Problem 1. MC

3. All instructions in a [✓ ***CISC***] / [✓ ***RISC***] / ***multi-core*** architecture (usually) have the same length.

9. SRAM will have higher costs to make but will generally also have higher latency compared to DRAM and the disk. [✓ True] / [✓ False]
lower → faster

7. The output of [✓ ***combinational***] / [✓ ***sequential***] circuits depends exclusively on the current input.

(e) Virtual address to physical address translation can happen simultaneously with the cache access. [✓ True] / [✓ False]

h. Function parameters passed in memory are always stored on the stack. [✓ True] / [✓ False]

(h) Operating system disallows two processes from ever sharing the same physical page at any given time. [✓ True] [✓ False]

7. A write-through cache system will ([✓ always] / [✓ sometimes] / never) write to both cache and memory.
If never in the cach

(i) For a given cache geometry (cache size, block size, associativity), a physically indexed cache is likely to require smaller tags than a virtually indexed cache. [✓ True] [✓ False]

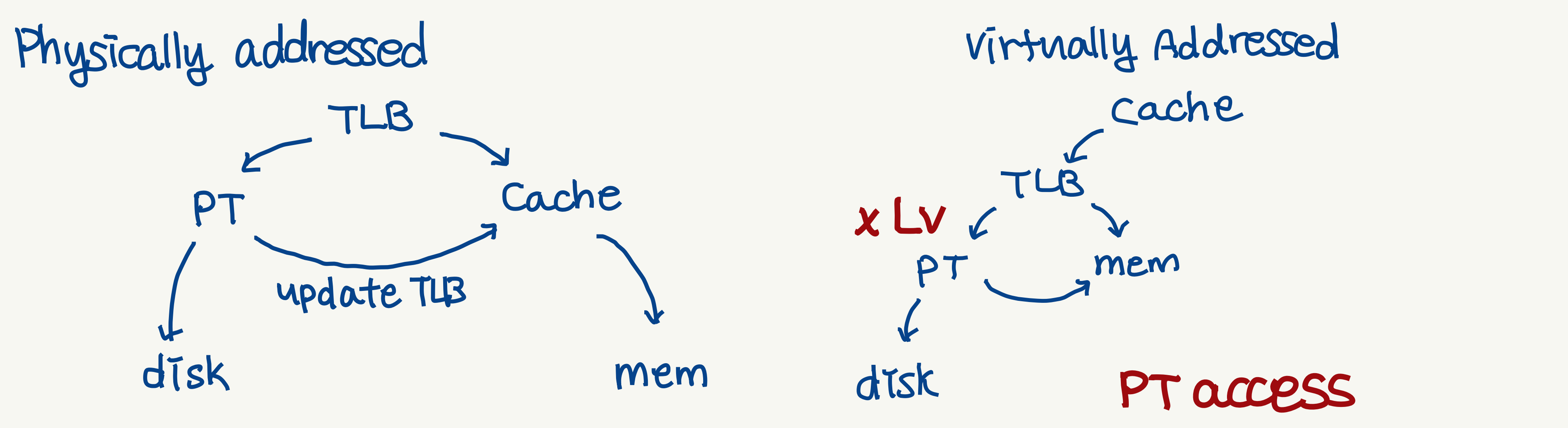
t. For a given cache size and block size, increasing associativity will reduce tag size. ~~True~~ False
→ fully associative

(k) Pipelining improves performance by reducing latency of an instruction. ○ True ○ False

(l) For a given program, memory accesses to its page table tend to exhibit lower spatial locality than memory accesses to its data. [✓ True] [✓ False]

(m) Tags in TLB are derived from physical page numbers. [✓ True] [✓ False]

Problem 2. Virtual Memory



PT access
Careful with the LEVEL

* If TLB miss, PT 끝까지 보내서
Physical address 받아야함.

Problem 3. Branch Prediction

Problem 3: Branch Prediction (F16)

Consider the following C code and corresponding LC2K assembly. Assume it is executed on a pipelined data-path with branch speculation discussed in class.

