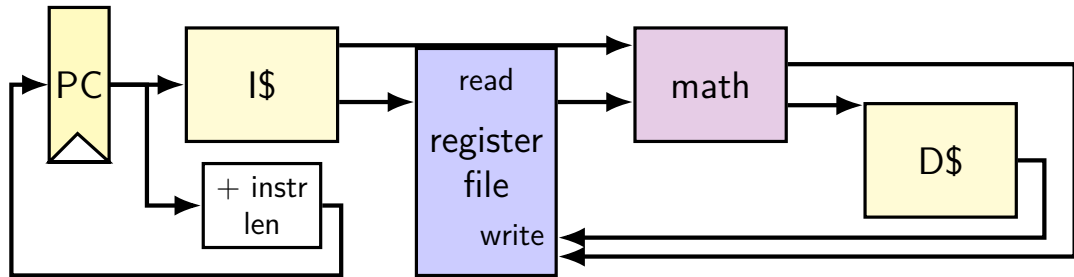
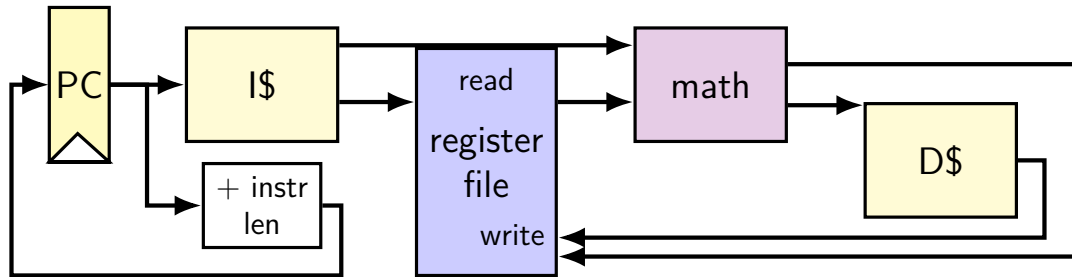


