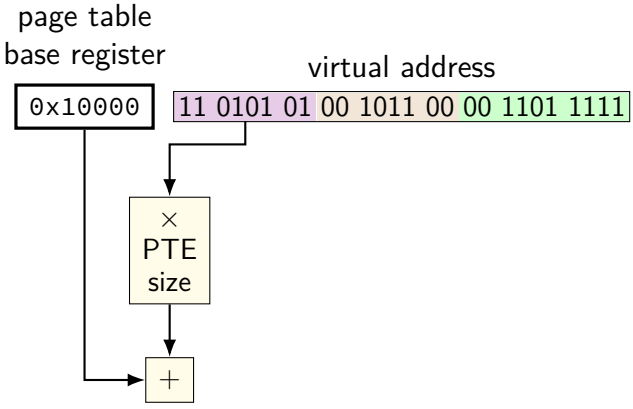
Virtual Memory

two-level page table lookup

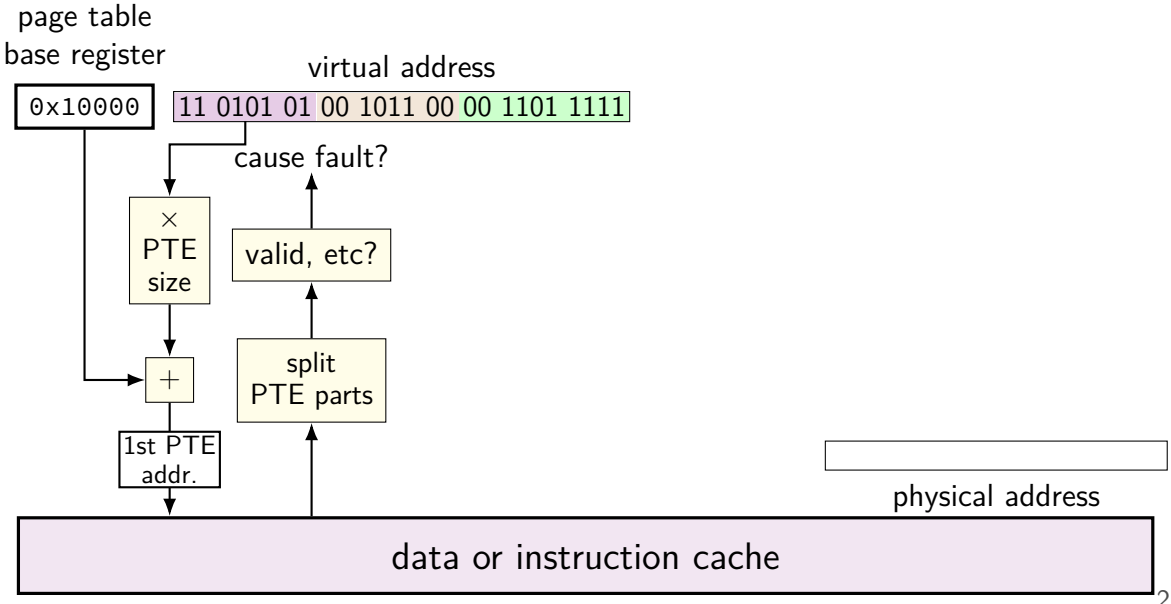
virtual address

VPN — split into two parts (one per level)

