Name _____ *Formatted For 2-Sided Printing*

Computer Architecture
LSU EE 4720
Final Examination

1 May 2019, 12:30–14:30 CDT

Problem 1	_____	(22 pts)
Problem 2	_____	(22 pts)
Problem 3	_____	(21 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (22 pts) Notice that the execution of the code fragment below suffers two stalls when executing on our 2-way superscalar MIPS implementation. The `add` stalls due to a dependence with `or` and the `sw` stalls due to a dependence with `add`. The MIPS implementation has three unconnected logic blocks that may be useful. Each must be connected to the opcode and func of the instruction in the appropriate slot. The output of the [=or] is 1 if the instruction is an `or`. The output of [uses rs] is 1 if the instruction uses the `rs` register as a source, likewise for [uses rt].

```
# Cycle          0  1  2  3  4  5  6  7  8
 or r1, r2, r0   IF ID EX ME WB
 add r3, r1, r4  IF ID -> EX ME WB
 sw r3, 4(r6)       IF -> ID ----> EX ME WB
 addi r6, r6, 8     IF -> ID ----> EX ME WB
```

(*a*) In the execution below the `sw` no longer stalls for `r3`. Add a bypass path that can be used by `sw` to get the `r3` value **in the execution below** but not for other cases.

☐ Add bypass path for `r3` so `sw` executes as shown. ☐ Label the path "Part a", and **do not add** unneeded bypass paths.

```
# Cycle          0  1  2  3  4  5  6  7  8
 or r1, r2, r0   IF ID EX ME WB
 add r3, r1, r4  IF ID -> EX ME WB
 sw r3, 4(r6)       IF -> ID EX ME WB
 addi r6, r6, 8     IF -> ID EX ME WB
```

(*b*) The `add` stalls due to the dependence with `or` carried by `r1`. Add control logic that detects such a dependence and connect it to the Stall ID OR gate at the lower right. The output of the logic should be `1` for any true dependence between two instructions in a group.

☐ Provide a stall signal when there is a dependence between the two instructions in `ID`.

(*c*) Notice that because the second operand is `r0`, the `or` just copies the value in `r2` to `r1`. Therefore the `add` could have used `r2` instead of `r1` and avoided the stall. Design hardware to perform such *substitutions*. The hardware, including control logic, should detect when an `or` is used as a copy (as above) and if so avoid the stall and deliver the correct source operand to the slot-1 instruction.

```
# Cycle          0  1  2  3  4  5
 or r1, r2, r0   IF ID EX ME WB
 add r3, r1, r4  IF ID EX ME WB
 sw r3, 4(r6)       IF ID EX ME WB
 addi r6, r6, 8     IF ID EX ME WB
```

☐ Detect the substitution opportunity and ☐ suppress the Stall ID signal (from the previous part).

☐ Make sure the slot-1 instruction uses the correct value ☐ and that both instructions execute correctly.

