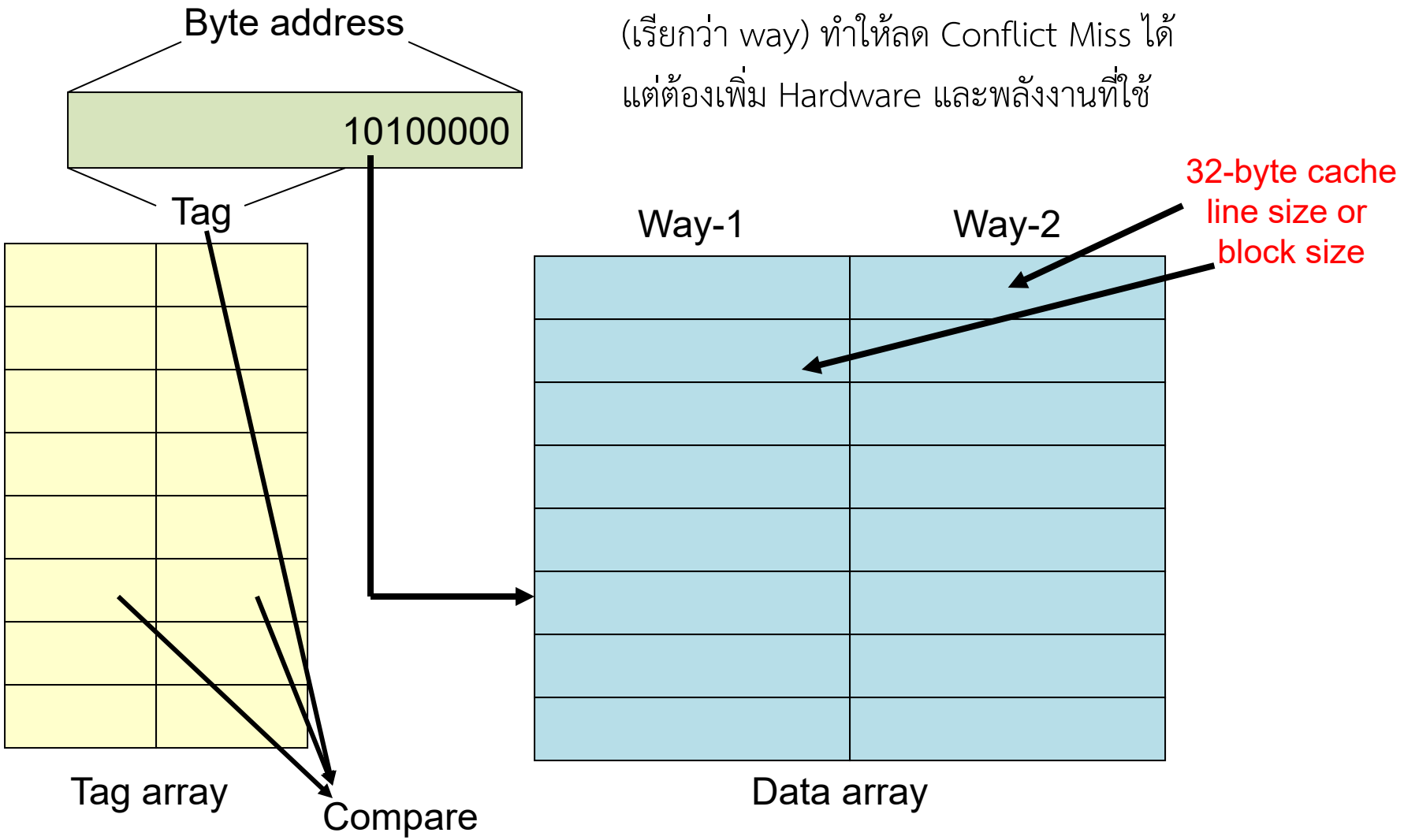
01076114
องค์ประกอบและสถาปัตยกรรมคอมพิวเตอร์
Computer Organization and Architecture

Memory Hierarchy

Set Associative

- จากที่ผ่านมา จะเห็นได้ว่าแคชในแบบ Direct-Mapped จะมีปัญหาเรื่อง Conflict Miss คือ มี 2 address ที่ map ลงใน cache ที่เป็น block เดียวกัน
- ในบางกรณีเช่น สมมติว่าโปรแกรม A มีการรัน 1,000 ลูป บังเอิญว่าข้อมูลที่ใช้ 2 ข้อมูล อยู่ในตำแหน่ง ที่ map ลงใน cache ที่เป็น block เดียวกันพอดี ก็จะทำให้เกิด cache miss จำนวน 2,000 ครั้ง ซึ่งจะทำให้โปรแกรมทำงานช้าลงอย่างมาก
- จึงได้มีการเสนอแนวคิดในการปรับปรุง cache โดยเพิ่มจำนวนชุดของ cache เข้าไป เรียกว่า Set Associative

Set Associative

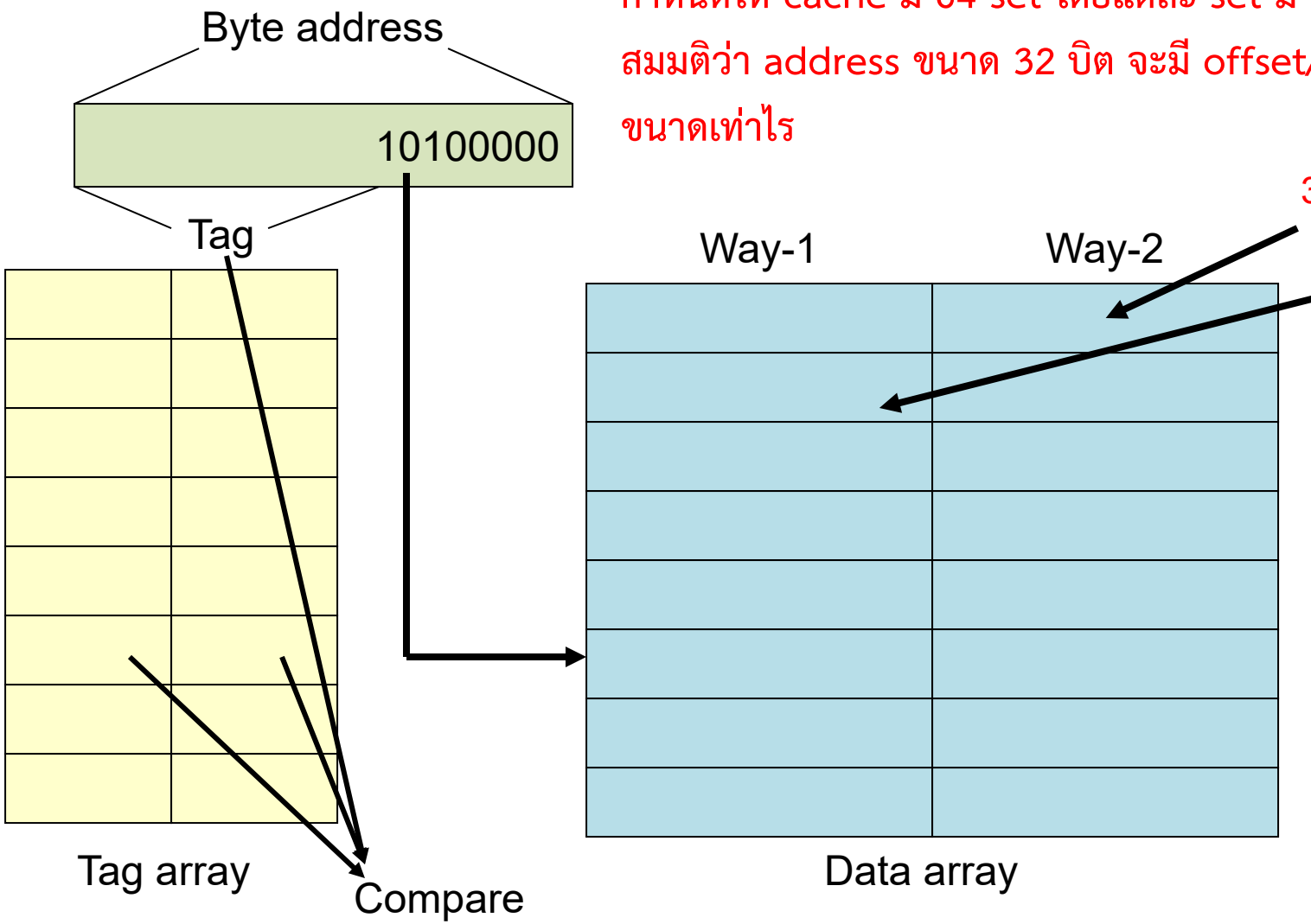


Set associativity เพิ่มตำแหน่งละ n set (เรียกว่า way) ทำให้ลด Conflict Miss ได้ แต่ต้องเพิ่ม Hardware และพลังงานที่ใช้

Set Associative

- จะเห็นว่า เราจะต้องเพิ่มขนาดของ cache และ tag เป็น 2 เท่าจากเดิม หรือ หากขนาดของ cache มีขนาดเท่าเดิม จำนวน block ก็จะต้องลดลงครึ่งหนึ่ง
- จาก slide 28 : line size ขนาด 32 ไบต์ ดังนั้น offset ขนาด 5 บิต
- สมมติจำนวน set = 256 ดังนั้น index = 8 บิต
- หากเป็น address ขนาด 32 บิต tag จะมีขนาด 32-(5+8) = 19 บิต
- ขนาดของ Cache (ไม่รวม tag) = 256 x 32 x 2 = 16 KB
- ขนาดของ Cache (รวม tag) = 256 x 32 x 8 x 2 + 256 x 19 x 2 = 140 Kbit
- การทำงานของแคชแบบนี้ 1) การแคชใน way ใดว่าง จะเข้าใช้งาน 2) หากแคชเต็มทั้ง 2 way จะมีการนำแคชใน way หนึ่งออกไป โดยวิธีการเลือก way ออกไปจะกล่าวถึงอีกครั้ง

Example : Set Associative



กำหนดให้ cache มี 64 set โดยแต่ละ set มี 64 ไบต์ 4 way

สมมติว่า address ขนาด 32 บิต จะมี offset/index/tag ขนาดเท่าไร

32-byte cache line size or block size

1 set = 64 ไบต์

ดังนั้น offset = 6 b

มี 64 set

ดังนั้น index = 6 b

Tag = 32-6-6

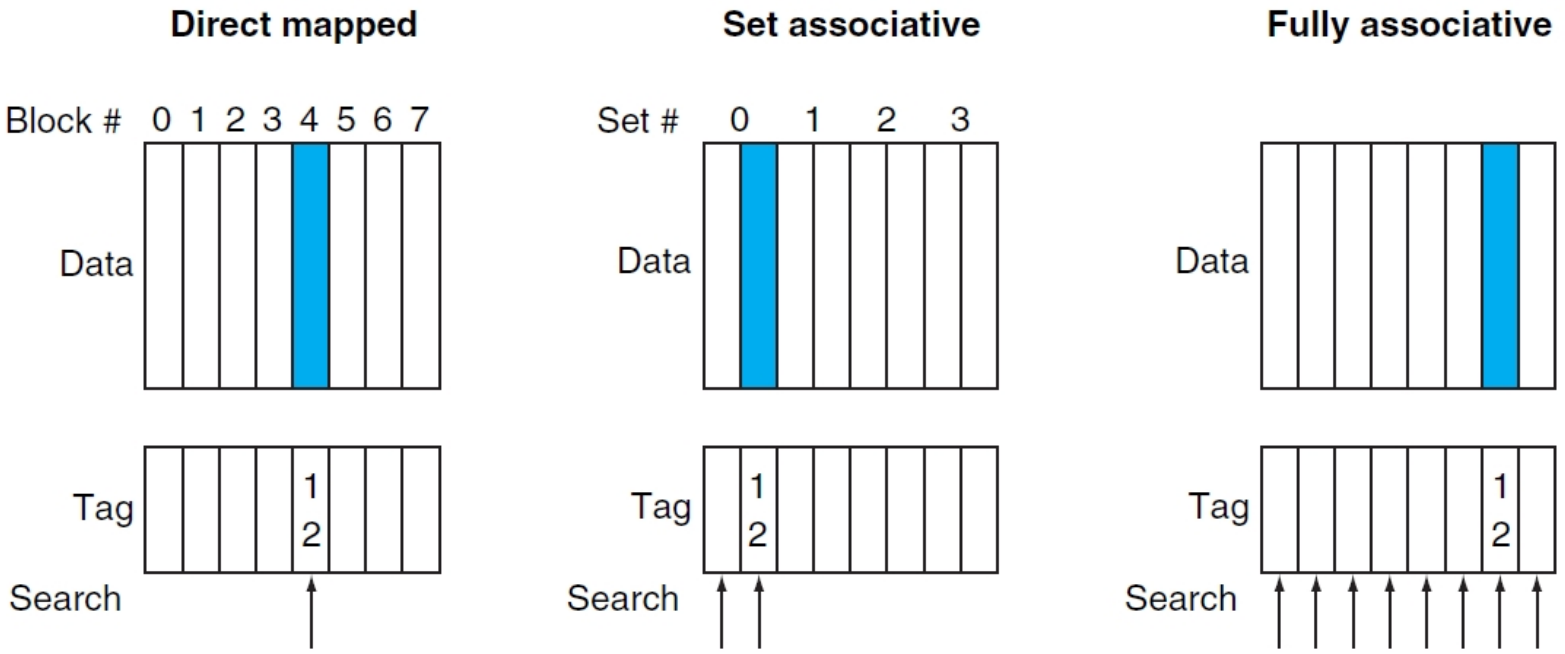
= 20 bit

Exercise

- แคชขนาด 32 KB เป็นแบบ 4 way set-associative โดยมี line size =32 B (Address ขนาด 32 บิต)
  - มีทั้งหมดกี่ set
  - ต้องใช้ Index, offset, tag ขนาดกี่บิต
  - Tag array ต้องมีขนาดเท่าไร

