Name Solution

Computer Architecture
EE 4720
Final Examination

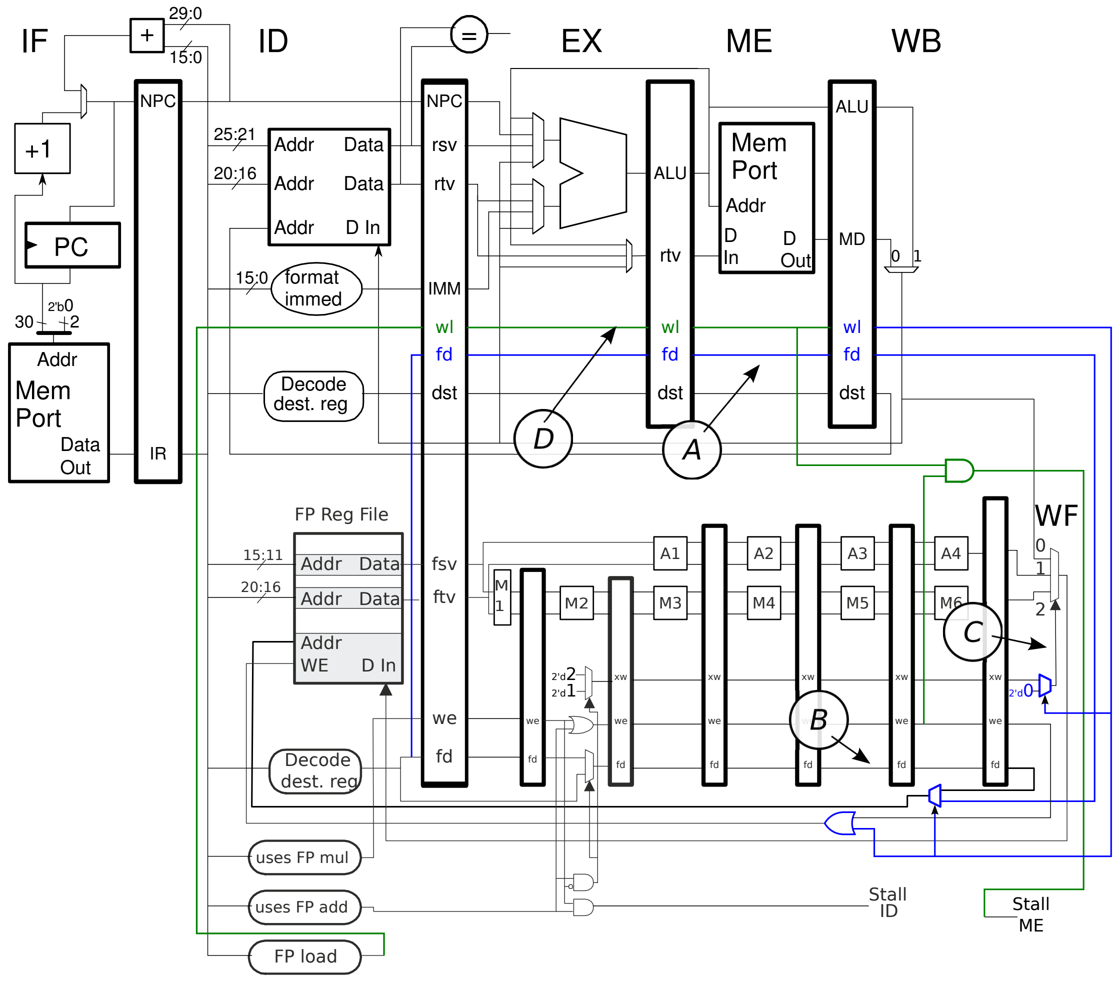
10 May 2014, 10:00–12:00 CDT

Problem 1		(15 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(5 pts)
Problem 6		(10 pts)
Problem 7		(25 pts)
Exam Total		(100 pts)

Alias Click Here

Good Luck!

Problem 1: (15 pts) Illustrated below is the **stall-in-ME** version of our MIPS implementation, taken from the solution to the midterm exam.



(*a*) Show a pipeline execution diagram of the code below on this pipeline.

(*b*) Wires in the diagram are labeled A, B, C, and D. Under your pipeline execution diagram show the values on those wires when they are in use.

