Politechnika Szczecińska
Wydział Informatyki

Marcin Radziewicz

Opracowanie algorytmów przekładu zdań w języku VHDL opisujących logikę kombinacyjną na równania boolowskie

Rozprawa doktorska

Promotor:
prof. dr hab. inż. Włodzimierz Bielecki

Szczecin 2008

Streszczenie

Gwałtowny rozwój technologii półprzewodnikowych otworzył przed projektantami układów scalonych zupełnie nowe perspektywy. Jednak wraz z coraz większą gęstością układów rosła złożoność procesu projektowego i wytwórczego. Konieczne stało się opracowanie nowych, bardziej elastycznych metod projektowania i wytwarzania układów scalonych. Taką nową jakością stały się języki opisu sprzętu (ang. *Hardware Description Languages*), które umożliwiają tworzenie nowego elementu scalonego w sposób bardzo podobny do tego, w jaki piszę się program komputerowy. Ich najważniejszymi zaletami są: operowanie na różnym poziomie abstrakcji w ramach tego samego projektu, a także możliwość symulacji i weryfikacji układu przed jego fizyczną implementacją. Język VHDL powstał na zlecenie amerykańskiego Departamentu Obrony, jako narzędzie służące dokumentowaniu projektów układów scalonych. Wraz ze wzrostem jego popularności zaczęto wykorzystywać także inne jego możliwości - przede wszystkim bardzo rozbudowane mechanizmy wspierające symulacje. Kolejnym etapem było opracowanie narzędzi umożliwiających syntezę kodu języka VHDL. Ponieważ jednak język ten nie był tworzony z myślą o syntezie to posiada wiele instrukcji i mechanizmów których nie da się bezpośrednio przełożyć na strukturę krzemową. Konieczne stało się zatem stworzenie pewnego podzbioru instrukcji, który byłby syntezowalny. Ponieważ dziedzina tego typu narzędzi została opanowana przez firmy komercyjne, zaowocowało to dużą ilością produktów. Konkurujące firmy starały się zwiększyć maksymalnie podzbiór syntezowalnych instrukcji języka VHDL. Ciągle jednak pozostaje wiele ograniczeń.

Treścią niniejszej pracy są nowe algorytmy części tylnej kompilatora, dokonujące przekładu syntezowalnych źródeł VHDL zawierających logikę kombinacyjną. Zaprezentowane rozwiązanie dotyczy instrukcji sekwencyjnych. Jako format wyjściowy wykorzystano równania boolowskie. Praca podzielona została na pięć rozdziałów. W rozdziale pierwszym przedstawione zostały okoliczności oraz uzasadnienie wyboru tematu, celu pracy, a także sformułowana została teza badań.

Rozdział drugi przedstawia stan wiedzy w dziedzinie projektowania i wytwarzania układów scalonych, języków opisu sprzętu (ze szczególnym uwzględnieniem języka VHDL), i narzędzi na nich opartych. Opisana została dokładnie budowa kompilatora takiego języka. Przedstawiono zalety równań boolowskich jako formatu wyjściowego. Najważniejszą część rozdziału stanowi opis badań pokrewnych do tematu rozprawy.

Rozdział trzeci stanowi sedno pracy. Zostały opisane w nim zagadnienia kompilacji źródeł w języku VHDL zawierających logikę kombinacyjną. Poszczególne instrukcje przedstawiane są w miarę możliwości oddzielnie, aby ułatwić poznanie zagadnienia. Prezentacja każdej z instrukcji kończy się podaniem zwięzłego algorytmu kompilacji stanowiącego podsumowanie badań.

Rozdział czwarty to weryfikacja i ocena zaproponowanych algorytmów. Przedstawiona została procedura testowa, zbiór testów, oraz otrzymane wyniki. Zaprezentowane zostały także rezultaty otrzymane za pomocą narzędzi komercyjnych.

Rozdział piaty stanowi podsumowanie otrzymanych wyników oraz całej pracy. Przedstawia także możliwe kierunki dalszych badań.

Słowa kluczowa: synteza logiczna, VHDL, równania boolowskie, techniki kompilacji.

Dane autora:
Marcin Radziewicz
Politechnika Szczecińska
Wydział Informatyki
Katedra Technik Programowania
ul. Żołnierska 49, 71-210 Szczecin
email: mradziewicz@wi.ps.pl

Dla Słońca,
które świeci nawet w nocy.

Spis treści

Spis rysunków

Spis tabel

Spis przykładów

Wykaz symboli i skrótów

ASIC (ang. *Application Specific Integrated Circuit*) półprzewodnikowy układ scalony projektowany na zamówienie użytkownika i realizujący konkretne funkcje

BLIF (ang. *Berkeley Logic Interchange Format*) format tekstowy opisu układu cyfrowego na poziomie struktury logicznej, umożliwiający przenoszenie danych projektu pomiędzy różnymi narzędziami CAD, rozwijany przez Berkeley University of California

CAD (ang. *Computer Aided Design*) komputerowe wspomaganie prac projektowych

CDFG (ang. *Control-Data Flow Graph*) graf przepływu i sterowania

CPLD (ang. *Complex Programmable Logic Devices*) układy programowalne o strukturze hierarchicznej

EDA (ang. *Electronic Design Automation*) określenie narzędzi wspomagających projektowanie układów scalonych

EDIF (ang. *Electronic Design Interchange Format*) format tekstowy opisu układu cyfrowego na poziomie struktury logicznej, umożliwiający przenoszenie danych projektu układu cyfrowego pomiędzy różnymi narzędziami EDA

ETPN (ang. *Extended Timed Petri Net*) odmiana sieci Petri

FPGA (ang. *Field Programmable Gate Array*) układy programowalne o strukturze komórkowej

FPLD (ang. *Field Programmable Logic Devices*) układy programowane przez użytkownika

FSM (ang. *Finite State Machine*) automat skończony/maszyna stanów

HDL (ang. *Hardware Description Language*) język opisu sprzętu

IP (ang. *Intellectual Property*) własność intelektualna

PLD (ang. *Programmable Logic Devices*) układy programowalne

QDIF (ang. *QuickLogic Data Interchange Format*) format opisu układu cyfrowego umożliwiający przenoszenie danych projektu pomiędzy różnymi narzędziami

RTL (ang. *Register Transfer Level*) poziom przesłań międzyrejestrowych

SIL (ang. *SPRITE Input Language*) odmiana grafu CDFG używana w syntezie

SystemC Język opisu sprzętu, będący rozszerzeniem C++

Verilog HDL język opisu sprzętu (standard IEEE-1364)

VHDL język opisu sprzętu (standard IEEE-1076)

1. Wstęp

Wraz z rozwojem technologii półprzewodnikowych i możliwościami tworzenia coraz szybszych, większych i doskonalszych układów scalonych zmieniało się podejście do procesu ich projektowania i wytwarzania. O ile kiedyś podstawową metodą tworzenia była praca na poziomie pojedynczego tranzystora, to przy złożoności współczesnych układów scalonych trudno sobie wyobrazić, aby mogło to dalej funkcjonować. Konieczne stało się znalezienie nowego podejścia do sposobu projektowania i wytwarzania elementów scalonych, co z kolei wymagało rozwoju odpowiednich narzędzi. Najważniejszą zmianą było wypracowanie kilku metodologii lub inaczej ścieżek projektowych różniących się między sobą parametrami takimi jak:

— czas i koszt niezbędny do uzyskania pojedynczego działającego układu (np. prototypu),
— koszt produkcji masowej,
— maksymalna częstotliwość i złożoność wewnętrzna układu (ilość bramek).

Taka różnorodność dostępnych technologii jest bardzo korzystna. Projektant mając do dyspozycji szeroki wachlarz potencjalnych metod projektowych może wybrać tą z nich, która najlepiej pasuje do stawianych przed nowym układem wymagań. Pozwala to znacząco skrócić czas i obniżyć koszty jakie trzeba ponieść aby wprowadzić nowy produkt na rynek, co w dobie ogromnej konkurencji w niemalże każdej dziedzinie życia ma ogromne znaczenie.

Drugą ważną zmianą było porzucenie statycznej reprezentacji projektowanego układu, czyli niepodzielnie panującego przez dziesięciolecia schematu. Został on zastąpiony przez kod źródłowy napisany w języku opisu sprzętu *HDL* (*ang. Hardware Description Language*). Języki te wywodzą się na ogół z klasycznych języków programowania, ale zostały wzbogacone o konstrukcje i semantykę potrzebną do modelowania, symulacji, oraz syntezy układu scalonego. Wykorzystanie takich języków sprawia, że praca z projektem jest dużo łatwiejsza, a inżynier ma poczucie większej kontroli nad tym co robi. Języki HDL umożliwiają projektowanie na różnych poziomach abstrakcji. Można więc zacząć od bardzo ogólnej specyfikacji i w miarę postępu prac rozbudowywać ją dodając nowe elementy, lub uszczegóławiając te które już istnieją. Języki opisu sprzętu posiadają także wbudowane mechanizmy wspierające symulację. Dzięki temu model projektowanego układu nie jest statyczny, lecz dynamiczny. Umożliwia to weryfikowanie projektu na długo przed jego fizyczną implementacją. Zalety tej nie sposób przecenić i bez wątpienia jest to największy skok jakościowy w dziedzinie projektowanie i produkcji układów zintegrowanych. Kolejną istotną cechą języków sprzętu jest to, że w większości przypadków są to rozwiązania które doczekały powszechnie obowiązujących i uznawanych standardów (np. IEEE). Ułatwia to korzystanie z narzędzi różnych producentów i przenoszenie projektów między nimi, o ile tylko zachowany zostanie wystarczająco wysoki poziom abstrakcji - nie należy korzystać z niestandardowych rozszerzeń danego pakietu.

Współczesne oprogramowanie do syntezy logicznej(*EDA - ang. Electronic Design Automation*) cechuje uniwersalność i kompleksowość. Obecnie tendencja w tej

dziedzinie jest taka, aby jeden pakiet wspierał wszystkie etapy procesu produkcji nowego układu scalonego. Ponieważ języki HDL zdobyły dużą ale i pełni zasłużoną popularność, to obecnie większość dostępnego na rynku oprogramowania oferuje wsparcie dla przynajmniej jednego takiego języka. Nie jest to jedyna dostępna droga wprowadzania myśli twórczej do środowiska projektowego. Zwykle możliwe jest użycie diagramów stanów, tablic przejść automatu skończonego, tablic funkcji, a także formatu EDIF. Oczywiście poszczególne rozwiązania mogą oferować więcej lub mniej formatów wejściowych, te wymienione stanowią pewien standard.

Jak zostało wspomniane wcześniej, istnieje wiele technologii projektowania i wytwarzania układów scalonych. W pracy tej skoncentrowano się na tak zwanych układach reprogramowalnych przez użytkownika w polu - FPGA (ang. *Field Programmable Gate Array*). Cechą szczególną tego typu elementów jest to, że ich zachowanie nie jest definiowane podczas produkcji, a dopiero później przez inżyniera w procesie programowania. Układ FPGA opuszczający fabrykę półprzewodników nie ma zaszytego w sobie żadnego algorytmu. Można go porównać do niezapisanej kartki papieru. Omówienie tej technologii znajduje się w rozdziale 2.7. Należy tylko wspomnieć, że możliwości tego typu układów są w tej chwili tak duże, iż możliwe stało się wykorzystanie ich tam gdzie do niedawna można było zastosować tylko układy projektowane od podstaw (ang. *full custom*).

Postęp w dziedzinie oprogramowania EDA musi nadążać za rozwojem technologii wytwarzania układów scalonych. Jest to konieczne, aby inżynierowie elektronicy byli w stanie w pełni wykorzystać możliwości nowych półprzewodników. Powyższa przyczyna powoduje, że zainteresowanie nowymi narzędziami do syntezy logicznej jest duże. Inżynierowie elektronicy oczekują, ze kolejne wersje oprogramowania EDA, będą oferowały większe możliwości, a ich obsługa będzie łatwa i przejrzysta. Powinny one zapewniać wsparcie dla coraz to nowszych technologii wytwarzania lub w przypadku układów reprogramowalnych rodzin elementów scalonych. Muszą umożliwiać dokonywanie różnorodnej optymalizacji tak , aby można było uzyskać układ scalony o pożądanych parametrach. Jest to presja jakiej poddawani są tworzący narzędzia dla potrzeb syntezy sprzętu.

1.1. Uzasadnienie wyboru tematu pracy

Jako najważniejszy powód rozpoczęcia prac nad wyżej wymienionym tematem należy podać jego aktualność. Zapotrzebowanie na narzędzia do szybkiej i efektywnej syntezy języka VHDL nie maleje. Obserwując ekspansję elektroniki użytkowej na coraz to nowe dziedziny życia nie należy się spodziewać, że tendencja ta się zmieni. Na rynku ciągle jest miejsce dla nowych, lepszych pod kątem funkcjonalności i wydajności produktów. Widząc to, firma Aldec Corporation zajmująca istotną pozycję wśród producentów oprogramowania wspomagającego proces tworzenia układów scalonych, zwróciła się z propozycją współpracy do Katedry Technik Programowania Wydziału Informatyki Politechniki Szczecińskiej. Celem wspólnego działania miało być opracowanie kompilatora języka VHDL o przemysłowych parametrach jakościowych. Podstawowe założenia projektu były następujące:

— zdolność do syntezy źródeł napisanych w języku VHDL,
— zgodność w zakresie zbioru syntezowalnych instrukcji języka VHDL z pakietem FPGA Express II firmy Synopsis,
— zastosowanie równań boolowskich jako formatu wyjściowego,

— możliwość łatwego przenoszenia narzędzia na inne systemy operacyjne.

Pierwszym krokiem procesu projektowego było poszukiwanie już istniejących rozwiązań, aby ocenić stan badań w rozpatrywanej dziedzinie. Etap ten miał dwa główne cele:

— znalezienie wiedzy dotyczącej sprawdzonych rozwiązań konkretnych problemów z dziedziny tworzenia narzędzi syntezy,
— wyszukanie ograniczeń i słabych stron narzędzi już istniejących (akademickich i komercyjnych).

Poszukiwania polegały na badaniu wszelkich materiałów dotyczących języka VHDL, syntezy oraz tworzenia kompilatorów, zarówno HDL jak i tradycyjnych. Studiowane były także obecne na rynku narzędzia do syntezy logicznej oparte na języku VHDL, zwłaszcza takie, których profil był zbliżony do założeń projektu. Ze względu na kooperacje z firmą komercyjną ważne było, aby znaleziona wiedza była kompleksowa i nadawała się do praktycznego zastosowania. Mimo dużej ilości dostępnych publikacji, tylko niewielka liczba z nich spełniała chociaż w części postawione na początku poszukiwań wymogi. W większości przypadków dokumenty nie były wystarczająco szczegółowe, a prezentowana przez nie wiedza dotyczyła tylko pewnego wycinka dziedziny. Nawet duże opracowania, dotyczące działających kompilatorów nie były przedstawione wystarczająco szczegółowo. Zarzut ten dotyczył przede wszystkim prac akademickich. Dokumentacja produktów komercyjnych nie zawiera informacji na temat wewnętrznej budowy danego narzędzia. Użytkownik otrzymuje „czarną skrzynkę", a dołączony podręcznik opisuje tylko co otrzyma się na wyjściu skrzynki, gdy poda się określoną wartość wejściową - kod VHDL. Informacje zdobyte w ten sposób nie były wystarczające, aby zbudować od podstaw kompilator spełniający wymogi projektu. Znacznie lepiej wyglądała sytuacja w przypadku drugiego celu poszukiwań. Możliwości i ograniczenia oprogramowania były i są precyzyjnie opisywane w towarzyszącej jej dokumentacji. To zrozumiałe podejście, gdyż wiedza o granicach stosowalności danego narzędzia jest kluczem, do sprawnego się nim posługiwania. Znalezione rozwiązania akademickie były mocno ograniczone co do składni i zasobu możliwych do wykorzystania instrukcji języka VHDL. Jasno wytyczone limity stosowalności mają także narzędzia komercyjne. Ich możliwości są oczywiście dużo większe.

Brak gotowych do zastosowania rozwiązań postawił przed uczestnikami projektu trudne zadanie. Ze względu na komercyjne podłoże prac należało w jak najkrótszym czasie opracować stosowalne w praktyce algorytmy. Punkt wyjścia do tego procesu stanowiła ogólna wiedza z dziedziny technik kompilacji, tworzenia narzędzi syntezy, kompilacji języka VHDL[13]. Oprócz tego starano się skorzystać ze sprawdzonych już rozwiązań, jeżeli tylko była taka możliwość. Podsumowaniem tej bardzo wstępnej fazy projektu było ustalenie zakresu prac, podział projektowanego systemu na moduły, a także określenie tzw. „kamieni milowych" (*mile stone*). Ze względu na duży stopień komplikacji zdecydowano, że analiza instrukcji *process* języka VHDL będzie podzielona na dwie części:

— związaną z generacją logiki kombinacyjnej,
— oraz odpowiedzialną za tworzenie logiki sekwencyjnej.

Ostatnim etapem był przydział osób do poszczególnych zadań projektowych. Następnie członkowie zespołu przystąpili do prac, których wynikiem był zbiór zweryfikowanych praktycznie algorytmów obejmujących wszystkie zagadnienia procesu tworzenia kompilatora dla potrzeb syntezy logicznej. Mimo iż narzędzie stworzone

zostało z myślą o zastosowaniach komercyjnych to wyniki prac zostały opublikowane w [17], co pozwoliło zmniejszyć lukę w powszechnie dostępnej wiedzy na temat tworzenia kompilatorów języków opisu sprzętu.

1.2. Zakres, cel, oraz teza pracy

Przedstawione w pracy algorytmy lokują się w części tylnej (końcowej) kompilatora która nazywana jest także generatorem kodu. Ponieważ pojęcie generator kodu wywodzi się z domeny tradycyjnych kompilatorów, to należy mieć na uwadze, że nazwa ta może być nieco myląca, bowiem rzeczywiste zadania jakie spełnia ten moduł kompilatora w przypadku języków HDL są inne.

Niniejsza praca koncentruje się na problemach kompilacji instrukcji sekwencyjnych języka VHDL, czyli tych które występują tylko wewnątrz bloku *process*, a w zależności od swojej postaci mogą powodować konieczność zsyntetyzowania logiki kombinacyjnej lub też sekwencyjnej. Ponieważ oba zagadnienia są bardzo obszerne, a także w większości rozdzielne, to podczas tworzenia wspomnianego kompilatora zostały rozdzielone. Zakres badań syntezy logiki sekwencyjnej dotyczył przede wszystkim instrukcji *wait*[14][102]. Konieczne było opracowanie skomplikowanych algorytmów syntezy jednostki sterującej bazującej na automacie skończonym. W pracach założono, że instrukcja *wait* może wystąpić dowolną ilość razy wewnątrz ciała procesu.

Drugim zagadnieniem dotyczącym instrukcji sekwencyjnych jest synteza logiki kombinacyjnej. Jest to temat równie obszerny i ze względu na różnorodność języka VHDL także bardzo skomplikowany. W niniejsze rozprawie zaprezentowana została wiedza umożliwiająca syntezę logiki kombinacyjnej w postaci równań boolowskich. Prezentowane algorytmy generują również prostą[1] logikę sekwencyjną. Jest to związane z zasada funkcjonowania instrukcji *if* oraz *case* języka VHDL. W pewnych postaciach kompilacja kodu zawierającego wspomniane instrukcje powoduje konieczność utworzenie elementów synchronizujących - przerzutników. Uwzględniając powyższe, pełny zakres badań przedstawionych w niniejszej pracy obejmuje:

— zagadnienia kompilacji instrukcji sekwencyjnych języka VHDL z wyłączeniem *wait*, *while*, oraz *loop*,

— problemy związane z tworzenia logiki sekwencyjnej dla instrukcji *if* oraz *case*.

Szczegółowo zostanie to wyjaśnione w rozdziale 3.

Zagadnienia prezentowane w pracy dotyczą nauk technicznych, a dokładniej dyscypliny Informatyka (projektowanie logiczne, języki programowania, metody kompilacji, inżynieria oprogramowania). Problemem naukowym wokół którego skoncentrowana jest niniejsza praca to uzyskanie wiedzy niezbędnej do przeprowadzenia procesu automatycznej syntezy układów cyfrowych ze źródeł VHDL. Umożliwi to poszerzenie istniejącego zakresu wiedzy, a dzięki upublicznieniu będzie ona dostępna dla wszystkich.

Cel rozprawy sformułowany został następująco:

> **Celem niniejszej pracy jest opracowanie, weryfikacja i ustalenie zakresu stosowalności algorytmów kompilatora umożliwiających syntezę logiki kombinacyjnej w postaci równań boolowskich ze źródeł w języku VHDL.**

[1] To znaczy taką, która nie zawiera maszyny stanów

W pracy weryfikowana jest następująca teza:

> **Możliwe jest opracowanie algorytmów kompilatora, pozwalających na syntezę logiki kombinacyjnej do postaci równań boolowskich z instrukcji sekwencyjnych języka VHDL oraz na ich zastosowanie w kompilatorach akademickich i przemysłowych.**

Narzędzie przewidziane do zastosowania przemysłowego musi cechować się pewną użytecznością i jakością generowanych przez nie wyników. Użyteczność w tym konkretnym przypadku definiowana jest jako czas kompilacji i ilości wykorzystywanych do tego celu zasobów systemowych. Jeżeli wartości te są zbyt wysokie praktyczna przydatność takiego narzędzia jest niewielka. Drugim równie istotnym czynnikiem jest jakość wygenerowanego układu scalonego. Pojęcie to jest dosyć szerokie i może obejmować np.:
— całkowitą powierzchnie zajmowaną przez układ, wyrażoną np. przez ilość bramek,
— maksymalną częstotliwość pracy,
— pobór prądu.
W prezentowanej pracy skupiono się na pierwszym czynniku czyli zajmowanej powierzchni. Optymalizacja więcej niż jednego parametru charakteryzującego element scalony jest trudna i wykracza poza zakres tematyczny niniejszego opracowania.

Metodyka badań użyta w pracy do osiągnięcia wyników teoretycznych polegała na analizie istniejących narzędzi do syntezy logicznej pod kątem ich mocnych i słabych stron. Na tej podstawie opracowane zostały wymogi odnośnie zbioru syntezowalnych instrukcji sekwencyjnych. Podczas opracowywania teorii korzystano z następującej wiedzy: teorii układów logicznych, budowy i syntezy układów cyfrowych, zasad specyfikacji projektu w języku VHDL oraz metod kompilacji. Rozwiązanie teoretyczne zostało zweryfikowane eksperymentalnie, poprzez testowanie implementacji proponowanych algorytmów na zbiorze testowym (rozdział 4). Wyniki badań zostały opublikowane w [15][80][81][82][83][84][85].

1.3. Organizacja pracy

Praca podzielona jest na pięć rozdziałów o następującej treści:

Rozdział 1 - Wstęp. Przedstawia uzasadnienie wyboru tematu, cel i tezę pracy oraz układ rozprawy.

Rozdział 2 - Wprowadzenie do problemu. Opisuje zagadnienia związane z projektowaniem układów scalonych, przedstawia dostępne technologie realizacji, omawia szeroko temat języków opisu sprzętu, w szczególności język VHDL oraz w krótkiej formie prezentowana jest budowa kompilatora języka opisu sprzętu. Całość uzupełnia omówienie dostępnych na rynku pakietów akademickich i komercyjnych, a także ocena stanu badań w dziedzinie.

Rozdział 3 - Algorytmy generacji równań boolowskich. To najważniejsza część rozprawy, w której opisane są algorytmy generacji równań boolowskich dla poszczególnych instrukcji języka VHDL. Rozdział zaczyna się od przedstawienia

wiedzy niezbędnej do dokonania kompilacji instrukcji *process*, a następnie omawiane są kolejno instrukcje sekwencyjne. Teoria ilustrowana jest przykładami.

Rozdział 4 - Weryfikacja przedstawionych algorytmów. W rozdziale tym prezentowane są wyniki testów, jakim poddane zostały algorytmy opisane w rozdziale 3. Testy obejmowały weryfikacje poprawności jak i ocenę praktycznej przydatności rozwiązania. Przedstawiona zostaje także złożoność obliczeniowa prezentowanych algorytmów.

Rozdział 5 - Podsumowanie. Dyskusja otrzymanych wyników, ocena rozprawy i przedstawienie kierunku dalszych badań.

1.4. Podsumowanie

W rozdziale powyższym przedstawiono cel oraz tezę rozprawy, a także uzasadniono dlaczego wyżej wymieniony temat badań został podjęty. Przedstawiona została ogólna sytuacja w dziedzinie wytwarzania układów scalonych oraz omówiony został projekt kompilatora języka VHDL w ramach którego powstała rozprawa.

2. Wprowadzenie do problemu

Rozdział ten ma za zadanie w zwięzły sposób wprowadzić w przedmiot i zakres rozprawy. Omawia technologie wytwarzania układów scalonych, ze szczególnym uwzględnieniem układów programowanych przez użytkownika - FPGA. W dalszej kolejności prezentowany jest proces syntezy i jego zadania na różnych poziomach reprezentacji projektu. Następnie przedstawione zostają języki opisu sprzętu, ich historia oraz najbardziej popularni reprezentanci. Na końcu omówiona została sytuacja w dziedzinie narzędzi do syntezy. Opis dotyczy zarówno prac naukowych jak i rozwiązań czysto komercyjnych.

2.1. Projektowanie układów cyfrowych

Proces projektowania układu scalonego[20][24][62] ulegał ogromnym przemianom wraz z rozwojem elektroniki. Obecnie jest on wieloetapowy i w znacznym stopniu zautomatyzowany. Najpierw inżynier wprowadza specyfikacje układu korzystając z wielu dostępnych sposobów. Następnie określa wymagania, takie jak: powierzchnia, maksymalna częstotliwość pracy czy pobór energii. Reszta wykonywana jest automatycznie. Oczywiście istnieje możliwość bezpośredniej, ręcznej ingerencji w projekt na każdym z etapów. Kluczowe znaczenie ma wybór technologii wytwarzania w jakiej ma być zrealizowany układ. Powinna być ona dopasowana do wymagań stawianych przed danym układem, gdyż poszczególne technologię różnią się między sobą w zakresie parametrów uzyskiwanych przez gotowy układ.

Rozwojem układów cyfrowych[61][90][103] kierują dwa czynniki: zapotrzebowanie rynku, oraz dostępność możliwych technologii realizacji praktycznej. Można powiedzieć, że jest to swojego rodzaju zamknięte koło. Zakładając, ze projektant chce stworzyć nowy układ zintegrowany (nowy szybszy procesor, interfejs komunikacyjny), może się okazać, że nie ma w danej chwili technologii pozwalającej na realizacje takiego projektu. Trzeba ją dopiero wymyślić. Nie ma znaczenia czy problem sprowadza się do niewystarczającej gęstości tranzystorów/bramek, czy też problemem jest inne kryterium takie jak szybkość lub zużycie energii. Powstaje impuls do rozwoju. Może też wystąpić scenariusz odwrotny, kiedy to techniki wytwarzania układów ewoluują szybciej niż zapotrzebowanie twórców. Taka sytuacja z kolei może skłonić projektantów do zainteresowania się problemami dotychczas niemożliwymi do realizacji sprzętowej, ze względu na ograniczenia technologii.

Obecnie, większość tworzonych układów stanowią układy specjalizowane - ASIC [49] (ang. *Application-Specific Integrated Circuit*). Są to elementy realizujące w całości jakieś określone zadanie lub algorytm. Wcześniejsze technologie półprzewodnikowe wymagały zastosowania wielu prostych elementów, które dziś są zastępowane przez jeden układ typu ASIC. Aby sprawnie posługiwać się tą nową technologią wytwarzania potrzebne są odpowiednie narzędzia. Ich rozwój musi dotrzymywać kroku rozwojowi procesów wytwórczych półprzewodników. Wykonanie nowego układu sca-

lonego wymaga ogromnych nakładów czasowych i finansowych. Bez odpowiedniego oprogramowania wspomagającego projektowanie, koszty te byłyby jeszcze większe, co sprawia, że postęp w tej dziedzinie jest bardzo istotny.

2.2. Metody projektowania

Dobór właściwej metody (technologii) wytwarzania jest kwestią absolutnie podstawową. Ten etap wpływa w decydujący sposób na przyszłe osiągi projektowanego układu, a także na koszt i czas jego wytworzenia. Obecnie coraz częściej projektuje się układy scalone wykorzystując prefabrykowane, gotowe elementy. Przypomina to budowanie konstrukcji z klocków. Istnieje kilka technologii różniących się między sobą. W zależności od rodzaju projektu oraz wymagań jakie ma on spełniać inżynier ma do wyboru trzy podstawowe drogi realizacji układu:
— układy w całości projektowane od podstaw (ang. *full-custom*),
— układy projektowane z wykorzystaniem gotowych komórek (ang. *semi-custom*),
— układy wykorzystujące macierze bramek - FPGA i MPGA.

2.2.1. Układy projektowane od podstaw

Podejście to pozwala uzyskać układy o najwyższych możliwych parametrach czasowych i o najwyższej gęstości upakowania. Konieczne jest jednak tworzenie projektu na najniższym możliwym poziomie abstrakcji - geometrycznym. To natomiast wymaga zaangażowania dużych środków finansowych, a czas realizacji jest długi. Problemy te dotyczą przede wszystkim krótkich serii, bo przy masowej produkcji koszty ulegają obniżeniu, a czas tworzenia prototypu nie ma takiego znaczenia. Obecnie tą metodą wytwarza się układy scalone tylko wtedy, gdy jest to naprawdę niezbędne (specyficzne zastosowania, procesory, fragmenty dużych projektów). Ze względu na rozwój innych technologii, układy tego typu projektuje się wtedy kiedy rzeczywiście potrzebne są osiągi najwyższej klasy, albo ilość sprzedanych egzemplarzy gotowego produktu zrekompensuje ogromne nakłady poniesione na wykonanie działającego prototypu. Czasem układy tego typu są nazywane po prostu ASIC.

2.2.2. Projektowanie z wykorzystaniem gotowych komórek

Metodologię tą można podzielić na:
— projektowanie z użyciem komórek standardowych,
— projektowanie z wykorzystaniem generatora komórek.

Projektowanie w oparciu o komórki standardowe. Użytkownik korzysta z predefiniowanej biblioteki komórek realizujących określone funkcję. Każda taka komórka jest opisana szeregiem parametrów takich jak szybkość, opóźnienia, zużycie energii, czy powierzchnia. Projektant musi dostosować swój projekt do posiadanej biblioteki. Drugi problem to konieczność aktualizacji komórek w przypadku zmiany technologii półprzewodnikowej.

Projektowanie z użyciem generatora makrokomórek (modułów). Sytuacja ta, tym różni się od poprzedniej, że używany jest specjalny program (generator), który tworzy fizyczną reprezentację projektu według zadanego opisu. Istnieje wiele rodzajów generatorów, w zależności od tego, jakiego rodzaju moduły mają tworzyć.

Metodologia projektowania układów w oparciu o gotowe komórki ma tę zaletę, iż jest zgodna z metodologią *full-custom*. Umożliwia to łączenie w ramach jednego układu obu sposobów projektowania.

2.2.3. Projektowanie z wykorzystaniem macierzy bramek

Układ składa się z macierzy niepołączonych ze sobą elementów. Zadaniem projektanta jest naniesienie sieci połączeń. Istnieją dwie podstawowe odmiany tego typu produktów różniące się sposobem wykonywania połączeń:
— układy programowane maską - MPGA (ang. *Mask Programmable Gate Array*),
— układy programowane przez użytkownika - FPGA (ang. *Field Programmable Gate Array*).

Układy programowane maską. Programowanie polega na wytworzeniu warstw łączeniowych w układzie półprzewodnikowym stanowiącym półprodukt. Oczywiście proces taki może odbywać się tylko w fazie produkcji. Dzięki jednak temu, że proces łączenia jest stosunkowo prosty, koszty oraz czas przygotowania takiego układu nie są wysokie.

Układy programowane przez użytkownika. Programowanie (łączenie) odbywa się już po za fabryką. W zależności od typu, układy te mogą być programowane raz (z anty-bezpiecznikami), lub też wielokrotnie (układy z pamięcią). Układy FPGA stanowią sedno niniejszej pracy i z tego powodu poświęcony im został osobny rozdział - 2.7.

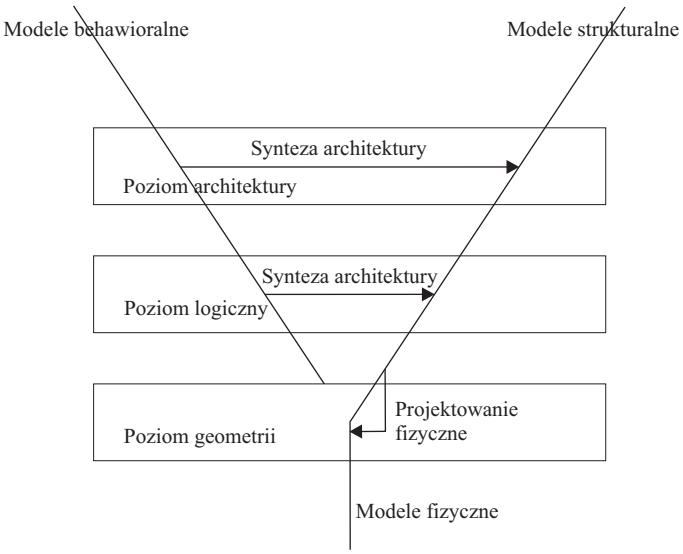
Zadaniem projektanta jest dobranie odpowiedniej metodologii projektowania do warunków projektu. Często prototypy wykonuje się przy pomocy technologii reprogramowanych, a następnie dopiero tworzy docelowy układ specjalizowany. Takie podejście pozwala zaoszczędzić czas i pieniądze. Układy FPGA cały czas są udoskonalane, dzięki czemu są stosowane tam, gdzie jeszcze nie dawno konieczne było użycie elementów full-custom. Rozwija się także projektowanie z wykorzystaniem mieszania technologii, polegające na produkowaniu hybryd ASIC-FPGA (np. procesor umieszczony wewnątrz macierzy bramek).

2.3. Modele i poziomy reprezentacji układu cyfrowego

Ze względu na stopień komplikacji współczesnych układów zintegrowanych proces ich projektowania jest złożony. Najpierw tworzony jest model, który odzwierciedla wszelkie cechy projektowanego przedmiotu, a następnie na jego podstawie realizuje się prototyp. Modeli może być wiele rodzajów, każdy uwypuklający nieco inne aspekty projektu. Dodatkowo mogą być one przedstawiane na różnych poziomach szczegółowości.

Projektując układ scalony (zwłaszcza przy użyciu HDL) można skorzystać z dwóch rodzajów modeli: Behawioralnego, w którym układ przedstawiony jest jako lista operacji, ich argumentów, oraz zależności między nimi.

Strukturalnego, gdzie układ podzielony jest na funkcjonalne wzajemnie połączone bloki takie jak np.: pamięć, jednostka arytmetyczno-logiczna, procesor. Podsumowując różnice między dwiema drogami projektowymi: specyfikacja behawioralna opisuje co dany układ ma robić, oraz w jakiej kolejności; opis strukturalny natomiast



Rysunek 2.1. Wykres Y Gajskiego i Kuhna - rodzaje modeli i poziomy reprezentacji układu scalonego. Źródło: [24].

dostarcza odpowiedzi na pytanie jak ma to być zrealizowane (przy pomocy jakich zasobów).

Podejście strukturalne jest z pewnością bliższe inżynierom sprzętowym, natomiast behawioralne programistom. Zgodnie z panującą tendencją podział na osoby zajmujące się stroną sprzętową, oraz odpowiedzialne za oprogramowanie powoli zaciera się. Najpierw realizuje się algorytm, a potem decyduje, która jego część będzie wykonywana przez oprogramowanie a co będzie zaimplementowane na poziomie sprzętu[100]. Prócz podziału na różne modele, układ scalony można projektować na różnych poziomach szczegółowości. W zależności od źródła można spotkać się z różną ich liczbą, ale najważniejsze są trzy:
— poziom architektury,
— poziom logiczny,
— poziom geometrii.
Poziomy te zostały przedstawione od najbardziej do najmniej abstrakcyjnego.

W zależności od wybranego sposobu modelowania, układ zdefiniowany na poziomie architektury może być listą operacji wraz z zależnościami między nimi (behawioralny), lub też połączeniem makroskopowych elementów stanowiących podsystemy układu (strukturalny).

Na poziomie logicznym w przypadku podejścia behawioralnego modelem będzie automat skończony, natomiast w przypadku strukturalnego sieć bramek i przerzutników.

Ostatnim poziomem reprezentacji układu jest poziom geometrii. Jest on niezależny od wybranej ścieżki projektowej. W tym miejscu spotykają się wszystkie metodologię. Stanowi on fizyczną mapę rozmieszczenia elementów w układzie, jego topografię. Historycznie był to pierwszy sposób tworzenia układów scalonych. Ponieważ w zależności od technologii istnieje możliwość mapowania projektu z dokładnością do pojedynczego tranzystora, podejście to oferuje ogromne możliwości optymalizacyjne, jednakże jest ono bardzo kosztowne i czasochłonne. Obecnie, w ten sposób projektuje się tylko fragmenty większych projektów.

2.4. Synteza

Synteza[3][24][59][60] jest procesem transformacji jednego modelu w drugi. Przyjmuje się, że dotyczy on zamiany modelu behawioralnego na strukturalny, bowiem reprezentacji behawioralnej nie można odwzorować bezpośrednio na poziomie fizycznym. Konieczna zatem jest zamiana jej na postać strukturalną i jest to właśnie zadanie procesu syntezy. Rysunek 2.1 przedstawia zależność między rodzajami modeli na różnych poziomach szczegółowości. W zależności od poziomu reprezentacji wejściowej, przejście do poziomu fizycznego może wymagać syntezy wieloetapowej.

Tak jak istnieją trzy poziomy złożoności, tak też mówi się o trzech poziomach syntezy:
— synteza poziomu architektury,
— synteza poziomu logicznego,
— synteza poziomu geometrycznego.
Wynik syntezy jednego poziomu, może być punktem wyjścia do syntezy poziomu niższego. Specyfikacja projektu na poziomie architektury zawiera przede wszystkim informacje o tym, co ma być zrealizowane, nie musi precyzować jak. Zadaniem procesu syntezy jest zatem wyodrębnienie zasobów niezbędnych do realizacji algorytmu implementowanego w tworzonym układzie. Przez zasoby rozumieć należy podsystemy takie jak elementy arytmetyczno-logiczne, pamięci i inne bloki funkcjonalne wysokiego poziomu. Z procesem tym wiążą się dwa pojęcia: kolejkowanie oraz alokacja[24][59]. Wynikiem syntezy architektury jest model strukturalny ścieżki danych oraz opis logiki sterującej układu.

Wynikiem syntezy poziomu logicznego jest strukturalny opis układu. Dane wejściowe może stanowić specyfikacja w postaci tablicy przejść automatu skończonego, schemat lub kod HDL. Opis ten może wprowadzić bezpośrednio projektant, ale coraz częściej jest to rezultat syntezy na poziomie wyższym, czyli architektury. Reprezentacja logiczna to sieć prostych elementów takich jak bramki czy przerzutniki. Ostatnim etapem syntezy na poziomie logicznym jest mapowanie (odwzorowanie) technologii. Decydująca kwestią jest tutaj wybrana wcześniej metoda projektowania (rozdział 2.2). Zupełnie bowiem inaczej, będzie wyglądać postępowanie w przypadku układów *full-custom*, *sem-custom*, czy też FPGA. W tym ostatnim przypadku mapowanie trzeba przeprowadzać pod kątem konkretnej rodziny układów.

Zadaniem syntezy poziomu geometrycznego jest fizyczna implementacja układu scalonego. Na podstawie wyników mapowania technologii rozmieszcza się poszczególne składowe bloki wewnątrz struktury krzemowej i przeprowadza łączenie. Przebieg tego procesu ściśle zależy od wybranej technologii projektowania. W przypadku układów projektowanych od podstaw, będą to maski niezbędne do wytworzenia wszystkich warstw układu. Gdy natomiast projekt będzie realizowany jako FPGA, wynikiem będzie lista antybezpieczników do przepalenia lub plik konfiguracyjny.

2.5. Języki opisu sprzętu

Języki opisu sprzętu (HDL, ang. *Hardware Description Language*) powstały jako odpowiedź na rosnącą złożoność projektowanych układów cyfrowych. Pod koniec lat siedemdziesiątych rozpoczęło się poszukiwanie nowych metod wytwarzania elementów zintegrowanych. Dostrzeżono wady podejścia polegającego na rysowaniu schematów, czy topologii fizycznej układu, ponieważ ze względu na rosnącą wielkość

tworzonych układów scalonych przestało się ono sprawdzać. Pierwszym krokiem rozwoju narzędzi EDA były programy umożliwiające rozmieszczanie i łączenie (ang. *place and route*). W dużej części nie były to narzędzia publicznie dostępne, a wewnętrzne produkty firm takich jak HP czy Intel. Rezultaty ich zastosowania były na tyle obiecujące, że stało się jasne iż nie ma powrotu do starych metodologii. Należało dalej zwiększać automatyzację prac projektowanych. Potrzebna była nowa koncepcja modelowania i dokumentowania projektów, narzędzie pozwalające projektantom lepiej panować nad coraz bardziej złożonymi układami. Odpowiedzią na ten problem okazały się być języki opis sprzętu. Umożliwiały one specyfikowanie projektów na różnych poziomach abstrakcji i przy pomocy różnych modeli. Języki HDL wyposażono w mechanizmy symulacyjne, co pozwoliło weryfikować projekty przed ich fizyczną realizacją. Program napisany w języku opisu sprzętu stanowi jednocześnie doskonałą dokumentacje. Mino wielu zalet, początkowo języki opisu sprzętu egzystowały głównie w środowisku akademickim. Punktem zwrotnym było pojawienie się języków HDL o ustalonym i opublikowanym standardzie. Dało to impuls do rozwoju samych języków, narzędzi na nich opartych, oraz popularyzacji tej metody projektowania. Powstało wiele języków opisu sprzętu, między innymi: VHDL[40], Verilog[43], SystemC[46], ABEL[106], ESIM[67] oraz HardwareC[53]. Występują między nimi oczywiście różnice wynikające z nieco odmiennego przeznaczenia, ale mają wiele cech wspólnych. Obecnie można zaobserwować unifikację w tej dziedzinie. Języki mające różne korzenie i pierwotne przeznaczenie zaczynają się coraz bardziej do siebie upodabniać. Następuje wzajemna wymiana zalet poszczególnych rozwiązań.

Modelowanie za pomocą narzędzi opartych na językach HDL[93] stało się dominującą metodologią projektowania układów scalonych. Tworzenie projektu przypomina pod wieloma względami klasyczne programowanie. Jest to zupełnie zrozumiałe, bowiem języki opisu sprzętu wywodzą się w znacznej mierze od swoich „miękkich" kuzynów (np. VHDL pochodzi od języka ADA). Nie należy jednak zapominać o istotnych różnicach. Przede wszystkim języki opisu sprzętu umożliwiają współbieżne wykonywanie instrukcji. Mało tego, ten sposób działania jest zakładany jako domyślny. Wynikiem działania tradycyjnego kompilatora jest wykonywalny program, natomiast w przypadku narzędzia do syntezy jest to opis fizycznej reprezentacji układu, ze wszystkimi wynikającymi z tego ograniczeniami. Inżynier sprzętowy musi bardzo dobrze wiedzieć jaki wynik da zastosowanie danej konstrukcji języka. Konwencjonalny programista ma o wiele większą wolność. Poza nielicznymi przypadkami, wystarczy, że skupi się tylko na implementacji algorytmu. Nie ma wielkiego znaczenia jakich dokładnie środków użyje. Kolejna różnica to możliwość symulacji kodu HDL. Nie można tego porównać do śledzenie wykonania programu. Jest to proces znacznie bardziej złożony. Kompilator języka HDL jest tylko jednym ze składników oprogramowania do syntezy. Całe środowisko projektowe zawiera wiele różnych współpracujących ze sobą narzędzi. Badając poszczególne pakiety można dojść do wniosku, że języki opisu sprzętu ciągle wyprzedzają swoje czasy. Nie ma bowiem środowiska, które potrafiłoby przełożyć pełen zbiór ich instrukcji na model fizyczny. Zawsze występują jakieś ograniczenia i wykluczenia.

