LSU EE 4720 **Homework 1** Solution **Due: 2 March 2007**

Problem 1: Without looking at the solution solve Spring 2002 Homework 2 Problem 2 parts a-c. Then, look at the solution and assign yourself a grade in the range [0,1].

Problem 2: If the value in register `r2` is not aligned (a multiple of four) the `lw` in the MIPS code below will not complete.

```
lw r1, 0(r2)
```

(*a*) Re-write the code so that `r1` is loaded with the word at the address in `r2`, whether or not it is aligned. For this part do not use instructions `lwl` and `lwr` (see the next part).

Solution shown below.

Grading Note: No one submitted a solution like the one below, that is, one that (correctly) combined data from just two `lw` *instructions. Most submitted solutions used four* `lb` *instructions, some of these used shift and OR instructions to insert each loaded byte into what would be the full word, one solution followed each* `lb` *by a* `sb`*, the* `sb` *instructions were relative to an aligned address.*

```
# Solution shown below.
# Comments show register contents for this example:
#
# r2 = 0x1001
# Memory contents:
#  Mem[0x1000] = x00, Mem[0x1001] = x11, Mem[0x1002] = x22, .. Mem[0x1007] = x77
# Therefore, want r1 = 0x11223344.
#
andi r4, r2, 3  # Extract "misalignment", m.     r4 = 1 (call this m)
sub r5, r2, r4  # Round down to aligned address. r5 = 0x1000
lw r6, 0(r5)    # Load first part.               r6 = 0x00112233
                #                                This has 4-m = 3 bytes we need.
lw r7, 4(r5)    # Load second part.              r7 = 0x44556677
                #                                This has other (1) byte we need.
sll r4, r4, 3   # Shift amt for 1st part.        r4 = 8*m = 8.
sll r6, r6, r4  # Shift 1st part into place.     r6 = 0x11223300
addi r8, r0, 32 # Constant for 2nd shift amt.
sub r8, r8, r4  # Shift amt for 2nd part.        r8 = 32 - 8m = 24.
srl r7, r7, r8  # Shift 2nd part into place.     r7 = 0x00000044
or r1, r7, r6   # Finally, combine them.         Voila! r1 = 0x11223344
```

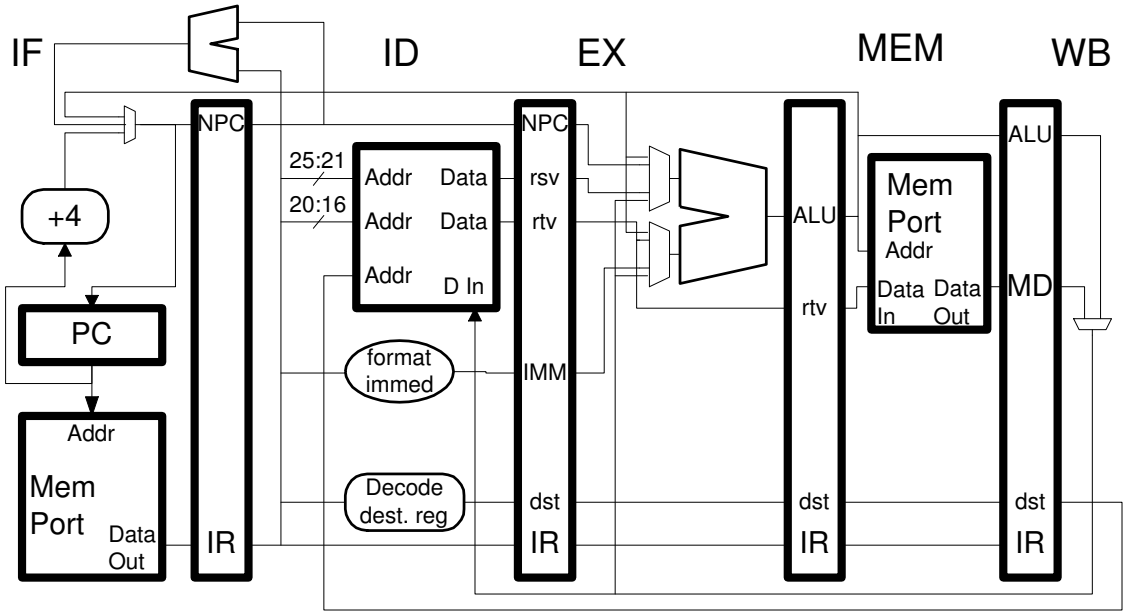
(*b*) Re-write the code, but this time use MIPS instructions `lwl` and `lwr`. *Hint: These instructions were not covered in class, try looking them up in the MIPS architecture manual conveniently linked to the* `http://www.ece.lsu.edu/ee4720/reference.html` *page..*

The solution is shown below.

Grading Note: About the only wrong solution was my own (the one below is correct), in which the `lwr` *used the same offset as the* `lwl`*. I hope at least some got the answer correct because they carefully read the* `lwr` *description and saw that the effective address for this instruction is of the least significant byte of the word, not the most significant byte required of* `lwl`*,* `lw`*,* `lh`*, and* `lhu`*.*

```
# Solution:
#
lwl r1, 0(r2)
lwr r1, 3(r2)
```

Problem 3: Consider how the `lwl` and `lwr` instructions might be added to the implementation below. There are two pieces of hardware that with minor modification would be able to merge the sub-words in a reasonable solution. Alternatively, a new piece of hardware to perform the merge can be added.



(*a*) Show how the hardware can be modified.

- If your solution relies on an existing component to perform the merge indicate which component and why that can be easily modified to do the merge.
- As with other MIPS instructions `lwl` must spend one cycle in each stage (except when stalling).

