Vol.2020-CE-154 No.14
2020/3/14

VHDL演習における初学者支援システムの基礎検討

藤枝 直輝[1,a)]　池田 朋弘[1]　高松 永輔[1]

概要：ハードウェア記述言語（HDL）の習得はディジタル回路の本格的な設計を学ぶために重要である一方，初学者がその演習に取り組むにはいくつかの問題がある．その中には，初学者が陥りやすい記述ミスや落とし穴に対し，適切な指示やフィードバックを行う仕組みに乏しい点がある．本研究では，我々が提案しているフロントエンドツール GGFront を用いた VHDL の演習環境に対して，(1) エラーに対して適切なメッセージや対処法を提示する機構と，(2) 所望のシミュレーション結果が得られているかを自動的に判定する機構を追加することで，この問題の解決に取り組む．本稿では，これらの機構のプロトタイプの設計と実装，動作例について述べる．

A preliminary study on a support system in VHDL practice for beginners

Naoki FUJIEDA[1,a)]　Tomohiro IKEDA[1]　Eisuke TAKAMATSU[1]

Abstract: Although learning a hardware description language (HDL) is important to understand practical digital circuit design, there are several problems for beginners to tackle with its practice. They includes a lack of mechanisms to give appropriate instructions and feedbacks to mistakes and pitfalls, into which beginners are likely to fall. In this study, we deal with this problem by adding two mechanisms to a pratice environment of VHDL using GGFront, a front-end tool we have proposed. The first one presents appropriate messages or advice to errors. The second one automatically checks if the desired simulation result is obtained or not. This report describes design, implementation, and working examples of prototypes of these mechanisms.

1. はじめに

電子工学や情報工学において，応用的なディジタル回路設計に対する理解の重要性が増してきており，その道具として，ハードウェア記述言語（HDL）の演習環境の整備もまた重要な課題になっている．いわゆる Moore の法則の終焉を背景に，ディジタルシステムには領域に特化したハードウェアとソフトウェアとの協調が求められるようになっている [1]．こうした設計のための基礎能力を身につけるには，HDL を用いた抽象的な記述法や階層化設計を身につけ，ある程度大きなディジタル回路の設計と検証を自ら実践してみることが近道である．

本研究室では，HDL の一種である VHDL を用いた演習を手軽に行う環境を整備するために，GGFront [2] というフロントエンドツールを開発している．本ツールの配布用パッケージ [3] では，講義科目の授業外学習として学生が各自で利用することを前提に，必要なシミュレータおよび波形ビューアを同梱している．これにより，ダウンロードしたパッケージを展開し USB メモリ等に持ち歩くことで，自宅や演習室などですぐに記述した VHDL を検証できる．これにより，初学者が HDL の演習に取り組むにあたっての障害の 1 つであった，環境導入の難しさの問題を解消することができた [2]．

しかしながら，初学者にとっての障害はこれだけではない．その 1 つに，初学者が陥りやすい記述ミスや落とし穴に対し，適切な指示やフィードバックを行う仕組みに乏しい点がある．例えば，VHDL では，算術演算関連のパッケージを宣言することなく信号同士の算術演算を行おうとするとエラーとなる．使用した演算子が ‘+’ であれば，このとき GGFront が用いる VHDL シミュレータ

[1] 愛知工業大学
Aichi Institute of Technology
[a)] nfujieda@aitech.ac.jp

である GHDL [4] では，`no function declarations for operator "+"` というメッセージが表示される[*1]．直訳すれば「+演算子の関数宣言がない」であり，慣れた設計者であればこのエラーがパッケージの宣言忘れに起因するとすぐに判断できる．しかし，エラーの発生箇所（演算子が実際に使用されている箇所）と問題の真の要因（パッケージ宣言の箇所）とが離れているため，初学者が適切なアドバイスなしに速やかに問題を解決するのは困難である．また，特に日本語母語話者である初学者の場合，そもそも英語で書かれたエラーメッセージそのものに抵抗感をもつことがある．こうしたエラーに対処することは，問題解決の能力を養う点では有用であるが，対処にあまりに時間を要してしまうと，本来の目的であるディジタル回路の本質的な理解に割ける時間が短くなってしまう．

