**Project Lab 2: CISC-24**

Eric Pires

Khang Nguyen

ECE 368

Team 2

Lab Date: April 7th, 2017

Due Date: May 5th, 2017



**Table of Contents**

[Introduction 2](#_Toc481755252)

[Methods and Testing Procedures 3](#_Toc481755253)

[Output Result Signal Multiplexing 3](#_Toc481755254)

[Addressing Mode Implementation 3](#_Toc481755255)

[Jump Instruction Mode Implementation 5](#_Toc481755256)

[Branch and Reset Instruction Mode Implementation 6](#_Toc481755257)

[Debug Unit Implementation 7](#_Toc481755258)

[Results 9](#_Toc481755259)

[Output Result Signal Multiplexing 9](#_Toc481755260)

[Addressing Mode Implementation 11](#_Toc481755261)

[Jump Instruction Mode Implementation 12](#_Toc481755262)

[Branch and Reset Instruction Mode Implementation 13](#_Toc481755263)

[Debug Unit Implementation 14](#_Toc481755264)

[Discussion 17](#_Toc481755265)

[Conclusion 19](#_Toc481755266)

[Reflection 20](#_Toc481755267)

[References 22](#_Toc481755268)

[Datasheet 23](#_Toc481755269)

[Random Access Memory (RAM) 8x24 Read and Write Unit 23](#_Toc481755270)

[Program Counter (PC) Unit 25](#_Toc481755271)

[Program Counter Latch 26](#_Toc481755272)

[Program Counter MUX (PC\_MUX) 27](#_Toc481755273)

[Instruction Register (IR) 28](#_Toc481755274)

[Decode Module 29](#_Toc481755275)

[General Purpose Register Unit 30](#_Toc481755276)

[Tri- state buffer 31](#_Toc481755277)

[Arithmetic logic Unit (ALU) 32](#_Toc481755278)

[Accumulator (AC) 36](#_Toc481755279)

[Branch Compare 37](#_Toc481755280)

[Status Register 38](#_Toc481755281)

[Debug MUX 39](#_Toc481755282)

Control Unit…………………………………………………………………………………...40

Abstract

This lab consisted of the completion of the CISC-24 Instruction Set Architecture (ISA) as well as the Debug Unit for the final Hardware Test. The previous design was modified to accommodate the new instructions in addition to detecting for all four addressing modes. Software simulation was conducted to verify the logic implementation before implementing on the Nexys 2 board. Datasheets for the CISC machine and the various components were constructed.

# Introduction

The goal of this lab was to complete the ISA of the CISC-24 machine and to fully implement the Debug Unit as a method of hardware demonstration. The newly implemented instructions involved the various addressing mode types, jump routines, branching and a system reset. In addition, detailed datasheets were written to summarize the functionality of the CISC machine. Once implemented, the CISC design was ported to the Nexys board for hardware testing. To complete this project, the following tools were used:

* Xilinx ISE Software
* ISim Simulation Software
* Digilent Adept Software
* Digilent Waveforms Software
* Nexys 2 FPGA
* Analog Discovery Logic Analyzer

# Methods and Testing Procedures

## Output Result Signal Multiplexing

In order to simplify the process of confirming the system output for all instructions, a multiplexor was added to select between three possible output signals dependent of the instruction. All arithmetic computations are stored within the accumulator. Register to Register and Memory to Register instructions are all stored within the input signal of the GPR. Memory to Memory and Register to Memory instructions are stored within the input data signal of RAM. All three of these signals were passes as inputs to the multiplexor as well as a select line and enable triggered by the control unit. The enable line was added to prevent the display of unnecessary input changes. Despite frequently changing throughout the datapath, the only time the signals are valid is during the write back state.

To test this, the simulation test bench was run and only the CPU\_OUT port was observed from the waveform. The results were matched to the expected outcomes based on the program written in RAM. Table 2, located at the end of the Methods and Procedures section, demonstrates the steps taken to accomplish this.

## Addressing Mode Implementation

Upon completion of the first lab, addressing mode detection was ignored in the CISC design. Therefore, the first step in the lab was to implement these features into the data path. The addressing mode is a two-bit signal that signifies how the register data is accessed, The following chart shows the four possible modes that can occur for instructions involved with register manipulation as well as a brief description of its functionality.

|  |  |  |
| --- | --- | --- |
| Signal | Operation | Description |
| 00 | Register Direct | Contents of selected register are treated as direct values to manipulate in datapath. |
| 01 | Register Indirect | Contents of selected register are treated as an address location to RAM. Contents of this RAM address is indirectly used to manipulate in datapath |
| 10 | Register Direct Auto Increment | Same as Register Direct, but contents are incremented by one |
| 11 | Register Indirect Auto Increment | Same as Register Indirect, but contents are incremented by one |

By accounting for the addressing mode, many of the previously implemented instructions now contain 3 additional variations. To start, the Register Indirect mode was implemented. This was accomplished by editing the CONTROLLER module. Inside the operand access states, a conditional statement was added to compare the low bit of the addressing mode signal. If the low bit was high, then the register mode became indirect. This indicates that the contents of the register are an address in memory to which the ALU should take in as an input. This involved setting the select signal of the ALU input to be the data output of RAM. In addition, the instruction input to the RAM module was selected to be the contents of the specified register.

For one operand instructions, the only ALU input involved is RA. Therefore the second input stays the same, as it does not involve any other addressing mode. However, two operand instructions require both register inputs. Therefore, both RA and RB inputs must both check for the addressing mode signal and must read from memory twice in the case that both registers are addressed indirectly. To achieve a successful outcome, the second register input cannot be manipulated at the same clock cycle as the first. If this happens, then both inputs will be given the same ram output value. The second input must check the addressing mode on a different clock cycle to properly read from memory and access the correct data. To accomplish this, the conditional statement mentioned in the previous paragraph was pasted inside the top\_exec state, separate of the operand access state. This inadvertently increased the timing for the write back state to properly compute the end result. Therefore, the GPR\_WRITE signal was also extended by one clock cycle to accommodate for this delay.

After the indirect mode was implemented, the auto increment mode was added. To check for this, another condition statement was written to compare the high bit of the addressing mode signal. If high, then the GPR\_MUX would take the direct or indirect result and increment it by one. Setting the most significant but of the select line high took these values and incremented them by one prior to writing back into GPR module.

To insure the functionality of the addressing mode, several instructions were added to RAM. These consisted of both one and two operand instructions with all possible addressing modes. The simulation test bench was then run to confirm the functionality of the implementation. Table 2, located at the end of the Methods and Testing Procedures section, demonstrates the steps taken to accomplish this.

## Jump Instruction Mode Implementation

The next implementation involved the jump operation. The instructions included JMPI ADDR, JSR ADDR, and RSR. The first instruction was easily implemented by pointing the instruction to the stop state after being decoded. Since no computation or memory management is being done, the controller will access the immediate address as the next memory location to be fetched. The PC modules OP input was set to “10” which points to the 19-bit immediate signal. The next instruction operated similarly to this one, but required the previous address to be saved prior to jumping. Instead of constructing a stack to hold the previous address, a latch was used to temporarily store the previous program count plus one. Adding by one will progress the PC to fetch the following instruction from where the user left off. Doing so will prevent the same JMP command from being executed over again. Despite working for this simple purpose, the implementation is not sufficient for the requirements of this project. The downside of this implementation will be presented in the Discussion. To return back to the original program count, the latched address was fed to a multiplexor that selected between the latch output and the decoded 19-bit immediate signal. This was then sent to the PCIN port of the PC module, requiring another control unit output signal to drive the component. Details of this component are demonstrated in the datasheet.

Upon implementation of these three commands, the simulation test bench was run to monitor the output and program count throughout execution of each instruction. Table 2, located at the end of the Methods and Testing Procedures section, demonstrates the steps taken to accomplish this.

