CME341 Dec. 21, 2017 Final Exam

Time: 3.0 hours,
Text Books, Notes and Computer Files Only
NO CELL PHONES or LAPTOPS

All questions are independent. Each assumes you are starting with the microprocessor constructed in the preamble.

If a question has more than 1 part, e.g. a) and b), then each part after the first part builds on the previous part . That is to say the modifications in each part builds upon the parts that precede it.

Before starting the exam download all the files in the folder "files_for_2017_final" on the class website to a folder on your H drive.
The path to this folder is Exam Files -> CME341_Exam_Files -> exams_2017_2018.

Remember that `from_PS` must be hardwired to zero in the program sequencer used in the final exam.

The instruction decoder enables the r register on all ALU instructions, which includes `NOPC8`, `NOPCF`, `NOPD8` and `NOPDF`. In your solutions the r register is to remain enabled on NOP instructions even if the question requires the NOP be changed to functional instruction.

1. Make the no-operation instruction that is referred to as NOPC8, which in machine code is 8'HC8, a "execute next instruction twice" instruction. The instruction that follows the NOPC8 instruction is to be executed twice. This is to be done by ensuring `pm_address` remains unchanged from the time `NOPC8` instruction is in the *ir* until after instruction that follows it is executed the first time. For example, if the instruction that follows a `NOPC8` is a successful jump and `NOPC8` is located in program memory at address `8'H7A`, then `pm_address` would be `8'H7B` during the time `NOPC8` is in the instruction register and it would be that value during the time the first of the two times that the successful jump instruction was in the *ir*. During the second time the successful jump instruction is in the *ir*, `pm_address` is set to the target address of the jump.

   Note: A NOPC8 instruction will **not** immediately follow a NOPC8 instruction.

   The program memory file for this question is called
   `CME341_2017_final_Q1.hex`.

   Answer for seed == 8'HAA is 16'H804d

```
     Listing for question 1
0000           org 8'H00;
0000 6E       start:  load i,#4'HE;
0001 51               load m,#4'H1;
0002 C8               NOPC8;    repeat next instruction
0003 BE               mov dm,i;
0004 6E               load i,#4'HE;
0005 C8               NOPC8;    repeat next instruction
0006 8F               mov x1,dm;
0007 C8               NOPC8;    repeat next instruction
0008 E1               jmp loop; should jump on second execution
0010                  align;

0010 28       loop:   load y0,#4'H8;
0011 D3               mulhi x1,y0;
0012 C8               NOPC8;    repeat next instruction
0013 CF               NOPCF;
0014 8C               mov x1,r;
0015 F1               jnz loop;
0020                  align

0020 E2       loop2: jmp loop2;  trapped
```

2. Make the NOPC8 instruction a " write $4'HF$ to registers `x0, x1, y0, y1, M, i` and `o_reg`. That is, NOPC8 is to write all registers, except the r register and data memory, with $4'HF$.

   Also make the NOPCF instruction a write $4'H5$ to registers `x0, x1, y0, y1, M, i` and `o_reg`. That is, NOPCF is to write all registers, except the r register and data memory, with $4'H5$.

   The program memory file for this question is called `CME341_2017_final_Q2.hex`**.**

   Answer for seed == 8'HAA is 16'Hd182

```
     Listing for question 2
0000           org 8'H00;
0000 C8       start: NOPC8; load registers with 4'HF
0001 21             load y0,#4'H1;
0002 C2             add x0,y0; r should be 0
0003 F0             jnz start; should not jump
0004 CF             NOPCF; load registers with 4'H5
0005 1B             load x1,#4'd11;
0006 DA             add x1,y1;  r should be 0
0007 F0             jnz start; should not jump
0008 E1             jmp loop;
0010                align;

0010 E1       loop:   jmp loop; trapped
```

3. Make NOPD8 a circular shift of r 1-bit to the left through the zero flag and NOPDF a circular shift of r 1-bit to the right through the zero flag. A circular shift 1-bit to the left through the zero flag shifts the bits in the r register 1-bit to the left with the most significant bit of r being shifted into the zero flag and the zero flag being shifted into the least significant bit of r. A circular shift 1-bit to the right through the zero flag is similar, but with the shift being in opposite direction. That is, the least significant bit of r is shifted into the zero flag and the zero flag is shifted into the most significant bit of r.

