Name Solution

# Computer Architecture
# EE 4720
# Midterm Examination

Monday, 19 March 2018, 9:30–10:20 CDT

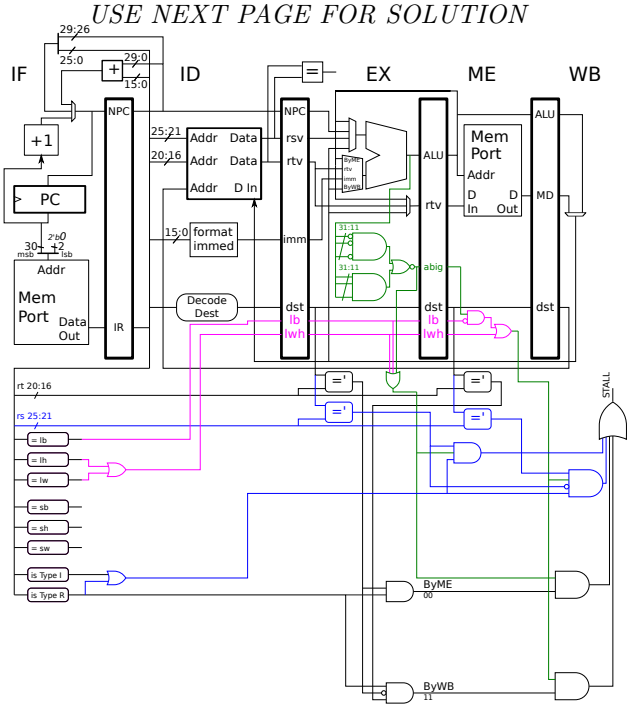
| | | |
|---|---|---|
| Problem 1 | _____ | (25 pts) |
| Problem 2 | _____ | (20 pts) |
| Problem 3 | _____ | (15 pts) |
| Problem 4 | _____ | (40 pts) |
| Exam Total | _____ | (100 pts) |

Alias or r4, 25, r6

*Good Luck!*

Problem 1: [25 pts] Appearing below is the solution to Homework 4, in which additional control logic is added for the 12-bit bypass paths. The illustrated hardware generates stall signals for a load/use dependence and for cases in which the value that needs to be bypassed is unknown or too wide for the 12-bit bypass paths.