また，適切なフィードバックという点で言えば，所望の回路が得られているかどうかを即時的に判定する仕組みを備えることも，課題の 1 つである．記述したものに対して即時的なフィードバックを返すことは，学習者のモチベーション維持のために重要である．現状の GGFront を用いた環境において，回路の動作チェックには，シミュレーションの結果得られた波形を目視で確認するか，あらかじめ出力をチェックするようなテストベンチを作成しておく方法を取っている．前者の方法は即時性に疑問が残る．また，後者の方法は演習課題ごとに適切なテストベンチを作成しなければならず，演習課題作成のコストが大きい．VUnit [5] などのテストの実施を自動化するフレームワークは存在するが，テストの作成が人の手によることには変わりない．

本研究ではこうした問題の解決法として，GGFront を用いた環境に (1) エラーに対して適切なメッセージや対処法を提示する機構と，(2) 所望のシミュレーション結果が得られているかを自動的に判定する機構を追加することを提案する．本稿では，これらの機構をそれぞれ**アドバイス機構**，**信号比較機構**と呼称する．アドバイス機構は，シミュレーションでエラーが発生した際に，GHDL によるオリジナルのエラーメッセージに加え，日本語によるエラー原因と対処法を付記するものである．信号比較機構は，シミュレーションの結果として得られる波形ファイルに注目し，手本となる波形ファイルの特定の信号波形が，生成された波形ファイルの中に現れるかを判定するものである．本稿では，これらの機構のプロトタイプの設計と実装，動作例について報告する．

[*1] エラーメッセージは，Xilinx 社の FPGA 開発環境である ISE では `+ can not have such operands in this context`，Vivado では `0 definitions of operator "+" match here` である．意味はいずれも同様である．

2. 関連研究

2.1 日本語を利用する取り組み

プログラミングにおいては，日本語母語話者である初学者が手軽にわかりやすいプログラミング学習をするための取り組みとして，日本語プログラミング言語がいくつか提案されている．兼宗ら [6] は，主に初中等教育における利用を想定したプログラミング言語としてドリトルを提案している．ドリトルはプロトタイプベースのオブジェクト指向言語であり，予約語の概念がない，階層的でない，記述に使用する記号類が少ないなど，既存のプログラミング言語の難解な要素を取り除いたという特徴がある．これにより，部品を再利用して短期間で高度な機能を作れるという，現代的なプログラミングの本質を学習者が理解しやすいものとなっている．また，教育目的ではなく，実務での適用性や習得コスト削減などを志向した日本語プログラミング言語に，Mind [7] がある．これらの提案は，プログラミングの本質の理解や生産性の向上のために新たな言語を設計・提案するものであり，本研究の目的である既存の言語（HDL）を習得させることに直接結びつくものではない．

また，ソフトウェア内部のメッセージを（日本語に限らず）国際化するためのライブラリに，gettext [8] がある．gettext は C 言語のライブラリであり，文字列定数を実行環境の言語に合ったものに置き換える関数 gettext を提供する．gettext 関数を使うようにソースコードを修正し，適当な翻訳をリソースファイルとして用意することで，実行環境の言語に合わせたメッセージが表示できるようになる．この仕組みにより，開発と翻訳の分業が可能である．しかしながら，この仕組みを利用するにはソースコードの修正が必要であり，大規模なソフトウェアではそれ自体に多くの時間を要する．また，VHDL の場合同一のエラーが異なる文脈で登場することがあり，対処法も文脈によって異なる場合がある．そのため，アドバイス機構の構築にあたっては，単にエラーメッセージを日本語訳するだけでなく，そのエラーが発生した文脈を分析し，適切な対処法やアドバイスを表示できることが望ましい．

2.2 プログラミング課題の自動採点

電子・情報系の大学や高専などにおけるプログラミング教育においては，Java [9] や C 言語 [10], [11] などを対象とした，様々な Web ベースの自動採点システムが開発されている．これらは教員の負担軽減や，学生への即時的なフィードバックに有用である．岩本ら [10] の提案するシステムでは，提出されたプログラムの正誤を判定するだけでなく，提出されたソースコードの中で類似するものを自動的に検出し，不正コピーの疑いとして教員に提示する機能も備えている．しかしながら，著者らの知る限り，HDL

