Name Solution

Computer Architecture
EE 4720
Final Examination

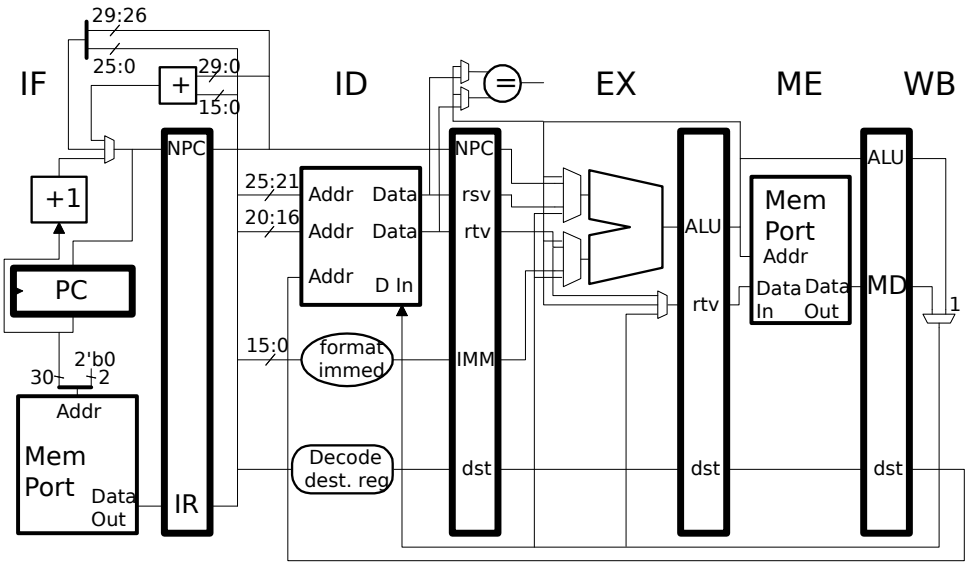
9 May 2011, 15:00–17:00 CDT

Problem 1		(20 pts)
Problem 2		(10 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

Alias Mississippi Rising

Good Luck!

Problem 1: (20 pts) The MIPS implementation below is similar to the one frequently used in class, except that it has bypass paths to the branch condition comparison unit.



(*a*) Show the execution of the code fragments below on the illustrated implementation up until the fetch of the first instruction of the second iteration. Be sure to account for the dependence on the branch condition.

☑ Pipeline execution diagram of code below.

```
#          SOLUTION
LOOP1:   # Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)        IF ID EX ME WB
 addi r2, r2, 4         IF ID EX ME WB
 bne  r2, r3 LOOP1         IF ID -> EX ME WB
 add r4, r4, r1               IF -> ID EX ME WB
#          Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)                        IF ID EX ME WB
```

