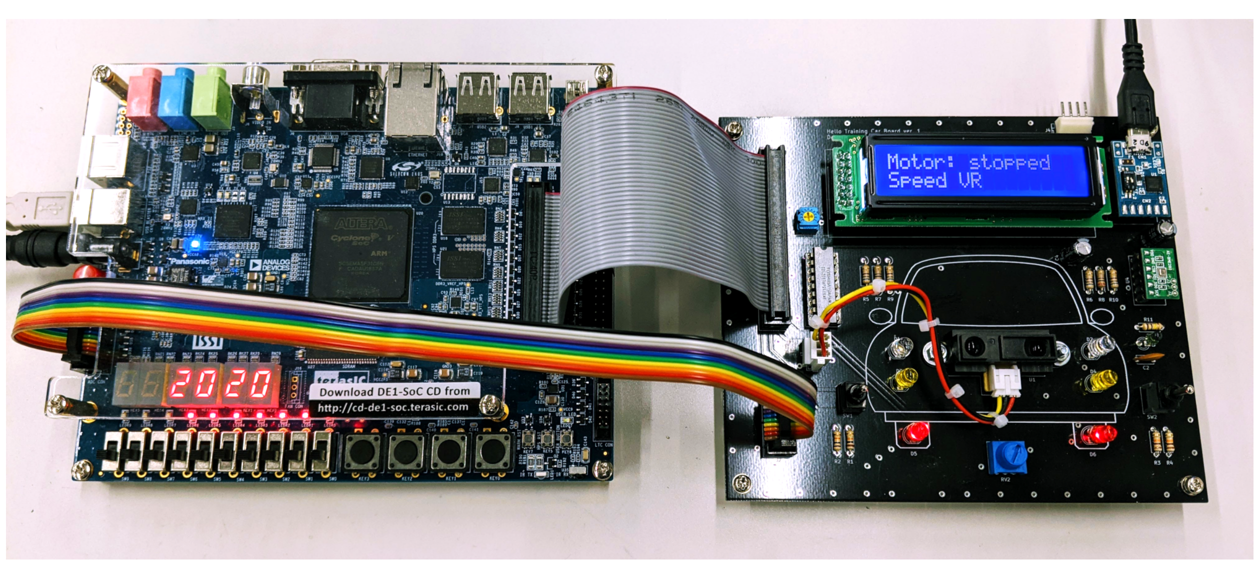
ソフトウェアコアを活用した制御システム構築



目次

第 1 章　ソフトコアプロセッサを活用した制御システム構築の概要　1
1.1　ソフトコアプロセッサとは　1
1.2　ソフトコアプロセッサ Nios II　2
1.3　Nios II を利用した制御システムの開発の流れ　2
1.3.1　Nios II および周辺回路の構成設定　3
1.3.2　Nios II EDS を用いた組込みソフトウェアの開発　4
1.4　実習の流れ　4

第 2 章　PIO（Parallel I/O）による入出力制御　7
2.1　FPGA ボード DE1-SoC の概要　7
2.2　ハードウェアの構築　8
2.2.1　構築するハードウェアの概要　8
2.2.2　プロジェクトの作成　9
2.2.3　Nios II および周辺回路の構成　12
2.2.4　最上位階層の作成と論理合成　26
2.2.5　ピンアサインとコンパイル　28
2.2.6　FPGA へのプログラミング　31
2.3　組込みソフトウェアの開発　35
2.3.1　Nios II SBT の起動　35
2.3.2　プロジェクトの作成　36
2.3.3　プロジェクトのビルドとプログラムの実行　38
2.3.4　`alt` 標準入出力関数によるデバッグ　41
2.3.5　Nios II Debug パースペクティブによるデバッグ　45
2.4　LED の制御　50
2.4.1　DE1-SoC の LED 回路　50
2.4.2　LED 制御プログラムの作成　51
2.5　スライドスイッチからの入力　56
2.5.1　DE1-SoC のスライドスイッチ回路　56
2.5.2　スライドスイッチ入力を使用するプログラムの作成　57
2.6　ボタンスイッチからの入力　62
2.6.1　DE1-SoC のボタンスイッチ回路　62
2.6.2　ボタンスイッチ入力を使用するプログラムの作成　63
2.7　7 セグメント LED の制御　68

2.7.1 DE1-SoC の 7 セグメント LED 回路 . . . 68
2.7.2 7 セグメント LED 制御プログラムの作成 . . . 69
2.8 まとめ . . . 76

第 3 章 実践的なハードウェア設計 77
3.1 ソフトウェアで記述した処理のハードウェア化 . . . 77
3.1.1 トレードオフを考慮したシステム設計 . . . 77
3.1.2 7 セグメント LED への数値表示処理のハードウェア仕様 . . . 78
3.1.3 7 セグメント LED への数値表示処理の実装 . . . 79
3.1.4 ゼロ埋め制御機能の実装 . . . 97
3.1.5 Nios II からの利用 . . . 108
3.2 ロジックアナライザを活用したチャタリング除去回路の設計 . . . 111
3.2.1 ロジックアナライザ Signal Tap の概要 . . . 111
3.2.2 実習の概要 . . . 111
3.2.3 波形観察用プロジェクトの作成 . . . 113
3.2.4 Signal Tap を用いたチャタリング現象の観察 . . . 114
3.2.5 チャタリング除去回路の設計 . . . 121
3.2.6 チャタリング除去回路の動作確認 . . . 129
3.2.7 制御への応用：ウィンカー回路の製作 . . . 131
3.2.8 Nios II からの利用 . . . 138
3.3 まとめ . . . 140

第 4 章 IP コアを活用した制御システム構築 141
4.1 製作する制御システムの仕様 . . . 141
4.1.1 モータ制御 . . . 144
4.1.2 ヘッドライト . . . 144
4.1.3 ウィンカー . . . 144
4.1.4 メータ . . . 145
4.1.5 文字表示器 . . . 146
4.2 ハードウェアの構築 . . . 146
4.2.1 プロジェクトの作成 . . . 147
4.2.2 Nios II および周辺回路の構成 . . . 147
4.2.3 最上位階層の作成と論理合成 . . . 157
4.2.4 ピンアサイン，コンパイル，プログラミング . . . 162
4.2.5 ウィンカーの動作確認 . . . 166
4.3 組込みソフトウェアの開発 . . . 166
4.3.1 コーディング規約 . . . 166
4.3.2 プロジェクトの作成 . . . 168
4.3.3 BSP の設定 . . . 169
4.3.4 プログラムの枠組み . . . 170
4.3.5 ヘッドライトの制御 . . . 176
4.3.6 キャラクタ LCD の制御 . . . 181

4.3.7 センサと A/D 変換 191
4.3.8 シリアル通信によるモータ制御 204
4.3.9 各種センサの制御への応用 212
4.4 応用課題 216
4.5 まとめ 220

第 1 章

ソフトコアプロセッサを活用した制御システム構築の概要

FPGA の大規模化に伴い，ソフトコアプロセッサ（FPGA のロジック要素を用いて実装されたプロセッサ）を使用して制御システムを構築する事例が増えてきました．そこで本実習では，ソフトコアプロセッサを活用した制御システムの構築方法の習得を目的として，FPGA 上にソフトコアプロセッサ Nios II と周辺回路を構築する方法，および FPGA のハードウェアデザインと Nios II とを連携させる組込みアプリケーションの開発方法について学習します．

この章では，ソフトコアプロセッサを活用した制御システム構築の概要について述べるとともに，実習の流れについて示します．

1.1 ソフトコアプロセッサとは

ソフトコアプロセッサとは，FPGA のロジック要素を用いて実装されたプロセッサです．ソフトコアプロセッサには，以下に示す利点があります [1–4].

- アプリケーションに合わせて，必要十分なペリフェラル（周辺回路），メモリ，I/O を無駄なく組み込める．
- プロセッサも含めて FPGA チップに集約することで，部品点数を減らすことができる．これは，基板の小型化（図 1.1），低消費電力化，低コスト化につながる．
- 製品の寿命を延長できる．
  - ハードプロセッサを用いる場合と比較して，部品の生産中止や入手困難のリスクが少ない．
  - ハードウェアとソフトウェアの両方を更新して，システムの性能を向上させることができる．

ソフトコアプロセッサの欠点としては，ハードプロセッサと比較して動作速度が低いことがあります [3]．しかし，ハードウェア構成の工夫や FPGA の性能向上によって，多くのアプリケーションにとっては十分な性能が得られます．

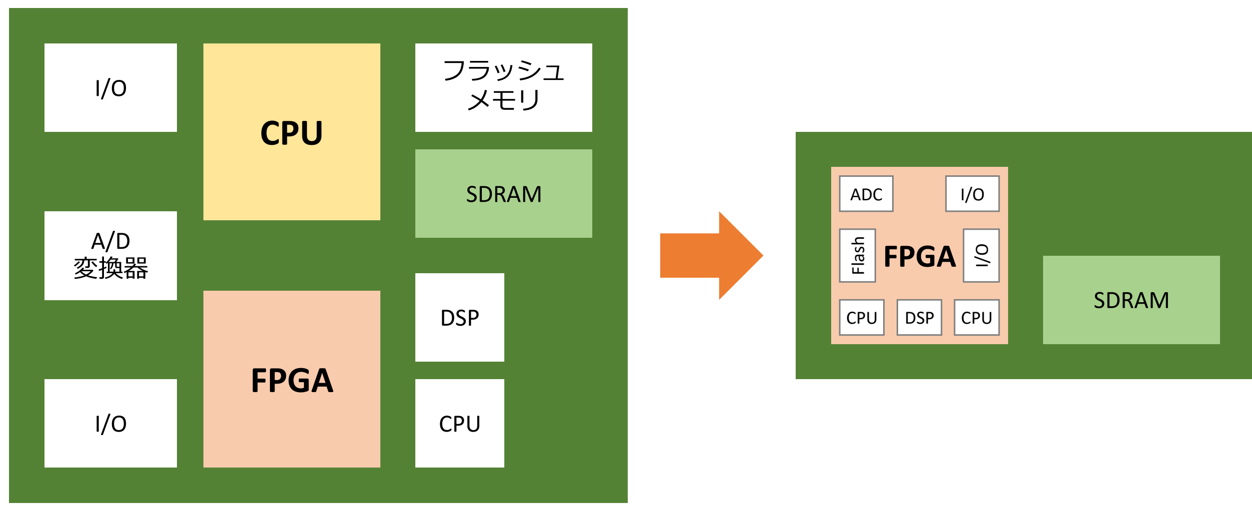


図 1.1　ソフトコアプロセッサの利用による基板の小型化．文献 [4] をもとに作成．

1.2　ソフトコアプロセッサ Nios II

Nios II は，インテル社（旧アルテラ社）製 FPGA に実装可能な 32 ビット RISC*1プロセッサコアです．ソフトコアであるため，キャッシュサイズ，演算器の実装，アドレスマップなど，通常のマイコンでは固定されている要素も設定可能となっています [2].

現在は，表 1.1 に示す 2 種類の Nios II コアが用意されています*2．開発者は，アプリケーションの要件に合わせて適切なコアを選択し，使用することができます．この実習では，無償で使用可能な Nios II/e を用いて制御システムを構築していきます．

表 1.1　Nios II のコアの種類

名称	特徴
Nios II/e	低消費電力，低コストであるエコノミー（economy）コア．無償で使用可能.
Nios II/f	高性能な高速（fast）コア．Nios II/e に対してハードウェア乗除算機，キャッシュメモリ，分岐予測などの機能が追加されている．有償.

