****

**Department of Electrical & Computer Engineering**

**N**orth **S**outh **U**niversity

# PROJECT Title: ISA Design

**Course Code:** **CSE332**

**Course name: Computer Organization and Architecture**

**Section – 02**

**Summer 2022**

**Submitted to:** Dr. Tanzilur Rahman (TnR)

Assistant Professor,

Department of Electrical & Computer Engineering,

**N**orth **S**outh **U**niversity

**Submitted by:** Group Members

|  |  |  |
| --- | --- | --- |
|  | **Name** | **ID** |
| **1.** | Most Airin Akter | 1812770042 |
| **2.** | Anusree Roy | 2011364042 |
| **3.** | Nayeema Islam | 2031653642 |

***Introduction:***

Our task is to design an ISA for a 10 bit RISC type CPU.

**Objectives:**

Our objectives is to design a 10 Bit ISA which can solve particular problems i.e. arithmetic operations, logical operations branching and handling loops. Here we are going to show the detail design of that ISA.

**How many Operands?**

* There are three operands, which we represented as **d**, **s and t**.

**Type of Operands?**

* Register based
* Memory based

**3.How many Operations? Why?**

We allocated 4 bits for the opcode, so the number of instructions can be executed is 2^4 or 16 .

**4 .Types of Operations?**

There will be in total five different types operations. The categories are:

- Arithmetic

- Logical

- Data transfer

- Conditional Branch

- Unconditional Branch .There are 12 different operations. The details are given below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Category** | **Instructions** | **Syntax** | **Meaning** | **Format** | **Opcode** |
| Arithmetic | Add | add $s0, $s1 | $s0 = $s1 + $s0 | R | 0000 |
| Subtract | sub $s0, $s1 | $s0 = $s1 - $s0 | R | 0001 |
| Add Immediate | addi $s0, offset | $s0 = $s0 + constant | I | 0010 |
| Logical | AND | and $s0, $s1 | $s0 = $s0 & $s1 | R | 0011 |
| OR | or $s0, $s1 | $s0 = $s0 | $s1 | R | 0100 |
|
| Conditional  Branch | Branch on equal | beq $s0, target | If ($s0 == $s1)  Go to target address | I | 0101 |
|
| Set on less than | slti $s0, offset | If ($s0 < constant) $s3=1;  Else $s3= 0; | I | 0110 |
| Unconditional  Jump | Jump | j target | Jump to target address | J | 0111 |
| Data Transfer | Load word | lw offset($s0) | $s1 = Memory[$s0 + constant] | I | 1000 |
| Store word | sw offset($s0) | Memory[$s0 + constant] = $s1 | I | 1001 |
|
|

**Instruction Description:**

add: It adds two registers and stores the result in destination register.

∙        Operation: $s0 = $s0 + $s1

∙        Syntax:  add $s0, $s1

sub: It subtracts two registers and stores the result in destination register.

∙        Operation: $s0 = $s0 - $s1

∙        Syntax: sub $s0, $s1

addi: It adds a value from register with an integer value and stores the result in destination register.

∙        Operation: $s0 = $s0 + offset

∙        Syntax: addi $s0, offset

and: It AND’s two register values and stores the result in destination register. Basically, it sets some bits to 0.

∙        Operation:  $s0 = $s0 && $s5

∙        Syntax: and $s0, $s5

or: It OR’s two register values and stores the result in destination register. Basically, it sets some bits to 1.

∙        Operation:  $s0 = $s0 || $s5

∙        Syntax: or $s0, $s5

nor: It NOR’s two register values and stores the result in destination register. Sometimes we use nor to get NOT of register value.

∙        Operation:  $s0 = $s0 nor $s1

∙        Syntax: nor $s0, $s1

beq: It checks whether the values of two registers are same or not. If it’s same it performs the operation located in the address at offset value.

∙        Operation: if ($s0==$s6) jump to offset

else goto next line

∙        Syntax: beq $s0, offset

bne: It checks whether the values of two registers are same or not. If it’s not same it performs the operation located in the address at offset value.

∙        Operation: if ($s0!=$s6) jump to offset

else goto next line

∙        Syntax: bne $s0, offset

slti: If $s0 is less than offset, $sp is set to one. It gets zero otherwise.

∙        Operation:   if $s0 < offset $s3= 1

else $s3 = 0

∙        Syntax: slti $s0, offset

lw: It loads required value from the memory and write it back into the register.