simple CPU



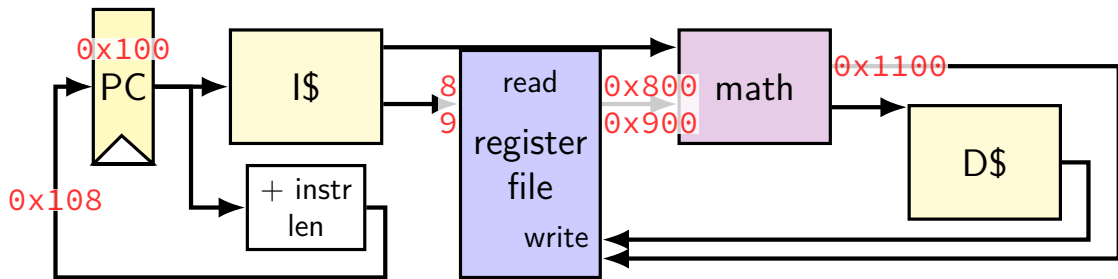
running instructions



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

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

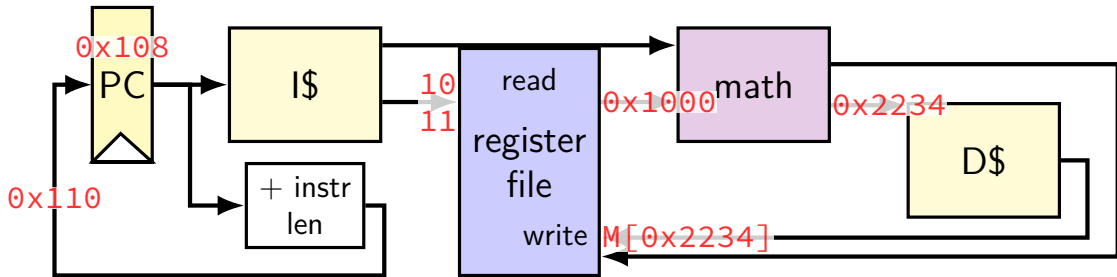
running instructions



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

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

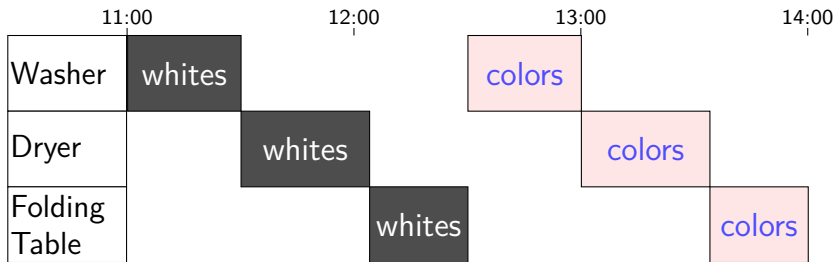
running instructions



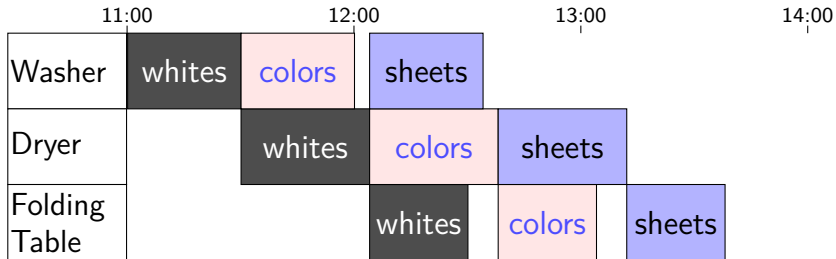
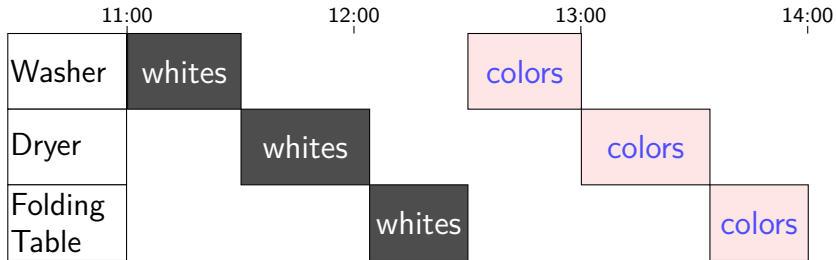
```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

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

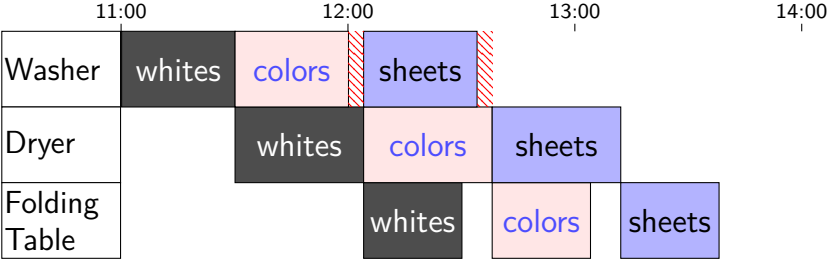
Human pipeline: laundry



Human pipeline: laundry

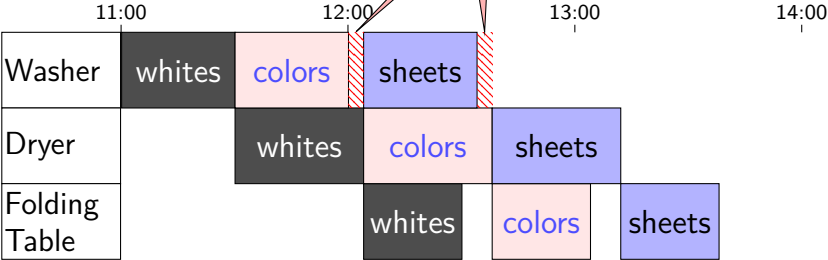


Waste (1)

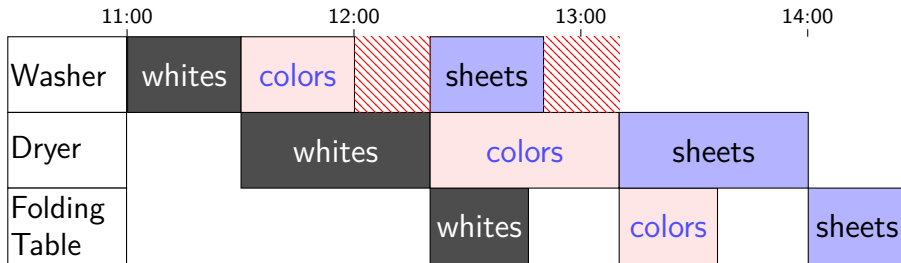


Waste (1)

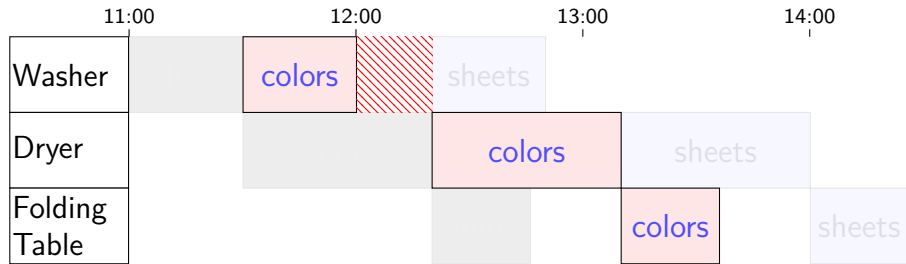
wasted time!



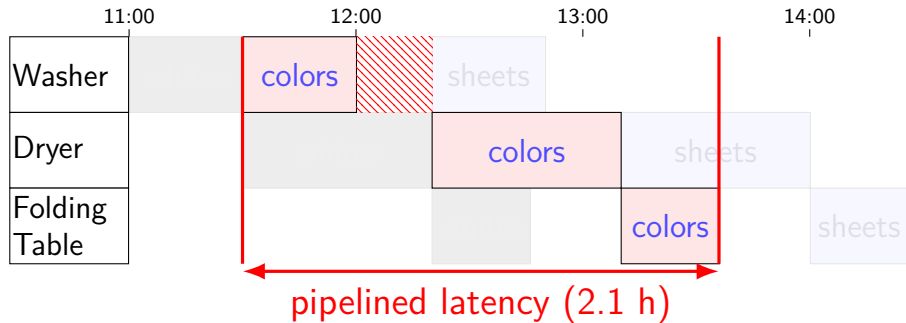
Waste (2)



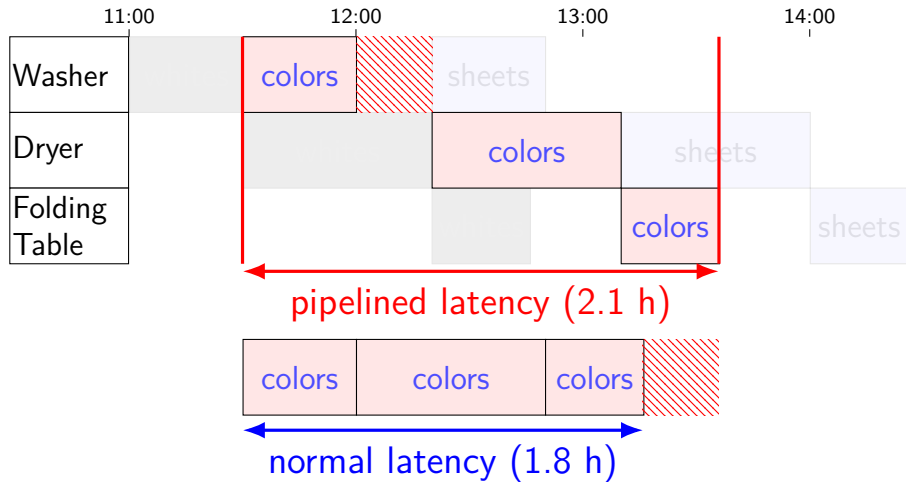
Latency — Time for One



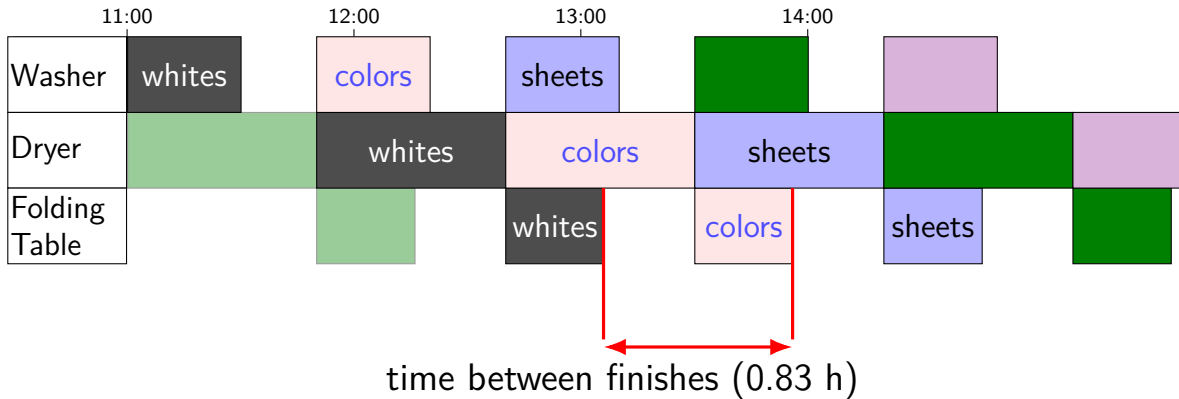
Latency — Time for One



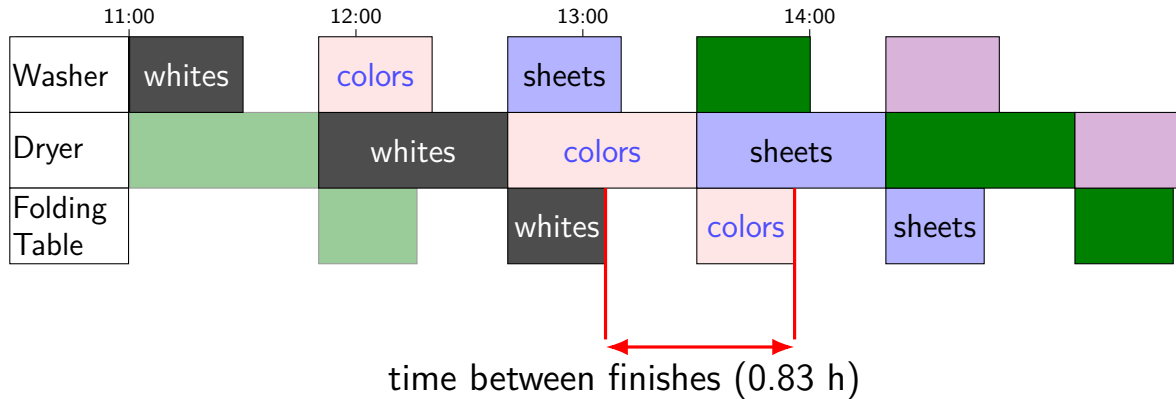
Latency — Time for One



Throughput — Rate of Many

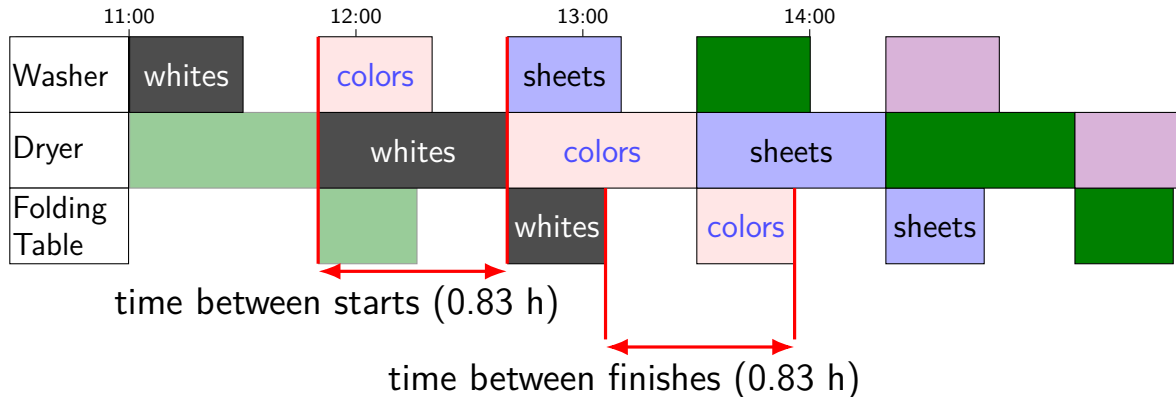


Throughput — Rate of Many



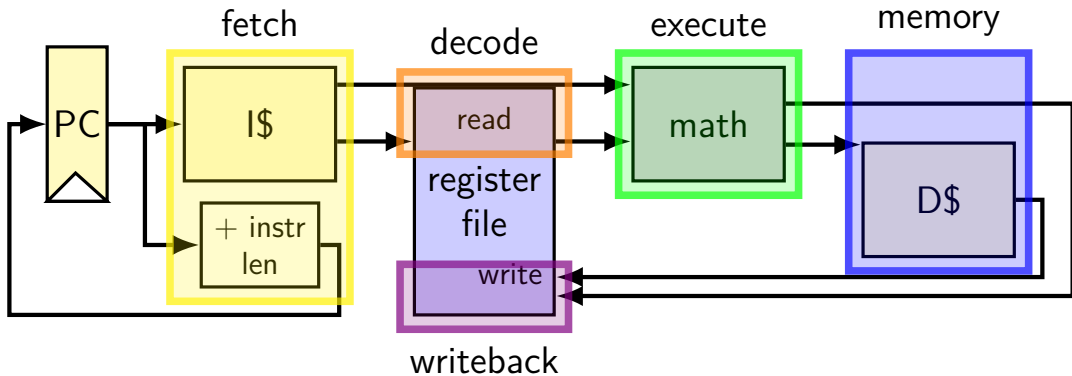
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many



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

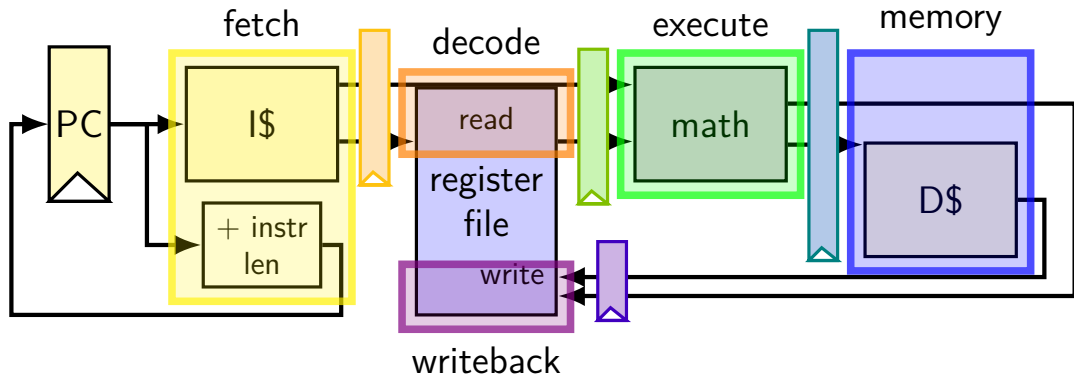
