Name Solution

Computer Architecture
EE 4720
Final Examination

4 May 2015, 10:00–12:00 CDT

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 I feel like this is a good alias.

Good Luck!

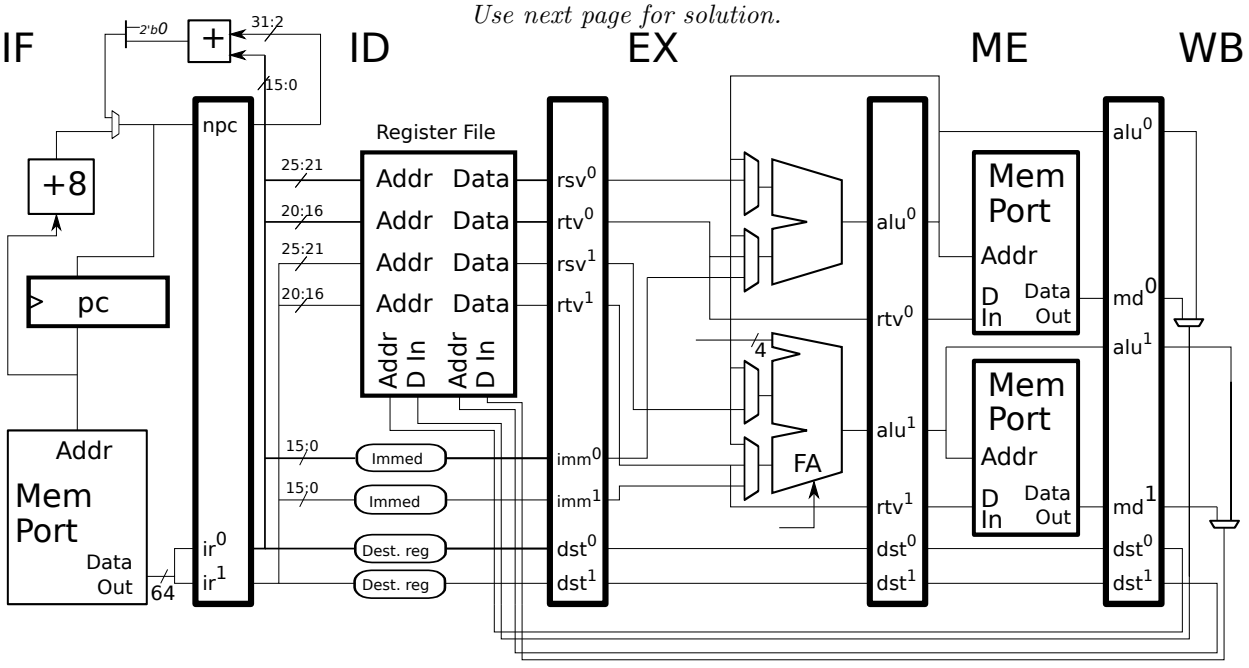
Problem 1: (20 pts) The instruction sequence below performs the calculation `r2+3+r5` and requires two MIPS instructions to do so. If these two instructions were in the same fetch group in a two-way superscalar processor then the second one would stall, this is shown in the execution below.

```
addi r1, r2, 3   IF ID EX ME WB
add  r4, r1, r5  IF ID -> EX ME WB
```

An *FA-ALU* has two regular inputs, and a third input reserved for small quantities, here for values < 16, and it can produce a result in the same time as an ordinary two-input ALU. An FA-ALU can be used to execute instruction pairs like the one above without a stall, call such an execution a *fused add*.

Illustrated below is a 2-way superscalar implementation with an FA-ALU. Most bypass paths have been removed to provide space for the solution to this problem. In addition to a 4-bit input for the third operand, the FA-ALU has a control input, `FA`, which should be set to `1` when the third input is to be used.

Use next page for solution.



Use next page for solution.

(*a*) Add datapath connections to this implementation so that the FA-ALU can be used for fused adds. *Hint: The datapath changes are not just for that new FA-ALU input.*

☑ Add datapath so that the correct operands are delivered to the FA-ALU for fused adds.

☑ Non-fusable sequences must still execute correctly. (Don't break existing functionality.)

☑ Make reasonable cost and clock frequency tradeoffs.

(*b*) Add control logic.

☑ Design logic to generate the `FA` signal for the FA-ALU and deliver it in the correct cycle.

☑ Consider ☑ instruction type, ☑ dependencies, and ☑ operand size.

☑ Add control signals for the datapath added in the previous part.

☑ Make reasonable cost and clock frequency tradeoffs.

