VE370 Final Review

Concepts

- L8 Data hazards
- L9 Control hazards
- L10 Exception
- L11 Cache
- L12 Virtual Memory
- L13 I/O & Interfaces

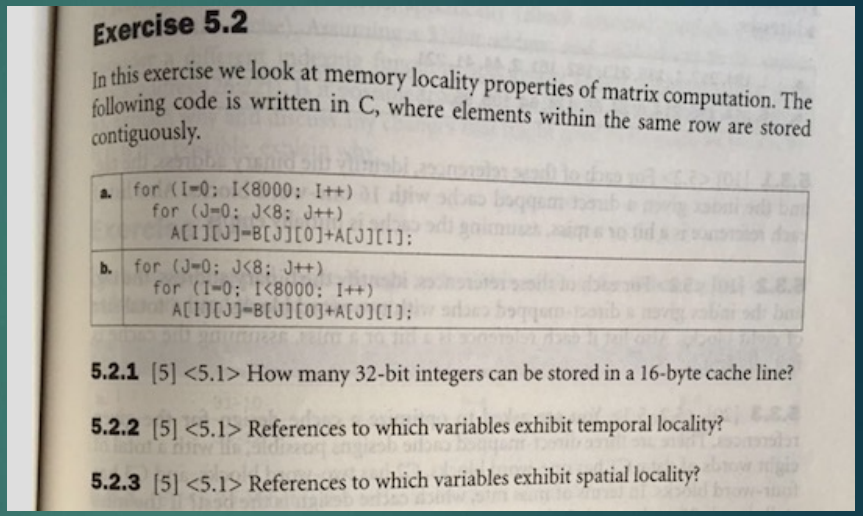
Hazard

- Review your project 2
- Familiar with all signals

L11 Cache

- Temporal and Spatial Locality, L11 p9
  - Temporal locality
    - Items that are accessed recently are likely to be accessed again soon
    - e.g., instructions in a loop
  - Spatial locality
    - Items near those that are accessed recently are likely to be accessed soon
    - E.g., sequential instruction access, array data
- HW7 5.2.2~5.2.3

5.2.1-3



Exercise 5.2

In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously.

a.
```
for (I=0; I<8000; I++)
  for (J=0; J<8; J++)
    A[I][J]=B[J][0]+A[J][I];
```

b.
```
for (J=0; J<8; J++)
  for (I=0; I<8000; I++)
    A[I][J]=B[J][0]+A[J][I];
```

5.2.1 [5] <5.1> How many 32-bit integers can be stored in a 16-byte cache line?

5.2.2 [5] <5.1> References to which variables exhibit temporal locality?

5.2.3 [5] <5.1> References to which variables exhibit spatial locality?

4 integers

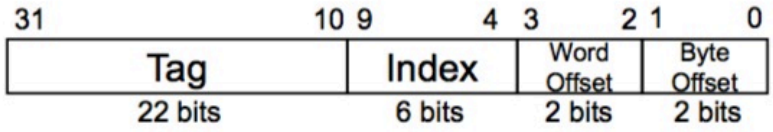
temporal: (a) I, J (b)I, J, B[J][0]

Spatial: (a) A[I][J] (b) A[J][I]

L11 Cache

- Block address, Word address, Byte address
  - e.g 4-word block.
  - 1111 0101 0101 1101,
  - byte:01, word:11, block: 1111 0101 0101
- Divided by 4 = shift left two bits
- modulo 4 = take the last two bits

- 64 blocks, 4 words/block
  - What cache block number does byte address 1200 map to?
  - Word number = 1200/4 = 300
  - Block (address) number = 300/4 = 75
- Block index in cache = 75 modulo 64 = 11



L11 Cache

- Direct Mapped Cache
  - Two-word block vs. Two-way set associative
  - Hw7 5.3(do it by yourself!)
- Set Associative Cache
  - Direct mapped, n-way set associative, fully associative
  - LRU(maybe MRU in the exam?)
  - Hw8 5.8(do it by yourself!)

