MIPS PROCESSOR IN C

for

CSCI 2500

Final Draft

Prepared by : 1. Ethan Cruz
2. Matthew Merritt
3. Samyuth Sagi

Submitted to : Dr. Kuzmin
Professor

December 11, 2021

Contents

1 Introduction

1.1 Purpose

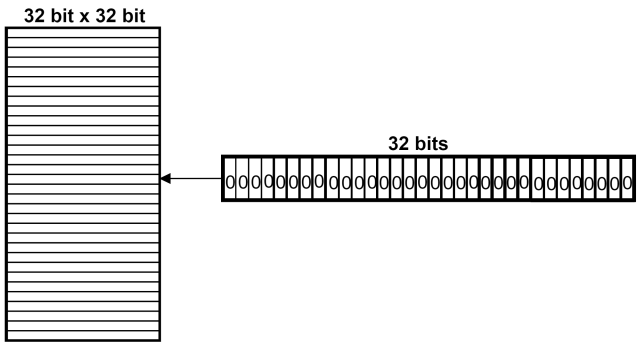
As noted by the Project instruction PDF (can be found in Additional Resources chapter), the goal of the project is to take the concepts and applications of those concepts learned over the course to their "logical conclusion". In doing so the final project is to create a "full gate-level circuit representing the datapath for a reduced, but still Turing complete, MIPS ISA" in C programming language.

1.2 Project Scope

Due to the large nature of a project of this scale, there have been some modifications and assumptions that can be made. The full definition of the project can be found in more detail on the Project instruction PDF (can be found in Additional Resources chapter), but for some highlights:

- Memory is to be addressed on the word (not byte) boundary.
- 3 separate 32 $\times$ 32-bit 2D arrays representing storage for the Register File, Instruction Memory, and Data Memory will be used instead of a d-flip-flop system. The diagram below is representative of a empty 32 by 32 bit string in the 32 by 32 bit storage system.
- Input format is modified for easier parsing

Storage System Representation:

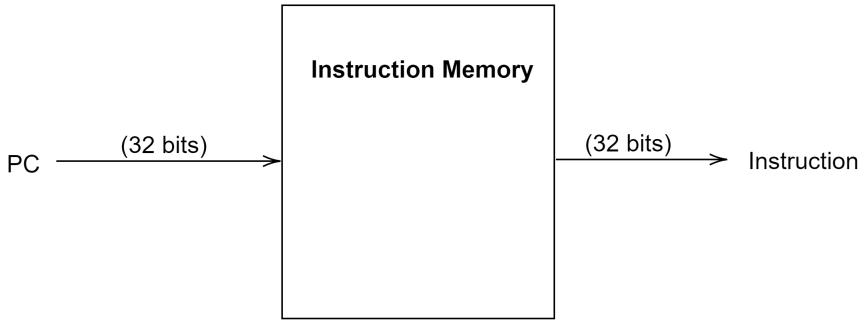


(Fig 1.1)

