

last time (1)

certificates + certificate authorities

cryptographic hashes

- hard-to-reverse summary

- specialized versions for password storage

key exchange

- generate secret + key share

- combine your secret + other's key share to get shared secret

TLS: everything together

last time (2)

review: single-cycle processor

pipelining idea (laundry analogy)

instructions as series of pipeline stages

latency = time for one (beginning to end)

throughput = rate for many

start: diminishing returns with pipelining

6:30p lab tomorrow

is in Olsson 018

anonymous feedback (1)

final exam: could it be remote?

deliberate decision I made early in the semester; has pros/cons

remote: tricky to balance for students not spending N hours on exam

not nice re: technical issues to give tight time limit remotely

remote: need to write questions that work in open-book/notes

anonymous feedback (2)

“Could you give some more examples with pipeline chart and a quick review of assembly again.”

we will have more examples with pipeline chart, b/c there are parts of pipelining we haven't talked about

re: (x86-64) assembly, not going to do a detailed review re: time but...

instruction operand=source, operand=source+destination

%XXX — some register (%rXX = 64 bits, %eXX = 32 bits)

\$123 — the constant 123

some_label — label = memory location

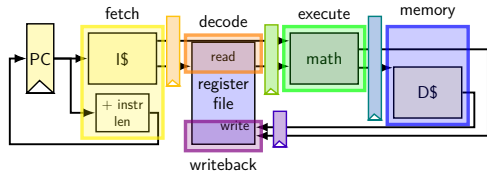
X(%rYY) = memory[X + %rYY]

cmp = set condition codes; jXX = jump based on condition codes

addq processor timing

// init. %r8=800, %r9=900, etc.

```
addq %r8, %r9    // R8+R9->R9
addq %r10, %r11  // R10+R11->R11
addq %r12, %r13  // R12+R13->R13
addq %r14, %r15  // R14+R15->R15
addq %r9, %r8    // R9+R8->R8
```

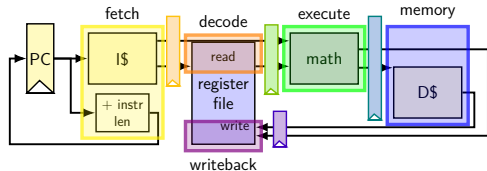


	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

addq processor timing

// init. %r8=800, %r9=900, etc.

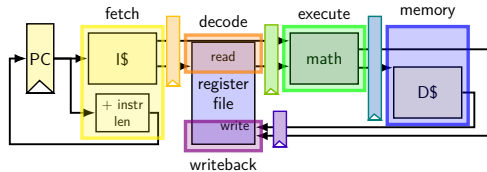
```
addq %r8, %r9    // R8+R9->R9
addq %r10, %r11  // R10+R11->R11
addq %r12, %r13  // R12+R13->R13
addq %r14, %r15  // R14+R15->R15
addq %r9, %r8    // R9+R8->R8
```



	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	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

addq processor timing

```
// init. %r8=800, %r9=900, etc.
addq %r8, %r9    // R8+R9->R9
addq %r10, %r11  // R10+R11->R11
addq %r12, %r13  // R12+R13->R13
addq %r14, %r15  // R14+R15->R15
addq %r9, %r8    // R9+R8->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	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

addq processor timing

// init. %r8=800, %r9=900, etc.

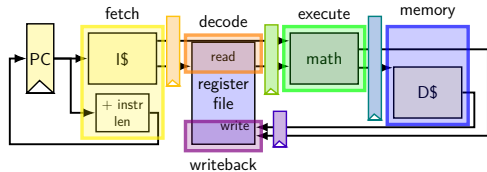
addq %r8, %r9 // $R8+R9 \rightarrow R9$

addq %r10, %r11 // $R10+R11 \rightarrow R11$

addq %r12, %r13 // $R12+R13 \rightarrow R13$

addq %r14, %r15 // $R14+R15 \rightarrow R15$

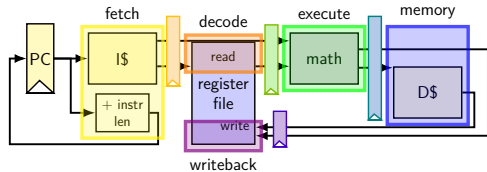
addq %r9, %r8 // $R9+R8 \rightarrow R8$



	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	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

addq processor timing

```
// init. %r8=800, %r9=900, etc.
addq %r8, %r9    // R8+R9->R9
addq %r10, %r11  // R10+R11->R11
addq %r12, %r13  // R12+R13->R13
addq %r14, %r15  // R14+R15->R15
addq %r9, %r8    // R9+R8->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	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

addq processor timing

// init. %r8=800, %r9=900, etc.

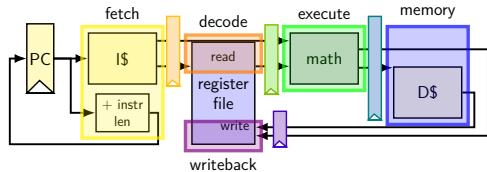
addq %r8, %r9 // $R8 + R9 \rightarrow R9$

addq %r10, %r11 // $R10 + R11 \rightarrow R11$

addq %r12, %r13 // $R12 + R13 \rightarrow R13$

addq %r14, %r15 // $R14 + R15 \rightarrow R15$

addq %r9, %r8 // $R9 + R8 \rightarrow R8$



	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	0x4	10	11	800	900	9				
3	0x6	12	13	1000	1100	11	1700	9		
4	0x8	14	15	1200	1300	13	2100	11	1700	9
5		9	8	1400	1500	15	2500	13	2100	11
6				900	1700	8	2900	15	2500	13
7							2500	8	2900	15
8									2500	8

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)

	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 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
	F	D	E	M	W
		F	D	E	M
				...	

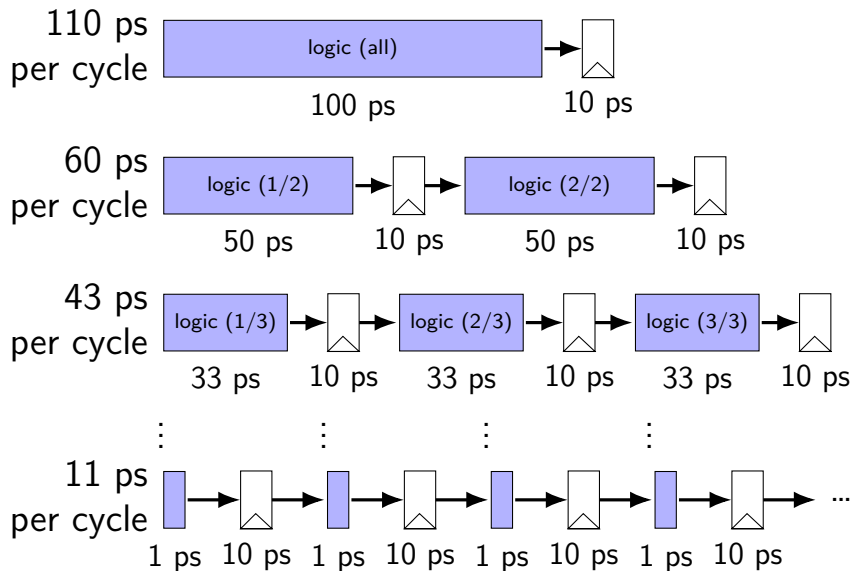
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...

