Name Solution

Computer Architecture

LSU EE 4720

Midterm Examination

Wednesday, 29 March 2023 9:30-10:20 CDT

Problem 1	______	(17 pts)
Problem 2	______	(20 pts)
Problem 3	______	(16 pts)
Problem 4	______	(16 pts)
Problem 5	______	(16 pts)
Problem 6	______	(15 pts)
Exam Total	______	(100 pts)

Alias With $\ll 10^{12}$ tokens.

Good Luck!

Problem 1: [17 pts] Candidate MIPS instruction `subir r1, 22, r3` is to compute `r1=22-r3`, which can't be done with a single existing MIPS instruction. The `22` is taken from instruction bits 15:0, which is the immediate field of Type-I instructions.

The `subir` instruction is to be encoded so that it can be executed by the implementation to the right **with the ALU computing** $X = A - B$, the same operation used by existing subtract instructions. Notice that in the implementation the **immediate connects to both** ALU inputs.

(*a*) Show how `subir r1, 22, r3` instruction would be encoded for this hardware.

☑ Show encoding of `subir r1, 22, r3`. Be sure to show ☑ the position of the fields and ☑ the field values for the sample instruction.

☑ Be sure that the encoding fits with the illustrated hardware and other MIPS instructions.

The solution appears below. The location of the immediate was given in the problem, bits 15:0. The opcode of every instruction in an ISA must be in the same place, for MIPS that is bits 31:26. The instruction has two register operands, a source, `r3` in the example, and a destination, `r1` in the example. Since the ALU will be computing $A - B$ the value of the source register, `r3`, must be delivered to the B ALU input. Only the `rt` value can reach the B input and so the source register must be encoded in the `rt` field. If the source is put in `rt`, the destination register number must be put in the `rs` field, which no other MIPS instruction does.

This encoding is shown below, with the destination, `r1` in the `rs` field and the source, `r3`, in the `rt` field. The immediate is shown in three different radices, full credit would have been given for any one of them, even decimal.

Opcode	RS	RT	Immed
subir	r1	r3	$22_{10} = 16_{16} = 10110_2$
31 26	25 21	20 16	15 0

