

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

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

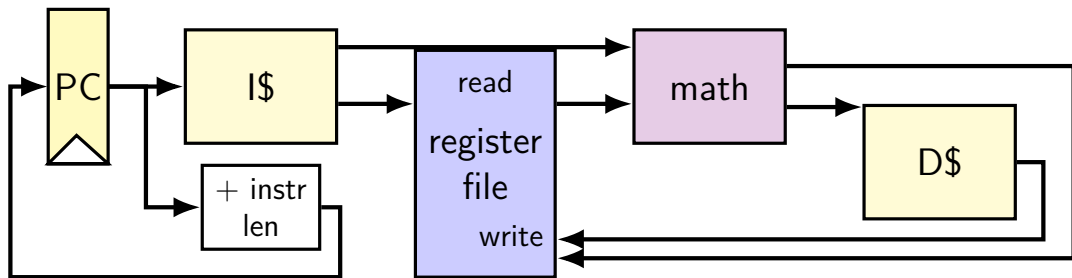
life HW: pattern of dividing up work in loop among multiple threads

alternate API idea: based on automating that:

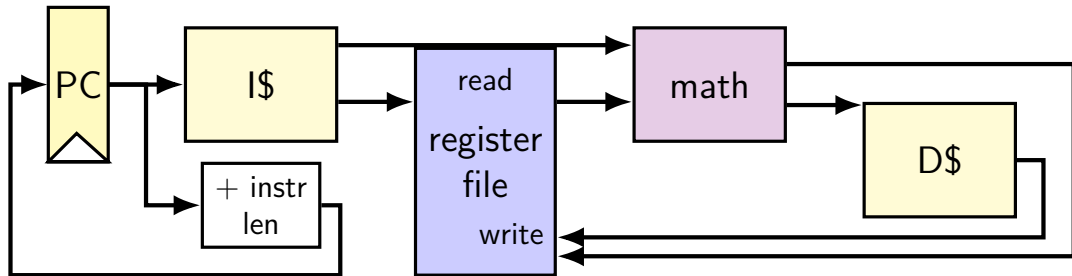
```
#pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        array[i] *= 2;
    }
```

subject of next week's lab

simple CPU



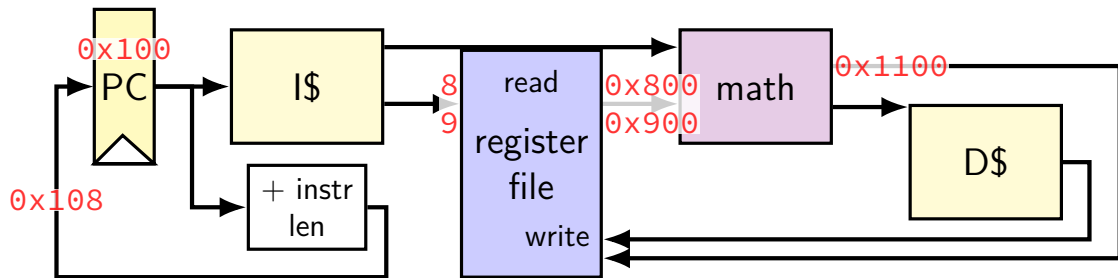
running instructions



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

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

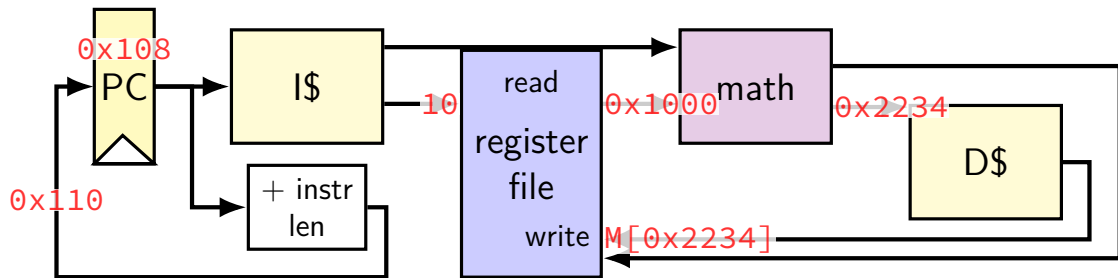
running instructions



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

```
...
%r8: 0x800
%r9: 0x1700
%r10: 0x1000
%r11: 0x1100
...
```

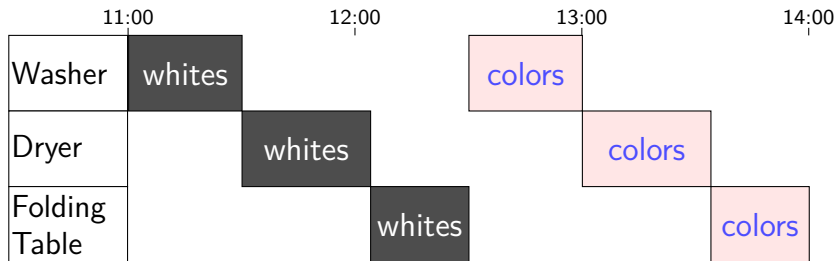
running instructions



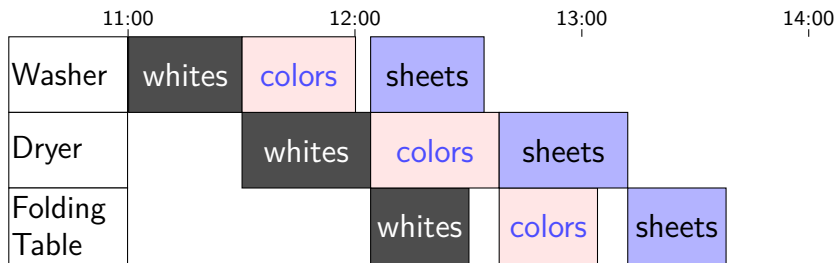
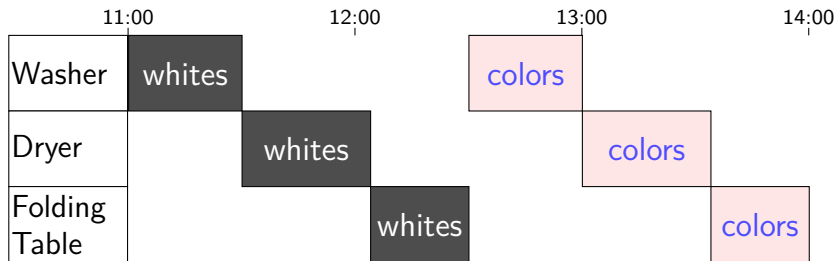
```
0x100: addq %r8, %r9
0x108: movq 0x1234(%r10), %r11
```

```
...
%r8: 0x800
%r9: 0x1700
%r10: 0x1000
%r11: M[0x2234]
...
```

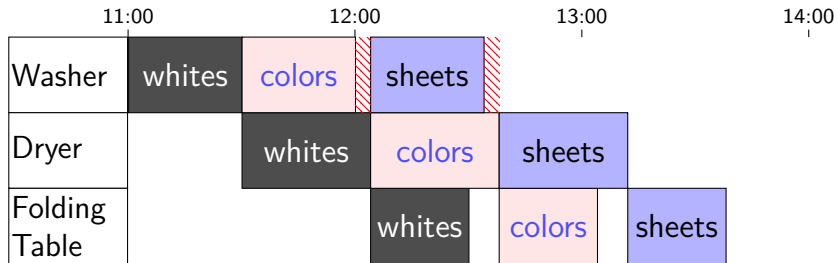
Human pipeline: laundry



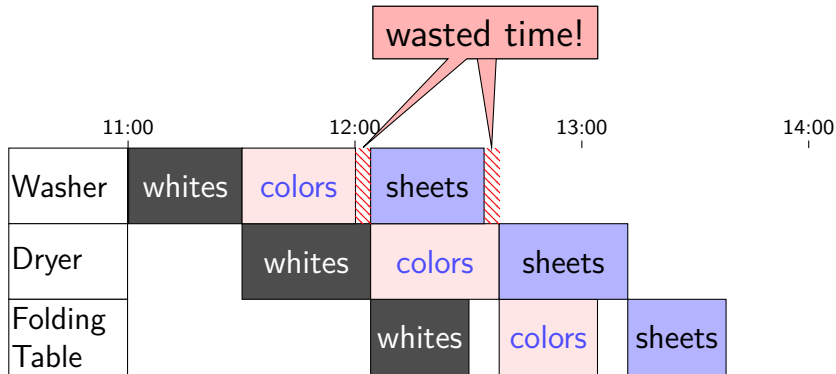
Human pipeline: laundry



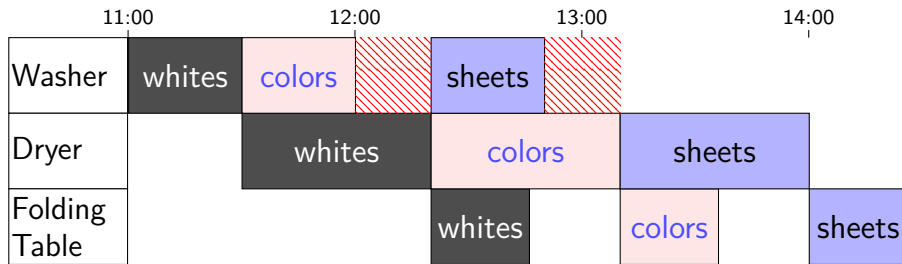
Waste (1)