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

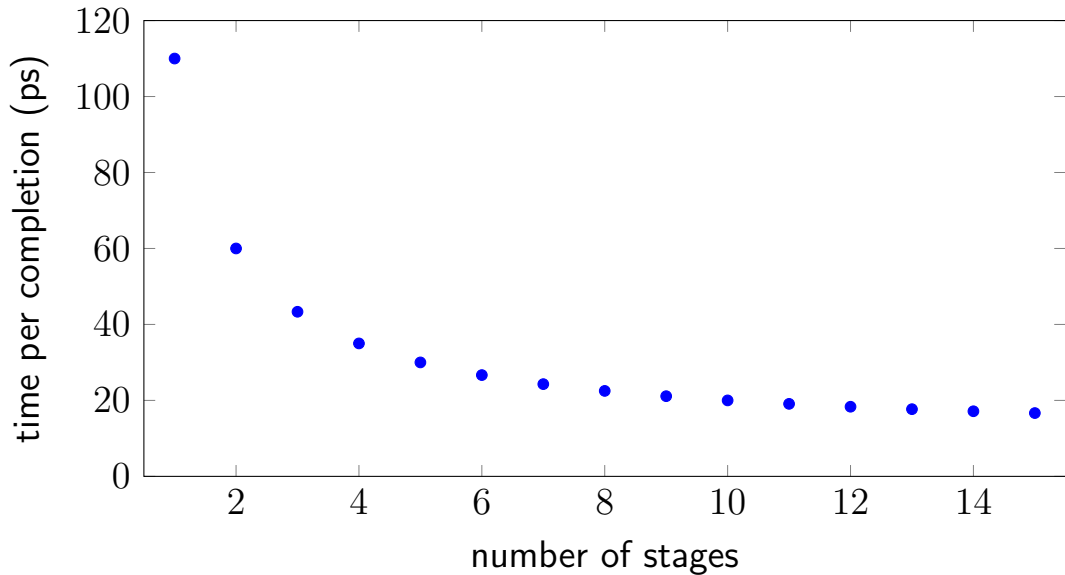
double number of pipeline stages (to 10) + decrease cycle time
from 500 ps to 250 ps — throughput?

