卒 業 論 文

割込み駆動のモータ制御プログラムからのバイナリ合成

関西学院大学 理工学部 情報科学科

27013522 大 迫 裕 樹

2017年3月

指導教員 石浦 菜岐佐 教授

卒業論文

割込み駆動のモータ制御プログラムからのバイナリ合成

大迫 裕樹

内容梗概

本研究では, モータの伝達関数モデルを用いた割込み駆動のセンサレス制御プログラムを, バイナリ合成システム ACAP を用いてハードウェア化する手法を提案する. 高位合成技術により, C 言語等で書かれたプログラムをハードウェア化できるが, 一般的な高位合成システムでは, 割込み処理を含むプログラムをそのままハードウェア化することはできない. そこで, 本研究では ACAP を用いて, 割込み駆動の制御プログラムからそれと等価なハードウェアを自動生成する. モータの制御プログラムを C 言語で記述し, コンパイルして得られる目的コードを必要なライブラリとリンクして, MIPS の機械語を生成する. それを ACAP へ入力することにより, 割込み信号で駆動される制御ハードウェアを自動生成する. ブラシ付き DC モータのセンサレス PID 制御プログラムを合成し, 制御システムが合成できることを確認した. 実験では, 約 21 µs 間隔で割込みをかけて PID 制御のシミュレーションを行うことができた. これは, MIPS ソフトコアプロセッサ比で約 18 倍の高速化である. また, 消費電力量において 66.9 % の削減が見込まれる. 更に, 本手法を用いることにより, 他の制御方法やモータモデルを利用する場合においても, 同様に合成することが可能である.

キーワード

高位合成, バイナリ合成, 割込み, モータ制御

目 次

第1章　序論

近年, モータは産業機器, ロボット, 洗濯機や冷蔵庫等の家庭用電気製品, 冷却ファンやハードディスクドライブを内蔵した PC 等, 幅広い製品で使用されている. また, 内燃機関を動力源としてきた自動車, 建設機械あるいはドローン等の輸送機器の電動化が急速に進みつつある. モータは日本国内の総電力消費量の 57.3%を消費しており [1], 省エネルギー化を達成するために, さらなるモータ制御の高度化が必要とされている.

モータのフィードバック制御では, センサから取得した回転速度や位置 (角度) 情報を元に PWM (pulse width modulation) 等でモータの駆動電力を調整し, その回転速度やトルクを目標値に近付ける. これに対し, センサ情報を利用しないセンサレス方式では, 制御系側でモータの物理的なモデルをシミュレーションすることによって速度やトルクを推定し, フィードバック制御を行う. モータのふるまいのシミュレーションには浮動小数点演算を多用するが, マイコン搭載の浮動小数点演算器の能力が限られていることもあり, 制御の高度化が進むに伴い, 必要とされる制御周期を実現できない状況が生じつつある [2].

処理の高速化を達成するアプローチの一つに, 再構成可能な集積回路である FPGA (field programmable gate array) 等を用いて制御処理をハードウェア化する方法が考えられる. FPGA が搭載している多数の乗算器を再構成することによって浮動小数点演算を並列に実行し, より短い制御周期でモータのふるまいを推定し, 制御することが可能になる.

C 言語等で書かれた制御プログラムを FPGA に実装可能なハードウェア回路に変換することは, 高位合成技術 [3] により比較的容易になってきている. しかし, 一般的な高位合成技術では, マイコンのアセンブリ言語で記述された割込み処理プログラムを合成することはできず, タイマやセンサからの割込みに基づく制御プログラムをそのままハードウェア化することはできない.

そこで本研究では, 割込み処理を含むプログラムを合成可能なバイナリ合成システム ACAP [4] を用いることにより, 割込み駆動のモータ制御をハードウェア実装する手法を提案する [5]. タイマ割込みにより駆動されるブラシ付き DC モータのセンサレス制御を, C 言語と MIPS のアセンブリ言語により記述し, これを ACAP で FPGA 上の回路に合成する.

これにより, 約 21 µs の周期で PID 制御が可能であり, その結果, CPU に比べて高精度な制御が実現できることも確認した. またこれに伴い, 省電性能の向上も見込まれる.

第2章　モータの制御とモータの動作モデルのバイナリ合成

2.1　モータのセンサレス制御

モータのセンサレス制御とは, モータの状態を検出するセンサを取り付けず, 制御に必要な回転数や磁極位置等を推定して駆動する制御方式である. どの部分をセンサレス化するかに関しては種々の方式があり, 回転子位置検出センサを省略する場合や回転数の検出センサを省略する場合等がある. センサレス制御は, 実際にエアコンや冷蔵庫のコンプレッサや産業用, 車載用モータ等に使用されている. センサレス化の背景には, センサがデバイスの小型化の妨げになることや, 小型化に伴って配線の物理的制約が厳しくなることが挙げられる. また, 制御によっては高価なセンサが必要になることもあるため, これを省略することによりコストを削減することも可能である. その他にも, 環境に依存するセンシングのノイズの影響を受けず, ロバスト性が向上するという長所もある.

一方で, センサレス制御においては, 実測値に基づく制御と比較して精度に劣るため精密な制御が困難であることや, ソフトウェア側に高速かつ複雑な演算が要求されるという課題がある.

モータのセンサレス制御の流れを図 2.1 に示す. この図では, タイマからの制御信号を受けてコントローラが動作し, それに従って実モータを駆動する電力を決定する. それと同時に, 入力電力値はモータシミュレータにも渡される. シミュレータはこれに基づいて実モータの状態を推定し, 推定結果値をフィードバックとしてコントローラに返す. コントローラは, このフィードバック値を用いて次の制御演算を行う. この図のうち, 制御システムに相当するのは破線内部である.