adding stages (one way)



divide running instruction into steps

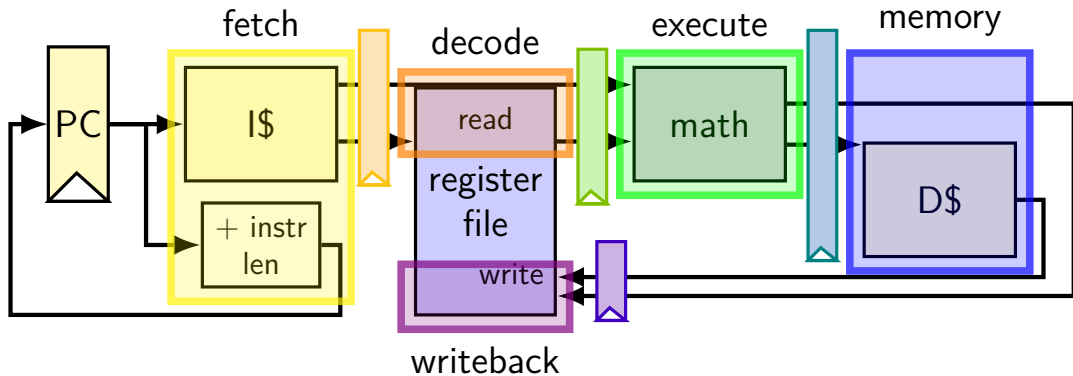
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

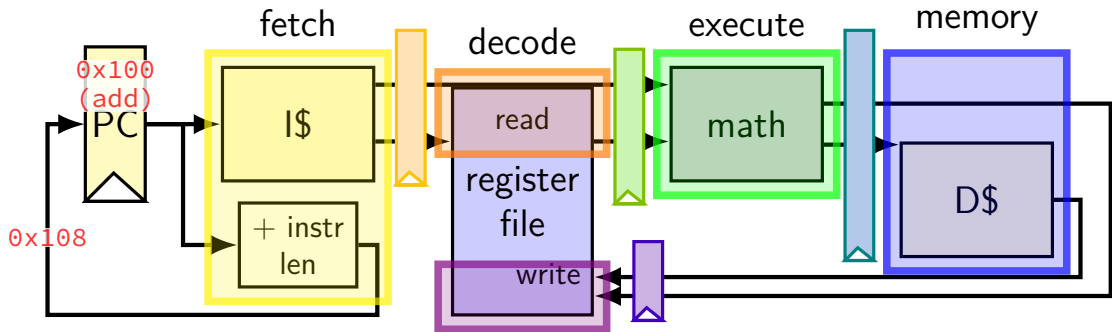


add 'pipeline registers' to hold values from instruction

running some instructions



running some instructions



writeback

cycle # 0 1 2 3 4 5 6 7 8

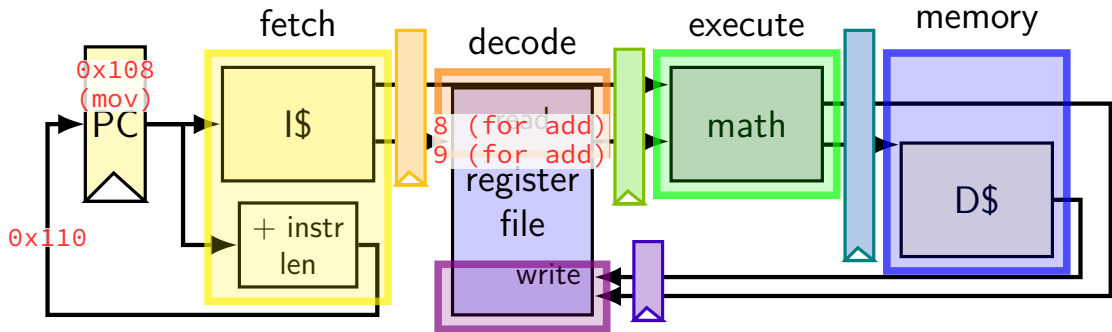
`0x100: add %r8, %r9`

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

`0x110: xor %r12, %r13`

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

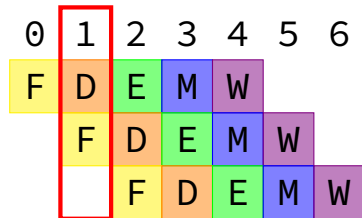


`0x100: add %r8, %r9`

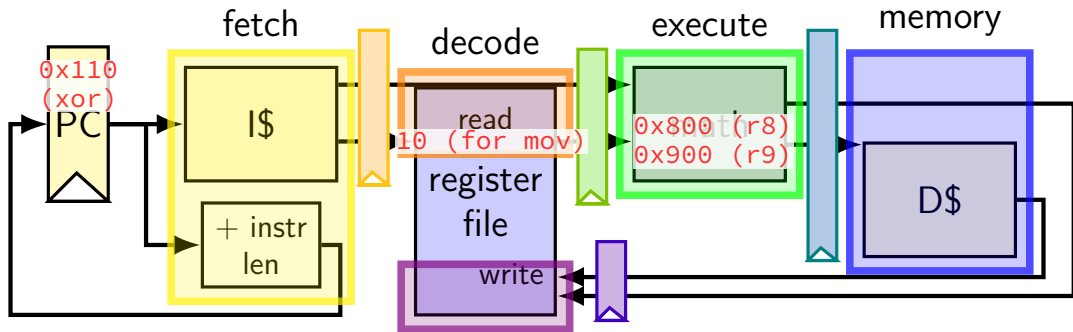
`0x108: mov 0x1234(%r10), %r11`

`0x110: xor %r12, %r13`

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



running some instructions



`0x100: add %r8, %r9`

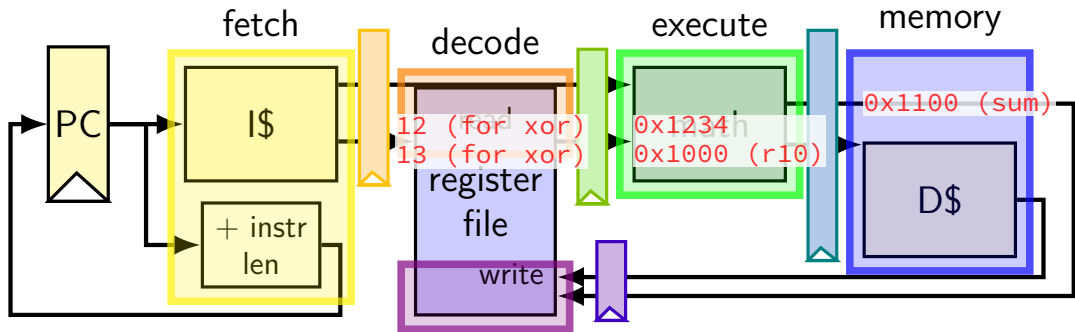
`0x108: mov 0x1234(%r10), %r11`

`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

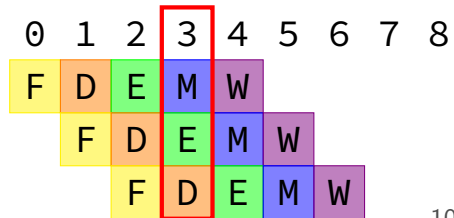


0x100: add %r8, %r9

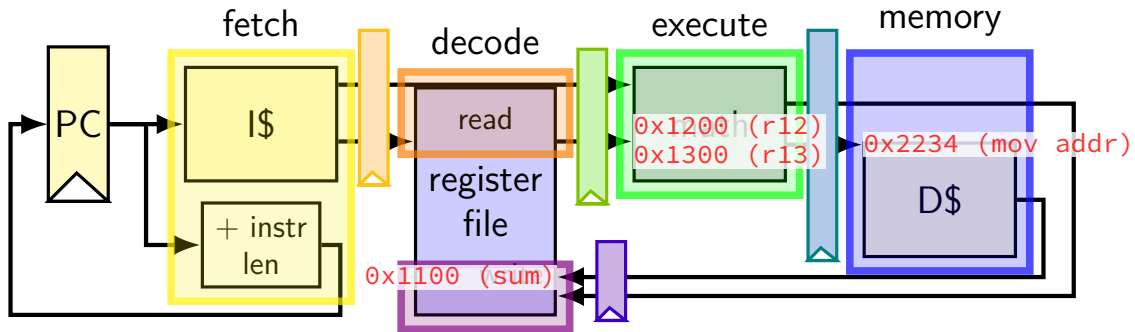
0x108: mov 0x1234(%r10), %r11

0x110: xor %r12, %r13

writeback cycle #



running some instructions



0x100: add %r8, %r9

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

0x110: xor %r12, %r13

cycle #

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

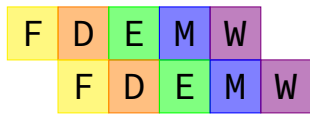
exercise: throughput/latency (1)