L11 Cache

- Block size Consideration
  - Pros and Cons
  - Hw7 5.3.4

Block Size Considerations

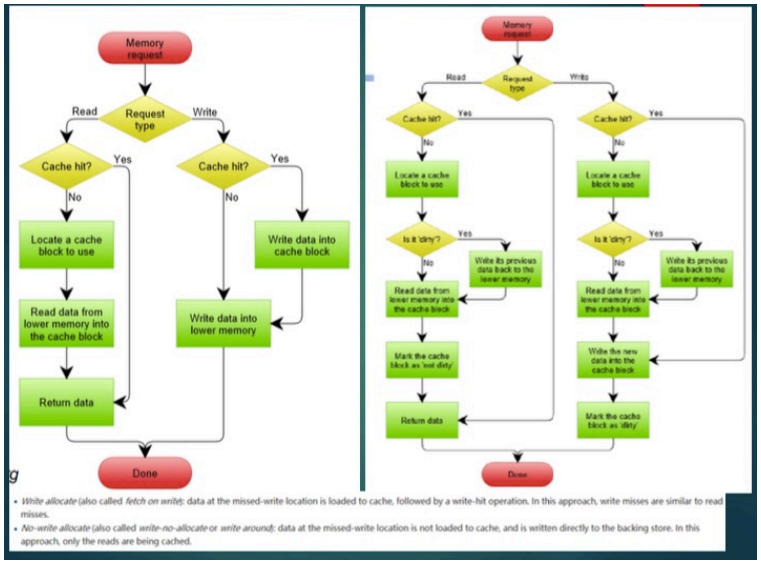
- ***Larger blocks should reduce miss rate***
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
- Larger miss penalty
  - Primarily result of longer time to fetch block
    - Latency to first word
    - Transfer time for the rest of the block
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

5.3.4

a.	Using equation on page 351, n = 14 bits, m = 0 (1 word per block) $2^{14} \times (2^0 \times 32 + (32 - 14 - 0 - 2) + 1) = 802$ Kbits Calculating for 16 word blocks, m = 4, if n = 10 then the cache is 541 Kbits, and if n = 11 then the cache is 1 Mbit. Thus the cache has 128 KB of data. The larger cache may have a longer access time, leading to lower performance.

L11 Cache

- Write through,Write back, Write Allocate
  - Write through: also update in the memory
  - Write back: dirty bits, update only when to be replaced
  - Flow chart
  - Hw8 5.5.2



L11 Cache

- Multilevel Cache
  - Hw 5.7.1, 5.7.3, 5.7.4~6, 5.8.4
- Calculation of AMAT, CPI

Cache Performance Example

- Given
  - I-cache miss rate = 2% (2 misses per 100 instructions)
  - D-cache miss rate = 4% (4 misses per 100 memory access instructions)
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 100% × 2% × 100 = 2
  - D-cache: 4% × 36% × 100 = 1.44
- Total CPI = base CPI + Miss (stall) cycles per instruction
  - Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

