Justin Carlson

2/19/2023

Prof. Mealy

CPE 233-07

Experiment 5: Limited RISC-V MCU

Executive Summary:

In this experiment, most of the MCU was assembled to handle a limited set of instructions. The PC, memory module, ALU, registers, immediate and branch address generators, and CU decoder and FSM were all connected to handle six instructions (add, addi, lw, sw, jal, lui).

Demo Video: https://youtu.be/bi5Et4oN9Q0

Questions:

1. Sometimes the “li” pseudoinstruction causes the assembler to generate two instructions and sometimes one instruction. Briefly describe why this is the case and list the two different instructions.

If the immediate value can be expressed by 12 bits, the li instruction represents something like “addi rd, x0, imm” since the immediate value is 12 bits wide. If it’s longer than 12 bits, the li instruction also uses a lui instruction to express the full immediate value.

2. Briefly describe what a t-cycle is in a context of an FSM.

A t-cycle is a single clock pulse in an FSM that represents a change in state, like from fetch to execute or vice versa. Multiple t-cycles are needed for one full instruction cycle (either two for normal instructions or three for load instructions).

3. Briefly describe who or what decides on the number of t-cycles a given computer architecture uses.

Different instruction sets will require different amounts of t-cycles depending on the amount of computing that needs to be done, and it will affect the number of FSM states and state changes.

4. Despite not mentioning the topics much in this course, setup and hold times are important in digital design. Your experiment worked, so briefly describe how this experiment handled meeting setup and hold time issues.

Setup and hold times are important to consider in cases where the computer runs at a very high frequency, and the data in one clock cycle must be calculated and handled in the same clock cycle as to not mess anything up. Since the MCU has a slow enough clock speed to handle the necessary computations in the RISC-V architecture on a single clock cycle, we don’t necessarily have to worry about setup and hold times becoming a problem.

5. Briefly describe whether the RISC-V OTTER wrapper registers the input data (as in input/output)

or not.

No, the input data is not saved to a register, and there's no change in clock edge that would trigger the input to behave like a register, as changes to the input come from the board itself.

6. Briefly describe the difference between port mapped I/O and memory mapped I/O.

MMIO occupies the same memory space as the program memory, while the Port Mapped IO occupies its own memory segment.

7. Could you use the addi instruction to add an immediate value of 0x7421 to another register?

Briefly explain why or why not.

No, since an immediate value is limited to a 12-bit space, the 15-bit value 0x7421 cannot be used with the addi instruction.

8. You can use either the jal or jalr instruction to call subroutines, but you can only use the

jalr instruction to return from subroutines. Briefly explain why this is the case.

Since a return address is required to return from a subroutine, we can use jalr which has already been linked to a register representing our return address, but since jal doesn't have a source register that can be used to return to, it can't exactly be used as a return instruction.

9. My homework assignment is to implement 25 new pseudoinstructions. Briefly describe what

"entity" I'll be using to complete this assignment.

To create new pseudo instructions, the RISC-V assembler itself must be modified to take in new keywords for instructions, and this process is completely independent of the MCU.

10. Describe a situation where a NOP instruction or a NOP-type instruction would be useful.

Anytime as the user we want to advance the PC by 4 without doing any real instruction, we can use the NOP to advance to the next instruction which could be useful for some timing applications. The nop instruction itself is basically adding 0 to the x0 register, and since the x0 register isn't writeable, this instruction essentially does nothing but advance the PC.

11. Briefly describe why the IOBUS_ADDR is an output from the ALU and not from a register or directly from memory.

If the data doesn't need to be saved at all either through the memory or the registers, there's no need for it to be stored, so we can just use it as an output for the MCU. If the data needs to be kept, then the MCU already has functions to writeback the result to a register or store it directly to memory, so it depends on the use-case.

12. Brief explain why the state diagram listed in Figure 20 is only a "partial" state diagram. For this question, only consider the timing diagram associated with this experiment. This question has nothing to do with interrupts.

Very few of the Moore/Mealy outputs of the FSM are unspecified in that diagram inside the state bubbles, so in a sense it's incomplete.