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

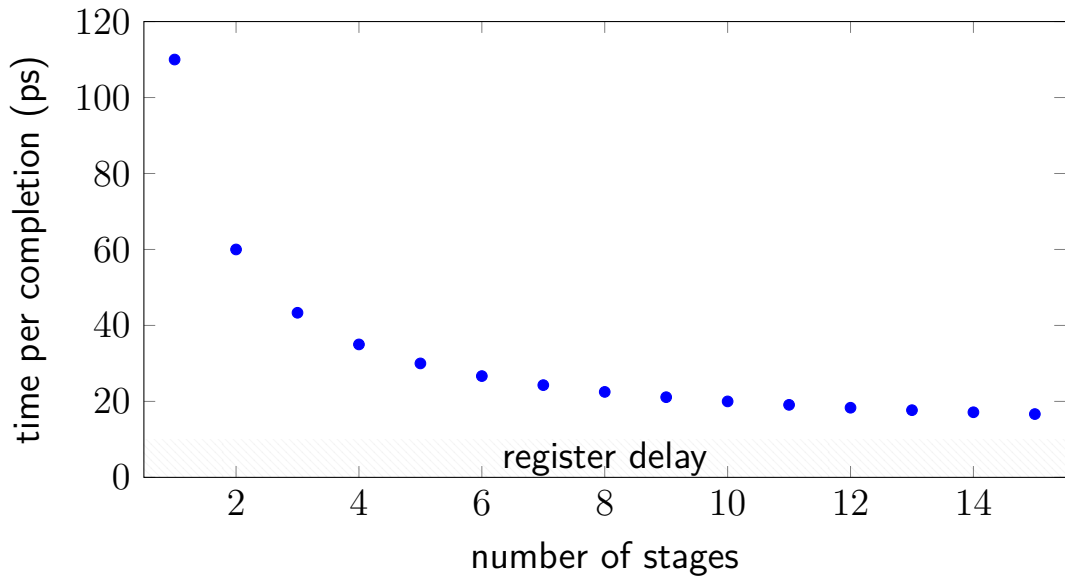
diminishing returns: register delays



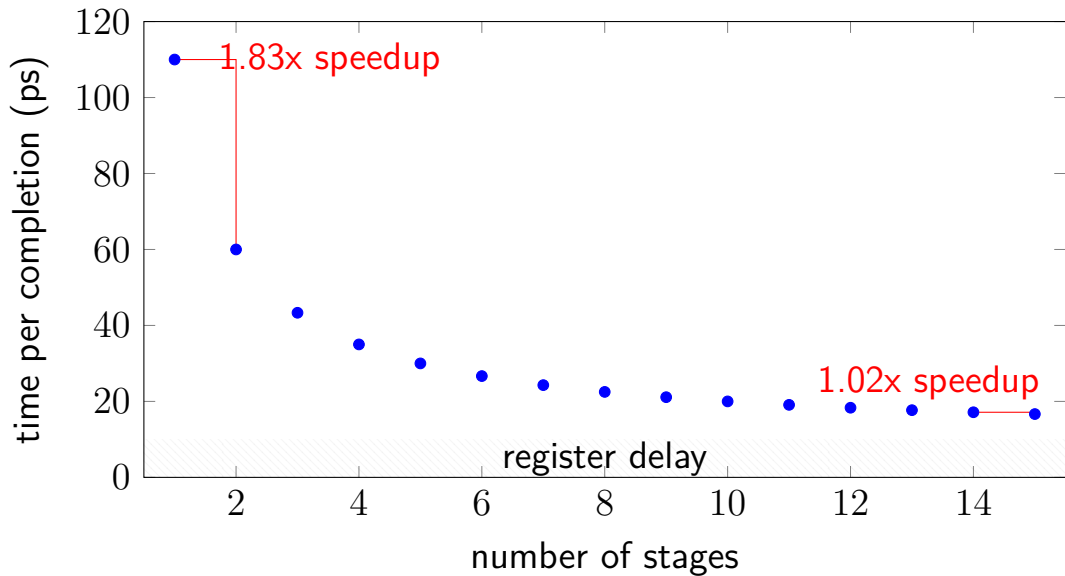
diminishing returns: register delays



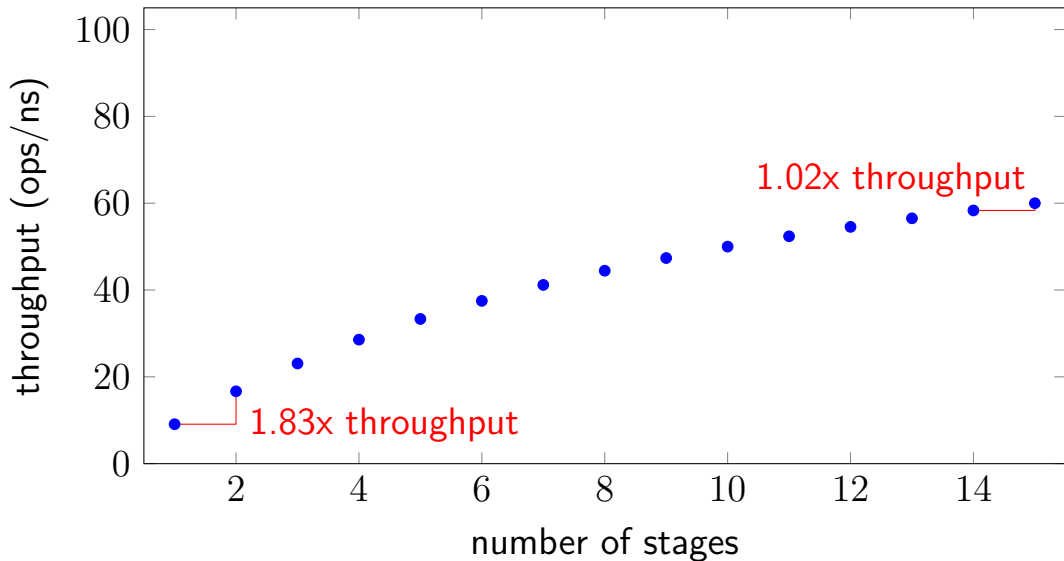
diminishing returns: register delays



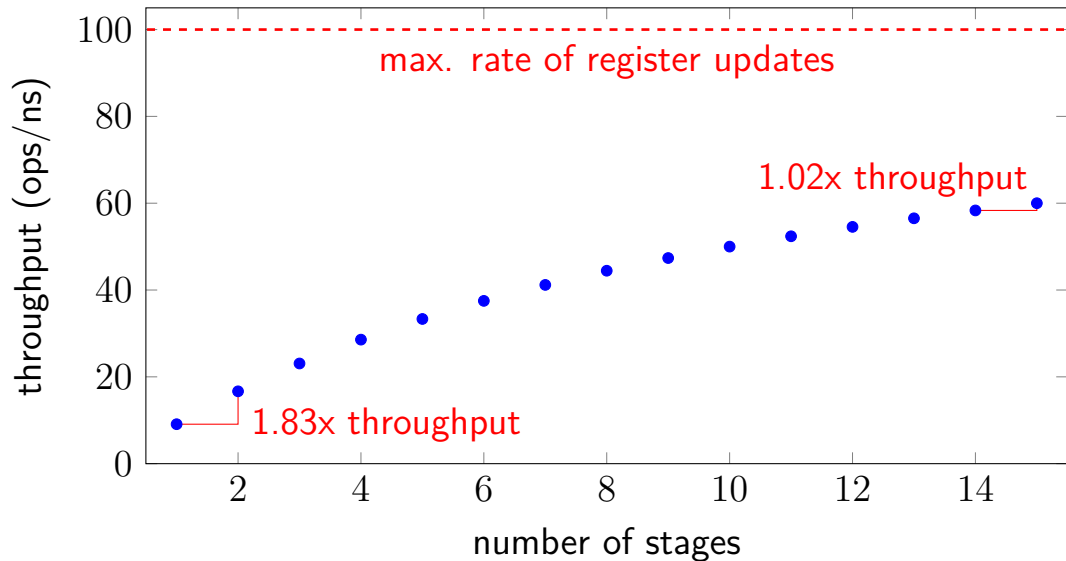
diminishing returns: register delays



diminishing returns: register delays



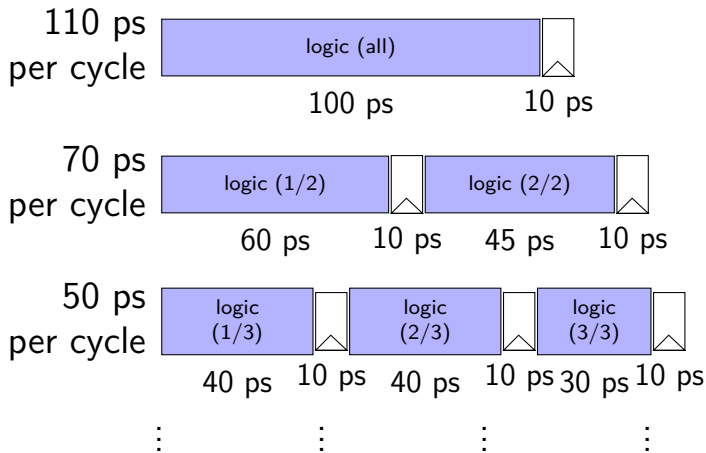
diminishing returns: register delays



diminishing returns: uneven split

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

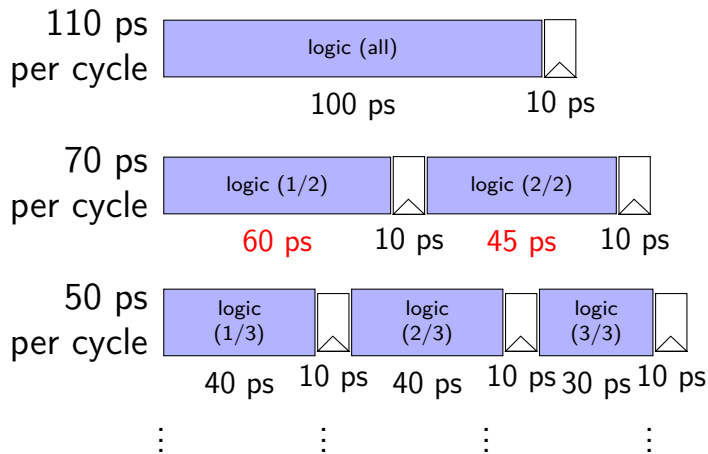
Probably not...



diminishing returns: uneven split

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

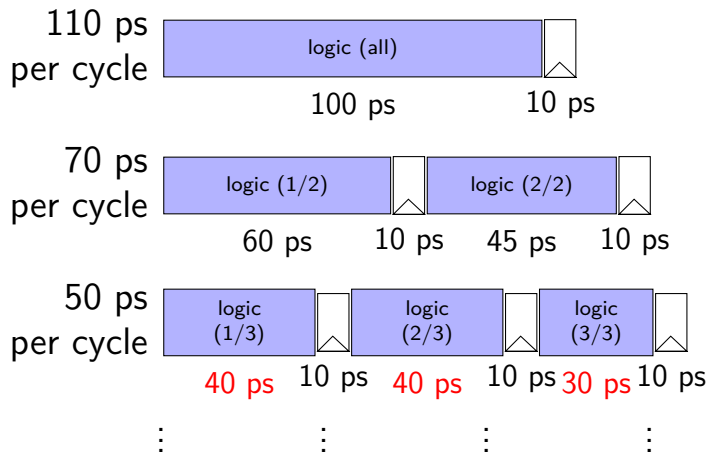
Probably not...



diminishing returns: uneven split

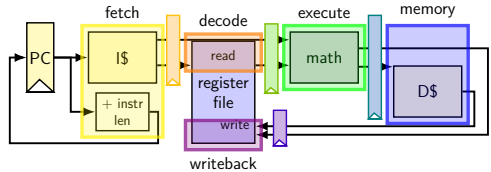
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...



a data hazard

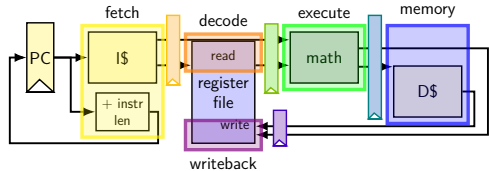
```
// initially %r8 = 800,
//                %r9 = 900, etc.
addq %r8, %r9    // R8 + R9 -> R9
addq %r9, %r8    // R9 + R8 -> R9
addq ...
addq ...
```



	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 hardware solution

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

how about hardware add nops?

