一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2024-85,RECONF2024-115(2025-01)

RTOS利用システムのフルハードウェア化における Pythonによるハードウェア設計

志賀　光[†]　石浦菜岐佐[†]　冨山　宏之[††]　神原　弘之[†††]

† 関西学院大学　〒669–1330 兵庫県三田市学園上ケ原 1
†† 立命館大学　〒 525–8577 滋賀県草津市野路東 1 丁目 1–1
††† 京都高度技術研究所　〒 600–8813 京都市下京区中堂寺南町 134 番地

あらまし　本稿では RTOS を用いて実装されたリアルタイムシステムを全てハードウェア化する手法において, ハードウェアの設計およびそのユニットテストを Python で記述する手法を提案する. RTOS 機能を提供するハードウェアの回路構成は, タスク数やタスクが利用するサービスのインスタンス数等のパラメータに依存する. 従来は, Perl スクリプトによりパラメータに応じた Verilog HDL 記述を生成していたが, テキストベースで記述を生成しているため, 可読性が低く, コード量も肥大化していた. 本稿では Python を用いて Verilog HDL の AST を記述することによりハードウェアの設計を行う. AST レベルでの記述の変換を導入することにより, 状態機械のレジスタ転送とワイヤへの代入を状態毎にまとめたり, 特定の機能を実現するために必要になる複数の式計算や代入をまとめて抽象化し, 可読性の向上と記述量の削減を図る. また, 設計したモジュールのユニットテストを Python で記述し, そこから Verilog HDL のテストベンチを生成することにより, パラメータに依存するハードウェアのテストの効率化を図る.
キーワード　リアルタイムシステム, RTOS, システム合成, 高位合成

Hardware Design Using Python for Full Hardware Implementation of RTOS-Based Systems

Hikaru SHIGA[†], Nagisa ISHIURA[†], Hiroyuki TOMIYAMA[††], and Hiroyuki KANBARA[†††]

† Kwansei Gakuin University,　1 Uegahara, Gakuen, Sanda, Hyogo, 669–1330, Japan
†† Ritsumeikan University, 1–1–1 Noji–Higashi, Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo–ku, Kyoto, 600–8813, Japan

Abstract　This paper proposes a method for describing hardware design and testing using Python in the context of a technique that converts real-time systems implemented with RTOS into entirely hardware-based systems. The circuit configuration of the hardware providing RTOS functionality depends on parameters such as the number of tasks and the number of service instances utilized by the tasks. Previously, Verilog HDL descriptions tailored to the parameters were generated using Perl scripts. However, this text-based generation approach results in low readability and inflated code size. In this work, hardware design is performed by describing the Abstract Syntax Tree (AST) of Verilog HDL using Python. By introducing AST-level transformations, we improve readability and reduce code size by grouping register transfers and wire assignments for each state in a state machine, as well as abstracting multiple expression evaluations and assignments required for specific functionalities. We also describe unit tests for the designed modules in Python and generate Verilog HDL testbenches from these descriptions, enabling efficient testing of parameter-dependent hardware.
Key words　Real-Time Systems, RTOS, System Synthesis, High-Level Synthesis

1. はじめに

近年の情報通信技術の発展により, 様々なデバイスやサービスが開発されつつあるが, それに伴い組込みシステムにはより高い機能が要求されるようになっている. 特に, 車載機器やロボットの制御には機能性に加え高い応答性能が要求される. このようなリアルタイムシステムの開発はリアルタイム OS (RTOS) を用いて行われるが, システムの高機能化に伴う処理量の増加に

よってリアルタイム性の実現が困難になりつつある.

RTOS を使用したシステムの応答性能を向上させる一手法として, RTOS の機能の一部もしくは大部分をハードウェア化する手法が提案されている. 文献 [1], [2], [3] は RTOS のスケジューリング機能を, 文献 [4], [5] は RTOS のほとんどの機能をハードウェア化している. しかし, タスクおよびハンドラはソフトウェアのままであるため, CPU 待ちやコンテキストスイッチによるオーバヘッドが発生する.

文献 [6] は RTOS の機能に加えタスク/ハンドラの全てをハードウェア実装する手法を提案している. 文献 [7] は文献 [6] のアーキテクチャにおいて, RTOS のサービス処理機能を集約することにより回路規模を削減する手法を提案している. また, 文献 [8] は文献 [7] のアーキテクチャにおいて, タスクを汎用的な高位合成ツールで合成できる制御手法を提案している.

