Texas A&M University
Computer Science 434
Compiler Design

Assignment Six

David Cox

November 14, 2015

Brief notes

Rather than a Readme, I thought it would be nice to submit a short, informal report. During the course of this lab, I reasoned that using the FPU would result in a performance improvement that was worth the time investment. Unfortunately, the instructions required to move data to and from the FPU increase the instruction count enough to negate any benefit due to the reduced number of loads / stores. Additionally, while using the FPU was fairly simple, I had to be very careful with how I kept track of my registers, which increased the complexity of the assignment quite a bit. I decided to back out of this last minute and ended up with a result that I'm happy with. Had the goal of the assignment been to reduce computation time rather than instruction count alone, I would have stuck with my original design as transferring data to and from the coprocessor is significantly faster than to / from memory.

Register spilling is performed once there are no more clean registers (and at jumps, calls, etc.). The register spilled in the event of a shortage is selected at random. To prevent registers from being spilled immediately after they were filled and before their value was used (huge problem if that happens), I added a mutex to each register in `regs` that could be set in the `Emit` functions. See the example below.

```
void Mips::EmitBinaryOp(BinaryOp::OpCode code, Location *dst,
                        Location *op1, Location *op2)
{
    rs = (Register) regs_pickRegForVar_T(op1, false);
    regs[rs].mutexLocked = true;
    FillRegister(op1,rs);
    RD_insert(op1,rs);

    rt = (Register) regs_pickRegForVar_T(op2, false);
    regs[rt].mutexLocked = true;
    FillRegister(op2,rt);
    RD_insert(op2,rt);

    rd = (Register) regs_pickRegForVar_T(dst, false);
    regs[rd].mutexLocked = true;

    Emit("%s %s, %s, %s\t", NameForTac(code), regs[rd].name,
         regs[rs].name, regs[rt].name);
    RD_insert(dst, rd);

    regs[rd].canDiscard = true;
    regs[rs].mutexLocked = false;
    regs[rt].mutexLocked = false;
    regs[rd].mutexLocked = false;
}
```

My last noteworthy optimization was the discarding of (some) binary expression results. With my FPU implementation, I was hoping to do something really fancy like waiting as long as possible to compute a value, then performing all of the binary operations sequentially, taking advantage of the FPU's compound multiply / add instructions. (The MIPS FPU allows for two registers to be multiplied then added to a third in one instruction.[1])

Having to abandon the FPU due to the increased instruction count was very disappointing. Fortunately, by discarding some intermediate values I was able to reduce some of the extra loads / stores associated with not having any FPU registers.

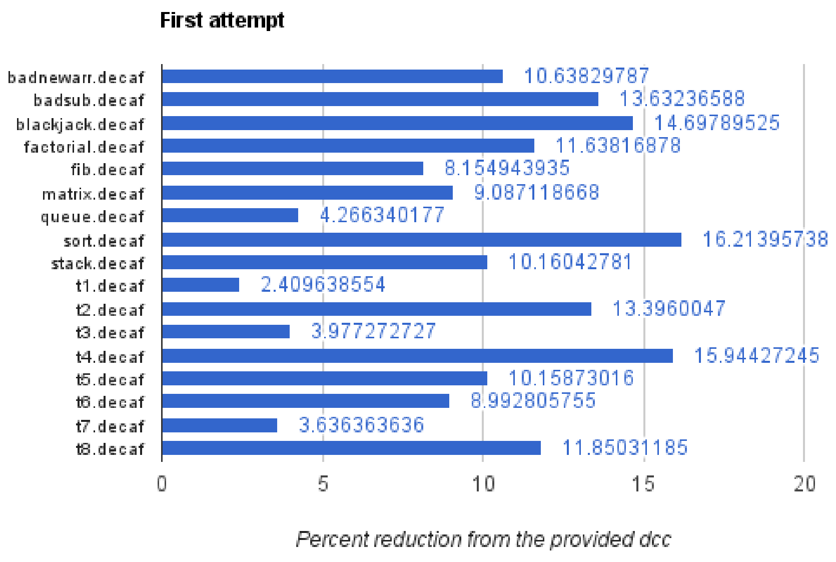
Below is a sample of output before dropping intermediates.

