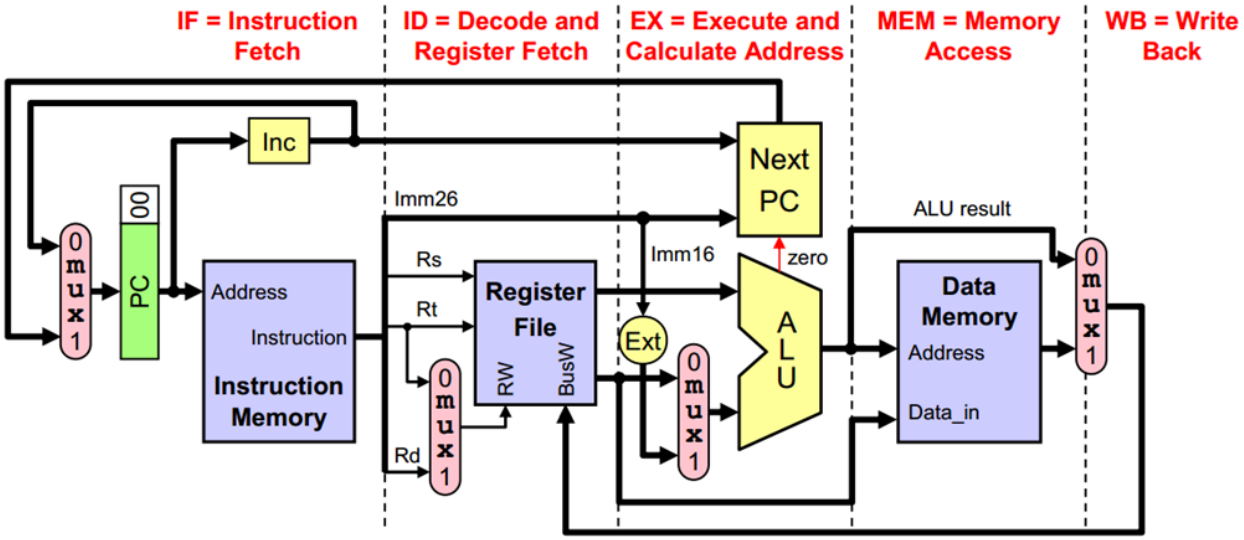
Team: William Sun

Introduction:

1. Architecture Name: FECES (Forward Error Correction Efficient Style)
2. Load-Store: Use fixed-encoding architecture like MIPS

Architectural Overview:

Use MIPS as reference for now.



Machine Specification:

1. Instruction Formats (2 bits)
    a. Memory and Comparison Operations (00)
        i. opcode(2'b) specifier(3'b) $rt(2'b) $rd(2'b)
        ii. ex: move $t, $s
    b. 1 Variable (01)
        i. opcode(2'b) specifier(1'b) imm(6'b)
        ii. ex: set i
    c. 2 Variable (10)
        i. opcode(2'b) specifier(3'b) $rt(2'b) $rd(2'b)
        ii. ex: add $s, $t
    d. 3-Variable (11)
        i. opcode(2'b) $rs(2'b) imm(5'b)
        ii. ex: beq $s, label
2. Operations
    a. Memory and Comparison Operations (00)
        i. 00_000 → move $t, $s; MEM [$t] = MEM [$s]
        ii. 00_001 → lw $t, $s; $t = MEM [$s]
        iii. 00_010 → sw $t, $s; MEM [$s] = $t

iv. 00_011 → seq $s, $t; $s = ($s == $t)
v. 00_100 → sne $s, $t; $s = ($s != $t)
vi. 00_101 → sgt $s, $t; $s = ($s > $t)
vii. 00_110 → slt $s, $t; $s = ($s < $t)
viii. SPACE FOR ONE MORE

b. 1 Variable (01)
i. 01_0 → set i; $r3 = SE(i) // NOTE: hardcode r3
ii. 01_1 → not $d; $d = ~($d)