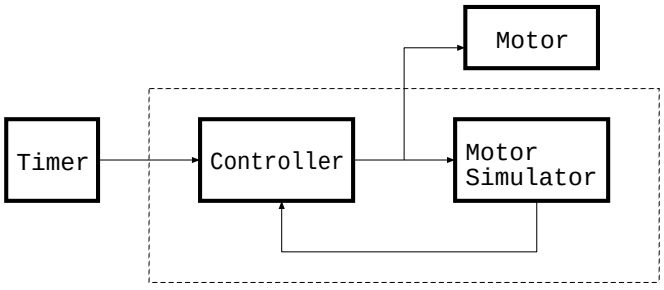


図 2.1: モータのセンサレス制御

2.2 PID 制御

制御手法の 1 つとして PID 制御がある [6]. これはフィードバック制御の一種であり, 制御対象のシステムへの入力値を, 出力値と目標値との偏差 (P 要素), 偏差の積分 (I 要素), 偏差の微分 (D 要素) の 3 つの要素によって決定するものである.

PID 制御を行う際のシステム全体の構成図を, 図 2.2 にブロック線図形式で示す. $G(s)$ が制御対象のシステムであり, 破線内部が PID 制御を行う部分である. $R(s)$ は目標入力, $E(s)$ は偏差である. これらの値に対し, 適切なシステムの応答 (制御出力) $Y(s)$ が得られるような制御入力 $X(s)$ を決定する. 制御入力の決定には P 要素, I 要素, D 要素 の 3 つの要素があり, 各要素を制御するパラメータである比例ゲイン K_P, 積分ゲイン K_I, 微分ゲイン K_D を適切に調整することにより, 応答時間, 定常偏差, 振動, オーバーシュート等を抑制する.

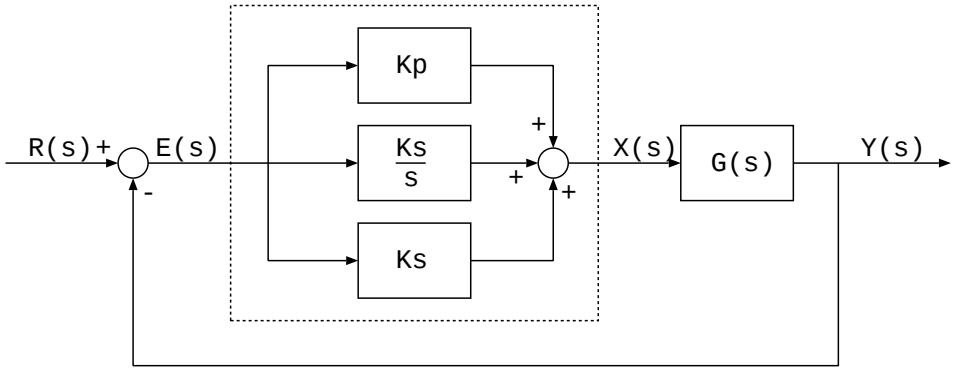


図 2.2: PID 制御

2.3 伝達関数によるモータモデル

一般に, システムの動作を伝達関数で表現し, それを時間領域表現に変換して状態方程式を得ることによってシステムのふるまいを調べることができる. これに基づいてシミュレーションを行うことによりモータの状態を推定することが可能である.

モータの種類は用途に応じて様々あるが, その 1 つにブラシ付き DC モータがある. これは, デジタル AV 機器, PC 周辺機器の駆動部分や電装用途によく用いられるものである.

ブラシ付き DC モータの伝達関数をブロック線図で表現したものを図 2.3 に示す. これに基づいてシミュレーションを行うことにより, 印加電圧 V_a と負荷トルク T_L から回転角速度 ω を求めることが可能である. 図中 s はラプラス演算子, R_a は巻線抵抗, L_a は巻線インダクタンス, T_e $(= L_a/R_a)$ は電気的時定数, K_T はモータ定数, D は粘性制動係数, J は慣性モーメント, K_E は逆起電力定数である.

このモータモデルの状態方程式を常微分方程式で記述し, 数値解析を行うことにより, 任意の時刻における回転速度 ω を求めることができる. 文献 [7] では, 同様のモデルを記述した C プログ

ラムを高位合成によって FPGA 上の回路に合成することにより, モータの高速なシミュレーションが可能であることを示している.

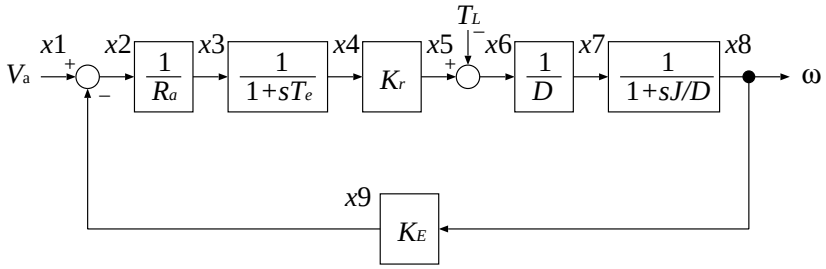


図 2.3: ブラシ付き DC モータの伝達関数のブロック線図 [6]

2.4 高位合成とバイナリ合成

高位合成 [3] は, C 言語等による動作記述からハードウェアの設計記述を合成する技術である. その中でも, アセンブリや機械語を入力として用いるものがあり, そのような合成技術はバイナリ合成 (binary synthesis) [8] とも呼ばれる. アセンブリや機械語を入力として用いることにより, ポインタや構造体, ライブラリ呼び出し等を含む広範なプログラムを合成対象とすることが可能である. バイナリ合成では, ソフトウェアの一部をハードウェア化することもできるが, 実行可能コード全体を合成することにより, そのプログラムおよびプロセッサと等価なハードウェアを合成することができる.

