Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

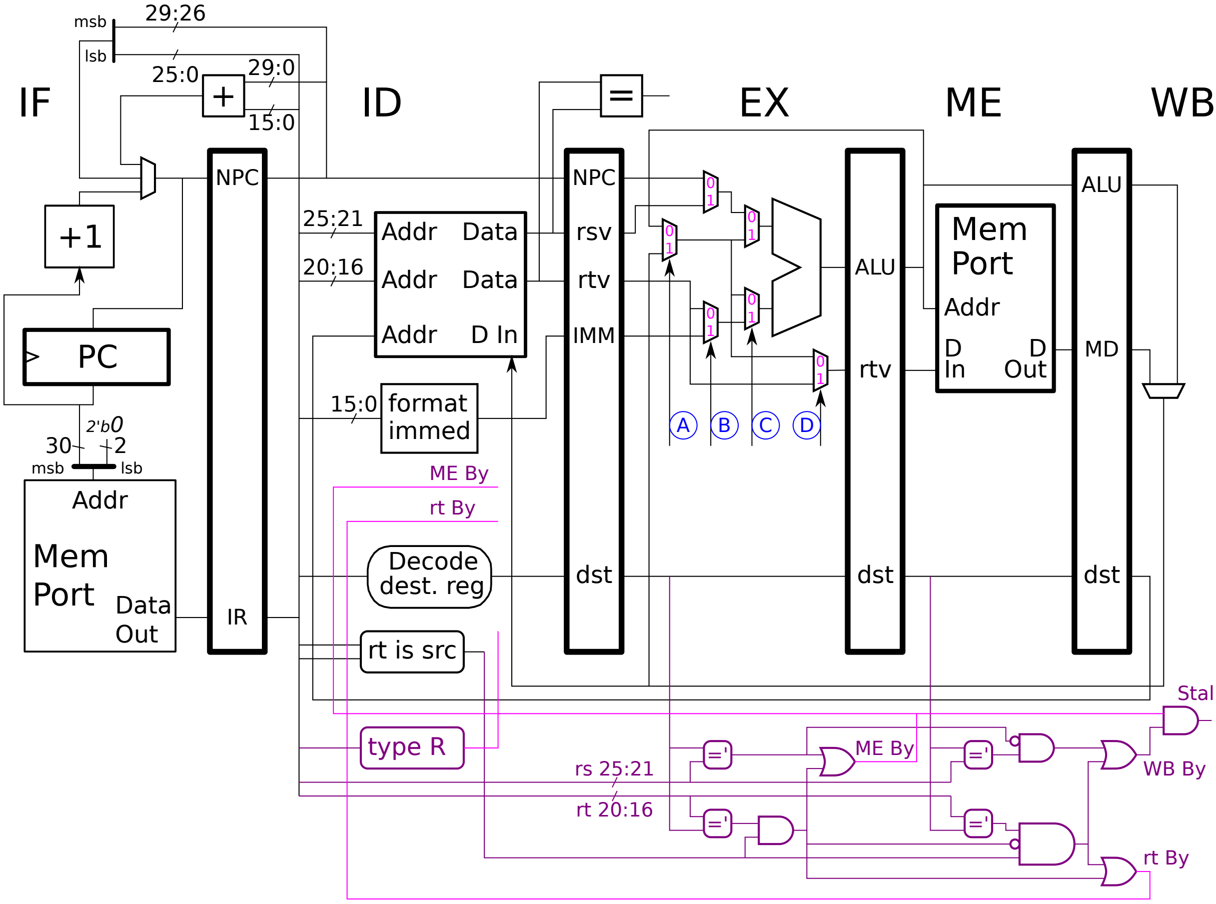
Wednesday, 3 April 2024 9:30-10:20 CDT

Problem 1		(18 pts)
Problem 2		(17 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(20 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias Clouds, be nice!

Good Luck!

