Name _____

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.**

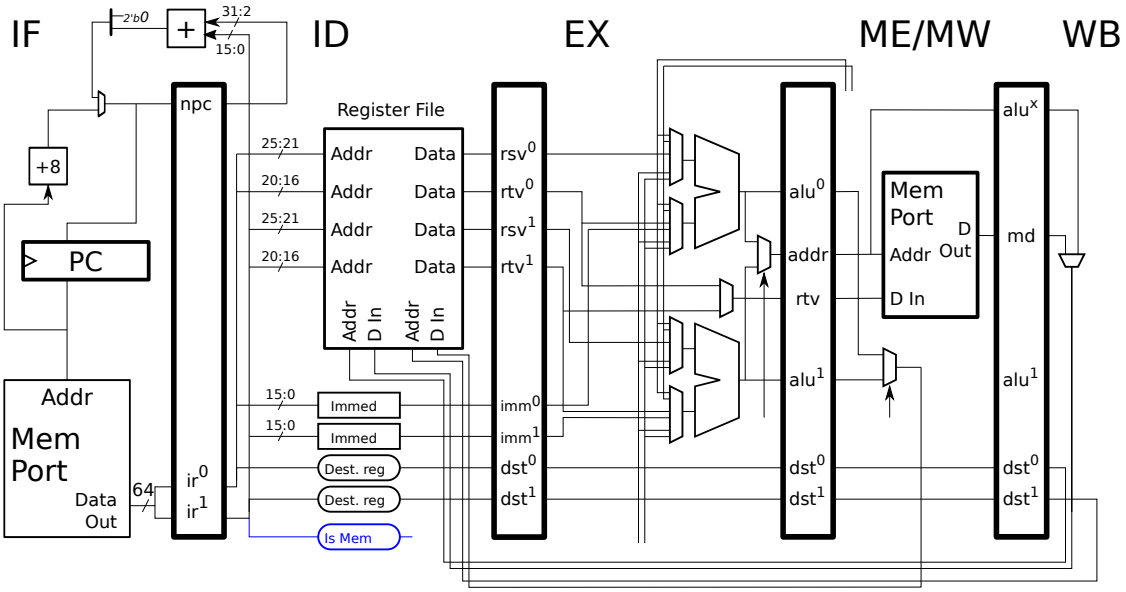
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

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

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.

(*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.

(*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`.

