Project Report
Zeus x16

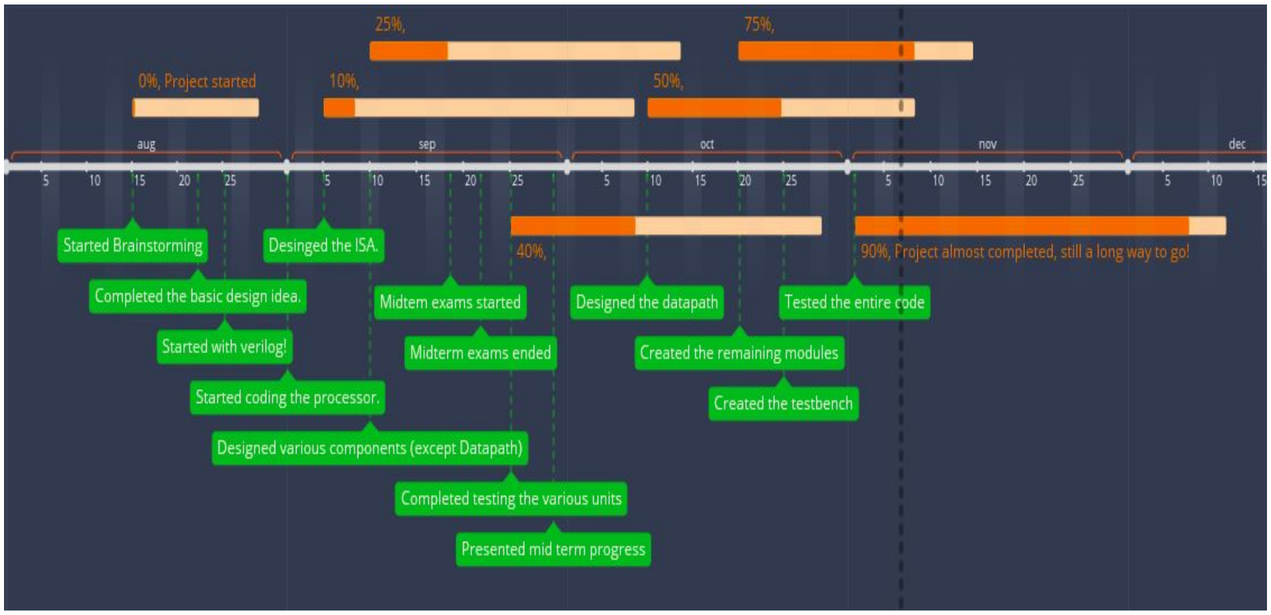
Team Members

16116061	Shaddy Garg
16116065	Shubham Maheshwari
16116068	Surya Saini
16116071	Swapnil Negi
16116076	Vishal Sharma
16116078	Yash Agrawal

Contents

Timeline

We were assigned the project in early August and after research for about 2 weeks we came up with strategy of dividing our group into two sub groups handling different components of our project with an aim to increase the efficiency of our work two folds. We all had frequent meetings updating ourselves with the current status of “Zeus - our very own processor” that we designed with an aim to learn the concepts of processor design and development. A detailed timeline is mentioned below, we aimed to stick at the proposed timeline but often deviated from the same, but due to our collective efforts and hardwork were able to achieve what we had imagined.



-- Project’s Timeline --

Abstract

Zeus x16 is a 16 bit microprocessor having an implementation of the powerful RISC instruction set architecture. Zeus helped us get the insight of the how the instructions are really processed by the machine, how the complex c++ codes we write are actually implemented by the machine. The major motivation behind the design was getting to know how simplified design can increase the efficiency of work. The simple ISA design helps us reduce the hardware cost many folds.

The exceptional feature of Zeus is its **own simplified assembler** which takes an input file from the user in a file format and give the processed output.

Zeus is aimed at being **highly user friendly**, it gives its users the utmost priority.

The report encompasses the entire experience of the developers working on it, right from building the ISA design to structuring the entire datapath, testing the code improving it one bit at a time.

Hope you enjoy coding with Zeus as much as we enjoyed building it.

Meet the new Generation Processor

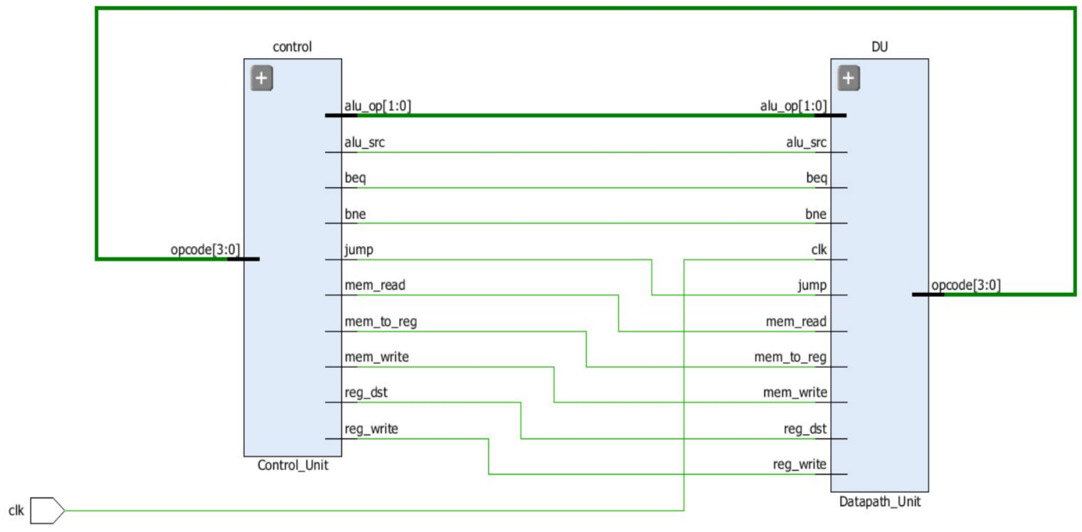
"All men of action are dreamers"

The name of our processor is inspired from the almighty Greek God "Zeus". Zeus is a fully functioning 16 bits processor that we developed with an aim to get a greater insight of the concepts of computer architecture and design.
The Concepts we studied in lectures, reference from various reading materials substantially increased our understanding of microprocessors, the philosophy behind their designs and working.

Zeus, the most powerful Greek God signifies the strength of our processor. Implementing a different ISA (16 bit) from the standard 32 bit RISC processor.

- With frequent group meetings we came up with the Instruction Set Architecture (ISA), the backbone of our processor. The ISA we came up with was different even from the standard 16 bit design and thus we were open to new possibilities to work upon as well as open to new issues that were bound to gives us hard time solving but in the process we were able to learn a lot about the internal design and how various components are connected to one another and how they interact with each other to produce the marvelous results we generally take for granted everyday.
- We came to appreciate the amount of hard work that had been passed down from one generation of designers to another and their brilliance that lead to the development of microprocessors. In honor of the legacy of microprocessors we poured every ounce of our knowledge and understanding into the development of Zeus.
- The next step was mastering Verilog that we had a brief experience thanks to the digital logic design course. We got on to implementing the datapath in verilog with testing at each stage by forcing the inputs and verifying the output with expected result.

