**1) Introduction**

The goal of the proposed ISA and processor design was to quickly and efficiently run two specific programs. The first of these programs was to take a number saved in the RAM, square the number without using multiplication, and save the result back into RAM. The second program was to take an array of 8-bit numbers, calculate the bit-width of each number, and save the number, location, and width of the widest number to RAM. If there were multiple numbers that were widest, then only the information of the first widest number would be saved.

With my ISA design, I was able to achieve the following results. Calculation of the square of a number N uses just about ((N-1)\*4) + 7 clock cycles. This was found by the fact that the main loop will only repeat (N-1) times since R3 is initialized with N, taking out a single iteration of the loop. Outside of the loop, there are 7 instructions that have to run, no matter the input. Locating the widest byte in an array can range from nearly 13 clock cycles up to (31 \* 6) + (4 \* 8) + 7 clock cycles. 13 clock cycles refers to if the first or second number in the array has a bit width of 8, which causes an early termination of the program. The equation is formed by the main loop, which does not store and will run for 31 numbers if none of the first 31numbers have a bit width of 8, though the last number could be any width. Then there is the storage sub-routine which will only run if there is a larger width found; This equation assumes that the first number is 0, and the remaining bit widths are found in order somewhere in the array, though only the last number will have a bit width of 8. There are 7 instructions that run regardless. The first number is checked within these 7 instructions.

Throughout the design process, I found several flaws. Originally this was limited to branches, which did not take into account the auto-incrementing nature of the Program Counter (PC). After this, in the design of the processor, I found several mistakes in the logic for the Instruction Decoder (ID). These were then fixed and re-optimized. More recently I discovered a lack of support for an input of 0 or 1 in the Square program. To fix this, I added a new branch to the ISA that would check the initial input of the program and would branch if the input were less than 2. The processor and code was then updated to reflect this change. Finally, it occurred to me that I only needed one init command, instead of the original four. Besides R4, all the other registers are loaded with a value from memory or the ALU before they are read from.

**2) ISA Review**

The Simple Architecture System ISA (SAS ISA) was designed to be simple yet effective. The ISA was built from the ground up to support the two programs the processor was meant to run. For this reason, this ISA, and in turn this processor has very limited use outside of these programs. None of the instructions have any modifiers; For example, the programmer cannot choose what register or memory location they would like to use for a certain instruction. This keeps true to the ideal of the ISA, as programming and assembling are now completely straightforward due to the lack of mutability. Each instruction has only one possible instruction byte.

SAS ISA Instructions:

|  |  |  |
| --- | --- | --- |
| Instructions | Byte Code | Description |
| HALT | 00 000 000 | Halts program execution by leaving the PC the same. |
| BRA | 01 000 001 | PC = PC – 10 Unconditional Branch |
| BEQ | 01 000 010 | PC = PC + 1 If R1 = 63 or R4 = 8 |
| BGT | 01 000 100 | PC = PC + 4 If R4 >= R3 |
| BR1 | 01 001 000 | PC = PC – 4 Unless R1 = 1 |
| BL2 | 01 010 000 | PC = PC + 4 If R1 < 2 |
| ADD | 10 000 000 | Adds R2 to R3, stores to R3 |
| ADDS | 10 000 001 | Adds Carry Flag to R4, stores to R4 |
| INC | 10 001 000 | Increments R1, stores to R1 |
| DEC | 10 001 001 | Decrements R1, stores to R1 |
| WDT | 10 010 000 | Calculates Bit Width of R2, stores to R3 |
| INITR4 | 10 100 000 | Initializes R4 to 0 |
| LD1 | 11 000 001 | Loads R1 with memory location 0 |
| LD2 | 11 000 010 | Loads R2 with memory location 0 |
| LD3 | 11 000 011 | Loads R3 with memory location 0 |
| LD4 | 11 000 100 | Loads R1 with 32 |
| LD5 | 11 000 101 | Loads R2 with memory location [R1] |
| LD6 | 11 000 110 | Loads R4 with R3 |
| ST1 | 11 100 001 | Stores R4 to memory location 1 |
| ST2 | 11 100 010 | Stores R3 to memory location 2 |
| ST3 | 11 100 011 | Stores R3 to memory location 3 |
| ST4 | 11 100 100 | Stores R2 to memory location 4 |
| ST5 | 11 100 101 | Stores R1 to memory location 5 |

Assembly Program:

1 INITR4 – This is the only register used without being loaded before hand

2 LD1 – Load the number to be squared into R1, which is the counter

3 LD2 – Load the number to be squared into R2, which is the adder

4 LD3 – Load the number to be squared into R3, which is the sum

5 BL2 – If R1 is less than 2, than

6 ADD – Increase the sum by the original number

7 ADDS – If the last line overflows, add the carry flag, as R4 holds the high bits

8 DEC – Decrement R1, the counter, to keep track of the loops

9 BR1 – If R1=1, then R3 holds [0]^2, else loop back to ADD

10 ST1 – Store R4 to [1], as asked by specifications (high bits)

11 ST2 – Store R3 to [2], as asked by specifications (low bits)

12 LD4 – Load R1 with 32, the start of the width array

13 LD5 – Load R2 with the memory location [R1]

14 WDT – Calculate the width of R2, storing to R3

15 ST3 – Store the bit width to [3] (these stores are in case all inputs are 0)

16 ST4 – Store the number to [4]

17 ST5 – Store the location to [5]

18 LD6 – Save the first width to R4

19 INC – Increment R1, holding our place in the array

20 LD5 – Load the next number into R2

21 WDT – Save the width to R3

22 BGT – Check the width against R4, which holds the largest width

23 ST3 – Store R3 to [3] if R3 > R4

24 ST4 – Store R2 to [4]

25 ST5 – Store R1 to [5]

26 LD6 – Load R4 with R3, since R3 > R4 and R4 holds the largest width

27 BEQ – If R1 = 63 or R4 = 8, then we can’t or don’t need to go further

28 BRA – Otherwise, we need to go to the next number, branch back to INC

29 HALT – Programs completed, halt execution

The JAVA simulator was made to be as easy to use as possible. An assembler was provided so that commands could be written in English characters, but then simulated with hex code. The simulator would take the program input, and an initial RAM state input, run the code as if we were working with the circuit we wish to create, and then output the final RAM state and a clock-by-clock listing of the state of the registers and execution details. All of these inputs and outputs are saved as txt files.

For this ISA, I decided that 4 registers would allow for the best implementation of the instructions, while also minimizing the number of registers used. For the Square program, one register is needed for keeping track of the number of additions; Another register is used to keep the original number to be squared for use in addition; A third register is used to hold the result of the additions, while a fourth register is used for results which require more than 8-bits, and represents the top 8-bits of a 16-bit value.

For the RAM, the Square program requires use of only 3 bytes, which it requests to be the first three of the RAM; the Width program will only access a 32-byte array starting at the 32nd location of RAM, and will save the results at the 4th, 5th, and 6th bytes of RAM. For this reason, the RAM need only be 64-bytes long, which requires only 6 bits for addressing.

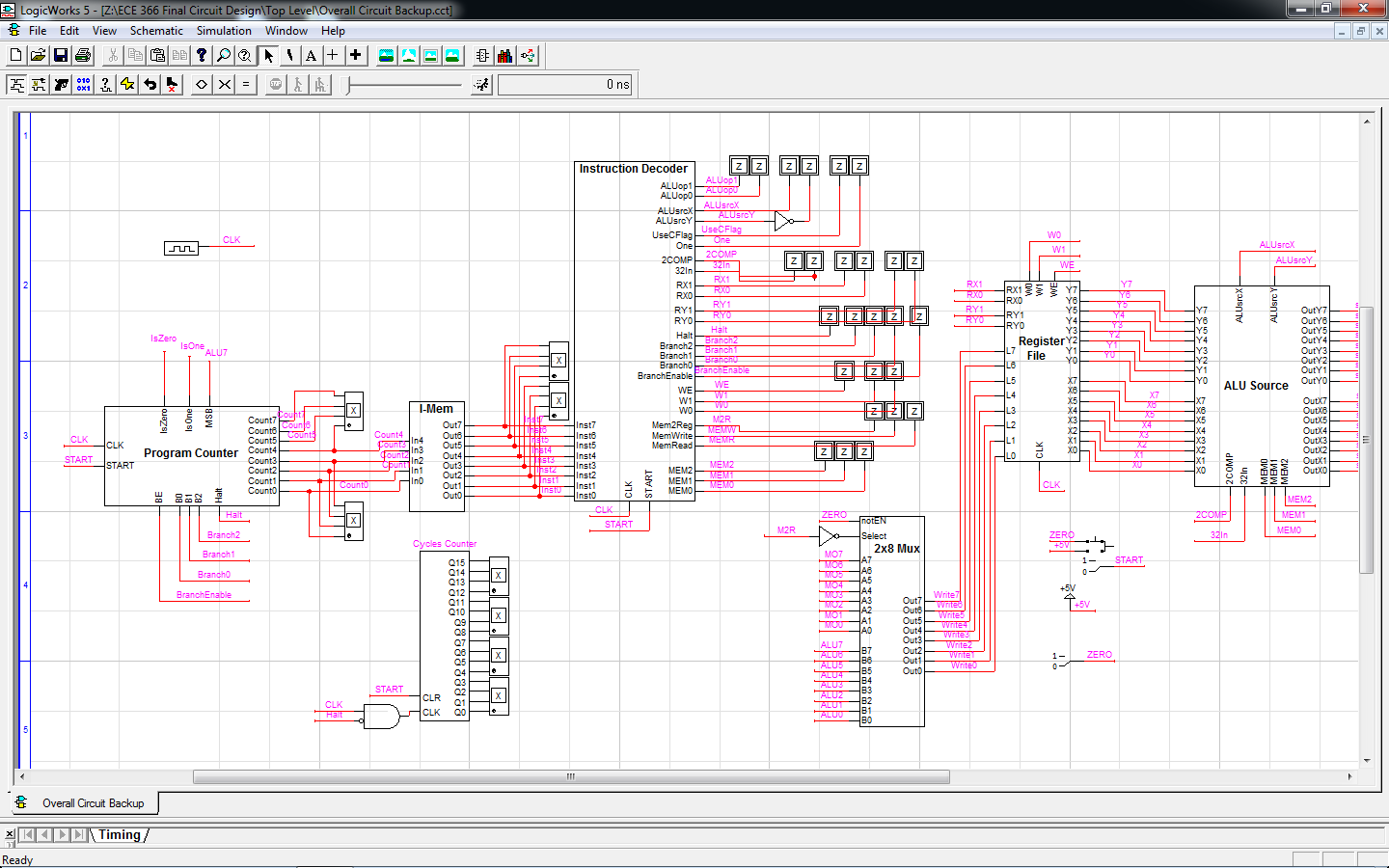
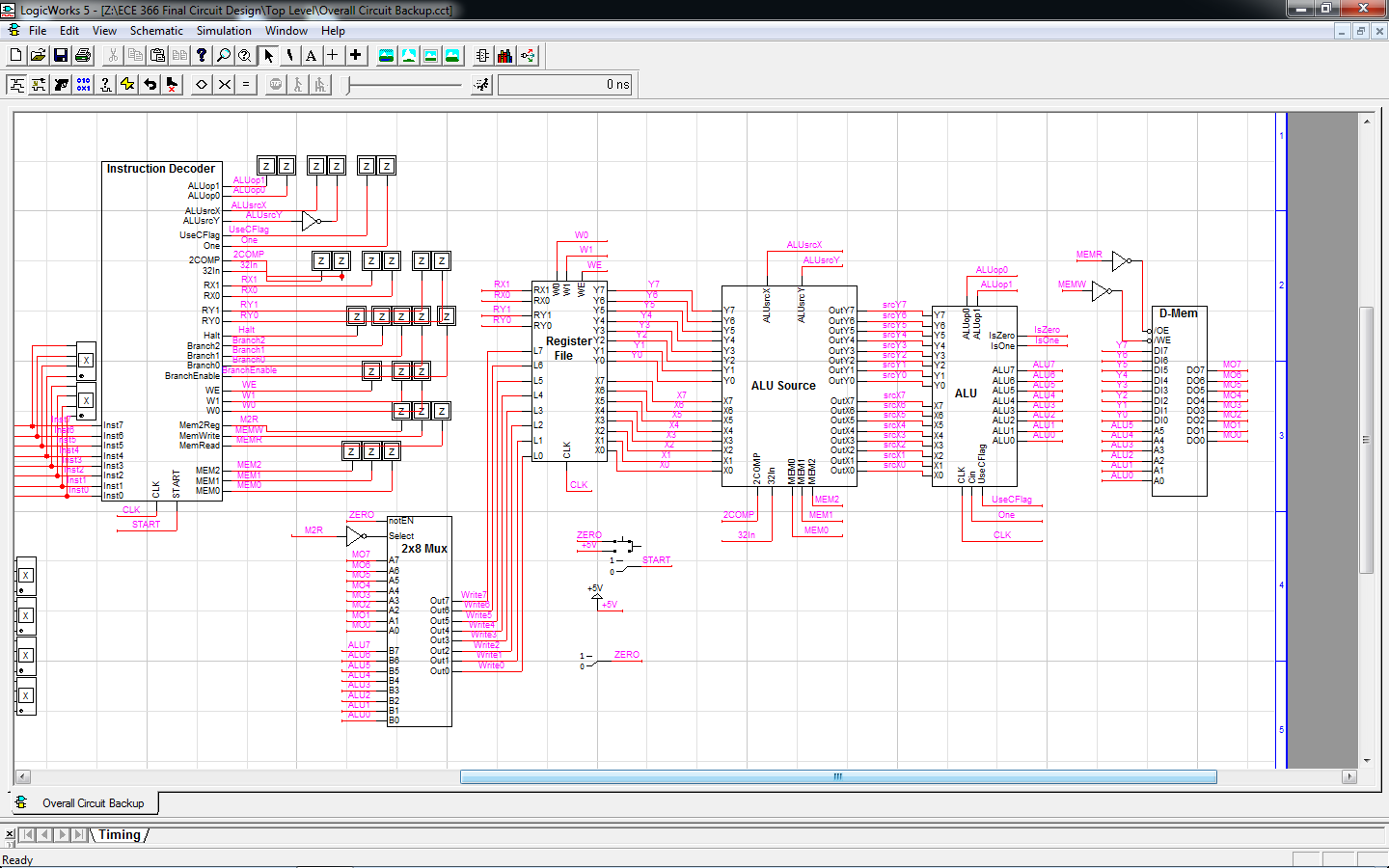
The PROM, which will hold the byte code for the programs, need only be as long as the programs are. At its final version, the program only requires 31 bytes. Therefore, a 32-byte PROM, requiring only 5-bits for addressing, was used.