Fully associative

- นอกเหนือจากแคชแบบ Direct mapped และ Set associative แล้ว ยังมีแคชที่เรียกว่า Fully associative โดยแคชประเภทนี้จะไม่มีฟิลด์ที่ใช้เป็น Index โดยนอกเหนือจากฟิลด์ที่เป็น offset แล้วที่เหลือจะเป็น Tag ทั้งหมด เช่น address ขนาด 32 บิต หาก line size ขนาด 32 บิต ขนาดของ tag จะเท่ากับ 32-5 = 27 บิต ดังนั้นตำแหน่งของแคช จะใช้ตำแหน่งไหนก็ได้
- ข้อดีของแคชแบบนี้ คือ จะ miss น้อยที่สุด ข้อเสียคือ เสียเวลาในการ search



Example

- สมมติว่าเรามีแคชขนาดเล็ก จำนวน 3 แคช โดยแต่ละแคชมีเพียง 4 block แต่ละ block มีขนาด 4 ไบต์ โดยเป็นแบบ Direct-mapped, 2-way set-associative และ Fully associative
- กำหนดให้ block address ที่จะอ้างถึง คือ 0, 8, 0, 6, 8
- กรณีของ Direct-mapped เนื่องจากมีเพียง 3 ค่า คือ 0, 6, 8 เมื่อนำมา mod 4 จะได้ cache block ตามตาราง

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

- และเมื่อนำมาใส่ในตารางแคช ตามลำดับการอ้างอิง จะได้ตามตารางนี้ (สีฟ้าหมายถึง new entry) จะเห็นว่าในกรณีนี้ Direct-mapped cache จะเกิด cache miss ทุกครั้ง

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Example

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

- สำหรับแคชแบบ 2-way Set-associative เนื่องจากแคชมีขนาดเท่าเดิม คือ 16 ไบต์ เมื่อนำมาใช้เป็นแบบ 2-way จำนวน set จะลดลงครึ่งหนึ่ง จึงเหลือเพียง 2 set ทำให้ Cache Set ที่อ้างอิงจะเท่ากับ 0 ทั้งหมด (ตามตาราง)
- อย่างไรก็ตาม แม้ Cache Set จะเท่ากับ 0 ทั้งหมด แต่เนื่องจากแต่ละ set จะมี 2 way คือ มีแคช 2 ชุด ดังนั้นเมื่อนำการอ้างอิงมาใส่ตาราง cache ตามลำดับ จะได้ตารางนี้

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

- จะเห็นว่าเกิด cache miss จำนวน 4 ครั้ง และเกิด cache hit จำนวน 1 ครั้งซึ่งดีกว่าแบบ Direct-Mapped สำหรับการเลือก block ที่จะนำออก (บรรทัดที่ 4) จะใช้แบบ LRU (กล่าวถึงต่อไป)

Example

- สำหรับกรณีของ Fully associative เนื่องจากทุกการอ้างอิง สามารถเก็บลงในบล็อกใดก็ได้ จึงมีการอ้างอิงแสดงตามรูป

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

- จะเห็นว่าการ Miss เกิดขึ้นน้อยที่สุดเมื่อเทียบกับแคชอีก 2 แบบ

Exercise

- แคชขนาด 8 KB เป็นแบบ 2 way set-associate โดยมี block size =16 Byte
- Address ของหน่วยความจำมีขนาด 32 บิต
- มีการอ้างหน่วยความจำดังนี้ (16 บิตแรกเป็น 0)
  - 0x1000, 0x1004, 0x1010, 0x11c0, 0x2000, 0x2003, 0x3005, 0x4004, 0x3f00, 0x2004, 0x1004
- ให้แสดงว่าครั้งไหน Hit ครั้งไหน Miss พร้อมแสดงตารางการใช้แคช (ให้ใช้ cache replacement แบบ LRU)

Exercise

- Cache ขนาด 8 KB แต่ละ block มีขนาด 16 ไบต์ = 8192 / 16 = 512 บล็อก
- เนื่องจากเป็นแบบ 2 way จึงมี 256 set ดังนั้น Offset = 4 บิต / index = 8 บิต / tag = 20 บิต

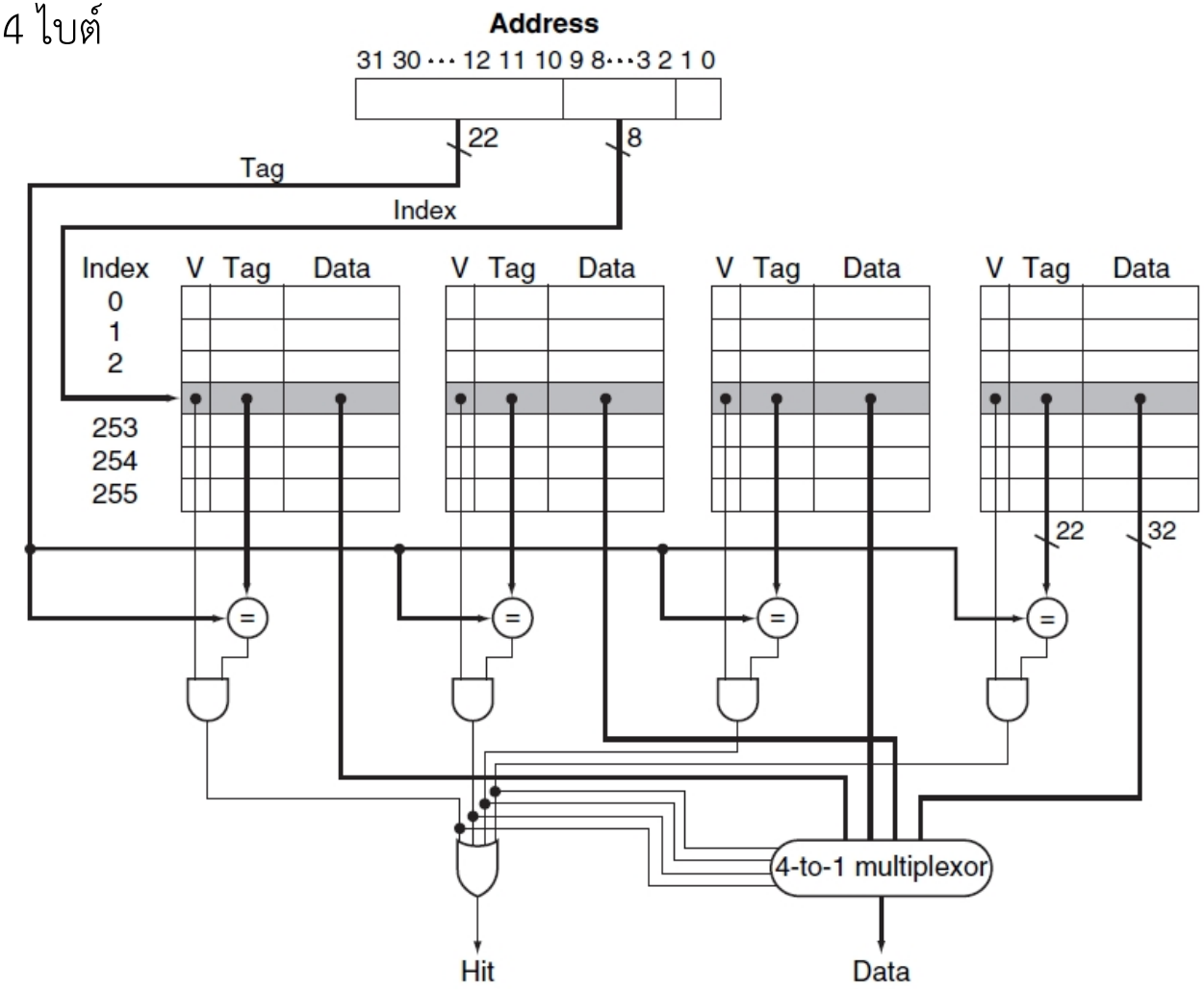
Address	Tag	Index	Way	Hit/Miss	Type
0x1000	0001	0000 0000	0	Miss	Compulsory
0x1004	0001	0000 0000	0	Hit	
0x1010	0001	0000 0001	0	Miss	Compulsory
0x11c0	0001	0001 1100	0	Miss	Compulsory
0x2000	0010	0000 0000	1	Miss	Compulsory
0x2003	0010	0000 0000	0	Hit	
0x3005	0011	0000 0000	0	Miss	Conflict
0x4003	0100	0000 0000	1	Miss	Conflict
0x4005	0100	0000 0000	1	Hit	
0x4006	0100	0000 0000	1	Hit	
0x1004	0001	0000 0000	0	Miss	Conflict

Cache Replacement

- ที่ผ่านมาได้กล่าวถึง การเลือก block ที่จะนำออก กรณีที่แคชใน set นั้นเต็มแล้ว ซึ่งการนำแคชใดออก จะมีผลต่อประสิทธิ์ภาพ โดยทั่วไปมีวิธี 3 วิธี ได้แก่
  - FIFO จะใช้หลักว่า บล็อกใดเข้ามาก่อน ก็ให้ออกไปก่อน
  - Random จะใช้หลักของการสุ่มในการเลือกบล็อกที่จะนำออกไป
  - ทั้งสองวิธีขั้นต้นเป็นวิธีการที่ไม่มีประสิทธิภาพ เพราะบล็อกที่นำออกไป อาจจะเป็นบล็อกที่จะอ้างอิงครั้งต่อไปก็ได้
  - วิธีการที่ใช้งานมากที่สุด คือ LRU (Least Recently Use) หรือ เลือกบล็อกที่มีการใช้งานน้อยที่สุดออก หรือ บล็อกที่ถูกใช้งานมาแล้วเป็นเวลานานที่สุด

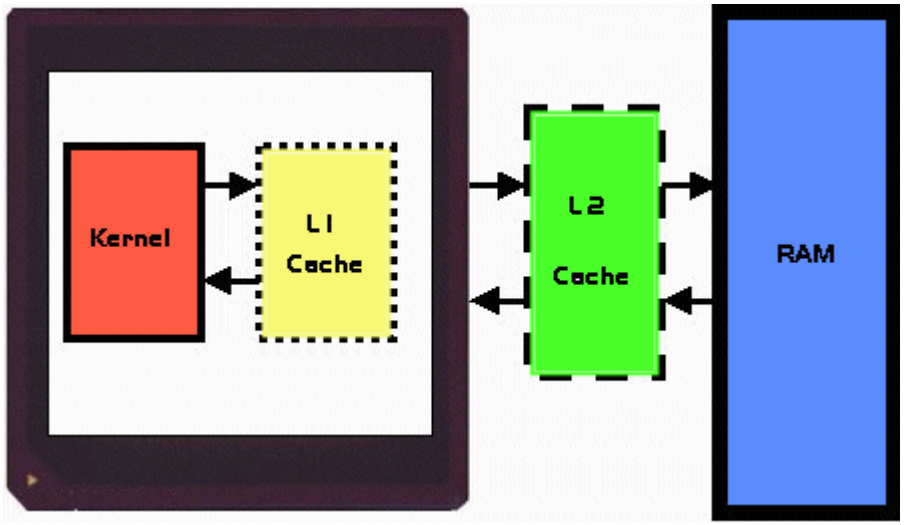
Hardware of 4-way Set-associative

- จากรูป block size ขนาด 4 ไบต์
  ดังนั้น offset = 2 บิต
- จำนวน block = 256
  ดังนั้น index = 8 บิต
- Tag = 32 – (n+m)
  = 32 – (8+2)
  = 22



Performance of Multilevel Caches

- ที่ผ่านมาเรากล่าวถึงแคช level เดียว คือ จาก Processor ไปที่แคช แล้วก็ไปที่หน่วยความจำ การจัดโครงสร้างแบบนี้ ยังมีประสิทธิภาพที่ไม่ดี เนื่องจากความเร็วของแคช กับหน่วยความจำ มีความแตกต่างกันมาก ทำให้ miss penalty มีมาก
- เช่น หากระบบมีหน่วยความจำที่มี access time = 100 ns และโพรเซสเซอร์ความเร็ว 4 GHz ค่า miss penalty = 100 ns / 0.25 ns = 400 clock cycles เลยทีเดียว
- การลด miss penalty วิธีหนึ่ง คือ การเพิ่มแคชให้มีหลาย level



Example

- ในระบบคอมพิวเตอร์หนึ่ง มีค่า CPI = 1.0 (กรณี perfect cache) กำหนด clock rate = 4 GHz กำหนดให้ access time ของหน่วยความจำ = 100 ns กำหนดให้ miss rate = 2 %
- ให้หาว่า ถ้าเพิ่ม cache L2 ที่มี access time 5 ns เข้าไป จะทำให้เร็วขึ้นเท่าไร กำหนดให้ miss rate ของ cache L2 ไปยังหน่วยความจำ = 0.5 %

