RTOS 利用システムのフルハードウェア化における サービス要求の到着順待ち解除

中原 正樹[1,a)]　石浦 菜岐佐[2]

概要：本稿では, RTOS を用いたシステムのフルハードウェア実装において, タスクのサービス待ちを到着順に解除する手法を提案する. リアルタイムシステムの応答性能を向上させる手法として, タスク/ハンドラおよび RTOS の機能を全てハードウェア化する手法が提案されている. 六車・安堂らのアーキテクチャでは, タスク間の同期・通信を複数のタスクが待っている場合, その待ちの解除 (サービスの獲得) はタスクの優先度の順に行われているが, RTOS やサービスによっては到着順の待ち解除が仕様になっている場合がある. 本稿では, 待ち解除の順序を同期・通信サービスのインスタンス毎に優先度順・到着順のいずれかに切り替えられる方法を提案する. 到着順の待ち解除を実現するために, 待ち解除の処理を他のサービス要求よりも先に実行するようにするとともに, 到着順を記録するハードウェアを実装する. 本手法に基づく管理ハードウェアを Verilog HDL で設計し, Xilinx Vivado 2020.2 を用いて FPGA (Xilinx Artix-7) をターゲットに論理合成した結果, 従来に比べて LUT 数が 4.8 %, FF 数が 5.4 %, クリティカルパス遅延が 23.3 % の増加で待ち解除順の切り替えを実現することができた.

キーワード：システム設計技術, リアルタイムシステム, RTOS, システム合成, ハードウェアアクセラレータ

Arrival Order Release of Wait of Service Requests in Full Hardware Implementation of RTOS-Based Systems

Nakahara Masaki[1,a)]　Ishiura Nagisa[2]

Abstract: This article proposes a method for releasing tasks' waiting in the arrival order in full hardware implementation of RTOS-based systems. A method for implementing both tasks/handlers and RTOS functions as hardware has been proposed as a measure to improve response performance of real time systems. Though tasks' waiting for RTOS services is released only in the order of the task priorities in the architecture proposed by Muguruma and Ando, some RTOS assumes release of waiting in the order of their arrival. This paper proposes a mechanism to realize both priority order release and arrival order release of tasks' waiting for services. For this purpose, the release of waiting requests is processed prior to the other requests, and a hardware module to record the arrival order of requests is implemented. Based on the proposed method, a management hardware that provides the functions of TOPPERS/ASP3 has been designed in Verilog HDL, which is synthesized using Xilinx Vivado (2020.2) targeting FPGA (Xilinx Artix-7). The new feature has been successfully implemented, at the cost of 4.8% and 5.4% increase in LUT count and FF count, respectively, and 23.4% increase in critical path delay.

Keywords: system design technology, real-time system, RTOS, system synthesis, hardware accelerator

1. はじめに

近年, 情報通信技術の発展により, 組み込みシステムには様々なサービスを提供するために益々高い機能が要求され

[1] 関西学院大学 大学院理工学研究科
Graduate School of Science and Technology, Kwansei Gakuin Univ.
[2] 関西学院大学 工学部
School of Engineering, Kwansei Gakuin Univ.
[a)] hjw14549@kwansei.ac.jp

るようになっている. 特に, 車載機器, 無人飛行機, ロボットの制御には高い機能とともに高い応答性能が要求される. このようなシステムはリアルタイム OS (RTOS) を用いて設計される. RTOS は入力イベントに対して予め指定された時間内にタスクの実行を完了するようにシステムを設計するための機能を提供する. しかし, システムの高機能化が進むにつれてその応答性能の確保は難しくなりつつある.

RTOS を用いたシステムの応答性能を向上させる手法として, RTOS の機能の一部または全てをハードウェア実装する方法が提案されている. 文献 [1] [2] [3] では RTOS のスケジューラのハードウェア化を行っており, 文献 [4] [5] では RTOS のほとんどの機能をハードウェア実装している. しかし, これらの手法ではタスクやハンドラはソフトウェアで実装されており, CPU 待ちやコンテキストスイッチによるオーバーヘッドが発生する.