c. 2 Variable (10)
i. 10_000 → add $s, $t; $s = $s + $t
ii. 10_001 → sub $s, $t; $s = $s - $t
iii. 10_010 → and $s, $t; $s = $s & $t
iv. 10_011 → nor $s, $t; $s = ~($s | $t)
v. 10_100 → xor $s, $t; $s = $s ^ $t
vi. 10_101 → sllv $t, $s; $t = $t << $s
vii. 10_110 → srav $t, $s; $t = $t >> $s
viii. SPACE FOR ONE MORE

d. 3 Variable (11)
i. 11 → beq $s label; if ($s == 1) pc += label

3. Internal Operands
a. Total: 16 registers
b. Temporary Use: 4 registers (r0 - r3)
i. During operation, use r0 - r3 as operands
ii. Use mov to move result into general storage
c. General Storage: 12 registers (r4 - r15)
4. Control Flow
a. Type
i. Only branch on equals to save bits
b. Target Address
i. pc += i
c. Maximum Distance
i. i < 2^5 = 32
5. Addressing Modes
a. Base Displacement Addressing
i. Adds an immediate to a register value to create a memory address
ii. lw, sw
b. PC-Relative Addressing
i. Uses the PC and adds the I-value of the instruction to create an address
ii. beq

Changelog

- Reorganize ISA opcode based on instruction format
  - Allows for more instructions
  - Allows for larger immediate values

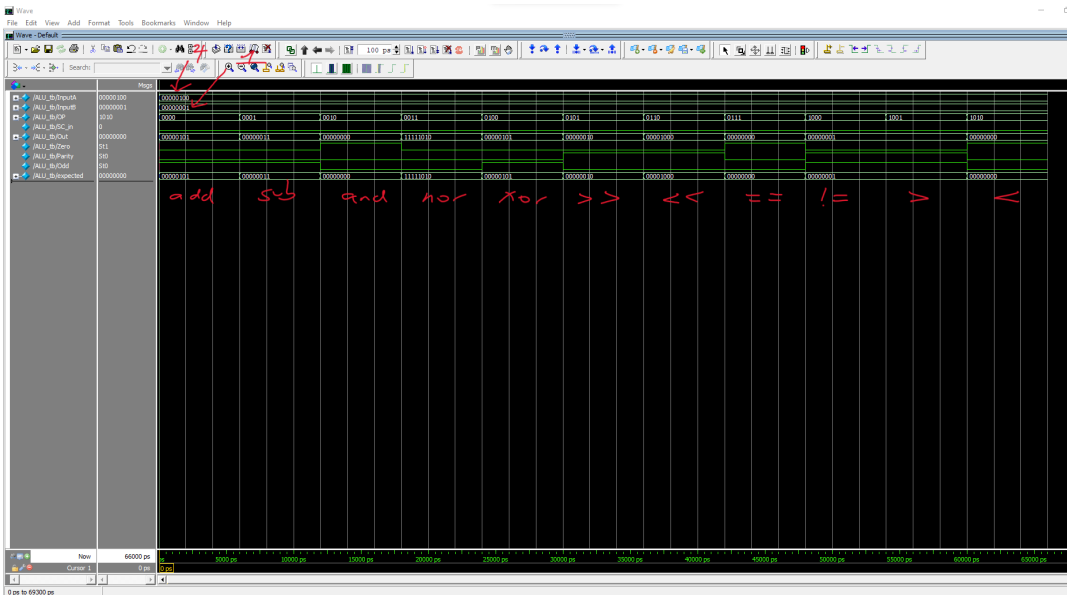
- Change 2 registers to 4
- Update control flow based on ISA changes
- Fix addressing modes issues

Programmer's Model:

1. The programmer should think of this machine as trying to conserve as many bits as possible. With only 9 bits total, we must split up the instruction encoding wisely. We use 2 bits to first split between major types of instructions. That gives us enough bits to set large immediate values (2^6). With variable instructions, we can use up all 9 bits, so we can split those further into specific instructions to increase the total number of instructions we can encode. Additionally, the programmer should abuse the hardware and use NOR as a universal gate to recreate other operations to save bits. Although this will increase CPI, it makes it easier to fit the 9-bit instruction limit.
2. Example
    a. C → $r0 = $r0 + $r1
    b. Assembly → add $r0, $r1
    c. Machine → 10_000 (add) 00 (r0) 01 (r1)

