Prof. Salma Elmalaki
Fall 2023

Lab 2 (100 Points)

In this lab, we want you to complete a Single Cycle MIPS Processor. Please check the associated Verilog files with this lab manual.

1 Single Cycle Processor

Figure 1 shows a single cycle processor.

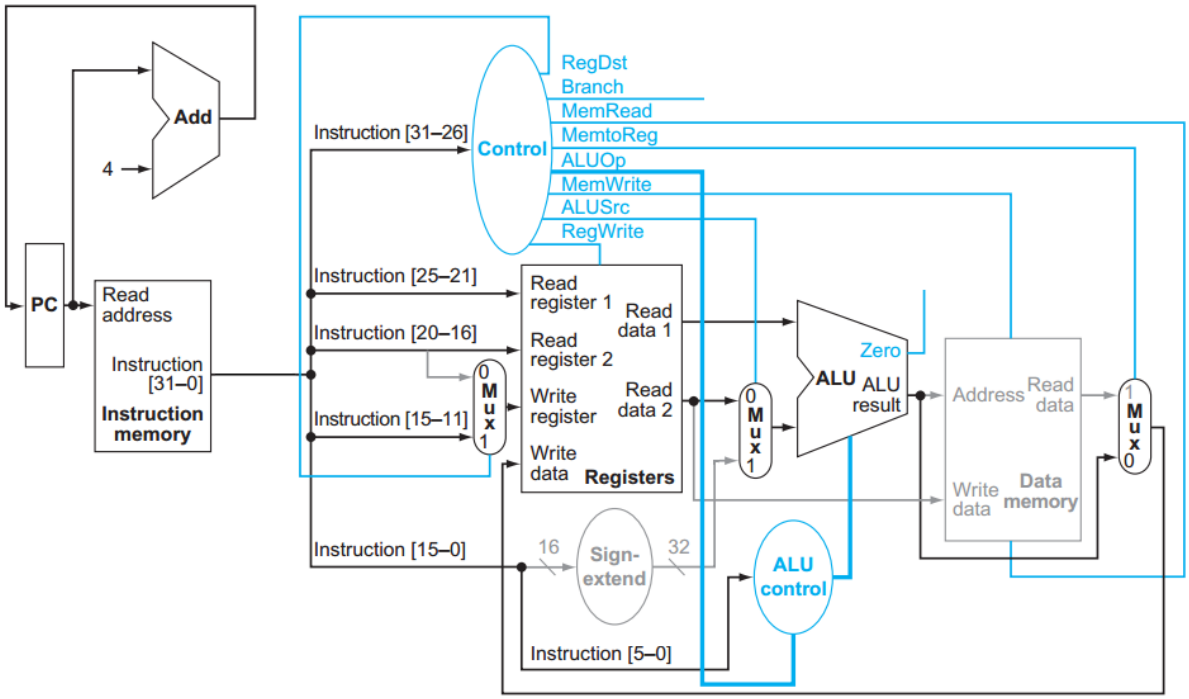


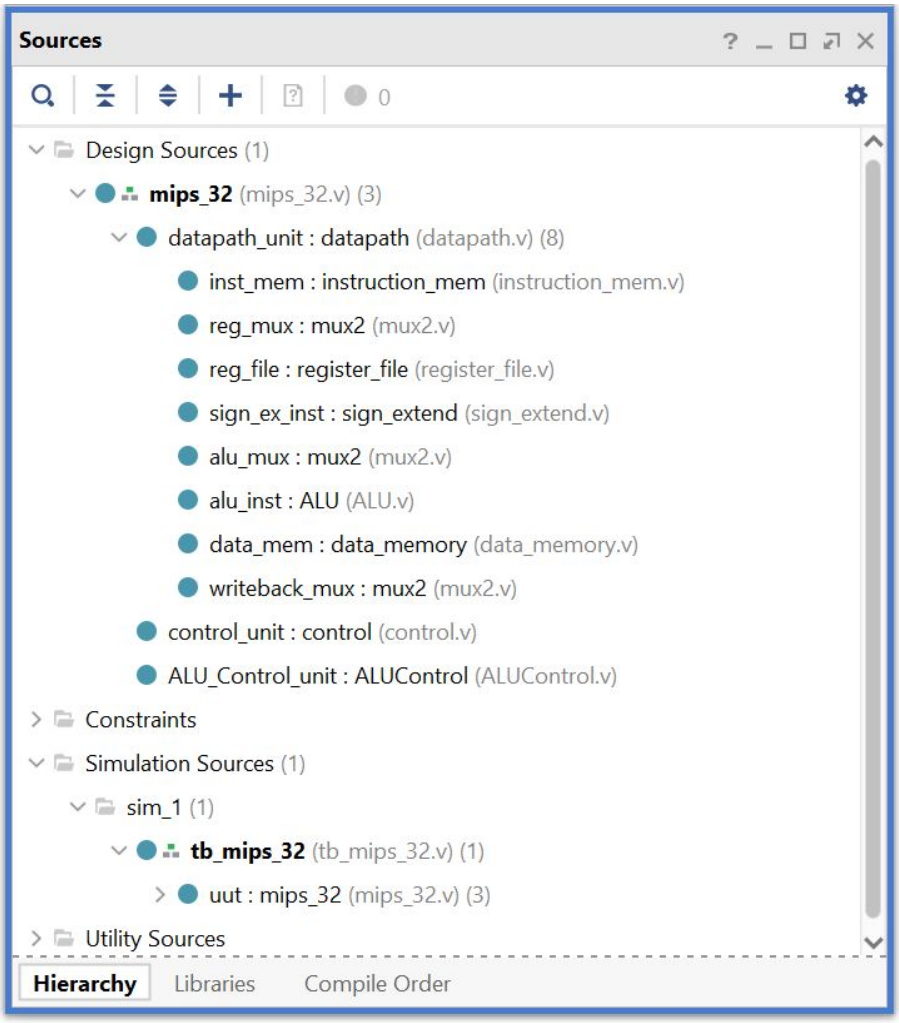
Figure 1 : Datapath.

The instruction execution starts by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the

PC and branch offset are summed) or from an adder that increments the current PC by four. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Create a new project in Vivado using the files listed below. Find these files under lab2 on Canvas.
mips_32.v, datapath.v, control.v, ALUControl.v, instruction_mem.v, mux2.v, register_file.v, sign_extend.v, ALU.v, data_memory.v, and tb_mips_32.

You will see this in the source window:

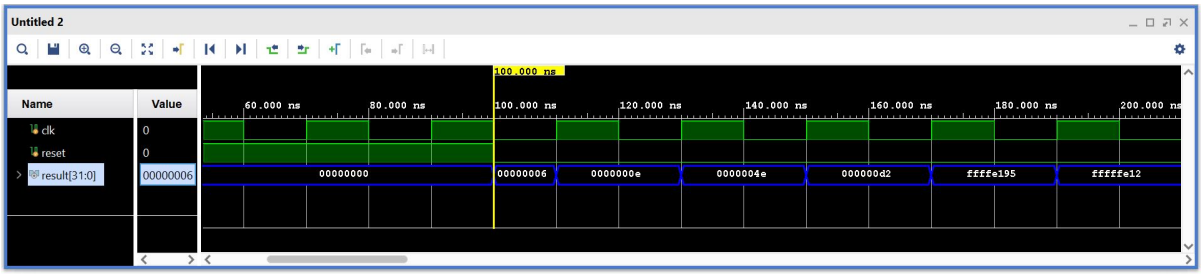


Now you have a mips processor which is able to run these 7 instructions:

Table 1

Name	Format	op	rs	rt	rd	shamt	funct	Comments
add	R	0	reg	reg	reg	0	32	add $s1,$s2,$s3
sub	R	0	reg	reg	reg	0	34	sub $s1,$s2,$s3
addi	I	8	reg	reg	n.a.	n.a.	n.a.	addi $s1,$s2,100
lw	I	35	reg	reg	n.a.	n.a.	n.a.	lw $s1,100($s2)
sw	I	43	reg	reg	n.a.	n.a.	n.a.	sw $s1,100($s2)
and	R	0	reg	reg	reg	0	36	and $s1, $s2, $s3
or	R	0	reg	reg	reg	0	37	or $s1, $s2, $s3

Run the simulation and see the waveform.



If you look at the testbench file (tb_mips_32.v) you see that the reset signal is set to "0" after $100ns$. So, for the first $100ns$ we are reseting the processor. Then at each rising edge of the clock we read a new instruction from the instruction memory and execute it to find the result. You can compare the values on the waveform with the expected results in the instruction_mem.v file.

We want to complete this processor and add a new set of instructions to it. To do that you need to know the architecture of the processor and how each of these instructions run.

Table 2: Instruction set.

