TLBs

memory HW

random memory image

page tables with 1-byte page entries

answer: 2-byte values read (or replaced) or "fault"

3 attempts per set of problems

- submitting only right and blank answers — doesn't count as attempt

keep getting new sets of problems until you get it right

cache accesses and multi-level PTs

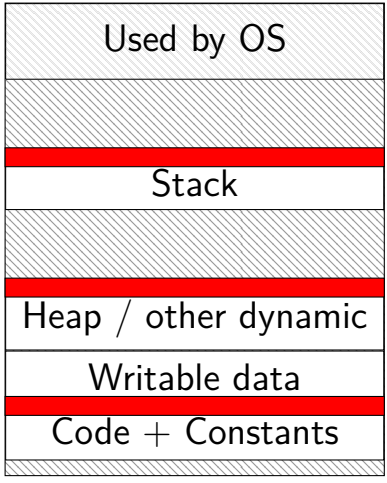
four-level page tables — four cache accesses per memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each memory access?

not acceptable

program memory active sets



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F...

small areas of memory active at a time
one or two pages in each area?

0x0000 0000 0040 0000

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

needed page table entries are very small

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

only caches the page table lookup itself
(generally) just entries from the last-level page table

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

not much spatial locality between page table entries
(they're used for kilobytes of data already)

page table entry cache

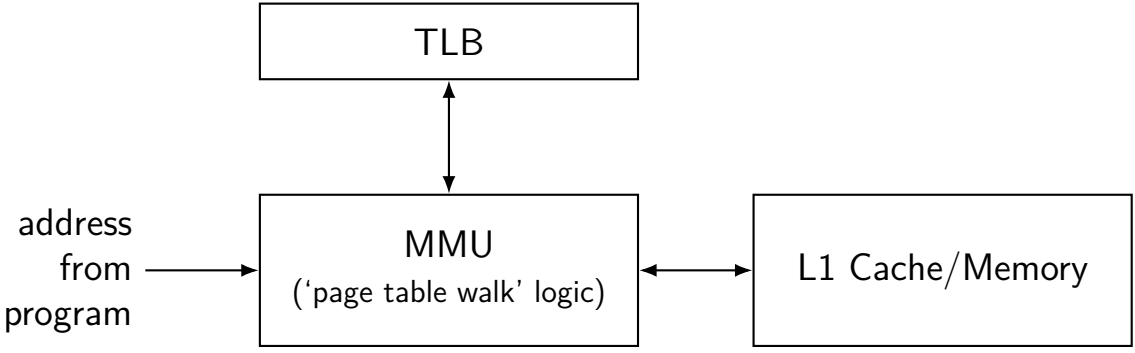
caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

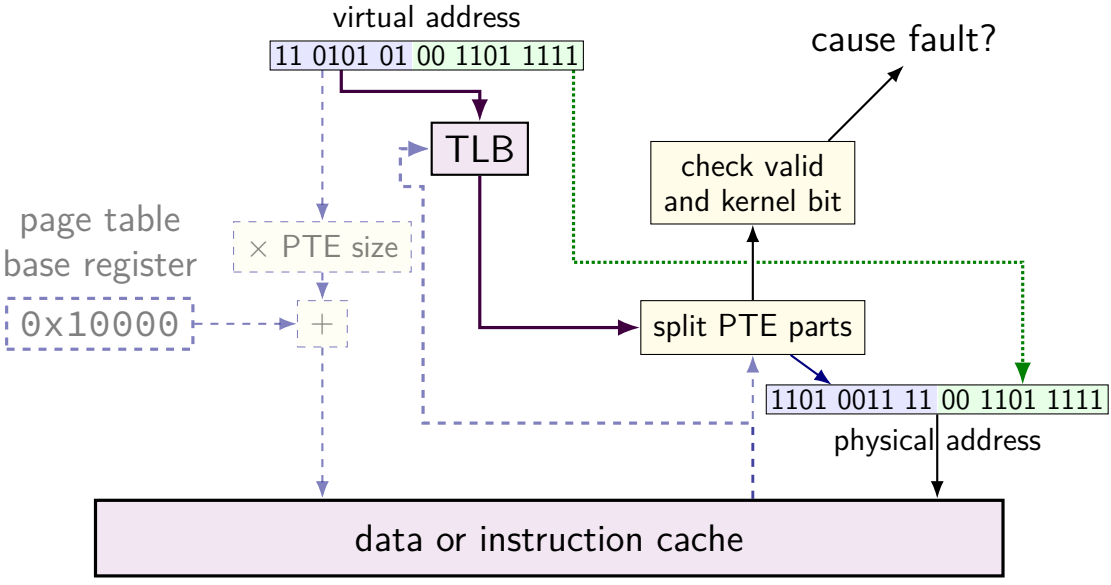
L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

few active page table entries at a time
enables highly associative cache designs

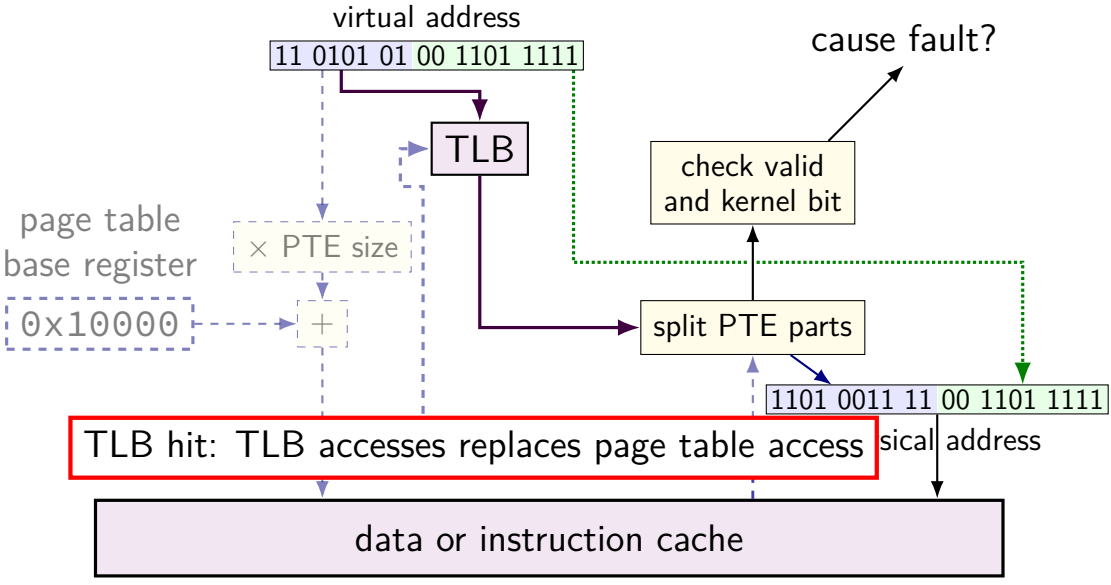
TLB and the MMU (1)



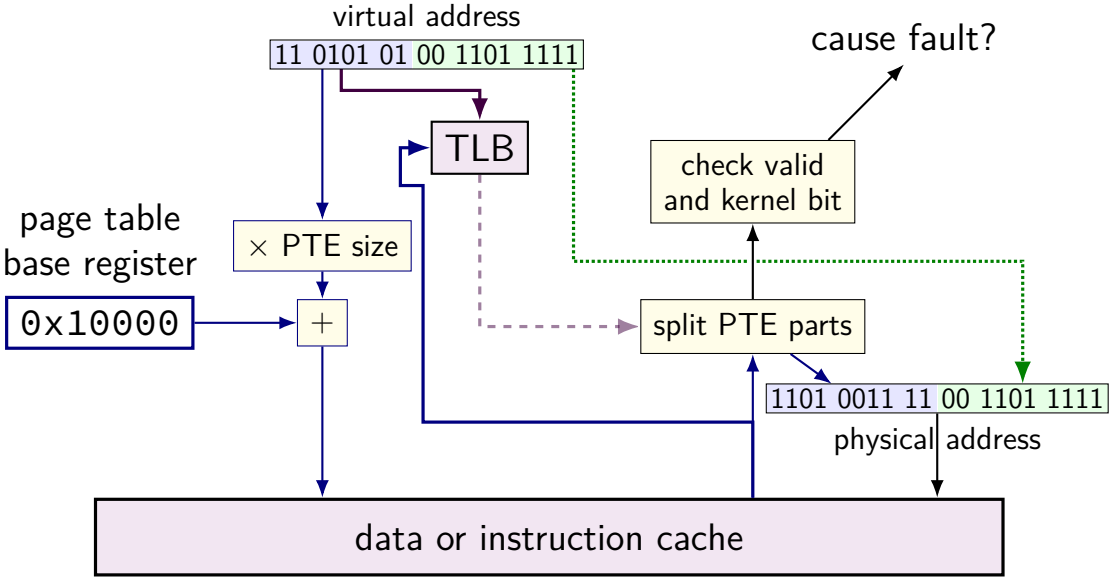
TLB and the MMU (2)



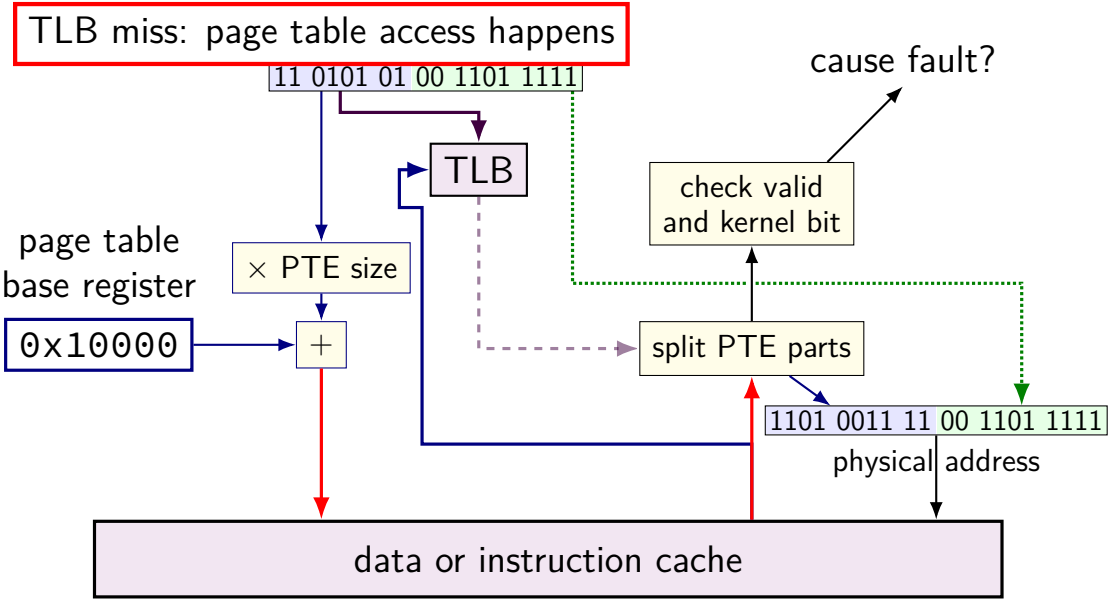
TLB and the MMU (2)



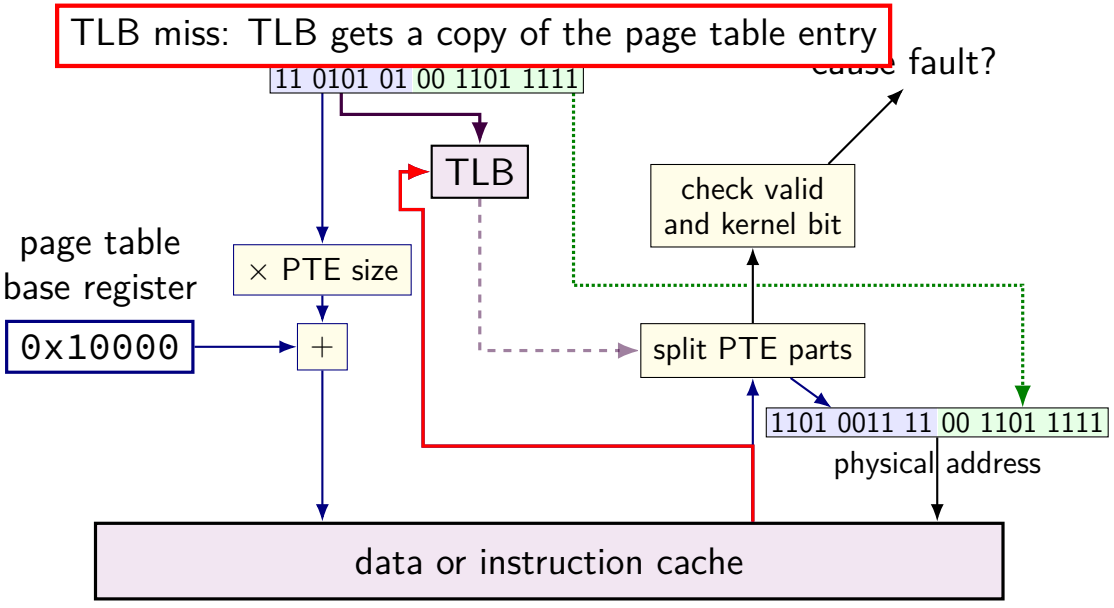
TLB and the MMU (2)



