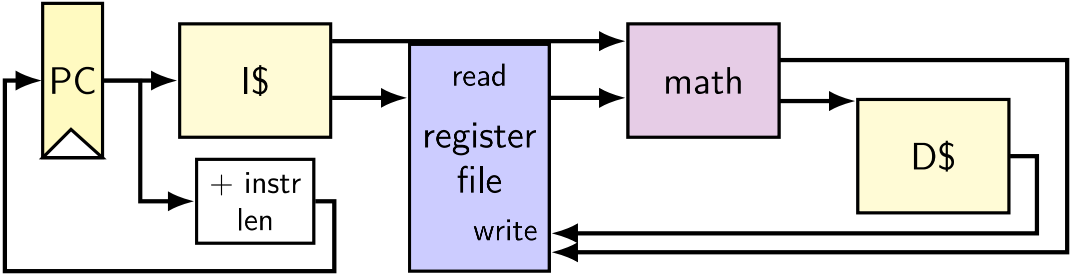
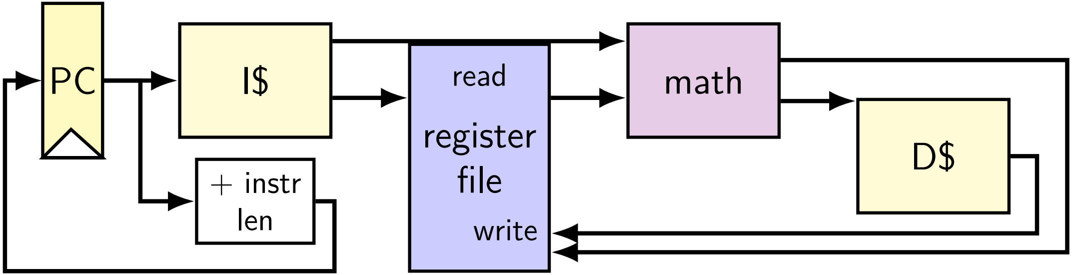
simple CPU



