EXTRA Project

CS 147-05

David-Eric Thorpe
Computer Science
San Jose State University
San Jose, CA USA
DavidEric.Thorpe@SJSU.edu

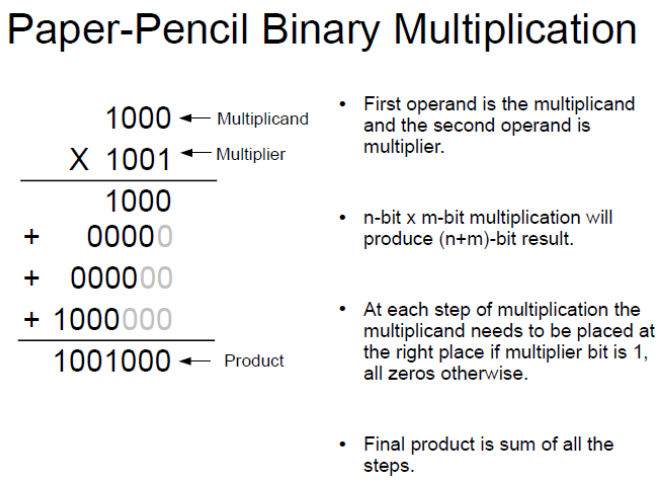
***Objective*— In previous projects we have implemented computer designs geared toward efficiency of time, while other resources like gate complexity, have been let grow unchecked. These implementations focus on implementations that allow for less complexity at the cost of time.**

I. Sequential MultiplieR

A. *Requirment*

When we learn to multiply large number we are taught to do it in a series of small steps. We literally separate the problem into easy to calculate components and then sum them. This is sequential multiplication.

Selecting the least significant digit we first multiply that across the multiplicand and shift that by the position of the LSD. Proceeding to the next LSD we perform the same steps until all digits are multiplied. Finally we sum this list of products and we have a full product. Binary is similar, except that each single multiplication is equivalent to an AND statement. Since 1&&1=1 0&&0=0 1&&0=0 0&&1=0



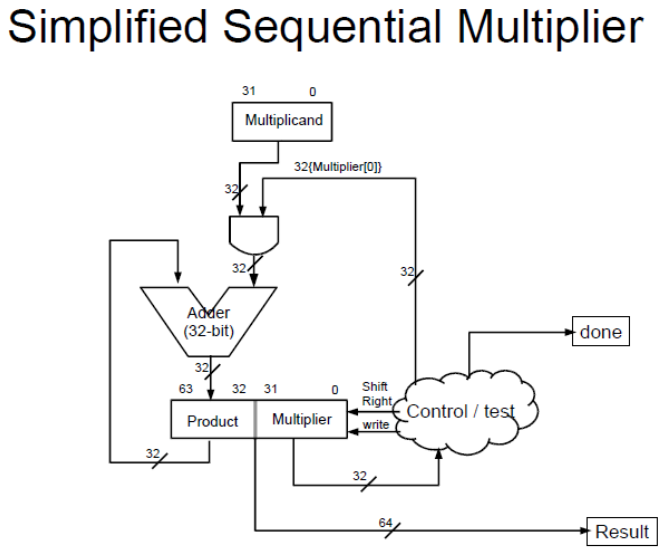
For many cases this is not an efficient implementation of a multiplier because it takes as many cycles to complete as digits in the multiplier, where as a combinational multiplier happens in one cycle.

But in situations of limited complexity where gates or wires or space may need be conserved this could be a more favourable implementation.

B. *Design & Implimentation*

Our implementation follows the Simplified Sequential model.

Performing each step and then taking a control signal for shift and done. It utilizes structures that will not be explained here: the 32-bit and gate and 32-bit adder.



Initially the result product/multiplier register is populated with data:

```
initial shiftNum = {6'b000001};
initial result = {{64'd0000},multiplier};

always @(load) begin
        shiftNum = {6'b000001};
        result = {{64'd0000},multiplier};
end
```

A load signal is also sent allowing for a degree of redundancy. This performs the same operation but allows future reinitialization.