☑ Show pipeline execution diagram. ☑ Show values of A, B, C, and D.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8
add.s f4,f5,f6   IF ID A1 A2 A3 A4 WF
sub.s f7,f8,f9      IF ID A1 A2 A3 A4 WF
lwc1 f1, 0(r2)         IF ID EX ME ----> WF
A                               1  1  1
B                            4  7
C                                  1  1  0
D                            1
# Cycle          0  1  2  3  4  5  6  7  8
```

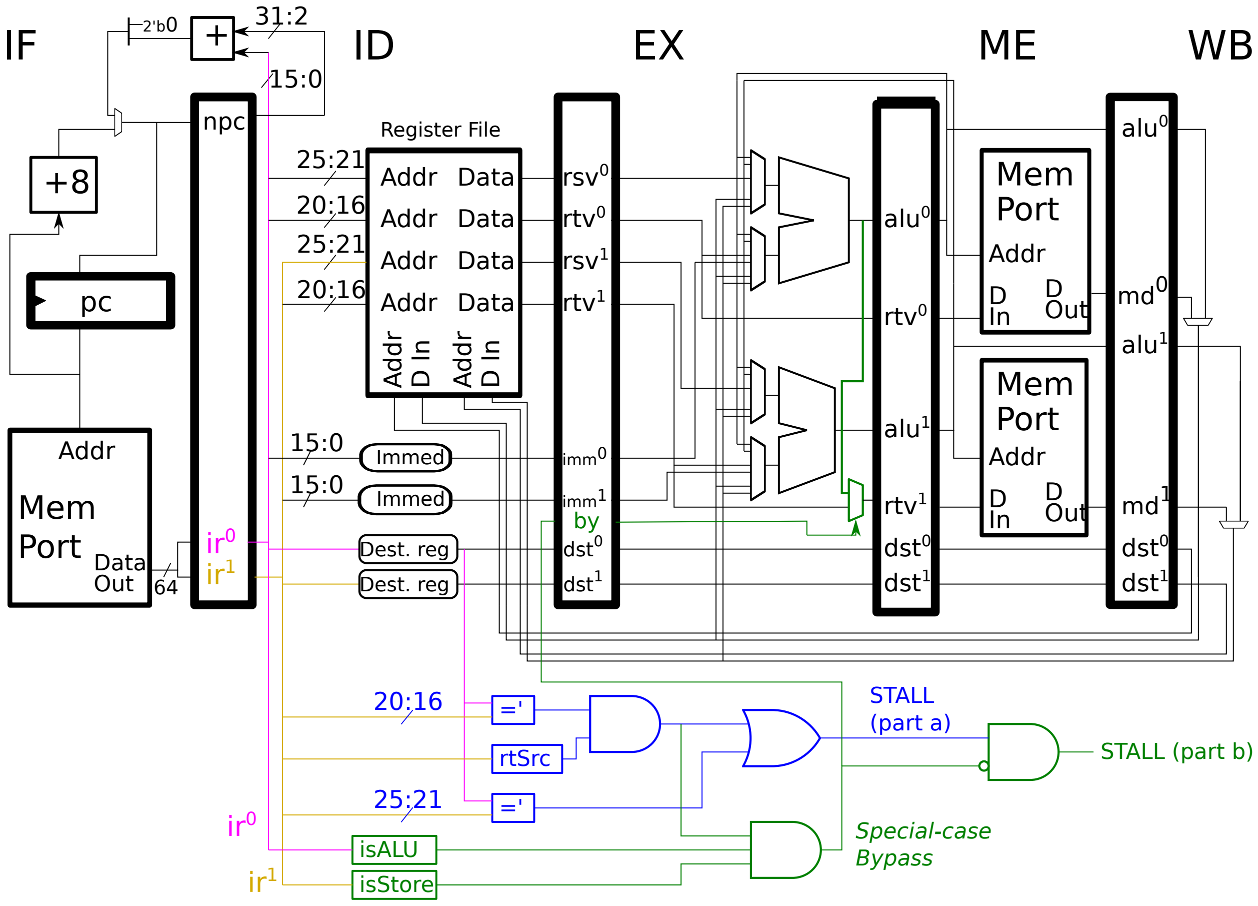
Solution appears above. Values on A are only used by the FP load instructions, in this case, `lwc1`, and so in the table above the values are only shown in cycles 5, 6, and 7. That said, in cycle 3 A would have a 4 and in cycle 4 it would hold a 7.

Values on B are used for destination of all other instructions that write the floating-point register file, for the code above the `add.s` and `sub.s` instructions. Notice that the value shown for B is for the instruction in the A3 stage.

Values on C control which source will be written back to the FP register file.

The value on D is 1 when a FP load is in the EX stage.

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block $\boxed{\text{isALU}}$ is 1 if the instruction's result is computed by the integer ALU. The output of logic block $\boxed{\text{rtSrc}}$ is 1 if the instruction uses the `rt` register as a source. The output of logic block $\boxed{\text{isStore}}$ is 1 if the instruction is a store.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☑ Control logic to detect true dependence in `ID` and assign `STALL`.

Solution appears above in blue. The stall signal for this part is the output of the OR gate. To help understand what's going on the IR (instruction register) for slot 0 is shown in purple and the IR for slot 1 is shown in gold. The logic compares the destination of the instruction in ID slot 0 with the two sources of the instruction in slot 1. If either matches, the stall signal is generated (which before part b would be the output of the OR gate). The $\boxed{\text{rtSrc}}$ logic is needed because the `rt` field might hold a destination register number, and we would not want to stall for that.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular instruction pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Note: There was a similar-sounding problem in last year's final, but the solutions are different.*