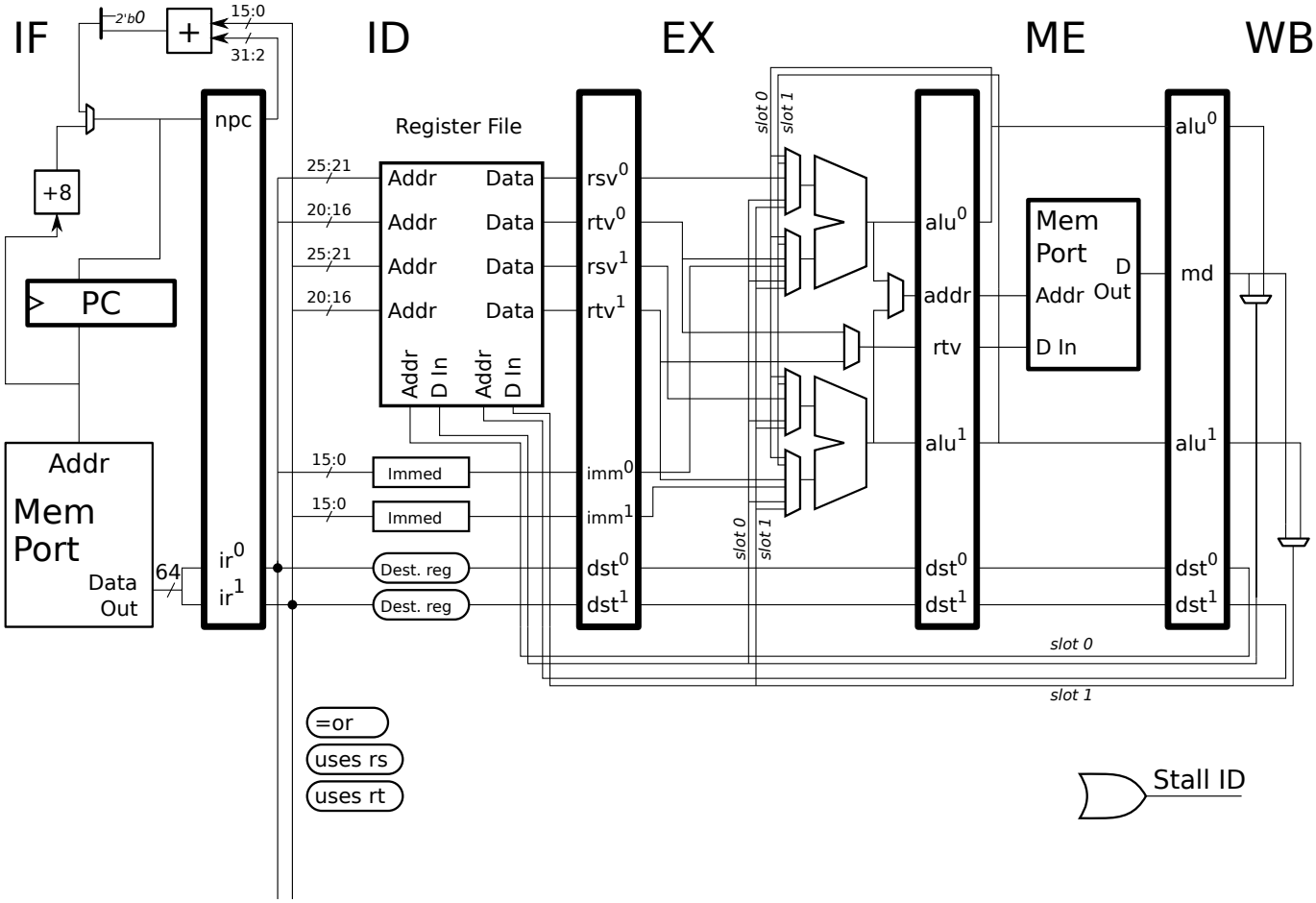
☐ Of course, pay attention to cost. Nothing added for this problem should touch 32 bits.

(*d*) *The following is a bonus question that did not appear on the original exam. Bonus for whom you ask? Definitely a bonus for those who took the class in the Spring 2019 semester and took a look at the posted*

exam. Those (you) will have an opportunity to make connections between concepts learned in the class and that will provide a deeper understanding and longer retention. Yes, the substitution hardware eliminates a stall. Suppose that `r2` had to be copied into `r1`. Provide an argument that substitution hardware is a waste of resources, illustrate with an example. Provide another argument—also with an example—that substitution hardware eliminates a stall that cannot be eliminated in another way. Whether substitution is a good idea will depend on whether the example illustrating its utility is representative of realistically compiled actual code.

☐ Argument against substitution hardware. ☐ Code example.

☐ Argument for substitution hardware. ☐ Code example.



Problem 2: (22 pts) Appearing below is our MIPS FP pipeline with the comparison units added.

(*a*) Show the execution of the following fragment on this hardware.

