Name Solution

Computer Architecture
EE 4720
Final Examination

9 December 2003, 10:00–12:00 CST

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

Alias 17 December 1903

Good Luck!

Problem 1: (20 pts) The execution of a MIPS code fragment on a dynamically scheduled machine is shown on the next page. The diagram shows the values on certain wires at certain cycles. For example, 4:65 means that at cycle 4 the labeled wire holds value 65. The physical register file table is completed, ID- and commit-map tables are blank.

- The FP add unit has 3 stages, the FP multiply unit has 5, and the EA and ME are used for loads and stores.
- All destination registers are floating point.
- WB and commit can be done in the same cycle (indicate with a WC).
- To keep things simple the result of every instruction is zero and there are no cache misses.

(*a*) The ID and commit register map tables are blank ...

☑ ... complete them (the ID and commit register map tables.)

☑ Show the correct architected register numbers, or for partial credit make them up. (Two are easy, the rest are interesting.)

☑ Show the initial values (just before cycle 0) in the ID and commit map tables.

Solution appears on next page.

(*b*) Complete the pipeline execution diagram.

☑ Be sure to show Q, RR, WB, C (or WC), and a possible functional unit.

(*c*) Write a program consistent with these tables and labels.

☑ Choose consistent instructions.

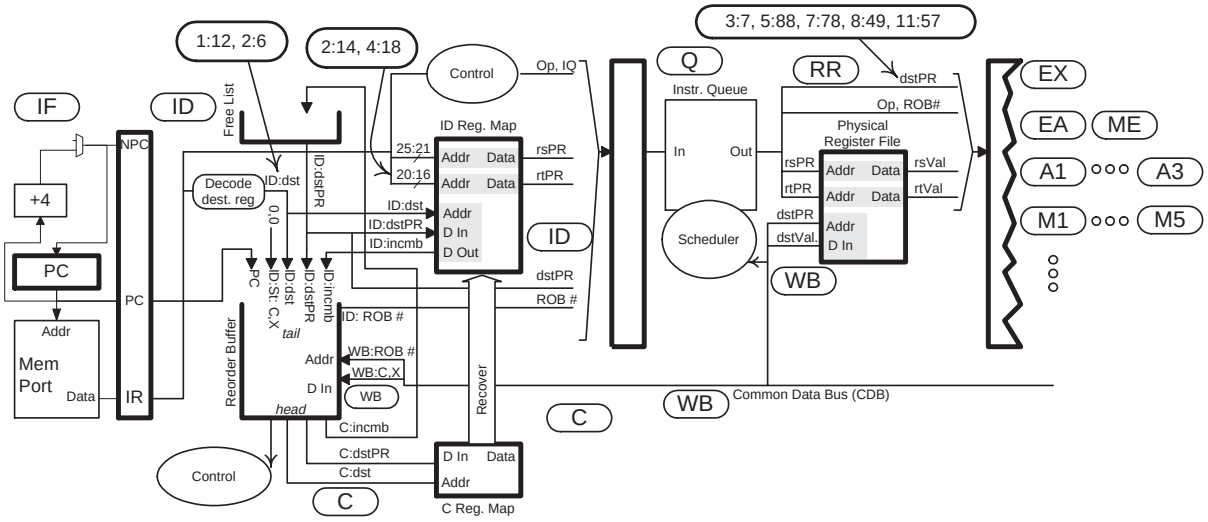
☑ Choose consistent registers. If a register number cannot be determined, use a question mark.