この課題を解決する一手法として, 文献 [6] はタスク/ハンドラおよび RTOS のサービス機能全てをハードウェア化する手法を提案している. タスクはそれぞれ独立に動作するハードウェアに合成され, 実行可能状態のタスクは全て並列に実行できるため, CPU 待ちやタスク切り替えのオーバーヘッドが無い. 文献 [7] [8] では, 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タスク管理ハードウェア側に集約することによって, 回路規模の削減を試みている.

文献 [7] [8] のアーキテクチャではタスク間の同期・通信サービスを複数のタスクが待っている場合, その待ちの解除 (サービスの獲得) はタスクの優先度の順に行われている. しかし RTOS によっては到着順に待ち解除を行うサービスや, 到着順か優先度順かを選択して待ち解除を行うサービスがある.

本稿では文献 [7] [8] のアーキテクチャにおいて, 優先度順の待ち解除に加えて, 到着順の待ち解除を行う手法を提案する. 到着順の待ち解除を可能にするため, 待ち解除を通常の処理よりも優先して行うようにし, 到着順を記録するハードウェアモジュールを実装する.

本手法に基づく管理ハードウェアを FPGA をターゲットに実装した結果, 手法 [8] と比べて, LUT 数が 4.8 %, FF 数が 5.4 %, クリティカルパス遅延が 23.3 % の増加で待ち解除順の切り替えを実現することができた.

2. RTOS を用いたシステムのフルハードウェア実装

2.1 概念

文献 [6] [7] [8] では, RTOS 機能を利用したプログラムを入力とし, これを実行するプロセッサと機能等価なハードウェアを合成する手法を提案している. この手法の概念を図 1 に示す. 左側のタスク (task$i$) は RTOS の管理の下で CPU により実行される. これらのタスクは高位合成により右側の独立したハードウェアモジュール Ti に合成される. マネージャ (manager) は RTOS の機能をハードウェア化したものであり, 各タスクの状態に基づいてその実行/停止を制御する信号を生成するとともに, タスク間の同期・通信等のサービスを提供する.

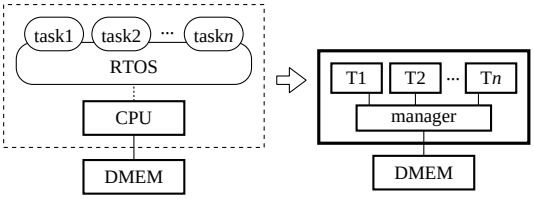


図 1: RTOS を用いたシステムのフルハードウェア実装

この手法で扱えるタスク数は 16 程度までに限られるが, タスクはハードウェア化されている上, 実行可能状態のタスクは全て並列に実行されるため CPU 待ちがない. またタスクスケジューリングやコンテキストスイッチによるオーバーヘッドがないため従来手法に比べてシステムの応答性能を大幅に向上させることができる.

2.2 六車・安堂のアーキテクチャ

文献 [7] [8] で提案されているアーキテクチャを図 2 に示す. T0 ∼ T2 はタスクモジュールである. S0 ∼ S3 は mutex や eventflag 等の RTOS のサービスを実行するサービスモジュールである. サービス間の干渉を避けるため, 一度に実行できるサービスは一つのみとしている. このため複数のタスクが同時にサービスを要求した場合は Request Arbiter (RA) がタスクの優先度に基づいて調停を行う. STATUS にはタスクの状態と優先度等が記憶されている. WAIT はどのタスクがどのサービスを待っているかを管理するモジュールである. 実行可能状態のタスクは全て並列に実行される. manager は STATUS にある各タスクの状態からタスクの実行/停止を制御する信号を生成する.

タスク T1 がサービスモジュール S2 (例えば mutex モジュール) にサービス i (例えば mutex のロック獲得) を要求した場合の処理の流れは次のとおりである.

(1) タスク T1 は TF_1 と TA_1 にサービス i の ID と引数をセットすることによりサービスを要求し, サービスの完了を待つ (サービス完了まで実行を停止する).

