LSU LSU EE 4720 **Homework 3** Solution **Due: 7 March 2014**

For this assignment read the ARM Architecture Reference Manual linked to `http://www.ece.lsu.edu/ee4720/reference.html`. *This assignment asks about the ARM A32 instruction set.*

Problem 1: Show the encoding of the ARM A32 instruction that is most similar to MIPS instruction `add r1, r2, r3`.

```
# Arm Add
add r1, r2, r3    # Registers
add Rd, Rn, Rm    # Register field symbols.
```

The encoding appears below. The `cond` field is set to 1110_2 because the instruction should execute unconditionally. The `fmt` field (name made up) set to zero to indicate a data processing instruction. The `opcode` and bit position 4 fields are set based on the description of the `add` instruction. The `S` is set to 0 because we don't want to write condition code registers (since the MIPS `add` doesn't either). The instruction should not shift, so we set `type` and `imm5` to zero. The `Rn`, `Rd`, and `Rm` fields values are based on the register numbers in the example.

cond	fmt	opcode	S	Rn	Rd	imm5	type		Rm
1110_2	0	0100_2	0	2	1	0	0	0	3
31 28	27 25	24 21	20	19 16	15 12	11 7	6 5	4	3 0

Problem 2: ARM instructions can shift one of its source operands, something MIPS cannot. With this feature the code below can be executed with a single ARM add instruction. Show the encoding of such an ARM A32 add instruction.

```
sll r1, r2, 12
add r1, r4, r1
```

The assembly language for the ARM A32 equivalent is:

```
# Arm assembler
add r1, r4, r2, LSL #12
```

The encoding of the ARM instruction appears below. Notice that it is the same as the ordinary `add`, but with a shift specified by putting a non-zero value in the `imm5` field.

cond	fmt	opcode	S	Rn	Rd	imm5	type		Rm
1110_2	0	0100_2	0	4	1	12	0	0	2
31 28	27 25	24 21	20	19 16	15 12	11 7	6 5	4	3 0

Problem 3: So, the ARM `add` instructions can shift one of its operands, something that MIPS would need two instructions to do. Since we have been working with MIPS for so long it would be natural for us to get protective of MIPS and defensive or jealous when hearing about wonderful features of other ISAs that MIPS doesn't have. To relieve these negative emotions lets add operand shifting to MIPS with a new `addsc` instruction. The `addsc` instruction will use MIPS' `sa` field to specify a shift amount. So instead of, for example, the following two instructions:

```
sll r1, r2, 12
```

```
add r1, r4, r1
```

We could use just

```
addsc r1, r4, r2, 12
```

where the "12" indicates that the value in `r2` should be shifted by 12 before the addition.

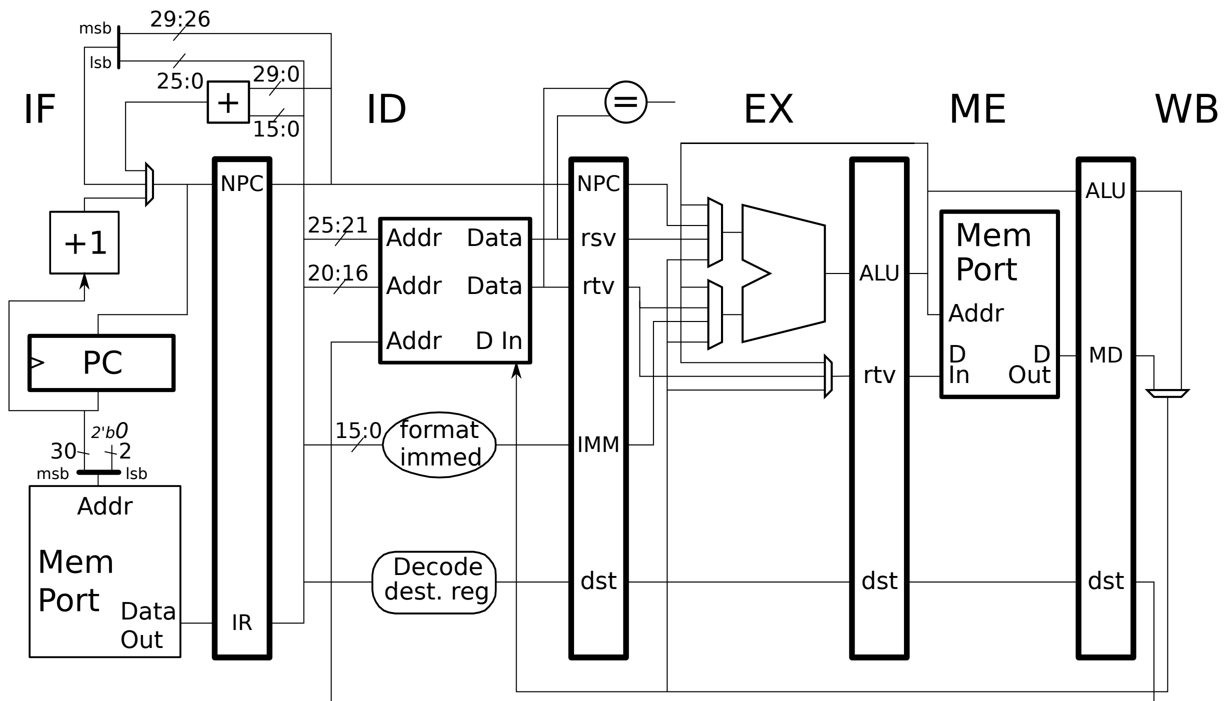
Modify our five-stage MIPS implementation so that it can implement this instruction. (See below for diagrams.)