∙        Operation: $s1 = MEM[$s0 + offset]

∙        Syntax:  lw offset($s0)

sw: It stores specific value from register to memory.

∙        Operation: MEM[$s0 + offset] = $s1

∙        Syntax:   sw offset($s0)

J: Jumps to the calculated address.

∙        Operation: PC = nPC

∙        Syntax: j target

**5 . How many Formats?**

We would like to use 2 formats for our ISA.

Register type – (R-Type)

Immediate Type – (I- Type)

Since Jump operation can be done in I type format by keeping rs , rt null , we don’t need J type format .

*R-Type ISA Format:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Op code** | **rs** | **rt** | **rd** |
| 4 bits | 2 bits | 2 bits | 2 bits |

*I-Type ISA Format:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Op code** | **rs** | **rt** | **immediate** |
| 4 bits | 2 bits | 2 bits | 2 bits |

***J-Type ISA Format:***

|  |  |
| --- | --- |
| **Op code** | **immediate** |
| **4 bits** | **6 bit** |

**6 . List of Registers:**

We’ve used 6 registers and assigned 2 bit for each of the registers. The Register we have selected are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Register Number** | **Register Name** | **Registers in binary** | **Comment** |
| $0 | $zero | 00 | This register contains 0. |
| $1 | s1 | 01 | General purpose register. Used for storing address/data |
| $2 | s2 | 10 | General purpose register. Used for storing address/data |
| $3 | s3 | 11 | General purpose register. Used for storing address/data |

**7) Addressing Modes :**

Each format can have at least one addressing mode. For R type format, there is only one addressing mode, that is register addressing. For I type format, there are 2 addressing modes – immediate addressing and base addressing.

**Translating Some HLL codes using our Designed 10 Bit ISA:**

**Arithmetic Operations:**

[Assume b is in $s1, d is in $s2, k is in $s0]

1. k = b + d

**add $s0 , $s1, $s2**

1. k = b – d

**sub $s0 , $s1, $s2**

***Logical Operations:***

[Assume a is in $s1, b is in $s2, k is in $s0]

1. k = a AND b

**AND $s0, $s1 ,$s2** //$s0 gets $s1 AND $s2

1. k = a OR b

**OR $s0, $s1, $s2** //$s0 gets $s1 AND $s2

***Data Transfer Operations:***

[Assume i is in $s1, A is in $s2 , g is in $s0]

1. g = A[i]

**sub $s0,$zero, $zero** //initialize $s0=0 to move data into $s0

**add $t0, $s1, $s2**  // here $s1 = i, so it becomes 2i

**add $t1, $t0, $t0** // 2i +2i= 4i

**add $t1, $t1, $s2**

**lw $s0, 0($t1)** // here we stored the value of A[i] in $s0

***Conditional Operations:***

[Assume i is in $s1, j is in $s2]

1. if ( i == j ){ **beq $s1, $s2, L**

i = i+1; **addi $s1, $s1, -1**

} else { **jmp exit**

i = i-1; L: **addi $s1, $s1, 1**

} **jmp exit**

**exit:**

assume i is in $s1, j is in $s2

1. if ( i <j) {

i = i+1;

} else {

i = i-1**;**

}

**slt $t0 , $s1 , $s2**

**beq $t0 , $zero , L**

**addi $s1, $s1, 1**

**jmp exit**

**L: Addi $s1, $s1, -1**

***Loop Type Operations:***

[Assume i = *$s0,* x = $s1]

1. for( int i = 0; i < 5; i++){

x=x\*2;

}

1. **add $ s0 , $zero , $zero**
2. **L : slti $t0 , $s0 , 5**
3. **beq $t0 , $zero , Exit**
4. **add $s1, $s1 , $ s1**
5. **Addi $s0 , $s0 , 1**
6. **jmp L**
7. **Exit**

**Limitations we have:**

1. In this is 10-bit CPU, only 12 operations were used and in maximum 14 operations can be used.
2. For bit limitation, the ‘Function’ and ‘Shift amount’ of R-type instruction were not used in the CPU. Instead, the instructions were passed to the Opcode portion of the ISA.
3. This CPU doesn’t have any error handling functionalities; thus the user must valid instructions in correct order that is supported by the CPU.
4. We have limited bits in R type format. Therefore we cannot perform logical operations like sll, srl. Since we cannot perform sll and srl we used this two operation by doing some manipulation with add and sub .
5. We cannot perform immediate operation whose size is larger than 2^2 = 4 bits.