last time (1)

single-cycle CPU review

- one possible CPU design that runs one instruction per cycle
- PC changes at beginning of cycle, cascades for other components to operation

pipelining idea

- laundry analogy
- opportunity: in single-cycle design, most components mostly idle
- assembly-line: step 1 of instr 1 then {step 2 of instr 1 + step 1 of instr 2}
- then {step 3 of instr 1 + step 2 of instr 2 + step 1 of instr 3} then ...
- adding registers to store values for each *stage*

last time (2)

pipelining limits

- cycle time determined by slowest stage
- time taken by new registers
- uneven split of stages
- doubling pipeline stages != half cycle time

data hazards

- solving by changing ISA?
- solving by *stalling* (insert nops)

anonymous feedback

"Why are we using kytos for the autograders when we can use gradescope? Having to wait over an hour for an "autograder" is unacceptable. I have never had gradescope take more than 2 minutes with any autograded submission and I've never had kytos take less than 5 minutes, often it taking an hour/running overnight. I understand there are conveniences with kytos (the cumulative performance), but if we are truly doing assignments that are autograded and provide instant feedback, we shouldn't need to spend several hours waiting. I'll propose some potential solutions that could remedy the problem: 1. If people are missing just 1-2 tests on the autograder, just bump their score up to 100 so they don't clog up the queue, 2. Extend the assignment for each person individually based on how long they've waited for their submission to be graded (maybe up to a certain number of submissions), 3. Use Gradescope!"

- life 'test your code' section which was pretty complete
  - given basically this assignment with no autograder feedback before
- from talking to other faculty, gradescope is not always as fast as you think (though, yes, when the queue is empty, it starts things sooner…)
- probably would have been better if autograder gave up on submissions that timeout a lot faster (which would've also been a problem under gradescope)

some notes on the lab (1)

map/reduce division

I expected one loop

- map strategy determines which items you do log(...) operation for
- reduce strategy determines how you do += to answer

okay to two loops, but much harder to make efficient

some notes on the lab (2)

why was atomic update reduce strategy slow?

- processors need to take turns having accumulator (`answer`)
- lots of synchronization time + mostly one thread works at a time

why was task queue strategy slow?

- processors need to take turns grabbing index to use next
- lots of synchronization time

what about few-to-many reduction with array?

- `results[thread_id] +=`
- problem: multiple thread values are in *same cache block*
- cores need to take turns having the block in their cache to write
- workaround: make results array be more spread out
- called "false sharing"

upcoming lab/HW logistics

addq processor: data hazard

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

addq processor: data hazard

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

data hazard

```
addq %r8, %r9  // (1)
addq %r9, %r8  // (2)
```

step#	pipeline implementation	ISA specification
1	read r8, r9 for (1)	read r8, r9 for (1)
2	read r9, r8 for (2)	write r9 for (1)
3	write r9 for (1)	read r9, r8 for (2)
4	write r8 for (2)	write r8 ror (2)

pipeline reads older value…

instead of value ISA says was just written

data hazard compiler solution

```
addq %r8, %r9
nop
nop
addq %r9, %r8
```

one solution: change the ISA

- all addqs take effect three instructions later
- (assuming can read register value while it is being written back)

make it compiler's job

problem: recompile everytime processor changes?

data hazard hardware solution

```
addq %r8, %r9
// hardware inserts: nop
// hardware inserts: nop
addq %r9, %r8
```

how about hardware add nops?

called stalling

extra logic:

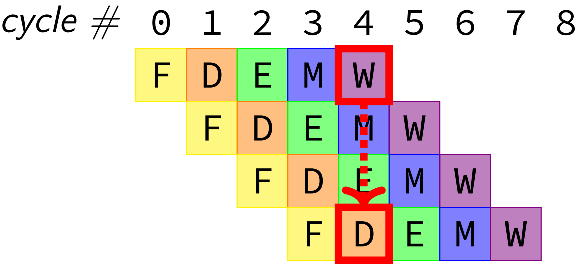
- sometimes don't change PC
- sometimes put do-nothing values in pipeline registers

stalling/nop pipeline diagram (1)

