

Changelog

17 April 2023: throughput/latency exercise: fix typo of 'support' for 'oppose'

last time

pattern for using monitors

lock(mutex)

while need to wait: cond_wait(cv, mutex)

use shared data

if (others can stop waiting) broadcast/signal cv

unlock(mutex)

counting semaphores — hold non-negative number

up/post — increment

down/wait — wait until positive, decrement

do bookkeeping of a count where 0 = wait

then will naturally wait at right times

transactions — do set of things atomically abstraction

quiz Q1a

```
pthread_mutex_lock(A)
```

```
pthread_mutex_lock(A)
```

likely to hang!

quiz Q1d

append_to_array(A, B)

AND append_to_array(A, C)

AND append_to_array(B, C)

consistent lock order:

A locked then B locked then C locked

no deadlock possible

quiz Q3/4

DequeueBoth: wait while 0 As or 0 Bs

must use `both_available` b/c EnqueueB must signal sometimes

EnqueueA: could cause there to no longer be 0As or 0 Bs

so must signal/broadcast both

can't do `both_available != 0`

`!=` not a condition variable operation

despite the name, no condition involved

quiz Q5/6

```
int SendAndReceiveValue(int thread_id, int value) {
    int other_thread_id = (thread_id + 1) % 3;
    sem_wait(&value_empty[other_thread_id]);
    values[other_thread_id] = value;
    sem_post(&value_ready[other_thread_id]);
    sem_wait(&value_ready[thread_id]);
    int received_value = values[thread_id];
    sem_post(&value_empty[thread_id]);
    return received_value;
}
```

want 1 send to set value before waiting: value_empty = 1

want 1 ready to set value before waiting: value_ready = 1

anonymous feedback (1)

“Some CSO homeworks are very time consuming and difficult. It would be nice if you could add a bonus hw and drop the lowest hw grade. We would really appreciate that :)”

implementing consistency: simple

simplest idea: only one run transaction at a time

implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: undo everything, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: **undo everything**, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

implementing consistency: optimistic

on read: copy version # for value read

on write: record value to be written, but don't write yet

on end transaction:

- acquire locks on everything

- make sure values read haven't been changed since read

if they have changed, just retry transaction

aside: openmp

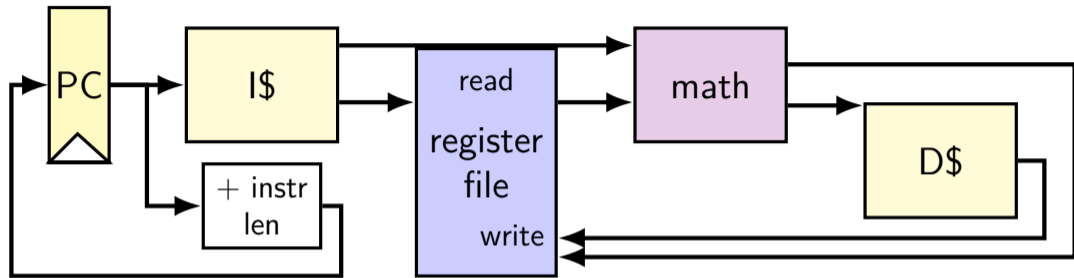
life HW: pattern of dividing up work in loop among multiple threads

alternate API idea: based on automating that:

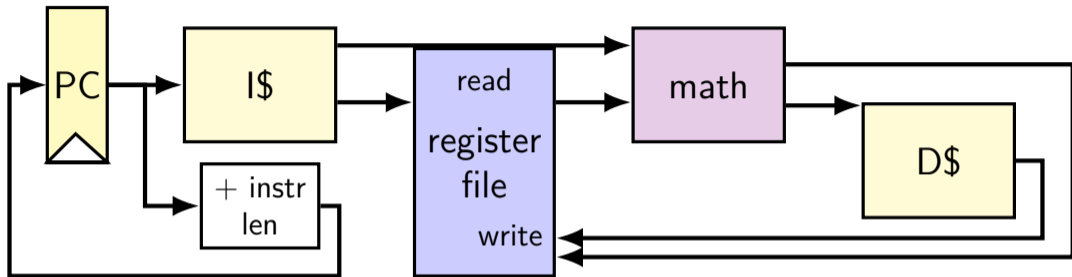
```
#pragma omp parallel for
  for (int i = 0; i < N; ++i) {
    array[i] *= 2;
  }
```

subject of tomorrow's lab

simple CPU



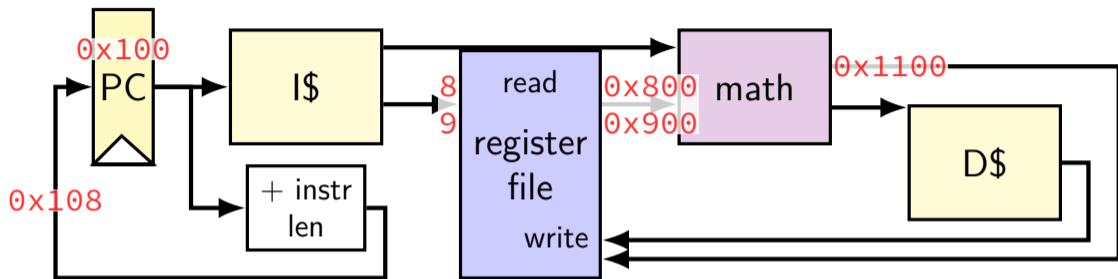
running instructions



```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

```
...
%r8: 0x800
%r9: 0x900
%r10: 0x1000
%r11: 0x1100
...
```

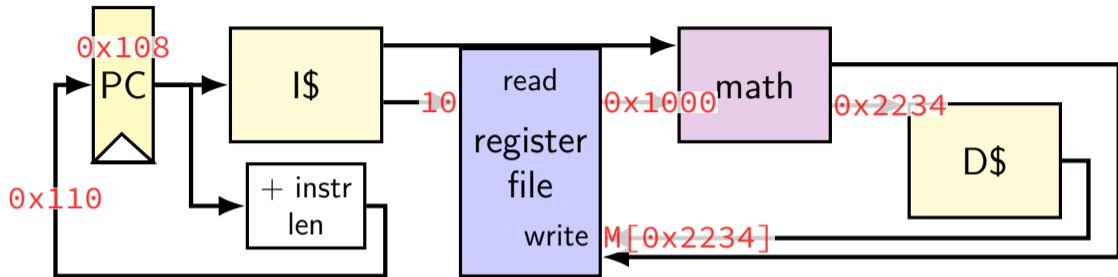
running instructions



