Name Solution

Computer Architecture
EE 4720
Final Examination

14 December 2006, 17:30–19:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(15 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias DVD-HD or Blu-ray?

Good Luck!

Problem 1: In the MIPS implementation on the next page some wires are labeled with cycle numbers and corresponding values. For example, c3:1 indicates that at cycle 3 the pointed-to wire will hold a 1. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. The ALU label shows the arithmetic operation performed at the indicated cycle.(20 pts)

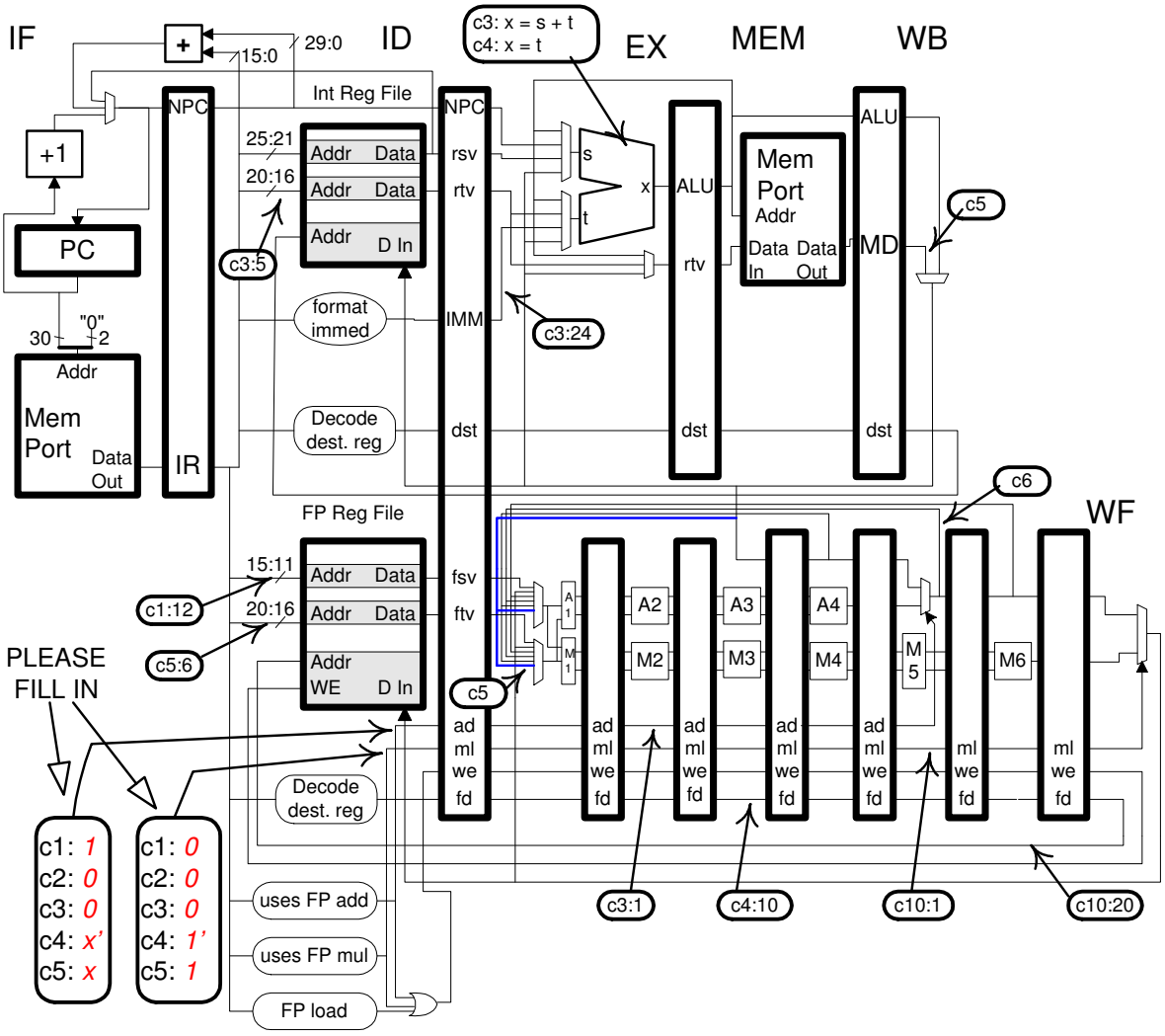
- There are no branches or other control-transfer instructions.
- There are no stalls.
- Every instruction writes the floating-point register file.
- Some instruction(s) read the integer register file.
- One instruction has only briefly been covered, make up a reasonable name for it if you don't remember it.

