

Name:

CMSC 411 Computer Architecture

Worksheet 2
Answers
Fall, 2002
Dr. Michelle Hugue
October 8, 2002

Problem 1: Hands-on MIPS Code

Please express the following register expressions as MIPS code fragments, using as few instructions as possible. The registers in each expression below have been loaded with the correct data, and no other integer registers are in use.

1. $R1 \leftarrow 164000$

Answer: What's the biggest 16 bit unsigned binary number? Give you a hint: it's all ones, or $2^{16} - 1 = 65535$

```
DADDUI R1, R0, #65535; 164000 won't fit in 16 bits!  
DADDU R1, R1, R1 R1 has 131070  
DADDUI R1, R1, #14930; R1 now contains 164000
```

Note that the DADDU and DADDUI are unsigned additions. The DADDUI is used to force the string of 16 ones to be zero extended, because its an unsigned value. Without the U, the string of 16 ones would be treated as the 2's complement value -1, and sign extended to a big, 64 bit negative one, which is NOT what we want.

The reason that DADDU is used has to do with exceptions... things that make pipelining hard to implement. In some machines, the DADDU will not cause condition codes to be set, where as the DADD will. However, having the middle instruction as DADDU or DADD doesn't change the answer. But, if the DADDUI's aren't used, the maximum immediate value will be a zero in the sign bit, and all the rest ones, giving 32767 as the decimal value.

2. $R2 \leftarrow 2(-15 + R3 - R2 + R5)$

Answer: Tedious, but you get there eventually.

```
DSUB R4, R5, R2; Computes R5-R2  
DADDI R6, R3, # -15 R3-15  
DADD R2, R4, R6 R2 contains half of its final value  
DADD R2, R2, R2 R2 final
```

Another way to do the last step would be to shift R2 one bit to the left (towards the sign bit, and fill with zero), because that's the same as multiplying by two.

While the above is acceptable, it might be better to use as few extra registers as possible. (Why or Why not?)

```
DSUB R2, R5, R2; Computes R5-R2  
DADD R2, R2, R3 new R2 plus R3  
DADDI R2, R2, # -15 subtract 15  
DADD R2, R2, R2 R2 final
```

This is an example of smart "compiling" exploiting the commutativity of addition, and the fact that adding two integers is lots faster than multiplying two integers in MIPS. (Assuming that data hazards aren't an issue, and in modern machines, for integer adds, they aren't.)

Why? because MIPS uses the floating point multiply hardware for integer multiplies, and the floating point divide hardware for integer divides. In general, to multiply two integers together, you must first

move them BOTH into floating point registers; use the integer multiply operator and store the result in a floating point register; and then, move the result back to a general purpose/integer register to use that value in other integer operations. Yuck.

3. $R3 \leftarrow R3' + 1$

Answer: FYI, it took me a few years for this one to sink in.

DSUB R3,R0, R3; Why? Two's complement math, of course.

4. $R4 \leftarrow (R5 \oplus R2)'$

Answer: The "XOR" part is no problem—a standard binary operator.

However, did you notice, there's no "NOT" operator, much less a "NOR" or and "XNOR"? You have to make yourself a word of all ones, and then "XOR" it with the word you want to complement. I once did it by assuming that the immediate operand, negative one (-1), below, would be sign extended.

```
XOR R4, R5, R2;  Vanilla XOR
XORI R4,R4, #-1; Does this sign extend?
                    If so, we're done.
```

Unfortunately, according to MIPS on-line resources (see MIPS.com) the logic functions zero-extend, regardless of the sign. That is, the previous instruction would be taking the exclusive-or of R4 with a word having 16 ones in bits 63-48, where bit 63 is the LSB bit. The remaining 48 bits would be zero. Considering how useful it is to have a quick way of getting a word of all ones, I assumed that XORI would sign extend. But, Noooooo... I can't find this tidbit in the book, any more than I can find which argument of the subtract functions is the one that is being subtracted from the other. But, apparently not. This is an apparently preferred way:

```
XOR R4, R5, R2;  Vanilla XOR
DADDI R8,R0, #-1; A 64 bit register with all 1's
XOR R4,R4, R8;  complement all bits of R4.
```

5. if (R2== 0) then $R4 \leftarrow R4 - 24 + R2$
 else $R4 \leftarrow 24 - R4 - R2$

Answer:

Because of the difficulties associated with control hazards (which occur when the PC must be modified to assure that the correct next instruction is fetched), it's important to minimize the number of branches where ever possible. So, standard practice is to preset the value of the output register, R4, then test, and then reset the value of R4 as needed.