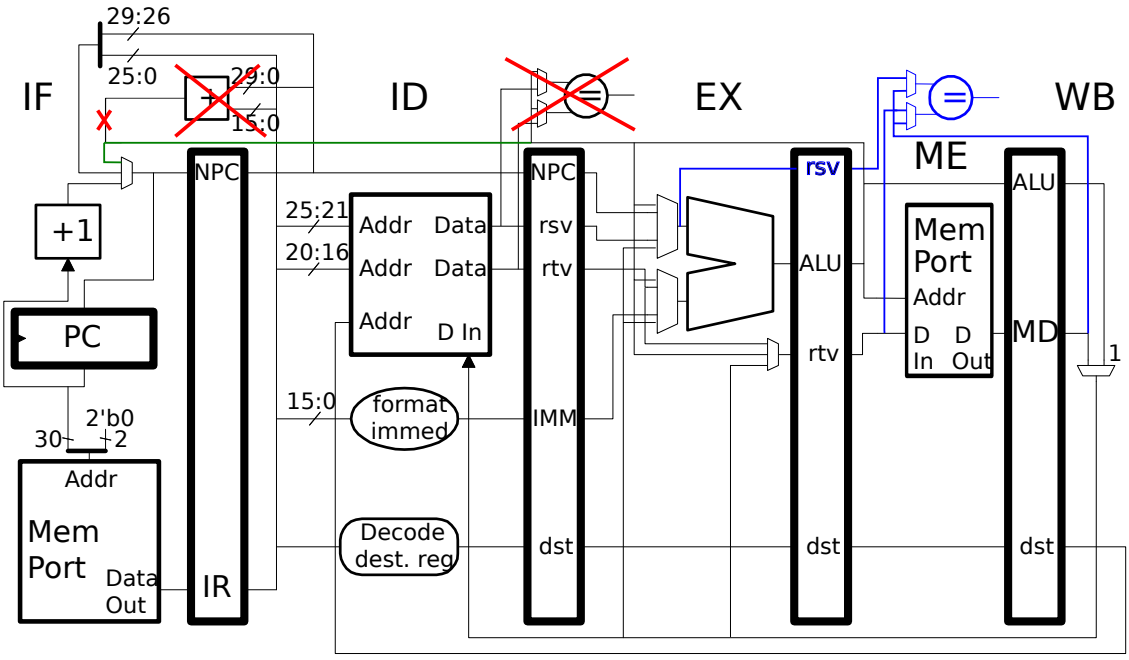
The solution appears above. Note that because of the dependence with the `addi` instruction the branch must stall one cycle, until cycle 4, at which time the branch can use the bypass from ME to ID.

☑ Pipeline execution diagram of code below.

```
#       SOLUTION
LOOP2: # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)       IF ID EX ME WB
 lw r2, 4(r2)          IF ID EX ME WB
 bne  r2, r0 LOOP2        IF ID ----> EX ME WB
 add r4, r4, r1               IF ----> ID EX ME WB
#       # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)                           IF ID EX ME WB
```

Solution appears above. Because the branch is dependent on a load, the `lw r2`, the bypass doesn't help and so the branch must stall until `lw r2` reaches WB, in cycle 5. At that time the load value will be passed through the register file to the branch condition comparison unit (the circled equal sign).

(*b*) The implementation below has hardware in `IF` (not shown) to predict branches. Since it has prediction hardware the branch resolution hardware (the hardware that computes the branch condition and branch target) could be moved to the `ME` stage. Move the branch resolution hardware. Note that the resolution hardware will be used only if the branch is mispredicted.



☑ Move resolution hardware to the `ME` stage.

☑ Add any bypasses to resolution hardware needed by code samples in this problem.

☑ Cross out unused hardware in `ID`.

☑ Avoid adding unnecessary hardware.

Solution appears above.

The comparison unit has been moved to the `ME` stage, where it gets inputs from an added `ME.rsv` latch and the existing `ME.rtv` latch. In the figure the old comparison unit is crossed out and the new (or moved) comparison unit and new connections for it appear in blue. Those connections provide bypasses of values from the two preceding instructions, but won't work if the immediately preceding instruction is a load. For that a bypass has been added from the `WB.MD` latch. (The output of the `WB`-stage mux could also be used, but any value in `WB.ALU` could also have been bypassed when the branch was in `EX`.

The `ID`-stage adder for computing the branch target has been removed. Instead, the ALU will compute the branch target using the existing `EX.NPC` and `EX.IMM` inputs. A new connection from `ME` to the `IF`-stage mux has been added so the branch target can reach the `PC`, that appears in green.

Problem 1, continued: Consider the implementation from the previous part.

(*c*) Resolving the branch in ME with prediction can hurt performance with the code below compared to resolving in ID with prediction when the branch is hard to predict. By how much will it hurt performance? *Note: An exact number was not required on the original exam.* Explain, use a diagram if necessary. *Hint: Resolve is where recovery starts for a misprediction. Another hint: pay attention to dependencies on the branch condition.*

```
#         SOLUTION
LOOP1:  # Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)       IF ID EX ME WB
 addi r2, r2, 4        IF ID EX ME WB
 bne  r2, r3 LOOP1        IF ID EX ME WB
 add r4, r4, r1               IF ID EX ME WB
 xor                             IF IDx
 or                                 IFx
#         Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)                          IF ID EX ME WB
```

☑ Resolving in ME hurts performance here by __1__ cycles.

