

Filière Systèmes Industriels

Orientation Infotronics

Travail de bachelor

Diplôme 2022

Xavier Clivaz

FPGA CI/CD

Professeur
Prof. Silvan Zahno

Expert
Jérôme Corre

Date de la remise du rapport
19.08.2022



Filière / Studiengang SYND	Année académique / Studienjahr 2021-22	No TB / Nr. BA IT/2022/73
Mandant / Auftraggeber <input checked="" type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>	Etudiant / Student Xavier Clivaz Professeur / Dozent Silvan Zahno	Lieu d'exécution / Ausführungsort <input checked="" type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>
Travail confidentiel / vertrauliche Arbeit <input type="checkbox"/> oui / ja <input checked="" type="checkbox"/> non / nein	Expert / Experte (données complètes) Jérôme Corre - jecensuisse@gmail.com 3D2Cut, Route de Sion 29, 3960 Sierre	

Titre / Titel

FPGA CI/CD - Automated hardware development toolchain

Description / Beschreibung

Continuous Integration (CI) and Continuous Deployment (CD) is a method widely used in the industry for software development. It allows to automatically verify and deploy a software projects. HEI has a fully working Gitlab Server with Docker and Kubernetes available to run Virtualmachines. CI/CD is not as widely adopted for FPGA/Hardware development.

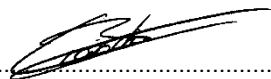
The goal of this thesis is to analyse the current state of the art for automated FPGA workflows (CI/CD), and implement a Gitlab based automated pipeline to automatically generate VHDL code from Spinal, run VHDL simulation and verification pipeline (CI) as well as a bitstream generation workflow (CD).

A proof of concept can be demonstrated by an automated board tester project of the FPGA-EBS Board.

Objectifs / Ziele

- State of the Art research of FPGA CI/CD Tools
- Setup a gitlab runner on a custom PC capable of running the developed pipeline
- Implementation of a detection system to start the appropriate CI/CD workflows only if necessary
- Implementation of a CI-workflow for automatic VHDL/Verilog generation from a HDL-Designer project
- Implementation of a CI-workflow for automatic VHDL testbench simulation and analysis with waveform files stored as manifests
- Implementation of a CD-workflow for automatic generation of Xilinx bitstream file as manifest
- (Optional) Implementation of a hardware based code evaluation with the help of a gitlabrunner and a Digilent Analog/Digital Discovery
- (Optional) Implementation of a CD-workflow for automatic generation of an Actel bitstream file as manifest.

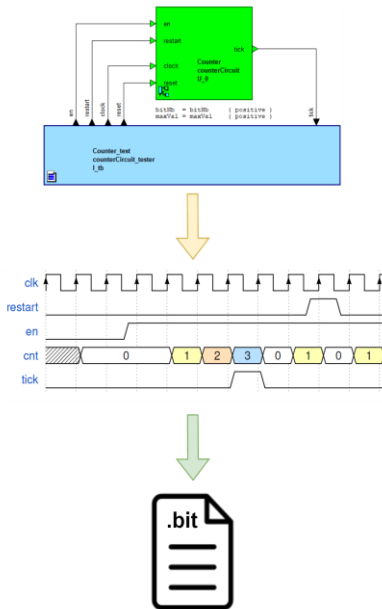
Signature ou visa / Unterschrift oder Visum

Responsable de l'orientation /
Leiter der Vertiefungsrichtung:

¹ Etudiant / Student :


Délais / Termine

Attribution du thème / Ausgabe des Auftrags:
16.05.2022Présentation intermédiaire / Zwischenpräsentation:
20-21.06.2022Remise du rapport final / Abgabe des Schlussberichts:
19.08.22, 12:00Expositions / Ausstellungen der Diplomarbeiten:
24-26.08.2022Défense orale / Mündliche Verfechtung:
Semaine/Woche 36 (5-9.09.2022)

¹ Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.
Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.



FPGA CI/CD – Chaîne d'outils EDA automatisée

Diplômant/e : Xavier Clivaz

Objectif du projet

L'objectif de ce travail est de concevoir un pipeline automatisé pour un développement matériel à l'aide de l'outil GitLab-CI et construit à partir de trois processus, la génération de fichiers VHDL, la simulation et la synthèse.

Méthodes | Expériences | Résultats

Afin de réaliser un pipeline automatisé, il a dû être adapté sur la plateforme d'hébergement GitLab avec l'outil GitLab-CI s'occupant de l'exécution des processus. Cet outil a été installé sur une machine physique ayant l'environnement Windows. Trois flux de travail ont été réalisés pour la construction du pipeline, la génération de fichiers VHDL, la simulation et la synthèse.

L'approche pour y parvenir consistait à dresser en premier lieu un état de l'art sur les logiciels EDA utilisés au sein du CI/CD. Une vue d'ensemble de toute l'automatisation du pipeline a ensuite été réalisée dans le but d'implémenter le tout plus facilement. En parallèle, l'implémentation a été élaborée processus par processus.

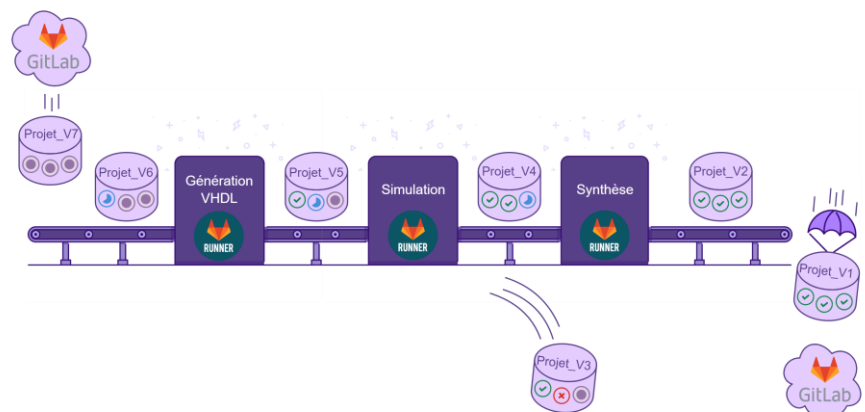
A la fin, une preuve de concept est démontrée. Cette pratique du CI/CD a entièrement pu être réalisée pour un développement matériel. Des tests sur un projet concret ont été effectués pour le prouver.

Travail de diplôme
| édition 2022 |

Filière
Systèmes industriels

Domaine d'application
Infotronics

Professeur responsable
Silvan Zahno
silvan.zahno@hevs.ch



Pipeline automatisé pour développement matériel avec la plateforme GitLab

Information à propos de ce rapport

Information de contact

Auteur : Xavier Clivaz
Etudiant Bachelor
HES-SO//Valais Wallis
Suisse
Email : xavierclivaz@hotmail.com

Déclaration d'honneur

Je, soussigné, Xavier Clivaz, déclare que le travail présenté est le résultat d'un travail personnel. Je certifie que je n'ai pas eu recours au plagiat ou à d'autres formes de fraude. Toutes les sources d'information utilisées et les citations de l'auteur ont été clairement mentionnées.

Lieu, date : 18 août 2022

Signature :



Remerciements

Ce travail de bachelor n'a pas été rédigé durant trois mois sans le soutien de plusieurs personnes. Ces contributeurs, qu'ils soient en étroite relation au projet ou bien à l'écart de ce monde numérique, il est l'heure de tous les remercier.

- Je tiens, en premier lieu, à remercier ma famille et singulièrement mes parents, sans qui, je ne pourrais réaliser tous mes vœux. Je peux ainsi, en toute circonstance, m'épanouir pleinement dans ma vie quotidienne. Il s'agit d'une force d'âme transmise qui me permettra toujours d'aller de l'avant malgré les obstacles rencontrés.
- Je remercie secondement mon responsable, M. Silvan Zahno, qui a toujours été à mon entière disposition. Il a su m'apporter ses précieux conseils tout au long du travail afin d'arriver au résultat désiré.
- Je remercie aussi les collaborateurs de la [Haute Ecole d'Ingénierie](#) et tout particulièrement M. Axel Amand. Ils ont continuellement été disponibles pour fournir leurs connaissances et leurs avis.
- Je remercie, en dernier lieu, mes camarades de classe pour leur soutien et sans qui, je n'aurais pu me changer les idées durant les courtes pauses de midi.

Résumé

Ancrée dans l'esprit des développeurs, la gestion de versions ne se présente plus. Cette technique a été complétée par la méthode du **DevOps** visant à automatiser les processus de développements logiciels à l'aide de l'intégration continue (**Continuous Integration**) et de la distribution continue (**Continuous Delivery**). Ayant observé l'optimisation de ces pratiques, la **Haute Ecole d'Ingénierie** souhaite à présent mettre en oeuvre ces moyens pour des développements matériels.

L'objectif de ce travail est d'implémenter un **pipeline** automatisé pour un développement matériel, plus particulièrement pour un module de développement **FPGA**. Ce **pipeline** doit être adapté sur la plateforme d'hébergement GitLab avec l'outil GitLab-CI s'occupant de l'exécution des processus. Cet outil doit être installé sur une machine physique ayant l'environnement Windows. Trois flux de travail doivent être réalisés pour la construction du **pipeline**, la génération de fichiers **VHDL**, la simulation et la synthèse.

L'approche pour y parvenir consiste à dresser en premier lieu un état de l'art sur les logiciels **EDA** utilisés au sein du **CI/CD**. Une vue d'ensemble de toute l'automatisation du **pipeline** est ensuite réalisée dans le but d'implémenter le tout plus facilement. En parallèle, l'implémentation est élaborée processus par processus.

A la fin, une preuve de concept est démontrée. Cette pratique du **CI/CD** a entièrement pu être réalisée pour un développement matériel. Des tests sur un projet concret ont été effectués pour le prouver.

Key words : CI, CD, FPGA, EDA

Table des matières

Remerciements	vii
Résumé	ix
Table des matières	x
Liste des figures	xii
Liste des tableaux	xiii
Liste des codes sources	xiii
1 Introduction	1
1.1 Problématique	2
1.2 Objectifs	3
1.3 Méthodologie de recherche	4
1.4 Structure du rapport	4
2 Analyse	5
2.1 Définitions	6
2.2 CI/CD pour développement matériel	7
2.3 Outils de CI	7
2.4 Génération de fichiers VHDL à partir de HDS	8
2.5 Simulation	9
2.6 Synthèse et implémentation	12
3 Conception	15
3.1 GitLab Runner	16
3.2 Pipeline	20
3.3 Flux de travail CI	22
3.4 Flux de travail CD	26
4 Implémentation	29
4.1 GitLab Runner	30
4.2 Pipeline	34
4.3 Flux de travail CI	43
4.4 Flux de travail CD	51
5 Validation	57
5.1 Tests des différents flux de travail	58

5.2	Tests sur carte de développement FPGA	64
5.3	Discussion	65
6	Conclusion	67
6.1	Résumé du projet	67
6.2	Comparaison avec les objectifs initiaux	67
6.3	Difficultés rencontrées	68
6.4	Perspectives d'avenir	68
A	Docker	69
A.1	Installer l'image de Gitlab Runner	69
A.2	Lier GitLab Runner au référentiel GitLab	69
B	ActiveTcl	73
B.1	Installer le programme ActiveTcl	73
C	Sous-modules	79
C.1	Intégrer un sous-module dans un répertoire	79
C.2	Cloner un répertoire avec un ou plusieurs sous-modules	82
C.3	Mettre à jour un sous-module dans un répertoire	82
D	Protocole de test - Projet chronomètre	83
	Bibliographie	97
	Glossaire	101
	Acronymes	103

Table des figures

1.1	Exemple de gestion de versions avec deux branches	1
1.2	Méthode DevOps	1
1.3	Illustration d'une automatisation de plusieurs processus	2
1.4	Carte de développement FPGA, FPGA-EBS	3
1.5	Pipeline automatisé avec la plateforme GitLab [1]	3
2.1	Chemin de programmation d'une carte FPGA	7
2.2	Arborescence des conversions pour la génération VHDL	9
2.3	Organigrammes montrant la procédure générale de simulation (à gauche) et les commandes à utiliser à chaque étape (à droite).	11
3.1	Schéma de fonctionnement avec GitLab Runner	16
3.2	Répertoire de test pour le service de GitLab Runner	17
3.3	Service de GitLab Runner opérationnel	18
3.4	Passage du pipeline de test avec succès	19
3.5	Passage du pipeline de test refusé	20
3.6	Pipeline des processus de développement matériel	21
3.7	Vue d'ensemble du workflow génération VHDL	22
3.8	Schéma fonctionnel du workflow génération VHDL	23
3.9	Vue d'ensemble du workflow simulation	24
3.10	Schéma fonctionnel du workflow simulation	25
3.11	Vue d'ensemble du workflow synthèse	26
3.12	Schéma fonctionnel du workflow synthèse	27
4.1	Préparation du fichier d'exécution de GitLab Runner	30
4.2	Enregistrement du service de GitLab Runner dans un référentiel partagé	31
4.3	Changement du nom de l'exécutable <i>PowerShell</i> dans le fichier config.toml	31
4.4	Vérification du service de GitLab Runner non opérationnel sur le référentiel partagé	32
4.5	Erreur de lancement du service de GitLab Runner	32
4.6	Fenêtre du gestionnaire de services Windows	33
4.7	Fenêtre des propriétés utilisateurs du service de GitLab Runner	33
4.8	Vérification du service de GitLab Runner opérationnel sur le référentiel partagé	34
4.9	Etapes du pipeline	35
4.10	Illustration d'un sous-module	38
4.11	Exemple de tâches présentes dans le programme HDS	45
4.12	Exemple de librairie, d'entité et d'architecture dans un projet	46
4.13	Emplacement des fichiers générés dans un exemple de projet	47
4.14	Emplacement du fichier concaténé dans un exemple de projet	48
4.15	Exemple d'affichage d'informations de la génération	48

5.1	Projet chronomètre	58
5.2	Meilleur cas : tous les jobs sont activés	59
5.3	Réussite du pipeline avec les quatre jobs	59
5.4	Meilleur cas : la simulation est désactivée	60
5.5	Réussite du pipeline sans la simulation	60
5.6	Réussite de la génération VHDL avec la simulation et la synthèse activées	61
5.7	Exemple d'artefacts déposés après le job de synthèse	61
5.8	Pire cas : erreur de génération VHDL	62
5.9	Echec du pipeline à cause du job de génération VHDL	62
5.10	Pire cas : erreur de simulation	63
5.11	Echec du pipeline à cause du job de simulation	63
5.12	Erreur de simulation	64
A.1	Installation de l'image Docker préconfigurée	69
A.2	Enregistrement du GitLab Runner dans un référentiel partagé	70
A.3	GitLab Runner lié	71
A.4	GitLab Runner opérationnel	71
B.1	Création d'un compte ActiveState	73
B.2	Sélection du programme et des paramètres	74
B.3	Sélection de la suite des paramètres	74
B.4	Téléchargement d'un fichier ZIP	75
B.5	Téléchargement d'un exécutable	75
B.6	Fenêtre du contrat de licence	76
B.7	Fenêtre des types de configuration	76
B.8	Fenêtre des options de configuration	77
B.9	Installation terminée	77
C.1	Création de deux projets dans GitLab	79
C.2	Clonage local du répertoire "Projet"	79
C.3	Ajout du sous-module dans le répertoire local "Projet"	80
C.4	Vérification du sous-module dans le répertoire local "Projet"	80
C.5	Poussée du répertoire local "Projet" au référentiel partagé	81
C.6	Vérification du sous-module inclus dans le dépôt "Projet" sur GitLab	81

Liste des tableaux

2.1	Comparatif des différents outils CI	8
-----	---	---

Liste des codes sources

3.1	Configuration du fichier <i>.gitlab-ci.yml</i>	18
4.1	Script : <i>.gitlab-ci.yml</i> , structure incomplète	36
4.2	Script : <i>.gitlab-ci.yml</i> , partie : job d'initialisation	37
4.3	Script : <i>.gitlab-ci.yml</i> , partie : job de génération VHDL	39
4.4	Script : <i>.gitlab-ci.yml</i> , partie : job de simulation	40
4.5	Script : <i>.gitlab-ci.yml</i> , partie : job de synthèse	41
4.6	Script : <i>.gitlab-ci.yml</i> , partie : variables d'environnement globales	42
4.7	Script : <i>GenVHDL_Workflow_main.tcl</i> , partie : outils requis	43
4.8	Script : <i>searchPaths.tcl</i> , partie : recherche HDS	44
4.9	Script : <i>GenVHDL_Workflow_main.tcl</i> , partie : variables d'environnement	44
4.10	Script : <i>GenVHDL_Workflow_main.tcl</i> , partie : lancement d'HDS en mode shell	46
4.11	Script : <i>GenVHDL_Workflow_HDS.tcl</i> , partie : configuration de la génération VHDL	46
4.12	Script : <i>GenVHDL_Workflow_HDS.tcl</i> , partie : génération de fichiers VHDL	46
4.13	Script : <i>GenVHDL_Workflow_HDS.tcl</i> , partie : concaténation des fichiers VHDL	47
4.14	Script : <i>GenVHDL_Workflow_main.tcl</i> , partie : suppression des importations de librairie	48
4.15	Script : <i>checkErrors.tcl</i> , partie : HDL Designer Series	49
4.16	Script : <i>Sim_Workflow_main.tcl</i> , partie : outils requis	49
4.17	Script : <i>Sim_Workflow_ModelSim.tcl</i> , partie : création de la librairie de simulation	50
4.18	Script : <i>Sim_Workflow_ModelSim.tcl</i> , partie : compilation	50
4.19	Script : <i>Sim_Workflow_ModelSim.tcl</i> , partie : simulation	51
4.20	Script : <i>checkErrors.tcl</i> , partie : ModelSim	51
4.21	Script : <i>Syn_Workflow_main.tcl</i> , partie : outils requis	52
4.22	Script : <i>Syn_Workflow_main.tcl</i> , partie : lancement d'ISE en mode shell	52
4.23	Script : <i>project_name.xise</i> , exemple de projet ISE	53
4.24	Script : <i>Syn_Workflow_ISE.tcl</i> , partie : ouverture d'un projet ISE	53
4.25	Script : <i>Syn_Workflow_ISE.tcl</i> , partie : création d'un projet ISE	54
4.26	Script : <i>Syn_Workflow_ISE.tcl</i> , partie : sauvegarde des propriétés des processus	54
4.27	Script : <i>Syn_Workflow_ISE.tcl</i> , partie : processus de synthèse	55
4.28	Script : <i>Syn_Workflow_ISE.tcl</i> , partie : fermeture du projet ISE	55
5.1	Script : <i>Sim_Workflow_main.tcl</i> , partie : outils requis	65

1 | Introduction

Ancrée dans l'esprit des développeurs, la gestion de versions ne se présente plus. Cette technique favorise le suivi et la maintenance des changements apportés au code d'un développement logiciel. Elle a l'avantage de garder un historique complet des modifications et de faciliter grandement le travail collaboratif sur un même projet.

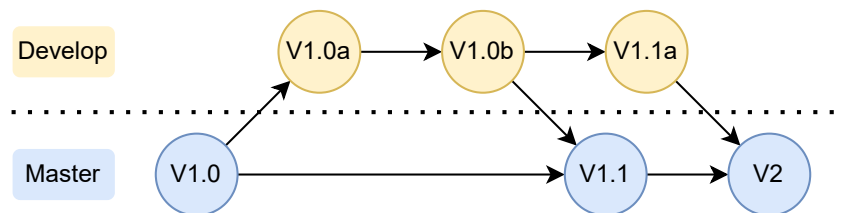


Figure 1.1 Exemple de gestion de versions avec deux branches

Effectivement, la figure 1.1 se compose de plusieurs versions ainsi que de branches différentes. Bien évidemment qu'il pourrait y avoir plus de branches pour que chaque développeur en possède une à lui seul. Cela permettrait de travailler chacun sur une part du projet sans qu'un conflit n'entre en jeu au sein des collaborateurs. Lorsqu'une partie est terminée, une version est alors créée. Si dans un futur proche, un problème venait à être perçu par un employé ou même un client, revenir à une version antérieure est tout à fait envisageable.

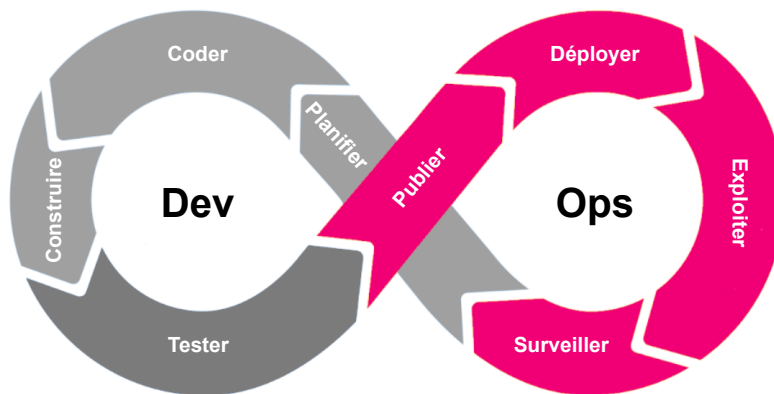


Figure 1.2 Méthode *DevOps*

D'autres méthodes viennent compléter la gestion de versions telle que le *DevOps* (figure 1.2). Celle-ci vise à automatiser les processus d'un développement logiciel entre les équipes de développement (partie grise de la figure 1.2) et les équipes d'exploitation (partie rose de la figure 1.2) pour renforcer la collaboration. Elle a donc élaboré l'intégration continue (*Continuous Integration*) (CI) et la distribution continue (*Continuous Delivery*) (CD) qui sont devenues les moyens de prédilection envers les développeurs. Ils permettent l'automatisation de plusieurs processus lors de chaque nouvelle version. Ces outils consistent généralement à construire, tester et déployer une conception logicielle (figure 1.5). La motivation de cette invention découle du gain de temps entre le lancement d'un projet et la commercialisation de celui-ci, exprimé en anglais par

"time to market". Effectivement, les machines sont en constante évolution alors que les êtres humains sont limités physiquement et moralement. De nombreux autres avantages peuvent être mentionnés tels que la collaboration en équipe, un débogage plus facile, une réduction des coûts, ainsi que la qualité et bien d'autres.

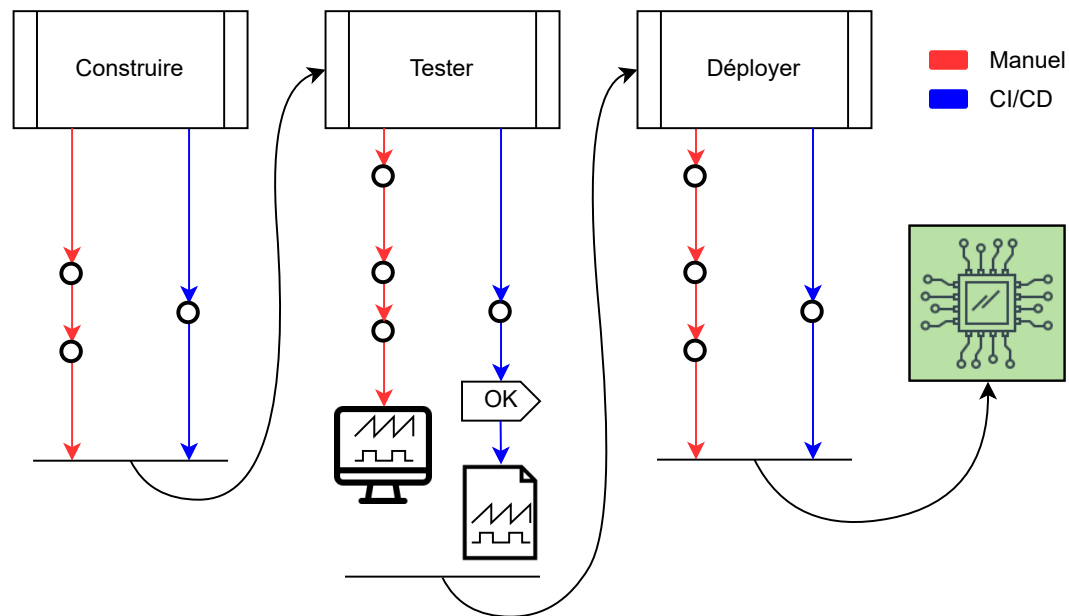


Figure 1.3 Illustration d'une automatisation de plusieurs processus

La figure 1.5 illustre parfaitement la distinction entre le monde manuel (flèches rouges) et le monde du CI/CD (flèches bleues) des processus. Les tâches de construction et de déploiement demandent nettement moins d'actions pour s'exécuter entièrement. La phase de test est un peu différente. Le résultat est généralement validé visuellement par le développeur à l'aide de comportements physiques ou virtuels. Le CI va valider lui-même les comportements en insérant des tests. Il va répondre par une réponse totale, soit oui, soit non, pour passer au déploiement. Un fichier contenant les résultats peut toujours être créé. Selon la réussite de tous les processus, le logiciel est transféré directement vers la carte électronique afin qu'elle soit programmée et fonctionnelle. Le tuyau dans lequel cheminent toutes ces tâches s'appelle un **pipeline**.

La **Haute Ecole d'Ingénierie (HEI)** fait partie intégrante des universités et entreprises travaillant avec ces moyens d'aujourd'hui. Elle met en place ce système pour des projets qui concernent le développement de produits logiciels.

1.1 Problématique

La **HEI** étant une université en même temps qu'une entreprise, elle traite une large gamme de spécialisations dans le domaine de l'industrie. Ayant observé l'optimisation de ces pratiques sur des développements logiciels, elle souhaite à présent mettre en oeuvre un système de processus automatisé consacré au développement matériel. Il s'agit d'un développement où l'on programme un circuit intégré à partir d'un **langage de description**

matériel (**Hardware Description Language**) (HDL) (figure 1.4). L'inexistence réside dans le fait que le CI pour ces développements est peu connu et faiblement exploité dans l'industrie.

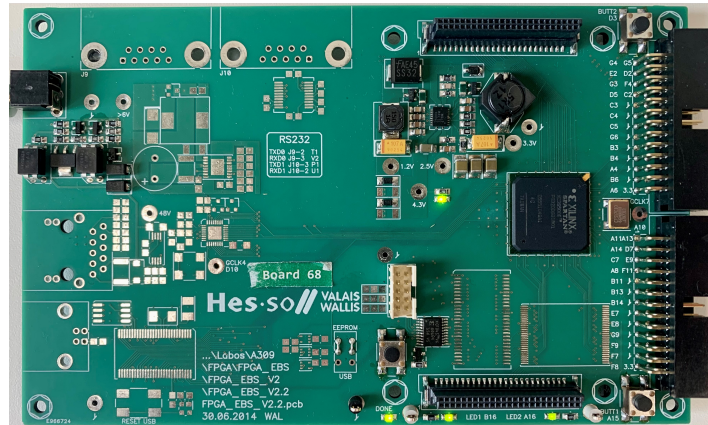


Figure 1.4 Carte de développement *FPGA*, *FPGA-EBS*

Il s'agit véritablement d'une problématique puisque ces conceptions matérielles sont tout aussi complexes que celles logicielles. Effectivement, aussitôt qu'un produit comporte plusieurs parties distinctes, la difficulté de les associer s'accroît diligemment. Sans ces outils CI, les parties sont généralement testées séparément, ce qui ne garantit en rien le fonctionnement global. Il vaut mieux procéder à des tests regroupant tous ces blocs simultanément.

Les principaux acteurs touchés sont les développeurs. Ils ne permettent pas de garantir la qualité du produit sans devoir passer un nombre inconsidérable d'heures supplémentaires au développement. Les clients sont aussi atteints en raison de leur position en bout de chaîne de production. Tous les retards accumulés leurs sont répercutés au déploiement. Il ne s'agit donc pas d'un avantage concurrentiel.

1.2 Objectifs

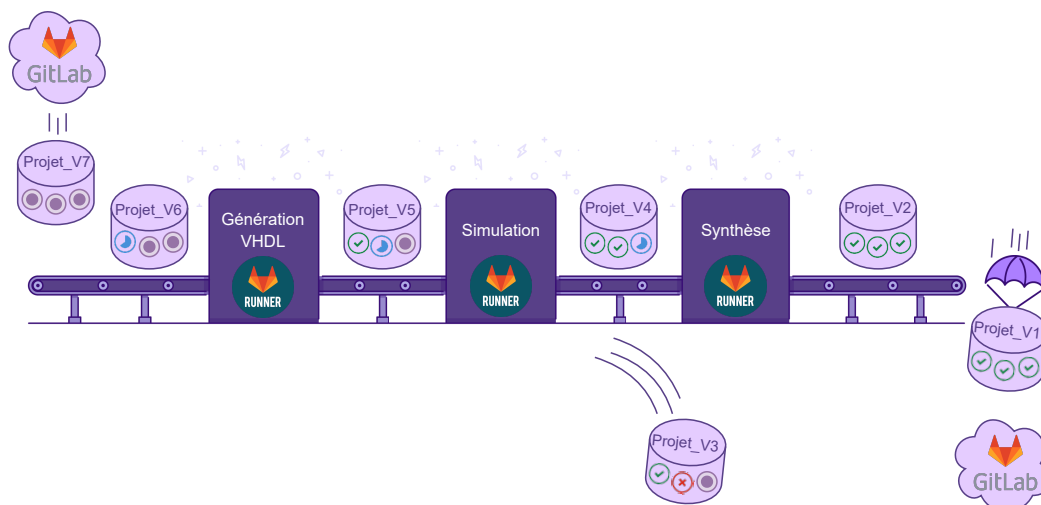


Figure 1.5 *Pipeline* automatisé avec la plateforme *GitLab* [1]

Le but premier de ce travail est d'implémenter un [pipeline](#) automatisé pour un développement matériel, plus particulièrement pour un module de développement [FPGA](#) (figure 1.4). Ce [pipeline](#) doit être adapté sur la plateforme d'hébergement GitLab [1] établie dans les serveurs de la [HEI](#). L'outil de GitLab s'occupant de l'exécution des processus, appelé GitLab Runner [2], doit être installé sur une machine physique afin d'éviter l'installation de tous les programmes de conception à chaque lancement du [pipeline](#). Il est demandé de concevoir plusieurs flux de travail. Premièrement, une génération du code source en [VHDL](#) doit être faite à partir du programme [HDL Designer Series](#) ([HDS](#)). En suivant, une simulation ainsi qu'une analyse doivent être élaborées. En dernier lieu, il reste la synthèse. Elle s'occupe de générer le fichier de configuration, appelé [bitstream](#), pour un [réseau de portes programmables sur site](#) ([Field-Programmable Gate Array](#)) ([FPGA](#)) du fabricant Xilinx [3], prêt à être déployé sur le module de développement. Il est aussi demandé de gérer l'utilisation des différents flux de travail qu'en cas de nécessité. A savoir que l'utilisateur est libre de lancer une simulation et/ou une synthèse à chaque nouvelle version du projet.

1.3 Méthodologie de recherche

La méthodologie de recherche utilisée dans cette thèse consiste tout d'abord à dresser un état de l'art sur les outils de développement matériel utilisés au sein du [CI/CD](#). Une fois la recherche scientifique terminée, une vue d'ensemble de toute l'automatisation du [pipeline](#) sera réalisée dans le but d'implémenter le tout plus facilement. Un choix des programmes utilisés sera aussi effectué. En parallèle, l'implémentation sera élaborée. Une validation des résultats sera faite lorsque l'implémentation sera terminée.

1.4 Structure du rapport

Le chapitre 2 (Analyse) contient plusieurs définitions et l'état de l'art. Il s'agit de définitions de termes importants qui constituent le projet. L'état de l'art concerne les outils actuels pour le développement matériel intégrés dans le [CI](#).

Le chapitre 3 (Conception) contient la vue d'ensemble de l'implémentation ainsi que le choix des outils observés.

Le chapitre 4 (Implémentation) contient la mise en oeuvre de l'outil d'[intégration continue](#) ([Continuous Integration](#)) en plus des flux de travail pour réaliser les objectifs visés.

Le chapitre 5 (Validation) contient l'analyse des résultats de l'implémentation et la validation de ceux-ci.

Le chapitre 6 (Conclusion) conclue la thèse en donnant l'état actuel de la recherche, les problèmes rencontrés ainsi que les étapes futures.

2 | Analyse

L'analyse permet tout d'abord d'expliquer les concepts fondamentaux qui sont travaillés tout au long du projet. Par après, un état de l'art est dressé sur les différents outils existants pour l'intégration et la livraison continue en ce qui concerne le développement matériel.

Table des matières

2.1	Définitions	6
2.1.1	FPGA	6
2.1.2	Intégration continue et livraison continue	6
2.1.3	Référentiel partagé	7
2.2	CI/CD pour développement matériel	7
2.3	Outils de CI	7
2.4	Génération de fichiers VHDL à partir de HDS	8
2.5	Simulation	9
2.5.1	Tests	9
2.5.2	Simulateurs	11
2.5.3	Compléments	12
2.6	Synthèse et implémentation	12

2.1 Définitions

2.1.1 FPGA

FPGA est l'acronyme de **réseau de portes programmables sur site (Field-Programmable Gate Array)**. Il s'agit d'un circuit intégré pouvant être programmable de manière matérielle autant de fois que nécessaire après sa fabrication. Il est composé d'un nombre incalculable de portes. Un HDL spécifique permet de câbler ces portes (**VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language (VHDL)**) dans le cadre de ce travail).