called **stalling**

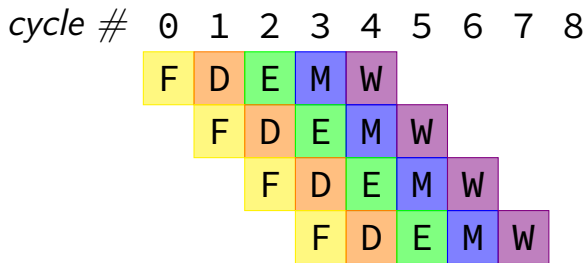
extra logic:

- 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



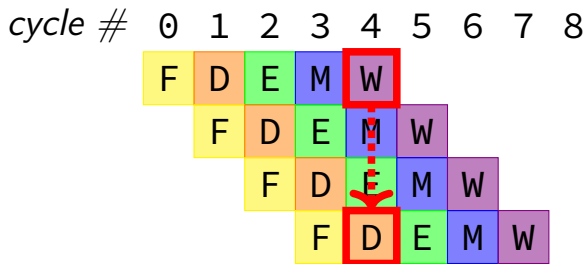
stalling/nop pipeline diagram (1)

add %r8, %r9

(nop)

(nop)

addq %r9, %r8



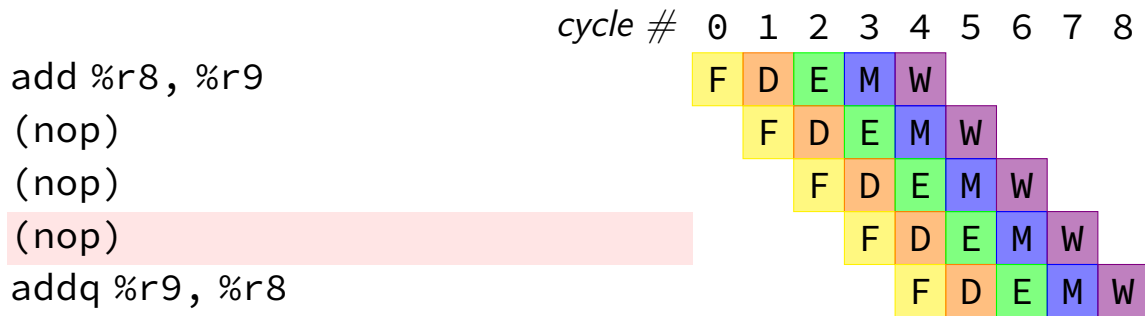
assumption:

if writing register value

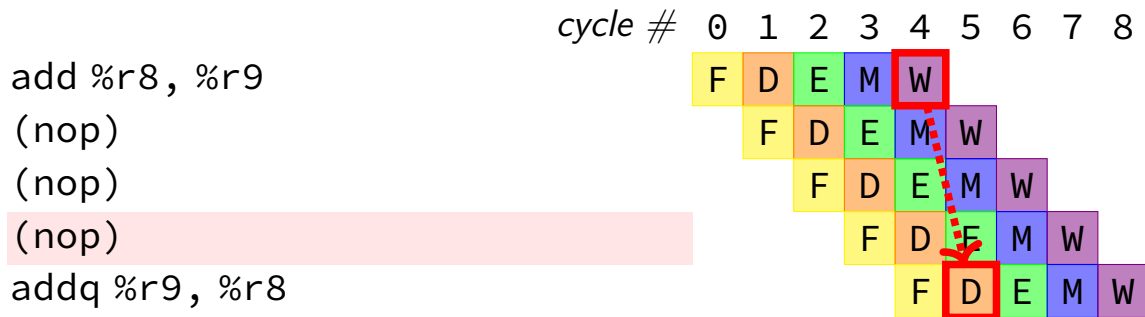
register file will return that value for reads

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)



stalling/nop pipeline diagram (2)



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
```

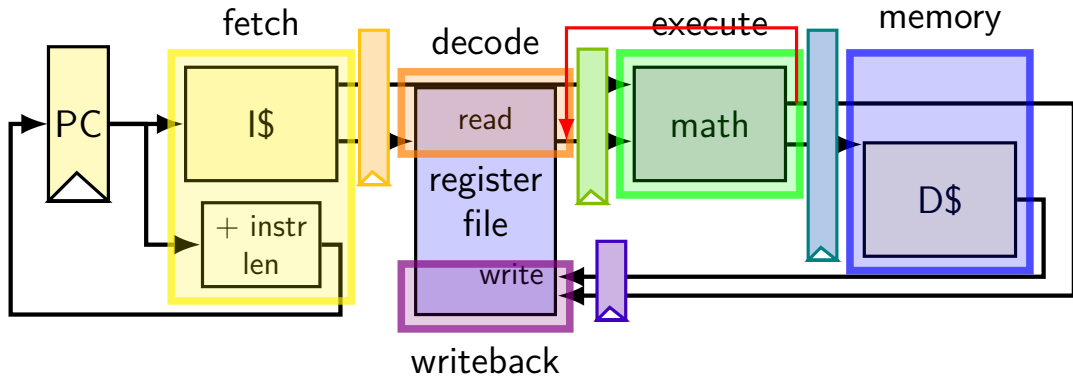
```
0x2: addq %r9, %r8
```

...

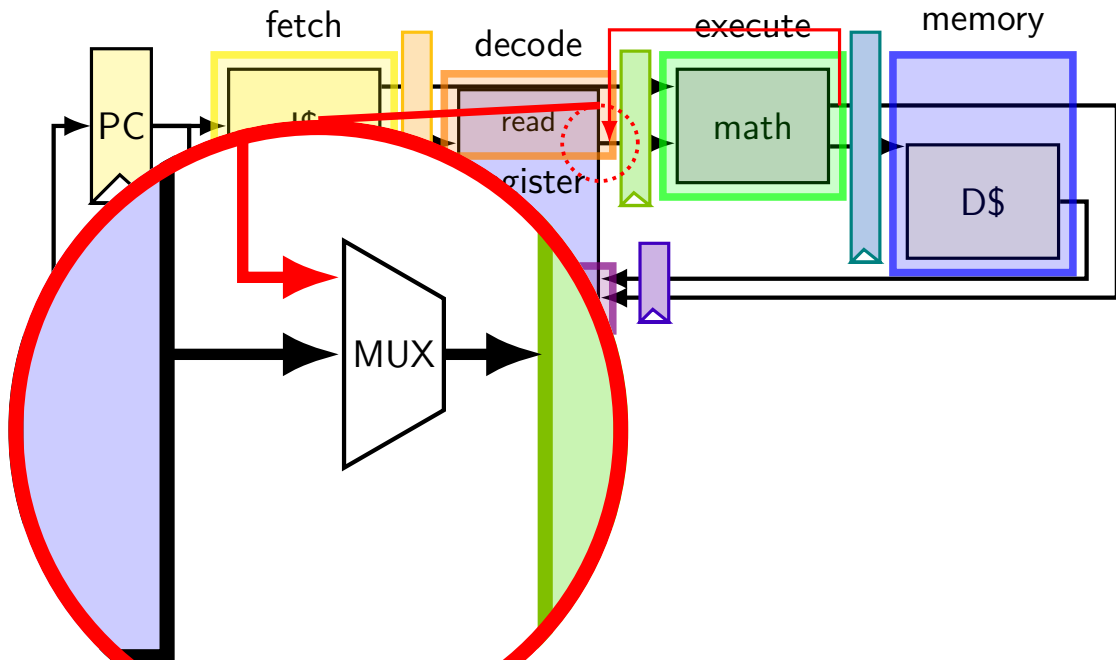
	fetch	fetch/decode		decode/execute			execute/memory		memory/writeback	
cycle	PC	rA	rB	R[rB]	R[rB]	rB	sum	rB	sum	rB
0	0x0									
1	0x2	8	9							
2		9	8	800	900	9				
3				900	800	8	1700	9		
4							1700	8	1700	9
5									1700	8

should be 1700

exploiting the opportunity



exploiting the opportunity



opportunity 2

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

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

