Tutorial 1

ENGN8537

Embedded Systems

Lab 2 Debrief

The `Reset_Delay` module makes sure the LCD doesn't start receiving commands for some time after the FPGA starts up. Noting that the clock is 50MHz, how long does that module wait before it sets the reset line high?

Try commenting out the `Reset_Delay` module instantiation and replacing it with

```
assign DLY_RST = 1;
```

The program should still work. Given that, what's the point of having this delayed reset? Hint: Refer to the Reset Function extract of the LCD data sheet shown below and think: What's different between how you're using the FPGA in labs compared to how it might be used in an Embedded System?

Lab 2 Debrief

Reset Function

Initializing by Internal Reset Circuit

An internal reset circuit automatically initializes the HD44780U when the power is turned on. The following instructions are executed during the initiali[illegible] busy state until the initialization ends (BF = 1). The busy state last[illegible] for 10 ms after V_{CC} rises to 4.5 V.

1. Display clear
2. Function set:
   DL = 1; 8-bit interface data
   N = 0; 1-line display
   F = 0; 5 × 8 dot character font
3. Display on/off control:
   D = 0; Display off
   C = 0; Cursor off
   B = 0; Blinking off
4. Entry mode set:
   I/D = 1; Increment by 1
   S = 0; No shift

Note: If the electrical characteristics conditions listed under the table Power Supply Conditions Using Internal Reset Circuit are not met, the internal reset circuit will not operate normally and will fail to initialize the HD44780U. For such a case, initial-ization must be performed by the MPU as explained in the section, Initializing by Instruction.

Lab 2 Debrief

When did VCC rise to 4.5 Volts? When you turned the board on!

Given that it takes you several seconds at least to program the FPGA and get the thing running, you absolutely have waited the requisite 10ms after power on before the FPGA starts talking to the chip, regardless of whether or not you have the explicit ‘reset_delay’ module waiting a further 20ms.

In a real Embedded System, the FPGA would be programmed in a non-volatile fashion and would be ready to execute as soon as the power came on. In that case, the reset_delay module would be critical.

This is an example of an “initial condition” assumption which is different between development and release configurations. These are common, watch out for them!

Lab 2 Debrief

Why use a PLL when the divider worked? The 640 x 480 resolution was chosen as it requires a VGA_CLK of 25MHz; this is able to be generated from both Verilog and the PLL. If we want a different resolution with a different clock that isn't an integer divisor of the 50MHz system clock, we'd absolutely need a PLL.

The lab procedure was just presenting the simple case in order that it might actually fit in the 3 hour time slot!

Lab 2 Debrief

Most people got the gradient working and had something like

VGA_B = (CounterX + CounterY) * 256 / (640+480)

This is correct and works well. A few people questioned whether doing multiplications and divisions was safe on an FPGA. There's no doubt that doing multiplications and, especially, divisions takes a relatively large amount of hardware and that that hardware can be quite slow.

But is that a problem?

Lab 2 Debrief

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil****. Yet we should not pass up out opportunities in that critical 3%.*

Donald Knuth

Sure division might be slow, but is it *too* slow?

Lab 2 Debrief

```
VGA_R = CounterX * CounterX / 1280;
VGA_G = CounterY * (1024 – CounterY) / CounterX;
VGA_B = CounterX * (1248 – CounterX) * CounterY;
```

[demo]

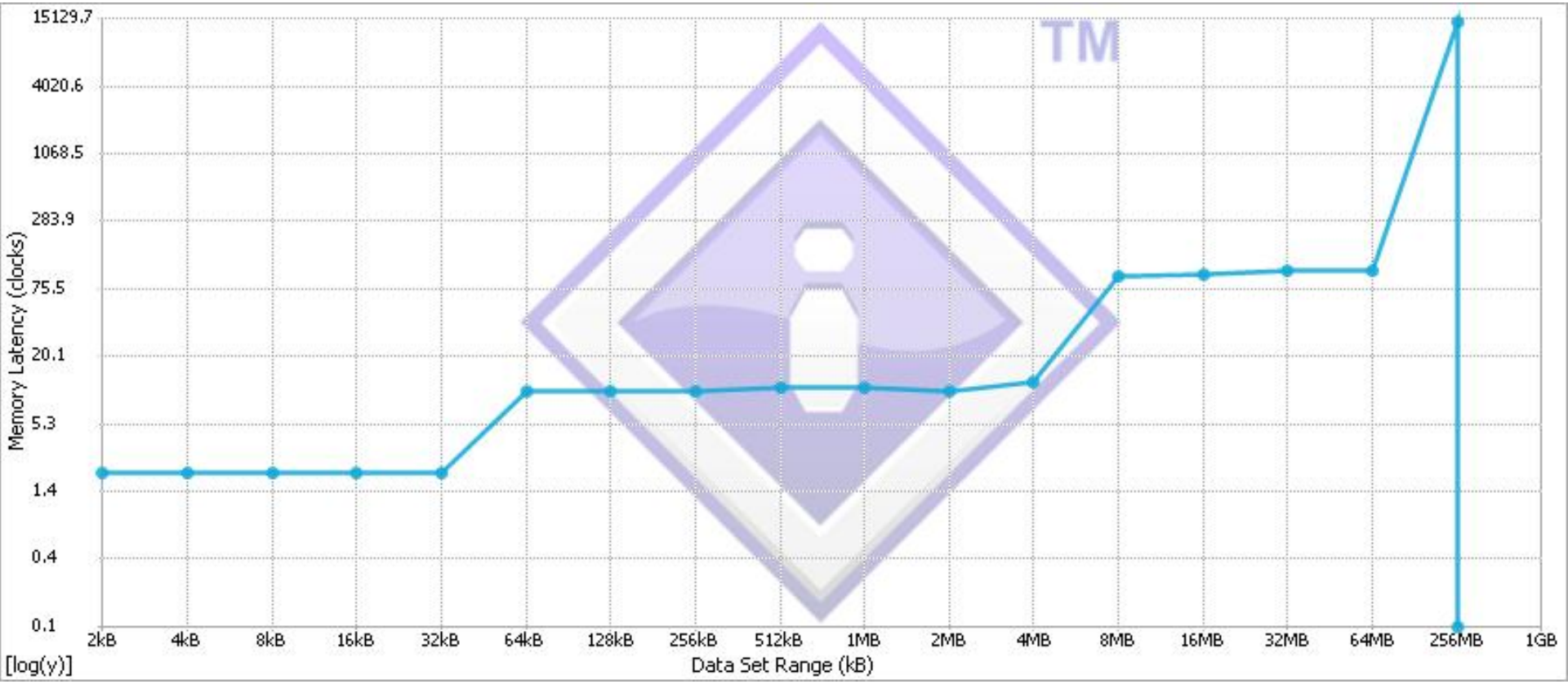
Tutorial Question 1:

Referring to the memory access latency benchmark on the next slide (and ignoring the last point at 0 which is a bug in the plotter!), answer the following:

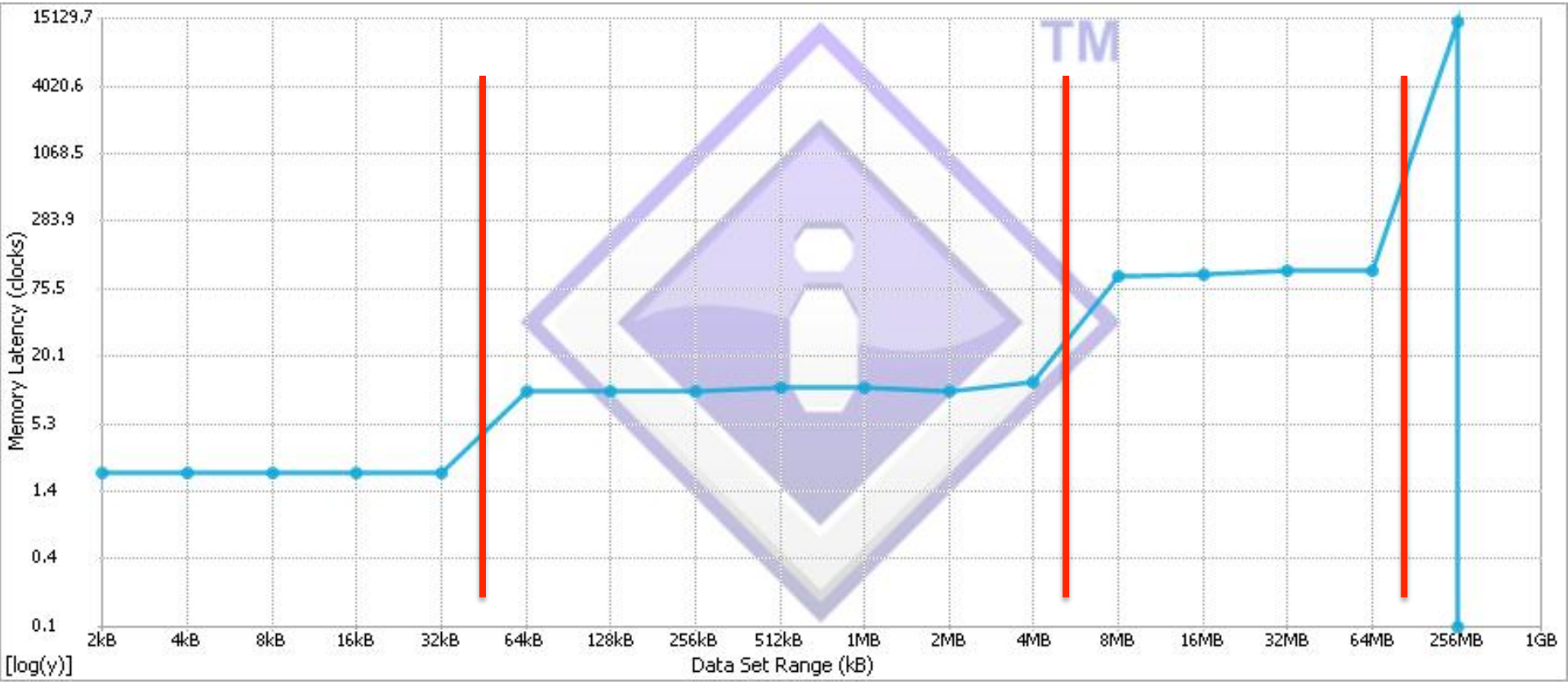
1. How many levels of cache does the processor have and (roughly) how big is each one? How big is Main Memory (RAM)?
2. One of the caches is 6MB, is it most likely to be Direct or Associative?

The plot is of steady-state sequential access latency. That is, a loop is run that accesses each memory location in turn, from address 0 up to the length under test. This loop is run several thousand times in succession and latency of each access timed and averaged. This means that any different timing on the first run through can be ignored.