2.1.2 Intégration continue et livraison continue

Plongé actuellement dans l'industrie 4.0, l'efficacité pour tout développement logiciel doit être de mise. Les développeurs ne souhaitent plus perdre leur temps sur des tâches chronophages telles que les phases de test de leurs développements. Les utilisateurs, quant à eux, restreignent de plus en plus leur délai pour obtenir l'application qu'ils désirent. C'est ainsi que des méthodes doivent permettre de répondre à ces innombrables contraintes. Il existe une pratique de plus en plus sollicitée qui se nomme le **DevOps**. Le **DevOps** se construit à partir d'un **pipeline d'intégration continue (Continuous Integration) (CI)** et de **distribution continue (Continuous Delivery) (CD)**. Il s'agit d'une manière d'automatiser une grande partie des **workflows** (figure 1.2). L'idée est de remplacer certaines tâches des développeurs (partie grisée de la figure 1.2) ainsi que des opérateurs (partie rosée de la figure 1.2) pour rendre le tout avantageux.

Le **CI**, qui concerne les développeurs, assure continuellement le fonctionnement global du programme après chaque modification apportée. Un projet logiciel comporte bien souvent de nombreuses parties qui rendent le tout complexe. Pour que plusieurs développeurs puissent travailler sur des parties distinctes, un processus de construction ainsi que de test sont lancés pour garantir qu'aucun conflit ne soit présent. Seulement par après, les modifications sont fusionnées avec le programme sur un référentiel partagé qui se situe sur une plateforme d'hébergement.

La **CD**, parfois similaire au déploiement continu, est le processus qui publie les mises à jour fusionnées sur le référentiel partagé. Ce référentiel est l'endroit visible par les opérateurs qui vont pouvoir procéder par la suite au déploiement du programme manuellement. Le déploiement continu est le déploiement du programme mais de manière automatique. Il s'agit de la phase de transfert de la mise à jour depuis le référentiel vers l'utilisateur.

Un **pipeline CI/CD** améliore considérablement la collaboration de plusieurs développeurs au sein du même projet. Elle optimise également la vitesse de développement et garantit la fiabilité et la qualité du logiciel. La communication entre la partie développement et opération est nettement plus rapide. Les clients obtiendront donc leur application dans un délai bien plus court pour bénéficier d'un avantage concurrentiel. Ils pourront aussitôt donner des informations sur l'utilisation du logiciel directement aux développeurs pour qu'ils recommencent leur cycle de mise à jour.

2.1.3 Référentiel partagé

Dans le milieu du développement logiciel et matériel, un référentiel partagé désigne un répertoire sur une plateforme d'hébergement permettant le développement collaboratif (GitLab [1] dans le cadre de ce travail). Il stocke les fichiers bruts et en fait différentes versions en fonction de chaque modification apportée. Un système de branche peut aussi être possible afin de travailler chacun sur une branche distincte pour éviter tout conflit. Généralement tous les hébergeurs rendent leur plateforme compatible au DevOps.

2.2 CI/CD pour développement matériel

Il faut tout d'abord bien comprendre la procédure de programmation d'un développement matériel pour un FPGA. Un chemin d'étapes est représenté à la figure 2.1. Il est indispensable de générer au départ un code VHDL. Un processus de synthèse s'en suit pour créer une netlist qui décrit le circuit électronique. Le placement et le routage, appelés implémentation, sont les phases qui suivent afin de positionner les composants du schéma dans la structure d'un FPGA et établir la liaison entre chaque composant. Ceci est sauvé dans un fichier bitstream. Pour terminer, il suffit de programmer la carte.



Figure 2.1 Chemin de programmation d'une carte FPGA

La figure 2.1 montre parfaitement qu'un développement matériel traverse différentes étapes tout comme un développement logiciel. Le développement matériel peut donc très bien passer par un pipeline automatisé. Les outils nécessaires vont quant à eux différer de ceux d'un développement logiciel.

Pour donner une idée concrète, il suffit de prendre l'exemple d'un circuit à développer pour une carte FPGA. Plusieurs développeurs collaborent sur ce même projet. Ils amènent chacun des modifications qu'ils vont ensuite vouloir centraliser sur un référentiel. Lors de chaque poussée, avant d'ajouter ces mises à jour, la génération des fichiers en code VHDL va être effectuée (partie construction de la figure 1.2). Seulement par après, une simulation peut être lancée depuis un banc de test pour vérifier le bon fonctionnement dans son ensemble (partie test de la figure 1.2). Si ceci est en ordre, la fusion des mises à jour peut être faite. La suite des opérations concerne le CD. Afin de rendre prêt l'opérateur à programmer la carte, il est primordial de synthétiser le code VHDL et l'implémenter en un fichier bitstream propre au fabricant de la carte FPGA (partie publication de la figure 1.2).

2.3 Outils de CI

Cette analyse sur les outils d'intégration continue (Continuous Integration) est tirée d'un ancien travail de diplôme [4].

Plusieurs outils sont existants pour l'exécution des différents workflows d'un pipeline. Une liste non exhaustive est présente ci-dessous. Elle regroupe les noms les plus connus :

- Gitlab-CI [5]

- Jenkins [6]
- Bitbucket Pipelines [7]
- Travis CI [8]

Une analyse de ces différents outils a été réalisée uniquement avec des cas d'utilisation gratuits.

Caractéristiques	GitLab-CI	Jenkins	Bitbucket-Pipelines	Travis CI
Dédier une machine physique comme exécuteur	oui	oui	oui	non
Possibilité de liaison avec Gitlab	oui	oui	non	oui
Temps disponible par mois pour le CI	400 min pour runner de GitLab	pas de limite	50 min pour runner de Bitbucket	pas de limite
Espace mémoire pour les artefacts	configurable / temps de garde configurable	Espace de la machine à disposition	1GB	aucun
Possibilité d'effectuer plusieurs tâches simultanément	oui	oui	oui	oui
Informe en temps réel les étapes du CI	oui	oui	oui	oui

Table 2.1 Comparatif des différents outils CI

GitLab-CI est l'outil le plus favorable dans le cadre de ce travail. La plateforme d'hébergement utilisée étant GitLab [1], il s'agit de son propre outil. De plus, il est déjà exploité dans d'autres projets de la HEI. GitLab-CI peut être auto-hébergé sur une machine physique à l'aide d'un petit logiciel lié aux serveurs de GitLab appelé GitLab Runner [2]. Un runner (GitLab Runner dans ce cas) est un programme qui exécute toutes les commandes implémentées dans le pipeline. De plus, sa durée d'utilisation pour le CI devient illimitée. Il est tout à fait possible de sauver tous les fichiers qui contiennent les informations de chaque étape du pipeline. Ces fichiers s'appellent des artefacts.

Jenkins est aussi un outil qui aurait pu être employé. Un runner existe pour être auto-hébergé sur une machine physique. Il est compatible avec la plateforme GitLab [1] et offre un espace considérable pour les artefacts. Il est cependant plus difficile à configurer et donc moins intuitif.

Bitbucket Pipelines est beaucoup plus restreint. Il ne permet pas la liaison avec la plateforme utilisée. Il est aussi limité par le stockage maximal des archives d'artefacts.

Tout aussi contraignant, Travis CI ne possède pas de runner pour une machine physique. Il n'est d'ailleurs pas possible de sauver les informations des différentes étapes.

Il est à noter que toutes les informations ont été mises à jour au mois de juin 2022.

2.4 Génération de fichiers VHDL à partir de HDS

Cette partie concernant la génération de fichiers VHDL est spécifique à l'environnement HDL Designer Series [9]. Il prend en charge le langage HDL pour la conception d'FPGA. Il est possible d'implémenter le design sous différents aspects tels que par une machine

d'état, des diagrammes en bloc ou encore directement dans le langage HDL (figure 2.2). A la fin de la conception, le passage par un langage HDL est obligatoire pour pouvoir par la suite synthétiser et programmer la carte FPGA.

Le logiciel HDS a la possibilité de convertir tous ces types d'implémentation directement en lignes de commande. Il s'agit d'un langage de script Tool Command Language (Tcl) qui remplace l'interface graphique. Toute une documentation [10] sur les commandes Tcl est fournie. Ceci est un avantage majeur puisque le CI exécute ses tâches uniquement par des lignes de commande.

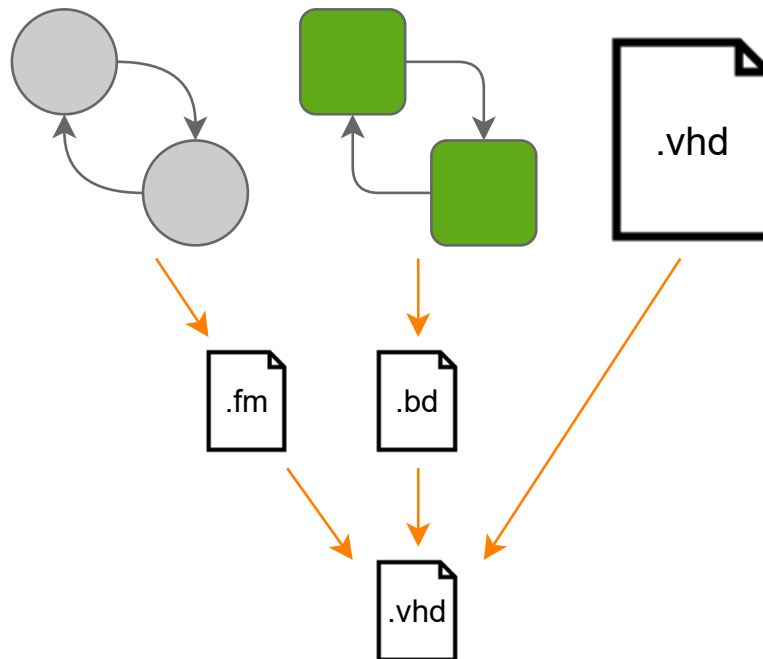


Figure 2.2 Arborescence des conversions pour la génération VHDL

2.5 Simulation

L'article scientifique [11] explique que cette étape de simulation dans un pipeline est spécifiquement importante car elle permet d'exécuter les tests dans un milieu déterministe invariant dans le temps et elle contribue beaucoup à la reproductibilité des résultats. Afin de maximiser les avantages qu'un tel développement puisse offrir, indiquer clairement les moyens et l'environnement de test exploités est capital. Les développeurs ne sont absolument pas certains que les résultats n'aient pas été affectés de manière involontaire lorsqu'ils lancent leurs exécutions manuellement sur leur propre machine. Effectivement, différentes versions d'outils peuvent exister tout comme des changements locaux sur leur machine.

2.5.1 Tests

La thèse [12] démontre l'utilisation de plusieurs outils pour cette étape de simulation. Elle dit utiliser un outil spécialement pour les tests unitaires appelé VUnit [13] et un banc de test pour les tests globaux appelé cocotb [14]. VUnit et cocotb sont des programmes permettant de mettre en oeuvre les tests avec le langage Python. L'auteur Alberto Martínez Cuesta [12] explique que ces outils ont été créés dans le but de ne pas utiliser des

langages de description matériels relativement compliqués, qui sont développés spécifiquement pour la conception et non pour les tests. La conception et la simulation doivent être traitées avec des démarches distinctes puisque ce sont des problèmes différents. La construction de tests est simplifiée par l'utilisation d'un outil logiciel qui est un langage de très haut niveau.

La documentation VUnit [13] explique que ce logiciel ne fait que compléter les méthodologies de test traditionnelles en favorisant l'approche de tester tôt et souvent. Il n'est donc pas possible de remplacer en totalité les tests avec ce programme. La vérification qu'offre celui-ci réduit donc la charge des tests en prenant en charge la détection automatique des bancs de test. L'avantage de VUnit [13] est qu'il peut être utilisé n'importe où dans la phase de test, autant au début qu'à la fin.

La documentation cocotb [14] dit que son nom signifie "COroutine based COsimulation TestBench". Il est spécifié qu'il requiert un simulateur pour simuler la conception HDL peu importe le système d'exploitation. Un de ses avantages est qu'il contient un support pour le CI. Il a spécifiquement été conçu pour réduire les charges générales de test. La documentation donne plusieurs atouts de programmer en Python. L'écriture de ce langage est rapide. Il est facile de coder d'autres langages par le biais de celui-ci. Python dispose de beaucoup de librairies. Il est aussi très populaire.

L'outil HDS n'est malheureusement pas très connu par les scientifiques au niveau de la simulation. Pourtant, dans sa documentation [15], Mentor Graphics explique posséder un assistant permettant d'automatiser le processus de vérification et de test des conceptions en utilisant les nouvelles techniques, la *méthodologie de vérification universelle (Universal Verification Methodology) (UVM)* et la *méthodologie de vérification ouverte (Open Verification Methodology) (OVM)*. Il dit aussi être capable de créer facilement des bancs de test UVM/OVM professionnels.

Un bref résumé de ces trois outils est présenté ci-dessous.

VUnit :

- Langage de test : Python
- Langage hardware : VHDL, SystemVerilog
- Systèmes d'exploitation supportés : Windows, Linux, macOS
- Spécialisation : spécifique aux tests unitaires du système
- Gratuit et open source

cocotb :

- Langage de test : Python
- Langage hardware : VHDL, SystemVerilog
- Systèmes d'exploitation supportés : Windows, Linux, macOS
- Spécialisation : banc de test pour des tests globaux du système
- Gratuit et open source

HDS :

- Langage de test : VHDL, SystemVerilog
- Langage hardware : VHDL, SystemVerilog

- Systèmes d'exploitation supportés : Windows, Linux
- Spécialisation : banc de test pour des tests globaux du système
- Payant avec une licence

2.5.2 Simulateurs

Pour exécuter les tests, différents simulateurs sont employés tels que GHDL [16], un outil totalement open source, ou encore ModelSim [17]. Le projet [18] basé sur **FPGA** a décidé d'utiliser ce premier simulateur en partie puisqu'il est plus adapté à l'intégration continue et de part sa gratuité, il n'y a aucun soucis de l'exécuter dans un nuage (ex : Docker). Martin Jeřábek [19] ajoute aussi que la vitesse de simulation de GHDL [16] est plus performante par rapport au second logiciel et qu'il ne peut pas y avoir de problème de licence. Il est en revanche uniquement valable pour le langage **VHDL**. Contrairement à l'outil ModelSim [17], aussi très populaire, il est commercial. Il est relativement lourd mais permet aussi la visualisation graphique des signaux générés. Il est valable pour le langage **VHDL** ainsi que Verilog. Il est à souligner que uniquement ModelSim [17] offre la possibilité d'afficher les signaux de manière graphique. La documentation officielle de GHDL [16] affirme ne pas avoir de visualisateur et que ce logiciel génère uniquement des fichiers qui peuvent être de type VCD, FST ou encore GHW. Néanmoins, Il existe un outil de visualisation, nommé GTKWave [20], compatible selon la documentation GHDL [16]. L'article [21] a pu tester cette combinaison avec succès. Ces scientifiques ont dressé un organigramme (figure 2.3) montrant la procédure générale en plus de leurs commandes.

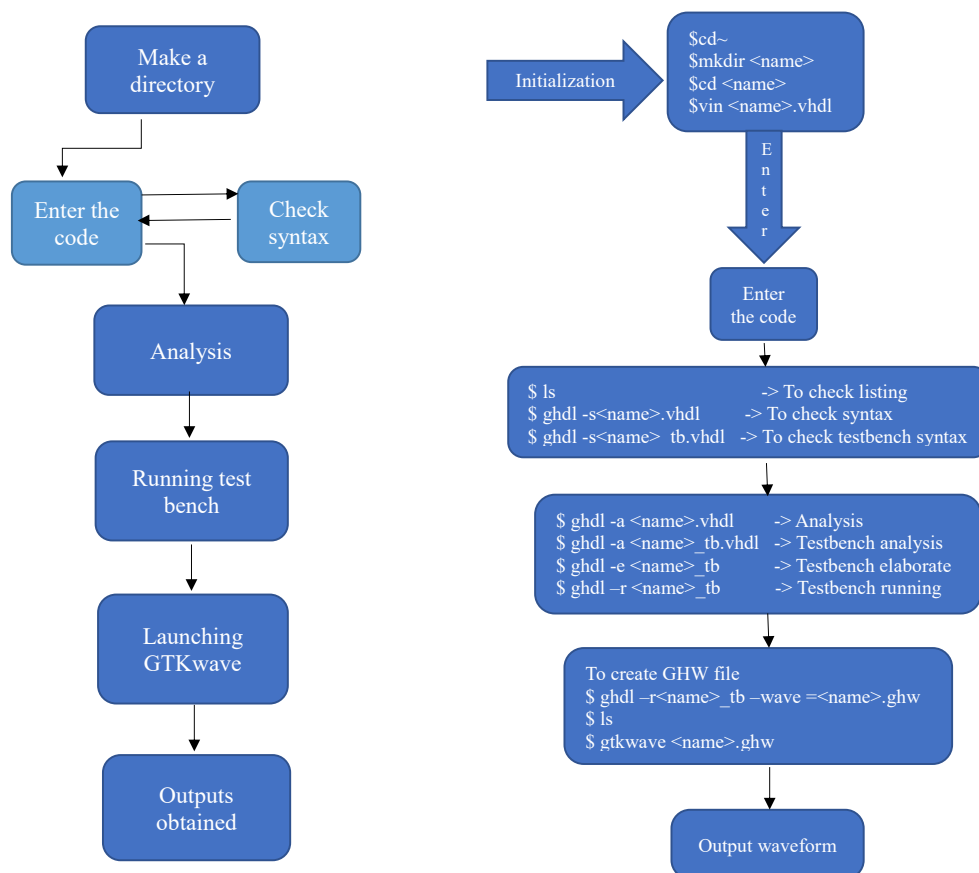


Figure 2.3 Organigrammes montrant la procédure générale de simulation (à gauche) et les commandes à utiliser à chaque étape (à droite).

Un bref résumé de ces deux outils est présenté ci-dessous.

GHDL :

- Gratuit et open source
- Langage hardware : VHDL
- Outils de test compatibles : VUnit, cocotb
- Systèmes d'exploitation supportés : Windows, Linux, macOS
- Visualisation graphique : non
- Traduction direct du langage VHDL en langage machine (gain de vitesse)

ModelSim :

- Payant avec une licence
- Langage hardware : VHDL, Verilog
- Outils de test compatibles : VUnit, cocotb
- Systèmes d'exploitation supportés : Windows, Linux
- Visualisation graphique : oui
- Couverture du code intégré

Ces deux simulateurs présentés sont compatibles avec les deux outils de test, VUnit [13] et cocotb [14]. De plus, ils peuvent fonctionner sous tous les systèmes d'exploitation, soit Windows, Linux et macOS, hormis macOS pour ModelSim [17].

2.5.3 Compléments

Alberto Martínez Cuesta [12] explique qu'il utilise, en complément de GHDL [16], un programme de couverture de code. Il s'agit de Gcov [22] qui mesure la fréquence d'exécution de chaque ligne de code. Cette technique permet de souligner les parties non testées et de mieux comprendre le fonctionnement du code. Puisqu'il ne comprend pas d'interface graphique, le programme Lcov [23] collecte ces données traitées pour les placer sur un graphique et les rendre plus lisibles.

2.6 Synthèse et implémentation

La formation au langage VHDL [24] qu'offre la Haute école d'ingénierie du canton de Vaud liste différents outils d'[automatisation de la conception électronique \(Electronic Design Automation\)](#) (EDA) pour la synthèse et l'implémentation. Elle parle de l'utilisation de ISE Design Suite [25], appartenant à Xilinx [3]. Depuis 2012, ce logiciel a cédé sa place au programme Vivado ML [26] réservé aux [FPGAs](#) de dernières générations. L'article [11] a pu intégrer celui-ci avec succès dans un [pipeline](#) de tests automatisés. Bien qu'il s'agisse uniquement de programmes commerciaux, il existe petit à petit d'autres synthétiseurs open source. Cette rubrique [21] explique que l'option synthèse est encore immature et en phase expérimentale. Cependant, si dans un futur proche cette option est développée, elle deviendra l'un des points forts des outils open source, la compatibilité avec tout [FPGA](#) dont la bibliothèque est disponible. GHDL [16] est l'exemple typique d'un outil libre en phase de devenir un synthétiseur. Il est clairement indiqué dans leur documentation [16] qu'ils sont en pleine période de test.

Cependant, l'implémentation qui s'occupe du placement et du routage reste une étape propre à chaque vendeur d'[FPGA](#) comme l'indique le professeur Etienne Messerli [\[24\]](#). Les outils spécifiques au fabricant Xilinx [\[3\]](#) sont identiques aux synthétiseurs, ISE Design Suite [\[25\]](#) et Vivado ML [\[26\]](#). Les [FPGAs](#) développés par Intel ont besoin de l'outil Quartus Prime [\[27\]](#).

Il est à noter que tous ces logiciels fournissent des commandes [Tcl](#) qui permettent de les incorporer dans le [CD](#). Elles sont définies dans chacune des documentations tels que celles de ISE Design Suite [\[28\]](#), Vivado ML [\[29\]](#) et Quartus Prime [\[30\]](#).

3 | Conception

La conception montre en premier lieu la vue d'ensemble des différentes parties qui constituent l'automatisation du [pipeline](#). Elle permet ensuite de tester la mise en place d'un outil en particulier. Un choix de certains programmes est aussi réalisé dans ce chapitre.

Table des matières

3.1	GitLab Runner	16
3.1.1	Vue d'ensemble	16
3.1.2	Evaluation de GitLab Runner	17
3.2	Pipeline	20
3.3	Flux de travail CI	22
3.3.1	Génération automatique de fichiers VHDL	22
3.3.2	Simulation et analyse automatique de bancs de test VHDL	24
3.4	Flux de travail CD	26
3.4.1	Synthèse automatique	26

3.1 GitLab Runner

Il est tout d'abord important de commencer par une étape de compréhension de l'application GitLab Runner [2]. Celle-ci va permettre de mieux appréhender l'évaluation qui s'en suivra, et de même l'implémentation de ce programme.

3.1.1 Vue d'ensemble

Un schéma de fonctionnement (figure 3.1) a été élaboré dans le but de mieux visualiser l'explication qui suit.

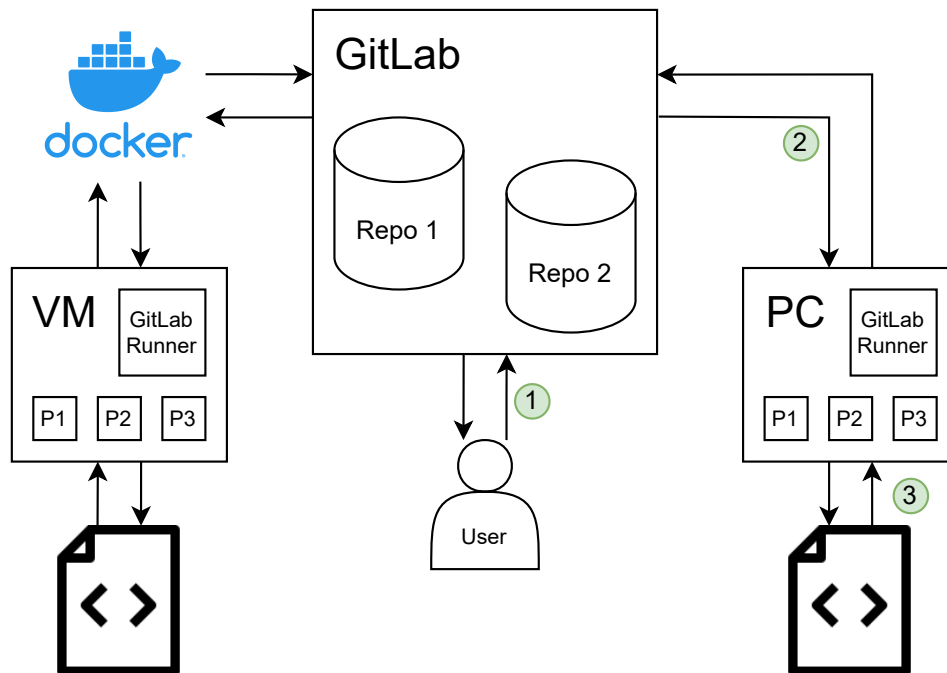


Figure 3.1 Schéma de fonctionnement avec GitLab Runner

Le développeur programme sa conception matérielle sur sa machine personnelle. Il communique ensuite avec la plateforme d'hébergement GitLab [1] lorsqu'il souhaite par exemple construire et tester son travail (point 1 de la figure 3.1). Afin d'effectuer ces exemples d'étape automatiquement, un service doit pouvoir gérer les programmes de construction et de test (cases P1, P2 et P3 de la figure 3.1) sur un environnement externe (une machine physique ou virtuelle). Ce service est fourni par l'outil GitLab Runner (case GitLab Runner de la figure 3.1). Les logiciels de développement vont s'exécuter en allant chercher le travail du développeur (point 2 de la figure 3.1). Ils vont travailler différents fichiers pour obtenir le résultat désiré (point 3 de la figure 3.1).

Ces services peuvent fonctionner dans des environnements distincts. Une bonne partie des utilisateurs travaille avec des environnements isolés appelés conteneurs. Chacun des conteneurs dispose d'une image virtuelle qui contient un programme dédié (programme P1 de la partie VM de la figure 3.1 par exemple). Le rassemblement de tous ces conteneurs forme une machine virtuelle, appelée "virtual machine" en anglais (partie VM de la figure 3.1). Ils peuvent être gérés par un seul outil (outil Docker [31] par exemple à la figure 3.1). Il est aussi possible d'aménager un environnement physique contenant toutes les applications indispensables sur une seule machine (partie PC de la figure 3.1).

Plusieurs avantages s'offrent à un environnement virtuel. Il est généralement léger. Il est facilement mis en place, duplicable et désinstallable. Il possède une très bonne portabilité. Il se porte facilement à des développements utilisant peu de moyens. A contrario, pour des développements plus conséquents, il ne va pas pouvoir accueillir des programmes demandant beaucoup de ressources. De plus, à chaque préparation d'une machine virtuelle, une installation de tous les logiciels est faite et durera plus ou moins longtemps selon la taille de ceux-ci. Il ne permet non plus pas la communication avec le monde physique pour des tests matériels par exemple.

Dans le cadre de ce travail, en raison des grandes ressources que demandent certaines applications de conception matérielle, l'implémentation du service qu'offre GitLab Runner est réalisée sur une machine physique. Cependant, pour procéder à une évaluation de ce service, un environnement virtuel est employé. Un programme appelé Docker [31] va permettre de disposer les conteneurs essentiels.

3.1.2 Evaluation de GitLab Runner

Une évaluation du fonctionnement de l'outil GitLab Runner est importante puisque l'automatisation complète du [pipeline](#) dans ce projet est gérée depuis son service. Il doit pouvoir oeuvrer correctement avec la plateforme d'hébergement GitLab [1] se trouvant sur les serveurs de la [HEI](#). Afin de l'évaluer sans difficulté, le service est installé dans un conteneur Docker [31]. Il s'agit donc d'un test rapide et efficace.

3.1.2.1 Installation

Il est d'abord nécessaire de créer un répertoire sur le référentiel. Par exemple, l'ajout de deux fichiers est suffisant comme le montre la figure 3.2. Le fichier nommé *file.txt* contient simplement un texte.

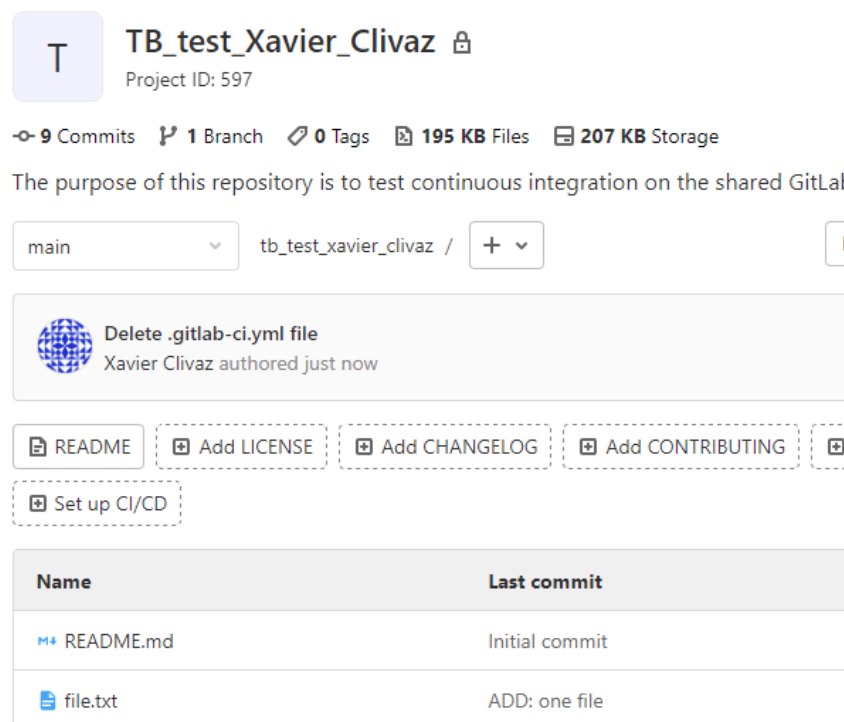


Figure 3.2 Répertoire de test pour le service de GitLab Runner

Par après, l'installation du logiciel Docker [31] est requise sur une machine physique. Une image pour GitLab Runner est déjà préconfigurée par GitLab. Existante sur le hub de Docker [32], elle peut directement être clônée sur la machine physique. La configuration du service est ensuite obligatoire pour effectuer la liaison avec le répertoire en question. Il reste à exécuter l'image pour pouvoir l'exploiter. Son état prêt, visible sur le référentiel, s'apparente à la figure 3.3. Les détails de l'installation se trouvent en annexe A.

Available specific runners

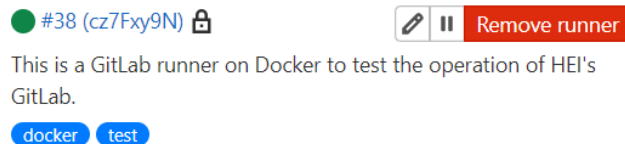


Figure 3.3 Service de GitLab Runner opérationnel

Afin que le [pipeline](#) puisse se lancer à l'aide du service, un script doit transmettre les informations et la démarche à suivre comme présenté au code 3.1.

```
1  image: alpine:latest
2
3  stages:           # List of stages for jobs, and their order of execution
4    - test
5
6  test-job:         # This job runs in the test stage, which runs first.
7    stage: test
8    tags:
9      - docker
10     - test
11  script:
12    - echo "Starting the pipeline"
13    - ls
14    - cat file.txt
```

Code source 3.1 Configuration du fichier `.gitlab-ci.yml`

Ce fichier spécifique à GitLab Runner, appelé `.gitlab-ci.yml`, indique l'image employée pour l'exécution des différentes tâches (ligne 1 du code 3.1). Il ordonne par après les étapes (ligne 3 du code 3.1). Uniquement un processus se présente dans cet exemple. Dans ce processus, des étiquettes, appelées "tags" en anglais, sont mentionnées pour faire appel à un service spécifique de GitLab Runner (lignes 8 à 10 du code 3.1). Les commandes qui doivent être lancées se retrouvent dans la tâche en question et plus précisément dans `script` (lignes 12 à 14 du code 3.1). Cet exemple va simplement demander de lister les fichiers du répertoire et de lire le contenu du fichier `file.txt`.

3.1.2.2 Test

Le premier test a pu être lancé. Il est noté dans le coin en haut à gauche de la figure 3.4 que le passage a été réalisé avec succès (message en vert). Une console affiche le cheminement du [pipeline](#). La commande "ls" a listé correctement les deux fichiers présents. Ainsi, la commande de lecture de fichier `cat` a été autorisée à se lancer. Le contenu "Hello world" du fichier `file.txt` est apparu sans erreur.

