RISC-V RV32IM

Digital VLSI Design Project

Project Report

Made by:

Aniketh Reddimi – 2018102014

Dabbiru Vamshi Kashyap – 2018102042

Course Instructor: Zia Abbas

Teaching Assistant: Salman Ahmed

Table of Contents

The RV32I Base ISA

RV32I is the base 32-bit integer ISA. It is a simple instruction set, comprising just 47 instructions, yet it is complete enough to form a compiler target and satisfy the basic requirements of modern operating systems and runtimes.

Eight of the instructions are system instructions (system calls and performance counters) that can be implemented as a single trapping instruction, reducing the number of mandatory user-level hardware instructions to 40. As with many RISC instruction sets, the remaining instructions fall into three categories: computation, control flow, and memory access. RISC-V is a load-store architecture, in which arithmetic instructions operate only on the registers, and only loads and stores transfer data to and from memory.

There are 31 general-purpose registers $x1$– $x31$, which hold integer values. Register $x0$ is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register $x1$ to hold the return address on a call. For RV32, the x registers are 32 bits wide, and for RV64, they are 64 bits wide.

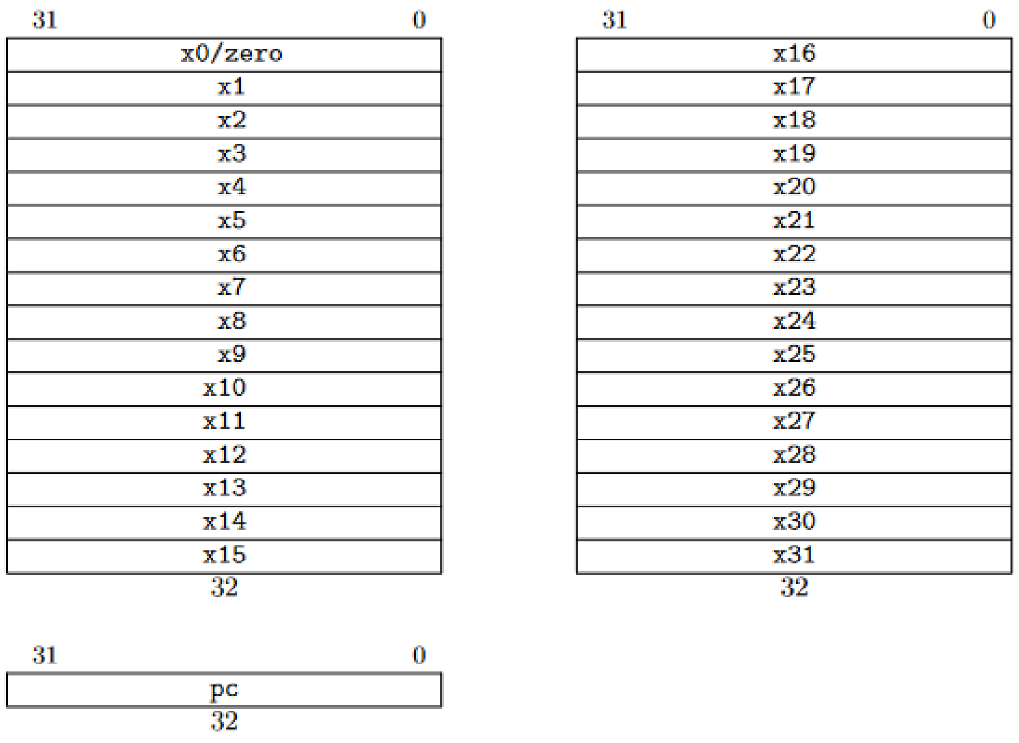


Figure 1 - RV32I user-visible architectural state

Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U). All area fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

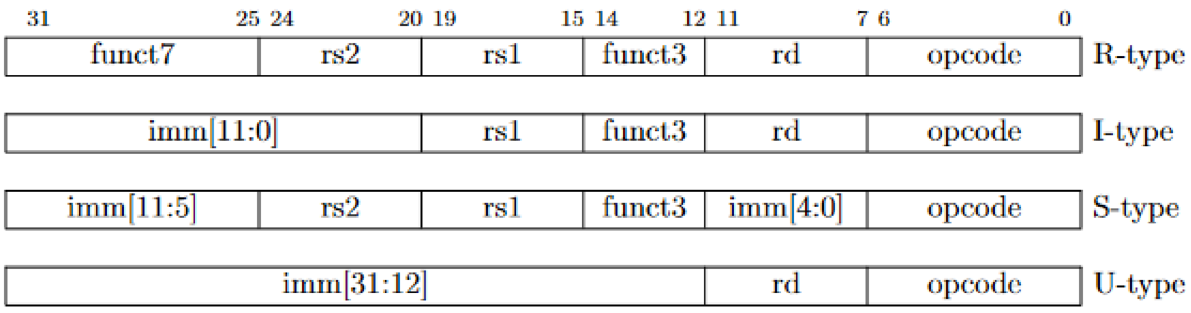

31 25 24 20 19 15 14 12 11 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[31:12] | rd | opcode | U-type