☑ Write a program consistent with these labels.

☑ Some registers can be determined exactly, others must be made up. **Use as many different register numbers as possible** while still being consistent with the labels.

☑ Fill in the block in the lower-left of the diagram.

Problem 1, continued:




```
# Solution Below and Above in Red
#
# Upper case shows single correct instruction or register, lower case
# shows one of several possible correct instructions or registers.

# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12
0x1000: ADD.s F10, F12, f14  IF ID A1 A2 A3 A4 _5 _6 WF
0x1004: LwC1 f16, 24(r1)        IF ID EX ME _3 _4 _5 _6 WF
0x1008: MTC1 F20, R5               IF ID EX ME _3 _4 _5 _6 WF
0x100c: mul.s f4, f16, f8             IF ID A1 A2 A3 A4 _5 _6 WF
0x1010: MUL.s f2, F10, F6                IF ID M1 M2 M3 M4 M5 M6 WF
# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12
```

Instruction at 0x1000: The c1:12 in ID indicates that the first source register is f12. The c4:10 in stage _3 directly provides the destination register, f10. The c6 in stage _5 indicates that this can't be a multiply, and since it reads the floating point register file it can only be an add.

Instruction at 0x1004: The c3:24 in EX indicates that this starts out in the integer pipeline (using an immediate of 24), the

c5 in WB indicates that it's a load, and the c5 in stage _1 shows that it must be a floating point load, such as `lwc1` (load word co-processor 1). There is no way to determine the destination or base register.

Instruction at `0x1008`: Like the previous instruction, this starts in the integer pipeline, indicated by the c3:5 in ID. The c4:x=t in EX indicates that its result is the rt register value (the instruction performs no computation). The c10:20 in WF indicates that this instruction writes register `f20`. Since it uses an rt value it can't be a load, so it's probably an `mtc1` (move to coprocessor 1), an instruction that moves a value from the integer register file to the floating-point register file.

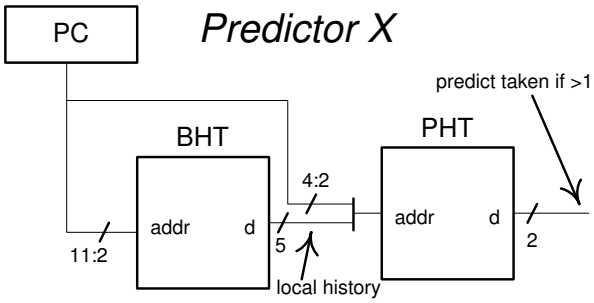
Instruction at `0x100c`: The c5 in _1 indicates that this instruction bypasses from the instruction in WB (the integer pipe) and so the second source register of this instruction must match the destination of the instruction at `0x1004`. Note that there is not enough information to determine what that register is, `f16` is an arbitrary choice.

Instruction at `0x1010`: The c10:1 in _1 indicates that this is a multiply. The c6 in _1 (or _5 from `0x1000`'s perspective) indicates a bypass with the instruction at `0x1000`, revealing the first source register, `f10`. The c5:6 in ID gives the second source register, `f6`. There is no way to determine the destination register.

AD Mux Control Signal Fill In: Values for the first fill-in are the control signals for the stage-_5 mux. A value of 0 indicates that the instruction will write a value from the integer pipeline (such as a `lwc1` or `mtc1`), a value of 1 indicates it will write a value produced by the floating-point adder (such as `add.s` or `sub.s`). For other instructions it doesn't matter what the value is. In the solution, shown in red, the x indicates either value is okay. This solution is valid when the instruction at `0x100c` is a `mul.s`; it could have been an `add.s` and if it were the value at c4 would be 1.