☑ Explanation.

Because the branch resolution hardware is in ME, when there is a misprediction the correct path instruction will only reach IF when the branch is in WB (the cycle after ME). For the code fragment above this occurs in cycle 6. Compare this to the execution of the code in part (a) of this problem, in which the branch target is fetched in cycle 5. So performance is hurt by one cycle.

Note that though on a misprediction execution takes one cycle longer, when there is not a misprediction execution takes one cycle less since the dependence stall is avoided. See the next part for an example of that.

(*d*) Resolving the branch in ME with prediction should help performance with the code below compared to resolving in ID with prediction. Explain why, preferably with an execution diagram. *Hint: Same hints as previous part.*

```
#      SOLUTION
LOOP2: # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)       IF ID EX ME WB
 lw r2, 4(r2)          IF ID EX ME WB
 bne  r2, r0 LOOP2        IF ID EX ME WB
 add r4, r4, r1               IF ID EX ME WB
#       Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)                     IF ID EX ME WB
```

☑ Resolving in ME helps performance because...

. . . because we can bypass the load value, in cycle 5 for the example, and so avoid stalls. Contrast this with the execution of the fragment in part (a) in which there is a two cycle stall every iteration. With prediction there is never a dependence stall for the branch, and two-instruction squashes only occur when the branch is mispredicted.

Problem 2: (10 pts) The diagrams below show the execution of code on two-way superscalar dynamically scheduled systems of the type described in class. Each diagram **has mistakes**. Identify and fix them. Also explain why code should not execute as illustrated. The answer should specify what would go wrong, or why the system would be particularly inefficient, as appropriate.

(*a*) Find and fix the two problems with commit in the execution below, and explain why execution would be incorrect or inefficient.

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
sub r4, r5, r6   IF ID Q  RR EX WB C
or  r7, r4, r8      IF ID Q  RR EX WB    C
# Cycle          0  1  2  3  4  5  6  7  8

# SOLUTION BELOW, UNFIXED PROBLEM ABOVE.
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
sub r4, r5, r6   IF ID Q  RR EX WB    C
or  r7, r4, r8      IF ID Q  RR EX WB    C
# Cycle          0  1  2  3  4  5  6  7  8
```

☑ Fix the two problems above.

☑ Describe one error or inefficiency with execution.

Commit is supposed occur in program order, but `sub` commits before the `lw`. That's fixed in the solution. If commit did occur out of order it might not be possible to recover the correct register values (in particular the register mappings) should an instruction raise an exception. For example, if some instruction raised an exception in cycle 6 it would be too late to stop the `sub`.

☑ Describe another error or inefficiency with execution.

The code is committing at a rate of one per cycle, but since it's two-way superscalar it should be able to commit at a rate of two per cycle. (If it could only commit at one per cycle then there would be little point in having hardware to fetch, decode, and execute two per cycle.) That is also fixed in the solution.

(*b*) Find and fix the problem with the code below. Note that the processor has two FP adders, named A and B. *Hint: Check dependencies.*

```
add.d f0, f2, f4      IF ID Q  RR A1 A2 A3 A4 WB C
sub.d f6, f0, f8      IF ID Q              RR A1 A2 A3 A4 WB C
add.d f10, f11, f12      IF ID Q           RR B1 B2 B3 B4 WB C
addi r1, r1, 4           IF ID Q              RR EX WB         C

