Name Solution

Computer Architecture

EE 4720

Final Examination

11 December 2007, 15:00–17:00 CST

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias ISA were, was I?

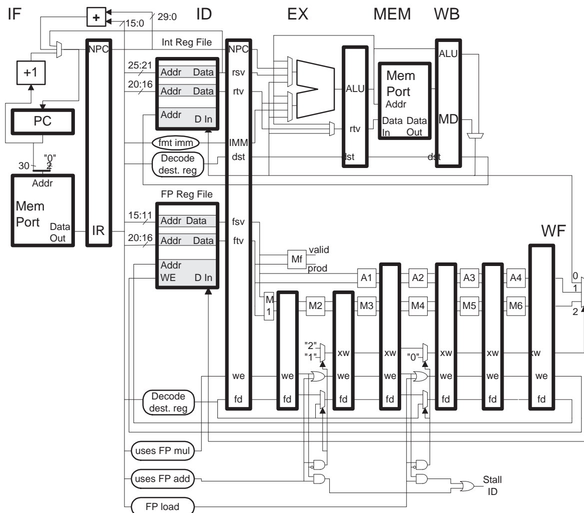
Good Luck!

Problem 1: (25 pts) Suppose it turns out that many `mul.d` instructions are executed with a 0 or 1 for one of the operands, in that case the product could be computed in less than the six stages used in the class implementation. The MIPS implementation below includes a new *multiply fast unit*, `Mf`, for such situations.

`Mf` has two inputs (unlabeled) and two outputs, `valid` and `prod` (they are not yet connected to anything). As with `A1` and `M1`, IEEE 754 doubles are expected at the inputs of `Mf`. If one of the inputs is 0 or 1 then the `valid` output will be `1` (otherwise it is `0`). If `valid` is `1` the `prod` output is the product of the two inputs; both outputs are available by the end of the cycle.

Call a multiply *fast* if one of its operands is 0 or 1.

USE NEXT PAGE FOR SOLUTION!



USE NEXT PAGE FOR SOLUTION!

(*a*) Add datapath and control logic for `Mf` meeting the requirements below:

☑ Add datapath so the `Mf` output will be written to the FP register file at the right time.

Datapath shown in blue on diagram on next page. The solution has the fast-multiply product pass through an extra stage so that `WF` is fifth stage, to avoid the special case of a fast multiply being the only four-stage instruction (making it hard to squash in certain circumstances).

☑ In order of decreasing priority: the added datapath must be correct, must not increase critical paths, and should use as little new hardware as possible.

The datapath does not add multiplexors at the inputs or outputs of any functional unit (such as `A4`), so it does not strain critical path. If hardware savings were more important than critical path the fast multiply product might have been muxed in after `A3` or `A4`.

☑ The control logic must detect and handle the new `WF` structural hazard with preceding instructions that's possible when writing a fast product.

The control logic appears in green on the next page. The logic labelled Control logic to insert fast multiply in the diagram detects and handles the new structural hazard. If one exists the fast multiply path is not taken. (The alternative would be to stall.)

☑ If `Mf` is used then the `mul.d`'s usual `WF` slot should be available for other instructions. For example, suppose the second instruction after `mul.d` is an `add.d`. If the multiply is normal the `add.d` would stall, if the multiply is fast the `add.d` shouldn't stall (at least for the `WF` conflict with `mul.d`).

The logic labelled Control to stop mult at M2 makes the multiply's `WF` slot available to other instructions by setting `we` to `0` if the fast multiply will use `WF`.

(*b*) Add bypass hardware and control logic so that the code below executes without a stall if the `mul.d` turns out to be fast. For this part assume that multiplexors at FP unit inputs won't add to critical path.