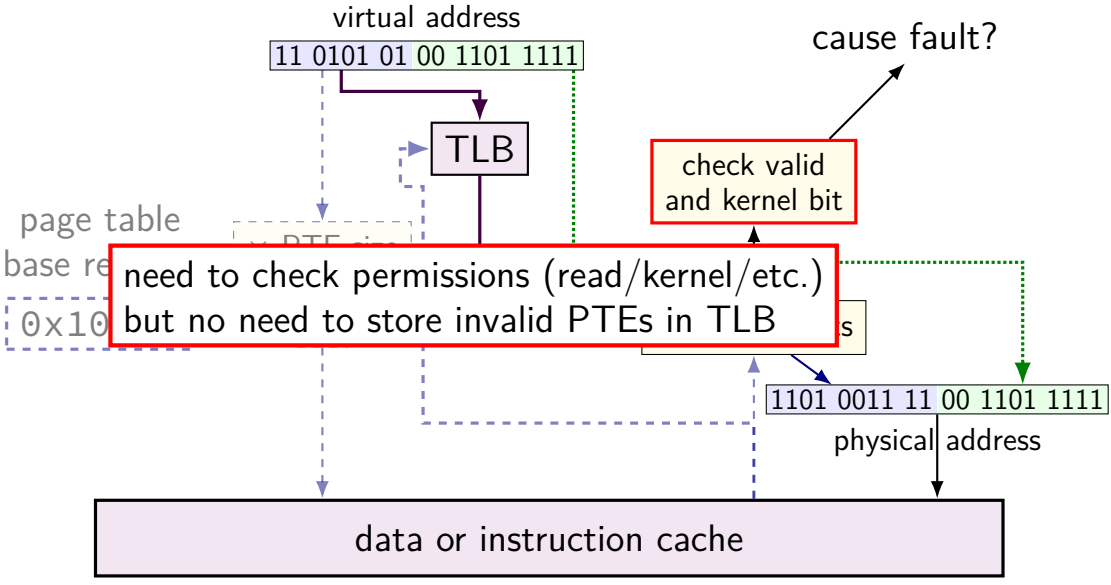
TLB and the MMU (2)



TLB and the MMU (2)



TLB and the MMU (2)



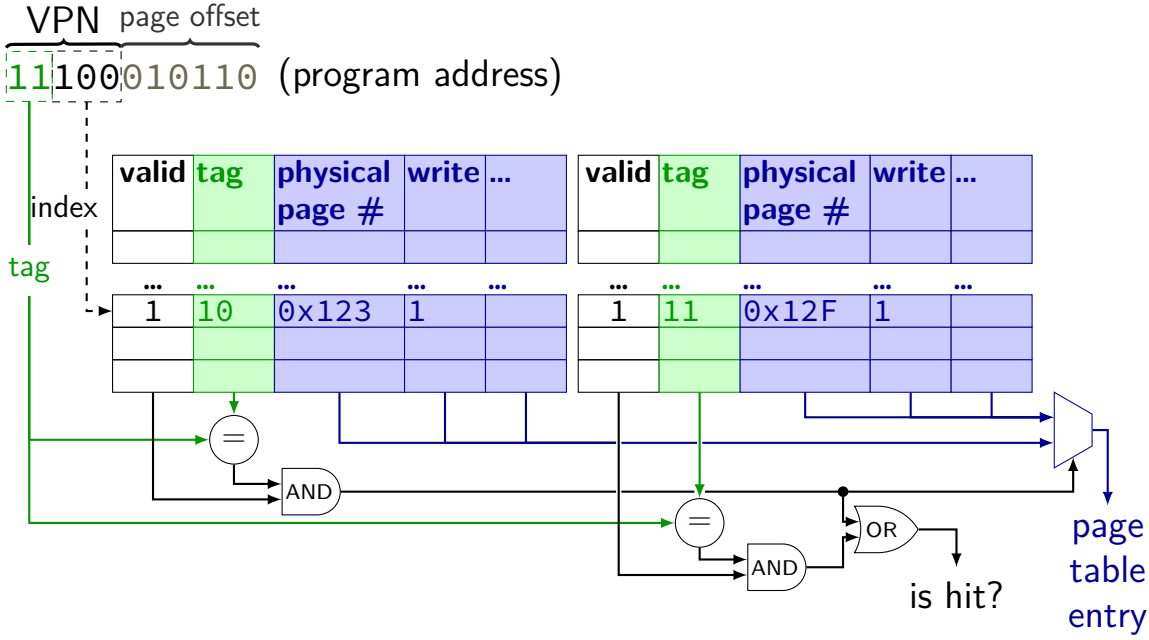
TLB and multi-level page tables

TLB caches valid last-level page table entries

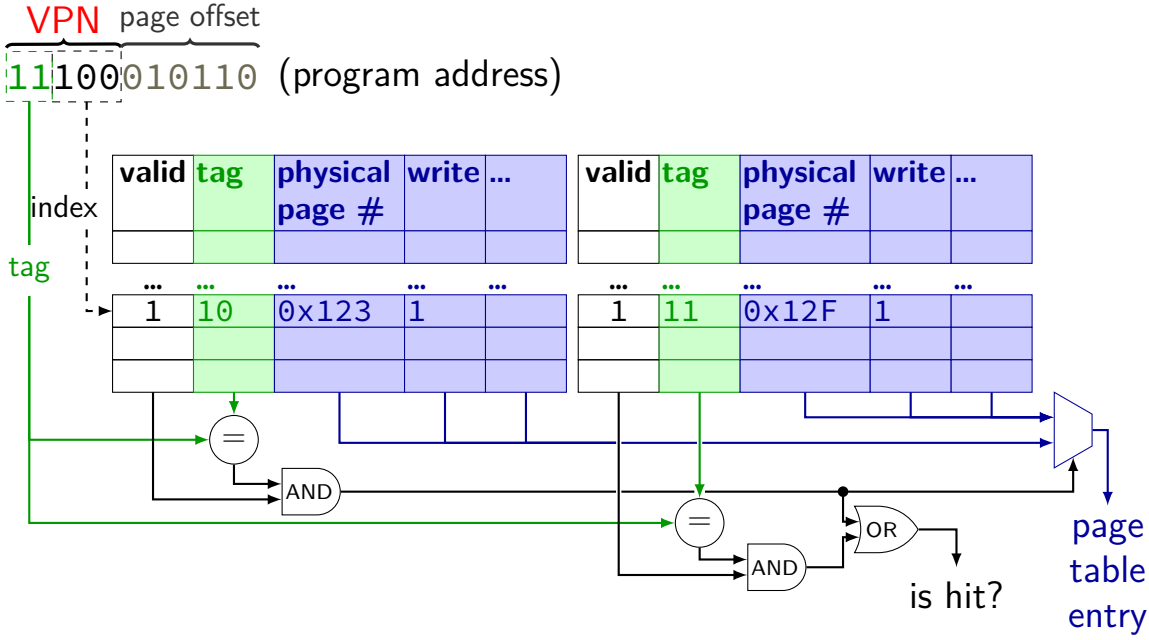
doesn't matter which last-level page table

means TLB output can be used directly to form address

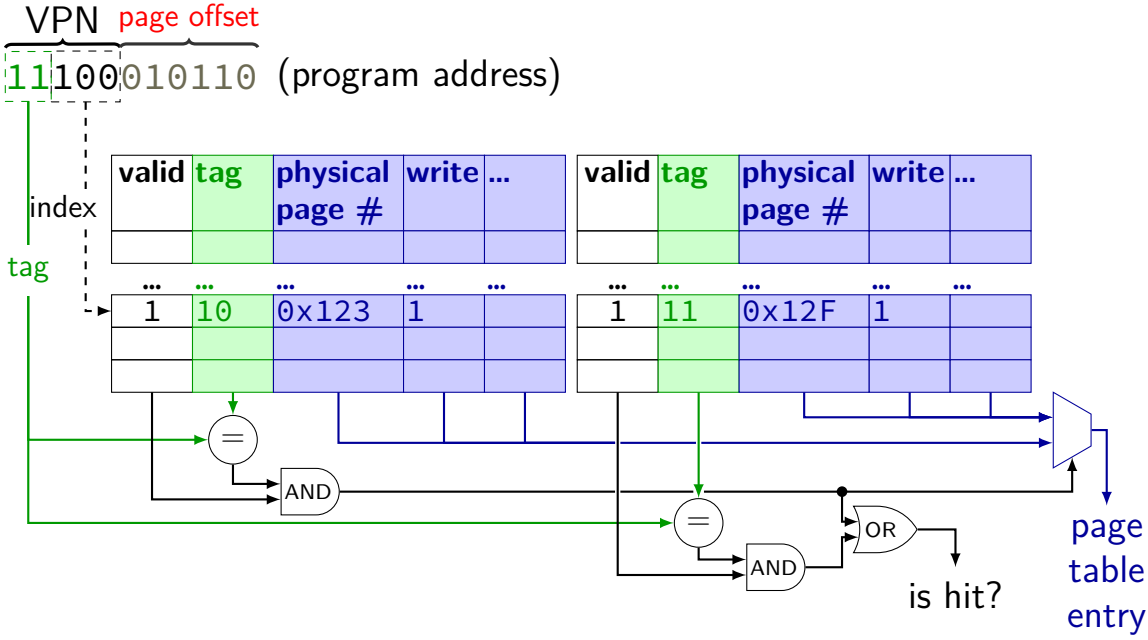
TLB organization (2-way set associative)



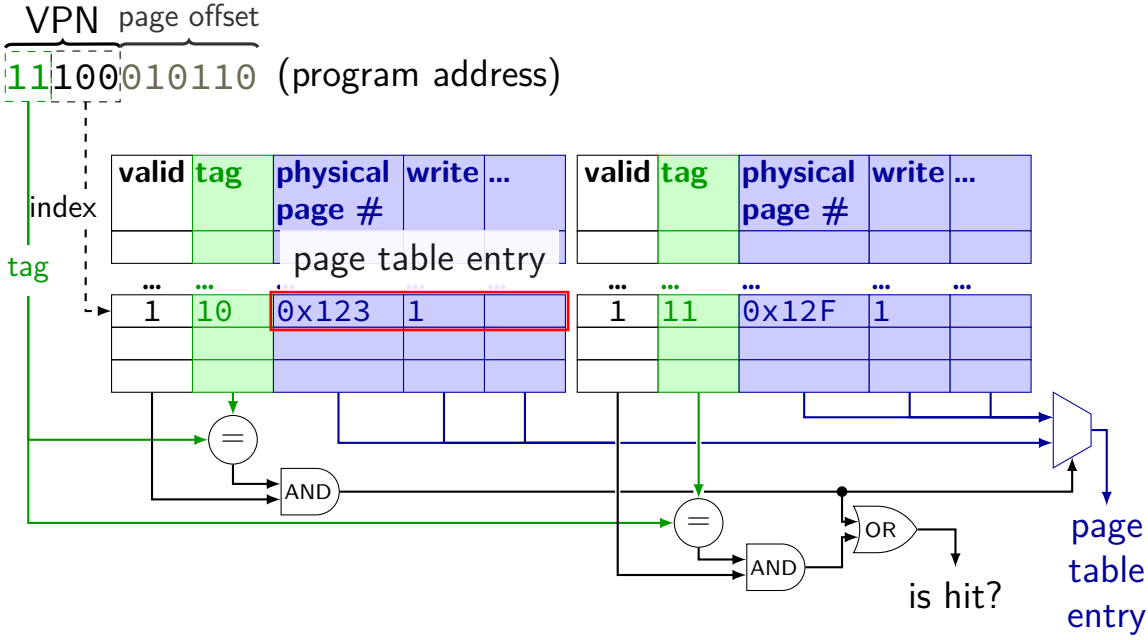
TLB organization (2-way set associative)



