CSCE 230 Project

Final Report

Group #9

Johnathan Carlson, Tyler Zinsmaster, Jake Ediger

This project details the creation and implementation of a Reduced Instruction Set Computer processor built in VHDL, implemented on an Altera DE1 Cyclone II FPGA development board.

Version	Changes/Additions	Date
Phase II	Added documentation for B-Type, J-type, and D-type instructions	11/2/17-11/8/17
Phase III	Extending documentation for instruction types, as well as the initial assembler	11/13/17-11/22/17
Phase IV	Finishing the basic processor	11/19/17
Phase V and Extra Credit	Finished assembler, Optional Phase V, Bonus IO, Demonstration	11/27/17-11/3

Overview

The goal of this project was to design and implement a 24 bit processor. This process began with designing rudimentary parts in VHDL, a programming language that is used to implement circuits on boards like the Altera DE1 FPGA. Designing a processor on in VHDL involves creating many individual components that combine and interact to perform various logical tasks based on specific input. Of these tasks, there were 4 general types of instructions, each implemented separately- J-type, B-type, R-type, and D-type. After each component was designed, it needed to be tested, integrated, optimized, and retested for the processor to work properly. This project was developed in 5 stages, the contents of which will be explained for the purposes of comprehension and instructional obligation.

Parts added in each phase:

Phase I- Phase I was not a true phase, as all the parts were created prior to the beginning of the project through various labs. The parts included the 16x16 register file, the 16-bit ALU (Arithmetic Logic Unit) that can

produce NZVC flags, and a skeleton for the Control Unit. The Control Unit was only capable of setting flags for R-Type instructions at this point.

Phase II- Integrated the parts created in phase I to create a basic data-path compatible with R-type instructions. Additional parts needed included the IR (instruction register), immediate extender, as well other minor components. The control unit was updated with relevant flags. The adders within the ALU were optimized with carry look-ahead logic.

Phase III- The remaining three types of instructions- J-type, D-type, and B-type, were implemented. A memory interface, as well as the instruction address generator (IAG), were added to the data-path. The control unit was updated to accommodate these new capabilities.

Phase IV-The final required phase, phase IV, was based in implementing I/O. With I/O, a memory interface was needed to be fully implemented, again updating the control unit. In addition, conditional logic for all required flags was fully implemented within the Control Unit.

Phase V and Extra Features- Many bonus features were added to increase the functionality and ease of use of the processor. The Phase V material included Implementing J-Type instructions and expanding the I/O communication capabilities with the board to include the other Hex display panels and red LEDS. This was originally planned to be a required phase but due to time constraints was made bonus material. In addition to the Phase V I/O expansion, the I/O Memory Interface was expanded to allow for GPIO expansion header data, and a component was made to translate binary inputs to the hex display into hex display notation.

Assembler-The assembler was initially created to be a very basic two-pass version to deal with instructions. As it was developed further, it became increasingly valuable for testing purposes. It allowed for efficient testing without re-compilation of the main program, as well catching small errors that would have gone undetected otherwise. The assembler was created in python.

Detailed Processor Development and Implementation

Five Stage Design

Instructions within the processor utilize a 5-stage design in order to ensure proper traversal through the data path for every instruction. Every clock "beat" of a high/low cycle moves an instruction from one stage to the next.

Stage 1: Instruction Fetch: In this stage, the Instruction Register reads a 23-bit value from the main memory, which becomes the Instruction.

Stage 2: Instruction Decode: In this stage, the instruction is passed from the Instruction register into various components, such as the Registry, Immediate block, and Control Unit.

Stage 3: Instruction Execute: In this stage, arithmetic logic is executed through the ALU, and NCVZ flags may be set. If the instruction is B or J type, flags are set within the Instruction Address Generator. All steps Stage 3 and onward can be conditional and are run based on if the pre-existing NCVZ flags within the Control Unit fulfill specified conditional logic.

Stage 4: Memory and I/O: In this stage, an instruction that would interact with memory (determined by Muxma) does so. Based on flags set by the Control Unit, an instruction may either read from or write to memory, and this also includes I/O memory addresses, which are controlled by internal flags within the I/O memory interface.

Stage 5: Write Back: In this stage, MuxY selects a desired value based on flags, which is passed into the registry if the rf_write flag is enabled for the instruction.

Control Unit

The control unit is arguably the most important part of the processor. It dictates all instruction specific logic and controls program flow. The control unit can be thought of as an interpreter for the processor, or, if one would rather, the nervous system that sends signals from the brain to the various organs within one's body. It

translates the individual bits within instructions to usable logic, and sets flags that determine the behavior of the individual components of the processor based on the required 5 steps that make up instruction flow and execution. The control unit was by no means complete at the beginning of phase II, when it was first implemented and used. Now, it has logic for all of the described instructions below, in addition to conditional flag checking which allows for program-level conditionals.

ALU

If the Control Unit is the nervous system of a processor, the ALU is an arm. Based on the ALU_op control flag, the ALU can execute arithmetic logic upon two inputs. It contains a full 16-bit adder, two Inversion selection Muxes, and a final Mux to determine what operation should be chosen, and whether either input should be inverted for subtraction. Though the ALU is set up to support 6 different potential operations, the final implementation only includes 5, as the MULT instruction was not completed. If an instruction is an R or D type instruction, the 15^{th} bit of the instruction is used as a way to determine whether or not the ALU should pass NCVZ flags to the PS Register.

Registry

The Registry is a collection of 16 16-bit registers, used for storing values for use elsewhere in the processor. Based upon inputs, it can be written to, or read from. It uses the rf_write flag to determine whether or not to write a 15 bit data value into a register address specified by the 4 bit input value RegD. It does this through use of a 4 to 16 bit decoder. By checking for the enable bit and the whether the corresponding decoder bit are both true, all 16 individual 16 bit registers decide whether or not to write the incoming data to them. As for the DataS and DataT outputs, the RegS and RegT selection values are placed into a 16 bit multiplexor which determines which register output corresponds to the desired register selection. The value of the R0 register is always 0, and the R15 register is reserved for Link based instructions, so as to make it possible to return to previous dataflow, though it is technically possible to override it through bad assembly code.

Buffer Registers

Based entirely on updating when on the rising edge of a clock, Buffer Registers are used throughout the processor to ensure that instructions follow the 5-stage path. Notably, BuffRegs within the processor act as buffers for information from both registry outputs, BuffRegA and BuffRegB, with BuffRegB outputting to both the MuxB, and the BuffRegM, which outputs to memory. The ALU_output passes through a buffer register into MuxY, and into the address MuxMa. MuxY outputs to BuffRegY, which passes its output into the Registry if rf_write is enabled.

