ECE 6463

Project Milestone 2

Design of a NYU-6463 RV32I Processor

Swarnashri Chandrashekhar - sc8781
Shrey Malhotra - sm10141
M Athish Subramanian - ms13606

Agenda

Problem Statement

We are asked to design a RV32I processor which implements arithmetic functions and also performs tasks such as instruction fetch, instruction decode, execution, memory access and write back in a multi-cycle manner.

The NYU-6463-RV32I processor is a 32-bit architecture which executes a subset of the open source RISC-V RV32I instruction set.

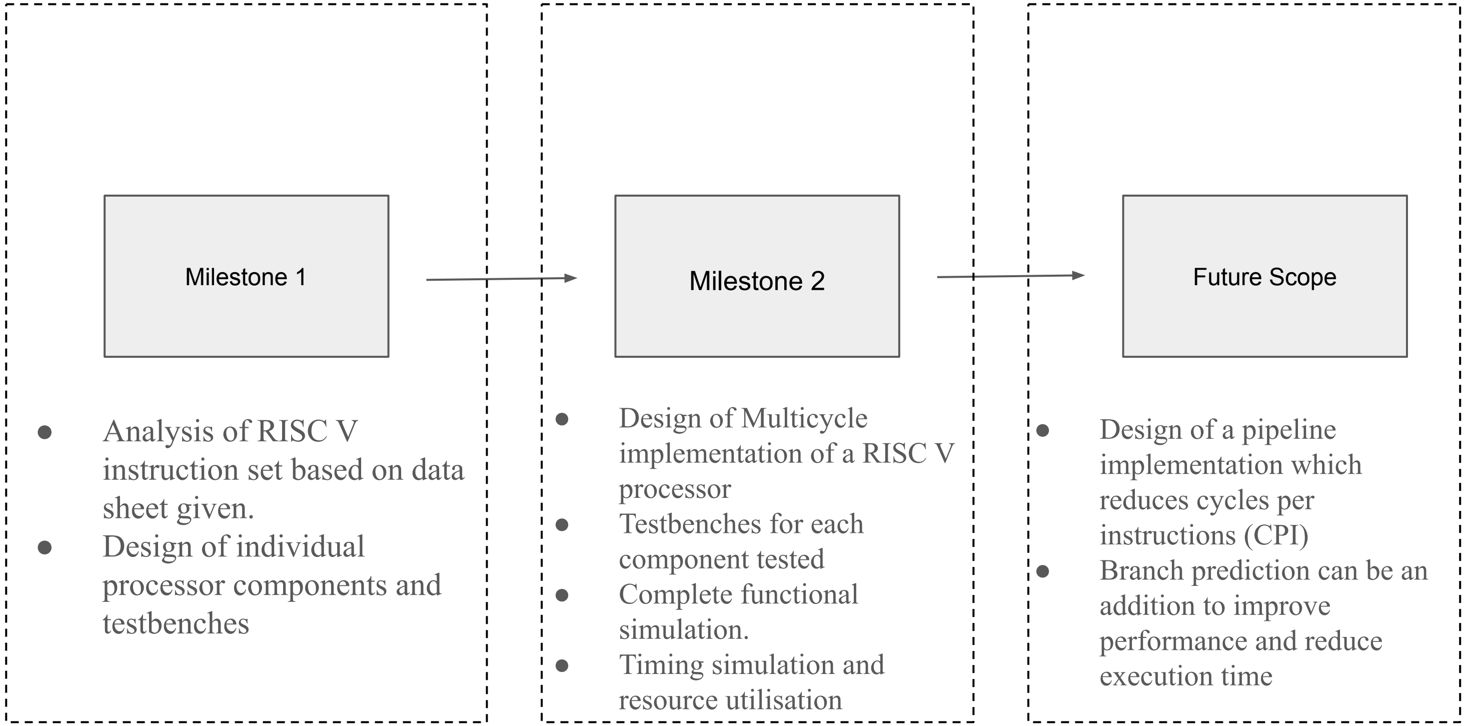
There are three main instruction types:

- Computational operations (from register file to register file)
- Load/store between memory and register file
- Control flow (jumps and branches to different parts opcode)