この手法において, RTOS の機能を実現するハードウェアは, リアルタイムシステムのタスク数や, それらが使用するミューテックス等のインスタンス数に応じたものが必要になる.

文献 [9] では Perl スクリプトを用いてパラメータに応じた Verilog HDL 記述を生成していた. しかし, このスクリプトはテキストベースで Verilog HDL 記述を生成しているため可読性や記述量に課題がある. また, 一旦特定のパラメータに対するハードウェアを Verilog HDL で設計してテストした後に, スクリプトを作成していたため, パラメータの値や仕様の変更後に生成されるハードウェアに対して効率的にテストを行えていなかった.

本稿では Verilog HDL によるハードウェア設計を Python を用いて AST ベースで行う手法を提案する. AST レベルで記述の変換を行うことにより, 可読性の高い設計記述から Verilog HDL 記述を生成し, 記述量の削減と保守性の向上を図る. さらに, Python を用いてハードウェアのパラメータに対応したユニットテストを記述し, テストベンチの自動生成を行うことにより, パラメータに依存するハードウェアのテストを効率化する.

2. RTOS 利用システムのフルハードウェア実装

2.1 概　　念

文献 [6] は RTOS の機能とタスクおよびハンドラ全てをハードウェア化する手法を提案している. その概念を図 1 に示す. 上図はソフトウェアで実装したシステムであり, タスク (TSK*i*) は RTOS の管理下で CPU により実行される. 文献 [6] の手法では, タスクのソースコードから下図のハードウェアを自動的に生成する. TSK*i* はタスクをハードウェア化したもので manager は RTOS の全機能をハードウェア化したものである. タスクは独立したモジュールとしてハードウェア化され, manager は各タスクの状態に基づいて実行/停止の信号を出力することによりタスクの実行制御を行う. またミューテックス等の RTOS のサービスもハードウェアとして実装される.

実行可能状態になったタスクは全て, 独立に並列に実行されるため, CPU 待ちとスケジューリングおよびコンテキストスイッチのオーバヘッドをなくせる. また, タスクをハードウェア化して高速実行できるため, システムの応答性能を格段に向上させることができる.

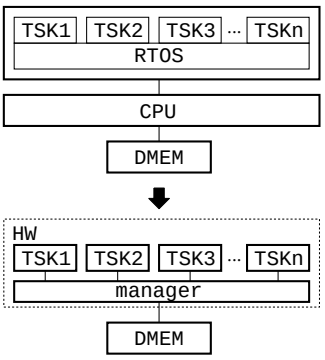

図 1: RTOS 利用システムのフルハードウェア実装 [6]

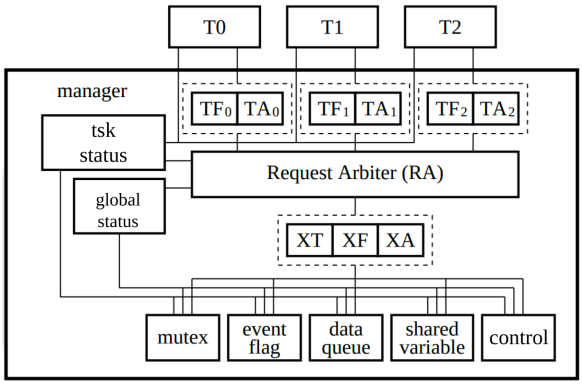

図 2: 文献 [10] のアーキテクチャ

本稿では文献 [10] のアーキテクチャ (図 2) を前提とする. T0, T1, T2 はタスクをハードウェア化したものである. これらはユーザが記述したタスクのソースプログラムから高位合成によりハードウェア化される. RTOS の機能をハードウェア化した manager は , Verilog HDL で実装されている. タスクの優先度, 実行状態等の各タスクのステータス情報は manager 内の状態レジスタ (tsk status) で管理されている.

manager の下部にある mutex, event flag, dataqueue, shared variable, control は, RTOS のサービス機能を提供するサービスモジュールである. mutex は排他制御, event flag はタスク間の同期, dataqueue はタスク間の通信, shared variable は共有変数の読み書きを提供するモジュールである. また, control はタスクの起動/休止や優先度の変更等のサービスを提供する.