	cycle # 0	1	2	3	4	5	6	7	8
`add %r8, %r9`	F	D	E	M	W				
`(nop)`		F	D	E	M	W			
`(nop)`			F	D	E	M	W		
`addq %r9, %r8`				F	D	E	M	W	

stalling/nop pipeline diagram (1)

```
add %r8, %r9
(nop)
(nop)
addq %r9, %r8
```



assumption:
if writing register value
register file will return that value for reads

not actually way register file worked in single-cycle CPU
(e.g. can read old %r9 while writing new %r9)

stalling/nop pipeline diagram (2)

	cycle # 0	1	2	3	4	5	6	7	8	
`add %r8, %r9`	F	D	E	M	W					
`(nop)`		F	D	E	M	W				
`(nop)`			F	D	E	M	W			
`(nop)`				F	D	E	M	W		
`addq %r9, %r8`						F	D	E	M	W

stalling/nop pipeline diagram (2)

	cycle #	0	1	2	3	4	5	6	7	8	
`add %r8, %r9`		F	D	E	M	W					
`(nop)`			F	D	E	M	W				
`(nop)`				F	D	E	M	W			
`(nop)`					F	D	E	M	W		
`addq %r9, %r8`							F	D	E	M	W

if we didn't modify the register file, we'd need an extra cycle

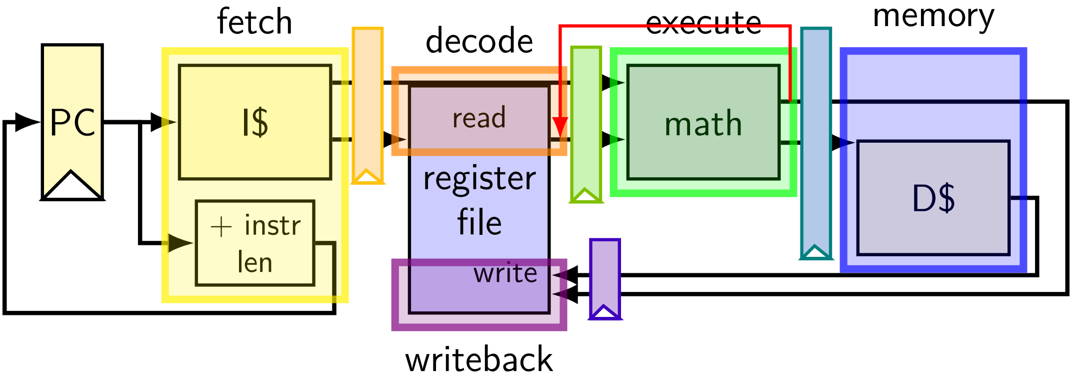
opportunity

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: addq %r9, %r8
...
```

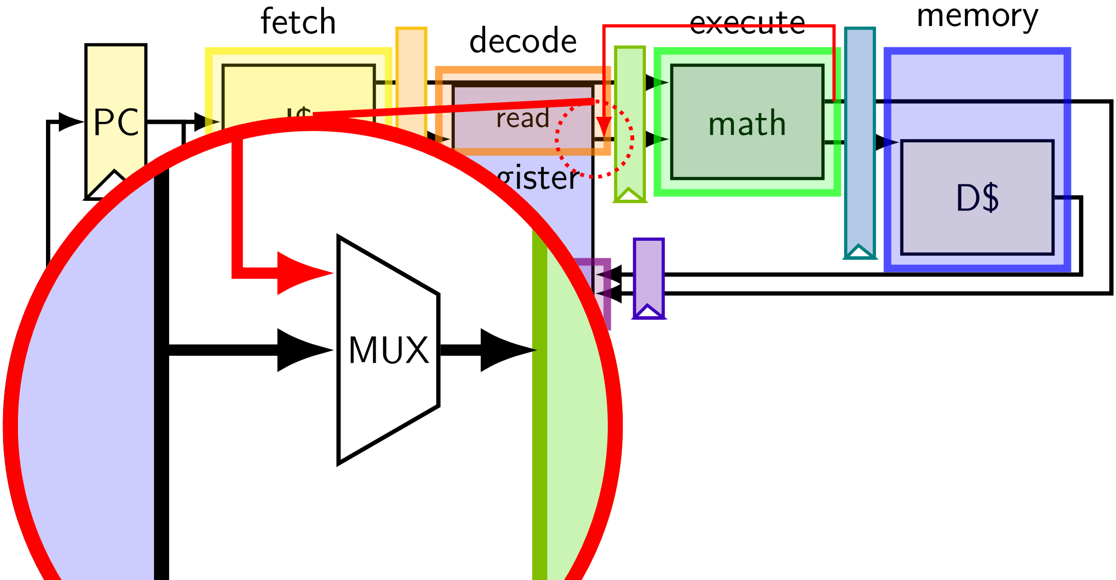
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



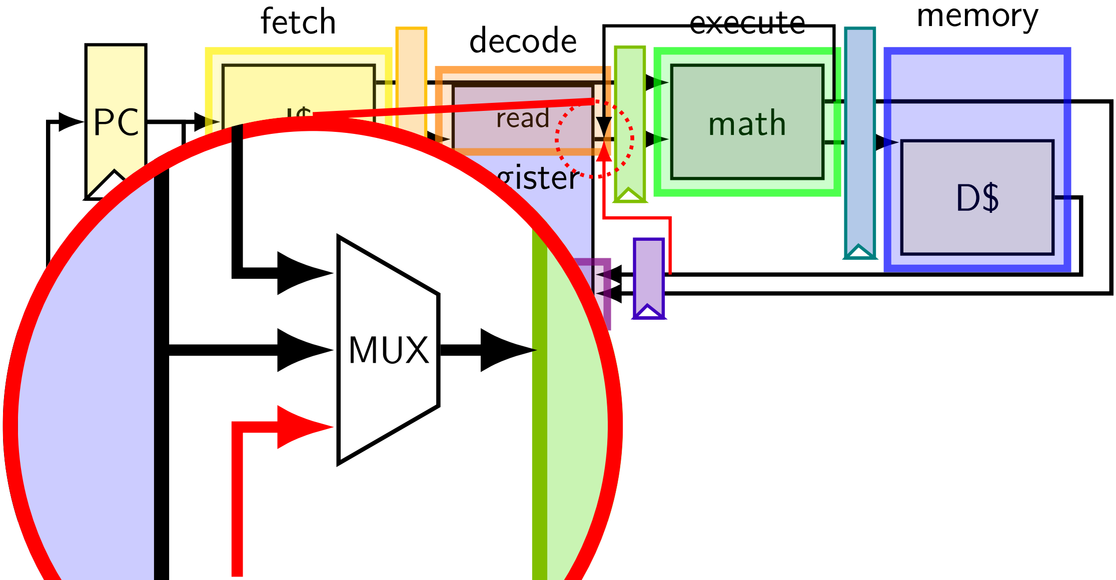
opportunity 2

```
// initially %r8 = 800,
//           %r9 = 900, etc.
0x0: addq %r8, %r9
0x2: nop
0x3: addq %r9, %r8
...
```

