HW 11 Solutions

Problem 1: Pipelining Performance (25 points)

A pipelined processor has 5 stages with critical path lengths as follows:

1. Instruction Fetch (IF), critical path length = 200 picoseconds (ps)
2. Instruction Decode/Register Read (ID), critical path length = 250 ps
3. Execute (EX), critical path length = 270 ps
4. Memory (MEM), critical path length = 280 ps
5. Write Back (WB), critical path length = 150 ps

Answer the following questions.

Part A: Cycle Time (5 points)

What is the cycle time of this processor?

Problem 1A Solution

In a pipelined processor, the longest stage sets the cycle time: 280 ps.

Part B: Cycle Count (3 points)

How many cycles will this processor take to execute 1 million independent instructions?

Problem 1B Solution

The first instruction in the sequence will take 5 cycles to finish. The second instruction will finish right behind it in the 6th cycle, the 3rd in the 7th, and so on for the remaining instructions. So the total number of cycles will be 1,000,004 cycles.

Part C: CPI (2 points)

What is the CPI of this processor when executing the 1 million instructions in Part B?

Problem 1C Solution

CPI = cycles per instruction = cycle count / instruction count = 1,000,004 / 1,000,000 = ~1

Part D: Non-Pipelined (Single-Cycle) Implementation (10 points)

Consider a non-pipelined version of this processor, in which these 5 stages are executed sequentially in a single cycle. Repeat Parts A-C for this single-cycle implementation.

Problem 1D Solution

Cycle time: *In a non-pipelined processor, a single instruction executes, from start to finish, in every cycle. So the cycle time is the total time to execute all stages: 200 + 250 + 270 + 280 + 150 ps = 1150 ps = 1.15 ns*

Cycle count: *In this processor, each instruction takes exactly 1 cycle, with no overlapping of execution. So the total number of cycles = the total number of instructions = 1,000,000.*

CPI: *By definition, the CPI of this processor is always exactly 1. You can confirm this by computing cycle count / instruction count = 1,000,000 / 1,000,000 = 1.*

Part E: Performance Comparison (5 points)

Which implementation executes these instructions faster: the pipelined implementation, or the single-cycle implementation? What is its speedup over the slower implementation?

Problem 1E Solution

It's tempting but wrong to directly compare the cycle counts you computed in Part B and Part D -- because the pipelined processor has faster cycles than the non-pipelined, it's not an apples-to-apples comparison.

What you have to do instead is compute the actual time (in seconds or some equivalent unit), using both the cycle count and the cycle time, for each of these implementations. Then you can correctly determine which is faster and take the ratio of execution times to compute the speedup.

For the pipelined processor, these instructions will take 1,000,004 cycles, and each cycle takes 280 ps (280e-12 s), giving you 0.00028 seconds or 0.28 ms as the execution time.

For the non-pipelined processor, these instructions will take 1,000,000 cycles, and each cycle takes 1.15ns (1.15e-9s), giving you 0.00115 seconds or 1.15 ms as the execution time.

From this, we can conclude that the pipelined processor is faster, and that its speedup over the non-pipelined processor is 1.15 ms / 0.28 ms = ~4.11.

Problem 2: Data Hazards and Forwarding (25 points)

Consider the following sequence of RISC-V instructions:

```
ADDI  sp, sp, -4 // Instruction A
LD    t0, 4(sp)  // Instruction B
ADD   a0, t1, t0 // Instruction C
LD    t1, 8(a0)  // Instruction D
LD    t2, 0(a0)  // Instruction E
ADD   a0, t1, t2 // Instruction F
```

In your solutions, please refer to the instructions by their labels A-F.

Part A: Read-after-Write Dependency List (6 points)

List **all** cases where an instruction in this sequence depends on the value written by a previous instruction. For each case, you need to specify the producing instruction, the consuming instruction, and the register involved. For example: “Producer: Instruction X. Consumer: Instruction Y. Register: s0.”

Problem 2A Solution

Each row of the table below corresponds to a read-after-write dependency in this set of instructions.

Producer	**Consumer**	**Register**
A	B	sp
B	C	t0
C	D	a0
C	E	a0
D	F	t1
E	F	t2

Part B: Ideal Performance (5 points)

If all of these instructions were completely independent, how many cycles would they take to execute on a 5-stage pipeline, and what would the CPI of this sequence be?