バイナリ合成システム ACAP [4] は, MIPS R3000 の機械語プログラムを入力として, これを 32bit の整数演算命令を並列に実行可能なハードウェアに変換する. ACAP の処理の流れを図 2.4 に示す. ACAP への入力は, 既存の機械語プログラムであってもよいし, アセンブリ言語や C 言語によるプログラムから得られるものであってもよい. 機械語プログラムは CDFG (control dataflow graph) に変換された後, 最適化, スケジューリング, バインディングの処理を施して, Verilog HDL に変換される.

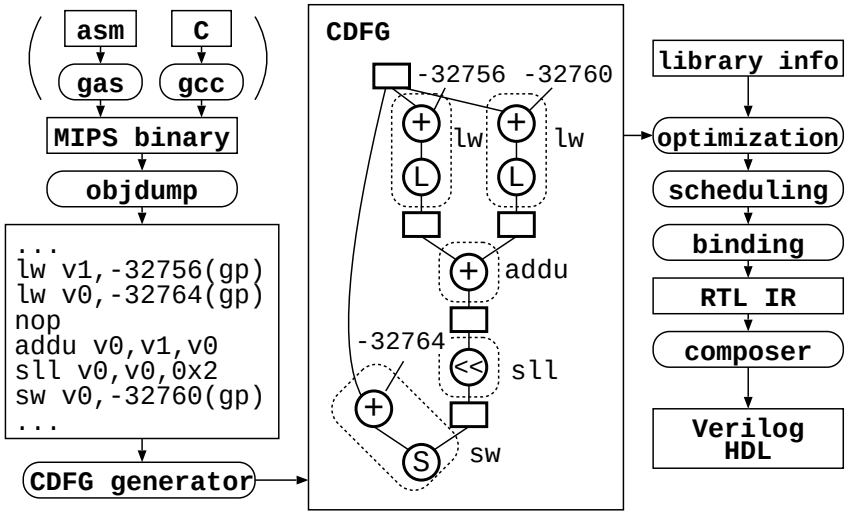


図 2.4: ACAP での高位合成の処理の流れ [4]

2.5 ACAP を用いた割込みハンドラを含むコードのハードウェア化

ACAP では, 割込みハンドラを含むプログラムのバイナリ合成が可能である [9]. 図 2.5 のように, プロセッサおよび割込みハンドラを含むプログラムに等価なハードウェアを, 通常処理 (main) と割込み処理 (int) を行う 2 つのモジュールとして合成する.

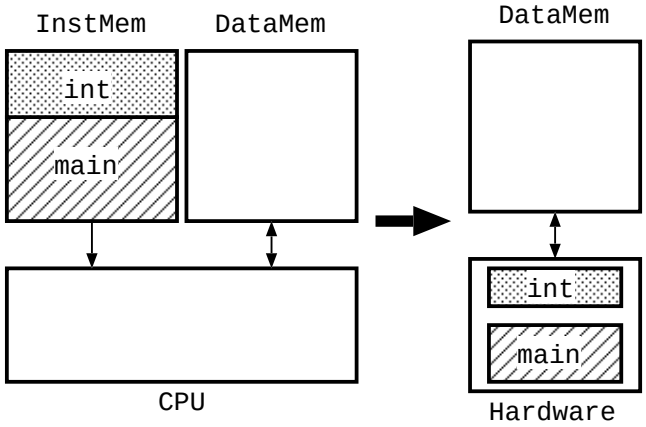


図 2.5: ACAP における割込みハンドラを含むコードのハードウェア化 [9]

生成されるハードウェアの詳細な構成を図 2.6 に示す. main_module と int_module はそれぞれ, 通常処理と割込み処理を行うモジュールである. 各モジュールが状態機械とデータパス, ALU 等の演算器, レジスタ等を持っている. そのため, 通常処理と割込み処理が独立に動作することができ, 割込みが発生しても通常処理を継続することができる.

CP0 は割込み信号の処理を行う演算器であり, MIPS のコプロセッサに相当する演算器, HW_sig, int_sig は割込み制御用のレジスタである. また, k0, k1 は, システムコールのためのレジスタの値を共有するためのレジスタファイルである.

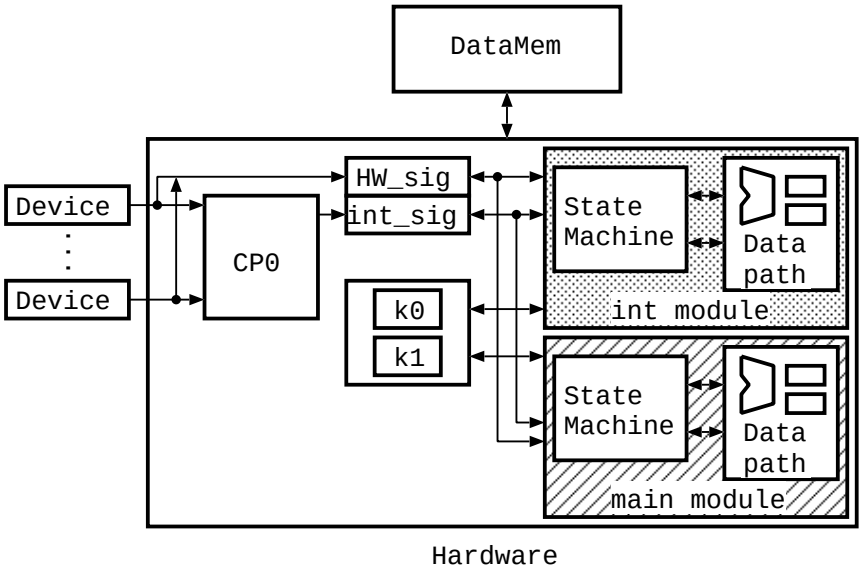


図 2.6: ACAP により合成されるハードウェアの構成 [9]

