Instructions

Objective

In this lab, you will implement a partial MIPS CPU combining your existing ALU and register file, along with an instruction memory, instruction decoder and various glue logic to implement a MIPS CPU with support for R and I type instructions. This CPU will only implement a portion of the MIPS instruction set, and will not support features such as load/store, jumping, or branching. To demonstrate the efficacy of your CPU, you will implement a simple MIPS assembler program to convert a binary number entered via the switches (`SW`) to decimal for display on the hexadecimal displays (`HEX0` ... `HEX7`).

In addition, you will define and implement a testing methodology that demonstrates the soundness of your design using test benches or any other tool of your liking to ensure that all functionality is correctly implemented. Finally, you will write a report describing this test methodology, and the results thereof.

Deliverables

- Your assignment directory, packed using the provided script.
- **You will be turning in two separate files, one `.7z` generated by the provided script, and one PDF containing your report**.
- Each group only needs to submit once via either partner.

Design Requirements

Your design must use the included `cpu` module with the module lines unchanged[1]. This module will be instantiated to test your design. Your binary-to-decimal converter will also be tested by executing it within your CPU. See *Appendix I* and *Appendix II* for a listing of R and I type instructions respectively.

R-Type instruction requirements:

- (i) The instructions add, addu, sub, subu, mult, multu, and, or, xor, nor sll, srl, sra, slt, sltu, mfhi, mflo, and nop must be implemented as described in *Appendix I*
- (ii) As an additional requirement, to enable access to GPIO the instructions `sra` and `srl` will be modified to read or write to/from the GPIO respectively, when their shift amount is `0`.
  - When the shift amount for either of these instructions is `0`, they simply copy the contents of the source register to the destination register unmodified. There are many other ways

[1] You may change `output` lines to `output logic` or vice-versa at your discretion, so long as the signal names and widths are unmodified.

to accomplish the same thing, but this gives us a convienient way to affect input/output without having load/store to access memory-mapped perhiphreals.
  - The value read from GPIO (in `sra`) will be stored in the destination register (rd).
  - The value written to GPIO (in `srl`) will be sourced from the source register (rs).

I-Type instruction requirements:

- (iii) The instructions lui, addi, addiu, andi, ori, xori, and slti must be implemented as described in *Appendix II*

Instruction Memory Requirements:

- (iv) Your CPU must implement an instruction memory of 4096 32-bit words, which should be initialized from a file called `instmem.dat`.
  - **HINT**: `$readmemh()` is synthesizeable and causes the memory it targets to be initialized with the contents of the vector file.

Assembler Program Requirements:

- (v) Your MIPS assembler program should be stored in a file named `bin2dec.asm`, which should be a MARS-compatible MIPS program. It must read a value from GPIO, and output a decimal version of the value via GPIO. It must read and output values in the method prescribed in requirement *(ii)*.
  - **HINT**: you may hard-code input values in MARS, and comment out the line where this occurs for the version of the program you turn in.
  - "Decimal representation" in this context means that when the value is shown in hex, it can be read correctly as decimal, for example, `0x1ef` should be converted to `0x495`.

MIPS Instruction Pipeline:

- (vi) Your CPU design will implement a pipeline with three stages. This is important because the instruction memory perform synchronous reads, and therefore we must wait one clock cycle after updating the PC for the new instruction to be retrieved.
  - (vi.a) The fetch stage (`F`) will retrieve the next value from the instruction memory.
  - (vi.b) The execute stage (`EX`) will decode the instruction and use the ALU to compute the result of the instructions.
  - (vi.c) The writeback stage (`WB`) will perform writes to the register file. This is needed because when we implement the `load` instruction, the value read from data memory will not be available until after the `EX` stage has completed[2].

