Virtual Memory and Dynamic Scheduling

Computer Architecture

ECE 6913

Brandon Reagen

Announcements

1) Exam
   1) Taking a while to grade the branch prediction question

2) Lab2
   1) < Take pulse >
   2) Please check-in with TAs with any questions!

3) Lab3
   1) Will assign this weekend

4) Today: Finish VM, intro OoO
   1) Take it easy, been a crazy week! (And we're still ahead)

Class "poll" results

You were all relatively positive..

    If you have complaints, please let me know!

Major feedback:

    - Too fast

    - Hard to hear, can't see when I walk to board

    - Want list of papers/readings to accompany lecture

    - Grading unfair: we're willing to work with you

        My goal is to have the grading as fair as possible.

CSAW (FREE)!
Learn about security and privacy

https://www.csaw.io/agenda

NYU | TANDON SCHOOL OF ENGINEERING

Virtual Memory

Virtual memory: Terminology

Page: A single unit of virtual memory. Like a line/block in caches

Frame: A single unit of physical memory (the page of PM)

Page fault: "Miss" in the virtual memory. No mapping exists between virtual and physical addresses, bring in from disk

VM address: Address produced by the program/processor by the program

PM address: Address used to access main memory

Address translation: Process of mapping virtual addresses to physical ones
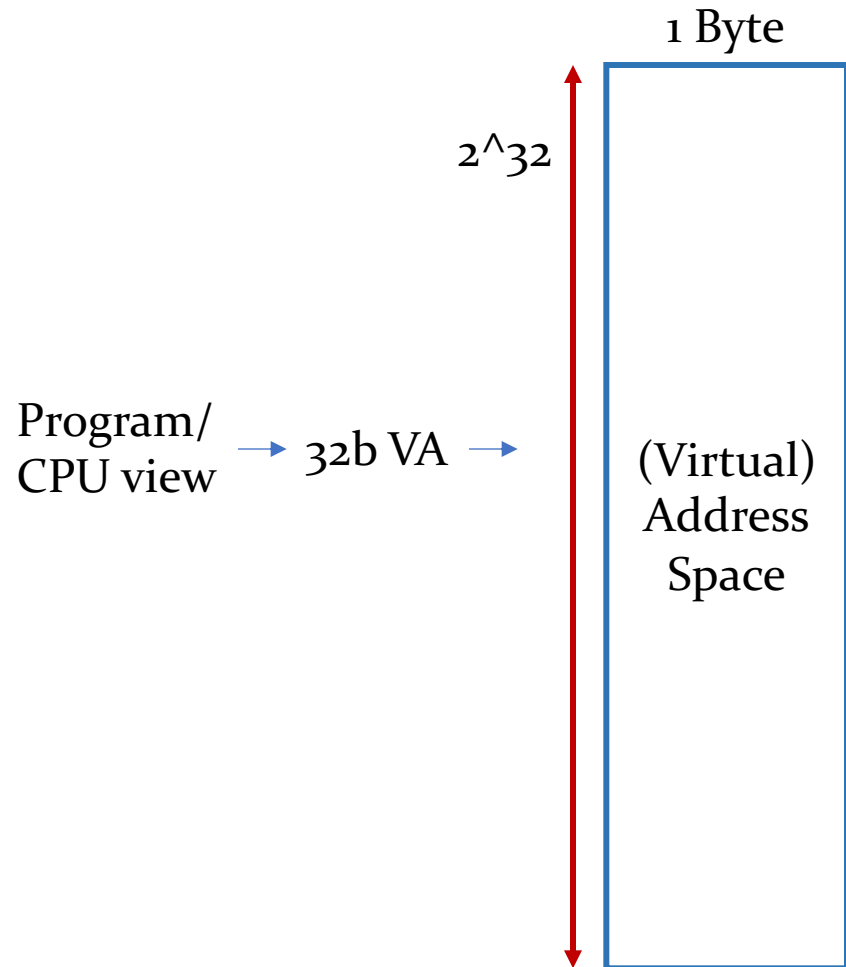
Relocation: Process of mapping physical addresses to virtual ones

A programmer's perspective (from the clouds..)

1 Byte

2^{32}

Program/
CPU view → 32b VA →

(Virtual)
Address
Space

A programmer's perspective (from the clouds..)

1 Byte

2^{32}

Program/ CPU view → 32b VA → (Virtual) Address Space
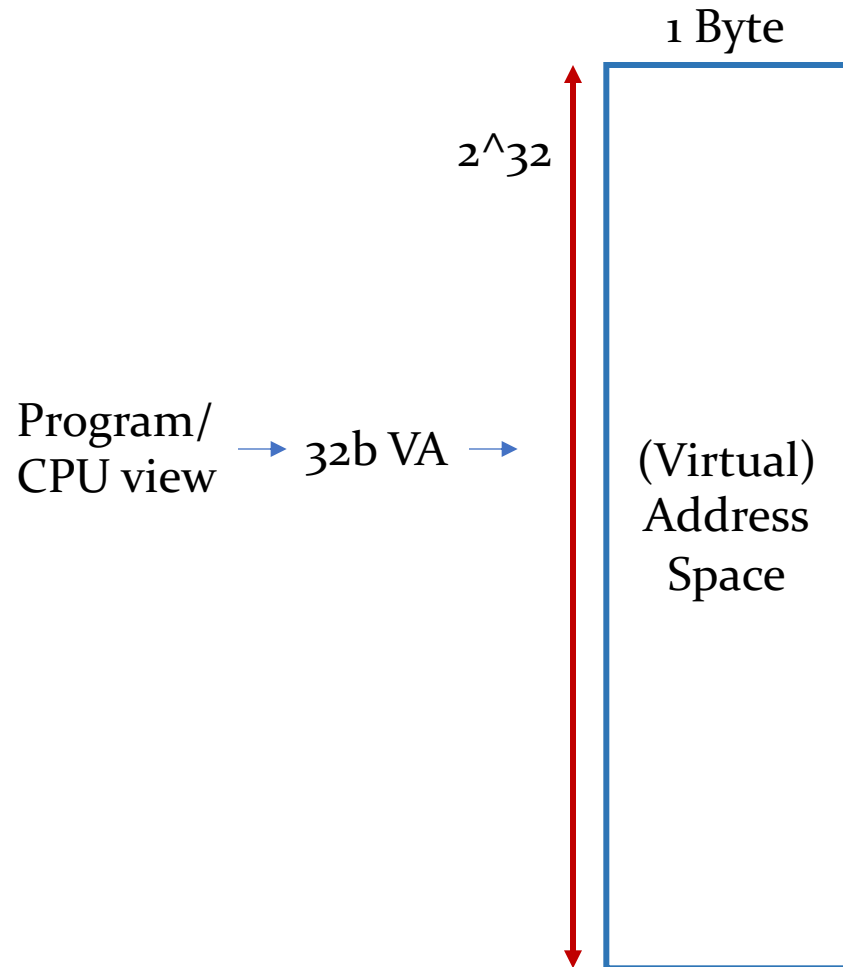
Once again the complexities are dumped on the hardware and OS
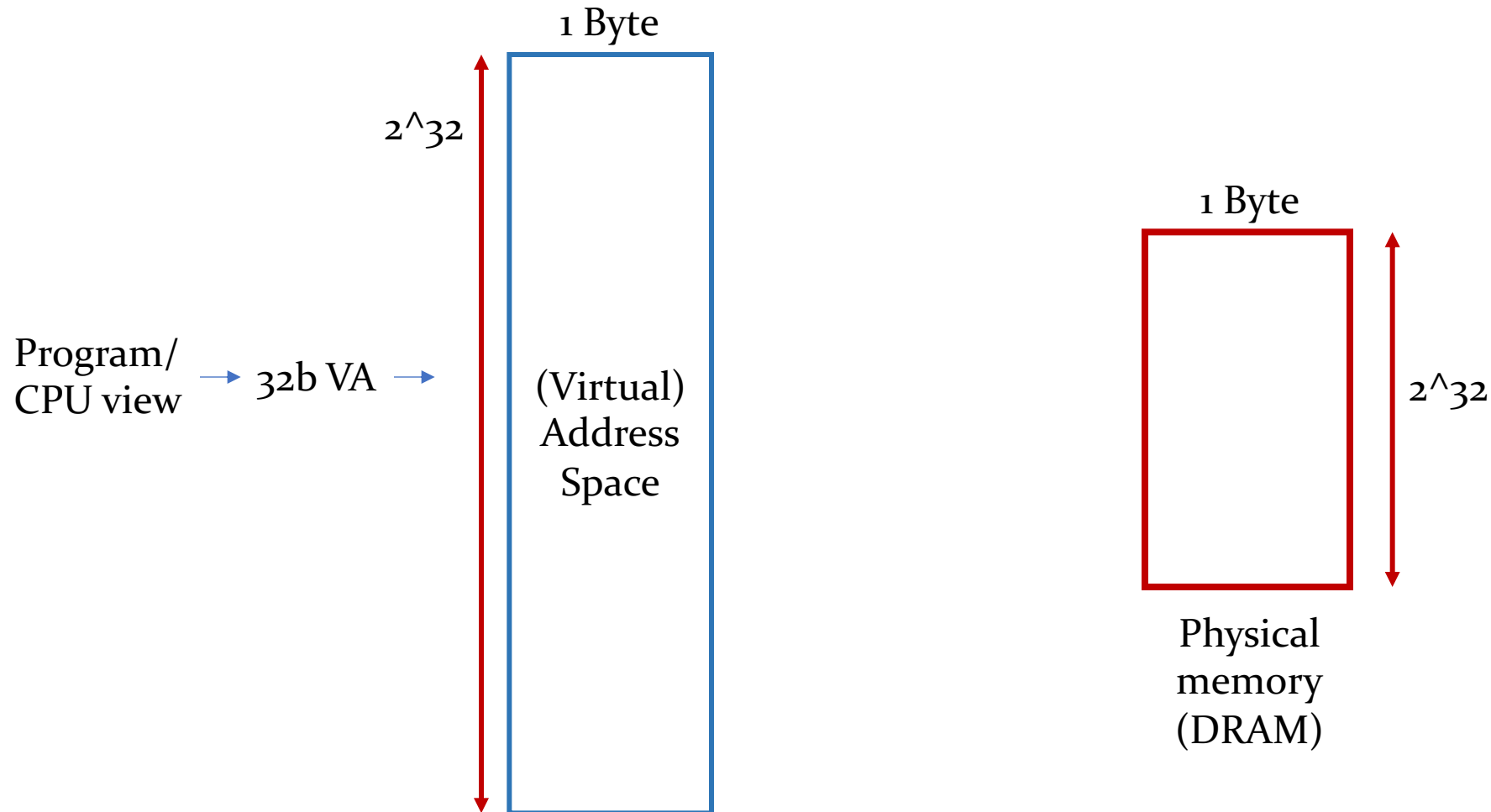
Problem 1: How does everything fit?

1 Byte

2^{32}

Program/
CPU view → 32b VA →

(Virtual)
Address
Space

1 Byte

2^{32}

Physical
memory
(DRAM)

Each program needs
4GiB in a 32b machine

You have tens-hundreds
of programs running right
now, how does it fit?

Problem 1: How does everything fit?

1 Byte

2^{32}

Program/ CPU view → 32b VA → (Virtual) Address Space

1 Byte

2^{32}

Physical memory (DRAM)
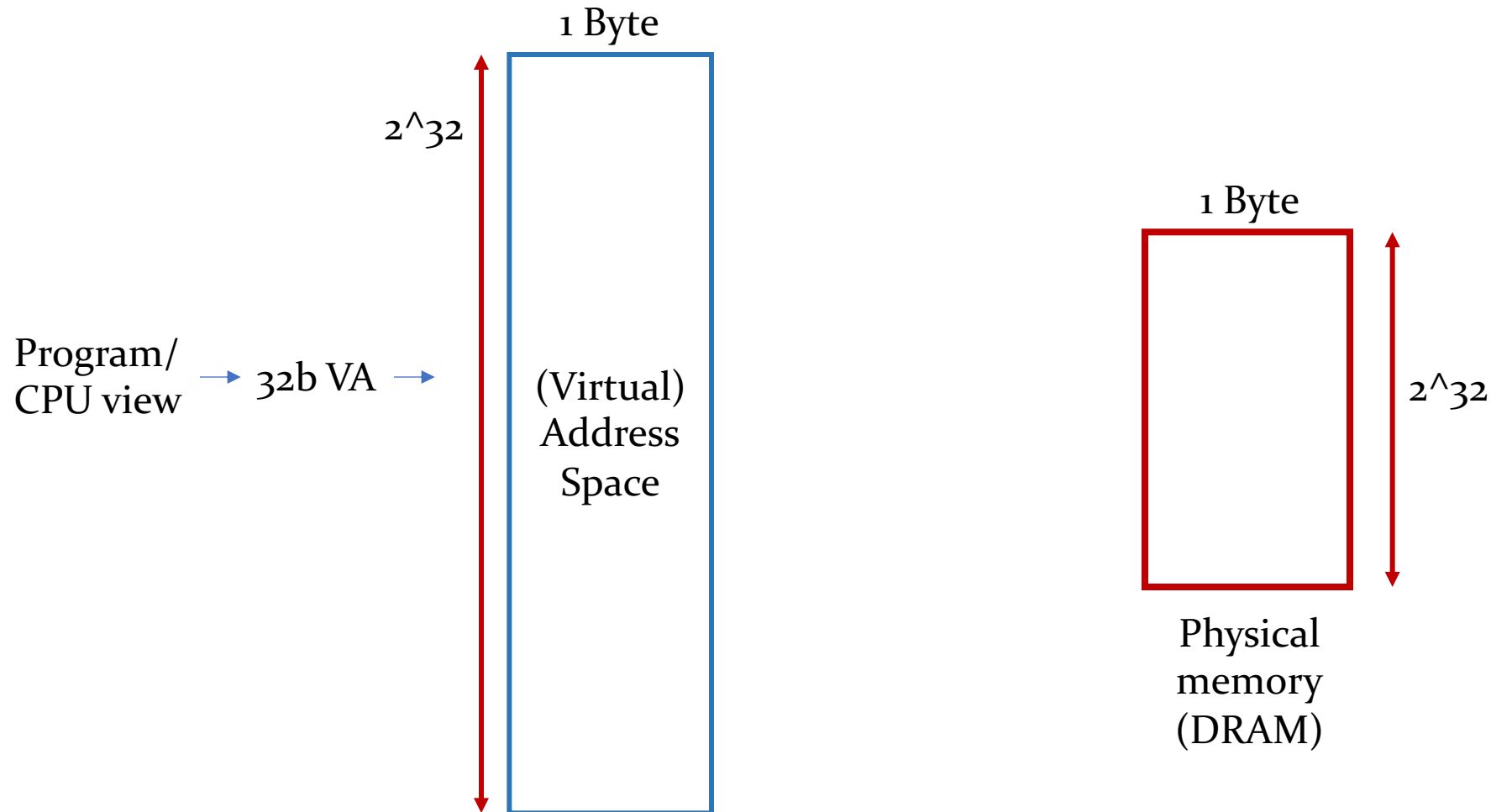
Each program needs 4GiB in a 32b machine

You have tens-hundreds of programs running right now, how does it fit?