- ในระบบเดิม miss penalty to memory = 100 ns / 0.25 ns = 400 cc. (clock cycles)
  Total CPI = base CPI + memory stall per instruction
  = 1.0 + 2% * 400 = 9
- ในระบบใหม่ miss penalty to L2 cache = 5 ns / 0.25 ns = 20 cc.
  Total CPI = base CPI + L2 stall per instruction + memory stall per instruction
  = 1.0 + 2% x 20 + 0.5% x 400 = 1+0.4+2.0 = 3.4
- ดังนั้นจะเร็วขึ้น = 9.0/3.4 = 2.6 เท่า

Example

- ในคอมพิวเตอร์เครื่องหนึ่ง มีค่า CPI = 1 สมมติว่ารันโปรแกรมหนึ่ง โปรแกรมนี้มีคำสั่ง load/store ประมาณ 40% ของคำสั่งทั้งหมด
- ในการรันพบว่า 85% ของคำสั่ง load/store พบ (hit) ใน Cache L1
- โดยคำสั่งที่เหลือพบใน Cache L2 อีก 50% และพบใน DRAM อีก 50%
- กำหนดให้ L1 ใช้เวลาทำงานเท่ากับ 1 cc. และ L2 ใช้เวลาทำงานเท่ากับ 10 cc. และ DRAM ใช้เวลา 100 cc.
- CPI ที่แท้จริงของโปรแกรมนี้ คือ เท่าไร
  - สมมติให้โปรแกรมมี 1000 คำสั่ง จะใช้ 1000 cc
  - Access ใน L2 จำนวน 15% ของ 400 คำสั่ง = 400 x 15% x 10
  - Access ใน DRAM จำนวน 50 % ของ 15% = 400 x 15% x 50% x 100
  - 4600 cc = CPI = 4.6

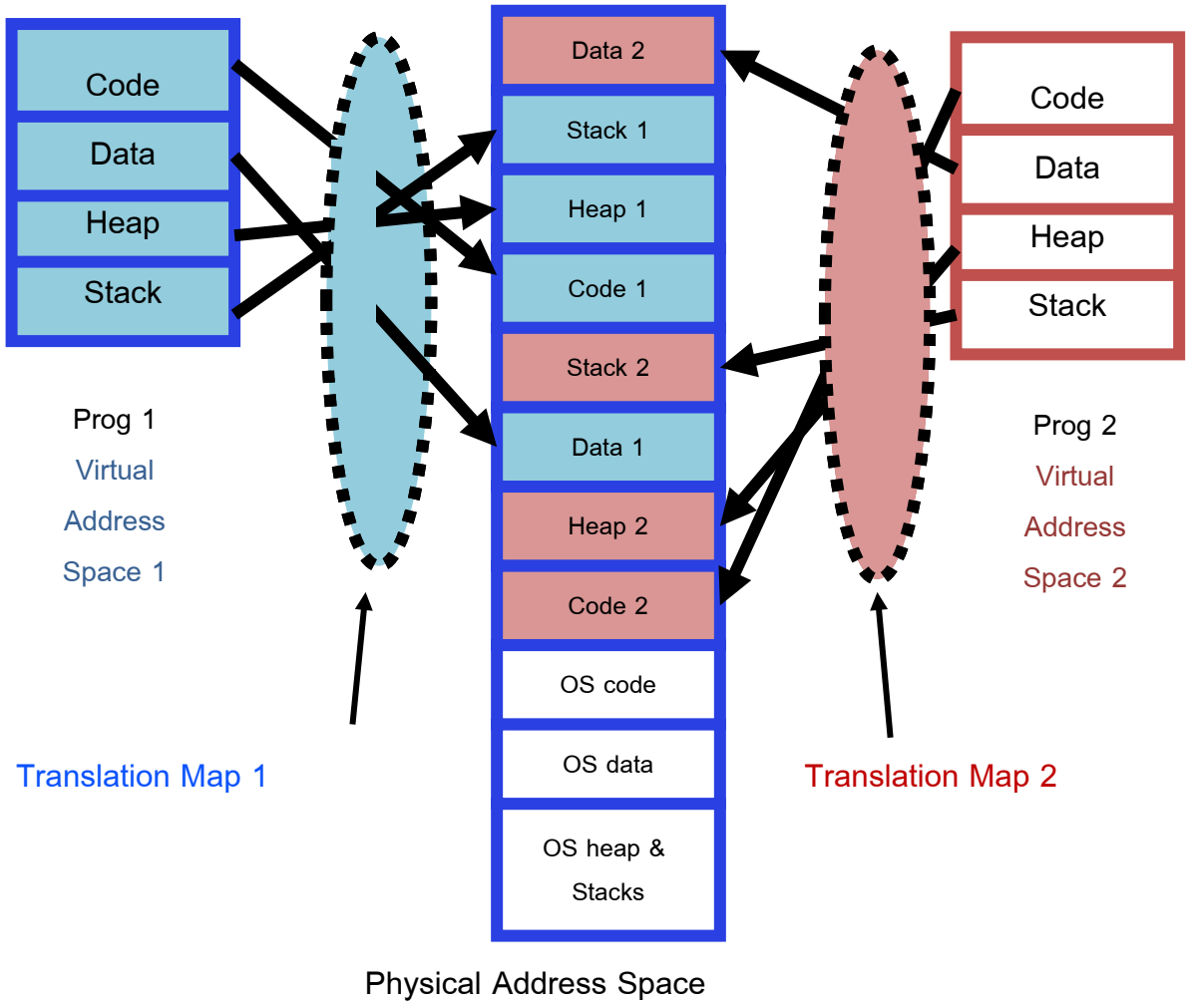
Exercise

- ในระบบคอมพิวเตอร์หนึ่ง มีค่า CPI = 1.0 (กรณี perfect cache) กำหนด clock rate = 2 GHz
- กำหนด Access time ของหน่วยความจำและ Cache ระดับต่างๆ ดังนี้
  - L1 = 4 cycle
  - L2 = 12 cycle (รวม L1 แล้ว)
  - L3 = 36 cycle (รวม L1 และ L2 แล้ว)
  - Dynamic RAM = 50ns + 36 cycle
- กำหนด hit rate ของ L1 = 85%, L2 = 80%, L3 = 80 % ที่เหลือพบใน DRAM
- ในโปรแกรมหนึ่งมีคำสั่ง Load/Store = 60%
- ถามว่าค่า CPI ที่แท้จริงของโปรแกรมนี้ เป็นเท่าไร

Exercise

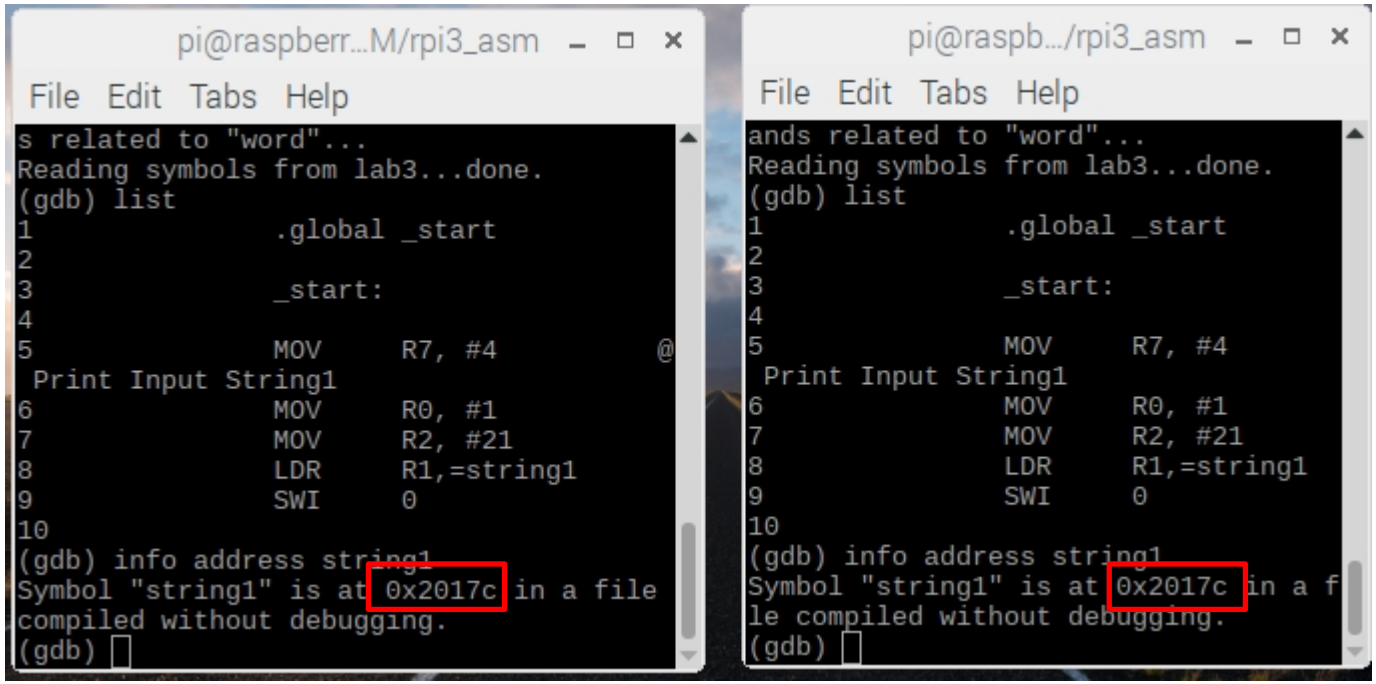
- สมมติให้โปรแกรมมี 1000 คำสั่ง ถ้าไม่มี delay เนื่องจาก cache จะใช้ 1000 clock
- มีคำสั่ง load/store 60% แสดงว่าใน 1000 แบ่งเป็น 400 cc. + (600 cc. + miss penalty)
- ใน 600 คำสั่ง (load/store) จะมีคำสั่งที่พบใน L1 = 600 x 85% = 510 คำสั่ง
  - L1 มี miss penalty = 4 ดังนั้น จะใช้เวลาเพิ่ม = 4 x 600 cc = 2,400 cc
- ในคำสั่งที่เหลือ 90 คำสั่ง จะมีคำสังที่พบใน L2 = 72 คำสั่ง
  - L2 มี miss penalty = (12-4) = 8 ดังนั้นจะใช้เวลาเพิ่ม = 8 x 90 = 720 cc
- ในคำสั่งที่เหลือ 18 คำสั่ง จะมีคำสั่งที่พบใน L3 = 14 คำสั่ง
  - L3 มี miss penalty = (36-12) = 24 ดังนั้นจะใช้เวลาเพิ่ม = 24 x 14 = 336 cc
- คงเหลืออีก 4 คำสั่งที่จะพบใน DRAM โดยใน 1 คำสั่ง จะใช้ 50 ns
  - DRAM จะมี miss penalty = 1 / 2 GHz = 0.5 ns = 100 cc. = 4 x 100 = 400 cc
- รวม (1000+2400+720+336+400)/1000 = 4.856 ดังนั้น CPI = 4.856

Address Translation



Address Translation

- เมื่อโปรแกรมโหลดลงในหน่วยความจำ Loader จะจองหน่วยความจำ โดยแบ่งเป็น 4 ส่วน Code, Data, Heap, Stack
- แต่ละโปรแกรมจะไม่เห็นตำแหน่งที่แท้จริงในหน่วยความจำ จะเห็นในสิ่งที่ OS สร้างสภาพแวดล้อมขึ้นมาให้เห็น (จะเห็นว่า 2 โปรแกรมโหลดพร้อมกัน แต่มี address เดียวกัน)