Problem 2B Solution

This is the same logic as the solution to Problem 1B: Instruction A will take 5 cycles to complete, and the following 5 instructions will complete in the next 5 cycles = 10 cycles. The CPI (cycles per instruction) = number of cycles / number of instructions = 10 / 6 = ~1.67

Part C: Performance with Stalling (6 points)

If the 5-stage pipeline has to stall on dependencies until the consuming instruction can read the value written to the register file by the producing instruction, how many cycles will this sequence of instructions take, and what is its CPI? As part of your answer, you should show a pipeline diagram that clearly illustrates which instruction is in which stage during each cycle.

Problem 2C Solution

As the pipeline diagram below shows, this sequence of instructions takes 18 cycles, so its CPI is 18 cycles / 6 instructions = 3.

Cycle #	IF	ID	EX	MEM	WB	Notes
1	A					
2	B	A				
3	C	B	A			Dependency check in ID: B has to stall here until A writes back to sp
4	C	B	--	A		
5	C	B	--	--	A	This cycle, A writes to sp and B reads the updated value.
6	D	C	B	--	--	Dependency check in ID: C has to stall here until B writes back to t0
7	D	C	--	B	--	
8	D	C	--	--	B	This cycle, B writes to t0 and C reads the updated value.
9	E	D	C	--	--	Dependency check in ID: D has to stall here until C writes back to a0
10	E	D	--	C	--	
11	E	D	--	--	C	This cycle, C writes to t0 and D reads the updated value.
12	F	E	D	--	--	Dependency check in ID: E is not dependent on any instructions currently in the pipeline, so it can move forward
13		F	E	D	--	Dependency check in ID: F is dependent on both D and E, so it must wait until both of them write back
14		F	--	E	D	
15		F	--	--	E	This cycle, F reads the updated values in t0 and t1 written by D and E
16			F	--	--	

17				F	--	
18					F	

Part D: Performance with Forwarding (8 points)

If this processor uses data forwarding whenever possible to bypass the register file and send results directly to where they are needed, how many cycles will this sequence of instructions take, and what is its CPI? As part of your answer, you should show a pipeline diagram that clearly illustrates which instruction is in which stage during each cycle, as well as any forwarding paths that are used.

Problem 2D Solution

As the pipeline diagram below shows, this sequence of instructions takes 12 cycles, so its CPI is 12 cycles / 6 instructions = 2.

Cycle #	**IF**	**ID**	**EX**	**MEM**	**WB**	**Notes**
1	A					
2	B	A				
3	C	B	A			
4	D	C	B	A		A's result is forwarded from EX/MEM registers to the ALU input, where B needs it.
5	D	C	--	B	A	C can't move forward to EX yet, since B hasn't produced its result from data memory yet. C stalls.
6	E	D	C	--	B	B's result is forwarded from the MEM/WB registers to the ALU input where C needs it.
7	F	E	D	C	--	C's result is forwarded from EX/MEM registers to the ALU input where D needs it
8		F	E	D	C	C's result is forwarded from MEM/WB registers to the ALU input where E needs it
9		F	--	E	D	F has to stall, since E hasn't gotten its result from memory yet

10			F	--	E	E's result is forwarded from MEM/WB registers to beginning of ALU
11				F		
12					F	

Problem 3: Control Hazards and Branch Prediction (25 points)

Consider the following sequence of outcomes for a particular branch instruction:

T, NT, T, NT, T, NT, T, NT….

Assume this sequence repeats indefinitely.

Part A: The Easy Part (6 points)

What is the **long-term** (steady-state) accuracy of the predict-taken predictor? What about predict-not-taken?

Problem 3A Solution

Predict-taken: *A predictor that always predicts T will be correct 50% of the time for this sequence.*

Predict-not-taken: *A predictor that always predicts NT will be correct the other 50% of the time.*

Part B: 1-bit predictor (4 points)

What is the long-term (steady-state) accuracy of the 1-bit predictor? If its initial state matters, explain.

Problem 3B Solution

Intuitively, the 1-bit predictor always predicts that the most recent branch outcome will happen again. Since the outcomes in this pattern alternate, we should expect it to therefore always be wrong, with an accuracy of 0%. And here's the table:

Initial Predictor State	**Branch Outcome**	**Correct?**	**Next Predictor State**
unknown	T	unknown	T
T	NT	no	NT
NT	T	no	T

T	NT	no	NT
NT	T	no	T

So in the long term, the accuracy of this predictor will converge to 0%.

Part C: 2-bit predictor (8 points)

What is the long-term (steady-state) accuracy of the 2-bit predictor? If its initial state matters, explain.

