Final Report

1. Team Name and Members

Team name: Architecture Dawgs

Team members: Veera Karri, Mabry Wilkinson, Siddhant Sutar

2. Meetings

Total number of meetings: 6

Name	Attendance	Absences	Total attendance time
Siddhant Sutar	6	None	20 hours
Veera Karri	6	None	20 hours
Mabry Wilkinson	6	None	20 hours

3. Implementation Diagram

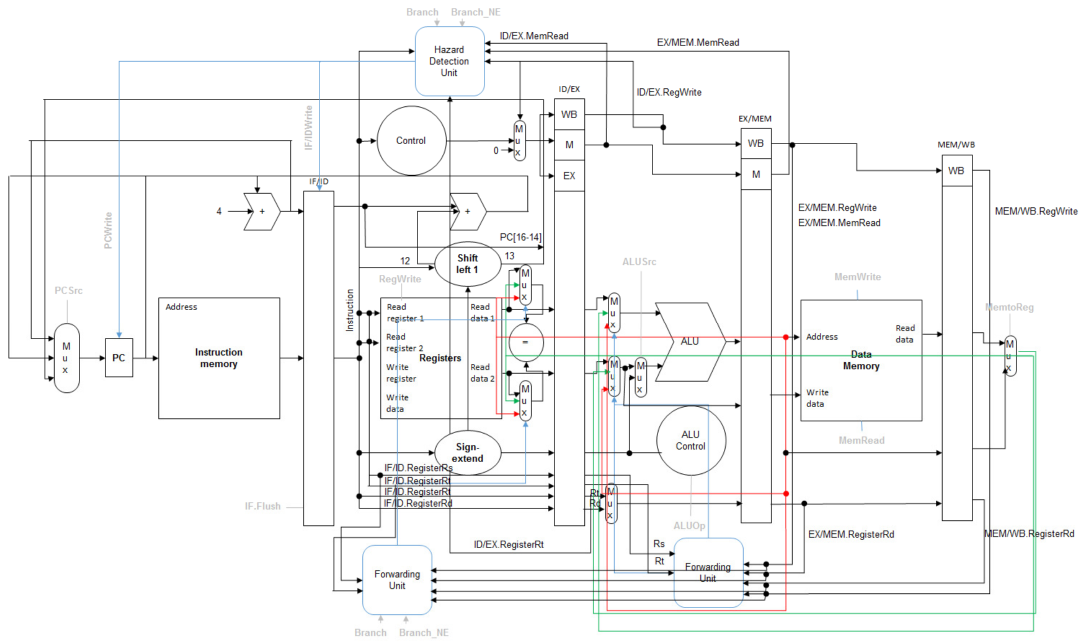

Fig. 1. Final datapath implementation. Blue lines indicate outputs from hazard detection and forwarding units. Green lines indicate forwarding path 01 (MemtoReg multiplexer output). Red lines indicate forwarding path 10 (ALU result output).

4. ALU Control

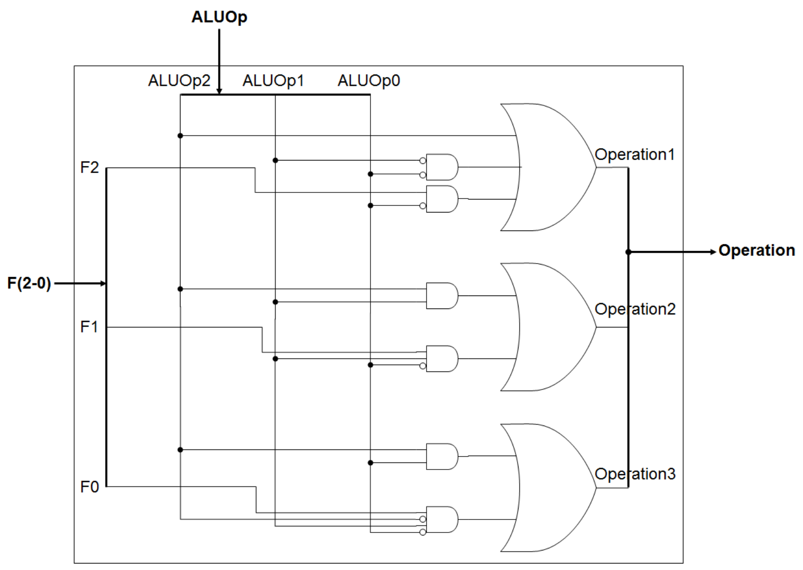

Fig. 2. ALU Control Unit.