Immediate Block

The immediate block takes in bits 14 down to 8 from the instruction register in stage 2, and, based on the extend flag set in the Control Unit, chooses how to extend this 7 bit string to a 15 bit string. The immediate value sent from this is used in MuxB as the other potential source of data besides the BuffRegB output.

Multiplexors

Throughout the processor are several Multiplexors, or Muxes, used to determine the flow of data based on a select variable. They are essential for proper datapath execution. Besides those mentioned within the descriptions of the various other components, notable multiplexors include MuxC, MuxB, MuxMa, MuxY, and Muxmem. MuxC uses c_select to determine what address of RegD is to be used for writing to the Registry, for R-Types, D-Types, Li, and 15 for linking. MuxB uses b_select to determine whether to use the RegT data output from BuffRegB, or the 15 bit value from the immediate block, to send into the ALU. MuxY determines what source of data should be sent into the BuffRegY and subsequently the Registry based on y_select, from either the ALU, Memory, a ReturnAddress from the Instruction Address Generator, or an Immediate value. MuxMa uses ma_select to determine whether the source of a memory address in stage 4 should be from the ALU or from the Instruction Address Generator. Muxmem uses mem_select from the Control Unit to select whether to use Main or I/O memory data output for MuxY input.

Main Memory

The Main Memory block is a collection of 1024 24 bit memory locations, each with a corresponding 16 bit address. This component was provided to us. The inner workings are not as well understood as other user-defined/modified components, and there was little documentation provided. Its capabilities, however, allow for the implementation of Load Word and Store Word instructions, and for the initialization of instructions outside of a test script. The memory is read from and/or written to based on the mem_read and mem_write flags, and takes in the MuxMa output as a memory address input. It also takes DataM from BuffRegM as a data input, (this is extended to 24 bits from 16 bits with a vhdl AND operator.) The main memory output goes to the Instruction Register in stage 1 as a part of the instruction fetch, and in stage four, the last 16 bits go into Muxmem so as to be potentially selected for input into MuxY.

I/O Memory

I/O memory is based around two things: External Inputs, and External Outputs. This is where the magic happens, so to speak, so that a user can interact with a board on a physical level. The I/O Memory block uses the same mem_write flag and the same memory addresses as the Main Memory, but also checks the first four bits of that address for internal logic. This internal logic allows for the selection of specific input/output pins that are mapped to physical locations on the DE1 board. Loading values from and storing values to these locations interact with visible elements of the board. These interactions and implementations are further detailed in the I/O section. The 15 bit data output goes into Muxmem just as the Main Memory data output.

PS Register

The PS register stores NCVZ flags into the Control Unit based on the ps_enable control flag, which is set based on instruction bit 15 for R and D types in stage 3. These flags are necessary for conditional logic.

Instruction Address Generator

The Instruction Address Generator, or IAG, makes use of a Program Counter, or, PC. This PC is output to MuxMa in stage 4, for potential use in memory. It is called a Counter because the IAG has an adder

component to it which increments the program counter by either 1, or by the address value difference to a label for a B-type instruction, selected from a MuxInc based on the inc_select flag. The pc_enable flag is used to determine when to send the value of the program counter to the MuxMa, and an internal mux called MuxPC uses the pc_select flag to determine whether to make the PC equal to a value from the BuffRegA register value, the Label/constant value, or the MuxInc value. Without this component, program flow would be impossible, and without modifications to the MuxPC, it would not be possible to take values from BuffRegA for J-type instructions. In addition, a buffer register within the IAG outputs to MuxY for potential data output selection.

Instruction Register

The Instruction Register, or IR, is the first step in the datapath. When the ir_enable flag is turned on, the Instruction Register takes in the next 24-bit address from the Main Memory, and upon next clock passes it on to the other components, where its bits are used for various purposes such as determining what registers to read/write from and what control flags to set in subsequent stages.

Instruction Types

- **R-type –** R-type instructions include Add, Sub, AND, OR, XOR, JR, SLL, and CMP, and are all standard instructions. Add, Sub, and CMP make use of adders within the ALU, and these adders are improved from the original design provided in instructions, utilizing carry look-up logical behavior. See "Processor Speed" section for further information. BONUS: SLL outputs the shifted data value by shifting the first input value by the value of the secondary input.
- **D-type –** D-type instructions deal with data transfer between the memory and CPU. These instructions include load word, store word, add immediate, sub immediate,, and Shift Left Logical Immediate. Add immediate and sub immediate utilize the alu by having their immediate values selected from a mux into the second alu input, rather than from the registry. Sub immediate is technically implemented within the hardware, but is unused in programs created through the assembler. This is because for sub immediate instructions, the assembler automatically inverts the value of the immediate and outputs an equivalent

add immediate instruction. The sub immediate hardware instruction works in cases where it is tested. The load word and store word instructions allow for the storing and loading of values from memory addresses. BONUS: Shift Left Logical Immediate (SLLI), works just as SLL does within R-Types, but instead of using a register value for the second input, the ALU takes the immediate value in from MuxB.

- **B-type-** B-type instructions deal with branching, the main way loops and functions can be implemented in assembly. Instructions include Branch and Branch and link. These utilize the loading of addresses into the IAG, based on the address of a label to which the instruction branches. Branch and link additionally saves the previous address to R15 for return at a later point in a program.
- **BONUS: J-type-** J-type instructions involve jumping commands. Jump, jump and link, and load immediate are all J-type instructions. Within the processor, J-type instructions utilize flags within the Control Unit at the different stages. This did not require major restructuring of the Control Unit or the rest of the design, as it was made with J-types in mind. However, Load Immediate necessitated an additional selection for the Multiplexor that chooses which register to save the MUX Y output into (assuming the corresponding flag for writing to the registry is true in this stage, which it is), due to the location of the output-register-specifying-bits within the instruction. J types also differ from B types in that while they both set flags that determine behavior of the Instruction Address Generator, J types do so a stage earlier than B types. This is because B types tend to require the address immediately after the specified branch address to be loaded next, so the instructions allow for the Instruction Address Generator to use its internal adder to increment the label address by 1 before outputting. J types need to jump directly to the address, as there is no address set by a label to which you would need to increment past. Jump and Link also saves the previous address to R15 for return at a later point in a program.