Problem 1: [18 pts] Appearing below is a **changed** version of the MIPS implementation appearing in Homework 3 and the 2020 midterm exam.



☑ In the table show the select signal values expected for the execution shown below. ☑ Use `X` for select signals that don't matter (that can be either `0` or `1`). ☑ **Don't forget** to check for dependencies

```
# Cycle             0     1     2     3     4     5     6     7
add r1, r2, r3      IF    ID    EX    ME    WB
ori r7, r1, 9             IF    ID    EX    ME    WB
sub r8, r9, r7                  IF    ID    EX    ME    WB
sw r7, 5(r6)                          IF    ID    EX    ME    WB
# Cycle             0     1     2     3     4     5     6     7
# SOLUTION
# Cycle             0     1     2     3     4     5     6     7
A                               X     0     0     1
B                               0     1     X     1
C                               1     1     0     1
D                               X     X     X     0
# Cycle             0     1     2     3     4     5     6     7
```

Problem 2: [17 pts] Appearing below is the implementation from the previous problem. It **is not identical** to the Homework 3 implementation. *See the last page of this exam for the Homework 3 Problem 3 solution.*

☑ Design the control logic for the A, B, C, and D select signals.

☑ Take advantage of existing logic, not much more logic is needed. ☑ Make sure that C works for the code fragment in the previous part. ☑ Don't forget that execution is pipelined.

Solution appears below in green (in the ID stage).

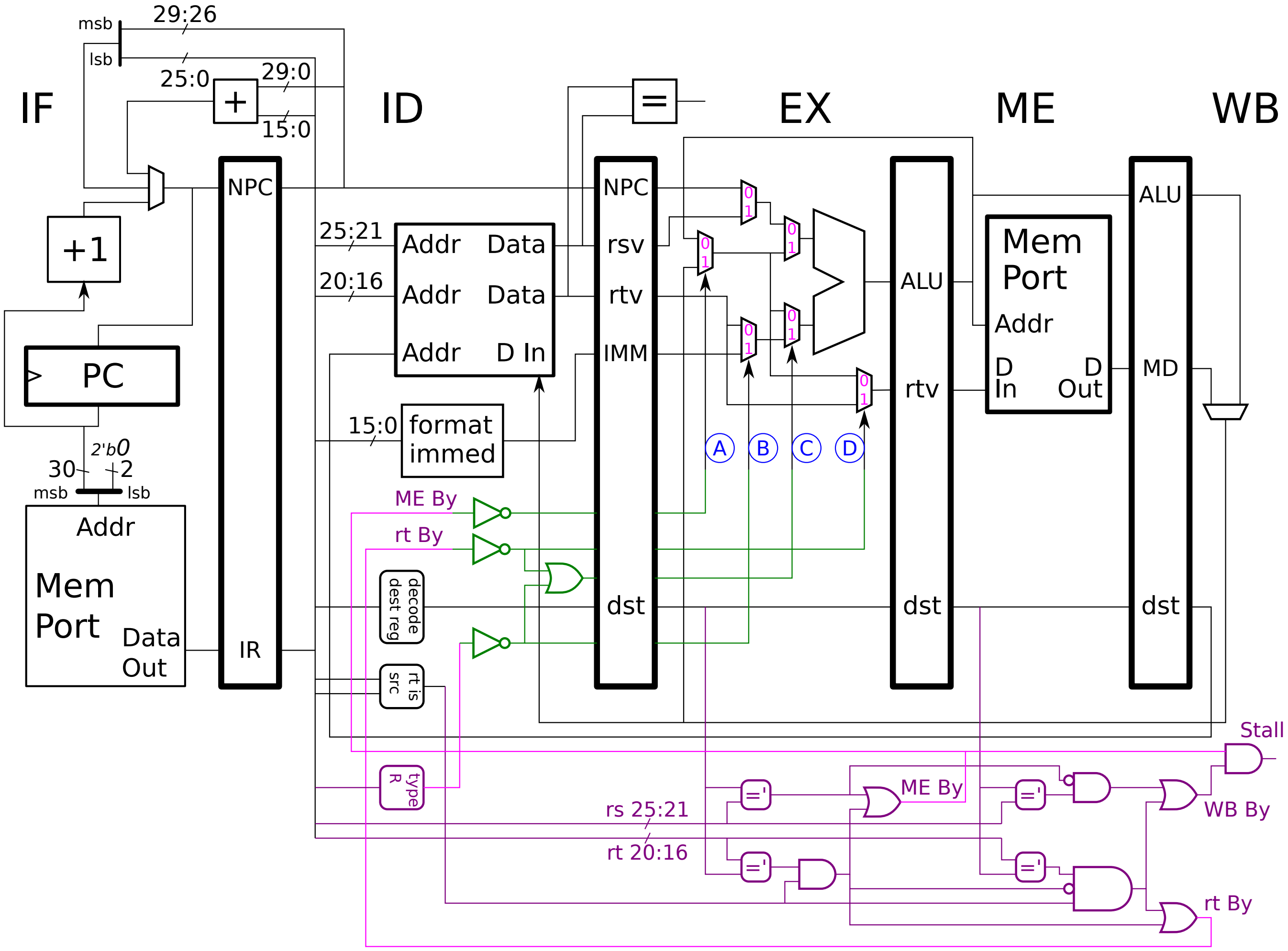
Select signal A uses the existing `ME By` signal, though inverted since a select signal of `1` connects the ME-stage bypass connection to the output. (In the Homework 3 solution the multiplexor inputs were numbered differently. In this exam all multiplexor inputs were numbered in the default way [0 at the top] to avoid confusion.)

