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: slower cache access time

Problem: too many control hazards
Solution: Loop unrolling with compiler

drawback: Larger code size

Problem: our carry lookahead adder is too slow
Solution: Hierarchical CLA

drawback: more hardware

Problem: we want to be able to use a larger immediate field in the MIPS ISA
Solution: Reduce # registers in ISA

drawback: more register spilling

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

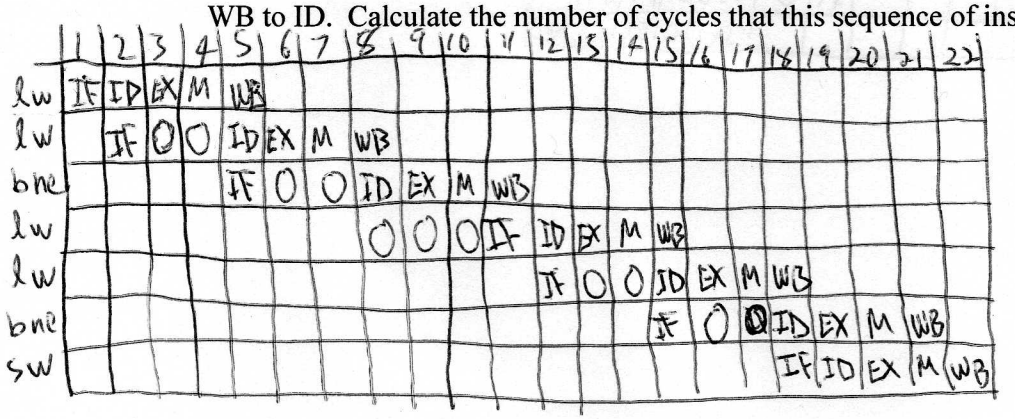
drawback:
more hardware

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:

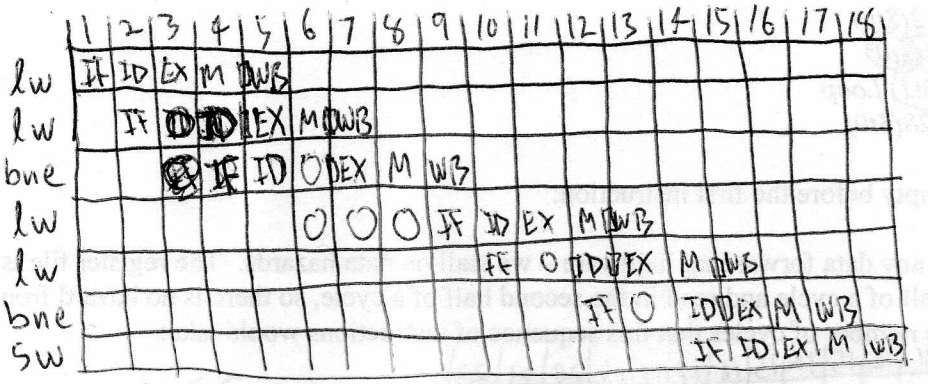
a. Suppose we do not have any data forwarding hardware – we stall on data hazards. The register file is still written in the first half of a cycle and read in the second half of a cycle, so there is no hazard from WB to ID. Calculate the number of cycles that this sequence of instructions would take:



22

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

18



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.

64-byte blk => 6 bit cache blk offset

1KB DM $ => 2^{10}

$\frac{2^{10}}{2^6} = 2^4$ => 4 bit index for 16 entries

Address in Binary	Cache Hit or Miss	Cache Miss Type
...0011011010000	M	Compulsory
...0000101100000	M	Compulsory
...0000011010000	M	Compulsory
...0011011100000	M	Conflict
...0011011010000	H	
...0000101100000	M	Conflict
...0000011010000	H	
...0011011100000	M	Conflict

index | cache blk 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:

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

1GB phy mem => 2^{30} $2^{30}/2^{15} = 2^{15}$

32KB page size => 2^{15}

b. How large (in bytes) is the page table? 2^{18} bytes

entry size = phy pagenum = 15bits, but assume 1 extra bit for protection => 16 bits = 2bytes