```
0x100: addq %r8, %r9  
0x108: movq 0x1234(%r10), %r11
```

```
...  
%r8: 0x800  
%r9: 0x1100  
%r10: 0x1000  
%r11: 0x1100  
...
```

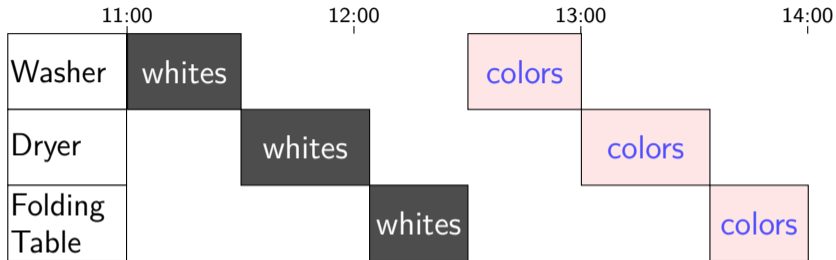

running instructions



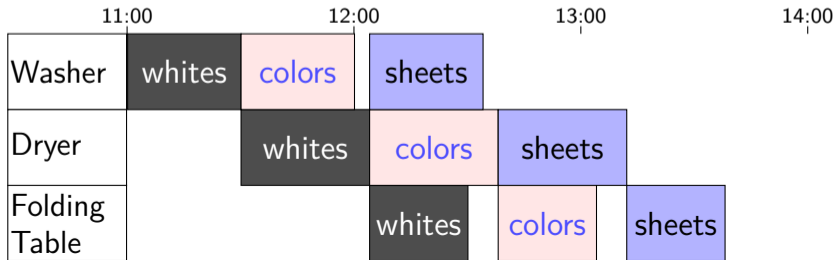
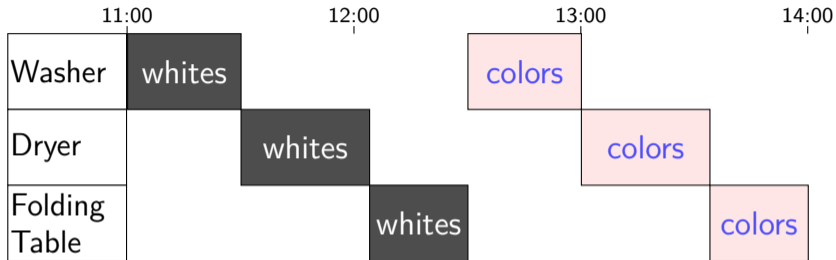
```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

```
...
%r8: 0x800
%r9: 0x1100
%r10: 0x1000
%r11: M[0x2234]
...
```

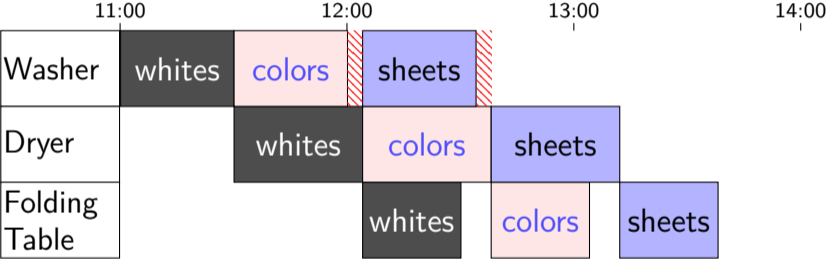
Human pipeline: laundry



Human pipeline: laundry

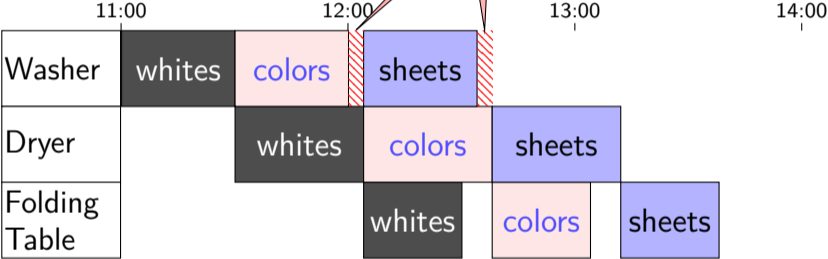


Waste (1)

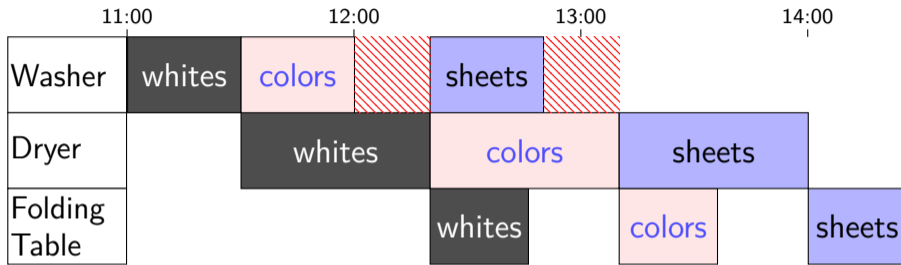


Waste (1)

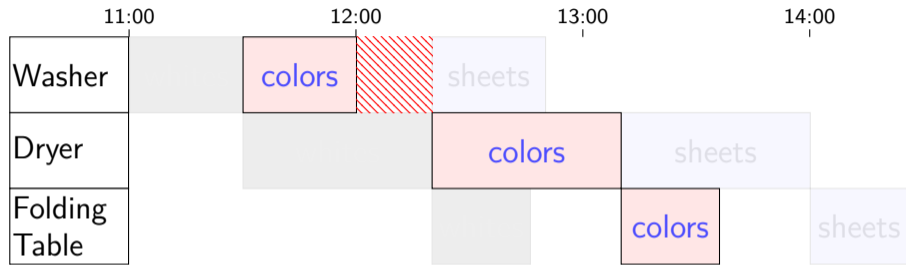
wasted time!