## Branch and Reset Instruction Mode Implementation

The next instruction to be implemented involved branching and a system reset. The instruction included BR MASK OFFSET, CLRS, and RESET. The first instruction required a modification of the PC module. Since branching was not define as a potential operation in the case statement for the OP input, one was implemented to take the current address and increment it by an offset. Therefore, an input for this offset value was added to the PC labeled OFST. This port takes the last 15 decoded bits of the instruction and concatenates in with nine zero bits to turn it into a 24-bit signal to allow the addition operator to work. This case was selected for whenever the OP value was set to 11. Since the branch case only incurs whenever the mask bits are equal to the status bits, a new component was created to simply compare the two values and send a high bit output when equal. This component, BRANCH\_COMPARE, can be found in the datasheet. After this, the control unit was modified to receive an additional input, MASK\_CMP, that tied to the output of the BRANCH\_COMPARE. The instruction was then added to the control unit op\_sel case. When detected, the stop case is called to adjust the PC value only if the MASK\_CMP bit is set. If this bit is not high, then the PC will be incremented by only one and will thus not branch, but proceed to the next memory location.

After implementing the branch instruction, the CLRS and RESET instructions were also added. This instruction simply cleared the output status bits when called. To implement this, a modified register file was created specifically for the status bits. This register, which can be found in the datasheet, latches a four-bit value an also contains an additional input bit CLR. This bit allows the control unit to specifically rese only this register as opposed to setting the global reset. This reset ties both software and hardware resets by performing an OR on the input reset switch on hardware and the software enabled reset bit assigned in the control unit.

## Debug Unit Implementation

The debug unit was designed to easily observe the CPU\_OUT signal on the Nexys board. An illustration of this unit can be found in the datasheet along with its functional inputs and outputs. To summarize, the unit uses a button input to simulate the system clock along with a switch to trigger the reset line. The last 16 bits of the output signal is brought out onto the seven-segment display. The remaining bits sent to the to Pmod connectors to be read by the Logic Analyzer. The status​ bits are passed to the on-board LEDs. The unit works when the user actuates the system clock. Upon completion of a given instruction, the segment display and logic analyzer senses the change and displays the output in hex form though the seven-segment display driver and Waveforms software. By doing this, each instruction can be carefully executed and the overall timing can be analyzed by multiplying the necessary button presses by the system clock (20ns).

Upon implementation, the design was synthesized and flashed onto the Nexys Board for hardware testing. To accomplish this, a ucf file was generated to map the necessary system I/O. The system clock was tied to BTN0, the segment clock was tied to the board’s crystal, the system reset was tied to switch 0, the segment display outputs were tied to the corresponding inputs on the board and the remaining 8 output bits and ALU status bits were tied to the Pmod connectors and LEDs respectively. This file can be found in the datasheet. The steps take to upload the code onto hardware can be found in Table 3 at the end of the Methods and Testing Procedures section. A table of the test program can be found in the Results.

*Table 2: Software Simulation Testing Procedure*

|  |  |
| --- | --- |
| **Step Description** | **Procedure Description** |
| Edit and Synthesize Code | Visually inspect provided code and incorporate missing vhd files from previous experiments. Run the synthesis tool to ensure the code is syntactical and functioning. |
| Simulate Behavioral Model | Use iSim to observe the triggered clock input and the output of each instruction |
| Match results with expected values | Use table to match instruction results with expected outcomes. Also ensure the timing of these results by observing the state location at the time the result appears. |

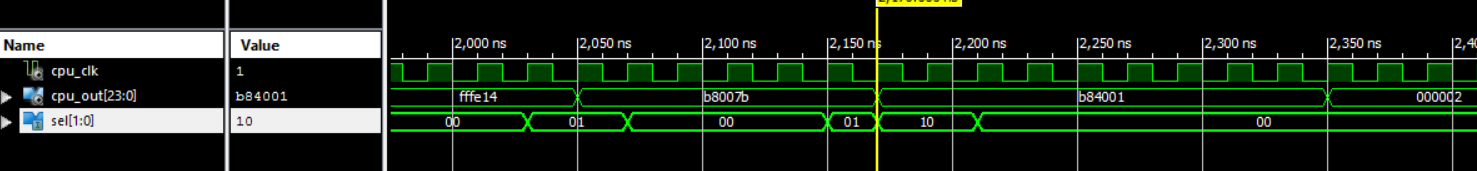
*Table 3: Hardware Testing Procedure*

|  |  |
| --- | --- |
| **Step Description** | **Procedure Description** |
| Edit and Synthesize Code | Visually inspect provided code and incorporate missing vhd files from previous experiments. Run the synthesis tool to ensure the code is syntactical and functioning. |
| Generate Programming File | Under the synthesis tool, run *Generate Programming File* to generate a bootable bit file for hardware implementation on the Nexys board. Upon completion, use the Digilent Adept tool to burn this file onto the FPGA’s RAM |
| Reset system | Flip the first switch high to clear out any garbage values prior to testing |
| Simulate system clock | Flip the first switch low to disable the reset, press the first button until the output displayed on the segment display changes. Observe the amount of presses required for subsequent instructions. Repeat for all available instructions. |

# Results

## Output Result Signal Multiplexing

To test the implementation of the output multiplexing, a test program was run containing all three possible writeback scenarios. These include arithmetic operations, memory write back, and register write back. The following results were gathered based on the simulation. The following image is a sample of how the data was collected.



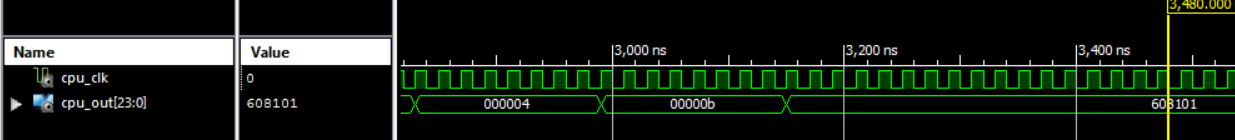
At 2,050 ns, the CPU\_OUT value is selected by the sel line to be 01, which points to the GPR input port. This corresponds to a register writeback. At 2,170 ns, the CPU\_OUT value is selected by the select line to be 10, which points to the data in port of RAM. This corresponds to a memory writeback. Finally, at 2,350 ns, the select line is 00 while the CPU\_OUT value is 2. This points to the output of the accumulator, corresponding to an ALU operation. All three input options were tested and confirmed given the following instruction set. The sample only shows a small portion of the simulated program. The following table shows the entirety of this program. Note that not all instructions are present, but the program invokes all three forms of writeback and contains register indirect addressing for both register inputs on a two-operand instruction.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Hex Input** | **RA Data** | **RB Data** | **MemA Data** | **MemB Data** | **Output Destination** | **Expected Result** | **Observed Result (Hex)** | **Pass/**  **Fail** |
| ADDI R0 123 | B8007B | 0 | NA | N/A | N/A | R0 | 123(7B) | 7B | Pass |
| ADDI R1 1 | B84001 | 0 | N/A | N/A | N/A | R1 | 1 | 1 | Pass |
| ADD R0 R1 | 800200 | 123 | 1 | N/A | N/A | R0 | 124(7C) | 7C | Pass |
| SUB R0 R1 | 880200 | 124 | 1 | N/A | N/A | R0 | 123(7B) | 7B | Pass |
| CLR R1 | 204000 | 1 | N/A | N/A | N/A | R1 | 0 | 0 | Pass |
| INC R0 | 280000 | 123 | N/A | N/A | N/A | R0 | 124(7C) | 7C | Pass |
| DEC R0 | 300000 | 124 | N/A | N/A | N/A | R0 | 123(7B) | 7B | Pass |
| SLL R0 x"B" | 40000B | 123 | N/A | N/A | N/A | R0 | 984(3D8) | 3D8 | Pass |
| SRL R0 1 | 480001 | 984 | N/A | N/A | N/A | R0 | 492(1EC) | 1EC | Pass |
| NEG R0 | 380000 | 492 | N/A | N/A | N/A | R0 | 492(F1EC) | FFF1EC | Pass |
| MVS R0 R3 | 500400 | -492 | 0 | N/A | N/A | R3 | 492(F1EC) | FFF1EC | Pass |
| MSM R0 x"FF" | 6080FF | -492 | N/A | N/A | N/A | RAM[FF] | -492(F1EC) | FFF1EC | Pass |
| MMS R1 x"0" | 684000 | N/A | N/A | N/A | x"B8007B" | R1 | x"B8007B" | B8007B | Pass |
| MVMI x"01" x"FF" | 5802FF | N/A | N/A | x"B84001" | N/A | RAM[FF] | x"B84001" | B84001 | Pass |
| ADDI R0 494 | B801EE | -492 | N/A | N/A | N/A | R0 | 2 | 2 | Pass |
| ADDI R7 5 | B9C005 | 0 | N/A | N/A | N/A | R7 | 5 | 5 | Pass |
| ADD [R0] [R7] | 821E00 | N/A | N/A | x"800200" | x"280000" | R0 | x"A80200" | A80200 | Pass |
| RST | 100000 | N/A | N/A | N/A | N/A | CPU\_OUT | 0 | 0 | Pass |