**3) Structure Overview**

Here I will delve into the complete design of the processor. The order will go from top-most level, then the next level, and so on, until we reach the end of the lowest level. For circuits found only in one higher-level circuit, its truth table will be attached to the higher-level circuit. The subsection for such a circuit will immediately follow the higher-level circuit's subsection. Subsections for circuits found all over the design will be placed after all the second-level circuit.

For the use of grounding certain inputs, I decided to use a switch set to 0; I found this to be more reliable than any of the grounds provided in the default libraries.

Overall Design:

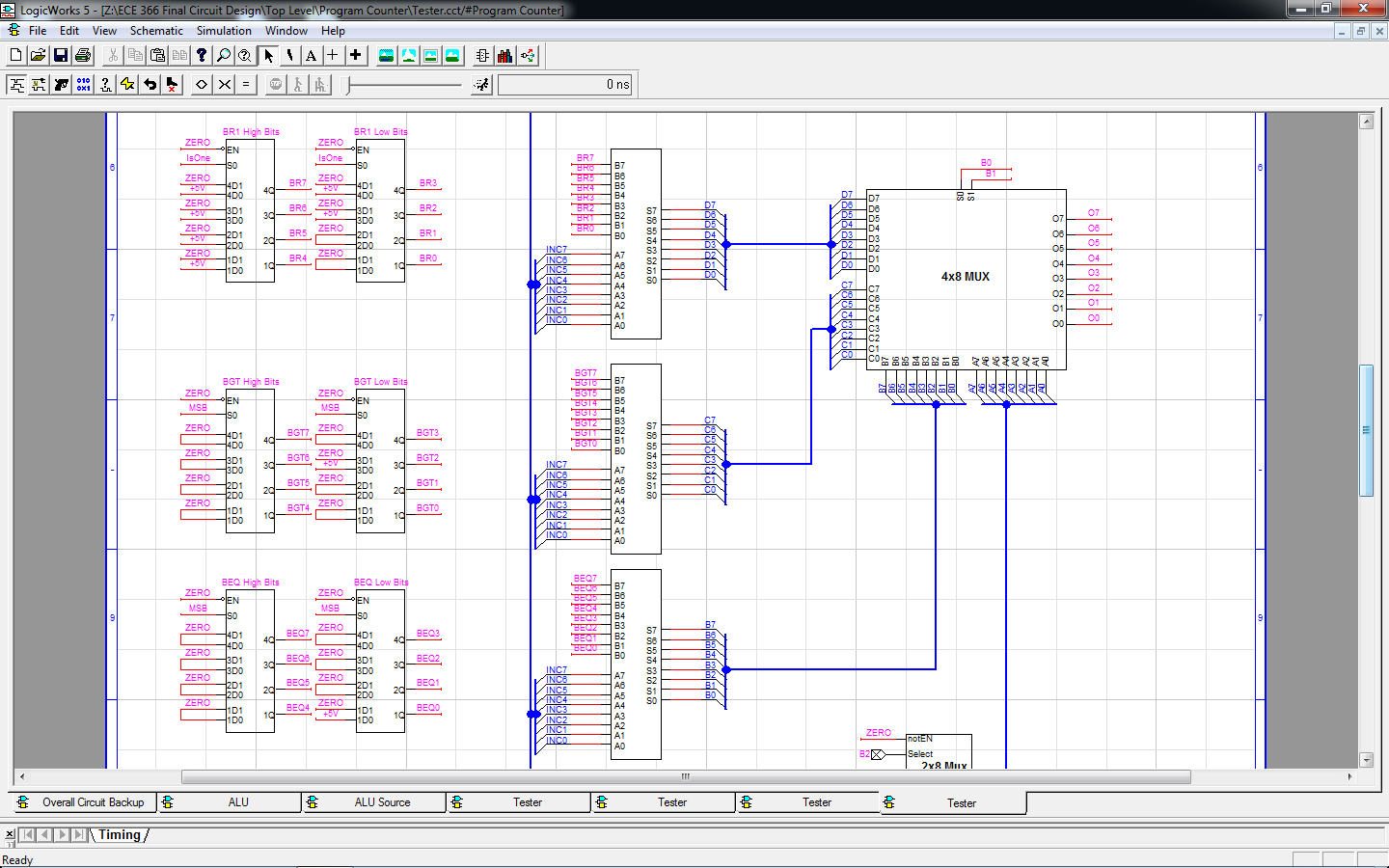
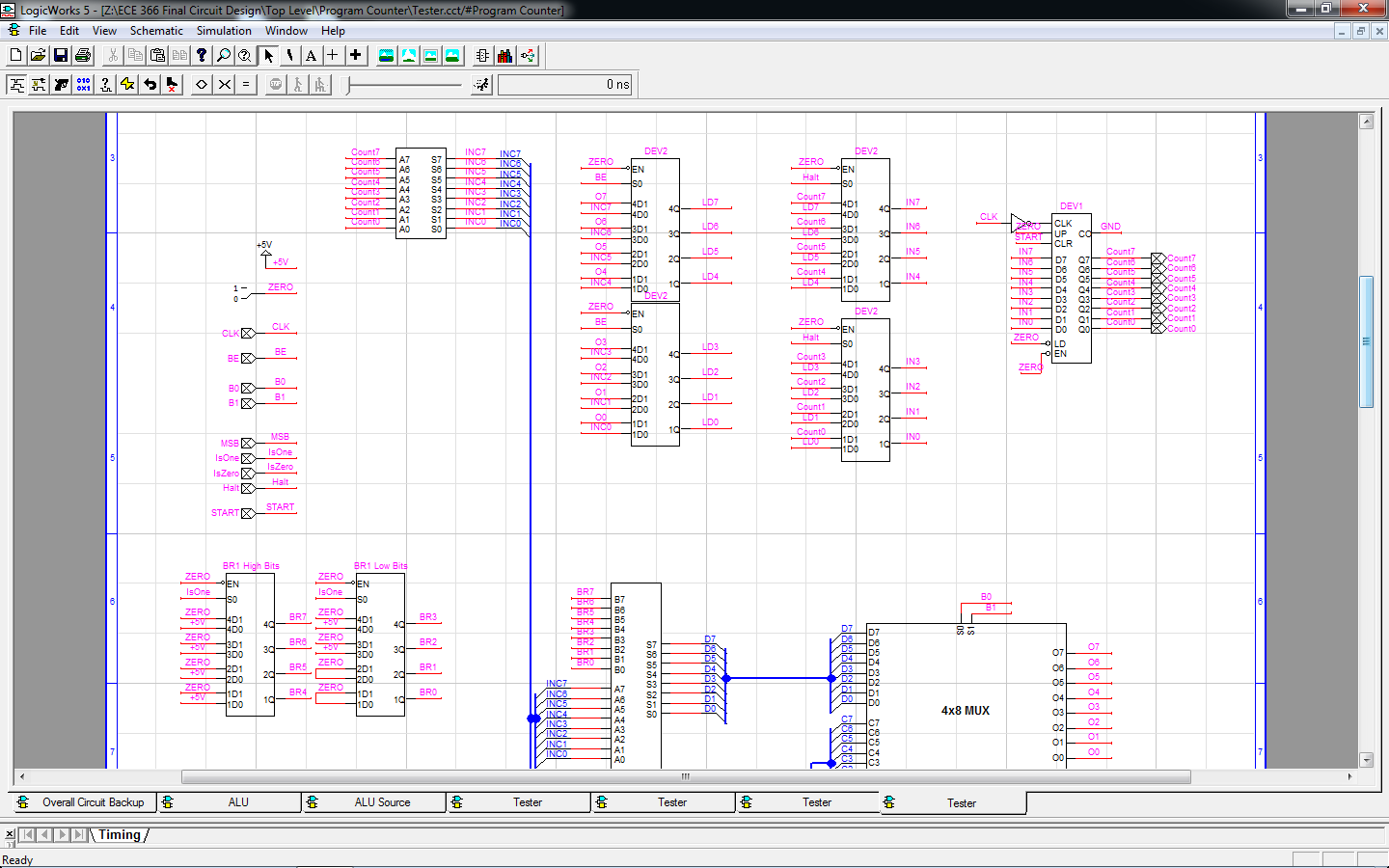
****