Smart Process Lab > ... > 2022-fpga_cicd > TB_test_Xavier_Clivaz > Jobs > #9784

passed Job #9784 triggered 3 minutes ago by Xavier Clivaz

```



1 Running with gitlab-runner 15.0.0 (febb2a09)
2   on This is a GitLab runner on Docker to test the operation of HEI's GitLab. PeubjFjZ
3   ✓ Preparing the "docker" executor 00:04
4     Using Docker executor with image alpine:latest ...
5     Pulling docker image alpine:latest ...
6     Using docker image sha256:e66264b98777e12192600bf9b4d663655c98a090072e1bab49e233d7531d1294 for alpine:latest with digest alpine@sha256:686d8c9dfa6f3ccfc8230bc3178d23f84eeaf7e457f36f271ab1acc53015037c ...
7   ✓ Preparing environment 00:01
8     Running on runner-peubjFjZ-project-597-concurrent-0 via ff8a20447758...
9   ✓ Getting source from Git repository 00:02
10     Fetching changes with git depth set to 50...
11     Reinitialized existing Git repository in /builds/SPL/bachelorthesis/2022-fpga_cicd/tb_test_xavier_clivaz/.git/
12     Checking out bd8c2683 as main...
13     Skipping Git submodules setup
14   ✓ Executing "step_script" stage of the job script 00:01
15     Using docker image sha256:e66264b98777e12192600bf9b4d663655c98a090072e1bab49e233d7531d1294 for alpine:latest with digest alpine@sha256:686d8c9dfa6f3ccfc8230bc3178d23f84eeaf7e457f36f271ab1acc53015037c ...
16     $ echo "Starting the pipeline"
17     Starting the pipeline
18     $ ls
19     README.md
20     file.txt
21     $ cat file.txt
22     Hello world
23   ✓ Cleaning up project directory and file based variables 00:01
24   Job succeeded

```

Figure 3.4 Passage du *pipeline* de test avec succès

Pour vérifier son total fonctionnement, une erreur a été glissée dans le répertoire. Le fichier *file.txt* a été renommé en *file1.txt*. Lors du test, une erreur a été détectée comme le montre la figure 3.5. Effectivement, la liste des fichiers s'est bien passée alors que la demande de lecture du fichier n'a pas pu être concrétisée en raison du fichier *file.txt* inexistant. Le parcours dans le *pipeline* a donc dû être interrompu.

Smart Process Lab > ... > 2022-fpga_cicd > TB_test_Xavier_Clivaz > Jobs > #9785

 Job #9785 triggered 1 minute ago by  Xavier Clivaz

```
1 Running with gitlab-runner 15.0.0 (febb2a09)
2   on This is a GitLab runner on Docker to test the operation of HEI's GitLab. PeubjFjZ
3   ✓ Preparing the "docker" executor 00:05
4     Using Docker executor with image alpine:latest ...
5     Pulling docker image alpine:latest ...
6     Using docker image sha256:e66264b98777e12192600bf9b4d663655c98a090072e1bab49e233d7531d1294 for alpine:latest with digest alpine@sha256:686d8c9dfa6f3ccfc8230bc3178d23f84eeaf7e457f36f271ab1acc53015037c ...
7   ✓ Preparing environment 00:01
8     Running on runner-peubjffz-project-597-concurrent-0 via ff8a20447758...
9   ✓ Getting source from Git repository 00:01
10    Fetching changes with git depth set to 50...
11    Reinitialized existing Git repository in /builds/SPL/bachelorthesis/2022-fpga_cicd/tb_test_xavier_clivaz/.git/
12    Checking out 1e3d4555 as main...
13    Skipping Git submodules setup
14  ✓ Executing "step_script" stage of the job script 00:01
15    Using docker image sha256:e66264b98777e12192600bf9b4d663655c98a090072e1bab49e233d7531d1294 for alpine:latest with digest alpine@sha256:686d8c9dfa6f3ccfc8230bc3178d23f84eeaf7e457f36f271ab1acc53015037c ...
16    $ echo "Starting the pipeline"
17    Starting the pipeline
18    $ ls
19    README.md
20    file1.txt
21    $ cat file.txt
22    cat: can't open 'file.txt': No such file or directory
23  ✓ Cleaning up project directory and file based variables 00:01
24  ERROR: Job failed: exit code 1
```

Figure 3.5 Passage du *pipeline* de test refusé

3.1.2.3 Conclusion

En conclusion de cette évaluation, cette intégration continue simple et efficace a permis de valider le fonctionnement de l'outil GitLab Runner sur le GitLab de la [HEI](#) à partir d'une image. Le référentiel est aussi opérationnel pour la suite du travail qui concerne la mise en place d'un service, non sur une image virtuelle, mais sur une machine physique.

3.2 Pipeline

Le *pipeline* est une section très importante. Il va mener la totalité des processus. Une conception de celui-ci a été réalisée graphiquement pour bien comprendre le déroulement des *workflows* (figure 3.6).

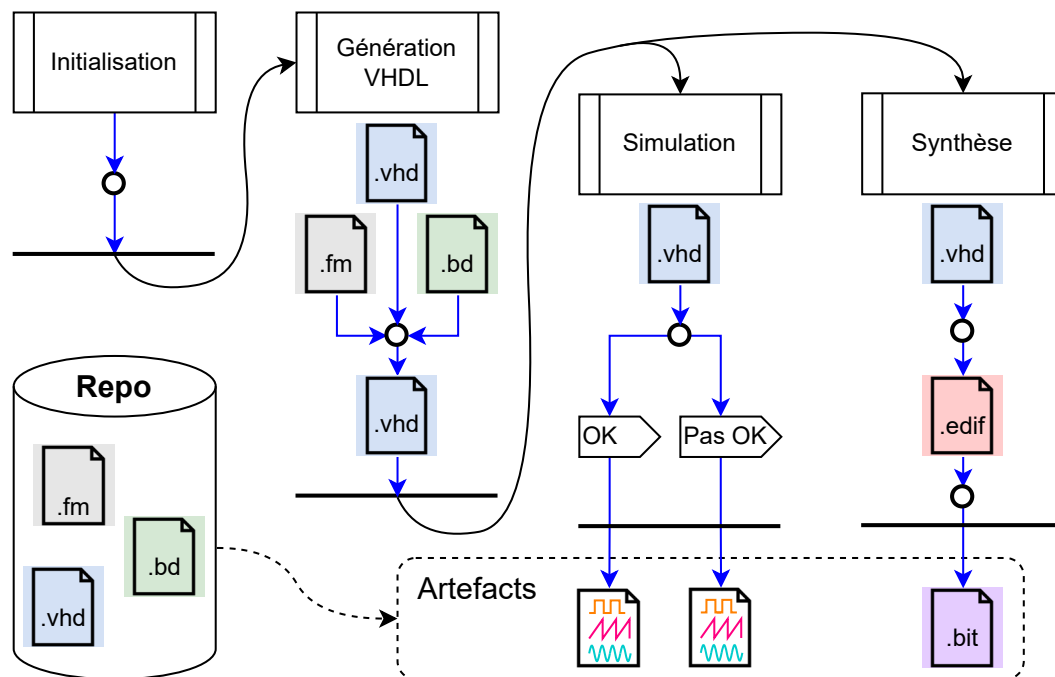


Figure 3.6 Pipeline des processus de développement matériel

L'idée première est de passer par une phase d'initialisation où plusieurs commandes peuvent être exécutées afin d'initialiser certaines choses dont auront besoin les tâches futures. Il peut aussi y avoir une vérification des programmes installés sur la machine où est installé le programme GitLab Runner.

La tâche suivante concerne la génération de fichiers **VHDL**. Le programmeur étant libre du langage pour développer sa conception, soit en machine d'état (.fm), soit en diagramme en bloc (.bd), soit en **VHDL** (.vhd), seul le **VHDL** peut être employé pour la simulation et la synthèse. Les fichiers sources (.vhd, .fm, .bd) sont récupérés dans le répertoire du projet pour ensuite être convertis. Même les fichiers .vhd passent par la génération mais ils ne seront pas affectés.

Deux **workflows** ont ensuite lieu d'être lancés si la génération s'est déroulée avec succès. Un d'entre eux est la simulation qui saisit les fichiers .vhd générés pour procéder à plusieurs tests à l'aide d'un banc de test. Une réponse totale va être rapportée pour savoir si le design matériel répond aux attentes. Dans tous les cas d'une simulation, les comportements de tous les signaux numériques sont stockés dans un **artefact**. Ceci permet par la suite de pouvoir les visualiser graphiquement de manière manuelle.

La synthèse est lancée pour créer une **netlist** (.edif) toujours à partir des fichiers **VHDL** générés. Dès lors, l'implémentation est effectuée pour obtenir un fichier **bitstream** (.bit). Ce fichier est sauvé en tant qu'**artefact** pour être déployé sur la carte de développement manuellement.

A partir du répertoire du projet, un accès à tous les **artefacts** est possible. Ils sont stockés en dehors de celui-ci.

3.3 Flux de travail CI

Le flux de travail **CI** concerne la partie des développeurs dans la méthode **DevOps** (figure 1.2). La génération de fichiers **VHDL** appartient à la construction du développement matériel. La simulation est présente dans le but de tester la conception dans son intégralité.

3.3.1 Génération automatique de fichiers **VHDL**

Ce processus de génération de fichiers **VHDL** est le point important pour la suite des opérations. Il met en place un **HDL** compatible à la simulation ainsi qu'à la synthèse.

3.3.1.1 Vue d'ensemble

Une vue d'ensemble a été réalisée dans le but de mieux comprendre la communication passant à travers le référentiel partagé et la machine physique contenant le service de GitLab Runner (figure 3.7).

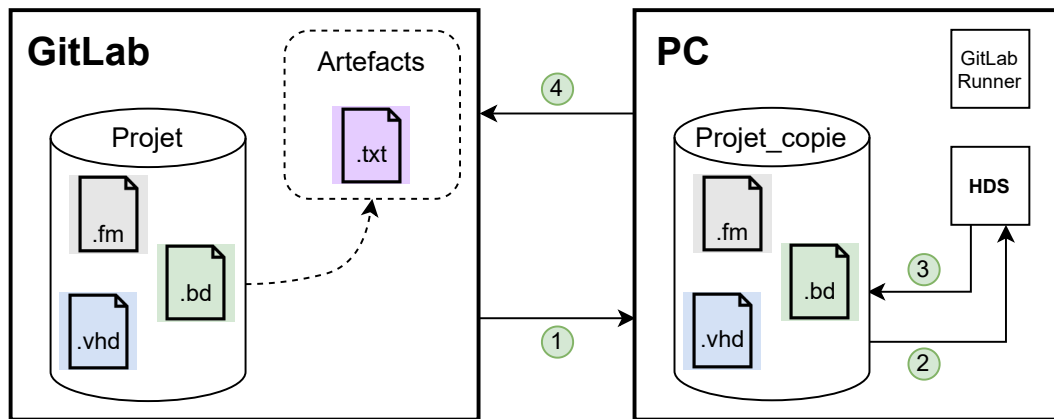


Figure 3.7 Vue d'ensemble du *workflow* génération **VHDL**

Avant que cette tâche de génération ne se lance, une copie du répertoire de projet est faite sur la machine physique (point 1 de la figure 3.7). Dès lors, le programme **HDS** peut commencer son travail à partir des fichiers de design contenus dans la copie (point 2 de la figure 3.7). Lorsque l'exécution est totalement réalisée, tous les fichiers **VHDL** générés sont sauvés dans le répertoire local de la machine (point 3 de la figure 3.7). Afin d'opérer une analyse visuelle de cette étape, un fichier texte contenant les informations de la génération est poussé dans les **artefacts** du projet sur la plateforme GitLab (point 4 de la figure 3.7).

3.3.1.2 Flux de travail

Pour aller plus dans les détails de cette sous-section, un schéma fonctionnel a été dressé à la figure 3.8. Il permet de visualiser facilement la conception de ce *workflow*. Il est placé en série aux autres *workflows* dans le **pipeline**. Son entrée est liée à la sortie de la première tâche d'initialisation du **pipeline**. Sa sortie se ramène à l'entrée de la tâche de simulation et de synthèse.

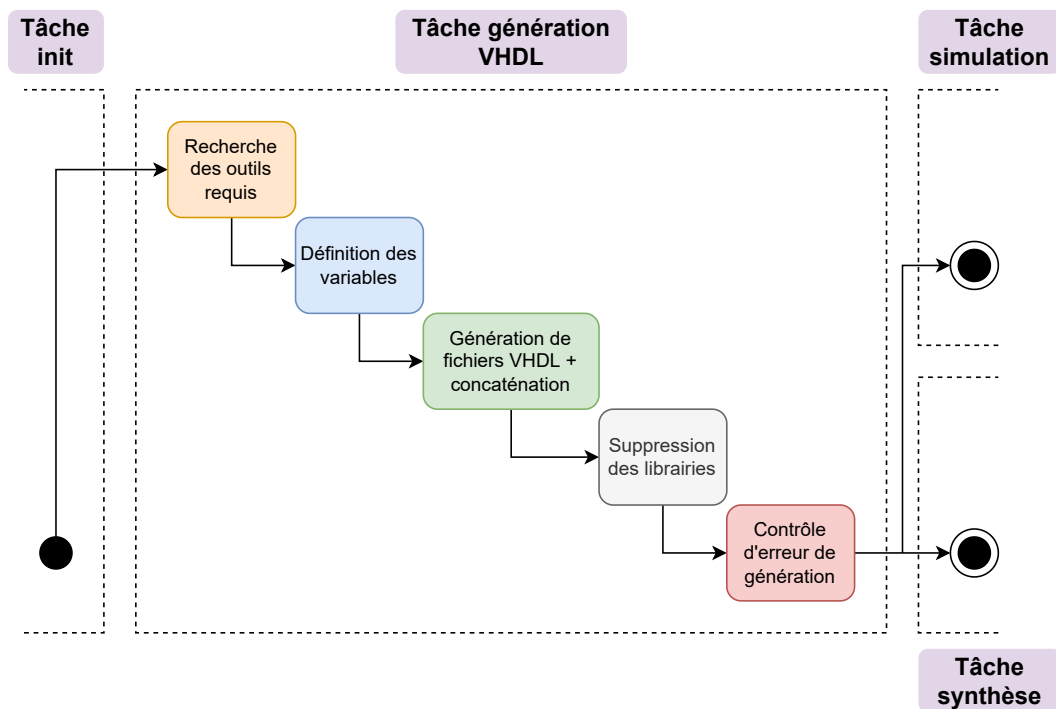


Figure 3.8 Schéma fonctionnel du *workflow* génération VHDL

Dans son intérieur, plusieurs points doivent être passés. Puisque GitLab Runner peut être installé sur n'importe quelle machine physique, il est primordial de vérifier en premier lieu les chemins d'accès aux différents outils requis à cette tâche. Par exemple, [HDS](#) est un logiciel essentiel pour ce *workflow*.

En raison de la portabilité de ce *pipeline*, il faut indiquer les variables d'environnement pour certains outils. Elles sont distinctes dans le cas où le projet et/ou les emplacements de certains paramètres sont différents. Une définition de toutes les variables locales est réalisée dans le but de modifier rapidement celles-ci uniquement en cas de nécessité.

Seulement après la configuration, la commande d'exécution de la génération peut être démarrée. Elle génère les fichiers [VHDL](#) de toutes les librairies. Elle s'en suit par la concaténation de ces fichiers générés pour en obtenir plus qu'un seul en sortie. La concaténation est utile pour la simulation et la synthèse. Effectivement, puisque ces fichiers générés appartiennent souvent à plusieurs librairies, il vaut mieux les réunir dans une seule librairie que de toutes les importer. Ceci facilite grandement le paramétrage des deux *workflows* futurs.

La concaténation ne fait que mettre ensemble tous les fichiers sans rien y modifier. Il reste donc toujours les lignes de code servant à importer les anciennes librairies qui sont maintenant obsolètes et inconnues. Une suppression de toutes celles-ci est alors menée.

Une vérification finale de la génération est opérée. Elle permet de valider ou non la tâche. Elle détermine ainsi si le *pipeline* doit être interrompu ou pas. Dans le cas d'un arrêt, les tâches futures sont annulées.

3.3.2 Simulation et analyse automatique de bancs de test VHDL

La simulation est l'étape clé qui indique l'état du fonctionnement global de la conception matérielle. Cette information est cruciale pour savoir si la carte de développement [FPGA](#) se comportera correctement.

3.3.2.1 Vue d'ensemble

La figure 3.9 illustre le fonctionnement général de la simulation.

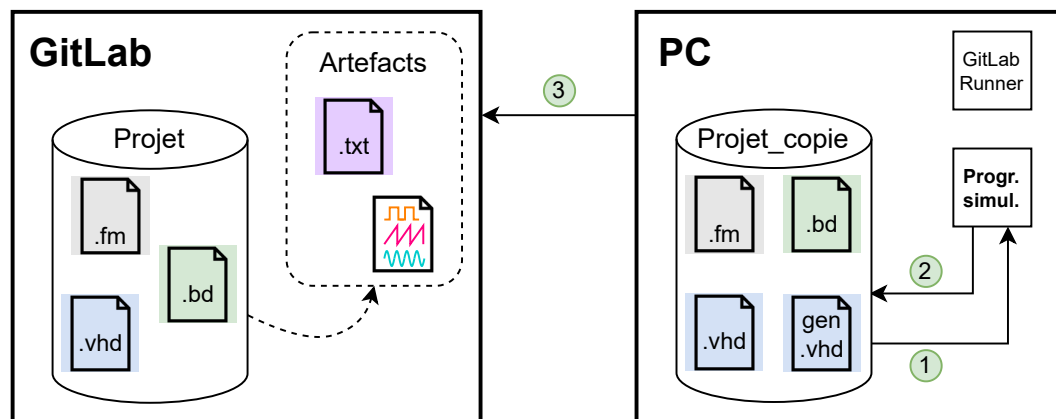


Figure 3.9 Vue d'ensemble du *workflow* simulation

Après que la génération des fichiers [VHDL](#) se soit déroulée avec succès, ces fichiers restent tels quels dans la copie du répertoire lors de l'étape de simulation. Le programme de simulation peut donc démarrer en allant récupérer ces fichiers (point 1 de la figure 3.9). Une fois la simulation terminée, le logiciel va fournir un fichier contenant les signaux stimulés pour les visualiser graphiquement ultérieurement si nécessaire (point 2 de la figure 3.9). Celui-ci va être transmis au référentiel partagé en tant qu'[artefact](#) pour que le développeur y ait accès. De plus, plusieurs journaux sur le déroulement de la simulation sont aussi enregistrés dans les [artefacts](#).

3.3.2.2 Flux de travail

Cette partie de la conception montre plus en détail les étapes de simulation et d'analyse à partir d'un banc de test. Un schéma fonctionnel est présent à la figure 3.10 pour illustrer ce [workflow](#). Dans le [pipeline](#), cette tâche est placée après la génération de fichiers [VHDL](#).

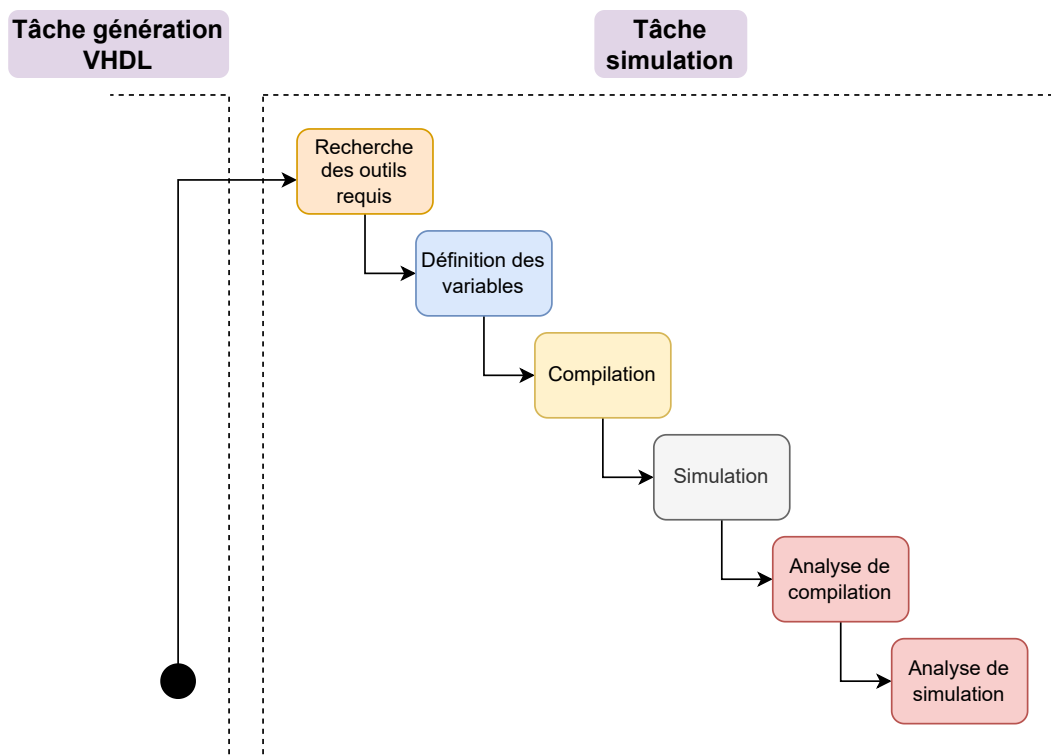


Figure 3.10 Schéma fonctionnel du *workflow simulation*

Les deux premières étapes sont semblables à la sous-section 3.3.1.2. Tout d'abord un passage concerne la recherche des outils requis sur la machine physique du GitLab Runner et ainsi la définition des variables locales et d'environnement.

L'étape suivante compile le fichier **VHDL** concaténé précédemment afin qu'il soit compréhensible par le logiciel de simulation.

La simulation est la phase qui stimule la conception du projet. Des données vont découler de cette étape pour au final pouvoir être analysées. Ce moment est important pour savoir si le design fonctionne intégralement et correctement.

Afin de s'assurer que tout se soit passé correctement, une analyse des données sortantes est effectuée pour la compilation et la simulation.

3.3.2.3 Choix des outils de simulation

Dans ce travail, le choix des outils permettant de simuler la conception matérielle est libre. La **HEI** utilise spécialement deux programmes dans le cadre de sa formation pour les futurs ingénieurs. **HDS** est le moyen de développer des bancs de test facilement et directement en **HDL**. ModelSim [17] est l'outil phare pour simuler les bancs de test avec leur conception.

Ayant donc un meilleur support de ces logiciels, le choix s'est porté à ces deux derniers pour la réalisation de ce travail. De plus, le cahier des charges indique premièrement de développer un banc de test en **VHDL**, ce qui est parfaitement envisageable à l'aide d'**HDS**. cocotb [14] peut aussi permettre la création de banc de test mais uniquement en langage Python.

3.4 Flux de travail CD

Le flux de travail **CD** concerne la partie des opérateurs dans la méthode **DevOps** (figure 1.2). La synthèse qui génère le fichier **bitstream** s'approprie à la publication du développement matériel. Il s'agit de la dernière étape avant le déploiement sur la carte de développement **FPGA**.

3.4.1 Synthèse automatique

La synthèse permet de générer le fichier **bitstream** qui peut ensuite directement être programmé manuellement dans la carte de développement **FPGA**.

3.4.1.1 Vue d'ensemble

La vue d'ensemble de la figure 3.11 montre le fonctionnement entre la plateforme d'hébergement et la machine physique du GitLab Runner.

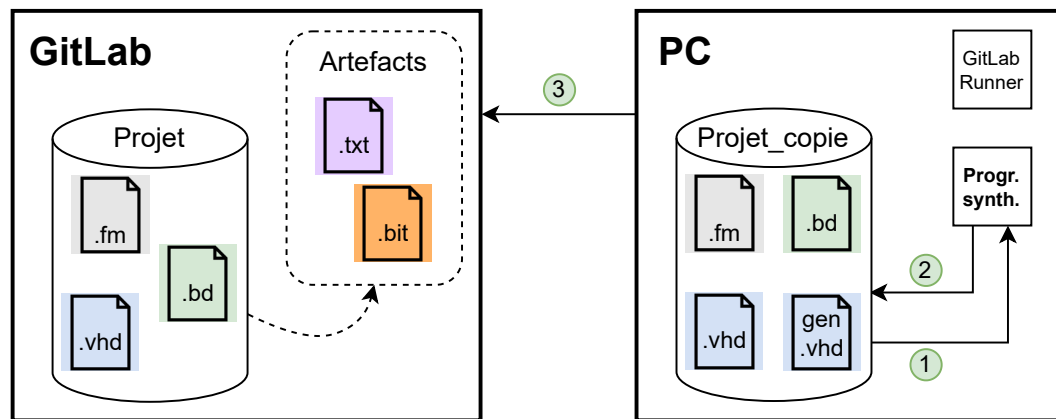


Figure 3.11 Vue d'ensemble du **workflow** synthèse

Il s'agit du même principe que pour la simulation. Si les fichiers **VHDL** ont été générés sans problème, le programme de synthèse va pouvoir récupérer le fichier concaténé (point 1 de la figure 3.11). Il synthétise le tout pour produire le fichier **bitstream** qu'il va stocker dans le répertoire local (point 2 de la figure 3.11). Un journal comportant les informations de synthèse est aussi sauvé. Ces deux fichiers sont poussés en dernier lieu dans les **artefacts** pour donner l'accès à l'opérateur (point 3 de la figure 3.11).

3.4.1.2 Flux de travail

Le **workflow** de synthèse est placé en sortie de la tâche de génération **VHDL**. Un schéma-bloc est représenté à la figure 3.12 pour présenter les détails de ce flux.

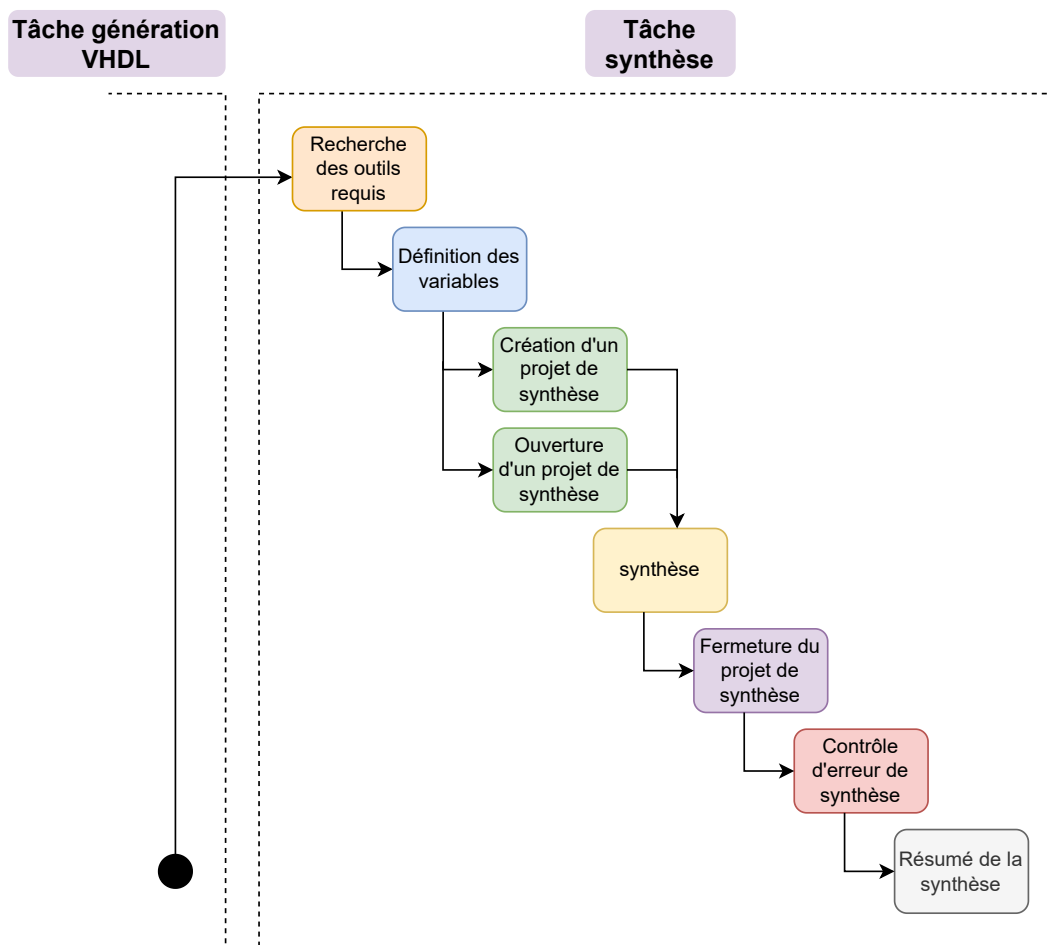


Figure 3.12 Schéma fonctionnel du *workflow* synthèse

Les deux premières tâches sont toujours identiques aux *workflows* précédents. Une recherche des outils nécessaires est faite en premier lieu. La suite concerne la définition des variables locales et d'environnement.

Avant le lancement de la synthèse, le programme nécessite un projet propre à lui puisqu'il doit contenir tous les paramètres tels que ceux de la carte *FPGA*. Dans le cas de ce travail, il est possible qu'un projet soit déjà existant dans le dépôt. Il suffit simplement de l'ouvrir. Si celui-ci n'est pas présent, la création d'un nouveau projet est réalisée.

La synthèse peut ensuite être exécutée. Plusieurs processus sont traités au cours de cette tâche. Les détails sont traités dans la partie implémentation 4. Celle-ci va générer le fichier de configuration propre à une *FPGA* du fabricant Xilinx [3].

Pour que la tâche se termine sans provoquer d'erreur, la fermeture du projet est recommandée.

Lorsque le programme a terminé de s'exécuter, un contrôle d'erreur est effectué. Il détermine le nombre d'erreur et d'attention de tous les processus de synthèse. Si une seule erreur est perçue, elle est déclarée et le *pipeline* s'interrompt.

Si tout s'est déroulé correctement, un sommaire des informations importantes de la synthèse est généré dans un fichier. Il permet de rendre facilement lisible ces données à l'opérateur. On peut retrouver par exemple le nombre de composants utilisés dans le circuit de la carte. Ces données renseignent afin d'améliorer le design en cas de besoin.

Après cette étape de synthèse, si la simulation est aussi terminée avec succès, le [pipeline](#) est terminé correctement.

3.4.1.3 Choix des outils de synthèse

Peu de choix s'offre aux outils de synthèse. Il est expliqué dans l'analyse [2](#) que ces logiciels sont généralement propres à chaque fabricant de carte [FPGA](#).

Dans ce travail, la carte [FPGA](#) utilisée est la Spartan-3E du fabricant Xilinx [\[3\]](#). Celui-ci possède actuellement deux programmes, ISE Design Suite [\[25\]](#) et Vivado ML [\[26\]](#). Seul ISE Design Suite est compatible avec cette carte employée, le choix se porte donc sur ce programme.

4 | Implémentation

Ce chapitre montre la démarche menée sur la réalisation du [pipeline](#) automatisé pour un développement matériel. L'implémentation a été réalisée avec les différents programmes choisis dans la conception [3](#). De plus, le [pipeline](#) a été conçu de manière à être le plus portable possible afin de l'utiliser dans une large gamme de développements matériels. Dans le cadre de ce travail, il a été construit autour d'un projet [EDA](#) de la [HEI](#).

Table des matières

4.1	GitLab Runner	30
4.1.1	Installation	30
4.1.2	Fonctionnement	32
4.2	Pipeline	34
4.2.1	Vue d'ensemble	34
4.2.2	Structure	35
4.2.3	Flux de travail	37
4.2.4	Configuration	41
4.3	Flux de travail CI	43
4.3.1	Génération automatique de fichiers VHDL	43
4.3.2	Simulation et analyse automatique de bancs de test VHDL	49
4.4	Flux de travail CD	51
4.4.1	Synthèse automatique	52

4.1 GitLab Runner

Dans le cadre de ce travail, le programme GitLab Runner doit se trouver sur un environnement physique. Les raisons sont évidentes. Les outils [FPGA](#) indispensables au développement matériel ne sont pas tous légers. Typiquement, le logiciel de synthèse nécessite une grande capacité de stockage pouvant franchir 20[GB]. Il n'est d'ailleurs pas recommandé de devoir les installer à chaque lancement du [pipeline](#) en raison de la durée. L'utilisation d'une image n'est alors pas convenable. Il s'agit dans ce travail d'installer ce service sur une machine physique comprenant déjà les applications avec Windows en tant que système d'exploitation. De plus, l'installation sur une machine rend possible le déploiement du programme directement sur la carte de développement [FPGA](#). Une analyse physique pourrait alors être opérée.

4.1.1 Installation

L'installation de GitLab Runner a été effectuée sur un ordinateur fixe de la [HEI](#) ayant Windows. Toutes les étapes mentionnées proviennent d'un guide d'installation [33] rédigé par GitLab [1]. Il est tout d'abord requis d'installer le gestionnaire de versions Git [34] ainsi que de télécharger GitLab Runner [33]. Il est demandé de créer un emplacement où le programme GitLab Runner va être installé. Son exécutable doit être déplacé dans le répertoire créé et renommé de préférence pour une meilleure lisibilité. Dans le cas de ce travail, la figure 4.1 montre l'emplacement du fichier qui est "C:\GitLab-Runner" avec le fichier binaire appelé "gitlab-runner.exe".