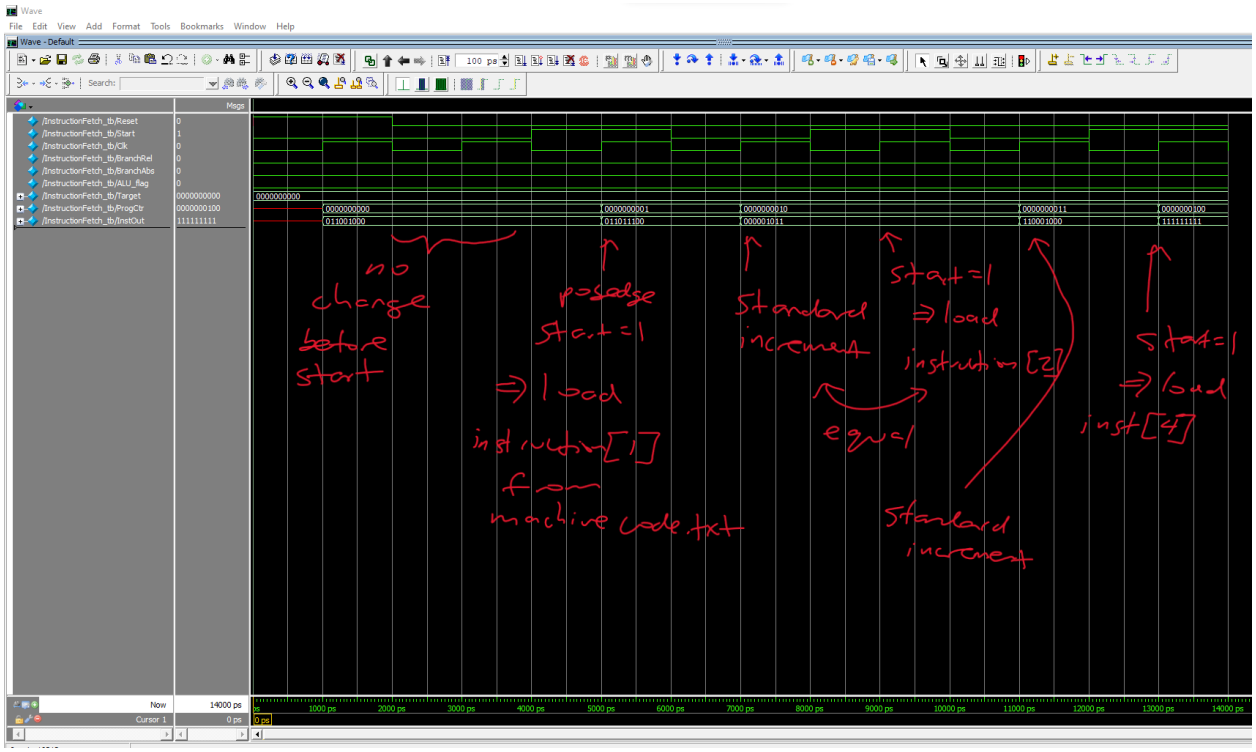
Working ALU:

```
VSIM 15> run -all
#                1000 YAY!! inputs = 04 01, OPcode = 0000, Zero 0
#                7000 YAY!! inputs = 04 01, OPcode = 0001, Zero 0
#               13000 YAY!! inputs = 04 01, OPcode = 0010, Zero 1
#               19000 YAY!! inputs = 04 01, OPcode = 0011, Zero 0
#               25000 YAY!! inputs = 04 01, OPcode = 0100, Zero 0
#               31000 YAY!! inputs = 04 01, OPcode = 0101, Zero 0
#               37000 YAY!! inputs = 04 01, OPcode = 0110, Zero 0
#               43000 YAY!! inputs = 04 01, OPcode = 0111, Zero 1
#               49000 YAY!! inputs = 04 01, OPcode = 1000, Zero 0
#               55000 YAY!! inputs = 04 01, OPcode = 1001, Zero 0
#               61000 YAY!! inputs = 04 01, OPcode = 1010, Zero 1
```



