Name ________________________________

Computer Architecture
EE 4720
Final Examination

8 May 2012, 12:30–14:30 CDT

Problem 1	______	(15 pts)
Problem 2	______	(10 pts)
Problem 3	______	(10 pts)
Problem 4	______	(20 pts)
Problem 5	______	(15 pts)
Problem 6	______	(30 pts)
Exam Total	______	(100 pts)

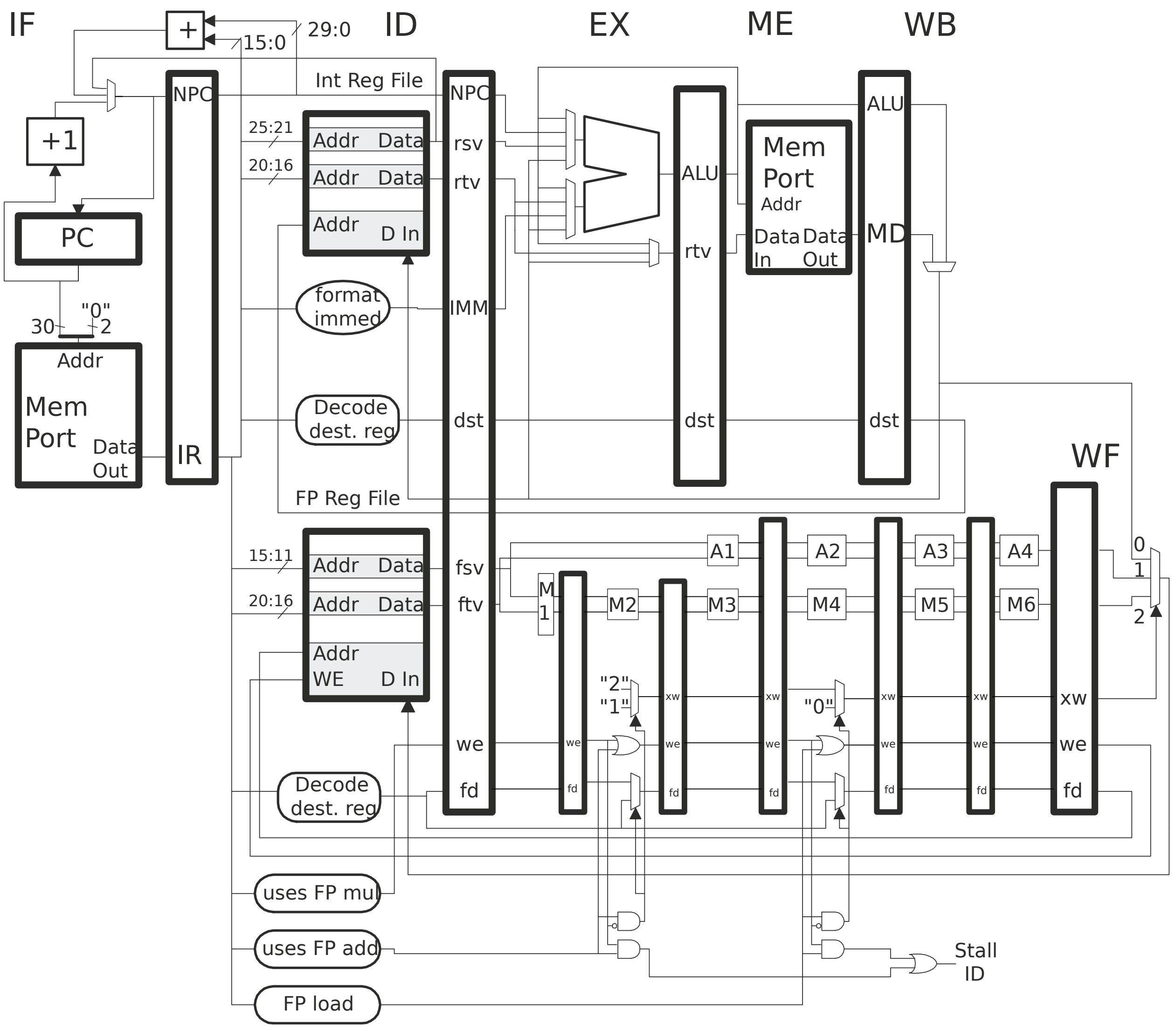
Alias ________________________________

Good Luck!

