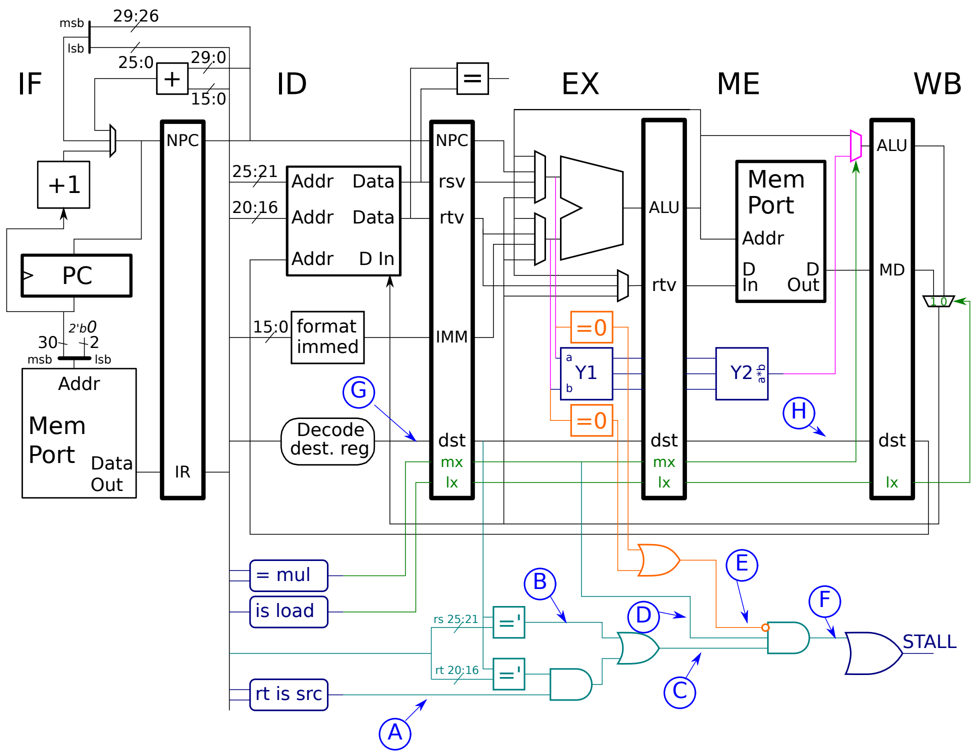
LSU EE 4720 **Homework 3** Solution **Due: 30 March 2020**

Please E-mail solutions of this assignment to koppel@ece.lsu.edu by the evening of the due date. PDF files are preferred. These can be generated by scanning software that you might have installed with a multifunction printer. A PDF can also be assembled from photos of a hand-completed copy. The disorganized homework penalty will be ignored for the remainder of the semester (unless we return early) so an E-mail with multiple image attachments will be accepted without penalty. **Do not** *physically mail them to my office address, I will not be able to pick them up.*

Problem 1: Appearing below is the solution to Homework 2 with labels added to some wires, which is followed by an execution of the code showing values on those labeled wires. The execution is based on the code fragment shown plus `nop` instructions before the first instruction (`addi`) and after the last instructions (`nop`).



```
# Cycle          0     1     2     3     4     5     6
addi R2, r0, 0   IF    ID    EX    ME    WB
mul R1, R2, r3         IF    ID    EX    ME    WB
add r4, R2, R1               IF    ID    EX    ME    WB
# Cycle          0     1     2     3     4     5     6
A                      0     1     1
B                      0     1     0
C                      0     1     1
# Cycle          0     1     2     3     4     5     6
D                            0     1     0
E                            1     1     1
F                      0     0     0     0
# Cycle          0     1     2     3     4     5     6
G                      2     1     4
H                                  2     1     4
# Cycle          0     1     2     3     4     5     6
```

(*a*) Refer to the table on the previous page for this problem. Notice that the value in the `B` row (above) in cycle 1 is `0`. According to the problem statement the instruction before `addi` is a `nop`.

