Department of Electrical and Computer Engineering
University of Texas at Austin

EE460N Spring 2021
Y. N. Patt, Instructor
Siavash Zangeneh, Ben Lin, Juan Paez, TAs
Final Exam
May 14th, 2021

Name:	Solutions
EID:	Extra Explanations

Part A:	Part B:
Problem 1: 10 points	Problem 6: 20 points
Problem 2: 10 points	Problem 7: 20 points
Problem 3: 10 points	Problem 8: 20 points
Problem 4: 10 points	Problem 9: 20 points
Problem 5: 10 points	

Total: 130 points

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Please read the following sentence, and if you agree, sign/print your name where requested: **I have not given or received any unauthorized help on this exam.**

Signature:

Good Luck!

General Instructions:

1. You are free to use anything in the Handouts section of the course website that is listed under "Powerpoint Presentations", "Other Handouts", "LC-3b Handouts". In particular, Appendix A and Appendix C may be of use. Anything else from the course website is not allowed. Anything from any textbooks or the Internet is also not allowed and considered unauthorized access.
2. Use of a calculator is not required but is permitted.
3. If you have any questions, join the class Zoom link and ask a TA. Do not stay on the Zoom call during the exam unless you have questions.
4. Announcements will be posted here. Check this page periodically throughout the exam.
5. You will take the exam by editing a Google Doc. Unlike exam 1, we will not accept handwritten answers by either printing the exam or editing a pdf using a tablet.
6. Read the instructions below.
7. **You are required to stop working on the exam promptly at 10:00 PM.**

Editing a Google Doc:

1. Open the Google Doc version of the exam from here.
2. Save a copy of the document to your Google Drive. Click "File"-> "Make a copy."
3. While working on the exam, **DO NOT expand any boxes that are given to you. Do not change the font size.** Feel free to show your work in the available space. If you need more space, you are writing too much.
4. When you are ready to submit your exam, click "File"-> "Print" and select "Save as PDF".
5. Upload the PDF to Gradescope **by 10:05 PM.**
6. If you fail to upload your exam to Gradescope on time, email your exam to the TAs as soon as possible. Late penalties may apply.

Problem 1 (10 points): Answer each question below. Use as many words as needed, but your answer must fit in the box without expanding it.

Note: On the final exam, you will get no points for any part of Problem 1 that you leave blank.

Part a (2 points): In the LC-3b, what register is used for the Stack Pointer?

R6

Part b (2 points): Does the stack grow towards 0x0000 or towards 0xFFFF?

x0000

Part c (2 points): Does the Stack Pointer have to be changed before pushing a value on the stack? If yes, what is the change?

It needs to be decremented by 2 before storing the value.

Part d (4 points): The PSR contains state information of the executing process. The LC-3b PSR contains the privilege mode bit, priority bits, and condition codes.

Why are the condition codes stored in the PSR? Explain, using a concrete example. If you do not show a concrete example, you will get no credit for your answer.

Suppose the user program has an ADD followed by a branch instruction and an interrupt is taken between the execution of the two instructions. After the processor returns from the interrupt routine, it should use the condition codes that the ADD instructions produced.

We can achieve this by including the condition codes in the PSR, so the condition codes are saved when taking the interrupt and restored after returning from the interrupt.

Problem 2 (10 points): A byte-addressable machine with a 64KB physical memory has an 8-way physically addressed, set associative cache. The cache is write-through, and it uses a pseudo-LRU replacement policy that uses 2 bits per way. The cache has 32 sets. The line size is 16 bytes.

Your job: Provide the following information. Show your work in the boxes provided. **You get no points if you get the correct answer without showing how you got it.**

The capacity of the data store:

Capacity = number of sets * number of ways per set * line size
= 32 * 8 * 16
= 2^(5+3+4) = 2^12 bytes = 4KB

Number of index bits:

Number of sets = 2^ index bits =
32 = 2 ^ 5 = 2 ^ index bits
Therefore, index bits = 5

Number of tag bits:

Tag bits = total address bits - index bits - byte on line bits
= 16 - 5 - 4 = 7

Number of bits in each tag store entry:

Bits in each tag store entry = tag bits + 1 (valid bit) + 0 (write-through cache doesn't need a dirty bit) + 2 (pseudo-LRU bits per way)

Bits in each tag store entry = 7 + 1 + 2 = 10

Problem 3 (10 points): You have just written a new program, but you are unhappy with its long runtime on a single processor. You know that the program has the property that part of the algorithm is parallelizable and the rest has no parallelism at all, i.e., it can only make use of one processor. The parallelizable part can make use of all the processors in your machine, working concurrently. So you run your program on a 16-processor machine and get a 4x speedup. For this problem, assume no communication delay between processors. Show your work.