Project Schedule

S.No	Month	Work
1	October	Preparation of the project components and design / analysis of schematics
2	November	Milestone 1 - Design and implementation of individual components with corresponding testbenches
3	December	Milestone 2 - Wiring of all components and high level synthesis of the complete processor

Project Flow



- Analysis of RISC V instruction set based on data sheet given.
- Design of individual processor components and testbenches

- Design of Multicycle implementation of a RISC V processor
- Testbenches for each component tested
- Complete functional simulation.
- Timing simulation and resource utilisation

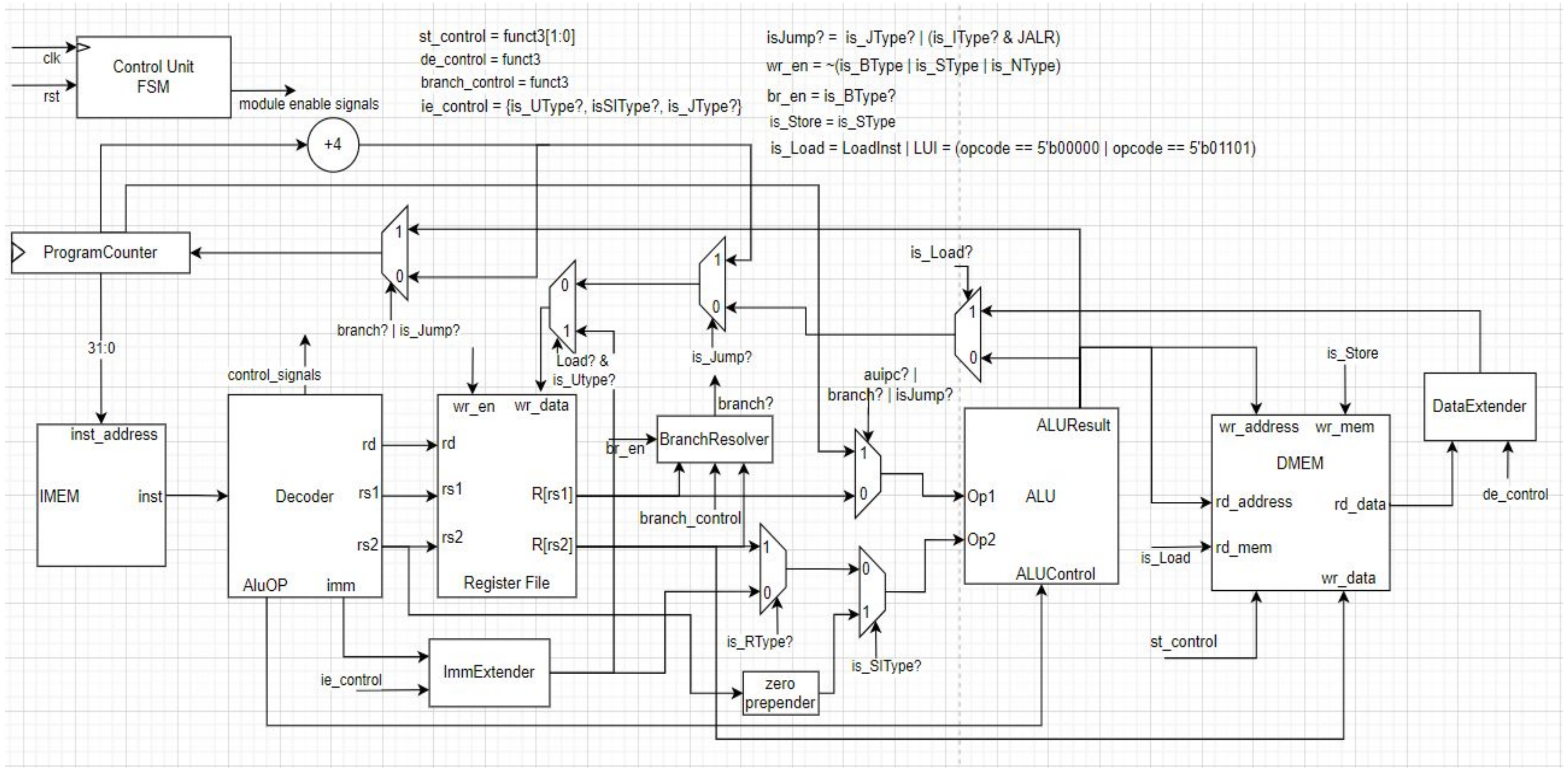
- Design of a pipeline implementation which reduces cycles per instructions (CPI)
- Branch prediction can be an addition to improve performance and reduce execution time