	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2	0x3	---	---	800	900	9				
3		9	8	---	---	---	1700	9		
4				900	800	8	---	---	1700	9
5							1700	9	---	---
6									1700	9

should be 1700

exploiting the opportunity



exercise: forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r8, %r10`		F	D	E	M	W			
`xorq %r8, %r9`			F	D	E	M	W		
`andq %r9, %r8`				F	D	E	M	W	

in subq, %r8 is ________ addq.

in xorq, %r9 is ________ addq.

in andq, %r9 is ________ addq.

in andq, %r9 is ________ xorq.

A: not forwarded from

B-D: forwarded to decode from {execute,memory,writeback} stage of

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r9, %r11`			F	D	E	M	W			
`movq 4(%r11), %r10`				F	D	E	M	W		
`movq %r9, 8(%r11)`					F	D	E	M	W	
`xorq %r10, %r9`						F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`movq 4(%r11), %r10`			F	D	E	M	W		
`movq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`movq 4(%r11), %r10`			F	D	E	M	W		
`movq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r9, %r11`			F	D	E	M	W			
`movq 4(%r11), %r10`				F	D	E	M	W		
`movq %r9, 8(%r11)`					F	D	E	M	W	
`xorq %r10, %r9`						F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`movq 4(%r11), %r10`			F	D	E	M	W		
`movq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