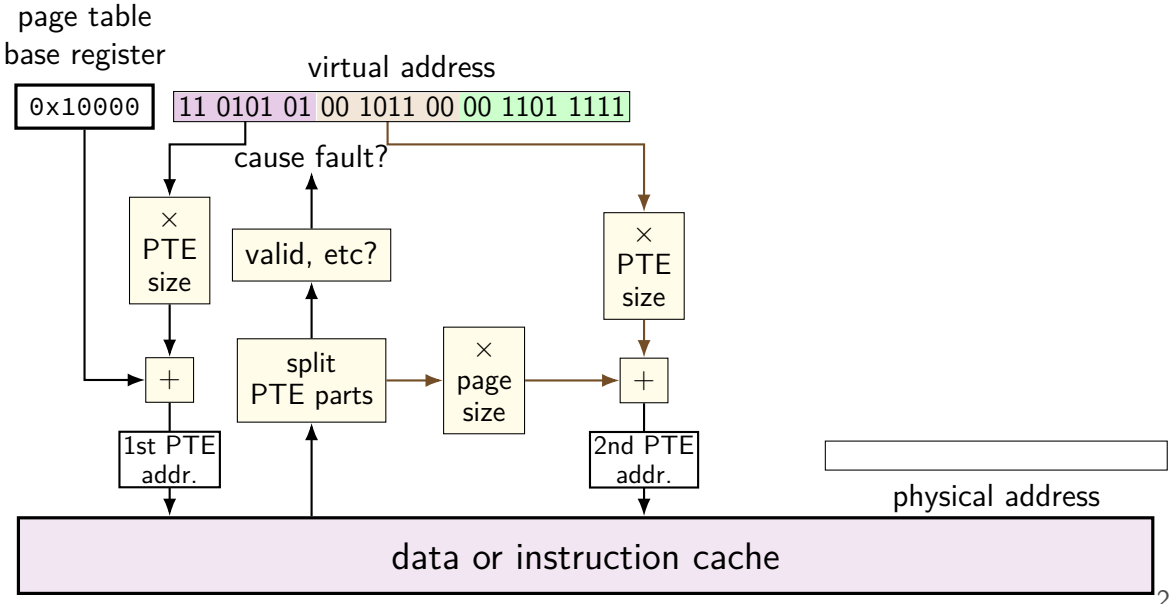
two-level page table lookup



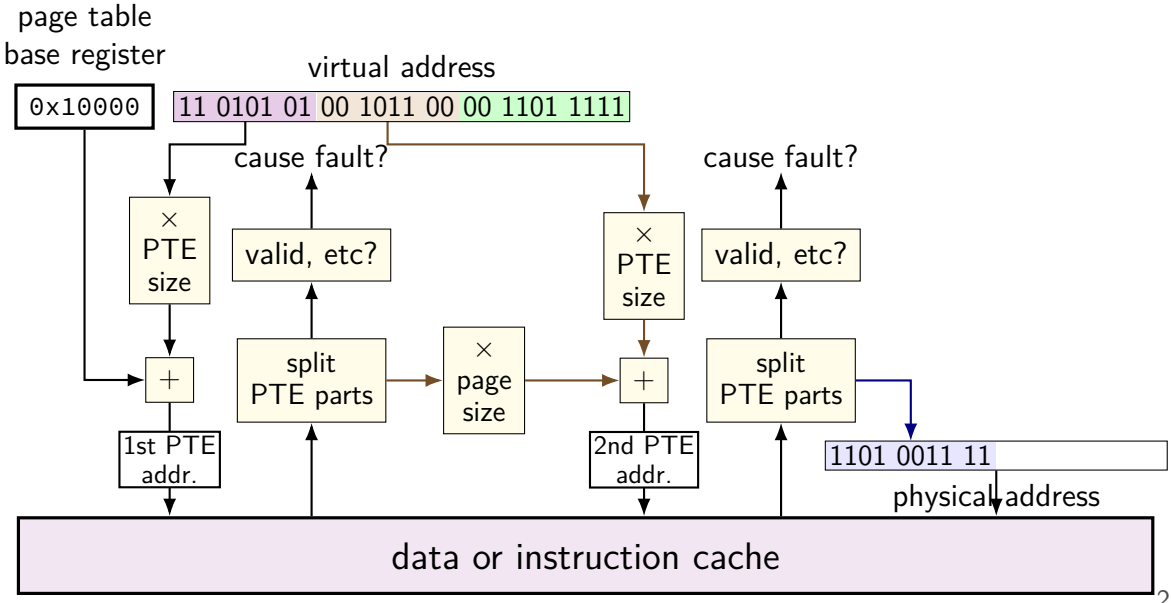
two-level page table lookup



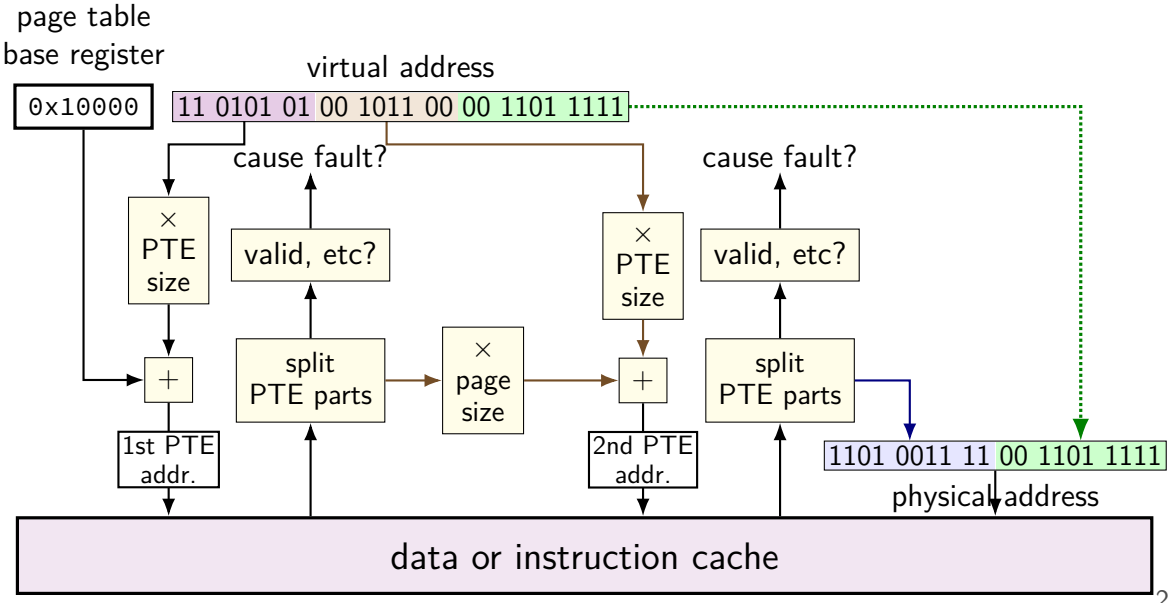
two-level page table lookup



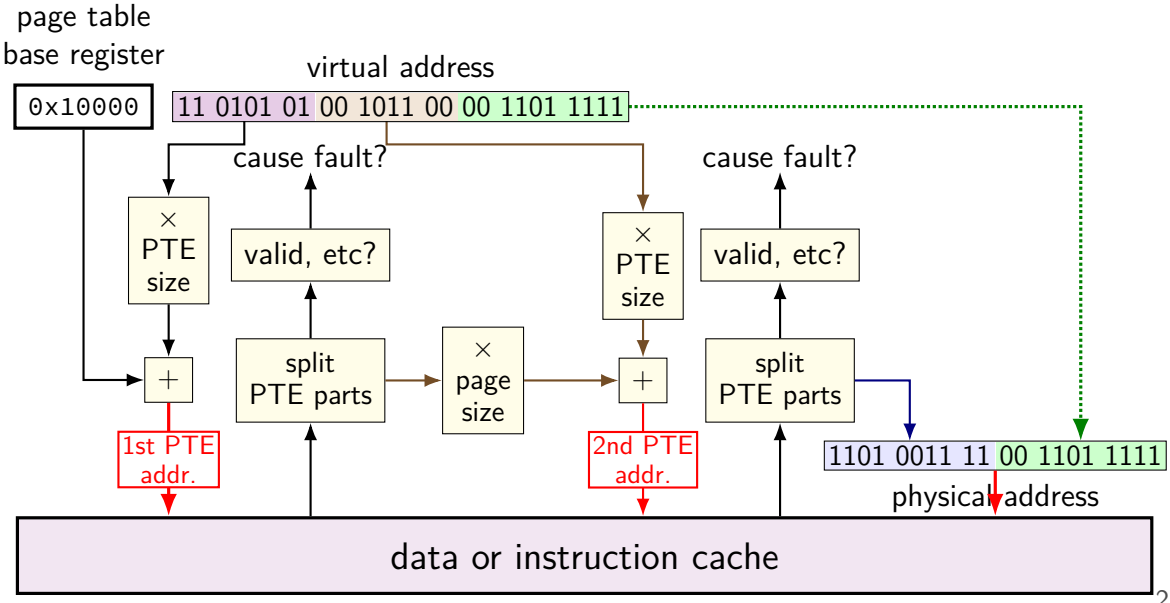
two-level page table lookup



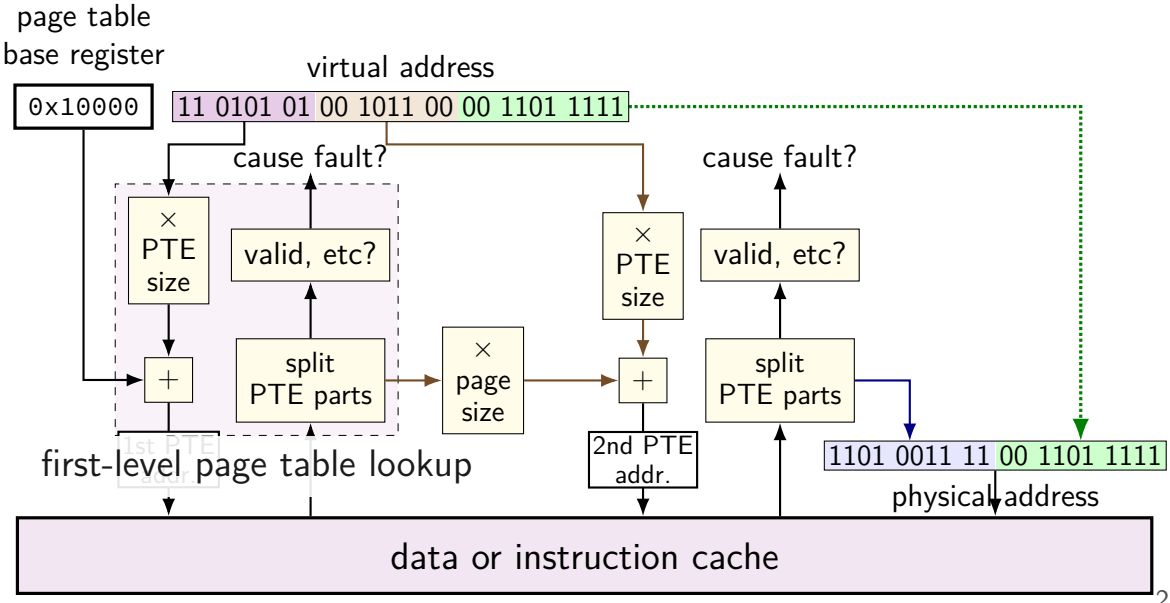
two-level page table lookup



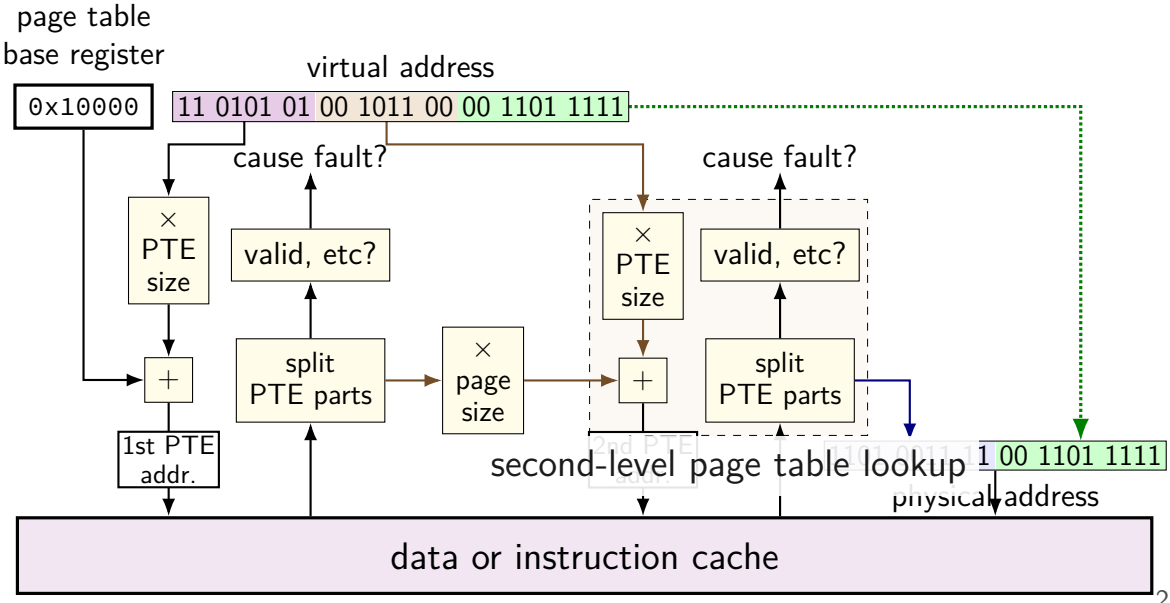
two-level page table lookup



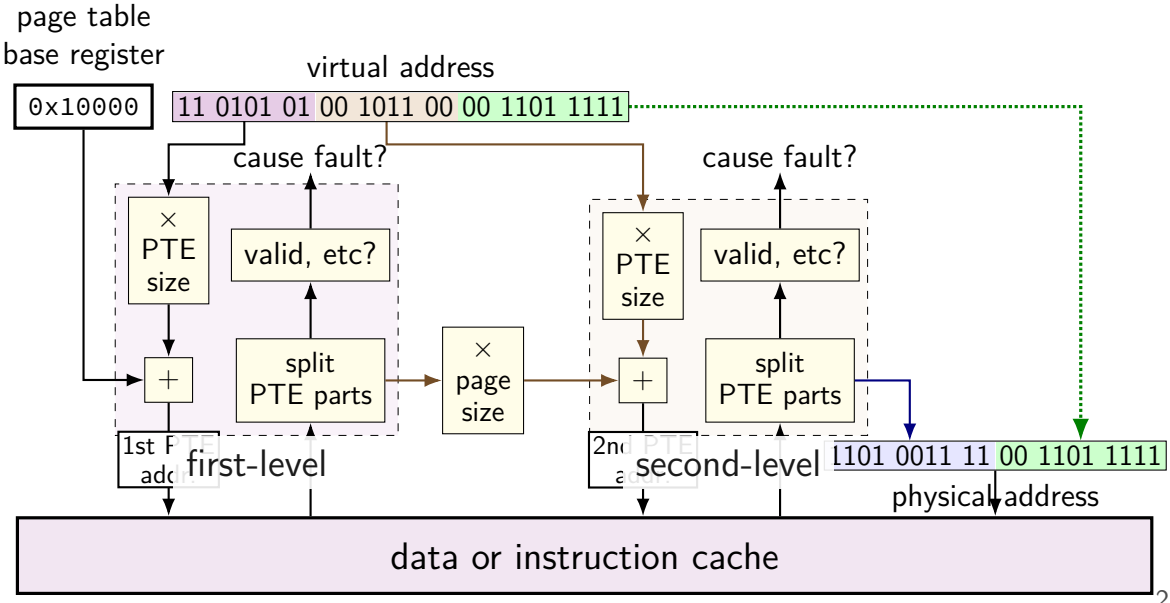
two-level page table lookup



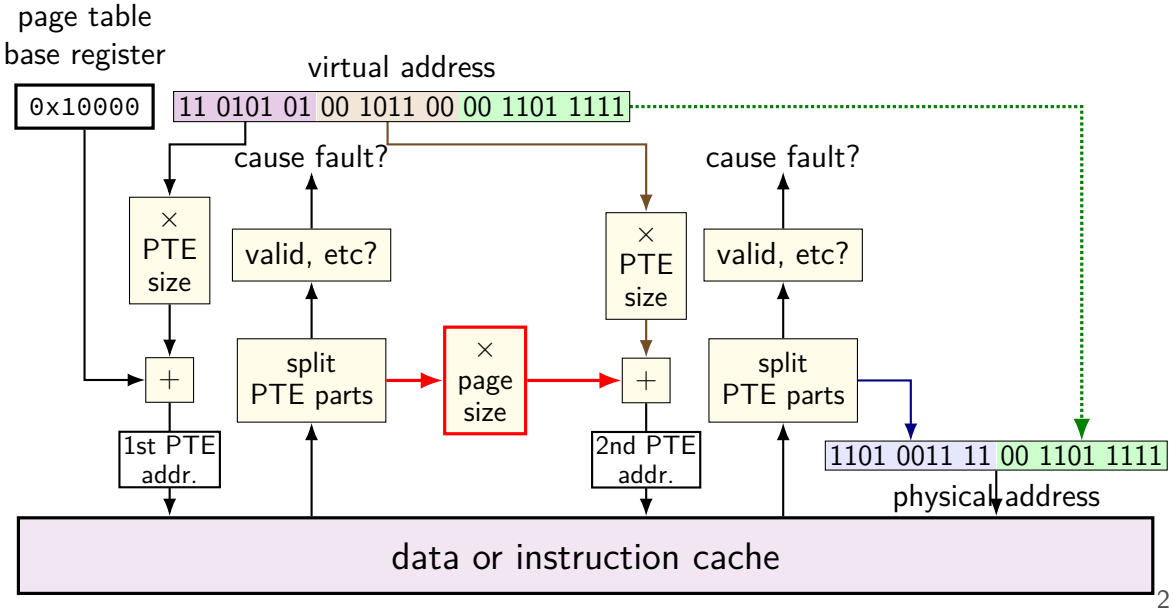
two-level page table lookup



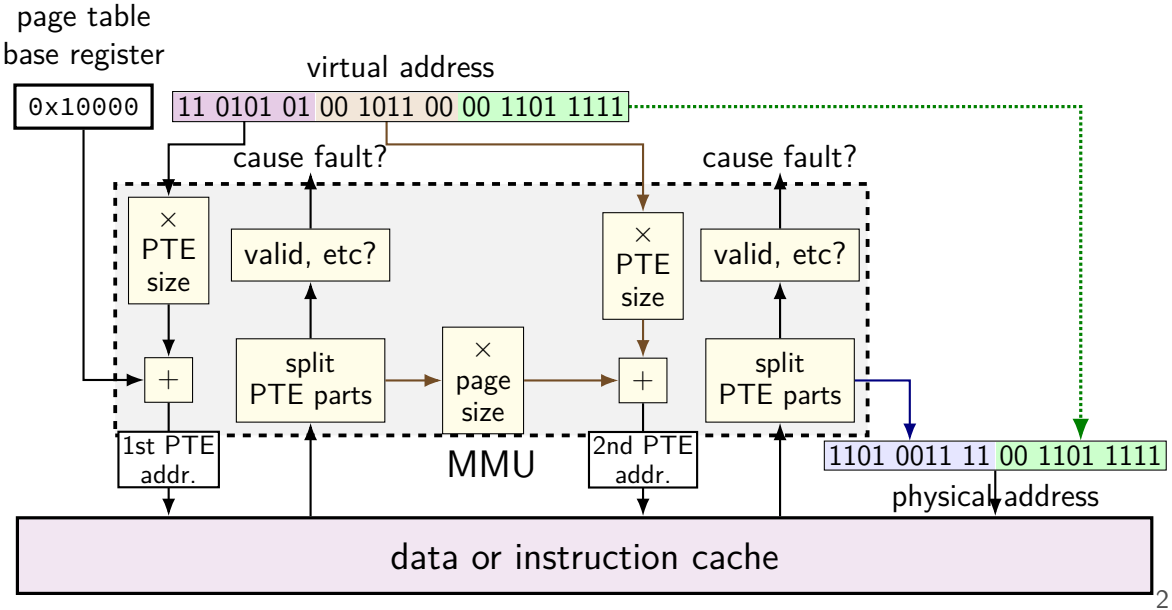
two-level page table lookup



two-level page table lookup



two-level page table lookup



2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

0x131 = 1 0011 0001

0x20 + 4 ×1 = 0x24

PTE 1 value:

0xD4 = 1101 0100

PPN 110, valid 1

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid `1`

PTE 2 addr:

`110 000` + `110` = `0x36`

PTE 2 value: `0xDB`

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid 1

PTE 2 addr:

`110 000` + `110` = `0x36`

PTE 2 value: `0xDB`

PPN `110`; valid 1

M[`110 001` (`0x31`)] = `0x0A`

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid 1

PTE 2 addr:

`110 000` + `110` = `0x36`

PTE 2 value: `0xDB`

PPN `110`; valid 1

M[`110 001` (`0x31`)] = `0x0A`

2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register `0x20`; translate virtual address `0x131`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x131` = `1 0011 0001`

`0x20` + 4 × 1 = `0x24`

PTE 1 value:

`0xD4` = `1101 0100`

PPN `110`, valid 1

PTE 2 addr:

`110 000` + `110` = `0x36`

PTE 2 value: `0xDB`

PPN `110`; valid 1

M[`110 001` (`0x31`)] = `0x0A`

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register `0x08`; translate virtual address `0x0FB`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register `0x08`; translate virtual address `0x0FB`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x0FB` = `011 111 011`

PTE 1 addr:

`0x08` + 3 ×1 = `0x0B`

PTE 1: `0xBB` at `0x0B`

PTE 1: PPN `101` (5) valid `1`

PTE 2 addr:

`101 000` + `111` = `0x2F`

PTE 2: `0xF0` at `0x2F`

