Exercises - The ARM-Power model

In the exercices below, you should use the `ppcmem` tool, available online at:

`http://www.cl.cam.ac.uk/~pes20/ppcmem`

1. Consider the MP example below:

```
PPC MP
"PodWW Rfe PodRR Fre"
Cycle=Rfe PodRR Fre PodWW
{
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=x;
}
 P0           | P1           ;
 li r1,1      | lwz r1,0(r2) ;
 stw r1,0(r2) | lwz r3,0(r4) ;
 li r3,1      |              ;
 stw r3,0(r4) |              ;
exists
(1:r1=1 /\ 1:r3=0)
```

   Adding syncs between the instrucitons on each side will forbid the behaviour, but that is rather expensive, so let's look for a cheaper solution. Dependencies are cheaper than syncs, let's try using them:

   (a) In some cases you might want to use dependencies, these instructions may be useful:
   `xor r1, r2, r2` – puts `0` in register `r1`, with a dependency from register `r2`.
   `stwx r1,r2,r3` – stores value at `r1` to the location at `r3` plus offset `r2`.
   `lwzx r1,r3,r5` – loads from location at `r3` with offset at `r2`, placing the result in `r1`.)

   (b) Run your new test in PPCMem, and make it produce the relaxed behaviour above.

   (c) Explain why the dependencies have not forbidden the behaviour.

   (d) Find the cheapest choice of syncs and dependencies on each thread that forbids the behaviour.

2. Consider the following tests read and write from a single location `x`.

```
PPC CoRW (CoFive)                      PPC CoWR (CoFour)
"PPC uniproc, basic reject (ws + rf)"  "PPC uniproc, basic reject (fr)"
{                                      {
0:r5=x; 1:r5=x;                        0:r5=x; 1:r5=x;
}                                      }
P0           | P1          ;           P0           | P1          ;
lwz r2,0(r5) |             ;           li r1,1      |li r1,2      ;
li r1,1      |li r1,2      ;           stw r1,0(r5) |stw r1,0(r5) ;
stw r1,0(r5) |stw r1,0(r5) ;           lwz r2,0(r5) |             ;
~exists (x=2 /\ 0:r2=2)                ~exists (x=1 /\ 0:r2=2)
```

   For each of the tests above, use PPCMem to explore the execution of the test and work out why the behaviour is forbidden.

3. In the lecture, we saw an execution of the code below that made a read request that was later invalidated. Use PPCMem to reproduce this behaviour in the abstract machine.

```
PPC RInvalidate
{ 0:r1=1; 0:r2=x; 1:r2=x; }
 P0           | P1           ;
 stw r1,0(r2) | lwz r1,0(r2) ;
              | lwz r2,0(r2) ;
exists (1:r1=1 /\ 1:r2=1)
```

Exercises - Data-race freedom, compiler optimisations

1. Which of the following programs are data race free? Justify your answer by either showing a 'racy' execution or by giving a reason why there cannot be a data race.

   (a)
   ```
   Thread 1: lock m; *x = 1; unlock m; *y = 1
   Thread 2: lock m; r = *x; unlock m; if (r = 1) then print *y
   ```

   (b)
   ```
   Thread 1: *y = 1; lock m; *x = 1; unlock m;
   Thread 2: lock m; r = *x; unlock m; if (r = 1) then print *y
   ```

   (c)
   ```
   Thread 1: *y = 1; lock m; *x = 1; unlock m;
   Thread 2: lock m; r = *x; unlock m; if (*x = 1) then print *y
   ```

   where `m` is a monitor, `x` and `y` shared-memory locations and `r` is a local variable. Assume that all memory locations are zero-initialised.

2. Let us assume that our language has the DRF principle as its memory model. Which of the following programs can output `42`? Why?

   (a)
   ```
   Thread 1: lock m; *x = 1; unlock m
   Thread 2: lock m; *x = 2; unlock m
   Thread 3: lock m; if (*x = *x) then print 42; unlock m
   ```

   (b)
   ```
   Thread 1: lock m1; *x = 1; unlock m1
   Thread 2: lock m2; *x = 2; unlock m2
   Thread 3: lock m1; lock m2;
             if (*x = *x) then print 42;
             unlock m2; unlock m1
   ```

   (c)
   ```
   Thread 1: lock m1; *x = 1; unlock m1
   Thread 2: lock m2; r = *x; unlock m2;
             if r = 1 then print 1
   ```

   where `m`, `m1`, `m2` are monitors, `x` is a shared-memory location and `r` is a local variable. Assume that all memory locations are zero-initialised.

3. Which of the following program transformations are correct under sequential consistency in any context? For the incorrect ones, give a context and an execution where the transformation introduces a new behaviour. For the correct ones argue how the original program could simulate the transformed one (without going into the details of the simulation relation).

   (a) `r1 = *x; r2 = *y  =>  r2 = *y; r1 = *x`

   (b) `*x = r1; r2 = *y  =>  r2 = *y; *x = r1`

   (c) `r1 = *x; *x = r1  =>  r1 = *x`

   (`r1` and `r2` are local variables, `x` and `y` are shared-memory locations).