**ECE 366 - Fall 2018**

**Project 3 - ISA Simulator**

**Group 10**

Amaan Baiyat

Alberto Espinoza

Justino Almazan

**ISA Introduction**

**Overview**: This ISA is called AAJ, the initials of the our team members. We decided to use the best of the three ISAs available to our team. We chose to use Alberto Espinoza’s ISA from project 2 since we were sure of his ISA’s ability to properly compute the p1 and p2 for data memory of A and B. When designing the python simulator for computation of instruction memory we had a lot of fun debugging. We included a file named “p2\_value\_verifier.py” that allowed us to compare all 100 lines with the target from p2 solely using python without implementing our ISA. This gave us an idea on what results for S and C we were going to need. This project was challenging but solvable, we had to think outside the box multiple times.

**Philosophy:** This ISA is strictly implemented to perform two tasks which are modular exponentiation and best matching count. Since the AAJ ISA works with less instructions then this will cause the ISA’s delay to be far less than what the MIPS CPU is. In case a user believed MIPS was irritating they will dislike our ISA even more.

**Instruction List**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction/**  **Syntax** | **Encoding/**  **Format** | **Functionality** | **Range** |
| Add $Rd, Rs | X 000 dd ss | $Rd = $Rd + $Rs  PC = PC + 1 | Rd, Rs = {R0, R1, R2, R3} |
| Addi $Rd, imm | X 001 dd ii | $Rd = $Rd + imm  PC = PC + 1 | Rd = {R0, R1, R2, R3}  imm = [-2, 1] |
| SLT $Rd, $Rs | X 010 dd ss | If $Rd <$Rs then branch = 1,  Else branch = 0  PC = PC + 1 | Rd, Rs = {R0, R1, R2, R3} |
| B imm | X 100 iiii | If branch = 1 then PC = PC + imm,  Else PC = PC + 1 | Imm = [-8, 7] |
| J imm | X 011 iiii | PC = PC + imm | Imm = [-8, 7] |
| LOAD $Rd, $Rs | X 1010 dd s | $Rd = MEM[$Rs]  PC = PC + 1 | Rd = {R0, R1, R2, R3}  Rs = {R0, R1} |
| STR $Rd, $Rs | X 1011 dd s | MEM[$rs] = $Rd  PC = PC + 1 | Rd = {R0, R1, R2, R3}  Rs = {R0, R1} |
| LSL $Rd | X 11000 dd | $Rd = $Rd << 1  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| NXOR $Rd, $Rs | X 11001 d a | $Rd = $Rd bitwise not xor $Rs  PC = PC + 1 | Rd, Rs = {R0, R1} |
| EQZ $Rd | X 11010 dd | If $Rd == 0 then branch = 1  Else branch = 0  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| COMP $Rd | X 11011 dd | $Rd = -$Rd = 2’s complement ($Rd)  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| RCVR $Rd | X 11100 dd | $Rd = $SRd for d = {0, 1, 2, 3}  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| RST $Rd | X 11101 dd | $Rd = 0  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| STSH $Rd | X 11110 dd | $SRd = $Rd for d = {0, 1, 2, 3}  PC = PC + 1 | Rd = {R0, R1, R2, R3} |
| END | 11111111 | PC = PC |  |

**Examples of our ISA instructions in use:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Opcode** | **Example** | **Result** |
| ADD | 000 | ADD $R2, $R0 | $R2 = $R2 + $R0  PC += 1 |
| ADDI | 001 | ADDI $R0, 0 | $R0 = $R0 + 0  PC +=1 |
| SLT | 011 | SLT $R1, $R0 | If $R1 < $R0 then branch = 1  Else branch = 0  PC += 1 |
| B | 010 | B 7 | If branch = 1 then PC = PC + 7  Else PC = PC + 1 |
| J | 100 | J 7 | PC = PC + 7 |
| LOAD | 1010 | LOAD $R1, $R0 | $R1 = MEM[$R0]  PC = PC + 1 |
| STR | 1011 | STR $R3, $R0 | MEM[$R0] = $R3  PC = PC + 1 |
| LSL | 11000 | LSL $R1 | $R1 = $R1 \* 2  PC = PC + 1 |
| NXOR | 11001 | NXOR $R2, $R0 | $R2 = $R2 bitwise (xor)’ $R0  PC = PC + 1 |
| EQZ | 11010 | EQZ $R2 | If $R2 equals 0 then branch equals 1 else branch equals 0  PC = PC + 1 |
| COMP | 11011 | COMP $R2 | $R2 = 2’s complement of $R2  PC = PC + 1 |
| RCVR | 11100 | RCVR $R0 | $R0 ← $SR0 |
| RST | 11101 | RST $R0 | $R0 = 0  PC = PC + 1 |
| STSH | 11110 | STSH $R0 | $SR0 ← $R0  PC = PC + 1 |
| END | 11111111 | END | PC gets moved to end beyond last line of code to ensure no more machine code is read from current program |

**Registers:**

This ISA uses 10 registers, four are declared as R0 R1 R2 R3. Meanwhile four others are used as temporary duplicate registers of R0 - R3. One register is used for the program counter and another is used to keep track of branch conditionals.

