Final Project: MIPS-like Microprocessor

Objective:

The objective of this project is to design, simulate, and implement a simple 32-bit microprocessor with an instruction set that is similar to a MIPS. Note: some of the details are intentionally omitted. You must use what you have learned throughout the semester to complete the project. You are free to implement the MIPS in VHDL any way that you like, as long as it can execute the provided test programs.

Logistics:

As discussed in class, this is essentially a "mini-project". It will be worth 350 points (3.5x more than a normal lab). The grading is based on the completion of a list of deliverables. When completed, each deliverable will earn the student some amount of points toward the total score of 350. The list of deliverables, their due dates, and their worth in points will be described later.

General architecture for the MIPS Computer:

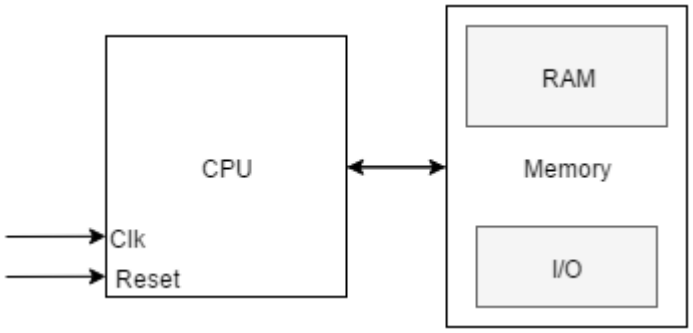


Figure 1. Overall architecture of the MIPS processor

The MIPS computer consist of the following:

- A 32-bit processor (CPU)
- A "memory module" that consists of a RAM and memory-mapped I/O
- The RAM consists of 256 32-bit words, mapped to address 0, and is initialized with a mif file that contains the program that will execute. Use the Altera 1-Port RAM megafunction (called altsyncram in earlier versions). The RAM uses word-aligned addresses, so you will need to remove the lower two bits of the 32-bit address when connecting to the RAM. In other words, for a 256-word RAM, the RAM address input would connect to (9 downto 2) of the 32-bit address. We aren't implementing load/store byte instructions, but if we did, you would use the lower two bits to select which of the 4 bytes of the 32-bit word to use.
- The I/O ports consist of two 32-bit input ports and one 32-bit output port with the following addresses. The output port is connected to the four 7-segment LEDs.
  INPORT0 $0000FFF8 INPORT1 $0000FFFC
  E.g. lw $s1, FFFC($zero) means $s1←(INPORT1)
  OUTPORT $0000FFFC
  E.g. sw $s1, FFFC($zero) means (OUTPORT) ← $s1

- Because the DE0 board has only 10 switches, each input port will share the same 10 switches. To load a value into each port, you will use two buttons as enable signals for the input ports. In other words, you would set the switches for the desired value on INPORT0, then press the enable button for INPORT0, which should store the switch settings to a register (the upper 22 bits should be set to 0). You would then change the switches for INPORT1, and then press a second button to enable INPORT1, which would store the input into a separate register.
- A separate reset, controlled by the third button, for the CPU and memory. Note that this reset should *not* reset the input ports. This separate reset is used to restart an application after changing the values of the input ports.

General architecture for the MIPS:

