Name Solution

Computer Architecture
EE 4720
Final Examination

7 May 2009, 17:30–19:30 CDT

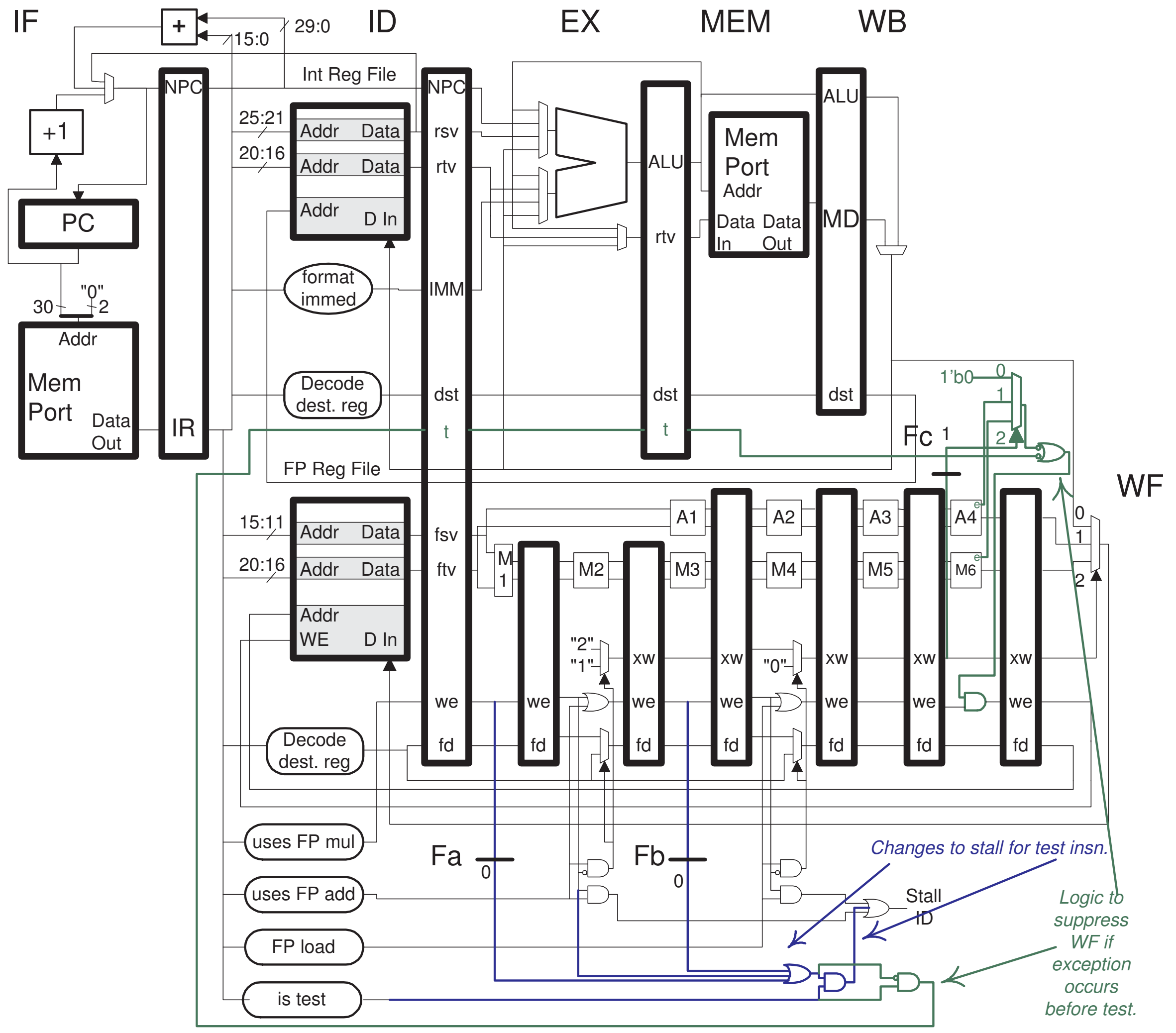
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias A(H1N1) · · · chooo!

Good Luck!

Problem 1: (20 pts) The MIPS implementation below, taken from the solution to last semester's final, includes hardware to implement an exception test instruction.

