Name Solution

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Tuesday, 27 April 2021 to Friday, 30 April 2021 16:00 (4 PM) CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. For example, don't Web-search the text of a problem unless the problem specifically allows it. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman. **Suspected violation of these rules will be reported to the Dean of Students as a violation of the Student Code of Conduct.**

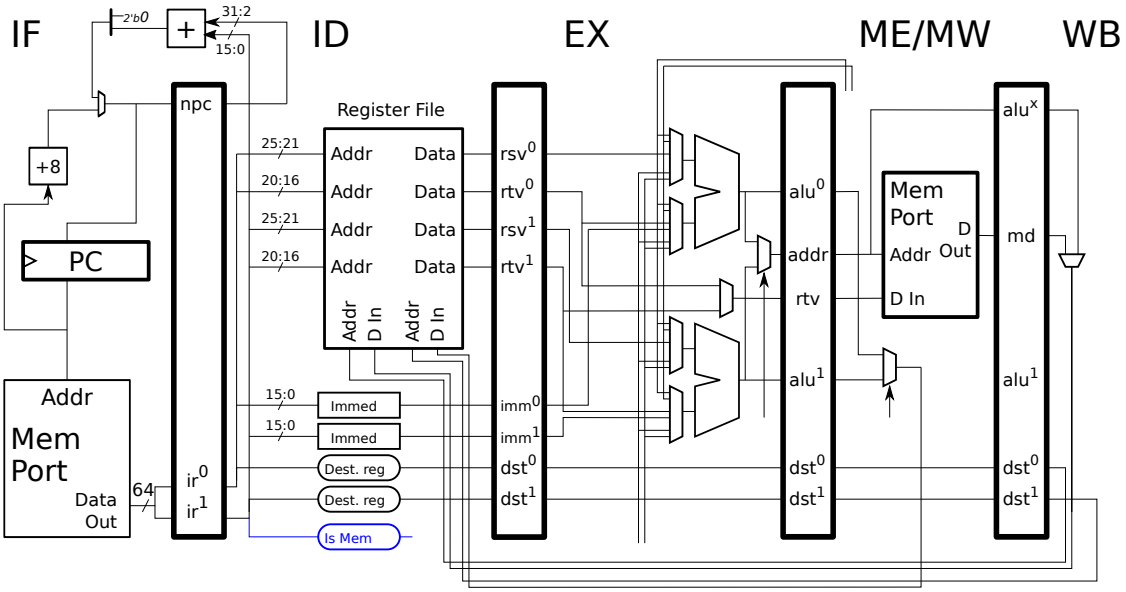
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \;\wedge\; r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Thank you for your effort in EE 4720!

Problem 1: (25 pts) Appearing below (and larger a few pages ahead) is an idea for a 2-way superscalar MIPS implementation with an unorthodox feature: The register file is written both by an instruction in the `ME/MW` stage (the former `ME` stage) and by an instruction in the `WB` stage. A memory instruction would have to write back in the `WB` stage, but a non-memory instruction could write back in either stage. Consider the execution below. For the first pair, `lw` and `addi`, the `lw` must use `WB` and the `addi` must use `MW`. The situation is similar for the second pair except that the memory instruction, `lb`, is in slot 1 rather than slot 0. When neither slot holds a memory instruction (the pair fetched in cycle 2) either one (but not both!) could use `MW`. If both use the memory port, the later one should stall. In execution diagrams label `MW` is used by an instruction that writes back in that stage, and `ME` is used by an instruction that will write back in the `WB` stage.

Put solution on diagram several pages ahead!



Put solution on diagram several pages ahead!

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID EX ME WB
addi r2, r2, 4   IF ID EX MW
sub r3, r4, r5      IF ID EX MW
lb r7, 5(r2)        IF ID EX ME WB
and r9, r10, r11       IF ID EX ME WB
or r12, r13, r14       IF ID EX MW
lb r16, 0(r7)             IF ID EX ME WB
lh r17, 2(r7)             IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

This idea has a potential cost benefit, but it must be thought through because the order in which registers are written can vary. (Which is a scary thing to those worried about correctness.) One cost benefit can be seen in the diagram. The `WB.alu1` pipeline latch is no longer needed. The label is still there but it is not connected (and won't be). Other cost-saving changes are part of the subproblems below.

☑ For all parts of this problem remember to pay attention to cost and performance. ☑ For example, don't connect a bypass path that will never be needed.

(*a*) The `D In` connections to the write ports of the register file have been changed, but the register number inputs, `Addr`, are the same. Modify the hardware so that the `Addr` inputs to the write ports get the correct register number. For this part don't worry about two instructions writing the same register number.

☑ Modify hardware (next page, not above) so that `Addr` inputs of the register file write ports get the correct register number.

Solution shown in green. If `mem1` is true then the slot 0 instruction writes in `MW` and the slot 1 instruction writes in `WB`. New multiplexors wrote the signals appropriately. Note that since `dst1` is not needed in `WB` it is not sent to `WB`.

(*b*) The diagram shows one cost savings: the `WB.alu1` pipeline latch is no longer needed. Notice that the four bypass connections to the `EX` stage are unconnected. Reconnect them as needed so that any dependency that could be bypassed in the unmodified superscalar can be bypassed here. Leave a bypass unconnected if not needed.

☑ Re-connect bypass paths (on next page, not above), possibly adding or modifying other hardware to bypass values.

Solution shown in purple. Note that because only one value is written back in `WB` only one bypass path is needed from `WB`. The bypass values from `ME` are taken from the pipeline latches, `ME.alu0` and `ME.alu1`, which is good because their values are available at the beginning of the clock cycle.

(*c*) Notice that there is an unconnected select signal in `EX` and `MW`. Design control logic for these and for any multiplexors used to provide the correct register numbers (the first subpart above). The Is Mem logic in `ID` should be helpful. *Hint: There is not that much to do for this part. The Is Mem block should come in handy.*

☑ Connect select signals in `EX` and `MW`.