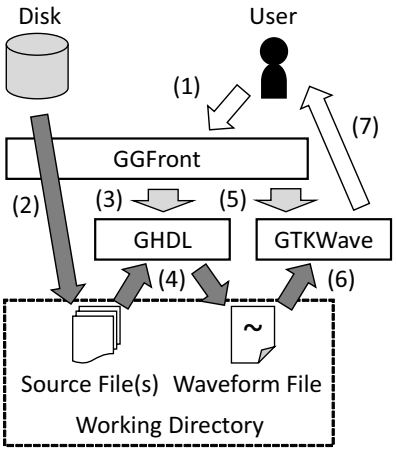

図 1　GGFront を用いた VHDL シミュレーションのフロー.
Fig. 1　VHDL simulation flow using GGFront.

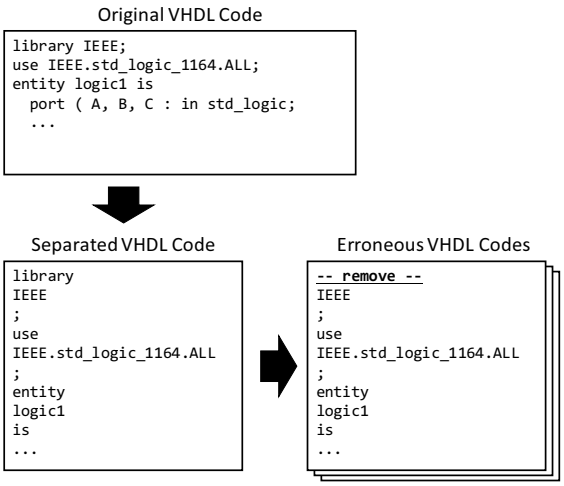

図 2　誤りのある VHDL コードの組の生成法.
Fig. 2　Generation of a set of erroneous VHDL codes.

を対象とした自動採点システムの開発事例は見られない.

提案する信号比較機構は，将来的にはこうした自動採点システムの主要な構成要素の 1 つとして利用できる可能性がある．VHDL を対象とした自動採点システムの構築は今後の課題とする.

3. 対象演習環境の概略

図 1 に，GGFront を用いた演習環境における VHDL シミュレーションのフローを示す．GGFront は VHDL シミュレータの GHDL [4] と波形ビューアの GTKWave [12] に対する GUI フロントエンドである．ユーザとのインタラクションを GTKWave とともに担当し（白い矢印），内部的にこれらの外部プログラムを実行する（薄いグレーの矢印）．その過程でソースファイルが作業ディレクトリにコピーされ，シミュレーション結果は波形ファイルとして作業ディレクトリに保存される（濃いグレーの矢印）．図 1 には明確に示していないが，もしソースファイルの解析時，またはシミュレーションの実行時に GHDL がエラーを返した場合は，GHDL のエラーメッセージをテキストファイルとして作業ディレクトリに保存し，そのファイルを開くことで，エラーを表示する.

本研究で提案するアドバイス機構は，GHDL がエラーを返した際にその内容を解析し，適切な説明・対処法を付加することで実現できる．アドバイス機構のプロトタイプについては 4 節で述べる．また，信号比較機構は GHDL が出力する波形ファイルに着目し，手本となる波形ファイルと生成された波形ファイルとの比較を行うことで実現できる．信号比較機構のプロトタイプについては 5 節で述べる.

4. アドバイス機構

4.1 エラーの洗い出し

アドバイス機構の構築にあたっては，VHDL の記述に誤りがあるときにどのようなエラーが出力されるのかを洗い出す必要がある．そのため，まず自動化されたエラーの洗い出しツールを提案する．本稿で対象とする記述の誤りの範囲は，記述中のいずれか 1 単語の書き間違いまたは脱落とする.