```
addq %r8, %r9
subq %r9, %r11
movq 4(%r11), %r10
movq %r9, 8(%r11)
xorq %r10, %r9
```

cycle #	0	1	2	3	4	5	6	7	8
addq %r8, %r9	F	D	E	M	W				
subq %r9, %r11		F	D	E	M	W			
movq 4(%r11), %r10			F	D	E	M	W		
movq %r9, 8(%r11)				F	D	E	M	W	
xorq %r10, %r9					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`movq 4(%r11), %r10`			F	D	E	M	W		
`movq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r9, %r11`			F	D	E	M	W			
`movq 4(%r11), %r10`				F	D	E	M	W		
`movq %r9, 8(%r11)`					F	D	E	M	W	
`xorq %r10, %r9`						F	D	E	M	W

some forwarding paths

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`		F	D	E	M	W				
`subq %r9, %r11`			F	D	E	M	W			
`movq 4(%r11), %r10`				F	D	E	M	W		
`movq %r9, 8(%r11)`					F	D	E	M	W	
`xorq %r10, %r9`						F	D	E	M	W

multiple forwarding paths (1)

```
addq %r10, %r8
addq %r11, %r8
addq %r12, %r8
```

cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8	F	D	E	M	W				
addq %r11, %r8		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

multiple forwarding paths (1)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r8`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding paths (2)

	cycle # 0	1	2	3	4	5	6	7	8
addq %r10, %r8	F	D	E	M	W				
addq %r11, %r12		F	D	E	M	W			
addq %r12, %r8			F	D	E	M	W		

multiple forwarding paths (2)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r12`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding paths (2)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r12`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

unsolved problem

cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			

combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

unsolved problem

cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			
`subq %rbx, %rcx`		F	D	D	E	M	W		

stall

combine stalling and forwarding to resolve hazard

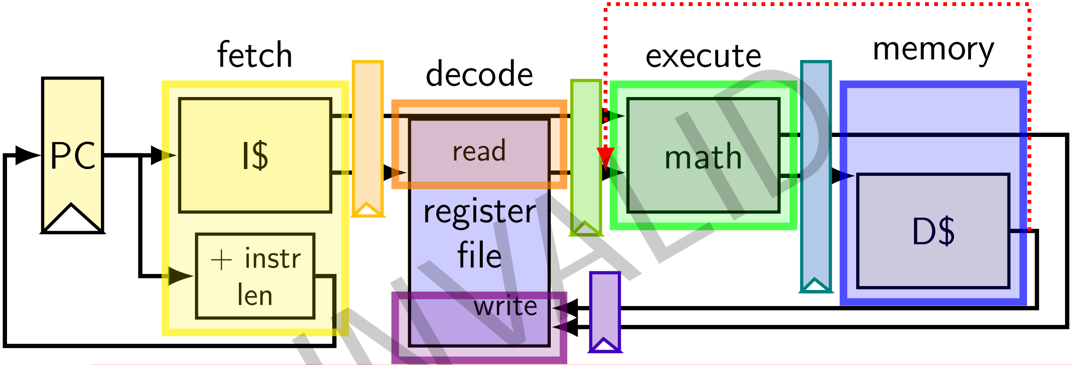
assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

solveable problem

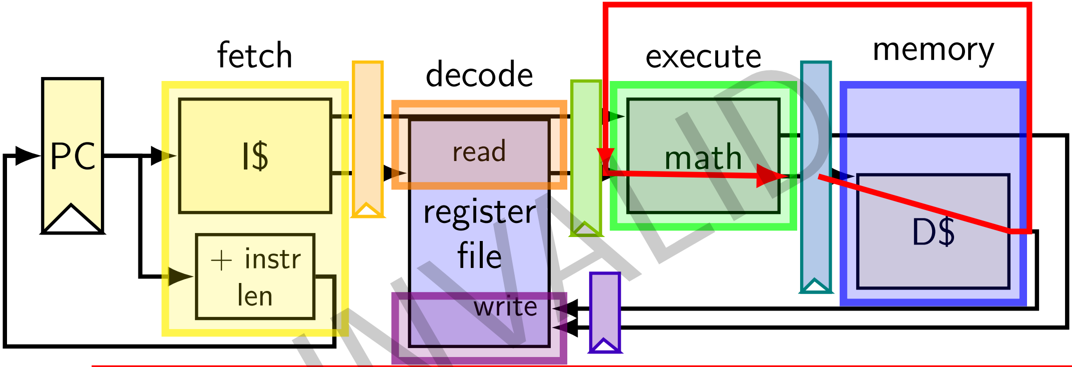
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`movq %rbx, 0(%rcx)`		F	D	E	M	W			

why can't we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

why can’t we...



clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

hazards versus dependencies

dependency — X needs result of instruction Y?

- has potential for being messed up by pipeline
- (since part of X may run before Y finishes)

hazard — will it not work in some pipeline?

- before extra work is done to “resolve” hazards
- multiple kinds: so far, *data hazards*

ex.: dependencies and hazards (1)