Since our instruction set featured 8 different types of operations, there was a need to implement three ALUOp bits, i.e., three ALUOp control signals. To design the ALU control unit, we took the following approach:

- For R-format instructions, map Funct code to Operation code (irrespective of ALUOp code)
- For `lw` and `sw` instructions, map Funct code and ALUOp code to 100 (addition operation)
- For I-format instructions, map ALUOp code to the Operation code

To assist with this process, the originally proposed Funct codes were modified. ALUOp2 bit for operations with a corresponding I-format (`add/addi`, `and/andi`, `or/ori`, `slt/slti`) instruction described in the instruction set was set to 1 to distinguish between these and the other instructions. Since we mapped the Operation code to the ALUOp code for I-format instructions, the Funct code for the corresponding arithmetic operation was set to the same Operation code. For example, for addition operation, the operation code was set to 100. Thus, the ALUOp code for `addi` was set to 100. Similarly, the Funct code for `add` was set to 100.

Taking each Funct bit into consideration, truth tables were constructed, and using sum of products (SOP) approach, the following equations were derived for the Operation bits:

```
Operation(2) = ALUOp2 + (~ALUOp1 * ~ALUOp0) + (F2 * ~ALUOp0)

Operation(1) = (ALUOp2 * ALUOp1) + (F1 * ALUOp1 * ~ALUOp0)

Operation(0) = (ALUOp2 * ALUOp0) + (F0 * ~ALUOp2 * ALUOp1 *
~ALUOp0)
```

Operation	**Code**
Add	100
Bitwise AND	101
Bitwise OR	110
Set Less Than	111
Bitwise XOR	000
Logical Shift Left	001
Logical Shift Right	010
Subtract	011

For the modified instruction set, please refer to Appendix 12.b.

5. Control Unit

The Control Unit in the ID stage generates RegDst, ALUSrc, ALUOp2, ALUOp1, ALUOp0, Jump, Branch, Branch_NE, MemRead, MemWrite, MemToReg, RegWrite control signals that are either referenced in the same stage (branch signals) or subsequent stages. The control unit takes in the first 4-bits of instruction (opcode) and sets the appropriate control signals based on the type of instruction decoded from the opcode. For writing to pipeline buffers, the control unit outputs the control signals to a multiplexer which has the Hazard Detection Unit output as its select input. If this value equals 1, hazard is not detected and the appropriate signals are written to the ID/EX buffer; if 0, hazard is detected and all the signals are associated with a 0 value when writing to the buffer for the subsequent stages to interpret a stall cycle.

6. Forwarding Unit

a. EXE Forwarding Unit

The EXE Forwarding Unit is used to forward the appropriate paths to the instruction currently in the EXE stage in case of a *regular* (non-load-use) *data hazard*. It takes in the addresses of Rs and Rt registers for the current instruction as well as the addresses of Rd registers, RegWrite (if preceding instruction writing to register) control signal from the EX/MEM and MEM/WB buffers (set by the previous instructions), and asserts the appropriate path(s) as the output if forwarding is required. If the destination register for the preceding instruction(s) matches Rs and/or Rt register for the current instruction, forwarding is required. For forwarding, the multiplexer input 00 indicates the originally read values (if forwarding not required), 01 indicates MemtoReg multiplexer output, and 10 indicates ALU result output. The EXE forwarding unit implemented in the simulator handles EXE hazards (forwarding from MEM to EXE), MEM hazards (forwarding from WB to EXE) as well as double data hazards if both the hazards are prevalent.