図 2 に，エラーの洗い出しツールで誤りのある VHDL コードを作成する手順を示す．まず, 元の（Original）VHDL コードを単語ごとに切り分け，全ての単語の間に改行を挿入することで，分割された（Separated）VHDL コードを作成する．これは VHDL の文法上，1 個以上の半角スペースと改行はすべてセパレータと扱われることを利用している．ここで単語とは，識別子やリテラル，あるいは「;」や「(」などのデリミタを指す．本来は「.」もデリミタの一種であるが，本プロトタイプではこれを切り分けることはしていない．そのため，図 2 に示すように，パッケージ宣言に頻出する `IEEE.std_logic_1164.ALL` はまとめて 1 単語として認識されている．次に，分割された VHDL コードの中から 1 行を選択し，コメント行，または別の単語で置き換える作業を行う．コメント行への置き換えにより単語の脱落，別の単語での置き換えにより単語の書き間違いを再現できる．図 2 には，1 行目をコメント行（`-- remove --`）に置き換えた場合を示す．これを全ての行に対して行うことで，誤りのある（erroneous）VHDL コードの組が生成される.

洗い出しツールでは，生成された VHDL コードのそれぞれに対して GHDL によるソースの解析を行い，得られたエラー・警告メッセージを記録する．全てのコードを解析したら，メッセージをその本文でソートしてから出力する．洗い出しツールは C#を用いて記述した.

図 3 に示す簡単な組合せ回路の VHDL 記述およびテストベンチをもとに，洗い出しツールを用いてエラーを列挙した．ただし，図中の NAND ゲートの出力には，内部信号として T という名前をつけている．分割された VHDL

表 1　簡単な組合せ回路とそのテストベンチから最もよく得られたエラー・警告メッセージ．
Table 1　Most frequent error and warning messages obtained from a simple combinatorial circuit and its test bench.

	Message	# of occurrence		
		Circuit	T.B.	Total
1	`__A__ is expected instead of __B__`	26	70	96
2	`library unit "__A__" was also defined in file "__B__"`	13	45	58
3	`missing __A__ at end of statement`	0	34	34
4	`primary expression expected`	2	29	31
5	`__A__ expected after __B__`	0	24	24

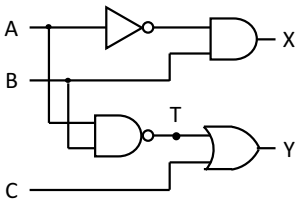

図 3　エラーの洗い出しに用いた組合せ回路．
Fig. 3　A simple combinatorial circuit used for enumeration of errors.

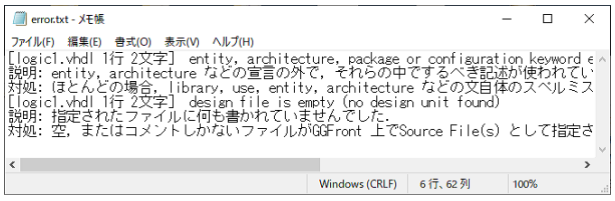

図 4　アドバイス機構を付加した GGFront のエラー出力の例．
Fig. 4　An example of error output of GGFront with an advice mechanism.

コードに対して行う処理は，コメント行への置き換えとした．

表 1 に，列挙されたエラー・警告を種類別に集計した結果，最も頻繁に得られたメッセージをまとめる．表中の「__」で囲まれた部分は，ファイルによって異なる語句が入る部分を示す．表右側の数字は，回路記述（Circuit）とテストベンチ（T.B.）のそれぞれで，メッセージが出現した回数を示す．Total の列はその和である．なお，洗い出しの結果得られたエラー・警告の種類は，49 種に及んだ．

単語を欠落させた場合，得られるエラーの多くは期待した単語が現れなかったことによる構文エラーであることが，表 1 から確認できる．第 2 位を除く 4 つのメッセージはいずれも構文エラーである．一方，第 2 位のメッセージは同名の回路が重複定義されているという内容の警告である．これは実際に重複定義があるわけではなく，単に作業ディレクトリに前回のコード解析結果が残ってしまったために発生しているものと考えられる．しかしながら，これもアドバイス機構で対処すべきメッセージの 1 つとした．