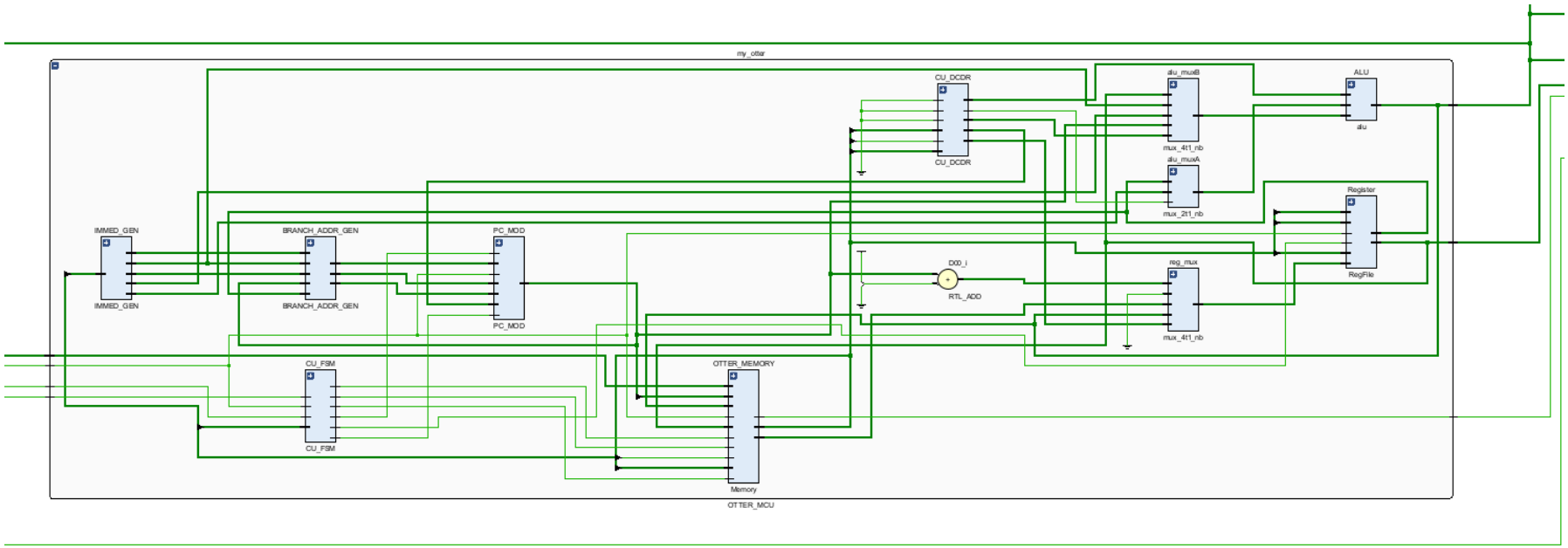

Figure 1. Elaborated Design

Verilog Code:

Top Level: (NOTE: This is the tightest line spacing I can do on this document without cutting characters off. Sorry, I'm not sure why MS Word is being this difficult)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Justin Carlson
//
// Create Date: 02/06/2023 03:01:28 PM
// Design Name:
// Module Name: OTTER_MCU
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module OTTER_MCU(
    input rst, intr, clk,
    input reg [31:0] IOBUS_IN,
    output reg [31:0] IOBUS_OUT, IOBUS_ADDR,
    output reg IOBUS_WR
    );

    // Program Counter logic
    wire reset, PCWrite;
    wire [1:0] pcSource;
    wire [31:0] jalr, branch, jal, PC;

    // Memory Logic
    wire memRDEN1, memRDEN2, MEMWE2;
    wire [31:0] ir, DOUT2;

    // Reg File Logic
```

```
wire [31:0] wd, rs1, rs2;
wire regWrite;
wire [1:0] rf_wr_sel;

// IMMED_GEN Logic
wire [31:0] u_type_imm, j_type_imm, b_type_imm, s_type_imm, i_type_imm;

// ALU Logic
wire alu_srcA;
wire [1:0] alu_srcB;
wire [3:0] alu_fun;
wire [31:0] result, srcA, srcB;

PC_MOD PC_MOD (
    .rst (reset),
    .PCWrite (PCWrite),
    .clk (clk),
    .pcSource (pcSource),
    .jalr (jalr),
    .branch (branch),
    .jal (jal),
    .PC (PC)   );