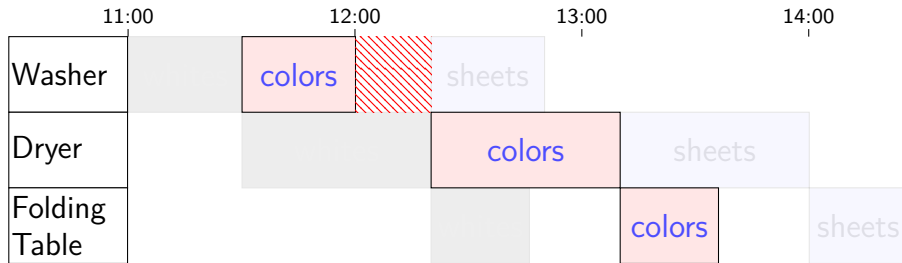
Waste (1)



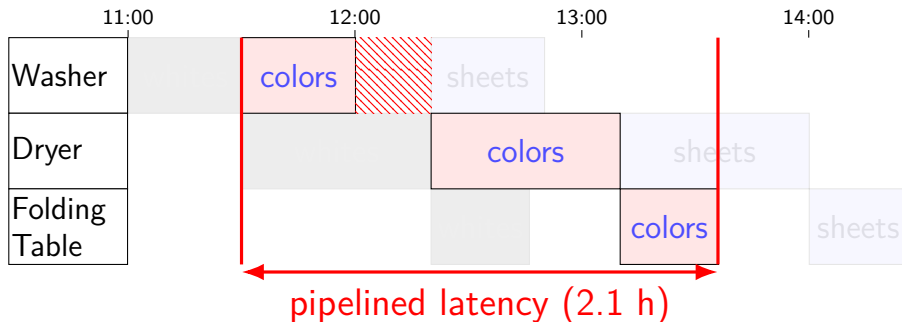
Waste (2)