I/O

Within Phase IV, the I/O Memory Interface was implemented, and with this, as one could guess, came basic I/O capabilities. Green LEDs, Push Buttons, Slider Switches, and the first out of 4 7-segment Hex display panels were implemented. The logic for these I/O devices within the I/O memory interface already existed within provided files, so the initial hurdle was to implement the logic outside of the interface. The main

difficulty regarding this was that, since the control flag to write to memory is the same for both normal Memory and I/O memory, you need a multiplexor in order to select the proper output data for use in the multiplexor that sends data through a buffer and then into the Registry. This multiplexor uses a control flag in stage 4 to determine what output to use, but this is a problem in itself. The way that one can distinguish addresses that interact with normal memory from those that interact with I/O is a 4-bit "Key" made up of the first four bits of the address passed into the memory interfaces. The answer seemed simple: pass the first four bits of the address to the Control Unit when it is determined from the address-selecting multiplexor, check within the Control Unit to see whether it is an I/O or regular memory address, and have the Control Unit output the proper selection flag in stage 4. However, this does not work, since the Control Unit is clock gated, you cannot give it data, have it perform logic, and output flags dependent on the data all in the same stage. You cannot get the address before stage 4, because the multiplexor that selects it does not have the proper output until then. The workaround that was implemented to fix this, is to take advantage of the fact that the above statement isn't strictly true. You can get the address before stage four, or at least, before what *would* be the address if the operation specified passes addresses into memory. How? Operations that interact with memory addresses that could potentially involve I/O require passing the address through the ALU. Thus, send the ALU_OUTPUT from stage 3 into the control unit, and then have the proper memory selection flag in stage 4. Concerns existed regarding whether this could cause outside issues, but looking through the data path for the processor, and all implemented instructions, it was determined that this would not be problematic. To clarify, however, the four-bit key that goes into the I/O memory itself comes from the address multiplexor output. The ALU_OUTPUT bits only go to the Control Unit in this scenario.

BONUS: During Phase V, the optional red LEDs and other three Hex display panels were added to the I/O Memory Interface and mapped to proper pins. In addition, a component was made and then placed within the interface to convert any binary value of up to four bits into a 7 bit equivalent Hex display value. Any bits past the fourth bit are ignored for the translation as the Hex display implementation has been set up for up for hex values from 0 to F for each panel. This decision to translate binary inputs within the VHDL saved what would have potentially been several hours of work and assembly code resources, by removing the need for in-program

translation and mapping every time one would want to use the Hex display. The benefit of this beyond the obvious time factor is that it saves vital assembly language and device memory resources, which increases available space for complexity of a program significantly.

There was interest in expanding I/O capabilities as much as possible without running out of potential 4-bit I/O memory identifier combinations. Using the DE1 reference manual as a guide, 4 GPIO Expansion header pins were mapped for additional functionality. One pin output the clock status for debugging, to ensure that the clock continued to work throughout tests. Another was an input pin, the use of which is as various in function as it is limited in scope, taking in either a 1 or a 0. Then, an output pin analogous to the input pin. Lastly, the reset pin was moved from the initial mapping of the first pushbutton, to a GPIO header pin. This allowed for greater variety of input from the pushbuttons while maintaining general reset capabilities. Though the GPIO in/out pins are largely unused, a red LED was connected to the output pin as a proof of concept to show that it works.

Testing:

The Altera software ModelSim was used to create waveforms based upon the processor outputs for given tests cases. A .do script file shows which waveforms will be displayed as well as setting certain input output variables that would not be set from memory, such as clock, reset, and some I/O functionality.

Modelsim is built into Quartus.

Bonus: Assembler

The assembler had already been partially completed and had limiting functionality prior to phase IV. During phase IV, the rest of the assembler was completed. The assembler was designed in python, and proved to be a valuable tool in testing as well as implementation.

The assembler used is known as a two-pass assembler, which was necessary to be sure everything was correct. It outputs a .mif file, as well as an organized debugging interface. An example of a mif file is further down the report. The assembler was versatile in that any file could be run and tested, without the need to be redesigned for each new test. It stores the first 8 memory addresses in an array for use with I/O. A clean output

provided easy debugging and testing. A variety of test files were used in the assembler, each testing a certain phase. Some phases required more than 1 test file to ensure correctness.

```
--------------------------------------------------------------------------------
 Assembler for CSCE230 project
 Made by Johnathan Carlson, Tyler Zinsmaster, Jake Ediger
--------------------------------------------------------------------------------
.main @0x9
.....end @0xe
.
--------------------------------------------------------------------------------
                    IODEF 00000000000000000000000 [IO] 0x0      Address: 0x0
BRANCH OVER DATA >  IODEF 10000000000000000000110 [IO] 0x800006 Address: 0x1
SLIDER SWITCHES ->  IODEF 00000000100000000000000 [IO] 0x8000   Address: 0x2
PUSH BUTTONS ----->  IODEF 00000000010000000000000 [IO] 0x4000   Address: 0x3
7-SEG DISPLAY ---->  IODEF 00000000001000000000000 [IO] 0x2000   Address: 0x4
GREEN LEDS ------>  IODEF 00000000000100000000000 [IO] 0x1000   Address: 0x5
                    IODEF 00000000000000000000000 [IO] 0x0      Address: 0x6
                    IODEF 00000000000000000000000 [IO] 0x0      Address: 0x7
add r0 r0 r0        RType 00000000010000000000000 [OK] 0x004000 Address: 0x8
main @0x9
addi r1 r0 8        DType 01100000000010000000001 [OK] 0x600801 Address: 0x9
jr r1               RType 00010000000000000010000 [OK] 0x100010 Address: 0xa
addi r1 r0 4        DType 01100000000001000000001 [OK] 0x600401 Address: 0xb
subi r1 r1 8        DType 01100000011110000010001 [OK] 0x607811 Address: 0xc
addi r1 r0 9        DType 01100000000010010000001 [OK] 0x600901 Address: 0xd
end @0xe
br end              BType 10000000111111111111111 [OK] 0x80FFFF Address: 0xe  Branch up: 1 (0xffff)
Filling Extra Memory: 0x400-0xf
--------------------------------------------------------------------------------
~~~~~~~~~~~~~~~~~~Assembly Compeleted~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--------------------------------------------------------------------------------
```

Figure 1: This image is a sample output of the assembler. The data is from one of the test files used in phase IV to test I/O with the FPGA, condensed to save room.

Speed Overview

When flashed to the Altera DE1 FPGA board, the processor has a frequency, or, clock speed, of 50MHz. This is the fastest clock speed available for use on the board without a signal generator. Within ModelSim, however, the processor functions properly with a minimum clock high/low cycle period 334ps. This is equivalent to a frequency of 2.994*10^9 Hz, which is 2.994GHz. So, theoretically the upper stable limit of our processor is just ever so slightly shy of the 3 GHz mark. This is a factor of 59.88 times faster than the fastest provided clock pin from the DE1 board. Given that the processor must work through 5 stages, it takes 5 clock beats for each instruction. Thus, the instructions per second is 598800000. For most tasks that this processor would be applied to, however, such an increase in clock speed would be of minimal benefit. In fact, due to the lack of a real-time clock, such a high clock speed would be of detriment to desired behavior in various scenarios. One such scenario would be the nightrider game used in the visual demonstration of the board, which depends on delay set through looped branching in order to be able to “capture” a flashing light within possible human

reaction time. With a higher clockspeed, the program would necessarily become more complicated to accommodate, and for a program that would rely upon similar timings, one which uses up most of the board's available resources already, this could create a real constraint on the available 1024 spaces in memory.

BONUS: Instead of using the form of adders detailed in early phase instructions, the processor implements faster addition via carry look-ahead.

With carry look-ahead addition logic, it is more efficient than regular logical addition for up to 4 bits at a time. Knowing this, our adder has four separate carry look-ahead mappings of 4 bits each for maximum efficiency. For a 4 bit carry look-ahead addition function, the logic is as such:

NOTE: s_(i) is sum bit, x_(i) is first bit, y_(i) is added bit, and c_(i) is initial carry bit

G is gen, P is prop, and the reason they are useful is that the final sum is the initial halfsum (xi XOR yi) XOR CarryIn. The Gen and Prop components allow for the simplified finding of the CarryIn logically.

