1. ***I Amdahl-ighted with Tradeoffs (10 points):*** Given the following problems, suggest one solution and give one drawback of the solution. Be brief, but specific.

EXAMPLE
Problem: long memory latencies
Solution: *Caches*
Drawback: *when the cache misses, the latency becomes worse due to the cache access latency*
We would not accept solutions like*: "do not use memory", "use a slower CPU", "cache is hard to spell", etc*

Problem: too many capacity misses in the data cache
Solution: Increase Cache size

drawback: Cache access is slower

Problem: too many control hazards
Solution: Use more bits in branch predictor

drawback: complicated to understand/implement

Problem: our carry lookahead adder is too slow
Solution: Use hierarchical CLA

drawback: Hardware is more complex

Problem: we want to be able to use a larger immediate field in the MIPS ISA
Solution: Take bits from register fields

drawback: higher chance of register spilling

Problem: the execution time of our CPU with a single-cycle datapath is too high
Solution: Use pipelined datapath

drawback: Difficult to implement in hardware

2. ***Hazard a Guess? (10 points):*** Assume you are using the 5-stage pipelined MIPS processor, with a three-cycle branch penalty. Further assume that we always use predict not taken. Consider the following instruction sequence, where the bne is taken once, and then not taken once (so 7 instructions will be executed total):

```
Loop :        lw $t0, 512($t0)
              lw $t1, 64($t0)
              bne $s0, $t1, Loop
              sw $s1, 128($t0)
```

Assuming that the pipeline is empty before the first instruction:

a. Suppose we do not have any data forwarding hardware – we stall on data hazards. The register file is still written in the first half of a cycle and read in the second half of a cycle, so there is no hazard from WB to ID. Calculate the number of cycles that this sequence of instructions would take:

22

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LW	IF	ID	EX	M	WB																	
LW		IF	ID	ID	ID	EX	M	WB														
BNE			IF	IF	IF	ID	ID	ID	EX	M	WB											
SW						IF	IF	IF	ID	EX	M	WB										
???									IF	ID	EX	M	WB									
???										IF	ID	EX	M	WB								
LW											IF	ID	EX	M	WB							
LW												IF	ID	ID	ID	EX	M	WB				
BNE													IF	IF	IF	ID	ID	ID	EX	M	WB	
SW																IF	IF	IF	ID	EX	M	WB

b. How many cycles would this sequence of instructions take with data forwarding hardware: 18

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LW	IF	ID	EX	M	WB													
LW		IF	ID	ID	EX	M	WB											
BNE			IF	IF	ID	ID	EX	M	WB									
SW					IF	IF	ID	EX	M	WB								
???							IF	ID	EX	M	WB							
???								IF	ID	EX	M	WB						
LW									IF	ID	EX	M	WB					
LW										IF	ID	ID	EX	M	WB			
BNE											IF	IF	ID	ID	EX	M	WB	
SW													IF	IF	ID	EX	M	WB

n-way set associative
…ect mapped : $n = 1$
fully assoc

n : n-way set associative
direct mapped : $n = 1$
fully associative : n = # of blocks

3. ***More $ More Problems (10 points):*** Find the data cache hit or miss stats for a given set of addresses. The data cache is a 1KB, direct mapped cache with 64-byte blocks. Find the hit/miss behavior of the cache for a given byte address stream, and label misses as compulsory, capacity, or conflict misses. All blocks in the cache are initially invalid.

$\# \text{ of Indices} = \frac{\text{Total Cache Size}}{\text{Block Size} \times N}$

$\# \text{ of Indices} = \frac{1KB}{64B \times 1} = \frac{2^{10}}{2^6}$

# of indices = 4 bits

offset : 64 B block : 2^6
6 bits

Address in Binary	Cache Hit or Miss	Cache Miss Type
...001|1011|010000	Miss	Compulsory
...000|1011|100000	Miss	Compulsory
...000|0011|010000	Miss	Compulsory
...001|1011|100000	Miss	Conflict
...001|1011|010000	Hit	
...000|1011|100000	Miss	Confict
...000|0011|010000	Hit	
...001|1011|100000	Miss	Conflict

tag | index | offset

4. ***The Trouble with TLBs (10 points)***: Consider an architecture with 32-bit virtual addresses and 1 GB of physical memory. Pages are 32KB and we have a TLB with 64 sets that is 8-way set associative. The data and instruction caches are 8KB with 16B block sizes and are direct mapped – and they are both virtually indexed and physically tagged. Assume that every page mapping (in the TLB or page table) requires 1 extra bit for storing protection information. Answer the following: $\frac{1GB}{32KB}$ $\frac{2^{30}}{2^{15}} = 2^{15}$

a. How many pages of virtual memory can fit in physical memory at a time? 2^{15} pages