1. What percent of your program is parallelizable?

Use Amdahl's law to solve for alpha, given S=4 and p=16.
S = 1 / ((alpha / p) + (1 - alpha))
4 = 1 / ((alpha / 16) + (1 - alpha))

alpha = 0.8 = 80%

2. What is the maximum speedup we can achieve with an infinite number of processors?

Use p -> infinity in Amdahl's law:

S = 1 / (1- alpha) = 1 / (1 - 0.8) = 5

3. Since you are not satisfied with the performance, you modify the program so that half of the previously serial portion of the program is now parallelizable. Now what is the maximum speedup we can achieve with an infinite number of processors?

By halving the serial portion, the serial portion becomes 10% of the total serial execution time (alpha = 1 - 0.1 = 0.9)

S = 1 / (1 - alpha) = 1 / (1 - 0.1) = 10

Problem 4 (10 points): We want to multiply 47 (the multiplicand) by 23 (the multiplier) using Booth's algorithm.

How many operations (additions and subtractions) are necessary to perform this multiplication using Booth's algorithm? Show your work.

Some students used a Booth algorithm different from what we studied in class.

Unfortunately, that algorithm gave an incorrect answer to part a. ONLY three operations are needed, rather than the four that were spelled out by that algorithm. That algorithm also included the yet-to-be-used bits of the multiplier in the accumulator, which allows the right shift of the intermediate result and the right shift of the multiplier bits to be done in a single operation. We can do the same thing with the algorithm I taught in class, but I chose not to in the interest of making the explanation clearer., In any case, if you used the web-based Booth algorithm and included the multiplier bits, I did not deduct any points for doing so. I admit I do not know why you used the sub-optimal web-based algorithm instead of what we did in class.

I assume you surfed the web BEFORE the exam to obtain the algorithm, and NOT during the exam since doing so during the exam would be against what was permitted.

Booth's algorithm operates on two bits of the multiplier at a time. The multiplier is 23 = 010111.
The 2-bit groups are: 01, 01, 11
11 requires one subtraction and produces a carry
01 with the carry requires one addition and does not produce a carry
01 without carry requires one addition and the multiply finishes.
Total = 2 additions, 1 subtraction

What is the intermediate result in the accumulator after doing the first operation? Show your work.

Since the first operation is the subtraction due to the first two bits of the multiplier, the accumulator gets the starting value of 0 minus the multiplicand.

Value of the accumulator = 0 - 47 = -47

Or in binary form:

-47 = 1111 1101 0001

Problem 5 (10 points): An out-of-order processor executes instructions based on the original Tomasulo algorithm without in-order retirement. The processor implements a 4-stage pipeline: Fetch, Decode, Execute, Writeback. The Execute stage of the pipeline contains *one pipelined adder* and *one pipelined multiplier*.

- Fetch and Decode take one cycle each.
- The result of a functional unit is broadcast during the writeback stage and is ready for use in the next cycle immediately after writeback.
- Fetch, Decode, and Write Back stages can only operate on one instruction at a time.
- ***An ADD takes 2 cycles*** to execute, and a ***MUL takes 4 cycles***.
- Both functional units have three-entry reservation stations that are initially empty.
- Instructions with no dependencies can start executing immediately after Decode.

The following snippet of code is executed.

```
Instruction 1: ADD  R0, R0, R0
Instruction 2: MUL  R3, R1, R2
Instruction 3: MUL  R4, R0, R3
Instruction 4: ADD  R5, R2, R4
Instruction 5: ADD  R0, R0, R0
```

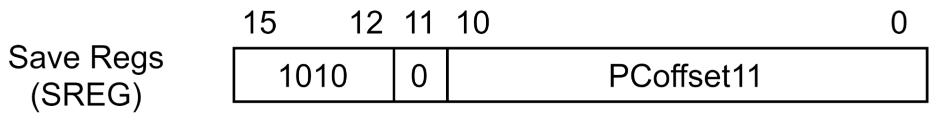
Complete the dynamic timing diagram below for the execution of the program snippet, as we have done in class, from the Decode (D) of Instruction 1 to the Write Back (W) of Instruction 5. We already filled in the Fetch (F) of all five instructions for you.

- Each row corresponds to one instruction.
- Use F, D, M (for MUL), A (for ADD), and W (for write back) to indicate what is going on with each instruction during each clock cycle.
- Use * to indicate a clock cycle when an instruction is waiting to continue processing.
- Use as many columns as needed.