Figure 2 - RISC-V base instruction formats

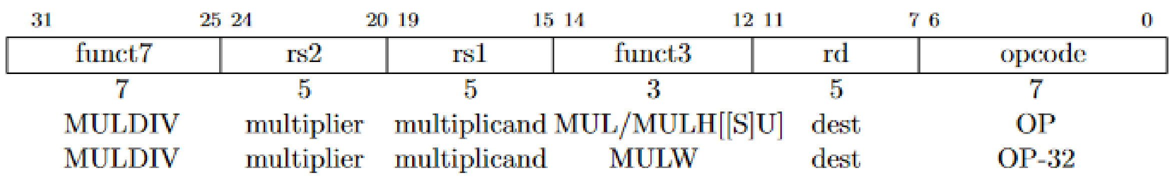
Instruction	Format	Meaning
`add   rd, rs1, rs2`	R	Add registers
`sub   rd, rs1, rs2`	R	Subtract registers
`sll   rd, rs1, rs2`	R	Shift left logical by register
`srl   rd, rs1, rs2`	R	Shift right logical by register
`sra   rd, rs1, rs2`	R	Shift right arithmetic by register
`and   rd, rs1, rs2`	R	Bitwise AND with register
`or    rd, rs1, rs2`	R	Bitwise OR with register
`xor   rd, rs1, rs2`	R	Bitwise XOR with register
`slt   rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu  rd, rs1, rs2`	R	Set if less than register, unsigned
`addi  rd, rs1, imm[11:0]`	I	Add immediate
`slli  rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli  rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai  rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi  rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori   rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori  rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti  rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui   rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to pc

Figure 3 - Listing of RV32I computational instructions

“M” Standard Extension

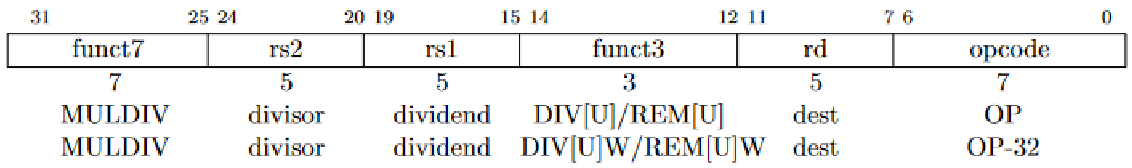
This symbol describes the standard integer multiplication and division instruction extension, which is named “M” and contains instructions that multiply or divide values held in two integer registers. We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

- **Multiplication Operations:**
  MUL performs an $XLEN - bit \times XLEN - bit$ multiplication and places the lower $XLEN$ bits in the destination register. $MULH$, $MULHU$ and $MULHSU$ perform the same multiplication but return the upper $XLEN$ bits of the full 2×$XLEN$-bit product, for $signed \times signed$, $unsigned \times unsigned$, and $signed \times unsigned$ multiplication respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: $MULH[[S]U]\ rdh$, $rs1$, $rs2$; $MUL\ rdl$, $rs1$, $rs2$ (source register specifiers must be in same order and rdh cannot be the same as $rs1$ or $rs2$). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

- **Division Operations:**
  DIV and $DIVU$ perform signed and unsigned integer division of $XLEN$ bits by $XLEN$ bits. REM and $REMU$ provide the remainder of the corresponding division operation. If both the quotient and remainder are required from the same division, the recommended code sequence is: $DIV[U]\ rdq, rs1, rs2$; $REM[U]\ rdr, rs1, rs2$ (rdq cannot be the same as $rs1$ or $rs2$). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