```
0x1000: add r1, r2, r3
0x1004: sw r1, 0(r5)
```

☑ Add the bypass path(s) needed so that the code executes without a stall.

☑ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

Solution appears in green.

The special case here that makes a stall unnecessary is that the `sw` does not need the result of the `add` until the end of the EX stage (or the beginning of ME). Therefore the result can be bypassed from the output of the slot-0 ALU to the input of the `EX/ME.rtv1` pipeline latch. The logic for this bypass path is in the EX stage, shown in green. (It would also be correct to put the bypass in the ME stage.)

The control logic for this special-case bypass is also shown in green. The three-input AND gate checks for a dependence between the destination of the slot-0 instruction and the `rt` source of the slot-1 instruction (top input), whether the slot-0 instruction's result is produced by the ALU (middle input), and whether the slot-1 instruction is a store (bottom input). If all are true the bypass signal is set to true. The bypass signal is used for the new `ID/EX.by` pipeline latch and for the AND gate suppressing the stall signal.

Problem 3: (15 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. Notice that B2's outcomes come in groups of three, such as 3 qs. The first outcome of each group is random and is modeled by a Bernoulli random variable with $p = .5$ (taken probability is .5). The second and third outcomes are the same as the first. For example, if the first q is T the second and third q will also be T. If the first s is N, the second and third s will also be N. Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T   T   T   N   N      T   T   T   N   N     ...
B2:   r   r   r   q   q      q   s   s   s   u    ...
B3:    T   T   T   T   T      T   T   T   T   T   ...
```

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

The accuracy is $\boxed{\frac{2}{5}}$. The work to compute this result is shown below. The prediction accuracy is based on the second pattern repetition because the counter value at the start of the pattern and the end of the pattern is the same, 1. (In contrast, the counter value at the start of the first repetition is 0 and at the end it is 1, so we can't use the first five outcomes of B1 to compute an accuracy.)

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

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

At the end of a group of three outcomes, call them r1, r2, and r3, the two-bit counter will be either 0 or 3. Consider two possible cases: In Case 1 the next three outcomes, call them q1, q2, and q3, are the same as the r; in Case 2 the next three outcomes are different. See the diagram below. The probability of a q1 falling into Case 1 is .5. Because the branch directions agree in Case 1 the three qs will be predicted correctly. In Case 2, where r and q are different, q1 and q2 will be mispredicted, but q3 will be correctly predicted. See counter values in the diagram below.

The overall prediction accuracy is the average of the two, $\boxed{\frac{1}{2}\left(\frac{3}{3}+\frac{1}{3}\right)=\frac{4}{6}}$.

```
Case 1: r and q agree.  Probability 0.5.
                   3   3   3   3     <- Counter value
B2:   r   r   r    q   q   q  ...
B2:   T   T   T    T   T   T  ...   <- Let q = T and r = T.
                                     <- Zero mispredictions of q.

Case 2: r and q disagree.  Probability 0.5
                   3   2   1   0     <- Counter value
B2:   r   r   r    q   q   q  ...
B2:   T   T   T    N   N   N         <- Let q = T and r = N.
                   x   x             <- Two mispredictions of q.
```

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

Three outcomes are sufficient. With two outcomes a local history of TT could appear before the third T or the first N.

[✓] What is the accuracy of the local predictor on branch B2, after warmup. [✓] Explain.

The local predictor will predict the first branch of a group of three with a 50% accuracy. (No predictor can do better or worse.) The second and third outcomes will be predicted with 100% accuracy after warmup. That's because the PHT entries corresponding to local histories ending with an N will saturate down to a zero, and those ending with T will saturate up to 3. (See the next part.) The overall accuracy is $\boxed{\frac{1}{3}\frac{1}{2} + \frac{2}{3}\frac{2}{2} = \frac{5}{6}}$.

[✓] What is the best local history size for branch B2, *taking warmup into consideration.* [✓] Explain.

For any local history size greater than zero the prediction accuracy will be $\frac{5}{6}$ after warmup. The longer the local history size is the longer the warmup time is. The minimum local history size needed is just one outcome. The local predictor will predict the same outcome as the most recent occurrence.

[✓] How many different GHR values will there be when predicting B3?

Let 312 312 312 312 indicate the state of the 12-bit GHR when predicting B3. Each digit indicates the branch affecting the corresponding bit position in the GHR. For example, branch B2 affects the rightmost bit and three other bits. Because B3 is perfectly biased we know that all the 3's must be T. The four 1s together can have five possible values, TTTN, TTNN, TNNT, NNTT, and NTTT. Considering just B1 and B3 there are $1 \times 5 = 5$ possible GHR values. To compute the number of possible patterns for branch B2 we need to consider three cases for the first 1 position: that it corresponds to a q1 (first of a group of three), a q2, or a q3. In each of those cases there are four distinct patterns. For example, in case 1 the first three 1's are from one group of three and the last 1 is from a different group of three. There are two possible outcomes for each group. For each case, one of the four patterns is all Ts and one is all Ns. The number of distinct patterns due to branch B2 is $3 \times 2 + 2 = 8$. The patterns due to B1 repeat every 5 occurrences of B3, while the patterns due to B2 sort of repeat every 3. Because 3 and 5 don't share a common factor > 1 the total number of patterns in the GHR is the product of the number of patterns due to B1 and B2. The total is $\boxed{5 \times 8 = 40}$.

Grading Note: For full credit one only needs to show an approach that will lead to the correct answer. The key insight that needs to appear is multiplying the five patterns of B1 *by something like $a2^b$, or some diagram that can be used to find the different GHR patterns.*

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries. Also, consider the same branch patterns, they are repeated below, along with the address of branches `B1` and `B2`. The branch predictors are part of a MIPS implementation.

```
0x1234: B1:  T   T   T   N   N      T   T   T   N   N     ...
0x1242: B2:   r   r   r   q   q      q   s   s   s   u    ...
        B3:    T   T   T   T   T      T   T   T   T   T   ...
```

(*b*) Choose an address for branch `B3` that will result in a BHT collision with branch `B1`.

