

last time

hazards in pipelines

hazard = extra work needed to make instruction run correctly in pipeline

data hazard = ...reading value with pending update

control hazard = ...can't compute next instruction to fetch

stalling (pause instructions until ready) to resolve hazards

forwarding — take pending value from later in pipeline

MUX to select versus value from register

compare register numbers to see if forwarding needed

can combine with stalling

branch prediction — guess what jump will do

if wrong; undo guess when actual outcome known

anonymous feedback (1)

pipeline assignment — deadline move?

I know we didn't completely cover branch prediction

think assignment text is enough + no quiz due next Tuesday

past experience: assignment is quicker to do than typical assignment

pipeline assignment — how partial credit?

rubric categories checking for things like:

one instruction per stage per cycle

instructions pass through stages in order + never skip stages

identifies when misprediction occurs

instruction X fetched after instruction Y

correctly identifies data hazard requiring stalling

...

on upcoming quiz

next quiz due Tuesday *after Thanksgiving*

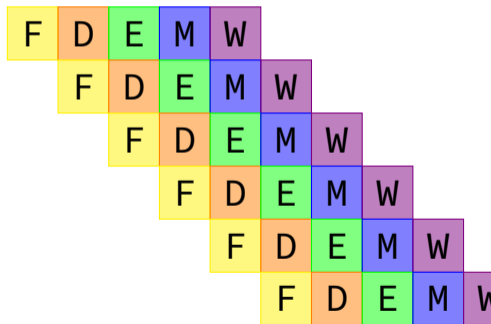
will release tomorrow (so you can start early if you want)

jXX: stalling?

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

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

cycle # 0 1 2 3 4 5 6 7 8



jXX: stalling?

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

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



jXX: stalling?

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

```
...
```

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

```
jne LABEL
```

```
(do nothing)
```

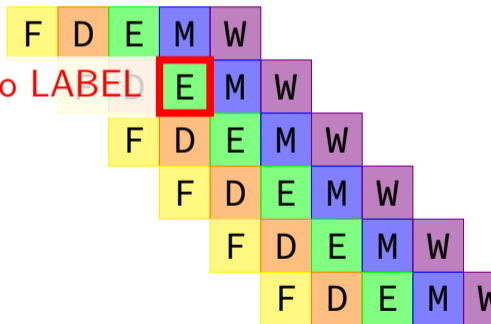
```
(do nothing)
```

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

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

```
...
```

cycle # 0 1 2 3 4 5 6 7 8



jXX: stalling?

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

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

cycle # 0 1 2 3 4 5 6 7 8



making guesses

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

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

speculate (guess): **jne** won't go to LABEL

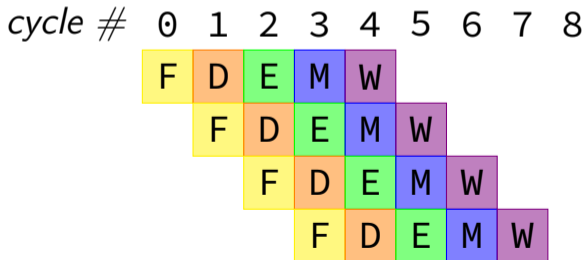
right: 2 cycles faster!; wrong: undo guess before too late

jXX: speculating right (1)

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

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

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



jXX: speculating wrong

cycle # 0 1 2 3 4 5 6 7 8

cmpq %r8, %r9

jne LABEL

xorq %r10, %r11

(inserted nop)

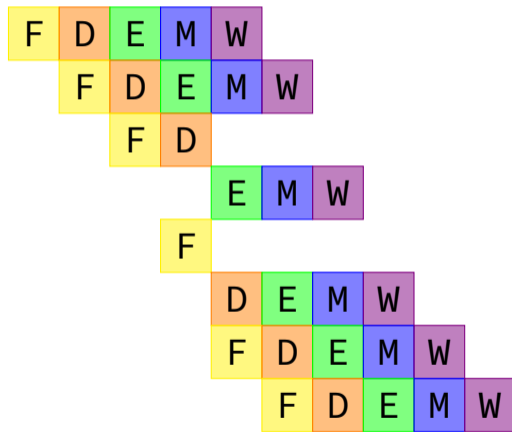
movq %r11, 0(%r12)

(inserted nop)

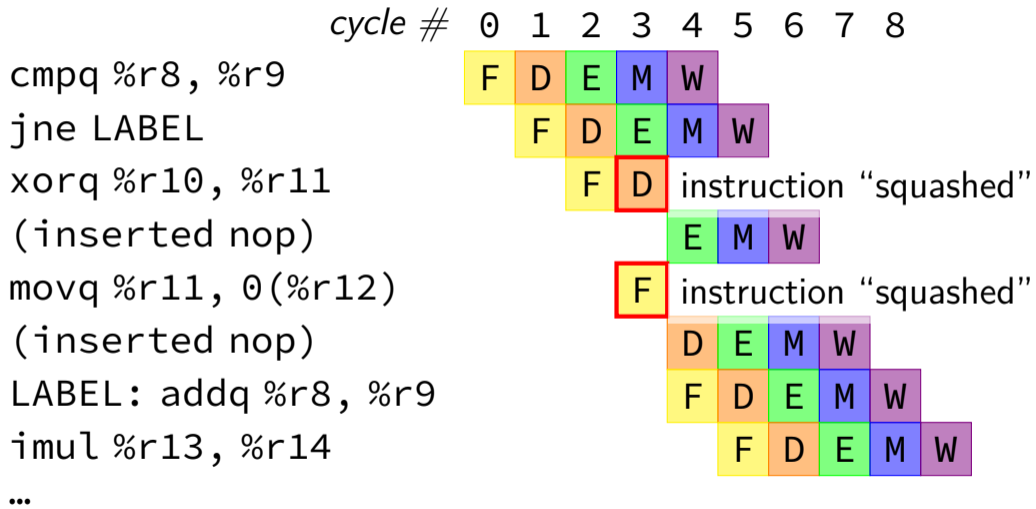
LABEL: addq %r8, %r9

imul %r13, %r14

...



jXX: speculating wrong



“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

more complicated pipelines: replace written values in cache/registers/etc.

performance

hypothetical instruction mix

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

performance

hypothetical instruction mix

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

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

stall: $3 \times .03 + 3 \times .05 + 1 \times .92 =$
1.16 cycles/instr. (1.19 ÷
1.09 ≈ 1.09x faster)

exercise: predict+forward (1)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

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

subq %r7, %r8

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

jle foo (taken)

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

...