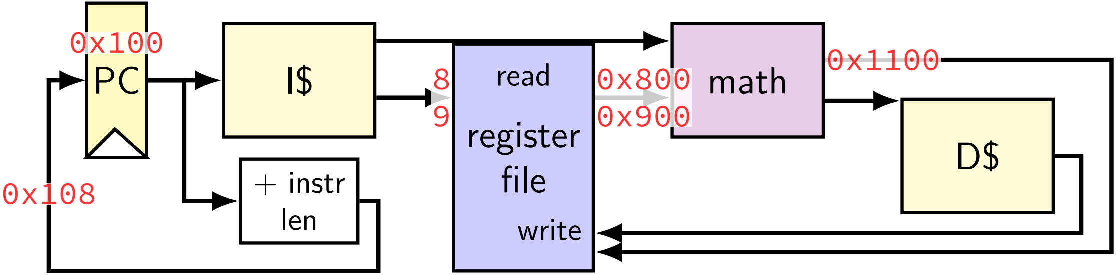
running instructions




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

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

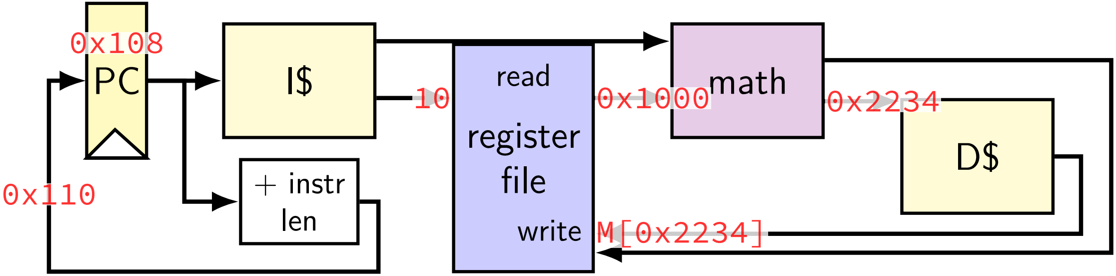
running instructions




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

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

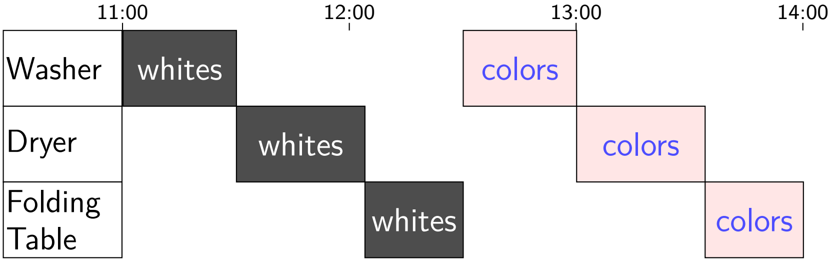
running instructions



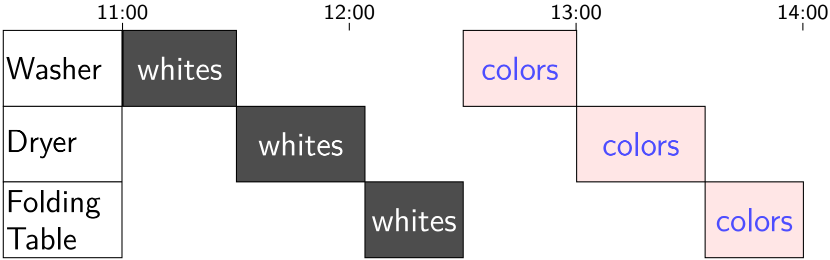

```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

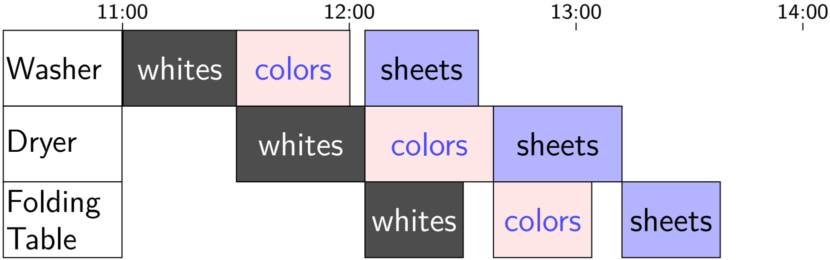
```
…
%r8: 0x800
%r9: 0x1100
%r10: 0x1000
%r11: M[0x2234]
…
```

Human pipeline: laundry

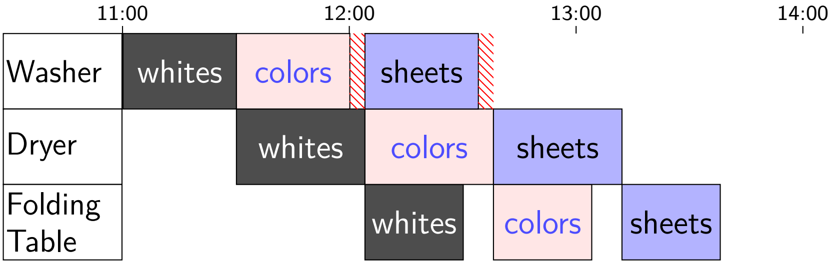


Human pipeline: laundry



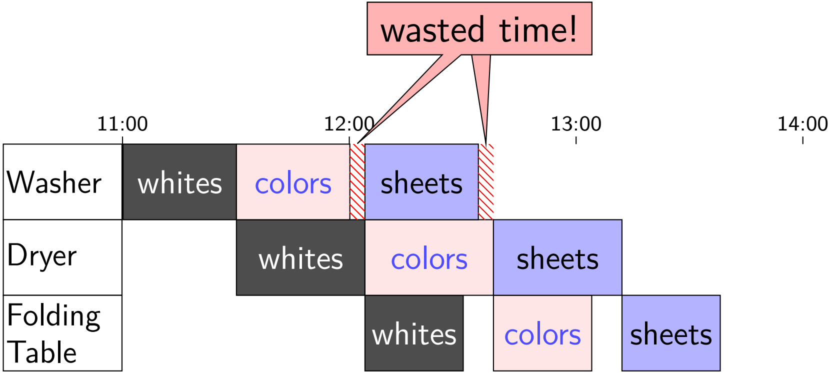


Waste (1)

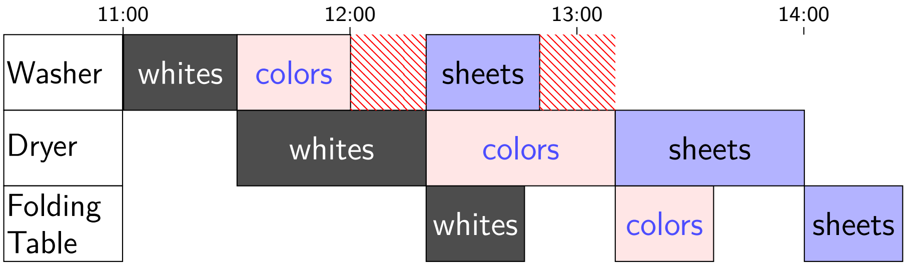


11:00
12:00
13:00
14:00
Washer
whites
colors