## Addressing Mode Implementation

To test the addressing mode implementation, additional commands were added to the test program detailed in the Output Result Signal Multiplexing section. Since the program had already included and confirmed the implementation of both register direct and indirect addressing modes on a two-operand instruction, the instructions only tested for auto increment direct and indirect addressing modes on both one and two-operand instructions. The following instructions were then added to the program memory (before the reset command).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Hex Input** | **RA Data** | **RB Data** | **MemA Data** | **MemB Data** | **Output Destination** | **Expected Result** | **Observed Result (Hex)** | **Pass/**  **Fail** |
| ADDI R6+ 3 | BD8003 | 0+1 | N/A | N/A | N/A | R6 | 4 | 4 | Pass |
| ADD R6+ R7+ | 85AE00 | 4+1 | 5+1 | N/A | N/A | R6 | 11 | B | Pass |
| ADDI [R6]+ 1 | BF8001 | N/A | N/A | 6080FF +1 | N/A | R6 | x“608101” | 608101 | Pass |

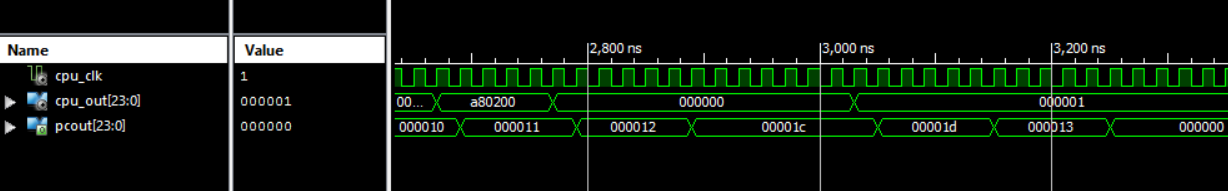


Here, the first command accesses the contents of the sixth register and adds it by one. The result will then be incremented by one. This instruction finishes at 2,640 ns The second command does the same but for two register inputs. This instruction finishes at 2,880 ns. Finally, the last instruction adds the contents of the addressed memory location by the immediate value and an additional bit. This finishes at 2,260 ns.

## Jump Instruction Mode Implementation

To test the jump instructions, each of three implemented instructions were added to the program memory used in the prior two sections. This included a JSR ADDR, RSR, and JMPI ADDR instruction. The RSR instruction was complimented with an ADDI instruction and was placed away from the other instructions in locations 1C and 1D. The CPU\_OUT signal and the PC value was observed for each instruction. The instruction table and waveform output can be found below. Note that the instructions were performed after the clearing R0.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Hex Input** | **RA Data** | **RB Data** | **MemA Data** | **MemB Data** | **Output Destination** | **Expected Result** | **Observed Result (Hex)** | **Pass/**  **Fail** |
| JSR 1C | D0001C | N/A | N/A | N/A | N/A | RAM[1C] | PC = 1C | PC = 1C | Pass |
| ADDI R0 1 | B80001 | 0 | N/A | N/A | N/A | R0 | 1 | 1 | Pass |
| RSR | D80000 | N/A | N/A | N/A | N/A | RAM[13] | PC = 13 | PC = 13 | Pass |
| JMPI 00 | C80000 | N/A | N/A | N/A | N/A | RAM[00] | PC = 00 | PC = 00 | Pass |

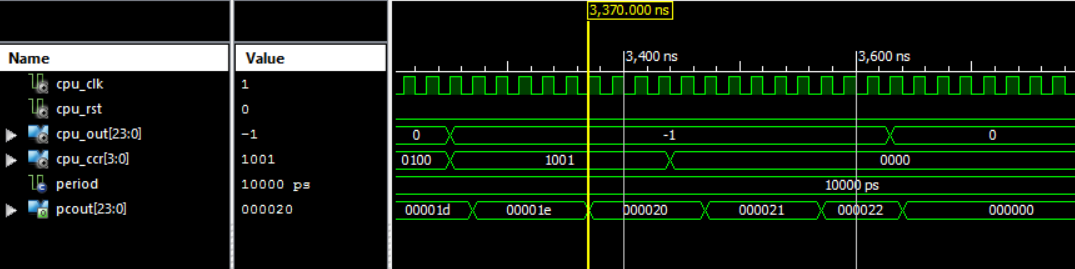
Around 2,900 ns, PCOUT changes from its original path and jumps to location 1C in ram. This shows that the PC has successfully jumped to a new location in memory. It then begins to increment normally after 3,040 ns, where the PC increments by one. At 3,160 ns, the PC was properly latched after calling the JSR instruction. This can be observed by the returned PC value on the RSR instruction. This was one plus the latched address (12+1 = 13). Therefore, it continues on the original instruction path after jumping to 1C. The other jump command, JMPI, does not save the previous PC value. Instead, it performs a direct jump to the immediate address and forgets

the current path of the PC.

## Branch and Reset Instruction Mode Implementation

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Hex Input** | **RA Data** | **RB Data** | **MemA Data** | **MemB Data** | **Output Destination** | **Expected Result** | **Observed Result (Signed)** | **Pass/**  **Fail** |
| JSR 1C | D0001C | N/A | N/A | N/A | N/A | RAM[1C] | PC = 1C | PC = 1C | Pass |
| CLR R7 | 21C000 | 5 | N/A | N/A | N/A | R7 | 0 | 0 | Pass |
| SUBI R7 1 | C00001 | 0 | N/A | N/A | N/A | R7 | -1 | -1 (1001 SB) | Pass |
| BR 1001 02 | E48002 | N/A | N/A | N/A | N/A | RAM[1F or 20] | RAM[20] | RAM[20] | Pass |
| CLRS | F80000 | N/A | N/A | N/A | N/A | CPU\_CCR | 0 | 0 | Pass |
| BR 1001 02 | E48002 | N/A | N/A | N/A | N/A | RAM[22 or 23] | RAM[22] | RAM[22] | Pass |
| RESET | 100000 | N/A | N/A | N/A | N/A | 0 for OUT, RAM[] | 0,RAM[0] | 0,RAM[0] | Pass |

To test the branching and reset instructions, the three were added into the used program memory from the last section. The newly implemented instructions subtracted the contents of R7 by one to force the negative and carry status bits high. This was then offset two locations ahead with a mask of 1001 (which signifies that negative and carry flags are high). Two memory locations ahead, the status bits are then cleared, followed by another branch request. The branch request is then followed by a system reset and a halt command. The following table shows these instructions along with the waveform output.