Problem 1: (15 pts) Consider the following method for implementing the MIPS32 integer multiply instruction `mul` (the one that writes ordinary registers, not to be confused with `mult`) on the implementation below. The full set of stages `M1` to `M6` perform floating point multiply, but stages `M2` to `M4` perform integer multiplication. The integer multiply `mul` will read and write registers from the integer register file but will use `M2` to `M4` to perform the multiplication. A sample execution appears below. *Grading Note: In the original exam there was an ID-stage stall in cycle 6, implying that there was no* `WB` *to* `EX` *bypass for the* `mul`*.*

```
# Cycle         0  1  2  3  4  5  6  7  8
add r1, r2, r3  IF ID EX ME WB
mul r4, r1, r5     IF ID M2 M3 M4 WB
xor r9, r10, r11      IF ID -> EX ME WB
sub r6, r4, r7           IF -> ID EX ME WB
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

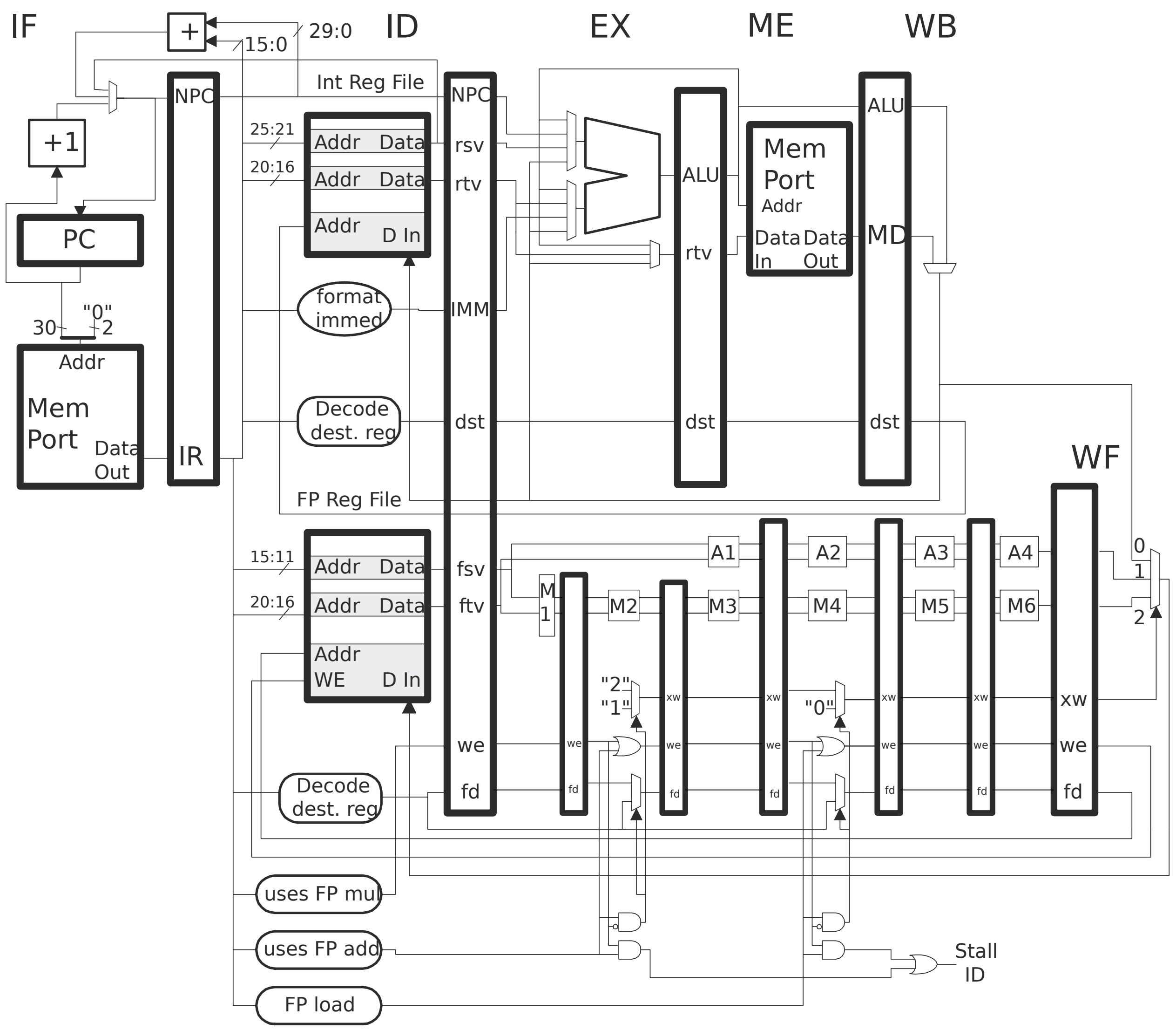
CONTINUED ON NEXT PAGE

(*a*) Modify the implementation below so that `mul` uses `M2` to `M4`. Provide any necessary bypass paths so that the code above executes as shown. For this part do not consider control logic. *Hint: Pay attention to source and destination registers.*

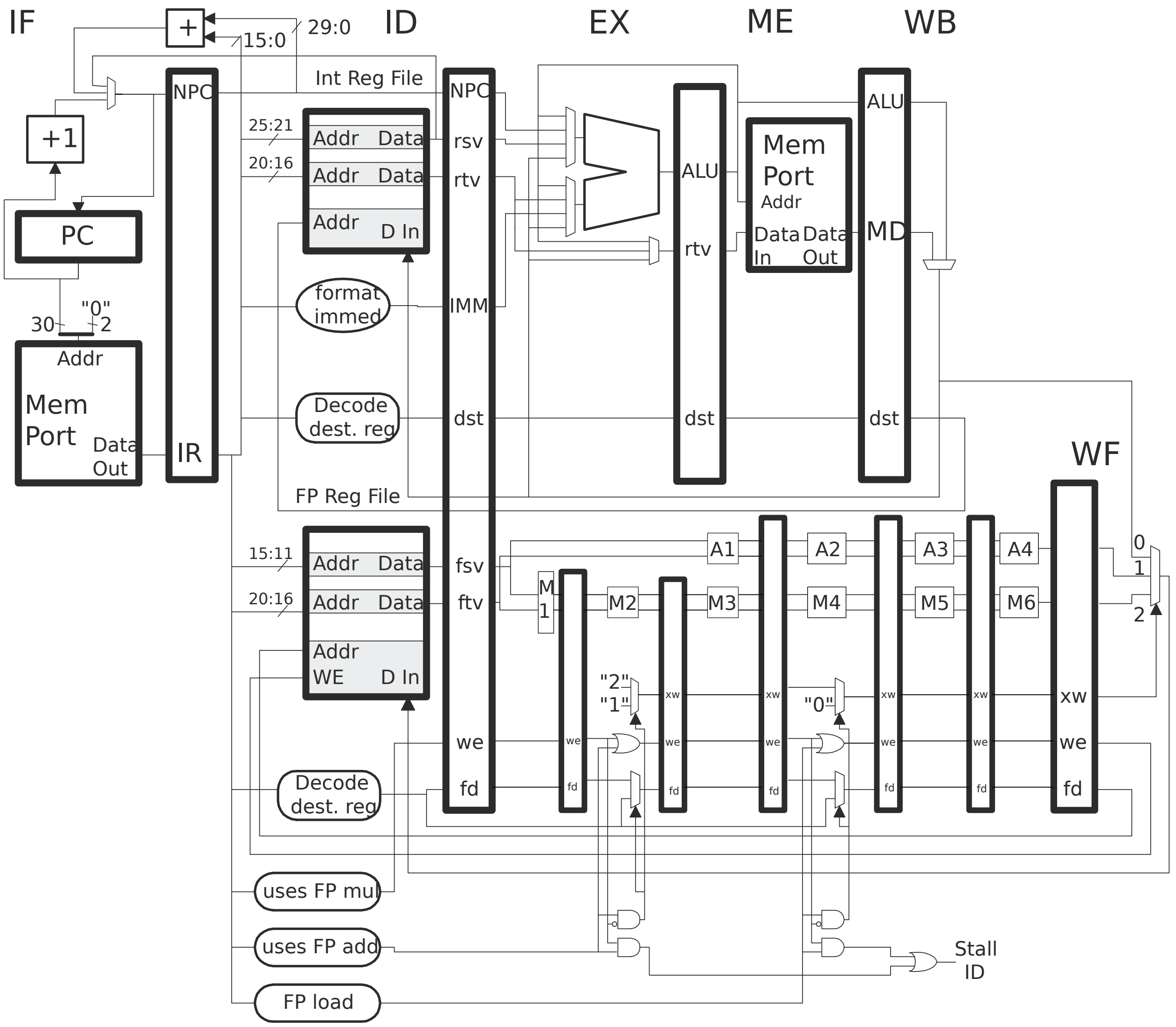
☐ Non-control logic modifications for integer `mul`.