第3章　モータ制御のバイナリ合成

本研究では, C 言語で記述された割込み駆動のモータ制御プログラムをバイナリ合成によりハードウェア化する. このプログラムは図 2.1 の破線内部の “モータシミュレータ” と “コントローラ” からなるセンサレス制御システムに相当するものである. 合成されるハードウェアは, マイコン実装の場合と同様にタイマからの割込み信号に同期して動作する.

本研究における割込み駆動制御プログラムからのバイナリ合成は, 制御手法 (コントローラの種類) や制御対象 (モータの種類) に依存するものではないが, 以下の記述においては, 合成対象としてブラシ付き DC モータの PID 制御システムを想定する. また, 制御は周期的もしくは非同期の外部割込みに基づいて行う.

3.1　ブラシ付き DC モータの PID 制御システムの伝達関数モデル

PID 制御とブラシ付き DC モータのシミュレータを含めたブロック線図は, 図 3.1 のように定数と 3 個の 1 次遅れ要素で表現できる.

PID の各要素 (比例要素, 積分要素, 微分要素) はまとめて 1 つのブロック “PID” として表している. “PID”, “Gain”, “Limiter” の処理がコントローラに相当する. ここでは, 制御の目標値 $\omega*$ を受け取り, モータへの印加電圧 $x1$ を決定している. 次に, モータシミュレータは, モータに $x1$ の電圧を印加した場合のふるまいを推定する. シミュレータは, 電機子抵抗 Ra, 電機子インダクタンス La, モータ定数 K, 慣性モーメント J, 粘性制動係数 D, 負荷トルク T_L を考慮し, 電機子電流 $x4$, モータ発生トルク $x5$, 回転角速度 $x8$ を推定する.

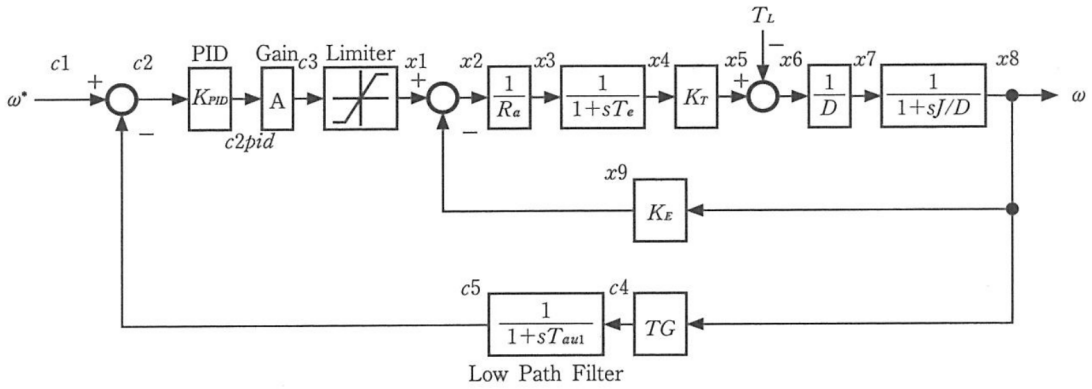


図 3.1: ブラシ付き DC モータの PID 制御の伝達関数のブロック線図 [6]

3.2 制御の記述

本研究のモータ制御を行うプログラムは図 3.2 [6] のように記述できる．関数 `PidControl` は, 制御量 (現在の回転速度) `pState->speed` と目標値 (目標速度) `pInput->wCmd` からモータへの印加電圧を計算し, これを制御入力として `pInput->wCmd` に設定する．33 行目の関数 `Pid` は, 比例ゲイン, 微分ゲイン, 積分ゲインに基づき, 比例要素, 積分要素, 微分要素の処理を行う．54 行目の関数 `DcLimit` は, 入力値が特定の範囲に含まれない場合にこれを範囲内の値に修正するものである．22 行目の関数 `Motor` はモータシミュレーションの呼び出しであり, 29 行目の関数 `Rk4` はルンゲ・クッタ法による一次遅れ要素の近似計算である．関数 `Motor` と関数 `Rk` については後述する．

3.3 モータの物理シミュレーションの記述

ブラシ付き DC モータの物理シミュレーションは, 図 3.3 [6] のように記述できる．関数 `Motor` は, 図 2.3 のブロック線図のモデルにおいて, 各モータパラメータを,

- 巻線抵抗 R_a=1Ω
- 巻線インダクタンス L_a=8mH
- 逆起電力定数 K_E=0.28Vs/rad
- トルク定数 K_r=0.28Nm/A
- 慣性モーメント J=0.005kgm^2
- 粘性制動係数 D=0.0002Nms/rad