```
0x2: nop
```

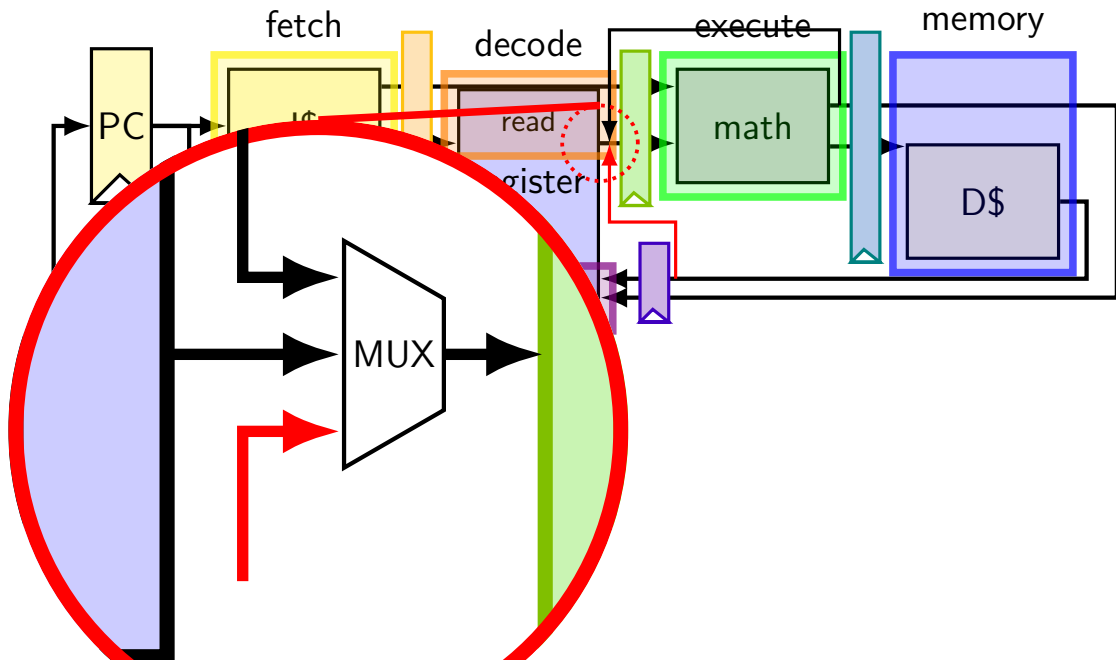
```
0x3: addq %r9, %r8
```

```
...
```

	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

subq %r8, %r10

xorq %r8, %r9

andq %r9, %r8

F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	

in subq, %r8 is _____ addq.

in xorq, %r9 is _____ addq.

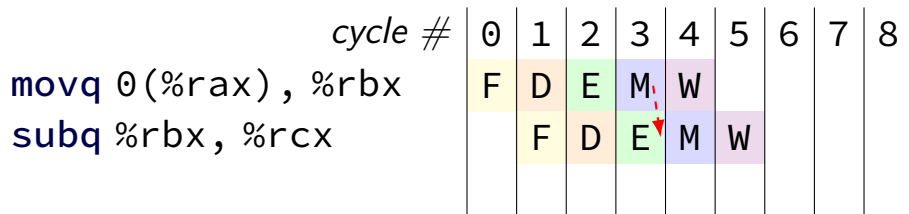
in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

A: not forwarded from

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

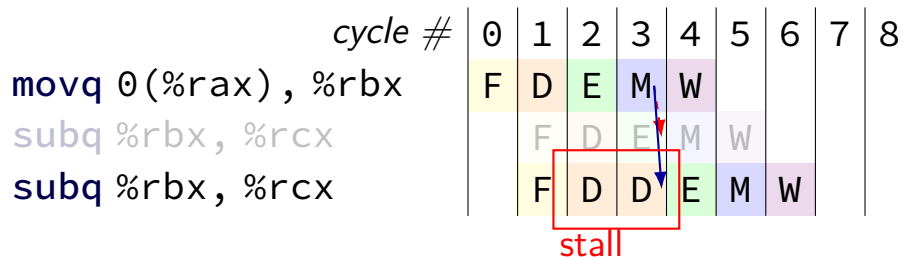
unsolved problem



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



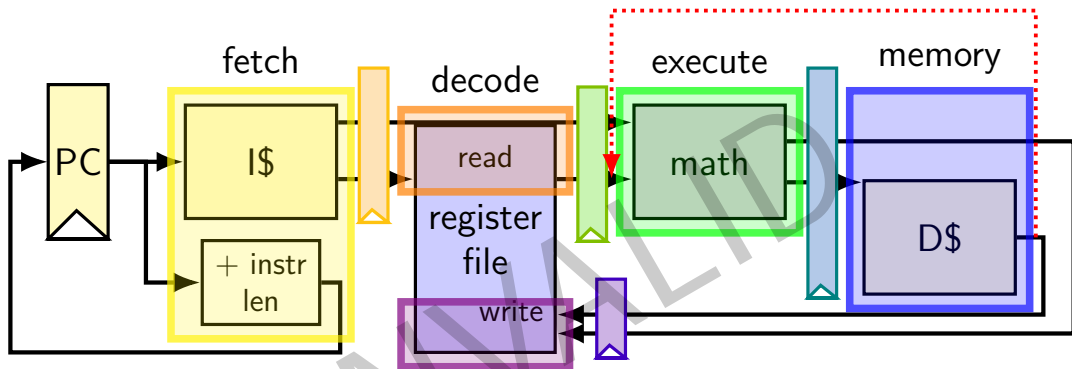
combine stalling and forwarding to resolve hazard

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

solveable problem

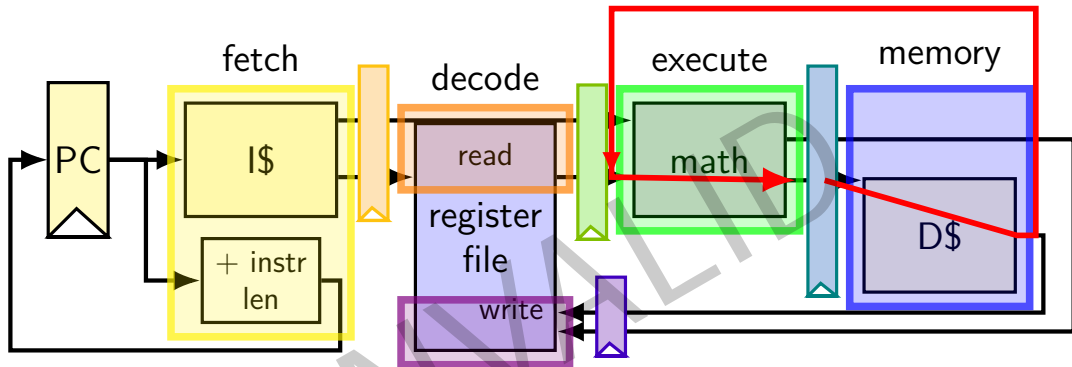
	cycle #	0	1	2	3	4	5	6	7	8
movq 0(%rax), %rbx		F	D	E	M	W				
movq %rbx, 0(%rcx)			F	D	E	M	W			

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...



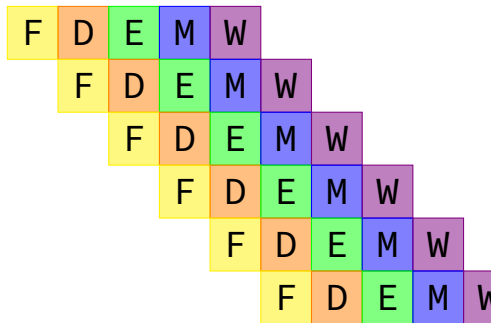
clock cycle needs to be long enough
to go through data cache AND
to go through math circuits!
(which we were trying to avoid by putting them in separate stages)

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)
...
```

