Final Exam Solutions

CSEE W3827 - Fundamentals of Computer Systems — April 22, 2021
Spring 2021 — Prof. Rubenstein

This final contains **??** questions (not counting question 0), totaling 90 points. Question 0 gives an additional 10 points. To get full credit you must answer all questions. **BOOKS AND NOTES ARE PERMITTED, ELECTRONIC DEVICES CAN BE USED FOR NON-COMPUTATIONAL PURPOSES AND NON-SEARCH PURPOSES.** The time allowed is 180 minutes, plus an additional 15 minute grace period to deal with upload/download issues.

Please format your exam as follows:

- Start each question on a separate page
- The file you upload should start with your UNI (preferably all lower-case).

Some advice:

- Be sure to leave some time to work on each problem. The right answer to each problem does not require a very long answer.
- Be sure to start every problem. And take some time to think about how to set the problem up before you start writing.

0. **10 points** The filename of your upload must start with your UNI (all lower case letters preferred) to receive these 10 points.

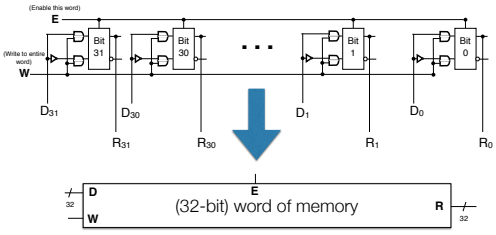
1. Consider the following 3-instruction code snippet that checks for equality of values in two locations of memory:

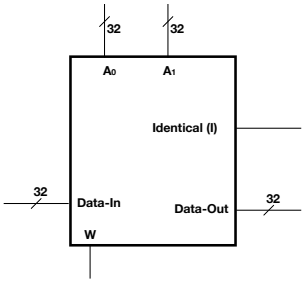
```
lw $s0, 0($s2)
lw $s1, 0($s3)
beq $s0, $s1 THEY_ARE_THE_SAME
```

The code uses three clock cycles, without even accounting for potential pipeline stalls. Suppose the comparison could be performed within memory itself using a single instruction we'll call **bme** (for branch if memory equal) of the form:

```
bme $s2, $s3, THEY_ARE_THE_SAME
```

where, as in the original snippet, $s2 and $s3 store the addresses where the comparison is being performed.





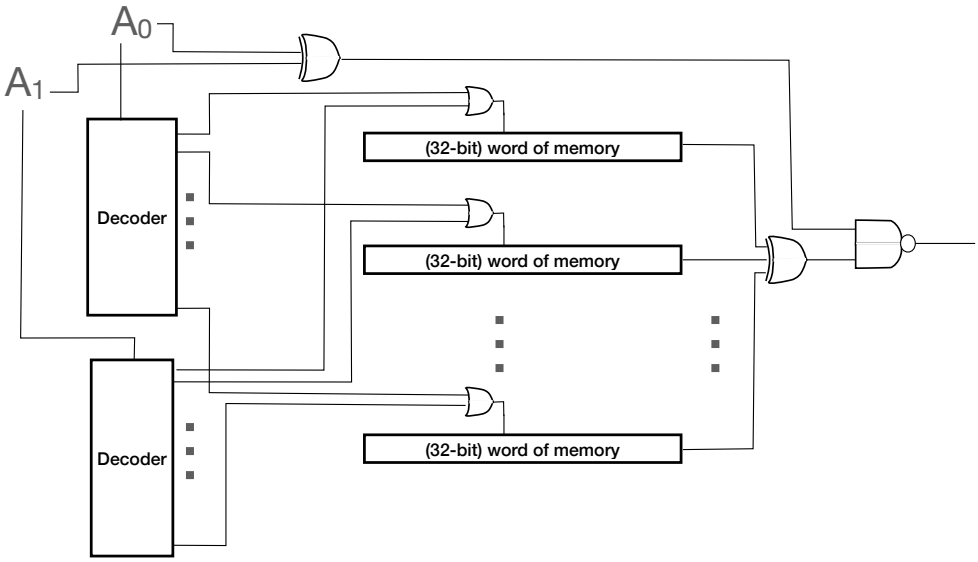
The figure on the left shows how a standard memory word slice is constructed. Using these standard memory word slices, design this new memory, depicted in the figure on the right, with the following properties:

- In addition to a first 32-bit address input (which we will call A_0), it has a second 32-bit address input, A_1.
- It has an additional 1-bit output I that equals 1 when the **contents** at addresses A_0 and A_1 are identical, and equals 0 otherwise.

