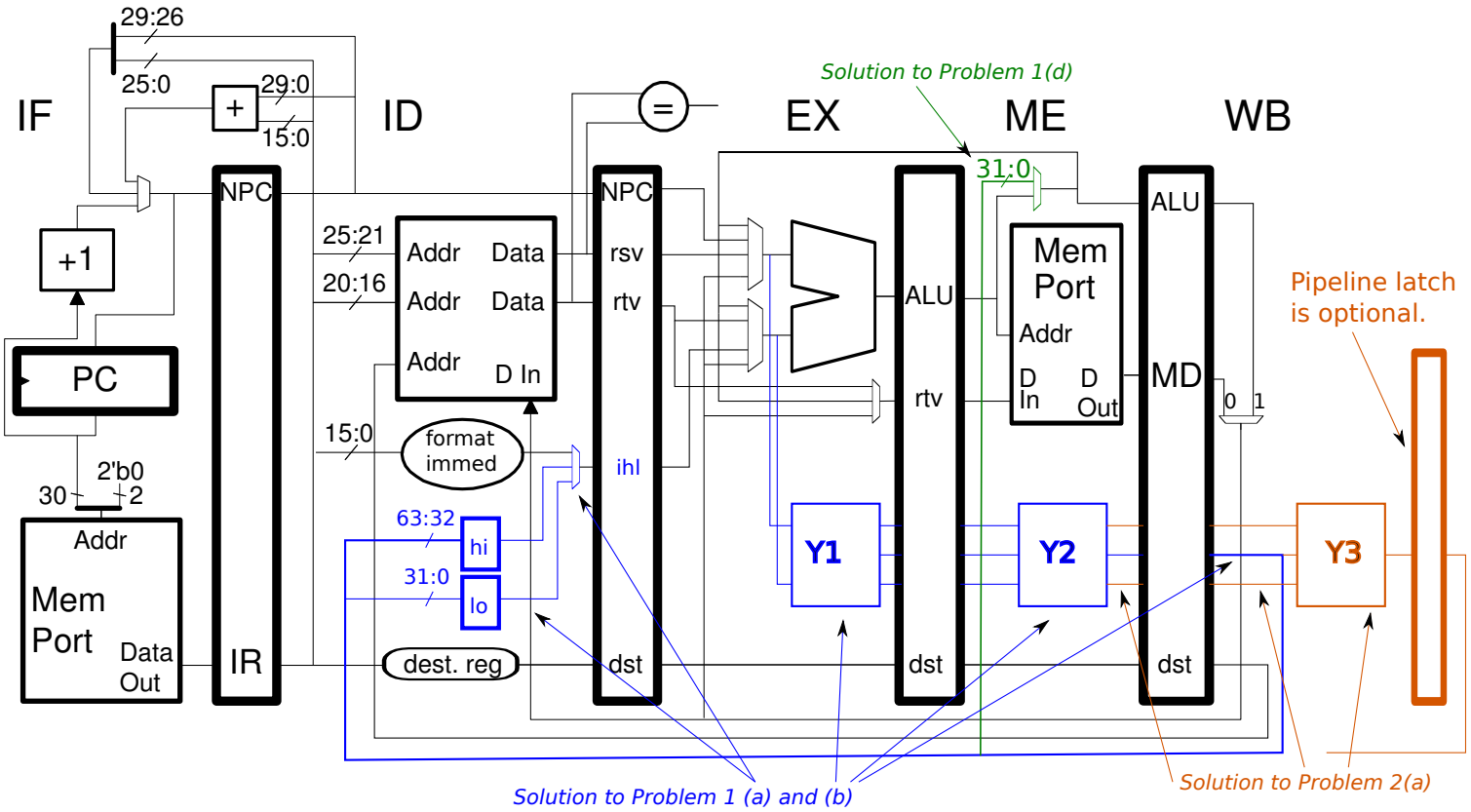
LSU EE 4720 **Homework 4** Solution **Due: 8 March 2013**

Problem 1: Recall that the MIPS-I `mult` instruction reads two integer registers and writes the product into registers `hi` and `lo`. To use the product the values of `lo` and `hi` (if needed) have to be moved to integer registers, done using a move from instruction such as `mflo`. In this problem these instructions will be added to the implementation below.



Consider an integer multiply unit that consists of two stages, `Y1` and `Y2`. The inputs to `Y1` are the 32-bit multiplier and multiplicand, and the output of `Y2` is the 64-bit product. Unit `Y1` has three 32-bit outputs named `s0`, `s1`, and `s2`; unit `Y2` has 3 32-bit inputs of the same name. As one would guess, the data from the `s0` output of `Y1` should be sent to the `s0` input to `Y2`, likewise for `s1` and `s2`.

As with other functional units, such as the ALU, inputs to `Y1` and `Y2` must be stable near the beginning of the clock cycle and the outputs must be stable near the end of the clock cycle. There is enough time to put a multiplexer before the inputs, or after the outputs (but not both).

Solve the two parts below together. That is, the hardware for part (a) might take advantage of the hardware for part (b) and *vice versa***.**

(*a*) Add the datapath hardware needed to implement the `mtlo`, `mthi`, `mflo`, and `mfhi` instructions. Both the ALU and the integer multiply unit have an operation to pass either input to its output unchanged. That is, let x denote the ALU output and let a and b its inputs. In addition to operations like $x = a + b$ and $x = a\&b$, the ALU can also perform a pass-a operation, that is, $x = a$ and a pass-b operation, $x = b$. The integer multiply unit also has pass-a and pass-b operations.