Average Memory Access Time

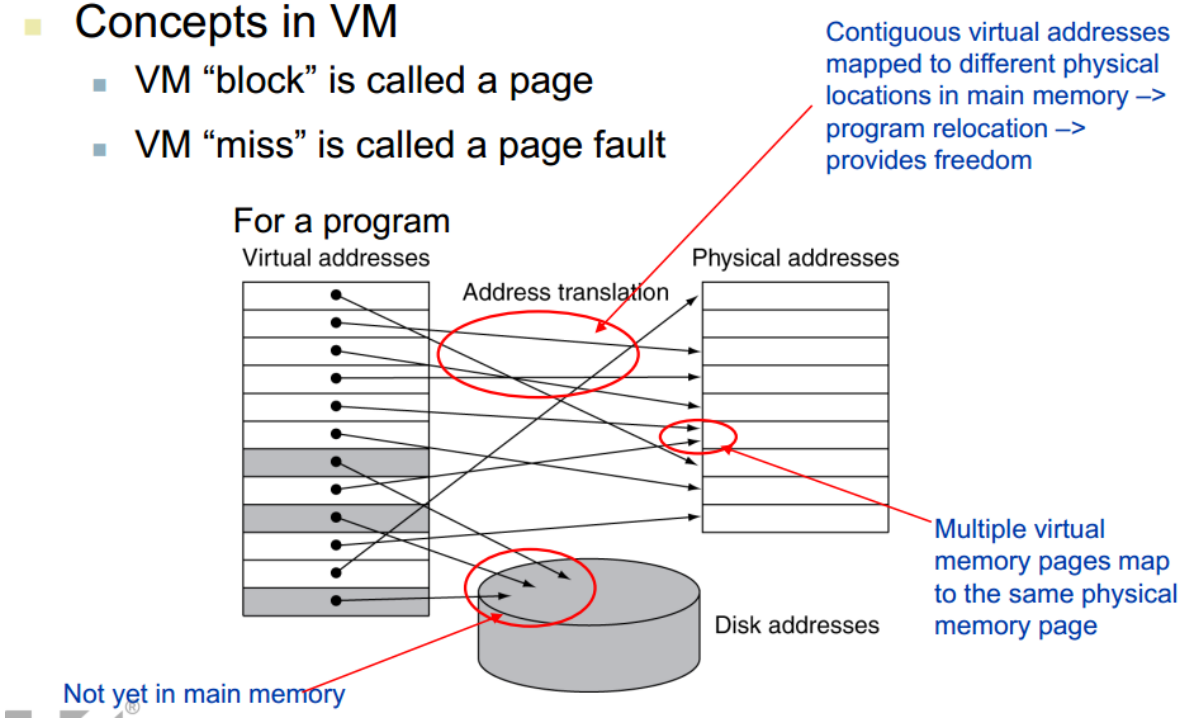
- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

L12 Virtual Memory

- What is VM, why use VM? L12p3
  - VM is the memory that programs think they have
    - VM allows sharing of main memory among many programs
      - Each gets a private virtual address space holding all of its code and data
      - CPU and OS translate **virtual addresses** to **physical addresses**
      - Protected from other programs
    - VM allows programs to exceed the main memory size limit
      - Programs are divided into self-contained, mutually exclusive small pieces – **overlay**
        - Each overlay contains both code and data
        - Each overlay is smaller than main memory
      - Overlays are loaded and swapped to fit in smaller main memory
      - Main memory size might not be a major issue with modern computers, still a concern for multithread processing and embedded computers
    - Managed jointly by CPU hardware and the operating system

