Principals of Operation

Brett Bloethner | CSCI E-93

# **Overview**

This system implements a basic 16-bit word size and an overall design similar to the MIPS architecture. At its core is a load/store design with 16 general purpose registers. Memory access is restricted to only load and store instructions and the novel feature included is a hardware multiplier.

# **Instruction Types**

### R-Type

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| opcode | | | | rs | | | | rt | | | | rd | | | |

R type instructions are triple register instructions used primarily for arithmetic operations. Rs is the register source while rt is the register target and rd is the register destination. Arithmetic operations are performed on the values of rs and rt while the result is stored as the value of register rd.

R-type instructions include: **ADD, SUB, AND, OR, XOR, NOR, SLT, SGT, SET**

### F-Type

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| opcode | | | | rs | | | | rd | | | | func | | | |

F-type instructions are double register instructions. Unlike the R-type instructions, F-type instructions don’t include a register target operand. Instead, rs is mutated using a constant value implied by the instruction (e.g. shift left logical shifts 1 and has no shift amount). This is done to free up the 4 lowest bits in the instruction word, allowing us to add a function operand and ultimately integrate more instructions.

F-type instructions include: **SLL, SLA, SRA, SWVA, LWVA**

### I-Type

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| opcode | | | | rd | | | | immediate | | | | | | | |

The I-type instruction includes an opcode as well as a registry destination and a 1-byte immediate value. This instruction type is used primarily by the or immediate and the load upper immediate instructions. By using these two instructions together the computer is able to load a 16bit word into a register which can then be included in other pseudo instructions. Branch on equal to zero also uses this instruction type to branch to an immediate address offset constant if rd is zero. All branch pseudo operations are then built upon the branch on zero instruction.

I-type instructions include: **ORI, LUI, BEZ**

### J-Type

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| opcode | | | | Address Immediate | | | | | | | | | | | |

The J-type instruction includes an opcode as well as a direct address constant of 12-bits. It is used for the jump instruction.

J-type instructions include: **J**

# **Instructions**

There are 19 instructions along with 11 pseudo instructions. One of my challenges was that I wanted to see if I could get all the instructions to fit within the 16 combinations allowed by the 4-bit opcode. I hope to streamline my instructions to all fit within into the 4-bit opcode and omit these function operands as I discover more about the project and identify certain instructions that may not really be needed.

## Arithmetic Instructions

**Add | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x00** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **ADD $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Add value in one reg to another, store product in a third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs + $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **no side effects** | | | | | | | | | | |

**Add Immediate | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **ADDI $rs, $rd, immediate** |
| **Purpose** | **Add const to value in a reg, store product in another reg.**  **lui $at, upper(immediate)**  **ori $at, lower(immediate)**  **add $rd, $rs, $at** |
| **Side Effects** | **Register $at is used temporarily to construct the immediate 16-bit value.** |

**Subtract | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x01** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **SUB $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Subtract const from value in a reg, store result in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs - $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **none** | | | | | | | | | | |

**Subtract Immediate | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **SUBI $rs, $rd, immediate** |
| **Purpose** | **Subtract const from value in a reg,store product in another reg.**  **lui $at, upper(immediate)**  **ori $at, lower(immediate)**  **sub $rd, $rs, $at** |
| **Side Effects** | **Register $at is used temporarily to construct the immediate 16-bit value.** |

**And | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x02** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **AND $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct bitwise AND between to reg values, store results in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs & $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**And Immediate | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **ANDI $rs, $rd, immediate** |
| **Purpose** | **Conduct bitwise AND between one reg value and a const, store result in another reg.**  **lui $at, upper(immediate)**  **ori $at, lower(immediate)**  **and $rd, $rs, $at** |
| **Side Effects** | **Register $at is used temporarily to construct the immediate 16-bit value.** |

**OR | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x03** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **OR $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct bitwise OR between two reg values, store results in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs | $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**XOR | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x04** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **XOR $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct bitwise XOR between two reg values, store results in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs ^ $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**NOR | R-TYPE | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x05** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **NOR $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct a bitwise NOR between two reg values, store results in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs NOR $rt** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