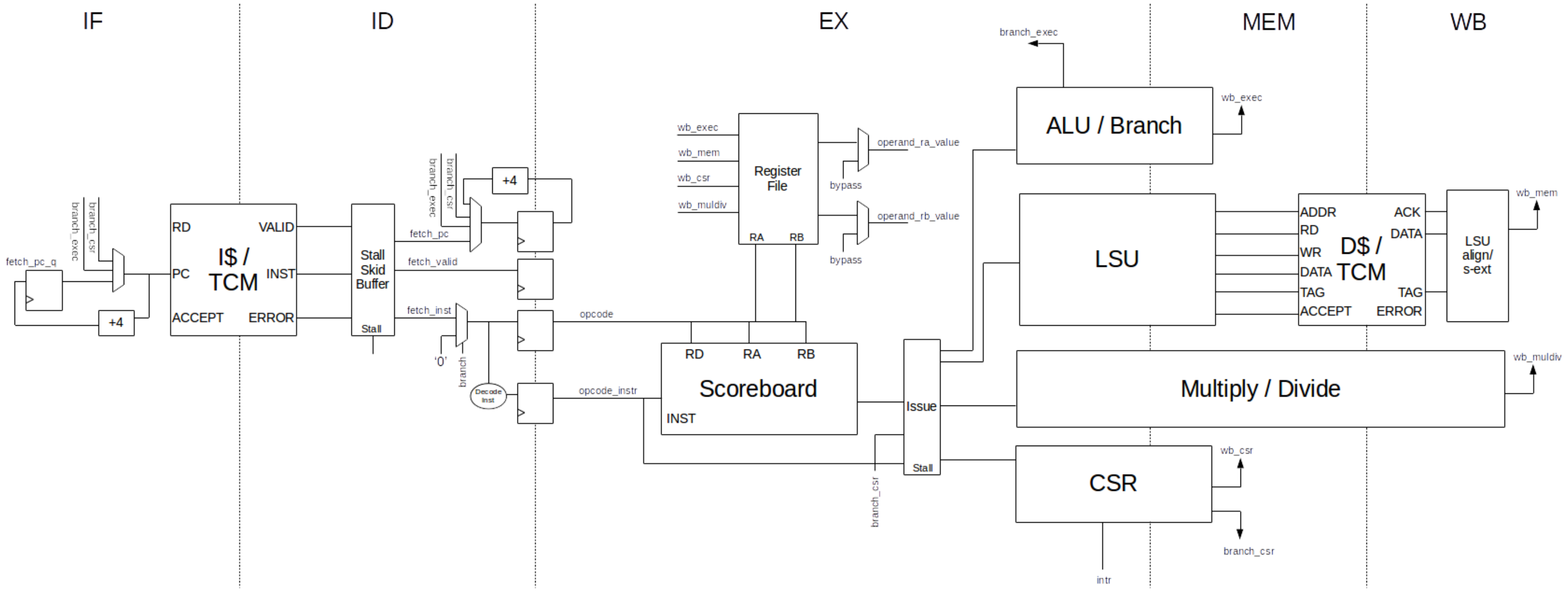


31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32

To summarize in a simple manner, RV32M adds four instructions that compute 32×32-bit products: MUL, which returns the lower 32 bits of the product; and $MULH$, $MULHU$ and $MULHSU$ which return the upper 32 bits of the product, treating the multiplier and multiplicand as signed, unsigned, and mixed, respectively. (The lower 32 bits of the product do not depend on the signees of the inputs.) The latter three instructions are essential for fixed-point computational libraries and enable an important strength reduction optimization: division by a constant can always be turned into multiplication by an approximate reciprocal, followed by a correction step to the upper half of the product [34]. There are also four instructions that perform 32-bit by 32-bit division: DIV and $DIVU$, for signed and unsigned division; and REM and $REMU$, for signed and unsigned remainder. Following C99, signed division rounds toward zero, and the remainder has the same sign as the dividend. Division by zero does not cause an exception; programming languages that desire this behaviour can branch on the divisor after initiating the division operation. This highly predictable branch should have little effect on performance.

Overview of the schematic

This is a shown as a Superscalar (dual-issue) in-order 6 or 7 stage pipeline. Things are toned downed later according to our requirements.



Components of the code

Our code is divided into multiple pipelines and each pipeline is joined with pipeline registers and such. Forwarding unit and Hazard Detection unit are implemented too.

Instruction Fetch (IF)

According to the instruction address in the program counter PC, an instruction is fetched from the instruction memory and sent to the decoding module. At the same time, the PC generates the instruction address required for the next instruction according to the auto-increment, but when a branch instruction is encountered, if the branch is established, the controller needs to send the jump address to the PC.

Instruction Decoding (ID)

The instruction obtained in the instruction fetch operation is analysed and decoded to determine the operation that this instruction needs to complete, to generate a corresponding operation control signal for driving various operations in the execution state.

Pipeline Registers

These are used to store and pass data. It takes inspiration from flip-flop with a negative-edge clock and asynchronous clear.

Register

We have written a register file module which operates such as reading the register and obtaining the operand according to the source register by the ID module and writing the destination register by the WB module to the corresponding register.

Execute (EX)

So according to the ALU control signal obtained by the instruction decoding, the instruction operation is executed specifically.

Memory access (MEM)

All operations that need to access the memory will be performed in this step. This step writes data to the storage unit (store word) specified by the data address in the memory or obtains the data in the data address unit (load word) from the memory.

Write Back (WB)

The result of the instruction execution or the data obtained by accessing the memory is written back to the corresponding destination register.

Hazard detection

Load-use data hazard not only needs to be forwarded, but also PC and IF/ID register stall, so the PCwre signal is given to control the blocking of the pipeline. rs1 and rs2 are to ensure that the source register number divided from the instruction is meaningful. MemRead = 1 indicates that the previous instruction is an load instruction.