Several wires on the diagram are broken with heavy solid lines and marked with `Fx` and a value (mostly `0`). These indicate *potential* fault locations. If there is no fault the wire acts normally, if there is a fault the wire is broken at the heavy line and the free half takes on the indicated value. For example, if fault `Fa` is present the bottom input to the OR gate is always zero however the `ID/EX.we` signal on the other side is unaffected.



The problem here is to detect which fault, if any, is present by running test programs. One test program, and a pipeline diagram, appears below. A handler has been set up that will set a `goodException` variable to 1 if register and memory values are as expected, otherwise it is set to -1. The `goodException` variable is initialized to 0 before each test.

Problem 1, continued:

```
# Test 1: The mul should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 * f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
mul.s f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6x
test                  IF ID -------> EX MEx
sw r1,0(r2)              IF -------> ID EXx
```

(*a*) When a test is run the exception handler is called (because the tests intentionally raise an exception). A handler is shown below, written in C, but the handler does not set `goodException` correctly (not even close). Modify the code so that `goodException` is set correctly based on the information provided by the Test 1 code and comments. That is, `goodException` is set to -1 if the exception could not be precise.

```
int handler_fp(Regs *regs){
 // Code below wrong, but shows how to read registers and memory.

 // SOLUTION BELOW (Including commented-out line.)
 // if ( regs->f10 == regs->f12 && is_nan(f14) && MemW(regs->r31) == 0x1234 )
 if ( regs->f0 == 1 && MemW(regs->r2) == 50 )
  goodException = 1; else goodException = -1;

}
```

The exception raised by an instruction is precise if it appears that execution cleanly reached the instruction and then jumped to the handler (without executing the instruction). The solution, above, checks for two problems that might occur when the exception originates in Test 1 (or other routines specially written for this handler): the faulting instruction should not write a register (`f0` in this case) and no following instruction should change a register or memory. The `regs->f0 == 1` condition makes sure the multiply did not write back (it would have written back a non-numeric value) and the `MewW(regs->rs) == 50` condition makes sure that the store did not execute (it would have changed the memory location to 20).

(*b*) Suppose Test 1 is run and `goodException` is set to -1 (it would be 1 in the no-fault case). Which of the faults (Fa, Fb, or Fc) could have been responsible for the `goodException` value?

☑ Faults that are definitely present. Explain.

None. Any of the faults could be present. If `Fa` or `Fb` were present then the multiply would stall one less cycle than necessary allowing the `sw` to write memory. If `Fc` were present then the `mul` would write memory (assuming that the adder exception output was 0).

☑ Faults that could be present. Explain.

See solution above.

Problem 1, continued:

(*c*) Develop tests to determine for certain whether each of the faults is present. Test 1 and each of your tests will be run, and based on the `goodException` values from each test one can say for certain which faults are present.

Assume that at most one of the faults is present. Your tests should look similar to Test 1.

☑ Tests (Code like Test 1)

```
# SOLUTION
#
# Test 2: Will only fail if Fc is present.
#         The mul should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 * f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
mul.s f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6x
test                  IF ID -------> EX MEx
nop                      IF -------> ID EXx

# Test 3: Will only fail if Fb is present.
#         The add should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 + f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
add.s f0, f2, f4   IF ID A1 A2 A3 A4x
test                  IF ID -> EX MEx
sw r1,0(r2)              IF -> ID EXx
```

Test 2 above is like Test 1 except there is no `sw`, so fault Fa and Fb won't cause incorrect operation, but Fc still will. Note that Test 2 would not be necessary if the handler could set a variable indicating the reason for failure (either the `sw` writing memory or the faulting instruction writing `f0`).

☑ Which combination of `goodException` values conclude `Fa` for certain.

`Fa` is present if Test 1 fails (`goodException`=-1) and Tests 2 and 3 pass (`goodException`=1). See the discussion about the Test programs.

☑ Which combination of `goodException` values conclude `Fb` for certain.

Test 3, `goodException`=-1. Test 3 can't be affected by `Fa` because the add instruction never passes through the `M1` stage and it can't be affected by `Fc` because the add uses input 1 to the multiplexor anyway.

☑ Which combination of `goodException` values conclude `Fc` for certain.