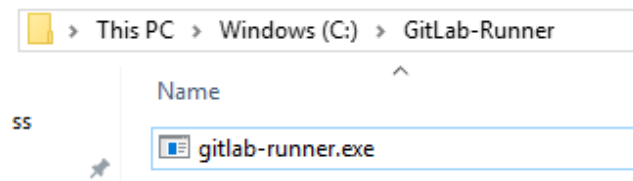


Figure 4.1 Préparation du fichier d'exécution de GitLab Runner

Il est demandé par la suite d'ouvrir une invite de commande en tant qu'administrateur et de se déplacer à l'emplacement du répertoire créé précédemment. L'enregistrement de GitLab Runner au répertoire en question sur le référentiel partagé peut ainsi être lancé par la commande suivante :

```
.\gitlab-runner.exe register
```

```

Administrator: Windows PowerShell
PS C:\windows\system32> cd C:\GitLab-Runner
PS C:\GitLab-Runner> ./gitlab-runner.exe register
Runtime platform arch=amd64 os=windows pid=12440 revision=febb2a09 version=15.0.0
Enter the GitLab instance URL (for example, https://gitlab.com/):
https://gitlab.hevs.ch/
Enter the registration token:
gt4xwqyz44CojxPrqgM8
Enter a description for the runner:
[WE7525]: This is the GitLab runner on Windows for the CI/CD of a hardware development.
Enter tags for the runner (comma-separated):
Windows,EDA
Enter optional maintenance note for the runner:

Registering runner... succeeded runner=gt4xwqyz
Enter an executor: custom, docker, docker-windows, docker-ssh, parallels, shell, ssh, virtualbox, docker+machine, docker-ssh+machine, kubernetes:
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
PS C:\GitLab-Runner>

```

Figure 4.2 Enregistrement du service de GitLab Runner dans un référentiel partagé

Les informations qui sont demandées d'être remplies sont similaires à celles expliquées dans l'annexe A. Pour une configuration sur Windows, l'exécuteur de commande est un [shell](#).

Lorsque ceci est rempli, il faut prêter attention au programme [shell](#) installé sur Windows qui s'appelle *PowerShell*. Effectivement, il existe plusieurs versions qui ne portent pas exactement le même nom pour les exécutables. La version 5.1 porte le nom de "powershell.exe" alors que la 6.0 "pwsh.exe". Le service de GitLab Runner est configuré pour lancer l'exécutable de dernière version. Afin de vérifier la version sur une machine, il suffit d'ouvrir une invite de commande et d'envoyer la commande suivante :

```

pwsh help

```

Si la commande est inconnue, la version présente est donc antérieure et l'exécutable est "powershell.exe". Il faut donc modifier un paramètre dans le fichier "config.toml" (figure 4.3) se trouvant dans le répertoire de GitLab Runner. La ligne 12 comporte le nom de l'exécutable présent sur la machine et celui-ci doit être en fonction de la version.

```

config.toml
1 concurrent = 1
2 check_interval = 0
3
4 [session_server]
5   session_timeout = 1800
6
7 [[runners]]
8   name = "This is the GitLab runner on Windows"
9   url = "https://gitlab.hevs.ch/"
10  token = "m5H-csc6ACiRPF9LxdTs"
11  executor = "shell"
12  shell = "powershell"
13  [runners.custom_build_dir]
14  [runners.cache]
15    [runners.cache.s3]
16    [runners.cache.gcs]
17    [runners.cache.azure]

```

Figure 4.3 Changement du nom de l'exécutable PowerShell dans le fichier config.toml

Chapitre 4. Implémentation

Pour terminer avec succès l'installation, la commande permettant l'installation du service de Gitlab Runner dans la machine est la suivante :

```
.\gitlab-runner.exe install --user yourUsername --password yourPassword
```

Attention, il faut être prudent sur quel utilisateur Windows le service est installé ! Un problème risque de survenir durant le [CI](#) si le service essaie d'exécuter des tâches devant être faites à l'aide d'un utilisateur différent. Il est donc fortement recommandé de spécifier l'utilisateur concerné pour l'installation. Il faut entrer l'utilisateur qui possède les différents outils de développement matériel. Si l'utilisateur n'est pas indiqué, l'installation va être effectuée par défaut sur l'utilisateur système. Si ceci est fait correctement, il est désormais possible d'aller vérifier que le service soit existant dans le référentiel en question comme le montre la [figure 4.4](#).

Available specific runners

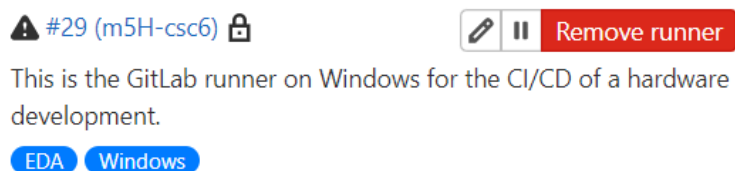


Figure 4.4 Vérification du service de GitLab Runner non opérationnel sur le référentiel partagé

Le service n'est actuellement pas fonctionnel puisqu'il n'a pas encore été lancé.

4.1.2 Fonctionnement

Il est désormais possible de démarrer le service dans PowerShell en tant qu'administrateur et tout en étant dans le répertoire du fichier exécutable. La commande est la suivante :

```
.\gitlab-runner.exe start
```

Il peut arriver au premier lancement du service qu'il rencontre un problème en raison d'un échec de connexion comme le montre la [figure 4.5](#).

```
PS C:\GitLab-Runner> .\gitlab-runner.exe start
Runtime platform arch=amd64 os=windows pid=2940 revision=febb2a09 version=15.0.0
FATAL: Failed to start gitlab-runner: The service did not start due to a logon failure.
```

Figure 4.5 Erreur de lancement du service de GitLab Runner

Pour y remédier, il suffit de se rendre dans le gestionnaire de services de Windows nommé "Services". L'accès peut se faire depuis la barre de recherche Windows. La fenêtre ressemble à la [figure 4.6](#).

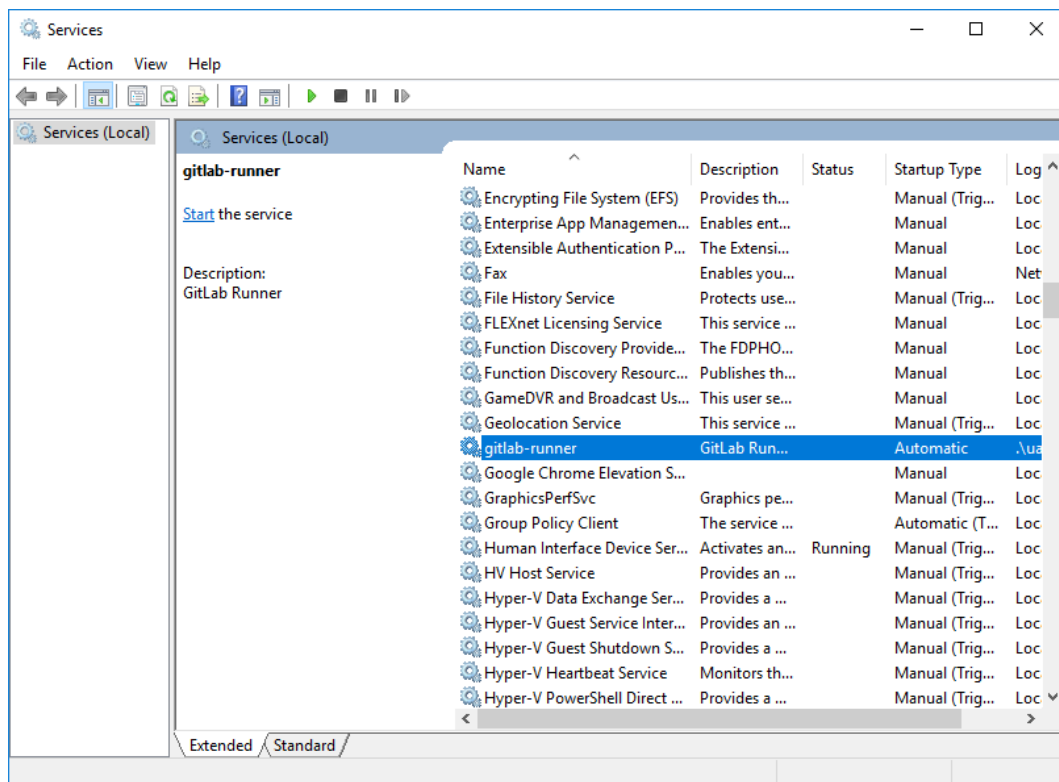


Figure 4.6 Fenêtre du gestionnaire de services Windows

Il faut ensuite chercher le service de GitLab Runner et accéder à ses propriétés en faisant un clic droit avec la souris sur le service et en sélectionnant "Propriétés". Il reste à se rendre dans l'onglet "Log On" et de rentrer à nouveau le nom d'utilisateur Windows ainsi que son mot de passe (figure 4.7). Si ceci est appliqué, le problème est censé être résolu. En redémarrant le service, il doit être exécuté avec succès.

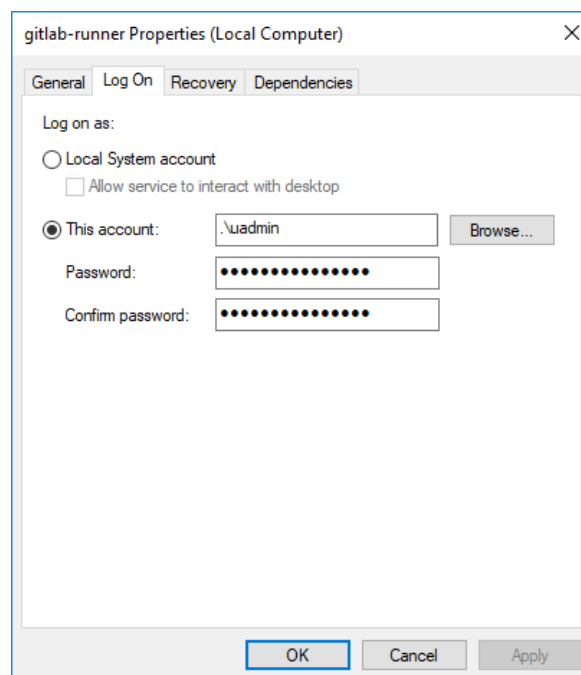


Figure 4.7 Fenêtre des propriétés utilisateurs du service de GitLab Runner

En allant vérifier dans le référentiel, une pastille verte doit être présente sur le service de GitLab Runner pour indiquer qu'il est opérationnel. La figure 4.8 le montre très bien.

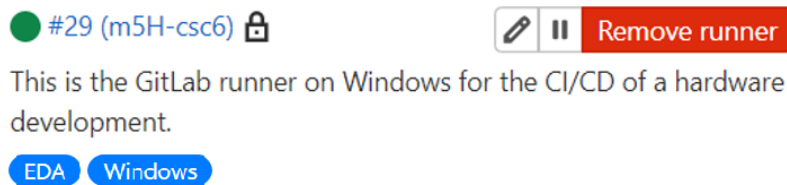


Figure 4.8 Vérification du service de GitLab Runner opérationnel sur le référentiel partagé

Il existe quelques autres commandes utiles à connaître pour le redémarrage et l'arrêt du service.

Le redémarrage :

```
.\gitlab-runner.exe restart
```

L'arrêt :

```
.\gitlab-runner.exe stop
```

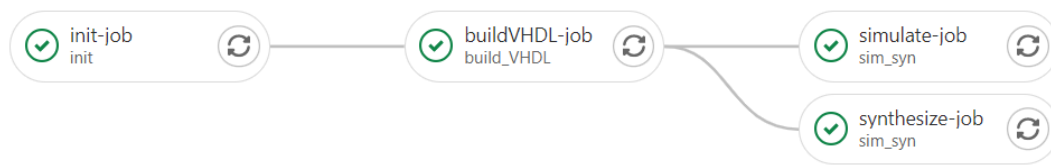
4.2 Pipeline

Dans ce travail, l'implémentation du **pipeline** est réalisée uniquement pour l'outil d'intégration GitLab-CI [5]. Tout son développement doit être implémenté dans un script spécifique à GitLab au langage **Yet Another Markup Language (YAML)**, toujours nommé `.gitlab-ci.yml`. Il s'agit de l'unique script qu'a à modifier le développeur pour chacun de ses projets afin de faire du **CI/CD** avec ses développements matériels.

En premier lieu, il est impératif d'installer un programme, appelé ActiveTcl, sur la machine du GitLab Runner. Il permet d'interpréter le langage **Tcl**. Il est requis puisque tous les outils de chaque **workflow** exécutent leurs actions uniquement à travers ce langage. Un guide sur son installation est disponible à l'annexe B.

4.2.1 Vue d'ensemble

Dans ce travail, le **pipeline** est constitué des quatre jobs expliqués dans la conception 3.2. Le mot job est un terme couramment utilisé dans GitLab pour définir un **workflow**. Ils sont répertoriés par étape comme le montre la figure 4.9. L'initialisation est la première phase ("init"), suivie de la génération **VHDL** ("build_VHDL") et enfin de l'étape ralliant la simulation et la synthèse ("sim_syn"). Cette dernière regroupe les deux **workflows** pour qu'ils puissent être exécutés en parallèle à condition qu'il y ait deux services de GitLab Runner.

Figure 4.9 Etapes du *pipeline*

Une certaine dépendance existe entre ces différents jobs qui est représentée par des lignes. La génération VHDL dépend de l'initialisation puisqu'elle a besoin du programme ActiveTcl [35] qui est vérifié dans l'initialisation. La simulation et la synthèse dépendent de la génération. Il est évident que si les fichiers VHDL ne sont pas générés, ces deux workflows ne pourront pas être lancés. Le but est donc d'indiquer que les jobs en ont besoin d'autres pour fonctionner. Dans le cas où un job ne se termine pas, les dépendants ne seront jamais exécutés.

La mise en parallèle de la simulation et la synthèse permet au développeur de ne lancer que la simulation, que la synthèse ou même les deux. Il est libre de choisir à sa convenance. De plus, en raison de leur indépendance l'un envers l'autre, la mise en série n'est pas optimale puisque le temps d'exécution du pipeline durerait bien plus longtemps.

4.2.2 Structure

Le script `.gitlab-ci.yml` exige une certaine structure pour être compréhensible par l'outil GitLab-CI [5]. Elle se compose essentiellement d'une partie contenant les étapes, appelé "stages" en anglais, (ligne 1 du code 4.1) et d'un bloc comportant les jobs (lignes 9 à 33 du code 4.1).

```

1  stages:                                # List of stages for jobs, and their order of execution
2    - init
3    - build_VHDL
4    - sim_syn
5
6  variables:
7    ...
8
9  init-job:                              # This job initializes the pipeline.
10   stage: init
11   ...
12
13  buildVHDL-job:                         # This job builds the VHDL files of HDS project.
14   stage: build_VHDL
15   ...
16
17  simulate-job:                          # This job simulates the HDS project.
18   stage: sim_syn
19   tags:
20     - ...
21   needs: [buildVHDL-job]
22   rules:
23     - ...
24   variables:
25     ...
26   script:
27     - ...
28   artifacts:
29     - ...
30

```

Chapitre 4. Implémentation

```
31     synthesize-job:           # This job builds the bitstream file of Xilinx.  
32         stage: sim_syn  
33         ...
```

Code source 4.1 *Script : .gitlab-ci.yml, structure incomplète*

La figure 4.1 représente la structure générale d'un [pipeline](#) utilisant l'implémentation de ce travail. Elle est incomplète hormis le job de simulation (lignes 17 à 29 du code 4.1) qui contient tous les mots clés en rouge utilisés dans ce script [YAML](#). Les mots clés sont donc expliqués plus bas à partir du job de simulation.

La partie "stages" est le premier point donnant l'ordre des étapes à réaliser (de haut en bas) comme illustré à la figure 4.9. Chacun des jobs contient donc une section nommée "stage" (lignes 10, 14, 18 et 32 du code 4.1) où le nom de l'étape correspondante est indiqué. Ce travail de diplôme inclut les quatres jobs de la figure 4.9, l'initialisation (ligne 9 du code 4.1), la génération [VHDL](#) (ligne 13 du code 4.1), la simulation (ligne 17 du code 4.1) et la synthèse (ligne 31 du code 4.1).

Pour tous les jobs, des étiquettes peuvent être définies, nommées "tags" en anglais (ligne 19 du code 4.1). Elles permettent de déterminer le service de GitLab Runner qui exécutera le job en question. Effectivement, plusieurs programmes GitLab Runner peuvent être en service sur différentes machines physiques ou virtuelles avec des outils distincts pour être spécifiques à certaines tâches.

Les dépendances expliquées dans la vue d'ensemble 4.2.1 sont liées à l'aide du mot clé "needs" (ligne 21 du code 4.1). Le job dépendant contient donc sa dépendance.

Des conditions peuvent être mises en place, appelées "rules" en anglais (ligne 22 du code 4.1). Elles permettent de ne lancer les jobs qu'en répondant à certaines conditions. Elles sont définies par le développeur.

Dans chaque job, des variables d'environnement locales ("variables") sont présentes (ligne 24 du code 4.1). Elles sont uniquement connues dans le job en question contrairement aux variables d'environnement globales (ligne 6 du code 4.1) connues dans tous les jobs. Les variables d'environnement facilitent grandement la tâche au développeur. Effectivement, tous les scripts utilisés dans les différents [workflows](#) exigent d'être totalement portables pour pouvoir être employés dans plusieurs projets matériels. Puisque beaucoup de variables locales aux scripts doivent être propres à chaque projet, le moyen simple et efficace de résoudre ce problème est le passage par les variables d'environnement. Elles ont donc l'avantage de paramétrer l'entièreté des scripts en étant modifiables uniquement et directement dans ce script [YAML](#). Les autres scripts n'ont alors jamais besoin d'être révisés.

Le coeur du job se situe dans l'onglet "scripts" (ligne 26 du code 4.1). C'est ici que les tâches de chaque travaux sont lancées.

La dernière section gère les [artefacts](#), nommé "artifacts" en anglais (ligne 28 du code 4.1). Il suffit simplement d'y indiquer le fichier avec son chemin qui doit être sauvé à la fin du job. Si le fichier mentionné n'est pas trouvable, le [pipeline](#) ne sera pas interrompu, seul l'[artefact](#) sera manquant.

4.2.3 Flux de travail

Il est question dans cette sous-section d'expliquer en détail les différents jobs du [pipeline](#).

4.2.3.1 Job d'initialisation

Ce premier job d'initialisation n'est pas très complexe. Puisqu'il est en début de chaîne, il ne possède pas de condition ni de dépendance. Il est démarré à chaque lancement du [pipeline](#). Il nécessite simplement du programme GitLab Runner installé sur une machine physique Windows ayant les outils pour du développement [EDA](#) (lignes 4 et 5 du code [4.2](#)). Il va exécuter deux actions différentes.

Il commence par contrôler que le programme ActiveTcl [\[35\]](#) installé au préalable soit bien présent. Cette vérification est capitale pour être certain que les scripts [Tcl](#) puissent être démarrés par la suite (lignes 14 à 19 du code [4.2](#)). Les scripts [Tcl](#) vont contenir toutes la actions que vont effectuer les logiciels de conception matérielle.

```

1      init-job:                # This job initializes the pipeline.
2      stage: init
3      tags:
4          - Windows
5          - EDA
6      variables:
7          PATH_TCLSH:        'C:\ActiveTcl\bin'
8      script:
9          #-----
10         # Control of tclsh program
11         #
12         - echo "Test to see if folder $PATH_TCLSH exists"
13         - >
14         if (Test-Path -Path $PATH_TCLSH) {
15             "tclsh found"
16         } else {
17             "ERROR, no valid installation of tclsh found"
18             exit 1
19         }
20         #-----
21         # Get submodules
22         #
23         - git submodule sync --recursive
24         - git submodule foreach --recursive git fetch
25         - git submodule update --init --recursive

```

Code source 4.2 Script : `.gitlab-ci.yml`, partie : job d'initialisation

La suite gère les sous-modules avec le contrôle de version Git [\[36\]](#). Les sous-modules sont des dépôts Git intégrés dans d'autres dépôts Git. Un exemple à la figure [4.10](#) (partie "GitLab") illustre un sous-module, "Scripts", inclus dans les projet Git "EDA1" et "EDA2". Les sous-modules sont généralement utilisés lorsque des répertoires sont communs à plusieurs projets. Pour ce travail, les scripts [Tcl](#) permettant la génération [VHDL](#), la simulation et la synthèse sont utilisables pour divers développements matériels de la [HEI](#). C'est pourquoi ils sont placés dans un répertoire Git appelé "Scripts".

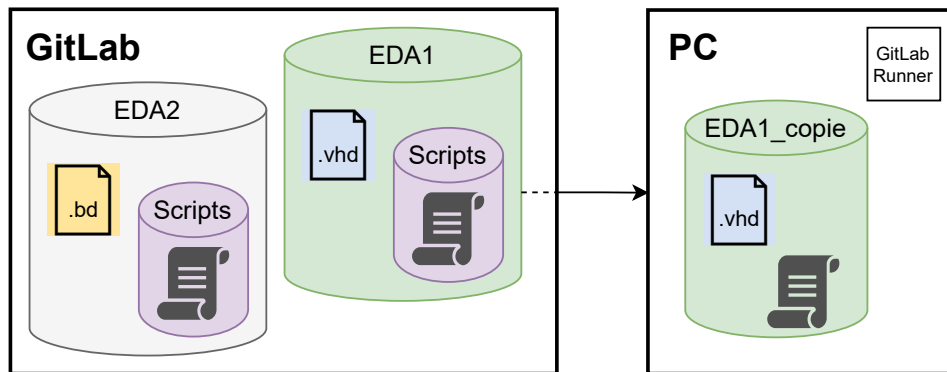


Figure 4.10 Illustration d'un sous-module

Pour qu'un répertoire local contienne les sous-modules, c'est un peu plus complexe. Premièrement, en clonant le répertoire du projet, les sous-modules ne sont pas inclus. Etant donné leur indépendance, ils doivent être clonés séparément mais toujours dans le répertoire local de la machine contenant GitLab Runner (partie "PC" de la figure 4.10).

Trois commandes sont donc appelées pour réaliser ce clonage séparé (lignes 23 à 25 du code 4.2). La première consiste à synchroniser récursivement les sous-modules du projet. Une fois terminé, la phase de clonage peut démarrer pour chaque sous-module toujours de manière récursive. Il reste à les initialiser et les mettre à jour aussi récursivement. Récursivement signifie passer à travers tous les sous-modules imbriqués. Attention à ce que les sous-modules soient à jour dans le répertoire du projet par rapport aux parents des sous-modules ! Ces trois commandes permettent de cloner uniquement les versions du répertoire du projet.

En annexe C, une explication parle de l'ajout de sous-modules dans un référentiel avec quelques informations complémentaires.

4.2.3.2 Job de génération VHDL

Le job suivant gère la génération des fichiers VHDL. Il est simplement dépendant du job d'initialisation puisqu'il requiert ActiveTcl [35] et il nécessite le service de GitLab Runner.

Plusieurs variables d'environnement locales sont employées. Quelques-unes doivent être définies par le développeur puisqu'elles concernent le projet. Elles sont expliquées dans la partie configuration 4.2.4. D'autres ne sont pas à modifier impérativement telle que la variable donnant le nom du journal généré.

La partie "script" contient une seule commande (ligne 28 du code 4.3). Il s'agit d'exécuter le programme ActiveTcl (exécutable : "tclsh") avec en paramètre le nom du script Tcl qui va faire la démarche pour générer tous les fichiers VHDL. Cette commande est inscrite dans une condition car les fichiers sont générés seulement si la simulation et/ou la synthèse sont activées. Sinon, un message est affiché sans interrompre le pipeline, signalant qu'aucun fichier n'a été généré.

En dernier lieu, seul le journal créé par HDS est sauvé en tant qu'artefact (ligne 34 du code 4.3).

```

1  buildVHDL-job:      # This job builds the VHDL files of HDS project.
2  stage: build_VHDL
3  tags:
4    - Windows
5    - EDA
6  needs: [init-job]
7  variables:
8    #-----
9    # Project
10   #
11   REQUIRE_LIBS:      1
12   HDP:                $CI_PROJECT_DIR/Prefs/hds.hdp
13   PREFS_DIR:         $CI_PROJECT_DIR/Prefs
14   TESTBENCH_LIB:     Chronometer_test
15   PROGRAM_LIB:       Board
16   PROGRAM_ENTITY:    ELN_chrono
17   PROGRAM_ARCH:      struct.bd
18   #-----
19   # Log
20   #
21   LOG_FILE:          genVHDL_log.txt
22   script:
23     #-----
24     # Run generation
25     #
26     - >
27       if (($SIMULATION_EN -eq "1") -or ($SYNTHESIS_EN -eq "1")) {
28         tclsh $SCRIPTS_DIR/GenVHDL_Workflow_main.tcl
29       } else {
30         "Simulation and synthesis are not active. Therefore, no VHDL
↪ file has been generated."
31       }
32   artifacts:
33     paths:
34       - $CI_PROJECT_DIR/$LOG_FILE

```

Code source 4.3 Script : `.gitlab-ci.yml`, partie : job de génération `VHDL`

4.2.3.3 Job de simulation

Ce job s'occupe de la simulation toujours à l'aide du même service de GitLab Runner que les deux jobs présentés antérieurement. Il est dépendant de la génération `VHDL` puisqu'il requiert obligatoirement ces fichiers afin de simuler un comportement. En raison du choix que peut entreprendre le développeur à exécuter la simulation, une condition est définie (lignes 8 à 10 du code 4.4). Si celle-ci est vraie (simulation désactivée), le job n'aura pas lieu. Sinon, il aura lieu seulement si le job de génération a été réalisé avec succès.

Les variables locales sont présentes pour définir le fichier de consigne pour la simulation, le nom des journaux à sauver et enfin la variable de Git [36] informant qu'il ne faut pas réinitialiser le répertoire local de la machine physique (ligne 21 du code 4.4). Effectivement, normalement après chaque job, le répertoire Git local est effacé pour être cloné à nouveau. Si cela est fait, tous les fichiers `VHDL` générés seront supprimés.

La partie "script" est similaire à celle de la génération `VHDL`. Le programme ActiveTcl [35] est appelé avec comme paramètre les commandes de simulation dans un script `Tcl`. En revanche, l'exécutable est appelé à chaque exécution de ce job.

Chapitre 4. Implémentation

Dans la partie des [artefacts](#), les journaux de compilation et de simulation ainsi que le fichier comportant les signaux numériques stimulés sont sauves (lignes 32 à 34 du code 4.4).

```
1  simulate-job:      # This job simulates the HDS project.
2      stage: sim_syn
3      tags:
4          - Windows
5          - EDA
6      needs: [buildVHDL-job]
7      rules:
8          - if: $SIMULATION_EN == "0"
9            when: never
10         - when: on_success
11      variables:
12          #-----
13          # Project
14          #
15          DO_PATH:      $SIMULATION_DIR/chronometer.do
16          #-----
17          # Log
18          #
19          LOG_COM_FILE:  com_log.txt
20          LOG_SIM_FILE:  sim_log.txt
21          #-----
22          # Git
23          #
24          GIT_CLEAN_FLAGS: none    # Keep VHDL files of build_VHDL stage
25      script:
26          #-----
27          # Run simulation
28          #
29          - tclsh $SCRIPTS_DIR/Sim_Workflow_main.tcl
30      artifacts:
31          paths:
32              - $SIMULATION_DIR/$LOG_COM_FILE
33              - $SIMULATION_DIR/$LOG_SIM_FILE
34              - $SIMULATION_DIR/$DESIGN_NAME.wlf
```

Code source 4.4 Script : *.gitlab-ci.yml*, partie : job de simulation

4.2.3.4 Job de synthèse

Le dernier job, étant la synthèse, demande le même service de GitLab Runner que les trois autres jobs. Il dépend aussi du job de génération [VHDL](#). Il contient la même condition que la simulation hormis qu'il possède une autre variable d'activation.

Une variable locale indique le fichier de contrainte utilisé pour synthétiser le développement. D'autres variables permettent de définir le nom des journaux qui seront sauves. La même variable Git que la simulation est paramétrée pour que le job ne réinitialise pas le dépôt local sur la machine du GitLab Runner.

Le bloc "scripts" est aussi similaire à la simulation. Il fait appel au programme ActiveTcl [35] pour interpréter le script [Tcl](#) de synthèse.

En dernier lieu, les [artefacts](#) sont sauves (lignes 34 à 37 du code 4.5).

```
1  synthesize-job:    # This job builds the bitstream file of Xilinx.
2      stage: sim_syn
```

```

3      tags:
4        - Windows
5        - EDA
6      needs: [buildVHDL-job]
7      rules:
8        - if: $SYNTHESIS_EN == "0"
9          when: never
10       - when: on_success
11     variables:
12       #-----
13       # Project
14       #
15       ISE_PROJECT_PATH: $CI_PROJECT_DIR/Board/ise/el_n_chrono.xise
16       UCF_PATH:          $CI_PROJECT_DIR/Board/concat/el_n_chrono.ucf
17       #-----
18       # Log
19       #
20       LOG_FILE:          syn_log.txt
21       SUMMARY_FILE:      syn_summary.txt
22       PROPERTIES_FILE:   process_properties.tcl
23       #-----
24       # Git
25       #
26       GIT_CLEAN_FLAGS:  none    # Keep VHDL files of build_VHDL stage
27     script:
28       #-----
29       # Run synthesis
30       #
31       - tclsh $SCRIPTS_DIR/Syn_Workflow_main.tcl
32     artifacts:
33       paths:
34       - $SYNTHESIS_DIR/$PROPERTIES_FILE
35       - $SYNTHESIS_DIR/*.bit
36       - $SYNTHESIS_DIR/$LOG_FILE
37       - $SYNTHESIS_DIR/$SUMMARY_FILE

```

Code source 4.5 Script : `.gitlab-ci.yml`, partie : job de synthèse

4.2.4 Configuration

Le développeur est amené à configurer quelques variables qui doivent être propres à son projet en question.

En commençant par expliquer les variables d'environnement se situant à partir de la ligne 6 du code 4.1, plusieurs d'entre elles ont été définies pour des intérêts différents. Des variables de projet y figurent (lignes 5 à 7 du code 4.6). Le nom du design ("DESIGN_NAME") donne le nom du fichier contenant les signaux numériques de simulation qui va être généré et sauvé dans les *artefacts*. Le nom inscrit n'a donc peu d'importance. Il est ensuite demandé de spécifier l'entité ("TESTBENCH_ENTITY") et l'architecture ("TESTBENCH_ARCH") du banc de test dans le cas où la simulation est activée dans le *pipeline*. Elles sont retrouvables dans le projet HDS comme le montre l'exemple de la figure 4.12. Sinon, le développeur peut laisser vide entre les guillemets et les variables seront ignorées. Il est important de mettre le nom complet de l'architecture avec son extension à la ligne 7 du code 4.6 !

Chapitre 4. Implémentation

La suite concerne les répertoires (lignes 11 à 13 du code 4.6) où sont contenus les scripts du `pipeline` ("SCRIPTS_DIR"), les fichiers de la simulation ("SIMULATION_DIR") et de la synthèse ("SYNTHESIS_DIR"). Ils n'ont pas la nécessité d'être modifiés sauf si le développeur souhaite simplement changer leur chemin et/ou leur nom.

En dernier lieu se trouvent les variables d'activation (lignes 17 et 18 du code 4.6). Le développeur choisit s'il active la simulation ("SIMULATION_EN") avec la valeur "1" ou non avec la valeur "0". Il a aussi le choix avec la synthèse ("SYNTHESIS_EN") en mettant à "1" ou à "0".

```
1  variables:
2      #-----
3      # Project
4      #
5      DESIGN_NAME:      "eln_chrono"
6      TESTBENCH_ENTITY:  "chronometer_tb"
7      TESTBENCH_ARCH:    "struct.bd"
8      #-----
9      # Directories
10     #
11     SCRIPTS_DIR:        $CI_PROJECT_DIR/Scripts
12     SIMULATION_DIR:     $CI_PROJECT_DIR/Simulation
13     SYNTHESIS_DIR:      $CI_PROJECT_DIR/Synthesis
14     #-----
15     # Jobs to start
16     #
17     SIMULATION_EN:      1
18     SYNTHESIS_EN:       1
```

Code source 4.6 Script : `.gitlab-ci.yml`, partie : variables d'environnement globales

En passant maintenant dans la partie des variables d'environnement locales, le job de génération `VHDL` en possède plusieurs. Elles sont expliquées à partir de la ligne 7 à 21 du code 4.3. Dans la section du projet, la variable "REQUIRE_LIBS" informe si des librairies externes d'`HDS` doivent être requises. Le chemin et le nom du projet `HDS` ("HDP") sont ensuite à spécifier, en plus du chemin des préférences `HDS` ("PREFS_DIR"). Si la simulation est activée, le nom de la librairie du banc de test doit être indiqué ("TESTBENCH_LIB"). Il reste à mentionner la librairie ("PROGRAM_LIB"), l'entité ("PROGRAM_ENTITY") et l'architecture ("PROGRAM_ARCH") pour la synthèse si elle aussi est activée. Une dernière variable censée ne pas être modifiée donne le nom du journal créé par `HDS` durant la génération et sauvé dans les `artefacts` par la suite.

Dans le job de simulation (lignes 11 à 24 du code 4.4), quelques variables d'environnement locales sont définies tels que le chemin et le nom du fichier `.do` ("DO_PATH"), les journaux ("LOG_COM_FILE" et "LOG_SIM_FILE") et encore la variable de Git ("GIT_CLEAN_FLAGS"). Le fichier `.do` est celui qui contient les commandes de simulation. Il est préalablement configuré par le développeur. La commande Git est paramétrée par la valeur "none". Ceci permet d'éviter que le répertoire local du GitLab Runner soit nettoyé et remis à zéro avant la simulation car les fichiers `VHDL` générés précédemment doivent toujours être présents.