sheets
Dryer
whites
colors
sheets
Folding Table
whites
colors
sheets

Waste (1)

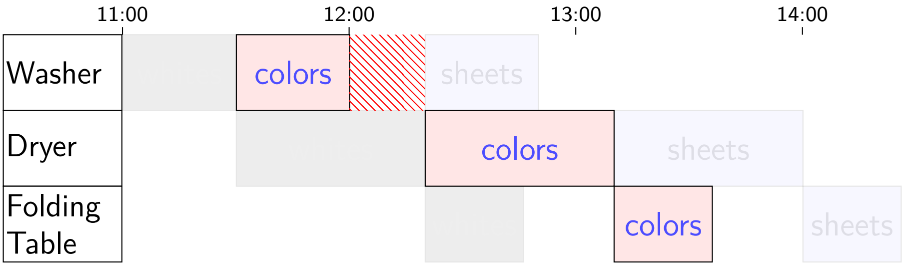

wasted time!
11:00
12:00
13:00
14:00
Washer
whites
colors
sheets
Dryer
whites
colors
sheets
Folding Table
whites
colors
sheets

Waste (2)

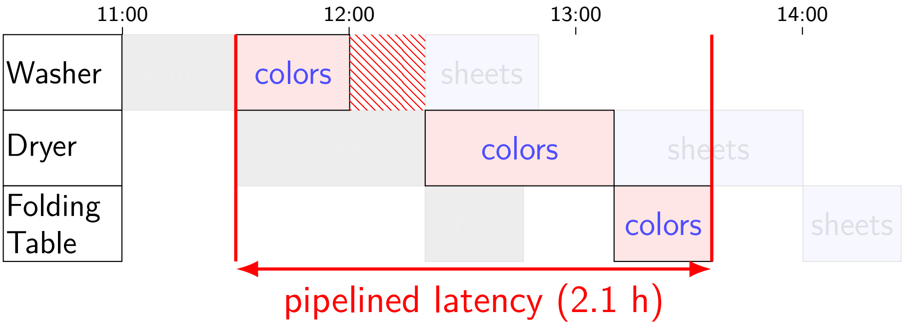


11:00
12:00
13:00
14:00
Washer
whites
colors
sheets
Dryer
whites
colors
sheets
Folding Table
whites
colors
sheets

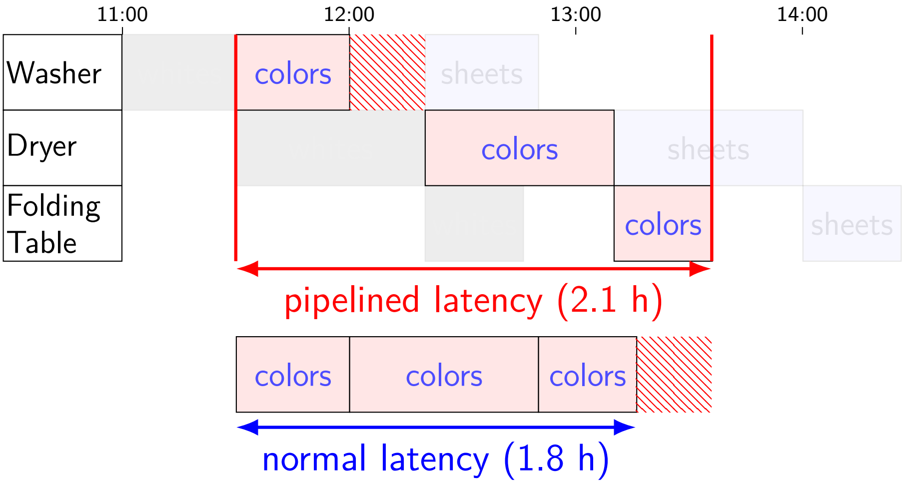
Latency — Time for One



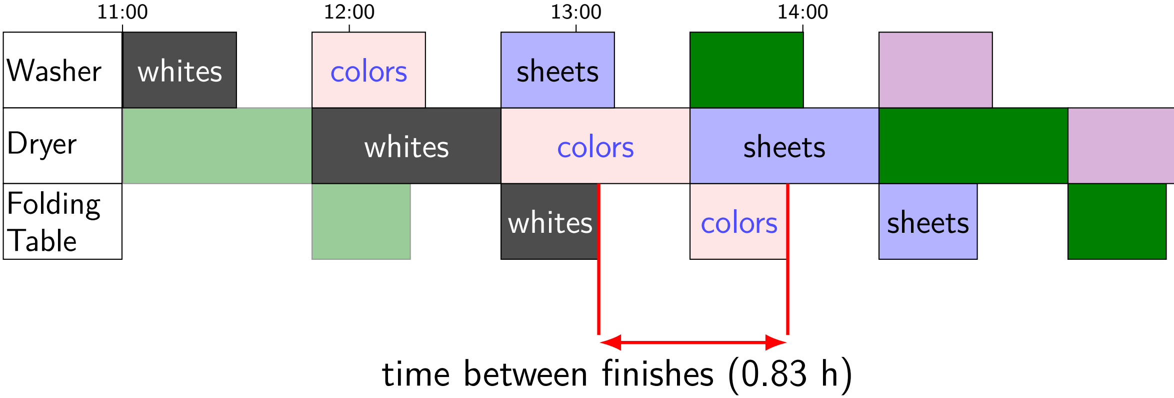
Latency — Time for One



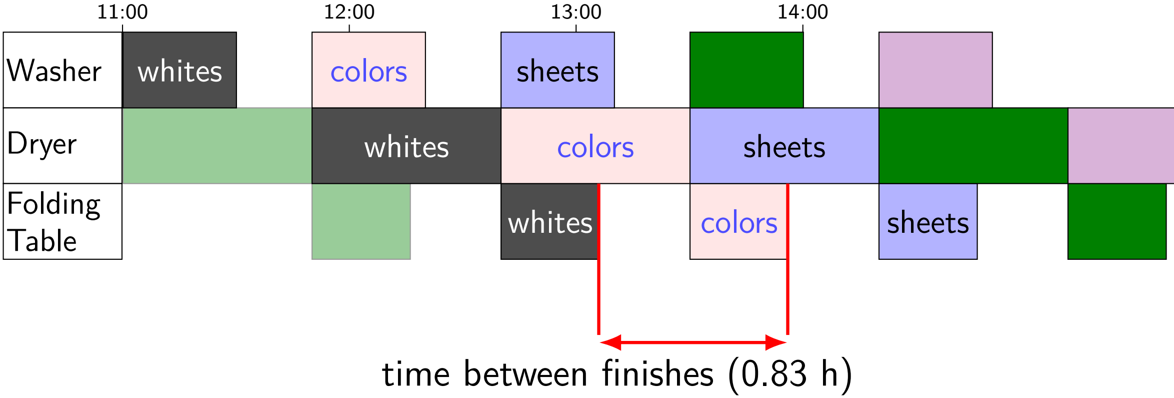
Latency — Time for One



Throughput — Rate of Many

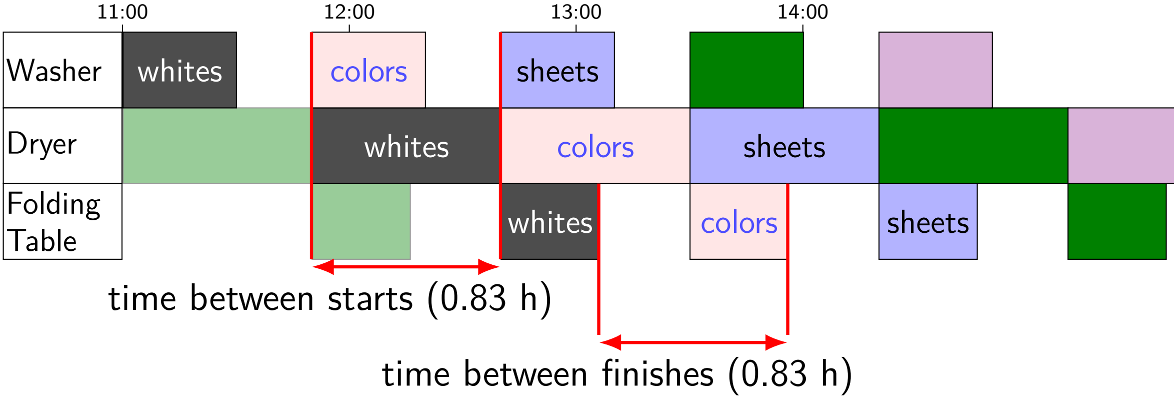


Throughput — Rate of Many



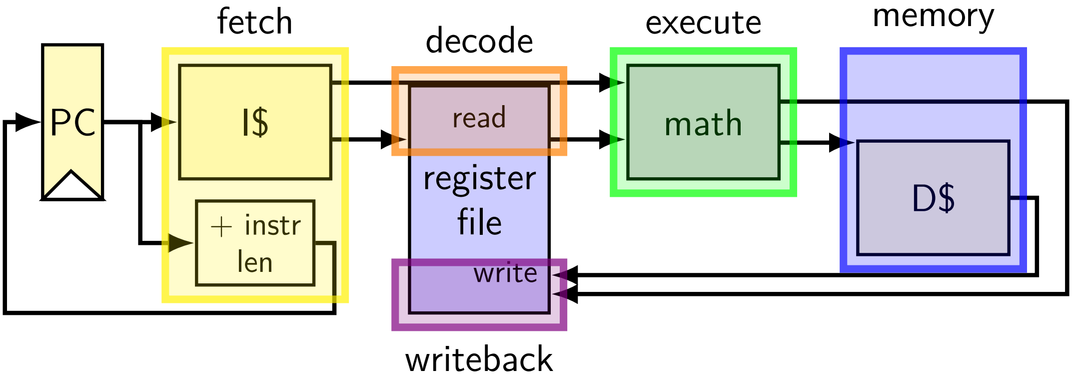
$$\frac{1\ \text{load}}{0.83\text{h}} = 1.2\ \text{loads/h}$$

Throughput — Rate of Many