PTE 2: PPN `111` (7) valid `1`

`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register `0x08`; translate virtual address `0x0FB`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x0FB` = `011 111 011`

PTE 1 addr:

`0x08` + 3 ×1 = `0x0B`

PTE 1: `0xBB` at `0x0B`

PTE 1: PPN `101` (5) valid `1`

PTE 2 addr:

`101 000` + `111` = `0x2F`

PTE 2: `0xF0` at `0x2F`

PTE 2: PPN `111` (7) valid `1`

`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register `0x08`; translate virtual address `0x0FB`

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

`0x0FB` = `011 111 011`

PTE 1 addr:

`0x08` + 3 ×1 = `0x0B`

PTE 1: `0xBB` at `0x0B`

PTE 1: PPN `101` (5) valid `1`

PTE 2 addr:

`101 000` + `111` = `0x2F`

PTE 2: `0xF0` at `0x2F`

PTE 2: PPN `111` (7) valid `1`

`111 011` = `0x3B` → `0x0C`

2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;

page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00–3	00 11 22 33
0x04–7	44 55 66 77
0x08–B	88 99 AA BB
0x0C–F	CC DD EE FF
0x10–3	1A 2A 3A 4A
0x14–7	1B 2B 3B 4B
0x18–B	1C 2C 3C 4C
0x1C–F	1C 2C 3C 4C

physical addresses	bytes
0x20–3	D0 D1 D2 D3
0x24–7	D4 D5 D6 D7
0x28–B	89 9A AB BC
0x2C–F	CD DE EF F0
0x30–3	BA 0A BA 0A
0x34–7	DB 0B DB 0B
0x38–B	EC 0C EC 0C
0x3C–F	FC 0C FC 0C

0x0FB = 011 111 011

PTE 1 addr:

0x08 + 3 ×1 = 0x0B

PTE 1: 0xBB at 0x0B

PTE 1: PPN 101 (5) valid 1

PTE 2 addr:

101 000 + 111 = 0x2F

PTE 2: 0xF0 at 0x2F

PTE 2: PPN 111 (7) valid 1

111 011 = 0x3B → 0x0C

current x86-64 page tables

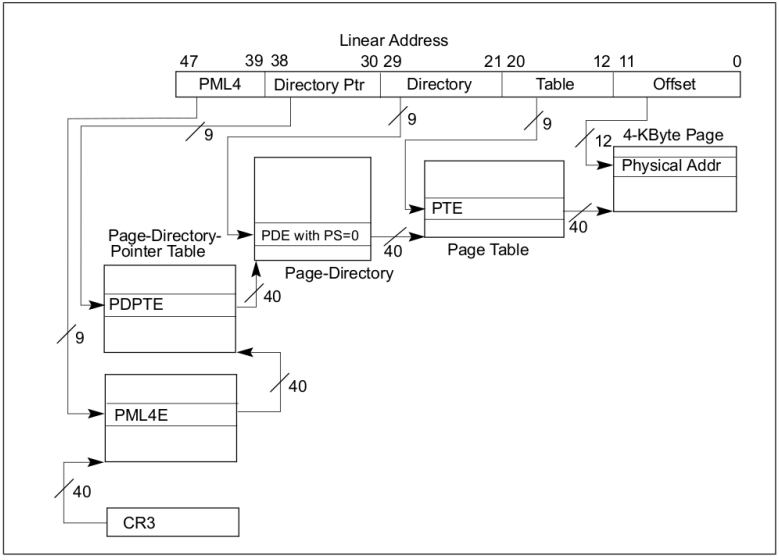


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

current x86-64 page tables

4-level page table

512 PTEs of 8 bytes each for each page table

choice: exactly one page per page table

allows OS to allocate new page table space in one page units

(just like program memory)

page table space exercise (1)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly one page allocated

how much space for page tables?

page table space exercise (1)

4-level page table

512 PTEs of 8 bytes each for each page table

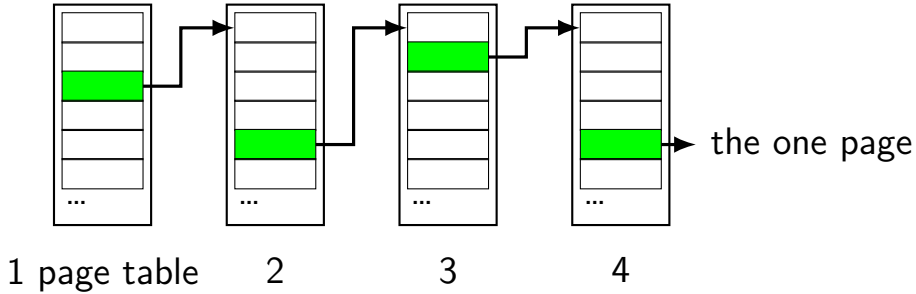
suppose a process has exactly one page allocated

how much space for page tables?

