TTA-based Co-design Environment (TCE) tools

Akshay Godse
MT2019504

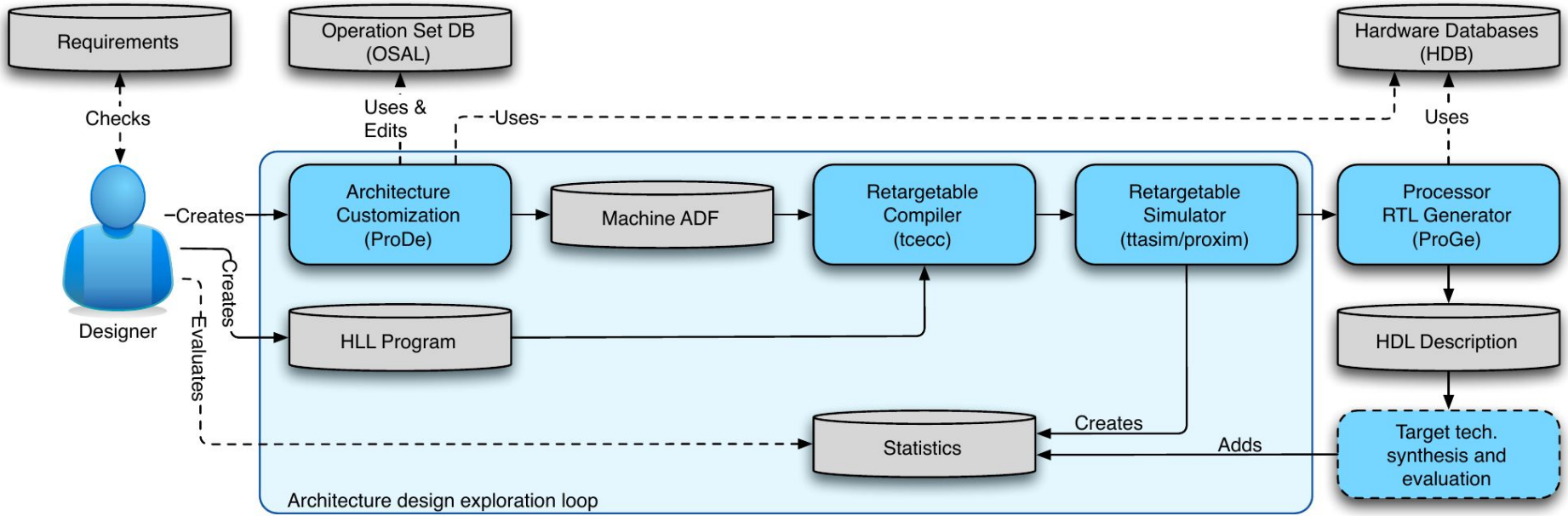
TCE

TCE development is led by the Customized Parallel Computing (CPC) group at the Tampere University, Finland.

source : http://openasip.org/

- TCE is an open application-specific instruction-set toolset
- It can be used to design and program customized processors based on the energy efficient TTA
- Complete retargetable co-design flow from high-level language programs down to synthesizable processor RTL (Verilog and VHDL)
- Processor customization points include the register files, function units, supported operations, and the interconnection network

TCE


Requirements
Operation Set DB (OSAL)
Hardware Databases (HDB)
Checks
Uses & Edits
Uses
Uses
Designer
Creates
Creates
Evaluates
Architecture Customization (ProDe)
Machine ADF
Retargetable Compiler (tcecc)
Retargetable Simulator (ttasim/proxim)
Processor RTL Generator (ProGe)
HLL Program
HDL Description
Statistics
Creates
Adds
Target tech. synthesis and evaluation
Architecture design exploration loop

Example:

a) **Figure 1** shows the acyclic **Control Data Flow Graph** of a differential equation integrator. Its behavioral description is given below. Assume a word length of 8 bits in the data path. Consider the resource constraint to be as follows : **Two Multiplier** and **One ALU** (this performs addition, subtraction and comparison). In the comparator mode the ALU implements the following behaviour : **if ($a_1 < a_2$) then z = 1 else z = 0**. As can be seen from the **CDFG** some of the computed values are re-used in the next iteration. Synthesize the fastest implementation of the above system. Show all the steps for your synthesis – scheduling, functional unit allocation, storage unit binding and interconnection binding for a point to point interconnection topology after clearly stating all the data transfers needed in your implementation.

while (x < a) do

$x_1 = x + dx;$

$u_1 = u + 3xudx - 3ydx;$

$y_1 = y + udx;$

$x = x_1;$

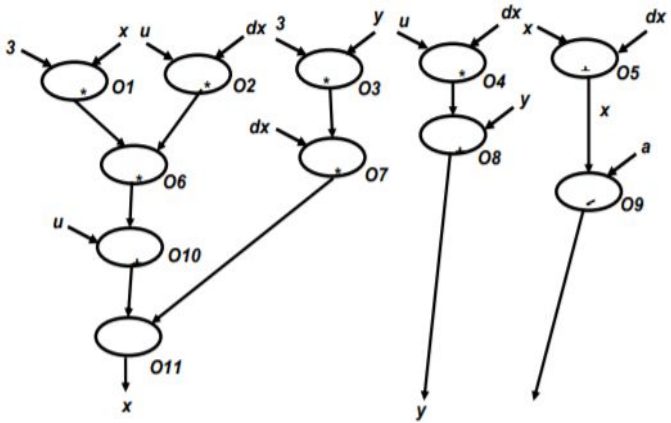
$y = y_1;$

$u = u_1;$

endwhile

[In **Figure 1** the different operations and the corresponding vertices associated with them in the CDFG are listed below :

Multiplier (*) : O1, O2, O3, O4, O6 and O7
Adder (+) : O5, O8 and O10
Comparator (<) : O9
Subtractor (-) : O11 **[30]**



Processor Architecture

- The TCE tool has some adf (architecture definition file) already provided in it which just enough resources that the TCE compiler can still compile C programs for it.

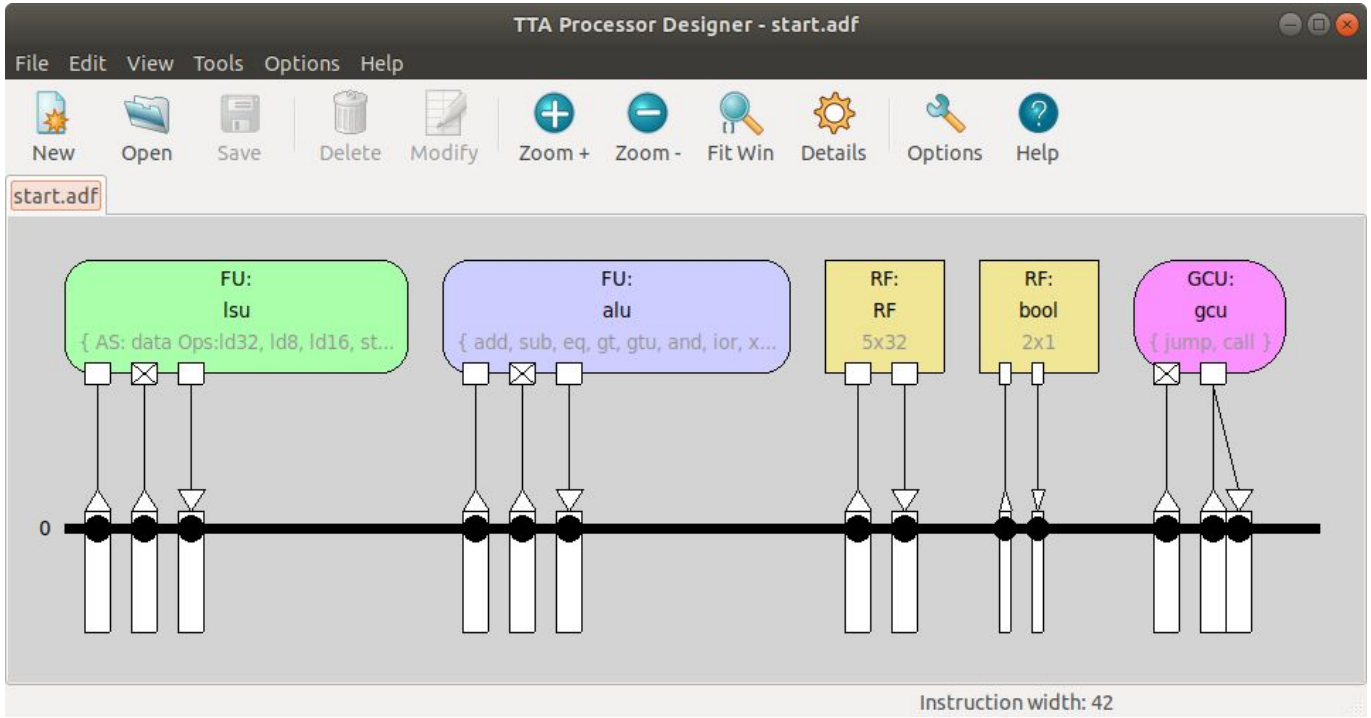
```
akshay@akshay-VivoBook-ASUSLaptop-X512FL-X512FL:~/tce-devel/tce/data/mach$ ls
ADF_Schema.xsd               minimal.adf     minimal_with_stdout.adf
minimal_64b_with_stdout.adf  minimal_be.adf
```