```vhdl
entity FAST4BITADDER is
   port(
        A : in std_logic_vector(3 downto 0);
        B : in std_logic_vector(3 downto 0);
        CarryIn : in std_logic;
        S : out std_logic_vector(3 downto 0);
        CarryOut : out std_logic
   );
end FAST4BITADDER;


--si = xi ⊕ yi ⊕ ci
--ci+1 = xiyi + xici + yici
--ci+1 = xiyi + (xi + yi)ci
--ci+1 = Gi + Pici
--Gi = xiyi and Pi = xi + yi
--ci+1 = Gi + PiGi-1 + PiPi-1ci-1

architecture CarryLookAheadAdd of FAST4BITADDER is
signal halfsum : std_logic_vector(3 downto 0);
signal gen : std_logic_vector(3 downto 0);
signal prop : std_logic_vector(3 downto 0);
signal C : std_logic_vector(3 downto 0);

begin

    halfsum <= A XOR B;
    gen <= A AND B;
    prop <= A OR B;
         C(1) <= gen(0) OR (prop(0) AND CarryIn);
         C(2) <= gen(1) OR (prop(1) AND C(1));
         C(3) <= gen(2) OR (prop(2) AND C(2));
      CarryOut <= gen(3) OR (prop(3) AND C(3));
    S(3 downto 1) <= C(3 downto 1) XOR halfsum(3 downto 1);
    S(0) <= halfsum(0) XOR CarryIn;


end CarryLookAheadAdd;
```

Figure 2: VHDL for the 4-bit Carry Look-Ahead adder implementation.

MIF File

An example .mif file, discussed above in the *Assembler* section.

