RISC-V based MYTH Workshop

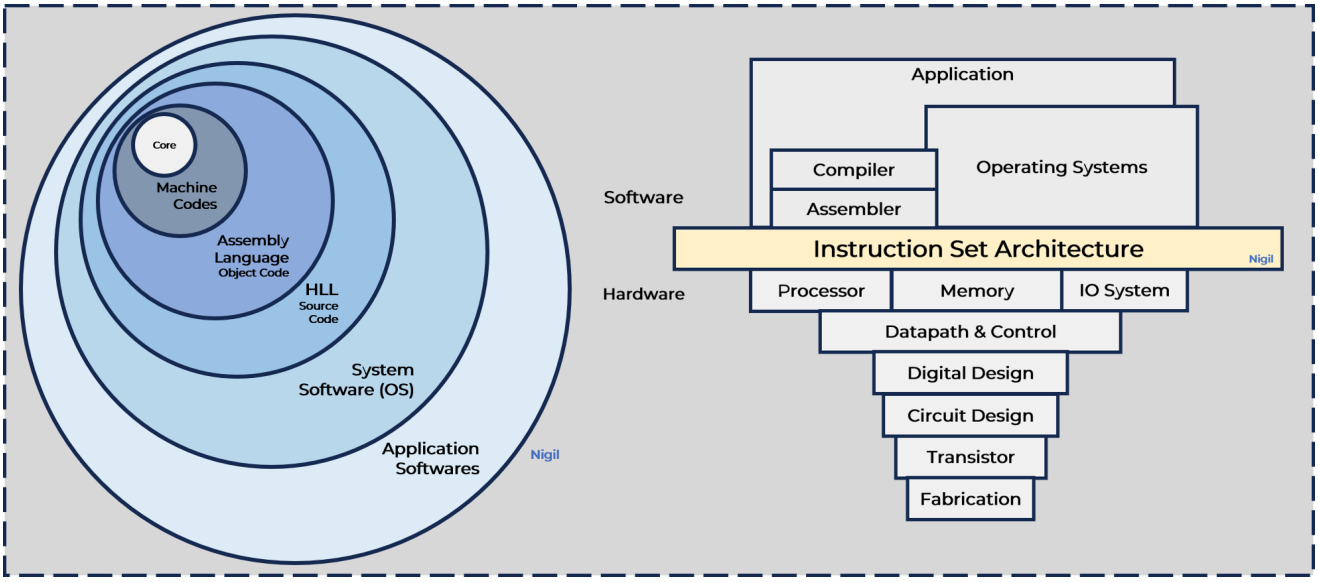
NIGIL MOHRA

RISC-V Design using Transaction-Level Verilog

This repository contains the documents, codes, and materials related to the RISC-V-based MYTH workshop, organized by NASSCOM India in collaboration with Steve Hoover and Kunal Ghosh.

The reference solutions for the lab and practices can be found in this Solution. The solutions do not include code; they include only the design visualizations.

Introduction to RISC-V Instruction Set using GNU Compiler Tool Chain and Spike Simulator



Softening the Hardware (Onion Layers) and Digital Machine Structure

The idea is to implement different `C` programs, convert them to object code (which is the assembly language), and run them using the RISC-V compiler built into the GNU Compiler Collection (GCC). Then, debugging is done using the Spike Simulator to develop a basic understanding of the Instruction Set Architecture (ISA) and how high-level language code is broken down into multiple instructions.

Apart from standard C/C++ and Fortran language support, the **GCC Compiler** supports multiple architectures and platforms, including RISC-V, ARM, x86, and many others, enabling developers to write code on one platform and later compile it to run on others.

The **Spike Simulator** is an **Instruction Set Simulator** specifically design for RISC-V, an open source instruction set architecture. It is an official simulator for RISC-V, used to model and simulate the processors.

Create a small `C` program that performs the addition of numbers from 1 to `N`. Run the program using the commands `gcc <FILE_NAME.c>` and `./a.out`.

Generate RISC-V Object File

Either one of the above two codes can be used to generate the object file.

```
riscv64-unknown-elf-gcc -O1 -mabi=lp64 -march=rv64i -o <OBJECT_FILE_NAME.o> <C_PRG_FILE_NAME.c>
```

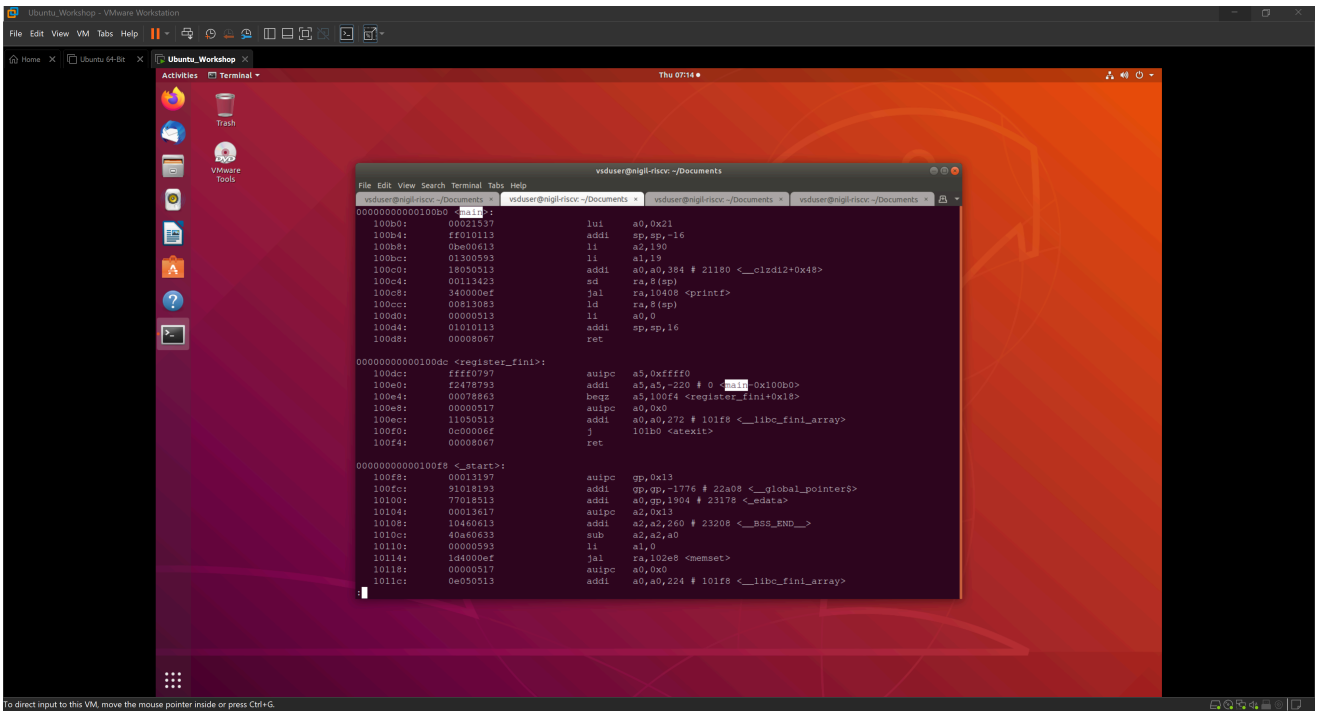
```
riscv64-unknown-elf-gcc -Ofast -mabi=lp64 -march=rv64i -o <OBJECT_FILE_NAME.o>
<C_PRG_FILE_NAME.c>
```

Assembly Code

The first command provides an extended version of the assembly code. The second command will provide a `piped` version of the code.

```
riscv64-unknown-elf-objdump -d sum.o
```

```
riscv64-unknown-elf-objdump -d sum.o | less
```



Assembly Language Codes - Sum of 'N' Numbers (-Ofast)

Debugging - Spike Simulator

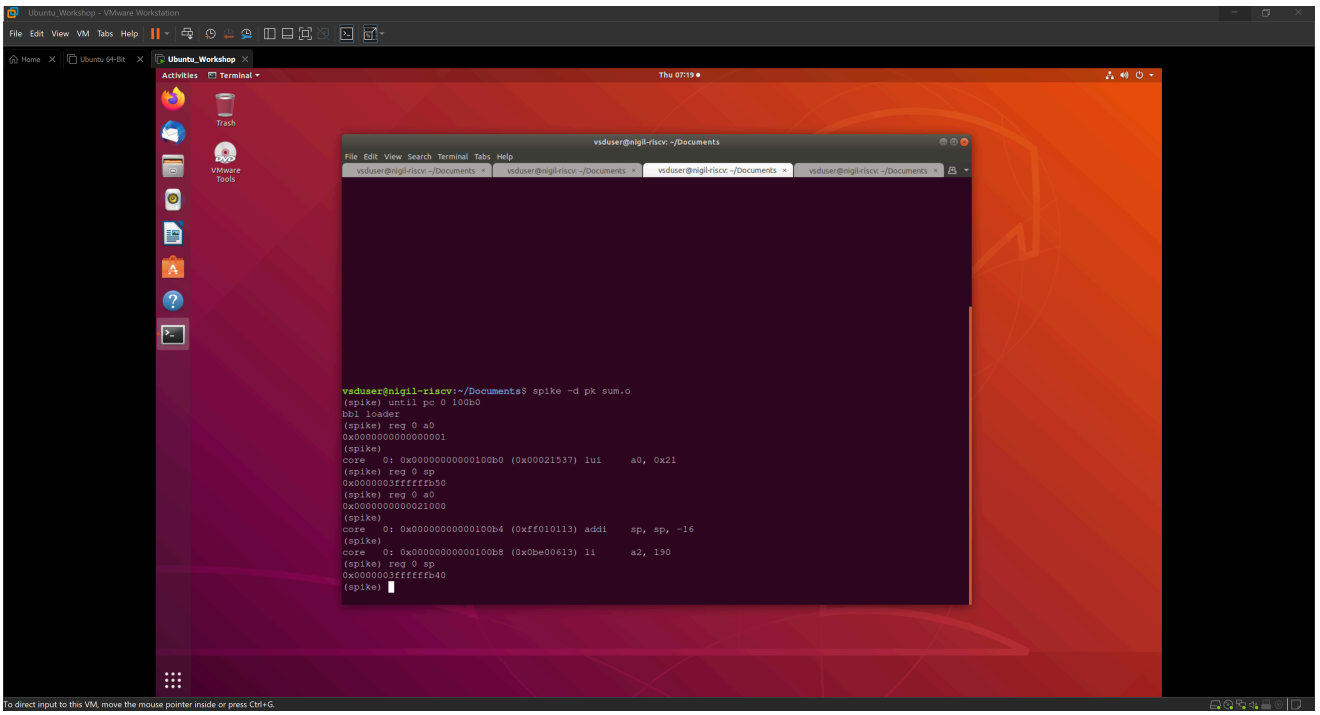
The Spike simulator is invoked using the spike disassemble command. By using the `until` command, the program can be executed starting from a particular address. Pressing `ENTER` executes the consecutive steps. The updates to the registers can be viewed using the `reg` command.