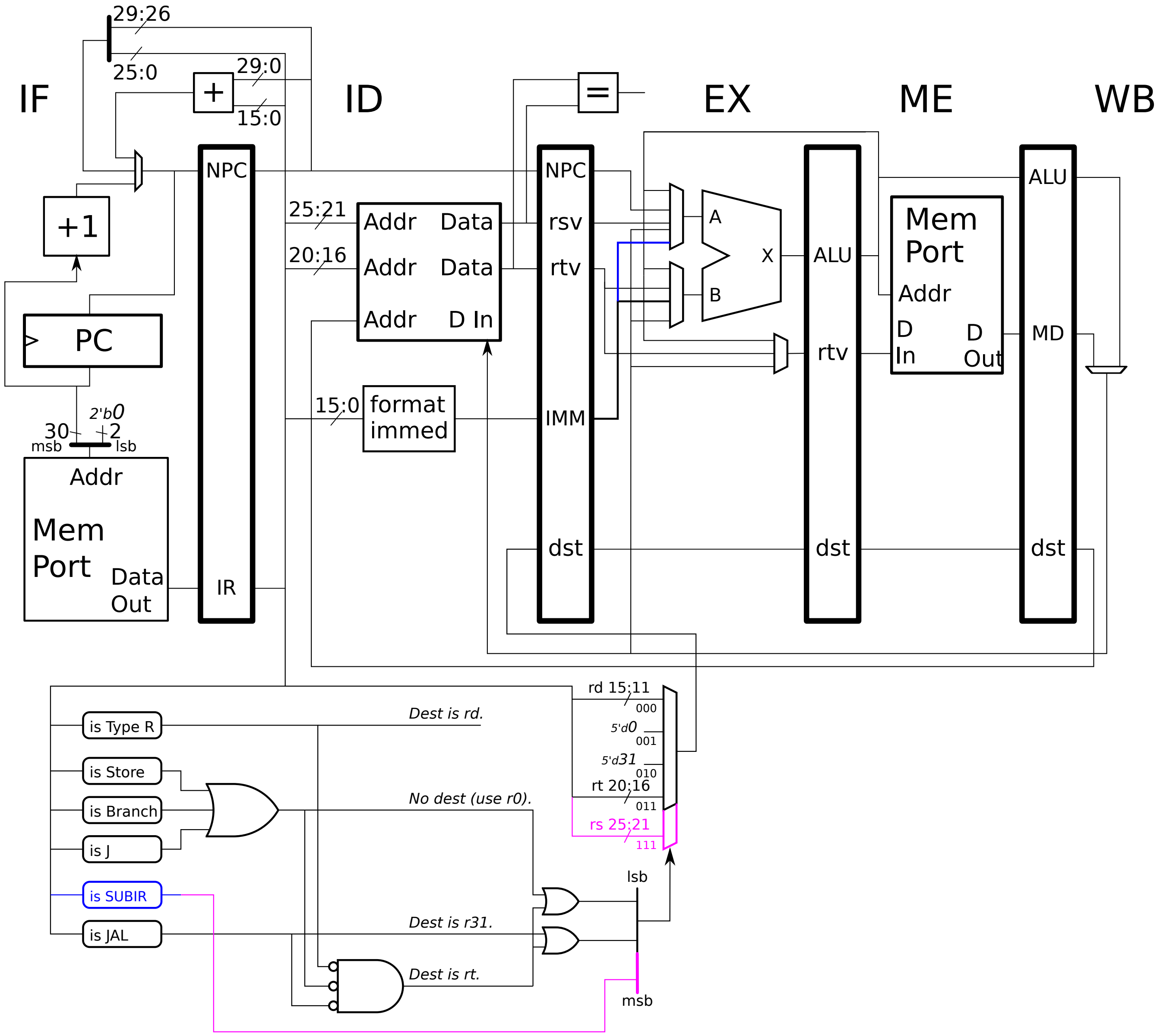
(*b*) Some control logic is shown for the implementation.

☑ Modify the control logic that computes `dst` so that `subir` executes