(2) RA は最も優先度が高いタスクの番号を XT に, TF_1 と TA_1 をそれぞれ XF, XA に書き込んでサービスモジュール S2 を起動する.

(3) S2 は要求された処理を実行し, XA に返り値を書き込む.

(4) RA はそれを TA_1 に転送して T1 に完了を知らせる.

(5) T1 は TA_1 から返り値を読み込んで実行を再開する.

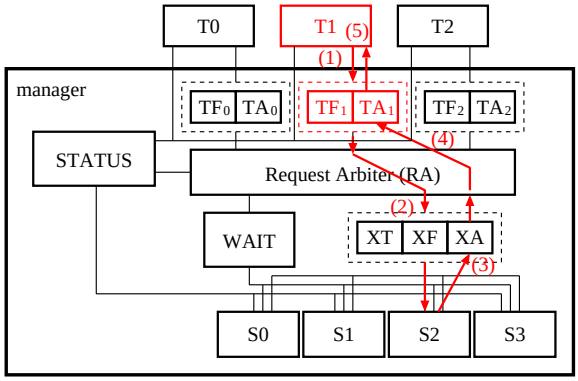


図 2: 文献 [7] [8] のアーキテクチャ

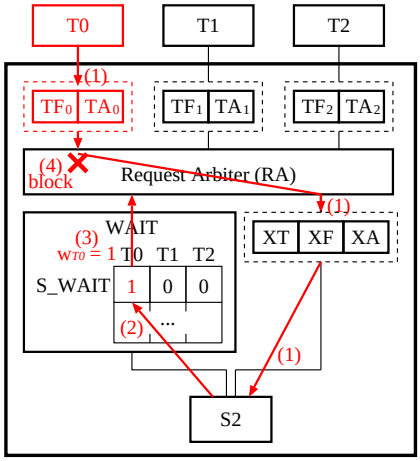


図 3: サービス待ちの処理の流れ

2.3 タスクのサービスの待ちとその解除

2.3.1 タスクのサービス待ち

RTOS のサービスの中には, タスクが要求したサービスを実行できない場合にタスクを待たせるものがある. 例えば, 他のタスクが獲得している mutex を要求した場合や, 空の queue からデータを受信しようとした場合, そのタスクは当該 mutex が解放されたり queue がデータを受信するまで待ち状態になる.

文献 [7] [8] のアーキテクチャでは, 待ちタスクをリスト等で管理するのではなく, 各タスクが待ち状態にあるかどうかを表すフラグと RA を利用してタスクの待ちを実現している.

タスクをサービス待ち状態にする処理の流れを図 3 に示す.

(1) タスク T0 がサービスモジュール S2 にサービス i を要求するが, それは直ちには実行できなかったとする.

(2) S2 は WAIT 内のフラグレジスタ S_WAIT[T0][i] をセットする. S_WAIT[t][s] はタスク t がサービス s を待っていることを表す.

(3) WAIT は RA に対して T0 がいずれかのサービスを待っていることを表す信号 w_{T0} を出す. w_{T0} の値は S_WAIT[T0][$*$] の論理和でありこの場合, $\mathrm{w}_{T0} = 1$ となる.

(4) RA は $\mathrm{w}_t = 1$ であるタスク t の要求をそのタスクの優先度に関わらずブロックする (要求を XT, XF, XA に書き込まないようにする). これにより, タスク T0 が TF_0, TA_0 に書き込んだ要求は処理されないまま待たされることになる.

2.3.2 サービス待ちの解除

サービスが実行できない原因が解消された場合, タスクのサービス待ちは解除される. 複数のタスクが同じサービスを待っていた場合, 待ちの解除は文献 [7] [8] のアーキテクチャではタスクの優先度順に行われる. この優先度順の待ち解除は RA の優先度による調停を利用して行われる.

タスクのサービス待ち解除の流れを図 4 に示す. (a) に示すようにタスク T0, T1 がサービスモジュール S2 の提供するサービス i を待っているとする. 各タスクの優先度は T0, T1, T2 の順に高いものとする.

(1) S2 はこの待ちを解除する場合, 一旦 i を待っている全てのタスクについて待ちを解除する. これは S_WAIT[$*$][i] を全てクリアすることにより行われる.