In the assembly code, by using the appropriate syntax, parts of the code can be easily navigated. For example, `/main` finds all instances of 'main,' and pressing 'n' repeatedly will help you find the correct instance.

```
spike -d pk <OBJECT_FILE_NAME.o>
```

```
(spike) until pc 0 <ADDRESS_TILL_EXECUTION>
```

```
(spike) reg <CORE> <REGISTER_NAME>
```



Debugging using Spike Simulator

The explanation of the shell commands can be found on the internet. Lectures on signed and unsigned integers were viewed, but no notes were made.

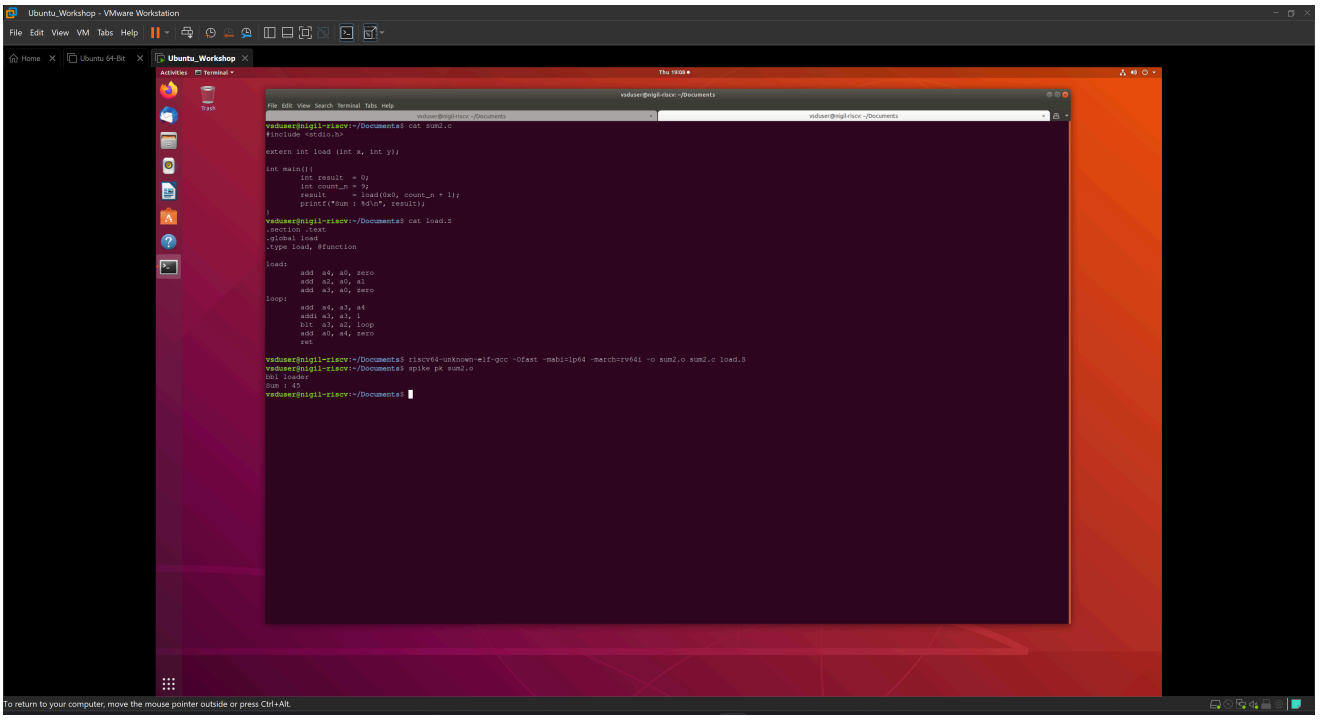
Introduction to Application Binary Interface (ABI)

An Application Binary Interface (ABI) is a set of rules and conventions that define how different components of a program interact at the binary level. It acts as an intermediary between various program modules or between the program and the operating system, ensuring seamless interoperability among software components, even when they are written in different programming languages or compiled with different compilers. Below is ABI symbolic register names for RV64I.

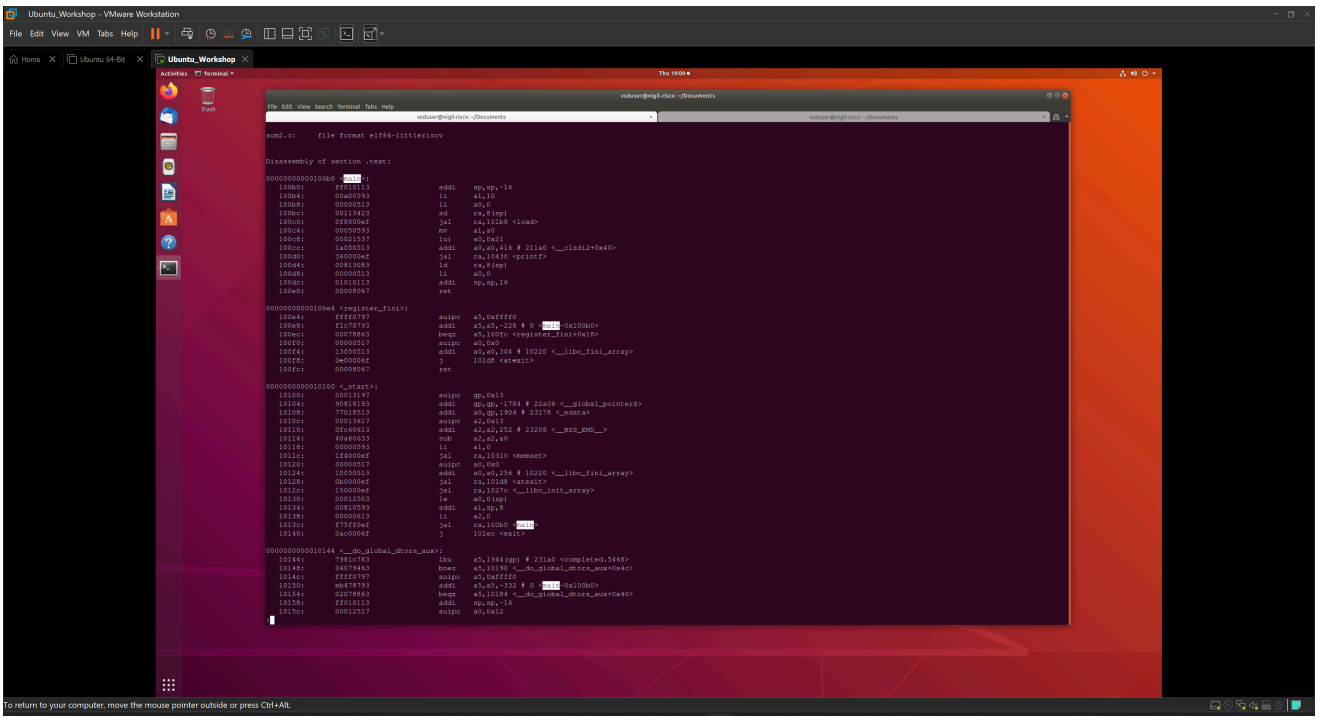
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

ABI RISC-V Symbolic Register Names

Lab : Call to Assembly Language from a High-Level Language Program



Assembly Language and High-Level (C Program) Codes



Main Program Showing Execution of `load.S' Assembly Call

Digital Logic Design with Transaction Level Verilog

For the initial simulation and design, the Makerchip IDE is used. Detailed explanations and guidance on using the IDE can be found here in this Link

Note that the Makerchip IDE platform does not recognize `TAB` for indentation; instead, use three spaces for indentation. `CTRL + ]` is used to indent the code to the right, and `CTRL + [` is used to indent the code to the left. Also, note that in TL-Verilog, using `*reset` refers to SystemVerilog code defined in macros.

Combinational Logic

As a standard approach for learning any hardware description language, the process starts with the implementation of basic logic gates. The logical operators are similar to those in Verilog, with the primary difference being that there is no need for explicit declaration of the inputs and outputs.