# SOLUTION BELOW, UNFIXED PROBLEM ABOVE.
add.d f0, f2, f4      IF ID Q  RR A1 A2 A3 A4 WB C
sub.d f6, f0, f8      IF ID Q              RR A1 A2 A3 A4 WB C
add.d f10, f11, f12      IF ID Q  RR B1 B2 B3 B4 WB          C
addi r1, r1, 4           IF ID Q  RR EX WB                      C
```

☑ Fix the problem above.

☑ Describe one error or inefficiency with execution.

The `addi` and the second `add.d` start execution later than they need to. Because neither of these instructions has dependencies with the others they can start execution without delay. That is shown in the solution above. Note that in this example there is no impact on execution time but if the second `add.d` were instead a divide instruction then the delayed start would slow down execution.

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 uses a bimodal predictor, one system uses a local predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history. The outcomes of branch `B1` form a repeating pattern, the outcome of `B2` can be modeled by a Bernoulli random variable with $p = .7$ (the branch is taken with probability .7), and `B3` is always taken.

```
    0  0   1   2   3   3   3   2   1     0   1   2   3   3   3   2   1 <- Bimo Ctr
B1:  N   T   T   T   T   T   N   N     N   T   T   T   T   T   N   N
         x   x               x   x         x   x               x   x  <- Bimo pred outc
B2:    R   R   R   R   R   R   R   R     R   R   R   R   R   R   R   R
B3:     T   T   T   T   T   T   T   T     T   T   T   T   T   T   T   T
```

For the questions below accuracy is after warmup.

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

The counter used to predict `B1`, and the outcome of those predictions is shown in the diagram above. (The counter is labeled `Bimo Ctr`.) Based on these outcomes the accuracy is $\boxed{\frac{4}{8} = .5 = 50\%}$.

☑ What is the approximate accuracy of the bimodal predictor on `B2`? Explain. *Hint: A correct answer would be slightly more or less (pick one) than.... For the exact answer one would need the state probability formula for a Markov chain.*

If the counter for `B2` were always 2 or 3, the accuracy would be .7. However two consecutive not-taken outcomes are far from impossible, and that would switch the prediction to not taken. Therefore the accuracy is less than .7. The exact probability that the counter value is i is $p_i = \frac{\omega-1}{\omega^4-1}\omega^i$, where $\omega = \frac{.7}{.3}$. The probability of a correct prediction is $p_2 + p_3 = .844828$. The probability of a correct prediction is then $0.7(p_2 + p_3) + 0.3(p_0 + p_1) = 0.5914 + 0.0466 = 0.638$.

☑ What is the accuracy of the local predictor on branch `B1`?

The branch `B1` pattern has a length of 8, which easily fits in the 10-outcome local history, and so $\boxed{\text{the accuracy is 100\%}}$.

☑ What is the warmup time of the global predictor on branch `B1`? Be sure to consider the effect of `B2`.