2 Circuit Design and Implementation

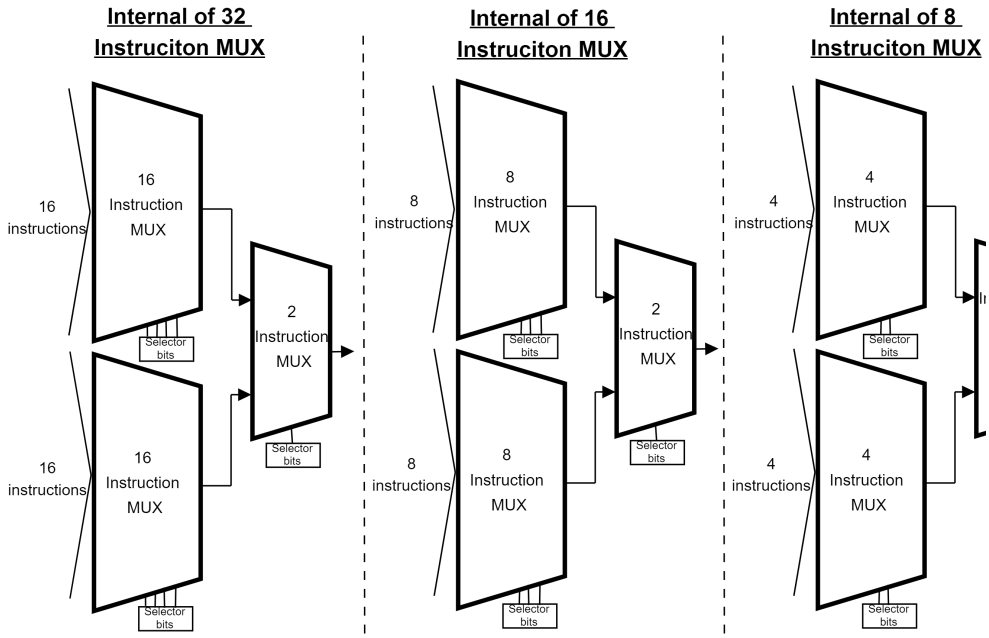
2.1 Instruction Memory

The Instruction Memory circuit uses a single large 32 bit input MUX (multiplexer) whose source is 32 bits, and output is also 32 bits (noted in Fig 2.1.1). The selector bits are the PC (Program Counter), as PC is stored as a binary representation of an integer that corresponds to what instruction to output. The input is the MEM_instruction which is the memory storage saved for each instruction of the program. The output is a 32 bit buffer which was selected from the Instruction's 32x32 memory.



(Fig 2.1.1)

The Reasoning for using a single 32 bit MUX is due to the clarity that is afforded by using similar gates compared to the approach that used a 5 bit decoder (which was also programmed and functional), but after consideration the team decided to go with the 32 MUX approach as it also took advantage of the 32 by 32 bit instruction storage. The abstract representation of the internal circuit that allows for the correct 32 bit instruction to be selected and set as output can be found in diagram below (fig 2.1.2).



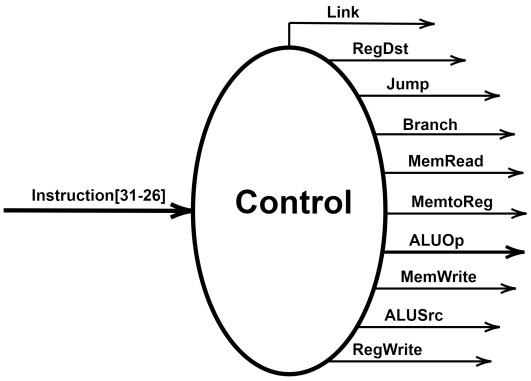
(Right side cutoff is intended to show further encapsulation.)

(Fig 2.1.2)

The Instruction Memory implementation design allowed for repeated usage of components e.g. 16, 8, 4 bit Instruction MUX's. As shown in the diagram there is less selector bits for each MUX at each level, and less Instructions to select from. With each lower level narrowing down which instruction to output. This process results in the instruction that corresponds to the PC to be written to the output (output_buffer in program).

2.2 Control

For the design of the control Unit, an additional control line was added to support the additional instructions Jump to Register (JR) and Jump and Link (JAL) (previous not supported on Figure 4.24 from class textbook - noted in Additional Resources). The additional control line, "Link", can be viewed as a part of the abstract diagram of the control unit below (Fig 2.2.0). It can be noted that each line's thickness is correlated to the amount of bits that are passed through.



(Fig 2.2.0)

The design of the control circuit uses several gates to decide what control lines should become active. The gates used for each control line output is based on an optimized sum-of-products form derived from a truth table. The truth table (Fig 2.2.1) represents what bits should be active from the current instruction given by Instruction Memory (Instruction[26-31]).

Instruction						
Name	Instruction[26-31]					
	Op1	Op2	Op3	Op4	Op5	Op6
sw	1	0	1	0	1	1
lw	1	0	0	0	1	1
addi	0	0	1	0	0	0
beq	0	0	0	1	0	0
jal	0	0	0	0	1	1
j	0	0	0	0	1	0
R-Type	0	0	0	0	0	0

(Fig 2.2.1)

In the following diagram (Fig 2.2.2) the impact of modifying the data path for the new instructions can be clearly seen. The JAL instruction requires in our data path an additional output control line *(Note: Footnote 1)*. To provide clarification for the implementation and design of the control unit, a table of the output for each control unit line is shown below.