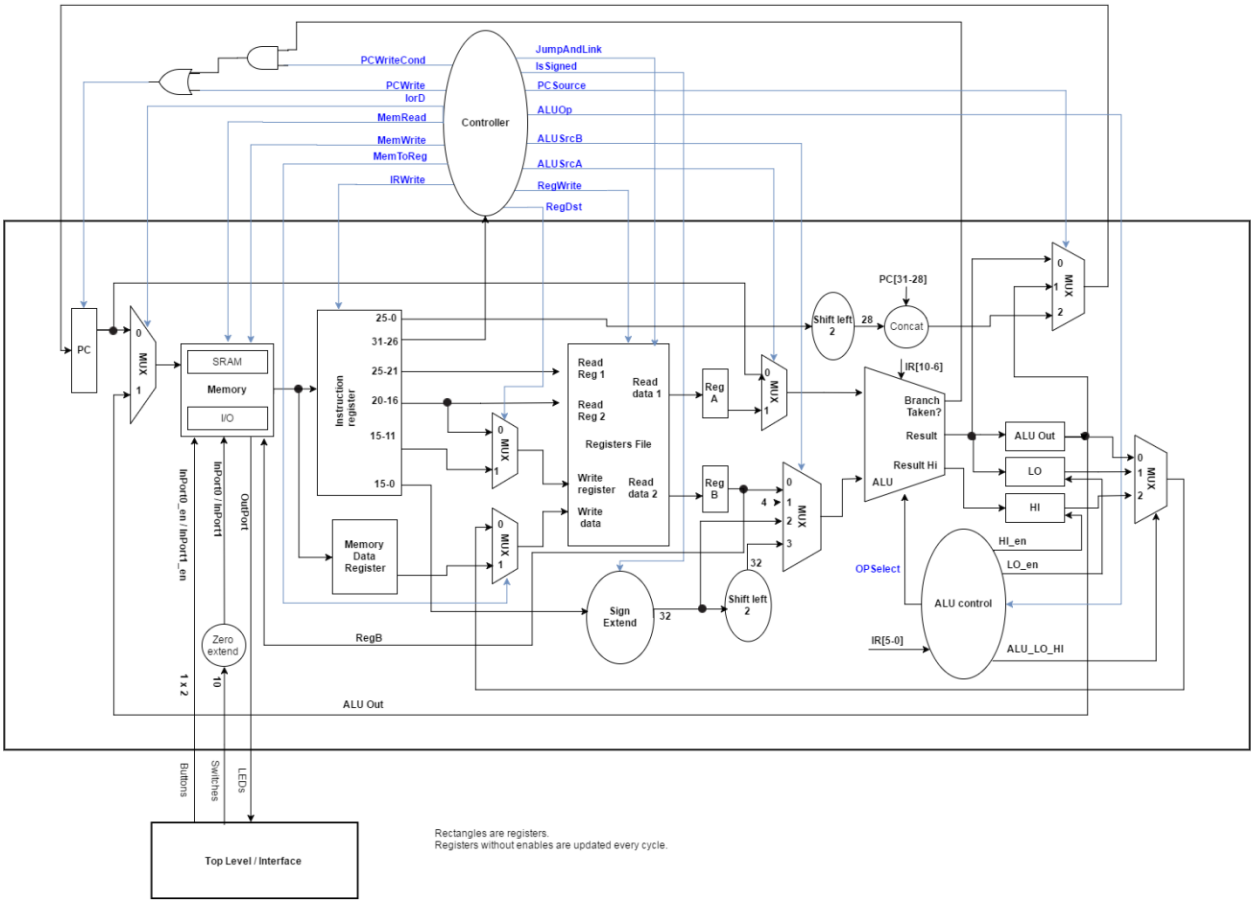


Figure 2. General architecture of the MIPS Computer.

The datapath consists of:

- **ALU** : performs all the necessary arithmetic/logic/shift operations required to implement the MIPS instruction set (see instruction set table at end of this document). The ALU also implements the conditions needed for branches and asserts the “Branch Taken” output if the condition is true. The ALU should have four inputs: two for the inputs to be processed, one for a shift amount (shown as IR[10-6]), and one for the operation select.

You can use whatever select values you want for the operations, but I would recommend looking over the encoding of the r-type instructions first to simplify the logic.

- **Register File**: 32 registers with two read ports and one write port.
- **IR:** The Instruction Register (IR) holds the instruction once it is fetched from memory
- **PC**: The Program Counter (PC) is a 32-bit register that contains the memory address of the next instruction to be executed.
- Some special-purpose registers, including **Data Memory Register, RegA, RegB, ALUout, HI,** and **LO.** These will be explained in lecture.
- **Controller** which controls all the datapath and the memory module. (The controller **<u>does not</u>** control writing to the **input ports**). Note that the ALU is controlled by a separate ALU Control unit that uses signals from both the main controller and the datapath. This will be explained in lecture. The design of the controllers is one of the main tasks of this project. You are welcome to add more control signals that are not shown in the datapath figure.
- **ALU Controller :** controls the all the ALU Operations. This logic is up to you to figure out, but it will become more clear after discussing the instructions in lecture.
- **Memory:** contains the RAM and memory-mapped I/O ports
- **Sign Extended:** convert a signed 16-bit input to its 32-bit representation when the signal "**isSigned**" is asserted.