Why would that value be `0` regardless of what instruction came before `addi`?

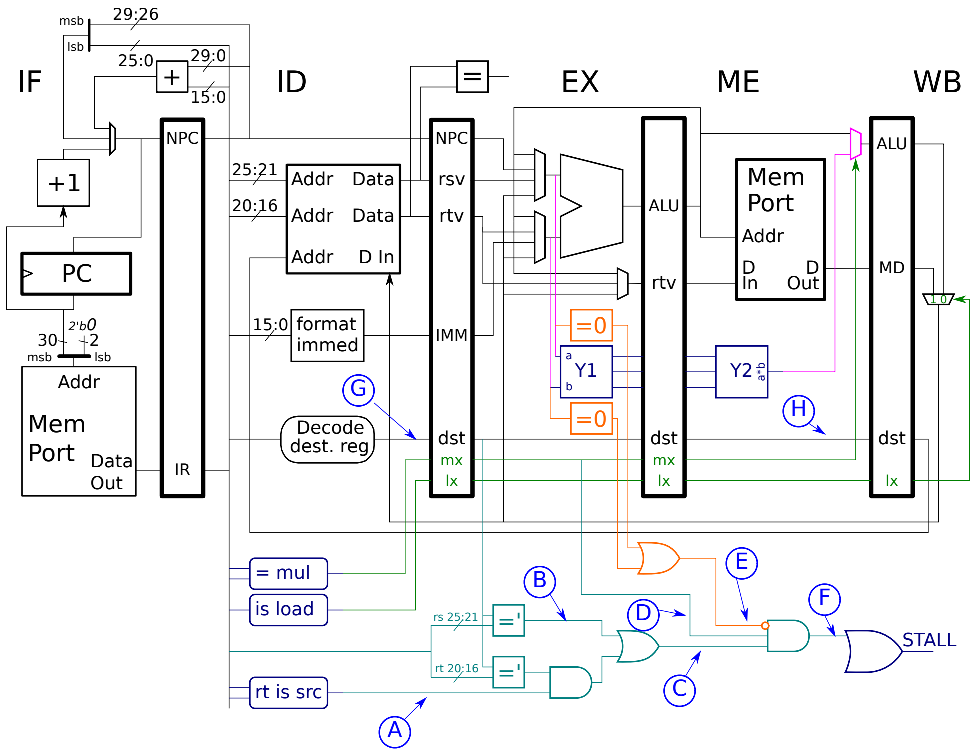
Short Answer: Because the `rs` register is `r0`, and the output of the $\boxed{='}$ comparison unit is `0` when either input is `0` (even if both inputs are `0`).

Explanation: The `B` signal is `1` when the `rs` register of the instruction in `ID` is the same as the destination register of the instruction in `EX`. The `rs` register is the register specified by bits 25:21 if the instruction, and is usually the first source register of the instruction when written in assembly language. In the code fragment above the `rs` register for the `addi` is `r0` and the `rs` register for the `mul` and `add` are `r2`. (Register `r2` is the same as R2, upper case is used only for emphasis.) Depending on the instruction the destination register might be in the `rd` field (most type R instructions), the `rt` field (many type I) instructions, or an implicit `r31` (the `jal` instruction). If an instruction does not write any general purpose (integer) register, the destination register 0 is used.

The output of the comparison unit $\boxed{='}$ is `1` when the two inputs (which are 5-bit quantities) are equal, with one exception: if both inputs are zero the output is `0`. The reason for the exception is that `r0` is not a real register, meaning that an instruction with `r0` as a source (such as `addi`) does not need to wait for an instruction to write `r0` (such as the `nop` before the `addi`).

Suppose the `addi r2, r0, 0` were changed to `addi r2, r7, 0`. Why would the value in the `B` row still be `0`?