これらのエラー・警告のうち，頻出した 29 種について，各メッセージに対する説明と対処法をまとめた．表 1 のように部分的に異なる語句が現れる場合に対応するため，メッセージは正規表現とし，マッチした文字列は説明や対処法でも利用できるようにした．まとめた結果は，タブ区切りのテキストファイル（TSV）として保存し，アドバイス機構から利用する．

4.2　アドバイスの追加

4.1 節で作成したメッセージの対応表をもとに，アドバイス機構のプロトタイプを構築し，GGFront に組み入れる．アドバイス機構は ReplaceGHDLError という名前の C#のクラスとして実装する．エラーメッセージを作業ディレクトリに保存する直前にこのクラスのメソッドを呼び出すことで，エラーメッセージに対応する説明と対処法を付加する．

アドバイス追加処理の概略を示す．ReplaceGHDLError クラスは事前にメッセージの対応表（TSV）を読み込み，その内容をリストに保存している．アドバイス追加処理はエラーメッセージを入力とする．入力を 1 行読み出し，リスト内の各メッセージと正規表現によるマッチングを行う．もしマッチするメッセージが見つかれば，対応する説明・対処法を元のメッセージ（入力）とともに出力する．リスト内にマッチするメッセージがない場合は，元のメッセージだけを出力する．これを入力の終わりまで繰り返す．

アドバイス機構を付加した GGFront のエラー出力の例として，**図 4** に，簡単な組合せ回路の VHDL 記述において冒頭にある `library` を削除した場合の，GGFront のエラー出力を示す．ここでは，コード冒頭で 2 個のエラーが発生しており，それぞれに日本語による説明と対処法が付加されていることが確認できる．

第 1 のエラーは，VHDL 記述の冒頭に現れうる単語は `entity` などいくつかのキーワードに限られているにもかかわらず，それらが現れていないことによる構文エラーである．アドバイス機構はこれに対し，文自体のスペルミスや欠落がないかの確認を促しており，適切な対処法を提示しているといえる．

一方，第 2 のエラーは，指定されたファイルが空であったという内容のものである．実際に空のファイルを指定し

た場合にもこのエラーは出現する．しかしながら図 4 の例の場合は，記述冒頭の library が見つからないために，記述の全体構造を GHDL が認識することができなかったことが，このエラーの原因である．すなわち，このエラーは第 1 のエラーに付随して生じており，第 1 のエラーに正しく対処することこそが，この場合の正しい対処法である．アドバイス機構はこれに対し，空のファイルが指定されていないかの確認を促しており，不適切な対処法を提示しているといえる．

以上をまとめると，現状のアドバイス機構のプロトタイプでは，エラーに対して日本語による説明と対処法を付加するという基本的な機能は実現できたものの，エラーが発生した文脈を正しく分析するには至っていない．図 4 の例のように，エラーはしばしば組になって出現し，一方のエラーがもう一方のエラーに付随して生じることがある．そのため，あるエラーが発生したか否かによって，付加する説明や対処法を切り替えることにより，この問題を解決できる可能性がある．その解決は今後の課題とする．

5. 信号比較機構

5.1 プロトタイプの実装

本稿で述べる信号比較機構のプロトタイプは以下のように定義する．2 つの波形ファイル A, B と，A に含まれる信号名（複数可）とを入力とする．ファイル A の指定した信号の波形と一致する波形をファイル B から探し出す．もし一致するものがあるならば，そのような信号名の組を全て出力する．ただし，探索対象はテストベンチからインスタンス化された検証対象の回路に含まれる信号とし，テストベンチ内部で宣言されている信号については，探索から除外する．

信号比較機構が想定する利用ケースは，あらかじめ教員がテストベンチと正しい回路記述とを作成し，そのシミュレーションの結果得られる，「手本」となる波形ファイルを学生に提供するものである．学生が正しい回路記述の作成に成功すれば，得られる出力信号や主要な内部信号の波形は手本と一致するはずである．一致を自動的にチェックすることができれば，記述の正誤を速やかに学生にフィードバックできる．また，課題の出題にあたっては，学生間での安直な記述のコピーを防ぐために，信号名などに制約を課す（例えば学籍番号に応じて，入力信号 A を A123 のように変更するなど）ことも考えられる．この場合でも正しく一致を確認できるよう，信号名は比較には用いず，結果の出力のみに用いる．

