Machine Organization and Assembly Language Programming

Problem Set #5

Due: Wednesday May 26th

In this assignment you are asked to program in the C language a *trace-driven* simulation of the 5-stage pipeline that is in your book. This assignment maybe done in groups of 2.

A trace-driven simulation takes as input a sequence of tuples (instructions, PC) that represents the execution of a given program. It then performs, in our case, all register transfer operations in the pipeline but it does not have to compute results since the sequence of instructions is already determined. The values that are, for example, loaded in a register as a result of a load operation or computed as the result of an arithmetic operation are not needed. What must be taken into account, though, is whether forwarding or stalling occurs. The output of the simulator is the number of cycles spent in the pipeline as well as various statistics that depend on the particular parts of the architecture that are of interest (see below).

The trace that you'll be given as input has only the following opcodes:

- R-R format: add, sll, slt
- Immediate format: addi, ori, lui
- Load-store format: lw
- branch format: bne

Moreover, the PC is not given in the trace because in this small trace all branches are taken. Your implementation for branches should be slightly different than that of the book because of the peculiarities of trace-driven simulation. When a branch is detected in stage ID, the instruction in stage IF should be stalled for 2 cycles: (1) one cycle for the branch to go into EX and determine whether the branch is taken or not, and, since the branch is taken, (2) a new PC is determined and a new instruction is fetched at the next cycle. In a real program, this instruction is at a new PC but in the trace, and in the simulation program, this new instruction is the one following the branch. In the simulation it's already in the IF/ID pipeline register. So stalling 2 cycles after recognition of the opcode for the branch is a faithful representation of the pipeline in this restricted case.

The main loop of the simulator looks like:

```
while the number of instructions to simulate is not reached
  begin
     simulate events in stage WB;
     simulate events in stage MEM;
     simulate events in stage EX;
     simulate events in stage ID;
     simulate events in stage IF;
  end
```

Of course, we have not included counters to count the number of cycles, the number of instructions etc.

A skeleton C-program to read in instructions will be given to you. Hints on how to decode instructions will also be given (i.e., part of "simulate events in stage ID").

The output of your simulation should be:

- How many cycles were simulated when the last instruction simulated exits the pipeline
- For each opcode, the number of instructions with that opcode that entered the pipeline (Including those that did not terminate at the end of the simulation)
- For each time there was forwarding, indicate the stage in which forwarding originated (EX/MEM or MEM/WB) and the instruction that received the forwarded results (you can give a number to each instruction starting with the first instruction having the number 1)
- For each time there was stalling (including branches) indicate the cause of the stalling and the instruction number of the stalled instruction.

The C-skeleton and the input trace will be given to you shortly. In the mean time, you should plan your program and decide on what you have to keep in each of the pipeline registers.

Example: Assume the following trace:

```
          la      $a0,  xyz          lui   $1, 0[xyz]          0x3c01 0000
                                     ori   $4,$1,0[xyz]        0x3424 0000
loop:     lw      $t1, 0($a0)        lw    $9, 0($4)           0x8c89 0000
          add     $t2,$t1,$t1        add   $10,$9,$9           0x0129 5020
          add     $t2,$t2,$t1        add   $10,$10,$9          0x0149 5020
          addi    $t0,$t0,1          addi  $8, $8,1            0x2108 0001
          sll     $t0,$t0,4          sll   $8, $8, 4           0x0008 4100
          bne     $t0,$0, loop       bne   $8, $0, loop        0x1500 fffc
```

In the left column is assembly language code; in the middle pseudo-instructions have been translated and symbolic names (for addresses and registers) have been removed; in the right column is how the trace will look like: a sequence of 32-bit words.

Assume that we want to simulate 4 instructions. You can verify that at the end of the simulation of "add $10,$9,$9", the next 4 instructions will be in various stages of the pipeline.

The type of output you would get for this trace is:

- Cycles simulated: 9
- Distribution of opcodes:
  - lui :1
  - ori :1
  - lw :1
  - add :2

  - addi :1
  - sll :1

  (Note that the "bne $8, $0, loop" has been fetched but not decoded when the simulation terminates, so we don't count its opcode)

- (this might be only a partial output for this example)
  Forwarding from EX/MEM between instructions 2 and 1
  etc...
- (this might be only a partial output for this example)
  Stalls: 1 cycle stall for instruction 4 (load conflict).
  etc.