Name Solution

Computer Architecture
EE 4720
Final Examination

13 May 2004, 17:30–19:30 CDT

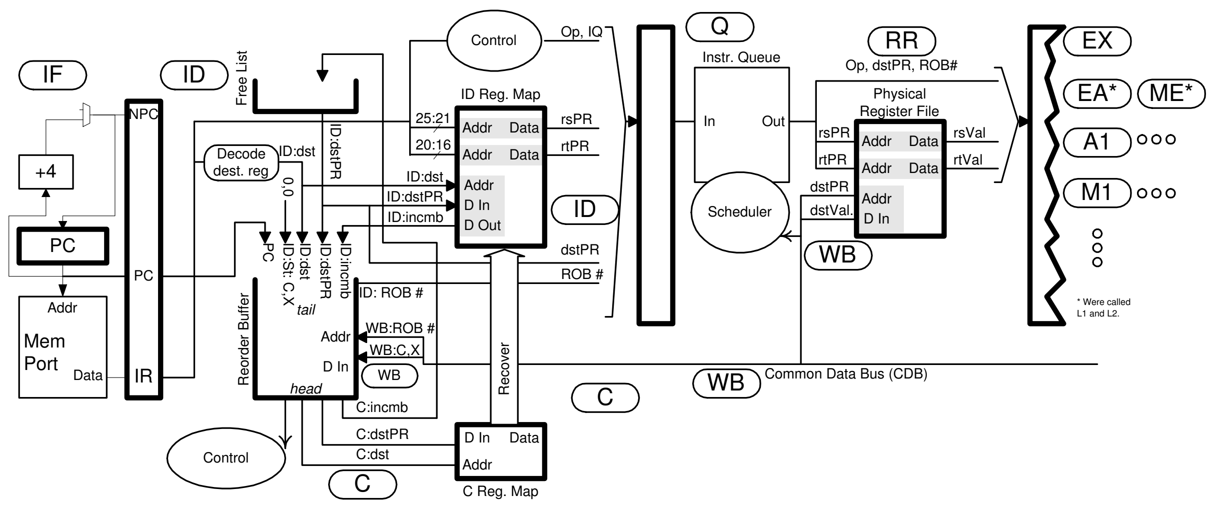
Problem 1		(19 pts)
Problem 2		(18 pts)
Problem 3		(19 pts)
Problem 4		(19 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias 17 December 1903

Good Luck!

Problem 1: The MIPS code below executes on the illustrated implementation. (19 pts)

- The implementation makes backup copies (not illustrated) of the ID map for each predicted branch instruction.
- The implementation is scalar, but there can be any number of writebacks per cycle.



The pipeline execution diagram on the next page shows the complete execution of the first instruction, a branch, and partial execution of the others. (Because of instructions before it, the branch enters `RR` after a wait of four cycles and commits after waiting another two.) The branch is mispredicted and some instructions will have to be squashed.

(*a*) Complete the pipeline execution diagram.

☑ Show where instructions are squashed.

☑ Show correct path instructions.

☑ Show each instruction until it is squashed or commits.

(*b*) Complete the tables:

☑ Physical register file. Assume the result of the load is 101, `xor` is 102, and `add` is 103. You can make up your own physical register numbers!

☑ Show where registers are removed from and put back in the free list.

☑ Complete the ID map. Don't forget to include the effect of branch misprediction recovery.

☑ Complete the commit map.