1.3　Nios II を利用した制御システムの開発の流れ

Nios II を利用した制御システムの開発では，これまでの FPGA の実習で使用してきた Quartus Prime や ModelSim に加えて，Platform Designer，Nios II EDS という 2 つのツールを新たに使用します．図 1.2 に開発の流れを示します．

*1 Reduced Instruction Set Computer．少数の単純な固定長命令のみを備え，実行効率を向上させているプロセッサ．
*2 過去には，標準的な性能の Nios II/s（standard）というコアが存在しました．

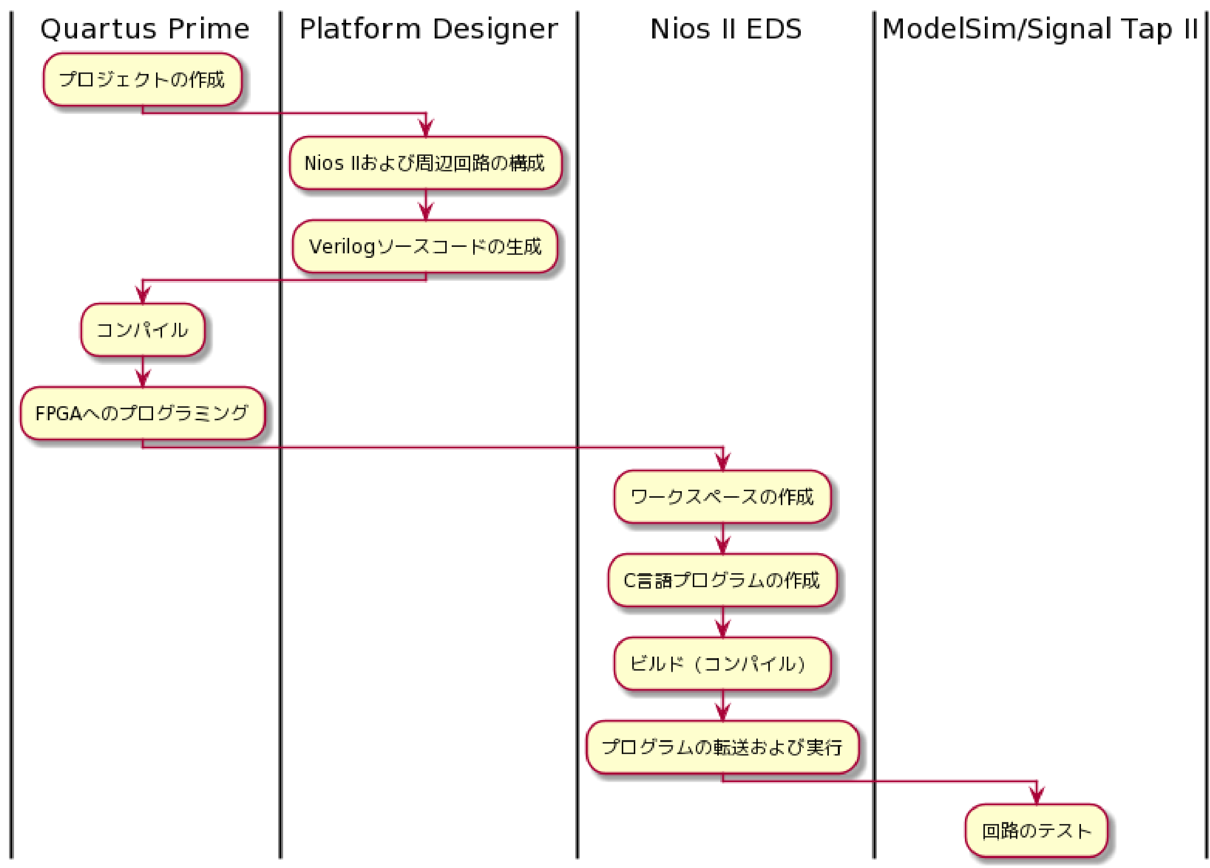


図 1.2　Nios II を利用した制御システムの開発の流れ

1.3.1　Nios II および周辺回路の構成設定

Nios II を動作させるには，プロセッサコアだけでなく，クロックやメモリ，周辺回路も含めて構成して，FPGA 上に実装する必要があります（図 1.3）．この構成設定には，Quartus Prime に同梱されている **Platform Designer** というソフトウェアを使用します．Platform Designer には，Nios II とともに様々な周辺回路の IP コア*3が用意されており，その中から必要なものを選択，設定してシステムに組み込むことができます．表 1.2 に，システムに組み込める周辺回路の例を示します．

Nios II および周辺回路の構成設定後は，Platform Designer の HDL 生成機能を使用して，構成したシステムの Verilog ソースコードを出力します．出力されたコードとユーザ回路のコードとを Quartus Prime を使用してコンパイルすることで，Nios II と周辺回路を含むシステム全体の回路が完成します．

*3 IP core：Intellectual Property core．Intellectual Property は知的財産という意味ですが，IP コアとは主に機能別に用意された再利用可能な回路部品の設計情報を指します [5]．

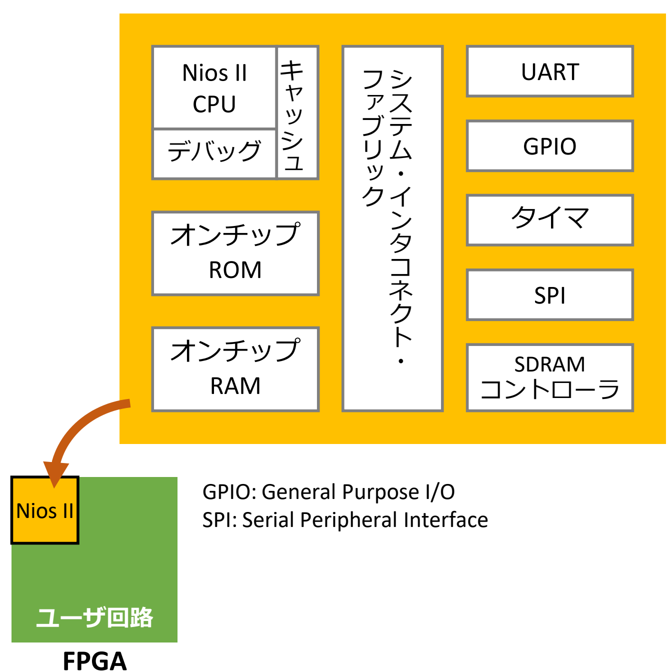


図 1.3　Nios II および周辺回路の構成例．文献 [2] をもとに作成．

表 1.2　システムに組み込める周辺回路の例

周辺回路	主な設定可能項目
PIO（パラレル IO）	ビット幅，入力/出力/双方向，割り込み
Timer	カウンタのビット幅，タイムアウトの周期
UART（RS-232C シリアルポート）	ボーレート，パリティ，データビット数，ストップビット数
A/D 変換器コントローラ	A/D 変換クロック周波数，チャンネル数

1.3.2　Nios II EDS を用いた組込みソフトウェアの開発

Nios II 上で動作する組込みソフトウェアの開発には，**Nios II EDS**[*4]という開発パッケージを使用します．このパッケージに含まれる Nios II SBT[*5]という Eclipse[*6]ベースの統合開発環境を用いて，C 言語プログラムの編集，コンパイル，デバッグを行います．Nios II EDS には，構成した Nios II および周辺回路に合わせてデバイスドライバを自動的に生成する機能もあり，これを利用すると保守性の高いプログラムを効率良く書くことができます．

1.4　実習の流れ

この節では，本実習の流れについて説明します．

第 2 章では，まず Nios II を含む制御システムのハードウェアを構築する方法について学習します．続いて，そのハードウェアに組み込んだ PIO（Parallel IO）という IP コアを使用して，基本的な入出力制

[*4] EDS：Embedded Design Suite.
[*5] SBT：Software Build Tools.
[*6] Eclipse はオープンソースの統合開発環境です．Java アプリケーションの開発等に広く用いられています．

御を行う組込みプログラムの開発を行います．制御対象の装置として，LED，スイッチ，7 セグメント LED を扱います．

第 3 章では，FPGA の利点を生かした実践的なハードウェア設計を 2 種類行います．最初に取り組むテーマは，ソフトウェアで実装した処理のハードウェア化です．第 2 章においてソフトウェア（C 言語）で実装する 7 セグメント LED の制御を例として，処理の速さや開発のしやすさといった様々な要素を考慮しながら，仕様策定および処理のハードウェア化に取り組みます．その際，回路の動作検証のため，テストベンチの記述および論理シミュレーションも行います．次の取り組むテーマは，FPGA に搭載できるロジックアナライザ機能の活用です．この機能を活用して実際の波形を観察しながら，適切なタイミングで動作するトグルスイッチのチャタリング除去回路を設計・実装します．さらに，実装したモジュールを応用して自動車のウィンカーを模した LED 点滅回路を設計し，階層設計手法を習得します．

第 4 章では，Platform Designer に含まれる IP コアを活用して様々な周辺装置を制御する方法について学習します．この章では，自動車の電装品を模した「簡易車載システム」の開発を通して，A/D 変換，シリアル通信，キャラクタ LCD の制御といった，より高度な機能を制御する方法を習得します．この課題では，FPGA 基板および追加の実習基板に実装された様々な周辺装置を同時に制御する必要があるため，それを行うための組込みプログラミング手法についても学習します．

第 2 章

PIO（Parallel I/O）による入出力制御

この章では，ソフトコアプロセッサ Nios II を使用する制御システムのハードウェアを構築する方法，および IP コア「PIO（Parallel I/O）」を使用して，Nios II の基本的な入出力制御を実装する方法について学習します．制御対象の装置として，LED，スイッチ，7 セグメント LED を扱います．

2.1　FPGA ボード DE1-SoC の概要

この節では，本実習で使用する FPGA ボード DE1-SoC について解説します．

DE1-SoC は，Terasic 社製の FPGA 開発ボードです．図 2.1 に DE1-SoC の写真および各部品の機能を示します．

DE1-SoC には，FPGA デバイス Cyclone V 5CSEMA5F31C6 と，豊富な入出力装置が実装されていま

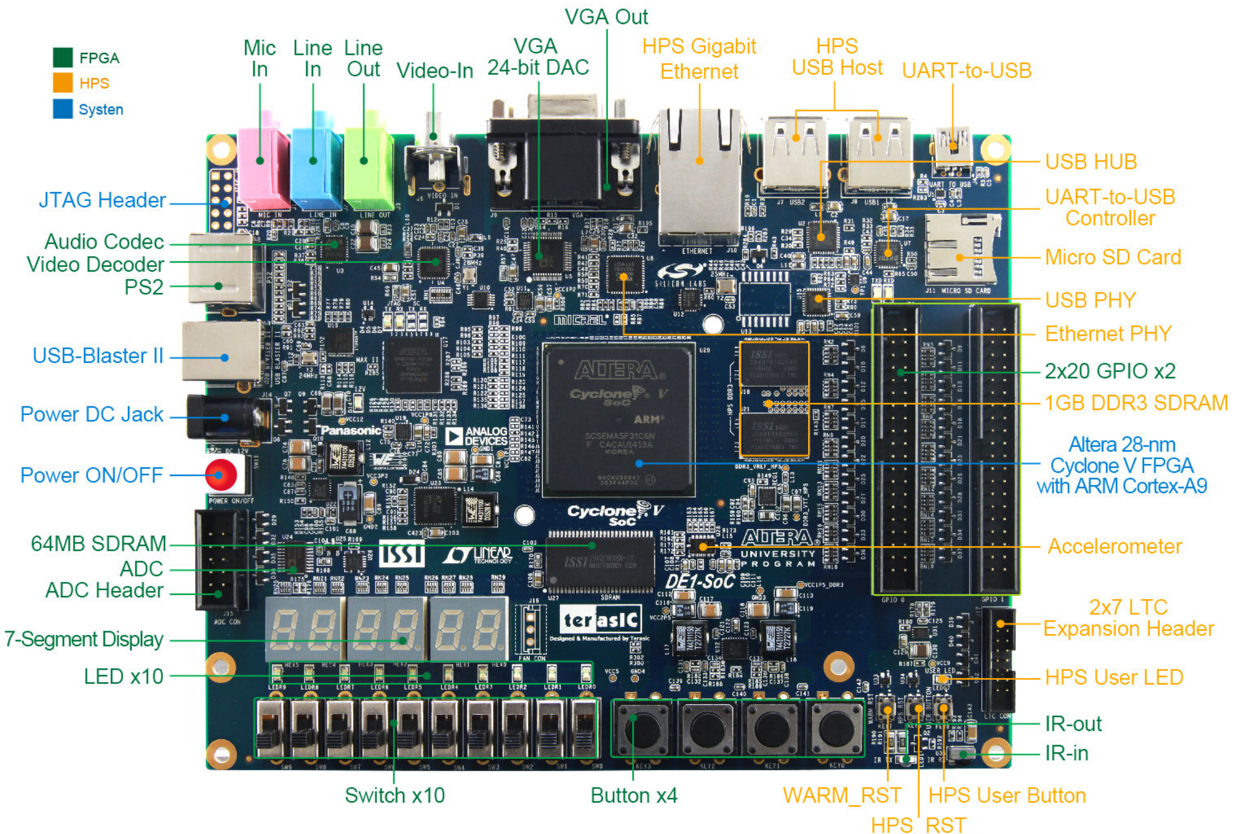


図 2.1　FPGA ボード DE1-SoC[6]

