

Electrical and Computer Engineering (ECE)

**Computer Organization & Architecture**

***CSE332\_ Section-01***

**ISA Design**

**Project Milestone 01**

**Faculty: Tanzilur Rahman (Tnr)**

**Group-0**

|  |  |  |
| --- | --- | --- |
| Serial  Number | NAME | **ID** |
| 01 |  |  |
| 02 |  |  |
| 03 |  |  |
| 04 |  |  |

***Objective:***

1. To design a new 12 bit single-cycle CPU that has separate Data and Instruction Memory.
2. To generate a machine code from a file containing assembly language. The assembler reads a program written in an assembly language, then translates it into binary code and generates output file containing machine code.

***Design:***

In our design, we allocate 4bits for Opcode, 4bits for source register and another 4 for destination register.

**Bit Size:** (11-0bits)

4bits (11-8) 4bits (7-4) 4bits (3-0)

|  |  |  |
| --- | --- | --- |
| Opcode | Destination | Source Operand |

* *Operands:* As our operands, we have proposed 2 individual registers where each of the registers contain 24(16 bit memory).
* *Types of Operands:* Because of using only 2 operands, our operands are register based where we can load and store values as temporary or fixed memory as well as assign immediate values and make paths with non-volatile memory.
* *Operations:* Total bit reserved for our operation code is 4. By which, we can perform 24=16 operations. That means, we are allowed to set 16 different instructions to perform.
* *Types of Operation:* Our design focusing on the following three categories of programs-

1. *Simple arithmetic & logic operations*
2. *Programs that require checking conditions*
3. *Loop type of programs*

That’s why we need this 5 category of operations-

1. Arithmetic
2. Logical
3. Data Transfer
4. Conditional Branch
5. Unconditional Jump

For completing these operations, we have to use some distinguish instructional operations. For Arithmetic (add, addi, sub), for logical (and, or, sll, srl), for data transfer we must use (lw, sw, in, out, disp), for condition check (beq, slt, slti) and for unconditional (we use -jump).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Category/**  **Instruction Type** | **Operation** | **Format** | **Opcode** | **Example/Syntax** | **Meaning/Comments** |
| Arithmetic | ADD | R | 0000 | add $s0, $s1 | $s0 = $s0 + $s1 |
| Addi | I | 0001 | addi $s0, 20 | $s0 = $s0 + 20 |
| Sub | R | 0010 | sub $s0, $s1 | $s0 = $s0 - $s1 |
| Logical | AND | R | 0011 | and $s0 , $s1 | $s0 = $s0&$s1 |
| Sll | R | 0100 | sll $s0, 4 | $s0=$t0 (shift left 2 times) |
| Data Transfer | LW | I | 0101 | lw $s0, 2 | $s0 = mem[$t0 + offset] |
| SW | I | 0110 | sw $s0, 2 | mem[$t0 + offset] = $s0 |
| IN | I | 0111 | in $s0 | Takes value from $in and pass to $s0 |
| OUT | I | 1000 | out $s1 | Load value of $out to $s1 |
| Disp | I | 1001 | disp | Display value from $out |
| Conditional | Beq | I | 1010 | beq $s0, L | If($s0 == $t0) then jump to L |
| Bne | I | 1011 | bne $s0,L | If($s0!= $t0) then jump to L |
| SLT | I | 1100 | slt $s0, $s1 | if ($s0 < $s1) then $t0 = 1  Else $t0 = 0 |
| SLTi | I | 1101 | slti $s0, 2 | if ($s0 < 2) then $t0 = 1 Else $t0 = 0 |
| Bge | I | 1110 | bge $s0,L | If($s0 >= $t0) then jump to L |
| Unconditional | J | J | 1111 | J L | Jump to level L |

**Formats**

In our design, we use 3 format of RISC architecture, they are-

* + R- Type
  + I -Type
  + J –Type

*R-Type MIPS Format for our ISA:*

|  |  |  |
| --- | --- | --- |
| Opcode | Source Register | Target Register |
| 4bit | 4bit | 4bit |

*I-Type MIPS Format for our ISA:*

|  |  |  |
| --- | --- | --- |
| Opcode | Source Register | Immediate |
| 4bit | 4bit | 4bit |

*J-Type MIPS Format for our ISA:*

|  |  |  |
| --- | --- | --- |
| Opcode | Target |  |
| 4bit | 8bit |  |

***List of Register:***

As we have mentioned, our source operands’ bit size are 4; so we have designed with 24 = 16 registers for each operands.

