THE UNIVERSITY OF KANSAS

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

EECS 743 – Advanced Computer Architecture

Spring 2016

Homework 5

Student Name: Student ID:

SRC® Carte™ Saturn Exercises

Lesson 02b: Loop Dependencies

Objectives:

This exercise will show how loop-carried scalar dependencies are identified and how to remove them to increase the performance of pipelined loops.

Background:

Loop-carried scalar dependencies can force the compiler to slow down a loop. If the next iteration's value of some variable 'x' is computed using the current iteration's value of 'x', then the next iteration cannot be activated until the current iteration's value is computed (*RAW Data Hazard*). An extreme example of this would be a loop containing the statement (x = x / i;). Since division is a long-latency operation, a loop containing this statement will be forced to slowdown.

Exercise 1:

- Open the folder “loop_dependencies_v1”.
- In this exercise, the MAP routine has the line

```
if (AL[i] < 128)
   accum = accum + AL[i];
```

  Here the next value of “accum” depends on a conditional add.

- Compile it in debug mode and note the report:

```
######################################################################
##################        INNER LOOP SUMMARY      ####################
loop on line 14:
    clocks per iteration:    2
    pipeline depth:         30
    loop-carried var 'accum' required 1 additional clock
######################################################################
```

- The MAP library has a family of accumulators that are helpful in eliminating loop slowdown in many cases. The accumulator call gets rid of the cyclic dependency on “accum”. The call prototype is:

cg_accum_**add**_32 (int   a, int in_en, int   r0, int rst_en, int   *r)
cg_accum_**add**_64 (int64_t a, int in_en, int64_t r0, int rst_en, int64_t *r)

  where

a	- value to apply to the internal accumulator
in_en	- enable bit to determine whether to apply **a** to the accumulator (low-order bit only)
r0	- reset value
rst_en	- reset enable, to determine whether to reset the accumulator (low-order bit only)
r	- accumulator output

- In a simple case such as this, the reset value will be zero and the reset should happen in the first iteration of the loop, so the reset enable should be "i==0". The value being accumulated will be "AL[i]" and the enable for the accumulation will be "AL[i] < 128".

- Convert the loop to use the accumulator, test it, and note that the loop is no longer slowed down. **(20 Points)**

Exercise 2:

- An accumulator is an example of a “stateful” functional unit, i.e. one that computes values based not only on the current inputs but also on some internal state that depends on its past inputs. When a loop as the one in the previous exercise terminates, the functional unit retains its state. If the loop is later re-entered, it can continue its accumulation (assuming that the reset enable input does not cause it to reset). This allows an accumulator to work in a nested loop.

V2 – ex_loop_dependencies.mc

```
#include <libmap.h>

void subr (int64_t I0[], int n0, int n1, int64_t *res, int64_t *time, int mapnum) {

    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)

    int64_t t0, t1, v, accum = 0;
    int i, j, sz;

    sz = n0 * n1;

    buffered_dma_cpu (CM2OBM, PATH_0, AL, MAP_OBM_stripe(1,"A"), I0, 1, sz*sizeof(int64_t));

    read_timer (&t0);

    for (i=0; i<n0; i++)
        for (j=0; j<n1; j++) {
               v = AL[i*n1+j];
               if (v < 128) accum = accum + v;
        }

    *res = accum;

    read_timer (&t1);

    *time = t1 - t0; }
```

- Open the folder “loop_dependencies_v2”.
- The code shows a nested loop accumulation. Here the inner loop is pipelined, and it is repeatedly re-entered as the outer loop iterates.
- Compile it in debug mode and note the report:

```
######################################################################
##################        INNER LOOP SUMMARY      ####################
loop on line 17:
    clocks per iteration:    2
    pipeline depth:         31
    loop-carried var 'accum' required 1 additional clock
######################################################################
```

- Modify the code so that it uses an accumulator that retains its state between invocations of the inner loop, and note the performance difference compared with the non-accumulator version. **(20 Points)**
  [Hints:
  - reset enable must be an expression that will reset it only on the first iteration of the first invocation.
  - for evaluating condition expressions use the single “&” instead of the double “&&” to allow both sides of the “&” to be computed concurrently.**]**

Exercise 3:

- As the previous exercise showed, an accumulator keeps its state when its loop is exited. Most of the time this is the desired behavior. However, if the accumulator's output is used in the termination test of the loop, the loop will be slowed down due to an *implicit* loop-carried scalar. The accumulator's output is used to determine the next value of a *hidden* control signal, "loop valid", to the accumulator → (*Control Hazard*).
- It is often ignored whether the state is maintained on loop exit. This is true for example if the accumulator is always reset at the start of its loop. The accumulator has a variant, **"cg_accum_add_64_np"**, that does ***not p****reserve* state. Since it does not need to freeze its state on exit, it does not need to use the "loop valid" signal, which breaks the dependency and eliminates any possible control hazards.

