David J. Marion

CE412 Advanced Computer Architecture

December 13, 2019

“It’s not an ARM, it’s a LEG.”

**Overall Conception**

The initial conception of the LEG microprocessor begins with five main parts; the arithmetic and logic unit(ALU), the register set, the control unit, the data path, and the instruction set. The first part to design is the instruction set, because the designs of the other four parts are based on the codes embedded within the instruction set itself.

Once there is an initial instruction set established the designing of the ALU and register set follows. The number of bits of the opcode within the instruction set determines the maximum number of ALU operations that can be processed by the ALU. The number of bits in the instruction set allotted for a register address determines the maximum number of registers in the register set.

Once these parts have been established, a data path is designed to facilitate the transfer of data to and from the ALU, the register set, and program memory(RAM). Then a control unit is designed to control the flow of data into registers or RAM from the ALU, for instance.

The overall concept of design comes from RISC architecture and single-cycle processing. Negative clock edge triggers latching of data into registers, whereas every other module requiring synchronization operates on a positive clock edge.

**Functional Design**

The functional design of the microprocessor includes the four functional parts mentioned above and several additional components that facilitate operations. The functional design of a microprocessor is based on the desired/required functionality for the processor. The components used in the design of the LEG microprocessor are listed and briefly detailed below:

Arithmetic Logic Unit(ALU) – performs NOT, AND, OR, XOR, ADD, SUBTRACT, COMPARE functions

Shifter – performs LSL, LSR, ASR, ROTL, ROTR functions

Register Set – eight general purpose registers(GPR:R0-R7), instruction register(IR), program counter(PC), and condition code register(CCR)

Data Path – 16-bit data bus and 16-bit address bus

Multiplexers

(1) 16X8 – selects which register(R0-R7) data to put on the ‘X’ input to ALU, shifter, and data bus.

(2) 16X8 – selects which register(R0-R7) data to put on the ‘Y’ input to ALU and Shifter

(3) 16X8 - selects the ALU output

(4) 16X8 – selects the Shifter output

(5) 16X4 – select between ALU output, Shifter output, ‘X’ bus to data bus, or CCR to data bus

3X8 Decoder – to select which general purpose register to enable

Demultiplexer – select whether the instruction’s 16-bit operand is data or a memory address

Bit Extender – extend with zeros the 16-bit data from the data bus to 32-bit data memory input

Bit Extender – extend with zeros the 8-bits of the CCR to 16-bit for the data bus

Program Counter – selects the memory address of the next instruction to be executed

Memory(RAM) – 32-bit random access memory for storing instructions and data

Bit Selector – used in the shifter, pulled to a value of zero to select the least significant nibble from ‘Y’ as the amount to shift ‘X’

**Instruction Set Design**

When designing an instruction set for a microprocessor one of the first things to consider is word size. The LEG processor is a RISC processor, which means it has a reduced instruction set and is designed for reduced instruction set computers. The word size for the LEG is 32 bits, where the two high-order bytes are the 16-bit instruction, and the two low-order bytes are the 16-bit operand.

The 32 bits of the instruction set are broken as follows:

Bit(31, 30) – This bit grouping is called Family. These two bits select the type of operation the processor is to perform. LEG operation types are data moves, ALU operations, Shifter operations, and for reading the CCR.

Bit(29, 28, 27) – This group of three bits is called Opcode and they select which operation the ALU or Shifter will perform.

Bit(26, 25, 24) – This group of three bits is called Source1. They select which general purpose register data as the first ALU/Shifter input.

Bit(23, 22, 21) – This three-bit grouping is called Source2/Destination. They select which general purpose register data as the second ALU/Shifter input and also selects which register to store the ALU/Shifter output data and data move data.

Bit(20) – System enable bit. When “0”, no operation is performed.

Bit(19) – RAM control bit. When “0”, data will be stored in RAM, “1” loads from RAM to IR.

Bit(18) – Address mode bit. When “0”, the operand is the data, “1” means the operand is the memory address where the data is stored. (“0” = Immediate mode, “1” = Absolute mode)

Bit(17) – Program counter control bit. When “0”, PC <= PC + 1, “1” PC receives an operand address.

