Name Solution

Computer Architecture
EE 4720
Final Examination

12 December 2002, 10:00–12:00 CST

Problem 1		(10 pts)
Problem 2		(17 pts)
Problem 3		(11 pts)
Problem 4		(17 pts)
Problem 5		(45 pts)
Exam Total		(100 pts)

Alias Currently Accurate, Full, and Complete

Good Luck!

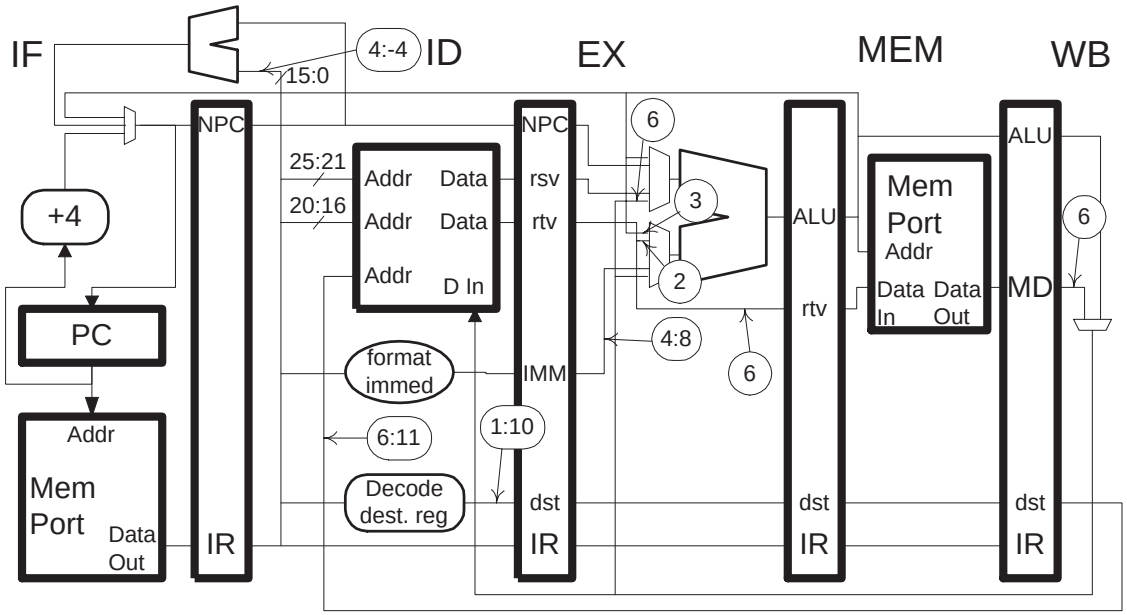
Problem 1: In the diagram below certain wires are labeled with cycle numbers and, in some cases, values that will then be present, for example, 2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions, use the PED to help solve the problem. (10 pts)

☑ Write a program consistent with these labels.

☑ Use labels for branch targets (if any) and label the target line.

Solution shown below.

Grading Note: Many people got this 100% correct, but I was hoping everyone would. For those currently taking EE 4720, study this well, it's not that difficult to get full credit.




```
 Cycle                  0  1  2  3  4  5  6  7  8
LOOP:
 add  $10, $2, $1       IF ID EX ME WB
 add  $9, $4, $10          IF ID EX ME WB
 lw   $11,  8($6)             IF ID EX ME WB
 bneq $1, LOOP                   IF ID EX ME WB
 sw   0($11), $7                    IF ID EX ME WB
```

Problem 2: The PED below shows execution on a defective 1-way dynamically scheduled machine using Method 3. A diagram appears on the next page. The circuitry that's supposed to restore the ID map after a branch misprediction is not working, the ID map is left unchanged. Everything else works correctly, in particular the free list is correctly restored to the exact state it was right after the `lw`. (17 pts)

(*a*) For this incorrect execution:

☑ Show where each instruction commits.

☑ Complete the ID map. (See the phys. reg. file for the free list.)