- *Hint 1: In the physical register file table, put a "1" next to the first (earliest) register removed from the free list, put a "2" next to the second register removed from the free list, and so on. Similarly, put a "1" next to the first register put back in the free list, etc. To figure out which physical register belongs to which instruction destination (easy) use the fact that certain events occur in program order.*
- *Hint 2: To figure out which architected register an instruction is writing (interesting) remember what causes a register to be put back in the free list.*

The solution appears on the next page. The key to completely solving the problem is understanding the physical register file. Physical registers are assigned in ID and that occurs in program order, and so the first physical register to be assigned (7) goes to the first instruction, the second physical register to be assigned (49) goes to the second instruction, etc. Physical registers are returned to the free list when instructions commit, so the first instruction commits at cycle 9, the second instruction commits at cycle 12, etc. When an instruction commits **it does not free its own physical register**, instead it frees the incumbent, the physical register used by the last instruction to write the same architected register. For example, when `ldc1` commits it frees physical register 7 which was assigned to `mul.s` because `mul.s` is the most recent instruction that writes `f12`, the same architected register that `ldc1` writes.

Every instruction goes to Q in the cycle after ID. The time an instruction uses RR is determined from the diagram (see the big bubble pointing to `dstPR`), the time it uses WB is determined by the physical register file (when it writes a value, 0. in each case here). The type of instruction is determined by the amount of time between RR and WB. For example, for the first instruction there are five cycles between RR and WB so it can only be using the multiply functional unit, and so it is most likely a multiply. (It's possible some other instruction uses the FP multiply unit, but in class it was always a multiply.)

Problem 1, continued: See previous page for instructions.




```
# Solution
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

mul.s f12, f14, f16   IF ID Q  RR M1 M2 M3 M4 M5 WC
add.s  f6, f12, f14      IF ID Q              RR A1 A2 A3 WC
ldc1.s f12, 0(r1)           IF ID Q  RR EA ME WB             C
add.s f12, f12, f18            IF ID Q     RR A1 A2 A3 WB       C
add.s f6, f12, f6                 IF ID Q              RR A1 A2 A3 WC
```

```
# ID Map       Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f6       95                 49       57
f12      33             7     88 78
# Commit Map   Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f6       95                                                49       57
f12      33                                        7           88 78
```

```
# Phys. Reg. File  Cy 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 7                      [                                   0.              ]
33                                                          ]
49                          [                                       0.          ]
57                                  [                                           0.
78                                      [                       0.
88                              [                   0.                      ]
95                                                                  ]
# Phys. Reg. File  Cy 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Problem 2: (20 pts) In all of the problems below please check the code samples carefully for dependencies. All implementations below are fully bypassed. Please check the code samples carefully for dependencies.

Grading Note: A latency of 3 means a four-stage adder. Most people did not get this right, but points were not deducted for using a four-stage adder. (The initiation interval still has to be correct.) I think this will be the last test where the term latency in this sense is used.

(*a*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 2.

☑ Show a pipeline execution diagram for the code.

```
# Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
add.s f2, f4, f6    IF ID A1 A1 A2 A2 WF
add.s f8, f2, f12      IF ID -------> A1 A1 A2 A2 WF
add.s f14, f10, f16       IF -------> ID -> A1 A1 A2 A2 WF
```

Solution above. Note that the stage names are A1 and A2.

(*b*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☑ Show a pipeline execution diagram for the code.

```
# Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
add.s f8, f2, f12      IF ID -------> A1 A2 A3 A4 WF
add.s f14, f10, f16       IF -------> ID A1 A2 A3 A4 WF
```

Problem 2, continued:

(*c*) The code below executes on a 2-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☑ Show a pipeline execution diagram for the code.

```
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 # Solution
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 add.s f8, f2, f12   IF ID ----------> A1 A2 A3 A4 WF
 add.s f14, f10, f16    IF ----------> ID A1 A2 A3 A4 WF
 and r1, r2, r3         IF ----------> ID EX ME WB
```

Note that the third instruction does not enter ID even though there is space. This ensures that instructions in ID are in program order. Though its possible to design control hardware to handle out-of-order instruction in ID its probably not worth the trouble.

Grading Note: Many solutions used a second FP adder. It's not wrong, but its not necessary.

(*d*) In a correct solution to the problem above there should be at least one instruction for which a precise exception is impossible. If that describes your solution, show a pipeline execution diagram below in which all instructions could raise precise exceptions (even though they don't). It's also possible that in a correct solution to the problem above all instructions can raise precise exceptions. If so, show a pipeline execution diagram below in which some instructions cannot raise a precise exceptions. In the absence of exceptions all pipeline execution diagrams must show correct execution.

☑ Show the appropriate pipeline execution diagram, or show how the one above would be different.

☑ Identify those instructions for which precise exceptions are impossible (above or below) and explain why.