Bit(16) – Memory clear bit. Analogous to an initialization bit; clears RAM, all general purpose registers, the condition code register, the instruction register, and resets the program counter to memory address zero.

Bit(15-0) – This group represents the 16-bit operand that can be data or a memory address.

**Register Set Design, simulation/testing**

The LEG has eight general purpose registers labeled R0 thru R7. These are 16-bit registers that receive data from the data bus as input and output the data to be used in microprocessor operations. The latching of data into the registers is contingent upon a latch enable signal from the 3X8 decoder and is negative-edge triggered. The decoder receives its control signal from the Source2/Destination group of bits.

The outputs of the eight registers are sent through the two 16X8 multiplexors that relatively select the registers to represent the ‘X’ and ‘Y’ inputs for the ALU and Shifter modules. The ‘X’ bus is also available to the main data bus for the purpose of moving data from one memory location to another. The ‘X’ bus multiplexer receives its control signal from the Source1 group of bits, and the ‘Y’ bus multiplexer receives its control signal from the Source2/Destination group.

The LEG has three other registers apart from the register set. The 16-bit program counter selects the address in memory for the next program instruction. It can be set to increment by one or to an address via the address from the 16-bit operand of the instruction. The instruction register holds the 32-bit instruction that it receives from memory.

The 8-bit condition code register holds the bitwise values of the eight condition codes. The contents of the CCR can be placed in a GPR and masked for detection in order to be used in program branch/jump instructions.

**ALU Design, simulation/testing**

The LEG processes six ALU operations and five Shifter operations. The ALU and Shifter were designed as separate units which reduced the number of bits needed for the opcode.

The ALU receives two 16-bit inputs and provides one output for computing purposes. Arithmetic operations of the ALU are addition and subtraction. The LEG does not compute negative numbers, only positive numbers from 0 to 65,535. Logic operations of the ALU are !, &, |, and ^. (NOT, AND, OR, XOR). The ALU also provides the eight bits stored in the CCR.

The Shifter of the LEG is designed to receive two 16-bit inputs and also provide a plethora of outputs. Shifting operations of the Shifter are logical shift left, logical shift right, arithmetic shift right, rotate left, and rotate right. In each of the shifting operations, the ‘X’ input is shifted by the binary value amount of the ‘Y’ input. A bit selector is used to decatenate the least significant nibble of the ‘Y’ input and use that nibble as the binary value shift amount.

**Data Flows**

The data coming from program memory to the instruction register is a 32-bit word consisting of 16 bits of instruction and 16 bits of operand. The 16 bits of the operand can be either data or an address in memory where the data is stored. A demultiplexer routes the operand to the appropriate bus. If the operand is an address, the demux routes the address onto the address bus that is connected to the program counter in order to select the address where the data is located. If the operand is data, the demux routes the data onto the main data bus where it can be latched into a register or 16-bit zero extended before going back into 32-bit RAM data.

In addition to receiving data from the operand demux, the main data bus also receives data from the ‘X’ bus for data moves and the outputs from ALU and Shifter operations. The ‘X’ bus, ALU, and Shifter outputs to the main bus are multiplexed to allow only one piece of data to the main bus per cycle. This multiplexer is known as the Family mux and also allows for the condition code register to be put on the bus, after the CCR is zero extended to 16 bits.

**Control System Design, simulation/testing**

Design of the control system begins with the instruction register(IR). For each instruction the IR holds the bits needed to control the operations of the microprocessor. In the control module, the 16 bits of the instruction set are split from the 32 bit word and essentially decoded and rerouted as control signals to the other functional modules of the microprocessor. The control unit is consequently mostly designed as a result of the preceding module designs. Information about each control signal is listed below:

Bit(31, 30) – control signal to select ‘X’ bus, ALU out, or Shifter out to main data bus (family)

Bit(29, 28, 27) – control signal to select ALU and Shifter output (opcode)

Bit(26, 25, 24) and Bit(23, 22, 21) – register select control signals(source1 and source2/destination)

Bit(20) – control signal to enable processing or perform “no operation” for time delays

Bit(19) – RAM control signal to load or store

Bit(18) – address mode control signal; indicates if operand is data or address

Bit(17) – control signal to program counter; increment by 1 or receive address

Bit(16) – initialize control signal; when set clears registers, RAM, sets PC to address 0

**Overall System, simulation/testing**

The overall design of the LEG microprocessor presents a simple and practical architecture. The LEG is a single-cycle RISC processor that includes the basic components of a microprocessor and performs a basic set of arithmetic and logic operations. It is a 32 bit system that allows for 16 bits of data or address. The LEG provides two different addressing modes: immediate and absolute addressing modes tell the processor whether the 16-bit operand is data or a memory address.