Dans le job de synthèse (lignes 11 à 26 du code 4.5), plusieurs groupes de variables sont définis. Concernant le projet, si un projet ISE est déjà existant dans le dépôt Git, le développeur peut le spécifier avec la variable "ISE_PROJECT_PATH". Sinon un nouveau y sera créé avant la synthèse et la variable "UCF_PATH" devra impérativement être définie par le nom du fichier de contrainte *.ucf* préalablement configuré. La suite implique les variables donnant le nom des journaux et fichiers sauvés plus tard dans les *artefacts*. La dernière variable "GIT_CLEAN_FLAGS" s'occupe du gestionnaire de version qui indique de ne pas réinitialiser le dépôt local comme la simulation.

4.3 Flux de travail CI

L'implémentation du flux de travail CI explique en détail la construction et la simulation de la conception matérielle. Il s'agit de la partie pratique de ce travail où plusieurs scripts ont été développés séparément dans le but d'automatiser ces *workflows*.

4.3.1 Génération automatique de fichiers VHDL

Le programme HDS peut s'utiliser de deux manières distinctes. Soit il est manipulé depuis une interface graphique, ou soit directement en ligne de commande Tcl. Généralement, l'interface graphique permet principalement la conception du design. Les lignes de commandes se conforment plus particulièrement au traitement des conceptions. Dans le cas du CI, le deuxième choix s'applique. Pour l'implémentation de ces lignes de commandes, quatre scripts sont employés afin d'exécuter les tâches propre à la génération de fichiers VHDL. Un script Tcl, nommé *GenVHDL_Workflow_main.tcl*, s'occupe de la configuration du programme HDS, de l'appel du programme HDS, de la suppression des importations de librairie après concaténation et de l'appel au script qui contrôle les erreurs. Le second, appelé *GenVHDL_Workflow_HDS.tcl*, gère les commandes spécifiques à l'environnement de conception HDS. Le script *searchPaths.tcl* cherche les outils requis à la tâche dans la machine physique du GitLab Runner. Et enfin le script *checkErrors.tcl* contrôle les erreurs qui auraient pu survenir durant la génération.

4.3.1.1 Configuration

Plusieurs étapes de configuration sont importantes pour éviter tout problème durant la génération.

```

1 source $scripts_dir/searchPaths.tcl
2
3 set require_libs      $env(REQUIRE_LIBS)
4 if {$require_libs == 1} {
5     set hei_libs_dir  [require_libs]
6 }
7 set hds_home          [require_hds]
```

Code source 4.7 Script : *GenVHDL_Workflow_main.tcl*, partie : outils requis

Tout d'abord, la recherche des outils requis est cruciale. Un outil peut être installé à un endroit distinct selon la machine physique utilisée par GitLab Runner. Il pourrait aussi de manière inattendue avoir été supprimé sans en connaître les raisons. Les lignes 3 à 7 du code 4.7 définissent les outils éventuellement requis pour la tâche. Dans ce cas, uniquement HDS est obligatoire. Il se peut que des librairies externes soient utilisées dans le projet et ainsi être demandées (ligne 4 à 6 du code 4.7). Le script *searchPaths.tcl*

Chapitre 4. Implémentation

permet de chercher ses moyens. Une inclusion de celui-ci est faite à la ligne 1 du code 4.7. Une partie de ce script est montré au code source 4.8. Il contient uniquement des méthodes qui sont uniques à chaque outil.

```
1  proc require_hds {} {
2      set hds_home C:/eda/MentorGraphics/HDS
3      if {[file exist $hds_home]} {
4          set hds_home C:/tools/eda/HDS
5          if {[file exist $hds_home]} {
6              puts " ERROR: No valid installation of HDL-Designer found"
7              exit 1
8          }
9      }
10     puts " Found HDL-Designer"
11     return $hds_home
12 }
```

Code source 4.8 Script : *searchPaths.tcl*, partie : recherche *HDS*

Le code 4.8 montre le moyen de recherche du programme *HDS*. Si celui-ci est requis, sa méthode est appelée (ligne 1 du code 4.8). Un premier chemin d'emplacement par défaut est alors testé (lignes 2 et 3 du code 4.8). S'il n'est pas existant, un second chemin est testé (ligne 4 et 5 du code 4.8). Si ce dernier n'est pas connu, une erreur est signalée et le *pipeline* est ainsi interrompu (ligne 6 et 7 du code 4.8). Autrement, dans le cas où il est trouvé, la variable contenant le bon chemin est retournée (ligne 11 du code 4.8) pour être sauvee et utilisée plus tard dans le code. Les recherches de chaque outils fonctionnent avec ce même principe expliqué.

```
1  if {$require_libs == 1} {
2      set ::env(HEI_LIBS_DIR) $hei_libs_dir
3  }
4  set ::env(HDS_HOME) $hds_home
5  set ::env(HDS_LIBS) $library_matchings
6  set ::env(HDS_USER_HOME) $user_prefs_dir
7  set ::env(HDS_TEAM_HOME) $team_prefs_dir
```

Code source 4.9 Script : *GenVHDL_Workflow_main.tcl*, partie : variables d'environnement

La suite des commandes s'occupe de préparer convenablement des variables d'environnement. Les chemins d'emplacement des outils essentiels sont d'abord sauves (lignes 2 et 4 du code 4.9). Une gestion des variables du programme *HDS* est sollicitée pour qu'il sache exactement quel projet traiter avec quelles préférences. Effectivement, il est presque obligatoire de sélectionner le projet dans le but d'être certain du design qui va être traité. S'il n'est pas sélectionné, le dernier projet ouvert sur la machine du GitLab Runner sera sélectionné. La ligne 5 du code 4.9 spécifie le chemin complet de ce projet. La variable d'environnement *HDS_LIBS* appartient à *HDS* qui est spécifiée dans le document [37] à la page 59.

HDS laisse l'utilisateur sélectionner ses préférences. Elles sont définies comme suit par la documentation [15] :

HDL Designer Series prend en charge les préférences d'équipe, qui sont destinées à être partagées par tous les membres d'une équipe travaillant sur la même conception, et les préférences utilisateur qui peuvent être définies

par chaque utilisateur individuel. Les préférences d'équipe comprennent les règles de nom de fichier HDL, les propriétés de génération, l'enregistrement des fichiers et les options de configuration de la gestion des versions.

Les préférences de l'utilisateur comprennent des options pour la configuration générale, la création de textes et de diagrammes, les contrôles de génération HDL et l'enregistrement des diagrammes. Vous pouvez également définir des préférences pour VHDL, Verilog, la gestion de la conception, HDL2Graphics et chaque type de diagramme graphique.

Dans le cadre de ce travail, les préférences sont utilisées pour gérer les tâches. Les tâches sont des processus que peut exécuter le programme HDS comme par exemple la génération de fichier VHDL avec la tâche appelée "Generate". La figure 4.11 montre un exemple de tâches existantes dans l'environnement.

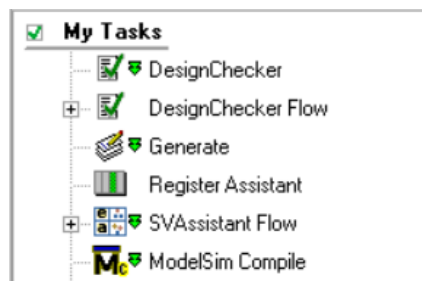


Figure 4.11 Exemple de tâches présentes dans le programme HDS

Chaque tâche peut être de type différent. Soit elle est propre à HDS, soit propre à un programme externe ou soit personnalisée par le développeur.

Aux lignes 6 et 7 du code 4.9, les emplacements des fichiers de préférence sont donnés. Les tâches sont inscrites dans ceux-ci. Ces deux variables d'environnement, `HDS_USER_HOME` et `HDS_TEAM_HOME` sont référencées dans la documentation [37] aux pages 61 et 62. Il faut faire très attention aux tâches incluses dans le répertoire des préférences (chemin d'accès à partir du répertoire Git : `Prefs/hds_user/v2019.2/tasks`). La tâche de concaténation `concatenate.tsk` doit obligatoirement être présente pour la suite des opérations. Si elle n'y est pas, celle-ci doit être copiée depuis les préférences installées par défaut par le programme HDS (généralement au chemin suivant : `C:/Users/name/AppData/Roaming/HDL Designer Series/hds_user/v2019.2/tasks`).

4.3.1.2 Génération

Lorsque la configuration est accomplie sans problème, l'environnement de développement HDS peut être exécuté avec des arguments particuliers à lui. Plusieurs arguments possibles sont donnés dans la documentation [37] aux pages 18 et 19. Dans ce cas, seul un argument est indispensable, celui qui demande de lire un script Tcl (argument "-tcl"). Il faut donc spécifier le script `GenVHDL_Workflow_HDS.tcl` comportant les commandes de génération. La ligne 1 du code 4.10 montre parfaitement cette explication. Le mot-clé "exec" permet le lancement du programme HDS suivi de son nom d'exécutable. En fin de ligne se trouve la partie pour générer un fichier texte qui va contenir toutes les informations découlant du programme HDS. Ce journal est nécessaire au contrôle d'erreur par la suite. L'exécution se fait dans une commande "catch" qui permet d'attraper

Chapitre 4. Implémentation

une erreur pouvant apparaître durant l'exécution. Il est d'ailleurs préférable de placer toute les commandes "exec" dans un "catch" pour permettre d'attraper et signaler une erreur correctement. Avec l'aide de la condition, l'erreur est signalée et va interrompre le [pipeline](#) (ligne 2 et 3 du code [4.10](#)).

```
1      if {[catch {exec hdl designer -tcl $tcl_script_hds > $log_file} e]} {  
2          puts "$e"  
3          exit 1  
4      }
```

Code source 4.10 Script : *GenVHDL_Workflow_main.tcl*, partie : lancement d'*HDS* en mode *shell*

Dans ce script *GenVHDL_Workflow_HDS.tcl* démarré par l'environnement *HDS*, la première étape consiste à paramétrer la génération (commande "setupGenerate") (ligne 4 du code [4.11](#)). Plusieurs paramètres pouvant être changés sont indiqués dans la documentation [\[10\]](#) aux pages 104 et 105. Dans ce cas, il est spécifié de toujours générer les fichiers *VHDL* malgré qu'ils soient inchangés (argument "-generateAlways"). De plus, pour que tous les blocs et les composants puissent être générés en langage *VHDL*, il est nécessaire de tous les passer à travers (argument "-throughCpt").

```
1      # Arguments :  
2      # - generateAlways : Always generate even if no changes have been made.  
3      # - throughCpt : Goes down through blocks and components.  
4      setupGenerate -generateAlways ON -throughCpt
```

Code source 4.11 Script : *GenVHDL_Workflow_HDS.tcl*, partie : configuration de la génération *VHDL*

Pour la commande de génération, il n'est pas nécessaire de passer par l'appel de la tâche *Generate* vue dans l'exemple de la figure [4.11](#). Ne pas l'utiliser permet de ne pas dépendre des tâches. La commande de génération spécifique à *HDS* (commande "runGenerate", document [\[10\]](#) à la page 103) se trouve aux lignes 2 et 5 du code [4.12](#). Elle demande d'abord le nom de la librairie ainsi que son entité et finalement son architecture du design. Ces arguments sont spécifiés par le développeur dans le script du [pipeline](#) *.gitlab-ci.yml*. Ils sont retrouvables dans le projet *HDS* comme le montre l'exemple de la figure [4.12](#). Il est évident de toujours mettre en argument l'entité la plus haute hiérarchiquement.

```
1      if {$simulation_en == 1} {  
2          runGenerate $testbench_lib $testbench_entity $testbench_arch  
3      }  
4      if {$synthesis_en == 1} {  
5          runGenerate $program_lib $program_entity $program_arch  
6      }
```

Code source 4.12 Script : *GenVHDL_Workflow_HDS.tcl*, partie : génération de fichiers *VHDL*

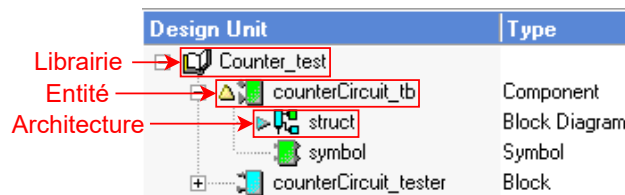


Figure 4.12 Exemple de librairie, d'entité et d'architecture dans un projet

Etant donné que le développeur a libre choix de lancer une simulation et une synthèse, la génération VHDL se fait en fonction (lignes 1 et 4 du code 4.12). Les fichiers VHDL sont générés chacun dans leur librairie respective (exemple à la figure 4.13).

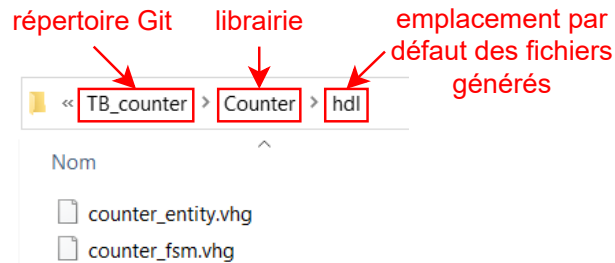


Figure 4.13 Emplacement des fichiers générés dans un exemple de projet

4.3.1.3 Concaténation

Puisque chaque librairie possède généralement plusieurs blocs de conception désormais écrits en VHDL chacun dans des fichiers séparés, il est préférable de les concaténer en un seul fichier pour la simulation et la synthèse.

```

1      if {$simulation_en == 1} {
2          runTask Concatenate $testbench_lib $testbench_entity [regsub {\.*}]
↪ $testbench_arch ""
3      }
4      if {$synthesis_en == 1} {
5          runTask Concatenate $program_lib $program_entity [regsub {\.*}]
↪ $program_arch ""
6      }

```

Code source 4.13 Script : GenVHDL_Workflow_HDS.tcl, partie : concaténation des fichiers VHDL

Pour exécuter la concaténation, aucune commande spécifique n'est disponible. Il est obligatoire de passer par l'appel d'une tâche. Pour rappel, cette tâche *concatenate.tsk* doit être présente dans les préférences. Une commande permet uniquement de lancer des tâches (commande "runTask", lignes 2 et 5 du code 4.13). Elle demande en argument premièrement le nom de la tâche (tâche *Concatenate* dans ce travail). La suite des arguments est similaire à la génération. Il est nécessaire de fournir la librairie, l'entité ainsi que l'architecture. En complément, l'architecture doit être passée sans l'extension de son fichier. Par exemple, il faut enlever *.bd* à la fin du nom pour un diagramme en bloc. A nouveau, la concaténation est lancée en fonction du choix du développeur (lignes 1 et 4 du code 4.13). Les fichiers VHDL sont concaténés chacun dans leur librairie respective (exemple à la figure 4.14). Le programme HDS a alors terminé d'exécuter le script *GenVHDL_Workflow_HDS.tcl*.

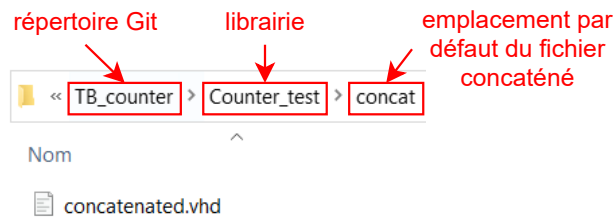


Figure 4.14 Emplacement du fichier concaténé dans un exemple de projet

Puisque la tâche de concaténation ne fait que lier les fichiers ensemble, les importations des anciennes bibliothèques y figurent encore dans le code. Afin d'éviter tout conflit, il est nécessaire de les supprimer. Pour cela, un script en langage Perl, appelé *trimLibs.pl*, a déjà été développé par la [HEI](#). Celui-ci s'exécute avec le programme Perl préinstallé avec [HDS](#).

```
1      if {$simulation_en == 1} {
2          exec $perl $trim $concat_testbench_dir/$concat_file
3      }
4      if {$synthesis_en == 1} {
5          exec $perl $trim $concat_program_dir/$concat_file
6      }
```

Code source 4.14 Script : *GenVHDL_Workflow_main.tcl*, partie : suppression des importations de librairie

En revenant dans le script *GenVHDL_Workflow_main.tcl* après l'exécution de [HDS](#), le code 4.14 va d'abord vérifier si la simulation et/ou la synthèse sont activées. Ensuite, les lignes 2 et 5 du code 4.14 vont exécuter le programme Perl avec le script *trimLibs.pl*. Deux paramètres au script sont demandés. Le premier concerne le fichier concaténé par [HDS](#) et le second l'emplacement du nouveau fichier concaténé avec son nom.

4.3.1.4 Contrôle d'erreur

Lorsque la génération est réalisée, une partie des informations obtenues dans le journal est semblable à la figure 4.15.

```
39 Performing hierarchical generation through components...
40 Checking which design units need saving
41 Incrementally generating HDL...
42 . .
43 TB_project_lib/state_machine_XYZ
44 Generating entity and architecture C:\GitLab-Runner\builds\sjc_PsUx\0\SPL\
45   bachelorthesis\2022-fpga_cicd\fpga_ci-cd_xavierclivaz\TB_project\
46   TB_project_lib\hdl\state_machine_xyz_struct.vhd
47 Generation completed successfully.
48 -----
49 Completed TCL file processing
```

Figure 4.15 Exemple d'affichage d'informations de la génération

Dans ces renseignements, la ligne 47 de la figure 4.15 indique l'état du déroulement. Soit la génération a été accomplie avec succès, soit aucune génération n'a été effectuée en raison d'aucun changement de conception ou soit une erreur est apparue.

```

1  proc checkErrorsHDS {log_file} {
2      set msg_success    "Generation completed successfully."
3      set msg_nothing    "Generated HDL up to date, nothing to do."
4      set msg_error      "Generation completed with errors."
5
6      set log [open $log_file r]
7      while {[gets $log line] != -1} { # Get each line of log file
8          if {$line == $msg_success} { # Success
9              puts $msg_success
10             }
11             if {$line == $msg_nothing} { # Nothing
12                 puts $msg_nothing
13             }
14             if {$line == $msg_error} { # Error
15                 puts $msg_error
16                 exit 1
17             }
18         }
19         close $log
20     }

```

Code source 4.15 Script : *checkErrors.tcl*, partie : *HDL Designer Series*

Afin d'éviter le lancement d'une simulation ou d'une synthèse en cas d'erreur, un appel à une procédure du script de test *checkErrors.tcl* est opéré à l'aide du journal en argument. Le principe pour trouver l'erreur est de traverser le journal ligne par ligne (ligne 7 du code 4.15) jusqu'à atterrir sur l'un des trois messages détaillés plus haut. Si le message d'erreur y apparaît, l'erreur est signalée et le *pipeline* est ainsi interrompu (ligne 15 et 16 du code 4.15).

4.3.2 Simulation et analyse automatique de bancs de test VHDL

Dans cette implémentation, seul ModelSim [17] est employé. Ce programme peut fonctionner sans interface graphique et être lancé directement par des lignes de commande au langage *Tcl*. Il possède des exécutables différents pour chaque tâche que l'utilisateur souhaite effectuer. Par exemple dans ce cas, les tâches de compilation ("*vcom*") et de simulation ("*vsim*") sont utilisées. Toutes les commandes exécutables sont listées et décrites dans ce document officiel de ModelSim [38] (pages 51 à 85).

Son implémentation est conçue principalement dans deux scripts distincts. Le premier est le script principal *Sim_Workflow_main.tcl* qui peut être utilisé pour d'autres programmes de simulation. Le second est *Sim_Workflow_ModelSim.tcl* qui contient les commandes propres à ModelSim.

4.3.2.1 Configuration

ModelSim est l'unique outil requis comme le montre le code 4.16. La ligne 3 du code 4.16 fait appel à une procédure qui va aller chercher le chemin d'accès du programme s'il est existant. Sinon un message d'erreur ainsi qu'une interruption du *pipeline* seront émis.

```

1  source $scripts_dir/searchPaths.tcl
2
3  set modelsim_home [require_modelsim]

```

Code source 4.16 Script : *Sim_Workflow_main.tcl*, partie : outils requis

Concernant les variables d'environnement, aucune n'est importante à paramétrer puisque le logiciel ModelSim est très indépendant.

```
1  if {[catch {exec vlib $sim_lib} e]} {  
2      puts "$e"  
3      exit 1  
4  }
```

Code source 4.17 Script : *Sim_Workflow_ModelSim.tcl*, partie : création de la librairie de simulation

Pour permettre la simulation du banc de test avec sa conception, ModelSim nécessite un répertoire de travail spécifique à lui seul. Une commande spéciale existe à cet effet à la ligne 1 du code 4.17 (commande "vlib" appartenant à ModelSim) qui va créer une librairie. Il s'agit d'une commande exécutable directement avec "exec". Un seul argument est demandé, celui qui indique le chemin et le nom du répertoire à concevoir. Cette commande se situe à nouveau dans un "catch" pour informer et interrompre correctement le pipeline en cas d'erreur durant l'exécution.

4.3.2.2 Compilation

La prochaine étape concerne déjà la compilation du code source VHDL en un code objet qui est un langage de très bas niveau compréhensible par ModelSim. Le script va donc exécuter la commande "vcom" avec plusieurs arguments (ligne 1 du code 4.18). Premièrement, la version du langage est indiquée (optionel). Ensuite, une option possible est la génération d'un journal en fichier texte permettant de recueillir les informations du déroulement de la compilation. Dans cette implémentation, elle est utilisée dans le but de pratiquer une analyse de la compilation et de pouvoir garder une trace dans les artefacts. L'argument suivant implique l'espace de travail. Il s'agit de la librairie créée précédemment au code 4.17. Finalement, il reste à mentionner le chemin et le nom du code source VHDL correspondant au banc de test. Dans ce cas, le fichier VHDL est celui concaténé et préparé spécialement pour la simulation à la ligne 2 code 4.14.

```
1  if {[catch {exec vcom ->$compiler_version -l $log_com_file -work $sim_lib  
↪ $hdl_path >@stdout} e]} {  
2      puts "$e"  
3      exit 1  
4  }
```

Code source 4.18 Script : *Sim_Workflow_ModelSim.tcl*, partie : compilation

4.3.2.3 Simulation

La suite concerne l'étape de simulation à l'aide de la commande exécutable "vsim" (ligne 1 du code 4.19). ModelSim se lance avec plusieurs arguments. Le premier ("do") concerne les tâches qu'a à faire ModelSim. Elles sont listées dans un fichier .do configurable par le développeur selon son projet. L'emplacement et le nom de celui-ci sont donc à mentionner. L'argument ("L") suivant demande la librairie de travail où se trouvent tous les fichiers compilés. Ensuite ("I"), à nouveau un journal est spécifié pour l'aperçu du déroulement de la simulation. Par après ("wlf"), le chemin et le nom du fichier, qui contiendra les signaux stimulés, sont spécifiés. L'argument ("c") informe ModelSim qu'il

doit simuler en mode ligne de commande et non avec l'interface graphique. Le dernier argument demande la structure du banc de test à simuler. Il est placé dans la librairie spécifique de ModelSim créée précédemment.

```

1  if {[catch {exec vsim -do $do_path -L $sim_lib -l $log_sim_file -wlf
↪ $sim_dir/$design_name.wlf -c $sim_lib.$testbench_entity}($testbench_arch)
↪ >@stdout} e]} {
2      puts "$e"
3      exit 1
4  }

```

Code source 4.19 Script : *Sim_Workflow_ModelSim.tcl*, partie : *simulation*

Une attention particulière doit être portée au fichier *.do*. Puisque celui-ci dicte les règles de simulation, il peut aussi indiquer la durée de simulation (exemple : "run 500 us"). Si aucune durée n'est limitée dans le banc de test, celle-ci doit l'être dans le fichier *.do*. Effectivement, si le temps de simulation est illimité, le [pipeline](#) sera bloqué dans la simulation jusqu'à ce que son délai soit échu. Chaque job a une durée de fonctionnement limitée qui peut être paramétrée par l'utilisateur.

4.3.2.4 Analyse

La dernière tâche de ce [workflow](#) concerne l'analyse. Deux analyses doivent être réalisées, une pour la compilation et une pour la simulation.

Une procédure d'analyse d'erreur pour ModelSim a été réalisée dans le script *checkErrors.tcl*. Elle prend en argument un journal généré. Dans le cas de la compilation, son journal est passé en revue ligne par ligne jusqu'à trouver la ligne informant le nombre d'erreur. Elle est ensuite traitée et affichée sur la console. Si une ou plusieurs erreurs sont détectées, un message est signalé et le [pipeline](#) est interrompu. Il s'agit du même principe pour l'analyse de la simulation. Le code 4.20 représente la méthode d'analyse.

```

1  set log [open $log_file r]
2  while {[gets $log line] != -1} {
3      if {[string match *$msg* $line]} {
4          if {[string match *$msg_success* $line]} {
5              puts $line
6          } else {
7              puts $line
8              exit 1
9          }
10     }
11 }
12 close $log

```

Code source 4.20 Script : *checkErrors.tcl*, partie : *ModelSim*

4.4 Flux de travail CD

L'implémentation du flux de travail [CD](#) explique en détail la synthèse de la conception matérielle. Celle-ci est la mise en pratique de la génération du fichier [bitstream](#) propre au fabricant Xilinx qui est automatisée par le biais du [pipeline](#) et de scripts.

4.4.1 Synthèse automatique

Expliqué dans la conception, la synthèse se fait à l'aide de l'unique logiciel Xilinx ISE Design Suite [25], appelé plus simplement ISE. Celui-ci se munit d'un mode ligne de commande afin de pouvoir exécuter des commandes en langage **Tcl**. Il n'a donc besoin d'aucune interface graphique pour générer un fichier **bitstream** à partir d'un design en **VHDL**.

Pour son implémentation, à nouveau deux scripts sont principalement développés uniquement pour la synthèse. *Syn_Workflow_main.tcl* est le script principal conçu pour être portable et donc utilisé par d'autres outils de synthèse. *Syn_Workflow_ISE.tcl* est le script spécifique à ISE contenant les commandes spécifiques à la synthèse.

4.4.1.1 Configuration

Le code 4.21 montre que l'outil nécessaire est ISE à l'exception d'**HDS**. **HDS** est présent car il possède dans son installation de base le programme Perl qui sera utilisé plus tard dans l'implémentation.

```
1 source $scripts_dir/searchPaths.tcl
2
3 set ise_version 14.7
4 set ise_home [require_ise $ise_version]
5 set hds_home [require_hds]
```

Code source 4.21 Script : *Syn_Workflow_main.tcl*, partie : outils requis

Comme dans les **workflows CI**, plusieurs variables locales et d'environnement sont définies.

4.4.1.2 Synthèse

Avant que la synthèse ne s'exécute, le lancement du programme ISE doit être réalisé à partir de la commande "exec" (ligne 1 du code 4.22). Son argument requiert un script **Tcl** possédant les commandes d'ISE. Toutes les informations découlant du logiciel sont sauveées dans un fichier texte pour le contrôle d'erreur par la suite. Comme tout exécutable, il est démarré dans un "catch" pour toute éventuelle erreur pouvant survenir. Il est à noter que les variables "errorCode" (ligne 2 du code 4.22) et "errorInfo" (ligne 3 du code 4.22) sont propres à la commande "catch". "errorCode" contient le code d'erreur s'il devait y avoir une erreur. Dans ce cas, "errorInfo" inclut le message d'erreur.

```
1 catch {exec $ise $tcl_script_ise > $log_file}
2 if {$errorCode != "NONE"} {
3     puts $errorInfo
4     exit 1
5 }
```

Code source 4.22 Script : *Syn_Workflow_main.tcl*, partie : lancement d'ISE en mode **shell**

Lorsque le programme de synthèse est exécuté, il requiert tout d'abord un projet propre à lui (exemple de projet au code 4.23). C'est pourquoi en début de script, un projet va être ouvert s'il en existe déjà un dans le répertoire Git ou il va être créé. C'est au développeur d'indiquer par une variable d'environnement "ISE_PROJECT_PATH" s'il

en existe un. Le fichier de projet (.xise) se compose principalement de deux parties, celle où sont inclus les fichiers nécessaires (ligne 6 à 13 du code 4.23) et celle où sont définies les propriétés des processus qui vont être lancés par la suite (ligne 15 à 20 du code 4.23).

```

1      <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2      <project xmlns="http://www.xilinx.com/XMLSchema"
3      ↪      xmlns:xil_pn="http://www.xilinx.com/XMLSchema">
4
5          <version xil_pn:ise_version="14.5" xil_pn:schema_version="2"/>
6
7          <files>
8              <file
9              ↪      xil_pn:name="E:/projects/ELN_chrono/Board/concat/el_n_chrono.ucf"
10             ↪      xil_pn:type="FILE_UCF">
11                 <association xil_pn:name="Implementation" xil_pn:seqID="0"/>
12             </file>
13             <file
14             ↪      xil_pn:name="E:/projects/ELN_chrono/Board/concat/el_n_chrono.vhd"
15             ↪      xil_pn:type="FILE_VHDL">
16                 <association xil_pn:name="Implementation" xil_pn:seqID="1"/>
17             </file>
18         </files>
19
20         <properties>
21             <property xil_pn:name="Add I/O Buffers" xil_pn:value="true"
22             ↪      xil_pn:valueState="default"/>
23             <property xil_pn:name="Allow Logic Optimization Across Hierarchy"
24             ↪      xil_pn:value="false" xil_pn:valueState="default"/>
25             <property xil_pn:name="Allow SelectMAP Pins to Persist"
26             ↪      xil_pn:value="false" xil_pn:valueState="default"/>
27             <property xil_pn:name="Allow Unexpanded Blocks" xil_pn:value="false"
28             ↪      xil_pn:valueState="default"/>
29         </properties>
30     </project>

```

Code source 4.23 Script : *project_name.xise*, exemple de projet ISE

Si un projet est déjà présent, en raison des chemins des codes sources pouvant être différents (lignes 7 et 10 du code 4.23), un script en langage Perl, appelé *update_ise.pl*, va modifier ceux-ci. Ils seront assurément correct. Ceci est établi à la ligne 1 du code 4.24. Il ne reste plus qu'à ouvrir le projet à partir de la commande "project open" (document [28], page 375) (ligne 6 du code 4.24).

```

1      if {[catch {exec $perl $update_ise $ise_project_path $hdl_file $ucf_file
2      ↪      >@stdout} e]} {
3          puts "$e"
4          exit 1
5      }
6
7      project open $ise_project_path

```

Code source 4.24 Script : *Syn_Workflow_ISE.tcl*, partie : ouverture d'un projet ISE

Dans le cas où le projet ISE doit être créé, une commande spécifique le permet à la ligne 1 du code 4.25 (commande "project new", document [28], page 374). Il faut indiquer en argument le chemin ainsi que le nom du projet à créer.

Chapitre 4. Implémentation

Plusieurs propriétés du projet sont tout d'abord demandées à être déclarées avec la commande "project set" (document [28], page 376). Il y a la famille d'**FPGA**, son modèle, son paquet ainsi que son grade de vitesse (lignes 4 à 7 du code 4.25). Elles sont listées dans la documentation [28] à la page 346.

Il reste essentiellement à ajouter tous les fichiers nécessaires à la synthèse (commande "xfile add") tels que le fichier **VHDL** concaténé (ligne 10 du code 4.25) et le fichier de contrainte (.ucf) (ligne 11 du code 4.25). Le fichier de contrainte sert à définir le câblage entre les ports du circuit et les broches du **FPGA**.