と設定したものである．22 行目や 26 行目で呼び出しいてる関数 `Rk` は 4 次のルンゲ・クッタ法 (RK4) により一次遅れ要素の近似解を求めるものである．RK4 は計算量と誤差のバランスが良く, 数値計算においてよく用いられている常微分方程式の数値解法である．その処理の C 言語記述を図 3.4 [6] に示す．

```
void PidControl(
    MotorInput *const pInput,
    MotorState *const pState,
    const float dt
)
{
    static float c5 = 0.f;
    static float work[2] = {};

    const float KP = 1.0f,
                KI = 0.0f,
                KD = 0.03f;

    const float GAIN  = 500.f;
    const float c1    = pInput->wCmd;
    const float c2    = c1 - c5;
    const float c2pid = Pid(c2, KP, KI, KD, work, dt);
    const float c3    = c2pid * GAIN;

    pInput->wCmd = DcLimit(c3);

    Motor(pInput, pState, dt);


    const float TG   = 10.0f / 314.159f;
      // 速度検査器設定 (3000/min: 10V)
    const float c4   = pState->speed * TG;
    const float TAU1 = 0.01f;
    c5 = Rk4(c4, c5, TAU1, dt);
}


float Pid(
    const float xin,
    const float kp,
    const float ki,
    const float kd,
          float work[],
    const float dt
)
{
    const float a = xin;
    const float b = (xin + work[0])/2.0f * dt + work[1];
    const float c = (xin - work[0])/dt;

    work[0] = xin;
    work[1] = b;

    const float xout = a * kp + b * ki + c * kd;
    return xout;
}


float DcLimit(const float input)
{
    const float MAX_V  = 80.f;
    const float MIN_V  = -80.f;

    return
          (MAX_V < input) ? MAX_V
        : (input < MIN_V) ? MIN_V
        :                   input;
}
```

図 3.2: モータの PID 制御 [6]

```
void Motor(
    MotorInput *const pInput,
    MotorState *const pState,
    const float dt
)
{
    static float x4  = 0.0f;
    static float x8  = 0.0f;
    static float x9  = 0.0f;

    const float Ra   = 1.0f;              //  電機子抵抗
    const float La   = 0.008f;            //  電機子インダクタンス
    const float TAUe = La / Ra;           //  時定数
    const float K    = 0.28f;             //  モータ定数 (KE = kT = K)
    const float J    = 0.005f;            //  負荷を含む慣性モーメント
    const float D    = 0.0002f;           //  粘性制動係数
    const float TAUj = J / D;

    const float x1   = pInput->wCmd;
    const float x2   = x1 - x9;
    const float x3   = x2 / Ra;
                x4   = Rk4(x3,x4,TAUe,dt);       //  電機子電流
    const float x5   = x4 * K;                   //  モータ発生トルク
    const float x6   = x5 - pInput->loadTorque; //  負荷トルクを印加
    const float x7   = x6 / D;
                x8   = Rk4(x7,x8,TAUj,dt);       //  回転角速度
                x9   = x8 * K;

    pState->speed    = x8;
    pState->current  = x4;
}
```

図 3.3: モータのシミュレーション [6]

```
// 4 次のルンゲ・クッタ法による近似解計算
float Rk4(
  const float xin,
  const float xold,
  const float tau,
  const float dt
)
{
    const float k1 = dt * (xin - xold) / tau;
    const float k2 = dt * (xin - (xold + k1 / 2.f)) / tau;
    const float k3 = dt * (xin - (xold + k2 / 2.f)) / tau;
    const float k4 = dt * (xin - (xold + k3)) / tau;

    const float xout = xold + (k1 + 2.f * k2 + 2.f * k3 + k4) / 6.f;
    return xout;
}
```

図 3.4: 4 次のルンゲ・クッタ法による近似解計算 [6]

3.4 タイマ割込み処理の記述

本研究では, コントローラがタイマからの割込みによって駆動されることを想定している. 割込みハンドラが割込みを受け付け, コントローラおよびシミュレータを起動する. 割込みハンドラの記述を図 3.5 に示す. TimerHandler はタイマ割込みが発生した時に呼び出される関数である. 6 行目の関数 GetTime で現在時刻を取得し, 前回の制御時刻からの経過時間 dt を算出する. また, 13 行目の GetCurrentLoad は現在の負荷トルクを取得する関数である. 16 行目で, これらの値を引き渡して, 実際の制御演算を行う関数 PidControl を呼び出している.

```
1  void TimerHandler()
2  {
3      // 前回の制御時刻
4      static float time_l;
5      // 現在の時刻
6      const float time = GetTime();
7
8      const float dt = time - time_l;
9      time_l = time;
10
11     MotorInput input = {
12         5.0f,                // 速度指令
13         GetCurrentLoad()     // 現在の負荷
14     };
15
16     PidControl(&input, &state, dt);
17
18     loopCount++;
19 }
```

図 3.5: 割込みハンドラ

メイン関数では, 割込みハンドラの登録等の初期化処理を行った後, 空の無限ループで終了処理まで待機する. その記述を図 3.6 に示す. 5 行目の関数 init_interrupt で割込み処理の初期化を, 6 行目の関数 register_exc_handler で割込みハンドラの登録を行っている. なお, init_interrupt, register_exc_handler, および割込み受け付けルーチンは, 文献 [9] と同様, インラインアセンブリを含む C プログラムおよび MIPS アセンブリプログラムで実装している. また, 変数 fContinue は制御の継続を示すフラグであり, ハードウェア化して実行した際に, 外部から書き換えて実行を停止するためのものである.

```
volatile int fContinue = 1;

int main()
{
    init_interrupt();
    register_exc_handler(
        EXC_Int0, TimerHandler
    );

    InitializeSim(); // 出力関連の初期化

    while ( fContinue ) /* VOID */;

    FinalizeSim();
    return 0;
}
```

図 3.6: 割込みハンドラの登録

3.5 制御記述からの高位合成

本研究でのハードウェアの合成手法を図 3.7 に示す．まず, C 言語で割込みハンドラと, そこから呼び出されるコントローラ及びシミュレータを記述する．これを MIPS アセンブリにコンパイルし, soft float 実行時ライブラリと割込み受け付け処理のライブラリをリンクして MIPS の実行可能バイナリを生成する．soft float は浮動小数点演算ユニットが無いプロセッサで浮動小数点演算命令を実行するためのライブラリであり, 浮動小数点演算を整数演算によるエミュレーションによって実行する．