(2) (1) によって, WAIT から RA に対して $\mathrm{w}_{T0} = 0$, $\mathrm{w}_{T1} = 0$ が送出され, i を待っていた T0 と T1 の要求のブロックが解除される.

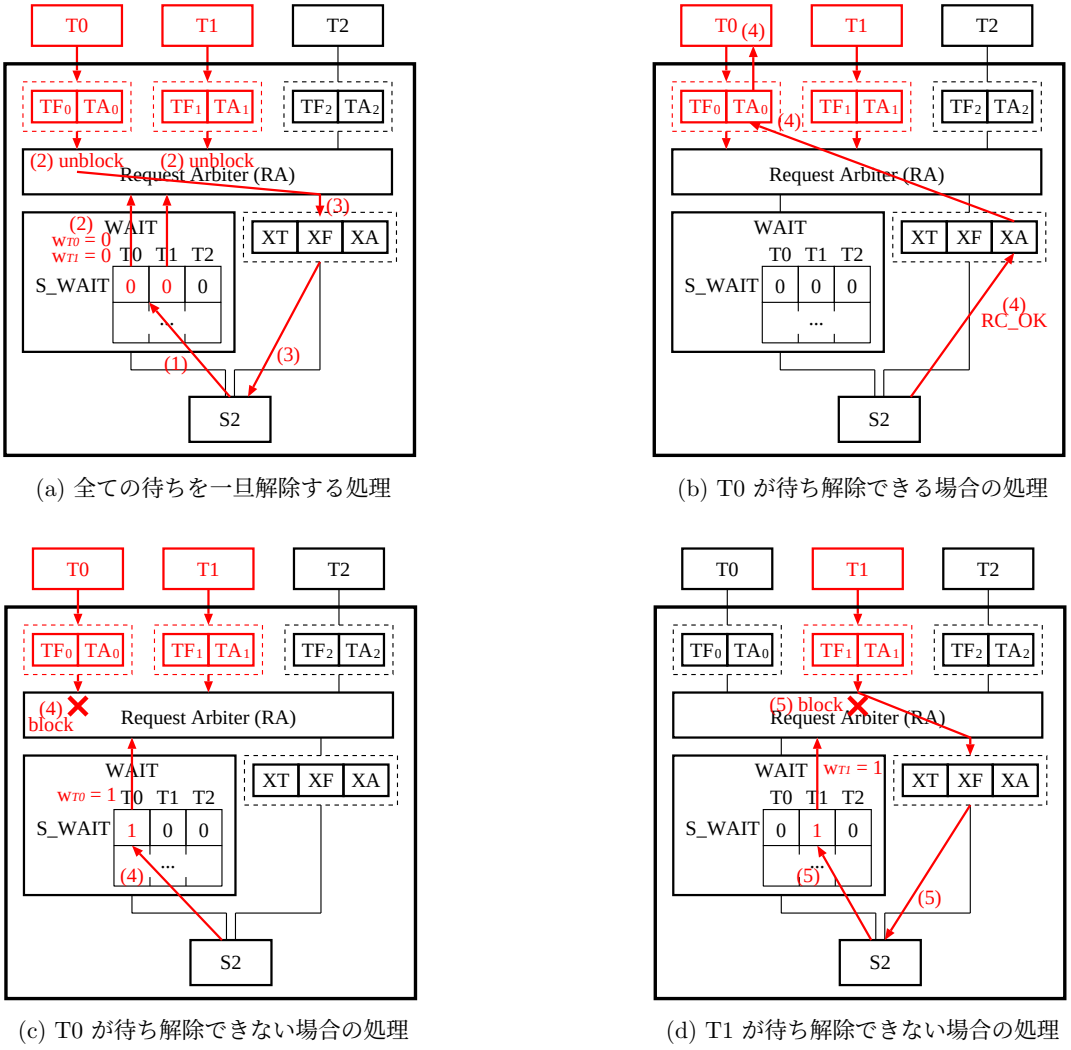
(3) RA は優先度が最も高いタスクの要求をサービスモジュールに送るので, T0 の要求が XT, XF, XA に書き込まれ S2 に伝えられる.

