6.823 Computer System Architecture
Module #2 Solutions
(Problems M2.1 – M2.15)

Last Updated: 10/10/2011

http://csg.lcs.mit.edu/6.823/

Problem M2.1: Cache Access-Time & Performance

Here is the completed Table M2.1-1 for M2.1.A and M2.1.B.

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag	3400	3000
		Data	3400	3000
Memory array	200×$\log_2$(# of rows) + 200×$\log_2$(# of bits in a row) + 1000	Tag	4217	4250
		Data	5000	5000
Comparator	200×(# of tag bits) + 1000		4000	4400
N-to-1 MUX	500×$\log_2$ N + 1000		2500	2500
Buffer driver	2000			2000
Data output driver	500×(associativity) + 1000		1500	3000
Valid output driver	1000		1000	1000

Table M2.1-1: Delay of each Cache Component

Problem M2.1.A **Access time: DM**

To use the delay equations, we need to know how many bits are in the tag and how many are in the index. We are given that the cache is addressed by word, and that input addresses are 32-bit byte addresses; the two low bits of the address are not used.

Since there are 8 (2^3) words in the cache line, 3 bits are needed to select the correct word from the cache line.

In a 128 KB direct-mapped cache with 8 word (32 byte) cache lines, there are $4\times2^{10} = 2^{12}$ cache lines (128KB/32B). 12 bits are needed to address 2^{12} cache lines, so the number of index bits is 12. The remaining 15 bits (32 – 2 – 3 – 12) are the tag bits.

We also need the number of rows and the number of bits in a row in the tag and data memories. The number of rows is simply the number of cache lines (2^{12}), which is the same for both the tag and the data memory. The number of bits in a row for the tag memory is the sum of the number of tag bits (15) and the number of status bits (2), 17 bits total. The number of bits in a row for the data memory is the number of bits in a cache line, which is 256 (32 bytes × 8 bits/byte).

With 8 words in the cache line, we need an 8-to-1 MUX. Since there is only one data output driver, its associativity is 1.

Decoder (Tag) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps
Decoder (Data) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps

Memory array (Tag) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000
= 200 × $\log_2(2^{12})$ + 200 × $\log_2$(17) + 1000 ≈ 4217 ps
Memory array (Data) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000
= 200 × $\log_2(2^{12})$ + 200 × $\log_2$(256) + 1000 = 5000 ps

Comparator = 200 × (# of tag bits) + 1000 = 200 × 15 + 1000= 4000 ps

N-to-1 MUX = 500 × $\log_2$(N) + 1000 = 500 × $\log_2$(8) + 1000 = 2500 ps

Data output driver = 500 × (associativity) + 1000 = 500 × l + 1000 = 1500 ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware, and find the maximum.

Time to tag output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (valid output driver time)
≈ 3400 + 4217 + 4000 + 500 + 1000 = 13117 ps

Time to data output driver
= (data decode time) + (data memory access time) + (mux time) + (data output driver time)
= 3400 + 5000 + 2500 + 1500 = 12400 ps

The critical path is therefore the tag read going through the comparator. The access time is 13117 ps. At 150 MHz, it takes 0.013117 × 150, or 2 cycles, to do a cache access.

Problem M2.1.B **Access time: SA**

As in M2.1.A, the low two bits of the address are not used, and 3 bits are needed to select the appropriate word from a cache line. However, now we have a 128 KB 4-way set associative cache. Since each way is 32 KB and cache lines are 32 bytes, there are 2^{10} lines in a way (32KB/32B) that are addressed by 10 index bits. The number of tag bits is then (32 – 2 – 3 – 10), or 17.

The number of rows in the tag and data memory is 2^{10}, or the number of sets. The number of bits in a row for the tag memory is now quadruple the sum of the number of tag bits (17) and the number of status bits (2), 76 bits total. The number of bits in a row for the data memory is four times the number of bits in a cache line, which is 1024 (4 × 32 bytes × 8 bits/byte).

As in 1.A, we need an 8-to-1 MUX. However, since there are now four data output drivers, the associativity is 4.