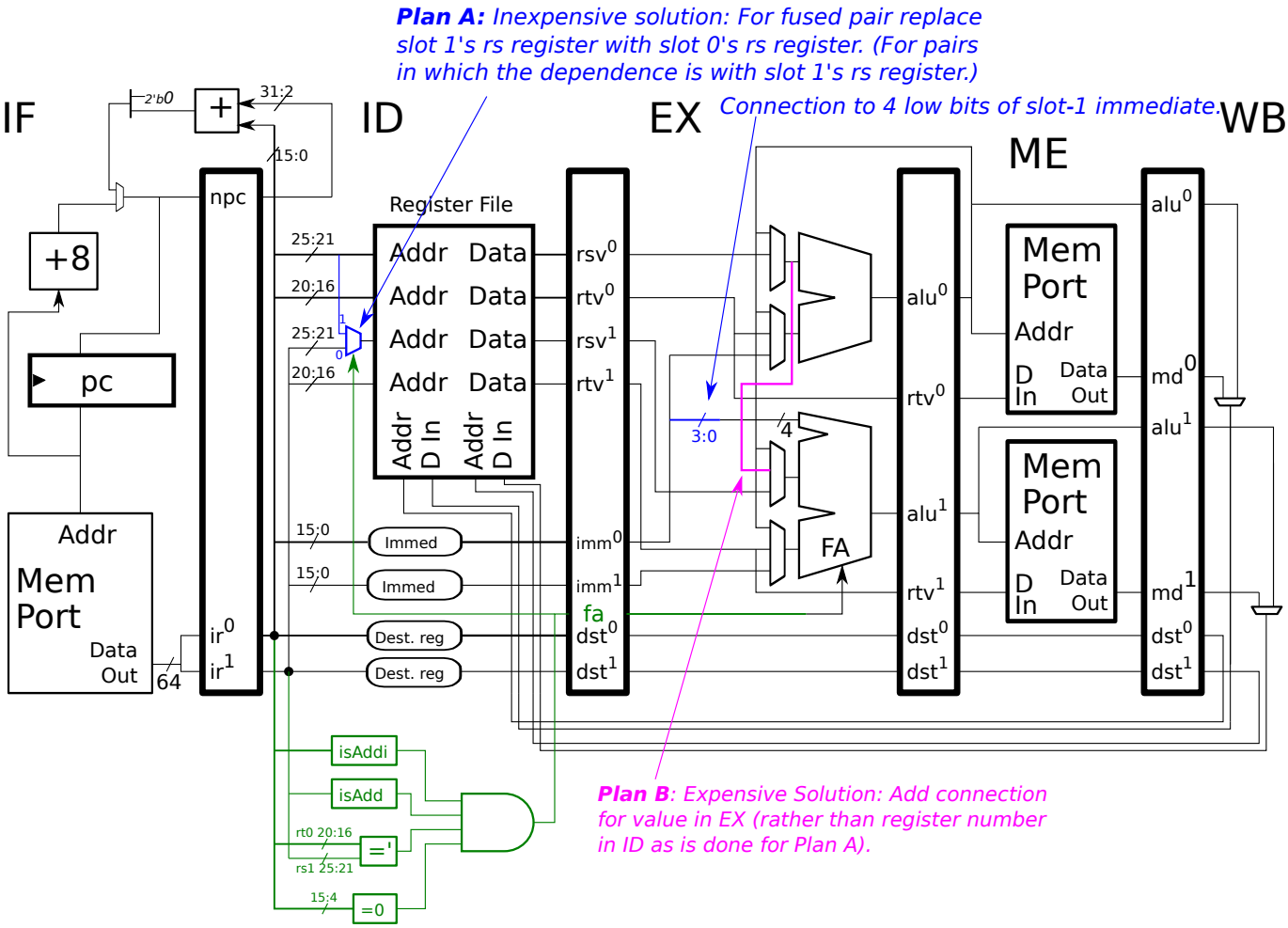
Problem 1, continued: You can use logic blocks such as isADD and isADDI in your solution.

```
addi r1, r2, 3    # Sample fusable sequence.
add  r4, r1, r5
```

Solution appears below. The datapath is in blue and control logic is in green, and an alternative solution appears in purple.

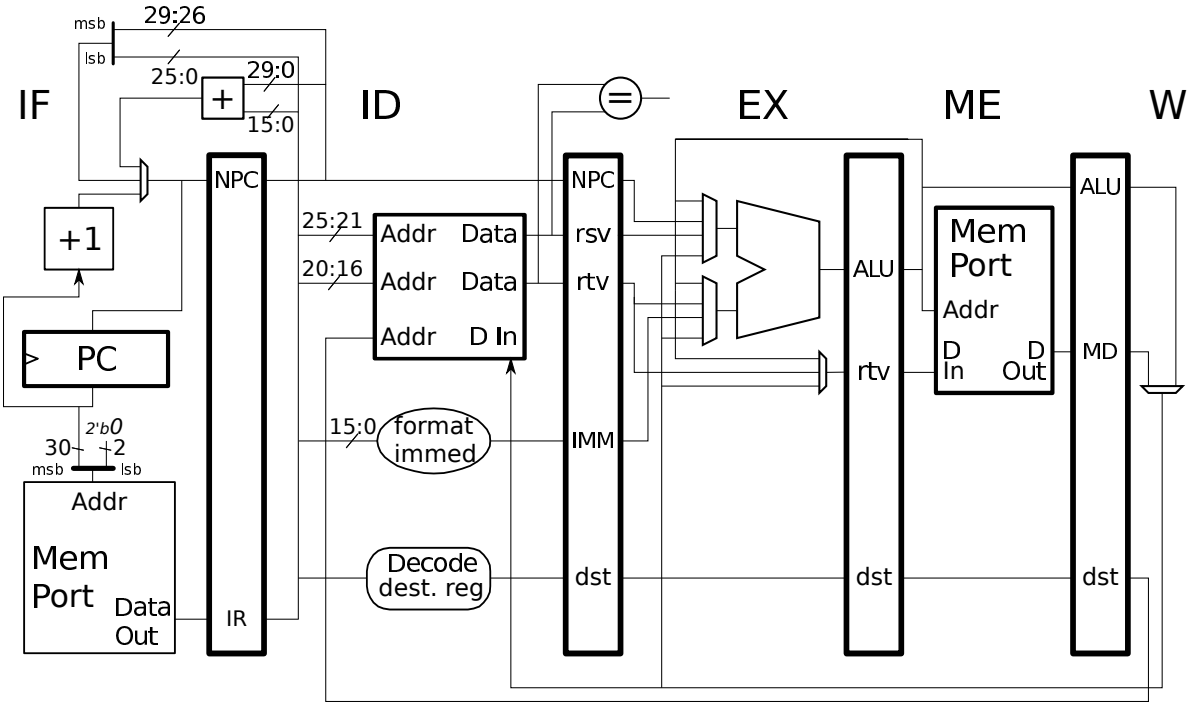
The datapath needs to be changed so that the immediate and `rs` register value from the slot 0 (`addi`) instruction are fed to the slot 1 FA-ALU. Connecting the immediate is straightforward. Two solutions are shown for the `rs` register value. In the Plan A solution the slot 1 `rs` register number is switched to the slot 0 `rs` register number. That requires just a 5-bit mux. In the not-as-good Plan B solution the value is moved, require an additional 32-bit mux input. Notice that the value is taken at the output of the upper ALU mux, this way a bypassed value can be used. That's possible in the Plan A solution is the output of the ID-stage mux is used for the bypass control logic (which isn't shown).