Components Involved

For project milestone 2, the following modules have been wired and designed with their respective testbenches:

- ALU
- Branch Resolver
- Program Counter
- Register File
- Data Memory
- Instruction Memory
- Data Extender
- Immediate Extender
- Decoder
- Control Unit

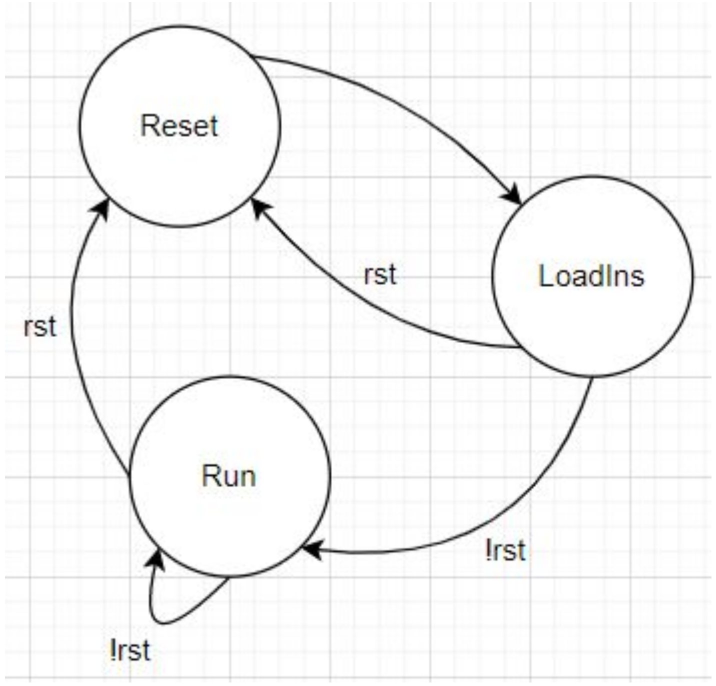
Design



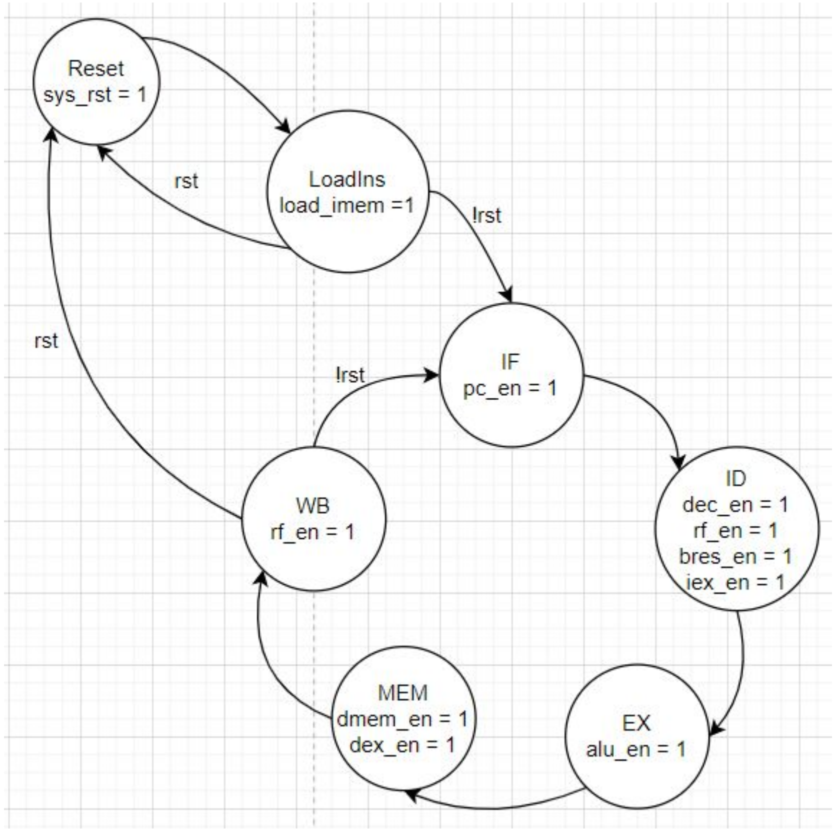
clk
rst
Control Unit FSM
module enable signals
st_control = funct3[1:0]
de_control = funct3
branch_control = funct3
ie_control = {is_UType?, isSIType?, is_JType?}
isJump? = is_JType? | (is_IType? & JALR)
wr_en = ~(is_BType | is_SType | is_NType)
br_en = is_BType?
is_Store = is_SType
is_Load = LoadInst | LUI = (opcode == 5'b00000 | opcode == 5'b01101)
+4
ProgramCounter
1
0
branch? | is_Jump?
31:0
control_signals
Load? & is_Utype?
is_Jump?
is_Load?
auipc? | branch? | isJump?
is_Store
inst_address
IMEM
inst
Decoder
rd
rs1
rs2
AluOP
imm
wr_en
wr_data
Register File
R[rs1]
R[rs2]
br_en
BranchResolver
branch?
branch_control
is_RType?
is_SIType?
zero prepender
ImmExtender
ie_control
ALUResult
ALU
Op1
Op2
ALUControl
wr_address
wr_mem
DMEM
rd_address
rd_mem
rd_data
wr_data
is_Load
st_control
DataExtender
de_control