b. ID Forwarding Unit

Similar to the EXE Forwarding Unit, the ID Forwarding Unit forwards appropriate paths to the branch (or branch not equal) instruction currently in the ID stage in case of a *regular* (non-load-use) *data hazard on branches* to support the implementation of new branch hardware in the ID stage. The forwarded values are then compared for equality using an equality comparator and the branch is taken depending upon the instruction (`beq` or `bne`). ID forwarding unit takes in the same inputs as the EXE forwarding unit with the addition of Branch and Branch_NE control signal inputs. Thus, this forwarding unit is only called upon if the instruction is a branch instruction. The outputs are the same as the EXE forwarding unit outputs.

7. Hazard Detection Unit

The Hazard Detection Unit from the ID stage detects *load-use data hazards* and is extended to deal with *data hazards on branches*. It inserts stall cycles within the subsequent stages (EXE, MEM, and WB) if a hazard is detected. For load-use data hazards, the unit takes in the address of the register Rt and the control signal MemRead from ID/EX buffer. After extending to support data hazards on branches, it additionally takes in control signals RegWrite from the ID/EX buffer and MemRead from EX/MEM buffer, as well as branch control signals (Branch and Branch_NE) for the instruction currently in the ID stage. The unit outputs a value of 0 if hazard is detected and 1 otherwise. This value is the select input for the multiplexer that outputs to write the control signals in ID/EX buffer. A load-use data hazard is detected if the preceding instruction is reading from memory (MemRead = 1) and its Rt register matches with the Rs or Rt of the current instruction. For load-use data hazard on branches, a hazard is detected if the Rd of the preceding instruction in EXE or MEM stage matches Rs or Rt of the current instruction as well as the condition that the control signal RegWrite (from ID/EX) or MemRead (from ID/EX or EX/MEM) is 1.

If a hazard is detected, the control signals values are set as 0 when writing to the ID/EX buffer. In the next clock cycle, the same instruction, now in EXE stage has 0 as its control signal values which signifies a bubble in the pipeline and an appropriate stall is inserted. These control signal values are carried over until this instruction finishes WB stage; thus stall cycles are subsequently inserted in MEM and WB stages as well. Similarly, if a hazard is detected, the global signals are PCWrite and IF/IDWrite are set to 0, signalling the processor to not write to the program counter and IF/ID buffer at the end of the clock cycle. During the next clock cycle, these signals are initialized back to 1 again.

a. Data dependencies and hazard conditions

```
         addi $2, $2, -2

        j END //Control hazard - generates 2 stall cycles after jump is
taken

LOOP:   addi $1, $1, -1 //Jump delay slot

        lw $3, 0($2)

        sub $0, $0, $0

        addi $0, $0, 1 //EXE hazard on Rs, Rt
        sll $0, $0, 4 //EXE hazard on Rs

        sll $0, $0, 4 //EXE hazard on Rs

        slt $3, $0, $3 //EXE hazard on Rs

        sub $0, $0, $0 //MEM hazard on Rs, Rt

        addi $0, $0, 1 //EXE hazard on Rs, Rt

        bne $3, $0, ELSE //Data hazard on branch (Rs)

IF:     srl $4, $4, 3 //Branch delay slot

        or $5, $5, $4 //EXE hazard on Rt

        sub $3, $3, $3

        addi $3, $3, 15 //EXE hazard on Rs, Rt

        sll $3, $3, 4 //EXE hazard on Rs

        addi $3, $3, 15 //EXE hazard on Rs, Rt

        sll $3, $3, 4 //EXE hazard on Rs

        sll $3, $3, 4 //EXE hazard on Rs

        sw $3, 0($2) //EXE hazard on Rt

        j END //Control hazard - generates 2 stall cycles after jump is
taken

ELSE:   sll $6, $6, 2

        xor $7, $7, $6 //EXE hazard on Rt

        sub $3, $3, $3

        addi $3, $3, 15 //EXE hazard on Rs, Rt

        sll $3, $3, 4 //EXE hazard on Rs

        addi $3, $3, 15 //EXE hazard or Rs, Rt

        sw $3, 0($2) //EXE hazard on Rt

        j END //Control hazard - generates 2 stall cycles after jump is
taken

END:    addi $2, $2, 2 //Branch delay slot

        sub $3, $3, $3

        slt $3, $3, $1 //EXE hazard on Rs

        sub $0, $0, $0

        addi $0, $0, 1 //EXE hazard on Rs, Rt
        bne $3, $0, EXIT //Data hazard on branch (Rs)

        j LOOP //Control hazard - generates 2 stall cycles after jump is
taken, also branch delay slot

EXIT: //Branch delay slot
```

