

TCE tutorial:

From C to VHDL as quickly as possible

- This tutorial shows how you can quickly turn your C code into parallel code and a VHDL-model of a processor using the TCE toolset
- The basic idea is that you compile the C code, pass the sequential code to the design space explorer tool and let it generate a processor design capable of running the program for you
- This tutorial has been tested with rev 3451 of TCE



Initial setup

- Download and unpack the tutorial file package
 - `wget http://tce.cs.tut.fi/tutorial_files/explorer_c2vhdl.tar.gz`
 - `tar -xvzf explorer_c2vhdl.tar.gz`
 - `cd c2vhdl`
- There is an application directory which contains an example C source code, the “complex multiply” benchmark from DSPstone



Compile the sources

- First compile the C sources of the programs into generic sequential TTA programs using the TCE frontend compiler
- Note that the compiler output file must be named 'sequential_program'
 - `gcc-tce -O2 -o application1/sequential_program application1/complex_multiply.c`



InitialMachineExplorer: Generating a Processor for the Program

- First we use “initial machine explorer” to create an initial processor design capable of running the input program
 - *explore -e InitialMachineExplorer -a application1/ -p build_idf=true ExplorerResults.dsdb*
 - -e <plugin> specifies the plugin to be used in exploring
 - -a <dir> specifies an application searchpath to be added to the ExplorerResults database
 - -p option passes the parameter to the plugin - in this case the initial machine explorer plugin is instructed to also generate an implementation description of the processor design (not only an architecture description) so we can generate a VHDL implementation for it
 - Last parameter is the design space data base (DSDB) to store the explored processor configurations (architecture and implementation description file pairs) for later use
- After running for a while, explorer should create a processor configuration to the DSDB with ID 1
 - This part of TCE is quite unoptimized so it can take several minutes to finish
 - To save time, ready-made results are included in 'ExplorerResultsPre.dsdb' which you can use instead of waiting to produce a new DSDB
 - The first configuration includes all the necessary resources but it is fully connected

SimpleICOptimizer: Optimizing Connectivity of the Processor

- Optimize the initial configuration using “simple IC optimizer” which removes unused connections from the processor
 - This phase can be skipped in case a fully connected machine is acceptable
 - *explore -e SimpleICOptimizer -s 1 ExplorerResults.dsdb*
 - -s <conf_ID> option defines the processor configuration to be optimized
- A new processor configuration with a reduced connectivity processor architecture has been added to the database
 - Again, to save time, the resulting DSDB of this phase has been pregenerated to ExplorerResultsPre2.dsdb



Inspecting the Results

- You can export the processor configurations from the database with:
 - `explore -w <conf_ID> <database_file>`
- It will generate <ID>.adf and <ID>.idf files
 - ADF is the architecture description, and IDF describes which implementations to use for each architectural component
 - `explore -w 1 ExplorerResults.dsdb # the configuration before connectivity reduction`
 - `explore -w 2 ExplorerResults.dsdb # the configuration after unused connectivity was removed`
- You can check how the architectures look like by running the Processor Designer GUI:
 - `prode 1.adf & prode 2.adf &`



...Inspecting the Results

➤ If you want to get a (rough) estimate of the die area and the longest path delay of the processors you can run the Cost Estimator:

- *estimate 1.adf 1.idf*
- *estimate 2.adf 2.idf*
- Again, this can take several minutes with the current unoptimized TCE code, so here is the output from the commands:

```
estimate 1.adf 1.idf
total area: 13324 gates
delay of the longest path: 15 ns

estimate 2.adf 2.idf
total area: 12181 gates
delay of the longest path: 15 ns
```

- The longest path delay is 15 ns for both, which equals to about 67 MHz maximum clock rate
 - The longest path was not at IC, thus reducing IC didn't improve the speed
 - Area was reduced by 8.5% with the IC reduction

Generating Code for the Processor

- Compile the sequential program for the generated architecture:
 - `schedule -t 2.adf -o application1/parallel_program.tpef application1/sequential_program`
- Then generate encoding (a binary encoding map file, BEM) for the instructions:
 - (A bug workaround: add a read connection to 'boolean' RF with 'prode 2.adf&' before the next step! TODO: Remove this step from the tutorial after the bug has been fixed.)
 - `createbem 2.adf`
 - This creates 2.bem
- Generate bit image of the instruction memory:
 - `cd application1`
 - `generatebits -b ../2.bem -d -t parallel_program.tpef ../2.adf`
- This creates `parallel_program.img` and `parallel_AS2.img` into the application directory:
 - Contents of the program memory and the AS2 data address space, respectively in zeros and ones ascii format

Inspecting instruction encoding

- You can examine the processor instruction encoding using BEM viewer:
 - `viewbem 2.bem | less`
 - BEM viewer outputs its results to terminal so it is useful to pipe the command with *less* or *more*
- The most interesting number is the total instruction width
 - If you exported the non-optimized configuration from the database and created BEM for it, you can see that reducing connections decreases instruction width
- Under the total instruction width is shown what instruction consists of
- There is also a more detailed explanation for each move slot



Generate the processor implementation

- Generate the processor VHDL implementation using Processor Generator:
 - `generateprocessor -b 2.bem -i 2.idf 2.adf`
- Creates folder 'proge-output' which includes the VHDL files of the processor
 - Generating a test bench for the processor automatically is not yet supported, but will be in the future
 - Test bench will provide the memory components, initialization code, and clock generation code for testing the TTA code in a VHDL simulator easily
- The processor implementation is now ready to be simulated in a VHDL simulator (if you provide a test bench!) and synthesized