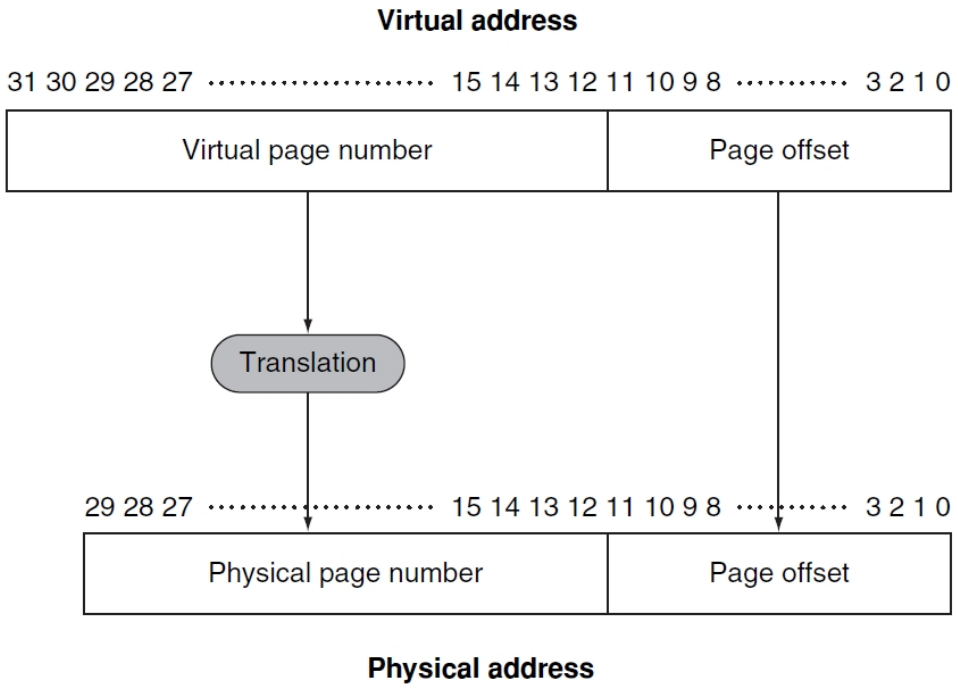


Address Translation

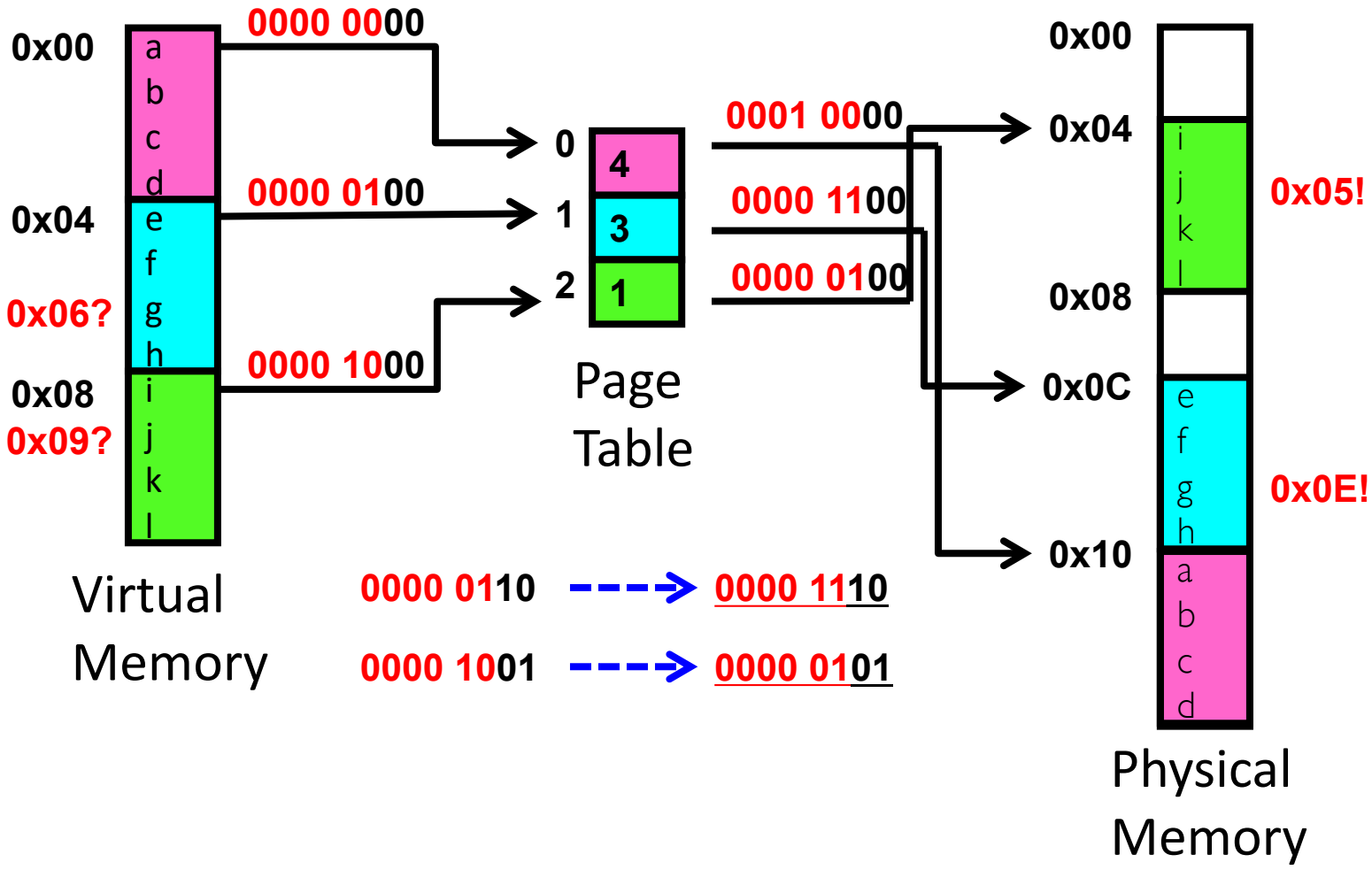
- Address ที่เห็นจากโปรแกรมจะเรียกว่า Virtual Address โดยจะมีลักษณะเป็นพื้นที่ต่อเนื่องกันเป็นผืนเดียวกันทั้งหมด
- Address ที่สามารถเห็นได้จะมีขนาดเท่ากับขนาดของ address เช่น address ขนาด 32 บิต โปรแกรมจะเห็นหน่วยความจำขนาด 4 GB
- แต่ Address ที่เข้าถึงได้จริง จะมีขนาดเท่ากับที่ Loader จองเอาไว้ให้เท่านั้น เมื่อมีการอ้าง Address ที่เกินขอบเขต จึงเกิด segmentation fault
- Address ที่เก็บข้อมูลและโปรแกรมจริงๆ จะเรียกว่า Physical Memory ซึ่งโปรแกรมและข้อมูลอาจจะอยู่ต่อเนื่องกัน หรือไม่ต่อเนื่องกันก็ได้
- เมื่อมีการอ้าง Virtual Address จากโปรแกรม จะต้องแปลงเป็น Physical Address เสียก่อน จึงจะไปอ่านหรือเขียนข้อมูลในหน่วยความจำได้
- กระบวนการแปลงนี้ เรียกว่า Address Translation หรือ Address Mapping

Address Translation

- เพื่อความสะดวก โปรแกรมจะถูกแบ่งเป็นส่วนๆ เรียกว่า page โดยมักจะมีขนาด 4K-16K (ในรูปมีขนาด 4K (2^{12}))
- ในการ map จะ map เฉพาะส่วน Virtual page number กับ Physical page Number เท่านั้น
- ขนาดของ Virtual Address คือ ขนาดของ Address แต่ขนาดของ Physical Address คือ ขนาดของหน่วยความจำจริง ซึ่งมักจะน้อยกว่า เช่น Rpi มีขนาด 1 GB (30 บิต)



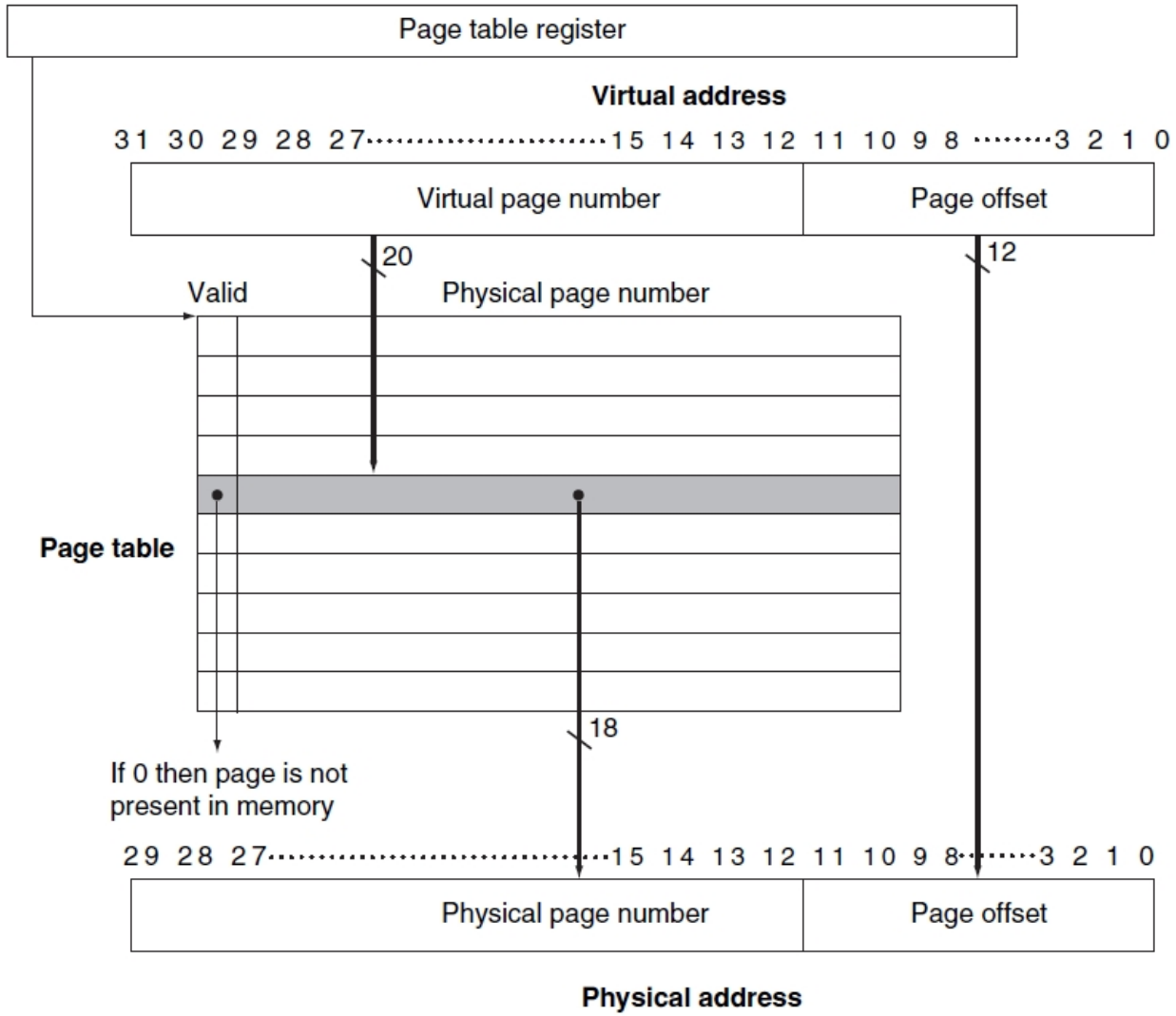
Example of Address Translation



Example of Address Translation

- ตัวอย่างใน slide ก่อนหน้า จะแสดงการทำงานของ Address translation แบบง่ายๆ โดยกำหนดให้ address มีขนาดเพียง 8 บิต
- มี page จำนวน 3 page เริ่มต้นที่ 0x00, 0x04, 0x08
- Page มีขนาด 4 ไบต์ ดังนั้น offset bit = 2 บิต
- ที่เหลืออีก 6 บิต จะเป็น page number ซึ่งจะมีได้สูงสุดไม่เกิน 64 page (2^6)
- Page Table จะทำหน้าที่ map ระหว่าง Virtual page number กับ Physical page number โดยจะต้องมีขนาด 64 ช่อง (จากรูปจะใช้เพียง 3 ช่องจาก 64)
- ตำแหน่ง 0x06 ของโปรแกรม อยู่ใน page ที่ 1 มี offset = 10_2 มี Virtual page number = 000001_2 ซึ่งเมื่อ lookup ในตาราง Page Table จะได้ Physical page number = 000011 ดังนั้นตำแหน่งจริงของ 0x06 คือ 000011 10 = 0x0E
- 0x02 ตรงกับตำแหน่งใดใน Physical memory ?

