LAB 8.1 – Full System Integration (Part 1)

Goals

- Learn how a processor is built.
- Learn how the processor communicates with the outside world.
- Implement a MIPS processor and demonstrate a simple "snake" program on the FPGA starter kit.

To Do

- Learn the components of the MIPS processor. We designed an ALU in Lab 5. Several other components are given.
- Assemble the components of the MIPS processor.
- Make changes to the processor to enable memory-mapped I/O write operations.
- If everything is correct, you should observe a tiny "snake" that loops on the 7-segment LED. Make changes to the system to control the speed at which the snake crawls. The speed should be based on the amount specified by the switches on the board.
- *[Optional] For the challenge seekers, try to change the snake's motion pattern. Additionally, you could try to change the direction of the snake's motion based on a switch input.*
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- This lab spans **two lab sessions**. To complete each lab session, you have to **show your work to an assistant before the deadline.** You will get **up to 7 points for each lab session.**

Introduction

This is it. This is the exercise we have been working towards the entire semester. During this week's exercise, we will finally put together a microprocessor and run your own programs. In order to see the processor in action, we will add some I/O interfaces to control and display a crawling snake on the 7-segment LED. Figure 1 shows a simplified block diagram of how the final system looks. This lab is divided into **two parts** spread over **two weeks**. In the first part, we put together the basic building blocks of a processor and try to run a "crawling snake" program. In the second part, we extend the I/O functionality to control the speed of the crawling snake using the switches present on the FPGA starter kit.

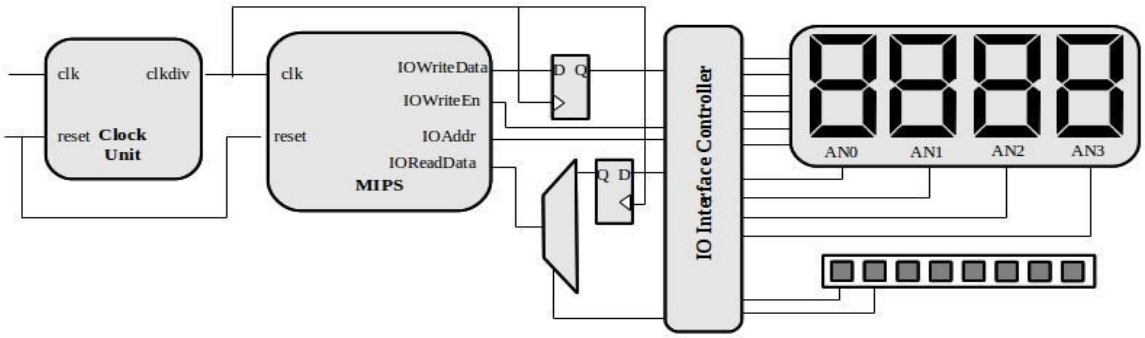


Figure 1. Simplified block diagram of this exercise

The MIPS processor that we will implement is closely based on the architecture presented in your textbook; specifically, the single-cycle processor depicted on page 383 in Figure 7.11. In the example code, we adhere to the same signal names and organization as the figure in the book to make it easier to follow the given

source code. The processor itself is not terribly complicated, and its performance pales in comparison to a modern processor. However, it is far from useless. The MIPS that we implement runs faster than 10 MHz and can execute 32-bit instructions. This easily beats state-of-the-art processors from the early 1980s (e.g., Intel 8086 and Motorola 68000). For example, anything that landed on the moon had a much less capable processor. The system clock on the FPGA board is 50 MHz, which is slightly faster than what we can use for our processor (the critical path is around 25 ns); however, the provided clock_div block generates a clock that is 4x slower (12.5 MHz), so you do not have to design it separately.

While the processor itself provides interesting functions, it alone cannot communicate with the external world. It requires input and output interfaces. To this end, we modify the MIPS processor to have standard I/O signals. In this standardized interface, we have a separate data output (`IOWriteData`) and data input (`IOReadData`) bus (each 32 bits wide). Since we want to address multiple I/O resources, we will also provide an address signal `IOAddr`. This address helps us to identify which I/O resource we are accessing (writing to the 7-segment display or reading from the register that holds the direction input, etc.). Finally, there is a signal `IOWriteEn` that tells us that we are actually performing a write operation (not a read I/O operation).

We use two registers[1]: (i) to store the current value to be displayed on the 7-segment display and (ii) the speed level for the crawling snake. We design the I/O interface controller that ties all these circuits together and enables the processor to access the various I/O circuits that we have added.

It is important to note that we could make most of these changes within the processor instead of using a separate block. We add these changes into a separate block because we do not want to re-design the processor every time we have a new I/O device. In other words, the MIPS processor (with the I/O interface) stays constant even if we completely change the I/O circuits.

[1] The challenge requires another register.