☑ Complete the commit map, include the initial state.

☑ Complete the physical register file.

☑ **In addition to other information** the physical register file should include a `[` in the cycle a register is removed from the free list and a `]` in the cycle it is returned to the free list.

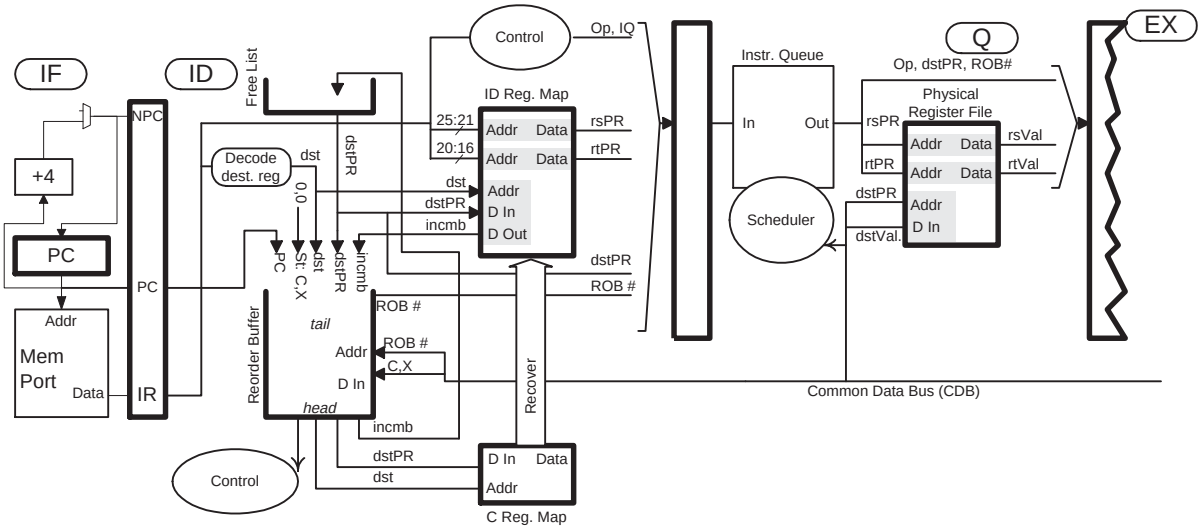
☑ Circle incorrect entries (due to the defect) in the tables below.

```
 # SOLUTION.  (See next page for notation and discussion.)
 # lw loads a 0x200, lb loads a 0x202
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 bne $s1, $s2  SKIP  IF ID            Q  B  WC
 lw  $t1, 16($t1)       IF ID       Q  L1       L2 WC (18)
 addi $t4, $t4, 3          IF ID Q  EX WB    x (squash) (20)
 ori $t5, $t4, 0x30           IF ID Q  EX WB x (squash) (23)
 nop ... (lots of nops)
SKIP:
 lb   $t1, 16($t1)                                IF ID Q  L1       L2 WC   (20)
 addi $t4, $t4, 3                                    IF ID Q  EX WB (23)   C
 ori  $t5, $t4, 0xc0                                    IF ID Q  EX WB (30)   C
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
ID Map
 t1   3                   18                         20
 t4   7                      20                         23
 t5  10                         23                         30
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Commit Map
 t1   3                                           18                   20
 t4   7                                                                   23
 t5  10                                                                      30
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Physical Reg File.  All values in hex.  Free List: 18, 20, 23, 30, 37
 3  100                                           ]
 7  101
 10 102
 18 103                   [                         200                  ]
 20 104                      [        104   ]         [                202{]}
 23 105                         [        134]            [       {107}       {]}
 30 106                                                    [       {1c7}
 37 107
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

Problem 2, continued: The illustration below is similar to one used in class; it is provided for reference.

In the solution on the previous page incorrect entries are surrounded by braces, for example `107`. The numbers in parenthesis on the PED lines are the physical register numbers assigned to the instructions. They are there as an aid in solving the problem.

Because of the defect the ID map is not recovered after the branch misprediction. This causes two execution problems. First (most people got this) the `addi` on the correct path reads the wrong `t4`: it should read the one in physical register 7 but it reads the one in 20. As a result the values written in the physical register file are wrong. The second problem (few got this) is that when instructions pass through ID they read the wrong incumbent. For example, the `addi` on the correct path should read 7 as the incumbent, but instead it reads 20. As a result the wrong register will be placed in the free list when `addi` commits, which is what part (b) is about.



(*b*) Suppose 1000 instructions later an instruction finds `0x4720` in register `t4` despite the fact that the code above wrote something else and no instruction wrote `t4` after that. (If the free list is used improperly the question might apply to `t5`, not `t4`. See the diagram for hints on proper use of the free list.)

Note: On the original exam register `t5` *was used instead of* `t4`. *Register* `t5` *can not change unexpectedly, though it still has an incorrect value.*

☑ How could that have happened? Be specific in how it's due to the defect.

As explained above, the correct path `addi` reads the wrong incumbent and so frees the wrong physical register, in particular the physical register holding `t4`. At some point that physical register will be assigned to a new instruction which will write a value in it, perhaps `0x4720`.

(*c*) How could the defect have gone undetected?!? (That's not the question.)

☑ Suppose the `lw` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Remember, the only defect is with recovering the ID map on a branch misprediction. *Note: Original exam omitted "on a branch misprediction."*

The mechanism used to recover the ID map on a branch misprediction is different than the one used for exceptions. The problem stated that other than ID recovery on a branch misprediction, everything was working correctly. The `lw` and `lb` instructions execute before the `addi` and `ori` instructions. When the load reaches the ROB head exception recovery will start, part of that is copying the Commit Map to the ID Map, replacing the incorrect ID Map with a correct one. For that reason the code executes correctly.

☑ Now suppose the `lb` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Once again, the only defect is with recovering the ID map on a branch misprediction.

See the solution to the part above.

Problem 3: The code below runs on three systems: one using a bimodal (one-level) predictor, I; one using a two-level local history predictor, L; and one using a two-level gshare predictor, G. The global history register is 16 bits and the local history is 16 bits. The branch history tables have 256 entries. Assume that each branch below has its own BHT entry. The branch outcomes for B2 and B3 are provided for your solving convenience. Note that B3 has the same pattern as B2 but at a different phase, the phase difference is important. (11 pts)