First, find the possible GHR values used to predict `B1`. The GHR patterns are: `tTrtTrtTrt`, `tTrtTrtNrt`, `tTrtNrtNrt`, `tNrtNrtNrt`, `tNrtNrtTrt`, `tNrtTrtTrt`. The lower-case `t`s are an outcome of `B3`, the `r`s are the outcomes of `B2`, and the upper-case letters are outcomes of `B1`; the rightmost outcome is the most recent. The reason these are referred to as patterns and not actual GHR values is because the outcome of branch `B2` is shown as an `r`, which can be either a `t` or `n`. Therefore for a pattern such as `tTrtTrtNrt` there are $2^3 = 8$ actual GHR values. Branch `B1` needs to be executed twice to warm up the GHR, and each subsequent GHR value needs to be seen three times to warm up the PHT entries. So the total warmup time is $3 + 6 \times 2^3 \times 2 = 99$ outcomes.

☑ What is the minimum local history size needed for a local predictor to predict `B1` with 100% accuracy?
$\boxed{\text{Five outcomes.}}$

If it were four outcomes the local history `TTTT` would occur before both a taken and not-taken outcome.

☑ What is the accuracy of a global predictor with a three-outcome global history on branch `B1`?

Branch `B1` can only be predicted using its own prior outcomes. With a global history length of 3 all `B1` can see is one of its prior outcome. If that prior outcome is `N` (making pattern `Nrt`) then the outcome will be `N` 2 out of 3 times (over a repetition of the

length-8 pattern). If that prior outcome were `T` the outcome would be `T` 4 out of 5 times. The `PHT` entries would predict the more common outcomes, so the accuracy would be $\frac{2+4}{3+5} = 0.75$.

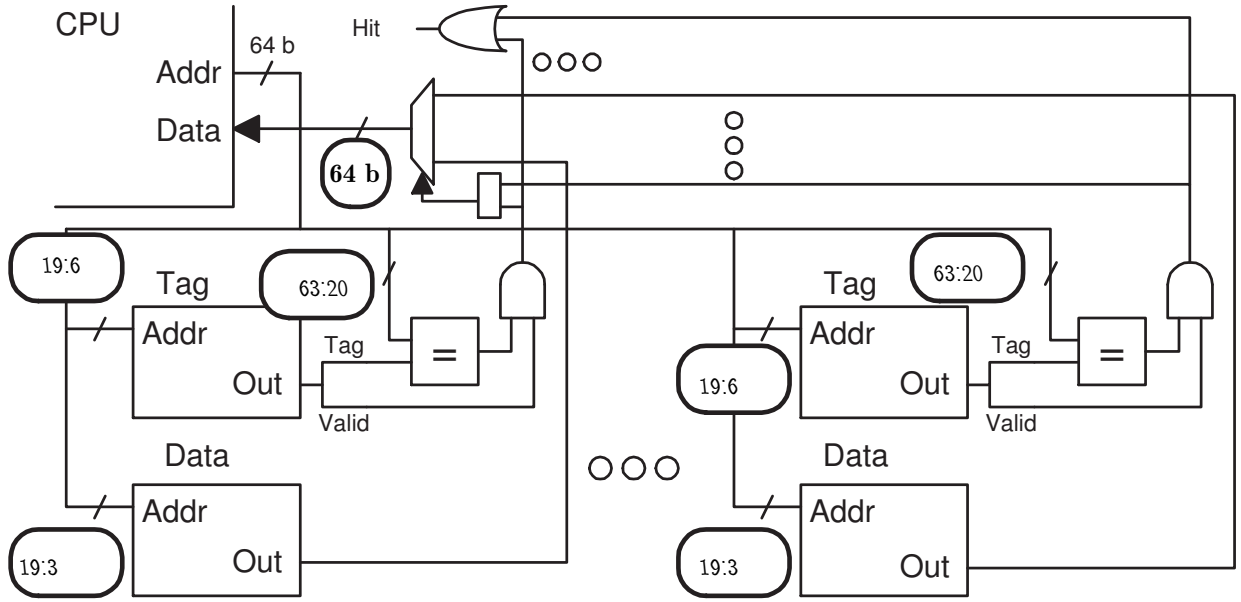
☑ What is the minimum global history size needed for a global predictor to predict `B1` with 100% accuracy?

Branch `B1` must see 5 prior outcomes to be predicted perfectly. That would require a global history length of $5 \times 3 = 15$.

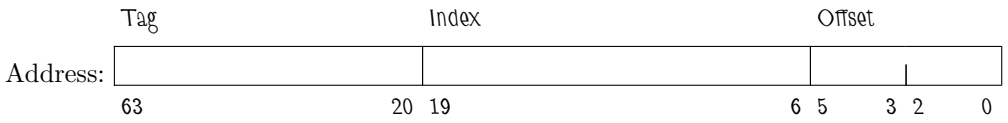
Problem 4: (20 pts) The diagram below is for a set-associative cache with a capacity of 16 MiB (2^{24} bytes), and a line size of 64 bytes. The system has the usual 8-bit characters. Each data store below has a capacity of 2^{20} bytes.

(*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.)



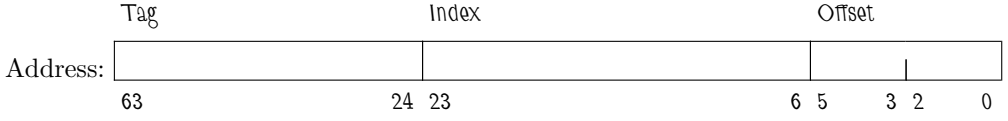
☑ Associativity:

The associativity is 16. The associativity is determined based on the given cache capacity, 2^{24} bytes, and the given capacity of an individual data store, 2^{20} bytes. There must be 16 data stores for the given cache capacity.

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

It's the cache capacity, 2^{24} characters, plus $16 \times 2^{20-6}\,(64 - 20 + 1)$ bits.

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



Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 256-byte line size. Each code fragment starts with the cache empty; consider only accesses to the arrays, `a`, `ax`, and `ay`.

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

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