This is a basic example of a simple **2-to-1 8-Bit Multiplexer**.

```
\TLV

$out[7:0] = $sel ? $in1[7:0] : $in2[7:0]
```

Lab : Combinational Calculator

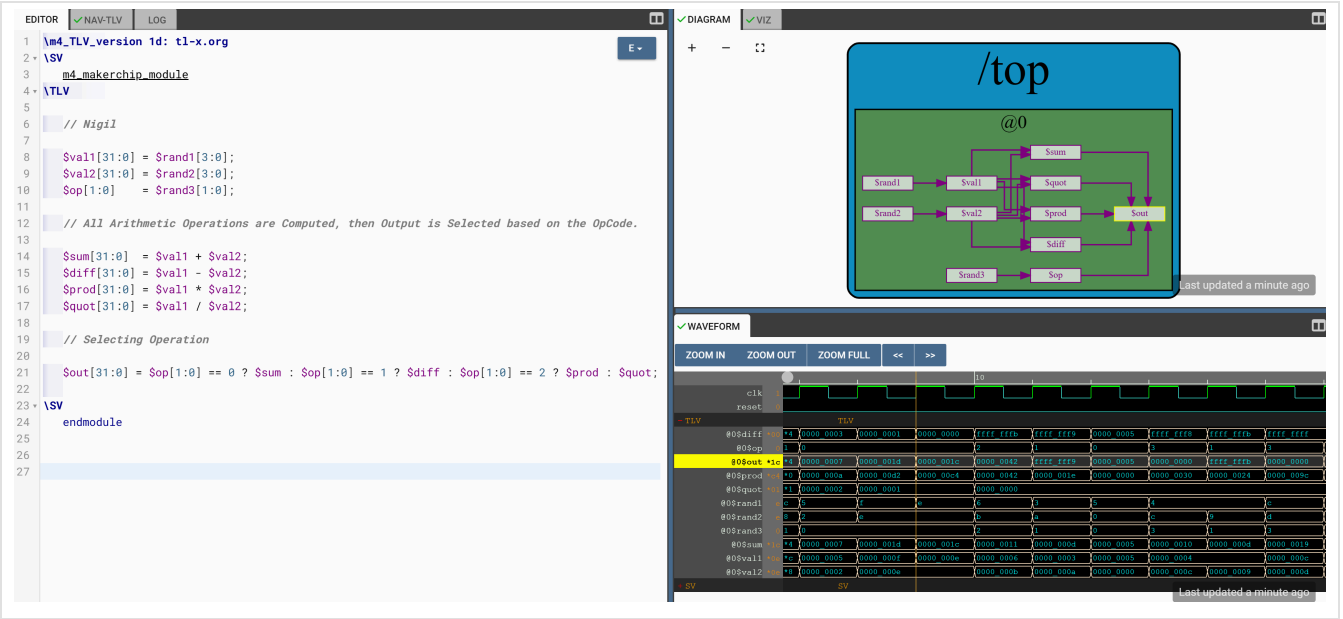


Figure 1. Combinational Calculator - Makerchip IDE Output

Sequential Logic

Once the combinational circuits are completed, the next step is to move on to the sequential circuits. A basic Fibonacci series is provided as an example for the sequential circuits. The circuit is constructed such that it enters the known state when a `RESET` signal is present. For the Fibonacci code, the known state is `1`. The syntax `>>1` provides the previous value of `$val`, and `>>2` provides the value of `$val` two states prior. Similarly, `>>x` will provide the value of `$val` `x` cycles prior.

```
\TLV

   // Fibonacci Series Example

   $num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

This is a simple **8-Bit Free Running Counter**.

```
\TLV

   // 8-Bit Free Running Counter

   |calc
      @0
         $reset = *reset;

         $cnt[7:0] = $reset ? 0 : (>>1$cnt + 1);

   // Limiting Simulation

   *passed = *cyc_cnt > 20;
   *failed = 1'b0;
```

The Makerchip IDE uses the open-source Verilator for simulation. It supports only two-state simulation and does not support don't care or high impedance states. The simulator will zero-extend or truncate when widths are mismatched.

Lab : Sequential Calculator

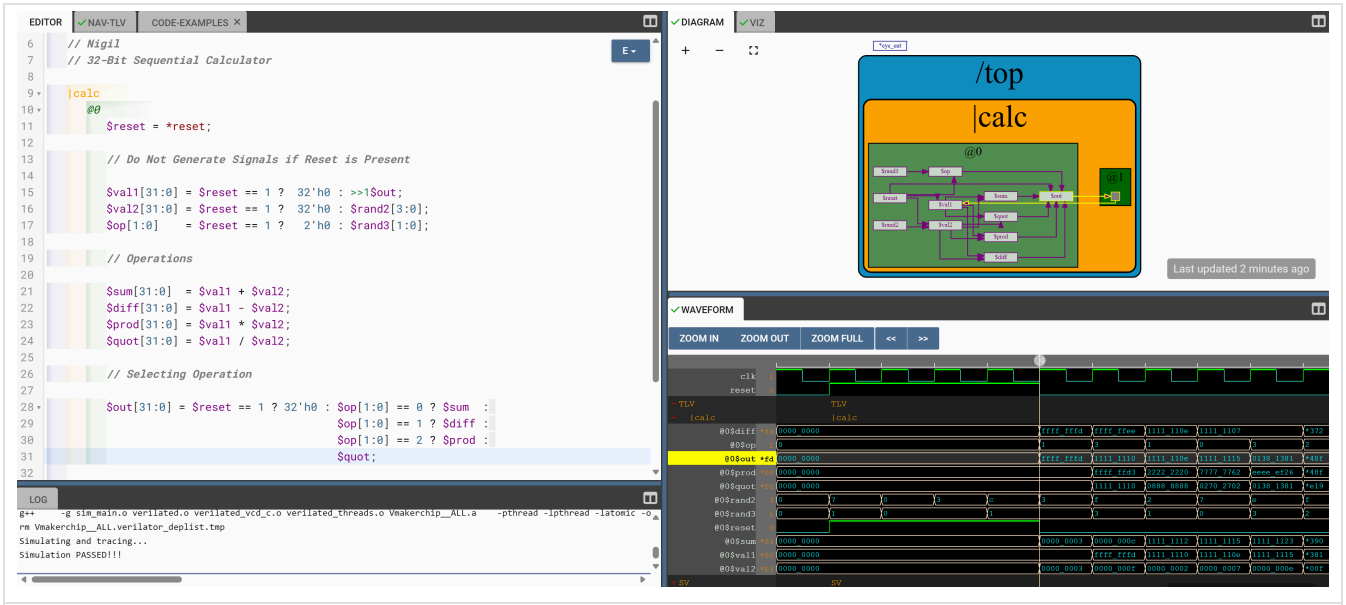
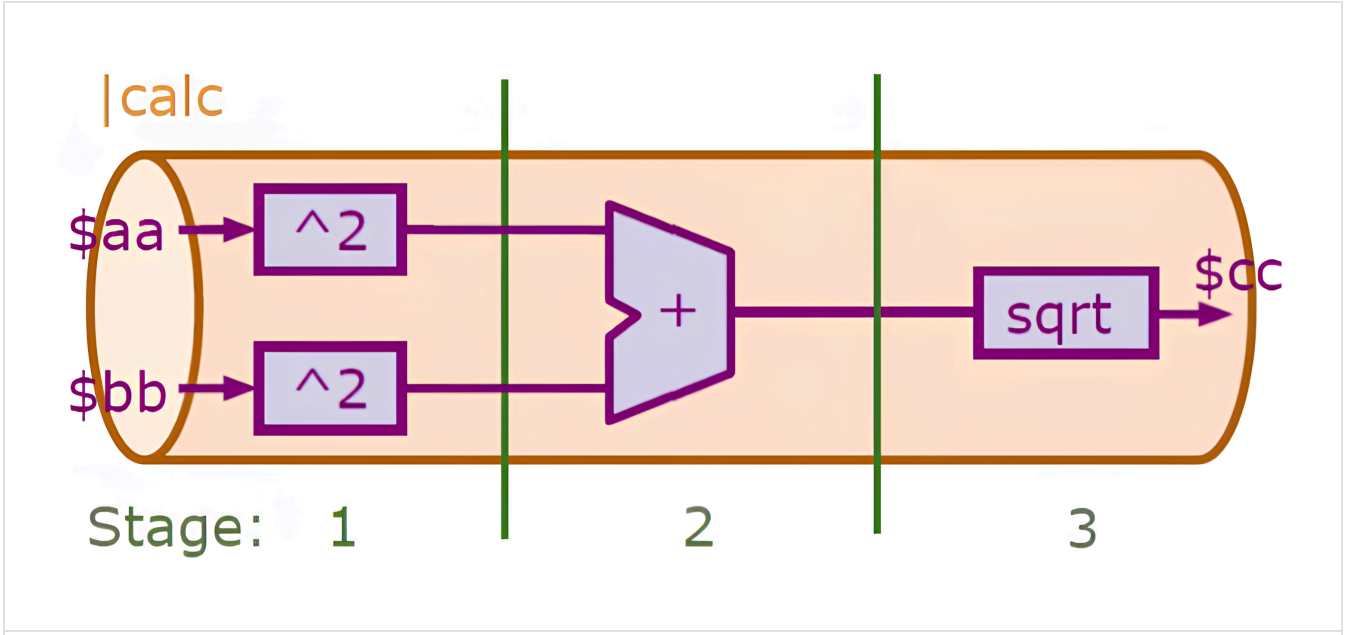


Figure 2. Sequential Calculator - Makerchip IDE Output

Pipelined Logic

Transaction-Level Verilog allows modeling of a design as timing abstracts. The following is a pipeline implementation of the **32-Bit Pythagorean Theorem**, which uses the timing-abstract concept. The green lines represent registers.



Architecture - A Simple Pipeline

```
\TLV

   |calc
      @1  // Stage
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqrt($cc_sq);
```

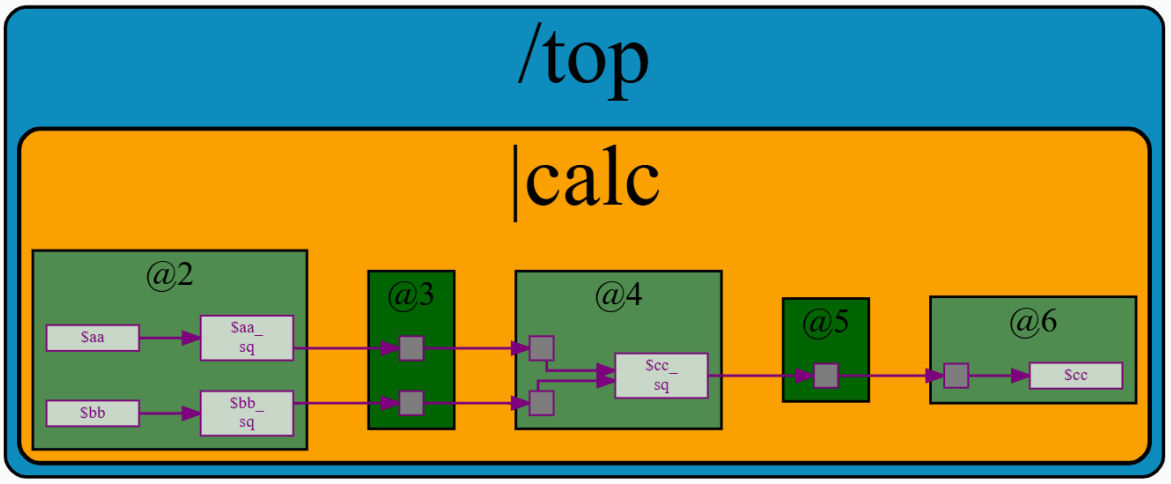
Stage one can be divided into two separate stages without affecting the behavior of the circuit. The pipeline stages are a physical attribute. TL-Verilog offers greater flexibility than SystemVerilog and helps avoid retiming issues.

Retiming the Pipeline : Pythagorean Theorem Example

```
\TLV

   // Design Under Test

   |calc
      @2 // Stage
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @4
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @6
         $cc[31:0] = sqrt($cc_sq);
```



Architecture - A Retimed Pipeline

Identifiers and Types (Misc)

The type of an identifier is determined by its symbol prefix and case/delimitation style. The first token must always start with two alphabet characters. Numbers cannot appear at the beginning of the tokens; they can only be at the end or in the middle. This should not be confused with number identifiers like `>>1`.

Token Name	Signal Type
`$lower_case`	Pipeline Signal
`$CamelCase`	State Signal
`$UPPER_CASE`	Keyword Signal

Lab : Error Conditions Within Computation Pipeline

The `ERROR_SIGNALS` are OR together to check the various error conditions that can occur within a computational pipeline. The idea of this exercise is to develop a visual understanding of how the TL-

Verilog code is translated into the diagram / architecture in the visualization pane.