It doesn't..

NYU | TANDON SCHOOL OF ENGINEERING

Problem 2: Where does everything go?

Program 1 → 32b VA → 2^{32}

1 Byte

(Virtual) Address Space

Program 2 → 32b VA

2^{32}

1 Byte

(Virtual) Address Space

...

?

1 Byte

2^{32}

Physical memory (DRAM)
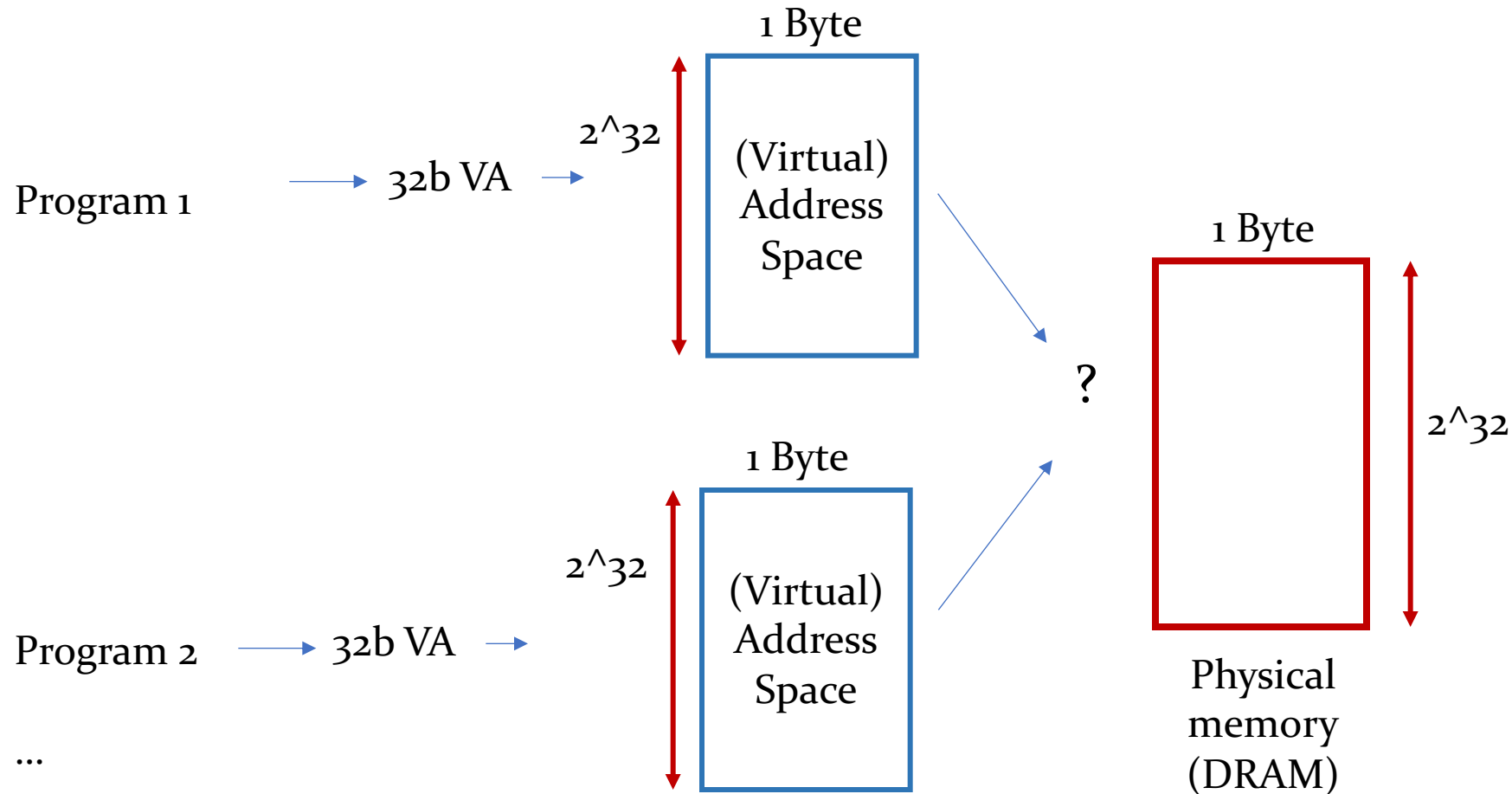
Each program has a VA space larger than the machine's physical memory.

How should we divide up the limited physical memory resource?

NYU | TANDON SCHOOL OF ENGINEERING

Problem 3: Memory protection

Program 1 $\rightarrow$ 32b VA $\rightarrow$ 2^{32}

1 Byte

(Virtual) Address Space

Program 2 $\rightarrow$ 32b VA $\rightarrow$ 2^{32}

1 Byte

(Virtual) Address Space

...

1 Byte

2^{32}

Physical memory (DRAM)
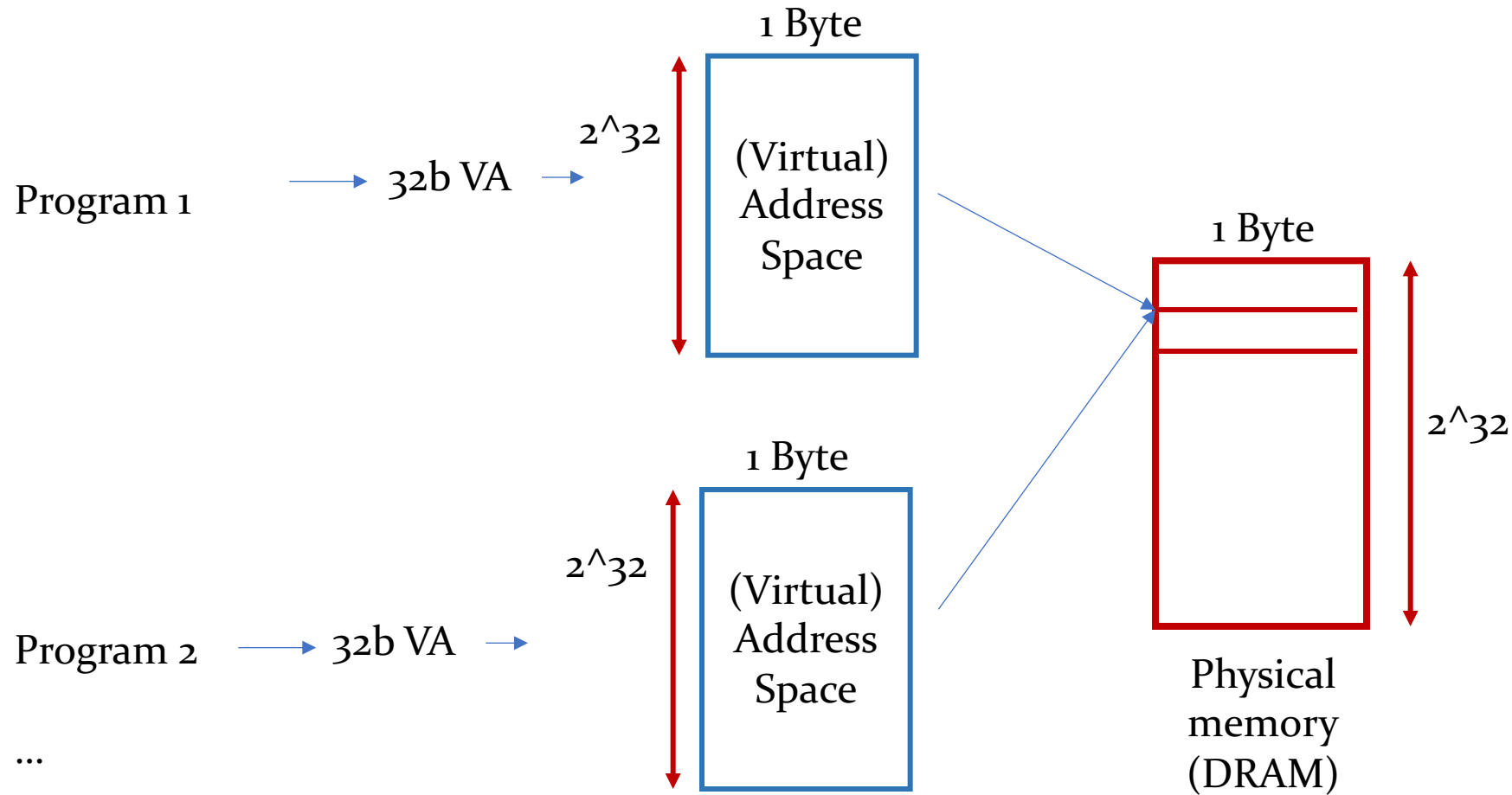
Protect against the benign and malicious case.

What's that mean?

Problem 4: What about sharing?

1 Byte

2^{32}

Program 1 → 32b VA → (Virtual) Address Space

1 Byte

Program 2 → 32b VA → 2^{32} (Virtual) Address Space

...

1 Byte

Physical memory (DRAM)

2^{32}
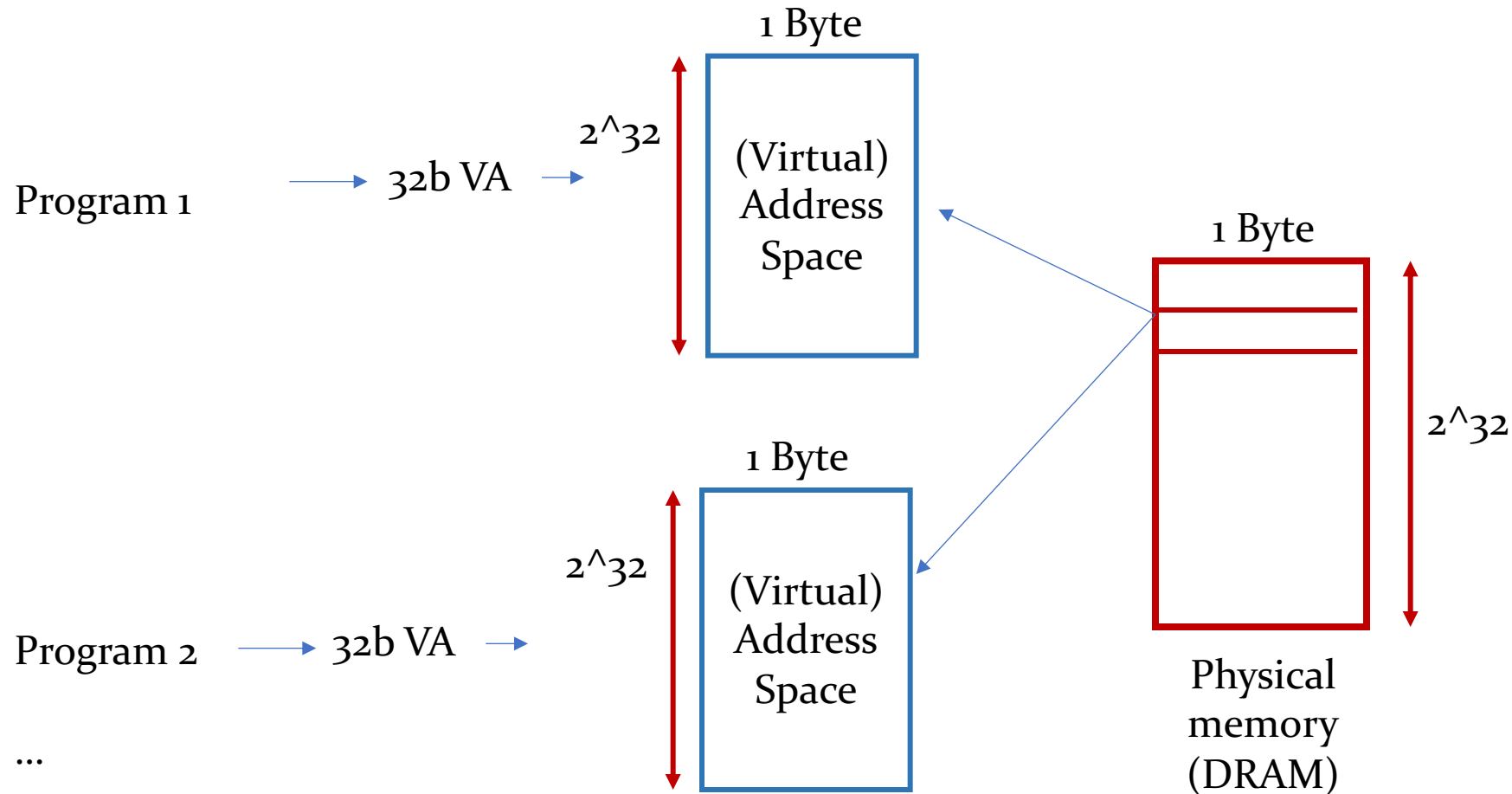
Why would we want to share?