 Memory OTTER_MEMORY (
    .MEM_CLK   (clk),
    .MEM_RDEN1 (memRDEN1),
    .MEM_RDEN2 (memRDEN2),
    .MEM_WE2   (memWE2),
    .MEM_ADDR1 (PC[15:2]),
    .MEM_ADDR2 (result),
    .MEM_DIN2  (rs2),
    .MEM_SIZE  (ir[13:12]),
    .MEM_SIGN  (ir[14]), // Signed value
    .IO_IN     (IOBUS_IN),
    .IO_WR     (IOBUS_WR),
    .MEM_DOUT1 (ir),
    .MEM_DOUT2 (DOUT2)  );

 RegFile Register (
    .wd   (wd),
    .clk  (clk),
    .en   (regWrite),
    .adr1 (ir[19:15]),
```

```
    .adr2 (ir[24:20]),
    .wa   (ir[11:7]),
    .rs1  (rs1), // Output registers
    .rs2  (rs2)  );

mux_4t1_nb  #(.n(32)) reg_mux  ( // Input registers MUX
    .SEL   (rf_wr_sel),
    .D0    (PC + 4), // Advanced PC
    .D1    (32'b0), // CSR_REG
    .D2    (DOUT2), // Memory load
    .D3    (result), // ALU Output
    .D_OUT (wd) );

IMMED_GEN IMMED_GEN (
    .ir (ir[31:7]),
    .u_type_imm (u_type_imm),
    .i_type_imm (i_type_imm),
    .s_type_imm (s_type_imm),
    .j_type_imm (j_type_imm),
    .b_type_imm (b_type_imm)   );

BRANCH_ADDR_GEN BRANCH_ADDR_GEN (
    .pc (PC),
    .rs1 (rs1),
    .i_type_imm (i_type_imm),
    .j_type_imm (j_type_imm),
    .b_type_imm (b_type_imm),
    .jalr (jalr),
    .branch (branch),
    .jal (jal)   );

mux_2t1_nb  #(.n(32)) alu_muxA  (
    .SEL   (alu_srcA),
    .D0    (rs1), // Register output
    .D1    (u_type_imm), // U-type IMM
    .D_OUT (srcA) );

mux_4t1_nb  #(.n(32)) alu_muxB  (
    .SEL   (alu_srcB),
    .D0    (rs2), // Register Output
    .D1    (i_type_imm),
    .D2    (s_type_imm),
    .D3    (PC),
```

```
   .D_OUT (srcB) );

alu ALU (
   .op_1 (srcA),
   .op_2 (srcB),
   .alu_fun (alu_fun), // ALU Selector
   .result (result)   );

CU_FSM CU_FSM(
   .intr     (intr),
   .clk      (clk),
   .RST      (rst),
   .opcode   (ir[6:0]),   // ir[6:0]
   .pcWrite  (PCWrite),
   .regWrite (regWrite),
   .memWE2   (memWE2),
   .memRDEN1 (memRDEN1),
   .memRDEN2 (memRDEN2),
   .reset    (reset)   );

CU_DCDR CU_DCDR(
   .br_eq     (1'b0),
   .br_lt     (1'b0),
   .br_ltu    (1'b0),
   .opcode    (ir[6:0]),    //-  ir[6:0]
   .func7     (ir[30]),    //-  ir[30]
   .func3     (ir[14:12]),    //-  ir[14:12]
   .alu_fun   (alu_fun),
   .pcSource  (pcSource),
   .alu_srcA  (alu_srcA),
   .alu_srcB  (alu_srcB),
   .rf_wr_sel (rf_wr_sel)   );