☐ Show execution up to second fetch of `lwc1`. ☐ Pay attention to dependencies, including the FP condition.

```
# Cycle:              0
LOOP:
 lwc1 f1, 0(r1)

 add.s f2, f1, f2

 add.s f4, f1, f3

 c.lt.s f2, f6

 bc1t LOOP # Taken

 addi r1, r1, 8

 swc1 f2, 4(r1)

 and r1, r1, r9
```

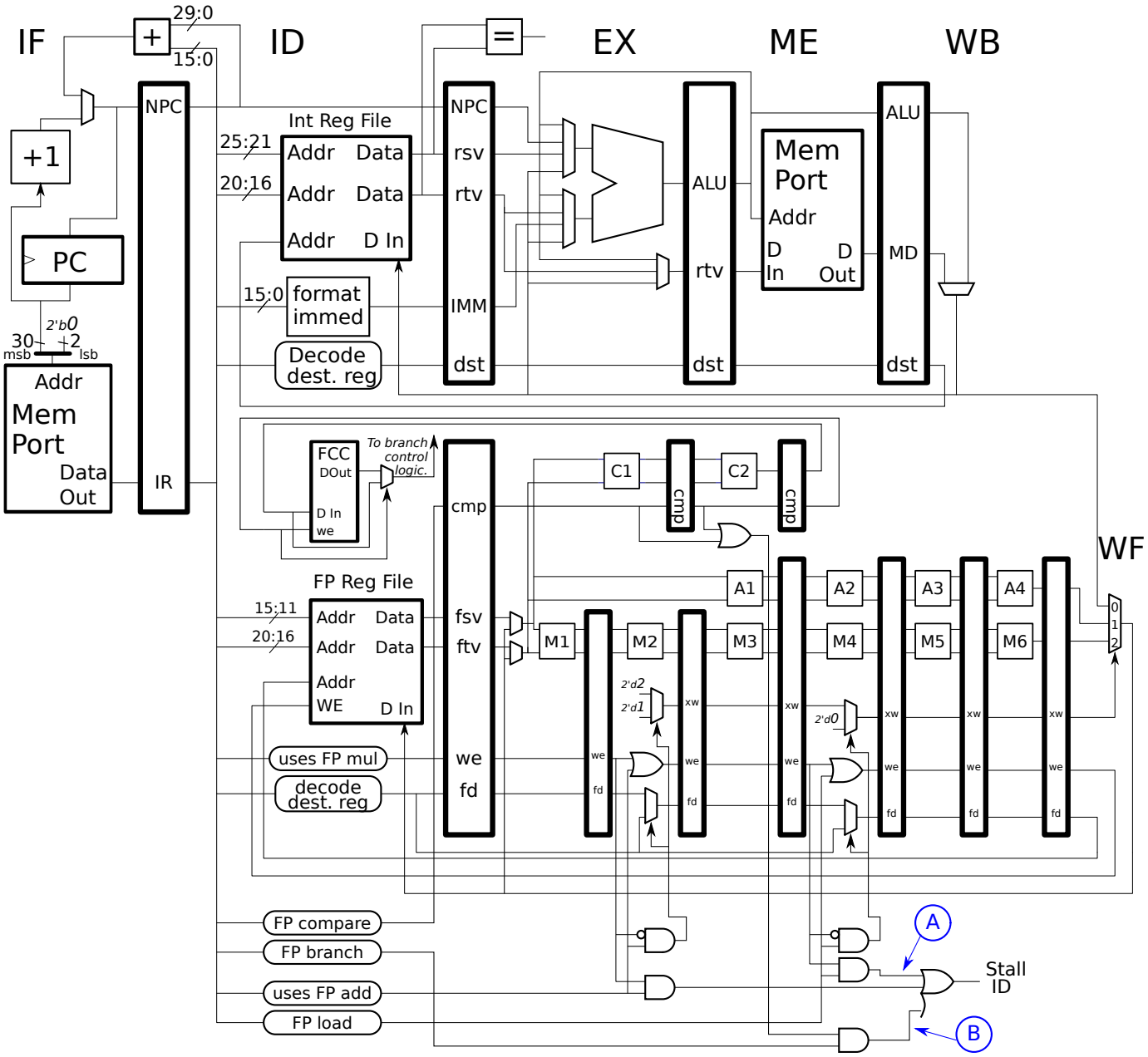
(*b*) Notice that there are two circled letters (in blue) in the lower part of the diagram. For each letter provide a code fragment that causes the labeled wire to go to logic 1.

