

CS252
Midterm 1 Practice Problems

3 October, 2007

These are practice problems to help you prepare for Midterm 1 which covers lectures 1-9. The midterm will be closed book and 80 minutes long, consisting of 3-4 problems.

Problem 1: Out-of-Order Scheduling

This problem deals with an out-of-order single-issue processor that is based on the basic MIPS pipeline and has floating-point units. The FPU has one adder, one multiplier, and one load/store unit. The adder has a two-cycle latency and is fully pipelined. The multiplier has a ten-cycle latency and is fully pipelined. Assume that loads and stores take 1 cycle (plus one cycle for write-back for loads).

There are 4 floating-point registers, F0–F3. These are separate from the integer registers. There is a single write-back port to each register file. In the case of a write-back conflict, the older instruction writes back first. Floating-point instructions (including loads writing floating point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue.

To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB).

The instructions are committed in order and only one instruction may be committed per cycle. The earliest time an instruction can be committed is one cycle after write back.

For the following questions, we will evaluate the performance of the code segment in Figure 1.1.

I ₁	L.D	F1, 5(R2)
I ₂	MUL.D	F2, F1, F0
I ₃	ADD.D	F3, F2, F0
I ₄	ADDI	R2, R2, 8
I ₅	L.D	F1, 5(R2)
I ₆	MUL.D	F2, F1, F1
I ₇	ADD.D	F2, F2, F3

Figure 1.1

Problem 1.A

For this question, we will consider an ideal case where we have unlimited hardware resources for renaming registers. Assume that you have an **infinite ROB** and that the physical (rename) register names refer to ROB tags as shown in L4-slide19.

Your job is to complete Table 1.1. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the new physical register names for each instruction, where applicable. Since we have an infinite supply of physical register names (infinite ROB), you should use a new physical register name (T0, T1, T2, etc.) each time an architectural register (F1, F2, R2, etc.) is written. Keep in mind that after an architectural register has been renamed, subsequent instructions that refer to that architectural register need to refer instead to the new physical register name.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I ₁	-1	0	1	2	L . D	T0	R2	-
I ₂	0	2	12	13	MUL . D	T1	T0	F0
I ₃	1				ADD . D			
I ₄					ADDI			-
I ₅					L . D			-
I ₆					MUL . D			
I ₇					ADD . D			

Table 1.1

Problem 1.B

For this question, assume that you have a **two-entry ROB**. Now you only have two physical (rename) register names (T0 and T1) corresponding to the 2 ROB entries. A physical register/ROB entry can be reused **one cycle** after the instruction commits. Any future reads from the destination physical register of the instruction can find the correct value in the architectural register file.

Your job is to complete Table 1.2. Fill in the cycle numbers for when each instruction enters the ROB, issues, writes back, and commits. Also fill in the physical register names for each instruction, where applicable.

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I₁	-1	0	1	2	L . D	T0	R2	-
I₂	0	2	12	13	MUL . D	T1	T0	F0
I₃	3				ADD . D			
I₄					ADDI			-
I₅					L . D			-
I₆					MUL . D			
I₇					ADD . D			

Table 1.2

Problem 2: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program:

```
loop:
    LW    R4, 0(R3)
    ADDI  R3, R3, #4
    SUBI  R1, R1, #1
b1:
    BEQZ  R4, b2
    ADDI  R2, R2, #1
b2:
    BNEZ  R1, loop
```

Assume the initial value of R1 is n ($n > 0$).

Assume the initial value of R2 is 0 (R2 holds the result of the program).

Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in lecture. We will be using a 2-bit predictor state machine, as shown below.

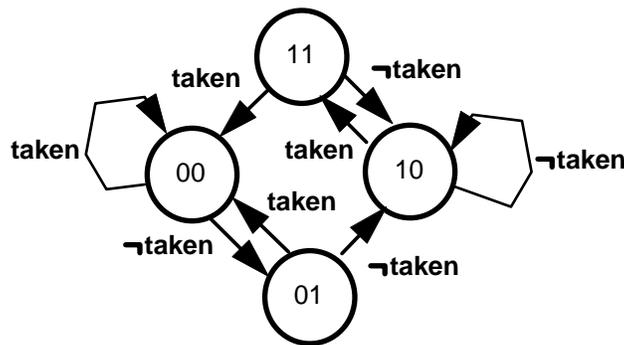


Figure 1: BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem 2.A**Program**

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem 2.B**2-bit branch prediction**

Now we will investigate how well our standard 2-bit history branch predictor performs. Assume the inputs to the program are $n=8$ and $p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0, \dots$ etc.; i.e. the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table 2.1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table 2.1 contains an entry for every time a branch (either b1 or b2) is executed. The Branch Prediction (BP) bits in the table are the bits from the BHT. If b1 is being executed, then the b1 bits from the BHT are to be filled in. If b2 is being executed, then the b2 bits from the BHT are to be filled in.