Below we see initialization of the control module and the clock signals.

```
initial bit = 1'b0;
initial load = 1'b1;
always @( posedge clk )
        if(ready) begin
                bit = 6'b100000;
                load = 1'b0;
                write = 1'b0;
                shift = 1'b0;
        end else begin
                write = 1'b1;
                #1 shift = 1'b1;
                write = 1'b0;
                #1 shift = 1'b0;
                bit = bit - 1'b1;
        end
```

The if statement provides a conuntdown timer. Ready will only be true is the bit pattern of bit is 0.

The system then proceeds through the else cyles paced by the system clock. Write signals will be set and shift signals shortly after, within the same clock cycle. This insures the operations happen sequentally.

Via the seq_mul module

```
always @(posedge write) begin
        result[63:32]=adderOut;
end
always @(posedge shift) begin
        result=shifterOut;
end
```

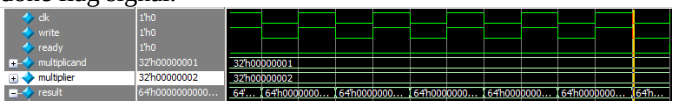
This moves data into and out of the the register and shifts it. Because a clock cycle has a high and low state the result is calculated in the high state and the sum is shifted in the low state.
This avoids conflicts in operation.
After 32 cycles the countdown will complete and the ready signal will be issued.

C. *Test and Simulation*

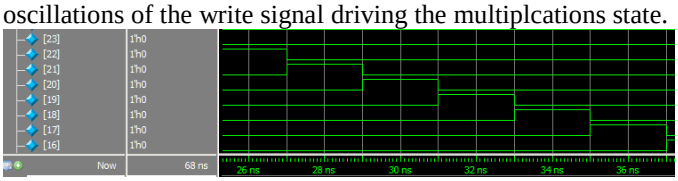
The test bed has two primary rolls. As usual it tests the module and in this case assumes the roll of control unit. It will sent the multiplicand and multiplier. Loop the circuit 32 times and rest the result.

```
initial begin
        multiplier = 32'b00010;
        multiplicand = 32'b0001;
        #1 clk = 1'b0; #1 clk = 1'b1;
        #1 clk = 1'b0; #1 clk = 1'b1;
        #1 clk = 1'b0; #1 clk = 1'b1;
        #1 clk = 1'b0; #1 clk = 1'b1;
        #1 clk = 1'b0; #1 clk = 1'b1;
        #1 clk = 1'b0; #1 clk = 1'b1;
```

… ect…
Validation is done via wave form. After a complete operation we would expect to see result contain the multiplication of the two components. We would expect to see a stepped series of shifts within the result over time. We would expect to see the done flag signal.



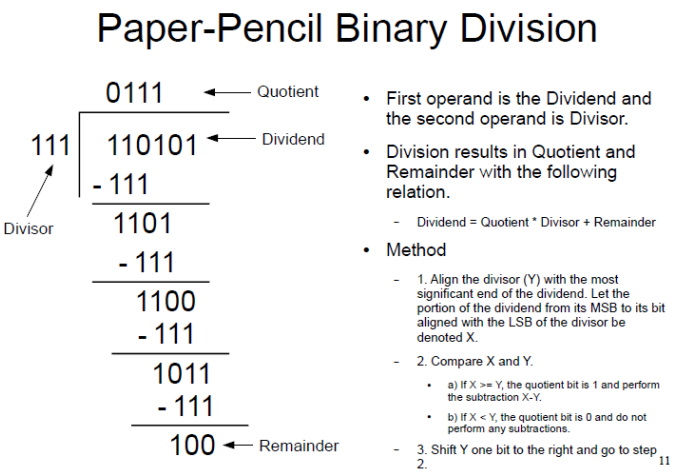
Here we see the ocilations of the clock which drive the oscillations of the write signal driving the multiplcations state.



