CS 61C Fall 2003 Midterm 2 solutions

1. Who's responsible for $1?

The assembler. $1 ($at) is reserved for use by MIPS pseudo-instructions that are translated *by the assembler* into multiple MIPS machine language instructions. Most registers are allocated by the compiler to hold temporary or long-term values, but a few have special purposes, and $1 is one of those.

One point, all or nothing.

2. Which is *not* a job of the linker?

COMPUTE BRANCH OFFSETS. The assembler does this. The relocation process doesn't change the offset, because both the branch instruction itself and the instruction to which it branches are moved, so the distance between them doesn't change.

[RELOCATION is the linker's main job: Each .o file thinks it starts at address zero, and the linker moves each one to its own actual starting address (code and data separately, etc.).]

[COMBINE .O FILES is the reason relocation is necessary; that's what the "link" in "linker" means -- to combine things.]

[RESOLVE EXTERNAL SYMBOLS is also part of the combining process, so that the .o files can refer to each other.]

Scoring: Two points, all or nothing.

3. Which disk for which purpose?

The seek time is the amount of time needed before a read or write operation can begin. A small seek time is good. Disk A has the smaller seek time.

The transfer rate is the speed with which data can be read or written once the operation begins. A large transfer rate is good (it's the speed, not the time required). Disk B has the larger transfer rate.

For small files, the startup time is most important, because we'll do many small operations. So disk A is the better choice.

For large, contiguous files (it's important that they're contiguous because then the entire file can be handled in a single transfer), the transfer rate

is most important, because there will be fewer operations, but each operation will take longer. So disk B is better.

Scoring: one point each, all or nothing.

4. Network protocol stack

The bottom of the stack is low in abstraction level, dealing directly with hardware; the top is highest in abstraction, working in terms of a single (virtual) network with reliable worldwide connections:

Application level: Mail, FTP, Instant Message, **HTTP** (Web browser), etc.

Transport level: Adds reliability to worldwide Internet: TCP and UDP (TCP provides ongoing streams of in-order bits; UDP provides single packets.)

Network level: Adds worldwide connection through multiple physical nets: IP (Internet Protocol)

Link level: Device drivers for local network hardware: **Ethernet**, Token Ring, Wireless, etc.

So TCP=Transport, HTTP=Application, IP=Network, Ethernet=Link, UDP=Transport.

Scoring, 2 points, minus 1/2 point per error, but no less than zero points.

5. How long for round trip?

125 bytes = 125*8 bits = 1000 bits. If the bandwidth is 1,000,000 bits per second, it takes 1/1000 second (1 millisecond) to send a packet.

The time for the round trip is

- 2 ms outbound latency
- 1 ms outbound packet
- 1 ms to prepare the ACK
- 2 ms inbound latency
- 1 ms inbound packet

which adds up to a total of 7 milliseconds.

Scoring: 2 points, all or nothing. Since the question said "how long, in milliseconds" we accepted 7 without units specified, but we also accepted the same time expressed in other units, e.g., 7000 microseconds.

6. Why can't interrupt handler use $sp from user?

THE INTERRUPT HANDLER MIGHT OVERFLOW THE APPLICATION'S STACK SPACE. That is,
the user's $sp might point right near the bottom of a page, and if the kernel allocates stack space by decreasing $sp, its new value might point to an invalid page. If the user program does that, the result will be a TLB miss (page fault) exception, and the kernel will figure out that the user program needs more stack space and will allocate another page. But if a page fault happens in the exception handler, especially at the beginning of the handler (where we're likely to be allocating stack space) before registers have been saved, various bad things will result, such as losing track of the original EPC, or an infinite loop of page faults from the page fault handler.

(More generally, the kernel can't assume anything about the user's use of registers. For all we know, $sp might not point to a stack at all, either because of a user program bug or because of a deliberate attempt to crash the kernel.)

[Interrupt in kernel but stack in user memory: This would be okay, if we were sure that $sp really does point to some usable user memory. The user program can't access kernel memory, but the kernel *can* access user memory!]