As seen above, the SUBI instruction return -1 (Port radix set to signed decimal) at 3,250 ns. The status bits can be seen on the CPU\_CCR as 1001. When the branch command is executed after the last command, the branch succeeds and can be confirmed by the new output value seen on PCOUT. 1E was incremented by an offset of 2, which brings the PC up to 20 at 3,370 ns. At 3,440 ns, the CLRS instruction completes and the status bits are all set low at this time. Therefore, when the branching instruction was called once more, the status did not match the mask. This can be seen at 3,570ns, since the PCOUT is only one address ahead instead of two. This means that the next instruction, reset, is run. The reset instruction finishes at 3,640ns and can be confirmed successful by observing the change within CPU\_OUT and PCOUT. Now, the program counter has been reset to 0 along with the output of the system.

## Debug Unit Implementation

To finalize the design, the program instructions were combined and implemented for the final test run. This program encompassed all instructions that were implemented for the design. Addressing mode implementations were done once for each operand style. Testing every single instruction that uses a register was not necessary, as they all fall under the same category of operand and will undergo the same process. The following table shows the output values on both simulation and hardware.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Hex Input** | **RA Data** | **RB Data** | **MemA Data** | **MemB Data** | **Output Destination** | **Simulated Result** | **Observed Result (Hex)** | **Pass**  **Fail** |
| ADDI R0 123 | B8007B | 0 | NA | N/A | N/A | R0 | 123(7B) | 7B (1010) | P |
| ADDI R1 1 | B84001 | 0 | N/A | N/A | N/A | R1 | 1 | 1 (0000) | P |
| ADD R0 R1 | 800200 | 123 | 1 | N/A | N/A | R0 | 124(7C) | 7C | P |
| SUB R0 R1 | 880200 | 124 | 1 | N/A | N/A | R0 | 123(7B) | 7B | P |
| CLR R1 | 204000 | 1 | N/A | N/A | N/A | R1 | 0 | 0 | P |
| INC R0 | 280000 | 123 | N/A | N/A | N/A | R0 | 124(7C) | 7C | P |
| DEC R0 | 300000 | 124 | N/A | N/A | N/A | R0 | 123(7B) | 7B | P |
| SLL R0 x"B" | 40000B | 123 | N/A | N/A | N/A | R0 | 984(3D8) | 3D8 | P |
| SRL R0 1 | 480001 | 984 | N/A | N/A | N/A | R0 | 492(1EC) | 1EC | P |
| NEG R0 | 380000 | 492 | N/A | N/A | N/A | R0 | -492(FE14) | FFF1EC | P |
| MVS R0 R3 | 500400 | -492 | 0 | N/A | N/A | R3 | -492(FE14) | FFF1EC | P |
| MSM R0 x"FF" | 6080FF | -492 | N/A | N/A | N/A | RAM[FF] | -492(FE14) | FFF1EC | P |
| MMS R1 x"0" | 684000 | N/A | N/A | N/A | x"B8007B" | R1 | x"B8007B" | B8007B | P |
| MVMI x"01" x"FF" | 5802FF | N/A | N/A | x"B84001" | N/A | RAM[FF] | x"B84001" | B84001 | P |
| ADDI R0 494 | B801EE | -492 | N/A | N/A | N/A | R0 | 2 | 2 (1001) | P |
| ADDI R7 5 | B9C005 | 0 | N/A | N/A | N/A | R7 | 5 | 5 (0000) | P |
| ADD [R0] [R7] | 821E00 | N/A | N/A | x"800200" | x"280000" | R0 | x"A80200" | A80200 (0100) | P |
| CLR R0 | 200000 | A80200 | N/A | N/A | N/A | R0 | 0 | 0 | P |
| JSR 1C | D0001C | N/A | N/A | N/A | N/A | RAM[1C] | PC = 1C | PC = 1C | P |
| CLR R7 | 21C000 | 5 | N/A | N/A | N/A | R7 | 0 | 0 | P |
| SUBI R7 1 | C00001 | 0 | N/A | N/A | N/A | R7 | -1 | -1 (1001) | P |
| BR 1001 02 | E48002 | N/A | N/A | N/A | N/A | RAM[1F or 20] | RAM[20] | RAM[20] | P |
| CLR R7 | 21C000 | 5 | N/A | N/A | N/A | R7 | 0 | 0 | P |
| CLRS | F80000 | N/A | N/A | N/A | N/A | CPU\_CCR | 0 | 0 | P |
| BR 1001 02 | E48002 | N/A | N/A | N/A | N/A | RAM[22 or 23] | RAM[22] | RAM[22] | P |
| ADDI R7 5 | B9C005 | 0 | N/A | N/A | N/A | R7 | 5 | 5 | P |
| ADDI R6+ 3 | BD8003 | 0+1 | N/A | N/A | N/A | R6 | 4 | 4 | P |
| ADD R6+ R7+ | 85AE00 | 4+1 | 5+1 | N/A | N/A | R6 | 11 | B | P |
| ADDI [R6]+ 1 | BF8001 | N/A | N/A | 6080FF +1 | N/A | R6 | x“608101” | 608101 | P |
| ADDI R4 FF | B900FF | 0 | N/A | N/A | N/A | R4 | FF | FF | P |
| ADDI R5 F | B9400F | 0 | N/A | N/A | N/A | R5 | F | F | P |
| AND R4 R5 | A10A00 | FF | F | N/A | N/A | R4 | F | F | P |
| OR [R4]+ R7 | AF0E00 | N/A | F | B9C005+1 | N/A | R4 |  |  | F |
| XOR R4 R5 | B10A00 | 000E01 | F | N/A | N/A | R4 |  |  | P |

Some issues were noted while executing this test. For one, the status bits would sometimes modify during instructions that did not involve the ALU. Since this component contained no enable line, it always operates and will compute values that are not specified by the instruction written in memory. Another issue was observing the output of the MMS command. The output can only be viewed on the rising edge of the clock. When the button is released, the output immediately changes value. It still appears and compares with the simulation, but is modified immediately when the clock edge falls. Apart from these issues, the instruction results all matched to the simulated results.

Since the use of an actuated system clock does not provide sufficient evidence of the system running at a desirable clock speed, the approach taken to estimate the performance of the system was to generate a timing report. This was done by allotting a timing constraint of 20ns (50Hz). Since the output signals cannot be shown at a perceptible rate if the system operated on this clock, the following analysis provided a detailed report of timing given the hardware specification.

Data Sheet report:

-----------------

All values displayed in nanoseconds (ns)

Clock to Setup on destination clock CPU\_CLK

---------------+---------+---------+---------+---------+

| Src:Rise| Src:Fall| Src:Rise| Src:Fall|

Source Clock |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|

---------------+---------+---------+---------+---------+

CPU\_CLK | 15.829| 7.906| 7.957| 6.291|

---------------+---------+---------+---------+---------+

Timing summary:

---------------

Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)

Constraints cover 834777 paths, 0 nets, and 5148 connections

Design statistics:

Minimum period: 15.914ns{1} (Maximum frequency: 62.838MHz)

------------------------------------Footnotes-----------------------------------

1) The minimum period statistic assumes all single cycle delays.

Analysis completed Wed May 03 19:49:57 2017

----------------------------------------------------------------------------

To summarize the report, the minimum timing constraint given both logic and routing delays in 15.829 ns and only one constraint did not meet this time. This ran at 15.914 ns, which still falls under the required 20ns clock speed. Therefore, based on the generated report, it can be assumed that the architecture falls within the timing constraint of 20ns.