What could we share?

How do we solve all these problems?

Hint: famous saying in computer science.

Indirection!

"All problems in CS can be solved by another level of indirection"

- David Wheeler

Add a level of indirection: Virt -> Phys mapping
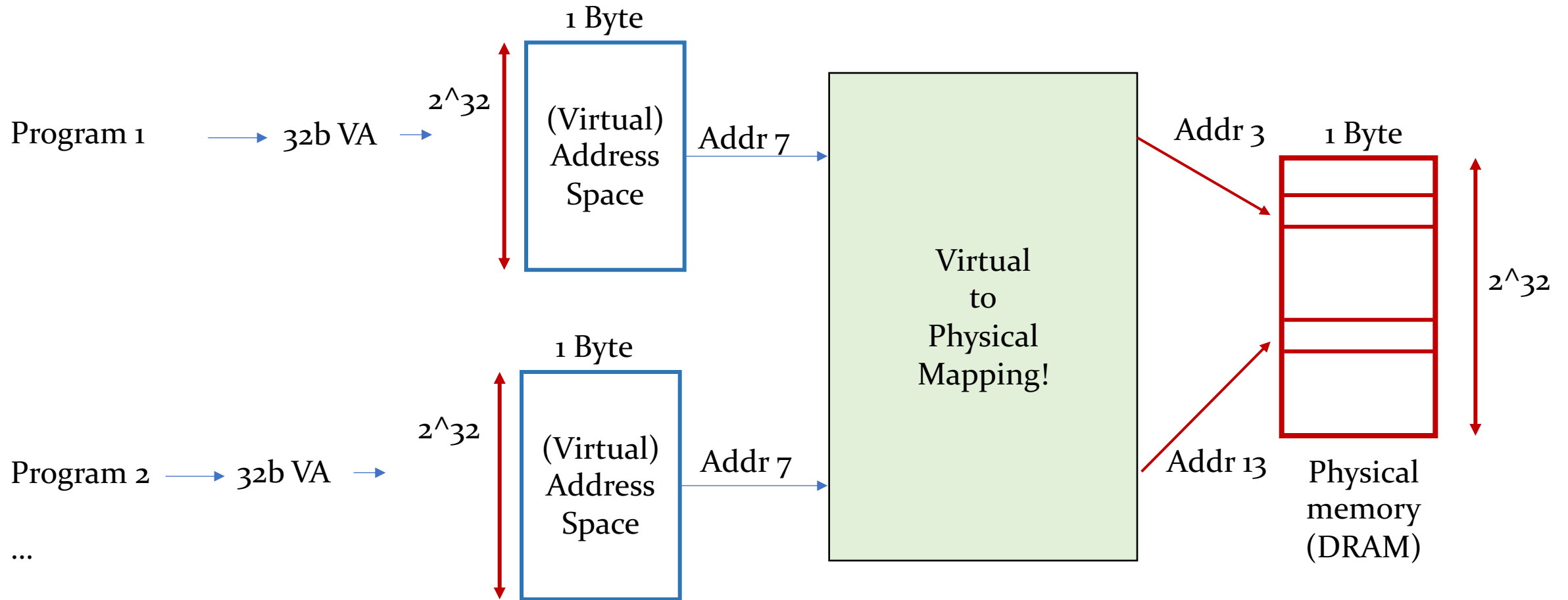
The hardware that handles address mapping, or Translation, is called the MMU: Memory Management Unit

What does this VM (indirection) fix?

1) Isolation of program address spaces
   1) Programs can't access each other's memory (regardless of intent)
   2) Users can't access privileged memory (OS code)

2) Significantly simplifies programmer's life!
   No need to worry about any of this:
   1) Write code like have all address space
   2) Don't worry about overwriting
   3) Code portable to machine with any size memory!

3) Provides means to use physical memory (DRAM) as a cache
   1) Memory space too big, so only store "hot" pages in DRAM
   2) Everything must live somewhere, so there we must rely on disk.

What's happening here?

What it looks like

Program 1 → 32b VA

Program 2 → 32b VA

...

Page size

Virtual
to
Physical
Mapping!

Page size

~ DRAM size /
page size

Empty

Physical
memory
(DRAM)
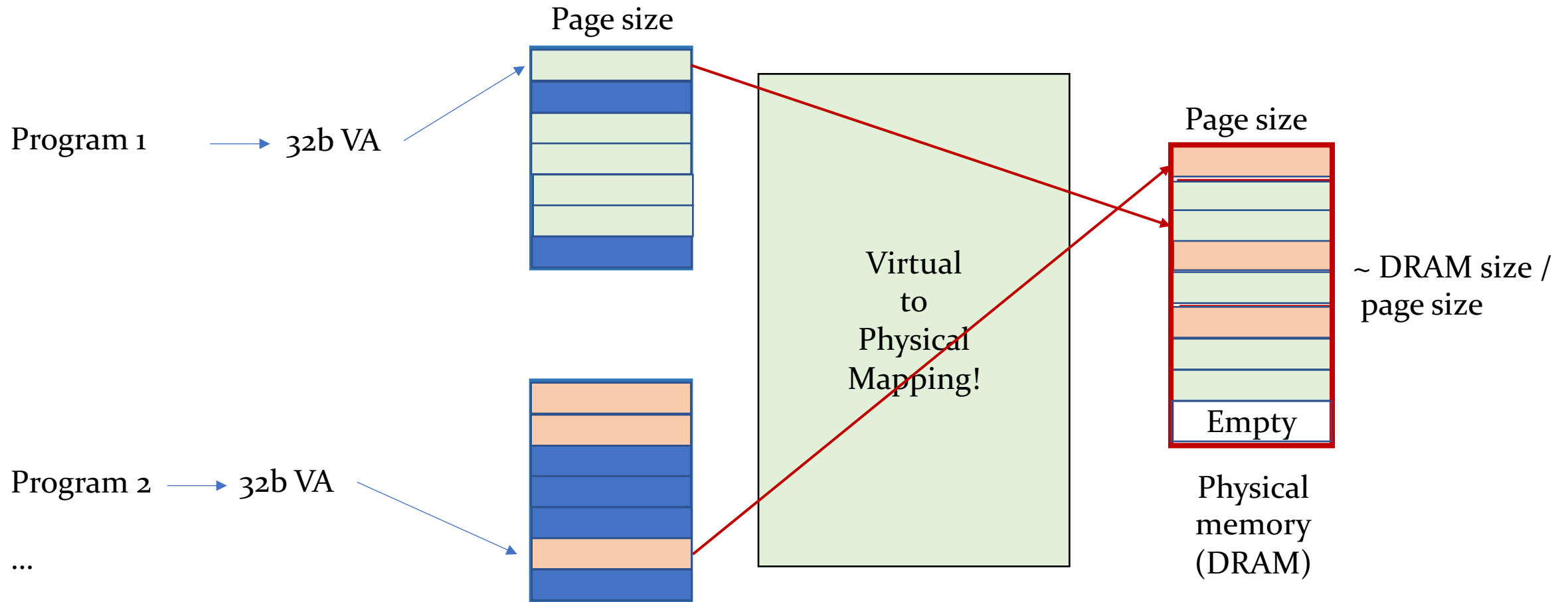
What it looks like

Remember, this isn't real!
It's just an illusion.

Page size

Program 1 → 32b VA

Virtual
to
Physical
Mapping!

Page size

~ DRAM size /
page size

Empty

Physical
memory
(DRAM)

Program 2 → 32b VA

...

Page fault!
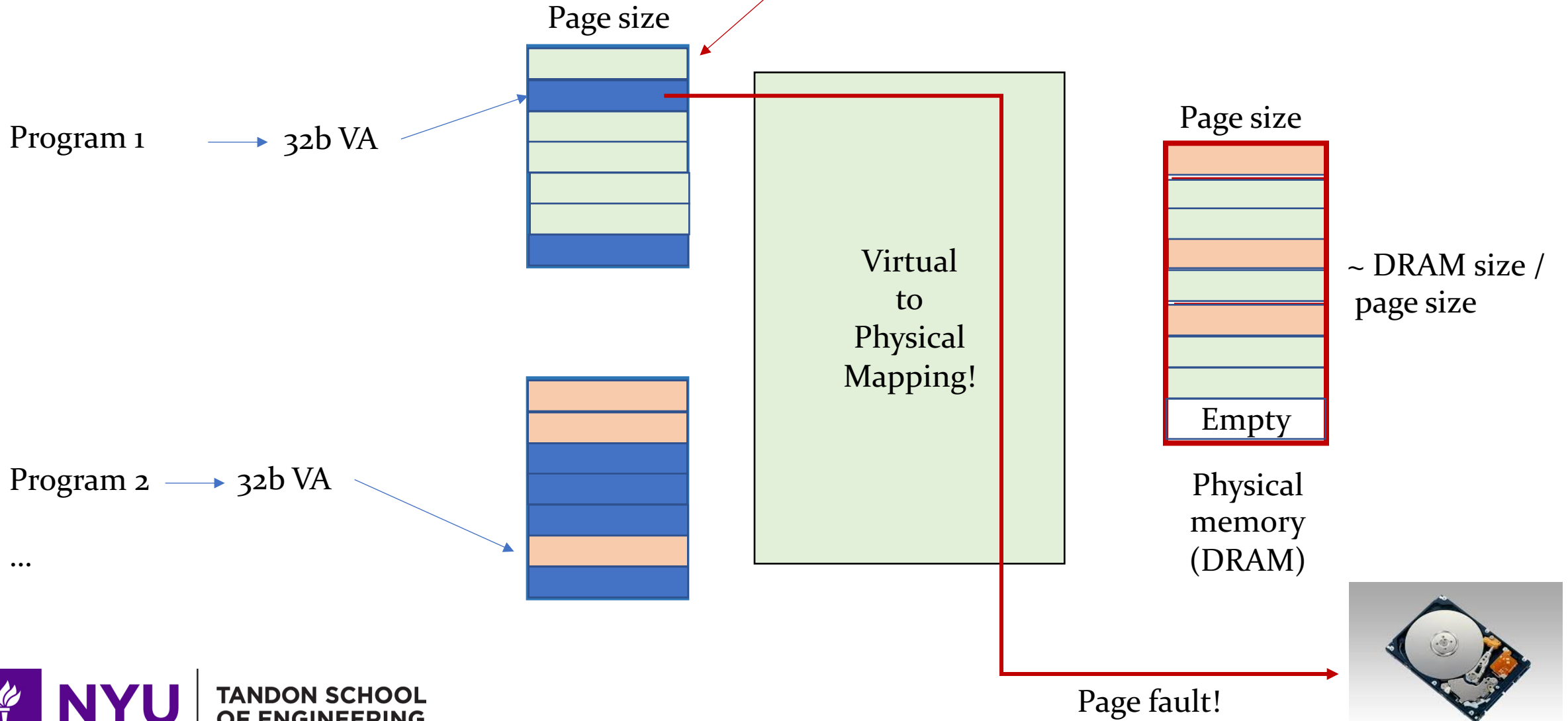
Addressing memory

Virtual address has two parts: virtual page number and page offset

The size of a page is determined by the page offset

    This means we don't have to translate it!

Translation: Convert a program's virtual page number to the machine's physical page (frame) number

Address translation - simplified

Virtual address

31 30 29 28 27 ···················· 15 14 13 12 11 10 9 8 ··········· 3 2 1 0

Virtual page number	Page offset

Translation

29 28 27 ···················· 15 14 13 12 11 10 9 8 ·········· 3 2 1 0

Physical page number	Page offset

Physical address
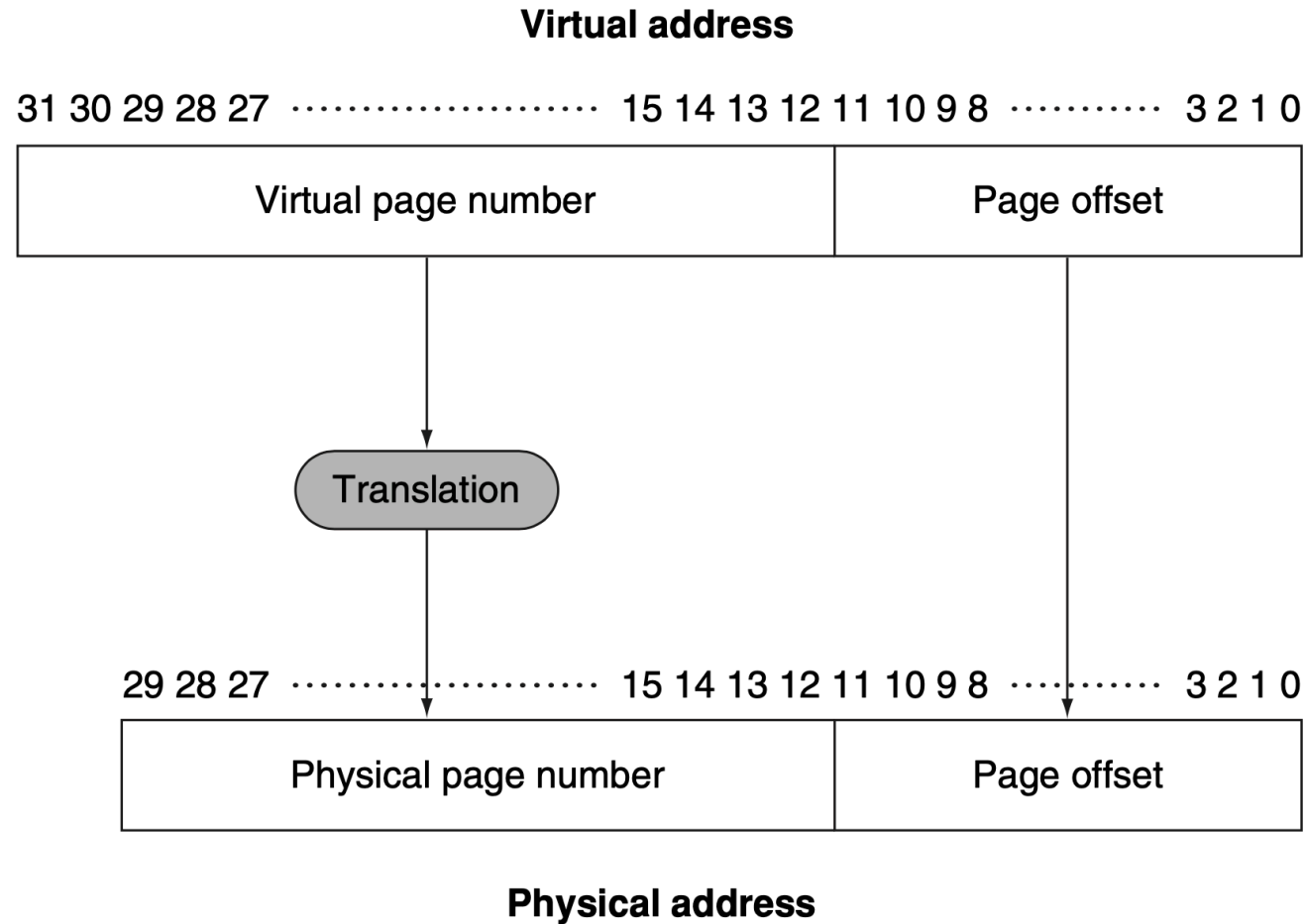
Placing and finding pages

Placement:

OS can map any virtual page to any physical page

Using full associativity minimizes conflicts for limited resource

What was the problem with fully-associative caches?