The control logic, in green, needs to check for three things: the presence of the two instructions, whether there is a dependence, and whether the immediate will fit in four bits. The solution only works for the arrangement of instructions shown above, and that's all that was needed for full credit. In real life one might want this trick to work for similar sequences, for example if the dependence were carried into the `rt` rather than the `rs` register.



Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken.

(*a*) Show the execution of the code below on the following implementation. The branch is taken.



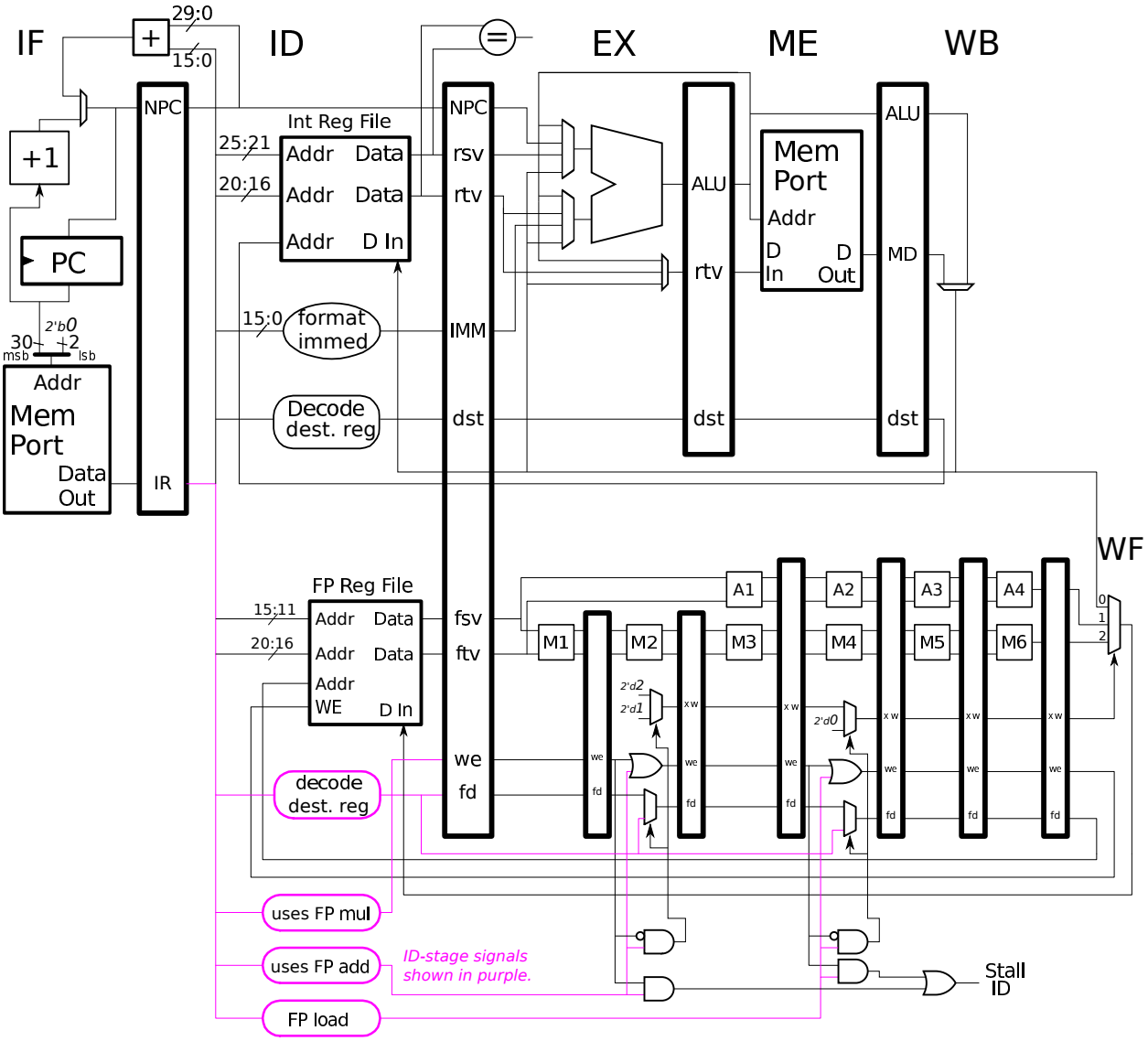
☑ Show execution. ☑ Doublecheck for dependencies.

