Paying Off Technical Debt of SoC Code-Bases Through Standards and Good Practices

Clyde Laforge, Hamza Boukabache, CROME Team

23 November 2022
Disclaimer

- Many possible solutions
- Simplifications were made
- Don’t hesitate to ask questions during or after the presentation: clyde.laforge@cern.ch
Goals

- Present concept of technical debt
- Present tools and methods used in the CROME project keeping it in check
Technical Debt
Technical Debt

In software development, technical debt [...] is the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.
Technical Debt: continued

Causes:

- Unexpected evolution of the project
- Time constraints
- Lack of oversight
- ...

Consequences

- Shorter time to market
- Hinders further development
- Incurs interests
Identification and Solutions to Technical Debt

Identification:
- Outdated documentation
- Band-aid bugfixes
- Parallel development

Solution:
- Re-factoring
CROME Project
CERN Radiation Monitoring Electronics (CROME)

Two configurations:

Conceptual view of CROME at CERN

- Plastic Air filled ionization chamber
- SPA6 cable
- High Radiation Area
- Low Radiation Area
- Up to 1km
- 1000V
- 100fA

Two configurations:
- CROME Rack at EH1 (North Area)

- Radiation Monitoring and processing units
- Uninterruptible Power Supply
- Includes a battery for continuous operation
Issues in CROME
A tasteless cocktail

- What can I touch?
- Noisy commits

Going further: separate source and build directories

- Reduces noise
- Build configurations
Configuration
**Code Style**

- Uniformity = less mental load

```verilog
startTxxDN <= '1' when wenxDI = '1'
    else '0' when (syncLowCntxDP = syncLowDlyC)
    else startTxxDP;

syncxDN <= '0' when (cpolG = '0' and coreClkRexDP = '1' and startTxxDP = '1') else
    '0' when (cpolG = '1' and coreClkFexDP = '1' and reqxDP = '0') else
    '1' when (syncLowCntxDP = 0 and cpolG = '0') else
    '1' when (syncLowCntNxDP = 0 and cpolG = '1') else
    syncxDP;

startTxxDN <= '1' when wenxDI = '1' else
    '0' when syncLowCntxDP = syncLowDlyC else
    startTxxDP;

syncxDN <= '0' when cpolG = '0' and coreClkRexDP = '1' and startTxxDP = '1' else
    '0' when cpolG = '1' and coreClkFexDP = '1' and reqxDP = '0' else
    '1' when syncLowCntxDP = 0 and cpolG = '0' else
    '1' when syncLowCntNxDP = 0 and cpolG = '1' else
    syncxDP;
```
Extras

- Vivado text base build
- Git guidelines
Configuration
-- configPkg.vhd.in
package configPkg is
  constant triplication : std_logic := @TRIPLICATION@;
  -- [...]
end package configPkg;

$ ./someScript --enable-debug

-- configPkg.vhd
package configPkg is
  constant triplication : std_logic := '0';
  -- [...]
end package configPkg;
GNU Autotools

- Good support
- Very powerful
- Support for software dependency checking
GNU Autotools: Steps

1. Replace configuration dependent values by @MY_VARIABLE@
2. Rename and add .in suffix at the end of the filename
3. Write configure.ac file
4. Enjoy
GNU Autotools: Example

```plaintext
-- configPkg.vhd.in
package configPkg is
  constant frontend_* : std_logic := @FRONTEND@;
  -- [...]
end package configPkg;

# configure.ac
AC_INIT([CMPU_hw], [CMPU_VERSION], [CROME-Support@cern.ch])

AC_ARG_ENABLE([frontend],
   [AS_HELP_STRING([--enable/frontend=[ion/neutron]],
      [selects the frontend (default is ion)]))]

AS_IF([test "x$enable_frontend" == xion],
   [AC_SUBST([FRONTEND], '1')],
   [test "x$enable_frontend" == xneutron],
   [AC_SUBST([FRONTEND], '0')],
   [echo "No frontend specified, defaulting to ionization chamber"
    AC_SUBST([FRONTEND], '1')])

AC_CONFIG_FILES([configPkg.vhd])
AC_OUTPUT
```
GNU Autotools: Example – User interface

$ autoreconf -i . # Creates configure from configure.ac
$ ./configure --enable-frontend=ion
[...]
$ cat configPkg.vhd
package configPkg is
    constant frontend_ion : std_logic := '1';
    -- [...]
end package configPkg;

$ autoreconf -i .
$ ./configure --enable-frontend=neutron
[...]
$ cat configPkg.vhd
package configPkg is
    constant frontend_ion : std_logic := '0';
    -- [...]
end package configPkg;
Live Demo!
Code style
vhdl-style-guide

- Open source
- Can fix code automatically
- Many options with good documentation
- Supports CI-friendly formats
# Makefile.in
lint: $(VHDL_SOURCES)
    vsg -f $(VHDL_SOURCES) -c linting/lintRules.yaml \ 
    --junit linting/lint_junit.xml \ 
    --quality_report linting/lint_quality_report.xml

# .gitlab-ci.yml
check_linting:
    stage: pre-checks
    image: vsg:latest
    script:
    - autoreconf -i
    - mkdir -p build && cd build
    - ../configure
    - make lint
artifacts:
    reports:
        junit: linting/lint_junit.xml
codequality: linting/lint_quality_report.xml
Vivado text-base build
Vivado Build

• **make centered workflow**
  • CI integration
  • Reduce number of entry points

• **Dynamic generation of project**
  • Straightforward to version control
  • Reproducible

• **Compatibility with GUI**
  • Usage of project mode
  • Work taken by the tool
Vivado build: Structure

make

files

VIVADO

+ tcl script
Vivado build: Dependencies

- block diagrams
- IPs
- external IO
- PS/PL IO
- vhdl source
- CROME.xpr
  - cosim
  - synth
    - impl
  - check_syntax
Vivado build: gotchas

- Source files cannot be added blindly to the project: collision if generated files are present in the source directory
  - List all source files explicitly

- Vivado's synthesis can fail without resulting in a non-zero error code
  - Use if clause with `get_property PROGRESS [get_runs synth_1]] != "100%``

- Each vivado run produces many files which are usually not wanted.
  - Disable them by using `vivado -nolog -nojournal -notrace [...]`

- Vivado takes time to launch
  - Reduce the number of steps
Conclusion

- A good technical debt is a managed technical debt
- No easy solution to paying it off
- GNU Autotools: Configuration/Software dependency checking
- vhdl-style-guide: style checking
- Make+tcl+vivado: text-based build
Bonus Round
Git

- Develop in a separate branch
- Do not consider work done as long it is not merged back to main
- Commits should be focused on a single purpose and include the minimum amount of modifications
- Use many small commits during development
- Clean up commit history using "git rebase -i" at the end of feature development
- Write meaningful commit messages: one-line summary followed by sh
Pitchfork project

- Ideas and advice on directory structure of code base
- Written for C++, but valuable in any case
Building

• My one hour build failed at the end due to a typo
  • Use vivado’s “check_syntax” before building
  • Command is not well-behaved, so post processing may be necessary

• Track files and not stages in make files

• Use “git status --ignored” to see if “make clean” does its job correctly