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Instruction 1	F	D	A	A	W															
Instruction 2		F	D	M	M	M	M	W												
Instruction 3			F	D	*	*	*	*	M	M	M	M	W							
Instruction 4				F	D	*	*	*	*	*	*	*	*	A	A	W				
Instruction 5					F	D	A	A	W											

Problem 6 (20 points):

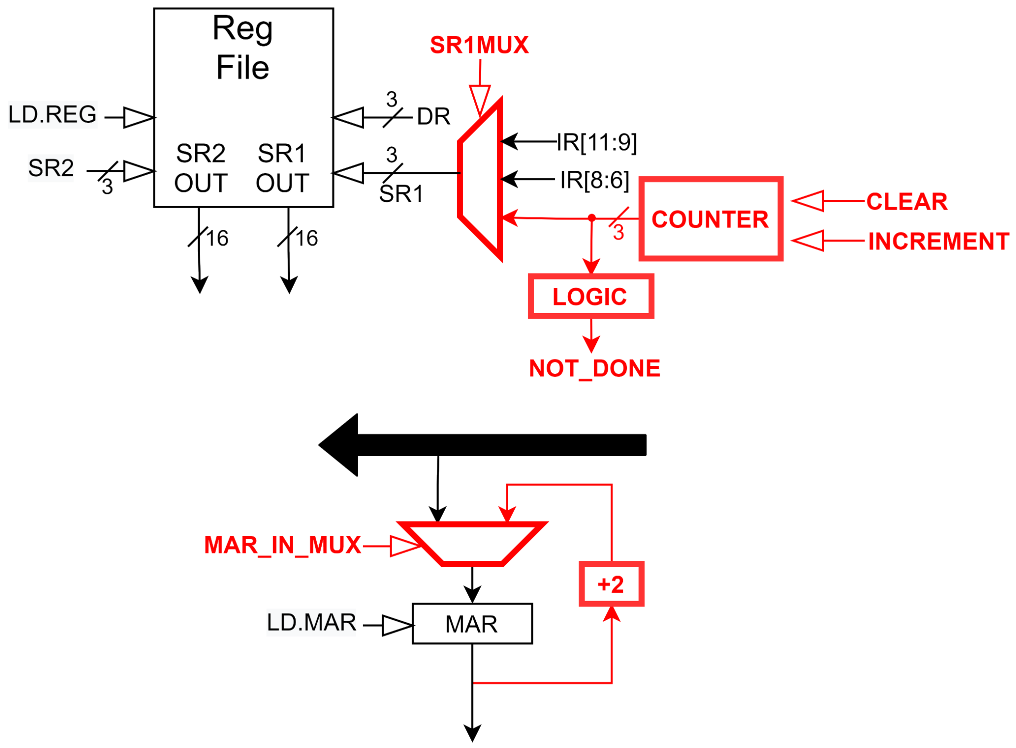
We will use the unused opcode 1010 to create a new instruction, Save Regs (SREG), that saves LC-3b general purpose registers R0-R4 to 5 contiguous memory words specified by PCoffset11.