   Note: The zero flag should operate as normal on other ALU instructions.

   Answer for seed == 8'HAA is 16'H3427

   The program memory file for this question is called
   `CME341_2017_final_Q3.hex`.

```
      Listing for Question 3
0000                  org 16'H0;
0000 13       start:  load x1,#8'H3;
0001 30               load y1,#8'H0;
0002 DA               add x1,y1; r=4'b0011, zero_flag = 1'b0
0003 E1               jmp loop1;
0010                  align
0010 D8       loop1:  NOPD8; shift left through zero flag
0011 F1               jnz loop1;
0012 E2               jmp loop2;  at this point zero_flag 1, r = 1000;
0020                  align
0020 DF       loop2:  NOPDF; shift right through zero flag
0021 F2               jnz loop2;
0022 E3               jmp loop3; at this point zero_flag 1, r = 0001;
0030                  align
0030 E3       loop3:  jmp loop3;
```

4. Increase the size of data memory to 32 words and add 1 D flip/flop, referred to as $i4$, to accommodate the extra address bit in the extended data memory. The extended i register will be referred to as i_{ext}, which is given by $i_{ext} = \{i4, i\}$. The extended i register, i.e. i_{ext}, is to be connected to the 5 address lines on the 32 word data memory.

   The most significant bit of i_{ext}, i.e. $i4$, is not affected by moves or loads to the i register. $i4$ is cleared with a sync_reset and also with a NOPC8 instruction and it is set with the NOPCF instruction.

   The auto increment acts on i_{ext} as if it is a single register. For example if $i_{ext} = 5'H0F$ and $m = 4'H1$ then an auto-increment would make $i_{ext} = 5'H10$, which would make $i4 = 1'b1$ and $i = 4'H0$.

   NB: Make `from_CU` = {`3'h0`, i_{ext}}.

   The program memory file for this question is called
   `CME341_2017_final_Q4.hex`.

   Answer for seed == 8'HAA is 16'H5aec

```
     Listing for question 4
0000                    org 8'H00;
0000 52        start:   load m,#4'H2;
0001 61                 load i,#4'H1;
0002 01                 load x0,#4'H1;
0003 E1                 jmp loop1;
0010                    align;

0010 C2        loop1: add x0,y0;
0011 94                mov y0,r; increment y0
0012 BA                mov dm,y0; dm[i_ext]=y0, for i_ext = 1, 3, 5, ..., 31.
0013 F1                jnz loop1;   execute loop1 16 times
0014 50                load m,#4'H0; remove auto-increment
0015 6D                load i,#4'HD;  i_ext = 5'H0D
0016 A7                mov o_reg, dm; o_reg = 4'H7
0017 CF                NOPCF;   i_ext = 5'H1D
0018 A7                mov o_reg, dm; o_reg = 4'HF
0019 C8                NOPC8;   i_ext = 5'H0D
001A A7                mov o_reg, dm; o_reg = 4'H7
001B E2                jmp loop2;
0020                    align;

0020 E2        loop2: jmp loop2;  trapped
```

5. (a) Modify your microprocessor to change the "jump" instruction to a "jump to subroutine" instruction. The `NOPC8` instruction is to be used as the return from subroutine instruction.

   A subroutine can not be called from within a subroutine.

   The program memory file for this question is called `CME341_2017_final_Q5.hex`.

   Answer for seed == 8'HAA is 16'H874a

   (b) First some advice. This question involves many detailed changes making it very tedious and very time consuming. It is strongly suggested that this question be done either last or second last.

   Now the question. Make alternate registers for registers x_0, y_0, and o_reg and call these registers x_{0s}, y_{0s}, and o_reg_s, where the subscript s stands for subroutine. The alternate registers are to be used in place of the original registers when an instruction that references them is located within a subroutine.