ML Mux Control Signal Fill In: Values for the second fill-in are the control signals for the WF-stage mux. The value should be 1 for multiply, or 0 for other instructions that write the FP register file. If an instruction doesn't write the FP register file the value doesn't matter. The solution appears in red. If the instruction at `0x100c` were an add the value at c4 would be a 0.

Problem 2: The code below runs on three systems which are identical except for the branch predictors. One system uses a bimodal predictor with a 1024-entry BHT, one uses a local history predictor with a 1024-entry BHT and a five-outcome history, and one uses Predictor X, illustrated below. (The PHT input is a concatenation of the local history and three branch PC bits.) The code below has two branches, B1 and B2, which execute in a repeating pattern as shown. There are no other branches in the code.(20 pts)



Note: In the original problem the input to Predictor X used an exclusive or rather than a concatenation.

```
LOOP:
 ..
0x1000: B1: bne r1,r2 SKIP1   N N N T T N N N T T N N N T T N N N T T ...
 ..
SKIP1:
 ..
0x1124: B2: bne r3,r4, SKIP2   N N N T T T N N N T T T N N N T T T ...
 ..
SKIP2
..
     j LOOP
```

☑ What is the accuracy of the bimodal predictor on branch B1 after warmup?

Accuracy on B1: $\boxed{\frac{2}{5}}$. Solution is straightforward.

Note: In the original exam the questions below asked about B1 rather than B2.

☑ What is the accuracy of the local predictor on branch B2 after warmup? (Do not ignore branch B1 when answering this part.)

Accuracy on B2: $\boxed{\frac{5}{6}}$. Branch B2 is predicted using six distinct local history patterns, three of which (`TNNNT`, `NNNTT` and `TTNNN`) are shared with B1. With one of the shared patterns, `NNNTT`, the next outcomes disagree: not-taken for B1 and taken for B2. Since B1 occurs slightly more often the PHT entry at address 3 (`NNNTT`) will eventually start predicting only B1 correctly. Therefore the predictions made using only five out of six patterns in B2 will be correct and the accuracy will be $\frac{5}{6}$.

☑ What is the minimum local history size for the local predictor to achieve 100% accuracy on branch B2 (without ignoring B1)?

Eight outcomes.

With eight outcomes B1 and B2 won't share patterns.

☑ What is the accuracy of Predictor X on branch B2?

Accuracy is 100%. Predictor X is similar to the local predictor except that by concatenating branch address bits with local history, two different branches (or at least those that differ in PC bits 4:2) with the same local histories will be stored in separate PHT entries. This will avoid the interference suffered by B1 and B2 in the five-outcome local history predictor.

☑ What is the minimum local history size for the Predictor X to achieve 100% accuracy on branch B2 (without ignoring B1)?

Minimum history size is three outcomes. With X one can ignore B1 because B1 and B2 will use separate entries. Then considering B2 alone, only three outcomes are needed.

☑ Explain why predictor X has lower or higher accuracy than the local predictor on branch B2.

Because it avoids interference.

☑ As indicated above, B1 is at address `0x1000` and B2 is at address `0x1124`. How would different branch addresses affect the answers above?

If bits 4:2 were the same then predictor X would perform the same as the local predictor. If bits 11:2 of the two branches were the same then the local predictor would perform more like a global predictor.

Problem 3: Consider the execution of MIPS code below. The code follows a large number of `nop` instructions. As can be seen the system below is statically scheduled. In the questions below "relatively simple" means simple compared to dynamic scheduling.

(15 pts)

```
# PC             Cycle:  0  1  2  3  4  5  6  7
0x1ff0: lh r1, 0(r2)     IF ID EX ME WB
0x1ff4: lw r4, 8(r2)     IF ID -> EX ME WB
0x1ff8: addi r6, r6, 1   IF ID -> EX ME WB
0x1ffc: xor r8, r9, r10  IF ID -> EX ME WB
0x2000: sub r11, r8, r13    IF -> ID EX ME WB
0x2004: and r14, r11, r16   IF -> ID -> EX ME WB
# PC             Cycle:  0  1  2  3  4  5  6  7
```

☑ What is the minimum fetch/decode width of a system that could produce that execution? (The x in x-way superscalar)

Minimum width: 4. It must be at least 4 because in cycle zero four instructions are being fetched at the same time.