- The next step in which we faced many difficulties was the phase of integration of various parts of the processor with each other. It was a very tedious task to integrate the parts with each other because different people followed different conventions of naming the variables. We rectified all the errors and finally got the processor running.
- Finally comes the testing phase. This was one of the most rigorous process because we had strict timelines and we needed to write all the tests for running the processor. We finally settled on three sets of instructions which could test all the capabilities of the processor on full strength.



Processor Overview

Motivation

The main aim of our project was work on something new, something highly fascinating, something that is aimed at increasing the efficiency of work, so designing a RISC processor was a perfect idea.

The motivation was getting to know how the basics of computer architecture, one of the most important and interesting aspect of computer science.

The goal of our work revolved around:-

- Design the reduced instruction set architecture for our processor.
- Design the datapath for the processor.
- Design the assembler for making the experience more user friendly.
- Study the concepts of pipelining and their scope of inclusion in our design.

Objectives

1. Instruction Set Architecture.
    - Designing a unique 16 bit ISA.
    - Deciding the functionality of ALU
    - Designing the ALU Control signals and Control Unit signals corresponding to each function
    - Implementing the above mentioned in Verilog
2. Datapath design.
    - Linking the components via a self designed datapath

3.) Report writing and preparing a presentation to describe Zeus.

ISA Design

ISA(Instruction Set Architecture) is the backbone of Zeus. Maximum amount of thought was given into designing of the architecture set and we began by exploring various ISAs, there benefits and drawbacks ruling out the possibilities based upon our goals.

Instruction Set Architecture :-

- No. of register: 8
- Length of registers: 16 bits

1.) Memory Access(Load/Store):

OP (4 bits)	Rs1 (3 bits)	Ws (3 bits)	Offset (6 bits)

1. OP :- To decide the format of instruction.
2. Rs1 :- Mentions the register in which the data will stored or loaded from the memory.
3. Ws :- Stores the data that will be stored in the memory
4. Offset :- Sets the offset which helps in deciding the address of the register from which to Load/Store

Sample instructions :-

1. Load Word:

    LD ws, offset(rs1) ws:=Mem16[rs1 + offset]

2. Store Word:

    ST rs2, offset(rs1) Mem16[rs1 + offset]=rs2

2.) Data Processing :-

OP (4 bits)	Rs1 (3 bits)	Rs2 (3 bits)	Ws (3 bits)	Useless (3 bits)

1. OP :- Decides the format of instruction.
2. Rs1 :- Stores Value of argument 1.
3. Rs2 :- Stores Value of argument 1.

4. Ws :- Stores the calculated value.
5. Useless :- Use of bits controversial.

Sample instructions :-

1. ADD:

   ADD ws, rs1, rs2 ws:=rs1 + rs2

2. LOGICAL SHIFT RIGHT:

   LSR ws, rs1, rs2 ws:=rs1 >> rs2

Branch:

OP (4 bits)	Rs1 (3 bits)	Rs2 (3 bits)	Offset (6 bits)

Sample instructions :-

1. Branch on Equal:
   a. BEQ rs1, rs2, offset
   b. Branch to (PC + 2 + (offset << 1)) when rs1 = rs2
2. Branch on Not Equal:
   a. BNE rs1, rs2, offset
   b. Branch to (PC + 2 + (offset << 1)) when rs1 != rs2

Jump:

OP (4 bits)	Offset (12 bits)

Sample instructions :-
Jump: JMP offset Jump to {PC [15:13], (offset << 1)}

ALU and Control Unit Implementation :-

Alu Control Unit enables Zeus to decide which Arithmetic function to operate based on the 16 bit command given. The file “alu_control.v” receives the signal from Control unit and based upon the ALU_OP it receives and transfers Alu_Cnt to ALU of the processor.

Control Unit supervises the functioning of the processor sending in control signal to various components of the processor setting their state as active(in use during the execution of the command). The first 4 bits of the command sets the opcode of our instruction and the file “Control_Unit.v” and sets signal values.

Instructions :-

INSTRUCTION	OPCODE	ALUOP
LOAD	0000	10
STORE	0001	10
ADD	0010	00
SUB	0011	00
MUL	0100	00
SLL	0101	00
SRL	0110	00
AND	0111	00
OR	1000	00
SLT	1001	00
XOR	1010	00
INVERT	1011	00
BEQ	1100	01
BNE	1101	01
JUMP	1110	11