タスクのサービスコール呼び出しは, manager の制御レジスタ (TF*i*, TA*i*) への書き込みに変換され, これらのレジスタに書き込みがあると, 指定されたサービスモジュールが起動される.

タスクは並列に実行されるが, サービス間の干渉を避けるためサービスは同時に 1 つのみ実行される. Request Arbiter (RA) は複数のタスクからサービス要求があった場合に, タスクの優先度に基づいてその調停を行う回路である.

manager は単相クロックで動作する同期式順序回路として設計されており, Verilog HDL で記述されている. manager および, 状態レジスタ, 各サービスモジュール, RA は, リアルタイムシステムのタスク数, サービスモジュールのインスタンス数等に依存してハードウェア構成が変わる. また, RTOS の種類によっ

てタスク優先度の上下限値やミューテックスのプロトコル等も異なる. このため, manager のハードウェア記述はこれらのパラメータに応じて生成することが必要になる.

2.2 ハードウェアの設計記述

Verilog HDL にはパラメータを用いてハードウェアを記述する枠組みが存在するが, manager のような複雑な回路の記述は難しい. そのため文献 [9] では, Perl スクリプトを用いて与えられたパラメータに対する Verilog HDL 記述を生成していた.

しかし, このスクリプトではテキストベースで Verilog HDL を生成, すなわち print 文によってパラメータに応じた Verilog HDL のテキストを生成している. このため, スクリプトの可読性が悪く, 行数も肥大化し, ハードウェアの仕様変更の際にスクリプトの保守性に課題が生じていた. またこの手法では, 特定のパラメータ値に対応する回路を Verilog HDL で設計し, その後スクリプトを用いてパラメータ値に応じた記述を生成していた. そのためパラメータの値を変更して生成されるハードウェアや仕様変更後に生成されるハードウェアに対するテストが効率的に行えていなかった.

3. Python によるハードウェア設計

3.1 概　　要

本稿では, 図 2 の管理ハードウェア (manager) を対象に, パラメータに応じたハードウェアの設計およびそのユニットテストを Python により記述する手法を提案する.

ハードウェアの設計は, Verilog HDL の抽象構文木 (AST) を Python で記述することにより行う. 一旦 AST を構築すると, それを解析して変換することができるので, これを利用して設計記述の可読性向上や記述量削減を図る. 具体的には, 状態遷移機械の記述において, Verilog HDL ではレジスタ転送とワイヤへの代入を分離して記述しなければならなかったが, これを各状態ごとにまとめて記述できるようにする. また, 特定の機能を実現するために複数の式の計算や代入が必要になる場合, これらをまとめて抽象化できるようにする.

テストに関しては, パラメータに依存するテストを Python で記述し, そこから Verilog HDL のテストベンチを自動生成できるようにする. これにより, パラメータを変更したハードウェアモジュールのテストを可能にする.

3.2 AST ベースでのハードウェア設計記述

3.2.1 Veriloggen による記述

Python でのハードウェア設計記述には Veriloggen [11] を用いる. Veriloggen は Python のライブラリとして実装されており Verilog HDL の構文木の構築, 変換等の機能を提供する. Python の汎用的な機能との組み合わせにより, パラメータに対応したハードウェアの設計が可能になる.

図 3 は, サービスモジュールの一つである優先度上限ミューテックスにおいて, タスクに設定する優先度を計算する回路 priority を, タスク数 2, ミューテックスのインスタンス数 2 の場合について記述した Verilog HDL 記述である. 12–31 行目の case 文において指定されたタスク番号 (X_TASK) が 0 の場合 (13–20 行目) と 1 の場合 (22–30 行目) の `priority` の計算を記述している.

```
1  module priority(
2    input [2-1:0] X_TASK, input [6-1:0] BASEPRI,
3    output [6-1:0] MAX_PRI
4  );
5
6    localparam ceiling0 = 2; localparam ceiling1 = 1;
7    reg [2-1:0] locker0; reg [2-1:0] locker1;
8
9    function [6-1:0] priority;
10     input [2 -1 :0] x_task; input [6-1:0] basepri;
11     begin
12       case(x_task)
13         0: begin
14           if(locker1[0]) begin
15             priority = (ceiling1 < basepri)? ceiling1
                     : basepri;
16           end else if(locker0[0]) begin
17             priority = (ceiling0 < basepri)? ceiling0
                     : basepri;
18           end else begin
19             priority = basepri;
20           end
21         end
22         1: begin
23           if(locker1[1]) begin
24             priority = (ceiling1 < basepri)? ceiling1
                     : basepri;
25           end else if(locker0[1]) begin
26             priority = (ceiling0 < basepri)? ceiling0
                     : basepri;
27           end else begin
28             priority = basepri;
29           end
30         end
31       endcase
32     end
33   endfunction
34
35   assign MAX_PRI = priority(X_TASK, BASEPRI);
36
37 endmodule
```