す．本実習では，入出力装置のうち，基板の下側にある以下の装置を使用します．

- LED 10 個（LEDR0〜LEDR9）
- スライドスイッチ 10 個（SW0〜SW9）
- ボタンスイッチ 4 個（KEY0〜KEY3）
- 7 セグメント LED 6 個（HEX0〜HEX5）
- A/D 変換器

また，基板の右側には 2 × 20 ピンの GPIO 拡張ヘッダがあり，基板外部の装置を接続できます．第 4 章では，この拡張ヘッダに外部基板を接続して，基板外部の装置を制御する実習を行います．

2.2　ハードウェアの構築

それでは，Nios II を使用する制御システムのハードウェアを構築していきましょう．図 1.2（p. 3）のように，ハードウェアの構築では **Quartus Prime** と **Platform Designer** を使用します．

2.2.1　構築するハードウェアの概要

構築するハードウェアの概要を図 2.2 に示します．今回は，Nios II を使用して DE1-SoC 上の LED，スライドスイッチ，ボタンスイッチ（4 個のうちリセット用の 1 個を除いた 3 個），7 セグメント LED の入出力制御ができるように，ハードウェアを構成します．FPGA に追加する各 IP コアの役割は次のとおりです．

- **Nios II/e**：システムの中心となる CPU コアです．無償のエコノミーコアを使用します．
- **RAM**：FPGA のロジック要素で構成されるメモリ（オンチップメモリ）です．プログラムの格納先およびスタックとして使用します．
- **JTAG UART**：JTAG 経由でホスト PC と通信します．デバッグ用の標準入出力として使用します．
- **PIO（Parallel I/O）**：パラレルポートです．DE1-SoC 上の入出力装置に接続し，入出力制御に使用します．

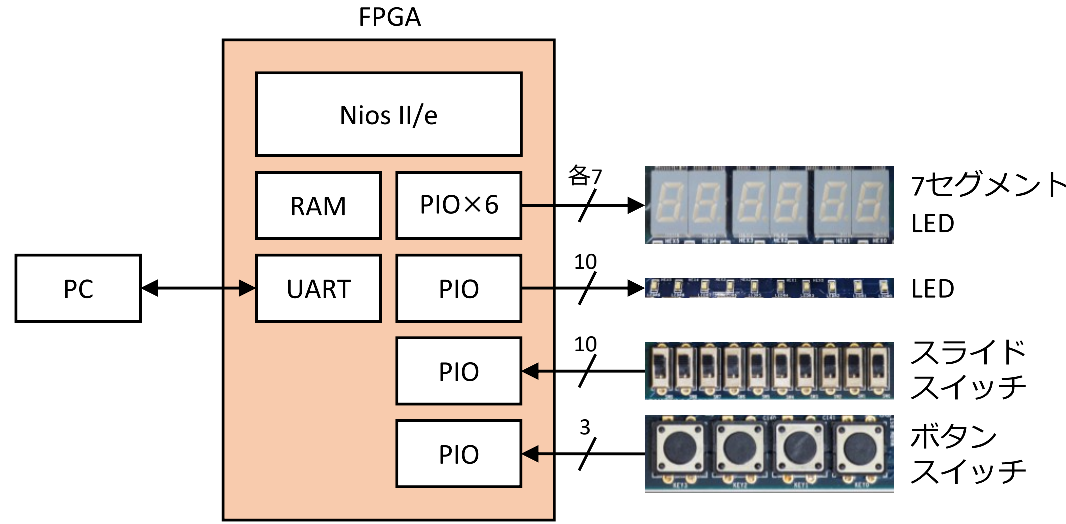


図 2.2　構築するハードウェアの概要

2.2.2　プロジェクトの作成

最初に，Quartus Prime 上でハードウェアデザインのプロジェクトを作成します．

Quartus Prime を起動し，メニューから「File」→「New Project Wizard...」を選択して（図 2.3），新規プロジェクト作成ウィザードを表示します．

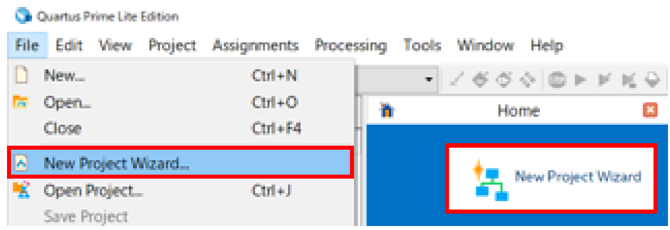


図 2.3　New Project Wizard の選択

作業フォルダ，プロジェクト名，最上位階層名を設定する画面（図 2.4）では，表 2.1 のように設定して，「Finish」ボタンをクリックします．

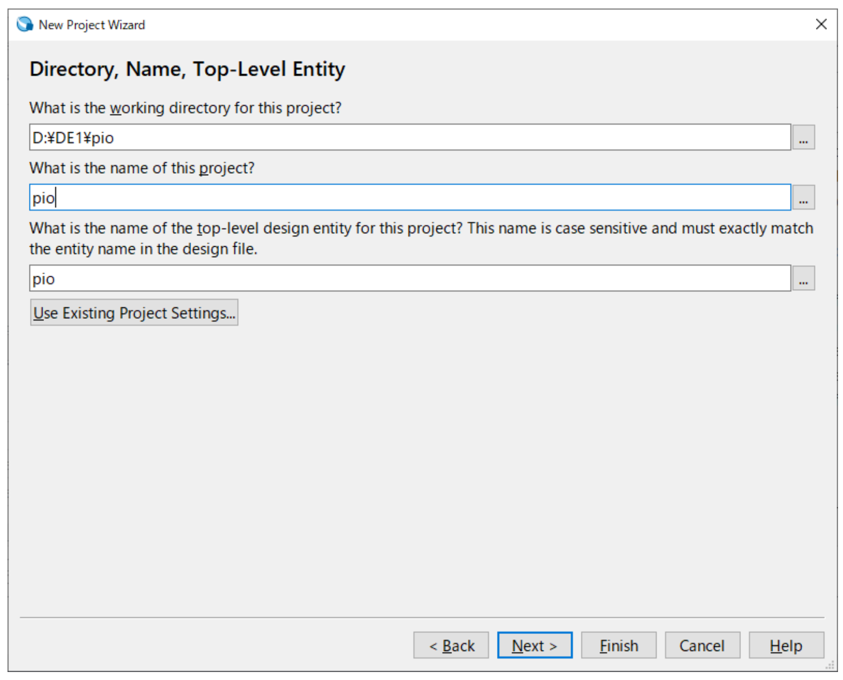


図 2.4　作業フォルダ，プロジェクト名，最上位階層名の設定画面

表 2.1　作業フォルダ，プロジェクト名，最上位階層名の設定

項目	設定値
作業フォルダ（working directory）	`D:\DE1\pio`
プロジェクト名（name of this project）	`pio`
最上位階層名（name of the top-level design entity）	`pio`

続いて表示される「Project Type（プロジェクトの種類）」，「Add Files（ファイルの追加）」の画面では，そのまま「Next」をクリックして次の画面に進みます．

FPGA デバイスを設定する「Family, Device & Board Settings」画面（図 2.5）では，「Board」タブを選択します．Family を「**Cyclone V**」，Development Kit を「**DE1-SoC Board**」に設定すると，適切な FPGA デバイスが選択されます．

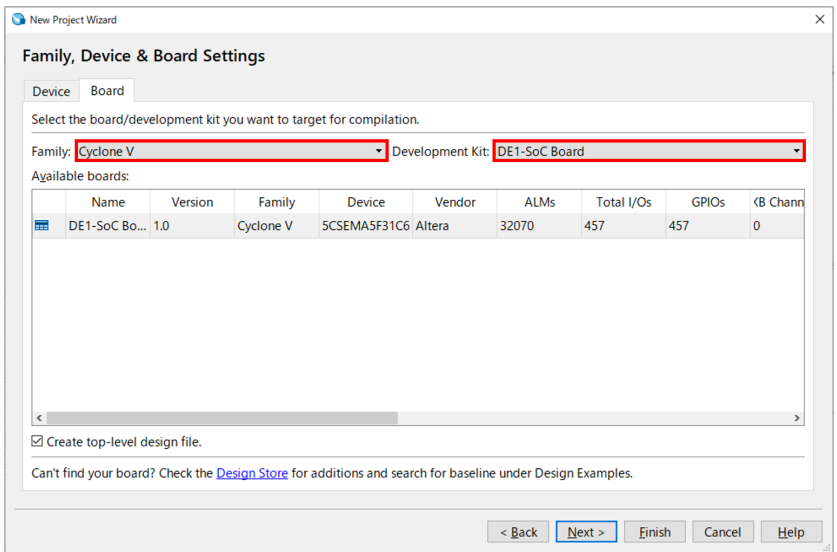


図 2.5　FPGA デバイスの設定

続いて表示される「EDA Tool Settings」の画面では，そのまま「Next」をクリックして次の画面に進みます．

最後に表示される「Summary」画面（図 2.6）で，作成するプロジェクトの概要を確認します．問題がなければ「Finish」をクリックします．誤りがあれば「Back」をクリックして，設定を修正してください．

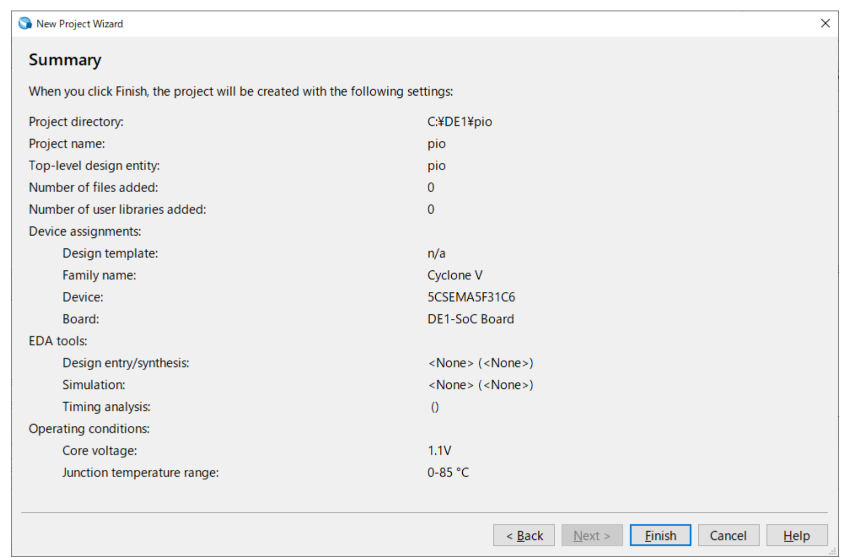


図 2.6　作成するプロジェクトの概要

新規プロジェクト作成ウィザードの完了後，FPGA ピンの電圧レベルを設定します．DE1-SoC に実装されている FPGA は 3.3 V レベルで動作します．これをプロジェクトに設定します．

メニューから「Assignments」→「Device...」（図 2.7）を選択し，「Device」画面を開きます．

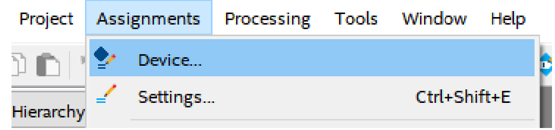


図 2.7　メニュー「Assignments」→「Device...」

「Device」画面では，中央付近の「Device and Pin Options...」ボタン（図 2.8）をクリックして，「Device and Pin Options」画面を開きます．