☑ Connect select signals for any multiplexors used for register numbers.

Solution appears in turquoise. Fortunately, that Is Mem logic block provides the exact signal needed for the select inputs.

Grading Note: some solutions, by also considering the case in which a memory instruction is in slot 0, were more complicated than they needed to be. There is no need to consider whether a memory instruction is in slot 0 because if there is not a memory instruction in slot 1 the execution will be correct whether or not there is a memory instruction in slot 0. (And what if there's a memory instruction in both slots? Should I answer that now (11 May 2021, 17:50:37 CDT) or should I make this a problem on the 2022 final exam?)

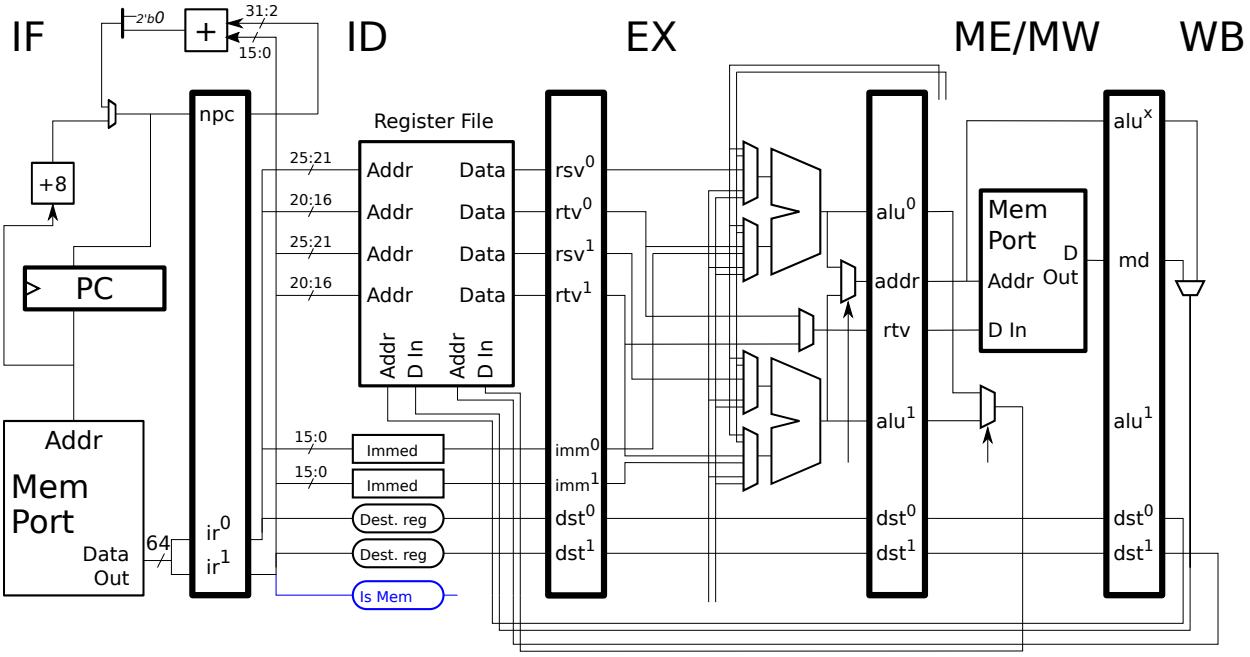
(*d*) Notice that in the execution below the `and` writes `r9` after the `or`. It looks like the `add` instruction will get the value written by the `and` rather than the `or`. That's not right!

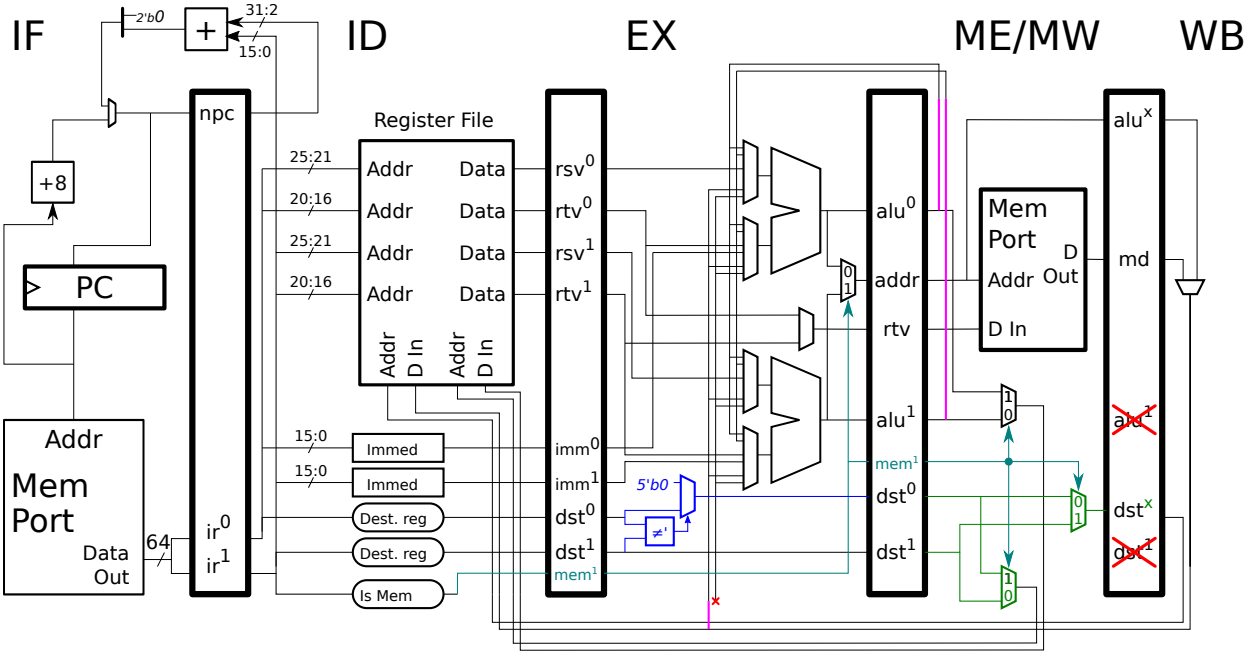
```
# Cycle            0  1  2  3  4  5 ... 9  10 11 12 13
and r9, r10, r11   IF ID EX ME WB
or r9, r7, r14     IF ID EX MW
# .. later..
add r1, r1, r9                          IF ID EX MW WB
```

☑ Add control logic to detect such WAW hazards and which will substitute `r0` for the destination of the earlier instruction.