(4) S2 で T0 の待ちが解除できた (例えば T0 が新たに mutex を獲得できる等) 場合, 図 4 (b) のように S2 は XA に正常終了の返り値 (RC_OK) を書き込んで完了を通知する. T0 は TA_0 から返り値を読み込んで実行を再開する. T0 の待ちが解除できなかった (eventflag でセットされたフラグが条件に一致しない等) 場合, 図 4 (c) のように S2 は再び S_WAIT[T0][i] をセットして T0 を待ち状態にする.

(5) T0 の待ちが解除されるか否かに関わらず, 図 4 (d) のようにタスク T1 の要求が S2 に伝えられる. T1 が待ち解除できなかった場合, S2 は再び S_WAIT[T1][i] をセットして T1 を待ち状態にする.

待ちタスクを一旦一斉に解除する理由は, RA で優先度最大のタスクを選択するため, および eventflag のように条件を満たす複数タスクの待ちを解除するサービスがあるためである.

このアーキテクチャでは RA の優先度調停を利用しているため優先度順の待ち解除しかできないが, RTOS によっては到着順あるいは両方の順序で待ち解除を行うサービスがある. 例えば, TOPPERS/ASP3 の message buffer は到着順の待ち解除を行い, mutex や eventflag はインスタンス毎に待ち解除が到着順か優先度順を指定できる. さらに, 優先度順に待ち解除を行うサービスであっても, 同じ優先度のタスクが複数あった場合にはその中で到着順に待ち解



(a) 全ての待ちを一旦解除する処理

(b) T0 が待ち解除できる場合の処理

(c) T0 が待ち解除できない場合の処理

(d) T1 が待ち解除できない場合の処理

図 4: サービス待ちの解除処理の流れ

除を行う仕様のため, これらへの対応が必要である.

また, このアーキテクチャでは RA において単純に優先度だけを用いて調停を行っているため, 待ちに関係しないタスクで優先度の高いものがサービスを要求した場合に, そちらの処理が優先されるため, 待ちの解除が遅れてしまうことも課題である.

3. サービス要求の到着順待ち解除

3.1 概要

本稿では, 文献 [7] [8] のアーキテクチャにおいて, サービス要求の待ち解除を優先度順だけでなく到着順でも行える手法を提案する. 待ち解除を優先度順, 到着順どちらで行うかはサービスのインスタンス毎に指定できるようにする.

待ち解除の処理と通常の調停処理を明確に分離するために, 待ち解除の処理を他のサービス要求よりも先に完了させるためのモードを RA に設ける. RA は優先度順, 到着順のうち指定された順に待ち解除を行えるようにする. この到着順の待ち解除に必要になる到着順を記録するモジュールを設ける.

本手法のシステム全体のアーキテクチャを 図 5 に示す. 文献 [7] [8] のアーキテクチャに加えて, 到着順記録モジュール (ARRIVAL) を追加している. ARRIVAL はタスクが manager にサービスを要求した順序を記録するモジュールであり, サービスの要求とその処理の完了毎にその情報を更新する.

サービスモジュールが WAIT にタスクの待ち解除を指示すると, WAIT はどのタスクの待ちを解除するか, および待ち解除を優先度順で行うか到着順で行うかを伝えて RA を待ち解除モードにする. RA は指定された待ちの解除を行い, 必要な全ての待ち解除を終えると通常のモードに戻る.