☑ Why is the fetch/decode width above a minimum and not an exact number?

It's still possible that the system is wider, say 8-way, but is limited to aligned fetch groups and that would explain why the instruction at `0x2000` was fetched in cycle 1 rather than 0.

☑ What caused the `and` to stall in cycle 4? Could the stall be avoided?

True data dependence with the `sub`. That could not be avoided.

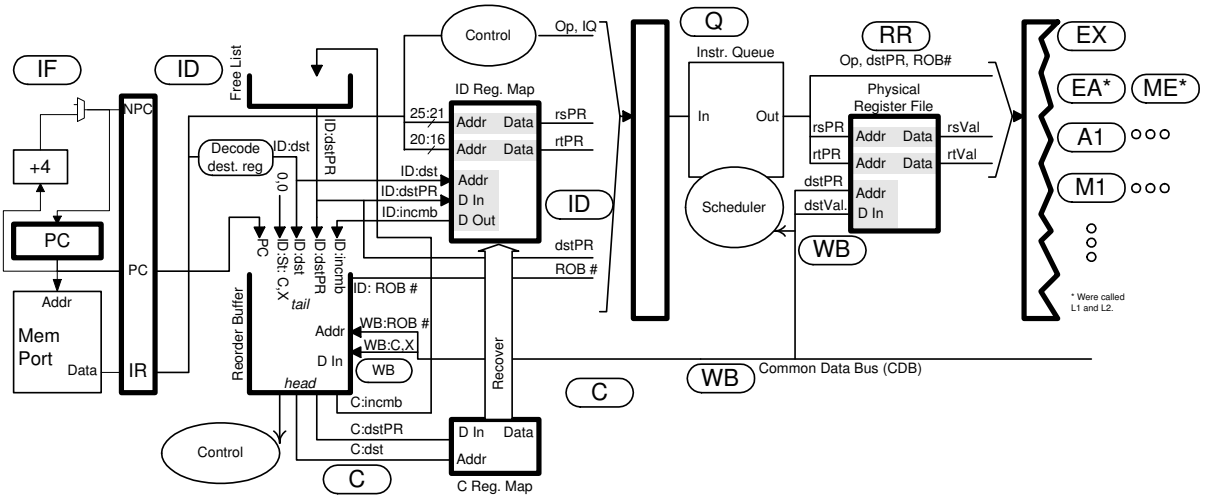
☑ What might have caused `lw` to stall? Suggest a relatively simple change to the implementation to avoid such stalls.

There may have been only one memory port in the ME stage, to be shared by all four instructions. A solution would be to include a second memory port.

☑ What might have caused `addi` and `xor` to stall? *Note: The original question also asked for a "relatively simple solution."*

If `addi` and `xor` did not stall and if new instructions moved into ID then the instructions in ID would be out of order (because of the `lw` which must stall). One could modify the control logic so that it could handle instructions being out of order in ID, the complexity would probably not be worth the effort.

Problem 4: Consider the dynamically scheduled implementation below. (15 pts)



(*a*) Where is the logic for finding the (data) dependencies that requiring bypassing most likely to be?

☑ Indicate on diagram.

Within the scheduler and instruction queue.

(*b*) Suppose due to a manufacturing error every entry in the ID register map is initialized to 12 after each reset, but the commit register map was properly initialized. Initial register values are not defined, so getting the wrong value for a register that was never written is not a problem here.

☑ Which code fragment below is more likely to encounter a problem? *Hint: It has something to do with the connection from the ID Register Map to the ROB.*

The first one.

```
# Fragment A
 add r1, r2, r3
 add r2, r1, r5
 add r1, r6, r7
 add r2, r8, r9
 nop
 ..

# Fragment B
 add r0, r2, r3
 add r0, r1, r5
 nop
 ...
```

☑ Explain what goes wrong.