Solution shown in blue. If both registers are the same the slot-0 register is changed to zero. The comparison unit is put in the **EX** stage to provide more time. (Okay, I'll be honest, I didn't want to squeeze it in.)

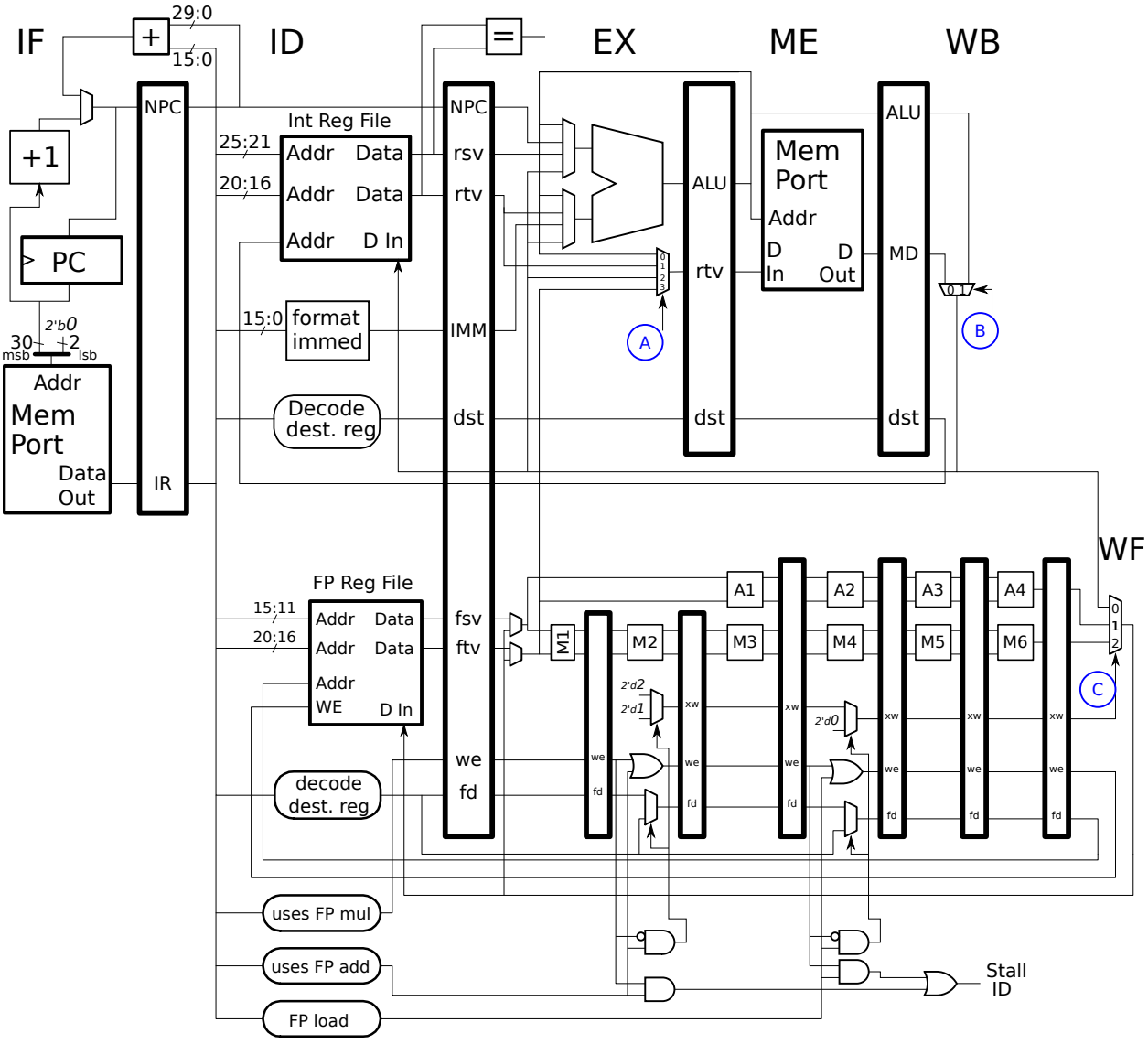
Use your favorite SVG or plain text editor on the SVG source for the implementation `https://www.ece.lsu.edu/ee4720/2021/fe-ss-px.svg` and logic gates `https://www.ece.lsu.edu/ee4720/2021/g.svg`.





Problem 2: (25 pts) Show the execution of the code fragments as requested below.

(*a*) The MIPS implementation below is similar to the one used in class, but with some added bypass paths for FP instructions and some labeled multiplexor select signals.



Appearing on the next page is a code fragment, and above the code fragment are the labels `A`, `B`, and `C`. These labels correspond to those used in the implementation.

Show the execution of the code below on this pipeline long enough to determine the IPC for a long number of iterations. Of course, that means the branch is taken. Show the value of each labeled select signal in those cycles it is being used.

☑ Show the execution on the illustrated implementation.

Solution appears below. The `add.s` stalls one cycle waiting for the loaded value. The `swc1` stalls until the `add.s`, writing `f3`, reaches `WF` and then bypasses the value to the input of `ME.rtv`.