```
WIDTH=24;
DEPTH=1024;
ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;
CONTENT BEGIN
 0: 000000000000000000000000;
 1: 100000000000000000001100;
 2: 000000001000000000000000;
 3: 000000000100000000000000;
 4: 000000000010000000000000;
 5: 000000000101000000000000;
 6: 000000000110000000000000;
 7: 000000000111000000000000;
 8: 000000000001000000000000;
 9: 000000000011000000000000;
 10: 000000001110000000000000;
 11: 000000001111000000000000;
 12: 000000000000000000000000;
 13: 000000000000000000000000;
 14: 010000000000101000000010;
 15: 010000000000100000000011;
 16: 011000000000000100000101;
 17: 011000000000000000000110;
 18: 010000000000000000100101;
 19: 010100000000000000110101;
 20: 100000001111111111111101;
[20..1023] : 000000000000000000000000;
```

Group Experiences throughout project

Despite constant hardship and difficulty, the project was completed on time and with proper behavior at each stage. Many a night of sleep was lost, tensions were high, and compilation was just another name for impatience.

Immediately, the group understood that when it came down to crunch time, an assembler would be vital for proper testing and programming functionality. Thus, even in stage 1, the assembler was always kept in mind

and improved upon until the final iteration was kept. This saved massive amounts of time in all stages 3 and up, and without it, it is certain that multiple checkoff dates would have been missed.

There was anomalous behavior upon the first attempts at manipulating IO beyond lighting a single LED, and after a few hours of testing, it became apparent that output pins used for debugging purposes within the top-level processor VHDL file were being mapped erroneously upon compilation and flashing to the board. Upon removal of these pins, further unwanted behavior was observed. However, this was rectified quickly and with due diligence, thanks to frustrated button-mashing. It was found that the inputs from the push buttons, while low by default on testing simulation software, was high by default when mapped as input pins to the board. This was true for all input pins on the physical hardware, and thus, simple inversion allowed for correct output.

