

**North South University**

Department of Electrical & Computer Engineering

**CSE332**

**Computer Organization and Architecture**

**Assembler Design**

|  |  |
| --- | --- |
| **Name** | **ID** |
| **Md. Jumar Alam** | **1610512042** |
| **A. B. M. Tanjeeb Alam** | **1510338642** |
| **Md. Jahid Khandaker** | **1611131042** |
| **Muhtasim Riffat** | **1511043042** |

**Submitted to**

**Dr. Tanzilur Rahman(Tnr)**

**Submission Date: 05/08/2018**

***Introduction:*** Our task was to design an assembler which will convert the assembly code to machine language.

***Objective:*** Our main goal was to generate a machine code from a file containing assembly language. The assembler reads a program written in an assembly language, then translate it into binary code and generates output file containing machine code.

***How to use:*** In the input file the user has to give some instructions to convert into machine codes. The system will convert valid MIPS instructions into machine language and generate those codes into output file.

***Input File:*** The input file is located in a folder named “inputs”. User will write down the MIPS code in this file.

**List of Tables**

***Register List***

We have selected registers from $s1-$s15 for general purpose and $sp for special storing purpose. We assigned 4 bits for each of the register as we know in the instruction field in our ISA containing the register rs and rd contains 4 bits each.

|  |  |  |
| --- | --- | --- |
| **Conventional Name** | **Register Number** | **Binary Value** |
| $sp | 0 | 0000 |
| $s1 | 1 | 0001 |
| $s2 | 2 | 0010 |
| $s3 | 3 | 0011 |
| $s4 | 4 | 0100 |
| $s5 | 5 | 0101 |
| $s6 | 6 | 0110 |
| $s7 | 7 | 0111 |
| $s8 | 8 | 1000 |
| $s9 | 9 | 1001 |
| $s10 | 10 | 1010 |
| $s11 | 11 | 1011 |
| $s12 | 12 | 1100 |
| $s13 | 13 | 1101 |
| $s14 | 14 | 1110 |
| $s15 | 15 | 1111 |

***Op-Code List:*** We have selected following op codes and assigned op-code binary values (4 bits) for each of the op codes.

|  |  |  |
| --- | --- | --- |
| ***Op-Code name*** | ***Type*** | ***Op-Code binary*** |
| add | R-type | 0000 |
| addi | I-type | 0001 |
| sub | R-type | 0010 |
| sll | I-type | 0011 |
| and | R-type | 0100 |
| nand | R-type | 0101 |
| lw | I-type | 0110 |
| sw | I-type | 0111 |
| dsp | I-type | 1000 |
| inp | I-type | 1001 |
| otp | I-type | 1010 |
| beq | I-type | 1011 |
| slt | R-type | 1100 |
| jmp | I-type | 1101 |

***Instruction Description***

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

* Operation: $d = $s + $d
* Syntax: add $d $s

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

* Operation: $d = $s - $d
* Syntax: sub $d $s

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

* Operation: $d = $s + offset
* Syntax: addi $d offset

sll: It shifts bits to the left and fill the empty bits with zeros. The shift amount is depended on the offset value.

* Operation: $d= $sp << offset
* Syntax: sll $s offset

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

* Operation: $d = $s && $d
* Syntax: and $d $s

nand: It NAND’s two register values and stores the result in destination register. Sometimes we use nand to get NOT of register value.

* Operation: $d=$s nand $d
* Syntax: nor $d $s

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

* Operation: $d = MEM[$sp + offset]
* Syntax: lw $d offset

sw: It stores specific value from register to memory.

* Operation: MEM[$sp + offset] = $s
* Syntax: sw $s, offset

**dsp:** It displays the value stored in the register $out

**inp:** It takes the input to a register

* Operation: $s = [$in = data]
* Syntax: inp $s

**otp:** It passes the data of a register to output

* Operation: $out = $s
* Syntax: otp $s

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

* Operation: if ($s==$sp) jump to offset

else goto next line

* Syntax: beq $s offset

slt: If $s is less than $d, $sp is set to one. It gets zero otherwise.

* Operation: if $s < $d $sp = 1

else $sp = 0

* Syntax: slt $d $s

jmp: Jumps to the calculated address.

* Operation: PC = nPC
* Syntax: jmp target

***Limitation:***

The user has to give spaces between instruction words and nothing else like “,” or “-” in between them in the “inputs” file. If user don’t follow this format the system will show a valid code as invalid.

# *Manual:*

To run the program, one needs to open the application file called “12BITAssembler.cpb” which is provided in the folder. If one wants to see the code then open the project file from the provided folder*, it is absolutely necessary that the folder which is containing the program, has a text file* ***(.txt)*** *called “****inputs***”. This is the file from where the assembler reads the assembly codes. The program reads the code from **“input.txt”** file and writes the corresponding binary code in a text file called **“outputs.txt”.** We have already provided an input file with corresponding output file in the project folder. If one wants to try his/her own assembly code then, he/she needs to write the codes to the application through the input file. One important thing to notice is that, each line of the input file can only contain one instruction and words must be separated by spaces.