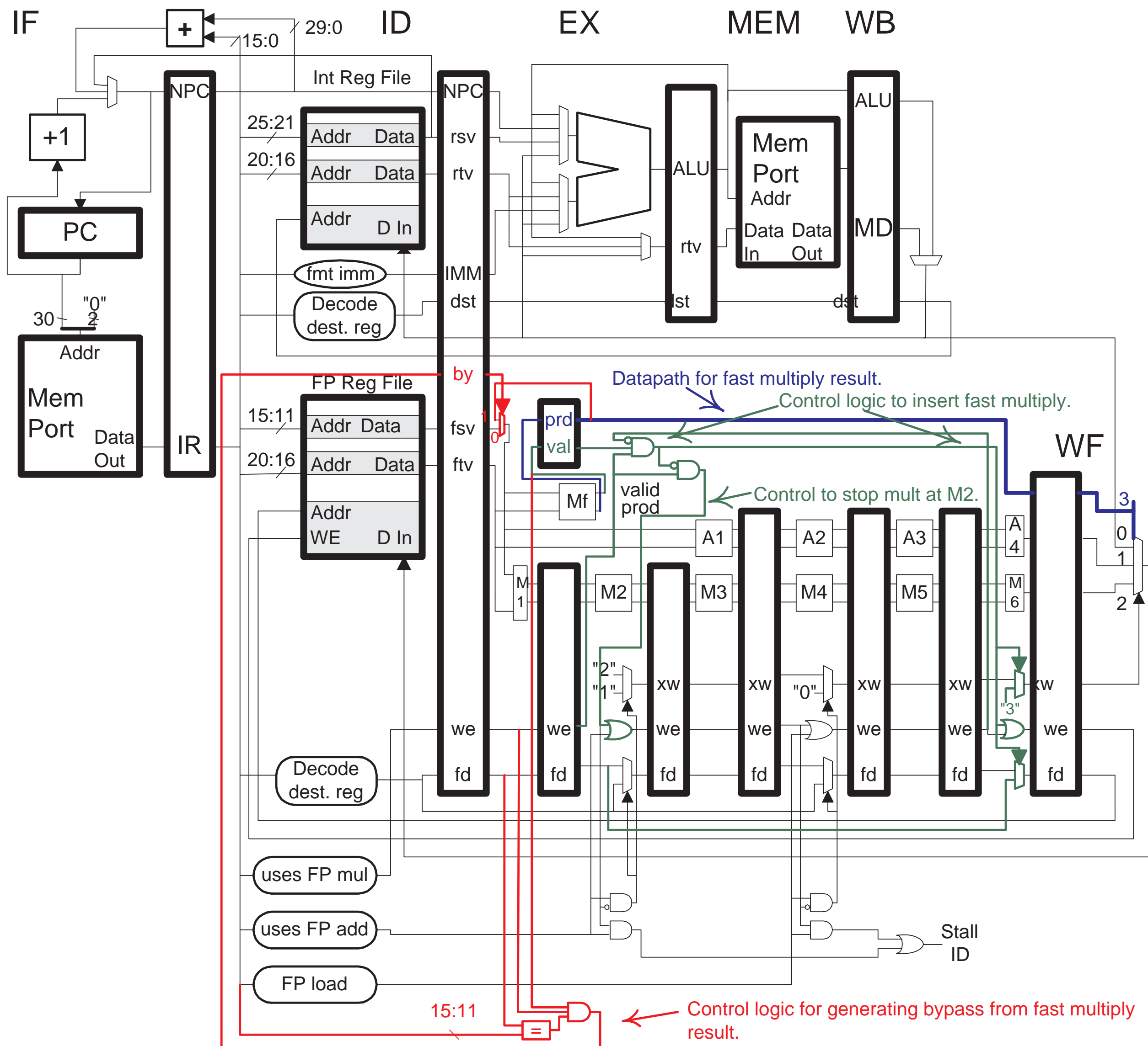
```
# Cycle              0  1  2  3  4  5  6  7
mul.d f2, f4, f6     IF ID MF Mx WF
add.d f8, f2, f10       IF ID A1 A2 A3 A4 WF
```

☑ Bypass hardware for case above.