Name	Format	op	rs	rt	rd	shamt	funct	Comments
Add	R	0	reg	reg	reg	0	32	add $s1, $s2, $s3
Addi	I	8	reg	reg	n.a.	n.a.	n.a.	addi $s1, $s2, 20
and	R	0	reg	reg	reg	0	36	and $s1, $s2, $s3
andi	I	12	reg	reg	n.a.	n.a.	n.a.	andi $s1, $s2, 20
Beq	I	4	reg	reg	n.a.	n.a.	n.a.	Beq $s1, $s0, L1
Lw	I	35	reg	reg	n.a.	n.a.	n.a.	lw $s1, 20($s2)
Nor	R	0	reg	reg	reg	0	39	nor $s1, $s2, $s3
Or	R	0	reg	reg	reg	0	37	or $s1, $s2, $s3
Slt	R	0	reg	reg	reg	n.a.	42	slt $s1, $s2, $s3
Sll	R	48	reg	reg	reg	Shift amount	0	sll $s1, $s2, 10
Srl	R	48	reg	reg	reg	Shift amount	2	srl $s1, $s2, 10
sra	R	48	reg	reg	reg	Shift amount	3	sra $s1, $s2, 10
Sw	I	43	reg	reg	n.a.	n.a.	n.a.	sw $s1, 20($s2)
Sub	R	0	reg	reg	reg	0	34	sub $s1, $s2, $s3
xor	R	0	reg	reg	reg	0	38	xor $s1, $s2, $s3
Mult	R	0	reg	reg	reg	0	24	mult $s1, $s2, $s3
div	R	0	reg	reg	reg	0	26	div $s1, $s2, $s3
jump	J	2	n.a.	n.a.	n.a.	n.a.	n.a.	J 2500

2 MIPS Instructions

MIPS has 3 different instruction types. Table 1 shows these three instruction formats. In our simple implementation of MIPS we only have some R-type instructions plus load word and store word and *addi* instructions which are I-type.

Table 3 : Instruction Format.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

2.1 R-Format

Below you see the R-Format instruction fields.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- op: Basic operation of the instruction, traditionally called the opcode.
- rs: The first register source operand.
- rt: The second register source operand.

- rd: The register destination operand. It gets the result of the operation.
- shamt: Shift amount.
- funct: Function. This field, often called the function code, selects the specific variant of the operation in the op field

Here is an example for add instruction:

```
add $s16,$s5,$s7

op        rs       rt       rd       shamt    funct
000000    00101    00111    10000    00000    100000
```

For shift instructions, we want to shift *rs* register *shamt* times and store back the result in the *rd* register. We are not using the *rt* register field for shift instructions.

```
sll $s1,$s3,2

op        rs       rt       rd       shamt    funct
110000    00011    00000    00001    00010    000000
```

Shift instructions here, are a little bit different from the MIPS architecture:

- The opcode for shift instructions is 48(110000).
- The second operand in the shift instructions is the *shamt* which comes from the sign extends unit. The sign extends unit output is 32 bit (imm_value) but we only want to use 5 bits of it. So for the shift amount use bits $[10:6]$ of the second input to your alu.
- The *rt* field in the shift instructions is always zero. (please look at the *sll* example)
- *sra* is shift right arithmetic. It means that we want to insert the sign bit. you can use the arithmetic shift operator but keep in mind that for this operator the first operand should be a signed value.

2.2 I-Format

Below you see the I-Format instruction fields. They are the same as R-Format except they have an address (or immediate value) field.

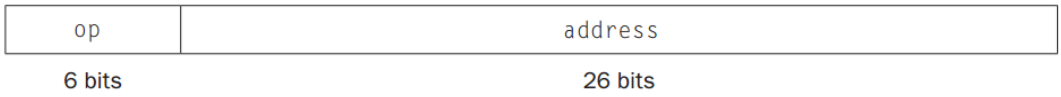
Here is an example for *beq* instruction. If register one and register three are equal, we want to branch to two instructions after the current instruction:

```
beq    $s3, $s1, #2

op        rs       rt       constant or address
000100    00011    00001    0000000000000010
```

2.3 J-Format

Below you see the J-Format instruction fields.



Here is an example for jump instruction with immediate value #2.

```
j       #2

op                  address
000010      00000000000000000000000010
```

3 MIPS processor

After you modify the code, your processor would have the following control path and datapath.