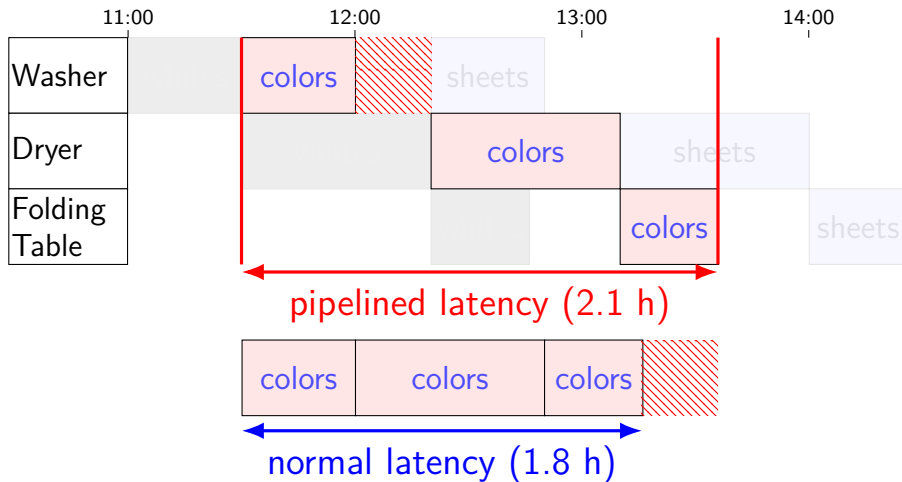
Latency — Time for One



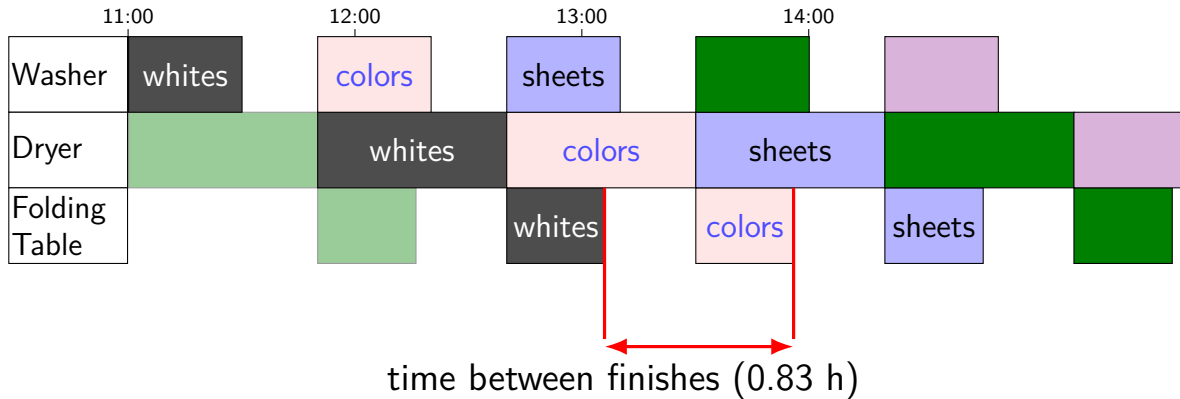
Latency — Time for One



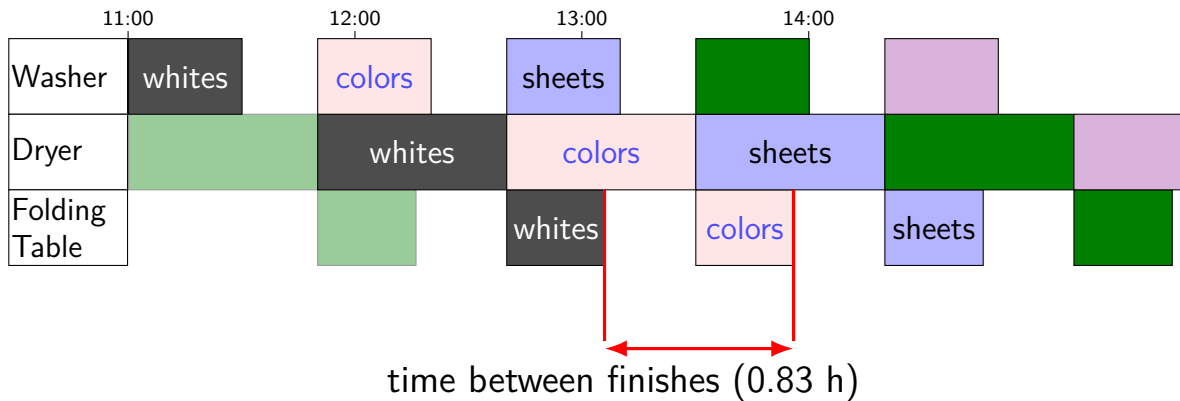
Latency — Time for One



Throughput — Rate of Many

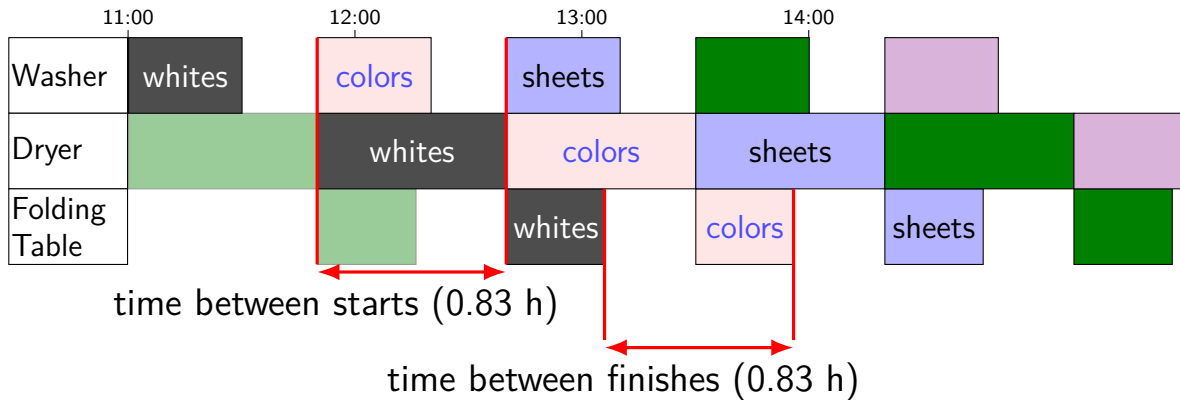


Throughput — Rate of Many



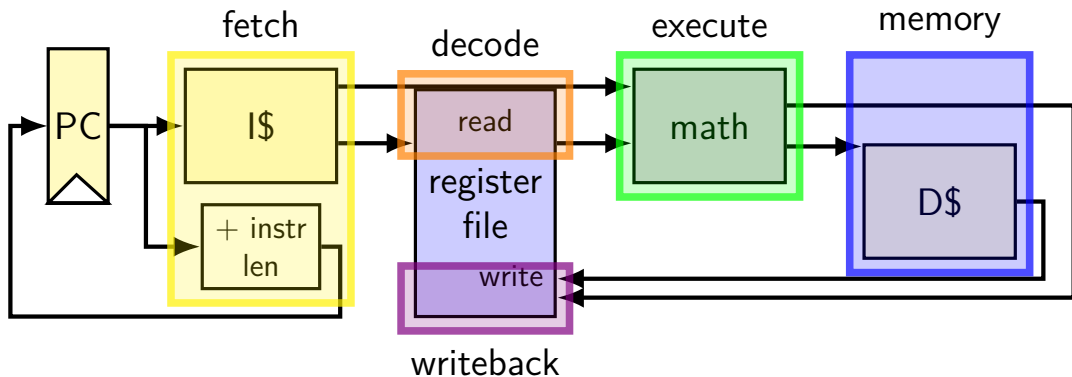
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many



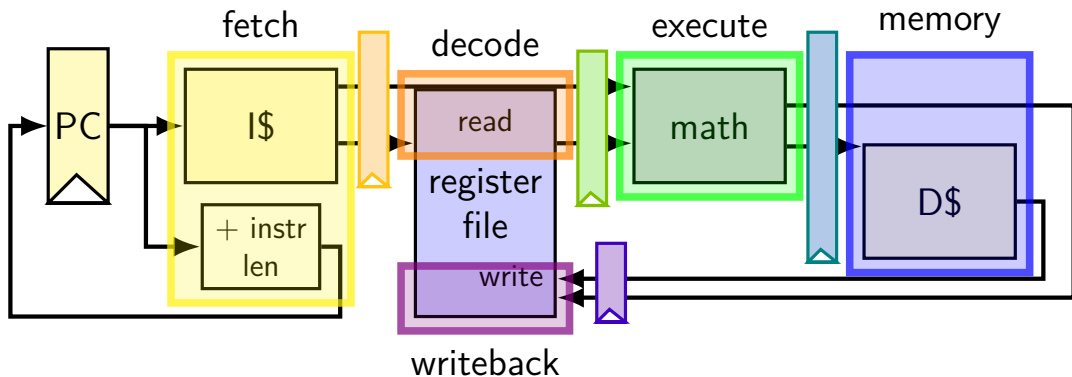
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

adding stages (one way)



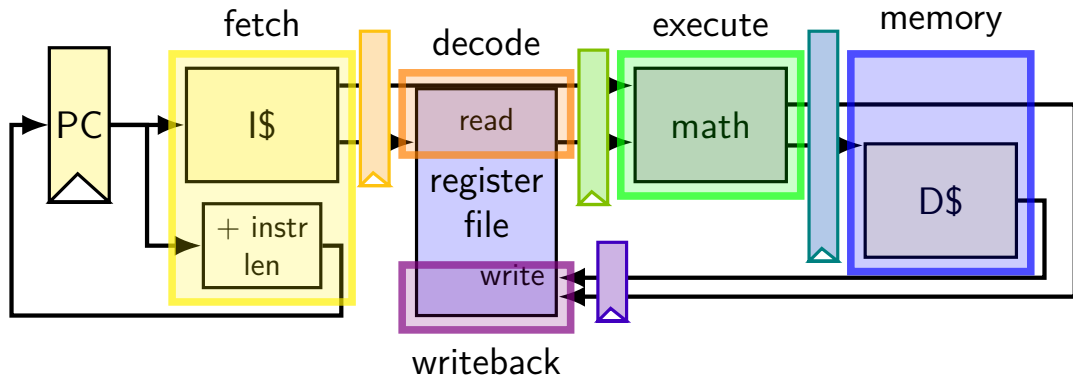
divide running instruction into steps
one way: fetch / decode / execute / memory / writeback

adding stages (one way)