Decoder (Tag) $= 200 \times (\#\text{ of index bits}) + 1000 = 200 \times 10 + 1000 = 3000$ ps
Decoder (Data) $= 200 \times (\#\text{ of index bits}) + 1000 = 200 \times 10 + 1000 = 3000$ ps

Memory array (Tag) $= 200 \times \log_2(\#\text{ of rows}) + 200 \times \log_2(\#\text{ bits in a row}) + 1000$
$= 200 \times \log_2(2^{10}) + 200 \times \log_2(76) + 1000 \approx 4250$ ps
Memory array (Data) $= 200 \times \log_2(\#\text{ of rows}) + 200 \times \log_2(\#\text{ bits in a row}) + 1000$
$= 200 \times \log_2(2^{10}) + 200 \times \log_2(1024) + 1000 = 5000$ ps

Comparator $= 200 \times (\#\text{ of tag bits}) + 1000 = 200 \times 17 + 1000 = 4400$ ps

N-to-1 MUX $= 500 \times \log_2(N) + 1000 = 500 \times \log_2(8) + 1000 = 2500$ ps

Data output driver $= 500 \times (\text{associativity}) + 1000 = 500 \times 4 + 1000 = 3000$ ps

Time to valid output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (OR gate time) + (valid output driver time)
= 3000 + 4250 + 4400 + 500 + 1000 + 1000 = 14150 ps

There are two paths to the data output drivers, one from the tag side, and one from the data side. Either may determine the critical path to the data output drivers.

Time to get through data output driver through tag side
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (buffer driver time) + (data output driver)
= 3000 + 4250 + 4400 + 500 + 2000 + 3000 = 17150 ps

Time to get through data output driver through data side
= (data decode time) + (data memory access time) + (mux time) + (data output driver)
= 3000 + 5000 + 2500 + 3000 = 13500 ps

From the above calculations, it's clear that the critical path leading to the data output driver goes through the tag side.

The critical path for a read therefore goes through the tag side comparators, then through the buffer and data output drivers. The access time is 17150 ps. The main reason that the 4-way set associative cache is slower than the direct-mapped cache is that the data output drivers need the results of the tag comparison to determine which, if either, of the data output drivers should be putting a value on the bus. At 150 MHz, it takes 0.0175×150, or 3 cycles, to do a cache access.

It is important to note that the structure of cache we've presented here does not describe all the details necessary to operate the cache correctly. There are additional bits necessary in the cache which keep track of the order in which lines in a set have been accessed. We've omitted this detail for sake of clarity.

D-map									
	line in cache								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
110	inv	11	inv	inv	inv	inv	inv	inv	no
136				13					no
202	20								no
1A3			1A						no
102	10								no
361							36		no
204	20								no
114									yes
1A4									yes
177								17	no
301	30								no
206	20								no
135									yes

	D-map
Total Misses	10
Total Accesses	13

4-way									LRU
	line in cache								Hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	11	inv	inv	inv	No
136						13			No
202	20								No
1A3		1A							No
102			10						No
361				36					No
204									Yes
114									Yes
1A4									Yes
177							17		No
301			30						No
206									Yes
135									Yes

	4-way LRU
Total Misses	8
Total Accesses	13

4-way									FIFO
	line in cache								Hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	Inv	inv	inv	11	inv	inv	Inv	No
136						13			No
202	20								No
1A3		1A							No
102			10						No
361				36					No
204									Yes
114									Yes
1A4									Yes
177							17		No
301	30								No
206		20							No
135									Yes

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem M2.1.D **Average latency**

The miss rate for the direct-mapped cache is 10/13. The miss rate for the 4-way LRU set associative cache is 8/13.

The average memory access latency is (hit time) + (miss rate) × (miss time).

For the direct-mapped cache, the average memory access latency would be (2 cycles) + (10/13) × (20 cycles) = 17.38 ≈ 18 cycles.

For the LRU set associative cache, the average memory access latency would be (3 cycles) + (8/13) × (20 cycles) = 15.31 ≈ 16 cycles.

The set associative cache is better in terms of average memory access latency.

For the above example, LRU has a slightly smaller miss rate than FIFO. This is because the FIFO policy replaced the {20} block instead of the {10} block during the 12^{th} access, because the {20} block has been in the cache longer even though the {10} was least recently used, whereas the LRU policy took advantage of temporal/spatial locality.

LRU doesn't always have lower miss rate than FIFO. Consider the following counter example: A sequence accesses 3 separate memory locations A,B and C in the order of A, B, A, C, B, B, B, …. When this sequence is executed on a processor employing a fully-associative cache with 2 cache lines and LRU replacement policy, the execution ends up with 4 misses. On the other hand, the same sequence will only produces 3 misses if the cache uses FIFO replacement policy. (We assume the cache is empty at the beginning of the execution).