With the ALU having six operations and the Shifter having five, the LEG is capable of processing a total of 11 arithmetic and logic operations. The ALU of the LEG also includes a comparator that outputs five different comparison values. Five comparison values plus the carry, negative and zero conditions gives the LEG a total of eight different condition codes that can be used in programs for branch/jump instructions.

**Conclusion**

Designing a microprocessor begins with an instruction set which is derived from the required functionality of the processor. Once an instruction set is established, the functional components are designed in accordance with the bit-width of different parts of the instruction. At a minimum, a processor needs to have a register set, an arithmetic and logic unit, and a control unit.

The purpose of a microprocessor is to take an input, process it, and provide an output. Essentially, a microprocessor is a gargantuan transfer function for processing inputs and providing the desired outputs.

As for simulation/testing of the LEG, modules of functional components were created in Verilog (see attachments), but were unable to be tested due to lack of experience and time.

*Design Tradeoffs*

The designing of the instruction set began with a word size of 16 bits but very soon grew to 32 bits when the operational constraints of 16 bits was soon realized. Having 32 bits for the word increases the functionality of the microprocessor and/or increases the size of the operand, depending on design choice/specifications.

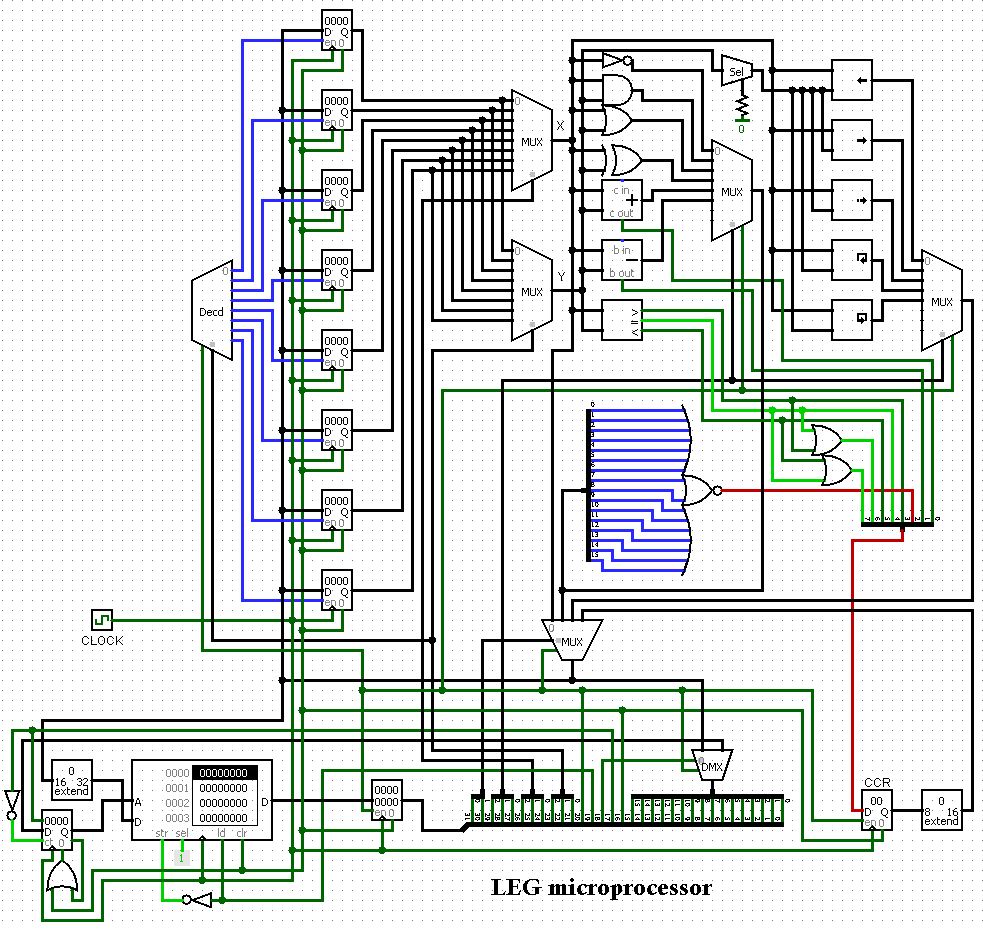
By using a separate ALU and Shifter we were able to use three bits for the opcode instead of four which freed up a much needed bit for system control.