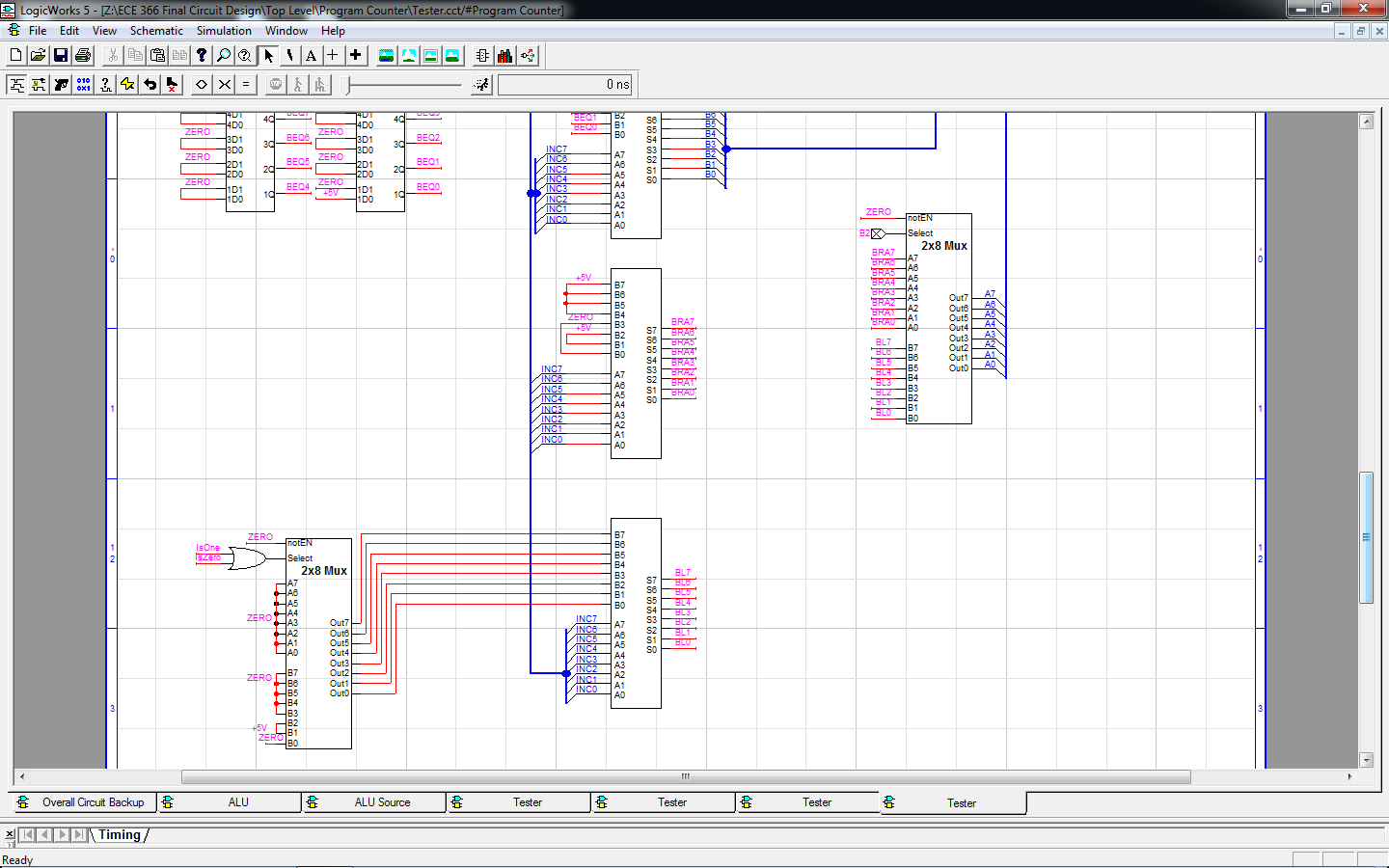
From Left-to-Right:

PC -> I-Mem -> ID -> Register File -> ALU Source Circuit -> ALU -> D-Mem

ALU/Mem Mux--^

Also present is the Cycle Counter, which keeps track of the total number of clock cycles that have run at a certain time.

PC/PC Update Design:



This circuit keeps track of, updates, and outputs the PC using several control signal inputs. This circuit is used at all times, regardless of the instruction.

From Left-to-Right, Top-To-Bottom:

PC Incrementor -> Branch Enable Mux -> Halt Mux -> Counter

BR1 Checker -> BR1 Adder -> Branch Select Muxes

BGT Checker -> BGT Adder

BEQ Checker -> BEQ Adder

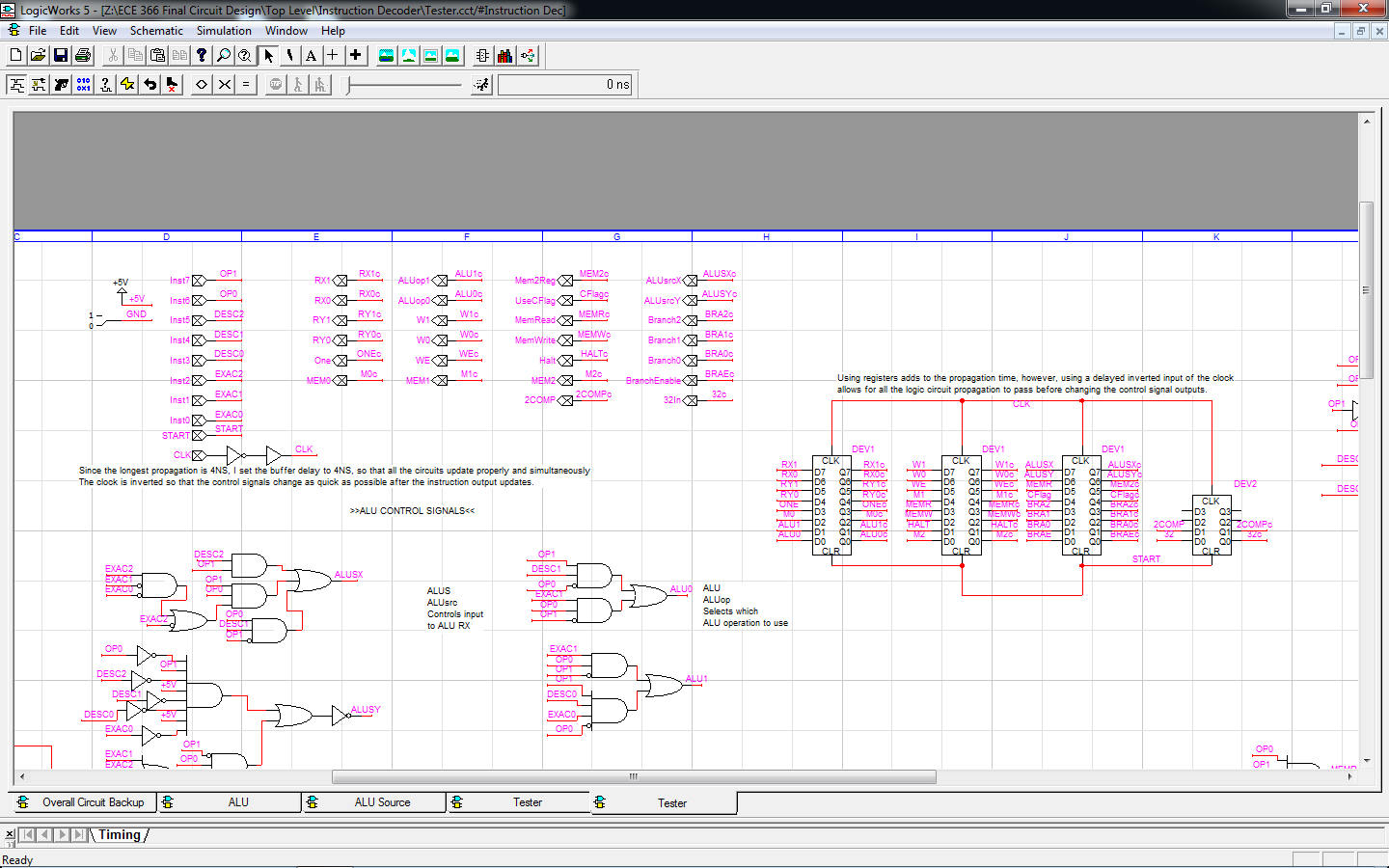
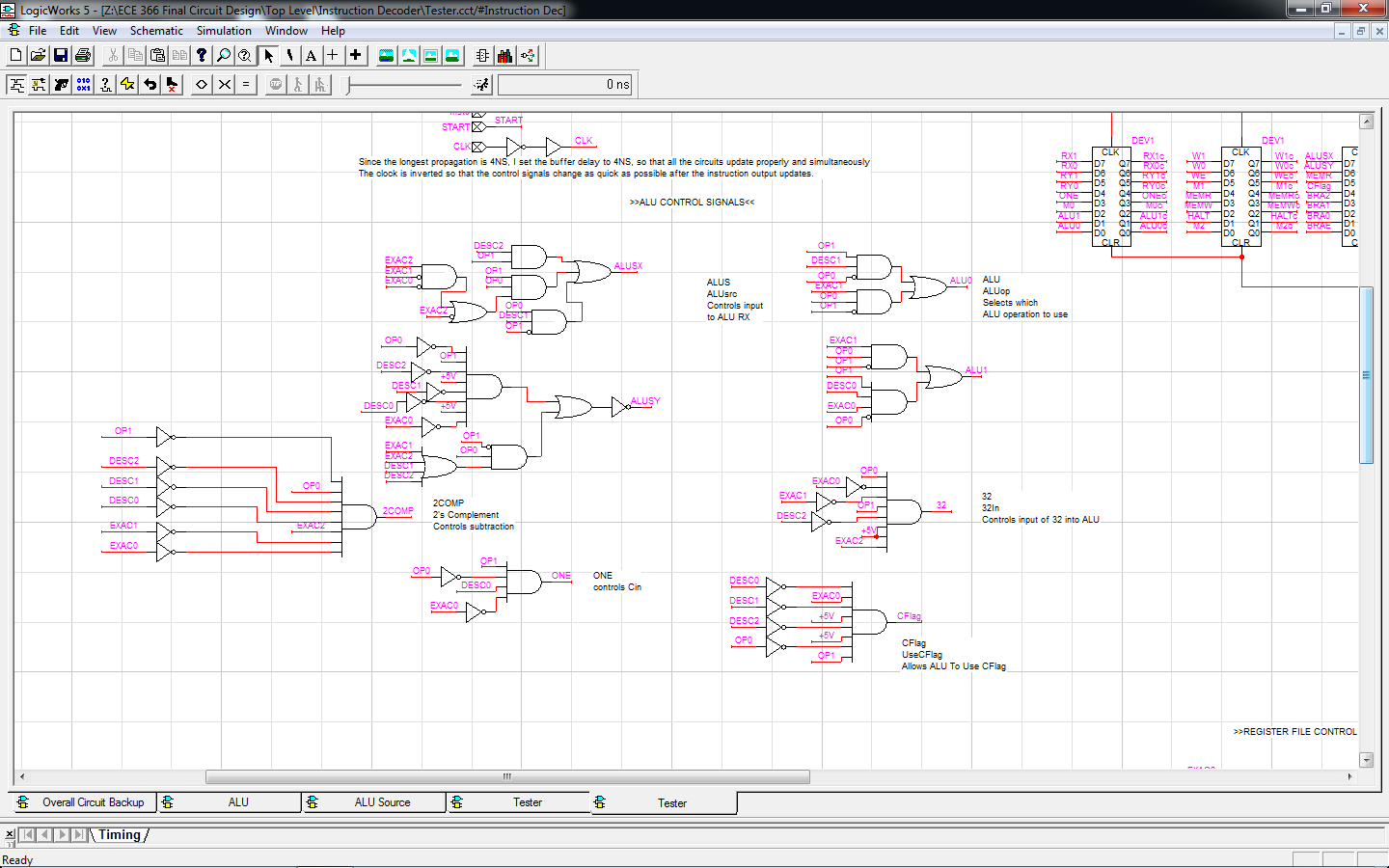
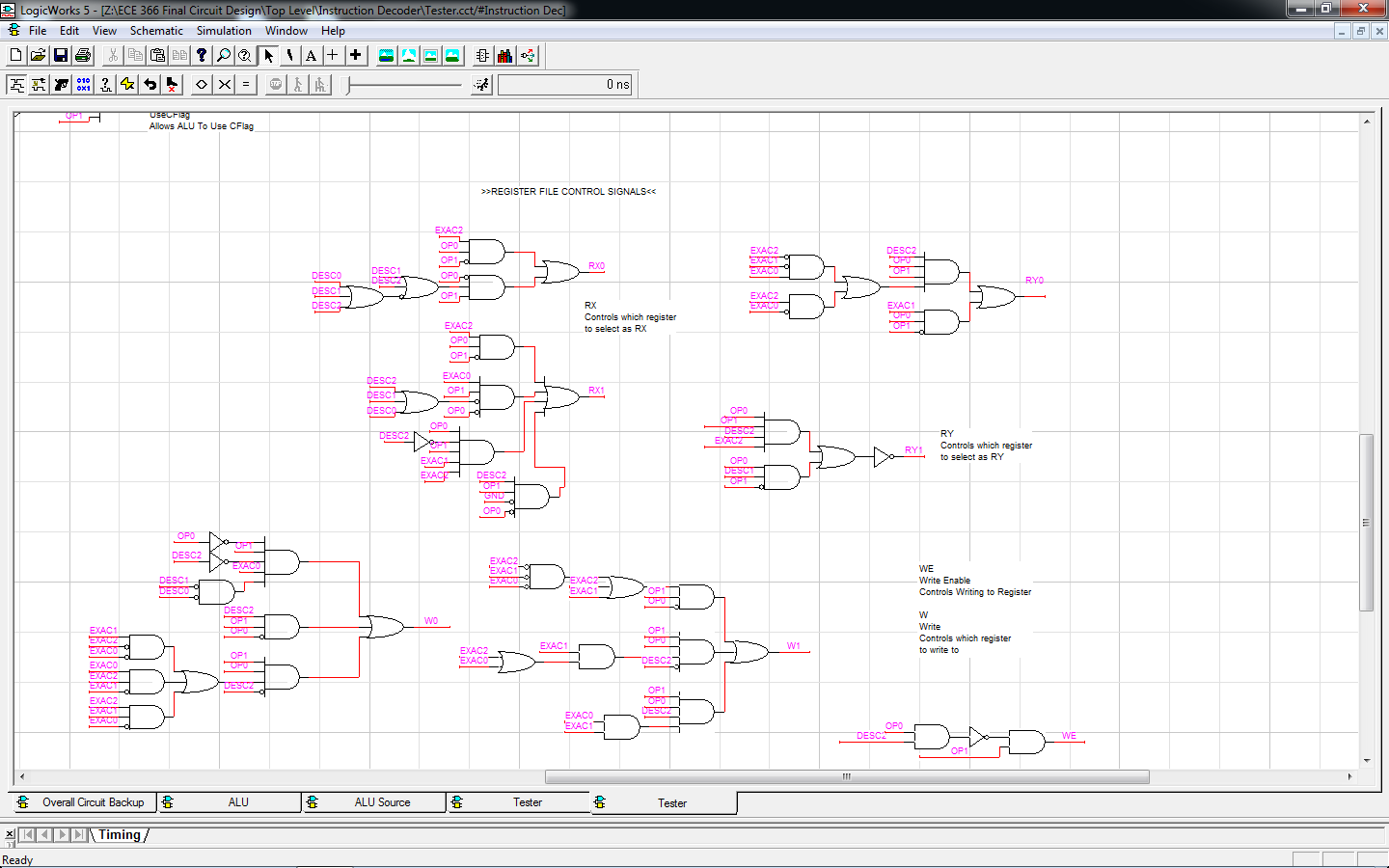
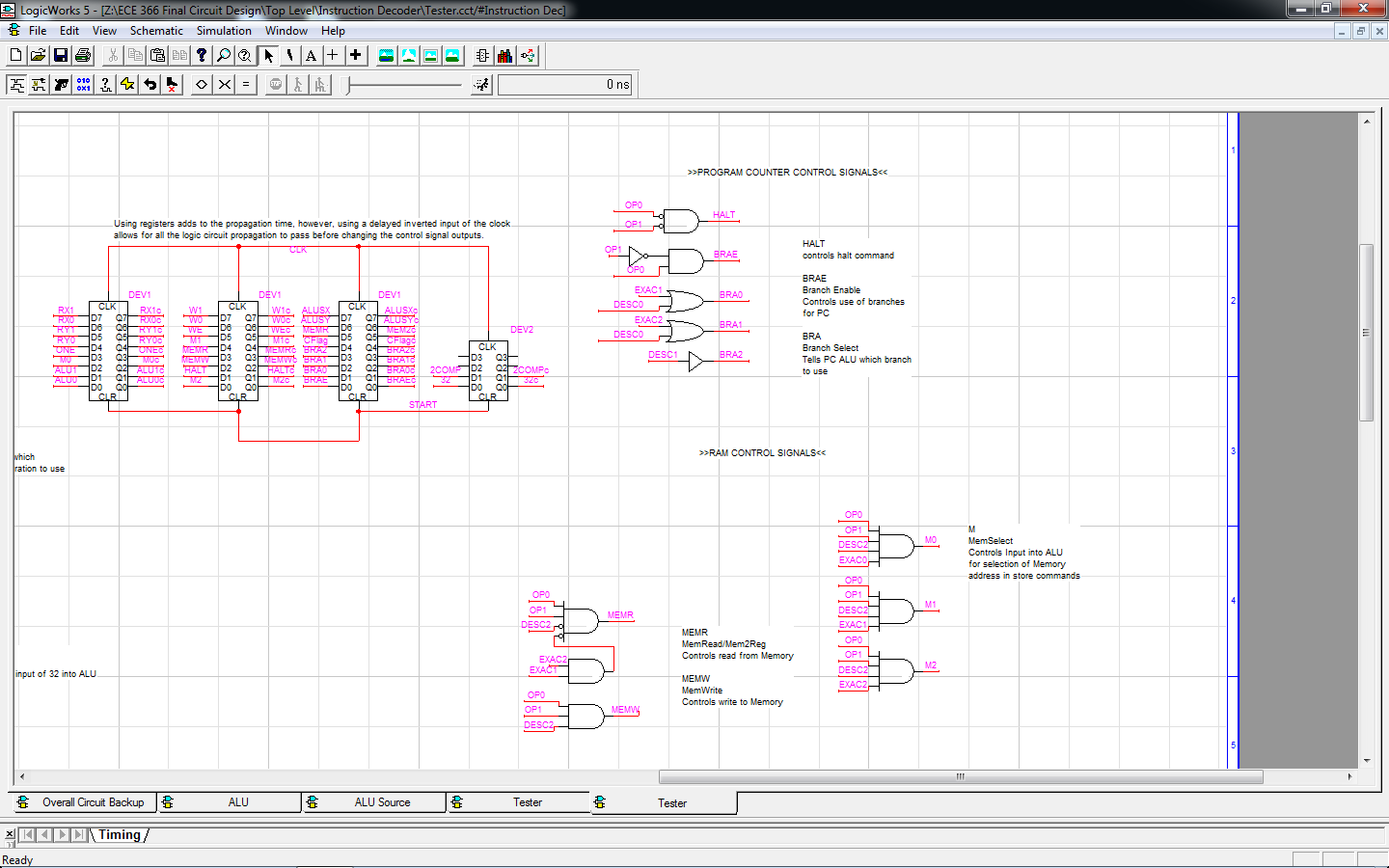
BRA Adder -> BL2/BRA Mux