Page offset $\log_2(32 \cdot 2^{10})$ 15 | PPN 30−15 15 | PTE 15+1 16 | # of page entries 32 − 15 17 | $2^{17} \times 16b = 2^{17}B \times 2B = 2^{18}$

b. How large (in bytes) is the page table? 2^{18} B [wk 9 monday Live]

$\frac{\#\text{entries in PT}}{\#\text{ of pages in PT}} = \frac{64 \times 8}{2^{17}} \quad \frac{2^9}{2^{17}} = \frac{1}{2^8} = \frac{1}{256}$

c. What fraction of the total number of page translations can fit in the TLB? $\frac{1}{256}$

TLB has 64 sets - 6 bits of VA are needed for index (2^6)
15 right most bits are page offset

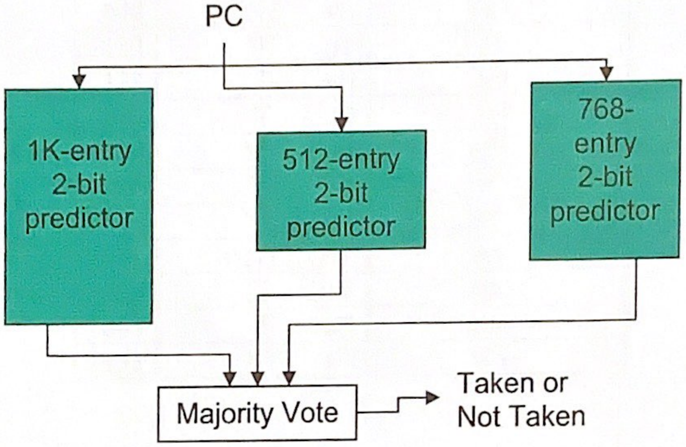
32 − 21 | 20 − 15 | 14 − 0
Tag | Index | offset

d. What bits of a virtual address will be used for the index to the TLB? Specify this as a range of bits – i.e. bits 4 to 28 will be used as the index. The least significant bit is labeled 0 and the most significant bit is labeled 31. 20 − 15

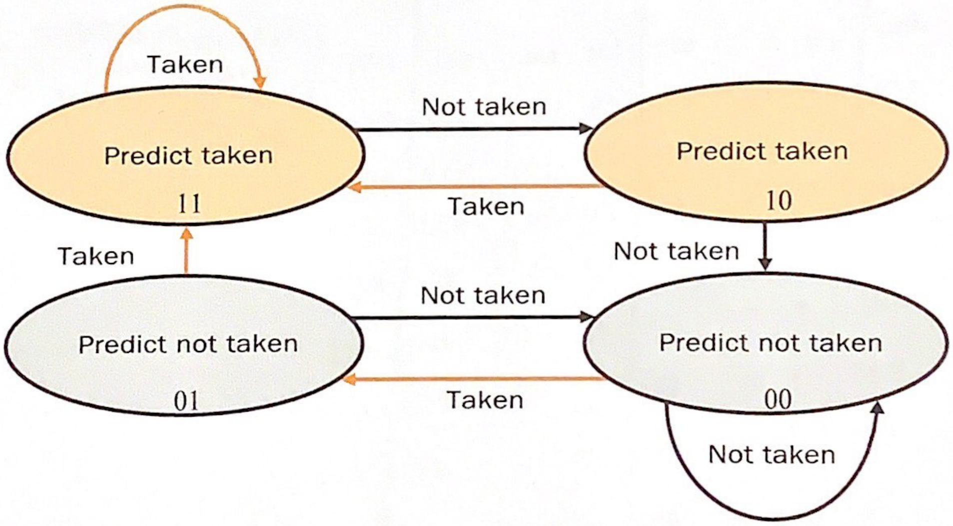
32-bit VA
1GB physical memory (2^{30})
↳ 30 bit physical memory
32KB Page (virtual)