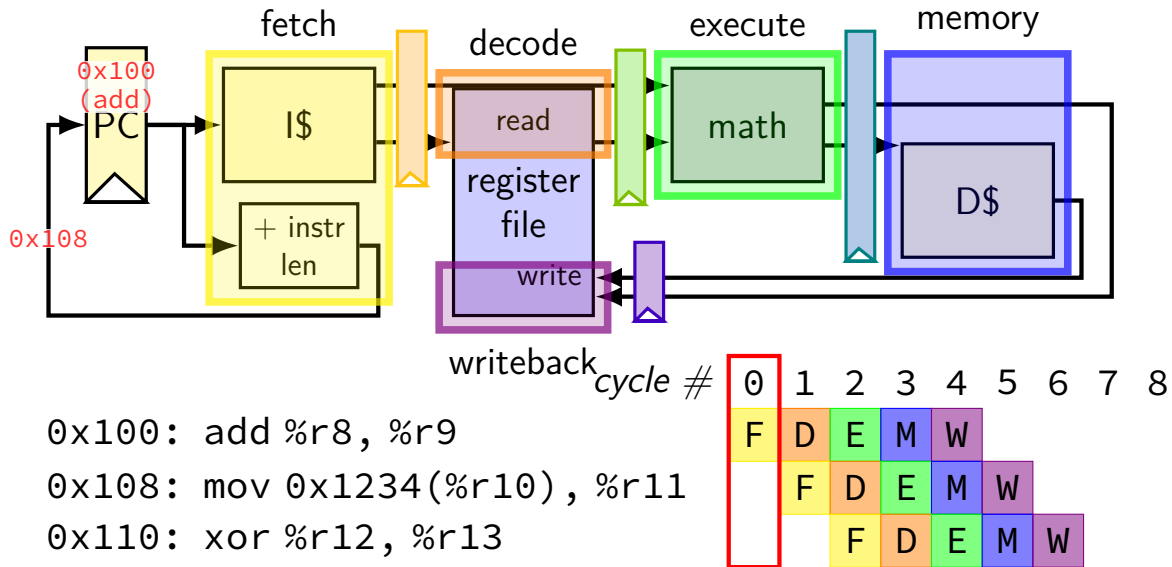


add 'pipeline registers' to hold values from instruction

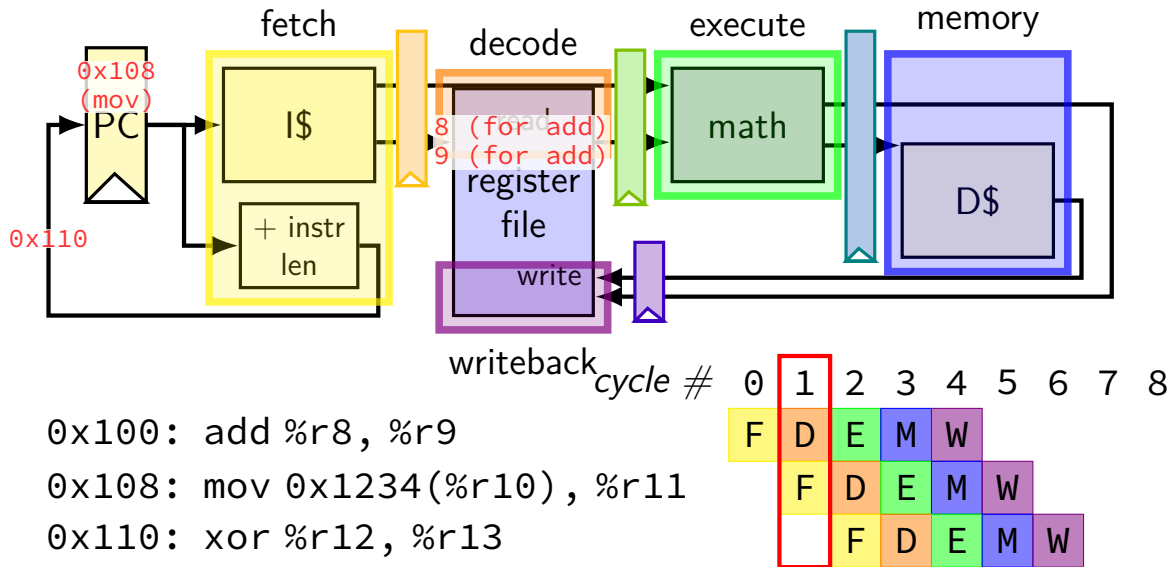
running some instructions



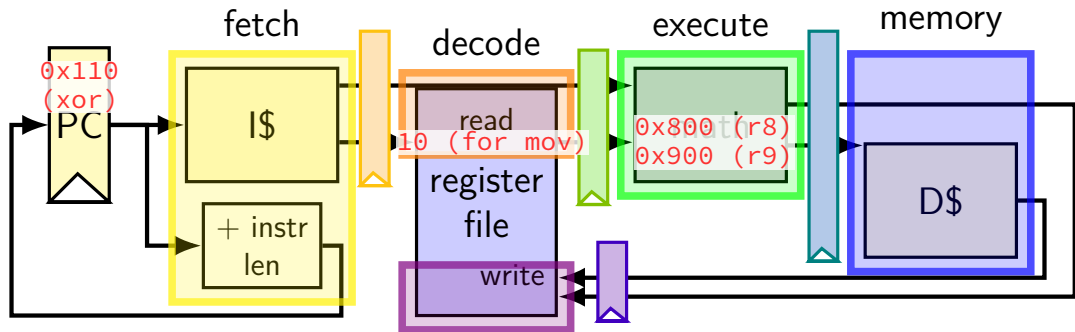
running some instructions



running some instructions



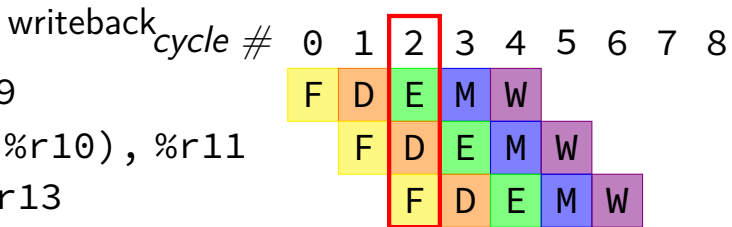
running some instructions



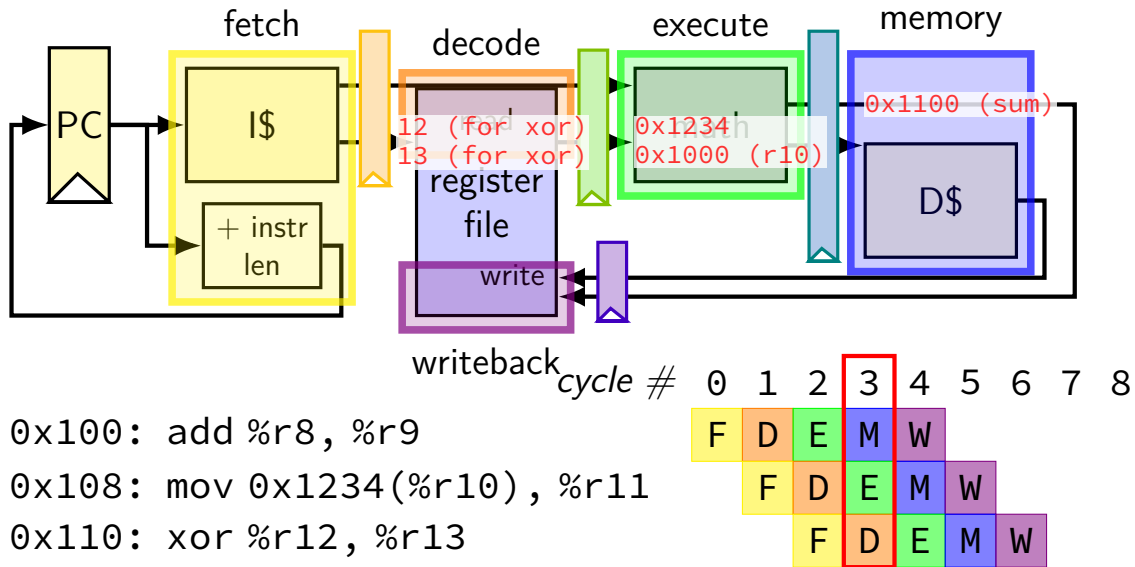
`0x100: add %r8, %r9`

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

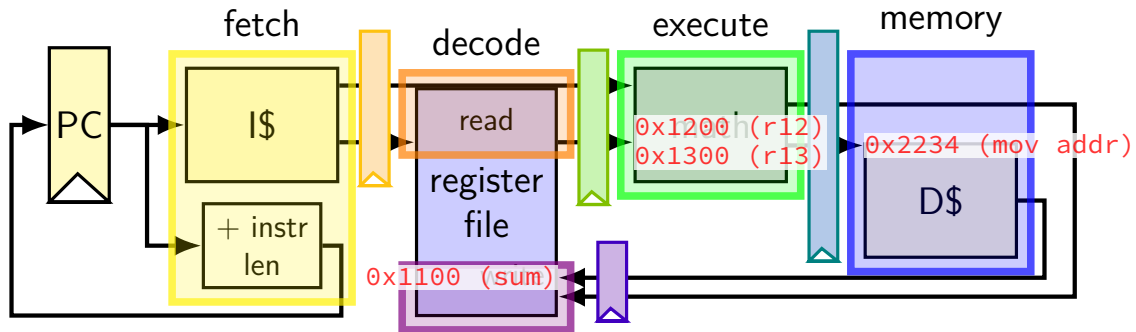
`0x110: xor %r12, %r13`



running some instructions



running some instructions



| | 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 | | |

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: ... | | | | | ... | | | | | |

support 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: ... | | | | | ... | | | | | |

support 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)

| | cycle # | 0 | 1 | 2 | 3 | 4 |
|-------------------------------|---------|---|---|---|---|---|
| 0x100: add %r8, %r9 | | F | D | E | M | W |
| 0x108: mov 0x1234(%r10), %r11 | | | F | D | E | M |
| 0x110: ... | | | | | | |

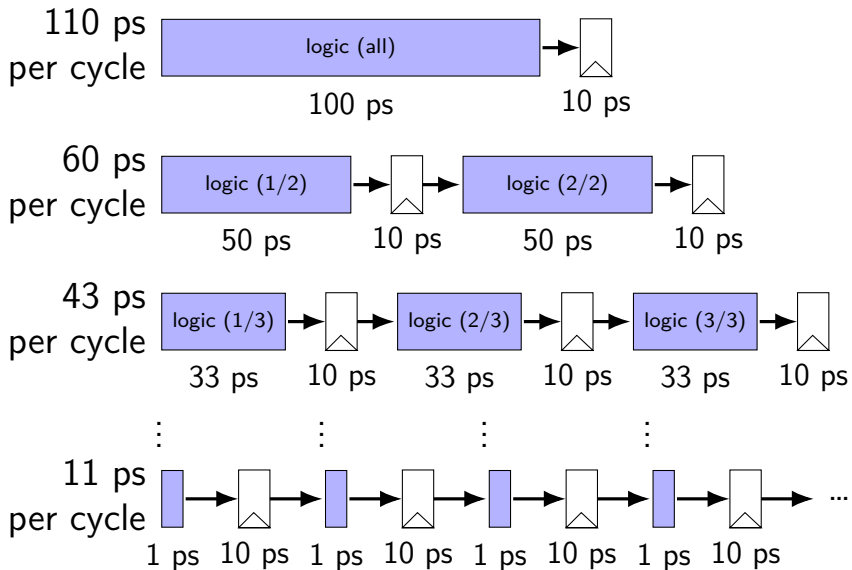
0x100: add %r8, %r9
0x108: mov 0x1234(%r10), %r11
0x110: ...