☑ Address for `B3` 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 `0x1234` (note that each hex digit spans four bits). For `B3` to use the same entry the four least significant hex digits should match. One such address is $\boxed{\texttt{0x11234}}$.

(*c*) How does the collision change the prediction accuracy of the bimodal predictor on the two branches?

☑ Change in `B1` and ☑ Change in `B3`.

For this particular case the accuracy of `B1` will improve because `B3` will keep the 2-bit counter from falling below 2. This will improve `B1`s accuracy to $\frac{3}{5}$. The accuracy of `B3` won't change. The Ns in `B1` will never decrement the counter twice in a row (because of `B3`'s outcome) and so the counter will never go below 2.

(*d*) (The answer to the following question does not depend on the sample branch patterns above.) Suppose we detect a BHT collision (perhaps by using tags). Why should we predict not taken?

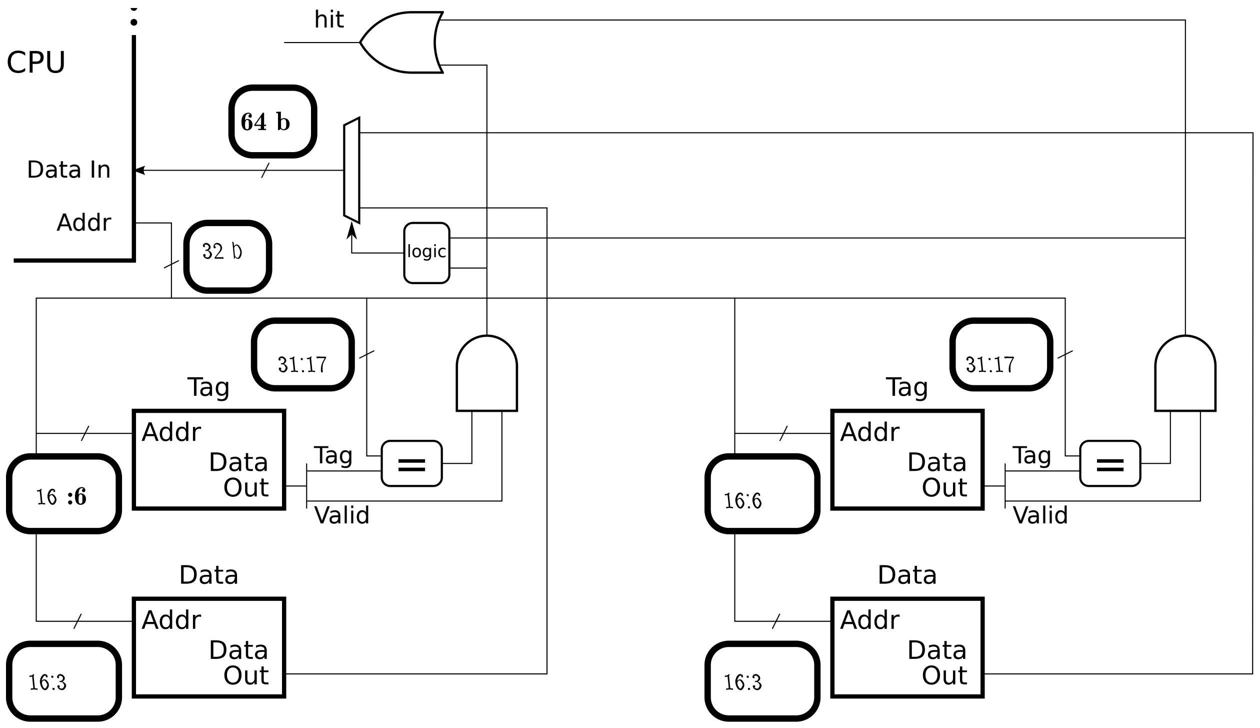
☑ Reason for predicting not-taken for a collision.

Because a collision indicates that the BHT entry we found is not for the branch we are trying to predict. Since it's for a different branch the target, which can be stored in the BHT, would very likely be wrong. So even though there's a .5 probability that a taken prediction is correct there is a much smaller chance that the target is correct.

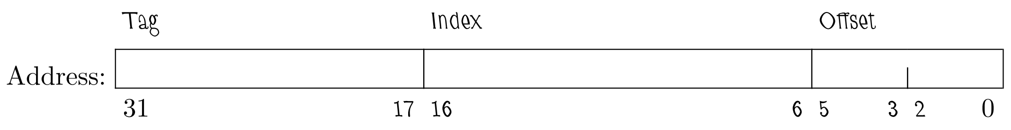
Problem 4: (15 pts) The diagram below is for a 256 kiB (2^{18} B) set-associative cache. Hints about the cache are provided in the diagram.

(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 $\boxed{\text{associativity is 2}}$. From the diagram we can see that there are two ways. Notice that there's no ellipsis between the ways, as there would be on a diagram in which not all the ways are shown.