```
addq    %rax,   %rbx
subq    %rax,   %rcx
movq    $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax,   %rbx
subq    %rax,   %rcx
movq    $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax,   %rbx
subq    %rax,   %rcx
movq    $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq   %rax,   %rbx
subq   %rax,   %rcx
movq   $100,   %rcx
addq   %rcx,   %r10
addq   %rbx,   %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage    // 5 stage
addq %rax, %r8      //            // W
subq %rax, %r9      // W          // M
xorq %rax, %r10     // EM         // E
andq %r8,  %r11     // D          // D
```

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                     // 4 stage   // 5 stage
addq %rax, %r8       //           // W
subq %rax, %r9       // W         // M
xorq %rax, %r10      // EM        // E
andq %r8,  %r11      // D         // D
```

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                     // 4 stage   // 5 stage
addq %rax, %r8       //           // W
subq %rax, %r9       // W         // M
xorq %rax, %r10      // EM        // E
andq %r8,  %r11      // D         // D
```

more hazards with more pipeline stages

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

result only available near end of second execute stage

where does forwarding, stalls occur?

	cycle #	0	1	2	3	4	5	6	7	8
(1)	`addq %rcx, %r9`	F	D	E1	E2	M	W			
(2)	`addq %r9, %rbx`									
(3)	`addq %rax, %r9`									
(4)	`movq %r9, (%rbx)`									
(5)	`movq %rcx, %r9`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

	0	1	2	3	4	5	6	7	8
cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`									
`addq %rax, %r9`									
`movq %r9, (%rbx)`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	E1	E2	M	W		
addq %rax, %r									
movq %r9, (%rbx)				F	D	E1	E2	M	W

r9 not available yet — can't forward here
so try stalling in addq's decode...

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`	F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	D	E1	E2	M	W		
`addq %rax, %r9`										
`addq %rax, %r9`			F	F	D	E1	E2	M	W	
`movq %r9, (%rbx)`				F	D	E1	E2	M	W	
`movq %r9, (%rbx)`					F	D	E1	E2	M	W

after stalling once, now we can forward

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
addq %rcx, %r9	F	D	E1	E2	M	W				
addq %r9, %rbx		F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W		
addq %rax, %r9			F	F	D	E1	E2	M	W	
movq %r9, (%rbx)				F	D	E1	E2	M	W	
movq %r9, (%rbx)					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8		
addq %rcx, %r9	F	D	E1	E2	M	W					
addq %r9, %rbx		F	D	E1	E2	M	W				
addq %r9, %rbx		F	D	D	E1	E2	M	W			
addq %rax, %r9			F	D	E1	E2	M	W			
addq %rax, %r9			F	F	D	E1	E2	M	W		
movq %r9, (%rbx)				F	D	E1	E2	M	W		
movq %r9, (%rbx)					F	D	E1	E2	M	W	
movq %rcx, %r9						F	D	E1	E2	M	W

control hazard

```
0x00: cmpq %r8, %r9
0x08: je   0xFFFF
0x10: addq %r10, %r11
```

	fetch	fetch→decode		decode→execute		execute→writeb	execute→writeback		…
cycle	PC	rA	rB	R[rA]	R[rB]	result	…	…	…
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

control hazard