```
\TLV

   // Error in Pipeline

   |error
      @1
         $err1 = $bad_input || $illegal_op;
      @3
         $err2 = $over_flow || $err1;
      @6
         $err3 = $div_by_zero || $err2;

   // Limiting Testing Cycles

   *passed = *cyc_cnt > 10;
   *failed = 1'b0;
```

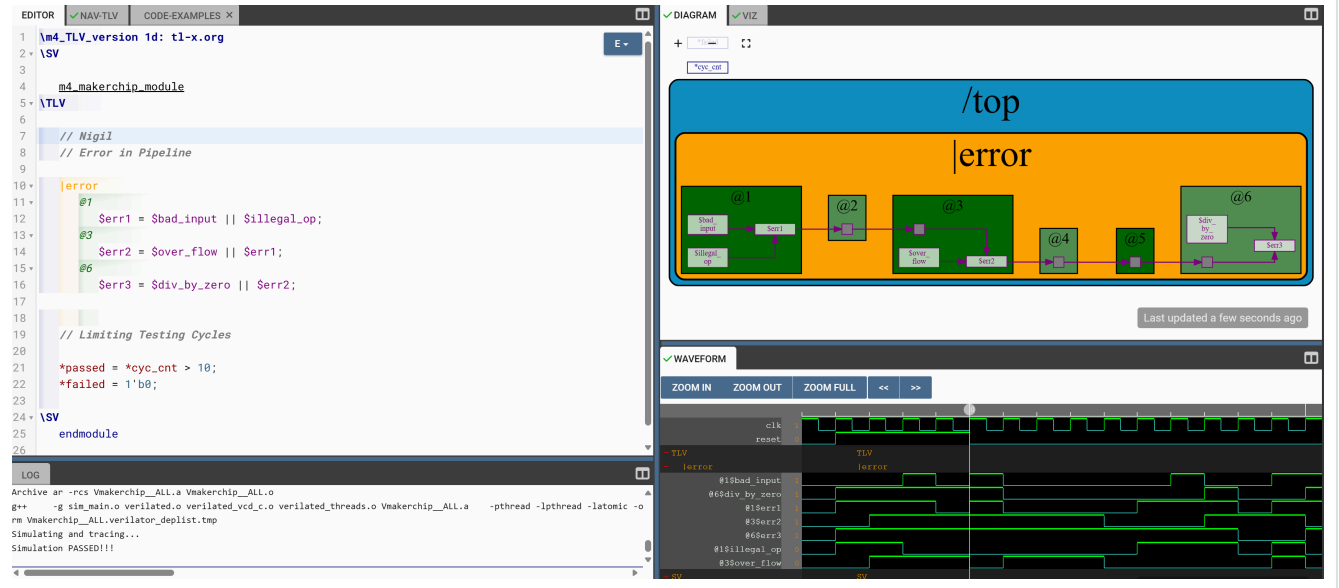


Figure 3. Errors in Pipeline During Computation - Makerchip IDE Output (Excercise)

Lab : Two-Cycle Calculator (Pipeline)

The calculation happens in the first cycle, and in the second cycle, the outputs are assigned based on the `VALID SIGNAL`, which is determined by `$reset | !cnt`.

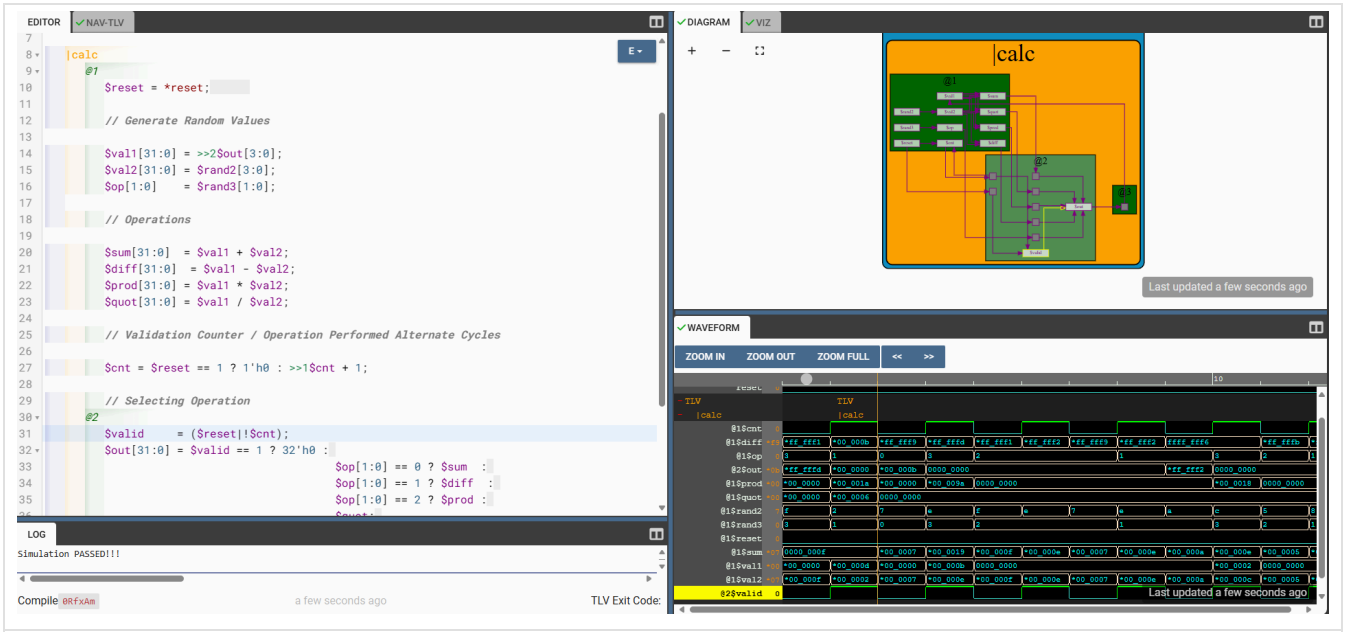


Figure 4. Two-Cycle (Pipelined) Calculator - Makechip IDE Output

Structure of TL-Verilog Code (Misc)

```
\m4_TLV_version 1d: tl-x.org
```

The above line specifies the version of TL-Verilog, and `tl-x.org` provides the documentation link. `M4` is a macro language, which, when used, expands in the navigation window of the Maker chip IDE, defining the input, output, clock, and reset signals of the module.

Validity

Validity offers easier debugging, cleaner design, better error checking, and automated clock gating. It allows Sandpiper to inject `DONT_CARES` when the inputs are not valid. The syntax of valid is `?$valid`.

Example : Accumulation of Distance

```
\TLV

   |calc
      @1
         $reset = *reset;

      ?$valid
         @1 // Stage
            $aa_sq[31:0] = $aa[3:0] * $aa[3:0];
            $bb_sq[31:0] = $bb[3:0] * $bb[3:0];
         @2
            $cc_sq[31:0] = $aa_sq + $bb_sq;
         @3
            $cc[31:0]    = sqrt($cc_sq);