1 page at each level (4KB each)

exactly one valid entry in each of them

page table space exercise (1)



4 page tables at 1 page/page table
plus 1 page of data
5 pages total

page table space exercise (2)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly two pages allocated:

one at address 0x0, one at address `0x20000000000`

how much space for page tables?

page table space exercise (2)

4-level page table

512 PTEs of 8 bytes each for each page table

suppose a process has exactly two pages allocated:
one at address 0x0, one at address `0x20000000000`

how much space for page tables?

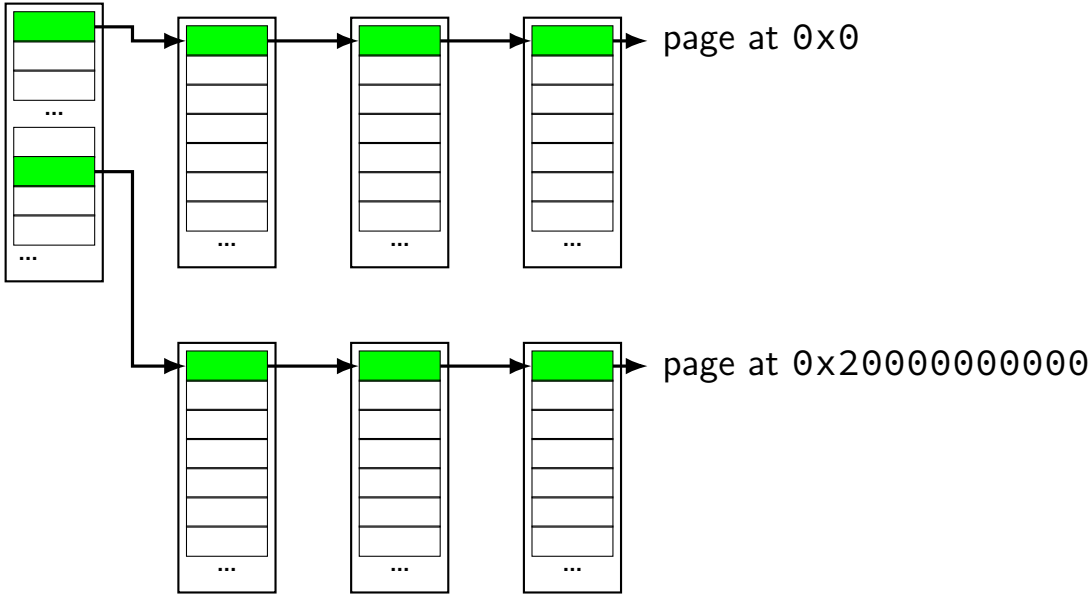
1 shared first-level PT, with two valid entries

two second-level PTs, each with one valid entry

two third-level PTs, each with one valid entry

two fourth-level PTs, each with one valid entry

page table space exercise (2)



cache accesses and multi-level PTs

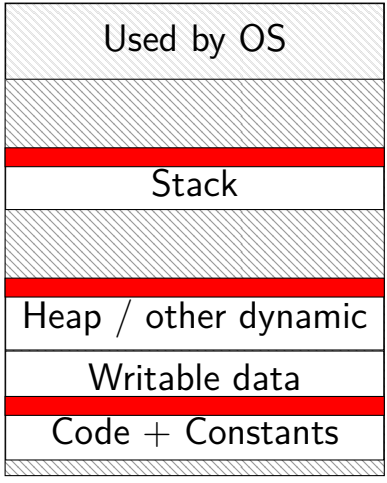
four-level page tables — four cache accesses per memory access

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

cache translated addresses

called a **TLB** (translation lookaside buffer)
small set-associative hardware cache in MMU
maps virtual page numbers to physical page numbers
contains complete page table entries for small number of pages

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

cache translated addresses

called a **TLB** (translation lookaside buffer)
small set-associative hardware cache in MMU
maps virtual page numbers to physical page numbers
contains complete page table entries for small number of pages

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

cache translated addresses

called a **TLB** (translation lookaside buffer)
small set-associative hardware cache in MMU
maps virtual page numbers to physical page numbers
contains complete page table entries for small number of pages

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)

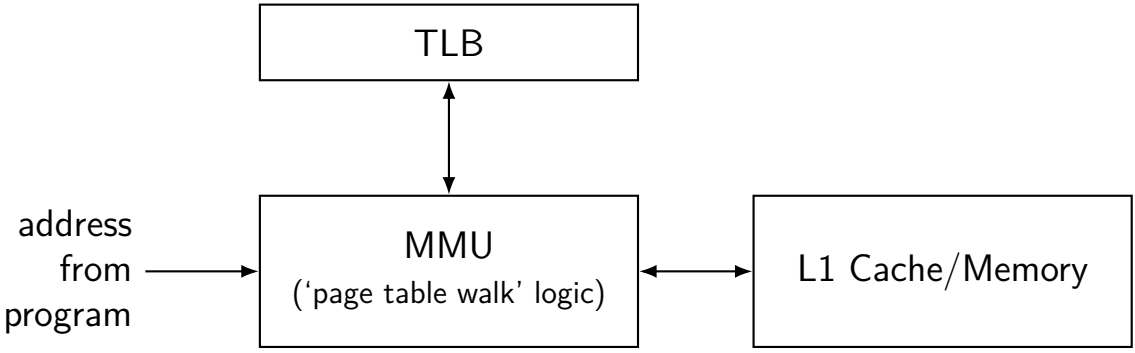
cache translated addresses

called a **TLB** (translation lookaside buffer)
small set-associative hardware cache in MMU
maps virtual page numbers to physical page numbers
contains complete page table entries for small number of pages

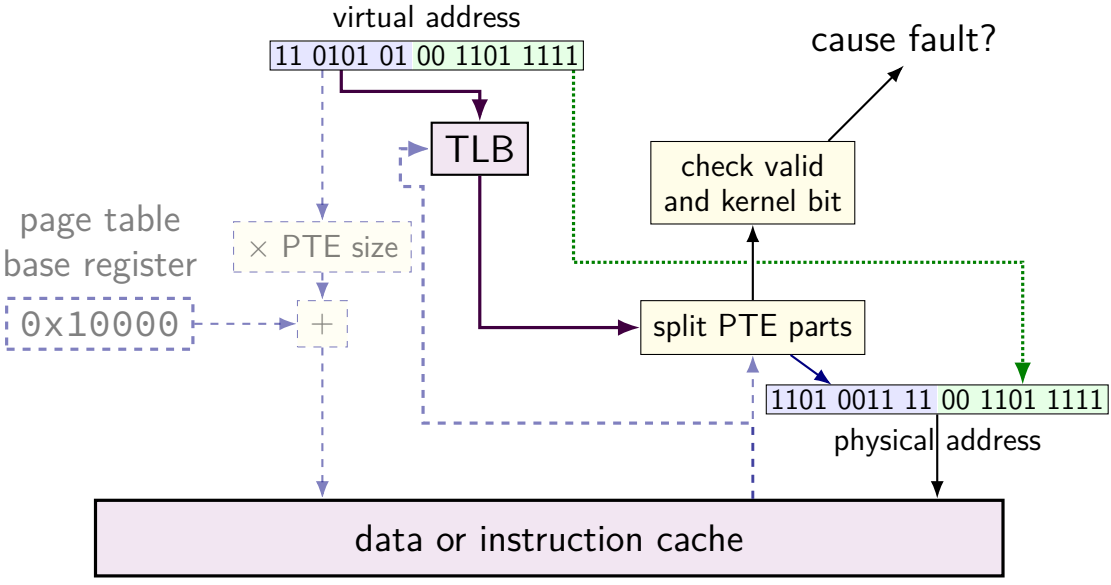
L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

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

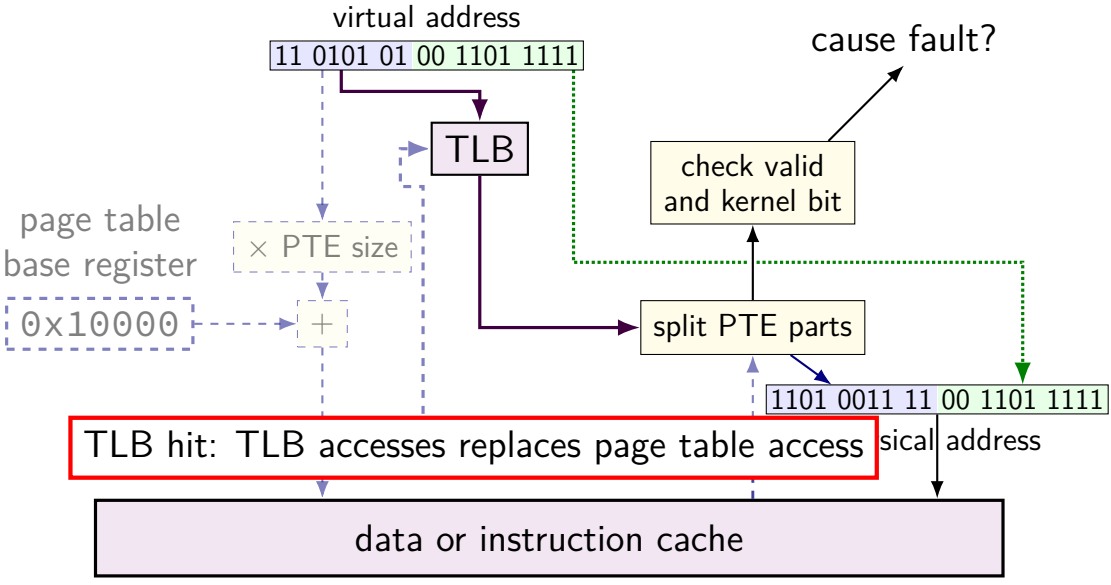
TLB and the MMU (1)