図 3: 最大優先度算出回路の Verilog HDL 記述 (タスク数 2, インスタンス数 2)

いるが, 15 行目と 17 行目の順序は定数 ceiling1 と ceiling0 の大きい方を先に書かなければならない. このような複雑な記述は Verilog HDL のパラメータではできない.

同じ `priority` モジュールを, タスク数 n, インスタンス数 m の場合について Veriloggen で記述した例が図 4 である. AST の記述は元の Verilog HDL の記述とほぼ対応している. 17 行目においてミューテックス獲得情報を記憶するレジスタを ceiling の値でソートしているため, 一般の n と m に対して図 3 のような記述を生成することができる.

3.2.2 状態機械の記述

Verilog HDL では, レジスタ転送と assign 文を同一の場所に記述することができない. 例えば図 5 のような状態機械を Verilog HDL で記述すると, 図 6 のようにレジスタへの代入は always 文中に, assign 文は always 文外に分離して書かなければならない. また case 文中に複数の assign 文を記述することはできない.

本稿では状態機械の各状態のレジスタ転送と assign 文を一箇所にまとめて記述できるようにし, AST 操作によりそれらを適切に分離した (正しい) Verilog HDL に変換する. 本手法による記述例を図 7 に示す. 22–48 行目のように各状態におけるレジスタ転送と assign 文を一箇所にまとめて記述できるようにする. 18 行目の `self.generate_combinational_circuit` メソッドは, `self.fsm()` で記述した AST を解析し, 出力信号の更新を assign 文および function 文に, レジスタ転送を always ブロック内に割り当てた AST を返す.

```python
import veriloggen as vg

def priority(mutex_ceiling_values: list[int], n: int):
  m = vg.Module("priority")
  X_TASK = m.Input("X_TASK", n.bit_length())
  BASEPRI = m.Input("BASEPRI", 6)
  MAX_PRI = m.Output("MAX_PRI", 6)

  ceilings = [
    m.Localparam(f"ceiling{i}", ceiling)
    for i, ceiling in enumerate(mutex_ceiling_values)
  ]
  lockers = [
    m.Reg(f"locker{i}", n.bit_length())
    for i, _ in enumerate(mutex_ceiling_values)
  ]
  ceiling_locker_list = sorted(zip(ceilings, lockers),
        key=lambda item: item[0].value)

  func = m.Function("priority")
  x_task = func.Input("x_task", n.bit_length())
  basepri = func.Input("basepri", 6)
  func.Body(
    vg.Case(x_task)(
      *[
        vg.When(task_number)(
          create_if_statement(
            [
              (
                locker[task_number],
                func(vg.Cond(ceiling < basepri,
                      ceiling, basepri)),
              )
              for ceiling, locker in
                    ceiling_locker_list
            ],
            (func(basepri)),
          )
        )
        for task_number in range(n)
      ]
    )
  )

  m.Assign(MAX_PRI(func.call(X_TASK, BASEPRI)))
  return m

priority([2, 1], 2).to_verilog("./priority.v")
```

図 4: 最大優先度算出回路の Python 記述 (タスク数 n, インスタンス数 m)

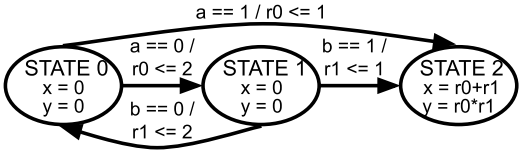

図 5: 状態遷移図

3.2.3 ロジックの抽象化

本手法では, 特定の機能を実現するために必要になる複数の式計算や代入をまとめて抽象化し, 可読性の向上と記述量の削減を図る.