correctly. ☑ **Do not** design control logic for the ALU multiplexors.

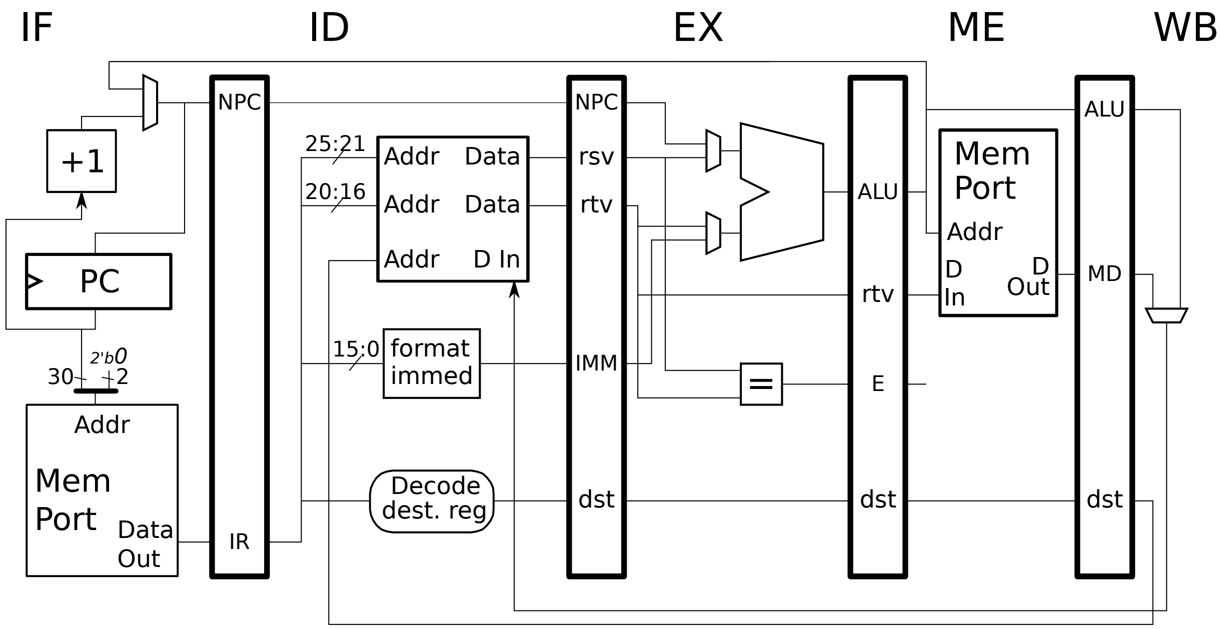
☑ The control logic should not break existing instructions.

