Final Project
12/9/2022
Professor Kuzmin
Taylor Mull, Shanelle Kan, Aiden Drover-Mattinen, Aj Pai

100dabloons Final Project

All Choices made:

We decided to focus on a basic instruction set architecture for a 32 bit MIPS. This architecture was a multi-cycle datapath that utilized single level memory. We chose to simulate how loading, storing, and adding information would progress through a multi-cycle datapath.

All of our MIPS instructions are of either R-format or I-format. As a team, we felt that these instructions would showcase various elements of the datapath thoroughly, while keeping the code as streamlined as possible. This meant we choose to not do a pipelined datapath and a single level memory for both instruction and data memory as it would keep the design of the processor more simple.

The majority of our code is based on the 12.9.6 and 12.9.7 datapath and CPU modules in Zybooks. We modified the ALUControl, Mult3to1, Mult4to1, and MIPSALU modules. The register file was modified from page 45 of the Verilog slides that we went over in class. It took us a lot of time and effort to truly understand how the Zybooks code works. We ran into lots of semantic and syntax errors within our verilog code which we spent multiple hours attempting to debug. We gathered a lot of information from various datapath diagrams from zybooks and then set out to make our own version of the MIPS processor.

Verilog code necessary to model a MIPS processor with memory:

The verilog code file is included.

Sample short MIPS program:

```
sw $2 0($3);
add $6 $4 $5
lw $2 0($3);
```

Description of the process of executing this sample short MIPS program:

In a multicycle datapath, including the one we implemented in Verilog, instructions are broken down into different steps, with each taking 1 clock cycle to execute. Each functional unit can be used multiple times for a particular instruction, provided that the datapath is used in different clock cycles. Instructions are broken down into these steps to reduce the necessary hardware and the instruction completion time. The lists below

describe the steps followed by our Verilog processor to complete the sw, add, and lw commands included in our sample MIPS program.

Save Word Process in Multicycle Datapath: sw $2 0($3);

1. Instruction Fetch
    - The IR register is set equal to the complete MIPS instruction in binary, which is fetched using the PC as an address in Memory: IR = Memory[PC]
    - The PC address is incremented by 4 using the ALU: PC = PC + 4
    - At this point the program does not know the identity of the instruction
2. Instruction Decode/Register Fetch
    - Read the values of the appropriate rs and rt registers from the register file using sliced versions of the IR register. Store the value of rs in the A register and the value rt in the B register: A = Reg[IR[25-21]] and B = Reg[IR[20-16]]
3. Memory Address Computation
    - Compute the memory address at which information is to be saved. ALUOut register is set equal to the sum of A and the sign extended version of the offset, obtained by slicing the IR register: ALUOut = A + sign-extend(IR[15-0])
4. Memory Access
    - The memory at the address of ALUOut is set equal to the value stored in the register B.

Add Process in Multicycle Datapath: add $6 $4 $5

1. Instruction Fetch
    - The IR register is set equal to the complete MIPS instruction in binary, which is fetched using the PC as an address in Memory: IR = Memory[PC]
    - The PC address is incremented by 4 using the ALU: PC = PC + 4
    - At this point the program does not know the identity of the instruction
2. Instruction Decode/Register Fetch
    - Read the values of the appropriate rs and rt registers from the register file using sliced versions of the IR register. Store the value of rs in the A register and the value rt in the B register: A = Reg[IR[25-21]] and B = Reg[IR[20-16]]
3. Execution
    - The values of A and B are summed together using the ALU and the result is stored in the ALUOut register: ALUOut = A + B
4. R-Type Completion
    - The appropriate register in the register file is written using a sliced version of the IR register. The value that is written into the register was previously stored in the ALUOut register: Reg[IR[15-11]] = ALUOut

Load Word Process in Multicycle Datapath: lw $2 0($3);