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

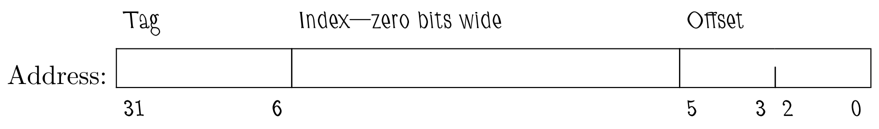
It's the cache capacity, 256 kiB bytes, plus $2 \times 2^{17-6}(32 - 17 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Lower bit position of the address going into the tag store gives the line size, $2^6 = 64$ characters.

[✓] Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Because the cache is fully associative the number of index bits is zero. The line size doesn't change so we don't change the offset bit positions. Instead we increase the size of the tag. The bit categorization appears below.



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 128 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

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

```
  double sum = 0;
  double *a = 0x2000000; // sizeof(double) == 8
  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 $2^7 = 128$ bytes is given. The size of an array element, which is of type double, is $8 = 2^3$ characters, and so there are $2^7/8 = 2^{7-3} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the $\boxed{\text{hit ratio is } \frac{15}{16}}$.

(*c*) Find the largest value for `BSIZE` for which the second `for` loop will enjoy a 100% hit ratio.

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

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

A key fact to understand when solving this problem is that two consecutive elements, say `b[0]` and `b[1]`, will be on two consecutive lines. (That's because each element, `Some_struct`, is the size of a line. `Some_struct` contains 16 doubles [14 doubles in the array, `a`] and the size of 16 doubles is $16 \times 8 = 128\,\mathrm{B}$ which is the line size.)

The size of the cache is $4\,\mathrm{MiB}$ which works out to $\frac{4\,\mathrm{MiB}}{128\,\mathrm{B}} = 2^{22-7} = 2^{15}$ lines. A hasty student might assume the problem solved at this point, and write $\boxed{\mathtt{BSIZE}{=}2^{15}}$, or as is shown in the solution as the equivalent C expression, `1 << 15`. In this case, haste is not being punished, that's the right answer. It's hasty because of the assumption that all 2^{15} lines could co-exist in the cache.

To determine whether they really can all be in the cache at the same time, consider the sequence of addresses broken into tag, index, and offset parts. Because the size of `Some_struct` is the same size as the cache line, we know that if the index of `&b[i].val` were x then the index of `&b[i+1].val` would be $x + 1$. Because the cache is 4-way set-associative the number of lines per way is $2^{15}/4 = 2^{13}$. That means the indices run from 0 to $2^{13} - 1 = 8191$. So when is `BSIZE`$=2^{13}$ we will encounter each index exactly once, and may completely fill each way. If `BSIZE`$=2^{14}$ then we would encounter each index twice, say for `i=0` and `i=32768`. Element `b[0]` might be placed in way 0 and element `b[32768]` might be placed in way 1. No problem. Since the cache is 4-way we can make `BSIZE` 2^{15} without a problem. If `BSIZE` were $2^{15} + 1$ then we would be using an index five times. The fifth time we use an index we would evict the data for element $i = 0$.