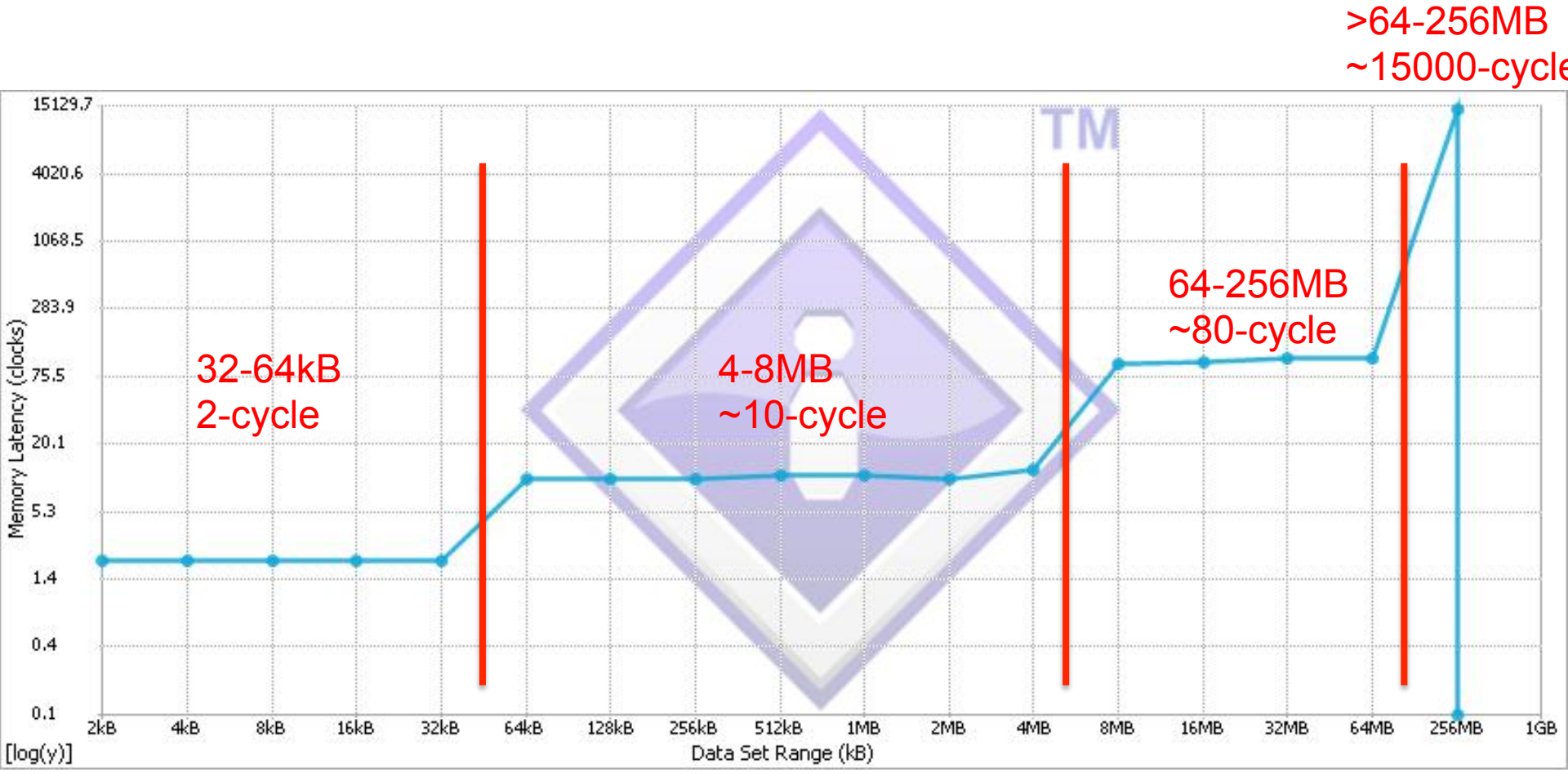
Tutorial Question 1:



Tutorial Question 1:



Tutorial Question 1:



Tutorial Question 1:

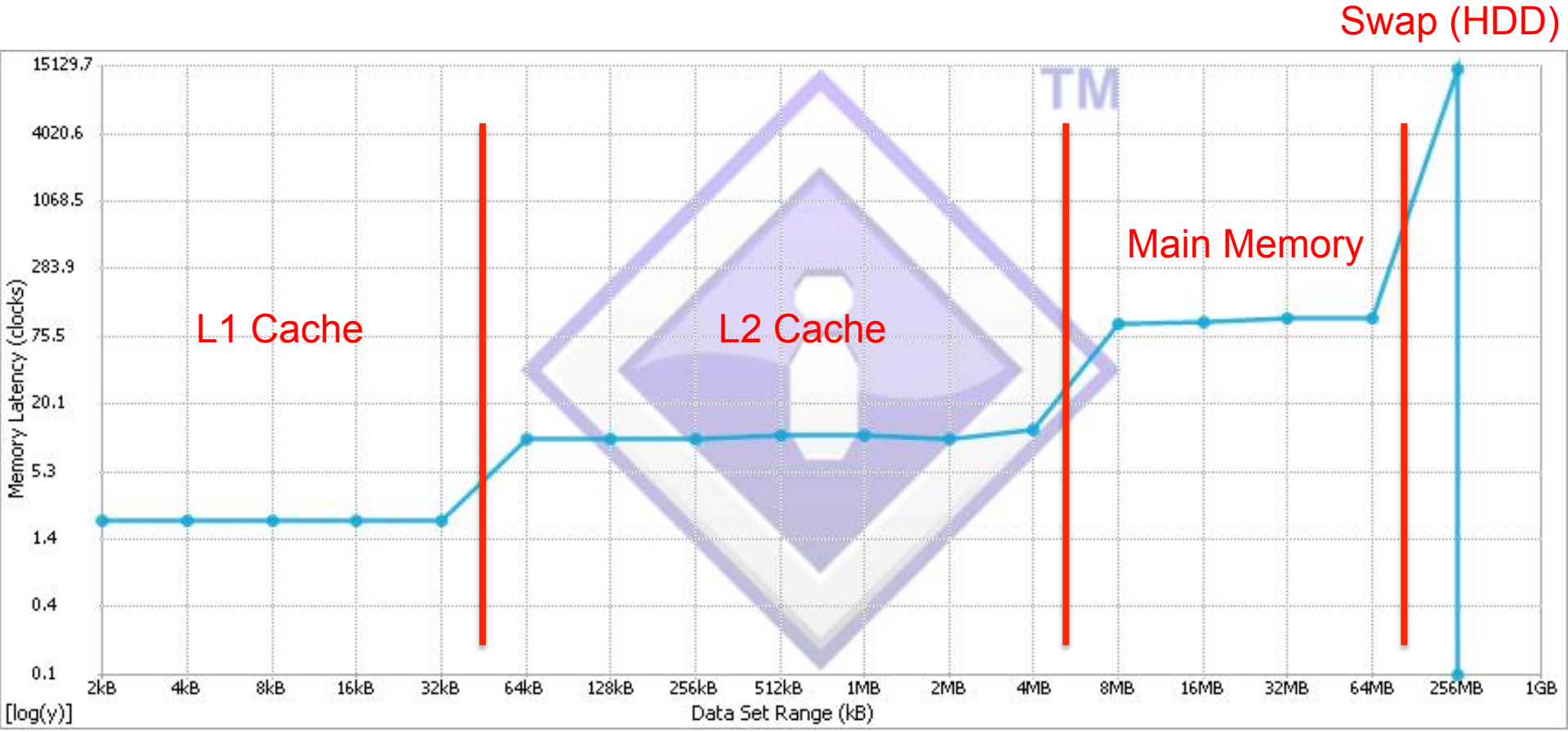
Hard drive contents can also be cached, this time in normal RAM. This is why if you have lots of programs open your computer it suddenly slows down: You've run out of real RAM and you're now using 'fake RAM' on the hard drive which is much slower.

This is also called 'swap'.

The fastest type of memory is the Cache memory inside the processor itself. We will discuss memories lower in the hierarchy later in the course, but today we concern ourselves only with Cache.

It is almost unimaginably small by today's standards: A few kB in L1 through to 8-24MB in L2/3

Tutorial Question 1:



Tutorial Question 1:

Referring to the memory access latency benchmark on the next slide (and ignoring the last point at 0 which is a bug in the plotter!), answer the following:

1. How many levels of cache does the processor have and (roughly) how big is each one? How big is Main Memory (RAM)?

Two levels of Cache

L1 is between 32- and 64kB, can't be exact
L2 is between 4- and 8MB, this agrees with the 6MB given in the second half of the problem.

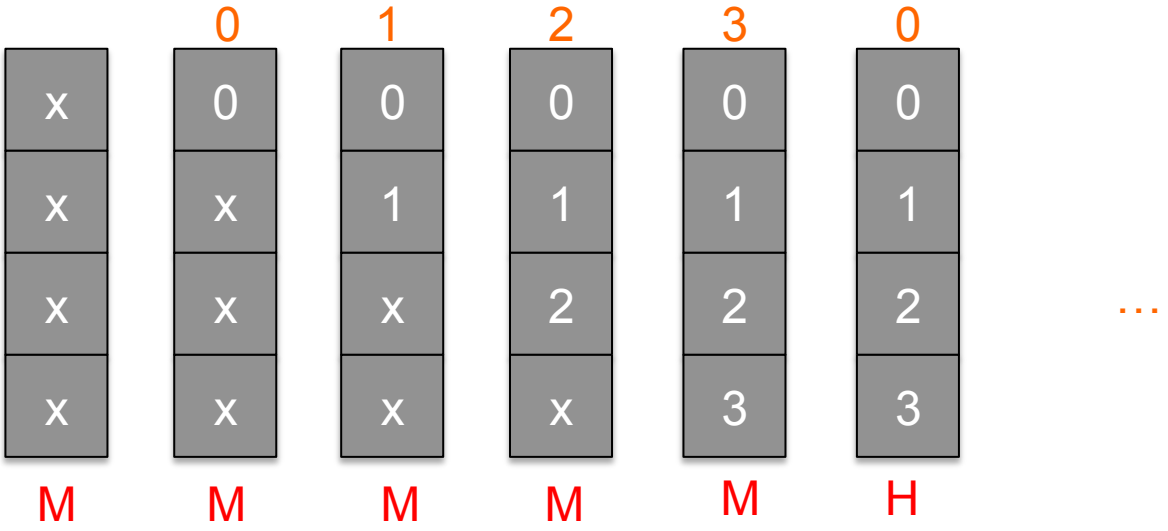
Main memory is between 64- and 256MB.