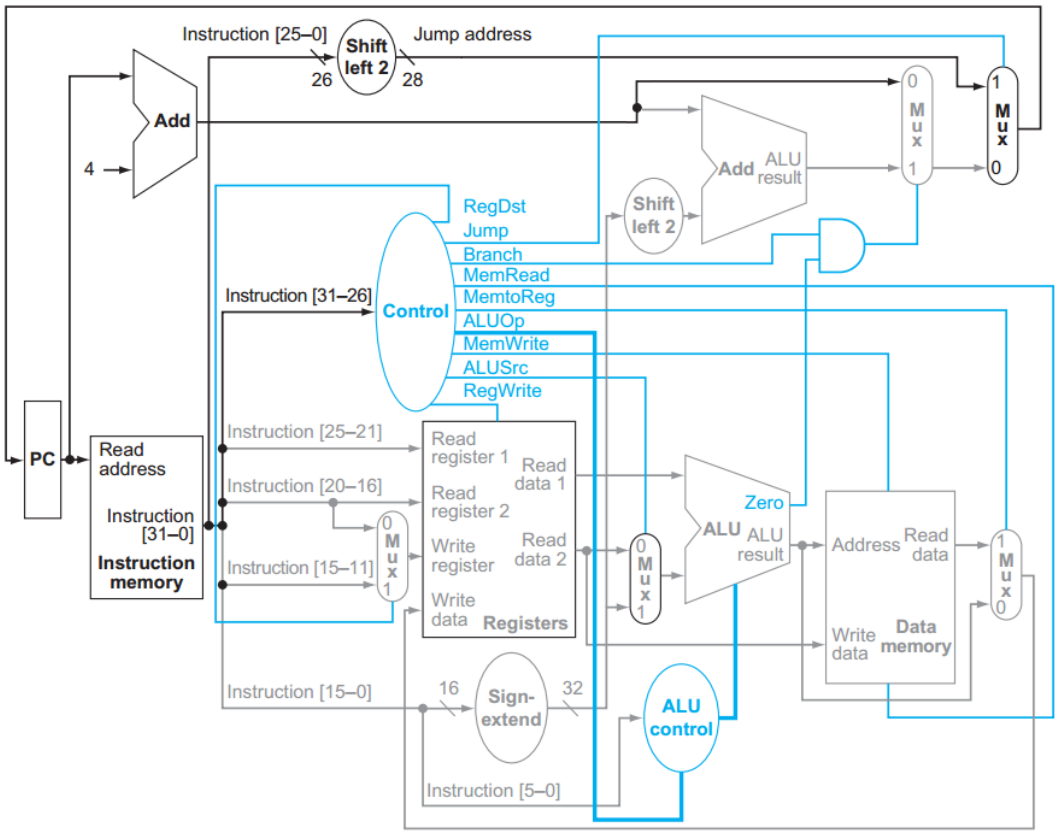


Figure 2: modified mips processor. For the purposes of this lab, PC is only 10 bits, so the jump address is only the lower 10 bits of the shifted instr[25:0].

Use Table 4 to modify control, ALUControl, and ALU files.

Table 4

Name	format	Instruction[31:26]	Instruction[5:0]	ALUop	alu_control
Add	R	000000	100000	10	0010
Addi	I	001000	-	00	0010
and	R	000000	100100	10	0000
andi	I	001100	-	11	0000
Beq	I	000100	-	01	0110
Lw	I	100011	-	00	0010
Nor	R	000000	100111	10	1100
Or	R	000000	100101	10	0001
Slt	R	000000	101010	10	0111
Sll	R	110000	000000	10	1000
Srl	R	110000	000010	10	1001
sra	R	110000	000011	10	1010
Sw	I	101011	-	00	0010
Sub	R	000000	100010	10	0110
xor	R	000000	100110	10	0100
Mult	R	000000	011000	10	0101
div	R	000000	011010	10	1011
jump	J	000010	-	-	-

4 Addressing

Pay attention that memories in MIPS are **byte addressable** as explained in the lecture. It is the reason that we increase program counter (pc) by 4. Also if you look at the instruction memory and data memory code you will see that we are not using the lower 2 bits of the address. This means the address is byte address and since we want to access words we can omit the lower 2 bits.

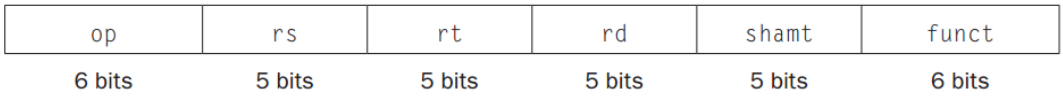
Note: In this lab, we didn't want to have a big instruction memory so we define the address line of the instruction memory with 10 bits. Hence, the PC is only 10 bits so after you calculate the address, use the lower 10 bits.

