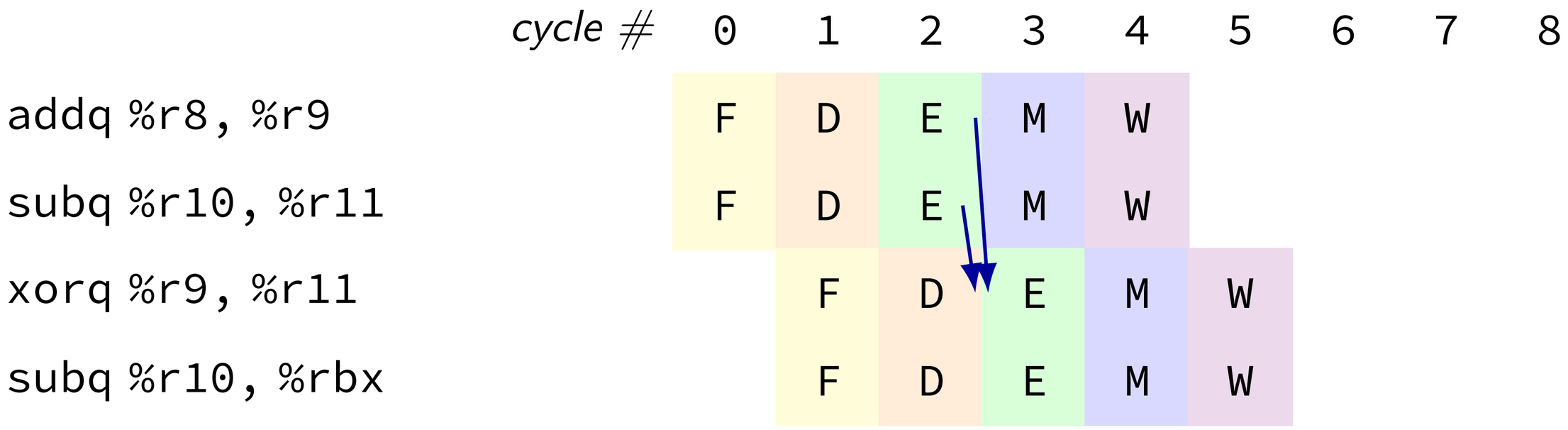
OOO

beyond pipelining: multiple issue

start *more than one instruction/cycle*

multiple parallel pipelines; many-input/output register file

hazard handling much more complex



beyond pipelining: out-of-order

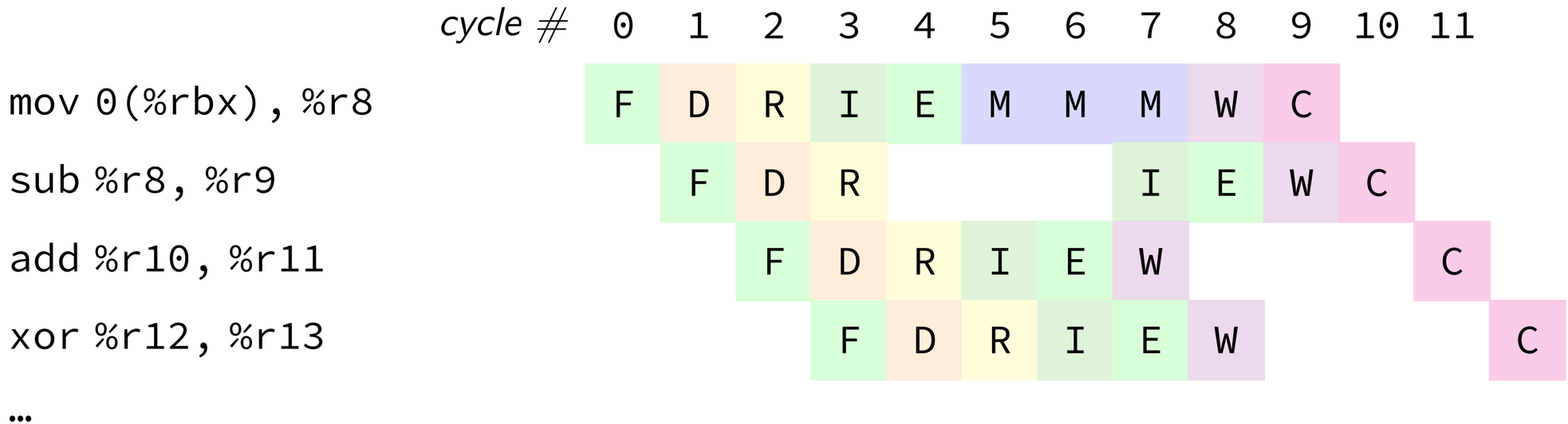
find *later instructions to do* instead of stalling

lists of available instructions in pipeline registers

- take any instruction with available values

provide *illusion that work is still done in order*

- much more complicated hazard handling logic



interlude: real CPUs

modern CPUs:

execute *multiple instructions at once*

execute instructions *out of order* — whenever *values available*

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- value in last stage may not be most up-to-date
- older value may be written back before newer value?

problems for branch prediction:

- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

- how to quickly find instructions that are ready?

out-of-order and hazards

out-of-order execution makes hazards harder to handle

problems for forwarding:

- *value in last stage may not be most up-to-date*
- older value may be written back before newer value?

problems for branch prediction:

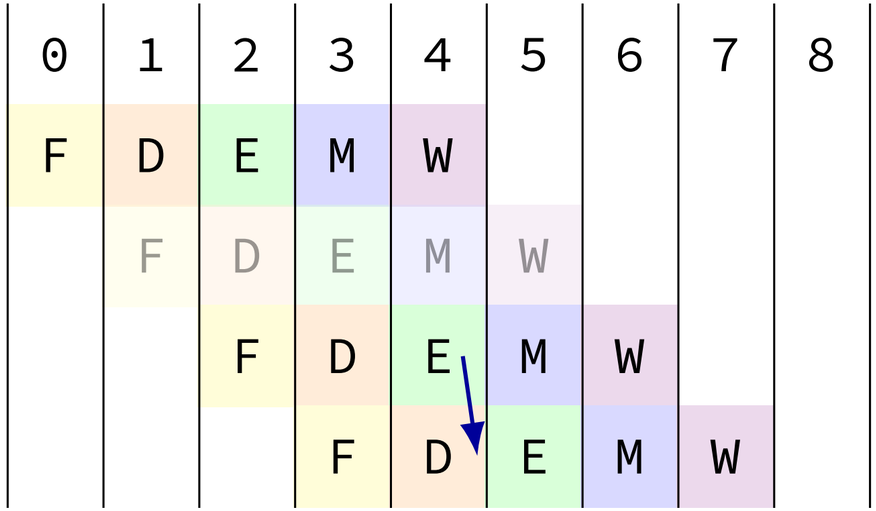
- mispredicted instructions may complete execution before squashing

which instructions to dispatch?

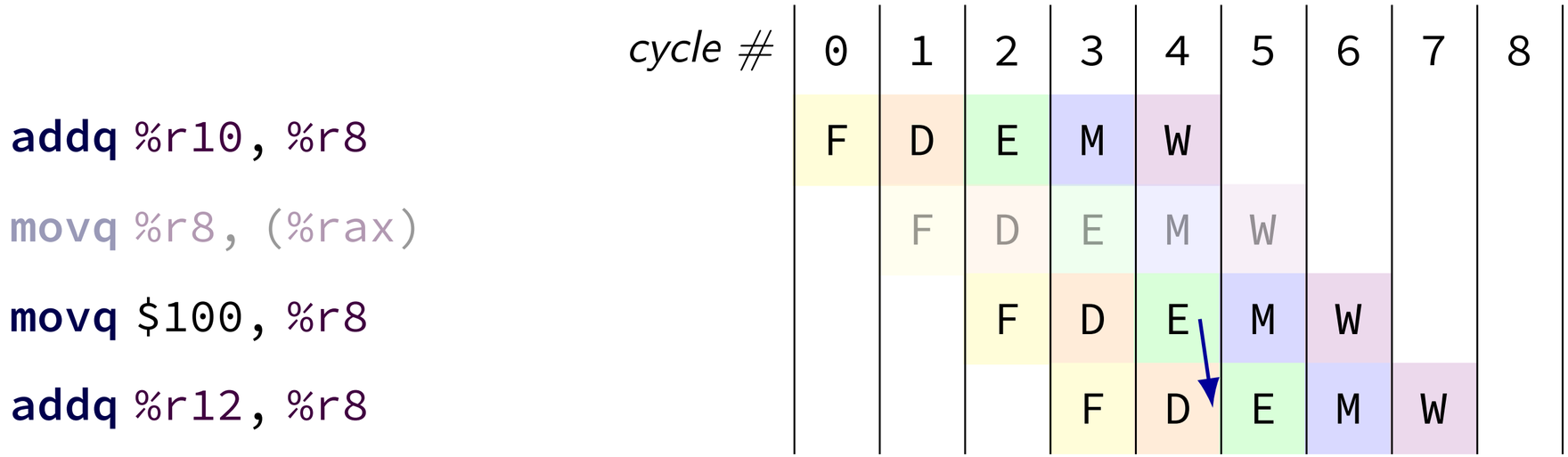
- how to quickly find instructions that are ready?

read-after-write examples (1)

```
addq %r10, %r8
movq %r8, (%rax)
movq $100, %r8
addq %r12, %r8
```

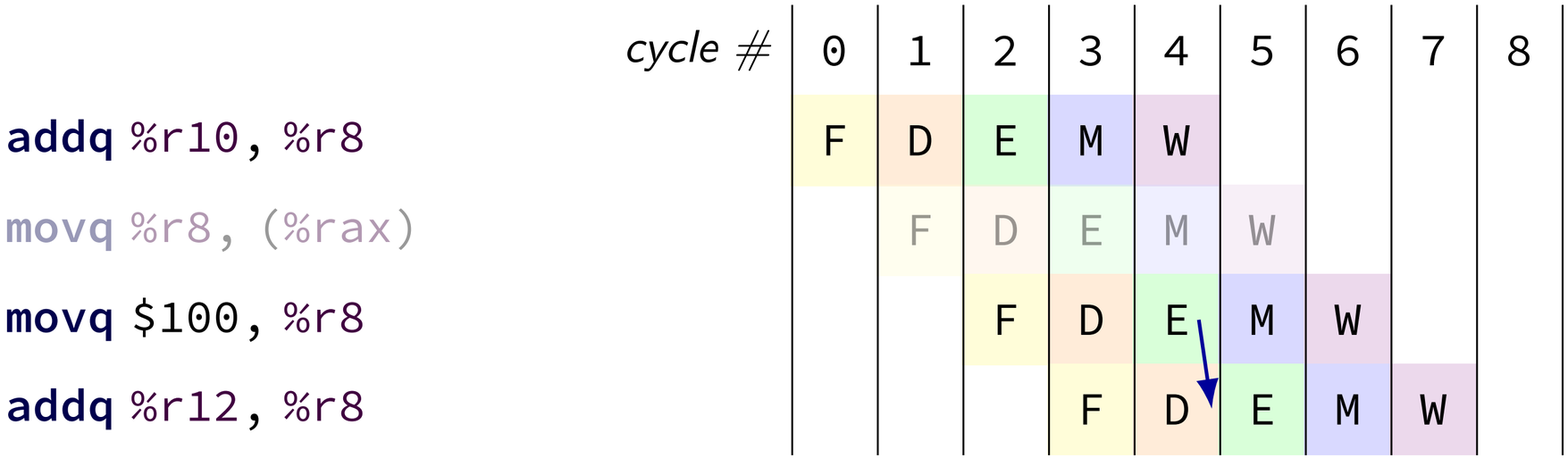


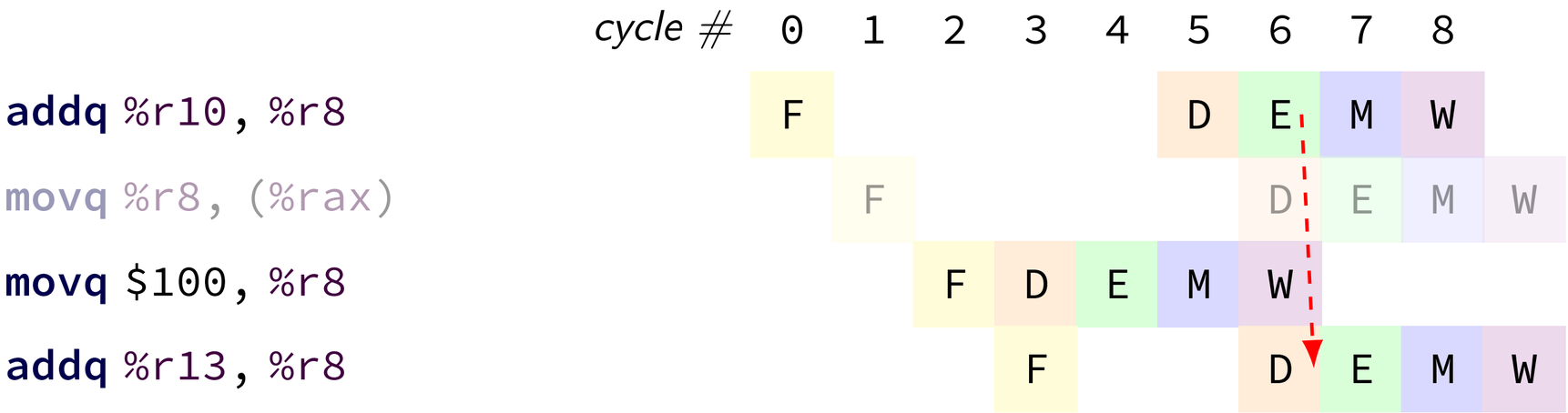
read-after-write examples (1)



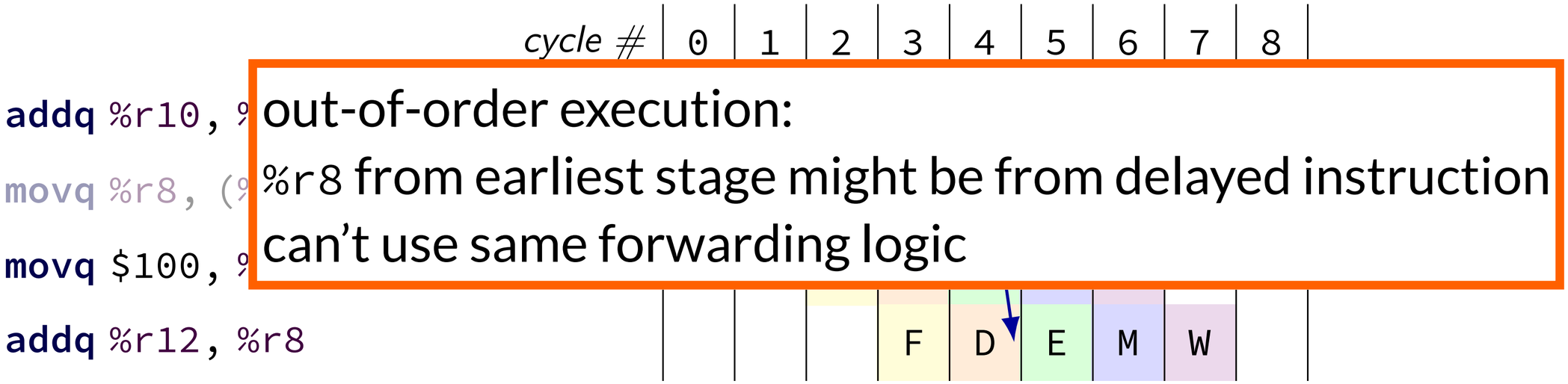
normal pipeline: two options for %r8?
choose the one from *earliest stage*
because it's from the most recent instruction

read-after-write examples (1)





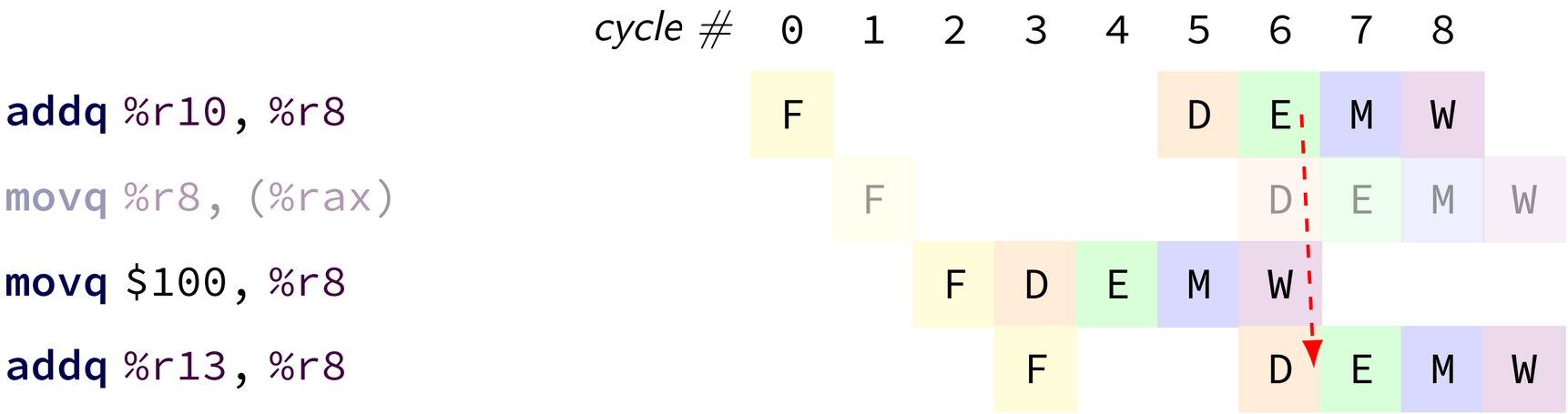
read-after-write examples (1)



out-of-order execution:

`%r8` from earliest stage might be from delayed instruction

can't use same forwarding logic



register version tracking

goal: track *different versions of registers*

out-of-order execution: may compute versions at different times

only forward the *correct version*

strategy for doing this: preprocess instructions represent version info

makes forwarding, etc. lookup easier

rewriting hazard examples (1)

addq %r10, %r8	addq $\%r10_{v1}, \%r8_{v1} \rightarrow \%r8_{v2}$
addq %r11, %r8	addq $\%r11_{v1}, \%r8_{v2} \rightarrow \%r8_{v3}$
addq %r12, %r8	addq $\%r12_{v1}, \%r8_{v3} \rightarrow \%r8_{v4}$

---

read different version than the one written

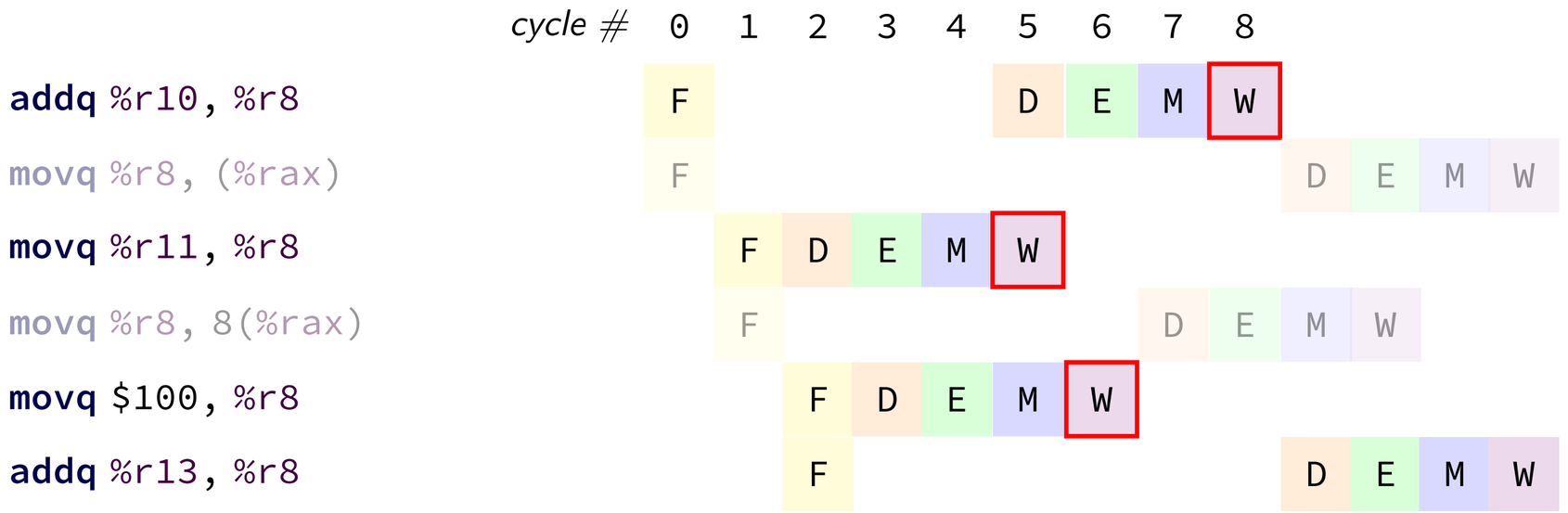
- represent with three argument psuedo-instructions

forwarding a value? must match version *exactly*

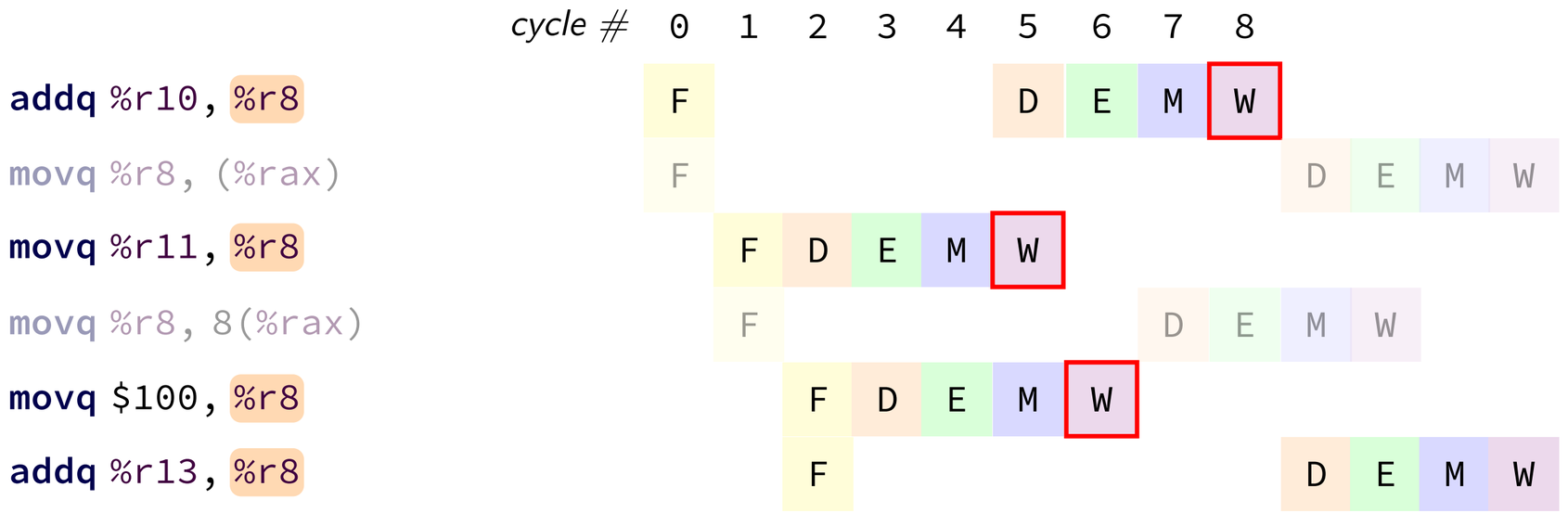
for now: version numbers

later: something simpler to implement

write-after-write example



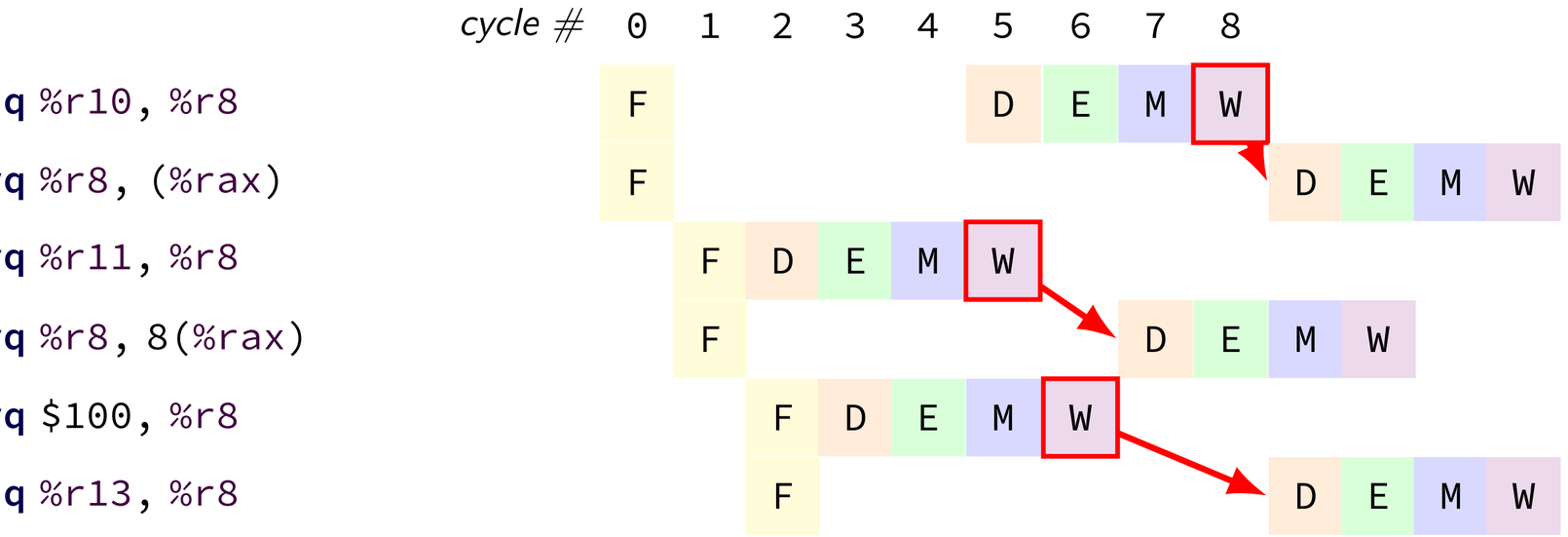
write-after-write example



many instructions producing differnt version of `%r8`
most recently run instruction may be writing *outdated version*

write-after-write example

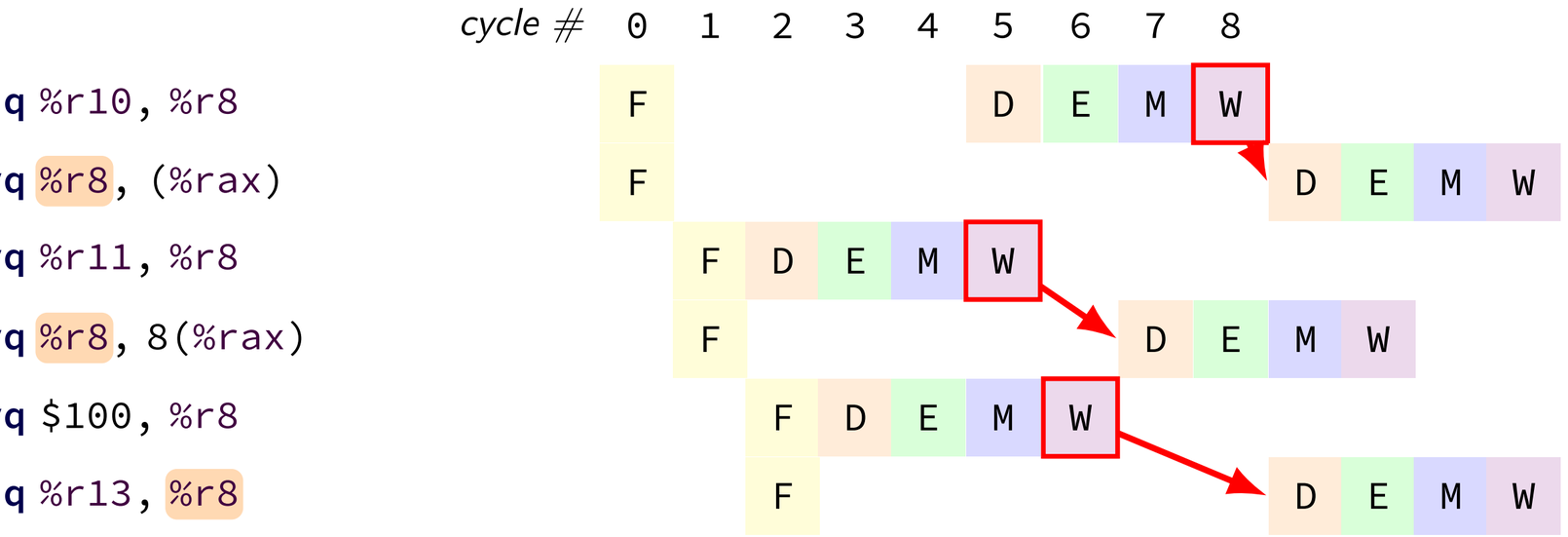
```
addq %r10, %r8
movq %r8, (%rax)
movq %r11, %r8
movq %r8, 8(%rax)
movq $100, %r8
addq %r13, %r8
```