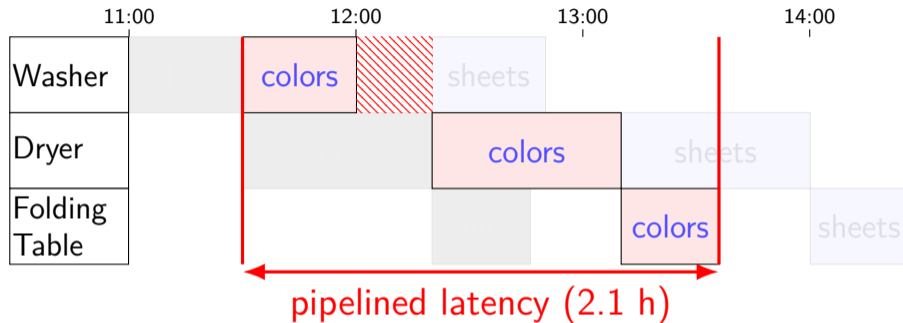
Waste (2)



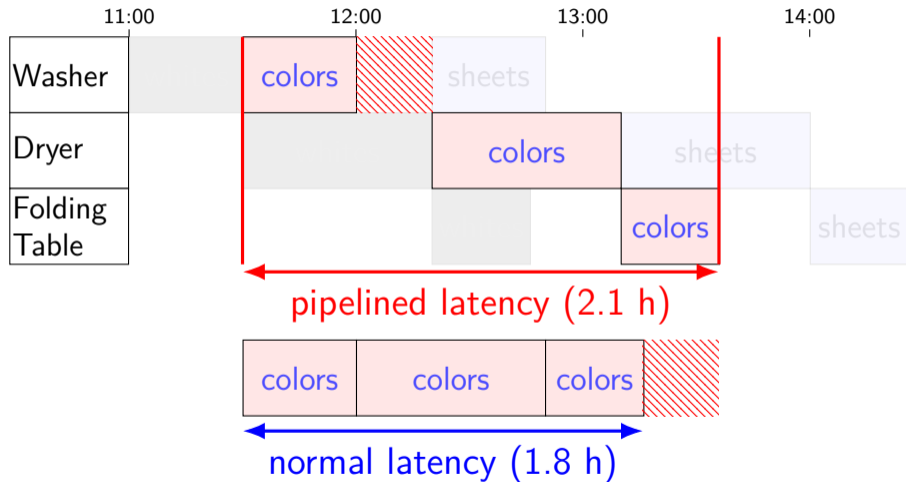
Latency — Time for One



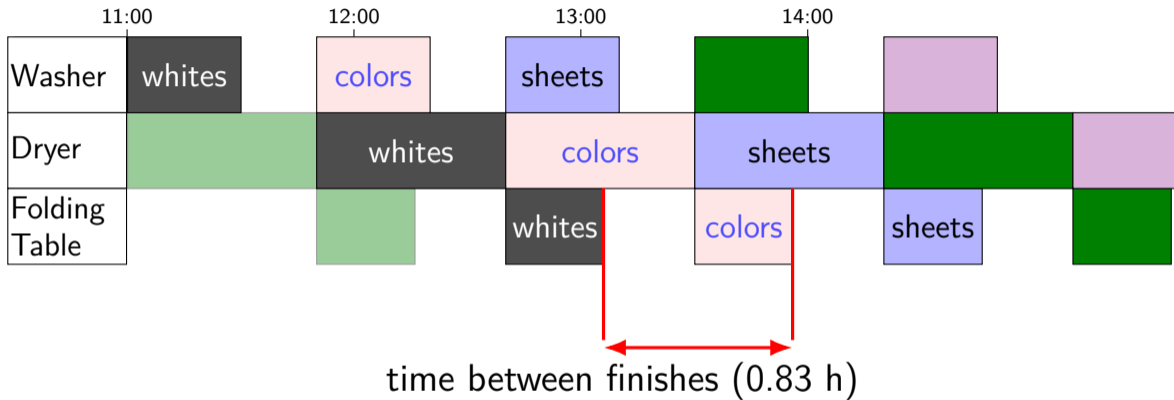
Latency — Time for One



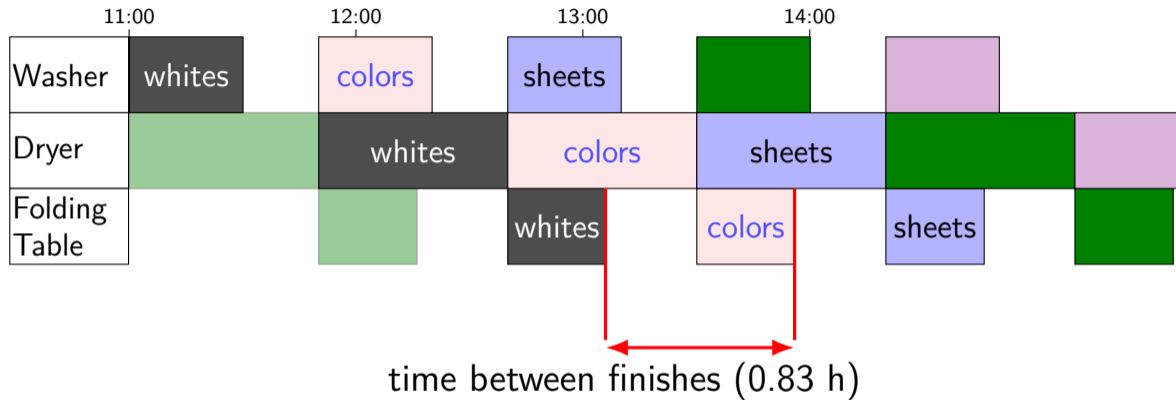
Latency — Time for One



Throughput — Rate of Many

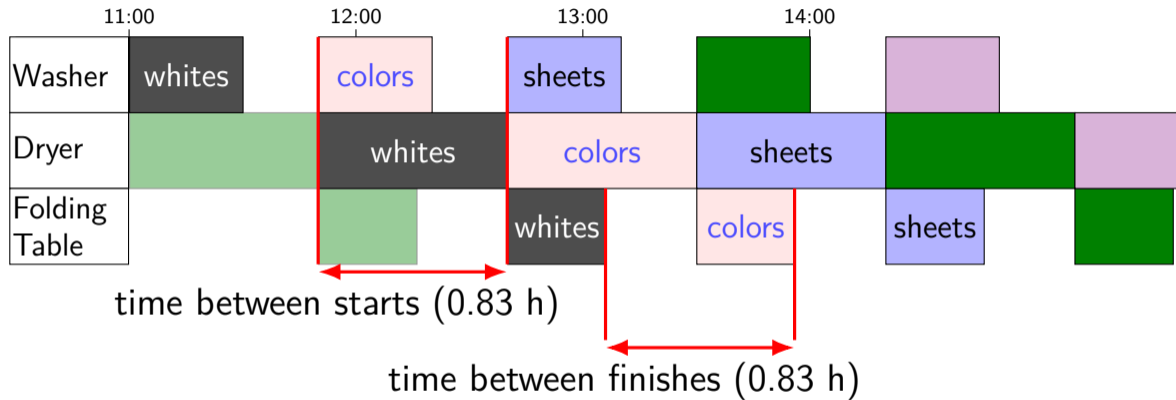


Throughput — Rate of Many



