LSU EE 4720 **Homework 5** **Due: 20 March 2017**

Problem 1: Complete MIPS routine `fxitos` so that it converts a fixed point integer to a single-precision floating point value as follows. When `fxitos` starts register `a0` will hold a fixed-point value i and register `a1` will hold the number of bits that are to the right of the binary point, d, with $d \geq 0$. For example, to represent $9.75_{10} = 1001.11_2$ we would set `a0` to `0b100111` and `a1` to `2`. (Or we could set `a0` to `0b100111000` and `a1` to 5.) When `fxitos` returns register `f0` should be set to $i/2^d$ represented as a single-precision floating point number.

Solve this problem by using a division instruction for $i/2^d$. (The floating division instruction can be avoided by performing integer arithmetic on the FP representation, but that's not required in this problem.)

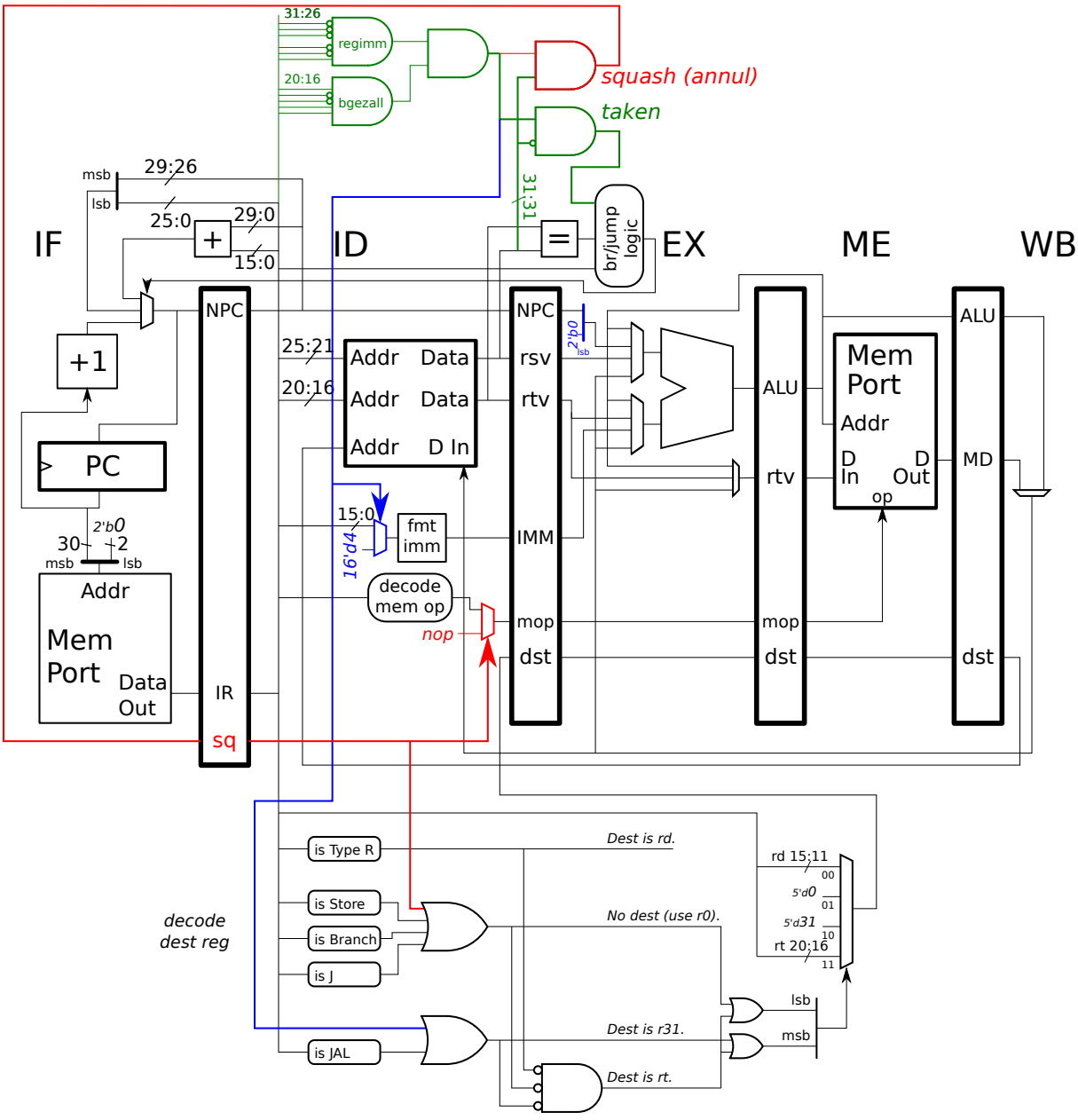
Submit the solution on paper. Your class account can be used to work on the solution. The `fxitos` routine and a testbench can be found in `/home/faculty/koppel/pub/ee4720/hw/2017/hw05/hw05.s`, follow the same instructions as for Homework 1.

```
fxitos:
        ##
        #
        # CALL VALUES:
        #  $a0: Fixed-point integer to convert.
        #  $a1: Number of bits to the right of the binary point.
        #
        # RETURN:
        #  [  ] $f0: The value as a single-precision FP number.



        jr $ra
        nop
```