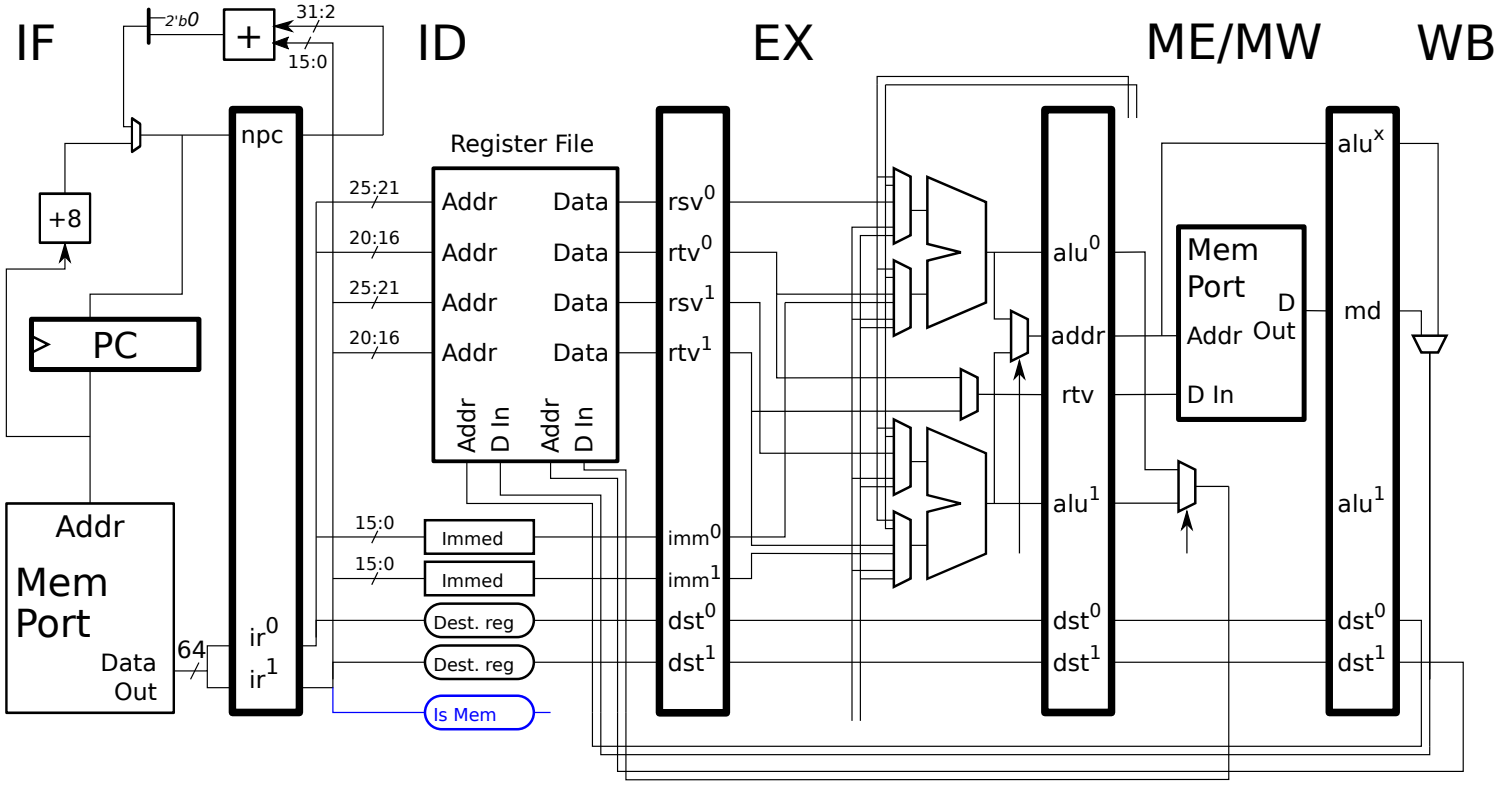
☐ Connect select signals for any multiplexors used for register numbers.