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

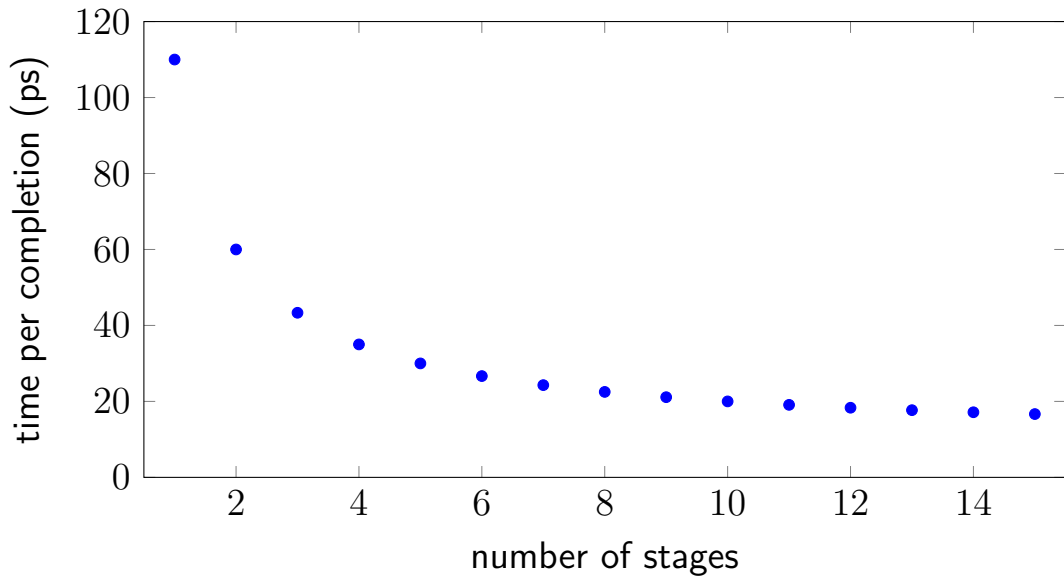
exercise: new throughput?