TLB : 64 sets, 8-way set associative
Data Cache : 8KB, 16B block - direct mapped (VIPT)
Instruction Cache : 8KB, 16B block - direct mapped (VIPT)

6. ***A Branch Too Far (10 points)***: One difficulty in designing a branch predictor is trying to avoid cases where two PCs with very different branch behavior index to the same entry of a 2-bit branch predictor. This is called destructive aliasing. One way around this is to use multiple 2-bit branch predictors with different sizes. This way, if two PCs index to the same entry in one predictor, they will not likely index to the same entry in the other predictor. We will evaluate a scheme with three 2-bit branch predictors – each with a different number of entries. The three predictors will be accessed in parallel, and the majority decision of the predictors will be chosen. So if two predictors say *taken* and the other predictor says *not taken*, the majority decision will be *taken*. The scheme looks like this:

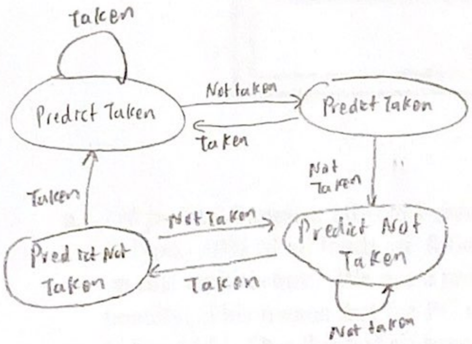


Each predictor has the following FSM:



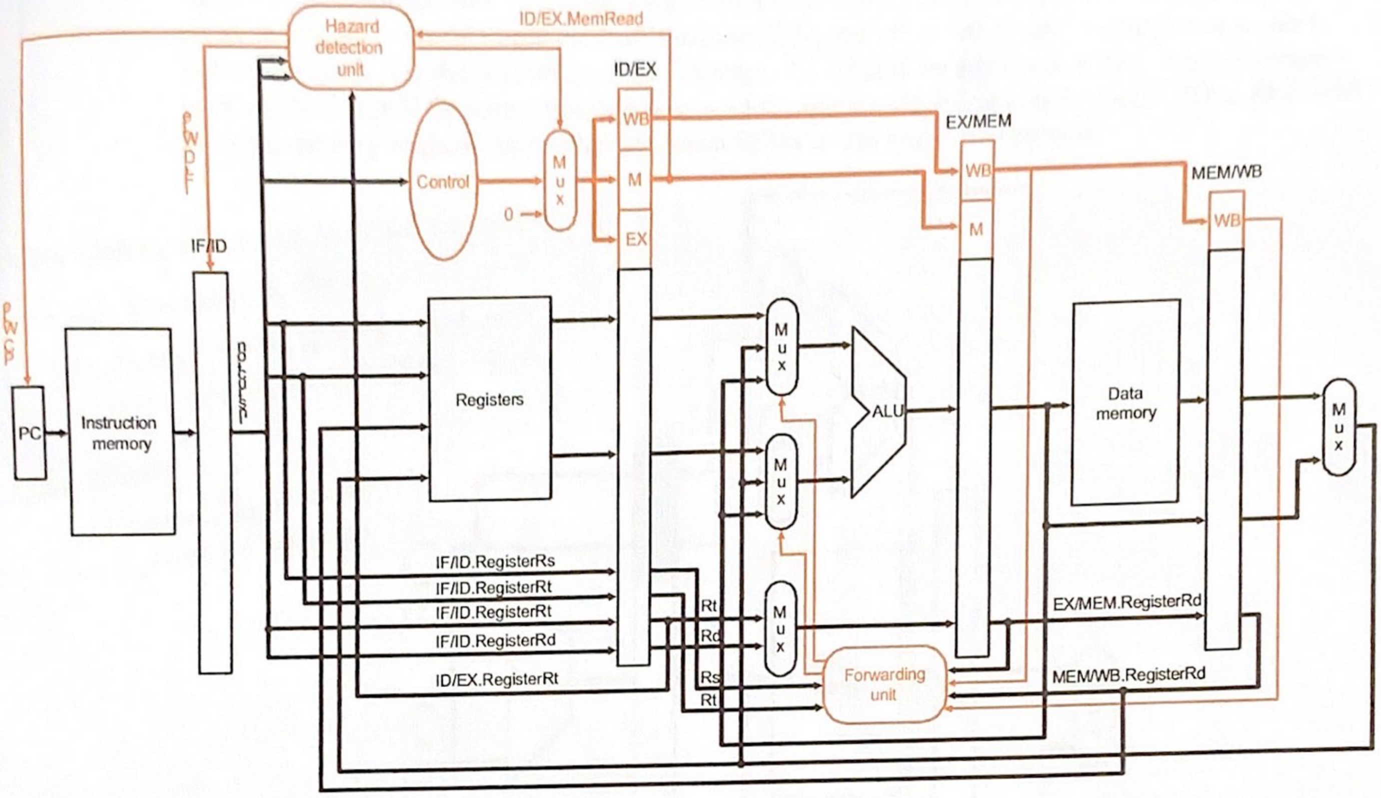
Evaluate the performance of this prediction scheme on the following sequence of PCs. The table shows the address of the branch and the actual direction of the branch (taken or not taken). You get to fill in whether or not the branch predictor would guess correctly or not. Each node of the FSM is marked with the 2-bit value representing that state. Assume that all predictors are initialized to 00. To find an index into a predictor, assume we use the simplified branch indexing formula: index = PC % predictor_size. The symbol % represents the modulo operator. Predictor_size will be different according to the predictor.