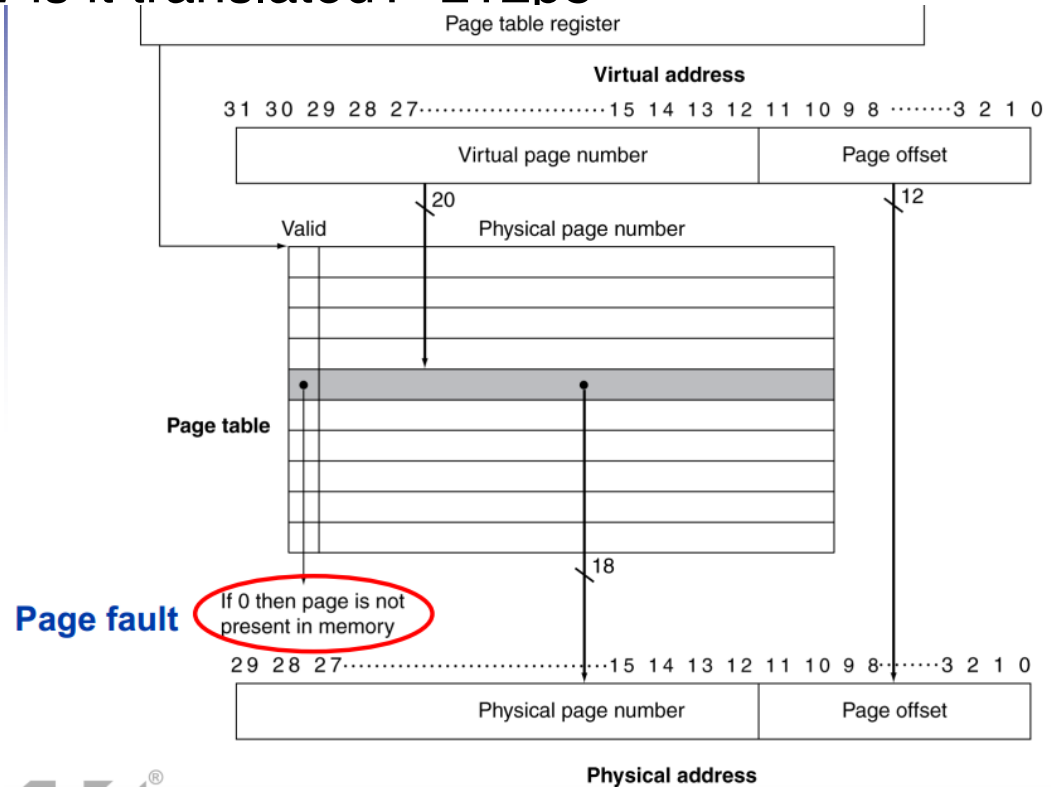
L12 Virtual Memory

- What is Page, What is Page fault? L12p4
  - Use main memory as a “cache” for hard disk
  - Concepts in VM
    - VM “block” is called a page
    - VM “miss” is called a page fault



L12 Virtual Memory

- How is it translated? L12p8



L12 Virtual Memory

- Page table related calculation (size, entries) L12p17

  Example:

  - Page size: 4KB
  - 32-bit virtual byte address (4GB)
  - 4 bytes per page table entry

  Number of page table entries = $2^{(32-12)} = 2^{20}$

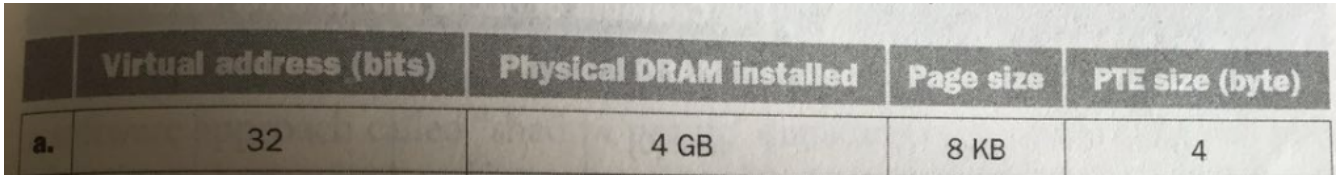
  - Up to, might be customized for each program and usually less than that

  Size of page table = number of page table entries x bytes/page table entry

  - Page table size = 2^{20} x 4 = 4 MB

L12 Virtual Memory

- Page table related calculation Example 5.11.1(a)



	Virtual address (bits)	Physical DRAM installed	Page size	PTE size (byte)
a.	32	4 GB	8 KB	4

For a single-level page table, how many page table entries are needed? How much physical memory is needed for storing the page table?

Page size: 8KB = 2^13 bytes.
32-13 = 19. -> 2^19 entries
Page table size: 2^19 * 4 = 2^21 bytes = 2MB

L12 Virtual Memory

- What is TLB and Why TLB? L12p19-20
  - Load part of the Page Table in cache
  - One translation per TLB entry
  - Status bits
    - Valid: empty or not
    - Dirty: target memory was changed
    - Reference: target memory was used
  - Full Associativity
    - Lower miss rate
    - Small TLB, access time not a major concern
    - LRU or random replacement

L12 Virtual Memory

- Example: 5.10.1

The following list is a stream of virtual addresses as seen on a system. Assume 4-KB pages and a four-entry fully associative TLB with LRU replacement policy. If pages must be brought in from disk, give them the next largest unused page number (i.e., starting at 13).

The Stream is 4095->31272->15789->15000->7193->4096->8912

L12 Virtual Memory

- Example: 5.10.1

  Pay attention to the base!