```
 # Solution
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 add.s f8, f2, f12   IF ID ----------> A1 A2 A3 A4 WF
 add.s f14, f10, f16    IF ----------> ID A1 A2 A3 A4 WF
 and r1, r2, r3         IF ----------> ID ----> EX ME WB
```

An instruction cannot raise a precise exception if an instruction after it writes back before the exception is detected. In the previous part suppose the third `add.s` raised an exception in cycle 10. It could not be precise because of the `and` instruction's writeback in cycle 9. One solution is to stall `and` so that it does not write back before the third `add.s`, that is what is shown above.

Problem 2, continued:

(*e*) The code below executes on a 2-way superscalar dynamically scheduled machine using method 3 (the only one covered this semester), the same one used in Problem 1. The FP add unit has a latency of 3 and an initiation interval of 1.

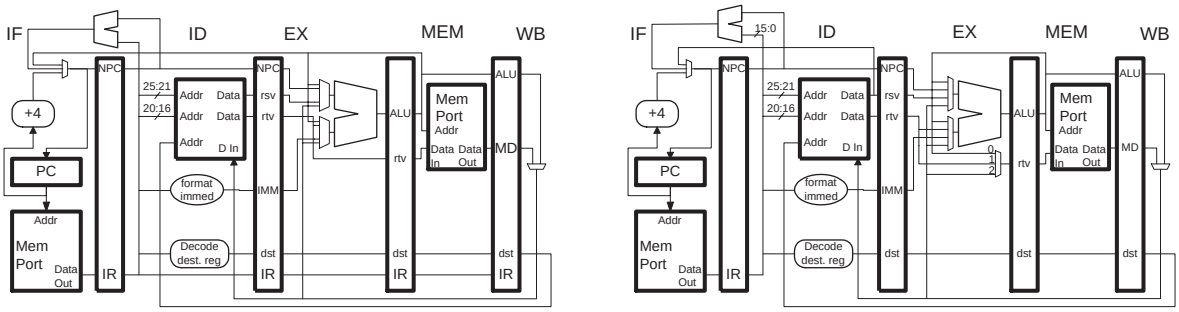
☑ Show a pipeline execution diagram. Don't forget the commit stage.

Assume an unlimited number of reorder buffer entries and physical registers.

```
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 # Solution
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID Q  RR A1 A2 A3 A4 WC
 add.s f8, f2, f12   IF ID Q              RR A1 A2 A3 A4 WC
 add.s f14, f10, f16    IF ID Q  RR A1 A2 A3 A4 WF       C
 and r1, r2, r3         IF ID Q  RR EX WB                   C
```

Grading Note: Many solutions had only one instruction committing in cycle 12. A 2-way superscalar processor must be able to commit two instructions per cycle. Also, many solutions used a second adder. A second FP adder is not needed in any of the problems.

Problem 3: (20 pts) Two MIPS implementations are illustrated below, the right one has a multiplexor at the input to the `EX/MEM.rtv` pipeline latch, the left one does not.



(*a*) Provide two code samples, one in which the multiplexor is useful and one in which it is not. Briefly explain.