PC	Actual Direction	Correctly Predicted?
128	T NT	NO
640	NT NT	YES
1152	NT NT	YES
128	T NT	NO
640	T NT	NO
1152	NT NT	YES
128	T T	YES
640	NT NT	YES
1152	NT NT	YES
128	T NT	NO
640	T NT	NO
1152	NT NT	YES



PC	1024-Entry			512-Entry			768-Entry		
128	128	01	NT	128	01	NT	128	01	NT
640	640	00	NT	128	00	NT	640	00	NT
1152	128	00	NT	128	00	NT	384	00	NT
128	128	01	NT	128	01	NT	128	11	NT
640	640	01	NT	128	11	NT	640	01	NT
1152	128	00	NT	128	10	T	384	00	NT
128	128	01	NT	128	11	T	128	11	T
640	640	00	NT	128	10	T	640	00	NT
1152	128	00	NT	128	00	T	384	00	NT
128	128	01	NT	128	01	NT	128	11	T
640	640	01	NT	128	11	NT	640		NT
1152	128		NT	128	10	T	384		NT

7. ***With Friends Like These... (30 points)***: Consider the scalar pipeline we have explored in class:



a. *(10 points)* Suppose 10% of instructions are stores, 15% are branches, 25% are loads, and the rest are R-type. 30% of all loads are followed by a dependent instruction. We have full forwarding hardware on this architecture. We use a predict not taken branch prediction policy and there is a 2 cycle branch penalty. This means that the PC is updated at the end of the EX stage – after the comparison is made in the ALU. One third of all branches are taken. There is an instruction cache with a single cycle latency and a miss rate of 10% and a data cache with a single cycle latency and a miss rate of 20%. We have an L2 cache that misses 5% – it has a 10 cycle latency – and memory has a 100 cycle latency. Find the TCPI for this architecture.