Problem M2.2: Pipelined Cache Access

Problem M2.2.A

Ben's initial datapath design is shown below:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-back

Alyssa suggests combining the third and fourth stages, which would result in the following design (used in the MIPS R4000 processor discussed in Appendix A of the textbook):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

This scheme allows an instruction to be read from the register file before it is known whether the instruction is valid. However, reading values from the register file does not affect processor state and thus does not affect the correctness of the program execution. If the tag check fails—meaning that the fetched instruction is invalid—the incorrect instruction can be replaced with a NOP in the Execute stage, and the processor can wait for the correct instruction to be brought into the I-cache.

That raises the question of whether Ben can similarly combine the data cache tag check stage with the write-back stage. Theoretically, the answer is yes, although the issues involved with combining these two stages make it highly impractical. Thus, both answers are acceptable—the important thing to consider is the reasoning used. Combining the last two stages would result in the following pipeline:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check & Write-Back

The obvious problem with this scheme is that a load instruction that misses in the data cache will write an incorrect value into the register file—therefore merging the stages does not work. This

is correct. However, one can also argue that the scheme can be made to work by modifying the pipeline. This argument is based on the fact that even if a load instruction places incorrect data into a register, the load can re-execute and place the correct data into the register, overwriting the wrong value. As a side note, it should be pointed out that allowing processor state to be incorrectly updated in a machine which implements precise interrupts would not work without substantial hardware modifications. However, ignoring the issue of interrupts (which had not been covered in lecture at the time of the problem set), there is a more fundamental issue with this approach. Ben's pipeline currently has no means of correctly re-executing the load instruction. Simply flushing the pipeline on a data cache miss and restarting execution with the load instruction does not work because of the following type of instruction:

```
LW R1, 0(R1)
```

If the load results in a D-cache miss, it will have overwritten the value in R1 before it re-executes, meaning that the incorrect address will be calculated the second time around. Another alternative is to store the address once it has been calculated in the Execute stage. This requires special address registers in each pipeline stage starting with D-Cache Address Decode. But another problem is the fact that cache access is pipelined, so a load in the write-back stage that has caused a D-cache miss has to be sent backwards in the pipeline (along with the correct address) in order to access the cache once the correct data has been fetched. This requires additional bypass paths in the processor. In general, speculatively updating processor state requires rollback mechanisms to be implemented. Backing up the pipeline is the approach used in the MIPS R4000 in the event of a data cache miss, but the tag check and write-back stages are separate.

Problem M2.2.B

Ben's current design does not work for data writes because the tag needs to be checked before the cache is updated. One solution is to add a fourth stage which handles the actual write in the event of a cache hit. However, unless the cache can handle two simultaneous accesses, this scheme does not allow a store to be in this fourth stage at the same time that another memory operation is in the D-Cache Array Access stage. A better solution is to use a delayed write buffer as shown in lecture. The store data is written into the write buffer, and if a hit occurs in the D-Cache Tag Check stage, the data will be written into the cache at a later time (for example, when the next store instruction is processed)—the processor can continue execution as normal. This requires load instructions to check the write buffer as well as the cache to ensure that the correct value is read. With this scheme, a three-stage pipeline can be maintained for the data cache.

Problem M2.2.C

Ben's final 8-stage pipeline is shown below:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

This pipeline uses direct-mapped instruction and data caches. Replacing these direct-mapped caches with set-associative caches could potentially reduce the miss rate, at a possible cost in hit time. However, a close examination of the pipeline and the diagram for a set-associative cache (seen in Problem M2.1.B) shows that the I-cache must be direct-mapped. For a set-associative cache, when a word is being read, the result of the tag check is used as an enable signal for the value being read. However, in the above pipeline, the instruction is needed at the beginning of the I-Cache Tag Check stage so that it can be decoded in parallel with the tag check. Thus, the I-cache must be direct-mapped.

For the data cache, the tag check occurs in its own stage. This makes it possible to use a set-associative cache, since the data for a load instruction isn't needed until the beginning of the Write-Back stage. However, in practice this would probably be a bad idea, since the extra delay required to wait for the tag check before driving out the data might lengthen the clock period.