Control Unit FSM

Single Cycle Mode



Multi Cycle Mode



Tests

Test Case 1: Sum of cubes of odd numbers in a given range

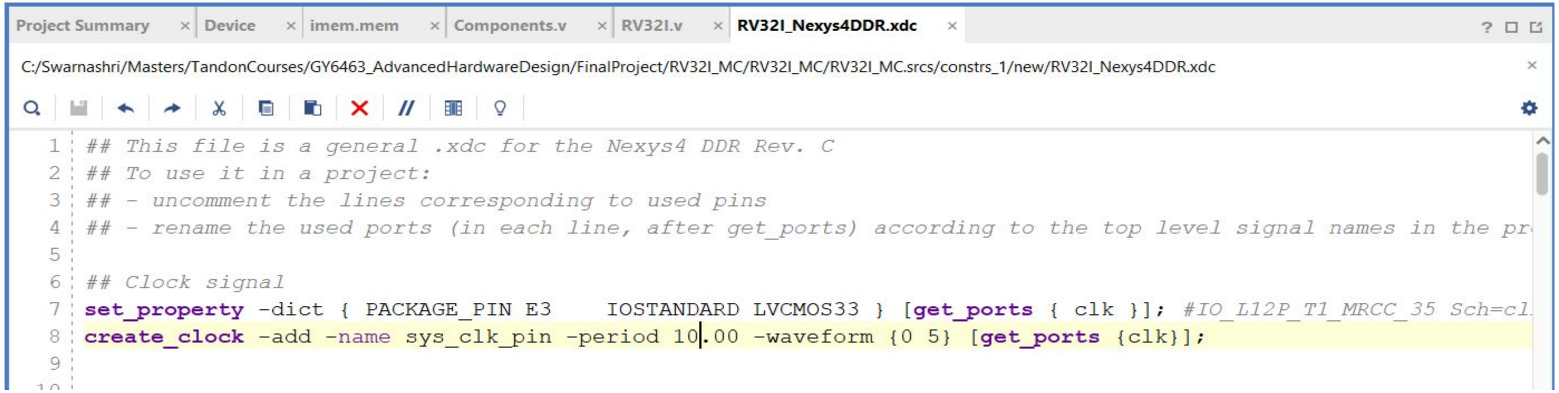
```
addi x5, x0, 0      # Limit 1
addi x6, x0, 4      # Limit 2
addi x7, x0, 2
add x30, x30, x0
add x28, x0, x5
beq x28, x0, if_zero
odd:
sub x28, x28, x7
bge x28, x7, odd
if_zero:
bne x28, x0, no_incr
addi x5, x5, 1
beq x0, x0, skip
no_incr:
addi x5, x5, 2
Skip:
add x28, x0, x6
odd2:
sub x28, x28, x7
bge x28, x7, odd2
bne x28, x0, no_incr2
addi x6, x6, 1
no_incr2:
bge x5, x6, exit
add x4, x0, x5
addi x11, x0, 1
addi x3, x0, 3
power:
bge x11, x3, skip_power
add x9, x0, x0
add x13, x0, x0
sum:
bge x9, x5, end
add x13, x13, x4
addi x9, x9, 1
beq x0, x0, sum
end:
add x4, x0, x13
addi x11, x11, 1
beq x0, x0, power
skip_power:
add x30, x30, x4
addi x5, x5, 2
beq x0, x0, no_incr2
exit: beq x0, x0, exit
```

Tests (Cont.)

