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

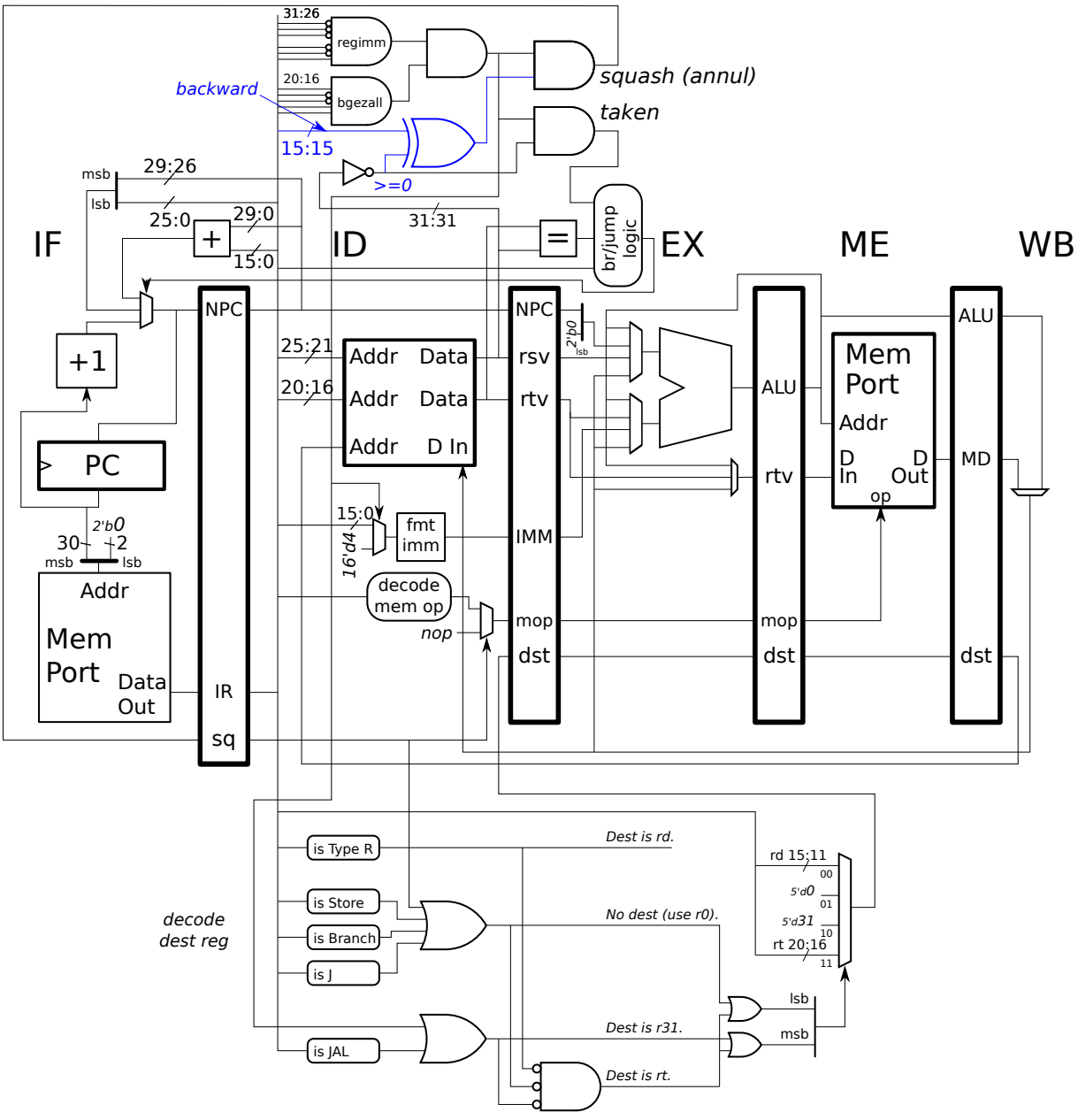
**Problem 1:** Complete MIPS routine `fxitos` so that it converts a fixed point integer to a single-precision floating point value as follows. When `fxitos` starts register `a0` will hold a fixed-point value $i$ and register `a1` will hold the number of bits that are to the right of the binary point, $d$, 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.