Tutorial Question 1:

One of the caches is 6MB, is it most likely to be Direct or Associative?

Let's examine what happens when a Direct Cache fills up. We'll demonstrate with four cache lines for simplicity, starting in an unknown state.

Access four locations:

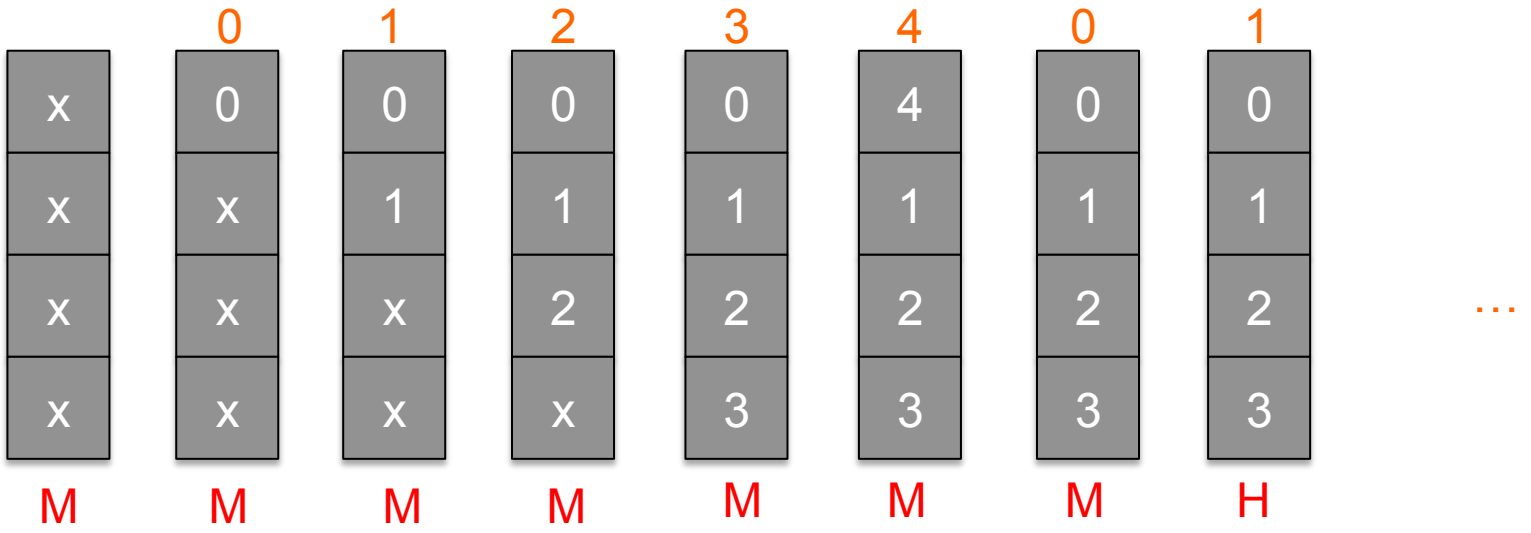


…

So in the steady state, after the first loop, we only 'hit'

Tutorial Question 1:

Access five locations:



So in the steady state, after the first loop, all accesses to the first line miss, everything else hits. Two out of five accesses are to the first line.

Tutorial Question 1:

In this case, address 0 and 4 are conflicted for the first line, 2/5 addresses miss.

If we accessed 6 locations, 0 and 4 would conflict for the first line and 1 and 5 would conflict for the second line: 4/6 addresses would miss.

With 7 locations, 6/7 addresses would miss.

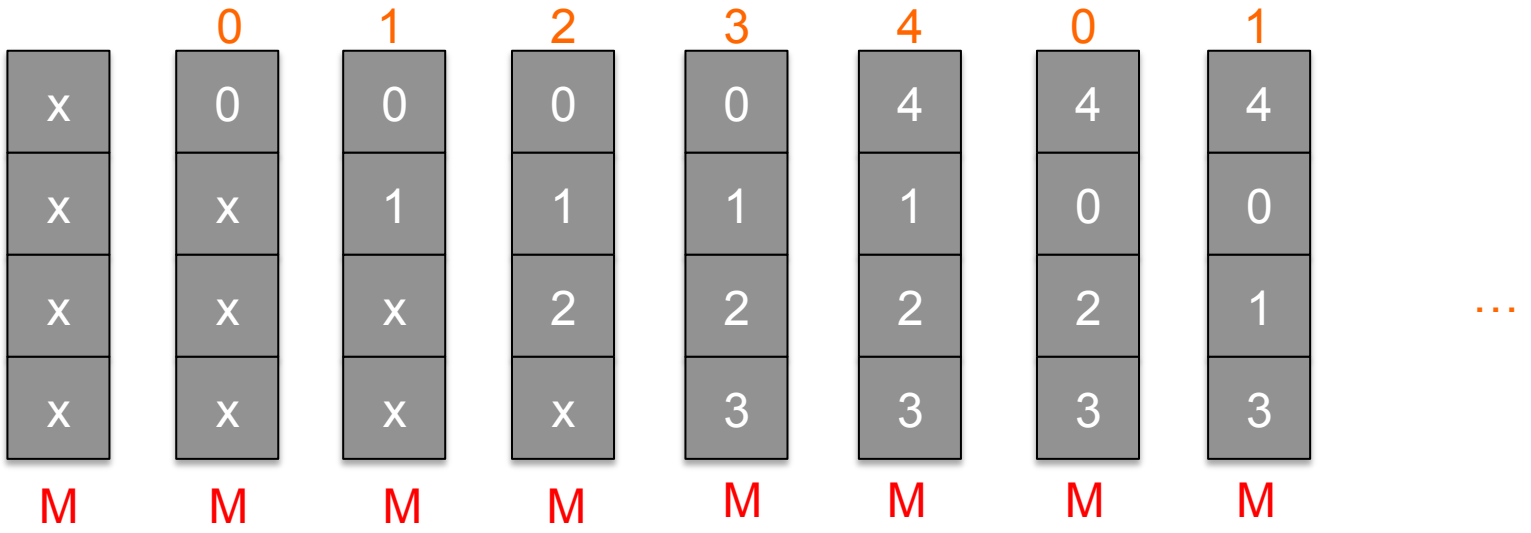
With 8 locations, 8/8 would miss.