In addition to the word slices, you may use any additional circuitry you think you need to complete the design (e.g., MUXes, Enablers, Decoders, AND gates, OR gates, NOT gates, XOR gates, etc.)

You need to only draw circuitry that shows how this extension is implemented. Do not worry about replicating the circuitry needed for normal operation (i.e., you don't have to show the circuitry that performs "normal" reads and writes from memory).

Answer: Two addresses need to be read from memory simultaneously, then compared, and if equal, return 1 (else return 0). This could be done with two decoders, each taking an address as the selector, and enabling the corresponding memory word (so the input to the memory cell enable is the OR from both decoder outputs). All memory words can then be XOR'd together (since all will be 0 except two of them). If the memory words are identical, the result will be all 0's, so just OR the bit results together, and complement.
One final (subtle) detail: if the two addresses were the same, then only one memory word would be XOR'd with all 0's, and if the memory word isn't 0, the above description would be wrong. So we should also return 1 for the case where the two addresses are the same.



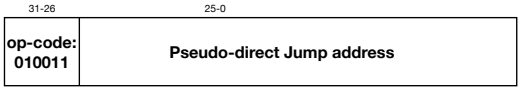
Note that another solution would use 2 MUXs, where the memory words are the inputs to the MUX's, and the addresses are the selectors. An XOR can combine the outputs from each MUX, and there is no need check if the addresses are the same.

2. (30 points) A procedure often places its own return address value (in the $ra register) on the stack when making its own call to another procedure. To return, the procedure must pull the return value off the stack and then return. Sample MIPS code is shown below:

```
## code for jump-to-procedure          ## code for return
addi $sp, $sp, -4                      lw $ra, 0($sp)
sw $ra, 0($sp)                         addi $sp, $sp, 4
jal LABEL                              jr $ra
```

Suppose additional instructions jalm and jrm are added to the architecture that write and read directly from the stack instead of using the $ra register: jalm pushes the return address onto the stack, and jrm pulls the return address from the stack, $ra is not used.

The rest of this probem involves extending the traditional single cycle architecture depicted on the next page.



(a) (15 pts) Given the format of the jalm instruction is pictured above (containing only op-code and 26 bits of an address), enhance the architecture to implement jalm. In particular:

i. (5 pts) How should the various fields exiting Control be set?

Answer: jalm must:

- send PC+4 to be written to memory
- send SP-4 as the address of memory to be written to
- Write this value of SP-4 into SP
- Write the new address (as specified by the 26 bits in the instruction) to the PC (as the j instruction would do)

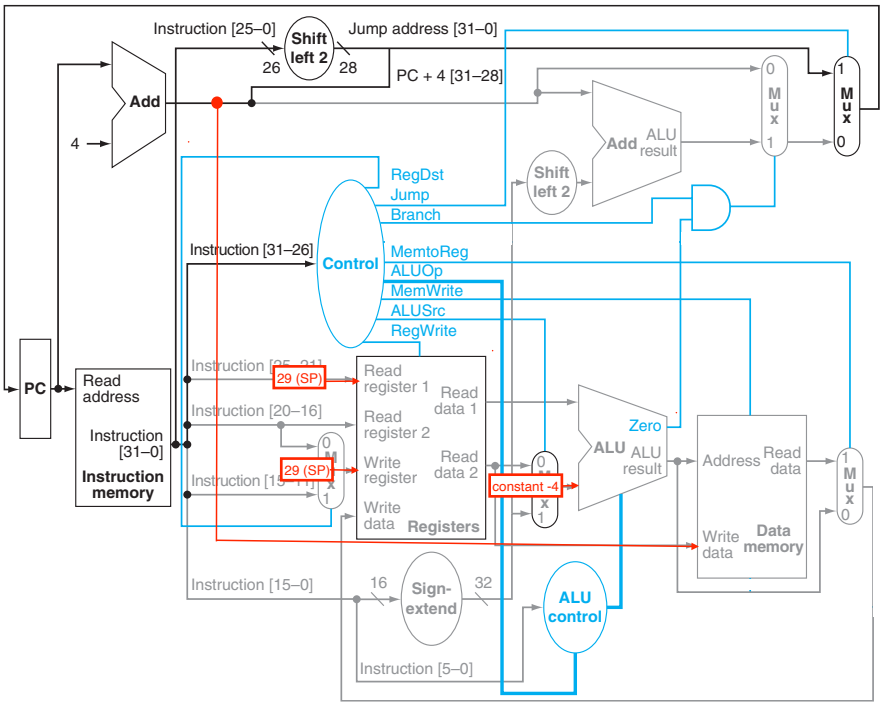
The way we can set the fields is as follows:

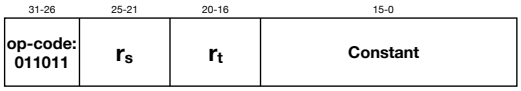
- RegDst: Depends on design: we will be "forcing" 29 (SP) into the register file, where this info cannot be embedded in the instruction.
- Jump: 1 (this will write the pseudo-direct address to the PC as the j instruction would)
- Branch: 0
- MemtoReg: 0 (we will write the ALU result to SP in reg file)
- ALUOp: opcode for subtract
- MemWrite: 1 (writing on the stack)
- ALUSrc: Depends on design (we are going to simply send a constant -4 into the bottom input of the ALU)
- RegWrite: 1 (writing to SP)

ii. (10 pts) Add any circuitry you need to properly implement the instruction. You only need to show circuitry to make jalm work: In particular, assume Control outpus an additional field IsJALM that equals 1 only when the current instruction is a jalm. You do not need to include MUX's that would select on this signal between the existing architecture circuitry and your new circuitry: we (the graders) would know where those are needed.

Hint: It might be useful to know that the stack pointer ($SP) is register 29 in the register file.

Answer: We need to have the register file read from SP (to get the current address that is the top of the stack), subtract 4 from the value (so have a constant of -4 fed into the ALU) and have the register file write to SP as well. So we need to "force" the constant 29 into one of the read register inputs and into the right register input of the register file. We also need to "force" the constant -4 into the ALU. This updates the SP appropriately. We also need to write PC+4 into the top of the stack, so we route the value of PC+4 directly to Write Data of memory. The changes are depicted below in red.





(b) (15 pts) The format for the jrm instruction is depicted above. Note that jalm instruction itself has no parameters, so you can fill in these fields in a way that might help solve the problem. Enhance the architecture to implement jrm. In particular:

i. (5 pts) How should the various fields exiting Control be set?

Answer: jrm must:

- send SP as the address of memory to be read from (note this means we can't use the ALU to add 4, or if we did, we can't send that result directly to memory).
- send the result out of memory to the PC
- Write the value of SP+4 into SP

We can also use the fields in the instruction to access the SP in the register file, rather than "force" constants through another mechanism (though this would also be fine).
In particular, we can set both r_s = r_t = 29 (stack pointer), and set the constant to 0 (so that the value of SP goes directly to the Address field of memory.
The way we can set the fields is as follows:

- RegDst: 0: read r_t, which is set to 29 (SP) the instruction.
- Jump: 1 (doesn't generally have to be 1, but simplifies our circuit diagram, where we just feed Read data of memory into the "jump" MUX).
- Branch: 0 (doesn't matter as above)
- MemtoReg: 0 (perhaps doesn't matter, but in our case we will take SP coming out of the ALU to feed back into the register file (after adding 4).
- ALUOp: opcode for add or subtract or OR (since our constant is 0)
- MemWrite: 0 (writing on the stack)
- ALUSrc: 1 (we want the constant of 0)
- RegWrite: 1 (writing to SP)