Problem 3C Solution

For this predictor, we should look at all 4 possible initial states.

Initial Predictor State	**Branch Outcome**	**Correct?**	**Next Predictor State**
ST	T	Yes	ST
ST	NT	no	WT
WT	T	Yes	ST
ST	NT	no	WT

We are now in a pattern that will repeat indefinitely, so we can conclude that the long-term accuracy is 50% if we start in ST.

Next, let's look at SNT:

Initial Predictor State	**Branch Outcome**	**Correct?**	**Next Predictor State**
SNT	T	no	WNT
WNT	NT	Yes	SNT
SNT	T	no	WNT
WNT	NT	Yes	SNT

This turns out to be exactly analogous to ST, with an accuracy of 50%.

Next, the weak states, starting with WT:

Initial Predictor State	**Branch Outcome**	**Correct?**	**Next Predictor State**
WT	T	Yes	ST
ST	NT	no	WT

WT	T	Yes	ST
ST	NT	no	WT

So this predictor ends up behaving similarly to the strongly taken states, with an accuracy of 50%.

Finally, WNT:

Initial Predictor State	**Branch Outcome**	**Correct?**	**Next Predictor State**
WNT	T	no	WT
WT	NT	no	WNT
WNT	T	no	WT
WT	NT	no	WNT

Finally, a different result. Starting with WNT, this predictor ends up replicating the behavior of the 1-bit predictor: shuttling between only the WT and WNT states, and always predicting the wrong result.

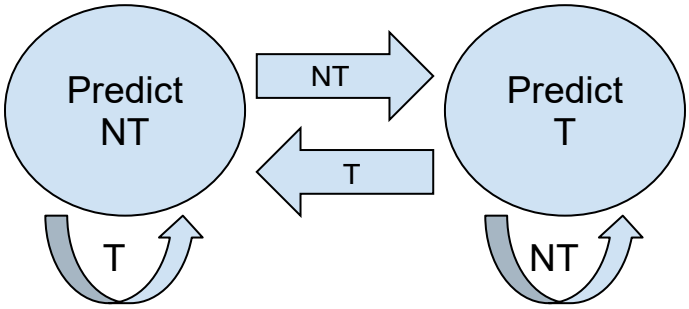
So in summary, the long-term accuracy of the 2-bit predictor is 50% for all initial states except for WNT, which is 0%.

Part D: Custom Predictor (7 points)

Design your own predictor that uses no more than 2 bits of state to maximize prediction accuracy for this pattern. Show the state diagram, and give the long-term accuracy of your predictor. If its initial state matters, explain.

Problem 3D Solution

An optimal predictor for this pattern would be exactly the opposite of the 1-bit predictor: it would always predict the OPPOSITE of the most recent outcome. So we can use the 1-bit predictor as a template but swap what each state predicts (or alternatively, you could swap the labels on all the arrows). The long-term accuracy of this predictor will be 100% (corrected from initial posting), and its initial state won't matter in the long term.



Problem 4: Cache Performance (25 points)

Consider a cache hierarchy with the following characteristics:

- Split L1 data and instruction caches. The instruction cache has a hit rate of 95%, and the data cache has a hit rate of 92%. Both L1 caches have access times of 1 cycle.
- A unified L2 cache with a hit rate of 80% and an access time of 7 cycles.
- A unified L3 cache with a hit rate of 75% and an access time of 12 cycles.
- A main memory with an access time of 150 cycles.

Part A: Reads only (9 points, 3 each)

Consider a program that somehow has no store instructions. 20% of its instructions are loads. For this program:

- What is the average memory access time of an instruction fetch?
- What is the average memory access time of a data read (a load)?
- How much time does the average **instruction** spend accessing memory?

Problem 4A Solution

For deeply nested cache hierarchies, I like to set up my AMAT calculations like this:

*AMAT = Access time of first level + First-level miss rate * (AMAT of subsequent levels)*

and recursively expand the parentheses (don't panic, I'll explain what this means).

The idea is to think of memory accesses like this: "All accesses first go to L1, which takes [L1 access time] cycles. Then, [L1 miss %] of those go to L2. Of the ones that go to L2, they all take an additional [L2 access time] cycles, and [L2 miss %] of those will need to go to L3…"

Here's what this calculation looks like for instruction fetches, which go to the L1 instruction cache first, then L2, L3, and finally main memory if needed:

*AMAT(Instruction Fetch) = 1 + 0.05 * (7 + 0.2 * (12 + 0.25 * 150)) = 1.845 cycles*

The 0.05 here is the L1 instruction miss rate (1-95%). The 0.2 is the L2 miss rate (1-80%), and the 0.25 is the L3 miss rate (1-25%).