The bypass logic appears in red. See the pipeline diagram above for stage abbreviations. Note that the bypass logic is in `Mf` because the `add.d` must make a stall decision when it is in `ID` and so it can't be done any later.

The logic for selecting the `WF` stage is computed when the `mul.d` is in `Mx` because that logic is more elaborate and would add to the critical path if it were in `Mf` (there is less time pressure in `Mx`).

Note that the `add.d` can bypass a fast multiply result even if the fast multiply won't write back.



Problem 2: (25 pts) Answer the following predictor questions.

(a) The code below executes on three systems, one uses a bimodal predictor with a 2^{14}-entry BHT, one uses a local predictor with a 2^{14}-entry BHT and a 7-outcome local history, and one uses a global predictor with a 7-outcome global history. Consider the execution of the code below, all branches are shown.

```
BIGLOOP:
...
B1: beq r1, r2, SKIP1   T T T T N T T T T N T T T T N T T T T N ...
...
SKIP1:
...
B2: bne r3,r4 SKIP2      T T T N N N T T T N N N T T T N N N T T T N N N ...
...
SKIP2:
...
j BIGLOOP
nop
```

For partial credit show work, not just answer.

☑ Accuracy of bimodal predictor on B1 after warmup:

Accuracy is $\frac{4}{5} = 80\%$.

☑ Accuracy of bimodal predictor on B2 after warmup:

Accuracy is $\frac{2}{6} \approx 33.3\%$.

☑ Accuracy of local predictor on B2 after warmup:

Accuracy is 100%

☑ Warmup time of local predictor on B2:

Warmup time is $6 + 6 \times 2 = 19$ executions of B2.

☑ Minimum local history size for 100% accuracy of local predictor **on B2** (without ignoring B1) (show work):

Minimum size is 5 outcomes. Four outcomes are not enough because pattern `TTTN` would occur in both B1 and B2 and the next outcomes would disagree: `T` for B1 and `N` for B2.

☑ Accuracy of global predictor on B2 after warmup:

Accuracy is $\frac{30-2}{30} \approx 93.3\%$. There are $5 \times 6 = 30$ possible GHR values (patterns). Twenty four of these will only occur when predicting B1 or predicting B2 (but not both). For example, pattern `tTnTnNn` can only occur when predicting B1 (because B1 never has 3 (or 2) consecutive `n` outcomes). (The upper-case letters are the outcomes of the branch being predicted, the lower-case letters are outcomes of the other branch.) Those 24 will be followed by correct predictions. There are two shared patterns in which the outcomes agree by happy coincidence: `tNtTtTn` and `nNtTtTt`. Pattern `tTtTtTn` is used twice by B1 with differing outcomes so the impact on the PHT entry cancels out, it is also used once by B2 and so the entry reaches a value suitable for B2, 0; the same with pattern `nTtTtTt`. For all patterns discussed so far correct predictions are made for B2. There are two in which B2 is incorrectly predicted: `tTtTtNn` and `nTtTtTt`. It is used once per unit by both B1 and B2, since B1 is more frequent the PHT entry reflects B1's next outcome: `T`. Summing it up, correct predictions will be made for B2 on 28 out of 30 patterns.

☑ Warmup time of global predictor on B2 (explain):

Warmup time is $6 + 5 \times 6 \times 2 = 66$ executions of B2. Based on number of distinct GHR patterns.

Problem 2, continued: Consider the execution of the code fragment below on a system using branch prediction. The value of `c` used in the switch statement is random, uniformly distributed over `a` to `z`, and outcomes are independent (like a 26-sided die). The branch implementing the `if ( x < 5 )` statement, which will be called the *if* branch, is taken 50% of the time.

```
int c = getchar(); // Unpredictable
switch (c) {
 case 'a': x = 3; j++; break;
 case 'b': x = 7; break;
 ...
 case 'z': x = 1; i++; break;
}
if ( x < 5 ) foo(); else bar();
```