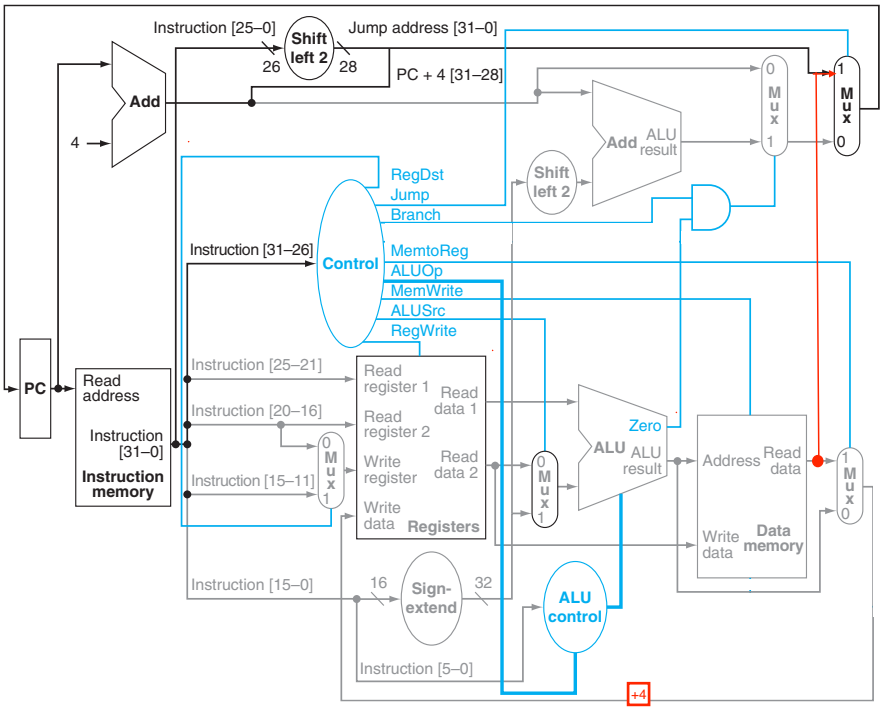
ii. (10 pts) Add any circuitry you need to properly implement the instruction. You only need to show circuitry to make jrm work: In particular, assume Control outpus an additional field IsJRM that equals 1 only when the current instruction is a jalm. You do not need to include MUX's that would select on this signal between the existing architecture circuitry and your new circuitry: we (the graders) would know where those are needed.

Hint: It's probably still useful to know that the stack pointer ($SP) is register 29 in the register file.