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

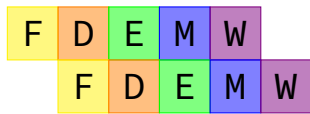
exercise: throughput/latency (1)

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?

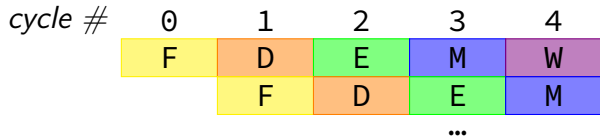
- A. 100 ps B. 500 ps C. 2000 ps D. 2500 ps E. something else

exercise: throughput overall?

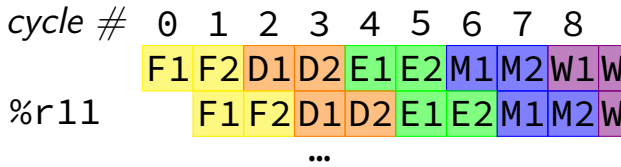
- A. 1 instr/100 ps B. 1 instr/500 ps C. 1 instr/2000ps D. 1 instr/2500 ps
E. something else

exercise: throughput/latency (2)

0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...



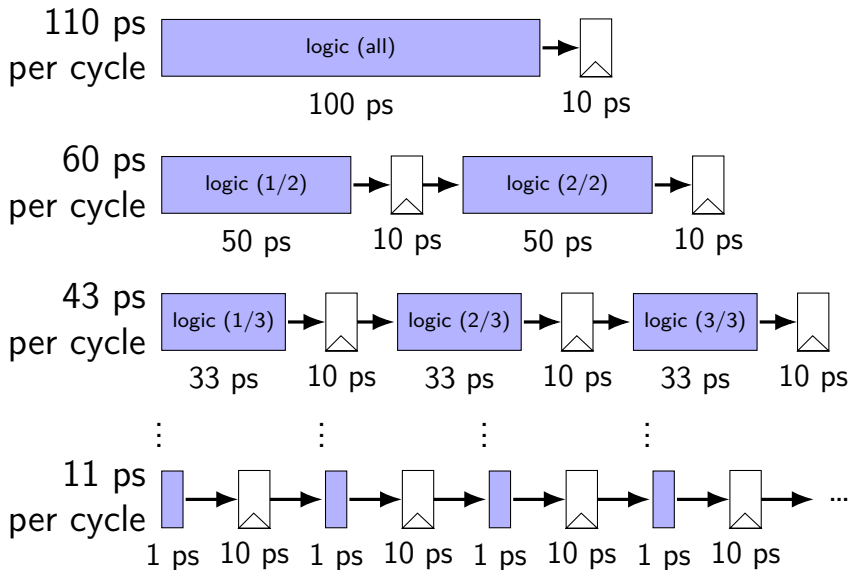
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...



double number of pipeline stages (to 10) + decrease cycle time from 500 ps to 250 ps — throughput?

- A. 1 instr/100 ps B. 1 instr/250 ps C. 1 instr/1000ps D. 1 instr/5000 ps
E. something else

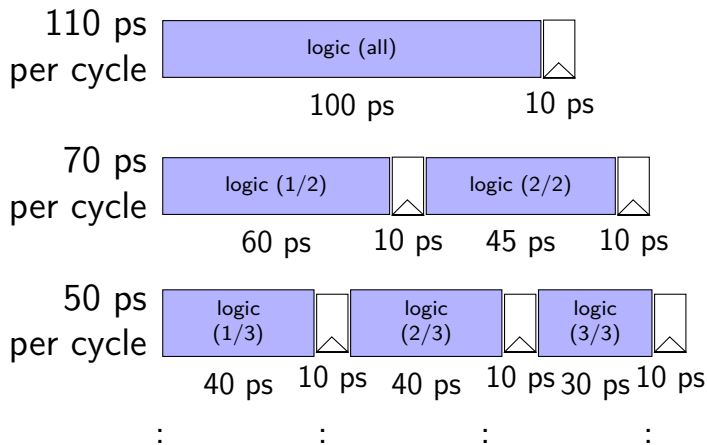
diminishing returns: register delays



diminishing returns: uneven split

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

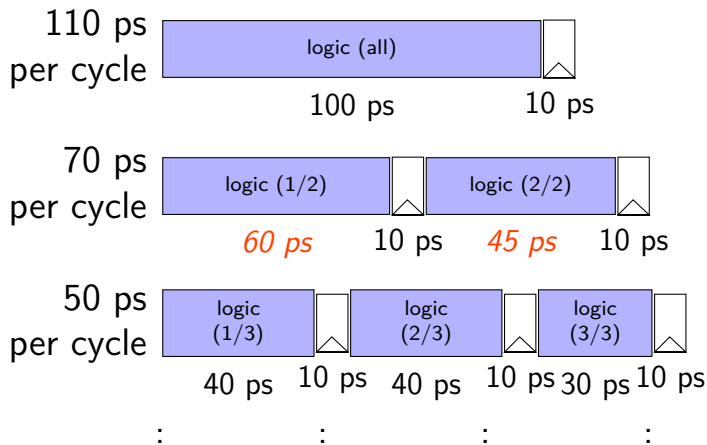
Probably not...



diminishing returns: uneven split

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

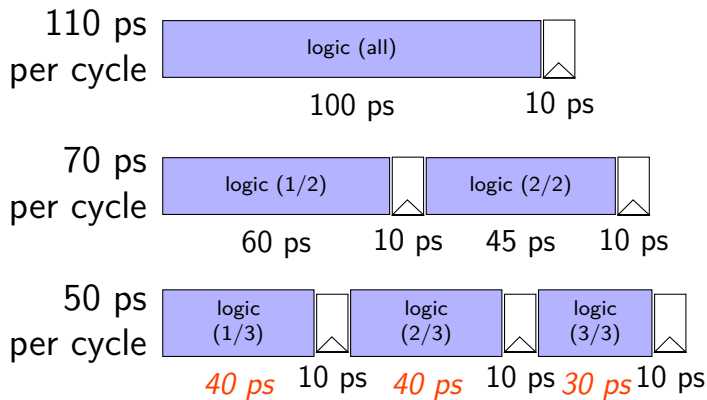
Probably not...



diminishing returns: uneven split

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

Probably not...



:

:

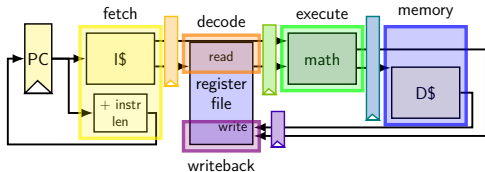
:

:

a data hazard