```
0x00: cmpq %r8, %r9
0x08: je   0xFFFF
0x10: addq %r10, %r11
```

	fetch	fetch→decode		decode→execute		execute→writeb	execute→writeback		…
cycle	PC	rA	rB	R[rA]	R[rB]	result	…	…	…
0	0x0								
1	0x8	8	9						
2	???	---	---	800	900				
3	???	---	---	---	---	less than			

0xFFFF if $R[8] = R[9]$; 0x10 otherwise

jXX: stalling?

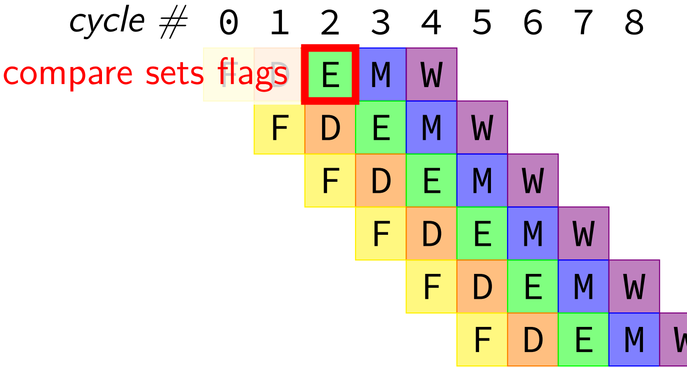
```
cmpq %r8, %r9
jne LABEL          // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

	cycle # 0	1	2	3	4	5	6	7	8
cmpq %r8, %r9	F	D	E	M	W				
jne LABEL		F	D	E	M	W			
(do nothing)			F	D	E	M	W		
(do nothing)				F	D	E	M	W	
xorq %r10, %r11					F	D	E	M	W
movq %r11, 0(%r12)						F	D	E	M
...									

jXX: stalling?

```
cmpq %r8, %r9
jne LABEL          // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

```
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cmpq %r8, %r9

jne LABEL — compute if jump goes to LABEL

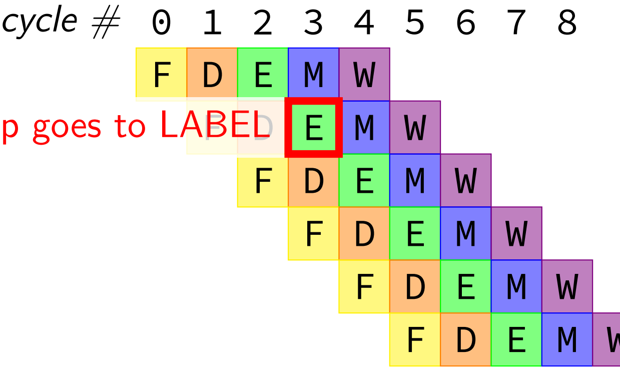
(do nothing)

(do nothing)

xorq %r10, %r11

movq %r11, 0(%r12)

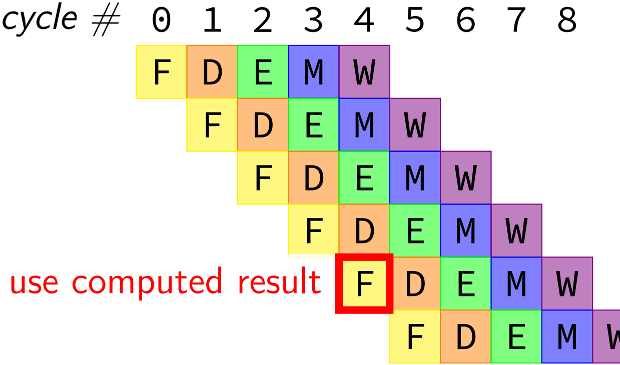
...



jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

```
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



making guesses

