MIPS-4712 Project Description

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. 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 300 points (3 x 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 300. The list of deliverables, their due dates, and their worth in points will be described later.

Overview of the MIPS Computer:

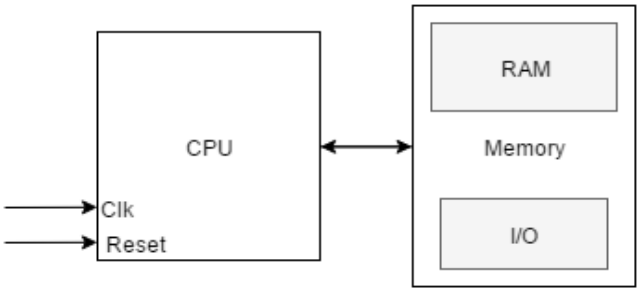


Figure 1. Overview of the MIPS processor

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 1-Port RAM component in the Quartus IP Catalog.
- The I/O ports consist of two 32-bit input ports and one 32-bit output port with the following addresses.

  INPORT0 $0000FFF8 INPORT1 $0000FFFC
  Ex: lw $s1, FFFC($zero) means $s1←(INPORT1)
  OUTPORT $0000FFFC
  Ex: sw $s1, FFFC($zero) means (OUTPORT) ← $s1

General architecture for the MIPS:

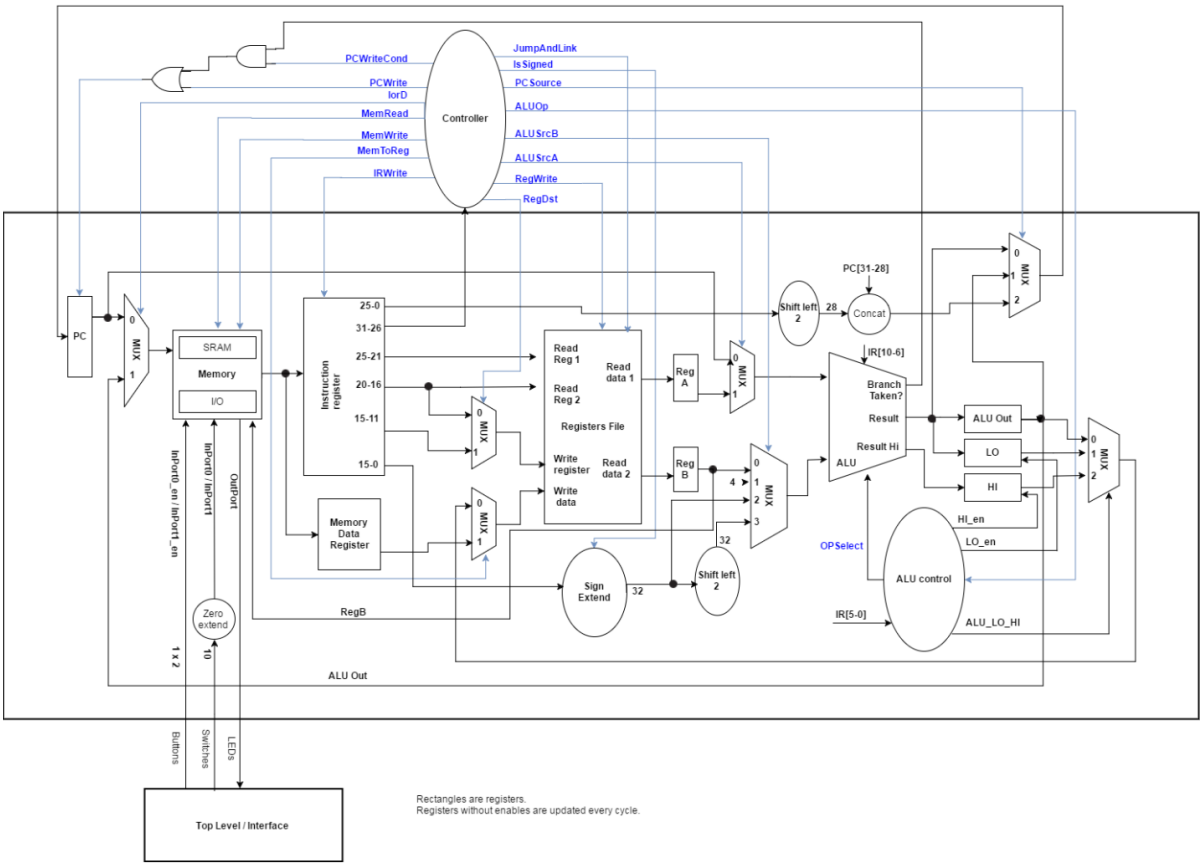


Figure 2. General architecture of the MIPS CPU (see included files for more detailed view).

Datapath (the main rectangle in the middle of Figure 2) 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.
- **Memory:** contains the RAM and memory-mapped I/O ports

- **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. Again, use the 1-Port RAM component in the Quartus IP Catalog. 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 are not 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.
- **ALU Control Unit:** some logic to determine the select code for ALU Operations. As described in lecture, this ALU Control Unit uses signals from both the Main Controller and the datapath.
- **Sign Extended:** converts a signed 16-bit input to its 32-bit representation when the signal "**isSigned**" is asserted.

Controller (see Figure 2): controls all the datapath and the memory module. (The Controller **does not** control writing to the **input ports**). More details later. The design of the Controller is one of the main tasks of this project. You are welcome to add more control signals that are not shown in Figure 2.

Controller signals:

- **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 Control Unit 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 input2 of the ALU.
- **RegWrite :** enables the register file
- **RegDst:** select between **IR20-16** or **IR15-11** as the input to the **"Write Reg"**

Other signals in the datapath:

- **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
- **HI_en:** enables the **HI** register
- **LO_en:** 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.