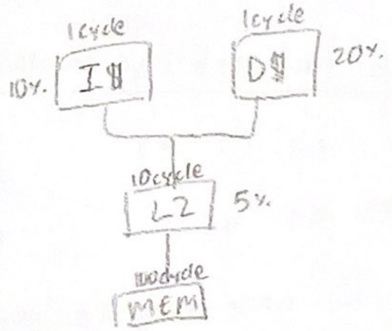
TCPI = 3.725

10% store

25% load → 30% load-use hazard

15% branch → 33% are taken, 2 cycle penalty

50% r-type



MCPI

I$ (1.0)(0.10)(10 + (0.05×100)) = 1.5

D$ (0.35)(0.20)(10 + (0.05×100)) = 1.05

BCPI

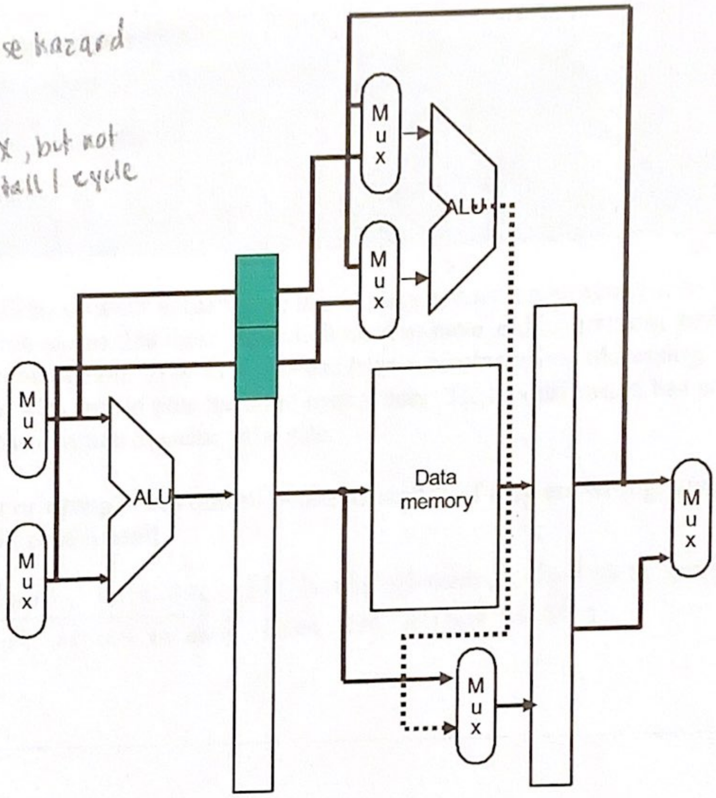
1.0 + (0.25)(0.30)(1) + (0.15)(0.33)(2) = 1.175

Load-Use Hazard Branch Hazard

b. *(5 points)* Your friend has a flash of brilliance – "I know a way to get rid of stalls in this pipeline. The reason we have to stall now is because a load can have a dependent instruction follow it through the pipeline, and we cannot forward the load's data until the end of the MEM stage – but the dependent instruction needs it at the beginning of the EX stage. So what if we add another ALU that recomputes what we did in EX if the instruction before it is a load and it is dependent on the load?" This ALU will be in the memory stage of the pipeline as shown below in this simplified picture:

We are stalling because of load-use hazard
- load finishes in MEM
- dependent needs it in EX, but not ready yet so have to stall 1 cycle

Proposed Solution
- add another ALU that recomputes the EX stage



Is your friend right or wrong? If they are wrong, give an example of when we would still need to stall.

They are right: ____________________

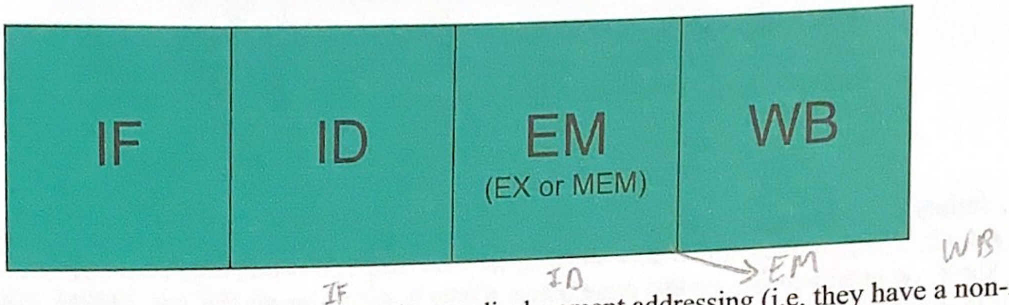
Or

Counter example: Wrong, lw followed by a dependent lw would stall

	1	2	3	4	5	6	7
? lw $t1, 0($t2)	IF	ID	EX	M	WB		
lw $t2, 0($t1)		IF	ID	EX1	EX2	M	WB

Basically: you don't want to extend EX stage since we need the result in MEM stage

c. *(5 points)* Another friend offers an alternative – using the original pipeline from part a, let's get rid of base + displacement addressing for loads and stores. Loads and stores can only use register addressing now. This will allow us to combine EX and MEM into one stage (called EM) and avoid the need to stall entirely. Instructions will either use the ALU or memory – but not both. There is still forwarding hardware, but now we only need to forward from the EM/WB latch to the EM stage ALU. The pipeline will now be:



Suppose that four fifths of loads actually use base + displacement addressing (i.e. they have a non-zero displacement), which means that these loads will need to have add instructions before them to do their effective address computation. Half of stores use base + displacement addressing, and these will also need to be replaced with an add plus the store instruction. This modification has no impact on the branch penalty or the instruction cache miss rate.

