F2) “Don’t let me fault” (22 pts, 30 min)

The specs for a MIPS machine’s memory system that has one level of cache and virtual memory are:
- 1MiB of Physical Address Space
- 4GiB of Virtual Address Space
- 4KiB page size
- 16KiB 8-way set-associative write-through cache, LRU replacement
- 1KiB Cache Block Size
- 2-entry TLB, LRU replacement

The following code is run on the system, which has no other users and process switching turned off.

```c
#define NUM_INTS 8192 // This many ints...
int *A = (int *)malloc(NUM_INTS * sizeof(int)); // malloc returns address 0x100000
int i, total = 0;
for(i = 0; i < NUM_INTS; i += 128) A[i] = i;
for(i = 0; i < NUM_INTS; i += 128) total += A[i]; // SPECIAL
```

a) What is the T:1:0 bit breakup for the cache (assuming byte addressing)? __9__ : __1__ : __10__

b) What is the VPN : PO bit breakup for VM (assuming byte addressing)? __20__ : __12__

For the following questions, only consider the line marked “SPECIAL”. Your answer can be a fraction.

c) Calculate the hit percentage for the cache

\[
\frac{1}{2} = 50%
\]

d) Calculate the hit percentage for the TLB

\[
\frac{7}{8} = 87.5%
\]

e) Calculate the page hit percentage for the page table

100%

Show all your work below...
To improve the run-time, we will use SIMD to differentiate four terms at a time:

```c
#include <immintrin.h>

void differentiate_SIMD(int32_t* A_prime, int32_t* A, size_t n) {
    int i, j = J_INIT;
    int32_t tmp[] = STEROIDS_INIT;
    __m128i j_on_steroids = _mm_loadu_si128(tmp);

    /* fringe */
    differentiate(A_prime, A, START + 1);

    /* translate j, j_on_steroids by START */
    j += START;
    j_on_steroids = _mm_add_epi32(j_on_steroids, _mm_set1_epi32(START));

    /* main loop */
    for (i = START; i < END; i += 4) {
        __m128i A_chunk = _mm_loadu_si128(&A[j]);
        __m128i j_on_steroids = _mm_add_epi32(j_on_steroids, _mm_set1_epi32(4));
        j += 4;
        _mm_storeu_si128(&A_prime[i], _mm_mul_epi32(j_on_steroids, A_chunk));
    }
}
```

The above has four undefined macros J_INIT, STEROIDS_INIT, START, and END. Select the one macro definition that leads to the best-performing correct behavior.

### iii. (3 points) Select the correct definition for the macro START:
- (a) `#define START 4`
- (b) `#define START ((n - 1)/4*4)`
- (c) `#define START 0`
- (d) `#define START (n % 4)`
- **(c) #define START ((n - 1) % 4)**

### iv. (3 points) Select the correct definition for the macro J_INIT:
- (a) `#define J_INIT 0`
- (b) `#define J_INIT 1`
- (c) `#define J_INIT 2`
- (d) `#define J_INIT 4`
- (e) `#define J_INIT 5`

### v. (3 points) Select the correct definition for the macro STEROIDS_INIT:
- (a) `#define STEROIDS_INIT {0, 0, 0, 0}`
- (b) `#define STEROIDS_INIT {1, 1, 1, 1}`
- (c) `#define STEROIDS_INIT {0, 1, 2, 3}`
- (d) `#define STEROIDS_INIT {1, 2, 3, 4}`
- (e) `#define STEROIDS_INIT {4, 4, 4, 4}`

### vi. (3 points) Select the correct definition for the macro END:
- (a) `#define END (n - 1)`
- (b) `#define END (n)`
- (c) `#define END ((n - 1)/4*4)`
- (d) `#define END (n/4*4)`
- (e) `#define END (n/4)`
Question 2: OpenMP (5 points total)
Circle the one choice that results in the fastest parallel code with the same output as the initial serial code.

Note: parallel/critical block around “...” => #pragma omp parallel/critical { ... }

i. (3 points)
Serial:
int i;
for (i = 0; i < len_list; i += 1)
    total += list[i];
for (i = 0; i < len_result; i += 1)
    result[i] = total*i;

Parallel:
1  for (int i = omp_get_thread_num(); i < len_list; i += omp_get_num_threads())
2     total += list[i];
3  for (int i = omp_get_thread_num(); i < len_result; i += omp_get_num_threads())
4      result[i] = total*i;

(a) Add a parallel block around lines 1-2 and a parallel block around lines 3-4
(b) Add a parallel block around lines 1-2, a parallel block around lines 3-4, and a critical block around line 2
(c) Add a parallel block around lines 1-4 and a critical block around line 2
(d) Add a parallel block around lines 1-4
(e) None of the above causes the code to return the correct result

ii. (2 points)
Serial:
for(int i = 0; i < m; i += 1) {
    total += i;
    res[i] = total;
}