(*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
```

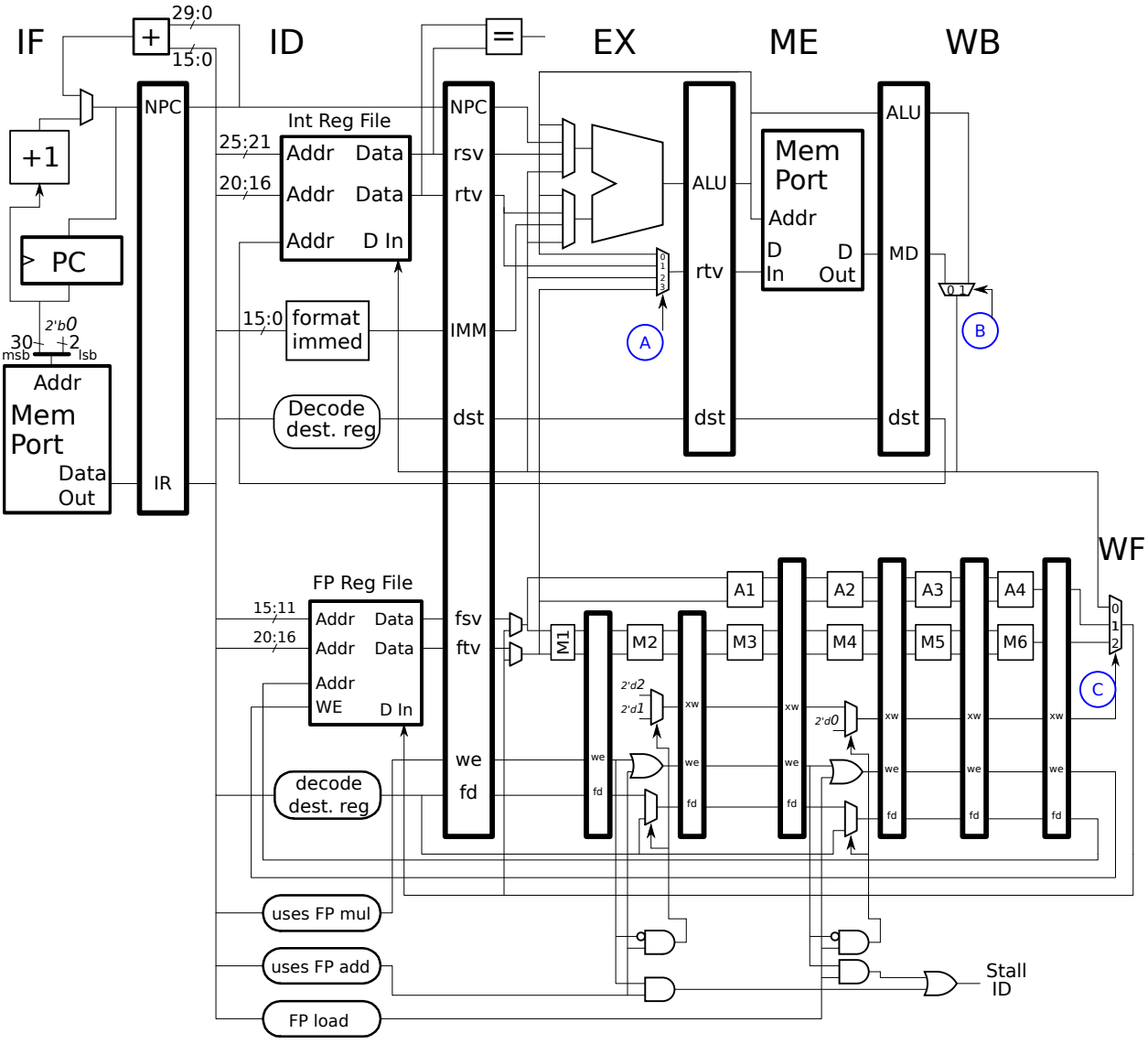
☐ Add control logic to detect such WAW hazards and which will substitute `r0` for the destination of the earlier instruction.

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.

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

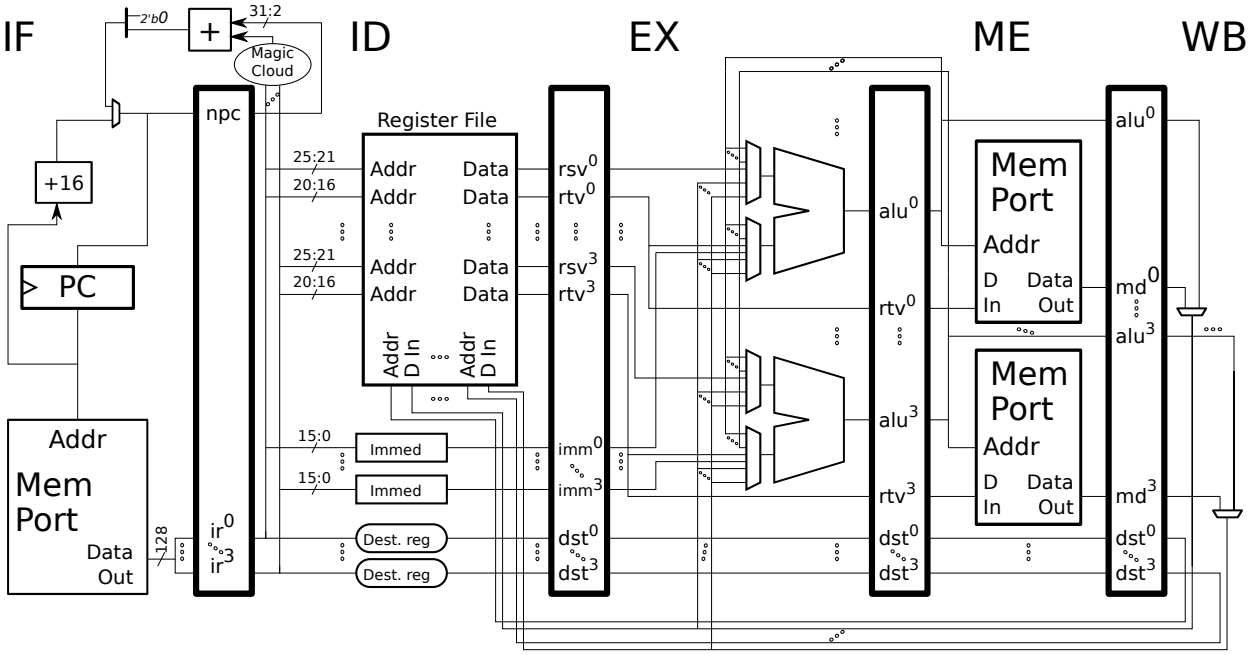
☐ Find the IPC for a large number of iterations.