- You can view the architecture using the graphical Processor Designer using following command (minimal.adf renamed to start.adf)

  prode start.adf &

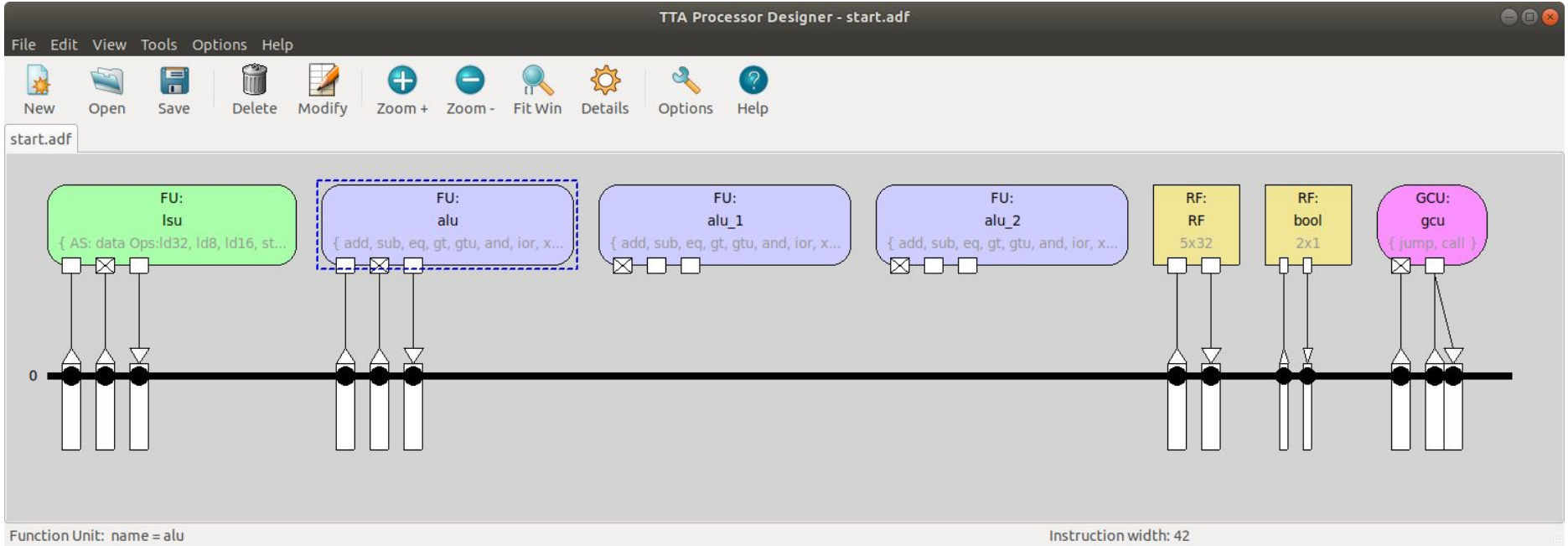
Processor Architecture

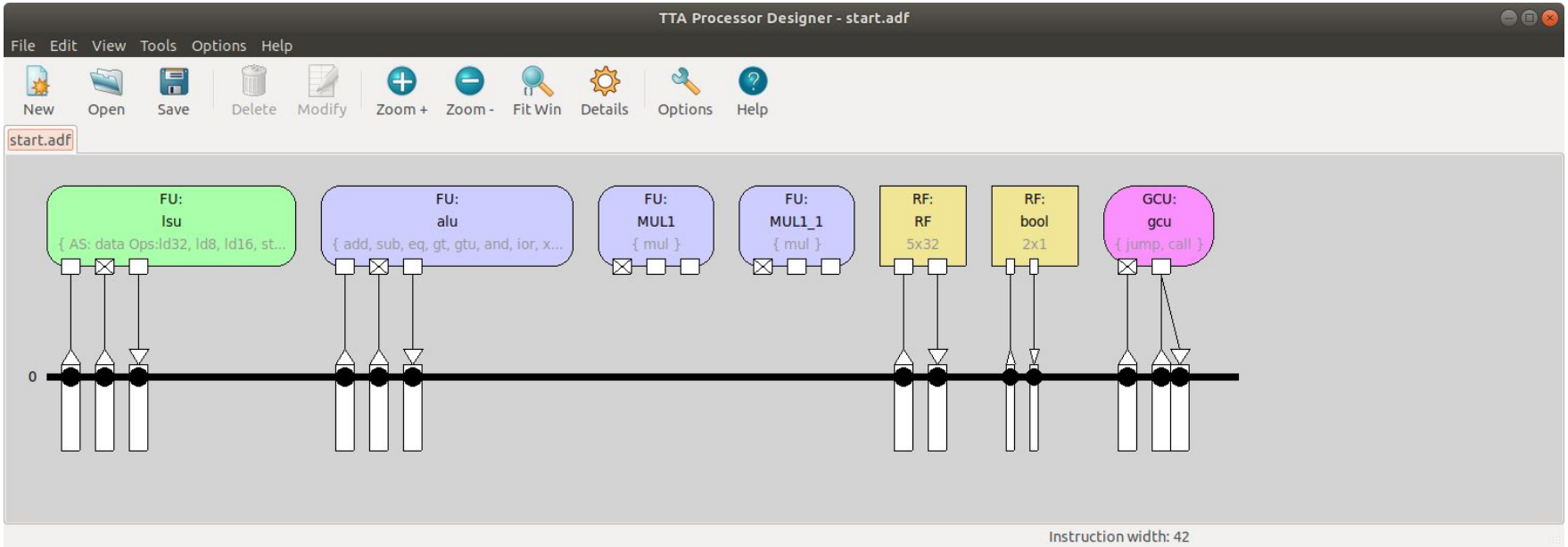
There is 1 bus and 5 units: global control unit (GCU), 2 register files, 1 arithmetic-logic unit (ALU), and 1 load-store unit (LSU).



Modifying Architecture according to Question

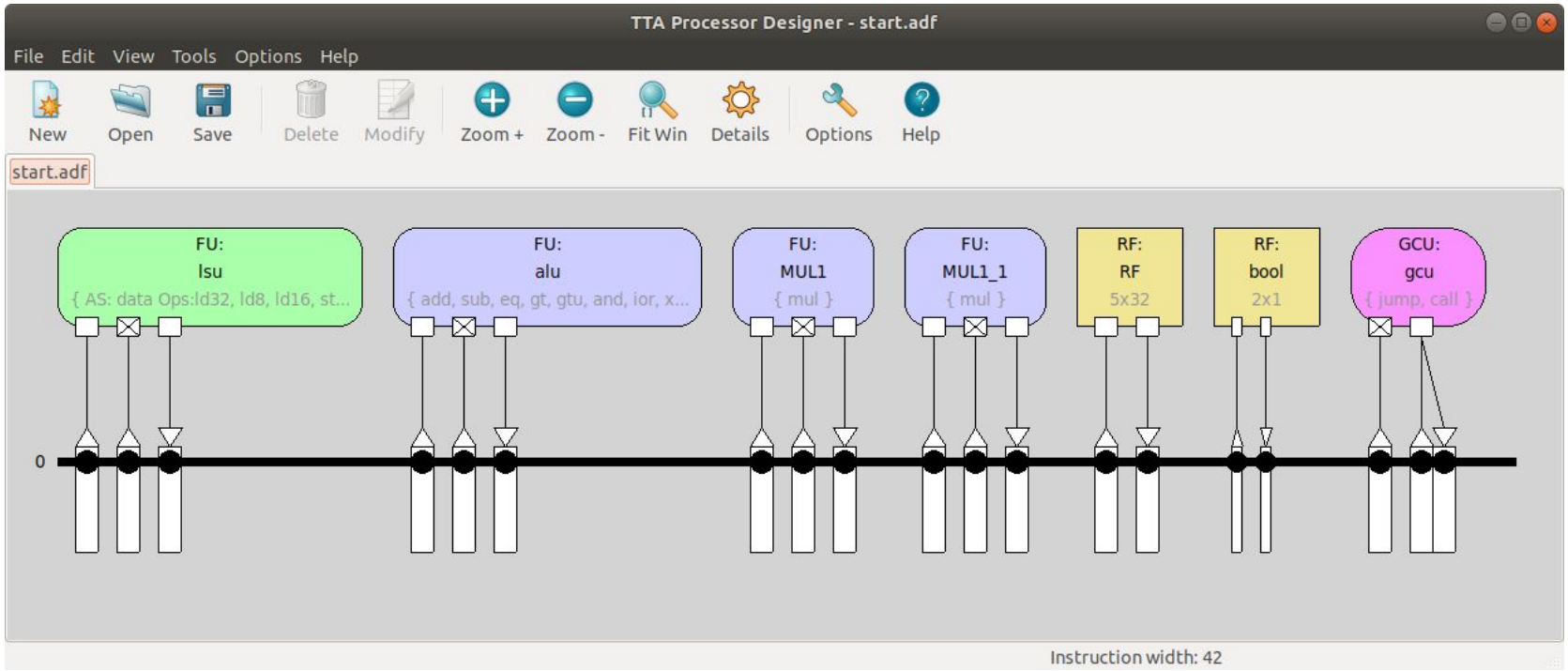
1. We are provided with 2 Multipliers and 1 ALU. So we can modify adf file according to our requirement.
2. We first copied the ALU twice pasted it. To copy select ALU then Ctrl+C and pasted it twice with Ctrl+V
3. Now ALU should perform only addition,subtraction and comparison operation only. So we kept only those operations
4. In next two ALU we kept only MUL operation.
5. So we got the Architecture according to our own requirements and we can save it to compile the C code later on.