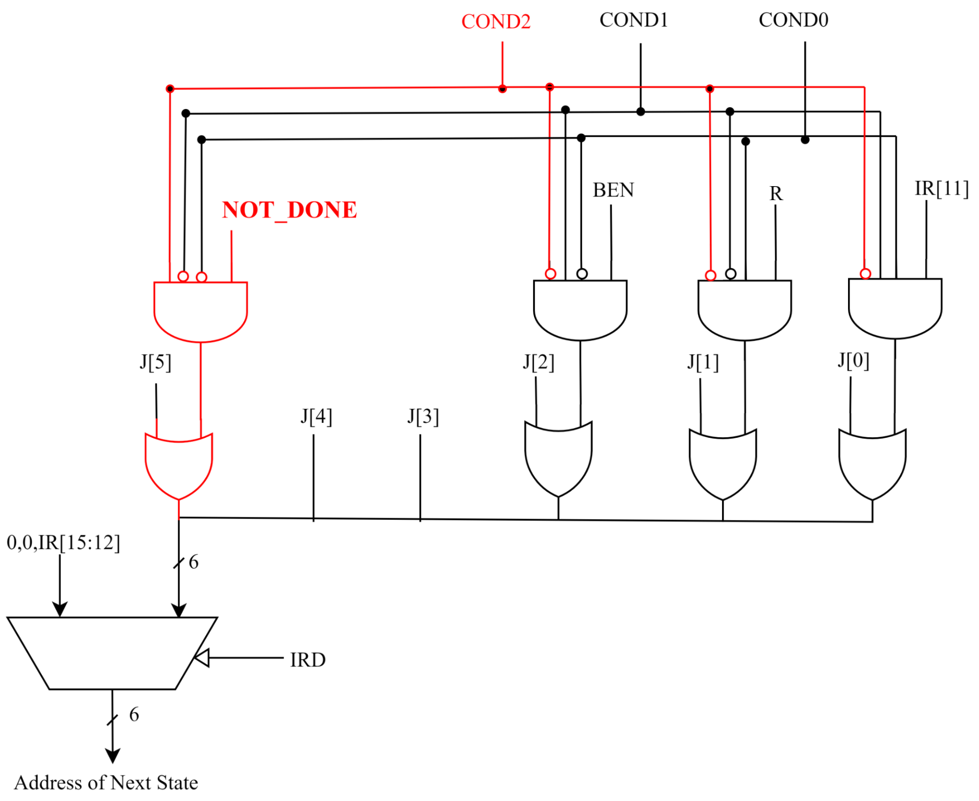
MEM[$PC^{\dagger}$ + LSHF(SEXT(PCoffset11),1)] = R0
MEM[$PC^{\dagger}$ + LSHF(SEXT(PCoffset11),1) + 2] = R1
MEM[$PC^{\dagger}$ + LSHF(SEXT(PCoffset11),1) + 4] = R2
MEM[$PC^{\dagger}$ + LSHF(SEXT(PCoffset11),1) + 6] = R3
MEM[$PC^{\dagger}$ + LSHF(SEXT(PCoffset11),1) + 8] = R4

where $PC^{\dagger}$ is the incremented PC. This instruction can be very useful to subroutines that have to save the registers it will need to perform its function before performing the actual function.

Datapath additions needed to make the instruction work are shown below in red. Note that we added three new control signals: MAR_IN_MUX, CLEAR, INCREMENT, and modified 1 existing control signal: SR1MUX. In addition, the signal NOT_DONE is computed and used by the microsequencer.

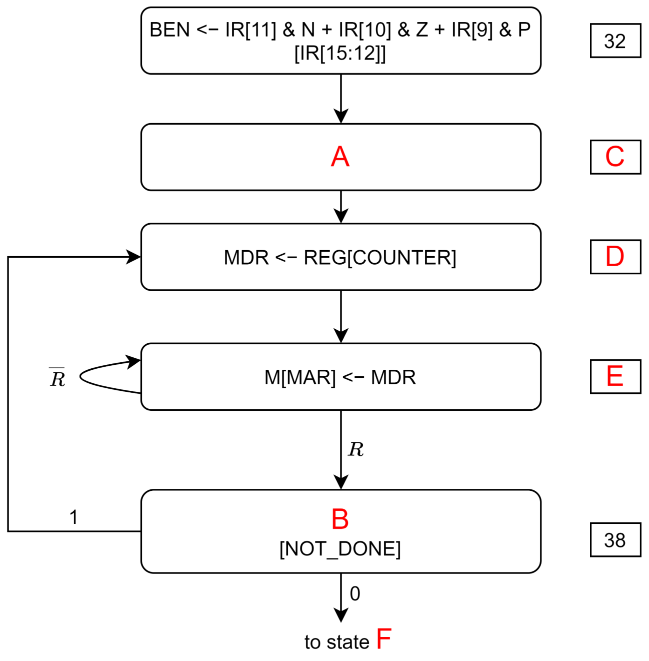


Additions to the microsequencer are shown below:



The state machine to complete the instruction is shown below.

REG[COUNTER] denotes the value of SR1OUT out when SR1 is equal to the 3 bit COUNTER.



Part a (5 points): Fill in the two missing parts (A and B) of their corresponding states of the state machine.

Fill in A:

Clear Counter
MAR ← PC + (SignExtend(PCOffset11) << 1)

Fill in B:

Increment Counter
MAR ← MAR + 2

Part b (3 points): Identify the state numbers C, D, E, and F.

C	D	E	F
10	50 or 51	36	18 or 19

Part c (5 points):

Complete the table with the values of the control signals for each state. If the value does not matter, put an X in the corresponding entry. For example, we have put an X for MAR_IN_MUX for state D.