Solution appears below.

Common mistakes: squashing the delay slot instruction (the `and`); not noticing that the branch depends on the `sub`.

```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
 add r4, r2, r3      IF ID EX ME WB
 lw r6, 8(r4)           IF ID EX ME WB
 sub r1, r6, r5            IF ID -> EX ME WB
 beq r7, r1 TARG              IF -> ID ----> EX ME WB
 and r8, r7, r10                    IF ----> ID EX ME WB
 or  r11, r12, r13
 xor r14, r11, r8
TARG:
 sw r1, 0(r2)                                  IF ID EX ME WB
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
```

(*b*) Show the execution of the MIPS FP code below on the following implementation. Assume that **there is** a bypass from `WF` to `M1` and `A1`.

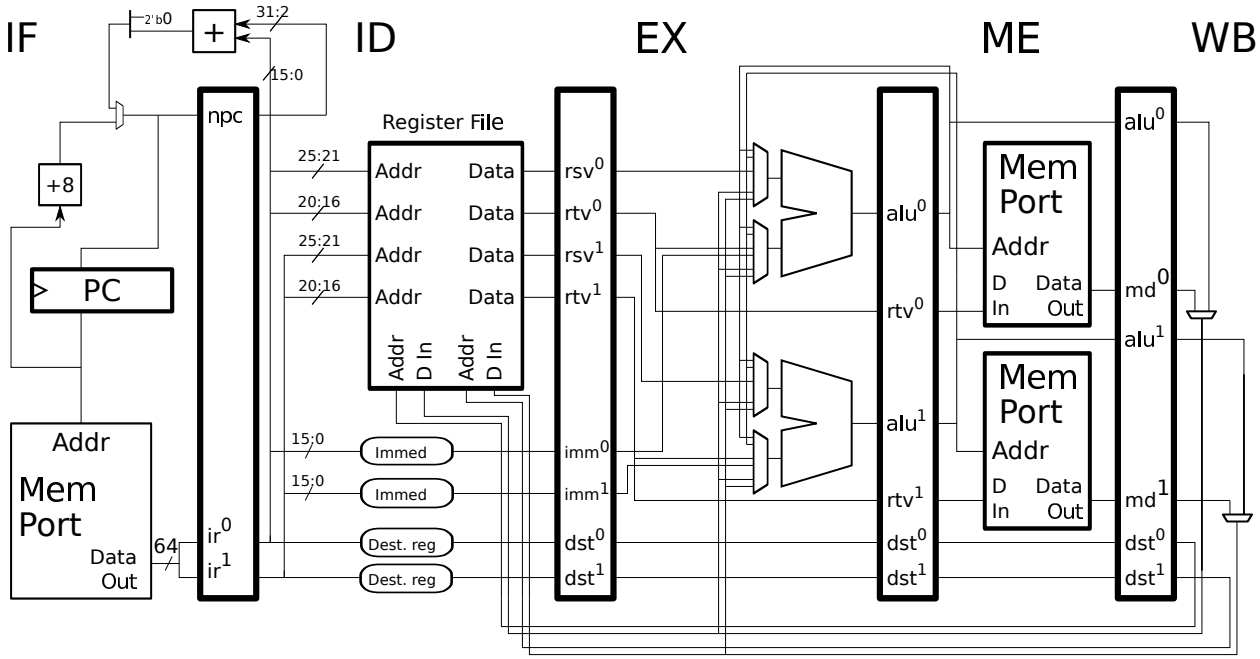


☑ Show code execution. ☑ Doublecheck for dependencies.

Solution appears below. Common mistakes: Not realizing that the `lwc1` uses `WF` (not `WB`) and therefore it must stall to avoid the structural hazard. Another common mistake is stalling in a stage other than `ID`.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
add.s  f6, f2, f3  IF ID A1 A2 A3 A4 WF
add.s  f1, f7, f8     IF ID A1 A2 A3 A4 WF
lwc1   f9, 0(r10)        IF ID ----> EX ME WF
mul.s  f4, f1, f9           IF ----> ID -> M1 M2 M3 M4 M5 M6 WF
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 n12 13 14
```

(*c*) Show the execution of the MIPS code below on the following 2-way superscalar implementation. The branch is taken. Fetch groups in this implementation must be 8-byte-aligned.



☑ Show code execution. ☑ Show squashed instructions with an `x`.

☑ Doublecheck for dependencies. ☑ Account for aligned fetch groups.

Solution appears below. Note that the `sll` is fetched only because the fetch unit can only fetch an address that's a multiple of 8, but it is squashed as soon as it arrives.

```
START: Instruction address of add is 0x1000
 # Cycle             0  1  2  3  4  5  6  7  8  9
 add r4, r2, r3      IF ID EX ME WB
 lw r6, 8(r4)        IF ID -> EX ME WB
 sub r1, r6, r5         IF -> ID -> EX ME WB
 beq r7, r15 TARG       IF -> ID -> EX ME WB
 and r8, r7, r10              IF -> ID EX ME WB
 or  r11, r12, r13            IFx
 xor r14, r11, r8

 sll r16, r17, 8                    IFx
TARG: Instruction address of sw is 0x2004
 sw r1, 0(r2)                       IF ID -> EX ME WB
 lui r15, 0x1234                       IF -> ID EX ME
 # Cycle             0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{30} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(*a*) Branch behavior is shown below. Branch B2 consists of a pattern in which there will be either 2, 4, or 6 T outcomes followed by exactly four N outcomes. After the fourth N the probability of exactly two Ts is $\frac{1}{3}$ and the probability of exactly four Ts is $\frac{1}{3}$.

Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  N    T    T    N    N    N      N    T    T    N    N    N   ...
B2:   T{2,4,6}  N    N    N    N      T{2,4,6}  N    N    N    N  ...
B3:    T    T    T    T    T    T      T    T    T    T    T    T ...
```

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

The accuracy is $\boxed{\frac{3}{6}}$. The work to compute this result is shown below, in which the counter was assumed to start at 3. (Any assumed start value is correct.) The prediction accuracy is based on the second pattern repetition because the counter value at the start of the pattern and the end of that pattern is the same, 0. (In contrast, the counter value at the start of the first repetition is 3 and at the end it is 0, so we can't use the first six outcomes of B1 to compute an accuracy, though by coincidence it would be correct.)

```
#  SOLUTION Work
   3   2    3    3    2    1    0      0    1    2    1    0    0 <- Counter values.
B1:  N    T    T    N    N    N      N    T    T    N    N    N   ...
     x              x    x              x    x    x              <- Mispred location.
                                      ----------------------------  <- Repeating pattern.
```

☑ What is the accuracy of the bimodal predictor on branch B2? ☑ Explain.

To answer this question consider each number of Ts separately. First, TTNNNN. When this pattern starts the counter will be zero (because every repetition ends with four Ns). So the two Ts will be mispredicted, bringing the counter to 2, and the first N will be mispredicted. The number of correct predictions is $6 - 3 = 3$ and the accuracy is $\frac{1}{2}$. Next consider TTTTNNNN. Here the first two Ts will be mispredicted as will the first two Ns. The number of correct predictions is $8 - 4 = 4$ and the accuracy is also $\frac{1}{2}$. Finally, in TTTTTTNNNN the first two Ts and Ns will be mispredicted. The number of correct predictions is $10 - 4 = 6$ and the accuracy is .6.

For an overall prediction accuracy we need to combine these numbers. Since they are all equally likely we will consider one pattern of each length, 6, 8, and 10 together as a group. The correct prediction $\boxed{\text{ratio is } \frac{3+4+6}{6+8+10} = \frac{13}{24} = .541667}$. If we wanted to take a weighted average of the prediction accuracies we would have to weight by both the probability of occurrence and the length. In other words if we computed $\frac{1}{3}\frac{3}{6} + \frac{1}{3}\frac{4}{8} + \frac{1}{3}\frac{6}{10} = \frac{8}{15}$ we would have given the $\frac{6}{10}$ accuracy too low a weight. Yes, it happens one third the time, but its length, 10 outcomes, is longer than the other two so it has a larger overall impact on execution time.

☑ What is the minimum local history size needed to predict B1 with 100% accuracy?

$\boxed{\text{Four outcomes}}$ are sufficient. With three outcomes a local history of NNN could appear before an N or a T.

☑ What is the accuracy of the local predictor on branch B2, after warmup. ☑ Explain.

Because there is randomness the local predictor won't be right 100% of the time. Notice that we always have four consecutive N's followed by two T's. That means if the most-recent four outcomes in the local history are N then a T is certainly the next outcome (and so the corresponding PHT entry will have warmed up to 3). Similarly, if the most recent PHT outcomes are a T followed by one, two, or three Ns the next outcome is certainly an N. The table below shows some possible local history values, the next outcome, and how often that outcome will be encountered. Stars indicate either a T or N.

Local History		Next Outcome Frequency
**TTN	N	Every Time
**TTNN	N	Every Time
**TTNNN	N	Every Time
**TTNNNN	T	Every Time
**TTNNNNT	T	Every Time
**TTNNNNTT	T	2/3 of the time
**TTNNNNTT	N	1/3 of the time
**TTNNNNTTT	T	Every Time
**TTNNNNTTTT	T	1/2 of the time.
**TTNNNNTTTT	N	1/2 of the time.
TTNNNNTTTTTT	N	Every time.

First consider TTNNNN. The Ts will always be predicted correctly because at least two Ts always follow the four Ns. The first N can be mispredicted because $\frac{2}{3}$ of the time the run of Ts is larger than 2. First, lets assume that the N is *always* mispredicted. Later, in an advanced solution, we'll use a better misprediction probability.

The following three Ns will be correctly predicted. So for the first pattern we have 5 correct predictions and one misprediction.

Next, consider TTTTNNNN. The first four Ts will likely be predicted correctly because after the first two Ts a third T is more likely than an N. Given that we have four Ts, the probability of a fifth one is $\frac{1}{2}$. Since the bias is even the 2-bit counter can predict either direction with equal likelihood. So the first N is mispredicted half the time for this pattern, and similarly the fifth T is mispredicted half the time for the 6 T pattern. Considering a set of all three patterns, the expected total number of mispredicts is just 2, so the accuracy is $\frac{(5+7.5+9.5)}{(6+8+10)} = \frac{11}{12} = .91666$.

Advanced Solution: The solution above ignores the fact that for pattern TTNNNN there is a chance that the first N is correctly predicted. That would occur if the local predictor saw at least two consecutive TTNNNN patterns before predicting a third consecutive TTNNNN. Here *saw* means that the local histories were the name for all three patterns when predicting the first N. A simple Markov chain can be used to determine the probability of the counter value for local history *NNNNTT (the local history used when predicting the first N). Let p_0 denote the probability that the counter value is 0, p_1 denote the probability that the counter value is 1, and so on. The problem states that the T probability is $2/3$ in this case. Based on this we can write a *balanced flow* equation: $ap_0 = (1-a)p_1$, where a is the T probability. Solving we get $p_1 = \frac{a}{1-a}p_0$. Between p_1 and p_2 we can write $ap_1 = (1-a)p_2$ and get $p_2 = \left(\frac{a}{1-a}\right)^2 p_0$. Skipping a few steps we get $p_0 = \frac{\omega-1}{\omega^4-1}$ where $\omega = \frac{a}{1-a}$. Using $a = \frac{2}{3}$ we find $p_0 = \frac{1}{15}$ and $p_1 = \frac{2}{15}$ and so the probability of correctly predicting the first N for the T2 case is $\frac{3}{15} = \frac{1}{5}$. The number of correct predictions for T2 is $5 + \frac{1}{5}$ or an accuracy of $\frac{13}{15} = .8667$. Similarly, the number for T4 is $6 + \frac{4}{5} + \frac{1}{2}$ or an accuracy of $\frac{73}{80} = .9125$ and for T6 is $8 + \frac{4}{5} + \frac{1}{2}$ or an accuracy of $\frac{93}{100}$. The overall correct prediction accuracy is $\frac{109}{120} = .9083$.

☑ What is the minimum GHR size to predict B1 with about 100% accuracy. ☑ Explain, and state any assumption about B2's behavior.

Based on the answer to the minimum-local-history-size question above, the minimum local history needed to predict B1 with 100% accuracy is four outcomes. But here we are dealing with a global history, which contains the outcomes of the most recently executed branches, which in this case is B1, B2, and B3. In the illustration below the outcomes of B1 have been made lower case, and the value of a six-outcome GHR is shown at time X (see the diagram) when we are about to predict B1.