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

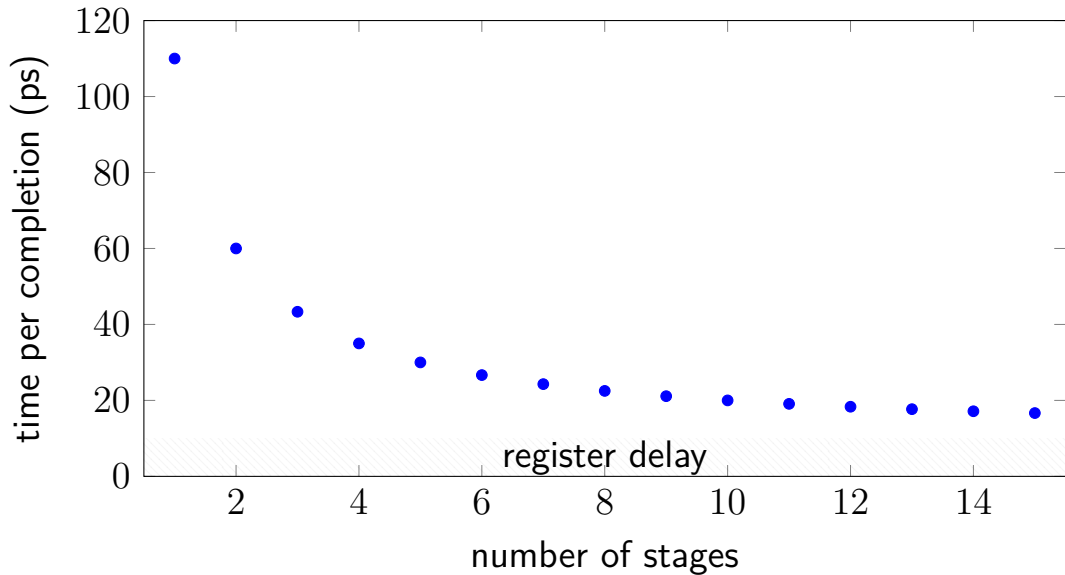
diminishing returns: register delays



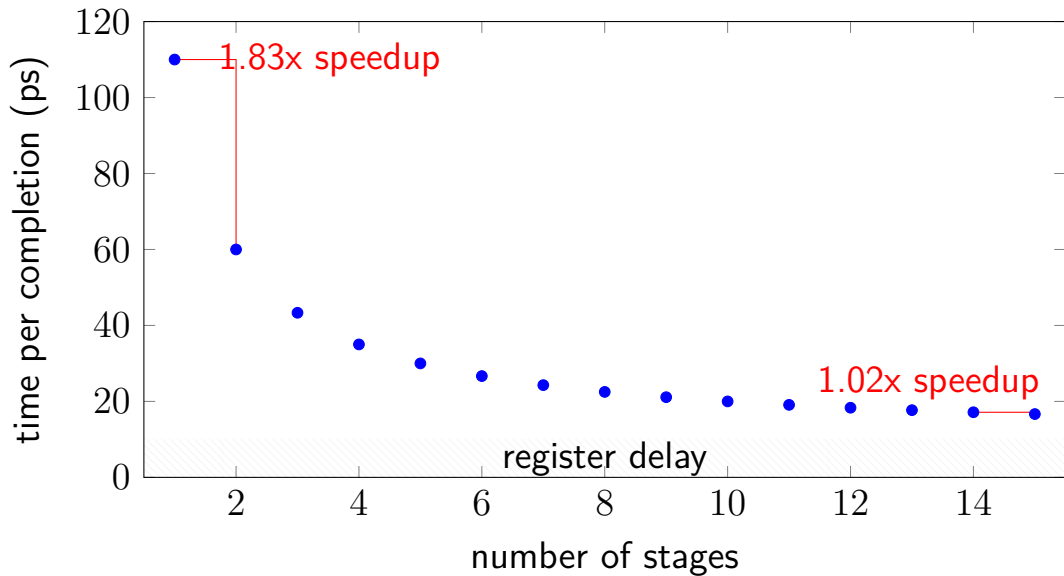
diminishing returns: register delays



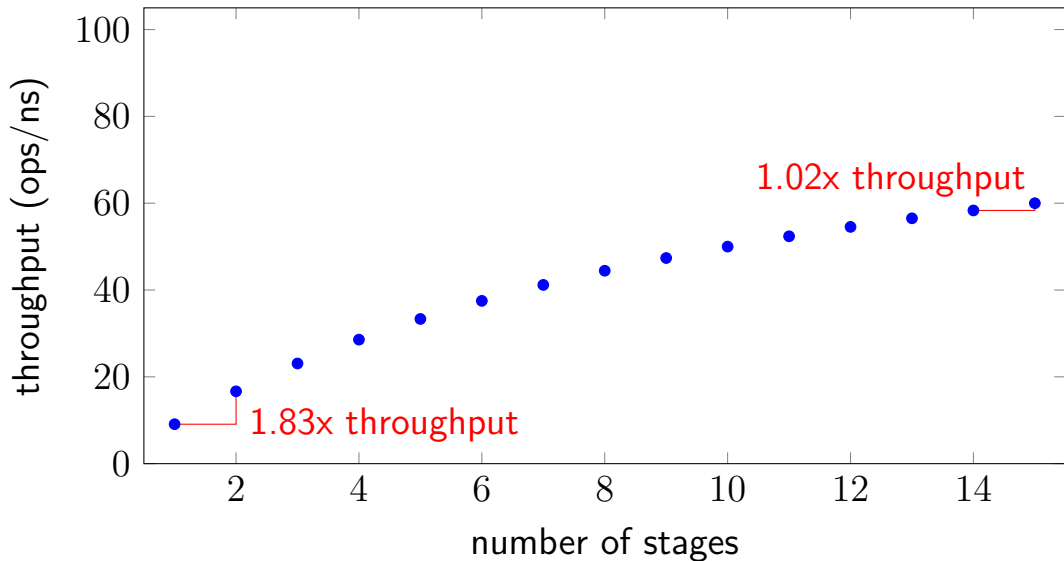
diminishing returns: register delays



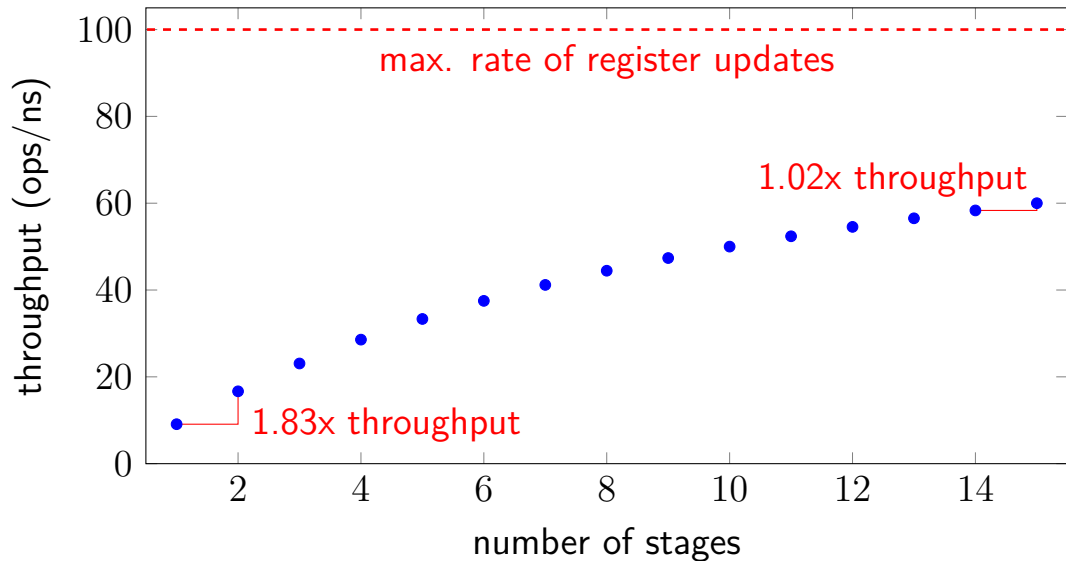
diminishing returns: register delays



diminishing returns: register delays



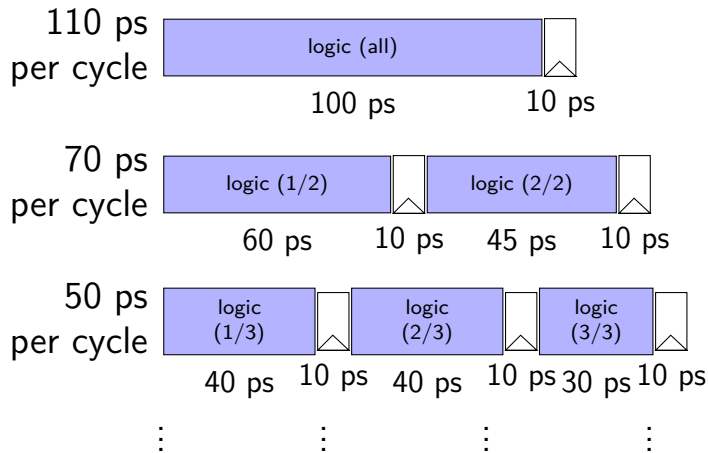
diminishing returns: register delays



diminishing returns: uneven split

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

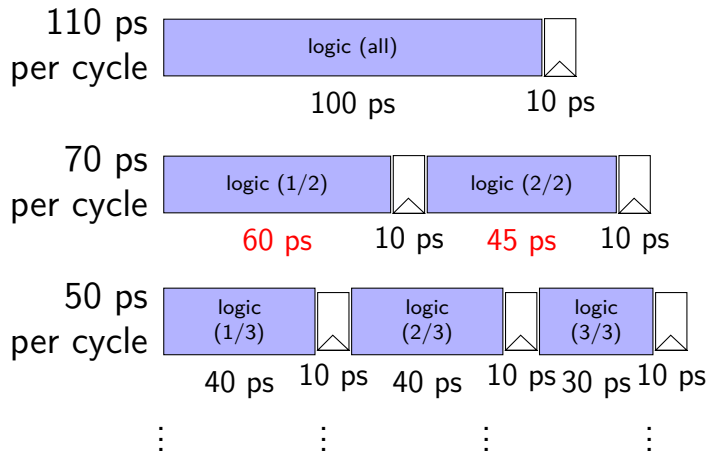
Probably not...



diminishing returns: uneven split

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

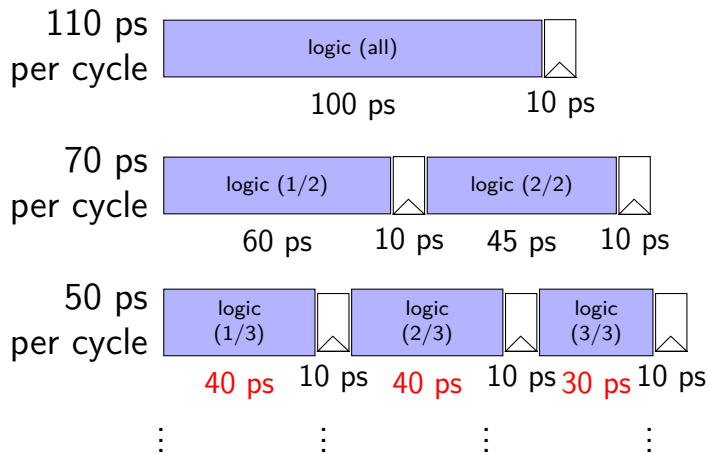
Probably not...



diminishing returns: uneven split

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

Probably not...



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**

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

opportunity

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

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

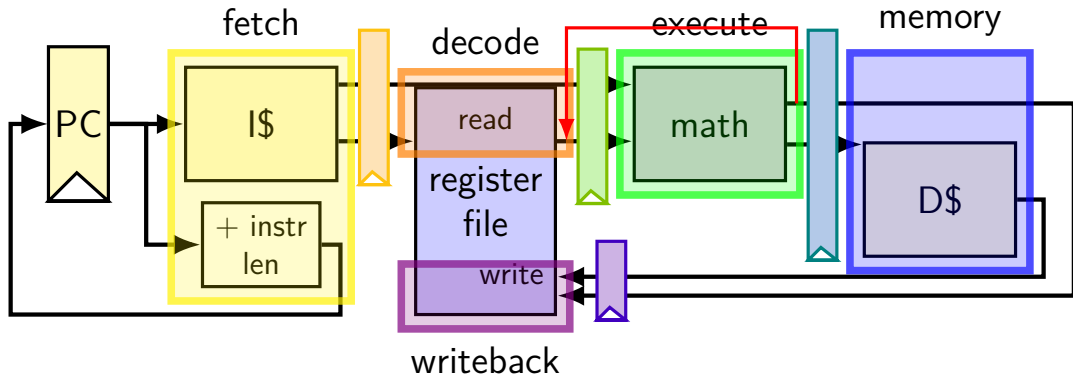
```
0x2: addq %r9, %r8
```

...

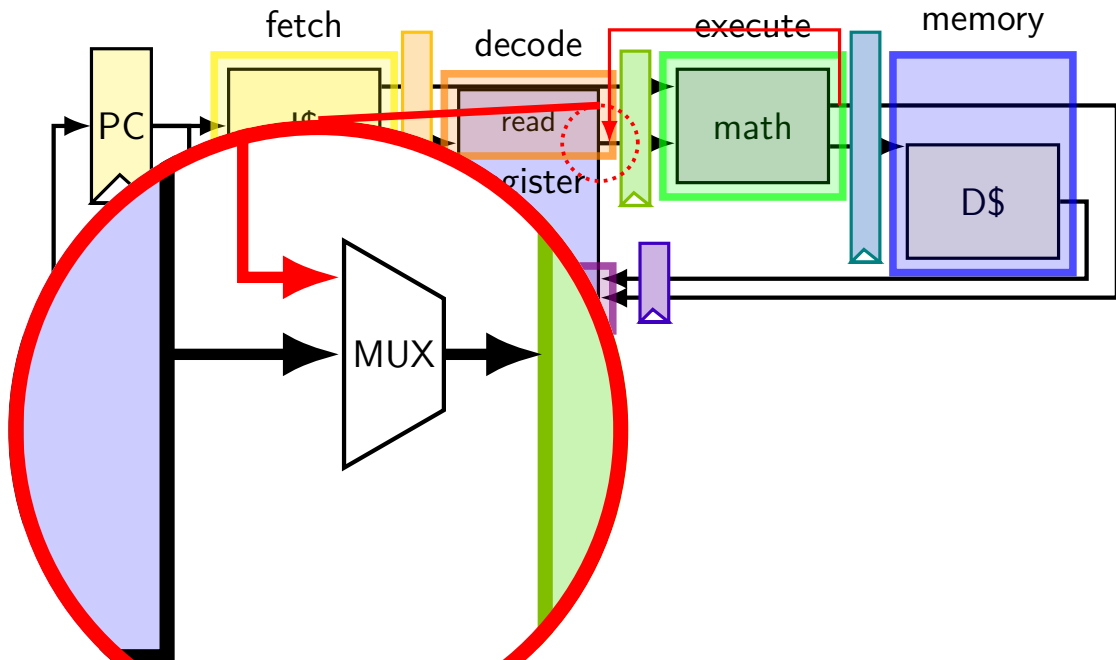
| | fetch | fetch/decode | | decode/execute | | | execute/memory | | memory/writeback | |
|-------|-------|--------------|----|----------------|-------|----|----------------|----|------------------|----|
| cycle | PC | rA | rB | R[rB] | R[rB] | rB | sum | rB | sum | rB |
| 0 | 0x0 | | | | | | | | | |
| 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
```

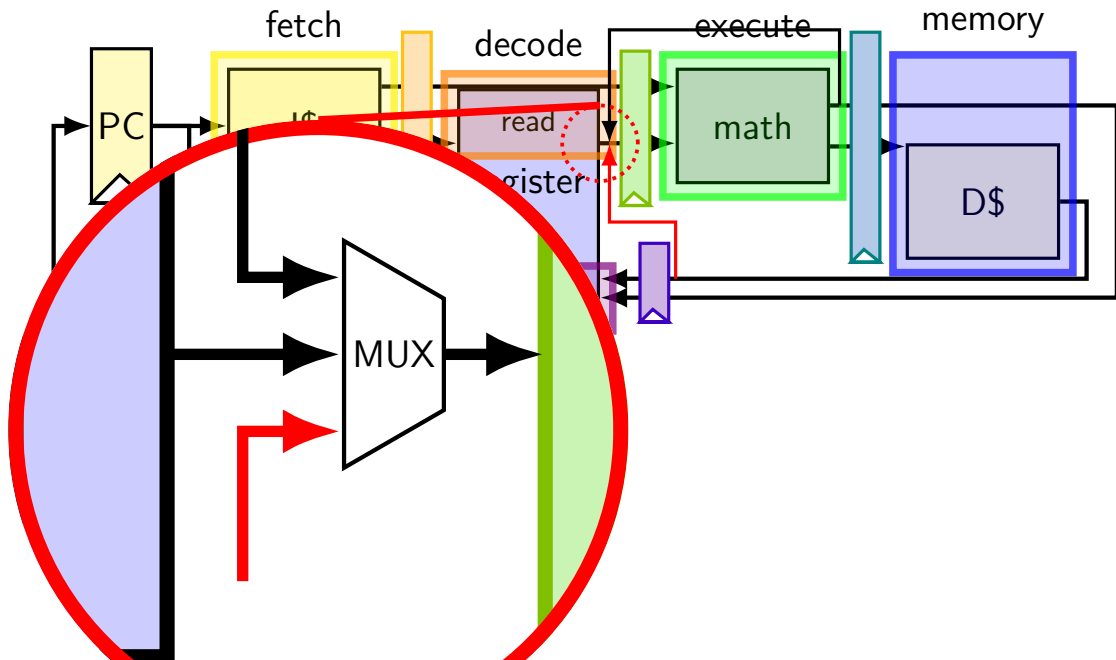
```
0x3: addq %r9, %r8
```