Working Instruction Fetch:

```
VSIM 7> run -all
# Check Reset
# Check nothing happens before Start
# Check first program was loaded
# Check second program was loaded
# Check third program was loaded
```



Milestone 2 Questions:

1. ALU Demonstration
    a. Demonstrated all 2 variable (opcode 10) and comparison (opcode 00) operations
2. Register File
    a. Using starter code
    b. Updated it to initialize 16 registers
3. ALU Non-Arithmetic Instructions
    a. No, my ALU only does arithmetic

Program 1:
FEC Transmitter

Description: Given a series of fifteen 11-bit message blocks in data mem[0:29], generate the corresponding 16-bit encoded versions and store these in data mem[30:59].

input MSW = 0 0 0 0 0 b11 b10 b09
LSW = b8 b7 b6 b5 b4 b3 b2 b1, where bx denotes a data bit

output MSW = b11 b10 b9 b8 b7 b6 b5 p8
LSW = b4 b3 b2 p4 b1 p2 p1 p16, where px denotes a parity bit

Algorithm:
p8 = ^(b11:b5)
p4 = ^(b11:b8,b4,b3,b2)
p2 = ^(b11,b10,b7,b6,b4,b3,b1)
p1 = ^(b11,b9,b7,b5,b4,b2,b1)
p16 = ^(b11:1,p8,p4,p2,p1)

Example:
Given 101_0101_0101
mem[1] = 00000101, mem[0] = 01010101, p8 = 0, p4 = 1, p2 = 0, p1 = 1, p16 = 0
mem[31] = 10101010 -- b11:b5, p8 = 1010101_0
mem[30] = 01011010 -- b4:b2, p4, b1, p2:p1, p16 = 010_1_1_01_0

Code:

```
addi $r0, 0             # r0 = 0
sw $r2, 0($r0)          #
loop: # while (i < 15)
# Complete with new ISA
```