Forwarding

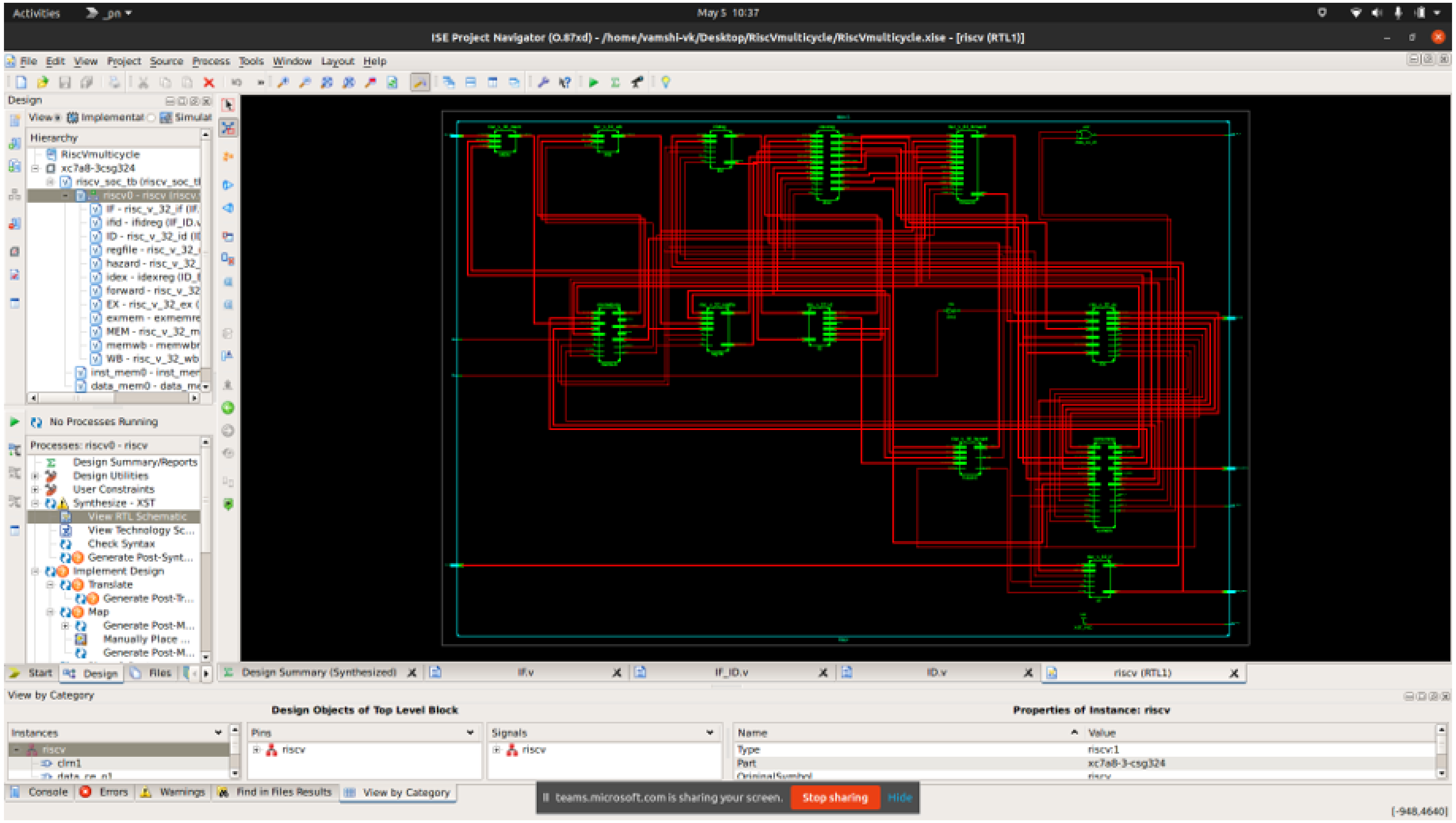
When a data hazard occurs, the data needs to be pushed forward before being written back to the register, otherwise the data read from the register file may be data that has not yet been updated. When the destination register number of the previous instruction is the same as the source register number of the current instruction, forwarding is required. Forwarding data may come from EX-stage ALUresult, MEM-stage ALUresult, or load-use hazard from the result of data storage access.