[Interrupt might occur while $sp is being updated: Updating the stack pointer is a single machine instruction:

```
addiu $sp, $sp, -framesize
```

and interrupts don't happen in the middle of an instruction!]

[Interrupt handlers don't need a stack: This might be true, but isn't likely; OS kernels are generally written in C, except for the code at the very beginning of the exception handler that sets up registers, and this C code is compiled by the regular old C compiler, using the regular old register conventions -- args in $4-$7, temps in $8-$15, etc.]

Scoring: 2 points, all or nothing.

7. Find the bugs.

Alas, we were wrong; there were 5 or 6 bugs, depending how you count. The ones we had in mind were:

Bug 1. After reading the status register with

```
lw $t1, 0($t0)
```

we want to check the Ready bit, but we also have the IE bit in this register, and most likely that bit is on, or we wouldn't be interrupting! So before the BEQ test, we must mask off all but the Ready bit:

```
andi $t1, $t1, 1
```

Bug 2. Suppose the input buffer is full, and therefore we dismiss the interrupt without reading the character from the receiver data register. What will happen? The receiver is still ready! So it immediately interrupts again, and we never get any work done -- in particular, we'll never get a chance to empty out that buffer. So we have to read the character, even though we can't store it into the buffer. The fix is to move the instruction

```
lw $t4, 4($t0)
```

from the beginning of the fourth "paragraph" of code a little earlier, somewhere in the third "paragraph", before the BEQ that ends it.

Bug 3. In the third and fifth paragraphs, the instruction

```
addiu $t2, $t2, 1
```

that increments the rec_nextIn pointer doesn't take into account the possibility of wraparound -- if we are at the end of the buffer, we have to go back to the beginning. So in both cases we should add the instruction

```
andi $t2, $t2, 7
```

which turns the value 8 into 0. (We meant these two as one bug, but some of you counted each instance as a separate bug.)

You found two more we hadn't thought of:

Bug 4: We forgot to save and restore $at, which is used by several assembler pseudo-instructions. We should have said

```
.set noat
sw $at, 24($sp)
.set at
```

and changed the -24 to -28 in the prologue, and the obvious corresponding changes in the epilogue.

Bug 5: The instruction

```
sb $t4, rec_buffer($t2)
```

combines a 32-bit address with an index register, but you can't fit that into a MIPS instruction. We should have said

```
la $t1, rec_buffer
add $t1, $t1, $t2
sb $t4, 0($t2)
```

... but in the grading we had some disagreement as to whether or not the MIPS assembler would accept what we wrote and turn it into these machine instructions, so we're not sure this is really a bug.

Scoring: 2 points per correctly reported bug, counting bug 3 as two bugs if you counted it that way, up to a maximum of 6 points.

But there were some NON-bugs that we did not accept:

Non-bug: "You should branch to intrp instead of xmt_intrp." This would in fact *add* a bug, a really bad one! Interrupt handlers can't loop, or the user processes -- the things the user really wants to do -- will never get a chance to run! Interrupt handlers just dismiss the interrupt if there is no work to do.

Non-bug: "The LUI leaves garbage in the right half of $t0." This myth keeps popping up. LUI sets the right half of the target register to zero. If it put garbage in the right half, how do you think an ORI instruction would fix that? We would just be ORing the value we want with that garbage, producing different garbage.

Non-bug: "Turn the LW for the data register into LB" or its mirror image non-bug, "Turn the SB for the rec_buffer into SW." The load has to be LW because I/O device registers are always read and written as full words; the store has to be SB because the buffer is a buffer of characters (bytes), not a buffer of integers (words). The LW will read a word, but only the low-order (rightmost) byte of that word is meaningful.

Non-bug: "You could eliminate an unnecessary instruction by ..." Even if this

were true, it wouldn't be a bug.

Non-bug: "You can't use the user's $sp." This is well-motivated, in light of the previous question. But we explicitly said in the project that one of the simplifications we were making (compared with a real operating system) is that the entire project runs in kernel mode, and the stack in particular can be trusted.