To estimate the mean time requirement for completion of an instruction, the following table shows the time taken for each instruction format acquired from observing the amount of button presses needed to finish each instruction.

|  |  |  |
| --- | --- | --- |
| **Instruction Type** | **Button Cycles** | **Total Time**  **(Cycles \* 20)** |
| OOP | 8 | 160ns |
| TOP | 6 | 120ns |
| JUMP | 5 | 100ns |
| Branch | 6 | 120ns |
| Memory to Memory Move | 7 | 140ns |
| Register to Memory Move | 6 | 120ns |
| Memory to Register | 7 | 140ns |
| Register to Register | 7 | 140ns |
| **Mean Time** | -- | **130ns (6.5 cycles)** |

# Discussion

The CISC-24 ISA called for a total of 88 instructions (Including all Addressing Modes) involving one operand, two operands, jump and branching operations with an Interrupt service routine. A total of 75 out of 89 instructions was implemented, completing 84% of the required ISA. The following table shows the instructions that were omitted from the final design.

|  |  |  |
| --- | --- | --- |
| **OP** | **Instruction** | **Reason for Omission** |
| 10010 | MUL SrcA, SrcB | Required heavy modification of ALU, stability of ALU took precedence over redesign. |
| 10011 | DIV SrcA, SrcB | Required heavy modification of ALU, stability of ALU took precedence over redesign. |
| 00011 | BLMR | Insufficient time allotted for implementation. Stability of final design took precedence. |
| 01110 | BLRM MemB | Insufficient time allotted for implementation. Stability of final design took precedence. |
| 01111 | BLMR MemB | Insufficient time allotted for implementation. Stability of final design took precedence. |
| 11101 | INTTGL | Interrupts not implemented Insufficient time allotted for implementation. Stability of final design took precedence. |
| 11110 | RTI | Insufficient jump implementation could not utilize this command. Stability of final design took precedence. |

The report previously mentioned an issue with the PC latching module. The design called for a stack to hold multiple PC values, allowing for jump subroutines to be nested. Since this stack was never constructed, the jump routines are not capable of being nested. Instead, only one jump routine can be successfully executed, latching the previous address in the register module. This presents an issue for the design, as iterative instructions cannot be nested.

The hardware test does not utilize the onboard 50Mhz clock. This design choice was made to provide an iterative means of demonstrating the results of the instructions. Viewing a signal output change in a matter of nanoseconds is physically impossible given the debug unit implementation. However, one way in which this could have been improved is by writing the results onto a memory module. The button input could then be used to increment the address location of this RAM module, showing the output data onto the seven-segment display. However, the ideal implantation would have been to use both a keyboard and VGA monitor to enter commands once at a time and view their outputs singularly. Unfortunate, this was not done to insufficient time and its complex implementation.

In hindsight, the CISC-24 machine operates continuously for a given program of 24-bit instructions as defined by the ISA. This program is imported within the contents of the 8x24 RAM module. The datapath unit consists of registers, multiplexors, buffers, an ALU, and a decoder that take the contents found within the RAM module and provide a logical path for data to be transported and stored for user interaction. The control unit is the workforce that dictates how and when this data path is directed based on the given instruction fetched from memory. The control unit must determine the operation and addressing mode of the given instruction to provide inputs to the datapath. The debug unit provides a means of actuation to this process on the Nexys 2 board. The designed component used a button to cycle through a written program and displaying the outputs on the board’s segment display, Pmod connectors, and LEDs.

# Conclusion

In conclusion to this design project, the fundamentals of computer architecture were examined and implemented using VHDL. By tracing the dataflow of the given instruction set, a complex instruction set machine was built to perform a majority of the required instructions. A state machine control unit was developed to dictate this datapath for each of the implemented instructions. Simulating a clock cycle through a button press helped determine the mean speed of the design by equating a single press to one clock pulse in a 50Mhz crystal. Addressing modes, branching, jump, and reset instructions were added in this experiment by modifying the previous design to accommodate extra control signals. Overall, the process of designing and testing each instruction was an incredibly laborious but rewarding process. Following set procedures for both simulation and hardware helped determine the functionality of this design. The References offers sources that were utilized in creating the datapath and RAM module for this design [1,2,3]. All other additions were done based on the foundation laid out by these sources.

# Reflection

The project was highly comprehensive and required much research and effort to comprehend and implement. However, much more could have been done to improve on the final design. To start, the missing instructions could have been implemented had we utilized our time more efficiently. We observed that a large amount of our work was dedicated to memory management instructions. Spending so much time on this aspect alone proved costly.

Looking back at the design, we suspect that one of the instructions was misunderstood. It was under the assumption that register indirect auto increment worked by increasing the result of the indirect value after the operation was performed. Further research suggests that the proper implementation was to increment the address itself by one, which would thus change the value completely rather than just adding by one after the instruction was completed. Regardless, the final design operates under this last assumption. If this is considered to be inoperable, than the completion percentage dips from 84% to 74% accounting for all instructions which utilize an Addressing Mode.

In retrospect to the course, we believe that a team of two students is sufficient for completing this assignment. However, we feel that the division of project labs was not sufficient. We feel that there should have been three practice labs along with three project labs. The last two practice labs could have been combined to allow more lab time for the project. The project lab should have designated a lab period specifically for memory operations. We feel that this aspect of the project was the most comprehensive and time consuming task. Additionally, we fell that the debug unit should have been covered more thoroughly. The last practice lab did a poor job in demonstrating how to utilize a VGA/Keyboard debug unit. The provided code was purposefully modified with a flaw that was never discussed or explained in class or in lab. Had this issue been addressed, we believe the hardware demonstration would have been much more comprehensive. The task of building a CISC machine from the ground up is incredibly extensive and impacts the time required to create an assembler to process and actuate this machine. Overall, we believe that the goal of designing both an assembler and CISC architecture from the ground up is too much to expect given our workload and time throughout the semester. Therefore, future efforts should be made to simplify one of these designs in order for both to be incorporated successfully and efficiently.

# References

[1] S. Roy, *Practical VHDL Samples*, 1st ed. Glasgow, Scotland: University of Glasgow, p. 8. [Accessed: 05- Apr- 2017].

[2] C. Riley, "Designing a CPU in VHDL", *Domipheus Labs*, 2015. [Accessed: 05- Apr- 2017].

[3] D. Noyes, "ECE368 Advanced Digital Design Lab", *GitHub*, 2015. [Online]. Available: https://github.com/Reiuiji/ECE368-Lab. [Accessed: 05- Apr- 2017].

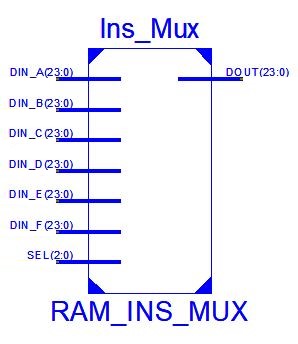
# Datasheet

## **Random Access Memory (RAM)** 8x24 Read and Write Unit

RAM is a behavioral module, which provide data storage for the CPU to run and execute instructions that are stored within.

For the purpose of this project, the designed RAM is a 8x24 block memory that used two signal buses for separate addresses.

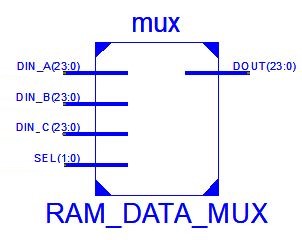
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| ADDR\_IN | 8 | IN | Signal indicates register address |
| DATA\_IN | 24 | IN | Data signal |
| CLOCK | 1 | IN | System Clock |
| READ\_WRITE | 1 | IN | Enable bit for Read and Write |
| DATA\_OUT | 24 | OUT | Result |

Along with RAM module, there are 6-to-1 MUX and 3-to-1 MUX to select data signal for RAM

* RAM\_INS\_MUX

This is a behavioral component which main purpose is to indicate location to read for instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN\_A | 24 | IN | Data signal from Program Counter Unit |
| DIN\_B | 24 | IN | Data signal from RA of GPR Unit |
| DIN\_C | 24 | IN | Data signal from IMM\_OOP of Decode Unit |
| DIN\_D | 24 | IN | Data signal from IMM\_MEM of Decode Unit |
| DIN\_E | 24 | IN | Data signal from IMM\_TOP of Decode Unit |
| DIN\_F | 24 | IN | Data signal from RB of GPR Unit |
| SEL | 3 | IN | Select signal to select location to read |
| DOUT | 24 | OUT | Result |



* RAM\_DATA\_MUX

This is a behavioral component which responsible to select contents to be written into RAM.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal** | **Bit Width** | | **Direction** | **Description** |
| DIN\_A | 24 | IN | | Data signal from RA of GPR Unit |
| DIN\_B | 24 | IN | | Data signal from IMM\_OOP of Decode Unit |
| DIN\_C | 24 | IN | | Data signal from DATA\_OUT of RAM |
| SEL | 2 | IN | | Select signal to select contents to write into RAM |
| DOUT | 24 | OUT | | Result |

## **PROGRAM COUNTER (PC) UNIT**

Program Counter is a behavioral module that identifies the procedure for obtaining the next instruction in memory.