Finding is hard.

Takes long time, or requires expensive parallel access

VM uses a Page Table:

a special, auxiliary data structure to tack VA -> PA mappings

Page tables

A page table takes virtual addresses as input and outputs physical address

  Track all virtual to physical mappings

  This means they can get quite large

Page tables live in memory

  A special register (page table register)
  points to the start of each processes page table

Quiz: How large is a single program's page table given:

  32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry

Page tables

A page table takes virtual addresses as input and outputs physical address

  Track all virtual to physical mappings

  This means they can get quite large

Page tables live in memory

  A special register (page table register)

  points to the start of each processes page table

Quiz: How large is a single program's page table given:

  32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry

  Num entries: 2^32 / 2^12 = 2^20.

  Total size: 2^20 * 2^2 <bytes per PTE> = 4 MiB

Page table example

Base address

Valid bit: indicates whether the mapping exists or not

Also lives in memory..

Page table register

Virtual address

31 30 29 28 27 ·············· 15 14 13 12 11 10 9 8 ········· 3 2 1 0

| Virtual page number | Page offset |

20

Valid   Physical page number

Page table

If 0 then page is not present in memory

18

29 28 27 ·············· 15 14 13 12 11 10 9 8 ········· 3 2 1 0

| Physical page number | Page offset |

Physical address

Program Address (VA)

PTE: Page Table Entry

Real address (PA)
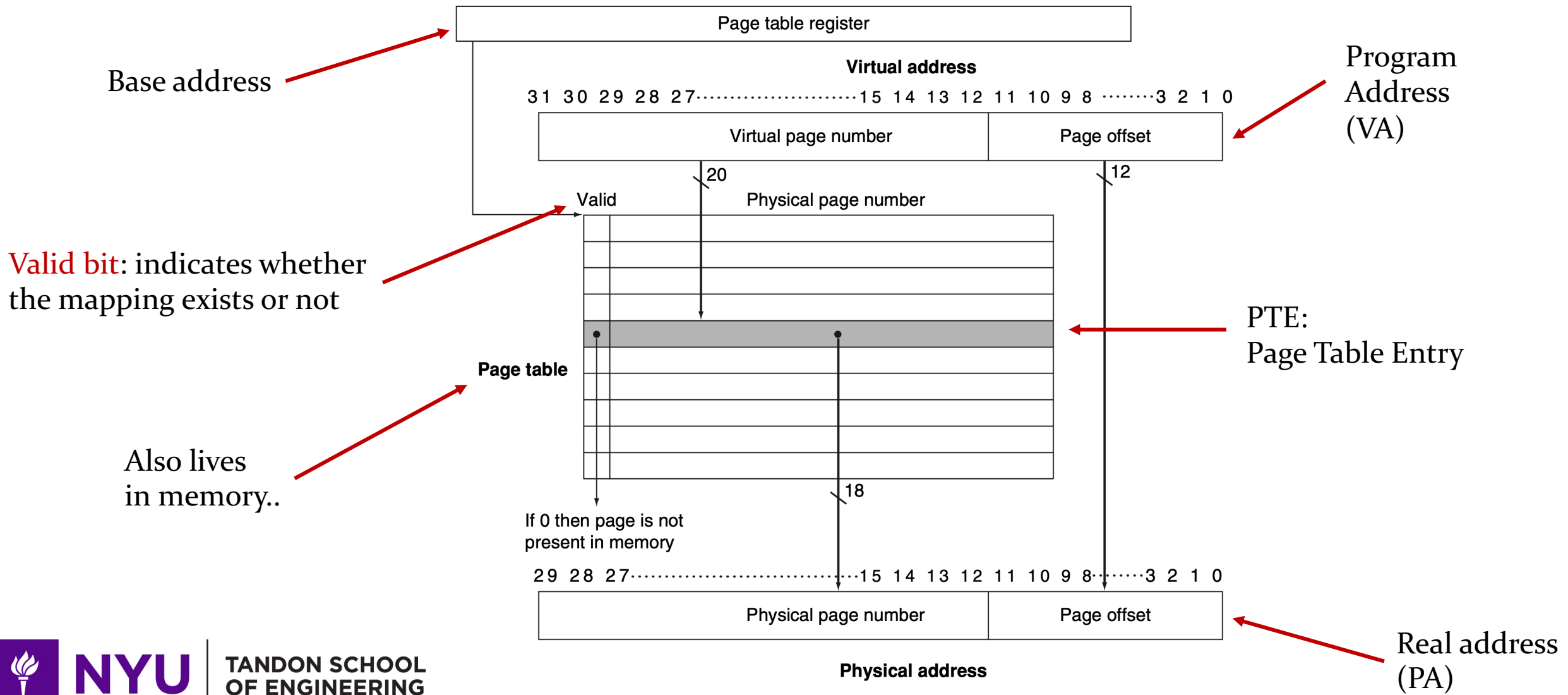
Page faults

If we access a PTE and the valid bit is zero we incur a page fault

> Raise an exception and give control to the OS

OS will go out and find the data stored on disk, bring it into main memory, and create a valid mapping in the page table

When process starts, OS creates "swap space" on disk

> This is a pre-allocation of all virtual addresses

OS creates data structure for where all program's VM addresses are stored on disk

> This can be part of the page table or separate structure

> Usually separate structures, enables PT optimizations

Page faults

Virtual page number

Page table
Physical page or
Valid   disk address

Physical memory

Disk storage
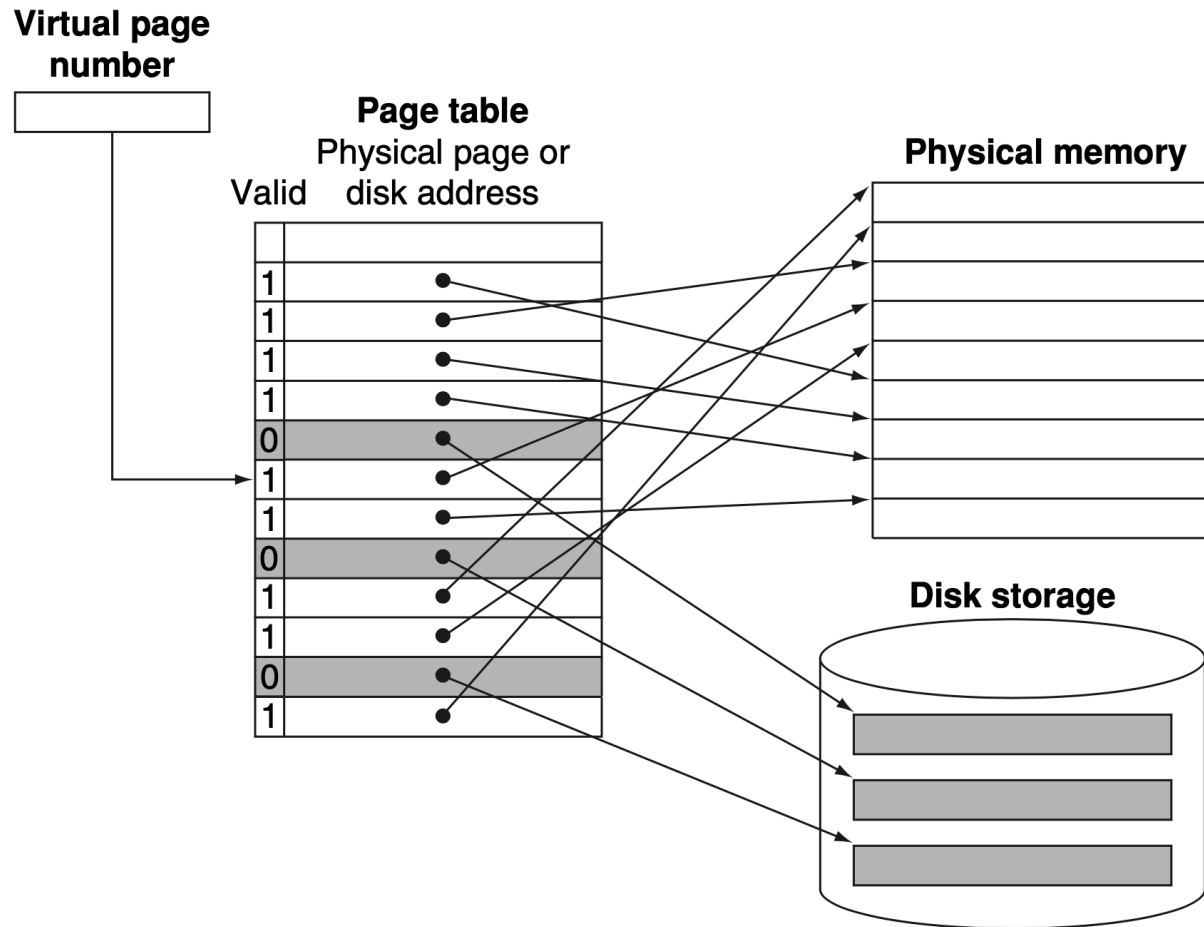
Page faults are bad because
we must go to disk.
Can't really get around that..

But we also don't like going to memory..
and the page table lives in main memory

What does this imply?

How can we fix this?

Translation Lookaside Buffer (TLB)

Each time we need to access memory, need to make 2 trips!

   One for translation, one for access event

We can cache virtual to physical address translations

This is what the TLB does:

   Leverage locality to store recently used translations close to processor

TLB is why professors spend so long thinking about paper names and titles

   Terrible name. <u>Translation Cache</u> much better.

Does this make sense?
a) We have caches
b) VM let's DRAM cache pages from disk
c) TLB caches translations to avoid trip to memory

NYU | TANDON SCHOOL OF ENGINEERING

TLB example

Ref bits used for replacement
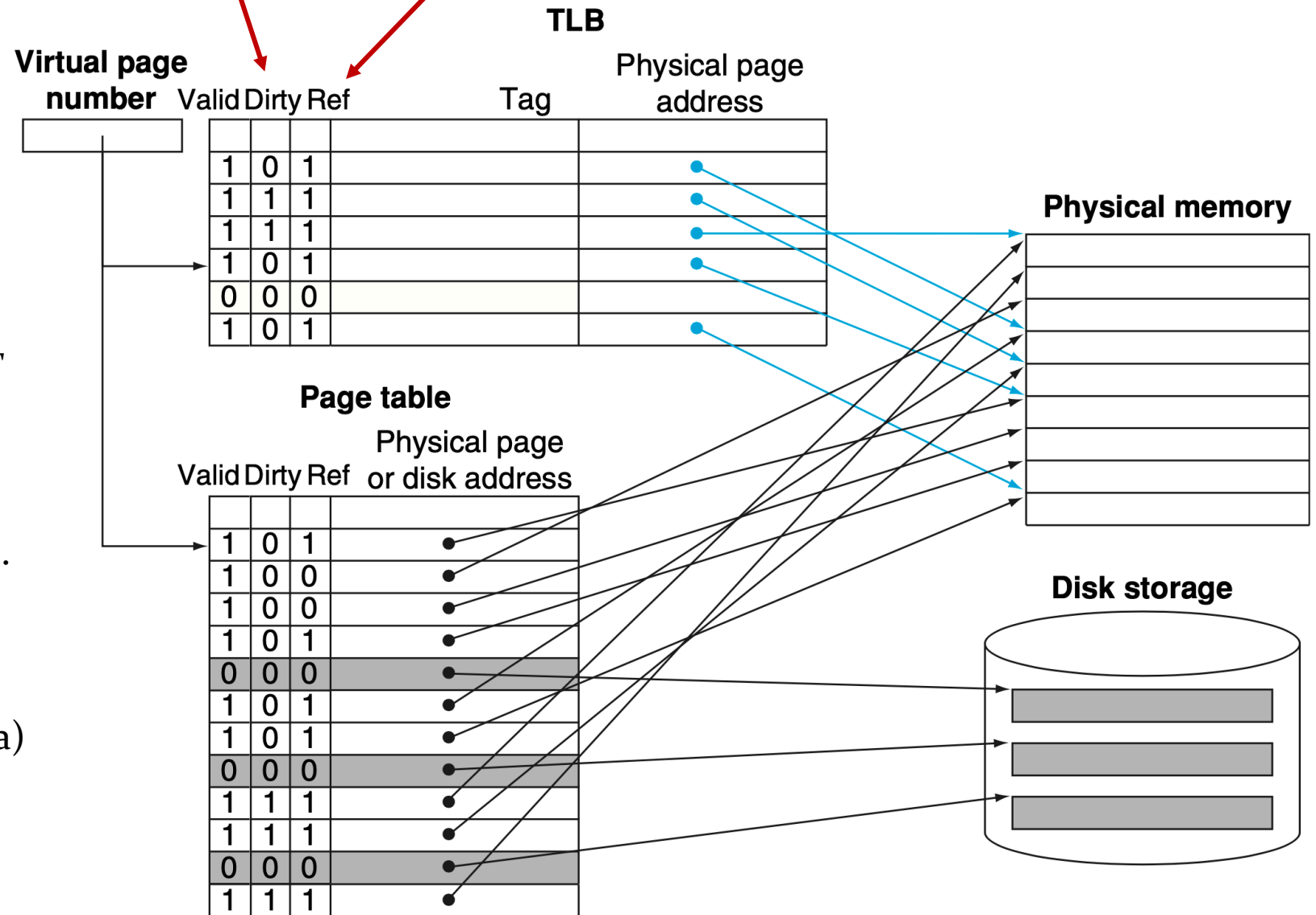
TLB holds a subset of all VA->PA translations

Hits in the TLB avoid going to the PT
Avoid 1 memory access!