multiple instructions that haven't started could need *different versions* of `%r8`

write-after-write example

```
addq %r10, %r8
movq %r8, (%rax)
movq %r11, %r8
movq %r8, 8(%rax)
movq $100, %r8
addq %r13, %r8
```



keeping multiple versions

for write-after-write problem: need to keep copies of multiple versions

both the new version and the old version needed by delayed instructions

for read-after-write problem: need to distinguish different versions

solution: *have lots of extra registers*

… and assign each version a new 'real' register

called *register renaming*

register renaming

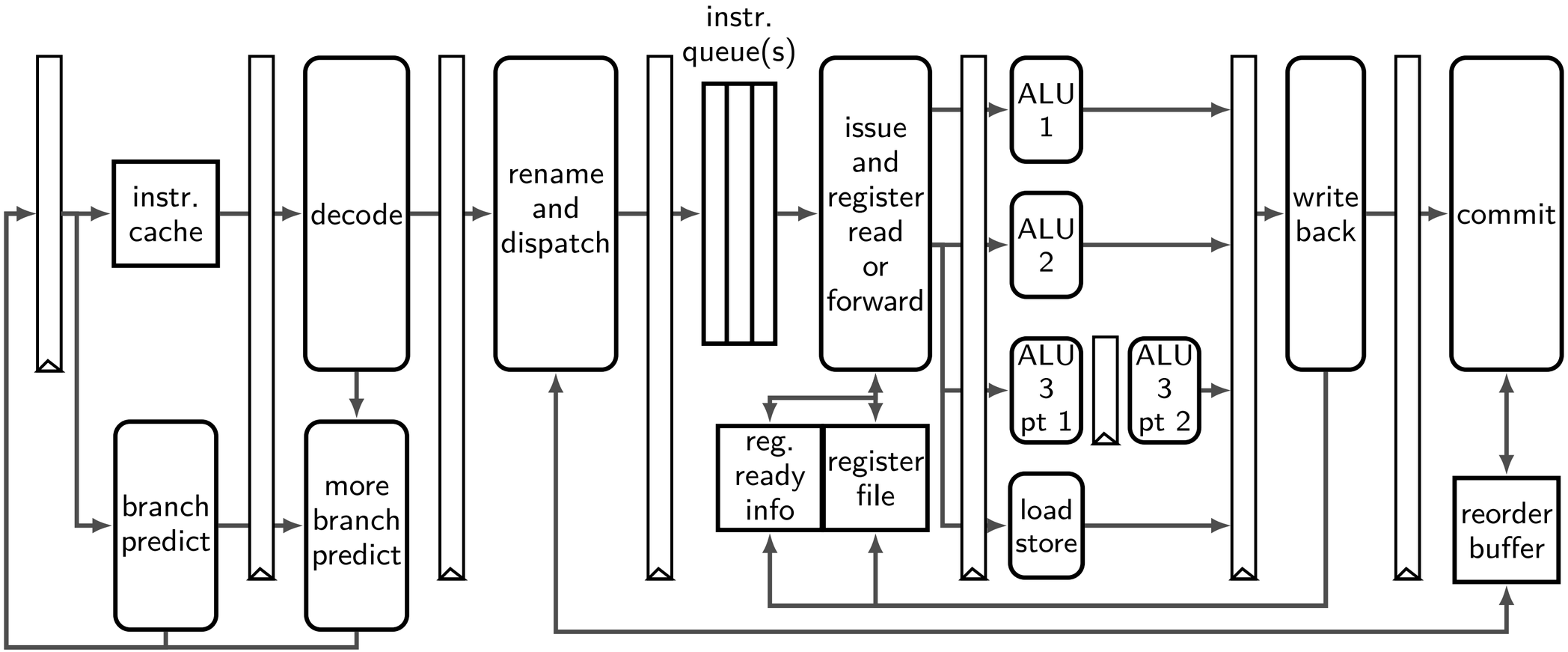
rename *architectural registers* to *physical registers*

different physical register for each version of architectural

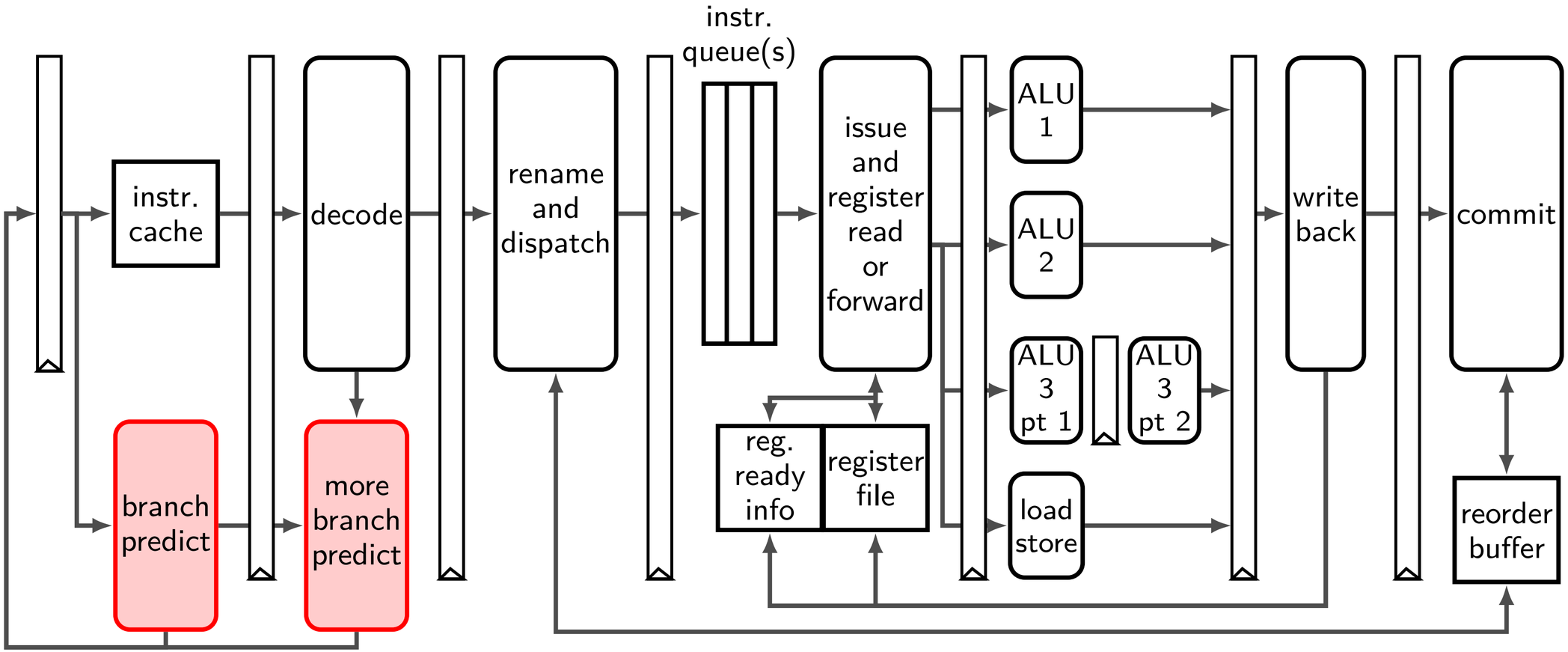
track which physical registers are ready

compare physical register numbers to do forwarding

an OOO pipeline

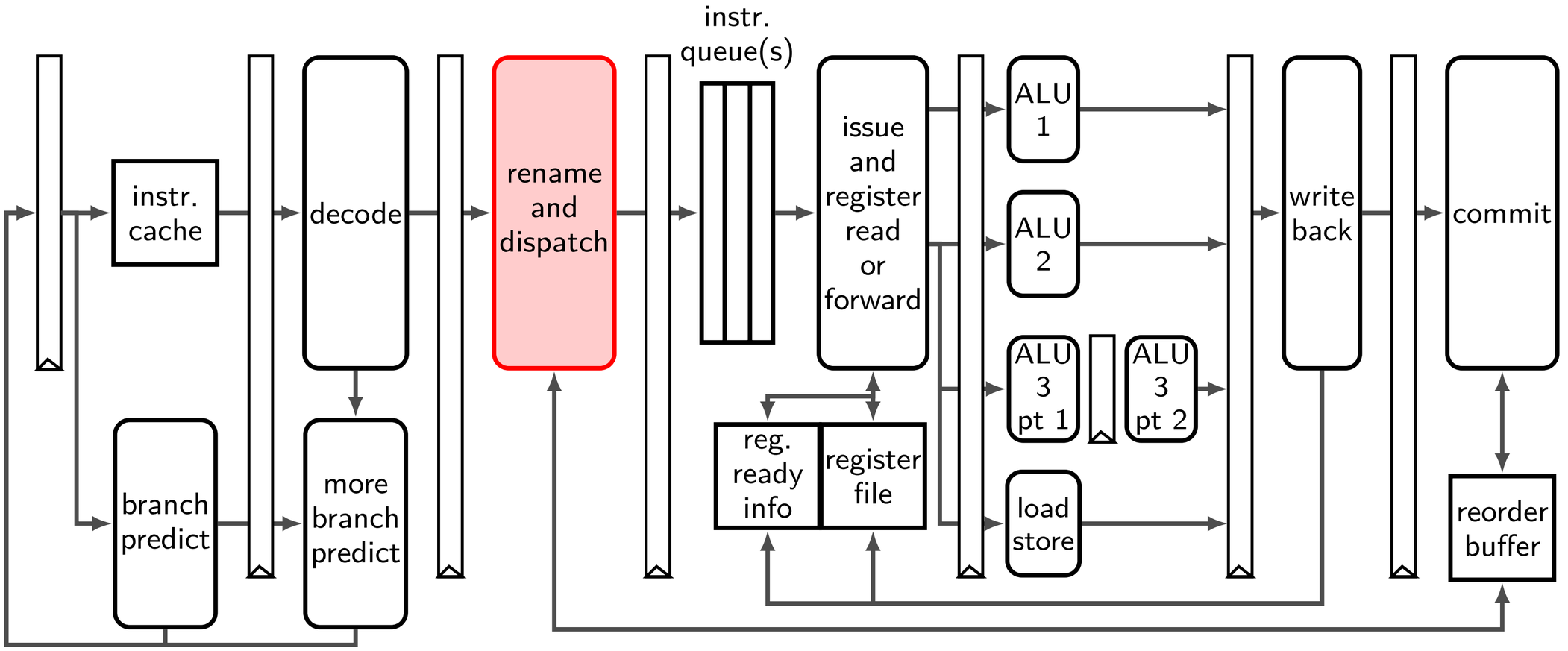


an OOO pipeline (branch prediction)



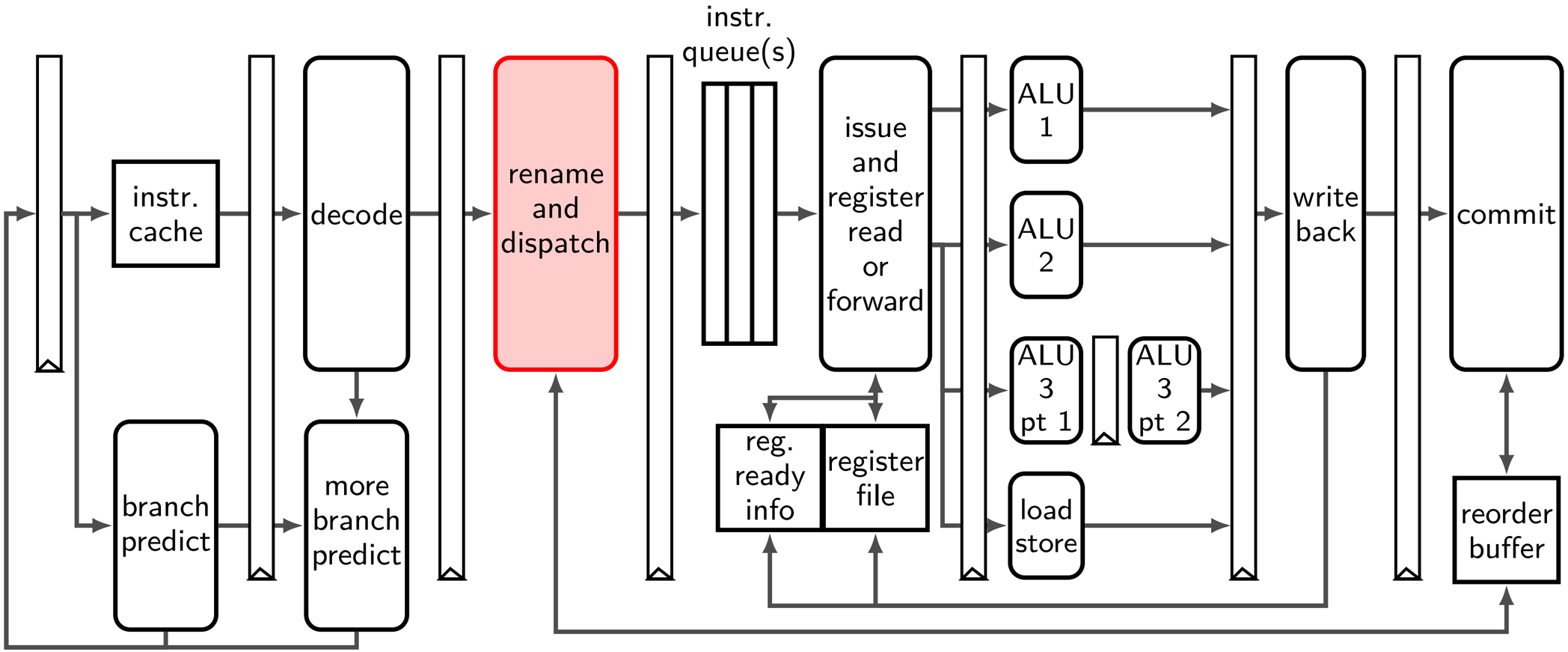
branch prediction needs to happen before instructions decoded
done with cache-like tables of information about recent branches

an OOO pipeline (register renaming)



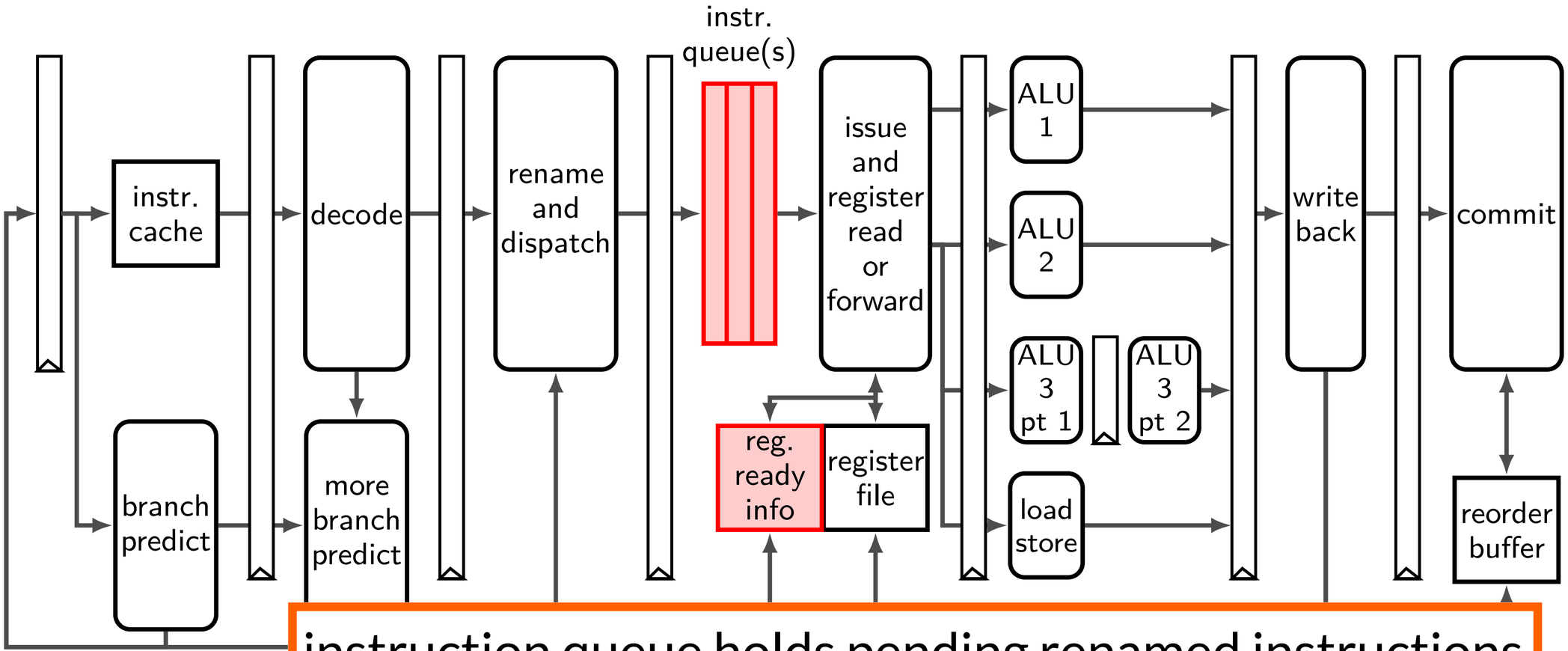
register renaming after decoding
where mapping from architectural to physical names kept

an OOO pipeline (register renaming)



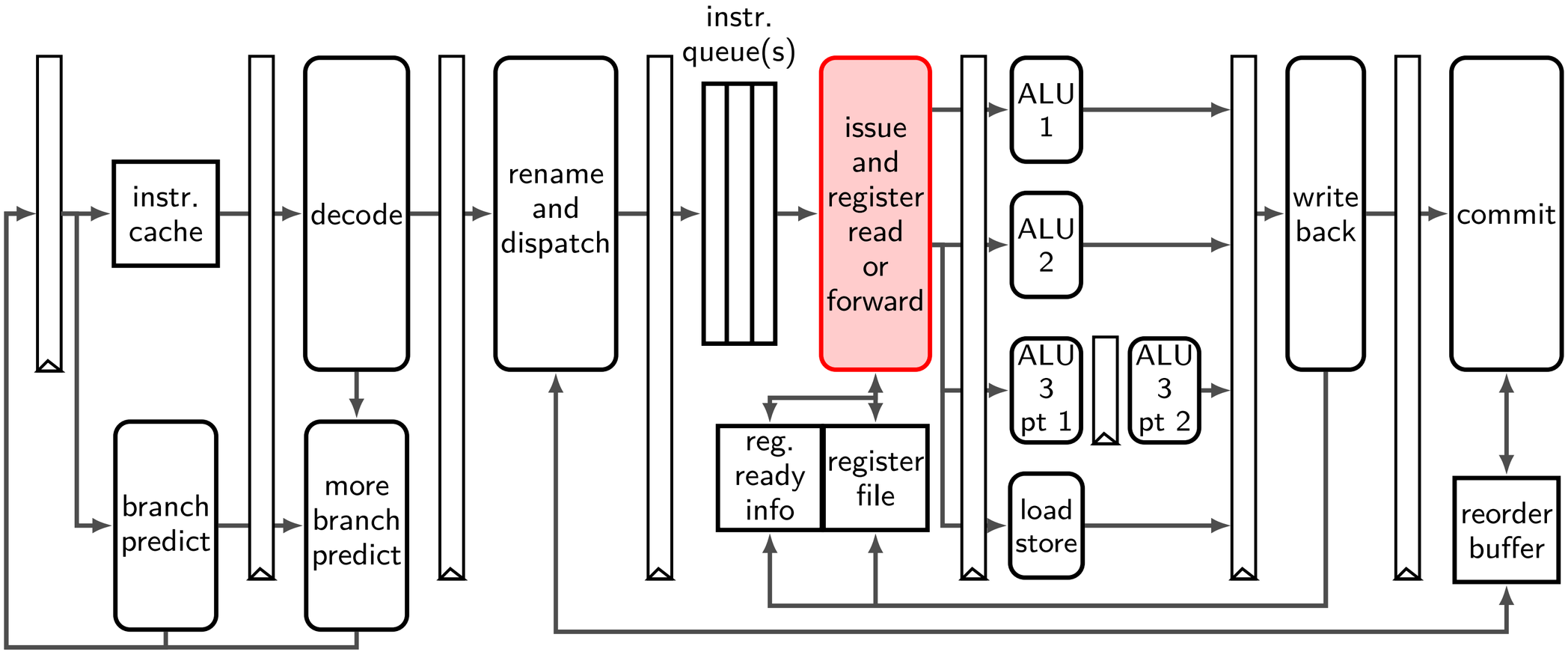
"dispatch" instructions = add to instruction queue
also requires preparing reorder buffer (for handling squashing)

an OOO pipeline (instruction queue)



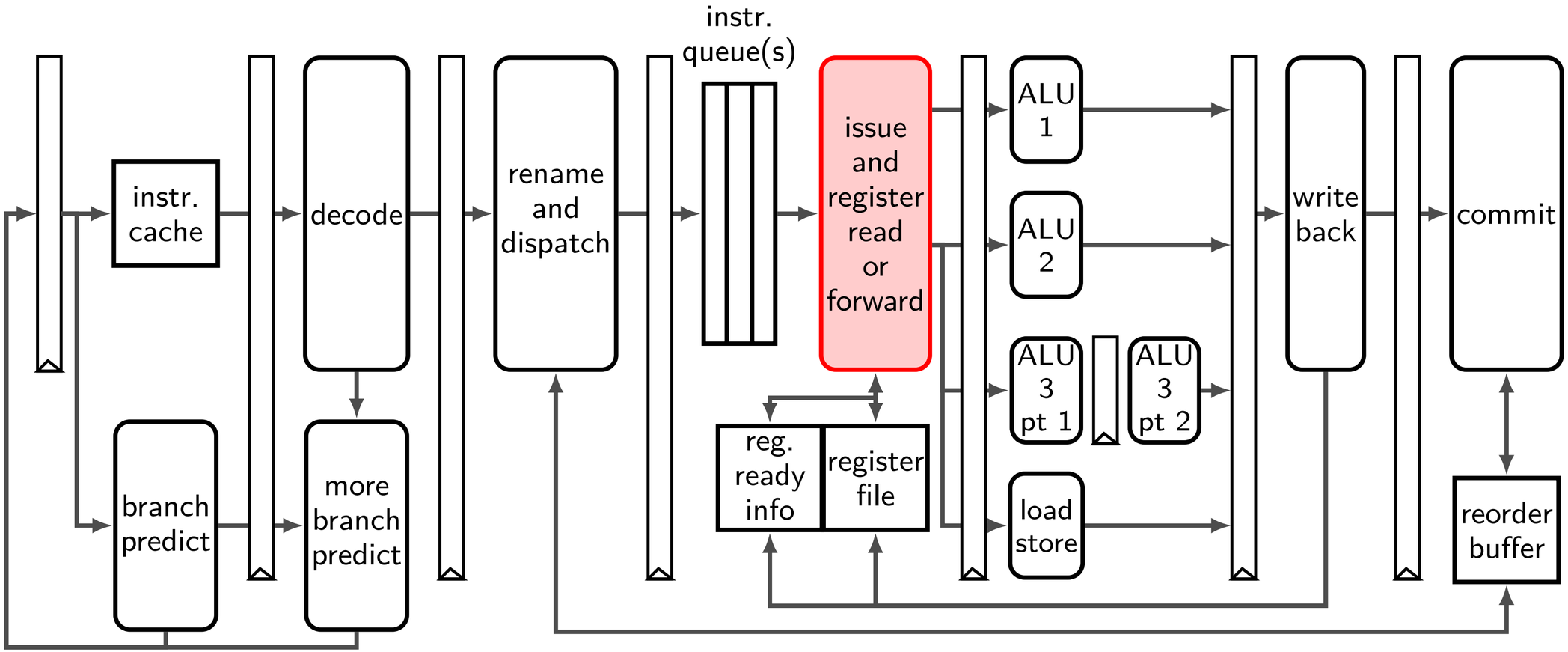
instruction queue holds pending renamed instructions combined with register-ready info to *issue* instructions (*issue* = start executing)

an OOO pipeline (starting instruction)



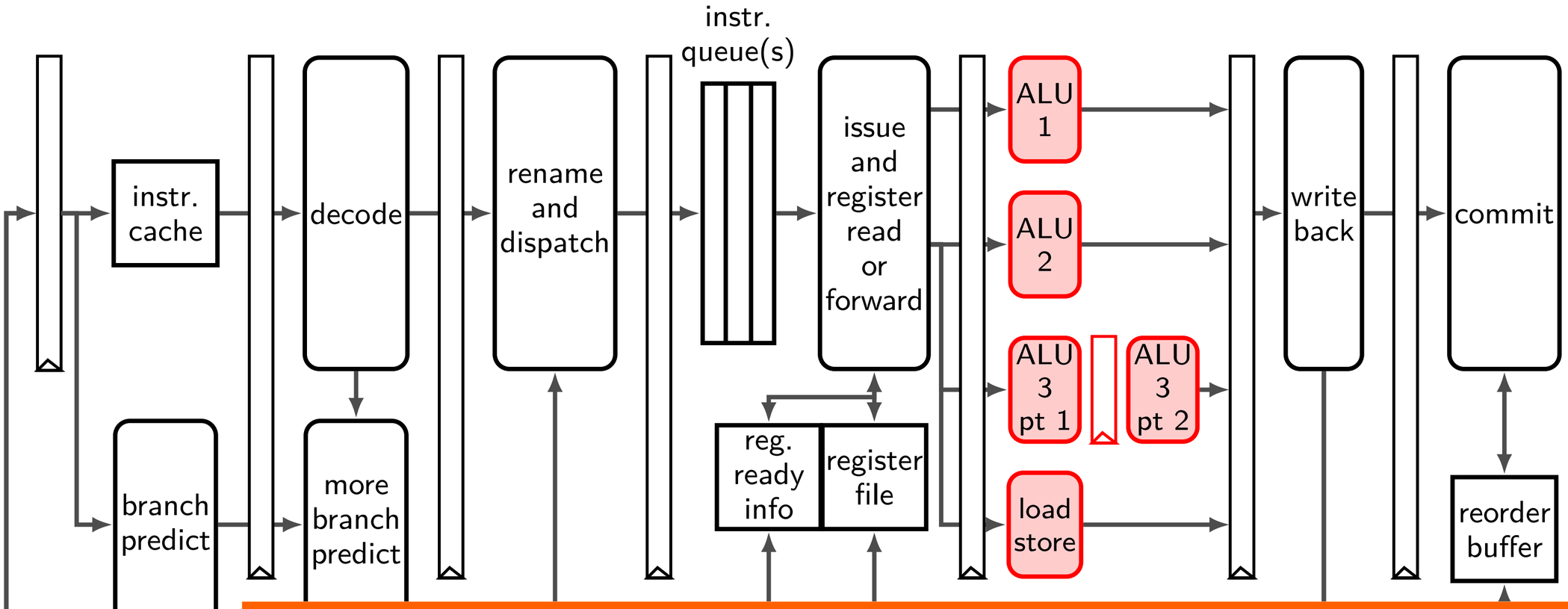
after selecting from instruction queue,
read from large register file and handle forwarding

an OOO pipeline (starting instruction)



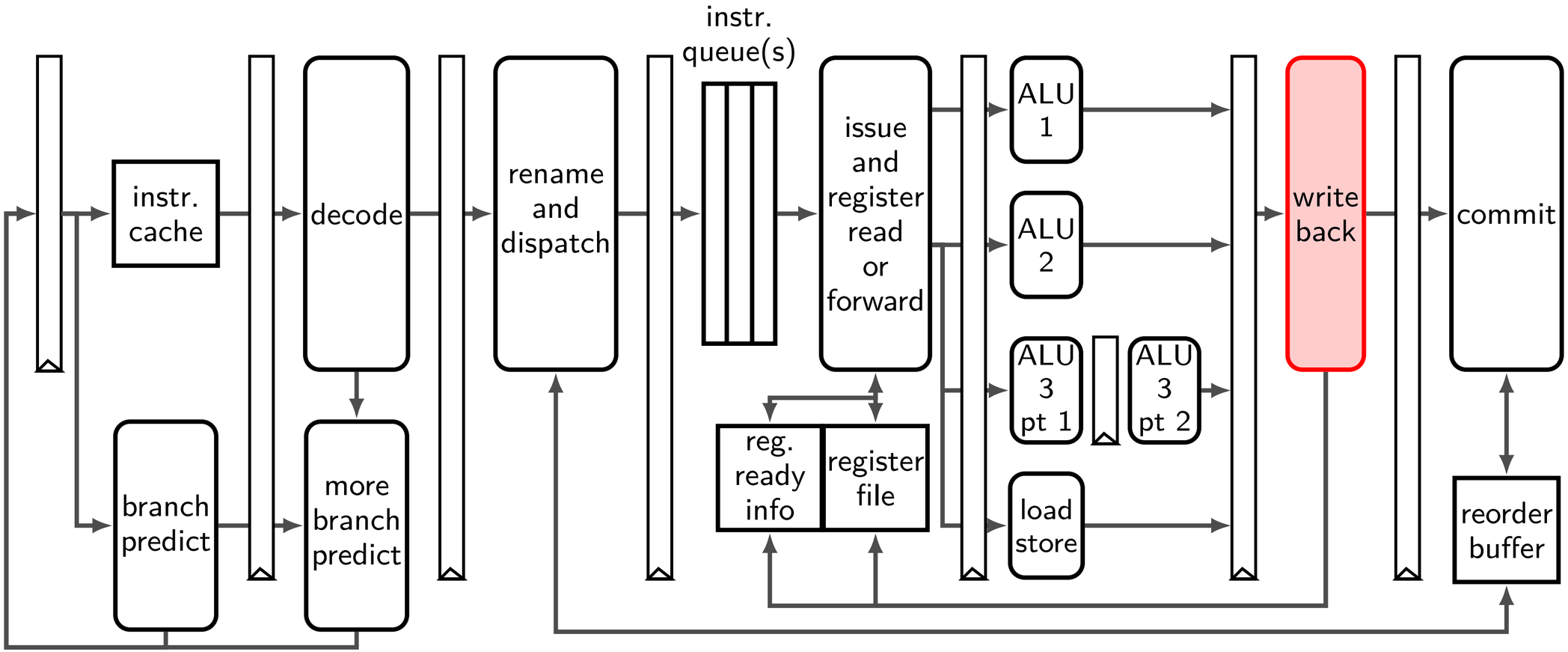
typically read 6+ registers at a time
extra data paths for forwarding (not drawn)

an OOO pipeline (execution units)