1. Instruction Fetch
    - The IR register is set equal to the complete MIPS instruction in binary, which is fetched using the PC as an address in Memory: IR = Memory[PC]
    - The PC address is incremented by 4 using the ALU: PC = PC + 4
    - At this point the program does not know the identity of the instruction
2. Instruction Decode/Register Fetch
    - Read the values of the appropriate rs and rt registers from the register file using sliced versions of the IR register. Store the value of rs in the A register and the value rt in the B register: A = Reg[IR[25-21]] and B = Reg[IR[20-16]]
3. Memory Address Computation
    - Compute the memory address at which information is to be read using a sliced version of the IR register. ALUOut register is set equal to the sum of A and the sign extended version of the offset, obtained by slicing the IR register: ALUOut = A + sign-extend(IR[15-0]).
4. Memory Access
    - The value at the memory address contained in the register ALUOut is stored in the MDR register: MDR = Memory[ALUOut]
5. Memory read Completion (write back)
    - The appropriate register in the register file is accessed using a sliced version of the instruction contained in the IR register. This register is set equal to the value contained in the MDR register.

Our Verilog CPU determines which process to follow based on control signals including:

- PCWriteCond: when 1, the PC is written if the Zero output from the ALU is also active
- PCWrite: when 1, the PC is written, the source is controlled by PCSource
- IorD: controls Mux that selects between PC and ALUOut for memory address
- MemRead: when 1, the content of memory at the location specified by the Address input is put on Memory data output
- MemWrite: Memory contents at the location specified by the Address Input is replaced by the value in the Write Data input.
- MemtoReg: controls Mux that selects between ALUOut and memory data for write data input to register file
- IRWrite: The output of the memory is written to the IR
- PCSource: controls the source that will be stored in the PC register for the next cycle through the datapath.
- ALUOp: controls which operation is performed by the ALU unit

- ALUSrcB: controls Mux that selects between register data B, constant 4, sign-extended immediate, sign-extended, shifted immediate for second operand input to ALU
- ALUSrcA: controls Mux that selects between PC and register data A for first operand input to ALU
- RegWrite: when 1, the general purpose register selected by the Write register number is written with the value of the Write data input

Inputs and Outputs of Each Module that is run:

```
ALUControl:
  INPUTS:
        ALUOp: 10
        Funct: xxxxxx
  OUTPUTS:
        ALUCtl: 0110


Mult4to1:
  INPUTS:
        B: x
        constant4: 4
        SignExtendOffset: x
        PCOffset: X
        ALUSrcB: 01
  OUTPUTS:
        ALUBin: 4


Mult3to1:
  INPUTS:
        ALUResultOut: x
        ALUOut: x
        JumpAddr: X
        PCSource: 00
  OUTPUTS:
        PCValue: x


ALU:
  INPUTS:
        ALUCtl: 0110
        ALUAin: 0
        ALUBin: 4
  OUTPUTS:
        ALUResultOut: 4294967292
```

```
Mult3to1:
  INPUTS:
        ALUResultOut: 4294967292
        ALUOut: x
        JumpAddr: 34078720
        PCSource: 00
  OUTPUTS:
        PCValue: 4294967292


ALUControl:
  INPUTS:
        ALUOp: 10
        Funct: 000000
Mult4to1:
  INPUTS:
        B: x
        constant4: 4
        SignExtendOffset: 0
        PCOffset: X
        ALUSrcB: 01
  OUTPUTS:
        ALUBin: 4


Mult4to1:
  INPUTS:
        B: x
        constant4: 4
        SignExtendOffset: 0
        PCOffset: 0
        ALUSrcB: 01
  OUTPUTS:
        ALUBin: 4
```

```
registerfile:
  INPUTS:
        rs: 00100
        rt: 00010
        Writereg: 00010
        Writedata: x
        RegWrite: 0
        clock: 1
  OUTPUTS:
        A: x
        B: x


Mult3to1:
  INPUTS:
        ALUResultOut: 4294967292
        ALUOut: 4294967292
        JumpAddr: 34078720
        PCSource: 00
  OUTPUTS:
        PCValue: 4294967292


Mult4to1:
  INPUTS:
        B: 1
        constant4: 4
        SignExtendOffset: 0
        PCOffset: 0
        ALUSrcB: 01
  OUTPUTS:
        ALUBin: 4
```

```
Mult3to1:
  INPUTS:
        ALUResultOut: 4294967292
        ALUOut: 4294967292
        JumpAddr: 0
        PCSource: 00
  OUTPUTS:
        PCValue: 4294967292


registerfile:
  INPUTS:
        rs: 00000
        rt: 00000
        Writereg: 00000
        Writedata: 4294967292
        RegWrite: 0
        clock: 1
  OUTPUTS:
        A: 2
        B: 1


Mult4to1:
  INPUTS:
        B: x
        constant4: 4
        SignExtendOffset: 0
        PCOffset: 0
        ALUSrcB: 01
  OUTPUTS:
        ALUBin: 4
```

```
registerfile:
  INPUTS:
        rs: 00000
        rt: 00000
        Writereg: 00000
        Writedata: 4294967292
        RegWrite: 0
        clock: 1
  OUTPUTS:
        A: x
        B: x


registerfile:
  INPUTS:
        rs: 00000
        rt: 00000
        Writereg: 00000
        Writedata: 4294967292
        RegWrite: 0
        clock: 1
  OUTPUTS:
        A: x
        B: x


Simulation Done
```