General Purpose Registers

These are R0 - R3 registers, they are used for executing basic instructions similar to the ones found in MIPS.

Temporary Save Registers

These are duplicates of R0 - R3 but we refer to them as SR0 SR1 SR2 SR3. Basically they can only be used with commands such as “STSH” and “RCVR”. What they do is save/load the value from their general purpose clone, so R0 can only save/load to SR0. This will allow us to ‘free’ R0 if its value is currently not in use but will be needed later, we can then use R0 for more instructions without losing its previous value.

Program Counter Register

This allows us to see which line of machine code in our imem files we are currently at. It changes after each instruction with either a pc+=1 or pc is determined by a branch/jump immediate value.

Branch Condition Register

This register depends on our instructions SLT and EQZ, depending on these instructions the branch register will either equal zero or one. If branch register equals one then when we come across a B instruction we will change pc register by the immediate following the B in the branch instruction. If branch register equals zero then we will update pc to increase by only one.

**This table indicates our register index:**

|  |  |
| --- | --- |
| Register Type | Supported Registers |
| General Purpose Registers | R0 R1 R2 R3 |
| Stash/Receiver Registers | SR0 SR1 SR2 SR3 |
| Program Counter Register | PC |
| Branch Condition Register | Branch |

**Control Flow:**

B (branch), J (jump), SLT (set less than), EQZ (equal to zero). Instruction SLT is use to compare two registers (rd and rs), rd has to be less rs in order for branch to equal one indicating condition is true, if not true than we will update pc by +1 when branch is encountered. EQZ is used to compare a register with zero, if it is true then branch is set to one causing a pc update by immediate when branch instruction is encountered. The branch instruction B can move backward 8 lines as well as moving forward 7 lines. The jump instruction J will update pc without necessity of a conditional and it also moves forward 7 and backwards 8, J is meant mostly for executing a loop in the machine code or imem.

**These are examples of our Control Flow instructions:**

|  |  |
| --- | --- |
| ISA instruction | Machine Code |
| B 4 | 01000100 |
| J -2 | 10111110 |
| SLT $R0, $R1 | 0010001 |
| EQZ $R2 | 01101010 |

**Data Memory Addressing:**

LOAD and STR instructions will access memory for reading and writing to a location. Taking an inspiration from MIPS we compute a general purpose register to equal a value, from this point we use that value to access memory locations for the LOAD and STR instructions. This is an inconvenience for our ISA users and for our team when it comes to debugging p1 and p2.

**This table is an example of our store and load instructions:**

|  |  |
| --- | --- |
| ISA Instructions | Machine Code |
| LOAD $R3, $R1 | 11010111 |
| STR $R3, $R0 | 11011110 |

**Q & A**

**1) What are the most significant advantages of your ISA (with regard to the two programs, hardware implementation, ease of programming, etc)? What are the main limitations? What are the main compromises that you have done to make things work, rather than perfecting everything?**

**2) What have you done towards the goals of low DIC and HW simplification? What could have been done differently to better optimize for each of the two goals, if to start over?**

**3) Reflect on this project(1-3) experience:**

**a) What did you learn from this project? What was the best / worst thing about it?**

**b) What advice would you give to someone taking this project in a future semester?**

**c) How would you describe the value of this project experience in a job interview?**

**Simulation Results**

**1) Execution Results for data\_mem[0] - data\_mem[5]**

**a)**

**b)**

**c)**

**d)**

**2) Execution process of target programs**

**ISA Package**

**1) Algorithms in Assembly Code:**

**P0:**

**P1:**

**P2:**

**2) Instruction Memory Machine Code:**

**P0:**

**P1:**

**P2:**

**3) Data Memory**

**Case C:**

**Case D:**

**4) Python Simulator Code:**

**5) Hardware Schematics:**

**a) ALU Schematic**

**b) CPU Datapath Design**

**c) Control Logic Truth Table**