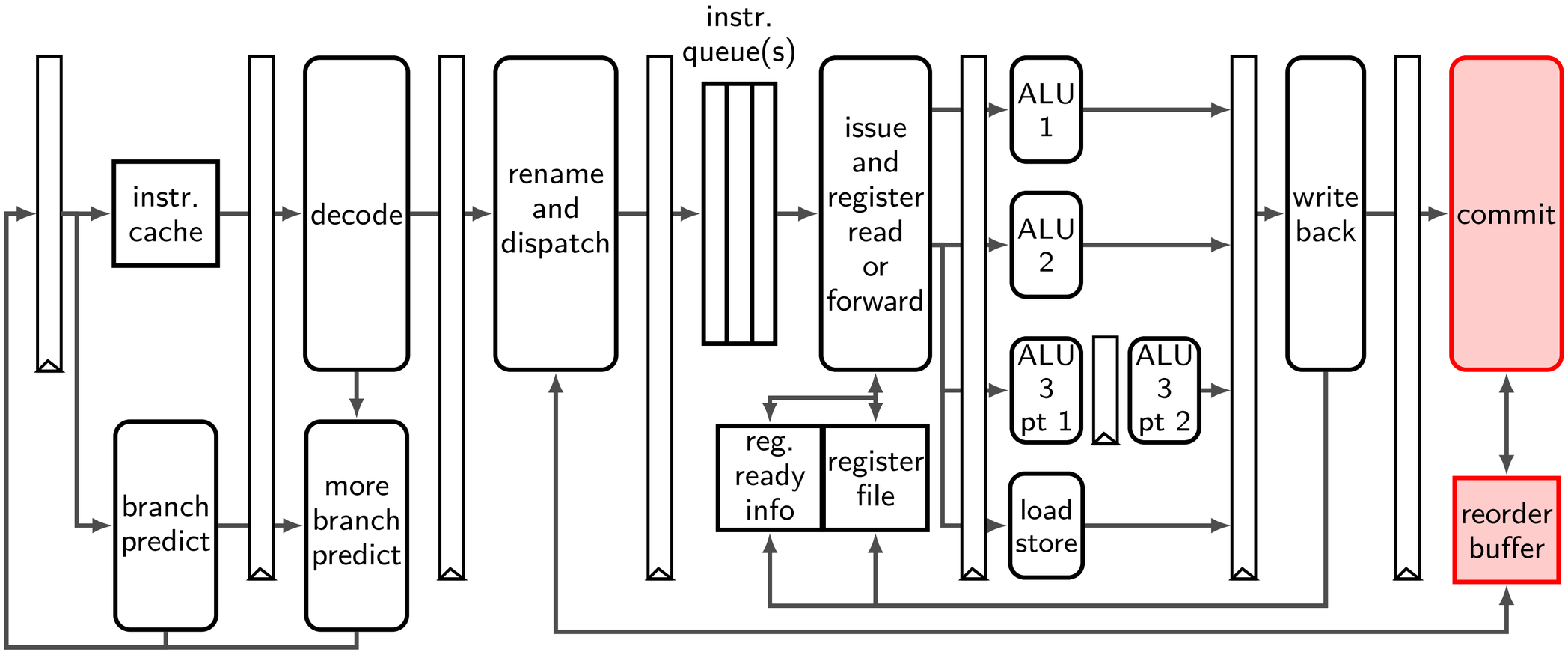
many *execution units* actually do math or cache load/store
some may have multiple pipeline stages
some may take variable time (data cache, integer divide, ...)

an OOO pipeline (writeback)



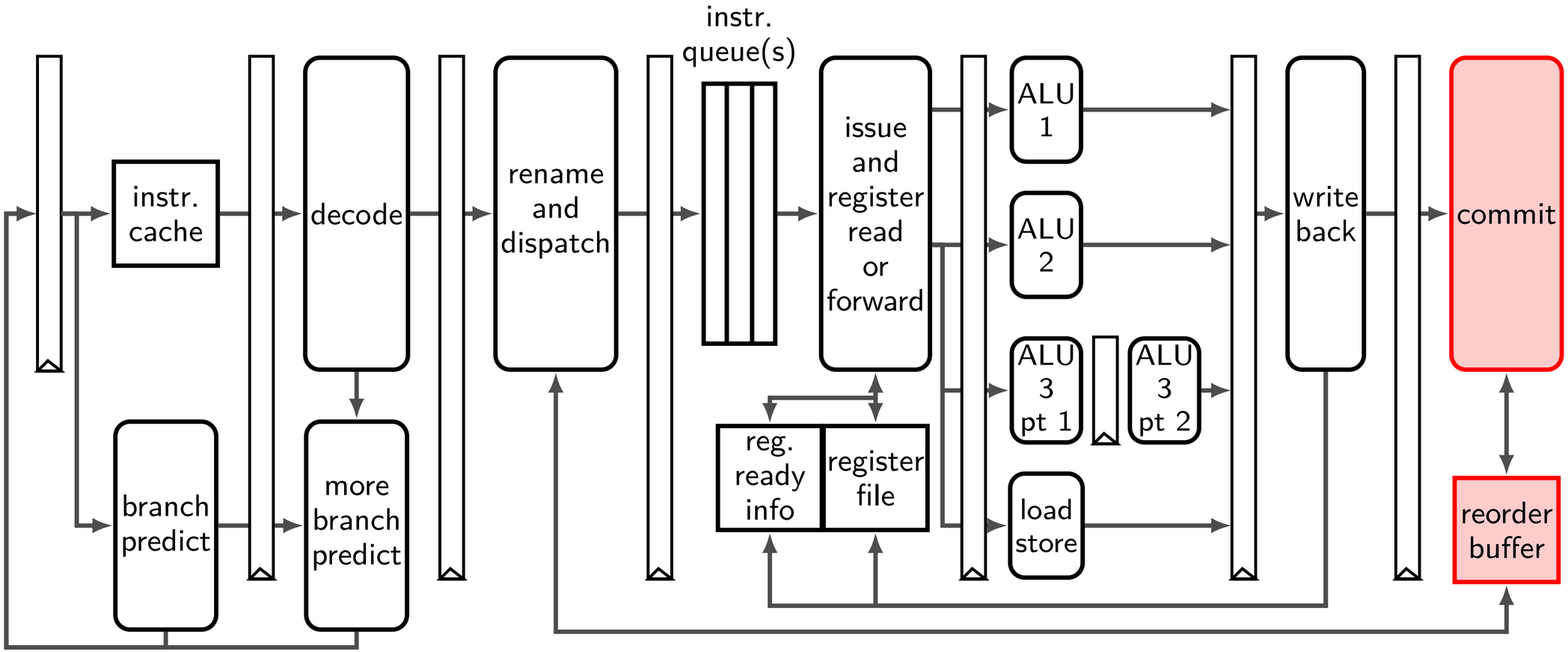
writeback to physical registers
register file typically supports writing 3+ registers at a time

an OOO pipeline (commit)



new commit (sometimes *retire*) stage finalized instruction
figures out when physical registers can be reused

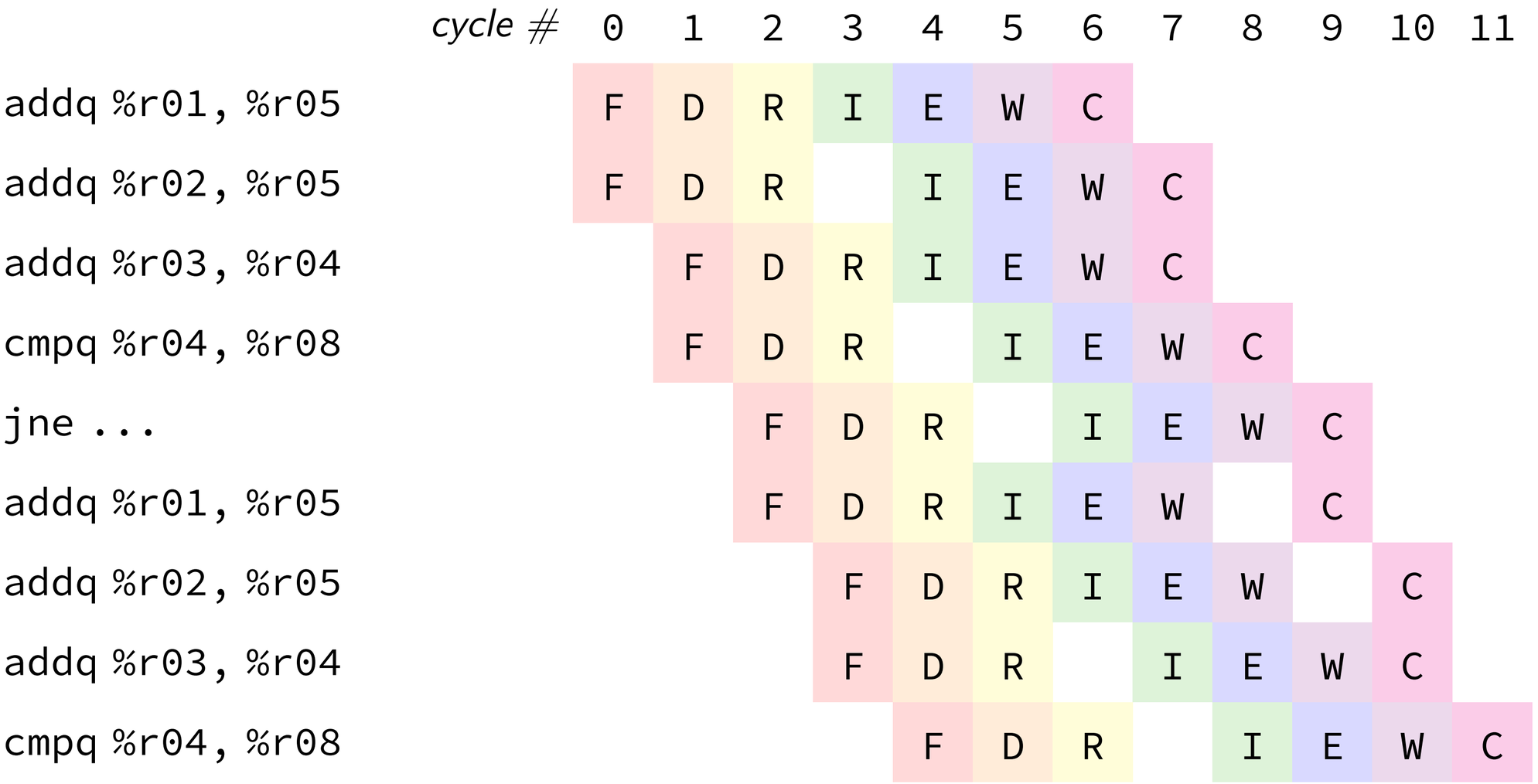
an OOO pipeline (commit)



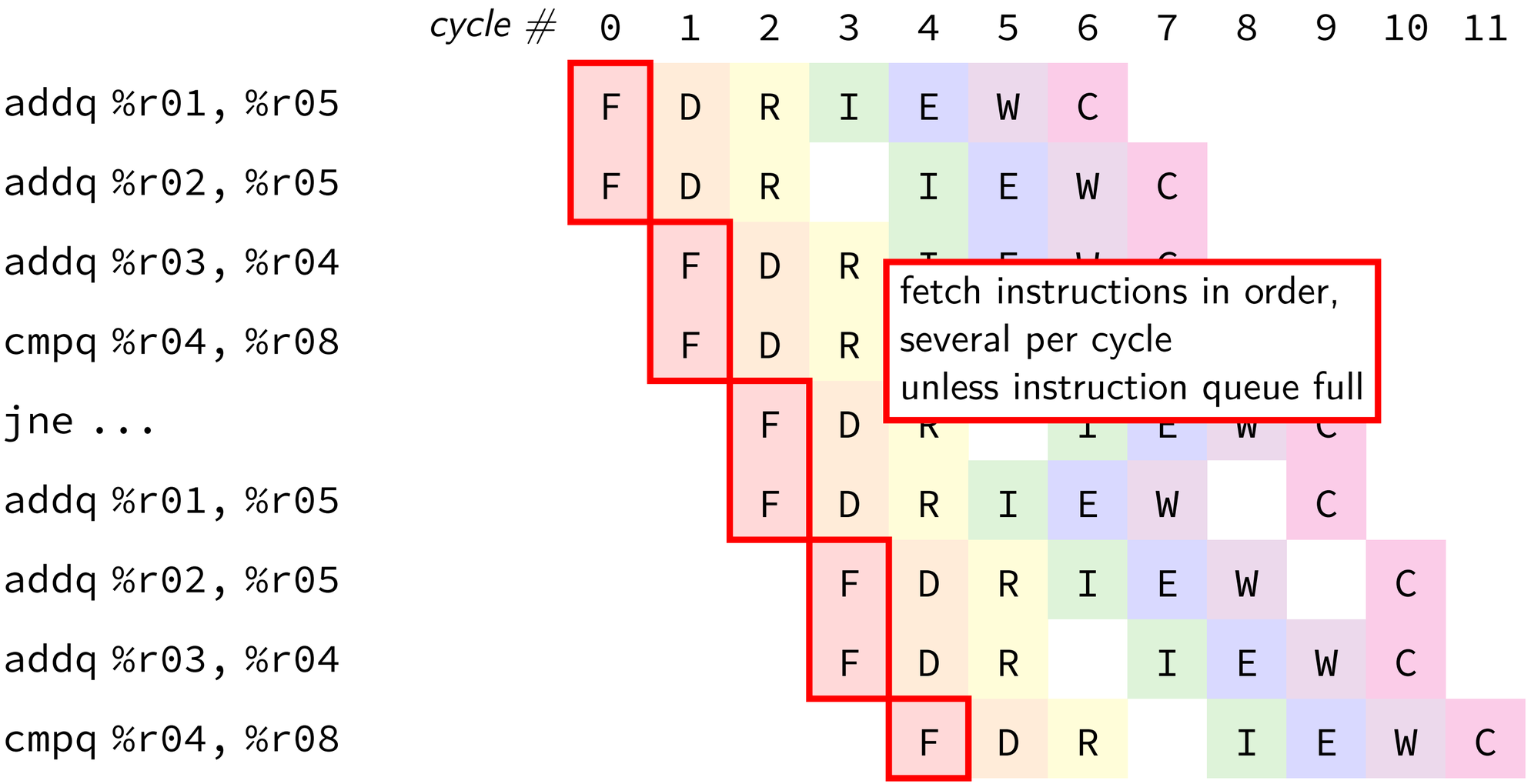
also tracks bookkeeping information if we need to undo instruction (because of branch misprediction/segfault/etc.)

an OOO pipeline diagram

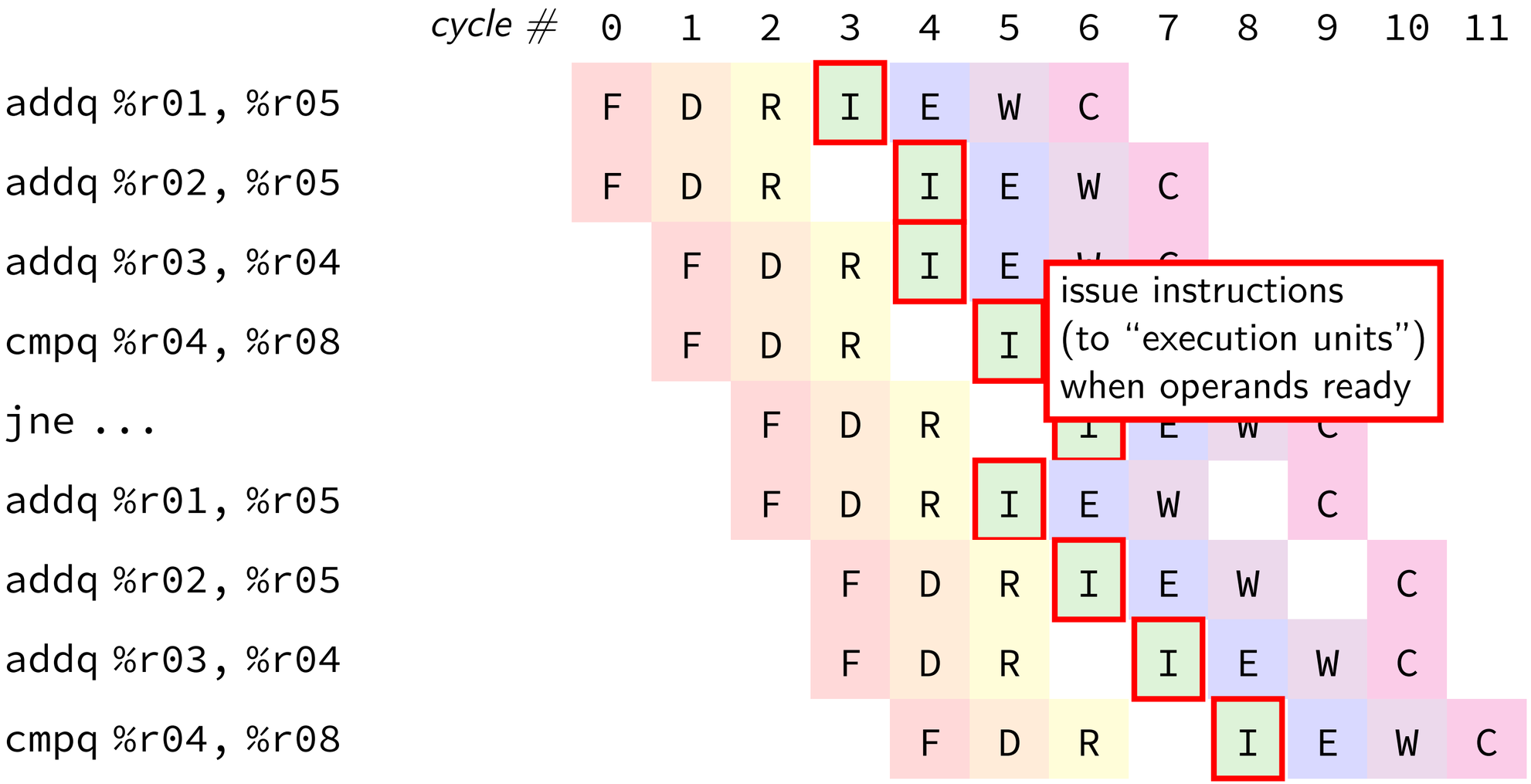
an OOO pipeline diagram



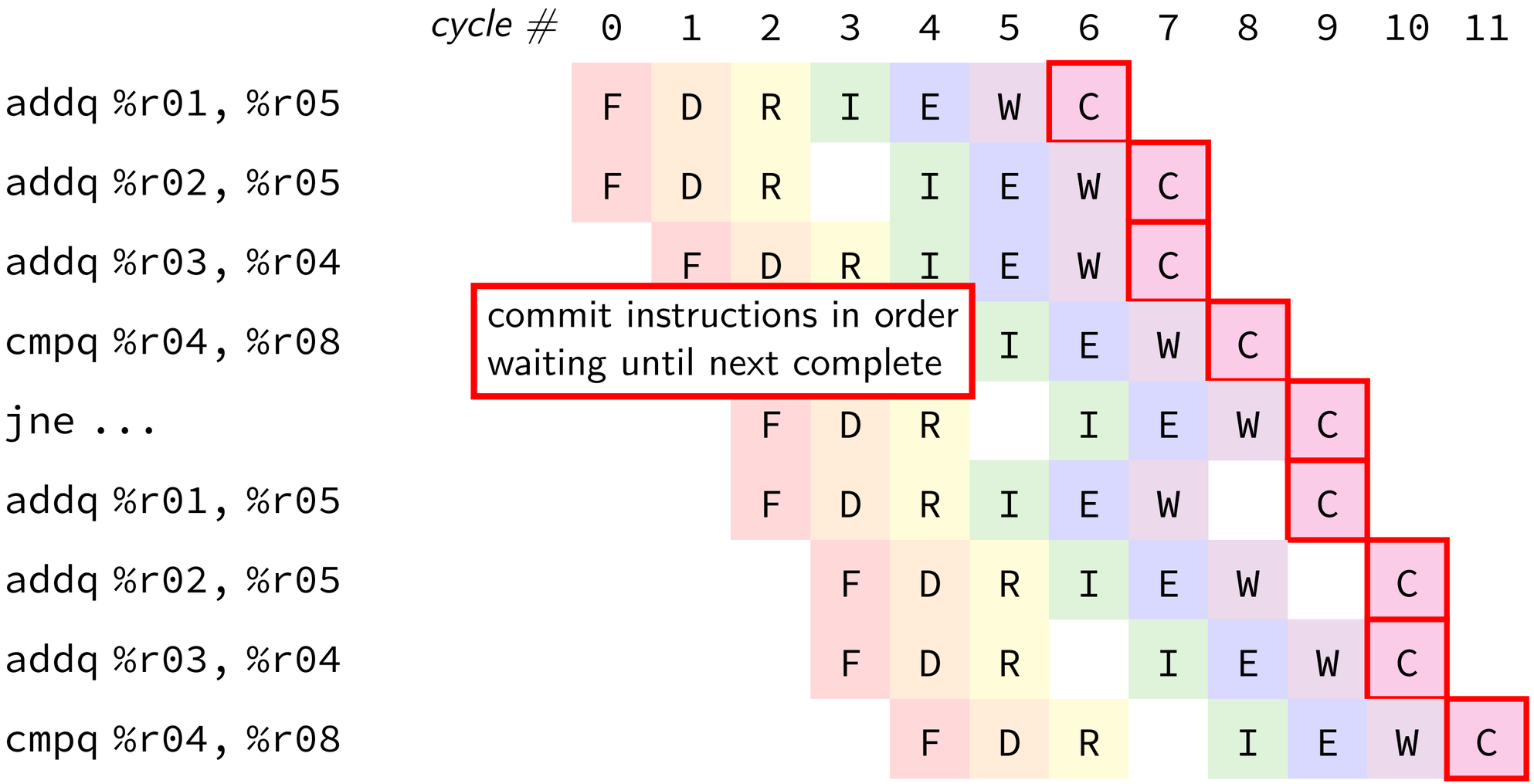
an OOO pipeline diagram



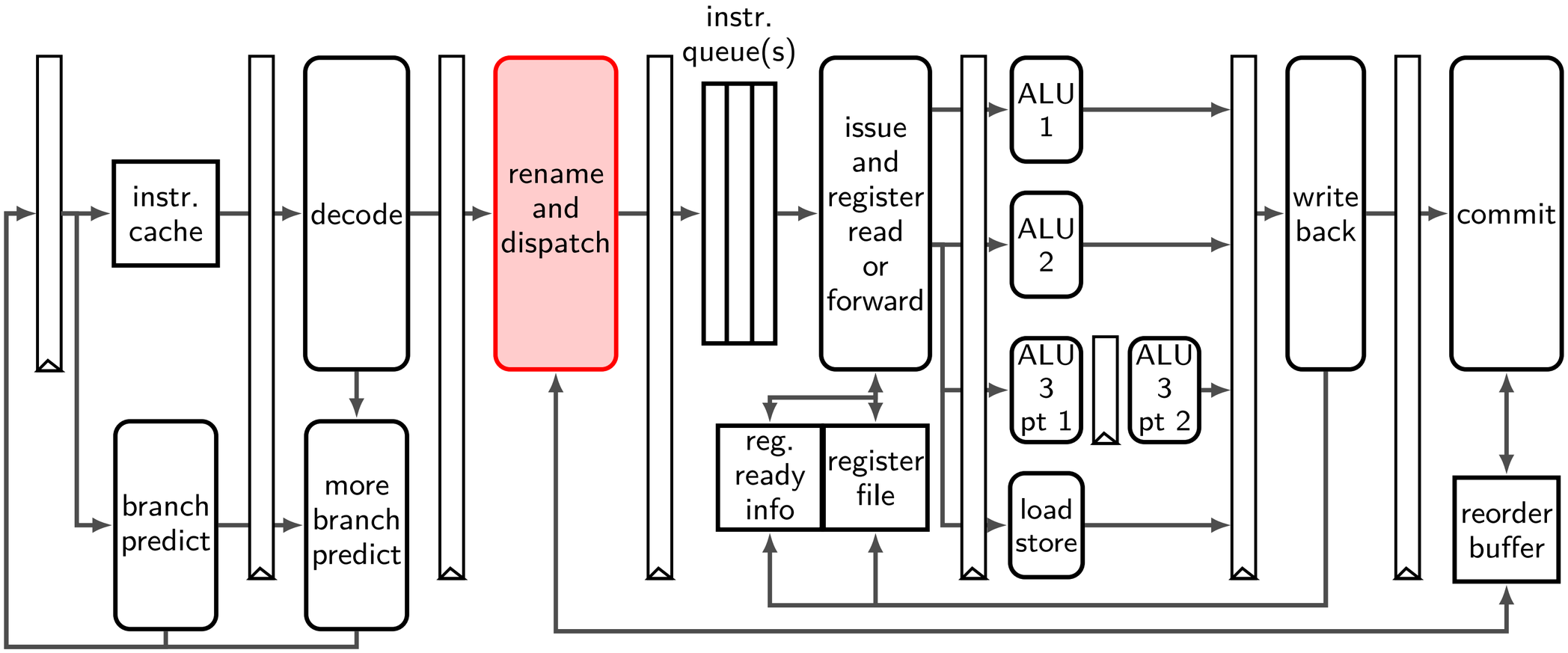
an OOO pipeline diagram



an OOO pipeline diagram



an OOO pipeline (register renaming)



register renaming

rename *architectural registers* to *physical registers*

- architectural = part of instruction set architecture

different name for each version of architectural register

register renaming state

register renaming state

original

```
add %r10, %r8    ...
add %r11, %r8    ...
add %r12, %r8    ...
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original

```
add %r10, %r8    ...
add %r11, %r8    ...
add %r12, %r8    ...
```

renamed

table for architectural (external) and physical (internal) name (for next instr. to process)

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming state

original

```
add %r10, %r8    ...
add %r11, %r8    ...
add %r12, %r8    ...
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
...	...
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
...	...

list of available physical registers added to as instructions finish

free reg list

%x18
%x20
%x21
%x23
%x24
...

register renaming example (1)

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

%x18
%x20
%x21
%x23
%x24
…

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
add %x07, %x18 → %x20
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~ ~~*%x18*~~ ~~*%x20*~~
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~*%x20*~~
%x21
%x23
%x24
…

register renaming example (1)

original	renamed
`add %r10, %r8`	`add %x19, %x13 → %x18`
`add %r11, %r8`	`add %x07, %x18 → %x20`
`add %r12, %r8`	`add %x05, %x20 → %x21`

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~~~%x18~~*~~%x20~~%x21*
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (1)

original

```
add %r10, %r8
add %r11, %r8
add %r12, %r8
```

renamed

```
add %x19, %x13 → %x18
add %x07, %x18 → %x20
add %x05, %x20 → %x21
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13%x18%x20~~%x21
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
…	…

free reg list

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (2)

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02
…	…

free regs

%x18
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
```

could be that %rax = 8+%r11
could load before value written!
possible data hazard!
not handled via register renaming
option 1: run load+stores in order
option 2: compare load/store addresses

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	%x07
%r12	%x05
%r13	%x02
…	…