   To make the operation clear two examples are given below.

   Example 1: A "`load x0,#4'H5`" instruction encountered within a subroutine loads x_{0s} with `4'H5`. When that same instruction is encountered in the main program it loads x_0 with `4'H5`.

   Example 2: A "`mov x1,y0`" instruction encountered within a subroutine moves y_{0s} to x_1. When that same instruction is encountered in the main program it moves y_0 to x_1.

   Change name of the register o_reg, which is the register used in the main program, to o_reg_m, where the subscript m stands for "main". Leave the microprocessor output called o_reg in tact, but make it the output of a multiplexer that selects either o_reg_m or o_reg_s. Make it select o_reg_s while instructions in a subroutine are executed. Make it select o_reg_m while instructions in the main program are executed.

   The select line for the o_reg multiplexer is to change on the clock edge that executes the "jump to subroutine" instruction and change again on the clock edge that executes the "return from subroutine" instruction.

   To further clarify, the o_reg multiplexer will select o_reg_s from the time the instruction that follows the "jump to subroutine" is in the instruction register up to and including the time the `NOPC8` instruction is in the instruction register.

   The alternate set of registers must be cleared with `sync_reset`.

   NB: make `from_CU` = $\{y_{0s}, x_{0s}\}$.

   Answer for seed == 8'HAA is 16'Hc117

```
                Program Listing for Question 5
0000            org 8'H0;
0000 01       start:  load x0,#4'H1;
0001 12               load x1,#4'H2;
0002 23               load y0,#4'H3;
0003 34               load y1,#4'H4;
0004 77               load dm,#4'H7; dm[0]=7
0005 6F               load i,#4'HF; i=F
0006 7F               load dm,#4'HF; dm[F]=F
0007 E3         jmp subr1;
0008 A0               mov o_reg,x0;
0009 A1               mov o_reg,x1;
000A A2               mov o_reg,y0;
000B A3               mov o_reg,y1;
000C A6               mov o_reg, i;
000D A7               mov o_reg,dm;
000E C2               add x0,y0;
000F A4               mov o_reg,r;
0010 E4            jmp subr2;
0011 E2               jmp trap;
0020                  align;

0020 DA       trap:   add x1,y1;
0021 F2               jnz trap;

0030                  org 8'H30;
0030 08       subr1: load x0,#4'H8; for b) x0s=8
0031 19               load x1,#4'H9;
0032 2A               load y0,#4'HA; for b) y0s=A
0033 3B               load y1,#4'HB;
0034 A7               mov o_reg,dm;  for b) o_regs=dm[i]=dm[F]=F
0035 C8               NOPC8
0040                  align;

0040 A0       subr2:  mov o_reg,x0;   for b) o_regs=x0s
0041 A1               mov o_reg,x1;   for b) o_regs=x1
0042 A2               mov o_reg,y0;   for b) o_regs=y0s
0043 A3               mov o_reg,y1;   for b) o_regs = y1
0044 A6               mov o_reg,i;    for b) o_regs=i=F
0045 C2               add x0,y0;      for b) r = 2
0046 A4               mov o_reg,r;    for b) o_regs=2
0047 C8               NOPC8
```

6. Modify your instruction decoder to add a 5-bit watch dog timer called `watch_dog`. The function of the watch dog timer is described below:

   - `watch_dog` is to be synchronously cleared by `sync_reset`.
   - `watch_dog` is incremented on every positive clock edge, except when it is being reset.
   - While `watch_dog` == 5'd17, it forces `sync_reset` high. To better explain, `watch_dog` forces a synchronous reset when it reaches `5'd17` so that, in the end, it goes from `5'd17` to `5'd0`.
   - If a group of 5 special instructions, where the first is **not** a NOPC8 instruction and the last 4 are NOPC8 instructions, occur in a program, then `watch_dog` is to be cleared on the clock edge that executes the last of the 4 NOPC8 instructions. (Recall an instruction is executed on the clock edge that occurs after it is in the instruction register.)