Because the destination of a `nop` is `r0` and $0 \neq 7$.



(*b*) Appearing below is a different code fragment. Complete the table so that it shows the values on the labeled wires.

The solution appears below. As a start to understanding the solution examine row G and H, these show the destination register of the instruction in ID and ME, respectively. Row D is also straightforward, it is 1 when the instruction in EX is a `mul`.

To solve line E one needs to determine if the value of at least one of the operands of the instruction in EX is zero. This is certainly false in cycle 3 when `sub` is in EX because `r2` must be non-zero (look at the `ori` instruction). But, `r1` must be zero and so B is true in cycle 4. Because the multiplicand of the first multiply is `r1` the product, `r3` must also be zero. Back in cycle 2, when the `ori` is in EX, there is no way to tell if `r6` is zero, so the B value is shown as `?`.

```
# Cycle          0     1     2     3     4     5     6     7
ori r2, r6, 7    IF    ID    EX    ME    WB
sub r1, r2, r2         IF    ID    EX    ME    WB
mul r3, r8, r1               IF    ID    EX    ME    WB
mul r5, r3, r4                     IF    ID    EX    ME    WB
```

```
# Cycle          0     1     2     3     4     5     6    ## SOLUTION
A                      0     1     1     1
B                      0     1     0     1
C                      0     1     1     1
# Cycle          0     1     2     3     4     5     6
D                            0     0     1     1
E                            ?     0     1     1
F                      0     0     0     0
# Cycle          0     1     2     3     4     5     6
G                      2     1     3     5
H                                  2     1     3     5
# Cycle          0     1     2     3     4     5     6
```

(*c*) Appearing below are completed tables, but without a code fragment. Show a code fragment that could have produced those table values.

The originally assigned problem contained an error which made it difficult to solve. Shown below is the originally table, followed by the intended table. The solution uses the intended table.

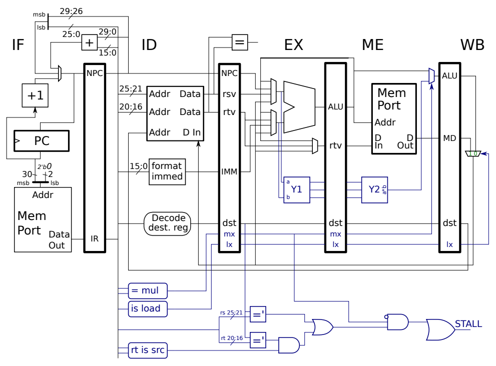
```
# As originally assigned. Contains an error in F at cycle 3.
# Cycle          0     1     2     3     4     5     6     7
A                      0     1           0     1
B                      0     1           1     0
C                      0     1           1     1
# Cycle          0     1     2     3     4     5     6     7
D                            0     1           0     0
E                            0     0           0     0
F                            0     0           0     0
# Cycle          0     1     2     3     4     5     6     7
G                      2     3           4     8
H                                  2     3           4     8
# Cycle          0     1     2     3     4     5     6     7


# Intended problem.
# Cycle          0     1     2     3     4     5     6     7
A                      0     1           0     1
B                      0     1           1     0
C                      0     1           1     1
# Cycle          0     1     2     3     4     5     6     7
D                            0     1     0     0     0
E                            0     0           0     0
F                            0     1     0     0     0
# Cycle          0     1     2     3     4     5     6     7
G                      2     3           4     8
H                                  2     3           4     8
# Cycle          0     1     2     3     4     5     6     7
```

Solution appears below. Lower case characters are used for register numbers and instructions which are one of several possible correct answers. For example, the first instruction `orI`, could have been any type I instruction that wrote a register, such as `addI` and `xorI`. The destination of `orI` must be `r2` (and so it is written as `R2`) but the first source could be any register.

```
# SOLUTION
# Cycle          0     1     2     3     4     5     6     7     8
orI R2, r2, 7    IF    ID    EX    ME    WB
MUL R3, R2, r2         IF    ID    EX    ME    WB
addI R4, R3, 9               IF    ID    ->    EX    ME    WB
sub R8, r10, R4                    IF    ->    ID    EX    ME    WB
# Cycle          0     1     2     3     4     5     6     7     8
```