```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

speculate (guess): jne won't go to LABEL

right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

cycle #	0	1	2	3	4	5	6	7	8
`cmpq %r8, %r9`	F	D	E	M	W				
`jne LABEL`		F	D	E	M	W			
`xorq %r10, %r11`			F	D	E	M	W		
`movq %r11, 0(%r12)`				F	D	E	M	W	
…									

jXX: speculating wrong

cycle #	0	1	2	3	4	5	6	7	8	
`cmpq %r8, %r9`	F	D	E	M	W					
`jne LABEL`		F	D	E	M	W				
`xorq %r10, %r11`			F	D						
`(inserted nop)`					E	M	W			
`movq %r11, 0(%r12)`				F						
`(inserted nop)`					D	E	M	W		
`LABEL: addq %r8, %r9`					F	D	E	M	W	
`imul %r13, %r14`						F	D	E	M	W
…										

jXX: speculating wrong

cycle #	0	1	2	3	4	5	6	7	8	
`cmpq %r8, %r9`	F	D	E	M	W					
`jne LABEL`		F	D	E	M	W				
`xorq %r10, %r11`			F	D	instruction “squashed”					
(inserted nop)					E	M	W			
`movq %r11, 0(%r12)`				F	instruction “squashed”					
(inserted nop)					D	E	M	W		
`LABEL: addq %r8, %r9`					F	D	E	M	W	
`imul %r13, %r14`						F	D	E	M	W
…										

“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

more complicated pipelines: replace written values in cache/registers/etc.

backup slides

modifying cache blocks in parallel

cache coherency works on cache blocks

but typical memory access — less than cache block
- e.g. one 4-byte array element in 64-byte cache block

what if two processors modify different parts same cache block?
- 4-byte writes to 64-byte cache block

cache coherency — write instructions happen one at a time:
- processor 'locks' 64-byte cache block, fetching latest version
- processor updates 4 bytes of 64-byte cache block
- later, processor might give up cache block

modifying things in parallel (code)

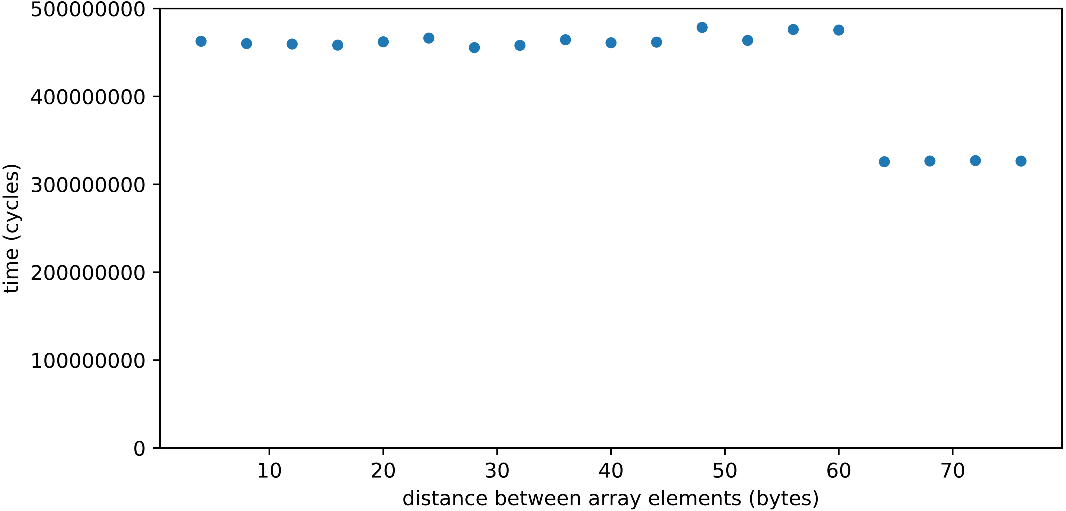
```
void *sum_up(void *raw_dest) {
    int *dest = (int *) raw_dest;
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {
        *dest += data[i];
    }
}

__attribute__((aligned(4096)))
int array[1024];  /* aligned = address is mult. of 4096 */

void sum_twice(int distance) {
    pthread_t threads[2];
    pthread_create(&threads[0], NULL, sum_up, &array[0]);
    pthread_create(&threads[1], NULL, sum_up, &array[distance]);
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
}
```

performance v. array element gap

(assuming `sum_up` compiled to not omit memory accesses)



false sharing

synchronizing to access two independent things

two parts of same cache block

solution: separate them