Problem 2.C**Branch prediction with one global history bit**

Now we add a global history bit to the branch predictor. Fill out Table 2.2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem 2.D**Branch prediction with two global history bits**

Now we add a second global history bit. Fill out Table 2.3. Again, compute the number of mispredicts you get for the same input.

Problem 2.E**Analysis I**

Compare your results from problems 2.B, 2.C, and 2.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For large n , what prediction scheme will work best? Explain briefly.

Problem 2.F**Analysis II**

The input we worked with in this problem is quite regular. How would you expect things to change if the input were random (each array element were equally probable 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n ?

What does this tell you about when additional history bits are useful and when they hurt you?

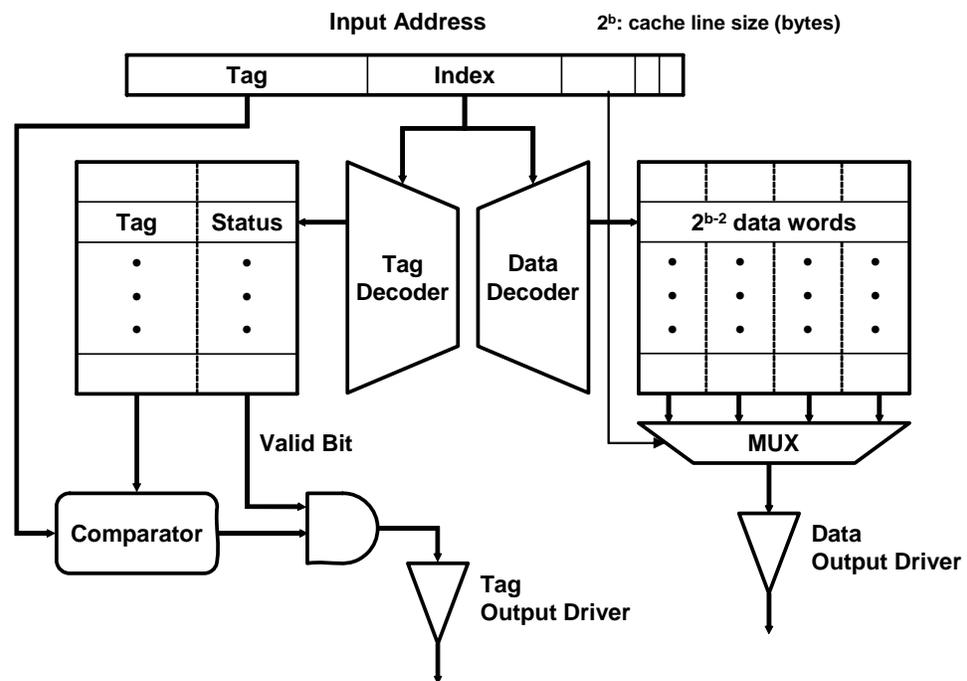
Problem 3: Cache Access-Time & Performance

Ben is trying to determine the best cache configuration for a new processor. He knows how to build two kinds of caches: direct-mapped caches and 4-way set-associative caches. The goal is to find the better cache configuration with the given building blocks. He wants to know how these two different configurations affect the clock speed and the cache miss-rate, and choose the one that provides better performance in terms of average latency for a load.

Problem 3.A

Access Time: Direct-Mapped

The following diagram shows how a direct-mapped cache is organized. To read a word from the cache, the input address is set by the processor. Then the index portion of the address is decoded to access the proper row in the tag memory array and in the data memory array. The selected tag is compared to the tag portion of the input address to determine if the access is a hit or not. At the same time, the corresponding cache block is read and the proper line is selected through a MUX.