```
Time->                  X
B1:  n    t    t    n    n    n      n    t    t    n    n    n   ...
B2:   T{2,4,6}  N    N    N    N      T{2,4,6}  N    N    N    N  ...
B3:    T    T    T    T    T    T      T    T    T    T    T    T ...
              ----------X
               tNT  nNT     <--- Value of 6-outcome GHR at time X.
```

Notice that this six-outcome GHR includes two outcomes of B1. Since we need *four* outcomes of B1 to predict with 100% accuracy we would need $\boxed{\text{a GHR size of } 4 \times 3 = 12}$ for the global predictor to predict B1 with 100% accuracy.

The analysis above assumes that there is always one occurrence of B2 between B1 and B3. The diagram makes that clear for the N outcomes of B2 but it's not clear for the T outcomes. In other words, from the diagram above one might conclude that there are 2, 4, or 6 T outcomes between B1 and B3. Since we were *invited* to make an assumption, lets make the easy one: that in all cases, N and T B2 outcomes, there is exactly one occurrence of B2 between B1 and B3.

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries.

Branch `B1` below is perfectly biased not taken. It executes r_1 times over a short time interval, followed by a big time gap before its next bunch of r_1 executions. Branch `B2` behaves similarly, except that it is biased taken and executes in bunches of length r_2 executions.