assign IOBUS_OUT = rs2; // Register Out
assign IOBUS_ADDR = result; // ALU Output mapped to IOBUS_ADDR
```

CU FSM:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer: James Ratner
//
// Create Date: 01/07/2020 09:12:54 PM
// Design Name:
// Module Name: top_level
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Control Unit Template/Starter File for RISC-V OTTER
//
//     //- instantiation template
//     CU_FSM my_fsm(
//        .intr     (xxxx),
//        .clk      (xxxx),
//        .RST      (xxxx),
//        .opcode   (xxxx),   // ir[6:0]
//        .pcWrite  (xxxx),
//        .regWrite (xxxx),
//        .memWE2   (xxxx),
//        .memRDEN1 (xxxx),
//        .memRDEN2 (xxxx),
//        .reset    (xxxx)   );
//
// Dependencies:
//
// Revision:
// Revision 1.00 - File Created - 02-01-2020 (from other people's files)
//          1.01 - (02-08-2020) switched states to enum type
//          1.02 - (02-25-2020) made PS assignment blocking
//                              made rst output asynchronous
//          1.03 - (04-24-2020) added "init" state to FSM
//                              changed rst to reset
//          1.04 - (04-29-2020) removed typos to allow synthesis
//          1.05 - (10-14-2020) fixed instantiation comment (thanks AF)
//          1.06 - (12-10-2020) cleared most outputs, added commentes
//
//////////////////////////////////////////////////////////////////////////////////

module CU_FSM(
```

```
  input intr,
  input clk,
  input RST,
  input [6:0] opcode,     // ir[6:0]
  output logic pcWrite,
  output logic regWrite,
  output logic memWE2,
  output logic memRDEN1,
  output logic memRDEN2,
  output logic reset
);

  typedef  enum logic [1:0] {
     st_INIT,
        st_FET,
     st_EX,
     st_WB
  }  state_type;
  state_type  NS,PS;

  //- datatypes for RISC-V opcode types
  typedef enum logic [6:0] {
      LUI    = 7'b0110111,
      AUIPC  = 7'b0010111,
      JAL    = 7'b1101111,
      JALR   = 7'b1100111,
      BRANCH = 7'b1100011,
      LOAD   = 7'b0000011,
      STORE  = 7'b0100011,
      OP_IMM = 7'b0010011,
      OP_RG3 = 7'b0110011
  } opcode_t;
     opcode_t OPCODE;    //- symbolic names for instruction opcodes

     assign OPCODE = opcode_t'(opcode); //- Cast input as enum


     //- state registers (PS)
     always @ (posedge clk)
      if (RST == 1)
          PS <= st_INIT;
      else
          PS <= NS;
```

```
always_comb
begin
    //- schedule all outputs to avoid latch
    pcWrite = 1'b0;    regWrite = 1'b0;    reset = 1'b0;
           memWE2 = 1'b0;     memRDEN1 = 1'b0;    memRDEN2 = 1'b0;

    case (PS)

        st_INIT: //waiting state
        begin
            reset = 1'b1;
            NS = st_FET;
        end

        st_FET: //waiting state
        begin
            memRDEN1 = 1'b1;
            NS = st_EX;
        end

        st_EX: //decode + execute
        begin
            pcWrite = 1'b1;
                          case (OPCODE)
                              LOAD:
                    begin
                       regWrite = 1'b0;
                       pcWrite = 1'b0;
                       memRDEN2 = 1'b1;
                       NS = st_WB;
                    end

                                  STORE:
                    begin
                       regWrite = 1'b0;
                       memWE2 = 1'b1;
                       NS = st_FET;
                    end

                                  BRANCH:
                    begin
                       NS = st_FET;
```

```
            end

                        LUI:
                           begin
               regWrite = 1'b1;
                                 NS = st_FET;
                           end

                        OP_IMM:  // Immediate operations
                           begin
                              regWrite = 1'b1;
                              NS = st_FET;
                           end

                     OP_RG3: // (Non-immediate ALU operations)
                        begin
                           regWrite = 1'b1;
                           NS = st_FET;
                        end

            JAL:
                           begin
                              regWrite = 1'b1;
                              NS = st_FET;
                           end

         default:
                           begin
                              NS = st_FET;
                           end

     endcase
end

st_WB:
begin
   regWrite = 1'b1;
   pcWrite = 1'b1;
   NS = st_FET;
   memRDEN2 = 1'b0;
end

default: NS = st_FET;
```

```
        endcase //- case statement for FSM states
    end

endmodule
```