```
BIGLOOP: LOOP:
 B1: bne $t1, $0, SKIP  # Random, independent, taken 25% of time.
 ...
 SKIP:
 B2: bne $t2, $t5, SKIP2 # Pattern: N N N N N N T T N N N N N N T T N N N N N N T T ....
 ...
 SKIP2:
 B3: bne $t7, $t8, SKIP3 # Pattern:  N N N N N T T N N N N N N T T N N N N N N T T N ....
 ...
 SKIP3:
 B4: bne $t9, $t10, LOOP # Iterates 50 times.
...
 j BIGLOOP
 nop
```

(*a*) Determine the prediction accuracy for each branch and each predictor. The accuracies should be after warmup. **Do not** compute the number of entries. Approximate the accuracy of B1 predictions.

☑ B1 on I: Accuracy: 70%

☑ B1 on L: Accuracy: 70%

☑ B1 on G: Accuracy: 70%

☑ B2 on I: Accuracy: 62.5%

☑ B2 on L: Accuracy: 100%

☑ B2 on G: Accuracy: 100%

☑ B3 on I: Accuracy: 62.5%

☑ B3 on L: Accuracy: 100%

☑ B3 on G: Accuracy: 85%

☑ B4 on I: Accuracy: 98%

☑ B4 on L: Accuracy: 98%

☑ B4 on G: Accuracy: 98%

Since branch B1 is taken 25% of the time the two-bit counter used to predict it will likely be 0 or 1 (with probability $\frac{36}{40}$), therefore the prediction accuracy of B1 is about 75% (it is exactly 70%, explained below). The accuracy is the same on all predictors because correlating the prediction with past occurrences of the same branch (local) or other branches (gshare) does not help predict the random branch.

The bimodal prediction accuracy for B2, B3, and B4 is straightforward.

Branch B2 and B3 follow a regular pattern that repeats every eight occurrences. This is small enough for the local predictor to achieve a 100% prediction accuracy. Each iteration of the loop has four branches, so the 16-bit GHR can "remember" four iterations. It so happens that B2 does whatever B3 did on the previous iteration. Gshare easily remembers one iteration back, so the prediction accuracy of gshare for B2 is 100%.

Determining the gshare prediction accuracy for B3 is more involved. The prediction accuracy depends upon how much of the repeating pattern the GHR holds. The pattern consists of four branches (B1-B4):