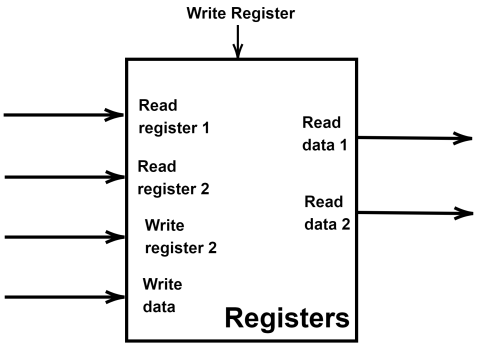
Instruction											
Name	Control Lines										
	RegDest	Jump	Branch	MemRead	MemWrite	MemtoReg	ALUsrc	ALUop1	ALUop2	RegWrite	Link
sw	0	0	0	0	1	0	1	0	0	0	0
lw	0	0	0	1	0	1	1	0	0	1	0
addi	0	0	0	0	0	0	0	1	1	1	0
beq	0	0	1	0	0	0	0	0	1	0	0
jal	0	1	0	0	0	0	0	X	X	0	1
j	0	1	0	0	0	0	0	X	X	0	0
R-Type	1	0	0	0	0	0	0	1	0	1	0

(Fig 2.2.2)

[1]

2.3 Register File

The Register File circuit is composed of two different combinational circuits. The first circuit is for reading and another for writing. These are expanded upon in the following subsections. The diagram below (2.3.0) is an abstract representation of the Register File.



(Fig 2.2.2)

2.3.1 Read Register

The Read Register circuit is composed of a similar design to the Instruction Memory design displayed in section 2.1. By using two 32 MUX, the 32 bits that correspond to data belonging to a register are selected and returned as the register data. Since MEM_Register is a 32 by 32 storage system that contains the values for each register, the "Read register" value is translated from a binary representation of a number to a selection of the register value (written to "Read data"). Because the Read register circuit assigns read data for two read register values, the 32 MUX must be repeated twice in order to properly set both. For more clarification on the implementation of the 32 MUX, refer to Fig 2.1.2 and the Additional Circuit Design Implementation chapter.

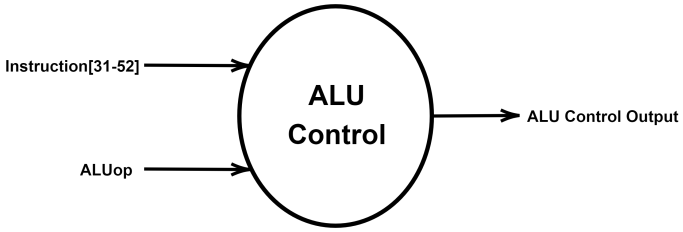
2.3.2 Write Register

The circuit design for Write Register's implementation uses a 5 to 32 decoder as a way to get the register value/address. The value at the address is only written over if RegWrite is true (done using an and gate and 2 multiplexer). Since there is conditional writing in a combinational circuit, we choose to go with the 5 to 32 decoder approach compared to the 32 MUX noted in above circuits.

[1] As per Rule 3 apart of the project rules document, we can "modify the function prototypes given in the template", permitting the additional Link output line

2.4 ALU Control

The ALU Control Circuit was designed similarly to the control circuit. It is created using a truth table of possible inputs and desired outputs and using SOP along with optimization through algebra simplifications to get the gates that circuit uses. For reference on the modified data path below is an abstract diagram of the ALU Control (Fig 2.4.0).



(Fig 2.4.0)

Since the input bit can come from multiple sources it is useful to mention that funct1-funct6 come from the function field (For more into refer to MIPS Reference in the Resource Chapter) also known as Instruction[0-5] and ALUop1 and ALUop2 are from the Control unit. The truth table for inputs and associated operation is below (Fig 2.4.1). There is an additional output from the ALU Control that allows for the operation of the JR instruction for the modified data path (more details can be found in the modified data path section 4.2).