b. Hazard management

i. *Regular (non-load-use) data hazard*

Most prevalent data hazard in the test program machine code. EXE forwarding unit handles EXE data hazards (forward from MEM stage to EXE stage) and MEM data hazards (forward from WB stage to EXE stage).

ii. *Load-use data hazard*

Hazard detection unit in the ID stage detects load-use data hazard and inserts an appropriate stall cycle based on the aforementioned functionality. The test program encounters no load-use data hazards.

iii. *Control hazard*

The branch and jump hardware in the ID stage deals with control hazards with the help of an equality comparator to detect the need to take a branch or a jump. In ID stage, based on the Branch/Branch_NE signals generated from the opcode, the comparison check is done. The branch is taken if any of the following conditions are satisfied:

- The instruction is a jump instruction.
- Branch = 1, and equality comparator returns True
- Branch_NE = 1, and equality comparator returns False

Subsequently, the **IF.Flush** signal is set to 1 whereby the instruction read in the IF stage (branch delay slot) is flushed to a NOP instruction (`0000 0000 0000 0000`) for the ID stage in the next cycle to interpret. Thus, if the branch is not taken, no branch penalty is paid.

iv. *Regular (non-load-use) data hazard on branches*

The ID forwarding unit handles non-load-use data hazard on branches by forwarding from MEM stage and/or WB stage to the branch instruction in ID stage. A data hazard on branch is encountered twice in the test program as appropriate branches are taken to `IF` or `ELSE` block inside the loop.

v. *Load-use data hazard on branches*

An extension to the hazard detection unit (as described above) in ID stage handles load-use data hazard on branches by inserting appropriate stall cycles. No load-use data hazard on branches are encountered in the test program.

8. Simulation results: memory and register file

a. Memory and register file contents at clock cycle 0

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0x0101	0000 0001 0000 0001
0x0012	0x0110	0000 0001 0001 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Instruction memory

Address	Hex Value	Binary Value
0x1000	0x14BE	00010100 10111110
0x1002	0x501E	01010000 00011110
0x1004	0x127F	00010010 01111111
0x1006	0x74C0	01110100 11000000
0x1008	0x0003	00000000 00000011
0x100A	0x1001	00010000 00000001
0x100C	0x0101	00000001 00000001
0x100E	0x0101	00000001 00000001
0x1010	0x00DF	00000000 11011111
0x1012	0x0003	00000000 00000011
0x1014	0x1001	00010000 00000001
0x1016	0x40C9	01000000 11001001
0x1018	0x08E2	00001000 11100010
0x101A	0x0B2E	00001011 00101110
0x101C	0x06DB	00000110 11011011

0x101E	0x16CF	00010110 11001111
0x1020	0x0719	00000111 00011001
0x1022	0x16CF	00010110 11001111
0x1024	0x0719	00000111 00011001
0x1026	0x0719	00000111 00011001
0x1028	0xA4C0	10100100 11000000
0x102A	0x501E	01010000 00011110
0x102C	0x0CB1	00001100 10110001
0x102E	0x0FB8	00001111 10111000
0x1030	0x06DB	00000110 11011011
0x1032	0x16CF	00010110 11001111
0x1034	0x0719	00000111 00011001
0x1036	0x16CF	00010110 11001111
0x1038	0x84C0	10100100 11000000
0x103A	0x501E	01010000 00011110
0x103C	0x1482	00010100 10000010
0x103E	0x06DB	00000110 11011011
0x1040	0x065F	00000110 01011111
0x1042	0x0003	00000000 00000011
0x1044	0x1011	00010000 00000001
0x1046	0x40C0	01000000 11000000
0x1048	0x5002	01010000 00000010
0x104A	0x0000	00000000 00000000