SESSION I

THE CRAWLING SNAKE

In this first part, we put together the building blocks of the MIPS processor. For this exercise, we provide you with a Vivado project that already contains many parts of the processor. Once you examine the code, you will realize that there is nothing mysterious about it. In fact, you could easily write all of it without problems. However, the instructions will guide you through the exercise by explicitly saying which parts of the code are relevant for which task. Note that the given code and architecture are correct, but not necessarily the best possible implementation.

Go to the course webpage and download the .zip file containing the archive with the Vivado project directory. Extract the directory in your working directory and start Xilinx Vivado. Open the project file lab8.xpr that is among the extracted files.

In the directory of the extracted files, you will find the following file structure:

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.*

MIPS Processor

In this part of the exercise, we build the MIPS processor, which is implemented in the provided file *MIPS.v*. This file, however, is incomplete since the main blocks are not yet connected (i.e., instantiated). Figure 2 shows the corresponding block-level overview of the MIPS processor. Note that the output signals are not connected because later in this exercise you have to decide how they should be implemented according to their description.

To complete building the MIPS processor, you have to finish the following tasks.

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.**

Memory Mapped I/O

In order to see the processor in action, we must extend the previously built MIPS processor to communicate with the external peripherals (e.g., 7-segment LEDs, switches, buttons). We will use a simple memory-mapped I/O architecture, i.e., part of the memory address space is reserved for I/O operations. When data is written to or read from this address space, it is intercepted by the circuit and re-routed to the I/O circuitry. To complete this part of the exercise, you must complete the signal assignments for `DataMemWrite`, `IOWriteData`, `IOAddr`, `IOWriteEn`.

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

[2] At the moment, we are only using 7 ALU instructions, and the ALU does not need all 6 signals of the Function field (of an R-type instruction). In Lab 9, we will need to modify the ALU a little bit and add support for more bits. This avoids the need to modify the controller for future changes.

Open the file *MIPS.v* and complete the assignments. The code includes comments designated to help you complete the incomplete signals. For example, the `IsIO` signal has already been implemented. In our processor, we reserve the address range 0x00007FF0 to 0x00007FFF for I/O operations.

