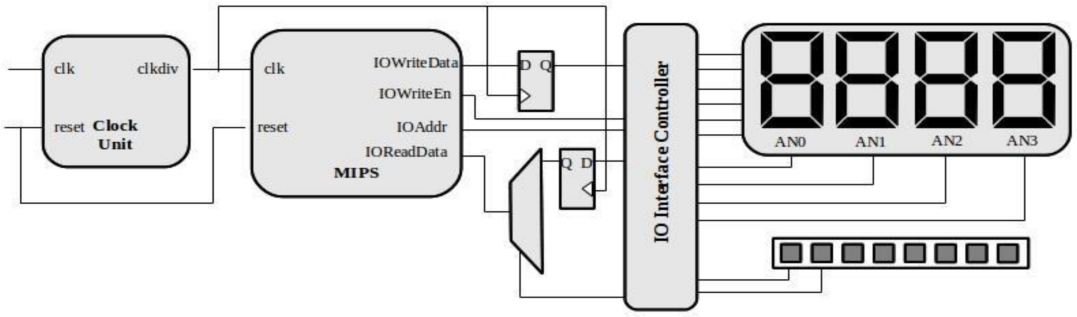
DDCA-u08a-lab



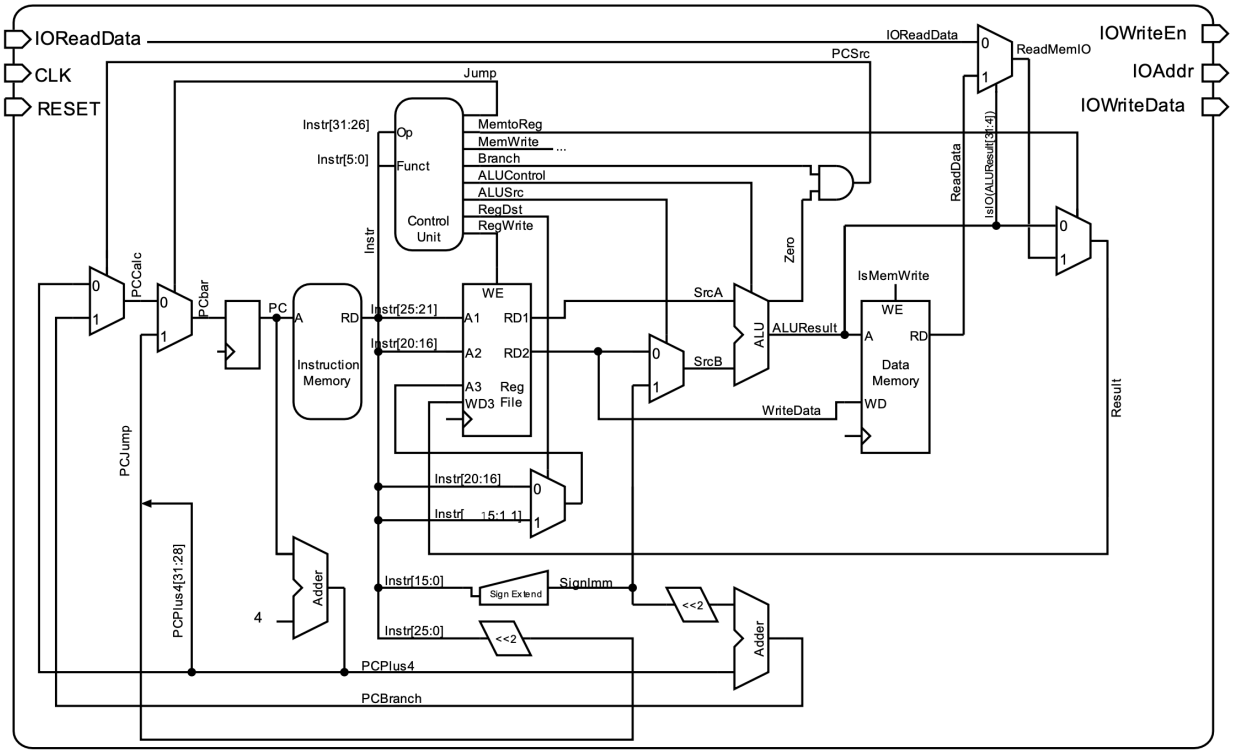
top.v	Top level hierarchy that connects the MIPS processor to the I/O on the FPGA board. *You will modify this file for Part 2.*
top.xdc	Constraints file of the top level. *You will modify this file for Part 2.*
MIPS.v	The main processor. *For Part 1, you have to add code inside of this file only.*
DataMemory.v (datamem_h.txt)	The initial content of the data memory (composed of 64 32-bit words). *The* datamem_h.txt *file contains the data part of the assembly program in a hexadecimal form. This module "loads" the data. You will only have to modify the .txt file if you do the challenges.*
InstructionMemory.v (insmem_h.txt)	The ROM (composed of 64 32-bit words) contains the program. *The* insmem_h.txt *file contains the assembly instructions we want to run on the MIPS processor in a hexadecimal form. This module "loads" the instructions. You will modify the .txt file for Part 2.*
RegisterFile.v	Register file that creates two instances of *reg_half.v* as read ports and has one write port. *This is the implementation of a register. You do not need to modify it.*
reg_half.v reg_half.ngc	Component describing a single port memory and binary description of how it is mapped in the FPGA. *These are used to implement the register. You do not need to modify it.*
ALU.v	ALU similar to the one from Lab 5. *You should not change anything in this file, but if you want, you can use your own implementation (just make sure that it works).*
ControlUnit.v	The unit that does the instruction decoding and generates nearly all the control signals. Table 7.5 on page 379 lists most of them and their truth tables (only the `AluOp` signal is generated differently in the exercise). *This is just a combinational circuit, and it's already given; you don't need to change anything here.*
snake_patterns.asm	Assembly program corresponding to the *datamem_h.txt* and *insmem_h.txt* dump files that displays a crawling snake on the 7-segment display when all the parts are connected properly. *You have to modify this file for Part 2, where you will also learn how to generate the dump files.*

