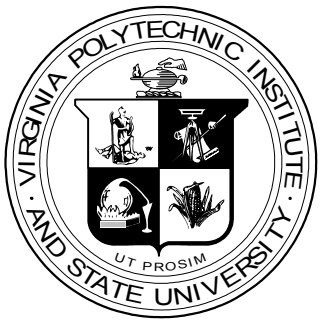

Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet and the MIPS reference card. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions, priced as marked. The maximum score is 100.
- Many of the questions involve making a conclusion and supporting it. Be sure you clearly state your conclusions, and that your supporting arguments are expressed clearly. Being vague or hedging your bets will not be rewarded.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name _____
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____
signed

1. We have discussed NAND (not-and) gates; they correspond to a logical operation usually symbolized using the '|' character.

   a) [5 points] Use a truth table to prove that: $\overline{A} = A \mid A$

```
A       ~A       A | A  ( == ~(A && A)
-----------------------------------
0        1         1
1        0         0
```

   b) [10 points] Use the rules of Boolean algebra (not a truth table) to prove that: $A \wedge B = (A \mid B) \mid (A \mid B)$

   The symbol $\wedge$ stands for the logical AND operator. Be sure to cite the specific rule used in each step of your proof.

```
(A | B) | (A | B) == ~(A && B) | ~(A && B)        // def'n of |
                  == ~[ ~(A && B) && ~(A && B) ] // def'n of |
                  == ~[~(A && B)]                // idempotency
                  == A && B                      // double negation
```

A somewhat shorter proof uses the result from part a:

```
(A | B) | (A | B) == ~(A | B)                     // part a
                  == ~(~(A && B))                // def'n of |
                  == A && B                      // double negation
```

---

2. [5 points] Explain clearly what is meant by the *cycle time* of a clock.

A clock is a free-running signal that alternates between two states, called high and low. The high duration is the length of time the clock stays in the high state before returning to the low state; the low duration is defined similarly.

The cycle time is the sum of the high duration and the low duration.

3. Consider the problem of implementing a circuit to perform addition of two 32-bit operands, represented in two's complement.

a) [10 points] Describe the logical test to determine whether overflow occurs when such an addition operation is carried out. Explain clearly why the test you describe is correct.

Overflow in signed addition occurs if and only if the carry-in Cin to the high bit is different from the carry-out Cout from the high bit.

If Cin == 0 and Cout == 1, then the high bits of both operands must have been 1, meaning both operands were negative. However, in that case, since Cin == 0, the high bit of the result must be 0, so the result will be non-negative which cannot be correct.

If Cin == 1 and Cout == 0, then the high bits of both operands must have been 0, meaning both operands were non-negative. However, since Cin == 1, the high bit of the result must be 1, so the result will be negative, which cannot be correct.

If Cin and Cout are both 0, then the high bits of the operands cannot both be 1. If both are 0, then the sign bit of the result will also be 0, and there is no overflow since there was no carry-in. If the high bits of the operands are different then no overflow is possible.

If Cin and Cout are both 1, then the high bits of the operands cannot both be 0. If both are 1, then the sign bit of the result will also be 1, and there no overflow. If they are different, then no overflow is possible.

b) [5 points] Draw the necessary circuitry to test for overflow in the adder described above. Do not show the adder, just the overflow logic. Be sure to provide clear, descriptive labels for all inputs and outputs.

The adder will produce the two carry bits, so we only need to add circuitry to compare them. The simplest version would be to XOR them.

4. [12 points] Suppose you are given a truth table that defines a Boolean function F of inputs $x_1, x_2, \ldots, x_n$. Describe how to create a sum-of-products expression for a Boolean function F. Do not merely give an example.

For each row of the table in which F == 1, form a product term $y_1 * y_2 * \ldots * y_n$, where $y_i == x_i$ if $x_i == 1$ and $y_i == \sim x_i$ if x == 0 in that row of the table.

Then form the sum of all the product terms.

For questions 5 – 8, feel free to refer to the copy of Fig 5.17 from P&H that was distributed along with the test.

5. Note the list of MIPS instructions that the given datapath supports.

   a) [5 points] For which of those instructions are one or both of the components labeled Sign extend and Shift left 2 needed?

All the I-type instructions use the Sign extend logic: beq, lw and sw

Only branch instructions use the Shift left 2 logic: beq

   b) [8 points] Choose <u>one</u> of the instructions you listed in the previous part and describe carefully how the components labeled Sign extend and Shift left 2 are used in the execution of that instruction.

For lw or sw, the sign extension logic is used to convert the 16-bit literal offset from the instruction to a 32-bit operand suitable as input to the ALU which will add the offset to a register value. The shift logic is not used.

For the beq instruction, the sign extension logic is used in the manner described above, although the recipient is not the primary ALU but an adder. The shift logic is used because for a branch instruction the offset is stored as a number of words and this must be converted to the corresponding number of bytes, which requires multiplying by 4 (which is equivalent to shifting 2 bits to the left).

6. [12 points] For which of the supported instructions should the ALUSrc control line to the multiplexor between the register file and the ALU be set to 1? Why?

The ALUSrc control line determines whether the value of the second operand to the ALU comes from the register file or from the low 16 bits of the instruction (via the sign extension logic).

If ALUSrc is set to 1, then the multiplexer will pass the bits from the instruction.

That will be necessary for the lw and sw instructions since those contain a literal offset which must be added to a value from the register file (the first operand to the ALU) in order to produce the address of the relevant location in memory.

7. [12 points] For which of the supported instructions should the MemtoReg control line to the multiplexor to the right of the data memory be set to 0? Why? Note: be careful about the labeling of the inputs to this one.

The MemtoReg control line determines whether the Write data input to the register file comes from the ALU or from data memory.

If MemtoReg is set to 0 then the value sent to the register file will come from the ALU.

That would be appropriate for the R-type instructions (add, sub, and, or, slt).

8. Consider the component labeled ALU control.

   a) [8 points] The component has an input line labeled Instruction[5-0]. Explain why those bits must be passed to the ALU control unit. Be sure to indicate for which supported instructions this is necessary.

Every R-type instruction has the op-field 000000. The specific arithmetic-logical instruction to be performed is actually specified by the funct field of the instruction, which is bits 5-0.

So, those bits must be sent to the ALU control logic so it can decode them and properly set the ALU control lines for the necessary operation.

   b) [8 points] The ALU control component also takes a 2-bit signal, labeled ALUOp, from the primary Control unit. According to P&H, the ALUOp bits are set differently for a `beq` instruction than for a `lw` or `sw` instruction. Explain why that must be true.

The beq, lw and sw instructions are not R-type and so they do not contain a funct field for the ALU control logic to use to set the ALU control lines. However, all three instructions do require the ALU to do something.

The beq instruction requires that the ALU subtract two registers (and set the Zero output appropriately but that is irrelevant).

The lw and sw instructions require that the ALU add a sign-extended offset to a value from the register file.

Therefore, the ALU control logic unit must know whether the instruction is beq or lw/sw in order to properly set the ALU control lines.