Test Case 2: Sum of cube of prime numbers less than 4

```
addi x2, x0, 4
addi x3, x0, 1
add x29, x0, x0
addi x13, x0, 3
loop1:
addi x3, x3, 1
addi x4, x0, 2
bge x3, x2, exit
loop2:
beq x4, x3, if_prime
add x7, x0, x3
repeat:
sub x7, x7, x4
bge x7, x4, repeat
beq x7, x0, loop1
addi  x4, x4, 1
beq x0, x0, loop2
if_prime:
add x10, x0, x3
addi x11, x0, 1
power:
bge x11, x13, pw_end
add x9, x0, x0
add x14, x0, x0
sum:
bge x9, x3, end
add x14, x14, x10
addi x9, x9, 1
beq x0, x0, sum
end:
add x10, x0, x14
addi x11, x11, 1
beq x0, x0, power
pw_end:
add x29, x29, x10
beq x0, x0, loop1
exit:
beq x0, x0, exit
```

Timing Simulation



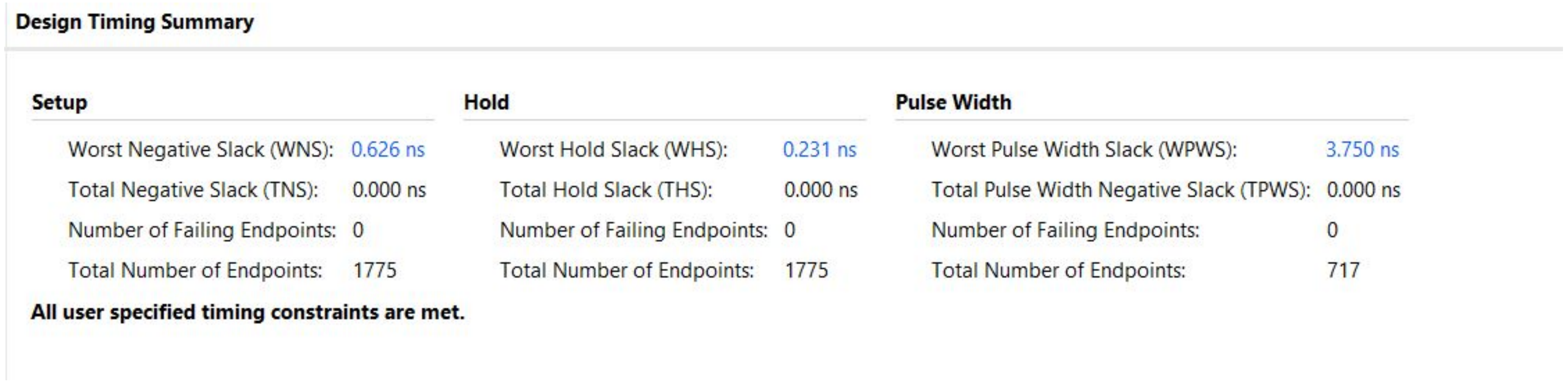
Project Summary × | Device × | imem.mem × | Components.v × | RV32I.v × | RV32I_Nexys4DDR.xdc ×

C:/Swarnashri/Masters/TandonCourses/GY6463_AdvancedHardwareDesign/FinalProject/RV32I_MC/RV32I_MC/RV32I_MC.srcs/constrs_1/new/RV32I_Nexys4DDR.xdc

```
## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the pr

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=cl
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