(*b*) Add control logic related to this implementation of `mul` to detect the structural hazard on `M2` and on `WB`. Also add control logic needed for a data dependence between `mul` and an immediately following instruction. All those signals should connect to the existing *Stall ID* signal and use a new $\boxed{\text{uses int mul}}$ logic block.

☐ Control logic for `WB` structural hazard, ☐ `M2` structural hazard, ☐ and the data dependence.



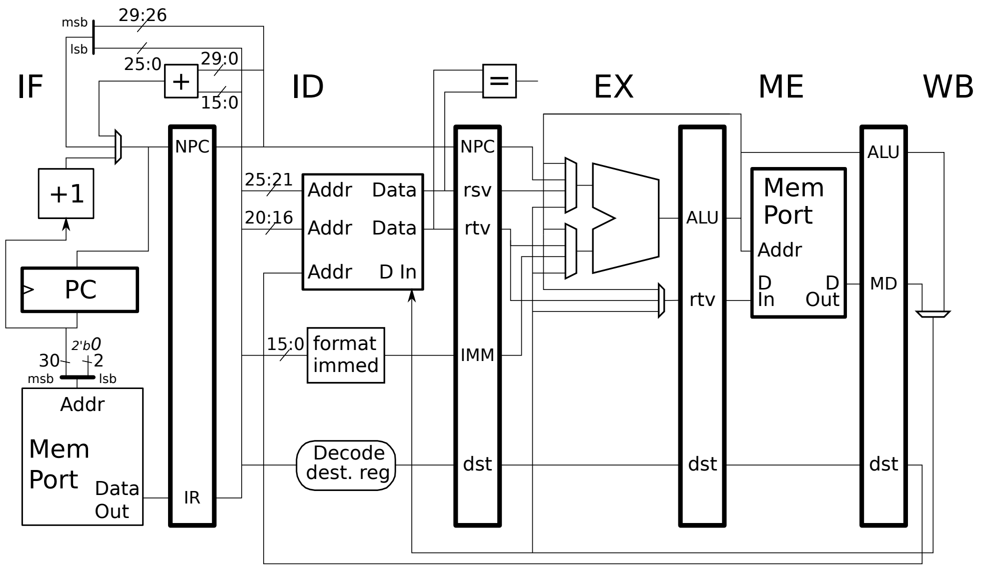
Problem 2: (10 pts) Show the execution of the instructions below on the illustrated implementation. Add any needed datapath and reasonable bypass paths.



☐ Show execution. ☐ Add datapath and reasonable bypass paths. ☐ Double-check for dependencies.