Problem M2.2.D

Pipelining the caches has a harmful effect on branches. If conditional branch instructions resolve in the Execute stage, then the processor's branch delay is 3 cycles, as shown by the following example in which there are no delay-slot instructions and the datapath is fully-bypassed:

```
    ADDI R1, R0, #1
    BEQ  R1, R0, L1
    SUB  R2, R3, R4
L1: AND  R5, R6, R7
```

	t1	**t2**	**t3**	**t4**	**t5**
IAD	BEQ				SUB
IAA	ADDI	BEQ			
ITC/ID		ADDI	BEQ		
EX			ADDI	BEQ	
DAD				ADDI	BEQ
DAA					ADDI
DTC					
WB					

Problem M2.2.E

Since a data cache access takes 3 cycles, it will take more cycles (as compared to the five-stage pipeline) to obtain the result of a load instruction. If an instruction depends on the load, a simple scheme is to wait until after the D-Cache Tag Check stage before bypassing the load value. This will ensure that the dependent instruction does not execute with incorrect data. An interlock can be used to implement this solution. If an instruction in the Instruction Decode stage needs to read the result of a load instruction that is either in the Execute, D-Cache Address Decode, D-Cache Array Access, or D-Cache Tag Check stages, then that dependent instruction will be stalled until the load reaches the Write-Back stage (at which point the load value will be bypassed to the Execute stage). This is illustrated by the below example.

```
LW R1, 0(R2)
ADD R3, R1, R2
```

	t1	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**
IAD	ADD						
IAA	LW	ADD					
ITC/ID		LW	ADD	ADD	ADD	ADD	
EX			LW				ADD
DAD				LW			
DAA					LW		
DTC						LW	
WB							LW

As shown by the above resource usage diagram, the load delay for this scheme is 3 cycles.

Problem M2.2.F

Another alternative to waiting until after the D-Cache Tag Check stage before bypassing the load value is to bypass the value at the end of the D-Cache Array Access stage. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline. In order to implement this scheme, only the program counter of the load instruction needs to be saved in the event of a tag mismatch. The load instruction will be nullified (as well as instructions behind it in the pipeline). When the **DataReady** signal is asserted (indicating that the load data is now available in the cache), the processor can restart the load instruction and continue as normal. The benefit of this scheme is that the load delay is now reduced to 2 cycles.

Problem M2.2.G

Even with the scheme in Problem M2.2.F, the load delay is 2 cycles, while it was only 1 cycle in the original 5-stage pipeline (although to be fair, the cycle time should be shorter in the 8-stage pipeline). One solution to this problem is the addition of a fast-path cache that can be accessed in one cycle. The resulting pipeline is shown below.

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	Fast-Path D-Cache Access and Tag Check & Slow Path D-Cache Address Decode	Slow-Path D-Cache Array Access	Slow-Path D-Cache Tag Check	Write-Back

The benefit of this approach is that a load instruction that hits in the fast-path cache will now have its value available at the end of the Slow-Path D-Cache Address Decode stage, whereas before it wasn't available until the end of the Slow-Path D-Cache Array Access stage. We can re-examine the instruction sequence from the solution to Problem M2.2.E:

```
LW R1, 0(R2)
ADD R3, R1, R2
```

If the fast-path cache always hits, the load delay will only be 1 cycle, which saves 1 cycle over the scheme from Problem M2.2.F and 2 cycles over the scheme from Problem M2.2.E. This scheme differs from having a single D-cache in the original 5-stage pipeline because the fast-path cache will be very small in order to avoid lengthening the cycle time. The idea is to keep the low miss rate of a large primary cache, the shorter cycle time available with a pipelined cache, and the single-cycle load delay associated with an unpipelined cache.

Problem M2.3: Victim Cache Evaluation

Problem M2.3.A **Baseline Cache Design**

Component	Delay equation (ps)		FA (ps)
Comparator	200×(# of tag bits) + 1000		6800
N-to-1 MUX	$500 \times \log_2 N + 1000$		1500
Buffer driver	2000		2000
AND gate	1000		1000
OR gate	500		500
Data output driver	500×(associativity) + 1000		3000
Valid output driver	1000		1000

Table M2.3-1

The **Input Address** has 32 bits. The bottom two bits are discarded (cache is word-addressable) and bit 2 is used to select a word in the cache line. Thus the **Tag** has 29 bits. The **Tag+Status** line in the cache is 31 bits.