```
...
```

| | 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

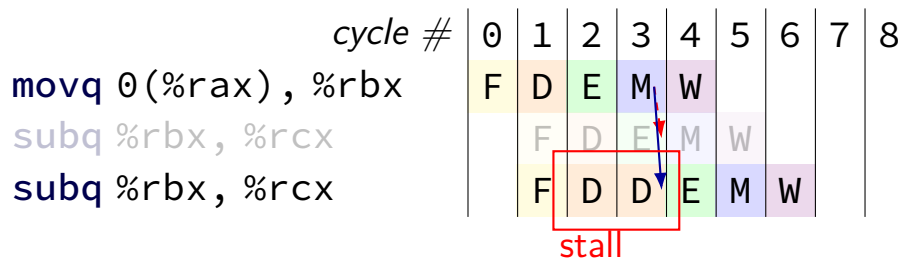
unsolved problem

| | 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



combine stalling and forwarding to resolve hazard

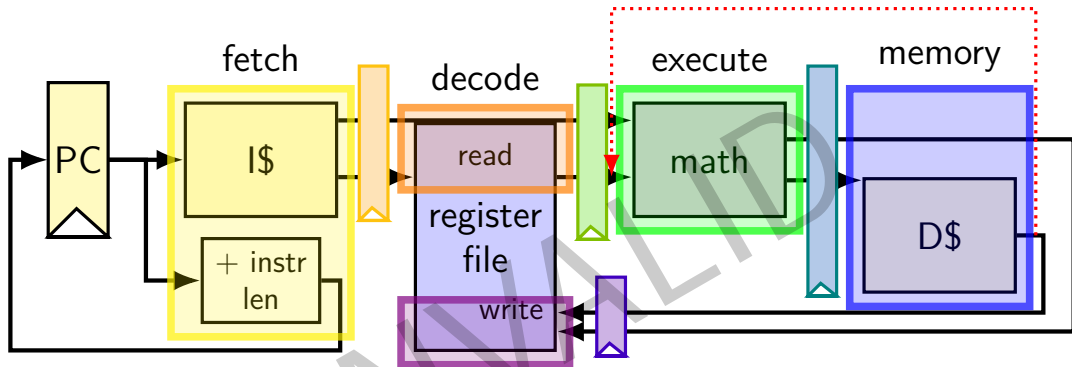
assumption in diagram: hazard detected in **subq**'s decode stage
(since easier than detecting it in fetch stage)

solveable problem

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

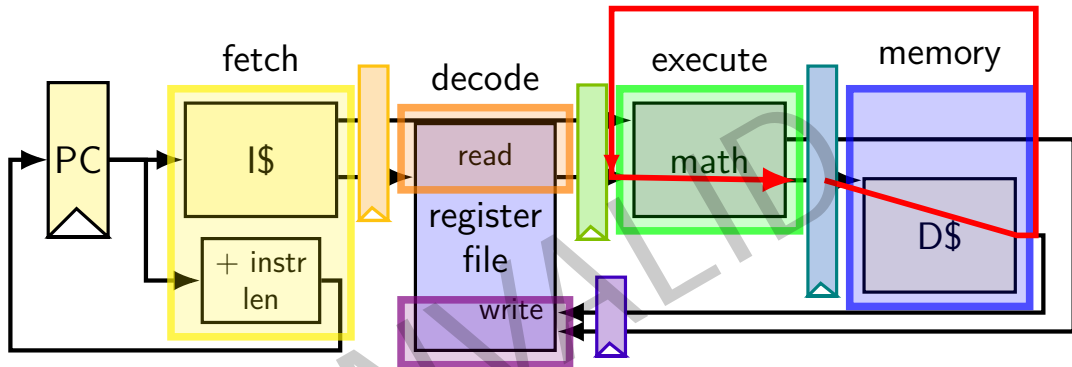
common for real processors to do this
but our textbook only forwards to the end of decode

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**

irmovq **\$100,** **%rcx**

addq **%rcx,** **%r10**

addq **%rbx,** **%r10**

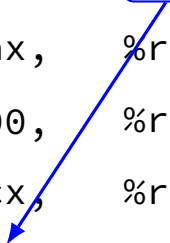
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

| | | |
|--------|--------|------|
| addq | %rax, | %rbx |
| subq | %rax, | %rcx |
| irmovq | \$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)

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

addq %rax, %rbx

subq %rax, %rcx

irmovq \$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
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```

The diagram illustrates data dependencies and hazards in a sequence of five assembly instructions. The instructions are: `addq %rax, %rbx`, `subq %rax, %rcx`, `irmovq $100, %rcx`, `addq %rcx, %r10`, and `addq %rbx, %r10`. Blue boxes highlight the source register `%rbx` in the first instruction and the destination register `%rcx` in the fourth instruction. Red boxes highlight the source register `%rcx` in the fourth instruction, the destination register `%r10` in the fourth instruction, and the source register `%rcx` in the fifth instruction. A blue arrow points from the `%rbx` box in the first instruction to the `%rcx` box in the fourth instruction. A red arrow points from the `%rcx` box in the third instruction to the `%rcx` box in the fourth instruction. Another red arrow points from the `%r10` box in the fourth instruction to the `%r10` box in the fifth instruction.

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> |
|------------------------------|-------------------|-------------------|
| <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) 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

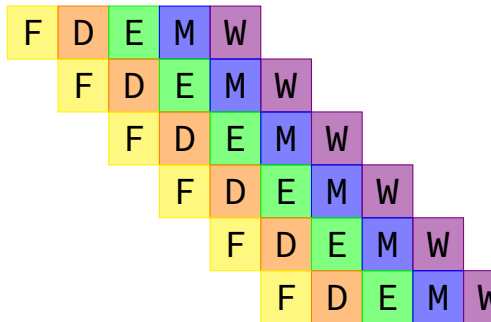
jXX: stalling?

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

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