        ## . . . . . . . . .
        #
        # Let d denote value of $a1, # of digits to the right of binary point.
        # Let i denote value of $a0, the fixed point number to convert.
        # Need to set $f0 to i / 2^d, where 2^d is 2 to the d'th power.

        addi $t0, $0, 1
        sllv $t2, $t0, $a1   # Construct value 2^d

        mtc1 $a0, $f1        # Move i to a FP register. (But it's still an int.)
        mtc1 $t2, $f2        # Move 2^d to a FP register. (Still an int too.)
        cvt.s.w $f11, $f1    # Convert i from an integer to SP FP.
        cvt.s.w $f12, $f2    # Convert 2^d from an integer to SP FP.

        jr $ra
        div.s $f0, $f11, $f12  # Compute i / 2^d.
```

**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.*




Solution appears above. Whether the branch target is before or after the branch can be determined by looking at the sign bit of the immediate value, which is bit 15 of `ID.IR` and is labeled `backward` in the diagram above. If that bit is

1 it means the immediate is negative and so the branch target is before the branch (to be 100% precise, it means that the branch target is before the delay slot instruction). Branch `bgezalllsu` is taken if `rsv` is not positive, that condition is provided at the output of the NOT gate and labeled >=0 above. Based on the description of `bgezalllsu` given in the problem statement, the delay-slot instruction should be annulled (squashed) if the XOR of these two conditions is true.

Note that in the solution to Homework 3 Problem 1 the NOT gate used to determine the branch-taken condition is shown there as a bubble at an input to the `TAKEN` AND gate. Here it is shown as a free-standing NOT gate so that its value can be used for the XOR. The `TAKEN` signal itself could have been used instead of the NOT gate output, but that would have resulted in a slightly longer critical path. Since a synthesis program could easily optimize the logic the variation to use is the one that's easier for humans to understand.

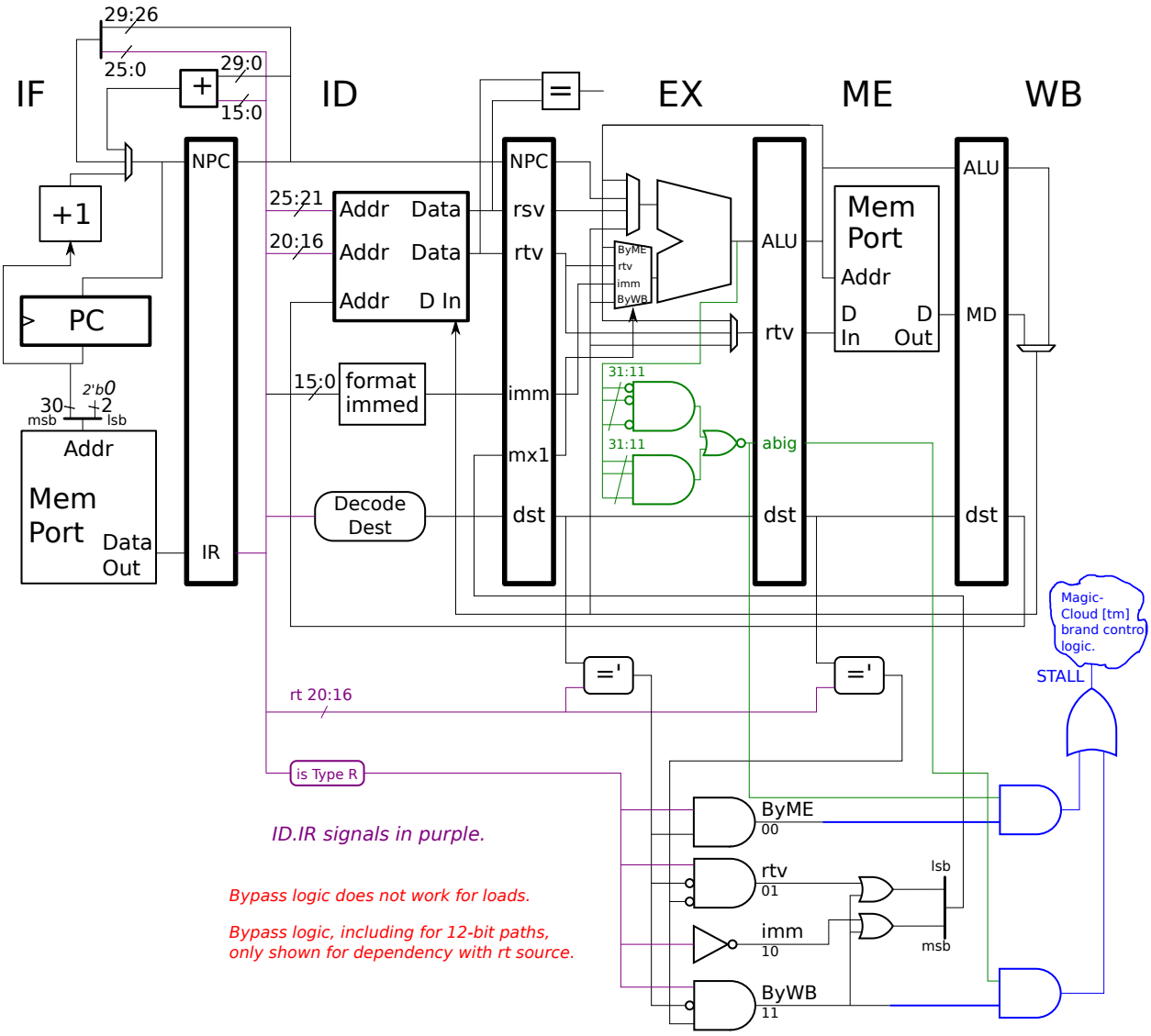
**Problem 3:** Suppose that an analysis of the execution of benchmark programs on our pipelined MIPS implementation shows that over 75% of bypassed values can be represented with 12 bits or fewer. A low-cost implementation takes advantage of this fact by using 12-bit bypass paths.

(*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`.

Solution starts on next page.




Solution appears above. Logic has been added at the output of the ALU (in the EX stage, shown in green) that checks whether the value is too big for the 12-bit bypass paths. It does so by checking whether bits 31 through 11 are either all 0 (a small positive value) or all 1 (a small negative value). (The reason that bits 31:11 rather than 31:12 are checked is to make sure that the sign bit of the 12-bit bypassable portion matches the parts that we won't bypass.) If the value at the output of the ALU is too big to bypass then the output of the NOR gate is 1, that signal is put in the `abig` (ALU output is big) pipeline latch.

Added control logic, shown in blue, checks this `abig` signal. If `ByME` is 1 that means the instruction in `ID` will need to use the `ByME` bypass path in the next cycle (when it is in `EX`). The upper blue AND gate checks whether the value in `EX` is too large to bypass, if so the stall signal is 1. If the instruction in `ID` will need to use the `ByWB` path and the `abig` bit in the `ME` stage is 1 we will also need to stall.

This logic only works for dependencies to the `rt` register (of the instruction in `ID`), and only when the producing instruction (the instruction in `EX` or `ME` while the consuming instruction is in `ID`) uses the ALU (not the memory port) to produce a value. See the examples below.

It would be a simple matter—simple enough for a midterm exam problem—to modify the logic to handle a dependency to the `rs` register of the instruction in `ID`, such as Example II below. On the other hand, bypassing for a `lw` is hopeless due to our usual critical path assumptions.

The logic generating the **abig** signal might be stretching the critical path since it doesn't get started until the ALU produces a value. If we really need such a signal we might ask the ALU guys whether they can design an ALU that produces an **abig**-like signal without threatening the critical path, perhaps taking advantage of carry-lookahead logic, for example.

