**MS-32 ©**

**Senior Design Project**

**Steven Sallack**

**011398099**

**Michael Diep**

**015330547**

**CECS 440 – M/W 11:00am**

**Due Date – 4/23/2018**

**Instructor: R.W. Allison**

Table of Contents

1. Abstract---------------------------------------------------4
2. Instruction Set Architecture-------------------------------5
   1. Harvard Memory Architecture and Organization------------5
   2. Machine Register Set------------------------------------7
   3. Data Types----------------------------------------------8
   4. Addressing Modes----------------------------------------9
   5. Instruction Set-----------------------------------------12
      1. R-type Instructions -------------------------------12
      2. I-type Instructions--------------------------------24
      3. J-type Instructions--------------------------------37
      4. Enhanced Instructions------------------------------39
   6. Interrupts and Reset-------------------------------------42
3. Verilog Implementation / Design / Verification--------------43
   1. Top Level Module-----------------------------------------43
      1. MIPS\_Test\_Module.v (Interrupt using JR)-------------43
      2. MIPS\_Test\_Module.v (Interrupt using RETI)-----------47
   2. Internal Components--------------------------------------50
      1. MIPS\_Top.v (Int. using JR)--------------------------50
      2. MIPS\_Top.v (Int. using RETI)------------------------56
         1. Instruction\_Unit.v-----------------------------64
            1. Load\_Reg\_32.v----------------------------66
            2. Data\_Memory\_1Kx32.v----------------------67
         2. Integer\_Datapath.v-----------------------------68
            1. Regfile32.v------------------------------71
            2. ALU\_top.v--------------------------------72

MIPS\_32.v---------------------------74

DIV\_32.v----------------------------80

MPY\_32.v----------------------------81

* + - 1. Pipelined\_MCU.v (Int.using JR)-----------------82
      2. Pipelined\_MCU.v (Int. using RETI)--------------89
      3. BranchControl.v--------------------------------96
      4. BreakCounter.v---------------------------------98
      5. ForwardingUnit.v-------------------------------99
      6. IO\_Module.v-----------------------------------101
      7. FlagCounter.v (Int. using RETI)---------------102
      8. InterruptServicer.v (Int. using RETI)---103
      9. ReturnServicer.v (Int. using RETI)------104
  1. Memory Modules------------------------------------------105
  2. Annotated Log Files-------------------------------------144
     1. Module 1-------------------------------------------144
     2. Module 2-------------------------------------------145
     3. Module 3-------------------------------------------146
     4. Module 4-------------------------------------------147
     5. Module 5-------------------------------------------148
     6. Module 6-------------------------------------------149
     7. Module 7-------------------------------------------150
     8. Module 8-------------------------------------------152
     9. Module 9-------------------------------------------153
     10. Module 10------------------------------------------155
     11. Module 11------------------------------------------156
     12. Module 12------------------------------------------158
     13. Module 13 (Int. using JR)--------------------------159
     14. Module 14 (Int. using RETI)------------------------161

1. Hardware Implementation------------------------------------163
   1. Top Level Diagram (Int. using JR)-----------------------163
   2. Top Level Diagram (Int. using RETI)---------------------164
   3. MIPS (Int. using JR)------------------------------------165
   4. MIPS (Int. using RETI)----------------------------------166
   5. Instruction Unit----------------------------------------167
   6. Integer Datapath----------------------------------------168
   7. IO Module-----------------------------------------------169
   8. Forwarding Unit-----------------------------------------170
   9. Break Counter-------------------------------------------171
   10. Branch Control------------------------------------------172
   11. Flag Counter (Int. using RETI)--------------------------173
   12. Interrupt Servicer (Int. using RETI)--------------------174
   13. Return Servicer (Int. using RETI)-----------------------175
2. Additional Discussion and/or Comments----------------------176
3. USB Flash Drive (contains project files)-------------------177

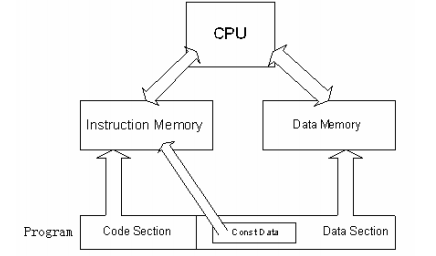
**Abstract:**

This document outlines the basic operation of the MS-32 ISA. MS-32 is a 32-bit reduced instruction set architecture based on the MIPS architecture. This document includes topics such as: Harvard Memory Architecture and Organization, Machine Register Set, Data Types, Addressing Modes, the Instruction Set itself, Interrupts and Reset, the Verilog Implementation with verification, and hardware implementation diagrams.