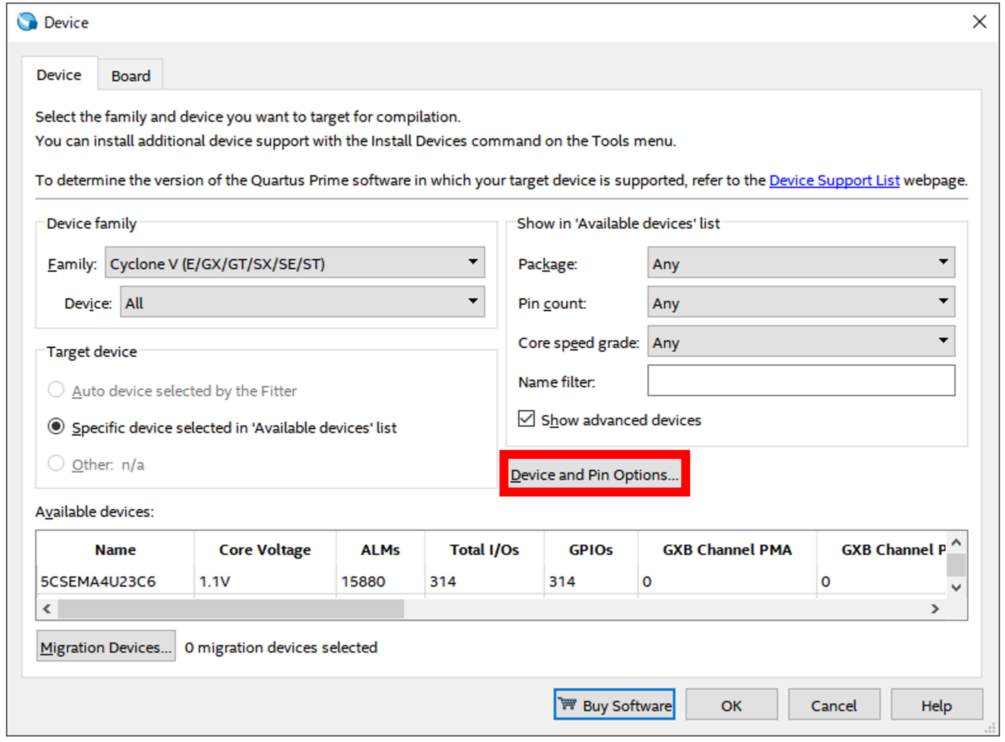


図 2.8　「Device」画面の「Device and Pin Options...」ボタン

「Device and Pin Options」画面では，左側の設定項目一覧から「Voltage」を選択し，「Default I/O standard」を「3.3-V LVCMOS」に設定します（図 2.9）．

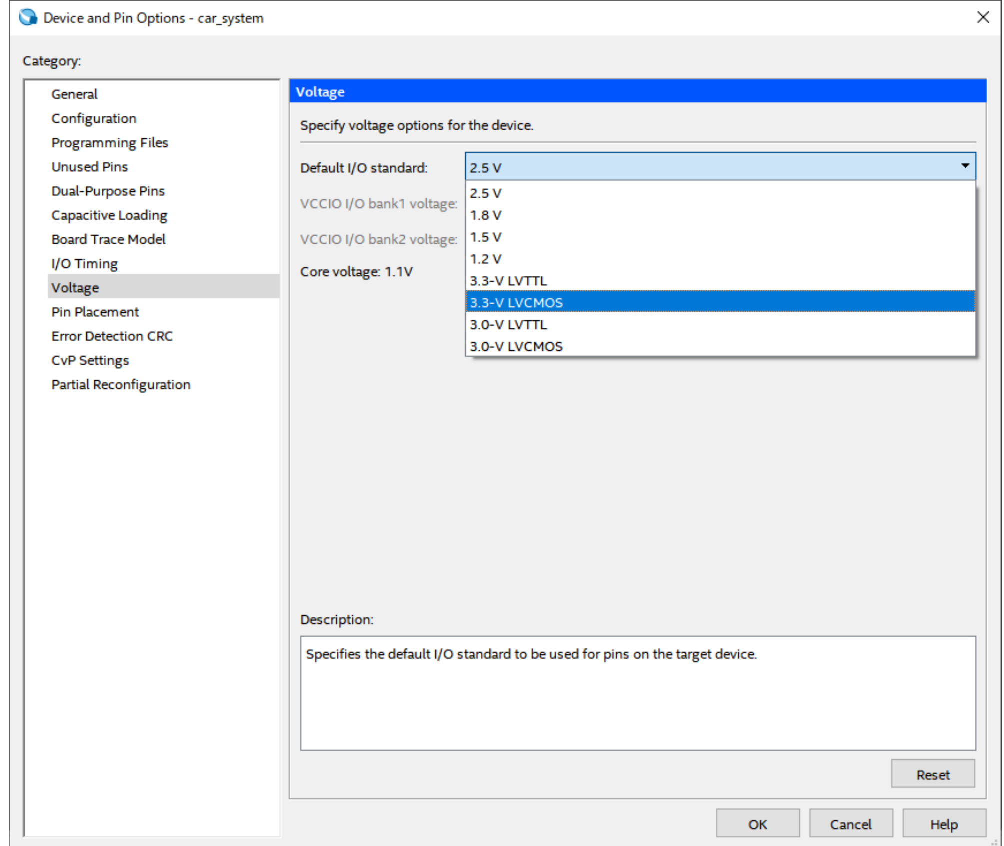


図 2.9　電圧レベルの設定：3.3-V LVCMOS

設定後，2 つの画面で「OK」ボタンをクリックして，設定を保存します．
以上の手順で，ハードウェアデザインのプロジェクトを作成できました．

2.2.3　Nios II および周辺回路の構成

ハードウェアデザインプロジェクトの作成後，Platform Designer を用いて，Nios II および周辺回路を構成します．

Platform Designer の起動

Platform Designer は，Quartus Prime のメニューから起動できます．「Tools」→「Platform Designer」（図 2.10）を選択すると，Platform Designer が起動します．

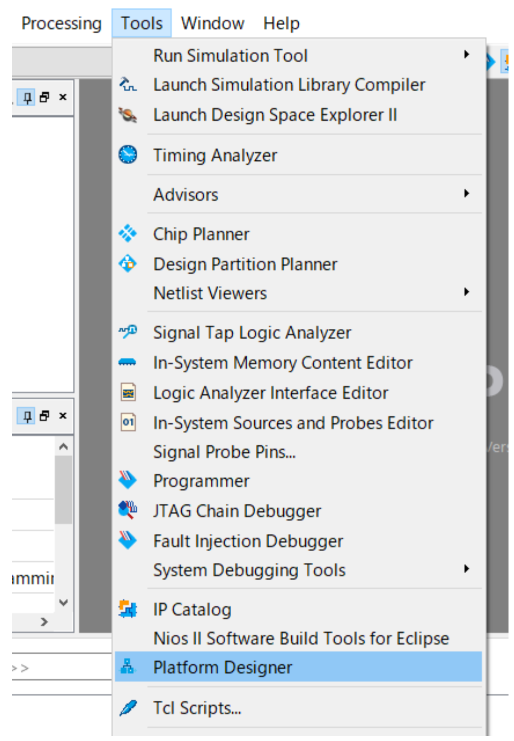


図 2.10　Platform Designer の選択

起動直後の Platform Designer の画面を，図 2.11 に示します．この段階では，マイコンシステムにはクロック源 `clk_0` のみが設置されています．「Export」列の内容から，`clk_0` にはクロック入力 `clk` とリセット入力 `reset` という 2 つの入力ポートがあることを確認できます．

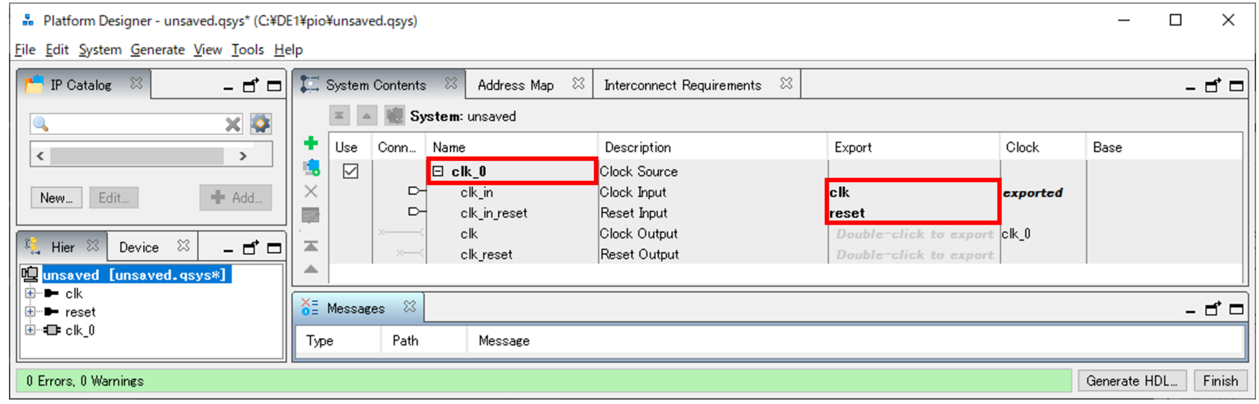


図 2.11　起動直後の Platform Designer の画面

Nios II の追加

CPU コアの Nios II/e を追加します．

画面左側の「IP Catalog」ペインにおいて「Processors and Peripherals」→「Embedded Processors」→「Nios II Processor」を選択した後，「Add」ボタンをクリックして（図 2.12），Nios II を追加します．

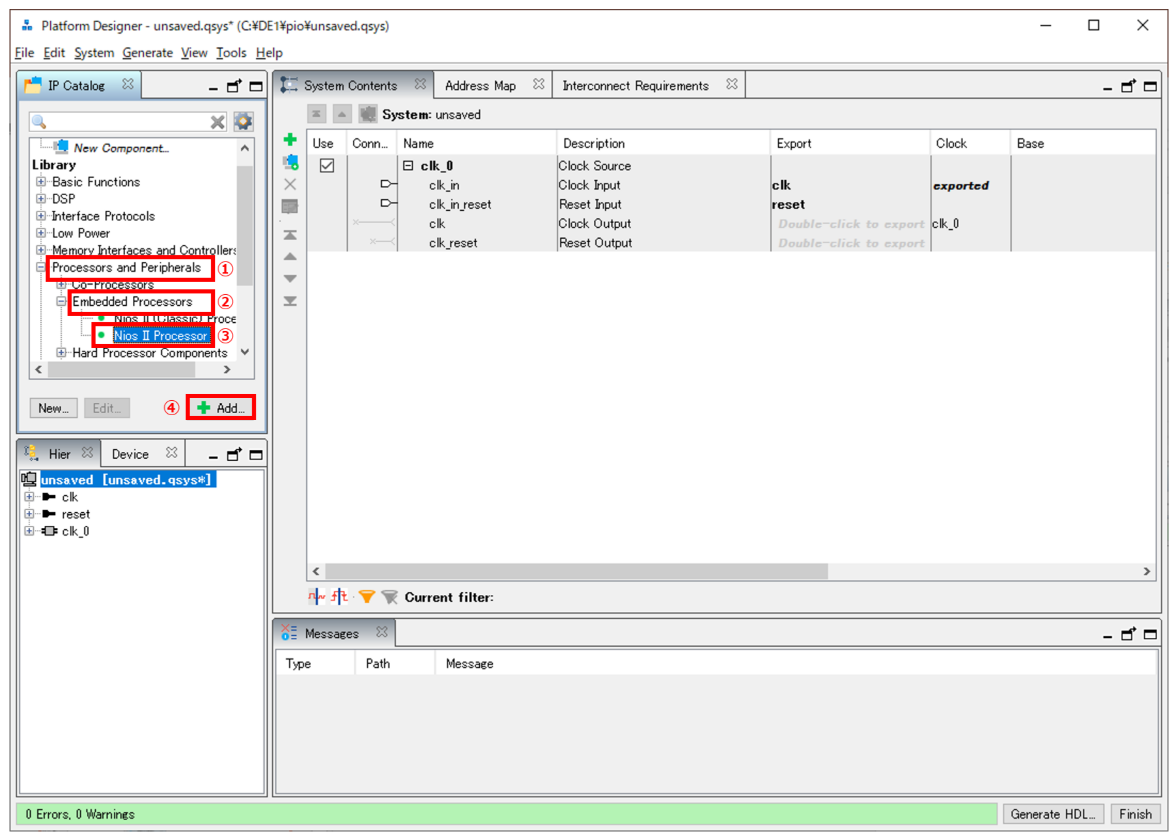


図 2.12　Nios II Processor の選択

続いて表示される Nios II の設定画面（図 2.13）において，「Nios II Core」を「Nios II/e」に設定して，エコノミーコアを選択します．選択後，「Finish」ボタンをクリックして，設定画面を閉じます．