W kolejnych rozdziałach przedstawione zostaną trzy najbardziej obecnie popularne języki służące do modelowania sprzętu: Verilog, SystemC oraz VHDL.

2.5.1. Verilog

Język ten został stworzony przez firmę Gateway Design Automation, jako narzędzie do modelowania układów około roku 1984. Twórcy języka chcieli stworzyć coś na wzór języka C, ale dla potrzeb projektowania układów scalonych. Drugim źródłem inspiracji był język opisu sprzętu Hilo. Prawie równocześnie z samym językiem powstał udany symulator. W roku 1990 GDA zostało przejęte przez Cadence Desing Systems. Firma ta zdecydowała się na uwolnienie języka, do czego doszło rok później. Aby zapewnić rozwój i utrzymanie standardu powołano organizacje o nazwie Open Verilog International (OVI), która doprowadziła do uzyskania w 1995 roku standardu IEEE o numerze 1364-1995[42] dla Verilog. Ponieważ zainteresowanie językiem nie malało, ukazały się dwa kolejne standardy, zawierające poprawki i rozszerzenia: IEEE 1364-2001[43] oraz IEEE 1364-2005[45].

Verilog[74] jest językiem o słabej typizacji i składni nawiązującej do języka C. Właściwie występują tylko dwa rodzaje danych:
— *reg* - *register* (rejestr) miejsce do przechowywania danych,
— *net* lub *wire* - połączenie między dwoma modułami.
Oba typy danych bazują na konstrukcji bitowej. Verilog umożliwia projektowanie w sposób behawioralny jak i strukturalny. Język umożliwia programowanie współbieżne jak i sekwencyjne. Można tworzyć projekty hierarchiczne. Język posiada także rozbudowane mechanizmy symulacyjne, a wbudowane prymitywy poziomu logicznego czynią z niego dobre narzędzie dla potrzeb syntezy[98].

Najnowszym dzieckiem w rodzinie Verilog jest SystemVerilog[92]. Język ten jest o wiele bardziej rozbudowany i uniwersalny. Wniósł nowe typy danych, w tym struktury, a nawet klasy, czyli metodykę obiektową. Istnieje także możliwość definiowania typów przez użytkownika. Oprócz tego, dla nowych typy danych wprowadzono silną typizację. Z innych mechanizmów SystemVerilog należy wymienić rozszerzone możliwości w zakresie weryfikacji. Język ten uzyskał także miano standardu IEEE 1800-2005[44].

2.5.2. SystemC

Nie jest to właściwie odrębny język, a jedynie rozszerzenie C++, składające się ze zbioru bibliotek dodającego nowe możliwości do tego popularnego języka. Całość jest tak skonstruowana, że może współpracować praktycznie z dowolnym kompilatorem C++. Pakiet służy do modelowania systemów jako całości, a nie tylko samego sprzętu, ale oczywiście wszystko zależy od użytkownika. Jest to właściwie pierwszy pakiet umożliwiający syntezę sprzętowo-programową[12]. Rozwojem SystemC zajmuje się Open SystemC Initiative (OSCI)[72]. W obecnej chwili jest już dostępny standard środowiska zaakceptowany przez IEEE o numerze 1666[46]. Najważniejsze cechy SystemC[18][73]:
— pełna zgodność syntaktyczna z C++,
— możliwość korzystania z natywnych typów C++, jak i z typów stworzonych na potrzebny modelowania sprzętu,
— możliwość modelowania na różnych poziomach abstrakcji: systemowym, architektury, oraz logicznym (RTL), oraz z wykorzystaniem obu metodyk (behawioralnej i strukturalnej),
— modułowość w tym także hierarchiczna w odniesieniu do projektu,
— wyodrębnienie procesów, jako definicji zachowania modułu,

— jawna specyfikacja portów wejściowych i wyjściowych z modułów, sygnały,
— jawna deklaracja zegara,
— mechanizm sterowania zdarzeniami,
— bogate możliwości symulacji.

Wiele mechanizmów jest bardzo podobnych do tych znanych z języka VHDL. SystemC jest bez wątpienia bardzo interesująca propozycją dla projektanta.

2.5.3. VHDL

Język ten jest przedstawiany najszerzej, ze względu na to, iż jest on powiązany z tematem niniejszej pracy. Skrót VHDL oznacza VHSIC HDL, czyli *Very High Speed Integrated Circuit Hardware Description Language*[104]. Pod tym wyjątkowo skomplikowanym akronimem kryją się prace prowadzone przez amerykańskie Ministerstwo Obrony (ang. *Department of Defense - DoD*). VHSIC to projekt, którego celem było opracowanie nowych wydajnych metod projektowania układów wielkiej skali integracji - VLSI. Pierwotnie VHDL był wykorzystywane jedynie do dokumentowania projektów, lecz z czasem jego rola w procesie tworzenia układów scalonych wzrosła. Najpierw jako narzędzia do symulacji (posiada bardzo rozbudowane możliwości w tym zakresie), w końcu także do syntezy logicznej. W 1987 VHDL uzyskał status standardu IEEE 1076-1987[39]. Specyfikacja ta została uaktualniona następnie w roku 1993[40]. VHDL wywodzi się w prostej linii z języka ADA, co było jednym z założeń programu VHISC. Spowodowało to oczywiście, że oba języki są do siebie bardzo podobne. Najważniejsze cechy VHDL[89][104] to :

— oddzielenie specyfikacji interfejsu układu od opisu jego zachowania,
— kontrola typów, kompilatory sprawdzają czy typ danych występujący w danym kontekście jest zgodny ze specyfikacją języka, pozwala to uniknąć wielu prostych błędów,
— równoległość, domyślnie wszystkie (z pewnymi wyjątkami) instrukcje wykonywane są równolegle, ma to odzwierciedlać zachowanie rzeczywistego układu elektronicznego,
— możliwość specyfikacji opóźnień w wykonywaniu instrukcji,
— mechanizm przeciążania operatorów, tak jak np. w języku C++,
— wbudowane mechanizmy wspomagające symulację projektu,
— możliwość specyfikowania projektu na różnych poziomach abstrakcji - behawioralnym, przepływu danych, strukturalnym.

Nie wszystkie te możliwości da się wykorzystać dla celów syntezy logicznej[21][86]. Niektórych instrukcji i konstrukcji nie można przełożyć na specyfikację rzeczywistego układu elektronicznego. W kolejnych podrozdziałach przedstawione zostaną instrukcje i mechanizmy języka VHDL związane z tematem pracy. Pozostałe informacje na temat języka VHDL można znaleźć w literaturze: [21][86][89][104].

Sposób przedstawienia projektu w języku VHDL

Projekt w języku VHDL przedstawiony jest przy pomocy tzw. jednostki projektowej (ang. *design entity*)[77][79]. Jest to podstawowy mechanizm abstrakcji. Jedna jednostka projektowa tworzy pewną wydzieloną całość i może zawierać w swoim wnętrzu dowolnie skomplikowany układ (może również zawierać inne jednostki projektowe). Jednostka projektowa składa się z dwóch elementów:

— deklaracji interfejsu - *entity*,
— deklaracji architektury - *architecture*.

Interfejs jednostki projektowej

W deklaracji interfejsu (słowo kluczowe *entity*) dokonujemy dwóch specyfikacji:
— określenia unikalnych właściwości jednostki projektowej (*generic*),
— określenia sposobu komunikacji ze światem zewnętrznym.
Unikatowe właściwości układu są pewnego rodzaju parametrami. Mają określony typ. Deklarujemy je po słowie kluczowym generic. Ich wartość może być podana bezpośrednio przy deklaracji interfejsu, lub też w instrukcji deklaracji komponentu.

Komunikacja ze światem zewnętrznym odbywa się przy pomocy *sygnałów*. Deklarując port określamy, w jaki sposób (ang. *mode*) będzie można z niego korzystać. Port może być:
— *in* - wejściowy, może być tylko czytany (nie może wystąpić jako lewa strona przypisania),
— *out* - wyjściowy, może być tylko zapisywany (nie można korzystać z jego wartości),
— *inout* - port może być czytany jak i zapisywany.
Do deklaracji sygnałów w interfejsie służy słowo kluczowe *port*, a wygląda ona następująco:

<deklaracja_portów> ::= *port (<lista_portów>);*
<lista_portów> ::=
<lista_portów>; lista_identyfikatorów: <tryb> typ
| lista_identyfikatorów: <tryb> typ
<tryb> ::=
in | out | inout

Identyfikatory na liście oddzielone są przecinkami.

Architektura jednostki projektowej

Architektura określa to, w jaki sposób funkcjonuje dany układu, specyfikuje jego zachowanie. Dana architektura zawsze odnosi się do konkretnego interfejsu. Jeden interfejs może być skojarzony z wieloma różnymi architekturami. Architektura składa się z dwóch części:
— części deklaratywnej,
— zbioru instrukcji współbieżnych.
W części deklaratywnej znajdują się deklaracje typów, podtypów, procedur funkcji oraz sygnałów. Druga część architektury zawiera instrukcje współbieżne. Oznacza to iż nie ma znaczenia w jakiej kolejności są one umieszczone, ponieważ i tak zostaną wykonane niezależnie od siebie. Dopuszczalne jest wystąpienie następujących instrukcji współbieżnych:
— instrukcja procesu,
— współbieżne przypisanie do sygnału,
— współbieżne wywołanie procedury,
— instrukcja konkretyzacji komponentu,
— instrukcja bloku.

Dane w języku VHDL

W języku VHDL mam trzy rodzaje danych:
— sygnały,
— zmienne,

— stałe.

Sygnały

Deklaracja sygnału ma następującą postać:

signal lista_ identyfikatorów : typ;

Sygnały reprezentują połączenia. Jeżeli odnieść się do rzeczywistych układów elektronicznych to są one odpowiednikami przewodów (lub ścieżek drukowanych). Sygnały można deklarować wewnątrz architektury i pakietu.

Zmienne

Postać deklaracji zmiennej:

variable lista_ identyfikatorów : typ_ zmiennej.

Zmienne są obiektami do chwilowego przechowywania informacji (komórkami pamięci). Mogą być deklarowane wewnątrz instrukcji procesu, procedury lub funkcji, a także wewnątrz architektury. W tym ostatnim przypadku mamy do czynienia ze szczególną formą zmiennej - tzw. zmienną dzieloną shared. Nie określa się trybu dostępu do zmiennej każda zmienna może być zapisywana i czytana.

Typy danych

Język VHDL pozwala na stosowanie wielu typów. Nie wszystkie niestety są dopuszczane przez syntezę. Oto cztery grupy typów, jakie występują w języku VHDL:
— skalarne - proste,
— złożone,
— wskaźnikowe,
— plikowe.
Typ wskaźnikowy i plikowy nie jest możliwy do zsyntezowania i nie będzie omawiany w pracy.

Skalary

VHDL posiada cztery typy skalarne: wyliczeniowy (*enum*), całkowity (*Integer*) fizyczny (*physical*) oraz zmiennoprzecinkowy (*floating point*). Tylko dwa pierwsze są syntezowalne. Typ wyliczeniowy to zbiór znaków lub identyfikatorów, z których każdy jest reprezentowany przez określoną wartość liczbową. W tzw. pakiecie STANDARD (jest to podstawowa biblioteka, którą posiada każde narzędzie opierające się na języku VHDL) znajdują się definicje następujących typów wyliczeniowych: *boolean*, *bit*, *character*.

Jedynym dopuszczanym przez syntezę typem liczbowym jest typ całkowity - *Integer*. Zmienna lub sygnał zadeklarowany jako *Integer* może przyjmować wartości z przedziału -2147483647 do 2147483647. Rozmiar ten może zostać ograniczony za pomocą klauzuli *range*.

Typy złożone

Są to typy, które składają się z typów prostych lub złożonych. Istnieją dwa rodzaje typów złożonych: tablice oraz rekordy.

Tablica to zbiór elementów tego samego typu (prostego lub złożonego). Dostęp do danego elementu odbywa się za pomocą indeksu. Tablice mogą być wielowymiarowe. Większość narzędzi syntezy dopuszcza deklarowanie tablic jednowymiarowych, ale

ich elementami mogą być inne tablice jednowymiarowe. Podczas deklaracji tablicy, możemy określić wartość lewej i prawej granicy zakresu indeksu, a także kierunek, w jakim zmieniają się indeksy kolejnych elementów. Standardowymi typami tablicowymi w języku VHDL są: *bit_ vector* oraz *string*.

Typ rekordowy w odróżnieniu od tablicy może składać się z elementów różnego typu. Mogą one być proste oraz złożone. Każdy element danego typu rekordowego jest dostępny pod określoną nazwą.

Instrukcja procesu

Ogólna postać instrukcji przedstawia się następująco:

```
[etykieta:] process [(lista_ czułości)] [is]
                [część deklaracyjna]
        begin
                [instrukcje sekwencyjne]
        end process [ etykieta ] ;
```

Instrukcja procesu tworzy nowy niezależny wątek, który reprezentuje zachowanie pewnej części projektu. Wszystkie instrukcje wewnątrz procesu wykonywane są w sposób sekwencyjny, podobnie jak ma to miejsce w klasycznych językach programowania. Kolejność ich występowania ma znaczenie. W instrukcji procesu można wyróżnić następujące części:
— etykietę,
— listę czułości,
— część deklaracyjną,
— zbiór instrukcji sekwencyjnych.

Etykieta jest opcjonalna, jeżeli jednak wystąpiła na początku procesu musi wystąpić na jego końcu (taka sama). Lista czułości określa sygnały, których zmiana będzie miała wpływ na wykonanie procesu. Po wykonaniu ostatniej instrukcji proces zostaje zawieszony, do czasu kiedy zmieni się wartość któregokolwiek z sygnałów znajdujących się na liście czułości. Jeżeli występuje lista czułości proces nie może zawierać instrukcji *wait*. Część deklaracyjna może zawierać deklaracje typów, zmiennych, stałych, funkcji oraz procedur. Po słowie kluczowym *begin* występują instrukcje sekwencyjne.

Instrukcje sekwencyjne

Wewnątrz instrukcji procesu mogą znaleźć się następujące instrukcję sekwencyjne:
— przypisania sygnału,
— przypisania zmiennej,
— oczekiwania - *wait*,
— wywołania procedury,
— warunkowa - *if*,
— przypadku - *case*,
— pętli - *for*, *loop*, *while*,
— kontynuacji - *next*,
— wyjścia - *exit*,
— pusta - *null*.

Instrukcja przypisania sygnału

Ogólna postać instrukcji przypisania sygnału przedstawia się następująco:

cel_przypisania <= wyrażenie;

Instrukcja przypisania sygnału powoduje zmianę wartości sygnału. Po lewej stronie może wystąpić nazwa sygnału lub agregat. Przypisanie może dotyczyć prostego sygnału, jego części, lub zakresu. Jeżeli instrukcja dotyczy agregatu, każdy z jego elementów musi być sygnałem. Należy pamiętać, że nowa wartość sygnału zacznie obowiązywać dopiero po zakończeniu danej iteracji instrukcji procesu, a także, że jeżeli dany cel jest przypisywany więcej niż jeden raz wewnątrz procesu, to dopiero ostatnie przypisanie jest wiążące. Wszystkie wcześniejsze są ignorowane. Ogólne warunki poprawności instrukcji przypisania:
— lewa i prawa strona muszą być tego samego typu,
— lewa i prawa strona muszą mieć taki sam rozmiar.
Cel (lewa strona) przypisania, może wystąpić w jednej z następujących postaci:
— prostego sygnału - *a*,
— sygnału indeksowanego - *b(2)*,
— zakresu sygnału - *c(0 to 3)*,
— pola rekordu - *d.x1*,
— agregatu - *(a,b,c)*.

Instrukcja przypisania zmiennej

Ogólna postać instrukcji przedstawia się następująco:

cel_przypisania := wyrażenie;

Instrukcja przypisania zmiennej powoduje zmianę bieżącej wartości zmiennej. Rozróżnienie przypisania sygnały i zmiennej jest konieczne w związku z zupełnie innym zachowaniem obu rodzajów danych. Lewa strona może być w postaci nazwy zmiennej lub agregatu. Przypisanie może dotyczyć zakresu zmiennej, jej części lub całości. Zasady poprawności danej instrukcji przypisania są takie same jak w przypadku przypisania sygnału. Zmiana wartości danej zmiennej zachodzi natychmiast.

Instrukcja *if*

Składnia instrukcji *if* ma następującą postać:

```
if warunek1 then
  instrukcje_sekwencyjne
[elsif warunek2 then
  instrukcje_sekwencyjne]
[else
  instrukcje_sekwencyjne]
end if;
```

Umożliwia warunkowe wykonanie bloku instrukcji sekwencyjnych. Sposób działania jest następujący: warunki wejścia do poszczególnych gałęzi sprawdzane są kolejno, aż do momentu w którym któryś z nich okaże się być spełniony (lub dojścia do końca całej instrukcji). Dochodzi do aktywacji gałęzi i wykonania całego znajdującego się wewnątrz kodu. Następnie sterowanie przechodzi za instrukcję *if*, nie

sprawdzając pozostałych gałęzi (o ile pozostały). Widać więc, że może wykonać się co najwyżej jedna gałąź. *elsif* i *else* mogą, ale nie muszą wystąpić. Gałąź *elsif* może wystąpić więcej niż jeden raz.

Instrukcja *case*

Składnia instrukcji *case* przedstawia się następująco:

```
case wyrażenie_sterujące is
    when wyrażenie_wejściowe =>
        [instrukcje sekwencyjne]
    [when wyrażenie_wejściowe =>
        [instrukcje sekwencyjne]]
    [when others =>
        [instrukcje sekwencyjne[[
end case;
```

Wyrażenie wejściowe może przyjąć następującą postać:
— *wyrażenie_wejściowe* := *wartość1 | wartość2 | ...,*
— *wyrażenie_wejściowe* := *wartość1 to wartość,*
— *wyrażenie_wejściowe* := *wartość1 dowto wartość.*

Możliwe jest łączenie powyższych postaci. Słowo kluczowe *others* oznacza te wszystkie wartości wyrażenia sterującego, które nie wystąpiły w żadnej innej gałęzi. Może być tylko jedna gałąź z *others*.

Instrukcja case umożliwia wybór jednej z wielu alternatyw. W zależności od wartości *wyrażenia sterującego* występującego po słowie kluczowym *case* aktywowana jest jedna z gałęzi ta, której *wyrażenie wejściowe* występujące po *when* ma wartość taką jak wyrażenie sterujące. Należy pamiętać, że instrukcja ta dopuszcza aktywację więcej niż jednej gałęzi. Po zakończeniu działania w jednej z nich, następuje przejście do sprawdzania *wyrażenia wejściowego* w kolejnej.

Instrukcja *for*

Pętla *for* ma następującą składnie:

```
[etykieta:] for identyfikator in zakres loop
                [ instrukcje sekwencyjne ]
            end loop [etykieta];
```

Ilość iteracji pętli zależy od wartości *zakresu*. *Identyfikator* jest zmienną iteracyjną pętli, przyjmuje kolejne wartości z *zakresu* pętli i nie musi być wcześniej zadeklarowany. Co więcej, wewnątrz pętli przysłoni ewentualną zmienną lub sygnał o takiej samej nazwie. *Zakres* jest wyrażeniem zgodnym z zasadami syntaktycznymi i semantycznymi wyrażenia *range* języka VHDL. Można zastosować wszystkie postacie tego wyrażenia dopuszczane przez gramatykę języka.

Jeżeli program zapisany w języku VHDL ma być syntezowalny muszą zostać spełnione założenia odnośnie pętli *for*:
— zakres musi być statyczny, tzn. musi istnieć możliwość wyznaczenia go na etapie kompilacji,
— wewnątrz pętli nie może znajdować się instrukcja *wait*.

Oba warunki mają zapobiec sytuacji, w której pętla wykonywałaby się w nieskończoność. Ograniczenia te są zgodne ze specyfikacją pakietu FPGA Express firmy Synopsis[91].

Instrukcja *next*

Ogólna postać instrukcji przedstawia się następująco:

next [etykieta] [when warunek];

Instrukcja *next* przerywa bieżącą iteracje pętli. Po jej wykonaniu sterowanie powraca na początek pętli i możliwe jest rozpoczęcie kolejnej iteracji. Jeżeli instrukcja ta zawiera etykietę to następuje skok na początek pętli o tej właśnie etykiecie. W przeciwnym wypadku odnosi się ona do pętli, w której dana instrukcja *next* bezpośrednio się znajduje. Wykonanie instrukcji *next* może być zależne od spełnienia warunku występującego po słowie kluczowym *when*. Instrukcja ta może wystąpić tylko w pętli. Jeżeli zawiera etykietę to musi znajdować się wewnątrz pętli oznaczonej tą etykietą.

Instrukcja *exit*

Ogólna postać instrukcji przedstawia się następująco:

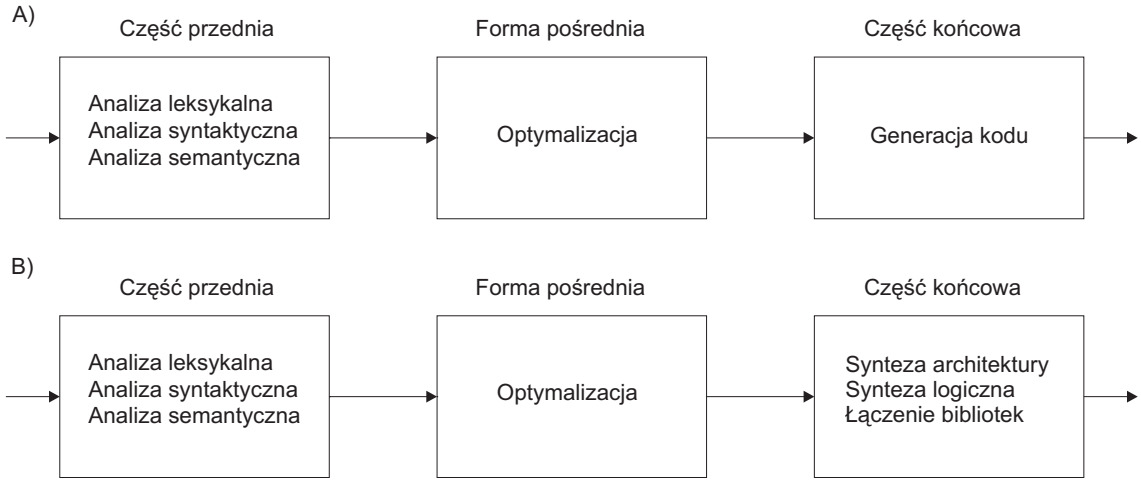
exit [etykieta] [when warunek];

Instrukcja *exit* powoduje przerwanie wykonywania pętli i przejście do pierwszej instrukcji po za nią. Jeżeli zawiera ona etykietę to przerywane jest wykonywanie pętli oznaczonej tą etykietą, w przeciwnym wypadku zakończeniu ulega pętla, w której dana instrukcja *exit* bezpośrednio się znajduje. Warunek ma takie samo znaczenie jak przy instrukcji *next*. Takie same są też wymogi dotyczące występowania instrukcji *exit*.

2.6. Budowa i zadania kompilatora języka opisu sprzętu

Zadaniem kompilatora języka opisu sprzętu jest przeniesienie modelu zapisanego w języku opisu sprzętu na niższy poziom abstrakcji. Poziom docelowy zależy od funkcjonalności danego narzędzia i wymagań projektu. Nie każdy kompilator musi umożliwiać syntezę do poziomu fizycznego, nie zawsze też projektant będzie takiej funkcjonalności potrzebował. Wszystko zależy od konkretnego przypadku. Wynikiem działania kompilatora jest model układu scalonego na innym (niższym) poziomie abstrakcji. Istnieje wiele rodzajów formatów wyjściowych, ale najczęściej wykorzystywanym jest tak zwana sieć połączeń (ang. *net-list*). Zawiera ona elementy składowe układu scalonego i mapę połączeń między nimi. Taki format ma tą zaletę, że pozwala na łatwe mapowanie technologii (rozdział 2.4).

Metodologie projektowania i implementacji kompilatorów języków opisu sprzętu wywodzą się w prostej linii z dziedziny kompilatorów tradycyjnych języków programowania[6][56][71]. Wynika to z faktu, że oba typy języków są ze sobą spokrewnione. Języki HDL powstawały i powstają najczęściej jako rozszerzenie klasycznych języków programowania. Rysunek 2.2 przestawia schemat obu rodzajów kompilatorów i trudno nie dostrzec uderzającego podobieństwa. Cześć dotycząca szeroko rozumianej analizy kodu (analiza leksykalna, syntaktyczna, oraz semantyczna) jest w obu przypadkach bardzo podobna. Odmienność na tym etapie wynika



Rysunek 2.2. Porównanie budowy kompilatorów: A) klasycznego oraz B) języka opisu sprzętu. Źródło: [24].

ze specyficznej składni języków HDL, np. konstrukcji wspierających równoległość, lub ograniczeń konkretnego narzędzia, np. nie wspierania pewnych mechanizmów języka. Na tym niestety kończą się podobieństwa, a pojawiają różnice. Zostanie to omówione szczegółowo w dalszej części tego rozdziału.

Wiedza dotycząca tworzenia klasycznych kompilatorów jest dobrze ugruntowana, a dziedzina ta ma długą, sięgającą lat pięćdziesiątych historię. Dostępnych jest wiele książek które krok po kroku opisują jak stworzyć tego typu oprogramowanie, dostosowane do indywidualnych potrzeb. Omówione są wszelkie aspekty kompilacji: analiza leksykalna, syntaktyczna, semantyczna, tworzenie kodu pośredniego, optymalizacja, zarządzanie pamięcią, oraz generacja kodu wynikowego. Jeżeli ktoś poszukuje rozwiązania w zakresie przetwarzania rozproszonego to taka wiedza także jest łatwo dostępna. Nie jest to tylko sucha teoria, bowiem często zawiera przykładowe algorytmy, które oczywiście są tylko pewnym uproszczonym modelem, ale znakomicie skracają czas potrzebny do stworzenie od podstaw nowego narzędzia. Nie ma przy tym znaczenia czy poszukiwane jest rozwiązanie dla języka imperatywnego, funkcjonalnego, obiektowego, czy logicznego. Praktycznie zawsze można znaleźć rozbudowaną wiedzę, która wymaga tylko dostrojenia do konkretnych potrzeb.

W przypadku kompilatorów języków HDL sytuacja jest inna. Brakuje wiedzy związanej ze specyfika tego typu narzędzi, czyli generowaniem reprezentacji układu scalonego. Bez wątpienia, jednym z powodów tego stanu rzeczy jest fakt, iż większość rozwiązań w tej dziedzinie stanowi własność firm, które nie są zainteresowane ich publikacją. Na dzień dzisiejszy nie ma dostępnych podręczników tej klasy i szczegółowości jak dla klasycznych kompilatorów. Szczegółowa sytuacja w tej kwestii przedstawiona została w rozdziale 2.9.

Po tym ogólnym wprowadzeniu przyszedł czas na dokładniejszy opis różnic w budowie obu typów kompilatorów. Jak widać na rysunku 2.2 w obu przypadkach narzędzie składa się z trzech części[5][24]:

— części przedniej - zajmującej się analizą leksykalną oraz syntaktyczną, tworzy ona formę pośrednią - będącej wewnętrzną reprezentacją analizowanego kodu źródłowego,
— formy pośredniej,

— części tylnej - generator kodu.

Część przednia zajmuje się analizą leksykalną, syntaktyczną i semantyczną. Jest więc zależna od specyfiki danego języka. Zadaniem analizatora leksykalnego jest poprawne rozpoznanie wszystkich typów leksemów takich jak: liczby, identyfikatory, operatory, oraz słowa kluczowe w strumieniu wejściowym. Następnie analizator syntaktyczny dokonuje przeglądu ciągu wygenerowanych leksemów pod kątem czy tworzą one zdania zgodne z gramatyką języka, usuwane są także komentarze. Ostatnim etapem jest sprawdzenie semantyki, czyli np. czy dwa operandy mają poprawne typy dla danej operacji. Każdy z analizatorów może być zrealizowany jako osobny program, a może też stanowić integralną część kompilatora (lub innego analizatora). To z ilu i jakich programów składa się kompilator jest już kwestią ściśle implementacyjną. W przypadku kompilatora do syntezy logicznej część przednia usuwa te konstrukcje języka, które nie mogą zostać zsyntetyzowane. Można też w tym kroku sprawdzić, czy instrukcje występujące w źródle spełniają ograniczenia jakie posiada dany kompilator[1].

Wynikiem działania części przedniej jest *forma pośrednia*. Stanowi ona wewnętrzny model kodu źródłowego i będzie inna w zależności od typu kompilatora, a tworzona jest najczęściej przez analizator semantyczny[2]. Można powiedzieć, że forma pośrednia spaja przód i tył kompilatora. Umożliwia także wykonanie wysoko-poziomowej optymalizacji[3]. Kompilatory klasyczne wykorzystują jako formę pośrednią np.: odwrotną notację polską, notację trójek, notacje czwórek. Wszystkie te formy są zapisem operacji i operandów. Ze względu na specyfikę zagadnień syntezy, kompilatory języków HDL najczęściej używają jako formy pośredniej różnych odmian grafu. W węzłach grafu znajdują się operację, a krawędzie oznaczają zależności. Taka postać dobrze uwypukla zależności między poszczególnymi operacjami oraz danymi co ma o tyle istotne znaczenie, że implementacja algorytmów w sprzęcie pozwala na zastosowanie przetwarzania równoległego.

Zadanie części tylnej polega na wygenerowaniu kodu wynikowego na podstawie formy pośredniej. Moduł ten jest zależny od przeznaczenia narzędzia. W przypadku klasycznego kompilatora będzie ukierunkowana na konkretną platformę sprzętowo-programową, a więc procesor oraz system operacyjny. W przypadku kompilatora języka HDL w części tej dokonuje się właściwego procesu syntezy (architektury i logicznej), łącznie z mapowaniem technologii i łączeniem bibliotek. W kroku tym niezależnie od typu kompilatora stosuje się rozbudowane mechanizmy optymalizacji, ściśle związane z docelową platformą.

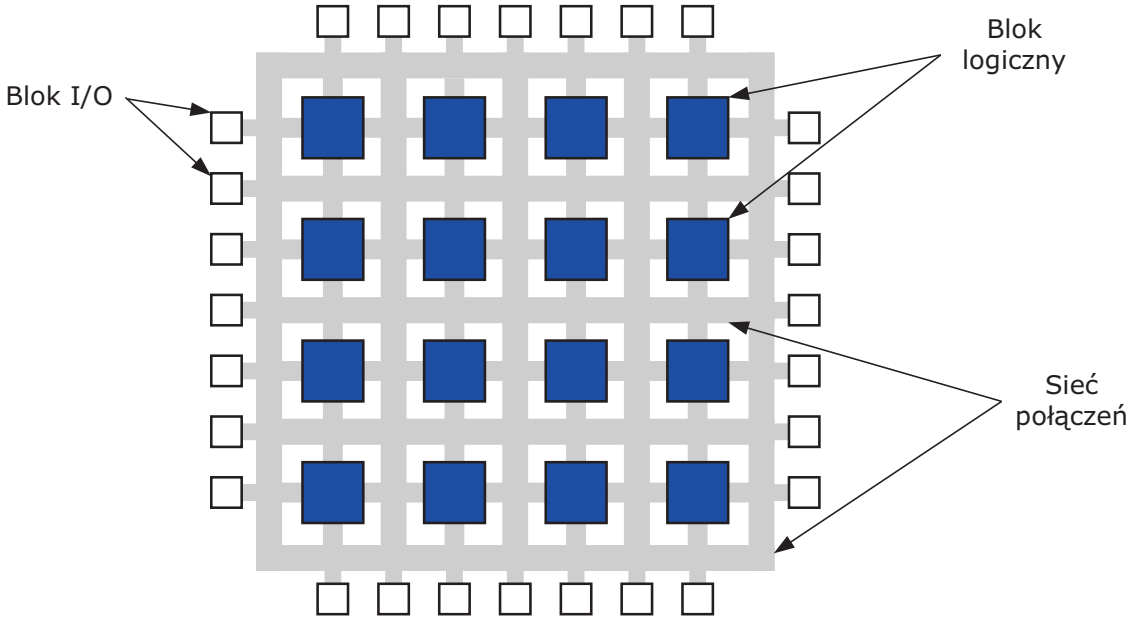
2.7. Układy FPGA

Field Programmable Gate Array[63] - czyli macierze bramek programowane przez użytkownika. Mianem tym określa się dużą rodzinę układów, których zachowanie, czyli realizowany algorytm nie jest zdeterminowany w momencie opuszczania fabryki półprzewodników. Implementacja algorytmu, polegająca na programowaniu odbywa się później i dokonuje jej projektant kształtując zachowanie układu według swojej woli (termin *„field programmable”*). Rodzina układów FPGA powstała, aby wypeł-

[1] Ograniczenia stosowania poszczególnych instrukcji języka HDL w syntezowalnym źródle są specyficzną kwestią każdego dostępnego kompilatora.

[2] Można oczywiście zastosować osobny podsystem/program.

[3] Optymalizacji niezależnej od docelowej architektury.



Rysunek 2.3. Schemat wewnętrznej budowy układu FPGA. Źródło: [63].

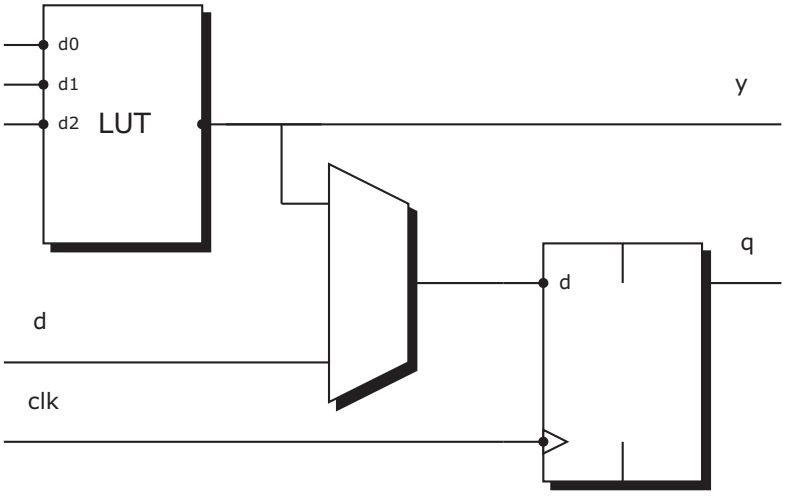
nić lukę między układami CPLD i ASIC. Pierwsze elementy tego typu datuje się na lata osiemdziesiąte, ale nie zdobyły one jednak popularności, ze względu na niezbyt duże oferowane możliwości. Gwałtowny rozwój rozpoczął się w dekadzie następnej i trwa do dnia dzisiejszego. Dzięki stałemu zwiększaniu się gęstości oraz polepszaniu się parametrów czasowych układy FPGA zaczynają zdobywać rejony do tej pory zarezerwowane dla technologii *full-custom*. Możliwość programowania (i reprogramowania) czyni z technologii FPGA idealne narzędzie do tworzenia prototypów nowych układów scalonych (najczęściej ASIC). Koszt realizacji pojedynczego projektu za pomocą tej technologii jest bardzo mały, a czas realizacji krótki, zwłaszcza jak zestawi się obie te wielkości z ich odpowiednikami przy zastosowaniu innych metod. Wadą FPGA jest koszt w przypadku produkcji masowej. Tutaj zdecydowanie ustępują technologii *full-custom*.

2.7.1. Budowa

Standardowy układ FPGA zbudowany jest według schematu przedstawionego na rysunku 2.3. Przypomina to trochę kratkę. Podstawowe elementy składowe to:
— bloki logiczne,
— bloki wejścia/wyjścia,
— sieć połączeń.

Blok logiczny

Pojedynczy blok logiczny zawiera z reguły w sobie moduł LUT (ang. *Look-up Table*) oraz przerzutnik. LUT służy do realizacji dowolnej funkcji logicznej, jedynym ograniczeniem jest liczba wejść, która jest oczywiście różna w zależności od konkretnego układu. Przerzutnik umożliwia synchronizację wyjścia bloku z innymi elementami i istnieje możliwość programowania trybu jego pracy: *latch*, *flip-flop*, oraz typu zbocza lub poziomu jakim jest wyzwalany. Rysunek 2.4 przedstawia przykładową architekturę bloku logicznego. Widać, że może on realizować logikę kombinacyjną i se-



Rysunek 2.4. Przykładowy schemat bloku logicznego układu FPGA. Źródło: [63].

kwencyjną. Może także służyć jako pamięć. Prezentowany model jest oczywiście bardzo prosty. W praktyce blok logiczny zwykle składa się z kilku (dwóch lub czterech) połączonych ze sobą takich modułów, co powoduje, że możliwe jest zrealizowanie funkcji logicznej o znacznie większej złożoności. Aby jeszcze zwiększyć możliwości takiego pojedynczego elementu umieszcza się wewnątrz układy arytmetyczno-logiczne.

Blok IO (wejścia-wyjścia)

Aby układ mógł współpracować z innymi elementami, musi istnieć możliwość doprowadzenia i wyprowadzenia z jego wnętrza sygnałów. Umożliwiają to wbudowane bloki wejścia-wyjścia. Przykład takiego bloku pochodzącego z rzeczywistego układu FPGA - Cyclone firmy Altera[8] pokazany jest na rysunku 2.5. Widać, że w zależności od zaprogramowania może pełnić funkcję wejścia lub wyjścia. Posiada przerzutniki, które pozwalają uczynić proces wymiany danych synchronicznym, ale oczywiście jest to zależne od projektanta.

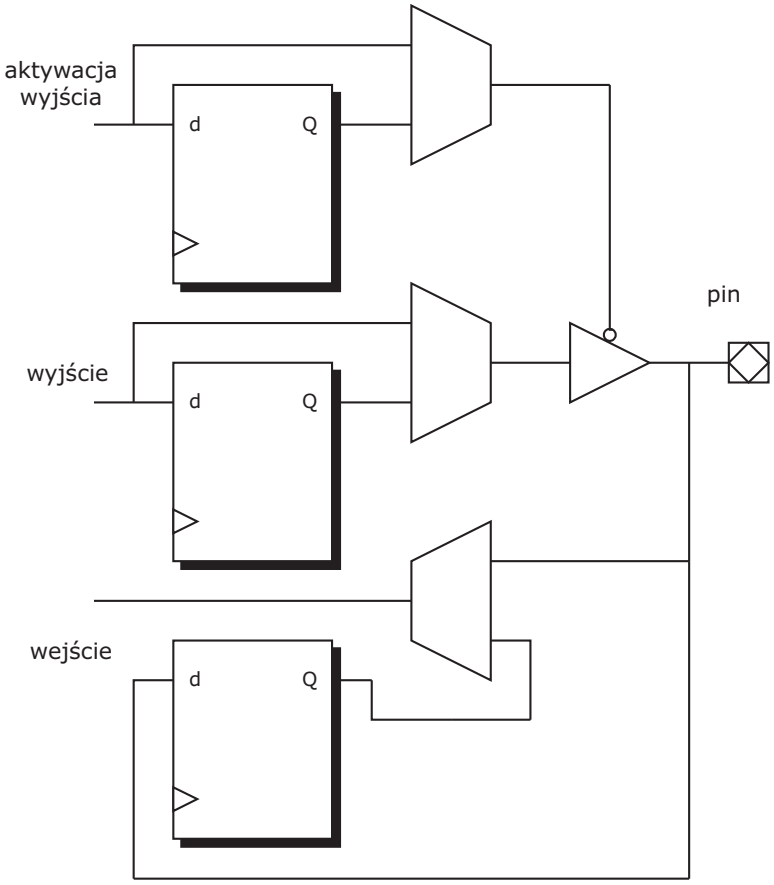
Sieć połączeń

Sieć połączeń umożliwia rozprowadzenie informacji między wszystkimi elementami znajdującymi się wewnątrz układu FPGA czyli blokami logicznymi oraz wejścia-wyjścia. Mimo, iż jest to bierny zasób nie należy lekceważyć jego istotności. Właściwe wykorzystanie ścieżek łączeniowych jest bardzo ważne, gdyż wpływa to na ostateczne parametry projektowanego układu.

Elementy dodatkowe

Powyższy opis dotyczył bardzo klasycznej architektury układów FPGA. Wraz ze wzrostem możliwości technologicznych, układy te wyposażane są w dodatkowe moduły funkcjonalne. Wspomniane zostało już wcześniej, że bloki logiczne mogą posiadać jednostki obliczeń arytmetycznych, ale nie jest to jedyne udogodnienie oferowane przez dzisiejsze układy programowalne. Układy FPGA mogą zawierać w sobie następujące dodatkowe moduły:

— pamięć RAM,
— procesor,



Rysunek 2.5. Przykładowy schemat bloku wejścia-wyjścia. Źródło: [8].

— dystrybutor (lub menadżer) sygnału zegarowego,
— interfejsy komunikacyjne.

Pamięć RAM. Wprawdzie bloki logiczne mogą być programowane, tak aby funkcjonować jako pamięć, ale biorąc pod uwagę ich pojemność trudno traktować takie rozwiązanie jako sensowne. Dzisiaj kiedy w układach FPGA implementowane są coraz bardziej skomplikowane algorytmy potrzebna jest duża ilość pamięci.

Procesor. Mikroprocesor stanowi doskonałe wsparcie. Można mu przekazać realizację operacji nie krytycznych czasowo, takich jak np. obsługa interfejsu użytkownika. Stanowi także doskonały kontroler całego układu. Procesory występują w dwóch odmianach: „miękkiej" (ang. *soft*) i „twardej" (ang. *hard*). Procesor „miękki" to wydzielona część normalnego układu FPGA, zaprogramowana tak, aby zachowywać się jak procesor. Wersja „twarda" to prawdziwy układ realizujący funkcje procesora, zatopiony wewnątrz elementu FPGA.

Menadżer sygnału zegarowego. Może się zdarzyć, że sygnał zegarowy. który jest doprowadzony z zewnątrz do układu FPGA nie jest odpowiedni dla jego działania. Menadżer zegara (ang. *clock manager*) umożliwia rozwiązanie takiego problemu, np. poprzez zmianę częstotliwości lub przesunięcia fazowego. Moduł taki często posiada możliwość dystrybuowania pochodnych sygnałów zegarowych o odmiennych para-

metrach wewnątrz kości FPGA. Zdarza się bowiem, że różne podsystemy układu pracują z innymi częstotliwościami.

Interfejsy komunikacyjne. Wprawdzie bloki wejścia-wyjścia umożliwiają wymianę danych między układem FPGA a jego otoczeniem, to nie zawsze jest to rozwiązania dobre. Wraz ze wzrostem ilości przesyłanych danych rośnie bowiem ilość wykorzystywanych ścieżek, a należy pamiętać, że powinny one mieć jak najbardziej zbliżoną długość aby nie było istotnych różnic w impedancji, gdyż może to powodować postawanie zakłóceń. Aby rozwiązać ten problem wyposaża się niekiedy układy FPGA w interfejsy gigabitowe.