Stream: 0x0FFF -> 0x7A28 -> 0x3DAD -> 0x3A98 -> 0x1C19 -> 0x1000 -> 0x22D0

Initial TLB:(base-10)

Valid	Tag	PP#	LRU
1	11	12	2
1	7	4	3
1	3	6	4
0	4	9	1

Initial Page Table:

Index	Valid	Physical Page or On Disk
0	1	5
1	0	D
2	0	D
3	1	6
4	1	9
5	1	11
6	0	D
7	1	4
8	0	D
9	0	D
10	1	3
11	1	12

L12 Virtual Memory

- Example: 5.10.1

Stream: 0x0FFF -> 0x7A28 -> 0x3DAD -> 0x3A98 -> 0x1C19 -> 0x1000 -> 0x22D0

The LRU column works as follows: The older an entry is, the lower its LRU number will be. Each time an entry is used, its LRU number is set to 4 (since it is now the most-recently used), and the other numbers are adjusted downward accordingly.

L12 Virtual Memory

- Example: 5.10.1

Stream: 0x0FFF -> 0x7A28 -> 0x3DAD -> 0x3A98 -> 0x1C19 -> 0x1000 -> 0x22D0

Since each page is 4 KB (2^12 bytes), the lowest 12 bits of the address are the page offset and can be ignored. The page number or tag is the remaining upper 20 bits.

If there is a match we have a TLB hit (H), and all we have to do is update the LRU bits.

If there is no match in the TLB, we have to check the page table. Use the page number (tag) as an index into the page table. If the entry is valid, we have a TLB miss (M). Evict the least recently used entry in the TLB and replace it with the new translation. Remember to update all of the LRU bits and set the valid bit as well.

Finally, if the entry in the page table is invalid, our page is on disk and we have a page fault (PF). Assign it a new page number (starting with 13), and set its valid bit in the page table. Then you must also update the TLB.

L12 Virtual Memory

Valid	Tag	PP#	LRU
1	11	12	2
1	7	4	3
1	3	6	4
0	4	9	1

- Example: 5.10.1

Stream: 0x0FFF -> 0x7A28 -> 0x3DAD -> 0x3A98 -> 0x1C19 -> 0x1000 -> 0x22D0

Address	(H, M, PF)?
0x0FFF	M
0x7A28	H
0x3DAD	H
0x3A98	H
0x1C19	PF
0x1000	H
0x22D0	PF

Valid	Tag	PP#	LRU
1	11	12	1
1	7	4	2
1	3	6	3
1	0	5	4

Valid	Tag	PP#	LRU
1	11	12	1
1	7	4	4
1	3	6	2
1	0	5	3

Valid	Tag	PP#	LRU
1	11	12	1
1	7	4	3
1	3	6	4
1	0	5	2

Valid	Tag	PP#	LRU
1	11	12	1
1	7	4	3
1	3	6	4
1	0	5	2

Valid	Tag	PP#	LRU
1	1	13	4
1	7	4	2
1	3	6	3
1	0	5	1

Valid	Tag	PP#	LRU
1	1	13	4
1	7	4	2
1	3	6	3
1	0	5	1

Valid	Tag	PP#	LRU
1	1	13	3
1	7	4	1
1	3	6	2
1	2	14	4

Index	Valid	Physical Page or On Disk
0	1	5
1	1	13
2	1	14
3	1	6
4	1	9
5	1	11
6	0	D
7	1	4
8	0	D
9	0	D
10	1	3
11	1	12

L12 Virtual Memory

- Relationship in Memory Hierarchy L12p28

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.

  - Assume cache uses physical addresses

Cache

Given

- 1K blocks in cache
- 16 words in each block
- 32-bit byte address 0x81002345 requested by CPU

Show address and contents of the target cache block

- 0x81002345=0b 10000001000000000010001101000101
- Byte offset: 2 word offset: $\log_2^{16}$=4 Index offset: $\log_2^{1048}$=10
- Tag size = 32-(4+10+2)=16 bits

Index	Valid	Tag	Word 1, word 2,…, word 16
0x8D	1	0x8100	Mem[204008D0], Mem[204008D1],…, Mem[204008DF]

Direct-mapped, 1 block = 4 words, Cache size = 16 words or 4 blocks (hence cache blocks are 0, 1, 2, 3). The series of address references given as word addresses are: 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17.

Fill in the following blanks:

Reference	Hit or Miss	Contents
1	?	?
4	?	?
8	?	?
….	?	?