TTA Processor Designer - start.adf
File Edit View Tools Options Help
New Open Save Delete Modify Zoom + Zoom - Fit Win Details Options Help
start.adf
FU:
lsu
{ AS: data Ops:ld32, ld8, ld16, st...
FU:
alu
{ add, sub, eq, gt, gtu, and, ior, x...
FU:
alu_1
{ add, sub, eq, gt, gtu, and, ior, x...
FU:
alu_2
{ add, sub, eq, gt, gtu, and, ior, x...
RF:
RF
5x32
RF:
bool
2x1
GCU:
gcu
{ jump, call }
0
Function Unit: name = alu
Instruction width: 42

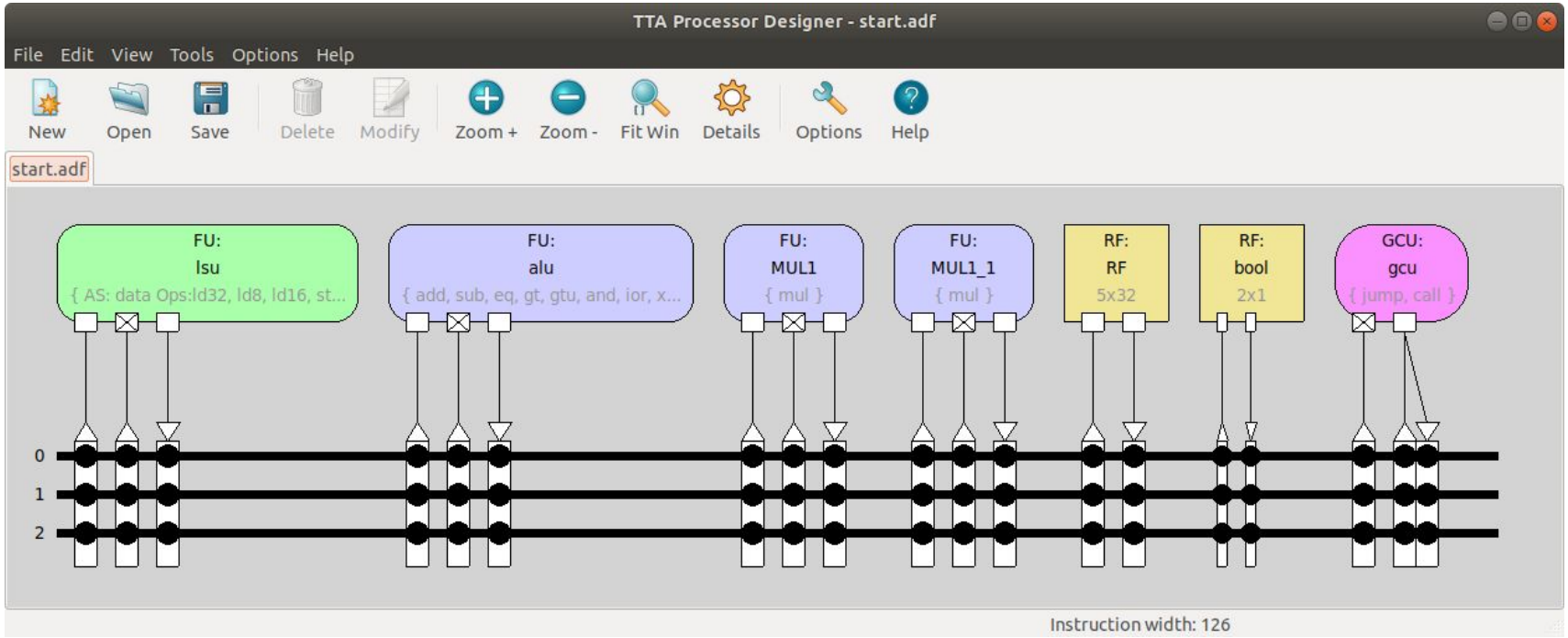

TTA Processor Designer - start.adf
File Edit View Tools Options Help
New
Open
Save
Delete
Modify
Zoom +
Zoom -
Fit Win
Details
Options
Help
start.adf
FU:
lsu
{ AS: data Ops:ld32, ld8, ld16, st...
FU:
alu
{ add, sub, eq, gt, gtu, and, ior, x...
FU:
MUL1
{ mul }
FU:
MUL1_1
{ mul }
RF:
RF
5x32
RF:
bool
2x1
GCU:
gcu
{ jump, call }
0
Instruction width: 42

To connect the FU to bus under Tools-->Fully Connect IC

Save this adf file for compilation.



Now if there is only one bus then there is no point of adding multiple FU's hence we will add 2 more bus so that all 3 FU's can be used simultaneously.
Click on bus Ctrl+C to copy and paste it twice with Ctrl+V and then Tools->Fully Connect IC



Compiling and simulating

We must compile the source code for this architecture with command:

tcecc -O3 -a assignment.adf -o assignment.tpef -k x main.c

- assignment.adf is the adf file we created
- main.c is the C code
- assignment.tpef is the output file which can be simulated later on
- -k will keep test symbol to verify results in simulation. It should be a global variable in c code. Here we have kept x variable to verify output.

Now we can use graphical user interface version of the simulator called Proxim with command:

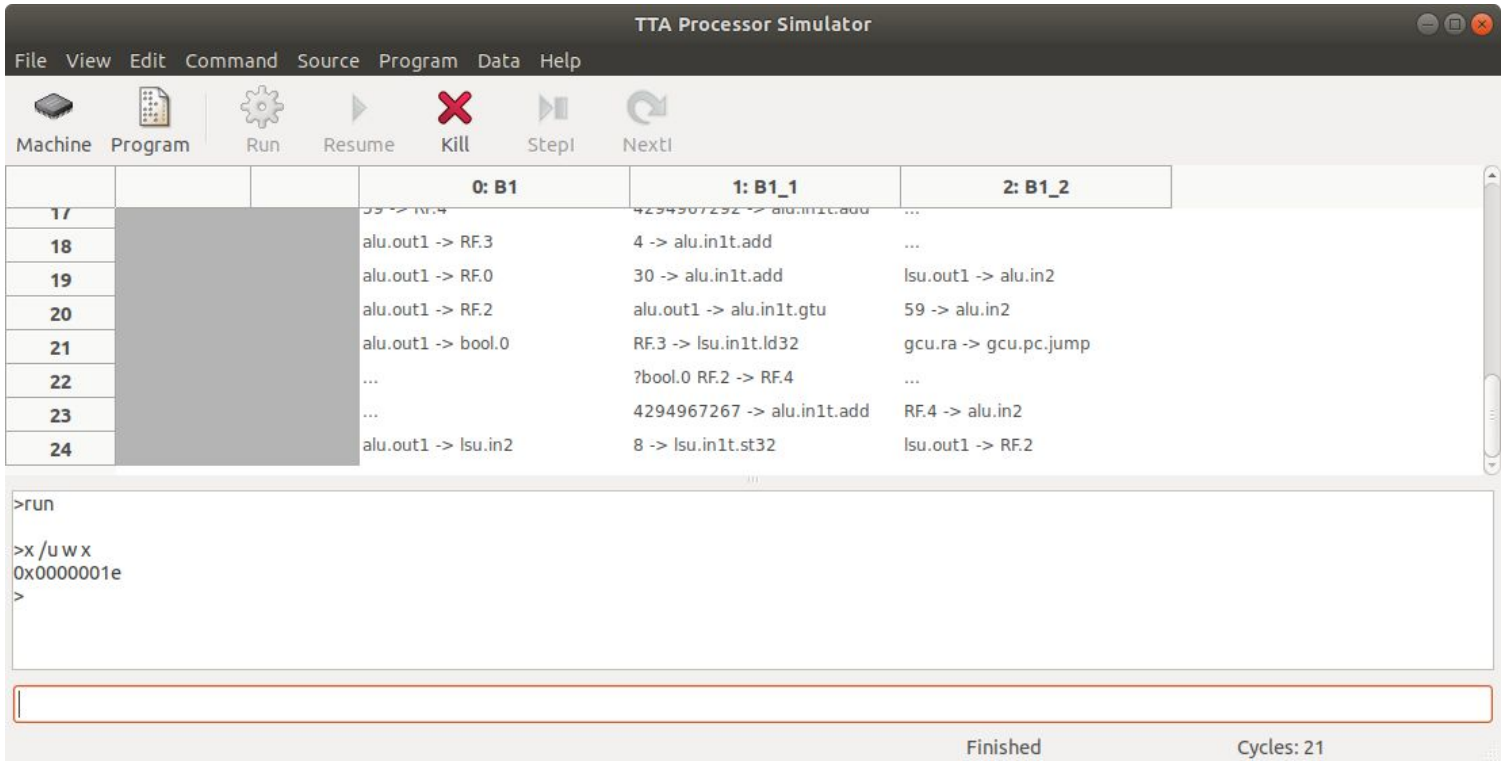
proxim assignment.adf assignment.tpef &

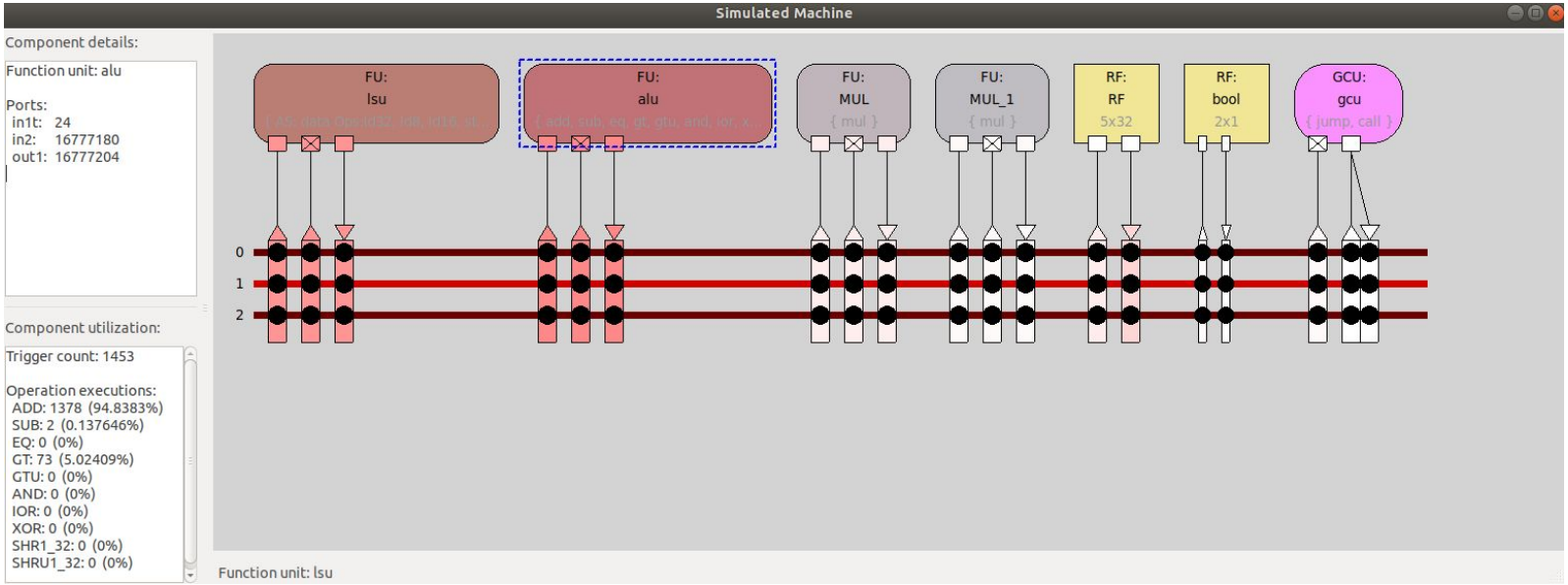
We can use command prompt of this simulator further:

eg.

x /u w result with this we can see result of simulation.

info proc stats with we can see resource utilizations.



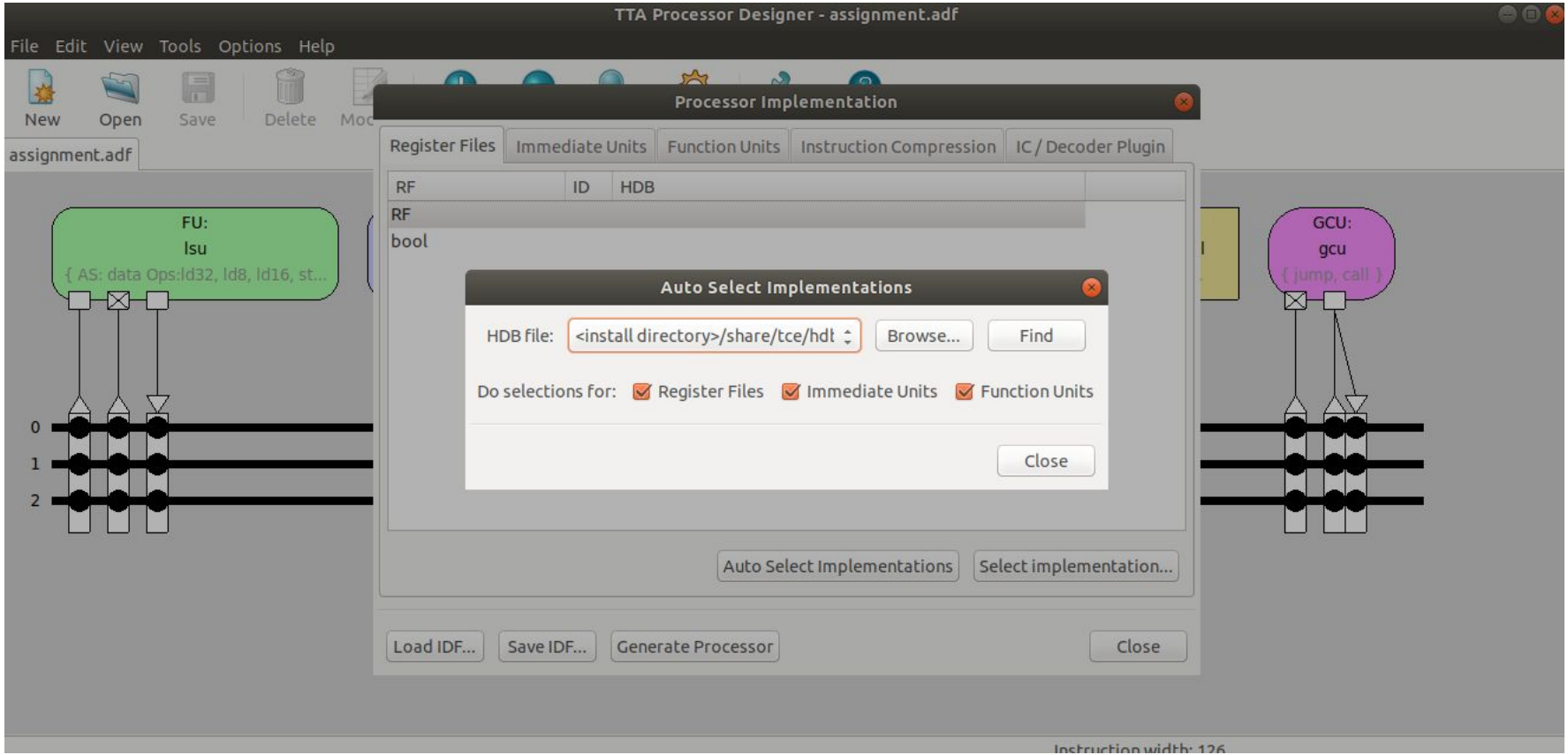

Simulated Machine
Component details:
Function unit: alu
Ports:
in1t: 24
in2: 16777180
out1: 16777204
Component utilization:
Trigger count: 1453
Operation executions:
ADD: 1378 (94.8383%)
SUB: 2 (0.137646%)
EQ: 0 (0%)
GT: 73 (5.02409%)
GTU: 0 (0%)
AND: 0 (0%)
IOR: 0 (0%)
XOR: 0 (0%)
SHR1_32: 0 (0%)
SHRU1_32: 0 (0%)
FU:
lsu
FU:
alu
{ add, sub, eq, gt, gtu, and, ior, x...
FU:
MUL
{ mul }
FU:
MUL_1
{ mul }
RF:
RF
5x32
RF:
bool
2x1
GCU:
gcu
{ jump, call }
0
1
2
Function unit: lsu

Generating the RTL and memory images

In this step we generate the RTL implementation of the processor, and the bit image of the parallel program.