Bonus problem: What would happen if the array `a` were 30 elements? In that case the size of an element would be two lines, and if the index for element $b[i]$ were x, the index for element $b[i + 1]$ would be $x + 2$. In that case half the cache would go unused, and so `BSIZE` would have to be made half as large.

Problem 5: (5 pts) The displacement in MIPS branches is 16 bits. Consider a new MIPS branch instruction, `bfeq rsn, rtn` (branch far), where `rsn` and `rtn` are 2-bit fields that refer to registers 4-7. As with `beq`, branch `bfeq` is taken if the contents of registers `rsn` and `rtn` are equal. With six extra bits `bfeq` can branch 64 times as far.

(*a*) Show an encoding for this instruction which requires as few changes to existing hardware as possible.

[✓] Encoding for `bfeq`. [✓] Explain how minimizes changes.

The encoding for `bfeq` is shown below. The 22-bit displacement is `is3,it3,Immed` (in Verilog notation).

Encoding for `bfeq`:

Opcode	is3	rs2	it3	rt2	Immed
31 26	25 23	22 21	20 18	17 16	15 0

In the encoding above the `rs` and `rt` register fields each have been shortened from five to two bits. (The original format I encoding is shown below.) Since the fields have been shortened but not moved the four remaining bits can be connected directly to the register file. (See the solution to the next part.)

MIPS I:

Opcode	rs	rt	Immed
31 26	25 21	20 16	15 0

The encoding below, which would receive partial credit, would result in more costly implementations. This inferior encoding is certainly more organized with `rs2` and `rt2` next to each other and with the immediate occupying 22 contiguous bits. However the multiplexors at the register file inputs would need to be five bits instead of three bits. Notice that it does not make a difference whether or not the immediate bits are contiguous, as they are below, or split, as they are in the solution above.

Inferior `bfeq`:

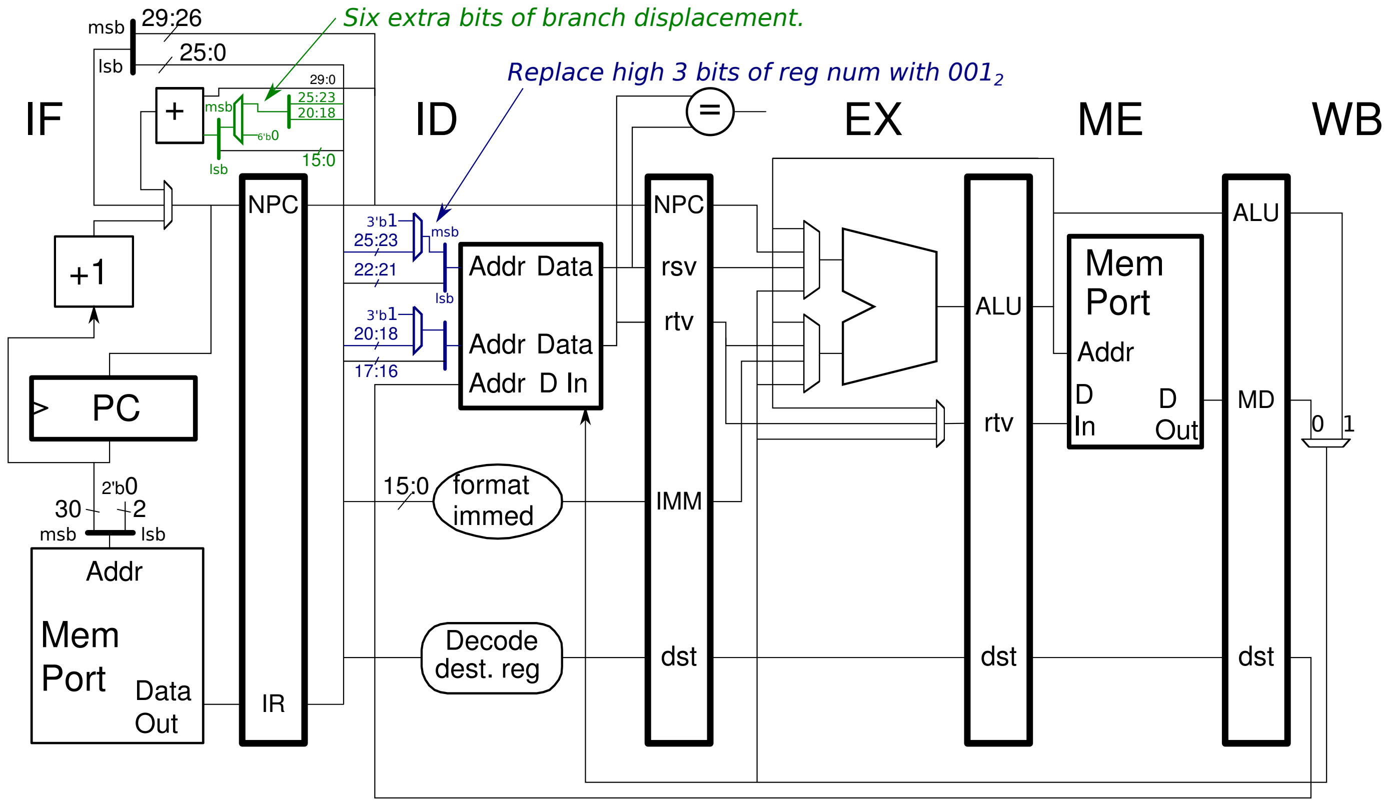
Opcode	rs2	rt2	imm22
31 26	25 24	23 22	21 0

(*b*) Modify the pipeline below to implement the new instruction. Use as little hardware as possible.

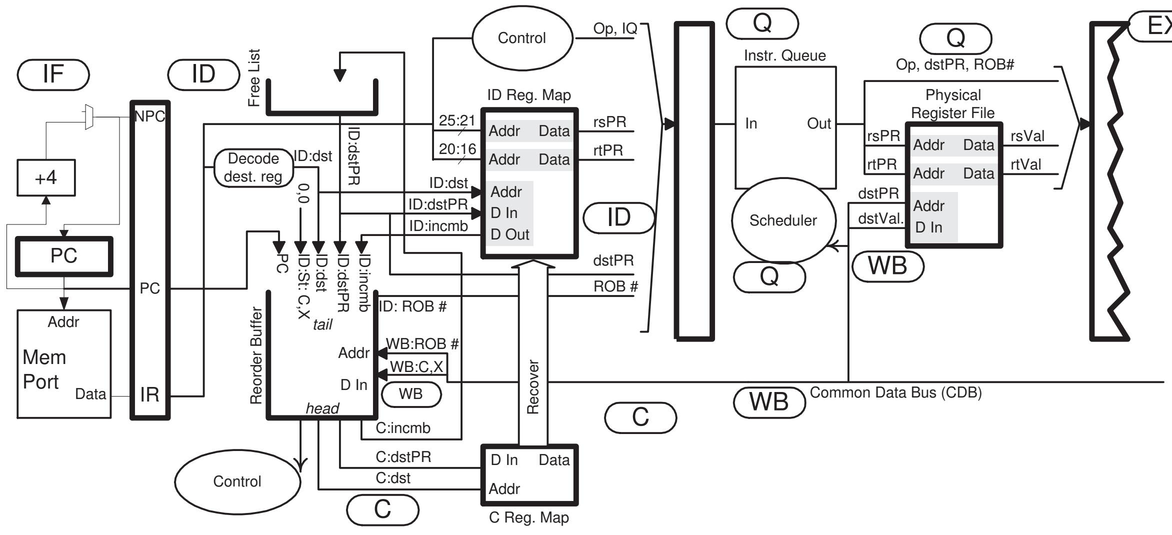
☑ Briefly show changes.

Changes appear below. At the register file address inputs, shown in blue, the high three bits of each register number is determined by a multiplexor. If a `bfeq` is present the upper inputs are used, making the upper three bits of each register number 001_2, otherwise the upper bits come from the instruction. The lower two bits are taken from the instruction regardless of whether `bfeq` is present.

If a `bfeq` is present the displacement includes the extra six bits, these changes are shown in green.



Problem 6: (10 pts) Illustrated below is a dynamically scheduled four-way superscalar MIPS implementation and the execution of code on that implementation.