```
# Mux Useful
add r1, r2, r3
sw r1, 0(r4)   # Mux used to bypass r1

# Mux Not Useful
add r1, r2, r3
add r4, r5, r1
sw r5, 0(r4)   # Not useful, r5 read from register file.
```

(*b*) Suppose version 5.11 of a compiler was written for the implementation on the left and is in the hands of customers. Version 5.99 of the compiler also includes the right implementation and is being released soon. Which two compilation options would you have to use to take advantage of the changes made for the right implementation? (The exact names of the compiler options is unimportant, but it should be obvious what they do.) Briefly explain why each option is necessary and how it would affect the code.

A compiler option is something put on the compiler command line or selected from a dialog box which tells the compiler how or what to compile.

The compiler for the left implementation would schedule instructions before a store so that the instruction producing the store value was more than two instructions before the store (the Mux Not Useful case). To take advantage of the new mux we would need to tell the compiler it is compiling for the right implementation and that it should optimize code. The options might be `-O` for optimization and `-impl=right` to select the right implementation. (Since many customers have the left implementation it would not be a good idea to compile for the right implementation by default.)

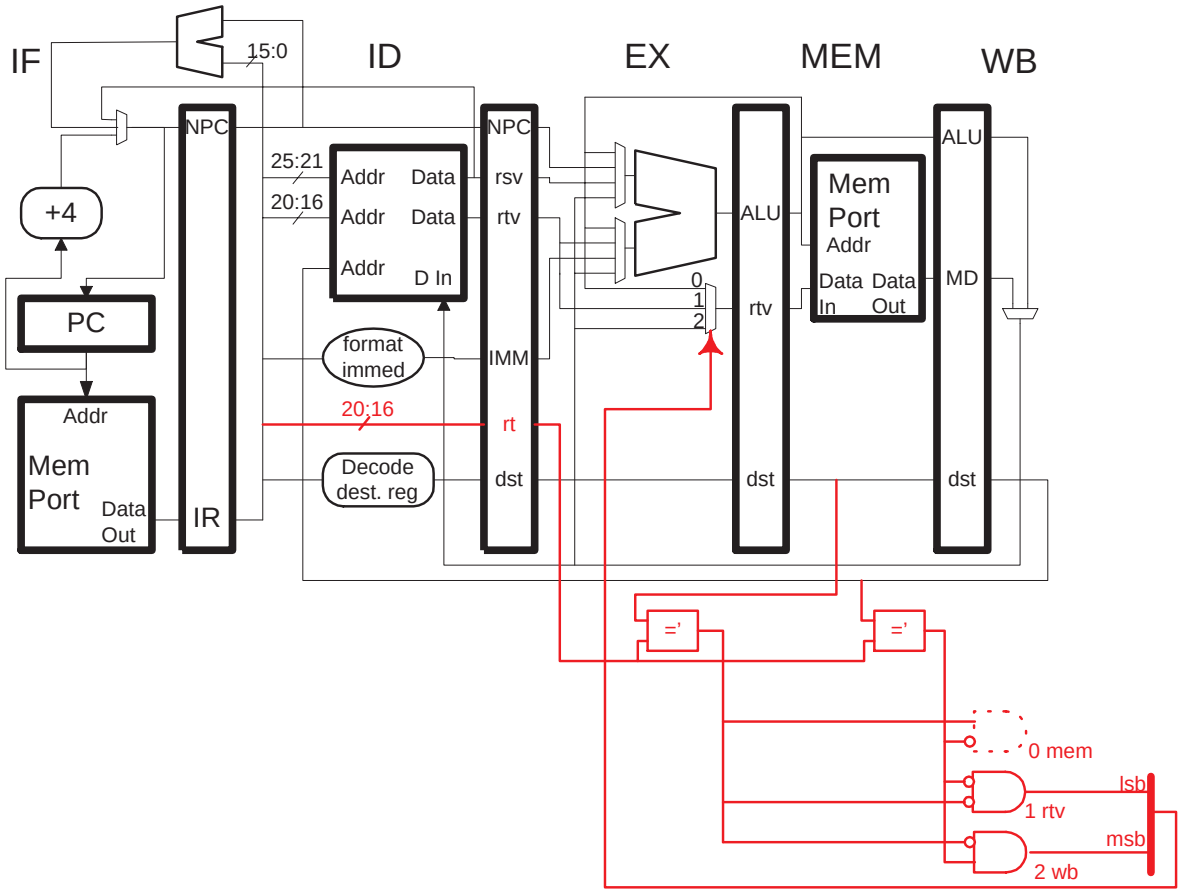
Grading Note: Many solutions correctly described what the compiler should do but did not specify anything like a compiler option. The question is asking about how compilers are used and so such answers did not receive full credit.

Problem 3, continued:

(*c*) Design the control logic for the `rtv` multiplexor. Unlike Homework 4, **the logic must be in the EX stage**. Where appropriate, show which bits are being used, e.g., `12:5`.

Changes shown in red below.

Grading Note: Too many solutions used `rtv` instead of `rt` in the =' . If it's not obvious why this is wrong then please review the basics of how the pipelined MIPS implementation works.



(*d*) There is a good reason why the control logic for the ALU input multiplexors should be in the ID stage that does not apply to the `rtv` multiplexor control logic. What is the reason, and why does it not apply to the `rtv` logic?