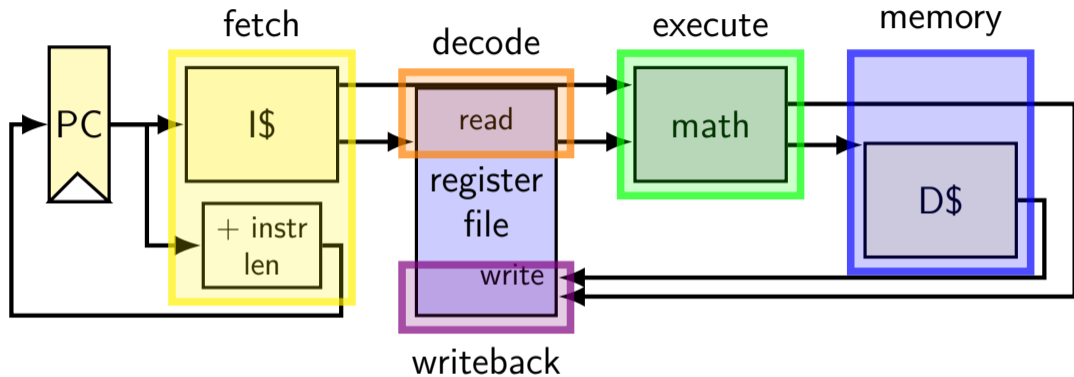
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many



$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

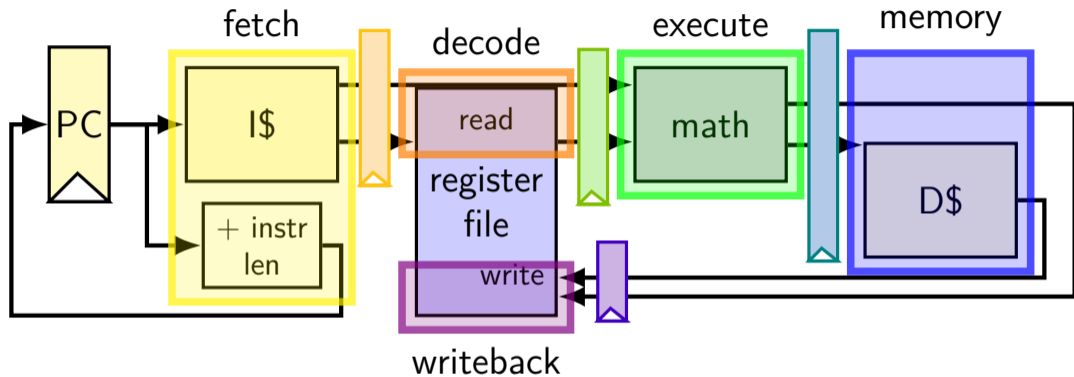
adding stages (one way)



divide running instruction into steps

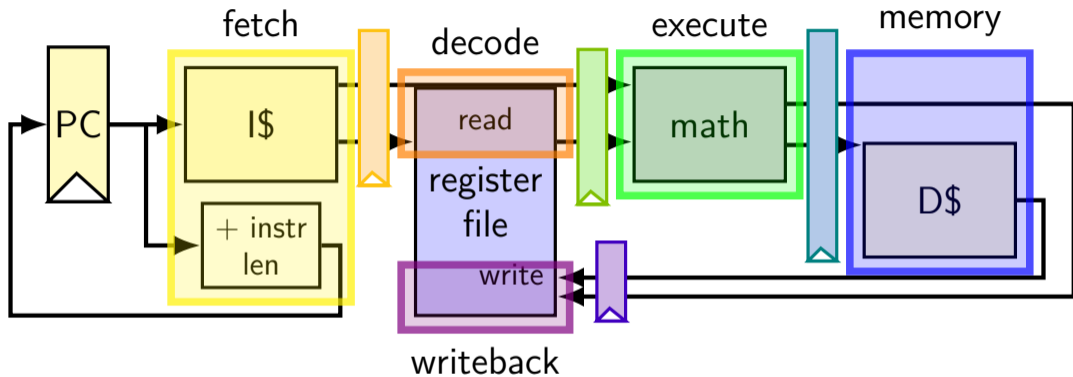
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