%x20
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	~~%x07~~*%x20*
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
~~*%x20*~~
%x21
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
movq 8(%x20), (memory) → %x21
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~%x18
%r9	%x17
%r10	%x19
%r11	~~%x07~~*~~%x20~~%x21*
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
~~%x20~~
~~%x21~~
%x23
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
movq 8(%x20), (memory) → %x21
movq $100 → %x23
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~ ~~*%x18*~~ ~~%x23~~
%r9	%x17
%r10	%x19
%r11	~~%x07~~ ~~%x20~~ %x21
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
~~%x20~~
~~%x21~~
~~*%x23*~~
%x24
…

register renaming example (2)

original

```
addq %r10, %r8
movq %r8, (%rax)
subq %r8, %r11
movq 8(%r11), %r11
movq $100, %r8
addq %r11, %r8
```

renamed

```
addq %x19, %x13 → %x18
movq %x18, (%x04) → (memory)
subq %x18, %x07 → %x20
movq 8(%x20), (memory) → %x21
movq $100 → %x23
addq %x21, %x23 → %x24
```

arch → phys register map

%rax	%x04
%rcx	%x09
…	…
%r8	~~%x13~~ ~~%x18~~ *~~%x23~~* *~~%x24~~*
%r9	%x17
%r10	%x19
%r11	~~%x07~~ ~~%x20~~ %x21
%r12	%x05
%r13	%x02
…	…

free regs

~~%x18~~
~~%x20~~
~~%x21~~
~~%x23~~
~~%x24~~
…

register renaming exercise

original

```
addq %r8, %r9
movq $100, %r10
subq %r10, %r8
xorq %r8, %r9
andq %rax, %r9
```

renamed

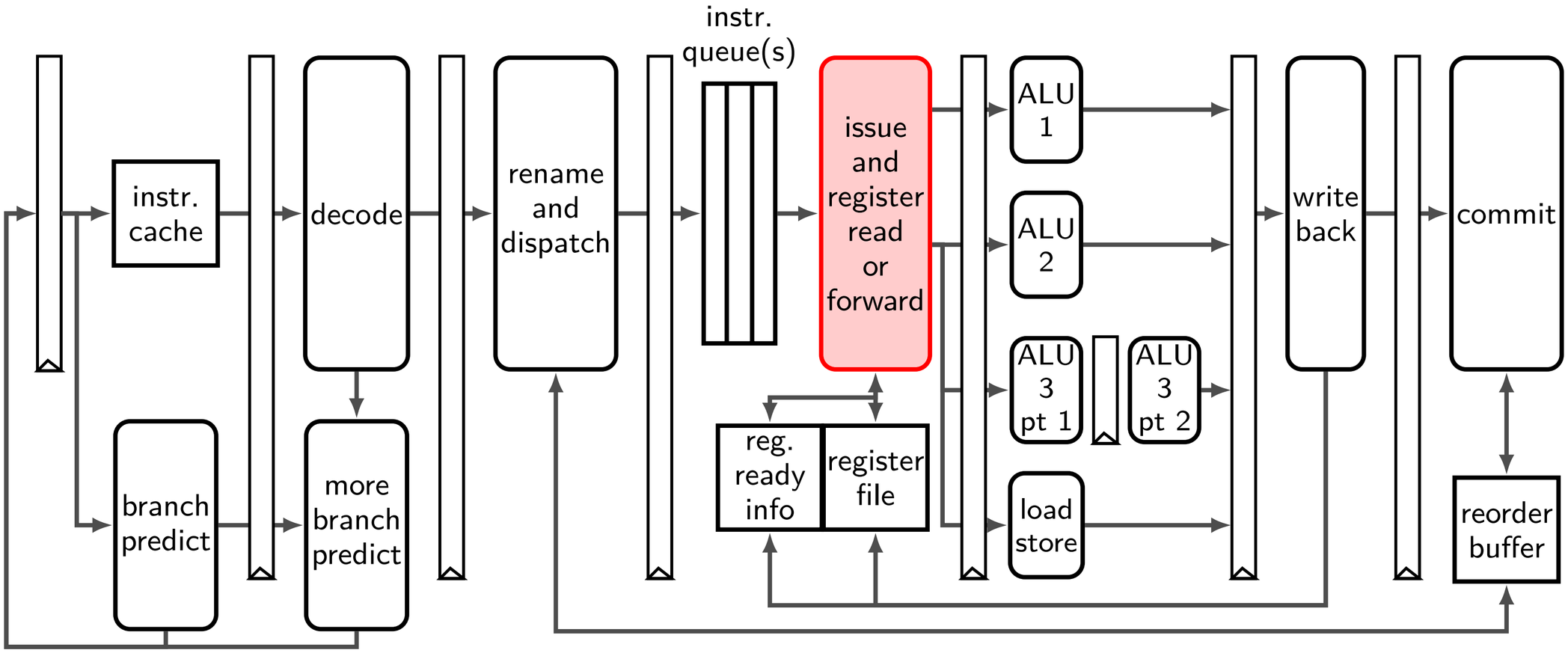
arch → phys

%rax	%x04
%rcx	%x09
…	…
%r8	%x13
%r9	%x17
%r10	%x19
%r11	%x29
%r12	%x05
%r13	%x02
…	…

free regs

%x18
%x20
%x21
%x23
%x24
…

an OOO pipeline (starting instruction)



instruction queue and dispatch

instruction queue and dispatch

instruction queue

#	*instruction*
1	**addq** %x01, %x05 → %x06
2	**addq** %x02, %x06 → %x07
3	**addq** %x03, %x07 → %x08
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit

ALU 1

ALU 2

...

instruction queue and dispatch

instruction queue

#	*instruction*
1	**addq** %x01, %x05 → %x06
2	**addq** %x02, %x06 → %x07
3	**addq** %x03, %x07 → %x08
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	...
ALU 1	**1**	
ALU 2		

instruction queue and dispatch

instruction queue

#	*instruction*
1	`addq %x01, %x05 → %x06`
2	`addq %x02, %x06 → %x07`
3	`addq %x03, %x07 → %x08`
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x10 → %x11`
8	`addq %x03, %x11 → %x12`
9	`cmpq %x04, %x12 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	pending
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	...
ALU 1	1	
ALU 2		

instruction queue and dispatch

instruction queue

#	*instruction*
1	**addq** %x01, %x05 → %x06
2	**addq** %x02, %x06 → %x07
3	**addq** %x03, %x07 → %x08
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ *ready*
%x07	pending
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*
ALU 1	1
ALU 2	—

...

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~**addq** %x01, %x05 → %x06~~
2	**addq** %x02, %x06 → %x07
3	**addq** %x03, %x07 → %x08
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ *ready*
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	...
ALU 1	1	**2**	
ALU 2	—	—	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~**addq** %x01, %x05 → %x06~~
~~2~~	~~**addq** %x02, %x06 → %x07~~
3	**addq** %x03, %x07 → %x08
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ *ready*
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	...
ALU 1	1	2	**3**	
ALU 2	—	—	—	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
4	`cmpq %x04, %x08 → %x09.cc`
5	`jne %x09.cc, ...`
6	`addq %x01, %x08 → %x10`
7	`addq %x02, %x10 → %x11`
8	`addq %x03, %x11 → %x12`
9	`cmpq %x04, %x12 → %x13.cc`
...	...

scoreboard

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	~~pending~~ ready
`%x08`	~~pending~~ ready
`%x09`	pending
`%x10`	pending
`%x11`	pending
`%x12`	pending
`%x13`	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	...
ALU 1	1	2	3	
ALU 2	—	—	—	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~**addq** %x01, %x05 → %x06~~
~~2~~	~~**addq** %x02, %x06 → %x07~~
~~3~~	~~**addq** %x03, %x07 → %x08~~
4	**cmpq** %x04, %x08 → %x09.cc
5	**jne** %x09.cc, ...
6	**addq** %x01, %x08 → %x10
7	**addq** %x02, %x10 → %x11
8	**addq** %x03, %x11 → %x12
9	**cmpq** %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ *ready*
%x10	~~pending~~ *ready*
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	...
ALU 1	1	2	3	**4**	
ALU 2	—	—	—	**6**	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~addq %x01, %x05 → %x06~~
~~2~~	~~addq %x02, %x06 → %x07~~
~~3~~	~~addq %x03, %x07 → %x08~~
~~4~~	~~cmpq %x04, %x08 → %x09.cc~~
5	jne %x09.cc, ...
~~6~~	~~addq %x01, %x08 → %x10~~
7	addq %x02, %x10 → %x11
8	addq %x03, %x11 → %x12
9	cmpq %x04, %x12 → %x13.cc
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	pending
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	...
ALU 1	1	2	3	4	
ALU 2	—	—	—	6	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
~~4~~	~~`cmpq %x04, %x08 → %x09.cc`~~
~~5~~	~~`jne %x09.cc, ...`~~
~~6~~	~~`addq %x01, %x08 → %x10`~~
~~7~~	~~`addq %x02, %x10 → %x11`~~
8	`addq %x03, %x11 → %x12`
9	`cmpq %x04, %x12 → %x13.cc`
...	...

scoreboard

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	~~pending~~ ready
`%x08`	~~pending~~ ready
`%x09`	~~pending~~ ready
`%x10`	~~pending~~ ready
`%x11`	~~pending~~ *ready*
`%x12`	pending
`%x13`	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	...
ALU 1	1	2	3	4	**5**	
ALU 2	—	—	—	6	**7**	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
~~4~~	~~`cmpq %x04, %x08 → %x09.cc`~~
~~5~~	~~`jne %x09.cc, ...`~~
~~6~~	~~`addq %x01, %x08 → %x10`~~
~~7~~	~~`addq %x02, %x10 → %x11`~~
~~8~~	~~`addq %x03, %x11 → %x12`~~
9	`cmpq %x04, %x12 → %x13.cc`
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	~~pending~~ ready
%x12	pending
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	...
ALU 1	1	2	3	4	5	**8**	
ALU 2	—	—	—	6	7	—	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~addq %x01, %x05 → %x06~~
~~2~~	~~addq %x02, %x06 → %x07~~
~~3~~	~~addq %x03, %x07 → %x08~~
~~4~~	~~cmpq %x04, %x08 → %x09.cc~~
~~5~~	~~jne %x09.cc, ...~~
~~6~~	~~addq %x01, %x08 → %x10~~
~~7~~	~~addq %x02, %x10 → %x11~~
~~8~~	~~addq %x03, %x11 → %x12~~
~~9~~	~~cmpq %x04, %x12 → %x13.cc~~
...	...

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	~~pending~~ ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	~~pending~~ ready
%x11	~~pending~~ ready
%x12	~~pending~~ *ready*
%x13	pending
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	...
ALU 1	1	2	3	4	5	8	**9**	
ALU 2	—	—	—	6	7	—	**...**	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
~~4~~	~~`cmpq %x04, %x08 → %x09.cc`~~
~~5~~	~~`jne %x09.cc, ...`~~
~~6~~	~~`addq %x01, %x08 → %x10`~~
~~7~~	~~`addq %x02, %x10 → %x11`~~
~~8~~	~~`addq %x03, %x11 → %x12`~~
~~9~~	~~`cmpq %x04, %x12 → %x13.cc`~~
...	...

scoreboard

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	~~pending~~ ready
`%x08`	~~pending~~ ready
`%x09`	~~pending~~ ready
`%x10`	~~pending~~ ready
`%x11`	~~pending~~ ready
`%x12`	~~pending~~ ready
`%x13`	~~pending~~ *ready*
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	...
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	...	

instruction queue and dispatch

instruction queue

#	*instruction*
~~1~~	~~`addq %x01, %x05 → %x06`~~
~~2~~	~~`addq %x02, %x06 → %x07`~~
~~3~~	~~`addq %x03, %x07 → %x08`~~
~~4~~	~~`cmpq %x04, %x08 → %x09.cc`~~
~~5~~	~~`jne %x09.cc, ...`~~
~~6~~	~~`addq %x01, %x08 → %x10`~~
~~7~~	~~`addq %x02, %x10 → %x11`~~
~~8~~	~~`addq %x03, %x11 → %x12`~~
~~9~~	~~`cmpq %x04, %x12 → %x13.cc`~~
...	...

scoreboard

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	~~pending~~ ready
`%x08`	~~pending~~ ready
`%x09`	~~pending~~ ready
`%x10`	~~pending~~ ready
`%x11`	~~pending~~ ready
`%x12`	~~pending~~ ready
`%x13`	~~pending~~ ready
...	...

execution unit	*cycle# 1*	*2*	*3*	*4*	*5*	*6*	*7*	...
ALU 1	1	2	3	4	5	8	9	
ALU 2	—	—	—	6	7	—	...	

instruction queue and dispatch ex

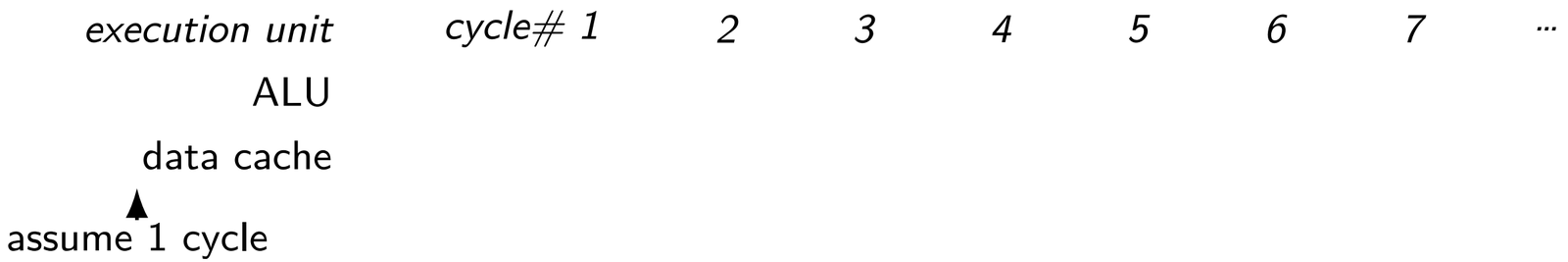
instruction queue and dispatch ex

instruction queue

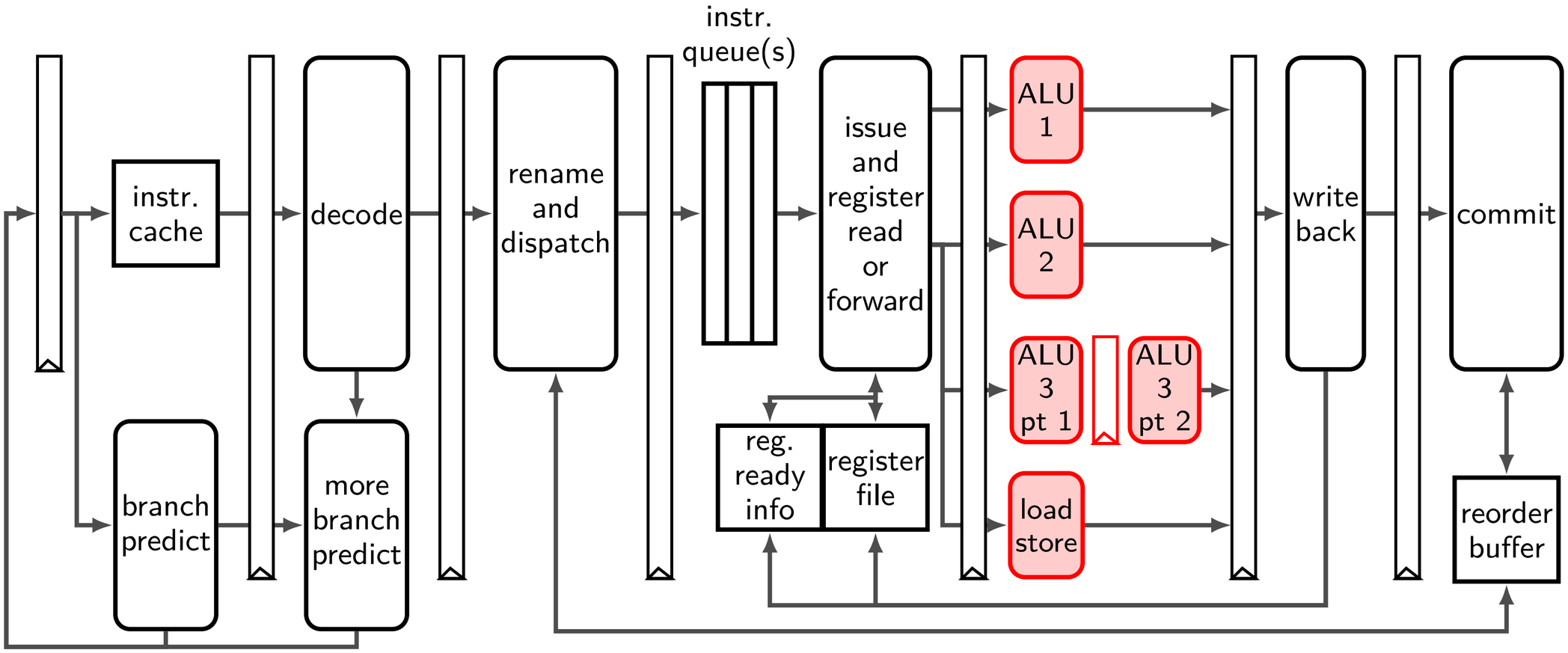
#	*instruction*
1	`movq (%x04) → %x06`
2	`movq (%x05) → %x07`
3	`addq %x01, %x02 → %x08`
4	`addq %x01, %x06 → %x09`
5	`addq %x01, %x07 → %x10`
…	…

scoreboard

reg	*status*
%x01	ready
%x02	ready
%x03	ready
%x04	ready
%x05	ready
%x06	
%x07	
%x08	
%x09	
%x10	
…	…



an OOO pipeline (execution units)



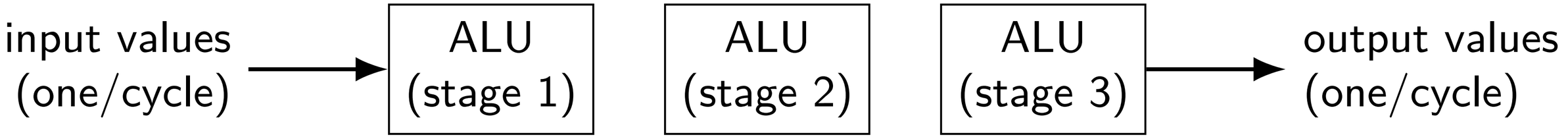
execution units AKA functional units (1a)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

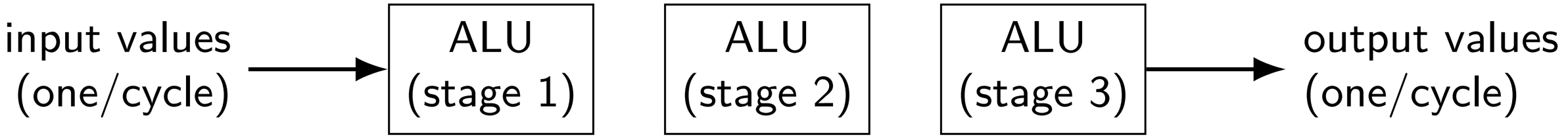
execution units AKA functional units (1a)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes pipelined:

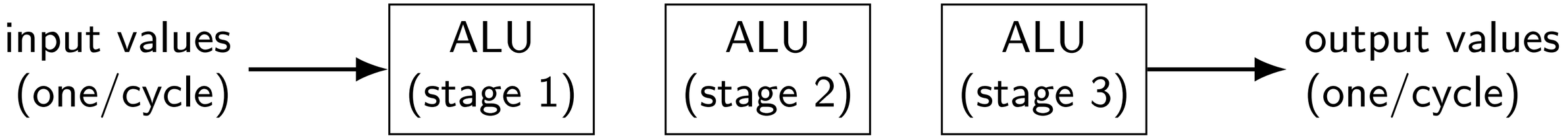
(here: 1 op/cycle; 3 cycle latency)



exercise: how long to compute $A \times (B \times (C \times D))$?

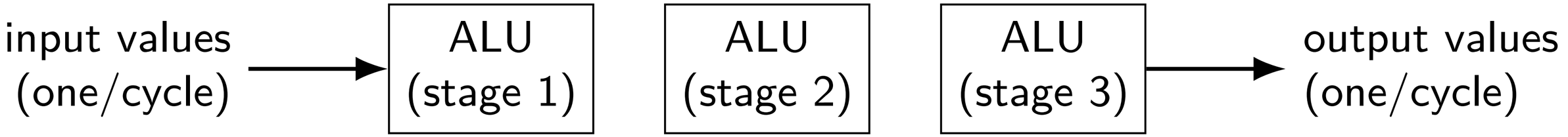
3×3 cycles + any time to forward values

execution units AKA functional units (1b)



exercise: how long to compute $(A \times B) \times (C \times D)$?

execution units AKA functional units (1b)



exercise: how long to compute $(A \times B) \times (C \times D)$?

assuming just one ALU:

	1	**2**	**3**	**4**	**5**	**6**	**7**
$A \times B$	1	2	3				
$C \times D$		1	2	3			
final					1	2	3

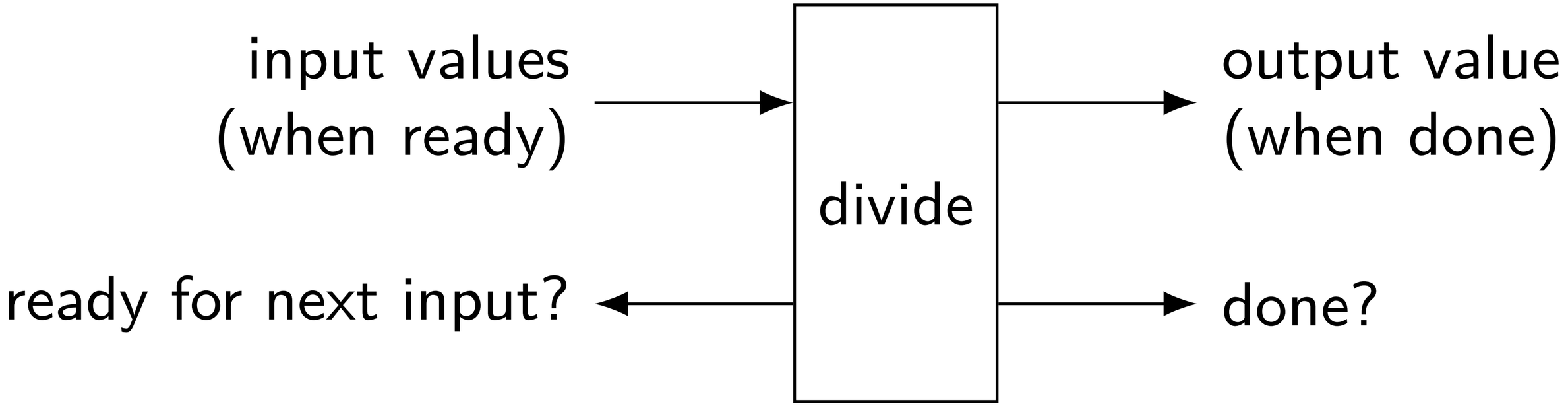
exercise part 2: how much would second ALU help?

execution units AKA functional units (2)

where actual work of instruction is done

e.g. the actual ALU, or data cache

sometimes unpipelined:



instruction queue and dispatch (multicycle)

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	pending
`%x04`	ready
`%x05`	ready
`%x06`	pending
`%x07`	ready
`%x08`	pending
`%x09`	pending
`%x10`	pending
`%x11`	pending
`%x12`	pending
`%x13`	pending
`%x14`	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit

ALU 1 (add, cmp, jxx)

ALU 2 (add, cmp, jxx)

ALU 3 (mul) start

ALU 3 (mul) end

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	pending
`%x04`	ready
`%x05`	ready
`%x06`	pending
`%x07`	ready
`%x08`	pending
`%x09`	pending
`%x10`	pending
`%x11`	pending
`%x12`	pending
`%x13`	pending
`%x14`	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
1	`add %x01, %x02 → %x03`
2	`imul %x04, %x05 → %x06`
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit	**cycle#**	**1**	**2**
ALU 1 (add, cmp, jxx)		**1**	
ALU 2 (add, cmp, jxx)		**–**	
ALU 3 (mul) start		**2**	
ALU 3 (mul) end			**2**

reg	*status*
%x01	ready
%x02	ready
%x03	pending
%x04	ready
%x05	ready
%x06	pending
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
3	`imul %x03, %x07 → %x08`
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
6	`add %x01, %x03 → %x11`
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit	*cycle# 1*		
		2	
ALU 1 (add, cmp, jxx)	1	**6**	
ALU 2 (add, cmp, jxx)	–	**–**	
ALU 3 (mul) start	2	**3**	
ALU 3 (mul) end		2	**3**

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ *ready*
%x04	ready
%x05	ready
%x06	pending *(still)*
%x07	ready
%x08	pending
%x09	pending
%x10	pending
%x11	pending
%x12	pending
%x13	pending
%x14	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
4	`cmp %x03, %x08 → %x09.cc`
5	`jle %x09.cc, ...`
~~6~~	~~`add %x01, %x03 → %x11`~~
7	`imul %x04, %x06 → %x12`
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit	*cycle# 1*	*2*	**3**	
ALU 1 (add, cmp, jxx)	1	6	–	
ALU 2 (add, cmp, jxx)	–	–	–	
ALU 3 (mul) start	2	3	**7**	
ALU 3 (mul) end		2	3	**7**

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ *ready*
%x07	ready
%x08	pending *(still)*
%x09	pending
%x10	pending
%x11	~~pending~~ *ready*
%x12	pending
%x13	pending
%x14	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
5	`jle %x09.cc, ...`
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit	*cycle#* 1	2	3	**4**	
ALU 1 (add, cmp, jxx)	1	6	–	**4**	
ALU 2 (add, cmp, jxx)	–	–	–	**–**	
ALU 3 (mul) start	2	3	7	**8**	
ALU 3 (mul) end		2	3	7	**8**

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ *ready*
%x09	~~pending~~ *ready*
%x10	pending
%x11	~~pending~~ ready
%x12	pending *(still)*
%x13	pending
%x14	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
8	`imul %x03, %x08 → %x13`
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

execution unit	*cycle#* 1	2	3	4	**5**
ALU 1 (add, cmp, jxx)	1	6	–	4	**5**
ALU 2 (add, cmp, jxx)	–	–	–	–	–
ALU 3 (mul) start	2	3	7	8	–
ALU 3 (mul) end		2	3	7	8

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ *ready*
%x13	pending *(still)*
%x14	pending
...	...

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
~~8~~	~~`imul %x03, %x08 → %x13`~~
9	`cmp %x11, %x13 → %x14.cc`
10	`jle %x14.cc, ...`
...	...

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ *ready*
%x14	pending
...	...

execution unit	*cycle#* *1*	*2*	*3*	*4*	*5*
ALU 1 (add, cmp, jxx)	1	6	–	4	5
ALU 2 (add, cmp, jxx)	–	–	–	–	–
ALU 3 (mul) start	2	3	7	8	–
ALU 3 (mul) end		2	3	7	8

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
~~8~~	~~`imul %x03, %x08 → %x13`~~
~~9~~	~~`cmp %x11, %x13 → %x14.cc`~~
10	`jle %x14.cc, ...`
...	...

reg	*status*
%x01	ready
%x02	ready
%x03	~~pending~~ ready
%x04	ready
%x05	ready
%x06	~~pending~~ ready
%x07	ready
%x08	~~pending~~ ready
%x09	~~pending~~ ready
%x10	pending
%x11	~~pending~~ ready
%x12	~~pending~~ ready
%x13	~~pending~~ ready
%x14	~~pending~~ *ready*
...	...

execution unit	*cycle#* 1	2	3	4	5	**6**
ALU 1 (add, cmp, jxx)	1	6	–	4	5	**9**
ALU 2 (add, cmp, jxx)	–	–	–	–	–	**–**
ALU 3 (mul) start	2	3	7	8	–	
ALU 3 (mul) end		2	3	7	8	

instruction queue and dispatch (multicycle)

instruction queue

#	*instruction*
~~1~~	~~`add %x01, %x02 → %x03`~~
~~2~~	~~`imul %x04, %x05 → %x06`~~
~~3~~	~~`imul %x03, %x07 → %x08`~~
~~4~~	~~`cmp %x03, %x08 → %x09.cc`~~
~~5~~	~~`jle %x09.cc, ...`~~
~~6~~	~~`add %x01, %x03 → %x11`~~
~~7~~	~~`imul %x04, %x06 → %x12`~~
~~8~~	~~`imul %x03, %x08 → %x13`~~
~~9~~	~~`cmp %x11, %x13 → %x14.cc`~~
~~10~~	~~`jle %x14.cc, ...`~~
...	...

execution unit	*cycle#* 1	2	3	4	5	6	7
ALU 1 (add, cmp, jxx)	1	6	–	4	5	9	10
ALU 2 (add, cmp, jxx)	–	–	–	–	–	–	–
ALU 3 (mul) start	2	3	7	8	–		
ALU 3 (mul) end		2	3	7	8		

reg	*status*
`%x01`	ready
`%x02`	ready
`%x03`	~~pending~~ ready
`%x04`	ready
`%x05`	ready
`%x06`	~~pending~~ ready
`%x07`	ready
`%x08`	~~pending~~ ready
`%x09`	~~pending~~ ready
`%x10`	pending
`%x11`	~~pending~~ ready
`%x12`	~~pending~~ ready
`%x13`	~~pending~~ ready
`%x14`	~~pending~~ ready
...	...