```
 B1: x   x   x   x   x   x   x   x    ...
 B2:  n   n   n   n   n   n   t   t   ...
 B3:   n   n   n   n   n   t   t   n  ...
 B4:    T   T   T   T   T   T   T   T ...
GHR: xnnTxnnTxnnTxnnTxnnTxntTxttTxtnT (repeats)
Iter: 0   1   2   3   4   5   6   7   0   1 ...
```

The **x** outcome for B1 can be either N or T, the outcomes for B2 and B3 are shown in lower case so they can be distinguished from B4 when looking at the GHR. The predictability of B3 depends on the iteration (see the diagram above). First consider iteration 0. The outcome is N and this outcome only occurs when the outcomes in the previous iteration (7) are **xtnT**. For iteration 0 the GHR needs to be five bits to see enough: **tnTxn** (the last **x** isn't needed). Similarly, iterations 6 and 7 can be predicted using only the previous iterations. Since the GHR is sixteen bits it is large enough to predict these iterations with 100% accuracy. Iterations 1 through 5 require more than five bits. Iteration 1 needs to see iteration 7, requiring 9 bits (which we have). For similar reasons iteration 2 needs 13 bits, iteration 3 needs 17 bits (bzzzt! [that's the GHR-is-too-small buzzer]), and iteration 4 needs 21 bits (bzzzt!). Iteration 5 also needs 21 bits (bzzzt!) since the GHR value at B3 for iterations 4 and 5 would be different. Since the GHR is only 16 bits the GHR contents will be identical when predicting B3 in iterations 3, 4, and 5. This would not be a problem if the outcome of B3 was the same in all these iterations (all taken or all not taken), but it's not taken in iterations 3 and 4 but taken in iteration 5. As a result the two-bit counter will mostly predict not taken but the branch will sometimes be taken. Because of the random branch there are really sixteen (four x's) two-bit counters for iterations 3-5 and they are accessed in random order. They will mostly be at zero or one. As a rough estimate, (which is enough for full credit) assume the counter is always zero or one, then the prediction accuracy of iterations 3 and 4 will be 100% and iteration 5 will be 0%. Taking into account that the counter is sometimes 2 and 3, (see below) we get a prediction accuracy of $\frac{12}{15} = 80\%$ for iterations 3 and 4 and a prediction accuracy of 20% for iteration 5.

The overall prediction accuracy for branch B3 is the average of each iteration: $\frac{1}{8}(5 * 100\% + 2 * 80\% + 20\%) = 85\%$. So gshare's accuracy on B2 is 85%.