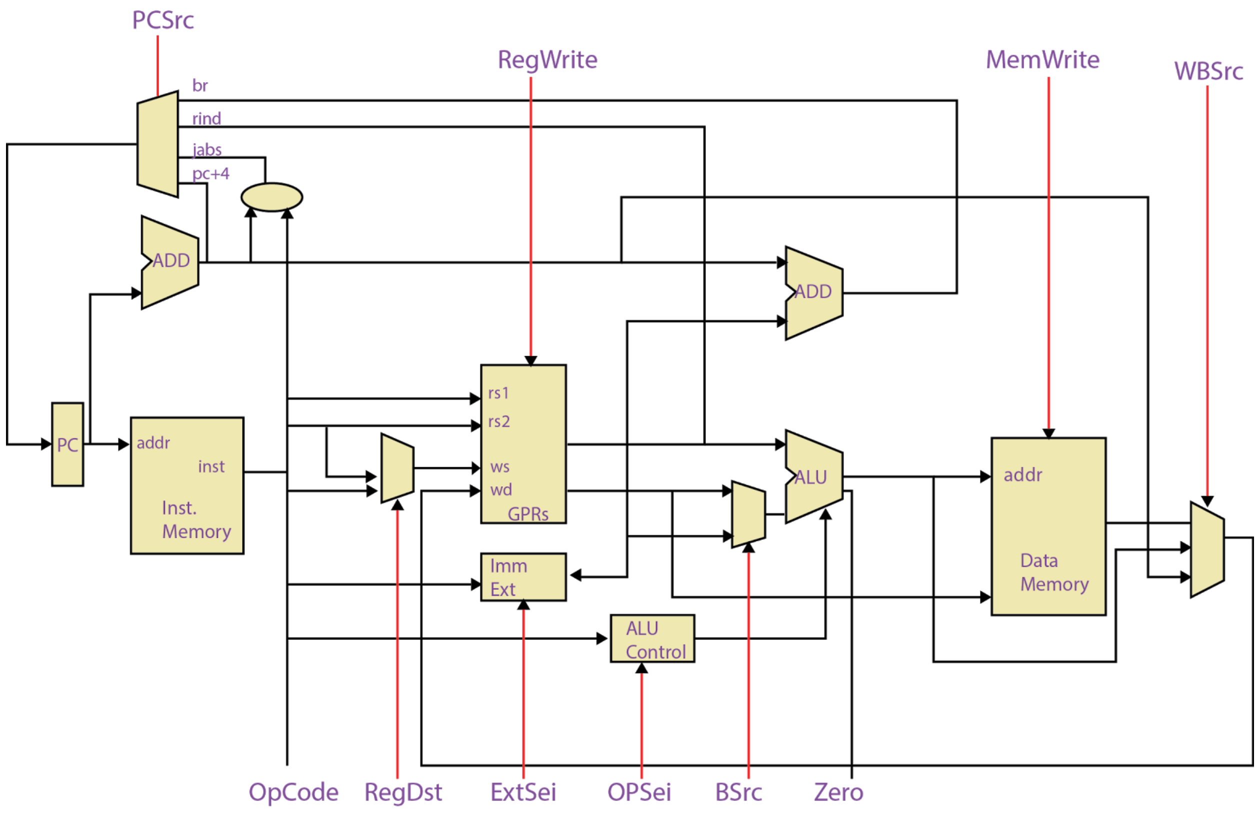
Control Unit Design

Instruction	Reg Dst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALU OP	Jump
Data-Processing	1	0	0	1	0	0	0	00	0
LW	0	1	1	1	1	0	0	10	0
SW	0	1	0	0	0	1	0	10	0
BEQ	0	0	0	0	0	0	1	01	0
BNE	0	0	0	0	0	0	1	01	0
J	0	0	0	0	0	0	0	00	1

1. Reg Dst - Specifies the register destination to be written
2. ALUSrc - Specifies the ALU control signal to carry out the function
3. MemToReg - Set's data memory to enable memory to be loaded in register
4. RegWrit - Enables register write
5. MemRead - Enables register read
6. MemWrite - Sets data memory to be written
7. Branch - Set when a branch is encountered with association with PC
8. ALUOP - Enables ALU to choose from different type of instructions
9. Jump - Set when a jump is encountered

Insight To Datapath

Datapath serves as the roadway linking all the components of the processor, sending signals to concerned components during the execution of the command. With the formation of datapath individual components could now interact with one another and Zeus was no longer just a bunch of files but a complete entity, capable of fully functioning as required.
Until now each component was individually tested and the results were as expected. With the formation of datapath Zeus faced the test of taking input from a file,storing them in register, reading the commands from “test.v”, sending the required portions of the code to the respective files, generating the to be expected control signals, forwarding the ALU_CONTROL to the ALU and calculating the final result.
And thus Zeus worked for the first time and gave the results it was expected to.



-- Datapath --

Datapath Design :-

The commands are taken as an input from the “test.data” files and are sequentially send via the **program counter**. The command reaches the **register block(GPRs.v)** and the argument values are stored in the register. The datapath has input of **al_op[1:0]** a two bit code describing the type of instruction to execute.The file “**Datapath_Unit.v”** also gives output as **OpCode[3:0]**, a four bit instruction which goes as input to **“Control_Unit.v”** where the status signals of the different components are set thus defining their functionality during the execution of the command.

1. Load/Store, D-type instructions(Arithmetic) and Branch Instruction :
Control_Unit.v generates the **Alu Control** signal which when executed in the file “**alu_control.v”** sends the signal to “**ALU.v”** and calculates the result which are displayed as waveforms.

2. Jump:
When a Jump is encountered, the datapath increments the program counter on the basis of the **offset** provided and the next instruction is loaded after the current command(JUMP) executes.

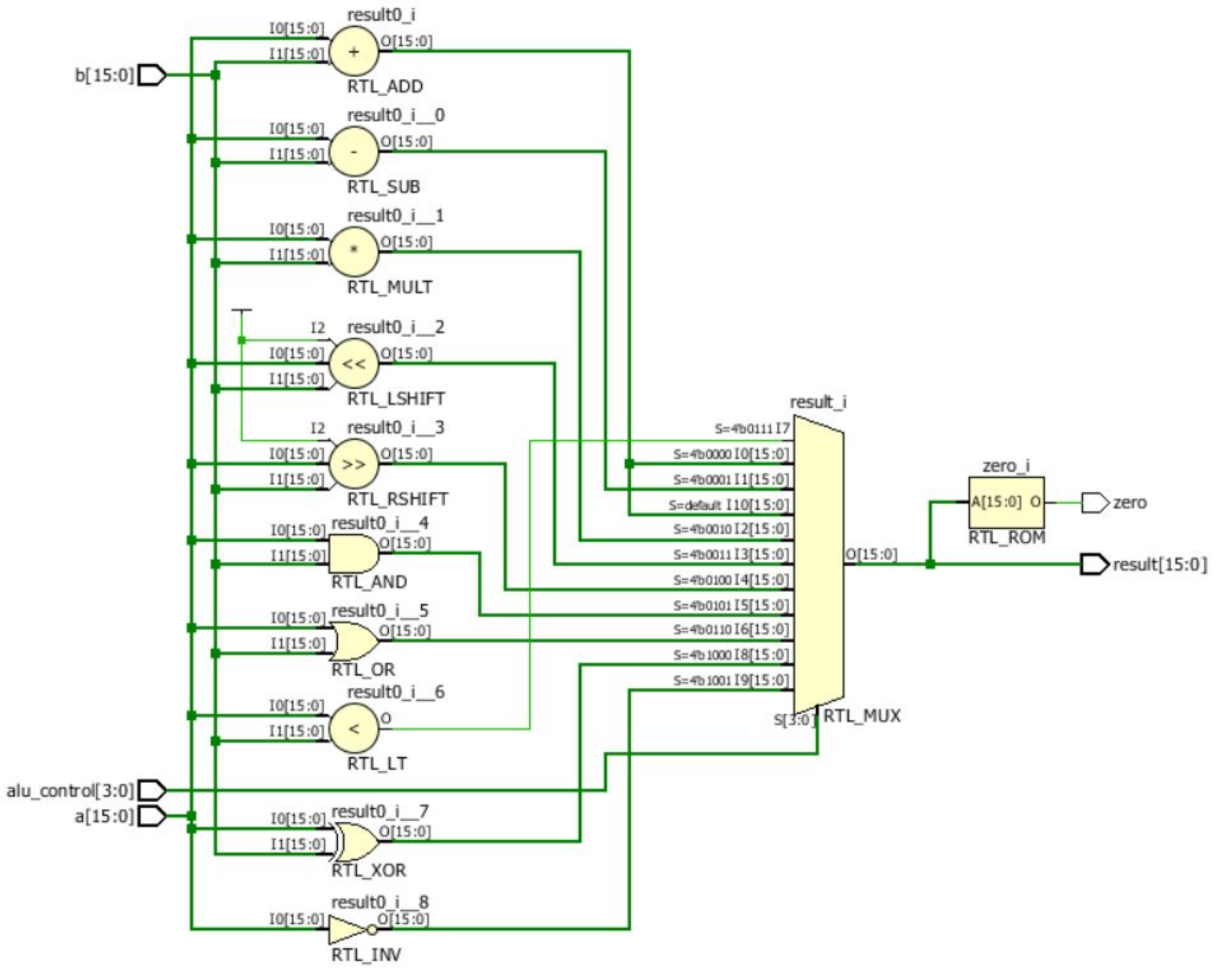
Implementing Zeus In Verilog

The verilog implementation of Zeus began with the coding up of individual components and then linking them via the datapath.