Problem 2: Appearing below is the MIPS hardware needed to implement `bgezall` from the solution to Homework 3. Recall that with `bgezall` the delay-slot instruction is annulled if the branch is not taken. Modify the hardware for new instruction `bgezalllsu` which executes like `bgezall` when the branch target is at or before the branch, but when the target is after the branch the delay-slot instruction is annulled when the branch is taken and allowed to execute normally if the branch is not taken. The opcode and `rt` values are the same as for `bgezall`. *Hint: this can be done with very little hardware, a gate or two, if that.*



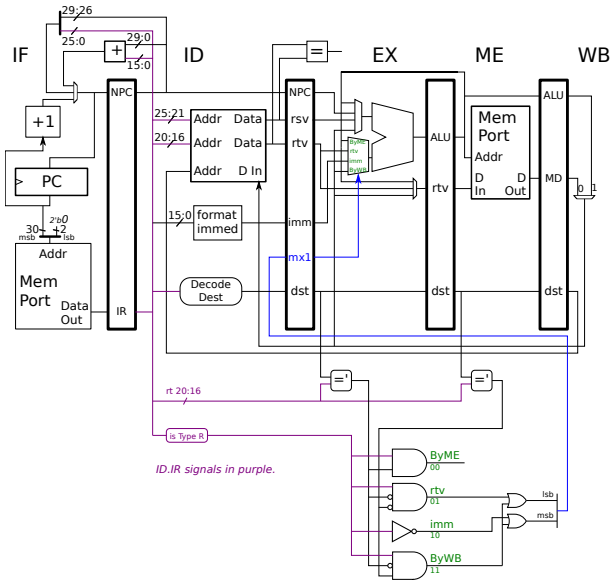
Problem 3: Suppose that an analysis of the execution of benchmark programs on our pipelined MIPS implementation shows that over 75% of bypassed values can be represented with 12 bits or fewer. A low-cost implementation takes advantage of this fact by using 12-bit bypass paths.

(*a*) The control logic below is intended for bypass paths that can bypass a full 32-bit value. Modify the control logic shown so that it works for 12-bit bypass paths. In your modified hardware add a stall signal to be used when values are too large to be bypassed.

- Indicate which parts of the added logic, if any, may lengthen the critical path.
- As always, avoid costly or slow hardware.