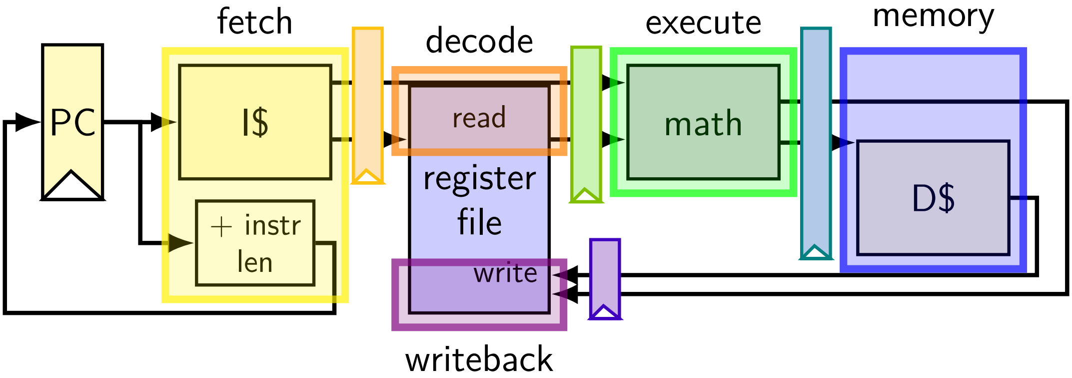
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

adding stages (one way)



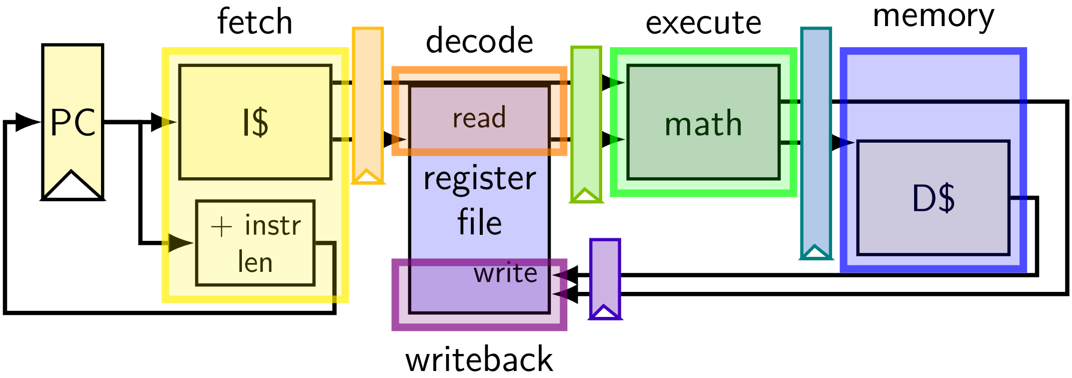
divide running instruction into steps
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

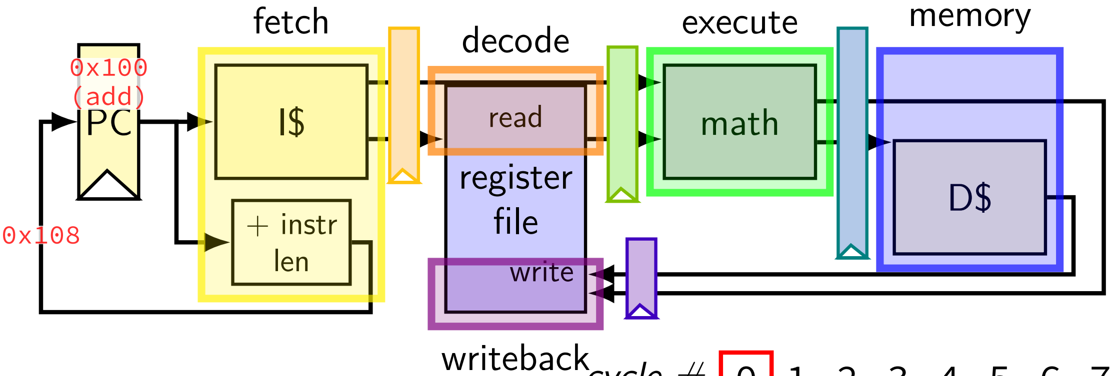


add 'pipeline registers' to hold values from instruction

running some instructions



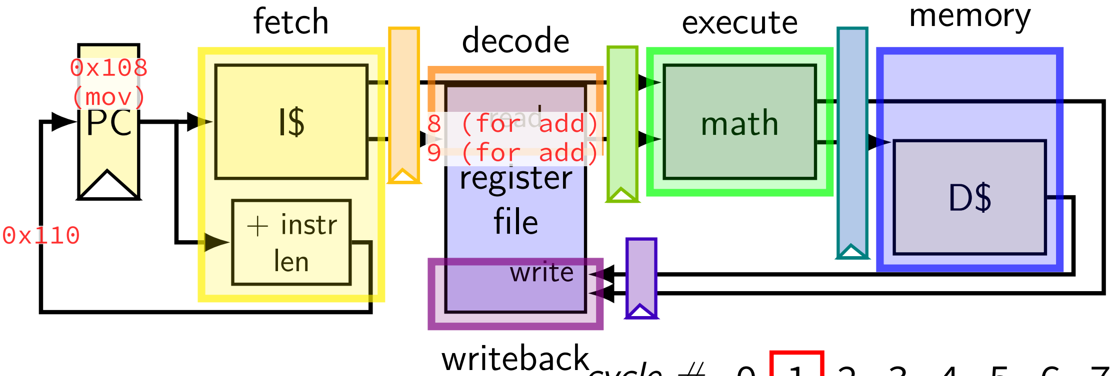
running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

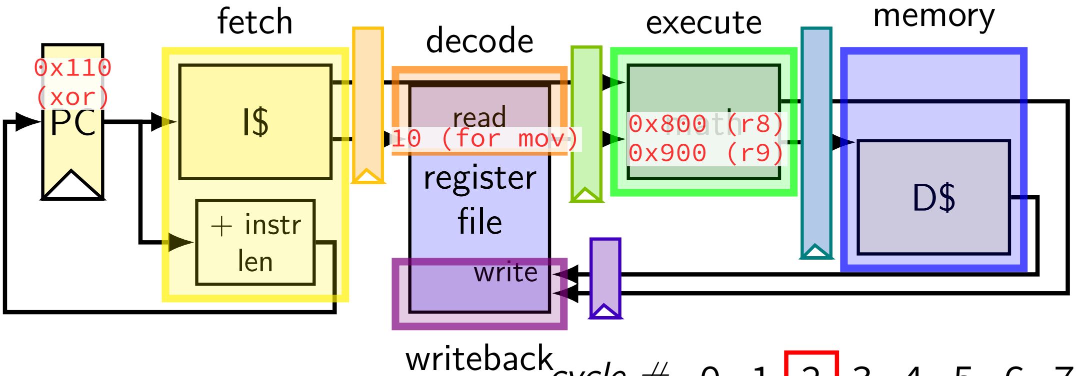
cycle #	0	1	2	3	4	5	6	7	8
0x100	F	D	E	M	W				
0x108		F	D	E	M	W			
0x110			F	D	E	M	W		

running some instructions



	cycle #	0	1	2	3	4	5	6	7	8
`0x100: add %r8, %r9`		F	D	E	M	W				
`0x108: mov 0x1234(%r10), %r11`			F	D	E	M	W			
`0x110: xor %r12, %r13`				F	D	E	M	W		

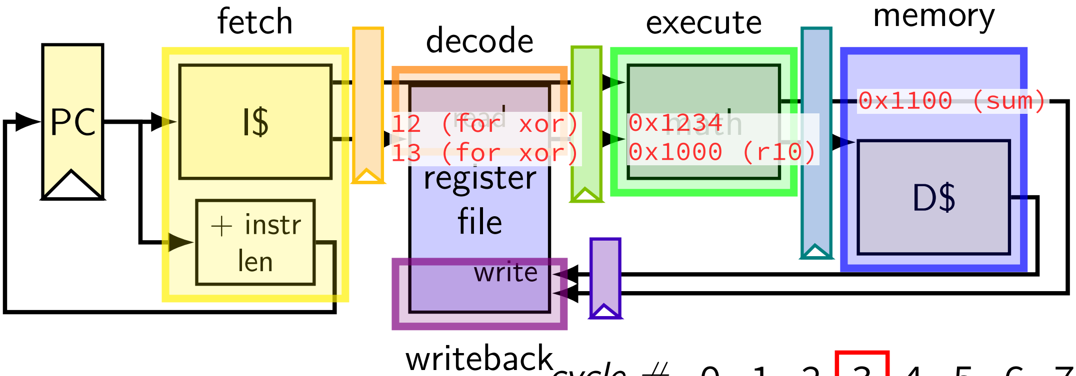
running some instructions