The following reminders are based on common mistakes made on the exam: Don't forget that since MIPS has delay slots the `addi` executes. Also don't forget that since the branch is taken the `xor` should not be executed. Based on the diagram the branch resolves in `ID`, meaning that in the next cycle, when the branch is in `EX`, the target will be in `IF`. That the branch resolves in `ID` can be seen by noting that the branch target input to the `PC` mux originates from the `ID` stage.

☑ Show the values of the labeled select signals, `A`, `B`, and `C` when they are in use.

The values are also shown below. Those values are only shown for cycle in which they are needed.

Signal `A` routes the correct store value for a store instruction in `EX`. In the execution below `swc1` is in `EX` in cycle 9 and its store value, specified by `f3`, is being written by the `add.s` which is in `WF`. Input 3 connects to the value in `WF` (admittedly taking a long path). `A` is only used in cycle 9 (below) and so a blank is shown for the other cycles. In real life `A` can have any value at those other cycles and execution would still be correct.

Signal `B` selects the value from the integer pipeline that will proceed to `WB` or `WF`. In cycles 4 and 5 (and again in 14) when the two loads write back `B` is zero to select the output of the memory port. In cycle 13, when `addi` is in `WB`, `B` is 1 to select the output of the ALU. Nothing is written back in the other cycles and so they are shown blank. (The `swc1` and `bne` reach the `WB` stage but they don't write a register.)

Signal `C` selects the value to be written to the FP register file. In cycles 4, 5, and 14, the value is 0 so that the load value has a path from the integer pipeline. In cycle 9 the `add.s` reaches `WF`, and the value of `C` is 1 to select the output of the `A4` functional unit.

☑ Find the IPC for a large number of iterations.

The instruction throughput, or IPC, is computed by dividing the number of instructions by the number of cycles. The number of instructions is 6. The number of cycles is based on the fetch of the first instruction of the loop (the `lwc1 f1`). That's at cycles 0 and 10, so the number of cycles for an iteration is 10. The instruction throughput is then $\boxed{\frac{6}{10-0} = 0.6\,\text{insn/cycle}}$.

Grading Note: In many—too many—answers the number of cycles is computed incorrectly. Just remember to use a common reference point on the two iterations, usually the fetch of the first instruction. This way the time for one iteration does not overlap the time for the next.

```
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14    SOLUTION
 A                                                3
 B                              0  0                        1  0
 C                              0  0              1            0
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f1, 0(r1)     IF ID EX ME WF                                 # 1st Iter
 lwc1 f2, 4(r1)        IF ID EX ME WF
 add.s f3, f1, f2         IF ID -> A1 A2 A3 A4 WF
 swc1 f3, 8(r1)              IF -> ID -------> EX ME WB
 bne r1, r2  LOOP                  IF -------> ID EX ME WB
 addi r1, r1, 12                               IF ID EX ME WB
 xor r5, r1, r6
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f1, 0(r1)                                   IF ID EX ME WF  # 2nd Iter
```

(*b*) There should have been stalls in the execution of the code above. Re-write the code so that it executes with as few stalls as possible and still computes the same result. It is okay to add extra instructions before and after the loop. For convenience assume that the code executes for at least two iterations. But don't unroll the loop.

☑ Re-write code to avoid stalls. ☑ Code must compute the same values. ☑ Can re-arrange instructions, change registers, and put instructions before the start of the loop.

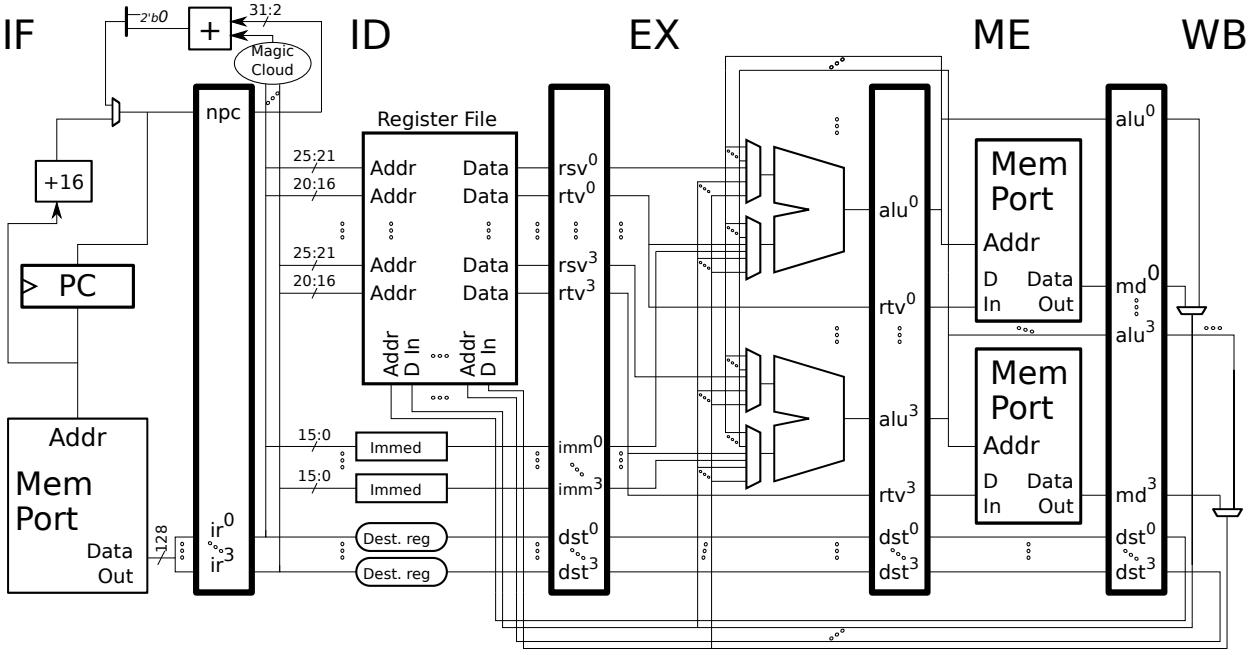
The solution appears below. The code is shown, followed by the execution of the code shown in dynamic instruction order.