      @4
         $tot_dist[63:0] = $reset ? '0 :
                           $valid ? >>1$tot_dist + $cc : // Accumulate
                                    >>1$tot_dist;        // Retain
```

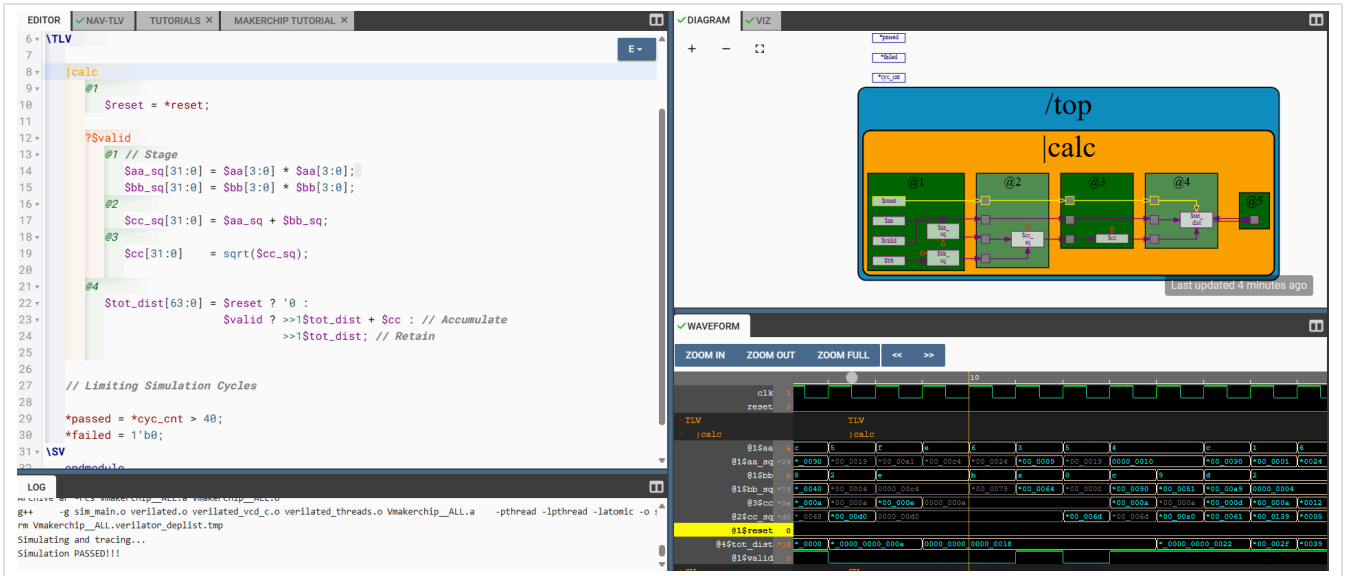


Figure 5. Total Distance Accumulator - Makechip IDE Output

A `VALID` signal is used to determine whether the distance is valid. If it is not valid, the previous value of the distance is held.

Lab : Two Cycle Calculator with Validity

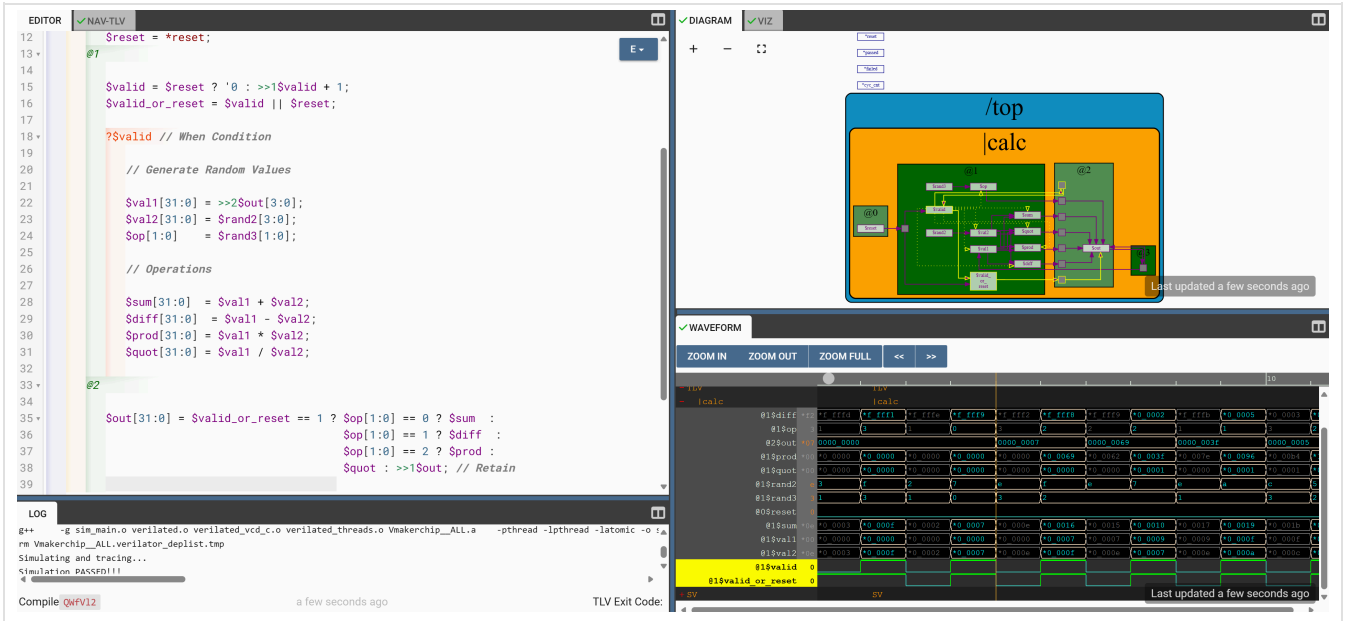


Figure 6. Pipelined Calculator with Validity Condition - Makechip IDE Output

Lab : Calculator with Single-Value Memory

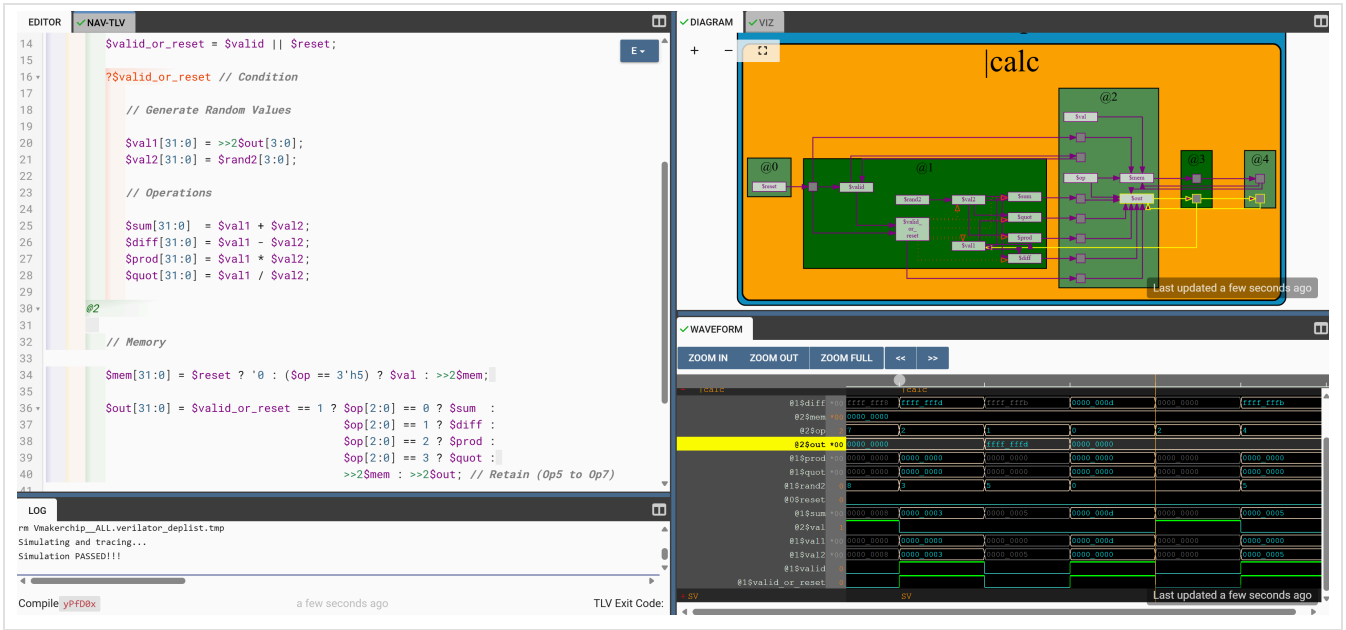


Figure 7. Pipelined Calculator with Single Value Memory - Makechip IDE Output

Lecture on Brief introduction on Hierarchy and Lexical Re-entrance using Conway's Game of Life is Skipped.

Basic RISC-V CPU Micro-Architecture

The RISC-V Shell can be found in the GitHub repository by Steve Hoover.

Fetch Address and Instruction

Program Counter and Instruction Memory

```
\TLV

   |cpu
      @0
         $reset = *reset;
         $pc[31:0] = (>>1$reset) ? '0 : >>1$pc + 32'h4;
```

For designing the instruction memory uncomment the macros `m4+imem(@1)` and `m4+cpu_viz(@4)`. Then `INST_MEM_EN` is activated in complement with the previous value of the `RST`.

```
// Below the Program Counter Statement

         $imem_rd_en          = !>>1$reset ? 1 : 0;
         $imem_rd_addr[31:0] = $pc[M4_IMEM_INDEX_CNT+1:2];