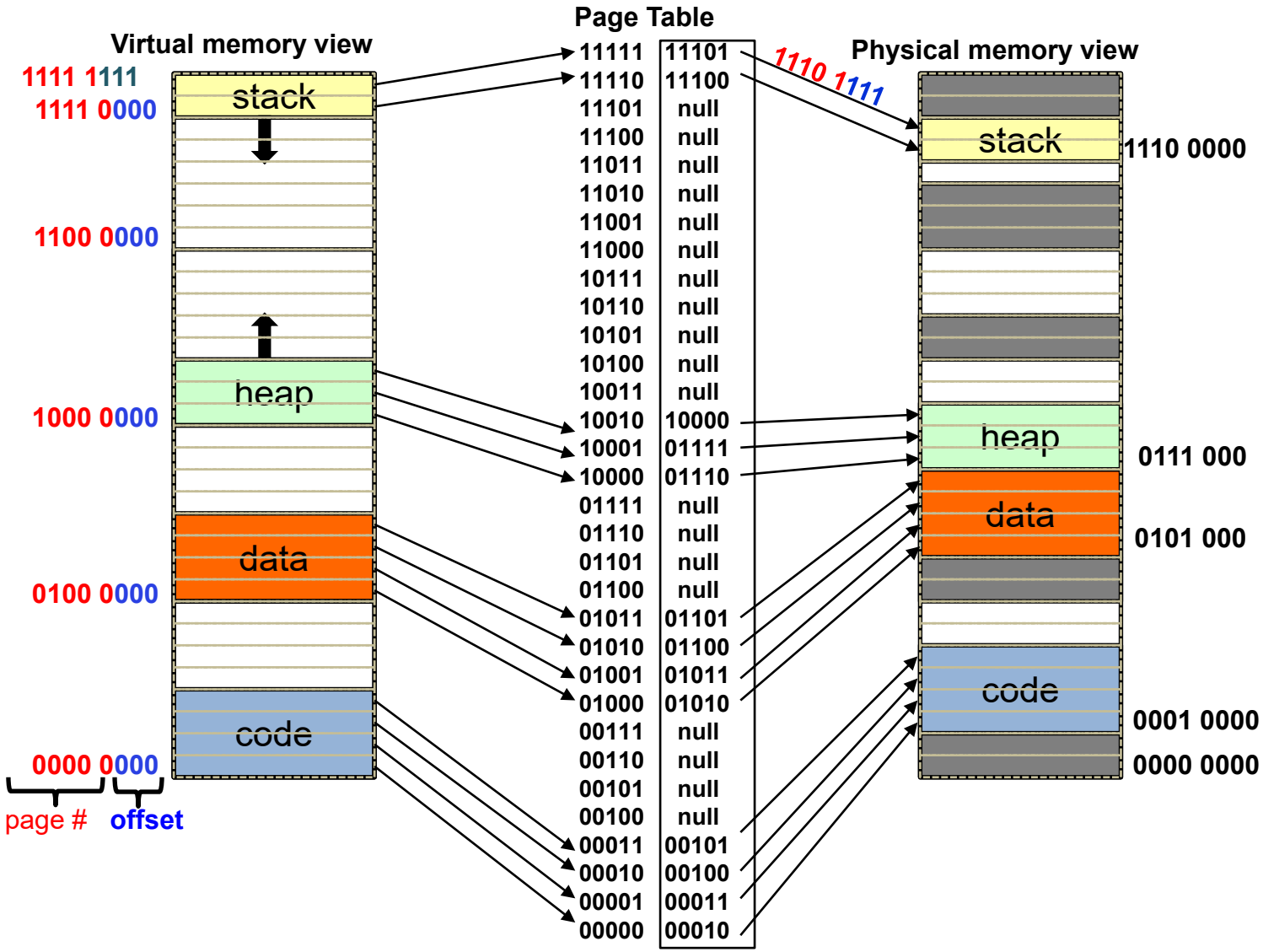
Address Translation



Address Translation

- Page Table จะอยู่ใน main memory โดยมีขนาดเท่ากับ $2^{\text{address bit} - \text{offset bit}}$ เช่น address = 32 บิต โดยมี page size = 4K (12 บิต) ดังนั้น Page Table จะมีขนาด 20 บิต และจำนวนแถวของ Page Table = 2^{20} = 1 ล้าน ถ้าแต่ละแถวใช้พื้นที่ 3 ไบต์ ขนาดของ Page Table = 3 MB
- คำถาม : Page Table จะใช้ร่วมกันระหว่างโปรแกรม หรือ แต่ละโปรแกรมมี Page Table 1 ตาราง
  - คำตอบ : แต่ละโปรแกรมจะมี 1 ตาราง
- บิต Valid ใช้สำหรับบอกว่า Page นี้อยู่ในหน่วยความจำหรือไม่
  - ถ้าเป็น 1 คือ อยู่ในหน่วยความจำ
  - ถ้าเป็น 0 คือ อยู่ใน Hard Disk

Example : 5 bit address translation

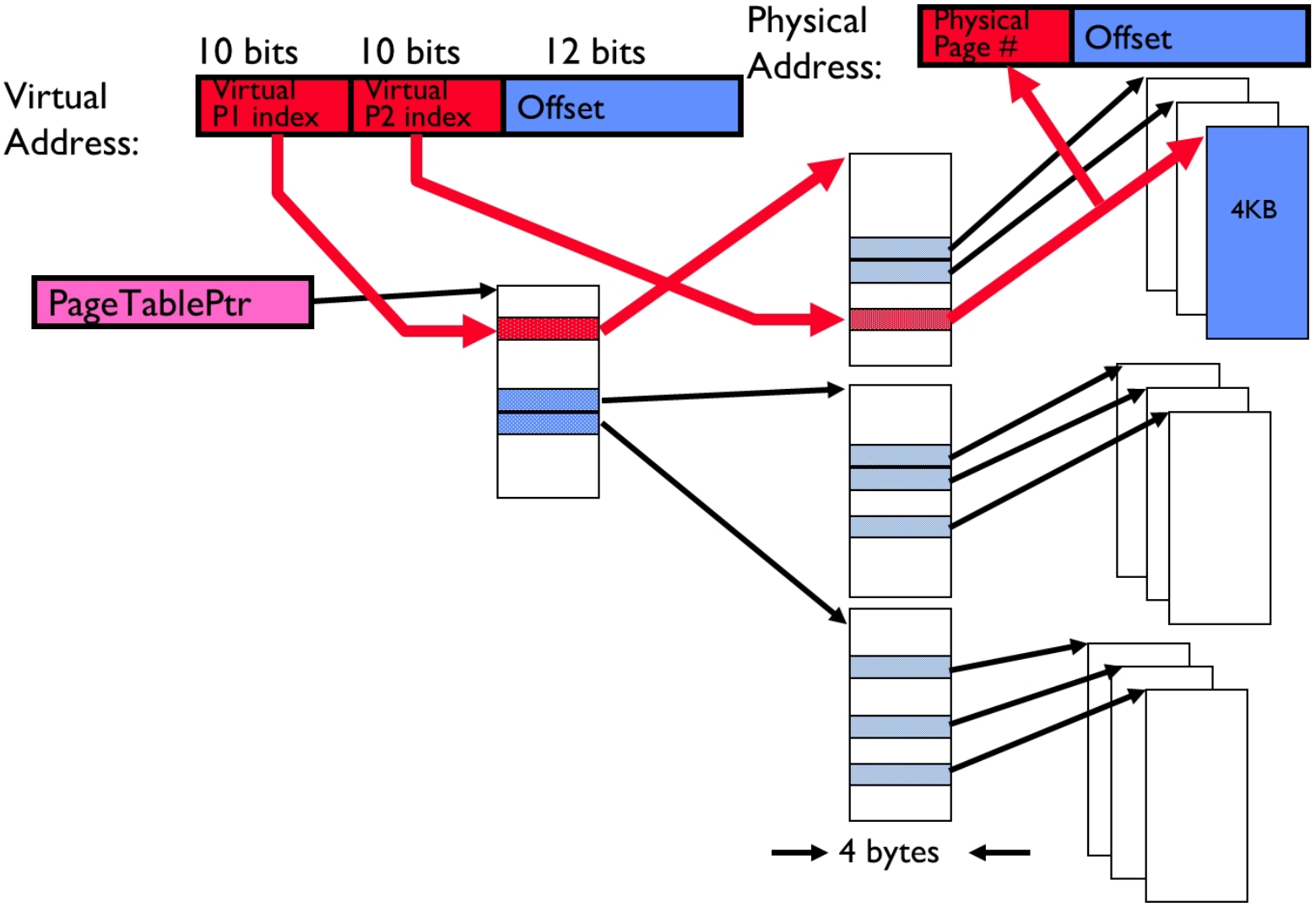


Exercise

- จากสไลด์หน้าที่แล้ว Page มีขนาดกี่ไบต์ และมีจำนวน page สูงสุดกี่ page
- Address 0x82 ของโปรแกรม ตรงกับ Address อะไรใน Physical Memory

- ในระบบคอมพิวเตอร์หนึ่งขนาด 32 บิต มี Page ขนาด 8 KB จะมี offset จำนวนกี่บิต และมีจำนวน page table เท่าไร

Two-Level Page Table



Two-Level Page Table

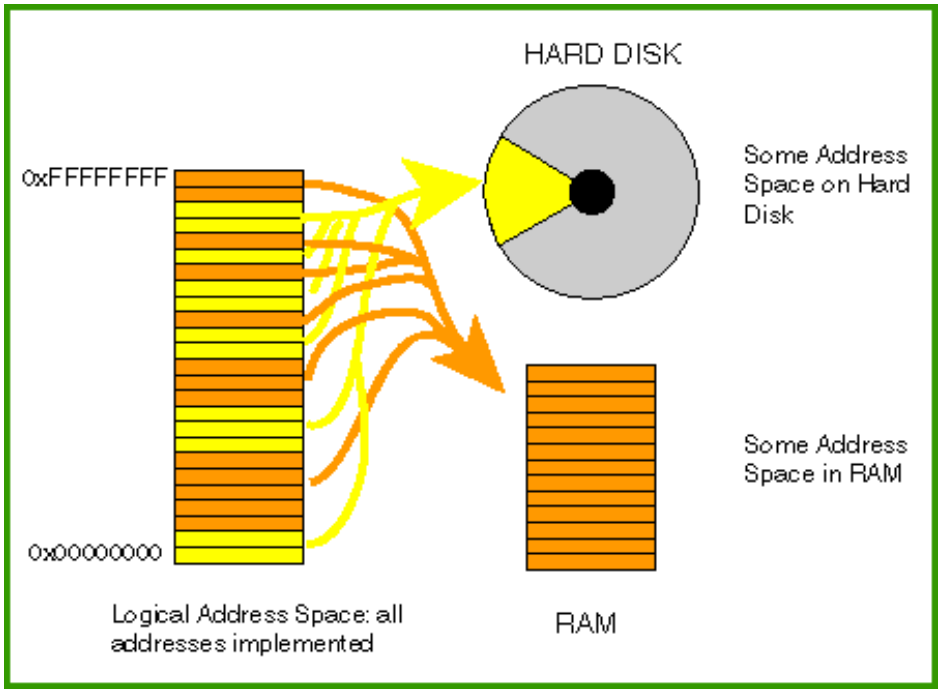
- Page Table ระดับแรกใช้ Index ขนาด 10 บิต ดังนั้น จะมีจำนวน Page = 1024 โดยแต่ละแถว มีขนาด 32 บิต ดังนั้นตารางจะมีขนาด 4 KB ตารางนี้จะมีตารางเดียว ตารางแรก
- Page Table ระดับ 2 ใช้ Index ขนาด 10 บิตเช่นกัน ดังนั้น จะมีจำนวน Page = 1024 โดยแต่ละแถว มีขนาด 32 บิต ดังนั้นตารางจะมีขนาด 4 KB โดยจำนวนตารางจะขึ้นกับการกระจัดกระจายของตำแหน่งโปรแกรมในหน่วยความจำ เช่น สมมติว่าโปรแกรมมีขนาด 1 MB โดย ขนาดของ Page = 4KB (12 บิต) ดังนั้นจะมีจำนวน Page เท่ากับ 256 Page ดังนั้นหากเป็น worst case คือมี Page Table ระดับที่ 2 จำนวน 256 ตาราง ก็จะสิ้นเปลืองเนื้อที่เพียง 256x4KB = 1MB แต่โดยทั่วไปจะไม่กระจายขนาดนั้น
- จะเห็นได้ว่า การใช้ Page Table แบบ 2 ระดับนี้ จะทำให้ประหยัดพื้นที่ของ Page Table ในหน่วยความจำได้ (ในการใช้งานจริง มักจะนิยม 3 ระดับ)

Exercise

- ระบบคอมพิวเตอร์หนึ่ง มี Virtual Address ขนาด 32 บิต แบ่งหน่วยความจำเป็น page มีขนาด 4KB มี Page Table แบบ 2 ระดับ โดย ระดับที่ 2 (PT ที่ชี้หน่วยความจำ) ขนาด 8 KB
  - จงหาขนาด offset, virtual p1 index, virtual p2 index