State #	CLEAR (0 or 1)	INCREMENT (0 or 1)	MAR_IN_MUX (BUS or +2)	SR1MUX (11.9, 8.6, COUNTER)	ADDR2MUX (ZERO, offset6, PCoffset9, PCoffset11)	LD.MAR (0 or 1)	COND (0 - 7)
C	1	0	BUS	X	PCoffset11	1	0
D	0	0	X	COUNTER	X	0	0
E	0	0	X	X	X	0	1
38	0	1	+2	X	X	1	4

Part d (2 points):

What is the LOGIC in the datapath required to compute the NOT_DONE signal used by the microsequencer?

(COUNTER != 4) : NAND(COUNTER[2], NOT(COUNTER[1], NOT(COUNTER[0]))
Alternatively (COUNTER < 4) : NOT(COUNTER[2])

Part e (5 points):

One can use the Save Regs instruction to save the caller register values on a subroutine call. However, the current method of using PCoffset11 in the instruction to specify where in memory to save the registers will not work for **recursive subroutine calls**. Why?

Since PC-relative addressing mode always points to a fixed memory location address, each instance of the recursive call will overwrite the saved registers in previous instances, making it impossible to restore the state when returning after the recursive call.

How can we fix this?

Instead of using PC-relative addressing mode, change the instruction behavior so that it pushes the saved registers on the stack.

After the fix, how many explicit operands will the Save Regs instruction require? Explain.

Zero. There is no need for any explicit operands because the stack pointer is implicitly R6 in the LC-3b ISA (as you know from problem 1, part a).

If you assumed that the stack pointer could be any general-purpose register, and therefore you need one operand to specify the stack pointer register, we graded that as the alternative correct answer. But note that in the LC-3b ISA, R6 is the stack pointer by convention, so assuming that R6 is implicitly the stack pointer is the better answer.

Problem 7 (20 points): We wish to enhance the LC-3b ISA by providing VAX-like virtual memory support as we studied in class. The 16-bit LC-3b addresses you are familiar with are virtual addresses, BUT user space and system space will follow the VAX model in this problem. That is, user space starts at address x0000, system space starts at address x8000, and we use a two-level page table scheme.

Physical memory is 32KB. Page size is 16B. The TLB contains 8 entries and is fully-associative. The TLB is used only for storing user process PTEs. A PTE is 2 bytes. For purposes of this question only, we will assume the PTE has the following form:

V	0000	PFN

We wish to execute the following three instructions sequentially in a program fragment:

At address x3000:	LDW R0, R1, #0	(instruction encoding: x6040)
At address x3002:	ADD R0, R0, #2	(instruction encoding: x1022)
At address x3004:	STW R0, R2, #0	(instruction encoding: x7080)

Before the execution of the three instructions, R1 = x6000, R2 = x6002, and the TLB is initially empty. After the execution of the three instructions, R0 = x4002.

Part a (3 points): Part a (3 points): To execute the three instructions, what user virtual addresses are accessed? Which ones are TLB hits?

Virtual Address	TLB Hit? (Yes/No)
x3000	No
x6000	No
x3002	Yes
x3004	Yes
x6002	Yes

The execution of the three instructions leads to the following physical address accesses. Note that the physical addresses are listed in numerical order, and NOT in the order they are accessed.

Physical addresses: x0400, x0500, x0502, x0504, x0600, x0800, x0802, x10C0, x1180

Part b (5 points): Specify the virtual address corresponding to each physical address. Write "N/A" if a physical address does not correspond to any virtual address. Two entries are given to you.

Physical Address	x0400	x0500	x0502	x0504	x0600	x0800	x0802	x10C0	x1180
Virtual Address	x8600	x3000	x3002	x3004	x8C00	x6000	x6002	N/A	N/A

Each TLB miss accounts for 3 physical accesses (system PTE access, user PTE access, instruction/data access).
Each TLB hit account for 1 physical access (instruction/data access)

So, following the results of part a, we have 3 instruction accesses, 2 data accesses, 2 user PTE accesses, 2 system PTE accesses.

PA x500, x502, x504 are on the same frame, so they must correspond to the instruction accesses with VA x3000, x3002, x3004, which are on the same page and have the same offsets as the PAs.

PA x800, x802 are on the same frame, so they must correspond to the data accesses with VA x6000, x6002 which are on the same page and have the same offsets as the PAs.

The given VAs x8600, x8C00 are system virtual addresses, so they must correspond to the user PTE accesses.

With all the other type of physical accesses accounted for, the remaining two (x10C0, x1180) must be the system PTE accesses, which do not correspond to any virtual addresses.