```
A

B

C

LOOP:
 lwc1 f1, 0(r1)

 lwc1 f2, 4(r1)

 add.s f3, f1, f2

 swc1 f3, 8(r1)

 bne r1, r2  LOOP

 addi r1, r1, 12

 xor r5, r1, r6
```

(*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.

(*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.

LOOP:

```
lw r10, 0(r1)

add r3, r10, r3

sw r3, 0(r5)

addi r5, r5, 4

bne r1, r9, LOOP

addi r1, r1, 4

lb r8, 0(r9)

xor r11, r8, r10
```

(*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.

```
LOOP:
 lw r10, 0(r1)
 add r3, r10, r3
 sw r3, 0(r5)
 addi r5, r5, 4
 addi r1, r1, 4
 lw r10, 0(r1)
 add r3, r10, r3
 sw r3, 0(r5)
 addi r5, r5, 4
 bne r1, r9, LOOP
 addi r1, r1, 4
 lb r8, 0(r9)
 xor r11, r8, r10
```

☐ 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.

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`?

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

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

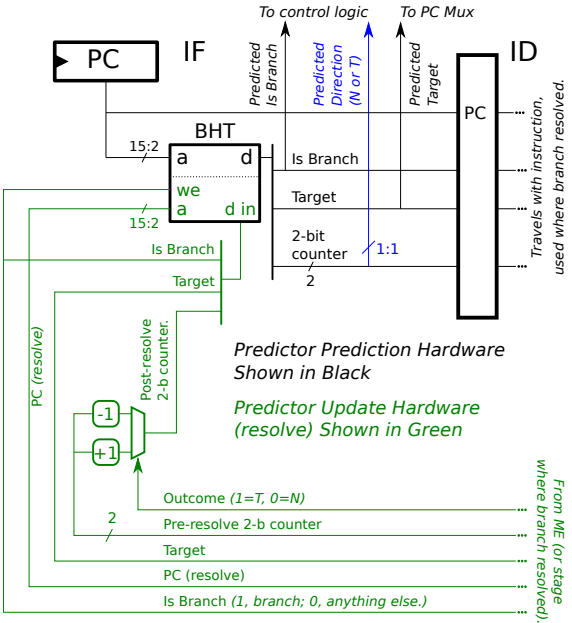
☐ What is the longest local history size for which branch B1 and branch B2 will interfere with each other on the local predictor? (The question is for a local predictor, not a global predictor.)

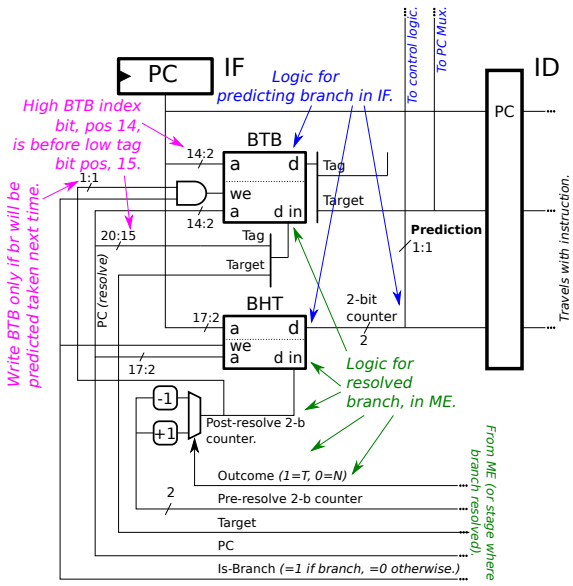
(*b*) If branch B1 at position 6 is mispredicted `T` (taken) then data at address $a + x$ will be brought into the cache. An adversary would like to learn the value of x. To do so the adversary, running in another process on the same core as the process using B1, will force a misprediction of 6. Then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

(*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.

```
# 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.

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.

(*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)
```

☐ Shorten code fragment.

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

☐ 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
```

☐ 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
```

(*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.

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

☐ 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.

(*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?