Is your friend right or wrong – will this eliminate all stalls? If they are wrong, give an example of when we would still need to stall.

They are right: Right, this will effectively eliminate load-use hazards as we can forward from EM without stalling

Or

Counter example: ______________________________

d. *(10 points)* A third friend has a different idea (it may be time for you to get new friends who don't talk about architecture all the time). Forget about trying to eliminate hazards – she says we should just use superpipelining and get a win on cycle time. Take the original architecture from part a – ignore the suggestions from b and c – and assume that the stages have the following latencies:

Stage	Latency (in picoseconds)
IF	200
ID	100
EX	200
MEM	200
WB	100

Your friend suggests a way to cut the IF, EX, and MEM stages in half – just increase the pipeline depth and make each of these stages into two stages. So your pipeline would now have IF1, IF2, ID, EX1, EX2, MEM1, MEM2, and WB stages – each of which would have 100 picosecond latency. Your friend also finds a way to do full forwarding between stages – even in the ALU – but loads are still a problem. In fact, load stalls will increase now because of this increase in pipeline depth. To help you figure out the new # of pipeline stalls from load data hazards, use the following table:

IF ID EX M WB
IF ID EX M WB

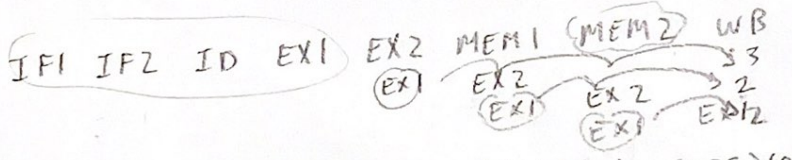
% of Loads	Distance of the next dependent instruction
30%	1 cycle
20%	Exactly 2 cycles later
20%	Exactly 3 cycles later
10%	Exactly 4 cycles later
10%	Exactly 5 cycles later
5%	Exactly 6 cycles later
5%	Exactly 7 or more cycles later

60%

So this means that 30% of loads are immediately followed by a dependent (i.e. 1 cycle later), 20% of loads have a dependent exactly 2 cycles later, 20% have a dependent 3 cycles later, and so on. These classifications are completely disjoint – the 20% of loads that have a dependent 2 cycles later do NOT have dependents 1 cycle later.

Find the TCPI of this new architecture: 6.675

Branch penalty : 4cycles
IF1, IF2, ID, EX1



1 cycle later → 3 stalls
2 cycle later → 2 stalls
3 cycle later → 1 stall

$$BCPI = 1.0 + \underbrace{(0.25)(0.3)(3) + (0.25)(0.2)(2) + (0.25)(0.2)(1)}_{\text{Load-Use hazard}} + (0.15)(0.33)(4) = 1.575$$

MCPI

L2 has "10 cycle latency". Now that our cycles are cut in half, in this problem, L2 has a 20 cycle latency. Same thing applies for MEM access latency.

Note : "10 cycle latency" doesn't mean it always takes 10 cycles regardless of cycle time. It means the latency is the equivalent to the time it takes for 10 cycles to finish at the current cycle speed.

$I\$ = (1.0)(0.10)(20 + (0.05 \times 200)) = 3.0$

$D\$ = (0.35)(0.20)(20 + (0.05 \times 200)) = 2.10$

$MCPI = 3 + 2.1 = 5.1$

$TCPI = 5.1 + 1.575$
$= 6.675$

Assume your target application will run 1M instructions. Find the execution time of this architecture for that application:

ET: 6.675×10^{-6} s

$$ET = IC \times CPI \times CT$$

$$ET = 1000000 \times 6.675 \times \underset{\downarrow \atop \text{100 picoseconds}}{(1 \times 10^{-12})} = 6.675 \times 10^{-6}\ s$$