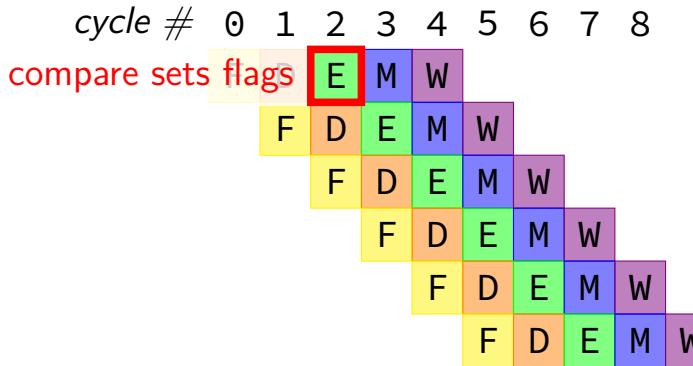
cycle # 0 1 2 3 4 5 6 7 8



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)
...
```



jXX: stalling?

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

```
...
```

```
cmpq %r8, %r9
```

```
jne LABEL
```

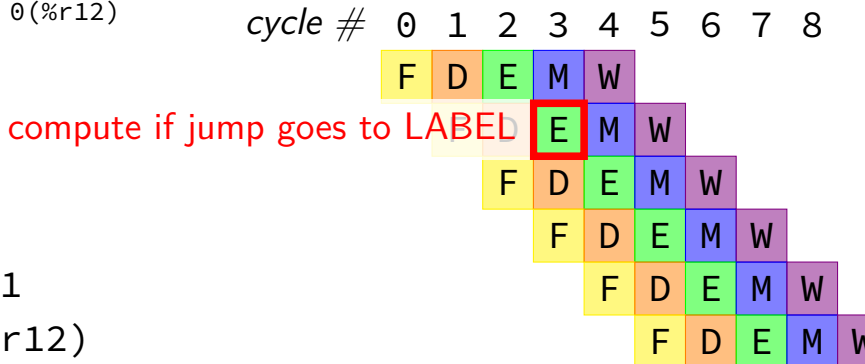
```
(do nothing)
```

```
(do nothing)
```

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

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

```
...
```

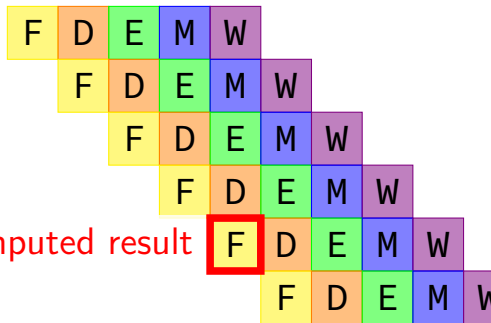


jXX: stalling?

```
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



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

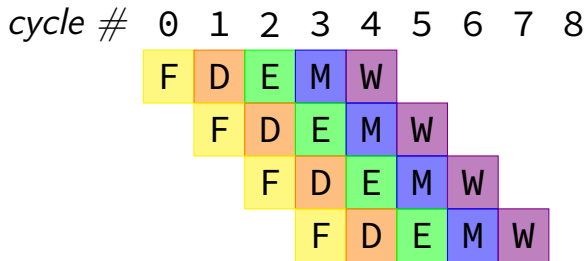
right: 2 cycles faster!; wrong: undo guess before too late

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

cycle # 0 1 2 3 4 5 6 7 8

cmpq %r8, %r9

jne LABEL

xorq %r10, %r11

(inserted nop)

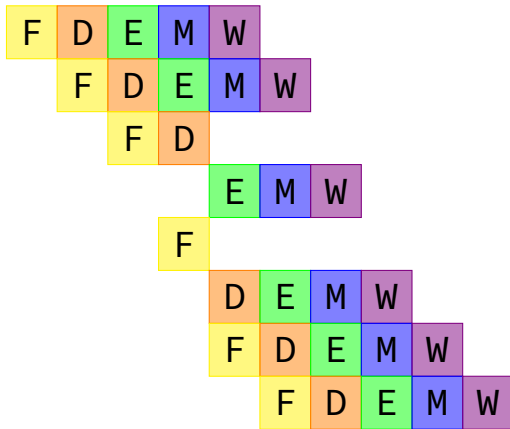
movq %r11, 0(%r12)

(inserted nop)

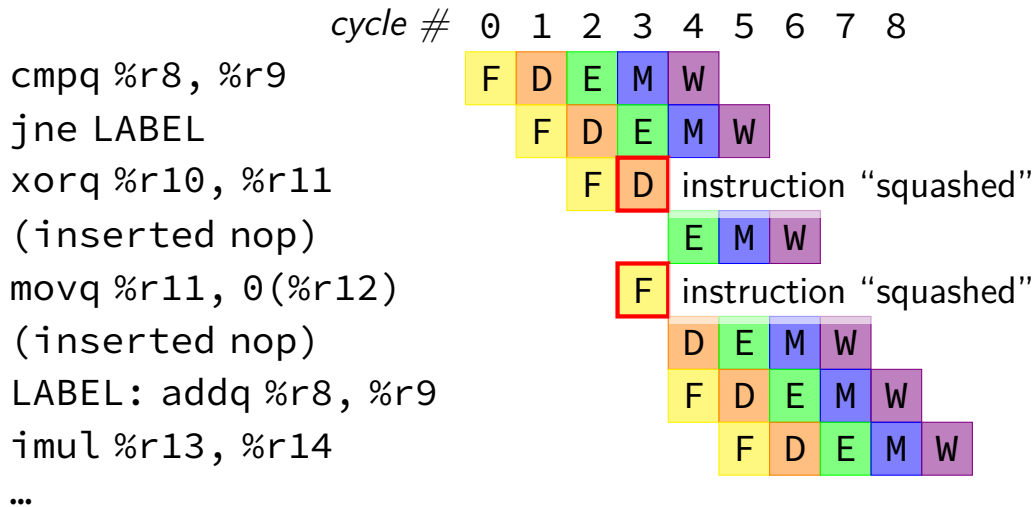
LABEL: addq %r8, %r9

imul %r13, %r14

...



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.

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*

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*

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**

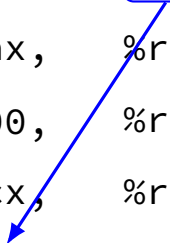
where are dependencies?