```
double sum = 0.0;
double *a = 0x20000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

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

The line size of 2^8 characters is given, the size of an array element is $8 = 2^3$ characters, and so there are $2^{8-3} = 2^5$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^5 elements, and so the next $2^5 - 1$ accesses will be to data on the line. The access at `i=32` will miss and the process will repeat. Therefore the hit ratio is $\frac{31}{32}$.

(*c*) The two code fragments below, labeled Method 1 and Method 2, perform the same computation but organize the data differently.

```
// Method 1
 struct Our_Struct { int x;  int y; };
 Our_Struct a[SIZE];

 for (int i=0; i<SIZE; i++) sum += a[ i ].x;
 for (int i=0; i<SIZE; i++) myfunc(sum,a[ i ].y);

// Method 2
 int ax[SIZE];
 int ay[SIZE];

 for (int i=0; i<SIZE; i++) sum += ax[ i ];
 for (int i=0; i<SIZE; i++) myfunc(sum,ay[ i ]);
```

☑ Which method has a higher hit ratio if the cache is small (or `SIZE` is large)? Explain

On each miss a line's worth of data is brought into the cache. With Method 1, because the data is organized into structures, a miss will bring in data which is not soon needed (the `y` values for the first loop and the `x` values for the second loop). When the cache is small those not-soon-needed values will be evicted from the cache before they are needed. Suppose the line size is 256 bytes and the integer sizes are 4 bytes each. A line with Method 1 holds $256/8 = 32$ elements, but a line using Method 2 holds $256/4 = 64$ elements. Therefore the hit ratio will be higher on the code above with Method 2 when the cache is small. If the cache were larger Method 1 would have about the same performance as Method 2 because the cache would be large enough to hold the not-soon-needed values long enough for the second loop.

Problem 5: (30 pts) Answer each question below.

(*a*) Eliminating bypass paths from an n-way superscalar statically scheduled processor would have a much larger effect than eliminating them from a scalar statically scheduled processor (like our 5-stage MIPS implementation).

☑ Describe a positive benefit of eliminating the bypass paths that's much larger for the n-way superscalar.

Full-Credit Answer: The cost of bypass paths in a superscalar processor is $\propto n^2$, much larger than their cost in a scalar processor, and much larger than an acceptable n-times higher cost.

Long Explanation: In an n-way, 5-stage superscalar processor derived from our familiar 5-stage MIPS implementation there would be bypasses from each of n ALU results in the ME stage and n writeback values in the WB stage, to both inputs to each of n ALU's in the EX stage, for a total of $4n^2$ multiplexer inputs. The cost of these bypass paths is much larger in the superscalar processor, even after taking into account the expected n times higher cost.

☑ Describe a disadvantage of eliminating the bypass paths that's a much bigger disadvantage for the n-way superscalar.

No bypass is needed in the scalar processor if dependent instructions are separated by two instructions (see diagram below), but in an n-way superscalar the separation must be at least $2n$ instructions. Therefore there would be many more added stalls in the superscalar processor.

```
add r1, r2, r3  IF ID EX ME WB
or  r6, r7, r8     IF ID EX ME WB
and r9, r7, r8        IF ID EX ME WB
sub r4, r1, r5           IF ID EX ME WB
```

(*b*) Consider a deeply pipelined system with $5n$ stages without bypass paths and that runs programs in which dependent instructions are far apart. Name two factors that will limit clock frequency for larger values of n. *Note: bypass paths were not mentioned in the original exam.*

☑ One frequency limiter.