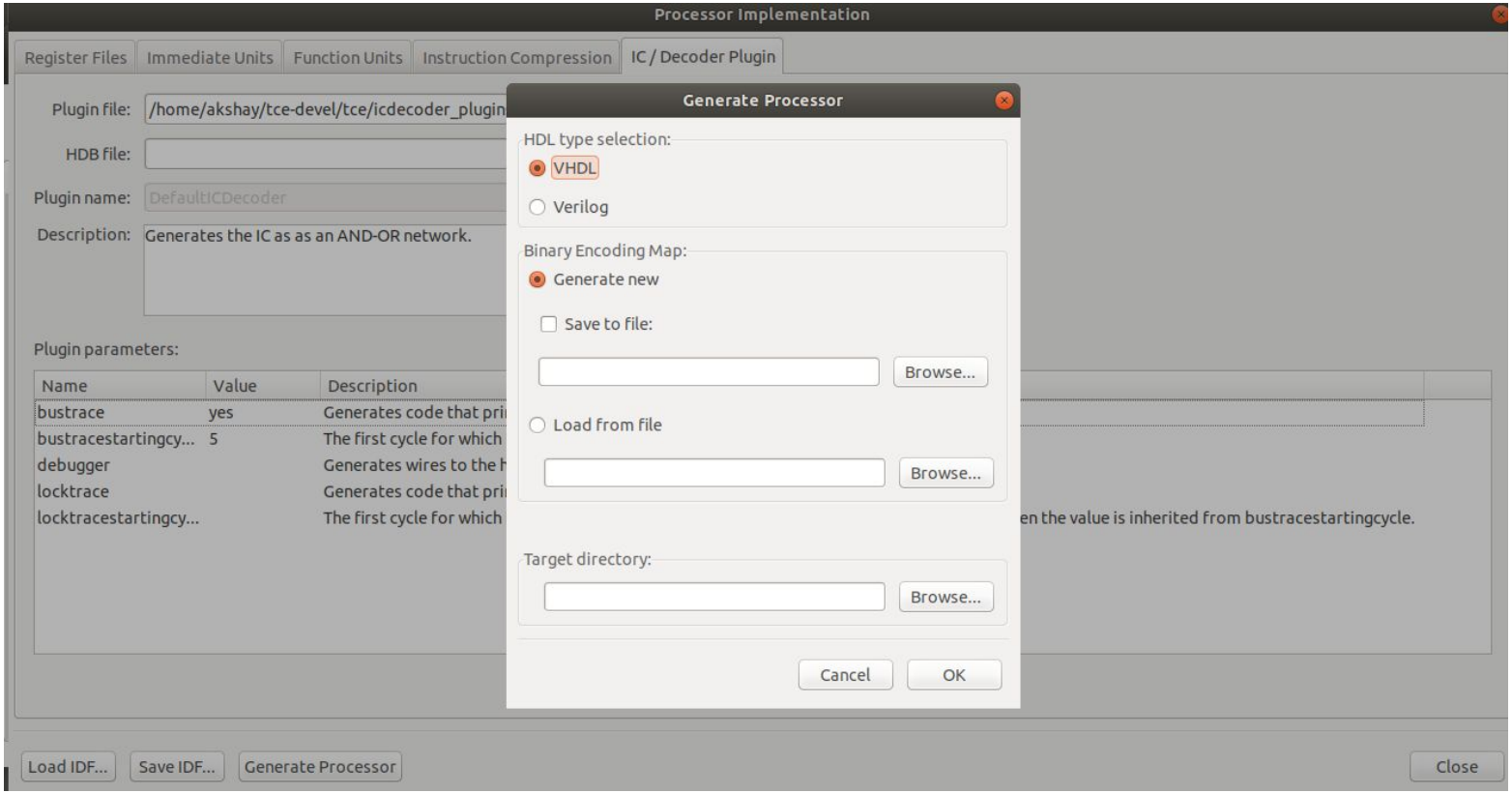
- Here we'll select implementations for the FUs which can be done in Tools>Processor Implementation
- TCE needs some data of the FU implementation in order to be able to generate processors that include the FU. So here we select HDB files for FU's in processor. (HDB -Hardware Database). HDB file contains VHDL implementation of FU's and Port information.
- We can enable bus tracing from the Implementation-dialog's IC / Decoder Plugin tab.The generated processor will now get a component which writes the bus value from every cycle to a text file for verification purposes.
- Now we can save IDF file.

We can either select HDB manually or auto select

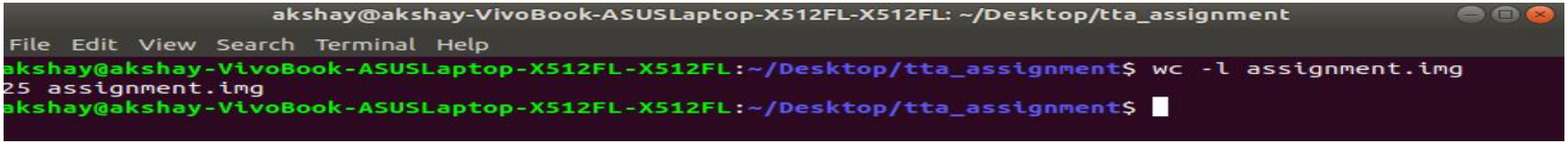


Processor implementation can be done in Verilog or VHDL

And here we can save memory bit image to create instruction memory and data memory images



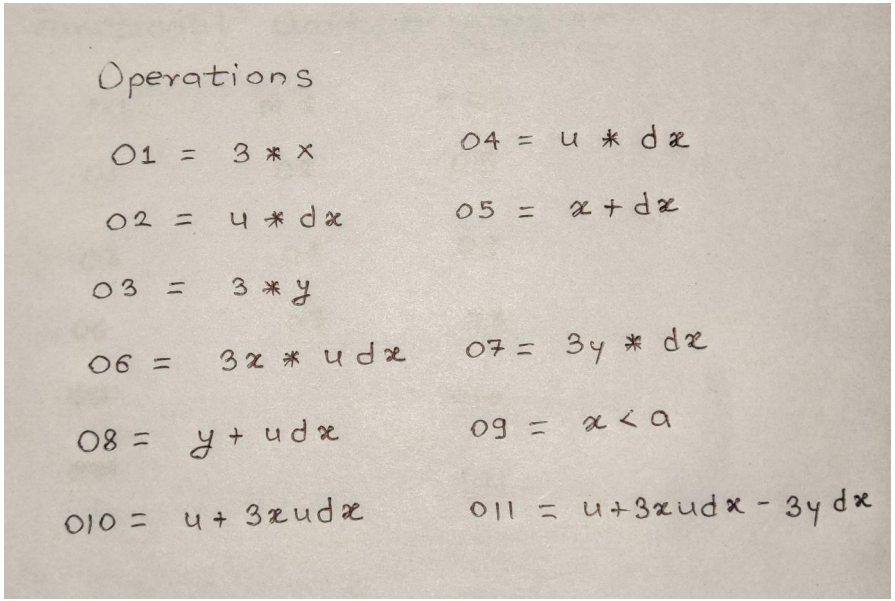
Now the file ’crc_with_custom_op.img’ includes the instruction memory image in “ascii 0/1” format.
Each line in that file represents a single instruction. Thus we can get the count of instructions by counting the lines in that file.