Select signal B uses the existing `type R` signal, also inverted because a `1` selects an immediate, something not used in any type-R instruction.

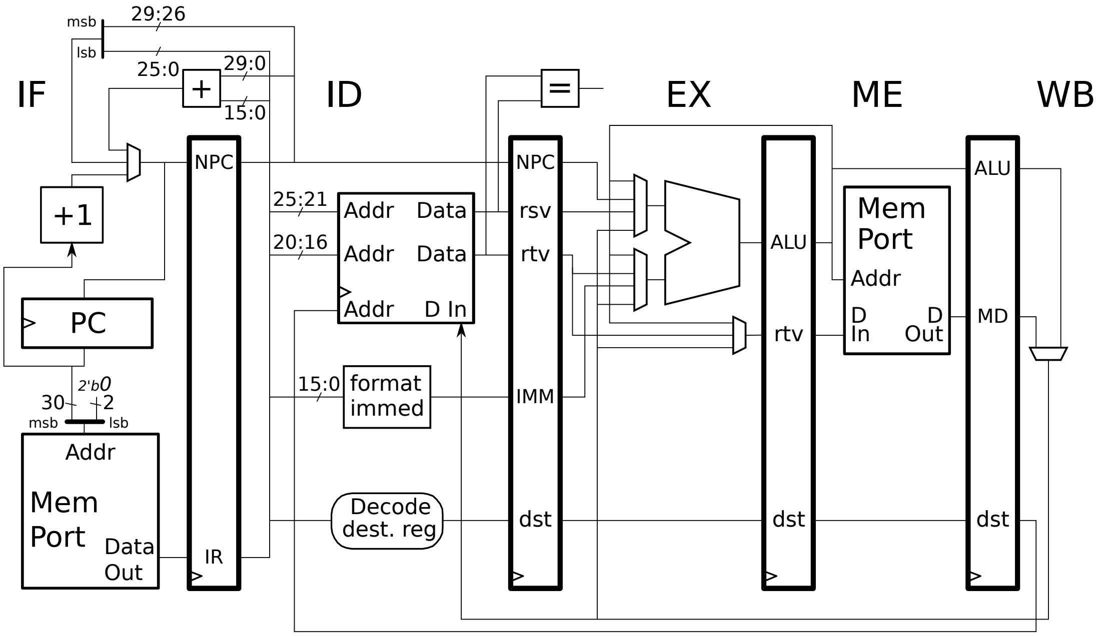
Select signal D uses the existing `rt By` signal, inverted too. Note that there is no need to check whether a store instruction is present, because if one weren't it would not matter what D was.