For data reads, the only difference is the L1 miss rate, which is 8% for the L1 data cache. So we replace the 0.05 with 0.08 and repeat the calculation:

*AMAT(Data Read) = 1 + 0.08 * (7 + 0.2 * (12 + 0.25 * 150)) = 2.352 cycles*

For the final part of this problem, you need to remember the relationship between instructions and memory accesses. Every instruction accesses memory once: when the PC is sent to the instruction memory. This is the instruction fetch, so every instruction spends at least 1.845 cycles accessing memory. Then, 20% of this particular program's instructions are loads, which means they have to do a 2.352 cycle data memory access as well.

*So on average, an instruction will spend 1.845 + 0.2 * 2.352 = 2.3154 cycles accessing memory.*

Part B: Reads and writes (16 points)

Consider a program for which 20% of its instructions are loads and 10% are stores.
Here are the write policies of the different levels of cache:

- Level 1 data: Write-allocate (with large blocks), fetch-on-write, write-back. 22% of evicted blocks are dirty.
- Level 2: Write-no-allocate and write-through. The write buffer is full 10% of the time.
- Level 3: Write-no-allocate and write-through. The write buffer is full 2% of the time.

For this program:

- How many read requests are made to the L1 instruction cache?
- How many write requests are made to the L1 instruction cache?
- How many read requests are made to the L1 data cache?
- How many write requests are made to the L1 data cache?
- How many read requests are made to the L2 cache?
- How many write requests are made to the L2 cache?
- How many read requests are made to the L3 cache?
- How many write requests are made to the L3 cache? Of these, how many stall?
- How many read requests are made to main memory?
- How many write requests are made to main memory? Of these, how many stall?

Problem 4B Solution

As mentioned in class, I managed not to give you a specific number of instructions for this problem, so any number you pick is fine. The path of least resistance is to pick 10,000, which is the number we used in the class examples, so these solutions will be based on that.

Read requests to L1-I: *Each of the 10,000 instructions has to be fetched, so the answer is just 10,000.*

Write requests to L1-I: *All instruction fetches are reads, not writes -- remember that the instruction memory in your processor diagram is read-only. So the answer is 0.*

Read requests to L1-D: *Read requests to L1-D come from load instructions, and 20% of instructions in this program are loads, so that gives us 20% * 10000 = 2000.*

Write requests to L1-D: *Write requests to L1-D come from store instructions, and 10% of instructions in this program are stores, so that gives us 10% * 10000 = 1000.*

Read requests to L2: *Read requests to L2 come from 3 sources. The first is instruction fetch misses coming from L1-I. Of the 10,000 instruction fetches sent to L1-I, 95% are hits, meaning 5% or 500 are misses. The second is L1-D read misses, which then get sent to L2. Of the 2,000 reads sent to L1-D, 92% are hits, meaning that 8% or 160 are misses. The third, and most surprising, is L1-D write misses. Because the L1 cache is write-allocate and fetch-on-write with large blocks, an L1-D write miss means that you need to read the rest of the bytes in the same block from L2. Of the 1,000 writes sent to L1-D, 8% or 80 are misses. So the total read requests to L2 = 500 instruction fetch misses + 160 read misses + 80 reads to fill up cache blocks on a write miss = 740.*

Write requests to L2: *The L1 cache is write-back, so it only writes to L2 when dirty data is evicted from L1. Evictions only happen a miss, so the relevant information is the number of misses in the L1 data cache and the likelihood that an evicted block is dirty: (160 + 80 data misses) * 22% = 52.8 write misses; let's round to 53.*

Read requests to L3: *L2 will only send read requests to L3 upon a read miss in L2. Because the cache is write-no-allocate, a write miss won't trigger a read to L3. So we just need to calculate the number of L2 read misses. This is the total number of L2 read requests (740) multiplied by the L2 miss rate (1-80% = 20%): 740 * 0.2 = 148.*

Write requests to L3: *L2 is write-through, so every L2 write gets sent to L3: 53. The only ones that stall are the ones issued when the write buffer is full, which is 10% of the time, so you get an average of 5.3 write stalls per 10,000 instructions.*

Read requests to main memory: *L3 will only send read requests to memory if it encounters a read miss, so this is exactly analogous to the L3 read requests we computed a moment ago. We take the total number of L3 read requests (148) multiplied by the L3 miss rate (25%): 37.*

Write requests to main memory: *All 53 writes sent to L3 will go to main memory; of those, 0.02*53 = 1.06 will stall.*

Problem 5: Cache Tracing (25 points)

You have a 2-way set-associative cache with 128 B of data (not counting metadata). Each cache block is 8 B (again, not counting metadata). Main memory addresses are 12 bits, and precise LRU is used as the replacement policy within each set.
Assume that the cache is initially empty (all blocks are invalid), and fill in the table below to trace the following sequence of read accesses.