5 Test New Instructions

To test each new instruction, you need to find the binary code of that instruction and add it to the instruction memory. For example to test this instruction:

srl r20,r12,#5

We need to find the 32 bit binary code. *srl* is a R-type instruction so we have to follow this format:



Based on Table 4, the opcode field (instruction[31:26]) for *srl* is 110000. this instruction wants to shift $r12$, 5 bits to the right and save the result in $r20$. So, $r12$ is the

source register or rs field, $r20$ is the destination register or rd, and 5 is the shift amount or $shamt$. So, we put 01100 in rs, 10100 in rd, and 00101 in $shamt$ fields. The rt field for shift instructions is always zero. To fill the $func$ field look at the table 4 instruction[5:0] which is 000010 for srl. Hence, the 32 bit code for this instruction will be:

110000 01100 00000 10100 00101 000010

You can add this binary code to the instruction memory and see the result on the waveform.
Chapter 2 in the book is helpful to write your own instructions.

6 Assignment Deliverables

Your submission should include the following:

- (70 points) Block designs. (mips_32.v, datapath.v, control.v, ALUControl.v, instruction_mem.v, mux2.v, register_file.v, sign_extend.v, ALU.v, data_memory.v)

  (The rubric is similar to how the points is calculated in the testbench that will be released to the class in the second week of this assignment)

- (30 points) A report in **pdf** format. Follow the rules in the "sample report".
  There is no need to put the waveform for all the instructions in the instruction memory. For each of the new implemented instructions, put a screenshot of its waveform showing it works correctly and briefly explain why.

  You have 11 new instructions to design. Every instruction explanation will be 2 points except *beq* and *j*, where each is 6 points.

Note 1: Compress all files (11 files : 10 .v files + report) into zip and upload to the **CANVAS** before deadline.

Note 2: Use the code samples that are given. **The modules' names and the port names should exactly look like the code sample otherwise you lose points**.

Note 3: Please make sure that the modules' definitions in your project (input and output signals) are the same as listed below:

Code 1: mips_32.

```
module mips_32(
    input clk, reset,
    output[31:0] result
    );
```

Code 2: datapath.

```
module datapath(
    input clk, reset,
    input reg_dst, reg_write,
    input alu_src,
    input mem_read, mem_write,
    input mem_to_reg,
    input [3:0] ALU_Control,
    input branch, jump,
    output [31:0] datapath_result,
    output [5:0] inst_31_26,
    output [5:0] inst_5_0
    );
```

Code 3: control.

```
module control(
    input reset,
    input[5:0] opcode,
    output reg reg_dst, mem_to_reg,
    output reg [1:0] alu_op,
    output reg mem_read, mem_write, alu_src, reg_write, branch, jump
    );
```

Code 4: ALUControl.

```
module ALUControl(
    input [1:0] ALUOp,
    input [5:0] Function,
    output reg[3:0] ALU_Control);
```

Code 5: mux2.

```
module mux2 #(parameter mux_width= 32) (
    input [mux_width-1:0] a,b,
    input sel,
    output [mux_width-1:0] y
    );
```

Code 6: instruction_mem.

```
module instruction_mem(
    input [9:0] read_addr,
    output [31:0] data
    );
```

Code 7: register_file.

```
module register_file(
    input clk, reset,
    input reg_write_en,
    input [4:0] reg_write_dest,
    input [31:0] reg_write_data,
    input [4:0] reg_read_addr_1,
    input [4:0] reg_read_addr_2,
    output [31:0] reg_read_data_1,
    output [31:0] reg_read_data_2
    );
```

Code 8: sign_extend.

```
module sign_extend(
    input [15:0] sign_ex_in,
    output reg [31:0] sign_ex_out
    );
```

Code 9: ALU.

```
module ALU(
    input [31:0] a,
    input [31:0] b,
    input [3:0] alu_control,
    output zero,
    output reg [31:0] alu_result);
```

Code 10: data_memory.

```
module data_memory(
    input clk,
    input [31:0] mem_access_addr,
    input [31:0] mem_write_data,
    input mem_write_en,
    input mem_read_en,
    output [31:0] mem_read_data);

reg [31:0] ram [255:0];
```

Note 4: we are going to test the content of the data memory, so it is crucial that you don't change the following names: mips_32, datapath, data_memory, ram)

Appendix

To debug your processor for correct functionality, you need to do it in steps (on the waveform):

1. Check if the instruction is fetched from the instruction memory correctly.
2. Check if the controller selects the control bits for that instruction correctly.
3. Check if registers and immediate values are extracted correctly.
4. Check if ALU is doing the right job on its operands based on that instruction.
5. Check if write back multiplexer works correctly.