

A 14-Bit Instruction Set Architecture Design

**Submitted by - Group 2**

Shahjalal Ekbal – 1712397642

Shad Abdullah – 1912215642

Khalid Bin Shafiq – 1911342642

Submitted to:

Dr. Tanzilur Rahman

CSE332.2

Summer 2021

**Introduction**

For our project, we have to design a CPU for a Vendor company. The CPU will have the following features.

* It will be a 14-bit single-cycle CPU.
* It will have separate Data and Instruction Memory.
* Instructions in ISA will solve logical, arithmetic, branching, and loop operations.
* ISA will be general-purpose enough to be able to run provided general programs.

**Design**

We have to use a 14-bit instruction set. So, we are using a 5-bit opcode design that will allow up to maximum instructions. Since, we are low in number of operations, it might give us the upper hand of easier implementation but also a chance of losing important functionalities. In order to maintain balance between easier development and losing important functionalities, we have decided to pick the most suitable and general-purpose operations which will allow us to write any general program.

**Operand**

For our project, we’ve used 0-3(depends on ISA format) operands in our ISA.

**Operand Type**

Register and Memory based operands.

**Operations**

In our design, we are going to use 32 operations. According to our ISA, it would be efficient to use 32 operations as they are easy to develop and they will be selected in a manner in which all the general-purpose program can be implemented using them.

**Operation Type**

There are 5 different types of operations.

1. Arithmetic

We have 5 arithmetic operations.

|  |  |  |
| --- | --- | --- |
| ADD | SUB | ADDI |

1. Logical

We have 7 different types of Logical operations.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| AND | OR | NAND | NOR | XOR | SLL | SRL |

Here, NAND and NOR are universal Gate. Universal gate can implement any logical operation. The other operations are provided as syntactic sugar.

1. Conditional

We have 11 conditional operations.

|  |  |  |  |
| --- | --- | --- | --- |
| SLTI | SLT | BNE | BEQ |

All the operations are capable of jumping or branching to the address provided either by an address or an indirect addressing. The exception here is SLT and SLTI which only sets or resets a register instead of performing a branching.

1. Data Transfer

For data transferring, we have 6 different operations

|  |  |  |  |
| --- | --- | --- | --- |
| LW | SW | DIN | DOUT |

Our CPU will be capable of reading or writing to memory, it will have enough space to solve any general program. The CPU will also have user interactions through the DIN and DOUT operations. LW and SW are used in register-memory data transfer. DIN and DOUT are used for user-register data transfer. MOV and MOVI will be useful in copying data between registers or to load data from immediate value.

1. Unconditional

We have 3 unconditional jumps in our ISA.

|  |
| --- |
| JUMP |

Unconditional jumps will be useful to implement any loop kind of structures in our programs. NOP will be useful to create delays in the programs, while JUMPR will be useful in case we have our code memory address stored in a register.

**Formats**

our instructions are subdivided in three formats.

* Register (R type)
* Immediate (I type)
* Jump (J type)

Our design will be able to perform any general kind of logical, arithmetic, if-else branching, looping, and external reading/writing operations.

**Register (R type)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Destination Register** | **1st Source Register** | **2nd Source Register** |
| op | rd | rs | rt |
| 5 bits | 3 bits | 3 bits | 3 bits |

**Immediate (I type)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **DestinationRegister** | **Source register** | **Immediate** |
| op | rd | rs | imm |
| 5 bits | 3 bits | 3 bits | 3 bits |

**Jump (J type)**

|  |  |
| --- | --- |
| **Opcode** | **Address** |
| op | ad |
| 5 bits | 9 bits |

All the instruction are 14 bits except DIN, DOUT, NOP. Their remaining bits are 0.

**Register Table**

We have selected registers from $zero, $sp (special register), $t0-$t4 and $s0-$s8 and assigned 3bits for each of the registers. The registers, rd contains 3 bits.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of the Registers** | **Register Number** | **Value Assigned** | **Register Purpose** |
| $zero | 0 | 000 | Hardwired to 0 |
| $sp | 1 | 001 | Special Purpose |
| $s0 | 2 | 010 | General Purpose |
| $s1 | 3 | 011 | General Purpose |
| $s2 | 4 | 100 | General Purpose |
| $s3 | 5 | 101 | General Purpose / Memory Address |
| $t0 | 6 | 110 | Temporary |
| $t1 | 7 | 111 | Temporary |

Register Use Conventions

1. $zero should always be untouched, and its value will always be 0.
2. $sp, this register should be used to compare its value with another operand during branching.
3. When loading or storing data from or to the data memory, **it is always assumed that the base address is stored in the $s3 register**.