As such, Direct mapped caches degrade ‘gracefully’ as they run out of space in a linear access pattern, getting gradually slower (gradually more misses) with each increase in size.

All accesses hit at the Cache size, all miss at twice the Cache size, and there’s a smooth line in between.

Tutorial Question 1:

How about an Associative Cache? Access of four addresses is going to be the same, now access five addresses. We will assume Least Recently Used eviction, this is the most common scenario.



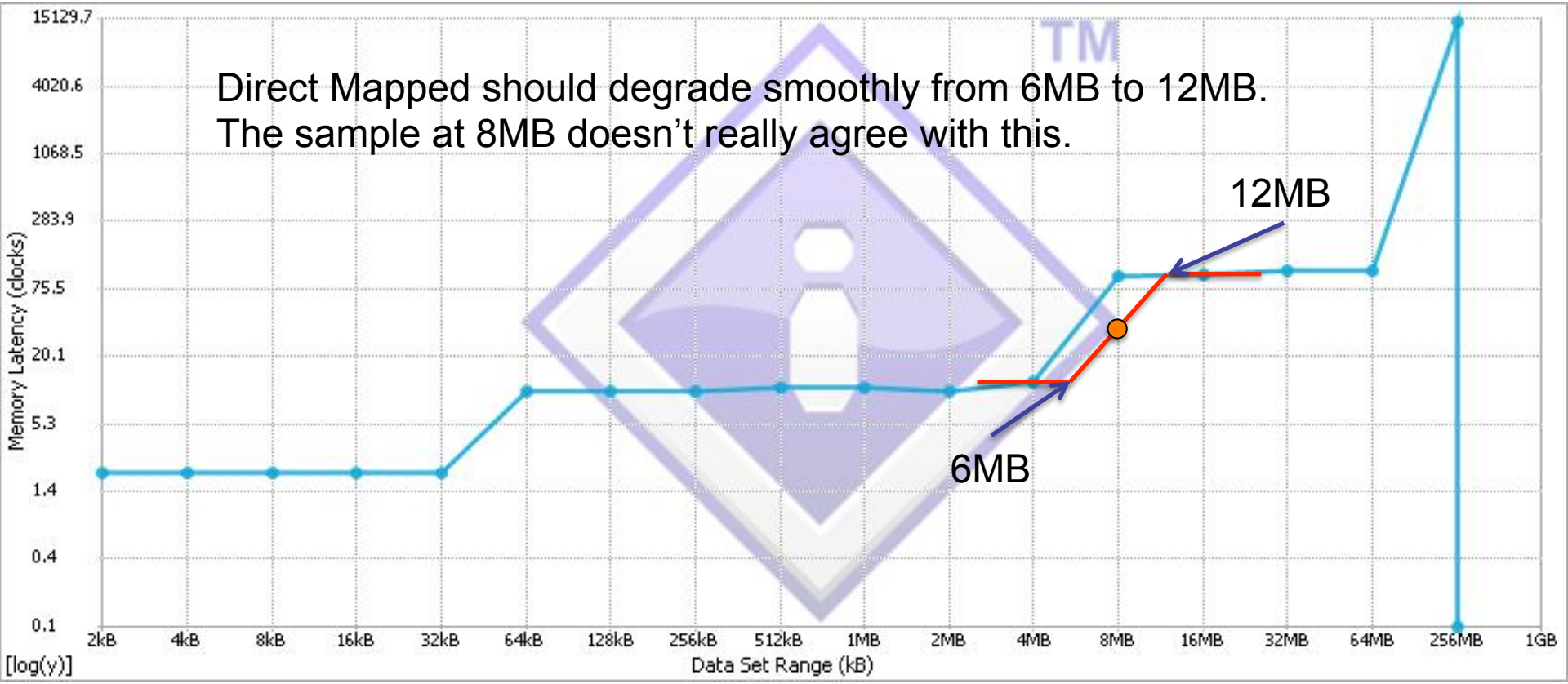
So in the steady state, after the first loop, all accesses miss!