Test 2, `goodException`=-1. Since there is no `sw` after the `test` instruction it can't fail because of there were not enough stalls, the only possibility is `Fc`.

Problem 2: (20 pts) Answer each question below. **Be sure to check each code fragment carefully for dependencies.**

(*a*) The loop below runs on a statically scheduled 4-way superscalar MIPS implementation.

☑ Show a pipeline execution diagram.

```
#                   SOLUTION
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
addi r2, r2, 8      IF ID EX ME WB
lw r1, 0(r2)        IF ID -> EX ME WB
add r3, r1, r4      IF ID -------> EX ME WB
bneq r2, r5 LOOP    IF ID -------> EX ME WB
sw r3, 4(r2)           IF -------> ID EX ME WB
(XXX)
addi r2, r2, 8                     IF ID EX ME WB
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

☑ Determine the IPC for a large number of iterations and assuming no cache misses.

Iterations start at cycle 0 and 5. There are no reasons why the second iteration would be any different than the first so the $\boxed{\text{execution rate is } \frac{5}{5} \text{ IPC.}}$

☑ Comment on the difference between the IPC and the potential IPC of the processor.

The potential is 4 but we only get 1. That's very inefficient for code without cache misses.

☑ Schedule the code to improve execution time.

```
# First iteration + 0x8
addi r2, r2, 8
lw r1, 0(r2)
add r3, r1, r4

# Second iteration + 0x10
addi r2, r2, 8
lw r1, 0(r2)

LOOP: # Cycles      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw r3, -4(r2)       IF ID EX ME WB
add r3, r1, r4      IF ID EX ME WB
lw r1, 8(r2)        IF ID EX ME WB
bneq r2, r5 LOOP    IF ID EX ME WB
addi r2, r2, 8         IF ID EX ME WB
(xxx) # Cycles      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw r3, -4(r2)             IF ID EX ME WB
add r3, r1, r4            IF ID EX ME WB
lw r1, 8(r2)              IF ID EX ME WB
bneq r2, r5 LOOP          IF ID EX ME WB
addi r2, r2, 8               IF ID EX ME WB
```

The scheduled code appears above. Execution is shown on an implementation that can bypass a value from `EX` to the branch condition comparison unit in `ID`. Work that was done in a single iteration in the original code is spread out over three iterations in the scheduled code above. For example, the value loaded by the `lw` in iteration 1 is used by the `add` in iteration 2 and the sum produced by the `add` in iteration 2 is used by the `sw` in iteration 3. This technique is called software pipelining.

Problem 2, continued:

(*b*) The code below (same as the previous problem) executes on a 4-way superscalar dynamically scheduled machine. Assume that branch prediction on this machine is perfect. Load instructions use the `EA` and `ME` stages, branch instructions use the `B` stage, store instruction only use `EA` (they write memory when they commit).

Though the machine is 4-way, assume an unlimited number of `WB`, `EX`, and `RR` stages.

☑ Show a pipeline execution diagram for two iterations.

```
# SOLUTION
LOOP: # Cycles    0  1  2  3  4  5  6  7  8  9  10 11
addi r2, r2, 8    IF ID Q  RR EX WB C
lw r1, 0(r2)      IF ID Q     RR EA ME WB C
add r3, r1, r4    IF ID Q           RR EX WB C
bneq r2, r5 LOOP  IF ID Q     RR B  WB       C
sw r3, 4(r2)         IF ID Q  RR EA WB       C
(XXX)
addi r2, r2, 8          IF ID Q  RR EX WB    C
lw r1, 0(r2)            IF ID Q     RR EA ME WB C
add r3, r1, r4          IF ID Q           RR EX WB C
bneq r2, r5 LOOP        IF ID Q     RR B  WB       C
sw r3, 4(r2)               IF ID Q  RR EA WB       C
LOOP: # Cycles    0  1  2  3  4  5  6  7  8  9  10 11
```

☑ Determine the IPC for a large number of iterations.

The execution rate is $\frac{5}{2} = 2.5$ IPC.

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^{14}-entry BHT. One system has a bimodal predictor, one system uses a local history predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history.

The code has three branches, B1, B2, and B3. The outcome of B1 is random, described by a Bernoulli random variable with $p = .5$ and is independent of everything. Branch B2 has a simple j-iteration loop pattern ($j - 1$ T's and an N) and branch B3 is a repeating pattern of k N's followed by k T's (see diagram below). Also from the diagram notice that B1 occurs just before B2 but that a set of j B1 and B2s occur between each B3, (similar to a branch in the homework and last semester's final).