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
      for (j=0; j != 2; j++) {
            x++;
      }
}
```

```
      lw    0  1  cnt1
      lw    0  2  cnt2
      lw    0  3  one
outer noop
      beq   0  1  exit        //B1
      noop
      lw    0  2  cnt2
inner beq   0  2  done        //B2
      add   4  3  4
      add   2  3  2
      beq   0  0  inner       //B3
done  add   1  3  1
      beq   0  0  outer       //B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

How many times do we not take a branch.
→ executed 31 times
30 not branch, 1 branch
→ executed 90 times
60 not branch, 30 branch
(J=0, J=1) × 30   (J=2) × 30
→ always branched
→ always branched

b) How many branches are predicted correctly if we predict Always-Not-Taken?

B1: 30, B2: 60, B3: 0, B4: 0

Predict backwards taken (T), forwards not taken (N) $7/10 = 0.7$
loop   if, conditions

```
void up_sampler(int *input, int *output){

      int f = 3;                          // upsampling factor
      int count = 0;
      int i;

      for(i = 0; i < 5; ++i){             // B1 LC2K: beq 2 3 endOuterLoop
                                          // NOT taken if i < 5
            int j;

            for(j = 0; j < f; ++j){       // B2 LC2K: beq 2 4 endInnerLoop
                                          // NOT taken if j < f
                  output[count] = input[i];
                  count ++;
            }                             // B3 LC2K: beq 0 0 innerLoop
                                          // Taken if 0 == 0
      }                                   // B4 LC2K: beq 0 0 outerLoop
                                          // Taken if 0 == 0

}
```

6번
3   5x4=20번
15
5

In the C code above, some branch instructions in LC2K are provided. Assume each branch is resolved before the next one.

1. How many branches are taken versus not taken? [6 - 1.5 per column]

Branch	B1	B2	B3	B4
# Taken	1	5	15	5
# Not taken	5	15	0	0

2. How many branches are correctly and incorrectly predicted using local 2-bit saturating counter, initialized to 01 (weakly not taken)? [6 - 1.5 per column]

Branch	B1	B2	B3	B4
# Correctly Predicted	~~6~~ 5	~~20~~ 15	14	4
# Incorrectly Predicted	~~0~~ 1	~~0~~ 5	1	1

NNNNNT   NNNTNNNT

11   10   T

$01 \rightarrow 00$ NT

Problem 4. Multi-Level PT

3. Now consider the following changes to the system: page size is changed to 2KB, while physical and virtual memory address size and the number of entries in the top-level page table are held constant. Any changes to second- or third-level page tables should happen to both.

Answer the following questions. $\overline{17}$ $\overline{x}$ $\overline{y}$ $\overline{11}$ $64-28=36$

a) The number of entries in each second- and third-level page table is now 2^__18__ entries [2]

b) The number of bits necessary to represent a physical page number is now __21__ bits [2] $32-11=21$

c) **In the OLD system,** accessing every element of a 1GB array would require a minimum of __1__ third-level page tables. **In the NEW system**, it would require a minimum of __2__ third-level page tables. [4] 2^{30}

$$\text{Page size} = 2^{13}\text{B}$$

$$\frac{2^{30}}{2^{13}} = 2^{17} \text{ pages needed}$$

$$\text{Page size} = 2^{11}\text{B}$$

$$\frac{2^{30}}{2^{11}} = 2^{19}$$

Say you have an LC2K program with ***1000*** instructions and has the following characteristics:

- 30% of instructions are lws
- 10% are sws
- 15% are adds
- 20% are nors
- 25% are beqs
- 20% of the instructions are immediately followed by an instruction dependent on it.
- 10% of the instructions are followed by an independent instruction and then by a dependent instruction.
- There are no other dependencies.
- You may assume that the above about data hazards is true no matter what instructions are involved.
- 70% of branches are not taken

1) Assume that we use predict-not-taken for branches, what is the runtime for the program? ***(3pts)***

Clock : 50 (ALU takes the most)

1000

+ 1000 × 0.3 × 0.2 (stalls for lw)

+ 1000 × 0.25 × (1−0.7) × 3 (wrong branch prediction)

+ 4 (empty the pipeline)

= 1289

final answer: 64450 ns

∴ 50 × 1289

answer:

3) Let's assume that we know that **80 % of cache accesses** are hits and **99.99 %** of main memory accesses are hits. If cache latency is **5 ns**, main memory access latency is **200 ns**, and disk latency is **10,000 ns**, what is the average access time to memory? Assume that everything we want to access would be in the disk. **(2pts)**

5 + (1−0.8) 200 + (1−99.99) 10,000

final answer: ns

4) Assume that it remains true that **99.99 % of main memory accesses are hits**, and all the latencies stay the same as the previous question. What is the threshold for cache hit rate that implementing a cache would help reduce memory access time? Choose > or <, fill out the blank and show your calculations. **(2pts)**

Hit rate (> / <) ___________%

5 + (1−x) · 200 + (1−99.99) 10,000 < 200 + (1− 99.99) · 10000

Classic performance problem

- Program with following instruction breakdown:

lw	10%	→ 1 cycle of stall
sw	15%	
beq	25%	→ 3 cycles of stall
R-type	50%	(add, nor)

- Speculate "always not-taken" and squash. 80% of branches not-taken (20% taken)
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time if cycle time is 100MHz?

our baseline of CPI (cycles per instruction) = $1 + \underset{\text{(control hazard)}}{3 \cdot 0.25 \cdot 0.2} + \underset{\text{(data hazard)}}{1 \cdot 0.1 \cdot 0.2} = 1.17$

nano = 10^{-9}

$\frac{100 \times 10^6 \text{ cycles}}{\text{second}} = \frac{0.1 \times 10^9}{\text{second}}$

$\frac{1}{0.1 \times 10^9} = 10ns$

∴ 10ns × 1.17 = 11.7ns

Problem 4: CPI of Pipelines (6 points)

Say you have an LC2K program with the following characteristics:

- 35% lw
- 20% sw
- 15% add/nor
- 30% beq
- 25% of instructions are immediately followed by an instruction dependent on it. You may assume that it is true no matter what instructions are involved.
- 70% of branches are not taken

a) What would be the expected CPI of your program using **detect-and-stall** to resolve data hazards and **detect-and-stall** for control hazards? Assume we are using the 5-stage pipeline described in class and the stalling is handled by the pipeline. Clearly show your work. **[3]**

$1 + (\underset{\text{add, lw}}{0.35 + 0.15}) \times \underset{\text{dependent}}{0.25} \times \textcircled{2} + 0.3 \times \textcircled{3} = 2.15$

b) What would be the expected CPI of your program using **detect-and-forward** to resolve data hazards and **predict-not-taken** for control hazards? Assume we are using the 5-stage pipeline described in class. Clearly show your work. **[3]**

$1 + \underset{\text{lw}}{0.35} \times \underset{\text{dependent.}}{0.25} \times 1 + 0.3 \times \underset{\text{miss predict.}}{0.3} \times 3$

Question 4: Pipeline Performance (6 points) V1

Consider a normal 5-stage LC2K pipeline as discussed in class with the following features:

- Using **detect-and-forward** to handle data hazards.
- Using **speculate-and-squash** to handle control hazards and always predict "Not Taken".
- Branches are resolved in the MEM stage.
- Data memory access (and the critical timing path in the MEM stage) is 10 ns, while the critical path in every other stage is 6 ns 5.5

Assume a benchmark with the following characteristics will be run on this pipeline:

- add/nor: 50%
- beq: 15%
- lw: 30%
- sw: 5%
- 40% of all branches are Taken
- 20% of lw instructions are immediately followed by a dependent instruction.
- 10% of lw instructions (disjoint from the previous percentage) are followed by a dependent instruction, but have a single non-dependent instruction between the lw and the dependent instruction.

What is the CPI of this pipeline when running this benchmark?

1 + 0.15 · 0.6 (3) + 0.3 · 0. 2(1) = 1.33 CP

Now, the MEM stage is split into two stages. It reduces the cycle time by splitting the data memory access latency equally between the MEM stages. Branches are resolved in the first MEM stage. second 면 4

What is the new CPI? 1 + 0.15 · 0.6 (3) + 0.3 · 0. 2 (2) + 0.3 · 0.1 (1)

= 1.42

Assuming 10 billion instructions execute, what is the total execution time for both the original (subquestion 1 above) and modified (subquestion 2 above) pipeline? *Show your work for credit.*

1.42 × 10billion × 6ns (slowest)