An implementation must merge some bytes of the value loaded from memory with the existing register contents. Here are three reasonable solutions:

Merge in memory port: For an ordinary load instruction the data-in to the memory port in the ME stage has the `rt` register value, which is ignored. Since the memory port "automatically" gets the old value and it retrieves the new value, have it merge the two together. Some of the hardware needed to do that, namely hardware to shift the loaded value into the correct place, is already there. (More on this later in the semester.) Making whatever changes are necessary to do the merge might not add to the critical path because most of the hardware is already there.

Merge in the WB stage using new hardware: The new hardware would have as inputs the `WB.MD` pipeline latch, a new `WB.rtv` pipeline latch, the low two bits of `WB.alu` (for alignment), and control bits. The output would connect to a new input of the WB mux. Because of the time needed for the merge with this alternative it might not be possible to bypass to EX from WB.

Merge within the register file: A slightly modified memory port shifts the loaded value into the correct position (it already does that for `lb` and `lh`) and the loaded value reaches the register file following the existing data path. A modified register file will write only those bytes of the register which are to change, the others will remain unchanged. New control logic (perhaps placed in the WB stage) will examine address bits to see which bytes of the register to write.

Here is a solution that would work but would be too slow:

Merge in the `ME` *stage (not the mem port):* New hardware merges the output of the memory port and `ME.rtv`. (The hardware would otherwise be the same as the merge-in-the-WB-stage option above.) This would be too slow because memory is considered to be on the critical path and anything delaying a signal between, say, the memory port out and the pipeline latch would be lengthening the critical path.

(*b*) Show how the code below would execute on your solution. **Pay attention to dependencies.** Feel free to propose an alternate solution to reduce the number of stalls.

```
# Execution for merge-in-memory-port solution.
# Cycle         0  1  2  3  4  5  6  7  8  9
add r1, r3, r4  IF ID EX ME WB
lwl r1, 0(r2)      IF ID ----> EX ME WB
sub r5, r1, r6        IF ----> ID -> EX ME WB
```

Problem 4: Consider these options for handling unaligned loads in the MIPS ISA which might have been debated while MIPS was being developed.

- *Option Lean*: All load addresses aligned. No special instructions for unaligned loads (*e.g.*, no `lwl` or `lwr`).

  Pro: Simple implementation. Con: Slower handling of unaligned loads.

- *Option Real*: All load addresses aligned. Special instructions for unaligned loads (*e.g.*, `lwl` or `lwr`).

  Pro: Quick handling of unaligned loads. Con: Small amount of additional data path complexity.

- *Option Nice*: Load addresses do not have to be aligned, however warn programmers that loads of unaligned addresses may take longer in some implementations.

  Pro: Simpler and shorter code. Con: Larger amount of additional datapath complexity. Slower code if implementations do take longer.

(*a*) For each option provide an advantage and a disadvantage.

See above.

(*b*) What kind of data would be needed to choose between these options? Consider both software and hardware data, be reasonably specific.

One needs to weigh the cost against the benefit for a typical implementation. The cost is the engineering effort and chip area needed for each option, the benefit is how much faster code will run.

For the cost one needs an estimate of how large and how complex the additional hardware would be.

For the benefit one would need to know how often code performs unaligned access. That data could be obtained by finding an existing ISA that imposes alignment restrictions and then analyzing benchmarks compiled for that ISA to determine how frequently they perform unaligned accesses. Then estimate the relative performance of the three options.

Evaluating the nice option requires another piece of data: how many programmers would avoid unaligned access because it might be slower. Gentle reader, I'm sure you would—or if you didn't you'd have a good reason, but what about the *typical* programmer? If programmers made no effort to avoid unaligned accesses then their code would be slower on some implementations. Fortunately, most machine language is "written" by compilers and compiler writers would likely pay attention to the warning, so for this analysis assume in nice option unaligned access is avoided as it should be.

Grading Note: Many answers discussed the conditions under which unaligned accesses would be made, rather than just suggesting measuring how much existing programs made unaligned access. Many solutions correctly noted that unaligned accesses are needed when data is to be packed tightly (avoiding the padding needed to meet alignment requirements). Nevertheless its much better to directly measure how often something is done (when possible) then to estimate how often it would be done. In this case because one would still have to gauge how often misalignment was actually necessary to save space (this would happen with arrays of structures of certain members (say, one integer and one character). One also might not account for unanticipated reasons for unaligned access.

(*c*) Using made up data pick the best option. Any choice would be correct with the right data.

Made up cost data: Cost of lean option, 0; cost of real option, 10 person-weeks + 1000 gates; cost of nice option, 20 person-weeks + 1200 gates.

Best proposed feature (nothing to do with unaligned loads): 5% improvement using 10 person-weeks and 1000 gates.

Made up benchmark data: Over all benchmarks, 20% of instructions are loads. Of these, 0.0001% are unaligned. With lean option each unaligned load would take about 5 times as many instructions as real option but that would still have negligible impact on performance, so choose lean.

Different made up benchmark data: Over all benchmarks, 20% of instructions are loads. Of these, 10% are unaligned. Assuming instruction count is proportional to performance for lean and real Lean, $0.8+0.18+0.02*10 = 1.18$; real, $0.8+0.18+0.02*2 = 1.02$, so the performance improvement of real or lean would 13.6% faster, well over the 5% needed to justify the cost (compared to the best proposed feature). Assuming no penalty for unaligned access, the nice option would yield about a further 2% improvement, which would not be worth the cost.

Grading Note: Many solutions provided made-up data describing only a small code fragment. The idea was to pick the kind of data a CPU manufacturer would use to actually decide which option to pursue, and so it would have to be based on real benchmarks or something equally convincing.