

This directory includes:

- README.docx
- C source code for simulation
- \amoebasat\_centralized: synthesis source code that uses centralized L update
- \amoebasat\_localized: synthesis source code that uses localized L update
- \amoebasat\_multi\_epsilon
- \benchmarks
- Some preprocessing source codes

To access Vivado 2020.1 on keter: `source /opt/Xilinx/Vivado/2020.1/settings64.sh`

Recommended flow: 1 -> 2 -> 3.2

### 1. Checking the input cnf file

Before using this solver, be sure to check if the cnf file has the correct format of 3-SAT:

- o Head of the file: `p cnf vars# clauses#`
- o Each clause is written on a single line, each line contains uniformly 4 numbers (3 literals and '0' ending)

```
-1 -3 4 0
-1 -5 6 0
-8 -7 9 0
```

To convert k-SAT into 3-SAT, run: `ruby conv3sat.rb "filename.cnf"`

⇒ generate a file "filename\_out.cnf"

the output vars# and clauses# are printed on the head of the file as "p cnf vars# clauses#"

To filled '0' to each line, run: `ruby pad0.rb "filename_out.cnf"`

⇒ generate the valid cnf file: filename\_out2.cnf

### 2. Run the AmoebaSAT solver (C simulation)

C++ source code (written with dynamic memory allocation)

Compile C++ code: `g++ amoebasat.cpp -o amoebasat`

Solve: `./amoebasat "filename.cnf"`

When solving SAT instances, the followings can be modified in the source code:

- o MAX\_N\_STEP: the max iterations#
- o MAX\_CONTRA: size of Contra; if input SAT instance is big that gives segmentation fault, try to increase this size
- o EPSILON: by default, it is set at 0.32 (converted into fixed point value as  $0.32 \cdot 2^{31} = 687194767$ )
- o N\_RUNS: by default, it is set to run 100 times for an instance

C source code (fixed memory allocation): old version, need manual modification & re-compilation

From the source code, edit the followings: N\_VARIABLE (vars#), N\_CLAUSE (clauses#), filename (cnf file path).

Compile: gcc amoebasat0.c -o amoebasat

Solve: ./amoebasat

Operations with Vivado HLS (better open the later versions)

### 3. Synthesize with Vivado HLS (amoebasat\_localized)

Go to directory: cd localized\_hls

Modify parameters in the file amoebasat.h: N\_VARIABLE, N\_CLAUSE (refer to preprocessing results to write in the correct parameters). Also modify the bit width of type f\_t when necessary:

```
typedef ap_ufixed<6,6> f_t;
```

Type f\_t need to have enough bit width to display N\_VARIABLE.

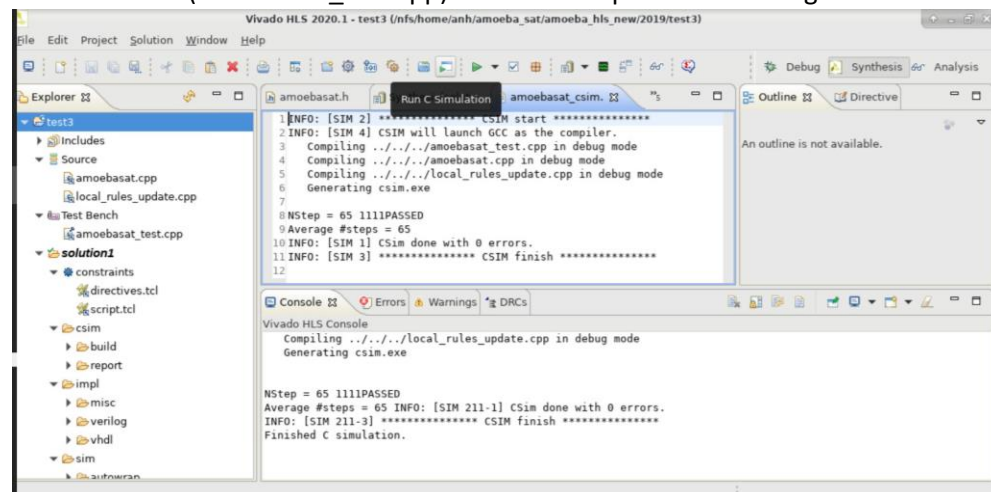
#### 3.1. Using GUI (recommend to check on small instances; if vars#>20, GUI synthesis may take very long time)

