Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for partial credit. You can use the extra white space and the backs of the pages for scratch work.

Problem 1. Caches (16 points)

Assume you have a partially empty two-way set associative cache with 8 lines per way and a block size of 4 (4 data words per line) as shown below. Also, assume that data and addresses are 32 bits. **Note that rather than showing the data stored in each data cell, the figure below shows the address of the memory location whose data will go into that cache entry.** For each cache line assume that way 0 (the left way) is the LRU.

**(A) (8 points)** Given the memory access sequence below, fill out the two-way set associative cache as much as possible. For each data entry, enter the address of the memory location whose data would go there if it is known. Also, compute the tag for each filled line.

**Memory Access Sequence: 0x7210, 0x4C34, 0x3A18, 0x721C, 0xAC4, 0x3A1C**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data 0</th>
<th>Data 1</th>
<th>Data 2</th>
<th>Data 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 1</td>
<td>0xE4</td>
<td>0x7210</td>
<td>0x7214</td>
<td>0x7218</td>
</tr>
<tr>
<td>Line 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 3</td>
<td>0xH2</td>
<td>0x98</td>
<td>0x5930</td>
<td>0x5934</td>
</tr>
<tr>
<td>Line 4</td>
<td>0x124</td>
<td>0x9240</td>
<td>0x9244</td>
<td>0x9248</td>
</tr>
<tr>
<td>Line 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 7</td>
<td>0xA2</td>
<td>0x5170</td>
<td>0x5174</td>
<td>0x5178</td>
</tr>
</tbody>
</table>
For 0x4C30-0x4C3C accepted either way 0 or 1 due to confusion with LRU 0 statement.

(B) (6 points) Of the memory accesses made in part A, which ones were hits and which ones were misses? Fill out the table below. Also provide the cache index for each memory access.

<table>
<thead>
<tr>
<th>Memory Access</th>
<th>Hit/Miss?</th>
<th>Index of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7210</td>
<td>Miss</td>
<td>1</td>
</tr>
<tr>
<td>0x4C34</td>
<td>Miss</td>
<td>3</td>
</tr>
<tr>
<td>0x3A18</td>
<td>Miss</td>
<td>1</td>
</tr>
<tr>
<td>0x721C</td>
<td>Hit</td>
<td>1</td>
</tr>
<tr>
<td>0xAC4</td>
<td>Hit</td>
<td>4</td>
</tr>
<tr>
<td>0x3A1C</td>
<td>Hit</td>
<td>1</td>
</tr>
</tbody>
</table>

(C) (2 points) Could the four 32-bit addresses 0x5960, 0x59E8, 0x6D14, and 0x9264 be resident in the cache at the same time? Explain why you chose your answer.

0x5960, 0x59E8, and 0x9264 all map to index 6. Since the cache is 2-way associative, all three of these addresses cannot be resident in the cache at the same time.

All four addresses can be resident at the same time:  YES  NO
Problem 2. Virtual Memory (18 points)

For the following questions, consider a RISC-V processor with 32-bit virtual addresses, 24-bit physical addresses, and a page size of 1024 (= $2^{10}$) bytes. The page table of this processor uses an LRU replacement policy to bring missing pages from disk, handled by a page fault handler.

(A) (2 points) Assuming that each page table entry includes a dirty (D) bit and a resident (R) bit, what is the size of the page table?

Number of entries in the page table: $2^{22}$

Width of each page table entry (bits): 16

The first 8 locations of the page table before the program below is executed are shown to the right; the least recently used page (“LRU”), and next least recently used page (“Next LRU”) are as indicated.

Assume that all the code and data for handling page faults is located on physical page 0 and that all the pages in physical memory are in use. You now run the following code.

```
// The PC address below is a virtual address
   .= 0x1804
    li x1, 0x13F8  // x1 = 0x13F8
    addi x2, x0, 4  // x2 = 4
    mv x3, x0       // x3 = 0
loop:
    lw x4, 0(x1)    // address = 0x1400
    add x3, x3, x4
    addi x1, x1, 4
    addi x2, x2, -1
    bnez x2, loop
```

(B) (4 points) Which physical pages have been accessed by executing the code above? List each unique physical page number accessed in the order they were first accessed.