```
0xee4720  B1:   N..N r1                          N..N r1
# ...
          B2:             T..T r2                          T..T r2
```

(*b*) Choose an address for branch `B2` that will result in a BHT collision with branch `B1`. (Branch `B1` is at address `0xee4720`.)

☑ Address for `B2` that results in a collision.

Branch `B1` will collide with `B3` in a BHT lookup if the bit values used to index the BHT for the two branches are the same. Since the BHT has 2^{14} entries and because MIPS instructions are 4-byte aligned we will use bits 15:2 of the branch address to index (to use as a lookup address for) the BHT. The value of those bits for `B1` is `0x4720 >> 2` or `0x11c8` but for simplicity think of the bits as `0x4720`. (Note that each hex digit spans four bits). For `B2` to use the same entry the four least significant hex digits must match. One such address is `0xaa4720`.

(*c*) Assume that branch `B1` and `B2` collide in the BHT. For what positive values of r_1 and r_2 will this collision be most harmful? For what range of positive values of r_1 and r_2 will collisions be least harmful?

☑ Worst values of r_1 and r_2 for collision. ☑ Explain.

The worst values are $r_1 = 1$ and $r_2 = 1$. For those values the predictor will see one occurrence of `B1` followed by one of `B2`, then one of `B1` and so on. The 2-bit counter will not saturate at any value. In the worst case all predictions will be wrong.

☑ Favorable range of positive values of r_1 and r_2 for collision. ☑ Explain.

Large values for r_1, r_2 or both. When execution switches from one branch to the other there will be two mispredictions (assuming both $r_1 \geq 2$ and $r_2 \geq 2$). The accuracy would be $\frac{r_1-2+r_2-2}{r_1+r_2}$, so with the sum large the accuracy will be close to 100%.

(*d*) A tag can be placed in the BHT to detect collisions. Which branch will this help?

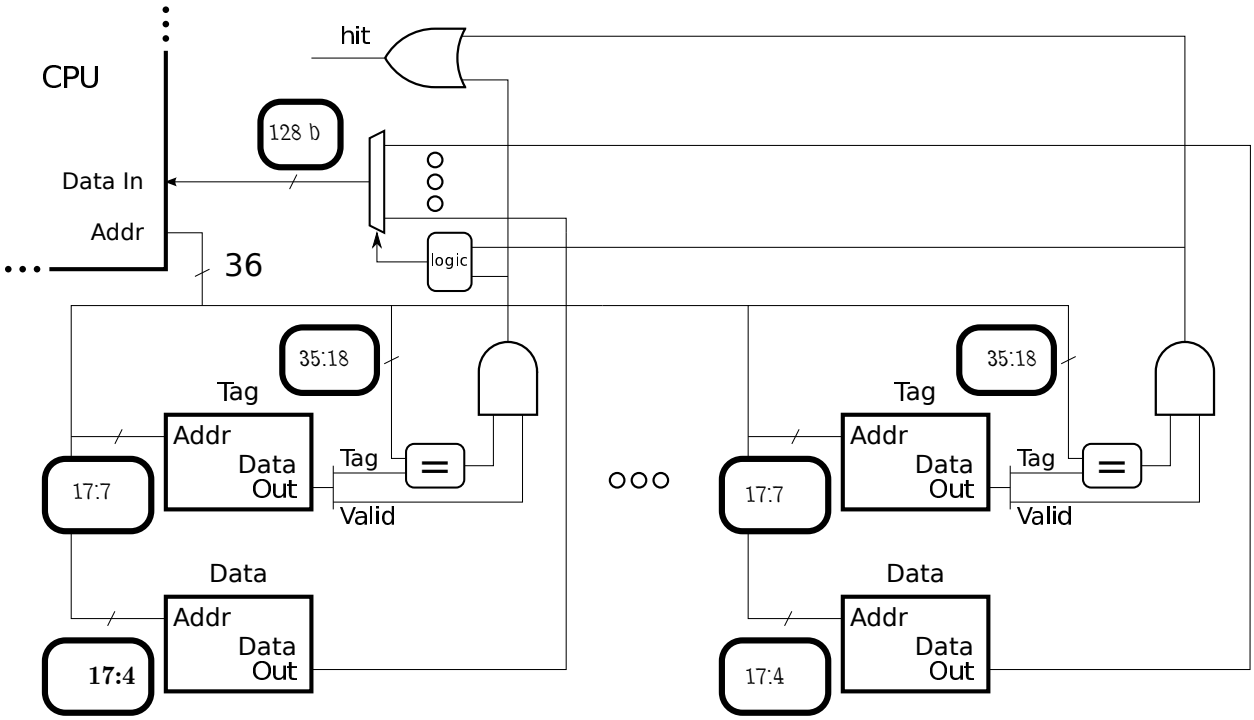
☑ Branch that would benefit from tag: `B1` or `B2`. ☑ Explain.

Branch `B1`. If there is a collision when predicting `B1` we will predict `B1` taken (because of `B2`'s history). A branch predictor that does not use a tag has no way of knowing that there's been a collision and so when predicting `B1` it will jump to the target of `B2` (not the target of `B1`). In the predictor with a tag the collision will be detected (by comparing the stored tag to the tag of the address of the branch being predicted). Since a collision means that the target address is wrong, the predictor will ignore the two-bit counter (which would be 3 in this example) and instead predict not taken, which has a chance of being correct. (When a collision has been detected you wouldn't predict taken because the target address is for some other branch sharing the entry, and so it won't be correct.) Predicting not taken for `B2` is wrong, but that's what we'd do with or without a tag.

Problem 4: (20 pts) The diagram below is for a 4 MiB (2^{22} B) set-associative cache with a line size of 128 bytes. Hints about the cache are provided in the diagram.

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

☑ Fill in the blanks in the diagram.



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

☑ Associativity:

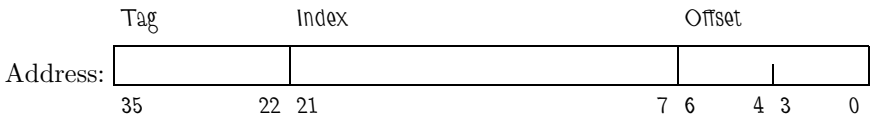
The associativity is 16. The associativity is determined based on the given cache capacity, 2^{22} bytes, and the capacity of an individual data store, 2^{18} bytes. Since the cache capacity is the sum of the data store sizes, the associativity must be $\frac{2^{22}}{2^{18}} = 2^{22-18} = 16$.

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

It's the cache capacity, 4 MiB, plus $16 \times 2^{18-7}\,(36 - 18 + 1)$ bits.

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

In a direct-mapped there is just one data store, which here must be 2^{22} bits, and so the low tag bit position must be $18 + 4 = 22$. The other bit positions remain the same.



Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 64 bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
half *a = 0x2000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

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

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^6 = 64$ bytes is given. The size of an array element, which is of type half, is $2 = 2^1$ characters, and so there are $2^6/2 = 2^{6-1} = 2^5 = 32$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^5 elements, and so the next $2^5 - 1 = 31$ accesses will be to data on the line, hits. The access at `i=32` will miss and the process will repeat. Therefore the hit ratio is $\frac{31}{32}$.

(*c*) The code appearing below is the same as the code from the 2014 Final Exam and 2015 Homework 6, except that two possibilities for `Some_Struct` are shown, Plan A, and Plan B. The caches too are identical except that here the line size is 64 bytes whereas in the prior problems it was 128 bytes. The value of `BSIZE` has been chosen to the largest value for which the hit ratio of the second loop will be 100% when using the Plan B struct. *Note: In the original exam* `norm_val` *was omitted from the first loop.*