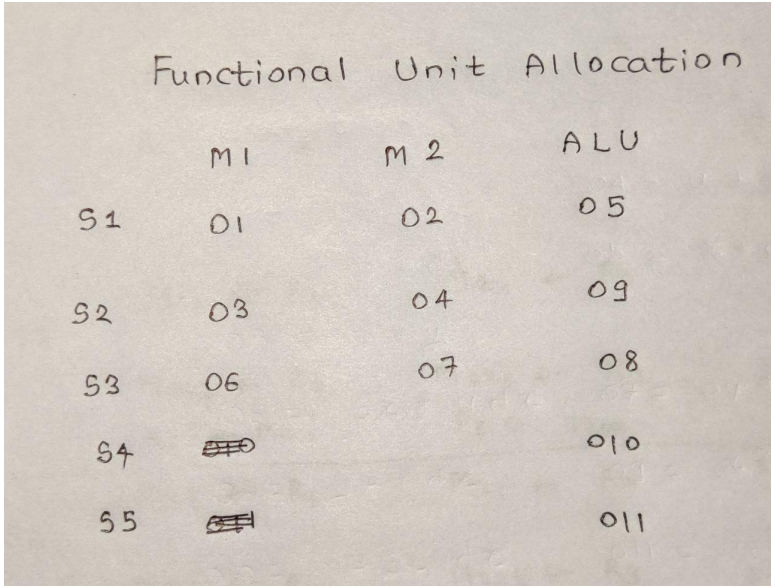


Manual Simulation of the Problem

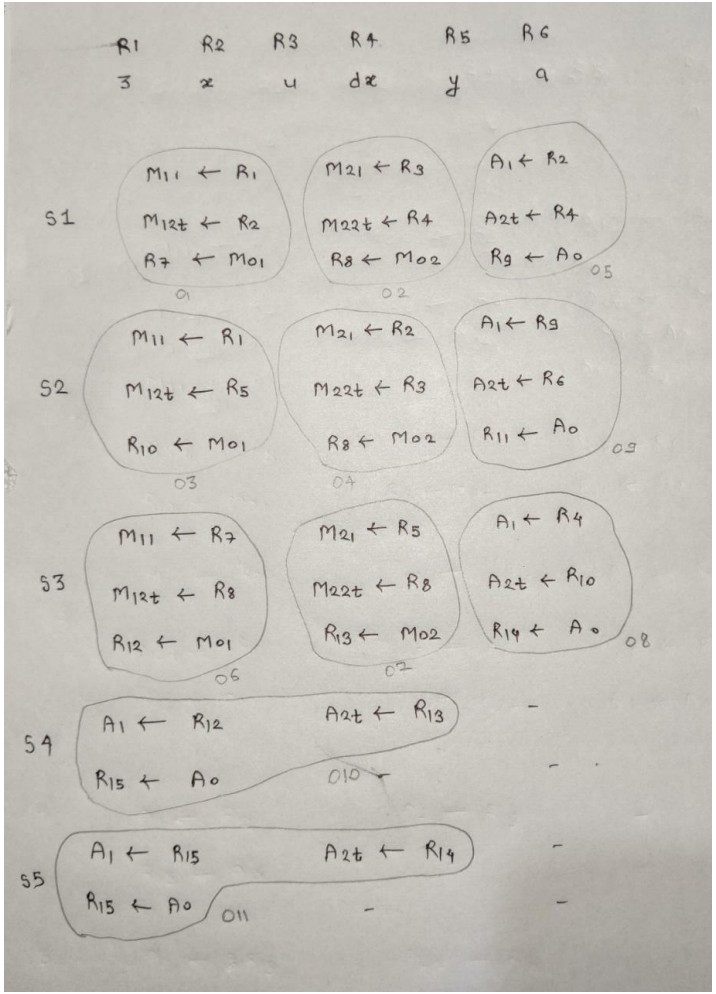
- As there are three buses available we should be able to use three move instructions at the same time and thus we can also use three FU’s at the same time.
- To reduce the number of registers and make the number of instructions minimum we can use the output registers of FU’s as input operand to another FU so that no intermediate register is required to store the result.
- We are assuming the second operand of FU’s to be the triggering operand.

Operations

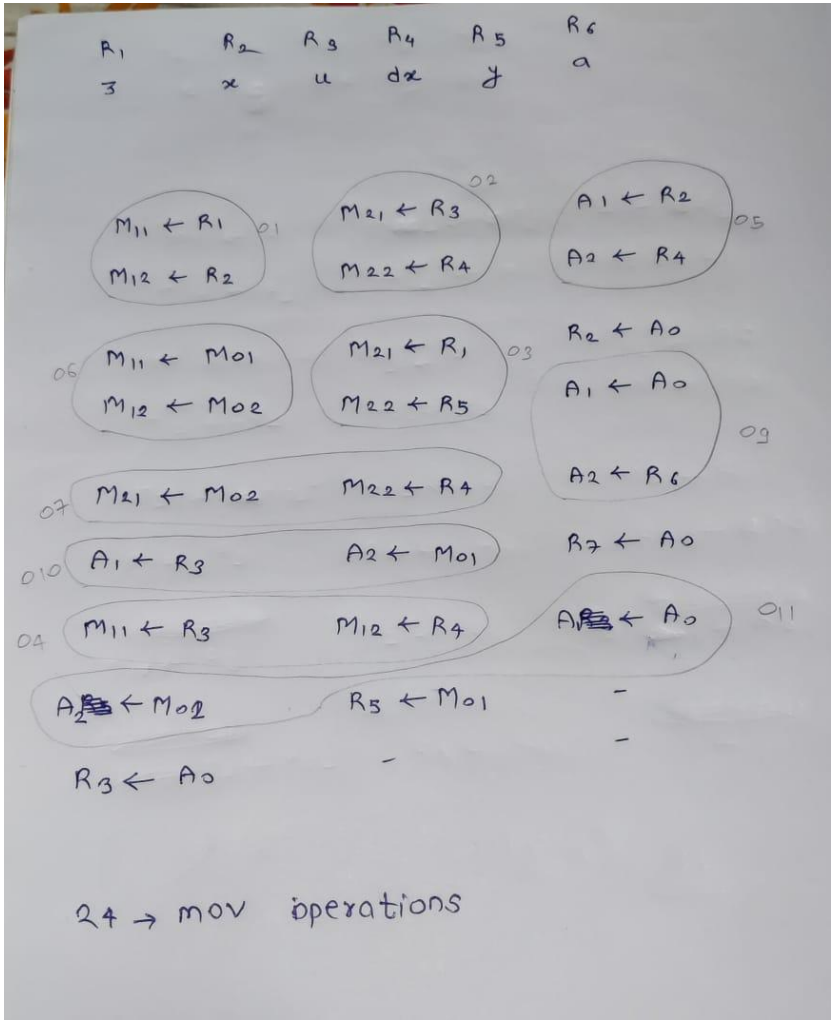

Operations
O1 = 3 * x
O2 = u * dx
O3 = 3 * y
O4 = u * dx
O5 = x + dx
O6 = 3x * u dx
O7 = 3y * dx
O8 = y + u dx
O9 = x < a
O10 = u + 3x u dx
O11 = u + 3x u dx - 3y dx


Functional Unit Allocation
M1
M2
ALU
S1
O1
O2
O5
S2
O3
O4
O9
S3
O6
O7
O8
S4
O10
S5
O11

- Implementation in TTA with MOV instruction.
- Further optimization can be done with using FU's o/p register as directly operand to next FU's i/p operand.
- This will help in reducing the number of MOV instructions and similarly register allocation will also be less.



As we can see the number of instructions is considerably less over here.



Issues encountered in C program

```
 2 #ifdef _DEBUG
 3 #include <stdio.h>
 4 #endif /* _DEBUG */
 5
 6
 7
 8 int x = 5;
 9
10 int main(void) {
11 int y = 1;
12 int u = 1;
13 int dx = 1;
14 int x1=0;int u1=0;int y1=0;
15 while(x<514){
16 x1 = x + dx;
17 u1 = u + 3*x*u*dx - 3*y*dx;
18 y1 = y + u*dx;
19 x = x1;
20 y = y1;
21 u = u1;
22 }
23
24     return 0;
25 }
```

- Don’t take variables like x,y,u,x1,dx,y1,u1 as local variables inside main function. Declare these variables as global then only the code will go inside while loop otherwise code will give the final answer in the Proxim simulator