#entries = # virtual pages = $2^{32}/2^{15} = 2^{17}$ page table size = $2 * 2^{17} = 2^{18}$

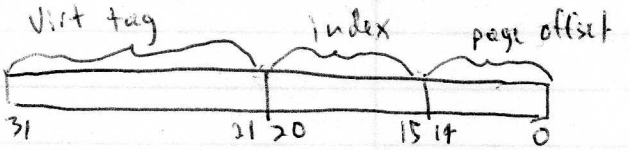
c. What fraction of the total number of page translations can fit in the TLB? 1/256

64 sets => 2^6 $\frac{2^6 \times 2^3}{2^{17}} = \frac{1}{2^8} = \frac{1}{256}$

8 ways => 2^3

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.

bits 15 to 20



64 sets => 6 bit index

5. ***Starting Some Static (Scheduling) (20 points):*** Consider the 2-way superscalar processor we covered in class – a five stage pipeline where we can issue **one ALU or branch instruction** along with **one load or store instruction** every cycle. Suppose that the **branch delay penalty is two cycles** and that we handle control hazards with branch delay slots (since the penalty is two cycles, and this is a 2-way superscalar processor, that would be four instructions that we need to place in delay slots). This processor has **full forwarding** hardware. This processor is a **VLIW** machine. How long would the following code take to execute on this processor assuming the loop is executed 200 times? Assume the pipeline is initially empty and give the time taken up until the completed execution of the instruction sequence shown here. First you will need to schedule (i.e. reorder) the code (use the table below) to reduce the total number of cycles required (but don't unroll it…yet).

Total # of cycles for 200 iterations: 1200

(Hint – schedule the code first for one iteration, then figure out how long it will take the processor to run 200 iterations of this scheduled code)

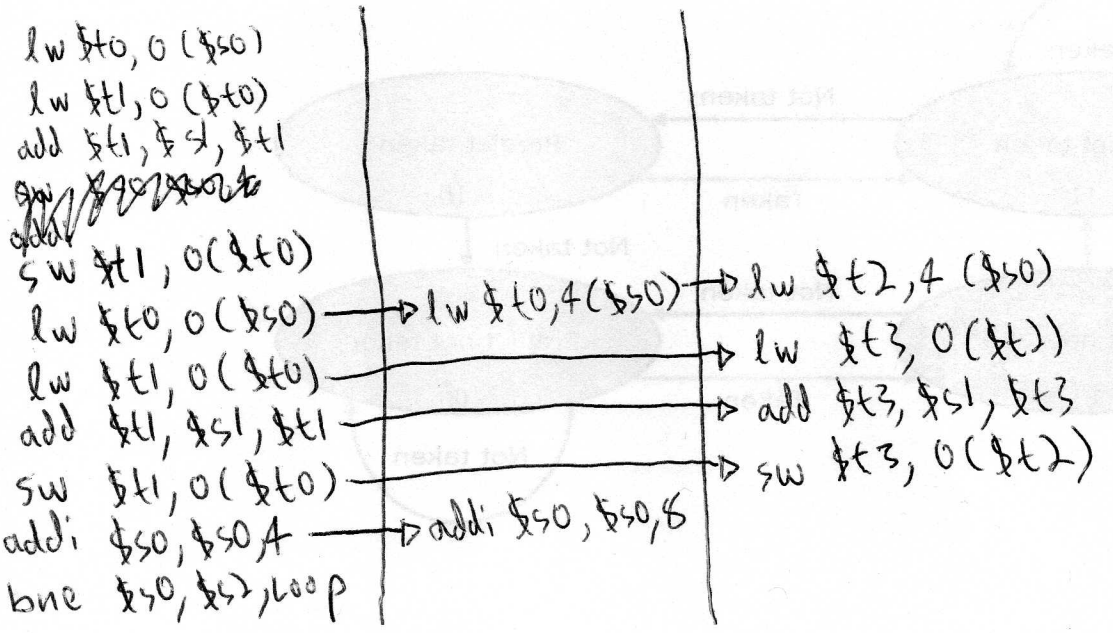
```
Loop:   lw $t0, 0 ($s0)
        lw $t1, 0 ($t0)
        add $t1, $s1, $t1
        sw $t1, 0 ($t0)    # you may assume that this store never goes to the same address as the first load
        addi $s0, $s0, 4
        bne $s0, $s2, Loop
```