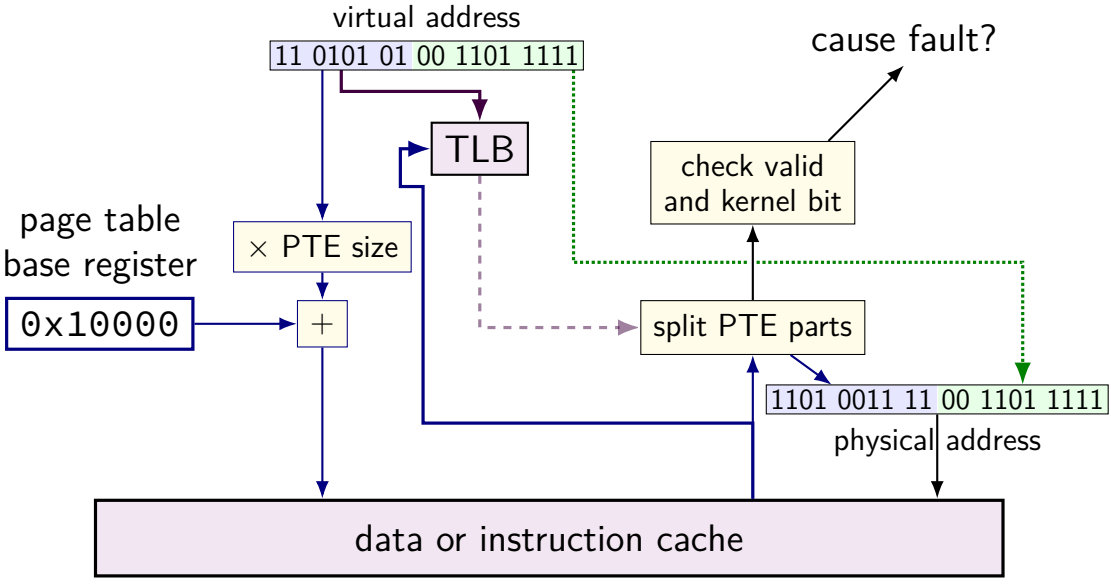
TLB and the MMU (2)



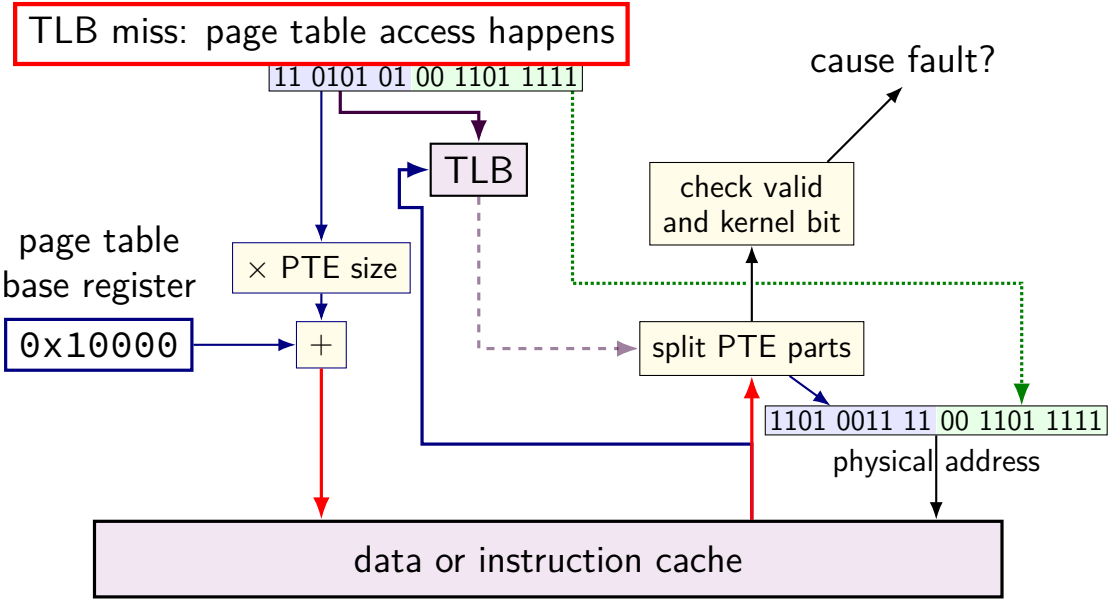
TLB and the MMU (2)



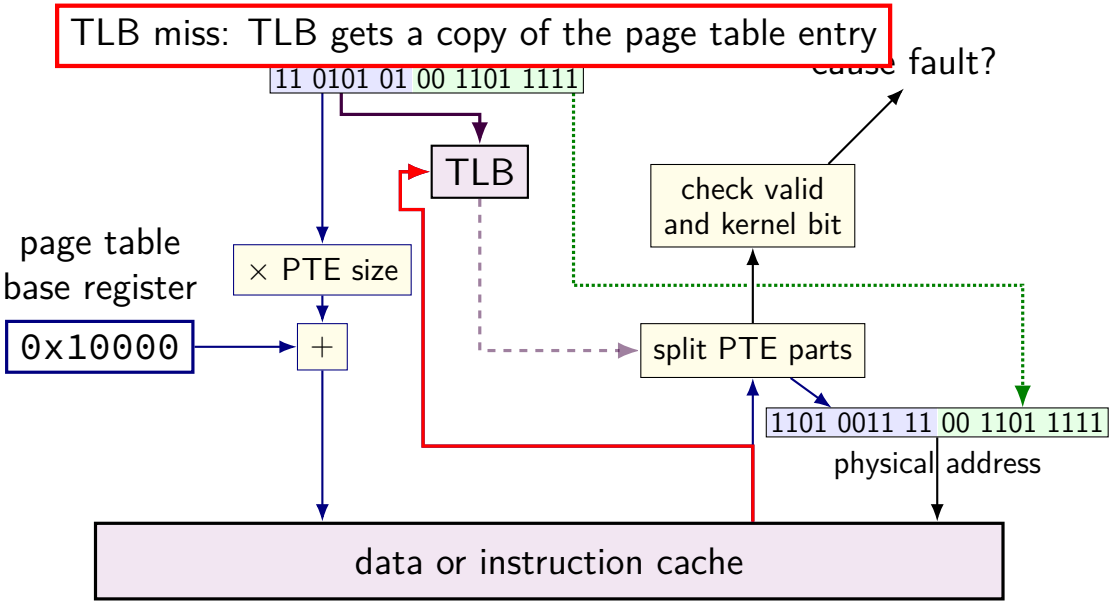
TLB and the MMU (2)



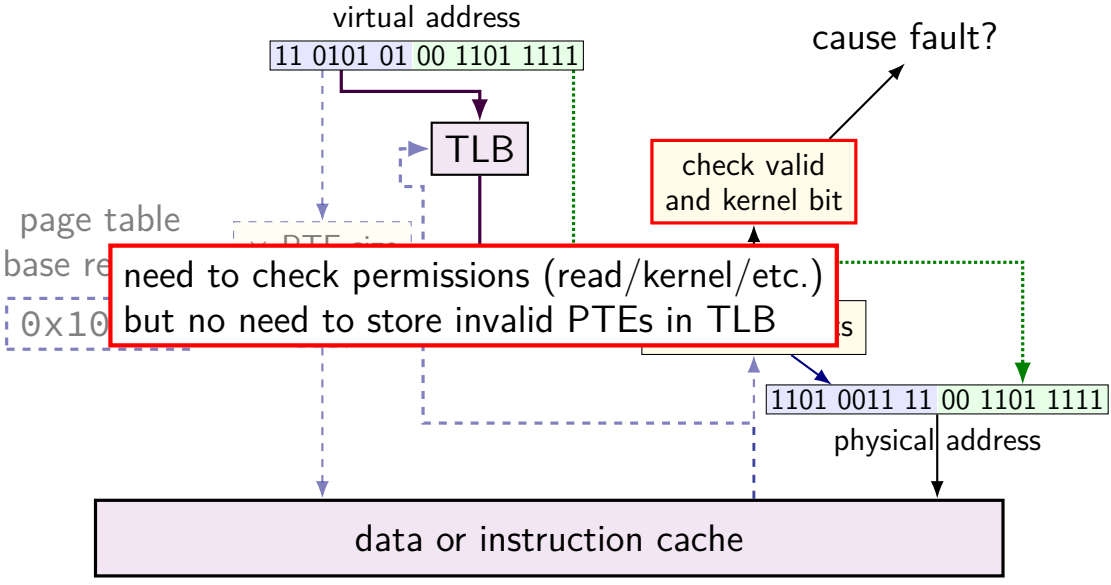
TLB and the MMU (2)



TLB and the MMU (2)



TLB and the MMU (2)