...

...

foo: **andq** %r9, %r8

if jle is *correctly predicted*:

in andq, %r9 is _____ addq.

in andq, %r8 is _____ subq.

A: not forwarded from [assume read while writing requires forwarding]

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

exercise: predict+forward (1)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r9

F D E M W

subq %r7, %r8

F D E M W

jle foo (taken)

F D E M W

...

...

...

foo: andq %r9, %r8

F D E M W

if jle is *correctly predicted*:

in andq, %r9 is _____ addq.

in andq, %r8 is _____ subq.

A: not forwarded from [assume read while writing requires forwarding]

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

exercise: predict+forward (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r8

F D E M W

subq %r7, %r8

F D E M W

jle foo (taken)

F D E M W

...

...

...

foo: andq %r9, %r8

if jle is *mispredicted* + resolved after jle's execute:

in andq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

A: not forwarded from [assume read while writing requires forwarding]

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

exercise: predict+forward (2)

cycle # 0 1 2 3 4 5 6 7 8

addq %r8, %r8

F D E M W

subq %r7, %r8

F D E M W

jle foo (taken)

F D E M W

(mispredicted)

F D

(mispredicted)

F

...

foo: andq %r9, %r8

F D E M W

if jle is *mispredicted* + resolved after jle's execute:

in andq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

A: not forwarded from [assume read while writing requires forwarding]

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

other pipelines?

showed fetch / decode / execute / memory / writeback

very common early pipeline design

not only option!

hazards versus dependencies

dependency — X needs result of instruction Y?

has potential for being messed up by pipeline
(since part of X may run before Y finishes)

hazard — will it not work in some pipeline?

before extra work is done to “resolve” hazards
multiple kinds: so far, *data hazards*

ex.: dependencies and hazards (1)

addq %rax, %rbx

subq %rax, %rcx

movq \$100, %rcx

addq %rcx, %r10

addq %rbx, %r10

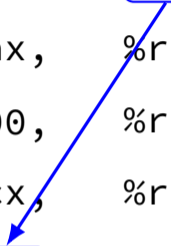
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
movq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10



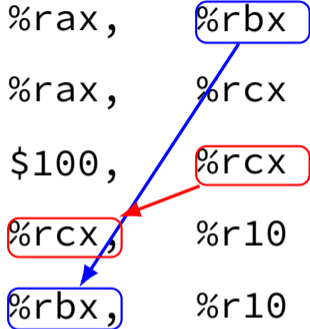
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
movq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10



where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
movq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10

```
graph TD; rbx1["%rbx"] -- blue --> rbx5["%rbx"]; rcx2["%rcx"] -- red --> rcx4["%rcx"]; rcx3["%rcx"] -- red --> rcx4; r104["%r10"] -- red --> r105["%r10"];
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

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

`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
<code>addq %rcx, %r9</code>		F	D	E1	E2	M	W			
<code>addq %r9, %rbx</code>										
<code>addq %rax, %r9</code>										
<code>movq %r9, (%rbx)</code>										

exercise: different pipeline

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

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

r9 not available yet — can't forward here
so try stalling in addq's decode...

exercise: different pipeline

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

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

after stalling once, now we can forward

exercise: different pipeline

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

	<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

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 for_loop_bottom
    add %edi, %eax
foo_loop_bottom:
    dec %edi // edi = edi - 1
    jg for_loop_top // if (edi > 0) goto for_loop_top
    ret
```

suppose `%edi = 3` (initially)

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

how many mispredictions for `je`? for `jg`?

backup slides

connecting things

how to (in hardware) connect A and B?

A

B

connecting things

how to (in hardware) connect A and B?

one wire carrying binary signals?

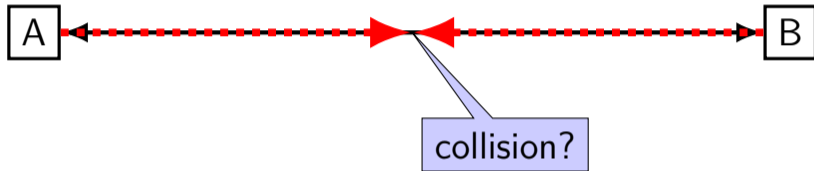
A

A diagram illustrating a single-wire connection between two components, A and B. A horizontal line represents the wire, with a small square box containing the letter 'A' at the left end and a small square box containing the letter 'B' at the right end.

B

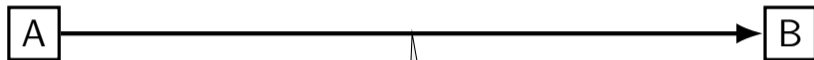
connecting things

how to (in hardware) connect A and B?



connecting things

how to (in hardware) connect A and B?



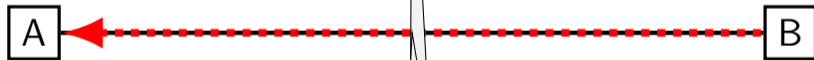
one-way communication only?
called *simplex*

connecting things

how to (in hardware) connect A and B?



...and later



taking turns, but one-way
called *half-duplex*
challenge: how to agree who's turn?

connecting things

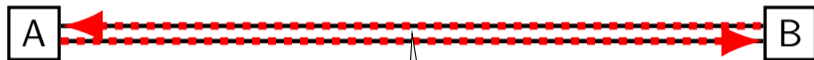
how to (in hardware) connect A and B?



both ways at the same time
called *full duplex* (or *duplex*)

connecting things

how to (in hardware) connect A and B?