Input/Output Ports

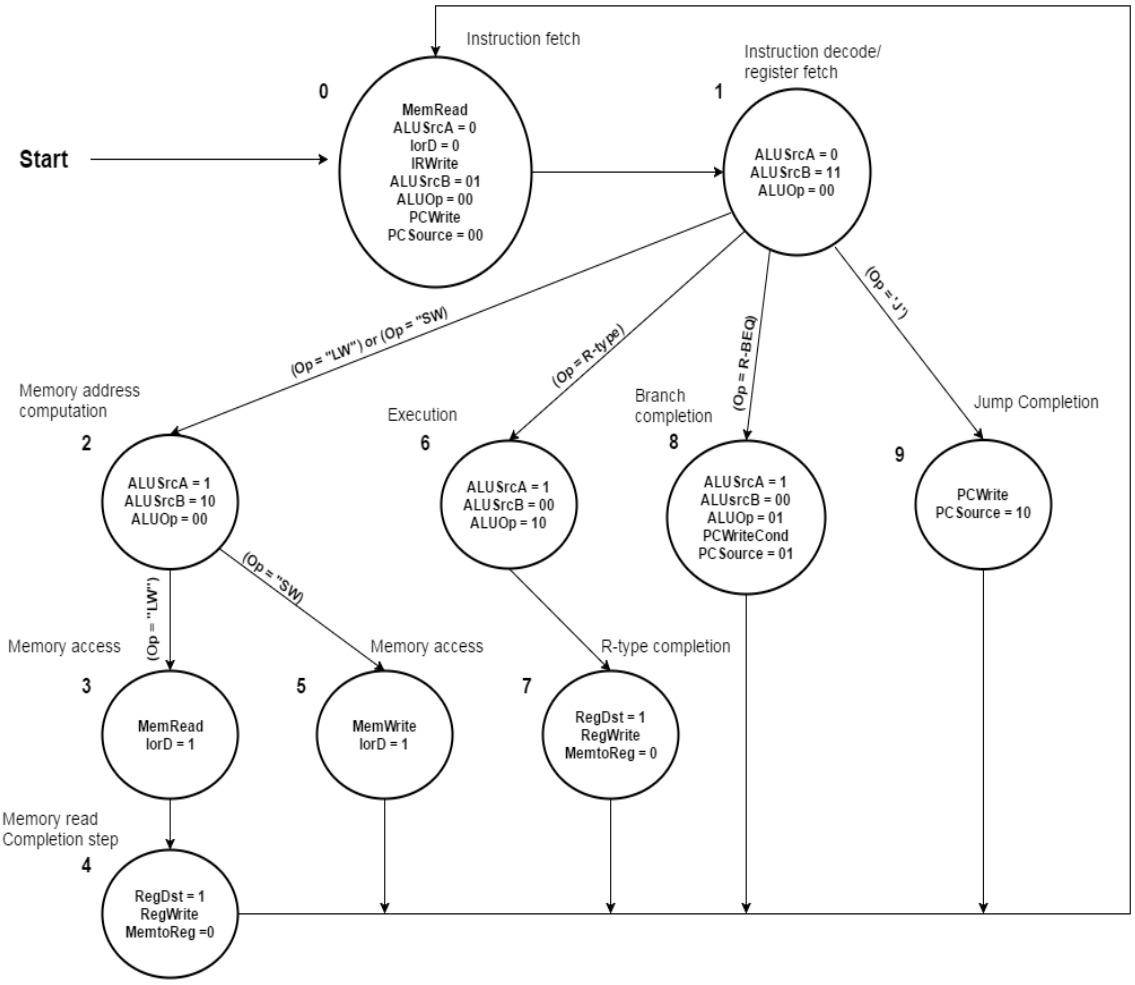
- Each input port will share the 9 switches, which means that you can only specify the lower 9 bits of data loaded through the ports. Because the ports share the same switches, you can only specify one port at a time.
- The 10^{th} switch will be used to specify which port is loaded. To load a value into each port, you will use one of the two buttons as an enable signal for the input ports. For example, you would set the lower 9 switches for the desired value on INPORT0, set the 10^{th} switch to low to select INPORT0, then press the enable button, which should store the switch settings to a register (the upper 23 bits should be set to 0). Note that you are responsible for determining the appropriate enable logic. You would do the same thing for INPORT1, but you would need to set the 10^{th} switch to high.
- The output port is connected to the four 7-segment LEDs.
- Reset for the CPU and memory is controlled by the second button. Note that this reset should *not* reset the input ports in your circuit. This separate reset is used to restart an application after changing the values of the input ports. A processor would normally not be designed this way, but we have to work around the limited switches and buttons on the board.

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

Figure 3 shows the general algorithm for the Controller of a MIPS CPU. Note that this figure is not complete and cannot used directly for the MIPS-4712 datapath. Figure 3 is intended to be a basic template for you to follow. You can add more states and signals as necessary. Note that the timing or your design may also differ.

Figure 3. General algorithm for designing a controller for a MIPS CPU

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



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: (submission to Canvas & demonstration)

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 deliverable. I would suggest creating a separate folder for each deliverable to make it easy to find your code.

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.

Deliverable 1 (40 points): Design and simulation of the ALU. No demonstration on the board is necessary. Create a testbench that when simulated shows (use ALU.A and ALU.B unless specified otherwise):