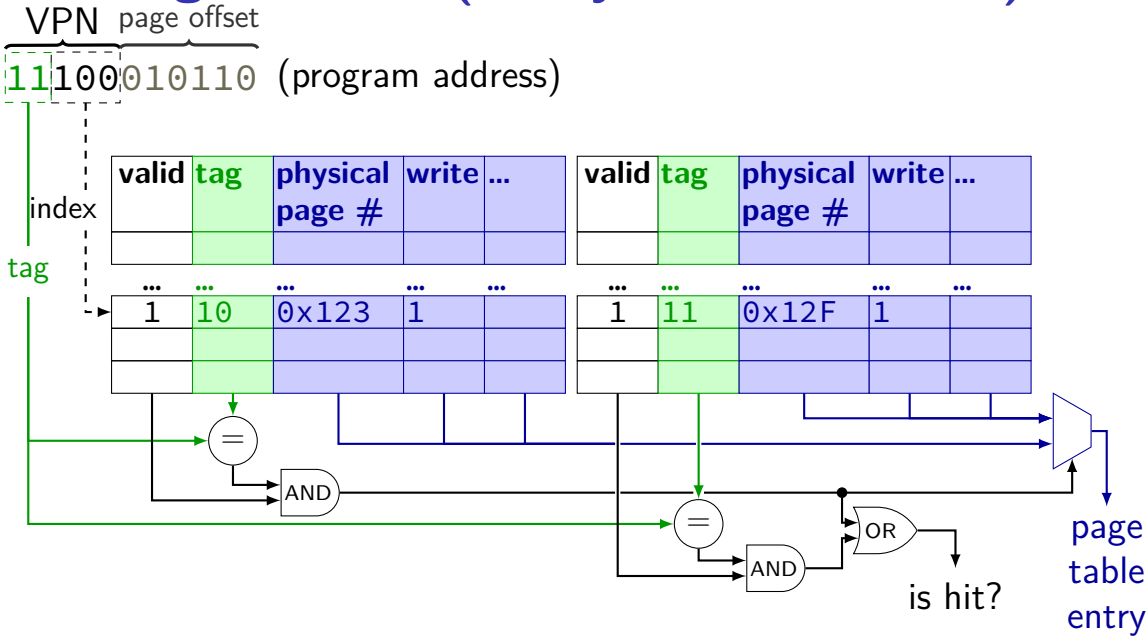
TLB and multi-level page tables

TLB caches valid last-level page table entries

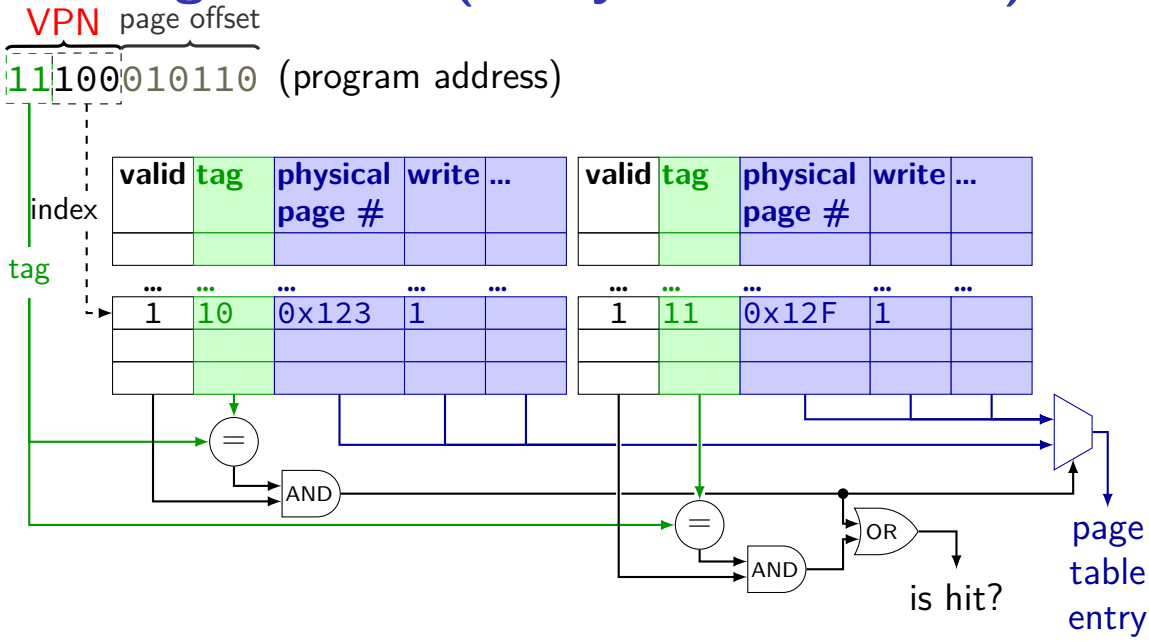
doesn't matter which last-level page table

means TLB output can be used directly to form address

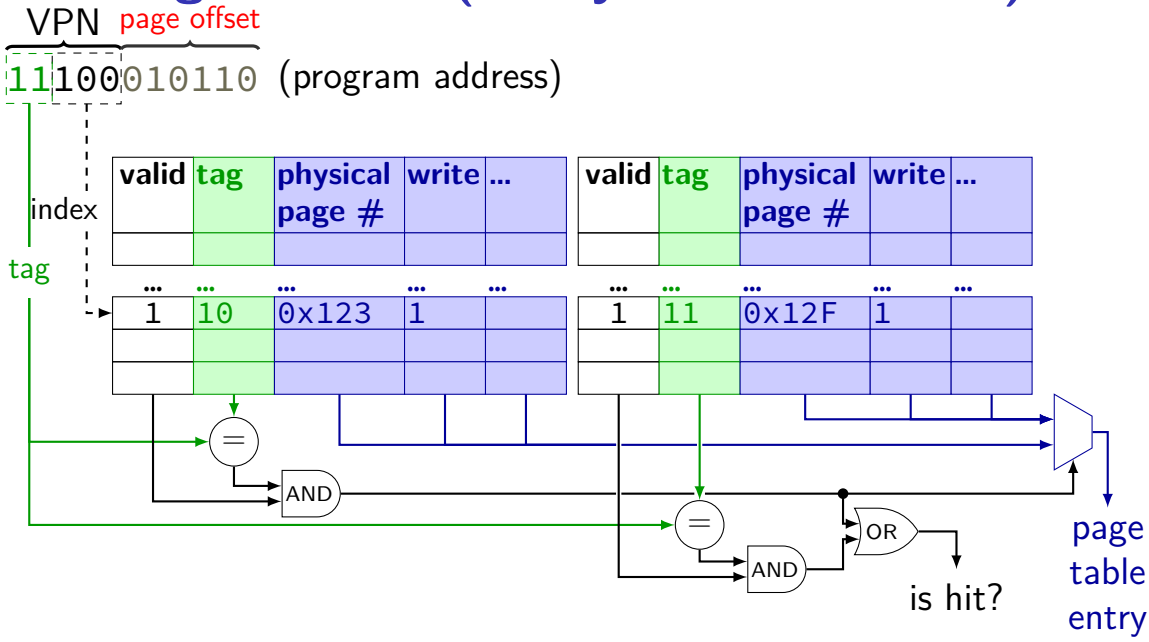
TLB organization (2-way set associative)



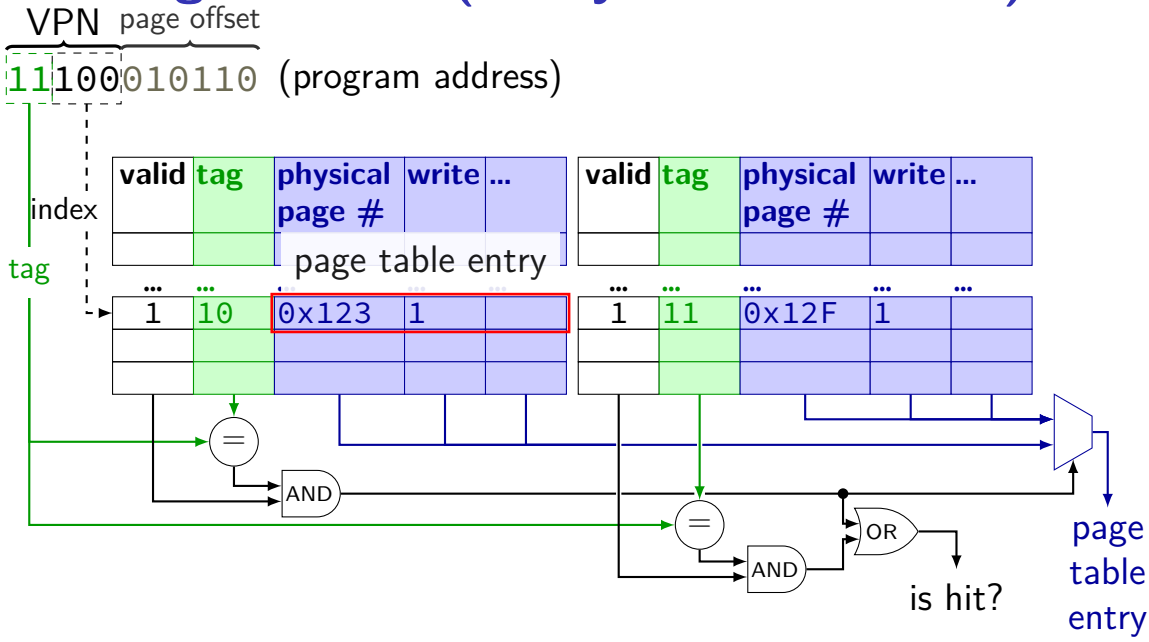
TLB organization (2-way set associative)



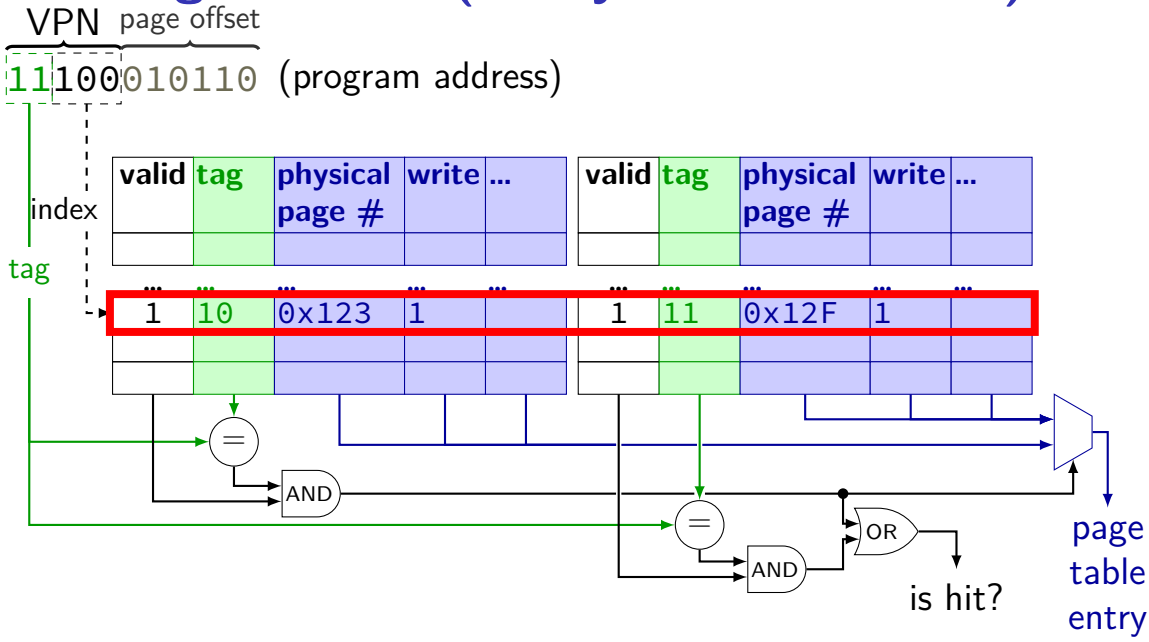
TLB organization (2-way set associative)