図 8 は X_SERV が 1 の時に, タスク番号 X_TASK の実行状態を TTS_RDY へ変更するための Verilog HDL 記述である. この処理を実行するためには 4 本のワイヤ SRW, SRX, SRT, SRWD に信号値を代入する必要があり, 1–23 行目でそれぞれのロジックを function 文で定義し, 25–28 行目で配線に値を代入している.

提案手法ではこの複数のロジックと代入をひとまとめにする. 例を 図 9, 図 10 に示す. 図 10 のように, write_status_regs という 1 つのメッソドで状態レジスタの更新に必要なロジックと代入を表現できる. 当該メソッドの定義は図 10 の通りである.

3.3 Python によるユニットテストの記述

本稿では設計したモジュールのユニットテストを Python を用いて記述し, Verilog HDL のテストベンチを自動生成する.

```
module sample_module(
  input [1-1:0] CLK, input [1-1:0] a, input [1-1:0] b,
  output [3-1:0] x, output [3-1:0] y
);

  reg [2-1:0] r0; reg [2-1:0] r1; reg [2-1:0] STATE;

  initial begin
    r0 = 2'd0;
    r1 = 2'd0;
    STATE = 2'd0;
  end

  always @(posedge CLK) begin
    case(STATE)
      0: begin
        if(a) begin
          r0 <= 1;
          STATE <= 2;
        end else begin
          r0 <= 2;
          STATE <= 1;
        end
      end
      1: begin
        if(b) begin
          r1 <= 1;
          STATE <= 2;
        end else begin
          r1 <= 2;
          STATE <= 0;
        end
      end
      2: begin
        STATE <= 0;
      end
    endcase
  end

  function [3-1:0] x_assign_func;
    input [1-1:0] CLK; input [1-1:0] a; input [1-1:0] b;
    begin
      case(STATE)
        0: begin
          x_assign_func = 0;
        end
        1: begin
          x_assign_func = 0;
        end
        2: begin
          x_assign_func = r0 + r1;
        end
      endcase
    end
  endfunction

  function [3-1:0] y_assign_func;
    input [1-1:0] CLK; input [1-1:0] a; input [1-1:0] b;
    begin
      case(STATE)
        0: begin
          y_assign_func = 0;
        end
        1: begin
          y_assign_func = 0;
        end
        2: begin
          y_assign_func = r0 * r1;
        end
      endcase
    end
  endfunction

  assign x = x_assign_func(CLK, a, b);
  assign y = y_assign_func(CLK, a, b);

endmodule
```

図 6: 状態機械の Verilog HDL 記述

本稿で対象とするハードウェアのユニットテストは, 時刻 t_1, t_2 ... で与えられた入力に対し, 時刻 s_1, s_2 ... に期待した出力が得られるか, により行えるものばかりである. したがって, 各時刻に対しての入力と期待出力だけを記述すれば良い.

テスト記述例を図 11 に示す. タスク数 n , ミューテックスのインスタンス数 m に対するテストを記述している. 17–41 行目が一つのテストケースであり, 0, 1, 2, ... 時刻における入力ポートへの入力値と出力ポートに期待される値を記述している.

この記述から, パラメータ m と n に応じた Verilog HDL のテストベンチを自動生成できる.

4. ハードウェア記述の生成結果

本手法により TOPPERS/ASP3 [12] 用の優先度上限ミューテックス, およびデーターキューのモジュールを設計した.

```python
import veriloggen as vg
from module_generator import ModuleGenerator


class SampleModule(ModuleGenerator):
  def module_generate(self):
    self.module = vg.Module("sample_mocule")

    self.declare_input_wires({"CLK": 1, "a": 1, "b": 1})
    self.declare_output_wires({"x": 3, "y": 3})
    self.declare_regs({"r0": 2, "r1": 2, "STATE" : 2})
    self.init_regs([
        (self.regs["r0"], 0),
        (self.regs["r1"], 0),
        (self.regs["STATE"], 0)
      ]
    )
    return self.generate_combinational_circuit(self.module, self.fsm())

  def fsm(self):
    return vg.Case(self.regs["STATE"])(
      vg.When(0)(
        vg.If(self.input_wires["a"])(
            self.regs["r0"](1),
            self.regs["STATE"](2),
        ).Else(
            self.regs["r0"](2),
            self.regs["STATE"](1)
        ),
        self.output_wires["x"](0),
        self.output_wires["y"](0),
      ),
      vg.When(1)(
        vg.If(self.input_wires["i1"])(
            self.regs["r1"](1),
            self.regs["STATE"](2),
        ).Else(
            self.regs["r1"](2),
            self.regs["STATE"](0),
        ),
        self.output_wires["x"](0),
        self.output_wires["y"](0),
      ),
      vg.When(2)(
        self.output_wires["x"](self.regs["r0"] + self.regs["r1"]),
        self.output_wires["y"](self.regs["r0"] * self.regs["r1"]),
        self.regs["STATE"](0)
      )
    )
```