Prediction

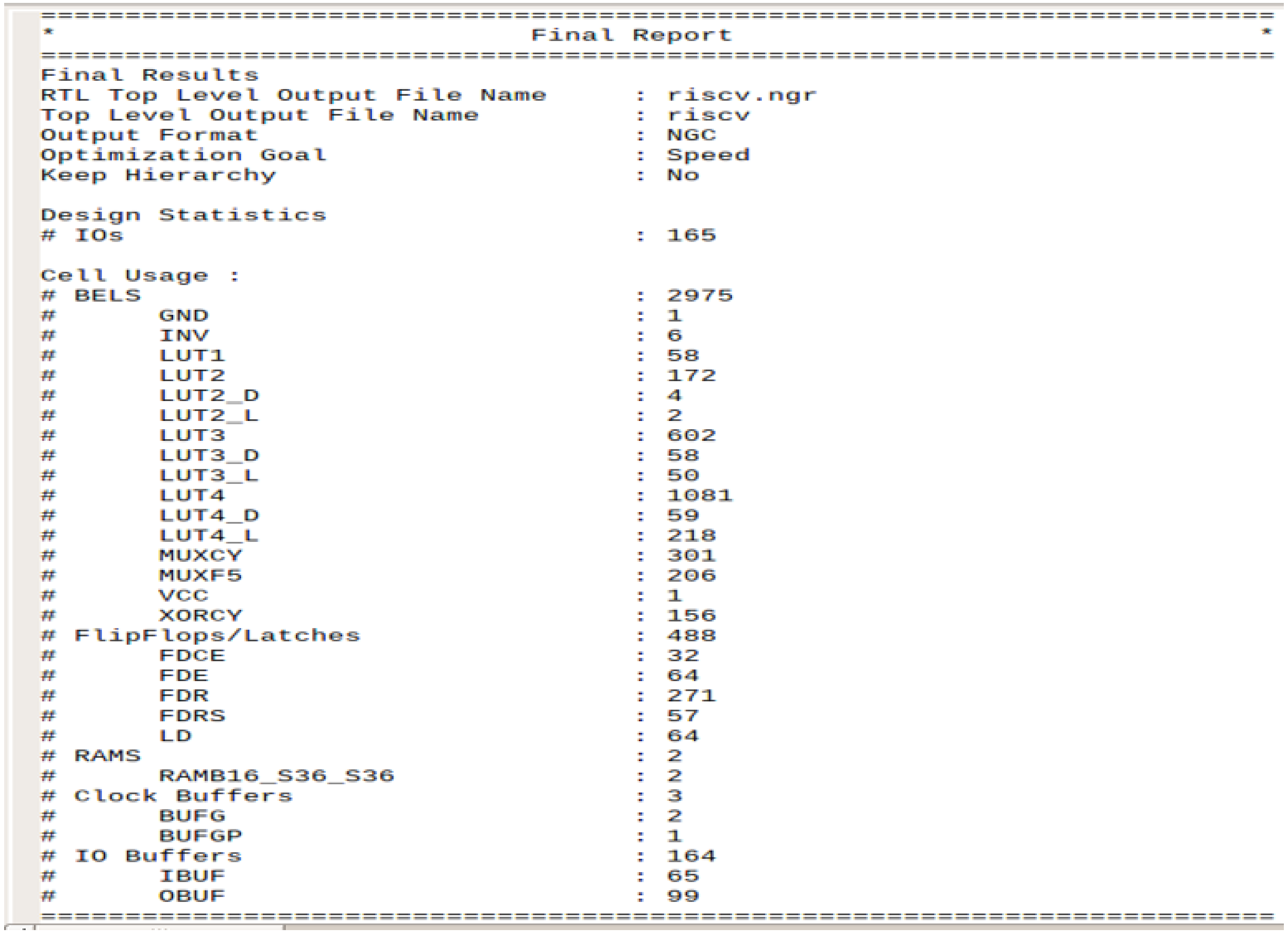
Stalling until the branch is complete is too slow. One improvement over branch stalling is to predict that the conditional branch will not be taken and thus continue execution down the sequential instruction stream. If the conditional branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If conditional branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards. The PCSrc will be modified if branch is taken. The PCSrc assert produce in ex state shown following, there are also other modify in other modules.