We have selected registers from $zero, $s0-$s6 and $t0-$t7 and assigned 4 bits for each of the register.

|  |  |  |  |
| --- | --- | --- | --- |
| **Register Number** | **Register Name** | **Binary Value** | **Commands** |
| 0 | $zero | 0000 | Ground to zero |
| 1 | $s0 | 0001 | Save |
| 2 | $s1 | 0010 | Save |
| 3 | $s2 | 0011 | Save |
| 4 | $s3 | 0100 | Save |
| 5 | $s4 | 0101 | Save |
| 6 | $s5 | 0110 | Save |
| 7 | $s6 | 0111 | Save |
| 8 | $t0 | 1000 | Temporary data  + Default Register |
| 9 | $t1 | 1001 | Temporary data |
| 10 | $t2 | 1010 | Temporary data |
| 11 | $t3 | 1011 | Temporary data |
| 12 | $t4 | 1100 | Temporary data |
| 13 | $t5 | 1101 | Temporary data |
| 14 | $t6 | 1110 | Temporary data |
| 15 | $t7 | 1111 | Temporary data |

***Addressing Modes:***

* *Register Addressing* – add $s1,$s2

> Operation - $s0 = $s0 + $s1

* *Immediate addressing* – addi $s4,11

> Operation - $s0 = $s0 + 11

* *Base addressing* - lw $s0, 2($s1)

> Operation - $s0 = mem[$s1 + offset]

* *Simple arithmetic*
* a = a + b , where a is in $s0, b is in $s1

Syntax: add $s0, $s1 Operation: $s0 = $s0 + $s1

//Here it adds the content of the source register 1($s0) to the contents of the source register 2($s1) and saves it in the destination register($s0).

* *Logic Operation*
* a = a AND b , where a is in $s0, b is in $s1

Syntax: and $s0, $s1 Operation : $s0 = $s0 & $s1

//It does bit by bit logical AND operation between two source registers contents.

* *Programs that require checking conditions*
* if (i == 10) , where i is in $s2

i = i + 4

else

i = i - 4

addi $t0, 10 \ $t0 = 10

beq $s2, L \ Here, compare $s2 = i with $t0 = 10. If equal then go to level L

addi $s1, -4 \ i = i – 4, $s2 = $s2 - 4

Jump exit \ Jump to exit, end of program

L: addi $s2, 4 \ i = i + 4, $s2 = $s2 + 4

Exit:

* *Loop type of program****s***
* for (i = 0, i < 5, i++) , where i is in $s2, a is in $s0

a = a + 4

sub $s2, $s2 \ $s2 = $s2 - $s2, clear register and initialize i to zero

sub $t0, $t0 \ clear temporary register

sub $s0, $s0 \ clear register where a is in

L1: slti $s2, 5 \ if (s2 < 5) then t0 = 0 else t0 = 1

beq $zero, Exit \ compare t0 with zero if equal then exit and end program

addi $s0, 4 \ $s0 = $s0 + 4

addi $s2, 1 \ increment i by 1

J L1 \ jump to level and repeat loop

***Benchmark:***

For this section, we have to test our CPU with multiple numbers of Instructions as suggested. We have to give input to all the instructions to solve a problem—for instance, a loop. All the instructions are then converted into a Hexadecimal format. Then we need to save the text file and loaded it on our processor's Instruction memory to run the instructions one after another.

But without instruction memory and ALU, we are not able to run those steps. But from ISA, we completely cover those instructions to run our desired programs.