Problem 2: Appearing below and on the next page is the solution to Homework 2 Problem 1. In this problem add hardware to handle a different and less special multiplication special case. Suppose that the middle output of the `Y1` stage of the multiplier held the correct product whenever the high 24 bits of its `b` input are zero. For example, when `b` is 1, 5, or 255. Call such values *small*. In all cases the correct product appears at the output of `Y2`.

Note: All outputs of `Y1` arrive with zero slack, even the center output with the small `b` special case. That means that nothing can be done with these values until the next clock cycle, at least without reducing the clock frequency.



(*a*) Add hardware to bypass the product to the ALU and to the `rtv` mux when `b` is small. (There is a larger diagram on the next page.) The bypass should allow the first code fragment below to execute without a stall.

(*b*) Add control logic to suppress the stall when it is possible to bypass.

In the first code fragment below the stall is avoided because the `b` value (which is the `rtv`) is small, in the second it is too large.

```
# Cycle          0  1  2  3  4  5  6  7
addi r1, r0, 23  IF ID EX ME WB
mul r2, r3, r1      IF ID EX ME WB
sub r4, r2, r5         IF ID EX ME WB
```

```
# Cycle          0  1  2  3  4  5  6  7
addi r1, r0, 300 IF ID EX ME WB
mul r2, r3, r1      IF ID EX ME WB
sub r4, r2, r5         IF ID -> EX ME WB
```

- Make sure that the changes don't break existing instructions.
- As always avoid costly solutions.
- As always pay attention to critical path.