```
struct Some_Struct { // Plan A
  double val;        // sizeof(double) = 8
  double norm_val;
  double a1[14];   };

struct Some_Struct {  // Plan B
  double val;         // sizeof(double) = 8
  double a1[7];
  double norm_val;
  double a2[7];    };

  const int BSIZE = 1 << 15;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;  // First loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second loop.
```

☑ Explain why the Plan A struct will result in better performance on the first loop than the Plan B struct.

Short Answer: With Plan A a miss to `b[i].val` also brings in `b[i].norm_val`, with Plan B there would be two misses (per iteration) and so Plan A is faster.

Detailed Answer: Notice that the first loop accesses members `val` and `norm_val`. In the Plan A struct these members are close to each other, and so the line containing `val` also contains `norm_val`. Each iteration of the first loop misses on the access to `val` and because they are close together hits on the access to `norm_val`. The `val` member is at the beginning of the Plan B struct, but `norm_val` is $8 \times 8 = 64\,\mathrm{B}$ from the beginning. A miss that brings in `b[i].val` will not bring in `b[i].norm_val` because they both can't fit on the same 64 B line. Both the Plan A struct and Plan B struct are $16 \times 8 = 128\,\mathrm{B}$. When using the Plan A struct the first loop will encounter `BSIZE` misses, but when using the Plan B struct there will be `2*BSIZE` misses, and so better performance is achieved with the Plan A struct.

☑ Suppose that the value of `BSIZE` is to be chosen based on the Plan A struct, to a maximum value for which the second loop enjoys a 100% hit ratio. Explain why that value is the same as for Plan B.

Short Answer: Because with Plan A the least significant index bit is always zero or always one and so half the cache is used.

As explained in the previous answer, when using the Plan A struct we only need one line per iteration, whereas with the Plan B struct we need two lines per iteration. So you'd think that with Plan B we'd need to make `BSIZE` half the size to avoid cache misses. But no.

FINISH

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

(*a*) Which kind of implementations were RISC ISAs originally designed to simplify? Explain how a RISC ISA feature achieves this goal.

☑ Type of implementation RISC designed to simplify.

Pipelined implementations.

☑ Choose a feature and ☑ explain how it achieves the goal.

One RISC feature is that the arguments of arithmetic instructions must be either immediates or registers, they cannot come from or be written to memory. If source and destination arguments could come from memory then either there would need to be a memory port before and after the `EX` stage, which would be expensive, or such instructions would have to pass through the pipeline multiple times, stalling prior instructions and complicating the design.

(*b*) What is the difference between a trap instruction and an instruction that raises an exception? Give an example of how each is used in a system with bug-free code.

☑ Difference between trap and exception.

A trap always causes an interrupt, that's its purpose, whereas in an instruction that raised an exception something went wrong.

☑ Example of what trap instruction used for.

A trap instruction can be used by user code to request a service from an operating system, for example, to read data from a file. In a typical system user code cannot itself, for example, read data from a file since it is not allowed to access those memory location that control the disk drive. It is not allowed access so that filesystem access policies can be enforced, for example.

☑ Example of what instruction exception used for.

Load and store instructions raise exceptions when they attempt to accesses invalid memory addresses, which might happen in buggy code. In bug-free code load and store instructions can still raise exceptions. This would happen for cases where the OS needs to adjust the memory system. For example, a load instruction encountering a TLB miss will raise an exception. The OS (acting as the exception handler) will read the page tables, update the TLB, and re-start the load.

Problem 5, continued:

(*c*) Consider a $5n$-stage scalar implementation and an n-way superscalar implementation, both derived from our 5-stage design with the goal of improving performance by as much as $n\times$. As n increases the clock frequency of the $5n$-stage implementation increases but the frequency of the n-way superscalar implementation decreases. Explain why. Assume that both types of system are well designed.

☑ Clock increases in the $5n$-stage system because ...

... the critical path in each stage is shorter as n increases and so the clock frequency can be increased. (In the $5n-$stage system each stage in the 5-stage system is divided into n stages each taking $\frac{1}{n}$ the amount of time, ideally.) Ignoring latch setup time and assuming stages can be perfectly divided, the clock frequency would be $n\phi$, where ϕ is the clock frequency of the 5-stage system. Accounting for a latch setup time of t_L, the clock frequency would be $\frac{n\phi}{t_L\phi(n-1)+1}$. It is this increase in clock frequency that results in higher performance.

☑ Clock decreases in the n-way system because ...

The design is becoming larger and so physical distances increase. A stage can have $n\times$ as much hardware, one set of hardware for each of n slots. However signals only go through one set of hardware, so there is no need for a reduction in clock frequency by a factor of n. The problem is that physical distances are increasing, forcing paths to be longer.

(*d*) A company is preparing a run of the SPEC benchmarks for their new system. While trying to tweak compiler flags to get the best performance they discover a new optimization technique and implement it in their compiler. They re-test with that compiler and disclose the test results. When is the use of the modified compiler allowed under SPEC rules? When isn't that allowed under SPEC rules?

☑ Modified compiler allowed provided that ...

☑ This is a beneficial because ...

... provided that the compiler is seriously made available to the public. This is beneficial because the modified compiler generates better code than the old one.

☑ Modified compiler isn't allowed if ...

☑ The modified compiler is not beneficial because ...

... isn't allowed if the compiler is not made public, or not seriously marketed as a product. (For example, if the compiler has a product number but does not appear in any catalog or promotional material.) Ordinarily a company would be happy to market a better compiler. If they are not doing so it might be because the compiler generates buggy code when using the new optimizations. So it is not beneficial because the compiler, though it works on the SPEC benchmarks, would generate buggy code when compiling anything else. That means that the performance numbers obtained on SPEC code do not reflect what a user might obtain on other code.