**Introduction:**

The task of this project is to design a new 16 bit RISC type of CPU that has separate Data and Instruction Memory.The end product of this CPU is that, we will be able to do a significant amount of important operations successfully.

**Design Specification:**

We used a Register based architecture, where we will write and read data on Register taking from Memory and re-sending it back to memory.

We will be using maximum three Operands. Operands are registers on which we will be doing operations and save to or fetch from register. The addressing field is 4-bit long, therefore there can be a maximum of 16 registers.

We will do 16 operations based on our developed architecture. These operations will be used for various arithmetic calculations, conditional check, and jump. Furthermore, all the operations are mentioned in “Instructions Table” Section.

We chose 3 formats which have been described at “Formats” section.

List of the registers are illustrated at “Register Table” section with their naming and functionalities.

**Formats:**

We have chosen 3 formats.

1. R - type instruction
2. I - type instruction
3. J - type instruction

R - type instruction is basically for operations like add, sub, etc. where we will be operating on 2 registers and keeping the result on a 3rd register. I - type instruction is basically for operations like addi, andi etc. where we will be operating on a single register and on an integer (4 bit binary value max). J - type instruction is basically for unconditional jump where we will be jumping to a particular register along with an integer value(12 bit binary value max) which will be indicating the amount of shift.

**R-TYPE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode**  **4-bit** | **Rs**  **4-bit** | **Rt**  **4-bit** | **Rd**  **4-bit** |

**I-TYPE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode**  **4-bit** | **Rs**  **4-bit** | **Rt**  **4-bit** | **Immediate**  **4-bit** |

**J-TYPE:**

|  |  |
| --- | --- |
| **Opcode**  **4-bit** | **Immediate**  **12-bit** |

**Reasoning for the above design type:**

At first, we are adopting a register based architecture. The only other type of architecture is accumulator based. We didn’t use the accumulator based architecture because with it, pipelining is impossible. And within that time, we would be able to do more operations using register based. Hence, gain more speed.

Now, comes why we used 3 types of formats. The reason is that, we will be using 3 types of operations - 1) where there will be need of integers like addi, andi etc., 2) some won’t need them like, add, sub, and etc. and 3) unconditional jump. In order to encompass these 3 types of operations, we are using 3 types of formats and not more than that.

Our CPU must be 16-bit in construction, so our instructions must not exceed 16 bits in length. This reduces the amount of flexibility we have in terms of formats. Reducing operation code length to 3 bits or increasing register bits does not help us because:

1. Opcode with 3 bits allow us a maximum of 8 operations. Due to that, we won’t be able to use many important operations. The fact that we are making a 16 bit RISC type CPU has already reduced our operations count. But it is true we will be able to add more registers.
2. If we decrease the number of operands by making each operand 2/3 bits, we get only 4/8 registers. That reduces our capabilities of storing data on registers. But it is true that we get the opportunity of adding more operations.

**Register Table:**

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Register Function** | **Binary Value** |
| $Zero | Zero Register | 0000 |
| $s0 | Storage Register 1 | 0001 |
| $s1 | Storage Register 2 | 0010 |
| $s2 | Storage Register 3 | 0011 |
| $s3 | Storage Register 4 | 0100 |
| $s4 | Storage Register 5 | 0101 |
| $s5 | Storage Register 6 | 0110 |
| $t0 | Temporary Register 1 | 0111 |
| $t1 | Temporary Register 2 | 1000 |
| $t2 | Temporary Register 3 | 1001 |
| $t3 | Temporary Register 4 | 1010 |
| $t4 | Temporary Register 5 | 1011 |
| $gp | Global Pointer | 1100 |
| $sp | Stack Pointer | 1101 |
| $fp | Frame Pointer | 1110 |
| $ra | Return Address | 1111 |