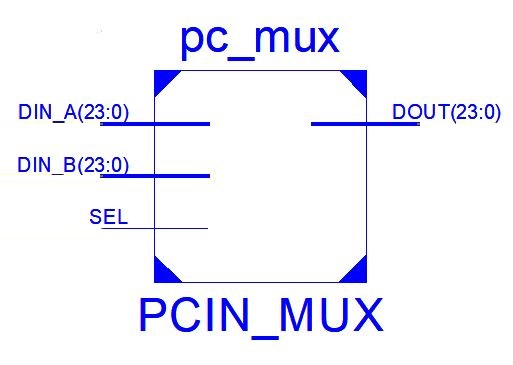
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal** | **Bit Width** | | **Direction** | **Description** |
| OFST | | 24 | IN | Offset signal from IMM\_BRN of DECODE Unit for Branch instruction |
| OP | | 2 | IN | Opcode signal from CONTROLLER Unit |
| PCIN | | 24 | IN | Data from PCIN\_MUX Unit |
| CLK | | 1 | IN | System Clock |
| EN | | 1 | IN | Enable bit from CONTROLLER Unit |
| RST | | 1 | IN | Reset bit to clear Program Counter from AC unit |
| PCOUT | | 24 | OUT | Result out |

|  |  |
| --- | --- |
| **Operation Code** | **PC Output** |
| 00 | PC (Does not change) |
| 01 | PC+1 (Next) |
| 10 | PCIN (Jump) |
| 11 | PC + OFST (Branch) |

## **Program Counter Latch**

The PC Latch is a behavioral register module that store data of the instruction before feeding the data to Program Counter.

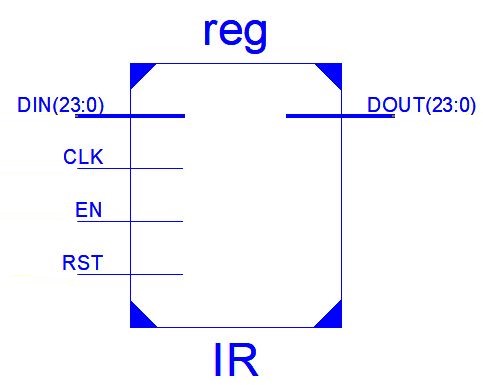
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN | 24 | IN | Data signal from PC\_OUT of Program Counter |
| CLK | 1 | IN | Clock System |
| EN | 1 | IN | Enable bit |
| RST | 1 | IN | Reset bit |
| DOUT | 24 | OUT | Data signal out that is sent to PCIN\_MUX |

****

## **Program Counter MUX (PC\_MUX)**

The PC\_MUX is a 3-to-1 MUX, behavioral module that selects which data input from the DECODE Unit and PC\_LATCH register to be fed to Program Counter Unit.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN\_A | 24 | IN | Data signal from IMM of DECODE Unit |
| DIN\_B | 24 | IN | Data signal from DOUT of PC\_LATCH |
| SEL | 1 | IN | Select bit |
| DOUT | 24 | OUT | Data signal to Program Counter Unit |



## **Instruction Register (IR)**

IR is a behavioral module that its main purpose is to hold the instruction that is currently being executed or decoded.

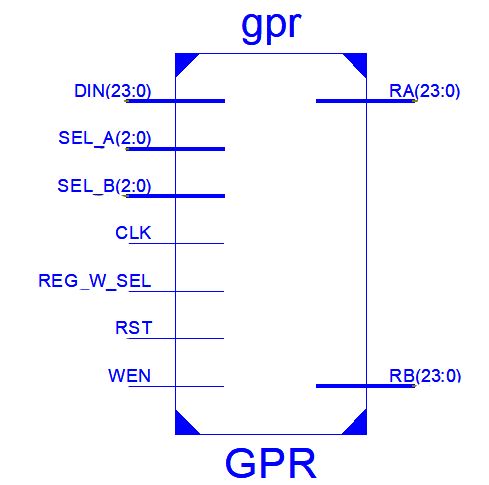
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN | 24 | IN | Data signal from DATA\_OUT of RAM Unit |
| CLK | 1 | IN | System Clock |
| EN | 1 | IN | Enable bit |
| RST | 1 | IN | Reset bit to clear the Instruction Register |
| DOUT | 24 | OUT | Result |

## **DECODE MODULE**

**The Decoder is a behavioral module, which takes in the fetched instruction and outputs the decoded segment that will be utilized within the datapath. The module simply outputs segments of the input instruction that map out to either an address, mask, or immediate value.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| **INS** | **24** | **IN** | **24 bit instruction data** |
| **CLK** | **1** | **IN** | **System clock** |
| **EN** | **1** | **IN** | **Enable bit for decoder** |
| **ADDR\_A** | **3** | **OUT** | **Address of Register A** |
| **ADDR\_B** | **3** | **OUT** | **Address of Register B** |
| **AM\_A** | **2** | **OUT** | **Addressing mode signal for source A** |
| **AM\_B** | **2** | **OUT** | **Addressing mode signal for source B** |
| **IMM** | **24** | **OUT** | **Immediate signal for Jump instructions** |
| **IMM\_BRN** | **24** | **OUT** | **Immediate signal for branching** |
| **IMM\_MEM** | **24** | **OUT** | **Immediate signal for memory** |
| **IMM\_OOP** | **24** | **OUT** | **Immediate signal for OOP instructions** |
| **IMM\_TOP** | **24** | **OUT** | **Immediate signal for TOP instruction** |
| **MSK** | **4** | **OUT** | **Mask bits (N,V,Z,C)** |
| **OP** | **5** | **OUT** | **OP code for current instruction** |

## **GENERAL PURPOSE REGISTER UNIT**

**General Purpose Register is a behavioral module. The component acts as a register bank, holding temporary data and instructions whilst calculations and algorithms are being run.**

**GPR unit holds an array of eight, 24 bit signals that account for each of the eight register locations (from R0 to R7). It reads from the two input source registers and will write to either the first or second source as determined by the register write enable.**

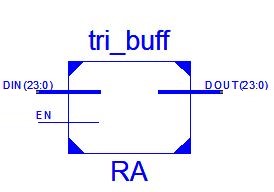
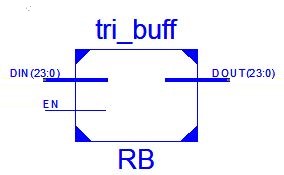
**Contents of specified source addresses are sent to RA or RB for CPU to utilize based on given instruction.**

**Datasheet**

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit width** | **Direction** | **Description** |
| **CLK** | **1** | **IN** | **System clock** |
| **DIN** | **24** | **IN** | **Data from GPR\_IN\_MUX unit** |
| **SEL\_A** | **3** | **IN** | **Select register A, determined by the DECODER unit** |
| **SEL\_B** | **3** | **IN** | **Select register B, determined by the DECODER unit** |
| **REG\_W\_SEL** | **1** | **IN** | **Write signal from the CONTROLLER Unit** |
| **RST** | **1** | **IN** | **Reset bit to clear the register bank** |
| **WEN** | **1** | **IN** | **Enable bit for write command, determined by the CONTROLLER Unit** |
| **RA** | **24** | **OUT** | **24 bit data to Register A** |
| **RB** | **24** | **OUT** | **24 bit data to Register B** |

## **Tri- state buffer**

The tri-state buffer is a behavioral module that control the data input for ALU unit. For the purpose of the project, there are two tri- state buffers for the ALU, named RA and RB. When enable is set high, the input passes through to the output. Otherwise, the output is set to 0.