Part c (3 points): What is the value of the User Process Base Register (PBR)? Show your work.

x8600 = PBR + Page number of (x3000) * size of PTE
x8600 = PBR + x300 * 2

Therefore, PBR = x8000

As discovered in part b, x8600 and x8C00 are the virtual addresses of the PTEs corresponding to VA x3000, x6000, respectively. We know that location x8600 is the PTE for x3000, and location x8C00 is the PTE for x6000 because the PTE addresses are in the same order as their page number. The solution uses the first pair (x8600 and x3000) to derive what is PBR. We could alternatively use the other pair (x8C00 and x6000). x8C00 = PBR + x600 * 2 → PBR = x8000

Part d (3 points): What is the value of the System Base Register (SBR)? Show your work.

x10C0 = SBR + Page number of (x8600) * size of PTE
x10C0 = SBR + x060 * 2

Therefore, SBR = x1000

The process is very similar to part c. We know that location x10C0 is the PTE for x8600, and location x1180 is the PTE for x8C00. The solution uses the first pair (x10C0 and x8600) to derive what is SBR. We could alternatively use the other pair (x1180 and x8C00).
x1180 = SBR + x0C0 * 2 → SBR = x1000

Note that when the address space is partitioned into two equal halves for the user and system regions, the most significant bit of the virtual address does not contribute to the page number.
For example, the page number for address x8000 is 0, which makes sense because x8000 is the first address in the system space. Similarly, page number for x8600, is x060, not x860.

Part e (6 points): What is the 2-byte word stored in each of the corresponding physical memory locations after the three instructions are executed?

Physical Address	x0400	x0500	x0502	x0504	x0600	x0800	x0802	x10C0	x1180
Content	x8050	x6040	x1022	x7080	x8080	x4000	x4002	x8040	x8060

The contents of x0500, x0502, x0504 are simply the encodings of the three instructions.

The contents of x0800, x0802 are the value of R0 before and after the ADD instruction, respectively. Since R=x4002 after the instructions are executed, the two values are x4000 and x4002.

The remaining location are user/system PTEs, so each location is x8000 + the PFN that they point to:

- Location x0400 is the PTE for VA x3000 with PA x0500, so MEM[x0400] = x8050
- Location x0600 is the PTE for VA x6000 with PA x0800, so MEM[x0600] = x8080
- Location x10C0 is the PTE for VA x8600 with PA x0400, so MEM[x10C0] = x8040
- Location x1180 is the PTE for VA x8C00 with PA x0600, so MEM[x1180] = x8060

Problem 8 (20 points): Consider the following addressing scheme for a byte addressable, 12-bit physical memory:

- PA[1:0]: Byte on Bus
- PA[4:2]: Bank
- PA[8:5]: Column
- PA[11:9]: Row

Assume it takes a fixed 20 cycles to open a DRAM row, regardless of whether something was previously in the row buffer, and that it takes 10 cycles to access (read or write) a column in an open row. The memory controller will always issue accesses in order; that is, if the next access targets a busy bank, the controller will wait until the bank becomes idle.

The following **for** loop operates on 32-bit integer arrays, A and B:

```
for (int i = 0; i < 24; i ++) {
     A[i] = B[i] / 2;
}
```

Part a (7 points): We execute this **for** loop using a scalar in-order processor. We know the following:

- Array A begins at physical address x400, and array B begins at physical address x600.
- "Divide by 2" takes 1 cycle to complete (using a right shift).
- Iteration control takes 1 cycle at the end of each iteration.
- There is no caching, and all local variables for iteration control are stored in registers.
- Each iteration of the loop: (1) reads an element of B, (2) right shifts it one bit, (3) stores the result in the corresponding element of A, and (4) performs iteration control.
- Since the processor is in-order, memory accesses are sent to memory one at a time, and a new access can only be initiated after the previous one has completed, even if there is opportunity for interleaving.

1. What is the size of a single DRAM chip in bytes, assuming the DRAM chips have an 8-bit data interface?

$2^{12}/2^{2} = 2^{10} = 1$ KB

The total capacity of memory = 2^12 bytes
Since there are no rank bits or channels bits, we have only one rank and channel.
But width = 4 bytes, and each chip has an 8-bit data interface, so each rank is made of 4 chips that work in lockstep to produce the 4-byte bus width.
Therefore, the capacity of each chip = total capacity of each rank of chips / number of chips

2. If all the row buffers are initially empty, how many times will a new row be opened during the execution of this **for** loop?

   48 times- each iteration, a row is opened to get an element from B, and a row is opened to get an element from A. Since each access to A and B targets a different row in the same bank, each access has to close the row that is in the row buffer and open a new row. Thus, we have to open a new row for every access.

