Department of Electrical and Computer Engineering
University of Texas at Austin

EE460N Fall 2020
Y. N. Patt, Instructor
Chester Cai, Sean Stephens, Arjun Ramesh, TAs
Final Exam
December 11th, 2020

Name:	Solution
EID:	sol012

Part A:	Part B:
Problem 1: 10 points	Problem 6: 20 points
Problem 2: 10 points	Problem 7: 20 points
Problem 3: 10 points	Problem 8: 20 points
Problem 4: 10 points	Problem 9: 20 points
Problem 5: 10 points	

Total: 130 points

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Please read the following sentence, and if you agree, sign/print your name where requested: I have not given or received any unauthorized help on this exam.

Signature:	*Hook 'Em*

Good Luck!

General Instructions:

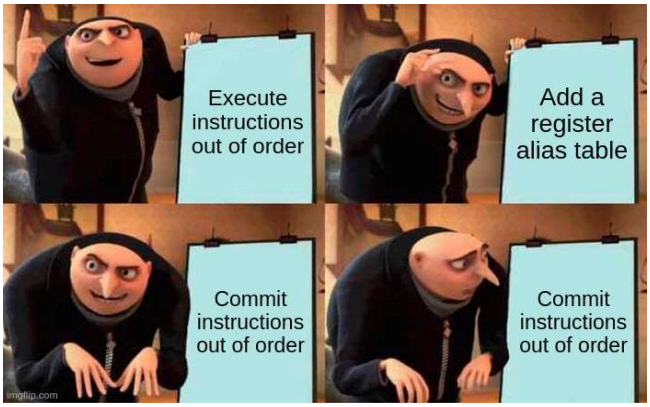
1. You are free to use anything in the Handouts section of the course website that is listed under "Course Related Handouts" or "LC-3b Handouts." In particular, Appendix A and Appendix C may be of use. Anything other than that from the course website, textbooks, or the Internet is not allowed and considered unauthorized access.
2. If you have any questions, join the class Zoom link and ask a TA. You do not need to stay on the Zoom call during the exam unless you have questions.
3. Announcements will be posted here. Check this page periodically throughout the exam.
4. You may take the exam by printing it, editing a PDF, or editing a Google Doc. Read the instructions for your preferred method below.
5. **You are required to stop working on the exam promptly at 10:00 PM.**

Printing or editing a PDF:

1. Download and save the PDF.
2. Edit the PDF to fill in answers with a software of your choice. Feel free to show your work in the available space. You may also choose to print the exam and solve it on paper.
3. When you are ready to submit your exam, save the edited PDF as "Final Exam <your name>"; if you printed your exam, scan in your written answers as a PDF with the same name. You may use a scanner or an app such as CamScanner.
4. Upload the PDF to Gradescope by 10:10 PM. The entry code for Gradescope is **9RPGX3**.

Editing a Google Doc:

1. Save a copy of the document to your Google Drive.
2. While working on the exam, **DO NOT expand any boxes that are given to you.** Feel free to show your work in the available space. If you need more space, you are writing too much.
3. When you are ready to submit your exam, click "File"-> "Print" and select "Save as PDF". Save the edited PDF as "Final Exam <your name>".
4. Upload the PDF to Gradescope by 10:10 PM. The entry code for Gradescope is **9RPGX3**.



Problem 1 (10 points): You are the new chief architect of Little Computer Company. You have lots of great ideas about improving their next generation CPU, but are told by the CEO that the ISA MUST remain unchanged. Which of the following improvements will satisfy this constraint?

For each improvement, write "Yes" if the improvement can be incorporated, or "No" if the improvement would not be allowed. Each correct answer earns 2 points. Each incorrect answer earns minus 2 points. Leaving an answer blank earns 0 points.

1. Adding a gshare branch predictor

   Yes

2. Adding a new instruction for vector processing

   No

3. Changing the memory bus width from 16 bits to 32 bits

   Yes

4. Making the cache larger

   Yes

5. Changing the size of the PTE

   No

Problem 2 (10 points): A byte-addressable machine with a 4GB physical memory has a 2-way physically addressed, set associative cache. The cache is write-back, and it uses LRU replacement policy. The line size is 64 bytes. The cache's data store has a capacity of 16KB.