   Connect `watch_dog` to the 5 least significant bits of `from_ID` as follows
   `from_ID[7:0]`={ 3'H0, `watch_dog` }.

   The program memory file for this question is called
   `CME341_2017_final_Q6.hex`.

   Answer for seed == 8'HAA is 16'Hf4b3

```
          Listing for question 6
  0000           org 8'H0;
  0000 C8       start:  NOPC8;
  0001 C8        NOPC8;
  0002 C8        NOPC8;
  0003 C8        NOPC8; do not reset watchdog timer
  0004 0F        load x0,#4'HF;
  0005 C8        NOPC8;
  0006 C8        NOPC8;
  0007 C8        NOPC8; do not reset watchdog timer
  0008 1F        load x1,#4'HF;
  0009 C8        NOPC8;
  000A C8        NOPC8;
  000B C8        NOPC8;
  000C C8        NOPC8; reset watchdog timer
  000D E1        jmp loop;
  0010           align;

  0010 E1       loop: jmp loop; trapped until watchdog timer goes off

Symbol Table
loop 10H
start 00H
```

7. First some advice: This question is at the least tricky, and perhaps difficult. It probably should be one of the last questions attempted. There are several ways to implement the circuit, at least one of which is very doable, but still a little tricky. One way uses a dual port ram to implement the queue. If that is the method used then it is recommended that the ram be clocked with ∼`clk`, instead of `clk`. The instructor found that using `clk` can produce clocking errors intermittently.

   Now the question: Modify your microprocessor to include a 8-word by 8-bit first-in-first-out queue (like a queue at the bank). NOPDF is to place the `PC` into the Queue and NOPC8 is to remove the oldest element in the queue and put it into an 8-bit register called `queue_reg`. The register `queue_reg` must be synchronously cleared by sync_reset.

   The registers that control the queue should also be initialized with `sync_reset`.

   Connect `queue_reg` to `from_PS` so that the test bench can verify the operation.

   Answer for seed == 8'HAA is 16'Ha013

```
     Listing for question 7
0000            org 8'H0;
0000 1F        start:  load x1,#4'HF;
0001 DF         NOPDF; queue up 8'H1
0002 C8         NOPC8; queue_reg = 8'H01
0003 DF         NOPDF; queue up 8'H03
0004 DF         NOPDF; queue up 8'H04
0005 C8         NOPC8; queue_reg = 8'H03
0006 DF         NOPDF; queue up 8'H06
0007 DF         NOPDF; queue up 8'H07
0008 DF         NOPDF; queue up 8'H08
0009 DF         NOPDF; queue up 8'H09
000A DF         NOPDF; queue up 8'H0A
000B E1         jmp harry;
0010            align;

0010 DF        harry: NOPDF; queue up 8'H10
0011 DF         NOPDF; queue up 8'H11
0012 C8         NOPC8; queue_reg = 8'H04
0013 C8         NOPC8; queue_reg = 8'H06
0014 C8         NOPC8; queue_reg = 8'H07
0015 C8         NOPC8; queue_reg = 8'H08
0016 C8         NOPC8; queue_reg = 8'H09
0017 C8         NOPC8; queue_reg = 8'H0A
0018 C8         NOPC8; queue_reg = 8'H10
0019 C8         NOPC8; queue_reg = 8'H11
001A E2         jmp trap;
0020            align
0020 E2        trap: jmp trap;
```

8. Modify your microprocessor to implement a zero overhead "do while $dm[i_x] \leq 4'H4$" loop, where i_x is the address of the data variable controlling the termination condition.

   The roles of the variables and registers involved are simplified to facilitate implementation in the exam environment. The following simplifications are made:

   - NOPCF is used to mark the beginning of the while loop. The first instruction in the loop is the instruction that follows NOPCF. Modify the operation of NOPCF as necessary to make the do while construct work.
   - A `load i` instruction will always immediately precede a "`NOPCF`" instruction. The value loaded into the i register is the address of data variable that controls the terminations of the loop. For example, if "`load i,#4'H5` precedes "`NOPCF`" then `dm[5]` controls the terminations of the loop.