Use high associativity,
we really don't want to go memory..

Note:
TLB is a cache (tag and data)
Page Table is not a cache (just data)

TLB

Virtual page number

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		
1	1	1		
1	1	1		
1	0	1		
0	0	0		
1	0	1		

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	
1	0	0	
1	0	0	
1	0	1	
0	0	0	
1	0	1	
1	0	1	
0	0	0	
1	1	1	
1	1	1	
0	0	0	
1	1	1	

Physical memory

Disk storage

TLB access process

On every memory event, access the TLB:

If TLB hit: the physical page number is used to make the physical address,

Ref bit set high. If write event: turn on dirty bit

If TLB miss: Determine if simple TLB miss or page fault

If PTE valid, processor looks up VA in PT,
loads PTE info in TLB, replays access

If page fault, invoke OS, trigger page fault event, go to disk,

create mapping in PT, load mapping into TLB, replay access

On eviction, only must write back reference and dirty bit
Mappings cannot change in the TLB!

Whole picture

Wait.. What's going on with this cache??
The data has more entries than tag array

It's a DM cache that uses offset to index.
This saves a mux
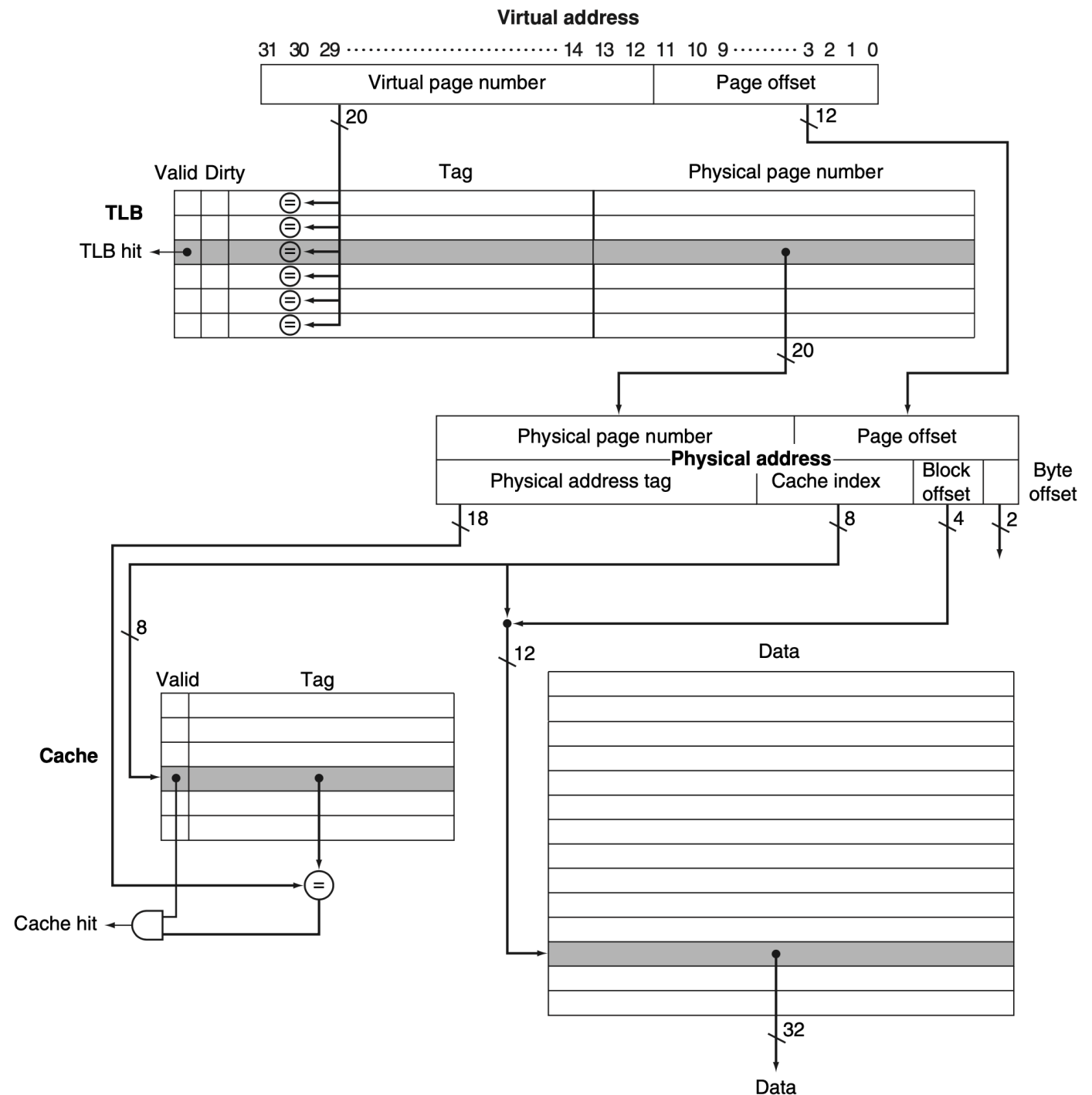
Virtual address

31 30 29 ······························ 14 13 12 11 10 9 ········ 3 2 1 0

Virtual page number	Page offset

20

12

Valid	Dirty		Tag	Physical page number

TLB

TLB hit

=
=
=
=
=
=

20

Physical page number	Page offset

Physical address

Physical address tag	Cache index	Block offset	Byte offset

18

8

4

2

8

Valid	Tag

12

Data

Cache

=

Cache hit

32

Data
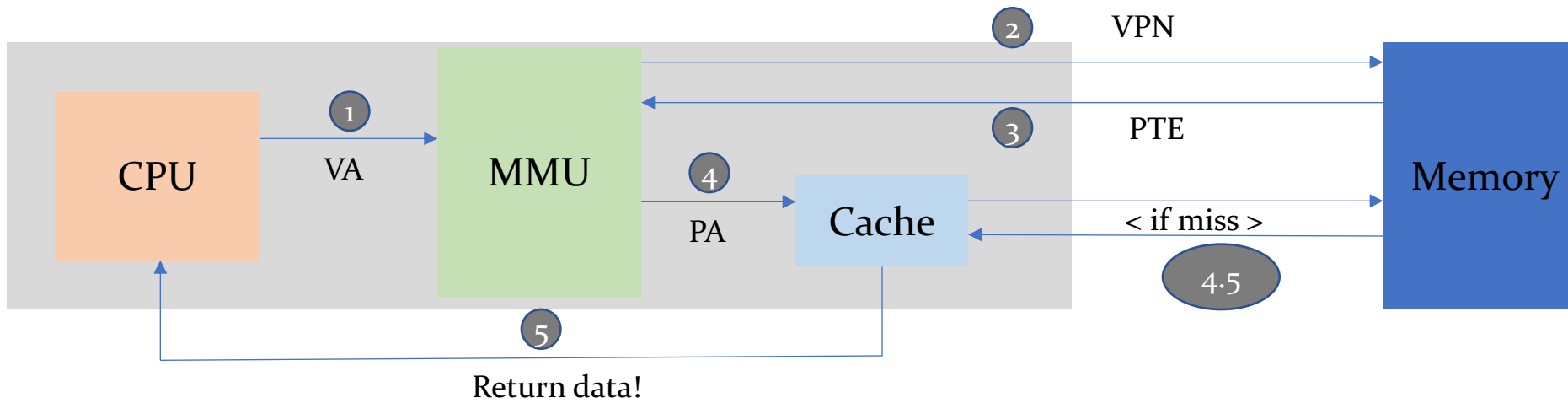
Recap: Page hit

Recap: Page fault
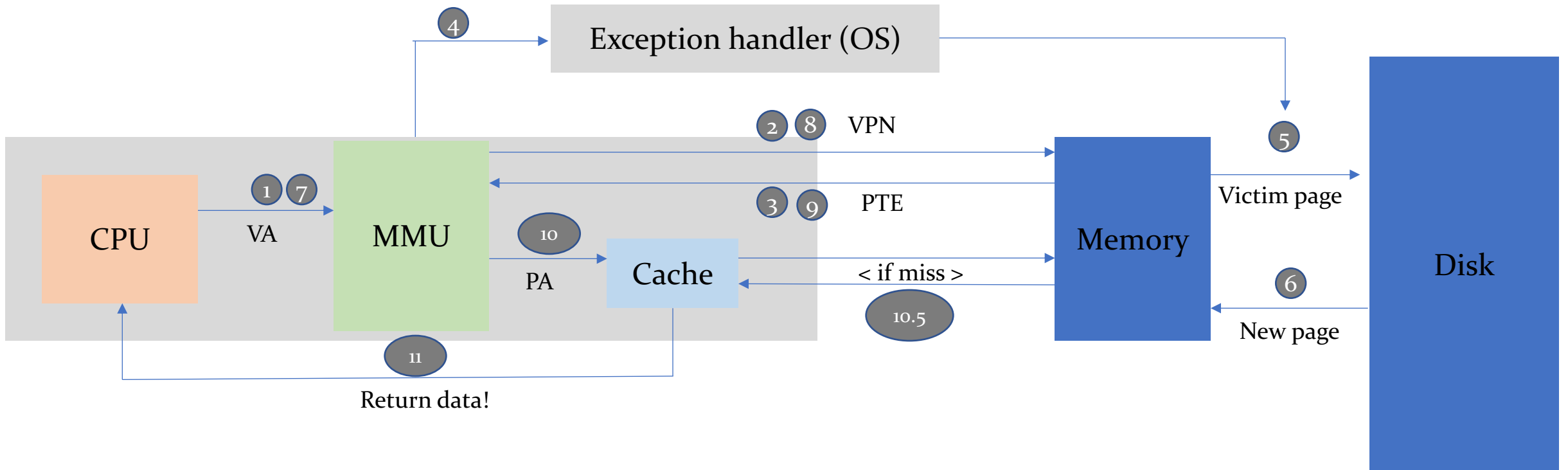
A lot of work..
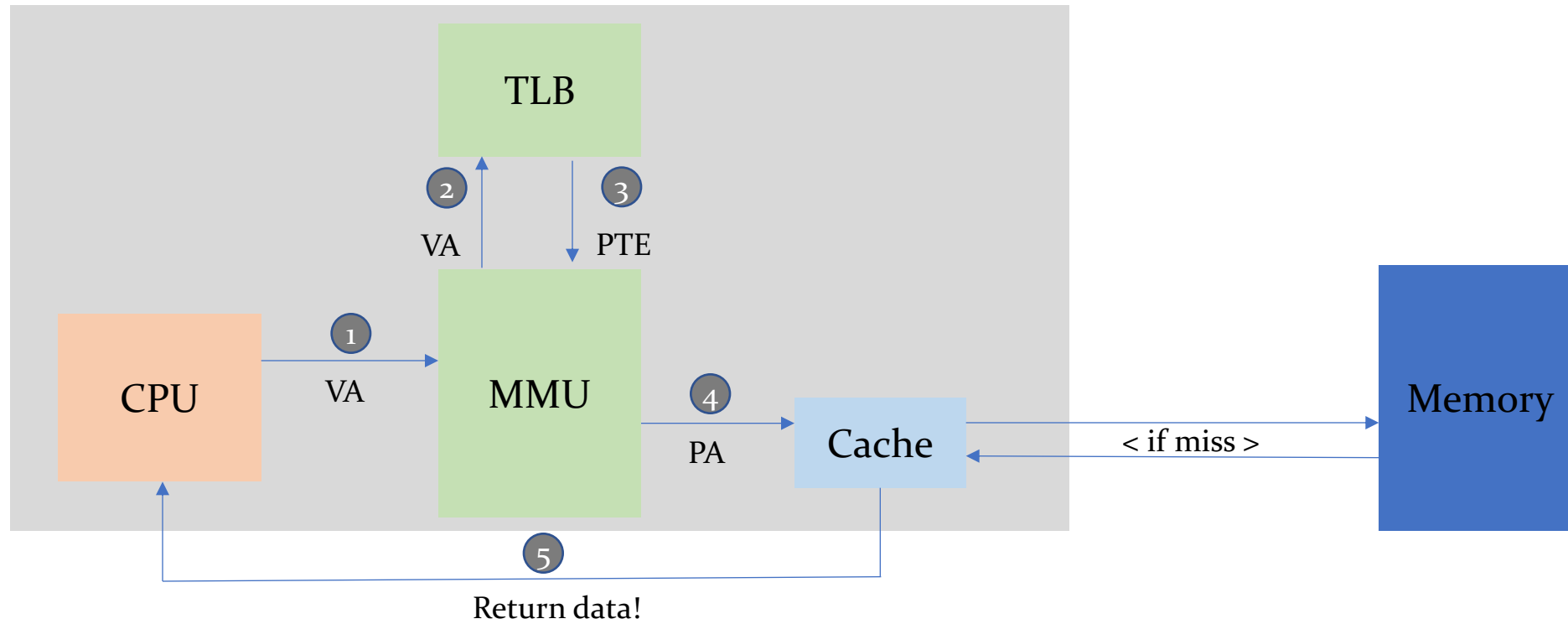
Recap: TLB hit