```
               A                    A                    A
     ------------------   ------------------   ------------------   ...
B1:  R  R  R  ...  R      R  R  R  ...  R      R  R  R  ...  R
B2:   T  T  T ...T   N     T  T  T ...T   N     T  T  T ...T   N
      !<---  j    --->!

B1&2:   A   A   A      A
B3:       N   N   N  ...  N   T   T   T  ...  T   N   N   N ...  N   T...
          !<---  k Ns  --->!  !<--- k Ts  ---->!  !<--- k Ns  --->!  ...
```

For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal on B1?

The branch is random and uncorrelated so there is no way prediction accuracy can be anything other than $\boxed{0.5}$.

☑ What is the accuracy of the bimodal on B2 in terms of j?

The Ts will all be correctly predicted the Ns will all be mispredicted, so the $\boxed{\text{accuracy is } \frac{j-1}{j}}$, for $j = 5$ that would be $\frac{4}{5} = 0.8$.

☑ What is the accuracy of the bimodal on B3 in terms of k?

When $k > 2$ there will be two mispredicts after each change from N to T or T to N. Therefore $\boxed{\text{the accuracy is } \frac{k-2}{k}}$. For $k = 4$ the accuracy is 0.5.

☑ What is the accuracy of the local predictor on B3 when $k \gg 10$ and $j < 5$ in terms of j and k?

When $j < 5$ the patterns generated by B2 cannot match the patterns generated by B3. The only PHT entries that will result in a misprediction of B3 are the all Ns, and all Ts, and they will only be wrong just at the N to T or T to N transition. Therefore $\boxed{\text{the accuracy is } \frac{k-1}{k}}$.

☑ What is the accuracy of the local predictor on B3 when $k \gg 10$ and $j > 10$ in terms of j and k?

With $j > 10$ both B2 and B3 will have local history TTTTTTTTTN. For B2 the next outcome is T, but for B3 the next outcome is N. B2 will generate the local history far more frequently and so it will keep the respective PHT entry at 3. So now branch B3 will mispredict at the transition (where its local history will be all Ns or all Ts) and at pattern TTTTTTTTTN. It's overall $\boxed{\text{accuracy will be } \frac{2k-3}{2k}}$.

☑ What is the accuracy of the global predictor on B3 when $j = 10$ and k is very, very large, in terms of j and k?

Because the local history size is 10 and $j = 10$ the GHR (global history register) contents seen when predicting B3 does not contain any past outcomes of B3, it only contains B1 and B2 outcomes, neither of which are helpful. Therefore at each transition there will be mispredicts until the PHT entries warmup, after which accuracy will be 1.0 until the next transition. If k is very very large we can say that the $\boxed{\text{accuracy will be very very close to 1.0}}$. Not satisfied? See the next problem.

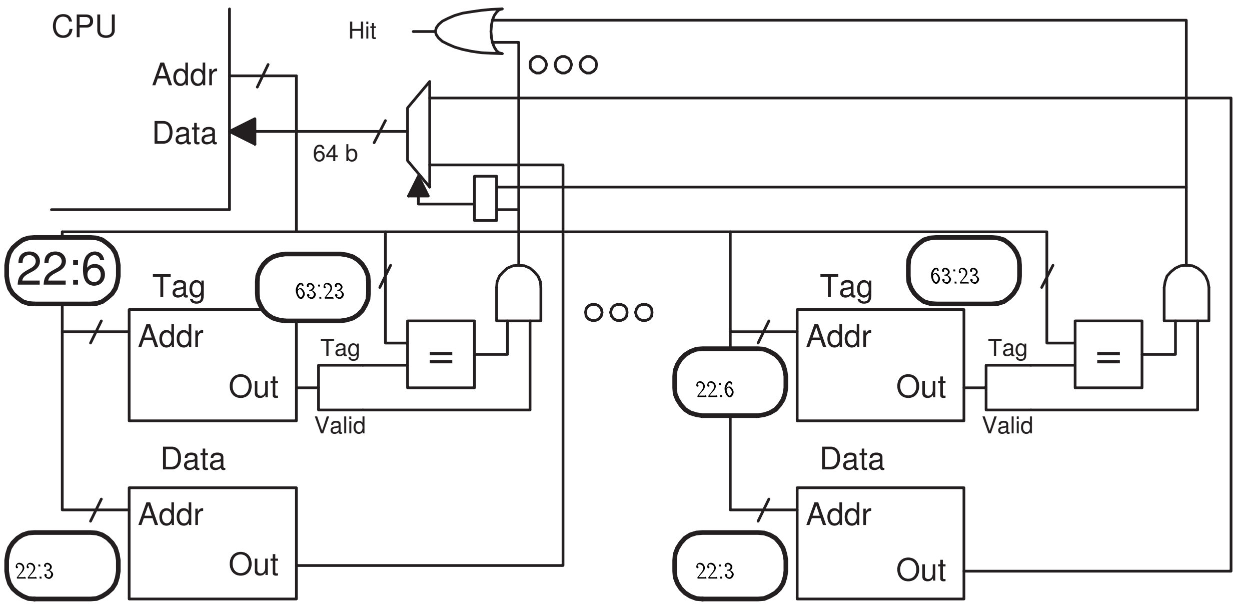
☑ What is the warmup time for B3 on the global predictor. (This warmup occurs whenever B3 switches from Ns to Ts or Ts to Ns.) in terms of j and k?

The global history encountered when predicting B3 contains 5 B1 outcomes and 5 B2 outcomes. The GHR contents will be **rTrTrTrTrN**, where **r** are the B1 outcomes and the other elements are B2 outcomes. Since B1 is random the **r** can be either N or T. There are five of them and so there are $2^5 = 32$ patterns of this form. Each corresponding PHT entry needs to warm up on a B3 transition, which requires two occurrences. The $\boxed{\text{warmup time is then } 2 \times 2^5}$ and the accuracy of the global predictor is $\frac{k-2\times 2^5}{k}$.

Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25}-character) set-associative cache with the usual 8-bit characters and a 64-bit address space.

(*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.)

	Tag	Index	Offset
Address:	63 … 23	22 … 6	5 3 2 0

☑ Associativity:

From the given tag store address bits, 22:6, we know the data store has a capacity of 2^{23} characters so for a cache capacity of 2^{25} characters the $\boxed{\text{associativity must be } \frac{2^{25}}{2^{23}} = 4}$.

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

It's the cache capacity, 2^{25} characters, plus $4 \times 2^{23-6}\,(64 - 23 + 1)$ bits.

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

	Tag	Index	Offset
Address:	63 … 25	24 … 6	5 3 2 0

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

$\boxed{\checkmark}$ What is the hit ratio running the code below? Explain