CU DCDR:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Ratner Surf Designs
// Engineer: James Ratner
//
// Create Date: 01/29/2019 04:56:13 PM
// Design Name:
// Module Name: CU_Decoder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// CU_DCDR my_cu_dcdr(
//   .br_eq     (),
//   .br_lt     (),
//   .br_ltu    (),
//   .opcode    (),    //-  ir[6:0]
//   .func7     (),    //-  ir[30]
//   .func3     (),    //-  ir[14:12]
//   .alu_fun   (),
//   .pcSource  (),
//   .alu_srcA  (),
//   .alu_srcB  (),
//   .rf_wr_sel ()   );
//
//
// Revision:
// Revision 1.00 - File Created (02-01-2020) - from Paul, Joseph, & Celina
//          1.01 - (02-08-2020) - removed unneeded else's; fixed assignments
//          1.02 - (02-25-2020) - made all assignments blocking
//          1.03 - (05-12-2020) - reduced func7 to one bit
//          1.04 - (05-31-2020) - removed misleading code
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////////

module CU_DCDR(
    input br_eq,
        input br_lt,
        input br_ltu,
    input [6:0] opcode,   //-  ir[6:0]
        input func7,          //-  ir[30]
    input [2:0] func3,    //-  ir[14:12]
    output logic [3:0] alu_fun,
    output logic [1:0] pcSource,
    output logic alu_srcA,
    output logic [1:0] alu_srcB,
        output logic [1:0] rf_wr_sel   );

    //- datatypes for RISC-V opcode types
    typedef enum logic [6:0] {
        LUI    = 7'b0110111,
        AUIPC  = 7'b0010111,
        JAL    = 7'b1101111,
        JALR   = 7'b1100111,
        BRANCH = 7'b1100011,
        LOAD   = 7'b0000011,
        STORE  = 7'b0100011,
        OP_IMM = 7'b0010011,
        OP_RG3 = 7'b0110011
    } opcode_t;
    opcode_t OPCODE; //- define variable of new opcode type

    assign OPCODE = opcode_t'(opcode); //- Cast input enum

    //- datatype for func3Symbols tied to values
    typedef enum logic [2:0] {
        //BRANCH labels
        BEQ = 3'b000,
        BNE = 3'b001,
        BLT = 3'b100,
        BGE = 3'b101,
```

```
    BLTU = 3'b110,
    BGEU = 3'b111
} func3_t;
func3_t FUNC3; //- define variable of new opcode type

assign FUNC3 = func3_t'(func3); //- Cast input enum

always_comb
begin
    //- schedule all values to avoid latch
           pcSource = 2'b00;  alu_srcB = 2'b00;    rf_wr_sel = 2'b00;
           alu_srcA = 1'b0;   alu_fun  = 4'b0000;

           case(OPCODE)
                  LUI:
                  begin
                      pcSource = 2'b00;
                      rf_wr_sel = 2'b11;
                   alu_srcA = 1'b1;
                   alu_srcB = 2'b00;
                   alu_fun  = 4'b1001;
                  end

                  JAL:
                  begin
                         pcSource = 2'b11;
                      rf_wr_sel = 2'b00;
                   alu_srcA = 1'b0;
                   alu_srcB = 2'b00;
                   alu_fun  = 4'b0000;
                  end

                  LOAD:
                  begin

                  pcSource = 2'b00;
                  rf_wr_sel = 2'b00;
                   alu_srcA = 1'b0;
```

```
        alu_srcB = 2'b00;
        alu_fun  = 4'b0000;
       end

       STORE:
       begin
              case(FUNC3)
                  3'b000: // SB
                  begin

                  end

                  3'b001: // SH
                  begin

                  end

                  3'b010: // SW
                  begin
                      pcSource = 2'b00;
                     rf_wr_sel = 2'b11;
                alu_srcA = 1'b0;
                alu_srcB = 2'b10;
                alu_fun  = 4'b0000;
                  end

                  default:
                  begin
                      pcSource = 2'b00;
                     rf_wr_sel = 2'b00;
                alu_srcA = 1'b0;
                alu_srcB = 2'b00;
                alu_fun  = 4'b0000;
                  end
              endcase
// INSERT GENERAL S-TYPE PARAMS HERE
       end
```