To precisely determine the prediction accuracy on B1 (and also B2, and to a very minor extent, B4) one needs to compute the probability that the counter value is less than 2. In the other cases the counter value is not effected by the random branch so their exact value can be computed, usually 0 or 3. Consider B1 using the bimodal predictor. Because it's taken only 25% of the time the counter will mostly be less than two, but because outcomes are independent there is a chance the branch will be taken twice in a row, leaving the counter at 2 (or even 3). Computing these probabilities is straightforward, and uses a standard technique in queuing theory: Markov chains. A Markov chain is a mathematical model that can be used to represent the counter and other systems which can be represented by state transition diagrams with probabilities associated with arcs. There are four states, denote them s_0 (count is 0), s_1 (count is 1), etc. There is an arc from s_0 to s_1 and it is associated with the transition probability (branch taken probability), 0.25, which we will denote p. There are similar arcs from s_1 to s_2 and s_2 to s_3. There are also arcs from s_3 to s_2, etc. representing the not-taken cases, they are associated with the not taken probability, 0.75 or $1 - p$.

It turns out it is surprisingly easy to solve for the state probabilities. The key observation is that the number of times the counter is incremented from 0 to 1 (or 1 to 2, etc) must be equal to the number of times it is decremented from 1 to 0 (or 2 to 1, etc) (plus or minus 1, which is insignificant). Overloading the notation, let s_0 denote the probability of the counter being 0. Then $s_0 p = s_1(1-p)$ (what goes up must come down, or more formally, a balanced flow equation). Generalizing, $s_i p = s_{i+1}(1-p)$ for $i \in \{0, 1, 2\}$. With a few substitutions and some manipulation we find $s_i = s_0 \left(\frac{p}{1-p}\right)^i$. Obviously $\sum_{i=0}^{3} s_i = 1$ and so $\sum_{i=0}^{3} s_0 \left(\frac{p}{1-p}\right)^i = 1$. Solving for s_0 we get $s_0 = 1 \Big/ \left[\sum_{i=0}^{3} \left(\frac{p}{1-p}\right)^i\right]$. Since the summation is only over four terms one could compute s_0 at this point, but we don't have to since there is a closed-form expression for that summation: $\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$

and so

$$s_0 = \left[\frac{p}{1-p} - 1\right] \Big/ \left[\left(\frac{p}{1-p}\right)^4 - 1\right].$$