```
# Example I
# Cycle         0  1  2  3  4  5    # Control logic works for this case.
add r1, r2, r3  IF ID EX ME WB
sub r4, r5, r1     IF ID EX ME WB   # Logic active in cyc 2, bypass in cyc 3.

# Example II
# Cycle         0  1  2  3  4  5    # Control logic NOT shown for this case.
add r1, r2, r3  IF ID EX ME WB
sub r4, r1, r5     IF ID EX ME WB

# Example III
# Cycle         0  1  2  3  4  5  6  # Control logic works for this case.
add r1, r2, r3  IF ID EX ME WB
sub r4, r5, r6     IF ID EX ME WB
xor r7, r8, r1        IF ID EX ME WB # Logic active in cyc 3, bypass in cyc 4.

# Example IV
# Cycle         0  1  2  3  4  5  6  # Control logic doesn't account for this case.
lw r1, 0(r2)    IF ID EX ME WB
sub r4, r5, r6     IF ID EX ME WB
xor r7, r8, r1        IF ID EX ME WB
```

(*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?

The compiler would need to know whether it was possible to use a bypass path for some pair of dependent instructions, which means the compiler would need to know whether the register value would fit in 12 bits. In most cases the compiler will not be able to tell the size of a value in a register because that can depend on input data that can vary from run to run or it might depend on other pieces of code that the compiler does not have access to. There are a few situations in which the compiler could figure it out. For example:

```
andi r1, r2, 0xff   # Value in r1 must be between 0-255.
addi r3, r1, 3      # Value in r3 must be between 3-258.
sub  r4, r5, r3     # Can use 12-bit bypass here for r3.
```

In the example above the values in **r1** and **r3** can easily fit in 12 bits (since the **r1** value was masked down to 8 bits). Therefore the compiler can assume a bypass can be used from the **addi** to the **sub** and so it will not was time finding instructions to put between them.

The compiler could also use profiling to determine at least a range of values for registers. The compiler doesn't need to be 100% sure that register values are small, since the hardware will stall if the values are too large.

The problem statement mentioned that 75% of bypassed values fit within 12 bits, and we might expect that a profiling analysis to come to the same conclusion. However that doesn't really help us because that doesn't mean that 75% of dependent instruction pairs pass values that fit within 12 bits.