PC = 0x1804 and rest of instructions: VPN = 0b110 = 6: PPN = 0x111
Data at 0x13F8: VPN = 0b100 = 4: PPN = 0xBED
Data at 0x13FC: VPN = 4
Data at 0x1400: VPN = 0b101 = 5: page fault removes LRU (VPN 3) and writes it back to disk, then it assigns PPN 0xE1D to VPN 5.

Physical page numbers: 0x111 (PC), 0xBED (First 2 loads), 0 (page fault), 0xE1D (next 2 loads)

(C) (4 points) Did any page fault occur while executing the code above? If yes, indicate which instruction (e.g., add x3, x3, x4) triggered the page fault and which virtual address it was trying to access. The address might refer to data accessed or to an instruction fetch. If not, write N/A in each subsection below.

Instruction: lw x4, 0(x1)
Virtual address: 0x1400
Our RISC-V processor has a **fully-associative, 4-entry TLB using an LRU replacement policy**. The TLB caches page table entries, which the page fault handler may modify. To ensure correct behavior, our RISC-V processor implements and uses a privileged instruction to invalidate a TLB entry. Assume that all page fault handler accesses do not affect the TLB.

The figure below shows the contents of the TLB before executing the code in the previous page. The least recently used entry (“LRU”) and next least recently used entry (“Next LRU”) in the TLB are as indicated. The TLB always selects the LRU entry for eviction, even if there are other invalid TLB entries that could be used instead.

<table>
<thead>
<tr>
<th>VPN (tag)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6</td>
<td>1</td>
<td>0</td>
<td>0x111</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>1</td>
<td>0xE1D</td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>1</td>
<td>0xABE</td>
</tr>
<tr>
<td>0x0</td>
<td>1</td>
<td>0</td>
<td>0xFED</td>
</tr>
</tbody>
</table>

(D) (6 points) Fill in the contents of the TLB after executing the code from the previous page. If an entry has not changed, you may write NO CHANGE over it instead of copying it. You do not need to indicate the LRU entry.

<table>
<thead>
<tr>
<th>VPN (tag)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO CHANGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4</td>
<td>1</td>
<td>0</td>
<td>0xBED</td>
</tr>
<tr>
<td>0x5</td>
<td>1</td>
<td>0</td>
<td>0xE1D</td>
</tr>
<tr>
<td>NO CHANGE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(E) (2 points) How many TLB misses occur while executing the code from the previous page?

**TLB Misses: 2**
Problem 3. Pipelining Combinational Circuits (16 points)

The following combinational circuit uses 6 F modules and 3 G modules to compute a 3 bit output Y from two 3-bit inputs A and B.

(A) (3 points) An F module has a propagation delay $t_{PD}$ of 8 ns and a G module has a $t_{PD}$ of 5 ns. What is the latency and the throughput of the combinational circuit above?

Latency (ns): ______ 37 _______ Throughput (ns$^{-1}$): ______ 1/37 _______

(B) (4 points) On the above circuit diagram, using filled-in circles, indicate appropriate locations to place ideal (zero-delay) registers to pipeline the circuit for a throughput of $\frac{1}{4t_{PD}(F)} = \frac{1}{32}$ ns. What is the latency of your pipelined design? Be sure to use a minimum number of registers and include a register on each output.

Latency (ns): _____ 64 _____
(C) (4 points) Repeat (B) to pipeline the circuit for a throughput of $\frac{1}{2} t_{PD(F)} = \frac{1}{16} \text{ns}$. Be sure to use a minimum number of registers and include a register on each output.

Latency (ns): ______ 48 _______
(D) (5 points) Repeat (B) to pipeline the circuit for a throughput of $\frac{1}{t_{PD(F)}} = \frac{1}{8} \text{ ns}$. Be sure to use a minimum number of registers and include a register on each output.