To avoid the `swc1` stall the `swc1` stores the value computed in the *previous iteration* (or by the prologue code in the first iteration). The `swc1` is put between the second `lwc1` and the `add.s` eliminating the load/use stall. Also note that the offsets on the stores have changed because they are loading a value that will be needed for the next iteration. The first iteration has stalls, but that's due to the prologue code. The second and subsequent iterations execute stall-free.

```
 # Solution Code
 #
 lwc1 f1, 0(r1)
 lwc1 f2, 4(r1)
 add.s f3, f1, f2
LOOP:
 lwc1 f1, 12(r1)
 lwc1 f2, 16(r1)
 swc1 f3, 8(r1)
 add.s f3, f1, f2
 bne r1, r2  LOOP
 addi r1, r1, 12
 xor r5, r1, r6

 # Solution Code Execution (Shown in dynamic instruction order.)
 #
 lwc1 f1, 0(r1)      IF ID EX ME WF
 lwc1 f2, 4(r1)         IF ID EX ME WF
 add.s f3, f1, f2          IF ID -> A1 A2 A3 A4 WF
 # Loop
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lwc1 f1, 12(r1)            IF -> ID EX ME WF
 lwc1 f2, 16(r1)                  IF ID -> EX ME WF
 swc1 f3, 8(r1)                      IF -> ID EX ME WB
 add.s f3, f1, f2                          IF ID A1 A2 A3 A4 WF
 bne r1, r2  LOOP                             IF ID EX ME WB
 addi r1, r1, 12                                 IF ID EX ME WB
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
 lwc1 f1, 0(r1)                                     IF ID EX ME WF
 lwc1 f2, 4(r1)                                        IF ID EX ME WF
 swc1 f3, 8(r1)                                           IF ID EX ME WB
 add.s f3, f1, f2                                            IF ID A1 A2 A3 A4 WF
 bne r1, r2  LOOP                                               IF ID EX ME WB
 addi r1, r1, 12                                                   IF ID EX ME WB
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
 lwc1 f1, 0(r1)                                                       IF ID EX ME WF
```

(*c*) Appearing below is a 4-way superscalar MIPS implementation. In this implementation fetch is not aligned (which makes things easier). Also, there is no branch prediction, which is how we have been doing things in class.



☑ Show the execution of the code below for enough iterations to determine IPC. (Note: There is no need to put slot numbers on the stage labels.) ☑ Don't forget that it is 4-way superscalar.

Solution appears below. To keep instructions in order in `ID`, the stall of one instruction in `ID` stalls all instructions ahead in `ID`. So, in cycle 1 and 2 the `add` is stalling for the `lw` value. That forces the `sw` and `addi` to stall to, even though the `bne` and `addi` are not waiting for anything.

The branch resolves in `ID` and so the target is not fetched until the branch is in `EX`, resulting in the squash of six instructions. Just because it would be nice to fetch the branch target in cycle 7, does not mean the hardware can actually do so.

The instruction throughput is $\frac{6}{8-0} = .75$ insn/cycle.

```
 # SOLUTION
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lw r10, 0(r1)       IF ID EX ME WB
 add r3, r10, r3     IF ID ----> EX ME WB
 sw r3, 0(r5)        IF ID -------------> EX ME WB
 addi r5, r5, 4      IF ID -------------> EX ME WB
 bne r1, r9, LOOP       IF -------------> ID EX ME WB
 addi r1, r1, 4         IF -------------> ID EX ME WB
 lb r8, 0(r9)           IF -------------> IDx
 xor r11, r8, r10       IF -------------> IDx
 some insn                                IFx
 some insn                                IFx
 some insn                                IFx
 some insn                                IFx
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lw r10, 0(r1)                               IF ID EX ME WB
 add r3, r10, r3                             IF ID ----> EX ME WB
```

(*d*) The code fragment below is to execute on the same 4-way superscalar MIPS implementation. Notice that loop body in the code fragment below contains two copies of the loop body from the loop in the previous subproblem. So one iteration of the loop below does the work of two iterations of the loop from the previous problem. This is the first step in the application of a technique called *loop unrolling*. The loop from the previous problem has been unrolled by *degree 2*. The next step is to re-arrange the instructions, and possibly eliminate those that are no longer needed. Complete this step, of course without changing what the code does. Finally, to eliminate all stalls *software pipelining* will need to be used: values computed used in one iteration will have to come from instructions executed in the prior iteration.

☑ Re-write the loop above so that it runs more efficiently on the 4-way MIPS implementation. ☑ Use fewer instructions, even if doing so does not help with the degree 2 unroll, in the expectation that it might be beneficial at higher unroll degrees.

The solution appears below. A single load/add/store calculation is spread across three iterations. For example, suppose `lw r10` loads something in iteration x. The `add r19` and `add r3` instructions will operate on that `r10` in iteration $x+1$ and the value computed by the `add r3` and the `add r13` instructions in iteration $x+1$ will be written by the `sw` instructions in iteration $x+2$. This technique is called *software pipelining*. The prologue code starts up the process. As can be seen from the execution, this avoids all stalls.