Some people gave correct English descriptions along with incorrect code to fix the bug. (The most common one was to say

```
andi $t2, $t2, 31
```

with 31 instead of 7.) We didn't penalize these answers, since we would have accepted the English alone.

8. Cache hit rate to improve performance.

Many people didn't understand what we meant by "improve average performance." All we meant by "average" is that the cache hit rate itself is an average; you can always come up with peculiar programs that violate locality of reference and therefore don't take advantage of the cache. Some people thought "average" meant that they should average the hit time and the miss time, and therefore ended up with a cache hit rate of 50%. If you think about it, that 50% comes from your averaging, not from anything about the particular cache data!

What we wanted you to do was to compare the performance of the system with this cache against the performance of a system with the same memory but no cache. In the no-cache system, every memory reference takes 200 ns.

With the cache, we have

```
hit cost: 30 ns
miss penalty: 200 ns
miss *cost*: 30+200 = 230 ns
```

("If the desired address is not found in the cache, *then* main memory is accessed"! The two don't happen in parallel, in other words.)

So if the hit rate is R, then

```
(R * 30) + ((1-R) * 230) = 200
```

for break-even performance.

30 R + 230 - 230 R = 200
30 R + 230 = 230 R + 200
230 = 200 R + 200
30 = 200 R
R = 30/200 = 3/20 = 15/100 = 0.15 = 15%

Scoring: 2 points, all or nothing.

9. Cache geometry.

The cache holds 64 Kb of data. Each cache slot contains 32 words, or 32*4 = 128 bytes. So there are

64K/128 = 1K/2 = 512 = 2^9

cache slots. (Alternatively, 64K/128 = 2^16 / 2^7 = 2^9.)

Each cache slot has 128 = 2^7 bytes. So the offset field, which is used to find a byte or word within the slot, must be 7 bits, regardless of anything else about the cache geometry.

(P&H subdivide the offset into the byte offset within a word, which is always the rightmost two bits, and the word offset within a cache slot, which is 7-2 = 5 bits. This makes sense to a hardware designer because the bus is a word wide, so the processor always makes memory requests for a word, because that's all the bus will give you. The LB and SB instructions extract a single byte from the word internally, so the word offset and the byte offset are used by different parts of the hardware. Still, they're both selecting data from within a cache slot, so they're both part of the overall offset.)

(a) If the cache is direct mapped, a "set" is a single cache slot, so there are 2^9 sets, and so the index is 9 bits. This leaves 32-9-7 = 16 bits for the tag.

(b) If the cache is 4-way set associative, then there are 512/4 = 2^9/2^2 = 2^7 sets, so the index is 7 bits, and the tag is 32-7-7 = 18 bits.

(Yeah, we forgot to say that an address is 32 bits, but generally you made that assumption, which was what we intended.)

Scoring: One point, all or nothing, for 16,9,7 in part (a); one point, all or nothing, for 18,7,7 in part (b).

Exception: We gave one point for the six numbers (a)18,9,5;(b)20,7,5. (Exactly those six numbers, one total point.) We also gave one point for the six numbers (a)16,9,5;(b)18,7,5. These solutions came from leaving out the two "byte offset" bits from the offset, under two slightly different interpretations. The second set of six is actually better, because it's more consistent, thinking in terms of 30-bit word addresses.

10. Effects of cache associativity.

(a) More associativity means fewer conflict misses -- we avoid the situation in which there are empty slots, but we can't use any of the empty slots for a particular request because only a particular set (or, for direct mapped, a particular slot) is eligible. So the hit rate will INCREASE, which is good.

(b) More associativity means we have to be able to get the value we want from any slot in a set (for fully associative, from any slot at all). This requires more complicated circuitry to route the value to or from the right slot, so the hit cost will INCREASE, which is bad.

For part (b), several people argued that the hit cost will not change, because we ask all the slots in parallel, not one after another. That's true; the hit cost isn't multiplied by the number of slots. The added hit cost is more modest; it is caused by a deeper selection circuitry, with more gate delays to get at a particular slot. Often the complexity of the addressing hardware is proportional to the log (base 2) of the number of slots.