```
add.s f2, f4, f6

sub.s f8, f10, f12

add r1, r2, r3

mul.s f14, f2, f8

swc1 f14, 0(r1)
```

Problem 3: (10 pts) The code fragments below execute on several different MIPS implementations. In all cases the loop iterates many times. A five-stage scalar system is shown for reference.



(*a*) Show the execution of the code below on our familiar scalar pipeline, above. Show enough iterations to compute the CPI, and compute the CPI.

☐ Execution for enough iterations to determine CPI.

☐ Find the CPI.

☐ Doublecheck for dependencies.

☐ Note that the first instruction in not part of the loop body.

```
 lw r2, 0(r10)
LOOP:

 lw r1, 0(r2)

 addi r2, r2, 4

 bne r2, r4 LOOP

 add r5, r5, r1
```

Problem 3, continued:

(*b*) Show the execution of the code below on a 4-way superscalar statically scheduled system without branch prediction. The superscalar system has five stages, aligned fetch, and can bypass between the same stages as can our scalar system. This means there are no bypass paths to the branch condition. Compute the CPI.

☐ Execution for enough iterations to determine CPI.

☐ Find the CPI.

☐ The code below is *different* than the previous part.

☐ Doublecheck for dependencies.

☐ Note that first instruction not part of loop body.

```
 addi r1, r0, 0

LOOP: # Address of insn below is 0x1000

 add r5, r5, r1

 lw r1, 0(r2)

 bne r2, r4 LOOP

 addi r2, r2, 4
```

(*c*) Show the execution of the code below on a four-way superscalar system with perfect branch prediction.

- ☐ Execution for enough iterations to determine CPI.
- ☐ Find the CPI.
- ☐ The code below is *different* than the first part.
- ☐ Doublecheck for dependencies.
- ☐ Note that first instruction not part of loop body.

```
 addi r1, r0, 0

LOOP: # Address of insn below is 0x1000

 add r5, r5, r1

 lw r1, 0(r2)

 bne r2, r4 LOOP

 addi r2, r2, 4
```

Problem 4: (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^{10}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 12-outcome local history, and one system uses a global predictor with a 12-outcome global history.

```
Insn        Branch
Addr        Outcomes
0x1000: B1  r   r   r   r   r   r   r   r    r   r   r   r   r   r   r   r
0x1010: B2   T   N   N   N   N   T   N   N    T   N   N   N   N   T   N   N
0x1020: B3    R   R   R   R   R   R   R   R    R   R   R   R   R   R   R   R
0x2000: B4     T   T   T   T   T   T   T   T    T   T   T   T   T   T   T   T
```

Branch `B1` is random, and can be described by a Bernoulli random variable with $p = .5$ (models a fair coin toss). The outcome of branch `B3` is the same as the most recent execution of `B1` (perhaps they are testing the same condition). For the questions below accuracy is after warmup.

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

☐ What is the accuracy of the bimodal predictor on `B3`?

☐ Considering BHT size, what is the approximate accuracy of the bimodal predictor on `B4`? ☐ Explain.

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

☐ What is the minimum local history size needed to predict `B2` with 100% accuracy?

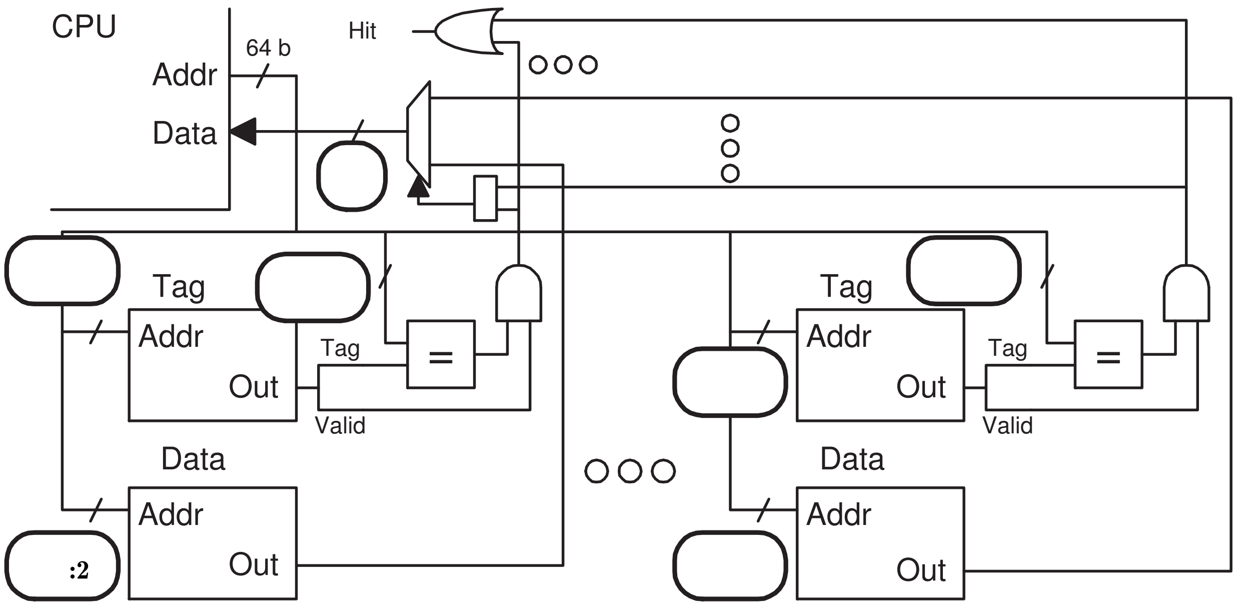
☐ What is the accuracy of the global predictor B3? ☐ Explain

☐ How many PHT entries are used by the global predictor to predict B2?

Problem 5: (15 pts) The diagram below is for an eight-way set-associative cache. The size of a tag is 42 bits and the size of a line is $128 = 2^7$ 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.)

Address:

☐ Cache Capacity (Indicate Unit!!):

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

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

Address:

Problem 5, 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 128-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

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

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

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

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

(*c*) Find the minimum positive value of `OFFSET` needed so that the code below experiences a hit ratio of 0% on accesses to `a`. Explain.

☐ Minimum positive `OFFSET` to achieve 0% hit ratio. ☐ Explanation.

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


 int OFFSET =                                      <--  FILL IN


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

Problem 6: (30 pts) Answer each question below.

(*a*) For the SPECcpu benchmark suite results can be reported using two tuning levels, *base* and *peak*. Consider a new level, *no-opt* in which the code is compiled without optimization. How valuable would no-opt tuning scores be to the people that care about SPECcpu scores? How different is no-opt tuning from base tuning?

☐ Value of no-opt tuning level?

☐ Difference between no-opt tuning and base tuning?

(*b*) The `lw` below will experience a TLB miss exception when it first executes, remember that this exception is considered routine and is not due to any kind of error. The word in memory at location `0x12345678` is `0x222`. Show the execution of this code on our five-stage static pipeline until the `add` instruction reaches writeback, and show the value in register `r1` when the handler starts (see code).

☐ Show execution from `lui` to when `add` reaches `WB`.

☐ `FILL IN` the value that will be in `r1` when the handler starts.