これを逆アセンブルしたものに, モジュールを指定するプラグマを追加する．制御ハードウェアは図 3.8 のように, 通常処理モジュールと割込み処理モジュールからなっており, どの処理をどちらのモジュールで行うかを指定する必要があるためである．今回は, シミュレーションを含む制御処理を割込みモジュールで行い, 空の `main` 関数および初期化処理, 終了処理のみを通常処理として実行する．

このアセンブリを ACAP でバイナリ合成することにより, Verilog HDL で記述された制御ハードウェアが得られる．なお, プログラム中の soft float による浮動小数点演算は, ACAP 内で, 浮動小数点演算器を用いた演算に置き換えられる．

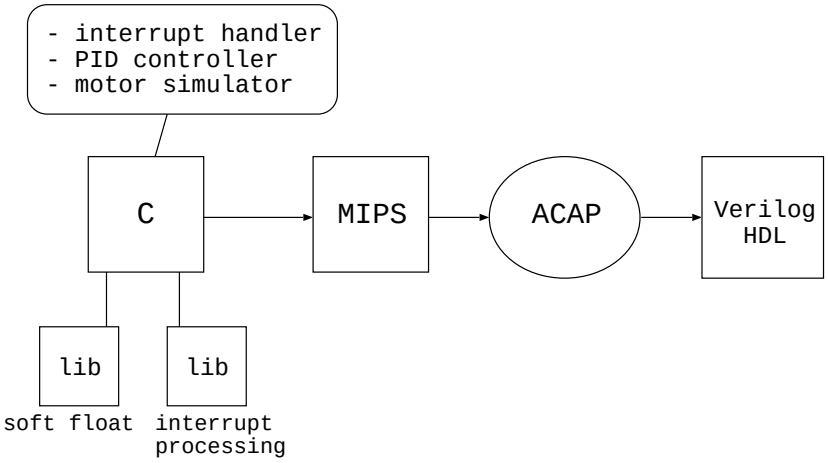


図 3.7: 割込み駆動の制御記述からの高位合成

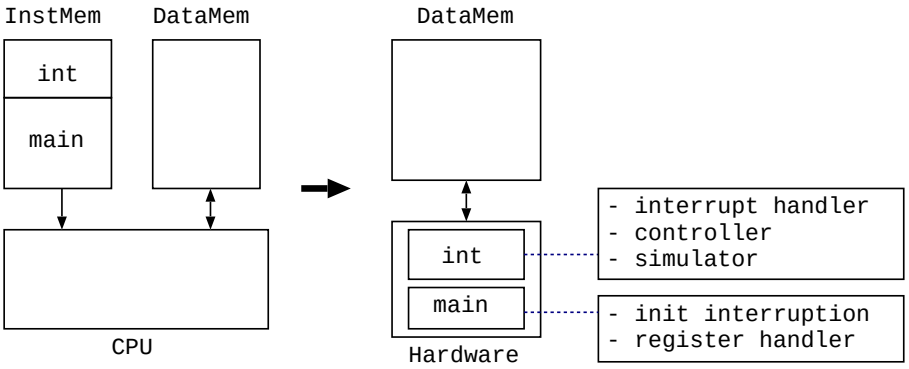


図 3.8: 合成されるハードウェアの構成

第4章　実験と考察

本研究で提案する手法に基づき, 割込み駆動のモータ制御プログラムを, バイナリ合成システム ACAP を用いて合成した. C プログラムからの MIPS アセンブリへのコンパイルには O0 オプションを用いた. 浮動小数点演算器に関しては, 加減算器と乗算器は [7] で設計されたものを, 除算器と比較演算器は Xilinx Core Generator により生成した IP コアを使用した.

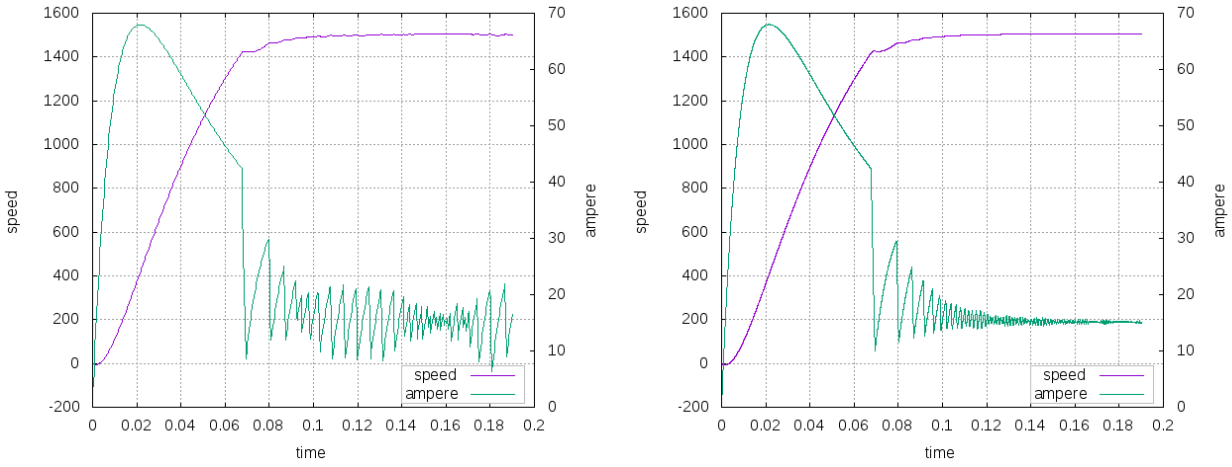
4.1　動作確認

作成した制御システムの動作確認を, x86 上, MIPS R3000 互換ソフトプロセッサ [10] 上, および本研究で生成したハードウェア上で行った.