Explain contributions of each group member:

Taylor worked on the write up, datapath diagrams, and development. Down below are all of the diagrams and breakdowns that she has drawn for the project, which helped us understand how we needed to block the code. She worked on the beginning of the write up, writing the choices made section and editing the rest of the document as well. For the development, she worked on the Register File module to make sure our registers were stored properly. Overall Taylor has put in 15 hours into this project over the week.

Shanelle worked on the write up, coded the logic for the processor, and development. In the write up she worked on the choices made section, the description of the program section and edited the rest of the document. In regards to the logic, she helped block out the code to take it from a diagram on a white board to a fully running program. During the development, she worked on the CPU Module and test bench to get the code running in verilog how it should. Overall Shanelle has put in 15 hours into this project over the week.

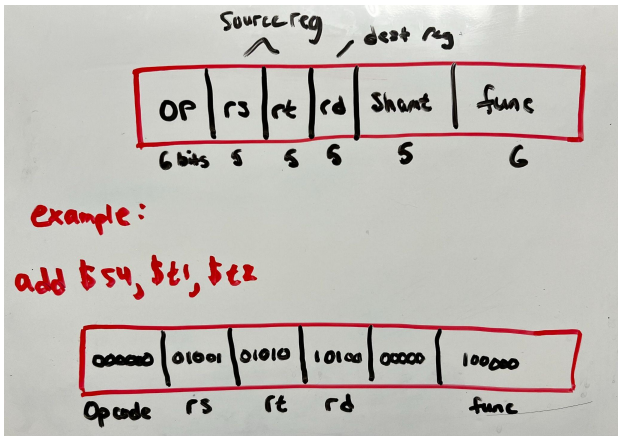
Aiden worked on the write up, datapath logic, and development. For the write up, he worked on the explanation of the mips program and edited the rest of the document. For the data path logic, he took the diagrams we had and worked with the group to break it down and block it out for the code to function. After breaking apart the diagrams he worked on the datapath module and some of the smaller module logic. Overall Aiden has put in 15 hours into this project over the last week.

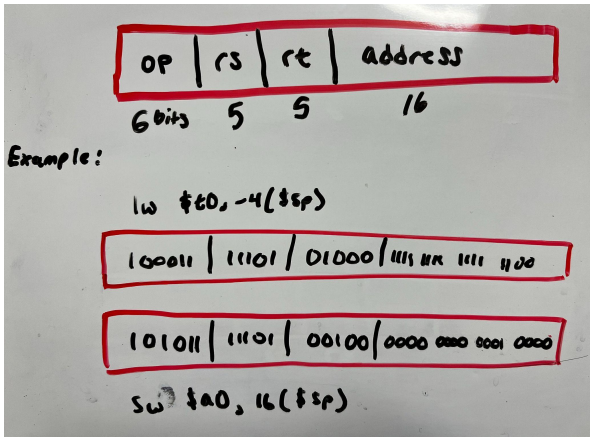
AJ worked on the write up, logic for the processor, and code development. For the write up he worked on the explanation of the mips program and edited the rest of the document where needed. For the logic he worked with the rest of the group to take the diagrams that we had and turn them into an outline for the program so the logic could be implemented. Finally, for the development he worked on the datapath module and smaller modules. Overall AJ has put in 15 hours into this project over the last week.