2.7.2. Programowanie

Układy FPGA dzielą się na dwie rodziny: programowalne wielokrotnie oraz jednokrotnie. Pierwsze z nich działają w oparciu o zewnętrzną pamięć SRAM (ang. *Static RAM*, która przechowuje informacje o bieżącej konfiguracji układu, drugie korzystają z tak zwanych anty-bezpieczników (ang. *antifuse*), czyli specjalnych połączeń które w normalnym stanie nie przewodzą, a po podaniu odpowiedniego napięcia ulegają przepaleniu co powoduje spadek ich impedancji. Pewnym połączeniem zalet obu powyższych technologii są układy korzystające z pamięci EEPROM lub FLASH do przechowywania konfiguracji. W zależności od wersji, są one programowane przy pomocy dodatkowego urządzenia, lub w docelowym miejscu pracy. Po zaprogramowaniu zachowują się jak elementy typu anty-bezpiecznik, ale można je reprogramować.

Układy oparte o SRAM. Elementy tego typu dominują na rynku, a ich możliwości odzwierciedlają zawsze aktualny stan wiedzy w dziedzinie FPGA[22]. Są to układy rzeczywiście „programowalne w polu", nie wymagane jest żadne zewnętrzne urządzenie, aby zmienić ich konfigurację, a sam proces jest bardzo szybki. Czyni to z nich idealne narzędzia do prototypowania. Wady to:
— ulotność,
— brak gotowości do działania po włączeniu zasilania (potrzebny jest czas na wczytanie konfiguracji),
— brak odporności na promieniowanie, co wyklucza pewne obszary zastosowań,
— ograniczone bezpieczeństwo IP Cores, istnieje możliwość wydobycia konfiguracji zaprogramowanego układu i wykorzystania jej przez konkurencje, chociaż są już pewne metody zabezpieczeń ograniczające prawdopodobieństwo takiego przypadku.

Układy typu anty-bezpiecznik. Zwykle elementy te są jedną generację za swoimi reprogramowalnymi kuzynami. Programowanie tego typu układu wymaga specjalnego urządzenia (programatora) i może być wykonane tylko raz. W pewnych sytuacjach istnieje możliwość poprawy implementacji, ale odbywa się to na zasadzie przepalania kolejnych anty-bezpieczników, zatem wyklucza jakieś zasadnicze zmiany. Technologia ta posiada wszystkie te zalety, których nie daje zastosowanie SRAM, czyli: nieulotność, odporność na promieniowanie, gotowość do pracy po włączeniu zasilania i zabezpieczenie projektu przez kradzieżą. Jednak największą zaletą jest to, że proces programowania umożliwia prostą i skuteczną weryfikację projektu.

2.7.3. Zastosowania FPGA

Możliwości wykorzystanie układów FPGA są bardzo szerokie. Zamiast np. stosować duża liczbę tradycyjnych elementów scalonych, można użyć jednej kości programowalnej. Inne przykłady, to implementacje skomplikowanych algorytmów takich jak transformata Fouriera, algorytmy przetwarzania obrazów, czy też kryptograficzne. Układy FPGA mogą także zastępować procesory DSP i realizować przetwarzanie sygnałów[66][94][95]. Bardzo ciekawą gałęzią zastosowań jest użycie elementu FPGA jako koprocesora w różnych, ale mniej lub bardziej klasycznych systemach komputerowych [22]. Może on być przeznaczony do z góry ustalonych zadań, lub tez reprogramowany w trakcie pracy systemu stosownie do potrzeb. Oczywiście taka architektura wymaga specyficznego podejścia ze strony twórców oprogramowania. Nie jest bowiem możliwe wykonanie klasycznego kodu przez koprocesor FPGA. Należy zdecydować które fragmenty efektywniej będzie zrealizować za pomocą klastycznego mikroprocesora[35].Mimo iż układy FPGA pracują z dużo niższymi częstotliwościami niż klasyczne procesory to uzyskuje się znaczne przyśpieszenie w przypadku złożonych algorytmów. Wynika to z większej efektywności tych elementów i możliwości równoległego wykonywania zadań, a także oddzielenia sterowania, od przetwarzania[34]. Bardzo ciekawym przykładem wykorzystania rekonfigurowanego koprocesora są prace nad przyśpieszeniem działania programów napisanych w języku Java[36][76][87]. Ma to szczególnie istotne znaczenie dla platform mobilnych, o niewielkich możliwościach obliczeniowych, gdzie język ten jest dominujący jeśli chodzi o tworzone aplikacje. Inny przykład to wykorzystanie standardowej platformy badawczo-wdrożeniowej FPGA do przyśpieszenia syntezy logicznej[23]. Uniwersalne przeznaczenie reprogramowalnego koprocesora to cel twórców projektu GARP[37]. Przedstawione w tej pracy wyniki testów pokazują jak duże możliwości tkwią w implementacji przynajmniej części algorytmów na poziomie sprzętu. Ostatnim przedstawionym zastosowaniem będzie projekt FXP - *Field Programmable Port Extender*[55]. Jest to reprogramowalna platforma to przetwarzania pakietów komunikacji sieciowej[4].

2.8. Równania boolowskie

Równania boolowskie[50][61] są jedną z wielu notacji służących do przedstawiania funkcji logicznych. Jest to format matematyczny, ściśle sformalizowany, bazujący na aksjomatach algebry Boole'a. Cecha ta pozwala na minimalizowanie wyrażeń logicznych za pomocą formalnych przekształceń, a także zaowocowała powstaniem wielu skutecznych sposobów optymalizacji równań boolowskich takich jak: metoda Karnaugha czy Quine'a-McCluskey'a. Dla potrzeb syntezy układów PLA oraz specjalizowanych (*semi-* i *full-custom*) opracowano bardziej zawansowane rozwiązania takie, jak synteza dwu- i wielopoziomowa. Zagadnienia optymalizacji wyrażeń logicznych są dobrze opisane w literaturze fachowej[24][57][58].

Inną ważna zaletą równań boolowskich, być może nawet najważniejszą jest możliwość ich symulacji[68]. Właściwość ta pozwala na przetestowanie układu scalonego przed jego fizyczną implementacją, jest więc to cecha o ogromnym znaczeniu praktycznym z punktu widzenia inżyniera projektanta. I wreszcie, ten matematyczny

[4] Pełna dokumentacja tego i pokrewnych projektów znajduje się na stronie: http://www.arl.wustl.edu/projects/fpx/

format jest całkowicie niezależny od docelowej architektury sprzętowej. Stanowi to o jego uniwersalności i pozwala zastosować jako medium służące do przenoszenia projektów z jednego środowiska EDA do drugiego.

Wszystkie te cechy równań boolowskich zdecydowały o ich wykorzystaniu podczas badań przedstawionych w niniejszej pracy.

Mimo powyższych zalet, równania boolowskie nie są zbyt często wykorzystywane jako format wyjściowy w narzędziach do syntezy logicznej. Oprogramowanie tego typu oferuje zwykle kilka opcji w tej dziedzinie takich jak: format EDIF (ang. *Electronic Design Interchange Format*), strukturalny VHDL, XNF (ang. *Xilinx Netlist Format*), czy QDIF (ang. *QuickLogic Data Interchange Format*). W praktyce równania boolowskie generują pakiety EDA firmy Altera: MAX PLUS II[7] oraz Quartus II[10]. Można je obejrzeć przeglądając raport kompilacji. Niestety nie są dostępne żadne informacje wyjaśniające w jaki sposób oba narzędzia tworzą równania.

2.9. Prace pokrewne

Mimo aktualności tematu, liczba opracowań akademickich dotyczących kompilacji języka VHDL nie jest duża. Sytuacja ta jest szczególnie trudna, gdy poszukiwania dotyczą nie wiedzy ogólnej, ale przekrojowych opracowań obejmujących wszystkie aspekty procesu tworzenia kompilatora HDL, będących czymś na wzór podręczników tradycyjnych metod kompilacji. Pośród publikacji przeważają opracowania fragmentaryczne, skupiające się na jakiejś jednej, pojedynczej instrukcji albo też dotyczące określonego typu układów (pamięci, arytmetyka, etc.). Inną niedoskonałością większości prezentowanych rozwiązań, jest ich zbytnia ogólnikowość, nie jest prezentowany algorytm, a jedynie jego zarys, z reguły wsparty stosunkowo mało skomplikowanym przykładem. Wyróżnikiem tej pracy jest przedstawienie właśnie takiego kompleksowego rozwiązania wraz z algorytmami i dużą liczbą przykładów.

Niedocenianie równań boolowskich jako formatu wyjściowego wynika z obecności w świecie syntezy innych standardów (np. EDIF). Aby narzędzia różnych producentów mogły współpracować, konieczne jest aby posługiwały się tym samym formatem danych. Jeżeli ktoś tworzy nowe system do syntezy, symulator, lub jakiekolwiek inne produkt z tej dziedziny, to oczywiście zależy mu aby było elastyczne i także było w stanie działać z innymi pakietami. Logiczną konsekwencja tego jest użycie do zapisu wyników formatu (lub formatów), które są popularne. Poniżej zaprezentowane zostaną najciekawsze akademickie projekty narzędzi do kompilacji języka VHDL.

2.9.1. Extended Timed Petri Net

Pierwszy interesujący projekt koncentruje się na wykorzystaniu jako formy pośredniej będącej odmianą sieci Petri[4] o angielskiej nazwie *Extended Timed Petri Net* (ETPN). Efektem prac jest dostosowanie systemu CAMAD[78] do pracy z językiem VHDL. System ten w sposób jawny dzieli projekt na część sterującą i ścieżkę danych. Pierwszym krokiem jest otrzymanie formy pośredniej[27], która jest następnie transformowana celem optymalizacji i uzyskania informacji niezbędnych do syntezy. Otrzymywany jest graf ścieżki danych, oraz ETPN reprezentującą maszynę stanów niezbędną do sterowania projektowanym układem. Formatem wyjściowym zmodyfikowanego sytemu systemu CAMAD jest strukturalny kod języka VHDL na

poziomie RTL[26]. W toku dalszych prac autorzy koncentrują się na mechanizmach równoległości języka VHDL, a konkretnie synchronizacji procesów[26][28].

Analizując powyższe prace, należy zacząć od zastanowienia się czy narzędzie stworzone do symulacji, jakim są sieci Petri, może spełniać dobrze rolę formy pośredniej w kompilatorze do syntezy logicznej. Być może autorzy wyszli z założenia, że skoro VHDL ma swoje korzenie w temacie symulacji i musiał być przystosowany do zagadnień syntezy, to sieci Petri też da się przystosować. Widać, że formalnie to się udało, brakuje jednak porównania z innymi rozwiązaniami, co pozwoliłoby na ocenę jakości rozwiązania. Twórcy wprawdzie prezentują wyniki działania algorytmów, ale odnoszą się one do ilości miejsc i tranzycji sieci Petri, jakie otrzymano dla kilku przykładowych układów.

W prezentowanych wynikach brak jest szczegółowego opisu tworzenia ETPN dla poszczególnych instrukcji języka VHDL. Autorzy prezentują tylko kilka przykładów. Nie ma także kompleksowego algorytmu kompilacji kodu VHDL do postaci ETPN. Zmniejsza to w dużym zakresie przydatność pracy z praktycznego punktu widzenia.

2.9.2. Pakiet Alliance

Kolejnym narzędziem, a właściwie pakietem narzędzi jest Alliance[31][33]. To kompletny system będący w stanie realizować wszystkie zadania syntezy, składający się z kilkunastu oddzielnych programów, a także bibliotek i generatorów komórek. Pakiet obejmuje między innymi takie programy jak:
— symulator VHDL,
— optymalizator boolowski,
— narzędzie do syntezy logicznej,
— ekstraktor maszyny stanów,
— narzędzia place and route,
— konwerter VHDL.

Jest to przykład naprawdę kompleksowego podejścia do tematu syntezy w oparciu o język VHDL. Niestety ta imponująca praca nie jest pozbawiona mankamentów. Tyczą się one zwłaszcza wcześniejszych wersji. Najpoważniejszym zastrzeżeniem jakie można mieć do tego projektu, to ograniczenia, które musi spełnić program źródłowy, aby Alliance był w stanie go zsyntetyzować. Plik wejściowy musi być w jednej z trzech dopuszczalnych postaci[30][33]:
— behawioralnej - chociaż według większości literatury ten sposób opisu układu określa się mianem *przepływu danych* (ang. *data flow*), która może zawierać tylko instrukcję współbieżne za wyjątkiem procesu,
— strukturalnej - zawierającej jedynie deklarację komponentów i sygnałów,
— automatu skończonego (ang. *Finite State Machine*) - w postaci dwóch procesów o sprecyzowanej formie.

Należy zaznaczyć, że powyższych form opisu, nie można mieszać w jednym pliku źródłowym, co stanowi spore ograniczenie funkcjonalności pakietu.

Wraz z wersją 5[29][31] oprogramowanie zyskało jednak nowe możliwości. Stało się to za sprawą kolejnego narzędzia dołączonego do pakietu. *Vasy* (*VHDL Analiser for Synthesis*), bo tak nazywa się ten program, to konwerter, który potrafi zamienić dowolny program źródłowy VHDL na postać przyswajalną przez pozostałe narzędzia pakietu Alliance. Według dokumentacji *vasy* jest w stanie zaakceptować na wejściu kod VHDL zgodny ze podzbiorem syntezowalnym tego języka, wspieranym przez produkty firmy Synopsys[29]. Niestety nie jest to narzędzie idealne, ponieważ nie

potrafi sprawdzić poprawności pliku wejściowego. Utrudnia to oczywiście szukanie błędu w projekcie. Ale jest to krok naprzód, a autorzy obiecują, że na tym nie poprzestaną.

Niestety niewiele wiadomo o tym jak Alliance funkcjonuje wewnątrz. W dokumentacji nie ma słowa na temat przebiegu analizy poszczególnych instrukcji, ani jakiego rodzaju informacja semantyczna jest gromadzona. Pod tym względem przypomina to bardzo sytuację znaną z produktów komercyjnych.

2.9.3. Wykorzystanie grafu SIL

Innym przykładem wykorzystania VHDL dla potrzeb syntezy logicznej są prace Mekenkampa. Zaproponował on wykorzystanie formy SIL (*ang. SPRITE Input Language*)[70], czyli odmiany grafu CDFG (ang. *Control-Data Flow Grach*), jako formy przejściowej dla systemu syntezy[64][65]. Istotą podejścia jest kompilacja języka opisu sprzętu, takiego jak np. VHDL, (chociaż nie jest to ograniczenie) do postaci grafu SIL, a następnie dokonywanie wszelkich dalszych kroków procesu syntezy już na tej formie pośredniej. Kompilacja odbywa się na poziomie pojedynczych instrukcji, bez uwzględniania kontekstu w jakim zostały one umieszczone. Dodatkowo otrzymany graf SIL uwzględnia te mechanizmy VHDL, które są właściwe dla symulacji (delta, pętla procesu). Takie podejście miało rozszerzyć podzbiór syntezowalnych instrukcji VHDL. Należy zadać sobie pytanie, czy tego typu postępowanie jest zasadne, bowiem sporą część pracy[65] zajmuje usuwanie niesytenzowalnych mechanizmów z grafu SIL. Z analizy prac wynikają następujące fakty:

— możliwe jest otrzymanie syntezowalnego źródła VHDL dla danego grafu SIL,
— kompilacja VHDL (lub innego HDL) do postaci SIL jest tylko etapem przygotowawczym,
— SIL jest czymś więcej niż tylko formą wewnętrznej reprezentacji układu logicznego w kompilatorze - to na niej dokonywana jest synteza,
— synteza odbywa się przy pomocy predefiniowanych transformacji i polega na wybraniu najbardziej odpowiednich z nich,
— transformacja to przejście od jednego grafu SIL, do drugiego,
— jest możliwe skonstruowanie oprogramowania dokonującego automatycznej transformacji, według określonych wcześniej wytycznych.

Narzędzie, które ma umożliwiać syntezę według opisanej metodologii, musi:

— być zorientowane na użytkownika (ang. *user-centred*), czyli zakładać świadomą obecność człowieka podczas całego procesu,
— kontrolować, czy konkretna wybrana przez operatora transformacja jest dopuszczalna.

Aby więc projektant był w stanie zamienić swój kod VHDL w układ scalony, musi najpierw dobrze poznać SIL oraz dostępne transformacje. Po kompilacji źródła VHDL do postaci grafu, powinien pracować tylko na nim, inaczej bowiem będzie musiał powtarzać transformację po każdej zmianie pliku wejściowego.

2.9.4. System VIS

VIS (ang. *Veryfication Interacting with Synthesis*) to dzieło dużego zespołu z Berkeley (University of California). Pakiet ten służy do weryfikacji oraz ewentualnej syntezy sekwencyjnych układów logicznych[96][97][101]. Narzędzie to korzysta z formy pośredniej o nazwie BLIF-MV[52], która jest rozwinięciem formatu BLIF (ang.

Berkeley Logic Interchange Format)[99]. W tej chwili pakiet potrafi skonwertować do postaci BLIF-MV tylko język Verilog, ale narzędzie ma taką budowę, że rozszerzenie go o moduł obsługi kolejnego języka opisu sprzętu nie jest problemem. Aby dokonać syntezy niezbędne jest wykorzystanie innego oprogramowania z Berkeley, mianowicie pakietu SIS[88]. Oba narzędzia współpracują poprzez format BLIF.

2.9.5. Wyniki badań Tomasza Wiercińskiego

Przedstawiając stan wiedzy w dziedzinie nie można pominąć wyników uzyskanych podczas prac nad wspomnianym wcześniej w rozdziale 1.1 projektem kompilatora, ale dotyczących innych jego elementów. Szczególne wyróżnienie należy się badaniom Tomasza Wiercińskiego, który zajmował się problemem generowania równań boolowskich dla źródeł VHDL zawierających logikę sekwencyjną[102]. Badania te koncentrowały się przede wszystkim na instrukcji *wait* języka VHDL[16]. Instrukcja ta umożliwia bardzo precyzyjne sterowanie wykonaniem bloku *process*, ale jest trudna do syntezy, gdyż jej użycie pociąga za sobą konieczność utworzenia automatu skończonego[1][2]. Dostępne na rynku narzędzia nakładają ograniczenia na stosowanie tej instrukcji w źródle przeznaczonym do syntezy[91]. Autorowi udało się znacząco zwiększyć zakres stosowalności tej instrukcji, w porównaniu do rozwiązań już istniejących na rynku. Badania przedstawione w niniejszej pracy oraz publikacje Tomasza Wiercińskiego uzupełniają się wzajemnie, dostarczając wiedzy niezbędnej do wykonania kompilacji i syntezy instrukcji sekwencyjnych języka VHDL.

2.10. Rozwiązania komercyjne

Rynek narzędzi do projektowania układów w oparciu o technologię FPGA jest bardzo szeroki. Poszczególne produkty różnią się miedzy sobą w zależności od swojego przeznaczenia, czyli miejsca w cyklu tworzenia układu scalonego. Mamy, więc: symulatory, pakiety do syntezy, narzędzia do weryfikacji po syntezie, oraz programy dokonujące alokacji zasobów modelu rzeczywistego układu (ang. *place and route*). Wreszcie dostępne są wielofunkcyjne kombajny, które integrują kilka, a nawet wszystkie etapy tworzenia nowego układu Reprogramowalnego. Przykładami firm które produkują oprogramowanie tego typu są: Altera, Xilinx, Synopsys, Mentor Graphics. Dwie pierwsze firmy są przede wszystkim producentami układów FPGA. Obie udostępniają za darmo oprogramowanie do syntezy logicznej, przygotowane do pracy z oferowanymi przez siebie układami. Dokumentacja dołączana do programów jest dokumentacją użytkownika. Nie można się z niej dowiedzieć, w jaki sposób program uzyskuje takie a nie inne wyniki. Prezentowana jest tylko informacja, jakie skutki będzie miało zastosowanie takiej, a nie innej konstrukcji języka VHDL. Jest to oczywiście utrudnienie dla badaczy, ale trudno winić firmy za taką postawę. Rywalizacja na rynku jest bardzo silna i publikowanie własnych rozwiązań, nawet jeśli by się je zastrzegło patentami byłaby zbyt dużym ułatwieniem dla konkurentów. Narzędzia komercyjne pozostają więc dla badacza „czarnymi skrzynkami", które można używać jako aplikacji referencyjnych, wykorzystywanymi do oceny własnych wyników. W następnym rozdziale omówione zostanie dokładniej jedno z narzędzi komercyjnych, aby zademonstrować lepiej to wszystkie co oferuje dostępne na rynku oprogramowanie.

2.10.1. Altera Quartus II

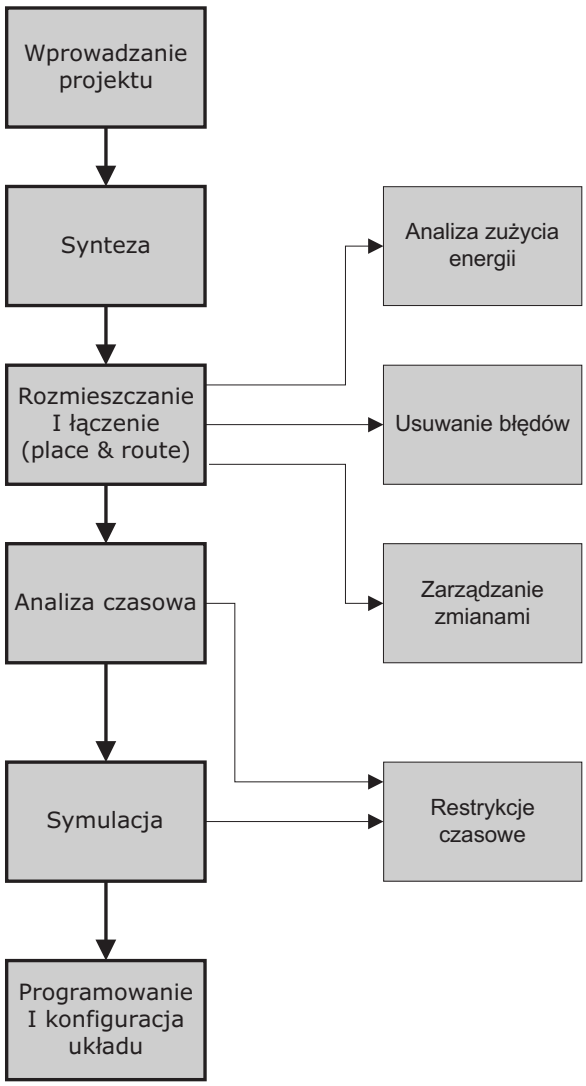
Program ten, wybrany został ze względu na przejrzystą i obszerną dokumentację. Narzędzia konkurencyjnych firm prezentują się bardzo podobnie. Prezentowane poniżej informacje łącznie z rysunkami pochodzą z oficjalnych dokumentów opublikowanych przez firmę Altera[9][10]. Quartus II jest zintegrowanym środowiskiem do tworzenia projektów systemów w oparciu o układy programowalne SOPC (ang. *System-On-a-Programmable-Chip*). Przez układy programowalne rozumiemy PLD, CPLD oraz FPGA. Ponieważ Altera jest przede wszystkim dostawcą tego typu układów, a oprogramowanie do projektowania i syntezy stanowi mimo wszystko działalność poboczną, oczywistym jest, że oferowane narzędzia są zorientowana na silikonowe rodziny oferowane przez firmę. Pakiet umożliwia współpracę i wykorzystanie w procesie tworzenia układu oprogramowania innych firm. Odbywa się albo za pomocą wbudowanego interfejsu, lub też za pomocą interfejsu zewnętrznego, czyli np plików EDIF.

Proces tworzenie projektu za pomocą Quartusa II przebiega według schematu przedstawionego na rysunku 2.6. Widać, że realizacja układu programowalnego rozbita jest na wiele etapów. Oprogramowanie firmy Altera wspiera projektanta na każdym z etapów.

Wprowadzanie projektu (ang. *Design Entry*). Zadaniem tego etapu jest dostarczenie oprogramowaniu modelu układu scalonego, którego fizycznej implementacji chcemy dokonać. Model układu może być w postaci:
— plików zawierających opis w języku HDL (VHDL, Verilog),
— sieci połączeń (ang. *net-list*), pliki formatu EDIF, pochodzące z innego narzędzia EDA,
— diagramu blokowego (pliki z rozszerzeniem bdf).
System umożliwia wczytywanie gotowych plików (HDL, diagram bloków) jak i posiada zintegrowane edytory do ich tworzenia. Szczegółowo możliwości w tym zakresie przedstawia rysunek 2.7. Oprócz wprowadzenia projektu jako takiego, można także określić ograniczenia jakie musi spełnić tworzony układ. Obostrzenia mogą dotyczyć:
— właściwości czasowych (ang. *timing closures*),
— przypisania nóżek (ang. *pin assignment*),
— zużycia energii.
Można także określić wszelkie ustawienia odnoszące się do pracy samego narzędzia, współpracy z innymi programami EDA.

Synteza (ang. *Synthesis*). Po wprowadzeniu projektu, można przystąpić do jego dalszej obróbki, czyli syntezy. Zadaniem tego etapu jest zamiana wprowadzonego modelu na postać możliwą do implementacji w sprzęcie. Schemat procesu syntezy za pomocą programu Quartus II przedstawia rysunek 2.8. Pakiet umożliwia wykorzystanie do tego celu narzędzi zewnętrznych. Wynikiem końcowym etapu jest baza danych zawierająca zoptymalizowany i zmapowany technologicznie projekt. Można oczywiście obejrzeć otrzymany model zarówno w wersji RTL, jak i po mapowaniu technologii.

Rozmieszczanie i łączenie (ang. *Place and Route*). Po wykonaniu mapowania technologii można przystąpić do wpasowywania projektu w konkretny układ programowalny. Quartus II na podstawia wprowadzonych wcześniej ograniczeń robi

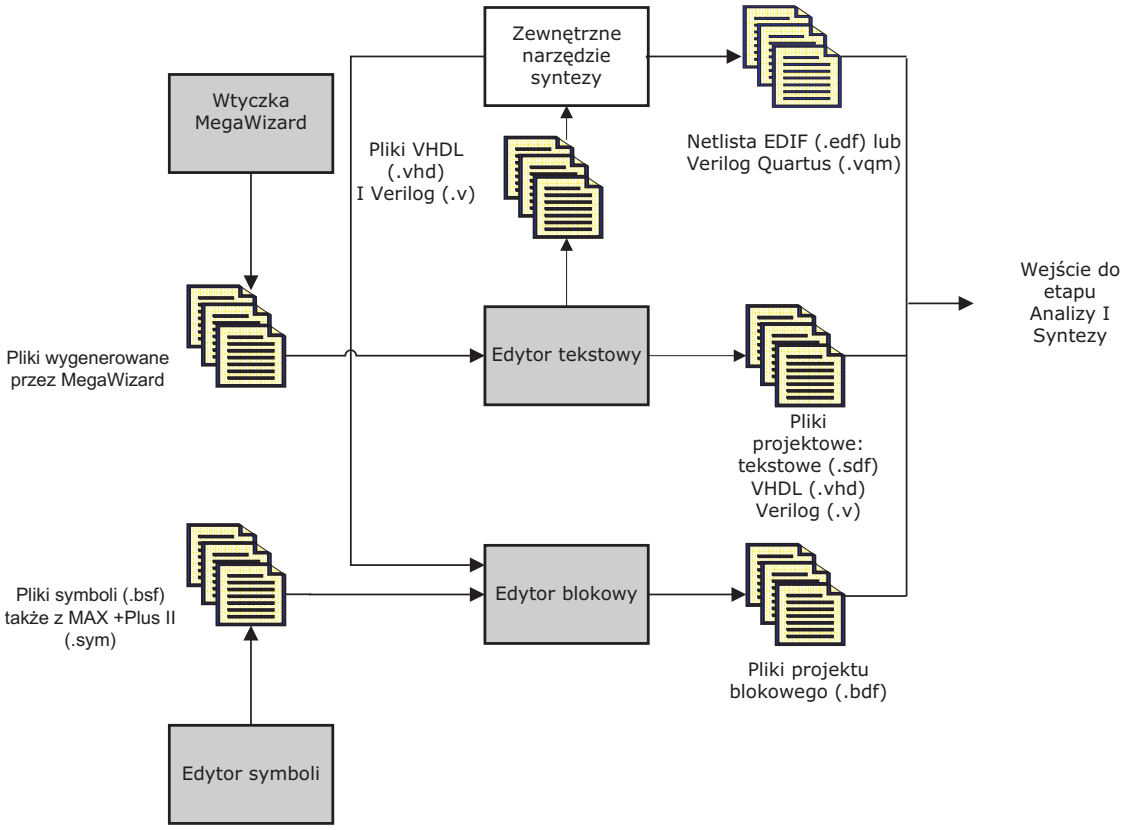


Rysunek 2.6. Fazy projektu w środowisku Altera Quartus II. Źródło: [9].

to automatycznie, ale oczywiście jest możliwość ingerencji w ten proces, aby uzyskać bardziej optymalny projekt. Jeżeli nie ma sposobu, aby spełnić wymagania użytkownika, program sygnalizuje błąd.

Analiza czasowa (ang. *Timing Analysis*). Zadaniem tego kroku jest uzyskanie informacji na temat właściwości czasowych projektowanego układu. Pozwala to sprawdzić, czy parametry te spełniają wymagania narzucone przez użytkownika. Lista charakterystyk, których badanie umożliwia program zawiera wiele pozycji. Można uzyskać informację między innymi o maksymalnej częstotliwości pracy układu *fMAX*.

Symulacja (ang. *Simulation*). Po rozmieszczeniu projektu w docelowym układzie, kolejnym krokiem jest symulacja, celem weryfikacji założeń i ograniczeń. Symulacja może być zrealizowana zarówno przez samego Quartusa jak i przez symulator zewnętrzny. Pakiet zapewnia interfejs natywny (*Native Link*©) do większości oprogramowania symulacyjnego dostępnego na rynku. W zależności od tego czy chcemy



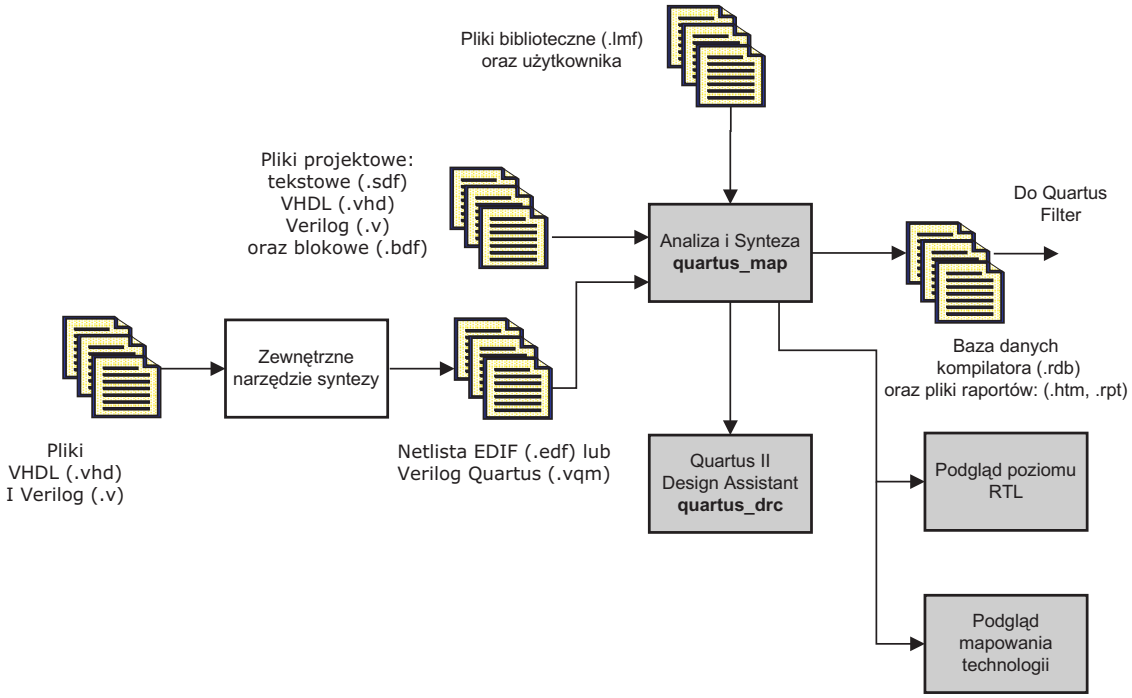
Rysunek 2.7. Wprowadzanie danych. Źródło: [9].

zweryfikować projekt pod kątem: poprawności funkcjonalnej, ograniczeń czasowych czy też zużycia energii, przebieg tego procesu będzie się trochę różnić.

Programowanie i konfiguracja układu (ang. *Programming*). To ostatni krok w procesie projektowym. Podczas niego tworzone są pliki programujące i konfigurujące fizyczny układ. Quartus II umożliwia wygenerowanie plików w różnych formatach, w zależności od potrzeb. Pakiet jest w stanie także zaprogramować układ scalony jeżeli posiadamy wspierany przez niego sprzęt.

2.11. Podsumowanie

W niniejszym rozdziale przedstawione zostały podstawowe informacje na temat procesu wytwarzania układów scalonych, a także wyjaśniono pojęcie syntezy logicznej. Omówione zostały krótko technologie produkcji współczesnych półprzewodników, wraz z typowymi przypadkami ich stosowania. Szczególne dużo miejsca przypadło układom FPGA. Rozdział ten przybliżył także czytelnikowi języki opisu sprzętu, ich historię oraz możliwości. Następnie opisany został stan wiedzy w dziedzinie, z uwzględnieniem zarówno świata akademickiego, jak i firm komercyjnych. Z analizy dostępnych rozwiązań wynika, że ciągle jeszcze nie ma narzędzia które by pozwalało w pełni wykorzystać możliwości jakie oferuje język VHDL. Nadal wiele mechanizmów jest niesyntezowalnych, lub ich stosowanie wymaga spełnienia określonych warunków. Powszechnie dostępna wiedza nie dostarcza odpowiedzi na pytanie, jak powinien wyglądać modelowy kompilator języka VHDL przeznaczony do zadań



Rysunek 2.8. Synteza w środowisku Quartus II. Źródło: [9].

syntezy logicznej. Brak jest przykładów kompleksowych i dobrze udokumentowanych rozwiązań. Ta luka w dostępnej wiedzy z zakresu metod kompilacji uzasadnia podjecie tego tematu w niniejszej pracy. Pojawia się bowiem potrzeba opracowania teorii dotyczącej projektowania i implementacji tego typu narzędzi.

3. Kompilacja źródeł w języku VHDL do postaci równań boolowskich

W niniejszym rozdziale przedstawione są zasady generacji równań boolowskich ze źródeł w języku VHDL. Prezentowana wiedza teoretyczna zawiera liczne przykłady co ułatwia jej zrozumienie. Przyjęta konwencja opisu jest taka, aby rozpatrywać każde zagadnienie kompilacji oddzielnie. Ułatwia to poruszanie się po pracy, a także jej zrozumienie. Przez pojęcie pojedynczego zagadnienia rozumie się wyodrębnioną, stanowiącą pewną całość grupę problemów i zadań kompilacji. Najczęściej będzie ono dotyczyło jednej konkretnej instrukcji, ale nie zawsze. Pierwszym krokiem jest prezentacja formatu równań boolowskich, jaki został wykorzystany w badaniach, a następnie omówione jest środowisko które pozwala na użycie instrukcji sekwencyjnych - instrukcja *process*. Po tym wstępie następuje przejście do opisu kompilacji poszczególnych instrukcji języka VHDL.

3.1. Wiadomości wstępne

W języku VHDL można korzystać z wielu typów danych: prostych, złożonych, wbudowanych, a także definiowanych przez użytkownika. Bez względu na to, jak skomplikowana jest ich budowa, są one tylko pewną umowną abstrakcją, która w procesie syntezy jest zamieniana na grupę bitów. Podstawowym elementem na poziomie równań, jest bowiem pojedynczy bit. Generalną zasadą tworzenia równań boolowskich jest to, że dla każdego bitu wszystkich sygnałów i zmiennych, (jeżeli stanowi on cel przypisania, gdziekolwiek wewnątrz procesu) tworzone jest jedno równanie, nawet, jeśli dany sygnał czy zmienna jest zapisywana wiele razy. Od tej reguły są pewne wyjątki (zwłaszcza odnoszące się do zmiennych), które zostaną szczegółowo omówione w dalszej części pracy. Nie mniej jednak jest to podstawowe założenie całego procesu kompilacji i generacji równań.

Wzorce równań będą prezentowane w postaci formuł matematycznych wraz z niezbędnym komentarzem. Taki zapis wydaje się być zasadny, gdyż jest krótki, przejrzysty i łatwy to zrozumienia.

W pracy tej nie zostaną zaprezentowane algorytmy tworzenia równań boolowskich dla wyrażeń, bowiem stanowi to przedmiot osobnych badań [54][69]. Istotna jest informacja, że autor rozprawy miał dostęp do zaimplementowanych metod generacji tychże równań.

3.2. Format równań boolowskich

Równania boolowskie są sposobem przedstawienia układu cyfrowego. Używając nawet tylko podstawowych operacji znanych z algebry Boole'a można zrealizować skomplikowane algorytmy. Nadają się one zarówno do opisu logiki kombinacyjnej,

jak i sekwencyjnej. Równania wykorzystywane w pracy bazują na tylko trzech operacjach logicznych: sumie, iloczynie oraz negacji. Jest to wystarczające to przedstawienia dowolnie skomplikowanej funkcji. Jedno pojedyncze równanie odpowiada jednemu bitowi sygnału lub zmiennej, aby więc odzwierciedlić bardziej skomplikowane operacje należy utworzyć grupę równań. Może ona reprezentować np. sumator lub przerzutnik.

Do oznaczania poszczególnych operacji wykorzystuje się następujące znaki:
— | dla sumy,
— & dla iloczynu,
— ! dla negacji.

Nazwy, które występują w równaniach, odpowiadają nazwom zmiennych i sygnałów znajdującym się w kompilowanym kodzie języka VHDL. Mogą wystąpić także nazwy pomocnicze, szczególnie w przypadku zmiennych i bloków równań tworzących przerzutniki.

Jeżeli dana zmienna, lub sygnał jest typu tablicowego, lub też do jej reprezentacji konieczne jest użycie więcej niż jednego bitu (np. *integer*), to w kodzie wynikowym będzie występować w postaci indeksowej. Do oznaczania indeksu używa się nawiasów okrągłych, a indeksy poszczególnych wymiarów oddziela się przecinkami.

3.3. Środowisko pracy prezentowanych algorytmów

Jak już zostało to wspomniane wcześniej, powyższa praca nie zawiera metod generacji równań dla wyrażeń, lecz nieustannie z ich korzysta. W związku z tym, należy opisać format i sposób wymiany informacji między modułem odpowiedzialnym za tworzenie równań dla wyrażeń.

Aby otrzymać równania dla przypisań wywołuje się funkcję (inną dla sygnałów i zmiennych), która jako parametr pobiera tablice leksemów składających się na przypisanie. Jako wynik zwracana jest wielowymiarowa tablica zawierająca równania w postaci łańcuchów znaków. Każde opisuje jeden elementarny bit sygnału lub zmiennej. Należy następnie tak otrzymane równania zapisać w wewnętrznych strukturach danych. W tym celu analizuje się lewą stronę (stanowiącą cel przypisania) każdego z nich. Ma ona postać nazwy, a jeżeli jest to typ tablicowy czy rekordowy, to każde równanie zawiera wszystkie nazwy i indeksy pośrednie. Aby odnaleźć w strukturach danych właściwe miejsce do zapamiętania informacji, konieczne jest więc dokonanie przejścia od pierwszego członu nazwy celu przypisania do samego jej końca.

3.4. Informacja semantyczna skojarzona z generowanymi równaniami

Dla jednego bitu sygnału, lub zmiennej zawsze powstaje jedno równanie (grupa równań), bez względu na ilość przypisań jaka wystąpiła wewnątrz instrukcji *process*. Aby więc uzyskać ostateczną formę równania należy wygenerować równania pośrednie, a następnie poskładać je w całość. Potrzebna jest do tego celu dodatkowa informacja semantyczna, tworzona i zbierana podczas analizy całego bloku instrukcji *process*. Ta informacja jest przechowywana wraz z równaniami boolowskimi.

Aby generacja równań przebiegała sprawnie konieczne było zaprojektowanie odpowiednich struktur do przechowywania równań pośrednich oraz skojarzonej infor-

macji semantycznej. Należy przy tym pamiętać, że dane w języku VHDL mogą przybierać bardzo skomplikowane postacie, takie jak tablice, czy też rekordy. Nawet taki prosty jak by się wydawało typ *integer* na poziomie RTL jest tablicą bitów. A tablice mogą być wielowymiarowe, nie mówiąc już o komplikacjach, jakie są powodowane przez rekordy. Struktura danych do przechowywania równań i niezbędnej informacji semantycznej dla danego sygnału, zmiennej musi, więc mieć postać taką, która umożliwi odwzorowanie dowolnie złożonego typu dopuszczanego przez język.

Można wyróżnić dwa zasadnicze poziomy w Strukturze Przechowującej Równania (SPR):
— poziom zmiennej (sygnału),
— poziom bitu.
Poziom zmiennej zawiera globalne informacje takie jak:
— unikalny identyfikator, każda zmienna czy też sygnał jest jednoznacznie identyfikowana, oraz posiada zbiór skojarzonej ze sobą informacji semantycznej[19],
— rozmiar, wyrażany ilością bitów niezbędną do przechowywania wszystkich jej możliwych wartości, w przypadku tablic wielowymiarowych będzie to ilość elementów niższego rzędu,
— wartość lewej i prawej granicy indeksu, w przypadku zmiennych (sygnałów) indeksowanych, jest to konieczne ze względu na możliwość dowolnego określania tych danych na etapie tworzenia typu, lub zmiennej.

W przypadku zmiennych (sygnałów) będących bytami złożonymi, informacja będzie w postaci hierarchii o wielu poziomach *zmiennej*.

Z każdym pojedynczym bitem skojarzona jest następująca informacja semantyczna:
— równanie boolowskie w postaci kombinacyjnej,
— równania logiki sekwencyjnej,
— typ logiki,
— w przypadku zmiennej informacja czy dany bit był już zapisywany czy nie.

Nie zawsze istnieje konieczność przechowywania równań obu typów (sekwencyjnych i kombinacyjnych). Dzieje się tak tylko wtedy, gdy podczas analizy danej instrukcji nie można określić z całą pewnością, jakiego rodzaju logika z niej powstanie po uwzględnieniu wszystkich instrukcji wewnątrz danego ciała procesu. Najczęściej dotyczy to przekładu instrukcji *if* oraz *case* i to tylko w przypadku zawierania przez nie przypisań do zmiennych.

3.5. Algorytm generacji równań boolowskich dla instrukcji *process*

Zgodnie z informacjami przedstawionymi we wstępie (rozdział 1.2), niniejsza praca prezentuje algorytmy niezbędne do wykonania automatycznego przekładu następujących instrukcji języka VHDL:
— procesu (ang. *process*),
— procedury (ang. *procedure*),
— funkcji (ang. *function*),
w zakresie logiki kombinacyjnej. Przekład wszystkich trzech konstrukcji wygląda bardzo podobnie. Różnice dotyczą:
— obsługi argumentów (procedura i funkcja),

— zwracania wyniki (funkcja),
— generowania przerzutników (proces).