##### o Preprocess:

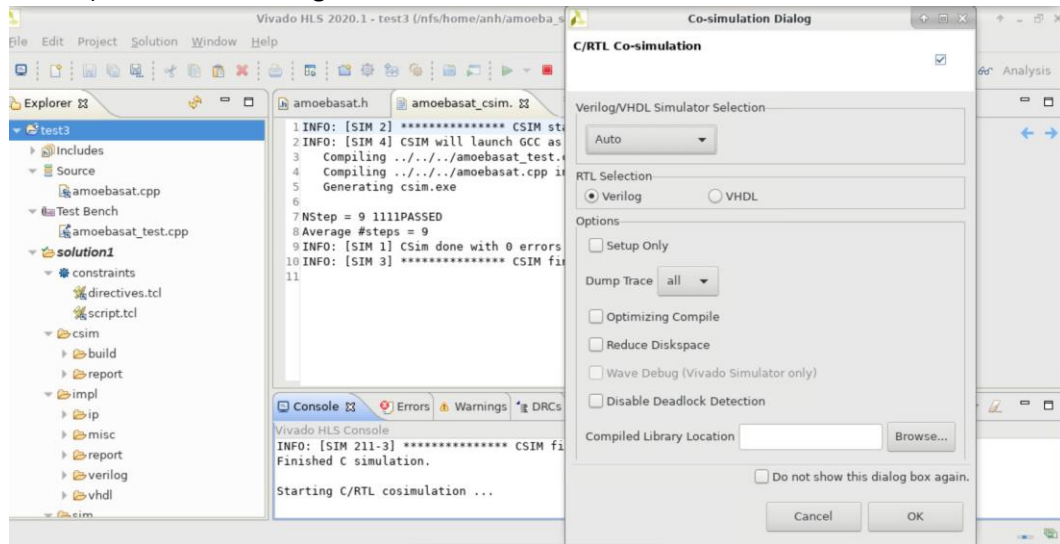
- Generate necessary files (local\_rules.h, local\_rules\_sign.h, local\_rules\_update.cpp, f.h, f\_sign.h):

Run: g++ preprocess\_localized.cpp -o preprocess  
./preprocess ../test.cnf

- Edit parameters in file "amoebasat.h": N\_VARIABLE, N\_CLAUSE, typedef ap\_ufixed<n,n> f\_t (n is the bitwidth needed to represent the literals; if N\_VARIABLE=100, n=7)
- Create the HLS project, choose top function as "amoebasat"
- Include these files: source files (amoebasat.cpp, local\_rules\_update.cpp) & test bench (amoebasat\_test.cpp) - see the left pane of below figure



- Run C simulation (optional): If “PASSED” is printed then the C source code works correctly. (Note: this simulation runs gcc/g++ within Vivado HLS, so almost the same with C simulation in section 2).
- Run C/RTL Co-simulation: hit the button (next 2 buttons on the side of ‘Run C Simulation’ button) -> OK at the dialog



Cosimulation results: Latency at Verilog is the clock cycles#

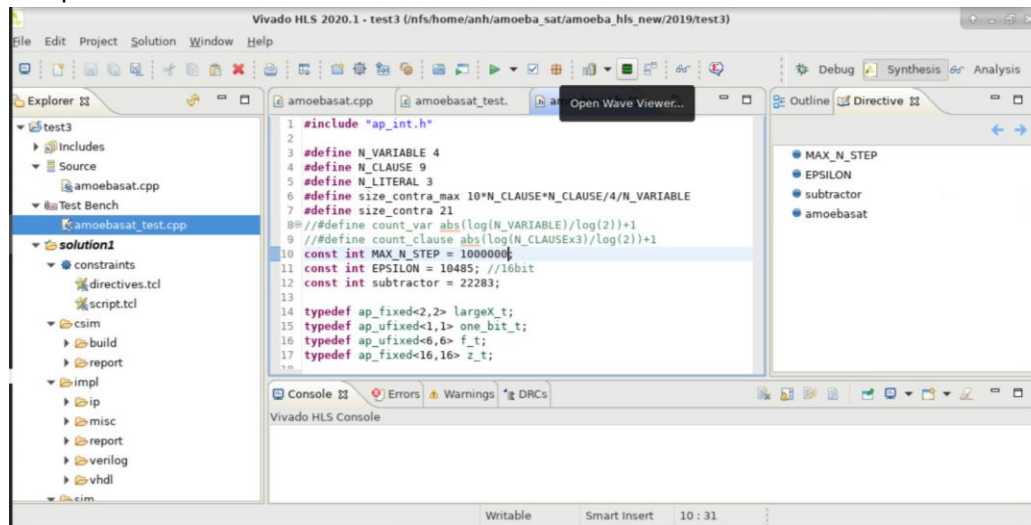
#### Cosimulation Report for 'amoebasat'