The **MUX**es are 2-to-1, thus N is 2. The associativity of the **Data Output Driver** is 4 – there are four drivers driving each line on the common **Data Bus**.

Delay to the **Valid Bit** is equal to the delay through the **Comparator**, **AND** gate, **OR** gate, and **Valid Output Driver**. Thus it is 6800 + 1000 + 500 + 1000 = 9300 ps.

Delay to the **Data Bus** is delay through MAX ((**Comparator**, **AND** gate, **Buffer Driver**), (**MUX**)), **Data Output Drivers**. Thus it is MAX (6800 + 1000 + 2000, 1500) + 3000 = MAX (9800, 1500) + 3000 = 9800 + 3000 = 12800 ps.

Critical Path Cache Delay: 12800 ps

Input Address	Main Cache									Victim Cache		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
00	0								N			N
80	8								N	0		N
04	0								N	8		Y
A0			A						N			N
10		1							N			N
C0					C				N			N
18									Y			N
20			2						N		A	N
8C	8								N	0		Y
28									Y			N
AC			A						N		2	Y
38				3					N			N
C4									Y			N
3C									Y			N
48					4				N	C		N
0C	0								N		8	N
24			2						N	A		N

Table M2.3-2

15% of accesses will take 50 cycles less to complete, so the average memory access improvement is 0.15 * 50 = 7.5 cycles.

Problem M2.4: Loop Ordering

Problem M2.4.A

Each element of the matrix can only be mapped to a particular cache location because the cache here is a Direct-mapped data cache. *Matrix A* has 64 columns and 128 rows. Since each row of matrix has 64 32-bit integers and each cache line can hold 8 words, each row of the matrix fits exactly into eight (64÷8) cache lines as the following:

0	A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]	A[0][7]
1	A[0][8]	A[0][9]	A[0][10]	A[0][11]	A[0][12]	A[0][13]	A[0][14]	A[0][15]
2	A[0][16]	A[0][17]	A[0][18]	A[0][19]	A[0][20]	A[0][21]	A[0][22]	A[0][23]
3	A[0][24]	A[0][25]	A[0][26]	A[0][27]	A[0][28]	A[0][29]	A[0][30]	A[0][31]
4	A[0][32]	A[0][33]	A[0][34]	A[0][35]	A[0][36]	A[0][37]	A[0][38]	A[0][39]
5	A[0][40]	A[0][41]	A[0][42]	A[0][43]	A[0][44]	A[0][45]	A[0][46]	A[0][47]
6	A[0][48]	A[0][49]	A[0][50]	A[0][51]	A[0][52]	A[0][53]	A[0][54]	A[0][55]
7	A[0][56]	A[0][57]	A[0][58]	A[0][59]	A[0][60]	A[0][61]	A[0][62]	A[0][63]
8	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	A[1][6]	A[1][7]
•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•

Loop A accesses memory sequentially (each iteration of *Loop A* sums a row in *matrix A*), an access to a word that maps to the first word in a cache line will miss but the next seven accesses will hit. Therefore, *Loop A* will only have compulsory misses (128×64÷8 or 1024 misses).

The consecutive accesses in *Loop B* will use every eighth cache line (each iteration of *Loop B* sums a column in *matrix A*). Fitting one column of matrix A, we would need 128×8 or 1024 cache lines. However, our 4KB data cache with 32B cache line only has 128 cache lines. When Loop B accesses a column, all the data that the previous iteration might have brought in would have already been evicted. Thus, every access will cause a cache miss (64×128 or 8192 misses).

The number of cache misses for Loop A: 1024

The number of cache misses for Loop B: 8192

Problem M2.4.B

Since *Loop A* accesses memory sequentially, we can overwrite the cache lines that were previous brought in. *Loop A* will only require 1 cache line to run without any cache misses other than compulsory misses.

For *Loop B* to run without any cache misses other than compulsory misses, the data cache needs to have the capacity to hold one column of matrix A. Since the consecutive accesses in *Loop B* will use every eighth cache line and we have 128 elements in a *matrix A* column, *Loop B* requires 128×8 or 1024 cache lines.

Data-cache size required for Loop A: 1 cache line(s)

Data-cache size required for Loop B: 1024 cache line(s)

Problem M2.4.C

Loop A still only has compulsory misses (128×64÷8 or 1024 misses).