Program 2:
Convert with new ISA

```
// W is data path width (8 bits)
// byte count = number of "words" (bytes) in reg_file
//   or data_memory
module top_level #(parameter W=8,
                   byte_count = 256)(
  input        clk,
               init,              // req. from test bench
  output logic done);             // ack. to test bench
```

```
// memory interface =
//   write_en, raddr, waddr, data_in, data_out:
  logic write_en;                // store enable for dat_mem

// address pointers for reg_file/data_mem
  logic[$clog2(byte_count)-1:0] raddr, waddr;

// data path connections into/out of reg file/data mem
  logic[W-1:0] data_in;
  wire [W-1:0] data_out;

/* instantiate data memory (reg file)
   Here we can override the two parameters, if we
     so desire (leaving them as defaults here) */
  dat_mem #(.W(W),.byte_count(byte_count))
    dm1(.*);                            // reg_file or data memory


// program counter: bits[6:3] count passes through for loop/subroutine
// bits[2:0] count clock cycles within subroutine (I use 5 out of 8 possible, pad w/ 3 no ops)
  logic[ 6:0] count;
  logic[ 8:0] parity;
  logic[15:0] temp1, temp2, temp_working;
  logic       temp1_enh, temp1_enl, temp2_en, temp3_en;
  logic       p8, p4, p2, p1, p0;
  logic[ 3:0] casenum;

  always_comb begin

    // hamming code
    parity[8] = ^temp1[15:9];
    parity[4] = (^temp1[15:12])^(^temp1[7:5]);
    parity[2] = temp1[15]^temp1[14]^temp1[11]^temp1[10]^temp1[7]^temp1[6]^temp1[3];
    parity[1] = temp1[15]^temp1[13]^temp1[11]^temp1[9]^temp1[7]^temp1[5]^temp1[3];
    parity[0] = ^temp1[15:1];

    // check for equality
    p8 = (parity[8] == temp1[8]);
    p4 = (parity[4] == temp1[4]);
    p2 = (parity[2] == temp1[2]);
    p1 = (parity[1] == temp1[1]);
    p0 = (parity[0] == temp1[0]);
    casenum = {p8, p4, p2, p1};
```

```
    // error detection
    if(p0 && casenum < 15) temp_working = 'b1000_0000_0000_0000; // two errors
    else if(!p0 && casenum == 15) temp_working = {5'b01000, temp1[15:9], temp1[7:5],
temp1[3]}; // p0
    else begin
      case(casenum)
        0:  temp_working = {5'b01000, !temp1[15], temp1[14:9], temp1[7:5], temp1[3]}; // b11
        1:  temp_working = {5'b01000, temp1[15], !temp1[14], temp1[13:9], temp1[7:5], temp1[3]};
// b10
        2:  temp_working = {5'b01000, temp1[15:14], !temp1[13], temp1[12:9], temp1[7:5],
temp1[3]}; // b9
        3:  temp_working = {5'b01000, temp1[15:13], !temp1[12], temp1[11:9], temp1[7:5],
temp1[3]}; // b8
        4:  temp_working = {5'b01000, temp1[15:12], !temp1[11], temp1[10:9], temp1[7:5],
temp1[3]}; // b7
        5:  temp_working = {5'b01000, temp1[15:11], !temp1[10], temp1[9], temp1[7:5], temp1[3]};
// b6
        6:  temp_working = {5'b01000, temp1[15:10], !temp1[9], temp1[7:5], temp1[3]}; // b5

        7:  temp_working = {5'b01000, temp1[15:9], temp1[7:5], temp1[3]}; // p8

        8:  temp_working = {5'b01000, temp1[15:9], !temp1[7], temp1[6:5], temp1[3]}; // b4
        9:  temp_working = {5'b01000, temp1[15:9], temp1[7], !temp1[6], temp1[5], temp1[3]}; // b3
        10: temp_working = {5'b01000, temp1[15:9], temp1[7:6], !temp1[5], temp1[3]}; // b2

        11: temp_working = {5'b01000, temp1[15:9], temp1[7:5], temp1[3]}; // p4

        12: temp_working = {5'b01000, temp1[15:9], temp1[7:5], !temp1[3]}; // b1

        13: temp_working = {5'b01000, temp1[15:9], temp1[7:5], temp1[3]}; // p2
        14: temp_working = {5'b01000, temp1[15:9], temp1[7:5], temp1[3]}; // p1

        default: temp_working = {5'b00000, temp1[15:9], temp1[7:5], temp1[3]}; // no errors
      endcase
    end

  end


  always @(posedge clk)
    if(init) begin
      count <= 0;
      temp1 <= 'b0;
      temp2 <= 'b0;
```

```
    end
    else begin
      count                   <= count + 1;
      if(temp1_enh) temp1[15:8] <= data_out;
      if(temp1_enl) temp1[ 7:0] <= data_out;
      if(temp2_en)  temp2       <= temp_working;
    end

  always_comb begin
// defaults
    temp1_enl        = 'b0;
    temp1_enh        = 'b0;
    temp2_en         = 'b0;
    raddr            = 'b0;
    waddr            = 'b0;
    write_en         = 'b0;
    data_in          = temp2[7:0];
    case(count[2:0])
      1: begin                // step 1: load from data_mem into lower byte of temp1
         raddr = 2*count[6:3] + 64;
         temp1_enl = 'b1;
      end
      2: begin                // step 2: load from data_mem into upper byte of temp1
         raddr = 2*count[6:3] + 65;
         temp1_enh = 'b1;
      end
      3: temp2_en    = 'b1;    // step 3: copy in output
      4: begin                // step 4: store from one byte of temp3 into data_mem
         write_en = 'b1;
         waddr = 2*count[6:3] + 94;
         data_in = temp2[7:0];
      end
      5: begin
         write_en = 'b1;      // step 5: store from other byte of temp3 into data_mem
         waddr = 2*count[6:3] + 95;
         data_in = temp2[15:8];
      end
    endcase
  end

// automatically stop at count 127; 120 might be even better (why?)
  assign done = &count;

endmodule
```

Program 3:
TODO