At one point during stage 3, there was a large level of confusion regarding the datapath and how each instruction flowed through the processor. Thus, a barely legible scribble of the top level datapath was created, and used for about a day before the group became fed up and created a neater one, which was promptly lost, leaving only the horrendous drawing.

Original plans for the header pins were audacious and involved transmission of signals between two or more development boards over radio frequencies. Due to time constraints and high levels of radio interference over the bands the transmitter/receiver worked within, this idea was discarded. It is still possible to have a wired single-bit communication channel between two boards, but this is not used in the demonstration due to fear of damaging one or both boards, and also due to having little use within the scope of the final demonstration.

Conclusion

Through every challenge, holdup, and unknown behavior, despite incredible odds and time constraints and setbacks and sleepless nights, the processor exists and works. All behaviors are well understood at this point, and through demonstration, presentation, and extensive testing, it is clear that the pain was not in vain. The experience of designing this 24 bit NIOS II style processor is not one that will soon be forgotten, if ever. The knowledge gained about processor function will be invaluable in the careers of those within the group.

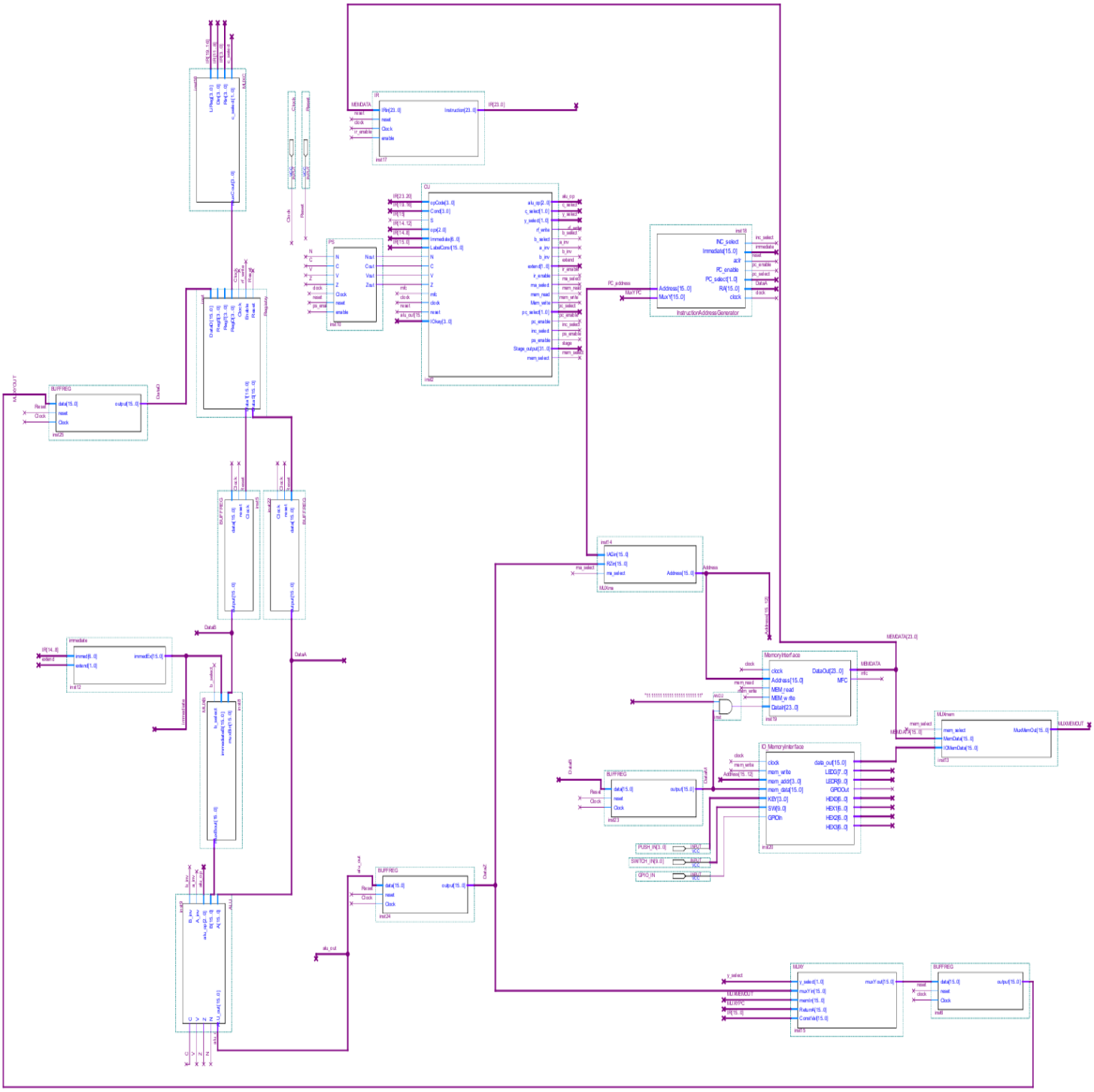
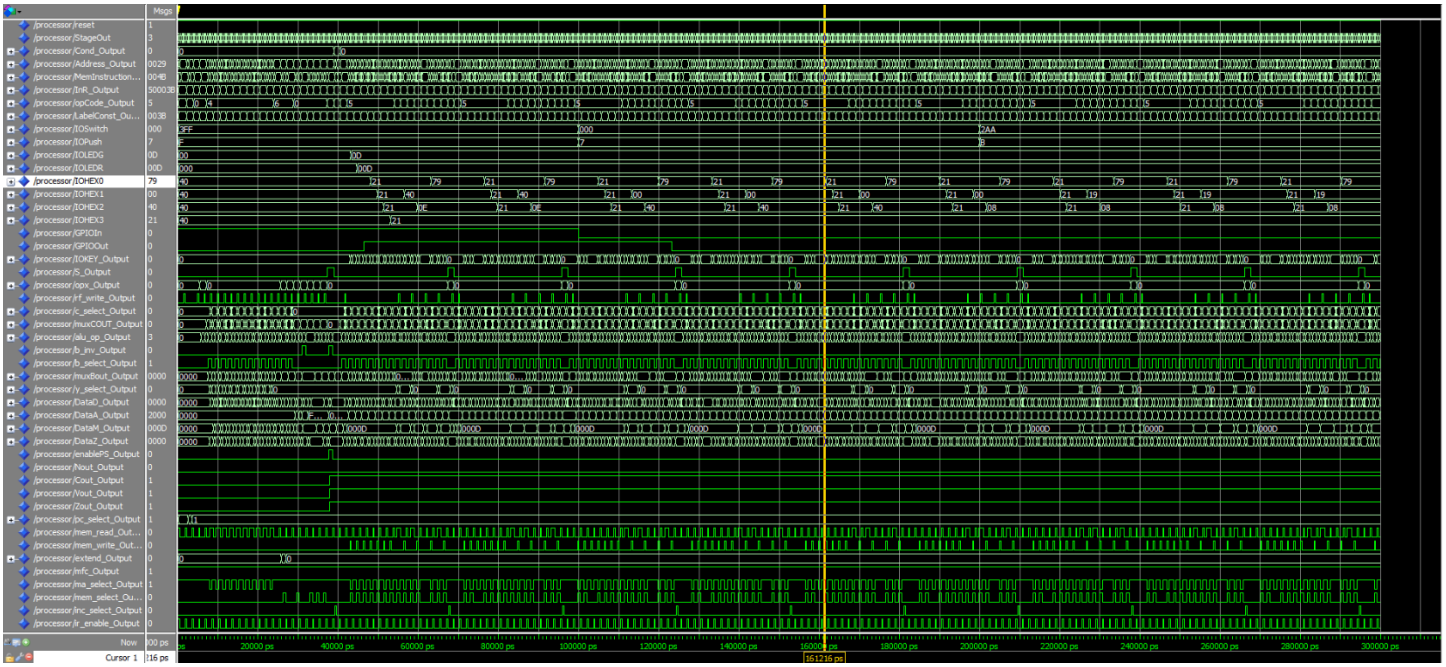
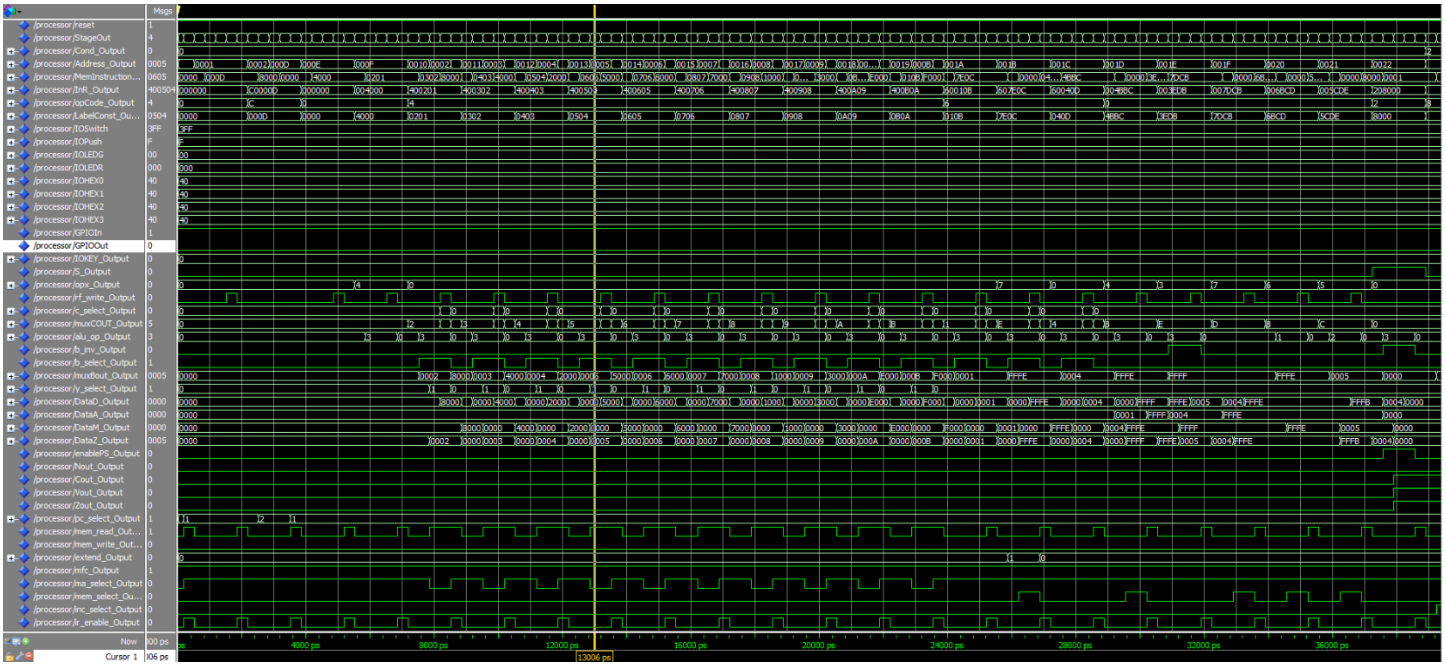


Figure 3: Full processor top level structure with pins as a block diagram.


/processor/reset
/processor/StageOut
/processor/Cond_Output
/processor/Address_Output
/processor/MemInstruction...
/processor/InR_Output
/processor/opCode_Output
/processor/LabelConst_Ou...
/processor/IOSwitch
/processor/IOPush
/processor/IOLEDG
/processor/IOLEDR
/processor/IOHEX0
/processor/IOHEX1
/processor/IOHEX2
/processor/IOHEX3
/processor/GPIOIn
/processor/GPIOOut
/processor/IOKEY_Output
/processor/S_Output
/processor/opx_Output
/processor/rf_write_Output