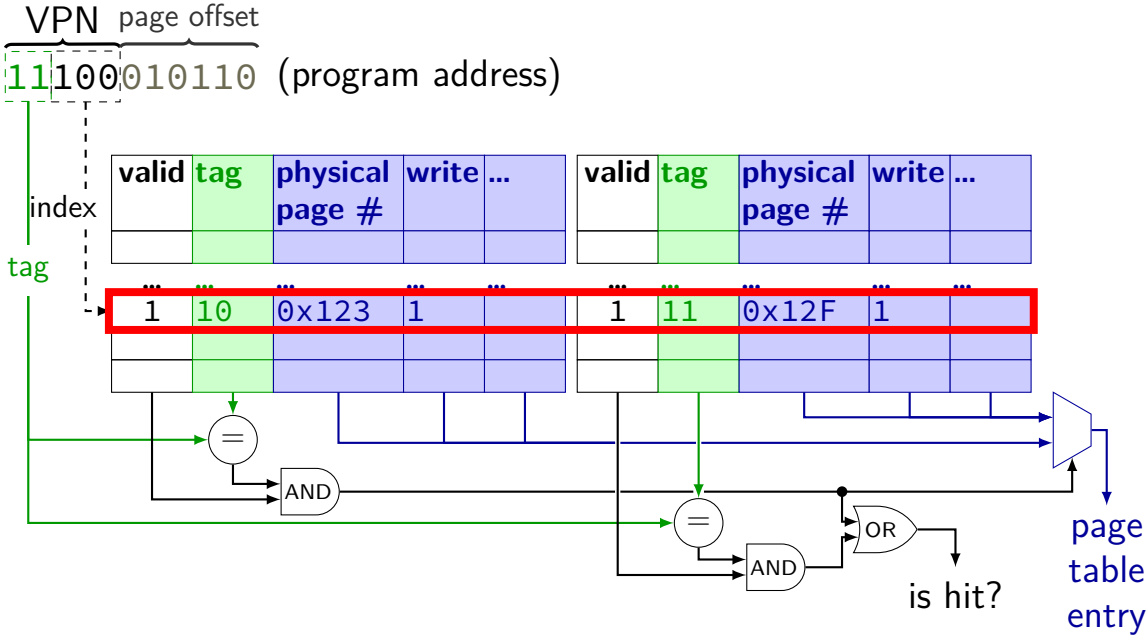
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

$64/4 = 16$ sets — 4 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual address — $36 - 4 = 32$ bit TLB tag

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

$1536/12 = 128$ sets — 7 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual address — $36 - 7 = 29$ bit TLB tag

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

256 entries in first level; 2048 in second

split the address `0x12345678`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

```
0001 0010 0011 0100 0101 0110 0111 1000
```

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address 0x12345678

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address `0x12345678`

- `0001 0010 0011 0100 0101 0110 0111 1000`
- 13-bit page offset `1 0110 0111 1000`
- $32 - 13 = 19$-bit VPN `0001 0010 0011 0100 010`
- 8-bit first part of VPN `0001 0010`
- 11-bit second part of VPN `0011 0100 010`

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address 0x12345678

- 0001 0010 0011 0100 0101 0110 0111 1000
- 13-bit page offset 1 0110 0111 1000
- $32 - 13 = 19$-bit VPN 0001 0010 0011 0100 010
- 8-bit first part of VPN 0001 0010
- 11-bit second part of VPN 0011 0100 010
- 7-bit TLB index 0100 010

address splitting exercise (3)

384-entry, 3-way set-associative TLB

32-bit virtual address; 8KB pages

2-level page table; 4 byte PTEs

- 256 entries in first level; 2048 in second

split the address 0x12345678

- 0001 0010 0011 0100 0101 0110 0111 1000
- 13-bit page offset 1 0110 0111 1000
- $32 - 13 = 19$-bit VPN 0001 0010 0011 0100 010
- 8-bit first part of VPN 0001 0010
- 11-bit second part of VPN 0011 0100 010
- 7-bit TLB index 0100 010
- $19 - 7 = 12$-bit TLB tag 0001 0010 0011

TLB example: address splitting

16-bit virtual addresses

64-byte pages

8-entry, 2-way TLB

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	0	????	—	0	????	—	?
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	
110100000001 (D01)	
000100001010 (10A)	
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	0	????	—	0	????	—	?
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

VPN

address (hex)	hit?
000100000000 (100)	
110100000001 (D01)	
000100001010 (10A)	
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	0	????	—	0	????	—	?
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	
110100000001 (D01)	
000100001010 (10A)	
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	0	????		way 1
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	
000100001010 (10A)	
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	1	1101	for VPN 110100	way 0
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	1	1101	for VPN 110100	way 1
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	hit
110100100001 (D21)	
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	1	1101	for VPN 110100	way 0
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	0	????	—	0	????	—	?

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	hit
110100100001 (D21)	hit
000011111100 (0FC)	
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	1	1101	for VPN 110100	way 0
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	1	0000	for VPN 000011		????	—	way 1

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	hit
110100100001 (D21)	hit
000011111100 (0FC)	miss
110011111000 (CF8)	
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	0001	for VPN 000100	1	1101	for VPN 110100	way 0
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	1	0000	for VPN 000011	1	1100	for VPN 110011	way 0

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	hit
110100100001 (D21)	hit
000011111100 (0FC)	miss
110011111000 (CF8)	miss
111100101000 (F23)	

VPN

tag index

TLB access pattern

64-byte pages

8 entries, 4 sets

index	V	tag	PTE	V	tag	PTE	LRU
00	1	1111	for VPN 111100	1	1101	for VPN 110100	way 1
01	0	????	—	0	????	—	?
10	0	????	—	0	????	—	?
11	1	0000	for VPN 000011	1	1100	for VPN 110011	way 0

address (hex)	hit?
000100000000 (100)	miss
110100000001 (D01)	miss
000100001010 (10A)	hit
110100100001 (D21)	hit
000011111100 (0FC)	miss
110011111000 (CF8)	miss
111100101000 (F23)	miss

VPN

tag index

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

option 2: TLB entries contain process ID

- set by OS (special register)
- checked by TLB in addition to TLB tag, valid bit

editing page tables

what happens to TLB when OS changes a page table entry?

invalid to valid — nothing needed

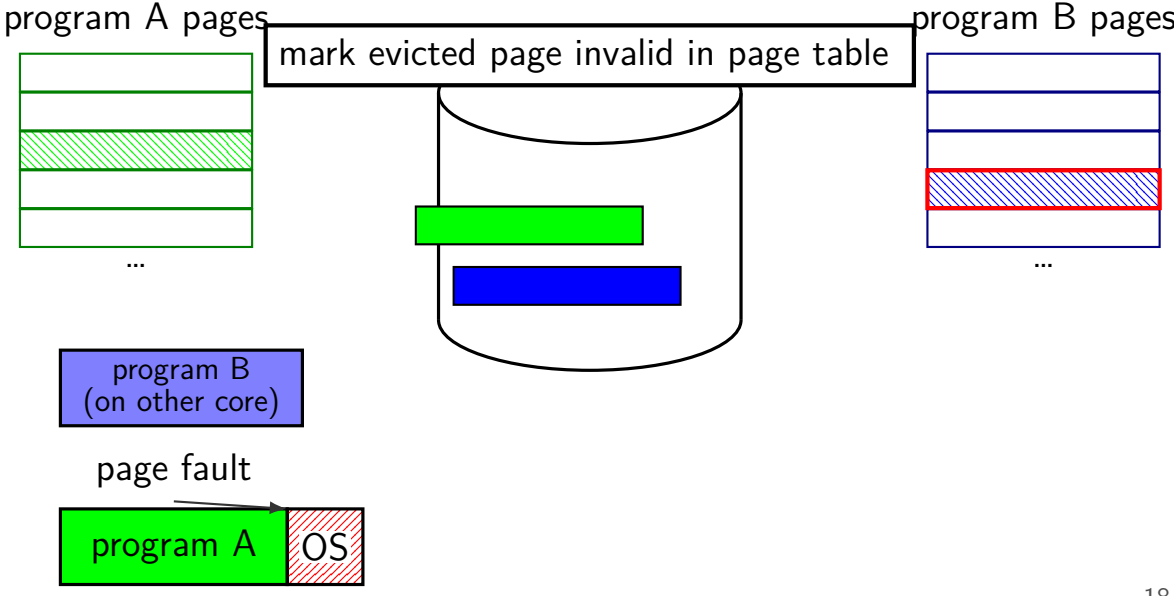
- TLB doesn't contain invalid entries
- MMU will check memory again

valid to invalid — OS needs to tell processor to invalidate it

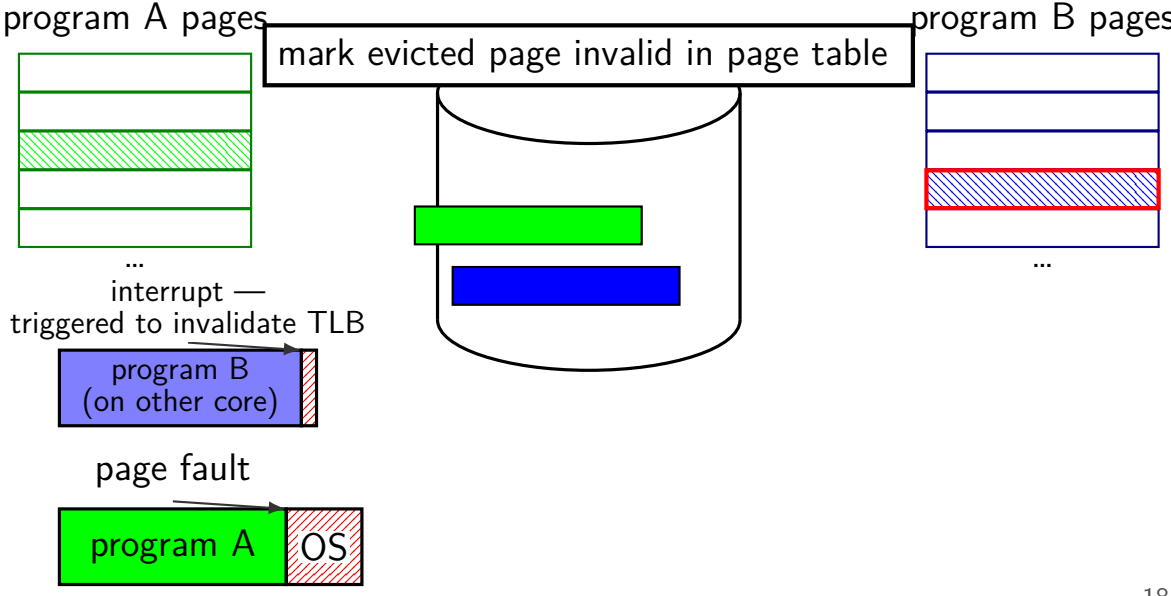
- special instruction (x86: `invlpg`)