☑ The control logic changes should be consistent with your answer to the previous part.

As described above, the destination register for `subir` must be placed in the `rs` field, something that no other MIPS instruction does. So, the logic that computes the destination register, `dst`, must be modified so that `dst` is set to the `rs` field when a `subir` is in `ID`. To do so a fifth input is connected to the mux and that input is set to the `rs` bits, 25:21. That fifth input is numbered 111_2 rather than 100_2 to simplify the logic.


IF
ID
EX
ME
WB
29:26
25:0
29:0
15:0
+
=
+1
PC
NPC
IR
30
2'b0
2
msb
lsb
Addr
Mem
Port
Data
Out
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
NPC
rsv
rtv
IMM
dst
A
B
X
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
is Type R
is Store
is Branch
is J
is SUBIR
is JAL
Dest is rd.
No dest (use r0).
Dest is r31.
Dest is rt.
rd 15:11
000
5'd0
001
5'd31
010
rt 20:16
011
rs 25:21
111
lsb
msb

Problem 2: [20 pts] Show the execution of the code fragments below on their accompanying MIPS implementations.



☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

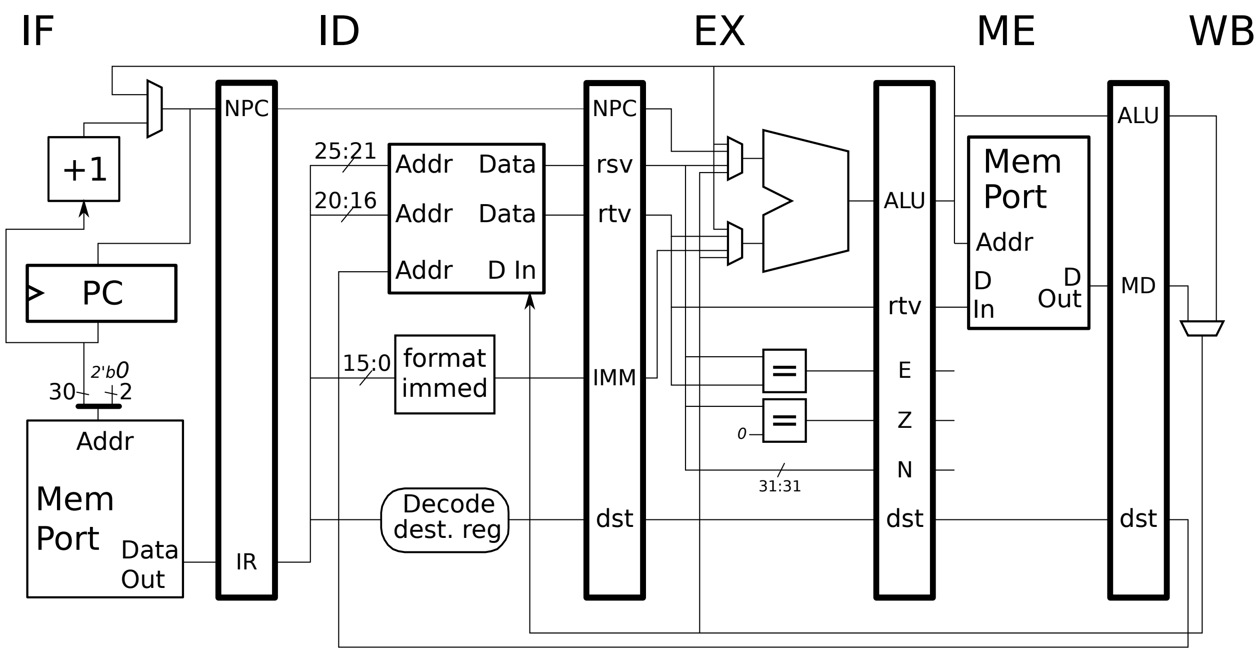
```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4   IF ID EX ME WB
lw   R3, 0(R1)      IF ID ----> EX ME WB
sw   r1, 4(R3)         IF ----> ID ----> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10
```

☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4   IF ID EX ME WB
sw   R1, 4(r3)      IF ID ----> EX ME WB
lw   r3, 0(r1)         IF ----> ID EX M WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10
```

The solution appears above. A common mistake was to not realize that register `r1` in the `sw` instruction is a **source** register, **not** a destination. Therefore there is no true dependence between `sw` and `lw`.

Problem 2, continued:



☑ Show the execution of the code fragment below on ☑ **the implementation above**. ☑ Be sure to check for dependencies.

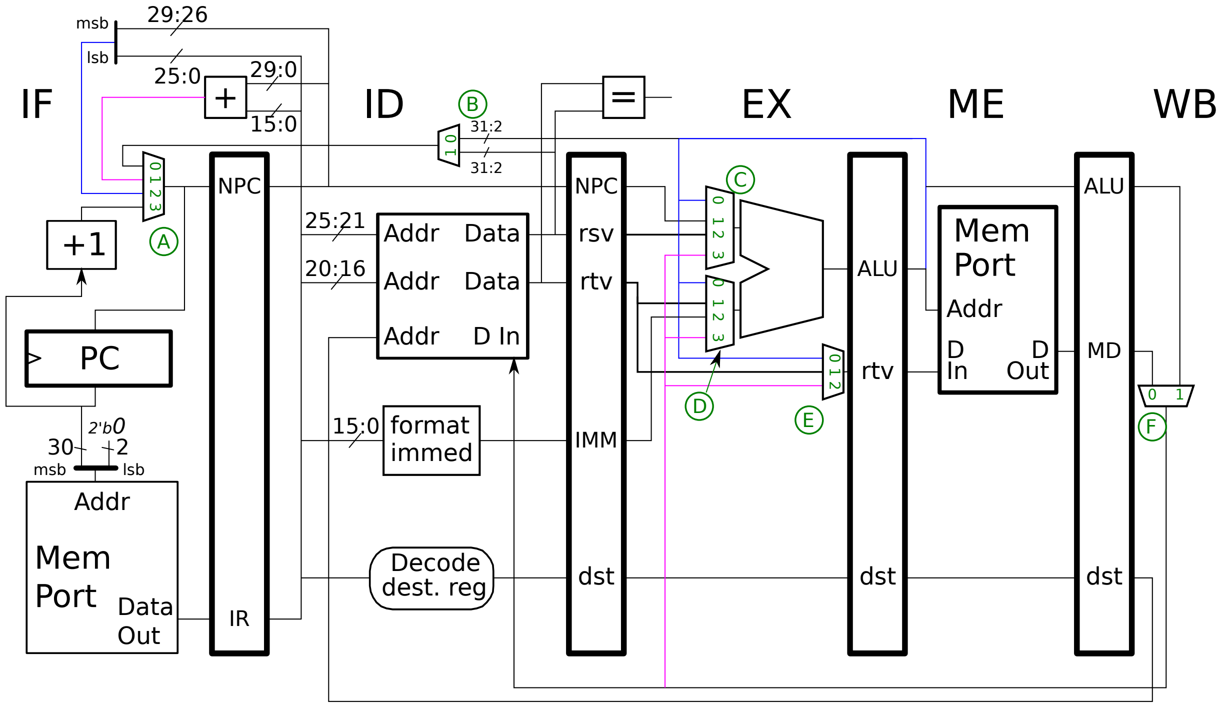
```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4   IF ID EX ME WB
lw   R3, 0(R1)      IF ID EX ME WB
sw   r1, 4(R3)         IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10
```

☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

The solutions appear above and below. Note that this implementation has bypass paths to the ALU but lacks bypass paths to the `EX/ME.rtv` latch, and so the `sw` below must stall until the fresh value of `r1` arrives in `ID`.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4   IF ID EX ME WB
sw   R1, 4(r3)      IF ID ----> EX ME WB
lw   r3, 0(r1)         IF ----> ID EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: [16 pts] Appearing below is the MIPS implementation with labeled multiplexor select signals from Homework 3. Following that is an execution diagram along with a row showing select signal values for the D multiplexor. The first instruction, `add`, is shown.



☑ Complete the code fragment so that it produces the values shown for D.