3. How many cycles will it take to execute the **for** loop?

   (30 + 1 + 30 + 1)*24 = 62*24 = 1488

In each iteration the access to both A[i] and B[i] are row misses. So the latency for each iteration is:

- 20 cycles to open a row to get B[i]
- 10 cycles to read B[i]
- 1 cycle to shift
- 20 cycles to open a row for storing A[i]
- 10 cycles to store A[i]
- 1 cycle for iteration control

So total latency per iteration is 62 cycles. The for loop is executed 24 times, ergo 62 times 24 cycles = 1488 cycles total.

Part b (13 points): Now consider a vector processor with 256-element 32-bit vector registers. We use this vector processor to perform the same task as the serial example above using the following vector code:

```
LVS 4            ; load vector stride (1 element is 4 bytes)
LVL X            ; load vector length
VLD V0, B        ; load vector B into vector register 0
VSHFR V0, V0, 1  ; shift every element in V0 right one bit
VST V0, A        ; store V0 into A
```

LVS and LVL each take 1 cycle. Vector loads and stores go directly to physical memory (i.e., no cache).

1. What numerical value should replace X in the line "LVL X"?

   24

2. If all the row buffers are initially empty, how many times will a new row be opened during the execution of this program?

16 times- we open 8 rows (one in each bank) to access B, and 8 rows to access A.

3. How many cycles will the instruction "`VLD V0, B`" take?

57 cycles- first 8 accesses need to open a row, so they each take 30 cycles (but can interleave).
Next 8 accesses can begin as soon as bank 0 is free, at cycle 30.
Last 8 accesses can begin as soon as bank 0 is free again, at cycle 40.
Final access ends at cycle 57. See diagram below.

```
      |----20----|---10---|    ; access 0
       |----20----|---10---|    ; access 1
        |----20----|---10---|    ; access 2
         |----20----|---10---|    ; access 3
          |----20----|---10---|    ; access 4
           |----20----|---10---|    ; access 5
            |----20----|---10---|    ; access 6
             |----20----|---10---|    ; access 7
                          |---10---|  ; access 8 (can start at cycle 30)
                           |---10---|  ; access 9
                            |---10---|  ; access 10
                             |---10---|  ; access 11
                              |---10---|  ; access 12
                               |---10---|  ; access 13
                                |---10---|  ; access 14
                                 |---10---|  ; access 15
                                   |---10---|  ; access 16 (can start at cycle 40)
                                    |---10---|  ; access 17
                                     |---10---|  ; access 18
                                      |---10---|  ; access 19
                                       |---10---|  ; access 20
                                        |---10---|  ; access 21
                                         |---10---|  ; access 22
                                          |---10---|  ; access 23

Final access starts at cycle 40 + 7 = 47 and ends at 57.
So the whole access sequence takes 57 cycles
```

4. How many cycles will the entire program take, assuming no vector chaining?

$\boxed{1 + 1 + 57 + 24 + 57 = 140}$

1 cycle for LVS, 1 cycle for LVL, 57 cycles for VLD, 24 cycles for VSHFR, 57 cycles for VST

= 140 total cycles.

Problem 9 (20 points)

A student has figured out that Dr. Patt uses a computer grader program to compute the final course grade of each student using the following four arrays:

```
int  FINAL_EXAM_GRADE_ARRAY[64];      /* 4 bytes each element, 256 bytes total*/
int  EXAM1_GRADE_ARRAY[64];           /* 4 bytes each element, 256 bytes total*/
int  EXAM2_GRADE_ARRAY[64];           /* 4 bytes each element, 256 bytes total*/
int  LAB_AVG_GRADE_ARRAY[64];         /* 4 bytes each element, 256 bytes total*/
```

The arrays store the {final exam, exam 1, exam 2, lab average} grade for each of the 64 students in the class.

The student strongly suspects that not all four grades are actually used in determining the students' grades. To test this, the student wrote the following dummy program that processed four dummy arrays (`DUMMY1`, `DUMMY2`, `DUMMY3`, `DUMMY4`), identical in size to the four grade arrays that Dr. Patt says he is using.