Parallel:
0  #pragma omp parallel
1   
2     for(int i = omp_get_thread_num(); i < m; i += omp_get_num_threads())
3     {
4     total += i;
5     res[i] = total;
6    }
7   
(a) Add a critical block around lines 2-6
(b) Add a critical block around lines 4-5
(c) Add a critical block around line 4
(d) The parallel code returns the correct result the fastest without adding anything
(e) None of the above causes the code to return the correct result
a) You are an intern at a massive hardware firm. Your first task is to design an “odd counter” circuit that receives a single bit input every cycle and outputs a single bit every cycle. It outputs a 1 if and only if it has seen an odd number of ones AND an odd number of zeros. It starts in a state where it has seen an even number of ones and an even number of zeros (remember, zero is an even number). As an example,

the input: $I: 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0$

will produce the output: $O: 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0$

Complete the FSM diagram below. The names of the states are arbitrary, but use S00 is the start state. Fill in the truth table on the right. The previous state is encoded in $(P_1, P_0)$, the next state is encoded in $(N_1, N_0)$, and the output is encoded as $O$. Make sure to indicate the value of the output on your state.

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_0$</th>
<th>$I$</th>
<th>$O$</th>
<th>$N_1$</th>
<th>$N_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

It’s a NAND gate: (not (and A B))

b) Rebuild this circuit with the fewest gates in the box to the right, using ONLY AND, OR and NOT gates.

c) Finally, your boss wants you to choose an XOR gate for the circuit to the right: The clock speed is 2Ghz, the setup, hold, and clock-to-q times of the register are 40, 70, and 60 picoseconds ($10^{-12}$ s) respectively. What range of XOR gate delays is acceptable? E.g., “at least W ps”, “at most X ps”, or “Y to Z ps”.

10 to 400 picoseconds

10 to 400 picoseconds

4 kibicircuits

d) You’re asked to create all the unique 3-to-2 circuits (i.e., 3 inputs: $I_2, I_1, I_0$ and 2 outputs), with one minor catch. Your circuit must ignore the value of $I_1$ if the value of $I_2$ is 1. How many different circuits will you have to make? Use IEC terminology, like 128 mebicircuits, 512 tebicircuits, etc.
F4) Don’t just sit and wait for another datapath you by!

On the right is the *single-cycle* MIPS datapath presented during lecture. *Ignore pipelining for the question.* Your job is to modify the diagram to accommodate a new MIPS instruction.

Your modification may use simple adders, shifters, mux chips, wires, and new control signals. If necessary, you may replace original labels.

We want to add a new MIPS instruction (we’ll call it `addpr` for “add to pointed reg”) that is almost identical to `addi` but with a twist. Instead of storing into the `rt` register the sum of the constant and the value of the register specified by `rs`, it stores into the `rt` register the sum of the constant and the value of the register specified by *the lowest 5 bits in memory at the address specified by the pointer stored in the `rt` register*. Said another way, first get the pointer stored in the `rt` register. Follow that pointer to get its value from memory. Take the lowest 5 bits of that value, treat it as a register number, and find out what is stored in that register. Add that to the immediate, and store it in the `rt` register.

a) Make up the syntax for the I-type MAL MIPS instruction that does it (show an example if the pointer lives in `$v0`, and the constant is 5). On the right, show the register transfer language (RTL) description of `addpr`.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addpr $v0, 5</code></td>
<td><code>R[rt] = R[MEM[R[rt]](4:0)] + SignExtImm;</code></td>
</tr>
<tr>
<td></td>
<td><code>PC = PC + 4;</code></td>
</tr>
</tbody>
</table>

b) Change as *little as possible* in the datapath above and list all changes below. You may not need all boxes.

(i) Add a mux whose output is tied to “Data Memory Adr” and whose input is either the ALU or busB `R[rt]`, driven by a control line called “MemAdr” whose value is either ALU or busB.

(ii) Add a mux whose output is tied to “Ra” and whose input is either Rs or the lowest 5 bits of “Data Memory Data Out”, driven by a control line called “RaSrc” whose value is either Rs or Mem.

(iii) 

c) We now want to set all the control lines appropriately. List what each signal should be, an intuitive name or {0, 1, x – don’t care}. Include any new control signals you added.