```
subq %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



making guesses

```
    subq %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

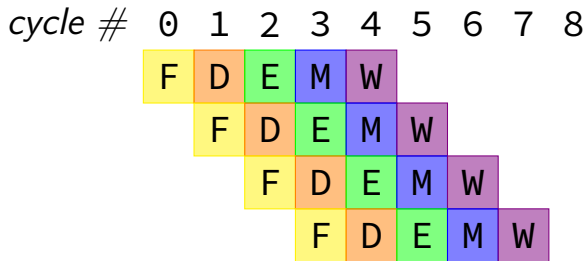
right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

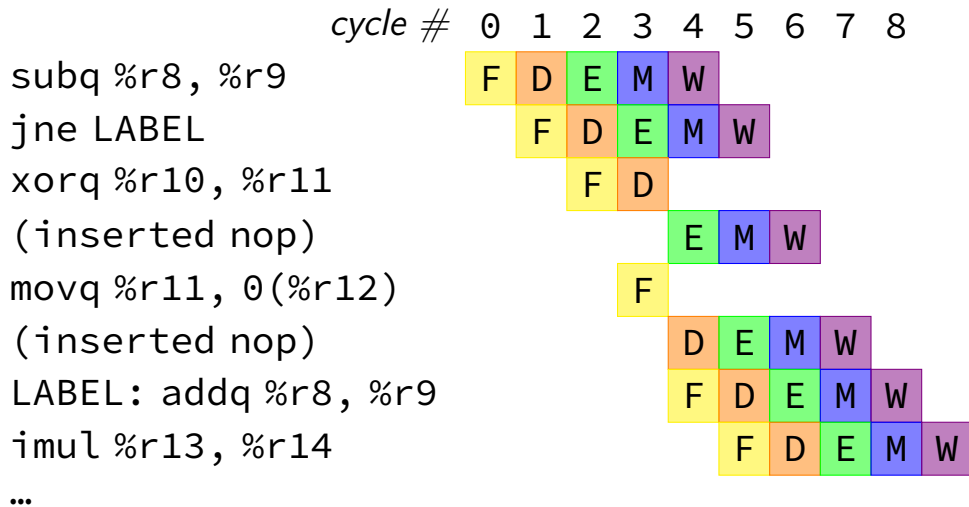
```
subq %r8, %r9
jne LABEL
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

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

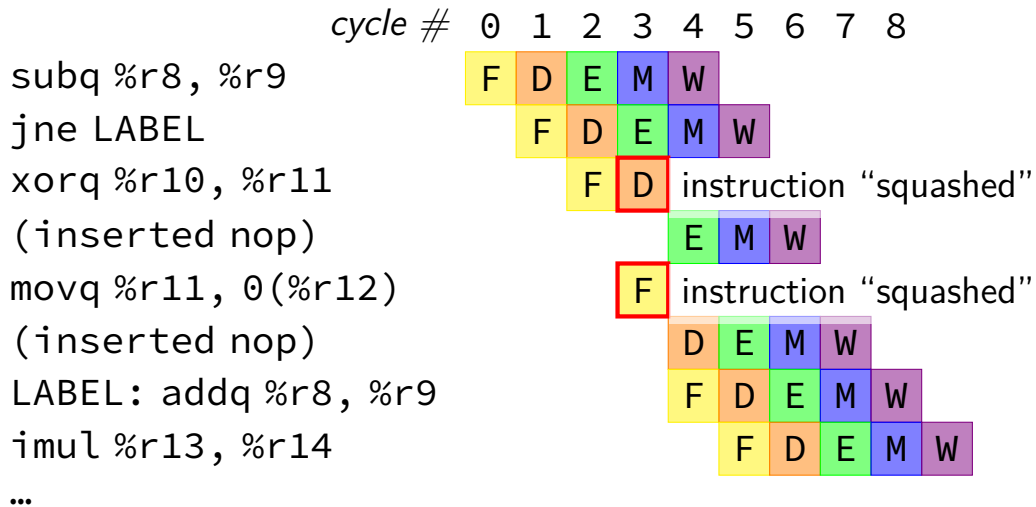
```
subq %r8, %r9
jne LABEL
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



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.

performance

hypothetical instruction mix

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

performance

hypothetical instruction mix

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

static branch prediction

forward (target > PC) not taken; backward taken

intuition: loops:

```
LOOP: ...
```

```
...
```

```
je LOOP
```

```
LOOP: ...
```

```
jne SKIP_LOOP
```

```
...
```

```
jmp LOOP
```

```
SKIP_LOOP:
```

exercise: static prediction

```
.global foo
foo:
```

```
    xor %eax, %eax // eax <- 0
```

```
foo_loop_top:
```

```
    test $0x1, %edi
```

```
    je foo_loop_bottom // if (edi & 1 == 0) goto .
```

```
    add %edi, %eax
```

```
foo_loop_bottom:
```

```
    dec %edi // edi = edi - 1
```

```
    jl .Lend_loop // if (edi < 0) goto .Lend_
```

```
    ret
```

suppose %edi = 3 (initially)

and using forward-taken, backwards-not-taken strategy:

branch target buffer

what can we do to predict `jmp *(%rax)`?

what if we can't decode LABEL from machine code for `jmp LABEL` fast?

will happen in more complex pipelines

BTB: cache for branches

| idx | valid | tag | ofst | type | target | (more info?) |
|------|-------|-------|------|------|----------|--------------|
| 0x00 | 1 | 0x400 | 5 | Jxx | 0x3FFFF3 | ... |
| 0x01 | 1 | 0x401 | C | JMP | 0x401035 | ---- |
| 0x02 | 0 | --- | --- | --- | --- | ---- |
| 0x03 | 1 | 0x400 | 9 | RET | --- | ... |
| ... | ... | ... | ... | ... | ... | ... |
| 0xFF | 1 | 0x3FF | 8 | CALL | 0x404033 | ... |

| valid | ... |
|-------|-----|
| 1 | ... |
| 0 | ... |
| 0 | ... |
| 0 | ... |
| ... | ... |
| 0 | ... |

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
...
0x400031:  ret
...
```

BTB: cache for branches

| idx | valid | tag | ofst | type | target | (more info?) |
|------|-------|-------|------|------|----------|--------------|
| 0x00 | 1 | 0x400 | 5 | Jxx | 0x3FFFF3 | ... |
| 0x01 | 1 | 0x401 | C | JMP | 0x401035 | --- |
| 0x02 | 0 | --- | --- | --- | --- | --- |
| 0x03 | 1 | 0x400 | 9 | RET | --- | ... |
| ... | ... | ... | ... | ... | ... | ... |
| 0xFF | 1 | 0x3FF | 8 | CALL | 0x404033 | ... |

| valid | ... |
|-------|-----|
| 1 | ... |
| 0 | ... |
| 0 | ... |
| 0 | ... |
| ... | ... |
| 0 | ... |

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
...
0x400031:  ret
...
```


BTB: cache for branches

| idx | valid | tag | ofst | type | target | (more info?) |
|------|-------|-------|------|------|----------|--------------|
| 0x00 | 1 | 0x400 | 5 | Jxx | 0x3FFFF3 | ... |
| 0x01 | 1 | 0x401 | C | JMP | 0x401035 | --- |
| 0x02 | 0 | --- | --- | --- | --- | --- |
| 0x03 | 1 | 0x400 | 9 | RET | --- | ... |
| ... | ... | ... | ... | ... | ... | ... |
| 0xFF | 1 | 0x3FF | 8 | CALL | 0x404033 | ... |

| valid | ... |
|-------|-----|
| 1 | ... |
| 0 | ... |
| 0 | ... |
| 0 | ... |
| ... | ... |
| 0 | ... |

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
...
0x400031:  ret
...
```

predicting ret: extra copy of stack

predicting ret — ministack in processor registers

push on ministack on call; pop on ret

ministack overflows? discard oldest, mispredict it later

| |
|---------------------|
| baz saved registers |
| baz return address |
| bar saved registers |
| bar return address |
| foo local variables |
| foo saved registers |
| foo return address |
| foo saved registers |

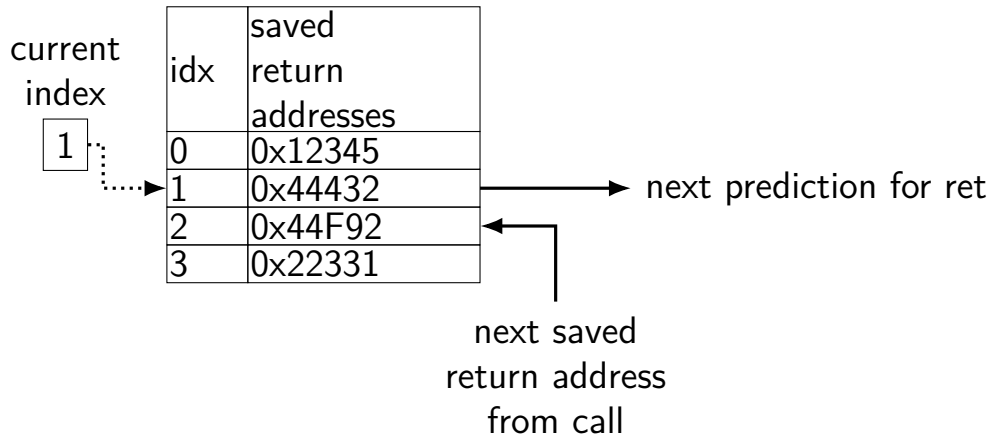
| |
|--------------------|
| baz return address |
| bar return address |
| foo return address |

(partial?) stack
in CPU registers

stack in memory

4-entry return address stack

4-entry return address stack in CPU



on call: increment index, save return address in that slot

on ret: read prediction from index, decrement index

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