Recap: TLB miss
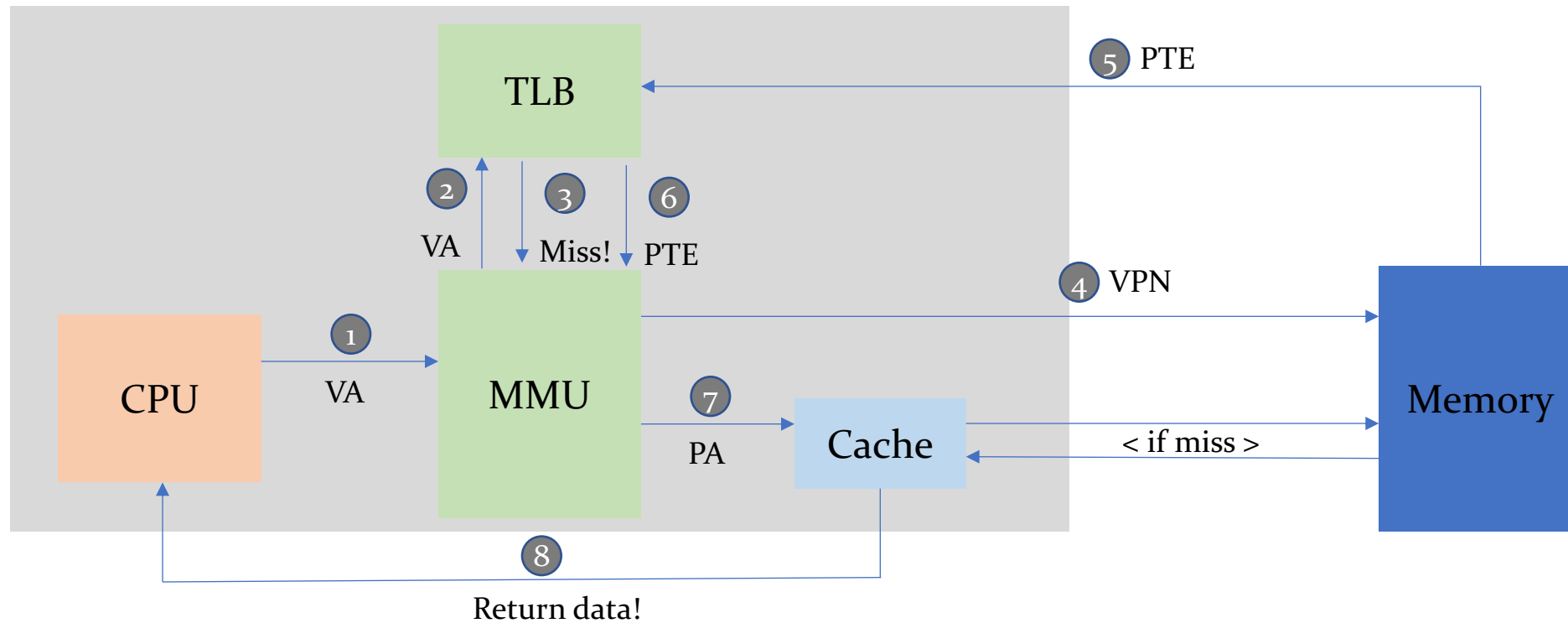
Common interview question:
"How does a load work?"

Examples!

Simple Memory System Example

Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes

13	12	11	10	9	8	7	6	5	4	3	2	1	0

VPN ← → **VPO**

Virtual Page Number   Virtual Page Offset

11	10	9	8	7	6	5	4	3	2	1	0

PPN ← → **PPO**

Physical Page Number   Physical Page Offset
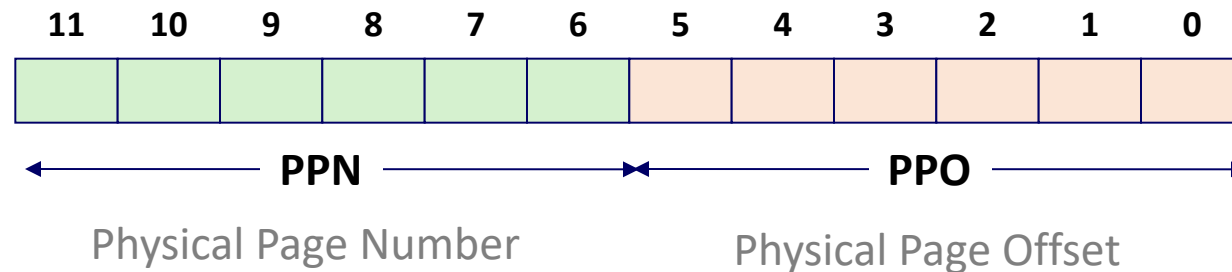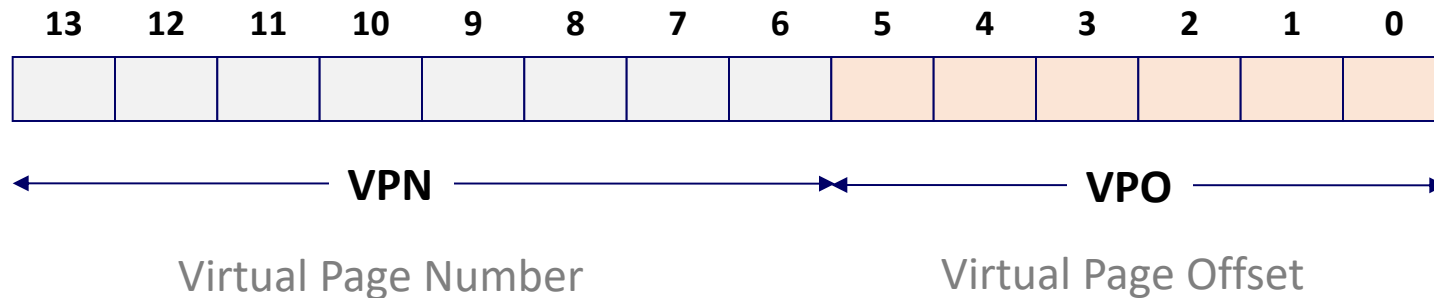
Simple Memory System Page Table

Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Simple Memory System TLB

16 entries

4-way associative

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Cache

16 lines, 4-byte block size

Physically addressed

Direct mapped

	11	10	9	8	7	6	5	4	3	2	1	0
	CT						CI				CO	
	PPN							PPO				
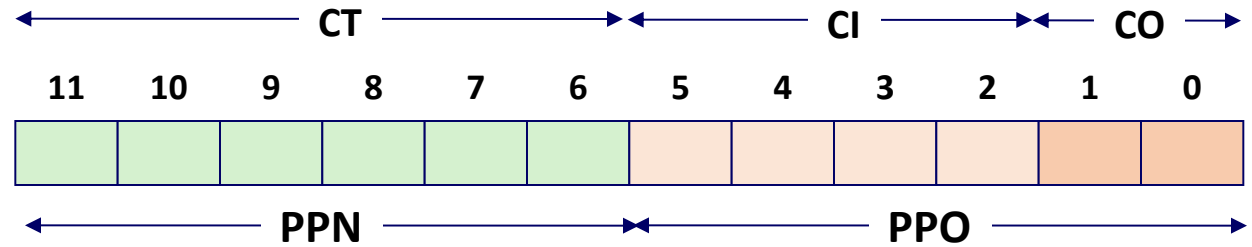
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #1

Virtual Address: `0x03D4`

13	12	11	10	9	8	7	6	5	4	3	2	1	0

VPN ___   TLBI ___ TLBT ____   TLB Hit? __   Page Fault? __   PPN: ____

Address Translation Example #1

Virtual Address: `0x03D4`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ————————————— VPO

VPN __0x0F__   TLBI __3__   TLBT __0x03__   TLB Hit? __   Page Fault? __   PPN: ____

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #1

Virtual Address: `0x03D4`

TLBT						**TLBI**							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								**VPO**					

VPN __0x0F__   TLBI __3__   TLBT __0x03__   TLB Hit? __Y__   Page Fault? __N__   PPN: __0x0D__

Address Translation Example #1

Virtual Address: `0x03D4`

	TLBT						TLBI						
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ← → VPO

VPN __0x0F__   TLBI __3__   TLBT ____ 0x03   TLB Hit? __Y__   Page Fault? __N__   PPN: __0x0D__

Physical Address

11	10	9	8	7	6	5	4	3	2	1	0

16 entry
DM cache

CO   CI   CT   Hit?   Byte

Address Translation Example #1

Virtual Address: `0x03D4`

TLBT							**TLBI**						
13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								**VPO**					

VPN __0x0F__   TLBI __3__   TLBT _____ 0x03   TLB Hit? __Y__   Page Fault? __N__   PPN: __0x0D__

Physical Address

11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	1	1	0	1	0	1	0	1	0	0

16 entry
DM cache

CO   CI   CT   Hit?   Byte

Address Translation Example #1

Virtual Address: 0x03D4

	TLBT						TLBI				VPO		

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN $\longleftrightarrow$ VPO

VPN __0x0F__   TLBI __3__   TLBT ____ 0x03   TLB Hit? __Y__   Page Fault? __N__   PPN: ____ 0x0D

Physical Address

	CT						CI				CO	

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN $\longleftrightarrow$ PPO

16 entry DM cache

CO **0**   CI **0x5**   CT **0x0D**   Hit?   Byte

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #1

Virtual Address: 0x03D4

	TLBT					TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ___ 0x0F   TLBI _3_   TLBT ____ 0x03   TLB Hit? _Y_   Page Fault? _N_   PPN: ____ 0x0D

Physical Address

	CT					CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

CO **0**   CI **0x5**   CT **0x0D**   Hit? **Y**   Byte **0x36**

Address Translation Example #2

Virtual Address: `0x0B8F`

13	12	11	10	9	8	7	6	5	4	3	2	1	0

VPN ___   TLBI ___   TLBT ____   TLB Hit? __   Page Fault? __   PPN: ____

Address Translation Example #2

Virtual Address: `0x0B8F`

VPN ___ 0x2E   TLBI _2_   TLBT ___ 0x0B   TLB Hit? __   Page Fault? __   PPN: ____

TANDON SCHOOL
OF ENGINEERING

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #2

Virtual Address: `0x0B8F`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	1	**0**	1	**1**	**1**	0	**0**	**0**	1	**1**	**1**	**1**

VPN ___ **0x2E**   TLBI _2_   TLBT **0x0B**   TLB Hit? **N**   Page Fault? __   PPN: ____

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Page table

We'll assume
Valid bit is zero

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #2

Virtual Address: `0x0B8F`

	TLBT						TLBI	

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	1	**0**	1	**1**	**1**	0	**0**	**0**	1	**1**	**1**	**1**

VPN ————————————— VPO

VPN **0x2E**   TLBI **2**   TLBT **0x0B**   TLB Hit? **N**   Page Fault? **Y**   PPN: **TBD**

Address Translation Example #3

Virtual Address: 0x0020

TLBT ◄──────────────────► TLBI ◄──►

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	0	0	0	0	1	0	**0**	0	**0**	**0**

◄────────────────── VPN ──────────────────► ◄────────────── VPO ──────────────►

VPN **0x00**   TLBI _0_   TLBT **0x00**   TLB Hit? __   Page Fault? __   PPN: ____

Physical Address

◄──────────── CT ────────────► ◄──────── CI ────────► ◄── CO ──►

11	10	9	8	7	6	5	4	3	2	1	0

◄──────────── PPN ────────────► ◄──────────── PPO ────────────►

CO___   CI___   CT ____   Hit? __   Byte: ____

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #3

Virtual Address: 0x0020

TLBT ← → TLBI →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	0	0	0	0	1	0	**0**	0	**0**	**0**

← VPN → ← VPO →

VPN **0x00**   TLBI 0   TLBT **0x00**   TLB Hit? N   Page Fault? __   PPN: ____

Physical Address

CT ← → CI → ← CO →

11	10	9	8	7	6	5	4	3	2	1	0

← PPN → ← PPO →

CO___   CI___   CT ____   Hit? __   Byte: ____

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #3

Virtual Address: 0x0020

	TLBT						TLBI						
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN **0x00**   TLBI **0** TLBT **0x00**   TLB Hit? **N**   Page Fault? **N**   PPN: **0x28**

Physical Address

	CT						CI			CO	
11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

CO **0**   CI **0x8**   CT **0x28**   Hit? __   Byte: ____

Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #3

Virtual Address: `0x0020`

TLBT								**TLBI**					
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	0	0	0	0	1	0	**0**	0	**0**	**0**
VPN								**VPO**					

VPN __0x00__   TLBI __0__   TLBT __0x00__   TLB Hit? __N__   Page Fault? __N__   PPN: __0x28__

Physical Address

CT						**CI**				**CO**	
11	10	9	8	7	6	5	4	3	2	1	0
1	**0**	1	0	**0**	0	1	0	**0**	0	**0**	**0**
PPN						**PPO**					

CO**0**__   CI__**0x8**__   CT __**0x28**__   Hit? __**N**__   Byte: __< from mem >__