Reference	Hit or miss	Comments
1	Miss	Hence fetch words 0, 1, 2, 3 (cache block 0).
4	Miss	Hence fetch words 4, 5, 6, 7 (cache block 1).
8	Miss	Hence fetch words 8, 9, 10, 11 (cache block 2).
5	Hit	Already in cache: word 5 (in cache block 1).
20	Miss	Hence fetch words 20, 21, 22, 23 (cache block 1), replaces words 4, 5, 6, 7.
17	Miss	Hence fetch words 16, 17, 18, 19 (cache block 0), replaces words 0, 1, 2, 3.
19	Hit	Word 19 already exists in cache block 0.
56	Miss	Hence fetch words 56, 57, 58, 59 (cache block 2), replaces words 8, 9, 10, 11.
9	Miss	Hence fetch words 8, 9, 10, 11 (cache block 2), replaces words 56, 57, 58, 59.
11	Hit	Word 11 already exists in cache block 2.
4	Miss	Hence fetch words 4, 5, 6, 7 (cache block 1).
43	Miss	Hence fetch words 40, 41, 42, 43 (cache block 2) replaces words 8, 9, 10, 11.
5	Hit	Word 5 already exists in cache block 1.
6	Hit	Word 6 already exists in cache block 1.
9	Miss	Hence fetch words 8, 9, 10, 11 (block 2).
17	Hit	Word 17 already exists in cache block 0.

Final state of the cache:

Cache block #	Words
0	16, 17, 18, 19.
1	4, 5, 6, 7.
2	8, 9, 10, 11.
3	Empty block.

2-way set-associative cache, 1 block = 1 word, Cache size = 16 words. LRU. The series of address references given as word addresses are: 1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17.

Fill in the following blanks:

Reference	Hit or Miss	Contents
1	?	?
4	?	?
8	?	?
....	?	?

Reference	Hit or miss	Comments
1	Miss	Hence fetch word 1 to set 1.
4	Miss	Hence fetch word 4 to set 4.
8	Miss	Hence fetch word 8 to set 0.
5	Miss	Hence fetch word 5 to set 5.
20	Miss	Hence fetch word 20 to set 4 (no need to replace word 4).
17	Miss	Hence fetch word 17 to set 1 (no need to replace word 1).
19	Miss	Hence fetch word 19 to set 3.
56	Miss	Hence fetch word 56 to set 0, (no need to replace word 8).
9	Miss	Hence fetch word 9 to set 1 (set 1 already contains two words, 1 & 17, hence word 1 is replaced by word 19 due to LRU).
11	Miss	Hence fetch word 11 to set 3 (no need to replace word 19).
4	Hit	Word 4 already exists in set 4.
43	Miss	Hence fetch word 43 to set 3 (set 3 already contains two words, 11 & 19, hence word 19 is replaced by word 43 due to LRU).
5	Hit	Word 5 already exists in set 5.
6	Miss	Hence fetch word 6 to set 6.
9	Hit	Word 9 already exists in set 1.
17	Hit	Word 17 already exists in set 1.

Hence the final state of the cache with LRU order shown right to left:

Set #	Block	Block
0	56	8
1	17	9
2		
3	43	11
4	4	20
5	5	
6	6	
7		

Multilevel Cache AMAT

- AMAT = L1 HT + L1 MR × L1 MP
  - Now L1 MP depends on other cache levels
- L1 MP = L2 HT + L2 MR × L2 MP
  - If more levels, then continue this chain
    - (i.e. $MPi = Ht^{i+1} + Mr^{i+i} \times Mp^{i+i}$)
  - Final MP is main memory access time
- For two levels:
  - AMAT = L1 HT + L1 MR × (L2 HT + L2 MR × L2 MP)

Example

- Processor specs: 1 cycle L1 HT, 2% L1 MR, 5 cycle L2 HT, 5% L2 MR, 100 cycle main memory HT
- Without L2:
  - AMAT1 = 1 + 0.02 × 100 = 3
- With L2:
  - AMAT2 = 1 + 0.02 × (5 + 0.05 × 100) = 1.2

HW9 Solution

5.10.1

b.