In the memory array, each row corresponds to a row in the cache. For example, a row in the tag memory array contains one tag and two status bits (valid and dirty) for the cache line. For direct-mapped caches, a row in the data array holds one cache line.

Now we want to compute the access time of the cache. Assume a 128-KB cache with 8-word (32-byte) cache lines. The address is 32 bits, and two LSB of the address is ignored since a cache access is word-aligned. The data output is also 32 bits, and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table 3.1, fill in the column for the direct-mapped (DM) cache in Table 3.1. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 or direct-mapped caches, A for A-way set-associative caches).*

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	$200 \times (\# \text{ of index bits}) + 1000$	Tag		
		Data		
Memory array	$200 \times \log_2 (\# \text{ of rows}) + 200 \times \log_2 (\# \text{ of bits in a row}) + 1000$	Tag		
		Data		
Comparator	$200 \times (\# \text{ of tag bits}) + 1000$			
N-to-1 MUX	$500 \times \log_2 N + 1000$			
Buffer driver	2000			
Data output driver	$500 \times (\text{associativity}) + 1000$			
Valid output driver	1000			

Table 3.1: Delay of each Cache Component

What is the critical path of this direct-mapped cache for a cache read? What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a 2-input gate (AND, OR) delay is 500 ps. If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

Problem 3.B

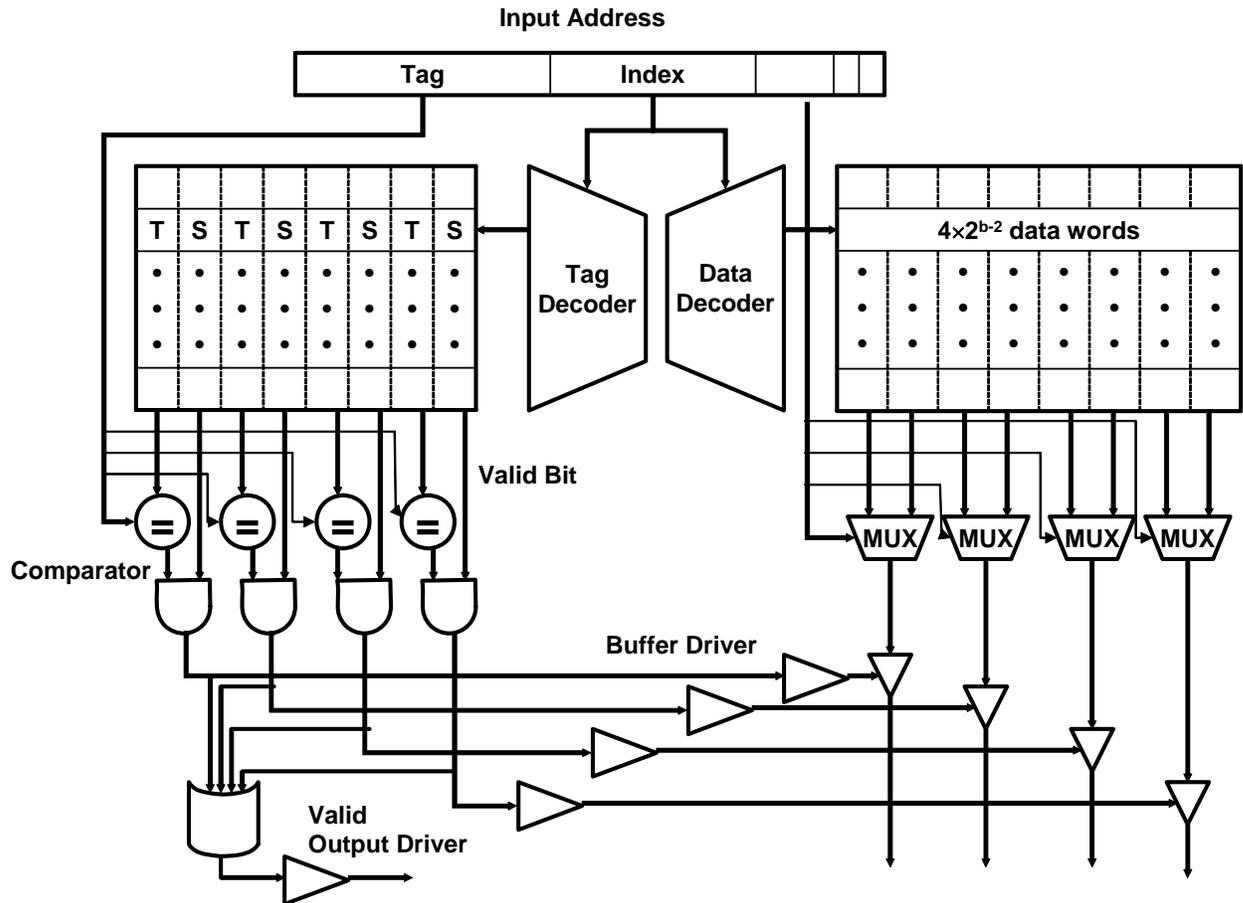
Access Time: Set-Associative

The implementation of a 4-way set-associative cache is shown in the following diagram. The index part of the input address is again used to find the proper row in the data memory array and the tag memory array. In this case, however, each row (set) corresponds to four cache lines (four ways). A row in the data memory holds four cache lines (for 32-bytes cache lines, 128 bytes), and a row in the tag memory array contains four tags and status bits for those tags (2 bits per cache line). The tag memory and the data memory are accessed in parallel, but the output data driver is enabled only if there is a cache hit.

Assume the total cache size is still 128-KB (each way is 32-KB), a 4-input gate delay is 1000 ps, and all other parameters (such as the input address, cache line, etc.) are the same

as part 3.A. Compute the delay of each component, and fill in the column for a 4-way set-associative cache in Table 3.1.

What is the critical path of the 4-way set-associative cache? What is the access time of the cache (the delay of the critical path)? What is the main reason that the 4-way set-associative cache is slower than the direct-mapped cache? If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?



Problem 3.C

Miss-rate analysis