Because of the fully-associative data cache, *Loop B* now can fully utilize the cache and the consecutive accesses in *Loop B* will no longer use every eighth cache line. Fitting one column of *matrix A*, we now would only need 128 cache lines. Since 4KB data cache with 8-word cache lines has 128 cache lines, *Loop B* only has compulsory misses (128×(64÷8) or 1024 misses).

The number of cache misses for Loop A: 1024

The number of cache misses for Loop B: 1024

Problem M2.5: Cache Parameters

Problem M2.5.A

TRUE. Since cache size is unchanged, the line size doubles, the number of tag entries is halved.

Problem M2.5.B

FALSE. The total number of lines across all sets is still the same, therefore the number of tags in the cache remain the same.

Problem M2.5.C

TRUE. Doubling the capacity increases the number of lines from N to 2N. Address i and address i+N now map to different entries in the cache and hence, conflicts are reduced.

Problem M2.5.D

FALSE. The number of lines doubles but the line size remains the same. So the compulsory "cold-start" misses stays the same.

Problem M2.5.E

TRUE. Doubling the line size causes more data to be pulled into the cache on a miss. This exploits spatial locality as subsequent loads to different words in the same cache line will hit in the cache reducing compulsory misses.

Problem M2.6: Microtags

Problem M2.6.A

A direct-mapped cache can forward data to the CPU before checking the tags for a hit or a miss. A set-associative cache has to first compare cache tags to select the correct way from which to forward data to the CPU.

Problem M2.6.B

tag	Index	offset

# of bits in the tag: ____21_____

# of bits in the index: _____6_____

# of bits in the offset: _____5_____

32-byte line requires 5 bits to select the correct byte.
An 8KB, 4-way cache has 2KB in each way, and each way holds 2KB/32B=64 lines, so we need 6 index bits.
The remaining 32-6-5=21 bits are the tag.

Problem M2.6.C

If the loTags are not unique, then multiple ways can attempt to drive data on the tristate bus out to the CPU causing bus contention.

(It is possible to have a scheme that speculatively picks one of the ways when there is as match in loTags, but this would require additional cross-way logic that would slow the design down, and would also incur extra misses when the speculation was wrong.)

Problem M2.6.D

The loTag has to be unique across ways, and so in a 4-way cache with 2-bit tags the tags would never be able to hold addresses that were different from a direct-mapped cache of the same capacity. The conflict misses would therefore be identical.

Problem M2.6.E

When a new line is brought into the cache, any existing line in the set with the same loTag must be chosen as the victim. If there is no line with the same loTag, any conventional replacement policy can be used.

Problem M2.6.F

No. The full tag check is required to determine whether the write is a hit to the cached line.

Problem M2.6.G

A 16KB page implies 14 untranslated address bits. An 8KB, 4-way cache requires 11 index+offset bits, leaving 3 untranslated bits for loTag.

Problem M2.6.H

If the loTags include translated virtual address bits, then each cache line must store the physical page number (PPN) as the hiTag. An access will hit if loTag matches, and the PPN in hiTag matches. The replacement policy has to maintain two invariants: 1) no two lines in a set have the same loTag bits and 2) no two lines have the same PPN. If two lines had the same PPN, there might be a virtual address alias. Because a new line might have the same loTag as an existing line, and also the same PPN as a different line, two lines might have to be evicted to bring in one new line.

A slight improvement is to only evict a line with the same PPN if the untranslated part of loTag is identical. If the untranslated bits are different, the two lines cannot be aliases.

Problem M2.7: Write Buffer for Data Cache

Problem M2.7.A

Little's law: T = 1 / (20*2) = 1 / 40
L = 100
Therefore, N = T*L = 2.5 (entries on average)

Problem M2.7.B

Stall = (Popcount(Wbuf) >= (N – 2)) . (IR == Store)

If you assume that you can figure out the number of store instructions in flight by decoding the IR in each stage, you will be able to eliminate (-2) in the answer above.

Problem M2.7.C

Stall = (Popcount(WBuf) + Popcount(Pipeline) >N)

If you assume in the previous question that you can figure out the number of store instructions in flight by decoding the IR in each stage, you may conclude the optimization does not make any change.