図 7: Python による状態機械の記述

```
function [1-1:0] SRW_f;
  input [1-1:0] CLK; input [4-1:0] M;
  input [4-1:0] X_SERV;
  input [TASK_SIZE_BIT_LENGTH-1:0] X_TASK;
  input [32-1:0] X_A0;
  begin
    if(X_SERV == 1) begin
      SRW_f = 1;
    end
  end
endfunction

function [6-1:0] SRX_f;
  input [1-1:0] CLK; input [4-1:0] M;
  input [4-1:0] X_SERV;
  input [TASK_SIZE_BIT_LENGTH-1:0] X_TASK;
  input [32-1:0] X_A0;
  begin
    if(X_SERV == 1) begin
      SRX_f = 6'b101;
    end
  end
endfunction
...
assign SRW = SRW_f(CLK,M,X_SERV,X_TASK,X_A0);
assign SRX = SRX_f(CLK,M,X_SERV,X_TASK,X_A0);
assign SRT = SRT_f(CLK,M,X_SERV,X_TASK,X_A0);
assign SRWD = SRWD_f(CLK,M,X_SERV,X_TASK,X_A0);
```

図 8: タスクを実行状態に変更する Verilog HDL 記述

表 1: 記述量の比較

(a) ミューテックス

#instance	Python	Verilog	#LUT	#FF	delay [ns]
1	247	942	47	8	5.802
2		1387	83	12	6.029
3		1832	123	16	5.957

(b) データーキュー

#instance	Python	Verilog	#LUT	#FF	delay [ns]
1	214	787	79	10	5.595
2		1214	57	10	6.011
3		1649	121	31	5.812

いずれのモジュールも, 1 つのモジュールで複数のミューテックス/データーキューのインスタンスを管理する. Python による記述設計から, タスク数を 4 としてインスタンス数 1, 2, 3 の Verilog HDL 記述を生成し, Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成を行った. データーキューは各インスタンスが 32 ビットデータを 4 個まで保持する.

結果を表 1 に示す. "#instance" はインスタンス数, "Python" は Python プログラムの行数, "Verilog HDL" は生成された Verilog HDL の行数, "#LUT" はルックアップテーブル数, "#FF" はフリップフロップを表す. Python を用いてハードウェアを設計することにより, インスタンス数 1 の Verilog HDL よりも少ない行数で任意のパラメータに対応した設計を記述できている.

5. むすび

本稿では, リアルタイムシステムのフルハードウェア化において, ハードウェア設計およびテストを Python で記述する手法を提案した. Python で記述した Verilog HDL の AST に変換を導入することにより, 可読性の向上と記述量の削減を実現した. また, パラメータに依存したモジュールの単体テストを Python で記述し, そこから Verilog HDL のテストベンチを生成することにより, 設計したモジュールのテストの効率化を図った.

高位合成により得られるタスクと manager モジュールを結合したハードウェアのテスト記述とそこからのテストベンチ生成が今後の課題として挙げられる.

謝　辞

本研究に関して有益な御助言を頂いた元立命館大学の中谷嵩之氏, 元関西学院大学の田村真平氏, およびご支援を頂いた関西学院大学石浦研究室の諸氏に感謝致します. 科研費 21K19776, 24K14885 の助成による.

文　献

[1] Y. Cho, S. Yoo, K. Choi, N.-E. Zergainoh, and A. A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. ASP-DAC 2005*, pp. 151–156 (Jan. 2005).