```

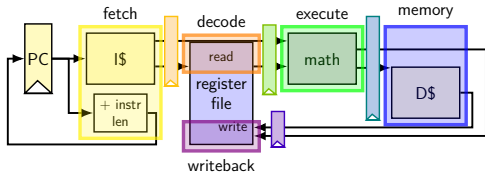
// initially %r8 = 800,
//                %r9 = 900, etc.
addq %r8, %r9    // R8 + R9 -> R9
addq %r9, %r8    // R9 + R8 -> R9
addq ...
addq ...
    
```



	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

a data hazard

```
// initially %r8 = 800,
//                %r9 = 900, etc.
addq %r8, %r9    // R8 + R9 -> R9
addq %r9, %r8    // R9 + R8 -> R9
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 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?

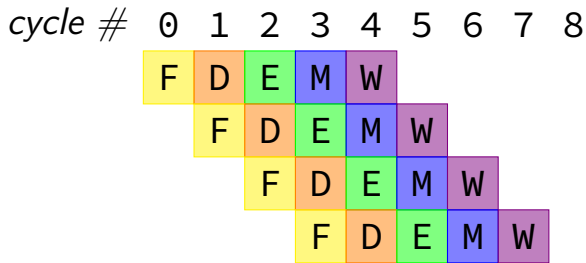
stalling/nop pipeline diagram (1)

add %r8, %r9

nop

nop

addq %r9, %r8



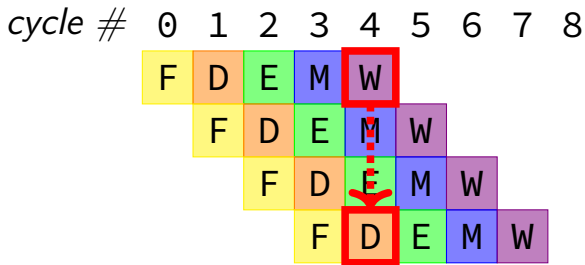
stalling/nop pipeline diagram (1)

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

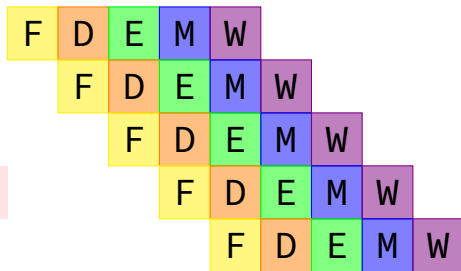
nop

nop

nop

addq %r9, %r8

cycle # 0 1 2 3 4 5 6 7 8



stalling/nop pipeline diagram (2)

add %r8, %r9

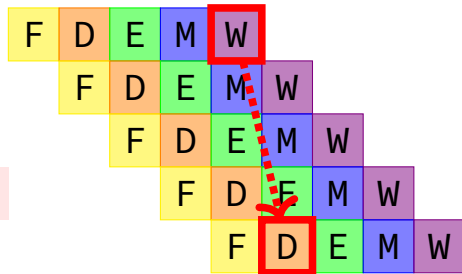
nop

nop

nop

addq %r9, %r8

cycle # 0 1 2 3 4 5 6 7 8



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

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

control hazard

0x00: cmpq %r8, %r9

0x08: je 0xFFFF

0x10: addq %r10, %r11

	fetch	fetch→decode		decode→execute		execute→write	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
```

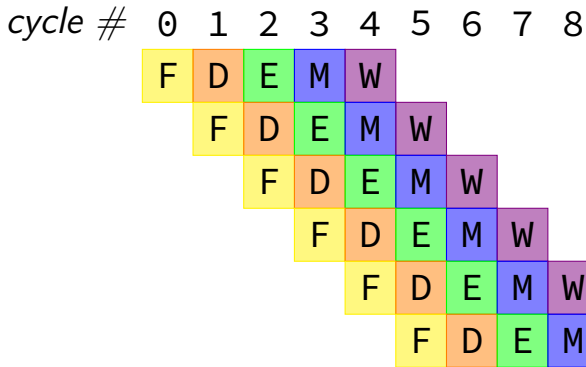
	fetch	fetch→decode	decode→execute	execute→write	execute→writeback	...			
cycle	PC	rA	rB	R[rA]	R[rB]	result
0	0x0								
1	0x8	8	9						
2	???	--	---	800	900				
3	???	--	---	---	---	less than			

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

jXX: stalling?

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

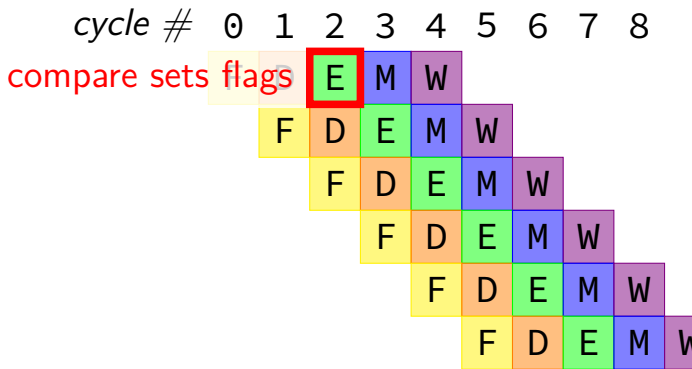
```
...
cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



jXX: stalling?

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

```
...
cmpq %r8, %r9
jne LABEL
(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)
```

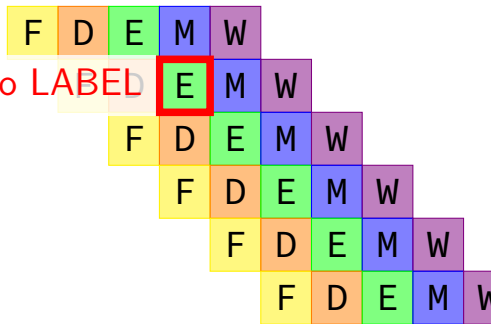
```
(do nothing)
```

```
xorq %r10, %r11
```

```
movq %r11, 0(%r12)
```

```
...
```

cycle # 0 1 2 3 4 5 6 7 8

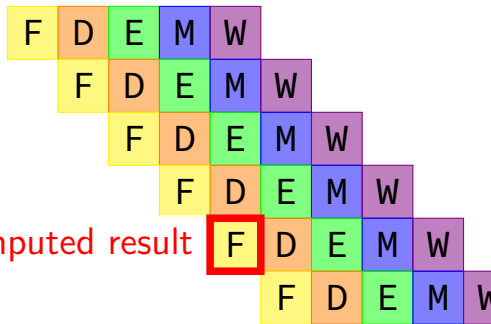


jXX: stalling?

```
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)
...
```

cycle # 0 1 2 3 4 5 6 7 8



use computed result

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

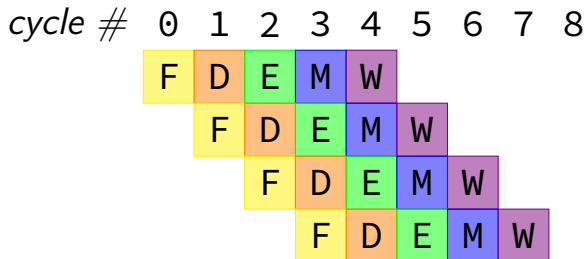
right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