Your job: Provide the following information

Number of cache lines in the data store:

16 KB / 64 B = 256 lines

Number of index bits:

256 lines / 2 lines per set = 128 Sets → 7 index bits

Total number of bits of storage required to implement the tag store:
(Feel free to leave this as an equation)

4 GB memory → 32 bit address
64 B lines → 6 bit offset
32 - 6 - 7 = 19 bit tag
Per-line bits: 1 valid + 1 dirty + 19 tag bits
Per-way bits: 1 LRU bit
256*(1 + 1 + 19) + 128*1 = 5504 tag store bits

Problem 3 (10 + 1 points): Assume a IEEE-like floating point data type, wherein each number consists of 7 bits, of which three bits are used for the exponent. We wish to compute the sum of A + B + C, where A, B, and C are represented as follows:

A:	0	1	1	0	1	0	1
B:	0	0	1	0	0	0	0
C:	0	0	1	0	0	0	0

The add instruction adds two 7-bit floating point numbers, rounds the result towards 0, and stores it in a 7-bit floating point register. Assume the microarchitecture has no round, guard, or sticky bits.

Part a (5 points): What is the sum if you add A+B first? i.e., (A+B) + C.

0	1	1	0	1	0	1

Part b (5 points): What is the sum if you add B+C first? i.e., A + (B+C).

0	1	1	0	1	1	0

Part c (1 bonus point): If you were asked to sum an array of N positive floating point numbers, some very large and some very small, in which of the three following orders would you perform the N-1 additions in order to make the result as accurate as possible? Explain your choice in 20 words or fewer.

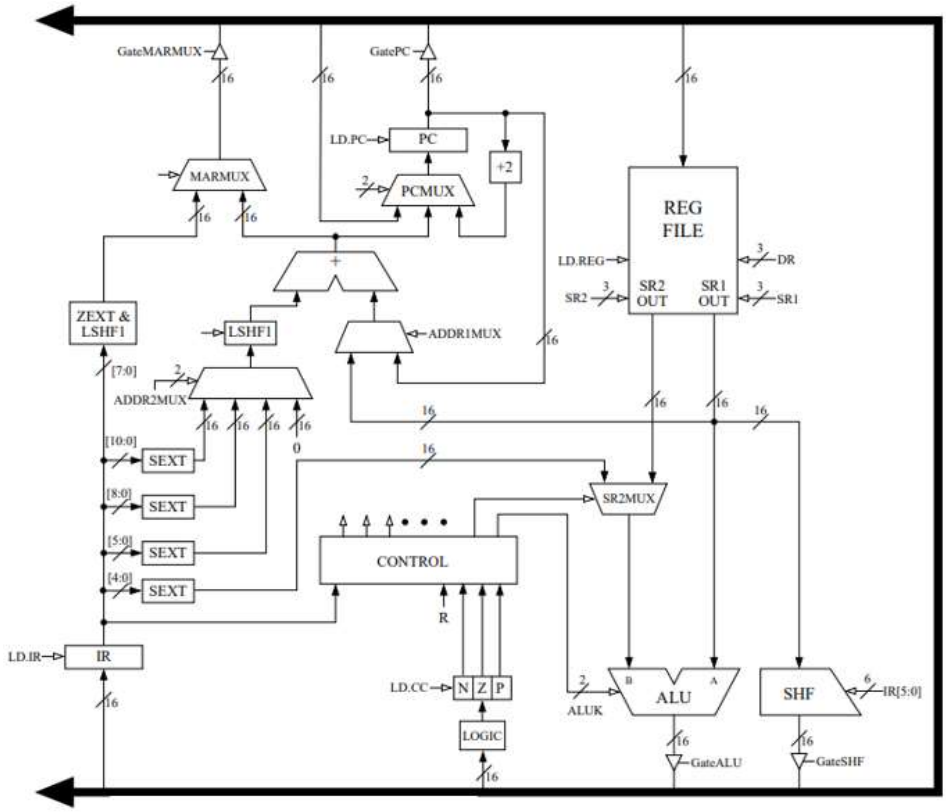
Option A: Sort the array from largest to smallest, then add the numbers from the beginning to the end of the array
Option B: Sort the array from smallest to largest, then add the numbers from the beginning to the end of the array
Option C: Don't sort the array, just add the numbers from the beginning of the array to the end of the array.

Option B, we want to add the smallest numbers together to maintain accuracy.