II. Instruction Set Architecture

Harvard Memory Architecture and Organization

Harvard architecture contains separate pathways for our data and our instructions. The instruction memory is stored separate from the data memory. For this architecture, each address, data or instruction, is 32-bits wide.



## “HARVARD COMPUTER ARCHITECTURE – Konstantin S. Solnushkin”

## “Both instruction and data memory modules are “byte addressable,” where each memory location holds one byte. Thus, 32-bit memory operands will be stored in four consecutive memory locations in “big endian” format – that is, the most significant 8-bits will be stored at the low address and there least significant bits will be stored at the high address. “ – R.W. Allison

## A visual depiction is shown here

## 

## “Instruction Set Architecture – R.W. Allison”

## Instruction Memory:

## This memory will contain the all of the instructions needed to execute the relevant program. In this project, we will be loading the memory manually with a Xilinx’s “$readmemh” tool.

## Data Memory:

## Initially this memory will be empty. The only way to access this memory will be through instructions Load Word (LW) and Store Word (SW). However, for this project, we will be loading the memory manually with a Xilinx’s “$readmemh” tool. After it is loaded it can be modified depending on the instruction memory.

## I/O Memory:

## Initially this memory will be empty. The only way to access this memory will be through enhanced instructions INPUT and OUTPUT.

Machine Register Set

**32-integer registers (all 32-bits wide)**

We have 32 general purpose 32-bit registers. We can use these registers to store values for operands as well as addresses for jumps.

**32-bit Program Counter**

Our 32-bit Program Counter contains the address of the current instruction being executed. Since our addresses are 32-bit wide, the Program Counter increments by 4 bytes after each instruction is finished executing. The Program Counter can be loaded as well, for instructions such as jumps or branches.

**5-bit Flags Register (IE, C, V, N, Z)**

IE – interrupt enable flag

iff IE is high, interrupts are valid

C – carry flag

Carry flag is high if the result of the operation is over 32-bits

V – overflow flag

Overflow flag is high when we the result has an unexpected, invalid sign change

N – negative flag

Negative flag is high if the data type is signed and the sign bit is a 1

Z – zero flag

Zero flag is high if the result of the operation is zero

Data Types

There are two possible data-types, 32-bit signed integers and 32-bit unsigned integers. During operations, our ALU will self-determine whether or not the value it is dealing with needs to be signed. For unsigned integers, overflow will never occur. The bitwise formats of the two data types are shown below

**32-bit signed integer**



“Instruction Set Architecture – R.W. Allison”

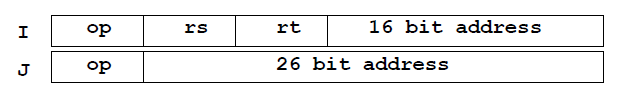
**32-bit unsigned integer**



“Instruction Set Architecture – R.W. Allison”

Addressing Modes

**Immediate (16-bit and 26-bit)**



“Instruction Set Architecture – R.W. Allison”

Immediate addressing mode can only be used with I-format and J-format instructions.

**16-bit (I-format)**

For arithmetic operations, the immediate 16-bit address will be used as an operand to calculate the result of the operation.

For branching operations, the immediate 16-bit address will specify how many instructions to move. It is good to note that this field is a signed address, therefore it is possible to jump forward and backward through instructions. To create their effective target address, the 16-bit field is shift left twice with a zero fill and is concatenated with the four most significant bits of the current program counter.

**26-bit (J-format)**

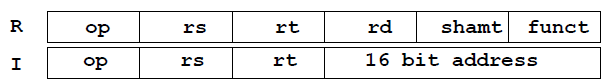
All jump instructions include a 26-bit address. To create their effective target address, the 26-bit field is shift left twice with a zero fill and is concatenated with the four most significant bits of the current program counter.