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

The diagram illustrates data dependencies between instructions. A blue arrow points from the `%rbx` operand in the first instruction (`addq %rax, %rbx`) to the `%rbx` operand in the fifth instruction (`addq %rbx, %r10`). A red arrow points from the `%rcx` operand in the third instruction (`movq $100, %rcx`) to the `%rcx` operand in the fourth instruction (`addq %rcx, %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 --> I4[addq %rcx, %r10]; I3[movq $100, %rcx] -- red --> I4; I4 -- red --> I5[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

	<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>

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>

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>
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>

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) 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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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

	<i>cycle #</i>	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→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

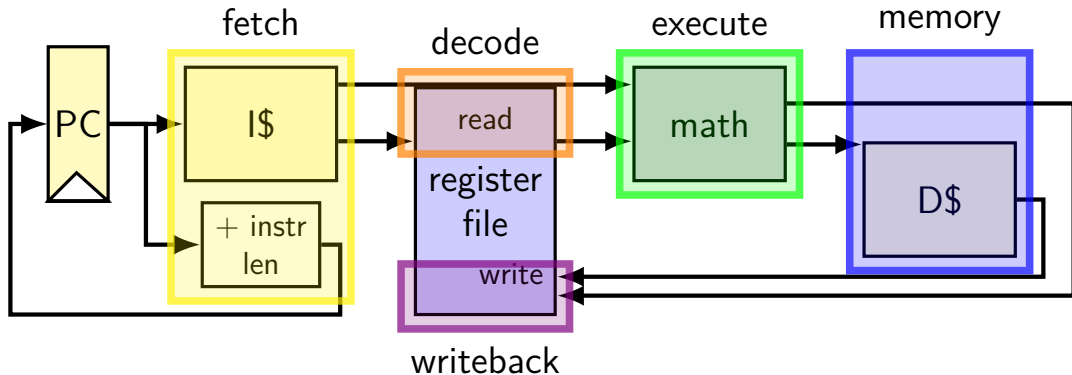
0x10: addq %r10, %r11

	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

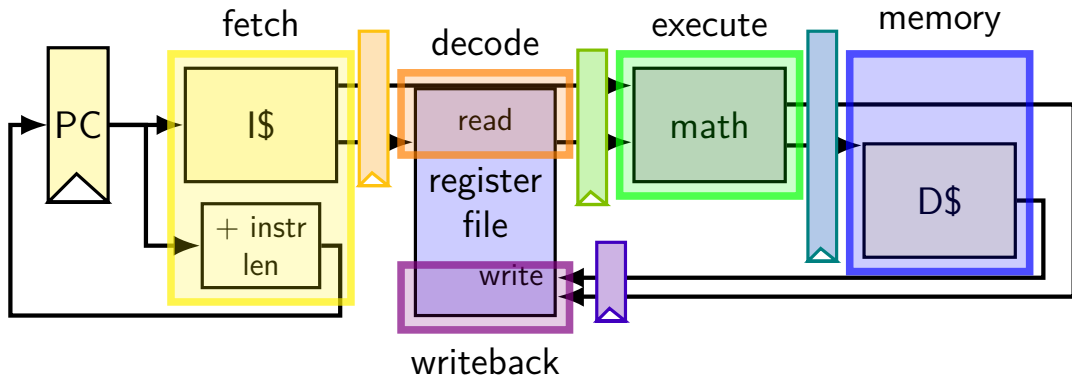
backup slides

adding stages (one way)



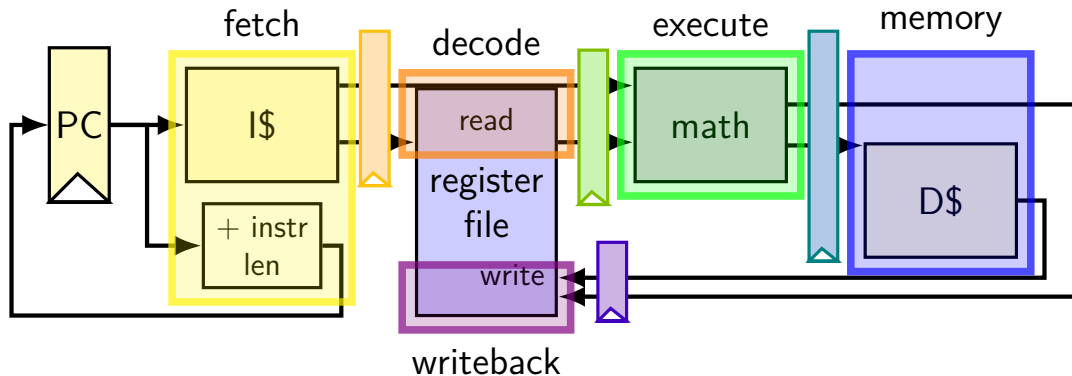
divide running instruction into steps
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