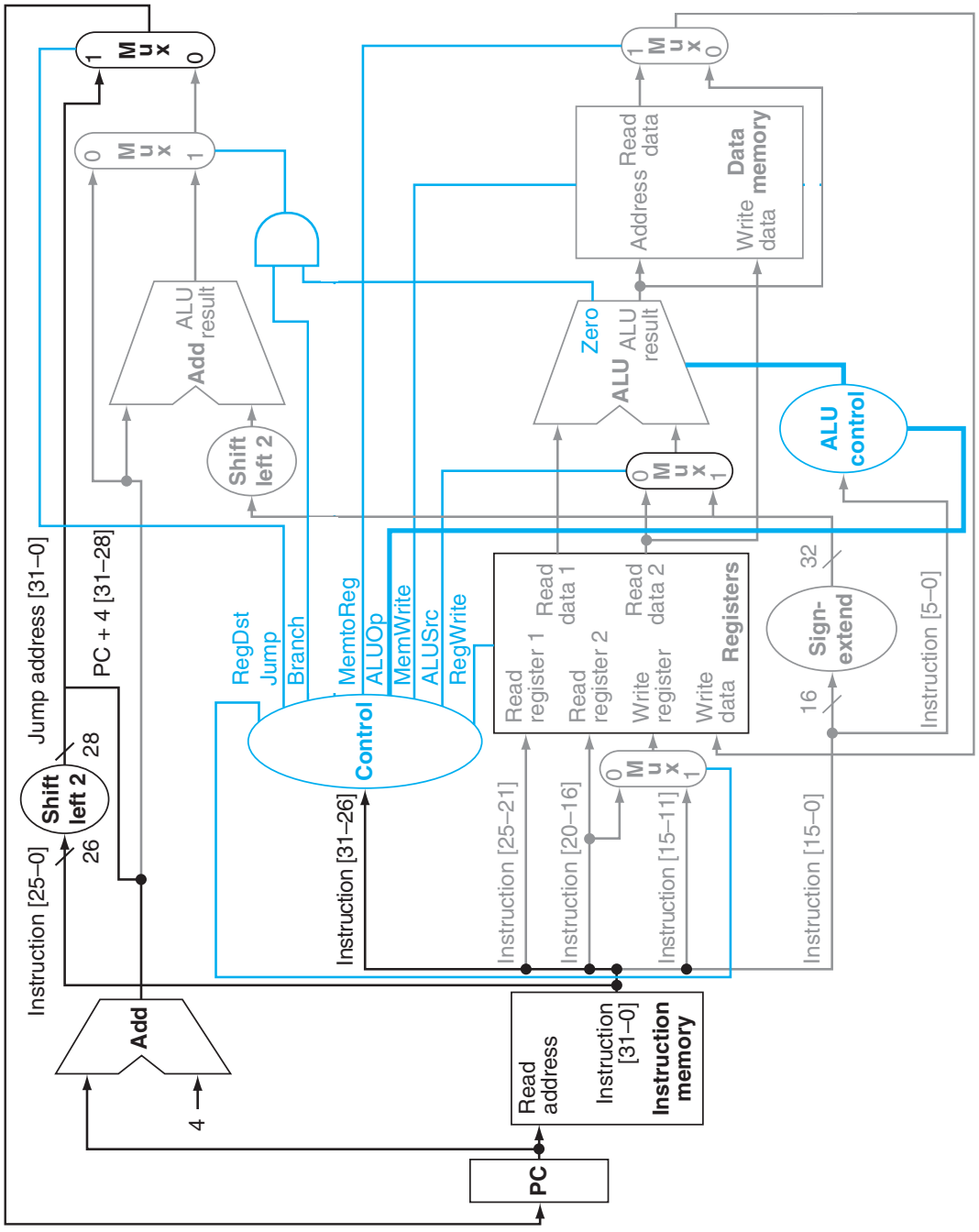
Answer: Again, we need to have the register file read from SP (to get the current address that is the top of the stack) - this time we use the $\$r_S$ field to provide 29 to Read Register 1. The Read Data 1 out of the register file gets sent to the ALU and added to the constant embedded in the jrm instruction (which is 0), and the result from the ALU is fed into the address input of memory to read out the address in the top of stack. The PC must then be set to this value (we do it below by passing it where the jump address would normally be - this is why we set the Jump flag of control to 1.

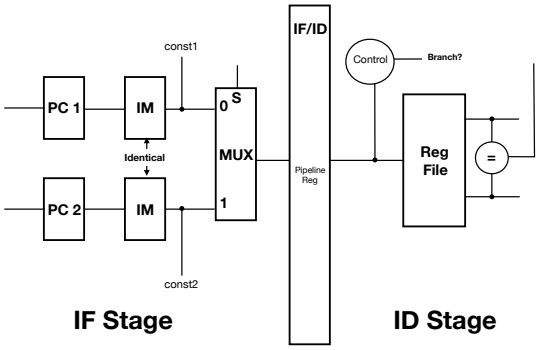
Last, SP must be set to SP+4 (popping the stack). We take the output from the ALU (MemToReg = 0) and add 4 to this result, and feed it into Write data of the register file. The Write register points to the $\$r_t$ field of the instruction, which again contains 29 (SP), so that SP gets written to with its previous value + 4.

The changes are depicted below in red.



(This figure is the single cycle architecture for problem 2.)





3. (30 points) Consider the enhancement to a pipelined architecture depicted above that prevents stalls after a branch instruction. In the IF stage, there are two program counters, each pointing to an identical instruction memory, each fetching an instruction. This enables the fetching of instructions at two different locations in memory. Constants that might be embedded within these instructions can be pulled out at this time as well, indicated as **const1** and **const2**. A MUX selects one of these instructions to be saved within the IF/ID pipeline register for the next stage

   The ID stage is the same as the final pipelined architecture covered in class: the instruction is parsed by Control, which, among other fields, indicates whether the instruction is a branch instruction (via the Branch? output). This instruction also pulls values from the register file and compares their equality (i.e., in case the instruction is beq or bne).