The dummy program simply adds all 256 values stored in the four dummy arrays. What is important is not what is computed, but rather which memory locations are accessed. We will explain how the student will use this dummy program on the next page.

```
int  DUMMY1[64];                /* 4 bytes each element, 256 bytes total*/
int  DUMMY2[64];                /* 4 bytes each element, 256 bytes total*/
int  DUMMY3[64];                /* 4 bytes each element, 256 bytes total*/
int  DUMMY4[64];                /* 4 bytes each element, 256 bytes total*/
int  x = 0;
for(int i = 0; i < 64; i++) {
      x = x + DUMMY1[i];
}
for(int i = 0; i < 64; i++) {
      x = x + DUMMY2[i];
}
for(int i = 0; i < 64; i++) {
      x = x + DUMMY3[i];
}
for(int i = 0; i < 64; i++) {
      x = x + DUMMY4[i];
}
```

To test his conjecture the student runs the dummy program on Dr. Patt's computer before Dr. Patt runs his grader program. Assume the computer's memory has a 16-bit physical address space, and no virtual memory (all addresses are physical). Assume the computer has a data cache that is:

- physically indexed, physically tagged
- 2KB in capacity
- direct mapped
- with a line size of 256 bytes

The dummy program fills the cache with data from the four `DUMMY` arrays. Assume the student knows where Dr. Patt's four grade arrays will be stored, and so he stores the dummy arrays exactly aligned with the four grader arrays in the cache.

Later, when Dr. Patt runs his grader program, the four grading arrays will evict the student's dummy arrays if and when the grader program accesses them. After Dr. Patt finishes computing final grades, the student sneaks back into Dr. Patt's lab and executes each of his four dummy **for** loops, timing their individual execution times. If a loop's dummy array had been kicked out of the cache by Dr. Patt's grader program, the running time for that loop would be long. If a loop's dummy array had not been kicked out of the cache, the running time would be fast, indicating that the corresponding grader array had never been accessed in computing student grades. Shame on Dr. Patt!

For the student's scheme to work, the four dummy arrays (`DUMMY1`, `DUMMY2`, `DUMMY3`, and `DUMMY4`) have to properly line up with the four grade arrays (`FINAL_EXAM_ARRAY`, `EXAM1_ARRAY`, `EXAM2_ARRAY`, and `LAB_AVG_ARRAY`) in the cache. The student can control this by carefully choosing the starting address for `DUMMY1`. He knows that:

- The four grade arrays are stored in one contiguous sequence of 1KB of physical memory.
- The first grade array `FINAL_EXAM_GRADE_ARRAY` is located at physical address 0x4000.
- The dummy arrays are also stored in one contiguous sequence of 1KB of physical memory, but not necessarily next to the grade arrays.
- Each int is 4 bytes, meaning each array is 256 bytes.

Part a (5 points): Suppose the student picks 0x4400 as the starting address for `DUMMY1`. Will he be able to tell which of the four grade arrays has been accessed? Why or why not?

No! Since the cache is 2KB, and line size is 256 bytes, there are 8 lines in the cache.
Since the cache is direct mapped, that means one line per set, or 8 sets. Bits [10:8] are the index bits that specify the set. Dr. Patt's arrays are at addresses x4000, x4100, x4200, x4300 corresponding to sets 0,1,2,3 in the cache. The dummy arrays are at addresses x4400, x4500, x4600, x4700 corresponding to sets 4,5,6,7 in the cache. Therefore, both sets of arrays fit in the cache and do not collide with each other. As a result, whether Dr. Patt's program accesses an array has no effect on the timing of memory accesses of the dummy arrays in the student's program.

Part b (5 points): Suppose the student picks 0x6000 as the starting address for `DUMMY1`. Will he be able to tell which of the four grade arrays has been accessed? Why or why not?

Yes! This time, the dummy arrays are at addresses x6000, x6100, x6200, x6300 corresponding to sets 0,1,2,3 in the cache. So each array has the same set number as its corresponding array in Dr. Patt's program. Since the cache is direct-mapped, for each of the 4 arrays, either the dummy array can be in the cache or Dr. Patt's array. Therefore, the student can know whether Dr. Patt's program has accessed each of the arrays by measuring the access time for the dummy arrays after the execution of Dr. Patt's program.

Part c (5 points): In general, what condition needs to be satisfied in order for the two sets of arrays to properly line up in the cache so the student would be able to tell which of the four grade arrays has been accessed?

Bit 10:8 (the index bits) of the addresses of each dummy array should be the same as its corresponding array in Dr. Patt's program.

Part d (5 points): How will the student's scheme be affected if the cache had the same capacity (2KB) and line size (256B), but is 2-way set associative instead of direct mapped? Will he still be able to tell which of the four grade arrays has been accessed? Why or why not?

No, now that the cache is 2-way set associative, even if the set numbers of the dummy arrays and Dr. Patt's arrays are the same, both arrays can be in the cache without kicking out the other array. Therefore, whether Dr. Patt accesses an array or not has no impact on the timing of accessing the dummy arrays in the student's program.