Cycle	1st Issue Slot (ALU or Branch)	2nd Issue Slot (LW or SW)
1	addi $s0, $s0, 4	lw $t0, 0($s0)
2		
3		lw $t1, 0 ($t0)
4	bne $s0, $s2, Loop	
5	add $t1, $s1, $t1	NOP
6	NOP	sw $t1, 0($t0)
7		
8		
9		
10		
11		
12		
13		

Now unroll the loop once to make two copies of the loop body. Schedule it again and record the total # of cycles for 200 iterations: 800

Cycle	1st Issue Slot (ALU or Branch)	2nd Issue Slot (LW or SW)
1	addi $s0, $s0, 4	lw $t0, 0 ($s0)
2		lw $t2, 0 ($s0)
3		lw $t1, 0 ($t0)
4		lw $t3, 0 ($t2)
5	add $t1, $s1, $t1	
6	bne $s0, $s2, Loop	sw $t1, 0 ($t0)
7	add $t3, $s1, $t3	NOP
8	NOP	sw $t3, 0 ($t2)
9		
10		
11		
12		
13		

sub 4 since addi moved ahead



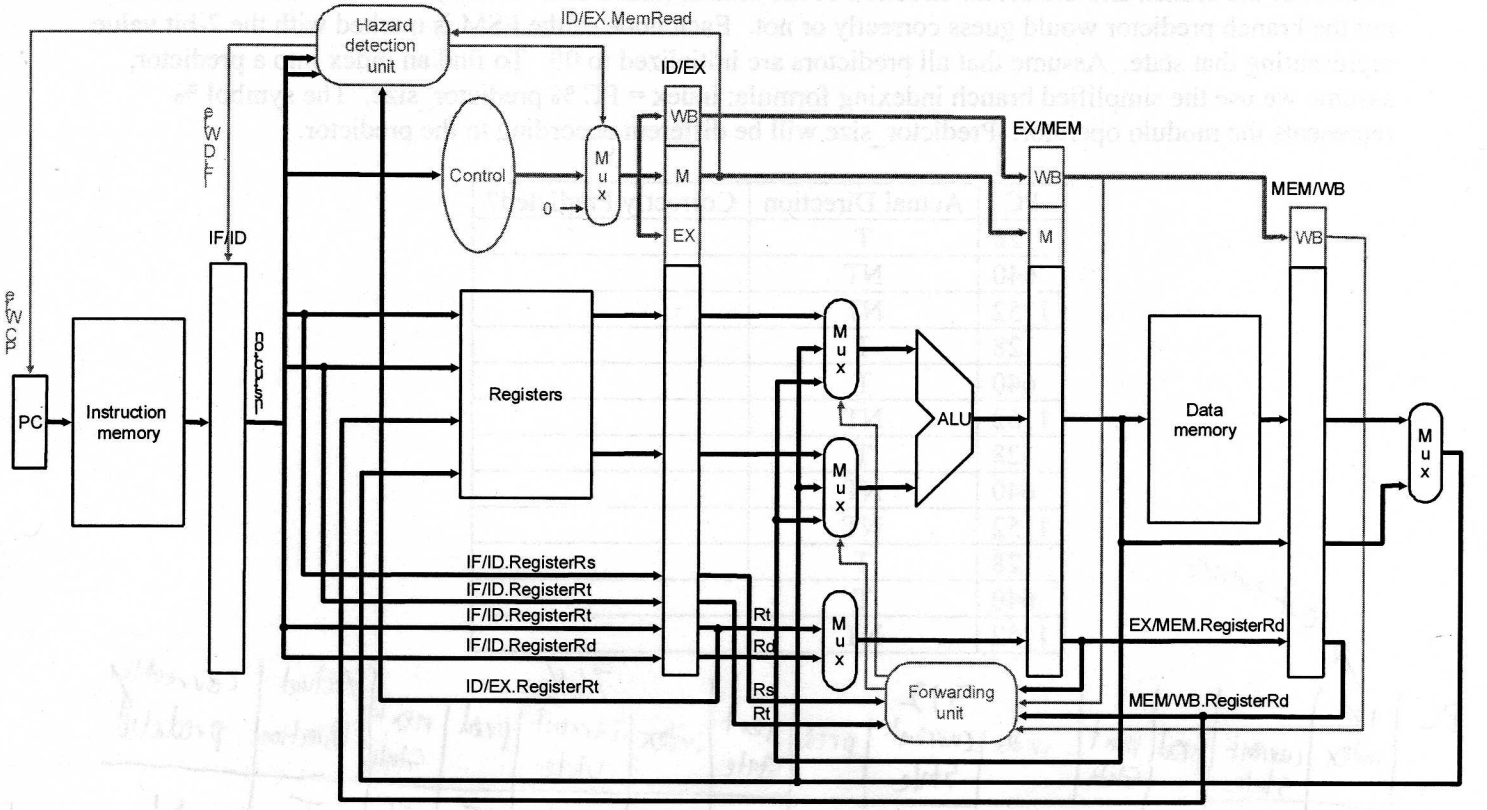
Evaluate the performance of this prediction scheme on the following sequence of PCs. The table shows the address of the branch and the actual direction of the branch (taken or not taken). You get to fill in whether or not the branch predictor would guess correctly or not. Each node of the FSM is marked 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	
640	NT	
1152	NT	
128	T	
640	T	
1152	NT	
128	T	
640	NT	
1152	NT	
128	T	
640	T	
1152	NT	