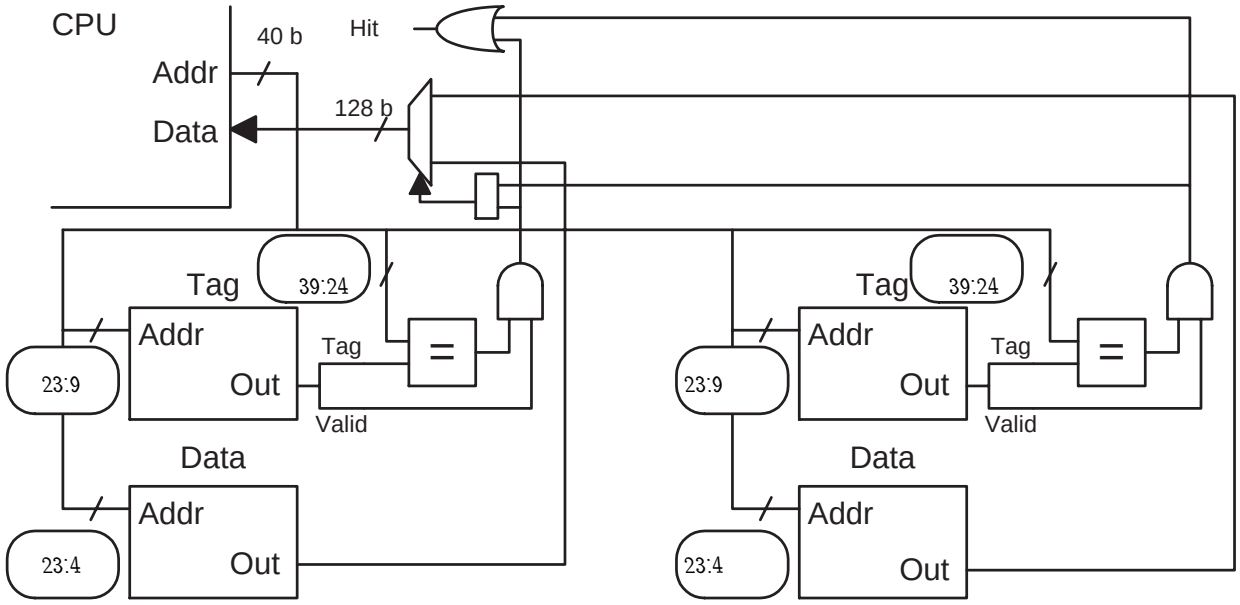
The ALU probably needs most of a clock cycle to compute. If the ALU input multiplexor logic were in EX then the ALU would not start useful computation until after the control logic generated the correct mux inputs and the muxen switched their inputs. Thus the control logic would be part of the critical path. Moving the logic to ID shortens the critical path. On the other hand, the `rtv` multiplexor is not part of a critical path, its output goes straight to the pipeline latch.

Grading Note: Some answered that the control logic for the ALU muxen would need information only available in ID, for example, the opcode. However, the `rtv` mux also needs information from ID (though not the opcode) so that reason applies to both. The ALU muxen would need the `rs` field value (and also `rt`) and one bit specifying whether the instruction uses the immediate, which could easily be passed through the pipeline latches.

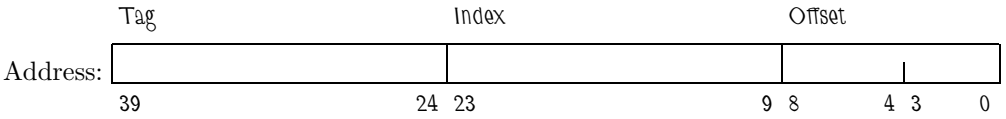
Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25} bytes) cache with 512-byte (2^9-byte) lines on a system with 8-bit characters.

(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.)

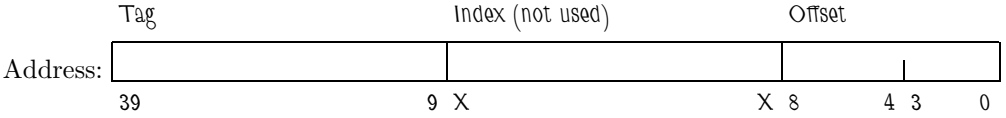


☑ Associativity: 2

☑ Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 \times 2^{24-9}(40 - 24 + 1)$ bits.

☑ Show the bit categorization for a **fully associative** cache with the same capacity and line size. *Note: Emphasis not included in original exam.*



Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

The element size is one character. The line size is $2^9 = 512$ characters and so 512 consecutive `i` iterations will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $511/512$. The total amount of memory accessed is 1024 characters, which is much smaller than the 32 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{511}{512}+1\right)=\frac{1023}{1024}.}$

(*c*) Find the hit ratio for the code below running on the cache from the first part. Consider only accesses to the arrays and assume the cache starts out cold. State any assumptions made.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
char *b = 0x2000000;  // sizeof(char) = 1 character
char *c = 0x3000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ] + b[ i ] + c[ i ];
```