```
  double sum = 0.0;
  char *a = 0x2000000;
  int i;
  int ILIMIT = 1 << 11;    // = 2^11

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

The line size of 2^6 characters is given, the size of an array element is 1 character, and so there are 2^6 elements per line. The first access, at `i=0`, will miss but bring in a line with 2^6 elements, the next $2^6 - 1$ accesses are to subsequent elements on the line and so will hit, so the hit ratio to the first line is $\frac{2^6-1}{2^6}$.

(*c*) The code below runs on a direct-mapped cache unrelated to the one above. When the code starts running the cache is cold, for the solution only count accesses to `array`.

```
struct My_Struct {
  double val;
  double something;
  double relative;
  double more_data[29];
};  // Total size: 32 * sizeof(double) = 256 bytes

const int SIZE = 1 << 12;
My_Struct array[SIZE];  // &array[0] = 0x1000000

void tri()
{
  double sum = 0;
  for ( int i=0; i<SIZE; i++ ) sum += array[i].val;
  const double avg = sum / SIZE;
  for ( int i=0; i<SIZE; i++ ) array[i].relative += array[i].val - avg;
}
```

$\boxed{\checkmark}$ Determine the minimum cache and line size needed so that there are no misses in the second `for` loop.

The $\boxed{\text{minimum line size is 32 characters}}$, the $\boxed{\text{minimum cache size 128 kiB}}$.