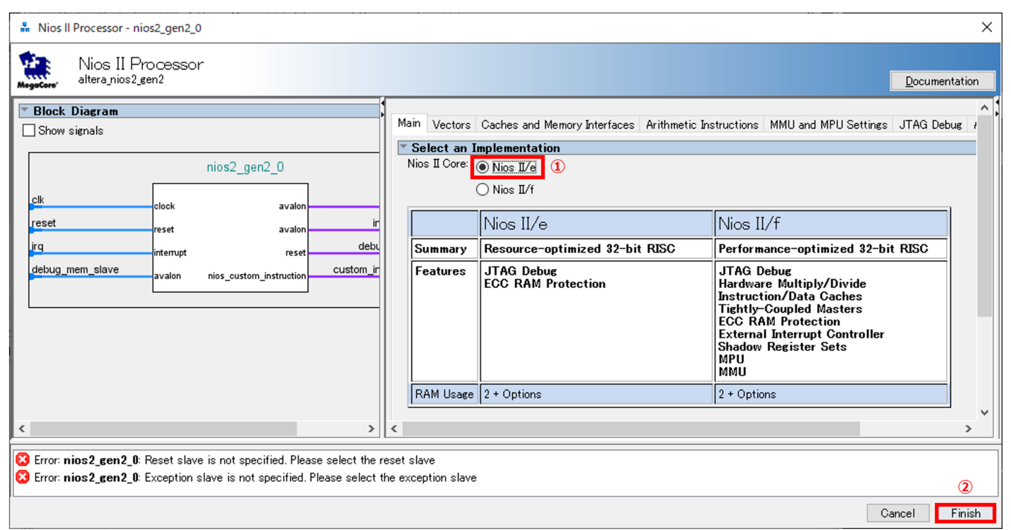


図 2.13　Nios II Core の設定

ここまでの操作で「System Contents」タブの内容が更新され，Nios II が追加されたことを確認できます．下部の「Messages」ペインに複数のエラーが表示されますが，後の作業の終了後に解消されますので，気にせず次の作業に進んでください．

RAM の追加

プログラムの格納先およびスタックとして使用する，オンチップメモリを追加します．

「IP Catalog」ペインにおいて「Basic Functions」→「On Chip Memory」→「On-Chip Memory (RAM or ROM) Intel FPGA IP」を選択した後，「Add」ボタンをクリックして（図 2.14），オンチップメモリを追加します．

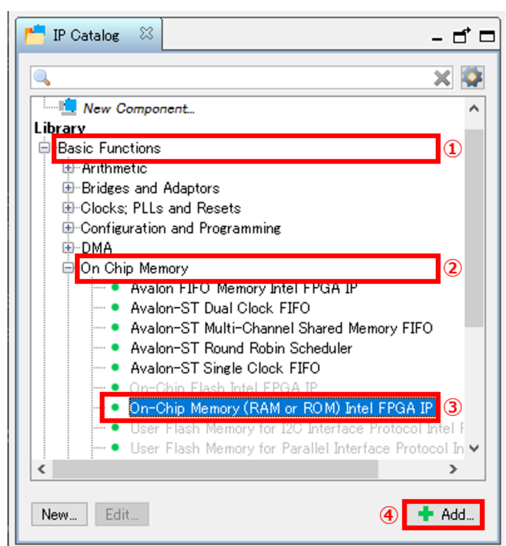


図 2.14　On-Chip Memory の選択

続いてオンチップメモリの設定画面（図 2.15）が表示されます．メモリにデータを書き込めるようにするため，「Memory type」（メモリの種類）を「RAM（Writable）」（既定値）に設定します．また，より多くのプログラムやデータを格納できるようにするため，「Total memory size」を 16384 バイト（16 KB）に設定します．

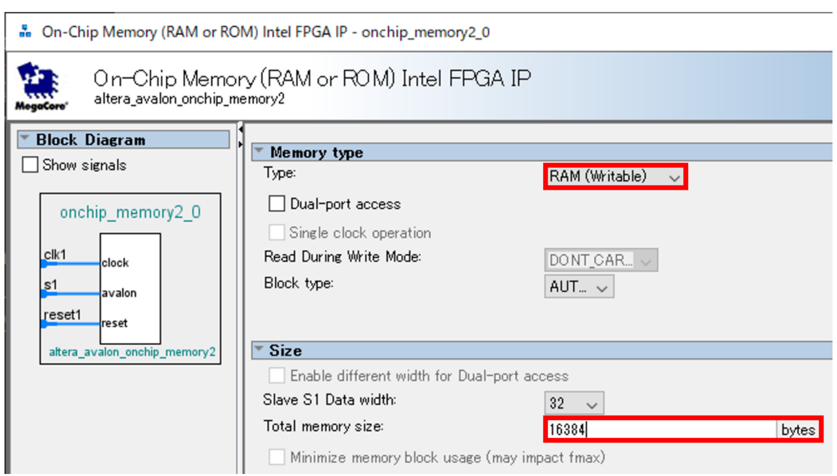


図 2.15　On-Chip Memory の設定

JTAG UART の追加

デバッグ用の標準入出力として使用する，JTAG UART を追加します．

「IP Catalog」ペインにおいて「Interface Protocols」→「Serial」→「JTAG UART Intel FPGA IP」を選択した後，「Add」ボタンをクリックして（図 2.16），JTAG UART を追加します．JTAG UART は既定の設定で使用できるため，続いて表示される設定画面は，そのまま「Finish」ボタンをクリックして閉じてください．

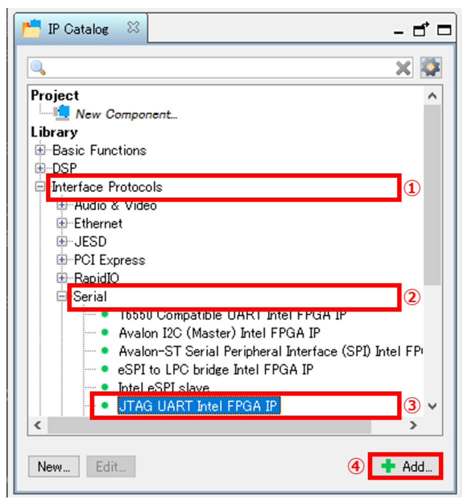


図 2.16　JTAG UART の選択

PIO の追加

入出力装置の制御を行えるようにするため，パラレルポートの IP コアである PIO（Parallel I/O）を追加します．

まず，10 個の LED と対応する PIO を追加してみましょう．「IP Catalog」ペインにおいて「Processors and Peripherals」→「Peripherals」→「PIO (Parallel I/O) Intel FPGA IP」を選択した後，「Add」ボタンをクリックして（図 2.17），PIO を追加します．

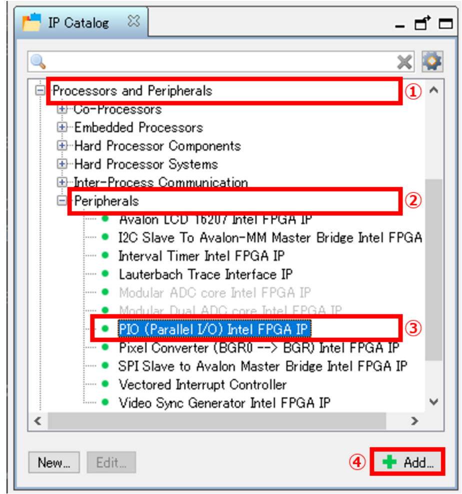


図 2.17　PIO の選択

続いて PIO の設定画面（図 2.18）が表示されます．この PIO を用いて，10 個の LED に対してデジ

タル値を出力したいので，「Width（1–32 bits）」（ビット幅）を 10 に，「Direction」（方向）を「Output」（出力）に設定します．

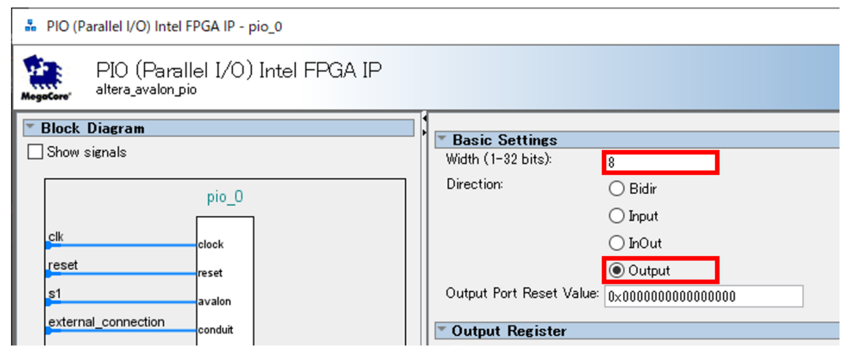


図 2.18　LED 用 PIO の設定

このまま PIO を複数追加していくと，それらは `pio_0`，`pio_1`，`pio_2`，…と連番で命名され，区別しにくくなります．そこで，追加した PIO を「`led`」に改名して，LED 用であることを明示しましょう．「System Contents」内の「`pio_0`」の上で右クリックしてメニューを表示し，「Rename」を選択します．テキストボックスが表示されたら「`led`」と入力し，Enter キーを押して確定させます．以上の手順を図 2.19 に示します．

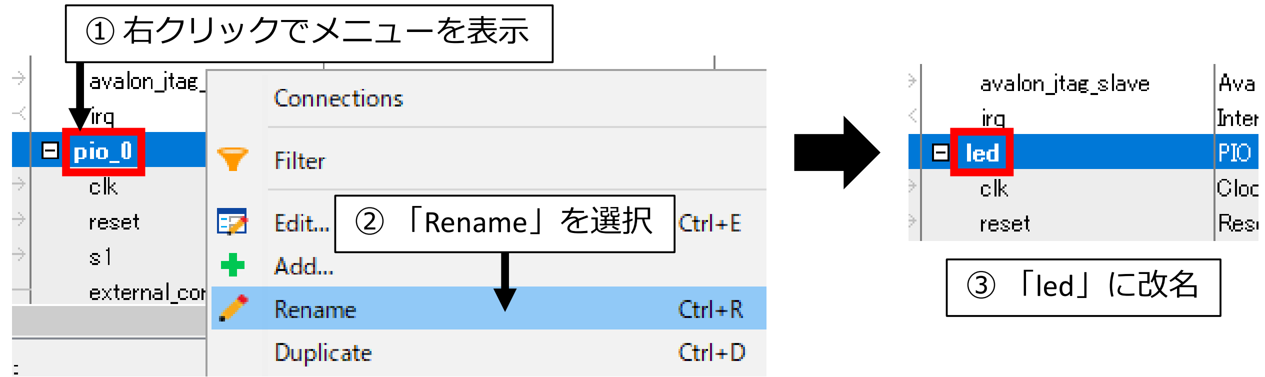


図 2.19　LED 用 PIO の改名

同様の手順で，他の入出力装置に対応する PIO も追加します．表 2.2 に示す PIO をマイコンシステムに追加してください．

表 2.2　追加する PIO

名称	ビット幅	方向	用途
`slide_sw`	10	Input（入力）	スライドスイッチ（SW0〜SW9）用
`button_sw`	3	Input（入力）	ボタンスイッチ（KEY1〜KEY3）用
`hex_0`	7	Output（出力）	7 セグメント LED「HEX0」用
`hex_1`	7	Output（出力）	7 セグメント LED「HEX1」用
`hex_2`	7	Output（出力）	7 セグメント LED「HEX2」用
`hex_3`	7	Output（出力）	7 セグメント LED「HEX3」用
`hex_4`	7	Output（出力）	7 セグメント LED「HEX4」用
`hex_5`	7	Output（出力）	7 セグメント LED「HEX5」用

IP コア間の配線

以上の作業で追加した IP コアに対して，システムの動作に必要な配線を行います．

最初に，clk_0 からのリセット信号を各 IP コアに接続します．リセット信号については自動配線機能が用意されているため，この機能を利用すると簡単に配線できます．メニューから「System」→「Create Global Reset Network」を選択して自動配線を行います（図 2.20）．

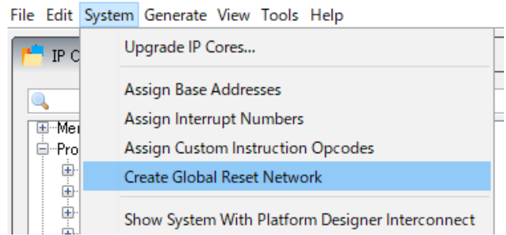


図 2.20　リセット信号の自動配線

次に，clk_0 からのクロック信号を各 IP コアに接続します．クロック信号の配線は手動で行う必要があります．clk_0 の出力 clk と各 IP コアの入力 clk（オンチップメモリのみ clk1）との交点「○」をクリックして「●」に変化させると，両者が接続された状態になります．追加したすべての IP コアに対して，この作業を行ってください．図 2.21 に示す状態になれば，クロック信号の配線は完了です．