Problem 4 (10 points): While doing Lab3, a student found the following bugs:

- A taken branch always jumps to the instruction stored immediately after the branch
- All load and store instructions always set MAR to BaseR instead of BaseR + offset6
- JSR always jumps to the instruction stored immediately after the JSR instruction.

A picture of the datapath is shown below for your convenience.



Assuming there are no other bugs in the student's implementation, which of the following is a possible explanation of the bugs above? Explain your reasoning in fewer than 20 words.

Option 1: The ADDR1MUX's output is always PC

Option 2: The ADDR2MUX's output is always 0

Option 3: The PCMUX's output is always PC+2

Option 2. PCoffset9, offset6, and PCoffset11 are effectively stuck at 0, causing all these issues.

Problem 5 (10 Points): The best algorithm for a program running on a single processor takes 6 seconds. 75% of the program can run in parallel. The rest of the program has to run sequentially.

Part a (5 points): Ignoring communication overhead, what is the minimum number of cores needed to run the program with a speedup of 3?

Using Amdahl's law: $S = 1 / (\alpha / p + 1 - \alpha)$
$3 = 1 / (0.75 / p + 1 - 0.75)$
$p = 9$

Notice here the actual run time of the original program does not matter.

Part b (5 points): What is the theoretical maximum speedup that can be obtained by running the program on an unlimited number of cores?

$S = 1 / (0.75 / \infty + 1 - 0.75)$
$S = 1 / (0 + 1 - 0.75)$
$S = 4$

Problem 6 (20 points): A user program consisting of four instructions shown below is executed on a machine that implements Tomasulo's algorithm with an ROB to retire instructions in program order.

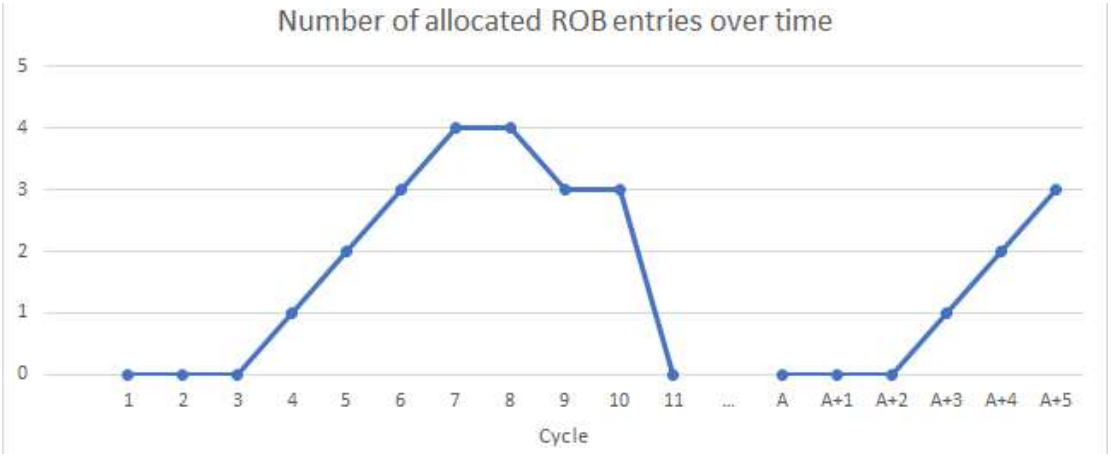
Inst 0	MUL R1, R0, R0
Inst 1	MUL R2, R1, R1
Inst 2	ADD R3, R4, R5
Inst 3	ADD R6, R2, R6

You are given the following information:

- The machine has a 5-stage pipeline: Fetch, Decode, Rename, Execute, and Retire.
- Fetch, Decode, Rename, and Retire each takes 1 cycle.
- Fetch, Decode, Rename, and Retires stages can only operate on one instruction at a time
- The machine has 1 non-pipelined adder and 1 non-pipelined multiplier.
- The result of each instruction is written to the ROB during the last cycle of its execution.
- Dependent instructions can start execution the cycle after the previous instruction finishes execution.
- During Rename, each instruction is assigned an ROB entry.
- All functional units share the same 5-entry reservation station
- The reservation station can be bypassed by instructions with no dependencies.