**Epilogue**

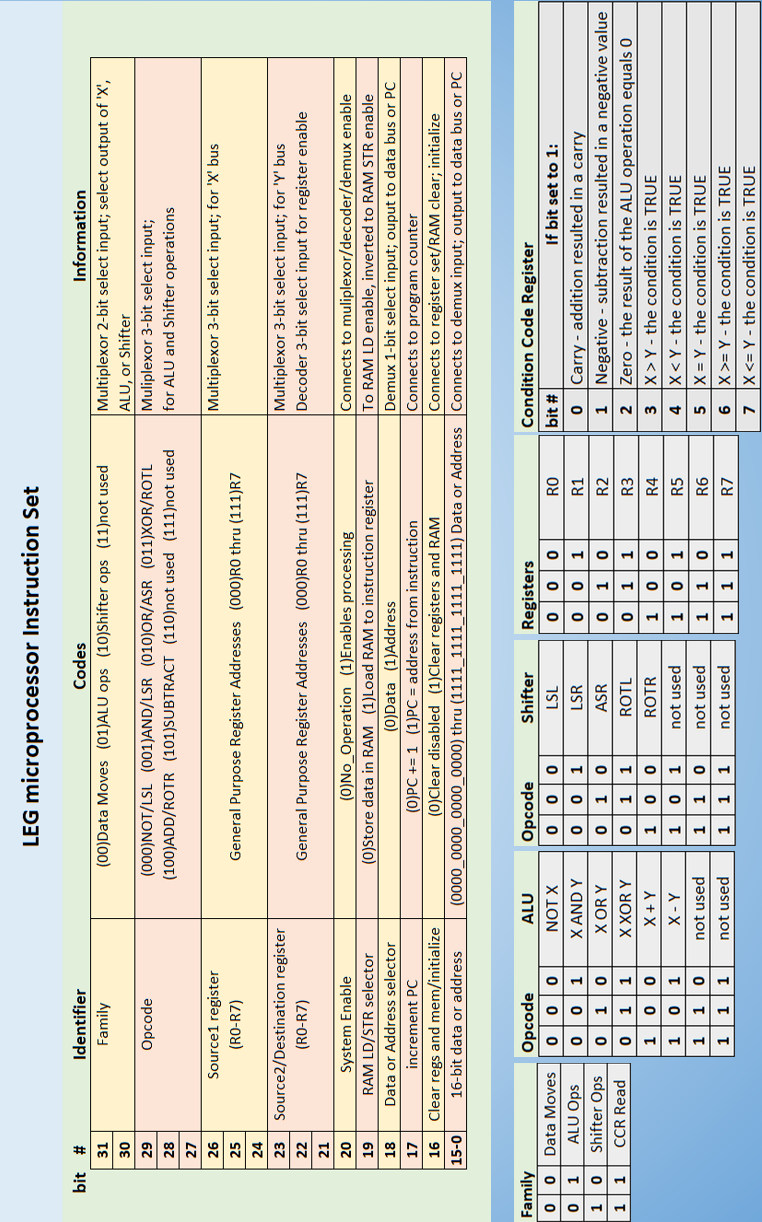
My perception of microprocessors and computer design has change greatly. Before this class, I had only an inkling of an idea of how a processor actually works. Now I feel as though I have a very good grasp of about 90% of the concepts for designing a RISC microprocessor. I’m still fuzzy on the overflow flag.