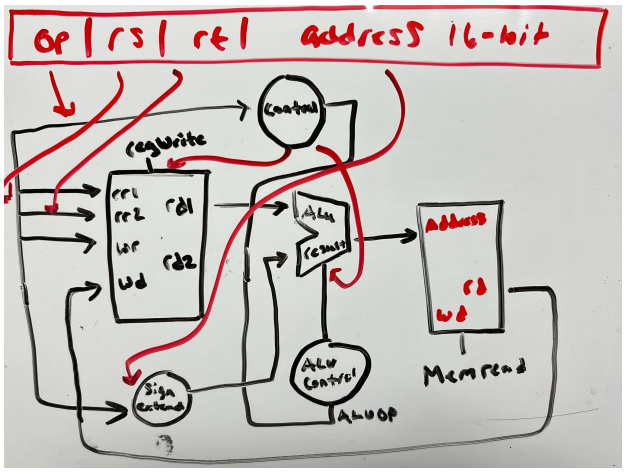
Our group demonstrated strong collaboration skills by working together to complete the project. By choosing to tackle the project as a team, we were able to overcome any challenges that arose and make progress towards the project's goals. Our approach allowed us to understand the project and complete it to the best of our abilities. Working together and effective communication are important skills, and we as a team utilized them well.

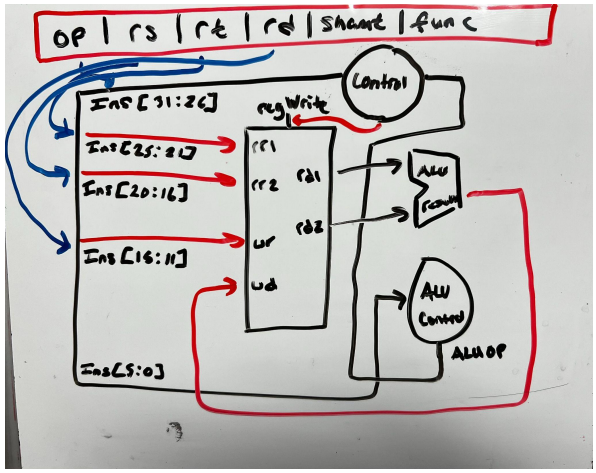
Diagram Process:

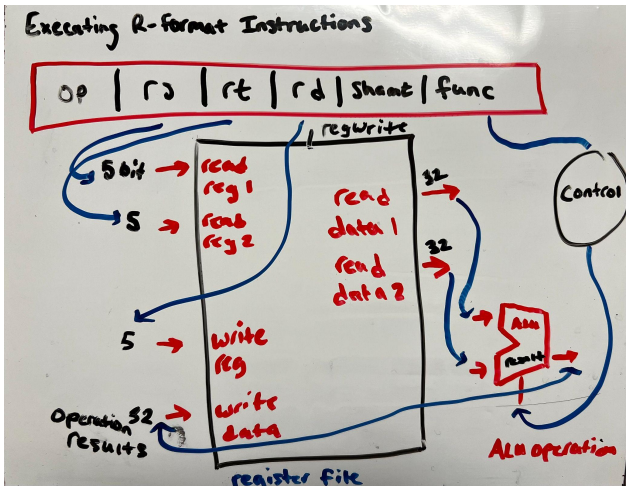
The diagrams below were our first attempts at building our understanding of how different types of instructions are executed in a broken down version of a multi-cycle datapath.