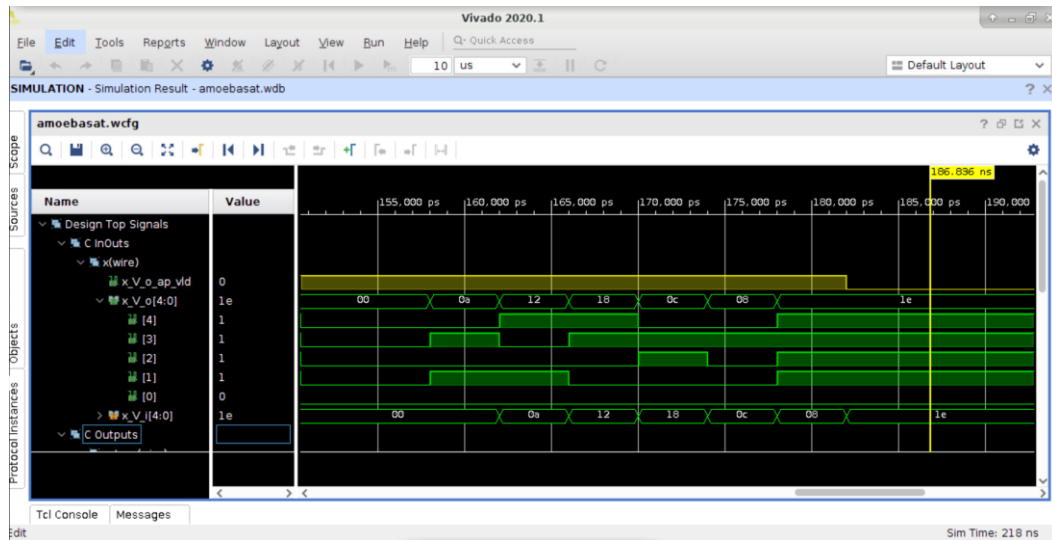
##### Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	20	20	20	NA	NA	NA

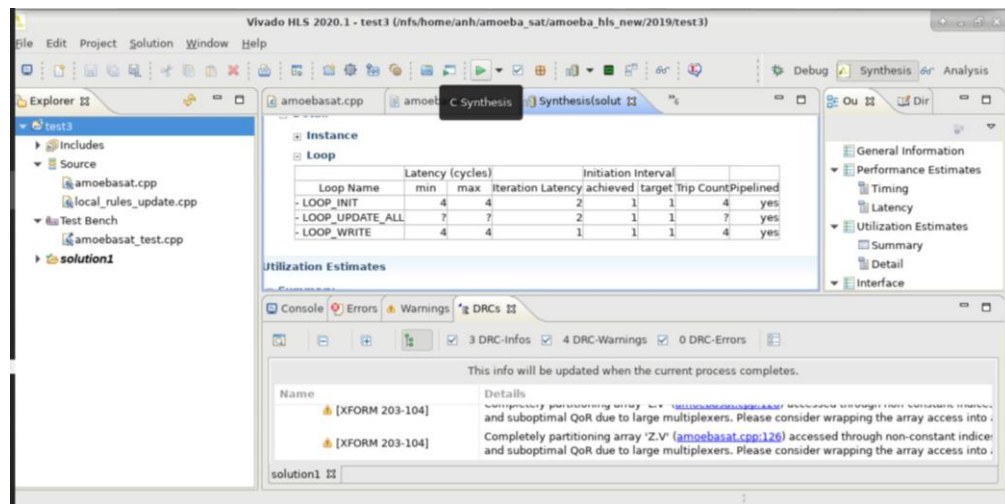
Export the report(.html) using the [Export Wizard](#)

To open waveform view:

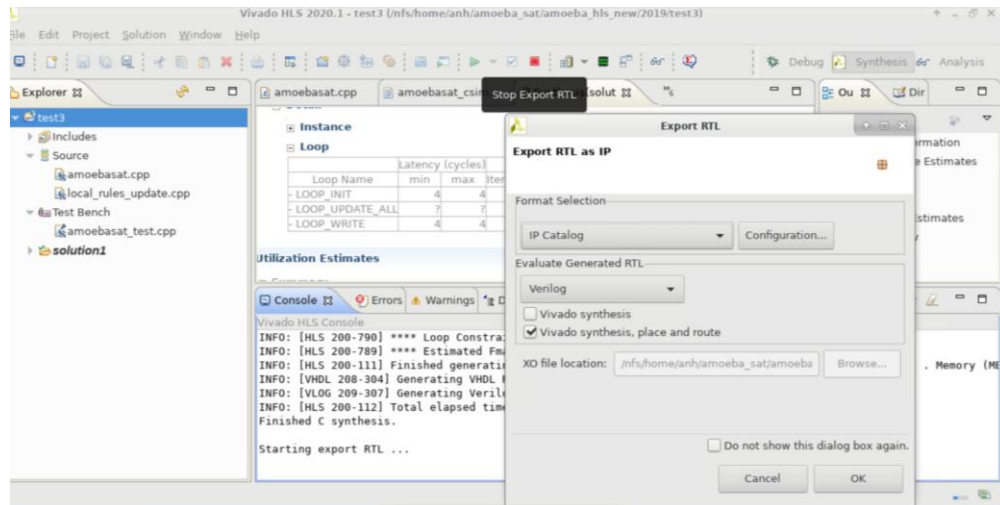




- Synthesize: Hit the C synthesis button  
Ideally, we should get Initiation Interval (achieved) = 1



- Export RTL: Hit the Export RTL button -> Check "Vivado Synthesis, place and route" if you want to get implementation results without manually opening Vivado



### Implementation results

Verilog	
SLICE	281
LUT	775
FF	660
DSP	0
BRAM	0
SRL	0
<b>Final Timing</b>	
	Verilog
CP required	5.000
CP achieved post-synthesis	2.807
CP achieved post-implementation	3.406
Timing met	

Final timing achieved post-implementation: implementation clock speed in ns

### 3.2. Using tcl script (recommended, much faster than GUI): run\_tcl.sh

To save more time, the followings (in green) are commented. Who wants to run all simulations can uncomment green parts

```
localized_hls > $ run_tcl.sh
1  open_project test
2  set_top amoebasat
3  add_files ./amoebasat.cpp
4  add_files ./local_rules_update.cpp
5  #add_files -tb test3/amoebasat_test.cpp -cflags "-Wno-unknown-pragmas"
6  open_solution "solution1"
7  set_part {xc7z030-ffv676-3}
8  create_clock -period 5 -name default
9  #config_export -format ip_catalog -rtl verilog -vivado_optimization_level 2
10 set_clock_uncertainty 12.5%
11 #source "./test3/solution1/directives.tcl"
12 #csim_design -compiler gcc
13 csynth_design
14 #cosim_design -trace_level all
15 export_design -flow impl -rtl verilog -format ip_catalog
```

The target device can be set at line 7.

Constraint for clock speed can be edited at line 8, unit is in nanosecond. For example, in the above figure, clock constraint is set at 5ns.

Run: `vivado_hls run_hls.tcl`

```
Solution:          solution1
Device target:     xc7z030-ffv676-3
Report date:       Sun Nov 21 18:26:58 JST 2021

#=== Post-Implementation Resource usage ===
SLICE:             300
LUT:               773
FF:                652
DSP:               0
BRAM:              0
SRL:               0
#=== Final timing ===
CP required:       5.000
CP achieved post-synthesis: 2.827
CP achieved post-implementation: 3.629
Timing met
```

#### 4. Synthesize other projects with Vivado HLS

- Centralized rules update: `cd centralized_hls`

Run preprocess: `g++ preprocess_centralized.cpp -o preprocess`  
`./preprocess "filename.cnf"`

Synthesize in a similar way to `localized_hls`: `vivado_hls run_hls.tcl`

- Multiple epsilons: `cd multi_epsilon`

Run analysis: `g++ analysis.cpp -o analysis`  
`./analysis "filename.cnf"`

After running analysis, connections# of each unit will be output to a file "conn.txt", line 1 is connections# of unit (1,0), line 2 is connections# of unit (1,1), line 3 refers to unit (2,0) and so on.

Please copy the numbers into excel and perform your own analysis, for ex., calculating average connections# of the instance. So you can find your efficient way to decide which threshold of connections# can separate units into groups. Refer to [An FPGA-based Stochastic SAT Solver Leveraging Inter-Variable Dependencies](#).

After deciding threshold to classify the units into different epsilon values, explore the epsilons set (in this code, units are classified into 2 groups): Modify parameters in the file `exploration.cpp`: `N_VARIABLE`, `N_CLAUSE`, `THRES`, `filename.cnf`

Then run exploration: `g++ exploration.cpp -o exploration`  
`./exploration`

After running, copy the result corresponding to each pair of epsilons (printed in console) into excel, and find the pair of epsilons where average time steps# is the least, that is the tuning set of epsilons.

Update epsilons and N\_VARIABLE, N\_CLAUSE in amoebasat.h

Run the synthesis: `vivado_hls run_hls.tcl`