```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

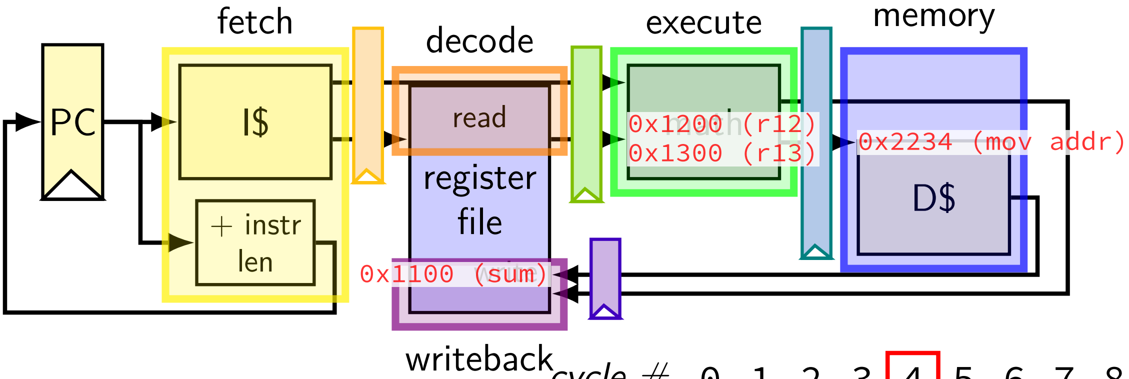
cycle #	0	1	2	3	4	5	6	7	8
0x100	F	D	E	M	W				
0x108		F	D	E	M	W			
0x110			F	D	E	M	W		

running some instructions



cycle #	0	1	2	3	4	5	6	7	8
`0x100: add %r8, %r9`	F	D	E	M	W				
`0x108: mov 0x1234(%r10), %r11`		F	D	E	M	W			
`0x110: xor %r12, %r13`			F	D	E	M	W		

running some instructions



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		

```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: xor %r12, %r13
```

why registers?

example: fetch/decode

need to store current instruction somewhere …while fetching next one

exercise: throughput/latency (1)

	cycle #	0	1	2	3	4	5	6	7	8
`0x100: add %r8, %r9`		F	D	E	M	W				
`0x108: mov 0x1234(%r10), %r11`			F	D	E	M	W			
`0x110: …`					…					

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
	F	D	E	M	W				
		F	D	E	M	W			
				…					