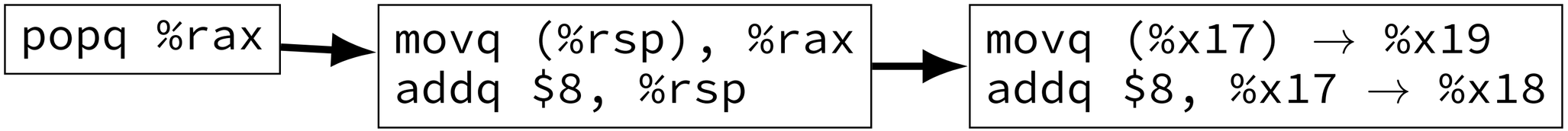
register renaming: missing pieces

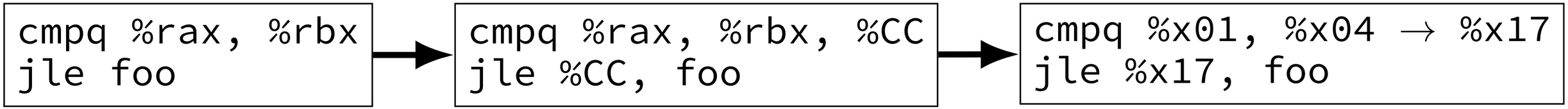
what about "hidden" inputs like `%rsp`, condition codes?

one solution: translate to intructions with additional register parameters

- making %rsp explicit parameter
- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones

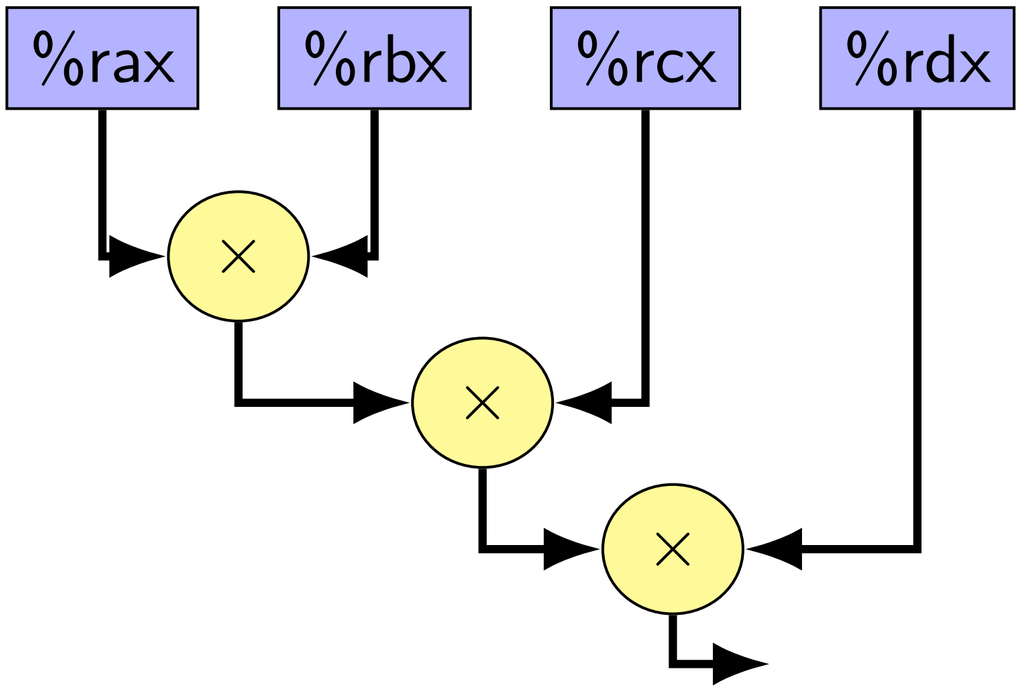




data flow visualization

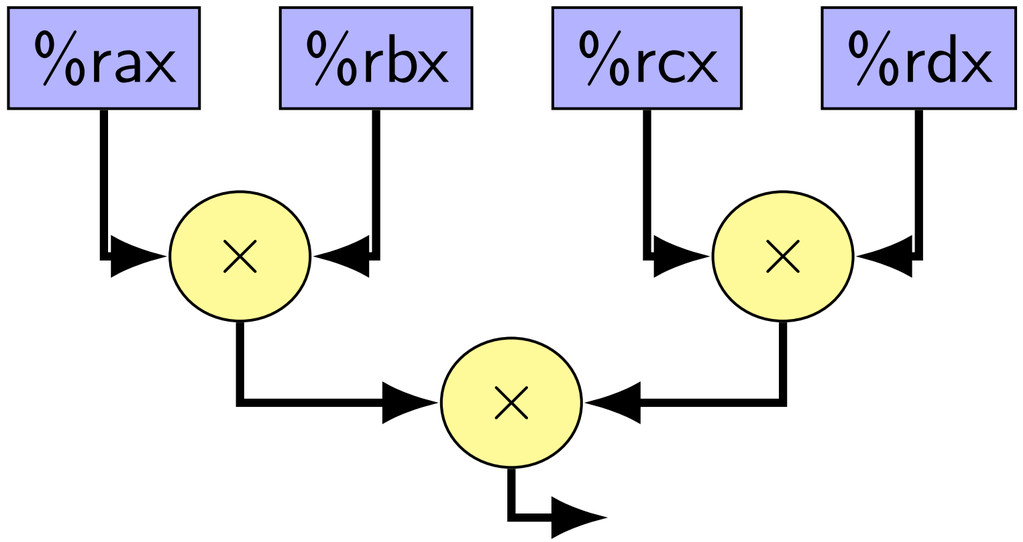
$$((a \times b) \times c) \times d$$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$$(a \times b) \times (c \times d)$$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



OOO limitations

can’t always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- programmer can adjust program to help this

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

- e.g. Intel Skylake: up to approx. 16 cycles (v. 2 for simple pipelined CPU)

OOO limitations

can't always find instructions to run

- plenty of instructions, but all depend on unfinished ones
- *programmer can adjust program to help this*

need to track all uncommitted instructions

- can only go so far ahead
- e.g. Intel Skylake: 224-entry reorder buffer, 168 physical registers

branch misprediction has a big cost (relative to pipelined)

- e.g. Intel Skylake: up to approx. 16 cycles (v. 2 for simple pipelined CPU)

some performance examples

some performance examples

```
example1:
    movq $10000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx 2.65 sec

```
example2:
    movq $10000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

my desktop: approx 2.65 sec

some performance examples

```
example1:
    movq $10000000000, %rax
loop1:
    addq %rbx, %rcx
    decq %rax
    jge loop1
    ret
```

about 30B instructions

my desktop: approx *2.65 sec*

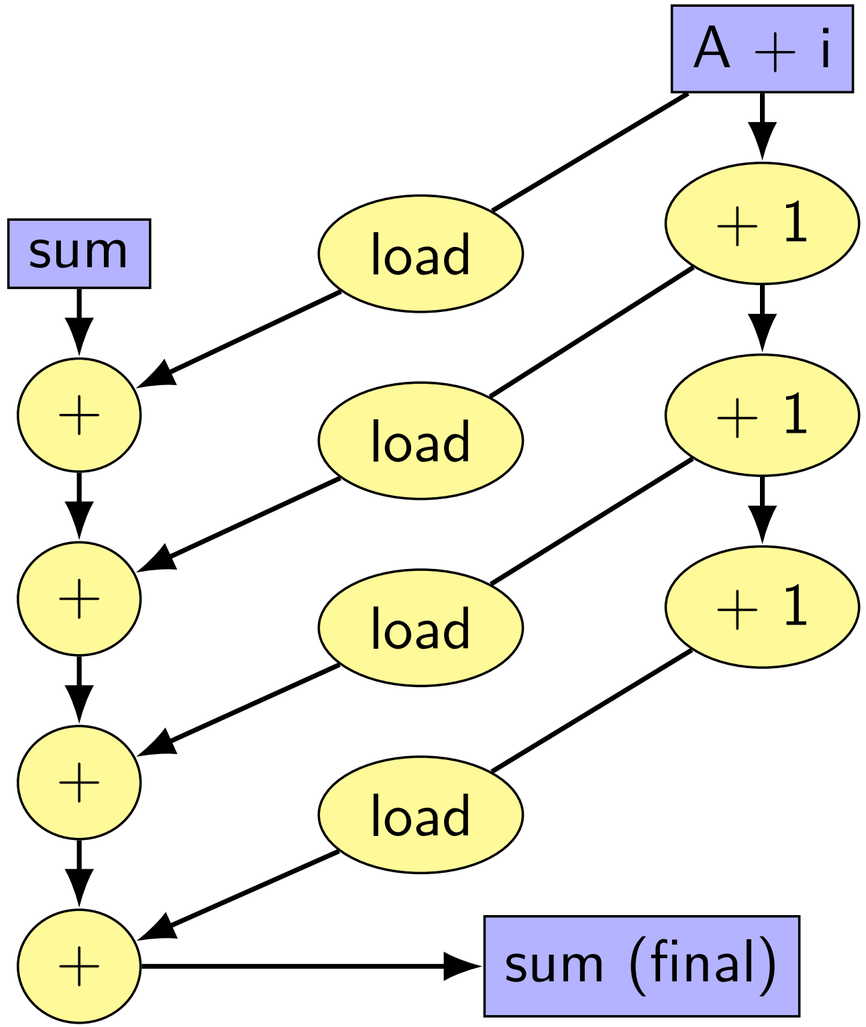
```
example2:
    movq $10000000000, %rax
loop2:
    addq %rbx, %rcx
    addq %r8, %r9
    decq %rax
    jge loop2
    ret
```

about 40B instructions

my desktop: approx *2.65 sec*

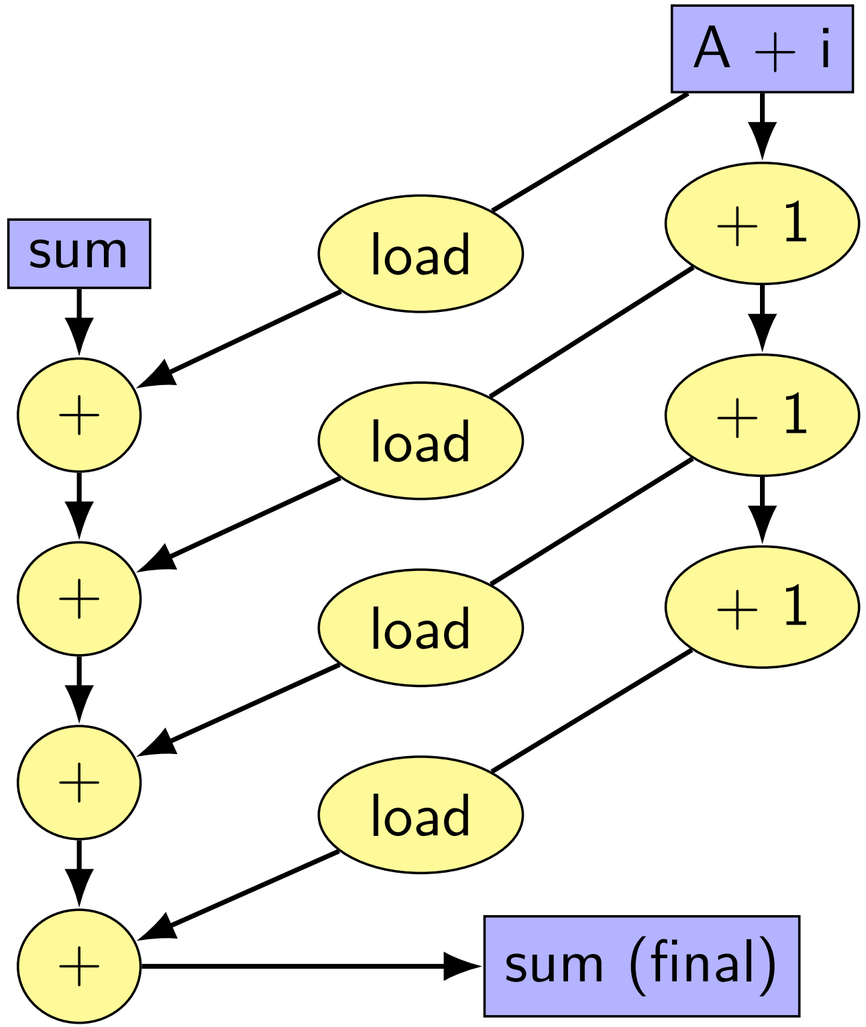
data flow model and limits

data flow model and limits



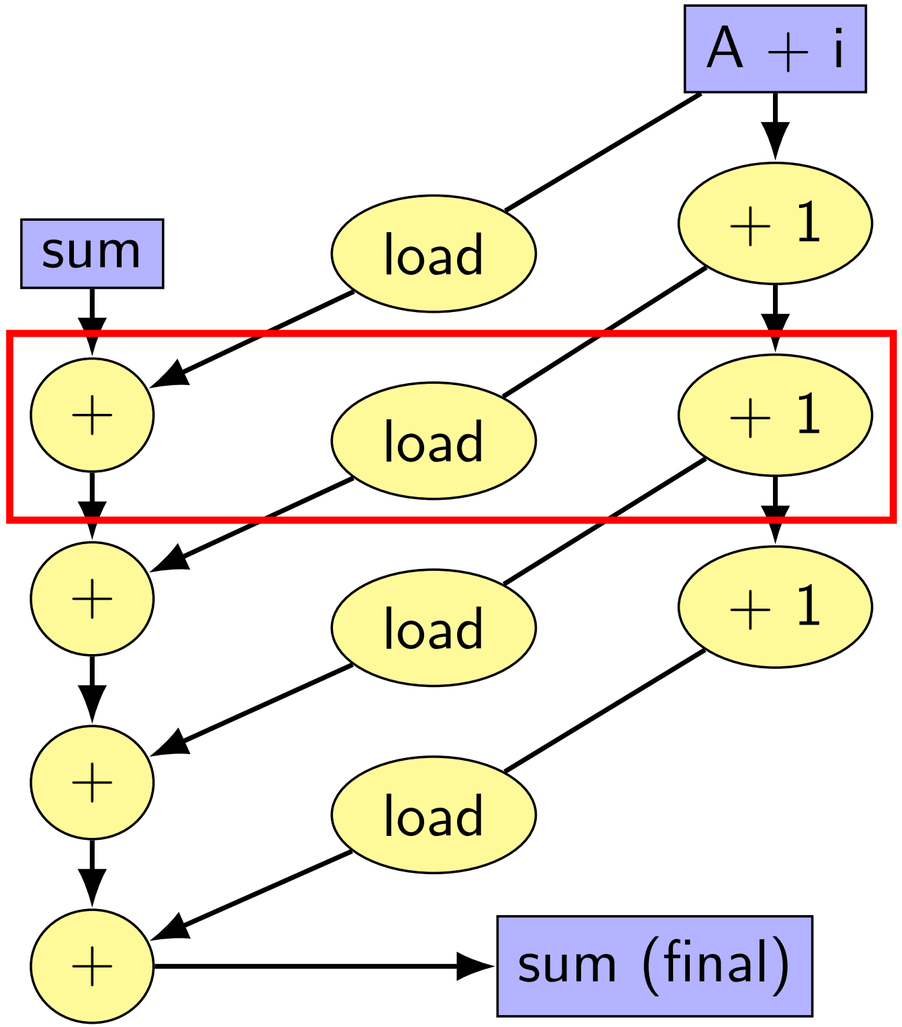

```
for (int i = 0; i < N; i += K) {
        sum += A[i];
        sum += A[i+1];
        …
}
```

data flow model and limits



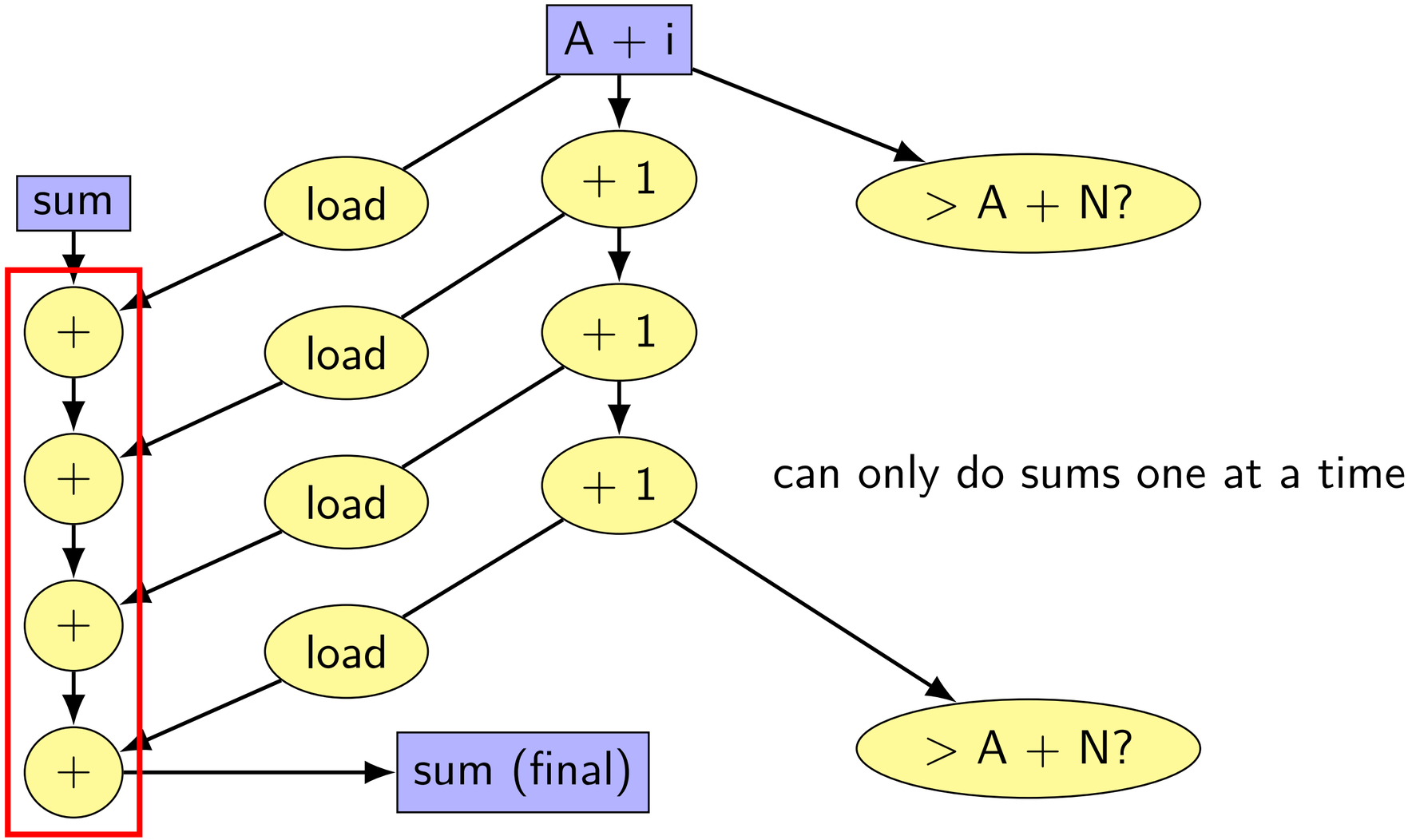
each yellow box = instruction
arrows = dependences
instructions only executed when dependencies ready

data flow model and limits



three ops/cycle (*if* each one cycle)

data flow model and limits



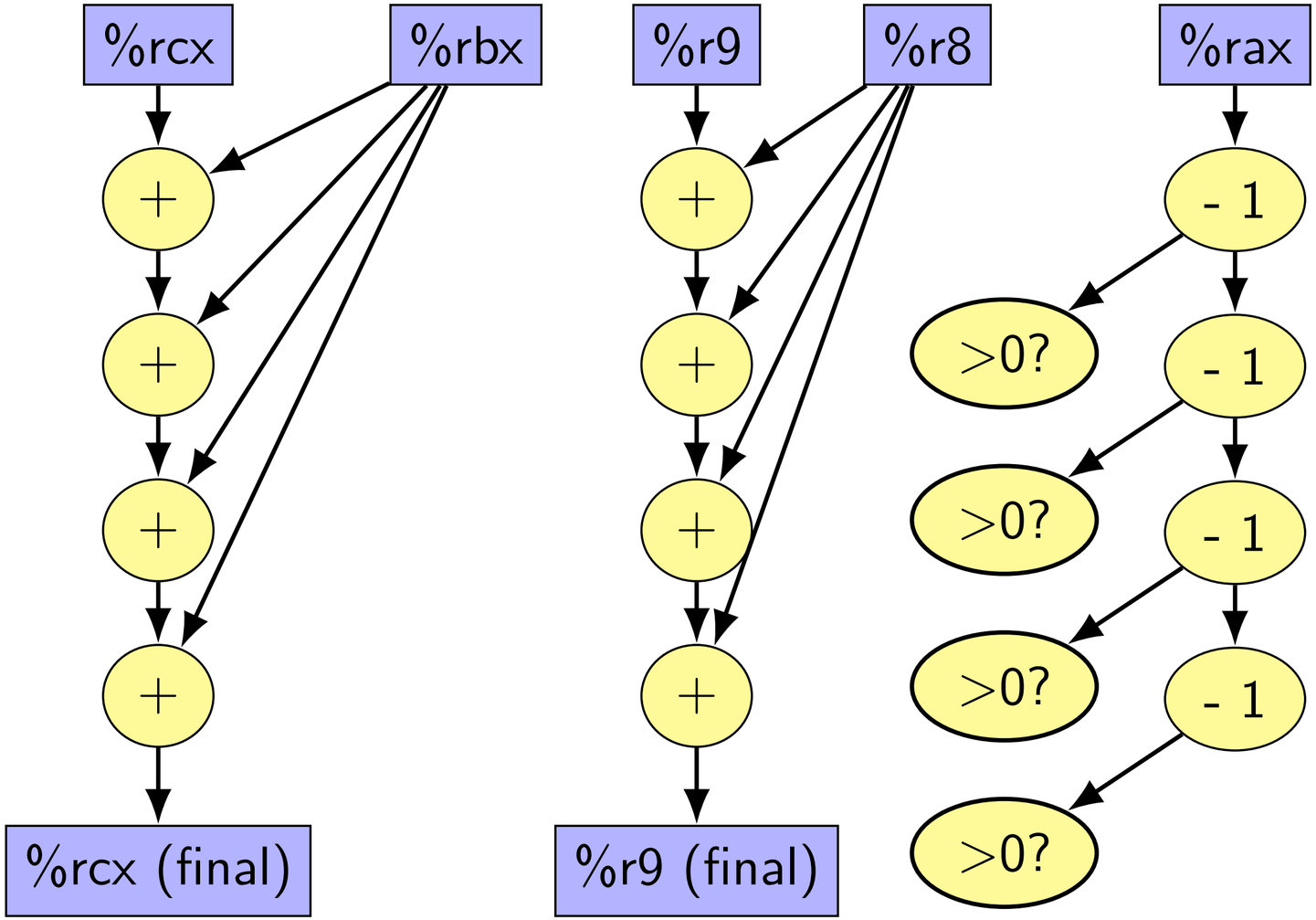
can only do sums one at a time

Backup slides

backup slides

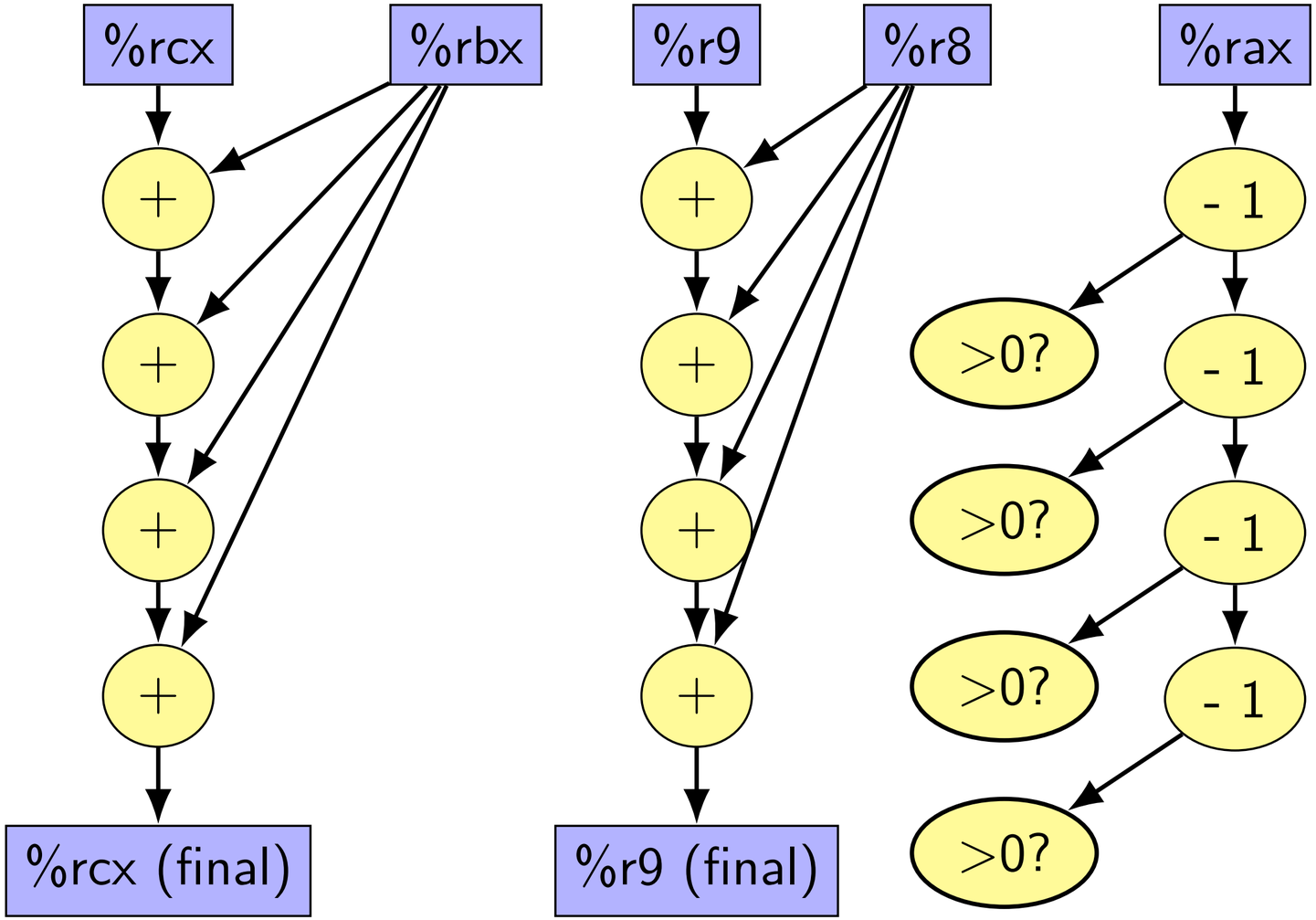
data flow model and limits (1)

data flow model and limits (1)




```
loop2:
        addq %rbx, %rcx
        addq %r8, %r9
        decq %rax
        jge loop2
```

data flow model and limits (1)



each yellow box = instruction

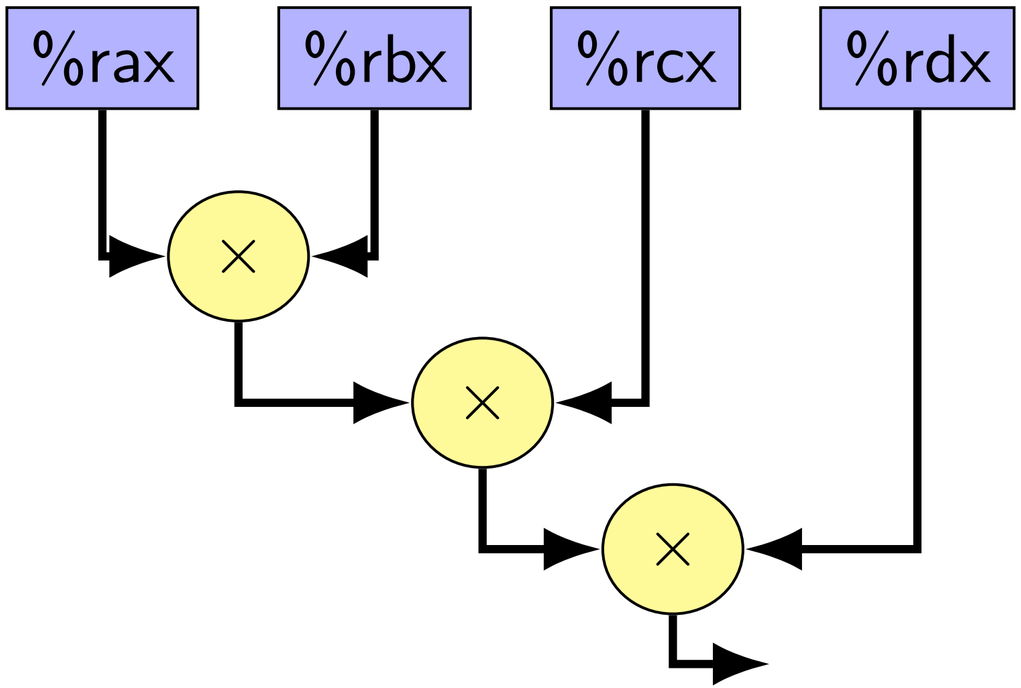
arrows = dependences

instructions only executed when dependencies ready

data flow visualization

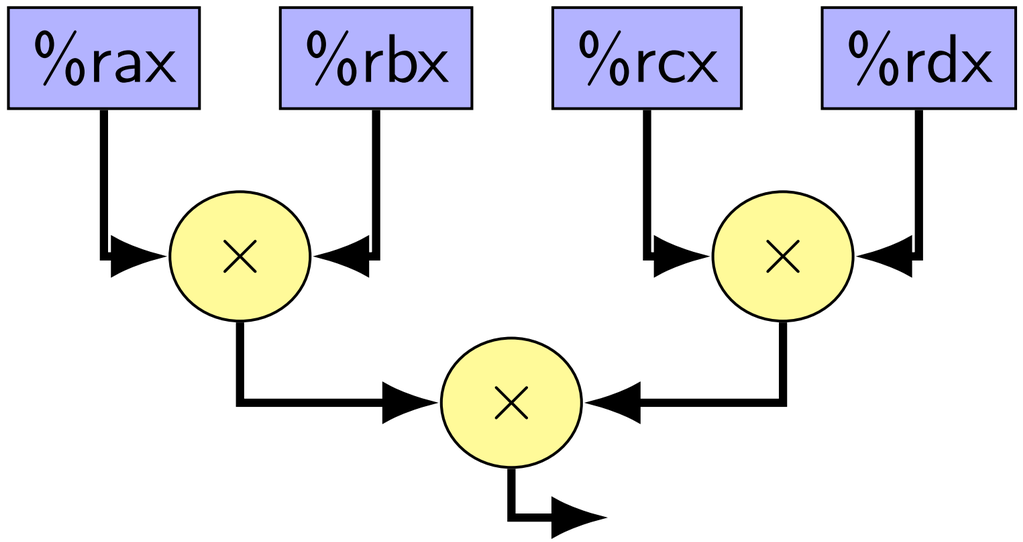
$$((a \times b) \times c) \times d$$

```
imulq %rbx, %rax
imulq %rcx, %rax
imulq %rdx, %rax
```



$$(a \times b) \times (c \times d)$$

```
imulq %rbx, %rax
imulq %rcx, %rdx
imulq %rdx, %rax
```



Intel Skylake OOO design

2015 Intel design — codename ‘Skylake’

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

- but some can handle more/different types of operations than others

2 load functional units

- but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

224-entry reorder buffer

- determines how far ahead branch mispredictions, etc. can happen

indirect branch prediction

`jmp *%rax` or `jmp *(%rax, %rcx, 8)` or `call *%rax` or ...