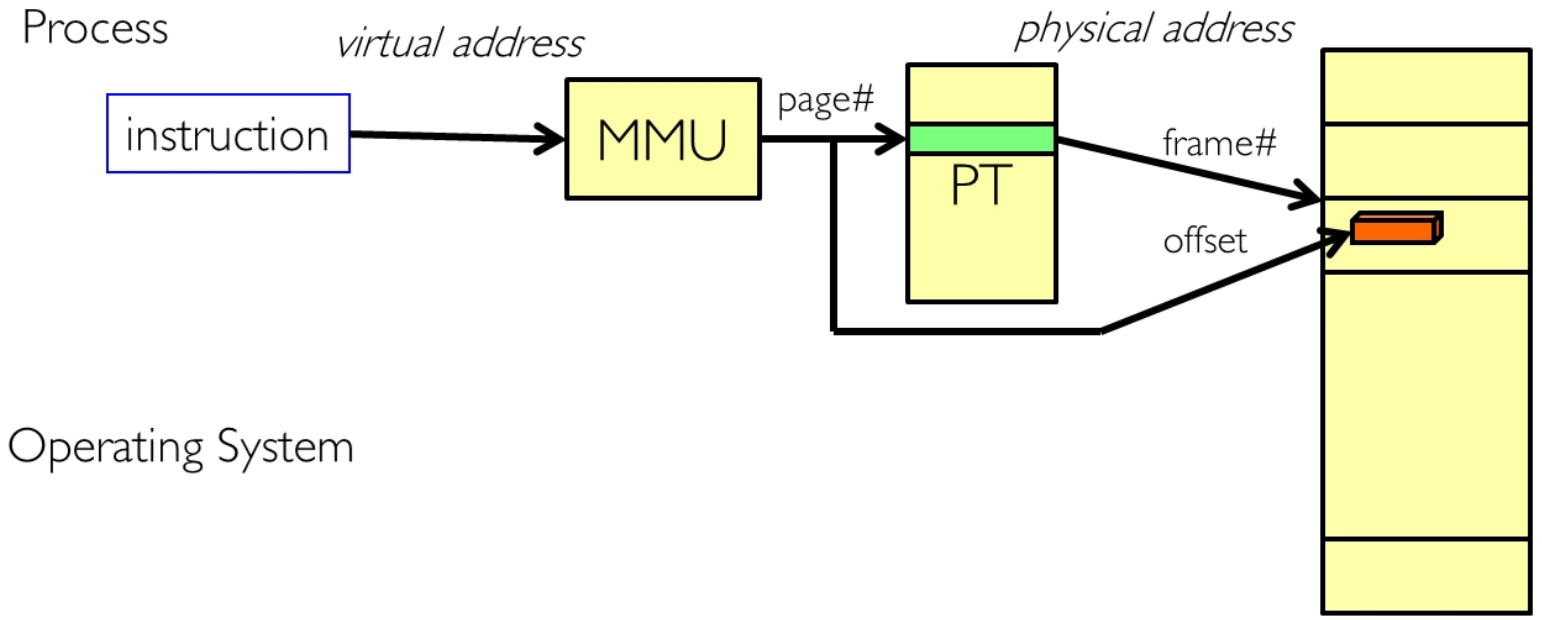
Virtual Memory

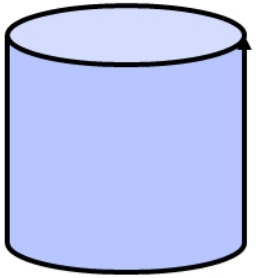
- หากจะมองว่า L2 Cache เป็นแคชของ main memory แล้วจะมองว่า main memory เป็น cache ของ Hard disk ได้หรือไม่?
- วิธีการนี้ เรียกว่า Virtual Memory เพราะเสมือนกับการนำส่วนหนึ่งของ Hard disk มาเป็นส่วนหนึ่งของหน่วยความจำ ทำให้เสมือนกับมีหน่วยความจำเพิ่มขึ้น ดังนั้นจึงสามารถโหลดโปรแกรมได้มากขึ้น (มีใครเคยโหลดโปรแกรมจน memory เต็มบ้าง?)



- แล้วมันจะไม่ช้าเหรอ?
  - ก็ใช้หลักการเดียวกับ cache (locality) คือ processor จะต้องเข้าถึง hard disk เมื่อเกิด miss จาก main memory เท่านั้น ดังนั้นหากโปรแกรมไม่มากเกินไป การ miss ก็จะเกิดขึ้นไม่มาก

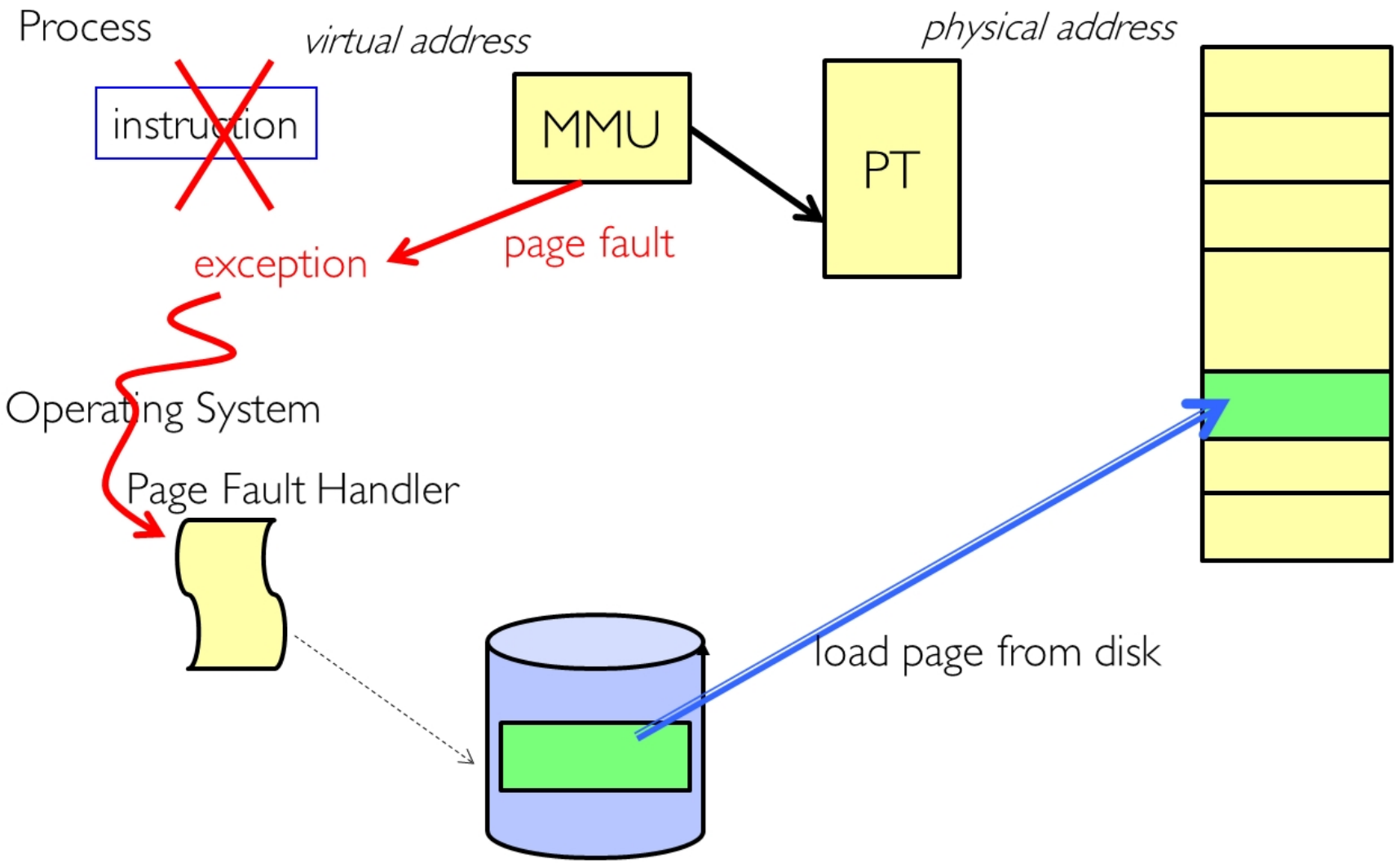
Page fault





กรณีที่มีการอ้างถึงหน่วยความจำ
และหน่วยความจำนั้นไม่อยู่ใน
cache และไม่อยู่ใน DRAM

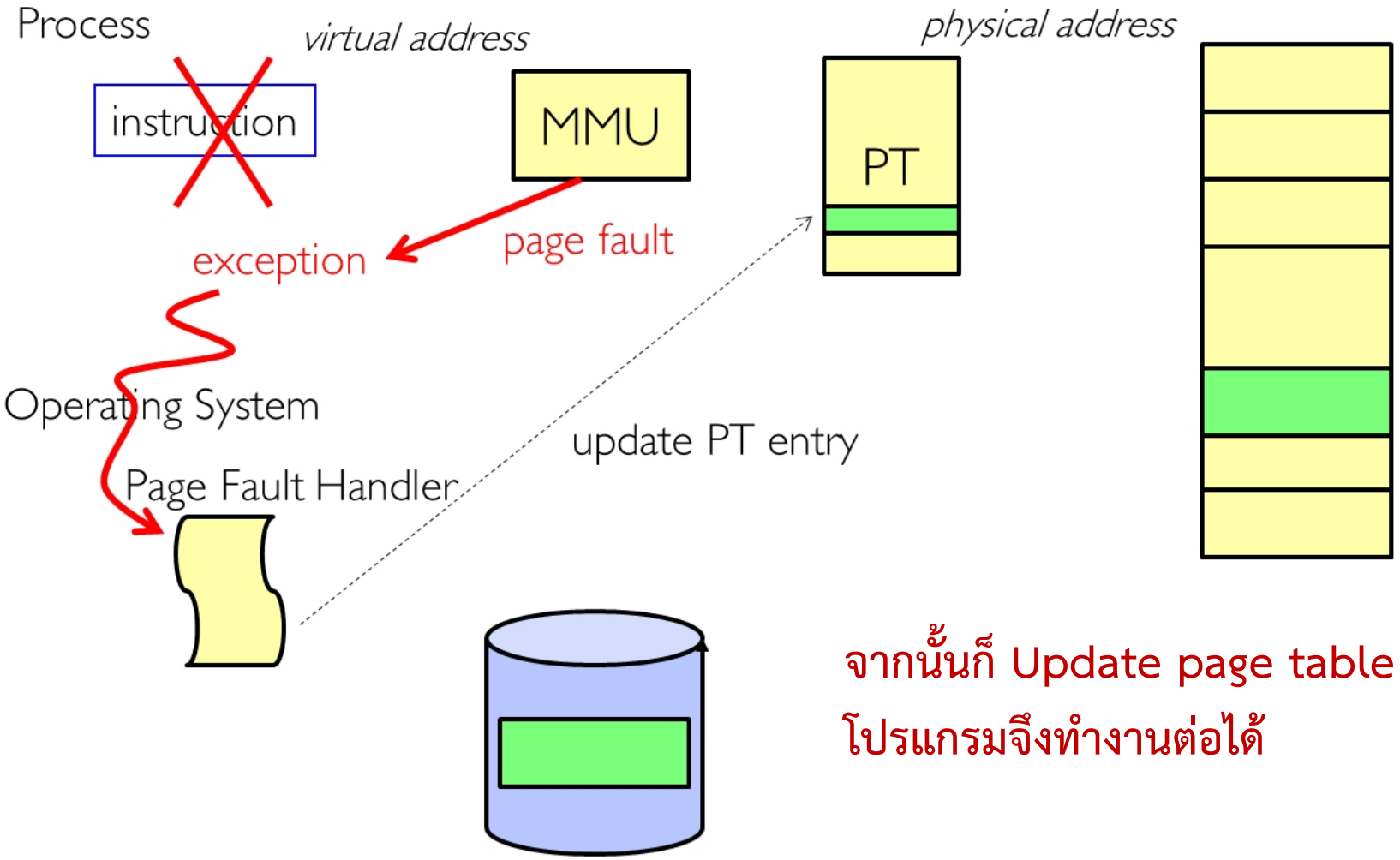
Page fault


Process
virtual address
physical address
instruction
MMU
PT
page fault
exception
Operating System
Page Fault Handler
load page from disk

Page fault

- แต่เมื่อไปอ่านใน Page Table แล้วพบว่า page ที่จะอ้างถึง อยู่ใน Hard Disk ก็ทำงานต่อไม่ได้ เพราะ Processor จะอ่านข้อมูลได้จาก RAM เท่านั้น เหตุการณ์นี้เรียกว่า page fault
- ซึ่งต้องขอความช่วยเหลือจาก OS โดย Processor จะเรียก exception ของ OS ขึ้นมาทำงาน (exception จะคล้ายกับที่เรียกใช้คำสั่ง SWI คือจะไปเรียกการทำงานของระบบขึ้นมา แต่ exception เกิดขึ้นจากฮาร์ดแวร์ เช่น divide by zero หรือ segmentation fault ที่เราเคยเจอ)
- OS จะเข้าไปเรียกโปรแกรมย่อยที่เรียกว่า exception handler ซึ่งมีหลายโปรแกรม สำหรับ page fault ก็จะเรียกว่า page fault handler ซึ่งจะทำหน้าที่โหลด page ที่ต้องการจาก hard disk มาไว้ในหน่วยความจำ

Page fault



จากนั้นก็ Update page table

โปรแกรมจึงทำงานต่อได้

Page fault

- Virtual Memory ทำงานด้วย Cache แบบใด
  - Direct-mapped
  - set-associative
  - Fully associative
- Page Replacement ใช้แบบใด
  - FIFO
  - Random
  - LRU
- Write through หรือ Write back