# Comparison Based Instructions

**Set Less Than | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x06** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **SLT $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct less than comparison between two reg values, store TF result in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **If $rs < $rt $rd = 1; else $rd = 0;** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**Set Greater Than | R-Type | Pseudo Instruction**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x07** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **SGT $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct greater than comparison between two reg values, store TF result in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **If $rs > $rt $rd = 1; else $rd = 0;** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**Set Equal To | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x08** | | | | **rs** | | | | **rt** | | | | **rd** | | | |
| **Format** | | | | | **SET $rd, $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct equal to comparison between two reg values, store TF result in third reg.** | | | | | | | | | | |
| **Operation** | | | | | **If $rs === $rt $rd = 1; else $rd = 0;** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**Set Less Than or Equal To | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **SLTE $rs, $rt, $rd** |
| **Purpose** | **Conduct less than or equal to comparison between two reg values, store TF result in third reg.**  **slt $c, $rs, $rt**  **set $rd, $rs, ts**  **or $rd, $rd, $c** |
| **Side Effects** | **Register $c is used temporarily to hold the result of less than operation.** |

**Set Greater Than or Equal To | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **SGTE $rs, $rt, $rd** |
| **Purpose** | **Conduct greater than or equal to comparison between two reg values, store TF result in third reg.**  **sgt $c, $rs, $rt**  **set $rd, $rs, ts**  **or $rd, $rd, $c** |
| **Side Effects** | **Register $c is used temporarily to hold the result of greater than operation.** |

# Logical

**Shift Left Logical | F-Type | Register Addressing with Function**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x09** | | | | **rs** | | | | **rd** | | | | **0000** | | | |
| **Format** | | | | | **SLL $rd, $rs** | | | | | | | | | | |
| **Purpose** | | | | | **Shift value of a reg left by 1 and store the result in a different reg. Zeros are shifted in.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs << 1** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects** | | | | | | | | | | |

**Shift Right Arithmetic | F-Type | Register Addressing with Function**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x09** | | | | **rs** | | | | **rd** | | | | **0001** | | | |
| **Format** | | | | | **SRA $rd, $rs** | | | | | | | | | | |
| **Purpose** | | | | | **Shift value of a reg right by 1 and store the result in a different reg. Sign bit is shifted in.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs >> 1** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

**Shift Right Logical | F-Type | Register Addressing with Function**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x09** | | | | **rs** | | | | **rd** | | | | **0010** | | | |
| **Format** | | | | | **SRL $rd, $rs** | | | | | | | | | | |
| **Purpose** | | | | | **Shift value of a reg right by 1 and store the result in a different reg. Zeros are shifted in.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rs >> 1** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

**OR Immediate | I-Type | Immediate Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0a** | | | | **rd** | | | | **immediate** | | | | | | | |
| **Format** | | | | | **ORI $rd, immediate** | | | | | | | | | | |
| **Purpose** | | | | | **Conduct a bitwise or on a reg and a constant. Store the results back in the original reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $rd | immediate** | | | | | | | | | | |
| **Side Effects** | | | | | **The old value in $rd is wiped when the result of the OR operation takes it place.** | | | | | | | | | | |

# Branching and Jumping Instructions

**Branch On Zero | I-Type Branching| Immediate Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0b** | | | | **rs** | | | | **offset** | | | | | | | |
| **Format** | | | | | **bez $rs, offset** | | | | | | | | | | |
| **Purpose** | | | | | **Branch to the offset address if the condition of a register is that it equals all zeros.** | | | | | | | | | | |
| **Operation** | | | | | **if $rs === 0 PC = PC + 2 + (offset << 1); else PC = PC + 2** | | | | | | | | | | |
| **Side Effects** | | | | | **Branch on zero isn’t very useful until combined with other comparison instructions to create branching pseudo instructions. See below pseudo instructions…** | | | | | | | | | | |

**Branch on Greater Than | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **BGT $rs, $rt, offset** |
| **Purpose** | **Branch to offset address if one reg value is greater than another.**  **slte $c, $rs, $rt**  **bez $c, offset** |
| **Side Effects** | **Register $c is used temporarily to hold the result of slte operation.** |

**Branch Less Than | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **BGT $rs, $rt, offset** |
| **Purpose** | **Branch to offset address if one reg value is less than another.**  **sgte $c, $rs, $rt**  **bez $c, offset** |
| **Side Effects** | **Register $c is used temporarily to hold the result of sgte operation.** |

**Branch on Equal To | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **BET $rs, $rt, offset** |
| **Purpose** | **Branch to offset address if one reg value is equal to another.**  **xor $c, $rs, $rt**  **bez $c, offset** |
| **Side Effects** | **Register $c is used temporarily to hold the result of xor operation. Xor is a logical operation but is using $c in this case because its being used for a comparison operation.** |