iii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0000	0000 0000 0000 0000
1 ($a1)	0x0005	0000 0000 0000 0101
2 ($a0)	0x0010	0000 0000 0001 0000
3 ($t0)	0x0000	0000 0000 0000 0000
4 ($v0)	0x0040	0000 0000 0100 0000
5 ($v1)	0x1010	0001 0000 0001 0000
6 ($v2)	0x000f	0000 0000 0000 1111
7 ($v3)	0x00f0	0000 0000 1111 0000

b. Memory and register file contents for each loop

After the first loop (ends at clock cycle 37)

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0x0110	0000 0001 0001 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0001	0000 0000 0000 0001
1 ($a1)	0x0004	0000 0000 0000 0100
2 ($a0)	0x0012	0000 0000 0001 0010
3 ($t0)	0xFF00	1111 1111 0000 0000
4 ($v0)	0x0008	0000 0000 0000 1000
5 ($v1)	0x1018	0001 0000 0001 1000
6 ($v2)	0x000F	0000 0000 0000 1111
7 ($v3)	0x00F0	0000 0000 1111 0000

<u>After the second loop (ends at clock cycle 66)</u>

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x0011	0000 0000 0001 0001
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0001	0000 0000 0000 0001
1 ($a1)	0x0003	0000 0000 0000 0011
2 ($a0)	0x0014	0000 0000 0001 0100
3 ($t0)	0xFF00	1111 1111 0000 0000
4 ($v0)	0x0001	0000 0000 0000 0001
5 ($v1)	0x1019	0001 0000 0001 1001
6 ($v2)	0x000F	0000 0000 0000 1111
7 ($v3)	0x00F0	0000 0000 1111 0000

After the third loop (ends at clock cycle 94)

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00F0	0000 0000 1111 0000
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0001	0000 0000 0000 0001
1 ($a1)	0x0002	0000 0000 0000 0011
2 ($a0)	0x0016	0000 0000 0001 0110
3 ($t0)	0x00FF	0000 0000 1111 1111
4 ($v0)	0x0001	0000 0000 0000 0001
5 ($v1)	0x1019	0001 0000 0001 1001
6 ($v2)	0x003C	0000 0000 0011 1100
7 ($v3)	0x00CC	0000 0000 1100 1100

After the fourth loop (ends at clock cycle 122)

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00FF	0000 0000 1111 1111
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0001	0000 0000 0000 0001
1 ($a1)	0x0001	0000 0000 0000 0001
2 ($a0)	0x0018	0000 0000 0001 1010
3 ($t0)	0x00FF	0000 0000 1111 1111
4 ($v0)	0x0001	0000 0000 0000 0001
5 ($v1)	0x1019	0001 0000 0001 1001
6 ($v2)	0x00F0	0000 0000 1111 0000
7 ($v3)	0x003C	0000 0000 0011 1100

<u>After the fifth loop (ends at clock cycle 150)</u>

i. Data memory

Address	Hex Value	Binary Value
0x0000	0x0000	0000 0000 0000 0000
0x0002	0x0000	0000 0000 0000 0000
0x0004	0x0000	0000 0000 0000 0000
0x0006	0x0000	0000 0000 0000 0000
0x0008	0x0000	0000 0000 0000 0000
0x000A	0x0000	0000 0000 0000 0000
0x000C	0x0000	0000 0000 0000 0000
0x000E	0x0000	0000 0000 0000 0000
0x0010	0xFF00	1111 1111 0000 0000
0x0012	0xFF00	1111 1111 0000 0000
0x0014	0x00FF	0000 0000 1111 1111
0x0016	0x00FF	0000 0000 1111 1111
0x0018	0x00FF	0000 0000 1111 1111
0x001A	0x0000	0000 0000 0000 0000
0x001C	0x0000	0000 0000 0000 0000
…	0x0000	0000 0000 0000 0000