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: …
```

cycle #	0	1	2	3	4
add	F	D	E	M	W
mov		F	D	E	M
				…	

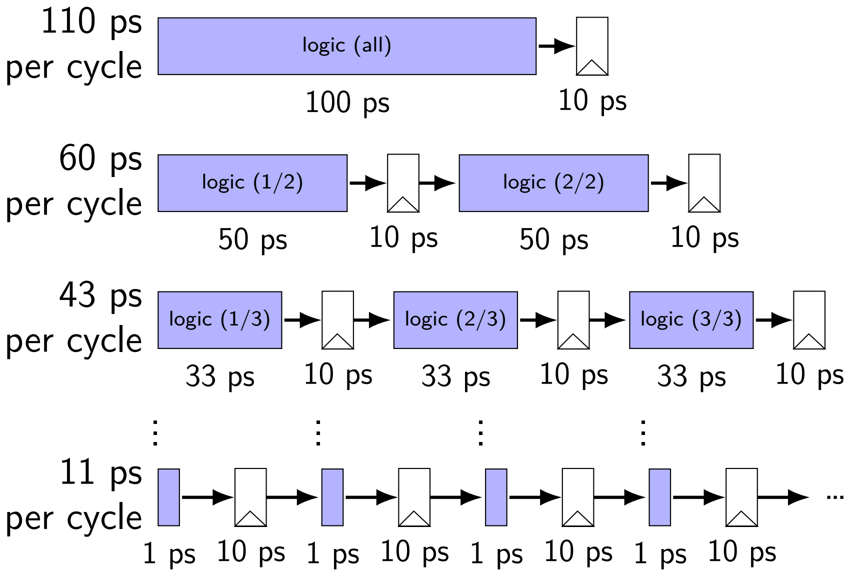
```
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: …
```

cycle #	0	1	2	3	4	5	6	7	8	
add	F1	F2	D1	D2	E1	E2	M1	M2	W1	W
mov		F1	F2	D1	D2	E1	E2	M1	M2	W
				…						

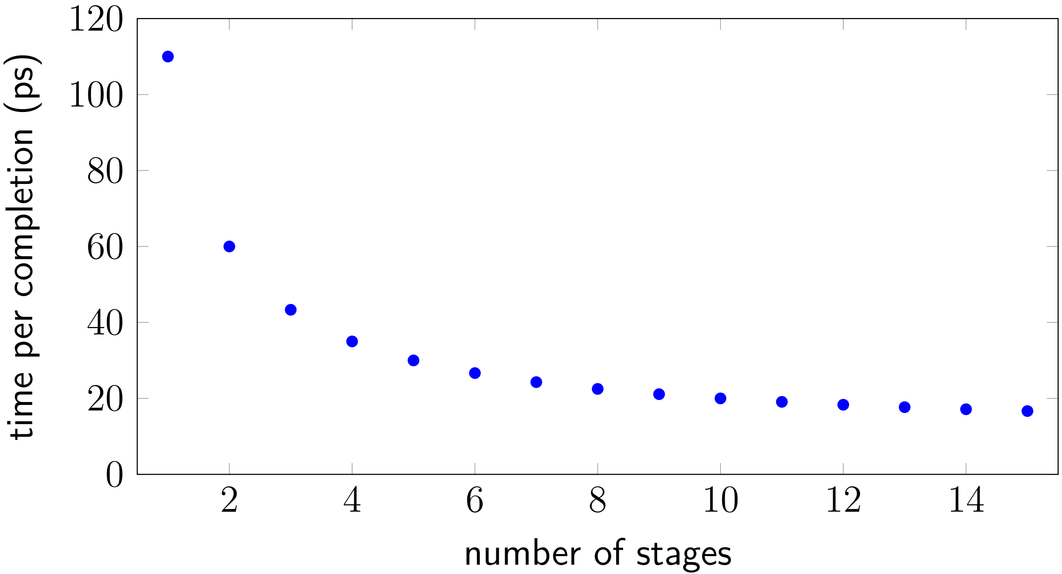
suppose we double number of pipeline stages (to 10) and decrease cycle time from 500 ps to 250 ps

exercise: new throughput?

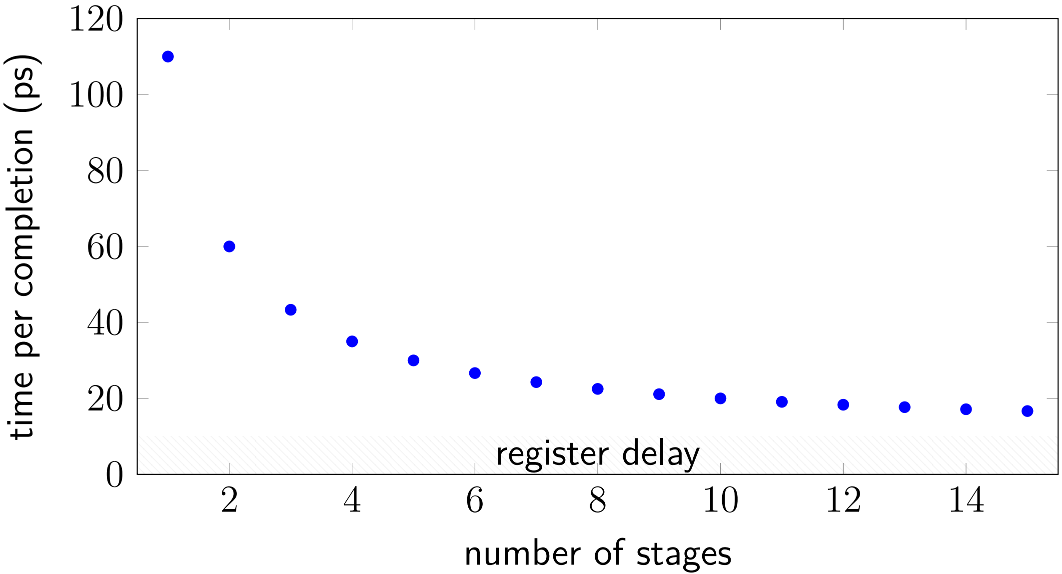
diminishing returns: register delays



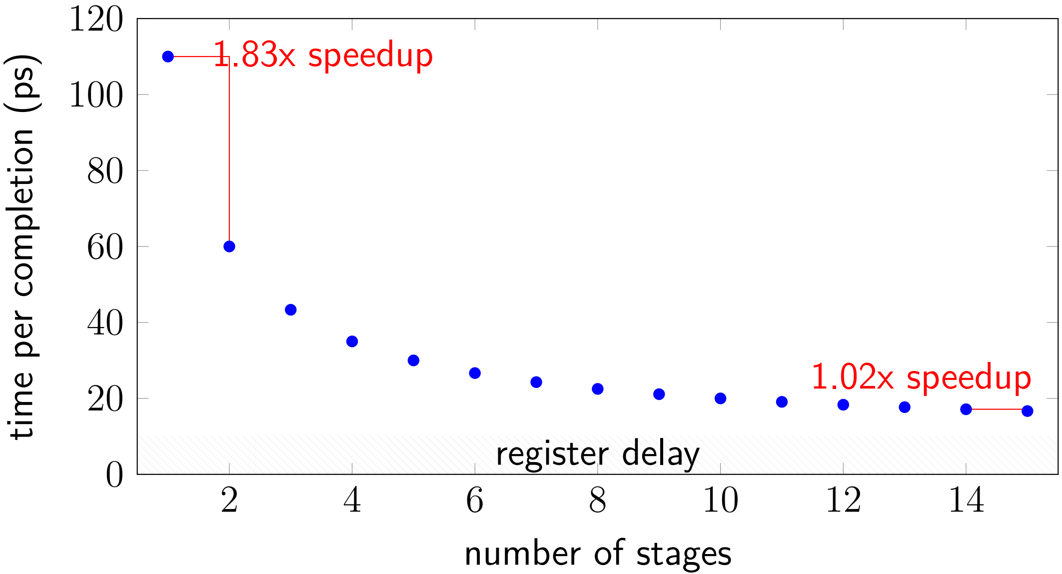
diminishing returns: register delays



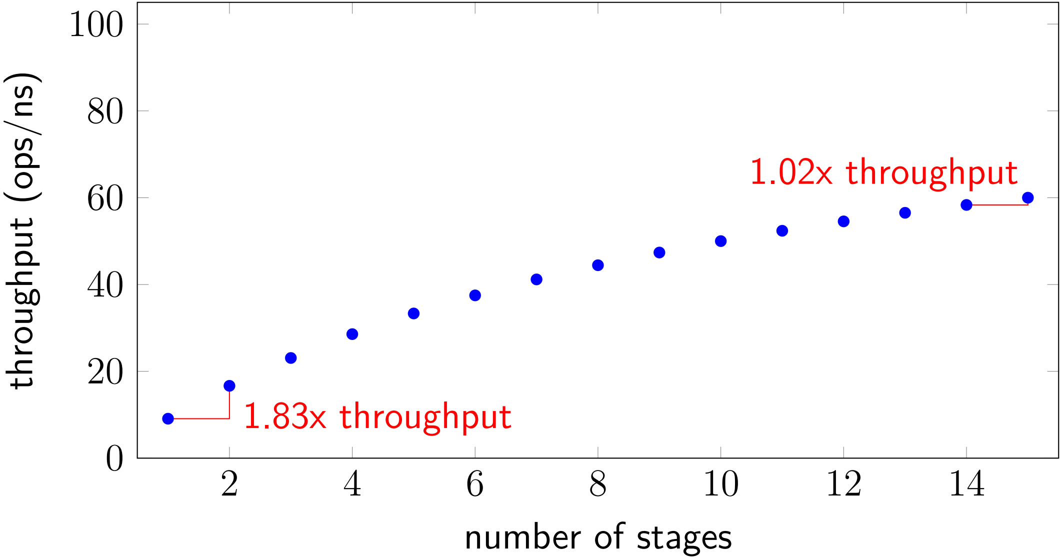
diminishing returns: register delays



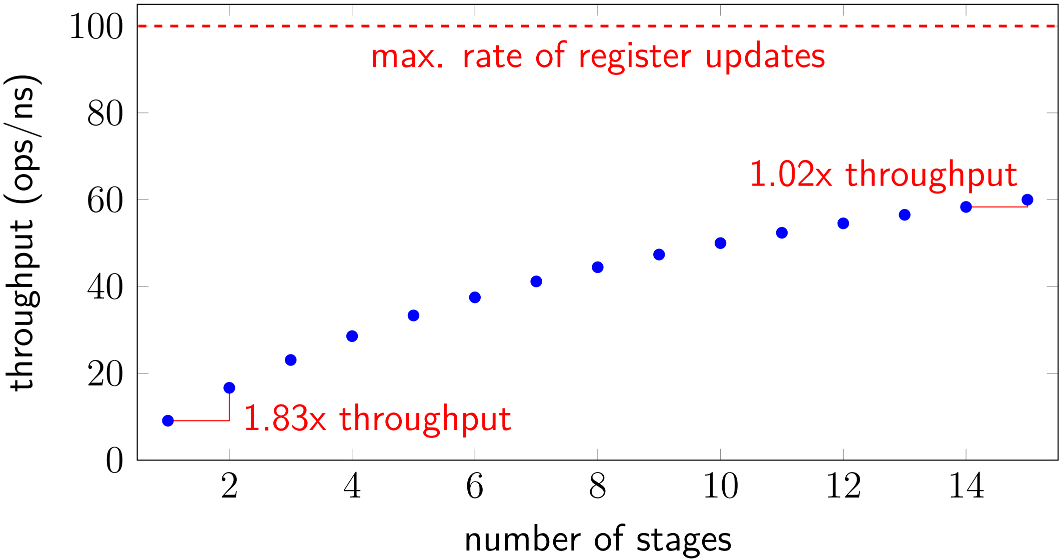
diminishing returns: register delays



diminishing returns: register delays