If you want to know more check out

Dynamic scheduling

Problem 1


```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

Program execution order (in instructions)

lw $10, 20($1)   IM   Reg   ALU   DM   Reg

sub $11, $2, $3   IM   Reg   ALU   DM   Reg

add $12, $3, $4   IM   Reg   ALU   DM   Reg

lw $13, 24($1)   IM   Reg   ALU   DM   Reg

add $14, $5, $6   IM   Reg   ALU   DM   Reg

Bonus question: What do we not like about this?

Resources available and not used!
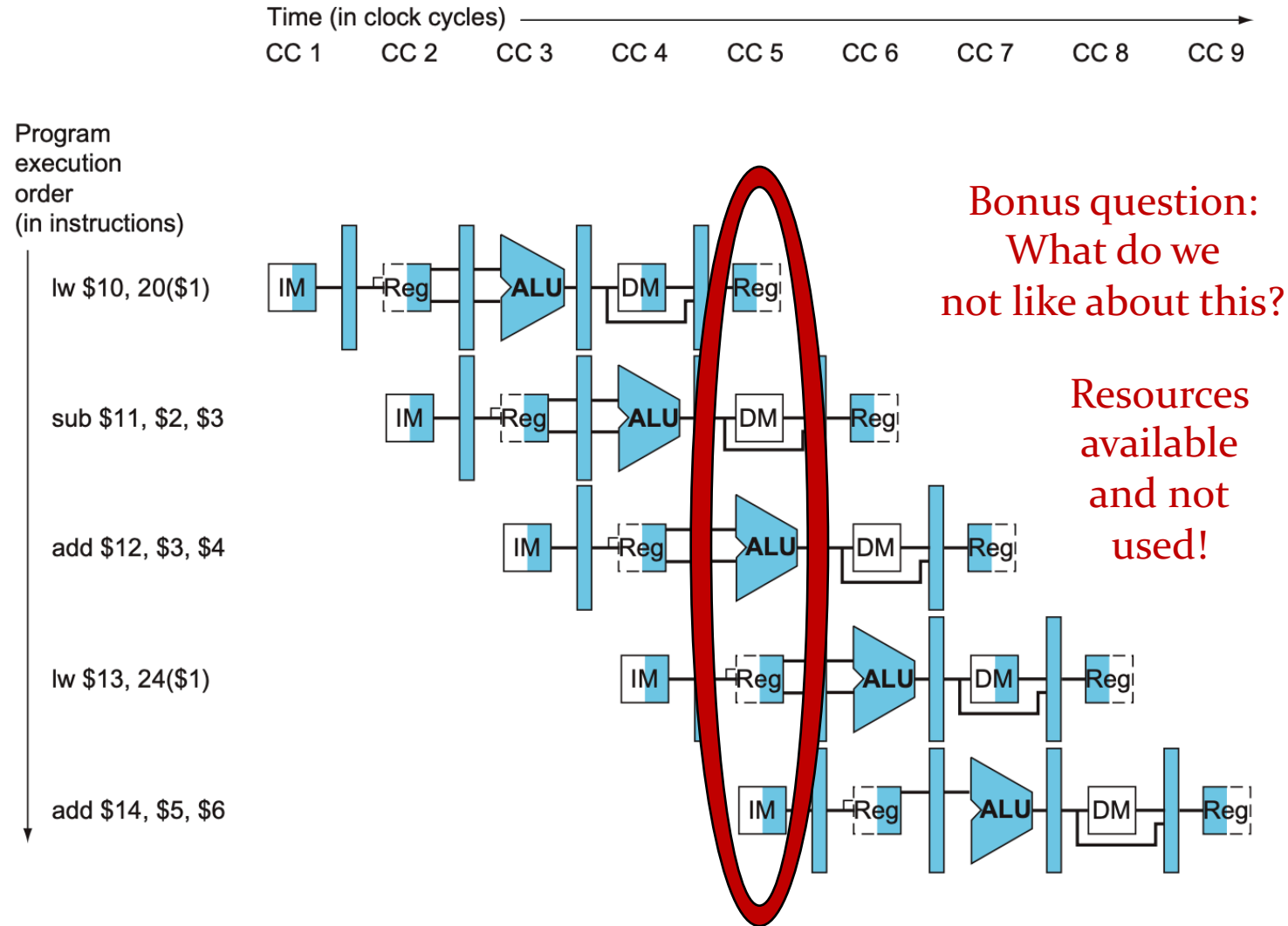
Problem 2: Unnecessary stalls (causes underutilization)

Add r3, r2, r1

Lw r2, 0(r0)

Sub r4, r2, r1

Add r6, r7, r8

CPI?

5/4 = 1.25

Add r3, r2, r1

Lw r2, 0(r0)

Add r6, r7, r8

Sub r4, r2, r1

CPI?

4/4 = 1.0!

Same functionality, better performance!

Solution: Data flow!

$$a := x + y$$
$$b := a \times a$$
$$c := 4 - a$$

Instructions dependent, even if results aren't.

This is how you write programs

It's very restrictive
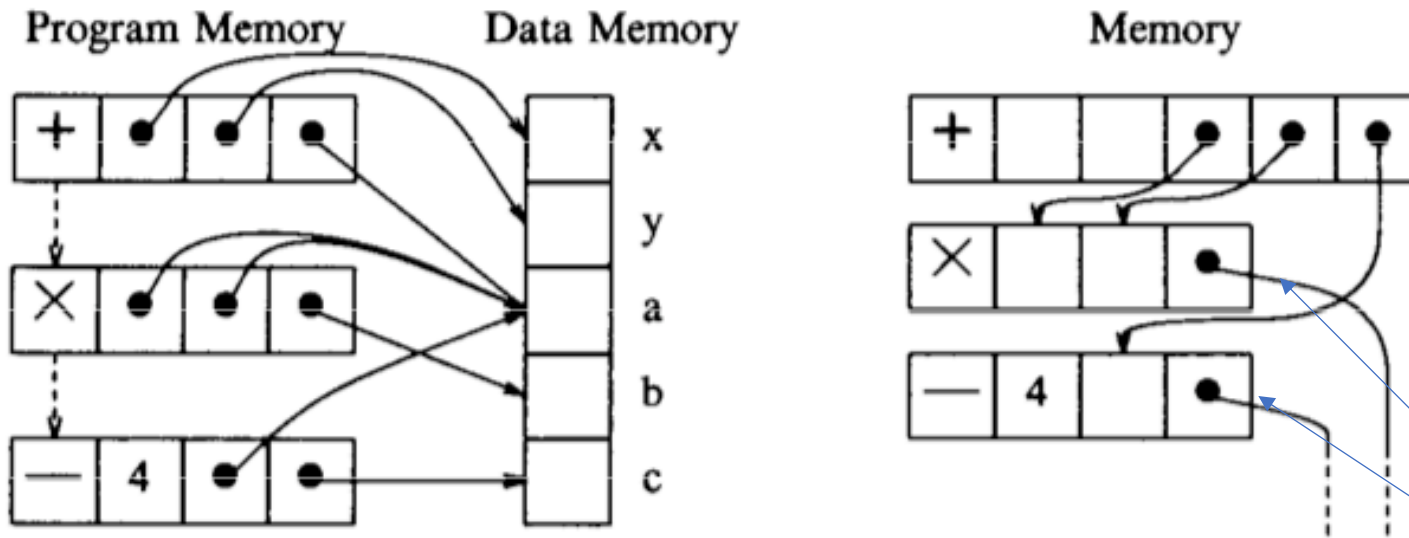
Only tracks real dependencies. "Which instructions need what I'm computing?"

This is what OoO machines do!

What can we say about these two?

Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

Parallel instructions

"Dataflow Machine Architecture" Heen, 1986.

Instruction-Level Parallelism (ILP)

Fine-grained parallelism

A measure of inter-instruction dependency in an app
- ILP assumes a unit-cycle operation, infinite resources, prefect frontend
  - Theoretically, how many instructions could I process per cycle?
- ILP != IPC
- IPC = # instructions / # cycles
- ILP is the upper bound of attainable IPC

Enabled and improved by RISC
- More ILP of RISC over CISC does not imply a better overall performance
  - CISC can be implemented like RISC

Limited by dependencies

Back to hazards…

Control: Where do I go next?

    - Caused by branch instructions

Data: There's more!

    - RAW: Read-After-Write

    - WAW: Write-After-Write

    - WAR: Write-After-Read

Why am I just telling you this now?

When we dynamically schedule, the program order is broken.

Must guarantee there are no artifacts!

Hazards limit ILP.

WAW:

    Add r2, r2, r2
    sub r2, r2, r2

WAR:

Add r2, r1, r1   RAW+WAR
Sub r1, r2, r2
WAW   Add r2, r3, r3   WAR
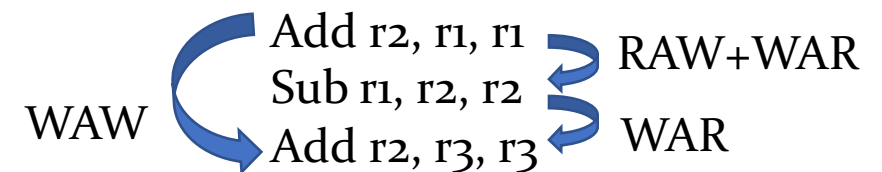
True/False dependencies and ILP

True dependency forces "sequentiality"

ILP = 3/3 = 1
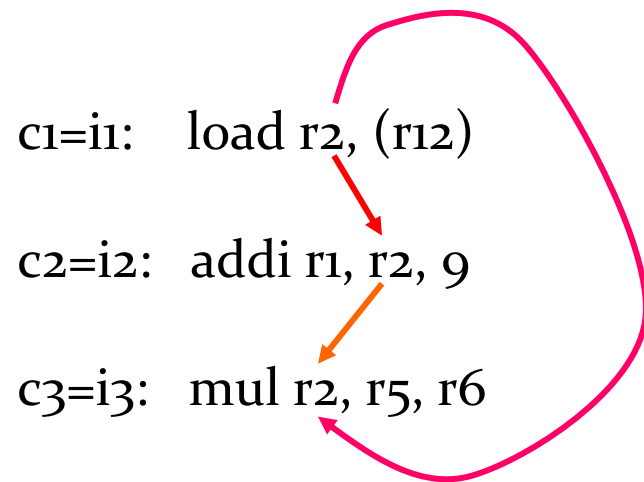
False dependency removed

ILP = 3/2 = 1.5

$c_1=i_1$:   load r2, (r12)

$c_2=i_2$:   addi r1, r2, 9

$c_3=i_3$:   mul r2, r5, r6

i_1:   load r2, (r12)

i_2:   addi r1, r2, 9

i_3:   mul r8, r5, r6

c_1:   load r2, (r12)

c_2:   addi r1, r2, 9   mul r8, r5, r6

Exploiting ILP

- Control speculation

       Branch prediction!

- Dynamic scheduling   today

       Reschedule program ordering of instructions

- Register renaming

       Break false dependencies

       Use uArch to provide illusion of "more" registers..

- Dynamic memory disambiguation

       Discuss later

Step 1: Construct data flow graph

i1: r2   = 4(r22)
i2: r10 = 4(r25)
i3: r10 = r2 + r10
i4: 4(r26) = r10
i5: r14 = 8(r27)
i6: r6   = (r22)
i7: r5   = (r23)
i8: r5   = r6 – r5