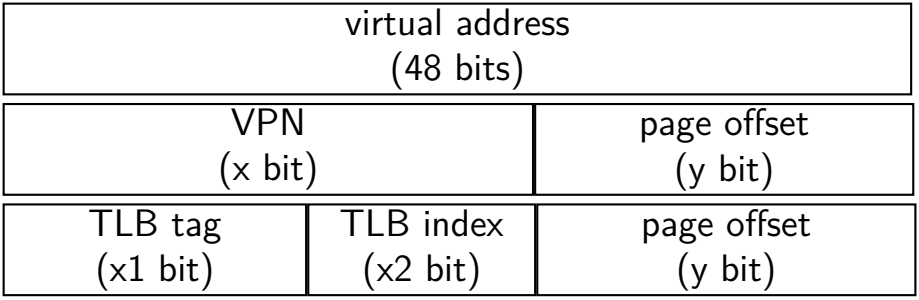
TLB organization (2-way set associative)



TLB organization (2-way set associative)



address splitting for TLBs



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)

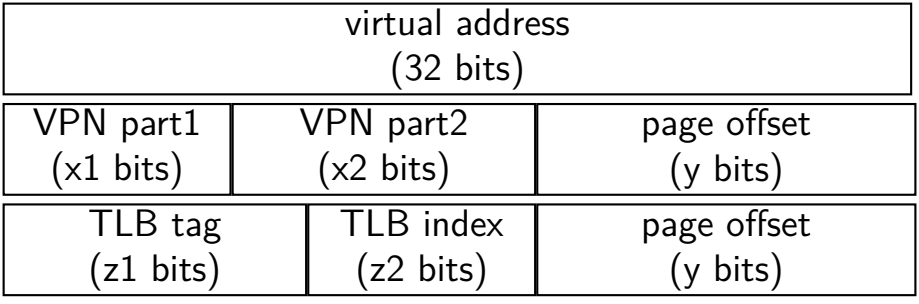
384-entry, 3-way set-associative TLB

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)

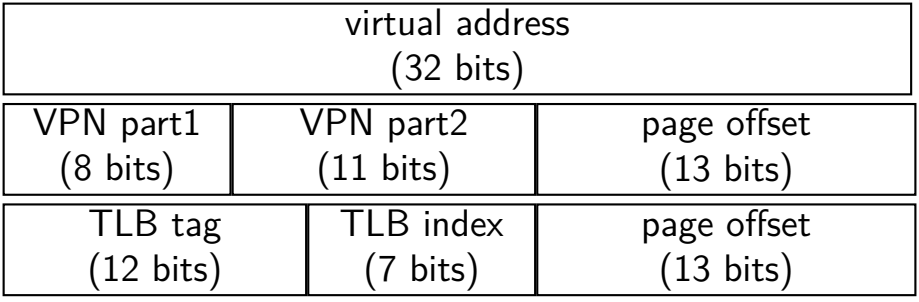
384-entry, 3-way set-associative TLB

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`



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 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	????	—	?

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

VPN

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

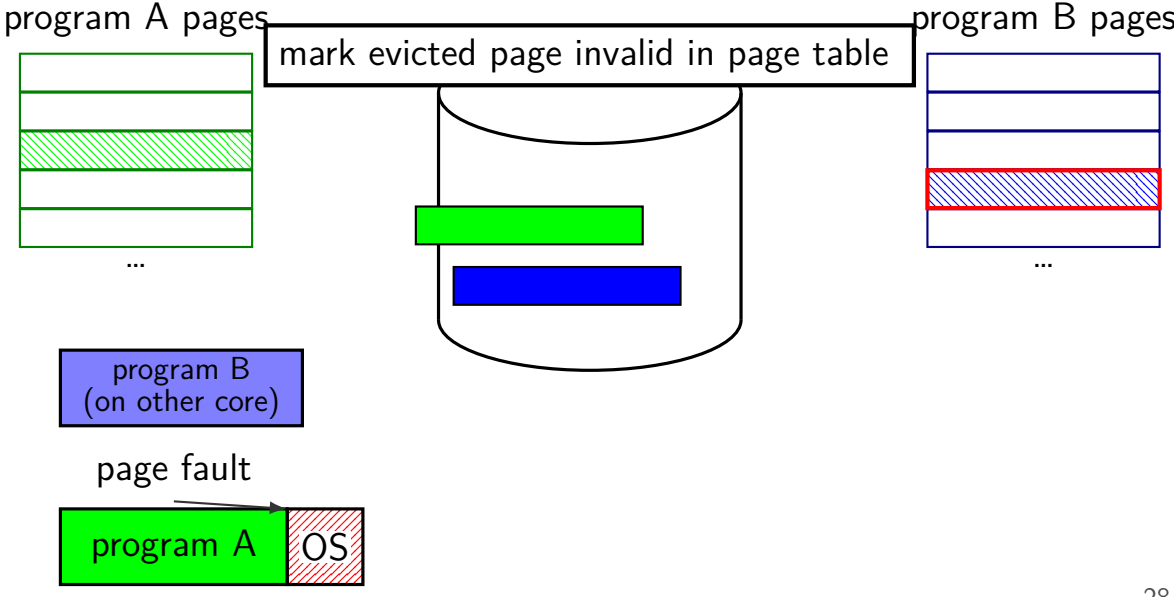
- TLB doesn't contain invalid entries
- MMU will check memory again

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

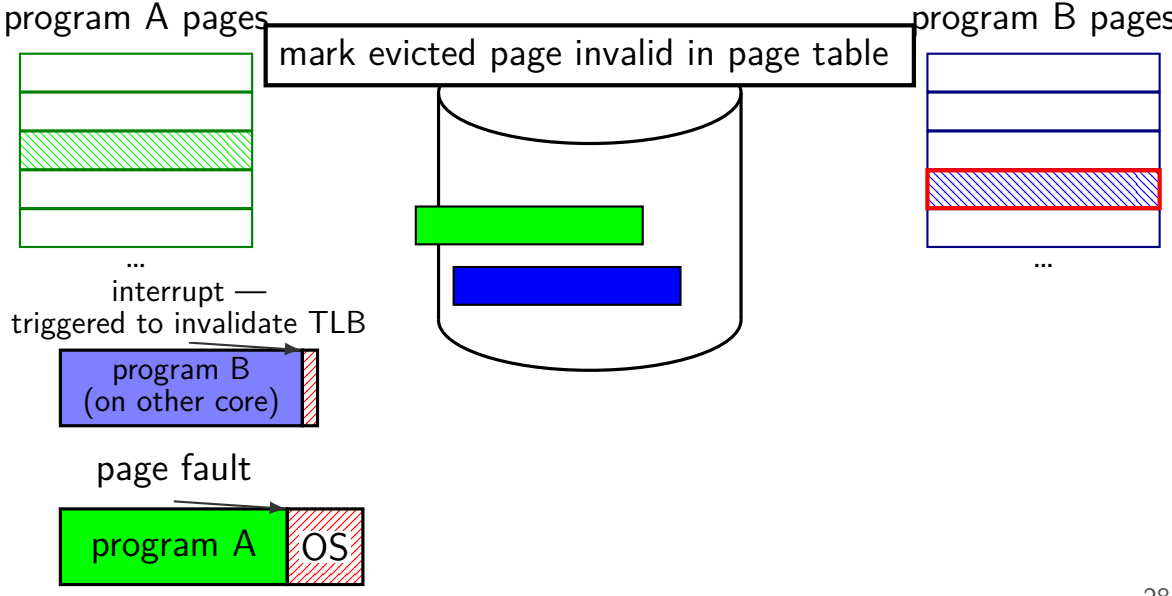
- special instruction (x86: `invlpg`)