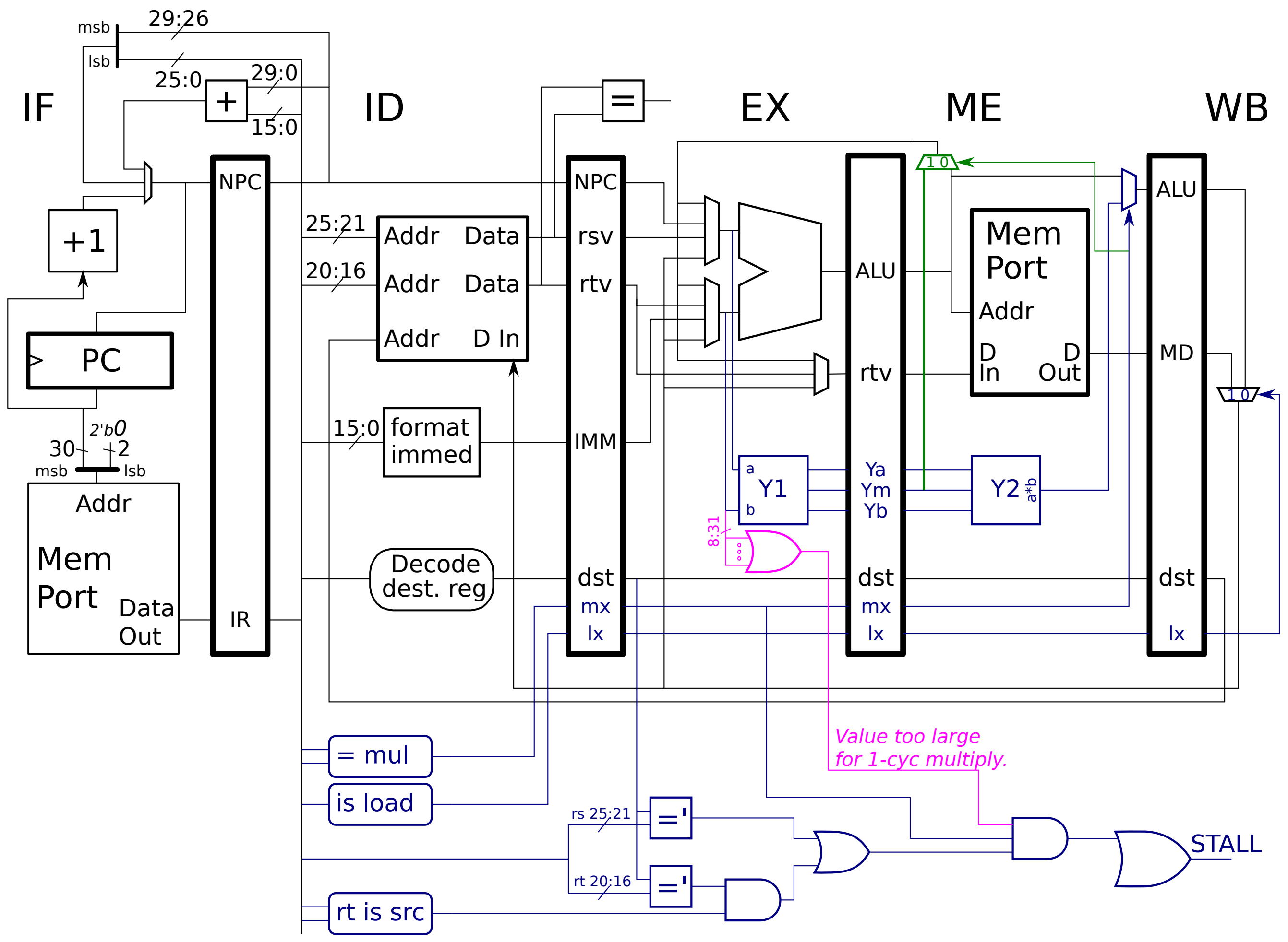
The SVG source for the illustration below is at `https://www.ece.lsu.edu/ee4720/2020/hw03-p2.svg`. It can be edited using Inkscape or any other SVG editor, or (not recommended) a text editor.

Solution appears below with part (a), the datapath, in green and part (b), the control logic, in purple.

A path is provided from the middle `Y1` output to the `ME`-to-`EX` bypass paths and uses a new multiplexor which selects between `ME.ALU` and `ME.Ym`. Pipeline latch register `ME.mx` is used as the select signal for this mux. Note that the mux does not affect the path from `ME.ALU` to the Mem Port Addr input. That's important because we always assume that the Mem Port Addr input and D Out are on the critical path and so anything that increases the length (time) of the path will slow the clock frequency.

For part (b) we need to suppress the stall if the `b` input to the multiplier is < 256. For unsigned values that is true if bits 8 to 31 are zero, which is easily checked by an NOR gate. The logic that is shown checks whether the value is not small (≥ 256), and uses that as an additional condition for the stall. (So the multiply-dependence stall will not be asserted if the value is small.)

Alternative Solution: Rather than adding a new multiplexor, the `ME.Ym` signal could have been connected to the three `EX`-stage muxen. This would have cost more and made the control logic more complicated, but it possibly would be faster, depending on how the five-input multiplexors were synthesized. Such solutions received full credit, even without the control logic for the `EX`-stage multiplexors.


IF
ID
EX
ME
WB
msb
lsb
29:26
25:0
29:0
15:0
+
=
+1
PC
NPC
IR
30
2
2'b0
Addr
Mem Port
Data Out
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format immed
Decode dest. reg
rsv
rtv
IMM
dst
mx
lx
ALU
rtv
Ya
Ym
Yb
Y1
a
b
Y2
a*b
8:31
Mem Port
Addr
D In
D Out
MD
1 0
= mul
is load
rt is src
rs 25:21
rt 20:16
='
Value too large for 1-cyc multiply.
STALL