3.2 RA の待ち解除モード

本稿における WAIT の構成を図 6 に示す. 文献 [7] [8] の WAIT 内にはどのタスクがどのサービスを待っているかを表す二次元フラグ (S_WAIT) のみであったが, 新たに

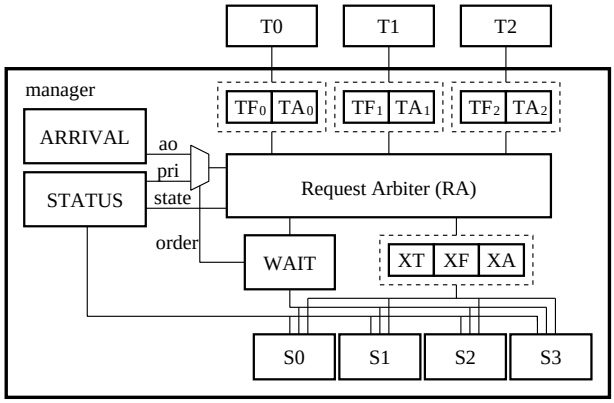


図 5: 本稿のアーキテクチャ

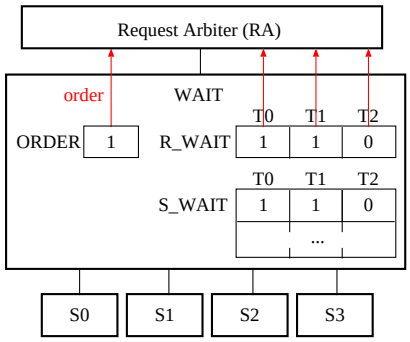


図 6: WAIT モジュール

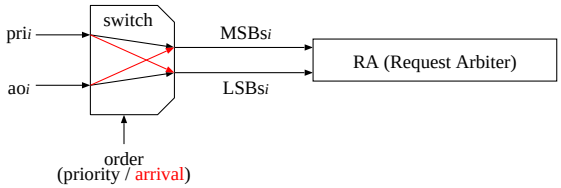


図 7: タスクの優先度と到着順の切り替え

待ち解除するタスクを表す一次元フラグ R_WAIT と, どちらの順序で待ち解除するかを表す 1 ビットのレジスタ ORDER を追加する. R_WAIT[t] はこれからタスク t の待ち解除を行うことを表す.

RA は R_WAIT[$*$] が一つでも 1 であれば待ち解除モードになる. 待ち解除モードの RA は R_WAIT[$*$] が 1 のタスクの中で待ち解除の順位が一番高いタスクの要求のブロックを解除する, という操作を繰り返し, R_WAIT[$*$] が全て 0 になれば通常モードに戻る.

サービスモジュール S がインスタンス i を待つタスクの待ち解除を行う場合の処理の流れは次のとおりである.

(1) サービスモジュール S は WAIT に対してインスタンス i およびどちらの順序で待ち解除をするかを指定して待ち解除を指示する.

(2) WAIT はインスタンス i を待っている全てのタスクの待ちを解除するために, S_WAIT[t][i]$=1$ となっている全てのタスク t について R_WAIT[$*$] をセットする. また, サービスモジュール S が指定した待ち解除の順序を ORDER にセットする. WAIT は RA に対して R_WAIT の値を送出して RA を待ち解除モードにするとともに ORDER を order に送出する.

(3) 待ち解除モードの RA は R_WAIT がセットされているタスクのうち指定された順序が一番のタスク t の要求をサービスモジュール S に送る.

(4) サービスモジュール S はタスク t からの要求を受け取ると, 具体的な待ち解除処理を行い, WAIT にタスク t が待ち解除できるか否かを伝える. タスク t の待ちが解除できる場合, WAIT は S_WAIT[t][i] と R_WAIT[t] をクリアし, タスク状態を待ち状態から実行可能状態にする. タスク t の待ちが解除できない場合には, R_WAIT[t] のみをクリアする. これによりタスク t の要求は再び RA でブロックされる.

待ち解除するタスクの数はサービス毎に異なる. mutex や CLR 属性の eventflag 等, 多くのサービスは待ちタスクを一つだけ解除する. この場合, サービスモジュールはその待ち解除が終わった時点で R_WAIT[$*$] を全てクリアし, RA を通常モードに戻す. これに対し, CLR 属性のない eventflag の場合は, 全ての待ちを解除するまで待ち解除の操作を継続する.

3.3 優先度順および到着順の待ち解除

待ち解除モードの RA は R_WAIT[$*$] の論理和が 1 の間, WAIT から order で指定された順に待ち解除を行う.