Instruction								
Name	Input							
	Funct1	Funct2	Funct3	Funct4	Funct5	Funct6	ALUop1	ALUop2
and	1	0	0	1	0	0	1	0
or	1	0	0	1	0	1	1	0
add	1	0	0	0	0	0	1	0
sub	1	0	0	0	1	0	1	0
slt	1	0	1	0	1	0	1	0
jr	0	0	1	0	0	0	1	0
addi	X	X	X	X	X	X	1	1
lw	X	X	X	X	X	X	0	0
sw	X	X	X	X	X	X	0	0
beq	X	X	X	X	X	X	0	1

(Fig 2.4.1)

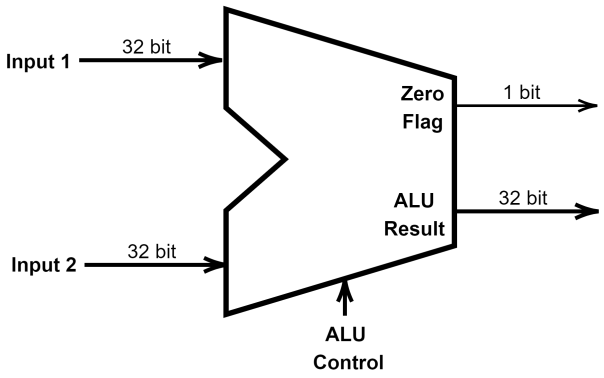
By using SOP and optimization we were able to get a clear and concise way of determining the gates for each Operation. This method also provided a straightforward way for passing the desired register value (at return address aka register address number 31) for the JR instruction. This feature to pass the return address was incredibly useful for our updated/modified data path, therefore becoming a part of our design and implementation for the ALU control. The truth table for the operation to perform on the ALU is displayed in the truth table below (Fig 2.4.2)

Instruction								
Name	Input							
	Funct1	Funct2	Funct3	Funct4	Funct5	Funct6	ALUop1	ALUop2
and	1	0	0	1	0	0	1	0
or	1	0	0	1	0	1	1	0
add	1	0	0	0	0	0	1	0
sub	1	0	0	0	1	0	1	0
slt	1	0	1	0	1	0	1	0
jr	0	0	1	0	0	0	1	0
addi	X	X	X	X	X	X	1	1
lw	X	X	X	X	X	X	0	0
sw	X	X	X	X	X	X	0	0
beq	X	X	X	X	X	X	0	1

(Fig 2.4.2)

2.5 ALU

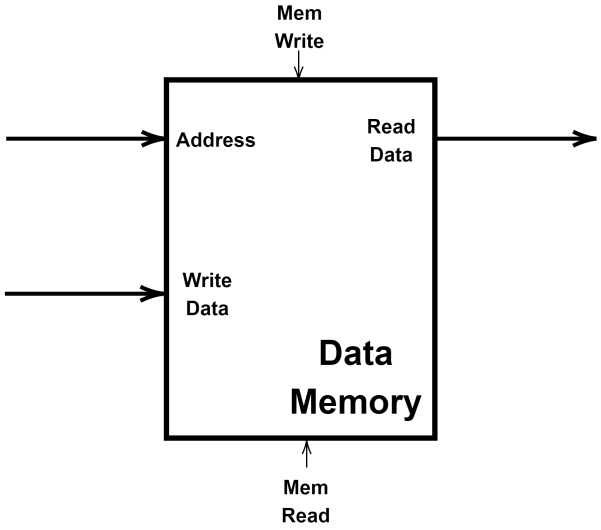
The design for the ALU is a simplistic ripple carry design that has been covered extensively in lecture. The reason our group choose to do ripple carry instead of carry look ahead was because although a carry look ahead would be faster, when it came to the simulation, the increase in speed would be fruitless. Because each line in C (the language the simulation is built in) is not run concurrently (for a sub circuit) in our program, the function for the previous adder would have to complete running before the return value of propagate or generate. For the implementation choice of identifying if the Zero Flag should be active, a 32 XOR gate was created. The 32 Zero gate would only become active if all 32 bits (of the result) would be 0s. To also note for implementation; two additional bits are within the ALU for passing the Most significant bit and set less than for negative results. An Abstract Diagram of the ALU is below for identification in the updated data path.