valid to other valid — OS needs to tell processor to invalidate it

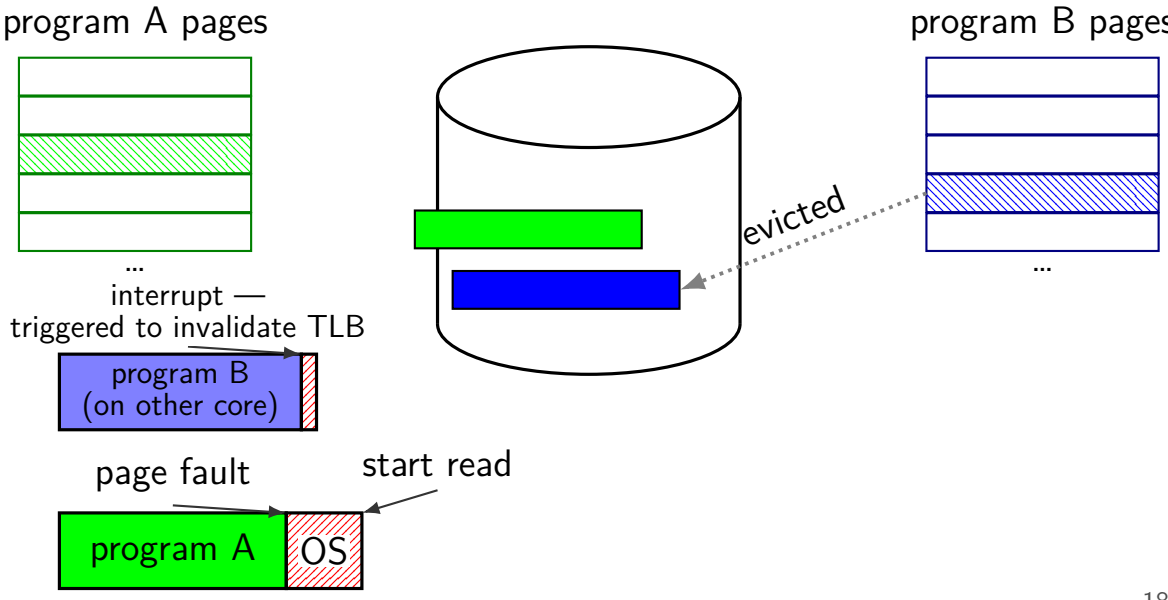
TLB shootdown



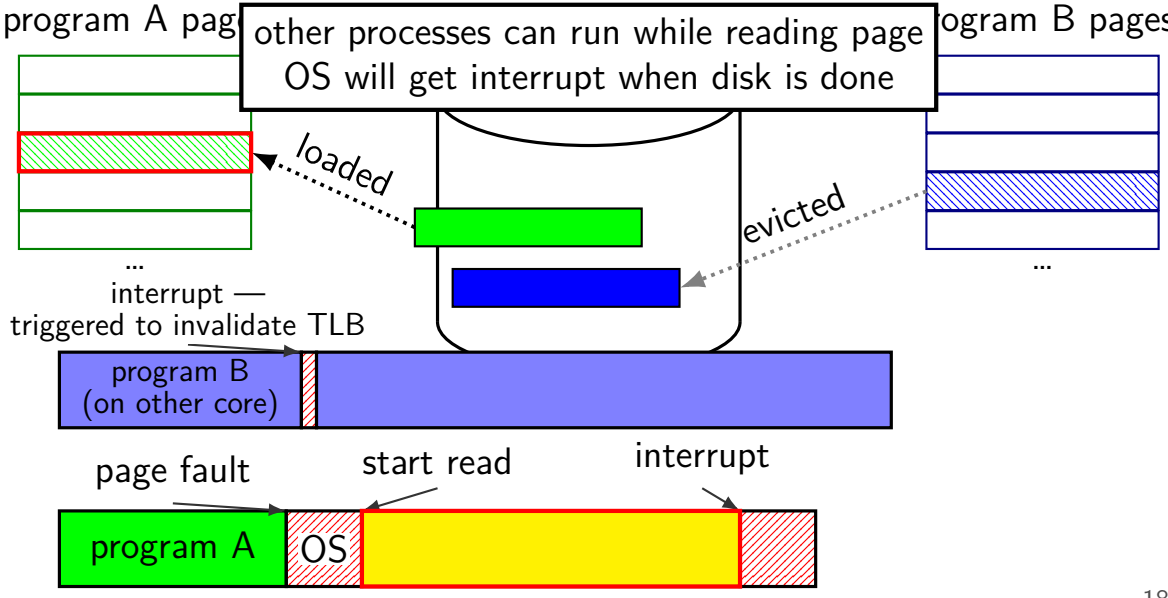
TLB shootdown



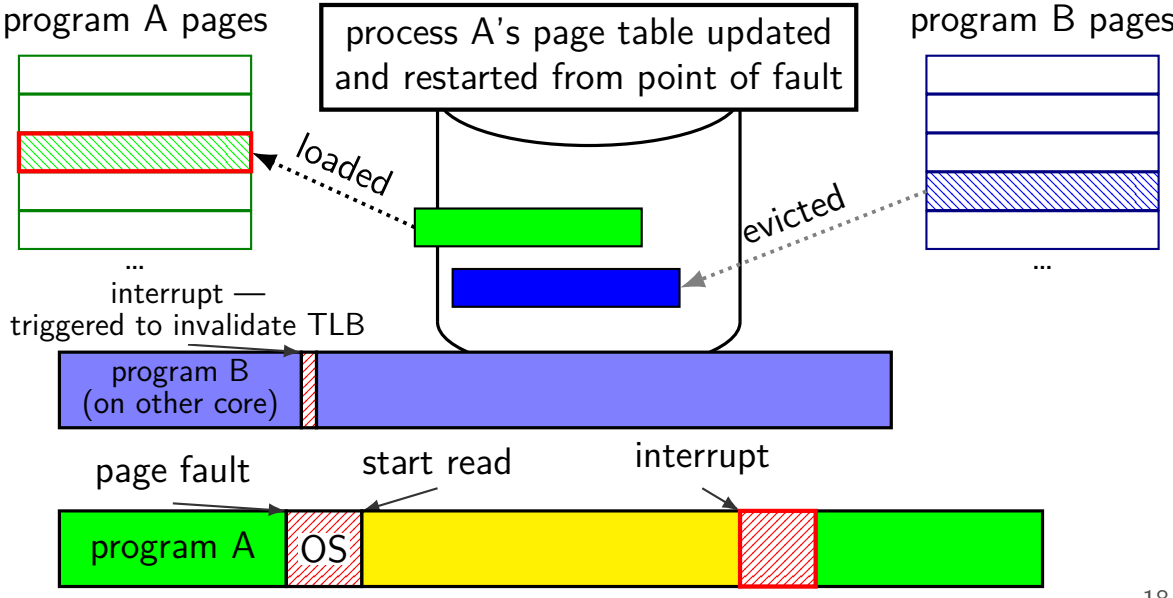
TLB shootdown



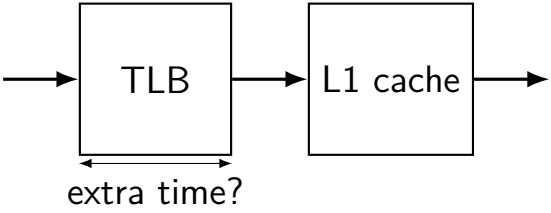
TLB shootdown



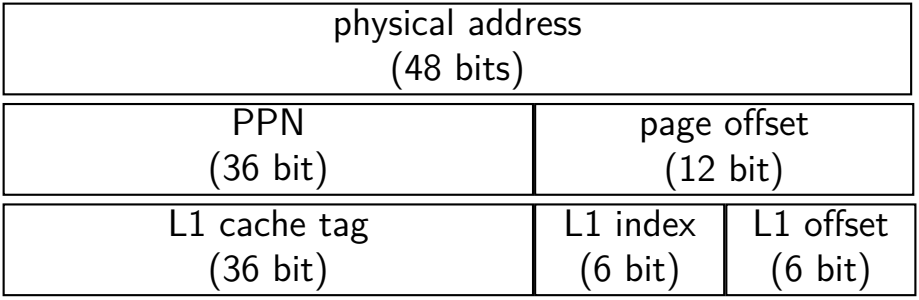
TLB shootdown



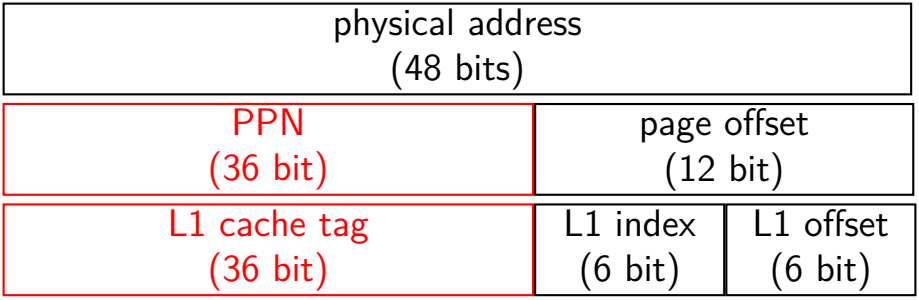
TLBs and performance



L1 caches and page numbers (Intel Skylake)



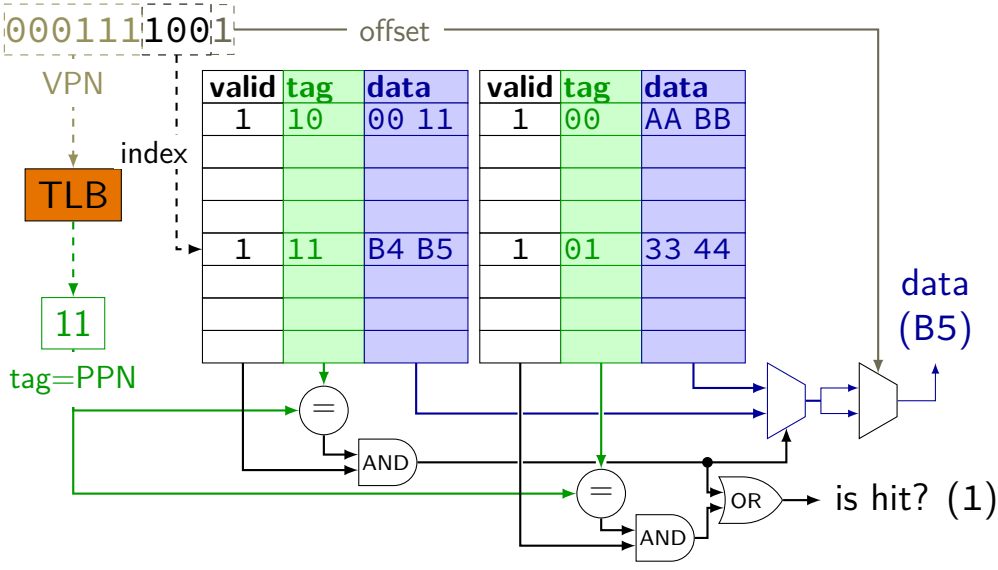
L1 caches and page numbers (Intel Skylake)



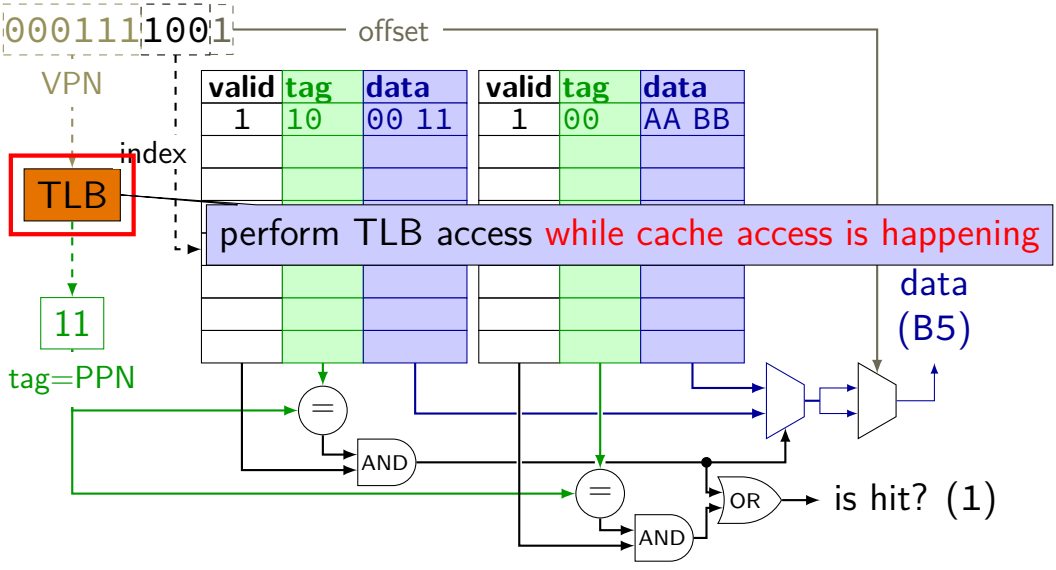
not a coincidence

why did Intel make this decision?

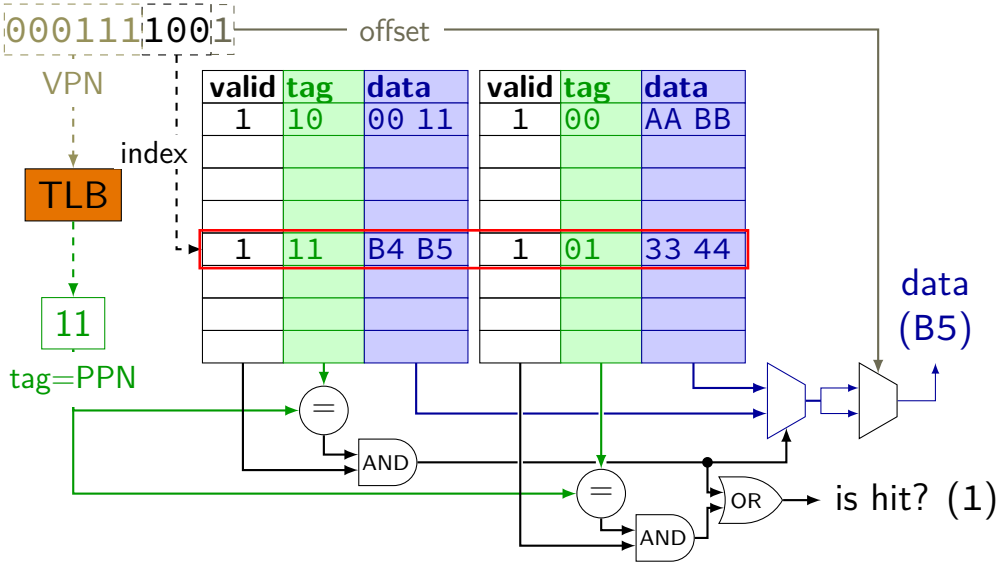
overlapping TLB and cache access



overlapping TLB and cache access



overlapping TLB and cache access



virtually-indexed, physically-tagged

called virtually-indexed, physically-tagged cache

requirement: index contained entirely in page offset

- do not need to do translation to start cache access

tag overlaps with PPN

- example: tag=PPN
- (but tag could include part of page offset, too)

do TLB access while retrieving cache set

most common design in current processors

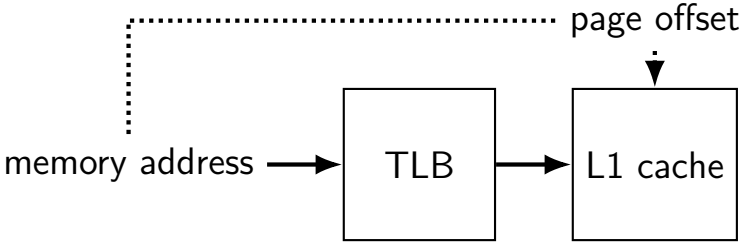
reason for highly associative (e.g. 8-way) L1 caches

physical caches

so far: caches use physical addresses:

means cache lookup can't complete without TLB

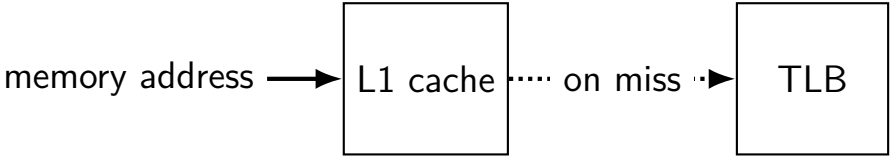
(and can't start without index from physical address)



virtual indexing (rarely used)

alternate option: have caches hold virtual addresses

alternate, rarely chosen design



faster lookup, but some things more complicated:

- need to invalidate caches on page table changes
- need to deal with multiple PTEs for some physical page ("aliasing")

address splitting

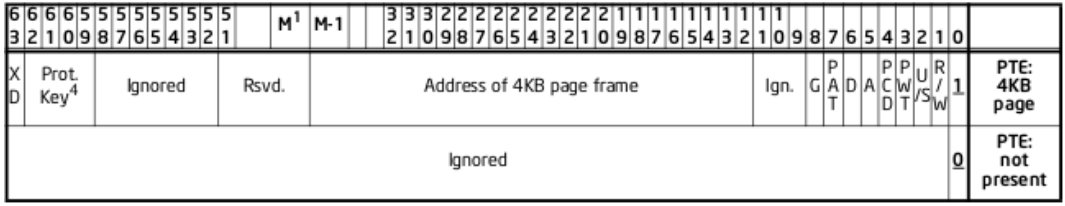
16-bit virtual addresses

64-byte pages

256B, 8-way L1 cache with 16B blocks

can TLB and cache access overlap?

x86-64 page table entries (1)



present = valid

R/W = writes allowed?

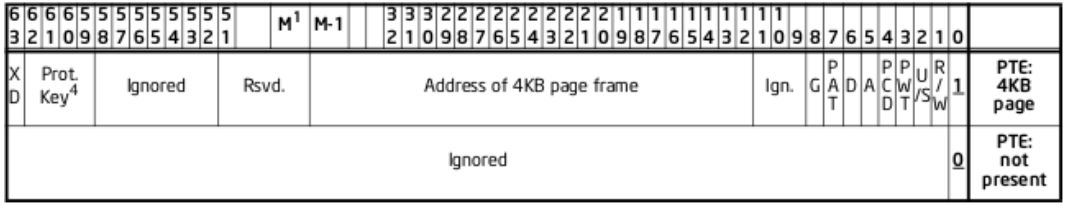
U/S = kernel-only? (“user/supervisor”)

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

x86-64 page table entries (1)



present = valid

R/W = writes allowed?

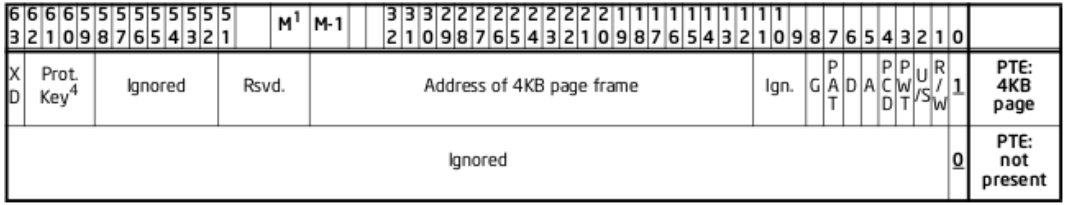
U/S = kernel-only? ("user/supervisor")

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = helps support replacement policies for swapping

x86-64 page table entries (1)



present = valid

R/W = writes allowed?

U/S = kernel-only? ("user/supervisor")

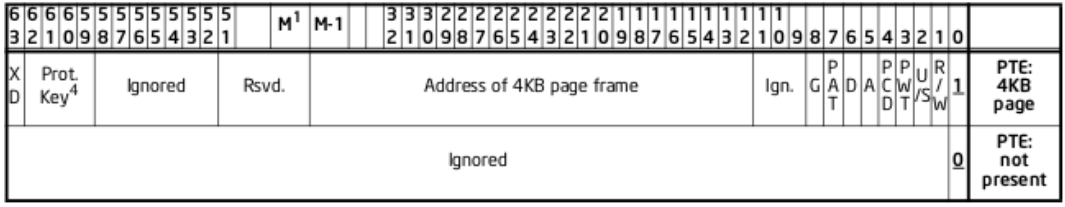
XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

helps support writeback policy for swapping

x86-64 page table entries (2)

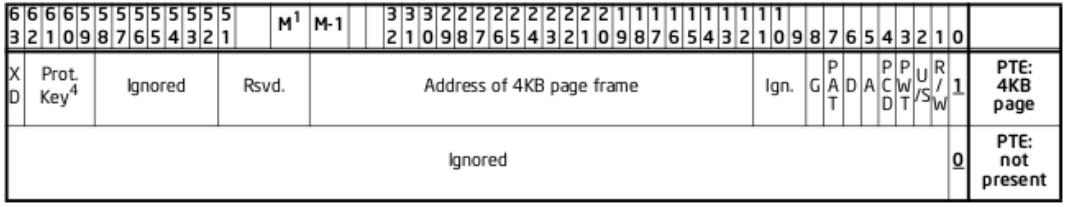


G = global? (shared between all page tables)

PWT, PCD, PAT = control how caches work when accessing physical page:

- can disable using the cache entirely
- can disable write-back (use write-through instead)
- multicore-related cache settings
- (and some other settings)

x86-64 page table entries (2)

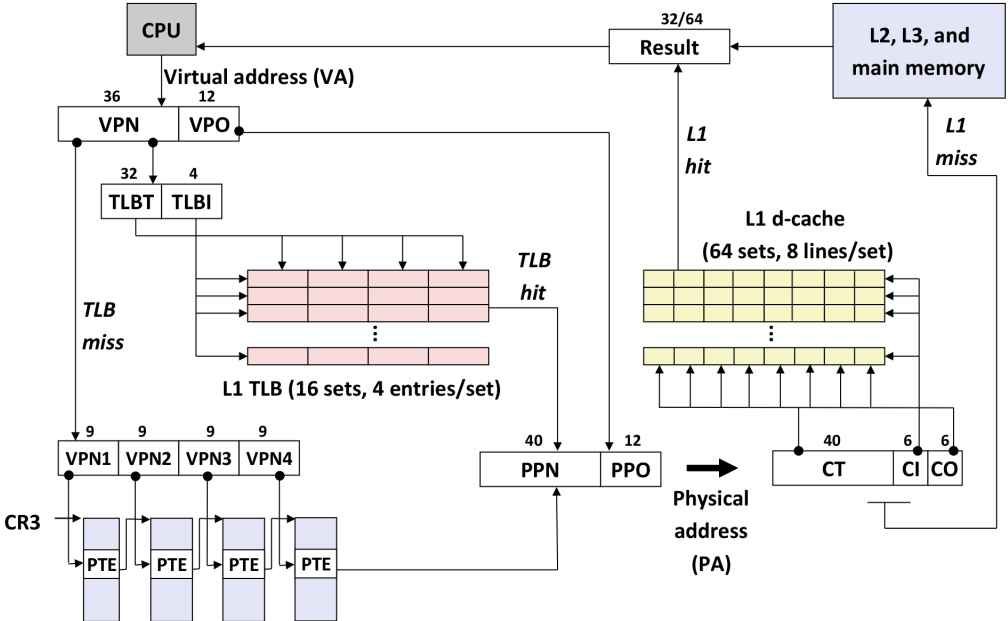


G = global? (shared between all page tables)

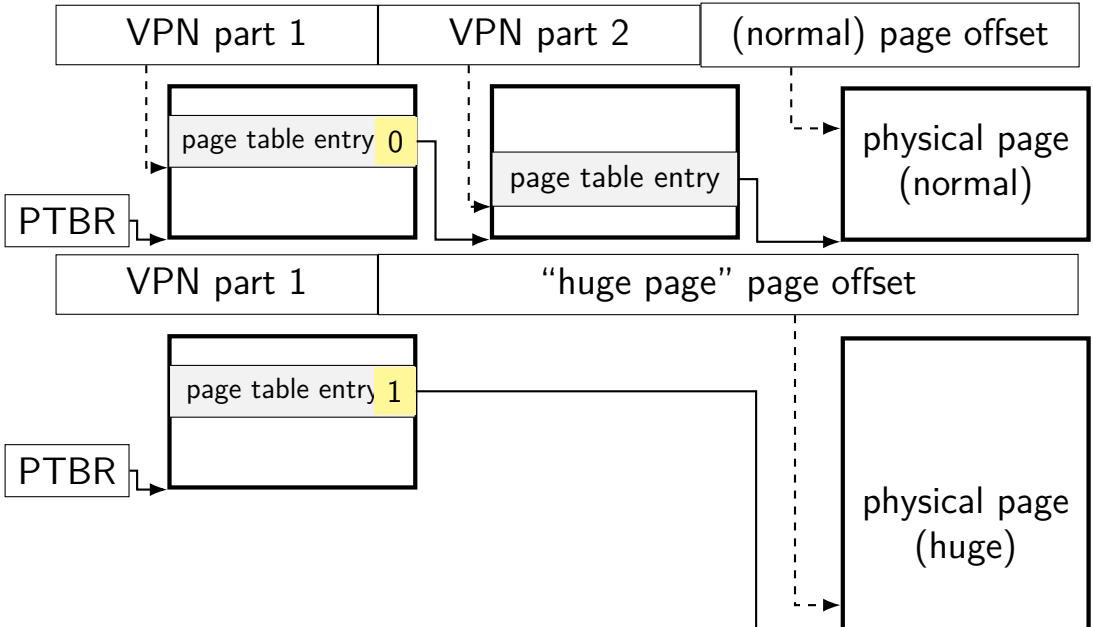
P CPU won't evict TLB entries on most page table base registers changes

can disable using the cache entirely
can disable write-back (use write-through instead)
multicore-related cache settings
(and some other settings)

book's diagram



“huge pages”


VPN part 1
VPN part 2
(normal) page offset
page table entry 0
page table entry
physical page (normal)
PTBR
VPN part 1
“huge page” page offset
page table entry 1
PTBR
physical page (huge)

big pages on x86-64

option for 2MB or 1GB pages instead of 4KB pages

first, second, third-level page table entries can point to either

- next page table (normal case), or
- a "huge page"

changes to TLB needed

processes can have mix of huge and normal apges

why big pages?

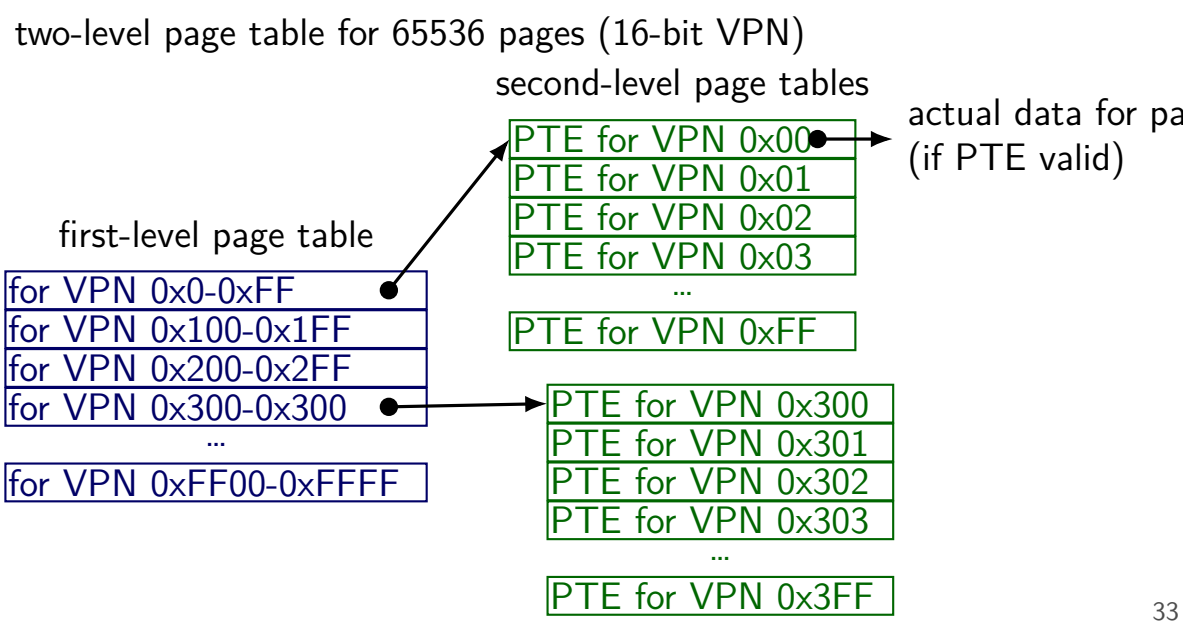
TLB misses can create same sort of problems as cache misses

can do cache blocking to help with TLB misses but...