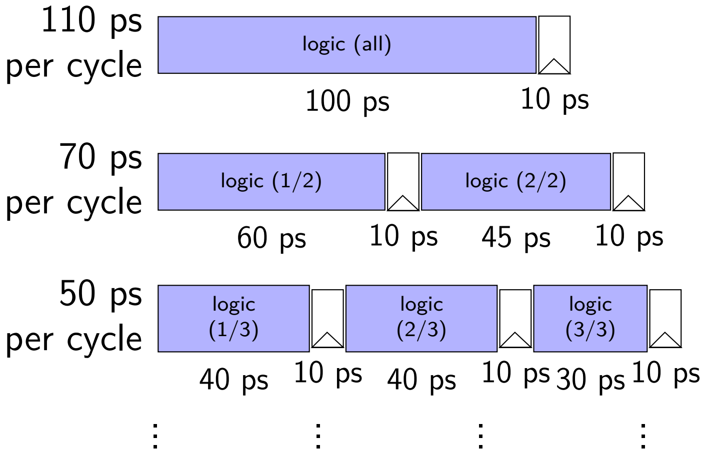
diminishing returns: register delays



diminishing returns: uneven split

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

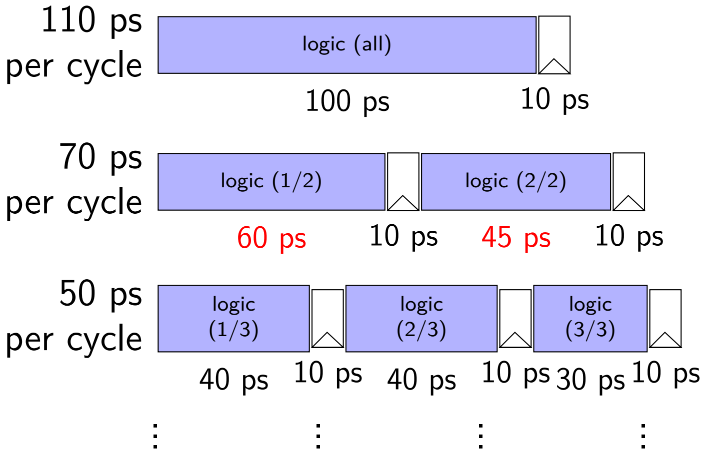
Probably not...



diminishing returns: uneven split

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

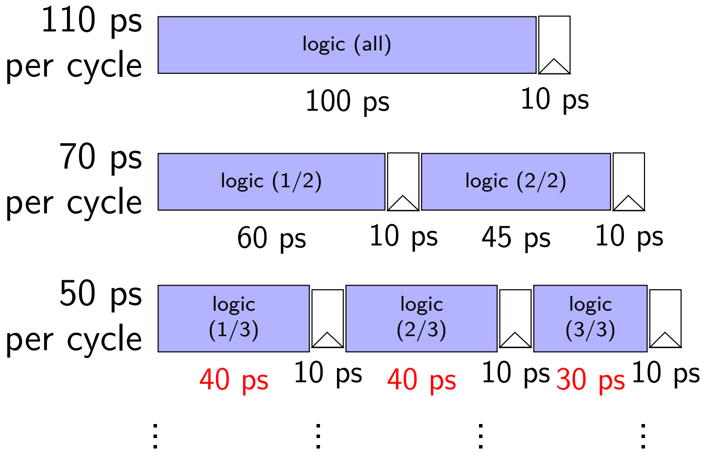
Probably not...



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				$\boxed{900}$	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)

	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	

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

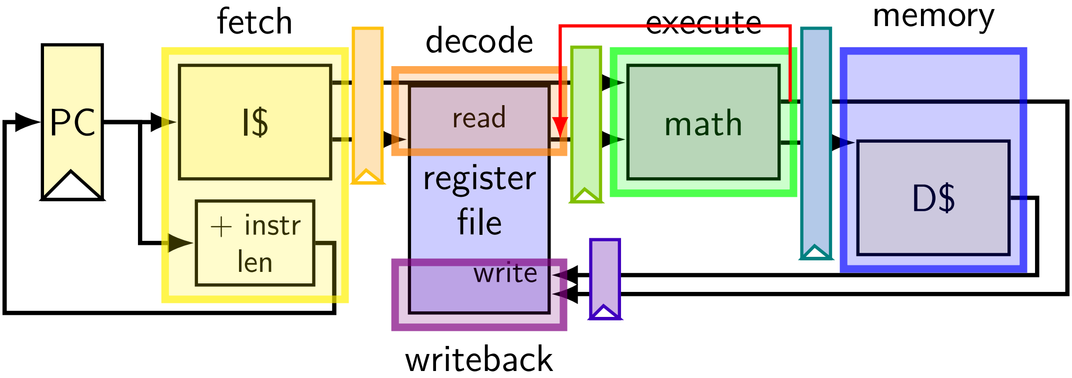
opportunity

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

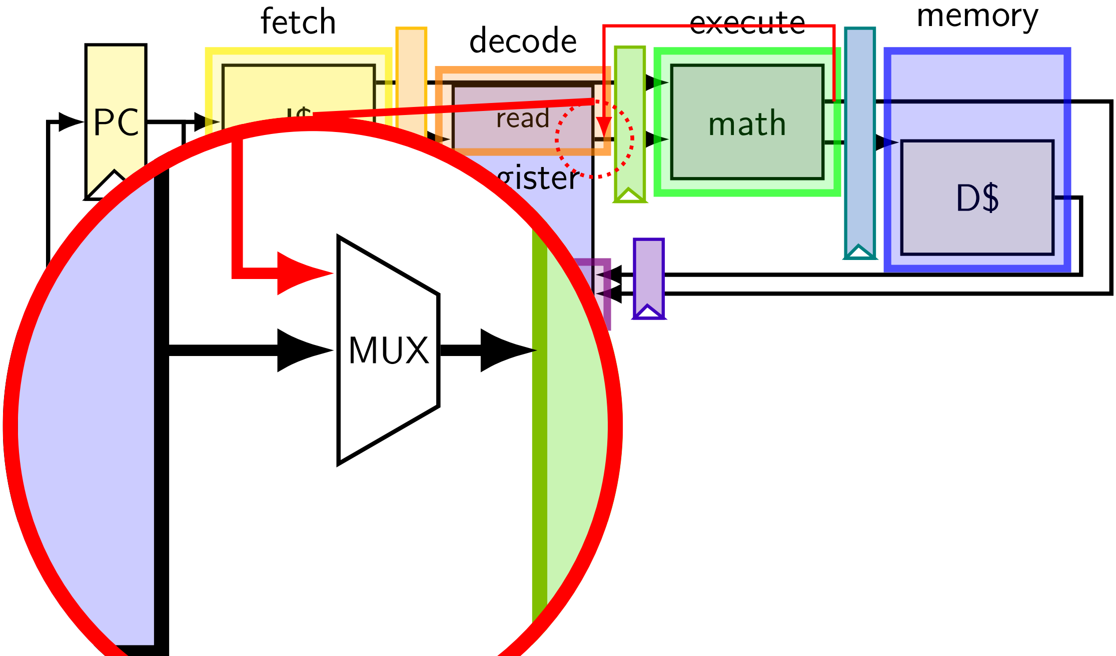
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



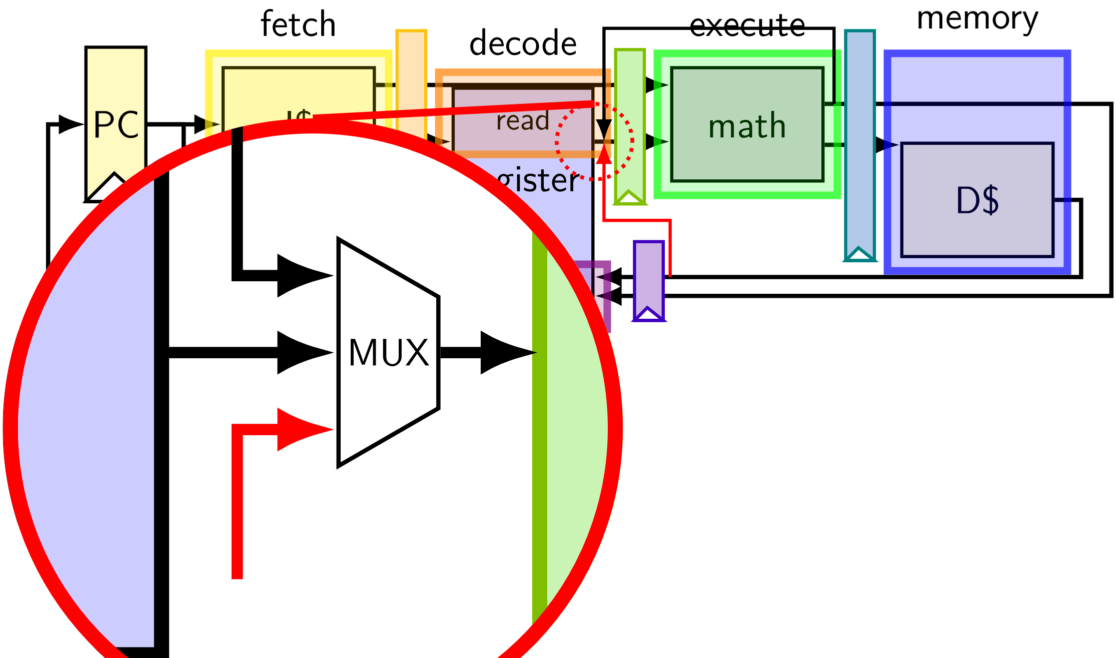
opportunity 2