(Fig 2.5.1)

2.6 Data Memory

The design of implementation for Data Memory was assembled similarly to the design/implementation of Write Register. The input for the circuit is an address and write data (both 32 bits). An abstract diagram can be seen below in (Fig 2.6.1) that can also be found on the final data path.

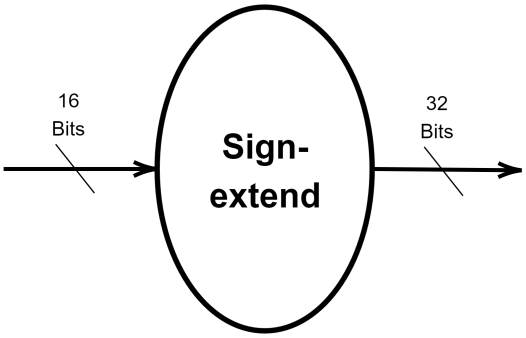


(Fig 2.6.1)

The Data Memory uses a 32 by 32 bit structure as noted in the Project Scope chapter (Fig 1.1). Using implementation methods mentioned prior of a 5 to 32 decoder and a 32 MUX, the design functionally was realised. The reason for usage of the 5 to 32 decoder and fixed for loop method with the "and" gate, was due to the conditional writing to the memory database. The "and" gate would only write data if both conditions were 1 (true). Using an "and" gate with MemWrite from the control unit was used. The implementation of a 32 MUX (similar to the implementation of previous 32 MUX) is for properly setting a read data output line.

2.7 Sign Extend

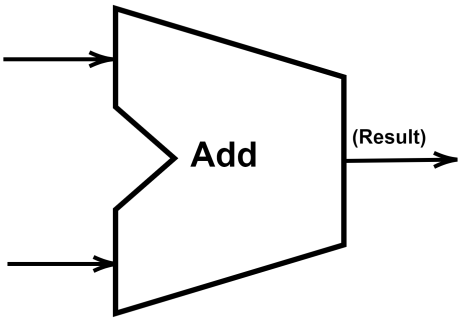
The Sign Extend circuit design is based on accounting for 2s-complement. The circuit's input is 16 bits from (Instruction[15-0]), and output is a single 32 bit bus. The circuit's purpose is as it's title, "extend the sign". To do this our implementation used a fixed for loop to copy all input bits from the input into the 32 bit buffer, and for the remaining 16 bits, the most significant bit is repeated.



(Fig 2.7.1)

2.8 Adder

The Adder circuit design is a very commonly used/implemented circuit in the class. The design was to add two 32 bit values and give their sum using smaller 1 bit adders. For our implementation, we used a ripple carry system that utilized a fixed for loop in order to set the output line bits to their proper value. An abstract diagram of the circuit is below (Fig 2.8.1), and can be found on the previous and modified data path.

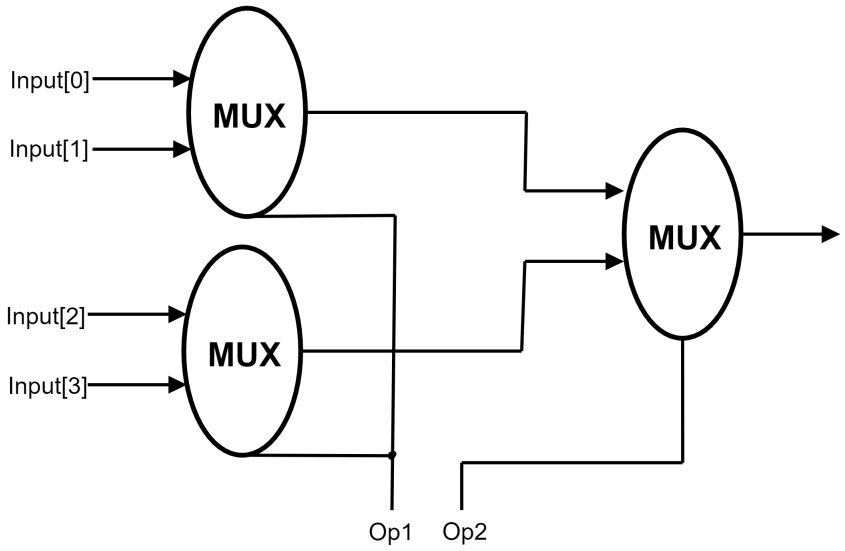


(Fig 2.8.1)

3 Additional Circuit Design and Implementation

3.1 Multiplexer

The Multiplexer Circuit is a vital component for our design and implementation of my teams modified data path. Within the data path multiplexers are expanded to use 4 input bits instead of 2 bits of input, the diagram below shows the internals of how this is implemented (Fig 3.1.1) AKA 4 bit input MUX.