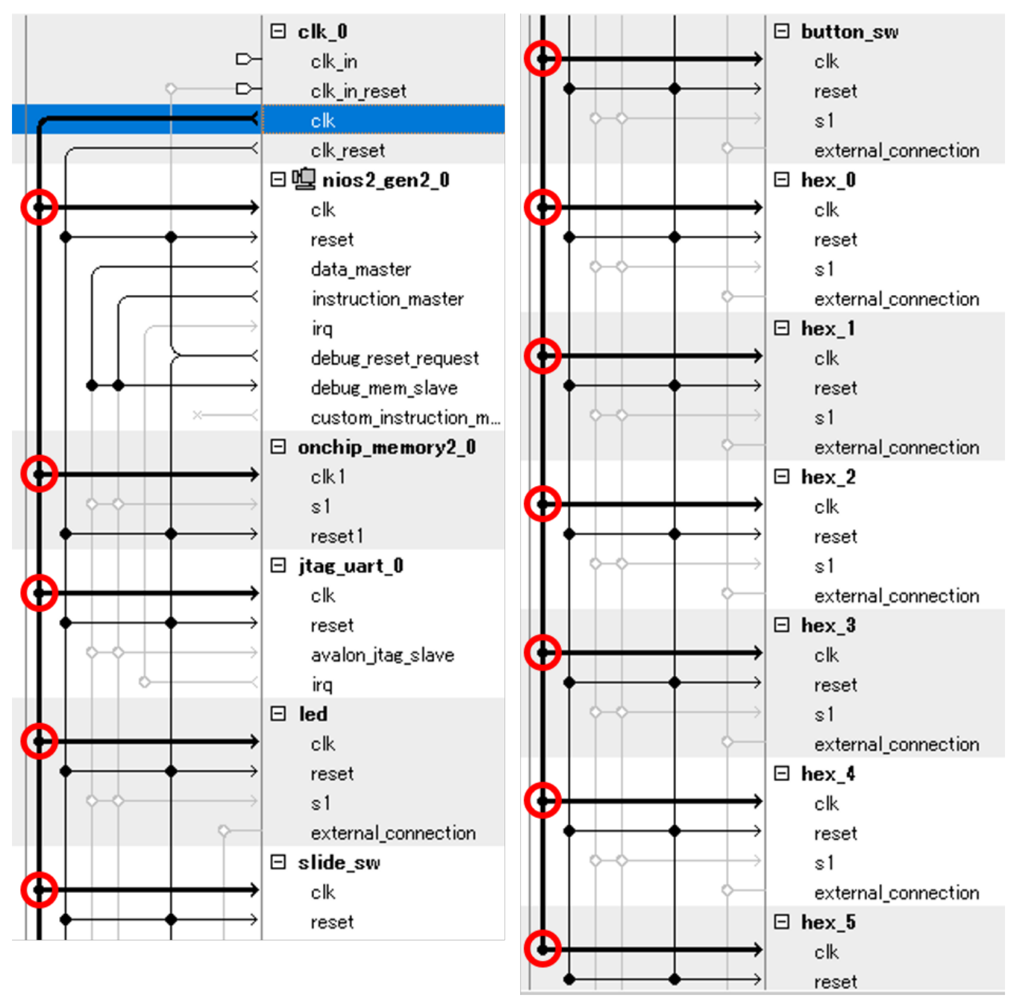


図 2.21　クロック信号の配線

続いて，Nios II から周辺回路を制御できるようにするため，これらをバス接続します．Nios II ではハーバードアーキテクチャが採用されており，メモリ領域やバス（信号線）が命令（instruction）用とデータ（data）用とに分離されています．Platform Designer においては，Nios II コアの出力 `instruction_master` が命令バス，出力 `data_master` がデータバスと対応します．これを踏まえて，オンチップメモリを `instruction_master` と `data_master` の両方に接続し，その他の周辺回路を `data_master` のみと接続してください[*1]．図 2.22 に示す状態になれば，バス接続は完了です．

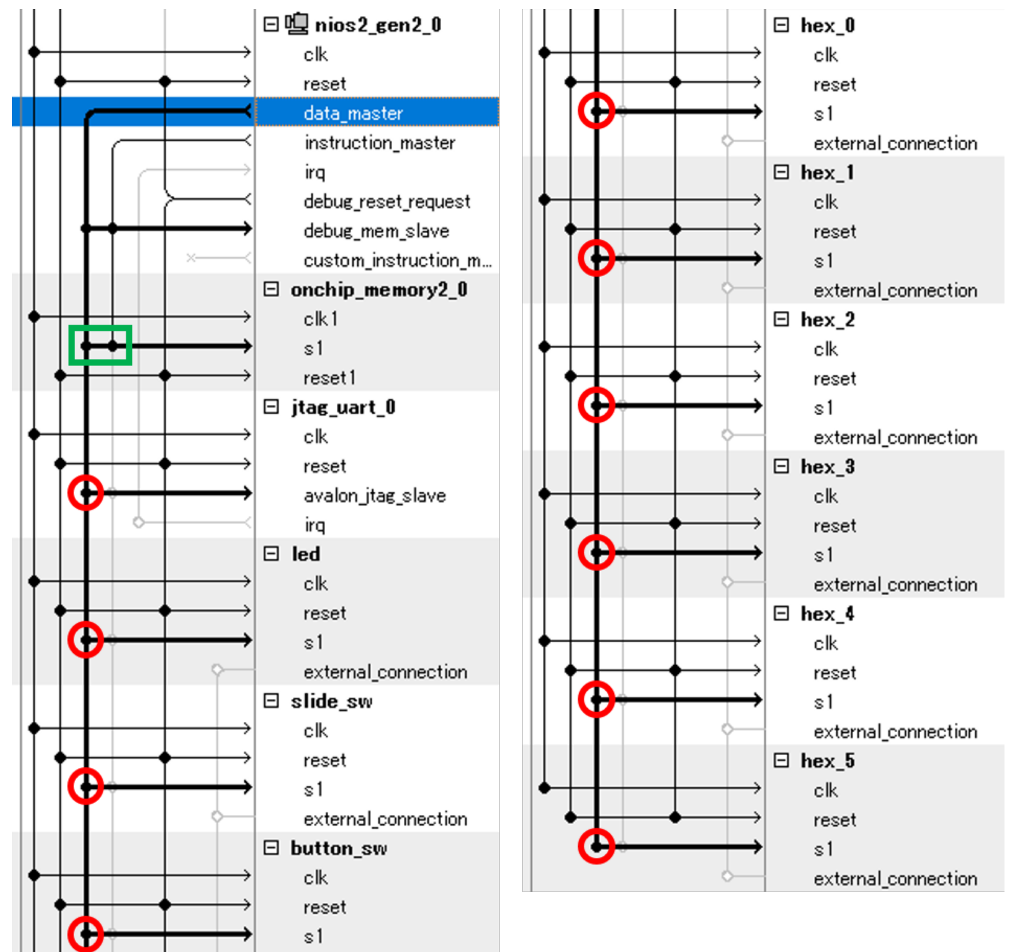


図 2.22　Nios II と周辺回路とのバス接続．オンチップメモリは，緑色の長方形で示すように，`instruction_master` と `data_master` の両方に接続する．その他の周辺回路は，赤色の円で示すように，`data_master` のみと接続する．

最後に，JTAG UART の IRQ を Nios II に接続します．本実習ではこの IRQ を使用しませんが，警告を止めるために必要です．「`jtag_uart_0`」の「`irq`」行における「IRQ」列の部分をダブルクリックすると，接続されます（図 2.23）．その際に表示される「0」は割り込み番号であり，必要に応じて変更できます．

[*1] プログラムとデータの両方を読み書きする必要があるため，メモリには命令バスとデータバスの両方を接続する必要があります．一方で，その他の周辺回路にはプログラムを配置する必要がないため，データバスのみを接続します．

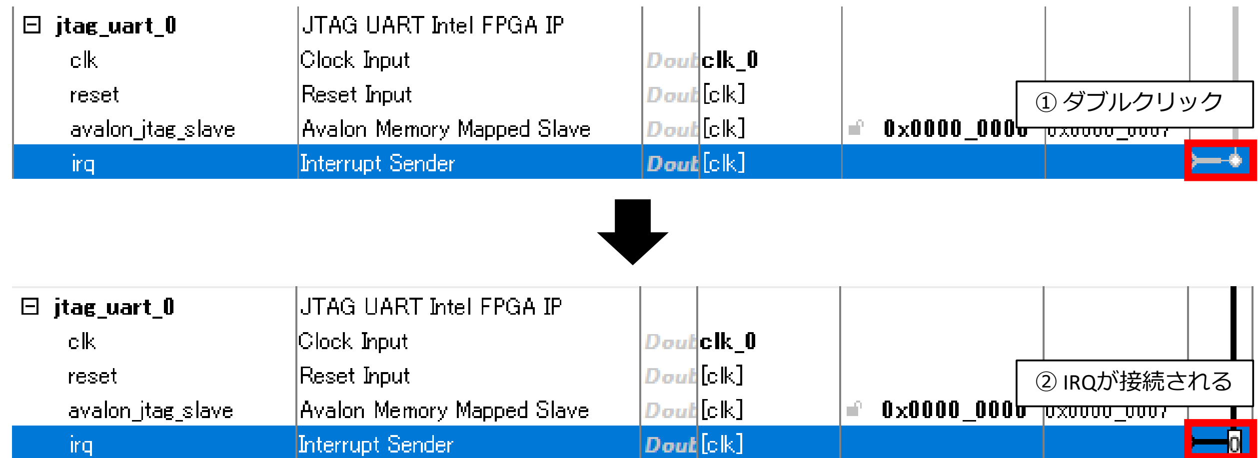


図 2.23　JTAG UART の IRQ の接続

Nios II のベクタ設定

Nios II を動作させるには，リセットベクタ（reset vector）と例外ベクタ（exception vector）を設定する必要があります．Nios II とメモリの接続によってその設定が可能となったため，ここで設定します．

「nios2_gen2_0」の上で右クリックしてメニューを表示し，「Edit...」を選択します（図 2.24）．続いて表示される Nios II の設定画面において「Vectors」タブを選択し，「Reset vector memory」および「Exception vector memory」を「onchip_memory2_0.s1」に設定します（図 2.25）．

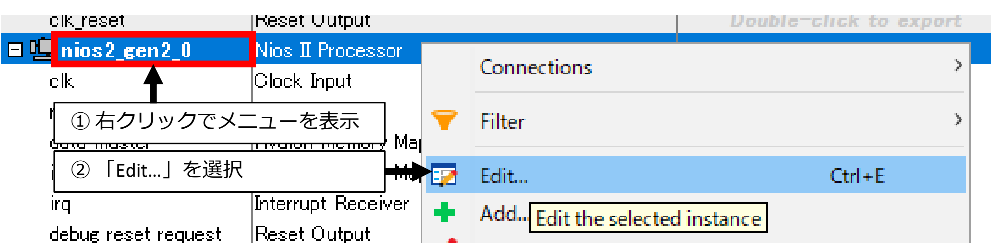


図 2.24　Nios II 設定画面の表示手順

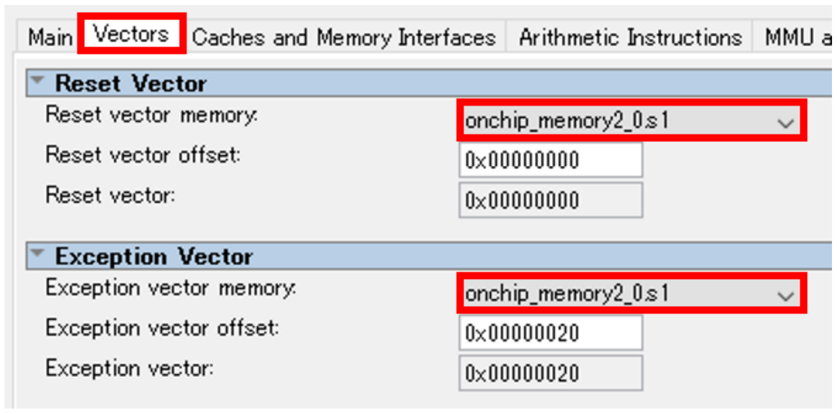


図 2.25　Nios II のベクタ設定

PIO のポート出力設定

Verilog コードから PIO を参照するには，PIO のポートを外部に出力する必要があります．ここでは，そのための設定を行います．

まず，LED 用の PIO「led」のポートを外部に出力してみましょう．led の external_connection 行において，「Export」列の部分をダブルクリックします．ダブルクリックすると，出力するポートの名前を設定できるようになるので，既定値のまま Enter を押して確定させます．「Export」列に「led_external_connection」が表示されれば，LED 用の PIO のポート出力設定は完了です（図 2.26）．