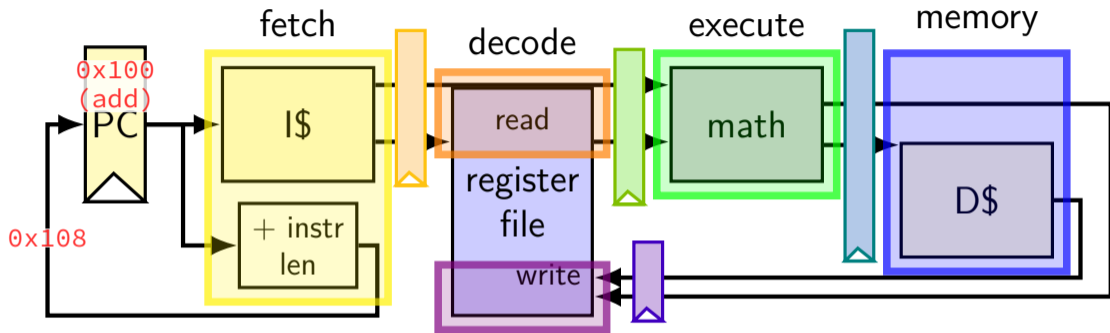


add 'pipeline registers' to hold values from instruction

running some instructions



running some instructions



`0x100: add %r8, %r9`

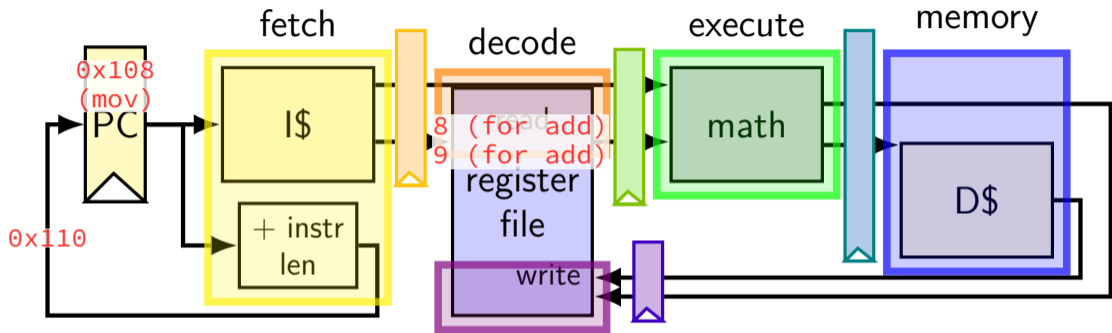
`0x108: mov 0x1234(%r10), %r11`

`0x110: xor %r12, %r13`

writeback cycle #

cycle #	0	1	2	3	4	5	6	7	8
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		

running some instructions

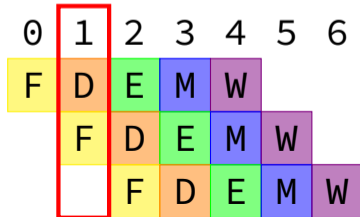


`0x100: add %r8, %r9`

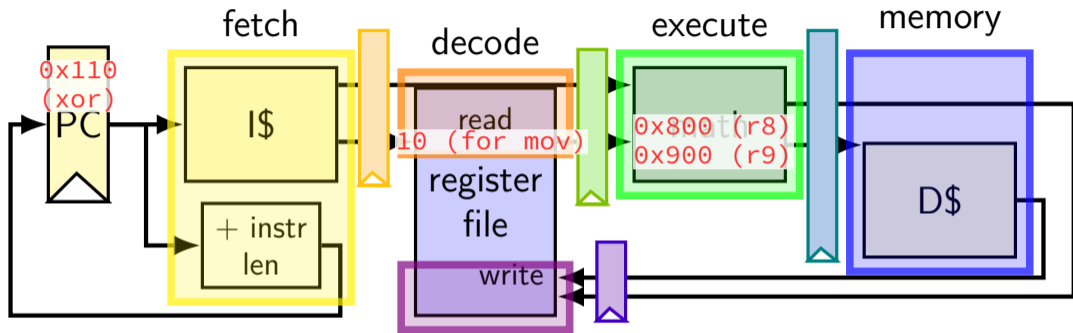
`0x108: mov 0x1234(%r10), %r11`

`0x110: xor %r12, %r13`

writeback cycle # 0 1 2 3 4 5 6 7 8



running some instructions



`0x100: add %r8, %r9`

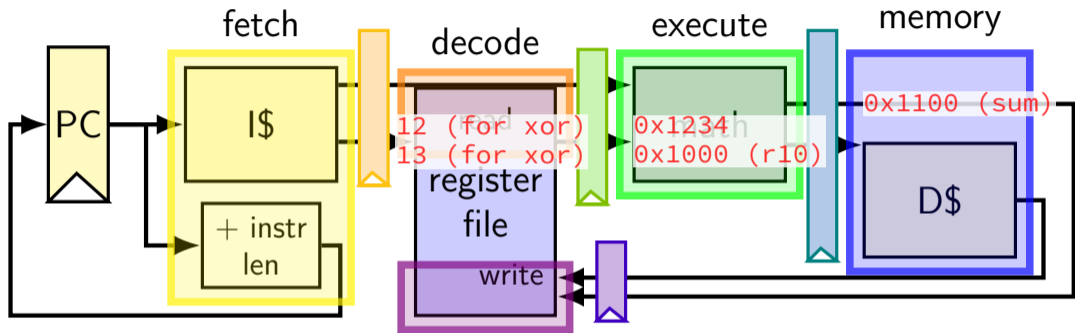
`0x108: mov 0x1234(%r10), %r11`

`0x110: xor %r12, %r13`

writeback cycle #

cycle #	0	1	2	3	4	5	6	7	8
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		

running some instructions

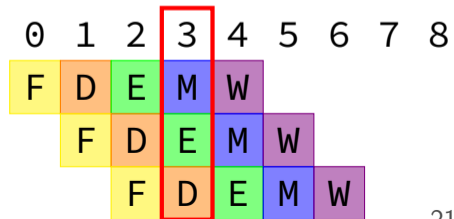


writeback cycle #

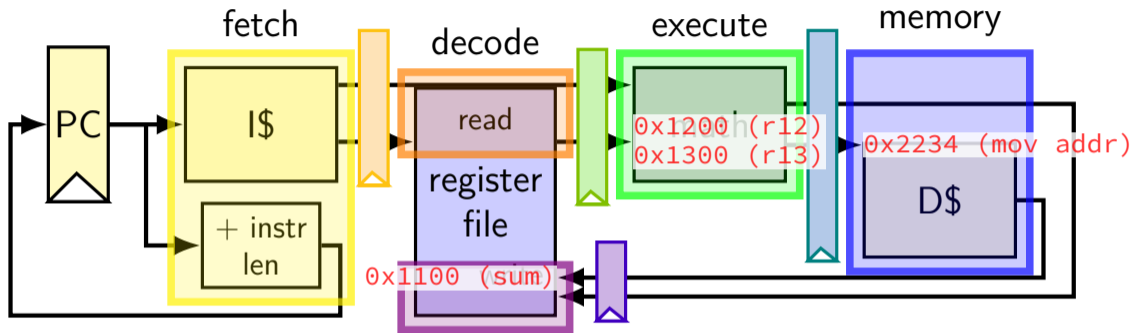
0x100: add %r8, %r9

0x108: mov 0x1234(%r10), %r11

0x110: xor %r12, %r13



running some instructions



0x100: add %r8, %r9

0x108: mov 0x1234(%r10), %r11

0x110: xor %r12, %r13

cycle #

	0	1	2	3	4	5	6	7	8
0x100	F	D	E	M	W				
0x108		F	D	E	M	W			
0x110			F	D	E	M	W		

why registers?

example: fetch/decode

need to store current instruction somewhere ...while fetching next one

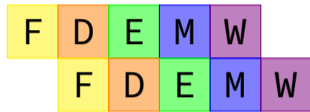
exercise: throughput/latency (1)

0x100: add %r8, %r9

0x108: mov 0x1234(%r10), %r11

0x110: ...

cycle # 0 1 2 3 4 5 6 7 8



...

suppose cycle time is 500 ps

exercise: latency of one instruction?

