Erich Viebrock

ECPE 173

**Lab Project 4**

Lab Goals:

For Lab #4, we will be adding to the functionality of the processor we began creating in Lab #3. Upon completion of this lab, our processor will include support for a program counter, the register file, and control, as well as the features designed in Lab #3.

Beta vs. Standard MIPS Processor:

Basically, the Beta processor supports fewer instructions. The most significant missing functionality is jumping. Without the ability to jump, we cannot do jumps, branch if equal, branch if not equal, and a whole host of other jump instructions.

Program Counter:

In order to create the program counter, the ALU from Lab #3 is used. We import the instruction, and then add 4 to it. However, if the control signal reset is selected, we reset the instruction memory to 0x0. This is done by creating 32 2 to 1 multiplexers.

.subckt pc clk reset ia[31:0]

Xxor\_b 0#29 VDD 0#2 ia[31:0] xorOUT\_b[31:0] xor2

Xxor\_c xorOUT\_b[31:0] Cout[30:0] 0 OUT[31:0] xor2

Xand\_a xorOUT\_b[31:0] Cout[30:0] 0 andOUT\_a[31:0] and2

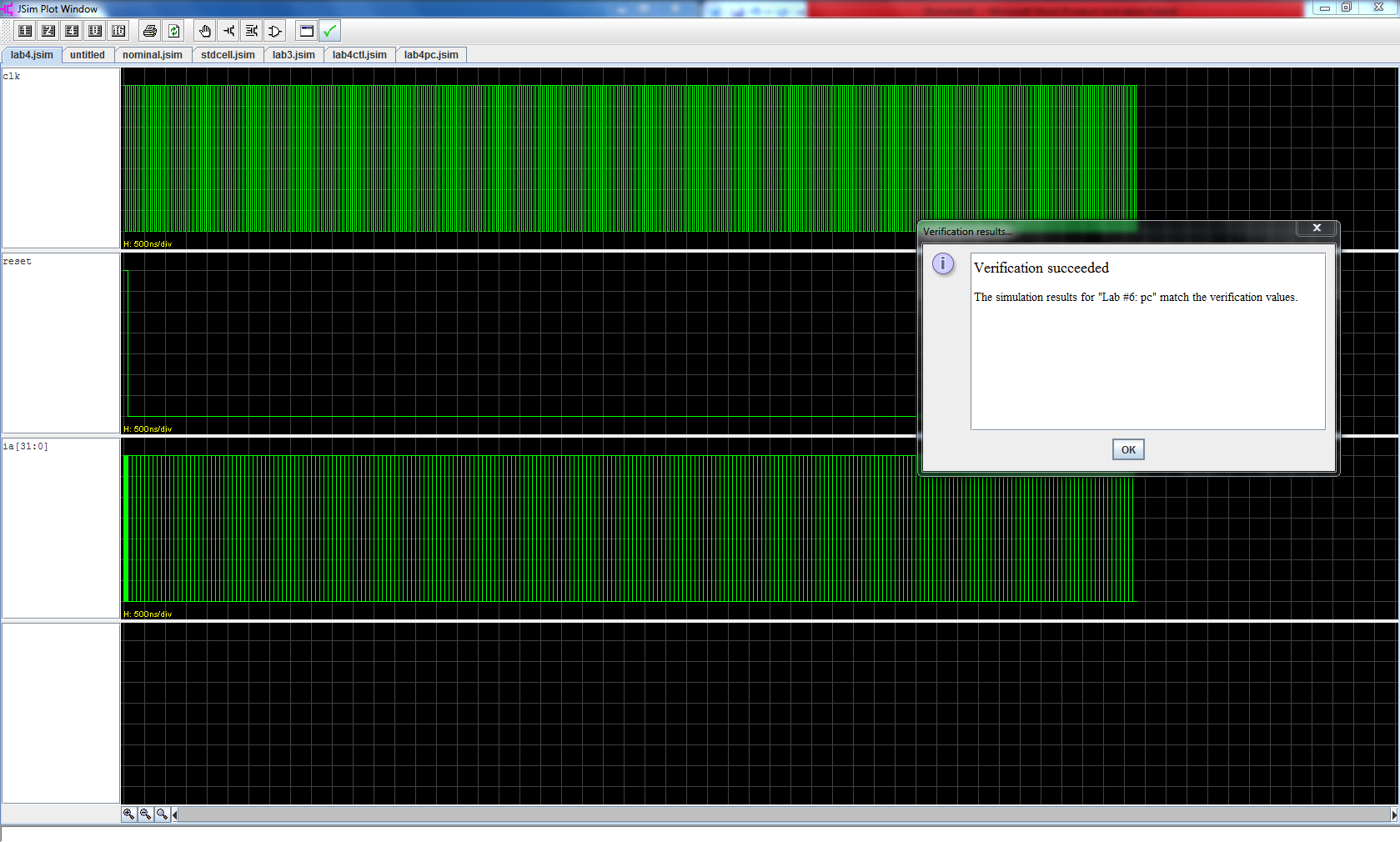
Xand\_b 0#29 VDD 0#2 ia[31:0] andOUT\_b[31:0] and2