Because of the flaw the first instruction gets `p12` as an incumbent for `r1` and the second instruction also gets `p12` as an incumbent for `r2`. After the third and fourth instructions commit `p12` will appear in the free list twice, and so later when they are removed from the free list the same physical register (`p12`) could be simultaneously assigned to two in-flight instructions. Each of these two instructions will expect to write its own register, instead they will be writing the same register, causing problems.

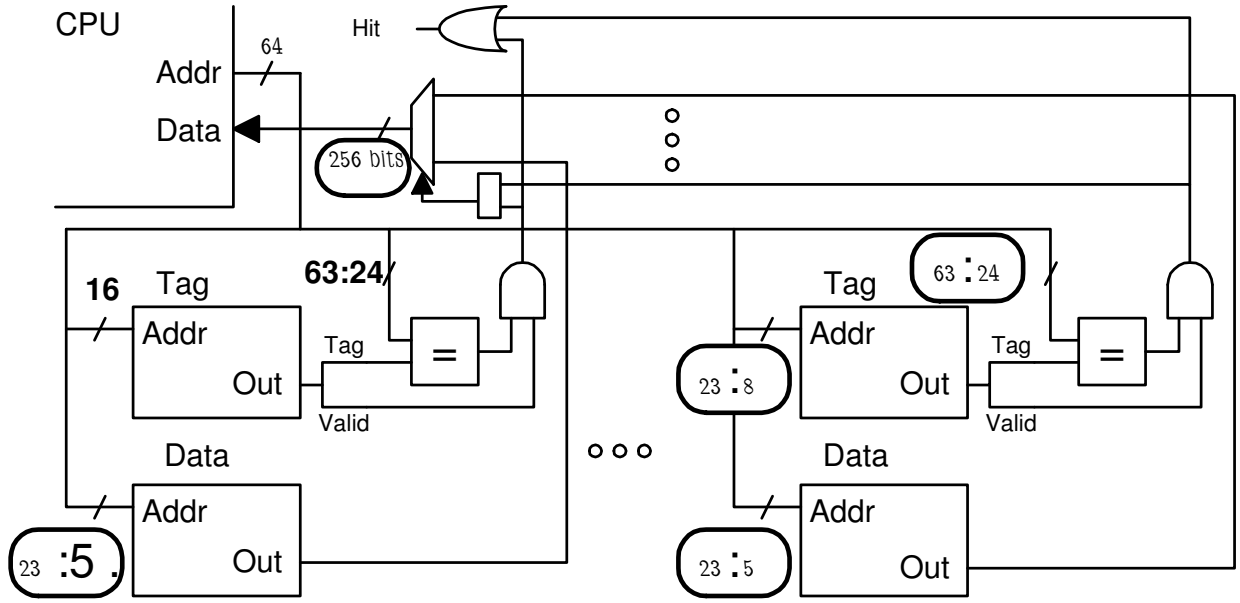
☑ Suppose millions of these defective implementations have already been manufactured. Suppose when turned on the processor starts executing code at address `0x1000`. What code could be put there to fix the problem? (It's not one of the fragments above because one of them only avoids it, later code could still trigger it.) *Hint: There's enough room for the answer below.*

If the instruction at `0x1000` raises an exception then the recovery mechanism will copy the commit map to the ID map, fixing the problem. The only tricky part is getting the exception handler in place before the first instruction executes.

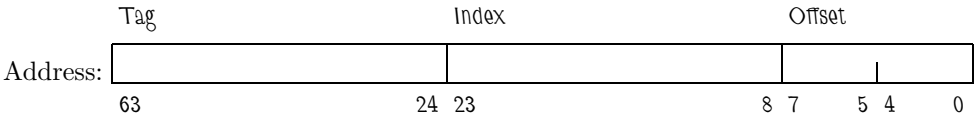
Problem 5: The diagram below is for a 64-MiB (2^{26}-character) set-associative cache on a system with the usual 8-bit characters. (15 pts)

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Associativity:

The cache is 4-way set associative.