*USE NEXT PAGE FOR SOLUTION*



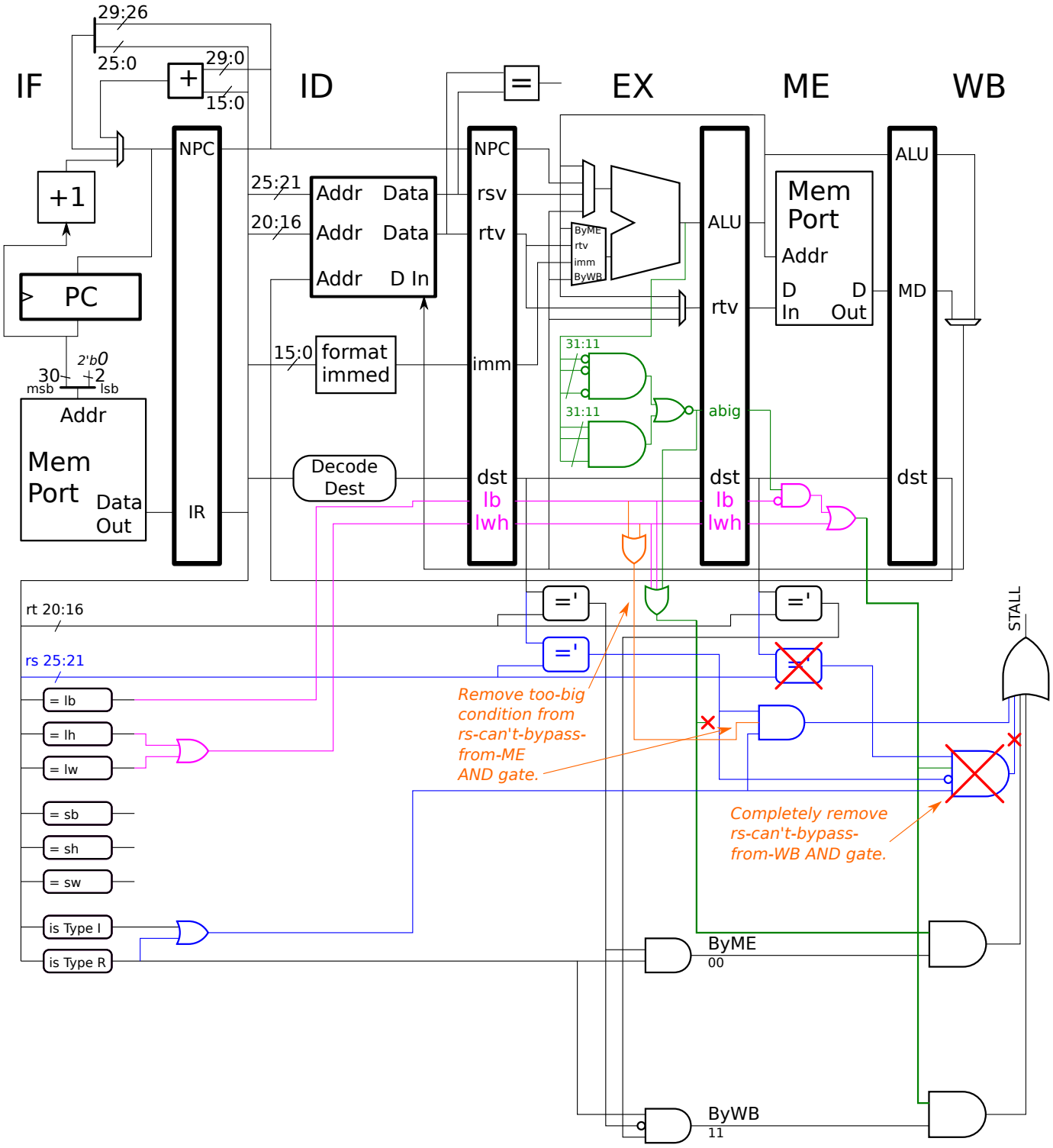

*USE NEXT PAGE FOR SOLUTION*

(*a*) Suppose, on further consideration, it was decided that full-sized, 32-bit bypass paths to the upper ALU mux were needed. Elsewhere 12-bit bypass paths would be retained. Modify the hardware so that a stall signal would no longer be generated for such too-big values to the upper ALU mux. Do so **without** affecting stalls for the 12-bit bypass paths to the lower ALU mux and without affecting stalls for load/use dependencies.

```
# Should not stall anymore, regardless of size of r1.
add $r1, $r2, $r3
lw $r4, 0($r1)

# Should still stall.
lw $r5, 0($r6)
addi $r7, $r5, 9

# Should still stall if r1 too big for 12-bit bypasses.
add $r1, $r2, $r3
sub $r5, $r6, $r1
```

☑ Remove hardware generating stall due to values too large for upper ALU bypass paths.

☑ **Do not** change stalls for load/use cases and bypasses to lower ALU mux.



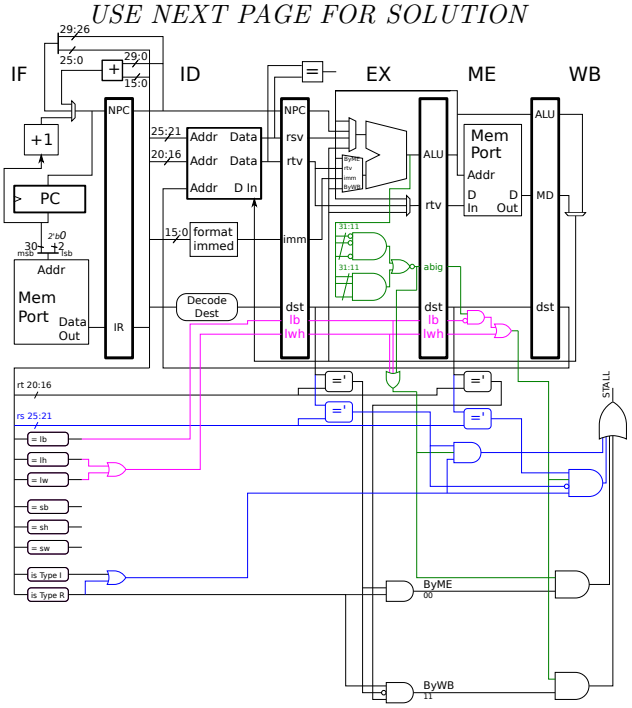

Solution appears above in orange.