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 8 lines with 16 bytes/line, and a 4-way set-associative cache of the same size. For the set-associative cache, Ben tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. For simplicity, assume that the addresses are only 12 bits. Complete the following tables for the direct-mapped cache and both types of 4-way set-associative caches showing the progression of cache contents as accesses occur (in the tables, 'inv' = invalid, and the column of a particular cache line contains the {tag,index} contents of that line). *You only need to fill in elements in the table when a value changes.*

D-map									
	line in cache								
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									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

		D-map
Total Misses		
Total Accesses		

4-way										LRU
	Address	line in cache								hit?
		Set 0				Set 1				
way0		way1	way2	way3	way0	way1	way2	way3		
110	inv	inv	inv	inv	11	inv	inv	inv	no	
136					11	13			no	
202	20								no	
1A3										
102										
361										
204										
114										
1A4										
177										
301										

206									
135									

4-way LRU

Total Misses	
Total Accesses	

4-way Address	line in cache								FIFO
	Set 0				Set 1				hit?
	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									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

4-way FIFO

Total Misses	
Total Accesses	

Problem 3.D

Average Latency

Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 4-way LRU 128-KB caches studied in 3.A and 3.B. What would be the average memory access latency in CPU cycles for each cache (assume that a cache miss takes 20 cycles)? Which one is better? For the different replacement policies for the set-associative cache, which one has a smaller cache miss rate for the address stream in 3.C? Explain why. Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

Problem 4: VLIW & Vector Coding

Ben Bitdiddle has the following C loop, which takes the absolute value of elements within a vector.

```
for (i = 0; i < N; i++) {
    if (A[i] < 0)
        A[i] = -A[i];
}
```

Problem 4.A

Ben is working with an in-order VLIW processor, which issues two MIPS-like operations per instruction cycle. Assume a five-stage pipeline with two single-cycle ALUs, memory with one read and one write port, and a register file with four read ports and four write ports. Also assume that there are no branch delay slots, and loads and stores only take one cycle to complete. Turn Ben's loop into VLIW code. A and N are 32-bit signed integers. Initially, R1 contains N and R2 points to A[0]. You do not have to preserve the register values. Optimize your code to improve performance but do not use loop unrolling or software pipelining. What is the average number of cycles per element for this loop, assuming data elements are as equally likely to be negative as non-negative?

Problem 4.B

Ben wants to remove the data-dependent branches in the assembly code by using predication. He proposes a new set of predicated instructions as follows:

- 1) Augment the ISA with a set of 32 predicate bits P0-P31.
- 2) Every standard non-control instruction now has a predicated counterpart, with the following syntax:

```
(pbit1) OPERATION1 ; (pbit2) OPERATION2
```

(Execute the first operation of the VLIW instruction if pbit1 is set and execute the second operation of the VLIW instruction if pbit2 is set.)