```



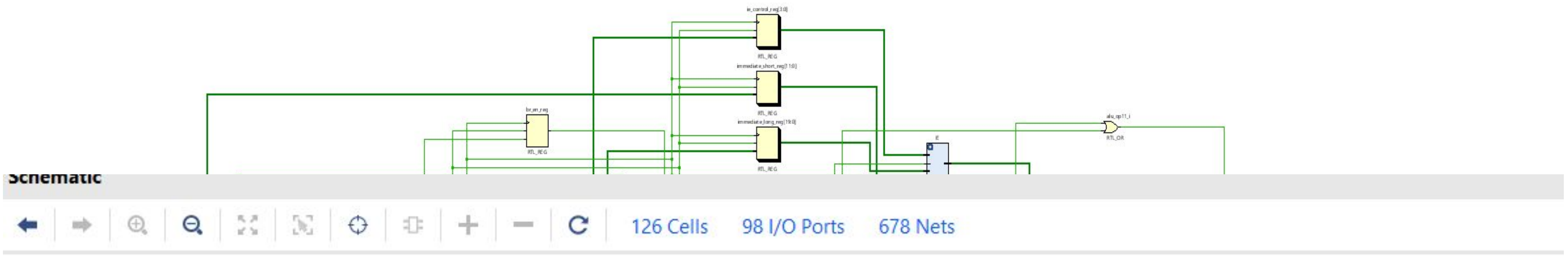
Design Timing Summary

```
Setup                                  Hold                                      Pulse Width

Worst Negative Slack (WNS):  0.626 ns  Worst Hold Slack (WHS):        0.231 ns  Worst Pulse Width Slack (WPWS):          3.750 ns
Total Negative Slack (TNS):  0.000 ns  Total Hold Slack (THS):        0.000 ns  Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0         Number of Failing Endpoints:   0         Number of Failing Endpoints:             0
Total Number of Endpoints:   1775      Total Number of Endpoints:     1775      Total Number of Endpoints:               717
```

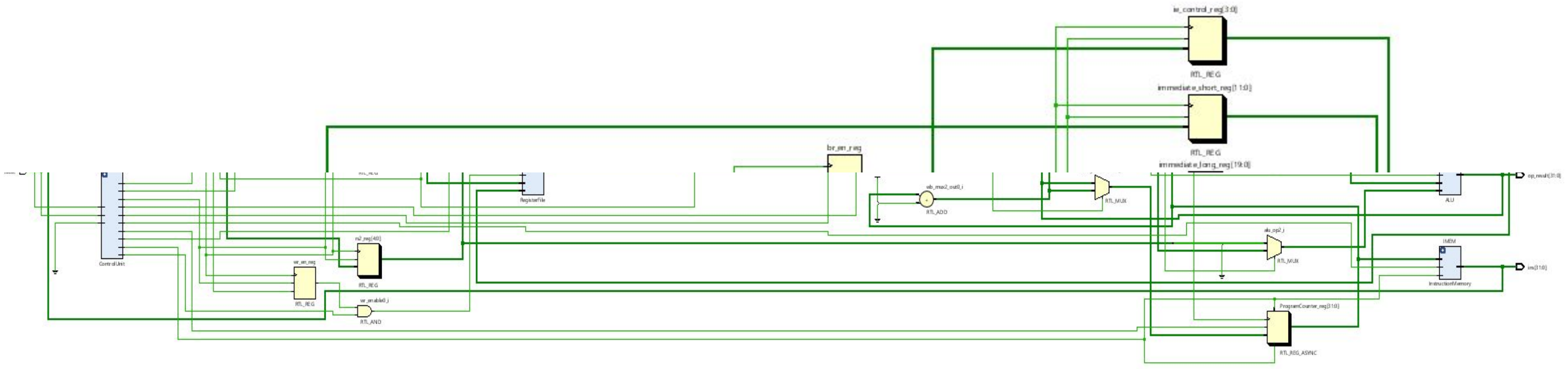
All user specified timing constraints are met.

Resource Utilisation



Schematic

126 Cells 98 I/O Ports 678 Nets



Future Scope

- Design
  - Pipelined architecture - By designing a pipeline instruction implementation, we can improve performance and reduce cycles per instructions.
  - Branch prediction can be implemented to mitigate costs of branching and reduce CPUs initially executed instructions one by one as they came in.
  - Out of Order Execution - Tomasulo’s machine with Reorder Buffer and Load Store Queue.
  - Multiple Functional Units.

- Test Bench
  - Automate high level test bench to verify the architectural state of the machine at every cycle based on the test cases in a file.