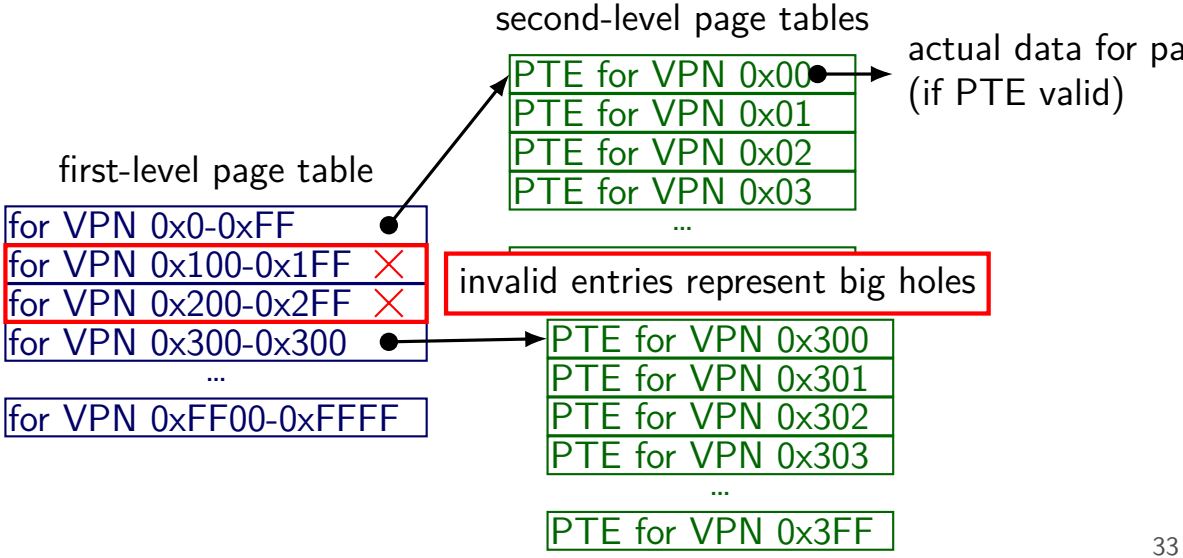
big pages are relatively easy to implement

might dramatically reduce TLB misses

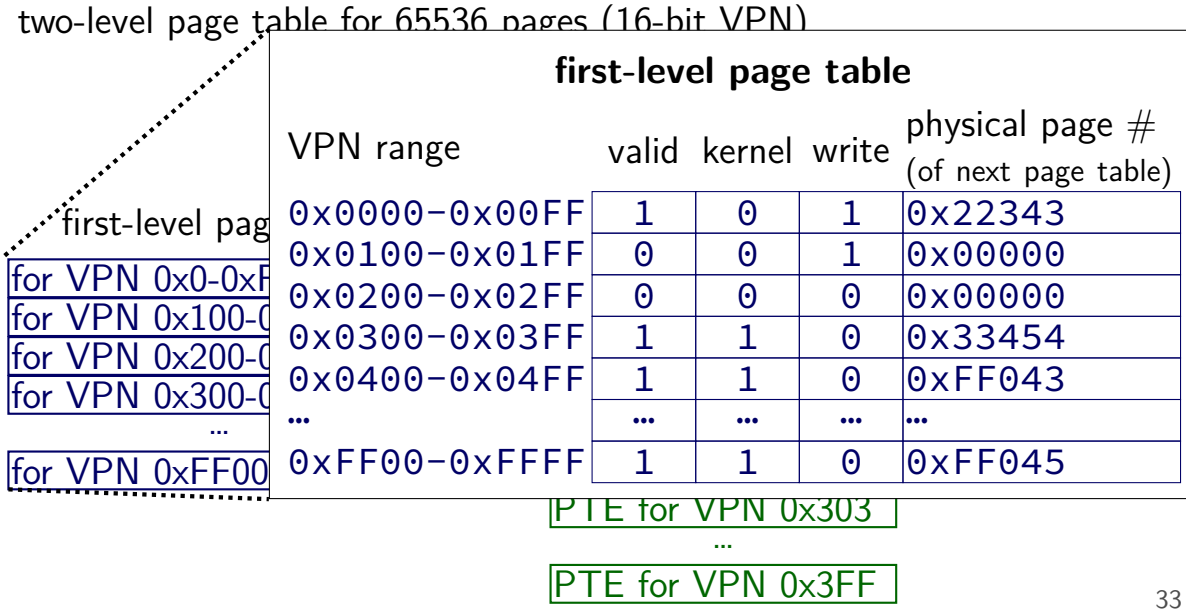
two-level page tables



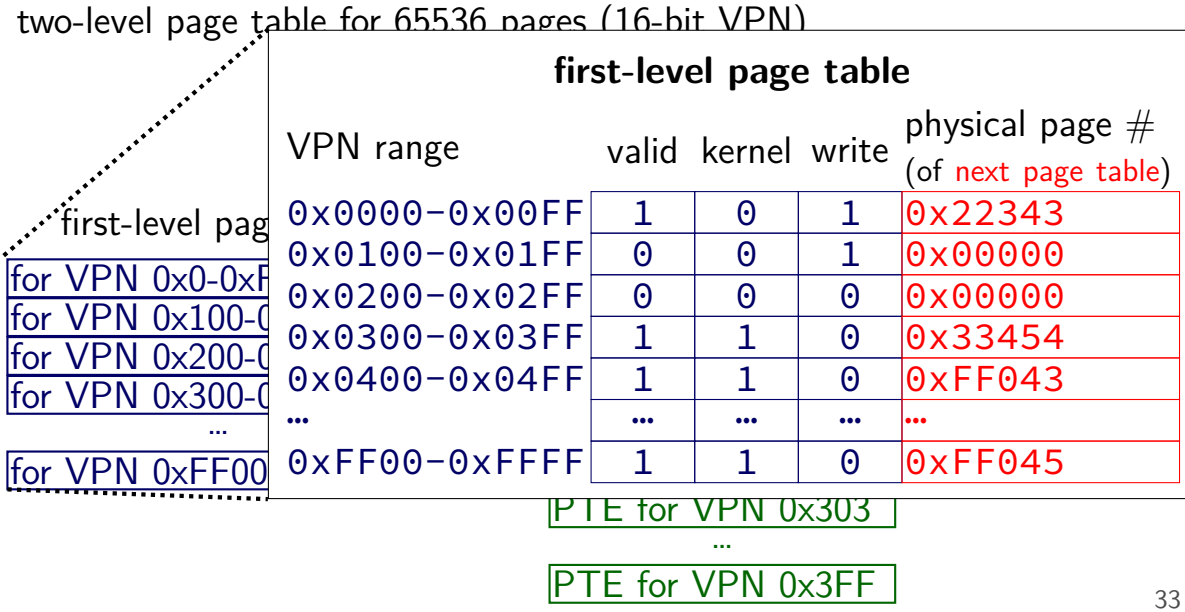
two-level page tables



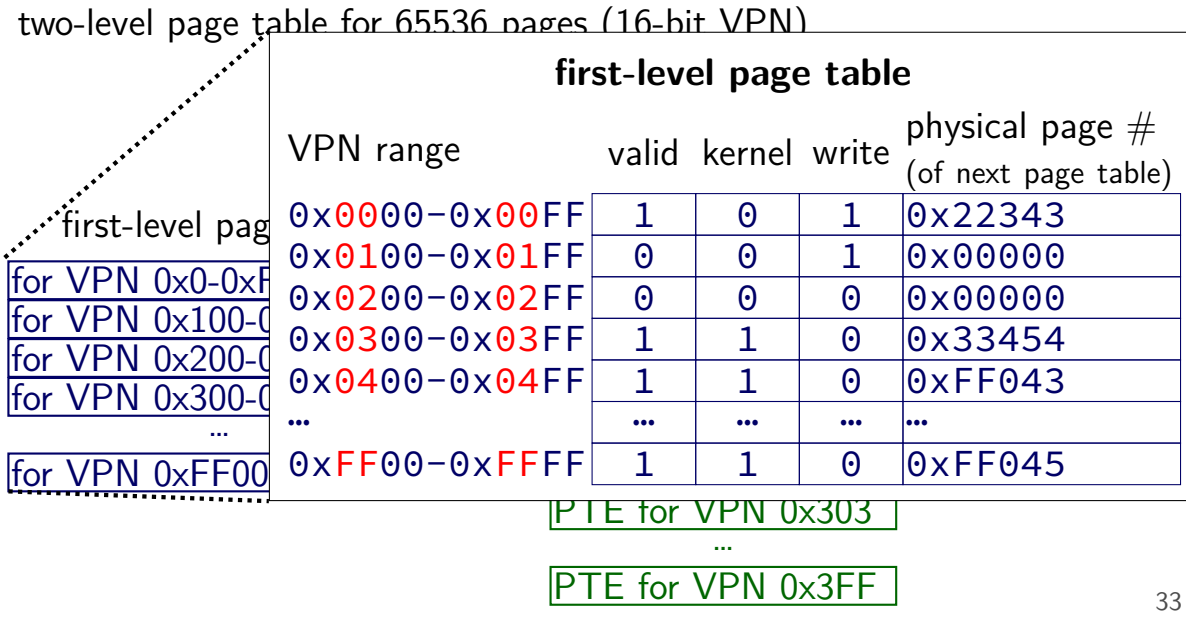
two-level page tables



two-level page tables

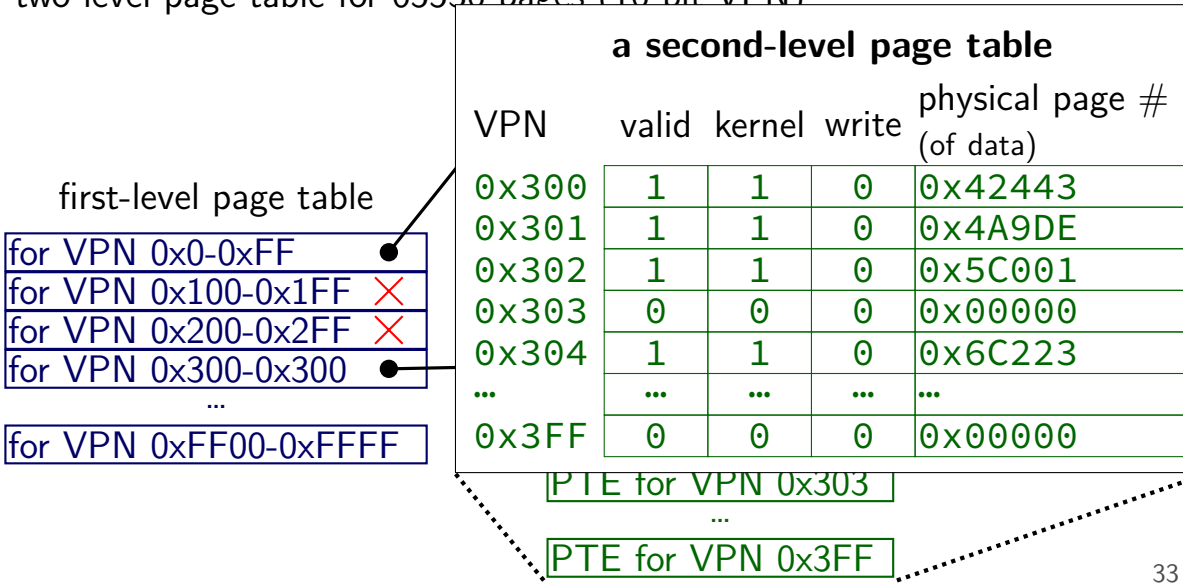


two-level page tables



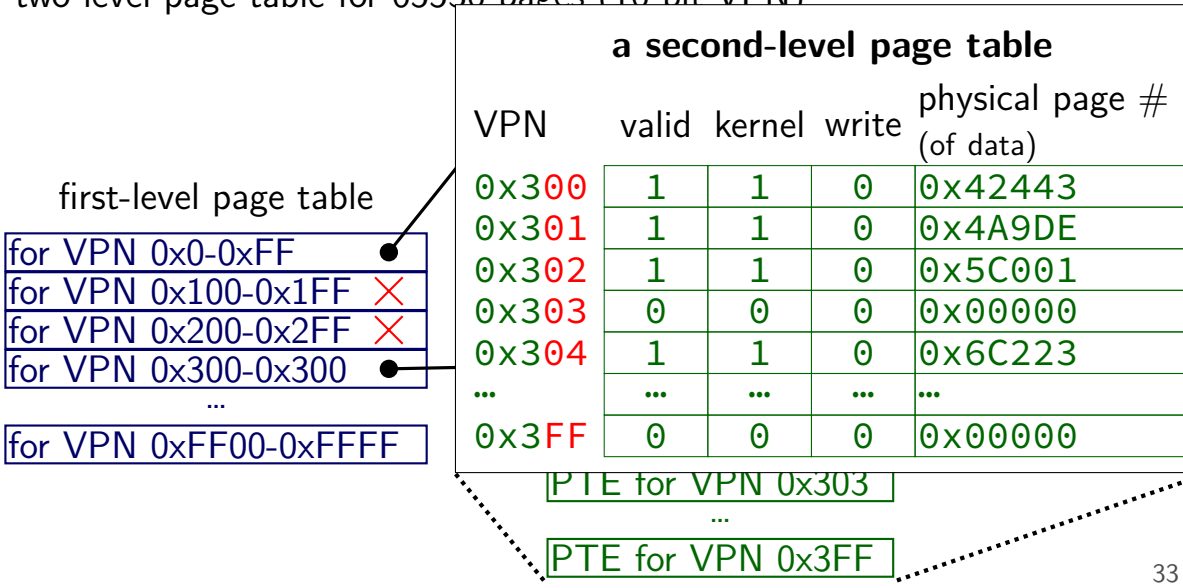
two-level page tables

two-level page table for 65536 pages (16-bit VPN)