      @1
         $instr[31:0] = $imem_rd_data[31:0];
```

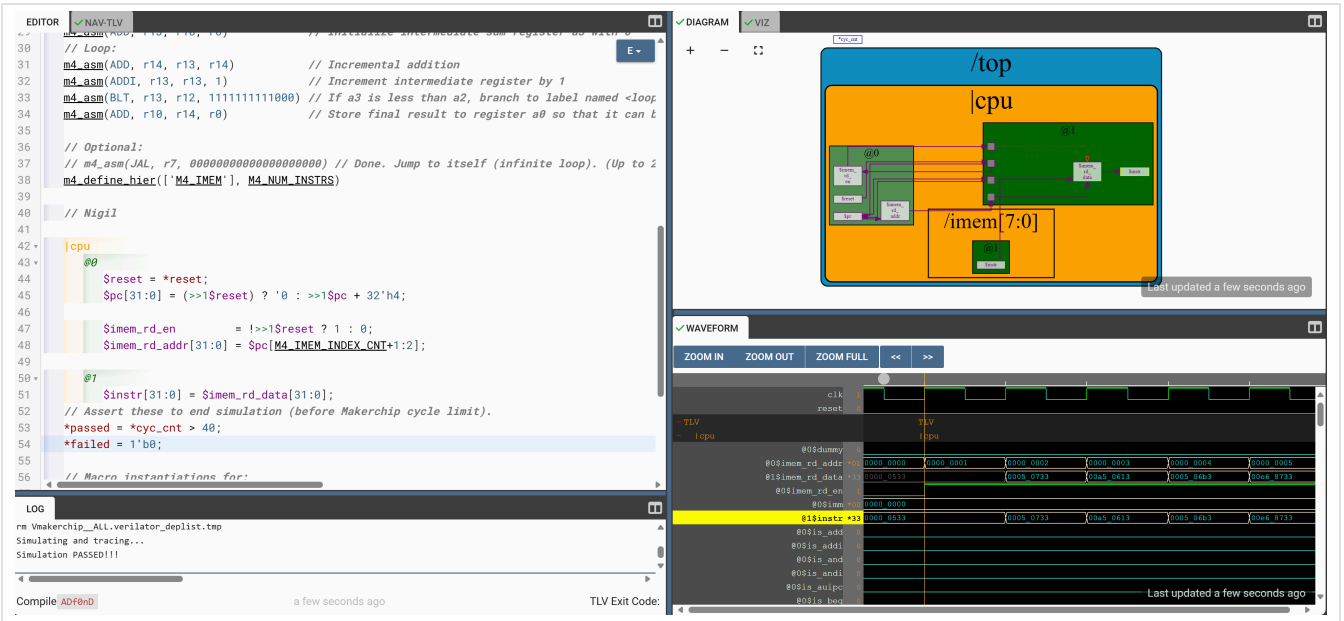


Figure 8. Fetch Address from Program Counter and Instruction Data - Makechip IDE Output

Decode Instructions

Decode Instruction Type

`instr[6:2]` determines instruction type: I, R, S, B, J and U. **The simple idea behind instruction decode logic design is to eliminate common cases between different instructions and create instances that identify the type of instruction based on their differences. For example, the difference can be a single bit, which can be represented as don't-cares**.

instr[4:2] instr[6:5]	000	001	010	011	100	101	110	111
00	I	I	-	-	I	U	I	-
01	S	S	-	R	R	U	R	-
10	R4	R4	R4	R4	R	-	-	-
11	B	I	-	J	I (unused)	-	-	-

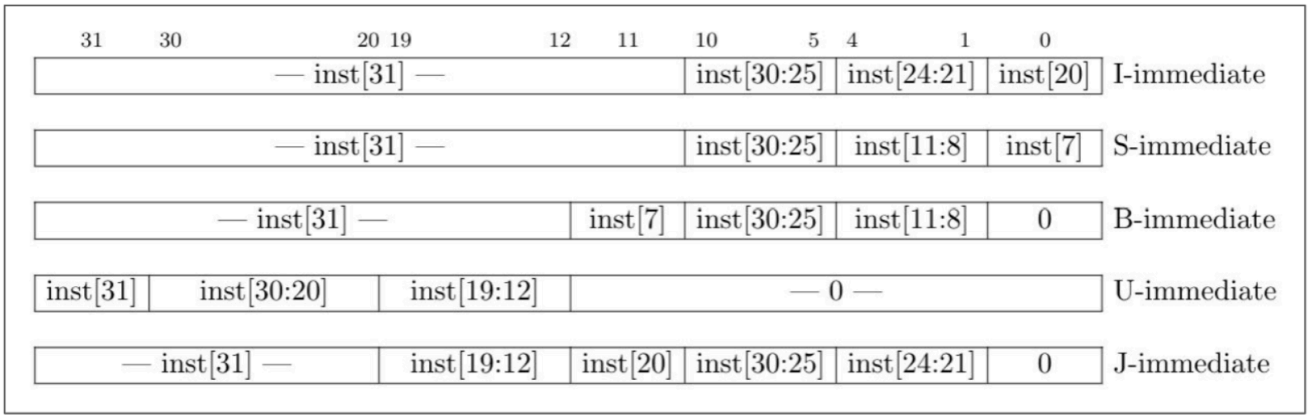
Instruction Types for RISC-V Processor

```
// Below Instruction Fetch Assignment

          $is_i_instr = $instr[6:2] ==? 5'b0000x || $instr[6:2] ==? 5'b001x0 || $instr[6:2] ==?
5'b11001;
          $is_r_instr = $instr[6:2] ==? 5'b01011 || $instr[6:2] ==? 5'b011x0 || $instr[6:2] ==?
5'b10100;
          $is_s_instr = $instr[6:2] == 5'b0100x;
          $is_u_instr = $instr[6:2] == 5'b0x101;
          $is_b_instr = $instr[6:2] == 5'b11000;
          $is_j_instr = $instr[6:2] == 5'b11011;
```

Decode Immediate Instructions

Form `$imm[31:0]` based on the instruction type.



Immediate Instruction Decode for RISC-V Processor

```
// Below Instruction Type Decode

         $imm[31:0] = $is_i_instr   ? {{21{$instr[31]}}, $instr[30:20]} :
                      $is_s_instr   ? {{21{$instr[31]}}, $instr[30:25], $instr[11:8],
$instr[7]} :
                      $is_b_instr   ? {{20{$instr[31]}}, $instr[7], $instr[30:25],
$instr[11:8], 1'b0} :
                      $is_u_instr   ? {$instr[31], $instr[30:20], $instr[19:12], 12'b0} :
                      $is_j_instr   ? {{12{$instr[31]}}, $instr[19:12], $instr[20],
$instr[30:21], 1'b0} :
                      32'b0;
```

Decode - Extracting Instruction Fields

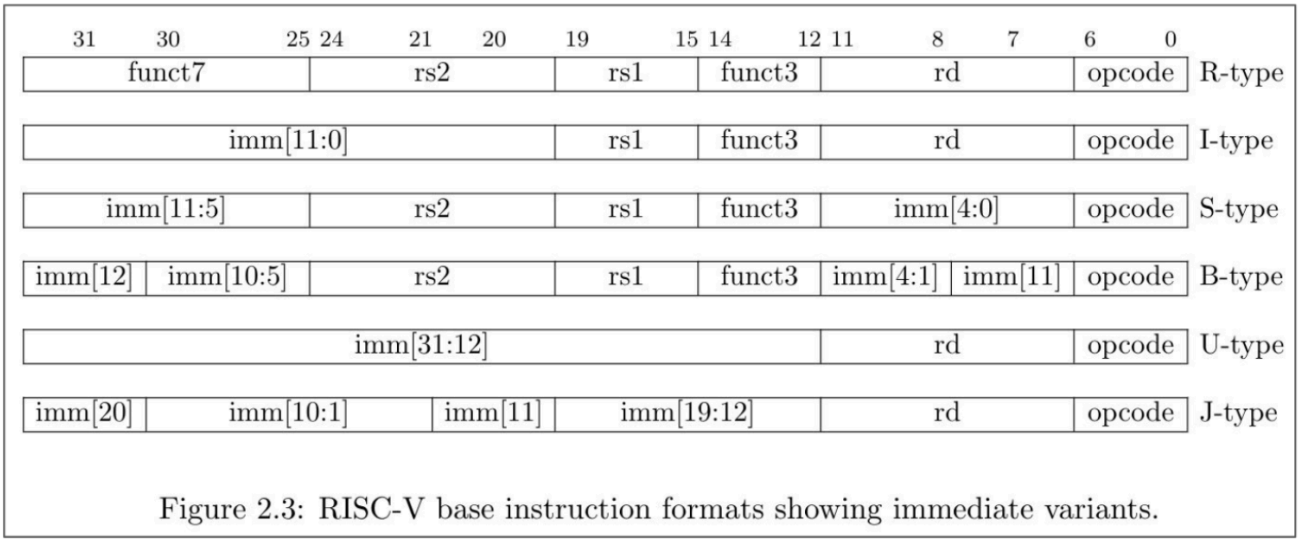


Figure 2.3: RISC-V base instruction formats showing immediate variants.

Immediate Instruction Decode for RISC-V Processor

```
// Below Immediate Instruction Decode, Note Immediate Instructions are Taken Care

         $rs1_valid    = $is_r_instr || $is_i_instr || $is_s_instr || $is_b_instr;
         $rs2_valid    = $is_r_instr || $is_s_instr || $is_b_instr;
         $rd_valid     = $is_r_instr || $is_i_instr || $is_u_instr || $is_j_instr;
         $funct3_valid = $is_r_instr || $is_i_instr || $is_s_instr || $is_b_instr;
```

```
$funct7_valid = $is_r_instr;

$opcode[6:0]     = $instr[6:0];

?$rs1_valid
   $rs1[4:0]     = $instr[19:15];

?$rs2_valid
   $rs2[4:0]     = $instr[24:20];

?$rd_valid
   $rd[4:0]      = $instr[11:7];

?$funct3_valid
   $funct3[2:0] = $instr[14:12];

?$funct7_valid
   $funct7[6:0] = $instr[31:25];
```

Decode Individual Instructions

imm[31:12]				rd	0110111
imm[31:12]				rd	0010111
imm[20|10:1|11|19:12]				rd	1101111
imm[11:0]		rs1	000	rd	1100111
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011
imm[11:0]		rs1	000	rd	0000011
imm[11:0]		rs1	001	rd	0000011
imm[11:0]		rs1	010	rd	0000011
imm[11:0]		rs1	100	rd	0000011
imm[11:0]		rs1	101	rd	0000011
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	010	rd	0010011
imm[11:0]		rs1	011	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011
0000000	shamt	rs1	001	rd	0010011
0000000	shamt	rs1	101	rd	0010011
0100000	shamt	rs1	101	rd	0010011
0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	001	rd	0110011
0000000	rs2	rs1	010	rd	0110011
0000000	rs2	rs1	011	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	101	rd	0110011
0100000	rs2	rs1	101	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011
fm | pred | succ		rs1	000	rd	0001111
000000000000		00000	000	00000	1110011
000000000001		00000	000	00000	1110011

RISCV-32 Basic Instruction Set

```
// Below Extract Instruction Fields Code

        $dec_bits[10:0] = {$funct7[5], $funct3, $opcode};