Problem 1, continued: Continued from previous page.

```
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 bneq r1, r2 SKIP      IF  ID  Q                   RR  B   WB          C

 lw r3, 0(r4)              IF  ID  Q   RR  EA              ME  WB          C

 xor r3, r8, r6                IF  ID  Q   RR  EX  WB        x

SKIP:

 add r3, r3, r7                    IF  ID  Q   RR  EX  WB    x IF  ID  Q   RR  EX  WB  C
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # ID Map

 Reg   Initial PR
 r1    19
 r2    20
 r3    27                      10  4   15                  10      4
 r4     2
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # Commit Map

 Reg   Initial PR
 r1    19
 r2    20
 r3    27                                                                  10          4
 r4     2
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 Physical Register file

 PR    Initial State
  2    [r4
  4    ]                           [r3             102     x]      [               103
 10    ]                       [r3                             101                     ]
 15    ]                               [r3             103 x]
 19    [r1
 20    [r2
 27    [r3                                                                 ]
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Branch Misprediction Recovery: The branch resolves (taken or not taken is determined) in cycle 8 but the misprediction recovery does not start until WB. To recover from the misprediction the wrong-path instructions `xor` and `add` are squashed and the ID register map is returned to the state it was when the branch was decoded (register `r4` mapped to 10). In the next cycle fetch proceeds on the correct path, fetching the `add` for a second time. The first time around, on the wrong path, the `rs` source for the add was mapped to physical register 4; the second time around it maps to physical register 10. Note that the add is fetched twice even though it is on both the taken and not-taken paths. It would be possible but probably not worth the trouble to build a processor that would not have to fetch the `add` a second time in this situation.

Before recovery starts in cycle 10 the `xor` and `add` execute normally, unaware of the squash which they are fated to suffer.

Note: WB occurs as soon as the result is computed (after EX, ME, etc), it never stalls. In hand-solved problems there can be any number of WB per cycle (to reduce tedium).

A common mistake was to show `add` being fetched once, it is fetched twice as explained above.

Problem 2: Answer each question about the illustrated MIPS implementation. (18 pts)

(*a*) Complete a pipeline execution diagram for the code below running on the illustrated implementation. Assume that any needed bypass connection is available and please check for dependencies.