待ち解除の順序変更は, RA への各タスクの優先度入力をタスクの到着順に切り替えることにより行う. このためのハードウェア構成を図 7 に示す. MSBs$_i$ と LSBs$_i$ はそれぞれ RA が調停に用いる各タスクの優先度の上位ビットと下位ビットである. order が priority の場合にはタスクのカレント優先度 pri$_i$ と 到着順 ao$_i$ をそれぞれ MSBs$_i$, LSBs$_i$ に出力し, order が arrival の場合にはその逆にする. これにより RA は優先度順の待ち解除において, 同じ優先度のタスクがあった場合には到着順で待ち解除を行うことができる.

3.4 到着順記録モジュール

サービス要求の到着順の記録に関しては, できる限り少ないビット数で到着順の情報を記録するために, サービスが到着した時刻ではなく順序のみを記録する. k 個のタスクがサービスの完了を待っている場合, 0 から始まる連続した整数 $0, 1, 2, \ldots, k-1$ で順番を記録する. 待っていないタスクの順は -1 とする. この到着順はタスクがサービスを要求, 完了する毎に更新する.

ARRIVAL モジュールの構成を 図 8 に示す. ORDER は各タスクの到着順を記録する一次元配列であり, MO は到着順の最大値を記憶するレジスタである.

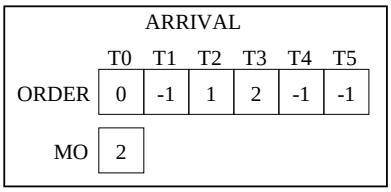


図 8: 到着順記録モジュール

タスク t がサービスを要求すると, MO をインクリメントし, ORDER[t] に MO をセットする. タスク t が要求したサービスが完了すると, タスク t より高い ORDER を全てデクリメントするとともに MO をデクリメントし, ORDER[t] を -1 にする. 例えば, 図 9 (a) に示すように, T4 がサービスを要求すると, MO を $2 \rightarrow 3$ にし, ORDER[T4] に 3 をセットする. 次に T2 がサービスを完了すると, 図 9 (b) のように ORDER[T2] より高い ORDER[T3], ORDER[T4] をそれぞれ 1, 2 にするとともに, MO を $3 \rightarrow 2$ にし, ORDER[T2] に -1 をセットする.

複数のタスクが同時にサービスを要求すると, 要求するタスク数を MO に加算し, タスク番号が小さい順に ORDER を記録する. 例えば, 図 9 (c) に示すように, T1 と T5 が同時にサービスを要求すると, MO を $2 \rightarrow 4$ にし, タスク番号が小さい順に ORDER[T1] に 3, ORDER[T5] に 4 をセットする.

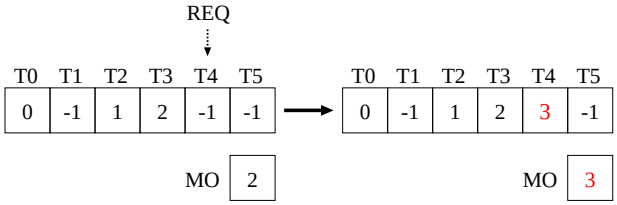
タスクの要求と同時にタスクが完了した場合には, 完了するタスクより高い ORDER を全てデクリメントし, 完了するタスクの ORDER を -1 にする. また要求するタスクの ORDER に MO をセットする. 例えば, 図 9 (d) に示すように, T2 がサービスを要求するとともに T4 がサービスを完了すると, ORDER[T4] より高い ORDER[T1], ORDER[T5] をそれぞれ 2, 3 にし, ORDER[T4] に -1 をセットする. また ORDER[T2] に MO をセットする.

