LSU EE 4720 **Homework 3** Solution **Due: 3 November 2004**

Problem 1: Do Problems 1 and 2 From Spring 2004 Homework 3 `http://www.ece.lsu.edu/ee4720/2004/hw03.pdf`. After completing the problems look at the solution and assign yourself a grade. The maximum grade should be 10 points, divide the points between problems as you wish.

Problem 2: A new instruction, `copyTreg rt, rs`, will read the contents of register `rt` and `rs` and will write the contents of `rs` to the register number specified *by the contents* of register `rt` (not into register `rt`). For example,

```
# Before:  $1 = 4,  $2 = 0x1234,  $4 = 0
copyTreg $1, $2
# After:   $1 = 4,  $2 = 0x1234,  $4 = 0x1234;
#          (Register $4 written with contents of register 2.)
```

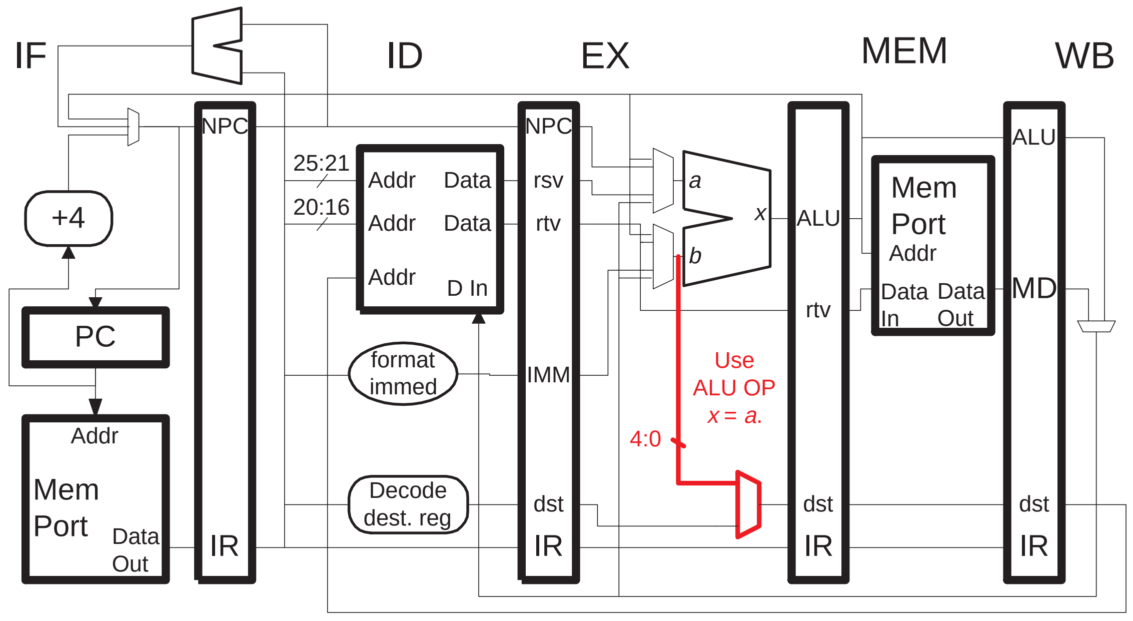
Note that this is a variation on Midterm Exam 1 Problem 3, with the destination, rather than the source, being specified in a register.

(*a*) Modify the pipeline below to implement this instruction.

(*b*) Add the bypass connections needed so that the code below executes correctly.

```
# Before:  $1 = 4,  $2 = 0x1234,  $4 = 0,  $5 = 0
addi $1, $0, 5
copyTreg $1, $2
# After:   $1 = 5,  $2 = 0x1234,  $4 = 0,  $5 = 0x1234;
```

Changes shown in red below. The major change is providing a path for the `rt` value to be used as a destination register, that is through the added multiplexer. It is connected to the output of the ALU mux so that it can make use of the already existing bypass connections. When `copyTreg` is in EX the ALU will be set to add its a input to zero (or otherwise set output x to whatever is on input a). The ALU might already be using such an operation for the `jal` and `jalr` instructions. (The modification solves both parts (a) and (b) of this problem.)