For branch B1 $p \to 0.25$ and so $s_0 = \frac{27}{40}$, $s_1 = \frac{9}{40}$, $s_2 = \frac{3}{40}$, and $s_3 = \frac{1}{40}$. The branch prediction accuracy for all predictors on B1 is exactly $\frac{3}{4}\frac{27+9}{40} + \frac{1}{4}\frac{3+1}{40} = 70\%$, reasonably close to the 75% estimate based on the assumption that the counter never reaches 2.

Grading Notes: Way too many people got B1 wrong, the most common wrong answer was 25%. That would only be correct if a taken prediction were made every time. (Yes, I understand that with more time ...) No one realized that 75% would be an upper bound on B1's prediction accuracy. Many had trouble with gshare, or decided not to spend time on it.

(*b*) A gshare predictor using a 15-bit GHR (instead of the 16-bit GHR used above) should give the same prediction accuracy for branch B2. How small can the GHR be made without changing the prediction accuracy of B2? Assume there are no collisions. *Note: The original question asked about B3.*

Branch B2 does whatever B3 did in the previous iteration, so the GHR can predict it at 100% accuracy with as few as three bits.

The original question asked about B3. Prediction accuracy is determined by how many previous branches the predictor can see. At four branches per iteration a 16 bit GHR holds four previous B3 outcomes. Reducing to 15 bits looses the oldest B3 outcome, however that information is in B2 because its outcome is the same as B3 in the previous iteration. Therefore the GHR can be reduced to 13 bits without changing predictor accuracy. (At 13 bits B2 is the oldest outcome.)

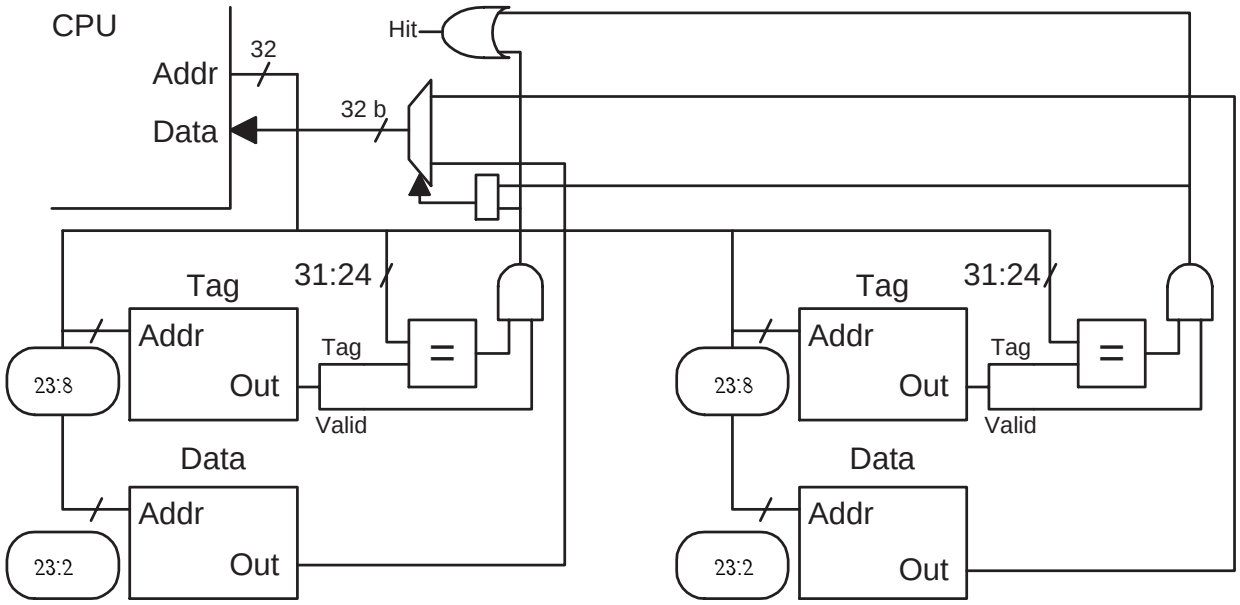
(*c*) For the local predictor, how small can the local history be made without affecting the prediction accuracy of B2 and B3?

To distinguish the first of the two taken branches from the last not-taken branch the local history must be at least six bits.

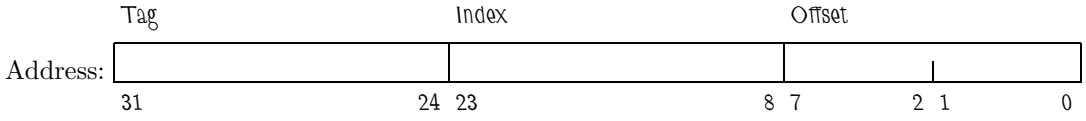
Problem 4: The diagram below is for a cache with 256-character lines on a system with 8-bit characters. (17 pts)

(*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: 2

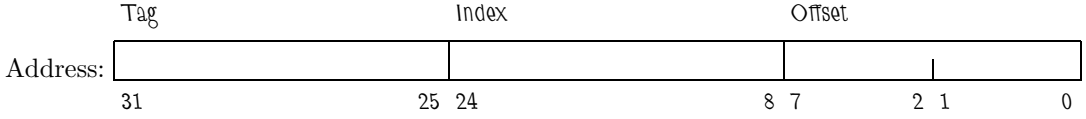
☑ Cache Capacity (Indicate Unit!):

It's $2 * 2^{24}$ characters (bytes in this case) or 32 MiB.

☑ Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 * 2^{16}(32 - 24 + 1)$ bits.

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



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.