```
LOOP: #   Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 lwc1 f2, 0(r1)     IF ID Q  RR EA ME WB C
 add.s f4, f4, f2   IF ID Q        RR A1 A2 A3 A4 WB C
 bne  r1, r2, LOOP  IF ID Q  RR B  WB                C
 addi r1, r1, 4     IF ID Q  RR EX WB                C
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 lwc1 f2, 0(r1)        IF ID Q  RR EA ME WB          C
 add.s f4, f4, f2      IF ID Q                 RR A1 A2 A3 A4 WB C
 bne  r1, r2, LOOP     IF ID Q  RR B  WB                         C
 addi r1, r1, 4        IF ID Q  RR EX WB                         C
#         Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*a*) On the diagram above indicate when each instruction will commit.

☑ Show commits on diagram above.

Commits shown above (they are indicated with a C). Note that commits must occur in program order and that there cannot be more than four commits per cycle. (Program order means that a commit for an instruction cannot occur before the commit for a preceding instruction.)

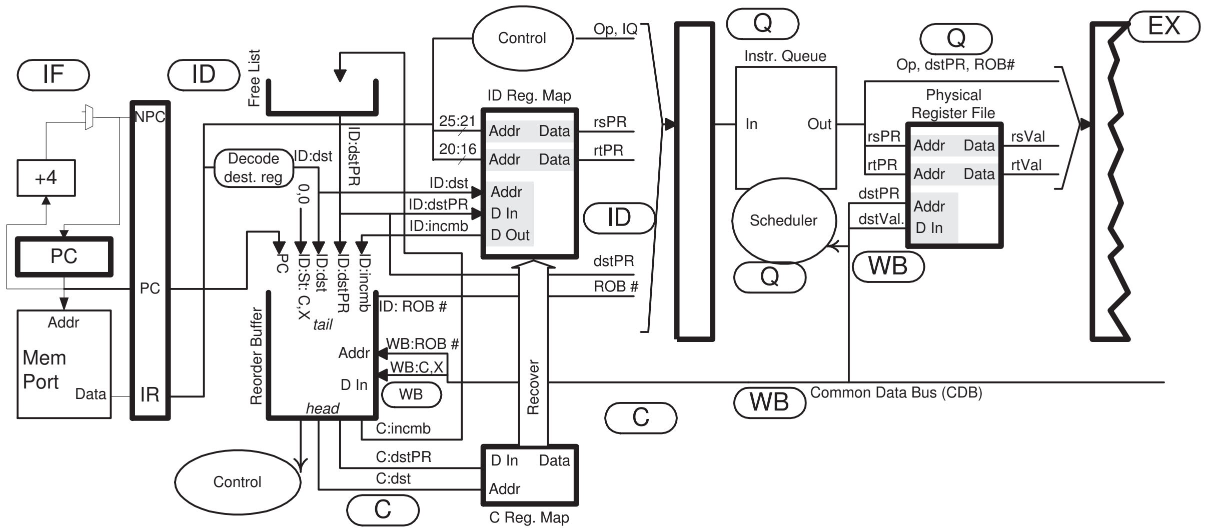
(*b*) What is the execution rate, IPC, for the code above for a large number of iterations assuming perfect branch prediction. Note that the system is dynamically scheduled.

☑ IPC for code above for large number of iterations.

Short answer: the code will execute at just 1 insn/cycle due to the `add.s`.

Longer answer: Notice that one source of the `add.s` instruction, `f4`, is the result of the `add.s` instruction in the prior iteration. This means that execution can go no faster than four cycles per iteration. The sample execution (the one in the unsolved problem, not just the solution) shows that there are bypass paths so the second iteration `add.s` can start as soon as the first iteration `add.s` result is ready. This suggests that four cycles per iteration is possible. Instruction fetch and decode goes much faster, one cycle per iteration. Eventually though the ROB will fill causing `IF` to periodically stall. On average each iteration will take four cycles and each iteration consists of four instructions, for an IPC of $\frac{4}{4} = 1$.

(*c*) On the next page there is a table showing the values of selected signals during the execution of the code, the signals are related to register renaming. Show values where indicated on the table. Note that `ID:incmb` is already shown in cycle 1, show its values for later cycle(s).



☑ Show values where indicated.