The clock period includes time for the logic within a stage and time for the latch itself. Ideally, the time for the logic is reduced by n, but the time for the latch remains constant. So the clock period can be no lower than the latch time.

☑ Another frequency limiter.

In the $5n$-stage system logic within each stage in the original 5-stage system must be split into n pieces. If that is done ideally the time for that logic would be t_l/n, where t_l is the logic time in a stage in the 5-stage system. But because logic cannot be perfectly divided some stages might take longer.

(*c*) Consider the BHT of a bimodal branch predictor. An entry would contain a CTI type, a two-bit counter, a branch target, and perhaps a tag. The tag would be used like a cache tag, but does not need to be as large. For this problem only consider branches.

☑ How large would the target need to be to predict MIPS branches? *Hint: The answer is not 32 bits.*

Since the address of the branch is known, the target field need only hold the displacement. The predictor hardware can then compute `PC + 4 + 4 * DISP`. So the target field size would be 16 bits.

☑ Explain what the predictor can do with the tag to improve prediction accuracy.

If the tag doesn't match then the target will very likely be wrong, in fact, the instruction being fetched might not be a branch (or other CTI) at all. So when the tag doesn't match predict not taken, which is correct for non-CTIs and about half the branches.

(*d*) A four-way statically scheduled processor is much less expensive than a four-way dynamically scheduled processor. Why would the dynamically scheduled processor run some code faster than the statically processor even when the code was compiled with a good compiler that targeted the specific four-way implementation?

☑ Specific advantage of dynamically scheduled processor for some code.

The compiler can schedule for fixed (always the same) latencies, such as those for floating point instructions, but the latency of load instructions depends upon where the data is found (*e.g.*, L1 cache, L2 cache, memory). In a region of code containing several loads, a compiler might be able to schedule one of those loads assuming an L2 cache hit (an L1 miss), but it would be difficult to find enough non-dependent instructions to schedule assuming all loads missed the cache. In contrast the dynamically scheduled processor could execute non-dependent instructions following any load that missed. Note: An L2 hit has on the order of a 15 cycle latency.

Another advantage is for branches. The compiler has most freedom to schedule within basic blocks, but in many types of programs they are small, about 10 instructions. Though a compiler can draw instructions from before and after a basic block this can slow code down because some of those *boosted* instructions will not need to be executed for some branch outcomes. In contrast the dynamically scheduled processor works with the actual path taken (actual branch outcomes) and so is not affected by branches.

☑ Describe properties of code that would run at the same rate on the two processors.

Code that has large basic blocks, loads that always hit the L1 cache, and that has large distances between dependent instructions. A compiler can do a good job scheduling this kind of code, and so the dynamically scheduled processor's advantages are won't apply.

(*e*) In many VLIW ISAs the bundle size is 3 slots. Describe a disadvantage of making a VLIW ISA with a larger bundle size.

☑ Disadvantage of larger bundle size.

Branch targets in VLIW ISA's must be at the start of a bundle. So if the bundle size were larger the compiler might need to insert more `nop` instructions at the end of a bundle to push a branch target to the beginning of a bundle.

(*f*) A tester measuring the performance of a system using SPECcpu has the source code to SPECcpu programs. Why is that important to the goals of the suite?

☑ Source code important to SPECcpu goals?

A goal of SPECcpu is to measure the performance potential of new implementations and new ISA's. For the performance potential of a new implementation of an existing ISA to be reached the code has to be compiled for that implementation. This can be done by the tester using the source code. (With the alternative of having SPECcpu distribute binaries [compiled code] the code might be compiled for some older implementation.) The source code is completely necessary new ISA's (say, if the tester had the very first system with this new ISA), since there would be no way for the binary to be packaged with SPECcpu.