As shown below, the `switch` construct is implemented using a dispatch table and a `jr`, and other jumps. The `switch` construct itself and the case blocks use no branches.

```
 # Part of code implementing switch construct.
 # Value in register $t1 has been computed using variable c.
 lw $t0, 0($t1)   # Load the address of the case statement corresponding to c.
 jr $t0           # Jump to case statement.
 nop
# Case 'a'
 addi $t5, $0, 3  # x = 3;
 j endswitch
 addi $t7, $t7, 1 # j++
...
```

(*b*) The predictors covered in class would all achieve just a 50% prediction accuracy on the *if* branch. Explain why they might do better if the case statements contained branches, but the values assigned to `x` are the same as the example above and the if branch is the same. *Note: The original exam did not have this part.*

☑ Why predictors might do better on *if* branch when cases have branches.

They might be able to tell which case statement was executed by the pattern of recent branch outcomes.

(*c*) Modify one of the predictors used in class so that it does better than 50% on the if branch. The predictor should also work well on similar switch statements and on predict well on code not having switch statements. Don't design a predictor for exactly the code above. For example, the predictor shouldn't somehow find or guess the value of `x`. *Hint: Think about the answer to the previous part. A correct solution requires just a small modification of one of the predictors shown in class.*

☑ Briefly explain the idea behind your predictor.

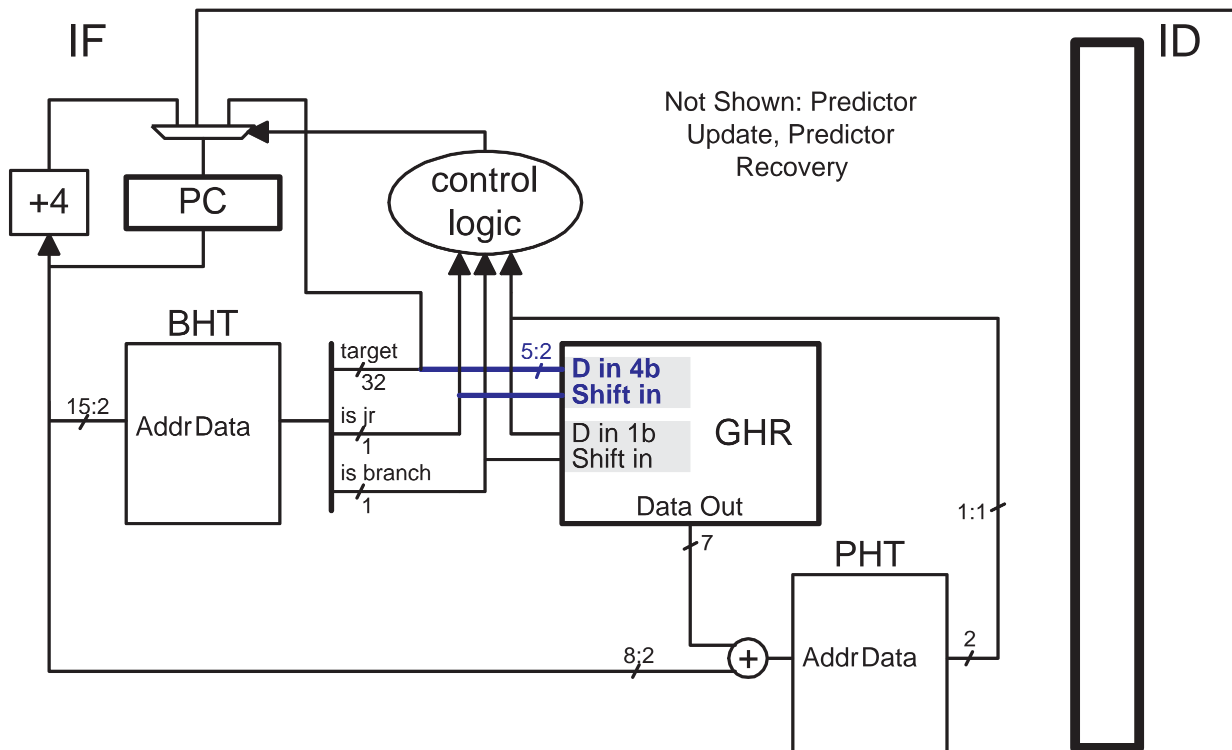
If the predictor could determine which case statement was executed then it could better, or maybe perfectly, predict the branch. A global or gshare predictor "knows" the outcome of prior branches (they are in the GHR) and uses that to make a prediction. Here the predictor needs to know which case statement was executed, or equivalently what the target of the `jr` instruction was. So a predictor for the code above would shift `jr` target addresses in the GHR, just as branch outcomes are shifted into the GHR. The entire target address would be too large, but the lower order bits, say 4 of them, might do.