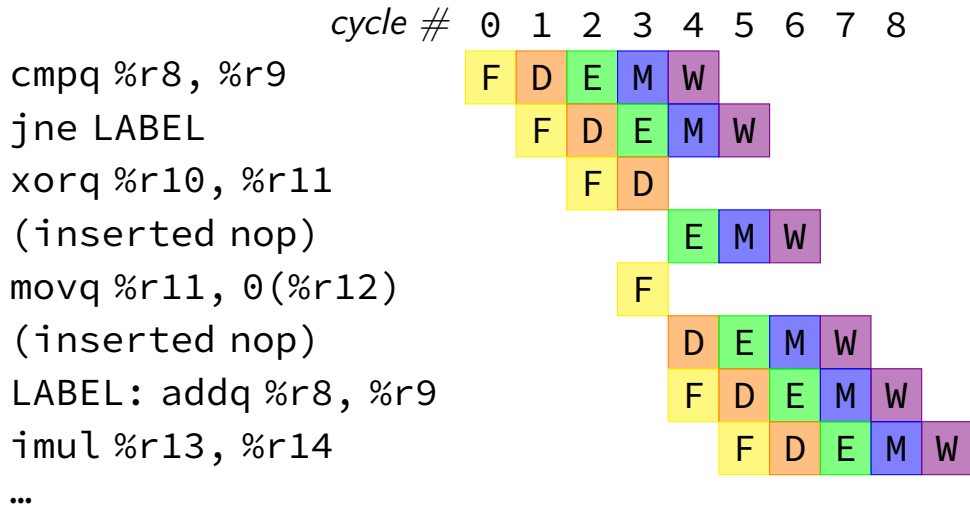
```
cmpq %r8, %r9  
jne LABEL  
xorq %r10, %r11  
movq %r11, 0(%r12)  
...
```

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

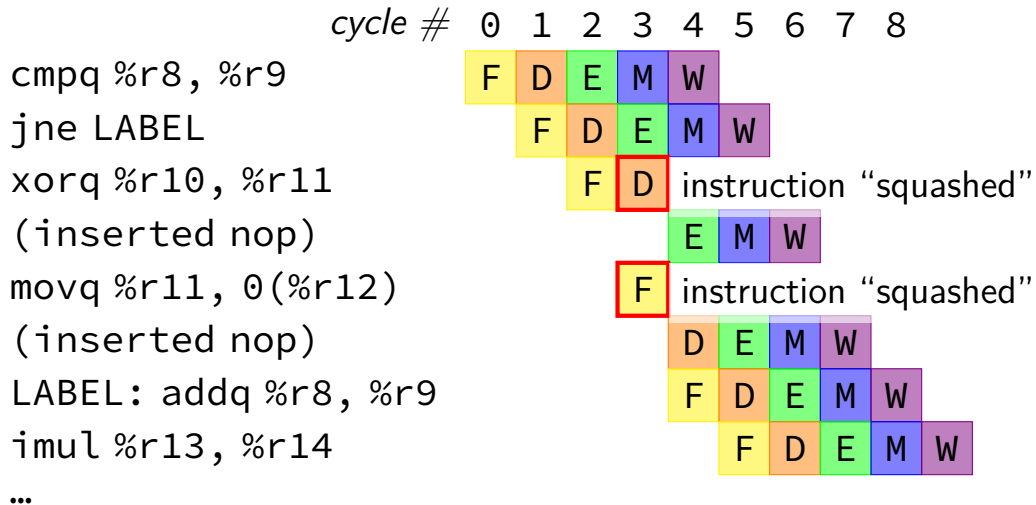
```
cmpq %r8, %r9  
jne LABEL  
xorq %r10, %r11  
movq %r11, 0(%r12)  
...
```



jXX: speculating wrong



jXX: speculating wrong



“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.

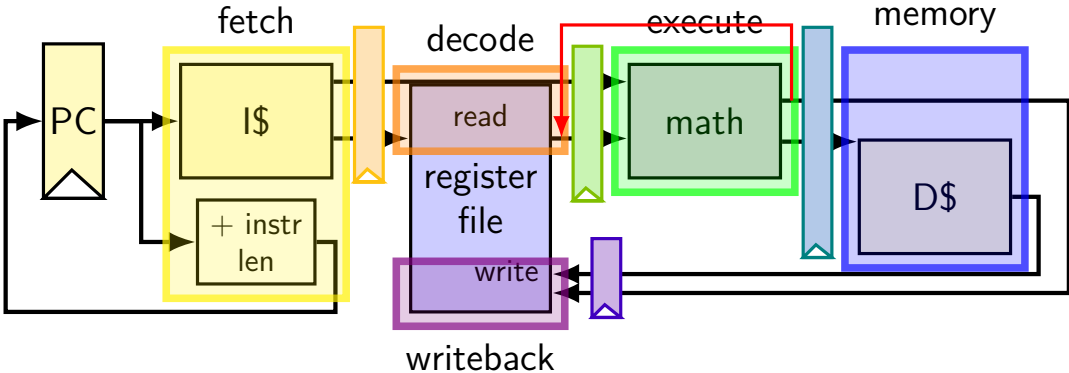
opportunity

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

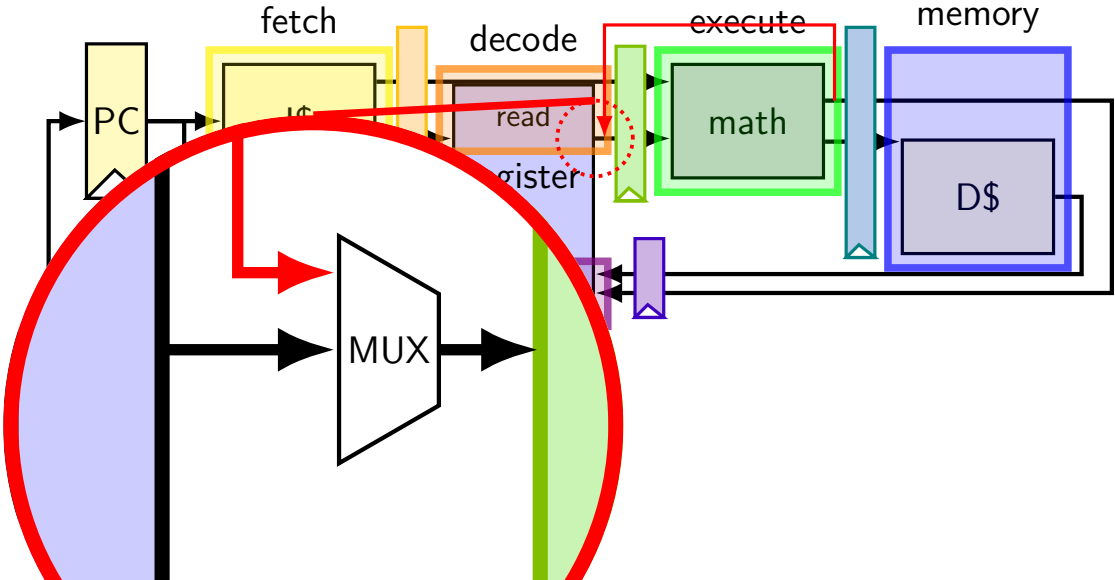
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



opportunity 2

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

```
0x0: addq %r8, %r9
```

```
0x2: nop
```

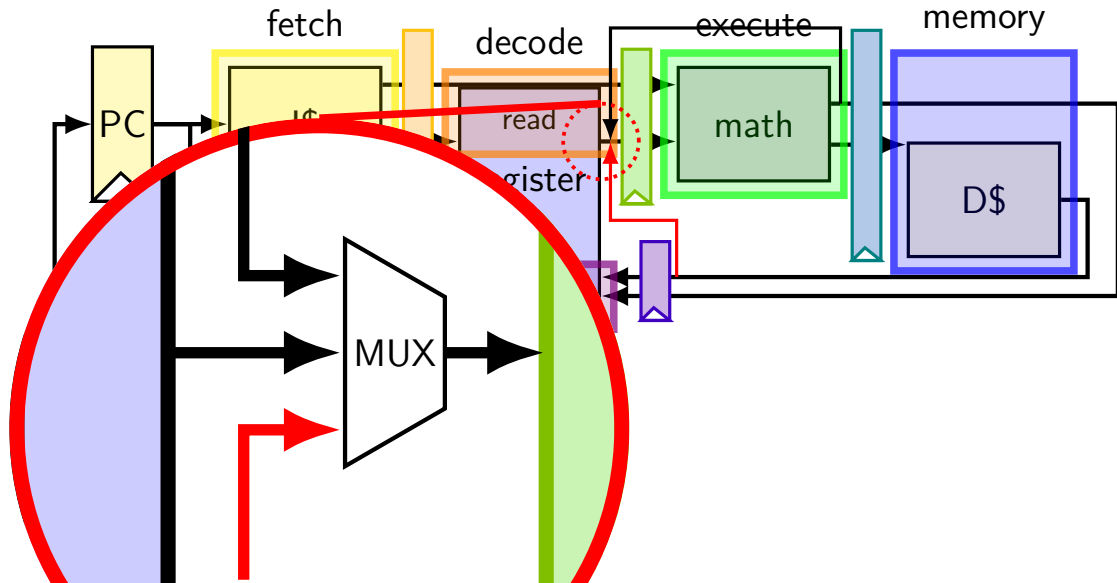
```
0x3: addq %r9, %r8
```

...

cycle	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
	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.