Part A: Tracing individual read accesses (15 points)

Complete the following table showing how each address is split into tag, index, and offset, as well as whether it is a hit or a miss and which addresses' data, if any, it evicts. These accesses are starting in sequential order, with the first one accessing an empty cache.

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Addresses evicted (if any)**
1010 1111 0011					
1011 1111 0010					
1010 1111 0010					
1011 1110 0010					
1010 1111 0011					

1011 1111 0011					
1010 1110 0010					
1011 1111 0010					
1011 1110 0011					

Problem 5A Solution

The key to this problem is knowing how to break up the address into tag, index, and offset.

The offset is based on the size of a block. Since this cache has 8B blocks, the offset is log_2(8) = 3 bits.

The index is based on the number of sets, which we have to calculate. We can start by calculating the number of blocks, which is the total size of the cache divided by the size of an individual block = 128B / 8B = 16 blocks. In a 2-way set-associative cache, each set has 2 blocks, so the number of sets is half the number of blocks = 8. The index is log_2 of the number of sets, so the index is 3 bits as well.

That leaves 12 - 3 - 3 = 6 bits for the tag.

To compute the remaining columns, you can draw a diagram of the cache and trace each access; you can see what that looks like in the solution to Part B.

In the tag/index/offset columns below, I'm replicating the spacing from the original 12-bit address; the spacing has no other meaning.

Address	**Tag**	**Index**	**Offset**	**H/M?**	**Addresses evicted (if any)**
1010 1111 0011	1010 11	11 0	011	M	--
1011 1111 0010	1011 11	11 0	010	M	--
1010 1111 0010	1010 11	11 0	010	H	--
1011 1110 0010	1011 11	10 0	010	M	--
1010 1111 0011	1010 11	11 0	011	H	--
1011 1111 0011	1011 11	11 0	011	H	--
1010 1110 0010	1010 11	10 0	010	M	--
1011 1111 0010	1011 11	11 0	010	H	--
1011 1110 0011	1011 11	10 0	011	H	--

Part B: Final cache state (10 points)

Draw the final state of the cache, clearly labeling each block and set. Your drawing should clearly show all relevant valid, tag, and LRU bits in the cache.

Problem 5B Solution

Here's the cache after the first access in the sequence:

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	A		0			0	
101	A		0			0	
110	B	*(0xaf0 to 0xaf7)	1	1010 11		0	
111	A		0			0	

Here's the cache after the second access in the sequence:

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	A		0			0	
101	A		0			0	
110	A	*(0xaf0 to 0xaf7)	1	1010 11	*(0xbf0 to 0xbf7)	1	1011 11

111	A		0			0	

Here's the cache after the third access in the sequence. The only change is that this access to Set 110, Way A flips the ALU bit for the set back to B:

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	A		0			0	
101	A		0			0	
110	B	*(0xaf0 to 0xaf7)	1	1010 11	*(0xbf0 to 0xbf7)	1	1011 11
111	A		0			0	

Here's the cache after the fourth and fifth accesses in the sequence. The fourth access fills Way A of set 100, while the fifth access doesn't change the cache at all: the LRU bit for set 110 already points away from Way A, which is what the fifth access touches.

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	B	*(0xbe0 to 0xbe7)	1	1011 11		0	
101	A		0			0	
110	B	*(0xaf0 to 0xaf7)	1	1010 11	*(0xbf0 to 0xbf7)	1	1011 11

111	A		0			0	

Here's the cache after the sixth and seventh accesses in the sequence. The sixth access flips the LRU bit of set 110, while the seventh access populates Way B of set 100.

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	B	*(0xbe0 to 0xbe7)	1	1011 11	*(0xae0 to 0xae7)	1	1010 11
101	A		0			0	
110	A	*(0xaf0 to 0xaf7)	1	1010 11	*(0xbf0 to 0xbf7)	1	1011 11
111	A		0			0	

And here is the final cache state. The eighth and ninth accesses are both hits, so they don't evict any data or bring any data in. The only change is to the LRU bit for set 100, since the most recent access to that set is to Way B.