Latency (ns) : \underline{40}
Problem 4. Pipelined Processors (20 points)

For the following questions, assume that you are running code on the 4-stage pipelined processor whose stages are shown below. Also assume that:

⇒ there is no bypassing,
⇒ the register file (RF) is not a bypass register file,
⇒ there are no early writebacks (meaning that every instruction must go through the last stage),
⇒ there is a register between pipeline stages (not a FIFO buffer), and
⇒ data memory responds immediately (in the next clock cycle) to any load request.

The problems will ask about pipeline hazards; we will say a hazard is observable if it causes the pipeline to stall.

Note: For some problems it might be helpful to draw a pipeline diagram. There are several blank pipeline diagrams on the next page that you can use as well as additional ones on the last page of the exam.

0x100    addi sp, sp, -4
0x104    srli a1, a0,  1
0x108    addi a4, a4, +1
0x10C    beqz a0, L0
0x110    andi a2, a1, 0x001
0x114    L0:    sub  a3, a0, a2
0x118    lw   a5, +4(sp)
0x11C    sw   a3, 0(sp)

(A) (3 points) Identify all the potential read-after-write (RAW) data hazards in the code above (including ones that are not observable). The number at the beginning of each line corresponds to the address of that line of code. For each hazard, write Lines a and b if a register is written in the line at address a and read in the line at address b. If there are more spaces than you need, leave the extra spaces blank.

1) Lines 0x104 and 0x110 srli a1 – andi a1
2) Lines 0x110 and 0x114 andi a2 – sub a2
3) Lines 0x114 and 0x11C sub a3 – sw a3
4) Lines 0x100 and 0x118 addi sp – lw sp
5) Lines 0x100 and 0x11C addi sp – sw sp
6) Lines _______ and _______
For each of the scenarios below, whenever you are asked to list the observable hazards, specify the number to the left of the hazard you are referring to from part A. So, if the hazards labelled (1) and (3) in part A of your answer are observable, the enter 1, 3 as your observable hazards response. Enter None if there are no observable hazards.

(B) (4 points) Which of the hazards from part A are observable when the branch (address 0x10C) is taken and is correctly predicted as being taken?

List of observable hazards or None: ______3 (0x114-0x11C, sub & sw)__________

See pipeline diagram on next page.

(C) (4 points) Which of the hazards from part A are observable when the branch is not taken and is correctly predicted as being not taken? How many nops are inserted (i.e., for how many cycles is the pipeline stalled) in this case?

List of observable hazards or None: ___2 (0x110-0x114, andi & sub), 3 (0x114-0x11C, sub & sw)___

Number of nops/stalled cycles: _____3__________

See pipeline diagram on next page.

(D) (4 points) Now reconsider part (C) where the branch is not taken and is correctly predicted as being not taken. However, this time assume that there is a cache miss fetching the lw instruction at address 0x118 causing a three cycle stall. Which hazards are now observable?

List of observable hazards or None: ____2 (0x110-0x114, andi & sub)____________

Delay of lw fetch lets sub complete by the time sw is in the DEC stage.

(E) (5 points) Once again reconsider part (C) where the branch is not taken and is correctly predicted as being not taken. However, now suppose we add bypassing from the writeback stage to the decode stage (or equivalently, we replace the register file with a bypass register file), and the instruction cache no longer misses. Now which hazards are observable? How many nops are inserted (i.e., for how many cycles is the pipeline stalled) in this case?