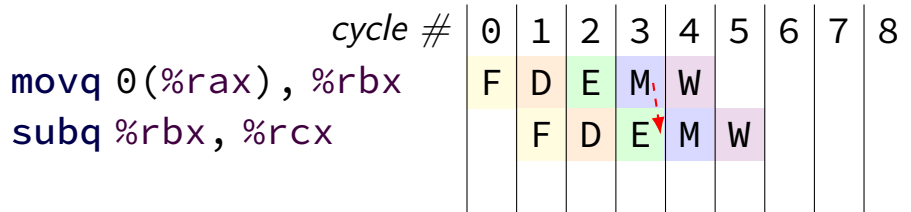
in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

A: not forwarded from

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

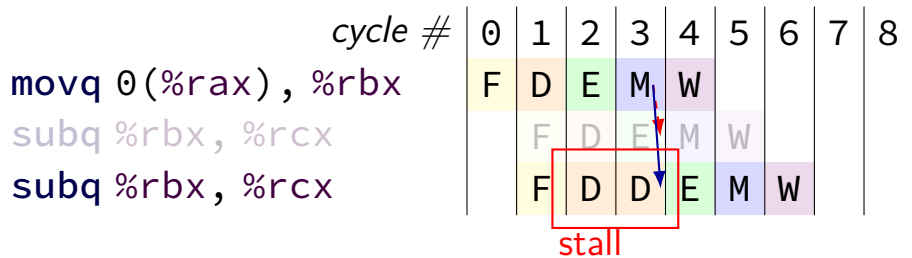
unsolved problem



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



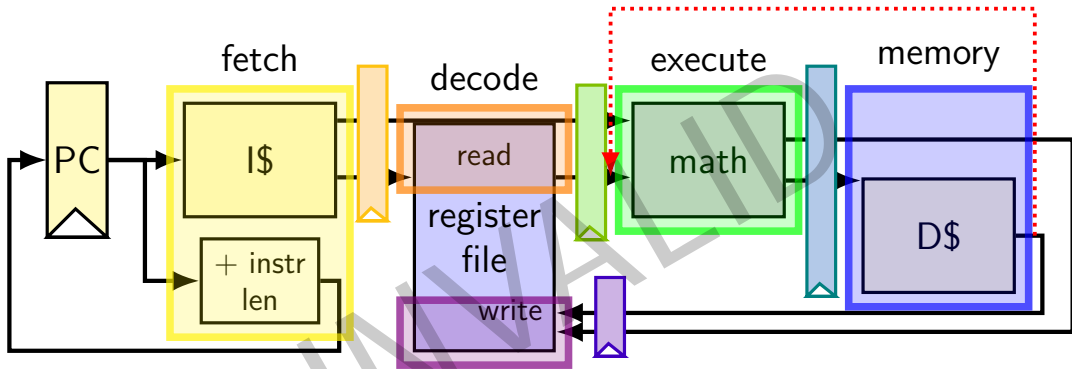
combine stalling and forwarding to resolve hazard

assumption in diagram: hazard detected in subq's decode stage
(since easier than detecting it in fetch stage)

solveable problem

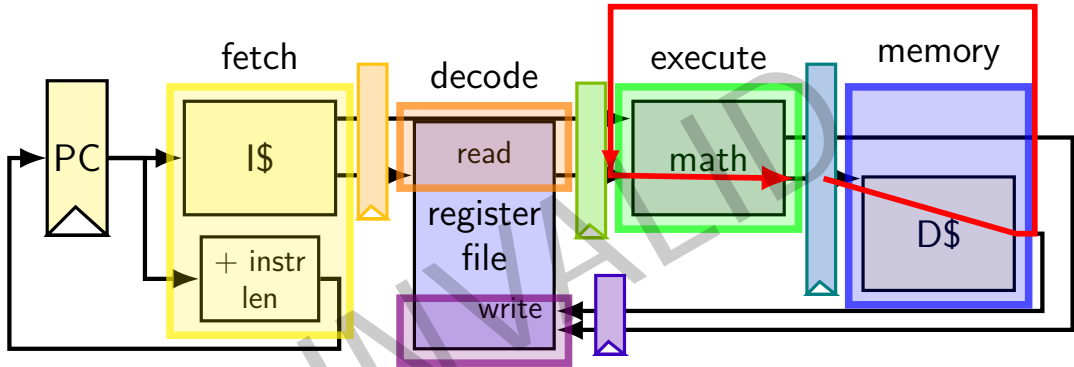
	cycle #	0	1	2	3	4	5	6	7	8
movq 0(%rax), %rbx		F	D	E	M	W				
movq %rbx, 0(%rcx)			F	D	E	M	W			

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*, *control hazards*

ex.: dependencies and hazards (1)

addq %rax, %rbx

subq %rax, %rcx

movq \$100, %rcx

addq %rcx, %r10

addq %rbx, %r10

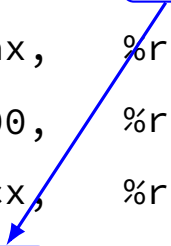
where are dependencies?

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



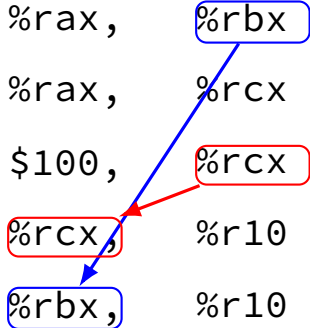
where are dependencies?

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

```
graph TD; I1["addq %rax, %rbx"] -- blue --> I5["addq %rbx, %r10"]; I2["subq %rax, %rcx"] -- red --> I4["addq %rcx, %r10"]; I3["movq $100, %rcx"] -- red --> I4; I4 -- red --> I5;
```

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

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

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

(assuming register file does not read while writing)

`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

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

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?

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

exercise: different pipeline

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

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

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, %r9										
movq %r9, 8(%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, 8(%rbx)					F	D	E1	E2	M	W	
movq %r9, 8(%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, 8(%rbx)				F	D	E1	E2	M	W	
movq %r9, 8(%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, 8(%rbx)				F	D	E1	E2	M	W		
movq %r9, 8(%rbx)					F	D	E1	E2	M	W	
movq %rcx, %r9						F	D	E1	E2	M	W

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.

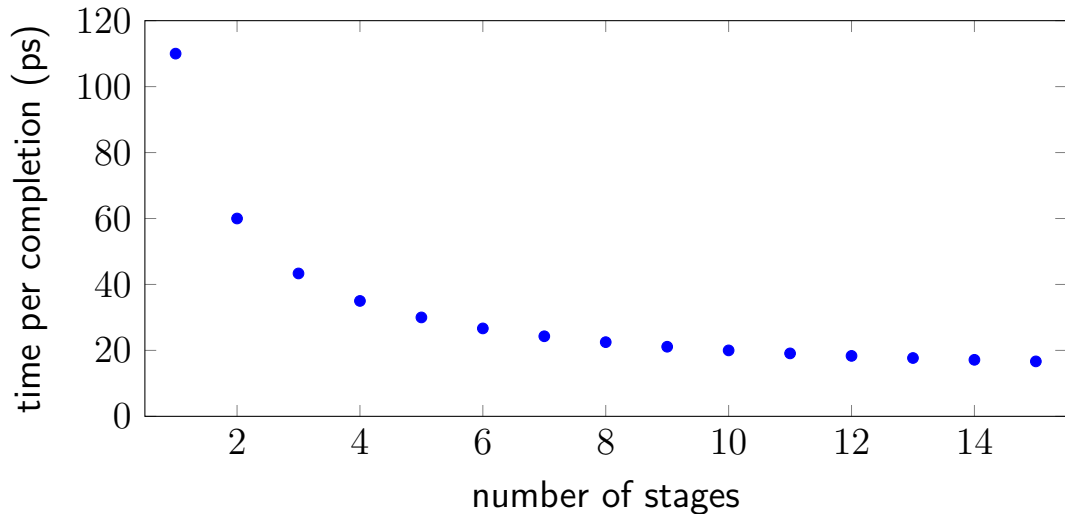
in andq, %r9 is _____ subq.

in andq, %r9 is _____ addq.