- Put the `hi` and `lo` registers in the `ID` stage.

- Do not write the `hi` and `lo` registers earlier than the `ME` stage.
- As always, cost is a criteria.
- Bypass paths will be added in the parts below.

Solution appears in blue above. The `mflo` and `mfhi` instructions, using a new `ID`-stage multiplexor, route the `hi` or `lo` value to the existing (though renamed) `ID/EX.ihl` pipeline latch, where it can easily take a path through the ALU (using a pass b operation) and continue on to write back the integer register file.

The `mtlo` and `mthi` use the ordinary multiply unit inputs (see the next problem), but the multiply unit uses a pass a (since the register is in the rs field). The multiply unit would need to have two versions of pass a, once to pass to the lower 32 bits of its output, and one to pass to the upper 32 bits. The control logic would also have to enable the appropriate register (`lo` or `hi`).

(*b*) Add the datapath hardware needed to implement the `mult` instruction. That is, put the `Y1` and `Y2` units in the appropriate stages, and connect them to the appropriate pipeline latch registers (adding new ones where necessary).

- Don't add new bypass paths, but take advantage of what is available.

Solution appears in blue, where `Y1` is placed in `EX` and `Y2` in `ME`. Notice that the multiply unit takes advantage of the multiplexors at the ALU's inputs. Also notice that the writeback occurs in the `WB` stage, but that the outputs connect directly to the `hi` and `lo` registers.

Because the output of the multiply unit is written to a fixed register pair the data can arrive at those registers, `hi` and `lo`, close to the end of the cycle. This is different to the writeback of the integer (general-purpose) register file, where one of 31 registers might be written and bypassing might also be performed and so more time is needed. For that reason, it might be possible to write back `hi` and `lo` in the `ME` stage, and such an answer did not loose points.

(*c*) Show the execution of the code below on your hardware so far. That is, your hardware should not have any new bypass paths, but existing bypass paths in the implementation can be used.

Solution appears below. Notice that there is a dependence between `sub` and `mult`, but that it can be handled by the bypass paths shared with the ALU and so the `mult` instruction does not stall. There is also a dependence between the `mult` and the `mflo`. Since there are no bypass paths for that, the `mflo` must stall two cycles.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9
sub r2, r6, r7   IF ID EX ME WB
mult r1, r2         IF ID Y1 Y2 WB
mflo r3                IF ID ----> EX ME WB
add r4, r3, r5            IF ----> ID EX ME WB
```

(*d*) Add bypass paths so that the code below (which is the same as in the previous part) can execute without a stall. Assume that an additional multiplexer delay is tolerable.

A bypass path has been added from `WB` to `ME`, the appears in green in the diagram. Notice that this bypass path leads to another bypass path, and so the `add` instruction receives the correct value of `r4`.

Grading Note: Some solutions had a bypass from `WB` *to* `EX`*. Such a bypass would be from the* `mult` *instruction to the* `add`*, which are not directly dependent. To be completely correct such solutions would have to explain how the control logic can detect such a bypassing opportunity.*

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9
sub r2, r6, r7   IF ID EX ME WB
mult r1, r2         IF ID Y1 Y2 WB
mflo r3                IF ID EX ME WB
add r4, r3, r5            IF ID EX ME WB
```

Problem 2: Continue to consider the implementation of the MIPS-I `mult` instruction. If MIPS designers thought that an integer multiply unit could be built with two stages they might not have used special registers, `hi` and `lo`, for the product.

(*a*) Show how the pipeline would look if the multiply unit had three stages, `Y1`, `Y2`, and `Y3`. There is no need to add bypass paths for this part.

Solution appears in orange where a `Y3` stage has been added in `WB` and a fifth pipeline latch has been added. Because the multiply unit writes a fixed register (as opposed to the register file) the timing constraints for the writes are less severe and so the added pipeline latch might not be necessary. If would be necessary if the physical distance between the multiplier output and the `ID` stage were large.

(*b*) Explain why there is much less of a need for the `hi` and `lo` registers with a two-stage multiply unit (the first problem) than with a three-stage unit (this problem).

One reason for having special `hi` and `lo` registers is to avoid the structural hazard during writeback. Consider the example below for the three-stage multiply unit, in which `WY` indicates the stage in which `mult` writes back. Both the `mult` and `add` instruction writeback in cycle 5. But because `mult` is writing back into its special registers there is no conflict. If `mult` wrote to the general-purpose registers there would have to be two write ports on the GPR file, which would be more costly than having the two special registers.

With a two-stage multiply unit, the multiply instruction can write back at the same time as other instructions, so there would not be a need for a second write port. There is still the problem of the multiply writing back 64 bits. The expensive solution is to widen the write port to 64 bits (and perhaps write a pair of registers, as is done for floating point). Another possibility is to have just one special register, for the high 32 bits.

```
# SOLUTION: Code execution for the three-stage multiply unit.
# Cycle          0  1  2  3  4  5
mult r1, r2      IF ID EX ME Y3 WY
add  r4, r5, r6     IF ID EX ME WB
```