まず, x86 ネイティブ環境, x86 上の MIPS クロス開発環境 (MIPS 用クロスコンパイラと MIPS の命令レベルシミュレーション を使用) で正しく制御ができることを確認した. 次に, MIPS ソフトコアプロセッサ上で MIPS バイナリが正しく実行できることを RTL シミュレーションで確認した. 図 2.1 のタイマに相当する割込み発生処理を Verilog HDL で記述し, 制御処理が周期的に起動されるようにした. その結果, x86 上で実行した場合と同様の処理が実装できていることが確認できた.

これらのプログラムの動作確認を行った後, 本研究の手法で合成した専用ハードウェアの動作確認の RTL シミュレーションを行った. 割込みを周期的に発生させ, 制御システムとして正常に動作することを確認した.

シミュレーション結果を, 図 4.1 に示す. (a) は MIPS ソフトコアプロセッサ, (b) は 本研究のハードウェアでの実験結果である. これは, 外乱として, モータに一定の負荷トルクをかけ続けた場合の立ち上がりから収束までの 0.2 秒間の回転速度と電流値をプロットしたものである. 回転速度が目標値に正しく収束しており, PID 制御が正常に行えていることが確認できる.



(a) MIPS (b) HW

図 4.1: シミュレーション結果

4.2 評価

バリナリ合成によって得られた Verilog HDL を, 論理合成ツール Xilinx ISE 14.7 によって, Xilinx Spartan 6 をターゲットに論理合成した. その結果を表 4.1 に示す. #FFs, #LUTs, Delay, Cycles は, それぞれ, フリップフロップ数, LUT 数, 遅延時間および 制御 1 周期に要するサイクル数を表す. 表中 “MIPS” は MIPS R3000 互換プロセッサ [10], “HW” は本研究で合成したハードウェアである. なお, プロセッサは浮動小数点演算器を持っておらず, 浮動小数点演算は soft float 実行時ライブラリにより行っている. また, “Period” は, 図 4.1 のシミュレーションに要した実行サイクル数と遅延時間から推定される制御周期である.

MIPS ソフトコアプロセッサで実装した場合の制御割込みの間隔は 23001 サイクルであり, 1 サイクル遅延が 16.078 ns であることから, 1 周期の制御に必要な処理が 369.8 µs で行えると推定される. 同様にして得られる HW の場合の制御周期は 20.6 µs となる. つまり, 制御システム全体をハードウェア化することにより, 約 18 倍高速に動作させることができている. 一方で, 回路規模は MIPS ソフトプロセッサに比べて, FF 数 1.52 倍, LUT 数 4.39 倍に増大している.

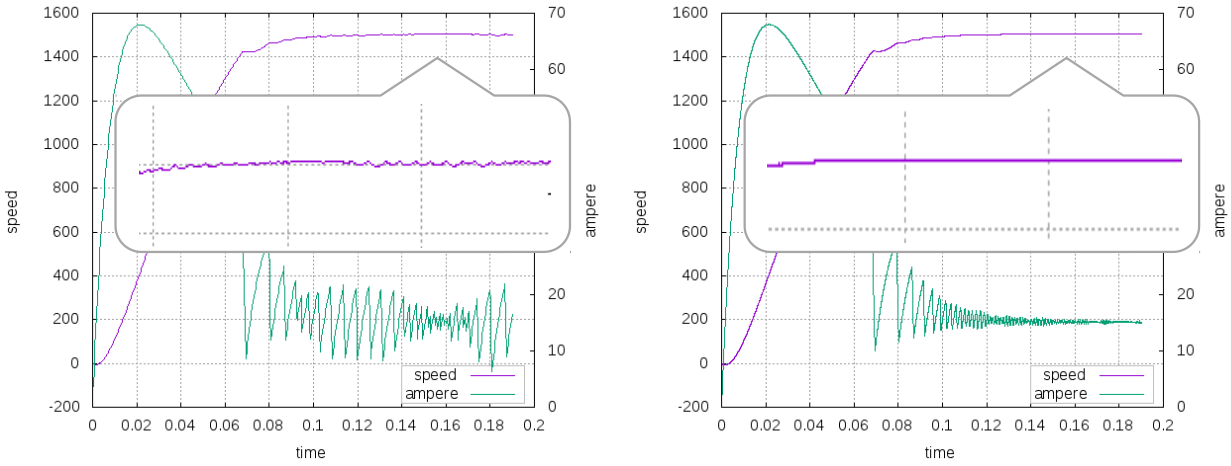
表 4.1: 論理合成およびシミュレーションの結果

	#FFs	#LUTs	Delay [ns]	Cycles	Period [µs]
MIPS [10]	1821 (1.00)	3788 (1.00)	16.078 (1.00)	23001 (1.00)	369.8 (1.00)
HW	2766 (1.52)	16617 (4.39)	23.676 (1.47)	871 (0.04)	20.6 (0.06)

4.3 考察

一般的に, 制御においては, 制御手法が同じであっても処理周期が短くなればなるほど制御の精度は向上し, 安定性が向上するとされている. しかし, 制御およびシミュレーションの演算は, 次の割込み発生までに終える必要がある. そのため, 短い時間で演算を完了することが制御の安定に直結する.