**Instructions Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No** | **Instruction** | **Name** | **Type** | **Basic Syntax** | **Example** |
| **1** | add | Addition | R | add [destination], [2 register for summation] | Add $s1, $s2, $s3 |
| **2** | sub | Subtraction | R | sub [destination], [2 register for subtraction] | Sub $s1,$s2, $s3 |
| **3** | and | And Operation | R | and [destination], [2 register for and operation] | And $s1,$s2, $s3 |
| **4** | or | Or Operation | R | or [destination], [2 register for or operation] | Or $s1,$s2, $s3 |
| **5** | nor | Nor Operation | R | nor [destination], [2 register for nor operation] | Nor $s1,$s2, $s3 |
| **6** | sll | Shift Left Logical | R | sll [destination], [source], immediate value | Sll $s1,$s2, 10 |
| **7** | srl | Shift Right Logical | R | srl [destination], [source], [ immediate value] | Srl $s1,$s2, 10 |
| **8** | slt | Set on Less Than | R | slt [destination], [2 register for lesser than operation] | Slt $s1,$s2, $s3 |
| **9** | Andi | And immediate | I | andi [destination], [source], [ immediate value] | andi $s1,$s2, 20 |
| **10** | Ori | Or immediate | I | ori [destination], [source], [ immediate value] | ori $s1,$s2, 20 |
| **11** | Addi | Add immediate | I | addi [destination], [source], [ immediate value] | Sub $s1,$s2, 15 |
| **12** | Lw | Load Word | I | lw [destination], immediate value(source) | Lw $s1, 20($s2) |
| **13** | Sw | Store Word | I | sw [destination], immediate value(source) | sw $s1, 20($s2) |
| **14** | Beq | Branch Equal | I | beq [destination], [source], [ immediate value] | Beq $s1, $s2,25 |
| **15** | Mult | Multiply | R | mult [destination], [2 registers for multiplication] | Mult $s1, $s2, $s3 |
| **16** | J | Jump | J | j [immediate value] | J 2500 |

**R-Type OPCodes:**

|  |  |
| --- | --- |
| **OPCODE** | **Binary Representation** |
| Add | 0000 |
| Sub | 0001 |
| And | 0010 |
| Or | 0011 |
| Nor | 0100 |
| Slt | 0101 |
| Sll | 0110 |
| Srl | 0111 |
| Mult | 1000 |

**I-Type OPCodes:**

|  |  |
| --- | --- |
| **Opcode** | **Binary Representation** |
| Addi | 1001 |
| Lw | 1010 |
| Sw | 1011 |
| Bne | 1100 |
| Andi | 1101 |
| Ori | 1110 |

**J-Type OPCodes:**

|  |  |
| --- | --- |
| **Opcode** | **Binary Representation** |
| J | 1111 |

**Instruction description:**

1. We aim to design a 16-bit CPU that will be able to run loops, conditional statements, access memory contents in an array data structure, initializing registers with values and a combination of these operations.

2. We have omitted the MULT (multiplication operation) in Arithmetic Operations section because we believe it would add complexity to the design when it is unnecessary to complicate things.

3. We have not kept a SUBi because the same effect can be achieved by ADDi with negative constant.

4. We need Load word (lw) and Store word (sw) instructions since we are using a register based architecture. These instructions help us to control the data to and from the memory and register.

5. We have only kept NOR operation because it is a universal operation. We can implement all other logical operations using NOR.

6. We have kept Branch if not equal (BNE) only and not also Branch if equal (BEQ) because it is a redundant practise to keep both of them at the same time. All operations can be done with BEQ or BNE only.

**QUESTIONS AND ANSWERS**

**ANS. TO THE Q. NO. 1**

Base address of A is considered in $t1 and address of B in $t2

A[1] = A[4] + B[4]

A[2] = A[4] && B[4]

A[3] = A[4] || B[4]

lw $t3, 16($t1)

lw $t4, 16($t2)

add $t5, $t3, $t4

and $t6, $t3, $t4

or $t7, $t3, $t4

sw $t5, 4($t1)

sw $t6, 8($t1)

sw $t7, 12($t1)