- Addition of 10 + 15 (i.e., A=10, B=15)
- Subtraction of 25 - 10
- Mult (signed) of 10 * -4. Make sure to show both the Result and Result Hi outputs shown in the datapath.
- Mult (unsigned) of 65536 * 131072. Make sure to show both the Result and Result Hi outputs shown in the datapath.
- And of 0x0000FFFF and 0xFFFF1234
- Shift ALU.A right logical of 0x0000000F by 4 (shift amount (00100 here) is inputted through the ALU input IR(10-6]).
- Shift ALU.A right arithmetic of 0xF0000008 by 1
- Shift ALU.A right arithmetic of 0x00000008 by 1
- Set on less than using A=10 and B=15 (for “set less than” instructions slt, slti, slu, and sltiu). ALU.Result should be $00000001.
- Set on less than using A=15 and B=10 (for “set less than” instructions slt, slti, slu, and sltiu). ALU.Result should be $00000000.
- Branch Taken output = ‘0’ for for 5 <= 0; ALU.A=5 (for blez instruction)
- Branch Taken output = ‘1’ for for 5 > 0; ALU.A=5 (for bgtz instruction)

Show synthesis results verifying no latches. Use whatever select values you want for each ALU operation, but make sure to have a table that specifies a mapping between select values and operations to make it easy for the TA. Turn in all files and the simulation waveform on Canvas.

Deliverable 2 (35 points): Design and simulation of the memory (RAM and ports). Create a testbench that demonstrates the following:

- Write 0x0A0A0A0A to byte address 0x00000000
- Write 0xF0F0F0F0 to byte address 0x00000004
- Read from byte address 0x00000000 (should show 0x0A0A0A0A on read data output)

- Read from byte address 0x00000001 (should show 0x0A0A0A0A on read data output)
- Read from byte address 0x00000004 (should show 0xF0F0F0F0 on read data output)
- Read from byte address 0x00000005 (should show 0xF0F0F0F0 on read data output)
- Write 0x00001111 to the outport (should see value appear on outport)
- Load 0x00010000 into inport 0
- Load 0x00000001 into inport 1
- Read from inport 0 (should show 0x00010000 on read data output)
- Read from inport 1 (should show 0x00000001 on read data output)

Deliverable 3 (25 points): Design and synthesis of a "skeleton" datapath.

- All components in the datapath must be created as an ENTITY.
- Create ARCHITECTURE for all the "standard" components in the datapath such as MUXs, registers, and any components that you know how to design.
- Create "placeholders" ARCHITECTURE for other components such as Register file, ALU control, etc. (any components that you do not know how to design yet).
- PORT MAP all the components together to make a complete datapath.
- Compile and synthesis the datapaath.
- Show the datapath to the TA using RTL viewer.

Deliverable 4 (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.

A MIF is provided for demonstrating a simulation of Deliverable 4, and instructions will be provided by each TA about how to prepare your simulation to show that the design is working.

It is highly recommended that you create your own MIFs to do more thorough testing. Your MIF should demonstrate loads and stores by loading from the input ports and displaying to the output port. For the R-type and I-type instructions, you can extend your MIF 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.

Deliverable 5 (100 points, 25 points each): Demonstrate test cases 1, 2, 4, and 7.

Deliverable 6 (50 points): Convert the GCD assembly code into a MIF file and demonstrate the correct functionality on the 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	$HI,$LO= $s * $t
mult unsigned	0x00	R	multu $s, $t	$HI,$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 += 4+TARGET
branch not equal	0x05	I	bne $s1,$s2, TARGET	if $s1/=$s2, PC += 4+TARGET
Branch on Less Than or Equal to Zero	0x06	I	blez $s1, TARGET	if $s1 <= 0, PC += 4+TARGET
Branch on Greater Than Zero	0x07	I	bgtz $s1, TARGET	if $s1 > 0, PC += 4+TARGET
Branch on Less Than Zero	0x01	I	bltz $s1, TARGET	if $s1 < 0, PC += 4+TARGET
Branch on Greater Than or Equal to Zero	0x01	I	bgez $s1, TARGET	if $s1 >= 0, PC += 4+TARGET
jump to address	0x02	J	j TARGET	PC = TARGET
jump and link	0x03	J	jal TARGET	$ra = PC+4 and PC = TARGET
jump register	0x00	R	jr $ra	PC = $ra
Fake instruction	0x3F		Halt	Useful for week 2 deliverables