今回のシミュレーションにおいても, 制御周期の短縮により制御が安定することが確認できた. 先程のシミュレーションでの定常状態におけるモータの回転速度の推移を拡大したものを, 図 4.2 に示す. ソフトコアプロセッサで実行した場合の (a) に比べ, 本手法によりハードウェア化した場合の (b) の方が安定度が高いといえる.



(a) MIPS　　(b) HW

図 4.2: モータの回転数の推移

また, モータの制御においては, 回転速度や電圧, 電流値の安定は余分な電力消費の抑制につながる. つまり, 図 4.2 の制御の安定度向上は, 省電性能の向上を示しているといえる.

“MIPS” と “HW” において, 各時刻における印加電圧値の 2 乗の総和を求めたものを, 表 4.2 に示す. これは, モータの過渡状態, 定常状態における 1 秒間の消費電力の評価値を表している. 本手法で合成したハードウェアは, “MIPS” に比べ, 過渡状態において約 45.6 %, 定常状態において約 66.9 %, 評価値が抑えられている. モータの消費電力値はこの評価値に比例するため, 実際の消費電力においても同様の削減が見込まれる.

表 4.2: 消費電力評価値の比較

	過渡状態	定常状態
MIPS [10]	5995.3 (1.00)	5999.4 (1.00)
HW	3261.9 (0.54)	1985.0 (0.33)

更に, 本研究における合成手法 (3.5 節) は, 制御対象やその制御手法に依存しない. つまり, より高度な制御方法を利用する場合や, 他のモータモデルを用いる場合でも, 同様に合成を行える汎用性を持っているといえる.

第5章　結論

本研究では, モータの伝達関数モデルを用いた割込み駆動のセンサレス制御プログラムを, バイナリ合成システム ACAP を用いてハードウェア化する手法を提案した.

割込みハンドラ, コントローラ, シミュレータを C 言語で記述し, コンパイルして得られる MIPS バイナリを入力として ACAP でハードウェア化することにより, 割込み駆動制御プログラムから, それと等価なハードウェアを自動生成することができる.

実際に, ブラシ付き DC モータのセンサレス PID 制御プログラムを合成し, RTL シミュレーションすることにより, 制御システムが合成できることを確認した. 実験では, 約 21 µs 間隔で割込みをかけて PID 制御のシミュレーションを行うことができた. これは, MIPS ソフトコアプロセッサ比で約 18 倍の高速化である. また, 消費電力量において 66.9 % の削減が見込まれる.

また, 本手法を用いることにより, より高度な制御方法や, 他のモータモデルを利用する場合においても, 同様の方法で, 割込み駆動の制御プログラムからの高位合成が可能である.

今後の課題として, 制御周期の更なる短縮と割込みを活用したさらに高度な制御への適用や, モータ以外の制御への応用等が挙げられる.

謝辞

本研究に際し, 多くの方々から御指導, 御支援を賜りました. ここに感謝の意を表します.

高位合成の研究に携わる機会を与えていただき, 研究に関する多くの御指導, 御支援を賜りました石浦菜岐佐教授に心より感謝いたします.

本研究に関する有益な議論の場を提供して頂き, 様々な視点から御指導いただきました京都高度技術研究所の神原弘之氏に感謝いたします. 本分野に対する高い見識を持ち, 様々な面で御助力いただきました立命館大学の冨山宏之教授に感謝いたします. 技術的な面で多くの御助言をいただきました元立命館大学の中谷嵩之氏に感謝いたします.

本研究に関してご協力, ご討議下さいました元関西学院大学の田村真平氏 (現 株式会社エムエスティ), 伊藤直也氏 (現 株式会社村田製作所) をはじめ, HLS チームの諸氏に深く感謝いたします. 研究のみならず生活面においても多くの御支援をいただいた関西学院大学理工学部石浦研究室の皆様に感謝いたします.

参考文献

[1] “電力機器の消費電力量に関する現状と近未来の動向調査,” `http://www.sicalliance.jp/science_data/bunken/fed-power-consume.pdf`, 財団法人新機能素子研究開発協会 (Mar. 2009).

[2] 江崎雅康: “なぜ FPGA でモータを制御するのか,” FPGA マガジン no.7, pp. 4–7, CQ 出版社 (Oct. 2014).

[3] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[4] Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama: “ACAP: Binary Synthesizer Based on MIPS Object Codes,” in *Proc. International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2014)*, pp. 725–728 (July 2014).

[5] 大迫 裕樹, 神原 弘之, 石浦 菜岐佐: “割り込み駆動のモータ制御プログラムからの高位合成,” 情報処理学会関西支部大会, A-01 (Sept. 2016).

[6] 高橋久: C 言語によるモーター制御入門講座 〜SH マイコンで学ぶプログラミングと制御技法〜, 電波新聞社 (Oct. 2007).

[7] 竹林陽, 伊藤直也, 田村真平, 神原弘之, 石浦菜岐佐: “高位合成系 ACAP を用いたモーターの浮動小数点モデルの FPGA 上での実行,” 情報処理学会関西支部大会, A-02 (Sept. 2014).

[8] Greg Stitt and Frank Vahid: “Binary synthesis,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 3, article 34 (Aug. 2007).

[9] 伊藤直也, 石浦菜岐佐, 冨山宏之, 神原弘之: “割込みハンドラを独立したモジュールとして実装するバイナリ合成,” 電子情報通信学会技術研究報告, VLD2015-106 (Jan. 2016).

[10] 神原弘之, 金城良太, 戸田勇希, 矢野正治, 小柳滋: “パイプラインプロセッサを理解するための教材: RUE-CHIP1 プロセッサ,” 情報処理学会関西支部大会, A-09 (Sept. 2009).