Tutorial Question 1:

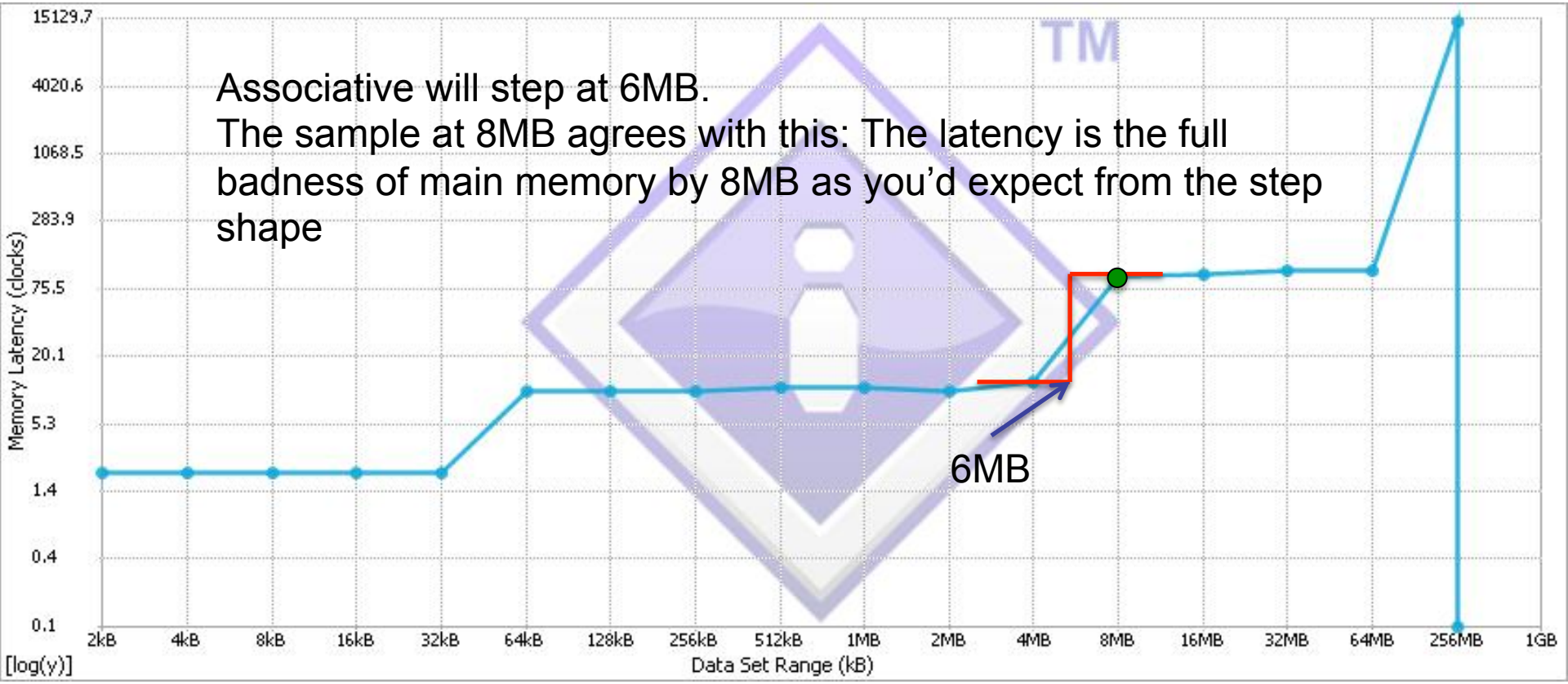
In the associative case, the oldest entry is always evicted. Given the linear access pattern, the oldest entry is, but definition, always the next one to be accessed!

This means that even with only one address more than will fit in the cache, an Associative Cache immediately slows down all the way.

Tutorial Question 1:



Tutorial Question 1:



Tutorial Question 2

Consider the pipelined execution of the following operations:

```
R4 = R3 + R2
R7 = R6 | R5
R8 = R7 - R4
```

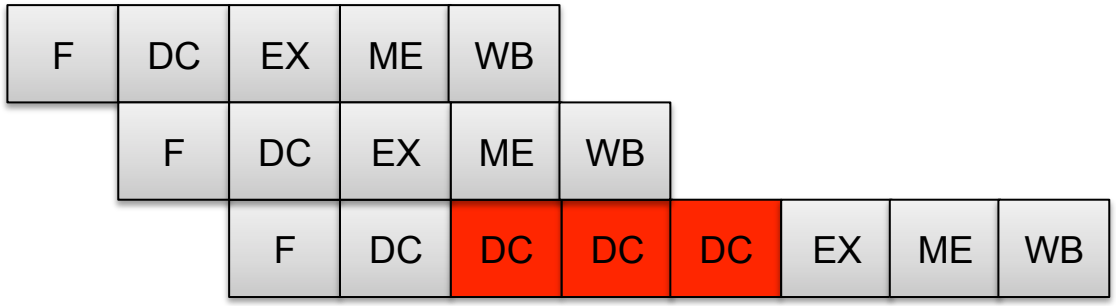
Tutorial Question 2

1. How many clock cycles would it take to complete these operations on a canonical RISC machine with no operand forwarding?
2. How many clock cycles would it take if the EX stage output could be forwarded to its input?
3. How many clock cycles would it take if the EX and MEM stage outputs could be forwarded to the EX input?
4. Now assume that instead of R5, the second operand of the OR has to be fetched from L2 Cache with a 10 cycle latency. Draw a diagram similar to Lecture 3, Slide 18 but with more operations in flight, showing the duration of the stall and how long the execution would take overall now

Tutorial Question 2

1. How many clock cycles would it take to complete these operations on a canonical RISC machine with no operand forwarding?

```
R4 = R3 + R2
R7 = R6 + R5
R8 = R7 - R4
```