Deliverables

- **RTL Schematic**



- **Synthesis Report**

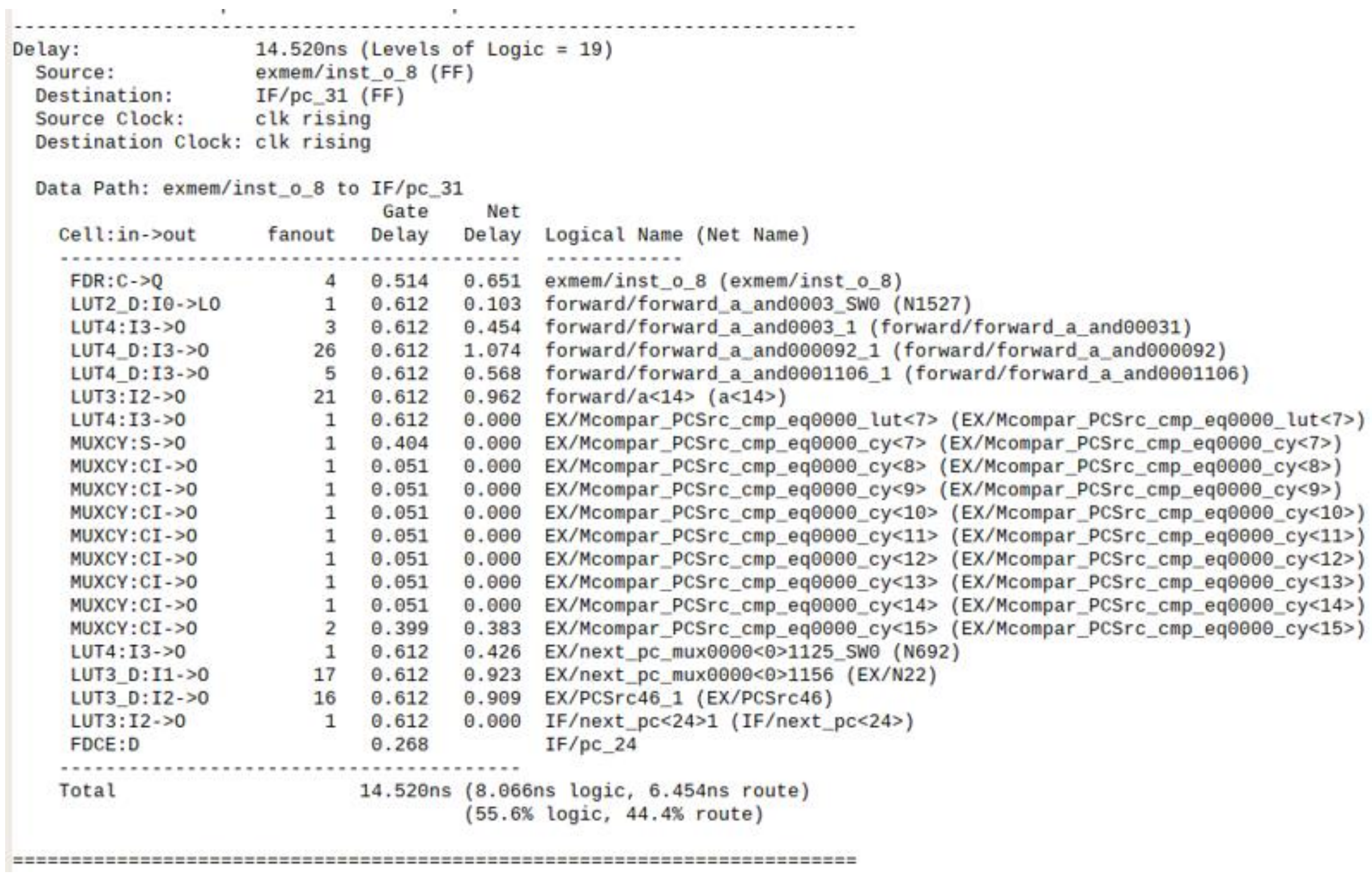


```
=========================================================================
*                          Final Report                                 *
=========================================================================
Final Results
RTL Top Level Output File Name     : riscv.ngr
Top Level Output File Name         : riscv
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : No

Design Statistics
# IOs                              : 165

Cell Usage :
# BELS                             : 2975
#      GND                         : 1
#      INV                         : 6
#      LUT1                        : 58
#      LUT2                        : 172
#      LUT2_D                      : 4
#      LUT2_L                      : 2
#      LUT3                        : 602
#      LUT3_D                      : 58
#      LUT3_L                      : 50
#      LUT4                        : 1081
#      LUT4_D                      : 59
#      LUT4_L                      : 218
#      MUXCY                       : 301
#      MUXF5                       : 206
#      VCC                         : 1
#      XORCY                       : 156
# FlipFlops/Latches                : 488
#      FDCE                        : 32
#      FDE                         : 64
#      FDR                         : 271
#      FDRS                        : 57
#      LD                          : 64
# RAMS                             : 2
#      RAMB16_S36_S36              : 2
# Clock Buffers                    : 3
#      BUFG                        : 2
#      BUFGP                       : 1
# IO Buffers                       : 164
#      IBUF                        : 65
#      OBUF                        : 99
=========================================================================
```