EXAMPLE:

|  |  |
| --- | --- |
| J  000010 | instr\_index |

The jump address would be {PC[31:29], instr\_index. 2’b00}.

**Register**



“Instruction Set Architecture – R.W. Allison”

Register addressing mode can only be used with R-format and I-format instructions, as J-format does not include register operands.

Register addressing mode is used anytime we want to modify a register in our memory. Registers can hold values to perform arithmetic operations. We can also use the R-Type instruction “Jump Register” to jump to an address specified by a register.

EXAMPLE R-TYPE:

In this case, this R-Type ADD will take R[rs] + R[rt] and place the result in R[rd].

EXAMPLE I-TYPE:

|  |  |  |  |
| --- | --- | --- | --- |
| BEQ  000100 | rs | rt | offset |

In this case, this I-Type BEQ will branch if R[rs] is equal to R[rt].

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | ADD  100000 |

**Register Indirect**



“Instruction Set Architecture – R.W. Allison”

Register Indirect addressing mode and only be used with the load and store.

For register indirect addressing mode, instead of containing a value for arithmetic operations, the register will contain a memory address; said memory address will contain the effective operand to be dealt with. This will be used with our interrupt when we need to push and pop items off the stack using our stack pointer

EXAMPLE:

In this example, we are addressing the base register indirectly. We would be using the base register as an address for our data memory. We would load R[rt] with M[base + offset].

|  |  |  |  |
| --- | --- | --- | --- |
| LW  100011 | base | rt | offset |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | ADD  100000 |

31 26 25 21 20 16 15 11 10 6 5 0 6 5 5 5 5 6

**FORMAT:** ADD rd, rs, rt **MIPS I**

**PURPOSE:**  To add two 32-bit integers.

**DESCRIPTION:** rd ← rs +rt

The 32-bit word value of General Purpose Register (rs) and 32-bit word value of General Purpose register (rt) are added and the 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs +rt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| If (result > 32 bits)  C = 1  else  C = 0 | N = Most significant bit of result (sign bit) | If (signRS=1 & signImm=1 & signResult=0  Or  signRS=0 & signImm=0 & signResult=1)  V = 1  Else  V = 0 | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | ADDU  100001 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** ADDU rd, rs, rt **MIPS I**

**PURPOSE:**  To add two unsigned 32-bit integers.

**DESCRIPTION:** rd ← rs +rt

The 32-bit word value of General Purpose Register (rs) and 32-bit word value of General Purpose register (rt) are added and the 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs +rt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| If (result > 32 bits)  C = 1  else  C = 0 | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | AND  100100 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** AND rd, rs, rt **MIPS I**

**PURPOSE:**  To perform a bitwise logical AND on two 32-bit integers.

**DESCRIPTION:** rd ← rs ANDrt

The 32-bit word value of General Purpose Register (rs) and 32-bit word value of General Purpose register (rt) are combined with a bitwise logical AND. The 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs andrt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| Not possible | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | 00000 | 00000 | 00000 | JR  001000 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** JR rs **MIPS I**

**PURPOSE:**  To jump to an instruction address specified in a register

**DESCRIPTION:** PC ← rs

Jump to the effective target address specified in General Purpose Register (rs)

**RESTRICTIONS:**

The effective target address in General Purpose Register (rs) must be aligned with the current PC. If either of the two least significant bits of rs are not zero, there will be an Address Error.

**OPERATION:**

PC <- rs

**EXCEPTIONS:**

None.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | NOR  100111 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** NOR rd, rs, rt **MIPS I**

**PURPOSE:**  To perform a bitwise logical NOT OR on two 32-bit integers.

**DESCRIPTION:** rd ← rs NORrt

The 32-bit word value of General Purpose Register (rs) and 32-bit word value of General Purpose register (rt) are combined with a bitwise logical NOT OR. The 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs norrt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| Not possible | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | OR  100101 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** OR rd, rs, rt **MIPS I**

**PURPOSE:**  To perform a bitwise logical OR on two 32-bit integers.

**DESCRIPTION:** rd ← rs ORrt

The 32-bit word value of General Purpose Register (rs) and 32-bit word value of General Purpose register (rt) are combined with a bitwise logical OR. The 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs orrt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| Not possible | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | SLT  101010 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SLT rd, rs, rt **MIPS I**

**PURPOSE:**  To set the result of a less than comparison.

**DESCRIPTION:** rd ← (rs < rt)

Compare if the value of General Purpose Register (rs) is less than the value of General Purpose register (rt). The result is placed in the General Purpose Register (rd). If rs is less than rt the result is 1 (true), else the result is 0 (false)

**RESTRICTIONS:**

None.