信号比較機構を構築するには，まずは波形ファイル（VCD）から，使われている信号の名前と波形とを抜き取る必要がある．**図 5** に，図 3 で示した組合せ回路とそのテストベンチをシミュレートして得られる波形ファイルの一部と，それを GTKWave により可視化したものを示す．波

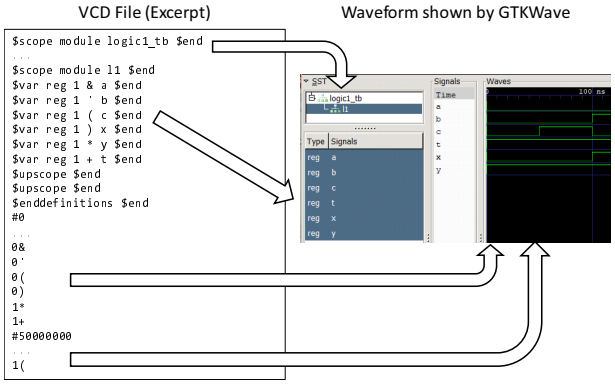

図 5 波形ファイル（VCD）のフォーマットの概略．
Fig. 5 Abstract of waveform format (VCD).

形ファイルはヘッダ（図 5 では省略），回路と信号の定義，波形の定義の 3 つの部分に分かれている．回路は$scope 文～$upscope 文で囲まれた部分で定義される．$scope 文を重ねがけすることで，回路の階層構造を表現する．各回路に含まれる信号は$var 文で定義され，信号の種類と幅，略記号と信号名を含む．回路と信号の定義の部分は，$enddefinition 文により終了する．波形の定義では，#で始まる行が時刻を表し，それ以外の行では，その時刻において遷移した信号を，信号値と略記号の組で示す．

例えば，テストベンチ（logic1_tb）で検証対象としている回路（l1）には入力 C があり，時刻 0 で信号値は ‘0’ であり，50 ns で ‘1’ となっていることが，GTKWave の画面から読み取れる．VCD ファイルでは信号定義の$var reg 1 (c $end の 1 文から，この信号の略記号として「(」が割り当てられていることがわかる．また，波形の定義では#0 のあとの 0(，#50000000 のあとの 1(から，時刻 0 でこの信号が ‘0’ となり，時刻 50,000,000 fs = 50 ns で ‘1’ となっている，すなわち GTKWave の画面との対応が取れていることが確認できる．

以上を踏まえ，信号比較機構のプロトタイプを Python で実装した．その主要な関数として，placeW と placeD の 2 つの関数を実装した．関数 placeW は，波形読み取りの前処理として，検証対象の回路が定義された部分を探し出し，その初めと終わりの行番号，およびその中に含まれる信号定義の数を返す．関数 placeD は，検証対象の回路が含む全ての信号について，信号名と波形の組を生成し，そのリストを返す．

関数 placeW では，ファイルを先頭から読み取り，まず$scope 文を探索する．求める初めの行番号は，$scope 文が 2 回目に出現した箇所の行番号である．$scope 文が 2 度出現したら，今度は$var 文と$enddefinition 文を探索する．求める終わりの行番号は，$enddefinition 文が出現した箇所の行番号である．また，信号定義の数はその間に出現した$var 文の数である．