We see here a bit being pushed through the result bit array. Every full clock cyle this bit is pushed closer to its final possition.
This testing was performed using different hardcoded values

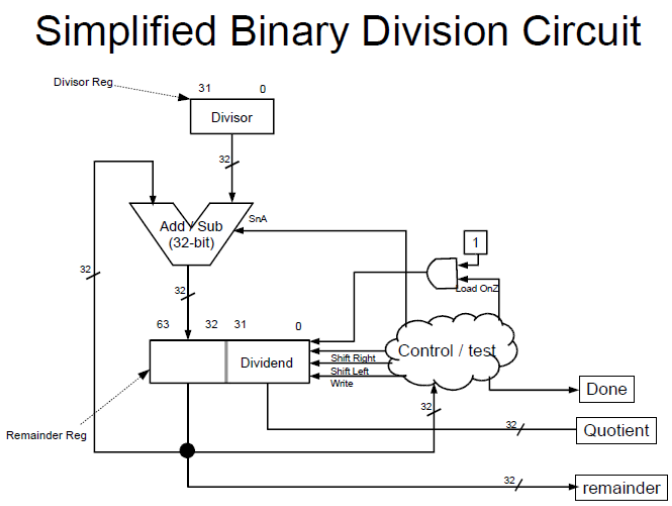
II. Sequential Div

A. *Design*



Our pen and paper version of division is much like multiplication in that we break it into component parts. In this case we work from most significant digit to least and we work with the full divisor against the MSD dividend. The result is shifted to occupy the placement of the dividend it was manipulated from.
Again, like the multiplier this has the draw backs in time and strength in simplicity.

B. *Implimentation*



This was implimented similarly to the seq_mul. The results from the subtractor's MSD were digit were tested to see if 1, meaning they were negative by 2's compliment. If so the operation of the subtraction was reversed added.

During the next low cycle the results were shifted, via control signal, so that once more the result could be returned to the adder with the divisor as the other operand.

C. *Testing*

This module was not completed adequately to allow for testing.

Testing would have proceeded by running 32 cycles of the clock and resting if the result matched expected results and if the done flag had been thrown.

III. Sequental Shift

A. *Requirement*

A shifter moves bits to in a significant or less significant direction. This shifter is designed to minimize the number of gates used via trade of for clock cycle, like previous sequential items.

Theo Jansen builds "Kinetic Sculptures" called Animaris.

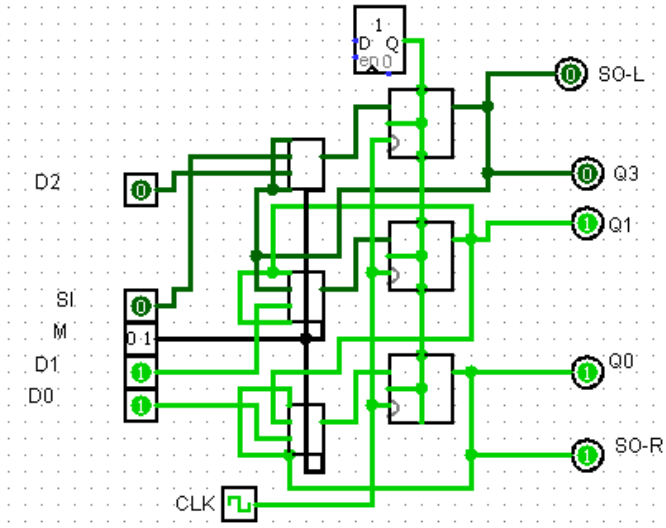
These objects move about a beach eviroment restricted by water and hills. Under gentle wind they can move freely and store air pressure for movement without wind. They also can "batten down" for weathering storms.

These stimulus responses might require a computer system and memory, which uses vacuum for bit states. As we may imagine complexity is very much a restriction here, where a gate might take up a few cm. When needing the result to choose a course of action, many cycles is prefered to the weight and volume of many gates.

B. *Design*

Our shifter runs a variable number of times, depending on the input.

Our shifter is implemented using the register shift method. Wherein one register drives the input to the next. These 32 registers are controlled by 32 multiplexers which allow for load, hold, shift left and shift right (by one bit).