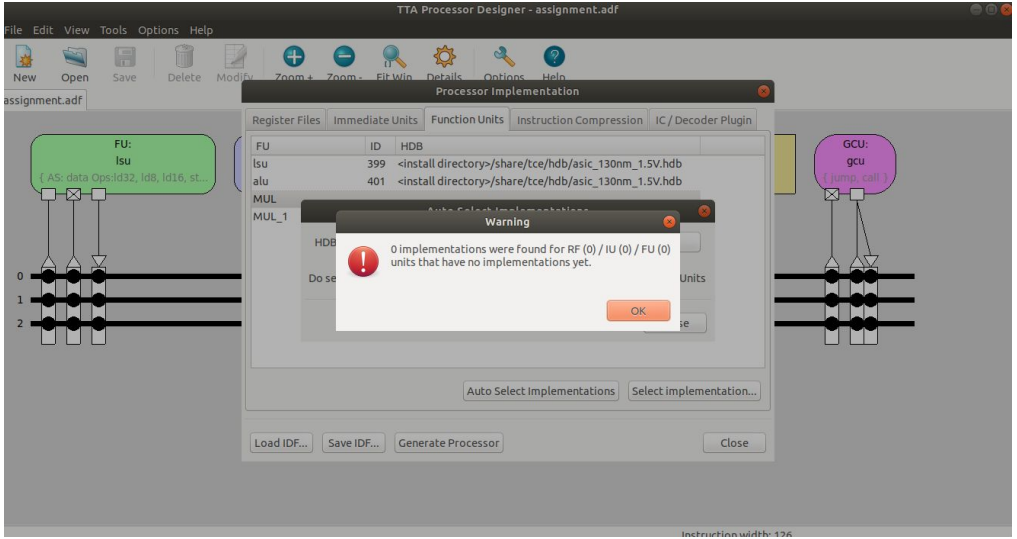
```c
int x = 11;
int y = 15;
int u = 21;
int dx = 7;
int x1=0;int u1=0;int y1=0;
//int t1_1,t1_2,t1,t2_1,t2;
int mul(int num1, int num2);
int main(void) {
while(x<514){
x1 = x + dx;
/*
t1_1 = mul(3,x);
t1_2 = mul(u,dx);
t1 = mul(t1_1,t1_2);
t2_1 = mul(3,y);
t2 = mul(t2_1,dx);
u1 = u + t1 - t2;
//y1 = y + t1_2;
*/
u1 = u + 3*x*u*dx - 3*y*dx;
y1 = y + u*dx;
x = x1;
y = y1;
u = u1;
}
    return 0;
}

/*int mul(int num1, int num2) {

   int result;
   _TCE_MUL(num1,num2,result);

   return result;
}*/
```

- We can force simulator to use a particular operation like shown in mul function just like shown in custom operation tutorial in manual.
- In the install directory /tce/opset/base.cc has all the the c code for operations in functional units mentioned in adf file.
- In the fig. below is the c code for MUL in the /tce/opset/base.cc file. This is similar to addition of custom operation

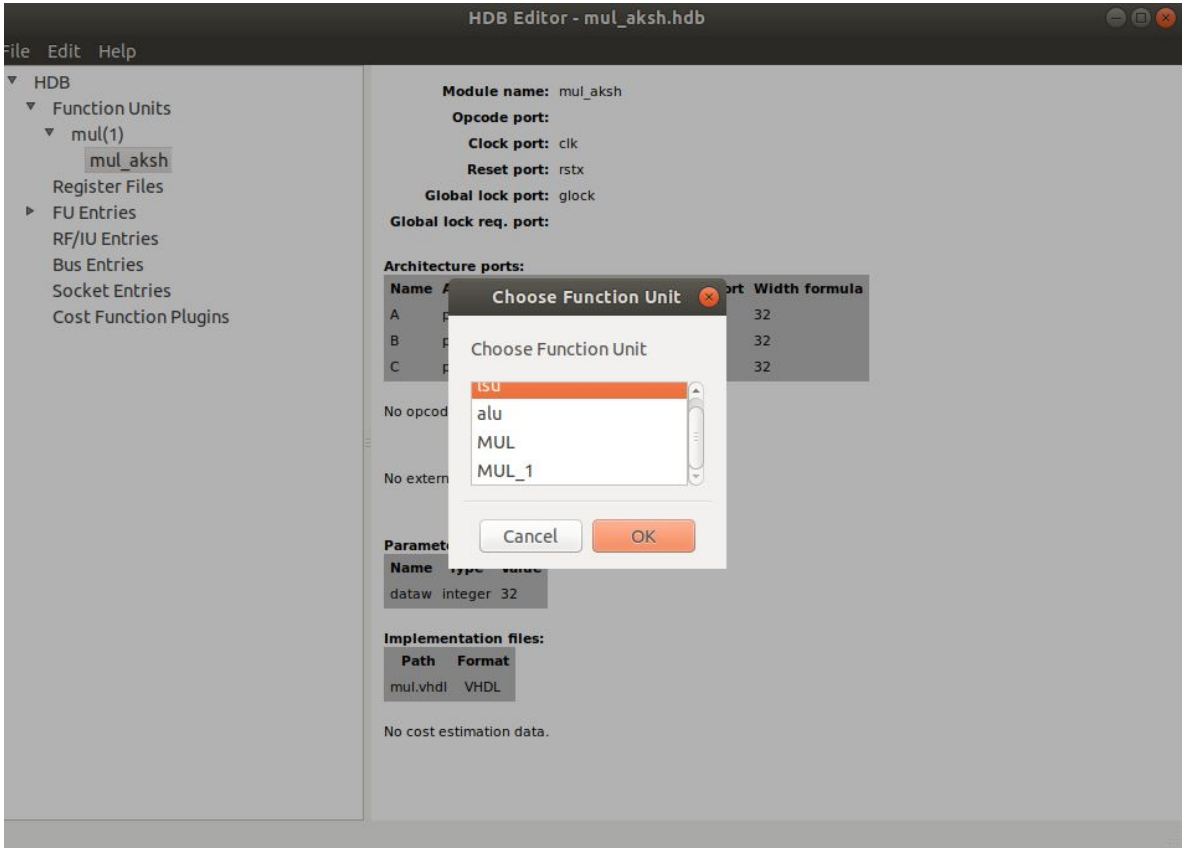
```
////////////////////////////////////////////////////////////////////////////////
// MUL - integer multiply
////////////////////////////////////////////////////////////////////////////////
OPERATION(MUL)

TRIGGER
    IO(3) = UINT(1)*UINT(2);
END_TRIGGER;

END_OPERATION(MUL)

////////////////////////////////////////////////////////////////////////////////
// DIV - integer divide
////////////////////////////////////////////////////////////////////////////////
OPERATION(DIV)

TRIGGER
    if (UINT(2) == 0)
        RUNTIME_ERROR("Divide by zero.")

    IO(3) = static_cast<SIntWord>(
        (static_cast<SIntWord>(UINT(1)) / static_cast<SIntWord>(UINT(2))));
END_TRIGGER;

END_OPERATION(DIV)
```

Problems in RTL synthesis.

- To generate the rtl synthesis we need the hdb files for each functional unit. For all operations in minimal adf hdb file is present but for MUL operation we have to create the hdb file just like shown in custom operation tutorial