Problem 1, continued:

(*b*) Suppose that the bypass paths to the `EX`-stage `rtv` mux were also just 12 bits wide. Modify the control logic on the next page so that a stall signal would be generated when appropriate for store instructions.

Note that a `lb` produces a value small enough for the 12-bit bypass paths and that the store value needed by a `sb` is always small enough for the 12-bit bypass paths. See the examples below.

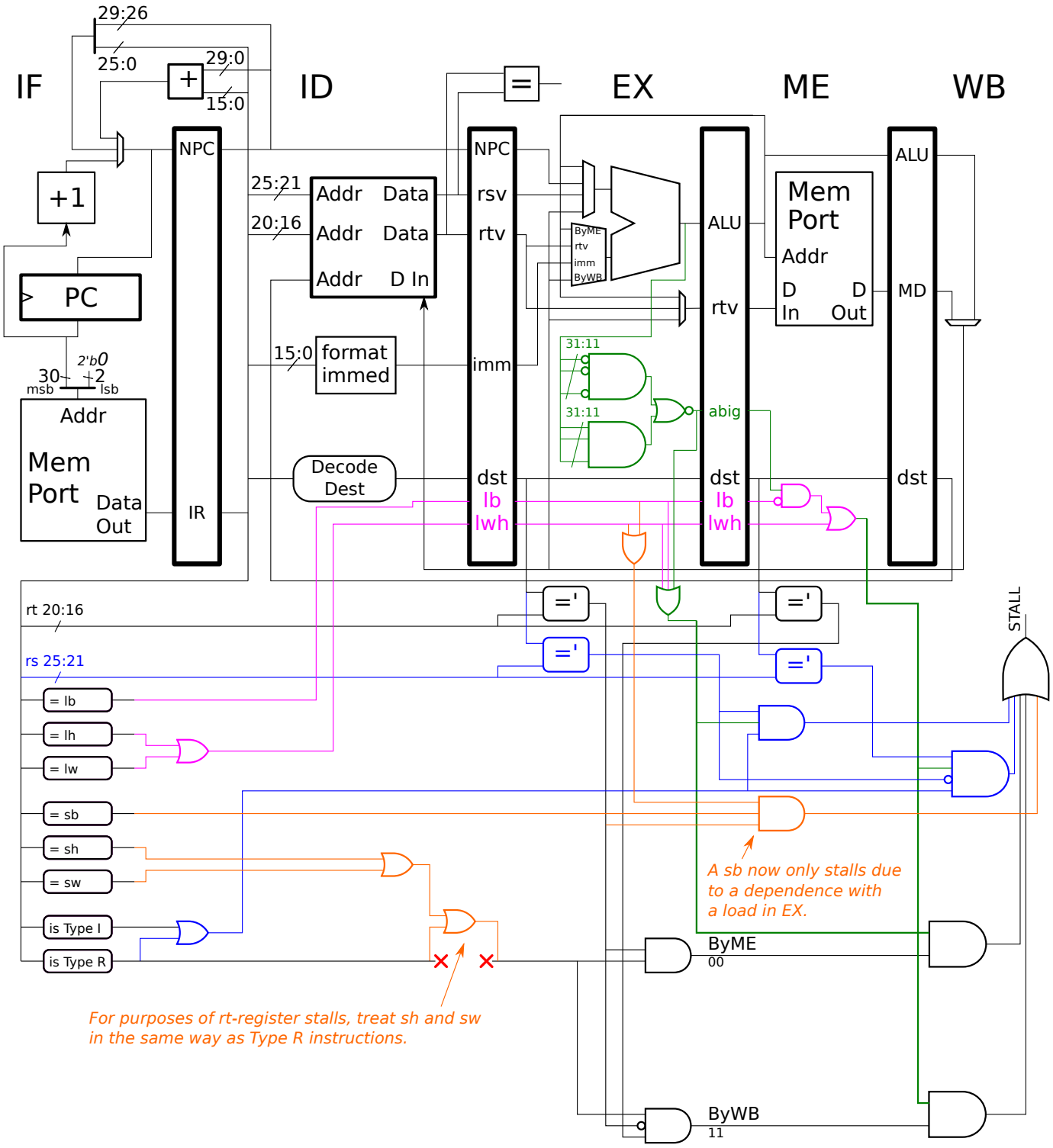
*USE NEXT PAGE FOR SOLUTION*




*USE NEXT PAGE FOR SOLUTION*

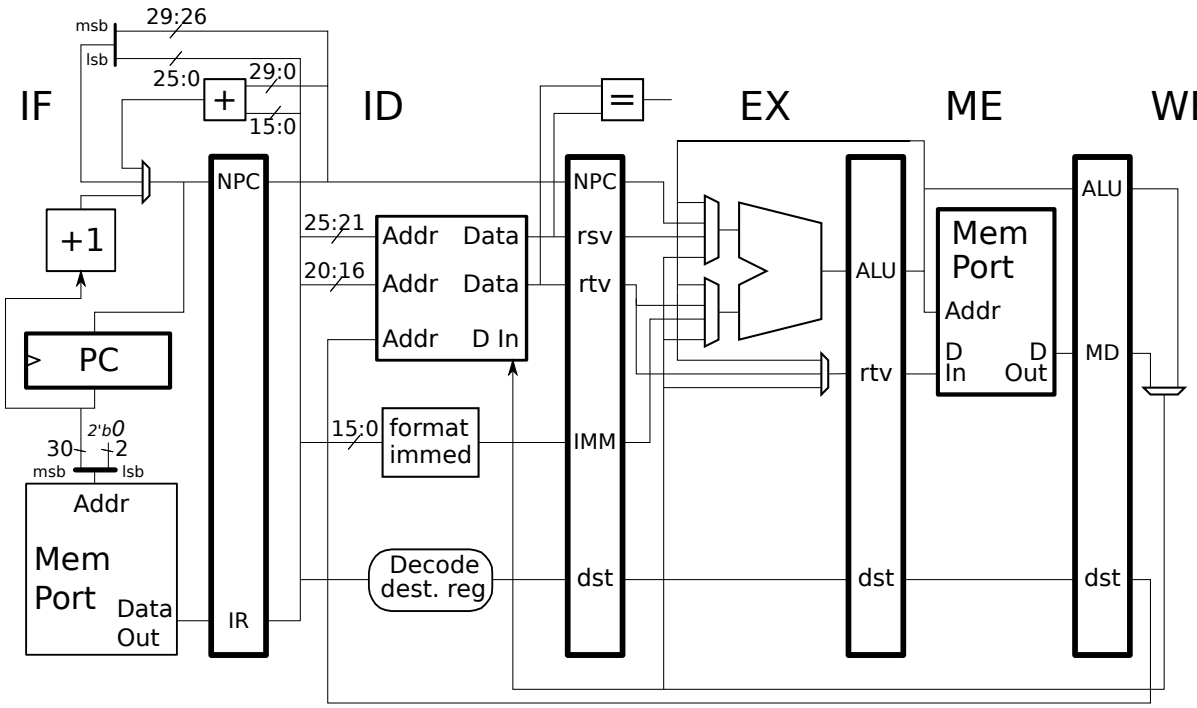
```
# sb should not stall for this dependency.
add $r1, $r2, $r3  IF ID EX ME WB
sb $r1, 0($r4)        IF ID EX ME WB

# sw should stall for one cycle.
lb $r1, 0($r5)     IF ID EX ME WB
sw $r1, 0($r4)        IF ID -> EX ME WB

# sh should stall only if r1 is too large.
add $r1, $r2, $r3  IF ID EX ME WB
sh $r1, 0($r4)        IF ID ----> EX ME WB
```

☑ Generate stall for unbypassable value from prior instructions to store value (not store address). ☑ Consider store size and any load size. (See examples on previous page.)




Solution appears above in orange.

Problem 2: [20 pts] Answer the following questions about two versions of our bypassed MIPS implementation.