ii. Register file

Register #	Hex Value	Binary Value
0 ($t1)	0x0001	0000 0000 0000 0001
1 ($a1)	0x0000	0000 0000 0000 0000
2 ($a0)	0x001A	0000 0000 0001 1010
3 ($t0)	0x00FF	0000 0000 1111 1111
4 ($v0)	0x0001	0000 0000 0000 0001
5 ($v1)	0x1019	0001 0000 0001 1001
6 ($v2)	0x03C0	0000 0011 1100 0000
7 ($v3)	0x03FC	0000 0011 1111 1100

9. Integrated output from simulation results

i. Data memory

Address	Hex value after each loop					
	Initial	1st	2nd	3rd	4th	5th
0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0002	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0004	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0006	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0008	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x000A	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x000C	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x000E	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0010	0x0101	0xFF00	0xFF00	0xFF00	0xFF00	0xFF00
0x0012	0x0110	0x0110	0xFF00	0xFF00	0xFF00	0xFF00
0x0014	0x0011	0x0011	0x0011	0x00FF	0x00FF	0x00FF
0x0016	0x00F0	0x00F0	0x00F0	0x00F0	0x00FF	0x00FF
0x0018	0x00FF	0x00FF	0x00FF	0x00FF	0x00FF	0x00FF
0x001A	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x001C	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
…	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000

ii. Register file

Register	Hex value after each loop					
	Initial	1st	2nd	3rd	4th	5th
0 ($t1)	0x0000	0x0001	0x0001	0x0001	0x0001	0x0001
1 ($a1)	0x0005	0x0004	0x0003	0x0002	0x0001	0x0000
2 ($a0)	0x0010	0x0012	0x0014	0x0016	0x0018	0x001A
3 ($t0)	0x0000	0xFF00	0xFF00	0x00FF	0x00FF	0x00FF
4 ($v0)	0x0040	0x0008	0x0001	0x0001	0x0001	0x0001
5 ($v1)	0x1010	0x1018	0x1019	0x1019	0x1019	0x1019
6 ($v2)	0x000f	0x000F	0x000F	0x003C	0x00F0	0x03C0
7 ($v3)	0x00f0	0x00F0	0x00F0	0x00CC	0x003C	0x03FC

10. Discussion

Optimization occurred during the planning and implementation phase. We encountered issues throughout the process particularly with control hazards such as jumping to the correct instruction after a branch statement. We eventually discovered the proper way to keep up with the Program Counter and were able to keep track of where to jump to and from throughout the program.

We utilized the functionality of GitHub and Google Docs throughout the project. This was useful as we were all able to see and make changes to files if necessary. The Google Sheets was used as a primary debugging and tracking system. We filled the sheet up with the machine code and were able to step through it systematically to ensure that the logic of our program was correct and the outputs were correct.

Exception handling was not implemented as it was not a requirement.

11. Final version of the simulator code (attached)

12. Appendix

a. Translation of assembly to machine code (updated)