Select signal C requires a bit more thought. It should be `1` if either the immediate is needed (not `type R`) or the unbypassed `rt` value is needed (not `rt By`). The OR gate computes that.

Note that the logic for the select signals is in the `ID` stage so that the select signals are ready at the very beginning of `EX`.



Problem 3: [15 pts] Show the execution of the MIPS code fragments on the implementation.



☑ Show the execution of the fragment below with ☑ the branch taken. ☑ Pay close attention to branch behavior.

The solution appears below. In MIPS there is a delay slot, which is why `add` executes. In the implementation above the branch target is fetched when the branch is in `EX`.

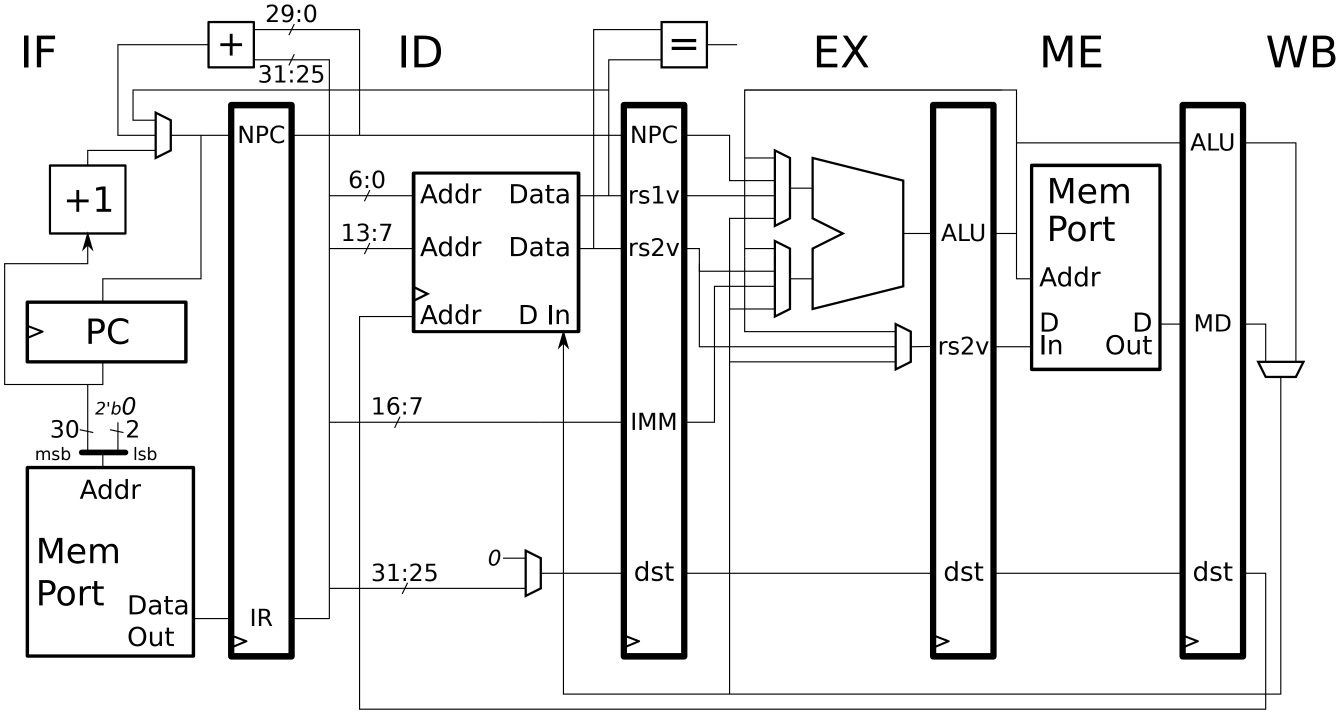
```
 # SOLUTION
 # Cycle             0  1  2  3  4  5  6
 beq r1, r1, SKIPA   IF ID EX ME WB
 add r2, r3, r4         IF ID EX ME WB
 sub r5, r6, r7
 ori r8, r9, 100
 xori r10, r11, 101
SKIPA:
 lw r12, 0(r14)            IF ID EX ME WB
 # Cycle             0  1  2  3  4  5  6
```