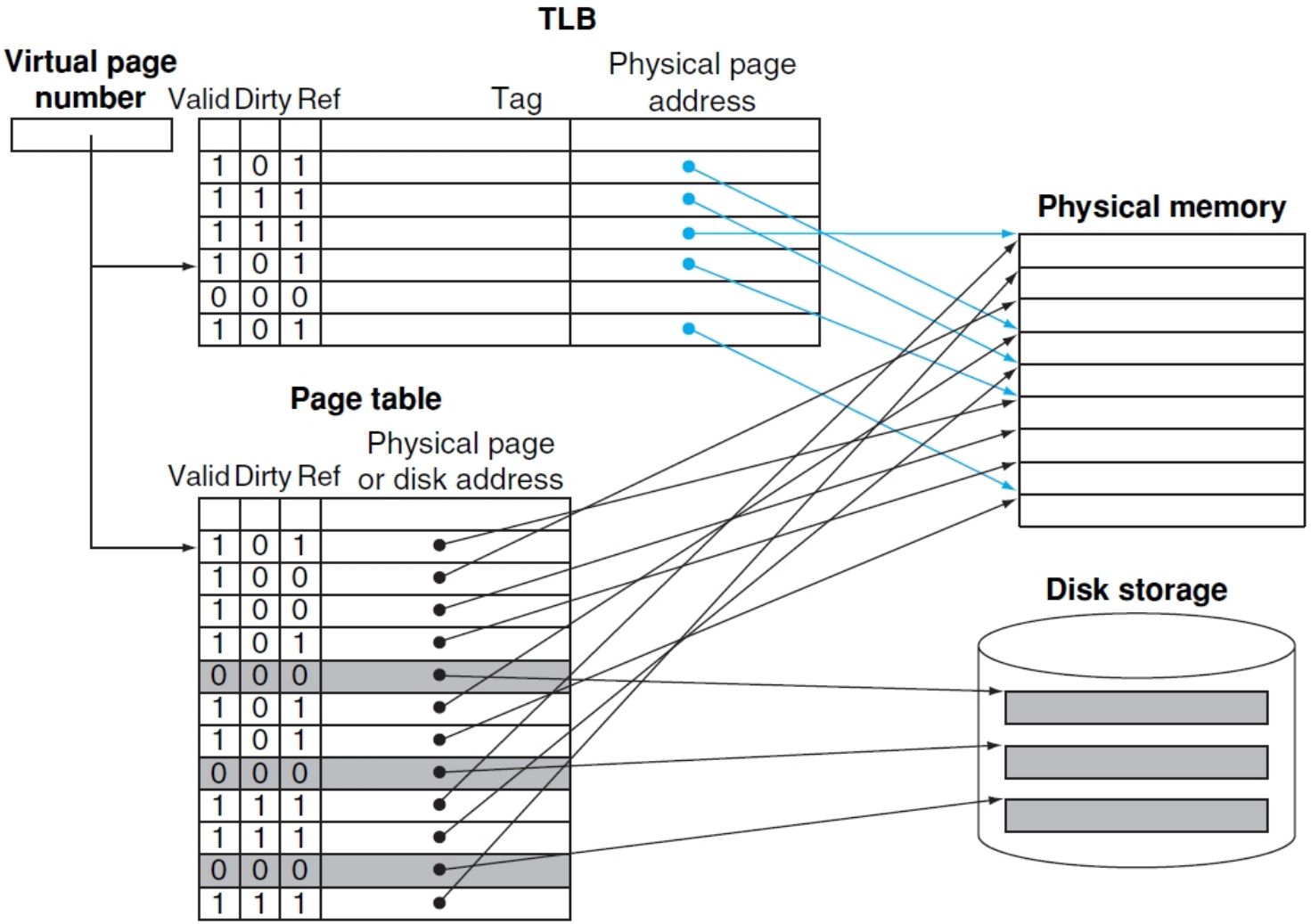
Making Address Translation Fast

- การนำแนวคิดเรื่อง Address Translation (Paging) มาใช้มีข้อดีหลายประการ
  - Virtual Address ต่อเนื่องกัน ทำให้ไม่มีปัญหาเรื่องการ Branch (ที่ไกลมากไป)
  - ระบบปฏิบัติการสร้าง Protection ระหว่างโปรแกรมได้ง่าย (ไม่สามารถกระโดดเข้าสู่พื้นที่ของโปรแกรมอื่นได้)
  - Loader ไม่ต้อง relocate โปรแกรมไปตาม Physical Address
  - สามารถใช้ Virtual Memory ได้
- ข้อเสียของ Address Translation ก็มี
  - ต้องเปลืองเนื้อที่ในหน่วยความจำ เพื่อใช้เป็น Page Table
  - ทำงานช้าลง เพราะแทนที่จะอ่าน/เขียนหน่วยความจำเลย กลับต้องไปอ่าน Page Table เสียก่อน จึงจะทราบ Physical address และหากมี Page Table แบบ Multi-level ก็ยิ่งช้าลงไปอีก เช่น Page Table 3 ชั้น ก็ต้องเสียเวลาอ่านหน่วยความจำเพิ่ม 3 ครั้ง

Making Address Translation Fast

- จึงได้มีการสร้าง cache ขึ้นมาอีกชนิดหนึ่ง ทำหน้าที่เก็บข้อมูลการ lookup ของ Page Table มีชื่อว่า Translation lookaside buffer (TLB)
  - มีตำแหน่งอยู่ในตัว Processor
  - มีขนาดเล็ก 16-512 entries
  - ส่วนมาก มักเป็นแคชชนิด Fully associative แต่ก็มีแบบที่เป็น Set-associative โดยบล็อกจะมีขนาด 1-2 page table entries (4-8 ไบต์)
  - Hit time ประมาณ 0.5-1 clock cycles
  - Miss rate ประมาณ 0.01%-1% เนื่องจากใน 1 Page Table จะชี้ไปที่หน่วยความจำขนาดประมาณ 4K ซึ่งโดยทั่วไป โปรแกรมขนาด 4K หรือข้อมูลขนาด 4K จะมี Locality ระดับหนึ่ง ทำให้ miss rate มีไม่มาก
  - เมื่อ TLB miss จะ lookup Page Table และนำข้อมูลมาเก็บใน TLB กรณีที่ TLB เต็ม การนำบาง entry ออก จะใช้ LRU

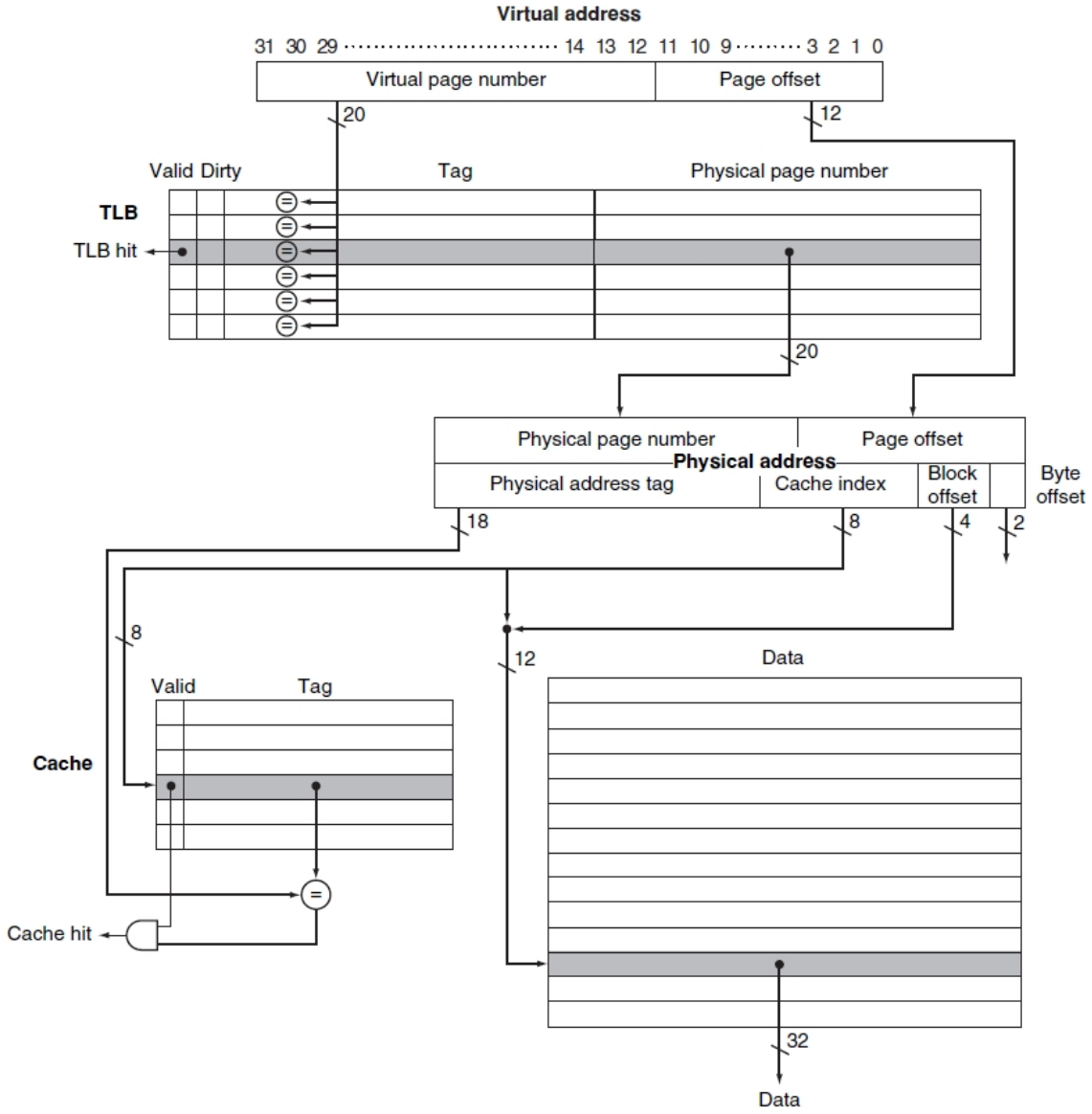
Translation lookaside buffer



Translation lookaside buffer

- รายละเอียดใน TLB
  - Tag คือ Virtual Page Number
  - Valid คือ มีการใช้งาน
  - Dirty คือ มีการ write ใน page นั้น
  - Reference คือ มีการอ้างถึงข้อมูลใน page นั้น (คือการ hit)
- เมื่อมี TLB miss เกิดขึ้น และมีการนำ TLB entry ออกไป จะ update status bit ทั้งหลายไปที่ page table
- กรณีที่ TLB miss จะพยายามโหลด mapping เข้ามาใหม่ แต่เมื่อพบว่า page นั้นไม่อยู่ในหน่วยความจำ ก็จะเป็น page fault แทน

Example


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
12
Data
Valid
Tag
Cache
Cache hit
32
Data

Example

- ในระบบคอมพิวเตอร์หนึ่ง มีการใช้ TLB แบบ Fully associative ขนาด 4 entry โดยใช้ replacement policy แบบ LRU
  โดยกำหนดให้หน่วยความจำมี Page ขนาด 4 KB กำหนดให้ Address ที่ Access มีลำดับดังนี้
  - 12948, 49419, 46814, 13975, 40004, 12707 และ 52236
- โดยมี TLB ดังนี้

TLB

Valid	Tag	Physical Page Number
1	11	12
1	7	4
1	3	6
0	4	9

Example

- และมี Page Table ดังนี้

Valid	Physical Page or in Disk
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12
1	7

Example

- จากลำดับการอ้าง Address และค่าเริ่มต้นใน TLB และ Page Table ให้แสดงสถานะสุดท้ายหลังจากสิ้นสุดการอ้าง address ตามที่กำหนด
- Page ขนาด 4 KB = 2^12 = offset 12 บิต
- Page Table มี 12 Page ดังนั้นขนาดไม่เกิน 4 บิต (2^16) เมื่อพิจารณาประกอบกับ Address ขนาดไม่เกิน 64KB (65536) ดังนั้นจึงอนุมานให้ index ที่ทำหน้าที่ชี้ Page มีขนาด 4 บิต
- 12948 = 0x3294 ดังนั้น Index = 3
  - เมื่อพิจารณาจากตาราง TLB ในช่อง Tag (เนื่องจาก Address ที่โปรแกรมอ้างเป็น Virtual Address ดังนั้น จึงใช้ข้อมูลในช่อง Tag) พบว่ามี 3 อยู่ แสดงว่าไม่ต้องใช้ Page Table โดยข้อมูลที่เก็บอยู่ที่ Page 6 การอ้างอิงนี้ไม่ทำให้ข้อมูลเปลี่ยนแปลง
- 49419 = 0xC10B ดังนั้น Index = C (12)
  - เมื่อพิจารณาจากตาราง TLB ไม่พบ ดังนั้นต้องอ่านจาก Page Table ในตำแหน่งที่ 12 พบว่า Valid คือ อยู่ในหน่วยความจำ Page 7
  - ดังนั้นจึงต้องกลับมา Update TLB โดยใน Entry ที่ 4 ว่าง โดยเปลี่ยนเป็น Valid Tag=12 , Physical Page = 7

Example

- 46814 =0xB6DE ดังนั้น Index = B (11)
  - เมื่อพิจารณาจากตาราง TLB พบว่ามี 11 อยู่ แสดงว่าไม่ต้องใช้ Page Table โดยข้อมูลที่เก็บอยู่ที่ Page 12 การอ้างอิงนี้ไม่ทำให้ข้อมูลเปลี่ยนแปลง
- 13975 = 0x3697 : Index = 3 (เหมือน address แรก)
- 40004 = 0x9C44 ดังนั้น Index = 9
  - เมื่อพิจารณาจากตาราง TLB ไม่พบ ดังนั้นต้องอ่านจาก Page Table ในตำแหน่งที่ 9 พบว่าอยู่ใน Disk
  - จึงต้องเรียก exception handler เพื่ออ่านข้อมูลใน Disk มาไว้ที่หน่วยความจำ
  - แต่เนื่องจากโจทย์ไม่ได้กำหนดข้อมูล ดังนั้นจึงสมมติว่า หลังจากอ่านข้อมูล Page นี้เข้ามาในหน่วยความจำแล้ว อยู่ใน block ที่ 8
  - ดังนั้นจะต้อง update บรรทัดที่ 9 ของ Page Table เป็น Valid = 1 และ Physical Page = 8
  - และต้องมา Update TLB แต่ TLB เต็มแล้ว ดังนั้นต้องมีการ replace ซึ่งเมื่อพิจารณาจากการอ้างอิง พบว่าการอ้างอิงที่ผ่านมาคือ 3,4,13 ดังนั้นในบรรทัดที่ 2 (tag = 7) ยังไม่มีการอ้างอิง จึงลบบรรทัดนี้ออก
  - และแทนที่ด้วย Tag = 9, Physical Page = 8