Even after this class has ended, I will continue my research on microprocessors. To me, they are one of the most amazing inventions that mankind has produced. I want to know the ins and outs of a microprocessor like the back of my hand. I will continue to learn Verilog and learn how to synthesize the code onto an FPGA and furthermore, write C code to program the processor on the FPGA. All of these skills are valuable skills in the field of computer engineering. I really appreciated the entry of Verilog into the curriculum.

I really enjoyed this class and learning about microprocessors. Two things I recommend for making this class better: having a book that will help with designing the assigned processor, and having had the time to simulate the whole processor in Verilog.



**Figure 1. – Complete design of the LEG microprocessor (created in Logisim)**



**Figure 2. – LEG instruction set**

**VERILOG Codes (only modules, no test benches)**

Some of the codes may be incomplete. Not all modules needed for the LEG were completed/attempted.

**ALU**

module alu(x, y, opcode, result, CCR);

parameter WIDTH = 16;

input [WIDTH-1:0] x; // X from register file

input [WIDTH-1:0] y; // Y from register file

input [2:0] opcode; // opcode from control unit

output reg result; // result of ALU operation

output reg [7:0] CCR; // condition code register

reg [WIDTH:0] sub\_result; // extra bit to capture flags

always @\*

begin

case

if( opcode == 3'b000 ) sub\_result <= !x; // NOT

if( opcode == 3'b001 ) sub\_result <= x & y; // AND

if( opcode == 3'b000 ) sub\_result <= x | y; // OR

if( opcode == 3'b001 ) sub\_result <= x ^ y; // XOR

if( opcode == 3'b010 ) sub\_result <= x + y; // ADD

if( opcode == 3'b011 ) sub\_result <= x - y; // SUBTRACT

endcase

CCR[0] = sub\_result[WIDTH]; // carry flag set if 1

CCR[1] = result[WIDTH-1]; // negative flag set if MSB is 1

CCR[2] = result==0 ? 1 : 0; // zero flag set if result is 16'h0000

CCR[3] = x > y ? 1 : 0; // COMPARE: 1 = TRUE

CCR[4] = x < y ? 1 : 0; // "

CCR[5] = x = y ? 1 : 0; // "

CCR[6] = x >= y ? 1 : 0; // "

CCR[7] = x <= y ? 1 : 0; // "

result = sub\_result[WIDTH-1:0]; // result of ALU operation

end

endmodule

**Bit Selector**

module bit\_selector( y, shift\_value );

input [15:0] y;

output reg [3:0] shift\_value;

always @\*

begin

output <= y[3:0];

end

endmodule

**Control Unit**

module control\_unit( instr\_reg, fam, opcode, src1\_reg, src2\_dest\_reg, sys\_enable, data\_out );

input reg [31:0] instr\_reg;

output [1:0] fam = instr\_reg[31:30]; // 2 bits for family operation

output [2:0] opcode = instr\_reg[29:27]; // 3 bits for opcode (ALU select, shifter select)

output [2:0] src1\_reg = instr\_reg[26:24]; // 3 bits select which register

output [2:0] src2\_dest\_reg = instr\_reg[23:21]; // 3 bits select which register

output [0] sys\_enable = instr\_reg[20]; // 0 = no operation, 1 = enable system ops

output [0] = instr\_reg[19]; //

output [0] = instr\_reg[18];

output [0] = instr\_reg[17];

output [0] = instr\_reg[16];

output [15:0] data\_out = instr\_reg[15:0];

endmodule

**RAM**

module ram( data\_out, data\_in, address, write, select, clock );

output[31:0] data\_out;

input [31:0] data\_in;

input [15:0] address;

input write;

input select;

input clock;

reg [31:0] mem [65535:0]; //memory

reg [31:0] data\_out;

always @ ( data\_in, address, write, select )

begin

if ( write && select )

begin

mem[address] = data\_in;

//$display( "write %8X into DM[%3X]", data\_in, address );

end

assign data\_out = select ? mem[address] : 32'hzzzzzzzz;

end

endmodule

**Register**

module register(clock, enable, data\_in, data\_out);

input clock;

input enable;

input [15:0] data\_in;

output reg [15:0] data\_out;

always @(negedge clock)

begin

if( enable )

begin

data\_out <= data\_in;

end

end

endmodule

**Register Mux**

module register\_mux( mux\_out, in0, in1, in2, in3, in4, in5, in6, in7, select);

output reg [15:0] mux\_out;

input [15:0] in0, in1, in2, in3, in4, in5, in6, in7;

input [2:0] select;

always @\*

begin

case( select)