☑ Show the execution of the fragment below. ☑ Be sure to check for dependencies.

The solution appears below. There is no way to bypass a load value from `ME` to `EX` in cycle 4, so the `add` stalls. Bypass paths can be used for all other dependencies.

```
 # SOLUTION
 # Cycle             0  1  2  3  4  5  6  7  8
 addi R5, r5, 4      IF ID EX ME WB
 lw R2, 0(R5)           IF ID EX ME WB
 add R1, R2, r3            IF ID -> EX ME WB
 sub r4, R1, r4               IF -> ID EX ME WB
 # Cycle             0  1  2  3  4  5  6  7  8
```

Problem 4: [15 pts] Appearing below is the implementation of Another RISC ISA (ARI) and incomplete diagrams for the encoding of its MIPS-like R and I formats.



☑ How many registers does ARI have?

The address inputs to the register file read ports (in `ID`) each are connected to 7 bits ($6 + 1 - 0 = 7$ and $13 + 1 - 7 = 7$), and so there are $\boxed{2^7 = 128 \text{ registers}}$.

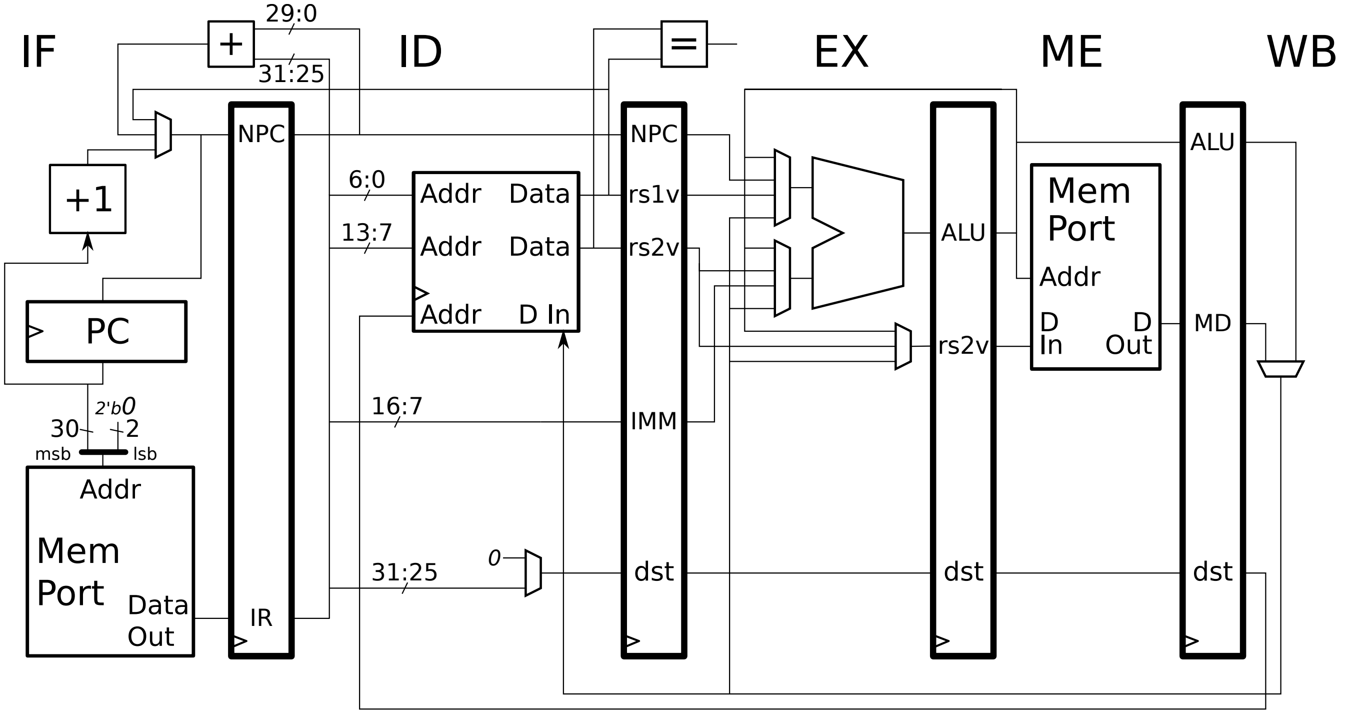
☑ What is ARI's immediate size?