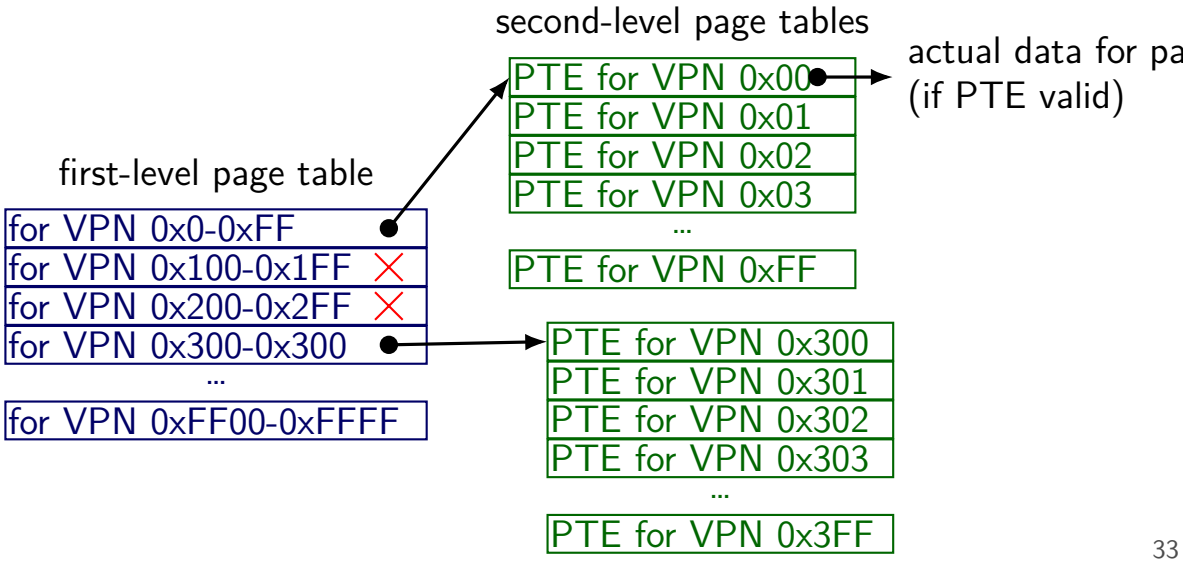


two-level page tables

two-level page table for 65536 pages (16-bit VPN)



two-level page tables

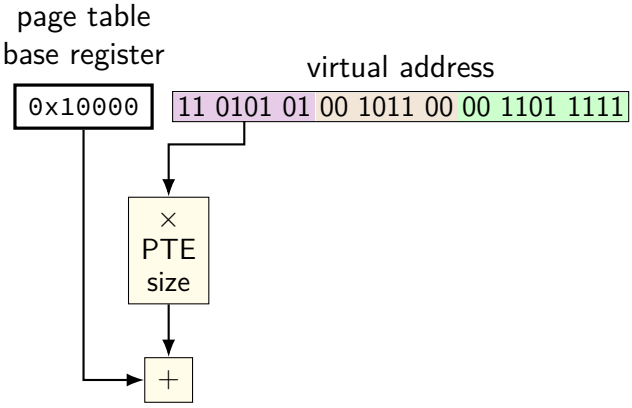


two-level page table lookup

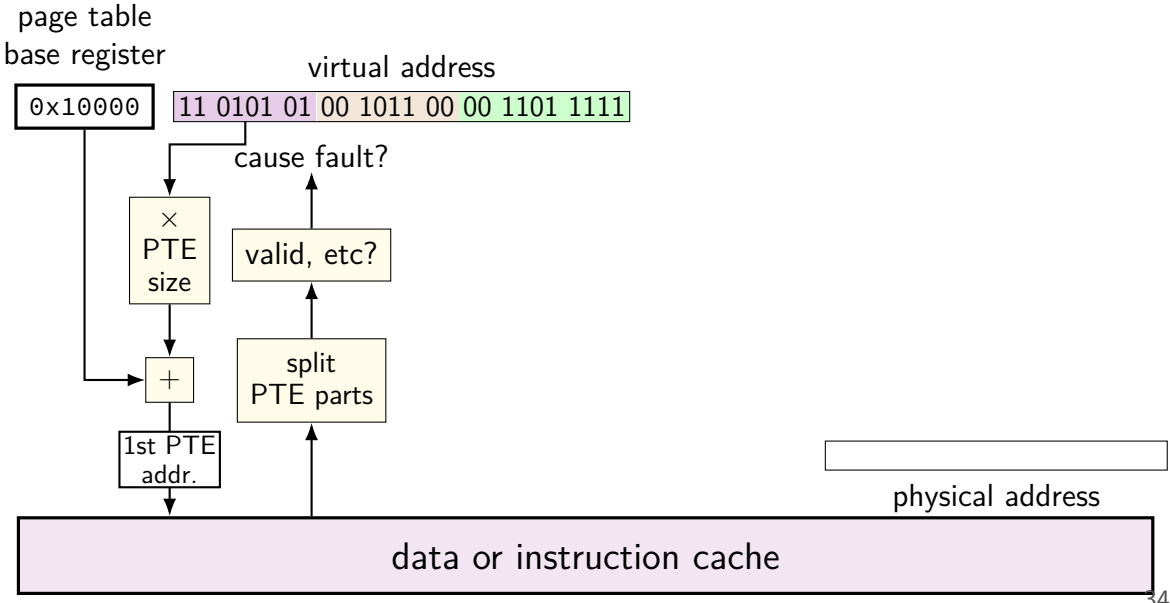
virtual address

VPN — split into two parts (one per level)