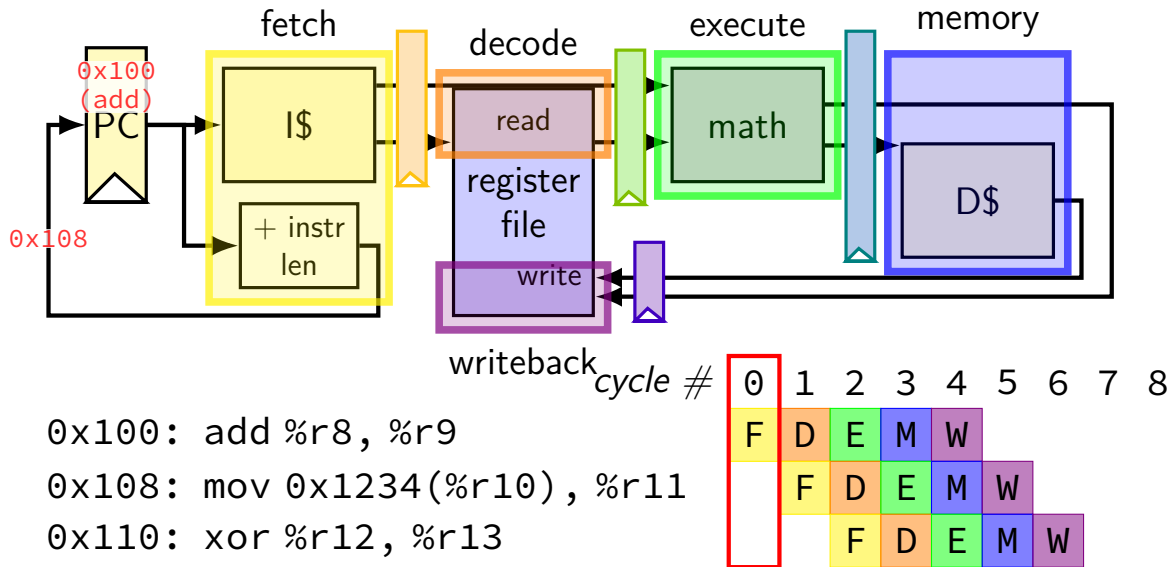


add 'pipeline registers' to hold values from instruction

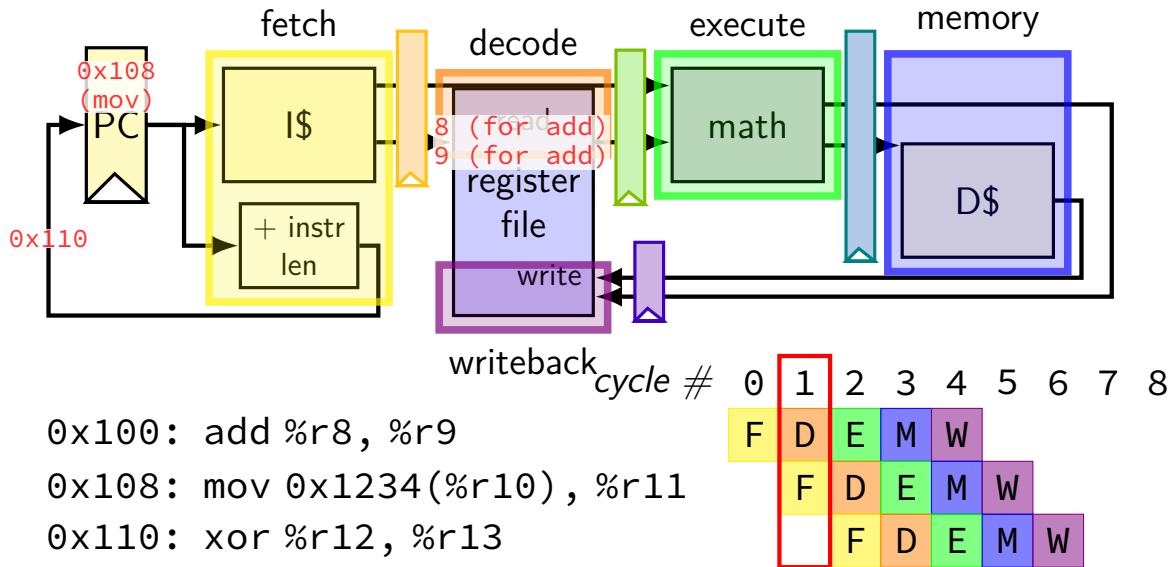
running some instructions



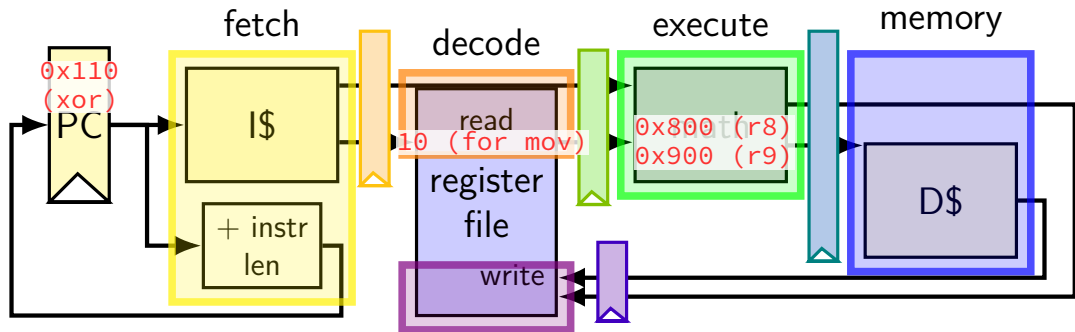
running some instructions



running some instructions



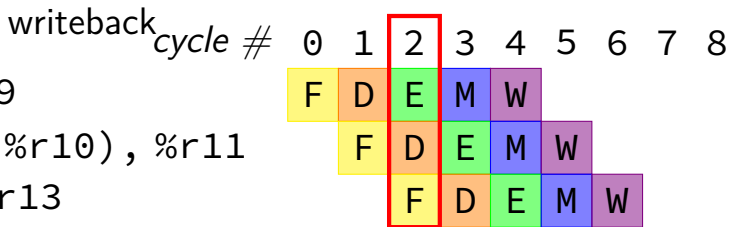
running some instructions



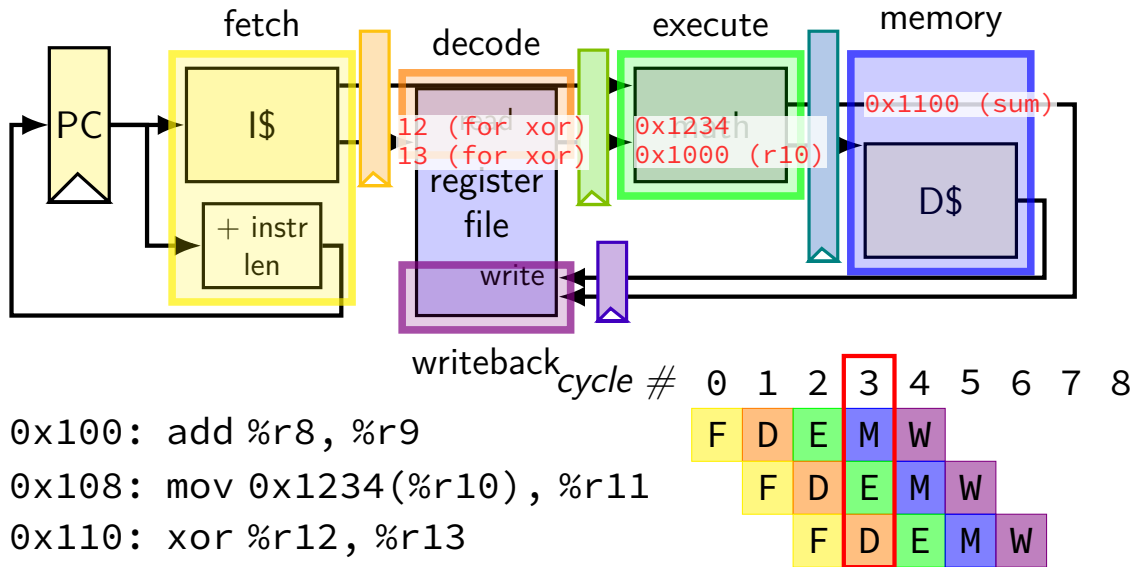
0x100: add %r8, %r9

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

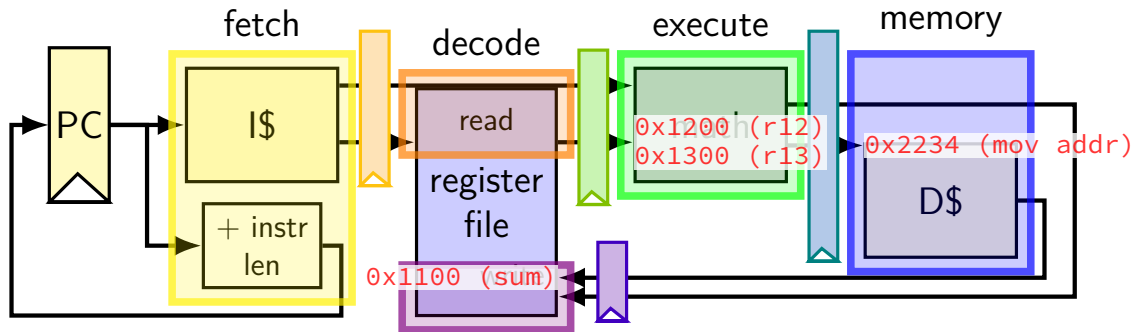
0x110: xor %r12, %r13



running some instructions



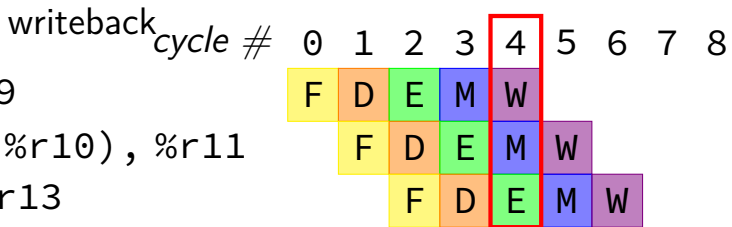
running some instructions



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: 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