here: duplex via multiple wires (simplest scheme)
can achieve effect electrically/etc. via one wire
example: cable Internet
(how is topic for ECE class)

connecting things

A

B

C

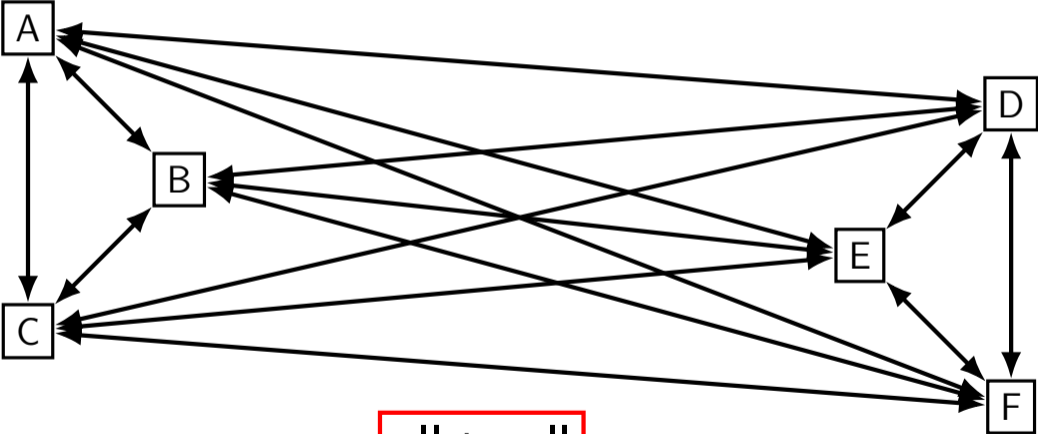
D

E

F

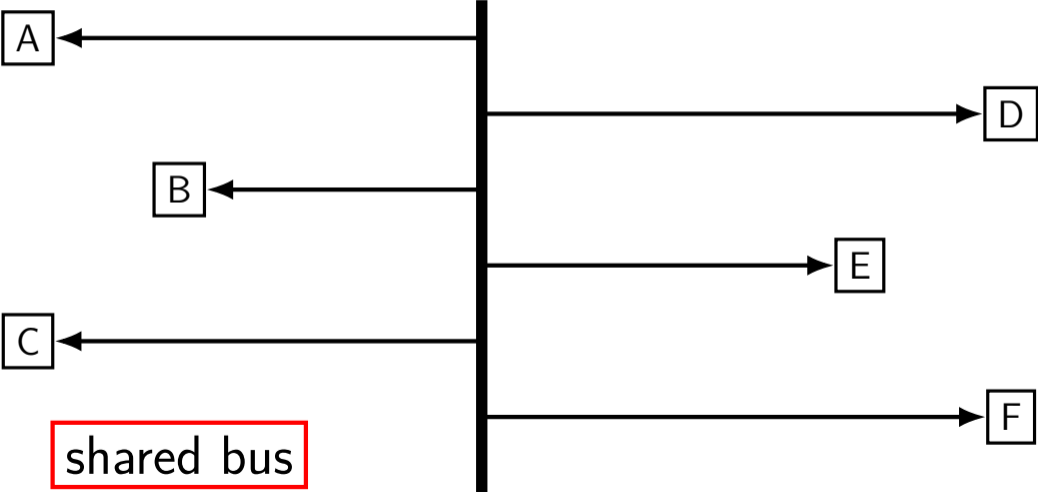
how to connect?

connecting things

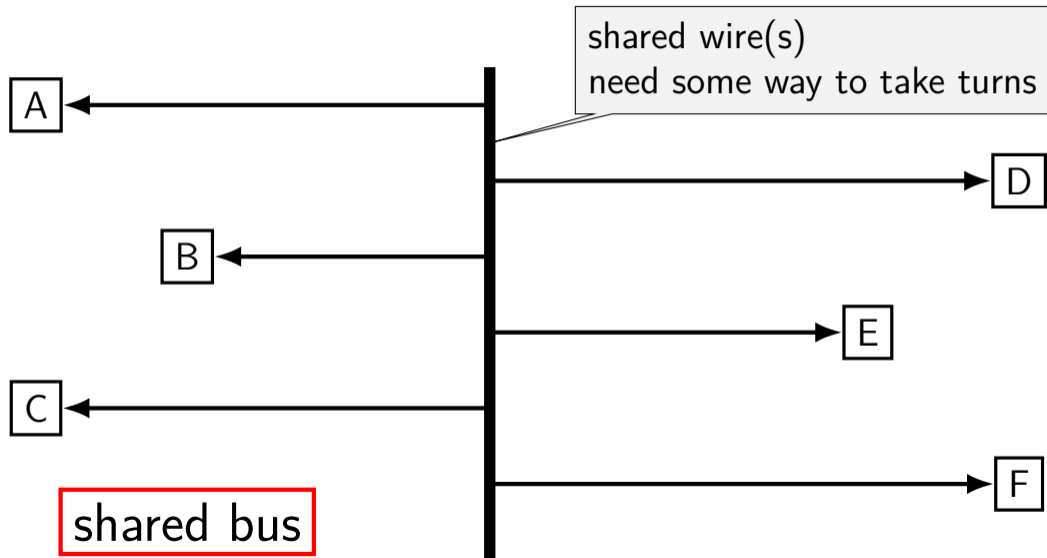


all-to-all

connecting things



connecting things



shared bus, really?

common for parts of internals of computers (topic later)

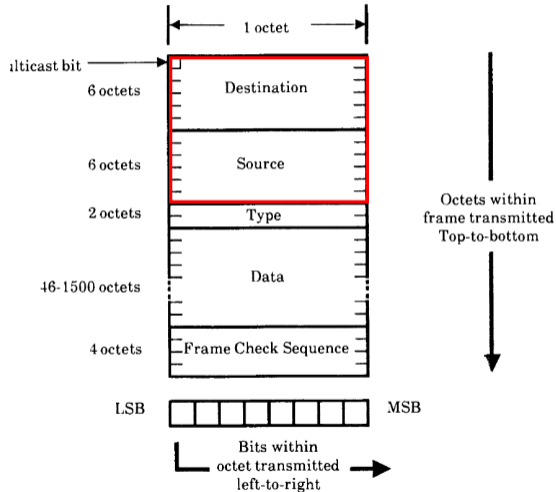
model for wifi

radio “channel” kinda similar to shared wire

how the early versions of Ethernet worked

“vampire taps” physically attached to shared cable

shared bus, messages for who?



messages need a 'header' to tell who it's to/from

everyone needs to filter out messages that aren't theirs

Figure 6-1: Data Link Layer Frame Format

taking turns on shared bus?