☐ Code fragment that makes A logic 1.

☐ Show its execution and ☐ indicate cycle at which A is 1.

☐ Code fragment that makes B logic 1.

☐ Show its execution and ☐ indicate cycle at which B is 1.


IF
ID
EX
ME
WB
WF
29:0
15:0
+
=
+1
PC
NPC
IR
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
Int Reg File
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
FCC
DOut
D In
we
To branch
control
logic.
cmp
C1
C2
cmp
cmp
FP Reg File
15:11
20:16
Addr
Data
Addr
Data
Addr
WE
D In
fsv
ftv
we
fd
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
2'd2
2'd1
2'd0
xw
we
fd
uses FP mul
decode
dest. reg
FP compare
FP branch
uses FP add
FP load
A
B
Stall
ID
0
1
2

Problem 3: (21 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B1` has a repeating pattern, two repetitions are shown. Branch `B2` repeatedly and randomly emits three sequences, `a`, `b`, and `c`. Sequence `a` is `NT`, sequence `b` is `NNNTT` (five outcomes), and sequence `c` is `NNNN NTTT` (eight outcomes). After finishing one sequence, a new one is started. Sequence `a` is chosen with probability .4, sequence `b` with probability .5, and `c` with probability .1.

Here are some examples of `B2` outcomes, with spaces placed between the sequences for clarity. Example 1: `NT NNNTT NT NNNNNTTT` (that's `a`, `b`, `a`, `c`). Example 2: `NNNNNTTT NT NT NNNTT NNNTT` (that's `c`, `a`, `a`, `b`, `b`).

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   T  N  N  N  N  T  T    T  N  N  N  N  T  T

B2:    (a, p=.4): NT    (b, p=.5): NNNT T    (c, p=.1): NNNN NTTT
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

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

☐ What is the accuracy of the local predictor on branch `B2`?

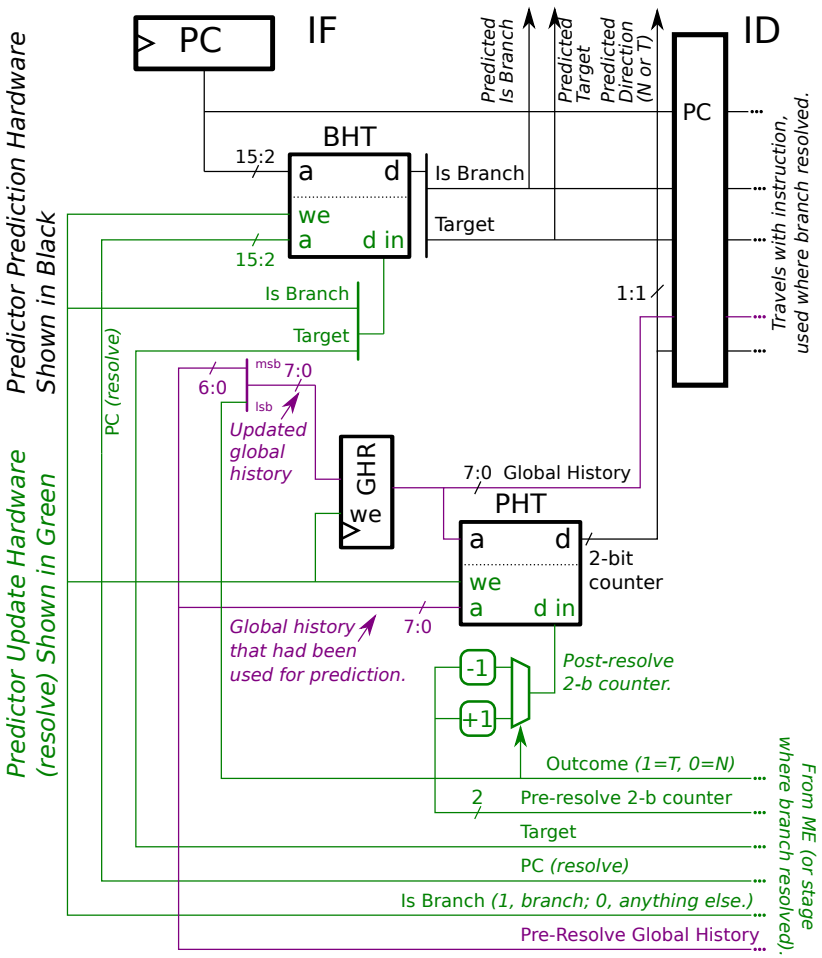
☐ What is the accuracy of the bimodal predictor on branch `B2`?