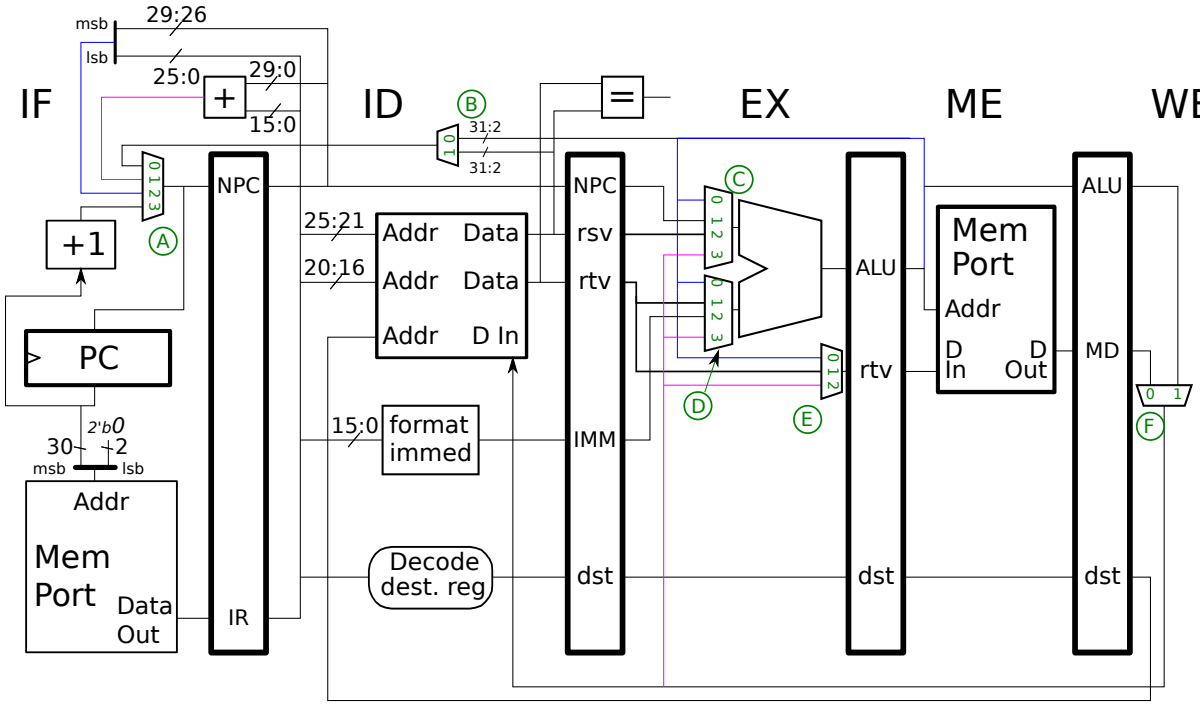

(*a*) Show the execution of the code below on the implementation illustrated above when the branch is taken.

☑ Show Execution.

☑ Check the code for dependencies.

```
# SOLUTION
# Cycles          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
lw r2, 0(r4)      IF ID EX ME WB
addi r1, r2, 3       IF ID -> EX ME WB
bne  r1, r3  TARG       IF -> ID ----> EX ME WB
ori r1, r9, 10                IF ----> ID EX ME WB
andi r11, r1, 14
# Cycles          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
TARG:
xor r20, r11, r1                       IF ID EX ME WB
```

(*b*) Each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.