<table>
<thead>
<tr>
<th>RegDst</th>
<th>RegWr</th>
<th>nPC_sel</th>
<th>ExtOp</th>
<th>ALUSrc</th>
<th>ALUctr</th>
<th>MemWr</th>
<th>MemtoReg</th>
<th>MemAdr</th>
<th>RaSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Rt(0)</code></td>
<td>1</td>
<td>+4</td>
<td><code>Sign</code></td>
<td><code>Imm(1)</code></td>
<td><code>Add</code></td>
<td>0</td>
<td><code>ALU(0)</code></td>
<td><code>busB</code></td>
<td><code>Mem</code></td>
</tr>
</tbody>
</table>

d) In the context of a single-cycle CPU, `lw` used to be the “critical path” instruction that really determined our fastest clock period, since it utilized the most components of our datapath. Using the terms below to create an expression that will determine how much slower our clock period will be if we also consider the `addpr` instruction: `PCRegClkToQ`, `InstMemAccess`, `ControlLogicDelay`, `RegFileAccess`, `ALUdelay`, `DataMemAccess`, `RegSetup`, `RegHold`, `MemSetup`, `MemHold`.

`RegFileAccess` (it has to do that twice. The rest of the terms are the same.)
**Question 8: Pipelined CPU (10 points total)**

Consider the 5-stage single-issued pipelined MIPS datapath consisting of Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Write-Back (WB).

You are given the following MIPS instruction sequence:

```
# $s0 to $s3 = 56, 30, 30, 7
# $t0 to $t4 = 7, 7, 7, 7, 7
    add $t0, $s0, $0
    and $t1, $t0, $s1
    or $t2, $t0, $s2
    sub $t3, $t0, $s3
    srl $t4, $t0, 2
```

Start numbering the cycles with 1 when the add instruction enters the IF stage.

For part i. to iii. assume that the datapath is broken and there is no forwarding and no stalling.

i. (2 points) What are the values of $t0$ to $t4$ at the end of Cycle 7?
   (a) 56, 24, 62, 7, 7
   (b) 56, 24, 62, 49, 7
   (c) 56, 6, 31, 7, 7
   (d) 56, 6, 7, 7, 7
   (e) None of the above

ii. (2 points) What are the values of $t0$ to $t4$ at the end of cycle 8?
    (a) 56, 24, 62, 49, 7
    (b) 56, 24, 62, 49, 14
    (c) 56, 6, 31, 49, 7
    (d) 56, 6, 31, 7, 7
    (e) None of the above

iii. (2 points) What are the values of $t0$ to $t4$ at the end of cycle 9?
     (a) 56, 24, 62, 49, 7
     (b) 56, 24, 62, 49, 14
     (c) 56, 6, 31, 49, 14
     (d) 56, 6, 31, 0, 7
     (e) None of the above

iv. (2 points) What instruction(s) is/are computing the wrong result(s) (choose the answer that includes ALL faulty instructions)?
    (a) add
    (b) and, or
    (c) and, or, sub
    (d) and, or, sub, srl
    (e) add, and, or, sub, srl

v. (2 points) Say we want to completely fix the problem from part iv. using forwarding. Which forwarding path(s) do we need to provide in order to execute the code sequence correctly (it is implied that multiplexers are inserted to join the forwarded signals with the original signals)?
    (a) Output of ALU in the EX stage back to the input of the ALU in the EX stage.
    (b) Output of ALU in the MEM stage back to the output of Register File in the ID stage.
    (c) Output of ALU in the MEM stage back to the input of ALU in the EX stage.
    (d) Both (a) and (b).
    (e) Both (a) and (c).
F4) **What do you call two L’s that go together?** (Continued)

The moving average (a type of low pass filter) is an operation commonly used to smooth noisy data. Here we compute a centered moving average of width \( WIDTH \) on an array of data of size \( \), where each element in our output array is the average of the current element, the previous \( (WIDTH-1)/2 \) elements, and the next \( (WIDTH-1)/2 \) elements. Assume that \( WIDTH \) is odd for simplicity and use zeros where “required” elements do not exist.

Example Input of : \[7, 2, 3, 4, 8, 6\]
Output for \( WIDTH=3 \): \[3, 4, 3, 5, 6, 4.6666667\]

Fit this problem to the MapReduce paradigm using a single map and reduce by filling in the blanks below. You may assume that you have access to the global variables \( WIDTH \) and \( SIZE \). We expect you to use C syntax with the addition of a few java-like pseudocode elements (e.g. arrays have \.length\).

```c
// receives data one element at a time
// Inputs: (key) is index \( i \), (value) is \( A[i] \)

map(int key, float value){
    for (int x = key-floor(WIDTH/2); x < key+floor(WIDTH/2); x++)
    {
        context.write( x, value);
    }
}

// outputs elements of centered moving average
// Outputs MUST be of the form:
// (key) is index \( i \)
// (value) is moving average of width \( WIDTH \) centered at \( i \)

reduce(int key, float[] values){
    float total = 0;
    // do not emit keys that do not exist in output array
    if ( (key >= 0) && (key < SIZE) ) {
        for (int x = 0; x < values.length; x++)
        {
            total += values[x];
        }
        context.write( key, total/values.length );
    }
}
```