i9: r4   = r14 * r5
i10: r15 = 12(r27)
i11: r7   = 4(r22)
i12: r8   = 4(r23)
i13: r8   = r7 – r8
i14: r8   = r15* r8
i15: r8   = r4 – r8
i16: (r28) = r8
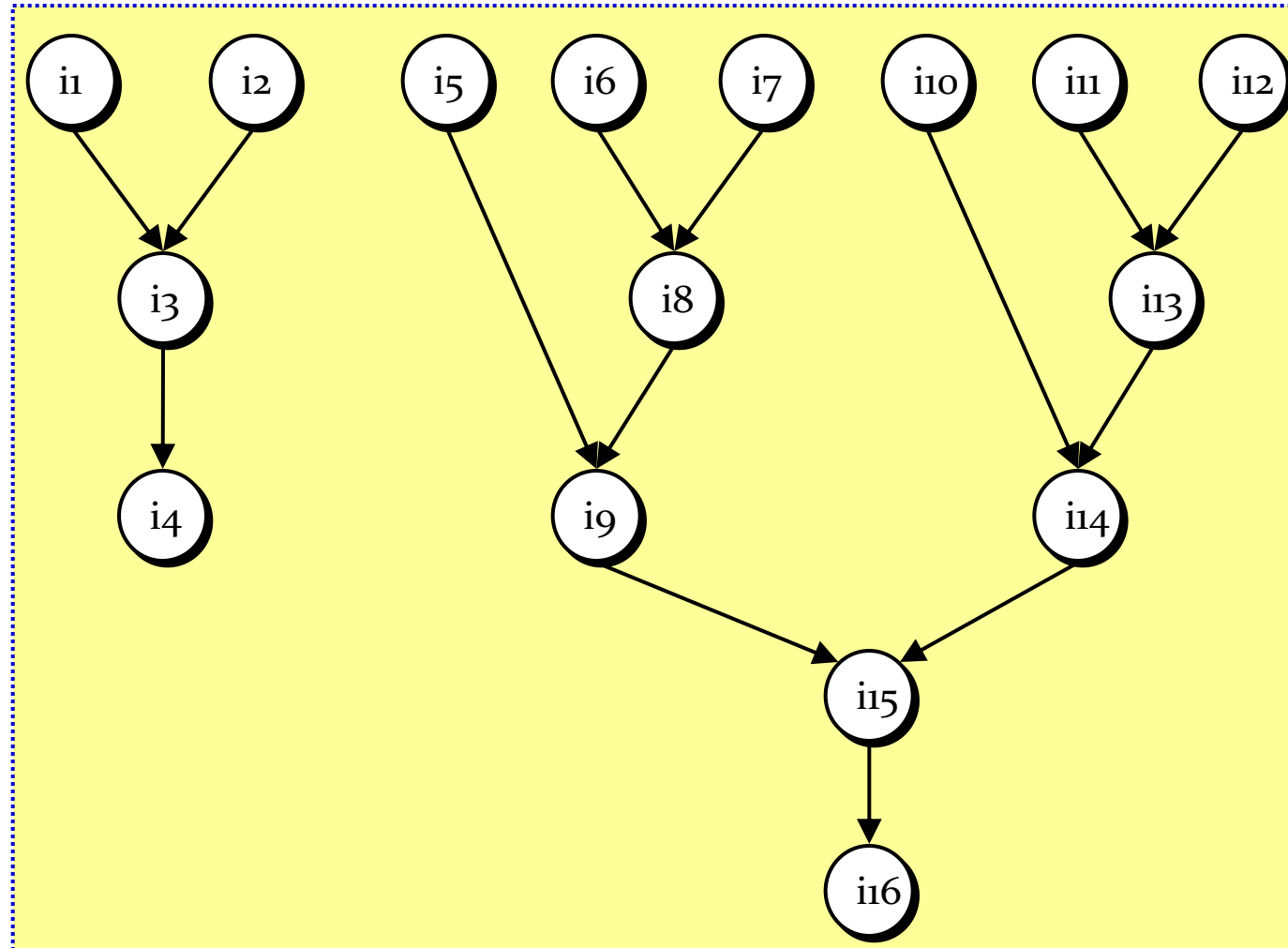
This is ideal.
Why not
possible?

Data Flow Graph (or Data Dependency Graph, DDG)

Step 1: Construct data flow graph

i1: r2   = 4(r22)
i2: r10 = 4(r25)
i3: r10 = r2 + r10
i4: 4(r26) = r10
i5: r14 = 8(r27)
i6: r6   = (r22)
i7: r5   = (r23)
i8: r5   = r6 – r5
i9: r4   = r14 * r5
i10: r15 = 12(r27)
i11: r7   = 4(r22)
i12: r8   = 4(r23)
i13: r8   = r7 – r8
i14: r8   = r15* r8
i15: r8   = r4 – r8
i16: (r28) = r8
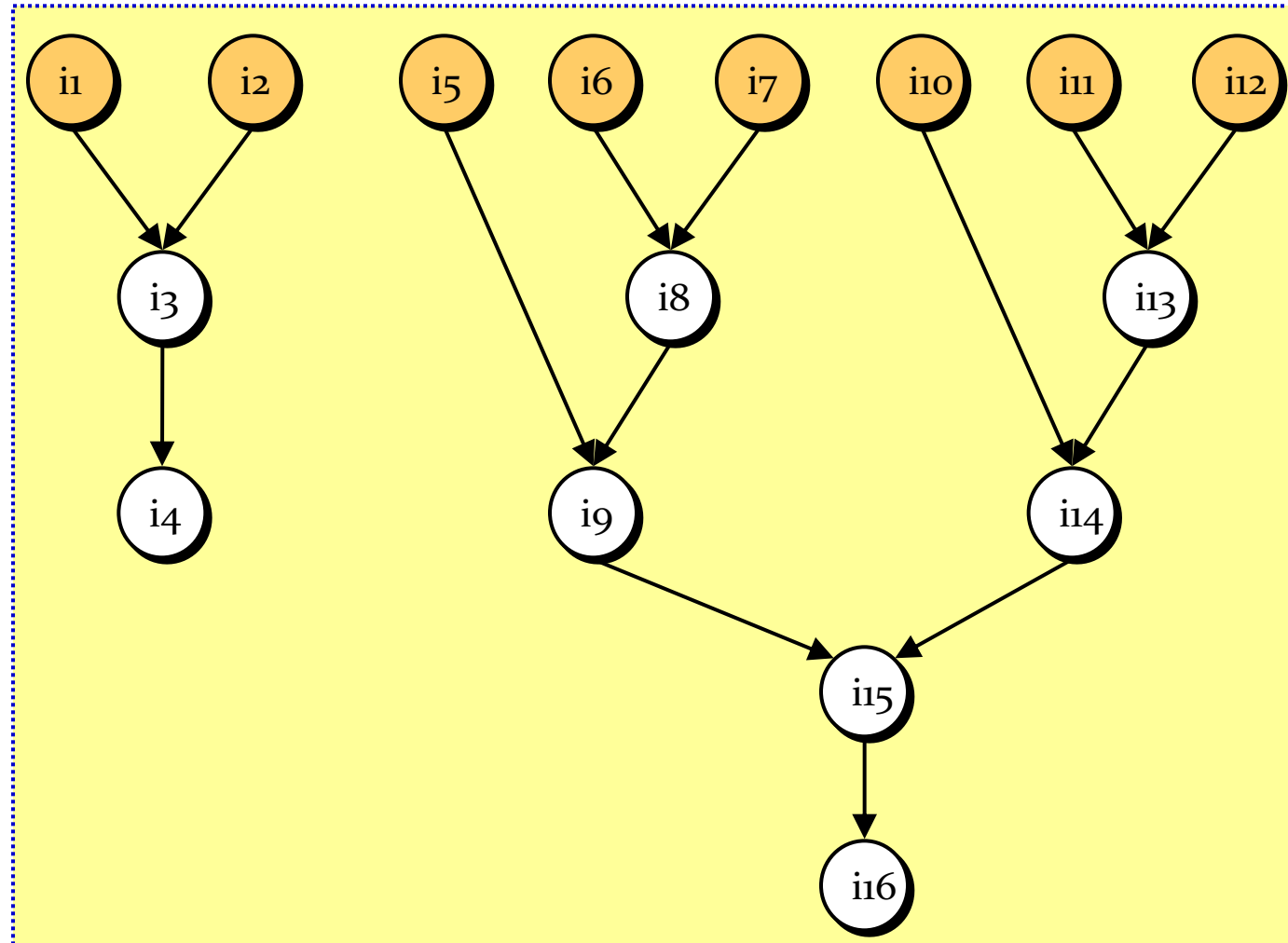
This is ideal.
Why not
possible?

ILP "Windows"

ILP = 1

$$R5 = 8(R6)$$

$$R7 = R5 - R4$$

$$R9 = R7 * R7$$

ILP = ?

ILP = 1.5

$$R15 = 16(R6)$$

$$R17 = R15 - R14$$

$$R19 = R15 * R15$$

NYU | TANDON SCHOOL OF ENGINEERING

Bigger window

R5 = 8(R6)

R7 = R5 – R4

R9 = R7 * R7

R15 = 16(R6)

R17 = R15 – R14

R19 = R15 * R15

Bigger => better ILP!

C1: R5 = 8(R6)   R15 = 16(R6)

C2: R7 = R5 – R4   R17 = R15 – R14   R19 = R15 * R15

R9 = R7 * R7

ILP = 6/3 = 2 better than 1 and 1.5

Larger window gives more opportunities

But what limits the window?

Impact of registers on ILP

R1 = 8(R0)

R3 = R1 – 5

R2 = R1 * R3

24(R0) = R2

R1 = 16(R0)

R3 = R1 – 5

R2 = R1 * R3

32(R0) = R2

When only 4 registers available

ILP =

Impact of registers on ILP

$R1 = 8(R0)$

$R3 = R1 - 5$

$R2 = R1 * R3$

$24(R0) = R2$

$R1 = 16(R0)$

$R3 = R1 - 5$

$R2 = R1 * R3$

$32(R0) = R2$

When only 4 registers available

$ILP = \#inst/\#cycles = 8/8 = 1$

Impact of registers on ILP

R1 = 8(R0)

R3 = R1 – 5

R2 = R1 * R3

24(R0) = R2

R1 = 16(R0)

R3 = R1 – 5

R2 = R1 * R3

32(R0) = R2

When more (8) registers available, rewrite code to improve ILP

ILP =

Impact of registers on ILP

When more (8) registers available, rewrite code to improve ILP

R1 = 8(R0)

R3 = R1 – 5

R2 = R1 * R3

24(R0) = R2

R5 = 16(R0)

R6 = R5 – 5

R7 = R5 * R6

32(R0) = R7

→

R1 = 8(R0)   R5 = 16(R0)

R3 = R1 – 5   R6 = R5 – 5

R2 = R1 * R3   R7 = R5 * R6

24(R0) = R2   32(R0) = R7

ILP = #inst / # cycles = 8/4 = 2!

Step 2: Execute nodes!

When can an instruction execute?
- All source operands are ready
- Execution unit available
- Destination is ready (to be written)

This is dynamic scheduling,
        we no longer follow the program order of instructions,

        instead use data flow to execute computation

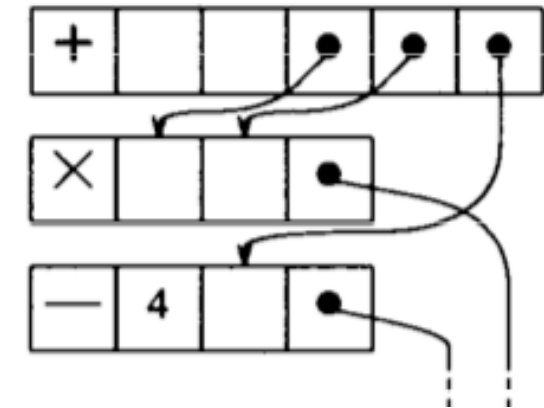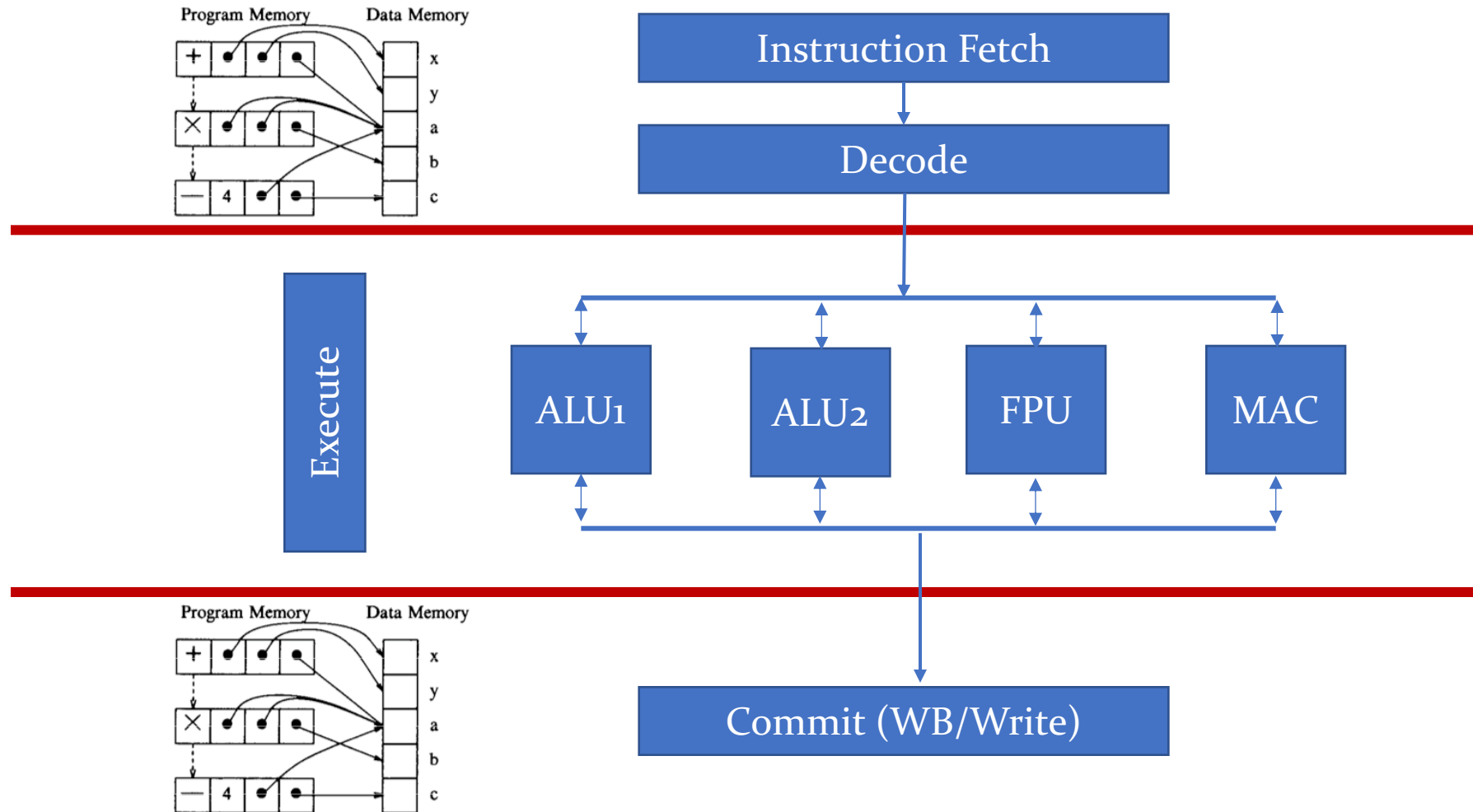Dynamic scheduling enables Out-of-Order execution

OoO General Scheme

Call this:
"Restricted dataflow"

Program Memory   Data Memory

Instruction Fetch

Decode

Execute

ALU1   ALU2   FPU   MAC

Commit (WB/Write)

Program Memory   Data Memory

Next time:
Tomasulo!

NYU TANDON SCHOOL OF ENGINEERING

OoO (or "O3" or OOO) execution

OoO execution ≡ out-of-order computation

OoO execution ≠ out-of-order retirement (commit)

Commit => when instruction updates the architectural state

No (speculative) instruction allowed to commit
until it is confirmed on the right path

Fetch and decode (i.e., front-end)

are still done in the program order   Why?