☑ Use C0 to carry `r1`, elsewhere `r1` not used.

```
# Cycle          0  1  2  3  4  5   SAMPLE SOLUTION
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☑ Use D3 to carry `r1`, elsewhere `r1` not used.

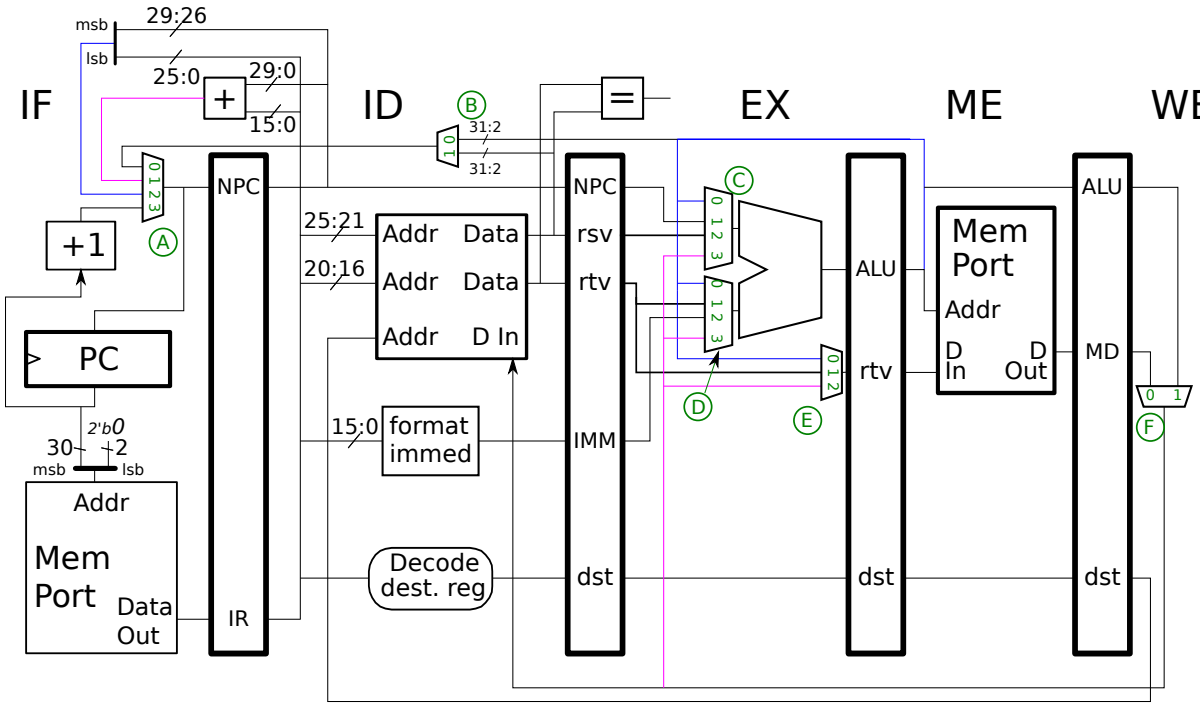
Solution appears below. Note that the D mux is used to bypass an `rt` value and so `r1` had to be the `rt` source of the `or` instruction. In other words `or r7, r1, r8` WOULD BE WRONG.

```
# Cycle          0  1  2  3  4  5  6    SOLUTION - D3 used in cycle 4.
add r1, r2, r3   IF ID EX ME WB
sub r4, r5, r6      IF ID EX ME WB
or  r7, r8, r1         IF ID EX ME WB
```

☑ Use E0 to carry `r1`, elsewhere `r1` not used.

Solution appears below. Note that the E mux is only used by store instructions, and that `rtv` is the store value, not the address.

```
# Cycle          0  1  2  3  4  5       SOLUTION - E0 used in cycle 3.
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r2)        IF ID EX ME WB
```




☑ Use A1.

Solution appears below. Only a branch uses the ID-stage adder and so the instruction must be some kind of a branch.

```
# Cycle           0  1  2  3  4      SOLUTION - A1 used in cycle 1.
beq r0, r0, TARG  IF ID EX ME WB
```

☑ Use B0.

Solution appears below. B1 carries the `rsv` to the PC, and so this could only be used by a `jr` or `jalr` instruction. B0 carries a value from the ALU to the PC, which could only reasonably be a bypassed value needed by a `jr` or `jalr` instruction. The existing connections to the ALU would allow it to compute the branch target, but there's already an ID-stage adder and so there is no need to do so.

```
# Cycle         0  1  2  3  4  5  6    SOLUTION - B0 used in cycle 3.
addi r1, r1, 4  IF ID EX ME WB
sub r2, r3, r4     IF ID EX ME WB
jr r1                 IF ID EX ME WB
```

☑ Use C1.

Solution appears below. The only instructions that would save the NPC would be a `jal` or a `jalr`. (There is already an ID-stage adder for the branch, so a branch would not use NPC.)

```
# Cycle          0  1  2  3  4        SOLUTION - C1 used in cycle 2.
jal SOMEWHERE    IF ID EX ME WB
```

Problem 3: [15 pts] The floating point code fragment below computes `f2 = f3 * f0 + f1`, where `f3` is a call argument and `f0` and `f1` are loaded from a table. A total of eight instructions are used to load the constants into `f0` and `f1`. Re-write the code so that fewer instructions are used. It is possible to load both registers using a total of two instructions.

☑ Load constants into `f0` and `f1` using two or three instructions.

☑ The constants must be loaded from the table.

