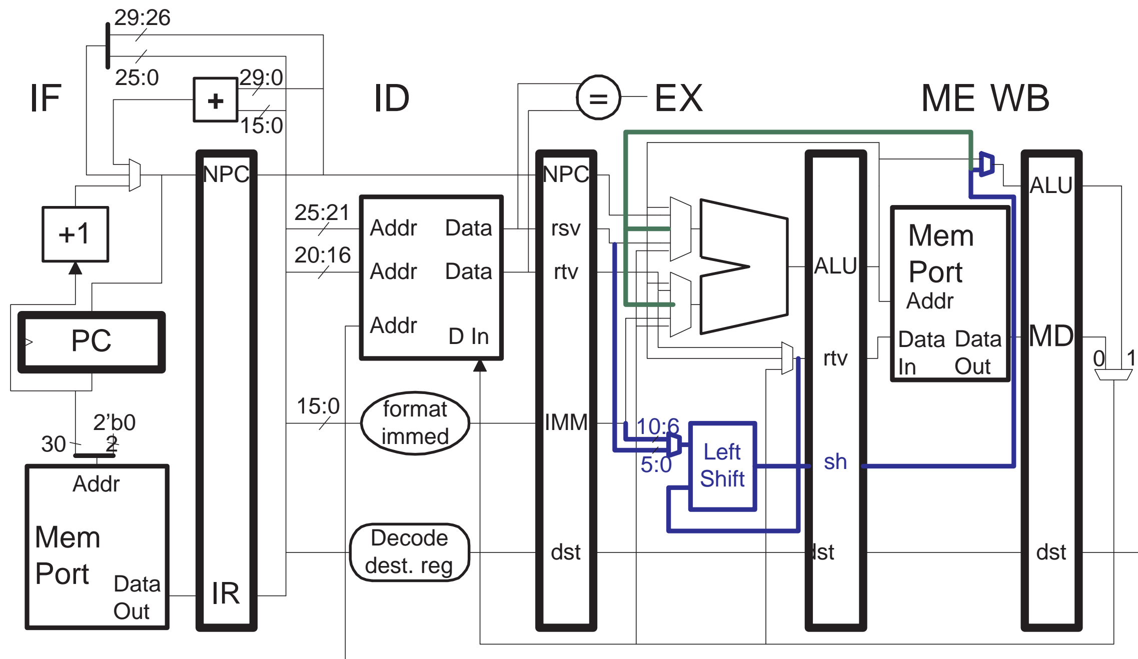
LSU EE 4720 **Homework 2** Solution **Due: 15 October 2008**

Problem 1: The hardware needed to implement shift instructions, such as `sll`, is not shown in the implementation below. (The ALU in the implementation below does not perform shift operations.) Add a separate shift unit to the implementation to implement the MIPS `sll` and `sllv` instructions. The shift unit has a shift amount input and an input for the value to be shifted.

- Show exactly where the shift-amount bits come from (including bit positions).
- Add bypass paths so that the code below can execute without a stall.
- The primary goal is to not slow the clock frequency, the secondary goal is to minimize added cost. This might affect where multiplexers are placed.

```
sll r1, r2, 3
add r3, r1, r4
```



The added hardware for the shift appears in blue and the needed bypass path appears in green.

The shifter would stall for any close dependencies on the shift amount, but such dependencies do not appear in the sample code.

A low-cost solution would have the ALU and shifter outputs go to a EX-stage mux, this would eliminate the need for any added logic beyond the EX stage. But since performance was the primary goal and the ALU output was likely on the critical path such a mux could not be added. Instead, the mux is placed in the ME stage.

The new ME-to-EX bypass path adds cost. If performance were not the primary goal that added bypass could be avoided by moving the existing bypass connection to the mux output. But that would add to the critical path in precisely the same way the EX-stage mux discussed above would.

Common Mistakes:

Many solutions had an unnecessary "format sa" block.

Many solutions passed an `sa` value through the ID/EX pipeline latch even though the sa bits were part of the `IMM` value.

Some solutions passed the shifted value through the ME/WB latch even though such latch bits could be avoided by using an ME-stage mux.

Problem 2: *To answer this question see the SPARC Joint Programming Specification, a description of the SPARC V9 ISA, linked to the course references page.* The SPARC V9 ISA is naturally big endian. Since many programs must read data using little-endian byte order, for example when reading a binary data file that was produced on a little-endian system, the programs need some way to get the data into big-endian order. If loading little-endian data were only a small part of what a program did then it could get by with some combination of ordinary instructions to convert the data to big-endian format. For programs spending substantial time reading little-endian data even a 9-instruction sequence may take too long.

The first instruction below is an ordinary load in SPARC V9, a 64-bit ISA (in which addresses and registers are 64 bits). The second instruction, `ldxle`, is made up; it's a load that assumes data is in little-endian byte order. The last instructions is a real SPARC instruction for loading little endian data.

```
! All load instructions below load 8 bytes into a register.
! Registers are 64 bits.

ldx [%l1], %l2      ! Ordinary load.  For big-endian data.

ldxle [%l1], %l2    ! Not a real SPARC insn.  For little-endian data.

ldxa [%l1] 0x88, %l2    ! SPARC's load for little-endian data.
```

(*a*) The `ldxa` instruction is an example of an alternate load instruction. The alternate load instructions are intended for three kinds of access. Briefly describe the three kinds and indicate which one is used above. What symbolic name does JPS1 give for 0x88 above?

Note: To answer this question one must read through material dealing with topics not yet covered, for example, the concept of multiple address spaces. It is only necessary that the concept of multiple address spaces is vaguely understood. The kind of access done by the `ldxa` should be clearly understood.

The symbolic name for `0x88` is `ASI PRIMARY LITTLE`.

The three kinds of access are:

Accesses to an alternate address space. The ASI acts like extra bits to put on the end of an address. For example, suppose a `ldxa` specified an ASI of `0x12` and an address of `0x00000000abcd0124`. The full address would be `0x1200000000abcd0124`. There are many uses of such ASIs, one is to allow OS code to access its own memory space and the memory space of a process it needs to work with.

Access to special machine registers. The ASI indicates which set of machine registers, and the address specifies a particular register. The machine registers are used to control hardware, for example, the memory system or a video card.

Variations on a normal memory access. This includes little-endian byte ordering, and also includes things like fault-free loads, and new data sizes (such as byte loads to a floating point register). In this use the ASI acts like an extension of the opcode field.

Grading Note: This was much harder than intended.

(*b*) Show the encoding for the three instructions above. The `ldx` and `ldxa` are real instructions, so it's just a matter of looking things up. For the `ldxle` make up an appropriate encoding.

The encodings appear below. There are two possible ways to encode the `ldx`: with an immediate (shown below) or with rs2 set to `g0`. The only difference between `ldx` and `ldxle` is the opcode. An unused opcode was found for `ldxle` using the opcode map, in particular table E-4.

op	rd	op3	rs1	i	simm13	
3	18 (%l2)	0x0b	17 (%l1)	1	0	`ldx [%l1], %l2`
31 30	29 25	24 19	18 14	13 13	12 0	

op	rd	op3	rs1	i	simm13	
3	18 (%l2)	0x2b	17 (%l1)	1	0	`ldxle [%l1], %l2`
31 30	29 25	24 19	18 14	13 13	12 0	

op	rd	op3	rs1	i	imm asi	rs2	
3	18 (%l2)	0x1b	17 (%l1)	0	0x88	0 (%g0)	`ldxa [%l1] 0x88, %l2`
31 30	29 25	24 19	18 14	13 13	12 5	4 0	