At some point during the instruction sequence, an interrupt occurs. In response, the machine immediately flushes all un-retired instructions. After executing the interrupt routine, the user program resumes execution.

The graph below shows the number of allocated ROB entries at the beginning of each cycle. A result is written to the ROB at the end of cycle 6 and another at the end of cycle 7. The allocated ROB entries during the interrupt service routine are not shown. The interrupt finishes on cycle A-1. We start fetching instructions from the user program again on cycle A.



Part a (2 points): In which cycle did the interrupt occur?

10

Part b (6 points): What is the latency for the add functional unit and multiply functional unit?

Adder: 1 cycle Multiplier: 4 cycles

Part c (2 points): What instruction is fetched during cycle A?

Inst 1

Part d (10 points): Complete the dynamic timing diagram below for the execution of the user program, as we have done in class. Show all instructions that are fetched regardless of whether they are committed or not.

- Use F, D, RN (for rename), A (for ADD), M (for MUL), and FL (for flush), and RT (for retire) to indicate what is going on with each instruction during each cycle
- Use * to indicate a clock cycle when an instruction is waiting
- Use as many rows/columns as needed

Cycle	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
I0	F	D	RN	M	M	M	M	RT				
I1		F	D	RN	*	*	*	M	M	FL		
I2			F	D	RN	A	*	*	*	FL		
I3				F	D	RN	*	*	*	FL		

Cycle	*A*	*A+1*	*A+2*	*A+3*	*A+4*	*A+5*	*A+6*	*A+7*	*A+8*	*A+9*	*A+10*	*A+11*
I1	F	D	RN	M	M	M	M	RT				
I2		F	D	RN	A	*	*	*	RT			
I3			F	D	*	*	*	A	*	RT		

Problem 7 (20 points):*(Note: You are allowed to use a calculator to solve parts of the question.)* A byte-addressable machine implements virtual memory with a 4 MB virtual address space using a three-level page table tree system similar to the x86 model. Each page table/page directory occupies exactly one frame of physical memory. All page tables live in physical address space.

The breakdown of the virtual address bits is as follows:

L1	L2	L3	Offset
5	5	5	7

Part a (2 points): How many PTEs can fit in a frame?

2^5 = 32

Part b (2 points): What is the PTE size in bytes?

2^7 / 2^5 = 4

Parts c, d and e

A user process runs on this machine that requires exactly 343,240 contiguous bytes of virtual memory and does not perform any dynamic memory allocation. The OS allocates at least this amount of memory to the process at the time it starts.

Part c (6 points): Assume the machine only supports a single page size (1 page = 1 frame). Find the minimum number of virtual pages and total number of page tables/directories at all levels required for the process. Show your work.

Virtual Pages: 2682 Page Tables: 88

Page size = 2^7 bytes = 128 B
Virtual pages = ceil(343,240 / 128) = 2,682

L3 PTEs = # virtual pages = 2,682
L3 page tables = ceil(2,682/32) = 84
L2 PTEs = # L3 page tables = 84
L2 page tables = ceil(84/32) = 3
L1 page tables = 1 (root)

Total page tables = 1 (L1) + 3 (L2) + 84 (L3) = 88

Part d (4 points): We wish to enhance this machine to support three different page sizes in the same way Intel did for the x86. What are the three page sizes?

2^7 = 128 B
2^(5+7) = 2^12 = 4 KB
2^(5+5+7) = 2^17 = 128 KB

Part e (6 points): The same process now runs on a machine with the three page sizes from Part d. The OS can allocate pages of different sizes to the same process at the same time. In this case, while keeping the internal fragmentation at an absolute minimum, it allocates the fewest number of pages possible. Find the total number of virtual pages and page tables/directories (at all levels) required for the process. Show your work.
(Note: Internal fragmentation is the difference between how much space is allocated to the process and how much space the process actually needs.)

Virtual Pages: 47 Page Tables: 3

Allocate as many pages as possible of the largest page size without going over the required amount of memory space. Then move to the medium size and repeat with the remaining space. Finally, repeat with the smallest size, but allocate one more (i.e. round up instead of down) so we allocate at least the required amount.

128 KB pages = floor(323,240 B / 128 KB) = 2
Remaining space = 323,240 B - 2*(2^17 B) = 81,096 B
4 KB pages = floor(81,096B / 4096 B) = 19
Remaining space = 81,096 B - 19*(4096 B) = 3,272 B
128 B pages = ceil(3,272 B / 128 B) = 26

virtual pages = 2 + 19 + 26 = 47

Each level (L1, L2, L3) has less than 32 PTEs so there is just one page table per level.