. 

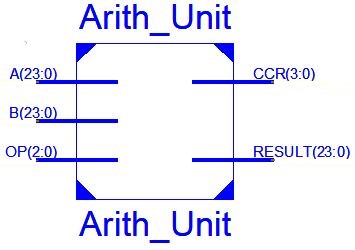
|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN | 24 | IN | Data signal from DIN\_B (for RA), DIN\_F(for RB) from RAM\_INS\_MUX |
| EN | 1 | IN | Enable bit |
| DOUT | 24 | OUT | Data signal to DIN\_A of ALU\_IN\_MUX\_A (for RA) and ALU\_IN\_MUX\_B (for RB) |

## **ARITHMETIC-LOGIC UNIT (ALU)**

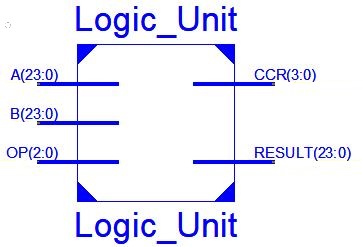
ALU is a structural module that is responsible for executing the OP code, including arithmetic and logical instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| OPCODE | 4 | IN | Instruction OPCODE from the CONTROLLER unit |
| RA | 24 | IN | Data bit from RAM |
| RB | 24 | IN | Data bit from RAM |
| CLK | 1 | IN | System Clock |
| ALU\_OUT | 24 | OUT | ALU result input |
| CCR | 4 | OUT | Condition Code Register bit (Z,C,N,V) |
| LDST\_OUT | 24 | OUT | Status bit |

The ALU is comprised of five different components:

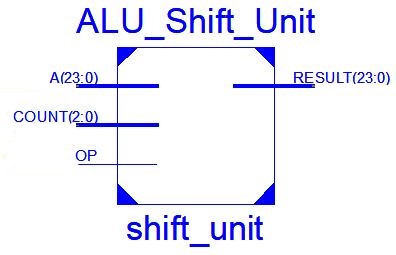
* ALU Arithmetic unit- executes instructions such as addition, subtraction, multiplication, division, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| A | 24 | IN | Data bit from RA |
| B | 24 | IN | Data bit from RB |
| OP | 3 | IN | Operand code from OPCODE |
| CCR | 4 | OUT | Condition code register (Z, C, N, V) |
| RESULT | 24 | OUT | RESULT of Arithmetic instruction |



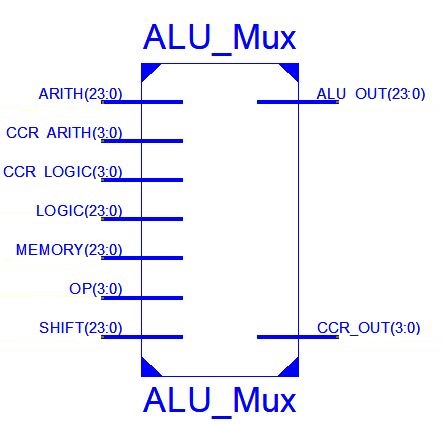
* ALU Logic unit- executes instructions such as AND, OR, CMP, etc

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| A | 24 | IN | Data bit from RA |
| B | 24 | IN | Data bit from RB |
| OP | 3 | IN | Operand code from OPCODE |
| CCR | 4 | OUT | Condition code register (Z, C, N, V) |
| RESULT | 24 | OUT | RESULT of Logic instruction |



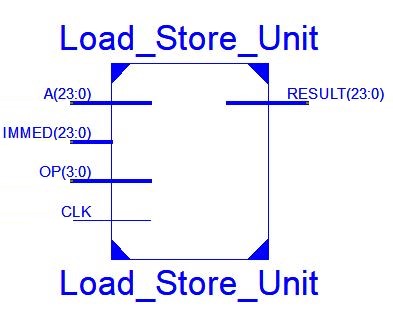
* ALU Shift unit- executes Shift Left, Shift Right instructions

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| A | 24 | IN | Data bit from RA |
| COUNT | 3 | IN | Required bit for Shifting operand |
| OP | 3 | IN | Operand code from OPCODE |
| RESULT | 24 | OUT | RESULT of Shift instruction |

* ALU MUX unit dictates the ALU module to be used based on the OP code. The following table outlines the utilized op codes for this component.

|  |  |
| --- | --- |
| **OP** | **Instruction** |
| 0000 | ADD |
| 0001 | SUB |
| 0010 | AND |
| 0100 | NEG |
| 0101 | XOR |
| 1000 | SL |
| 1001 | SR |
| 1010 | ADDI |
| 1101 | OR |
| 1111 | CLR |

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| ARITH | 24 | IN | Arithmetic result from Arithmetic Unit |
| CCR ARITH | 4 | IN | CCR bit status (Z, C, N, V) of Arithmetic Unit |
| CCR Logic | 4 | IN | CCR bit status (Z, C, N, V) of Logic Unit |
| LOGIC | 24 | IN | Logic result from Logic Unit |
| MEMORY | 24 | IN | Result from the the LOAD STORE Unit |
| OP | 4 | IN | Operand code from OPCODE |
| SHIFT | 24 | IN | Shift result from Shift Unit |
| ALU\_OUT | 24 | OUT | Final result of the ALU |
| CCR\_OUT | 4 | OUT | Final CCR bit status (Z,C,N,V) of the ALU |

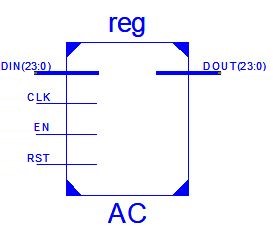


* ALU Load Store unit- work as an internal memory for ALU Arithmetic, Logic and Shift Unit to execute.

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Bit Width | Direction | Description |
| A | 24 | IN | Data from RA of ALU |
| IMMED | 24 | IN | Data from RB of ALU |
| OP | 4 | IN | Operand code from OPCODE |
| CLK | 1 | IN | System Clock |
| RESULT | 24 | OUT | Selected result of Arith, Logic and Shift Unit |

## **ACCUMULATOR (AC)**

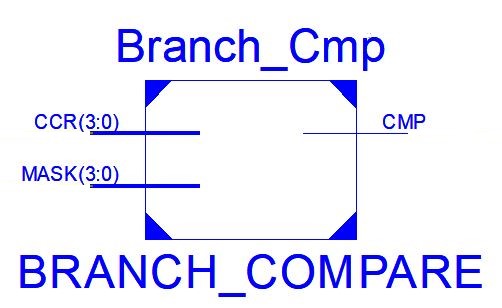
Accumulator is a behavioral module which is instantiated as a basic 24-bit register. When enabled, the component will save the current input data and send it to the output port. This holds the result of the ALU and is also connected for both memory and register write back.



|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN | 24 | IN | Data signal from ALU\_OUT result from ALU |
| CLK | 1 | IN | System Clock |
| EN | 1 | IN | Enable bit |
| RST | 1 | IN | Reset Bit |
| DOUT | 24 | OUT | Data signal as result to GPR\_IN\_MUX and DEBUG\_MUX |

## **Branch Compare**

Branch Compare is a behavioral module that predicts the next fetch address to keep the pipeline full with correct sequence of instructions, based on comparing the CCR and MASK data signals for equality.