- A. 100 ps B. 500 ps C. 2000 ps D. 2500 ps E. something else

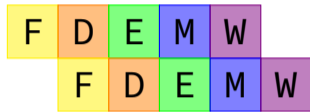
exercise: throughput/latency (1)

0x100: add %r8, %r9

0x108: mov 0x1234(%r10), %r11

0x110: ...

cycle # 0 1 2 3 4 5 6 7 8



...

suppose cycle time is 500 ps

exercise: latency of one instruction?

- A. 100 ps B. 500 ps C. 2000 ps D. 2500 ps E. something else

exercise: throughput overall?

- A. 1 instr/100 ps B. 1 instr/500 ps C. 1 instr/2000ps D. 1 instr/2500 ps
E. something else

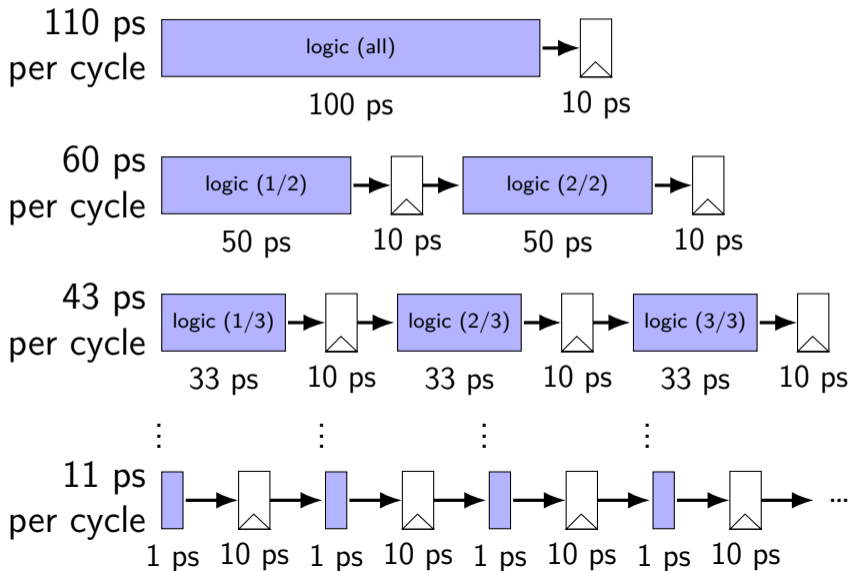
exercise: throughput/latency (2)

	<i>cycle #</i>	0	1	2	3	4
0x100: add %r8, %r9		F	D	E	M	W
0x108: mov 0x1234(%r10), %r11			F	D	E	M
0x110:	

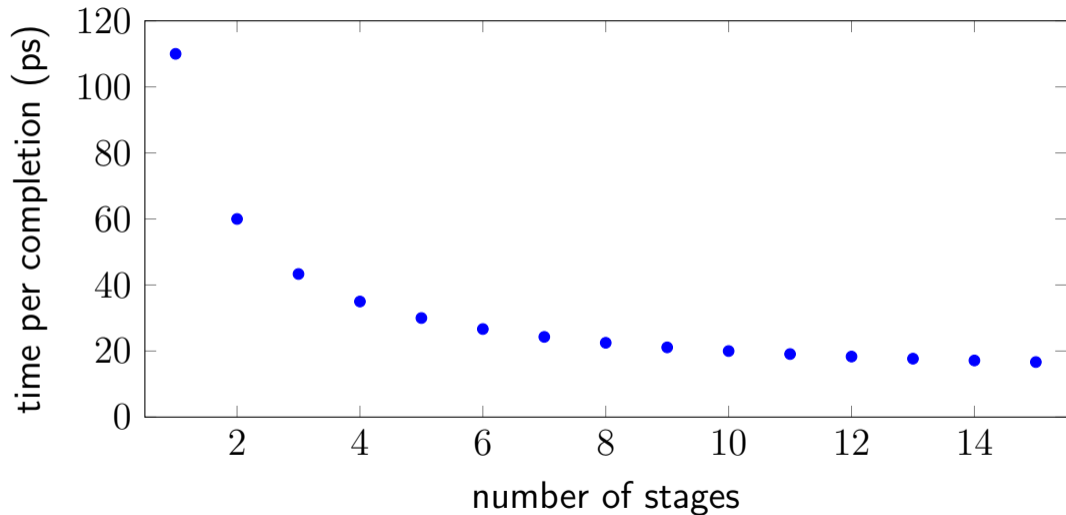
	<i>cycle #</i>	0	1	2	3	4	5	6	7	8	
0x100: add %r8, %r9		F1	F2	D1	D2	E1	E2	M1	M2	W1	W2
0x108: mov 0x1234(%r10), %r11			F1	F2	D1	D2	E1	E2	M1	M2	W1
0x110:						

suppose we double number of pipeline stages (to 10) and decrease cycle time from 500 ps to 250 ps