- example: implementing switch statement
- example: implementing polymorphic method call

want to predict *target address*

BTB could store one possibility

really want to take advantage of context

- example: guess whether we'll call Rectangle.GetArea or Circle.GetArea

prediction idea: instead of tables containing taken/not taken...

... have table containing predicted target

- one implementation: hashtable keyed by recent branches taken

reorder buffer: on rename

reorder buffer: on rename

arch → phys reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer: on rename

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
...	...

free list

%x19
%x23
...
...

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
...	...	...	...	...
31	0x129f	%rax / %x12		

reorder buffer contains instructions started,
but not fully finished new entries created on rename
(not enough space? stall rename stage)

reorder buffer: on rename

arch → phys reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x07
…	…

free list

%x19
%x23
…
…

reorder buffer (ROB)

remove here on commit →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

place newly started instruction at end of buffer
remember at least its destination register
(both architectural and physical versions)

reorder buffer: on rename

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ *%x19*
…	…

free list

~~%x19~~
%x23
…
…

reorder buffer (ROB)

remove here on commit →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

next renamed instruction goes in next slot, etc.

reorder buffer: on rename

arch → phys reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x23
…
…

reorder buffer (ROB)

remove here on commit →

add here on rename →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x23		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		
32	0x1230	%rdx / %x19		

reorder buffer: on commit

reorder buffer: on commit

arch → phys. reg
for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

remove here on commit →

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31		
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34		
19	0x1249	%rax / %x38		
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		

reorder buffer: on commit

arch → phys. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

remove here on commit →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

instructions marked done in reorder buffer when computed but not removed ('committed') yet

reorder buffer: on commit

arch → phys. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
…

arch → phys reg for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	%x23
%rdx	%x21
…	…

remove here on commit →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24		
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓

commit stage tracks architectural to physical register map for committed instructions

reorder buffer: on commit

arch → phys. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

arch → phys reg for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ *%x24*
%rdx	%x21
…	…

when next-to-commit instruction is done update this register map and free register list and remove instr. from reorder buffer

remove here on commit →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
14	0x1233	%rbx / %x24	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

reorder buffer: on commit

arch → phys. reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	~~%x07~~ %x19
…	…

free list

~~%x19~~
%x13
…
%x23

arch → phys reg for committed

arch. reg	phys. reg
%rax	%x30
%rcx	%x28
%rbx	~~%x23~~ %x24
%rdx	%x21
…	…

remove here when committed →

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
15	0x1239	%rax / %x30		
16	0x1242	%rcx / %x31	✓	
17	0x1244	%rcx / %x32		
18	0x1248	%rdx / %x34	✓	
19	0x1249	%rax / %x38	✓	
20	0x1254	PC		
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12		✓
32	0x1230	%rdx / %x19		

when next-to-commit instruction is done update this register map and free register list and remove instr. from reorder buffer

reorder buffer: commit mispredict (one way)

reorder buffer: commit mispredict (one way)

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
…	…

arch → phys reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

reorder buffer: commit mispredict (one way)

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x12
%rcx	%x17
%rbx	%x13
%rdx	%x19
…	…

arch → phys reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

when committing a mispredicted instruction…
this is where we undo mispredicted instructions

reorder buffer: commit mispredict (one way)

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

arch → phys reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	~~✓~~	
~~15~~	~~0x1239~~	~~%rax / %x30~~	~~✓~~	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	~~✓~~	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	~~✓~~	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	~~✓~~	
~~19~~	~~0x1249~~	~~%rax / %x38~~	~~✓~~	
20	0x1254	PC	✓	✓
21	0x1260	%rcx / %x17		
…	…	…	…	…
31	0x129f	%rax / %x12	✓	
32	0x1230	%rdx / %x19		

copy commit register map into rename register map so we can start fetching from the correct PC

reorder buffer: commit mispredict (one way)

arch → phys reg for new instrs

arch. reg	phys. reg
%rax	%x38
%rcx	%x32
%rbx	%x24
%rdx	%x34
…	…

arch → phys reg for committed

arch. reg	phys. reg
%rax	~~%x30~~ %x38
%rcx	~~%x31~~ %x32
%rbx	~~%x23~~ %x24
%rdx	~~%x21~~ %x34
…	…

free list

~~%x19~~
%x13
…
…

…and discard all the mispredicted instructions (without committing them)

reorder buffer (ROB)

instr num.	PC	dest. reg	done?	mispred? / except?
~~14~~	~~0x1233~~	~~%rbx / %x24~~	✓	
~~15~~	~~0x1239~~	~~%rax / %x30~~	✓	
~~16~~	~~0x1242~~	~~%rcx / %x31~~	✓	
~~17~~	~~0x1244~~	~~%rcx / %x32~~	✓	
~~18~~	~~0x1248~~	~~%rdx / %x34~~	✓	
~~19~~	~~0x1249~~	~~%rax / %x38~~	✓	
20	0x1254	PC	✓	✓
~~21~~	~~0x1260~~	~~%rcx / %x17~~		
…	…	…	…	…
~~31~~	~~0x129f~~	~~%rax / %x12~~	✓	
~~32~~	~~0x1230~~	~~%rdx / %x19~~		

better? alternatives

can take snapshots of register map on each branch

- don’t need to reconstruct the table
- (but how to efficiently store them)

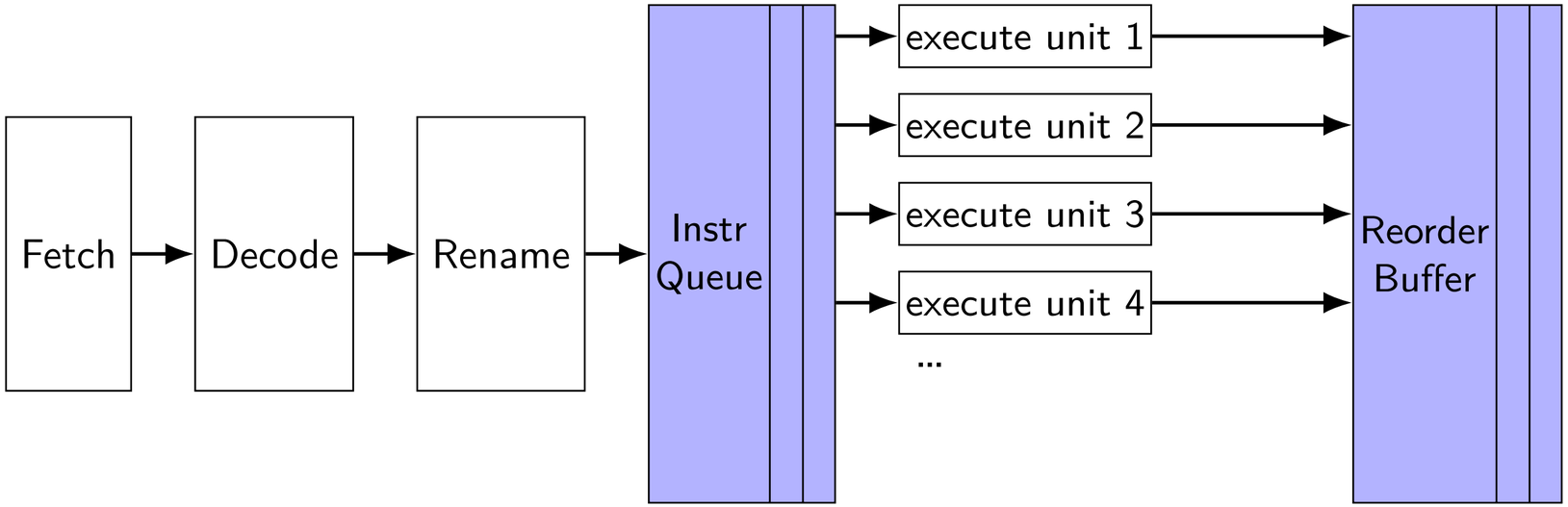
can reconstruct register map before we commit the branch instruction

- need to let reorder buffer be accessed even more?

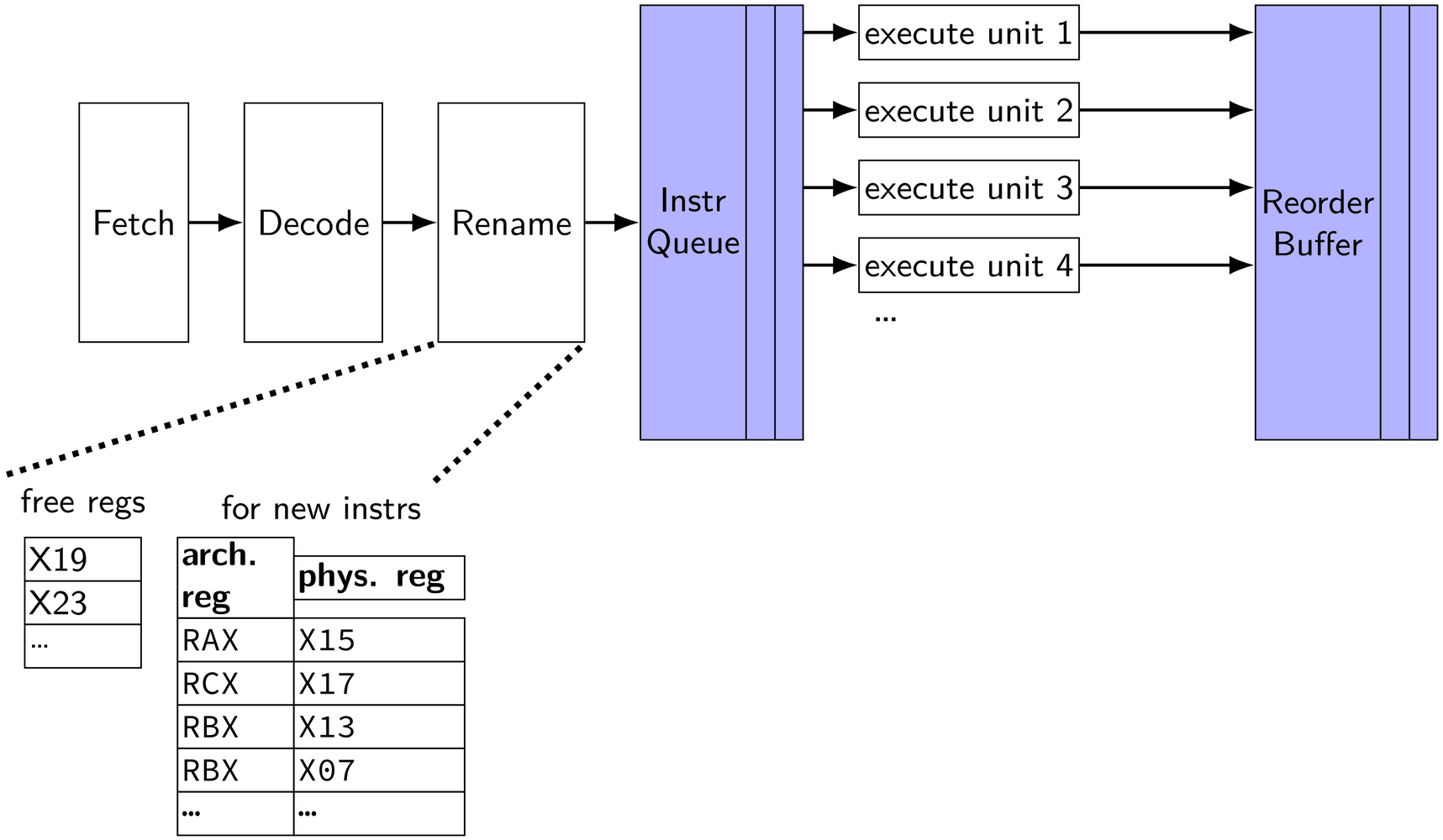
can track more/different information in reorder buffer

exceptions and OOO (one strategy)

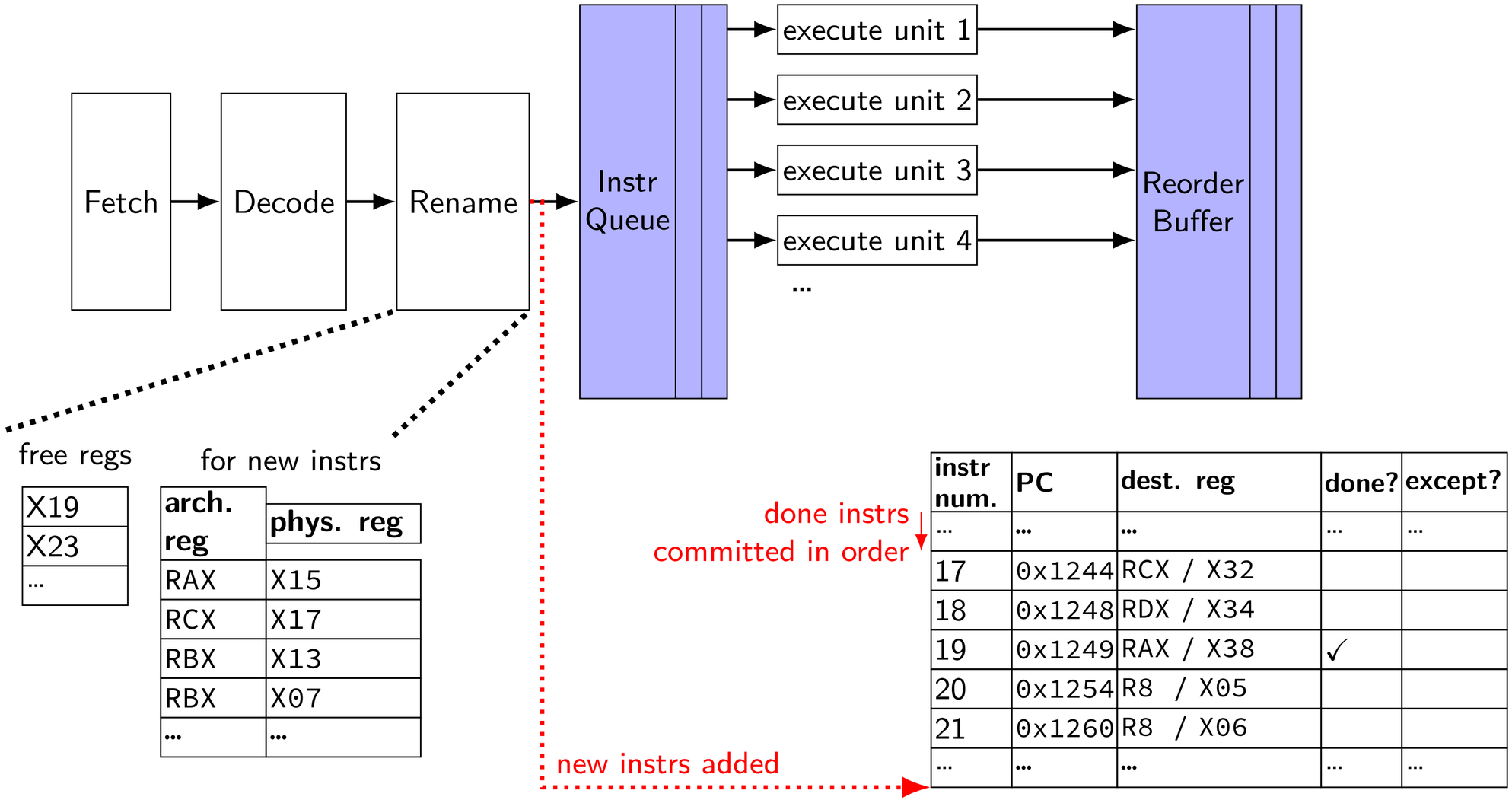
exceptions and OOO (one strategy)



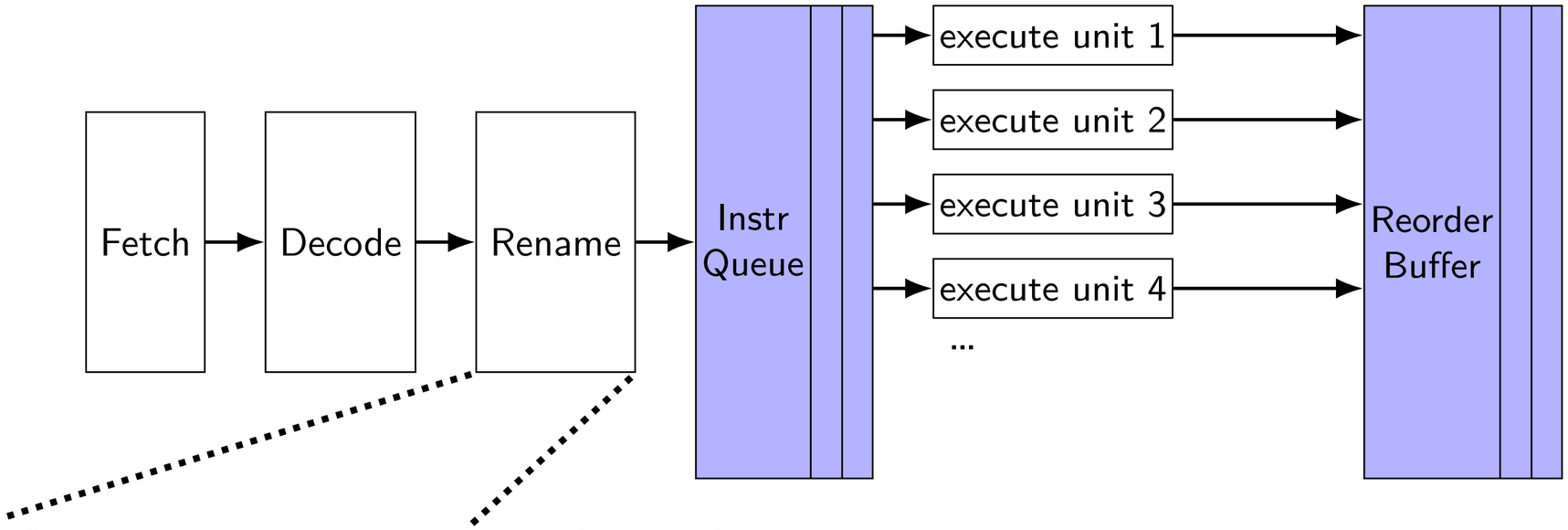
exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

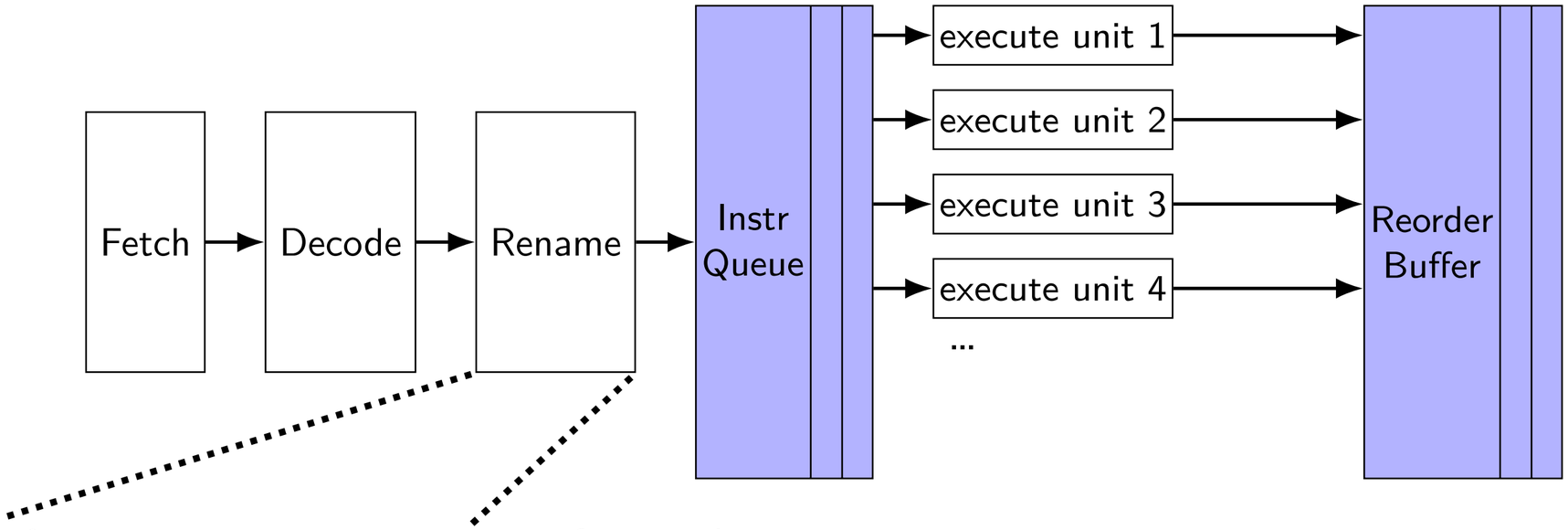
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ *X32*
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

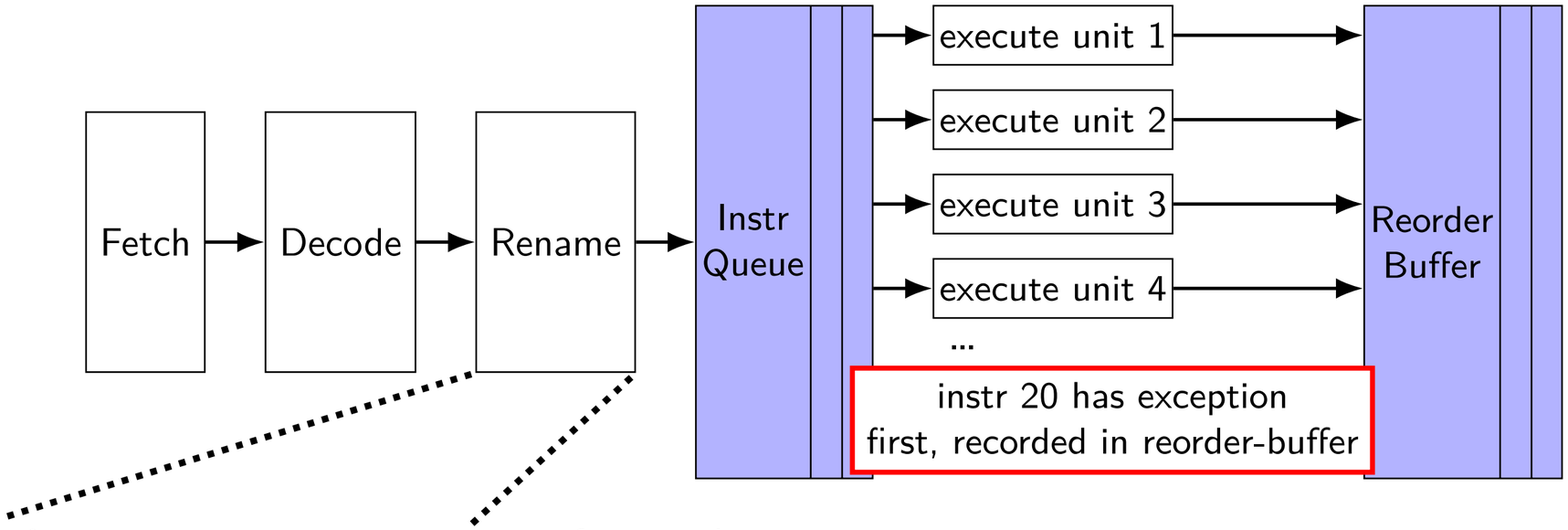
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

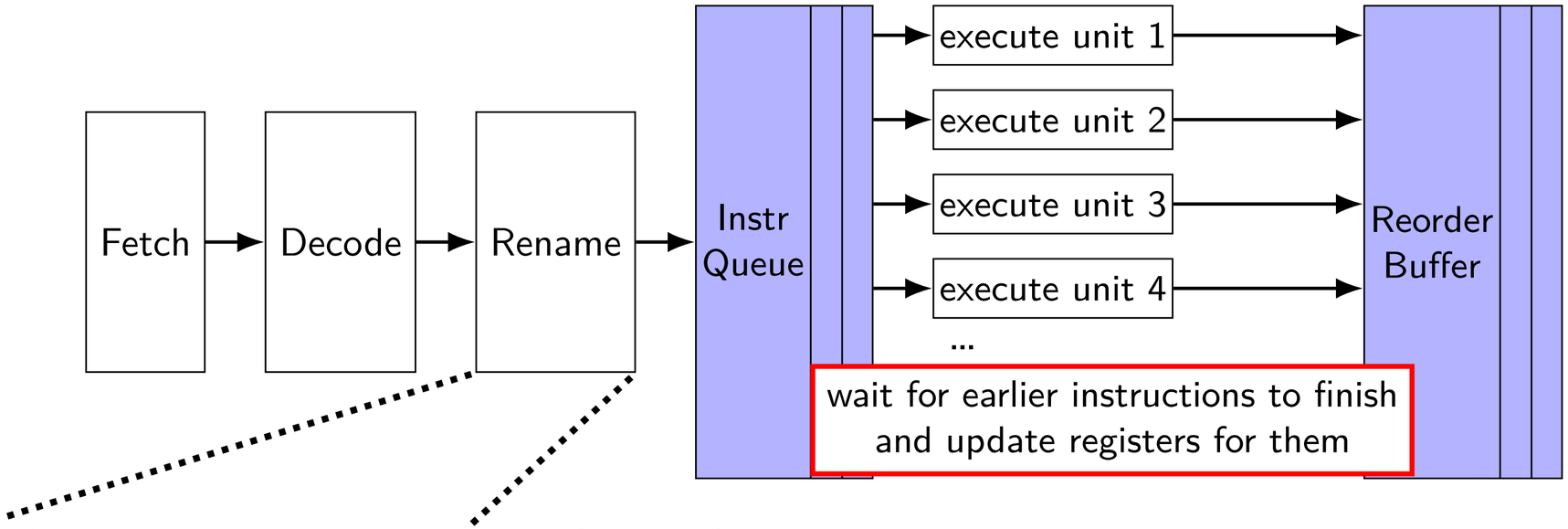
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

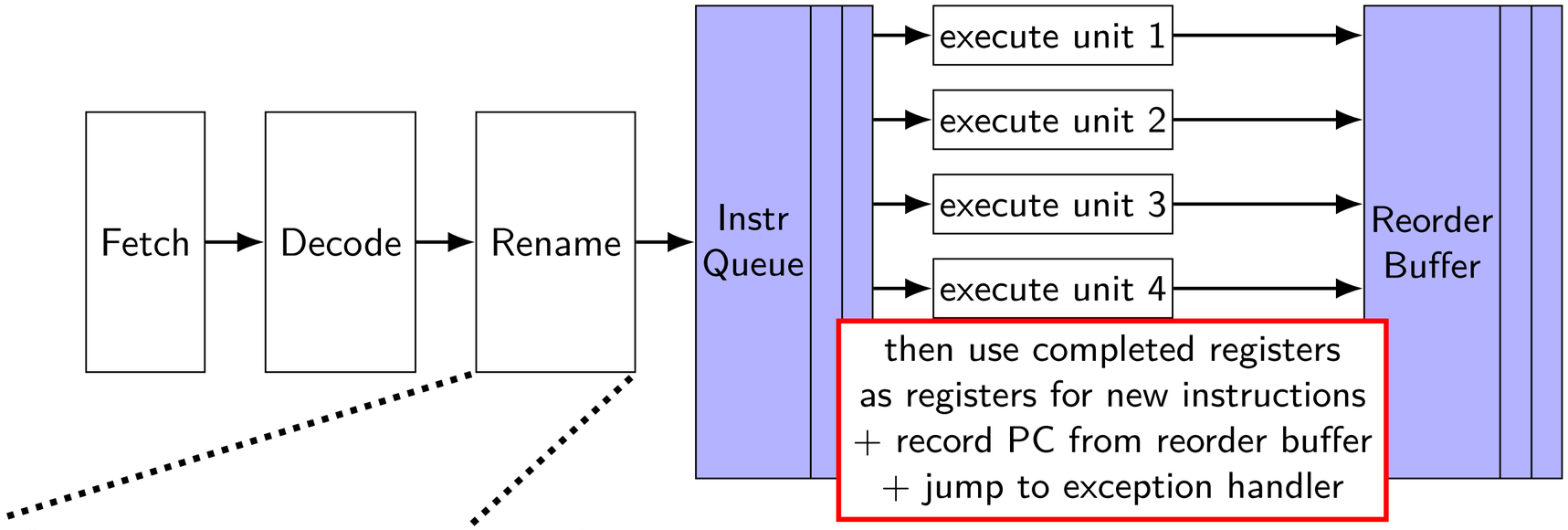
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ *X38*
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ *X34*
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
~~18~~	~~0x1248~~	~~RDX / X34~~	✓	
~~19~~	~~0x1249~~	~~RAX / X38~~	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