Xor andOUT\_a[31:0] andOUT\_b[31:0] Cout[31:0] or2

Xpc\_a reset#32 OUT[31:0] 0#32 ia\_tmp[31:0] mux2

Xpc\_b ia\_tmp[31:0] CLK#32 ia[31:0] dreg

.ends



Register File:

To create the register file, there are 3 signals: ra, rb, and rc. However, because there are only 2 inputs, A mux needs to be used to select between rb and rc, with ra2sel used as the select. Also, the read timing for the $zero register can be decreased by a hardwiring the data of a $zero register. Because the $zero register is a constant, 0, we can directly map the data 0 to the $zero register address.

.subckt regfile clk werf ra2sel ra[4:0] rb[4:0] rc[4:0]

+ wdata[31:0] radata[31:0] rbdata[31:0]

Xregfile

+ vdd 0 0 ra[4:0] adata[31:0] // A read port

+ vdd 0 0 ra2mux[4:0] bdata[31:0] // B read port

+ 0 clk werf rc[4:0] wdata[31:0] // write port

+ $memory width=32 nlocations=32

Xrb\_mux ra2sel#5 rb[4:0] rc[4:0] ra2mux[4:0] mux2

Xzilch\_and\_a ra[3:0] zilch\_OUT\_a and4

Xzilch\_and\_b zilch\_OUT\_a ra[4] zilch\_OUT\_b and2

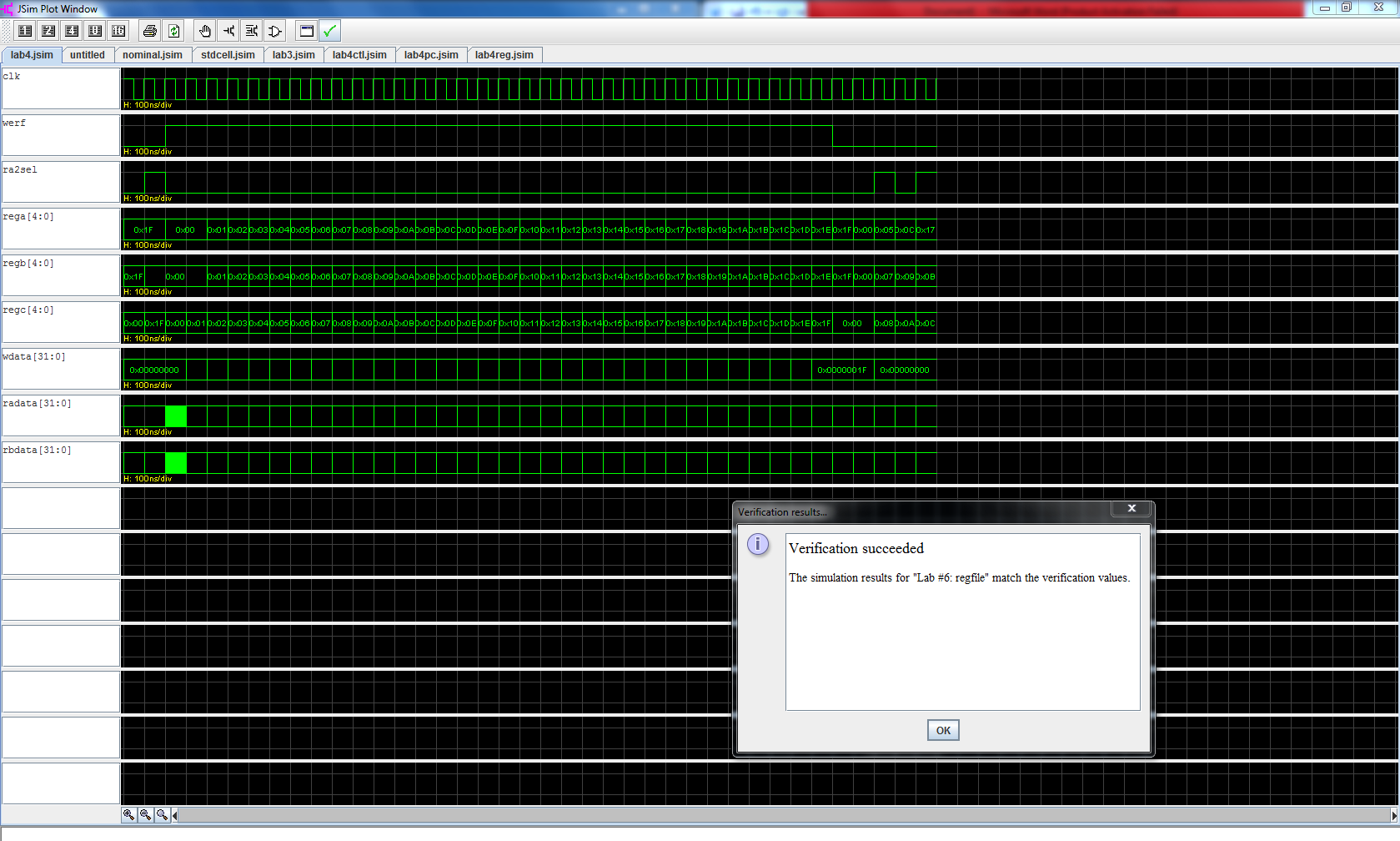
Xzilch\_mux zilch\_OUT\_b#32 adata[31:0] 0#32 radata[31:0] mux2