token ring

- one machine has a 'token' = can send
- send special message to pass to another machine

free-for-all: collision detection + retry

- detect if you're transmitting when someone else is
- wait (usually randomized amount of time) and retry

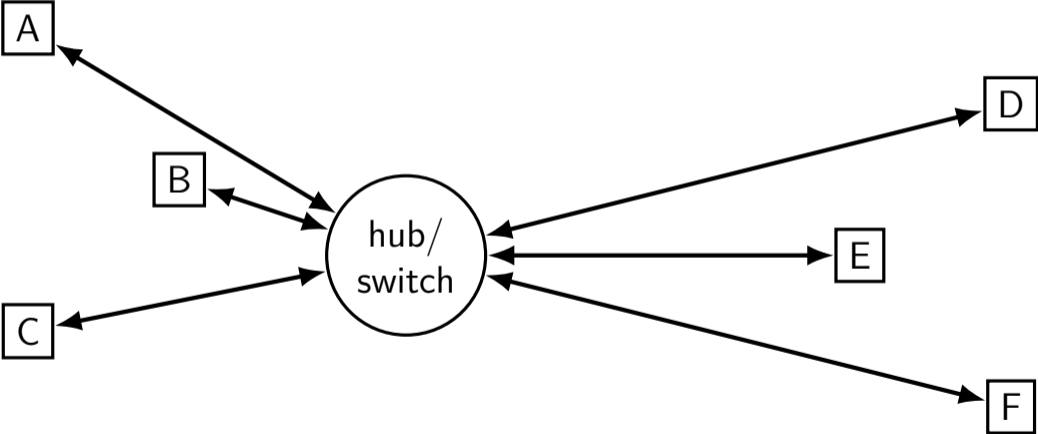
coordinating machine transmits timeslots

- part of common cellphone design (TDMA: time division multiple access)

make bus support multiple transmitters?

- requires understanding how interference works
- another part of common cell phone design

connecting things



what does the hub do?

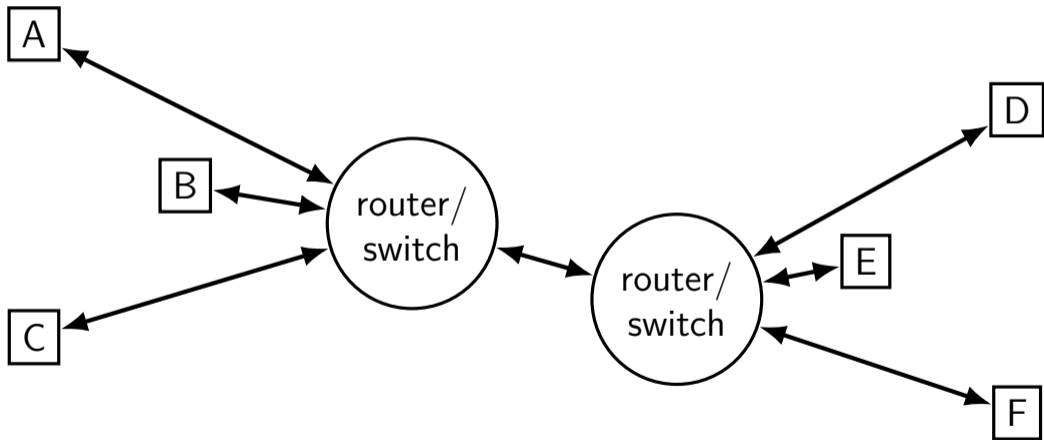
simple version:

imitate shared bus: copy messages to everyone else
something to handle two messages sent at once

less simple:

read “header” on message + send to destination only
requires some way to figure out destinations
queue of messages waiting to be sent

connecting things



more complicated designs

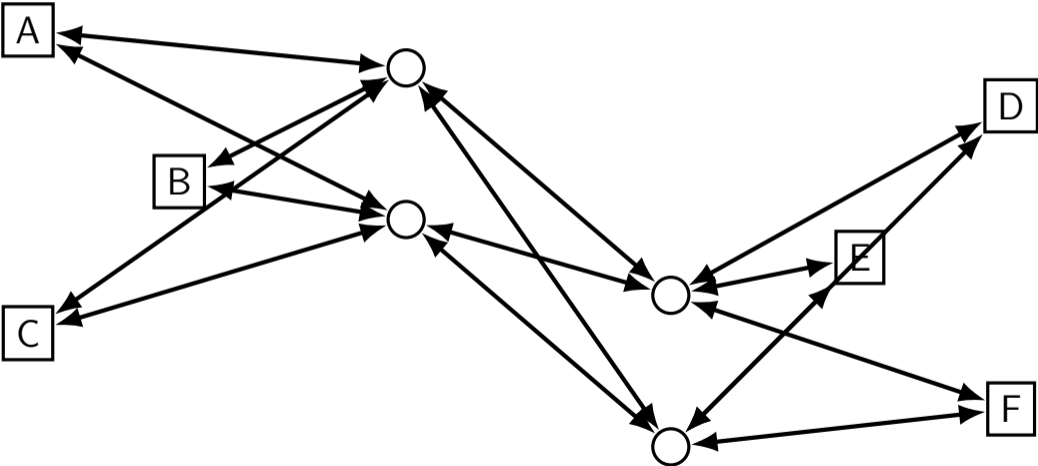
hierarchies

networks of networks

“internetworks”

so far still have single points of failure

connecting things



individual computers are networks

individual computers are (kinda) networks of...