**OPERATION:**

If (rs < rt)

rd <- 1

else

rd <- 0

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| Not possible | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | SLTU  101011 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SLTU rd, rs, rt **MIPS I**

**PURPOSE:**  To set the result of an unsigned less than comparison.

**DESCRIPTION:** rd ← (rs < rt)

Compare if the unsigned value of General Purpose Register (rs) is less than the unsigned value of General Purpose register (rt). The result is placed in the General Purpose Register (rd). If rs is less than rt the result is 1 (true), else the result is 0 (false)

**RESTRICTIONS:**

None.

**OPERATION:**

If (rs < rt)

rd <- 1

else

rd <- 0

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| Not possible | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | 00000 | rt | rd | sa | SLL  000000 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SLL rd, rt, sa **MIPS I**

**PURPOSE:**  To perform a logical shift left on a word by a set number of bits.

**DESCRIPTION:** rd ← rt <<sa (logical)

The contents of General Purpose Register (rt) are shifted to the left with a zero fill. The shift amount if designated by sa. The 32-bit word result is placed into General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd <- {rt(31-sa)- 0), 0sa}

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| C = last most significant bit to get shifted left | N = Most significant bit of result (sign bit) | If (signRT = 0 & signRD = 1 or signRT = 1 & signRD = 0)  V = 1  Else  V = 0 | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | 00000 | rt | rd | sa | SRL  000010 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SRL rd, rt, sa **MIPS I**

**PURPOSE:**  To perform a logical shift right on a word by a set number of bits.

**DESCRIPTION:** rd ← rt >>sa (logical)

The contents of General Purpose Register (rt) are shifted to the right with a zero fill. The shift amount if designated by sa. The 32-bit word result is placed into General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd <- {0sa, rt(31- (31-sa))}

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| C = last least significant bit to get shifted right | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | SUB  100010 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SUB rd, rs, rt **MIPS I**

**PURPOSE:**  To subtract two 32-bit integers.

**DESCRIPTION:** rd ← rs -rt

The 32-bit word value of General Purpose Register (rs) is subtracted from the 32-bit word value of General Purpose register (rt) and the 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs -rt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| If (result > 32 bits)  C = 1  else  C = 0 | N = Most significant bit of result (sign bit) | If (signRS=0 & signImm=1 & signResult=1  Or  signRS=1 & signImm=0 & signResult=0)  V = 1  Else  V = 0 | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SPECIAL  000000 | rs | rt | rd | 00000 | SUBU  100011 |

31 26 25 21 20 16 15 11 10 6 5 0

6 5 5 5 5 6

**FORMAT:** SUBU rd, rs, rt **MIPS I**

**PURPOSE:**  To subtract two unsigned 32-bit integers.

**DESCRIPTION:** rd ← rs -rt

The 32-bit word value of General Purpose Register (rs) is subtracted from the 32-bit word value of General Purpose register (rt) and the 32-bit result is placed in the General Purpose Register (rd).

**RESTRICTIONS:**

None.

**OPERATION:**

rd ← rs -rt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| If (result > 32 bits)  C = 1  else  C = 0 | N = Most significant bit of result (sign bit) | Not possible | If (result = 0)  Z = 1  Else  Z = 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| BEQ  000100 | rs | rt | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** BEQ rs, rt, offset **MIPS I**

**PURPOSE:**  To compare General Purpose Registers then perform

a PC-Relative conditional branch.

**DESCRIPTION:** if (rs==rt), branch

The offset is shifted left 2 bits to create an 18-bit signed offset. It is then added to the address of the next instruction (instruction after branch). That address is used as the effective address of the branch. If the condition is met, the branch is executed.