☑ Draw a diagram of the predictor.

☑ Show tables such as BHT and PHT.

The diagram below is a gshare predictor modified so that it would accurately predict the branch. The material in blue shows changes specific to this problem. The size of the BHT and the size of the local history match the first part of this problem.

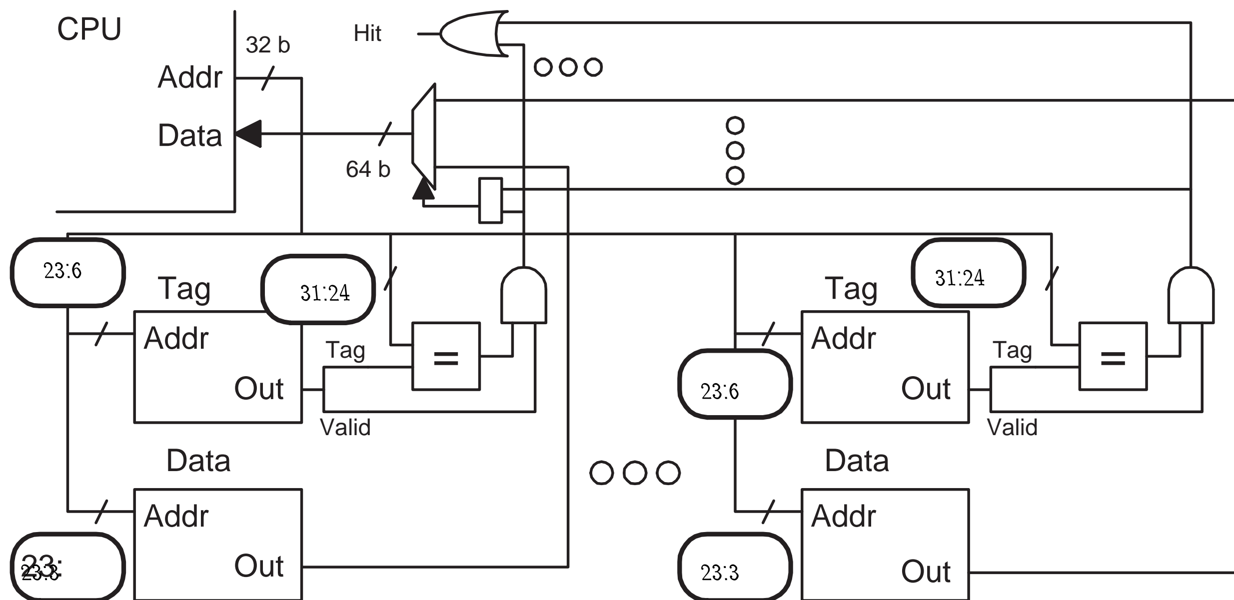
In a more realistic predictor the BHT would just store a branch displacement (16 bits) and a separate jump target buffer (JTB) would hold 30-bit addresses (or maybe just the other 14 bits) for jumps.