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. RSP '06*, pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. CODES+ISSS '03*, pp. 45–51 (Oct. 2003).

```python
from module_generator import ModuleGenerator
import veriloggen as vg
from var_enum import StatusRegister

def fsm(self):
  return vg.If(self.input_wires["X_SERV"] == 1)(
    self.write_status_regs(
      self.input_wires["X_TASK"], {
        StatusRegister.AccessItem.Tskstat: StatusRegister.Tskstat.TTS_RDY
      }
    )
  )
```

図 9: Python によるタスクの状態更新の記述

```python
class StatusRegister:
  class AccessItem(Enum):
    Tskstat = 0
    Tskpri = 1
    ...

  class Tskstat(Enum):
    TTS_RUN = 0 #実行可能
    TTS_RDY = 1 #実行可能状態
    ...

  class Tskwait(Enum):
    TTW_SLP = 0 #起床待ち
    TTW_DLY = 1 #時間経過待ち
    ...

def write_status_regs(self, task_number : any,
     access_item_dict : dict[StatusRegister.
     AccessItem, any]):
  if StatusRegister.AccessItem.Tskstat in
       access_item_dict:
    assign_value = access_item_dict[StatusRegister.
         AccessItem.Tskstat]
    if isinstance(assign_value, StatusRegister.Tskstat
         ):
      access_item_dict[StatusRegister.AccessItem.
           Tskstat] = self.local_params[assign_value]

  if StatusRegister.AccessItem.Tskwait in
       access_item_dict:
    assign_value = access_item_dict[StatusRegister.
         AccessItem.Tskwait]
    if isinstance(assign_value, StatusRegister.Tskwait
         ):
      access_item_dict[StatusRegister.AccessItem.
           Tskwait] = self.local_params[assign_value]

  assign_statement_list = [
    self.output_wires["SRX"](self.get_srx(
         access_item_dict)),
    self.output_wires["SRW"](1),
    self.output_wires["SRT"](task_number),
  ]
  for access_item, value in access_item_dict:
    assign(access_item(value))

  return assign_statement_list
```

図 10: `write_status_regs` の定義

```python
from testbench_generator import TestbenchGenerator

test_target = test_bench.read("terget.v")

n = 4
m = 3

in_port = {
  "X_SERV": 4,  "M": 4,
  "X_A0": 32,  "X_TASK": n.bit_length()
}

out_port = {
  "sa0":32,  "srx": 6,  "srwd": 51
}

test_case0 = {
  "description" : "lock mutex",
  0 : {
    "X_SERV":"2",  "M":"1",
    "X_A0":0, "X_TASK":n-1
  },
  1: {
    "sa0":"0", "srx":"001000",
    "srwd":"0320"
  },
  2 : 省略...
}

test_case1 = {
  "description" : "lock mutex",
  0 : {
    "X_SERV":"2", "M":"1",
    "X_A0":m-1, "X_TASK":0
  },
  1: {
    "sa0":"0", "srx":"001000",
    "srwd":"0320"
  },
  2 : 省略...
}

test_cace2 = ...
create_testbench(test_case0, test_case1, test_case2)
```

図 11: Python によるユニットテスト記述

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," in *IEICE Trans. Fundamentals*, vol. E82-A, no. 11, pp. 2375–2382 (Nov. 1999).

[5] N. Maruyama, T. Ishihara, and H. Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. SASP 2010*, pp. 58–63 (June 2010).

[6] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of full hardware implementation of RTOS-based systems," in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[7] 六車 伊織, 石浦 菜岐佐, 安堂 拓也, 冨山 宏之, 神原 弘之: "RTOS 利用システムのフルハードウェア化におけるサービス処理機能の集約," 信学技報, VLD2020-75 (Mar. 2021).

[8] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama, and H. Kambara: "Full hardware implementation of RTOS-based systems using general high-level synthesizer," in *Proc. SASIMI 2022*, pp. 2–7 (Oct. 2022).

[9] H. Minamiguchi, N. Ishiura, H. Tomiyama, and H. Kanbara: "Automatic generation of management module for full hardware implementation of RTOS-based systems," in *Proc. ITC-CSCC 2023*, pp. 473–478 (June 2023).

[10] M. Nakahara and N. Ishiura: "Arrival order processing of service requests in full hardware implementation of RTOS-based systems," in *Proc. ITC-CSCC 2023*, pp. 467–472 (June 2023).

[11] Shinya Takamaeda-Yamazaki: "Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL" in *Proc. ARC 2015*, pp. 451–460 (Apr. 2015).

[12] TOPPERS project: "TOPPERS 第 3 世代カーネル (ITRON 系) 統合仕様書 Release 3.6.0," `https://www.toppers.jp/docs/tech/tgki_spec-360.pdf` (Mar. 2023).