An iteration takes just 4 cycles now, compared to 8 cycles in the original code. Also, each iteration computes twice as many values, and so the code runs 4 times faster than the original.

```
 # SOLUTION - Code
 #
 # Prologue
 lw r10, 0(r1)
 lw r12, 4(r1)
 add r3, r10, r3
 add r13, r12, r3
 lw r10, 8(r1)
 lw r12, 12(r1)
 #
 # Main Loop
LOOP:
 add r19, r10, r12
 sw r3, 0(r5)
 sw r13, 4(r5)
 add r3, r13, r10
 lw r10, 16(r1)
 lw r12, 20(r1)
 add r13, r13, r19
 addi r5, r5, 8
 bne r1, r9, LOOP
 addi r1, r1, 8
 #
 # Post-Loop Code
 lb r8, 0(r9)
 xor r11, r8, r10
```

```
 # Solution Code Execution (Omitting prologue)
 #
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
 add r19, r10, r12    IF ID EX ME WB
 sw r3, 0(r5)         IF ID EX ME WB
 sw r13, 4(r5)        IF ID EX ME WB
 add r3, r13, r10     IF ID EX ME WB
 lw r10, 16(r1)          IF ID EX ME WB
 lw r12, 20(r1)          IF ID EX ME WB
 add r13, r13, r19       IF ID EX ME WB
 addi r5, r5, 8          IF ID EX ME WB
 bne r1, r9, LOOP           IF ID EX ME WB
 addi r1, r1, 8             IF ID EX ME WB
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
 add r19, r10, r12                IF ID EX ME WB
 sw r3, 0(r5)                     IF ID EX ME WB
 sw r13, 4(r5)                    IF ID EX ME WB
 add r3, r13, r10                 IF ID EX ME WB
 lw r10, 0(r1)                       IF ID EX ME WB
 lw r12, 4(r1)                       IF ID EX ME WB
 add r13, r13, r19                   IF ID EX ME WB
 addi r5, r5, 8                      IF ID EX ME WB
 bne r1, r9, LOOP                       IF ID EX ME WB
 addi r1, r1, 8                         IF ID EX ME WB
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: (25 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with a 10-outcome local history.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  T  T  N  T  N    T  T  T  N  T  N    T  T  T  N  T  N   <- Outcome
     1  2  3  4  5  6    1  2  3  4  5  6    1  2  3  4  5  6   <- Outcome Pos.

BA:

B2:   N  N  N ...  N  N  T  T  T    N  N  N ...  N  N  T  T  T
      1  2  3     11 12 13 14 15    1  2  3     11 12 13 14 15  <- Outcome Pos.
```

☑ What is the accuracy of the bimodal predictor on branch B1?

The accuracy is $\boxed{\frac{4}{6}}$. The work is shown below. The accuracy is based on the repeating pattern, shown underlined below.

```
   SOLUTION WORK
   0   1  2  3  2  3   2   3  3  3  2  3   2   3  3  3  2  3    <- 2b Counter
B1:  T  T  T  N  T  N    T  T  T  N  T  N    T  T  T  N  T  N   <- Branch Outcome
     x  x     x     x            x     x            x     x   <- Pred. Outcome
                     --------------------                     <- Repeat
```

☑ What is the accuracy of the local predictor on B1 ignoring B2.

The 10-outcome history is longer than the period of B1, six outcomes, and so the accuracy is 100%.

☑ What is the accuracy of the local predictor on B2 ignoring B1.

In B2 there are twelve consecutive Ns, and so a local history of 10 Ns, that is, NNNNNNNNNN, will occur three times: when predicting positions 11, 12 and 13. The outcome of positions 11 and 12 will be N, decrementing the counter, and the outcome at 13 will be T, incrementing the counter. After warmup the counter will be 0 or 1 and so position 13 will be mispredicted. The overall accuracy will be $\frac{14}{15}$. *B2 in the original exam had 11 consecutive* N*s, and in that case the accuracy would likely be* $\frac{13}{14}$ *but in the worse case could be* $\frac{12}{14}$.

☑ What is the longest local history size for which branch B1 and branch B2 will interfere with each other on the local predictor? (The question is for a local predictor, not a global predictor.)

Five outcomes. Local history NTTTN precedes a T in B1 and a N in B2. Any B1 local history of length 6 would have to contain at least four Ts, no local history of B2 can contain four Ts, so there can be no interference with a six outcome local history.

(*b*) If branch B1 at position 6 is mispredicted `T` (taken) then data at address $a + x$ will be brought into the cache. An adversary would like to learn the value of x. To do so the adversary, running in another process on the same core as the process using B1, will force a misprediction of 6. Then the adversary will make careful timing measurements of loads to addresses starting at a to determine what address was loaded, and so learning x.

Branch BA is part of the adversary's code. Find a pattern for BA that will force B1 to be mispredicted by the 10-outcome local predictor at position 6 (but not at other positions). The misprediction does not need to occur every time position 6 in B1 is executed.

☑ Pattern for BA that forces misprediction of B1 at position 6 on 10-outcome local predictor.

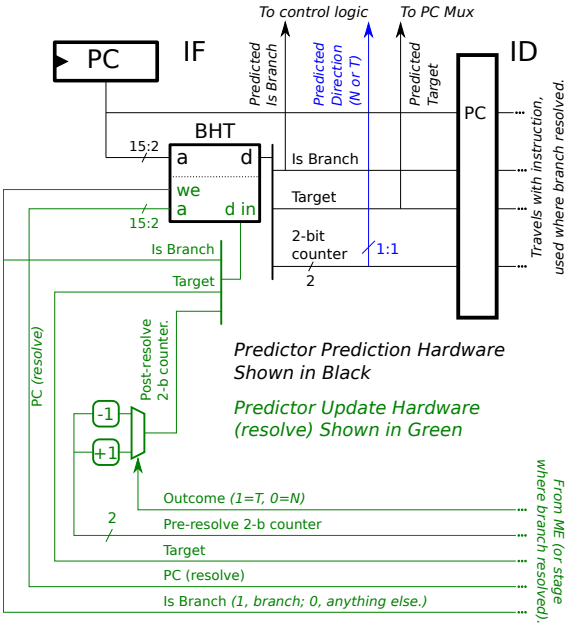
For B1 outcome 6 to be mispredicted `T` branch B2 must have a matching local history that precedes an outcome of `T`, and that pattern must occur more frequently. The B1 local history when predicting outcome 6 is `TTNTN TTTNT` (the space is for readability). One choice for BA would be the pattern repeating `TTNTN TTTNT T`, but that can be shortened to `NTN TTTNT T`. To induce a misprediction BA must occur more frequently than B1. In the timing shown below two repeats of the BA pattern occur between each pattern of B1. (There is no reason why the branch outcomes of B1 and BA needed to be neatly interleaved with each other.)