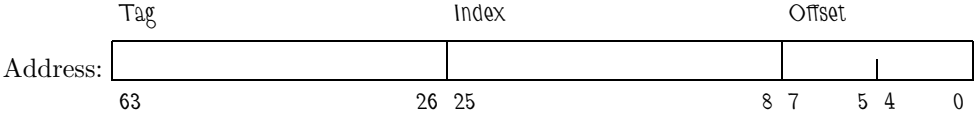
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $4 \times 2^{24-8}\,(64 - 24 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^8 = 256$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 5, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 10;  // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i * 4 ];
```

The line size is $2^8 = 256$ characters and the size of an array element is one character. The miss on the first will bring in 256 characters, a line. The second iteration will access data on this line. The line will be "used up" at $i = \frac{256}{4} = 64$, and so for each miss there are 63 hits. The hit ratio is $\frac{63}{64}$.

(*c*) The code below also runs in the cache from part a. Find the minimum values of `JLIMIT` and `JSTRIDE` that will result in the code below having a 0% hit ratio.

☑ `JSTRIDE` and `JLIMIT`

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i, j;
int ILIMIT = 1 << 10;

int JLIMIT =                                          // FILL IN

int JSTRIDE =                                         // FILL IN

for(i=0; i<ILIMIT; i++)
 for(j=0; j<JLIMIT; j++)
  sum += a[i + j * JSTRIDE ];
```

Notice that the pattern of accesses when `i=0` is the same as when `i=1`, etc., so the strategy is to make sure that whatever is cached when `j=0` is evicted by the time `j=JLIMIT`.

One way of doing that is to set `JSTRIDE` so large that each access will use a different tag (but have the same index). Since the tag bits start at bit 24 and the array element size is a character, `JSTRIDE = 1 << 24`. Since the cache is four-way set associative it can only hold four lines with the same index and different tags. So with `JSTRIDE = 1 << 24` the line brought in when `j=0` will be evicted (assuming LRU replacement) when `j=4`. By setting `JLIMIT=8` no data brought in on one `i` iteration will be present in the next one.

Problem 6: Answer each question below.

(*a*) In MIPS, SPARC, and many other RISC ISAs memory accesses are aligned and so any instruction that uses an un-aligned address, such as `0x1001` for a MIPS word, will raise an exception usually resulting in the program exiting with a Bus Error.

Perhaps due to the stress of an upcoming code freeze for a product release, a mysterious programmer at Software Company X secretly hacked the operating system of their SPARC computers so that loads and stores to un-aligned addresses would complete correctly, as though un-aligned accesses were not forbidden. There would be no more bus errors. (5 pts)

☑ How might the mysterious programmer have done it?

Modify the exception handler that is used for un-aligned accesses. If the exception were raised by a load the handler might get the data using two load instructions, say one to address `0x1000` and one to address `0x1004`, and then put together the low three bytes of the data from `0x1000` and the high byte of `0x1004`, and put that data in the destination register of the faulting load instruction. Finally, the handler would resume execution at the instruction following the load. To the program it would appear as though the load worked.

☑ Should Software Company X reward or punish the mysterious programmer? Explain.

Punish! The unaligned accesses are due to bugs in the program. The modified handler hides the bugs from the company's programmers but customers, since they don't have the hacked OS, will see them. Even if customers could be given the modified handler the programmers might still want to avoid unaligned access since it slows the program down.

(*b*) The SRAM used to implement caches is costly but in many cache designs can provide 1- or 2-cycle hit latencies. If cost were not an issue, could one use SRAM for the entire memory system and get 1- or 2-cycle latencies on *all* memory accesses?(5 pts)

☑ Explain.

No, because the time for a cache hit includes not just the time to retrieve the data from the SRAM but also the time needed for decoding the addresses and moving the data to the CPU. If SRAM were used for all memory it would have to be spread over many chips and so the part of the access latency would be chip crossing delays, contributing to an access latency larger than 1 or 2 cycles.

(*c*) Manufacturers have a great interest in having their processors score high in SPECcpu. Given this strong interest how can we be sure that benchmark selection and the run & reporting rules have not been chosen to favor a particular manufacturer? (This isn't kindergarten, so "because it's not allowed" is not a good answer.)(5 pts)

☑ Explain.

SPEC has representatives from many manufacturers. If someone proposed benchmarks or rules that favor one manufacturer's products representatives from other manufacturers can be expected to forcefully object.