BL2 Checker -> BL2 Adder

Truth Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Halt = 1 |  |  |  | Count = Count |
| Halt = 0 | BE = 0 |  |  | Count + 1 = Count |
|  | BE = 1 | B = 000 |  | Count - 9 = Count |
|  |  | B = 001 | MSB = 1 | Count + 1 = Count |
|  |  |  | MSB = 0 | Count + 2 = Count |
|  |  | B = 010 | MSB = 1 | Count + 1 = Count |
|  |  |  | MSB = 0 | Count + 5 = Count |
|  |  | B = 011 | IsOne = 1 | Count + 1 = Count |
|  |  |  | IsOne = 0 | Count - 4 = Count |
|  |  | B = 100 | IsOne = 1 | Count + 5 = Count |
|  |  |  | IsZero = 1 | Count + 5 = Count |
|  |  |  | IsZero = IsOne = 0 | Count + 1 = Count |

ID Design:



This circuit takes the Instruction Code Input and outputs the correct control signal values related to the inputted instruction. This circuit is used at all times, since it must decode every instruction.

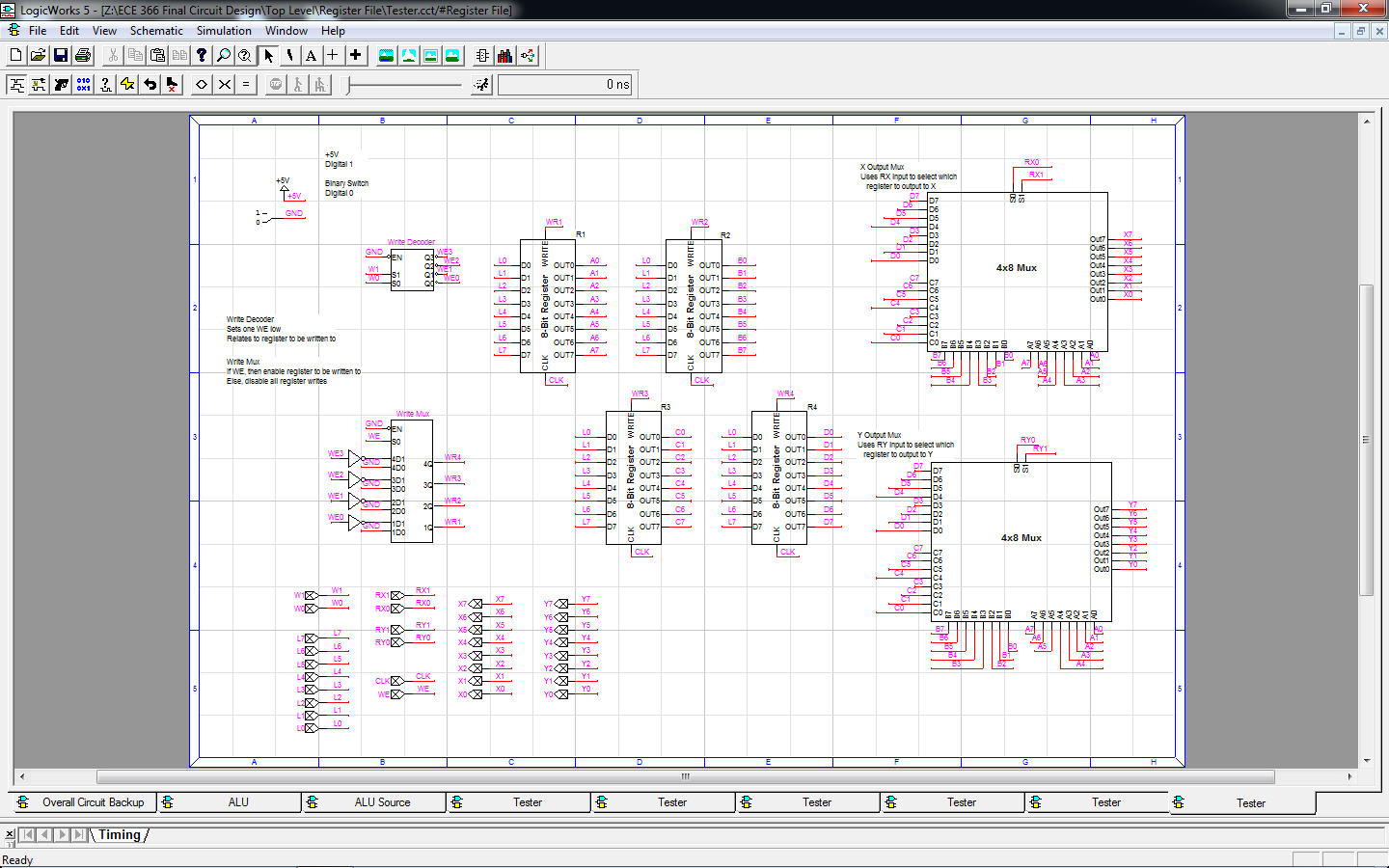
From Left-To-Right, Top-To-Bottom:

Registers -> Program Counter Signals

ALU Signals -> Memory Signals

Register File Signals

Register File:



This circuit controls writing to the registers and outputting of register values. This circuit is used for every instruction minus the halt command, because every other instruction requires input or output to/from the registers.