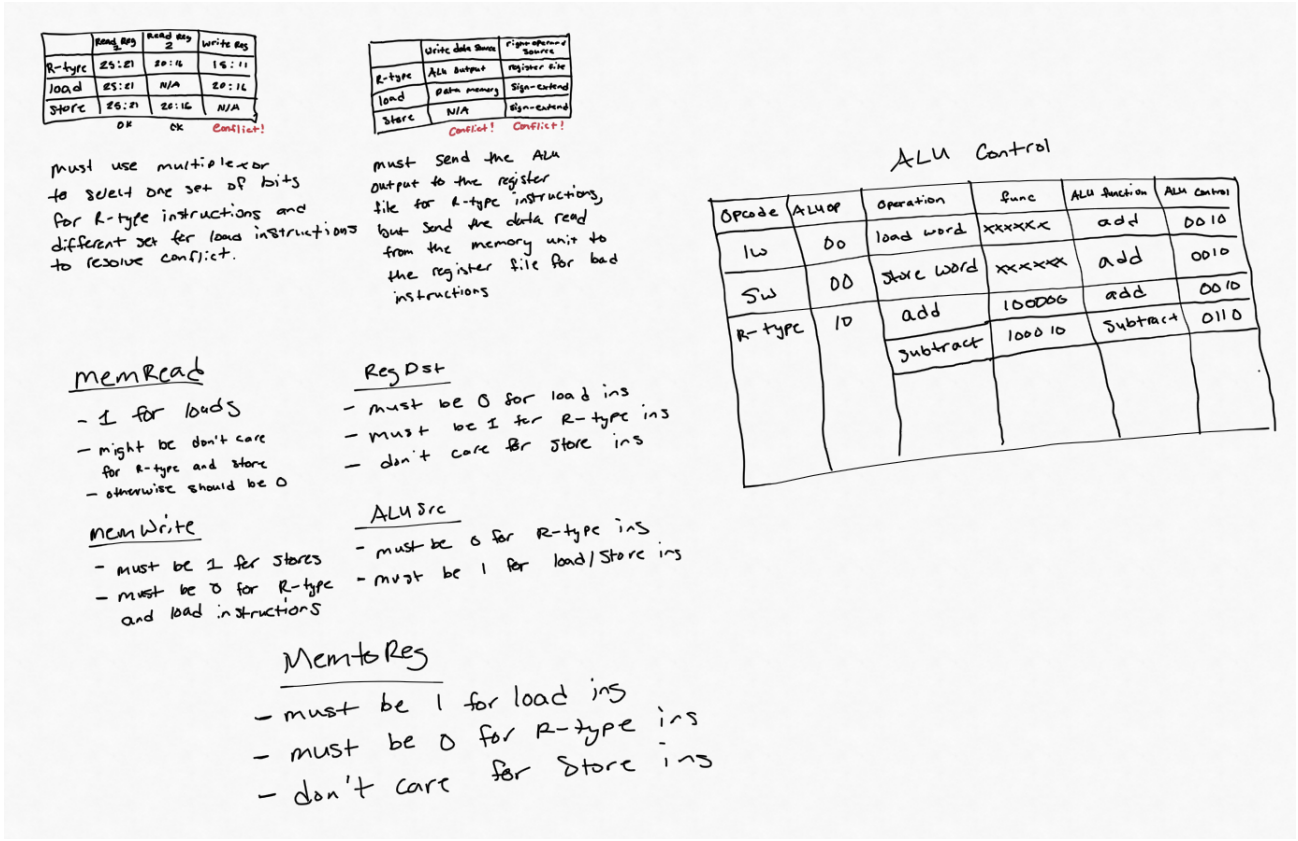






Conflict Resolution and ALU Control:

In a non-pipelined datapath, only one instruction can be executed at a time, and each instruction must be completed before the next one can begin. The multi-cycle datapath that we created may have conflicts between R-type and load instructions if they share resources such as registers or memory access units. Each instruction typically takes multiple clock cycles to execute, and different instructions may require different numbers of cycles to complete. If a load instruction and an R-type instruction try to access the same resource at the same time, there will be a conflict that must be resolved. This was our teams thought process on how to deal with those conflicts and some basic outlining for our control signals.



References section:

https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec08.pdf
http://staff.ustc.edu.cn/~han/CS152CD/Content/COD3e/CDSections/CD5.8.pdf
https://www.cs.umd.edu/~meesh/cmsc311/clin-cmsc311/Lectures/lecture33/multi_cycle.pdf

Bloom, Gedare. “MIPS Single Cycle Sequential Processor.” *YouTube*, 27 Aug. 2021, www.youtube.com/watch?v=MVNGdje5AFc.

TheMobius6. “MIPS Multicycle Datapath Instruction Steps Tutorial.” *YouTube*, 5 Dec. 2011, www.youtube.com/watch?v=23NONE7u94A.

Nelson, Brent. Designing Digital Systems With SystemVerilog (v2.1). Independently published, 2021.
http://staff.ustc.edu.cn/~han/CS152CD/Content/COD3e/CDSections/HP_AppB.pdf