Xzilch\_and\_ba ra2mux[3:0] zilch\_OUT\_ba and4

Xzilch\_and\_bb zilch\_OUT\_ba ra2mux[4] zilch\_OUT\_bb and2

Xzilch\_muxb zilch\_OUT\_bb#32 bdata[31:0] 0#32 rbdata[31:0] mux2

.ends



Control:

For control, a ROM is much easier to implement than logic gates. By using a ROM, a truth table can be created, showing which control bits are high for a respective op code. Once all op codes have the desired outputs, a “moe” signal was created to be active when data is read from memory. To achieve this, the op code corresponding to a load data instruction was sent through a series of xor and and gates, which would all make a 4 input and gate true if the op code corresponded to a load data instruction. The output of the 4 input and gate was mapped to the moe output.

.subckt ctl reset id[31:26] ra2sel bsel alufn[4:0] wdsel[1:0] werf

+ moe wr

Xrom VDD 0 0 id[31:26]

+ pcsel[2:0] ra2sel asel bsel wdsel[1:0] alufn[4:0] wr werf wasel

+ $memory width=16 nlocations=64 contents=

+ (

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000011000000010 // 011 000

+ 0b0001011000000100 // 011 001

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000100000010 // 100 000

+ 0b0000000100001010 // 100 001

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000000100101010 // 100 100

+ 0b0000000100111010 // 100 101

+ 0b0000000101101010 // 100 110

+ 0b0000000000000000

+ 0b0000000111000010 // 101 000

+ 0b0000000111110010 // 101 001

+ 0b0000000110110010 // 101 010

+ 0b0000000111001010 // 101 011

+ 0b0000000101000010 // 101 100

+ 0b0000000101001010 // 101 101

+ 0b0000000101011010 // 101 110

+ 0b0000000000000000

+ 0b0000010100000010 // 110 000

+ 0b0000010100001010 // 110 001

+ 0b0000000000000000

+ 0b0000000000000000

+ 0b0000010100101010 // 110 100

+ 0b0000010100111010 // 110 101

+ 0b0000010101101010 // 110 110

+ 0b0000000000000000

+ 0b0000010111000010 // 111 000

+ 0b0000010111110010 // 111 001

+ 0b0000010110110010 // 111 010

+ 0b0000010111001010 // 111 011

+ 0b0000010101000010 // 111 100

+ 0b0000010101001010 // 111 101

+ 0b0000010101011010 // 111 110

+ 0b0000000000000000

+ )