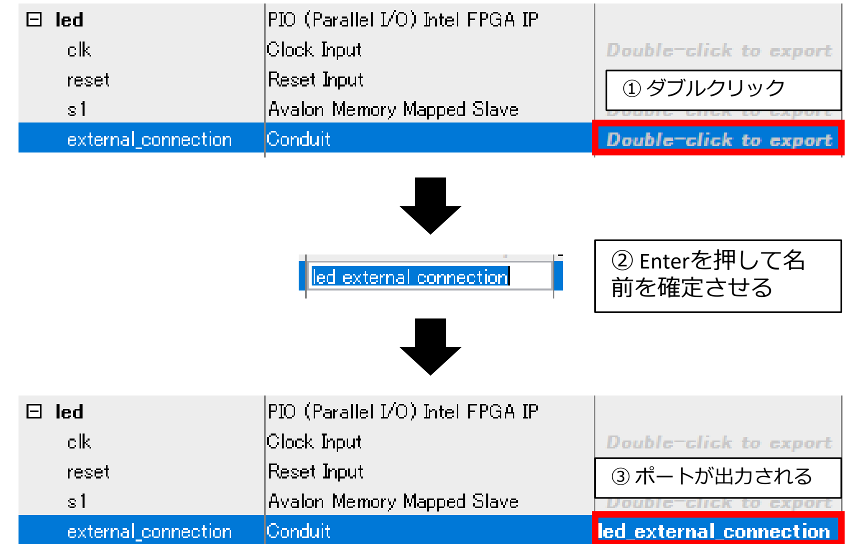


図 2.26　LED 用 PIO のポート出力設定

以上の作業を，追加したすべての PIO に対して行います．完了後の状態を図 2.27 に示します．

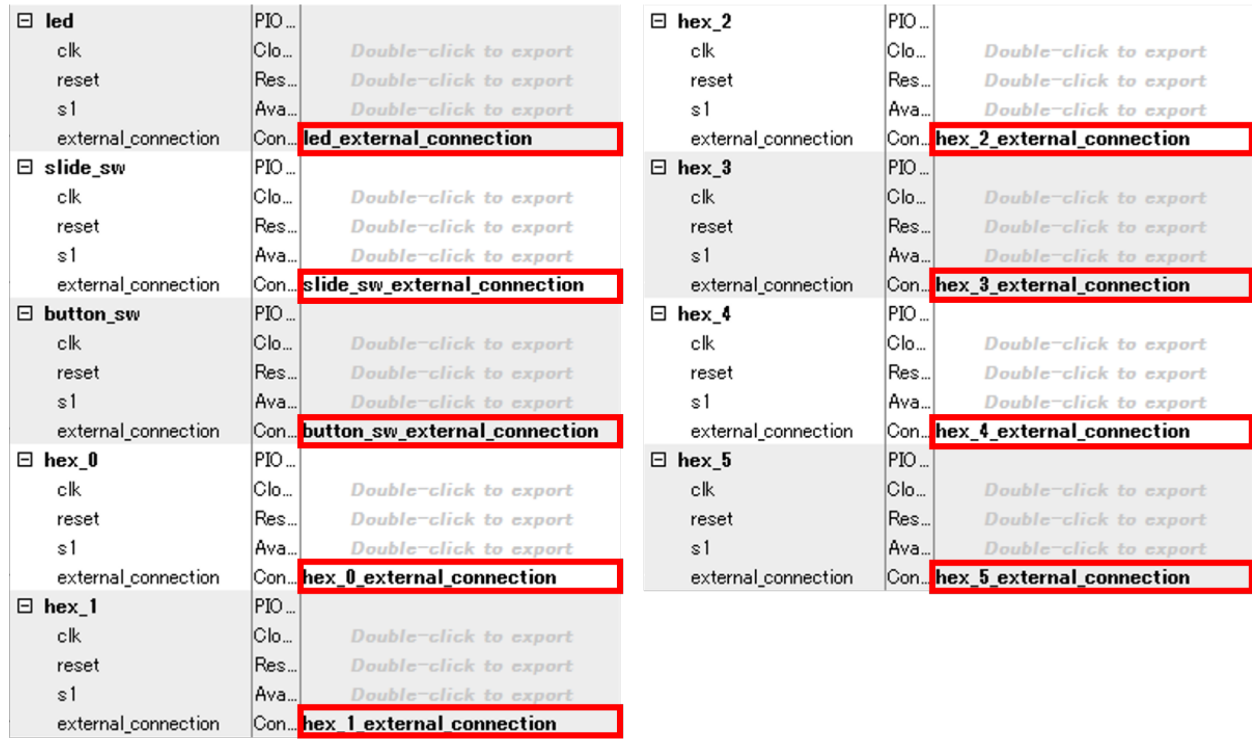


図 2.27　PIO のポート出力設定完了後の状態

アドレスの割り当て

以上で各 IP コアの設定および配線は完了しましたが，メモリや周辺回路のアドレスが重なっているため，複数のエラー表示が出ています．「System Contents」タブの隣にある「Address Map」タブを選択すると，各 IP コアのアドレスが一覧表示され，アドレスの重なりを確認できます（図 2.28）．そこで，最後にアドレス割り当てを行い，エラーを解消します．

System Contents　Address Map　Interconnect Requirements

System: pio　Path: nios2_gen2_0

	nios2_gen2_0.data_master	nios2_gen2_0.instruction_master
button_sw.s1	0x0000 - 0x000f	
hex_0.s1	0x0000 - 0x000f	
hex_1.s1	0x0000 - 0x000f	
hex_2.s1	0x0000 - 0x000f	
hex_3.s1	0x0000 - 0x000f	
hex_4.s1	0x0000 - 0x000f	
hex_5.s1	0x0000 - 0x000f	
jtag_uart_0.avalon_jtag_slave	0x0000 - 0x0007	
led.s1	0x0000 - 0x000f	
nios2_gen2_0.debug_mem_slave	0x0800 - 0x0fff	0x0800 - 0x0fff
onchip_memory2_0.s1	0x0000 - 0x3fff	0x0000 - 0x3fff
slide_sw.s1	0x0000 - 0x000f	

図 2.28　アドレスの重なり

アドレスの編集は「System Contents」タブの「Base」列から行えます．まず，オンチップメモリのアドレスを `0x10000` 番地から割り当ててみましょう．「`onchip_memory2_0`」の「`s1`」行において，「Base」列の部分をダブルクリックします．ダブルクリックすると先頭アドレスを設定できるようになるので，「`0x10000`」を入力し，Enter を押して確定させます．末尾のアドレスが自動で計算されて `0x13fff` となり，メモリ容量が 16 KB（= `0x4000` バイト）であることを確認できます．以上の手順を図 2.29 に示し

ます．

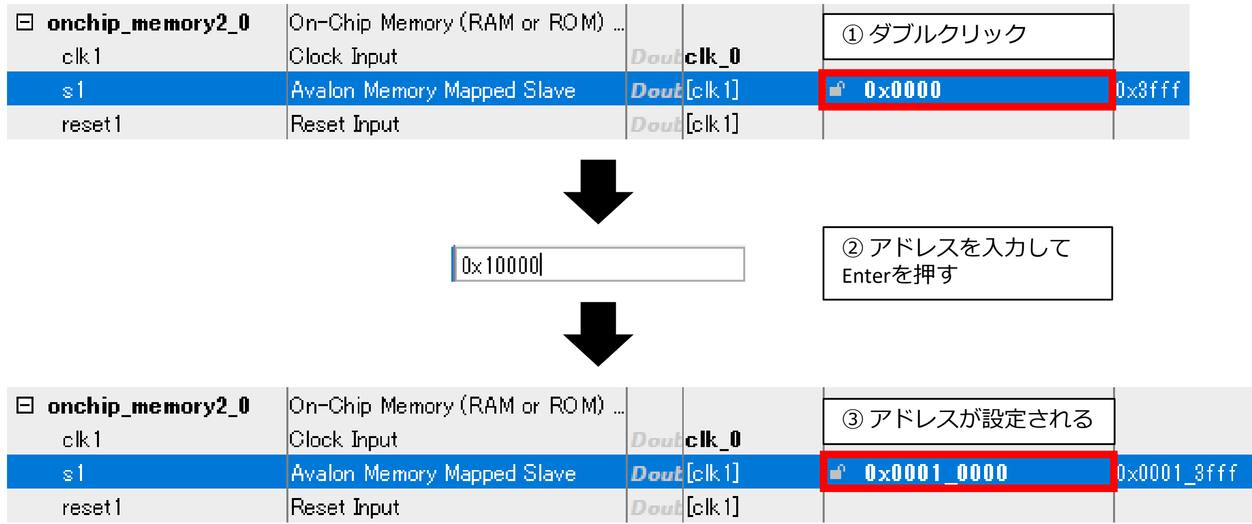


図 2.29　アドレスの割り当て

同様の手順で，他の周辺回路のアドレスも設定します．表 2.3 に従って，先頭アドレスを設定します．設定後，「Address Map」タブの内容が図 2.30 のようになったことを確認してください．

表 2.3　各 IP コアの先頭アドレス

IP コア名	信号名	先頭アドレス
jtag_uart_0	avalon_jtag_slave	0x20000
led	s1	0x30000
slide_sw	s1	0x30100
button_sw	s1	0x30200
hex_0	s1	0x31000
hex_1	s1	0x31100
hex_2	s1	0x31200
hex_3	s1	0x31300
hex_4	s1	0x31400
hex_5	s1	0x31500

System Contents | Address Map | Interconnect Requirements

System: pio **Path:** jtag_uart_0.irq

	nios2_gen2_0.data_master	nios2_gen2_0.instruction_master
button_sw.s1	0x0003_2000 - 0x0003_200f	
hex_0.s1	0x0003_1000 - 0x0003_100f	
hex_1.s1	0x0003_1100 - 0x0003_110f	
hex_2.s1	0x0003_1200 - 0x0003_120f	
hex_3.s1	0x0003_1300 - 0x0003_130f	
hex_4.s1	0x0003_1400 - 0x0003_140f	
hex_5.s1	0x0003_1500 - 0x0003_150f	
jtag_uart_0.avalon_jtag_slave	0x0002_0000 - 0x0002_0007	
led.s1	0x0003_0000 - 0x0003_000f	
nios2_gen2_0.debug_mem_slave	0x0000_0800 - 0x0000_0fff	0x0000_0800 - 0x0000_0fff
onchip_memory2_0.s1	0x0001_0000 - 0x0001_3fff	0x0001_0000 - 0x0001_3fff
slide_sw.s1	0x0003_0100 - 0x0003_010f	

図 2.30 アドレス割り当て後の Address Map

以上の手順で，Nios II および周辺回路を構成できました．下部の「Messages」ペインにエラーおよび警告が出ていないことを確認して，次の Verilog コードの生成に進んでください．

Verilog コードの生成

構成した Nios II および周辺回路の Verilog コードを生成し，ハードウェアデザインのプロジェクトに追加できるようにします．

メニューから「Generate」→「Generate HDL...」を選択します（図 2.31）．

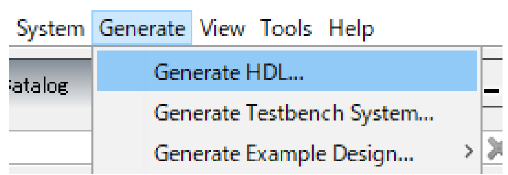


図 2.31 「Generate HDL...」の選択

続いて HDL 生成の設定画面が表示されますが，ここではそのまま「Generate」ボタンをクリックします．その後，「Save changes to ～?」と，これまでの構成設定を保存するか確認された場合は，「Save」をクリックして保存します．ファイル名は「`nios2_pio.qsys`」としてください．

構成設定の保存後，Verilog コードの生成が始まります．正常にコードが生成されれば，図 2.32 の画面が表示されますので，「Close」をクリックして閉じます．

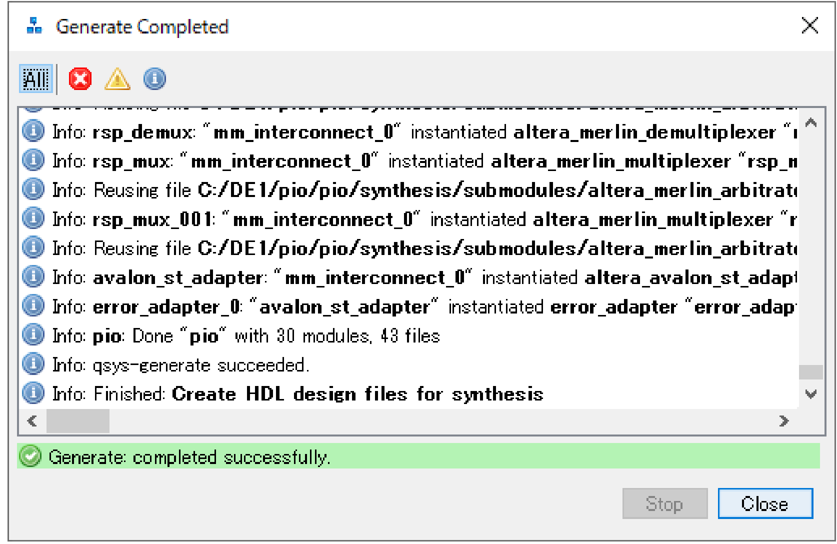


図 2.32　Verilog コードの生成完了

Nios II および周辺回路のプロジェクトへの追加

Quartus Prime に戻って，生成したコードをハードウェアデザインのプロジェクトに追加します．メニューから「Add/Remove files in Project...」を選択します（図 2.33）．