The individual components include :-

- Alu.v
- Control_Unit.v
- Data_Memory.v
- GPRs.v
- Instruction_Memory.v
- Parameter.v
- Risc_16_bit.v
- alu_control.v

Alu.v :



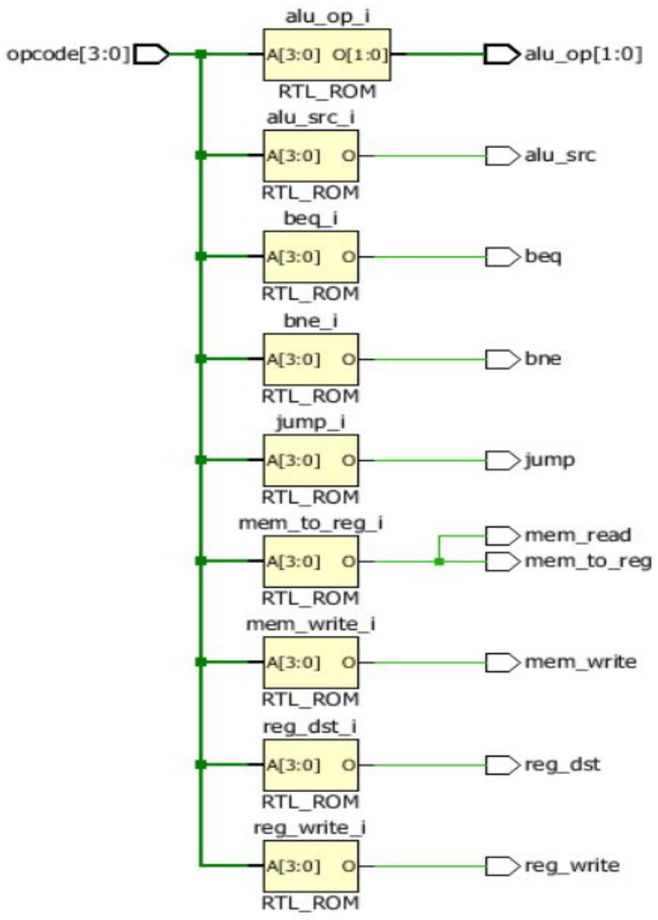
-- Schematic of ALU unit --

ALU is mainly needed in execution stage of the processor. It is used for computation of all necessary processes. It operates on data taken from the previous stage i.e. decode the ALU_Cnt
In this file, we have used switch case to define all different types of operations.
For example:

```
4'b0000:result =A+B;
```

This result is then send to register file for further computation.

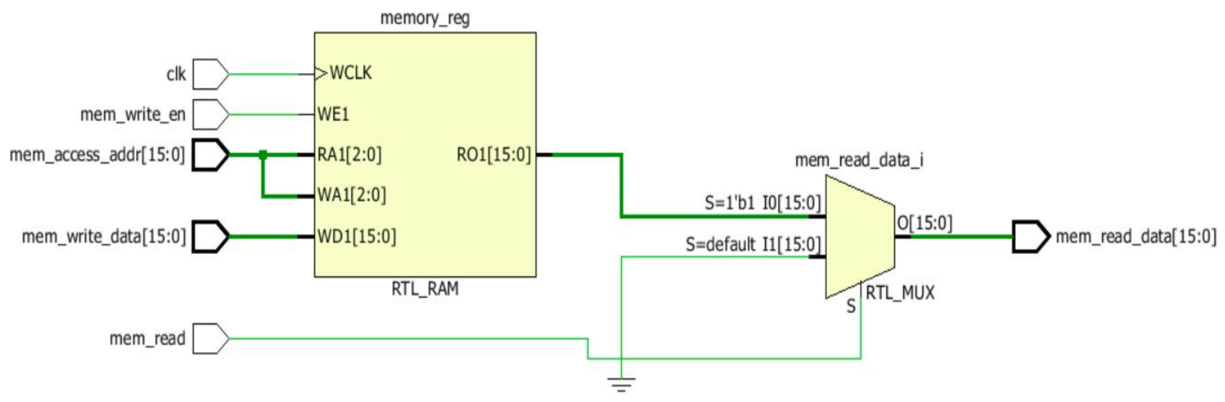
Control_Unit.v :



-- Schematic of control unit --

The **control unit** (CU) is a component of CPU. It is used to direct the operations of the processor. It let's the computer know, arithmetic **unit** and input and output devices on how to respond to a instructions.
In this file, we are defining all the signals the are sent to multiplexers from control unit which helps it decide which signal he want to send further. Different sets of codes are decided for every opcode.

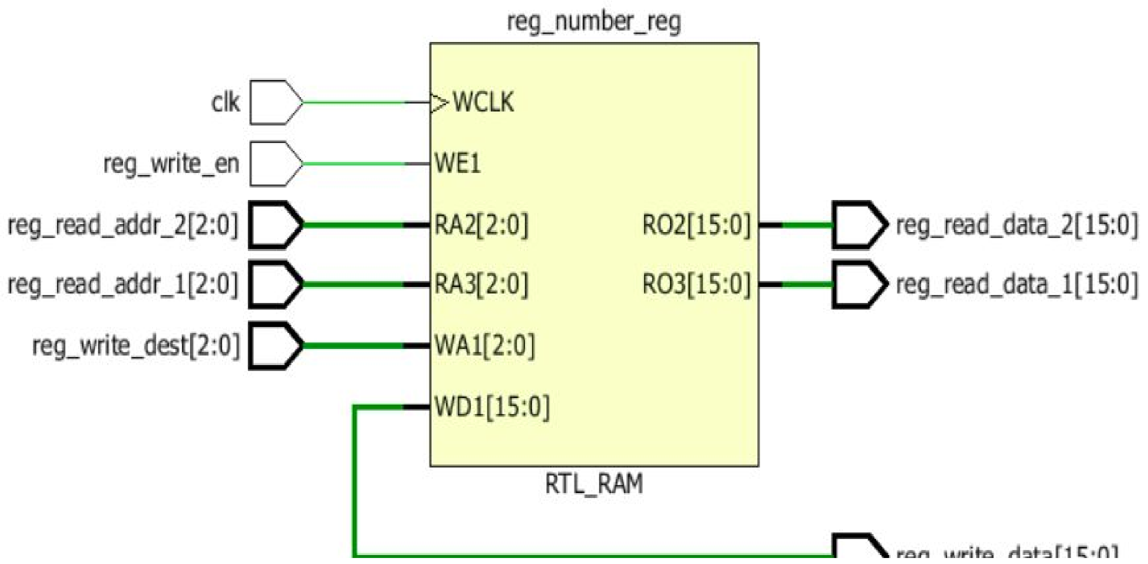
Data_Memory.v :



-- Schematic of data memory --

Data memory is used for storing and keeping all data required for the proper operation of the programs. It loads the initial data from test.prog file. It is mainly used in load and store operations. Memory write wire is used to inform the data memory, which of the above operation to perform. Memory address and memory data is provided which helps it store data to the memory register.