Ograniczenia powyższe oznaczają, że dopuszczalne są tylko te instrukcje sekwencyjne języka VHDL, dla których można wygenerować równania logiki kombinacyjnej. Od tej zasady są pewne wyjątki, które związane są ze sposobem funkcjonowania instrukcji *if* oraz *exit*. Zagadnienie to wyjaśnione jest w rozdziale 3.9.2.

Prezentowane opracowanie powstało jako część znacznie większej, opisującej cały zakres zagadnień dotyczących kompilacji kodu VHDL do postaci równań boolowskich teorii. Wiedza ta powstała w związku z konkretną potrzebą dotyczącą zrealizowania w pełni funkcjonalnego kompilatora umożliwiającego syntezę. Ponieważ zagadnienie to jest bardzo obszerne, zostało podzielone na mniejsze części, stanowiące logiczną całość części. Z tych powodów kompilacja instrukcji *process* została rozdzielona na część kombinacyjną i sekwencyjną. Mimo jednak tego podziału, zagadnienie to stanowi pewną całość, a zatem konieczne jest takie do niego podejście, które umożliwiałoby przekład bez dokonywania wcześniejsze analizy, z jakiego rodzaju logika mamy do czynienia. Jednym słowem nadrzędny algorytm generowania równań boolowskich dla instrukcji *process* musi zawierać algorytmy cząstkowe dla każdej z dopuszczalnych konstrukcji.

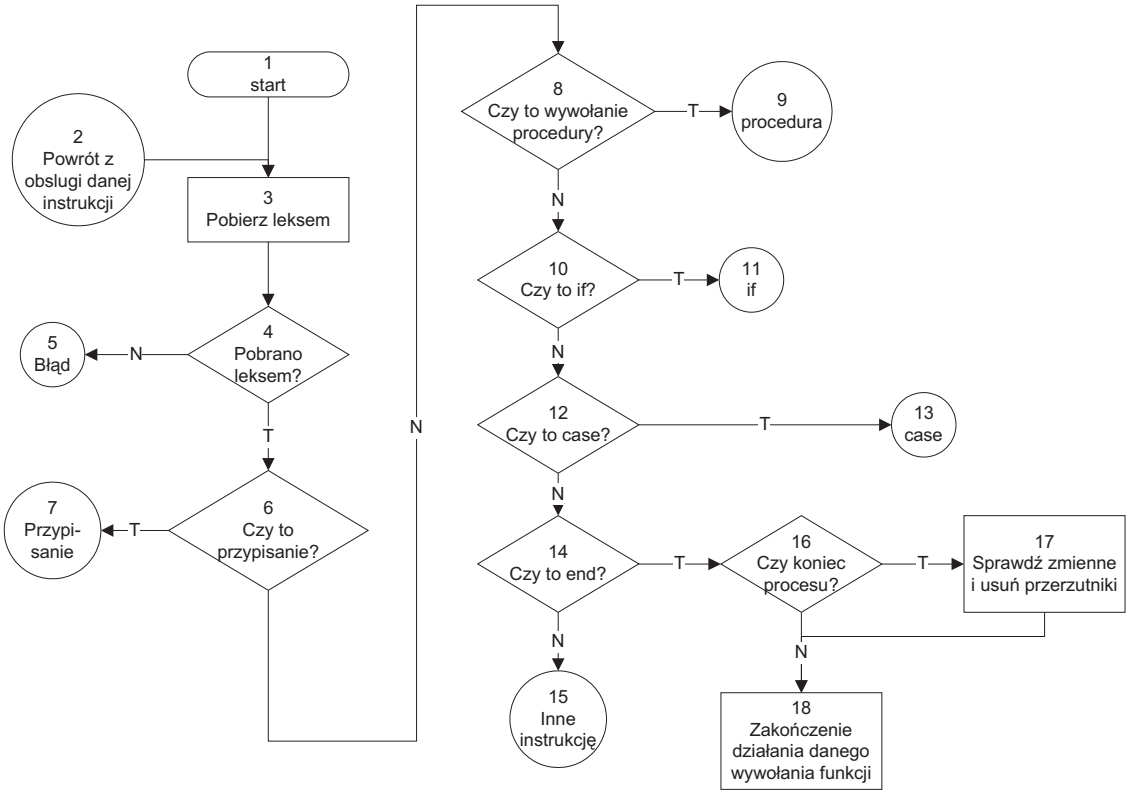
Kompilacje kodu instrukcji *process* języka VHDL można podzielić na dwa zasadnicze etapy:
— eliminacji pętli *for*,
— generacji równań.

Przed przystąpieniem do właściwego zadania, jakim jest generacji równań, należy usunąć z kodu programu instrukcję pętli *for*, bowiem nie można jej w prosty sposób zamienić na równania. Dokonuje się tego poprzez tak zwane rozwinięcie pętli, czyli wypisanie wszystkich instrukcji znajdujących się wewnątrz tyle razy ile wynosi liczba iteracji. Zostało to przedstawione szczegółowo w rozdziale 3.12.

Na rysunku 3.1 pokazany został główny algorytm generacji równań. Danymi wejściowymi są leksemy kodujące poprawne zdania języka VHDL oraz skojarzona z nimi informacja semantyczna. Należy jednak zaznaczyć, że na tym etapie analiza semantyczna, nie jest jeszcze pełna, gdyż kilka jej elementów realizują prezentowane algorytmy. Spowodowane jest to unikalnymi cechami języka VHDL, które powodują, że metody tworzenia kompilatorów znane z konwencjonalnych języków programowania nie dają się w pełni zaadaptować.

Algorytm pobiera kolejno leksemy i na podstawie tworzonych przez nie zdań identyfikuje kolejne konstrukcje języka VHDL(krok 3 i 4). Należy jednak przypomnieć, że nie dokonuje się w tym miejscu analizy syntaktycznej, ponieważ miała ona miejsce wcześniej. Po zidentyfikowaniu rodzaju instrukcji wywoływany jest właściwy algorytm cząstkowy przeznaczony do jej obsługi. Algorytm główny działa w sposób rekurencyjny. Z chwilą natrafienia na zagnieżdżenie, czyli pewien wyodrębniony zakres widoczności wywołuje sam siebie. Zagnieżdżenie stanowią:
— gałąź *if* lub *case*,
— blok pętli *for*,
— kod znajdujący się za instrukcjami *next* i *exit*.

Pierwsze dwa przypadki nie wymagają specjalnego wyjaśnienia. Natomiast ostatni zostanie opisany w jednym z następnych rozdziałów (3.12). Z chwilą natrafienia na słowo kluczowe *end*, funkcja kończy pracę w bieżącym zagnieżdżeniu. Jeżeli jest to *end* kończący cały *process*, to konieczne jest sprawdzenie, z których zmiennych



Rysunek 3.1. Algorytm przekładu ciała procesu. Źródło: opracowanie własne.

odczyt wartości odbywa się przed zapisaniem do nich jakiejkolwiek informacji. Czynność ta pozwala na dokonanie optymalizacji polegającej na usunięciu zbędnej logiki sekwencyjnej - przerzutników (16 i 17). Następnie wygenerowane równania zostają zapisane do pliku. Podczas tego zapisu, w przypadku, gdy dla danego bitu otrzymaliśmy logikę sekwencyjną tworzony jest przerzutnik typu zatrzask (ang. *latch*). Blok 15, to wszystkie te instrukcje, które nie stanowią przedmiotu niniejszej pracy, a więc:

— *wait*,
— *while loop*,
— *loop*,
— wywołanie procedury (częściowo),
— wywołanie funkcji.

Pierwsze trzy instrukcje, reprezentują logikę sekwencyjną, co może powodować konieczność wygenerowania centralnej jednostki sterującej będącej automatem skończonym. Problem ten stanowi jedno z trudniejszych zagadnień projektowanego kompilatora [102].

Wywołanie procedury jest procesem złożonym i wymaga współdziałania kilku modułów kompilatora. Część zagadnień mieści się w temacie niniejszej pracy i zostaną one omówione. Chodzi tu o sytuację, kiedy procedura zawiera argumenty typu *out* lub *inout*. W takim przypadku konieczne jest określenie czy wewnątrz procedury nie nastąpiło przypisanie do takiego argumentu, bo to jest równoznaczne z wystąpieniem instrukcji przypisania wewnątrz procesu.

Funkcje występują tylko w wyrażeniach, a więc nie mieszczą się w temacie pracy.

3.6. Generacja równań dla przypisań

Jest to najważniejsza instrukcja z punktu widzenia generacji równań boolowskich, gdyż to ona powoduje utworzenie nowych równań. Jeżeli wewnątrz bloku *process* nie znajdzie się ani jedno przypisanie, to wynikiem kompilacji będzie pusty plik. Mimo pozornej prostoty jest to instrukcja która potrafi sprawić spore kłopoty podczas kompilacji. Przekład sprowadza się do wygenerowania równań boolowskich dla wyrażenia stanowiącego prawą stronę przypisania, oraz zidentyfikowaniu celu przypisania. Istnieją jednak różnice w zależności od tego czy analizujemy przypisanie do sygnału, czy też do zmiennej. Podstawowe problemy generacji równań boolowskich dla instrukcji przypisania:

— identyfikacja celów przypisania - utrudniana przez mnogość dostępnych typów danych, oraz możliwość stosowania mechanizmów typu agregacja czy wycinek,
— występowanie dwóch różnych bytów mogących stanowić cel przypisania (sygnał, zmienna), wymagających nieco odmiennej obsługi.

3.6.1. Identyfikacja celów przypisania

W języku VHDL lewa strona przypisania (cel) może mieć bardzo skomplikowaną postać (rozdział 2.5.3). Mechanizmy typu agregacja czy wycinek utrudniają analizę instrukcji i poprawne określenie które bity są tak naprawdę zmieniane. Do tego należy dodać jeszcze cale bogactwo typów dopuszczalnych w tym języku i obraz problemu będzie pełny. O ile tablice jednowymiarowe i proste rekordy nie komplikują kompilacji tak bardzo, to należy pamiętać o tym, że dopuszczalne są tablice których elementami są rekordy lub inne tablice. Proces identyfikacji celu przypisania sprowadza się do dwóch zasadniczych czynności:

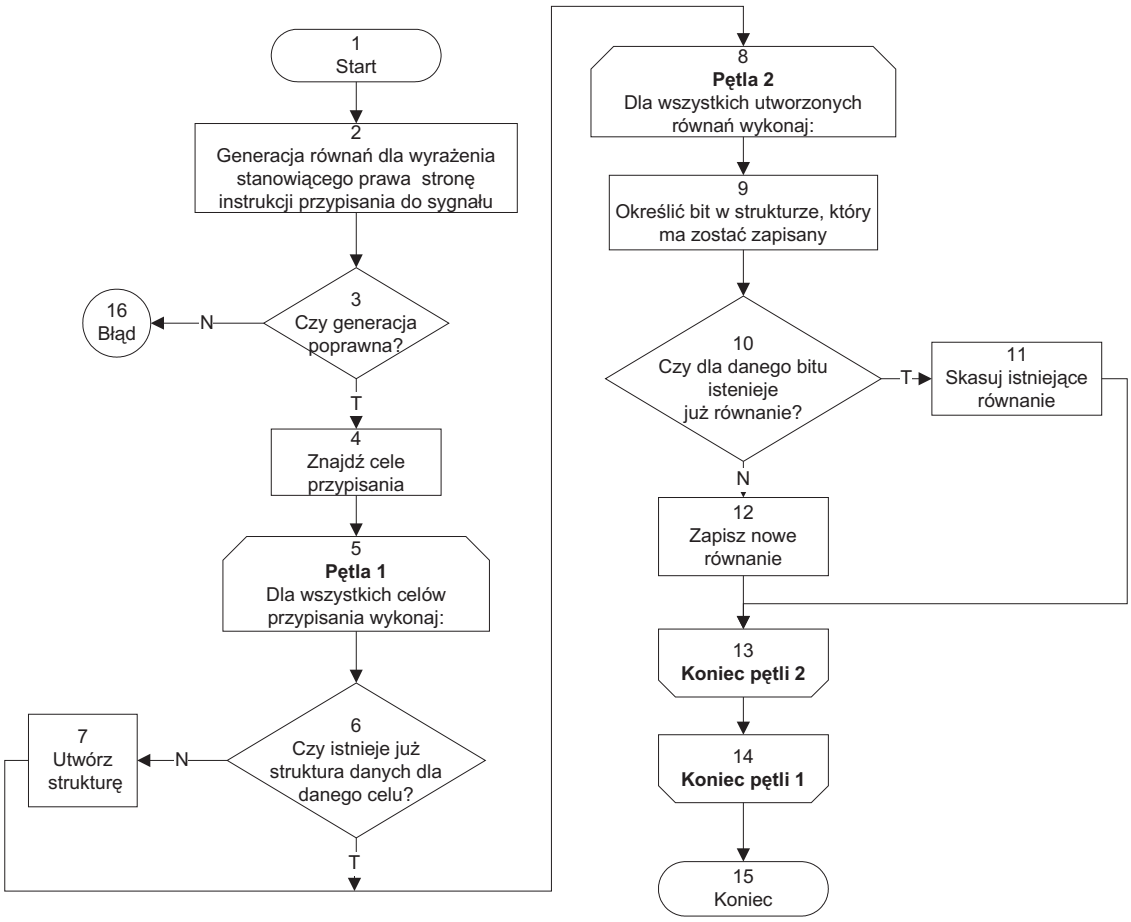
— ustalić listę sygnałów lub zmiennych, które występują do lewej stronie przypisania, przypisanie może dotyczyć kilku sygnałów lub zmiennych jednocześnie (agregacja),
— dla każdej pozycji z powyższej listy określić, które bity mają zostać zmienione, tutaj problemem są tablice i rekordy, oraz możliwość występowania wycinków.

3.6.2. Przypisanie do sygnału

Zgodnie ze specyfiką języka VHDL, tylko ostatnie, znajdujące się wewnątrz procesu przypisanie do sygnału jest wiążące. Znaczy to, że jeżeli będzie kilka kolejnych instrukcji przypisania dla tego samego celu przypisania, to tylko ostatnia z nich będzie powodować faktyczną zmianę wartości sygnału. Wszystkie poprzedzające zmiany zostaną zignorowane. Dodatkowo, nowa wartość zacznie obowiązywać dopiero po zakończeniu działania całego procesu.

Algorytm postępowania z instrukcją przypisania do sygnału jest następujący (rysunek 3.2):

1. Utworzenie równań boolowskich (2-3). Pierwszą czynnością jest wygenerowanie równań dla wyrażenia znajdującego się po prawej stronie znaku przypisania. Oczywiście sprawdzane jest czy równania wygenerowane zostały poprawnie.
2. Zidentyfikowanie celów przypisania (4). Należy znaleźć nazwy sygnałów, które występują po lewej stronie przypisania i stanowią cel przypisania.
3. Dla każdego sygnału stanowiącego cel przypisania odszukać odpowiadającą mu strukturę danych, a jeżeli jeszcze nie istnieje to utworzyć ją (5-7).



Rysunek 3.2. Algorytm generacji równań boolowskich dla instrukcji przypisania do sygnału. Źródło: opracowanie własne.

4. Dla każdego sygnału zidentyfikować bity, które mają zostać przypisane (8-14). Jeżeli dla danego bitu istnieje już równanie, to należy je usunąć i zastąpić nowym.

Przykład 3.1 pokazuje wynik działania algorytmu generowania równań dla przypisania do sygnału. Widać, że mimo iż sygnał z zmieniany jest trzy razy, to tylko ostatnia instrukcja przypisania powoduje faktyczną zmianę jego wartości.

```
entity test is
    port (
                a,b,c    : in BIT_vector(0 to 1);
                z: out BIT_vector(0 to 1)
    );
end test;

architecture arch of test is
begin
        process
        begin
            z<=a;
            z<=b;
            z<=c;
        end process;
end arch;
-- Otrzymane równania
z(0)=(c(0));
z(1)=(c(1));
```

Przykład 3.1. Generacja równań boolowskich dla przypisania do sygnału.

3.6.3. Przypisanie do zmiennej

Zmienne reprezentują pamięć, czyli miejsce do tymczasowego przechowywania informacji. Przypisanie do zmiennej, w odróżnieniu od tego do sygnału działa natychmiast. Powoduje to, że zmienna może mieć wielokrotnie zmienianą wartość podczas jednej aktywacji procesu. Problem ten utrudnia generację równań. Rozwiązaniem jest tworzenie nowego miejsca w pamięci za każdym razem, gdy dana zmienna jest przypisywana. W ten sposób powstaje historia zmian wartości danej zmiennej. W proponowanym algorytmie dokonuje się to za pomocą zmiany nazw. Każde przypisanie z wyjątkiem ostatniego, powoduje utworzenie zmiennej o nowej nazwie, powiązanej semantycznie z tą która jest celem przypisania.

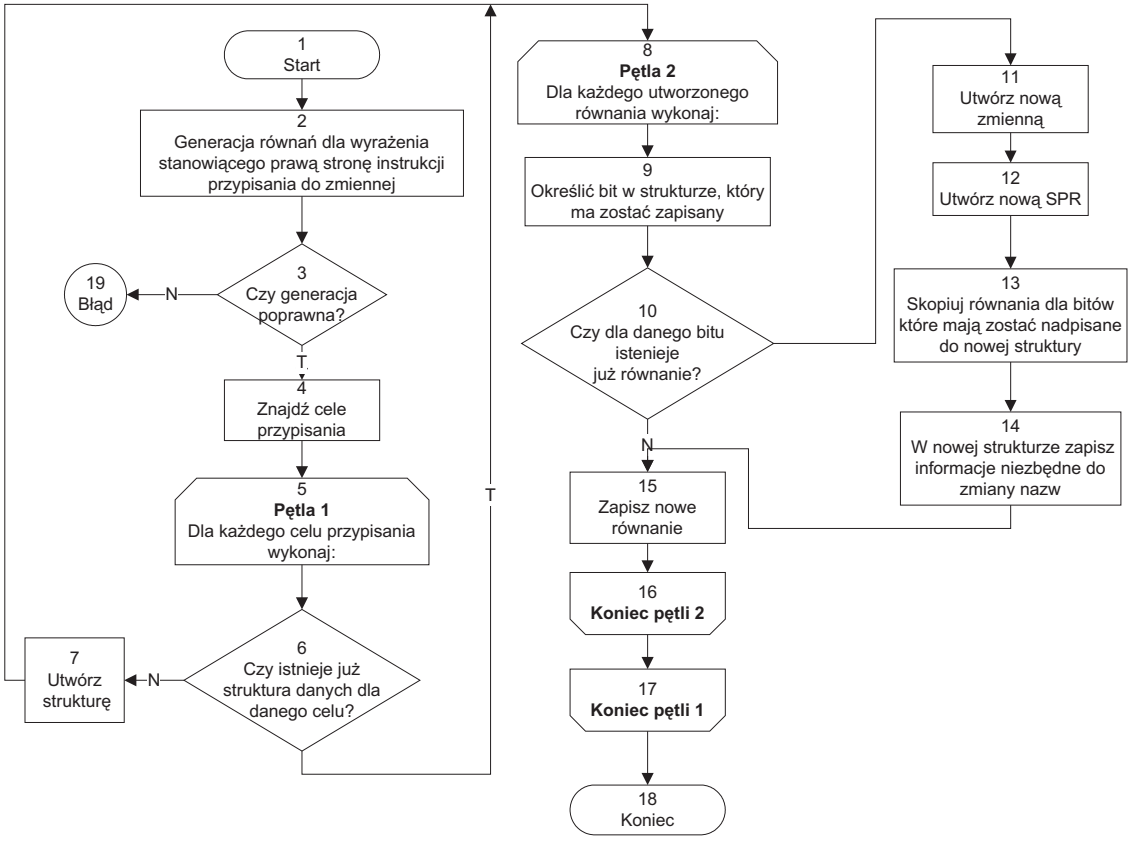
Algorytm generacji równań dla zmiennych wygląda następująco (rysunek 3.3):

1. Generacja równań boolowskich (2-3). Procedura tworzenia równania dla przypisania do zmiennej jest odmienna od procedury generacji równania dla przypisania do sygnału.
2. Identyfikowanie celów przypisania (4).
3. Dla każdej zmiennej stanowiącej cel przypisania odszukać odpowiadającą jej SPR, a jeżeli jeszcze nie istnieje to utworzyć ją (5-7).
4. Dla każdego celu przypisania zidentyfikować bity, które mają zostać przypisane (9). Jeżeli dla danego bitu istnieje już równanie (10) to:
   — utworzyć nową SPR i nową zmienną (11-12),
   — w nowej SPR zapisać te z bitów dotychczasowej struktury, dla których wygenerowane zostały równania w obecnym przypisaniu (13),
   — zapisać informację niezbędną do zmiany nazwy zmiennej w nowej strukturze SPR (14).
5. Zapisać nowe równanie (15). Dla wszystkich bitów ustawić to pole w strukturze danych, które odpowiada za wskazanie czy dany bit był już zapisany czy nie.

```
entity test is
    port (
        a,b,c: in bit;
        z,z1: out bit
    );
end test;

architecture test5 of test is

begin
        process
        variable tmp: bit;
        begin
                tmp:=c or a;
                z<=tmp and b;
                tmp:=a;
                z1<=tmp and b;
        end process;
end test5;

-- otrzymane równania
tmp_1_0=(c|a);
z=(tmp_1_0&b);
tmp=(a);
z1=(tmp&b);
```

Przykład 3.2. Generacja równań dla przypisania do zmiennej.

Przykład kodu źródłowego 3.2 obrazuje sposób generowania równań dla przypisania do zmiennej. Zmienna *tmp* jest dwukrotnie zapisywana. Po pierwszej zmianie wartości tworzona jest nowa zmienna (*tmp*_1_0), która jest następnie wykorzy-



Rysunek 3.3. Algorytm generacji równań boolowskich dla instrukcji przypisania do zmiennej. Źródło: opracowanie własne.

stywana dalej. Ostatnie przypisanie nie powoduje już tworzenia nowego miejsca w pamięci.

3.7. Wywołanie procedury

Poprawna generacja równań boolowskich wewnątrz instrukcji *process* wymaga uwzględnienia wszystkich równań cząstkowych. Równanie cząstkowe powstaje w wyniku analizy instrukcji przypisania, *if*, oraz *case*. Wywołanie procedury także może zaowocować otrzymaniem równań cząstkowych. W języku VHDL, jeżeli definiowana jest procedura, to dopuszczalne jest aby mogła zmieniać wartość przekazanych do niej argumentów. Aby było to wykonalne parametr formalny musi być zdefiniowany jak *out* lub *inout*. Zwrócenie wartości w ten sposób w obrębie procesu jest równoznaczne z wystąpieniem instrukcji przypisania. Można więc określić taką sytuacje mianem obecności przypisania ukrytego.

Wywołanie procedury jest procesem złożonym, wymagającym współdziałania różnych modułów kompilatora[69]. Z punktu widzenia niniejszej pracy, istotny jest wpływ jaki może ono mieć na wartości sygnałów i zmiennych wewnątrz bloku *process*.

Na przykładzie 3.3 pokazane jest wywołanie procedury, która zwraca informacje za pomocą argumentu typu *out*. Wartość zwrócona przypisana jest do sygnału *z* użytego jako parametr aktualny.

```
entity test is
    port (
        a,b: in bit_vector(1 to 2) ;
        z: out bit_vector(1 to 2)
    );
end test;

architecture test6 of test is

procedure proc (A,B: in bit_vector; C: out bit_vector) is
   begin
         c<=a or b;

   end procedure proc;
begin
        process
        begin
                proc(a,b,z);
        end process;
end test6;

-- Otrzymane równania
z(1)=(((a(1)|b(1))));
z(2)=(((a(2)|b(2))));
```

Przykład 3.3. Wywołanie procedury mającej parametry typu *out*.

3.8. Generacja równań boolowskich dla instrukcji *if* i *case*

Obie powyższe instrukcje umożliwiają warunkowe wykonanie bloków kodu. Mimo różnic w działaniu, proces generacji równań boolowskich przebiega w przypadku obu z nich bardzo podobnie. Odmienne postępowanie dotyczy:
— zasady sprawdzania warunków wejściowych,
— postaci warunków wejściowych.

Zadaniem niniejszego rozdziału, jest zaprezentowanie zasad kompilacji instrukcji *if* oraz *case* do postaci równań boolowskich. Zanim jednak do tego dojdzie konieczne jest dokładne zrozumienie sposobu funkcjonowania obu instrukcji z uwzględnieniem ograniczeń narzuconych przez syntezę logiczna. W końcowej części rozdziału zamieszczony został kompletny algorytm generacji równań boolowskich.

3.8.1. Definicje

Na początek przedstawione zostanie kilka pojęć używanych w dalszej części bieżącego rozdziału.

Warunek wejściowy lub aktywacji. *Warunkiem wejściowym* gałęzi, nazywane jest równanie boolowskie, którego wartość decyduje o tym czy dana gałąź zostanie aktywowana czy też nie.

Elementarny warunek wejściowy. *Elementarny warunek wejściowy* stanowi część *warunku wejściowego*. Jest to wyrażenie znajdujące się między słowami kluczowymi *if* (*elsif*). Mianem tym określa się także wygenerowane na podstawie owego wyrażenia równanie boolowskie.

Wyrażenie sterujące. *Wyrażenie sterujące* występuje tylko w przypadku instrukcji *case*. Jest to wyrażenie znajdujące się między słowami kluczowymi *case* oraz *of*.

Wartość wejściowa. Jest to wartość jaką musi przyjąć *wyrażenie sterujące*, aby mogła nastąpić aktywacja danej gałęzi instrukcji *case*. Jedna gałąź może mieć wiele takich wartości, podanych jawnie, lub za pomocą zakresu (*range*).

3.8.2. Sposób działania instrukcji

Działanie obu instrukcji zostało wyjaśnione w rozdziale 2.5.3. Podsumowując, najważniejszą różnicą między nimi jest sposób sprawdzania warunków wejściowych. W przypadku instrukcji *case* każda gałąź traktowana jest niezależnie. Oznacza to, że może nastąpić jednoczesna aktywacja więcej niż jednej z nich. Nie jest także wymaga rozłączność warunków wejścia gałęzi. Takie zachowanie nie jest niestety akceptowane, gdy dany kod VHDL ma być poddany syntezie. Natomiast instrukcja *if* pozwala na aktywowanie tylko jednej z gałęzi. Po tym zdarzeniu, sterowanie przechodzi do kodu za blokiem *if*. Może wystąpić sytuacja, w której warunki wejściowe gałęzi się pokrywają, ale nie ma to znaczenia, ponieważ zawsze wykona się kod tylko jednej gałęzi.

Obie instrukcje mają też specjalną gałąź, która uaktywniana jest wtedy, gdy warunek wejściowy żadnej innej nie jest spełniony. W przypadku *if* nazywa się ona *else*, natomiast w instrukcji *case* nosi nazwę *others*.

Dowolnej postaci instrukcję *case* można zastąpić blokiem instrukcji *if*. Odwrotna transformacja zależność nie jest prawdziwa. Wynika z tego jasno, że *if* jest bardziej uniwersalny.

3.8.3. Ogólna postać równania wynikowego dla instrukcji *if* oraz *case*

Ogólna postać równania wynikowego dla sygnału czy zmiennej jest taka sama bez względu na to, której instrukcji (*if* lub *case*) dotyczy (wzór 3.1). Jego podstawowymi elementami są równania cząstkowe, które tworzone są dla przypisań wewnątrz gałęzi obu instrukcji. Wynika więc jasno, że jeżeli wewnątrz *if*(*case* nie będzie ani jednej instrukcji przypisania, nie zostaną utworzone żadne równania wynikowe.

$$X = \sum_{i=1}^{n} Wwe_i \cdot X_i \tag{3.1}$$

gdzie:
— i - numer gałęzi,
— n - ilość gałęzi,
— X - lewa strona przypisania (cel przypisania),
— X_i - równanie boolowskie dla prawej strony przypisania w i-tej gałęzi,
— Wwe_i - równanie boolowskie warunku wejściowego (aktywacji) dla i-tej gałęzi.

Sens równania 3.1 jest następujący:
— aktywacją każdej z gałęzi zarządza inny warunek wejściowy,
— w każdej z gałęzi znajduje się przypisanie do rozpatrywanego bitu danego sygnału lub zmiennej, na podstawie którego tworzone jest równanie boolowskie,
— warunki wejściowe różnych gałęzi muszą być wzajemnie rozdzielne, to znaczy, że nie powinno być sytuacji w której możliwa jest aktywacja więcej niż jednej gałęzi,

— spełnienie danego warunku wejściowego powoduje, że pozostałe warunki mają wartość logiczną false, w związku z tym o wyniku równania boolowskiego wyrażonego wzorem 3.1 zadecyduje równanie otrzymane na podstawie przypisania z aktywnej gałęzi.

Rozdzielność warunków wejściowych jest jednym z ograniczeń, jakie musi spełnić instrukcja *case*, aby mogła być poddana syntezie. Pozostałe ograniczenia przedstawione są w rozdziale 3.8.6. W przypadku *if* sytuacja wygląda nieco inaczej. Jednoznaczność aktywacji zapewniana jest przez odpowiednie skonstruowanie równań warunków wejściowych (rozdział 3.8.5).

3.8.4. Generacja warunków wejściowych

Warunki wejściowe decydują o tym, które przypisanie (z której gałęzi) stanie się aktywne i nada wartość całemu równaniu boolowskiemu. Jest to punkt wskazujący na różnice między obydwiema analizowanymi instrukcjami. Poniżej przedstawione są formuły tworzenia równań boolowskich warunków wejścia, dla *if* oraz *case*.

3.8.5. Postać równania warunku wejściowego dla instrukcji *if*

Postać równania warunku wejściowego dla $i-tej$ gałęzi instrukcji *if* przedstawia wzór 3.2:

$$We_i = (\prod_{j=1}^{j<i} !W_j) \cdot W_i \tag{3.2}$$

gdzie:
— i - numer gałęzi, dla której tworzony jest warunek,
— j - indeks gałęzi poprzedzających daną gałąź,
— $!W_j$ - zanegowane równanie boolowskie elementarnego warunku wejścia do gałęzi if, która poprzedza daną gałąź (w przypadku gałęzi pierwszej nie występuje),
— W_i - równanie boolowskie elementarnego warunku wejścia do bieżącej gałęzi (w przypadku else nie występuje).

Wzór 3.2 stanowi dosłowną interpretację sposobu działania instrukcji *if* opisanego w rozdziale 2.5.3. Dana gałąź o indeksie i, będzie wykonana wtedy i tylko wtedy, gdy żadna z gałęzi ją poprzedzających nie została aktywowana, a jej elementarny warunek wejściowy jest spełniony. Gałęzie sprawdzane są kolejno i tylko jedna z nich może być aktywna. Taki sposób generacji warunku wejściowego zapobiega jednoczesnej aktywacji więcej niż jednej gałęzi. W przypadku, kiedy dwie gałęzie miały by taki sam elementarny warunek wejściowy (najprawdopodobniej w skutek błędu programisty), to i tak zostanie aktywowana tylko ta, która jest pierwsza w kolejności.

3.8.6. Postać równania warunku wejściowego dla instrukcji *case*

Zgodnie z tym co zostało napisane wcześniej, *case* pozwala na niezależną analizę warunków wejściowych wszystkich swoich gałęzi. W efekcie może nastąpić aktywacja więcej niż jednej. Nie jest możliwe utworzenie reprezentacji sprzętowej dla takiej konstrukcji. Konieczne stało się wprowadzenie pewnych restrykcji dotyczących składni i semantyki, natomiast sam sposób działania *case* pozostał bez zmian.

Aby dana instrukcja *case* mogła zostać poddana syntezie[91], musi spełniać następujące warunki:

— wyrażenie sterujące może być typu integer, typu wyliczeniowego, lub jednowymiarową tablicą elementów typu wyliczeniowego,
— wartości wejściowe muszą być statyczne, tzn. ich wartość musi być określona na etapie kompilacji,
— nie może zaistnieć sytuacja, w której dana wartość wejściowa występuje w więcej niż jednej gałęzi,
— rozmiar każdej wartości wejściowej (ilość bitów) musi być taki sam jak wyrażenia sterującego,
— każda z możliwych wartości wyrażenia sterującego, musi być uwzględniona w wartościach wejściowych, lub *case* musi zawierać gałąź *others*.

Tworzenie równania warunku wejściowego dla gałęzi instrukcji *case* to proces dwuetapowy. Pierwszym krokiem jest wygenerowanie równań wartości wejściowych zgodnie ze wzorem 3.3:

$$W_k = \prod_{l=0}^{p-1} \left\{ \begin{array}{rcl} l < n \wedge W_{k,l} = 1 & \Rightarrow & S(l) \\ l < n \wedge W_{k,l} = 0 & \Rightarrow & !S(l) \\ l \geq n \wedge l < p & \Rightarrow & !S(l) \end{array} \right\} \tag{3.3}$$

gdzie:
— W_k - kolejna wartość wejściowa,
— S - nazwa wyrażenia sterującego (nazwa),
— $S(l)$ - kolejny bit wyrażenia sterującego,
— W_l - kolejny bit oryginalnego wyrażenia wejściowego,
— n - ilość bitów oryginalnego wyrażenia wejściowego,
— p - ilość bitów wyrażenia sterującego,
— l - indeks kolejnego bitu.

Każda gałąź może mieć wiele wartości wejściowych. Zgodnie z ograniczeniami instrukcji *case*, wartości wejściowe muszą mieć taki sam rozmiar w bitach jak wyrażenie sterujące. Jeżeli oryginalna wartość wejściowa ma mniejszą liczbę bitów, to musi zostać rozszerzona do właściwej wielkości poprzez powielenie najbardziej znaczącego bitu. Należy pamiętać, że wartości wejściowe mogą być podane z użyciem wyrażenia zakresu *range*.

Mając wygenerowane wartości wejściowe pozostaje utworzyć równania warunków wejściowych do poszczególnych gałęzi. Postać równania boolowskiego dla warunku wejściowego $i-tej$ gałęzi *case* przedstawia równanie 3.4:

$$Wwe_i = \sum_{k=1}^{m} Ww_k \tag{3.4}$$

gdzie:
— W_k - równanie kolejnej wartości wejściowej dla danej gałęzi,
— k - indeks kolejnej wartości,
— m - liczba wszystkich wartości wejściowych dla danej gałęzi,
— i - numer gałęzi.

Z tworzeniem równań warunków wejściowych związane są dwie dodatkowe operacje do wykonania:
— konieczność sprawdzenie czy dana wartość wejściowa nie wystąpiła już wcześniej (w innej gałęzi),
— w przypadku gdy dana instrukcja *case* nie posiada gałęzi *others*, należy sprawdzić czy zbiór wartości wejściowych ze wszystkich gałęzi jest równy zbiorowi dopuszczalnych wartości wyrażenia sterującego.

Aby możliwe było wykonanie powyższych operacji konieczne jest zapamiętywanie wygenerowanych wartości wejściowych, celem ich późniejszego przeszukiwania. Problem jaki się tutaj pojawia wiąże się z ilością koniecznych do zapamiętania wartości. Zakładając, że wyrażenie sterujące jest typu *Integer* to liczba wartości jakie mogą wystąpić wynosi: $2^{32} - 1$. W przypadku typów wyliczeniowych liczba wartości może być znacznie większa.

Alternatywna postać równania warunku wejściowego dla gałęzi *others*

Mimo iż równanie 3.4 jest prawidłowe dla wszystkich przypadków, to dla gałęzi *others* w pewnych sytuacjach łatwiejsza do zastosowania jest inna, prezentowana przez wzór 3.5, postać warunku wejściowego. Wymowa wzoru jest następująca: gałąź *others* jest aktywowana tymi wartościami wejściowymi, które nie występują w pozostałych gałęziach.

$$Wwe_{others} =! \left(\sum_{i=1}^{n-1} Wwe_i \right) \tag{3.5}$$

gdzie:
— n - liczba gałęzi,
— Wwe_i - równanie boolowskie warunku wejściowego i-tej gałęzi, case.

Takie równanie jest łatwiejsze do wygenerowania, gdyż korzystamy z wcześniej obliczonych wartości. W przypadku stosowania standardowego wzoru (3.4) konieczne jest znalezienie tych wszystkich wartości, które nie zostały uwzględnione w pozostałych gałęziach, co jak zostało napisane wcześniej stanowi problem.

3.8.7. Przykłady

```
entity test is
    port (
        a,b,c: in BIT;
        s: integer range 0 to 3;
        z: out BIT
    );
end test;

architecture test5 of test is
begin
        process
        begin
                if s = 2 then
                        z<=a;
                elsif s=3 then
                        z<=b;
                else
                        z<=c;
                end if;
        end process;
end test5;

-- Otrzymane równanie

z= (s(1)&!s(0))&(a)                                    --if
 | (!(s(1)&!s(0))&(s(1)&s(0)))&(b)                     --elsif
 | (!(s(1)&!s(0))&(!(s(1)&s(0))))&(c);                 --else
```

Przykład 3.4. Generacja równań dla instrukcji *if*.

Przykład 3.4 przedstawia przykład instrukcji *if* wraz z otrzymanymi równaniami. Kolejne linie równania wynikowego odnoszą się do kolejnych gałęzi instrukcji.

Tekstem wytłuszczonym zaznaczone te części równań, które reprezentują warunki wejściowe.

Źródło 3.5 prezentuje instrukcję *case*, wraz z równaniami otrzymanymi podczas jej kompilacji. Widać wartości wejściowe otrzymane zgodnie ze wzorem 3.3. Równanie dla gałęzi *others* zostało utworzone z wykorzystaniem formuły 3.5. Tekstem wytłuszczonym zaznaczono równania warunków wejściowych.

```
entity test is
    port (
                a,b,c,d: in BIT_vector(0 to 1);
                 s1 : in integer range 0 to 6;
                 z: out BIT_vector(0 to 1)
    );
end test;

architecture arch of test is
begin
        process (a, b)
        begin
                case s1 is
                        when 2 => z<=a;
                        when 1 => z<=b;
                        when 6 => z<=c;
                        when others => z<=d;
                end case;
        end process;
end arch;

-- Otrzymane równania

z(0)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(0))))        --I
        |(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(0)))))    --II
        |(((((!s1(0))&(s1(1)))&(s1(2))))&((c(0)))))     --III
        |((!(((((((!s1(0))&(s1(1)))&(!s1(2))))        --others
        |((((s1(0))&(!s1(1)))&(!s1(2)))))
        |((((!s1(0))&(s1(1)))&(s1(2)))))&((d(0))));
z(1)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(1))))        --I
        |(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(1)))))    --II
        |(((((!s1(0))&(s1(1)))&(s1(2))))&((c(1)))))     --III
        |((!(((((((!s1(0))&(s1(1)))&(!s1(2))))        --others
        |((((s1(0))&(!s1(1)))&(!s1(2)))))
        |((((!s1(0))&(s1(1)))&(s1(2)))))&((d(1))));
```

Przykład 3.5. Generacja równań dla instrukcji *case*.

3.9. Generacja równań w przypadku braku instrukcji przypisania dla danego sygnału lub zmiennej, w jednej lub kilku gałęziach instrukcji *if* lub *case*

Wzór 3.1 sprawdza się tylko w sytuacji, kiedy w każdej z gałęzi *if* lub *case* występuje przypisanie dla danego bitu sygnału lub zmiennej. Dodatkowo, jeśli rozpatrujemy instrukcje *if*, to musi ona posiadać blok *else*. Niestety, jest to sytuacja idealna, a z taką nie zawsze można mieć do czynienia. Istotą problemu jest to, że jeżeli nie ma przypisań we wszystkich gałęziach, to wartość bitu za blokiem *if* (*case*) będzie nieokreślona. Taka sytuacja jest kłopotliwa, bo wymaga zapamiętania wartości bitu między kolejnymi wywołaniami pętli procesu. Powoduje to konieczność zastosowania przerzutnika typu zatrzask, czyli logiki sekwencyjnej. Można by oczywiście rozwiązać ten problem przez nałożenie ograniczeń na projektanta, ale z oczywistych względów nie jest to najlepsza droga. Konieczne jest, zatem takie przystosowanie

lub rozwinięcie wzoru 3.1, aby obejmował on wszystkie przypadki. Problem braku przypisania dla danego bitu wewnątrz wszystkich gałęzi *if* lub *case* można podzielić na dwa przypadki:

— wartość danego bitu została określona przed blokiem *if* (*case*), to znaczy, że nastąpiło przypisanie wartości do tego bitu, w taki sposób który nie powodował potrzeby tworzenia logiki sekwencyjnej,
— wartość danego bitu nie została określona przed blokiem instrukcji *if* (*case*).

3.9.1. Wartość danego bitu została określona przed blokiem *if* (*case*)

Ten scenariusz pozwala uniknąć konieczności stosowania logiki sekwencyjnej. Wprawdzie nie wszystkie gałęzie zawierają przypisanie do danego bitu, ale ponieważ przed blokiem *if*(*case*) miał on wartość ustaloną, to można ja wykorzystać. Przebieg postępowania jest następujący:

— określić które gałęzie nie zawierały przypisań do danego bitu, a które zawierały,
— dla wszystkich gałęzi w których nie nastąpiło przypisanie do danego bitu, do równania wynikowego podstawić wartość bitu nadaną mu przed blokiem instrukcji *if*(*case*).

Formuła matematyczna tak tworzonego równania wyrażona jest wzorem 3.6:

$$X = \sum_{i=1}^{n} Wwe_i \cdot X_i \cdot Z_i + \sum_{i=1}^{n} (Wwe_i \cdot !Z_i) \cdot Y_i \tag{3.6}$$

gdzie:

— X, Wwe_i, X_i, i, n - tak jak równaniu ogólnym (3.1),
— Z_i - zmienna przyjmująca wartość jeden, gdy w $i-tej$ gałęzi występuje przypisanie do X, oraz zero w przeciwnym wypadku,
— Y_i - równanie dla prawej strony przypisania, występującego przed instrukcją warunkową.

```
entity test is
    port (
        a,b,c: in BIT;
        s: integer range 0 to 3;
        z: out BIT
    );
end test;

architecture test5 of test is
begin
        process
        begin
                z<=c;
                if s = 2 then
                        z<=a;
                elsif s=3 then
                        z<=b;
                end if;
        end process;
end test5;

-- Otrzymane równanie

z=((((s(1)&!s(0))))&((a)))                          --if
|((((!((s(1)&!s(0)))))&(((s(1)&s(0)))))&((b))))     --elsif
|!((((s(1)&!s(0))))                                 --pozostałe
|(((!((s(1)&!s(0)))))&(((s(1)&s(0)))))&((c));
```

Przykład 3.6. Generacja równań boolowskich dla instrukcji *if* - brak przypisań we wszystkich gałęziach ale występuje przypisanie przed blokiem.

Unikanie zbędnych przerzutników jest jednym z podstawowych działań optymalizacyjnych. Wykorzystanie uprzednio nadanej bitowi wartości to właśnie umożliwia. Nie ma potrzeby zapamiętywania wartości danego sygnału czy zmiennej między kolejnymi wywołaniami instrukcji procesu.