[2]There is no data memory in this lab, this is just background information

General requirements:

- (vii) When `KEY0` is pressed, your CPU should reset itself (i.e. this should be your `rst` signal).
- (viii) A specific testing strategy must be **defined** and **implemented** to ensure that the implemented code performs as it should. This might take the form of a Verilog test bench, a ModelSim TCL script, or some other method. Any test scripts or code must execute on the Swearingen Linux lab computers. A `README` file (or similar documentation) describing how to execute the test suite should be provided. The testing code may assumed that a DE2-115 will be connected to the computer it is running on and may be programmed if needed.
- (ix) You must connect and utilize the connections in the `cpu` module as described by their comments and this document.
- (x) You should implement a top-level module in the `CSCE611_lab_ri.sv` file which instantiates your CPU, connects the switches (`SW`) to the GPIO input, and the HEX displays (`HEX0` … `HEX7`) to the GPIO output.
- (xi) When the program counter overflows, it should reset to the value of 0, this is the default behavior of integer overflow in Verilog.

Suggested Approach

There are many possible ways to implement a CPU meeting the specified requirements. This section presents one valid way of meeting the design requirements, and provides some background commentary. If you wish, you may choose another approach.

You will most likely want to implement a "control unit" module. This will in effect implement a lookup table (i.e. combinational logic) which will consider the decoded instruction, and the `alu_zero` signal to determine the state of all control signals in the CPU for the next cycle.

Suggested control unit inputs:

- `instruction[5:0]`
- `instruction[10:6]` (not needed until branches are implement)
- `instruction[31:26]`

Suggested control unit outputs:

- `alusrc` - determines which signal should be used as the ALU `b` input in the next cycle between `readdata1`, `readdata2`, and `instruction[15:0]` either sign or zero extended as appropriate.
- `rdrt` - select the register file `writeaddr` between `instruction[20:16]` and `instruction[15:11]` depending on if the instruction is I or R type.

  - **HINT**: look closely at the instruction encoding differences between I and R type. Notice that in I type the `rt` register is used as the destination, rather than `rd`.

- `regwrite` - register file write enable.
- `regsel` - selects register file `writedata` between the ALU `lo` output, GPIO input, the `lo` register (`mflo`), and the `hi` register (`mfhi`).
- `op` - the ALU operation.
- `shamt` - the ALU shift amount.
- `enhilo` - enable writing to the `hi` / `lo` registers (for mult only).
- `gpiowe` - enable writing to the GPIO register (only for `srl` with `shamt`=0).

You will also want to create several 32 bit registers to store intermediate values:

- `lo` - stores the lower 32 bits of a mult operation
- `hi` - stores the upper 32 bits of a mult operation
- `gpio_out` - stores the 32 bit value to output via GPIO (updated using `srl` with `shamt`=0), the `gpiowe` signal is the write enable for this register.
- `PC` - the program counter, stores the current address in instruction memory, and increments by one every cycle (in later labs, we will add additional logic to handle overriding it's next value to implement jumps and branches).

You should keep pipelining in mind when building your design - it is suggested to suffix any pipeline stage specific signals and registers with `_F` for fetch, `_EX` for execution, and `_WB` for writeback.

For your testing approach, there are many techniques, one that is straightforward to implement is to write one or more test programs that will result in different values in different registers depending on what instructions are working. You may want some output values that are derived from inputs that have data hazards to test your register file write bypassing. A single test case might look something like:

1. Read an assembled MIPS program via `instmem.dat` and reset the CPU.
2. Execute the CPU for some number of cycles.
3. Check the state of the registers via testbench against known-good ("ground truth") values.
    - You might establish ground truth by observing the behavior of the simulator.

You may wish to keep in mind that your CPU will be tested for a grade by changing the contents of `instmem.dat`, then stimulating the GPIO input and observing the GPIO output. It may be wise to implement some test cases that exercise this behavior.

Although the grading rubric does not require you to test your code on the DE2-115 board, this is highly encouraged – if a design works only on the board or only in the simulator, this can belie deeper design flaws. It is thus a good "sanity check" to ensure that your design works in both.

Report

You must write a report. The report should be written using a reasonable choice of font and other stylistic options. The report **must** be in PDF format.