```
#    SOLUTION
#    Cycle            0  1  2  3  4  5  6  7  8
add R1, r2, r3        IF ID EX ME WB
sub r4, r5, R1           IF ID EX ME WB
xor R6, r7, R8              IF ID EX ME WB
addi r9, r10, 11               IF ID EX ME WB
or  r11, r12, R6                  IF ID EX ME WB
#    Cycle            0  1  2  3  4  5  6  7  8
D:                          1  0  1  2  3
```

Problem 4: [16 pts] Rewrite each code fragment below so that it uses fewer instructions.

☑ Simplify code fragment.

```
addi r1, r0, 123
add  r1, r1, r2

# SOLUTION
addi r1, r2, 123
```

☑ Simplify code fragment.

```
lw r1, 0(r2)
addi r2, r2, 4
lw r2, 0(r2)

# SOLUTION
lw r1, 0(r2)
lw r2, 4(r2)
```

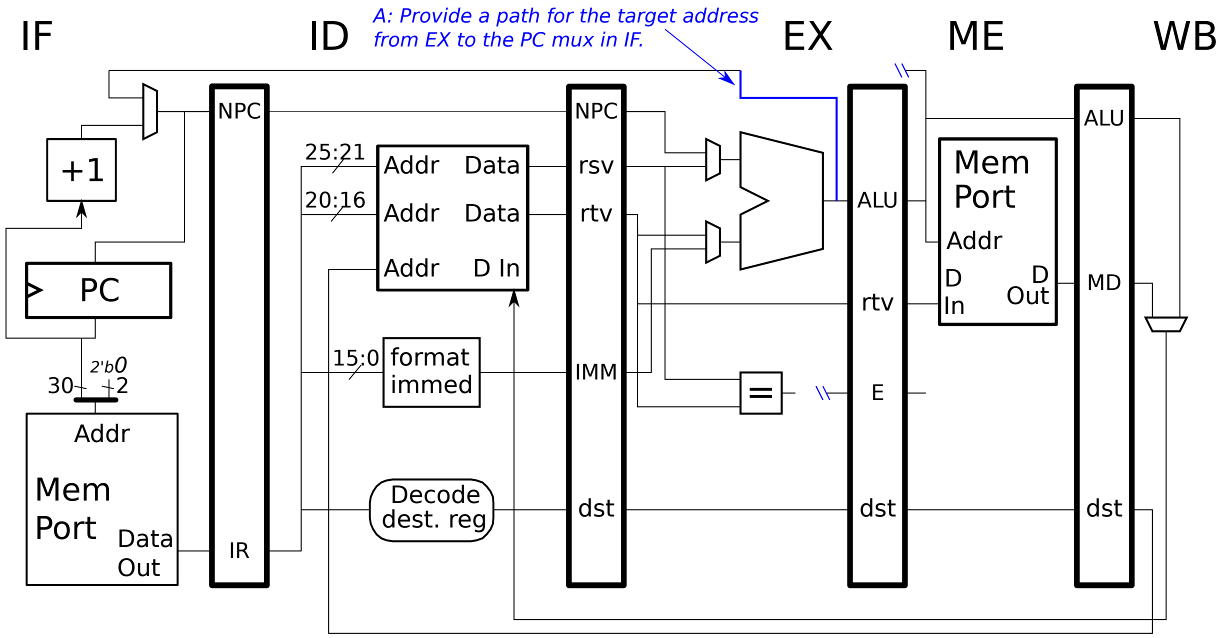
☑ Simplify code fragment.

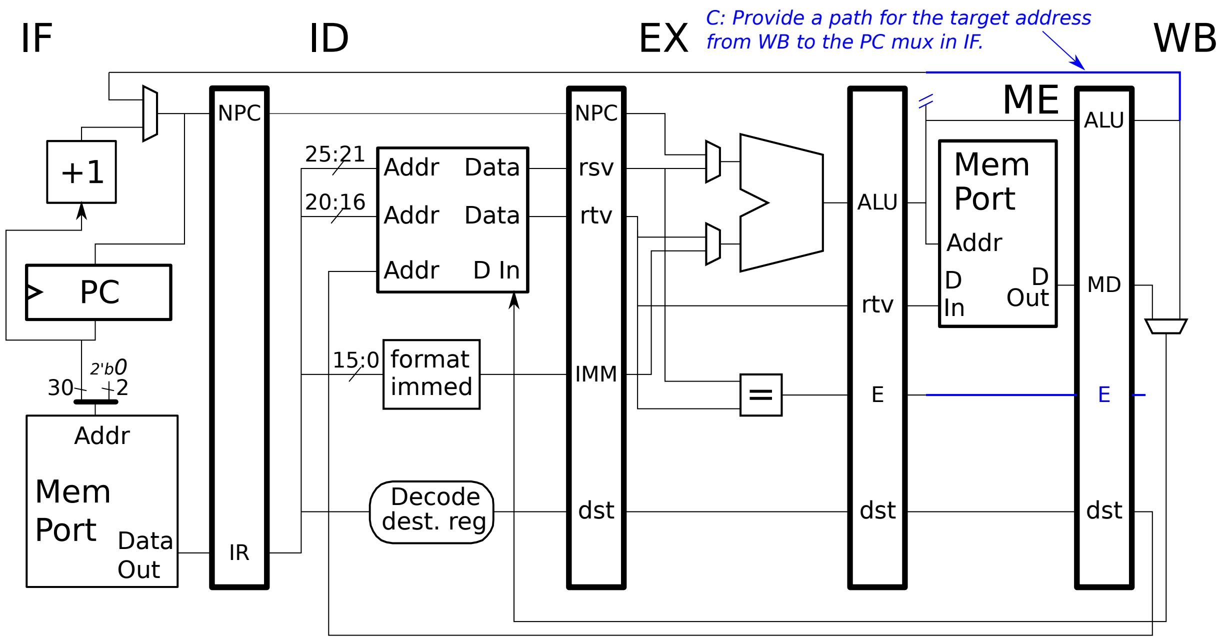
```
sub r1, r2, r3
beq r1, r0,  TARG
lw r1, 0(r4)