Par la suite, il est toujours possible de modifier certains paramètres. Dans le cadre de ce travail, pour éviter tout problème avec l'affichage sur l'écran LCD qui peut être ajouté à la carte de développement **FPGA**, une propriété a été modifiée au processus de synthèse (ligne 14 du code 4.25).

```
1      project new $synthesis_dir/$design_name.xise
2
3      # Project settings
4      project set family    $family
5      project set device    $device
6      project set package   $package
7      project set speed     $speed
8
9      # Add files to the project (must come after the settings)
10     xfile add $hdl_file
11     xfile add $ucf_file
12
13     # Removes the LCD screen problem
14     project set "Other XST Command Line Options" "-use_new_parser yes" -process
↪     "Synthesize - XST"
```

Code source 4.25 Script : *Syn_Workflow_ISE.tcl*, partie : création d'un projet ISE

Puisque plusieurs processus sont employés pour la génération du fichier **bitstream**, une liste de toutes les propriétés de chaque processus est réalisée dans un fichier **Tcl** (code 4.26). Elle a pour but de renseigner le développeur en cas de problème ou autre. Le fichier sera sauvé ultérieurement dans les **artefacts**. L'idée est de passer à travers chaque processus (ligne 2 du code 4.26) et de lire ensuite chaque propriété (ligne 4 du code 4.26). Ce code a été conseillé par un exemple du document officiel de Xilinx [28] à la page 401.

```
1      set pf [open $properties_path w]
2      foreach ISE_app $process_list {
3          puts $pf "# ***** Properties for < $ISE_app > *****"
4          foreach prop [project properties -process $ISE_app] {
5              set val [project get "$prop" -process "$ISE_app"]
6              if {$val != ""} {
7                  puts $pf "project set \"$prop\" \"$val\" -process \"$ISE_app\""
8              }
9          }
10     }
11     close $pf
```

Code source 4.26 Script : *Syn_Workflow_ISE.tcl*, partie : sauvegarde des propriétés des processus

Le projet est désormais prêt à être synthétisé. Il existe un processus de synthèse appelé "Synthesize - XST" (document [28], page 347). Il est lancé à partir de la commande "process run" (document [28], page 369) (ligne 1 du code 4.27). Cette partie de code 4.27 a été proposée dans le document [28] à la page 401. Le processus est exécuté dans une commande "time" pour connaître le temps écoulé du processus. Si aucune erreur est attrapée par la commande "catch", une lecture du statut du processus est faite (ligne 5 du code 4.27). Elle permet de connaître si la synthèse a eu lieu correctement. Si ceci n'est pas le cas, une erreur est signalée et le [pipeline](#) est interrompu (lignes 9 et 10 du code 4.27).

```

1  if {[catch {time {process run "Synthesize - XST"}} synthTime]} {
2      puts "Synthesis run failed to launch."
3      exit 1
4  } else {
5      set my_status [process get "Synthesize - XST" status]
6      if {($my_status == "up_to_date") || ($my_status == "warnings")} {
7          puts "Synthesis run completed successfully, runtime: $synthTime"
8      } else {
9          puts "Synthesis run failed, runtime: $synthTime"
10         exit 1
11     }
12 }

```

Code source 4.27 Script : *Syn_Workflow_ISE.tcl*, partie : processus de synthèse

Afin de générer le fichier [bitstream](#), deux autres processus doivent être exécutés successivement. Le suivant concerne l'implémentation par le biais du processus "Implement Design". Il s'agit en réalité d'un ensemble de sous processus tels que "Translate" (document [28], page 351), "Map" (document [28], page 352), "Place & Route" (document [28], page 354) et "Generate Post-Place & Route Static Timing" (document [28], page 364). Le dernier processus s'occupe de générer le fichier [bitstream](#) avec le processus "Generate Programming File" (document [28], page 356). Le code permettant de les exécuter n'est pas inscrit dans cette partie d'implémentation puisqu'il s'agit du même procédé que le code 4.27 hormis le nom des processus.

Pour que la synthèse se termine correctement, il est conseillé de fermer le projet ISE avant de quitter le programme. Une commande appelée "project close" (document [28], page 372) répond à ce point.

```

1  if {[project close]} {
2      puts "Correct closure of the project"
3  } else {
4      puts "Error: Incorrect closure of the project"
5      exit 1
6  }

```

Code source 4.28 Script : *Syn_Workflow_ISE.tcl*, partie : fermeture du projet ISE

4.4.1.3 Contrôle d'erreur

Lorsque la synthèse est terminée, un passage à travers le journal généré par ISE est effectué. Il permet de vérifier le nombre d'erreurs et d'avertissements de chaque processus. Le principe est de compter le nombre d'avertissements et d'erreur dans chacune des

étapes y compris l'ouverture ou la création du projet ISE. Lorsqu'une étape se termine, les nombres comptés sont affichés et contrôlés. A la moindre erreur détectée, elle est signalée et le [pipeline](#) est interrompu.

4.4.1.4 Création du sommaire de synthèse

Malgré que toutes les informations de synthèse figurent dans le journal généré par ISE, le nombre de lignes est inconsidérable. La recherche demande donc un temps colossal. C'est pourquoi, un sommaire des informations importantes a été développé pour faciliter grandement le développeur voulant chercher par exemple le nombre de bascules utilisées dans le circuit [FPGA](#). Le code va récupérer généralement les rapports générés dans chaque processus pour les concaténer dans un sommaire. Ces rapports sont toujours retrouvables dans le journal de synthèse.

5 | Validation

Dans ce travail de [CI/CD](#), la validation prouve le fonctionnement général d'un [pipeline](#) automatisé pour un développement matériel. La preuve peut uniquement être réalisée à partir de projets [EDA](#). Cette partie se repose donc principalement sur un projet existant et mis à disposition par la [HEI](#). Une preuve de concept est aussi démontrée sur la carte de développement [FPGA](#) mise à disposition.

Table des matières

5.1	Tests des différents flux de travail	58
5.1.1	Meilleurs cas	58
5.1.2	Pires cas	61
5.2	Tests sur carte de développement FPGA	64
5.3	Discussion	65

5.1 Tests des différents flux de travail

La preuve du fonctionnement général de l'implémentation se base sur un protocole de test réalisé à l'aide d'un projet matériel. Ce projet, mis à disposition par la [HEI](#), est un chronomètre. Il a été conçu dans le but de faire tourner une aiguille chaque seconde sur un cadran d'horloge mécanique (figure 5.2). Celui-ci peut être démarré par l'action d'un bouton, tout comme l'arrêt, le redémarrage et la remise à zéro.

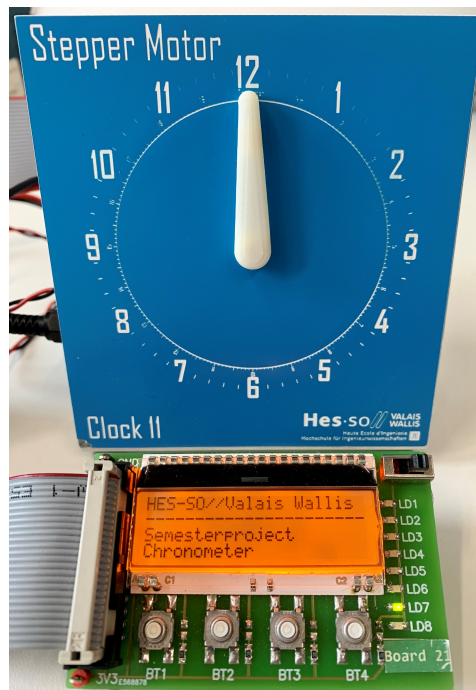


Figure 5.1 *Projet chronomètre*

Plusieurs scénarios ont été testés afin de s'assurer que l'implémentation ne possède aucun défaut majeur. Le premier scénario prend en compte les meilleurs cas d'utilisation du [pipeline](#) automatisé. Il considère que le projet du chronomètre a été développé sans erreur et que le développeur a configuré le fichier [YAML](#) correctement. Le second scénario concerne les pires cas d'utilisation. Il peut arriver que la conception matérielle contienne des erreurs et/ou le développeur ait mal configuré le fichier [YAML](#).

Dans les tests réalisés, il est important de savoir que seul un service de GitLab Runner a été en fonction. La simulation et la synthèse ne sont donc pas lancées en parallèle.

5.1.1 Meilleurs cas

Cette section met en avant les tests de l'implémentation qui ont été réalisés dans les meilleures conditions. Un diagramme de séquence a été fait à la figure 5.2 pour comprendre un des meilleurs cas.

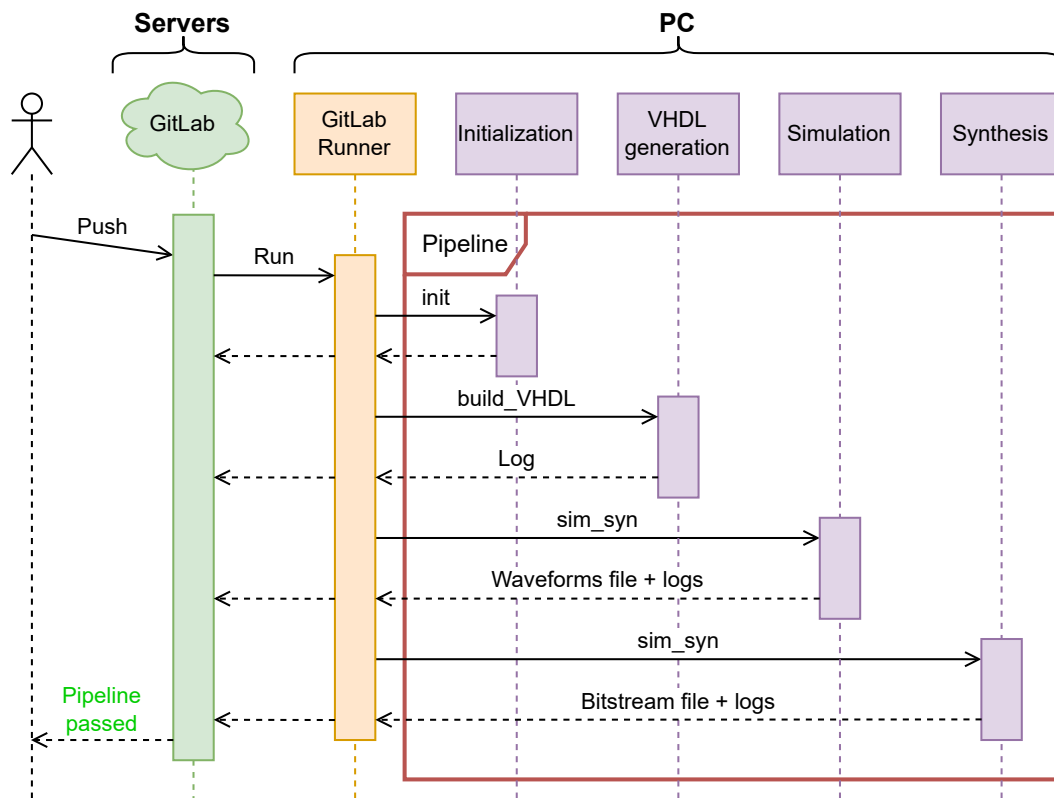


Figure 5.2 Meilleur cas : tous les jobs sont activés

Ce diagramme part du principe que le projet ne contient aucune erreur. Le développeur pousse son travail sur le référentiel partagé. GitLab Runner est lancé pour exécuter tous les jobs du **pipeline**. Lorsque chaque job est terminé, il retourne les **artefacts** jusqu'au serveur GitLab. L'utilisateur peut alors les récupérer. De plus, un message de réussite est signalé au développeur (figure 5.3).

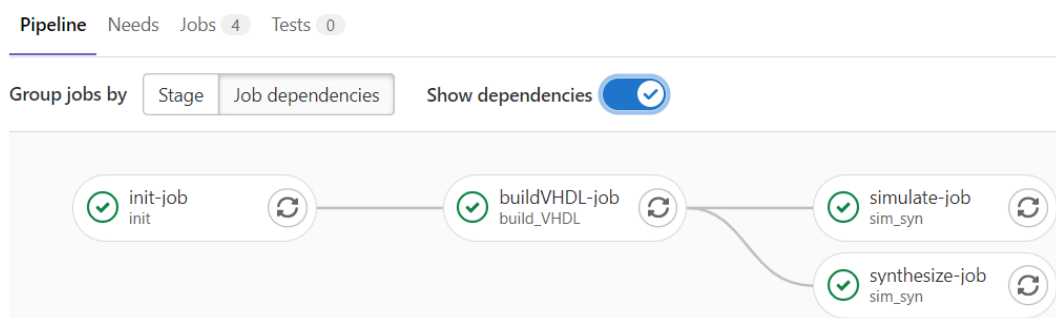


Figure 5.3 Réussite du **pipeline** avec les quatre jobs

Le diagramme de séquence de la figure 5.4 montre un second meilleur scénario. Il est identique à celui de la figure 5.2 hormis que le développeur a décidé de ne pas activer la simulation. Le **pipeline** peut malgré cela être lancé en ignorant simplement le job de simulation.

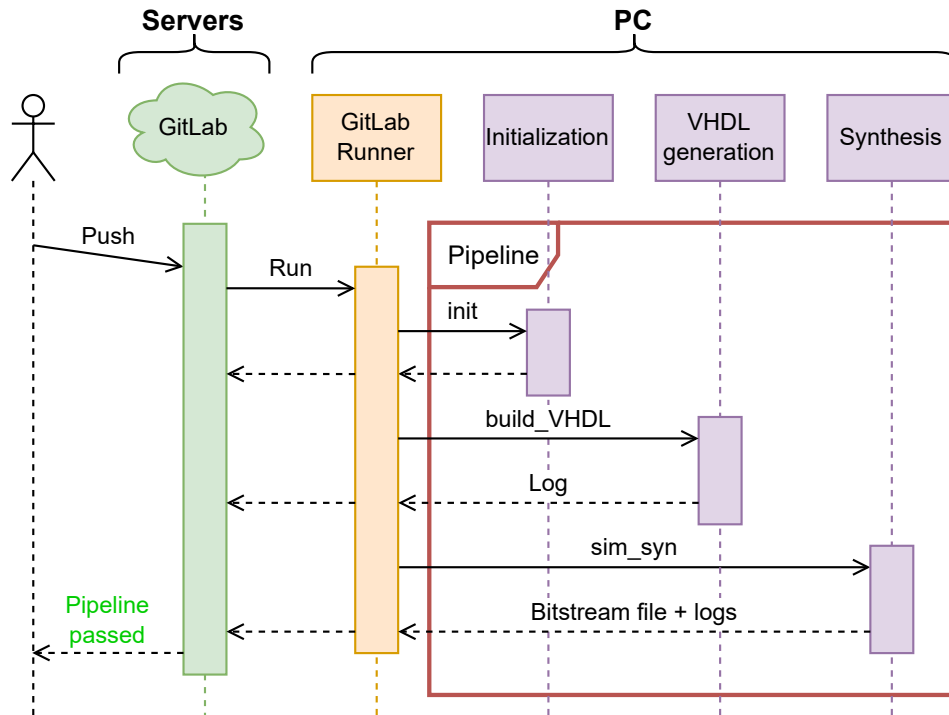


Figure 5.4 Meilleur cas : la simulation est désactivée

Ces deux scénarios ont pu être testés à l'aide du projet chronomètre (figures 5.3 et 5.5). Aucun problème n'est survenu durant cette phase de test. Les diagrammes de séquence ont bien été respectés. Bien évidemment qu'il existe aussi le cas d'utilisation où la synthèse est désactivée. Il s'agit de tests généraux pour ces trois cas. Ils vérifient uniquement que le passage dans le pipeline se soit bien déroulé.

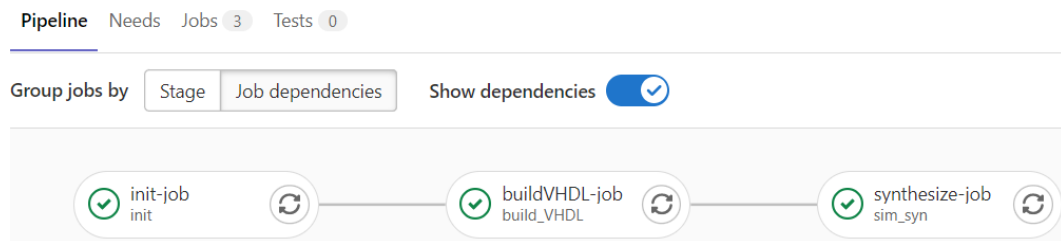


Figure 5.5 Réussite du pipeline sans la simulation

Divers cas ont été effectués plus en détail dans chacun des jobs. En commençant par l'initialisation, le programme ActiveTcl a pu être trouvé. En se référant à la génération VHDL, les fichiers ont toujours été générés en fonction de l'état d'activation de la simulation et de la synthèse (figure 5.6). La simulation est soit ignorée ou soit activée. La synthèse quant à elle a été contrôlée en la lançant avec et sans un projet ISE. Les deux cas se sont bien passés.

```

249 Check generation error
250 -----
251 Generation completed successfully. (for simulation)
252 Generation completed successfully. (for synthesis)
253
254 End time : 18-08-22 15:12:24
255 Elapsed time : 00:00:06
256
257 -----
258 Finished script
259 -----
261 Uploading artifacts for successful job
262 Version:      15.0.0
263 Git revision: febb2a09
264 Git branch:   15-0-stable
265 GO version:   go1.17.7
266 Built:        2022-05-19T19:34:09+0000
267 OS/Arch:      windows/amd64
268 Uploading artifacts...
269 Runtime platform arch=amd64 os=windows pid=8068 revisi
270 C:\GitLab-Runner\builds\P5B6HuZ4\0\SPL\bachelorthesis\2022-fpga_cicd\tb_xavierclivaz_fpga
g.txt: found 1 matching files and directories
271 Uploading artifacts as "archive" to coordinator... 201 Created id=12188 responseStatus=2
273 Cleaning up project directory and file based variables
275 Job succeeded

```

Figure 5.6 Réussite de la génération *VHDL* avec la simulation et la synthèse activées

La figure 5.7 montre un exemple d'*artefacts* générés durant la synthèse et déposés sur la plateforme d'hébergement GitLab.

Artifacts / Synthesis

Name	Size
..	
eln_chrono.bit	277 KB
process_properties.tcl	10 KB
syn_log.txt	186 KB
syn_summary.md	11.9 KB

Figure 5.7 Exemple d'*artefacts* déposés après le job de synthèse

5.1.2 Pires cas

Cette section met en avant les tests de l'implémentation qui ont été réalisés dans les pires conditions. Un diagramme de séquence a été élaboré à la figure 5.8 pour comprendre un des pires cas.

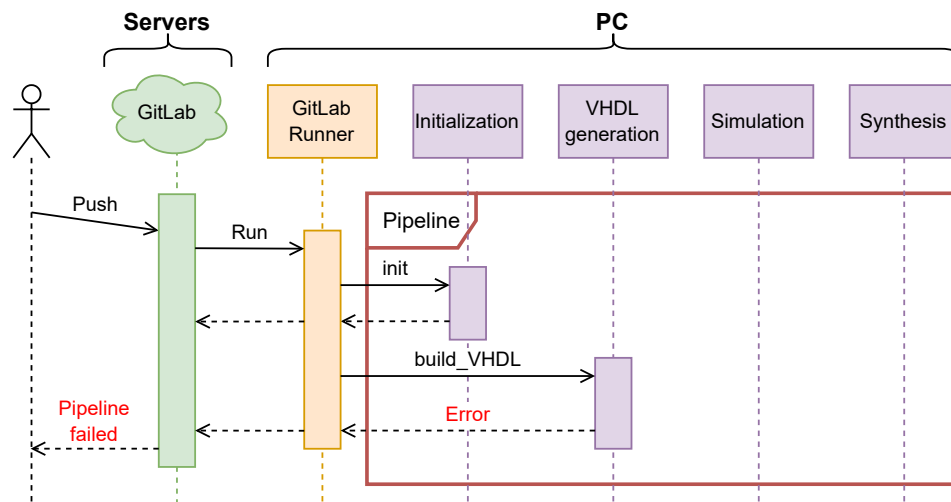


Figure 5.8 Pire cas : erreur de génération VHDL

Ce diagramme part du principe que le projet contient une erreur dans la conception matérielle. Le développeur pousse son travail sur le référentiel partagé. GitLab Runner est lancé pour exécuter tous les jobs du [pipeline](#). Lorsque le job de génération est exécuté, il aperçoit une erreur et la signale jusqu'à l'utilisateur. Le [pipeline](#) est donc interrompu (figure 5.9).

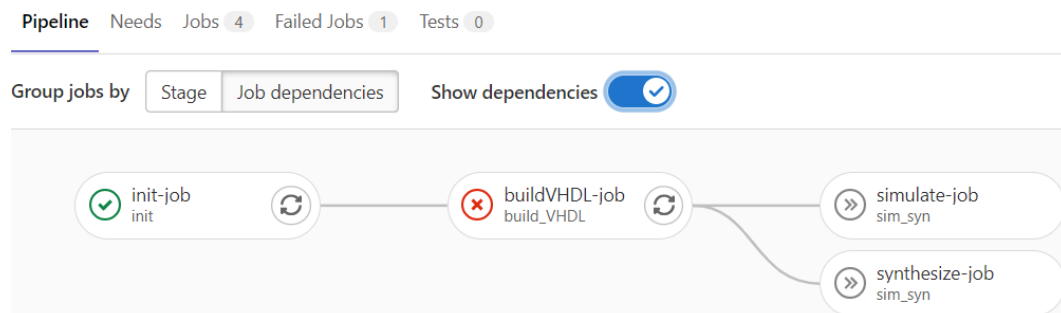


Figure 5.9 Echec du [pipeline](#) à cause du job de génération VHDL

Un autre cas peut être une erreur de simulation (figure 5.10). Cette fois-ci, la conception possède un problème de comportement. Etant donné que la simulation et la synthèse sont indépendantes l'une de l'autre, la synthèse va pouvoir s'exécuter après l'erreur détectée dans la simulation (figure 5.11).

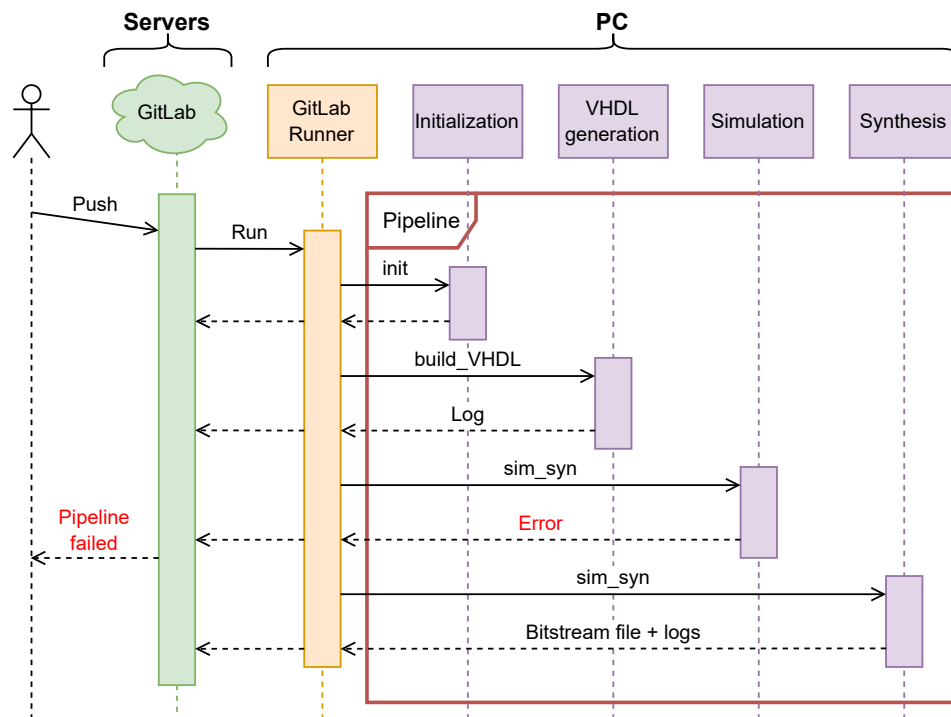


Figure 5.10 Pire cas : erreur de simulation

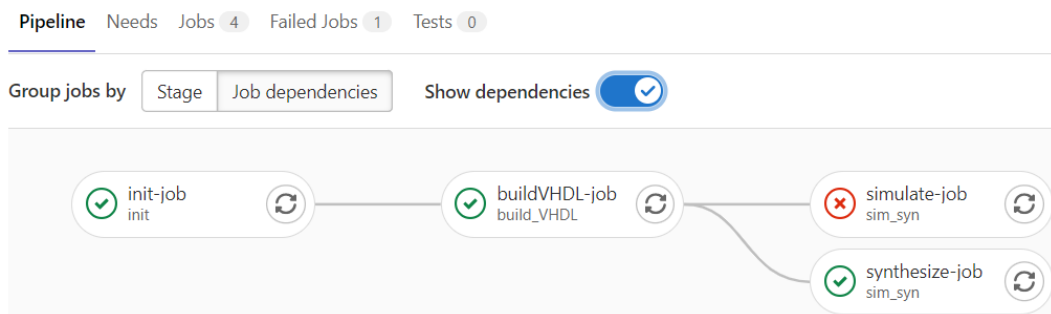


Figure 5.11 Echec du pipeline à cause du job de simulation

En allant plus en profondeur dans les jobs, plusieurs tests ont été réalisés. En ce qui concerne l'initialisation, les tests se sont bien déroulés avec des messages d'erreur pertinents. Le programme ActiveTcl [35] a par exemple été supprimé pour vérifier le comportement du job.

Dans le travail de la génération VHDL, plusieurs cas ont été contrôlés tels que des variables du script YAML incorrectes ou la tâche de concaténation inexistante. Tous les tests ont eu des résultats cohérents. Ils informaient généralement correctement l'erreur. De plus, un test a été fait avec une erreur dans la conception du chronomètre. Cette erreur a bien été signalée.

Au niveau de la simulation, des essais ont principalement été réalisés avec des variables du fichier **YAML** mal configurées. Tous les tests étaient concluants. Plusieurs erreurs ont été glissées dans le banc de test de la conception pour voir apparaître des messages d'erreurs pertinents. La figure 5.12 montre une autre erreur. Il s'agit du cas où le développeur n'a pas spécifié le bon fichier **.do**.

```
154 # C:\GitLab-Runner\builds\P5B6HuZ4\0\SPL\bachelorthesis\2022-fpga_cicd\tb_xavierclivaz_fpga\ELN_chrono_ci_cd/Simulation/wrong.do
155 # invalid command name "C:\GitLab-Runner\builds\P5B6HuZ4 SPL\bachelorthesis\2022-fpga_cicd b_xavierclivaz_fpgaELN_chrono_ci_cd/Simulation/wrong.do"
156 VSIM 2> #
157 # <EOF>
158 # End time: 18:50:26 on Aug 14,2022, Elapsed time: 0:00:01
159 # Errors: 1, Warnings: 0
160
161 End time of ModelSim : 14-08-22 18:50:26
162 Elapsed time of ModelSim : 00:00:04
163
164 Check compilation error
165 -----
166 Errors: 0, Warnings: 2
167
168 Check simulation error
169 -----
170 # Errors: 1, Warnings: 0
172 Cleaning up project directory and file based variables
174 ERROR: Job failed: exit status 1
```

Figure 5.12 Erreur de simulation

Pour terminer, des variables dans le job de synthèse ont aussi été configurées incorrectement. Le fichier de contrainte a par exemple été corrompu pour vérifier la réaction du **pipeline**. Tous les tests étaient aussi concluants.

5.2 Tests sur carte de développement **FPGA**

Quelques tests ont été effectués sur la carte de développement **FPGA** toujours à partir du projet chronomètre. Il s'agit dans ce cas de prouver le concept. Pour preuve, le design du projet chronomètre doit avoir été généré en un fichier de configuration lisible et cohérent par la carte de développement.

Le premier test consistait à vérifier que le fichier **bitstream** programme correctement la carte **FPGA** en partant d'un projet ISE existant dans le dépôt. La carte a donc été programmée et l'écran LCD a pu afficher un texte (figure 5.1). En appuyant sur le bouton "BT1" de la carte d'extension (figure 5.2), le chronomètre a démarré. Son aiguille tourne chaque seconde. En appuyant sur le bouton "BT2", l'aiguille a été arrêtée. En appuyant sur le bouton "BT3", l'aiguille est revenue à la valeur douze du cadran. En ayant aussi testé d'appuyer sur le bouton "BT1" après que l'aiguille était arrêtée, l'aiguille a pu reprendre son comptage. Ce test a été un succès.

Un second test similaire était prévu dans le cas où aucun projet ISE n'était pas disponible dans le dépôt. Le job de synthèse a dû créer ce projet. Le fichier **bitstream** a été généré convenablement. Après la programmation, l'écran a affiché le bon texte. Les actions sur les boutons ont été entreprises similairement au premier test. L'essai a aussi été un succès.

Un mauvais cas de programmation a été effectué. L'idée a été de modifier le fichier de contrainte UCF en changeant une piste de la carte [FPGA](#) comme le montre le code 5.1. "start_n" correspond à l'entrée du signal démarrant le chronomètre. "D3" correspond à la pin de sortie de la carte [FPGA](#). Cette pin a été changée dans le but d'être reliée à un autre bouton.

```
1 NET "start_n" LOC = "D3" ;
```

Code source 5.1 *Script : Sim_Workflow_main.tcl, partie : outils requis*

Après avoir programmé ce fichier de configuration erroné, l'aiguille a commencé par tourner chaque seconde sans rien n'y toucher. Le bouton "TB1" ne fonctionne plus puisque celui-ci a été remplacé par un autre. Le bouton "BT2" stope l'aiguille uniquement s'il est maintenu enfoncé. Lorsqu'il est relâché, l'aiguille continue à tourner. Le bouton "BT3" remet le chronomètre à zéro uniquement si le bouton "BT2" est enfoncé. Ce test montre véritablement que le fichier a été généré convenablement selon le fichier de contrainte UCF.

5.3 Discussion

En conclusion, l'implémentation peut être validée. Un protocole de test se trouvant en annexe [D](#) a été élaboré dans le but de tester l'entièreté de l'implémentation. Les résultats ont généralement été concluants. Aucun défaut majeur n'a été soulevé. Le [pipeline](#) a toujours exécuté ses tâches comme il le devait ainsi que les différents [workflows](#). Les réussites ont toujours été rapportées tout comme les erreurs. Seules quelques erreurs rapportées étaient trop floues. Il était donc compliqué de déceler la véritable problématique ayant déclenché l'erreur.

6 | Conclusion

6.1 Résumé du projet

Ce projet a eu pour but de réaliser de l'intégration continue ([Continuous Integration](#)) et de la [distribution continue \(Continuous Delivery\)](#) conçues pour du développement matériel de type [FPGA](#). Un [pipeline](#) a alors été construit au fur et à mesure des trois mois de travail à disposition. Trois flux de travail ont dû être implémentés dans le [pipeline](#), la génération de fichiers [VHDL](#), la simulation et la synthèse. Ces flux sont présents dans le but de développer un fichier de configuration à partir d'une conception matérielle prêt à être déployé sur la carte de développement [FPGA](#). Avant d'implémenter cela, il a fallu mettre en oeuvre un service permettant de gérer ce [pipeline](#). Puisque le référentiel partagé utilisé appartient à GitLab, le service utilisé provient du programme GitLab Runner. Il a été installé convenablement sur une machine physique ayant l'environnement Windows. Arrivé au bout des trois mois, ce projet a pu se concrétiser intégralement selon le cahier des charges.

6.2 Comparaison avec les objectifs initiaux

Ce projet a été un succès. Tous les objectifs ont été atteints. Une preuve de ce concept a été démontrée à partir d'un développement matériel conçu par la [HEI](#).

L'énergie fournie par le développeur est nettement moins conséquente avec le [CI/CD](#). Effectivement, pour développer un fichier de configuration à partir d'une conception matérielle tout en passant par la simulation, le nombre de manipulations est relativement grand. Plusieurs programmes doivent à chaque fois être démarrés, le développeur doit passer à travers différents boutons présents sur l'interface graphique des programmes pour avancer comme le [pipeline](#) automatisé. Une connaissance des étapes est donc nécessaire pour n'en rater aucune. Il s'agit aussi d'une moins bonne fiabilité. Le [CI/CD](#) possède en permanence la même procédure qui permet de garantir cette fiabilité.

En parlant de toutes les étapes devant être réalisées, le gain de temps est aussi précieux pour le développeur. Il a initialement à configurer un seul fichier pour que l'entièreté du [pipeline](#) puisse fonctionner. Ceci se fait au début d'un projet comme l'installation d'un service de GitLab Runner. Ceci dit, après chaque lancement du [pipeline](#) et jusqu'à l'achèvement, le développeur peut consacrer son temps à d'autres tâches qu'il a à réaliser.

Un point à relever concerne les outils de développement [EDA](#). Ils doivent être installés sur une machine physique contenant GitLab Runner de la même manière que sur la machine d'un développeur. A partir du [CI/CD](#), l'avantage est que le développeur n'a plus besoin d'installer ses outils sur sa machine. De plus, si le projet regroupe plusieurs développeurs, une seule licence des programmes serait nécessaire. Il y a donc un certain avantage au niveau du coût d'un développement.

6.3 Difficultés rencontrées

Au cours de ce travail, plusieurs difficultés se sont présentées dans le développement du concept. La première est survenue durant l'implémentation du [workflow](#) de génération [VHDL](#). [HDS](#), installé correctement sur la machine physique contenant GitLab Runner, refusait de générer les fichiers [VHDL](#) à chaque lancement du [pipeline](#). Le problème était dû à une mauvaise installation du programme GitLab Runner car l'utilisateur Windows de celui-ci était différent de l'utilisateur Windows possédant le programme [HDS](#). Le problème peut être évité en configurant le bon utilisateur à l'installation de GitLab Runner.

Une seconde difficulté a été l'implémentation du [workflow](#) de génération [VHDL](#). Il s'agissait du premier flux à implémenter dans ce concept. L'expérience n'était pas encore présente. Beaucoup de recherches ont donc dû être réalisées sur le fonctionnement d'un flux. De plus, étant donné que le programme [HDS](#) ne possède pas beaucoup de documentation dans les recherches scientifiques, son implémentation n'a pas été simple. Il a fallu découvrir tout son mécanisme uniquement à l'aide des documents fournis par [HDS](#). Une grande partie du temps de ce travail a été consacré à cette tâche.

6.4 Perspectives d'avenir

Plusieurs horizons peuvent être perçus pour la suite du développement de ce concept. Premièrement, la [HEI](#) peut désormais mettre en place ce travail dans ses développements matériels. Il est suffisamment complet et fonctionnel. Il peut totalement être exploité tout en étant amélioré en parallèle.