```
 # Solution
 # Cycle           0  1  2  3  4  5  6  7  8  9  10
 add.d f6, f2, f4  IF ID A1 A2 A3 A4 WF
 add.d f4, f6, f2     IF ID -------> A1 A2 A3 A4 WF
```

Note that the stall is in ID, an IF stall for the second `add.d` is impossible because there is no way to detect the dependency while the second `add.d` is still being fetched.

(*b*) In the diagram below add the bypass connection(s) needed for the code above. **Do not** add bypass connections that are not needed. (Don't worry if it's a tight squeeze in the diagram, but be legible.)

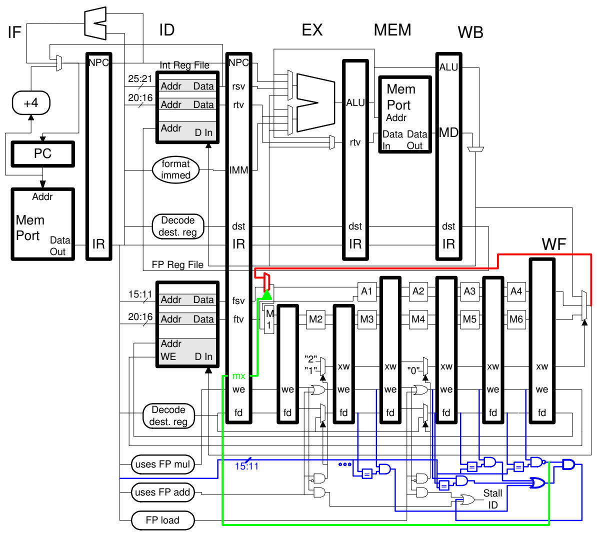
Solution shown in **red bold**. The output of the added multiplexor connects to both A1 and M1, though the problem would be correctly solved if it just connected to A1. By connecting it to both it can be used to bypass results to instructions that use the multiply unit. Similarly, the input of the added multiplexor comes from the output of the WF mux, though it could have come from the adder output in WF.

(*c*) Add control logic that generates the stall signal encountered in the code above.

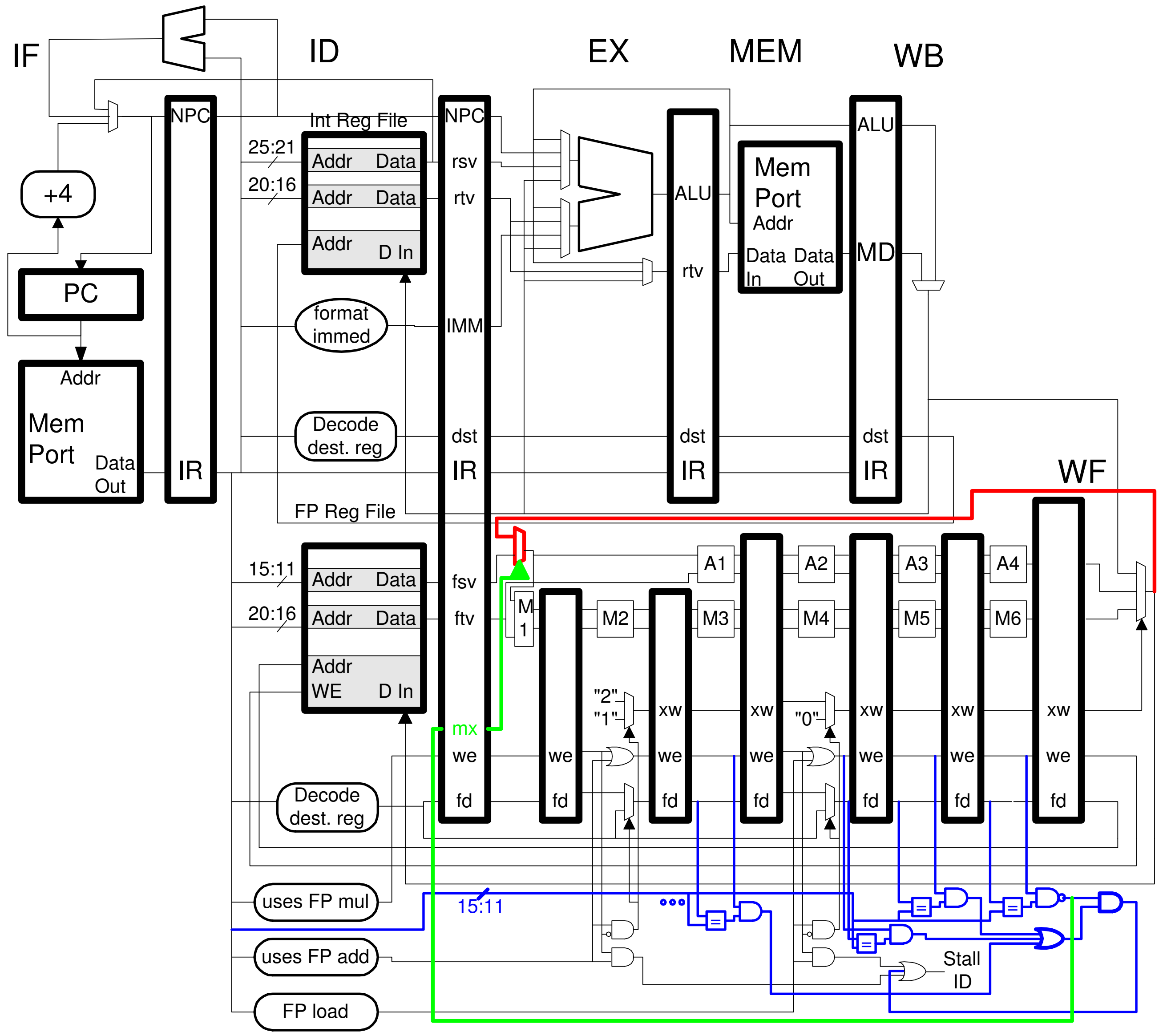
Solution shown in **blue bold**. To generate a stall the hardware needs to detect a match between registers to be written back, in the `fd` pipeline latches, and the `fs` register number. If there is a match between `fs` and the `fd` in A4, the value can be bypassed and there is no need to stall. If there is no such match but there is a match between `fs` and an instruction in A1, A2, or A3 then a stall signal is generated. A match between `fs` and `fd` is detected by comparing register numbers using a $\boxed{=}$ and checking whether the respective instruction writes anything at all, by looking at the `WE` (write enable) bit.

(*d*) Add control logic for the bypass multiplexor added in a previous part.

Solution shown in **green bold**. The logic used to generate the stall signal already checks if there is a match between `fs` and the `fd` of the instruction in the stage before writeback. That signal is used as the multiplexor control signal. Note that when the mux control signal is zero it will bypass otherwise it will use `fsv`.



*** Larger implementation diagram on next page.


IF
ID
EX
MEM
WB
WF
NPC
Int Reg File
25:21
20:16
Addr Data
Addr Data
Addr D In
rsv
rtv
+4
PC
format immed
IMM
Addr
Mem Port
Data Out
IR
Decode dest. reg
dst
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr WE D In
fsv
ftv
mx
we
fd
ALU
rtv
Mem Port Addr
Data In
Data Out
MD
A1
A2
A3
A4
M 1
M2
M3
M4
M5
M6
"2"
"1"
"0"
xw
Decode dest. reg
uses FP mul
uses FP add
FP load
15:11
Stall ID

Problem 3: The following questions refer to the assembly language code below. Branch outcomes are shown. (19 pts)