   The idea here is to simultaneously perform branch prediction where one PC (e.g., PC1) assumes the conditional will be false, and where the other (e.g., PC2) assumes the conditional will be true.

   Note for full credit, your solution should work even for code such as what follows:

```
   beq $s0, $s1, LABEL1
   beq $s2, $s3, LABEL2
LABEL1:
   beq $s4, $s5, LABEL3 #somewhere else not shown
LABEL2:
   beq $s6, $s7, LABEL4 #somewhere else not shown
```

   (a) (15 points) Write equations that indicate the next values of the PCs. For instance, in the regular 1-PC architecture, the equations would look like:

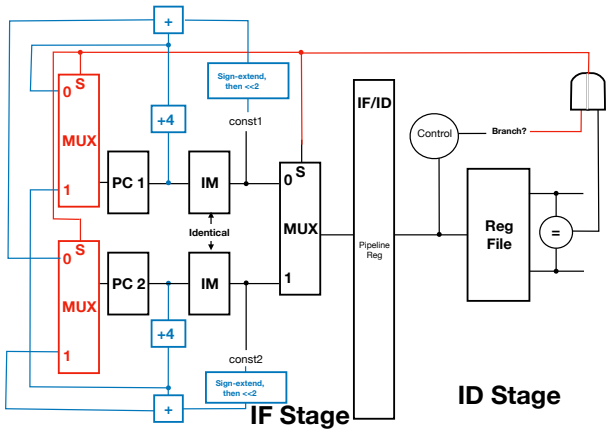
   - PC = PC + 4 + 4*const when the instruction in the ID stage is a branch and the conditional is true.
   - Otherwise, PC = PC + 4

Answer: Detailed explanation provided here: Let's look at a snapshot in a particular cycle t. There is an instruction I in the ID stage, that was pulled from the instruction memory at some address A. Assuming I belongs there (i.e., not a hazard), what should follow it into the pipeline? As described above, if I is not a branch, or it is but the conditional is false, then the instruction $A + 4$ (which we call I_1) should follow. Else, it is a branch and the conditional is true, and $A + 4 + 4 + C_I$ should follow, where C_I is the 16-bit constant embedded in instruction I.
So let's assume that PC1 = $A + 4$ and PC2 = $A + 4 + 4 * C_I$ (such that PC1 has the "regular" pipeline instruction coming next and PC2 has the branch-and-conditional-true instruction coming next. These instructions are being pulled from instruction memory during clock cycle t, and let's call them I_1 and I_2 respectively.
If I was a branch and the conditional was true, then I_2 should proceed through the pipeline, else I_1 should proceed through the pipeline.
What do we set PC1 and PC2 to? If I_1 proceeded through the pipeline, then PC1 held the correct next instruction after I, and the instruction that should follow I_1 will be located either at PC1 + 4 (in the case that I_1 is not a branch or the conditional is false) or at PC1 $+4 + C_1$, where C_1 is the 16-bit constant embedded in I_1 (for the case that I_1 is a branch and the conditional is true).
If I_2 proceeded through the pipeline, then PC2 held the correct next instruction after I, and the instruction that should follow I_2 will be located either at PC2 +4 (in the case that I_2 is not a branch or the conditional is false) or at PC2 $+4 + C_2$, where C_2 is the 16-bit constant embedded in I_2 (for the case that I_2 is a branch and the condional is true.
So:

- PC1 = PC2 + 4 and PC2 = PC2 + 4 + 4 * const2 when instruction I is a branch and the conditional is true
- PC1 = PC1 + 4 and PC2 = PC1 + 4 + 4 * const1 otherwise

(b) (15 points) Complete the diagram above by drawing the circuitry that feeds into each PC (setting its value for the next clock cycle) as well as the circuitry that feeds into the selector of the MUX that decides which instruction remains in the pipeline.

Answer: All MUXes feed off of whether the instruction is branch AND conditional is true. PC2's instruction is selected if so, else PC1's instruction is selected, and the PCs are updated using the formulae from part (a). The figure depicts the MUXes and their feeds in red, and the computations for the next values of the PCs in blue.



You have reached the end of the exam. Have a great Summer!