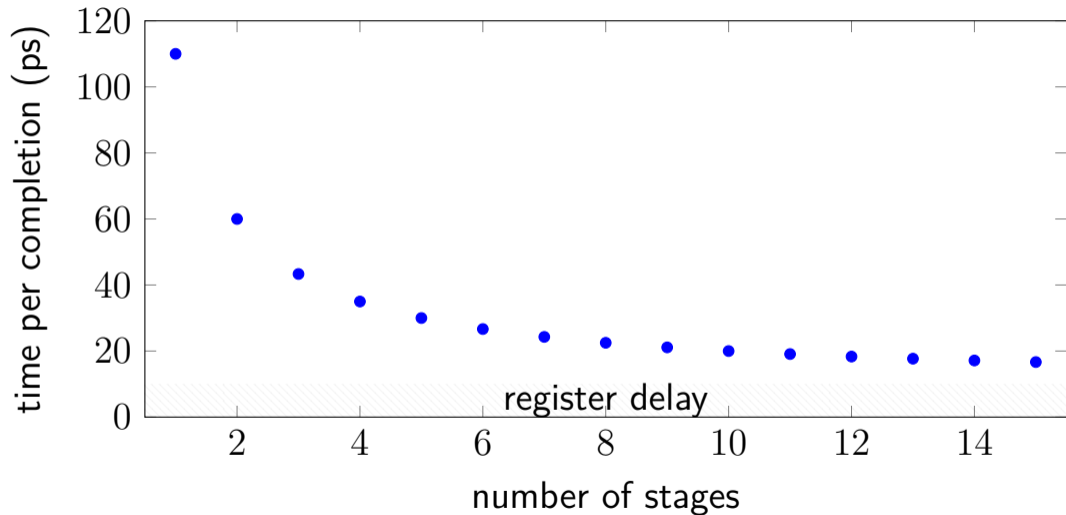
diminishing returns: register delays



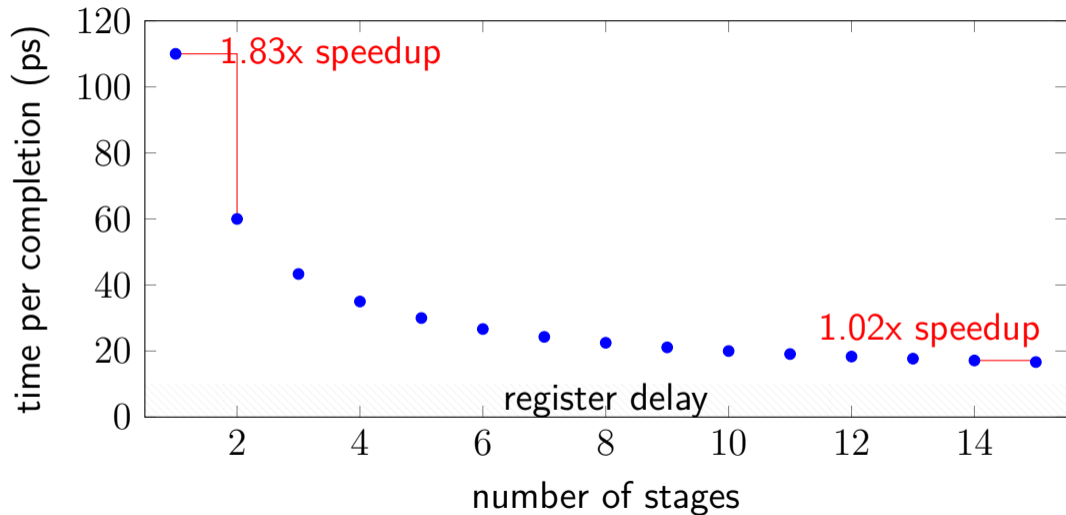
diminishing returns: register delays



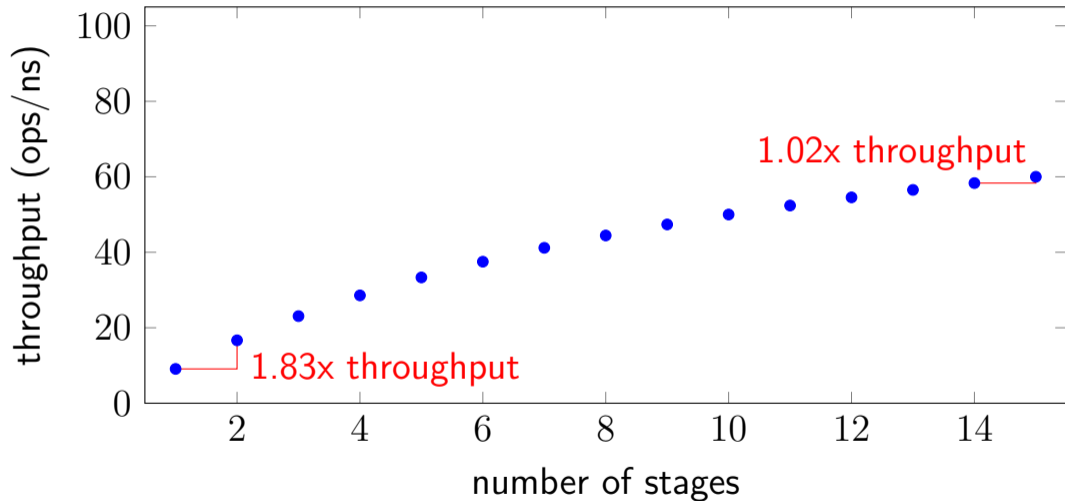
diminishing returns: register delays



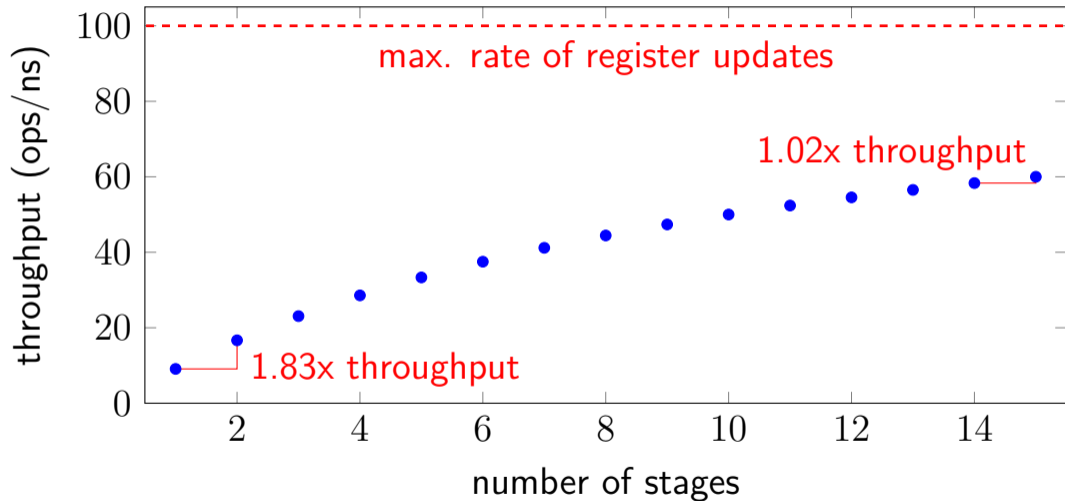
diminishing returns: register delays



diminishing returns: register delays



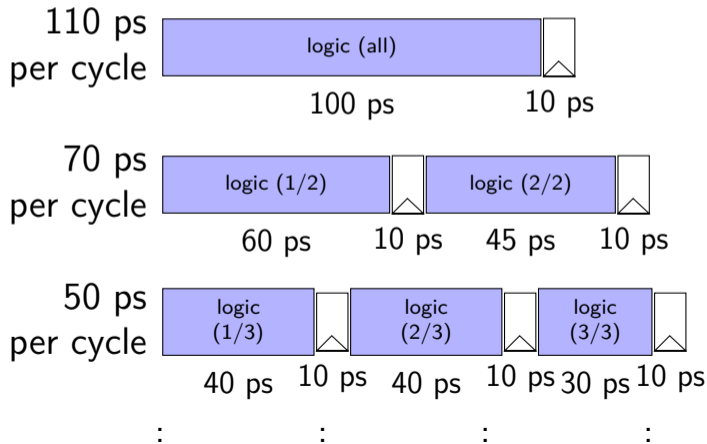
diminishing returns: register delays



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

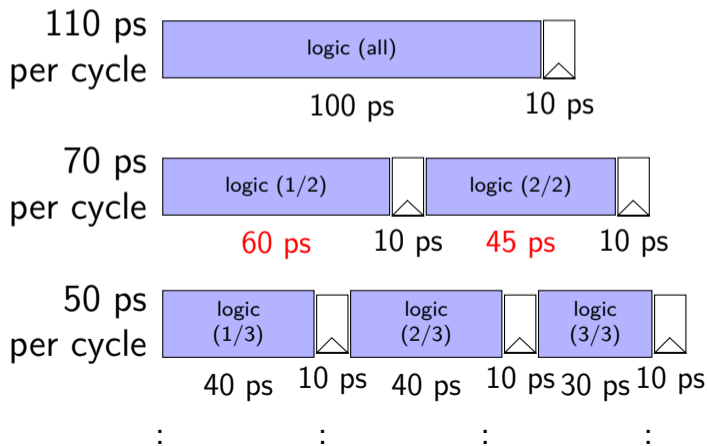
Probably not...



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

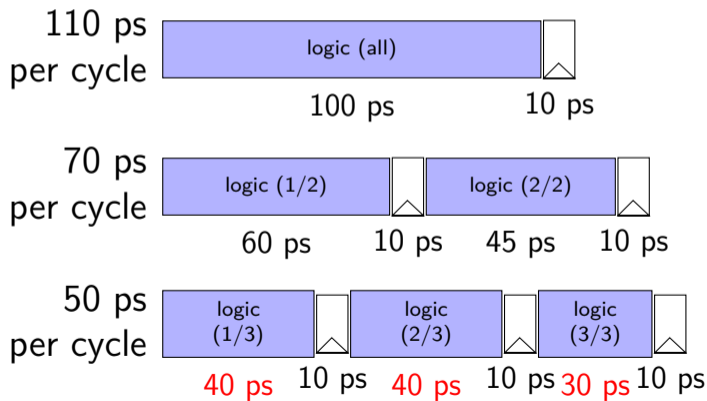
Probably not...



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

Probably not...



: : : :

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 for (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**

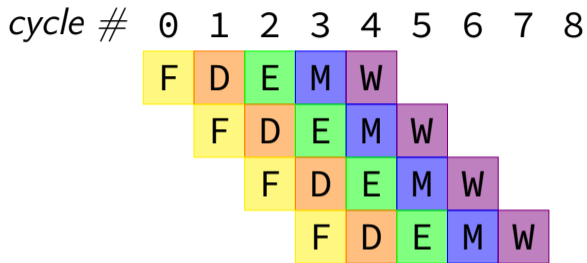