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}
2 4EC, Page Fault, disk => physical page D, (=> TLB slot 3)
7 8F4, Hit in TLB
4 AC0, Miss in TLB, (=> TLB slot 0)
B 5A6, Miss in TLB, (=> TLB slot 2)
9 4DE, Page Fault, disk => physical page E, (=> TLB slot 3)
4 10D, Hit in TLB
B D60 Hit in TLB

TLB

Valid	Tag	Physical Page
1	4	9
1	7	4
1	B	C
1	9	E

Page table

Valid	Physical page
1	5
0	disk
1	D
1	6
1	9
1	B
0	disk
1	4
0	disk
1	E
1	3
1	C

5.10.2

a.

Virtual page number: Address >> 14 bits
H: Hit in TLB, M: Miss in TLB hit in page table, PF: Page Fault

0, 3, 1, 1, 0, 0, 1 (M, H, PF, H, H, H, H)

TLB

Valid	Tag	Physical Page Number
1	1	13
1	7	4
1	3	6
1	0	5

Page table

Valid	Physical page or in disk
1	5
1	13
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

Larger page sizes allow for more addresses to be stored in a single page, potentially decreasing the amount of pages that must be brought in from disk and increasing the coverage of the TLB. However, if a program uses addresses in a sparse fashion (for example randomly accessing a large matrix), then there will be an extra penalty from transferring larger pages compared to smaller pages.

5.10.2

b.

Binary address: (all hexadecimal), {bits 15–14 virtual page, 13–0 page offset}
0 24EC, Miss in TLB, (=> TLB slot 3)
1 38F4, Page Fault, disk => physical page D, (=> TLB slot 1)
1 0AC0, Hit in TLB
2 35A6, Page Fault, disk => physical page E, (=> TLB slot 2)
2 14DE, Hit in TLB
1 010D, Hit in TLB
2 3D60, Hit in TLB

TLB

Valid	Tag	Physical Page
1	B	C
1	1	D
1	2	E
1	0	5

Page table

Valid	Physical page
1	5
1	D
1	E
1	6
1	9
1	B
0	disk
1	4
0	disk
0	disk
1	3
1	C

Larger page sizes allow for more addresses to be stored in a single page, potentially decreasing the amount of pages that must be brought in from disk and increasing the coverage of the TLB. However, if a program uses addresses in a sparse fashion (for example randomly accessing a large matrix), then there will be an extra penalty from transferring larger pages compared to smaller pages.

5.10.3

a.

Virtual page number: Address >> 12 bits

0, 7, 3, 3, 1, 1, 2 => 0, 111, 011, 011, 001, 001, 010

2 way set associative:
Tag: VPN >> 1 bit

TLB

Valid	Tag	PPN	Valid	Tag	PPN
1	0	5	1	01	14
1	0	13	1	01	6

Direct-mapped:
Tag: VPN >> 2 bits

TLB

Valid	Tag	Physical Page Number
1	0	5
1	0	13
1	0	14
1	0	6

The TLB is important to avoiding paying high access times to memory in order to translate virtual addresses to physical addresses. If memory accesses are frequent, then the TLB will become even more important. Without a TLB, the page table would have to be referenced upon every access using a virtual addresses, causing a significant slowdown.

5.10.3

b.

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}

2 4EC, Page Fault, disk => physical page D, (=> TLB set 0, slot 1)
7 8F4, Miss in TLB, (=> TLB set 1, slot 1)
4 AC0, Miss in TLB, (=> TLB set 0, slot 0)
B 5A6, Miss in TLB, (=> TLB set 1, slot 0)
9 4DE, Page Fault, disk => physical page E, (=> TLB set 1, slot 1)
4 10D, Hit in TLB
B D60 Hit in TLB

2-way set associative TLB {bits 15–12 virtual page => bits 15–13 tag, bits 12 set}
(note: time stamps according to at start physical page numbers)