Źródło 3.6 przedstawia sytuację, gdy w bloku *if* brakuje gałęzi *else*, co może wymagać utworzenia równań logiki sekwencyjnej. Ponieważ jednak wartość sygnału z została określona przed instrukcją *if* nie trzeba tego robić w tym przypadku.

```
entity test is
        port (
                a,b,c,d: in BIT_vector(0 to 1);
                s1 : in integer range 0 to 6;
                z: out BIT_vector(0 to 1)
        );
end test;

architecture arch of test is
begin
        process (a, b)
        begin
                z<=d;
                case s1 is
                        when 2 => z<=a;
                        when 1 => z<=b;
                        when 6 => z<=c;
                end case;
        end process;
end arch;

-- Otrzymane równania

z(0)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(0))))        --I
|(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(0)))))    --II
|(((((!s1(0))&(s1(1)))&(s1(2))))&((c(0)))))    --III
|!((((((!s1(0))&(s1(1)))&(!s1(2))))|((((s1(0))          --pozostałe
&(!s1(1)))&(!s1(2)))))
|((((!s1(0))&(s1(1)))&(s1(2)))))&((d(0)));
z(1)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(1))))        --I
|(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(1)))))    --II
|(((((!s1(0))&(s1(1)))&(s1(2))))&((c(1)))))    --III
|!((((((!s1(0))&(s1(1)))&(!s1(2))))|((((s1(0))          --pozostałe
&(!s1(1)))&(!s1(2)))))
|((((!s1(0))&(s1(1)))&(s1(2)))))&((d(1)));
```

Przykład 3.7. Generacja równań boolowskich dla instrukcji *case* - brak przypisań we wszystkich gałęziach ale występuje przypisanie przed blokiem.

Przykład 3.7 to instrukcja *case* nie posiadająca gałęzi *others*. Dzięki przypisaniu przed blokiem *case* nie trzeba tworzyć przerzutnika dla sygnału z. Ponownie tłustym drukiem wyróżnione zostały te części równania, które reprezentują warunki wejściowe.

3.9.2. Wartość danego bitu nie została określona przed blokiem instrukcji *if* (*case*)

Brak przypisań w gałęziach *if*(*case*) dla danego bitu sygnału(zmiennej), w sytuacji, kiedy jego wartość nie została wcześniej określona oznacza, że trzeba zastosować logikę sekwencyjną. Konieczne jest zapamiętanie wartości sygnału czy zmiennej między kolejnymi wywołaniami instrukcji process. Technicznie realizowane jest to przy użyciu przerzutnika typu D wyzwalanego poziomem. Jest on potocznie nazywany *zatrzaskiem* (ang. *latch*). Dzięki temu, jeżeli chociaż raz zostanie nadana wartość sygnałowi lub zmiennej(konkretnemu bitowi), to wartość ta będzie pamiętana. Do przedstawienia przerzutnika nie wystarczy jedno równanie boolowskie, potrzebna

jest większa ich ilość. Najważniejsze są dwa z nich, przedstawione wzorami 3.7 i 3.8. Pierwsze steruje wejściem D przerzutnika, drugie wejściem sygnału zegarowego C. Pozostałe równania tworzone są na podstawie wzorca. Wzorców jest kilka rodzajów. Różnią się między sobą np. poziomem którym są wyzwalane.

$$D = \sum_{i=1}^{n} Wwe_i \cdot X_i \cdot Z_i \tag{3.7}$$

$$C = \sum_{i=1}^{n} Wwe_i \cdot Z_i \tag{3.8}$$

gdzie:
— Wwe_i, X_i, i, n - tak jak równaniu ogólnym,
— Z_i - zmienna przyjmująca wartość jeden, gdy w $i-tej$ gałęzi występuje przypisanie do X, oraz zero w przeciwnym wypadku
— D - wejście D przerzutnika,
— C - wejście zegarowe przerzutnika.

```
entity test is
    port (
        a,b: in BIT;
        s: integer range 0 to 3;
        z: out BIT
    );
end test;

architecture test5 of test is
begin
        process
        begin
                if s = 2 then
                        z<=a;
                elsif s=3 then
                        z<=b;
                end if;
        end process;
end test5;

-- Otrzymane równania

-- pragma asyn(t-1);
--{
C_tmp0=(((s(1)&!s(0))))|((((!((s(1)&!s(0)))))
        &(((s(1)&s(0))))));
D_tmp0=((((s(1)&!s(0))))&((a)))
        |(((((!((s(1)&!s(0)))))&(((s(1)&s(0)))))&((b))));
-- latch(C_high,D)
S_tmp0=D_tmp0&C_tmp0;
R_tmp0=C_tmp0&!D_tmp0;
z(t)=S_tmp0|(!R_tmp0&z(t-1));
--}
```

Przykład 3.8. Generacja równań boolowskich dla instrukcji *if* w sytuacji braku przypisania w jednej z gałęzi.

Podczas przekładu instrukcji *if* oraz *case* zawsze tworzone są równania zarówno kombinacyjne jak i sekwencyjne. Dopiero po przeanalizowaniu całego kodu VHDL znajdującego się w instrukcji process, można podjąć decyzje o tym, jaka postać równań jest właściwa. Problem dotyczy szczególnie zmiennych, bo w ich przypadku przeprowadza się optymalizację przerzutników.

Przykłady 3.8 oraz 3.9 przedstawiają odpowiednio instrukcje *if* oraz *case* w przypadku gdy konieczne jest zastosowanie logiki sekwencyjnej. Przerzutniki utworzono na podstawie wzorca (załącznik A).

```
entity test is
        port (
                a,b,c,d: in BIT_vector(0 to 1);
                s1 : in integer range 0 to 6;
                z: out BIT_vector(0 to 1)
        );
end test;

architecture arch of test is
begin
        process
        begin
                case s1 is
                        when 2 => z<=a;
                        when 1 => z<=b;
                        when 6 => z<=c;
                end case;
        end process;
end arch;

-- Otrzymane równania

-- pragma asyn(t-1);
--{
C_tmp0=(((((!s1(0))&(s1(1)))&(!s1(2))))
          |((((s1(0))&(!s1(1)))&(!s1(2)))))
          |((((!s1(0))&(s1(1)))&(s1(2))));
D_tmp0(0)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(0))))
          |(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(0)))))
          |(((((!s1(0))&(s1(1)))&(s1(2))))&((c(0))));
-- latch(C_high,D)
S_tmp0(0)=D_tmp0(0)&C_tmp0;
R_tmp0(0)=C_tmp0&!D_tmp0(0);
z(t,0)=S_tmp0(0)|(!R_tmp0(0)&z(t-1,0));
--}
-- pragma asyn(t-1);
--{
C_tmp1=(((((!s1(0))&(s1(1)))&(!s1(2))))
          |((((s1(0))&(!s1(1)))&(!s1(2)))))
          |((((!s1(0))&(s1(1)))&(s1(2))));
D_tmp1(1)=(((((((!s1(0))&(s1(1)))&(!s1(2))))&((a(1))))
          |(((((s1(0))&(!s1(1)))&(!s1(2))))&((b(1)))))
          |(((((!s1(0))&(s1(1)))&(s1(2))))&((c(1))));
-- latch(C_high,D)
S_tmp1(1)=D_tmp1(1)&C_tmp1;
R_tmp1(1)=C_tmp1&!D_tmp1(1);
z(t,1)=S_tmp1(1)|(!R_tmp1(1)&z(t-1,1));
--}
```

Przykład 3.9. Generacja równań boolowskich dla instrukcji *case* w sytuacji braku przypisania w jednej z gałęzi.

3.10. Przypadki szczególne kompilacji instrukcji *if* oraz *case*

Aby opis procesu generacji równań boolowskich dla obydwu instrukcji był kompletny należy wspomnieć o sytuacjach, które można uznać za przypadki szczególne. Wymagają one nieco innego podejścia, bądź też dodatkowych działań ze strony kompilatora. Poniżej przedstawione są takie nietypowe scenariusze.

3.10.1. Wartość warunku danej gałęzi instrukcji *if* jest statyczna

Niekiedy wartość elementarnego warunku wejściowego instrukcji *if* jest statyczna, to znaczy, że jest możliwość wyznaczenia tej wartości na etapie kompilacji danego przykładu. Dobrze jest taką sytuację wykryć i odpowiednio potraktować. Jeżeli bowiem wartość tegoż warunku wynosi „0", to dana gałąź nie zostanie nigdy aktywowana, a co za tym idzie należy ją całkowicie pominąć podczas kompilacji. Natomiast, gdy warunek ma wartość logiczną „1", to taka gałąź będzie zawsze aktywna, co oznacza, że sterowanie wejdzie tylko do tej gałęzi, pomijając wszystkie inne. Spowoduje to de facto, że wszystko to, co znajduje się wewnątrz takiej gałęzi wykona się jak by nie było w ogóle objęte warunkiem.

3.10.2. Sytuacja wyjątkowa dotycząca gałęzi *others* instrukcji *case*

Warunki wystąpienia tego scenariusza:
— kolejne wartości wejściowe ze wszystkich gałęzi, pokrywają wszystkie możliwe wartości wyrażenia sterującego,
— dany blok instrukcji case posiada gałąź others.
W takim przypadku, gałąź others nigdy nie zostanie aktywowana, można ją więc pominąć. Taka sytuacja nie jest traktowana jednak jako błąd.

3.10.3. Zasady eliminacji przerzutników dla zmiennych

W rozdziale 3.9.2 opisana została sytuacja w której dla sygnału lub zmiennej tworzony jest przerzutnik. Było to związane z koniecznością zapamiętania wartości między kolejnymi wywołaniami instrukcji procesu. Ponieważ jednak zmienne występują jedynie lokalnie wewnątrz procesu, w ich przypadku zasady które opisane zostały wcześniej nie zawsze mają zastosowanie. Istnieją trzy warunki które muszą być spełnione aby dla danej zmiennej konieczne było wygenerowanie przerzutnika[11]:
— zmiennej musi być przypisywana wartość wewnątrz instrukcji *if* lub *case*,
— musi brakować instrukcji przypisania dla danej zmiennej (konkretnego bitu) w przynajmniej jednej z gałęzi wyżej wymienionej instrukcji,
— musi istnieć konieczność zapamiętania wartości zmiennej między kolejnymi aktywowaniami instrukcji procesu.

Zapamiętanie wartości zmiennej jest konieczne wtedy, gdy jej wartość jest najpierw czytana (np. występuje po prawej stronie jakieś przypisania), dopiero później zapisywana.

Eliminacje zbędnych przerzutników można przeprowadzić dopiero po przeanalizowaniu całego bloku instrukcji procesu. Wtedy bowiem dopiero dysponujemy informacją niezbędną, aby tego dokonać. Oczywiście można zrezygnować z tego kroku, gdyż nie wpływa on na poprawność działania układu; byłoby to jednak nierozsądne z punktu widzenia optymalizacji. Jeżeli tylko pojawia się możliwość pozbycia się kłopotliwej logiki sekwencyjnej to należy ją wykorzystać.

3.11. Algorytm generacji równań boolowskich dla instrukcji *if* oraz *case*

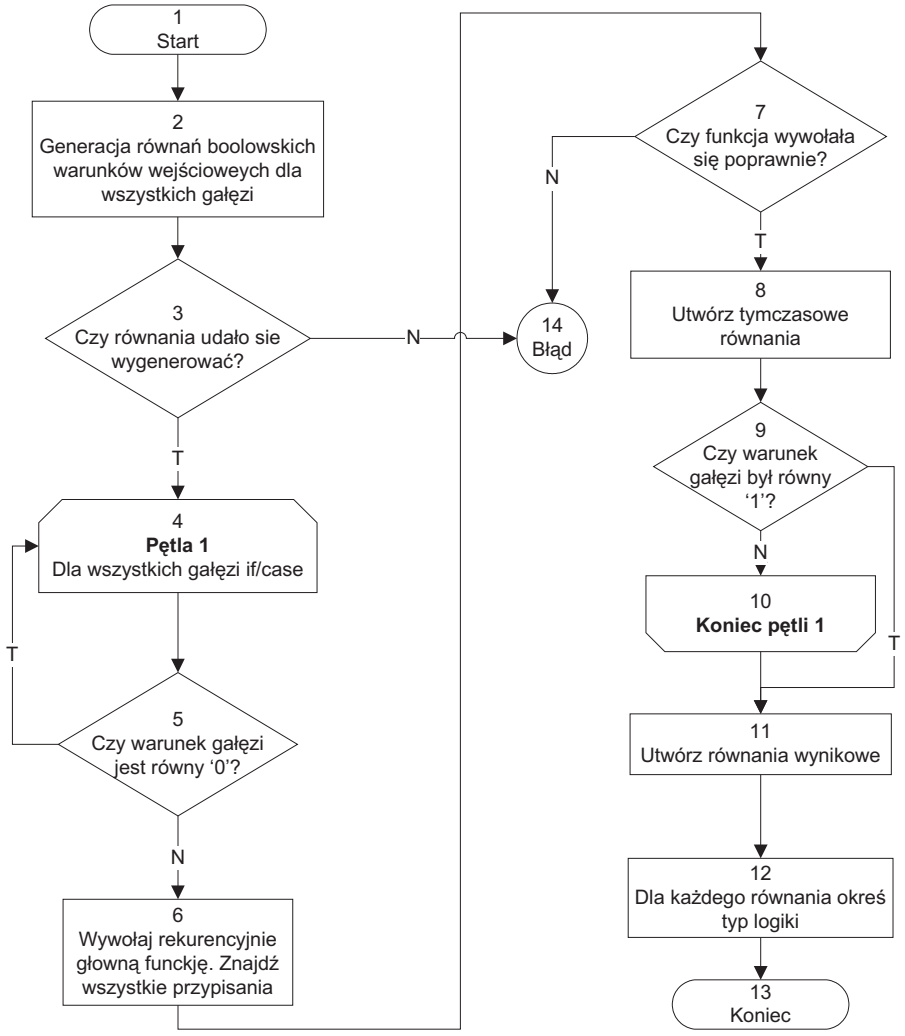
Na rysunku 3.4 przedstawiony został algorytm generacji równań dla instrukcji *if* oraz *case*. Poniżej krótkie omówienie poszczególnych jego elementów.

1. Generacja równań warunków wejściowych. W kroku tym (blok 2 i 3) tworzone są równania zgodnie ze wzorami 3.2, 3.3 i 3.4. Dla instrukcji *case* bardzo ważne jest sprawdzenie, czy wartości wejściowe się nie powtarzają, oraz czy pokrywają wszystkie możliwe wartości wyrażenia sterującego. Jeżeli tak to należy pominąć ewentualną gałąź *others*. W przypadku gdyby generacja elementarnych warunków wejściowych nie powiodła się cały proces jest oczywiście przerywany.
2. Wyszukiwanie wszystkich przypisań dla sygnałów i zmiennych w poszczególnych gałęziach *if* oraz *case* (blok 4-7). Dla każdej gałęzi wywoływana jest rekurencyjnie główna funkcja kompilacji procesu. Wcześniej jednak sprawdzane jest czy warunek aktywacji danej gałęzi nie jest równy „0", czyli *false*. W takiej sytuacji dana gałąź zostaje pominięta. Główna funkcja kompilacji procesu, zwraca listę przypisań (równania oraz niezbędne informacje semantyczne), o ile oczywiście wykonała się poprawnie.
3. Tworzenie równań tymczasowych (blok 8). W kroku tym tworzy się równania zgodnie ze wzorami 3.1 oraz 3.7 i 3.8. Nie można jeszcze określić typu logiki, więc generowane są równania w dwóch wersjach.
4. Sprawdzenie czy warunek wejściowy do danej gałęzi był równy „1" (blok 9). Jeżeli wynik sprawdzenia był pozytywny, to należy przerwać dalszą analizę takiej instrukcji *if*. Jeżeli była to jej pierwsza gałąź to, wszystkie przypisania znajdujące się wewnątrz należy traktować, tak jak by instrukcji *if* w ogóle nie było.
5. Utworzenie równań wynikowych (blok 11). Generowane są równania wynikowe. Dla tych sygnałów, które nie miały przypisań we wszystkich gałęziach, próbuje się zastosować wzór 3.6.
6. Ustalenie typu logiki (blok 12). Dla wszystkich bitów wszystkich sygnałów, które były przypisywane w danym *if* lub *case* ustala się typ logiki kombinacyjna czy sekwencyjna. Równanie dla danego sygnału (zmiennej) będzie w postaci kombinacyjnej, jeżeli spełniony jest jeden z poniższych warunków:
   — występuje po lewej stronie przypisania we wszystkich gałęziach, oraz instrukcja *if* zawiera gałąź *else*,
   — brak jest przypisania w jednej lub więcej gałęzi, ale występuje przypisanie przed instrukcją *if*,
   — warunek wejścia do gałęzi ma wartość logiczną „1".
   W pozostałych przypadkach otrzymujemy równania logiki sekwencyjnej.

3.11.1. Algorytm generacji warunków wejściowych dla instrukcji *case*

Tworzenie warunków wejściowych dla instrukcji *case* jest zadaniem na tyle skomplikowanym, że wymaga przedstawienia osobnym diagramie (3.5) wraz ze stosownym omówieniem. Oprócz równań warunków wejściowych funkcja ta dostarcza kilku pomocniczych informacji.

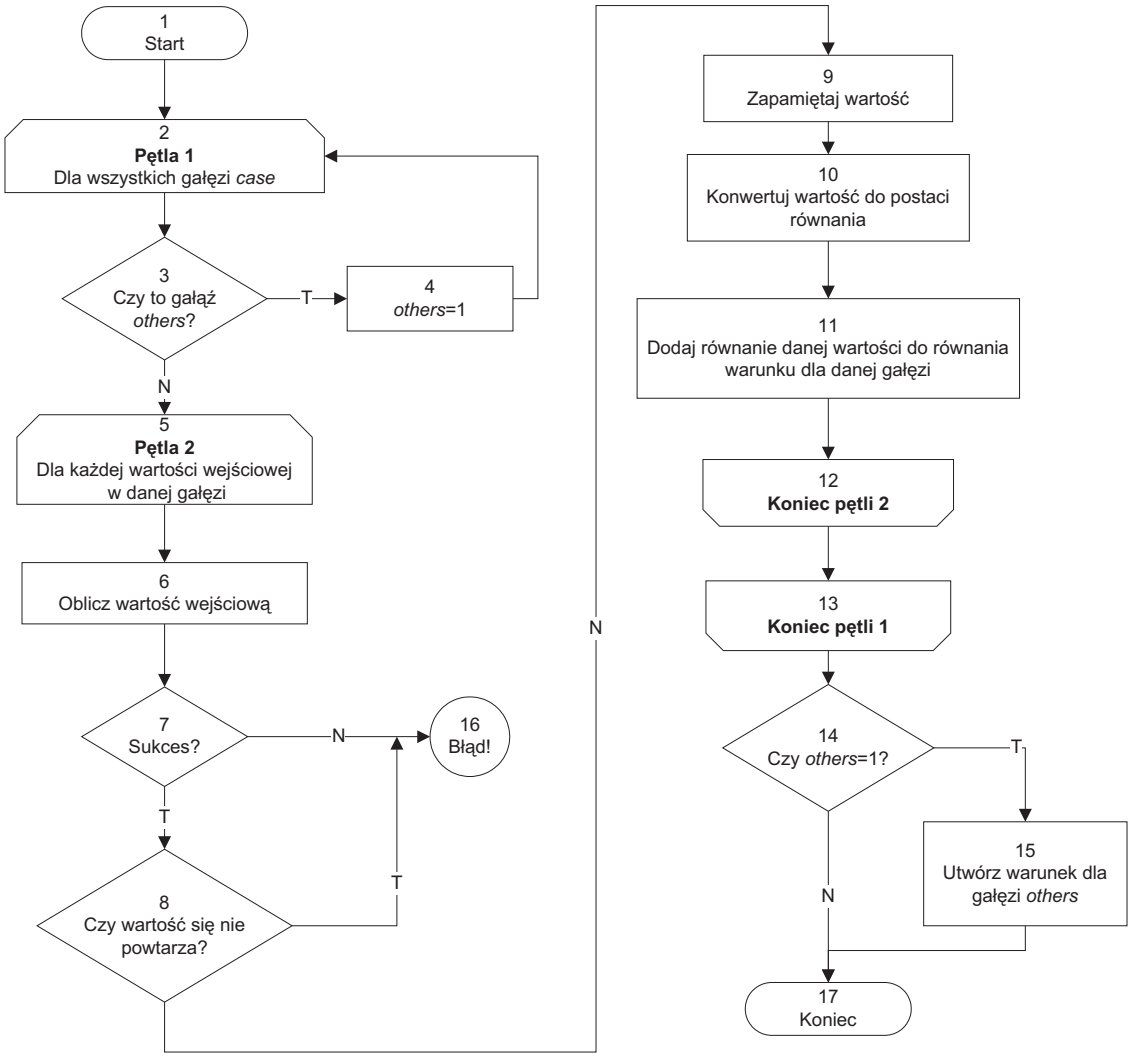
1. Funkcja generująca równania wartości wejściowe składa się z dwóch pętli: zewnętrznej, która porusza się po gałęziach instrukcji *case*, oraz wewnętrznej zajmującej się obliczaniem kolejnych wartości wejściowych.
2. Pierwszym krokiem (3,4) jest sprawdzenie czy dana gałąź nie jest gałęzią typu *others*. Zgodnie ze składnią języka VHDL gałąź taka może wystąpić w dowolnym miejscu instrukcji *case*. Jeżeli wynik sprawdzenia jest pozytywny, to należy zapamiętać numer gałęzi i przejść do tworzenia warunku dla następnej.
3. Obliczenie wartości wejściowej (7-9). Nie jest to zadanie proste, gdyż zgodnie ze specyfikacją języka, może ona być podana w różnej formie, z zakresem war-



Rysunek 3.4. Algorytm generacji równań dla instrukcji *if* oraz *case*. Źródło: opracowanie własne.

tości włącznie. Po obliczeniu wartości należy sprawdzić czy nie wystąpiła ona już wcześniej. Jest to jeden z większych problemów. Należy sobie uzmysłowić, iż rozmiar wyrażenia sterującego instrukcji *case* nie ma limitu. Może ono mieć, np. dwieście pięćdziesiąt sześć bitów szerokości. Do zapamiętania takiej wartości potrzebny jest typ, który zapewnia stosowną ilość miejsc znaczących. Aby uzyskać informację czy dana wartość wystąpiła wcześniej, należy przechowywać wszystkie dotychczas obliczone wartości. Powoduje to znaczne zwiększenie zapotrzebowania na zasoby systemowe, a przeglądanie takiej tablicy zwiększa czas kompilacji. Rozwiązaniem jest konwersja obliczonych wartości do postaci zajmującej jak najmniej miejsca, zastosowania np. funkcji skrótu.

4. Konwersja wartości do postaci równania (10). Obliczona wartość zamieniana jest następnie na formę przedstawioną przez wzór 3.3. Jeżeli rozmiar jej jest mniejszy niż rozmiar wyrażenia sterującego, to należy dokonać rozszerzenia. Dokonuje się tego przez zwielokrotnienie bitu znaku.
5. Tworzenie warunku (11). Równanie dla danej wartości dodaje się do równania warunku dla danej gałęzi, zgodnie ze wzorem 3.4.



Rysunek 3.5. Algorytm generacji warunków wejściowych dla instrukcji *case*. Źródło: opracowanie własne.

6. Gałąź *others* (14,15). Jeżeli wystąpiła gałąź others to należy utworzyć dla niej warunek wejściowy zgodnie ze wzorem 3.5.
7. Koniec. Po wygenerowaniu równań funkcja kończy swoje działanie. Oprócz równań zwracane są następujące informacje:
   — ilość gałęzi,
   — czy wystąpiła gałąź *others*,
   — czy wystąpiły wszystkie wartości wyrażenia sterującego.

   Uzyskanie ostatniej informacji nastręcza problemy. Najprościej było by wprowadzić licznik, którego wartość byłaby zwiększana po zidentyfikowaniu kolejnej wartości wejściowej. Po zakończeniu analizy, funkcja dokonywałaby sprawdzenia, czy wartość licznika jest równa liczbie dostępnych wartości wyrażenia sterującego, która wynosi: 2^n, gdzie n - jest ilością bitów wyrażenia sterującego. Trudno jednak zastosować tą metodę w przypadku tablic o elementach będących typami wyliczeniowymi jak np.: *bit_ vector*.

3.12. Generacja równań boolowskich dla instrukcji *for*

Generacja równań boolowskich dla pętli *for* języka VHDL jest zagadnieniem skomplikowanym. Poziom trudności jest zdecydowanie wyższy niż w przypadku wcześniej opisywanych instrukcji. Spowodowane jest to po pierwsze ograniczeniami jakie stawia synteza, po drugie specyfiką samej instrukcji. Pętla *for* może zawierać w sobie dowolną ilość instrukcji *next* i *exit*. Instrukcje te, w przypadku aktywacji zmieniają przepływ sterowania wewnątrz pętli, sprawiając że pewne fragmenty kodu nie zostają wykonane. W sytuacji gdy występuje kilka zagnieżdżonych pętli, to dana instrukcja *next* lub *exit* może nie tylko odwołać się do pętli w której bezpośrednio została umieszczona, ale także do wszystkich wyżej w hierarchii. Stanowi to istotny problem podczas kompilacji. W przypadku klasycznego kompilatora taka specyfika działania stanowiła by tylko drobną komplikacje, ale dla narzędzia mającego dokonać syntezy stanowi to poważny problem. Wynikiem kompilacji musi być bowiem w pełni funkcjonalna struktura sprzętowa.

Podsumowując powyższe oraz uwzględniając informacje przedstawione w rozdziale 2.5.3, można wyróżnić trzy główne problemy generacji równań dla pętli *for*:
— określenie liczby iteracji pętli,
— możliwość stosowania przez programistów układów zagnieżdżonych pętli,
— występowanie instrukcji *next* oraz *exit*, mogących oddziaływać na przepływ sterowania w pętli lub układzie pętli.

Zgodnie z wymogami projektu [70], którymi było zachowanie zgodności z kompilatorem firmy Synopsys FPGA Express[91], pętla *for* musi spełniać dwa warunki, aby została uznana za syntezowalna:
— ilość iteracji pętli musi być podana w sposób statyczny, czyli możliwy to określenia na etapie kompilacji,
— wewnątrz pętli nie można korzystać z instrukcji *wait*.

Wymagania te mają na celu zagwarantowanie skończonego czasu wykonania pętli.

Kompilacje pętli *for* można wykonać stosując jedno z dwóch podejść:
— tworząc automat skończony sterujący pętlą,
— dokonując zamiany pętli na jej alternatywną postać liniową.

Pierwsza metoda polega na zaprojektowaniu odpowiedniej maszyny stanów dla pętli *for*. Wiąże się to niestety z koniecznością zastosowania dużej ilości logiki sekwencyjnej, a to zawsze stwarza problemy. Syntezowanie automatu skończonego dla układów kilku wzajemnie zagnieżdżonych pętli wydaje się dosyć trudne. Rozwiązanie to jednak o ile oczywiście udałoby się je opracować, byłoby prawdopodobnie korzystniejsze pod kątem wymaganych zasobów.

Drugi sposób jest prostszy. Wymaga mniej pracy, a najważniejszą zaletą jest uniknięcie tworzenia skomplikowanej maszyny stanów. Zsyntetyzowany układ będzie posiadał lepsze charakterystyki czasowe, wyższą dopuszczalną częstotliwość pracy, ale będzie to okupione większą zajmowaną powierzchnią, a co za tym idzie wyższym zużyciem energii. Ponieważ niniejsza rozprawa zajmuje się problematyką logiki kombinacyjnej to zastosowanie metody linearyzacji pętli *for* jest de facto obligatoryjne.

W dalszej części rozdziału przeanalizowane zostało zachowanie pętli *for*. Przedstawiono kilka scenariuszy, zaczynając od pojedynczej pętli bez instrukcji *next* oraz *exit*, stopniowo przechodząc do sytuacji bardziej skomplikowanych takich jak pętle zagnieżdżone. Pokazany został wpływ instrukcji *next* i *exit* na zachowanie pętli.

Rozdział kończy się prezentacją algorytmu generacji równań boolowskich dla pętli *for*.

3.12.1. Alternatywna postać liniowa

Pierwszym krokiem procesu kompilacji pętli *for* jest uzyskanie *alternatywnej postaci liniowej* tejże pętli. Definiowana ona jest następująco:

> *Alternatywną postacią liniową* (APL) pętli *for*, określać będziemy równoważny pod względem uzyskiwanego wyniku, blok instrukcji sekwencyjnych nie zawierający jednak ani jednej instrukcji *for*.

Tworzenie APL będzie przebiegać różnie w zależności od postaci i liczby wzajemnie zagnieżdżonych pętli. Aby zgromadzić wiedzę potrzebną do opracowania uniwersalnego algorytmu należało przeanalizować następujące przypadki:

1. pojedyncza pętla *for*,
   — z instrukcją *next*,
   — z instrukcją *exit*,
2. dwie pętle (zagnieżdżone),
   — z instrukcjami *next*,
   — z instrukcjami *exit*,
3. trzy pętle (zagnieżdżone),
   — z instrukcjami *next*,
   — z instrukcjami *exit*.

Analiza kodu zawierającego więcej zagnieżdżonych instrukcji *for* jest bezcelowa, bowiem wszystkie mogące wystąpić tam sytuację da się sprowadzić do wyżej wymienionych przypadków.

O ile nie zostało wspomniane inaczej, wszystkie instrukcję *next* i *exit*, prezentowane dalej, będą zawierać warunek aktywacji. Mogą być dwie odmiany takiej postaci obu instrukcji:

— bezpośrednia - zgodna ze składnią, warunek występuje po słowie kluczowym *when*,
— pośrednia - kiedy *next*(*exit*) znajduje się wewnątrz instrukcji warunkowej *if* lub *case*.

3.13. Model pojedynczej pętli *for*

Pojedyncza pętla *for*, nie zawierająca instrukcji *next* oraz *exit* jest najprostszym przypadkiem do kompilacji. Model ten posłuży do przedstawienia generalnych kroków, które należy wykonać aby utworzyć liniową postać pętli. Te kroki to:

— określenie pierwszej i ostatniej wartości zmiennej iteracyjnej, oraz ilości iteracji,
— wypisanie wszystkich instrukcji znajdujących się wewnątrz bloku *for*, tyle razy ile wynosi ilość iteracji,
— zastąpienie każdego wystąpienia zmiennej iteracyjnej, właściwą dla danej iteracji wartością liczbową (literałem).

Powyższe zasady obowiązują dla wszystkich postaci pętli *for*. Ilustrację ich wykorzystania stanowi przykład 3.10. Widać, że instrukcje z wnętrza pętli zostały powtórzone tyle razy ile wynosi liczba iteracji - 3. Liczby oznaczone gwiazdką oznaczają początek kolejnych iteracji.

```
for i in 1 to 3 loop
        a(i) := b(i);
        c(i) := d(i);
end loop;

a(1) :=  b(1);  --(*1)
c(1) :=  d(1);
a(2) :=  b(2);  --(*2)
c(2) :=  d(2);
a(3) :=  b(3);  --(*3)
c(3) :=  d(3);
```

Przykład 3.10. Zamiany pętli *for* na postać liniową.

3.13.1. Wpływ instrukcji *next*

Sytuacja się nieco komplikuje i opisane powyżej zasady stają się niewystarczające. Kod znajdujący się za instrukcją *next*, może nie zostać wykonany. Jest to przedstawione na rysunku 3.6. W sytuacji, gdy warunek aktywacji *next* jest spełniony, żadna z instrukcji znajdujących się między nim, a końcem pętli nie zostanie wykonana. Sterowanie wraca na początek pętli. Aby dokonać zamiany takiej pętli na postać APL, należy zastąpić instrukcję *next*, blokiem *if* z jedną gałęzią, wewnątrz której umieszcza się wszystkie instrukcje znajdujące się pierwotnie za *next*. Postać warunku aktywacji *if* obrazuje wzór 3.9:

$$We_i = !Wnext_i \tag{3.9}$$

gdzie:
— $Wnext_i$ - równanie boolowskie dla warunku instrukcji next w i-tej iteracji,
— i - numer iteracji.

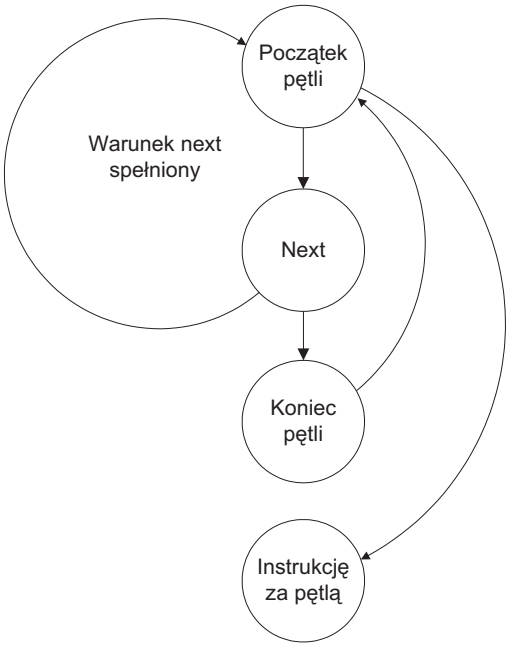
Przykład 3.11 przedstawia zamianę pętli *for* z instrukcją *next* na postać liniową. Widać utworzone instrukcje *if*, które zawierają fragment kodu znajdujący się pierwotnie za instrukcją *next*.

```
for i in 1 to 3 loop
        a(i) := b(i);
        next when a(i)=b(i);
        c(i) := d(i);
end loop;

a(1) := b(1);              --(*1);
if not (a(1)=b(1)) then
        c(1) := d(1);
end if;

a(2) := b(2);              --(*2);
if not (a(2)=b(2)) then
        c(3) := d(3);
end if;

a(3) := b(3);              --(*3);
if not (a(3)=b(3)) then
        c(3) := d(3);
end if;
```

Przykład 3.11. Pętla *for* z instrukcją *next*.

3.13.2. Wpływ instrukcji *exit*

Instrukcja ta powoduje większe komplikacje niż *next*, ponieważ w przypadku, gdy jej warunek aktywacji zostanie spełniony, nie tylko nie wykona się kod znajdu-



Rysunek 3.6. Przepływ sterowania w pętli z instrukcją *next*. Źródło: opracowanie własne.

jący się za nią, ale także następne iteracje. Zachowanie takie obrazuje rysunek 3.7. Zamiana pętli *for* z instrukcją *exit* na postać liniową wymaga wykonania poniższych czynności:

1. Utworzyć blok *if* o jednej gałęzi, w której umieszcza się cały kod znajdujący się za *exit*. Warunek aktywacji tworzy się zgodnie z równaniem 3.10:

$$Wwe_i = !Wexit_i \tag{3.10}$$

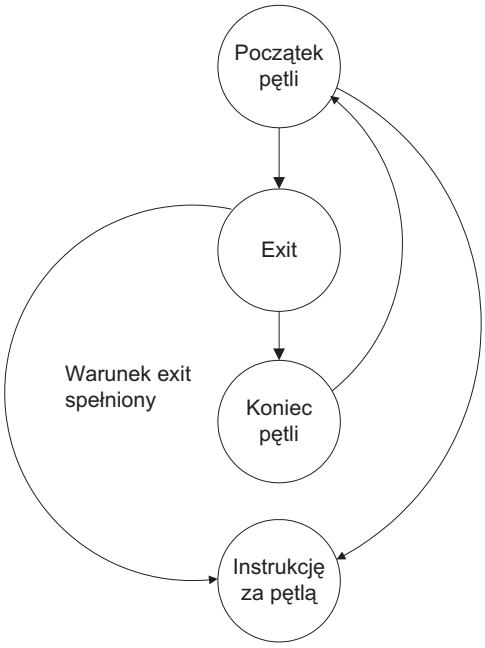
   gdzie:
   — $Wexit_i$ - równanie boolowskie dla warunku instrukcji exit w i-tej iteracji,
   — i - numer iteracji.
2. Dla wszystkich iteracji z wyjątkiem pierwszej, należy utworzyć blok *if* o jednej gałęzi zawierający wszystkie instrukcję znajdujące się w bloku pętli *for*. Gałęzią tą będzie sterował *warunek wykonania iteracji*, utworzony według wzoru 3.11:

$$W_i = \prod_{j=1}^{i-1} !Wexit_j \tag{3.11}$$

   gdzie:
   — i - numer iteracji, dla której tworzony jest warunek,
   — W_j - warunek wykonania $i - tej$ iteracji,
   — $Wexit_j$ - warunek aktywacji instrukcji exit w kolejnej iteracji.

   Sens równania 3.11 jest następujący: aby dana iteracja mogła zostać wykonana, w żadnej z iteracji ją poprzedzających instrukcja *exit* nie mogła zostać aktywowana.

Przykład 3.12 pokazuje pętle *for* z instrukcją *exit* oraz jej postać liniową. Widać, że wykonanie każdej iteracji poza pierwszą uzależnione jest od spełnienia warunku - wzór 3.11.



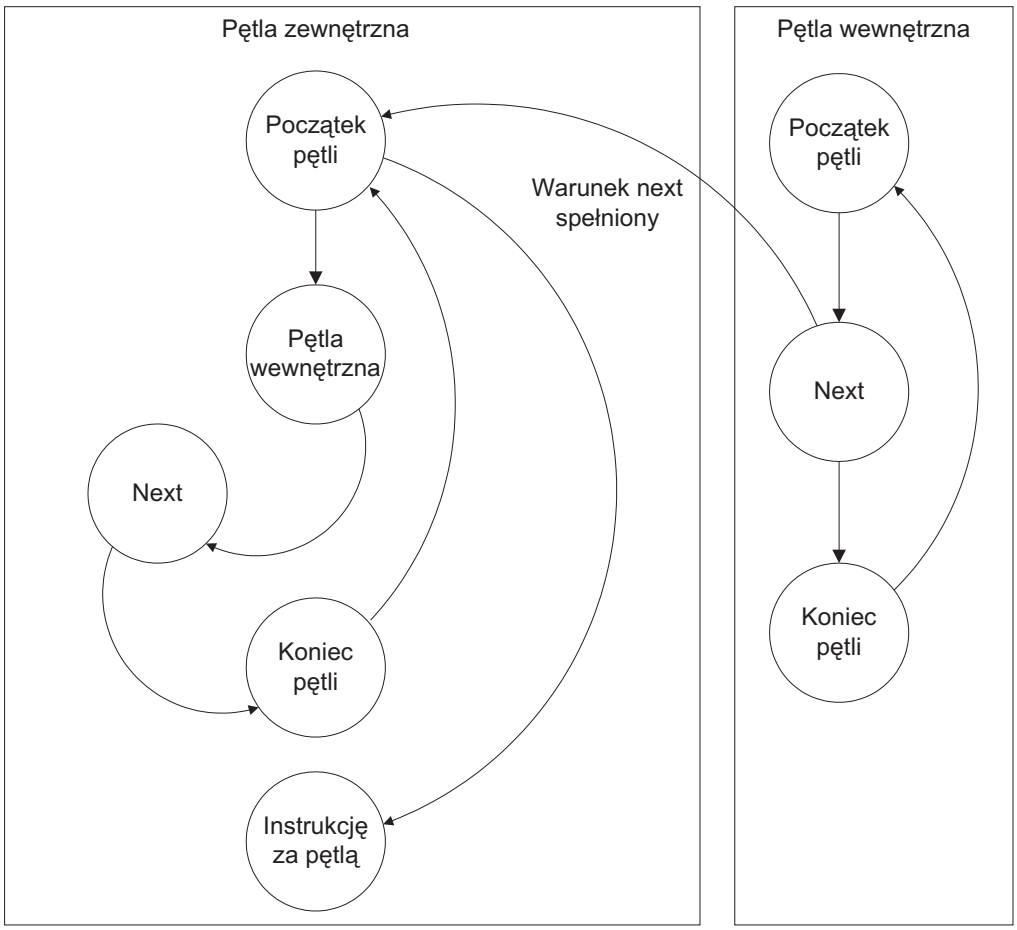
Rysunek 3.7. Przepływ sterowania w pętli *for* z instrukcją *exit*. Źródło: opracowanie własne.

```
for i in 1 to 3 loop
        a(i) := b(i);
        exit when a(i)=b(i);
        c(i) := d(i);
end loop

a(1) := b(1);                                   --(*1)
if not (a(1)=b(1)) then
        c(1) := d(1);
end if;

if not (a(1)=b(1)) then                         --(*2)
        a(2) := b(2);
        if not (a(2)=b(2)) then
                c(2) := d(2);
        end if;
end if;

if (not(a(1)=b(1))and (not(a(2)=b(2)))) then --(*3)
        a(3) := b(3);
        if not (a(3)=b(3)) then
                c(3) := d(3);
        end if;
end if;
```

Przykład 3.12. Pętla *for* wraz z instrukcją *exit*.

3.14. Model dwóch zagnieżdżonych pętli *for*

Układ dwóch zagnieżdżonych pętli jest już znacznie trudniejszy do analizy i kompilacji. Poszczególne instrukcje *next* i *exit* mogą odnosić się do pętli w której się bezpośrednio znajdują, ale także do pętli która jest wyżej. Przepływ sterowania komplikuje się, rośnie ilość dodatkowej logiki, którą należy sztucznie utworzyć.



Rysunek 3.8. Przepływ sterowania w układzie dwóch zagnieżdżonych pętli *for* zawierającymi instrukcje *next*. Źródło: opracowanie własne.

3.14.1. Dwie pętle zawierające instrukcję *next*, odnoszące się do pętli zewnętrznej

Model przepływu sterowania w takiej konfiguracji pętli przedstawia rysunek 3.8. Jak widać powrót na początek pętli zewnętrznej może spowodować zarówno *next* znajdujący się bezpośrednio w niej, jak i ten który znajduje się w pętli wewnętrznej. Analiza instrukcji *next* znajdującej się w pętli zewnętrznej nie dostarcza nowych informacji, jej zachowanie jest takie samo jak w pojedynczej pętli. W przypadku *next* z pętli wewnętrznej sytuacja wygląda inaczej. Jej aktywacja niesie następujące skutki:

— przerwanie wykonania bieżącej iteracji pętli wewnętrznej, nie wykonają się także pozostałe iteracje tej pętli, gdyż sterowanie wraca do pętli zewnętrznej,
— nie wykona się żadna z tych instrukcji pętli zewnętrznej, które znajdują się za blokiem pętli wewnętrznej.