関数 placeD では，先に求めた信号定義の数だけ以下の

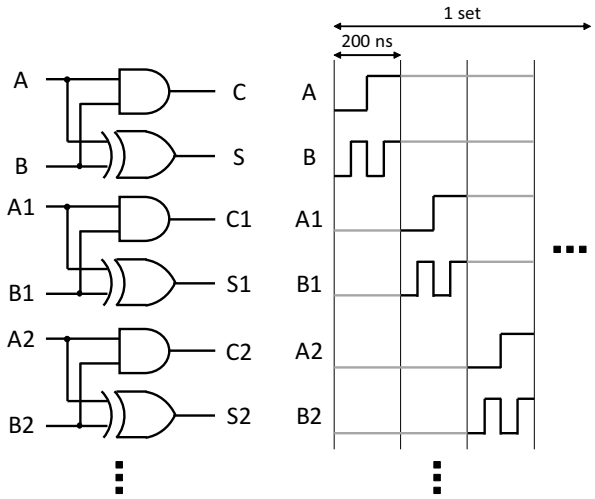

図 6　信号比較機構の評価に用いる回路とテストベンチ.
Fig. 6　Circuit and test bench for evaluation of signal comparison mechanism.

処理を繰り返す．ただし，ループ定数を i $(i = 1, 2, ...)$ とする．まず `$scope` 文と`$upscope` 文で回路名を認識しながら，i 個目の`$var` 文を探索する．得られた回路名と信号名・略記号を記憶する．次に，波形の定義部分を 1 行ずつ読み取る．それが`#`で始まる行であれば，時刻を波形のリストに追加する．またそれが信号値と記憶した略記号の組であれば，信号値を波形のリストに追加する．最後に，ファイルの終端に到達したら，得られた波形のリストと回路名・信号名の組を，出力のリストに追加する．信号の比較は，得られた波形のリスト同士を単に`==`演算子で比較するだけで行える．

5.2　プロトタイプの評価

信号比較機構のプロトタイプの性能評価のため，同一の波形ファイルを入力とし，回路中の全ての信号を対象とした比較を行い，その実行時間を測定する．この場合，回路中の各信号の波形が互いに異なっていれば，各信号は自分自身とだけ一致する．

評価のために，半加算器を複数個並列に並べた回路と，そのテストベンチを用意した．**図 6** に，評価に用いる回路とテストベンチの概略を示す．入力 A，B と出力 C，S をもつ半加算器を複製し，入力には A1, B1, A2, B2, ... と，出力には C1, S1, C2, S2, ... と名前をつける．これらの波形を互いに異なったものとするため，入力波形の最初の 200 ns では A, B を 4 通りの入力全てが試せるように変化させ，次の 200 ns では A1, B1 を同様に変化させる．これを最後の半加算器の入力まで行うことで得られる入力波形を 1 セットと定義し，これを指定されたセット数だけ繰り返す．

評価に先立って，半加算器の個数 H を 5, 10, 15, 20，入力波形のセット数 S を 1, 2, 3 に設定し，それぞれに対して

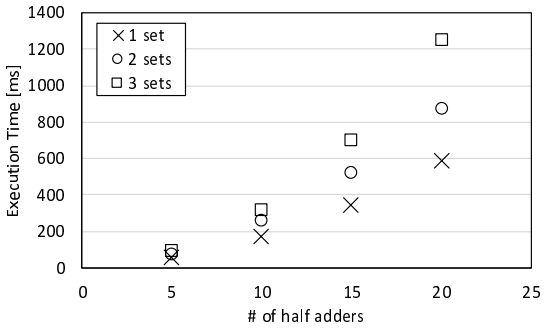

図 7　信号比較機構のプロトタイプの実行時間.
Fig. 7　Execution time of prototyped signal comparison mechanism.

GHDL でシミュレーションを行い，波形ファイルを得た．これらのファイルをもとに性能評価を行う．評価は Ryzen 5 2400G を搭載したデスクトップ PC で行う．Python は Windows 10 上の Python 3.7.3 を用いる．測定は 5 回行い，その実行時間の平均値を用いる．

図 7 に，信号比較機構のプロトタイプの実行時間を示す．横軸は半加算器の個数 H，縦軸は実行時間 [ms] であり，セット数 S に応じて異なるマーカーでプロットしている．グラフより，実行時間はおおむね半加算器の個数 H の 2 乗に比例して増加しているように見える．

しかしながら，本評価ではシミュレーション時間（波形の長さ）も半加算器の個数に比例している．具体的には，図 6 からも見て取れる通り，シミュレーション時間 T は $0.2HS$ [μs] となる．これを考慮して分析したところ，実行時間は概ね $H(T + 3)$ に比例していることが判明した．また，実行時間の中で支配的であったのは関数 placeD であった．