```
int *a = 0x100000;  // sizeof(int) = 4 characters
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

There are $\frac{256}{4} = 64$ elements per line. During the first iteration the hit ratio is $\frac{63}{64}$, during the second it is 1 since the entire array can be cached. The total hit ratio is $\boxed{\frac{127}{128}}$.

(*c*) In the problem below, only consider accesses to the arrays.

```
int *a = 0x10000;    // sizeof(int) = 4 characters
int *b = 0x20000;
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ] + b[ i ];
```

☑ What is the minimum size of a direct mapped cache for which the program above will have a 100% hit ratio on the second j iteration? Explain.

The total size of the two arrays is 8192 characters but if the cache were 8192 characters the index bits for the two arrays would be the same and they could not be simultaneously cached. The only way for the two arrays to have different index bits is to make sure the index part of the address includes bit 16 (bit 16 of `0x10000` is 1, bit 16 of `0x20000` is zero). Such a cache would be $\boxed{2^{17} = 131,072 \text{ characters.}}$ What a difference!

☑ What is the minimum size of a two-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

In this case the cache can be made the combined size of the arrays, $\boxed{\text{8192 characters}}$. Corresponding elements (say, when `i=3`) have the same index but the cache can handle the two different tags.

☑ What is the minimum size of a three-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

Again, waste. Why couldn't it be a four-way? One third of the cache goes to waste, the size is $\boxed{3 * 4096}$. Note that if it were a four way the size could be 8192 characters.

Problem 5: Answer each question below.

(*a*) In the PED below for a statically scheduled 2-way superscalar machine the `xor` instruction stalls in IF even though there is an empty space in ID. (5 pts)

```
add s1, s2, s3    IF ID EX ME WB
or  s4, s1, s5    IF ID -> EX ME WB
xor s6, s7, s8       IF -> ID EX ME WB
and s9, s10, s11     IF -> ID EX ME WB
```

☑ Why?

If `xor` did move in to ID then the instructions in ID would be out of order and ordinarily the decode logic is designed for in-order instructions, so it couldn't handle them.

☑ Would it be difficult or would it be impossible to modify the implementation (but keeping it statically scheduled) so that such an instruction could move to ID? Explain.

Tedious but not impossible. Logic would have to consider case where instruction in slot zero follows instruction in slot one.

(*b*) In a dynamically scheduled system why should store instructions wait until they are ready to commit before storing the data? (6 pts)

Because it might be squashed, and if it wrote before being squashed there would be no way easy to recovery the previous memory contents.

(*c*) The same code fragment executes on 1-way, statically scheduled systems with the specified FP add functional unit(s). For each show a pipeline execution diagram. Don't forget to consider *all* structural hazards and check carefully for dependencies. All adders take a total of four cycles to compute a result (latency is 3). (5 pts)

☑ One adder, `A`, initiation interval: 1.

```
# Solution
add.d f0, f2, f4    IF ID A1 A2 A3 A4 WB
add.d f6, f0, f8       IF ID -------> A1 A2 A3 A4 WB
add.d f10, f0, f12        IF -------> ID A1 A2 A3 A4 WB
```

☑ One adder, `A`, initiation interval: 2.

```
# Solution
add.d f0, f2, f4    IF ID A1 A1 A2 A2 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A2 A2 WB
add.d f10, f0, f12        IF -------> ID -> A1 A1 A2 A2 WB
```

☑ One adder, `A`, initiation interval: 4.

```
add.d f0, f2, f4    IF ID A1 A1 A1 A1 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A1 A1 WB
add.d f10, f0, f12        IF -------> ID -------> A1 A1 A1 A1 WB
```

☑ Two adders, `A` and `B`, each has initiation interval: 4.

```
add.d f0, f2, f4    IF ID A1 A1 A1 A1 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A1 A1 WB
add.d f10, f0, f12        IF -------> ID B1 B1 B1 B1 WB
```

(*d*) One problem in Homework 5 was to analyze the execution of an unoptimized program to compute π. Many people got the question below wrong, here's your chance to get it right! (5 pts)

☑ Should computer engineers analyze the performance of processors running unoptimized code? Explain.

They should analyze the kinds of programs that people run, which is optimized. Software is compiled without optimization only for special circumstances such as debugging.