     Notice from the program listing that there are instructions inside the loop that change the contents of the i register.
   - The loop length will be exactly six instructions.
   - The test to see if the contents of $dm[i_x]$ is less than or equal to $4'H4$ is made while the last instruction in the do-while loop is in the instruction register. Normally the result of the last instruction would be predicted and factored into the decision, but in this case the last instruction in the loop does not alter $dm[i_x]$, which for the program for this question is $dm[7]$, so its result does not need to be predicted to make the proper decision.

     If $dm[i_x] \leq 4'H4$ is true, the next instruction executed is the first instruction in the do-while loop. If $dm[i_x] > 4'H4$, then the next instruction executed is the instruction following the last instruction in the loop.

   Hint: The circuit can be built anyway you want without affecting the scrambler output. If you are having trouble seeing a solution perhaps you will find the suggestions below helpful.

   Build the circuit in the program sequencer. Modify the instantiation of the program sequencer to take in i, dm (i.e. data memory output), the register enable for data memory (i.e. `reg_enable[7]`) and `NOPCF`. Use `NOPCF` to transfer i and dm to newly created registers i_x and dm_ix, the latter being the register that holds a copy of the decision variable, which is $dm[i_x]$. Of course further modifications are necessary to make sure dm_ix is an up to date copy of $dm[i_x]$.

   Answer for seed == 8'HAA is 16'82ba

```
     Listing for question 8
0000                 org 8'H0;
0000 21       start: load y0,#4'H1;
0001 67              load i,#4'H7;
0002 71              load dm,#4'H1; dm[7]=1
0003 68              load i,#4'H8;
0004 7F              load dm,#4'HF; dm[8]=F
0005 67              load i,#4'H7; ix = 7
0006 CF              NOPCF;
0007 67              load i,#4'H7;  i=7,  first instruction in the loop
0008 87              mov x0,dm; xo=dm[7]
0009 C2              add x0,y0;  r = dm[7]+1
000A BC              mov dm,r;   dm[7]=dm[7]+1
000B 68              load i,#4'H8; i=8
000C 70              load dm,#4'H0; dm[8]=0; last instruction in the loop
000D 00              load x0,#4'H0;
000E E1              jmp trap;
0010                 align;

0010 E1       trap:  jmp trap;
```

9. Modify your microprocessor to extend the size of program memory to 1024 words. Of course the extended memory will have 10 address bits, which means a bigger program memory address and a bigger program counter will have to be used. Name the bigger variables `pm_address_big` and `pc_big`.

   All instructions, except the jump and conditional jump instructions work in exactly the same way.

   The target address of jump and conditional instructions will be 10 bits with the most significant 2 bits of the address being the least significant 2 bits of the argument of the jump instruction, which is $ir[1:0]$, and the other 8 bits will be the y_0 and x_0 registers so that the address of jump will be $\{ir[1:0], y_0, x_0\}$.

   The port list for the microprocessor also has an 8-bit output that channels `pc` to the test bench. Connect the least significant 8 bits of `pc_big` to `pc` to satisfy this channel.

   The port list for the microprocessor also has an 8-bit output that channels `pm_address` to the test bench. Connect the least significant 8 bits of `pm_address_big` to `pm_address` to satisfy this channel.

   The program memory file for this question is called `big_program_memory_Q9.hex`.