<u>The controller signals:</u>

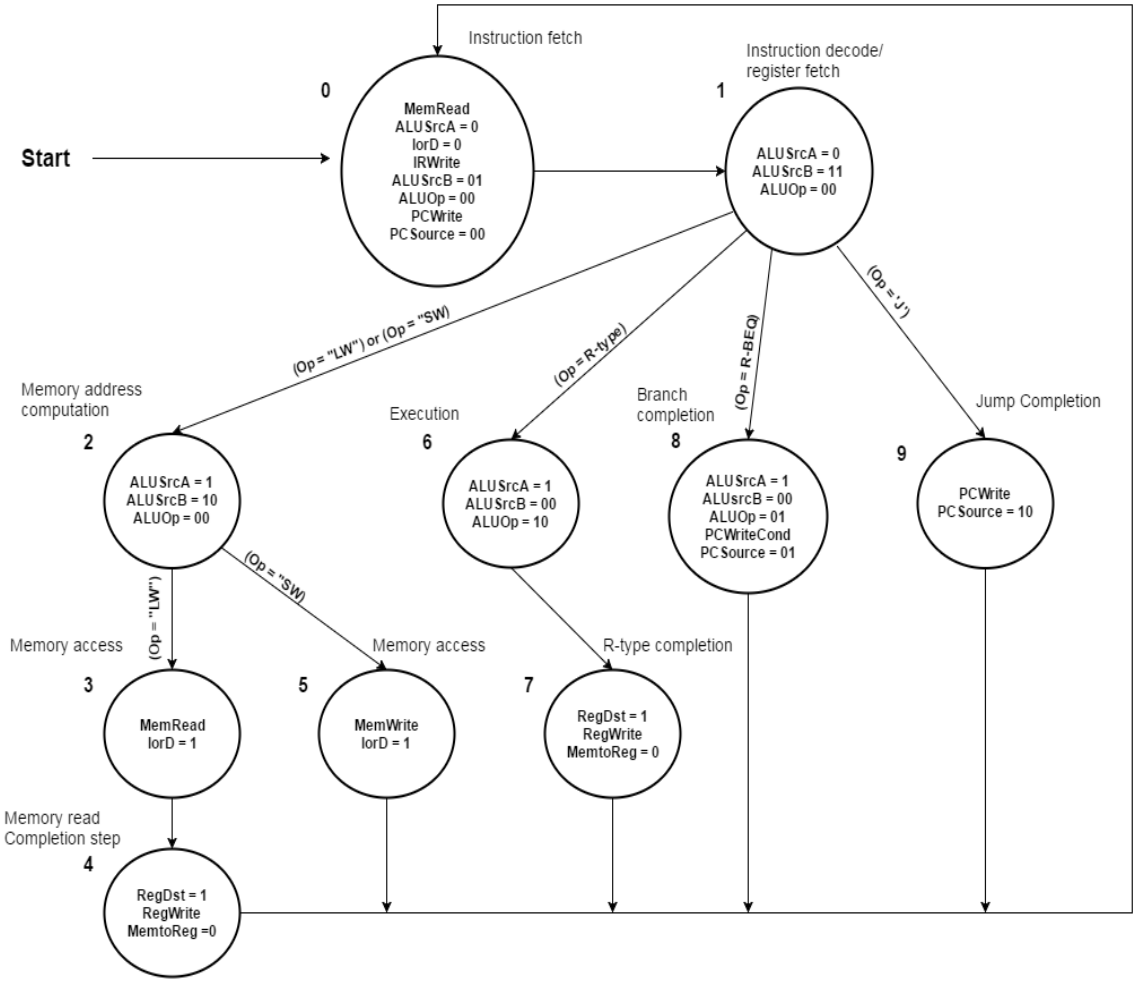
- **PCWrite:** enables the PC register.
- **PCWriteCond:** enables the PC register if the "**Branch**" signal is asserted.
- **IorD:** select between the PC or the ALU output as the memory address.
- **MemRead:** enables memory read.
- **MemWrite:** enables memory write.
- **MemToReg:** select between "**Memory data register**" or "**ALU output**" as input to **"write data**" signal.
- **IRWrite:** enables the instruction register.
- **JumpAndLink:** when asserted, $s31 will be selected as the write register.
- **IsSigned:** when asserted, "Sign Extended" will output a 32-bit sign extended representation of 16-bit input.
- **PCSource:** select between the "ALU output", "ALU OUT Reg", or a "shifted to left PC" as an input to PC.
- **ALUOp:** used by the ALU controller to determine the desired operation to be executed by the ALU. It is up to you to determine how to use this signal. There are many possible ways of implementing the required functionality.
- **ALUSrcA:** select between **RegA** or **Pc** as the input1 of the ALU.
- **ALUSrcB:** select between **RegB**, "**4**", **IR15-0**, or "**shifted IR15-0**" as the input2 of the ALU.
- **RegWrite :** enables the register file
- **RegDst:** select between **IR20-16** or **IR15-11** as the input to the **"Write Reg"**