GPRs.v :

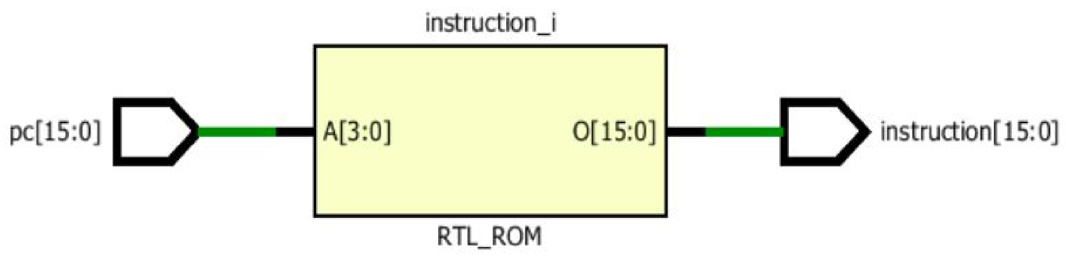


-- Schematic of General Purpose Register --

A register is a discrete memory location within the central processing unit of the device that is designed to hold instructions and temporary data. A modern CPU normally consists of many internal registers. Major part of it is general purpose registers that the programmer can use to hold intermediate results whilst working through a calculation or algorithm. It stores address, data and and write destination and then are further send to alu for computation.

Instruction_Memory.v :

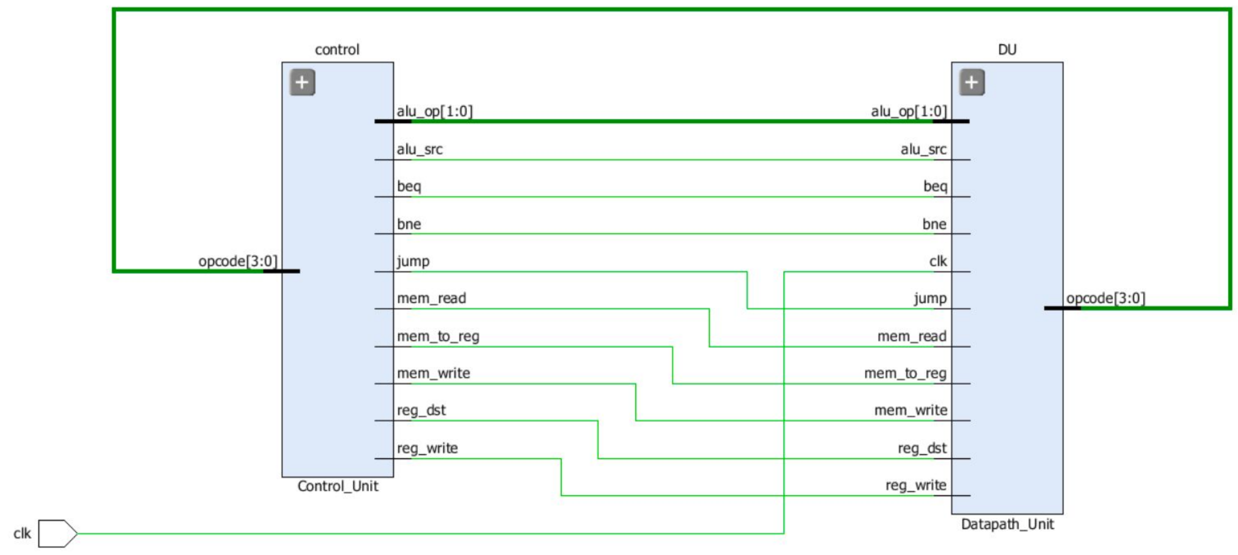
-



- Schematic of instruction memory --

An **instruction memory** holds the instruction currently being executed or decoded. In simple processors each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and ultimately executed, which can take several steps. Instruction address is loaded from the program counter to instruction memory, it decodes it and extracts the data and send to general purpose register for further functioning of the processor.

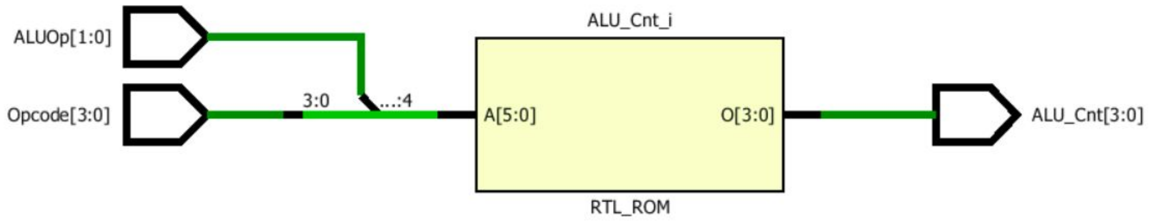
Risc_16_bit.v :



-- 16-bit RISC --

This module is used to control all other modules of the processor. It sends signals to datapath and control unit that controls the functioning of the processor.

Alu_control.v :



-- Schematic of ALU Control --

ALU control unit is an integral part of the processor. Its primary function is to control the ALU generating a 4 bit ALU_Cnt corresponding to the incoming ALUOp provided by the Control Unit and opcode supplied by the datapath. Using ALU control unit, we are able to reduce the redundancy of instructions. For example, LOAD instruction is a simple ADD instruction for the ALU and to implement it, we haven't used different operation in ALU. Instead we generate the same ALU_Cnt for the ADD and LOAD type instruction, thus reducing the redundancy of code. In the future if one wants to enable some complex instructions, ALU Control is the part where one has to edit it mainly keeping the ALU intact. ALU will perform the basic instructions according to the signal generated by the ALU control Unit.

ALU control is a very basic unit and is completely implemented using a switch case statement. We basically merge the ALUOp and opcode together and generate a signal ALU_Cnt on the basis of that.
Example:
If a user wants to load a variable, to the ALU its a simple ADD command, therefore, combining the opcode which is 0000 for the LOAD and ALUOp which control unit returns as 10 for LOAD. It checks the entries corresponding to 100000 and finds 0000 which is basically the ALU_Cnt for ADD instruction.

