Homework #5
Due November 19th
20 points

Purpose: The purpose of this homework is to learn about pipelining, and all the particulars that involves.

Task: Your task, in a nutshell, is to take Homework #4 and pipeline it. Your pipeline will be a conventional 5-stage MIPS pipeline. When in doubt, the ultimate resource for your questions is the R2000 processor [except for changes listed in this handout and HW4].

Particulars:

Use SMOK.

Pipeline:

The pipeline is:

Fetch -> Register Read -> Execute -> Memory Access -> Register Writeback

I refer you to Figure 6.40 in your book for a detailed explanation of the pipeline stages.

Branch delay slots:

Note, HW4 has post-incremented addressing for branches. That is the target of a branch is PC + 4 + Immediate. We will continue to have this, but change the MIPS conventions slightly. The instruction slot *after* a branch is called a "delay slot". The delay slot comes about because that instruction will be fetched and in the pipeline before you are able to even know that you have just fetched a branch. We are going to follow the conventions of *always* executing the instruction in that delay slot. Different processors handle this differently. Some will execute it if the branch is taken, but then "squash" the instruction if the branch is not taken (squashing means to remove it from the pipeline). However, as I said, to make this a little easier we will always execute it.

So, your machine, when it encounters an control instruction (jump, jal, branching, etc) it will always execute 1 additional instruction past the control instruction. However, it will only execute just one. Since the branch will not be resolved until later in your pipeline your pipeline will have "bubbles" in it when a branch is encountered. These bubbles come about because you should stop fetching once you've fetched this 1 additional instruction past a branch and wait for the branch target to be resolved and start fetching from there.

Load Delays:

Accessing memory will be an issue. Your processor should support interlocks that detect instruction sequences like:

```
LW      $1, 0($2)
ADD     $3, $1, $4
```

It should STALL the processor on the ADD for 1 cycle. The result of a LOAD is not valid until the following instruction. I.e.:

```
LW      $1, 0($2)
ADD     $0, $0, $0                  <---- No-operations (NOP)
ADD     $3, $1, $4
```

Other Hazards:

You should put in other appropriate logic such that no other hazards in the machine are a problem. I.e., consider this sequence of instructions:

```
ADD     $1, $2, $3
ADD     $2, $1, $4
```

You will need a forwarding network to handle this. See Figure 6.40 from your book.

Memory layout

Use the same memory layout as HW4.

Instructions:

Support the following instructions:

Arithmetic:
- ADDU, ADDIU
- SUBU
- SLTIU, SLTU
- ORI, OR
- SRL, SLL <--------- these are new for HW5

Load/Store:
- LW, SW

Branch:
- JAL, JR
- BEQ, BNE

Startup code:

Modify the startup code from HW4 to take into account the delay slots. Either stick a NOP in there or do something clever (like move an instruction down there that is useful).

Normal code:

Write a new procedure in MIPS assembly and hand compile it down to the instructions you implemented for HW4/5. This procedure is called countbits(int x). What this procedure does is it counts the number of 1 bits in a word. In C this procedure looks like:

```
int
countbits(
        int     x)
        {
        int     v;

        v = 0;
        while( x != 0 )
                {
                if ( (x & 1) != 0 )
                        v = v + 1;
                x = x >> 1;
                }
        return ( v );
        }
```

Try and be "optimal" about this and not use the result of a load until 2 cycles later; try and fill the delay slot of control-flow instructions with meaningful instructions, not just no-ops.

EXTRA CREDIT:

YOU CAN ONLY GET CREDIT FOR THESE IF YOUR BASIC PIPELINE ABOVE WORKS. YOU CAN ASK THE TA'S QUESTIONS ABOUT THESE ONCE YOUR BASIC HW5 WORKS.

(2 points) Add a branch predictor. This will be a simple predictor that simply predicts the last address of the branch. This is nominally called a "branch target buffer" (BTB). What you do is take the address of the control instruction in the 2nd pipeline stage and feed the low bits of the address to a table (use a memory of 16 entries). From this table start fetching in the next cycle from the outcome of this table. When you resolve the actual branch target write this target into the table location of the branch address. If you predicted the branch target correctly, then you can keep fetching and executing. HOWEVER, if you predicted incorrectly you will need to squash the instructions that where incorrectly fetched (but not the delay slot instruction!).

(2 points) Make a wide-issue LIW (long instruction word) machine. This machine has a different instruction set than MIPS. Here it is:

Each instruction is 64 bits. This instruction has two instruction bundles in it:

[instruction 0][instruction 1]

Each bundle looks like a conventional MIPS instruction EXCEPT instruction 0 must draw its SOURCE operands from registers 0 - 15 and its destination operand MUST be a register between register 16 - 31. On the other hand instruction 1 must draw its source from registers 16-31 and its destination must be in 0 - 15.

MAKE REGISTER 0 AND 16 HARDWIRED TO ZERO

LOAD/STORE instructions must go in bundle 0. (Hence loads will address from registers 0-15, but write to a register in 16-31, and STORE's will draw their operands from registers 0 - 15).

CONTROL-FLOW instructions must go in bundle 1

To do this extra credit, split the register file apart into 2 register files of 16 registers each. The strange destination/source restrictions outlined above are to get around SMOK issues, but partitioned register files are a reality in computer architecture so its not *that* unrealistic.

Rewrite your code and the startup code in this new LIW format.