**RESTRICTIONS:**

None.

**OPERATION:**

target\_offset ← signExtend({offset,00})

Check\_condition ← (rs == rt)

if (Check\_condition)

PC ← PC + target\_offset

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

Conditional branches have an effective range of ±128Kbytes. To perform a branch that can reach a further destination, use Jump (j) or Jump Register (jr).

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| BNE  000101 | rs | rt | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** BNE rs, rt, offset **MIPS I**

**PURPOSE:**  To compare General Purpose Registers then perform

a PC-Relative conditional branch.

**DESCRIPTION:** if (rs≠rt), branch

The offset is shifted left 2 bits to create an 18-bit signed offset. It is then added to the address of the next instruction (instruction after branch). That address is used as the effective address of the branch. If the condition is met, the branch is executed.

**RESTRICTIONS:**

None.

**OPERATION:**

target\_offset ← signExtend({offset,00})

Check\_condition ← (rs ≠ rt)

if (Check\_condition)

PC ← PC + target\_offset

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

Conditional branches have an effective range of ±128Kbytes. To perform a branch that can reach a further destination, use Jump (j) or Jump Register (jr).

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| BLEZ  000110 | rs | 00000 | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** BLEZ rs, offset **MIPS I**

**PURPOSE:**  To compare a General Purpose Register to ‘0’ then

perform a PC-Relative conditional branch.

**DESCRIPTION:** if (rs ≤ 0), branch

The offset is shifted left 2 bits to create an 18-bit signed offset. It is then added to the address of the next instruction (instruction after branch). That address is used as the effective address of the branch. If the condition is met, the branch is executed.

If the contents of (rs) are less than or equal to (0), (sign bit = 1 or contents == (0)) then the branch is executed at the calculated effective address.

**RESTRICTIONS:**

None.

**OPERATION:**

target\_offset ← signExtend({offset,00})

Check\_condition ← (rs ≤ 0)

if (Check\_condition)

PC ← PC + target\_offset

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

Conditional branches have an effective range of ±128Kbytes. To perform a branch that can reach a further destination, use Jump (j) or Jump Register (jr).

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| BGTZ  000111 | rs | 00000 | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** BGTZ rs, offset **MIPS I**

**PURPOSE:**  To compare a General Purpose Register to ‘0’ then

Perform a PC-Relative conditional branch.

**DESCRIPTION:** if (rs > 0), branch

The offset is shifted left 2 bits to create an 18-bit signed offset. It is then added to the address of the next instruction (instruction after branch). That address is used as the effective address of the branch. If the condition is met, the branch is executed.

If the contents of (rs) are greater than (0), (sign bit = 1 and contents ≠ 0) then the branch is executed at the calculated effective address.

**RESTRICTIONS:**

None.

**OPERATION:**

target\_offset ← signExtend({offset,00})

Check\_condition ← (rs > 0)

if (Check\_condition)

PC ← PC + target\_offset

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

Conditional branches have an effective range of ±128Kbytes. To perform a branch that can reach a further destination, use Jump (j) or Jump Register (jr).

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| ADDI  001000 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** ADDI, rt, rs, immediate **MIPS I**

**PURPOSE:**  To add an immediate constant to a 32-bit integer contained in a General Purpose Register.

**DESCRIPTION:** rt ← rs + immediate

The 16-bit signed immediate constant (sign extended) is added to the 32-bit value in the General Purpose Register (rs) and the result is placed in the General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

rt ← ((signExtend(immediate)) + rs)

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| 1 if result > 32 bits  else 0 | 1 if sign = 1  Else 0 | 1 if signRS=1 & signImm=1 & signResult=0  Or  signRS=0 & signImm=0 & signResult=1  Else 0 | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| SLTI  001010 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** SLTI rs, rt, immediate **MIPS I**

**PURPOSE:**  To get the result of a signed less-than comparison between a General Purpose Register and an immediate constant.

**DESCRIPTION:** rt ← (rs < immediate)

Compares the signed values, General Purpose Register (rs) and the sign extended immediate constant, placing the result in General Purpose Register (rt). If rs is less than the immediate constant, the result is 1, else it is 0.