The cache is two-way set associative with a tag field that starts at bit 24, which would be the sixth hexadecimal digit in the arrays' starting address. Therefore the three arrays will have the same index (for the same value of `i`). Since at most cached two lines can have the same index the first access to `c` will result in the eviction of either `a` or `b`, if an LRU replacement policy is used then `a` will be the unlucky line. The next access to `a` will therefore be a miss instead of a hit, with LRU replacement `b` will be evicted. This pattern will continue resulting in a hit ratio of 0%.

(*d*) Modify the addresses of `a`, `b`, and `c` to maximize hit ratio. Explain how the modified addresses improve hit ratio.

The misses will be avoided if the three arrays use disjoint indices. There are many ways of doing this. One way is to keep the arrays close together, in that case: `a = 0x1000000, b = a + 1024 = a + 0x400 = 0x1000400`, and `c = b + 0x400 = 0x1000800`.

Problem 5: (20 pts) Answer each question below.

(*a*) The diagram below shows the branch outcome patterns for two branches.

```
# Loop contains only the branches shown.
BIGLOOP:
 Solution: Counter Value:      0  0  0  0  0  1  2  1  0  0  0  1  2  1  0  0  0  1  2
B1: 0x1000 beq $t1, $t2, SKIP1   N  N  N  N  T  T  N  N  N  N  T  T  N  N  N  N  T  T
 Solution: Misprediction        N- N- N- N- NX NX TX N- N- N- NX NX TX N- N- N- NX NX
...
B2: 0x1200 beq $v0, $v1, SKIP2    T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
...
    0x2010 j BIGLOOP
```

☑ How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table (BHT)?

The prediction accuracy is 50% (for a large number of iterations).

Grading Note: There were two common errors.

Error 1: Many predicted a branch *after* updating the counter using the branch outcome. (It should be obvious why this is wrong, or if not wrong, cheating.)