List of observable hazards or None: _____2 (0x110-0x114, andi & sub)____________

Number of nops/stalled cycles: _____1__________

Bypass from WB to DEC saves one cycle for sub so only 1 NOP instead of 2.
<table>
<thead>
<tr>
<th>Branch taken</th>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>srli</td>
<td>srli</td>
<td>srli</td>
<td>srli</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>beqz</td>
<td>beqz</td>
<td>beqz</td>
<td>beqz</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>Sub</td>
<td>sub</td>
<td>sub</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw</td>
<td>lw</td>
<td>sw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch not taken</th>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>srli</td>
<td>srli</td>
<td>srli</td>
<td>srli</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>beqz</td>
<td>beqz</td>
<td>beqz</td>
<td>beqz</td>
</tr>
<tr>
<td></td>
<td>andi</td>
<td>andi</td>
<td>andi</td>
<td>andi</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>Sub</td>
<td>sub</td>
<td>sub</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch not taken 2</th>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>sub</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.004 Spring 2019 - 10 of 17 - Quiz #3
Problem 5. Ephemeral History Registers (EHRs) (16 points)

Your UROP is focusing on creating a dedicated piece of hardware using Bluespec to do certain types of calculations, and you’ve been tasked with pipelining the module. You decide to start by brushing up on your use of EHRs to ensure that the pipeline stages (first and second) can run in parallel. You are provided with the circuit diagram and conflict matrix of EHRs below.

(A) (4 points) For each of the following sets of rules, fill in the corresponding conflict matrix for the rules with C representing conflicting, CF representing conflict free, and < and > representing that the corresponding rule of the row appears to execute before and after the corresponding rule of the column, respectively. Assume there are only 4 EHRs w, x, y, and z in use in each version.

**Version 1**

```
rule first;
  x[0] <= z[0] + 1;
  y[1] <= x[0];
endrule

rule second;
  z[0] <= x[1] >> 1;
  y[0] <= w[0] - 2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>second</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

**Version 2**

```
rule first;
  w[1] <= y[0] % 2;
  y[0] <= x[1];
endrule

rule second;
  w[0] <= x[0] + 8;
  z[1] <= y[0] << 2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>second</td>
<td>&lt;</td>
<td>C</td>
</tr>
</tbody>
</table>
Now that you’ve had some practice with EHRs, you move on to using EHRs to successfully pipeline the module from your UROP’s system. Below are the parts of the implementation of the module relevant to the pipelining. (Note: this is only part of the full code, and registers a, b, and writing are updated by other parts of the module. However, you should not worry about whether this snippet conflicts with any other part of the module.)

Reg#(Bit#(32)) a <- mkReg(0);  _____
Reg#(Bit#(32)) b <- mkReg(0);  _____
Reg#(Bit#(32)) init <- mkRegU();  _____
Reg#(Bit#(32)) res <- mkRegU();  _EHR_
Reg#(Bit#(8)) random <- mkRegU();  _EHR_
Reg#(Bool) writing <- mkReg(False);  _____

rule doInit;
  init <= initialize(a , b , random[1] );
endmethod

rule doComp;
  res[0] <= compute(init , random[0] );
  random[0] <= getRand(init , random[0] );
endrule

rule writeRes if (writing );
  log.write(res[1] , random[1] );
endrule

In order for the pipelining to work properly, the module has been specified to follow this conflict matrix.

<table>
<thead>
<tr>
<th></th>
<th>doInit</th>
<th>doComp</th>
<th>writeRes</th>
</tr>
</thead>
<tbody>
<tr>
<td>doInit</td>
<td>C</td>
<td>&gt;</td>
<td>CF</td>
</tr>
<tr>
<td>doComp</td>
<td>&lt;</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>writeRes</td>
<td>CF</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