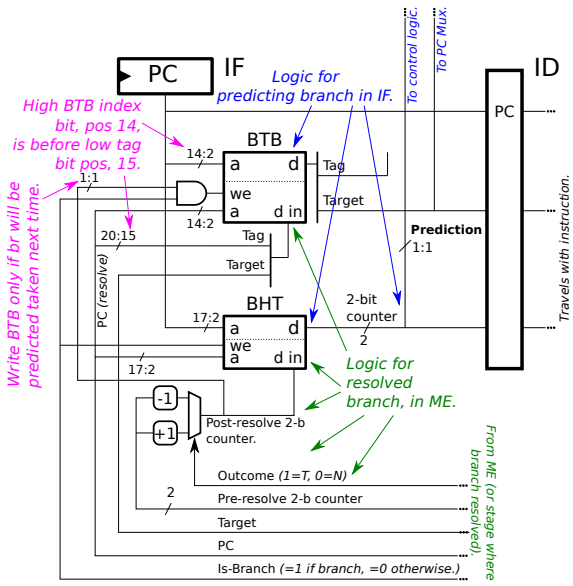
```
B1:  T  T  T  N  T  N                        T  T  T  N  T  N   <- Outcome
     1  2  3  4  5  6                        1  2  3  4  5  6   <- Outcome Pos.

BA:                     NTNTTTNTTNTNTTTNTT                     NTNTTTNTTNTNTTTNTT

     Outcomes of BA shown below for clarity, shown above with intended timing.
BA:  N  T  N  T  T  T  N  T  T    N  T  N  T  T  T  N  T  T
     1  2  3  4  5  6  7  8  9    1  2  3  4  5  6  7  8  9
                         -------------------------------     <- Local History
```

(*c*) Below on the left is a plain bimodal predictor as first described in class. Below on the right is a refined version that makes better use of storage by splitting the BHT into two tables, a *branch target buffer*, holding the target and a *tag*, and a BHT holding only the two-bit counters. The tag field replaces the `IsBranch` field and is used like a cache tag. (Though a cache tag would include bits `31:15`.) The control logic will compare the tag retrieved from the BTB with bits `20:15` of the PC, if they match a prediction will be made using the 2-bit counter from the BHT, if the tag doesn't match it is assumed that the instruction is not a branch. See 2017 Homework 8 Problem 3 for additional explanation.





The memory address and behavior of two branches from two programs, A and B, are shown below. One program runs better on the plain predictor than the BTB predictor, the other program runs better on the BTB predictor. Identify which is better, and why, as requested below.

```
# Program A
B1: 0x9000:  T T T ...
B2: 0x1000:   T T T ...
```

☑ Prediction of branches in A better on ⊗ *Plain* or ◯ *BTB* predictor. ☑ Explain why other predictor does worse on this program. ☑ Pay attention to address of branches.

For Program A the two branches, B1 and B2, differ in bit position 15 and so they would use different entries in the plain predictor's BHT. For that reason the plain predictor would correctly predict them.

On the BTB predictor the same BTB entry would be used for B1 and B2 since their addresses are identical at bit positions 14:2. So, when predicting B2 the BTB would return a tag for B1. Because the tag would not match no prediction would be made. (And, if a prediction were made the target address would be wrong since it's the target of B1.) Something similar happens when B1 is predicted.

```
# Program B
B3: 0x11000:  N N N T  N N N T  N N N T    ...
B4: 0x21000:   T T T N  T T T N  T T T N    ...
```

☑ Prediction of branches in B better on ◯ *Plain* or ⊗ *BTB* predictor. ☑ Explain why other predictor does worse on this program. ☑ Pay attention to address of branches.

The addresses of branches B3 and B4 are identical in bit positions 15:2 (and so also 14:2) but differ in bits 17:2. A prediction of B3 in the plain predictor's BHT will retrieve the 2-bit counter and target address of B4, and so there will possibly be a misprediction. B4 could similarly be mispredicted by the plain predictor. In a worst case both would be mispredicted.

In the BTB predictor branches B3 and B4 will use separate BHT entries, so their 2-bit counters will be entirely their own. The BHT is always written, but the BTB is only written if the branch would be predicted taken. For that reason only branch B4 is written to the BTB. So, when B3 arrives there will be a miss in the BTB (the tag won't match), and so the predictor will assume that the instruction being fetched is either not a branch or a not-taken branch. For B3 that will be correct three out of four times. When B4 is predicted the BTB will hit. The 2-bit counter from the BHT will be 3 or 4 and so the branch will be predicted taken. That will be correct 3 out of 4 times, and the target will also be correct.

Problem 4: (25 pts) Answer each question below.

(*a*) A program runs with fewer stalls on an 2-way superscalar than on an 8-way superscalar statically scheduled processor. Both have the same clock frequency and are otherwise comparable.

☑ Does that mean the program runs faster on the 2-way processor? ☑ Explain.

No, of course not. Consider a stall of one instruction (the **sub** in the example below) waiting for a value computed by another (the **add** in the example below). Because the pipeline is the same depth (five stages) on both the 2-way and 8-way systems (because they are otherwise comparable), the number of cycles it takes for the value to be bypassable will be the same on both. That is, on both systems the **sub** can bypass the value two cycles after the **add** is in **ID**. There is no stall in the 2-way system because the **sub** is fetched one cycle later.

```
# Execution on 2-way. No Stall!
add R1, r2, r3     IF ID EX ME WB
xor r9, r10, r11   IF ID EX ME WB
sub r4, R1, r5        IF ID EX ME WB
or r12, r13, r14      IF ID EX ME WB
# Cycle            0  1  2  3  4  5

# Execution on 4-way. No slower!!
add R1, r2, r3     IF ID EX ME WB
xor r9, r10, r11   IF ID EX ME WB
sub r4, R1, r5     IF ID -> EX ME WB
or r12, r13, r14   IF ID -> EX ME WB
```

(*b*) Shorten the code fragments below.