**Branch Greater Than or Equal To | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **BGTE $rs, $rt, offset** |
| **Purpose** | **Branch to offset address if one reg value is greater than or equal to another.**  **slt $c, $rs, $rt**  **bez $c, offset** |
| **Side Effects** | **Register $c is used temporarily to hold the result of sgte operation.** |

**Jump| I-Type Jumping | Immediate Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0c** | | | | **target** | | | | | | | | | | | |
| **Format** | | | | | **J target** | | | | | | | | | | |
| **Purpose** | | | | | **Jump to address target after sign extension and left shift of 1** | | | | | | | | | | |
| **Operation** | | | | | **PC = target << 1** | | | | | | | | | | |
| **Side Effects** | | | | | **PC counter will jump to a new address and execute that instruction.** | | | | | | | | | | |

# Load / Store Instructions

**Store Word Variable Address | F-Type | Register Addressing with Function**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0d** | | | | **rs** | | | | **address** | | | | **0001** | | | |
| **Format** | | | | | **SWVA $rs, offset($address)** | | | | | | | | | | |
| **Purpose** | | | | | **Store a word from a reg into memory at the address given in another reg** | | | | | | | | | | |
| **Operation** | | | | | **memory[$address + offset] = $rs;** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

**Store Word | Pseudo Instruction**

|  |  |
| --- | --- |
| **Format** | **SW $rs, address** |
| **Purpose** | **Store a word from a reg into memory at the address given in a constant.**  **lui $at, upper(address)**  **ori $at, lower(address)**  **swva $at, $rs** |
| **Side Effects** | **Assembler will make use of $at for constructing the 16-bit word via load upper immediate and or immediate from the address constant provided.** |

**Load Word Variable Address | F-Type | Register Addressing with Function**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0d** | | | | **rd** | | | | **address** | | | | **0010** | | | |
| **Format** | | | | | **LWVA $rd, offset($address)** | | | | | | | | | | |
| **Purpose** | | | | | **Load a word into a reg from the memory unit using an address stored in a reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = memory[$address + offset];** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

**Load Word | Pseudo**

|  |  |
| --- | --- |
| **Format** | **LW $rd, address** |
| **Purpose** | **Load a word into a reg from the memory unit using an address given in a constant.**  **lui $at, upper(address)**  **ori $at, lower(address)**  **lwva $rd, $at** |
| **Side Effects** | **Assembler will make use of $at for constructing the 16-bit word via load upper immediate and or immediate from the word in memory at the specified address.** |

**LUI**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0e** | | | | **rd** | | | | **immediate** | | | | | | | |
| **Format** | | | | | **LUI $rd, imm** | | | | | | | | | | |
| **Purpose** | | | | | **Load a constant into the upper 8 bits of a reg, making the lower 8 bits all zeros.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = (imm << 8);** | | | | | | | | | | |
| **Side Effects** | | | | | **The destination registers lower 8 bits will be zeros. This is usually followed up by a ORI to fill the reg with a full word of data without any instructions.** | | | | | | | | | | |

# Novel Feature Instructions

**MULT | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0f** | | | | **rs** | | | | **rt** | | | | **0000** | | | |
| **Format** | | | | | **MULT $rs, $rt** | | | | | | | | | | |
| **Purpose** | | | | | **Multiply $rs and $rt and store the two resulting values in HI and LO in the multiplier hardware unit** | | | | | | | | | | |
| **Operation** | | | | |  | | | | | | | | | | |
| **Side Effects** | | | | | **SPR $HI and $LO will now contain the result of the multiplication operation and the data must be dealt with accordingly.** | | | | | | | | | | |

**MFHI | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0f** | | | | **rd** | | | | **SBZ** | | | | **0001** | | | |
| **Format** | | | | | **MFHI $rd** | | | | | | | | | | |
| **Purpose** | | | | | **Move contents of the $HI register in the multiplication hardware to a general purpose reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $HI** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

**MFLO | R-Type | Register Addressing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **0x0f** | | | | **rd** | | | | **SBZ** | | | | **0010** | | | |
| **Format** | | | | | **MFLO $rd** | | | | | | | | | | |
| **Purpose** | | | | | **Move contents of the $LO register in the multiplication hardware to a general purpose reg.** | | | | | | | | | | |
| **Operation** | | | | | **$rd = $LO** | | | | | | | | | | |
| **Side Effects** | | | | | **No side effects.** | | | | | | | | | | |