        $is_beq  = $dec_bits ==? 11'bx_000_1100011;
```

```
         $is_bne  = $dec_bits ==? 11'bx_001_1100011;
         $is_blt  = $dec_bits ==? 11'bx_100_1100011;
         $is_bge  = $dec_bits ==? 11'bx_101_1100011;
         $is_bltu = $dec_bits ==? 11'bx_110_1100011;
         $is_bgeu = $dec_bits ==? 11'bx_111_1100011;
         $is_addi = $dec_bits ==? 11'bx_000_0010011;
         $is_add  = $dec_bits ==? 11'b0_000_0110011;
```

Lab : Single-Cycle RISC-V Fetch and Decode Implementation

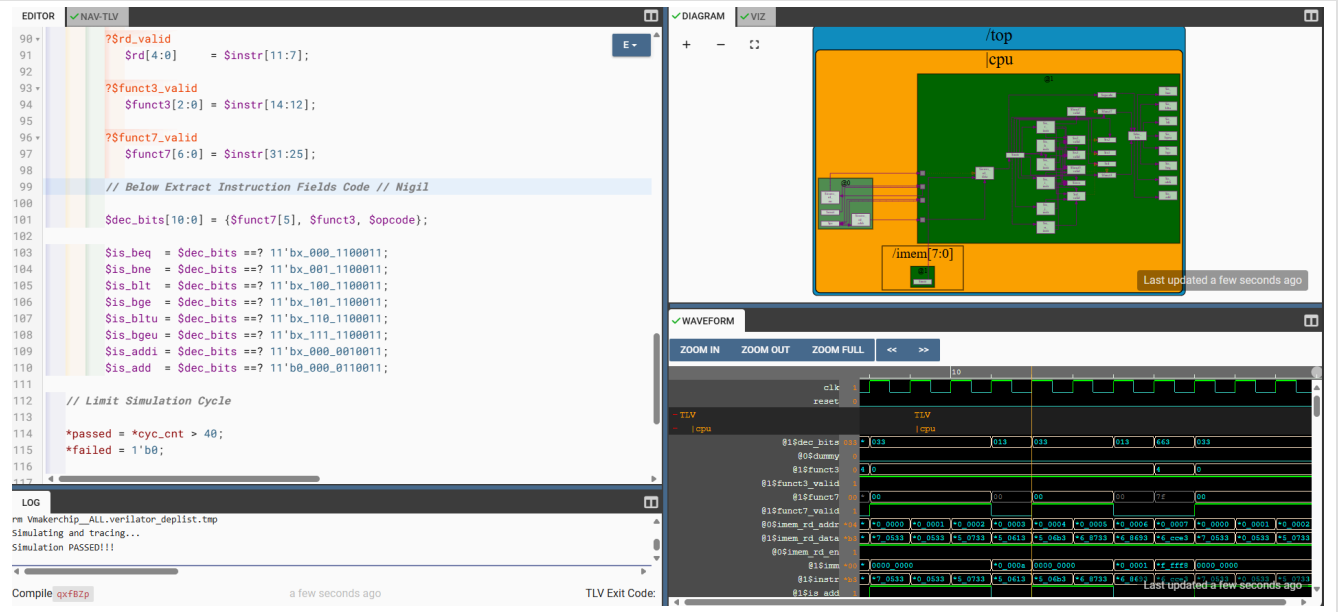
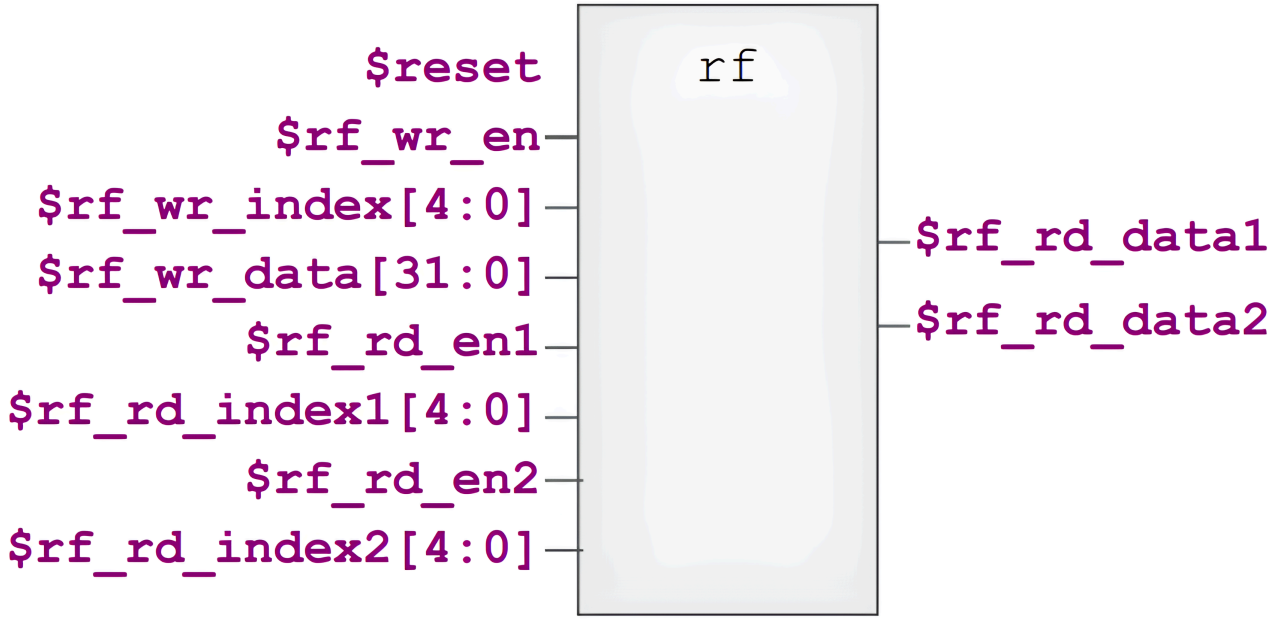


Figure 9. RISCV-32 Fetch and Decode Implementation - Makerchip IDE Output

Register File Read and Write

Use the decoded fields to write and read data to the registers. To generate the register file, uncomment the macro `m4+rf (@1, @1)`.



Dual-Read and Single Write Register File

Read

```
// Below Basic Instruction Set Decode, Read

         ?$rs1_valid
            $rf_rd_en1         = $rs1_valid;
            $rf_rd_index1[4:0] = $rs1[4:0];

         ?$rs2_valid
            $rf_rd_en2         = $rs2_valid;
            $rf_rd_index2[4:0] = $rs2[4:0];

         $src1_value[31:0] = $rf_rd_data1;
         $src2_value[31:0] = $rf_rd_data2;
```

Write

```
         // Below Arithemtic and Logic Unit, Register Write

         $rf_wr_en = ($rd == 5'h0) ? 1'b0 : $rd_valid;

         ?$rf_wr_en
            $rf_wr_index[4:0] = $rd[4:0];

         $rf_wr_data[31:0]  = $result[31:0];
```

Simple Arithmetic and Logic Unit (ALU) Design

```
         // Arithmetic and Logic Unit, Below Register File

         $result[31:0] = $is_addi ? $src1_value + $imm :
                         $is_add  ? $src2_value + $src1_value : 32'bx;
```

Branch Instructions

The Program Counter is modified to calculate the branch address based on the immediate value. If the `TAKEN_BRANCH` is high, the Program Counter is updated with the branch address; otherwise, the address is incremented by 4 by default.

```
         // Updated Program Counter

         $pc[31:0] = (>>1$reset) ? '0 : >>1$taken_br ? >>1$br_tgt_pc : >>1$pc + 32'h4;

         // Branch Instructions Check

         $taken_br = $is_beq  ? ($src1_value == $src2_value) :
                     $is_bne  ? ($src1_value != $src2_value) :
                     $is_blt  ? ($src1_value <  $src2_value) ^ ($src1_value[31] !=
$src2_value[31]) :
                     $is_bge  ? ($src1_value >  $src2_value) ^ ($src1_value[31] !=
$src2_value[31]) :
                     $is_bltu ? ($src1_value <= $src2_value) :
                     $is_bgeu ? ($src1_value >= $src2_value) :
                     1'b0;