☑ Shorten code fragment.

```
addi r1, r0, 0x4755
sll r1, r1, 16
addi r1, r1, 0x4720
lw r1, 0(r1)
```

Solution appears below. Many forgot that they could use an offset in the load.

```
# SOLUTION
lui r1, 0x4755
lw r1, 0x4720(r1)
```

☑ Shorten code fragment.

```
lw r3, 0(r1)
addi r2, r0, 4
add r1, r1, r2
lbu r1, 0(r1)

# SOLUTION
lw r3, 0(r1)
lbu r1, 4(r1)
```

(*c*) Appearing below are three code fragments. These are to run on a machine with vector instructions and four-lane vector units. Indicate whether each fragment can easily be replaced by a vector instruction, and the reason why or why not.

☑ Fragment below ◯ *can* or ⊗ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
mul.s f7,  f8,  f9
sub.s f10, f11, f12
```

The code performs three different operations, **add**, **mul**, and **sub**. A vector instruction performs the same operation on multiple sets of operands (bit positions within vector registers).

☑ Fragment below ◯ *can* or ⊗ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s F1,  f2,  f3
add.s f4,  F1,  f6
add.s F7,  f8,  f9
add.s f10, F7, f12
```

Typical vector instructions do not support dependencies like that. Certainly not arbitrary dependencies.

☑ Fragment below ⊗ *can* or ◯ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
add.s f7,  f8,  f9
add.s f10, f11, f12
```

The code sequence can easily be replaced by a vector instruction.

(*d*) Let's break something! As we know, an implementation of an ISA must execute a program as defined in the ISA. The Broeken Company is going to sell a 4-way superscalar statically scheduled five-stage "implementation" of ARM A64, call that the Broeken4 implementation. (There is no space between the Broeken and the 4. That's so you don't notice the 4.) We know how smoothly our five-stage scalar MIPS implementation handles branches: zero penalty because of the delay slot. But A64 branches don't have a delay slot. That didn't stop the Broeken Company from changing things a bit. In the Broeken4 implementation the number of delay slots for a branch can vary from 4 to 7. If a branch is in slot 0 of its fetch group there are 7 delay slots. Branches in slots 1, 2, and 3 have 6, 5, and 4 slots, respectively. If the branch is not stalled by dependencies there will be zero branch penalty. This chip performs about 20% better than the (rule-abiding) competition on performance and energy efficiency benchmarks.

Computer engineering professors obviously will hate the Broeken company for ignoring what an implementation of an ISA should do. But what about others? Gauge the reaction of each group below.

Note: If this were an in-class exam questions would be shorter.

☑ Compiler writers will be ◯ *positive, happy, and supportive* or ⊗ *negative, irritated, and obstreperous* . ☑ Explain.

It will be difficult to fill all those delay slots. Depending on personality type, putting in `nop` instructions will make compiler writers feel like failures or make them feel like they are being set up to take the blame for an unworkable idea (the large number of delay slots).

☑ Buyers of laptops and desktops using Broeken4 will be ◯ *positive, satisfied* or ⊗ *negative, seeking a refund* . ☑ Explain.

They will be negative because most software won't run. They expected their new computer to run their old software.

☑ Engineers using Broeken4 in embedded devices, such as the controller chip in a microwave oven, will be ⊗ *positive, satisfied* or ◯ *negative, seeking a new supplier* . ☑ Explain.

Software is usually custom written for an embedded system, and so compatibility is not an issue. They will be happy because of the performance and efficiency improvements.

(*e*) The execution below is taken from an illustration of how exceptions work from a class lecture. The `lw` raises an exception in cycle 4 and in response the handler starts in cycle 5. The first instruction that the handler executes is `sw`. It is likely that the handler is saving registers to the stack before attending to the event causing the exception. Presumably the handler will save every register that it plans to modify (or to be safe, all registers). To reduce the number of registers saved, why not split the work between the interrupted code and the handler. As with the ABI rules for procedure calls, why not have the interrupted code save the caller-save registers it wants preserved (`t0-t9`, etc.) so that the handler would only need to save the callee-save registers it plans to overwrite (`s0-s7`, etc.)?

```
# Cycle:                0  1  2  3  4  5 ...          99 100 ...
        add  r1, r2, r3  IF ID EX ME WB
        lw   r6, 0(r1)      IF ID EX ME*x            IF ID EX ME WB
        or   r5, r6, r7        IF ID EXx                IF ID EX MEx
        xor  r10, r11, r12        IF IDx                   IF ID EXx
        and  r20, r21, r22           IFx                      IF IDx
...
Handler:
        sw ...                          IF ...                      IF ...
        ...
        eret (exception return)                 IF ID EX ME WB
```

☑ Why can't interrupted program save the caller-save registers before the handler starts, potentially reducing work?

Short answer: This is an absurd idea! Where would all that register save code be placed? Surely the effort needed to find it when there was an exception or to jump over it when there wasn't will be much greater than any reduction in the number of registers saved.

Explanation: For caller-save registers to be saved, routines to save those registers need to be written and put somewhere. It would be no problem for a compiler to write such code. The problem is a register-save routine would need to be written for every instruction that could raise an exception, which would be a huge number. Putting such register-save routines before each instruction would be a waste for instructions like loads, because they only raise exceptions infrequently, but the register save code would be executed each time, and for nothing. Or, some scheme could be developed to either locate a register save routine for each exception raising instruction, or to skip over such code if it were placed just before the instruction.

Also, instructions in the register save code itself could raise exceptions.