```
LOOP:
 0x1000:
B1: beq r1,r2 SKIP1     T  T  T  N  N  T  T  T  N  N  T  T  T  N  N
 nop                                                           !
 add r5, r6, r7                                                A
SKIP1:                                                         !
 # **Ten** arithmetic instructions only until next branch, no other instructions.
 #                                                             !
B2: beq r3,r4 SKIP2      T  N  T  N  T  N  T  N  T  N  T  N  T  N  T  N
 nop
 add r8, r9, r10
SKIP2:
 add r11, r12, r13
 j LOOP
 nop
```

(*a*) Suppose the code above runs on a system using a bimodal branch predictor with a 256-entry branch history table. What would be the best prediction accuracy of branch B1 and B2 after warmup. *Hint: "Best" applies to* B2 *but not* B1.

☑ Accuracy of B1. Just 40%.

☑ Accuracy of B2. At best, 50%.

☑ Why is there a "best" accuracy?

For B2, if its entry in the BHT starts out at 1 it will mispredict B2 consistently, for an accuracy of 0%.

(*b*) Suppose the code above runs on a system using a local predictor with a 256-entry BHT. What is the smallest local history size that would allow the two branches above to be predicted with 100% accuracy?

☑ Smallest size. Explain

The smallest local history size is three bits. If it were two bits then for B2 it could not distinguish the third consecutive taken from the first not taken because both would have a local history of TT. With three bits the local history would be NTT for the third taken and TTT for the first not-taken. Branch B1 would only need one bit (but in a real system that would interfere with other branches), the local history would be the larger of the two, 3. Note that there would be no interference.

Problem 3, continued:

(*c*) Suppose the code runs on a system using a gshare branch predictor with a 10-bit GHR. Show the contents of the GHR at time `A` (see the right-hand side of the diagram).

☑ Gshare GHR contents:

The global history register would hold: `NTNNTTTNTT`, with the most recent outcome the rightmost character.

(*d*) Suppose the code runs on a PPC970 (or a POWER4) processor. Show the contents of what it uses for a GHR at time `A`. State any assumptions. *Note: This part based on a homework assigned Spring 2004, and would not be asked other semesters.*

☑ PPC970 GHR contents:

The PPC970 predictor GHR inserts one bit per fetch group, which has as many as eight instructions and must be contiguous. A fetch group that does not contain a branch is treated like one having a not-taken branch.

To solve the problem one must figure out where the fetch groups are. Start at B2. That fetch group would end at B2+1 (the `nop`) and start with one of the arithmetic instructions, B2-6. The table below shows the last ten fetch groups (before A). The GHR would be `TnTtTntTnT`, as in the homework, an uppercase T or N is for a branch instruction, lower-case t is for the jump and lower-case n is for a group of instructions without a control transfer.

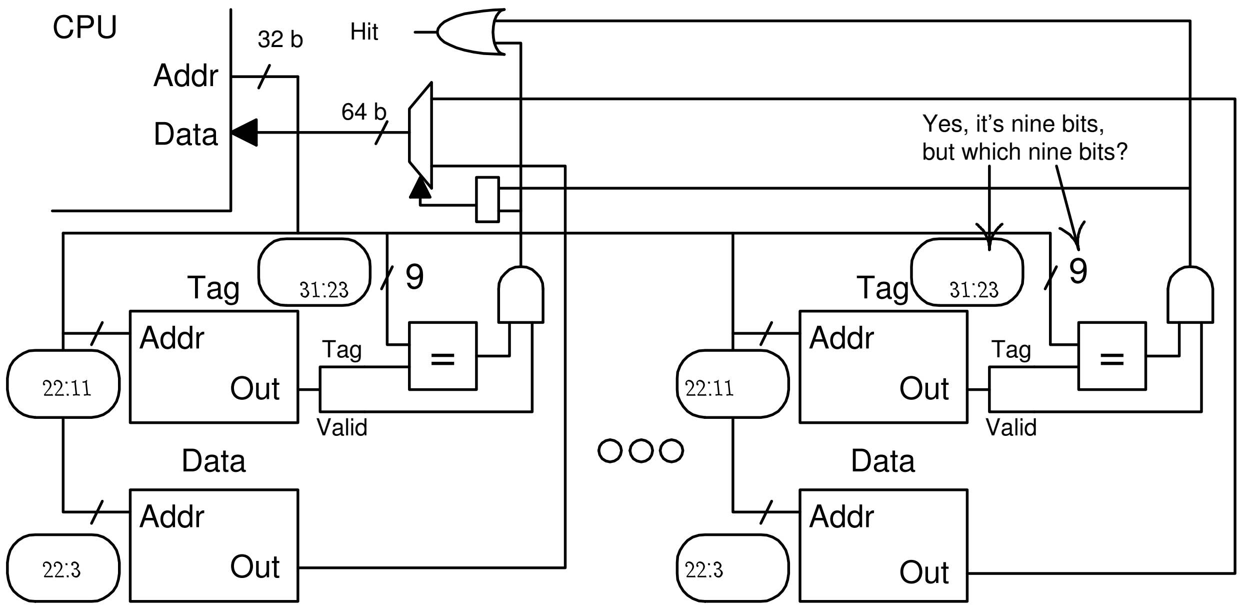
`Fetch Groups:`

Instructions	Branch/Jump/none	Outcome
B1 to B1+1	B1 Taken:	T
SKIP1 to B2-6	No cti.	n
B2-6 to B2+1	B2 Taken:	T
SKIP2 TO SKIP+2	Jump.	t
B1 to B1+1	B1 Taken:	T
SKIP1 TO SKIP2-6	NO cti.	n
SKIP2-5 to SKIP2+2	B2 not taken, jump:	t
B1 to B1+1	B1 Taken:	T
SKIP1 to B2-6	No cti.	n
B2-6 to B2+1	B2 Taken:	T

Problem 4: (19 pts) The diagram below is for a four-way set-associative cache on a system with 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.)

	Tag	Index	Offset
Address:	31 … 23	22 … 11	10 … 3, 2 … 0

☑ Cache Capacity (Indicate Unit!!):
Capacity is 4×2^{23} characters (32 MiB).

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

It's the cache capacity plus $4 \times 2^{23-11}(32 - 23 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is 2^{11} characters.

Note: Too many gave "bits" for the unit (but meant characters).

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

	Tag	Index	Offset
Address:	31 … 25	24 … 11	10 … 3, 2 … 0

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