- **Delay**

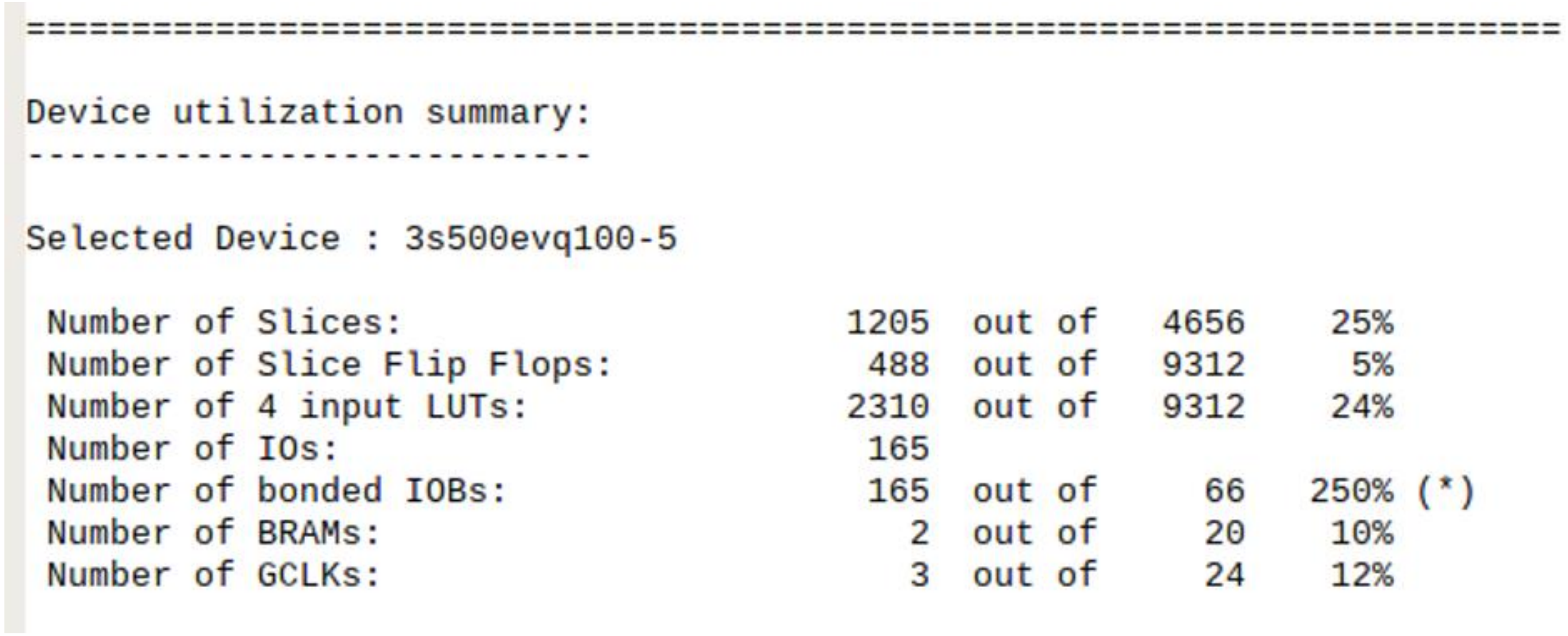


```
---------------------------------------------------------------------------
Delay:               14.520ns (Levels of Logic = 19)
  Source:            exmem/inst_o_8 (FF)
  Destination:       IF/pc_31 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: exmem/inst_o_8 to IF/pc_31
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDR:C->Q              4   0.514   0.651  exmem/inst_o_8 (exmem/inst_o_8)
     LUT2_D:I0->LO         1   0.612   0.103  forward/forward_a_and0003_SW0 (N1527)
     LUT4:I3->O            3   0.612   0.454  forward/forward_a_and0003_1 (forward/forward_a_and00031)
     LUT4_D:I3->O         26   0.612   1.074  forward/forward_a_and000092_1 (forward/forward_a_and000092)
     LUT4_D:I3->O          5   0.612   0.568  forward/forward_a_and0001106_1 (forward/forward_a_and0001106)
     LUT3:I2->O           21   0.612   0.962  forward/a<14> (a<14>)
     LUT4:I3->O            1   0.612   0.000  EX/Mcompar_PCSrc_cmp_eq0000_lut<7> (EX/Mcompar_PCSrc_cmp_eq0000_lut<7>)
     MUXCY:S->O            1   0.404   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<7> (EX/Mcompar_PCSrc_cmp_eq0000_cy<7>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<8> (EX/Mcompar_PCSrc_cmp_eq0000_cy<8>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<9> (EX/Mcompar_PCSrc_cmp_eq0000_cy<9>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<10> (EX/Mcompar_PCSrc_cmp_eq0000_cy<10>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<11> (EX/Mcompar_PCSrc_cmp_eq0000_cy<11>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<12> (EX/Mcompar_PCSrc_cmp_eq0000_cy<12>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<13> (EX/Mcompar_PCSrc_cmp_eq0000_cy<13>)
     MUXCY:CI->O           1   0.051   0.000  EX/Mcompar_PCSrc_cmp_eq0000_cy<14> (EX/Mcompar_PCSrc_cmp_eq0000_cy<14>)
     MUXCY:CI->O           2   0.399   0.383  EX/Mcompar_PCSrc_cmp_eq0000_cy<15> (EX/Mcompar_PCSrc_cmp_eq0000_cy<15>)
     LUT4:I3->O            1   0.612   0.426  EX/next_pc_mux0000<0>1125_SW0 (N692)
     LUT3_D:I1->O         17   0.612   0.923  EX/next_pc_mux0000<0>1156 (EX/N22)
     LUT3_D:I2->O         16   0.612   0.909  EX/PCSrc46_1 (EX/PCSrc46)
     LUT3:I2->O            1   0.612   0.000  IF/next_pc<24>1 (IF/next_pc<24>)
     FDCE:D                    0.268          IF/pc_24
    ----------------------------------------
    Total                    14.520ns (8.066ns logic, 6.454ns route)
                                      (55.6% logic, 44.4% route)

===========================================================================
```