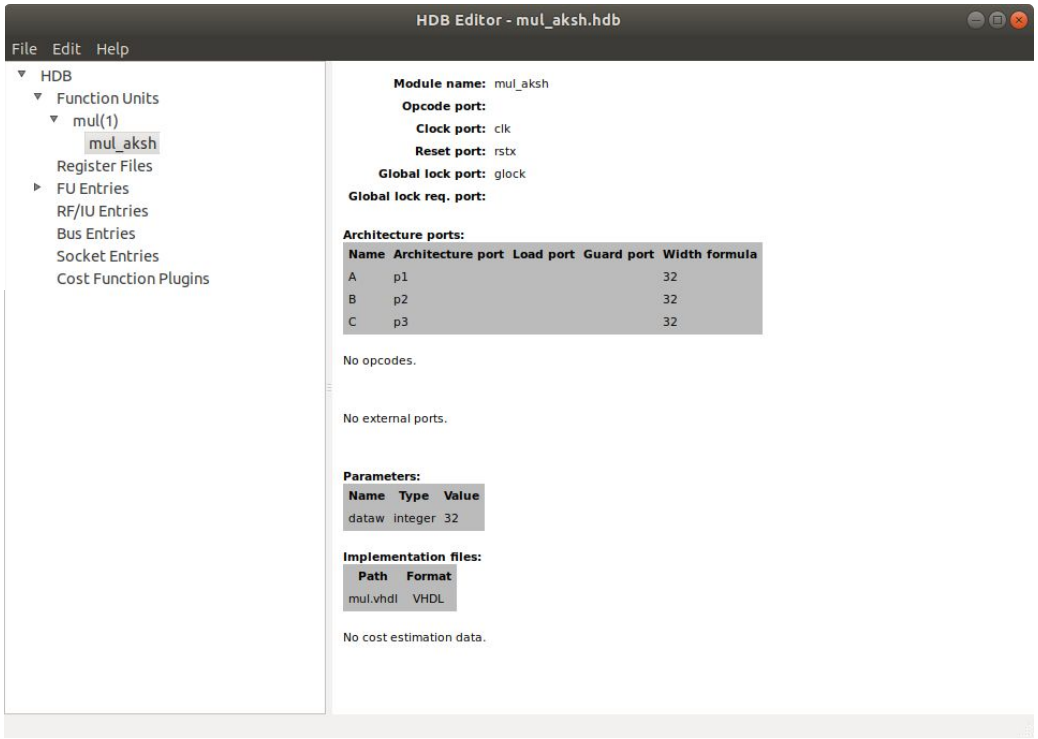


We created a hdb file and then selected FU from adf file

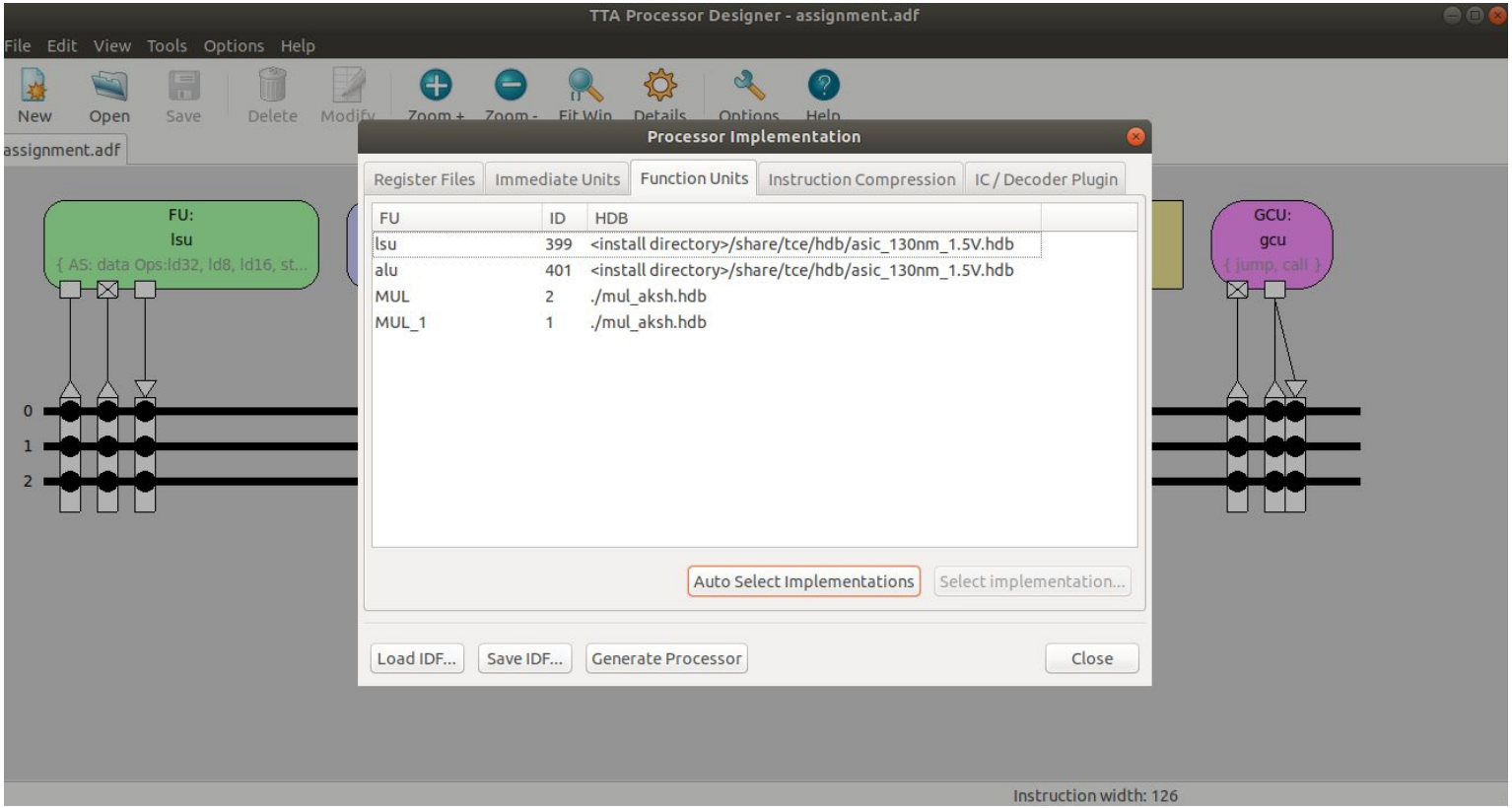


We found mul.vhdl already present in /tce/hdb/vhdl/fu in this directory.
In the hdb editor we created mul_aksh file and then we added implementation parameter as mentioned in the vhdl file and we can see it in the window as well

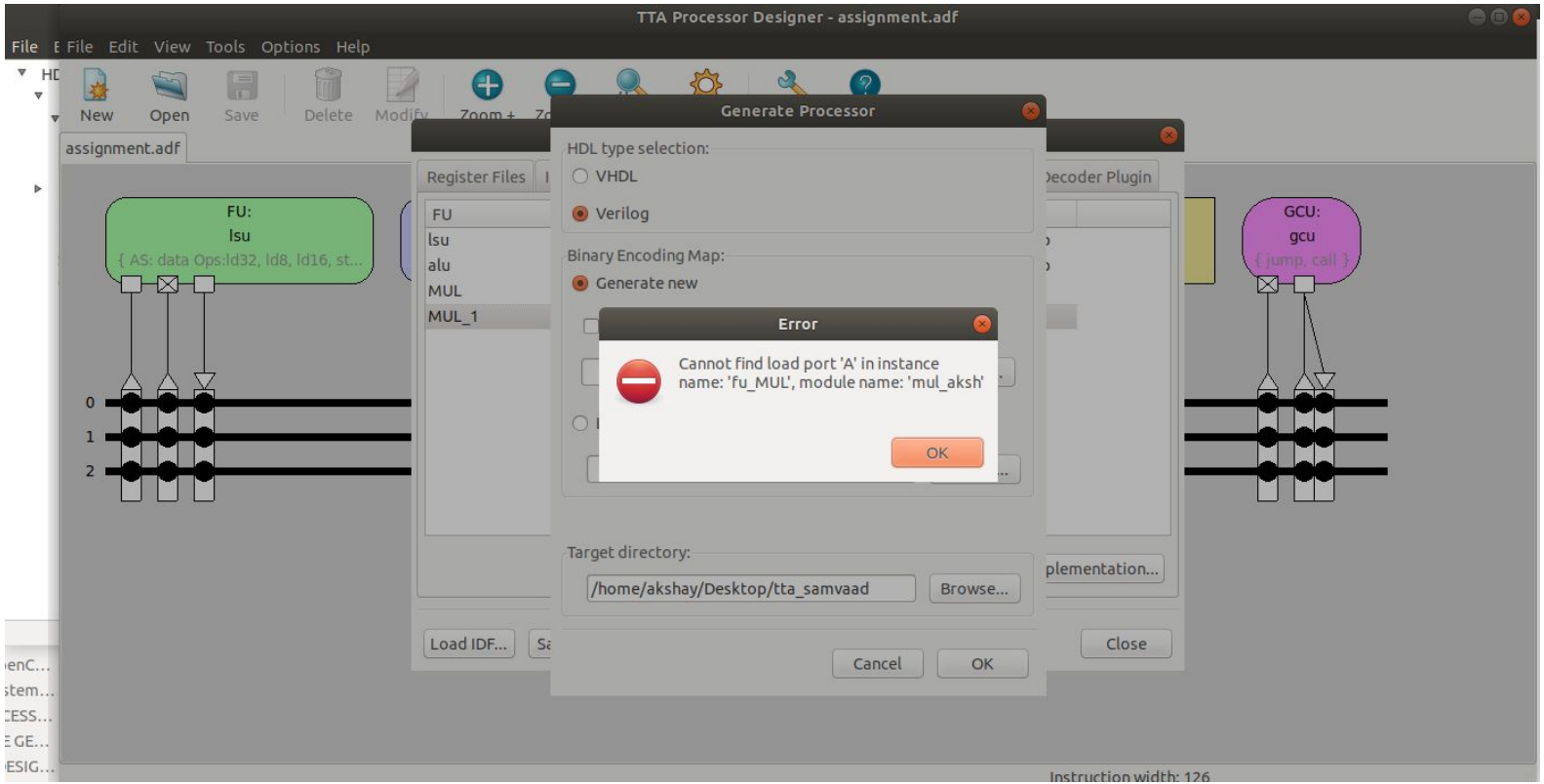
```
entity mul_arith is
  generic (
    dataw : integer := 32);
  port(
    A : in  std_logic_vector(dataw-1 downto 0);
    B : in  std_logic_vector(dataw-1 downto 0);
    P : out std_logic_vector(dataw-1 downto 0));
end mul_arith;
```



Now in prode tool we select this hdb file for MUL operation



At the final stage getting this error.
Even after changing port name A to t1data still same error.



Resources:

- http://openasip.org/
- https://github.com/akshaygodse13/PE_TTA/tree/main/tta_samvaad