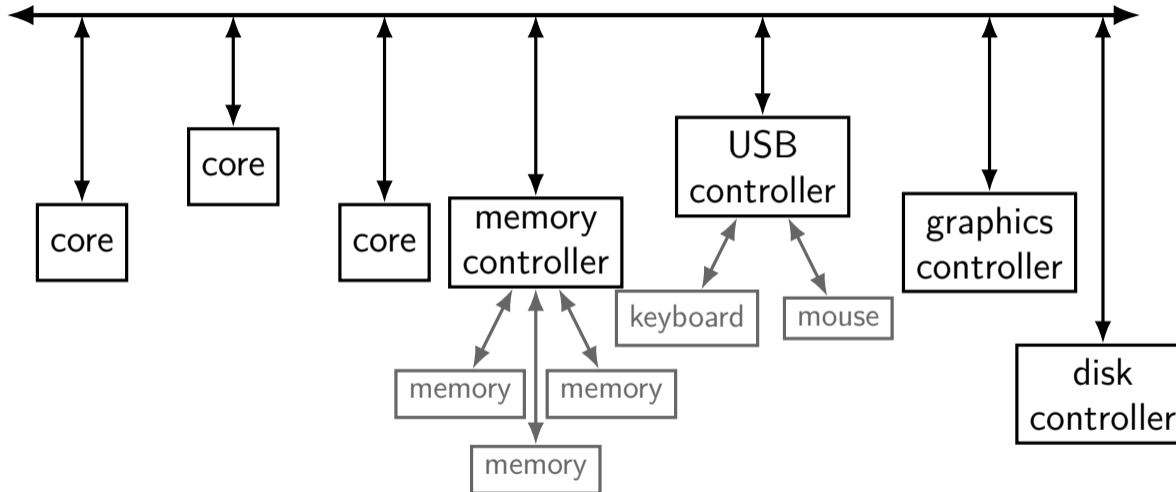
- processors

- memories

- I/O devices

so what topology (layout) do those networks have?

the "bus"



example: 80386 signal pins

name	purpose	
CLK2	clock for bus	timing
W/R#	write or read?	metadata
D/C#	data or control?	
M/IO#	memory or I/O?	
INTR	interrupt request	
...	other metadata signals	
BE0#-BE3#	(4) byte enable	address
A2-A31	(30) address bits	
DO-D31	(32) data signals	data

example: AMD EPYC (1 socket)

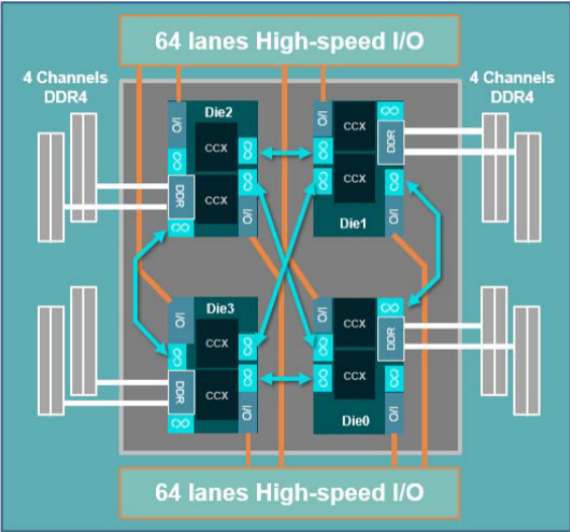


Fig. 21. Single-socket AMD EPYC™ system (SP3).

Figure from Burd et al, "Zeppelin: An SoC for Multichip Architectures" (IEEE JSSC Vol 54, No 1)

example: Intel Skylake-SP

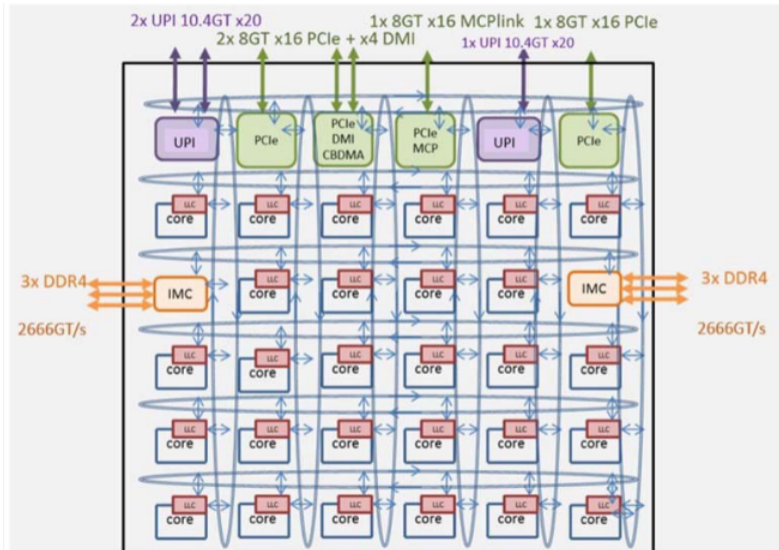
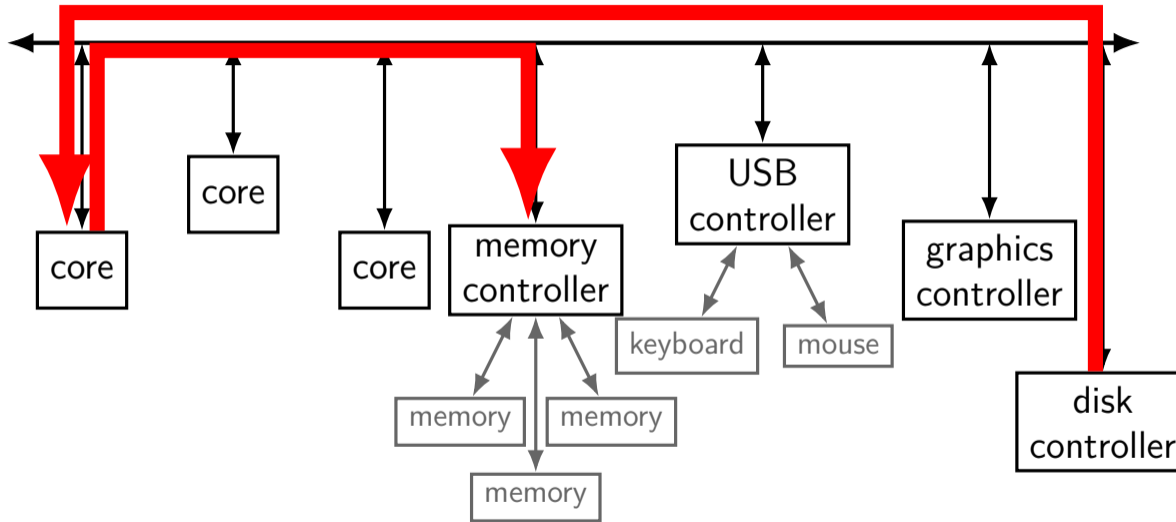
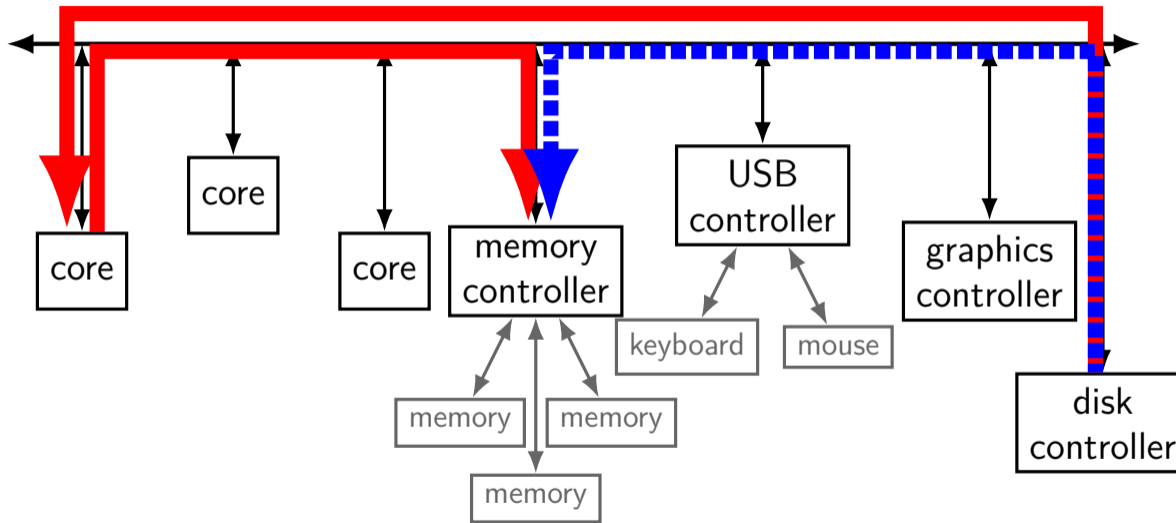