3'b000: mux\_out = in0;

3'b001: mux\_out = in1;

3'b010: mux\_out = in2;

3'b011: mux\_out = in3;

3'b100: mux\_out = in4;

3'b101: mux\_out = in5;

3'b110: mux\_out = in6;

3'b111: mux\_out = in7;

endcase

end

endmodule

**Register Decoder**

module reg\_selector(enable, select\_reg, reg\_en,);

input enable; // enables writing to a register (vs. RAM)

input [2:0] select\_reg; // selects the register to enable

output [7:0] reg\_en;

always @\*

begin

if( enable )

begin

case( select )

3'b000: reg\_en = 8'b0000\_0001;

3'b001: reg\_en = 8'b0000\_0010;

3'b010: reg\_en = 8'b0000\_0100;

3'b011: reg\_en = 8'b0000\_1000;

3'b100: reg\_en = 8'b0001\_0000;

3'b101: reg\_en = 8'b0010\_0000;

3'b110: reg\_en = 8'b0100\_0000;

3'b111: reg\_en = 8'b1000\_0000;

endcase

end

end

endmodule

**Register Set**

`include register.v

`include register\_selector.v

`include register\_mux.v

module register\_set(clock, data\_in, which\_reg, enable, load, data\_out);

input clock;

input [15:0] data\_in;

input [2:0] x\_select;

input [2:0] y\_select;

input enable;

input load;

output [15:0] data\_out;

wire [7:0] register\_select;

wire [15:0] R0\_out, R1\_out, R2\_out, R3\_out, R4\_out, R5\_out, R6\_out, R7\_out;

register\_selector reg\_sel( register\_select, which\_reg );

register R0(.clock(clock), .enable(register\_select[0]), .data\_in(data\_in), .data\_out(R0\_out));

register R1(.clock(clock), .enable(register\_select[1]), .data\_in(data\_in), .data\_out(R1\_out));

register R2(.clock(clock), .enable(register\_select[2]), .data\_in(data\_in), .data\_out(R2\_out));

register R3(.clock(clock), .enable(register\_select[3]), .data\_in(data\_in), .data\_out(R3\_out));

register R4(.clock(clock), .enable(register\_select[4]), .data\_in(data\_in), .data\_out(R4\_out));

register R5(.clock(clock), .enable(register\_select[5]), .data\_in(data\_in), .data\_out(R5\_out));

register R6(.clock(clock), .enable(register\_select[6]), .data\_in(data\_in), .data\_out(R6\_out));

register R7(.clock(clock), .enable(register\_select[7]), .data\_in(data\_in), .data\_out(R7\_out));

register\_mux src1( x, R0\_out, R1\_out, R2\_out, R3\_out, R4\_out, R5\_out, R6\_out, R7\_out, x\_select );

register\_mux src2( y, R0\_out, R1\_out, R2\_out, R3\_out, R4\_out, R5\_out, R6\_out, R7\_out, y\_select );

endmodule

**Shifter**

`include bit\_selector.v

module shifter( shifter\_select, shifter\_enable, x\_input, y\_input, shifter\_output );

input [2:0] shifter\_select;

input shifter\_enable;

input [15:0] x\_input;

input [3:0] y\_input;

output [15:0] shifter\_output;

always @\*

begin

if( shifter\_enable )

begin

case( shifter\_select )

3'b000: shifter\_output = x\_input >> y\_input; //LSR shifter\_output=x\_input << y\_input;

3'b001: shifter\_output = x\_input << y\_input; //LSL shifter\_output=x\_input >> y\_input;

3'b010: shifter\_output = {15, x\_input[15:1]}; //ASR shifter\_output=x\_input >>> y\_input;

3'b011: shifter\_output = {x\_input[14:0], x\_input[15]}; //ROTL shifter\_output=x\_input rot> y\_input;

3'b100: shifter\_output = {x\_input[0], x\_input[15:1]}; //ROTR shifter\_output=x\_input <rot y\_input;

endcase

end

end

endmodule