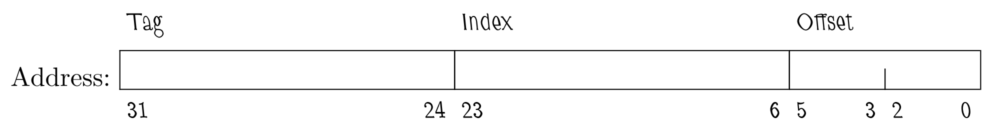
Problem 3: (20 pts) The diagram below is for a 256-MiB (2^{28}-character) set-associative cache with a line size of 64 characters on a system with the usual 8-bit characters.

(*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.)



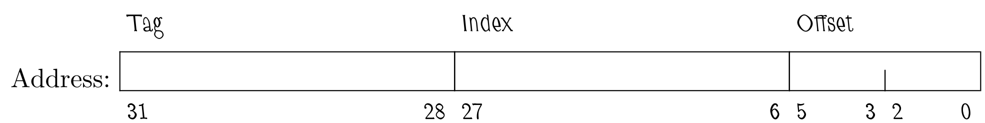
☑ Associativity:

Based on the high bit position used for the data store, 23, the data store size is 2^{24} characters. Since the total cache capacity is 2^{28} characters the cache must be $\frac{2^{28}}{2^{24}} = 16$-way set associative.

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

It's the cache capacity plus $16 \times 2^{24-6}(32 - 24 + 1)$ bits.

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



Problem 3, continued:

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

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

```
double sum = 0.0;
long *a = 0x2000000;     // sizeof(long) = 8 characters.
int i;
int ILIMIT = 1 << 10;     // = 2^10

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

The line size of 64 characters is given, the size of an array element is 8 characters. The miss on the first iteration will bring in a line which is 64 characters or 8 `long`s, the first `long` will be accessed. The second iteration will access the second `long` on this line. The line will be "used up" at $i = \frac{64}{8} = 8$, and so for each miss there are 7 hits. **The hit ratio is $\frac{7}{8}$.**

(c) Consider a 1 MiB (2^{20} byte) *direct mapped cache* with a line size of 256 characters (not the same as the one from parts a and b) for a system with a 32-bit address space. Suppose this cache has the following defect: a particular bit position in the tag comparison unit will match even if the tags differ. That is, if the bad bit position were 2 then tags `0x5` and `0x1` would match. Other cache hardware functions correctly. The cache is write through and write around.

☑ Complete the program below so that it finds the bad bit position in a small amount of time and assigns it to `badbit`.

☑ For maximum partial credit briefly describe your strategy.

- The cache is empty when the program starts.
- Assume that any address can be read or written.

```
char *a = 0x1000;
int bad_bit = -1;  // At end should be set to position of bad tag bit.


// SOLUTION
unsigned int i;

// Initialize elements to zero.
// Since cache is write-around these initialized elements won't be cached.
// Tag values are: 1, 2, 4, ..., 4096
//
for ( i=20; i<32; i++ ) a[ 1 << i ] = 0;

// Initialize this element to 1.
// Tag value is 0.
a[0] = 4720;
int dummy = a[0]; // Make sure a[0] is in cache.

// Each element fetch should miss the cache and find a zero. If
// the tag comparison is wrong then it will hit and find the 4720.
for ( i=20; i<32; i++ )
 if ( a[ 1 << i ] == 4720 ) break;