(*b*) Appearing below is a diagram of our global predictor. Notice that the GHR is not updated until the branch resolves. Modify the predictor so that the GHR is updated when the branch is being predicted (in `IF`) using the *predicted* outcome. When the branch resolves check whether the prediction was correct, and if not (if it was mispredicted) write the correct history into the GHR.

The following is interesting background material omitted from the original exam. The importance of updating the GHR using the predicted outcome increases with the number of post-branch instructions that are in the pipeline at the time a branch resolves. Consider our five-stage pipeline with branches resolving in `ME`. In that case there are just three post-branch instructions. For an 8-way superscalar pipeline there would be $3 \times 8 = 24$ instructions. One or more of those 24 instructions could itself be a branch. In the unmodified design below those branches would have been predicted with a GHR that lacked the outcome of the resolving branch and those that followed. The problem is much greater in dynamically scheduled systems where over 100 instructions can be *in flight*. For that reason global-like predictors in dynamically scheduled systems use designs like the one requested for this problem.

☐ Add hardware to detect whether the resolving branch has been mispredicted.

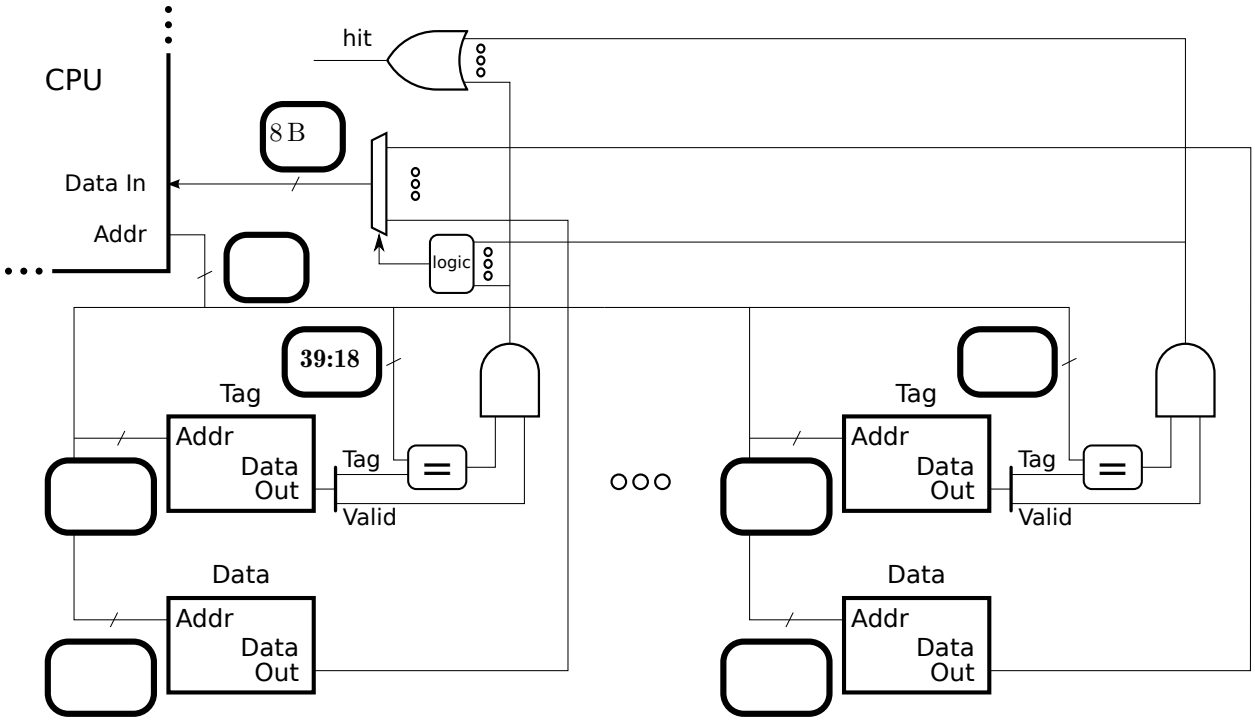
☐ During prediction write GHR based on prediction, ☐ during resolve apply corrected GHR if branch mispredicted.