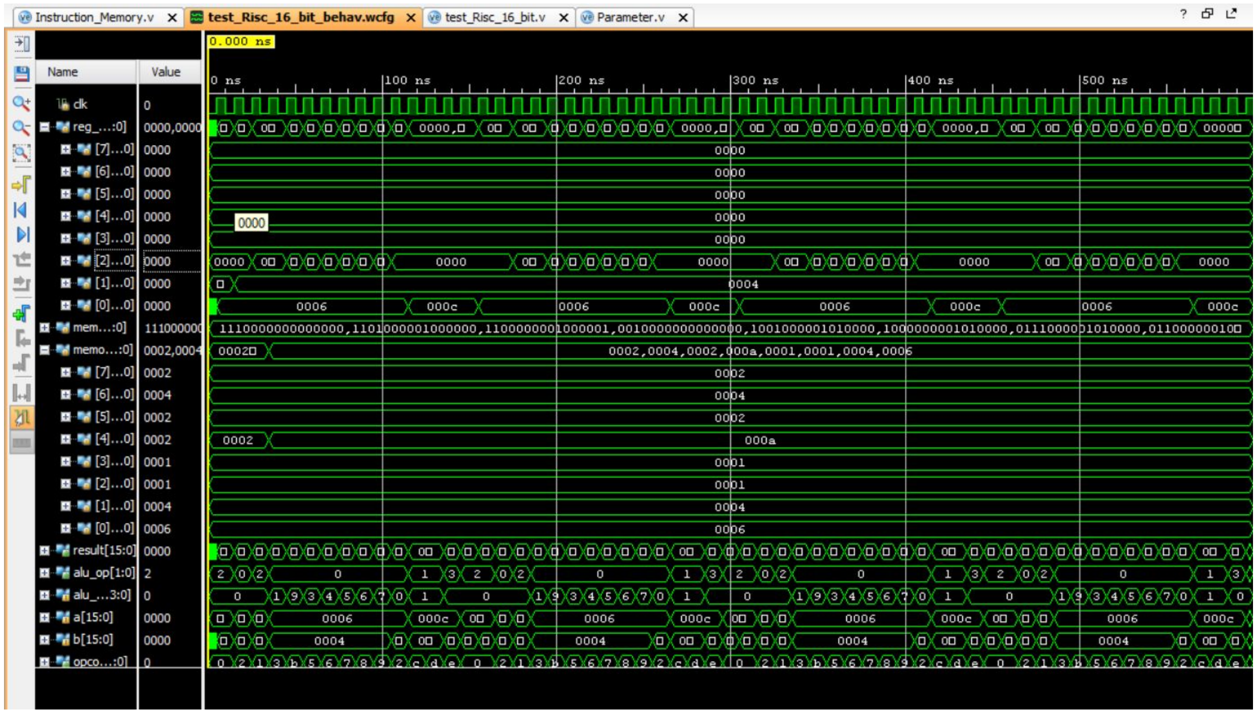
Some Testbench Waveforms

Zeus has the capability of handling several types of instructions. The following example illustrates some of them.

Code:

```
0000_0100_0000_0000 // load R0 <- Mem(R2 + 0)
0000_0100_0100_0001 // load R1 <- Mem(R2 + 1)
0010_0000_0101_0000 // Add R2 <- R0 + R1
0001_0010_1000_0000 // Store Mem(R1 + 0) <- R2
0011_0000_0101_0000 // sub R2 <- R0 - R1
1011_0000_0101_0000 // invert R2 <- !R0
0101_0000_0101_0000 // logical shift left R2 <- R0<<R1
0110_0000_0101_0000 // logical shift right R2 <- R0>>R1
0111_0000_0101_0000 // AND R2<- R0 AND R1
1000_0000_0101_0000 // OR R2<- R0 OR R1
1001_0000_0101_0000 // SLT R2 <- 1 if R0 < R1
0010_0000_0000_0000 // Add R0 <- R0 + R0
1100_0000_0100_0001 // BEQ branch to jump if R0=R1, PCnew= PC+2+offset<<1 = 28 => offset = 1
1101_0000_0100_0000 // BNE branch to jump if R0!=R1, PCnew= PC+2+offset<<1 = 28 => offset = 0
1110_0000_0000_0000 // J jump to the beginning address
```

Corresponding Output:

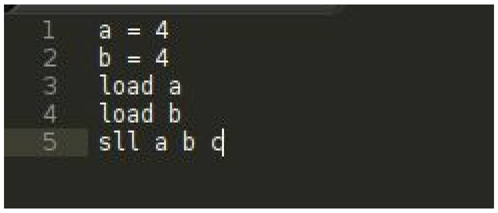


-- Waveform of test case 1 --

Assembler Code

This is the special feature of Zeus. It has an assembler of its own that makes it much more user friendly. The example is about running an 'sll instruction'

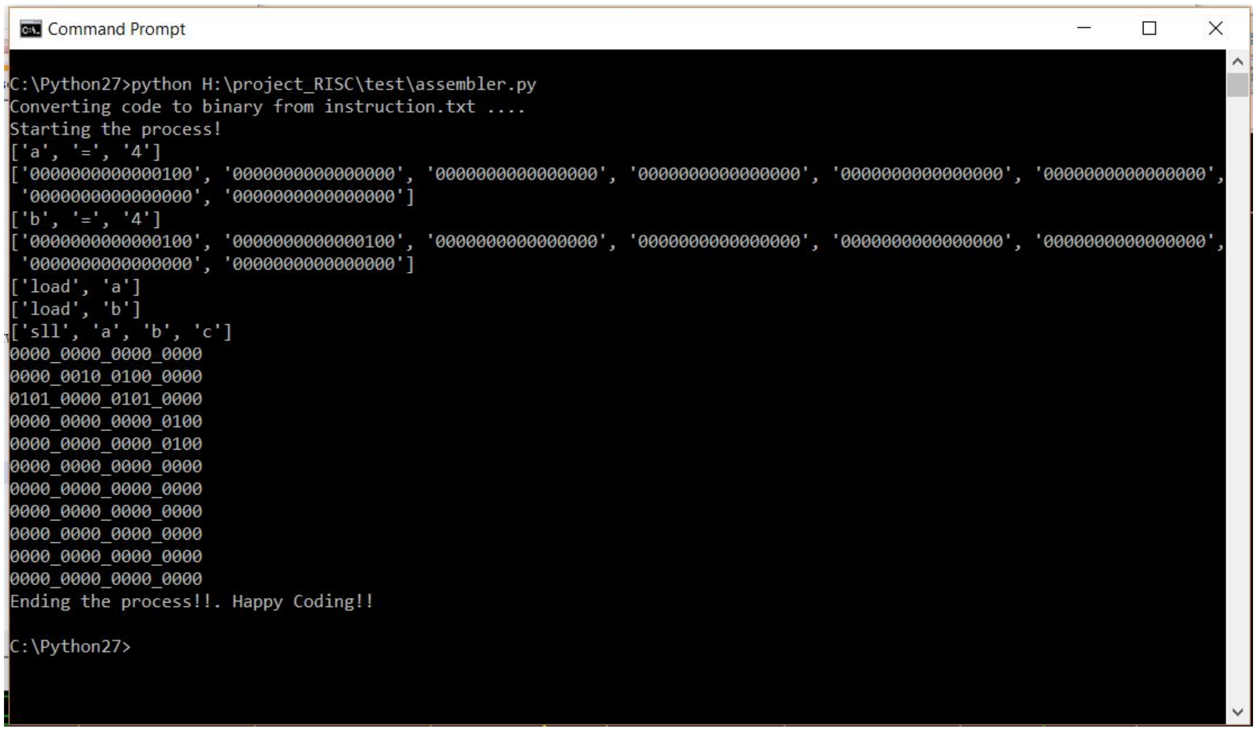
Instruction typed in Zeus' assembler code.



```
a = 4
b = 4
load a
load b
sll a b c
```

-- Zeus assembler code --

Running the python script via the command prompt.