The report must include the following information:

- Three to five paragraphs describing your testing strategy (rationale, design decisions, etc.). Imagine the reader as a new person having just joined your team who will need to use your tests.
- A justification of how the test cases you have written demonstrate and prove the correctness of the design under test. The precise nature of this justification will vary depending on your approach and style. It may include text, figures, tables, code listings, etc. Roughly 1-3 (but no more than 5) pages is expected.
  - Some examples of suitable justifications would include (but are not limited to):
    * A short (~ 1 sentence) description of each test case in a suite of test cases (which might be implented using non-synthesizeable Verilog in ModelSim, as a ModelSim TCL script, by instrumenting SignalTap, or by some other method).
    * Annotated screenshots of ModelSim waveforms which demonstrate correct operation.
    * Annotated screenshots of SignalTap waveforms or memory monitors which demonstrate correct operation.
    * Commented source code for a software simulator, and a script which generates inputs to ModelSim and checks outputs against the simulator's results.
  - Examples of **insufficent** justifications would include (but are not limited to):
    * A description of a fully manual test procedure.
    * A video or photographs of a fully manual test procedure.
    * A collection of test cases which leaves significant portions of the design untested.
    * A collection of test cases or vectors without any description.
- A short analysis (1-3 paragraphs) of your implemented design using your test methodology. You should indicate if your design works, and if not why it does not.

The specific organization, layout, prosaic style, etc. of your report is left up to your discretion and good judgment.

Rubric

- (A) 18 points – correct implementation of R type instructions.
  - One point per instruction (18 instructions total).

- (B) 21 points – correct implementation of I type instructions.
  - 3 points per instruction (7 instructions total).
- (C) 11 points – correct implementation of MIPS assembler program to perform binary to decimal conversion.
- (D) 40 points – lab report.
  - (D.1) 10 points – description of testing strategy.
  - (D.2) justification of tests.
    - (D.2.a) 10 points – clear description of specific tests or test cases implemented.
    - (D.2.b) 10 points – thoroughness of test cases (partial credit awarded for tests that are partially thorough, proportional to throughoghness)
  - (D.3) 10 points – analysis of implemented design.
- (E) 10 points – pre-lab assignment.
  - Points are awarded based on number of correctly completed problems, to be submitted via Moodle/Dropbox. See `prelab.pdf`.

Maximum score: 100 points.

Additionally, the following may cause you to lose points:

- If the code provided in your submission **does not match** what is shown in your report (i.e. in screenshots, code listings, etc.), you will be given a failing grade on the assignment.
  - This includes functional descriptions of your code. For example, if you did not implement any of the ALU operations, but your report indicated you did, you will receive a failing grade.
- **If your report is submitted without your project code (or your project code without your report), you will be given a failing grade on the assignment.**
- Your code may be run through Moss, and your report through plagiarism detection software such as TurnItIn. **Plagiarism and other forms of cheating will be reported to the academic honesty department, which may result in a grade penalty.**
- Failure to follow the styling guidelines will result in a one letter grade penalty **(** $\frac{1}{10}$ **of maximum points for the assignment)**.
  - The same styling requirements apply as for the ALU lab.
- Failure to follow requirements *(ii)* or *(vii)*, or modifying the signal names or widths of the `cpu` module will make it impossible to grade your code, and you will receive a score of 0 on all relevant rubric sections (A, B, and C).

Appendix I - MIPS R-Type Instructions

add *rd*, *rs*, *rt* :

Puts the sum of the integers in the register *rs* and the register *rt* into register *rd* and checks for overflow.