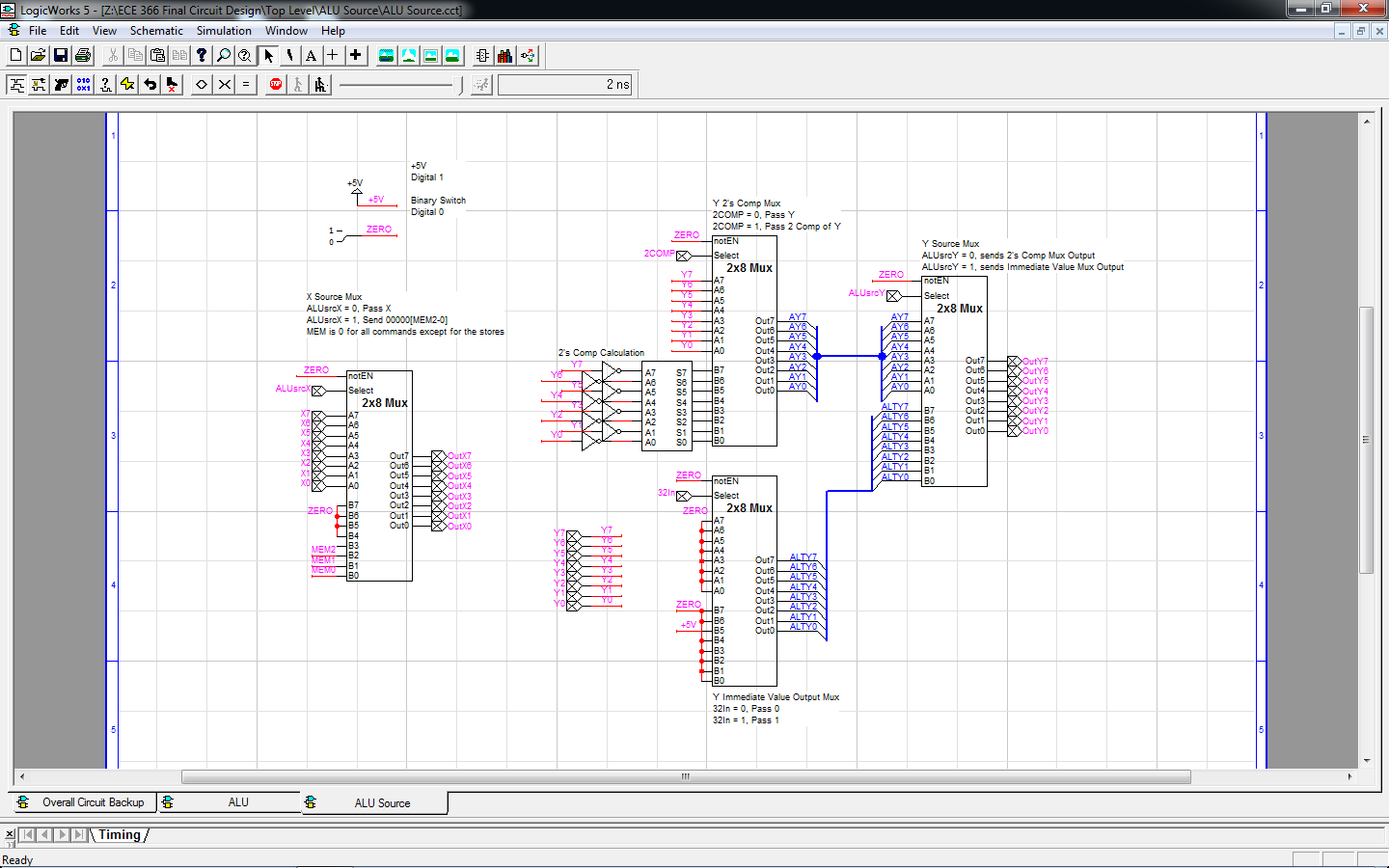
From Left-To-Right:

Write Enable/Select Circuit -> Registers (Clockwise from top left: 1, 2, 4, 3) -> X/Y Output Muxes

8-Bit Register:

This circuit takes a basic register and expands on it to allow for selective writing to the register.

Used in this circuit: 2x8 Mux, Reg-8

ALU Source Circuit:

This circuit takes in several control signals to select what to output by either manipulating the inputs from the register file or by using completely different values that are hardcoded to certain control signals. This circuit used by every instruction, as the proper source must be set for each command (minus the halt command, which only makes use of the PC and ID.

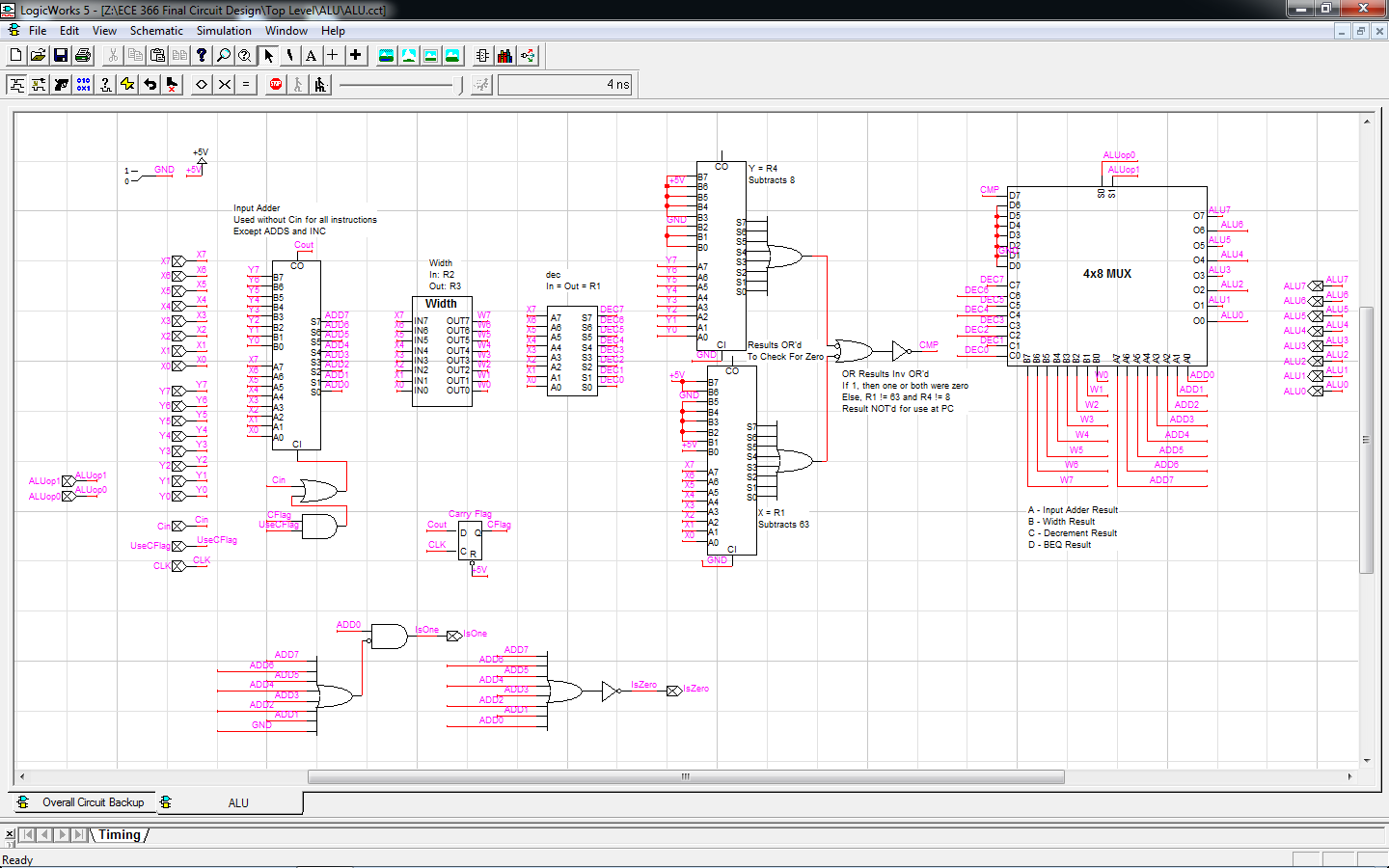
The X source part takes the X input and four control signals to decide whether to output X as is, or to output an 8-bit value whose three least significant bits are set by MEM2..0. MEM is 000 for all instructions besides the store commands. This makes it possible to send a value of 0 into the ALU's X input. This circuit only makes use of a 2x8 mux.

The Y Source part takes the Y input and three control signals to decide whether to output Y, a 2's Complement of Y, a value of 32, or a value of 0 This circuit uses three 2x8 muxes and an 8-bit incrementor.

From Left-To-Right:

ALU X Select -> ALU Y Select

ALU:

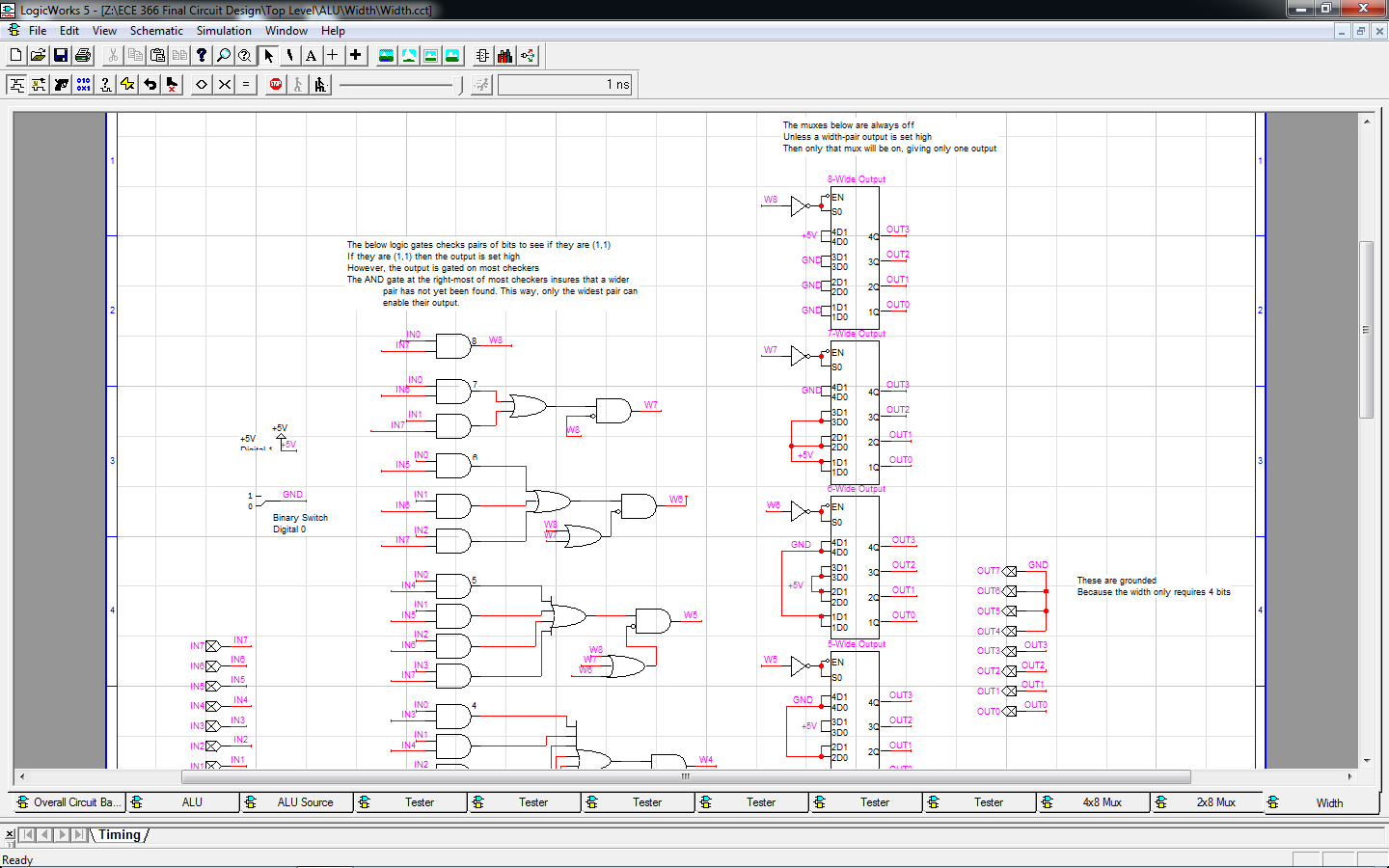


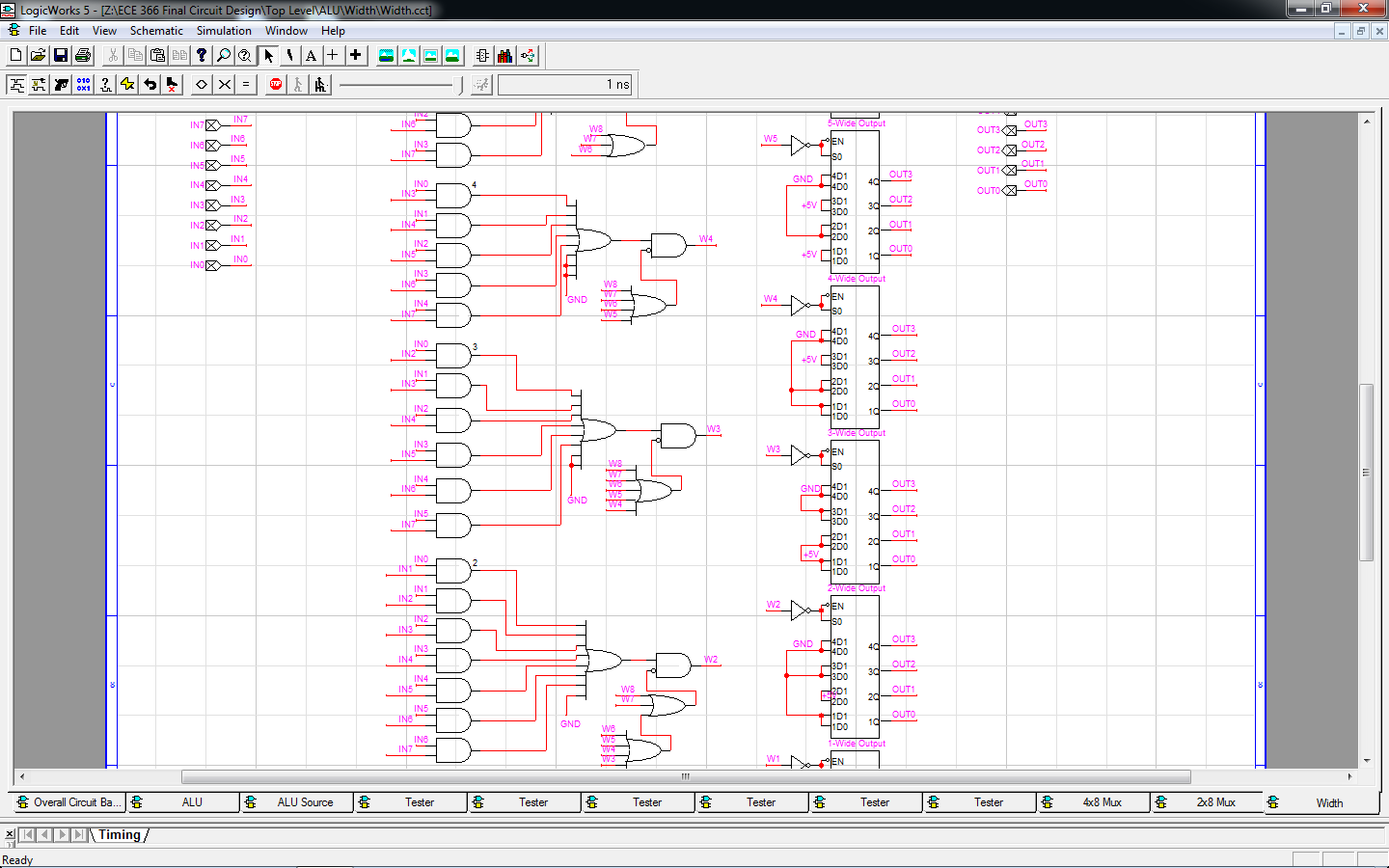
This circuit takes two inputs and several control signals, to output one value and a couple control signals. The inputs will be put through all the circuits, and the control signals will control the mux select. Besides halt, this circuit is used by all instructions, though a few commands only use the ALU as a pass-through (Input + 0 = Input, for branch checks and memory calls). This circuit also outputs a set of control signals, used for branches and the ADDS instruction.

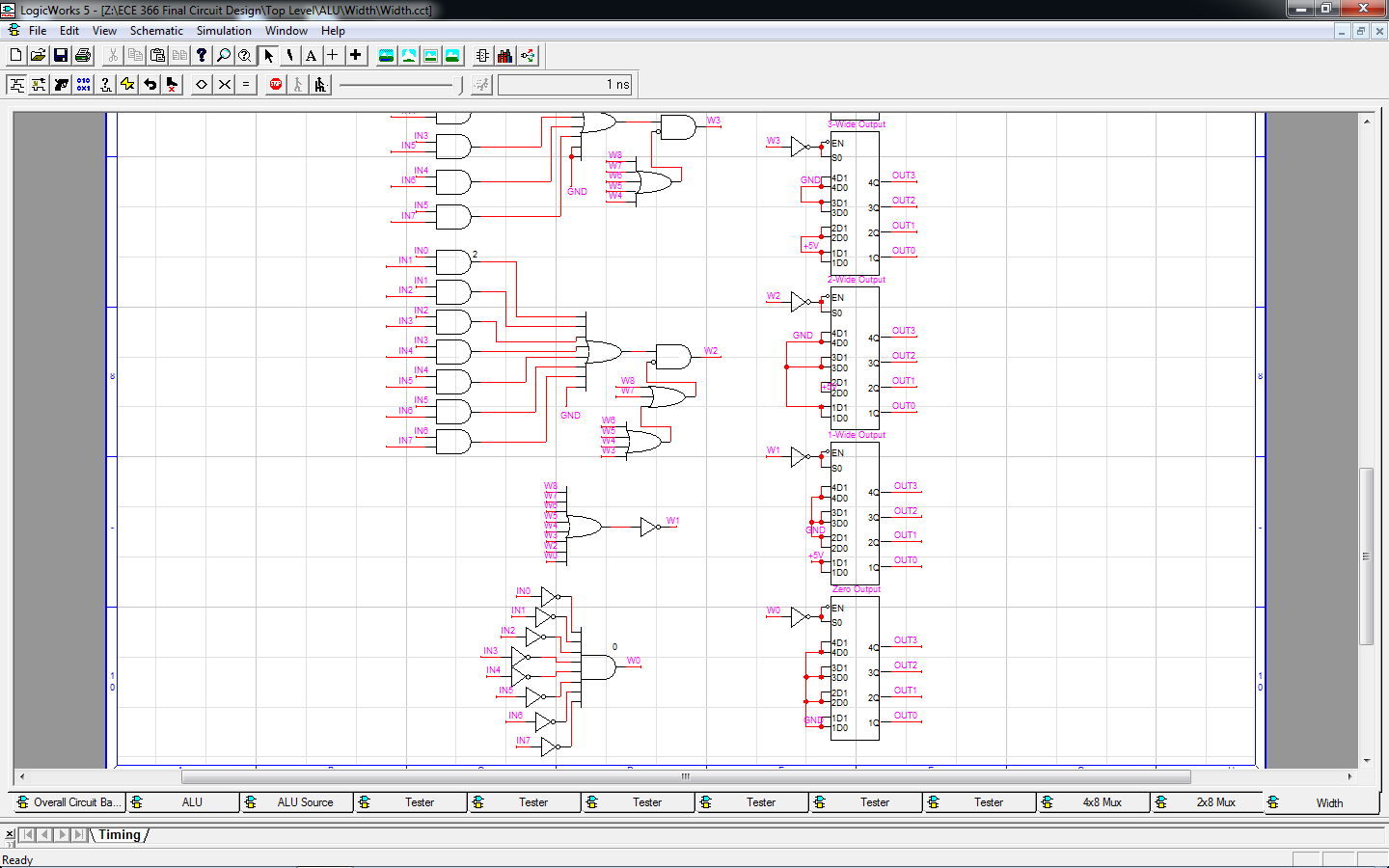
From Left-To-Right, Top-To-Bottom:

Input Adder -> Width Circuit -> Decrementor -> BEQ Value Checker -> 4x8 Mux

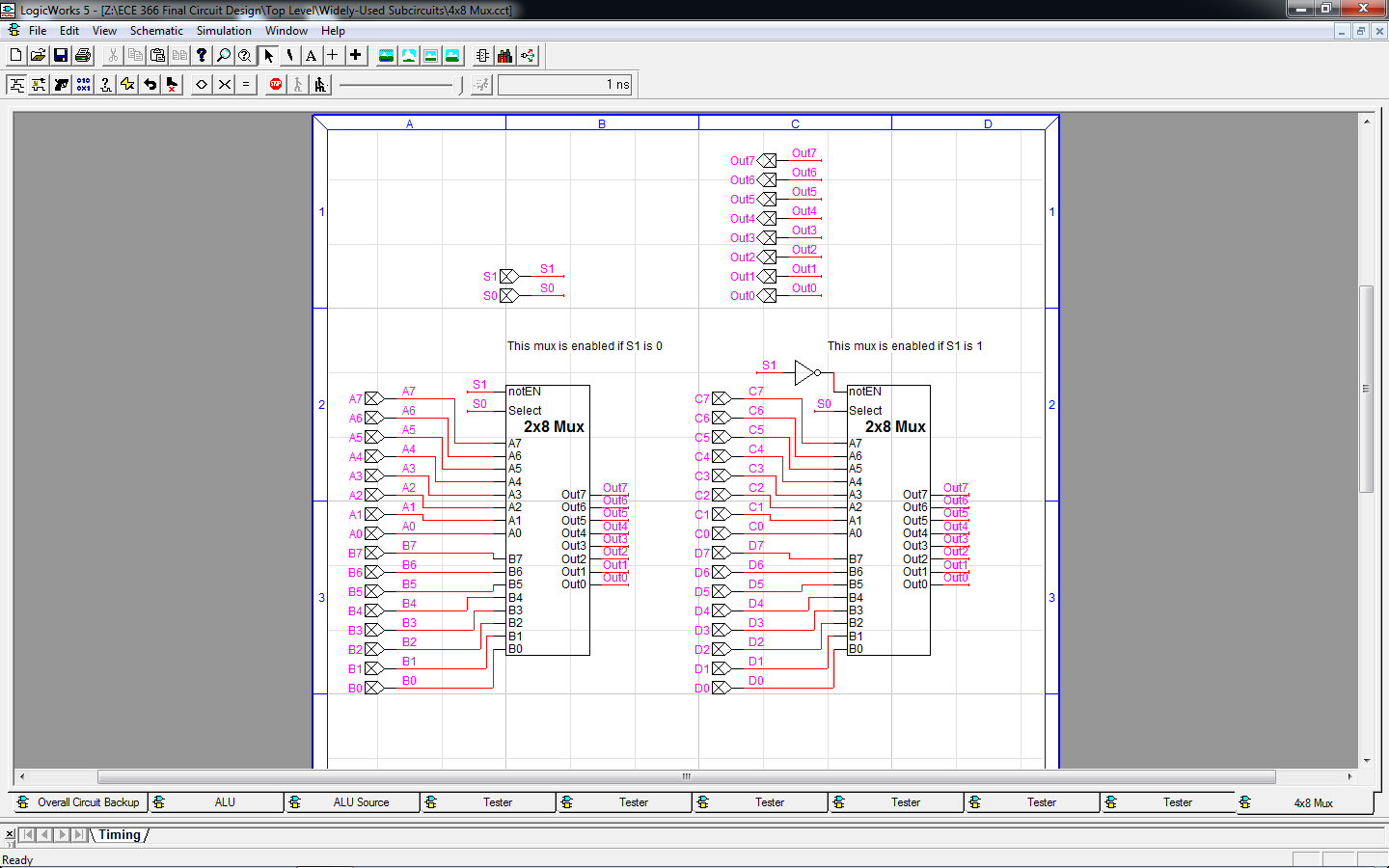
IsOne Circuit -> IsZero Circuit

Width Circuit:

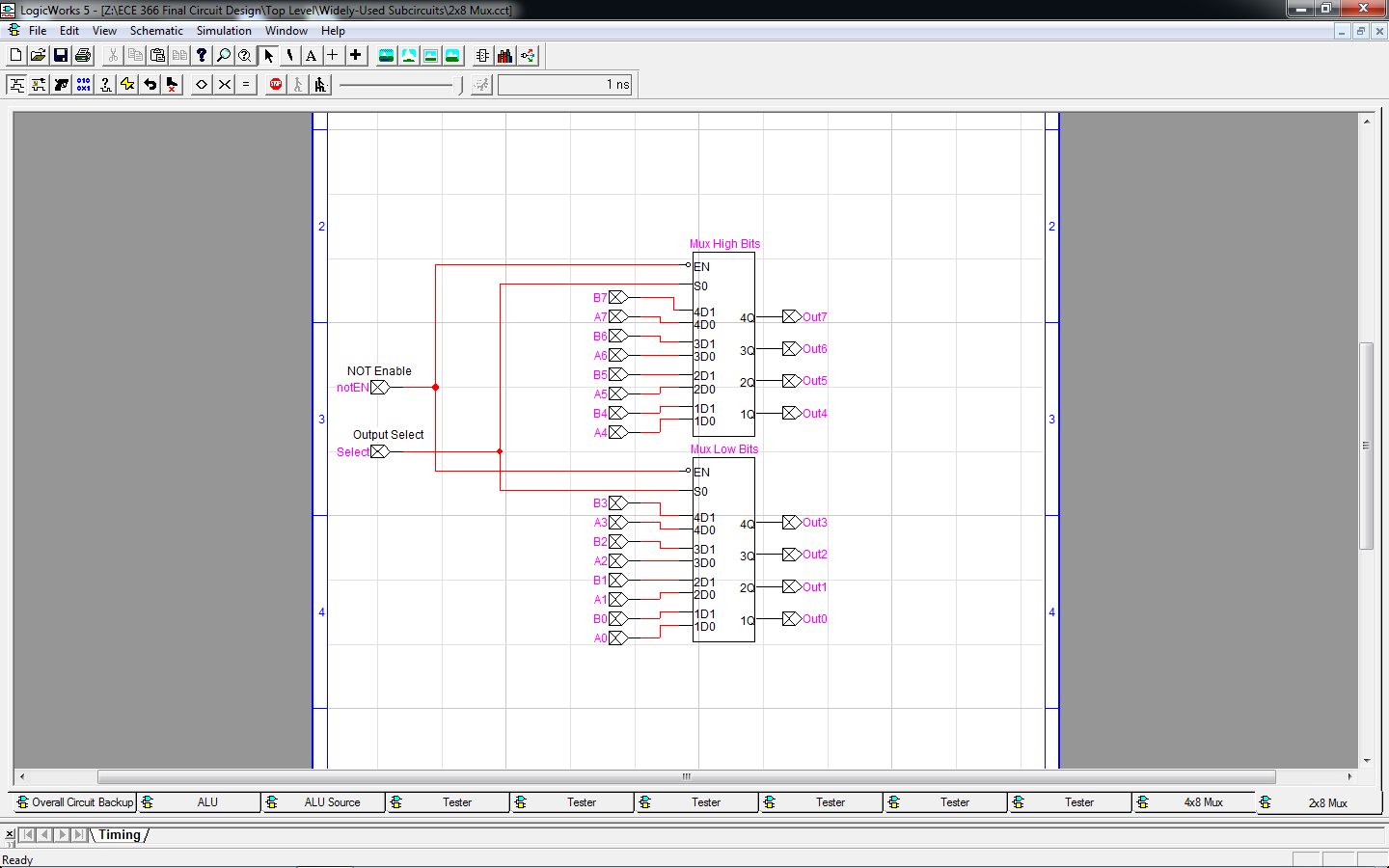




This circuit takes an 8-bit input and outputs the bit-width of the input. This is accomplished by checking every set of two bits. Depending on the sets that are (1,1), we can then find the widest pair, and output the width of that pair. This circuit consists of AND gates, OR gates, and nine 2x4 muxes (One for each possible width: 0-8).

4x8 Mux:

This circuit takes four 8-bit inputs and uses two control signals to select which input to output. This circuit uses four 2x4 muxes found in the default libraries. As this circuit is present in the register file, PC circuit, and the ALU, it is used by all instructions.

2x8 Mux:

This circuit takes two 8-bit inputs and uses a control signal to select which input to output. This circuit uses two 2x4 muxes found in the default libraries. This circuit is present in all self-made second-level circuits, minus the ID. Therefore, every instruction makes use of this circuit.

**4) Branches**

The SAS ISA utilizes 5 branches, 6 if you count halt as a branch, due to it's PC=PC-1 nature. Keep in mind that the PC auto-increments before any control signals are checked. These branches are as follows: BRA, BEQ, BGT, BR1, and BL2.

BRA:

This is the only non-conditional branch. It will automatically set PC to PC-10.

BEQ:

Branch PC+1 if EQual. This branch is taken if R1 is equal to 63 or if R4 is equal to 8. Otherwise, the PC will auto-increment, as in normal operation. 63 and 8 are subtracted from R1 and R4, respectively, in the ALU. The ALU sets the MSB to 0 if either operation returns a 0. The branch is taken if the MSB control signal is 0.

BGT:

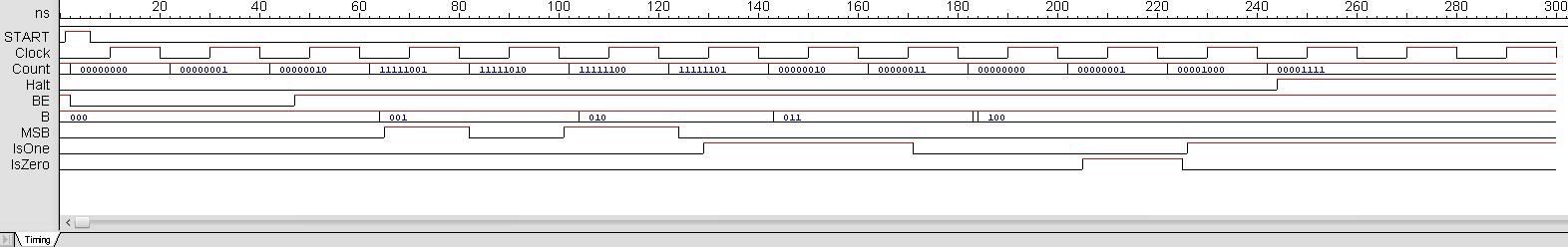
Branch PC+4 if Greater Than. This branch is taken if R4 is greater than or equal to R3. Otherwise, the PC will auto-increment normally. R3 is outputted by the Register File, and the ALU Source circuit outputs the 2's complement of R3. R4 and the output from the ALU Source circuit are then added together. If the ALU Output's MSB is 0, then the branch is taken.

BR1:

Branch PC-4 if R1 is not 1. This branch uses the IsOne output from the ALU. If IsOne is 0, then the branch is taken.

BL2:

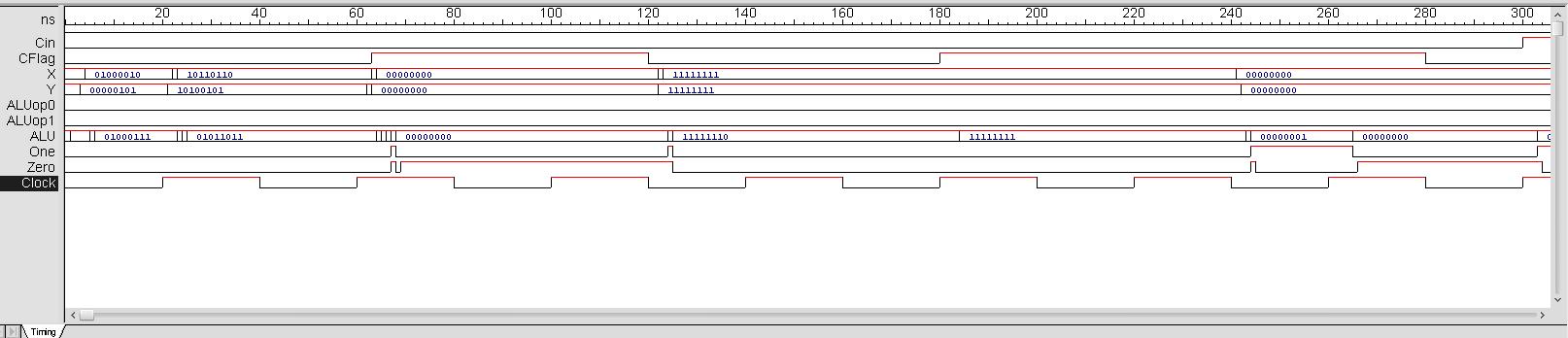
Branch PC+6 if R1 is Less than 2. This branch takes the IsOne and IsZero outputs from the ALU. If either signal is 1, then the branch is taken.



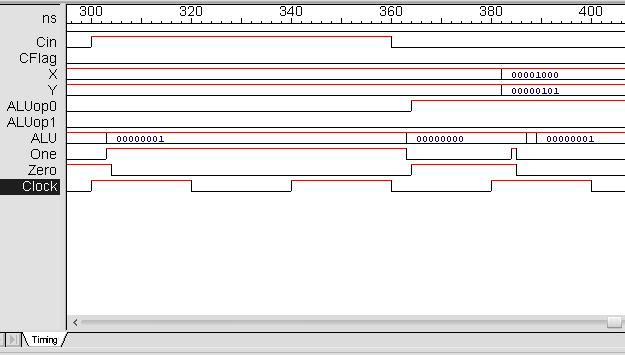
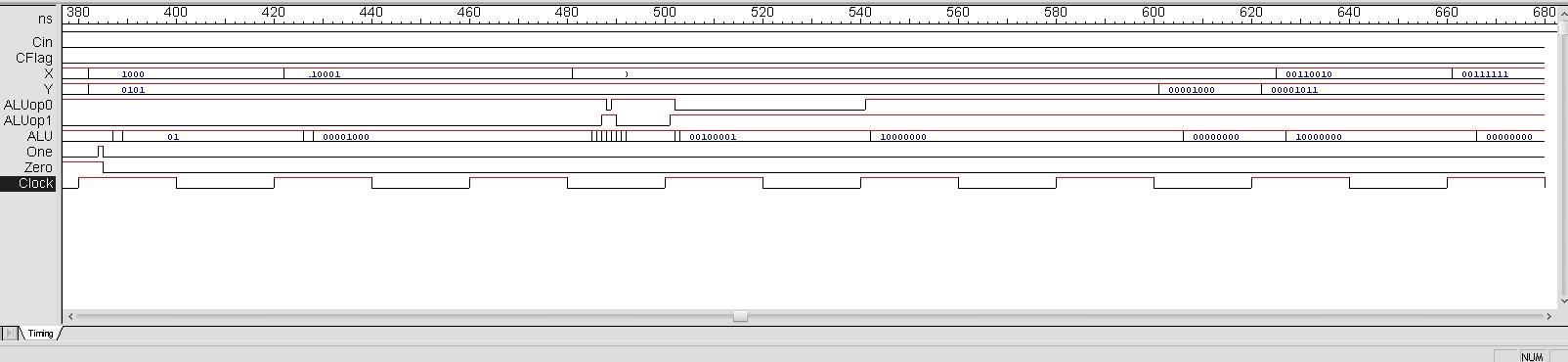
In the first two cycles, the PC is just incrementing. Then in the next three cycles, we take the unconditional branch BRA. In the sixth cycle, BEQ is used, but falls due to the MSB being 1, which will mean that R1 does not equal 63 or R4 does not equal 8. In the seventh cycle, the MSB is 0 (one of the two conditions is met), and the BEQ branch is taken. The eight cycle attempts to use BGT, but since MSB is 1 (R4 < R3), the branch is not taken. The next cycle has MSB set to 0 (R4 >=R3), so the branch is taken.

**5) Timing Diagrams**

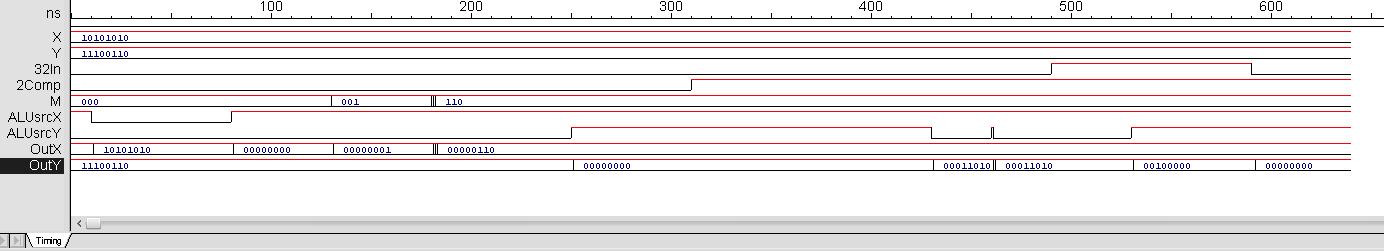
Due to problems with Logicworks5, I was unable to get the processor to run completely after the demo. Even in the demo, there were a few errors from control signals propagating quicker than others, which I attempted to fix, until LW5 broke the circuit. Despite all this, I was able to get all the pieces working individually, and have provided the timing diagrams I extracted from their tester circuits.