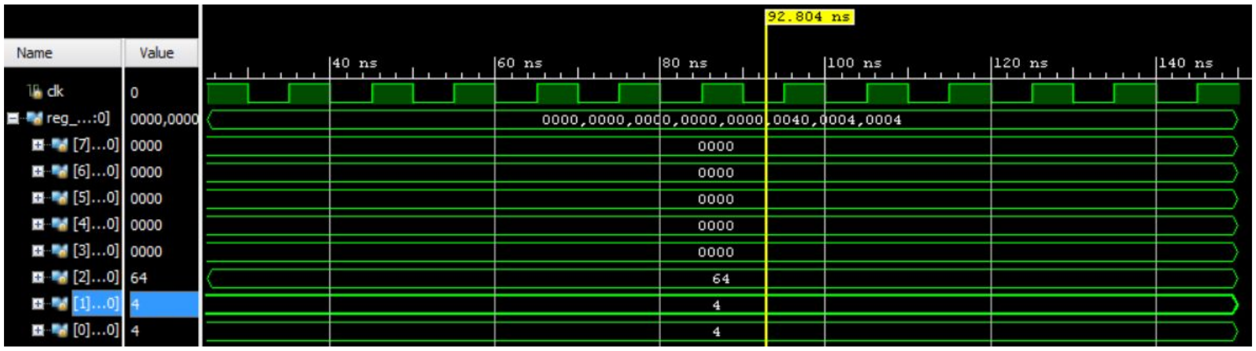


```
C:\Python27>python H:\project_RISC\test\assembler.py
Converting code to binary from instruction.txt ....
Starting the process!
['a', '=', '4']
['0000000000000100', '0000000000000000', '0000000000000000', '0000000000000000', '0000000000000000', '0000000000000000',
 '0000000000000000', '0000000000000000']
['b', '=', '4']
['0000000000000100', '0000000000000100', '0000000000000000', '0000000000000000', '0000000000000000', '0000000000000000',
 '0000000000000000', '0000000000000000']
['load', 'a']
['load', 'b']
['sll', 'a', 'b', 'c']
0000_0000_0000_0000
0000_0010_0100_0000
0101_0000_0101_0000
0000_0000_0000_0100
0000_0000_0000_0100
0000_0000_0000_0000
0000_0000_0000_0000
0000_0000_0000_0000
0000_0000_0000_0000
0000_0000_0000_0000
0000_0000_0000_0000
Ending the process!!. Happy Coding!!

C:\Python27>
```

-- code assembled in python --

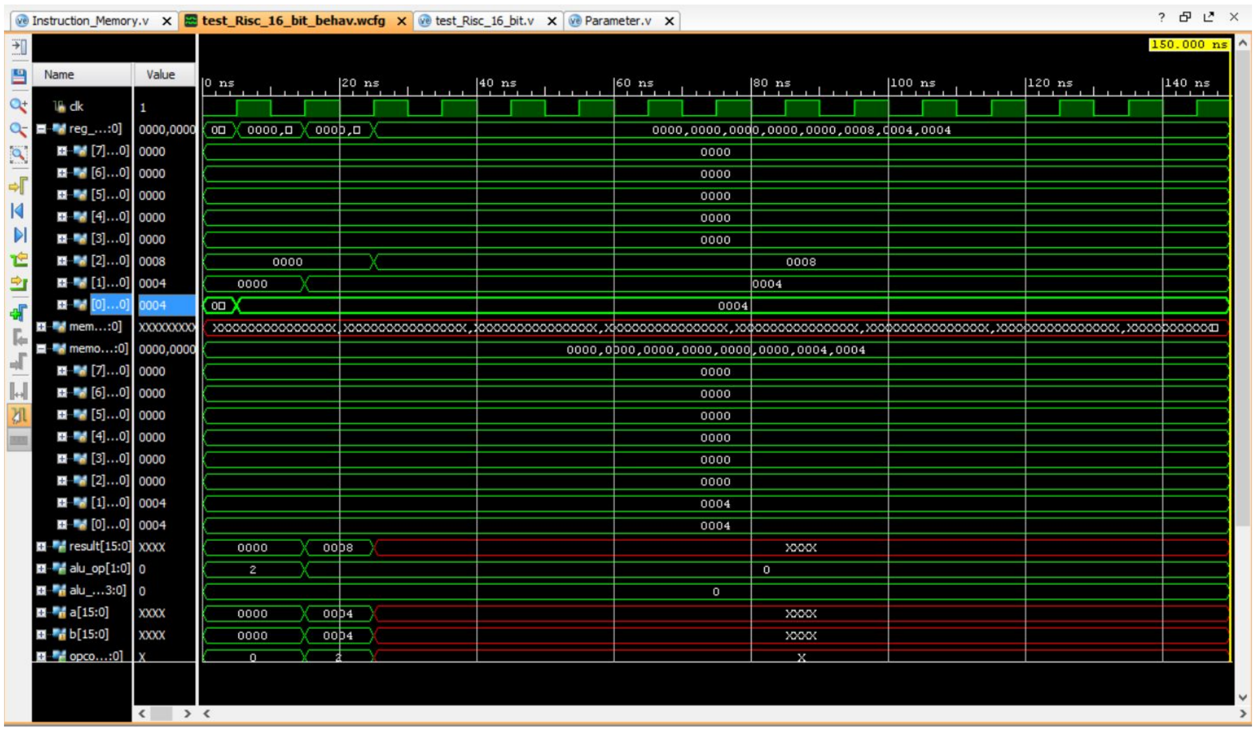
Simulating the generated code in verilog.