(B) (8 points) In order to satisfy this conflict matrix, for each register that needs to be an EHR, convert all uses of the register into that of an EHR by placing brackets [ ] with indices in them that give EHR orderings that correctly implement the specified conflict matrix. You may only use indices of 0 or 1. Also write EHR on the blank line next to the register declaration to indicate that the register is being converted to an EHR. Use the minimum number of EHRs.
After your additions, you find that the stages of the module correctly execute in parallel, but you discover a bug. Since your UROP’s system is not writing results every cycle (i.e. `writing` is sometimes false), the previous stages may continue to execute with new inputs and overwrite the result that was not yet written. You know that the system will wait a while in between printing results, so you decide that using a FIFO queue to store the pending results will be better than stalling your pipeline for a long time. This FIFO queue will replace the `res` register used in `doComp` and `writeRes`, as shown in green below. Assume that all the EHR modifications you made above are present in the code below.

```plaintext
Reg#(Bit#(32)) a <- mkReg(0);
Reg#(Bit#(32)) b <- mkReg(0);
Reg#(Bit#(32)) init <- mkRegU();
FIFO#(8, Bit#(32)) res <- mkFifo(8, Bit#(32));
Reg#(Bit#(8)) random <- mkRegU();
Reg#(Bool) writing <- mkReg(False);

rule doInit;
    init <= initialize(a , b , random );
endmethod

rule doComp;
    res.enq(compute(init , random ));
    random <= getRand(init , random );
endrule

rule writeRes if (writing );
    log.write(res.first(), random );
    res.deq();
endrule
```
(C) (4 points) In order to maintain the same conflict matrix from part B, what type of FIFO must res be replaced with: Pipeline or Bypass? For reference, their implementations are given below.

**Correct FIFO Type:** Pipeline FIFO .... Bypass FIFO

**Pipeline FIFO**

```verilog
module mkFifo (Fifo#(1, t));

  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);

  method Action enq(t x) if (!v[1]);
    v[1] <= True; d <= x;
  endmethod

  method Action deq if (v[0]);
    v[0] <= False;
  endmethod

  method t first if (v[0]);
    return d;
  endmethod
endmodule
```

**Bypass FIFO**

```verilog
module mkFifo (Fifo#(1, t));

  Ehr#(2, t) d <- mkEhr(?);
  Ehr#(2, Bool) v <- mkEhr(False);

  method Action enq(t x) if (!v[0]);
    v[0] <= True; d[0] <= x;
  endmethod

  method Action deq if (v[1]);
    v[1] <= False;
  endmethod

  method t first if (v[1]);
    return d[1];
  endmethod
endmodule
```
Problem 6: Semaphores (14 points)

The Chiquita Fruit Company has acquired a fancy multi-process machine that they want to use to print the word BANANA. The machine has two types of processes – called BA and NA – that can coordinate their execution via shared semaphores which respond to the standard signal(S) and wait(S) procedures. Since we need twice as many NAs as BAs, there’s one BA process and two NA processes running on the machine. Assume that execution may switch between any of the three processes at any point in time.

Their summer intern wrote many versions of the BA and NA code, then made a list of the first six characters they printed (any subsequent printed characters were not recorded) when run on the machine:

(a) BANANA  (b) ANANAB  (c) BNAAA  (d) BANAA  (e) BANB  (f) NABANA

For each version below, please indicate which of the strings above, if any, it could have printed by circling the corresponding letters (e.g., circle a and b if the code could print BANANA and ANANAB but not any of c-f).

(A) (3 points)

Process BA
Loop1:
print(“B”)
print(“A”)
goto Loop1

Process NA
Loop2:
print(“N”)
print(“A”)
goto Loop1

circle all strings (a-f) that could be printed: … a … b … c … d … e … f

A can’t precede B or N.

(B) (8 points)

Semaphore s = 0; Semaphore t = 0;

Process BA
Loop1:
print(“B”)
print(“A”)
signal(s)
signal(s)
wait(t)
goto Loop1

Process NA
Loop2:
wait(s)
print(“N”)
print(“A”)
signal(t)
goto Loop2

circle all strings (a-f) that could be printed: … a … b … c … d … e … f

NA execution can be interleaved.
(C) (3 points) A clever 6.004 student observes that interchanging two lines in the Process BA code of part (B) will ensure that only BANANA will be printed. Show which two lines she interchanged by circling them in the Process BA code in part (B).

Circle 2 lines to interchange in process BA code in part (B)

See oval in part B.
<table>
<thead>
<tr>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>WB</th>
</tr>
</thead>
</table>

END OF QUIZ 3!
HAVE A GREAT SUMMER!