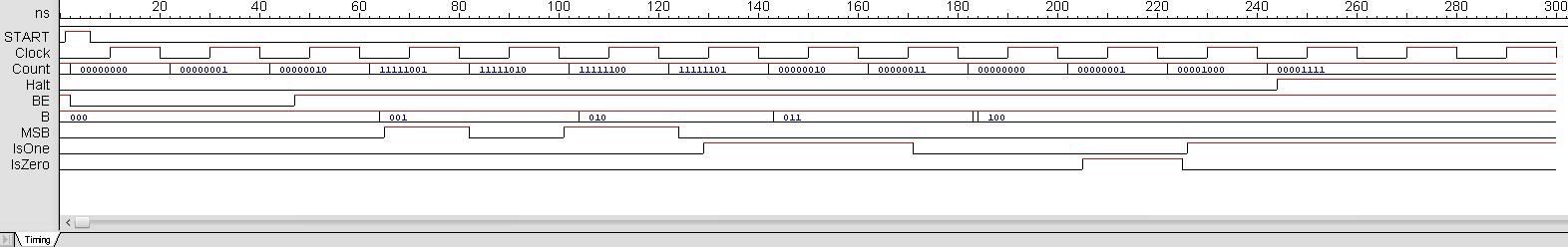
****

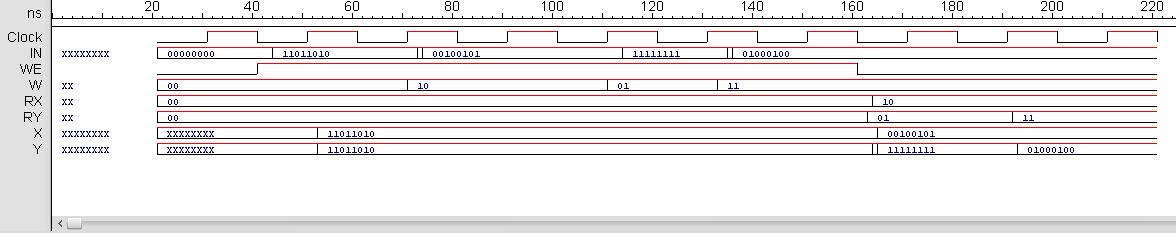
ALU:

****

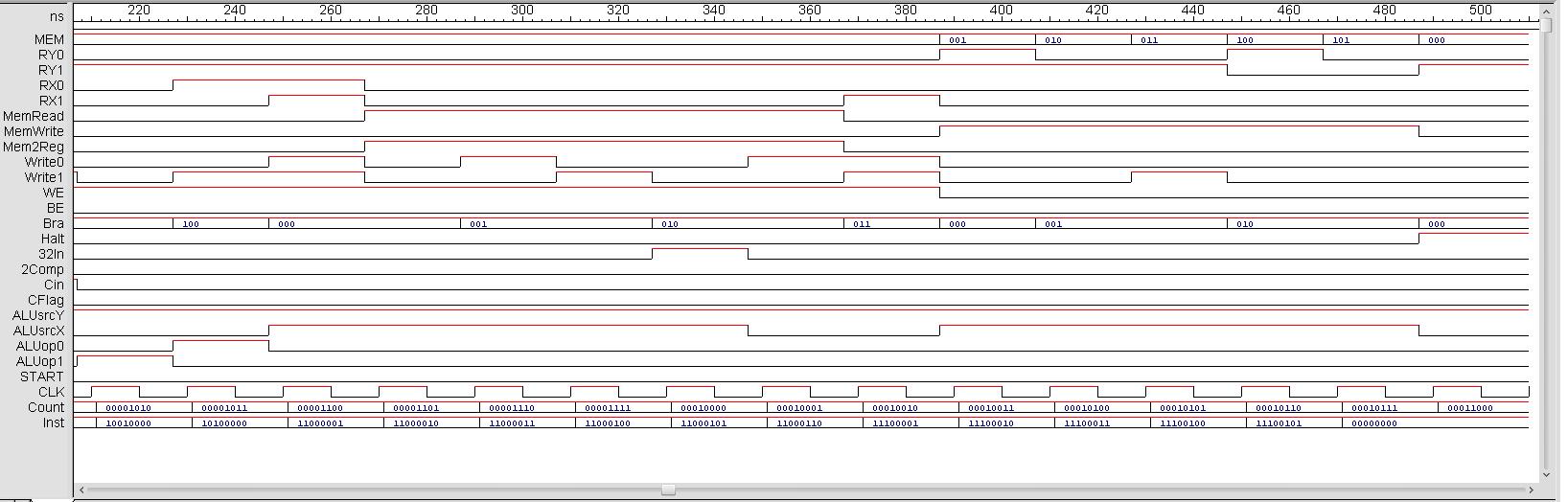
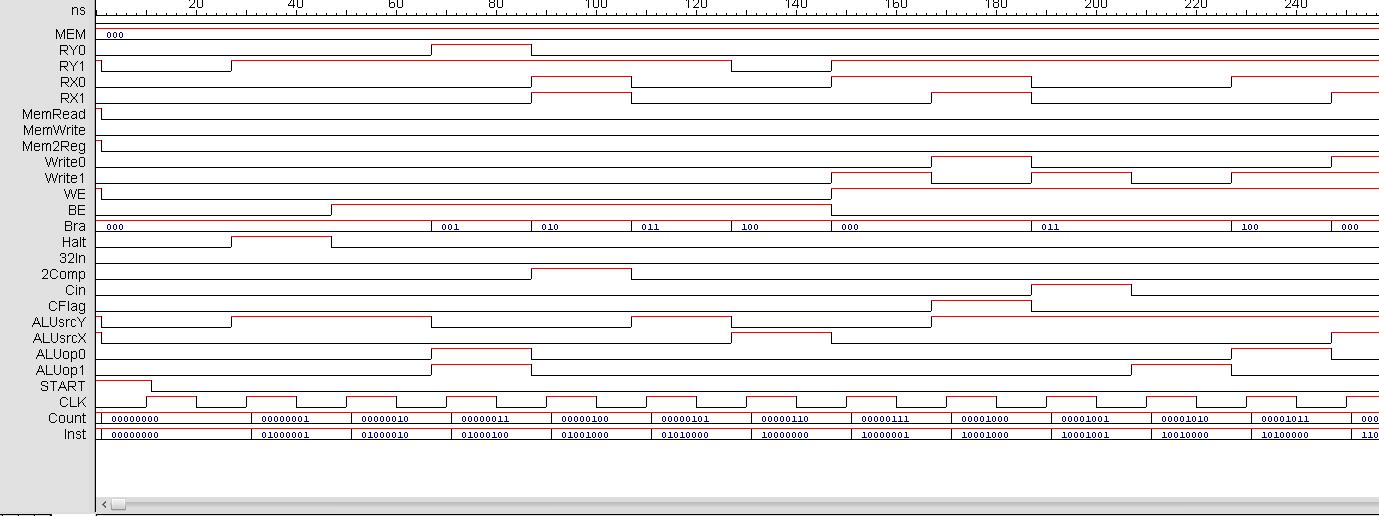
ALU Source:

****

Program Counter:

Register File:

Instruction Decoder:



**6) Other details**

All of the pictures in this report are available in the folder provided, along with all the circuit files.

**7) Answers to the Questions**

A) The changes I have made are as follows: Branch lengths, New branch command. The branch lengths were changed due to the fact that the original branches did not take into account the auto-incrementing nature of the PC. The new branch command was added upon realization that the Square program would not work for an input of zero or one. This branch would check the value and branch if the input was less than two. Zero squared is zero and one squared is one, therefore the input is the output.

B) I was unable to obtain any sort of dynamic instruction counts. I do not believe that there are any easily implemented, or even easily conceptualized, methods to further optimize the instruction counts of my ISA.

C) I feel I was very successful in optimizing for ease of design. None of the instructions were overly difficult to create circuits for, and the only difficulty came from creating the instruction decoder. This was because I designed the control signals after designing the byte code for the instructions, so multiple-use signals (Rx, Ry, etc.) had large logic circuits. If I had to optimize the design, I would rework the byte code so that control signals were easier to design circuits for. Propagation was also a problem with the instruction decoder, which I attempted to fix by using clocked registers.

D) Extending my circuit into a multi-cycle implementation would not require too much effort. The instruction decoder would require the most work, because the rest of the circuit would work as is, due to the fact that most of the sub-circuits do not make use of the clock. Additional control signals would need to be added and the existing signals would need to be reworked. From here, we could attempt a pipe-lined implementation. This would be significantly more difficult. Now we will have multiple instructions running at once. So then we will have to either force no-ops, which will hurt performance, or figure out how to properly forward for every instruction, which could require a lot of work.

E) If the priority of the ISA had been ease of programming, I would end up with an identical ISA. This is due to the fact that none of the instructions in this ISA require any modifiers. Most ISAs would use some form of "Str R4, [1]" for their first store instruction, my ISA uses a much simpler, shorter "st1". By removing all modifiers from every instruction, the programmer no longer needs to worry about how to write an instruction for an immediate load or a memory load. Each instruction does just one job, keeping programming completely straight-forward.

F) Nearly every instruction makes use of every high-level sub-circuit, with the exceptions being HALT and BRA. These would be the two shortest instructions. To ease in finding the instruction with the longest execution time, we can take out the I-Mem, the Instruction Decoder, the Register File, and the D-Mem, as these five will result in identical timings for every instruction. Some instructions will take an extra 1NS for writing to the Register File, which we will take into account later. We can leave the ALU Source for later, as the longest propagation will be 4NS for BGT, and the most common propagation will be 1-2NS. Inside the ALU, however, we can see that all the instructions will have a propagation of 1NS. That is, except for the width and beq instructions. The width circuit inside the ALU has a total propagation time of 6NS. In comparison, the beq circuit has a total propagation time of 4NS. The width instruction will use 1NS in the PC, and the beq instruction will take 3NS. So for the three instructions mentioned, the delays are approximately as follows:

Command: PC + Register File (Write vs No Write) + ALU Source + ALU = Comparative Propagation Time

WDT: 1NS + 1NS + 1NS + 6NS = 9NS

BGT: 3NS + 0NS + 4NS + 1NS = 8NS

BEQ: 3NS + 0NS + 2NS + 4NS = 9NS

Using this rough deduction, it shows that the width command and the BEQ command are both the longest propagating instructions.

G) If the priority had been to keep clock times short, then I would increase the amount of instructions in the ISA and, in turn, the size of the program. For example, the width instruction would have to be split into a set of IsWdt# (IsWdt8, IsWdt5, etc.) instructions. This would allow me to check the widths without the added AND gates and muxes used in my current circuit to keep from having multiple outputs, which would save 2NS in the longest instruction. I would also have to rework the branch circuitry to minimize the propagation time of those instructions.

H) While I could get all my sub-circuits working properly, putting them all together would always, somehow, break one of them.

I)

i. These labs taught me a great deal. From lab 1A to lab 3B, there was always a new step in the design process that I got to experience. These labs seem like real-world applications of computer organization. We got to deal with many of the intricacies of processor design. From building an ISA to fit certain criteria, to the ever-present threat of hazards propagating through out the processor. These labs showed us how certain things we have learned in other classes can make a big impact in designing a processor, like logic optimization, which is useful here because it can decrease propagation times, leading to shorter clock times, giving us better performance.

ii. The best thing about these labs was that not only did we build a processor, but also we did it as if it were a job assigned to us in the real world. We went through the motions of acquiring a set of criteria, designing an ISA around it, creating simulators to test our design, and then actually creating all the sub-circuits for the final processor.

iii. The worst thing about these labs was LogicWorks5. LW5 loves to crash. Running a circuit with tied inputs could crash LW5. Trying to use the built-in Timing Diagram Print option causes LW5 to crash. When LW5 is not crashing, it will randomly lose some signals and buses in the timing diagram. Sometimes closing a circuit and reopening it could make a working circuit turn into a massive mess.

iv. Paper and Pen are your friend. So are Excel and Word. Plan everything out, save everything in a easy-to-reach spot. Excel is perfect for when you are trying to figure out the values of control signals from just the instruction code. Also, don't use LW5 in a Windows emulator on a Mac. It feels even buggier there then it does on a Windows PC. If you have a Mac, invest in setting up bootcamp.

Also, when using LW5, be sure to save often and save the timing diagrams often. Otherwise, when LW5 eventually breaks something, you may be left with a circuit that will not work, with hardly any evidence of where the break was. If it does break, do not use all your time to fix your circuit, or even try to recreate the whole circuit. You *will* run out of time and be left debating whether to turn it in as is or keep trying to get a better result with a late penalty. It’s better to show where things work and where things don’t.

v. It would be much better for students to receive the lab assignments at least a week in advance to the demos. Especially towards the later labs, assignments would be posted the weekend before the lab was to be demoed. With demos on Tuesday, this can be a problem for students who may live on campus but go home every weekend, as I do.

vi. These labs were a great learning experience. The way that the labs were setup and presented, I got a real good idea of how processor design and computer organization is tackled in real-life applications. These labs also sought to instill good habits, such as creating a JAVA simulator for a proposed ISA to insure that it works as expected.