```
 lui r1, 0x1234

 lw r1, 0x5678(r1)

 add r2, r2, r1


HANDLER:
 # Value of r1 is                               <- FILL IN
 sw ...
 ...
```

(*c*) The instruction below is not a good candidate for a RISC ISA. Explain why in terms of hardware, not just in terms of a rule the instruction would violate. `add (r1), r2, (r3)`

☐ Instruction above unsuitable for RISC because the implementation ...

☐ Provide a quick sketch to illustrate your answer.

(*d*) With the aid of a diagram, explain why a $5n$-stage pipeline would need fewer bypass paths than a 5-stage, n-way superscalar implementation. (Both implementations are statically scheduled.)

☐ Diagram showing why there are fewer bypass paths in $5n$-stage pipeline than 5-way superscalar.

(*e*) Why is it more important to have a good compiler for a superscalar statically scheduled system than a scalar statically scheduled system?

☐ Compiler more important for superscalar because ...

(*f*) Consider the executions of the MIPS code below on a 4-way superscalar dynamically scheduled system of the type discussed in class (method 3). The first execution is correct, the others, though they would run the program correctly, have problems. Describe the problems by completing the statements below.

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q        RR EX WB C
lh r1, 0(r6)     IF ID Q  RR EA ME WB    C
sub r7, r1, r3   IF ID Q           RR EX WB C
```

☐ The one-commit-per-cycle execution below is silly because . . .

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q        RR EX WB C
lh r1, 0(r6)     IF ID Q  RR EA ME WB       C
sub r7, r1, r3   IF ID Q           RR EX WB    C
```

☐ The stalls shown below would be necessary on a statically scheduled pipeline because . . .

☐ . . . but should not occur on a dynamically scheduled one because . . .

```
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
add r3, r1, r4   IF ID Q  ----> RR EX WB C
lh r1, 0(r6)     IF ID Q  ----> RR EA ME WB C
sub r7, r1, r3   IF ID Q  -----------> RR EX WB  C
```