- The addsc should execute without a stall.
- Don't break existing instructions.
- Don't increase the critical path by more than a tiny amount.
- Keep an eye on cost.

Assume that both the ALU and shift unit take most of the clock period. **This means if the ALU and shifter are in the same stage and output of the shifter is connected to the ALU, the critical path will be doubled. (Of course, doubling the critical path would be disastrous for performance.)**

There are several ways to solve this, one possibility includes adding a sixth stage, another possibility uses a plain adder (not a full ALU) in the EX stage.

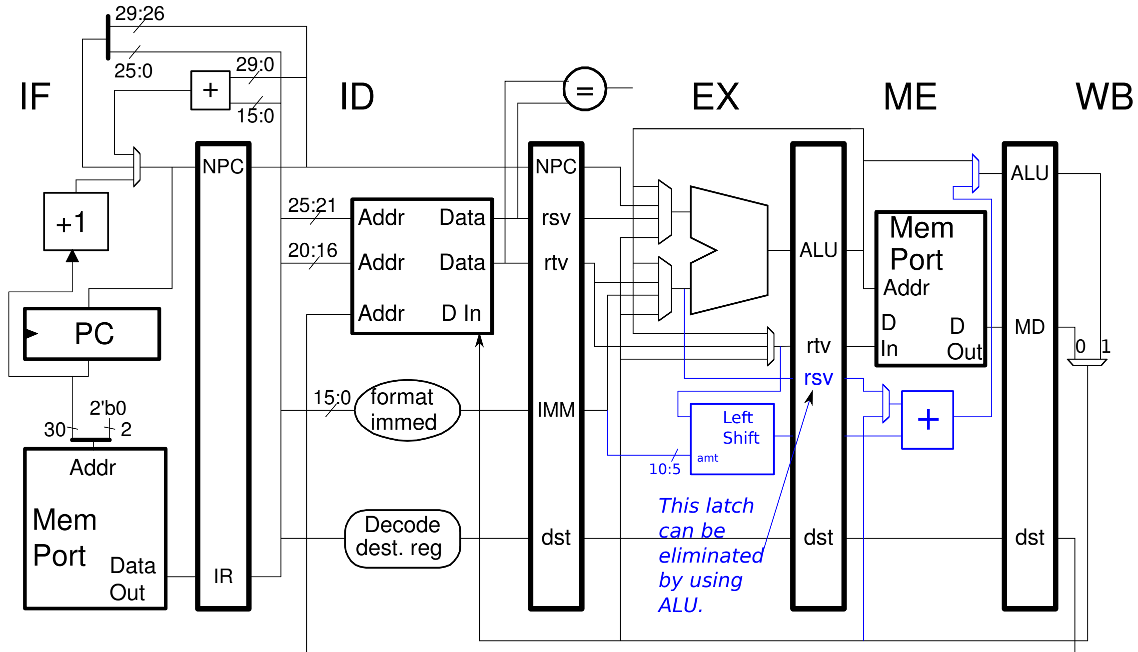
Add hardware to the implementation below. Source files for the diagram are at:
`http://www.ece.lsu.edu/ee4720/2013/mpipei3.pdf`,
`http://www.ece.lsu.edu/ee4720/2013/mpipei3.eps`,
`http://www.ece.lsu.edu/ee4720/2013/mpipei3.svg`. The `svg` file can be edited using Inkscape.



To see how a shift unit can be added to MIPS see Fall 2010 Homework 3.

One big choice is between adding a stage or adding an adder. If a stage is added then additional pipeline latches are needed, so the decision should be based on the cost of the latches versus the cost of the adder. An additional factor is the number of bypass paths needed. In the solution below the decision was made to add and adder, but in two different ways.

In the first version `rsv` is added to the `EX/ME` pipeline latch so that `rsv` will be available in the `ME` stage.



In the second version (below) the ALU is used to bring `rsv` to the `ME` stage. In this second version, when an `addsc` instruction is in the `EX` stage the ALU will perform a *Pass A* operation in which the upper ALU input is passed to the output unchanged. Also, a multiplexor is saved by using the `ME`-stage adder to pass results of non-scaled add instructions, this is done by setting the Left Shift unit to output a zero when a non-scaled add instruction is in `EX`.