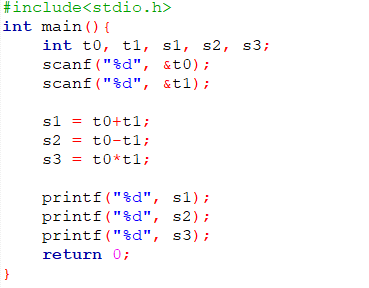
**Instructions Table**

Table for the instructions, instruction types, their opcode and addressing modes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Category** | **Instruction** | **For**  **mat** | **Syntax** | **Meaning** | **Op**  **code** | **Addressing Mode** |
| Logical | AND | R | AND rd, rs, rt | rd = rs&rt | 00000 | Register |
| Logical | OR | R | OR rd, rs, rt | rd = rs | rt | 00001 | Register |
| Arithmetic | ADD | R | ADD rd, rs, rt | rd = rs + rt | 00111 | Register |
| Arithmetic | SUB | R | SUB rd, rs, rt | rd = rs - rt | 01000 | Register |
| Logical | NAND | R | NAND rd, rs, rt | Rd=rs NAND rt | 00010 | Register |
| Logical | NOR | R | NOR rd, rs, rt | rd = rs NOR rt | 00011 | Register |
| logical | XOR | R | XOR rd, rs, rt | rd=rs XOR rt | 00100 |  |
| Arithmetic | ADDi | I | ADDi rs, 2 | rs = rs + 2 | 01001 | Immediate |
| Data Transfer | LW | I | LW rs, imm | rs=Mem[$s3+imm] | 01010 | Indirect |
| Data Transfer | SW | I | SW rs, imm | Mem[$s3+imm] = rs | 01011 | Indirect |
| Logical | Sll | I | Sll rs, imm | rs=rs<<2 | 00101 | Immediate |
| Logical | Srl | I | Srl rs, imm | rs=rs>>2 | 00110 | Immediate |
| Conditional | Beq | I | Beq rs,imm | If (rs==$sp)  Go to line imm | 01100 | Direct |
| Conditional | Bne | I | Bne rs,8 | If(rs!=$sp)  Go to line imm | 01101 | Direct |
| Conditional | Slt | R | Slt rd,rs,rt | If(rs<rt) rd=1,else  rd=0 | 01110 | Register |
| Conditional | Slti | I | Slti rs,imm | If(rs<imm)  rs=1,else  rs=0 | 01111 | Immediate |
| Unconditional | Jump | J | J address | Jump to line imm | 10000 | Direct |
| Data Transfer | Din | R | Din rs | User input is stored in rs | 10001 | Register |
| Data Transfer | Dout | R | Dout rs | From rs,content is displayed | 10010 | Register |

**Example Code**

1. Output the sum of an array that is stored in memory 3 to 6. Perform addition, subtraction and multiplication operation between two integers.



L1: *DIN $t0* ; Store user input in t0

L2: *DIN $t1* ; Store user input in t1  
L3: *ADD $s1, $t0, $t1* ; Add t0 and t1 and store in s1  
L4: *SUB $s2, $t0, $t1* ; Subtract t1 from t0 and store in s2

L5: *MUL $s3, $t0, $t1* ;Multiply t0 by t1 and store in s3

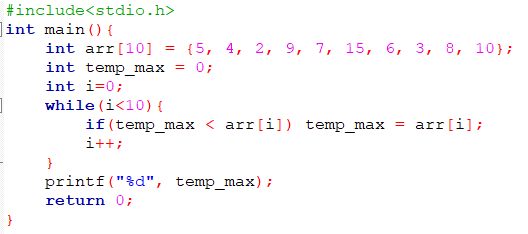
L6: *DOUT $s1* ;Display content of s1

L7: *DOUT $s2*  ; Display content of s2

L8: *DOUT $s3* ; Display content of s3

Execution time: 8 clock cycles

1. Find the maximum of an array that is stored from memory 3 to 12 using Loop Structure:



L0: *MOVI $s3, 3* ; $s3 is default address register

L1: *LW $s1, 0* ; **Loop Starts here**, load current array value  
L2: *SLT $s2, $t0, $s1* ; if (t0 < s1) s2 = 1, else s2 = 0

L3: *AND $sp, $sp, $zero* ; t0 is still maximum  
L3: *BEQ $s2, L6* ; t0 is still maximum

L4: *AND $t0, $zero, $t0* ; L5: *ADD $t0, $s1, $t0* ; t0 swapped with $s1  
  
L6: *ADDi $s0, 1* ; $s0 is loop counterL7: *MOVI $sp, 10* ; Loop needs to iterate 10 times

 L8: *BNE, $s0, L1* ; **Loop Ends** here

L9: *DOUT $t0* ; Display Maximum

Execution time: 82 clock cycles