-- sll output waveform --

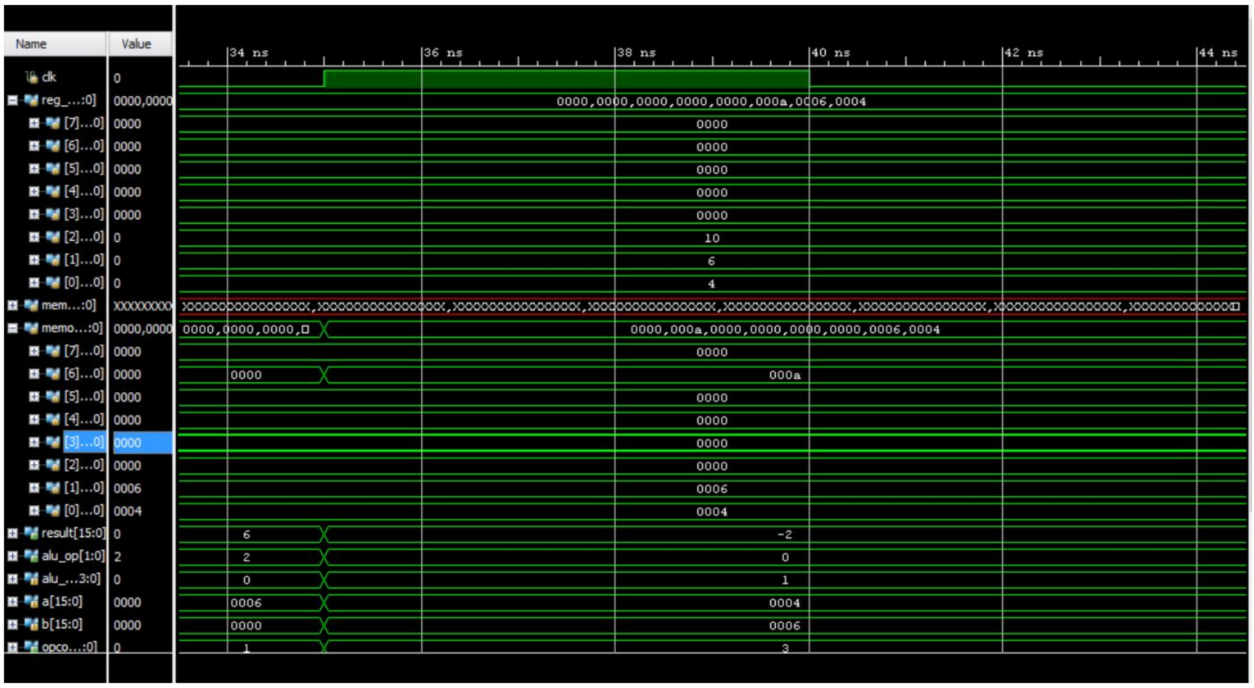
The above instruction, thus does 4*2^4=64.

Further an example of simple add instruction:
4+4=?
Yes, Zeus does that for us.



-- Add 4, 4 result waveforms --

The next major milestone that Zeus had to overcome was handling the negative results, so what must be the output corresponding to the instruction **subtract 4** and **6**?
Let's put Zeus to test:



```
Name            Value       |34 ns      |36 ns      |38 ns      |40 ns      |42 ns      |44 ns
clk             0
reg_...:0]      0,0000,0000   0000,0000,0000,0000,0000,000a,0006,0004
  [7]...0]      0000          0000
  [6]...0]      0000          0000
  [5]...0]      0000          0000
  [4]...0]      0000          0000
  [3]...0]      0000          0000
  [2]...0]      0             10
  [1]...0]      0             6
  [0]...0]      0             4
mem...:0]       XXXXXXXX      XXXXXXXXXXXXXXXX,XXXXXXXXXXXXXXXX,XXXXXXXXXXXXXXXX,...
memo...:0]      0000,0000     0000,0000,0000,0  X  0000,000a,0000,0000,0000,0000,0006,0004
  [7]...0]      0000          0000
  [6]...0]      0000          0000  X  000a
  [5]...0]      0000          0000
  [4]...0]      0000          0000
  [3]...0]      0000          0000
  [2]...0]      0000          0000
  [1]...0]      0006          0006
  [0]...0]      0004          0004
result[15:0]    0             6     X  -2
alu_op[1:0]     2             2     X  0
alu_...3:0]     0             0     X  1
a[15:0]         0000          0006  X  0004
b[15:0]         0000          0000  X  0006
opco...:0]      0             1     X  3
```

-- Testing subtraction of a larger number from smaller --

Zeus was able to handle the above instruction and gave -2 as the result, thus working perfectly and overcoming one of the major milestone, handling signed bits.

What we learned

Creating Zeus was a golden opportunity, an experience that enhanced our overall understanding of how the CPU works, the role various components play in the execution of a command. Working on Zeus gave us a clear insight into the internal functioning of a processor.

While designing Zeus we faced issues, found out the benefits and drawbacks of various ISA's that exist, but mostly we came to appreciate the amount of hard work that was inputted in the development of the first processor. So within the scopes of our current knowledge and capabilities we sought to make Zeus to carry on the legacy of its predecessors.

The biggest change we found was that now we have implemented an ISA and datapath ourselves, understanding the concepts in books were now relatively simpler.The flow of bits were now somehow visible and and the logic behind the particular design was more understandable.

As the ISA was entirely our own brain child, the issues took a major portion of our time and massive hard work to debug. In doing so we developed a greater insight into the designed datapath and especially how it is implemented in verilog. The data transfer between various component files.

Designing and developing Zeus was a golden opportunity to learn, grow and implement our knowledge and we sincerely thanks Dr.Vaskar Raychoudhury to provide us with this opportunity. We hope you enjoy reading about Zeus as much as we enjoy creating it.

Conclusion

The report on Zeus x16 highlights the features of a microprocessor and the fascinating ideas incorporated in it. Zeus x16 provided was a platform for us to learn about the basic idea of what microprocessor really is, how it really works and how simplicity is the boss.

When we started working on the processor, we weren't even proficient with verilog but with series of success and failure we were able to make our own microprocessor, one of its kind, at least for us. It was really a lively and a fascinating time building Zeus x16.

Zeus x16 taught us not just the basics of computer architecture but also enhanced our life skills like punctuality and teamwork (even this report is a collective effort of all of us working at a time). It also taught us to to rise stronger every time we fall.

So, we hope our effort provides you with a basic idea of how to build a microprocessor. Hope you like our little effort in this direction.

Future Prospects

This project was a great learning experience for us but we still need to improve on many aspects of Zeus x16. We could name this project as “Zeus x16 1st Generation”. There are many more generations to come. We look forward to incorporate the following features in Zeus:

1. Right now, Zeus has 3 useless bits for data processing type of instructions. We are looking forward to use them.
2. Zeus has an ALU implemented using internal verilog functions of add and other things. We look forward to implementing them using Logic Gates so that Zeus can be faster than before.
3. We look forward to implement a RISC pipeline system for Zeus so that it can handle instructions parallely.
4. We would like to make Zeus multi core having at least four cores so that it can process instructions way faster than now.
5. We strongly wish to upgrade our assembler so that it can unroll loops, handle branching and other things.
6. We would like to implement a disassembler so that you can see the output from the processor on your command prompt.
7. Finally, we look forward to writing a script which can do all of the following processes - assembling, simulation and disassembly so that you give the input and it gives you the output on screen.

The future looks exciting for Zeus as it has many features to implement and we have so much to do. The current processor is just an indication of what is about to come. We have worked hard on this project and would surely carry this zeal afterwards to implement all the above proposed features and mighty Zeus bless us, would be successful in doing so.

Bibliography

1. **Computer Organisation and Design** by David Patterson and John Hennessey
2. **Computer Organization and Architecture** by William Stallings
3. **Computer System Architecture** by M.Mano
4. **Digital Logic and Computer Design** by M.Mano
5. http://vol.verilog.com/ To Understand the advanced functioning of Verilog
6. **Class Lectures and Discussions(Major inspiration) :-**
    - **The Use of Large register File, Register Windows, Global Variables**
    - **Assembly Language Programming with MIPS ISA, Basics of Datapath Design**
    - **CPU(Datapath) Design 2**
    - **CPU(Datapath) Design 3**