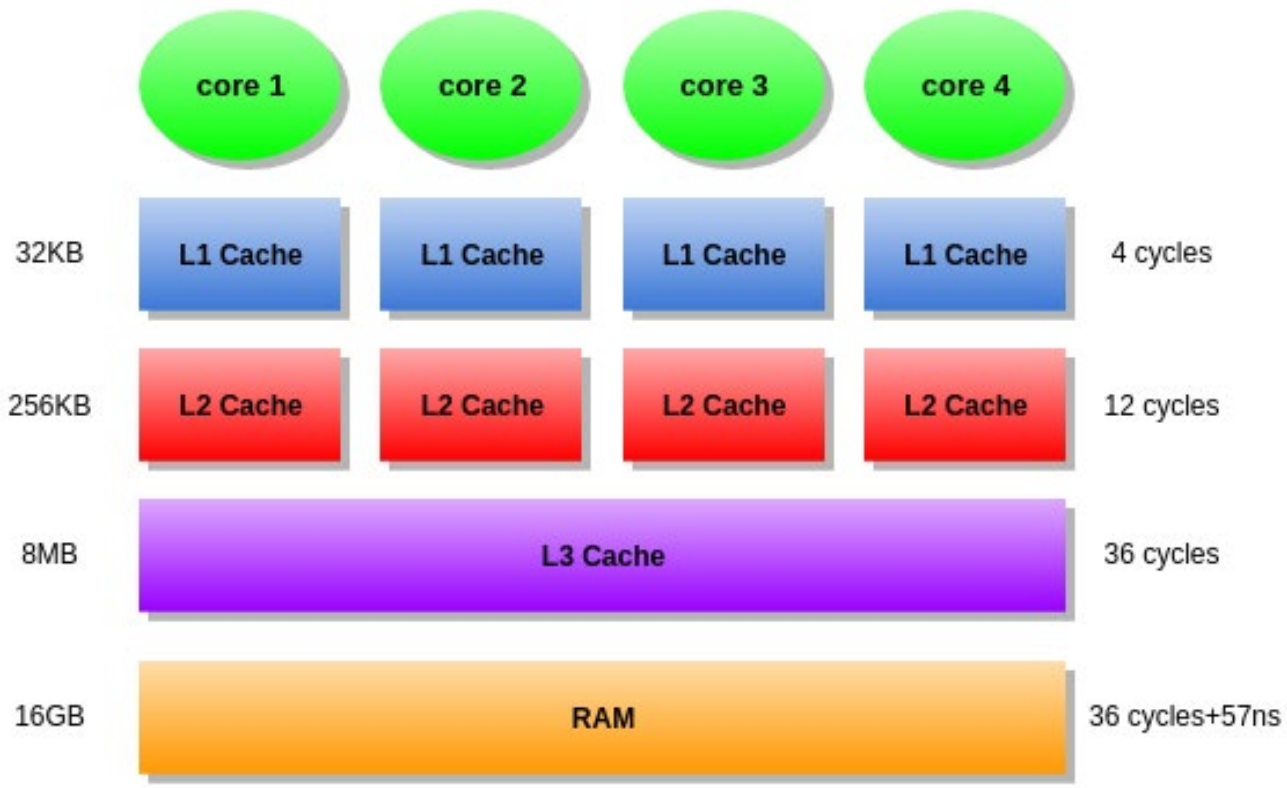
Cache

- ตารางสรุป สถานการณ์ เฉพาะการ Hit/Miss ของ Cache แต่ละประเภท
  - TLB miss หมายถึง ไม่พบข้อมูลใน TLB
  - Page Table miss หมายถึง Page invalid
  - Cache miss หมายถึง ไม่พบใน cache

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

Cache in Multiprocessor

- CPU ส่วนใหญ่ที่ใช้ในคอมพิวเตอร์ปัจจุบันจะเป็น Multi-Core Processor ซึ่งโดยทั่วไปจะมีหน่วยความจำ Cache L1, L2 แยกไปแต่ละ Core แต่สำหรับ L3 อาจจะรวมหรือแยกขึ้นกับการออกแบบ แต่ที่แน่ๆ คือ จะใช้หน่วยความจำ (RAM) ร่วมกัน



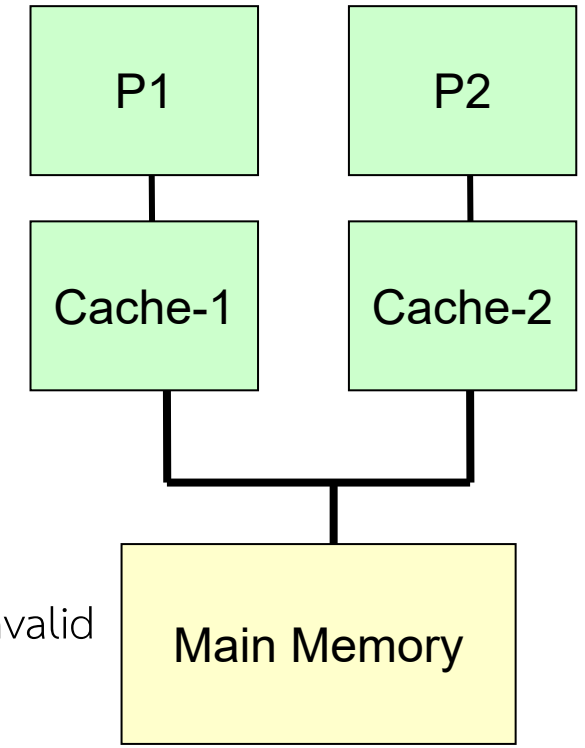
Cache in Multiprocessor

- นอกจากนั้น การเขียนโปรแกรมปัจจุบันที่เป็น Multi-threading ทำให้ใน 1 โปรแกรมสามารถแยกออกเป็นหลาย thread และแยกกันไป run ในแต่ละ Core ซึ่งในโปรแกรมเดียวกัน ก็อาจจะมีการใช้หน่วยความจำร่วมกันได้
- ซึ่งทำให้เกิดเหตุการณ์ที่เรียกว่า Cache Coherence (ความสอดคล้องของแคช) จากรูป CPU A และ B อ่านข้อมูล X ตามลำดับ ต่อมา CPU A มีการเขียนข้อมูล 1 (สมมติเป็น write through) แต่ข้อมูลในแคชของ CPU B ยังเป็น 0 อยู่ ทำให้เกิดความไม่สอดคล้องกัน

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

Cache Coherence Protocol

- เพื่อแก้ปัญหาข้างต้น จึงได้ออกแบบวิธีการแก้ปัญหาเรียกว่า Cache Coherence Protocol โดยตัวหนึ่งที่นิยมใช้ คือ Snooping-Based Protocols ลองมาดูการทำงาน
- โพรเซสเซอร์ P1 read x ซึ่งไม่พบใน Cache-1, จึงร้องขอไปที่ memory ซึ่งได้รับ X มาเก็บใน cache โดยจะอยู่ใน shared state
- โพรเซสเซอร์ P2 read x ซึ่งไม่พบใน Cache-2, จึงร้องขอไปที่ memory, P1 snoop bus แต่เนื่องจากเป็นการอ่าน จึงไม่ต้องทำอะไร memory ส่ง x มาเก็บใน cache โดยจะอยู่ใน shared state เช่นกัน
- P1 ต้องการ write X, P2 ซึ่ง snoop bus อยู่พบว่าเป็นการเขียน จึงเปลี่ยน status ของ x ใน cache-2 ให้เป็น invalid จากนั้น x จึงเปลี่ยนทั้งใน cache-1 และ memory
- เมื่อ P2 ต้องการอ่าน x จะต้องมาอ่านใน memory ใหม่ เพราะ invalid



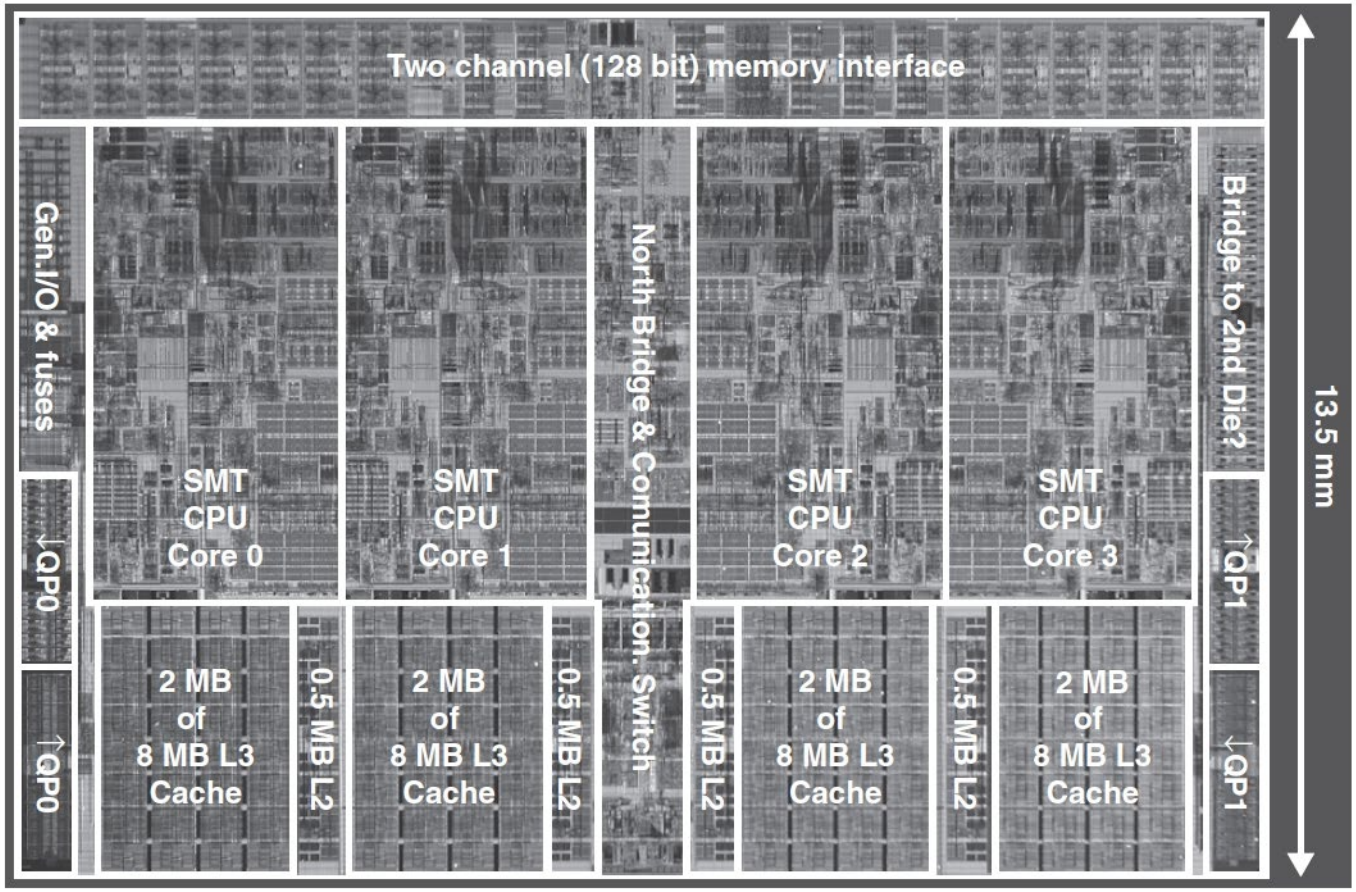
Cache Coherence Protocol

- จากตาราง จะเห็นได้ว่าในบรรทัดที่ 3 เมื่อ CPU A ต้องการ write ไปที่ X จะทำให้ X ใน CPU B เป็น Invalid คือ ลบหายไปจาก Cache
- เมื่อ B ต้องการอ่าน X จึงต้องไปอ่านจากหน่วยความจำ ซึ่งขณะนั้นมีค่าเป็น 1 แล้ว X ใน B จึงมีค่าเป็น 1 และไม่เกิด Inconsistency

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Real Stuff

- ภาพ Die ของ Processor Intel Nehalem ขนาด 13.5 x 19.6 มม. มี L1 Cache 32KB+32KB (I+D) L2 Cache = 512 KB, shared L3 ขนาด 8 MB



Real Stuff

- เป็นการเปรียบเทียบระหว่าง Processor 2 ตัว โดยมี Virtual Address 48 บิตเท่ากัน (256 TB) แต่ Intel อ้าง Physical ได้น้อยกว่า (16 TB) มี page size ขนาดเท่ากัน
- AMD มี L1 แบบ Fully associative, แต่ Intel เป็น 4-way set associative, L2 เป็นแบบ 4-way set associative เหมือนกัน

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
Virtual address	48 bits	48 bits
Physical address	44 bits	48 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement The L2 TLB is four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB has 512 entries TLB misses handled in hardware	1 L1 TLB for instructions and 1 L1 TLB for data per core Both L1 TLBs fully associative, LRU replacement 1 L2 TLB for instructions and 1 L2 TLB for data per core Both L2 TLBs are four-way set associative, round-robin Both L1 TLBs have 48 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

Real Stuff

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles

Real Stuff

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

For your attention