/processor/c_select_Output
/processor/muxCOUT_Output
/processor/alu_op_Output
/processor/b_inv_Output
/processor/b_select_Output
/processor/muxBout_Output
/processor/y_select_Output
/processor/DataD_Output
/processor/DataA_Output
/processor/DataM_Output
/processor/DataZ_Output
/processor/enablePS_Output
/processor/Nout_Output
/processor/Cout_Output
/processor/Vout_Output
/processor/Zout_Output
/processor/pc_select_Output
/processor/mem_read_Out...
/processor/mem_write_Out...
/processor/extend_Output
/processor/mfc_Output
/processor/ma_select_Output
/processor/mem_select_Ou...
/processor/inc_select_Output
/processor/ir_enable_Output
Now
Cursor 1
20000 ps
40000 ps
60000 ps
80000 ps
100000 ps
120000 ps
140000 ps
160000 ps
180000 ps
200000 ps
220000 ps
240000 ps
260000 ps
280000 ps
300000 ps
161216 ps


/processor/reset
/processor/StageOut
/processor/Cond_Output
/processor/Address_Output
/processor/MemInstruction...
/processor/InR_Output
/processor/opCode_Output
/processor/LabelConst_Ou...
/processor/IOSwitch
/processor/IOPush
/processor/IOLEDG
/processor/IOLEDR
/processor/IOHEX0
/processor/IOHEX1
/processor/IOHEX2
/processor/IOHEX3
/processor/GPIOIn
/processor/GPIOOut
/processor/IOKEY_Output
/processor/S_Output
/processor/opx_Output
/processor/rf_write_Output
/processor/c_select_Output
/processor/muxCOUT_Output
/processor/alu_op_Output
/processor/b_inv_Output
/processor/b_select_Output
/processor/muxBout_Output
/processor/y_select_Output
/processor/DataD_Output
/processor/DataA_Output
/processor/DataM_Output
/processor/DataZ_Output
/processor/enablePS_Output
/processor/Nout_Output
/processor/Cout_Output
/processor/Vout_Output
/processor/Zout_Output
/processor/pc_select_Output
/processor/mem_read_Out...
/processor/mem_write_Out...
/processor/extend_Output
/processor/mfc_Output
/processor/ma_select_Output
/processor/mem_select_Ou...
/processor/inc_select_Output
/processor/ir_enable_Output
Now
Cursor 1
4000 ps
8000 ps
12000 ps
16000 ps
20000 ps
24000 ps
28000 ps
32000 ps
36000 ps
13006 ps

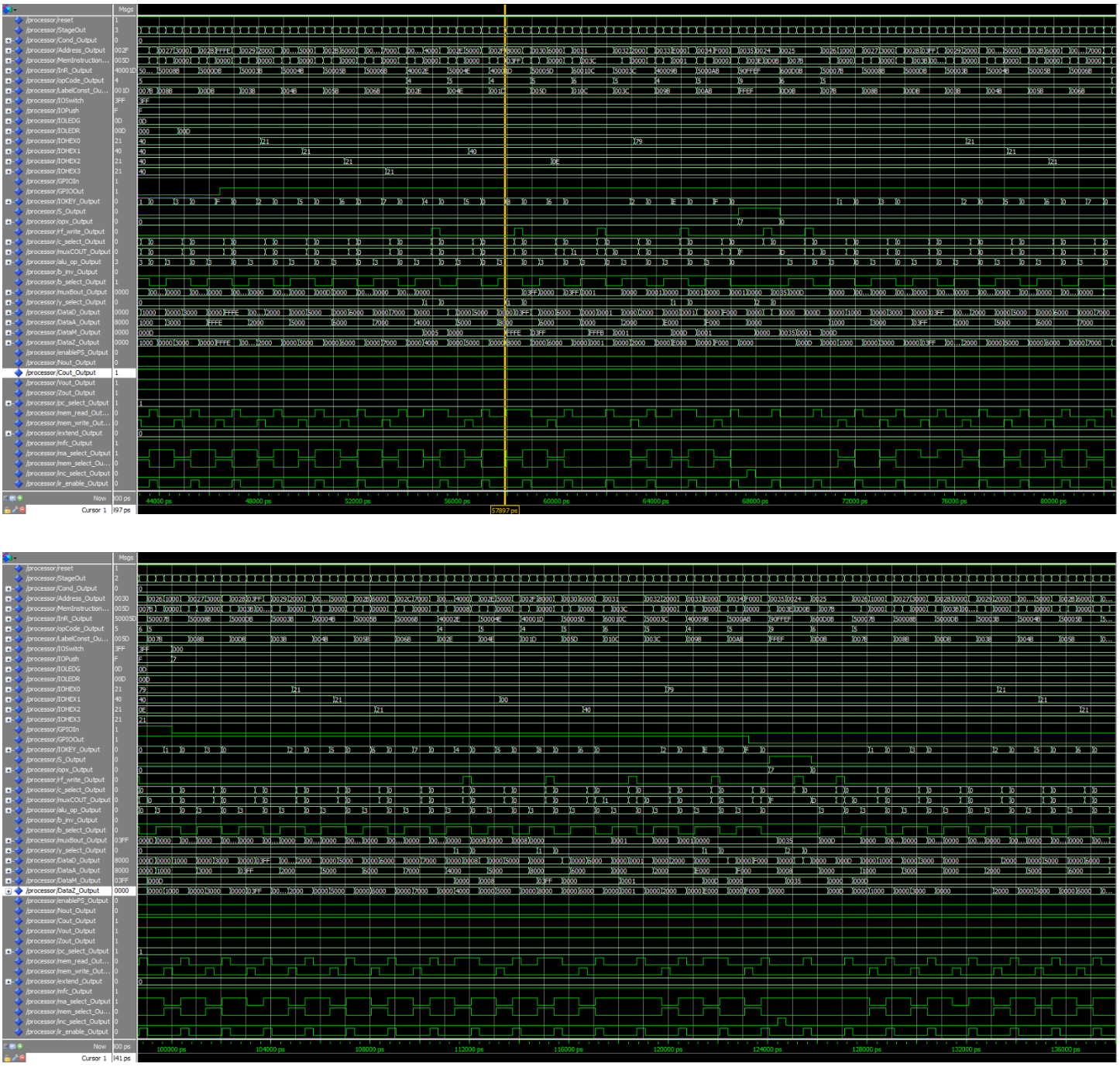


Figure 4: Waveforms for a fully comprehensive testing suite utilizing all components.

Testing Suite:

```
force reset 1 0
force clock 0 0, 1 167  -repeat 334
force IOPush 1111 0, 0111 100000, 1011 200000
force IOSwitch 1111111111 0, 0000000000 100000, 1010101010 200000
force GPIOin 1 0, 0 100000
run 300000
```

IODEF 00000000000000000000000 [IO] 0x0 Address: 0x0

JUMP OVER DATA --> IODEF 11000000000000000000001101 [IO] 0xc0000d Address: 0x1

SLIDER SWITCHES -> IODEF 00000000100000000000000 [IO] 0x8000 Address: 0x2

PUSH BUTTONS ----> IODEF 000000000100000000000000 [IO] 0x4000 Address: 0x3

7-SEG DISPLAY0 --> IODEF 000000000010000000000000 [IO] 0x2000 Address: 0x4

7-SEG DISPLAY1 --> IODEF 000000000101000000000000 [IO] 0x5000 Address: 0x5

7-SEG DISPLAY2 --> IODEF 000000000110000000000000 [IO] 0x6000 Address: 0x6

7-SEG DISPLAY3 --> IODEF 000000000111000000000000 [IO] 0x7000 Address: 0x7

GREEN LEDS ------> IODEF 000000000001000000000000 [IO] 0x1000 Address: 0x8

RED LEDS --------> IODEF 000000000011000000000000 [IO] 0x3000 Address: 0x9

GPIO IN ---------> IODEF 000000001110000000000000 [IO] 0xe000 Address: 0xa

GPIO OUT --------> IODEF 000000001111000000000000 [IO] 0x1000 Address: 0xb

MAX VAL ---------> IODEF 111111111111111111111111 [IO] 0xffffff Address: 0xc

IODEF 000000000000000000000000 [IO] 0x0 Address: 0xd

add r0 r0 r0 RType 000000000100000000000000 [OK] 0x004000 Address: 0xe

ldw r1 2 r0 DType 010000000000001000000001 [OK] 0x400201 Address: 0xf

ldw r2 3 r0 DType 010000000000001100000010 [OK] 0x400302 Address: 0x10

ldw r3 4 r0 DType 010000000000010000000011 [OK] 0x400403 Address: 0x11

ldw r4 5 r0 DType 010000000000010100000100 [OK] 0x400504 Address: 0x12

ldw r5 6 r0 DType 010000000000011000000101 [OK] 0x400605 Address: 0x13

ldw r6 7 r0 DType 010000000000011100000110 [OK] 0x400706 Address: 0x14

ldw r7 8 r0 DType 010000000000100000000111 [OK] 0x400807 Address: 0x15

ldw r8 9 r0 DType 010000000000100100001000 [OK] 0x400908 Address: 0x16

ldw r9 10 r0 DType 010000000000101000001001 [OK] 0x400A09 Address: 0x17

ldw r10 11 r0 DType 010000000000101100001010 [OK] 0x400B0A Address: 0x18

test @0x19

addi r11 r0 1 DType 011000000000000100001011 [OK] 0x60010B Address: 0x19

subi r12 r0 2 DType 011000000111111000001100 [OK] 0x607E0C Address: 0x1a

addi r13 r0 4 DType 011000000000010000001101 [OK] 0x60040D Address: 0x1b

add r11 r11 r12 RType 000000000100101110111100 [OK] 0x004BBC Address: 0x1c

sub r14 r13 r11 RType 000000000011111011011011 [OK] 0x003EDB Address: 0x1d

and r13 r12 r11 RType 000000000111110111001011 [OK] 0x007DCB Address: 0x1e

or r11 r12 r13 RType 000000000110101111001101 [OK] 0x006BCD Address: 0x1f

xor r12 r13 r14 RType 000000000101110011011110 [OK] 0x005CDE Address: 0x20

cmp r0 r0 RType 001000001000000000000000 [OK] 0x208000 Address: 0x21

beq cond BType 100000100000000000000001 [OK] 0x820001 Address: 0x22 Branch down: 1 (0x1)

jr r15 RType 000100000000000011110000 [OK] 0x1000F0 Address: 0x23

cond @0x24

addi r11 r0 13 DType 011000000000110100001011 [OK] 0x600D0B Address: 0x24

stw r11 0 r7 DType 010100000000000001111011 [OK] 0x50007B Address: 0x25

stw r11 0 r8 DType 010100000000000010001011 [OK] 0x50008B Address: 0x26

stw r11 0 r13 DType 010100000000000011011011 [OK] 0x5000DB Address: 0x27

stw r11 0 r3 DType 010100000000000000111011 [OK] 0x50003B Address: 0x28

stw r11 0 r4 DType 010100000000000001001011 [OK] 0x50004B Address: 0x29

stw r11 0 r5 DType 010100000000000001011011 [OK] 0x50005B Address: 0x2a

stw r11 0 r6 DType 010100000000000001101011 [OK] 0x50006B Address: 0x2b

ldw r14 0 r2 DType 010000000000000000101110 [OK] 0x40002E Address: 0x2c

stw r14 0 r4 DType 010100000000000001001110 [OK] 0x50004E Address: 0x2d

ldw r13 0 r1 DType 010000000000000000011101 [OK] 0x40001D Address: 0x2e

stw r13 0 r5 DType 010100000000000001011101 [OK] 0x50005D Address: 0x2f

addi r12 r0 1 DType 011000000000000100001100 [OK] 0x60010C Address: 0x30

stw r12 0 r3   DType 0101000000000000000111100 [OK] 0x50003C Address: 0x31

ldw r11 0 r9   DType 0100000000000000010011011 [OK] 0x40009B Address: 0x32

stw r11 0 r10   DType 0101000000000000010101011 [OK] 0x5000AB Address: 0x33

bal cond   BType 10010000111111111101111 [OK] 0x90FFEF Address: 0x34 Branch up: 17 (0xffef)

jalnv @0x35

li r1 62   JType 11100001000000000111110 [OK] 0xE1003E Address: 0x35

jal 55   JType 11010001000000000110111 [OK]28 0xD10037 Address: 0x36

nop   NO OP 00000000010000000000000 [OK] 0x004000 Address: 0x37

ldw r2 3 r0   DType 0100000000000001100000010 [OK] 0x400302 Address: 0x38

ldw r7 4 r0   DType 0100000000000010000000111 [OK] 0x400407 Address: 0x39

ldw r8 5 r0   DType 0100000000000010100001000 [OK] 0x400508 Address: 0x3a

ldw r9 6 r0   DType 0100000000000011000001001 [OK] 0x400609 Address: 0x3b

ldw r10 7 r0   DType 0100000000000011100001010 [OK] 0x40070A Address: 0x3c

ldw r3 8 r0   DType 0100000000000100000000011 [OK] 0x400803 Address: 0x3d

ldw r13 9 r0   DType 0100000000000100100001101 [OK] 0x40090D Address: 0x3e

ldw r14 2 r0   DType 0100000000000001000001110 [OK] 0x40020E Address: 0x3f

ldw r1 12 r0   DType 0100000000000110000000001 [OK] 0x400C01 Address: 0x40

addi r5 r0 4   DType 0110000000000010000000101 [OK] 0x600405 Address: 0x41

addi r11 r0 14 DType 0110000000000111000001011 [OK] 0x600E0B Address: 0x42

addi r12 r0 10 DType 0110000000000101000001100 [OK] 0x600A0C Address: 0x43

loop @0x44

ldw r4 0 r2   DType 0100000000000000000100100 [OK] 0x400024 Address: 0x44

ldw r6 0 r14   DType 0100000000000000011100110 [OK] 0x4000E6 Address: 0x45

stw r6 0 r13   DType 0101000000000000011010110 [OK] 0x5000D6 Address: 0x46

stw r5 0 r3   DType 0101000000000000000110101 [OK] 0x500035 Address: 0x47

stw r0 0 r7   DType 0101000000000000001110000 [OK] 0x500070 Address: 0x48

stw r0 0 r8   DType 0101000000000000010000000 [OK] 0x500080 Address: 0x49

stw r0 0 r9   DType 0101000000000000010010000 [OK] 0x500090 Address: 0x4a

stw r0 0 r10   DType 0101000000000000010100000 [OK] 0x5000A0 Address: 0x4b

cmp r5 r4   RType 00100000100000000001010100 [OK] 0x208054 Address: 0x4c

beq leds   BType 10000010000000000000000001 [OK] 0x820001 Address: 0x4d Branch down: 1 (0x1)

br loop   BType 10000000111111111110101 [OK] 0x80FFF5 Address: 0x4e Branch up: 11 (0xfff5)

leds @0x4f

addi r6 r0 13   DType 0110000000000110100000110 [OK] 0x600D06 Address: 0x4f

stw r6 0 r3   DType 0101000000000000000110110 [OK] 0x500036 Address: 0x50

stw r6 0 r7   DType 0101000000000000001110110 [OK] 0x500076 Address: 0x51

stw r0 0 r13   DType 0101000000000000011010000 [OK] 0x5000D0 Address: 0x52

stw r12 0 r8   DType 0101000000000000010001100 [OK] 0x50008C Address: 0x53

stw r11 0 r9   DType 0101000000000000010011011 [OK] 0x50009B Address: 0x54

stw r6 0 r10   DType 0101000000000000010100110 [OK] 0x5000A6 Address: 0x55

ldw r4 0 r2   DType 0100000000000000000100100 [OK] 0x400024 Address: 0x56

cmp r5 r4   RType 00100000100000000001010100 [OK] 0x208054 Address: 0x57

beq leds   BType 10000010111111111110110 [OK] 0x82FFF6 Address: 0x58 Branch up: 8 (0xfff6)

br loop   BType 10000000111111111101010 [OK] 0x80FFEA Address: 0x59 Branch up: 20 (0xffea)

nop   NO OP 00000000010000000000000 [OK] 0x004000 Address: 0x5a