Error 2: Many computed the prediction ratio using the wrong number of branches. The repeating pattern contains six branches and that's what the prediction should be based on. The prediction outcomes have to repeat too which is why one should not base the prediction ratio on the first few branches.

☑ What is the minimum size of the BHT for which the accuracy in the previous part is possible? Explain.

The minimum size is $2^8 = 256$ entries.

If the first and second branch use the same entry in the BHT then they will interfere with each other. The first nine bits of the addresses of the two branches are identical. If those bits were used to index (as an address for) the BHT then the two branches would share an entry. The two low bits, being zero, are not used to index the BHT, and so in a table with a 7-bit address (2^7 entries) the two branches share an entry. If the table had an 8-bit address then the two branches would use separate entries and so the prediction accuracy would be preserved.

☑ Why might it be pointless to perform branch prediction in the ID stage of the 1-way statically scheduled pipeline used in class?

Because the branch outcome is determined in the ID stage and so there is no need to predict it there.

(*b*) Answer the following questions about exception codes as defined for SPARC V8 and using the class terminology.

☑ What is an exception code number (or trap type)? *Note: In the original exam the question was shorter: "What is an exception code?"*

It is a number that identifies the type of trap, hardware interrupt, or exception.

☑ How is it obtained for traps?

The trap instruction adds its `rs1` operand to an immediate or `rs2` operand, the low seven bits of the sum is the exception code.

☑ How is it obtained for hardware interrupts?

The exception code corresponds to the interrupt request port that was used to raise the interrupt.

☑ How is it obtained for exceptions?

It originates with the hardware that detects the exception, for example, an illegal opcode by the instruction decode hardware or segmentation fault by the data memory port.

☑ How is it used to start the handler?

The exception code is combined with the contents of trap base register to form the address of the handler. The high bits of the address are the contents of the TBR, the middle eight bits are the exception code, and the low four bits are zero (each table entry is 16 characters [4 instructions]).

(*c*) Consider two processors, one is a 6-way superscalar implementation of an ordinary ISA, say MIPS, the other is a 6-way implementation of a VLIW ISA, say Itanium (IA-64).

☑ Describe two features of Itanium (or some other VLIW ISA) that would allow it to execute faster than the superscalar implementation. Explain how the features allow faster execution.

Feature 1: Dependencies are specified in the bundle. This allows faster implementations by reducing the complexity of—and critical path length through—the logic checking for dependencies. With a shorter critical path the clock frequency can be higher.

Feature 2: Instructions arranged into bundles and CTI targets limited to bundle starts. In a non-VLIW superscalar processor in which instruction fetch is aligned on fetch groups unneeded (and ultimately unused) instructions may be fetched that precede the branch target because they are part of the same fetch group. With targets limited to bundle starts VLIW implementations do not suffer this inefficiency. A superscalar processor that was not restricted to fetch-group aligned fetches would have more complex logic in the fetch stage (or the cache port) to rearrange instructions, that would impact clock frequency or add stages, increasing branch penalty.

Feature 3: Instruction placement in VLIW is limited, for example, a VLIW ISA could forbid memory instructions in the first slot of a bundle. With such restrictions an implementation would require fewer multiplexors to send instructions to the proper functional unit. This certainly simplifies the logic (providing more room for cache or for other performance improving features) and might reduce critical path length.

Grading Note: Many people described the features but did not describe how they would improve performance.

☑ If the implementation for the VLIW ISA were faster why might the superscalar implementation still be better from a business perspective?

Suppose a company has many customers using an existing non-VLIW ISA. If the company were to offer a VLIW processor to customers using implementations of the non-VLIW ISA as the only way of getting higher performance from the company, those customers would have to re-compile their applications or buy new applications. Not every customer would be willing to do so. Some customers may be choosing the company's processor over competitors' to avoid the hassle of porting their applications to the competitors ISA. If such customers have to re-compile anyway they may switch to the competitors product.