Problem 8 (20 points): Consider a machine with no cache, 4 KB of physical memory, and the following address format:

11 … n	n-1 … 5	4 … 2	1 … 0
Row	Column	Bank	Byte on Bus

Matrix A is a 16x16 matrix of 32-bit integers stored in *row-major* order beginning at physical address x300. The following code is run on the machine to sum the matrix, accessing each element in *column-major* order.

```
int sum = 0;
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        sum = sum + A[j][i];
    }
}
```

- All loop variables and sum are stored in registers.
- It takes 15 cycles to open a row.
- It takes 5 cycles to load a value from the row buffer.
- All row buffers are initially empty.
- Memory requests are issued in order.

Part a (2 points): Which elements of the 16x16 matrix are stored in Bank 2?

Columns 2 and 10

Part b (4 points): The access to A[0][0] is issued on Cycle 1 and returns on Cycle 20. The access to A[1][0] is issued on Cycle 21 and returns on Cycle 25. The access to A[3][0] returns on Cycle 50. How many row bits are in the physical address?

A total of 4 accesses are performed, all to the same bank. Row buffer misses take 15 + 5 = 20 cycles, and hits take 5 cycles. Therefore, 50 cycles means 2 RB misses and 2 RB hits were taken.So A[0][0] is on Row 0, and A[2][0] on Row 1.

A[0][0] is at address x300 (b0011 0000 0000)
A[1][0] is at address x340 (b0011 0100 0000)
A[2][0] is at address x380 (b0011 1000 0000)

We see that the miss occurred when bit 7 changed. So, n = 7. Total number of row bits = 5.

Part c (3 points): On what cycle does the access to A[15][0] return?

Every other access hits, so every pair of accesses takes 25 cycles.
16 accesses = 8 pairs
8 * 25 = 200
Cycle 200

Part d (3 points): On what cycle does the access to A[15][15] return?

Accesses must be issued in order. The final access to Bank 0 is issued on Cycle 196, which means the first access to Bank 1 is issued on Cycle 197 and returns on Cycle 216. With no interleaving, it would have returned on Cycle 220. Thus we save 4 cycles from interleaving on the first access to each bank after Bank 0.

200 + 196 * 15 = 3140

Zero-padding

Zero-padding is a technique which can be used to improve the performance of accessing a matrix by adding a column of zeros. Below is an example of a zero-padding.

$$Z = \begin{pmatrix} z_{00} & z_{01} \\ z_{10} & z_{11} \end{pmatrix} \qquad Z = \begin{pmatrix} z_{00} & z_{01} & 0 \\ z_{10} & z_{11} & 0 \end{pmatrix}$$

We zero-pad matrix A such that it is stored as a 16x17 matrix in physical memory. We run the same code again on the zero padded matrix. **Note**: We still set size to be 16, not 17, since it doesn't affect our final sum.

Part e (4 points): On what cycle does the access to A[15][0] return?

The padding makes the number of elements and number of banks relatively prime, so we now have interleaving on column-major order accesses. Within the column, we cycle through all banks twice. A[0][0] (Bank 0) returns on 20, A[1][0] (Bank 1) on 21…up to A[7][0] (Bank 7) which returns on 27. The access to A[8][0] loops back to Bank 0, so it returns on 40. Counting up to A[15][0] we arrive at our answer: 47.

Part f (4 points): On what cycle does the access to A[15][15] return?

Every other column access time = (20 - 6) + 27 (we skip a bank when we loop around)
Total access time = 47 + 41*15 = 662.
Alternatively, we can consider computing perfect banking, giving us 20*31 + 27 = 647, and add the 1 cycle cost for skipping a bank, giving 647 + 15 = 662.