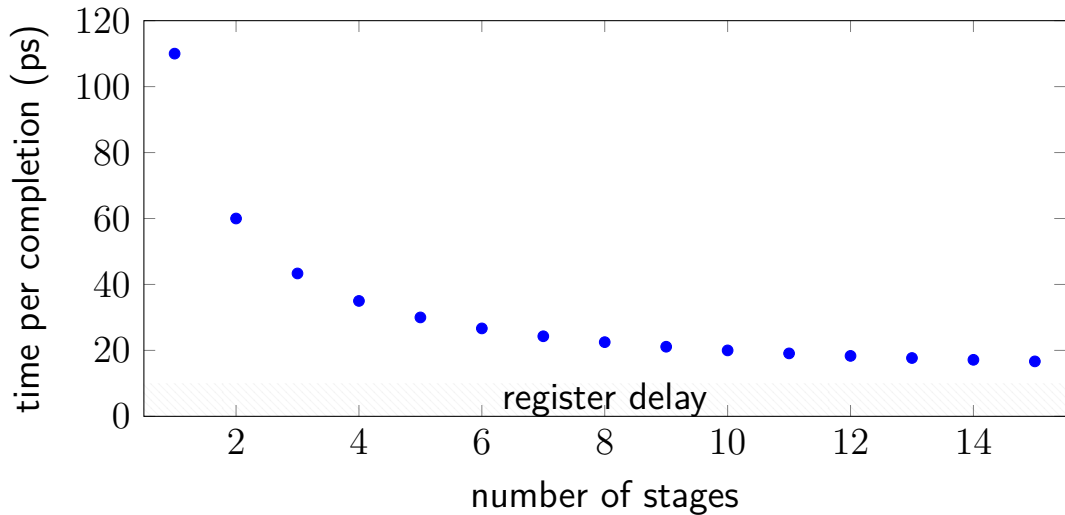
A: not forwarded from

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

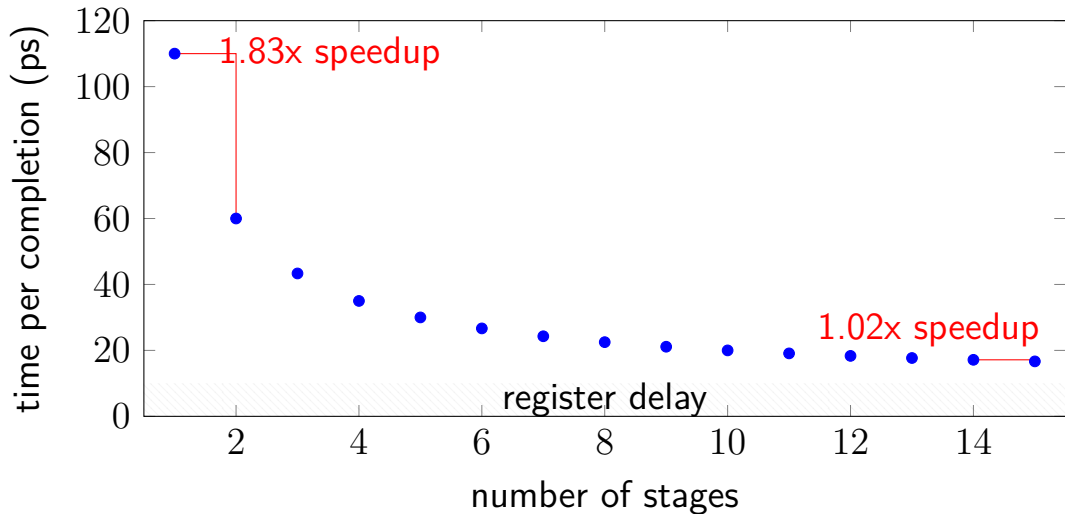
diminishing returns: register delays



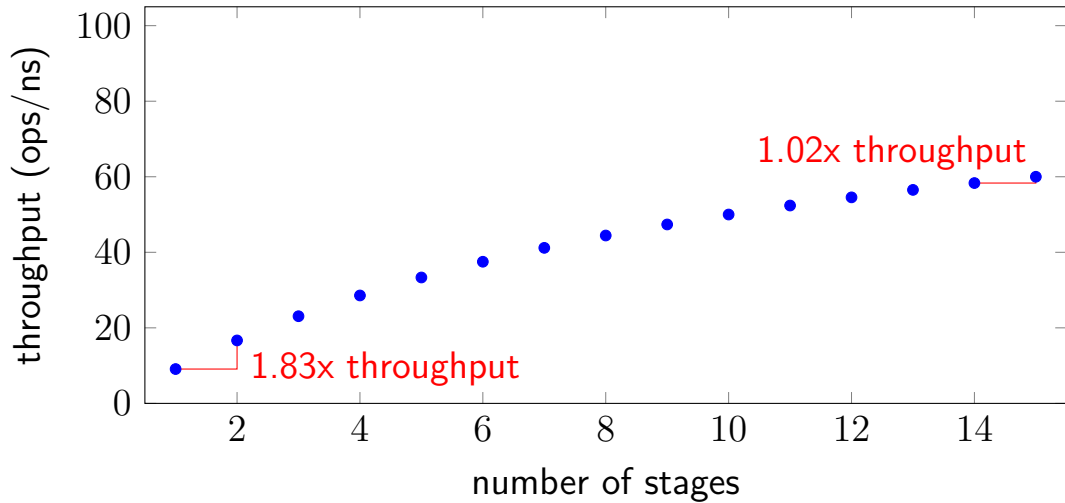
diminishing returns: register delays



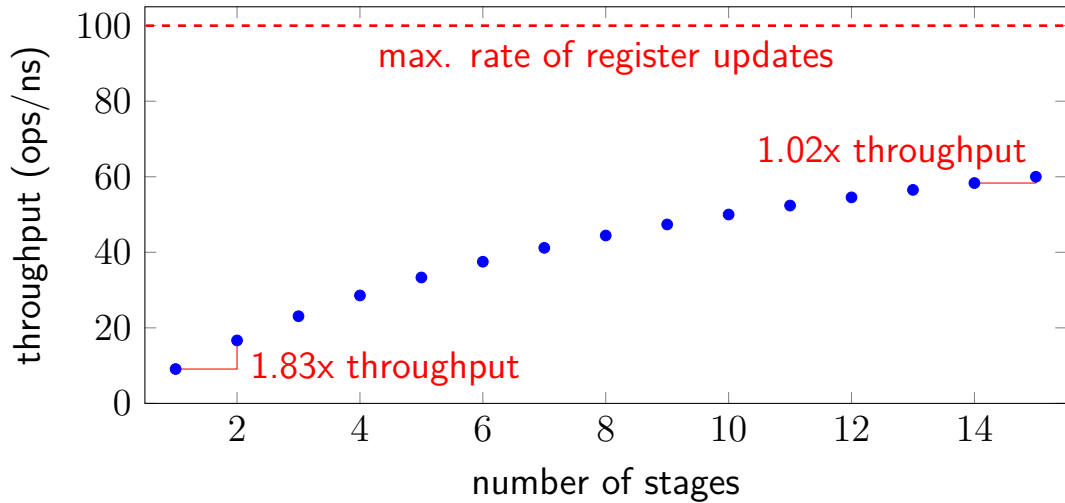
diminishing returns: register delays



diminishing returns: register delays



diminishing returns: register delays



importance of prediction

5-stage pipeline, predict not-taken versus always stall

hypothetical instruction mix

			cycles	cycles
taken jXX	3%	(predicted)	3	(stall)
non-taken jXX	5%	not-taken)	1	no predict)
others	92%		1*	1*

importance of prediction

5-stage pipeline, predict not-taken versus always stall

hypothetical instruction mix

		cycles	cycles
taken jXX	3%	(predict)	(stall)
non-taken jXX	5%	not-taken)	no predict)
others	92%	1*	1*

predict: $3 \times .03 + 1 \times .05 + 1 \times .92 = 1.06$ cycles/instr.

stall: $3 \times .03 + 3 \times .05 + 1 \times .92 = 1.16$ cycles/instr.

$(1.16 \div 1.06 \approx 1.09x \text{ faster})$

prediction and OOO

deeper pipeline — much higher misprediction penalty