Une perspective serait de compléter ce concept avec d'autres outils [EDA](#). Au vu de sa portabilité, ce concept possède une base solide pouvant intégrer facilement d'autres programmes sans affecter ceux déjà présents. Cette amélioration permettrait de mettre en place ce concept dans une plus grande variété de projets [EDA](#). De plus, ajouter des logiciels de simulation créant des bancs de test en langage logiciel permettrait de distinguer la conception de la simulation.

Pour aller plus loin dans le concept, il serait encore plus avantageux que le [pipeline](#) automatisé programme directement la carte de développement [FPGA](#) pour procéder automatiquement à une analyse physique des signaux. Ce flux de travail pourrait sans soucis être ajouté après la synthèse. Une étude du comportement dans un environnement physique est de bien meilleure qualité qu'une simulation virtuelle. Toutes les informations seraient directement retournées au développeur en passant par le référentiel partagé.

A | Docker

Cette annexe explique tout d'abord l'installation de l'image Docker préconfigurée comportant le programme GitLab Runner. La suite décrit la liaison du logiciel GitLab Runner installé virtuellement avec un projet sur le référentiel partagé.

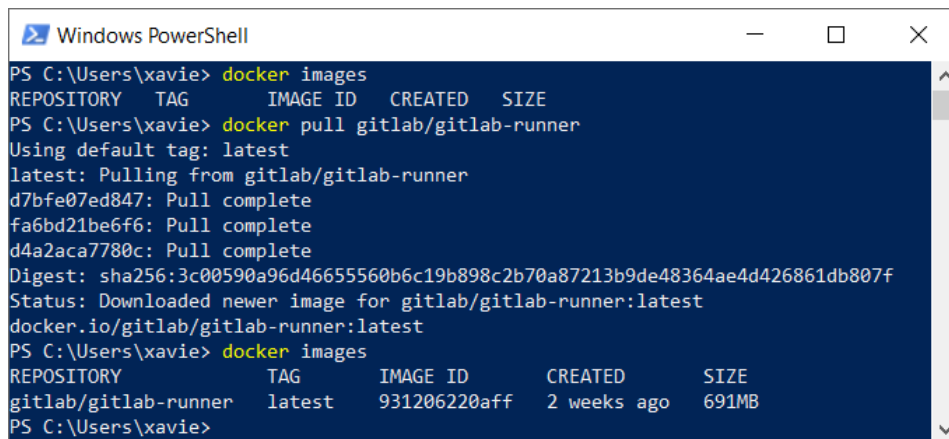
A.1 Installer l'image de Gitlab Runner

Il est tout d'abord important de télécharger le programme Docker sur une machine physique. Toutes les indications sur l'installation sont données à cette référence [39].

Par la suite, il faut ouvrir l'invite de commande et exécuter la commande suivante afin de clôner l'image préconfigurée :

```
docker pull gitlab/gitlab-runner
```

La figure A.1 montre le clônage de l'image.



```
Windows PowerShell
PS C:\Users\xavie> docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
PS C:\Users\xavie> docker pull gitlab/gitlab-runner
Using default tag: latest
latest: Pulling from gitlab/gitlab-runner
d7bfe07ed847: Pull complete
fa6bd21be6f6: Pull complete
d4a2aca7780c: Pull complete
Digest: sha256:3c00590a96d46655560b6c19b898c2b70a87213b9de48364ae4d426861db807f
Status: Downloaded newer image for gitlab/gitlab-runner:latest
docker.io/gitlab/gitlab-runner:latest
PS C:\Users\xavie> docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
gitlab/gitlab-runner latest 931206220aff 2 weeks ago 691MB
PS C:\Users\xavie>
```

Figure A.1 Installation de l'image Docker préconfigurée

L'image est désormais prête à l'utilisation. Son installation peut être vérifiée à l'aide de la commande suivante comme le montre la dernière commande de la figure A.1 :

```
docker images
```

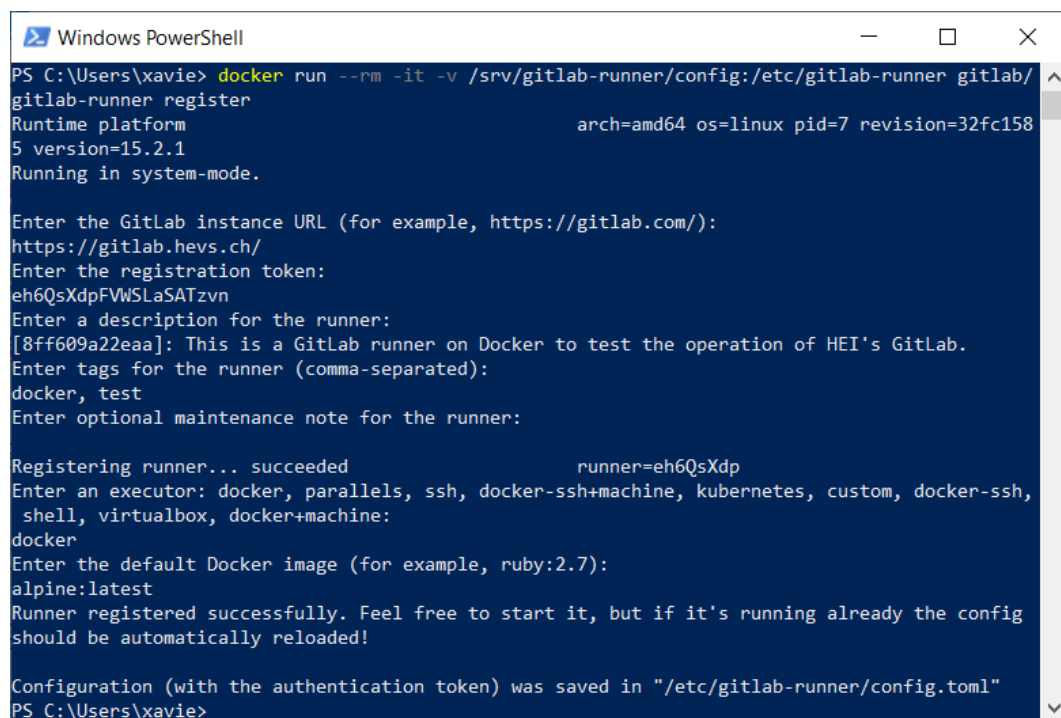
A.2 Lier GitLab Runner au référentiel GitLab

Il est à présent important de lier le service de GitLab Runner au référentiel en question. Pour ce faire, toujours avec l'invite de commande, la commande à exécuter est la suivante :

Annexe A. Docker

```
docker run --rm -it -v /srv/gitlab-runner/config:/etc/gitlab-runner  
→ gitlab/gitlab-runner register
```

Quelques informations sont demandées (figure A.2). Tout d'abord un lien vers le serveur de GitLab doit être spécifié. Ce lien est retrouvable sur le dépôt du projet dans l'onglet "Settings -> CI/CD -> Runner -> Specific runners". Un jeton d'enregistrement est aussi présent pour être écrit dans la prochaine information demandée. Une description peut ensuite être notée suivie d'étiquettes permettant d'identifier le GitLab Runner par la suite. Par après, il est demandé de spécifier l'exécuteur des commandes du [pipeline](#). Il s'agit de "docker" dans ce cas. En dernier lieu, une image doit être spécifiée. Il s'agit de l'image où les commandes vont être exécutées par le service de GitLab Runner. Dans ce cas, une petite image fonctionnant avec Linux est choisie. Suffisante pour réaliser une évaluation, elle se nomme "alpine".



```
Windows PowerShell
PS C:\Users\xavie> docker run --rm -it -v /srv/gitlab-runner/config:/etc/gitlab-runner gitlab/
gitlab-runner register
Runtime platform                                arch=amd64 os=linux pid=7 revision=32fc158
5 version=15.2.1
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
https://gitlab.hevs.ch/
Enter the registration token:
eh6QsXdpFVWSLaSATzvn
Enter a description for the runner:
[8ff609a22eaa]: This is a GitLab runner on Docker to test the operation of HEI's GitLab.
Enter tags for the runner (comma-separated):
docker, test
Enter optional maintenance note for the runner:



Registering runner... succeeded                                runner=eh6QsXdp
Enter an executor: docker, parallels, ssh, docker-ssh+machine, kubernetes, custom, docker-ssh,
shell, virtualbox, docker+machine:
docker
Enter the default Docker image (for example, ruby:2.7):
alpine:latest
Runner registered successfully. Feel free to start it, but if it's running already the config
should be automatically reloaded!



Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
PS C:\Users\xavie>
```

Figure A.2 Enregistrement du GitLab Runner dans un référentiel partagé

Lorsque l'enregistrement est terminé, GitLab Runner peut être visible dans le répertoire sur le référentiel comme le montre la figure A.3. GitLab Runner est alors lié mais pas encore fonctionnel.

Available specific runners

 #38 (cz7Fxy9N) 

  Remove runner

This is a GitLab runner on Docker to test the operation of HEI's GitLab.

docker test



Figure A.3 GitLab Runner lié



Il reste à démarrer l'image de Docker sur la machine physique afin que le service puisse fonctionner. Pour ce faire, la commande à exécuter est la suivante :

```
docker run -d --name gitlab-runner --restart always -v  
→ /srv/gitlab-runner/config:/etc/gitlab-runner -v  
→ /var/run/docker.sock:/var/run/docker.sock gitlab/gitlab-runner:latest
```

Comme le montre la figure A.4, le [pipeline](#) peut désormais se lancer à l'aide du programme installé virtuellement dans un conteneur de Docker.

Available specific runners

 #38 (cz7Fxy9N) 

  Remove runner

This is a GitLab runner on Docker to test the operation of HEI's GitLab.

docker test

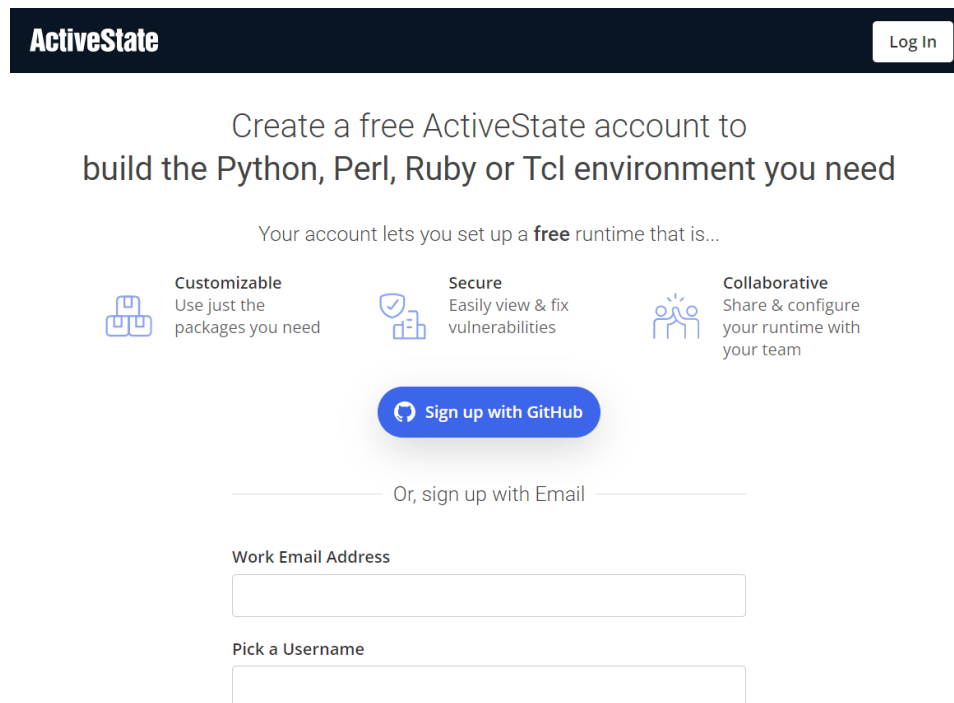
Figure A.4 GitLab Runner opérationnel

B | ActiveTcl

Cette annexe explique simplement l'installation du programme ActiveTcl. Il est indispensable pour exécuter les actions des différents programmes de développement matériel puisqu'ils travaillent tous avec le langage [Tcl](#) en mode lignes de commande. Ce logiciel possède une version gratuite pouvant être installée sur n'importe quel système d'exploitation. Cette version gratuite offre suffisamment d'outils pour ce travail.

B.1 Installer le programme ActiveTcl

En se dirigeant tout d'abord sur le site de l'entreprise ActiveState [\[35\]](#), un compte est requis avant le téléchargement du programme. Sa création peut être faite à partir d'un compte Github (figure [B.1](#)).



The screenshot shows the ActiveState website's account creation page. At the top, there is a dark blue header with the 'ActiveState' logo on the left and a 'Log In' button on the right. Below the header, the main heading reads 'Create a free ActiveState account to build the Python, Perl, Ruby or Tcl environment you need'. A subtext states 'Your account lets you set up a **free** runtime that is...'. Three features are listed with icons: 'Customizable' (stack of boxes icon) with the text 'Use just the packages you need', 'Secure' (shield icon) with 'Easily view & fix vulnerabilities', and 'Collaborative' (two people icon) with 'Share & configure your runtime with your team'. A prominent blue button with a GitHub logo says 'Sign up with GitHub'. Below this, a link says 'Or, sign up with Email'. Underneath are two input fields: 'Work Email Address' and 'Pick a Username'.

Figure B.1 Création d'un compte ActiveState

Dans le cadre de ce travail, un compte a été créé depuis une adresse e-mail. Lorsque ceci est fait, une page ressemblant à la figure [B.2](#) s'ouvre. Il faut sélectionner le langage [Tcl](#) avec la version recommandée et le système d'exploitation Windows.

Annexe B. ActiveTcl

Select a language

Python Perl Ruby Beta ⓘ Tcl

Select a Tcl version

Recommended

8.6.12 8.6.11.1 ⓘ Need other versions?

Choose operating systems

We'll build a runtime from source for each selected operating system. ⓘ Creating a multi-OS project?

☐ Linux Ubuntu 20.10 groovy

☒ Windows 10

☐ MacOS 10.15 Catalina

Figure B.2 Sélection du programme et des paramètres

En descendant dans la page web, il reste à sélectionner le bouton "Finish Creating Project" comme le montre la figure B.3.

Project visibility ⓘ Working on something sensitive?

☒ Public The details of this runtime will be publicly visible.

☐ Private Only you can see it.

What would you like to do next?

Add Packages before Installing Tcl
We'll build the packages from source and they'll be included when you install Tcl.
[Search Package Catalog](#)

Start Using Tcl
Install Tcl now and install packages later on the command line using the State Tool.
[Finish Creating Project](#)

Figure B.3 Sélection de la suite des paramètres

Une petite page identique à la figure B.4 s'ouvre et il faut la quitter en cliquant sur la croix en haut à droite de la figure. Il s'agit de la page pour télécharger le programme au format ZIP.

B.1 Installer le programme ActiveTcl

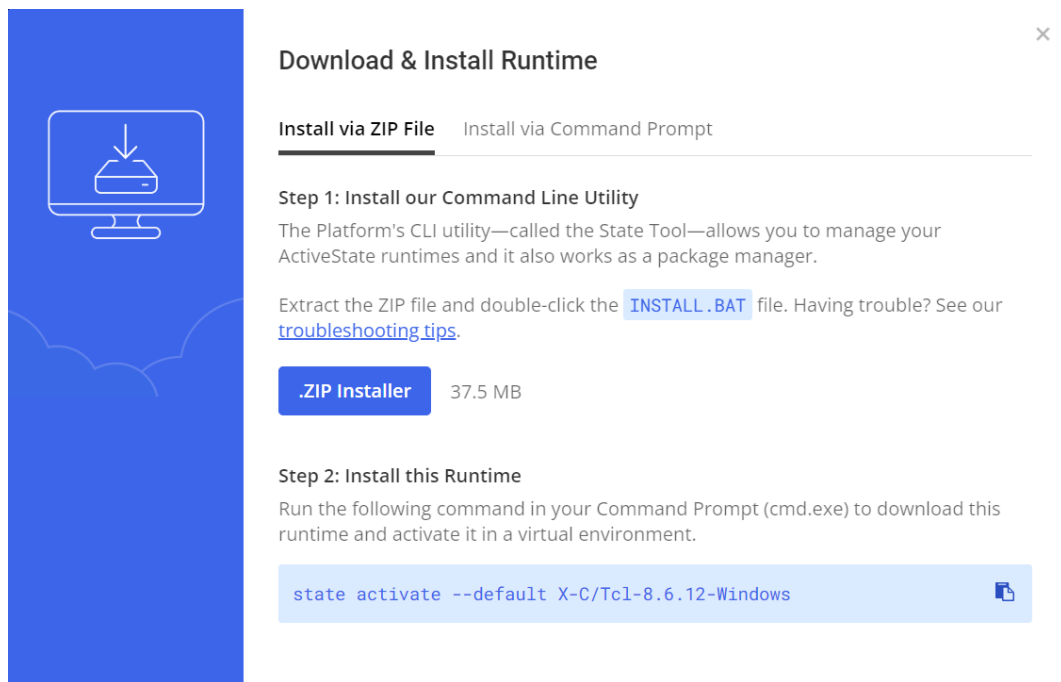


Figure B.4 Téléchargement d'un fichier ZIP

Lorsque la petite fenêtre est quittée, la page ressemble à celle de la figure B.5. Elle met à disposition un exécutable qui doit être téléchargé.

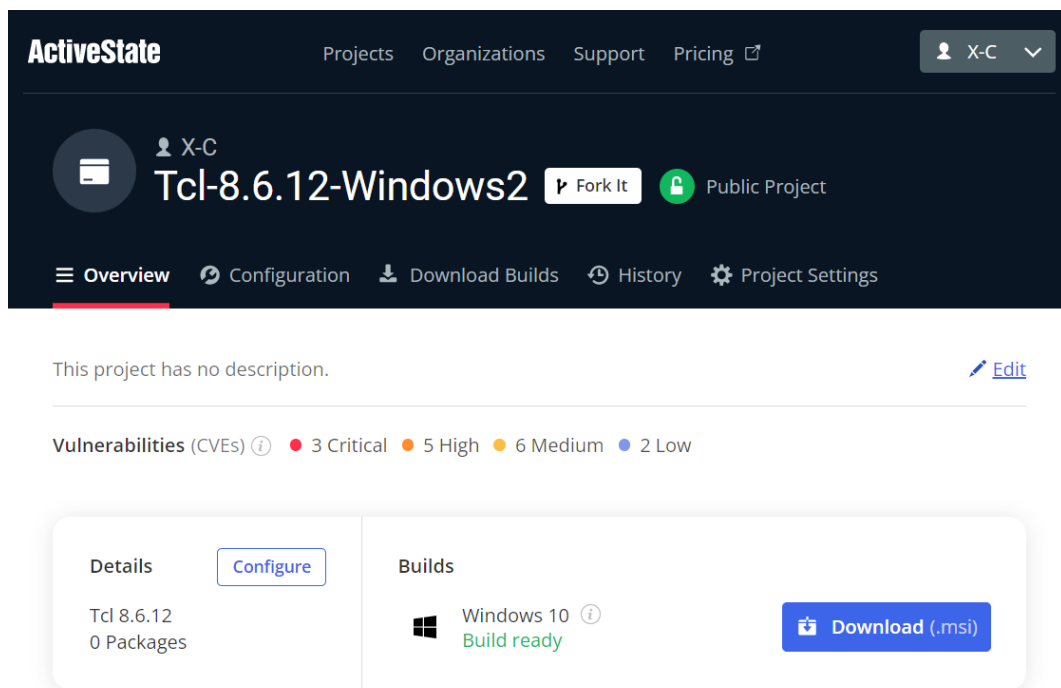


Figure B.5 Téléchargement d'un exécutable

Une fois l'exécutable téléchargé et lancé, un menu de bienvenue apparaît et il faut cliquer sur "Next >". Il faut ensuite accepter les conditions générales comme montré à la figure B.6.

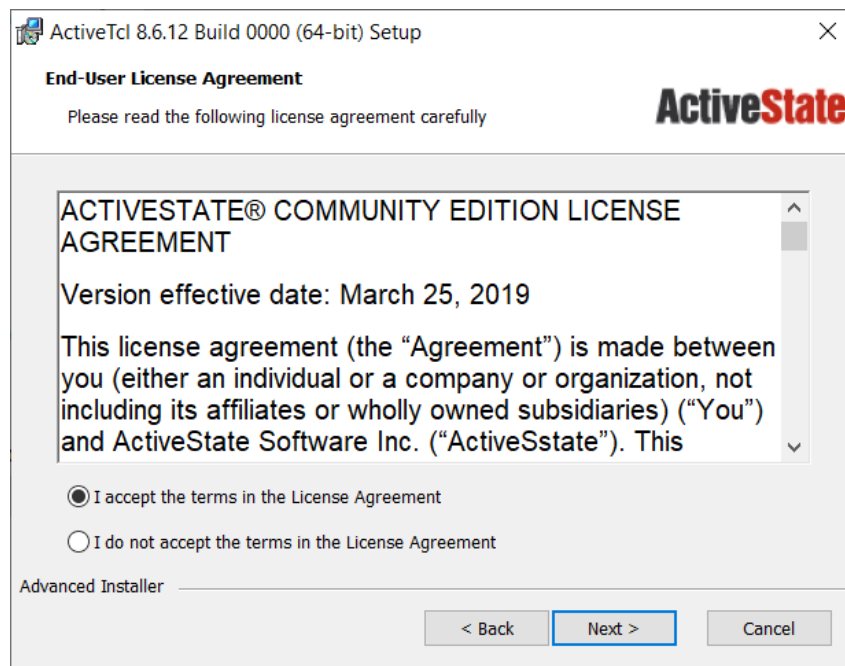


Figure B.6 Fenêtre du contrat de licence

Il est demandé ensuite quel type d'installation doit être faite. Il est recommandé de sélectionner "Typical" (figure B.7).

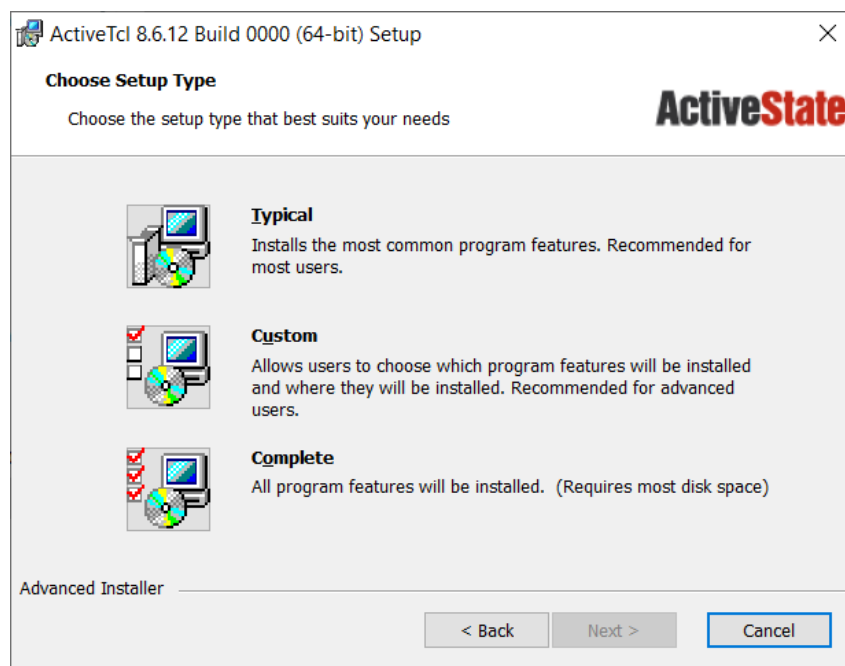


Figure B.7 Fenêtre des types de configuration

Des options de configuration sont par la suite demandées (figure B.8). Il faut que la case "Add Tcl to the PATH environment variable" soit cochée pour que le programme ActiveTcl soit reconnu dans tout le système d'exploitation Windows.

B.1 Installer le programme ActiveTcl

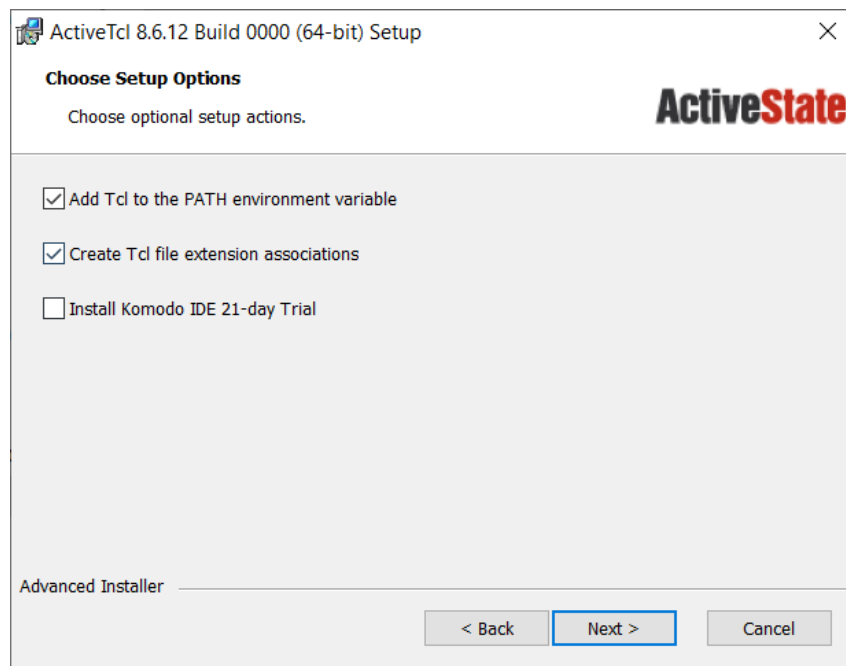


Figure B.8 Fenêtre des options de configuration

L'installation peut ensuite être lancée. Lorsqu'elle est terminée, un message informe la réussite de l'installation (figure B.9).



Figure B.9 Installation terminée

C | Sous-modules

Dans le monde du contrôle de version Git [36], l'intégration de sous-modules dans un répertoire n'est pas simple. Une explication sur l'ajout de sous-modules dans un répertoire vaut la peine d'être détaillée. D'autres informations complémentaires sont aussi apportées.

C.1 Intégrer un sous-module dans un répertoire

Dans cette explication, deux répertoires sont créés dans la plateforme d'hébergement GitLab [5] comme le montre la figure C.1. Il y a le répertoire principal nommé "Projet" et le répertoire secondaire appelé "Scripts". L'idée est de vouloir faire de "Scripts" un sous-module du dépôt "Projet"

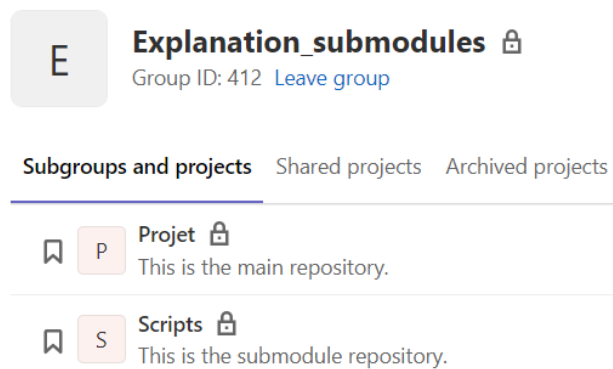


Figure C.1 Création de deux projets dans GitLab

Le répertoire "Projet" est d'abord cloné localement sur la machine physique (figure C.2). La commande est la suivante :

```
git clone <projet_url>
```

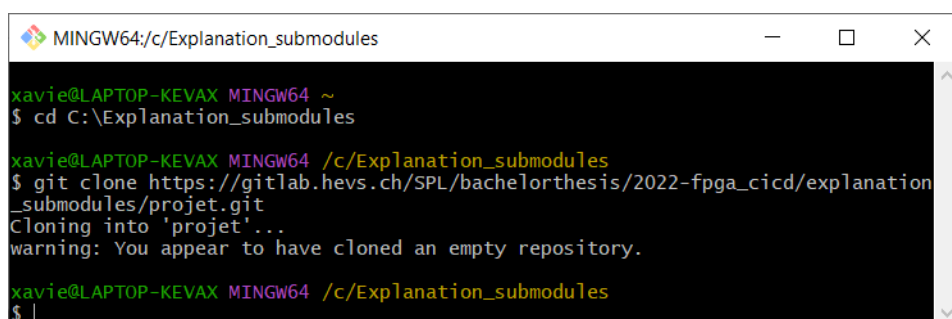


Figure C.2 Clonage local du répertoire "Projet"

Annexe C. Sous-modules

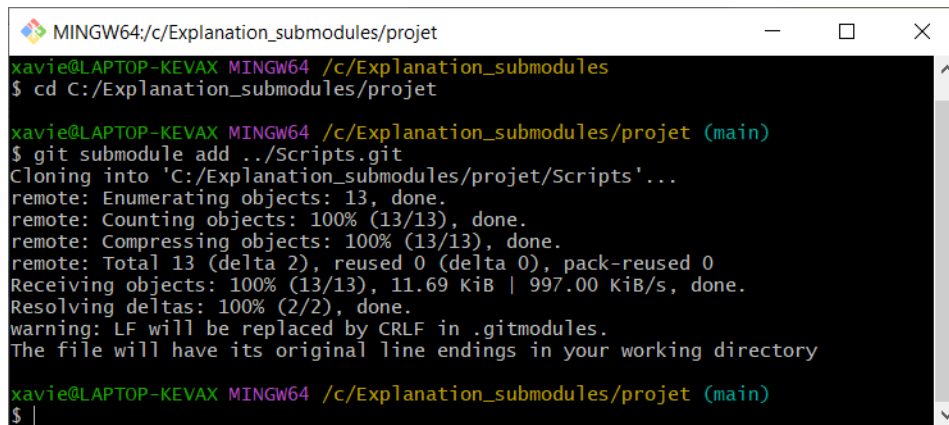
Il faut ensuite se déplacer dans le répertoire local pour pouvoir ajouter un sous-module avec la commande :

```
cd C:/path/to/repo
```

L'ajout d'un sous-module se fait à partir de la commande suivante :

```
git submodule add <submodule_url>
```

Dans cet exemple, le lien vers le sous-module "Scripts" doit être spécifié. Puisque "Scripts" est un répertoire privé, le lien ne donne pas l'accès. Il faut passer par un lien relatif comme le montre la figure C.3. Les deux dépôts se situent au même niveau dans GitLab. A partir de "Projet", il suffit donc de descendre d'un étage avec les doubles points et remonter dans le répertoire "Scripts" en spécifiant son nom.



```
MINGW64/c/Explanation_submodules/projet
xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules
$ cd C:/Explanation_submodules/projet

xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules/projet (main)
$ git submodule add ../Scripts.git
Cloning into 'C:/Explanation_submodules/projet/Scripts'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 13 (delta 2), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (13/13), 11.69 KiB | 997.00 KiB/s, done.
Resolving deltas: 100% (2/2), done.
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory

xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules/projet (main)
$ |
```

Figure C.3 Ajout du sous-module dans le répertoire local "Projet"

Le sous-module est à présent cloné dans le répertoire locale "Projet". La figure C.4 le confirme.