(*e*) Consider a new data type, binary coded trianary. This 32-bit data type consists of 16 radix-3 digits, each coded with two bits, in the same way a BCD data type would consist of 8 radix-10 digits. (8 pts)

☑ Why might 3-fingered, three-armed aliens (one hand per arm) find this data type useful?

For the same reason we've adopted a radix-10 number system they might adopt a radix-9 or radix-3 number system. Binary-coded trianary would have the same advantage for them.

☑ Explain how the data type would be useful in accessing arrays in which the element size was a power of three.

Suppose the size were $3^3 = 81$ and we wanted to access element 21. Instead of multiplying by $21 * 81$ we would shift the trianary 21 by six bits (three trianary digits). Either we would use a new machine instruction to convert it to binary or the array would be 25% padding.

Grading Note: A common incorrect answer was that it would be easy to increment an index. That's easy anyway since it would take one cycle to add 81 (incrementing the address by one element).

In a meeting next week you plan to argue that this data type should be included in the next revision of the ISA, one of several proposed new data types. Only one will be chosen. *Note: The following sentence was not in the original exam.* Three-fingered aliens are **not** expected to make up a large portion of your customer base.

☑ How will you argue that the data type should be included?

First, show that multiplying by a power of three and computing modulo 3 (or a power) would be only a single cycle with the new data type, versus many cycles (especially for modulo now). Then argue that existing software performs enough modulo-3 arithmetic to benefit from the extension (if recompiled).

Grading Node: Many answers involved aliens. They were fun to read.

☑ What will you do in the next week to prepare your argument?

Analyze existing code to find how often they perform power-of-three multiplication and modulo operations.

The following incorrect answer was common: write example programs to demonstrate the data types usefulness. It's wrong because the example programs don't tell us how often the data type would be used in real programs, which is very important. A sample program should not be necessary to demonstrate that a modulo 3 operation can be done in 1 cycle rather than 15 (it would be argued by the design of the arithmetic unit, in this case comparing an existing division/modulo unit to a shifter or logic unit [for masking to do the modulo]).

(*f*) In Homework 6 the performance of a linear search of an array and linked list were compared. On the dynamically scheduled systems the search was much faster on the array. Now consider the array program on a statically scheduled system. (8 pts)

```
\scoreable Would the array program perform as well compared
it the linked list program?  Consider
the effect of memory latency and line size.\par

\solution{The array program would still perform better than the linked
list program, but with longer memory latency or shorter line size the
array program on the dynamically scheduled system would run
substantially faster.  }
```

☑ Explain. *Hint: the solution above was intentionally included.*

The array program does better because a single miss brings in multiple lines. Both statically and dynamically scheduled systems enjoy this benefit. If the line size is short or memory latency is large the dynamically scheduled system could have misses to more than one line at once, overlapping two time consuming memory accesses. This is something the statically scheduled processor could not do.

(*g*) Predicated instructions can be used to avoid branches. (8 pts)

☑ Why do predicated instructions have maximum benefit over branches that skip over one instruction (compared to predicated instructions used to avoid branches that skip over more than one instruction)?

Because of the overhead needed for branches, especially in superscalar machines. Even on a one-way machine the branch overhead is worse than sometimes wasting time by executing a predicated instruction with a false predicate. As more instructions are skipped over by the branch, the benefit of predication drops since the branch overhead remains fixed while the waste of executing predicated instructions increases.

☑ Suppose the compiler is considering whether to replace a branch that skips n instructions with predicated instructions on a dynamically scheduled machine. How could the compiler use an estimate of prediction accuracy, f_{correct}, and resolution time, t_{resolve}, as well as implementation details to make its decision?

The performance potential lost due to mispredicted branches is proportional to the resolution time (the time needed to determine the branch condition). If a branch has a low prediction accuracy and a high resolution time then it will be responsible for squashing many instructions, and so the waste of predication is a better alternative. The higher $t_{\text{resolve}}(1 - f_{\text{correct}})$ the more instructions (higher n) one could predicate.