The immediate size is $\boxed{16 + 1 - 7 = 10 \text{ bits}}$.

☑ Why is it possible to implement an instruction like `lw r1, 4(r2)` but not an instruction like `sw r1, 4(r2)` on the implementation above?

Answer: Because the instruction bits for the immediate and `rs2` overlap.

Explanation: Both the `lw` and `sw` use the immediate (along with the `rs1` value) to compute the memory address. The `sw` would need to store a register value, `r1` in the example, to memory. The register to store can't be encoded in the `rs1` field because the `rs1` field is needed for the base address (`r2` in the example). Furthermore the register to store can't be encoded in the `rs2` filed because that overlaps the immediate field. So there is no way to encode a `sw` that includes an offset (the 4 in the example). There's no problem with the `lw` since `r1` is a destination, and those bits don't overlap the immediate.



☑ In the spaces below complete ☑ ARI R and ☑ ARI I instruction formats consistent with the implementation. ☑ Be sure to show the opcode field and any opcode extensions that are needed.

The solution appears below. The opcode is chosen by using available **overlapping** bits in both formats after the bit positions for `rs1`, `rs2`, `rd`, and the immediate are claimed. That leaves bits 24:17, which is big enough for 256 instructions. The three leftover bits in the type R format are labeled `res`, for reserved. They might be used as an opcode extension field.

How realistic is ARI? Not very. Not many ISAs have 128 general-purpose registers. (Itanium is an exception.) Also, the 10-bit immediate is on the small size, it could have been made a few bits larger without shrinking the opcode field too much.

ARI R: `add r1, r2, r3`

rd	opcode	res	rs2	rs1
1	add	0	3	2
31 25	24 17	16 14	13 7	6 0

ARI I: `addi r4, r5, 6`

rd	opcode	imm	rs1
4	addi	6	5
31 25	24 17	16 7	6 0

Problem 5: [20 pts] Answer each question below.

(*a*) Show the contents of the destination register after each MIPS I instruction below executes.

```
#            Initially r1 = 0x12345678
sll r2, r1, 16
#                  ☑  r2 = 0x56780000   # SOLUTION

srl r3, r1, 16
#                  ☑  r3 = 0x1234       # SOLUTION

or  r4, r2, r3
#                  ☑  r4 = 0x56781234   # SOLUTION
```

(*b*) Given the MIPS code below, why might execution never reach the `or` instruction?

```
lw $a0, 0($t0)
jal SOME_CONVENTIONAL_STANDARD_LIBRARY_FUNCTION
addi r31, r31, -8
or $s1, $s1, $v0
```

☑ The `or` instruction won't be reached because:

Each time the `jal` executes the return address (automatically written to `r31`) is changed by the `addi` from the `or` instruction address to the `jal` instruction address.