Problem M2.8: Virtual Memory Bits

Problem M2.8.A

The answer depends on certain assumptions in the OS. Here we assume that the OS does everything that is reasonable to keep the TLB and page table coherent. Thus, any change that OS software makes is made to both the TLB and the page table.

However, the hardware can change the U bit (whenever a hit occurs this bit will be set) and the M bit (whenever a page is modified this bit will be set). Thus, these are the only bits that need to be written back. Note that the system will function correctly even if the U bit is not written back. In this case the performance would just decrease.

It is also important to note, that if the entry is laid out properly in memory, all the hardware-modified bits in the TLB can be written back to memory with a single memory write instruction. Thus it makes no difference whether one or two bits have been modified in the TLB, because writing back one bit or two bits still requires writing back a whole word.

Problem M2.8.B

An advantage of this scheme is that we do not need the TLB Entry Valid bit in the TLB anymore. One bit savings is not very much.

A disadvantage of this scheme is that the kernel needs to ensure that all TLB entries always are valid. During a context switch, all TLB entries would need to be restored (this is time-consuming). And, in general, whenever a TLB entry is invalidated, it will have to be replaced with another entry.

Problem M2.8.C

Changes to exceptions: "Page Table Entry Invalid" and "TLB Miss" exceptions are replaced with exceptions "TLB Entry Invalid" and "TLB No Match"

The TLB Entry Invalid exception will be raised if the VPN matches the TLB tag but the (combined) valid bit is false. When this exception is raised the kernel will need to consult the page table entry to see if this is a TLB miss (valid bit in page table entry is true), or an access of an invalid page table entry (valid bit in page table entry is false). Depending on what the cause of the exception was, it will then have to perform the necessary operations to recover.

The TLB No Match exception will be raised if the VPN does not match any of the TLB tags. If this exception is raised the kernel will do the same thing it did when a TLB Miss occurred in the previous design.

Problem M2.8.D

When loading a page table entry into the TLB, the kernel will first check to see if the page table entry is valid or not. If it is valid, then the entry can safely be loaded into the TLB. If the page table entry is not valid, then the Page Table Entry Invalid exception handler needs to be called to create a valid entry before loading it into the TLB. Thus we only keep valid page table entries in the TLB. If a page table entry is to be invalidated, the TLB entry needs to be invalidated.

Changes to exceptions: Page Table Entry Invalid exception is not raised by the TLB anymore.

Problem M2.8.E

The solution for Problem M2.8.C ends up taking two exceptions, if the PTE has the combined valid bit set to invalid. The first exception will be the TLB No Match exception, which will call a handler. The handler will load the corresponding PTE into the TLB and restart the instruction. The instruction will cause **another** exception right away, because the valid bit will be set to invalid. The exception will be the TLB Entry Invalid exception.

The solution for Problem M2.8.D will only take one exception, because the handler for Page Table Entry Invalid exception will get called by the TLB Miss handler. When the instruction that caused the exception is restarted, it will execute correctly, because the handler will have created a valid PTE and put it in the TLB.

Thus Bud Jet's solution in M2.8.D will be faster.

Problem M2.8.F

Yes, the R bit can be removed in the same way we removed the V bit in 8.D. When loading a page table entry into the TLB we check if the data page is resident or not. If it is resident, we can write the entry into the TLB. If it is not resident, we go to the nonresident page handler, loading the page into memory before loading the entry into the TLB. Thus, we only keep page table entries of resident pages in the TLB. In order to preserve this invariant, the kernel will have to invalidate the TLB entry corresponding to any page that gets swapped out. There's no performance penalty since the page was going to be loaded in from disk anyway to service the access that triggered the fault.

Problem M2.8.G

The OS needs to check the permissions before loading the entry into the TLB. If permissions were violated, then the Protection Fault handler is called. Thus, we only keep page table entries of pages that the process has permissions to access.

Problem M2.8.H

Whenever a page table entry is loaded into the TLB the U bit in the page table PTE can be set. Thus, we do not need the U bit in the TLB entry anymore.

Whenever a Write Fault happens (store and W bit is 0) the kernel will check the page table PTE to see if the W bit is set there. If it is not set the old Write Fault handler will be called. If the W bit is set, then the kernel will set the M bit in the PTE, set the W bit in the TLB entry to 1, and restart the store instruction. Thus, the M bit is not needed in the TLB either, and hence, TLB entries do not need to be written back to the page table anymore.