31-26	25-21	20-16	15-11	10-6	5-0
000000	rs	rt	rd	00000	100000

addu rd, rs, rt :

Puts the sum of the integers in the register *rs* and the register *rt* into register *rd* and does not check for overflow.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100001

sub rd, rs, rt :

Subtracts the integer in register *rt* from the integer in register *rs*, putting the result into register *rd* and checks for overflow.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100010

subu rd, rs, rt :

Subtracts the integer in register *rt* from the integer in register *rs*, putting the result into register *rd* and does not check for overflow.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100011

mult rs, rt :

Multiplies the signed integers in registers *rs* and *rt*. Leave the low-order word of the product in register

lo and the high-order word in register *hi*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	00000	00000	011000

multu rs, rt :

Multiplies the unsigned integers in registers *rs* and *rt*. Leave the low-order word of the product in register *lo* and the high-order word in register *hi*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	00000	00000	011001

and rd, rs, rt :

Puts the logical AND of the integers from register *rs* and *rt* into register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100100

or rd, rs, rt :

Puts the logical OR of the integers from register *rs* and *rt* into register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100101

xor rd, rs, rt :

Puts the logical XOR of the integers from register *rs* and *rt* into register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100110

nor rd, rs, rt :

Puts the logical NOR of the integers from register *rs* and *rt* into register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	100111

sll *rd, rt, shamt* :

Performs logical shift of contents of register *rt* left by *shamt* bits, putting results into register *rd*. Vacated bits are filled with 0's.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	*rt*	*rd*	*shamt*	000000

srl *rd, rt, shamt* :

Performs logical shift of contents of register *rt* right by *shamt* bits, putting results into register *rd*. Vacated bits are filled with 0's.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	*rt*	*rd*	*shamt*	000010

sra *rd, rt, shamt* :

Performs aritmetic shift of contents of register *rt* right by *shamt* bits, putting results into register *rd*. Vacated bits are filled with replicas of sign bit.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	*rt*	*rd*	*shamt*	000011

slt *rd, rs, rt* :

If the signed integer in register *rs* is less than the signed integer in register *rt* then store the value 0x00000001 in register *rd*, otherwise store the value 0x00000000 there.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	101010

sltu *rd, rs, rt* :

If the unsigned integer in register *rs* is less than the unsigned integer in register *rt* then store the value 0x00000001 in register *rd*, otherwise store the value 0x00000000 there.

31-26	25-21	20-16	15-11	10-6	5-0
000000	*rs*	*rt*	*rd*	00000	101011

mfhi *rd* :

Copies the contents of the hi register to register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	00000	rd	00000	010000

mflo *rd* :

Copies the contents of the lo register to register *rd*.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	00000	rd	00000	010010

nop :

No operation. Used to insert a stall into a pipeline.

31-26	25-21	20-16	15-11	10-6	5-0
000000	00000	00000	00000	00000	000000

Appendix II - MIPS I-Type Instructions

lui *rt*, *imm* :

Load the immediate value *imm* into the upper half-word of register *rt*. The lower bits of the resgister are set to 0.

31-26	25-21	20-16	15-0
001111	00000	*rt*	*imm*

addi *rt*, *rs*, *imm* :

Put the sum of the integer in register *rs* and the sign extended immediate value *imm* into the register *rt* and check for overflow.

31-26	25-21	20-16	15-0
001000	*rs*	*rt*	*imm*

addiu *rt*, *rs*, *imm* :

Put the sum of the integer in register *rs* and the sign extended immediate value *imm* into the register *rt* and do not check for overflow.

31-26	25-21	20-16	15-0
001001	*rs*	*rt*	*imm*

andi *rt*, *rs*, *imm* :

Put the logical AND of the value in the register *rs* and the zero extended immediate value *imm* into register *rt*.

31-26	25-21	20-16	15-0
001100	*rs*	*rt*	*imm*

ori *rt*, *rs*, *imm* :

Put the logical OR of the value in the register *rs* and the zero extended immediate value *imm* into register *rt*.

31-26	25-21	20-16	15-0
001101	*rs*	*rt*	*imm*

xori *rt*, *rs*, *imm* :

Put the logical XOR of the value in the register *rs* and the zero extended immediate value *imm* into register *rt*.

31-26	25-21	20-16	15-0
001110	*rs*	*rt*	*imm*

slti *rt, rs, imm* :

If the signed integer in register *rs* is less than the sign extended immediate *imm* then store the value 0x00000001 in register *rt*, otherwise store the value 0x00000000 there.

31-26	25-21	20-16	15-0
001010	*rs*	*rt*	*imm*