Problem 9 (20 points):

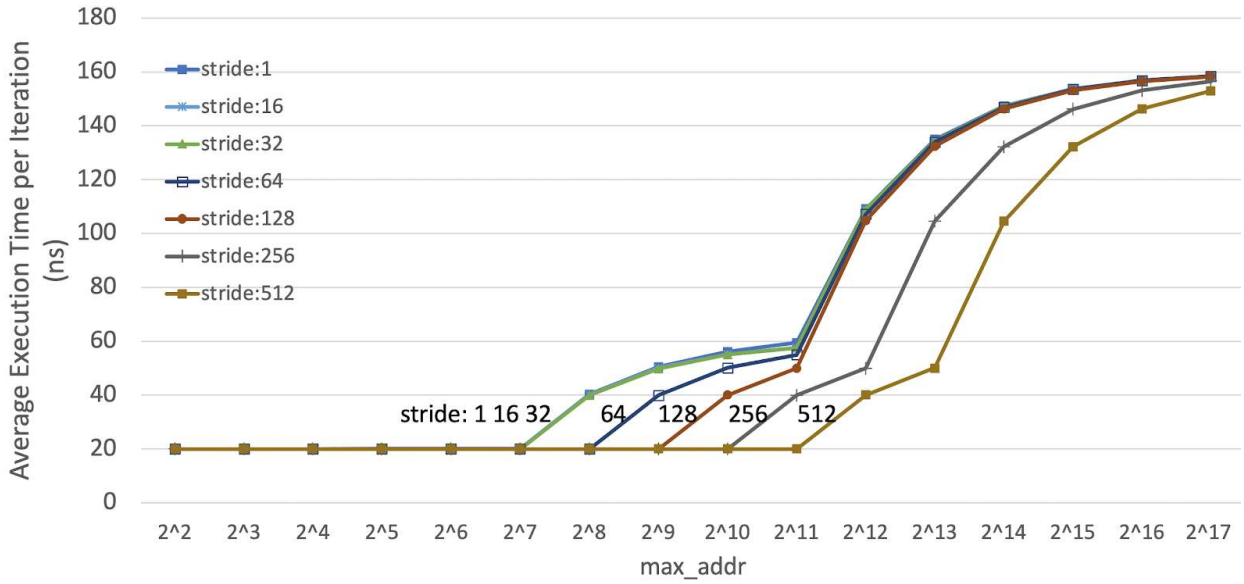
We are trying to figure out parameters of a cache hierarchy by running the following microbenchmark.

```
for i in 0..N:
      Addr = (random_int() * stride) % max_addr    // % is a modulo operator
      LoadByte from Mem[Addr]
```

Assume the following:

- `N` is a large number (we observe steady-state behavior).
- `stride` and `max_addr` are powers of two (as shown in the chart below).
- There are two levels of caches (L1 and L2) and main memory in the memory system.
- The execution time is determined only by cache and memory access latencies.
- All caches and memories have constant latency.
- All sizes and associativities are power of two numbers.
- Assume inclusive caches and LRU replacement where necessary.
- All cache lines are 16 bytes in size.

We measure the execution time of each run while varying the values of `stride` and `max_addr`. The following plot shows the average execution time per loop iteration, where each data point represents a separate experiment with different `stride` and `max_addr` values. Note that the lines for `stride`=1, 16, and 32 are all overlapped.



(Problem continues on the next page)

Answer the following questions.

Part a (4 points): What is the total cache size (in bytes) for each level of cache? Explain your answer briefly.

L1: 128 B	L2: 2048 B

Beyond these max_addr sizes, we observe that the latencies start to shoot up, indicating the respective caches are getting full.

Part b (6 points): What is the access latency (in ns) of each level of cache and main memory (starting from the previous level, e.g., CPU→L1, L1→L2, ..)? Round each latency to the nearest ten. Explain your answer briefly.

L1: 20 ns	L2: 40 ns	Main memory: 100 ns

When max_addr and stride are small enough to fit in the L1, the average latency is 20ns. As L1 misses increase, the latency goes up and converges around 60ns, which means that the L2 access latency is 40ns. Similarly, the overall latency at large max_addr converges at 160ns. Thus, the main memory latency is 100ns.

Part c (10 points): What is the set-associativity for each level of cache? Explain your answer briefly.

L1: 4-way	L2: 16-way

Even when max_addr is larger than the cache size, when stride is large enough such that all accesses map to and fit in a single set, misses in the corresponding cache drop. This happens at (max_addr=256, stride=64) for L1. The 4 (=256/64) addresses fit in a set, implying L1 is 4-way. Similarly, (max_addr=4096, stride=256) for L2 (4096/256 = 16-way).