The array element size is two characters (sizeof(short int). The line size is $2^{11} = 2048$ characters and so 1024 consecutive `i` iterations will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $1023/1024$. The total amount of memory accessed is 1024 characters, which is much smaller than the 32 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{1023}{1024}+1\right)=\frac{2047}{2048}.}$

(*c*) Choose values for the address of `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled with the minimum number of accesses. *Note: The original exam read "minimum number of misses," is easier to do.* The address of `b` must be greater than `a` and as small as possible.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i;


short int *b =


int ILIMIT =


int ISTRIDE =

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

The cache is four-way set associative with a tag field that starts at bit 22, which would be the fifth hexadecimal digit in the array `a`'s starting address. To fill the array with the fewest accesses each array access must be to a different line, that is, no line can be accessed twice. This can be achieved by setting `ISTRIDE` to half the line size. That is, $\boxed{\texttt{ISTRIDE = 1 << 10;}}$ (that is, 2^{10}). Half because the array element size is two and so, for example, `a[0]` and `a[1]` will be on different lines and the indices of those lines will differ by 1. We would like `a` and `b` to access different lines and obviously `a` would load one half of the cache and `b` would load the other half. Therefore `b` should be set to `a`'s address plus half the cache capacity and so $\boxed{\texttt{b = a + (1 << 24);}}$ which is $\boxed{\texttt{b = 0x200000;}}$ (that is, $b = a + \frac{4\times 2^{23}}{2} = a + 2\times 2^{23} = a + 2^{24}$). There are many ways to determine what `ILIMIT` should be. Using the approach taken so far, `ILIMIT` should be set so that `a[0]` to `a[ILIMIT*ISTRIDE]` is half the cache capacity. Solve `2*ILIMIT*ISTRIDE==1<<24` or in math notation $2i_{\text{ilimit}}i_{\text{stride}} = 2^{24}$, rearranging: $i_{\text{ilimit}} = \frac{2^{24}}{2i_{\text{stride}}}$, substituting: $i_{\text{ilimit}} = \frac{2^{24}}{22^{10}} = 2^{13}$. So $\boxed{\texttt{ILIMIT = 1 << 13;}}$.

Problem 5: Answer each question below.

(*a*) One way to improve the performance of an implementation is to redesign the processor so that it uses more stages. Increasing the number of stages beyond five will yield a big improvement in performance, but at some point adding stages will have little effect.(5 pts)

☑ Give a reason for this limit having to do with the pipeline latches.

Signals must remain stable a certain amount of time at the input to a pipeline latch. In a five stage implementation the total time to execute an instruction includes five of these delays. With more stages there will be more delay, and eventually this will be a majority of the time needed to execute an instruction. One place this delay is seen is in misprediction recovery.

☑ Give a reason for this limit having to do with program characteristics.

If a functional unit is split across multiple stages then an instruction closely following and dependent on one using the unit must stall on a statically scheduled implementation. Adding stages will only add stalls in such cases. This limiter of performance gain is more severe for programs with more close dependencies. (If there are few close dependencies then there would be fewer stalls.) The limit also applies to dynamically scheduled systems in which the rate at which instructions commit would not increase linearly with higher clock frequency when there are enough dependencies.

(*b*) An instruction does not raise precise exceptions (because the ISA says it does not have to). (5 pts)

☑ Name something its handler cannot do (but could do if the exception were precise).

It cannot resume execution at the instruction immediately following the faulting instruction, nor can it re-execute the instruction itself. This is because if exceptions are not precise the handler might be called when the program executes several instructions past the faulting instruction.

(*c*) The code below uses an indirect load. Re-write it using MIPS instructions. (5 pts)