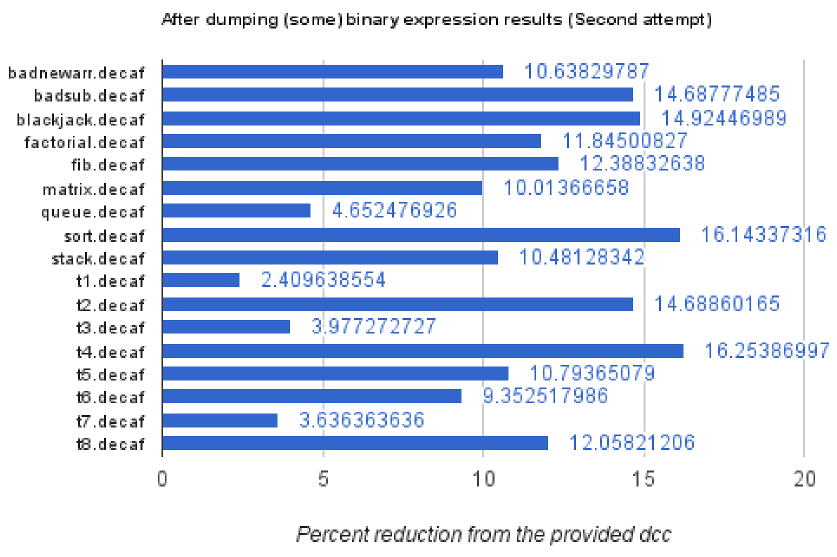
```
    # c = _tmp39
      move $t5, $t4    # move (copy) _tmp39 from $t4 to c in $t5
    # _tmp40 = c + one
      sw $t6, -168($fp) # spill _tmp38 from $t6 to $fp-168
      add $t6, $t5, $t1
    # c = _tmp40
      move $t5, $t6    # move (copy) _tmp40 from $t6 to c in $t5
```

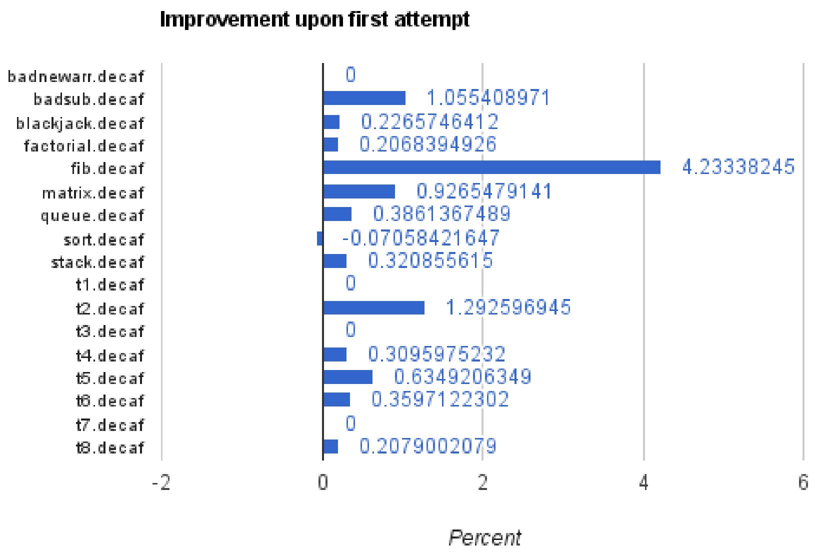
And after:

```
    # c = _tmp39
      move $t4, $t5    # move (copy) _tmp39 from $t5 to c in $t4
    # _tmp40 = c + one
      add $t5, $t4, $t1
    # c = _tmp40
      move $t4, $t5    # move (copy) _tmp40 from $t5 to c in $t4
```

To better visualize the performance benefit of this optimization, I plotted the change in percent reduction of instructions relative to the provided solution between my implementations.







As seen in the third chart, there's a small performance improvement in most cases, making the optimization worth keeping. The largest performance improvement appears in the fibonacci test case. This is due to the significant number of repeated additions on a single variable. Test case T2 also involved repeated additions and saw a fair improvement.

Lastly, I'd like to mention that the registers `s0-s7` were never used. I attempted to implement the register saving but it broke several test cases and I ran out of time / willpower to fix it. I anticipate that a significant reduction in loads / stores could be achieved this way. I also tried using (abusing) the `s0-s7` registers as extra temporary registers, but this was useless because they all had to be spilled before jumps and calls, negating any performance benefit.

[1] http://math-atlas.sourceforge.net/devel/assembly/mips-iv.pdf