Na wydruku 3.13 zaprezentowane są dwie zagnieżdżone pętle *for*. Pętla wewnętrzna zawiera instrukcje *next* odnoszącą się do pętli zewnętrznej. Powoduje to powstanie *warunków wykonania iteracji* pętli wewnętrznej (linie 6 i 20) oraz *warunków wykonania części dalszej* pętli zewnętrznej (linie 12 i 26).

```
loop1: for i in 0 to 1 loop
        a1(i) := b1(i);
        loop2: for j in 0 to 1 loop
                a2(i,j) := b2(i,j);
                next loop1 when a2(i,j)>b2(i,j);
                c2(i,j) := d2(i,j);
        end loop loop2;
-- część pętli loop1 za pętlą wewnętrzną
        c1(i) := d1(i);
end loop loop1;

{1}     a1(0) := b1(0);                          --(*I)
{2}     a2(0,0) := b2(0,0);                      --(*1)
{3}     if not (a2(0,0) > b2(0,0)) then
{4}             c2(0,0) := d(0,0);
{5}     end if;
{6}     if not(a2(0,0) > b2(0,0)) then           --(*2)
{7}             a2(0,1) := b2(0,1);
{8}             if not (a2(0,1) > b2(0,1)) then
{9}                     c2(0,1) := d(0,1);
{10}            end if;
{11}    end if;
{12}    if not( a2(0,0) > b2(0,0))
        and not (a2(0,1) > b2(0,1)) then
{13}            c1(0) := d1(0);
{14}    end if;
{15}    a1(1) := b1(1);                  --(*II)
{16}    a2(1,0) := b2(1,0);              --(*1)
{17}    if not (a2(1,0) > b2(1,0)) then
{18}            c2(1,0) := d(1,0);
{19}    end if;
{20}    if not(a2(1,0) > b2(1,0)) then  --(*2)
{21}            a2(1,1) := b2(1,1);
{22}            if not (a2(1,1) > b2(1,1)) then
{23}                    c2(1,1) := d(1,1);
{24}            end if;
{25}    end if;
{26}    if not( a2(1,0) >b2(1,0))
                        and not (a2(1,1) > b2(1,1)) then
{27}            c1(1) := d1(1);
{28}    end if;
```

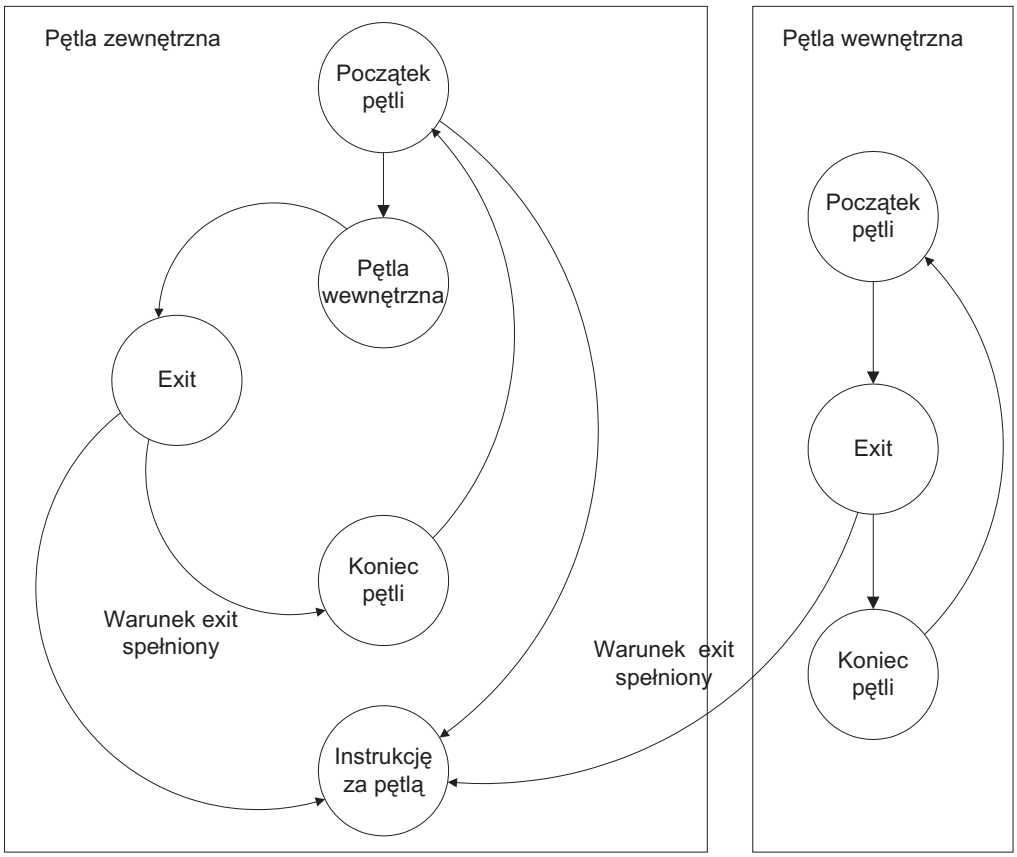
Przykład 3.13. Dwie pętle *for* wraz z instrukcjami *next*.

3.14.2. Dwie pętle zawierające instrukcję *exit*, odnoszące się do pętli zewnętrznej

Model przepływu sterowania w takiego układu pętli przedstawia rysunek 3.9. Zachowanie instrukcji *exit* znajdującej się w pętli zewnętrznej będzie identyczne jak w przypadku pojedynczej pętli. Można ją więc pominąć. Aktywacja *exit* umieszczonego wewnątrz pętli zagnieżdżonej powoduje:

— przerwanie wykonania bieżącej iteracji pętli wewnętrznej, nie wykonają się także pozostałe iteracje tej pętli, gdyż sterowanie wraca do pętli zewnętrznej,
— nie wykonanie się wszystkich tych instrukcji pętli zewnętrznej, które znajdują się za blokiem pętli wewnętrznej,
— przerwanie wykonania pętli zewnętrznej.

Przykład 3.14 stanowi ilustrację powyższego zachowania. Instrukcja *exit* powoduje przerwanie obu pętli. Konieczne jest utworzenie *warunków wykonania iteracji* dla pętli zewnętrznej (linia 15) i wewnętrznej (linie 6 i 21). Oprócz tego powstają *warunki wykonania części dalszej* (linie 12 i 27).



Rysunek 3.9. Przepływ sterowania w układzie dwóch zagnieżdżonych pętli zawierających instrukcje *exit*. Źródło: opracowanie własne.

3.14.3. Podsumowanie

Analiza zachowania dwóch zagnieżdżonych pętli *for* zawierających instrukcje *next* i *exit* jest trudniejsza niż przypadku z pojedynczą pętlą. Przepływ sterowania jest bardziej skomplikowany. Wzajemne oddziaływanie pętli i obu instrukcji powoduje, że pewne fragmenty wykonają się wtedy i tylko wtedy, gdy spełnione będą strzegące ich warunki. Istnieją trzy takie warunki:

1. Warunek wykonania iteracji pętli zewnętrznej. Może pojawić się w każdej z iteracji z wyjątkiem pierwszej. Jego spełnienie decyduje o tym czy dana iteracja zostanie wykonana. Warunek ten powstaje gdy:
   — pętla zewnętrzna zawiera instrukcje *exit*,
   — pętla wewnętrzna zawiera instrukcje *exit* z etykietą pętli zewnętrznej.
2. Warunek wykonania instrukcji znajdujących się w pętli zewnętrznej, za blokiem pętli wewnętrznej. Powstaje gdy:
   — pętla wewnętrzna zawiera instrukcje *exit* z etykietą pętli zewnętrznej,
   — wewnątrz pętli wewnętrznej znajduje się instrukcja *next* z etykietą pętli zewnętrznej.
3. Warunek wykonania iteracji pętli wewnętrznej. Powstaje gdy:
   — pętla wewnętrzna zawiera instrukcję *exit*,
   — pętla wewnętrzna zawiera instrukcje *next* z etykietą pętli zewnętrznej.

```
loop1: for i in 0 to 1 loop
        a1(i) := b1(i);
        loop2: for j in 0 to 1 loop
                a2(i,j) := b2(i,j);
                exit loop1 when a2(i,j)>b2(i,j);
                c2(i,j) := d2(i,j);
        end loop loop2;
-- część pętli loop1 za pętlą wewnętrzną
        c1(i) := d1(i);
end loop loop1;

{1}     a1(0) := b1(0);                          --(*I)
{2}     a2(0,0) := b2(0,0);                      --(*1)
{3}     if not (a2(0,0) > b2(0,0)) then
{4}             c2(0,0) := d(0,0);
{5}     end if;
{6}     if not(a2(0,0) > b2(0,0)) then  (*2)
{7}             a2(0,1) := b2(0,1);
{8}             if not (a2(0,1) > b2(0,1)) then
{9}                     c2(0,1) := d(0,1);
{10}            end if;
{11}    end if;
{12}    if not( a2(0,0) > b2(0,0)) and not
                        (a2(0,1) > b2(0,1)) then
{13}            c1(0) := d1(0);
{14}    end if;

{15}    if not( a2(0,0) > b2(0,0)) and not --(*II)
                (a2(0,1) > b2(0,1)) then )
{16}            a1(1) := b1(1);
{17}            a2(1,0) := b2(1,0);              --(*1)
{18}            if not (a2(1,0) > b2(1,0)) then
{19}                    c2(1,0) := d(1,0);
{20}            end if;
{21}            if not(a2(1,0) > b2(1,0)) then  --(*2)
{22}                    a2(1,1) := b2(1,1);
{23}                    if not (a2(1,1) > b2(1,1)) then
{24}                            c2(1,1) := d(1,1);
{25}                    end if;
{26}            end if;
{27}            if not( a2(1,0) >b2(1,0))
                and not (a2(1,1) > b2(1,1)) then
{28}                    c1(1) := d1(1);
{29}            end if;
{30}    end if;
```

Przykład 3.14. Dwie pętle *for* wraz z instrukcjami *exit*.

3.15. Model trzech zagnieżdżonych pętli *for*

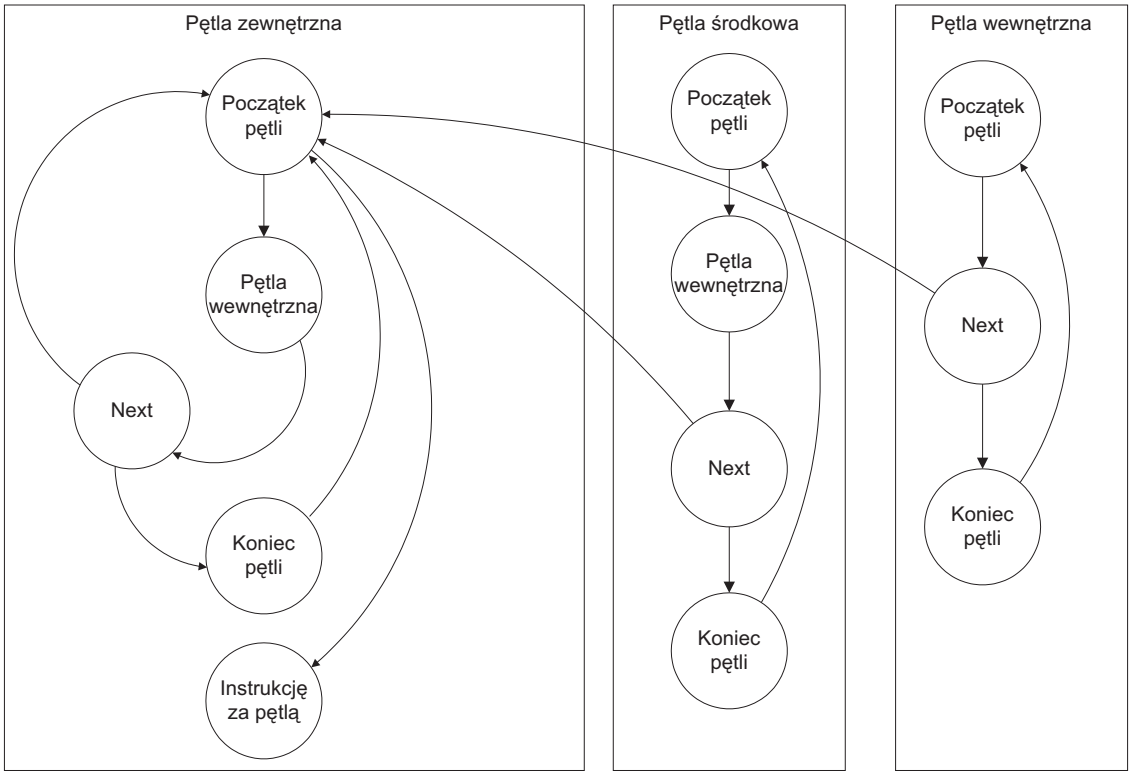
Model ten stanowi ostatni etap analizy zachowania pętli *for*. Tak jak w poprzednim rozdziale, tak i tutaj należy skupić się na sytuacji kiedy instrukcje *next* i *exit* znajdują się w pętli wewnętrznej i środkowej, i odnoszą się do pętli zewnętrznej.

3.15.1. Pętle zawierają instrukcje *next*

Układ taki prezentowany jest na rysunku 3.10. Aktywacja instrukcji *next* z pętli wewnętrznej powoduje:

— przerwanie wykonania bieżącej iteracji pętli wewnętrznej, nie wykonają się także pozostałe iteracje tej pętli, gdyż sterowanie wraca do pętli zewnętrznej,

— w bieżącej iteracji nie wykonają się te wszystkie instrukcje pętli środkowej, które znajdują się za blokiem pętli wewnętrznej, następne iteracje pętli środkowej także nie wykonają się,



Rysunek 3.10. Przepływ sterowania w układzie trzech pętli *for* z instrukcjami *next*. Źródło: opracowanie własne.

— nie wykonanie się wszystkich tych instrukcji pętli zewnętrznej, które znajdują się za blokiem pętli środkowej.

Aktywacja instrukcji *next* znajdującej się w pętli środkowej pociąga za sobą następujące zdarzenia:

— przerwanie bieżącej iteracji pętli środkowej, nie wykonają się także następne iteracje,
— nie wykonanie się wszystkich tych instrukcji pętli zewnętrznej, które znajdują się za blokiem pętli środkowej.

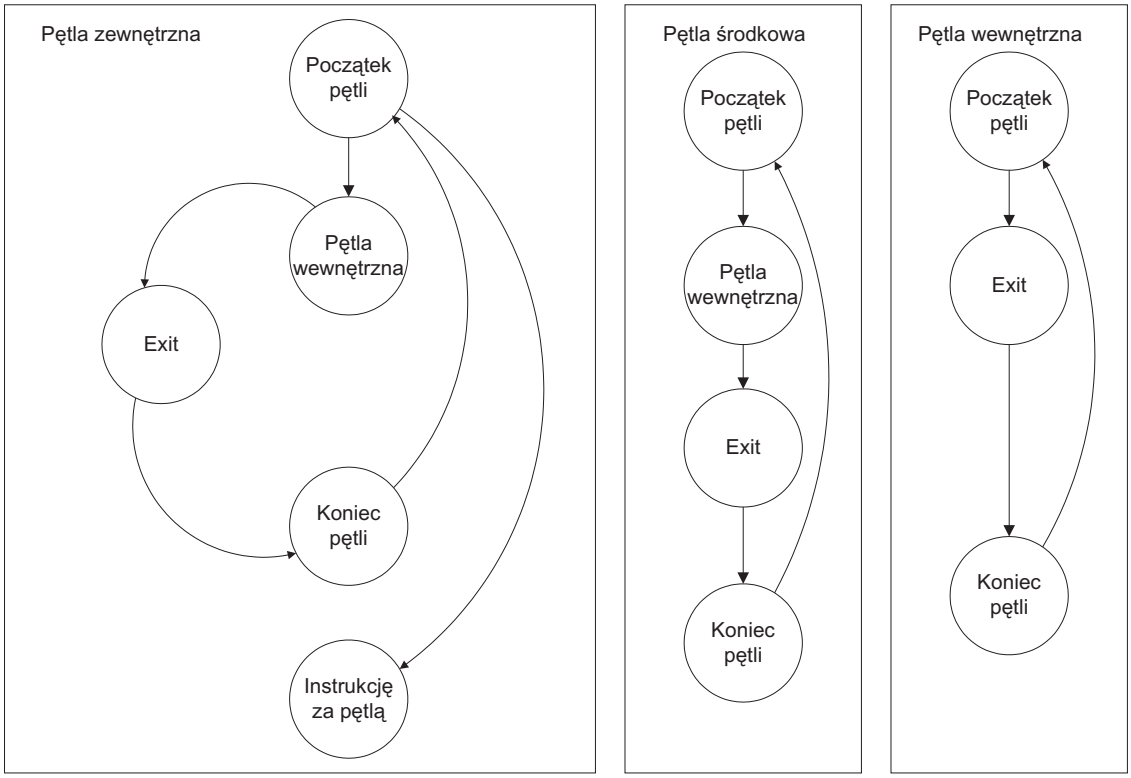
3.15.2. Pętle zawierają instrukcje *exit*

Rysunek 3.10 ukazuje przepływ sterowania w takim układzie. Aktywacja instrukcji *exit* znajdującej się w pętli wewnętrznej powoduje następujący ciąg zdarzeń:

— przerwanie wykonania bieżącej iteracji pętli wewnętrznej, nie wykonają się także pozostałe iteracje tej pętli, gdyż sterowanie wraca do pętli zewnętrznej,
— nie wykonanie się wszystkich tych instrukcji pętli środkowej, w jej bieżącej iteracji, które znajdują się za blokiem pętli wewnętrznej, przerwanie wykonania pętli środkowej,
— nie wykonanie się wszystkich tych instrukcji pętli zewnętrznej, w jej bieżącej iteracji, które znajdują się za blokiem pętli środkowej, przerwanie wykonania pętli zewnętrznej.

Aktywacja instrukcji *exit* z pętli środkowej powoduje:

— przerwanie wykonania bieżącej iteracji pętli środkowej, nie wykonają się także pozostałe iteracje tej pętli, gdyż sterowanie wraca do pętli zewnętrznej,



Rysunek 3.11. Przepływ sterowania w układzie trzech pętli *for* z instrukcjami *exit*. Źródło: opracowanie własne.

— nie wykonanie się wszystkich tych instrukcji pętli zewnętrznej, w jej bieżącej iteracji, które znajdują się za blokiem pętli środkowej, następuje przerwanie wykonania pętli zewnętrznej.

3.15.3. Podsumowanie

W konfiguracji składającej się z trzech pętli *for* oraz nieokreślonej liczby instrukcji *next* oraz *exit* wykonanie fragmentów kodu lub całych iteracji będzie uzależnione od spełnienia strzegących ich warunków. Te warunki to:

1. Warunek wykonania instrukcji znajdujących za pętlą środkową. Tworzy go:
   — każda instrukcja *exit*, przerywająca działanie pętli zewnętrznej, a znajdująca się pętli środkowej lub wewnętrznej,
   — instrukcja *next* znajdująca się w pętli środkowej, lub wewnętrznej, a której celem jest pętla zewnętrzna.
2. Warunek wykonania iteracji pętli środkowej. Powstaje, gdy:
   — istnieje chociaż jedna instrukcja *exit* przerywająca tą pętle (w tej pętli lub wewnętrznej),
   — w pętli wewnętrznej jest instrukcja *next*, której celem jest pętla zewnętrzna.
3. Warunek wykonania instrukcji znajdujących się za pętlą wewnętrzną. Powstaje, gdy:
   — w pętli wewnętrznej znajduje się instrukcja *exit*, przerywająca działanie pętli środkowej, lub zewnętrznej,
   — w pętli wewnętrznej znajduje się instrukcja *next*, odnosząca się do pętli środkowej, lub zewnętrznej.

4. Warunek wykonania iteracji pętli wewnętrznej. Powstaje, gdy:
   — w pętli wewnętrznej znajduje się instrukcja *exit* odnosząca się do dowolnej pętli,
   — w pętli wewnętrznej znajduje się instrukcja *next*, a jej celem jest pętla środkowa, lub zewnętrzna.

3.16. Algorytm generacji równań boolowskich dla instrukcji *for*

Powyższe przykłady pozwalają uzmysłowić sobie jak skomplikowana może być generacja równań boolowskich dla pętli *for*. Należy pamiętać, że zaprezentowane przykłady i tak nie pokazują wszystkich problemów na jakie można natrafić. Nie ma tutaj bowiem sytuacji, kiedy *next* czy *exit* znajdują się wewnątrz instrukcji warunkowych *if* albo *case*. Nie pokazany został także przypadek łączny, gdy oba rodzaje instrukcji znajdują się wewnątrz pętli. Mimo wszystko analiza zachowania pętli *for* w przedstawionych tutaj różnych postaciach dostarczyła wiedzy niezbędnej do opracowania uniwersalnego algorytmu kompilacji. Widać bowiem wyraźnie zależności między omawianymi instrukcjami, co z kolei stanowi znaczący krok na drodze do rozwiązania problemu. Pozostało tylko zaprezentowaną wiedzę uporządkować i przedstawić w algorytmicznej, gotowej do praktycznej implementacji postaci.

Cały proces zamiany instrukcji *for* na alternatywną postać liniową APL należy podzielić na następujące etapy:
— analiza i konwersja instrukcji *next* oraz *exit*,
— analiza i konwersja pętli *for* jako całości.
Proponowany podział ułatwia pracę z tak dużym zagadnieniem. Instrukcje *next* i *exit* działają na innym poziomie niż pętla, a do tego są między nimi duże różnice. Wszystko to sprawia, że przedstawienie jednego całościowego algorytmu jest trudne, a dodatkowo nie byłby on czytelny.

3.16.1. Wiadomości wstępne - definicje

Algorytmy zaprezentowane w dalszej części pracy bazują na pewnych pojęciach. W tym miejscu zostaną one przedstawione i zdefiniowane.

Część dalsza. Niech będą dane dwie pętle a i b, przy czym niech b będzie wewnątrz a. *Częścią dalszą* pętli a, określać będziemy wszystkie te jej instrukcje, które znajdują się za blokiem pętli b. Dla pętli b, *część dalsza* nie występuje.

Niech dane będzie n pętli *for* o numerach od 1 do n, przy czym pętla o numerze n będzie najbardziej zagnieżdżona. W każdej pętli, z wyjątkiem tej o numerze n, mogą wystąpić dwa warunki:
— *warunek wykonania iteracji*,
— *warunek wykonania części dalszej*.

Niech dane będą instrukcje $next_{ij}$ lub $exit_{ij}$, przy czym indeksy oznaczają:
— i - numer pętli w której znajduje się instrukcja,
— j - numer pętli do której odnosi się instrukcja,
w sytuacji gdy spełniony jest warunek $j \leq i$.

Uwzględniając powyższe:
— warunek danej instrukcji $exit_{ij}$ wystąpi w *warunkach wykonania iteracji* pętli o numerach $k = i..j$,

— warunek danej instrukcji $exit_{ij}$ wystąpi w *warunkach wykonania części dalszej* wszystkich pętli o numerach $k = i..j-1$,
— warunek danej instrukcji $next_{ij}$ będzie częścią składową *warunku wykonania iteracji* dla wszystkich pętli o numerach $k = i+1..j$,
— warunek danej instrukcji $next_{ij}$ będzie częścią składową *warunku wykonania części dalszej* dla pętli o numerach $k = i..j-1$.

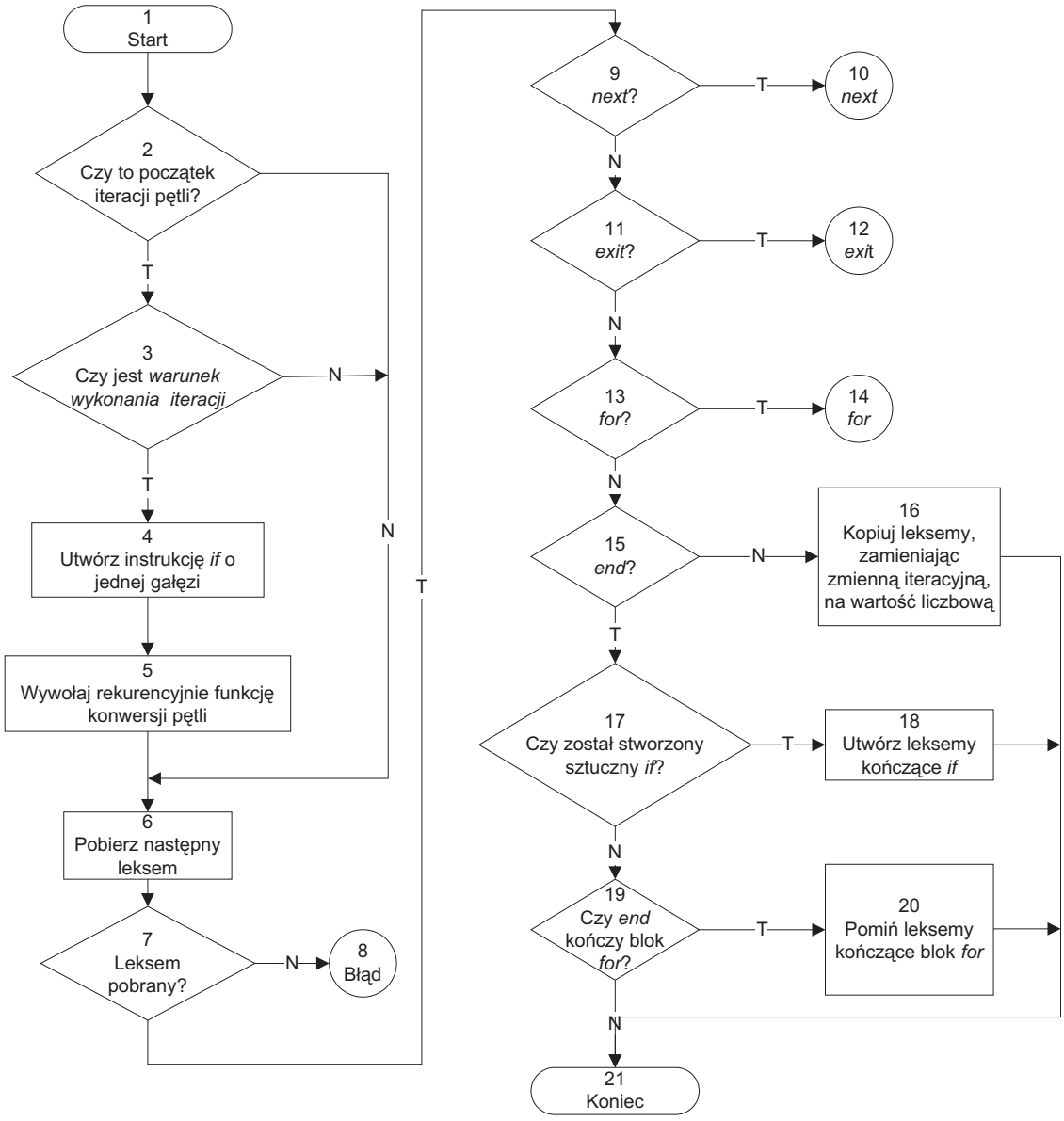
Do powyższych warunków dochodzą warunki pojedynczych instrukcji *next* i *exit*.

Algorytm kompilacji instrukcji *for* (a także *next* i *exit*), pracuje w nieco innym kontekście, niż ten zajmujący się instrukcjami *if* i *case*. Dzieje się tak, dlatego że jego zadaniem nie jest wygenerowanie równań, a jedynie stworzenie *alternatywnej postaci liniowej*. Krok ten poprzedza właściwą generację równań. Dane wejściowe dla algorytmu stanowi zbiór leksemów składający się na instrukcję process języka VHDL, poddany wcześniejszej analizie syntaktycznej i semantycznej (z kilkoma wyjątkami). Algorytm działa w sposób rekurencyjny. Wynikiem jego działania jest zbiór leksemów, ale już bez instrukcji *for*, *next* i *exit*. Całość składa się z trzech algorytmów cząstkowych:
— algorytmu funkcji konwersji - główna pętla analizująca ciało instrukcji *process* rekurencyjnie w poszukiwaniu pętli *for* i instrukcji towarzyszących - *next* i *exit*,
— algorytmu linearyzacji pętli *for* - zamienia instrukcję *for* na APL,
— algorytmu obsługi *next* i *exit*.

3.16.2. Algorytm funkcji konwersji pętli *for*

Najpierw zostanie omówiony algorytm stanowiący trzon całego procesu tworzenia APL czyli funkcja konwersji. Przedstawia go rysunek 3.12. Jego sposób funkcjonowania przedstawia się następująco:

1. **Sprawdzenie czy dla bieżącego kontekstu istnieje *warunek wykonania iteracji* (bloki 2-5)**. Pierwszym krokiem jest określenie czy funkcja konwersji znajduje się na początku kolejnej iteracji pętli. Dokonuje się tego sprawdzając informację semantyczną. Jeżeli weryfikacja jest pozytywna, to następną czynnością jest zbadanie, czy istnieje warunek wykonania tej iteracji. Jeśli tak to należy utworzyć leksemy bloku *if* o jednej gałęzi. Następnie funkcja konwersji wywoływana jest rekurencyjnie.
2. **Pobranie leksemów kolejnej instrukcji (bloki 6-8)**. Funkcja stara się zidentyfikować następną instrukcję i wywołać odpowiedni algorytm obsługi (bloki 9-14). Jeżeli dany leksem nie jest częścią instrukcji *for*, *if*, *case*, *next* to jest on po prostu kopiowany (blok 16). Podczas tej czynności sprawdza się czy nie jest on identyfikatorem reprezentującym zmienną iteracyjną. Jeżeli jest, to w takim przypadku zamienia się go na leksem literału o wartości takiej, jaką ma w bieżącej iteracji danej pętli zmienna iteracyjna.
3. **Obsługa końca bieżącego kontekstu (bloki 15,17-20)**. Leksem *end* oznacza koniec zagnieżdżenia. W zależności od tego do jakiej instrukcji należy dany *end* podejmowane są następujące działania:
   — *for* - pomijamy leksemy bloku *for*,
   — *process* - funkcja kończy swoje działanie,
   — w pozostałych przypadkach leksemy kopiuje się.
   Wcześniej jednak sprawdzane jest czy w danym kontekście wywołania funkcji konwersji nie został utworzony sztuczny *if*, a jeżeli tak to generowane są leksemy stanowiące zamknięcie tejże instrukcji.

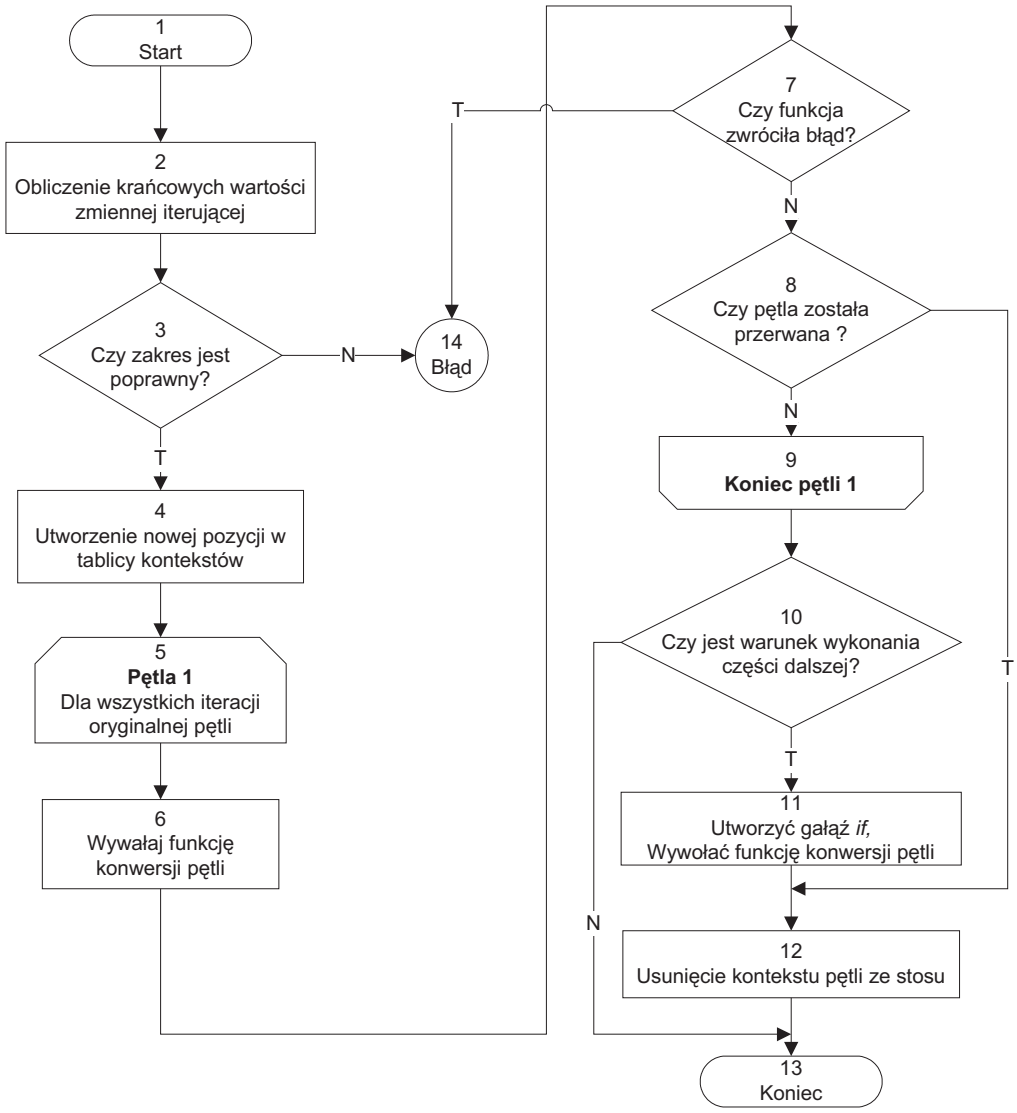


Rysunek 3.12. Algorytm funkcji konwersji pętli *for*. Źródło: opracowanie własne.

3.16.3. Linearyzacja pętli *for*

Kolejnym algorytmem, który należy omówić jest algorytm linearyzacji instrukcji *for* (rysunek 3.13). Poniżej znajduje się opis najważniejszych jego elementów:

1. **Określenie wartości granicznych zmiennej iteracyjnej**. Pierwszym krokiem jest obliczenie liczby iteracji pętli (zakresu). Jest to kłopotliwe, gdyż zakres może być podany w różnych postaciach. Czynności składające się na ten krok:
   — obliczenie dolnej i górnej wartości zmiennej iteracyjnej,
   — sprawdzenie czy wartości te są statyczne, tzn. czy wynik jest liczbą, a nie równaniami,
   — dla zakresu w postaci *liczba*1 *to liczba*2 sprawdzenie czy *liczba*1 jest mniejsza lub równa *liczba*2,
   — dla zakresu w postaci *liczba*1 *downto liczba*2 sprawdzenie czy *liczba*1 jest większa lub równa *liczba*2.



Rysunek 3.13. Algorytm konwersja pętli *for* do postaci liniowej. Źródło: opracowanie własne.

2. **Utworzenie kontekstu pętli**. Kontekst to inaczej informacja semantyczna skojarzona z pętlą. Składają się na nią następujące elementy:
   — *warunek wykonania iteracji*,
   — *warunek wykonania części dalszej*,
   — numer pętli,
   — etykieta, jeśli dana pętla ją posiada.
   Ponieważ nie ma ograniczeń co do ilości zagnieżdżonych pętli, kontekst skonstruowany jest jako stos. Na wierzchu znajduje się informacja dotycząca bieżącej pętli.
3. **Zamiana iteracji na postać liniową** (bloki 5-9). Tyle razy ile wynosi liczba iteracji wywoływana jest funkcja konwersji pętli. W każdym wywołaniu globalna informacja semantyczna zawiera bieżącą wartość zmiennej iteracyjnej, którą należy zastąpić identyfikator. Po każdym wywołaniu funkcji sprawdza się czy wykonała się poprawnie, tzn. nie zwrócony został kod błędu. Należy też sprawdzić czy iteracja lub nawet cała pętla nie została przerwana w sposób bezwarunkowy

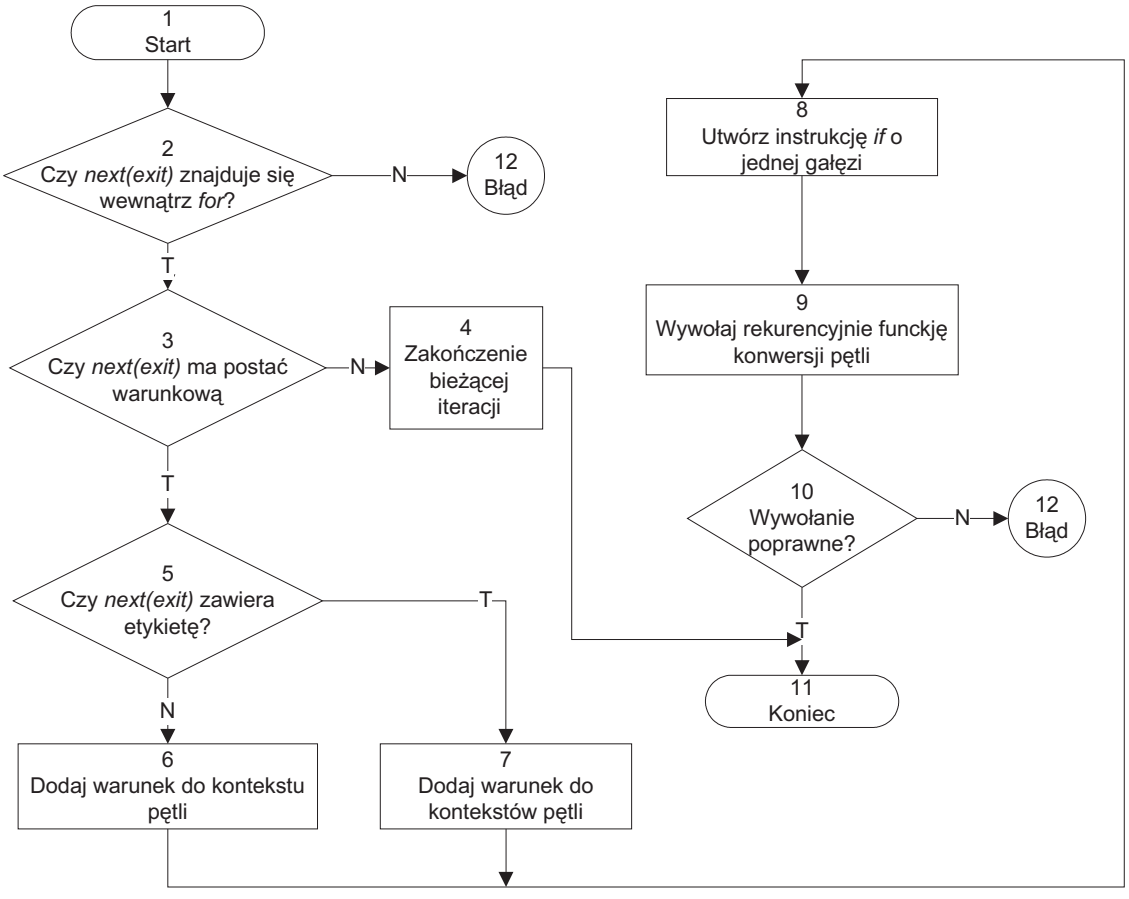
(*next*, *exit* nie zawierające warunku aktywacji). W takiej bowiem sytuacji analiza danej pętli zostaje zakończona.

4. **Obsługa warunku wykonania części dalszej** (bloki 10-11). Po wykonaniu wszystkich iteracji należy sprawdzić czy warunek wykonania części dalszej jest pusty. Jeżeli nie, to należy utworzyć leksemy bloku *if* i wywołać rekurencyjnie funkcję konwersji.
5. **Usunięcie kontekstu pętli ze stosu** (blok 12). Po zakończeniu analizy pętli, informacja semantyczna z nią związana nie jest już potrzebna.

3.16.4. Algorytm obsługi instrukcji *next* i *exit*

Jako ostatni zostanie opisany algorytm usuwania instrukcji *next* oraz *exit*. Mimo oczywistych różnic w działaniu obu komend, metoda postępowania jest bardzo podobna. Odmienność dotyczy oddziaływania obu instrukcji na warunki sterujące pętlą bądź układem pętli (rozdział 3.16.1). Algorytm przedstawiony jest na rysunku 3.14, a poniżej opisano dokładnie najważniejsze jego elementy.

1. **Sprawdzenie czy dany *next*(*exit*) znajduje się wewnątrz *for***. Instrukcje *next* i *exit* nie mogą się znajdować poza pętlą (blok 2).
2. **Określenie postaci instrukcji**. Jeżeli nie jest to postać warunkowa, należy natychmiast zakończyć analizę bieżącej iteracji.
3. **Sprawdzenie czy dany *next*(*exit*)** posiada etykietę. Obecność etykiety oznacza, że dana instrukcja może się odnosić do pętli innej niż ta, w której się bezpo-



Rysunek 3.14. Algorytm obsługi instrukcji *next* i *exit*. Źródło: opracowanie własne.

średnio znajduje. Należy sprawdzić czy istnieje pętla o takiej etykiecie. Oprócz tego dana instrukcja *next*(*exit*) musi się znajdować w tej pętli bezpośrednio lub pośrednio.

4. **Dodanie warunku instrukcji do kontekstów pętli** (blok 6-7). W punkcie tym zastosowana jest teoria przedstawiona w rozdziale 3.16.1. W zależności od instrukcji z jaką mamy do czynienia, oraz czy odnosi się ona do pętli w której się bezpośrednio znajduje, czy też nie, modyfikujemy *warunki wykonania iteracji* oraz *wykonania części dalszej* odpowiednich pętli. Tutaj znajduje swoje odbicie różnica w funkcjonowaniu między instrukcją *next* oraz *exit*. Warunki są w postaci ciągu leksemów.
5. **Stworzenie bloku *if***. Następnym krokiem jest uzależnienie wykonania kodu znajdującego się za instrukcją *next* (*exit*) od ich warunku. Należy wygenerować instrukcję *if* o jednej gałęzi i warunku aktywacji będącym negacją warunku aktywacji instrukcji *next* (*exit*) według wzorów 3.9 lub 3.10. Następnie wywoływana jest funkcja konwersji pętli.

3.17. Możliwości optymalizacji algorytmów kompilacji instrukcji *for*

Rozdział ten przedstawia doświadczenia implementacyjne autora i stanowi wskazówkę dla ewentualnych naśladowców odnośnie tego, jak zwiększyć szybkość działania narzędzia do syntezy logicznej wykorzystując prezentowane algorytmy. Mimo iż algorytm określa w znacznym stopniu parametry czasowe realizujących go programów, to nie należy lekceważyć jakości implementacji. Podjęcie właściwych decyzji na tym etapie może wpłynąć znacznie na końcowe wyniki.

Po dokonaniu implementacji przedstawionych algorytmów kompilacji pętli *for* przystąpiono do fazy testowania. Analiza wydajności wykazała, że najwięcej czasu zabiera generacja równań dla *warunków wykonania iteracji* i *warunków wykonania części dalszej*. Powstają one ze złożenia zanegowanych warunków instrukcji *next* i *exit*.

Przykład 3.15 przedstawia zamianę pętli *for* na formę liniową zgodnie z algorytmem. Pewne fragmenty kodu zostały wyróżnione. Są to zanegowane warunki aktywacji instrukcji *exit* z pierwszej iteracji. Jak widać powtarzają się aż trzy razy. Ta sama sytuacja dotyczy warunku z drugiej iteracji. On z kolei powtarza się dwa razy. Jest to zgodne z zasadami przekładu, które głoszą, że aby dana iteracja mogła się wykonać w żadnej z iteracji ją poprzedzających nie może nastąpić przerwanie pętli. Powoduje to, że każda następna iteracja, ma dłuższe warunki aktywacji, a dodatkowo warunek ów składa się z elementów, które wystąpiły już wcześniej.

Po zidentyfikowaniu problemu, należy zadać sobie pytanie, czy istnieje jego rozwiązanie? W tym przypadku problemem jest to, że kompilacja instrukcji *for* zamienia tylko tą pętle na postać liniową, a nie generuje równań boolowskich. Natomiast aby uzyskać efekt optymalizacyjny, należy takie równania generować właśnie podczas usuwania pętli *for*. Konieczne jest zatem zmodyfikowanie algorytmów tak aby dostosować je do nowych warunków. Praktycznie realizowane jest to następująco. W algorytmie konwersji *next* i *exit* wprowadzone zostały zmiany, polegające na tym, że dla warunków aktywacji obu tych instrukcji generowane są równania boolowskie. Następnie równania te są dodawane do kontekstów pętli bloki (6 i 7 rysunek 3.14).