**RESTRICTIONS:**

None.

**OPERATION:**

if (rs < (signExtend(immediate)))

rt ← 1

else

rt ← 0

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| SLTIU  001011 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** SLTIU rs, rt, immediate **MIPS I**

**PURPOSE:**  To get the result of an unsigned less-than comparison between a General Purpose Register and an immediate constant.

**DESCRIPTION:** rt ← (rs < immediate)

Compares the unsigned values, General Purpose Register (rs) and the zero extended immediate constant, placing the result in General Purpose Register (rt). If rs is less than the immediate constant, the result is 1, else it is 0.

**RESTRICTIONS:**

None.

**OPERATION:**

if (rs < (signExtend(immediate)))

rt ← 1

else

rt ← 0

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| ANDI  001100 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** ANDI rd, rs, immediate **MIPS I**

**PURPOSE:**  To perform a bitwise logical AND with an immediate constant.

**DESCRIPTION:** rt ← rs ANDimmediate

The 16-bit immediate constant is zero extended and a bitwise logical AND is performed with the contents of the General Purpose Register (rs). The result is then placed into General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

rt ← zeroExtend(immediate) & rs

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 if sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| ORI  001101 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:**  ORI rs, rt, immediate **MIPS I**

**PURPOSE:**  To perform a bitwise logical OR with an immediate constant.

**DESCRIPTION:** rt ← rs ORimmediate

The 16-bit immediate constant is zero extended and a bitwise logical OR is performed with the contents of the General Purpose Register (rs). The result of the operation is then placed in the General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

rt ← zeroExtend(immediate) | rs

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 if sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| XORI  001110 | rs | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** XORI rs, rt, immediate **MIPS I**

**PURPOSE:**  To perform a bitwise logical Exclusive OR with an immediate constant.

**DESCRIPTION:** rs ← rt XOR immediate

The 16-bit immediate constant is zero extended and a bitwise logical XOR is performed with the contents of the General Purpose Register (rs). The result of the operation is then placed in the General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

rt ← zeroExtend(immediate) ^ rs

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 if sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| LUI  001111 | 00000 | rt | immediate |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** LUI rt, immediate **MIPS I**

**PURPOSE:**  To load an immediate constant into the upper 16 bits of a word.

**DESCRIPTION:** rt ← {immediate,016}

The 16-bit immediate constant is shifted left 16 bits and concatenated with 16 bits of zero in the low-end 16 bits. The 32-bit result is then placed into General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

rt ← {(shl(immediate,16))|| 016}

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 if sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| LW  100011 | base | rt | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** LW rt, offset(base) **MIPS I**

**PURPOSE:**  To load a word from memory as a signed value.

**DESCRIPTION:** rt ← memory[base+offset]

The base is added to the 16-bit signed offset to create the effective memory address. The word at that address in memory is loaded into the General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

mAddr ← sign\_extend(offset) + register[base]

rt ← memory[mAddr]

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 if sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| SW  101011 | base | rt | offset |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** SW rt, offset(base) **MIPS I**

**PURPOSE:**  To store a word to memory.

**DESCRIPTION:** memory[base+offset] ← rt

The base is added to the 16-bit signed offset to create the effective memory address. The word at that address in memory is loaded with the contents of General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

mAddr ← sign\_extend(offset) + register[base]

memory[mAddr] ← rt

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |
| --- | --- |
| J  000010 | instr\_index |

31 26 25 0

6 26

**FORMAT:** J target **MIPS I**

**PURPOSE:**  To jump to an instruction address within the 256 MB aligned region

**DESCRIPTION:**

The least significant 28 bits of the jump address are 26 bits of instr\_index shifted left twice, with zero fill. The most significant 4 bits of the jump address are the exact most significant 4 bits as the program counter. Jumps to the effective target address

**RESTRICTIONS:**

None.