Figure from Tam et al, "SkyLake-SP: A 14nm 28-Core Xeon® Processor" (ISSCC 2018)

extra trips to CPU

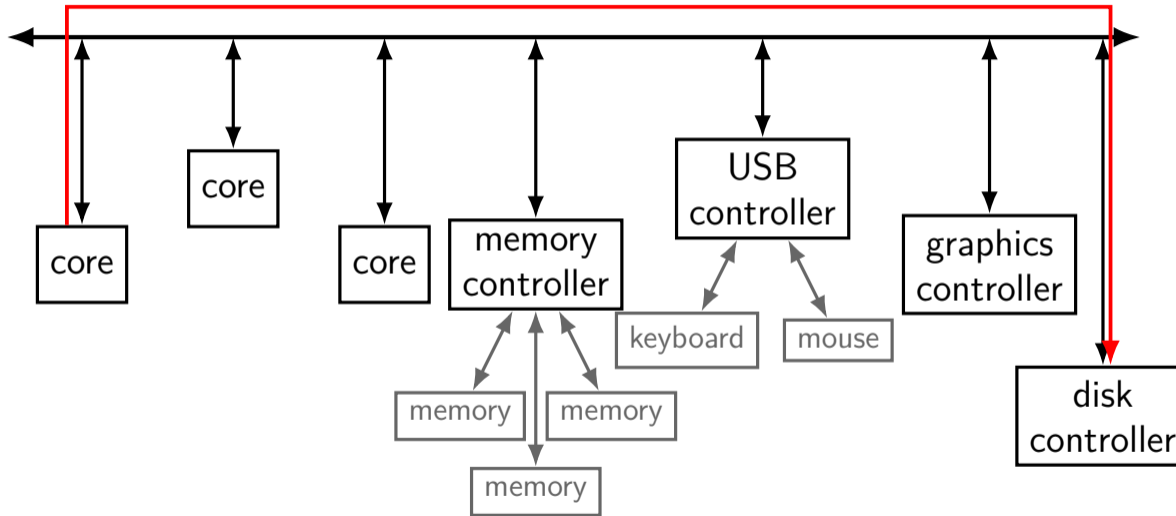


extra trips to CPU



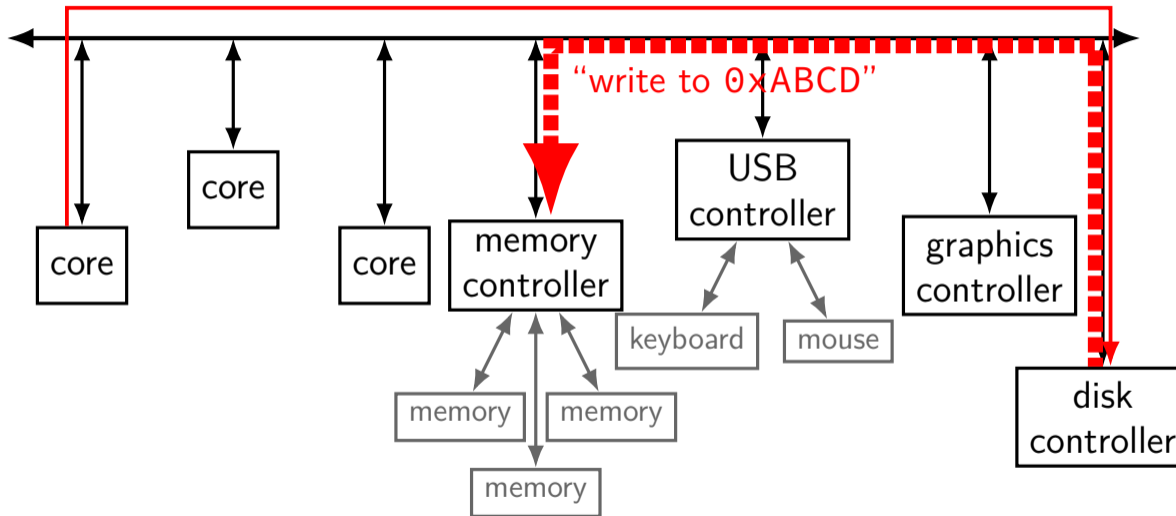
DMA

“place data at 0xABCD”