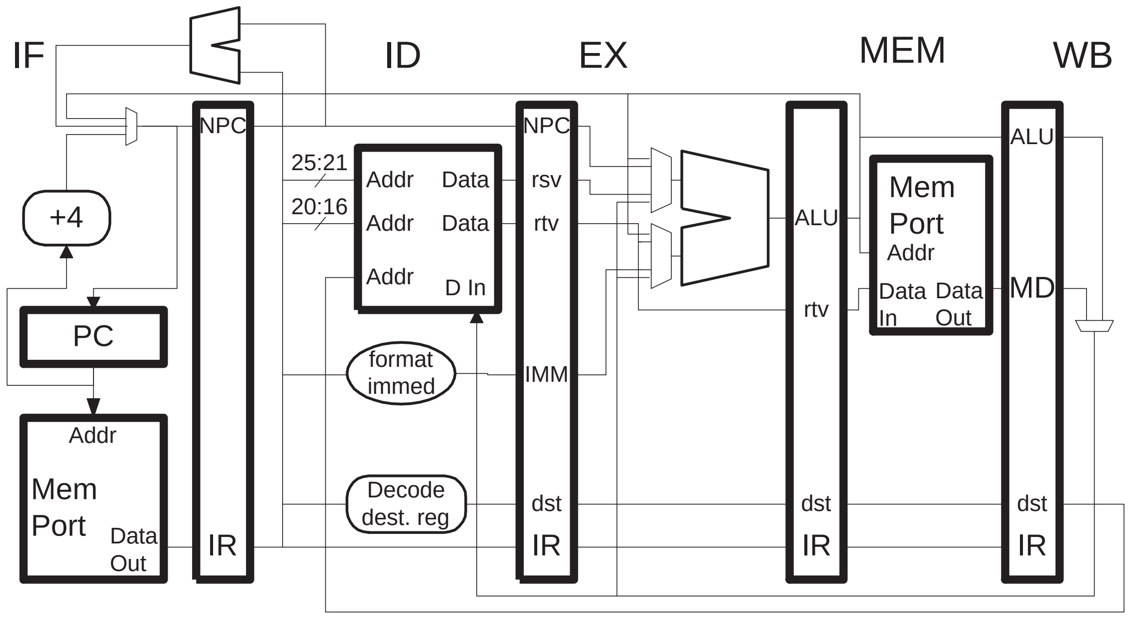


Problem 3: In the problem above the register number to *write to* is in a register. Here consider `copyFreg rt, rs` in which, like the test question, the register to *copy from* is in a register. That is,

```
# Before:  $1 = 2,  $2 = 0x1234,  $4 = 0
copyFreg $4, $1
# After:   $1 = 2,  $2 = 0x1234,  $4 = 0x1234;
#          (Register $4 written with contents of register 2.)
```

Explain a difficulty in implementing this instruction on the pipeline below without vitiating its sublime elegance.

The pipeline and ISA have been designed so that register read can start in the ID stage as soon as the instruction arrives (and in parallel with decoding activities). To implement `copyFreg` the register file has to be read twice, first using the register number found in the `rs` field of the instruction, call the retrieved value `rsv`, and a second time using `rsv` as the register number. The two register reads obviously cannot be done at the same time. There is no way to retrieve the register without either reading the register file twice in the same cycle (stretching the clock), having the instruction spend two cycles in ID (which would be inelegant, providing a new kind of stall), or having the instruction read the register file a second time while it is in EX (requiring a third read port or else introducing new structural hazards as instructions in ID and EX both vie for the same register port).

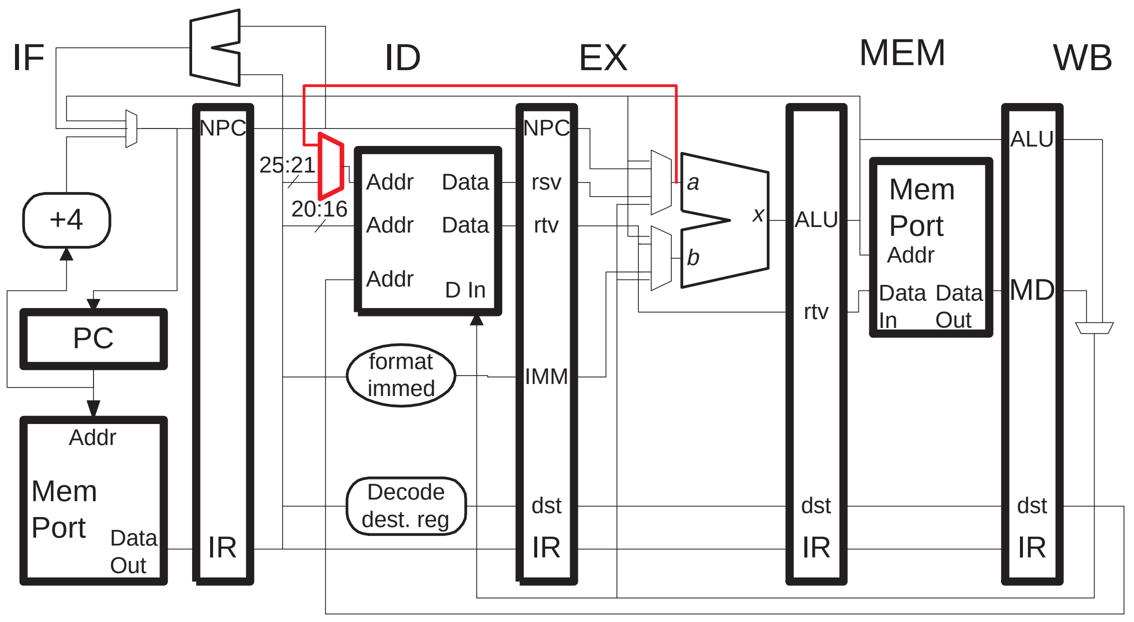


Problem 4: No, we are not vitiators. Instead consider `copyBreg rd` in which the source register to read is specified *in the* `rs` register of the preceding instruction, that value is written into the `rd` register of this instruction. (Okay, maybe we are vitiators.) For example,

```
# Before:  $1 = 2,  $2 = 0x1234,  $4 = 0
add      $0, $1, $0   # Instruction below uses rs ($1 here) of this insn.
copyBreg $4
# After:   $1 = 2,  $2 = 0x1234,  $4 = 0x1234;
#          (Register $4 written with contents of register 2.)
```

Implement this instruction on the pipeline above (from the previous problem).

Changes shown in red below. The added multiplexer does have a small impact because its control signal needs to be generated close to the beginning of the cycle. As with the previous problem, by taking the signal at the output of the ALU mux bypassed values are available. In the code sample above the bypass connections are not needed. The "maybe we are vitiators' remark is the comment on the awkwardness of having one instruction use as a source operand the source operand of a preceding instruction. Such instructions are rare if they exist at all.


IF
ID
EX
MEM
WB
NPC
NPC
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
rsv
rtv
a
b
x
ALU
ALU
Mem
Port
Addr
Data
In
Data
Out
MD
+4
PC
Addr
Mem
Port
Data
Out
IR
format
immed
IMM
Decode
dest. reg
dst
IR
rtv
dst
IR
dst
IR