   The answer for seed == 8'HAA: is 16'H1FF1

```
     Listing for question 9
0000                    org 8'H0;
0000 0E        start:   load x0,#4'HE;
0001 2F                 load y0,#4'HF;
0002 E2                 jmp 8'H20; jump to 10'H2FE
0003 31                 load y1,#4'H1;
0004 1D                 load x1,#4'HD;
0005 21                 load y0,#4'H1
0006 00                 load x0,#4'H0
0007 E0                 jmp 8'H0; jump to loop
0010                    align

0010 DA        loop:    add x1,y1;
0011 8C                 mov x1,r;
0012 F0                 jnz 8'H0; jump to loop if zf=0
0013 22                 load y0,#4'H2;
0014 E0                 jmp 8'H00; jump to trap
0020                    align;

0020 E0        trap:    jmp 8'H00; jump to trap

                        org 10'H2FE
02FE 03                load x0,#4'H3; target of first jump
02FF 20                 load y0,#4'H0;
0300 E0                 jmp 8'H0; jump to 10'H003
```

10. Modify your microprocessor to implement a zero-overhead loop. To do that you will need to build two 4-bit registers: one called `loop_count` and the other called `loop_length_minus_1`.

    The register `loop_count` is loaded upon entry to the loop with the number of times the loop is to be repeated, i.e. the loop is executed one more time than the number in `loop_count`.

    The first instruction in the loop is the one following a `NOPCF`, which for this question is to be interpreted as a "load loop count" instruction. The `NOPCF` instruction is to move the contents of the r register into `loop_count`.

    The `NOPC8` instruction is to changed to be a "move x_0 to `loop_length_minus_1` instruction. That is to say `NOPC8` is effectively a
    "`loop_length_minus_1` $= x_0$ instruction.

    `loop_length_minus_1` must be cleared with `sync_reset` and written to by the `NOPC8` as explained above. `loop_length_minus_1` will be written with a value that is the length of the loop minus 1 sometime before the loop is entered. I.e. If `loop_length_minus_1` = 4'd2, then loop contains 3 instructions.

    `loop_count` must be synchronously cleared with `sync_reset` and loaded with the value in r by a `NOPCF` instruction.

    Make `from_PS = { loop_length_minus_1, loop_count } .`

    Answer for seed == 8'HAA is 16'Hbfeb

```
        Listing for Question 10
0000                    org 8'H0
0000 01       start:    load x0,#4'H1;
0001 22                 load y0,#4'H2;
0002 31                 load y1,#4'H1;
0003 C2                 add x0,y0;
0004 C8                 NOPC8;   mov x0 to loop_length_minus_1
0005 CF                 NOPCF;   mov r to loop_count
0006 CA       lp1_strt: add x0,y1;  start of first loop
0007 9C       lp1_end:  mov y1,r;   end of first loop
0008 02                 load x0,#4'H2;
0009 C8                 NOPC8;   mov x0 to loop_length_minus_1
000A CF                 NOPCF;   mov r to loop_count
000B 00       lp2_strt:  load x0,#4'H0;
000C 01                    load x0,#4'H1;
000D 02       lp2_end:   load x0,#4'H2;
000E E1                 jmp trap;
0010                    align

0010 E1       trap:     jmp trap;
```

11. The effect of this question is to add a second r register called r_a. r_a differs from r in that it is 8-bits and also in that it accumulates the result, i.e. $r_a = \texttt{alu\_out} + r_a$. r_a is to replace r while a flip/flop called `acc_mode` is set. The construction of `acc_mode` and the way that r_a replaces r is described below.

    The flip/flop `acc_mode` is to be cleared with `sync_reset` and also cleared with the `NOPC8` instruction. It is to be set with the `NOPCF` instruction.

    The 8-bit r_a is to be cleared with sync_reset. Its function is to accumulate the 4-bit `alu_out` on all but the NOP ALU instructions while `acc_mode` == `1'b1`, i.e. $r_a = \texttt{alu\_out} + r_a$, and remain unchanged while `acc_mode` == `1'b0`.

    To be perfectly clear, NOPs `NOPC8, NOPCF, NOPD8` and `NOPDF` are to have no effect on r_a even if they occurs when `acc_mode` == `1'b1`.

    The r register will never be enabled at the same time as r_a. The r register is to be enabled on all ALU instructions executed while `acc_mode` == $1'b0$.

    The zero flag is not to be affected by r_a. The zero flag is to be 1'b1 when and only when $r = 0$, regardless of the value of `acc_mode`.