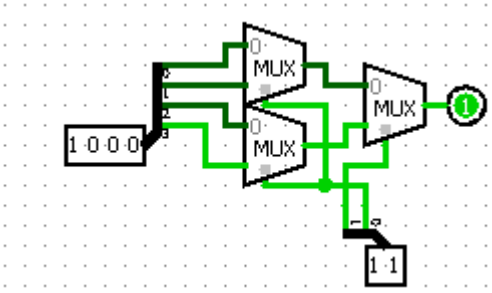


See attached file for 3 bit shifting example.

The 32-bit version uses 32 4x1 mux and 32 single bit registers.

The control unit, not shown sends the direciton signals, loads the data and takes the result. It will run the number of times to complete the circuit and resturn the completed results of the shift.

The 4x1 1-bit multiplex behaves much like its 32-bit brethren described elsewhere. Using 3 1-bit mux.



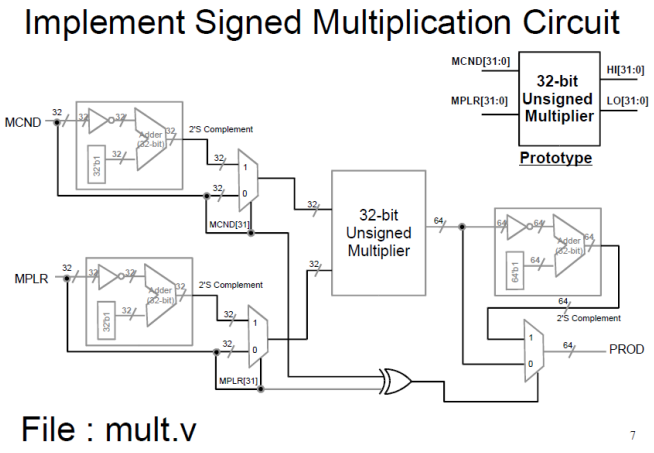
This brings the total mux count for this shifter to a little over 3*32= 96. While the total for a 1 clock barrel shifter is a little

over 2*(32*5)=320. 30% gate reduction at the cost of x32 time increase.

The control bit for the mux are the same for the shifter, thus can be passed through

```
genvar i;
generate
        for (i=0; i<=31; i=i+1)
        begin : input_mux_gen_loop_
                if(i!=0 && i!=31)
                        mux_4x1 mux(result[i], operand[i], operand[i+1], operand[i-1], operand[i], control);
                if(i==31)
                        mux_4x1 mux(result[i], operand[i], 1'b0, operand[i-1], operand[i], control);
                else
                        mux_4x1 mux(result[i], operand[i], operand[i+1], 1'b0, operand[i], control);
        end
endgenerate
```

The entire process was wrapped in a modified version of the signed muliplication wrapped used for comb_mul. But this version passed through the signals for ready, load, and shift.



C. *Testing*

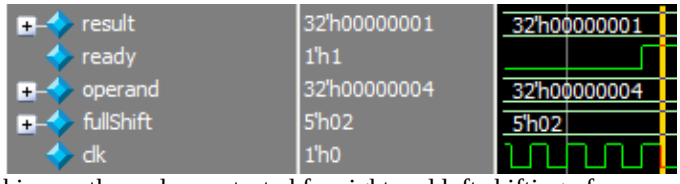
Testing of this was implemented much like the testing of other modules. Uniquely here I implemented an instance of the clock module which required a deviation from normal testing as it will run indefinitely without a stop signal.

```
initial begin
        operand = 32'b00100;
        leftNotRight = 1'b0;
        fullShift = 5'b00010;
        #100;
        $stop;
end
```

To stop the code my testbench needed to end with $stop;

As before restults were not printed to the screen but were observed in wave form.

We see here the ready flag is thrown and the result of the above operation on the above operatand of two shifts has returned 1.