Problem 4: (10 pts) The diagram below is for a 4 MiB set-associative cache with a line size of 32 B. The character size is the usual 8 bits. Other information about the cache can be deduced using hints in the diagram. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $32 = 2^5$.

(*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 below. Label the sections appropriately. (Index, Offset, Tag.)

Address: ☐ ☐ ☐

0

☐ Associativity:

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

Address: ☐ ☐ ☐

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 B (which is 2^6 B). The code fragment starts with the cache empty; consider only accesses to the array.

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

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int ILIMIT = 1 << 11;    // = 2^11

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

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 5: (25 pts) Answer each question below.

(*a*) Appearing below are simple C routines and corresponding MIPS assembler code. C variable names match the MIPS registers to which they were assigned. Register `v0` is used for the return value. The first C routine, `proc1`, operates on 32-bit signed integers. Further below are two similar C routines, `proc2` and `proc3`, each followed by the MIPS routine written for `proc1`—which is wrong because the MIPS routine is only correct for `proc1`. Rewrite those MIPS routines for `proc2` and `proc3`. Note that `int16_t` is a signed 16-bit integer and `uint8_t` is an unsigned 8-bit integer.

```
int32_t proc1(int32_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Code below is correct for proc1.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☐ Modify MIPS code for `proc2`. Pay attention to ☐ size and ☐ sign. ☐ Eliminate any unneeded instructions.

```
int16_t proc2(int16_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc2.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☐ Modify MIPS code for `proc2`. Pay attention to ☐ size and ☐ sign. ☐ Eliminate any unneeded instructions.

```
uint8_t proc3(uint8_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc3.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

(*b*) The statement below is based on a lack of understanding of how compilers work. Explain the misunderstanding and otherwise correct the statement.

> *It takes a great deal of effort to write a correct and effective compiler optimizer. Therefore optimizers are written for popular high-level languages such as C++11 but not for less popular languages such as COBOL.*

☐ The misunderstanding about compilers is:

☐ How does that change the conclusion about which languages get better optimization?

(*c*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock.

☐ Compute the peak execution rate in units of instructions per second of ☐ Chip A and ☐ Chip B.

☐ Why would Chip A run faster on simple code, such as the routines used in the homework assignments?

☐ Which chip might be less expensive? Explain.

(*d*) Answer the following questions about ISAs.

☐ Implementations of VLIW ISAs are supposed to be less costly and have higher performance than superscalar implementations of conventional ISAs. Is Intel Itanium a good example of that? Explain

☐ What important concept came out of the development IBM System/360?

☐ VAX is a good example of which ISA type?

☐ True or false: IA-32 (a.k.a. x86) was widely adopted because of its elegant design? Explain.

(*e*) The SPECcpu benchmarks can be run at two tuning levels, *base* and *peak*. Base scores are useful to those running software developed using typical practices.

☐ What kind of computer buyers should use peak scores?

☐ How do the SPEC rules for preparing base and peak runs differ?