```
// 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

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

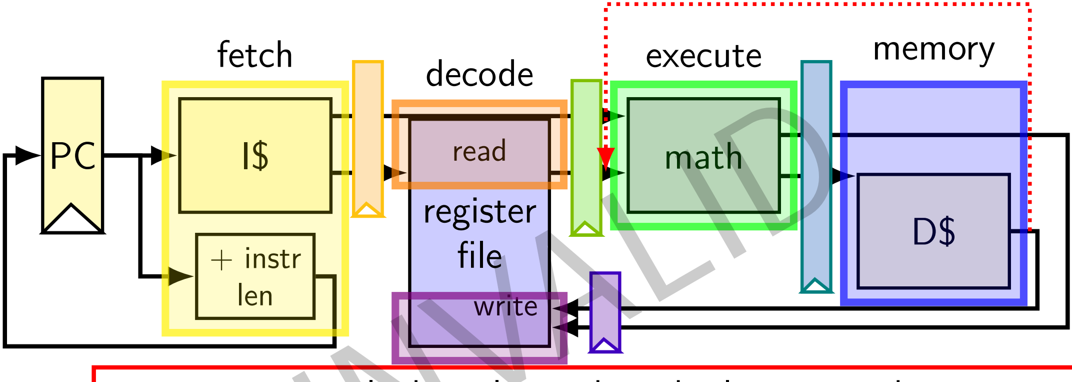
assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

solveable problem

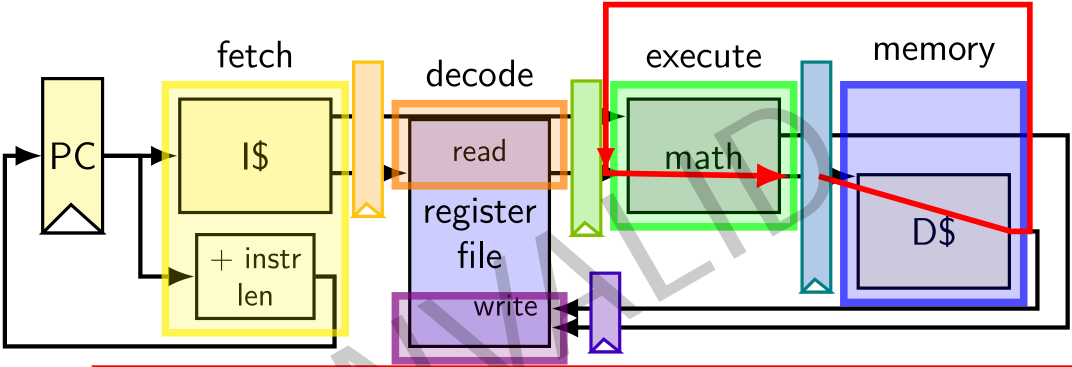
cycle #	0	1	2	3	4	5	6	7	8
`movq 0(%rax), %rbx`	F	D	E	M	W				
`movq %rbx, 0(%rcx)`		F	D	E	M	W			

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

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, %r9`			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

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→writeback	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)
...
```

	cycle # 0	1	2	3	4	5	6	7	8	
`cmpq %r8, %r9`	F	D	$\boxed{\text{E}}$	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	W
...										

compare sets flags

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	compute if jump goes to LABEL			F	D	$\boxed{E}$	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)
...
```

	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`	use computed result					$\boxed{F}$	D	E	M	W	
`movq %r11, 0(%r12)`							F	D	E	M	W
...											

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.

performance

hypothetical instruction mix

kind	portion	cycles (predict not-taken)	cycles (stall)
taken jXX	3%	3	3
non-taken jXX	5%	1	3
others	92%	1*	1*

* — ignoring data hazards

performance

hypothetical instruction mix

kind	portion	cycles (predict not-taken)	cycles (stall)
taken jXX	3%	3	3
non-taken jXX	5%	1	3
others	92%	1*	1*

* — ignoring data hazards

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