Xxor\_moe\_a id[31] VDD moeOUT\_a xor2

Xxor\_moe\_b id[30] 0 moeOUT\_b xor2

Xxor\_moe\_c id[29] 0 moeOUT\_c xor2

Xxor\_moe\_d id[28] VDD moeOUT\_d xor2

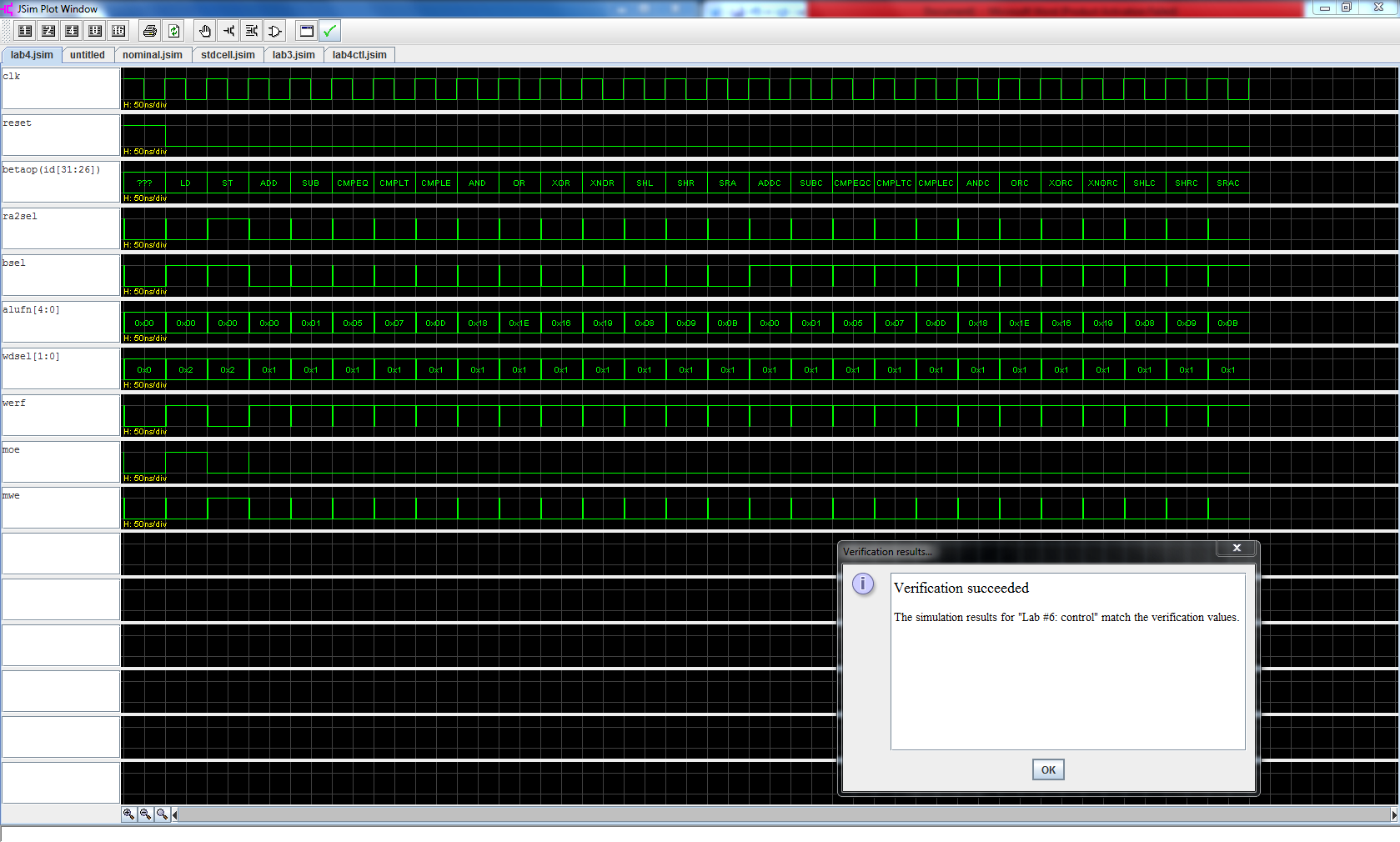
Xxor\_moe\_e id[27] VDD moeOUT\_e xor2

Xxor\_moe\_f id[26] VDD moeOUT\_f xor2

Xmoe\_a moeOUT\_a moeOUT\_b moeOUT\_c moeOUT\_d moeOUT\_g and4

Xmoe\_b moeOUT\_g moeOUT\_e moeOUT\_f VDD moe and4

.ends



Beta:

Unfortunately, I was unable to tie all my components together. I tied the program counter and register file together, but could not include the control logic. My error resided in the “ma” register, which I used a buffer with my ALU output.

.subckt beta clk reset ia[31:0] id[31:0] ma[31:0] moe mrd[31:0] wr

+ mwd[31:0]

\* PC \*

Xpc clk reset ia[31:0] pc

\* CONTROL \*

Xctl reset id[31:26] ra2sel bsel alufn[4:0] wdsel[1:0] werf

+ moe wr ctl

\* REG FILE \*

Xbuffer\_a id[15:0] id\_dummy[15:0] buffer

Xbuffer\_b id[15]#16 id\_dummy[31:16] buffer

Xmux\_a bsel#32 id\_dummy[31:0] rbdata[31:0] b[31:0] mux2

Xalu alufn[4:0] radata[31:0] b[31:0] out[31:0] z v n alu

Xmux\_b wdsel[0]#32 wdsel[1]#32 0#32 out[31:0] mrd[31:0] 0#32 wdata[31:0] mux4

Xregfile clk werf ra2sel id[20:16] id[15:11] id[25:21]

+ wdata[31:0] radata[31:0] rbdata[31:0] regfile

\* RANDOM \*

Xbuffer\_c rbdata[31:0] mwd[31:0] buffer

Xbuffer\_d out[31:0] ma[31:0] buffer

.ends