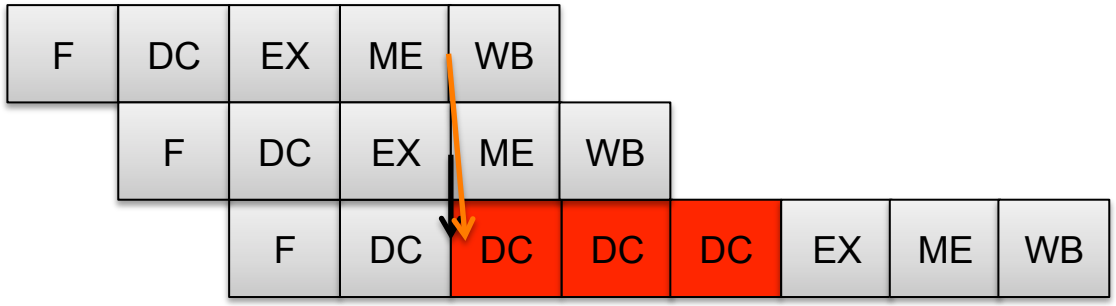


10 cycles with no forwarding

Tutorial Question 2

2. How many clock cycles would it take if the EX stage output could be forwarded to its input?

```
R4 = R3 + R2
R7 = R6 + R5
R8 = R7 - R4
```

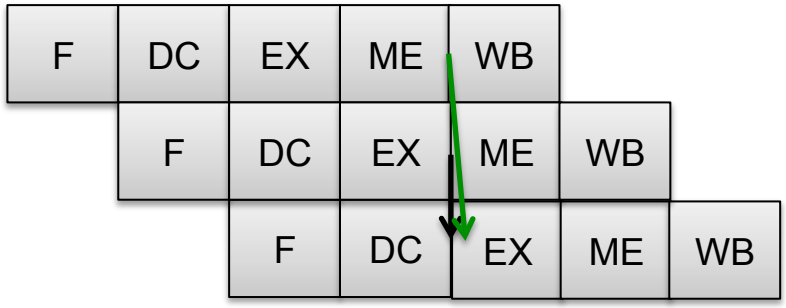


10 cycles with EX forwarding only as the first result has already moved on and can't be forwarded by the time we're ready for it!

Tutorial Question 2

3. How many clock cycles would it take if the EX and MEM stage outputs could be forwarded to the EX input?

```
R4 = R3 + R2
R7 = R6 + R5
R8 = R7 - R4
```



7 cycles with EX and MEM forwarding; everywhere 'downstream' of the execution can be forwarded back at will.

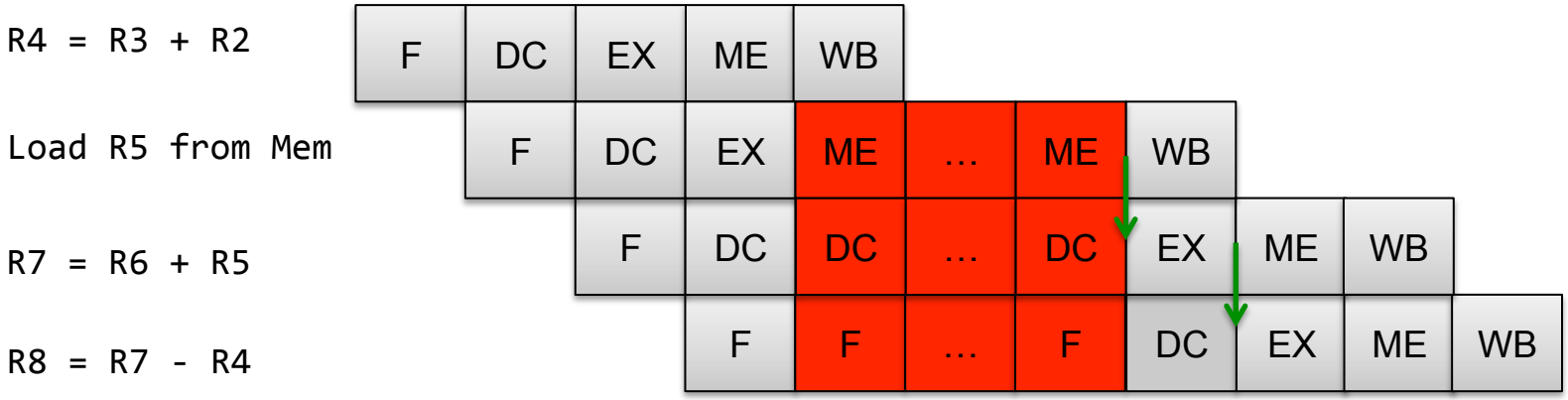
Tutorial Question 2

This seems like a bit of a trick question but raises an important point: Operand Forwarding requires more complexity than was initially presented. Just having the output of the EX stage forwarded back is insufficient, you have to have a forwarding path back from every downstream unit for full safety!

Perhaps not an issue in a 5-stage pipeline, but in a larger pipeline it can lead to significant extra complexity, particularly with respect to timing constraints.

Tutorial Question 2

4. Now assume that instead of R5, the second operand of the OR has to be fetched from L2 Cache with a 10 cycle latency. Draw a diagram similar to Lecture 3, Slide 18 but with more operations in flight, showing the duration of the stall and how long the execution would take overall now



18 cycles with memory stall, EX and MEM forwarding