SOLUTION shown in blue.

```
LOOP:     Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)      IF  ID  Q   RR  EA  ME  WB  C
 add.s f4, f4, f2    IF  ID  Q           RR  A1  A2  A3  A4  WB  C
 bne  r1, r2, LOOP   IF  ID  Q   RR  B   WB                      C
 addi r1, r1, 4      IF  ID  Q   RR  EX  WB                      C
LOOP:     Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)          IF  ID  Q   RR  EA  ME  WB              C
 add.s f4, f4, f2        IF  ID  Q                       RR  A1  A2  A3  A4  WB  C
 bne  r1, r2, LOOP       IF  ID  Q   RR  B   WB                                  C
 addi r1, r1, 4          IF  ID  Q   RR  EX  WB                                  C
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID:dstPR 0 (lwc1)        65  62         for  register  f2
ID:dstPR 1 (add.s)       97  69         for  register  f4
ID:dstPR 3 (addi)        60  79         for  register  r1
# Show values for signals below, including incmb.
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID:incmb 0 (lwc1)        83  65
ID:incmb 1 (add.s)       20  97
ID:incmb 3 (addi)        67  60

ID:rsPR 0 (lwc1)         67  60

ID:rsPR 1 (add.s)        20  97

ID:rsPR 3 (addi)         67  60

ID:rtPR 1 (add.s)        65  62                                  <- rt, not rs
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
WB:dstPR 0 (lwc1)                                65  62
WB:dstPR 1 (add.s)                                           97              69
WB:dstPR 3 (addi)                            60  79
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

Problem 7: (25 pts) Answer each question below.

(*a*) Describe how cost and performance limit the practical largest value of width (value of n) in an n-way superscalar implementation.

✓ Cost limiter.

Bypass paths become a major element when n is large. Consider an n-way superscalar design derived from the 5-stage statically scheduled MIPS implementation used in class. The `EX` stage will have n 2-input ALUs, requiring a set of bypass paths for each of the $2n$ inputs. The `ME` and `WB` stage will each hold n results. The number of multiplexor inputs needed to provide bypass paths from those $2n$ results to the $2n$ inputs is $4n^2$. At 32- or 64 bits each, that will quickly dominate cost.

✓ Performance limiter.

There will be several performance limiters. One major limiter is the lack of sufficient instructions to execute in parallel. In a statically scheduled n-way superscalar design there cannot be true dependencies between the n instructions in a group to avoid stalls. If there are dependencies then the group will spend two or more cycles in `ID`, reducing performance. The larger n is the more frequently such dependencies will occur. Dynamic scheduling helps but does not eliminate the problem.

Another major performance limiter with large n are branches. Branches occur frequently in integer code, perhaps once every five or six instructions. In practical designs execution could only reach up to the first taken branch in a group. (In impractical [academic] designs multiple branches can be predicted per cycle and instructions can be fetched from multiple non-adjacent areas per cycle).

(*b*) What is the most important factor in determining the size of a level 1 cache?

✓ Most important factor in L1 cache size.

Clock frequency and load latency. The amount of time it takes to retrieve data from a memory is a function of its size, the larger the slower. Typically designers would set a target for the clock frequency and for the number of cycles it would take to retrieve data from the L1 cache. The largest L1 size that can meet these requirements would be chosen. For example, suppose we chose two cycles for an L1 hit (the pipeline might have two memory stages, `ME1` and `ME2`), and suppose we chose a clock period of $0.7\,\mathrm{ns}$. That would give us $1.4\,\mathrm{ns}$ to retrieve data from the cache. We would choose the largest L1 size that could provide the data in $1.4\,\mathrm{ns}$. The remaining chip area might be used for an L2 cache.

(*c*) Suppose the 16-bit offset in MIPS `lw` instructions was not large enough. Consider two alternatives. In alternative 1 the offset in the existing `lw` instruction is the immediate value times 4. So, for example, to encode instruction `lw r1, 32(r2)` the immediate would be 8. In alternative 2 the behavior of the existing `lw` is not changed but there is a new load `lws r1, 32(r2)`, in which the immediate is multiplied by 4. Note that alternative 2 requires a new opcode. Which instruction should be added to a future version of MIPS?

✓ Should choose alternative 1 or alternative 2? ✓ Explain.

Alternative 2, `lws`, should be chosen so that existing software continues to run correctly.

(*d*) The SPECcpu suite comes with the source code for the benchmark programs. How does that help with the goal of measuring new ISAs and implementations?

✓ Source code helps with testing new implementations because:

Since we have the source code we can compile the benchmarks ourselves. In fact, for SPECcpu testers are required to compile the code for themselves, using the compiler of their choosing (within reason). If we are testing a new implementation then we would want to use a compiler that can optimize for that implementation. If we are testing a new ISA then there would be no choice but to use a new compiler, since old compiler would not be able to generate code for it.

(*e*) What's the difference between a 4-way superscalar implementation (of say MIPS) and a VLIW system with a 4-slot bundle?

✓ Difference between superscalar and VLIW.

The superscalar implementation must be able to find and handle dependencies between any pair of instructions in a group (the group of 4 instructions traveling together through ID and beyond) and must be able to handle any instruction in any slot within a group. In contrast, the VLIW implementation need only handle a subset of possible instruction in each slot. For example, slot 0 might never have branch instruction and slot 3 might never have a memory instruction. Furthermore, dependencies between instructions in a bundle might be forbidden. The expectation is (was) that these differences would enable less expensive or higher performance VLIW implementations.