extra logic:

- sometimes don't change PC

- sometimes put do-nothing values in pipeline registers

stalling/nop pipeline diagram (1)

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



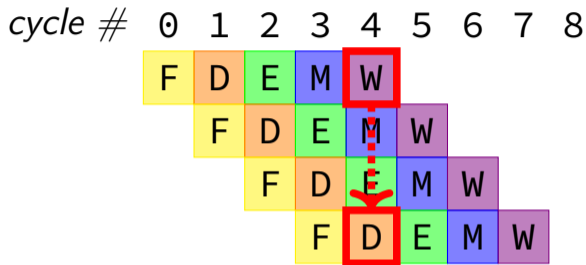
stalling/nop pipeline diagram (1)

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)

add %r8, %r9

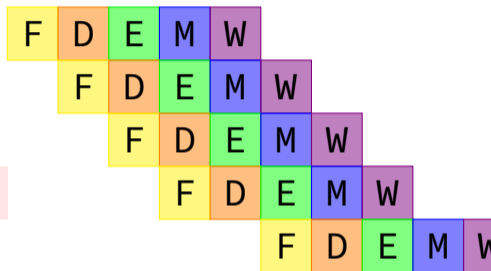
(nop)

(nop)

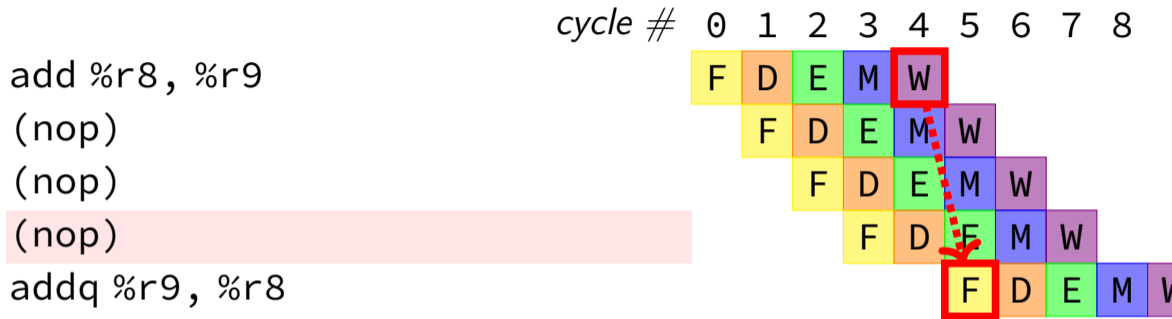
(nop)

addq %r9, %r8

cycle # 0 1 2 3 4 5 6 7 8



stalling/nop pipeline diagram (2)



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

backup slides

exercise: forwarding paths (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

subq %r8, %r9

ret (goes to andq)

andq %r10, %r9

in subq, %r8 is _____ addq.

in subq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

in andq, %r9 is _____ addq.

A: not forwarded from

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