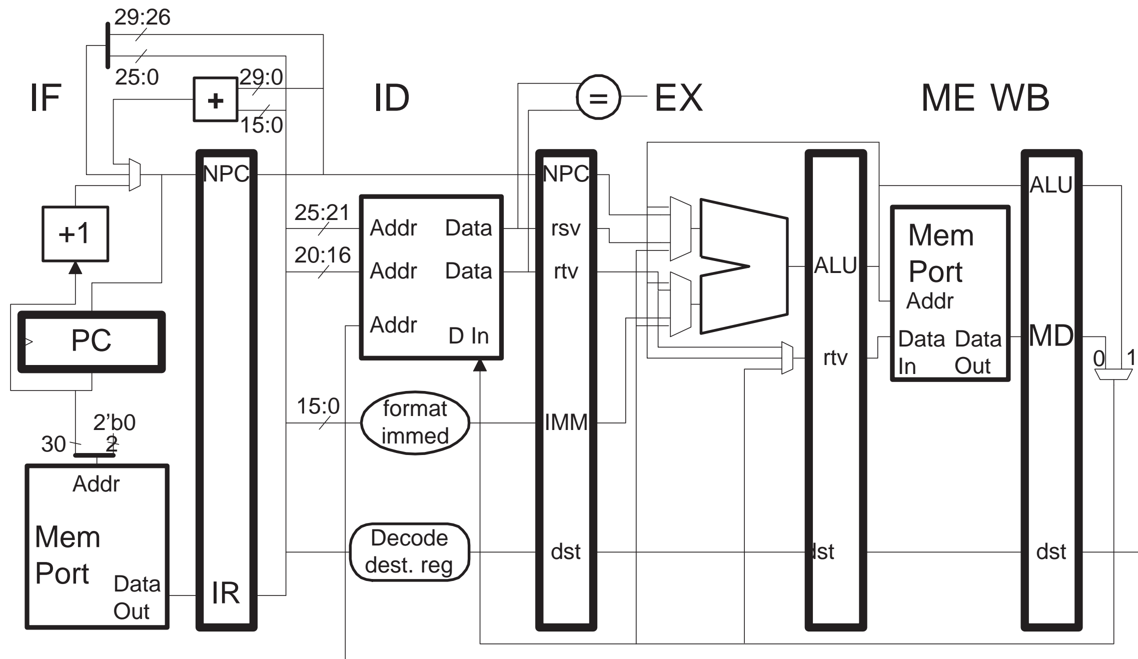
LSU EE 4720 Homework 2 Due: 15 October 2008

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
```



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.

(*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.