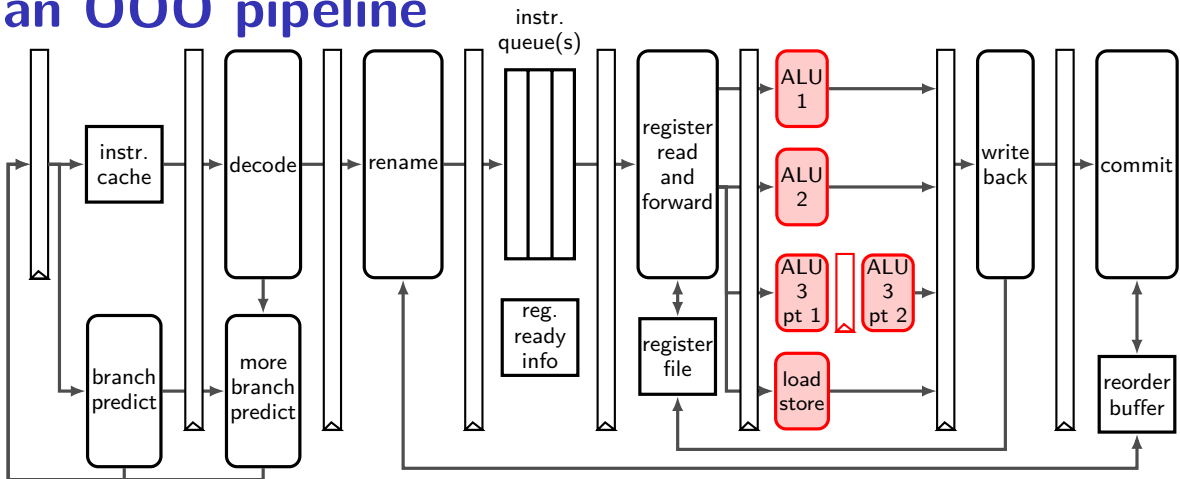
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (issue = start executing)

an OOO pipeline



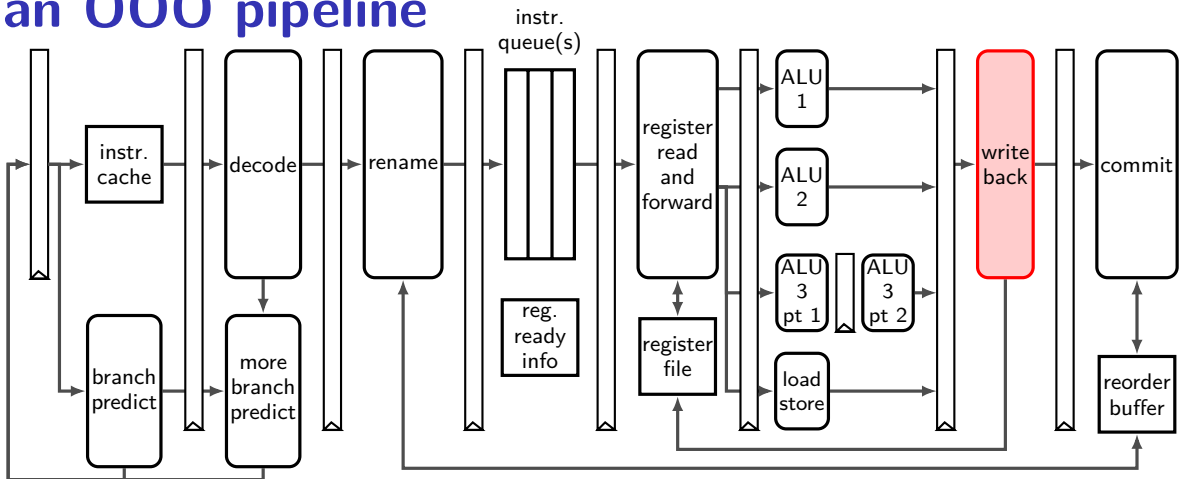
read from much larger register file and handle forwarding
register file: typically read 6+ registers at a time
(extra data paths wires for forwarding not shown)

an OOO pipeline



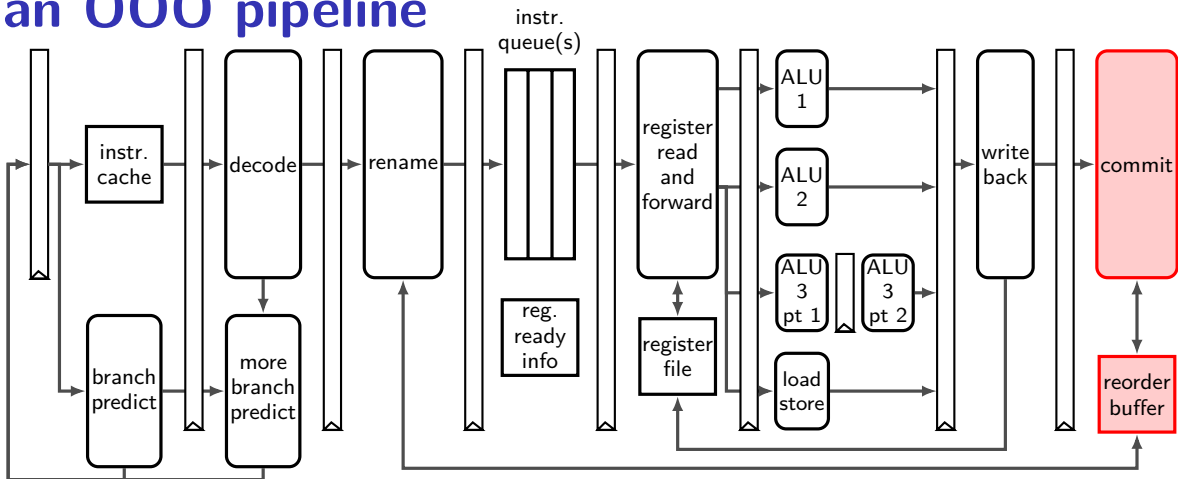
many *execution units* actually do math or memory load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline



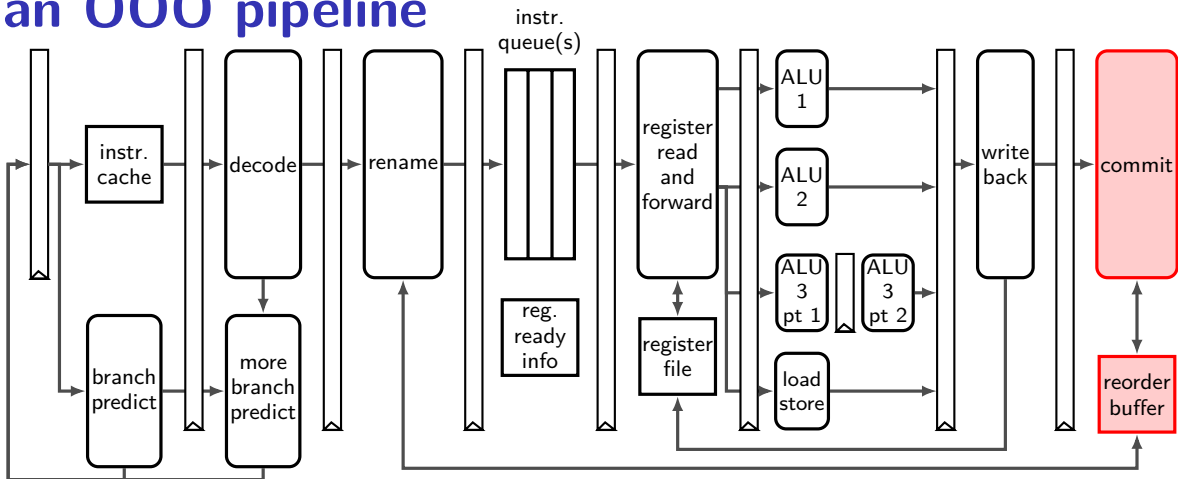
writeback results to physical registers
register file: typically support writing 3+ registers at a time

an OOO pipeline



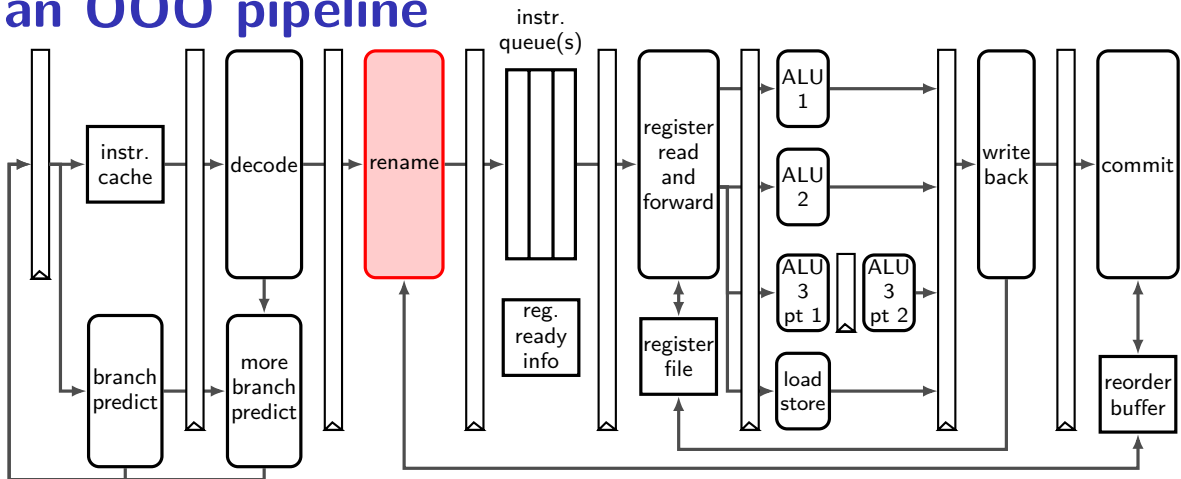
new commit (sometimes *retire*) stage finalizes instruction figures out when physical registers can be reused again

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
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 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
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
...	...

list of available physical registers
added to as instructions finish

free reg list

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

register renaming state

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

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	renamed
add %r10, %r8	add %x19, %x13 → %x18
add %r11, %r8	
add %r12, %r8	

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	renamed
add %r10, %r8	add %x19, %x13 → %x18
add %r11, %r8	add %x07, %x18 → %x20
add %r12, %r8	

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

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

%x21
%x23
%x24
...

register renaming example (2)

original

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

renamed

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

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

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

renamed

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

arch → phys
register map

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

free
regs

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

register renaming example (2)

original

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

renamed

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

arch → phys
register map

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

free
regs

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

register renaming example (2)

original

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

renamed

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

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