<u>Other signals:</u>

- **IR31-26 (**the OPCode): Will be decoded by the controller to determine what instruction to execute.
- **IR5-0**: If the instruction is as R-type, this signal will be decoded by the ALU controller to determine the desired operation to be executed by the ALU.
- **IR10**-6: For shift instructions, this set of bits specifies the shift amount.
- **Other IR ranges are instruction specific and will be explained in lecture**.
- **OPSelect:** will be used by the ALU to execute the desired operation
- **Load_HI:** enables the **HI** register
- **Load_LO:** enables the **LO** register
- **Alu_LO_HI :** select between ALU out, LO, or Hi as the write data of register file.
- **Branch:** gets asserted if the branch condition is true.

Opcode fetch, decode, execute cycle for the CPU controller:

Note: - This is not the complete. You are to add more states and signals to successfully executes all the instructions. Note that the timing or your design may also differ based on the read-latency of your RAM.



Source: Patterson and Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd ed

Figure 3. General algorithm for designing a MIPS CPU

Execution steps

All instructions :

Step 1: - Fetch instruction, store in IR, PC = PC + 4

Step 2: - Decode instruction
- "Look ahead" steps: Read in rs and rt registers to A and B, respectively. Compute target branch address using lower 16 bits of instruction --> ALUOut

Memory access:

Step 3: - Compute memory address

Step 4: - If lw: Retrieve data from memory at specified address and place in MDR
- If sw: Write data (B register) to memory at specified address

Step 5 - (lw only): Write contents of MDR to specified register

R-type:

Step 3: - Perform specified operation --> ALUOut

Step 4: - Write ALUOut contents to specified register

Branch:

Step 3: - Compare two registers
- Use Zero/Branch output to determine if they are equal
- Determine if we branch to the address in ALUOut or to PC+4

Deliverables: (prepare to show to your TA)

For each deliverable, do the following:

- Create a neat drawing of your circuit, or a finite state machine for the controller.
- Submit your VHDL files on Canvas.
- **Have simulations prepared to demo the correct functionality.** These simulations should make it easy to see the functionality of each deliverable. Add annotations to explain. For larger simulations (e.g., multiply test case), selectively show some key parts of the waveform. **Turn in these simulations on Canvas along with your code.**
- On Canvas, there will be a submission link for each week's deliverables. I'd suggest creating a separate folder for each deliverable to make it easy to find your code. If you work ahead, turn in the deliverables in the specified weeks (not the week you finished it).

Part of the grading of the deliverable is your understanding/explanation of your design. Of course, blatant inability to explain your finite state machine and/or your code is evidence of cheating and will be dealt with as such.

NOTE: You must attend lab each week unless you have demoed all deliverables. Missing a lab will result in -20 points. Unless you are completely finished, you have to stay and work on the project with the help of your TA.

Week 1: At a minimum, you are to complete Deliverables 1 and 2 by the end of the lab.

Deliverable 1 (15 points): Design and simulation of the ALU. No demonstration on the UF-4712 board is necessary. Show the TA a simulation waveform that shows the correct operation of each operation. Show synthesis results verifying no latches. For week 1, use whatever select values you want for each ALU operation. Turn in all files and the simulation waveform on Canvas.

Extra Credit (10 points): create an exhaustive testbench that tests every possible ALU input combination using assert statements and show the TA that no assertions fail. For this exhaustive test, reduce the width of the ALU to 8 bits or your simulation will never finish.

Deliverable 2 (20 points): Design and simulation of the datapath and memory (RAM and ports). You must illustrate and explain to the TA the operation of each control signal that you are using for the datapath. At a minimum, you must show each component outputting a value using inputs from relevant components (e.g., the ALU should be tested with all the possible mux inputs). Your testbench should act similarly to the controller, but does not need to execute instructions. Turn in all files and the simulation on Canvas.

Week 2: At a minimum, you are to complete Deliverable 3 by the end of the lab.