         // Branch Instruction Address Update
```

```
            $br_tgt_pc[31:0] = $pc + $imm;
```

Lab : Single-Cycle RISC-V32I Implementation

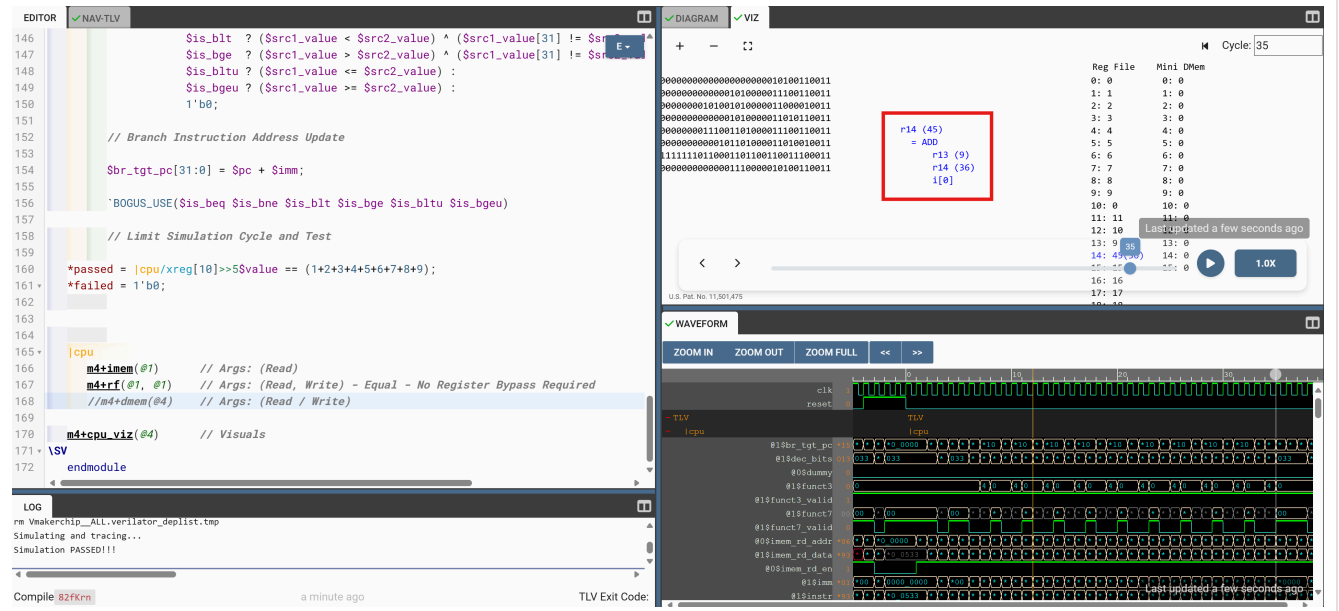


Figure 10. Single Cycle RISC-V Micro-Architecture Implementation - Makerchip IDE Output

Pipelining the RISC-V CPU Micro-Architecture

Pipelining and Hazards

Pipelining is done to improve the throughput of instruction execution by allowing multiple instructions to be processed simultaneously, but at different stages of execution. In a pipelined processor, the execution of an instruction is divided into several stages **Fetch, Decode, Execute, Memory Access and Write Back**.

While pipelining offers significant performance benefits, it also introduces some challenges like **Data Hazards, Control Hazards (Branch), Structural Hazards, Pipelined Stalls and Bubble Insertions**.

The **Data Hazards** occur when one instruction depends on the result of another instruction that has not completed its execution. **Read-after-Write (RAW)** is when an instruction reads data that is yet to be written by a previous instruction. **Write-after-Write (WAW)** is when two instructions write to the same register, but the write order must be preserved. **Write-after-Read (WAR)** is when a register is written to before it is read by a following instruction.

The **Control Hazards** arise from branch instructions (conditional jumps), where the processor needs to determine the next instruction to execute. If a branch is mis-predicted, the pipeline must be flushed, and the correct instructions need to be fetched, resulting in a performance penalty.

The **Structural Hazards** occur when the hardware resources required by multiple instructions in the pipeline conflict.

Lab : Pipelined RISC-V Processor

Based on the RISC-V architecture in `D04_SLIDE37`, modify the pipeline design by changing the macro `m4+rf(@1, @1)` to `m4+rf(@2, @3)`. Additionally, add a `VALID_SIGNAL` to validate the instructions.

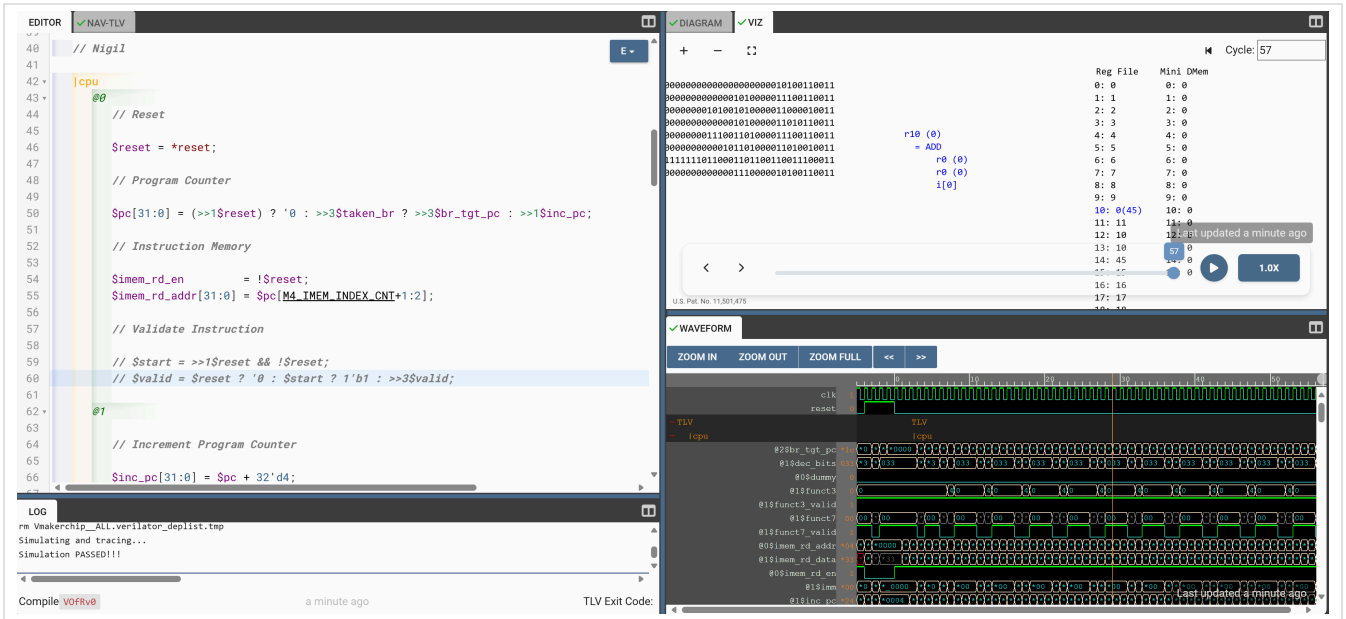


Figure 11. Pipelined RISC-V Micro-Architecture Implementation - Makerchip IDE Output

Data Memory and Jump Instruction Support

Lab : Data Memory - Load and Store Instructions (Memory Access and Write-Back)

The data memory has a similar implementation to the instruction memory. Uncomment the macro `m4+dmem(@4)`. Add `VALID_SIGNALS` for load and store instructions to avoid data hazards.

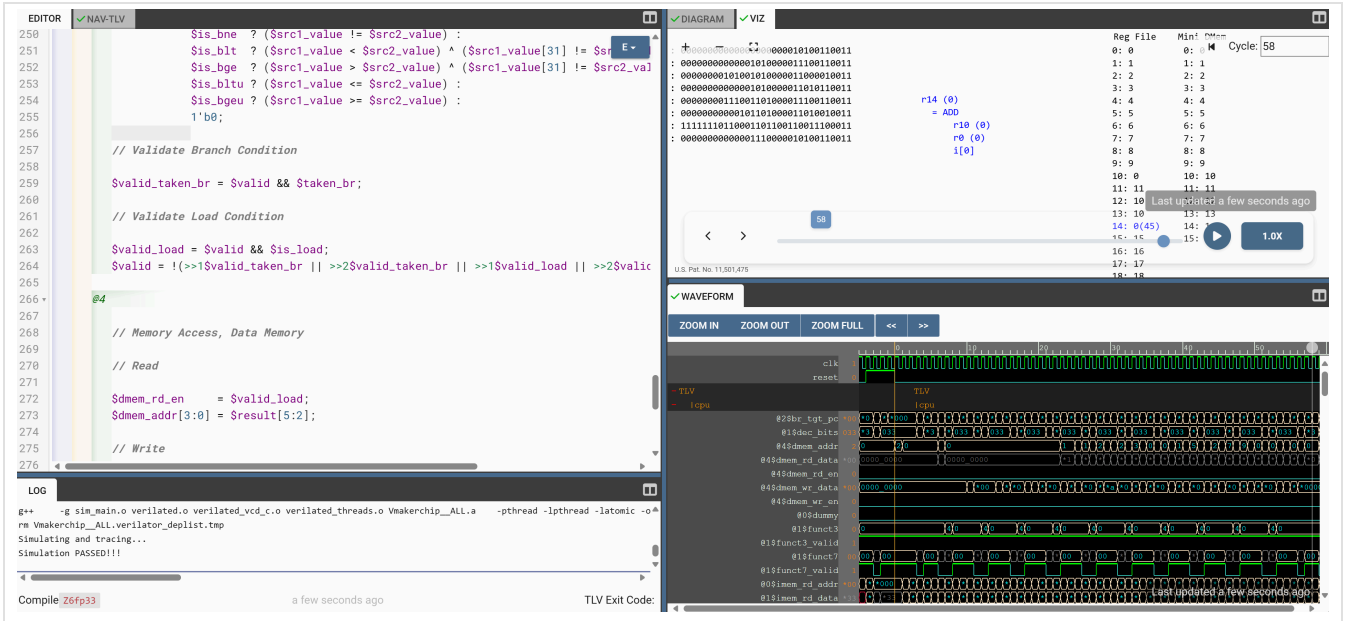


Figure 12. Pipelined RISC-V Micro-Architecture with Load and Store Instruction Implementation - Makerchip IDE Output

Lab : Five Stage Pipelined RISC-V Processor (Complete Implementation)

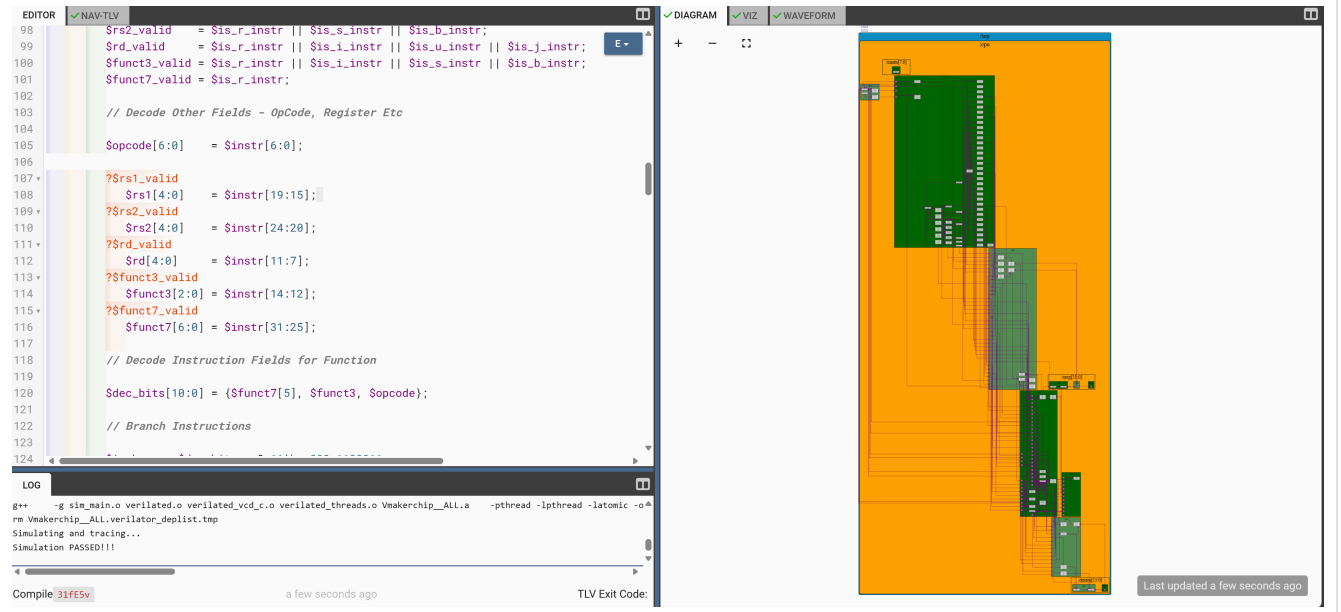


Figure 13. Five State Pipelined RISC-V Micro-Architecture - Makerchip IDE Output

Acknowledgments

1. Steve Hoover Founder, Redwood EDA
2. Kunal Ghosh Co-founder, VSD Corp. Pvt. Ltd.
3. Shivani Shah Hardware Architect, Nvidia