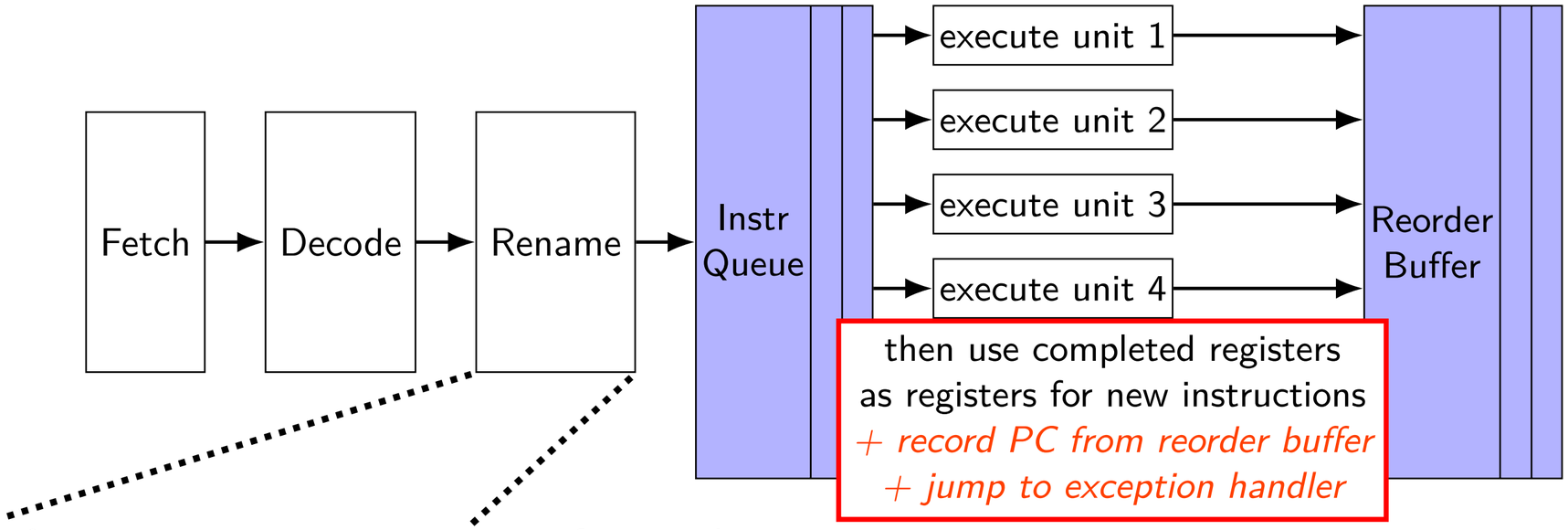
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

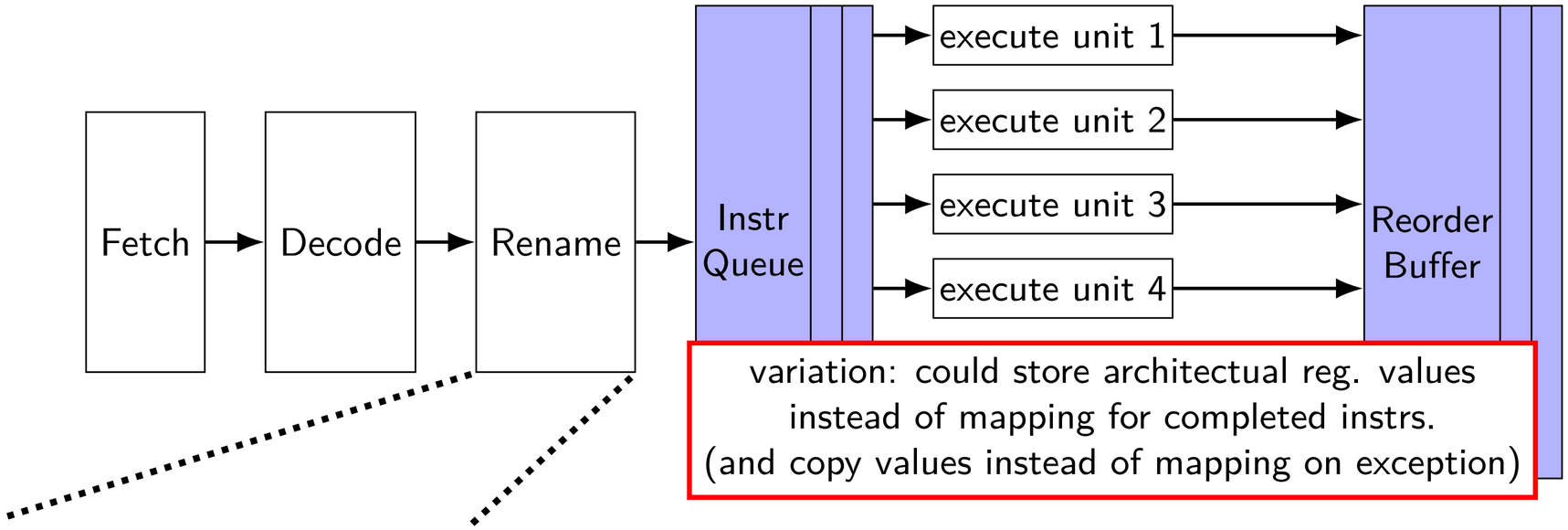
arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
…	…

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
…

for new instrs

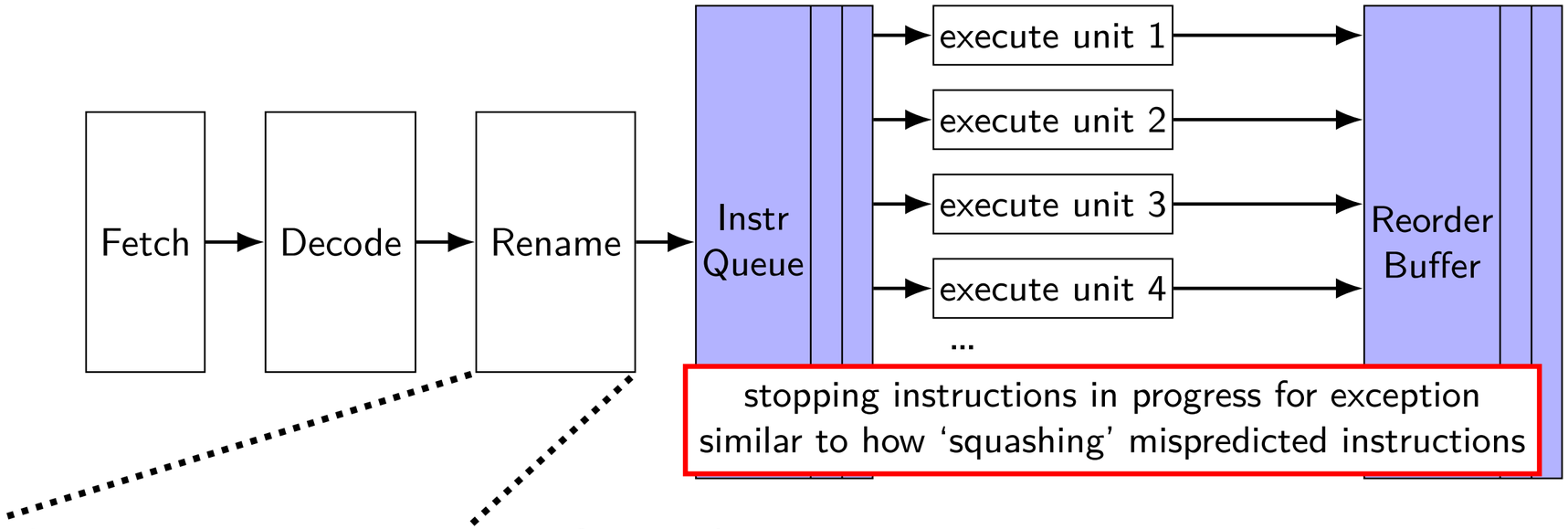
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
…	…

for complete instrs

arch. reg	value
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
…	…

instr num.	PC	dest. reg	done?	except?
…	…	…	…	…
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
…	…	…	…	…

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

the open-source BROOM pipeline

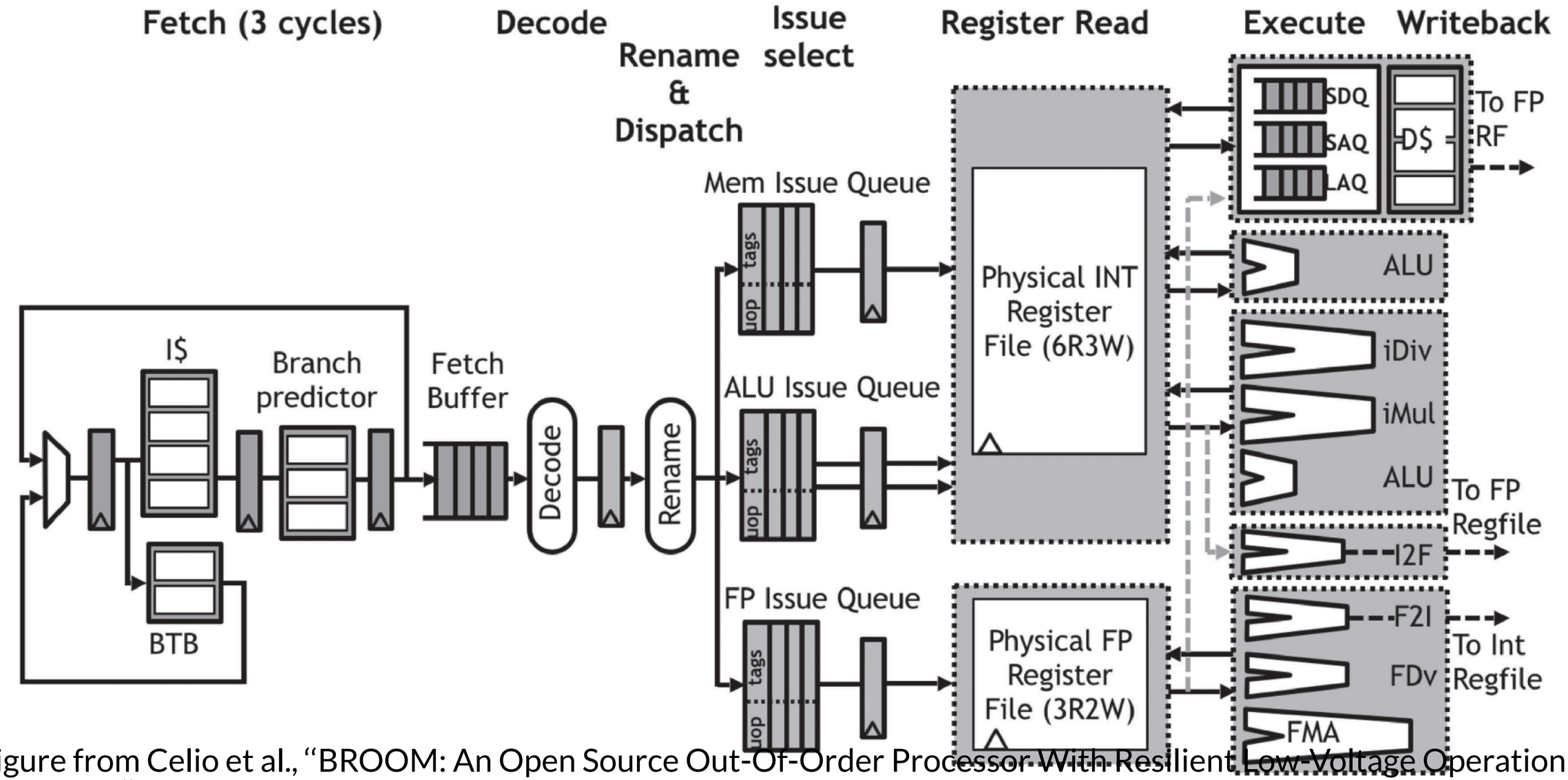
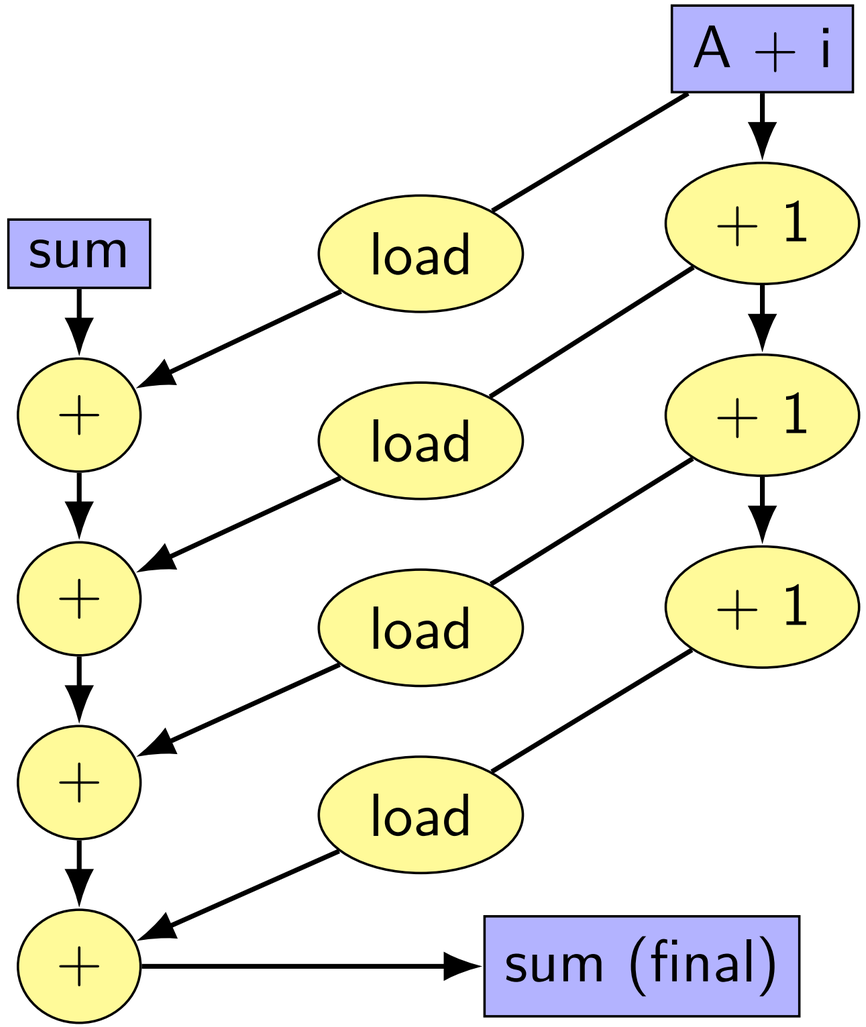


Figure from Celio et al., "BROOM: An Open Source Out-Of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS"

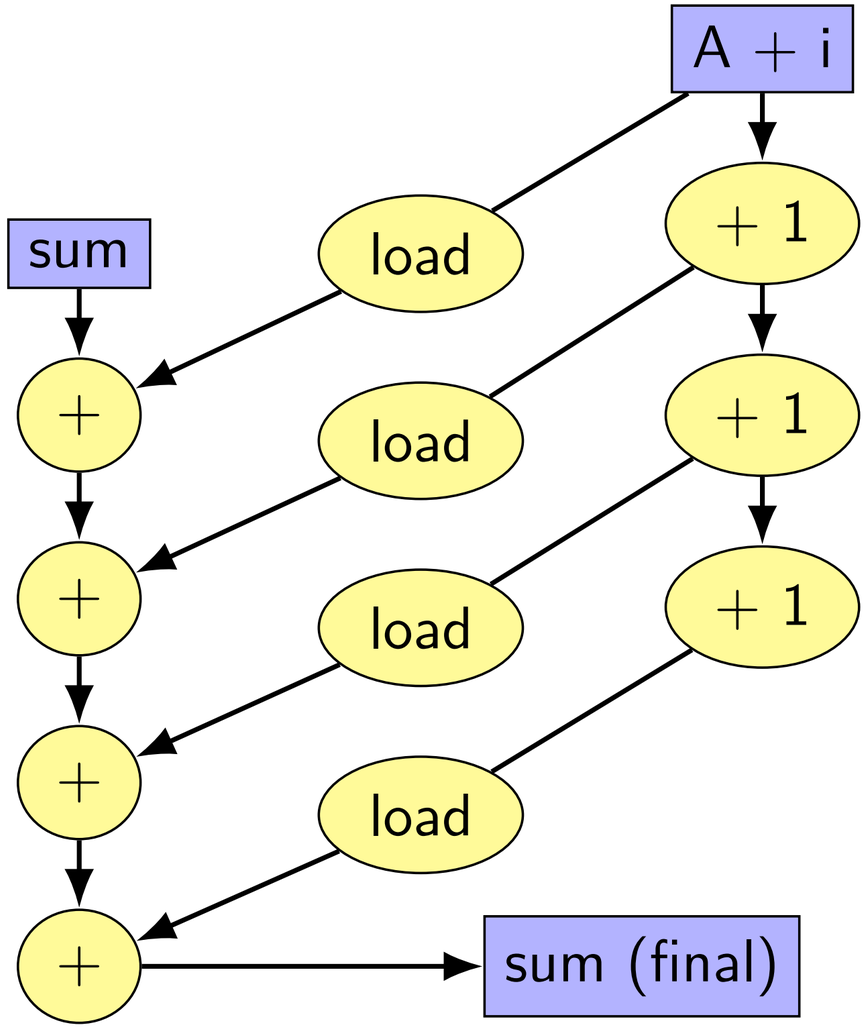
data flow model and limits

data flow model and limits



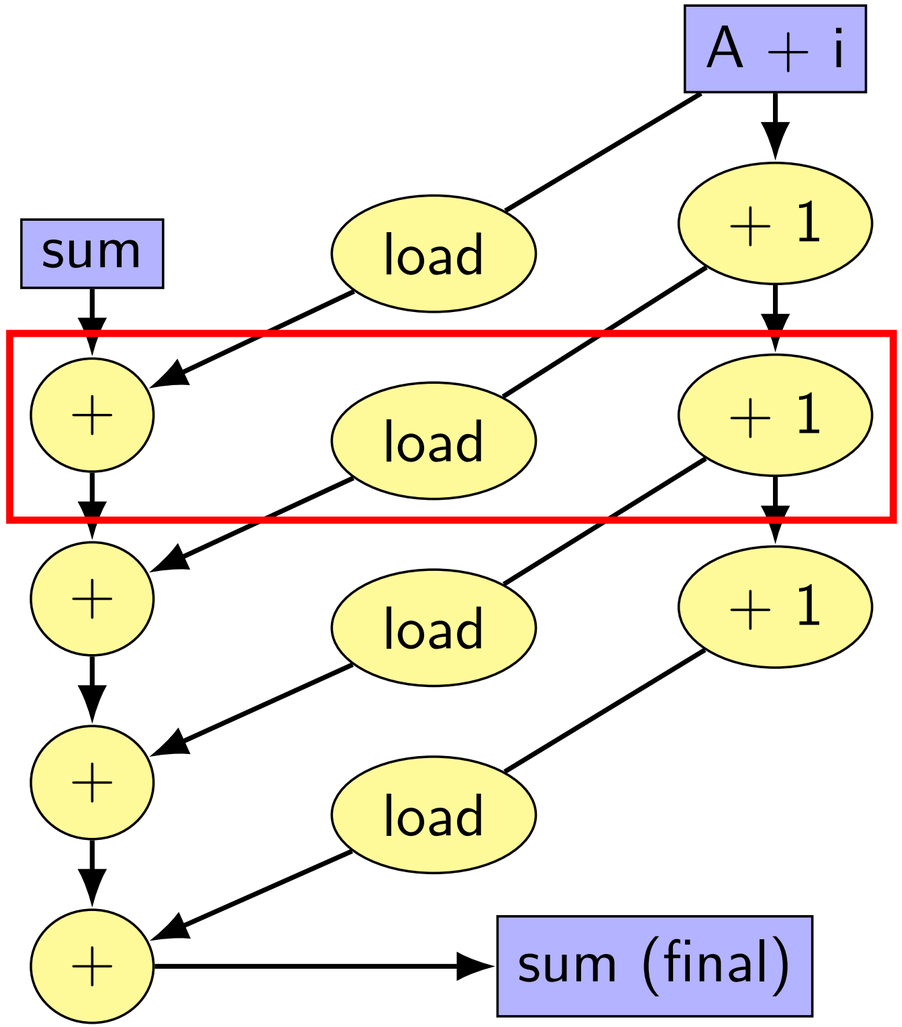

```
for (int i = 0; i < N; i += K) {
        sum += A[i];
        sum += A[i+1];
        …
}
```

data flow model and limits



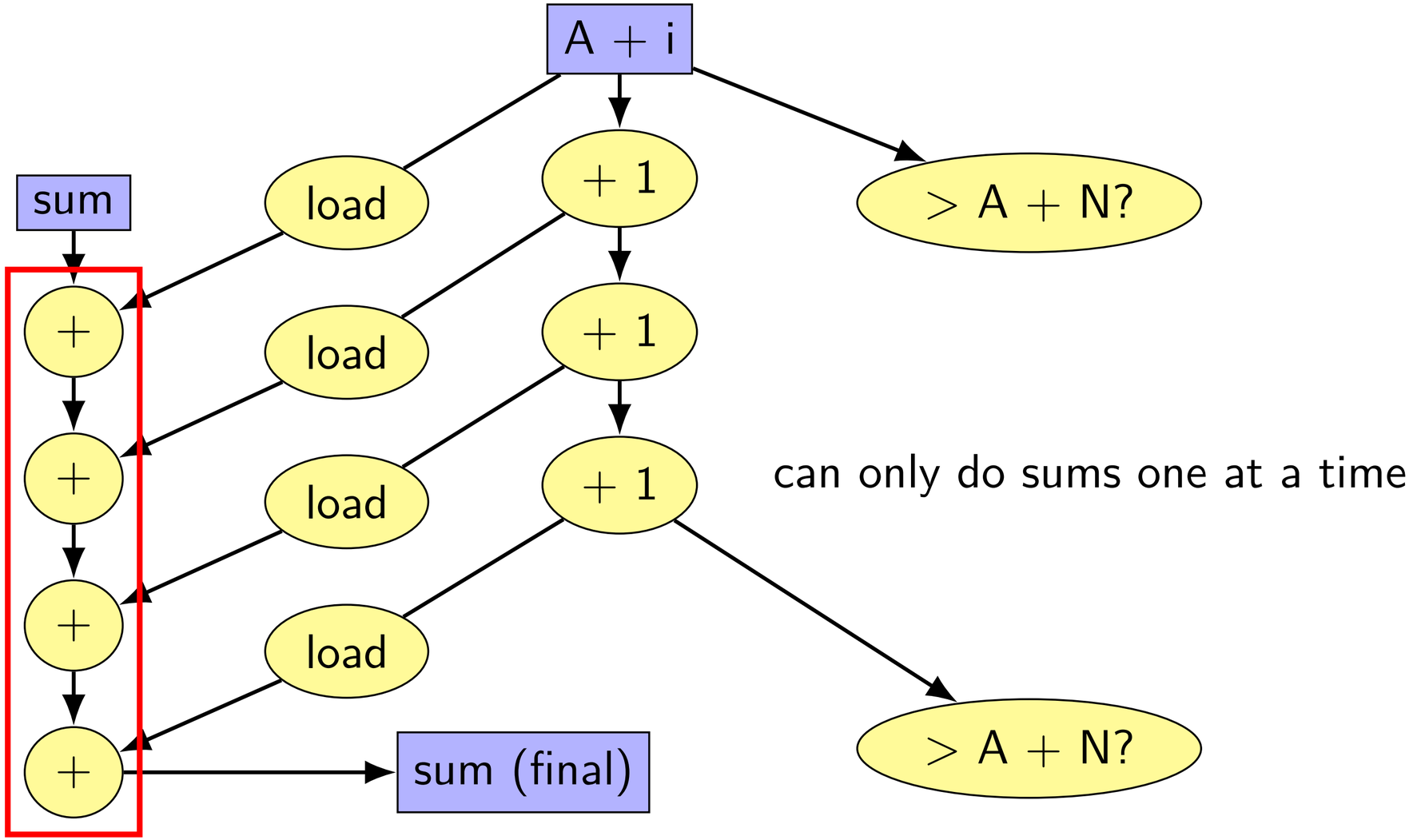
each yellow box = instruction
arrows = dependences
instructions only executed when dependencies ready

data flow model and limits



three ops/cycle (*if* each one cycle)

data flow model and limits



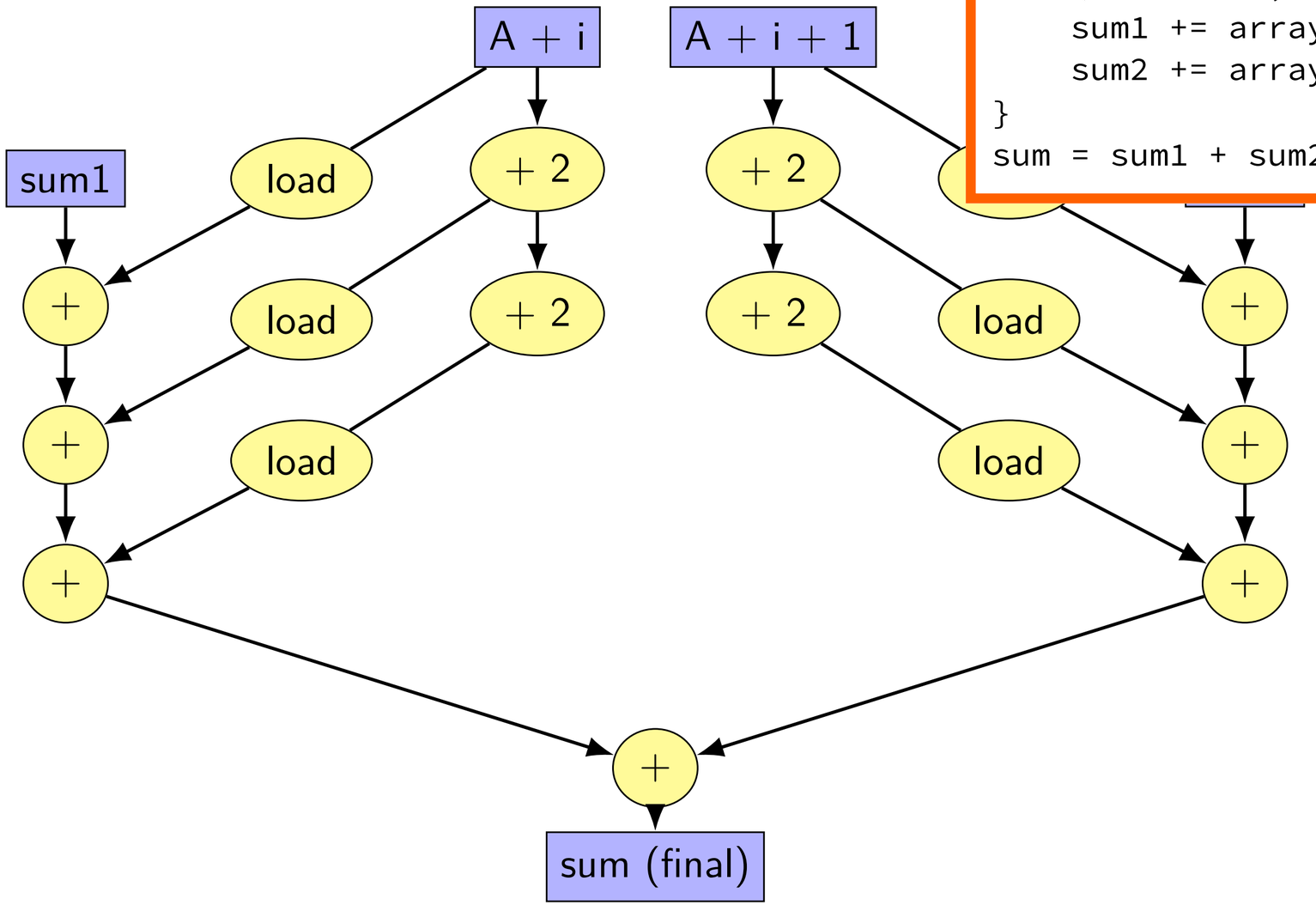
can only do sums one at a time

better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```

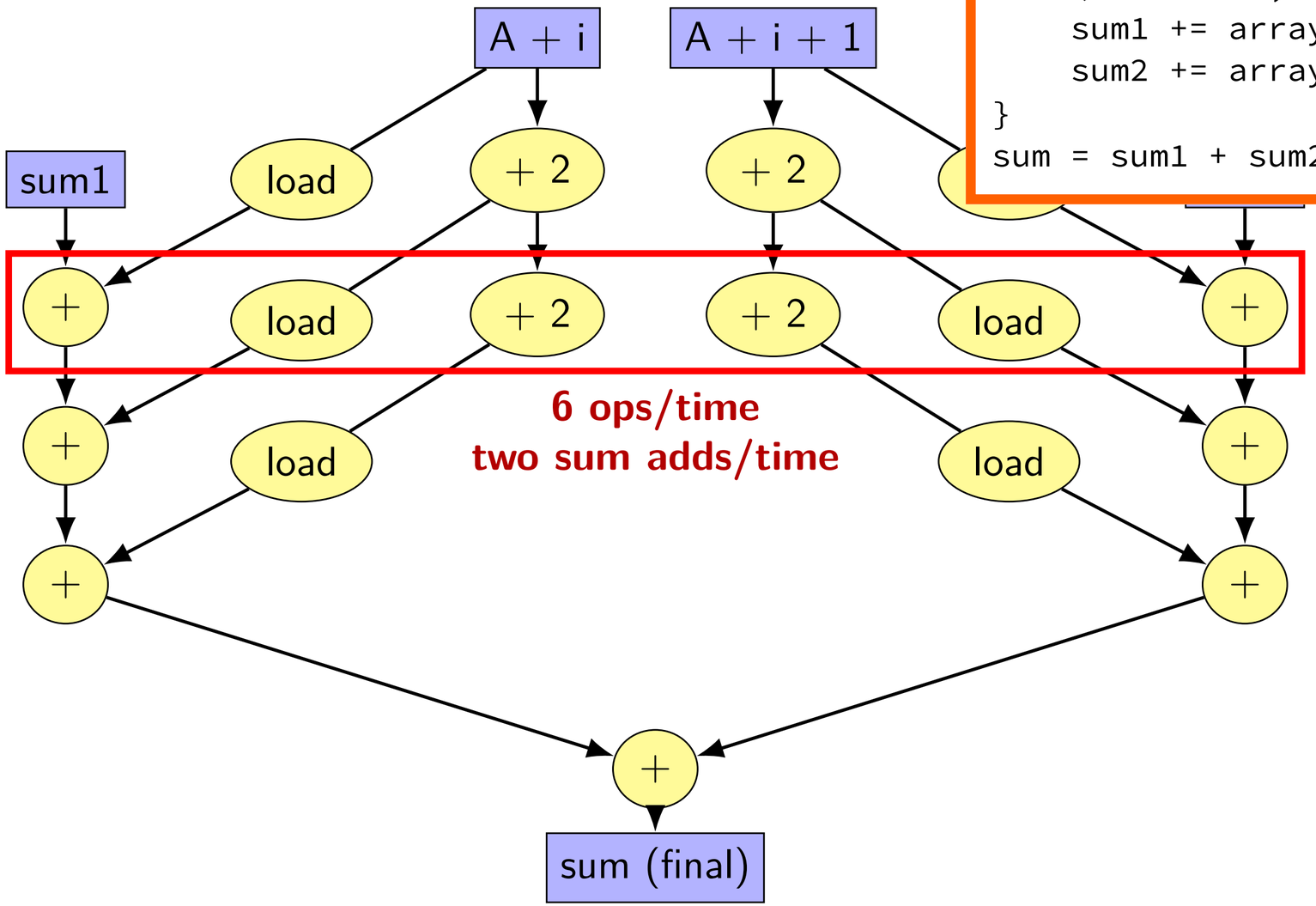
better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```