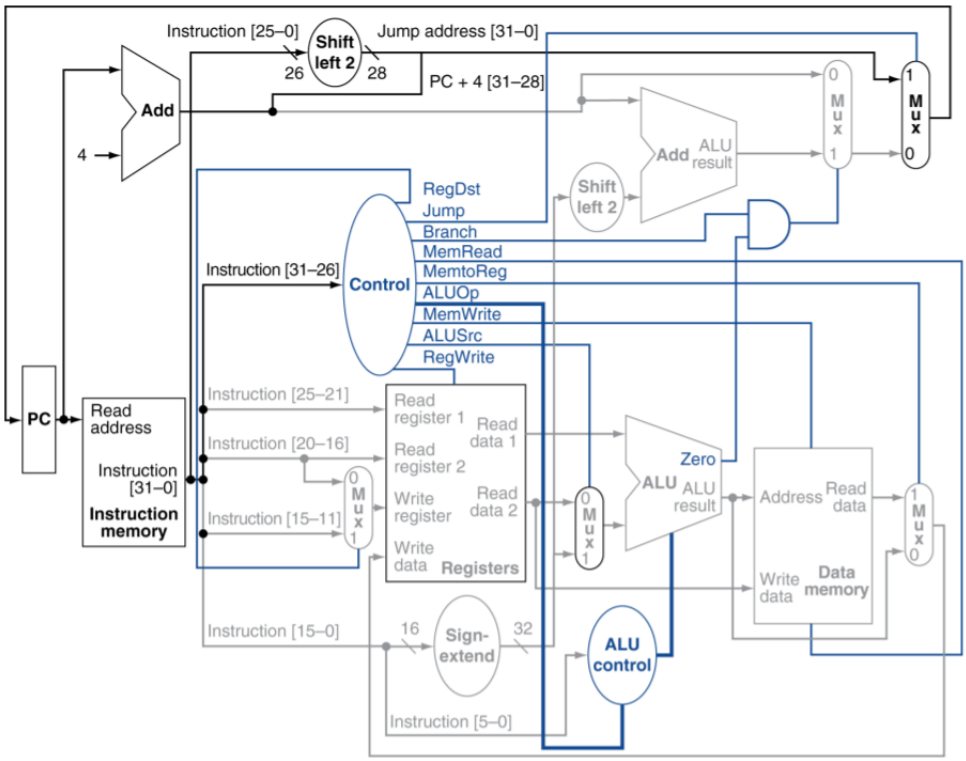
(Fig 3.1.1)

Within circuits described above, many use a 32 MUX that passes the corresponding value from the storage system through entire 32 bit strings to the output, this is first mentioned in the Instruction Memory section.