- 3) Include a set of compare operations that conditionally set a predicate bit:

```
CMPLTZ pbit,reg      ; set pbit if reg < 0
CMPGEZ pbit,reg      ; set pbit if reg >= 0
CMPEQZ pbit,reg      ; set pbit if reg == 0
CMPNEZ pbit,reg      ; set pbit if reg != 0
```

Eliminate all forward branches from Part A with the new predicated operations. Try to optimize your code but do not use software pipelining or loop unrolling.

What is the average number of cycles per element for this new loop? Assume that the predicate-setting compares have single cycle latency (i.e., behave similarly to a regular ALU instruction including full bypassing of the predicate bit).

Problem 4.C

Unroll the predicated VLIW code to perform two iterations of the original loop before each backwards branch. You should use software pipelining (and you may also use rotating register files) to optimize the code for both performance and code density. What is the average number of cycles per element for large N?

Problem 4.D

Now Ben wants to work with a vector processor with two lanes, each of which has a single-cycle ALU and a vector load-store unit. Write-back to the vector register file takes a single cycle. Assume for this part that each vector register has exactly N elements.

Ben can also eliminate branches from his code by using vector masks. He proposes a new set of maskable vector instructions as follows:

- 1) Augment the ISA with a set of 32 1-bit mask registers M0–M31.
- 2) Every standard non-control instruction now has a masked counterpart, with the following syntax:

`OP.m {OPERANDS} M#`

(This tells the vector machine to execute the instruction if M_n is set. An example of a masked instruction is `lv.m vA rA M1.`)

- 3) Include a set of compare operations that conditionally set a mask register:

```
mltz mreg,reg      ; set mreg if reg < 0
mgez mreg,reg      ; set mreg if reg >= 0
meqz mreg,reg      ; set mreg if reg == 0
mnez mreg,reg      ; set mreg if reg != 0
```

Vectorize Ben's C loop, and replace all branches using vector masks. What is the average number of cycles per element for this loop in the steady state for a very large value of N?

Problem 4.E

Modify the code from Part D to handle the case when each vector register has m elements, where m is less than N and is not necessarily a factor of N .

Problem 5: 64-bit Virtual Memory

This problem examines page tables in the context of processors with a 64-bit addressing.

Problem 5.A

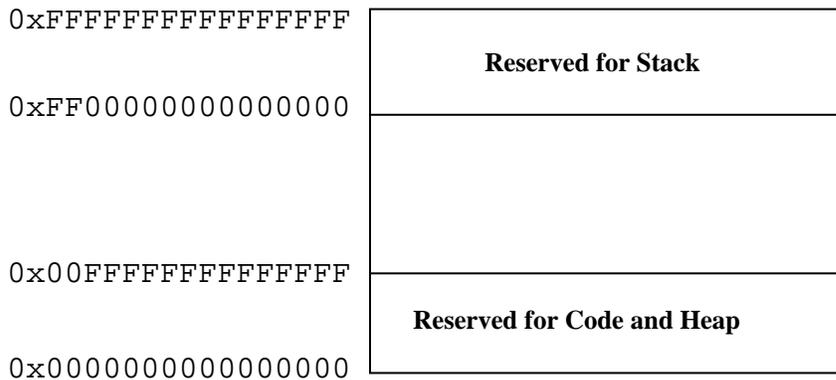
Single level page tables

For a computer with 64-bit virtual addresses, how large is the page table if only a single-level page table is used? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

Problem 5.B

Let's be practical

Many current implementations of 64-bit ISAs implement only part of the large virtual address space. One way to do this is to segment the virtual address space into three parts as shown below: one used for stack, one used for code and heap data, and the third one unused.



A special circuit is used to detect whether the top eight bits of an address are all zeros or all ones before the address is sent to the virtual memory system. If they are not all equal, an invalid virtual memory address trap is raised. This scheme in effect removes the top seven bits from the virtual memory address, but retains a memory layout that will be compatible with future designs that implement a larger virtual address space.