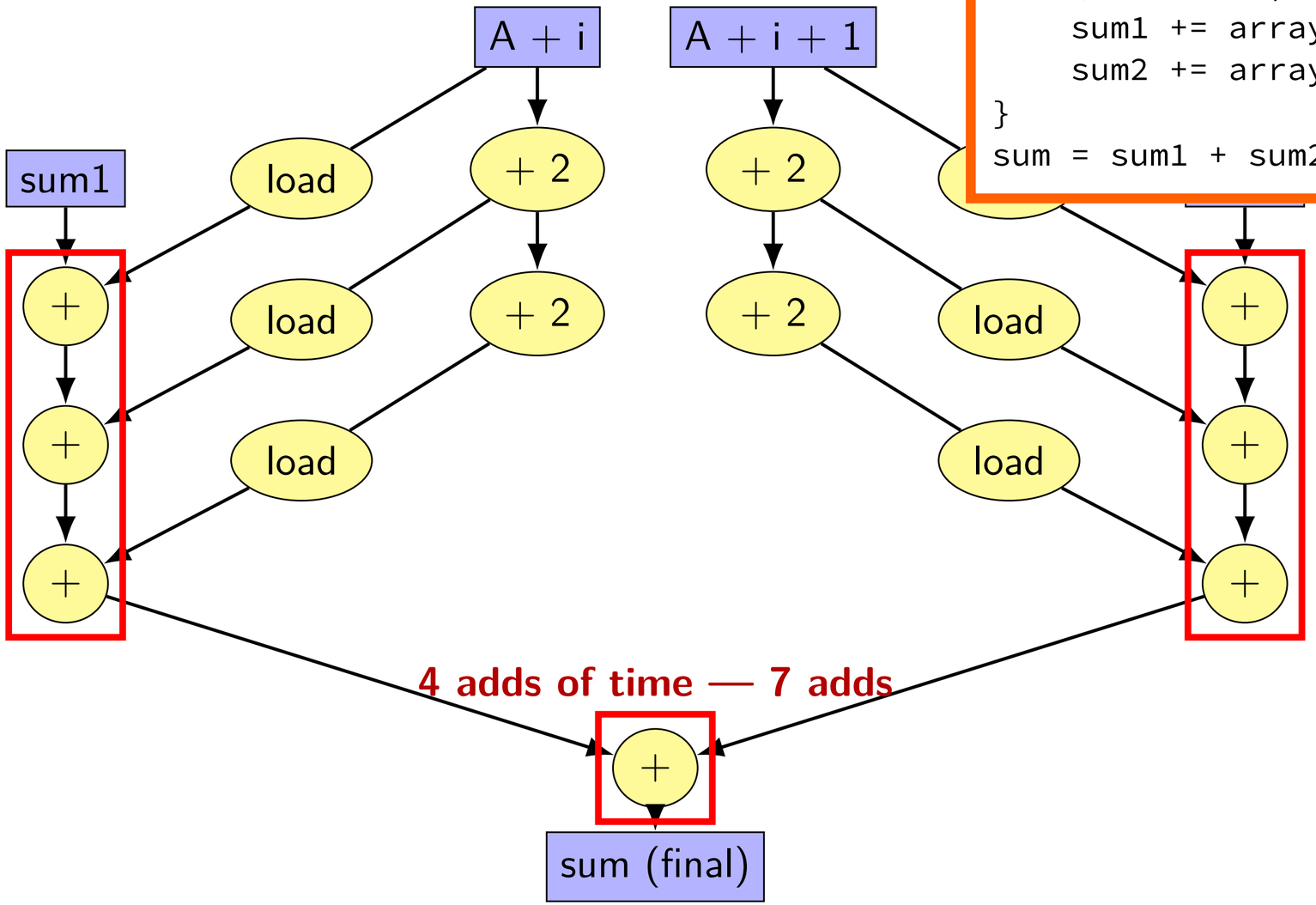
better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```



better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```



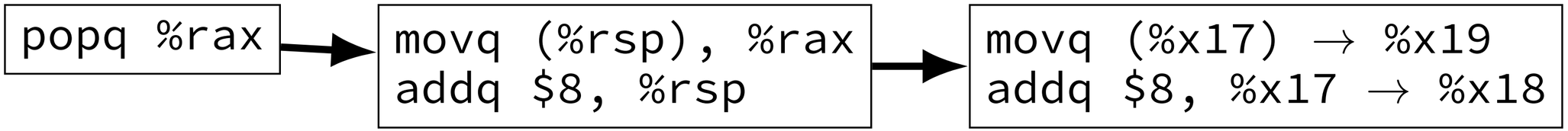
register renaming: missing pieces

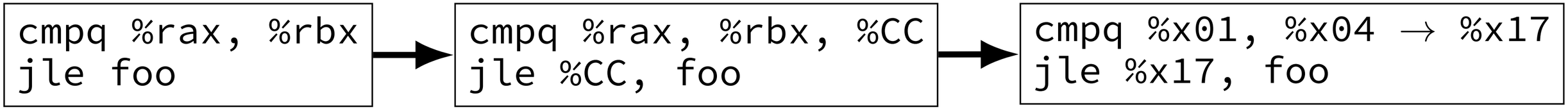
what about "hidden" inputs like `%rsp`, condition codes?

one solution: translate to intructions with additional register parameters

- making %rsp explicit parameter
- turning hidden condition codes into operands!

bonus: can also translate complex instructions to simpler ones

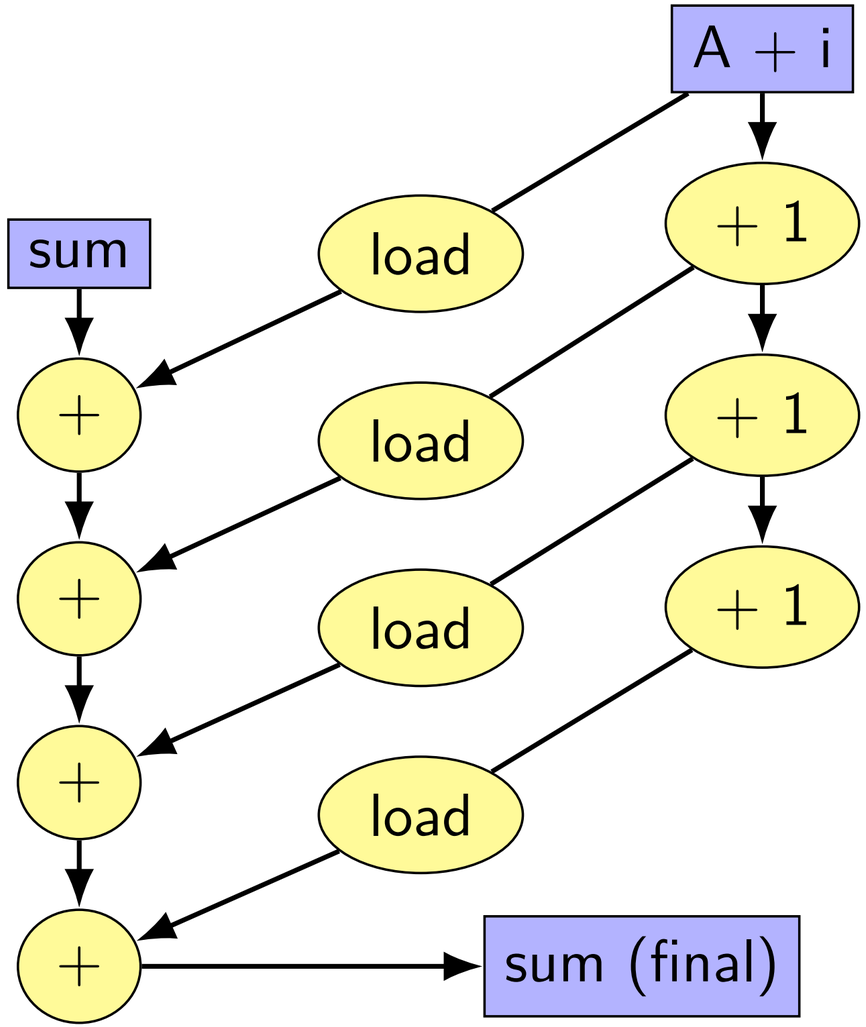




data flow as loop optimization

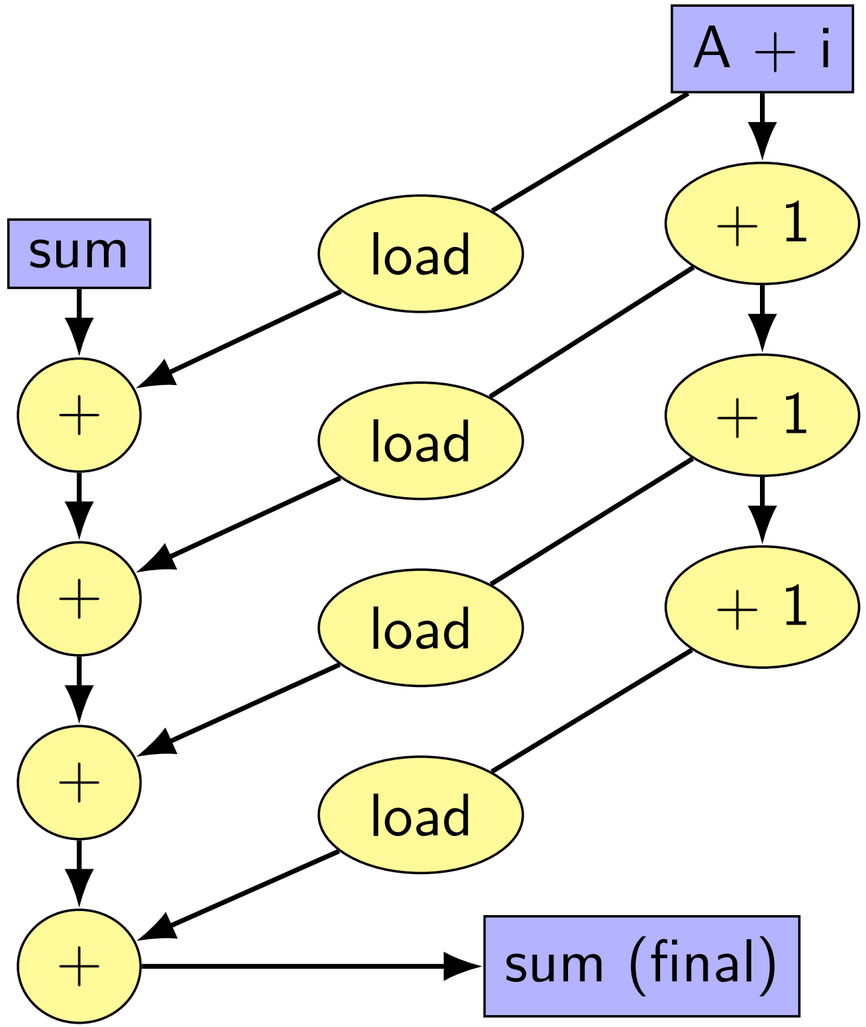
data flow model and limits

data flow model and limits



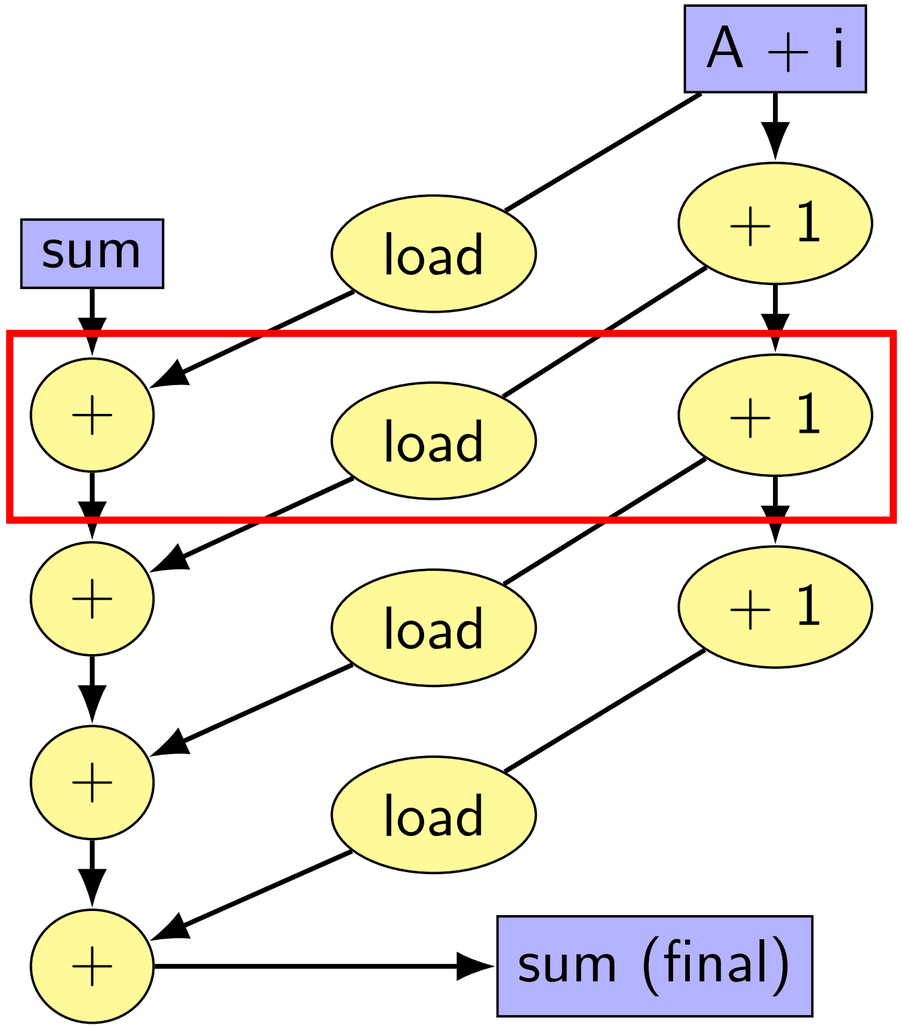

```
for (int i = 0; i < N; i += K) {
        sum += A[i];
        sum += A[i+1];
        …
}
```

data flow model and limits



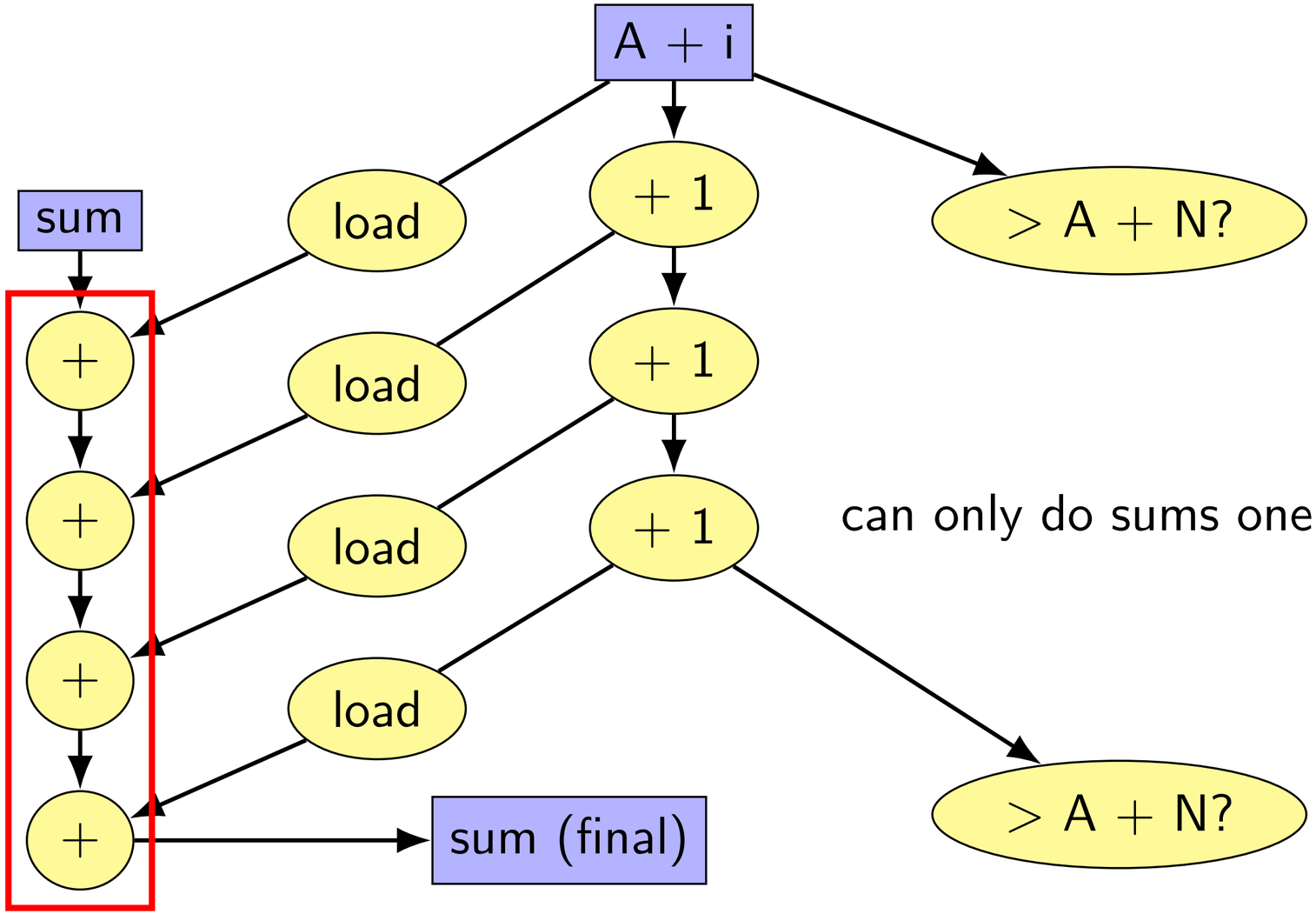
each yellow box = instruction
arrows = dependences
instructions only executed when dependencies ready

data flow model and limits



three ops/cycle (*if* each one cycle)

data flow model and limits



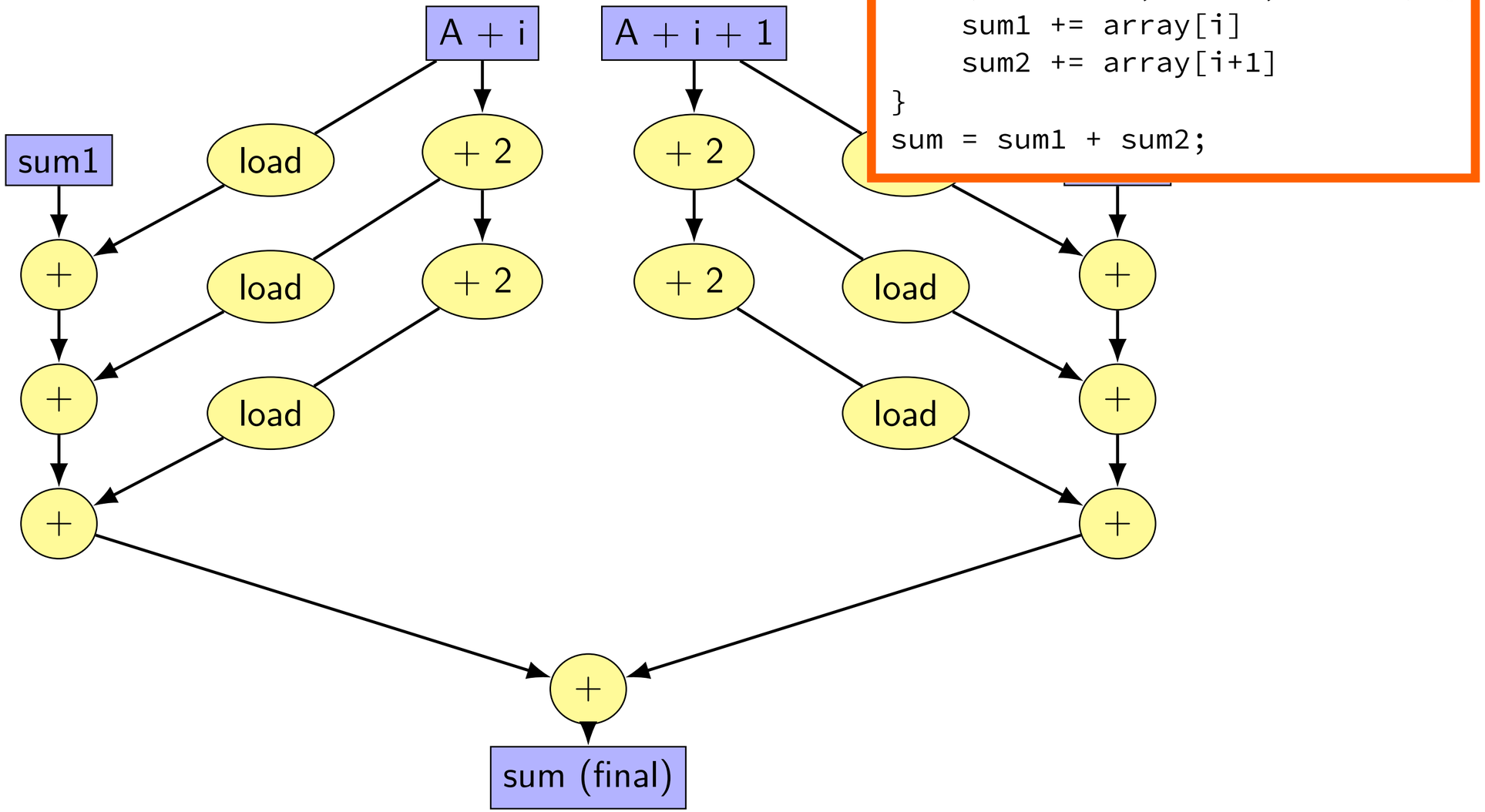
can only do sums one at a time

better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```

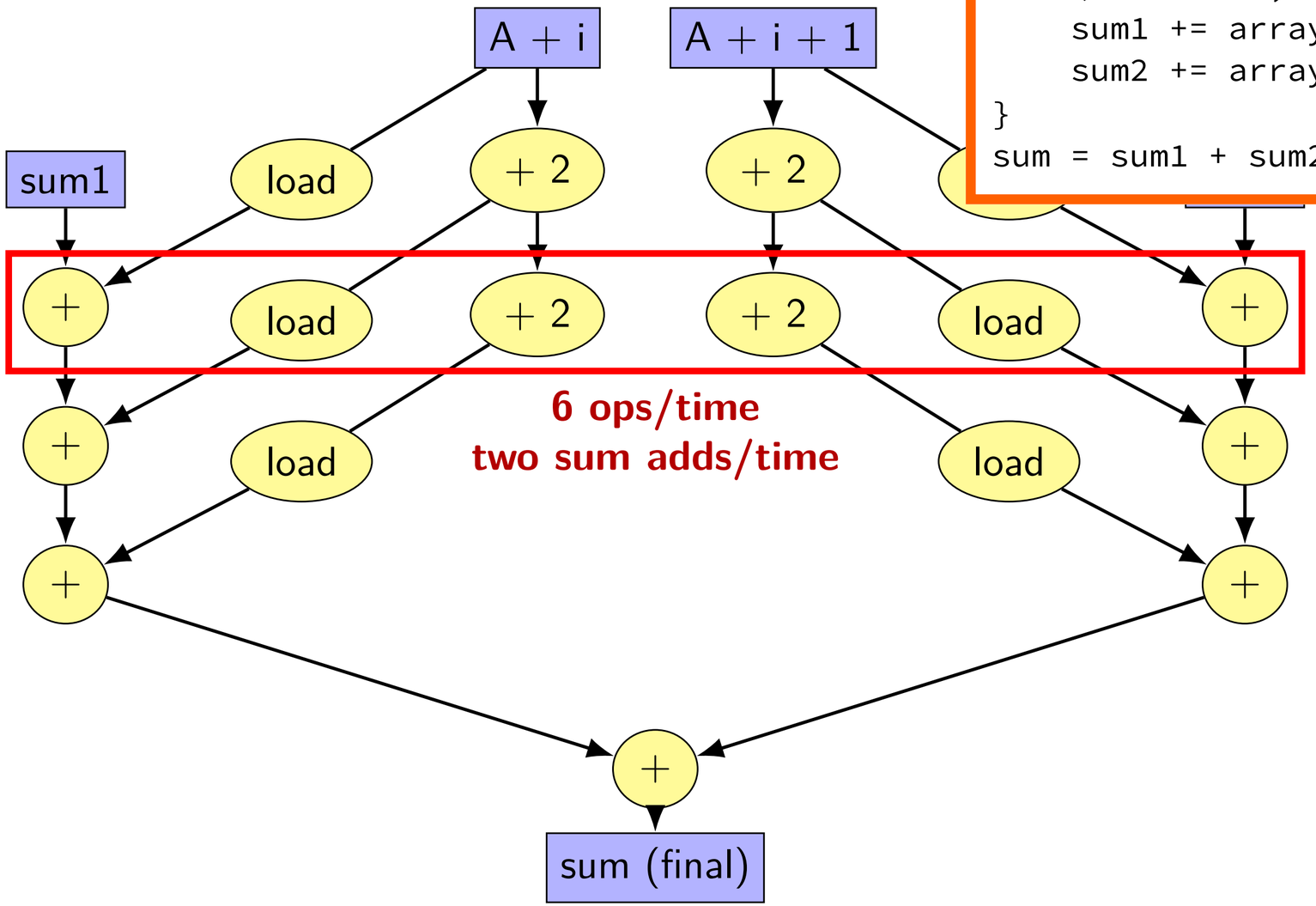
better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```



better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```



better data-flow

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < N; i += 2) {
    sum1 += array[i]
    sum2 += array[i+1]
}
sum = sum1 + sum2;
```