1024 entries

PC	1K index	current state	pred	next state	512 index	current state	pred	next state	768 index	current state	pred	next state	Actual Direction	correctly predicted?
128	128	00	NT	01	128	00	NT	01	128	00	NT	01	T	N
640	640	00	NT	00	128	01	NT	00	640	00	NT	00	NT	Y
1152	128	01	NT	00	512	00	NT	00	384	00	NT	00	NT	Y
128	128	00	NT	01	128	00	NT	01	128	01	NT	11	T	N
640	640	00	NT	01	128	01	NT	11	640	00	NT	01	T	N
1152	128	01	NT	00	512	00	NT	00	384	00	NT	00	NT	Y
128	128	00	NT	01	128	11	T	11	128	11	T	11	T	Y
640	640	01	NT	00	128	11	T	10	640	01	NT	00	NT	Y
1152	128	01	NT	00	512	00	NT	00	384	00	NT	00	NT	Y
128	128	00	NT	01	128	10	T	11	128	11	T	11	T	Y
640	640	00	NT	01	128	11	T	11	640	00	NT	01	T	N
1152	128	01	NT	00	512	00	NT	00	384	00	NT	00	NT	Y

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

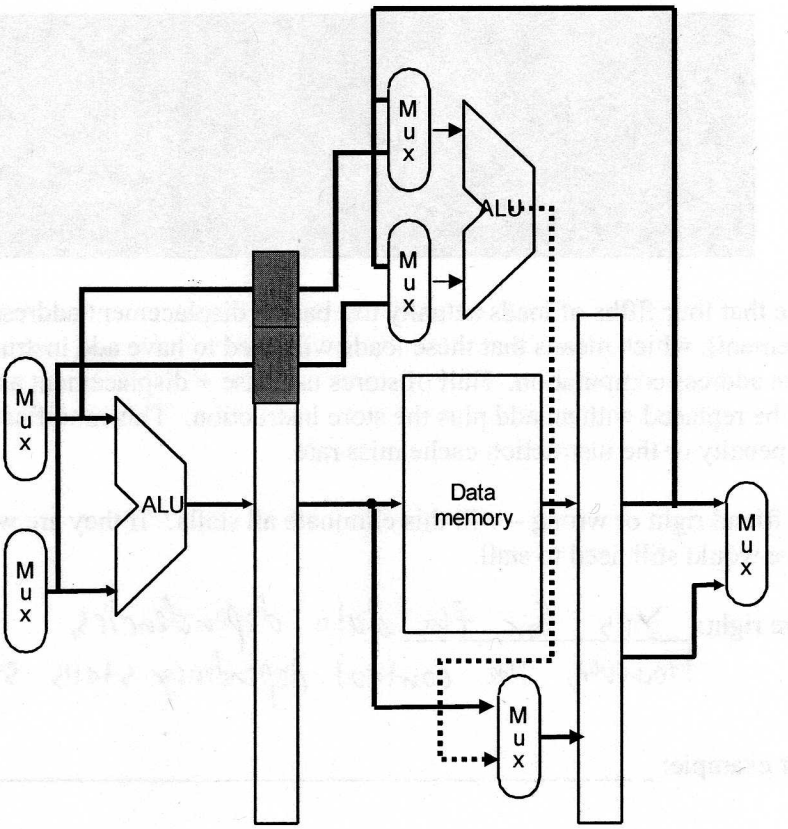
TCPI = BCPI + MCPI = 1.175 + 2.55

BCPI = $1 + \underbrace{0.3 \times 0.25 \times 1}_{\text{lw depend}} + \underbrace{\frac{1}{3} \times 0.15 \times 2}_{\text{br penalty}} = 1.175$

MCPI = $\underbrace{0.1 \times (10 + 0.05 \times 100)}_{\text{inst misses}} + \underbrace{(0.1 + 0.25) \times 0.2 \times (10 + 0.05 \times 100)}_{\text{data misses}}$

$= 2.55$

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:



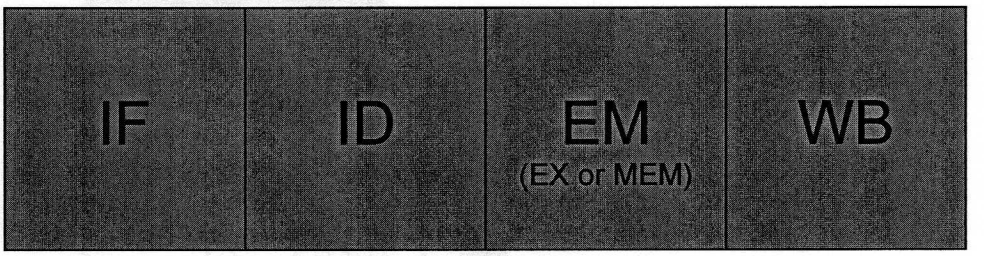
Is your friend right or wrong? If they are wrong, give an example of when we would still need to stall.

They are right: _____

Or

Counter example: lw → lw dependency will still cause bubble. lw needs all 5 stages. Result of EX is used as input address of DM.

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: yes for the data dependencies,
However, the control dependency stalls still exist.

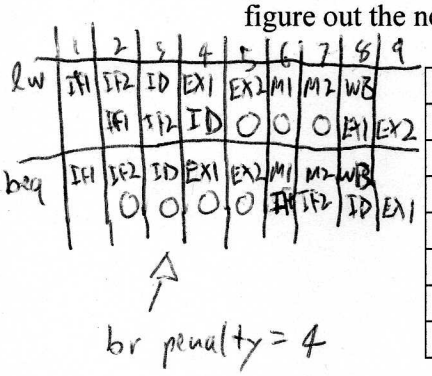
Or

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:



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

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: 4.125

$TCPI_{new} = BCPI_{new} + MCPI_{new} = 1.575 + 2.55$

$BCPI_{new} = 1 + \underbrace{0.25(0.3\times3 + 0.2\times2 + 0.2\times1)}_{lw\ depend} + \underbrace{\frac{1}{3}\times 0.15 \times 4}_{br\ penalty} = 1.575$

$MCPI_{new} = MCPI_{old} = 2.55$

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

ET: 4.125×10^{-4} seconds

CT = 100 pico seconds $= 100 \times 10^{-12}$ seconds

$$\frac{\text{cycles}}{\text{inst}} \times \frac{\text{sec}}{\text{cycle}} \times \text{Inst} = \text{sec}$$

$$ET = TCPI_{new} \times CT \times inst$$

$$= 4.125 \times 100 \times 10^{-12} \times 10^{6} = 4.125 \times 10^{-4} \text{ seconds}$$