if ( i != 32 ) bad_bit = i - 20;
```

Problem 4: Answer each question below.

(*a*) (6 pts) A trap instruction is sort of like a procedure call (*e.g.*, `jal`) to the operating system.

☑ Describe a difference between a trap and `jal` in how the target address is specified.

A `jal` specifies the low bits of the target address. A trap does not specify any address at all, the target address is a particular entry in the trap table, and the trap table address is defined by the MIPS ISA and built into the hardware.

☑ Describe another important difference between a trap and a `jal`.

A trap switches the processor to privileged mode.

(*b*) (6 pts) A `log` (logarithm) instruction is to be added to an ISA. Group E wants to define the `log` instruction as producing the IEEE 754 double representation that is closest to the exact result. Group A would define `log` as producing any result within a certain number of bits of the exact result. Group A argues that the precision of an exact result is not needed and their approximate result is sufficient. *Group E agrees with this,* they want an exact result for other reasons. *Hint: Think about the reasons for separating ISA and implementation.*

☑ Why might group A want an approximate result?

It can be computed faster than an exact result.

☑ Why might group E want an exact result?

Code would run exactly the same way on different implementations. (Note that the A-group definition of `log` would allow to implementations to produce different results, so long as they were close enough to an exact result.)

(*c*) (6 pts) Consider two scalar MIPS implementations, implementation A is similar to the one covered in class with the familiar stages `IF ID EX ME WB` while implementation B has stages `IF I1 I2 I3 I4 EX ME WB`. The two implementations run at the same clock frequency and are similar in other ways.

☑ Explain why implementation A does not need a branch predictor, or at best would only gain a small amount of performance.

Because the branch target and direction are available in time for `IF`, so long as there is no unbypassable dependence. Consider the code execution below which is on a processor with an aggressive EX-to-ID bypass for the branch condition. The branch has the direction and target ready in cycle 2, which is in time for the fetch of the target which starts in cycle 3. There is no need for a branch prediction. If the branch condition were from a load instruction (or if the aggressive bypass were not possible) the branch would have to stall and so branch prediction would help, but only a little.

```
 # SOLUTION example
 # Cycle            0  1  2  3  4  5  6  7
 sub r1, r2, r3     IF ID EX ME WB
 bneq r1,r5  TARG      IF ID EX ME WB
 xor r9, r10, r11         IF ID EX ME WB
TARG:
 add r12,r13,r14             IF ID EX ME WB
```

☑ Explain why a branch predictor would help implementation B much more than implementation A.

Because the branch would not be resolved until after several instructions past the delay slot instruction were fetched, these instructions would have to be squashed if the prediction were wrong. In the example below three instructions, `or` and two others not shown, are squashed because the branch is mispredicted not-taken.

```
 # SOLUTION example
 # Cycle            0  1  2  3  4  5  6  7
 sub r1, r2, r3     IF I1 I2 I3 I4 EX ME WB
 bneq r4,r5  TARG      IF I1 I2 I3 I4 EX ME WB
 xor r9, r10, r11         IF I1 I2 I3 I4 EX ME WB
 or                          IF I1 I2x
TARG:
 add r12,r13,r14             IF I1 I2 I3 I4 EX ME WB
```

☑ Consider the performance of implementation A and implementation B when branch prediction is perfect for both. Which (if any) is faster, and by how much? Explain, state any assumptions made.

They would be the same performance because the only other events that could slow execution, dependence stalls, would be just as frequent and just as long.

(*d*) (6 pts) The code below executes on a two-way superscalar dynamically scheduled machine similar to the one presented in class. The `sub` instruction reads the value of `r1` from the register file in cycle 10, `xori` writes a value for `r1` in cycle 6 and `lw` writes a value for `r1` in cycle 9.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12
lw r1, 0(r2)     IF ID Q  RR EA          ME WB C
add r3, r9, r1   IF ID Q                 RR EX WB C
or  r6, r3, r8      IF ID Q                 RR EX WB C
xori r1, r4, 5      IF ID Q  RR EX WB             C
sub  r5, r1, r3        IF ID Q                 RR EX WB C
```

☑ Briefly explain why the code runs correctly despite the fact that `lw` writes *after* `xori`.

Because registers are renamed, meaning `lw` writes a different physical register than `xori` and so there is no confusion.

(*e*) (6 pts) Modern VLIW ISAs are designed with modern implementations in mind, unlike decades old RISC ISAs.

☑ Show the contents of a typical VLIW bundle.

A bundle typically contains three RISC style instructions plus some dependency information.

☑ Provide an example of a VLIW feature that's designed to make implementation easier. Explain how it does so.

The dependency information eases design of the control logic, perhaps shortening critical paths.