The MIPS R10000 does something similar. Because a 64-bit address is unnecessarily large, only the low 44 address bits are translated. This also reduces the cost of TLB and cache tag arrays. The high two virtual address bits (bits 63:62) select between user, supervisor, and kernel address spaces. The intermediate address bits (61:44) must either be all zeros or all ones, depending on the address region.

How large is a single-level page table that would support MIPS R10000 addresses? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

Problem 5.C

Page table overhead

A three-level hierarchical page table can be used to reduce the page table size. Suppose we break up the 44-bit virtual address (VA) as follows:

VA[43:33]	VA[32:22]	VA[21:12]	VA[11:0]
1 st level index	2 nd level index	3 rd level index	Page offset

If page table overhead is defined as (in bytes):

$$\frac{\text{PHYSICAL MEMORY USED BY PAGE TABLES FOR A USER PROCESS}}{\text{PHYSICAL MEMORY USED BY THE USER CODE, HEAP, AND STACK}}$$

Remember that a complete page table page (1024 or 2048 PTEs) is allocated even if only one PTE is used. Assume a large enough physical memory that no pages are ever swapped to disk. Use 64-bit PTEs. What is the smallest possible page table overhead for the three-level hierarchical scheme?

Assume that once a user page is allocated in memory, the whole page is considered to be useful. What is the largest possible page table overhead for the three-level hierarchical scheme?

Problem 5.D

PTE Overhead

The MIPS R10000 uses a 40 bit physical address. The physical translation section of the TLB contains the physical page number (also known as PFN), one “valid,” one “dirty,” and three “cache status” bits.

What is the minimum size of a PTE assuming all pages are 4KB?

MIPS/Linux stores each PTE in a 64 bit word. How many bits are wasted if it uses the minimum size you have just calculated? It turns out that some of the “wasted” space is recovered by the OS to do bookkeeping, but not much.

Problem 5.E

Page table implementation

The following comment is from the source code of MIPS/Linux and, despite its cryptic terminology, describes a three-level page table.

```
/*  
 * Each address space has 2 4K pages as its page directory, giving 1024  
 * 8 byte pointers to pmd tables. Each pmd table is a pair of 4K pages,  
 * giving 1024 8 byte pointers to page tables. Each (3rd level) page  
 * table is a single 4K page, giving 512 8 byte ptes.  
 *  
 * /
```

Assuming 4K pages, how long is each index?

Index	Length (bits)
Top-level (“page directory”)	
2 nd -level	
3 rd -level	

Problem 5.F

Variable Page Sizes

A TLB may have a *page mask* field that allows an entry to map a page size of any power of four between 4KB and 16MB. The page mask specifies which bits of the virtual address represent the page offset (and should therefore not be included in translation). What are the maximum and minimum reach of a 64-entry TLB using such a mask?

The R10000 actually doubles this reach with little overhead by having each TLB entry map *two* physical pages, but don’t worry about that here.

Table 2.1: Behavior of branch prediction

System State		Branch Predictor	Behavior		Updated Values
PC	R3/R4	BP bits	Predicted Behavior	Actual Behavior	New BP bits
b1	4/1	10	N	N	10
b2	4/1	10	N	T	11
b1	8/0	10	N	T	11
b2	8/0	11	N	T	00
b1	12/1				
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table 2.2: Behavior of branch prediction with one history bit

System State			Branch Predictor		Behavior		Updated Values		
PC	R3/R4	history bit	BP bits		Predicted Behavior	Actual Behavior	New BP bits		New history
			set 0	set 1			set 0	set 1	
b1	4/1	1	10	10	N	N	10	10	0
b2	4/1	0	10	10	N	T	11	10	1
b1	8/0								
b2	8/0								
b1	12/1								
b2	12/1								
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									
b1									
b2									

Table 2.3: Behavior of branch prediction with two history bits
 Note: history bits = 10 maps to BP set 2.

System State			Branch Predictor				Behavior		Updated Values				
PC	R3/R4	history bits	BP bits				Predicted Behavior	Actual Behavior	New BP bits				New Hist.
			set 0	set 1	set 2	set 3			set 0	set 1	set 2	set 3	
b1	4/1	11	10	10	10	10	N	N	10	10	10	10	01
b2	4/1	01	10	10	10	10	N	T	10	11	10	10	10
b1	8/0												
b2	8/0												
b1	12/1												
b2	12/1												
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													
b1													
b2													