Machine Code	Assembly Code
addi $2, $2, -2	0001 010 010 111110
j END	0101 000000011110
LOOP: addi $1, $1, -1	0001 001 001 111111
lw $3, 0($2)	0111 010 011 000000
sub $0, $0, $0	0000 000 000 000 011
addi $0, $0, 1	0001 000 000 000001
sll $0, $0, 4	0000 000 100 000 001
sll $0, $0, 4	0000 000 100 000 001
slt $3, $0, $3	0000 000 011 011 111
sub $0, $0, $0	0000 000 000 000 011
addi $0, $0, 1	0001 000 000 000001
bne $3, $0, ELSE	0100 000 011 001001
IF: srl $4, $4, 3	0000 100 011 100 010
or $5, $5, $4	0000 101 100 101 110
sub $3, $3, $3	0000 011 011 011 011
addi $3, $3, 15	0001 011 011 001111
sll $3, $3, 4	0000 011 100 011 001
addi $3, $3, 15	0001 011 011 001111
sll $3, $3, 4	0000 011 100 011 001
sll $3, $3, 4	0000 011 100 011 001
sw $3, 0($2)	1010 010 011 000000
j END	0101 000000011110
ELSE: sll $6, $6, 2	0000 110 010 110 001
xor $7, $7, $6	0000 111 110 111 000
sub $3, $3, $3	0000 011 011 011 011
addi $3, $3, 15	0001 011 011 001111
sll $3, $3, 4	0000 011 100 011 001
addi $3, $3, 15	0001 011 011 001111
sw $3, 0($2)	1010 010 011 000000
j END	0101 000000011110
END: addi $2, $2, 2	0001 010 010 000010
sub $3, $3, $3	0000 011 011 011 011
slt $3, $3, $1	0000 011 001 011 111
sub $0, $0, $0	0000 000 000 000 011
addi $0, $0, 1	0001 000 000 000001
bne $3, $0, EXIT	0100 000 011 000000
j LOOP	0101 000000000010
EXIT:	0000 0000 0000 0000

b. Updated instruction set (with Funct codes)

The Funct codes were updated from the initial proposal to satisfy the requirements and to match with the ALU control hardware.

Name	Mnemonic	Operation	Opcode	Func	Format
Add	`add`	`add $s1, $s2, $s3` \$s1 = \$s2 + \$s3	`0000`	`100`	R
Add Immediate	`addi`	`addi $s1, $s2, 100` \$s1 = \$s2 + 100	`0001`	-	I
Bitwise AND	`and`	`and $s1, $s2, $s3` \$s1 = \$s2 & \$s3	`0000`	`101`	R
Bitwise AND Immediate	`andi`	`andi $s1, $s2, 100` \$s1 = \$s2 & \$s3	`0010`	-	I

Branch if Equal	`beq`	`beq $s1, $s2, $s3`
		if (\$s == \$t) pc += i << 2
			`0011`	-	I

Branch if Not Equal	`bne`	`bne $s1, $s2, $s3`
		if (\$s != \$t) pc += i << 2
			`0100`	-	I

Jump to Address	`j`	`j addr` Pc += 1 << 2	`0101`	-	J
Load Word	`lw`	`lw $s1, addr` \$t = MEM [\$s + i]:4	`0111`	-	I
Bitwise XOR	`xor`	`xor $s1, $s2, $s3` \$d = \$s ^ \$t	`0000`	`000`	R
Bitwise OR	`or`	`or $s1, $s2, $s3`	`0000`	`110`	R

		$d = $s | $t			
Bitwise OR Immediate	`ori`	`ori $s1, $s2, 10` $t = $s | ZE(i)	`1000`	-	I
Set to 1 if Less Than	`slt`	`slt $s1, $s2, $s3` $d = ($s < $t)	`0000`	`111`	R
Set to 1 if Less Than Immediate	`slti`	`slt $s1, $s2, 10` $t = ($s < SE(i))	`1001`	-	I
Logical Shift Left	`sll`	`sll $s1, $s2, 3` $d = $t << a	`0000`	`001`	R
Logical Shift Right	`srl`	`srl $s1, $s2, 3` $d = $t >>> a	`0000`	`010`	R
Subtract	`sub`	`sub $s1, $s2, $s3` $d = $s - $t	`0000`	`011`	R
Store Word	`sw`	`sw $s1, addr` MEM [$s + i]:4 = $t	`1010`	-	I

c. Instructions to run the simulator

i. Ensure that simulator.py and machine_code.mips are in the same directory.
ii. Use version of Python 3.0 or greater
iii. On the terminal, run: $python simulator.py
iv. The simulator generates output files for memory (data and instruction) and register file in a separate directory at the beginning of every clock cycle and every time the loop iteration ends (skip the first file generated for loop because of rescheduled code).
v. The files are also available at our GitHub repository.