Set #	LRU	Way A			Way B		
		Data	Valid	Tag	Data	Valid	Tag
000	A		0			0	
001	A		0			0	
010	A		0			0	
011	A		0			0	
100	A	*(0xbe0 to 0xbe7)	1	1011 11	*(0xae0 to 0xae7)	1	1010 11
101	A		0			0	
110	A	*(0xaf0 to	1	1010 11	*(0xbf0 to	1	1011 11

		0xaf7)			0xbf7)		
111	A		0			0	

Problem 6: Cache Sizing (25 points)

Calculate the total amount of metadata required to implement the following cache designs. ***Give units (bits, bytes, etc.) for each answer.***

Part A (9 points)

A direct-mapped, write-back cache with 1 MB of data (not counting metadata) and 16B blocks. Memory addresses are 32 bits.

Problem 6A Solution

*For all parts of this problem, we will need to compute the total metadata in the cache = (metadata per block) * (number of blocks)*

For each cache design, here's how we'll compute the metadata per block:

- *1 valid bit*
- *Tag bits: need to calculate*
- *LRU bits: as directly stated or implied by the cache mapping and replacement policies*
- *1 dirty bit: for write-back caches only*

Now let's talk about the particulars of Part A.

The number of blocks in this cache = 1 MB (2^20 B) / 16 (2^4) B = 2^16 blocks.

A direct-mapped cache doesn't need LRU bits to choose a block -- the index bits choose a single unique block. So the number of LRU bits is 0.

This is a write-back cache, so there will be 1 dirty bit per block.

Finally, to calculate the tag, we need to figure out the number of offset and index bits. The number of offset bits is log_2 of the block size = log_2(16) = 4. The number of index bits is log_2(# sets). For a direct-mapped cache, there is 1 block per set, so the number of index bits is log_2(# blocks) = 16. This leaves 32 - (4+16) = 12 address bits for the tag.

*So each block has 1 valid bit + 12 tag bits + 0 LRU bits + 1 dirty bit = 14 bits of metadata. Across the whole cache, that is 14*2^16 bits of metadata = 917,504 bits = 114,688 bytes = 112 KB.*

Part B (8 points)

A 2-way set-associative, write-through cache with 1 MB of data (not counting metadata) and 32B blocks. This cache uses LRU as its replacement policy.

Problem 6B Solution

See the solution to Part A for the generic approach to this type of problem.

Here are the specifics for Part B:

The number of blocks in this cache = 1 MB (2^20 B) / 32 (2^5) B = 2^15 blocks.

For a 2-way set-associative cache, you need 1 LRU bit for every 2-block set = 0.5 LRU bits/block.

This is a write-through cache, so there's no dirty bit.

Finally, for the tag, we need to compute the number of offset and index bits. The number of offset bits is log_2 of the block size = log_2(32) = 5. The number of index bits is log_2(# sets). For a 2-way set-associative cache, there are 2 blocks per set (2 "ways"), so the number of index bits is log_2(# blocks/2) = 14. This leaves 32 - (5+14) = 13 address bits for the tag.

*So each block has 1 valid bit + 13 tag bits + 0.5 LRU bits + 0 dirty bits = 14.5 bits of metadata. Across the whole cache, that is 14.5*2^15 bits of metadata = 475,136 bits = 59,392 bytes = 58 KB.*

Part C (8 points)

A fully associative, write-back cache with 1 MB of data (not counting metadata) and 32B blocks. This cache uses approximate LRU as its replacement policy, requiring 3 LRU bits per cache block.

Problem 6C Solution

See the solution to Part A for the generic approach to this type of problem.

Here are the specifics for Part B:

The number of blocks in this cache = 1 MB (2^20 B) / 32 (2^5) B = 2^15 blocks.

The problem specifies that the replacement algorithm requires 3 LRU bits per block.

This is a write-back cache, so there's 1 dirty bit per block.

Finally, for the tag, we need to compute the number of offset and index bits. The number of offset bits is log_2 of the block size = log_2(32) = 5. The number of index bits is log_2(# sets). In a fully associative cache, your "set" is the entire cache, so there are 0 index bits. This leaves 32 - 5 = 27 address bits for the tag.

*So each block has 1 valid bit + 27 tag bits + 3 LRU bits + 1 dirty bit = 32 bits of metadata. Across the whole cache, that is 32*2^15 bits of metadata = 1,048,576 bits = 131,072 bytes = 128 KB.*