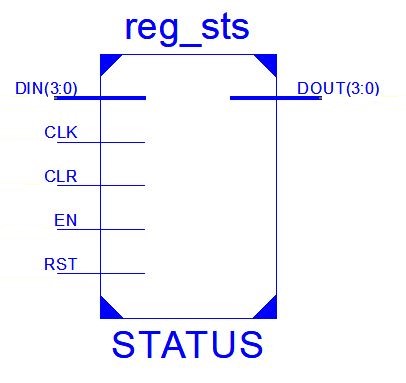


|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| CCR | 4 | IN | Data signal of CCR from Status Register |
| MASK | 4 | IN | Mask data signal from Decode Unit |
| CMP | 1 | OUT | Prediction result after comparing CCR and MASK data |

## **Status Register**

The Status register is a behavioral module that contains flags which reflect the state of the ALU after the instructions are executed. This register stores the status bits to allow the hardware debug unit to display.

For the purpose of this project, there are four flags for this design

* Zero flag (Z)- indicate the result of the instruction (arithmetic or logical operation) is zero.
* Carry flag (C)- indicates that the result of an unsigned operation is overflow.
* Negative flag (N)- indicates that the result of the instruction is negative.
* Signed Overflow (V)- indicates that the result of a signed operation is overflow.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN | 4 | IN | Data signal contains flag values from the ALU |
| CLK | 1 | IN | System clock |
| CLR | 1 | IN | Clear bit |
| EN | 1 | IN | Enable bit to set Status |
| RST | 1 | IN | Reset |
| DOUT | 4 | OUT | Data signal that indicate state of the processor |

## **DEBUG MUX**

The component is a 24 bit 3-to-1 MUX, behavioral module which receive data from the Accumulator Unit, GPR\_IN\_MUX and RAM\_DATA\_MUX to transfer contents onto the hardware display upon completion of the given instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| DIN\_A | 24 | IN | Data signal from DOUT of AC |
| DIN\_B | 24 | IN | Data signal from DOUT of GPR\_IN\_MUX |
| DIN\_C | 24 | IN | Data signal from DOUT of RAM\_DATA\_MUX |
| SEL | 2 | IN | Select signal from OUT\_SEL of CONTROLLER Unit |
| EN | 1 | IN | Enable bit |
| RST | 1 | IN | Reset bit |
| DOUT | 24 | OUT | Data out to CPU\_OUT as result for CPU |



**ALU\_Store**

GPR input is multiplexed to select the contents of the accumulator. Addressing mode is checked to see if more than one register is not directly addressed. The status bit latch is enabled to display the contents of the CCR on board.

**Reg\_Store**

GPR input is multiplexed and the write enable bit is set to write the computed contents within the register bank. The register select bit is also set to insure the correct register is written to. The output multiplexor selects the register output as the valid output signal to be shown on board.

**Mem\_Store/Write**

RAM Data input is multiplexed to determine the data source to write back into RAM.

MVMI incurs an additional state called Mem\_write that allows for an additional cycle to select a RAM address for Memory to Memory movement. The output multiplexor selects either the GPR input or RAM input as the valid output signal to be shown on board.

**Stop**

Disables all write back signals enabled from previous states unless the instruction requires addition clock cycles to read and write (Ex. MVMI, Indirect Addressing mode for both SrcA and SrcB. The Output multiplexor is enabled to display the result of the finished instruction. The PC is assigned an OP code based on the instruction. It is either incremented normally, branched by a computed offset, or replaced with the input address signal from the PC module to jump.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Bit Width** | **Direction** | **Description** |
| CLK | 1 | In | System Clock |
| RST | 1 | In | Reset bit to force the state machine back to the first state (start) when high |
| OP | 5 | In | Op code for current instruction. Always the first 5 bits of the instruction |
| AM\_A | 2 | In | Decoded Adressing mode for SrcA |
| AM\_B | 2 | In | Decoded Adressing mode for SrcB |
| MASK\_CMP | 1 | In | Mask compare bit that signifies MASK = NVZC when high |
| PC\_OP | 2 | Out | Program Count op code that controls how the next instruction is fetched |
| PC\_EN | 1 | Out | Enable bit for the Program Count |
| IR\_EN | 1 | Out | Enable bit for the Instruction Register |
| DECODE\_EN | 1 | Out | Enable bit for the Decoder |
| GPR\_WE | 1 | Out | Write Enable bit for the General Purpose Register |
| GPR\_W\_SEL | 1 | Out | Write Select bit for the General Purpose Register. SrcA = 0, SrcB = 1 |
| GPR\_MUX\_SEL | 3 | Out | Select signal for the GPR Input Multiplexor |
| RA\_EN | 1 | Out | Enable bit for the RA buffer |
| RB\_EN | 1 | Out | Enable bit for the RB buffer |
| ALU\_SEL\_A | 3 | Out | Select signal for the ALU Input Multiplexor for RA |
| ALU\_SEL\_B | 3 | Out | Select signal for the ALU Input Multiplexor for RB |
| ALU\_A\_EN | 1 | Out | Enable bit for the RAM Data Input Latch for RA |
| ALU\_B\_EN | 1 | Out | Enable bit for the RAM Data Input Latch for RB |
| ALU\_OP | 4 | Out | OP code for the arithmetic operation determined by instruction |
| AC\_EN | 1 | Out | Enable bit for the Accumulator |
| MEM\_RWE | 1 | Out | Read/Write bit for the RAM module. Read = 0, Write = 1 |
| DATA\_MUX\_SEL | 2 | Out | Select signal for the RAM Data Input Mulitplexor |
| INS\_MUX\_SEL | 3 | Out | Select signal for the RAM Instruction Input Multiplexor |
| OUT\_SEL | 2 | Out | Select signal for the Debug Output Multiplexor |
| OUT\_EN | 1 | Out | Enable bit for the Debug Output Multiplexor |
| CLR | 1 | Out | Software Reset assigned within the Control Unit |
| PCL\_EN | 1 | Out | Enable bit for the Program Count Latch |
| PCIN\_EN | 1 | Out | Enable bit for the Program Count Address Input Multiplexor |
| STS\_EN | 1 | Out | Enable bit for the Status Bit Latch |
| STS\_CLR | 1 | Out | Reset bit for the Status Bit Latch |

|  |  |  |
| --- | --- | --- |
| **State** | **Input Signals** | **Output**  **State(s)** |
| Start | (ALL) | Fetch |
| Fetch | RAM  IR | Decode |
| Decode | PC\_E  DCD | Op\_sel |
| OP\_sel | OP | Oop\_access  Top\_access  Reg\_store  Mem\_store  Stop  OP\_SEL |
| Oop\_access | OP  RA/B  AM\_A  ALU\_A/B  RAM\_INS\_SEL  RAM, DCD | Oop\_exec |
| Top\_access | RA/B  AM\_A  ALU\_A  RAM  DCD | Alu\_store |
| Oop\_exec | OP  AC | Alu\_store |
| Top\_exec | ALU\_B  AOP  AC  RAM\_INS\_SEL  RAM  DCD,  ALU\_A\_E  AM\_B | Alu\_store |
| ALU\_store | GPR\_MUX  GPR\_WRITE  AM\_A/B  GPR\_SEL  STS\_E | Stop |
| Reg\_store | GPR\_MUX  GPR\_WRITE  OP  GPR\_SEL | Stop |
| Mem\_store | RAM\_INS\_SEL  GPR\_SEL  DCD  OP  RAM\_DATA\_MUX  OUT\_MUX  GPR\_MUX  GPR\_WRITE  RAM  OUT\_MUX | Stop  Mem\_write |
| Mem\_write | RAM\_INS\_SEL  RAM  OUT\_MUX  OUT\_E | Stop |
| Stop | RAM  GPR\_WRITE  RAM\_INS\_SEL  RAM\_DATA\_SEL  PC\_E  OP  PC\_O  PCL\_E  MASK\_CMP  AC  AM\_A/B  OUT\_E | Start |