V3 – ex_loop_dependencies.mc

```
#include <libmap.h>

void subr (int64_t I0[], int64_t Out[], int num, int64_t thr, int64_t *idx, int64_t *time,
           int mapnum) {
    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)

    int64_t t0, t1, v, accum = 0;
    int i;

    buffered_dma_cpu (CM2OBM, PATH_0, AL, MAP_OBM_stripe (1,"A"), I0, 1, num*8);

    read_timer (&t0);
    for (i=0; (i<num) & (accum<thr); i++) {
           cg_accum_add_64 (AL[i], AL[i]<128, 0, i==0, &accum);
           BL[i] = accum;
    }
    *idx = i;
    read_timer (&t1);

    buffered_dma_cpu (OBM2CM, PATH_0, BL, MAP_OBM_stripe (1,"B"), Out, 1, *idx*8);

    *time = t1 - t0; }
```

- Open the folder "loop_dependencies_v3".
- The code shows a loop which terminates based on the accumulator value.
- Compile it in debug mode and note the slowdown message:

```
######################################################################
##################        INNER LOOP SUMMARY      ####################
loop on line 15:
    clocks per iteration:    9
    pipeline depth:         36
    loop-carried var 'LOOP VALID' required 8 additional clocks
######################################################################
```

- Modify the code so that it uses the "_np" version of the accumulator, and note that the loop-carried slowdown disappears. **(20 Points)**

Exercise 4:

V4 – ex_loop_dependencies.mc

```
#include <libmap.h>

void subr (int64_t I0[], int64_t Out[], int num, int64_t *nvals, int64_t *time, int mapnum) {

    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)

    int64_t t0, t1, v;
    int i, idx = 0;

    buffered_dma_cpu (CM2OBM, PATH_0, AL, MAP_OBM_stripe (1,"A"), I0, 1, num*8);

    read_timer (&t0);
    for (i=0; i<num; i++)
            if (AL[i] < 128)
                BL[idx++] = AL[i];
    *nvals = idx;
    read_timer (&t1);

    buffered_dma_cpu (OBM2CM, PATH_0, BL, MAP_OBM_stripe (1,"B"), Out, 1, idx*8);

    *time = t1 - t0; }
```

- Open the folder “loop_dependencies_v4”.
- The code shows a common situation, where an array index is conditionally incremented in the loop.
- Compile it in debug mode and note the slowdown message:

```
######################################################################
##################        INNER LOOP SUMMARY      ####################
loop on line 15:
    clocks per iteration:    2
    pipeline depth:         30
    loop-carried var 'idx' required 1 additional clock
######################################################################
```

- Modify the code using accumulators to eliminate loop slowdown and all conditionals. **(20 Points)**
  [Hint – using accumulator **cg_accum_add_32** could be useful.**]**

Submission Instructions

After downloading and extracting the source files to the folder "HW05_source_files":

1) Rename the folder "HW05_source_files" to "HW05_<your last name>", for example "HW05_El-Araby".

2) Complete the code for the following files as you see necessary to implement the required function:
    a. "loop_dependencies_v1/main.c"
    b. "loop_dependencies_v1/ex_loop_dependencies.mc"
    c. "loop_dependencies_v2/main.c"
    d. "loop_dependencies_v2/ex_loop_dependencies.mc"
    e. "loop_dependencies_v3/main.c"
    f. "loop_dependencies_v3/ex_loop_dependencies.mc"
    g. "loop_dependencies_v4/main.c"
    h. "loop_dependencies_v4/ex_loop_dependencies.mc"

3) Run the "./make_clbr" script in all folders to remove the unncessary files such that each folder contains only the modified source code.

4) Your written solution to exercise 1, exercise 2, exercise 3, and exercise 4 should be appended to the folder "HW05_<your last name>".

5) Compress the main folder to "HW05_<your last name>.zip", for example "HW05_El-Araby.zip" and upload it to blackboard before the due date and time.

NOTE: Homework submission is a "Single Attempt", i.e. carefully review everything that you want to submit before hitting the "submit" button and make sure that you have uploaded all documents you want to submit and have not missed anything.