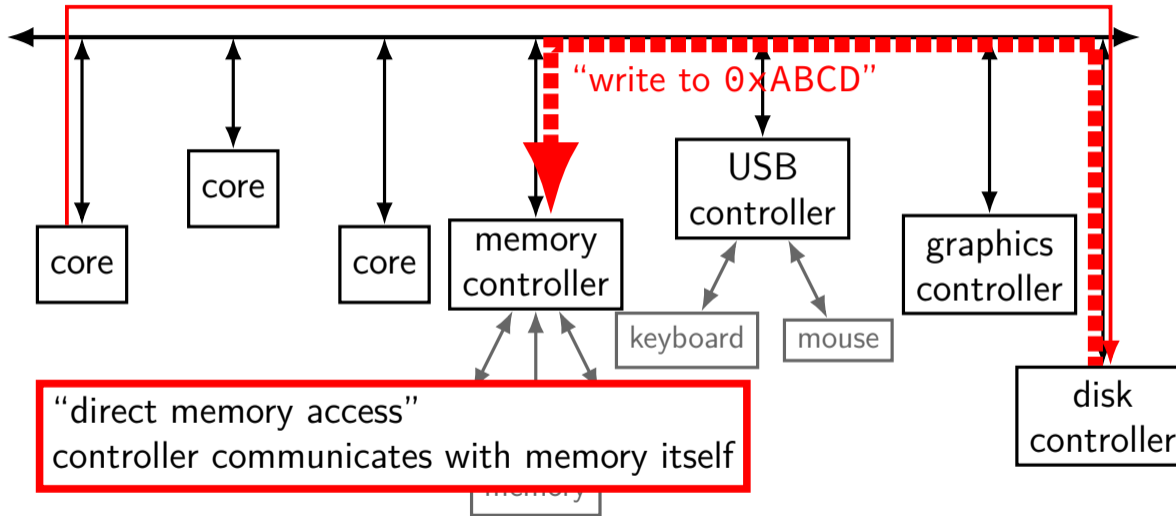
DMA

“place data at 0xABCD”

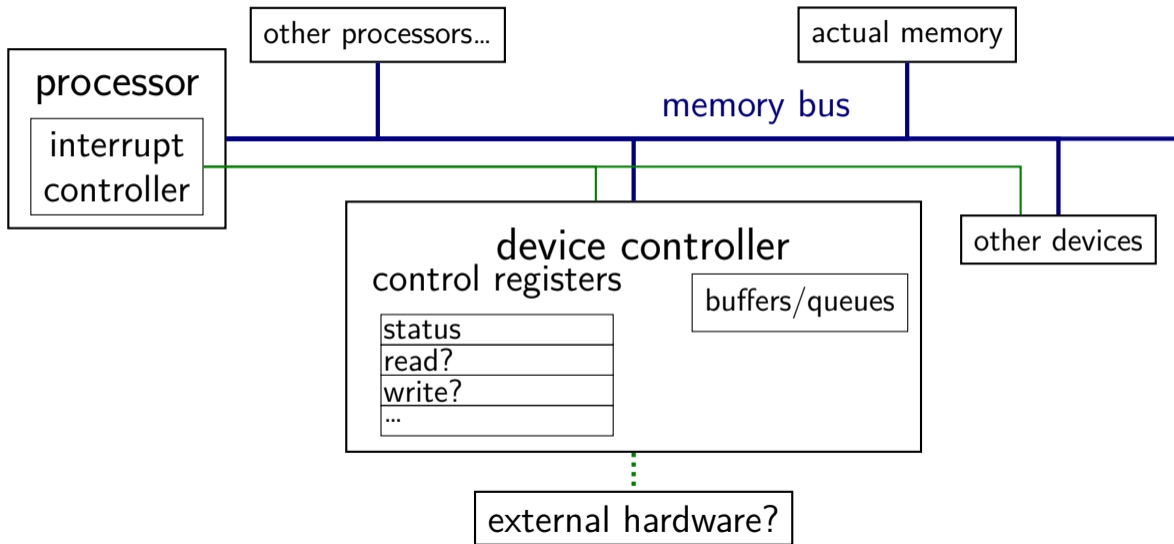


DMA

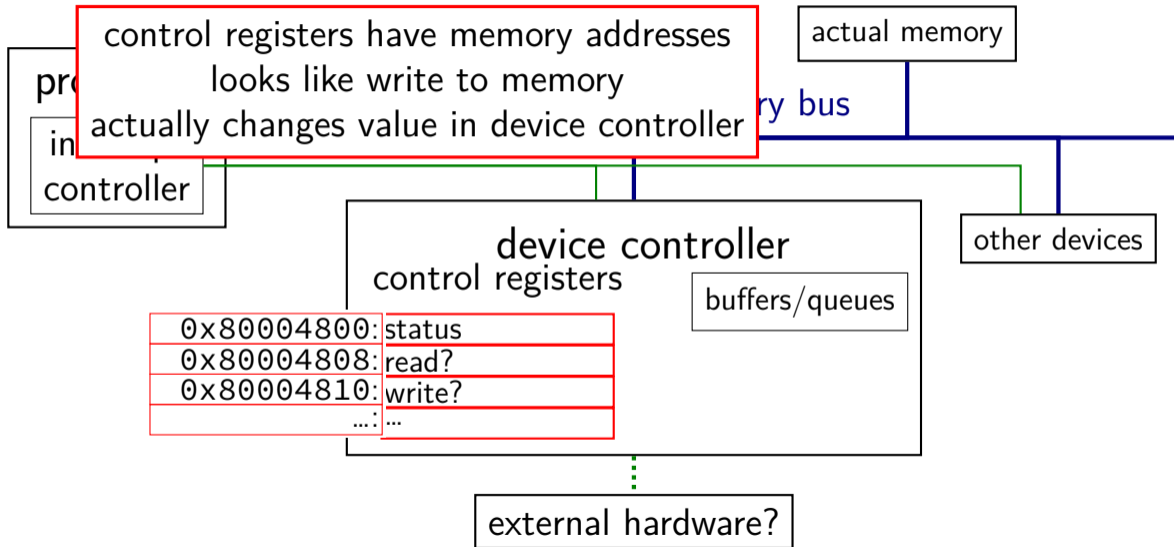
“place data at 0xABCD”