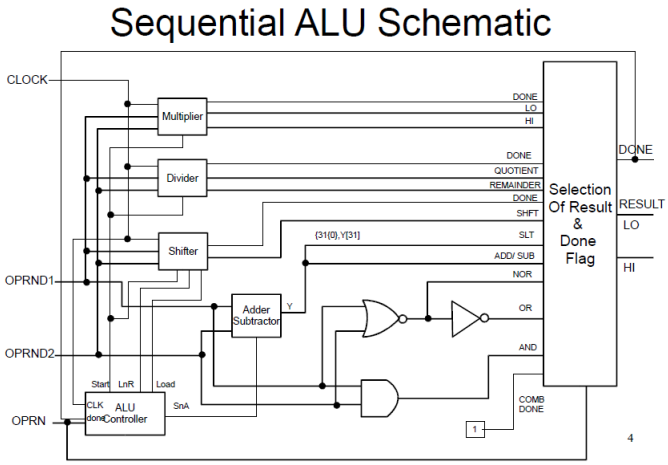
In this was the code was tested for right and left shifting of various amounts.

IV. ALU

A. *Requirments*

The ALU is the component that drives the mathematics. Recent designs have relied on the fact that any operation will be completed in 1 clock cycle but in sequential mathematics the fast completion time is scarified for fewer gates. Our implementation though does not optimize for shorter operations, which is to say every operation, no matter how trivial, is considered to take 32 clock cycles. This allows for simple prediction. Implementation otherwise would be quite challenging

B. *Implimentation*



For this implementation the ALU needs to be adjusted to for the new modules input and output.

```
        mulControl mul(.result[0:31](mulWire), readyMul, OP1, OP2, clk);
//old   mult mult_32Bit_Signed(highWire, mulWire, OP1, OP2);
```

The notable new component is the ready signal.

```
        shiftControl sc(shftWire, readyShift, OP1, OP2, lnr, clk);
//barrel_shifter b1(shftWire, OP1, OP2, lnr);
```

This is handled by a new mux that also operates on the operand like the old. The allows other components to “check” and see if the result is actually ready. I have not constructed a 1bit 16x1 but under use the 32-bit one.

```
mux32_16x1 doneMux(muxWire, 32'bZ, 1'b1, 1'b1, readyMul,
        readyShift, readyShift, 1'b1, 1'b1, 1'b1, readyDiv,
        32'bZ, 32'bZ, 32'bZ, 32'bZ, 32'bZ, 32'bZ, OPRN[3:0]);

mux32_16x1 resultMux(muxWire, 32'bZ, addSubWire, addSubWire, mulWire,
        shftWire, shftWire, andWire, orWire, norWire, divWire,
        32'bZ, 32'bZ, 32'bZ, 32'bZ, 32'bZ, 32'bZ, OPRN[3:0]);
```

Everything beside the mul, div and shift are expected to take be imiediately abailable

Each new module will need to have its own distinct signal as they will otherwise conflict.

The division was not a part of the 'cs147sec05' Instruction Set for which this hardware is designed. Future revisions of the laguage may assign the available function 0x28 to this and the mux32_16x1 has been so designed to allow for this.

```
divControl di(divWire, readyDiv, OP1, OP2, clk);

mux32_16x1 m1(muxWire, addSubWire, addSubWire, addSubWire, mulWire,
        shftWire, shftWire, andWire, orWire,
        norWire, divWire, nullWire, nullWire,
        nullWire, nullWire, nullWire,
        nullWire, OPRN[3:0]);
```

Other operations are not expected to be used but have been buffered to a null wire for erroneous results to erroneous instructions.

C. *Testing*

Because the Seq_div was not complete for this module testing for this utilized a behavioral model. DivFast. This is a crude module that simply uses higher level functions to implement the expected result.

```
module divFast(divWire, readyDiv, OP1, OP2, CLK);
        output [31:0] divWire;
        output readyDiv;
        reg [31:0] divWire;
        reg readyDiv;
        input OP1;
        input OP2;
        input CLK;
        always @(posedge CLK) begin
                divWire = OP1/OP2;
                readyDiv = 1'b1;
        end
endmodule
```