```
lw r1, @(r2)   # Load using indirect addressing.

# Solution.
lw r1, 0(r2)
lw r1, 0(r1)
```

(*d*) Every integer instruction that reads a GPR uses the `rs` and `rt` fields (or just one of them). What would be the disadvantage of a modified ISA in which some instructions use other fields to specify GPR source registers? (5 pts)

Multiplexors would be needed at the register file read ports to select the correct field for the instruction being decoded.

(*e*) Consider two options for the design of a load/store unit (LSU) for a dynamically scheduled system. The aggressive option would execute the code below quickly, while the conservative option would execute it more slowly. *Hint: the conservative option is how the LSU was described in class. Note: The original problem used* `r1` *for the address, which was not intended, but the answer with* `r1` *is similar.*

(5 pts)

```
div.d f2, f4, f6
sw r1, 0(r9)
sh r2, 0(r9)
sb r3, 0(r9)
lw r4, 0(r9)
```

☑ What is it about that code that requires special treatment?

The value to load is spread across three stores, bits 31:24 from the store byte, bits 23:16 from the `sh`, and bits 15:0 from the `sw` (assuming big-endian byte ordering).

☑ Describe how the aggressive option would execute the code.

An aggressive LSU would combine the data from the three stores when bypassing to the load.

☑ Give a brief argument for the conservative option, feel free to include made-up data (for the purposes of this test only!).

Studies show that only 1% of loads [warning, made up data] get their data from multiple stores (as in the example above). The cost of implementing an LSU that can handle such rare cases is high and so it is not worth implementing. The conservative solution would be cheaper and would be almost as fast.