4. 実装と実験

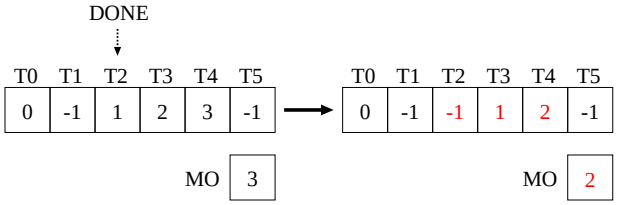
提案した手法に基づき, サービスのインスタンス毎に優先度順・到着順を指定して待ち解除を行える管理ハードウェア (manager) を設計した. manager 内のサービスモジュールは eventflag モジュールのみ実装し, フラグのセットを要求する一つのタスクとフラグ待ちを要求する三つのタスクにより manager の動作確認を行った.

manager および eventflag モジュールは Verilog HDL で記述し, タスクは高位合成系 Xilinx Vitis HLS 2020.2 によりハードウェア記述を自動生成した. これらのハードウェア記述を Xilinx Vivado 2020.2 で Xilinx FPGA Artix-7 (xc7a100tcsg324-3) をターゲットに論理合成した.

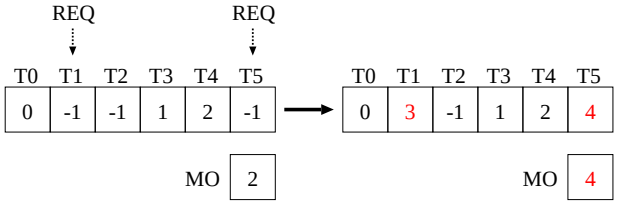
合成した各モジュールの回路規模を表 1 に示す. #LUT はルックアップテーブル数, #FF はフリップフロップ数である. 手法 [8] と比べて LUT 数が 4.8 % 増, FF 数が 5.4 % 増と小さいオーバーヘッドで到着順の待ち解除機能を追加することができた.



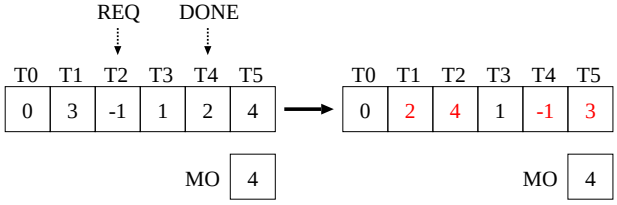
(a) 一つのサービス要求



(b) 一つのサービス完了



(c) 同時に二つのサービス要求



(d) 同時に一つのサービス要求と完了

図 9: 到着順の記録

表 1: 回路規模

	手法 [8]		本手法	
module	#LUT	#FF	#LUT	#FF
manager	544	407	611	433
eventflag	206	64	186	64
task	12	11	12	11
total	772	482	809	508

表 2: クリティカルパス遅延

	手法 [8]	本手法
delay[ns]	5.604	6.907

合成したシステムのクリティカルパス遅延を表 2 に示す. 手法 [8] に比べて, 23.3 % の増加となった. この増加はタスクの到着順を記録する回路や RA の順序比較のビット数が増えたことが原因であると考えるが, 依然高速である.

5. おわりに

本稿では, RTOS を利用したシステム全体のハードウェア化におけるサービス要求の到着順待ち解除を行う手法を提案した. 待ち解除を先に行う手法, 到着順記録モジュールの実装方式および到着順記録を用いて優先度順, 到着順両方の待ち解除を行う手法を提案した.

本手法に基づき, 優先度順と到着順の待ち解除を行える管理ハードウェアを論理合成ツール Vivado 2020.2 および高位合成システム Xilinx Vitis HLS を用いて実装できることを確認した. また回路規模の増加を約 5 % に抑えることができた.

現時点では, このアーキテクチャの manager の設計は手動で行っているため, 今後はその自動生成を実装する予定である. また, タスク数が増えた場合の回路規模やクリティカルパス遅延の評価や, それに基づく回路の最適化が今後の課題である.

謝辞 本研究に関して有益な御助言を頂いた京都高度技術研究所の神原弘之氏, 立命館大学の冨山宏之氏, 元立命館大学の中谷嵩之氏, 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. 本研究の一部 JSPS 科研費 19H04081 の助成による.

参考文献

[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).

[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] 安堂 拓也, 石井 雄吾, 石浦 菜岐佐, 冨山 宏之, 神原 弘之: RTOS 利用システムの汎用高位合成系を用いたフルハードウェア化, 信学技報, VLD2021-51 (Jan. 2022).