connecting devices

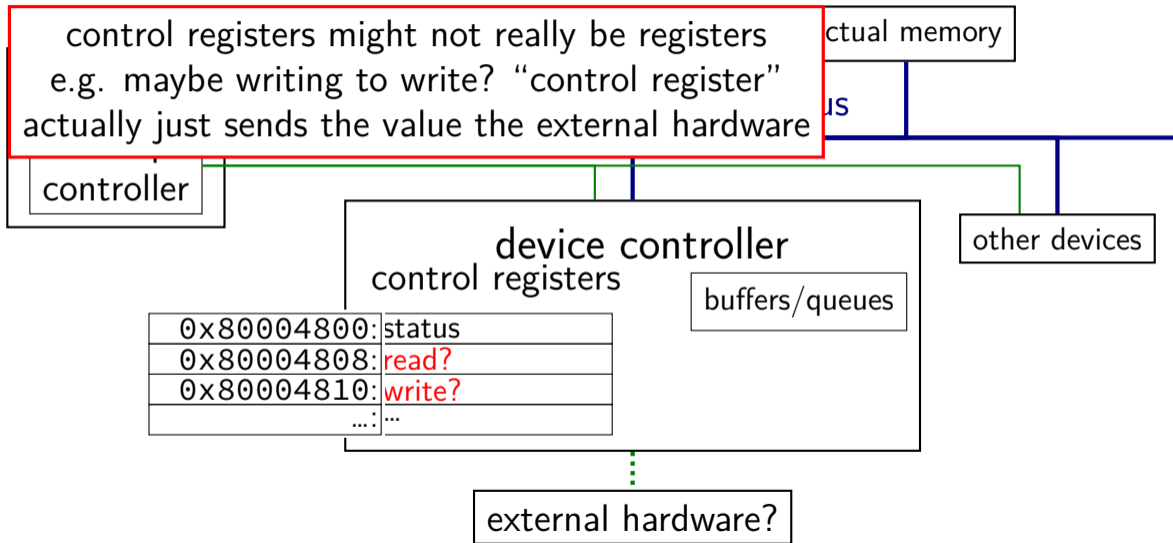


connecting devices

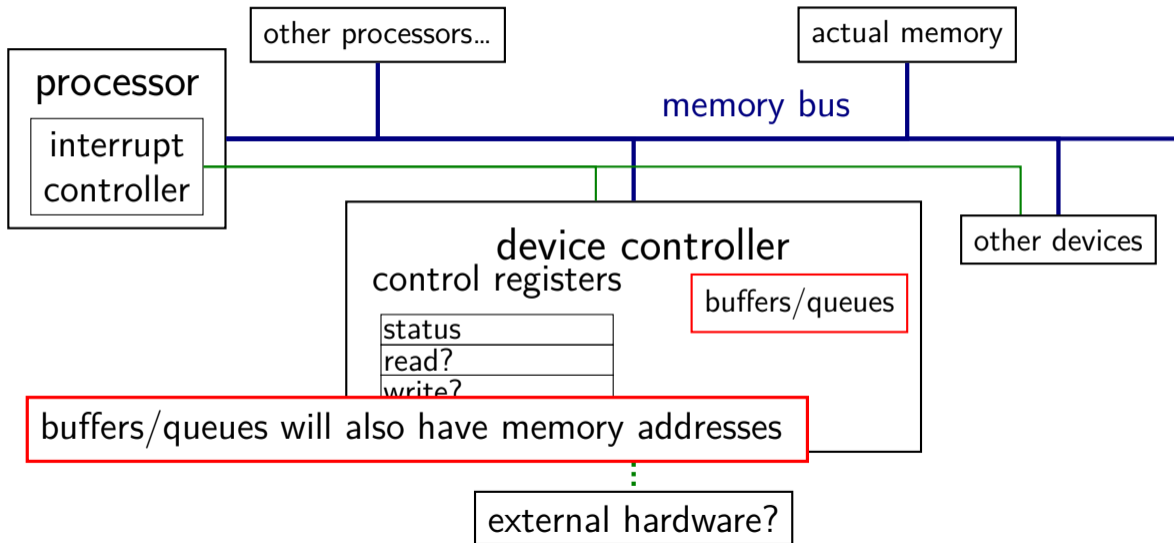


connecting devices

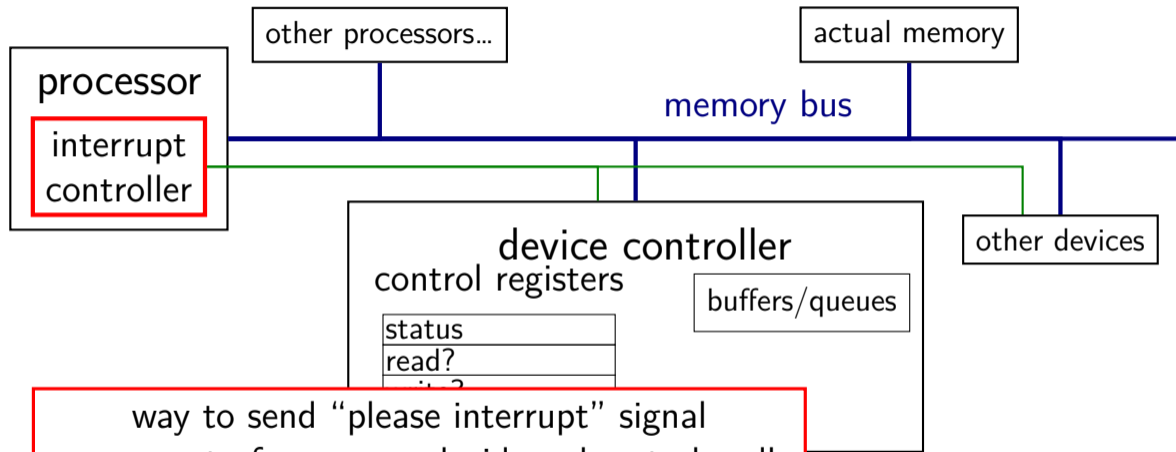
control registers might not really be registers
e.g. maybe writing to write? "control register"
actually just sends the value the external hardware



connecting devices



connecting devices



way to send “please interrupt” signal
component of processor decides when to handle
(deals with ordering, interrupt disabling,
which of several processors handles it, ..., etc.)