$\boxed{\checkmark}$ Explain your answer.

Both `for` loops iterate over the entire array, the first `for` loop only accesses the `val` member but the second `for` loop accesses both the `val` and `relative` member. The line size will be chosen so that on a miss in the first `for` loop both members will be on a line. If line size were not restricted to a power of two then a line size of 24 characters would be sufficient, but they are so the minimum line size is 32 characters. Such a line can hold both `val` and `relative`, but whether it will depends on the alignment of the structure. From the given address of `array[0]` (which is the address of `array[0].val`) the offset bits are obviously zero (meaning it is at the beginning of a line). Since the struct size is a multiple of 32 `array[i].val` will also be on the beginning of a line.

The number of lines needed to store the array is 2^{12} and so the total $\boxed{\text{cache size needed is } 32 \times 2^{12} = 2^{17}}$ or 128 kiB.

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

(*a*) In a dynamically scheduled machine one would like to be able to have a large number of instructions in flight to, say, find something to do while waiting for data from memory. Which part of the dynamically scheduled machine is most difficult to scale up to support a large number of in-flight instructions?

☑ Part that's difficult to scale, and reason why.

The scheduler. It needs to monitor the status of the source registers of every instruction in the instruction queue so that it can find instructions that are ready for execution. It does this each cycle based on the destination registers that will be written by the earlier rounds of instructions. It is difficult to do this quickly for a large number of instructions.

(*b*) How might profiling improve the performance of the following C code:

```
if ( a > b ) { x = d / e; } else { y = q / f; }
```

☑ Explain how profiling is used.

First the code is compiled with a special profiling switch. It is then run on what is hoped to be typical input data, while running it will record branch direction counts and other information to a profile file. The code will be compiled a second time using another profiling switch. The compiler will read the profile file and use that information to make optimization decisions. See the next answer.

☑ Explain why this profiled code might be faster than code compiled without profile feedback.

A taken branch or jump slows execution more than a not-taken branch, even with correct prediction. Based on a profile run the compiler might learn that in the `if` statement above the `else` part is rarely executed. It will organize the code so that two taken branches are needed to execute the `else` part but zero branches (or jumps) are needed for the `if` part.

(*c*) In the first implementation of a company's ISA the integer multiply instruction was slow. An Engineer working on the second implementation is deciding whether to speed up the multiply instruction. To decide he plans to analyze some benchmark runs, but he can't decide whether the code should be optimized. The compiler was designed for the first implementation.

The engineer is trying to determine if the multiply instruction is used often enough to justify the effort to improve it. To do that he runs benchmark programs using some kind of simulator that can tell him how many times a multiply was used while running the code. If, say, the multiply accounts for less than 0.00001% of executed instructions then it's not worth the effort.

☑ What is the disadvantage of counting multiply usage in code compiled with optimization?

Since the multiply was slow in the first implementation the compiler might have used it less often, perhaps substituting adds and shifts. However, the compiler would only look for alternatives to a multiply if optimization were turned on. With optimization off the compiler generates straightforward code without tricks.

Since the compiler is avoiding use of the multiply instruction the number of multiplies would be undercounted. That would be bad if the compiler's alternative were slower than the speeded-up multiply instruction in the second implementation.

☑ What is the disadvantage of counting multiply usage in code compiled without optimization?

Without optimization there might be too many multiplies. Techniques like dead-code elimination and constant folding would eliminate multiplies no matter how fast the instruction is.

(*d*) A memory system that can fetch 2^w chars of data is less expensive and faster if the data address is a multiple of 2^w. (That's one reason for the alignment restriction.)

☑ How do VLIW ISAs take advantage of that?

Instruction bundle sizes are set to a power of two and control-transfer instructions can only jump to the beginning of a bundle. Therefore the memory port used for instruction fetch can enjoy the cost and performance benefits of alignment.

(*e*) Compared to a RISC implementation, say the 5-stage MIPS, what additional logic does a CISC implementation require between the IF-stage memory port output and decode?

☑ Additional logic for CISC.

Since instruction length varies it will need logic to shift the instruction into position and logic to combine the bits fetched in one cycle with a piece of the instruction fetched in an earlier cycle.