    The rules for when to use r_a and when to use r as the source in a move instruction are as follows:

    - Any move instruction with a source ID of 4'H4 encountered while `acc_mode` == `1'b0` must use the r register.
    - Any move instruction with a source ID of 4'H4 encountered while `acc_mode` == `1'b1` must use the most significant four bits of r_a, i.e. $ra[7:4]$

    NB: Connect r_a to `from_CU` to give it a channel to the testbench, i.e. make
    `from_CU` = r_a.
    r is to remain connected to the testbench as before.

    Answer for seed == 8'HAA is 16'H2498

```
        Listing for question 11
0000                    org 8'H00;
0000 03       start:    load x0,#4'H3;
0001 25                 load y0,#4'H5;
0002 E1                 jmp loop1;
0010                    align;

0010 C2       loop1:    add x0,y0;
0011 84                 mov x0,r;
0012 F1                 jnz loop1;
0013 CF                 NOPCF;    acc_mode = 1'b1
0014 E2                 jmp seg_2;
0020                    align;

0020 C7       seg_2:    com x0; ra = F
0021 C7                 com x0; ra = 1E
0022 C7                 com x0; ra = 2D
0023 C7                 com x0; ra = 3C
0024 C7                 com x0; ra = 4B
0025 A4                 mov o_reg,r; o_reg = ra[7:4] = 4
0026 C8                 NOPC8; acc_mode = 0;
0027 C7                 com x0; r = F;
0028 A4                 mov o_reg,r; o_reg = F
0029 CF                 NOPCF; acc_mode = 1
002A C7                 com x0; ra = 5A
002B A4                 mov o_reg,r; o_reg = 5
002C E3                 jmp trap;
0030                    align;

0030 E3       trap:     jmp trap;
```

12. First some advice. It is believed that this is one of, if not the easiest question. It can be done with a few simple modifications to the instruction decoder.

    Now the question. Modify your microprocessor to change the move instruction to a "move source indirect" instruction.

    The "move source indirect" instruction uses the least significant 3 bits of `o_reg` as the ID for the source register. Furthermore, "move source indirect" **does not** implement "move `i_pins` to a register". The source ID the least significant 3 bits of `o_reg`.

    For an indirect move i is to be auto incremented based on the source and destination ID fields in the move instruction, as it always has been. While the ID of the source of the move will now be the least significant 3 bits of `o_org`, these three bits will not be used to determine whether or not i is to be auto-incremented. For example the mov instruction `mov x0,dm` will auto-increment the i register regardless of the contents of `o_reg` and `mov x0, x1` will not auto-increment the i register, even if `o_reg` $= 4'H7$ and points to dm.

    NB: A reminder that an indirect move can never move `i_pins` to a register. For example, if `o_reg` contains the ID for x_1, i.e. `o_reg` == `4'H1`, and the "move source indirect" instruction is `mov x1,x0`; then it moves x_1 to x_1.

    Answer for seed == 8'HAA is 16'Hbaa9

```
     Listing for question 12
0000                    org 8'H00;
0000 50        start:   load m,#4'H0;      m = 0, and will remain 0 throughout
0001 18                 load x1, #4'H8;
0002 2A                 load y0,#4'd10;
0003 3B                 load y1,#4'd11;
0004 DA                 add x1,y1;          r = 3
0005 44                 load o_reg,#4'd4;   o_reg points to r
0006 88                 mov x1,x0;          indirect move, x1 = r = 3
0007 43                 load o_reg,#4'H3;   o_reg points to y1
0008 88                 mov x1,x0;          indirect move, x1 = y1 = B
0009 6F                 load i,#4'HF;
000A 79                 load dm,#4'H9;      dm[F]=9
000B 47                 load o_reg, #4'H7;  o_reg points to dm
000C 88                 mov x1,x0;          indirect move, x1 = dm[F] = 9
                                  ; do not enable $i$ for auto-increment
000D E1                 jmp trap;
0010                    align;

0010 E1        trap:    jmp trap;
```