Z każdą sztucznie tworzoną instrukcją *if*, obojętnie czy obejmuje ona kod znajdujący się za instrukcjami *next* (*exit*), czy też cały blok iteracji, skojarzona została informacja semantyczna, zawierająca wygenerowane równanie jej aktywacji. Dzięki temu czas kompilacji dla niektórych testów zmniejszył się dziesięciokrotnie.

```
for i in 1 to 3 loop
        e(i) := f(i);
        exit when a(i)=b(i);
        c(i) := d(i);
end loop;

e(1) := f(1);                          --(*1)
if not (a(1)=b(1)) then
        c(1) := d(1);
end if;

if not x(a(1)=b(1)) then                    --(*2)
        e(2) := f(2);
        if not (a(2)=b(2)) then
                c(2) := d(2);
        end if;
end if;
if not (a(1)=b(1)) and (not (a(2)=b(2))) then --(*3)
        e(3) := f(3);
        if not (a(3)=b(3)) then
                c(3) := d(3);
        end if;
end if;
```

Przykład 3.15. Przykład ilustrujący zasady optymalizacji przekładu instrukcji *for*.

3.18. Podsumowanie

Rozdział ten zawiera opis autorskich algorytmów przekładu instrukcji sekwencyjnych języka VHDL. Cała kompilacja sterowana jest przez główny algorytm do którego „doczepione" są algorytmy odpowiedzialne za generację równań dla poszczególnych instrukcji. Prezentacja algorytmów została poprzedzona dokładnym wyjaśnieniem sposobu w jaki widziana jest dana instrukcja języka VHDL od strony syntezy. Omówione zostały wszystkie związane z tym problemy i ograniczenia, jakie muszą być narzucone w niektórych przypadkach. Wskazane zostały także te punktu, w których działanie algorytmów można jeszcze poprawić. Dzięki wiedzy z zdobytej podczas procedury testowej udało się ustalić, które operacje zajmują najwięcej czasu i przeprojektować je.

4. Weryfikacja przedstawionych algorytmów

Zadaniem niniejszego rozdziału jest przedstawienie dowodów na to, że prezentowane algorytmy pozwalają uzyskać poprawne wyniki i spełniają wymagania zapisane we wstępie rozprawy (rozdział 1.2). Ze względu na charakter zagadnienia, jedyna możliwa do zastosowania metoda polegała na wykonaniu praktycznej implementacji algorytmów, a następnie poddaniu jej procesowi testowania. Implementacja taka powstała w ramach projektu kompilatora opisanego w [17].

Ponieważ tworzony kompilator miał mieć funkcjonalność produktu komercyjnego, procedura testowa była złożona. Zbiór przykładów testowych wynosił ponad trzy tysiące plików, pogrupowanych według mechanizmów języka VHDL, których skuteczność działania miały za zadanie sprawdzić. Następnym krokiem było sprawdzenie jak kompilator radzi sobie z dużymi projektami przemysłowymi. W tym celu firma ALDEC dostarczyła zbiór komercyjnych projektów tzw. *IP Cores*.

4.1. Elementy składowe tworzonego kompilatora

Budowa typowego kompilatora przeznaczonego do syntezy logicznej została omówiona w rozdziale 2.6. Narzędzie tworzone na Wydziale Informatyki Politechniki Szczecińskiej, nie odbiega zasadniczo od tego schematu. Pewne różnice jednak występują. Ma to związek z odmiennym od dotychczas stosowanych formatem wyjściowym. Podstawowe podsystemy naszego narzędzia to:
— analizator leksykalny,
— analizator syntaktyczny,
— analizator semantyczny,
— generator równań boolowskich,
— postprocesor.

Analizator leksykalny. Zadaniem analizatora leksykalnego[25] jest zamiana pliku źródłowego VHDL na ciąg leksemów, czyli słów kluczowych, operatorów, identyfikatorów, stałych etc.). Każdy kategoria leksemów ma inny numer.

Analizator syntaktyczny. Następnym krokiem jest sprawdzenie źródła pod względem poprawności syntaktycznej. Analizator syntaktyczny[25] czyta ciąg leksemów wygenerowany przez analizator leksykalny i bada czy zdania, które tworzą należą do zdań gramatyki języka VHDL. Na tym etapie sprawdza się także czy zdanie te spełniają ograniczenia syntezy logicznej (częściowo).

Analizator semantyczny. Kolejny etap to analiza semantyczna[19]. Oprócz sprawdzenia poprawności źródła pod kątem semantyki, analizator tworzy dodatkową i niezbędną informację skojarzoną z leksemami. Ta informacja to między innymi:

— typ danych,
— nazwa identyfikatora,
— rozmiar w przypadki tablic.

Generator równań boolowskich. Jest to główny moduł kompilatora - odpowiedzialny za generację równań boolowskich. Dane wejściowe stanowi ciąg leksemów, odczytywany z pliku wraz ze stosowną informacją semantyczną. Generator rozpoznaje poszczególne instrukcje języka VHDL i wywołuje stosowne podprogramy. Wyjściem generatora jest plik zawierający równania boolowskie. Jest to najbardziej złożona część kompilatora. Jej znaczną część stanowi implementacja przedstawionych w niniejszej pracy algorytmów.

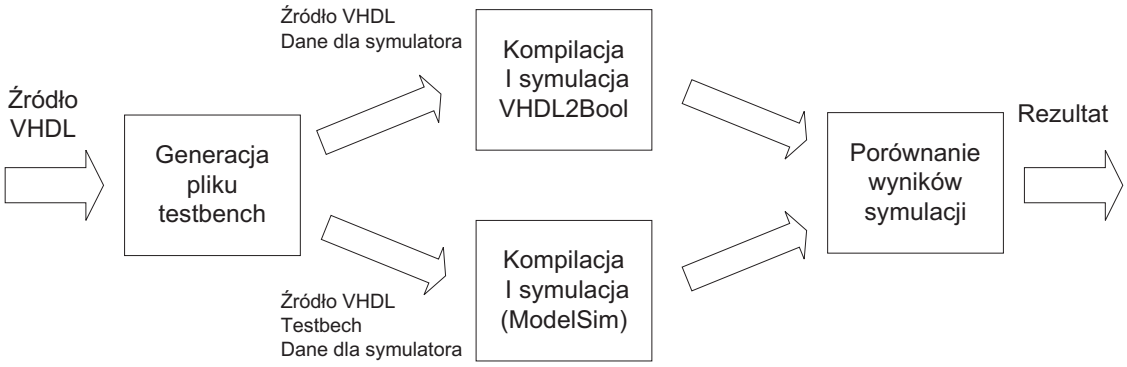
Postprocesor. Ostatnim elementem kompilatora jest tak zwany postprocesor[47] [48]. Wykonuje on pewne operacje, których nie mógł zrealizować generator albo z powodu braku na etapie swojej pracy pewnych informacji, albo też ze względu na trudności implementacyjne i nieoptymalność takiego rozwiązania. Do zadań postprocesora należy:
— korygowanie równań dla logiki sekwencyjnej,
— korygowanie równań dla logiki synchronicznie-asynchronicznej,
— połączenie równań o takich samych lewych stronach reprezentowanych nazwami sygnałów typu resolved,
— minimalizacja równań boolowskich,
— eliminacja zmiennych roboczych w uzasadnionych przypadkach,
— wyszukiwanie dwóch lub więcej sygnałów o takiej samej nazwie po lewej stronie równań i nie zadeklarowanych jako *resolved*: sygnalizowanie błędu w takim przypadku.

Wejściem postprocesora jest plik z równaniami boolowskimi oraz pliki tworzone przez analizator semantyczny i generator. Na podstawie tych informacji postprocesor realizuje swoje zadania tworząc w pełni poprawny plik wynikowy zawierający poprawione równania.

4.2. Weryfikacja poprawności rozwiązania

Ze względu na dużą ilość testów do wykonania, procedura testowa została zautomatyzowana. Na rysunku 4.1 pokazany jest jej przebieg. Składała się z następujących etapów:

1. Generacja pliku *testbench*. Każdy utworzony plik *testbench* ma nazwę taką jak oryginalne źródło VHDL tylko zawierają dodatkowy przyrostek „_tb". Oprócz tego program generujący tego typu pliki, tworzy kilka dodatkowych plików, których format i przeznaczenie jest opisane w rozdziale 4.2.1.
2. Kompilacja i symulacja - ModelSim. W kroku tym uruchamiany jest pakiet ModelSim w trybie wsadowym. Dokonuje on kompilacji i symulacji źródła VHDL razem z plikiem *testbench*. Symulator korzysta ze specjalnie przygotowanego pliku zawierającego wartości sygnałów i zmiennych. Na wyjściu otrzymujemy plik zawierający wyniki symulacji. Za jego stworzenie odpowiedzialny jest kod programu *testbench*.
3. Kompilacja i symulacja - VHDL2Bool. W odróżnieniu od *ModelSima* to narzędzie nie ma zintegrowanego symulatora. Najpierw, więc program źródłowy poddany



Rysunek 4.1. Przebieg procedury testowej. Źródło: opracowanie własne.

jest kompilacji, a dopiero później symulacji, przy pomocy oddzielnego programu. Należy zaznaczyć, że nie korzystamy z pliku *testbench*. Niezbędne dane dla symulatora równań boolowskich zapisane są w pliku o odpowiednim formacie. Symulator uruchamiany jest z następującymi parametrami:
— nazwą pliku zawierającego równania boolowskie,
— nazwą pliku zawierającego dane niezbędne dla symulacji,
— nazwą pliku, do którego zostaną zapisane wyniki symulacji,
— nazwą pliku, do którego zapisane zostaną ewentualne błędy występujące podczas symulacji.

4. Porównanie wyników symulacji. Jest to ostatni etap. Rezultaty otrzymane z dwóch różnych narzędzi są ze sobą porównywane. Wcześniej jednak należy przekształcić plik wyjściowy uzyskany z programu ModelSim do postaci takiej, jaką generuje symulator równań boolowskich. Wprawdzie istnieje możliwość napisania przykładu *testbench*, który tworzyłby plik w odpowiednim formacie od razu, ale byłoby to dosyć kłopotliwe w związku z ograniczeniami języka VHDL w zakresie operacji na plikach i łańcuchach znaków.

4.2.1. Składniki środowiska testowego

Zautomatyzowane środowisko testowe zawierało następujące elementy:
— skrypt sterujący,
— zbiór testów,
— zbiór plików testbench,
— źródło poprawnych wyników symulacji,
— symulator równań boolowskich,
— narzędzie porównujące wyniki.

Skrypt sterujący. Jest to krótki program wsadowy napisany w powłoce (ang. *shell*) *bash*. Jako parametr wywołania podaje się nazwę pliku źródłowego VHDL, który ma być testowany. Skrypt wywołuje kolejno programy dokonujące testowania. Przed rozpoczęciem testowania, przygotowuje środowisko kasując pliki powstałe w poprzednim cyklu. Skrypt wymaga do poprawnego działania, aby wszelkie programy narzędziowe jak np. *sed* czy ModelSim znajdowały się w systemowej ścieżce przeszukiwania.

Zbiór testów. Większość testów stanowiły oryginalne przykłady dostarczone przez firmę ALDEC. Użyte zostały podzbiory dotyczące instrukcji *if*, *case* oraz *for*. Są to bardzo proste źródła, mające za zadanie weryfikację wyżej wymienionych instrukcji w różnych, ale generalnie typowych przypadkach. Poszczególne testy z danej grupy są do siebie bardzo podobne. Przy ich pomocy sprawdzane są następujące aspekty poprawności:
— poprawność przekładu dla wszelkich dopuszczalnych postaci danej instrukcji,
— poprawność przekładu w zależności od typów danych,
— wykrywanie sytuacji błędnych i odpowiednia na nie reakcja.

Zbiór przykładów *testbench* Aby można było przeprowadzić symulację programu w języku VHDL, należy nadać wartości poszczególnym sygnałom i zmiennym. Można tego dokonać na dwa sposoby:
— tworząc tak zwane „waveforms", czyli przebiegi czasowe wartości sygnału, zmiennej,
— generując plik *testbench*, dla danego testu.
Drugie rozwiązanie jest efektywniejsze. Plik *testbench* jest łatwiejszy do analizy. Zadaniem programu *testbench* jest zmiana wartości sygnałów (zmiennych) badanego przykładu, zgodnie z ustalonym porządkiem. Może on być napisany w VHDL, co jest to bardzo częstym przypadkiem. Ale też może to być zupełnie inny język np. C, Tcl, Python, lub też metajęzyk symulatora. W badaniach prezentowanych w niniejszej pracy wykorzystano do tego celu język VHDL. Testy były tworzone ręcznie lecz automatycznie, przy pomocy prostego, autorskiego generatora plików *testbench*[1]. Generator dokonuje analizy kodu i na ich podstawie generuje następujące informacje:
— plik *testbench*,
— plik z danymi dla symulatora ModelSim,
— skrypt sterujący dla symulatora ModelSim,
— plik z danymi dla symulatora równań boolowskich (VHDL2Bool).
Postać pliku *testbench* wymaga omówienia. Jest on tworzony według pewnego, określonego wzoru. Jego zadania są następujące:
— wczytywanie wartości sygnałów (zmiennych) wejściowych w kolejnych krokach symulacji z pliku,
— zapisywanie rezultatów symulacji do pliku.
Ze względu na wspomnianą wcześniej ułomność operacji wejścia-wyjścia w języku VHDL, pliki wejściowy i wyjściowy korzystają z bardzo uproszczonych formatów.

Generator przykładów *testbench*, składa się z dwóch części, które stanowią oddzielne programy:
— Parsera - którego zadaniem jest wyłuskanie ze źródła sygnałów i zmiennych które stanowią wejście albo wyjście układu. Oczywiście dołączana jest niezbędna informacja semantyczna, czyli typ, rozmiar itp. Parser został napisany w języku C/C++ przy pomocy dobrze znanych narzędzi wspomagających tworzenie analizatorów leksykalnych i syntaktycznych *flex* i *bison*.
— Generatora - na podstawie informacji otrzymanych od parsera tworzy on wszystkie opisane wcześniej pliki. Do stworzenia tego programu wykorzystano język C#.

[1] Problem tworzenia plików testowych, ze względu na swoją złożoność może stanowić przedmiot zupełnie oddzielnych badań.

Źródło poprawnych wyników symulacji Weryfikacja odbywa się poprzez porównanie wyników działania testowanego narzędzia z rezultatami otrzymanymi z referencyjnego produktu. Ważne było, aby wzorcowy kompilator generował poprawne wyniki. Konieczne więc było zastosowanie sprawdzonego pakietu komercyjnego, takiego jak ModelSim firmy Mentor Graphics (wersja 5.7d). Ponieważ testowanie odbywało się w sposób automatyczny, wykorzystano tryb wsadowy tego symulatora. W trybie tym symulator wykonuje komendy zapisane w pliku.

Symulator równań boolowskich Program dokonujący symulacji równań boolowskich. Dzięki niemu możliwe jest sprawdzenie czy równania wygenerowane przez kompilator reprezentują poprawnie działający układ scalony. Chodzi tu zarówno o testowanie samego kompilatora, jak i weryfikacje dokonanej za jego pomocą syntezy. Narzędzie to[68] zostało opracowane na potrzeby projektu opisanego w [17]. Do swojego działania symulator potrzebuje następujących informacji:
— plik z równaniami boolowskimi, które mają być symulowane,
— plik zawierający wartości sygnałów (zmiennych), w każdym kroku symulacji.

Narzędzie porównujące wyniki Jest to prosty program, którego zadaniem jest stwierdzenie czy wyniki otrzymane z symulatora równań boolowskich, oraz pakietu ModelSim są tożsame. Wejście stanowią pliki zawierające rezultaty obu symulacji. Program został napisany w języku C#.

Programy pomocnicze i narzędzia Oprócz wymienionych wcześniej programów, aby procedura testowa mogła być w pełni zautomatyzowana, należało użyć kilka narzędzi pomocniczych. Najważniejszym z nich było środowisko Cygwin. Jest to zestaw programów świetnie znanych użytkownikom systemów Unix i kompatybilnych, dostępny na platformę Windows. Podczas testowania lub tworzenia programów wspomagających testowanie użyto następujących elementów tego pakietu:
— powłoka *bash*,
— edytor strumieniowy - *sed*,
— generator analizatorów leksykalnych - *flex*,
— generator analizatorów syntaktycznych - *bison*,
— kompilator C/C++ - *gcc*.
Programy napisane w języku C# powstały przy użyciu pakiety Borland C# Builder, oraz Visual Studio .NET firmy Microsoft.

4.2.2. Wyniki

Poddanie tworzonego narzędzia wyczerpującej procedurze testowej przedstawionej powyżej pozwoliło uzyskać odpowiedź na pytanie, czy zaprezentowane rozwiązanie działa poprawnie czy też nie. Wyniki testów wskazują jednoznacznie, że tak. Kompilator, którego części odpowiedzialne za przekład instrukcji *if*, *case* oraz *for* zostały zaimplementowane według opisanych w rozdziale 3 algorytmów funkcjonuje prawidłowo. Wyniki, które otrzymano z symulatora równań boolowskich, są tożsame z tymi uzyskanymi z referencyjnego narzędzia komercyjnego - ModelSim.

4.3. Ocena przydatności praktycznej rozwiązania

Przeznaczeniem narzędzia tworzonego na Wydziale Informatyki Politechniki Szczecińskiej miało być zastosowanie przemysłowe. Kompilator taki oprócz oczywiście stu procentowej poprawności przekładu musi charakteryzować się jeszcze dobrymi parametrami wydajnościowymi takimi jak:
— czas kompilacji,
— wykorzystanie zasobów systemowych - pamięci.
Jeżeli powyższe wielkości będą zbyt duże to takie narzędzie nie będzie nadawało się do praktycznego zastosowania w przemyśle. Konieczne jest zatem zmierzenie obu wielkości. Ze względu na zakres tematyczny pracy, pomiary czasu kompilacji i zużycia pamięci dotyczyły tylko generatora równań boolowskich.

4.3.1. Procedura testowa

Środowisko testowe różni się nieznacznie od tego użytego do sprawdzenia poprawności. Konieczne było zmodyfikowanie kodu kompilatora tak, aby można było wykonać pomiary czasu kompilacji i zużycia pamięci w sposób możliwie jak najdokładniejszy. O ile pomiar pierwszej wielkości nie stanowił problemu, to zmierzenie zapotrzebowania na pamięć wymagało wykorzystania zaawansowanych mechanizmów śledzenia programu środowiska Windows. Pomiar zajętości pamięci wprowadził spory narzut czasowy, aby zatem zmierzony czas kompilacji odpowiadał rzeczywistemu, należało testy przeprowadzić oddzielnie dla obu wielkości.

4.3.2. Zbiór testów

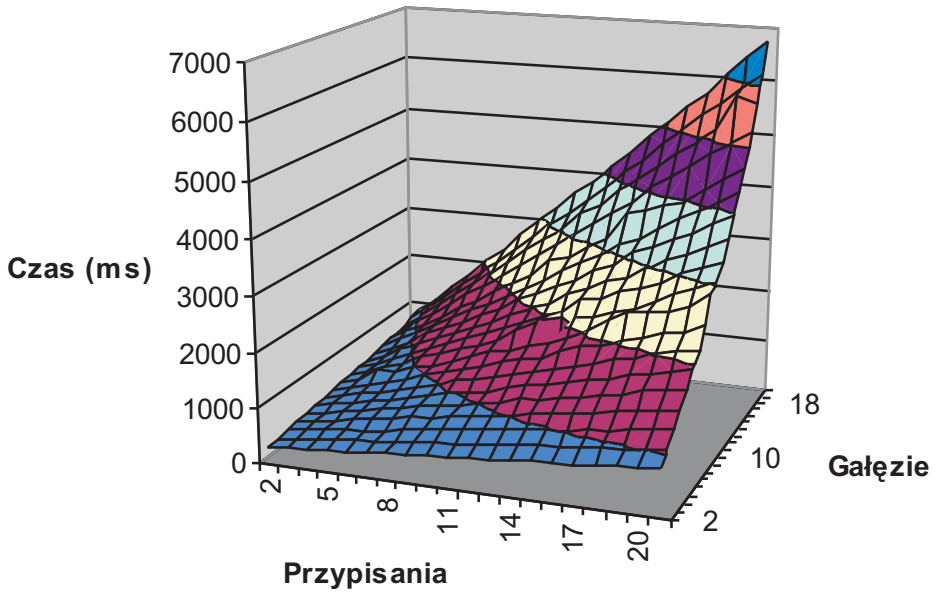
Ocena właściwości wydajnościowych wymagała przygotowania specjalnych testów. Testy te nie są specjalnie skomplikowane. Ich zadaniem jest pokazanie, jak zachowuje się kompilator podczas przekładu różnych postaci instrukcji *if*, *case* oraz *for*. Testy te, wraz z programem użytym do ich generacji znajdują się na dołączonej do pracy płycie CD (dodatek B).

4.3.3. Wyniki

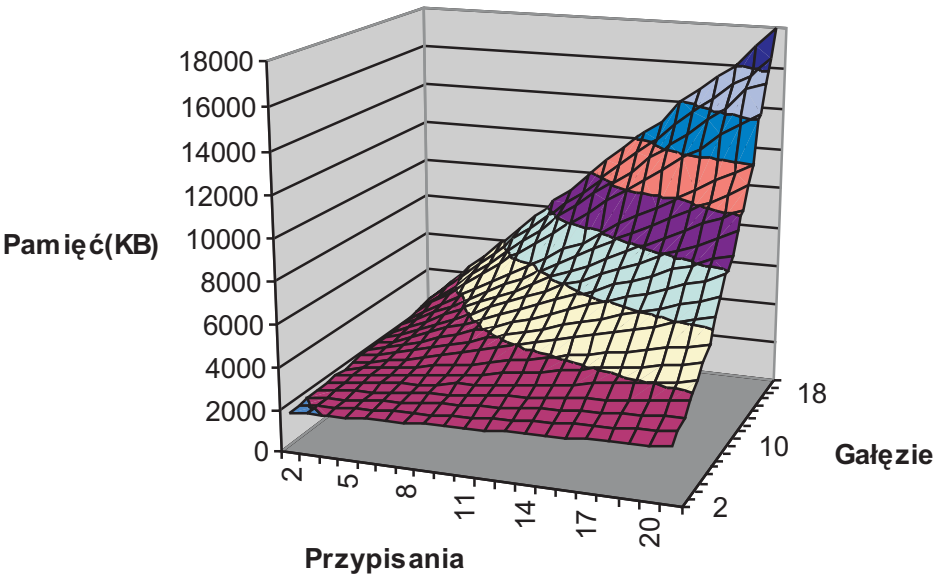
Na wykresach 4.2 i 4.3 przedstawione charakterystyki czasu kompilacji oraz zużycia pamięci uzyskane dla instrukcji *if*. Parametrami użytymi do wyznaczenia obu charakterystyk są: liczba gałęzi instrukcji *if* oraz liczba instrukcji przypisania pojedynczej gałęzi. Widać, ze obie mierzone wielkości rosną wprost proporcjonalnie do wzrostu parametrów.

Wykresy 4.4 oraz 4.5 zawierają takie same charakterystyki, ale dla instrukcji *case*. Tutaj także uzyskano liniową zależność między parametrami, a mierzonymi wielkościami.

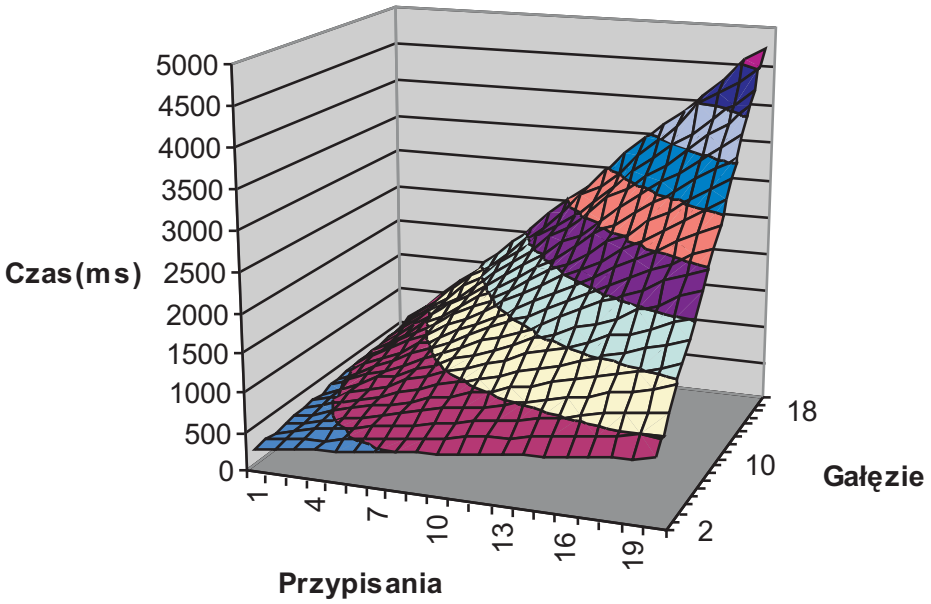
Ostatnie dwa wykresy 4.6 i 4.7 dotyczą pętli *for*. W tym przypadku parametrami charakterystyk są: ilość iteracji pętli i ilość instrukcji przypisania w pętli. Także i dla tej instrukcji czas kompilacji i zajętość pamięci rośnie liniowo wraz ze wzrostem wartości parametrów.



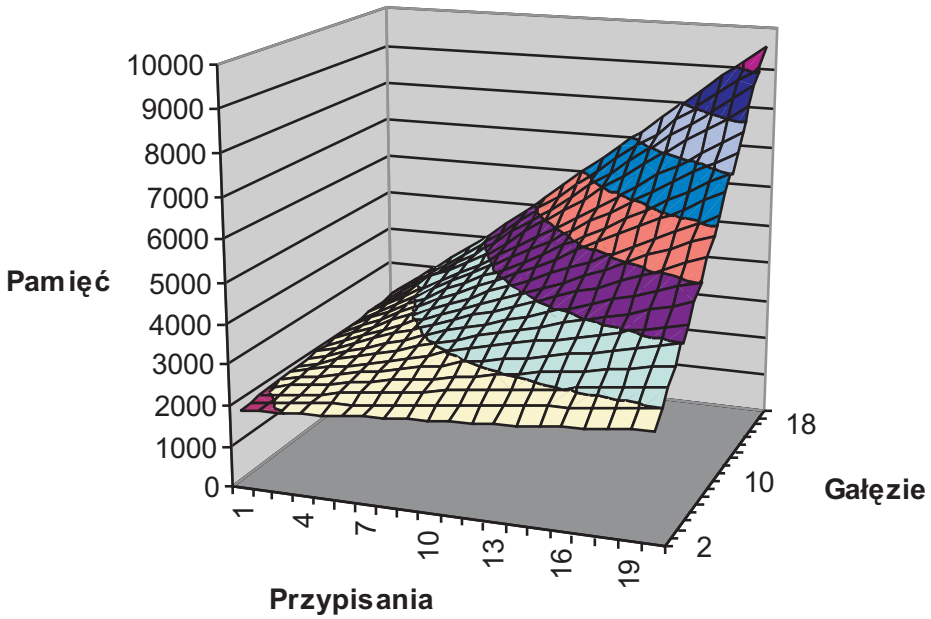
Rysunek 4.2. Zależność czasu kompilacji od ilości gałęzi instrukcji *if*, oraz liczby przypisań. Źródło: opracowanie własne.



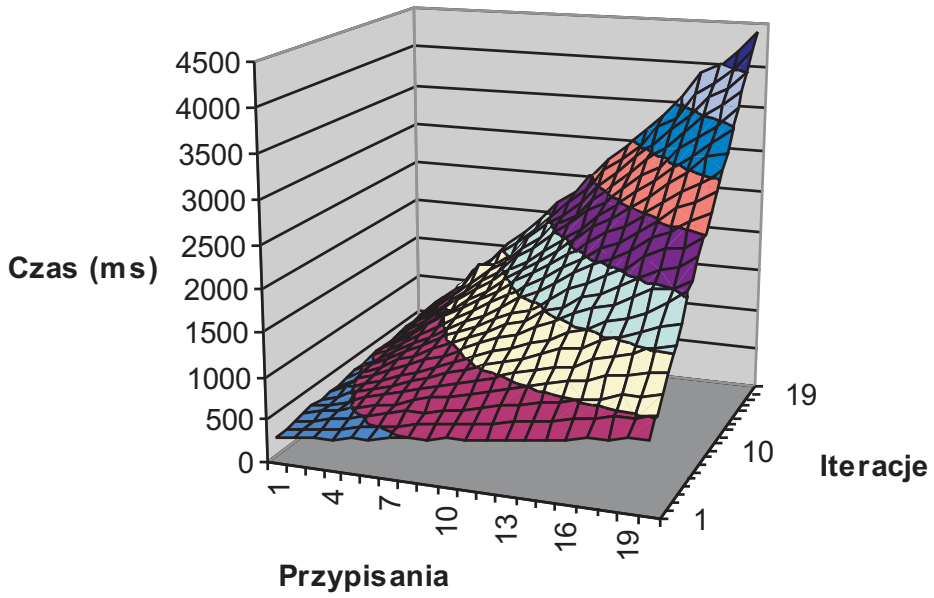
Rysunek 4.3. Zależność zużycia pamięci podczas kompilacji od ilości gałęzi instrukcji *if*, oraz liczby przypisań. Źródło: opracowanie własne.



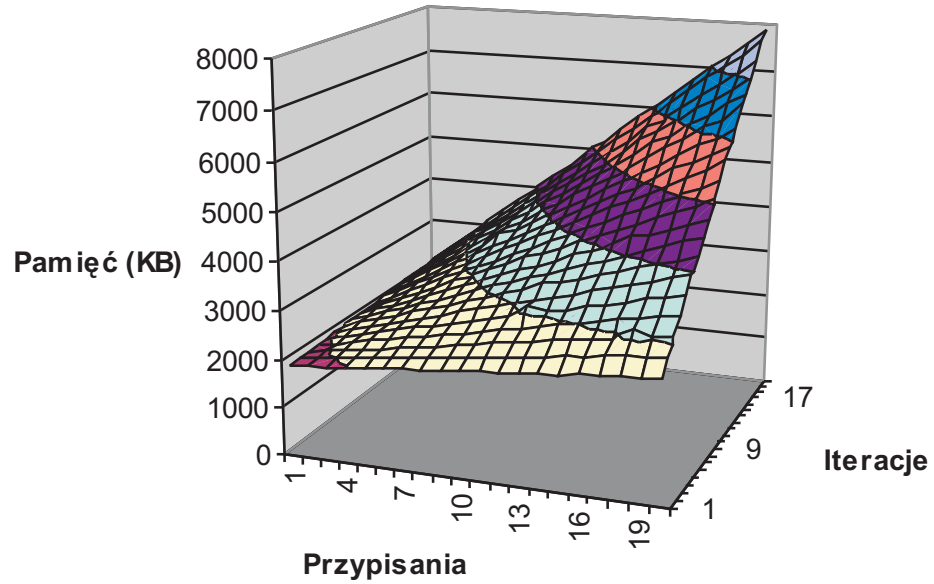
Rysunek 4.4. Zależność czasu kompilacji od ilości gałęzi instrukcji *case*, oraz liczby przypisań. Źródło: opracowanie własne.



Rysunek 4.5. Zależność zużycia pamięci podczas kompilacji od ilości gałęzi instrukcji *case*, oraz liczby przypisań. Źródło: opracowanie własne.



Rysunek 4.6. Zależność czasu kompilacji od ilości iteracji pętli *for*, oraz liczby przypisań. Źródło: opracowanie własne.



Rysunek 4.7. Zależność zużycia pamięci podczas kompilacji od ilości iteracji pętli *for*, oraz liczby przypisań. Źródło: opracowanie własne.

Wykazana w powyższych testach, liniowa zależność między ilością danych wejściowych, a wykorzystaniem zasobów systemowych podczas kompilacji, pozwala na zastosowanie proponowanych algorytmów w profesjonalnych kompilatorach.

4.4. Generacja równań dla przykładu przemysłowego

Aby ostatecznie potwierdzić prawidłowość przedstawionych algorytmów, konieczne było wykazanie, iż potrafią one poradzić sobie z rzeczywistym, dobrze znanym problemem, który z racji swojej złożoności obliczeniowej, jest dobrym kandydatem do implementacji sprzętowej. Takim zagadnieniem testowym może być:
— transformata Fouriera,
— algorytmy przetwarzania obrazów,
— algorytmy kryptograficzne,
— popularny procesor,
— układ komunikacyjny, lub sprzęgający oraz wiele innych.

4.4.1. Wybór problemu testowego

Wybranie właściwego zagadnienia testowego, nie jest zadaniem prostym, mimo teoretycznie wielu możliwości. Istotnym ograniczeniem jest temat prezentowanej pracy. Ponieważ dotyczy ona logiki kombinacyjnej, test powinien przede wszystkim dotyczyć tych zagadnień, które się z tym typem logiki wiążą. I tu pojawił się problem, bo trudno sobie wyobrazić jakiś bardziej skomplikowany projekt układu logicznego zrealizowany całkowicie bez użycia logiki sekwencyjnej. Nawet, jeśli uda się zrealizować sam algorytm, to przecież nie działa on w próżni, potrzebne jest dostarczenie danych wejściowych i wyprowadzenie wyników. Raczej trudno wyobrazić sobie dokonanie tego, bez choćby szczątków logiki sterującej, np. przerzutników, umożliwiających synchronizację ze środowiskiem testowym.

Innym ograniczeniem jest sama procedura weryfikacji. Znacznie łatwiej jest weryfikować nawet bardzo skomplikowaną implementację jakiegoś algorytmu obliczeniowego, niż procesora czy układu komunikacyjnego. Wprawdzie i tak należy napisać przykład testujący (ang. *testbench*), ale w drugim przypadku będzie to o kilka poziomów prostsze. Podsumowując zagadnienie, realizacja którego powinna posłużyć do weryfikacji opisywanych w niniejszej pracy algorytmów powinno spełniać następujące kryteria:
— być problemem/algorytmem przede wszystkim obliczeniowym,
— nie powinien wymagać pamięci do swojego działania, takiej którą trzeba by było wypełnić przed rozpoczęciem obliczeń, danymi wejściowymi, a po ich zakończeniu przesłać jej zawartość do innego podsystemu,
— pracować na strumieniu niezależnych od siebie danych wejściowych.
Po dokonaniu rozeznania udało się znaleźć algorytmy spełniające podane wyżej ograniczenia: operacje dodawania, mnożenia, oraz dzielenia zmiennoprzecinkowego[51] dla liczb pojedynczej precyzji.

4.4.2. Reprezentacja zmiennoprzecinkowa

Format zmiennoprzecinkowy służy do przechowywania liczb zawierających część ułamkową. Jej cechą szczególną jest ograniczona precyzja[32], wyrażona ilością cyfr,

niewspółmiernie mała w stosunku do zakresu możliwych do przechowania wartości. Istnieje kilka standardów przechowywania wartości zmiennoprzecinkowym, ale najszerzej implementowanym jest standard IEEE o numerze 754[38]. Określa on dokładnie rodzaje i postacie liczb zmiennoprzecinkowych, omawia zagadnienia dotyczące wykonywania na nich operacji arytmetycznych, a także dopuszczalne sposoby zaokrąglania wartości. Standard ten bazuje na formacie binarnym, co ułatwia implementacje w powszechnie stosowanych komputerach. Ogólna postać liczby zapisanej w ten sposób wyrażona jest następującym wzorem 4.1:

$$(-1)^s \cdot 2^{E-bias} \cdot (b_0 \cdot b_1 \cdot b_2 \cdot \ldots \cdot b_{p-1}) \tag{4.1}$$

gdzie:
— p - liczba znaczących bitów - określa precyzję liczby,
— E_min - najmniejsza wartość wykładnika,
— E_max - największa wartość wykładnika,
— s - 0 lub 1, określa znak liczby,
— E - wartość wykładnika z przedziału $< E_{min}; E_{max} >$,
— b_i - 0 lub 1, wartość danego bitu mantysy liczby,
— $bias$ - stała o jaką powiększony jest wykładnik.

Istnieje także standard IEEE-854[41], który dopuszcza inne systemy liczbowe prócz binarnego. Stanowi on uogólnienie w stosunku do standardu IEEE-754.

Standard IEEE-754 definiuje dwie klasy liczb zmiennoprzecinkowych
— pojedynczej precyzji (ang. single),
— podwójnej precyzji (ang. double).
Oto parametry obu tych klas:

Parametr	Formaty	
	Pojedynczy (single)	Podwójny (double)
s	1	1
p	24	53
E_{max}	+127	+1023
E_{min}	−126	−1022
$bias$	+127	+1023
Ilość bitów wykładnika	8	11
Rozmiar całej liczby w bitach	32	64

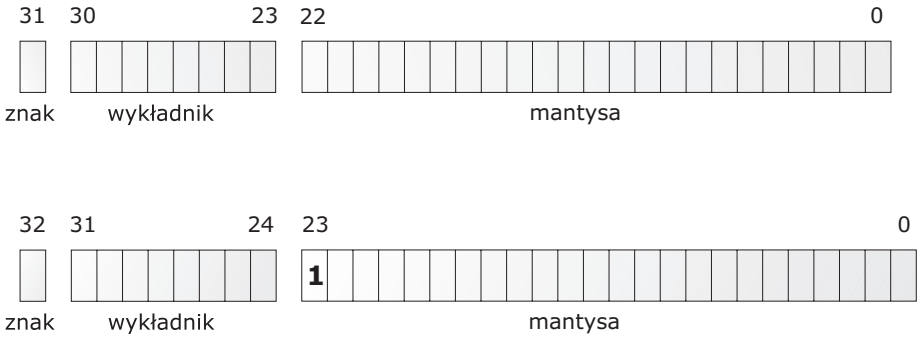
Tabela 4.1. Liczby zmiennoprzecinkowe według standardu IEEE-754. Źródło: [38].

4.4.3. Postać liczby pojedynczej precyzji

Ponieważ testy odnoszą się tylko do jednej klasy liczb, opis zostanie ograniczony właśnie do niej. Ze wzoru 4.1 oraz tabeli 4.1 wynika, że liczba taka składa się z:
— jednego bitu znaku,
— ośmiu bitów wykładnika,
— dwudziestu czterech bitów mantysy.
Jednocześnie jednak jej całkowity rozmiar wynosi trzydzieści dwa bity, czyli o jeden mniej niż wynosi suma powyższych elementów. Spowodowane jest to tym, że liczba zmiennoprzecinkowa przechowywana jest w postaci znormalizowanej, a to oznacza,

że wartość jej mantysy należy do przedziału $[1, 2)$, co w wydaniu binarnym przybiera postać:$1.XXXXX\ldots$.

Widać, że jedynka występuje zawsze, więc można ją pominąć przy zapisie, jest to tak zwany *ukryty bit* (*ang. hidden bit*). Rysunek 4.8 przedstawia w sposób poglądowy liczbę zmiennoprzecinkową w postaci typowej, oraz z wyszczególnionym *ukrytym bitem*.



Rysunek 4.8. Budowa liczby zmiennoprzecinkowej. Źródło: [38].

4.4.4. Specjalne wartości

Pewne wartości liczby zmiennoprzecinkowej są zarezerwowana i mają specjalne znaczenie. Taka sytuacja dotyczy chociażby liczby zero. Nie można jej zapisać w postaci zmiennoprzecinkowej bezpośrednio, ponieważ mantysa zawsze należy do przedziału od jednego do dwóch. Konieczne było zatem arbitralne wyznaczenie w standardzie IEEE-754 wartości, która to zero będzie reprezentować. W sumie arytmetyka zmiennoprzecinkowa używa trzech specjalnych wartości liczbowych, które reprezentują wartości inne, niż wynikałoby to z analizy wcześniejszej teorii:

— zero, gdy $E = 0$ i $M = 0$;
— $+/-$ nieskończoność (ang. infinity), gdy $E = 255$ i $M = 0$; pojawia się, gdy otrzymany wynik wykracza poza dopuszczalny zakres wartości liczby zmiennoprzecinkowej,
— NaN (ang. Not a Number), gdy $E = 255$ i $M <> 0$; powstaje, gdy operacja nie może zostać przeprowadzona na podanych argumentach, lub z jakiegoś innego powodu, jest to oznaczenie błędu.

4.4.5. Operacje zmiennoprzecinkowe

Dla dwóch liczb zmiennoprzecinkowych $x_1 = 2^E \cdot M_1$ oraz $x_1 = 2^E \cdot M_2$ gdzie E oraz M oznaczają wykładnik oraz mantysę operację arytmetyczne definiuje się następująco:

$$x_1 + x_2 = (M_1 \cdot 2^{E_1 - E_2} + M_2) \cdot 2^{E_2} \tag{4.2}$$
$$x_1 + x_2 = (M_1 \cdot 2^{E_1 - E_2} + M_2) \cdot 2^{E_2} \tag{4.3}$$
$$x_1 \cdot x_2 = (M_1 \cdot M_2) \cdot 2^{E_1 + E_2} \tag{4.4}$$
$$x_1 / x_2 = (M_1 \cdot M_2) \cdot 2^{E_1 - E_2} \tag{4.5}$$

Przy czym należy pamiętać o wykonywaniu operacji na wykładnikach, od których odjęto wartość przesunięcia (bias).

Aby można było wykonać dodawanie lub odejmowanie obie liczby muszą mieć taki sam wykładnik. Oznacza to, że liczbę o mniejszym wykładniku należy doprowadzić do postaci, w której ten wykładnik jest równy wykładnikowi drugiej liczby. Zwiększenie wykładnika o jeden oznacza podzielenie mantysy przez dwa. W przypadku dużej różnicy miedzy obiema liczbami może dojść do utraty precyzji.

Po wykonaniu operacji może zaistnieć potrzeba znormalizowania wyniku. Sprowadza się to do przesuwania mantysy i stosownej modyfikacji wykładnika.

Ostatnią czynnością będzie określenie znaku wyniku, w zależności od typu i wartości liczb biorących udział w operacji.

4.4.6. Zaokrąglanie

Z operacjami na liczbach zmiennoprzecinkowych wiążę niestety ograniczona dokładność. Wynika ona z postaci z relatywnie niewielkiej ilości cyfr używanej do reprezentowania dużych wartości liczbowych. Mantysa to tylko dwadzieścia cztery cyfry binarne. W związku z tym konieczne jest odpowiednie zaokrąglania liczb. Standard IEEE-754 dopuszcza cztery sposoby dokonywania takiej operacji:
— zaokrąglania do $+\infty$ (plus nieskończoność),
— zaokrąglania do $-\infty$ (minus nieskończoność),
— zaokrąglanie do zera,
— zaokrąglanie do najbliższej wartości (ang. round to nearest).
Pełna implementacja standardu 754 powinna obejmować wszystkie powyższe metody z zaznaczeniem, że domyślnym sposobem jest zaokrąglanie do najbliższej wartości. Aby skorzystać z pozostałych użytkownik musi zmienić ustawienia środowiska obliczeniowego.

4.4.7. Normalizacja

Jeżeli po operacji zmiennoprzecinkowej wartość mantysy liczby nie należy do przedziału $[1, 2)$, to konieczna jest normalizacja. Polega to na przesunięciu przecinka w lewo lub w prawo, przy okazji dokonując stosownej inkrementacji lub dekrementacji wykładnika.

4.4.8. Procedura testowa i otrzymane wyniki

Proces testowania składał się z dwóch etapów. Wykorzystane zostały w tym celu procedury przedstawione w rozdziałach 4.2 i 4.3. Po otrzymaniu równań boolowskich dokonano ich symulacji, a następnie przeprowadzono porównanie uzyskanych rezultatów z wynikami wzorcowymi. Kolejnym krokiem był pomiar czasu kompilacji oraz zużycia pamięci. Ostatnim elementem było oszacowanie rozmiaru utworzonego elementu poprzez policzenie liczba bramek logicznych (AND, OR, NOT) oraz przerzutników. Wyniki te prezentowane są w tabeli 4.2. Kod źródłowy zaimplementowanych operacji zmiennoprzecinkowych znajduje się na płycie CD dołączonej do niniejszej pracy (dodatek B).