☑ What will happen instead is:

Each time `SOME_CONVENTIONAL_STANDARD_LIBRARY_FUNCTION` returns it is immediately called again, forming some kind of an infinite loop.

(*c*) Register `r9` holds the address of the middle of a large memory allocation, and so all the MIPS `lb` instructions below execute with no problem. Not so for the `lw` instructions.

```
lb r11, 0(r9)  # Will execute correctly.
lb r12, 5(r9)  # Will execute correctly.
lb r13, 10(r9) # Will execute correctly.
lb r14, 15(r9) # Will execute correctly.

lw r1, 0(r9)
lw r2, 5(r9)
lw r3, 10(r9)
lw r4, 15(r9)
```

☑ Why won't the rest of the MIPS code execute to completion?

Because memory address in the first or second `lw` will be unaligned (not a multiple of 4) and so the unlucky load will raise an unaligned access exception.

☑ What are the maximum and minimum number of `lw` instructions that will execute before an error occurs, and ☑ briefly explain how the maximum and minimum number are determined by the exact value of `r9`.

The maximum number is one, and the minimum number is zero.

If the value of `r9` is a multiple of 4, say `0x1000`, then the first `lw` will execute correctly but the second, attempting to load from `0x1005`, will raise an unaligned address exception and never finish.

If the value of `r9` is not a multiple of 4, say `0x1001`, then the first `lw` will raise the exception and execution will never even reach the second `lw`.

(*d*) Simplify MIPS the code fragment below.

```
lbu r1, 0(r10)
lbu r2, 1(r10)
sll r1, r1, 8
or r1, r1, r2
sh r1, 2(r10)
# Note: r1 and r2 not used again.
```

☑ Simplify the code fragment ☑ without changing what it does.

```
# SOLUTION
lh r1, 0(r10)
sh r1, 2(r10)
```

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

(*a*) In class we described three families of ISAs, CISC, VLIW, and RISC.

☑ How do VLIW ISAs differ from both RISC and CISC ISAs?

In a VLIW ISA multiple instructions are grouped into *bundles*, typically containing three instructions. Branch and jump targets are always to the first instruction in a bundle. In contrast CISC and RISC ISAs jump and branch targets can be to any instruction. Also in VLIW ISAs each slot of a bundle can have different restrictions on the kind of instruction that can be placed there. For example, loads and stores may not be allowed in slot 2, and branches might not be allowed in slots 0 and 1. Lacking bundles, RISC and CISC ISAs lack such restrictions.

(*b*) Identify the ISA family of the following ISAs:

MIPS: ○ *CISC* ○ *VLIW* ⊗ *RISC*

Arm A64: ○ *CISC* ○ *VLIW* ⊗ *RISC*

Itanium: ○ *CISC* ⊗ *VLIW* ○ *RISC*

Intel 64 /IA-32: ⊗ *CISC* ○ *VLIW* ○ *RISC*

VAX: ⊗ *CISC* ○ *VLIW* ○ *RISC*

(*c*) The statement below is wrong.

> *CISC ISAs can have large immediate values, but at the cost of having large instructions. That is why programs in CISC ISAs are large compared to those in RISC ISAs.*

☑ What is correct in the statement above?

The first sentence is correct, but perhaps a little misleading. Misleading because CISC instruction sizes vary, and so though large-immediate instructions are of course large, other CISC instructions can be small, say one byte for a `nop`.

☑ What is wrong in the statement above?

First of all, CISC instruction sizes vary, and so some instructions are smaller than typical RISC instructions. Also, since CISC instructions are more powerful, a single CISC instruction does the same work as multiple RISC instructions, so though that CISC instruction is larger than one RISC instruction, it is smaller than the total size of the RISC instructions it replaces.

Appearing below is part of the solution to Homework 3 Problem 3. It may be helpful in solving Problem 2 in this exam.