```
        .data
famous_constants: # 0x10010300
        .float 2.718282818
        .float 3.141592654
        .text
pie:
        # Load f0 with first element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x300
        lw $t1, 0($t0)
        mtc1 $t1, $f0

        # Load f1 with second element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x304
        lw $t1, 0($t0)
        mtc1 $t1, $f1

        # Don't modify the code below this line.
        mul.s $f2, $f3, $f0
        add.s $f2, $f2, $f1
```

Solution on next page.

Two solutions appear below, A and B. In both solutions `mtc1` instructions are avoided by loading directly into floating-point registers. The `ori` instructions are avoided by using the load offset to add on the lower 16 bits of the famous constants table address. In Solution A a `ldc1` instruction is used to load both of the floating-point registers with just one instruction. Solution B uses two instructions.

*Grading Note:* The idea of using a `ldc1` to load both registers was from a student's solution, my best solution was the three-instruction version.

```
        .data
famous_constants: # 0x10010300
        .float 2.718282818
        .float 3.141592654
        .text
pie:
        # SOLUTION A -- Two Instructions
        lui $t0, 0x1001
        ldc1 $f0, 0x300($t0)   # Note: loads both f0 and f1.


        # SOLUTION B -- Three Instructions
        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)
        lwc1 $f1, 0x304($t0)


        # Don't modify the code below this line.
        mul.s $f2, $f3, $f0
        add.s $f2, $f2, $f1
```

Problem 4: [40 pts] Answer each question below.

(*a*) In class we said that MIPS-I lacks an instruction like `bgt` (branch greater-than) because the magnitude comparison would take a little too long. MIPS-I does have `beq` and `bne` instructions that compare two registers. However, SPARC v8 has a `bgt` instruction, but it is done in such a way that here is no risk of critical path impact.

☑ How is the actual SPARC `bgt` different than a hypothetical MIPS `bgt`?

The SPARC `bgt` uses the condition code register to determine if the branch should be taken, whereas the MIPS `bgt` would compare the contents of two registers.

☑ How does that difference avoid critical path impact in resolve-in-ID implementations?

Short Answer: The branch condition is computed in SPARC using a 4-bit condition code register rather than retrieving and then examining $2 \times 32 = 64$ bits of the two registers, which would take longer.

Explanation: The integer condition code register, `icc`, is only four bits and so checking for a particular bit configuration won't take much time. For `bgt` the hardware would check that the `N` (negative) and `Z` (zero) bits are both zero (meaning that the last `cc` operation produced a value that was positive). Further, there is no need to retrieve `icc` from anywhere. In contrast, for an instruction like `bgt r1, r2, TARG` the contents of registers `r1` and `r2` would first have to be retrieved, which would take some time, and then a magnitude comparison would have to be made.

☑ Explain why a `bgt r1, r2, TARG` would not have a big critical path impact if it were resolved in EX.

Because the `r1` and `r2` values would be available at the beginning of the clock cycle, rather than the middle as they would be in `ID`.

(*b*) In the MIPS `add` instruction the `sa` field must have a value of zero. Consider a future version of MIPS in which the `sa` field would hold a scale factor, `s`. The result of the add would be `rsv + rtv * s`. Suppose that analysis of users' programs found that such an instruction would be **very useful** and that it could **easily be implemented in hardware**. Should the `add` be extended in that way? If not, suggest another way of providing the scaled add.

☑ Should `add` be extended to compute `rsv + rtv*s`?

No. No! Noooo!!!!!

☑ Explain a possible objection and suggest an alternative way of including the instruction.

Since the `sa` field must be zero in MIPS-I instructions `add` instructions written for MIPS-I would produce the wrong answer on implementations with the scaled add. An alternative would be to define the instruction result as `rsv + rtv * ( sa + 1 )`, so that when `sa` is zero the instruction behaves like the original `add`. The scaling feature is still available (and can reach 32 instead of 31). The assembly language format for the instruction could be defined something like `add rd, rs, rt, s` with `sa = s - 1` .

*Grading Note:* Many solutions argued that the scaled add would be cumbersome to implement. Ordinarily, that would be a good point, but the problem said to suppose that the hardware would be easy to implement. In the original exam boldface was not used for emphasis.

(*c*) The MIPS-I assembly instruction below is invalid. Explain why and replace it with one that correctly adds two double-precision values.