4 Modified Data Path

4.1 Previous Data Path

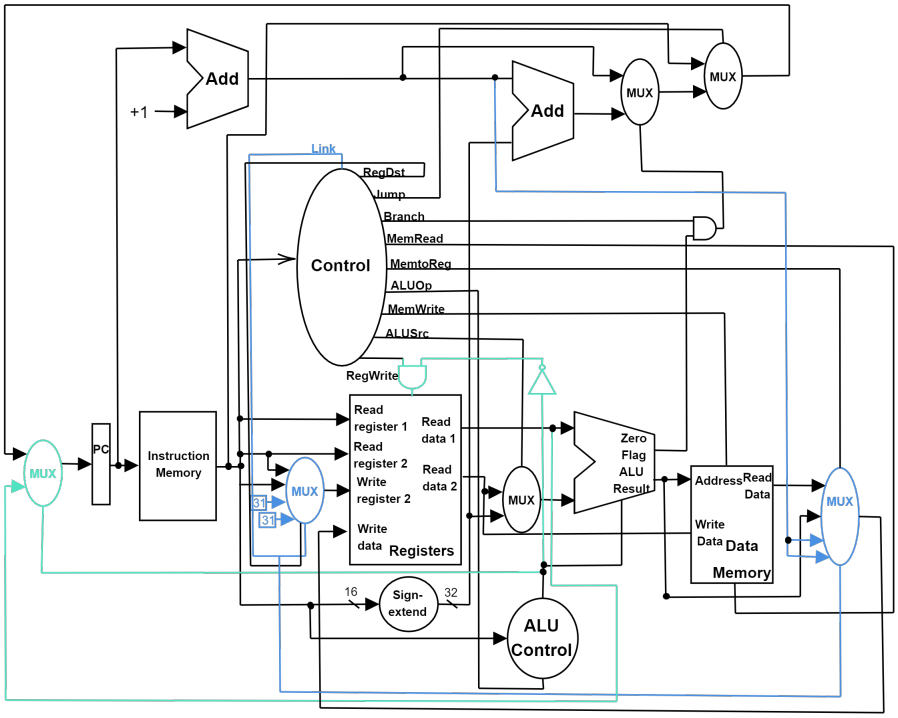
This is the original data path from the Course Textbook (Textbook cited in Additional Resources) referenced by the project outline ("Datapath With Jumps Added"). The original data path does not support JAL and JR instructions.



(Fig 4.1.1)

4.2 Modified Data Path For New Instruction

The diagram below (4.2.1) is the modified data path that accounts for JR and JAL instructions (modified Figure 4.24 from course textbook). The lines/components in light blue are additions/modifications that are for the JR instruction. The lines/components in darker blue are additions/modifications that are for the JAL instruction.



(Fig 4.2.1)

For JAL, coming out of control is an additional line called link that acts as the selector bit for a couple of 3 multiplexers (In reality these are 4 multiplexers with the last two inputs being from the same lines). The multiplexer choosing Write register is modified to take two additional inputs which are constant 31s (5 lines with a constant 1). This is to choose $ra to write to. In addition, reg write must be on for JAL. The multiplexer coming out of the data segment has an additional two inputs and selector (link). The additional two inputs come from the calculated next instruction. This is so you jump to the line after the JAL instruction in MIPS.

For JR there is a special additional control line coming out of ALU control. This serves two functions, the first is that it overrides register write (when it is on you cannot write to a register hence the "and" gate with a "not" from the control coming out of ALU control). The second is that it also acts as a selector bit for a new 2-multiplexer that feeds into the PC register. This multiplexer chooses between the original input and an input from Read Data 1 (so it can get the address from that register).

5 Resources

5.1 Additional Resources

- C Programming Language
  - Kernighan, B. W., & Ritchie, D. M. (2006). The C programming language.
- Class Textbook
  - David A.; Hennessy, John L.. Computer Organization and Design MIPS Edition (ISSN) (p. 859). Elsevier Science. Kindle Edition.
- Mathcha (Diagram Building Resource)
  - https://www.mathcha.io/
- MIPS Reference Sheet
  - https://inst.eecs.berkeley.edu/ cs61c/resources/MIPS_Green_Sheet.pdf
- Google Sheets (Truth Table Building Resource)
  - https://www.google.com/sheets/about/
- Project Instructions
  - https://submitty.cs.rpi.edu/courses/f21/csci2500/display_file?course_material_id=665

6 Contributions

6.1 Ethan Cruz

Created Instruction Memory, Read/Write Register, and Memory Data. This includes implementation of the 32 mux and use of 5 to 32 decoder. Also edited the final document for write-up.

6.2 Matthew Merritt

Created the parsing portion of the program; to translate any valid supported instruction to 32 bit binary representation. Also worked to create the data path with great assistance from Samyuth Sagi. Finally working on creating all diagrams present in this write up and the write up itself (along with help from group member for clarifications on their components).

6.3 Samyuth Sagi

Worked on creating all the basic components including the multiplexors, ALUs, and any gates or circuits not included by default. Created truth tables for Control and ALU Control and also implemented them. Contributed to the creation of the datapath by creating rough diagrams and helping in the coding portion. Also worked on debugging.