First, notice that if R2 is zero then the final value of R4 is the negative (or two's complement) of the final value of R4 when R2 is not zero. So, all we have to do is compute one of the two potential R4 values, and negate if the test goes the other way. (Read the code, and then read this sentence again.)

```
DADDI R4, R4, #-24; R4 has (R4 - 24)
DADD R4, R4, R2; final value of R4 if R2 is 0.
BEQZ R2, done; if R2= 0, we are done
DSUB R4, R0, R4; handle R2 non-zero
done: (next command)
```

That is, we have only one control hazard that is associated with the single conditional branch.

Problem 2: Hands OFF the ISA questions

Briefly answer the following questions, explaining your answer for full credit, of course.

2.1 True or False: The ISA uniquely determines the specifications of the hardware on which it is to be executed.

Nope. Can run same ISA on different platforms.

2.2 Why is the encoding of the ISA so important in designing a computer architecture? That is, speculate as to why so much of chapter 2 is spent justifying the decisions made in designing MIPS.

See cheat sheet. The ISA influences both CPI and IC. That just might have an effect on the execution time.

2.3 Why is byte alignment an issue? That is, why might an ISA forbid one to load a word from a specific address?

Typically because the memory is only wired for aligned accesses. In simple terms, it costs more money (and time delays) to support more than the byte, halfword, word, and doubleword used in the MIPS64. As stated in chapter 2, words are at addresses which are $0 \bmod 4$, while aligned doubleword addresses are $0 \bmod 8$.

For example, in each pair of MIPS instructions below, the first instruction results in a byte alignment error, while the second doesn't. To see this, compute the *effective address* for each operation—that is, assuming you know what one is.

Also, why might allowing the first instruction below cause problems?

Again, time. You'd have to fetch two aligned words to get all the pieces of the misaligned word.

And, are there any load/store instructions that are immune from byte alignment errors in MIPS?

Hint: What size entities have addresses that are $0 \bmod 1$ in a byte addressable memory?

```
LW R2, 7(R0) ; 7 is not 0 mod 4
LW R4, 128(R0) ; 128 is 0 mod 4
```

```
DADDI R8, R0, #99 ;
SD R3, 17(R8) ; 116 is not 0 mod 8
SD R5, 29(R8) ; 128 is still 0 mod 8
; not dest, source format (!)
```

```
DADDI R9, R0, #1050;
L.D F2,-1 (R9) ; 1049 is not 0 mod 8
L.D F4,-26(R9) ; 1024 is 0 mod 8
S.D F11, 998(R9) ; 2048 is 0 mod 8
; Okay for MIPS 64
; F11 is NOT a DP FPR pair for MIPS 32
```

Problem 3: Hands ON the Pipe-Like Pipe

3.1 What do I mean by “pipe-like pipe”? That is, why do I constantly refer to it that way, instead of merely “the pipeline”?

Think of a hose. Water goes in one end and is supposed to come out the other, with no early exit. Even though the ALU instructions do nothing in the MEM stage, they have to occupy that stage for at least one clock cycle, even in the absence of stalls.

Similarly, Store instructions do nothing in the WB stage. However, they still occupy it because the next instruction in the pipe is forced to spend a similar clock cycle in the MEM stage right behind it. That is, unless the operation of the MEM stage is modified too, there’s no point to ending some instructions earlier than others if no instruction can take advantage of the extra cycle.

3.2-4 Note: solutions to the remaining problems are in .doc and html format elsewhere.