Scoring: one point each, all or nothing.

11. VM address calculation.

The page size is 16K = 2^14 bytes. Therefore, the 32-bit address consists of 14 bits of offset on the right, and 32-14 = 18 bits of page number on the left.

Since 14 and 18 aren't multiples of four, these fields are not whole hexadecimal digits; I found it easiest to convert the given virtual address to binary:

```
0000 0000 0010 0001 0010 0100 1111 1000
                /\
     page number /  \ offset
```

Regrouping the page number into groups of four bits, from right to left, we get

00 0000 0000 1000 0100

which is 0x00084. That's in row 1 (the second row) of the TLB, where we find the corresponding physical page number: 0x050a, which is

00 0000 0101 0000 1010

which we regroup into

0000 0001 0100 0010 10

Combining that with the original offset we have

```
0000 0001 0100 0010 1010 0100 1111 1000
                  /\
       page number /  \ offset
```

which gives 0x0142a4f8.

Most problems came from trying to make the offset or the page number fit into the overall address as hex digits.

Scoring: Two points, all or nothing.

12. Comparing physical and virtual sizes.

These aren't silly questions; virtual address space isn't exactly a size of any physical thing, but it is the size of memory-as-seen-by-applications, and it's reasonable to think that that should bear some relationship to the actual size of memory.

The computer on your desk (or your lap) probably has 4Gb of virtual address space, but it probably doesn't have nearly that much physical memory; a typical size might be 256Mb, which is 1/16th of the address space. Why can a program address more "memory" than actually exists? Of course the answer is locality of reference; only the program's WORKING SET -- the addresses that it has used recently -- has to be in physical memory to allow the program to run. So this is answer A.

The opposite situation is uncommon these days, but was quite common when 16-bit-wide computers (such as the original IBM PC and the DEC PDP-11) were

used as the first desktop machines. (In those days, serious computing was done on 32-bit or 36-bit computers that filled entire rooms.) When I (BH) was a high school teacher, we had a PDP-11 running Unix, with, if I remember correctly, half a megabyte of main memory. But a PDP-11 program could only address 64Kb (2^16) of memory! Why didn't the rest of the physical memory go to waste? Because we had 40 terminals connected to this machine, with 40 students MULTITASKING (answer C). Several people's programs were all in physical memory at once, so the operating system could often switch from one to another without waiting for disk operations.

The only common wrong answer was to get them backwards (C and A respectively); most of you understood that TLB (caching the page table) and LRU (a particular replacement policy) are irrelevant to this question.

Scoring: One point each, all or nothing.

13. What are the page table flag bits for?

The Valid bit is used to indicate that a particular virtual page is actually present in physical memory right now, because it's part of the program's WORKING SET of recently used pages. (A)

The Writeable bit is used to indicate that the user program is allowed to write into the page, because it's data unique to this process, rather than SHARED CODE which can't be written because that would affect other processes running the same program. (F)

The Referenced bit is used to indicate that this page has been used within the past clock tick (because the OS clears all the Referenced bits once per tick); this is important to help the OS remember when each page was last used, so that we can implement a LEAST-RECENTLY-USED REPLACEMENT policy. (D)

The Dirty bit is used to keep track of whether any changes have been made to the data in memory since the page was loaded into memory, so that when we have to remove the page to make room for another needed virtual page, we know whether or not we need to WRITE BACK the modified data to the disk. (C)

Answer G (heap allocation) is sort-of relevant to the Valid bit, because the working set has to do with allocated memory, but there's nothing special about the heap; code pages and stack pages can be Valid or not Valid, too.

Answer H (set associativity) is sort-of relevant to the Referenced bit, because without associativity there's no need for any replacement policy; a direct mapped cache has only one candidate slot for a given job. But

set associativity is for caches, not VM, in which pages of physical memory are always fully associative.

Answers B and E aren't even close. Virtual memory never uses write-through, and if it did, pages would *never* be dirty. And if we used random page replacement, we wouldn't care which pages have been referenced recently.

Scoring: 1/2 point each, all or nothing.