```
add.d $f0, $f1, $f2
```

Double precision instructions must use even numbered registers. A correct version appears below.

```
add.d $f0, $f10, $f2  # SOLUTION
```

(*d*) What is the difference between a dependency and a hazard?

☑ The difference between a dependency and a hazard is:

A dependency describes a relationship between instructions in a program, while a hazard describes a potential problem an implementation can have executing instructions.

(*e*) Identify the type of dependence between each pair of instructions below, and indicate the corresponding hazard.

☑ The type of dependence is: Output . The corresponding hazard is: WAW

```
add r1, r2, r3
sub r1, r5, r6
```

☑ The type of dependence is: True . The corresponding hazard is: RAW

```
add r1, r2, r3
sub r4, r1, r6
```

Note: The dependency above is known by three names: true, data, and flow. Full credit would be given for any of those names.

☑ The type of dependence is: Anti . The corresponding hazard is: WAR

```
add r1, r4, r3
sub r4, r2, r6
```

(*f*) In class we said that the lifetime of an ISA can be decades and so it must be carefully designed to take into account current and future implementation technologies. Is IA-32 (80x86) a good example of this rule? Explain.

☑ IA-32 ◯ *is* ⊗ *is not* a good example of this rule because ….

Short Answer: … because despite being designed to fit on small chips of the time, it has been implemented in successively larger chips, and done so in a way that overcomes shortcomings (such as a limited number of registers and segmented addressing) to the point where IA-32 implementations were faster than implementations of much well-designed RISC ISAs. If the rule were always correct IA-32 implementations would be slower.

Details: The earliest ISA that could be called IA-32, implemented by the Intel 8086 and less costly 8088, was limited by the number of transistors that could fit on chips of the time. The ISA was never designed to last decades and to host a robust operating system. It was IBM's second choice for CPU in their initial entry into the personal computer market, the IBM PC. Their first choice was the Motorola 68000, but Motorola could not make enough of them for IBM's initial order and so the 8088 it was. The IBM PC was spectacularly successful, spawning a large market for IBM-compatible (and so 8088-compatible) software. Switching ISAs in the IBM PC, say to 68000 (which had problems of its own), would risk alienating customers, so IBM continued using the 8088 and its descendants. Intel deftly made improved implementations of IA-32 (or pre IA-32), overcoming its various weaknesses. Early RISC implementations would easily outperform IA-32 implementations, but some time in the aughts IA-32 implementations took the top spot, at least in the SPECcpu benchmarks. The ISA too was extended from the original 8088, to the first "real" IA-32, implemented in the 80386, up until the present. Currently IA-32 is sort of a subset of Intel 64. Adding features to an ISA does not change what's already there and that just made implementations that much harder. Many Intel 64 implementations work by *cracking* Intel 64 (or IA-32) instructions into *micro-ops*. Micro-ops are kind of like RISC instructions, and they are executed by something kind of like a RISC implementation. RISC implementations don't have to crack instructions, making implementations less expensive and easier to design.

IA-32 was not designed to last decades. According to the rule it should not have been a long-term success because future implementations would be hobbled by shortsighted ISA features. But that didn't happen and so IA-32 is not a good example of the rule. (Future implementations weren't hobbled because Intel could afford to put a large number of engineers on the design team to work around the ISA shortcomings.)

(*g*) Indicate the most appropriate ISA family for each ISA below.

☑ Intel 64 is considered ◯ *RISC* ⊗ *CISC* ◯ *VLIW*

☑ Itanium is considered ◯ *RISC* ◯ *CISC* ⊗ *VLIW*

☑ MIPS is considered ⊗ *RISC* ◯ *CISC* ◯ *VLIW*

☑ SPARC is considered ⊗ *RISC* ◯ *CISC* ◯ *VLIW*

☑ VAX is considered ◯ *RISC* ⊗ *CISC* ◯ *VLIW*