**OPERATION:**

PC <- {PC(31-28),instr\_index, 00}

**EXCEPTIONS:**

None.

|  |  |
| --- | --- |
| JAL  000011 | instr\_index |

31 26 25 0

6 26

**FORMAT:** JAL target **MIPS I**

**PURPOSE:**  To jump to a procedure call within the 256 MB aligned region

**DESCRIPTION:**

Stores the current Program Counter after the call instruction into General Purpose Register 31. The least significant 28 bits are 26 bits of instr\_index shifted left twice, with zero fill. The most significant 4 bits of the jump address are the exact most significant 4 bits as the program counter. Jumps to the effective target address.

**RESTRICTIONS:**

None.

**OPERATION:**

$r31 <- PC + 4

PC <- {PC(31-28),instr\_index, 00}

**EXCEPTIONS:**

None.

|  |  |  |  |
| --- | --- | --- | --- |
| INPUT  011100 | 00000 | rt | input |

31 26 25 21 20 16 15 0 6 5 5 16

**FORMAT:** INPUT rt, input **MIPS I**

**PURPOSE:**  To load an input value into a General Purpose Register.

**DESCRIPTION:** rt ← input

The 16-bit signed input data will be sign extended and placed into General Purpose Register (rt).

**RESTRICTIONS:**

None.

**OPERATION:**

Rt ← signExtend(input)

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | 1 If sign = 1  Else 0 | X | 1 if result = 0  Else 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| OUTPUT  011101 | 00000 | rt | output |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** OUTPUT rt, output **MIPS I**

**PURPOSE:**  To output a General Purpose Register’s contents.

**DESCRIPTION:** output ← rt

The 32-bit signed General Purpose Register (rt) data will be outputted.

**RESTRICTIONS:**

None.

**OPERATION:**

Output(rt)

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

|  |  |  |  |
| --- | --- | --- | --- |
| RETI  011110 | 00000 | 00000 | 016 |

31 26 25 21 20 16 15 0

6 5 5 16

**FORMAT:** RETI **MIPS I**

**PURPOSE:**  To return from an interrupt service routine.

**DESCRIPTION:** PC ← PC before ISR, flags ← flags before ISR

The flags that were pushed to the stack when the interrupt happened will be popped and placed back into the respective flag registers. Then, the PC address that was pushed to the stack when the interrupt happened will be popped and placed back into the PC.

**RESTRICTIONS:**

None.

**OPERATION:**

Flags ← flags\_before\_isr

PC ← pc\_before\_isr

**EXCEPTIONS:**

None.

**PROGRAMMING NOTES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Carry** | **Negative** | **Overflow** | **Zero** |
| X | X | X | X |

**Interrupts:**

The CPU that will execute the instructions specified in this ISA will have a minimum of one external interrupt request (INTR) input and one interrupt acknowledge (INTA) output. In response to an external interrupt request, the processor must save the state of the machine by “pushing” the PC and Flags register onto the stack. The Program Counter (PC) will be loaded with the “interrupt vector” stored in the last location of Data Memory (i.e. address 3FC­h­).

PC ← dM[3FCh] //retrieve interrupt vector

The “return-from-interrupt” instruction (RETI) must “Pop,” in reverse order, the Flags register and PC from off the stack.

**Reset:**

In response to an external reset input, the processor will do the following:

PC← 0000\_0000h, $sp ← 0000\_03FCh, //initialize PC and $sp registers

Flags(IE, N, Z, O, C) ← 5’b000002  //initialize Flags Register

**\*The Interrupt and Reset Section of this document reflect requirements created by Robert Allison, Professor at CSULB.**

**V. Additional Discussion and/or Comments**

Complications and solutions:

* To implement pipelining, a forwarding unit is used to detect when an operation requires the result of a prior instruction.
* The break instruction stops execution too early and doesn’t allow prior instructions to finish. A counter is used to let the processor finish before the simulation is ended.
* The flags don’t get placed into the present state flags register at the correct time when returning from an interrupt with RETI instruction. A counter is used to wait the proper amount of time before placing the flags in the register.
* To implement the RETI instruction, a separate control unit (Return Servicer) is used which consists of a counter and an array of instructions and control signals to accomplish the necessary tasks.
* A similar concept is used when we see an interrupt (memory module 14 only, Interrupt Servicer).
* Most of the above solutions are not ideal and may slow the system down marginally. Overall, the speed increase of the pipelined versus multicycle architecture is approximately three-fold.