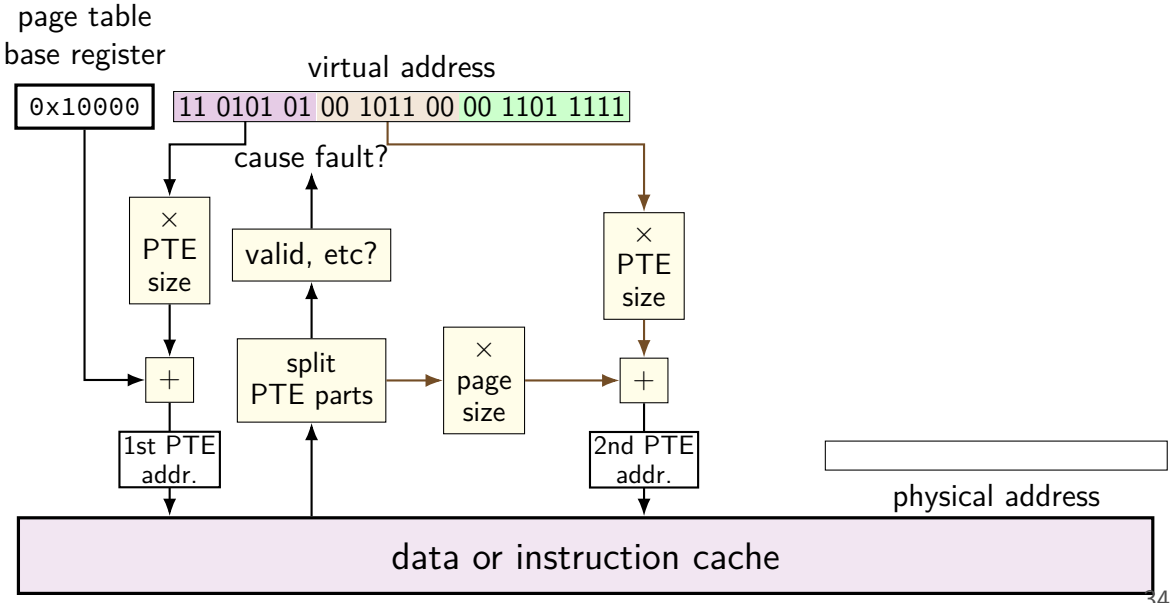
two-level page table lookup



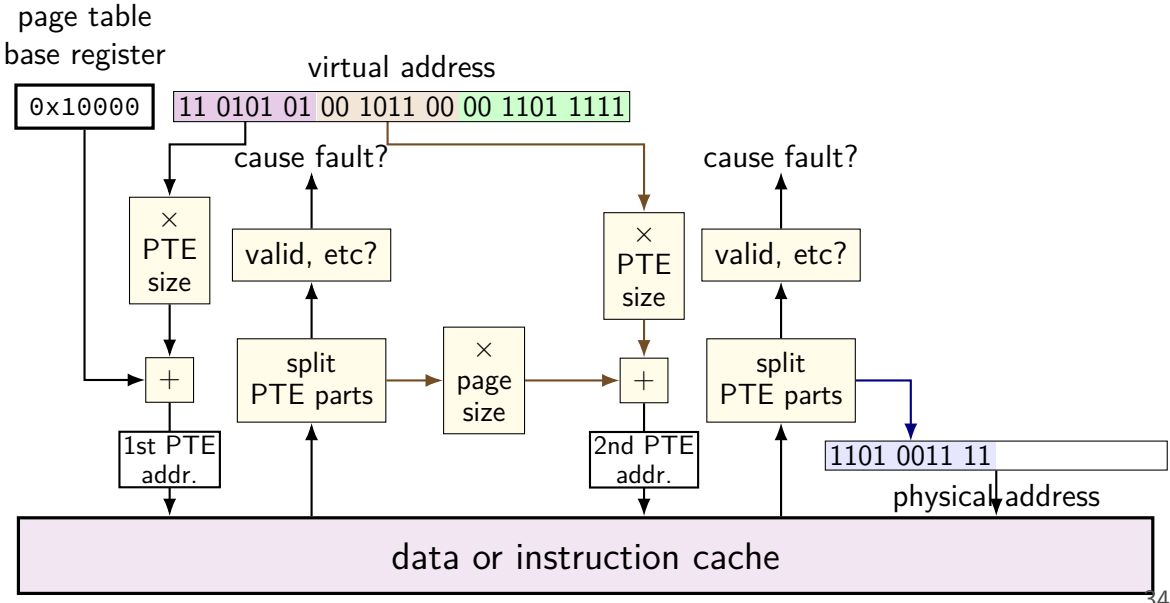
two-level page table lookup



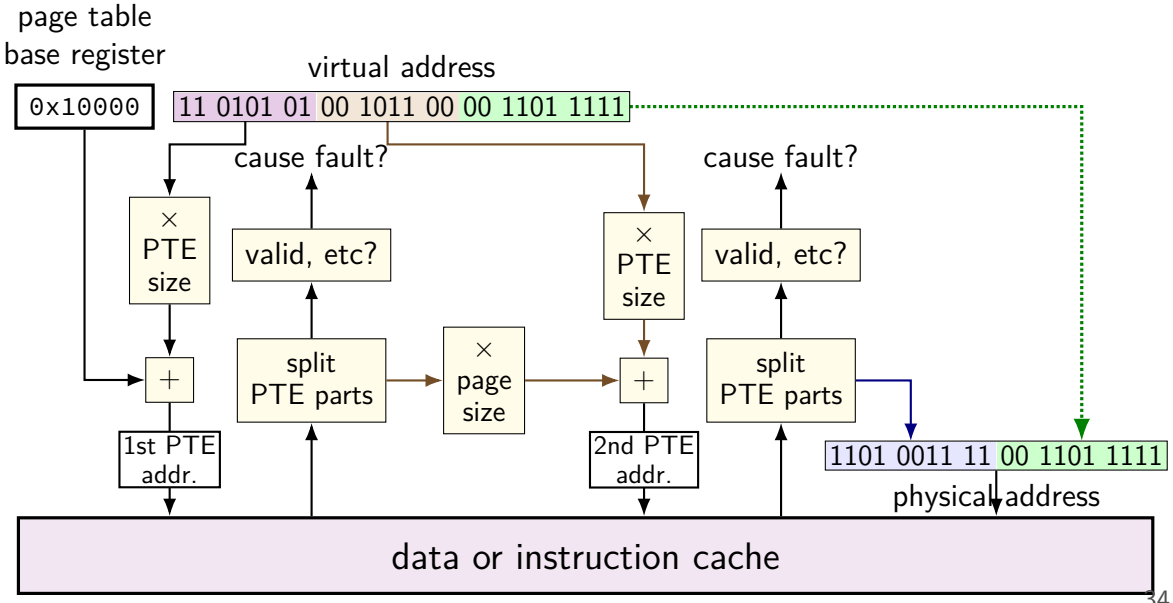
two-level page table lookup



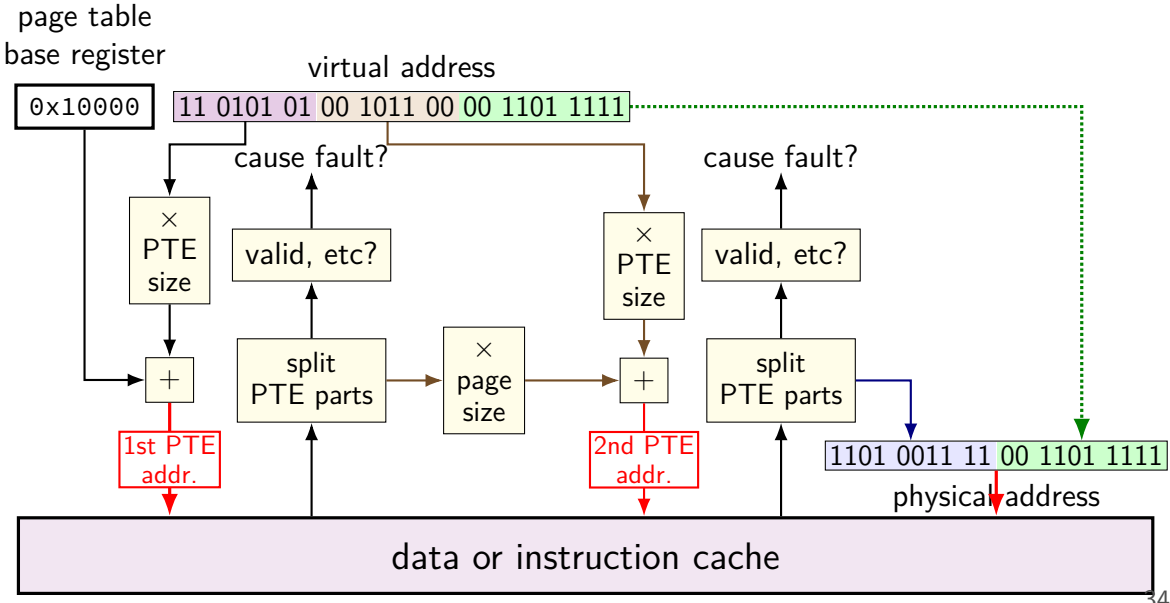
two-level page table lookup



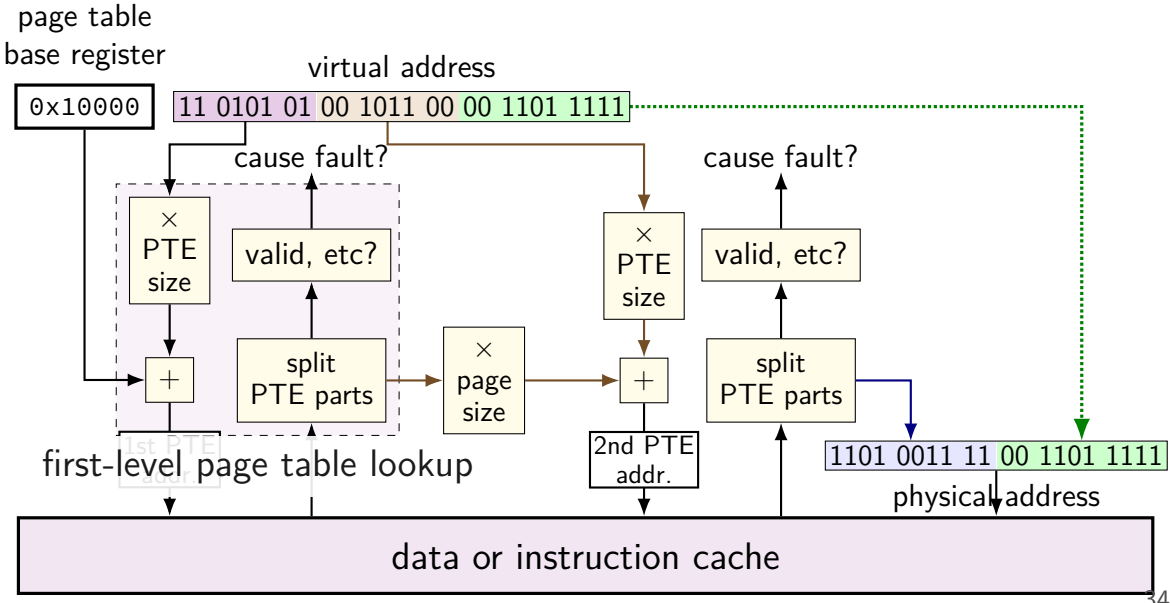
two-level page table lookup



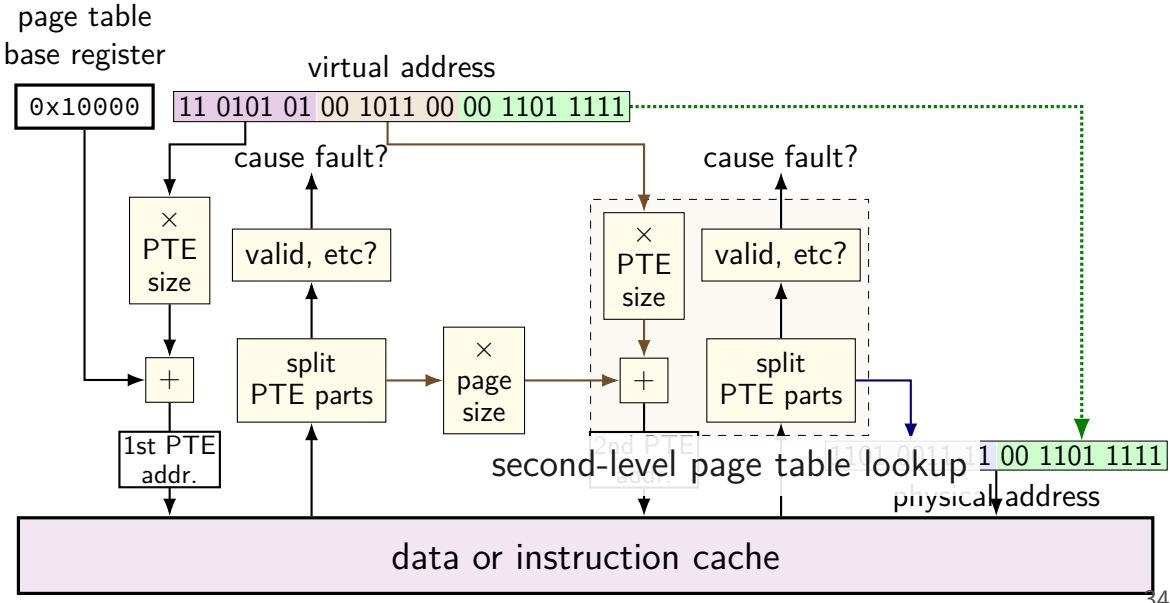
two-level page table lookup



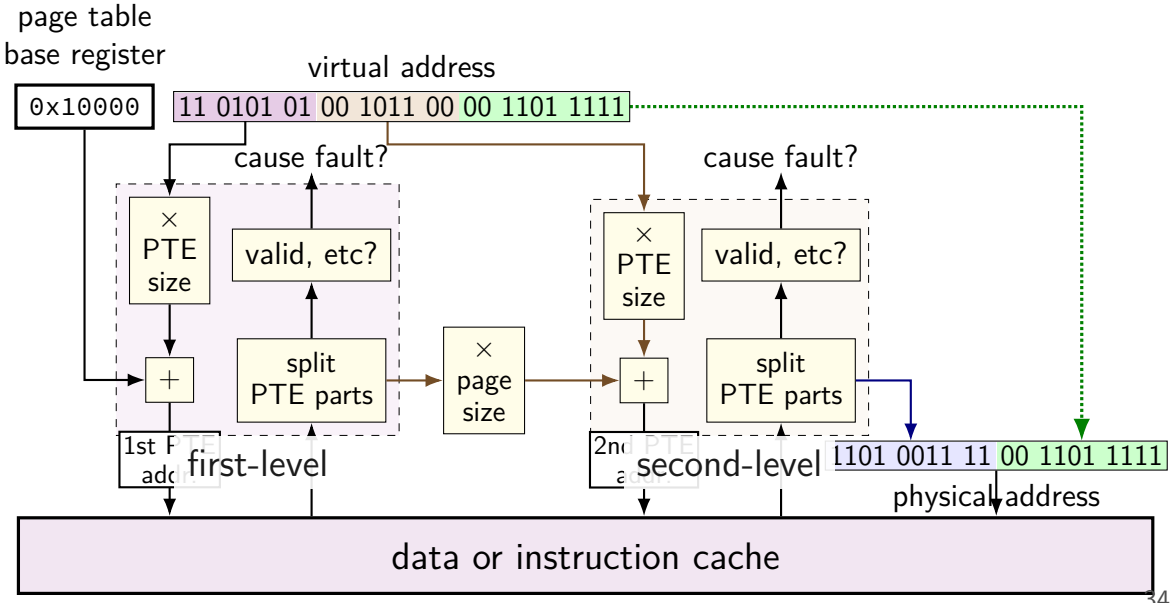
two-level page table lookup



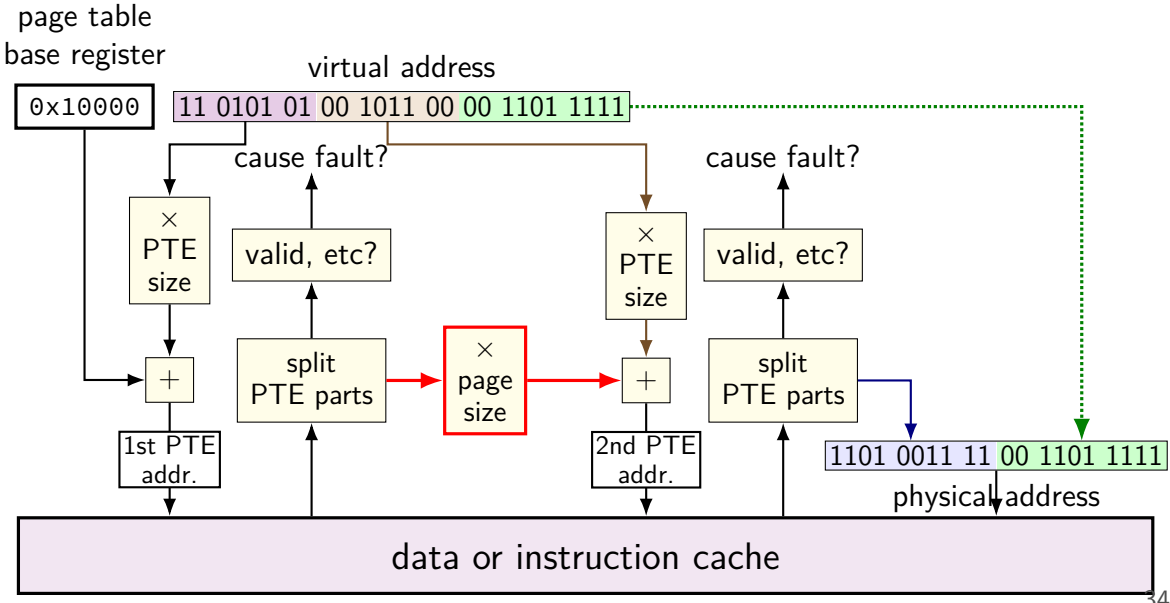
two-level page table lookup



two-level page table lookup



two-level page table lookup



two-level page table lookup