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

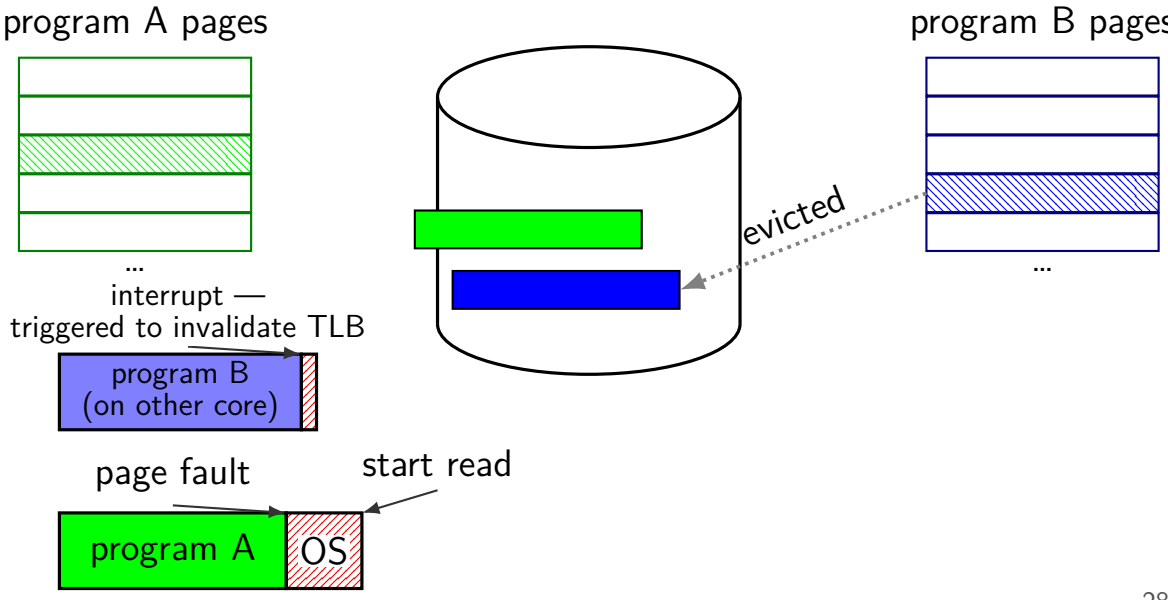
TLB shootdown



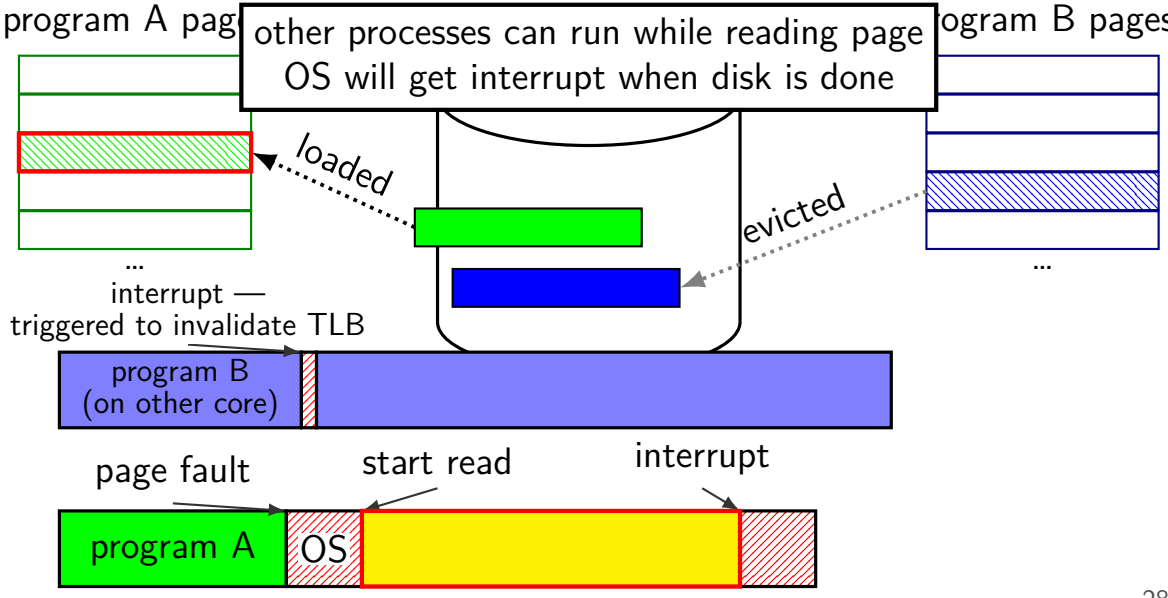
TLB shootdown



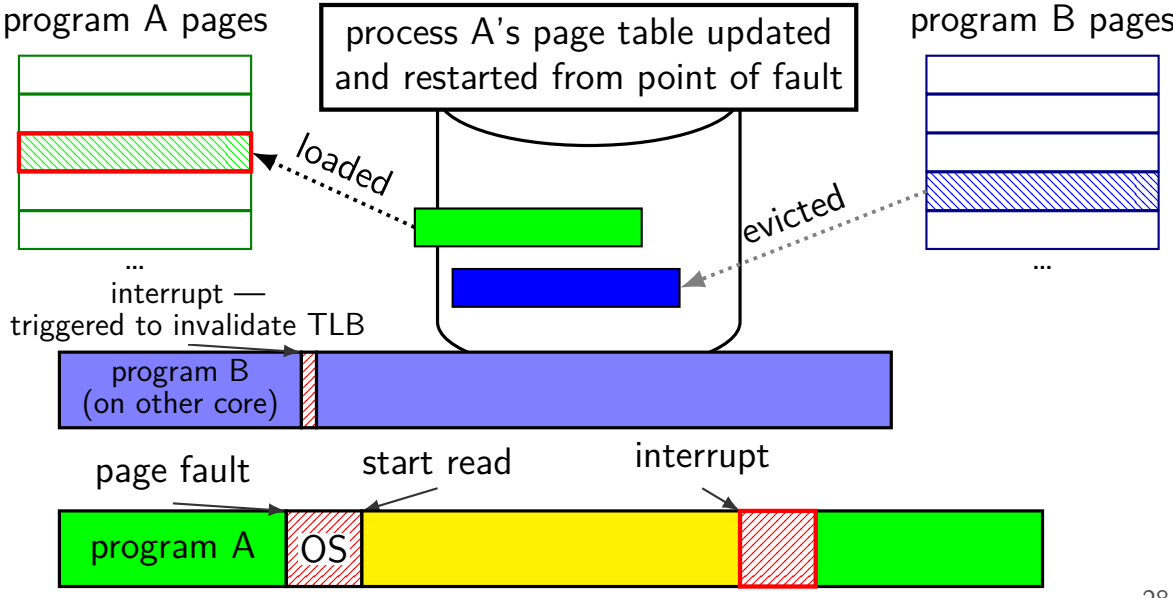
TLB shootdown



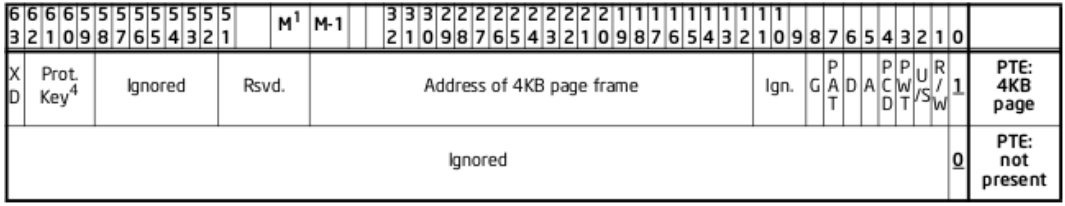
TLB shootdown



TLB shootdown



x86-64 page table entries (1)



present = valid

R/W = writes allowed?

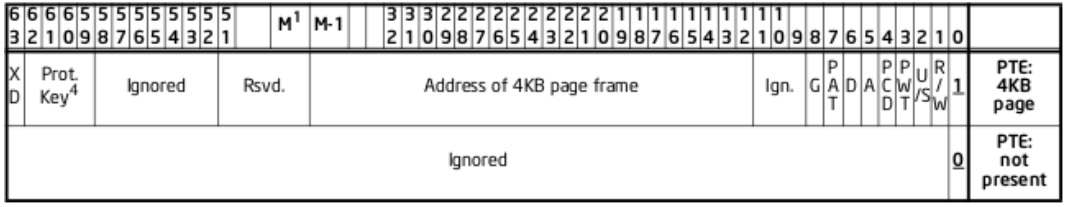
U/S = kernel-only? (“user/supervisor”)

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?

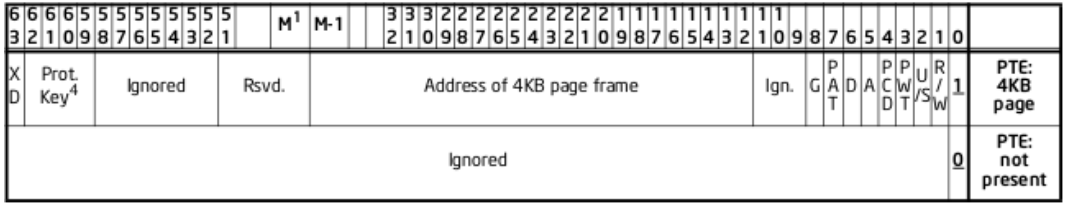
U/S = kernel-only? ("user/supervisor")

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")

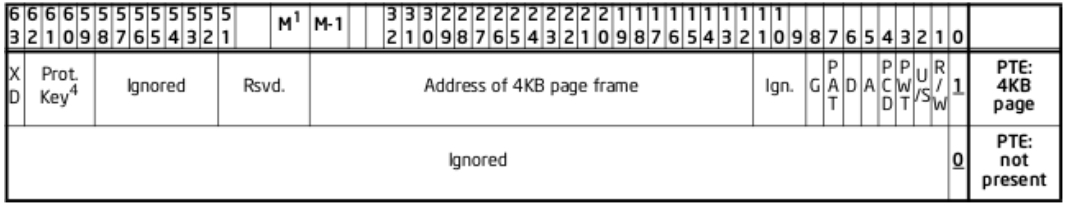
XD = execute-disable?

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)



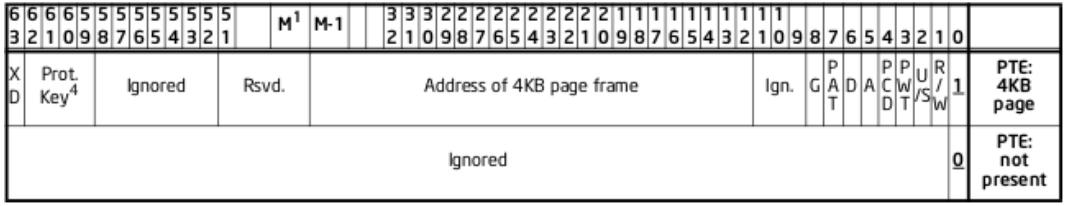
63	62–61	60–59	58–52	51–M	M-1–32	31–12	11–9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	M^1	M-1	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																0	PTE: not present

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