Deliverable 3 (50 points): Initial design of the controller to support memory-access instructions (LW, SW), all R-type instructions, and all I-type instructions. Branch and jump instructions will not be tested because they require non-sequential execution. Create a MIF file that demonstrates loads and stores by loading from the input ports and displaying to the output port. Demoing the R-type and I-type instructions is up to you, but you can extend your MIF file to show that these instructions are working. Use multiple MIF files if necessary. Implement the fake halt instruction to prevent the MIPS from reading past the end of your MIF file. You may demonstrate the functionality in simulation or on the FPGA, but make sure to prepare waveforms in either case that demonstrate the correct functionality. Turn in all files and the simulation on Canvas.

Week 3: Turn in all files and the simulations on Canvas for each of the deliverables.

Deliverable 4 (100 points, 25 points each): Demonstrate test cases 1,2,4, and 7.

Deliverable 5 (65 points): Convert the GCD assembly code into a MIF file and demonstrate the correct functionality on the board.

Deliverable 6 (100 points): Demonstrate the correct functionality of the bubble_sort.mif on your board.

Selected Subset of MIPS Instructions (See Excel sheet for more details)

Instruction	OpCode (Hex)	Type	Example	Meaning
add - unsigned	0x00	R	addu $s1, $s2, $s3	$s1 = $s2 + $s3
add immediate unsigned	0x09	I	addiu $s1, $s2, IMM	$s1 = $s2 + IMM
sub unsigned	0x00	R	subu $s1, $s2, $s3	$s1 = $s2 - $s3
sub immediate unsigned	0x10 (not MIPS)	I	subiu $s1, $s2, IMM	$s1 = $s2 - IMM
mult	0x00	R	mult $s, $t	$LO= $s * $t
mult unsigned	0x00	R	multu $s, $t	$LO= $s * $t
and	0x00	R	and $s1, $s2, $s3	$s1 = $s2 and $s3
andi	0x0C	I	andi $s1, $s2, IMM	$s1 = $s2 and IMM
or	0x00	R	or $s1, $s2, $s3	$s1 = $s2 or $s3
ori	0x0D	I	ori $s1, $s2, IMM	$s1 = $s2 or IMM
xor	0x00	R	xor $s1, $s2, $s3	$s1 = $s2 xor $s3
xori	0x0E	I	xori $s1, $s2, IMM	$s1 = $s2 xor IMM
srl -shift right logical	0x00	R	srl $s1, $s2, H	$s1 = $s2 >> H (H is bits 10-6 of IR)
sll -shift left logical	0x00	R	sll $s1, $s2, H	$s1 = $s2 << H (H is bits 10-6 of IR)
sra -shift right arithmetic	0x00	R	sra $s1, $s2, H	See XLS sheet
slt -set on less than signed	0x00	R	slt $s1,$s2, $s3	$s1=1 if $s2 < $s3 else $s1=0
slti -set on less than immediate signed	0x0A	I	slti $s1,$s2, IMM	$s1=1 if $s2 < IMM else $s1=0
sltiu- set on less than immediate unsigned	0x0B	I	sltiu $s1,$s2, IMM	$s1=1 if $s2 < IMM else $s1=0
sltu - set on less than unsigned	0x00	R	sltu $s1,$s2, $s3	$s1=1 if $s2 < $s3 else $s1=0
mfhi -move from Hi	0x00	R	mfhi $s1	$s1= HI
mflo -move from LO	0x00	R	mflo $s1	$s1= LO
load word	0x23	I	lw $s1, offset($s2)	$s1 = RAM[$s2+offset]
store word	0x2B	I	sw $s1, offset($s2)	RAM[$s2+offset] = $s1
branch on equal	0x04	I	beq $s1,$s2, TARGET	if $s1=$s2, PC += TARGET
branch not equal	0x05	I	bne $s1,$s2, TARGET	if $s1/=$s2, PC += TARGET
Branch on Less Than or Equal to Zero	0x06	I	blez $s1, TARGET	if $s1 <= 0, PC += TARGET
Branch on Greater Than Zero	0x07	I	bgtz $s1, TARGET	if $s1 > 0, PC += TARGET
Branch on Less Than Zero	0x01	I	bltz $s1, TARGET	if $s1 < 0, PC += TARGET

Branch on Greater Than or Equal to Zero	0x01	I	bgez $s1, TARGET	if $s1 >= 0, PC += TARGET
jump to address	0x02	J	j TARGET	PC = TARGET
jump and link	0x03	J	jal TARGET	$ra = PC+8 and PC = TARGET
jump register	0x00	R	jr $ra	PC = $ra
Fake instruction	0xFF		Halt	Useful for week 2 deliverables