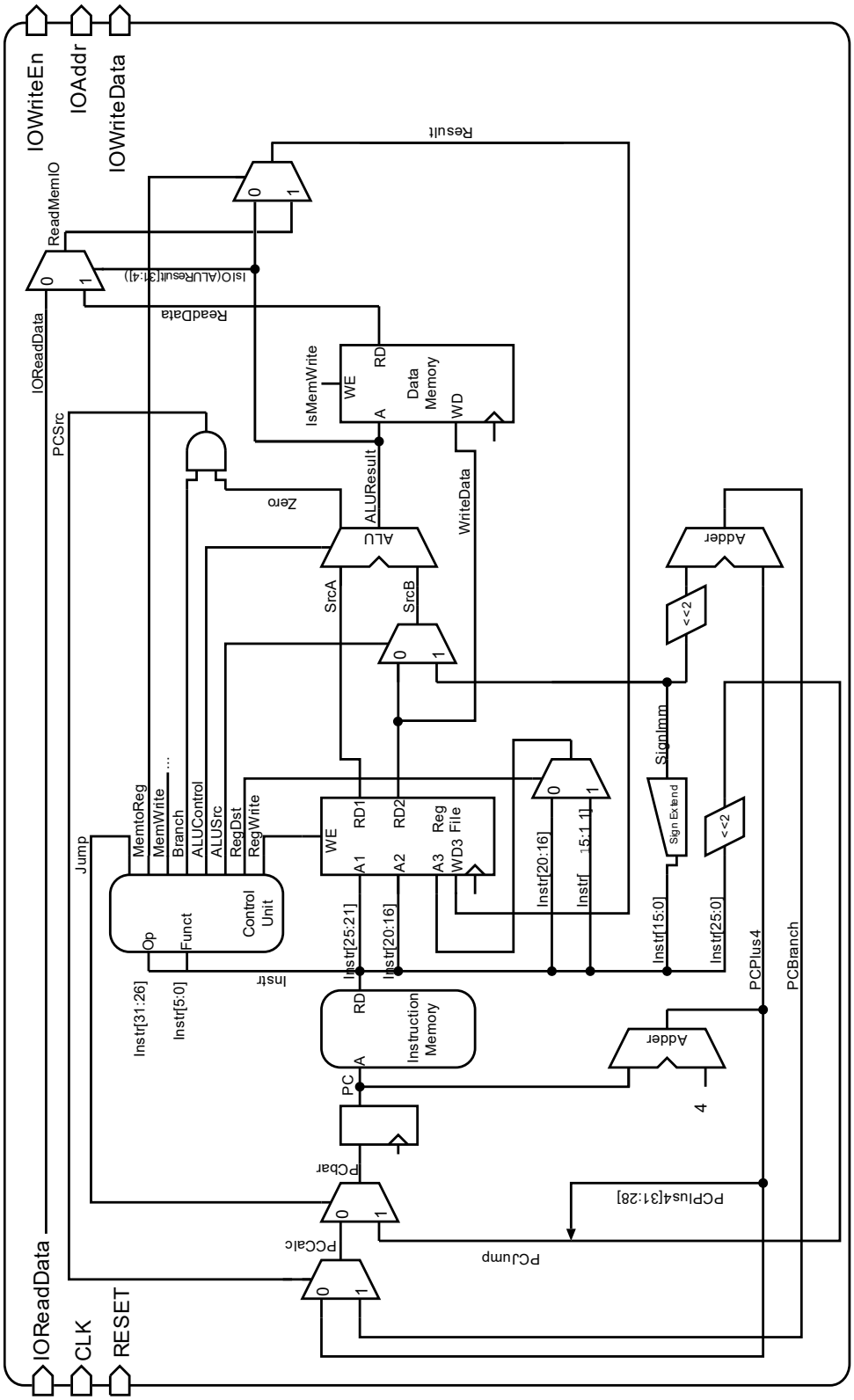


Fig 2. Block diagram of the MIPS processor that we will implement in this exercise. This diagram is almost identical to Fig 7.14 on page 387 of your textbook

Crawling snake on the 7-segment LED

In this first part of the exercise, we only need a 28-bit output register that contains the value to be sent to the four 7-segment LEDs. We assign a separate address for this register as given in the table below:

Address	Direction	Width	Description
`0x00007FF0`	`out`	`28 bits`	Value to be sent to the 7-segment display.

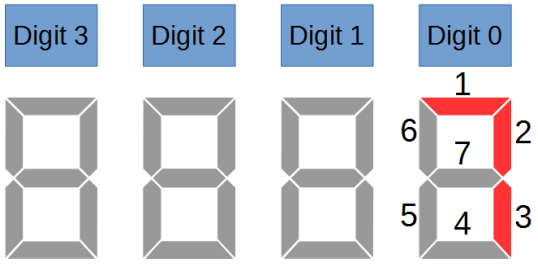
Therefore, the following instruction in the assembly program:

```
sw $t0, 0x7FF0($0)
```

writes the content of the register $t0 to the address 0x00007FF0[3], which represents this other register outside of the processor. Additional external circuitry (already implemented in *top.v*) converts these values so that they can be displayed on the FPGA board. In this exercise, we will not use the decoder circuit from Lab 3 since there are no digits to be displayed. We directly use the contents of the 28-bit register to simulate the crawling snake.

It would be easy if the four 7-segment LEDs had individual signal lines. In practice, displays are rarely driven by such parallel connections. Since humans can only distinguish movement that is slower than 20-50 ms, each of the four 7-segment LEDs is enabled sequentially. On the FPGA board, all four displays have a separate enable pin (AN3, AN2, AN1 and AN0), and the segment connections are shared among all displays. The enable pins are active low, meaning that the segment displays the number if the corresponding AN signal is 0 and does not display anything if it is 1. To display all four numbers, we need four clock cycles. In the first cycle, AN3 is set to 0, all others are set to 1, and the pattern to be displayed is applied. In the next clock phase, only AN2 is activated, and the second number is applied, and so on. Once the last number is displayed, the loop starts with the first number again. We will see our pattern as long as this process is repeated swiftly enough (faster than 20 ms). This logic has already been implemented for you in the *top.v* file and the *top.xdc* contains the correct mapping.

The content of the 28-bit register defines the 7-segment display as follows:



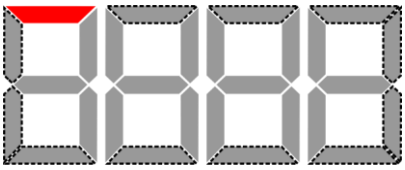
28-bit Register value for the figure on the left:

7 6 5 4 3 2 1	7 6 5 4 3 2 1	7 6 5 4 3 2 1	7 6 5 4 3 2 1
0000000	0000000	0000000	0000111
Digit 3	Digit 2	Digit 1	Digit 0

[3] We are using the 'Compact Data at Address 0' configuration of the MIPS. In this configuration, the MMIO address range is from 0x00007FF0 to 0x00007FFF. We will use this range as it is easier for us (only 16-bit addresses are used).

Assembly Program

At this point, We have a processor that is able to do memory-mapped I/O, and we have added an interface that uses the 7-segment display and several buttons. Now we need to make everything work. For this purpose, we need an assembly program that implements the crawling snake. This has already been implemented for you in the *snake_patterns.asm* file. Try to understand the code: it has been sufficiently commented. The files *datamem_h.txt* and *insmem_h.txt* contain the data and instruction op-codes corresponding to the file *snake_patterns.asm.* We will learn how to generate these hexadecimal dump files in Part 2.



Generate the programming file in Xilinx Vivado and program the FPGA. You should see a crawling snake moving, as shown in the figure above. Fix any problems you might have. Show your results to an assistant.