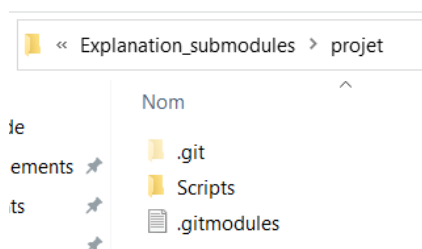


Figure C.4 Vérification du sous-module dans le répertoire local "Projet"

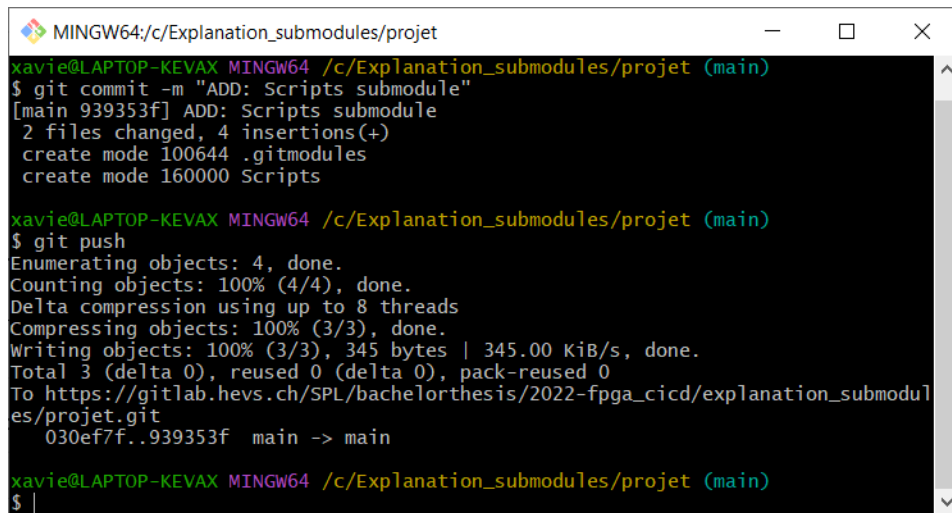
Il reste à pousser ce dépôt local "Projet" sur le référentiel partagé. En premier lieu, il faut faire un "commit" des changements locaux :

```
git commit -m "Message"
```

C.1 Intégrer un sous-module dans un répertoire

Seulement après, les changements peuvent être poussés :

```
git push
```



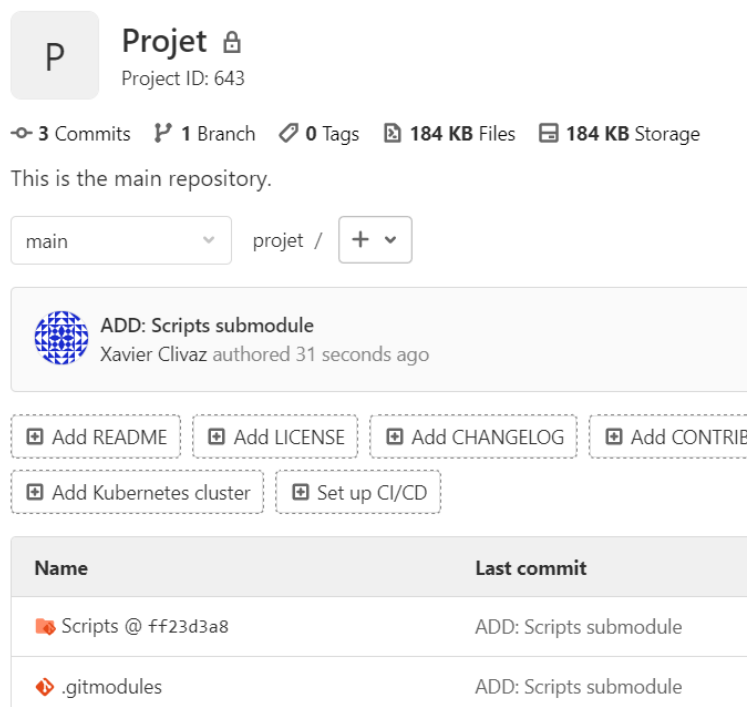
```
MINGW64/c/Explanation_submodules/projet
xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules/projet (main)
$ git commit -m "ADD: Scripts submodule"
[main 939353f] ADD: Scripts submodule
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 Scripts


xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules/projet (main)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 345.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://gitlab.hevs.ch/SPL/bachelorthesis/2022-fpga_cicd/explanation_submodules/projet.git
   030ef7f..939353f  main -> main

xavie@LAPTOP-KEVAX MINGW64 /c/Explanation_submodules/projet (main)
$ |
```

Figure C.5 Poussée du répertoire local "Projet" au référentiel partagé

La vérification du sous-module peut à présent être faite dans GitLab. La figure C.6 montre avec succès l'intégration de "Scripts" dans "Projet".




Projet  Project ID: 643

3 Commits 1 Branch 0 Tags 184 KB Files 184 KB Storage

This is the main repository.

main projet / +

 **ADD: Scripts submodule**
Xavier Clivaz authored 31 seconds ago

Add README

Add LICENSE

Add CHANGELOG

Add CONTRIB

Add Kubernetes cluster

Set up CI/CD



Name	Last commit
 Scripts @ ff23d3a8	ADD: Scripts submodule
 .gitmodules	ADD: Scripts submodule

Figure C.6 Vérification du sous-module inclus dans le dépôt "Projet" sur GitLab

C.2 Cloner un répertoire avec un ou plusieurs sous-modules

Pour cloner un répertoire contenant déjà un ou plusieurs sous-modules, une seule commande peut être utilisée :

```
git clone --recurse-submodules <repo_url>
```

Cette commande clone le dépôt et initialise et met à jour tous les sous-modules présents puisque sa méthode est récursive.

C.3 Mettre à jour un sous-module dans un répertoire

Pour exemple, le dépôt "Scripts" a été mis à jour sur la plateforme GitLab. Il faut à présent mettre à jour le répertoire "Projet" qui contient ce sous-module. La première étape consiste à fusionner cette mise à jour au répertoire local par la commande suivante :

```
git submodule update --remote --merge
```

Pour pouvoir mettre à jour le dépôt "Projet" sur GitLab, il faut d'abord ajouter les changements locaux avec la commande suivante :

```
git add <submodule_file>
```

Ensuite un "commit" doit être fait :

```
git commit -m "Message"
```

Pour terminer, il reste à pousser cette mise à jour :

```
git push
```


D | Protocole de test - Projet chronomètre

S : Sunny case (Meilleur cas)
W : Worst case (Pire cas)
INIT : Job initialisation
GEN : Job génération VHDL
SIM : Job simulation
SYN : Job synthèse
HW: Hardware (matériel)

Test ID	S_INIT_1
Description	Le test consiste à vérifier que le job d'initialisation fonctionne correctement lorsque le programme ActiveTcl est installé.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Le programme ActiveTcl est installé sur la machine physique du GitLab Runner.
Personel	Xavier Clivaz
Résultat	Le job d'initialisation s'est déroulé avec succès.
Conclusions	TEST OK : Le job a trouvé le programme. De plus, il a cloné correctement le dépôt sur la machine physique y compris le sous-module "Scripts" présent.
Mesures	

Test ID	S_INIT_2
Description	Le test consiste à vérifier que le job d'initialisation fonctionne correctement lorsque le dépôt ne contient aucun sous-module.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Aucun sous-module n'est présent dans le dépôt.
Personel	Xavier Clivaz
Résultat	Le job d'initialisation s'est déroulé avec succès.
Conclusions	TEST OK : Le job a lancé les commandes Git propre aux sous-modules sans affecter le répertoire et le pipeline.
Mesures	

Test ID	W_INIT_1
Description	Le test consiste à vérifier que le job d'initialisation soit interrompu correctement en enlevant le programme ActiveTcl.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Le programme ActiveTcl a été déplacé sur la machine physique du GitLab Runner.
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant qu'aucune installation n'est valide pour l'exécuteur "tclsh".
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_INIT_2
Description	Le test consiste à vérifier que le job d'initialisation ne se lance pas lorsque le développeur n'a pas spécifié le bon service de Gitlab.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Dans ce script, les étiquettes faisant références à un service GitLab ont été modifiées aléatoirement pour ne faire référence à aucun service.
Personel	Xavier Clivaz
Résultat	Le pipeline reste bloqué à l'entrée du job d'initialisation avec un message avertissant que les étiquettes ne font références à aucun service de GitLab Runner.
Conclusions	TEST OK : Le job d'initialisation est resté bloqué.
Mesures	

Test ID	S_GEN_1
Description	Le test consiste à vérifier que le job de génération VHDL fonctionne correctement en activant la simulation et la synthèse.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Les variables d'activation de la simulation "SIMULATION_EN" et de la synthèse "SYNTHESIS_EN" valent "1".
Personel	Xavier Clivaz
Résultat	Le job de génération VHDL s'est déroulé avec succès. Le journal a été sauvé dans les artefacts.
Conclusions	TEST OK : Tous les fichiers VHDL de simulation et de synthèse ont été générés correctement. Ceci peut être confirmé par le contenu du journal. Ils ont aussi été concaténés et mis à jour par le script de hâchage.
Mesures	

Test ID	S_GEN_2
Description	Le test consiste à vérifier que le job de génération VHDL fonctionne correctement en désactivant la synthèse.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la synthèse "SYNTHESIS_EN" vaut "0".
Personel	Xavier Clivaz
Résultat	Le job de génération VHDL s'est déroulé avec succès. Le journal a été sauvé dans les artefacts.
Conclusions	TEST OK : Uniquement les fichiers VHDL de simulation ont été générés. Ceci peut être confirmé par le contenu du journal. Ils ont aussi été concaténés et mis à jour par le script de hâchage.
Mesures	

Test ID	S_GEN_3
Description	Le test consiste à vérifier que le job de génération VHDL fonctionne correctement en désactivant la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SIMULATION_EN" vaut "0".
Personel	Xavier Clivaz
Résultat	Le job de génération VHDL s'est déroulé avec succès. Le journal a été sauvé dans les artefacts.
Conclusions	TEST OK : Uniquement les fichiers VHDL de synthèse ont été générés. Ceci peut être confirmé par le contenu du journal. Ils ont aussi été concaténés et mis à jour par le script de hâchage.
Mesures	

Test ID	S_GEN_4
Description	Le test consiste à vérifier que le job de génération VHDL fonctionne correctement en désactivant la simulation et la synthèse.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Les variables d'activation de la simulation "SIMULATION_EN" et de la synthèse "SYNTHESIS_EN" valent "0".
Personel	Xavier Clivaz
Résultat	Le job de génération VHDL s'est déroulé avec succès avec un message d'avertissement signalant qu'aucun fichier n'a été généré car la simulation et la synthèse ne sont pas activées.
Conclusions	TEST OK : Aucun fichier VHDL n'a été généré.
Mesures	

Test ID	W_GEN_1
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en mettant un chemin des scripts erroné.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "SCRIPTS_DIR" valant "\$CI_PROJECT_DIR/XYZ"
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que le fichier GenVHDL_Workflow_main.tcl ne peut pas être lu.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est apparu. Il indique uniquement que le chemin ou le fichier est introuvable.
Mesures	

Test ID	W_GEN_2
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en n'incluant pas la tâche de concaténation dans les préférences.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Le fichier "concatenate.tsk" a été supprimé des préférences.
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que la tâche de concaténation est manquante ou que le chemin des préférences est erroné.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_GEN_3
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en mettant un nom du projet HDS erroné.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "HDP" valant "\$CI_PROJECT_DIR/Prefs/wrong_name.hdp".
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant qu'une erreur est apparue dans la génération au niveau simulation et qu'il faut vérifier le nom de la librairie, de l'entité, de l'architecture et du projet.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est apparu. Le message n'est malheureusement pas très précis.
Mesures	Il vaudrait mieux connaître quel nom de variable est erroné si possible.

Test ID	W_GEN_4
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en mettant un nom des préférences HDS erroné.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "PREFS_DIR" valant "\$CI_PROJECT_DIR/wrong_prefs".
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que la tâche de concaténation est manquante ou que le chemin des préférences est erroné.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_GEN_5
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en n'incluant pas les librairies externes à HDS que demande le projet.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "REQUIRE_LIBS" valant "0".
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que la génération de la simulation s'est terminée avec des erreurs.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est apparue. L'erreur est compréhensible dans le journal généré par HDS. Il suffit de le parcourir la console du job pour remarquer que les erreurs font références à tous les composants du circuit faisant partis des librairies externes.
Mesures	

Test ID	W_GEN_6
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en ne spécifiant pas la librairie de simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "TESTBENCH_LIB" ayant été supprimée.
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que la variable "TESTBENCH_LIB" est manquante dans le script <i>.gitlab-ci.yml</i> .
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_GEN_7
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement en spécifiant une mauvaise entité de synthèse.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "PROGRAM_ENTITY" valant "wrong_entity".
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant qu'une erreur est apparue dans la génération au niveau synthèse et qu'il faut vérifier le nom de la librairie, de l'entité, de l'architecture et du projet.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est apparu. Le message n'est malheureusement pas très précis.
Mesures	Il vaudrait mieux connaître quel nom de variable est erroné si possible.

Test ID	W_GEN_8
Description	Le test consiste à vérifier que le job de génération VHDL soit interrompu correctement lorsqu'une erreur provient de la conception du circuit.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Dans la librairie "Board" et l'entité "EIN_chrono" du projet HDS, un fil ("restartSynch") a été déconnecté de l'entrée d'un bloc (entrée "restart" du bloc "chronoCircuit").
Personel	Xavier Clivaz
Résultat	Le pipeline est interrompu avec un message signalant que la première génération (simulation) s'est terminée avec succès et que la seconde (synthèse) avec des erreurs.
Conclusions	TEST OK : Les jobs suivants n'ont pas été lancés et le message d'erreur est suffisamment parlant pour comprendre l'erreur. Il suffit de passer à travers la console du job pour trouver précisément l'erreur.
Mesures	

Test ID	S_SIM_1
Description	Le test consiste à vérifier que le job de simulation fonctionne correctement en activant la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SIMULATION_EN" vaut "1".
Personel	Xavier Clivaz
Résultat	Le job de simulation s'est déroulé avec succès. Les journaux et le fichier contenant les signaux stimulés ont été sauvés dans les artefacts.
Conclusions	TEST OK : La simulation s'est déroulée correctement. Ceci peut être confirmé par le contenu des journaux. Les signaux peuvent être vus manuellement pour vérifier leur comportement.
Mesures	

Test ID	S_SIM_2
Description	Le test consiste à vérifier que le job de simulation ne démarre pas en désactivant la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SIMULATION_EN" vaut "0".
Personel	Xavier Clivaz
Résultat	Le job de simulation ne s'est pas lancé et le pipeline n'a pas été interrompu.
Conclusions	TEST OK : Le job de simulation a été ignoré.
Mesures	

Test ID	W_SIM_1
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement en spécifiant un mauvais chemin au fichier <i>.do</i> de ModelSim.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "DO_FILE" valant "\$SIMULATION_DIR/wrong.do".
Personel	Xavier Clivaz
Résultat	Le job est interrompu avec un message signalant une erreur dans la simulation. La compilation ne contient aucune erreur. Il n'a pas affecté le job de synthèse qui a pu être réalisé avec succès.
Conclusions	TEST OK : Le message d'erreur est apparu sans être très précis. Dans le journal de simulation généré par ModelSim et affiché sur la console du job, un message informe qu'une commande n'est pas valide avec le chemin du fichier <i>.do</i> . On peut simplement supposer qu'il en est la cause.
Mesures	Il serait mieux qu'une information plus pertinente soit générée par ModelSim.

Test ID	W_SIM_2
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement en raison d'une réinitialisation du répertoire Git local sur la machine physique du GitLab Runner.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "GIT_CLEAN_FLAGS" valant "-i". Cette variable permet de nettoyer les fichiers de manière interactive.
Personel	Xavier Clivaz
Résultat	Le job est interrompu avec un message signalant que la variable "GIT_CLEAN_FLAGS" doit valoir "none". Il n'a pas affecté le job de synthèse qui a pu être réalisé avec succès.
Conclusions	TEST OK : Le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_SIM_3
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement lorsque le développeur n'a pas spécifié le bon banc de test.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis les variables "TESTBENCH_LIB" valant "Chronometer", "TESTBENCH_ENTITY" valant "chronoCircuit" et "TESTBENCH_ARCH" valant "masterVersion.bd". Il s'agit de véritable librairie, entité et architecture ne faisant pas référence à un banc de test.
Personel	Xavier Clivaz
Résultat	Le job de génération VHDL a été lancé avec succès puis le job de simulation a été interrompu avec un message signalant une erreur du chargement du design.
Conclusions	TEST OK : Les fichiers VHDL ont pu être générés puisque la structure spécifiée est existante. Le message d'erreur n'est pas très précis. Il y a un manque d'information dans l'erreur que signale ModelSim.
Mesures	

Test ID	W_SIM_4
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement lorsqu'une erreur survient dans la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Une assertion a été ajoutée dans le code VHDL du banc de test pour émettre volontairement une erreur ("error").
Personel	Xavier Clivaz
Résultat	Le job de simulation a été interrompu après la simulation avec une erreur reportée.
Conclusions	TEST OK : Le message d'erreur n'est pas précis. Le détail de l'erreur est retrouvable sur la console du job. Puisque l'assertion était une erreur ("error"), la simulation s'est malgré tout terminée à la fin.
Mesures	

Test ID	W_SIM_5
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement lorsqu'un échec survient dans la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Une assertion a été ajoutée dans le code VHDL du banc de test pour émettre volontairement un échec ("failure").
Personel	Xavier Clivaz
Résultat	Le job de simulation a été interrompu durant la simulation avec une erreur reportée.
Conclusions	TEST OK : Le message d'erreur n'est pas précis. Le détail de l'erreur est retrouvable sur la console du job. Puisque l'assertion était un échec ("failure"), la simulation ne s'est pas terminée à la fin.
Mesures	

Test ID	W_SIM_6
Description	Le test consiste à vérifier que le job de simulation soit interrompu correctement lorsqu'un signal inconnu a été ajouté à la simulation.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Un signal appelé "wrong" a été ajouté dans le fichier <i>.do</i> de ModelSim.
Personel	Xavier Clivaz
Résultat	Le job de simulation a été interrompu après la simulation avec trois erreurs reportées.
Conclusions	TEST OK : Le message d'erreur n'est pas précis. Le détail de l'erreur est retrouvable sur la console du job. Il indique que l'objet "wrong" n'a pas été trouvé. Cependant, ModelSim indique qu'il y a trois erreurs alors que seule celle décrite ici est présente.
Mesures	

Test ID	S_SYN_1
Description	Le test consiste à vérifier que le job de synthèse fonctionne correctement en activant la synthèse et en apportant un projet ISE existant.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SYNTHESIS_EN" vaut "1" et la variable "ISE_PROJECT_PATH" comporte le chemin avec le nom du projet.
Personel	Xavier Clivaz
Résultat	Le job de synthèse s'est déroulé avec succès. Tous les artefacts ont été sauvés.
Conclusions	TEST OK : La synthèse s'est déroulée correctement. Ceci peut être confirmé par le contenu du journal en apercevant que le projet ISE apporté a bien été sélectionné. Le fichier bitstream peut être programmé manuellement sur la carte de développement.
Mesures	

Test ID	S_SYN_2
Description	Le test consiste à vérifier que le job de synthèse fonctionne correctement en activant la synthèse et en n'apportant pas un projet ISE existant.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SYNTHESIS_EN" vaut "1" et la variable "ISE_PROJECT_PATH" vaut "". "ISE_PROJECT_PATH" est considérée comme inexistante.
Personel	Xavier Clivaz
Résultat	Le job de synthèse s'est déroulé avec succès. Tous les artefacts ont été sauvés.
Conclusions	TEST OK : La synthèse s'est déroulée correctement. Ceci peut être confirmé par le contenu du journal en apercevant qu'un nouveau projet ISE a été créé. Le fichier bitstream peut être programmé manuellement sur la carte de développement.
Mesures	

Test ID	S_SYN_3
Description	Le test consiste à vérifier que le job de synthèse ne démarre pas en désactivant la synthèse.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. La variable d'activation de la simulation "SYNTHESIS_EN" vaut "0".
Personel	Xavier Clivaz
Résultat	Le job de synthèse ne s'est pas lancé et le pipeline n'a pas été interrompu.
Conclusions	TEST OK : Le job de synthèse a été ignoré.
Mesures	

Test ID	W_SYN_1
Description	Le test consiste à vérifier que le job de synthèse soit interrompu correctement en spécifiant un mauvais chemin à un projet ISE existant.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "ISE_PROJECT_PATH" valant "\$CI_PROJECT_DIR/Board/ise/wrong.xise".
Personel	Xavier Clivaz
Résultat	Le job est interrompu avec un message signalant que le projet est introuvable et qu'il faut vérifier la variable du projet dans le script <i>.gitlab-ci.yml</i> .
Conclusions	TEST OK : Le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_SYN_2
Description	Le test consiste à vérifier que le job de synthèse soit interrompu correctement en spécifiant un mauvais chemin au fichier de contrainte UCF pour la création d'un nouveau projet ISE.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement hormis la variable "UCF_PATH" valant "\$CI_PROJECT_DIR/Board/concat/wrong.ucf" et la variable "ISE_PROJECT_PATH" étant ignorée.
Personel	Xavier Clivaz
Résultat	Le job est interrompu avec un message signalant que le fichier UCF est introuvable et qu'il faut vérifier la variable du fichier UCF dans le script <i>.gitlab-ci.yml</i> .
Conclusions	TEST OK : Le message d'erreur est suffisamment parlant pour comprendre l'erreur.
Mesures	

Test ID	W_SYN_3
Description	Le test consiste à vérifier que le job de synthèse soit interrompu correctement en apportant volontairement une erreur de syntaxe dans le fichier de contrainte UCF pour la création d'un nouveau projet ISE.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement en ignorant la variable "ISE_PROJECT_PATH". Un ";" a été supprimé en fin de ligne d'une assignation de connexion dans le fichier de contrainte UCF.
Personel	Xavier Clivaz
Résultat	Le job est interrompu avec un message signalant que l'implémentation a échouée.
Conclusions	TEST OK : Le message est trop vague pour connaître l'erreur exact. Il n'est pas simple de remonter à l'erreur.
Mesures	Il serait mieux qu'une information plus pertinente soit générée par ISE.

Test ID	W_SYN_4
Description	Le test consiste à vérifier que le job de synthèse se déroule avec succès malgré qu'un artefact n'est pas trouvable.
Prescription de test	Toutes les variables du script <i>.gitlab-ci.yml</i> sont configurées correctement. Seul le chemin de l'artefact du fichier bitstream a été modifié pour que GitLab ne le retrouve pas.
Personel	Xavier Clivaz
Résultat	Le job s'est déroulé avec succès. Un message d'attention indique qu'aucun fichier ne correspond à l'artefact modifié dans le <i>.gitlab-ci.yml</i> .
Conclusions	TEST OK : Le message ne commet aucune erreur.
Mesures	

Test ID	S_HW_1
Description	Le test consiste à vérifier que le fichier bitstream programme correctement la carte de développement FPGA avec un projet ISE existant.
Prescription de test	Le pipeline est lancé avec toutes les variables du script <i>.gitlab-ci.yml</i> configurées correctement. Le fichier bitstream est généré et récupéré dans les artefacts. Il est programmé manuellement dans la carte de développement FPGA.
Personel	Xavier Clivaz
Résultat	Le fichier de configuration a été programmé avec succès. Les boutons fonctionnait correctement avec le chronomètre mécanique.
Conclusions	TEST OK : La synthèse a généré correctement le fichier bitstream à partir d'un projet ISE existant.
Mesures	

Test ID	S_HW_2
Description	Le test consiste à vérifier que le fichier bitstream programme correctement la carte de développement FPGA sans projet ISE existant.
Prescription de test	Le pipeline est lancé avec toutes les variables du script <i>.gitlab-ci.yml</i> configurées correctement. Le fichier bitstream est généré et récupéré dans les artefacts. Il est programmé manuellement dans la carte de développement FPGA.
Personel	Xavier Clivaz
Résultat	Le fichier de configuration a été programmé avec succès. Les boutons fonctionnait correctement avec le chronomètre mécanique.
Conclusions	TEST OK : La synthèse a généré correctement le fichier bitstream à partir d'un projet ISE créé dans le pipeline.
Mesures	

Test ID	W_HW_1
Description	Le test consiste à vérifier que le fichier bitstream ne programme pas correctement la carte de développement FPGA sans projet ISE existant et avec le fichier de contrainte mal configuré.
Prescription de test	Le pipeline est lancé avec toutes les variables du script <i>.gitlab-ci.yml</i> configurées correctement. Le fichier de contrainte UCF est modifié au niveau d'un bouton relié à un mauvais connecteur. Le fichier bitstream est généré et récupéré dans les artefacts. Il est programmé manuellement dans la carte de développement FPGA.
Personel	Xavier Clivaz
Résultat	Le fichier de configuration a été programmé avec succès. Le chronomètre a démarré sans que le bouton "start" ait été appuyé.
Conclusions	TEST OK : La synthèse a généré un mauvais fichier bitstream à partir d'un projet ISE créé dans le pipeline.
Mesures	

Bibliographie

- [1] *The One DevOps Platform / GitLab*. GitLab. URL : <https://about.gitlab.com/> (visité le 01/06/2022).
- [2] *GitLab Runner*. GitLab. URL : <https://docs.gitlab.com/runner/> (visité le 15/08/2022).
- [3] *Xilinx - Adaptable. Intelligent.* Xilinx. URL : <https://www.xilinx.com> (visité le 27/06/2022).
- [4] Marc BERGUERAND. « Gitlab Intégration Continue avec «On Target Testing» ». Thèse de Bachelor. Haute Ecole d'Ingénierie, 2019, p. 13. URL : <https://doc.rero.ch/record/329441> (visité le 30/05/2022).
- [5] *GitLab CI/CD*. GitLab. URL : <https://docs.gitlab.com/ee/ci/> (visité le 30/05/2022).
- [6] *Jenkins*. Jenkins. URL : <https://www.jenkins.io/> (visité le 30/05/2022).
- [7] *Bitbucket Pipelines - Continuous Delivery*. Atlassian Bitbucket. URL : <https://bitbucket.org/product/features/pipelines> (visité le 30/05/2022).
- [8] *Travis CI - Test and Deploy Your Code with Confidence*. Travis CI. URL : <https://travis-ci.org/> (visité le 30/05/2022).
- [9] *HDL Designer*. Siemens. URL : <https://eda.sw.siemens.com/en-US/ic/hdl-designer/> (visité le 07/06/2022).
- [10] Mentor GRAPHICS. *HDL Designer Series™ Tcl Reference Manual*. 2019.
- [11] Carlos SERRANO et al. « Hardware-in-the-Loop Testing of Accelerator Firmware ». In : *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems* (2020). ISBN : 9783954502097, p. 3. ISSN : 2226-0358. DOI : [10.18429/JACoW-ICALEPCS2019-TUAPP01](https://doi.org/10.18429/JACoW-ICALEPCS2019-TUAPP01). (Visité le 01/06/2022).
- [12] Alberto MARTÍNEZ CUESTA. « Diseño e implementación de un jammer configurable en FPGA ». Thèse de Master. Université polytechnique de Valence, 2020, p. 13...22. (Visité le 31/05/2022).
- [13] *VUnit : a test framework for HDL — VUnit documentation*. VUnit. URL : <https://vunit.github.io/index.html> (visité le 02/06/2022).
- [14] *Welcome to cocotb's documentation! — cocotb 1.6.2 documentation*. cocotb. URL : <https://docs.cocotb.org/en/stable/index.html> (visité le 02/06/2022).
- [15] Mentor GRAPHICS. *HDL Designer Series™ User Manual*. 2019. (Visité le 24/06/2022).
- [16] *GHDL : free and open-source analyzer, compiler, simulator and (experimental) synthesizer for VHDL*. GHDL. URL : <https://ghdl.github.io/ghdl/> (visité le 31/05/2022).
- [17] *ModelSim*. Siemens. URL : <https://eda.sw.siemens.com/en-US/ic/modelsim/> (visité le 31/05/2022).

Bibliographie

- [18] Peter URAN. « Design of an FPGA-based Data Acquisition System for a Shore-based Maritime Radar Network ». Thèse de Master. Université norvégienne de sciences et de technologie, 2021, p. 39...41. (Visité le 01/06/2022).
- [19] Martin JEŘÁBEK. « Open-source and Open-hardware CAN FD Protocol Support ». Thèse de Master. Université technique de Prague, 2019, p. 29...40. (Visité le 02/06/2022).
- [20] Welcome to GTKWave. GTKWave. URL : <http://gtkwave.sourceforge.net> (visité le 31/05/2022).
- [21] TAHA, Bilal IQBAL et Abdul WASAY MUDASSER. « Design and Development of Soft Core Microprocessor using Open-Source EDA tools ». In : *Journal of Xi'an University of Architecture Technology* (2020). ISSN : 1006-7930. (Visité le 01/06/2022).
- [22] Test Coverage Program. Gcov. URL : <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html> (visité le 02/06/2022).
- [23] LCOV - the LTP GCOV extension. Lcov. URL : <http://ltp.sourceforge.net/coverage/lcov.php> (visité le 02/06/2022).
- [24] Etienne MESSERLI. « Outils EDA ». In : (2020). URL : http://reds.heig-vd.ch/share/cours/CSN/Pres_CSN_02_Tools_EDA.pdf (visité le 08/06/2022).
- [25] ISE Design Suite. Xilinx. URL : <https://www.xilinx.com/products/design-tools/ise-design-suite.html> (visité le 08/06/2022).
- [26] Vivado ML. Xilinx. URL : <https://www.xilinx.com/products/design-tools/vivado.html> (visité le 08/06/2022).
- [27] Quartus Prime. Intel. URL : <https://www.intel.fr/content/www/fr/fr/software/programmable/quartus-prime/overview.html> (visité le 08/06/2022).
- [28] XILINX. *Command Line Tools User Guide*. 2013.
- [29] XILINX. *Vivado Design Suite Tcl Command Reference Guide*. 2022.
- [30] INTEL. *Intel® Quartus® Prime Pro Edition User Guide*. 2022.
- [31] Docker : Home. Docker. URL : <https://www.docker.com> (visité le 15/08/2022).
- [32] Docker Hub Container Image Library. Docker. URL : <https://hub.docker.com> (visité le 15/08/2022).
- [33] Install GitLab Runner on Windows. GitLab. URL : <https://docs.gitlab.com/runner/install/windows.html> (visité le 10/06/2022).
- [34] Git - Download for Windows. Git. URL : <https://git-scm.com/download/win> (visité le 10/06/2022).
- [35] Download Tcl For Free. ActiveState. URL : <https://www.activestate.com/products/tcl/> (visité le 16/08/2022).
- [36] Git. Git. URL : <https://git-scm.com> (visité le 16/08/2022).
- [37] Mentor GRAPHICS. *Start Here Guide*. 2019. (Visité le 24/06/2022).
- [38] Mentor GRAPHICS. *ModelSim® SE Command Reference Manual*. 2018. (Visité le 30/06/2022).

- [39] *Install Docker Desktop on Windows*. Git. URL : <https://docs.docker.com/desktop/install/windows-install/> (visité le 10/06/2022).

Glossaire

artefact Un artefact est un fichier créé durant la conception logicielle contenant des informations de développement. [xiii](#), [8](#), [21](#), [22](#), [24](#), [26](#), [36](#), [38](#), [40](#), [41](#), [42](#), [43](#), [50](#), [54](#), [59](#), [61](#)

bitstream Un bitstream est un fichier de configuration contenant une succession de bits. [4](#), [7](#), [21](#), [26](#), [51](#), [52](#), [54](#), [55](#), [64](#)

DevOps DevOps est composé de deux noms anglais : **D**evelopment et **O**perations. Ils signifient en français développement et exploitation. [ix](#), [xii](#), [1](#), [6](#), [7](#), [22](#), [26](#)

netlist Une netlist est un schéma électronique. [7](#), [21](#)

pipeline Un pipeline est similaire à un [workflow](#). Il peut aussi être compris comme un tuyau guidant plusieurs [workflows](#). [ix](#), [x](#), [xii](#), [xiii](#), [2](#), [3](#), [4](#), [6](#), [7](#), [8](#), [9](#), [12](#), [15](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [27](#), [28](#), [29](#), [30](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [41](#), [42](#), [44](#), [46](#), [49](#), [50](#), [51](#), [55](#), [56](#), [57](#), [58](#), [59](#), [60](#), [62](#), [63](#), [64](#), [65](#), [67](#), [68](#), [70](#), [71](#)

shell Un shell est un interpréteur de commande qui possède un interface utilisateur en ligne de commande. [xiv](#), [31](#), [46](#), [52](#)

time to market "Time to market" est une expression anglaise désignant la durée de production d'un produit. Elle est traduite en français par "délai de commercialisation". [2](#)

workflow Un workflow désigne en français un **flux de travail**. Il s'agit d'une série de tâches traitant séparément un flux de données. [xii](#), [6](#), [7](#), [20](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [34](#), [35](#), [36](#), [43](#), [51](#), [52](#), [65](#), [68](#), [101](#)

Acronymes

CD distribution continue (**C**ontinuous **D**elivery). ix, x, 1, 2, 4, 5, 6, 7, 13, 15, 26, 27, 29, 34, 51, 53, 55, 57, 67

CI intégration continue (**C**ontinuous **I**ntegration). ix, x, xiii, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 22, 23, 25, 29, 32, 34, 43, 45, 47, 49, 52, 57, 67

EDA automatisation de la conception électronique (**E**lectronic **D**esign **A**utomation). ix, 12, 29, 37, 57, 67, 68

FPGA réseau de portes programmables sur site (**F**ield-**P**rogrammable **G**ate **A**rray). ix, xi, xii, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 24, 26, 27, 28, 30, 54, 56, 57, 64, 65, 67, 68

HDL langage de description matériel (**H**ardware **D**escription **L**anguage). 2, 3, 6, 8, 9, 10, 22, 25

HDS HDL Designer Series. x, xii, xiv, 4, 5, 8, 9, 10, 22, 23, 25, 38, 41, 42, 43, 44, 45, 46, 47, 48, 49, 52, 68

HEI Haute Ecole d'Ingénierie. vii, ix, 2, 4, 8, 17, 20, 25, 29, 30, 37, 48, 57, 58, 67, 68

OVM méthodologie de vérification ouverte (**O**pen **V**erification **M**ethodology). 10

Tcl Tool Command Language. 9, 13, 34, 37, 38, 39, 40, 43, 45, 49, 52, 54, 73

UVM méthodologie de vérification universelle (**U**niversal **V**erification **M**ethodology). 10

VHDL VHSIC-HDL, **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage. ix, x, xii, xiii, xiv, 4, 5, 6, 7, 8, 9, 11, 12, 15, 21, 22, 23, 24, 25, 26, 29, 34, 35, 36, 37, 38, 39, 40, 42, 43, 45, 46, 47, 49, 50, 52, 54, 60, 61, 62, 63, 67, 68

YAML **Y**et **A**nother **M**arkup **L**anguage. 34, 36, 58, 63, 64