# **Registers**

The register strategy is similar to MIPS in this design except there are 16 16-bit general purpose registers. The general-purpose registers are outlined in the table below. Some GPR registers are reserved for specific uses while others can be used however the programmer likes. I anticipate having to change these register assignments as the project progresses. The design also includes a program counter register, instruction memory register, and memory data register similar to MIPS. There are $HI and $LO special purpose registers listed for the hardware multiplier.

|  |  |  |
| --- | --- | --- |
| Register # | Name | Usage |
| $0 | $zero | Hard wired to value 0 |
| $1 | none | Temporary register for anything |
| $2 | $at | Reserved for assembler |
| $3 | $c | Reserved for results of comparison operation involved in pseudo instructions |
| $4 -$5 | $v1-$v2 | Fn return values |
| $5 - $7 | $a1-$a3 | Fn arguments |
| $8 -$10 | $t1-$t3 | Temporary not preserved |
| $11-$13 | $s1-$s3 | Temporary preserved by subprogram |
| $14 | $sp | Stack pointer |
| $15 | $ra | Return address |
| N/A | $HI | Stores upper half of the results of mult in the hardware multiplier (not a GPR and not in the register file) |
| N/A | $LO | Stores lower half of the results of mult in the hardware multiplier (not a GPR and not in the register file) |

# **Control Codes**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | WriteRegDest | WriteReg | MemToReg | WriteMem | ReadMem | Branch | ShouldLUI | ALUsrc | mult | Jump | opcode | ALUfn | ALU Action |
| ADD | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x00 | XXXX | Add |
| SUB | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x01 | XXXX | Sub |
| AND | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x02 | XXXX | AND |
| OR | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x03 | XXXX | OR |
| XOR | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x04 | XXXX | XOR |
| NOR | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x05 | XXXX | NOR |
| SLT | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x06 | XXXX | Set on < |
| SGT | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x07 | XXXX | Set on > |
| SET | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x08 | XXXX | Set on = |
| SLL | 01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x09 | 0000 | L Shift |
| SRA | 01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x09 | 0001 | A Shift |
| SRL | 01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x09 | 0010 | L Shift |
| ORI | 00 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x0a | xxxx | OR |
| BEZ | xx | 0 | 0 | 0 | 0 | 1 | 0 | x | 0 | 0 | 0x0b | xxxx | Sub |
| J | xx | 0 | 0 | 0 | 0 | 0 | 0 | X | 0 | 1 | 0x0c | xxxx | xxxx |
| SWVA | xx | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x0d | 0001 | Pass thru |
| LWVA | 00 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x0d | 0010 | Pass thur |
| LUI | 00 | 1 | 0 | 0 | 0 | 0 | 1 | X | 0 | 0 | 0x0e | xxxx | xxxx |
| MULT | xx | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x0f | 0000 | m. hdwr |

# **Novel Feature**

The novel feature I’m planning on implementing is a hardware multiplier. Originally, I wanted to do pipelining but my expertise with computer architecture is very much so at the novice level. To add to the difficulty, I plan to try and get all of the necessary instructions addressable by a 4-bit opcode strategy, leaving me with 16 possible real instructions. I’m curious to see if this would be enough if I compliment the 16 or fewer instructions with creative pseudo instructions. Right now, I have all opcode combinations occupied and one instruction type (F-type) implements a function operand to further expand on the opcodes. I hope to cut down on the number of instructions I have so that I can remove that function operand, allowing me to add more functionality where the F-type instruction and its 4 function operand bits were previously used.

# **Assembler Language**

The assembler language is also loosely modeled after MIPS and makes a heavy use of pseudo instructions although they may appear and feel like real instructions. The assembler makes heavy use of $at while creating pseudo instructions designed to mimic “immediate” instructions and makes heavy use of the $c register for pseudo instructions involving comparison operations. A typical assembler operation would include the operation symbol followed by a tab followed by the operands with each operand comma separated like below

ADD $t3, $t1, $t2

Comments could then be added by using the number sign where everything after the number sign will be ignored until the next return/line break.

# example add operation

ADD $t3, $t1, $t2 # add t2 to t1 and set as value in t3

Data is loaded by a .data directive and where the label is separated from the value by a semicolon and a tab and all data is words so it’s not necessary to put the data type unless its an ascii string. The .main directive delineates where the program execution will being.

# example add operation

.data

var1: 3 # create a var with value of 3

var2: .ascii “whats up”

.main

ADDI $t3, $t1, var1 # add var1 to t1 and set as value in t3