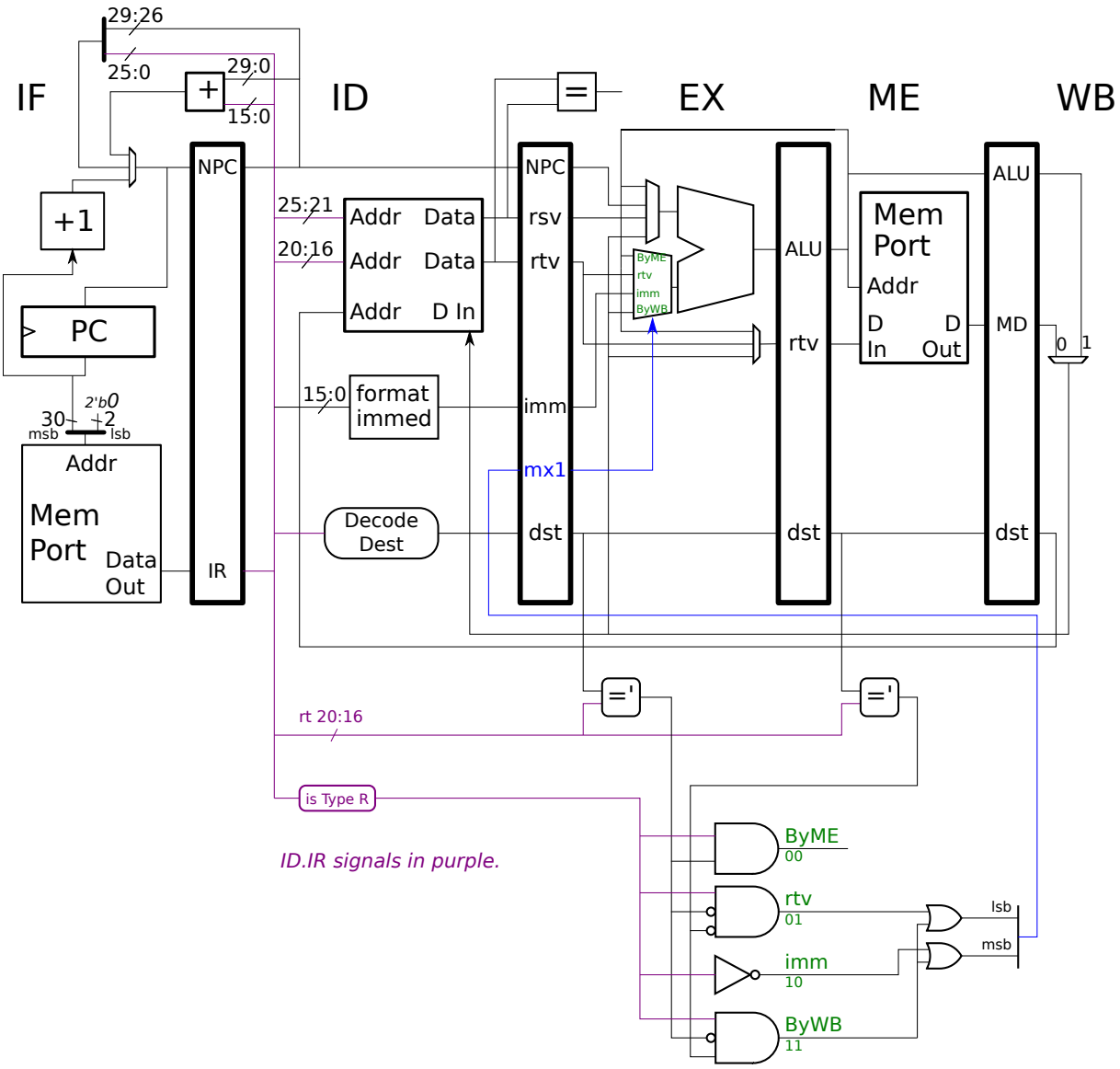
Attention perfectionists: An Inkscape SVG version of the implementation below can be found at `http://www.ece.lsu.edu/ee4720/2017/mpipei3c.svg`.

It's small on purpose, use next page for solution.



It's small on purpose, use next page for solution.

(*b*) Why would it be far more challenging for a compiler to optimize for these 12-bit paths than for ordinary full-width bypass paths?



ID.IR signals in purple.