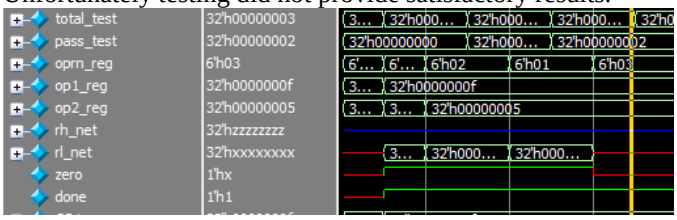
Testing of the ALU involved a modified version of the a_alut_tb used in alternate projects.
This testbed now had to account for the delay in seq_mul, seq_shift, and the ready flags.
The instantiation for instance now looked like this:

```
// Instantiation of ALU
ALU ALU_INST_01(rh_net, rl_net, done, zero, op1_reg, op2_reg, oprn_reg);
```

As before the high results though were disgarded.
A while loop was used to delay checking of the results until the done flag was thrown.

```
while(!done) begin
#1;
end
```

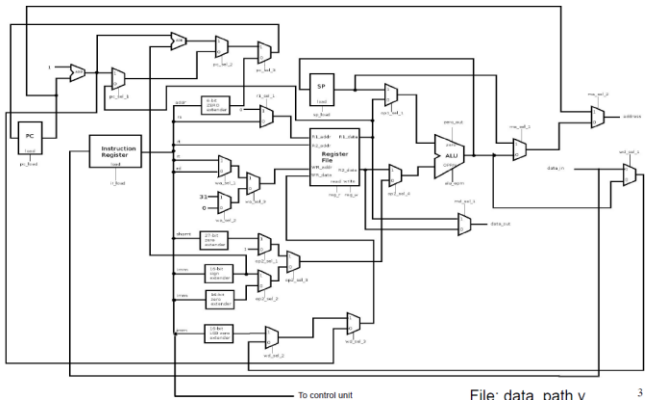
Unfortunately testing did not provide satisfactory results.



This is perhaps due to the early state of the done flag, but there was not time for futher revision.

V. Data Path

The data path routes signals during system operation. Via control signals sent from the Control Unit. This is the best representation of the "state' machine as the control unit drives the machine state forward.



A. *Implimentation*

Our previous datapath needs only additionally be able to route signals to the control unit to inform it that the ALU operation has been completed.

```
module DATA_PATH(DATA_OUT, ADDR, ZERO, INSTRUCTION, DATA_IN, CTRL, CLK, RST, ready);

        // output list
        output [`ADDRESS_INDEX_LIMIT:0] ADDR;
        output ZERO;
        output [`DATA_INDEX_LIMIT:0] DATA_OUT, INSTRUCTION;
        output ready;
```

This of course is pushed up from the ALU.

```
ALU alu(aw, ZERO, ready, op1_sel_1w, op2_sel_4w, CTRL[20:15]);
```

In our Da Vinci system implimenation this data will need to pass through the processor.

```
// instantiation section
// Control unit      (CTRL, READ, WRITE, ZERO, INSTRUCTION, CLK, RST)
CONTROL_UNIT cu_inst (CTRL, READ, WRITE, ZERO, INSTRUCTION, CLK, RST, ready);

// data path            (DATA_OUT, ADDR, ZERO, INSTRUCTION, DATA_IN, CTRL, CLK, RST)
DATA_PATH data_path_inst (DATA_OUT, ADDR, ZERO, INSTRUCTION, DATA_IN, CTRL, CLK, RST, ready);
```

Requiring a new wire

```
// net section
wire ZERO, ready;
```

And into the control unit

```
module CONTROL_UNIT(CTRL, READ, WRITE, ZERO, INSTRUCTION, CLK, RST, ready);
```

With the proper declaration

```
// input signals
input ZERO, CLK, RST, ready;
```

Finnaly the control unit must be adjusted to allow for these new conditions.

The div instruction must be added to case the EXE this is similar to the instruction in the alu testbench, that holds the machine state until the done flag is signaled

```
while(!done) begin
#1;
end
```

B. *Testing*

Because we did not have a working alu the datapath could not be tested. Testing would have involved use of custom

instructions to test the memory output for operations involving mul, div, and shift.