Valid	Tag(/Time)	Physical Page
1	4 /4	9
1	2 /2	D
1	B /5	C
1	9 /4	E

Binary address: (all hexadecimal), {bits 15–12 virtual page, 11–0 page offset}
2 4EC, Page Fault, disk => physical page D, (=> TLB slot 2)
7 8F4, Hit in TLB
4 AC0, Miss in TLB, (=> TLB slot 0)
B 5A6, Miss in TLB, (=> TLB slot 3)
9 4DE, Page Fault, disk => physical page F, (=> TLB slot 1)
4 10D, Hit in TLB
B D60 Hit in TLB

direct-mapped TLB {bits 15–12 virtual page => bits 13–12 TLB slot}

Valid	Tag	Physical Page
1	4	9
1	9	F
1	2	D
1	B	C

The TLB is important to avoiding paying high access times to memory in order to translate virtual addresses to physical addresses. If memory accesses are frequent, then the TLB will become even more important. Without a TLB, the page table would have to be referenced upon every access using a virtual addresses, causing a significant slowdown.

5.10.4

	Virtual address size	Page size	Page table entry size
a.	32 bits	4 KB	4 bytes
b.	64 bits	16 KB	8 bytes

5.10.4 [5] <5.4> Given the parameters in the table above, calculate the total page table size for a system running five applications that utilize half of the memory available.

#Page offset bit = $\log_2^{\text{page size}}$

#Page number bit = #address bit - #page offset bit

page table size per application = # page entries * entry size

a.	4 KB page = 12 offset bits, 20 page number bits 2^{20} = 1 M page table entries 1 M entries × 4 bytes/entry = ~4 MB (2^{22} bytes) page table per application 2^{22} bytes × 5 apps = 20.97 MB total
b.	virtual address size of 64 bits 16 KB (2^{14}) page size, 8 (2^3) bytes per page table entry 64 – 14 = 40 bits or 2^{40} page table entries with 8 bytes per entry, yields total of 2^{43} bytes for each page table Total for 5 applications = 5 × 2^{43} bytes

5.11.1

a.	virtual address 32 bits, physical memory 4 GB page size 8 KB or 13 bits, page table entry 4 bytes or 2 bits #PTE = 32 – 13 = 19 bits or 512K entries PT physical memory = 512K × 4 bytes = 2 MB
b.	virtual address 64 bits, physical memory 16 GB page size 4 KB or 12 bits, page table entry 8 bytes or 3 bits #PTE = 64 – 12 = 52 bits or 2^{52} entries PT physical memory = $2^{52} \times 2^{3} = 2^{55}$ bytes

5.11.2

a.	virtual address 32 bits, physical memory 4 GB page size 8 KB or 13 bits, page table entry 4 bytes or 2 bits #PTE = 32 – 13 = 19 bits or 512K entries 8 KB page/4 byte PTE = 2^{11} pages indexed per page Hence with 2^{19} PTEs will need 2-level page table setup. Each address translation will require at least 2 physical memory accesses.
b.	virtual address 64 bits, physical memory 16 GB page size 4 KB or 12 bits, page table entry 8 bytes or 3 bits #PTE = 64 – 12 = 52 bits or 2^{52} entries 4 KB page/8 byte PTE = 2^{9} pages indexed per page Hence with 2^{52} PTEs will need 6-level page table setup. Each address translation will require at least 6 physical memory accesses.

5.12.1(a)

Reference	Hit or Miss	Set 0	Set 0	Set 1	Set 1
0	M	0			
2	M	0	2		
4	M	4	2		
0	M	4	0		
2	M	2	0		
4	M	2	4		
0	M	0	4		
2	M	0	2		
4	M	0	4		

0 HIT

5.12.1(b)

Reference	Hit or Miss	Set 0	Set 0	Set 1	Set 1
0	M	0			
2	M	0	2		
4	M	4	2		
2	H	4	2		
0	M	0	2		
2	H	0	2		
4	M	4	2		
0	M	4	0		
2	M	2	0		

2 HIT

5.12.2(a)

Reference	Hit or Miss	Set 0	Set 0	Set 1	Set 1
0	M	0			
2	M	0	2		
4	M	0	4		
0	H	0	4		
2	M	2	4		
4	H	2	4		
0	M	2	0		
2	H	2	0		
4	M	4	0		

3 HIT LRU is not always the best way