```
OP_IMM:
begin
        case(FUNC3)
                3'b000: // instr: ADDI
                begin
                        pcSource = 2'b00;
                        rf_wr_sel = 2'b11;
                        alu_srcA = 1'b0;
                        alu_srcB = 2'b01;
                        alu_fun  = 4'b0000;
                end

                default:
                begin
                        pcSource = 2'b00;
                        alu_fun = 4'b0000;
                        alu_srcA = 1'b0;
                        alu_srcB = 2'b00;
                        rf_wr_sel = 2'b00;
                end
        endcase
end

OP_RG3:
begin
    case(FUNC3)
         3'b000: // ADD & SUB
         begin
             case(func7)
                 1'b0: // ADD
                 begin
                     alu_fun = 4'b0000;
                 end

                 1'b1: // SUB
                 begin
                     alu_fun = 4'b1000;
                 end
```

```
                endcase
                pcSource = 2'b00;
                        alu_srcA = 1'b0;
                        alu_srcB = 2'b00;
                        rf_wr_sel = 2'b11;
            end

            default:
            begin
                pcSource = 2'b00;
                        alu_fun = 4'b0000;
                        alu_srcA = 1'b0;
                        alu_srcB = 2'b00;
                        rf_wr_sel = 2'b00;
            end
        endcase
    end

    default:
    begin
             pcSource = 2'b00;
             alu_srcB = 2'b00;
             rf_wr_sel = 2'b00;
             alu_srcA = 1'b0;
             alu_fun = 4'b0000;
    end
    endcase
end

endmodule
```

Hardware Problem:

Consider you're using a MCU to solve a given problem. You always have the choice of "doing stuff" using the MCU (firmware) or adding external peripherals to your circuit to "do the stuff". List the pros and cons of having the MCU do the required work vs. having external peripherals do required work.

Pros:

- The MCU is literally built to be a computer, and it's incredibly capable of handling many basic computations
- All the MCU needs to compute data are assembly instructions, and the rest is handled by the existing hardware.
- The MCU is reasonably optimized, and the design itself is pretty efficient for a number of types of computations.

Cons:

- The RISC-V ISA is pretty barebones, and operations like multiplication must be done manually with RISC-V instructions.
- Floating point computation is not possible with the MCU, and an FPU would need to be integrated to allow for that functionality which would be incredibly challenging.
- Depending on the tasks, more complicated hardware might be required in order to do certain things, and there's no guarantee the MCU could keep up with more advanced computations.

Programming Problem:

Write a RISC-V MCU assembly language subroutine that converts a 4-digit BCD value in the four lower nibbles in x11 to a binary value in the same register (x11). The BCD value will never be greater than 9999. Make sure your subroutine does not permanently change any registers other than x11. Also include a brief but complete written description of the algorithm you used in your solution. Minimize the amount of code in your solution.

This algorithm takes each of the four nibbles, multiplies each value by the given power of 10 depending on its place in the BCD and adding this value together into a single number.

```
preserve: addi sp, sp, -8
sw x20, 0(sp)
sw x22, 4(sp)
li x20, 0

thousands: mv x22, x11
srli x22, x22, 12 // Focus on thousands nibble
andi x22, x22, 0xf // Mask

loop1: beq x22, x0, hundreds // Check if thousands are done multiplying
addi x20, x20, 1000 // Increment by 1000
addi x22, x22, -1 // Decrement loop
j loop1

hundreds: mv x22, x11
srli x22, x22, 8 // Focus on hundreds nibble
andi x22, x22, 0xf // Mask

loop2: beq x22, x0, tens // Check if hundreds are done multiplying
addi x20, x20, 100 // Increment by 100
addi x22, x22, -1 // Decrement loop
j loop2

tens: mv x22, x11
srli x22, x22, 4 // Focus on tens nibble
andi x22, x22, 0xf // Mask
```

```
loop3: beq x22, x0, ones // Check if tens are done multiplying
addi x20, x20, 10 // Increment by 10
addi x22, x22, -1 // Decrement loop
j loop3

ones: mv x22, x11
andi x22, 0xf // Ones nibble
addi x20, x20, x22 // Add ones to accumulated value
mv x11, x20 // Move to x11

restore: lw x20, 0(sp) // Restore context
lw x22, 4(sp)
addi sp, sp, 8
ret
```