# SOLUTION
beq r2, r3,  TARG
lw r1, 0(r4)
```

Problem 5: [16 pts] Appearing below are two identical illustrations of one of our MIPS implementations. To the right are three executions of a code fragment, only one of which is possible on the implementation.

Identify the execution that is possible. For each of the executions that is not possible modify one of the illustrations below so that it is. The modification is very simple, just consider the target address. A few well chosen lines will suffice. No logic gates.

The solution appears to the right and below. The branch target address must appear at the PC mux in `IF` in the cycle before the target is in `IF`. In Execution A the target is in `IF` in cycle 3, so the target address must appear at the PC mux in cycle 2, when the `bne` is in `EX`. Therefore for Execution A the path to the PC mux must be moved from `ME` to `EX`, that's shown in blue. Similar reasoning applies to Execution C.





☑ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ⊗ *No.*

☑ If not, modify the implementation so that it is and ☑ label your modifications *A*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IFx
 lui r5, 0x1234
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10              IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
```

☑ Is the execution below consistent with the unmodified implementation? ⊗ *Yes* or ◯ *No.*

☑ If not, modify the implementation so that it is and ☑ label your modifications *B*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF IDx
 lui r5, 0x1234             IFx
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10                 IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
```

☑ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ⊗ *No.*

☑ If not, modify the implementation so that it is and ☑ label your modifications *C*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF ID EXx
 lui r5, 0x1234             IF IDx
 ori r5, r5, 0x6789            IFx
TARG:
 xor r8,r9,r10                    IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
```

Problem 6: [15 pts] Answer each question below.

(a) Company A and B both come out with a new computer each year. Company A changes both the ISA and implementation each year. Company B changes only the implementation each year but uses the same ISA.

☑ Which company is following accepted practice?

Company B.

☑ Which company's customers are more likely to stay with the company when it is time to upgrade to a new model? ☑ Explain.

Company B, because they do not need to re-write their software.

(b) In MIPS `nop` is a pseudo instruction.

☑ What is a pseudo instruction?

It is something that can be used as though it were a machine instruction in assembly language, even though no such instruction is defined by the ISA or recognized by implementations. Instead, an assembler will translate a pseudoinsruction into a machine instruction (or if necessary multiple machine instructions) that performs the operation defined for the pseudoinsruction. Pseudoinstructions are provided as a convenience. For example, in MIPS it is easier to type `nop` than to type some other instruction that does nothing, such as `sll r0, r0, 0`.

☑ Does having too many pseudo instructions make implementations too expensive? ☑ Explain.

No, since they do not affect the hardware. For example, consider a collection of 500 pseudo-instructions including `plus1 RS` which is translated to `addi RS, RS, 1`, pseudo-instruction `left1 RS` which is translated to `sll RS, RS, 2`, …, and `less1 RT, RS` which is translated to `slti RT, RS, 1`. All of these pseudoinstructions translate into an existing machine instruction so their presence does not affect the ISA and therefore the implementation.

(*c*) The first code fragment below, from code presented in the course, loads element `i` of an array of integers. (Here integers are four bytes.) Complete the second code fragment so that it loads element `i` from an array of shorts (A short is two bytes.).

```
# C CODE                     #  ASM REGISTER = C VARIABLE NAME
# int *a; ...        # $s1 = a;  $t0 = i    sizeof(int) = 4 chars.
# x = a[i];

sll $t5, $t0, 2      # $t5 -> i * 4;  Each element is four characters.
add $t5, $s1, $t5    # $t5 -> &a[i]  (Address of a[i].)
lw $t1, 0($t5)       # x = a[i];   $t1 -> a[i]
```

☑ Complete code below so that it loads a short.

```
# C CODE                     #  ASM REGISTER = C VARIABLE NAME
# short *a; ...        # $s1 = a;  $t0 = i    sizeof(short) = 2 chars.
# x = a[i];

# SOLUTION
sll $t5, $t0, 1      # $t5 -> i * 2;  Each element is two characters.
add $t5, $s1, $t5    # $t5 -> &a[i]  (Address of a[i].)
lh $t1, 0($t5)       # x = a[i];   $t1 -> a[i]
```