- **Device Utilization**



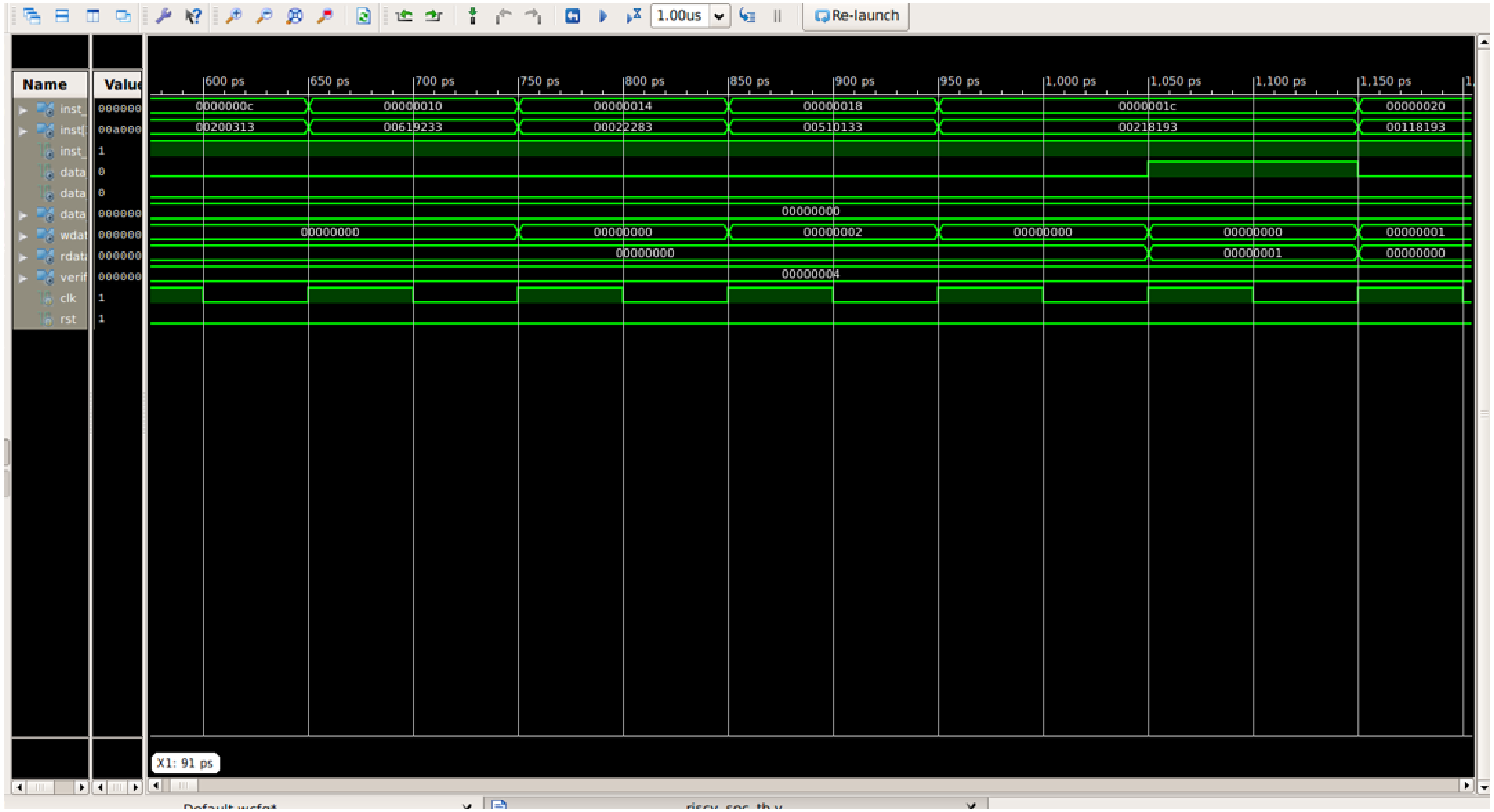
```
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 3s500evq100-5

 Number of Slices:                      1205  out of   4656    25%
 Number of Slice Flip Flops:             488  out of   9312     5%
 Number of 4 input LUTs:                2310  out of   9312    24%
 Number of IOs:                          165
 Number of bonded IOBs:                  165  out of     66   250% (*)
 Number of BRAMs:                          2  out of     20    10%
 Number of GCLKs:                          3  out of     24    12%
```

- **Timing Diagram**



Screen Record of the Deliverables

Running of the code and all the required deliverables have been screen recorded and uploaded on the google drive

Link:
https://drive.google.com/file/d/1wFNdwBWVXMJTaXrlAu0uMh7QKlT23EC7/view

Resources

- https://arxiv.org/pdf/2010.16171.pdf
- https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
- https://github.com/ultraembedded/biriscv