このことは次のように理解される．関数 placeD では信号の個数と同じ回数のループが実行されるため，placeD の実行時間は H に比例する．また，ループの各イテレーションでは，波形ファイルの波形の定義部分を 1 行ずつ読み取る．波形の定義部分の行数はシミュレーション時間 T に概ね比例するため，実行時間もまた T に概ね比例する．定数部 $+3$ は T に依存しない，回路名などの読み取り部分であると考えられる．

図 7 より，半加算器を 20 個並べた回路の 3 セット（12 μs）のシミュレーション波形を比較するのに要した時間は 1250 ms であった．より大きな回路，長時間のシミュレーション結果を比較することも想定されるため，即時性を求めるならば波形の取り込みに要する時間をより短くすることは必須である．プロトタイプの実装では，ファイルへのアクセスを最小化するために Python の標準モジュールである linecache を用いたが，それだけでは不十分であったと考えられる．改善にはアルゴリズムの見直しが必要であるが，今後の課題とする．

6. おわりに

本稿では，GGFront を用いた VHDL の演習環境を更に初学者にとって利用しやすいものとするため，アドバイス機構と信号比較機構という 2 つの機構を提案し，そのプロトタイプについて述べた．いずれのプロトタイプも，要求される基本的な機能は実現できたものの，文脈に応じたアドバイスの付加や波形ファイルの読み出しの高速化といった，改善すべき点が明らかとなった．

今後はこれらのプロトタイプの構築により得られた知見をもとに，提案機構を GGFront，あるいは VHDL の自動採点システムへと統合することをめざす．

参考文献

[1] Hennessy, J. L. and Patterson, D. A.: コンピュータアーキテクチャ定量的アプローチ 第 6 版，エス・アイ・ビー・アクセス (2019).
[2] 藤枝直輝: 手軽でポータブルな VHDL 演習環境のためのフロントエンドツール GGFront の開発，情報処理学会研究報告 2018-CE-147，pp. 15:1–15:6 (2018).
[3] 藤枝直輝：GGFront, (online), available from ⟨https://sites.google.com/site/nfproc/ggfront⟩ (accessed 2020-01-25).
[4] Gingold, T.: GHDL, (online), available from ⟨http://ghdl.free.fr/⟩ (accessed 2020-02-04).
[5] Asplund, L.: VUnit — VUnit Documentation, (online), available from ⟨https://vunit.github.io/⟩ (accessed 2020-02-04).
[6] 兼宗　進，御手洗理英，中谷多哉子，福井眞吾，久野　靖: 学校教育用オブジェクト指向言語「ドリトル」の設計と実装，情報処理学会論文誌プログラミング，Vol. 42, No. SIG 11 (PRO 12), pp. 78–90 (2001).
[7] スクリプツ・ラボ: 日本語プログラミング言語 Mind,（オンライン），入手先 ⟨https://www.scripts-lab.co.jp/mind/whatsmind.html⟩（参照 2020-02-04）.
[8] Free Software Foundation: gettext — GNU Project, (online), available from ⟨https://www.gnu.org/software/gettext/⟩ (accessed 2020-02-04).
[9] Kitaya, H. and Inoue, U.: An Online Automated Scoring System for Java Programming Assignments, *International Journal of Information and Education Technology*, Vol. 6, No. 4, pp. 275–279 (2016).
[10] 岩本　舞，中村真人，小島俊輔，中嶋卓雄: 不正コピー検出手法を備えたオンラインジャッジシステムの開発，情報処理学会論文誌教育とコンピュータ，Vol. 1, No. 4, pp. 38–47 (2015).
[11] 蜂巣吉成，吉田　敦，阿草清滋: プログラムの誤り修正課題および正誤判定プログラムの自動生成，情報処理学会論文誌教育とコンピュータ，Vol. 3, No. 1, pp. 64–78 (2017).
[12] Bybell, T.: GTKWave, (online), available from ⟨http://gtkwave.sourceforge.net/⟩ (accessed 2020-02-04).