Kolejnym krokiem było sprawdzenie jak z powyższymi źródłami VHDL radzą sobie narzędzia komercyjne. Testy zostały przeprowadzane z wykorzystaniem dwóch stosunkowo łatwo dostępnych pakietów:
— Xilinx ISE 6.2,
— Altera Quartus II 4.

Przykład testowy	Elementy logiczne					czas(s)	zużycie pamięci(MB)
	AND	OR	NOT	suma	przerzutniki		
dodawanie	17118	2025	13786	32929	32	13,83	6,45
mnożenie	49061	9761	39517	98339	32	50,30	40,22
dzielenie	68376	8896	50763	128035	32	297,89	70,73

Tabela 4.2. Wyniki kompilacji przykładów operacji zmiennoprzecinkowych. Źródło: opracowanie własne.

Wyniki te mają charakter poglądowy, gdyż ciężko jest dokonać ich porównania z wynikami otrzymanymi z narzędzia VHDL2bool, a nawet między sobą. Oba pakiety dokonują syntezy dla z góry określonej rodziny układów FPGA. Bloki logiczne takich układów potrafią realizować skomplikowane funkcję logiczne[105][8] co sprawia, że porównywanie ich z siecią bramek nie jest do końca wiarygodne i miarodajne. W przypadku produktu firmy Xilinx jest o tyle łatwiej, że jest dostępna ogólną ilość bramek układu co pozwala na przybliżone porównania wyników. Tabela 4.3 przedstawia otrzymane rezultaty. Układem, który został wybrany dla potrzeb syntezy był Spartan2E, przy czym dla dodawania i mnożenia wystarczyła wersja 100k bramek, dla dzielenia konieczny był układ większy (200k).

przykład testowy	ilość bramek	elementy logiczne			czas(s)	zużycie pamięci(MB)
		LUT	IOB	przerzutniki		
dodawanie	100k	626/2400	97/182	32/2400	11	68,43
mnożenie	100k	1072/2400	97/182	32/2400	14	72,66
dzielenie	200k	2775/4704	97/146	32/4704	39	104,40

Tabela 4.3. Wyniki syntezy wybranych układów arytmetyki zmiennoprzecinkowej za pomocą oprogramowania Xilinx ISE 6.2. Architektura docelowa: Spartan2E. Źródło: opracowanie własne oraz [105].

Tabela 4.4 prezentuje wyniki otrzymane z programu Quartus II. Układ wybrany do testów to Cyclone. Niestety firma nie podaje wielkości układu w bramkach.

przykład testowy	elementy logiczne			czas(s)	zużycie pamięci(MB)
	LUT	IOB	przerzutniki		
dodawanie	679/2910	98/104	32/2910	19	42,40
mnożenie	1172/2910	98/104	32/2910	24	43,50
dzielenie	2733/2910	98/104	32/2910	59	45,00

Tabela 4.4. Wyniki syntezy wybranych układów arytmetyki zmiennoprzecinkowej za pomocą oprogramowania Altera Quartus II 4. Architektura docelowa: Cyclone. Źródło: opracowanie własne oraz [8].

Wyniki testu wydajności programu VHDL2Bool (czas kompilacji i zużycie pamięci) pokazują, że prezentowane algorytmy posiadają efektywność wymaganą w rozwiązaniach przemysłowych. W przypadku zużycia pamięci widać wyraźną przewagę VHDL2Bool, zwłaszcza nad produktem firmy Xilinx. Wyjątkowo duży czas

kompilacji, jaki uzyskano dla przykładu zawierającego dzielenia zmiennoprzecinkowe, wynika z niedoskonałości implementacji zaprezentowanych w pracy algorytmów. Porównanie wielkości układów otrzymanych za pomocą wszystkich trzech narzędzi ma charakter przybliżony. Rozmiar układów Spartan2E jakie musiały zostać użyte podczas testów wskazuje jednak, że kompilator VHDL2bool i w tej dziedzinie osiąga rezultaty na poziomie produktów przemysłowych.

4.5. Złożoność obliczeniowa algorytmów

Oszacowanie złożoności obliczeniowej[75] dla algorytmów kompilatora jest niezwykle trudne, ze względu na wewnętrzną złożoność takiego programu. Szacowanie złożoności obliczeniowej ma na celu pokazanie jak zależy liczba operacji realizowanych przez program w stosunku do różnych ilości danych wejściowych. Powyższa krótka definicja sygnalizuje dwa problemy, które trzeba rozwiązać, aby przystąpić do szacowania złożoności obliczeniowej:
— określić precyzyjnie pojęcie operacji,
— zbadać granice zbioru danych wejściowych.

4.5.1. Operacja

Właściwe wybranie operacji, która będzie podstawą do szacowania jest bardzo ważne. Oto warunki jakie powinna spełniać taka operacja, aby można było użyć jej w procesie estymacji złożoności obliczeniowej:
— częstość jej wykonywania musi zależeć od danych wejściowych,
— powinna realizować działanie, będące istotą ocenianego algorytmu,
— powinna się cechować odpowiednio dużą granulacją, aby jej działanie nie wymagało zagłębiania się w implementacje.

Poszukując operacji, która zostanie użyta do oszacowania złożoności obliczeniowej, należy koncentrować się na tych fragmentach algorytmu, które związane są z przetwarzaniem danych wejściowych. Operacja użyta do szacowania powinna dotyczyć istoty ocenianego algorytmu, a nie stanowić jakiegoś działania pobocznego, wspomagającego, które może być realizowane różnie w zależności od implementacji. Wreszcie, musi mieć ona odpowiednio dużą granulacje. Jeżeli ma posłużyć do oceny algorytmu, to musi być widoczna na tym właśnie poziomie, a nie na poziomie implementacji. Implementacja jest zależna, od użytego języka, narzędzia, docelowego systemu operacyjnego, a także konkretnych wymagań stawianych projektowi. Te czynniki szacując złożoność obliczeniową należy wyeliminować.

4.5.2. Zbiór danych wejściowych

Kolejny problem to granice zbioru danych wejściowych. Idealnie byłoby gdyby był on skończony. W takim przypadku, można po prostu określić złożoność dla wszystkich możliwych przypadków, lub też, jeśli byłoby ich zbyt wiele ograniczyć się do przypadku najprostszego i najbardziej złożonego. Sytuację, gdy dane wejściowe nie są ograniczone, można podzielić na dwa warianty:
— postać danych wejściowych nie zmienia się, zmienia się tylko ilość,
— zmienia się zarówno ilość, jak też postać danych wejściowych.

Zmienia się tylko ilość danych. W zależności od danego problemu, można zastosować następujące podejścia:
— analityczne,
— szacowanie statystyczne.

Wariant pierwszy występuje wtedy, gdy analizując dany algorytm, możemy wyznaczyć arbitralnie ilość operacji niezbędną do przetworzenia określonej ilości danych wejściowych (kwantu). Uzyskaną w ten sposób złożoność obliczeniową (nazwijmy ją bazową) można następnie wykorzystać do obliczeń dla większych ilości danych wejściowych (wielokrotności pierwotnego kwantu).

Jeżeli jednak nie można wyznaczyć funkcji opisującej złożoność obliczeniową na podstawie analizy algorytmu, należy spróbować zrobić to przy pomocy narzędzi statystycznych. Konieczne jest oszacowanie ilości operacji dla różnych ilości danych wejściowych, równo od siebie oddalonych. W ten sposób uzyskuje się punkty wykresu funkcji wyrażającej złożoność obliczeniową. Następnie wyznacza się wzór takiej funkcji używając do tego celu odpowiednich narzędzi statystycznych. Niekiedy z resztą z samego przebiegu wykresu można się rodzaju zależności domyślić.

Zmienia się ilość i postać danych. Taka sytuacja znacznie utrudnia szacowanie złożoności obliczeniowej. W pewnych przypadkach da się zastosować nieco rozszerzone podejście analityczne opisane w poprzednim punkcie, tym razem jednak taką procedurę trzeba będzie zastosować kilka krotnie, za każdym razem dla innego typu danych wejściowych. Warunkiem, aby takie szacowanie dało się przeprowadzić jest oczywiście to, aby cały dopuszczalny zbiór danych wejściowych dało się podzielić na podzbiory składające się z elementów takiego samego typu. Wynikiem końcowym jest zbiór funkcji zależności obliczeniowej.

4.5.3. Złożoność obliczeniowa kompilatora

Przedstawione powyżej podejścia, nie bardzo można zastosować do algorytmów kompilatora. Na przeszkodzie stoi niczym nieograniczony z góry zbiór danych wejściowych. Nie tylko nie ma on ograniczenia na ilość danych, ale również na ich postać. Poszczególne konstrukcje języka VHDL mogą być ze sobą łączone w jednym pliku wejściowym, bez żadnych ograniczeń. Nawet wydawałoby się, tak użyteczna metoda, jak analiza statystyczna, okazuje się zbyt ograniczona, aby dało się ją wykorzystać. Na przeszkodzie staje niemożność sklasyfikowania danych wejściowych. Kolejnym rozwiązaniem, które można by próbować zastosować to stworzenie uniwersalnego modelu złożoności obliczeniowej dla programu w języku VHDL. Takie podejście jest technicznie do zrealizowania, jednak posługiwanie się nim w praktyce raczej byłoby kłopotliwe. Właściwie należałoby stworzyć osobne narzędzie, które na podstawie danego pliku zawierającego źródła VHDL obliczałoby ilość operacji. Jednak jego praktyczna przydatność byłaby raczej znikoma, bo trudno sobie wyobrazić, aby przed dokonaniem kompilacji dokonywać szacowania, jak bardzo skomplikowany będzie to proces.

Pozostaje więc oszacowanie złożoności dla każdej z instrukcji VHDL oddzielnie. Nie jest to metoda pozwalająca na dokładne określenie przebiegu funkcji opisującej złożoność obliczeniową, dostarcza jednak pewnych przesłanek co do rzeczywistych nakładów niezbędnych do wykonania przekładu danego źródła.

Podstawą do określenia złożoności jest właściwe zdefiniowanie operacji elementarnej. Musi ona spełniać warunki przedstawione wcześniej w rozdziale 4.5.1. Na-

stępnie bazując na tym trzeba znaleźć liczbę operacji niezbędną do kompilacji poszczególnych instrukcji języka VHDL.

4.5.4. Operacja elementarna

Operacją O, której częstość będzie wyrażała złożoność obliczeniową nazywać będziemy zestaw działań dokonujących generacji równania boolowskiego dla jednego bitu danego sygnału lub zmiennej Nie ma przy tym znaczenia, czy w wyniki otrzymujemy jedno równanie (logika kombinacyjna), czy też zbiór równań (logika sekwencyjna). Założone jest, że generacja w obu przypadkach zajmuje taką samą ilość obliczeń. Jest to oczywiście uproszczenie.

4.5.5. Instrukcja przypisania

Jeżeli dane jest n instrukcji przypisania każda o m_i bitów, to liczba operacji dana jest poniższym wzorem (4.6):

$$O = \sum_{i=1}^{n} m_i \tag{4.6}$$

Można tą zależność uprościć. Każda instrukcja składa się z określonej liczby jednobitowych przypisań. Zamiast więc rozpatrywać złożoność w rozbiciu na poszczególne instrukcję, można ją określić na zbiorze pojedynczych jednobitowych przypisań. Wtedy zależność uprości się do postaci wzoru 4.7, gdzie N jest liczbą wygenerowanych jednobitowych równań. Widać jasno, że liczba operacji zależy liniowo od ilości przypisań.

$$O = N \tag{4.7}$$

4.5.6. Instrukcje *if* oraz *case*

Obie konstrukcje są bardzo podobne do siebie, więc będą analizowane razem. Zgodnie z algorytmem przekładu (rozdział 3) należy znaleźć wszystkie instrukcje przypisania wewnątrz gałęzi *if* oraz *case*, a następnie wygenerować równania wynikowe. Ogólny wzór opisujący taką sytuacje wyrażony jest poniższym równaniem (4.8):

$$O = \sum_{i=1}^{k} n_i + N \tag{4.8}$$

gdzie:
— k - ilość gałęzi instrukcji,
— n_i - liczba jednobitowych przypisań w danej gałęzi,
— N - liczba różnych jednobitowych celów przypisania (opisana jest wzorem 4.9):

$$N = \left| \bigcup_{i=1}^{k} n_i \right| \tag{4.9}$$

Załóżmy, że liczba przypisywanych bitów w każdej z gałęzi jest taka sama. Wówczas wzór przybierze prostszą postać (4.10):.

$$N = n \Leftrightarrow O = n(k+1) \tag{4.10}$$

Dla danej ilości gałęzi k liczba operacji zależy liniowo od liczby jednobitowych przypisań.

4.5.7. Pętla *for*

Jeżeli dana jest pętla for o k iteracjach, zawierająca wewnątrz swojego bloku n jednobitowych przypisań to ilość operacji jaka jest konieczna, do wykonania przekładu takiej pętli wyraża się wzorem 4.11. Jest to zgodne z zasadą działania pętli *for*. Kod pętli jest wykonywany tyle razy, ile wynosi liczba iteracji (rozdział 3.12). Sytuacja ta oczywiście odnosi się do pętli w jej najprostszej postaci, czyli bez instrukcji *next* i *exit*.

$$O = kn \tag{4.11}$$

4.5.8. Ograniczenia przedstawionego sposobu szacowania złożoności obliczeniowej

Przedstawiona metoda szacowania złożoności obliczeniowej, ma jednak pewne wady. Sprawiają one, że obliczenia wykonane za pomocą powyższego wzoru nie zawsze są dokładne.

Pierwsza wada dotyczy instrukcji przypisania. Po prawej stronie może wystąpić wywołanie funkcji lub też operacja arytmetyczna. W takim wypadku, ilość równań, jaka rzeczywiście jest tworzona, może być większa. O ile jeszcze dla operacji arytmetycznych można określić zależność między ilością bitów wejściowych, a ilością równań wejściowych, to w przypadku wywołania funkcji sytuacja jest patowa. Wykonanie takiej funkcji może oznaczać utworzenie dodatkowych równań.

Kolejnym problemem jest instrukcja *for*. Komplikacje zaczynają się, kiedy występuje ona razem z instrukcjami *next* oraz *exit*. Wtedy pojawiają się bloki *if*, które zmieniają pierwotną ilość operacji. W przypadku pojedynczej pętli, jest jeszcze możliwe w miarę proste oszacowanie złożoności, ale nie należy zapominać, że mogą wystąpić układy kilku zagnieżdżonych pętli, z instrukcjami *next* i *exit*, odwołującymi się nie tylko do pętli w której się bezpośrednio znajdują.

4.6. Podsumowanie

Niniejszy rozdział przedstawił ocenę zaproponowanych algorytmów pod kątem realizowania przez nie celu postawionego w pracy. Ze względu na temat badań najlepsze rezultaty można było uzyskać za pomocą testów polegających na kompilacji plików VHDL zawierających badane instrukcje w różnych postaciach. Pierwszym krokiem była implementacja algorytmów jako modułu kompilatora do syntezy logicznej [17]. Do testowania wykorzystano trzy grupy testów:

— testy sprawdzające poprawność generacji równań dla poszczególnych instrukcji,
— testy oceniające wydajność algorytmów,
— rozbudowane przykłady testowe służące do weryfikacji całości koncepcji.

Wyniki procesu testowania pokazują, iż przedstawione algorytmy są w stanie generować poprawne równania boolowskie reprezentujące logikę kombinacyjną, mając jako dane wejściowe instrukcje sekwencyjne języka VHDL. Wartości czasu kompilacji i zużycia pamięci podczas tego procesu, które udało się uzyskać za pomocą pilotażowej implementacji pozwalają uznać, że zastosowane algorytmy mają efektywność pozwalającą na ich użycie w profesjonalnych przemysłowych i akademickich narzędziach syntezy. Wnioski te potwierdza także, wyrażony ilością bramek i przerzutników, rozmiar generowanych układów scalonych (tabela 4.2). Zaproponowane rozwiązanie generuje wyniki zbliżone to wyników istniejących kompilatorów.

5. Podsumowanie

Niniejszy rozdział odpowiada na pytanie w jaki sposób udało się osiągnąć cel postawiony we wstępie rozprawy, a także jak dowiedziono prawdziwości tezy. Zawiera on także ocenę końcową zaprezentowanych wcześniej algorytmów, z wykazaniem ich silnych, ale też i słabych stron. Podsumowanie wskazuje także kierunek w jakim powinny podążyć dalsze badania.

5.1. Ocena zaproponowanego rozwiązania i osiągniętych wyników

Punktem wyjścia dla dokonania rzetelnej i obiektywnej oceny niniejszej rozprawy musi być sprawdzenie otrzymanych wyników pod kątem ich zgodności z celem pracy przedstawionym we wstępie. Aby dokonać dokładniejszej analizy, cel rozprawy opisany w rozdziale 1.2 został rozbity na następujące elementy składowe:
— opracowanie algorytmów automatycznej generacji równań boolowskich dla sekwencyjnych instrukcji języka VHDL,
— weryfikacja algorytmów pod kątem poprawności,
— określenie granic stosowalności.
Przebieg realizacji punktu pierwszego został dokładnie opisany w rozdziale 3, ale w celu podsumowania pewne informacje przedstawione zostaną skrótowo raz jeszcze. Zaproponowane w pracy rozwiązanie składa się z dwóch grup algorytmów:
— algorytmów konwersji pętli *for* do postaci liniowej,
— algorytmów generacji równań boolowskich.

Pierwsza grupa zajmuje się zamianą pętli *for* na postać liniową, która tej instrukcji nie zawiera. Oprócz usunięcia samej pętli, konieczne jest także odpowiednie skonwertowanie instrukcji towarzyszących - *next* oraz *exit*. Istotny problem stanowią układy zagnieżdżonych pętli wprowadzające dodatkowe komplikacje, niespotykane w innych językach programowania. Ponieważ cały proces linearyzacji jest złożony i trudno go było zaprezentować w postaci jednego algorytmu, został on podzielony na kilka części:

Algorytm sterujący konwersją (rozdział 3.16.2) - rekurencyjny algorytm skanujący źródło VHDL w poszukiwaniu pętli *for*. Odpowiada za tworzenie sztucznych bloków instrukcji *if*, zawierających kod, którego wykonanie jest uzależnione od warunków implikowanych przez występowanie instrukcji *next* oraz *exit* w pętli. Algorytm ten wywołuje także wszystkie pozostałe algorytmy procesu linearyzacji.

Algorytm linearyzacji pętli *for* (rozdział 3.16.3) - uruchamiany jest w chwili napotkania pętli. Odpowiada za określenie zakresu wartości zmiennej iteracyjnej, oraz ilości iteracji pętli. Wywołuje także algorytm sterujący konwersją dla każdej iteracji.

Algorytm usuwania instrukcji *next* i *exit* (rozdział 3.16.4) - usuwa ze źródła instrukcje *next* i *exit*, generuje odpowiednie warunki sterujące wykonaniem poszczególnych iteracji.

Po usunięciu pętli, można przystąpić do generacji równań boolowskich. Czynność tą realizuje druga grupa algorytmów składająca się z następujących części:

Algorytmy translacji procesu (rozdział 3.5) - główny moduł, wywoływany rekurencyjnie dla każdego zagnieżdżenia. Skanuje ciąg leksemów w poszukiwaniu poszczególnych instrukcji języka VHDL. Następnie wywołuje stosowne algorytmy cząstkowe.

Algorytmu generacji równań dla instrukcji przypisania sygnału (rozdział 3.6.2) - jego zadaniem jest wygenerowanie równań boolowskich.

Algorytmu generacji równań dla instrukcji przypisania zmiennej (rozdział 3.6.3) - jak wyżej, ale sytuacja dotyczy zmiennej.

Algorytmu wywołania procedury - wywołuje procedurę.

Algorytmu generacji równań dla instrukcji *if* (rozdział 3.11) - tworzy równania boolowskie dla instrukcji *if*.

Algorytmu generacji równań dla instrukcji *case* (rozdział 3.11) - jak wyżej, ale dla instrukcji *case*.

Algorytm optymalizacji przerzutników dla zmiennych (rozdział 3.10.3) - usuwa zbędne przerzutniki z końcowych równań.

Wynikiem działania algorytmów jest zbiór równań boolowskich. Dla każdego bitu sygnału i zmiennej, dla której występuje przypisanie wewnątrz bloku *process* generowane jest jedno równanie.

Zaproponowane algorytmy zostały zaimplementowane i stanowią jeden z modułów kompilatora VHDL2Bool (rozdział 4.1). Następnie przeprowadzona została procedura testowa, której zadaniem było sprawdzenie czy zaproponowane rozwiązanie generuje poprawne wyniki. Procedura ta szczegółowo została opisana w rozdziale 4.2. Wyniki testów potwierdziły, iż jest możliwe generowanie poprawnych równań boolowskich dla instrukcji sekwencyjnych języka VHDL.

Ostatnim krokiem w badaniach było określenie granic stosowalności algorytmów. Przyjęte zostały następujące kryteria tej oceny:

— czas kompilacji kodu VHDL,
— zużycie pamięci podczas kompilacji,
— rozmiar wygenerowanego układu FPGA.

Dokładne rezultaty powyższych testów przedstawione zostały w rozdziałach 4.3 oraz 4.4. Całość została uzupełniona wyznaczeniem teoretycznej złożoności obliczeniowej algorytmów (rozdział 4.5). Znaczące jest, że wyniki testów i szacunkowa złożoność obliczeniowa są zgodne.

Na podstawie wyników testów należy uznać, iż tezę pracy przedstawioną we wstępie udało się obronić. Algorytmy które zostały opracowane, generują poprawne równania boolowskie, a ich efektywność (wyjaśnienie pojęcia znajduje się w rozdziale 1.2) jest porównywalna z rozwiązaniami komercyjnymi. Należy przy tym pamiętać, iż w odróżnieniu od tych ostatnich, badania prezentowane w pracy zostały upublicznione.

Zaletą prezentowanych algorytmów jest ich kompleksowość i szczegółowość. Celem badań było opracowanie wiedzy sformalizowanej i łatwej do zastosowania w praktyce. Te sformalizowane algorytmy stanowią istotny wkład w istniejącą teorię. Wzorem do naśladowania były podręczniki tradycyjnych metod kompilacji. Pod tym

względem jest to praca unikatowa, gdyż trudno znaleźć w literaturze podobne opracowania. Przedstawiona wiedza ma w dużej mierze charakter uniwersalny, mimo iż format wyjściowy jest nietypowy dla tego typu narzędzi. Oprócz teorii, praca zawiera wiele informacji praktycznych. Omawiane są zagadnienia implementacji algorytmów. Przedstawione została także metodologia testowania implementacji.

5.2. Kierunek dalszych prac i badań

Przedstawione w niniejszej pracy rozwiązanie nie jest optymalne i z pewnością w toku dalszych prac można je poprawić.

Najważniejszą zmiana jaką można by rozważać byłaby próba kompilacji instrukcji *for* z wykorzystaniem mechanizmów logiki sekwencyjnej. W tej chwili konstrukcja ta jest transformowana do postaci kombinacyjnej przez dostępne na rynku narzędzia. Takie działanie nie wydaje się do końca efektywne, gdyż często prowadzi do marnotrawienia zasobów. Podejście takie jest wynikiem dokładnego interpretowania semantyki języka VHDL, gdzie logika sekwencyjna powstaje tylko w ściśle określonych przypadkach. W przypadku jednak pętli *for* zwłaszcza, gdy mamy do czynienia z instrukcjami *next* i *exit* korzystniejsze może okazać się stworzenie maszyny stanów, niż stosowania podejścia opartego na linearyzacji pętli. Najlepszym wyjściem byłoby umożliwienie programiście wyboru drogi syntezy. Najpierw jednak należy zbadać jak użycie maszyny stanów na wyjściowy układ.

Kolejną modyfikacja to oparcie narzędzia na dobrze zdefiniowanej formie pośredniej. Pozwoliłoby to oddzielić analizę kompilowanego języka od procesu syntezy. W konsekwencji otworzono by drogę do łatwego uzyskania całej rodziny narzędzi, dostosowanych do innych niż VHDL języków, oraz różnych formatów wyjściowych.

A. Wzorce przerzutników

Zgodnie z informacjami z rozdziału 3 w pewnych przypadkach równania boolowskie wygenerowane przez kompilator mają postać sekwencyjną - tworzą przerzutnik. Są one generowane według wzorów 3.8 i 3.7, ale same z siebie stanowią tylko szkielet przerzutnika. Aby utworzyć w pełni funkcjonalny element konieczne jest zastosowanie szablonu. W badaniach wykorzystywany był przerzutnik typu D wyzwalany poziomem, tak zwany zatrzask (ang. *latch*). Poniżej zaprezentowane są szablony dla takiego elementu w wersji aktywowanej poziomem wysokim i niskim.

```
-- pragma asyn(t-1)
{
C_tmp=C
D_tmp=D
-- latch(C_high,D)
S_tmp=D_tmp&C_tmp
R_tmp=C_tmp&!D_tmp
Q(t)=S_tmp|(!R_tmp&Q(t-1s))
}
```

```
-- pragma asyn(t-1)
{
C_tmp=C
D_tmp=D
-- latch(C_low,D)
S_tmp=D_tmp&!C_tmp
R_tmp=!C_tmp&!D_tmp
Q(t)=S_tmp|(!R_tmp&Q(t-1))
}
```

B. Zawartość płyty CD

Płyta CD dołączona do niniejszej pracy zawiera kod źródłowy kompilatora VHDL, skompilowane programy a także zbiór użytych testów i wszystkie narzędzia wspomagające. W celu lepszej orientacji opisana zostanie struktura katalogów. Oto ona:

— *VHDL2bool* - zawiera źródła i skompilowane programy składające się na kompilator VHDL w którym zaimplementowane są niniejsze algorytmy.
— *Narzedzia do testow* - zawiera wszystkie programy pomocnicze użyte podczas weryfikacji.
— *Testy* - zbiór testów użytych podczas badań. Zawiera wszystkie przykłady testowe użyte podczas badań.

Bibliografia

[1] Adamski M., Węgrzyn M., Wolański P. Simulating and synthesising of reconfigurable logic controllers using vhdl. In *42. Internationales Wissenschaftliches Kolloquium : Informatik und Automatisierung im Zeitalter der Informationsgesellschaft*, volume Band 1, pages 522–527, Ilmenau, Niemcy, 1997. Technische Universität Ilmenau, Ilmenau, Technische Universität Ilmenau.

[2] Adamski Marian. Application specific logic controllers for safety critical systems. In *14th World Congress of IFAC International Federation of Automatic Control*, volume Vol. Q : Transportation Systems : Computer Control, pages 519–524, Beijing, Chiny, 1999. Oxford, International Federation of Automatic Control.

[3] Adamski Marian. Specyfikacja, analiza i synteza reprogramowalnych mikrosystemów cyfrowych. *Reprogramowalne Układy Cyfrowe - RUC '99 : Materiały II Krajowej Konferencji Naukowej*, strony 11–20, Polska, 1999. Szczecin, Wydaw. i Drukarnia Instytutu Informatyki Politechniki Szczecińskiej.

[4] Adamski Marian. Bezpośrednia implementacja sieci Petriego w reprogramowalnych układach cyfrowych. *Reprogramowalne Układy Cyfrowe - RUC 2000 : Materiały III Krajowej Konferencji Naukowej*, strony 131–138, Polska, 2000. Szczecin, Wydaw. i Drukarnia Wydziału Informatyki Politechniki Szczecińskiej.

[5] Aho A. V.,Sethi R., Ullman J. D. *Kompilatory - reguły, metody, narzędzia.* Wydawnictwo Naukowo-Techniczne, 2002.

[6] Allen R., Kennedy K. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[7] Altera Corporation. *MAX+PLUS II Getting Started Manual.* Altera Corporation, wersja 8.1, 1997.

[8] Altera Corporation. *Cyclone Device Handbook, Volume 1.* Cyclone Device Handbook, Volume 1, 2005.

[9] Altera Corporation. *Introduction to Quartus II version 5.0.* Altera Corporation, www.altera.com, April 2005.

[10] Altera Corporation. *Quartus II Version 5.0 Handbook.* Altera Corporation, www.altera.com, 2005.

[11] Bhasker J. *A VHDL Synthesis Primer - Second Edition.* Star Galaxy Publishing, 1998.

[12] Bhasker J. *A SystemC Primer, Second Edition.* Star Galaxy Publishing, second edition, 2004.

[13] Bielecki W., Hyduke S., Drążkowski R., Liersz M., Radziewicz M., Błaszyński P. Organizacja kompilatora do syntezy układów logicznych z syntezowalnego podzbioru języka VHDL. *Materiały 4 Sesji Naukowej Informatyki*, 1999.

[14] Bielecki W., Hyduke S., Radziewicz M. Generowanie równań boolowskich dla procesu z instrukcją wait języka VHDL. *Materiały 3 Krajowej Konferencji Naukowej: RUC 2000*, 2000.

[15] Bielecki W., Radziewicz M. Translacja pętli for języka VHDL do postaci równań boolowskich. *Elektronika - konstrukcje, technologie, zastosowania*, 7/2007, 2007.

[16] Bielecki W., Wierciński T. Generacja maszyny stanów dla procesu z wieloma instrukcjami oczekiwania wait. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL*

do projektowania układów logicznych. Wydział Informatyki Politechniki Szczecińskiej, 2002.

[17] Bielecki Włodzimierz. *Kompilator języka VHDL do projektowania układów logicznych*, chapter Organizacja kompilatora języka VHDL do projektowania układów logicznych, strony 1–12. Pracownia Poligraficzna Wydziału Informatyki Politechniki Szczecińskiej, 2002.

[18] Black David C., Donovan Jack. *SystemC: From the Ground Up*. Kluwer Academic Publishers, 2004.

[19] Błaszyński Piotr. *Analizator semantyczny do generowania reprezentacji modelu statycznego w kompilatorze języka VHDL*. Rozprawa doktorska, Wydział Informatyki Politechniki Szczecińskiej, 2004.

[20] Brown Stephen. *Fundamentals of Digital Logic with VHDL Design*. McGraw-Hill, 2004.

[21] Cohen Ben. *VHDL Coding Styles and Methodologies*. Springer, second edition, 1999.

[22] Compton Katherine, Hauck Scott. An introduction to reconfigurable computing. In *IEEE Computer*, April 2000.

[23] Cong Jason, Peck John. On acceleration of logic synthesis algorithms using FPGA-based reconfigurablecoprocessors. Technical report, Department of Computer Science University of California, Los Angeles, CA90024, 1997.

[24] De Micheli G. *Synteza i optymalizacja układów cyfrowych*. Wydawnictwo Naukowo-Techniczne, Warszawa, 1998.

[25] Drążkowski Robert. Analiza leksykalna i syntaktyczna języka VHDL używanego do generacji równańboolowskich. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania ukladów logicznych*. Wydział Informatyki Politechniki Szczecińskiej, 2002.

[26] Eles Petru, Kuchcinski Krzysztof, Peng Zebo. Synthesis of systems specified as interacting VHDL processes. *Integr. VLSI J.*, 21(1-2):113–138, 1996.

[27] Eles Petru, Kuchcinski Krzysztof, Peng Zebo, Minea Marius. Compiling VHDL into a high-level synthesis design representation. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 604–609, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[28] Eles Petru, Kuchcinski Krzysztof, Peng Zebo, Minea Marius. Synthesis of VHDL concurrent processes. In *EURO-DAC '94: Proceedings of the conference on European design automation*, pages 540–545, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[29] Équipe Achitecture des Systémes et Micro-Électronique. *Alliance documentation for version 5.0*. Laboratoire MASI/CAO-VLSI, Institut de Programmation Université Pierre etMarie Curie (PARIS VI), http://www-asim.lip6.fr/recherche/alliance/doc/.

[30] Équipe Achitecture des Systémes et Micro-Électronique. *Alliance documentation for version 3.2*. Laboratoire MASI/CAO-VLSI, Institut de Programmation Université Pierre etMarie Curie (PARIS VI), http://www-asim.lip6.fr/recherche/alliance/olddoc/, 1 edition, 1992.

[31] Équipe Achitecture des Systémes et Micro-Électronique. Alliance: A complete CAD system for VLSI design, 2004.

[32] Goldberg David. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[33] Greiner Alain, Pêcheux François. Alliance: A complete set of CAD tools for teaching VLSI design, 1992.

[34] Guo Z., Najjar W., Vahid F., Vissers K. A quantitative analysis of the speedup factors of FPGAs over processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA*

12th international symposiumon Field Programmable Gate Arrays, pages 162–170, New York, NY, USA, 2004. ACM Press.

[35] Gupta R. K., De Micheli G. Hardware-software co-synthesis for digital systems. Stanford University, Stanford, CA 94305., Stanford University, Stanford, CA 94305., 1993.

[36] Ha Y., Vanmeerbeeck G., Schaumont P., Vernalde S., Engels M., Lauwereins R., De Man H. Virtual java/fpga interface for networked reconfiguration. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific designautomation*, pages 558–563, New York, NY, USA, 2001. ACM Press.

[37] Hauser J. R., Wawrzynek J. Garp: A MIPS processor with a reconfigurable coprocessor. University of California, Berkeley, University of California, Berkeley, 1997.

[38] IEEE Standards Board. *IEEE Std 754-1985 Standard for Binary Floating-Point.* IEEE, 1985.

[39] IEEE Standards Board. *IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual.* IEEE Standards Board, 1991.

[40] IEEE Standards Board. IEEE Standard VHDL Language Reference Manual IEEE std 1076-1993. In *IEEE Standards Board*, 1994.

[41] IEEE Standards Board. *IEEE Std 854-1987 Standard for Radix-Independent Floating-Point Arithmetic.* IEEE, 1994.

[42] IEEE Standards Board. *1364-1995 IEEE Standard Hardware Description Language Based on the Verilog(Hardware Description Language).* IEEE, 1995.

[43] IEEE Standards Board. *1364-2001 IEEE Standard for Verilog Hardware Description Language.* IEEE, 2001.

[44] IEEE Standards Board. *1800-2005 IEEE Standard for System Verilog: Unified Hardware Design, Specificationand Verification.* IEEE, 2005.

[45] IEEE Standards Board. *1364-2005 IEEE Standard for Verilog Hardware Description Language.* IEEE, 2006.

[46] IEEE Standards Board. *1666-2005 IEEE Standard SystemC Language Reference Manual.* IEEE, 2006.

[47] Jaworski Paweł. Postprocesor kompilatora języka VHDL. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania układów logicznych.* Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[48] Jaworski Paweł. Wersje implementacyjne postprocesora kompilatora języka VHDL. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania układów logicznych.* Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[49] John M., Smith S. *Application-Specific Integrated Circuits.* Addison-Wesley Professional, first edition, 1997.

[50] Kalisz Józef. *Podstawy elektroniki cyfrowej.* Wydawnictwa Komunikacji i Łączności, Warszawa, 1998.

[51] Koren Israel. *Computer Arithmetics Algorithms.* A. K. Peters, second edition, 2002.

[52] Kukimoto Yuji. *BLIF-MV.* University of California Berkeley, May 1996.

[53] Ku D., De Micheli G. Hardwarec – a language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990.

[54] Liersz Marcin K. Algortym generowania równań boolowskich dla instrukcji przypisania zawierającejodwołania do tablic w języku VHDL. *Kompilator języka VHDL do projektowania układów logicznych.* Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[55] Lockwood J. W., Naufel N., Turner J. S., Taylor D. E. Reprogrammable network packet processing on the Field Programmable Port Extender (FPX). *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, feb 2001.

[56] Louden Kenneth C. *Compiler Construction: Principles and Practice.* Course Technology, 1997.

[57] Łuba T., Markowski M. A., Zbierzchowski B. *Kompilatory układów logicznych.* Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa, 1995.

[58] Łuba Tadeusz. *Specjalizowane układy cyfrowe w strukturach PLD i FPGA.* Wydawnictwo Komunikacji i Łączności, Warszawa, 1997.

[59] Łuba Tadeusz. *Synteza układów logicznych.* Wyższa Szkoła Informatyki Stosowanej i Zarządzania, Warszawa, wydanie drugie, 2001.

[60] Łuba Tadeusz (redaktor). *Synteza układów cyfrowych.* Wydawnictwa Komunikacji i Łączności, wydanie pierwsze, 2003.

[61] Majewski Władysław. *Układy logiczne.* Wydawnictwa Naukowo-Techniczne, 2003.

[62] Martin Kenneth W. *Digital Integrated Circuit Design.* Oxford University Press, 2000.

[63] Maxfield Clive "Max". *Design Warrior's Guide to FPGAs.* Elsevier, 2004.

[64] Mekenkamp G. E., Middelhoek P. F. A., Molenkamp B. E., Hofstede J., Krol T. A syntax based VHDL to CDFG translation model for high-level synthesis. In *VIUF Proceedings Spring 1996*, pages 89–97, febrary 1996.

[65] Mekenkamp G.E. *A New Approach to VHDL-Based Synthesis.* PhD thesis, University of Twente, January 1998.

[66] Meyer-Baese Uwe. *Digital Signal Processing Using Field Programmable Gate Arrays.* Spinger, 2001.

[67] Miller E., Squire J. esim: A structural design language and simulator for computer architecture education. *Workshop on Computer Architecture Education*, June 2000.

[68] Mościcki Mirosław. Organizacja narzędzi do testowania i weryfikacji równań boolowskich. *Materiały 3 Krajowej Konferencji Naukowej: RUC '2000 Szczecin*, strony 381–388, 2000.

[69] Mościcki Mirosław. Generowanie równań boolowskich dla funkcji i procedur języka vhdl. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania układów logicznych.* Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[70] Molenkamp E, Mekenkamp G. E., Hofstede J, Krol T. SIL: an intermediate for syntax based VHDL synthesis. In *VIUF Proceedings*, pages 5.1–5.9, April 1995.

[71] Muchnick Steven. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[72] Open SystemC Initiative. https://www.systemc.org.

[73] Open SystemC Initiative (OSCI). *SystemC 2.0 User's Guide.* Open SystemC Initiative (OSCI), www.systemc.org, 2002.

[74] Palnitkar Samir. *Verilog® HDL: A Guide to Digital Design and Synthesis.* Prentice Hall PTR, second edition, 2003.

[75] Papadimitriou Christos H. *Złożoność obliczeniowa.* Wydawnictwo Naukowo-Techniczne, 2002.

[76] Parnis J., Lee G. Exploiting FPGA concurrency to enhance JVM performance. In *CRPIT '04: Proceedings of the 27th conference on Australasian computer science*, pages 223–232, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[77] Pedroni Volnei A. *Circuit Design with VHDL.* The MIT Press, 2004.

[78] Peng Zebo. Synthesis of VLSI systems with the Camad design aid. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 278–284, Piscataway, NJ, USA, 1986. IEEE Press.

[79] Perry Douglas L. *VHDL : Programming By Example.* McGraw-Hill Professional, fourth edition, 2002.

[80] Radziewicz Marcin. Generacja równań boolowskiach dla instrukcji for języka VHDL. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania układów logicznych*. Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[81] Radziewicz Marcin. Przekład instrukcji if oraz case języka VHDL do postaci równań boolowskich. Włodzimierz Bielecki, redaktor, *Kompilator języka VHDL do projektowania układów logicznych*. Wydział Informatyki Politechniki Szczeciñskiej, 2002.

[82] Radziewicz Marcin. Synteza instrukcji sekwencyjnych if, case, for języka VHDL do postaci równań boolowskich. *Reprogramowalne układy cyfrowe: Materiały siódmej Krajowej Konferencji Naukowej: RUC '2004*, 2004.

[83] Radziewicz Marcin. Translacja instrukcji sekwencyjnych języka VHDL do postaci równań boolowskich. Andrzej Stateczny, redaktor, *Metody informatyki stosowanej w technice i technologii*, wydanie 8, 2005.

[84] Radziewicz Marcin. Translacja instrukcji sekwencyjnych języka VHDL. *Pomiary, Automatyka, Kontrola - materiały konferencji Reprogramowalne układy cyfrowe RUC'06*, 7bis, 2006.

[85] Radziewicz Marcin. Translation of VHDL sequential statements. *Journal of Environmental Studies - materiały konferencji Advanced Computer Systems 2007*, 2007.

[86] Roth Ch. H. Jr. *Digital Systems Design Using VHDL*. PWS Publishing Company, 1997.

[87] Schoeberl Martin. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[88] Sentovich E. M., Singh K. J., Lavagno L., Moon C., Murgaj R., Saldanha A., Savoj H., Stephan P. R., Brayton R. K., Sangiovanni-Vincentelli A. SIS: A system for sequential circuit synthesis. Technical report, University of California, Berkeley, May 1992.

[89] Skahill Kevin. *Jêzyk VHDL. Projektowanie programowalnych układów logicznych*. Wydawnictwa Naukowo-Techniczne, wydanie drugie, 2004.

[90] Skorupski Andrzej. *Podstawy techniki cyfrowej*. Wydawnictwa Komunikacji i Łączności, 2001.

[91] Synopsys Inc. *FPGA Express VHDL Reference Manual*, December 1997.

[92] System Verilog. http://www.systemverilog.org.

[93] Tariov A., Mąka T. Opracowanie biblioteki modeli bazowych w języku VHDL dla sprzętowej implementacji wielopotokowych, bitowo-szeregowych procesorów DSP. *Materiały 6 Sesji Naukowej Informatyki Szczecin: INFORMA Wydawnictwo Wydziału Informatyki Politechniki Szczecińskiej*, 2001.

[94] Tariov A., Mąka T., Maciaszczyk R., Tariova G. Struktury jednostek przetwarzających dla procesorów DWT. *Reprogramowalne układy cyfrowe: Materiały VIII Krajowej Konferencji Naukowej: RUC*, 2005.

[95] Tariov A., Tariova G. Zorientowane sprzętowo algorytmy realizacji bazowych operacji dyskretnej transformaty falkowej. *Pomiary, Automatyka, Kontrola nr 7bis - materiały konferencji Reprogramowalne układy cyfrowe RUC'06*, 2006.

[96] The VIS Group. VIS: A system for verification and synthesis. Computer-Aided Design, 1996.

[97] The VIS Group. VIS tutorial. Formal Method in Computer-Aided Design, November 1996.

[98] Thomas Donald E., Moorby Philip R. *The Verilog® Hardware Description Language*. Springer, fifth edition, 2002.

[99] University of California Berkeley. *Berkeley Logic Interchange Format (BLIF)*. University of California Berkeley, July 1992.

[100] Van Den Hurk J., Jess J. *System Level Hardware/Software Co-Design: An Industrial Approach*. Kluwer Akademic Publishers, 1998.

[101] Villa T., Swamy G., Shiple T, VIS Group. *VIS User's Manual*. University of Colorado at Boulder, Fabraury 1996.

[102] Wierciński Tomasz. *Generowanie równań boolowskich dla syntezowalnych źródeł języka VHDL opisujących logikę sekwencyjną*. Rozprawa doktorska, Wydział Informatyki Politechniki Szczecińskiej, 2005.

[103] Wilkinson Barry. *Układy cyfrowe*. Wydawnictwa Komunikacji i Łączności, 2000.

[104] Wrona Włodzimierz. *VHDL język opisu i projektowania układów cyfrowych*. Wydawnictwo Pracowni Komputerowej Jacka Skalskiego, 1998.

[105] Xilinx Corporation. *Spartan-IIE 1.8V FPGA Family:Complete Data Sheet*. Xilinx Corporation, http://www.xilinx.com, 2004.

[106] Xilinx Corporation. *StateCAD Manual*, 2005.