What's the difference between `MemWrite`, `DataMemWrite`, and `IOWriteEn`?

			Signal		
			`MemWrite`	`DataMemWrite`	`IOWriteEn`
Instruction	SW instruction	on DataMem address	1	1	0
		on IO address		0	1
	Non-SW instruction	D.C. (don't care)	0	0	0

Open the file *MIPS.v*. Note that all the required signals are already declared at the top of the module. Use the block diagram in Figure 2 as a reference to add the correct instantiation for:

- **Instruction Memory**: Note that the address of the instruction to be read is determined by the PC (program counter). The PC is always incremented by 4 to fetch the next instruction from memory. We add 4 instead of 1 because each address in memory stores one byte, and each MIPS instruction requires four bytes of memory. Therefore, we can throw away the 2 least significant bits of the address (because they are not necessary here) and use the next 6 bits (7 to 2) for its 64 words.
- **ALU**: The given (or your) ALU from Lab 5 has 4 bits for `AluOp`, whereas the controller generates a six-bit value that is the function field of an R-type instruction. You will have to select the 'correct' four bits to connect here[2].
- **Data Memory**: Just like the instruction memory, use the 6 most significant bits (7 to 2) of the actual address.
- **Control Unit.**




```
assign IsIO = ALUResult[31:4] == 28'h00007ff;
assign IsMemWrite = MemWrite & ~IsIO;
assign IOWriteData = WriteData;
assign IOAddr = ALUResult[3:0];
assign IOWriteEn = MemWrite & IsIO;
```

Report

(1)

Which MIPS instructions do you think would produce wrong outputs if the ControlUnit signal *RegWrite* is *'stuck at 0'*, i.e., *RegWrite* always has the value 0? In other words, which MIPS instructions depend on the control signal *RegWrite*?

All Mips instructions that store something back into the register won't work anymore.
In the controlUnit we can see all affected operations listed:
`OP_RTYPE`, `OP_LW` and `OP_ADDI`

---

(2)

Explain why a 6-bit address is enough for the instruction and data memory. *(Hint: think about the size of the memory.)*

A 6-bit address can represent $2^6 = 64$ distinct values. In both the `DataMemory` and `InstructionMemory` modules, the memory arrays are declared as `reg [31:0] DataArr [63:0]` and `reg [31:0] InsArr [63:0]`. Thus 6 bits are enough to uniquely address each data and instruction in memory.

(3)

As you might have noticed, there are three different counters used in this lab. One is present in the *snake_patterns.asm* file, the second is in the clock_div module, and the third is the `DispCount` signal for the 7-segment display. Explain the functions of each of these three counters/dividers in a sentence or two each.

`snake_pattern.asm`

This counter keeps track of the position in the loop of snake patterns. Making it go faster, loops through the pattern faster.

`clock_div.v`

This counter is responsible for slowing down the frequency of the processor. It is incremented on every tick of the hardware clock and every x ticks, it sends a *divided* clock output.

```
// Instantiate an internal clock divider that will
// take the 50 MHz FPGA clock and divide it by 5 so that
// We will have a simple 10 MHz clock internally
```

`DispCount`, `top.v`

This counter determines, which seven segment display to turn on (send logical 0). `DispDigit` controls, which specific segment is turned on, but for all AN. For this Reason, we need to turn on only the one that is needed.

```
always @ ( * ) begin
        case (DispCount[15:14])
                2'b00: begin
                        AN = 4'b1110;
                        DispDigit = DispReg[6:0];
                end   // LSB
                2'b01: begin
                        AN = 4'b1101;
                        DispDigit = DispReg[13:7];
                end   // 2nd digit
                2'b10: begin
                        AN = 4'b1011;
                        DispDigit = DispReg[20:14];
                end   // 3rd digit
                2'b11: begin
                        AN = 4'b0111;
                        DispDigit = DispReg[27:21];
                end   // MSB, default
        endcase
end
```