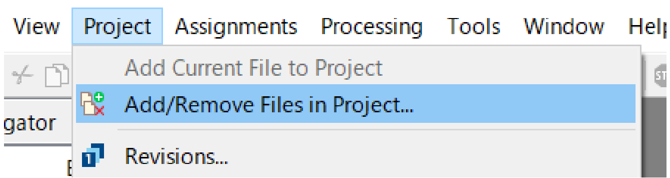


図 2.33　「Add/Remove files in Project...」の選択

続いて表示される画面において，「...」ボタンをクリックしてファイル選択画面を表示し，「`nios2_pio\synthesis\nios2_pio.qip`」（プロジェクトのフォルダを起点とした相対パス）を追加します（図 2.34）．

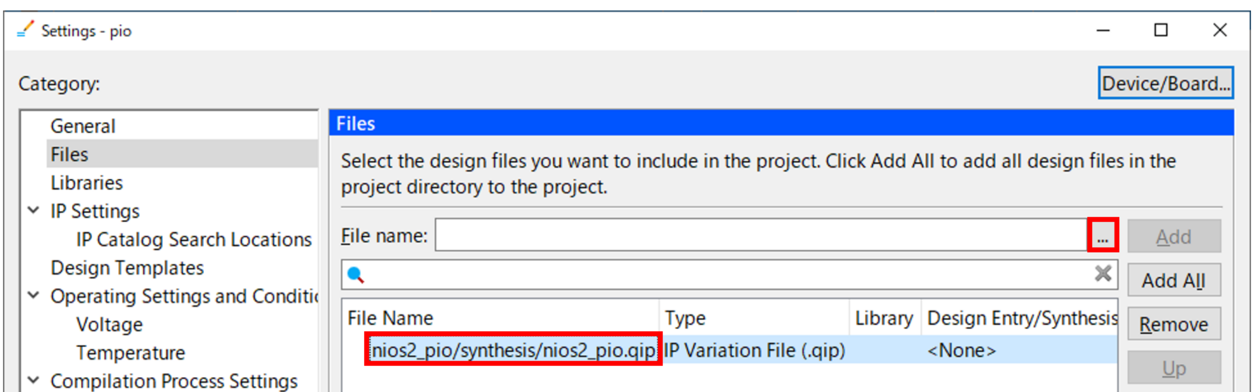


図 2.34　`nios2_pio.qip` の追加

2.2.4　最上位階層の作成と論理合成

Nios II を含むマイコンシステムは，ハードウェアデザインの最上位階層に組み込んで使用します．この最上位階層を作成し，システム全体の回路を論理合成します．

最上位階層のモジュール名はプロジェクト名と同じ名前の「pio」とし，これを Verilog ファイル pio.v に作成します．このモジュールでは，クロック，リセットおよび入出力装置用のポートを定義し，構築したマイコンシステムと接続します．リスト 2.1 の内容を pio.v に入力してください．Nios II および周辺回路をまとめたモジュール nios2_pio については，Platform Designer のメニュー「Generate」→「Show Instantiation Template...」（図 2.35）から表示できる記述例（図 2.36）をコピー＆ペーストすると，簡単に記述できます．

リスト 2.1　ハードウェアデザインの最上位階層（pio.v）

```
1  module pio (
2    input CLK,
3    input RESET_N,
4
5    input [9:0] SW,
6    input [3:1] KEY_N,
7
8    output [9:0] LEDR,
9    output [6:0] HEX0,
10   output [6:0] HEX1,
11   output [6:0] HEX2,
12   output [6:0] HEX3,
13   output [6:0] HEX4,
14   output [6:0] HEX5
15 );
16
17 // ボタンスイッチの値を反転して正論理に変える
18 wire [2:0] KEY;
19 assign KEY = ~KEY_N[3:1];
20
21 nios2_pio u0 (
22   .button_sw_external_connection_export (KEY),
23   .clk_clk                              (CLK),
24   .hex_0_external_connection_export     (HEX0),
25   .hex_1_external_connection_export     (HEX1),
26   .hex_2_external_connection_export     (HEX2),
27   .hex_3_external_connection_export     (HEX3),
28   .hex_4_external_connection_export     (HEX4),
29   .hex_5_external_connection_export     (HEX5),
30   .led_external_connection_export       (LEDR),
```

```
31   .reset_reset_n                        (RESET_N),
32   .slide_sw_external_connection_export  (SW)
33 );
34
35 endmodule
```

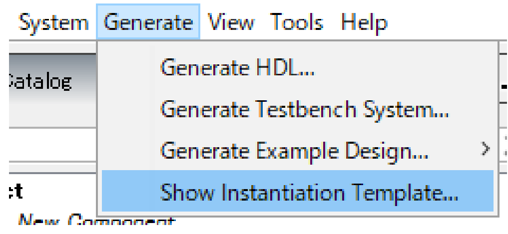


図 2.35　「Show Instantiation Template...」の選択

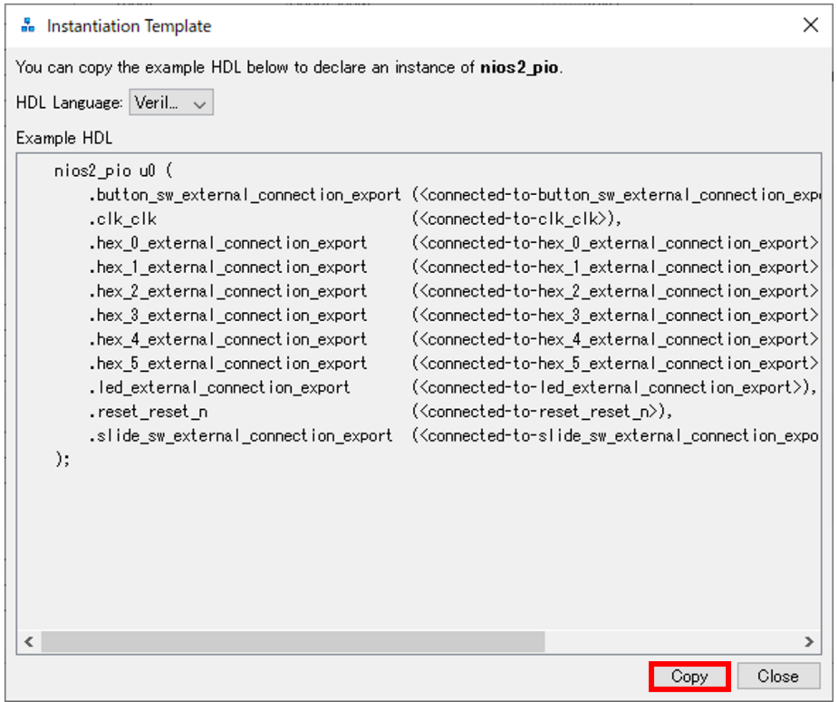


図 2.36　モジュール nios2_pio の記述例のコピー

最上位階層の作成後，左側にある「Tasks」ペインの「Analysis & Synthesis」（図 2.37）をダブルクリックして，論理合成を行います．論理合成に失敗した場合は，最上位階層のソースコードを確認し，修正してください．

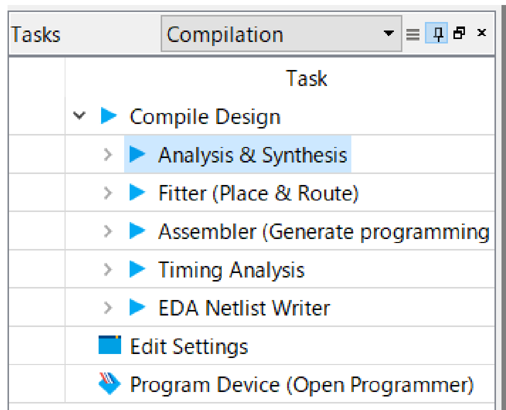


図 2.37　Analysis & Synthesis

2.2.5　ピンアサインとコンパイル

論理合成を行った後，ピンアサインとコンパイルを行い，FPGA へプログラミングできるようにします．

ピンアサインで必要になる，信号とピンとの対応を表 2.4 に示します．これらを手作業で割り当てることもできますが，量が非常に膨大でありミスしやすいため，ここではピンアサイン設定ファイルを取り込む方法を紹介します．

まず，Windows のエクスプローラーを使用して，指導員が配布したピンアサイン設定ファイル `pio_pin_assignment.qsf` を `D:\DE1\pio\` 内にコピーします．続いて，Quartus Prime に戻り，メニューから「Assignments」→「Import Assignments...」を選択します（図 2.38）．

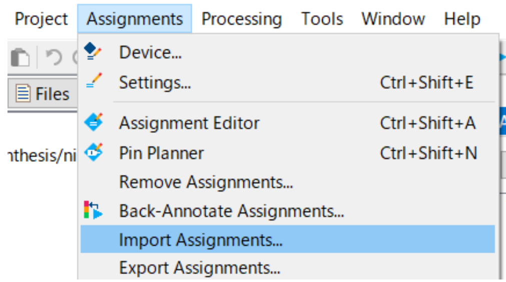


図 2.38　Import Assignments

選択すると，図 2.39 に示す画面が表示されます．「...」ボタンをクリックして `D:\DE1\pio\pio_pin_assignments.qsf` を選択します．「OK」ボタンをクリックすると，ピンアサイン設定が取り込まれます．

表 2.4　ピンの割り当て

Node Name	Direction	Location	割り当て先の機能
CLK	Input	PIN_AF14	50 MHz クロック
RESET_N	Input	PIN_AA14	ボタンスイッチ KEY0（負論理）
SW[0]	Input	PIN_AB12	スライドスイッチ SW0
SW[1]	Input	PIN_AC12	スライドスイッチ SW1
SW[2]	Input	PIN_AF9	スライドスイッチ SW2
SW[3]	Input	PIN_AF10	スライドスイッチ SW3
SW[4]	Input	PIN_AD11	スライドスイッチ SW4
SW[5]	Input	PIN_AD12	スライドスイッチ SW5
SW[6]	Input	PIN_AE11	スライドスイッチ SW6
SW[7]	Input	PIN_AC9	スライドスイッチ SW7
SW[8]	Input	PIN_AD10	スライドスイッチ SW8
SW[9]	Input	PIN_AE12	スライドスイッチ SW9
KEY_N[1]	Input	PIN_AA15	ボタンスイッチ KEY1（負論理）
KEY_N[2]	Input	PIN_W15	ボタンスイッチ KEY2（負論理）
KEY_N[3]	Input	PIN_Y16	ボタンスイッチ KEY3（負論理）
LEDR[0]	Output	PIN_V16	LEDR0
LEDR[1]	Output	PIN_W16	LEDR1
LEDR[2]	Output	PIN_V17	LEDR2
LEDR[3]	Output	PIN_V18	LEDR3
LEDR[4]	Output	PIN_W17	LEDR4
LEDR[5]	Output	PIN_W19	LEDR5
LEDR[6]	Output	PIN_Y19	LEDR6
LEDR[7]	Output	PIN_W20	LEDR7
LEDR[8]	Output	PIN_W21	LEDR8
LEDR[9]	Output	PIN_Y21	LEDR9
HEX0[0]	Output	PIN_AE26	7 セグメント LED HEX0（負論理）
HEX0[1]	Output	PIN_AE27	
HEX0[2]	Output	PIN_AE28	
HEX0[3]	Output	PIN_AG27	
HEX0[4]	Output	PIN_AF28	
HEX0[5]	Output	PIN_AG28	
HEX0[6]	Output	PIN_AH28	
HEX1[0]	Output	PIN_AJ29	7 セグメント LED HEX1（負論理）
HEX1[1]	Output	PIN_AH29	
HEX1[2]	Output	PIN_AH30	
HEX1[3]	Output	PIN_AG30	
HEX1[4]	Output	PIN_AF29	
HEX1[5]	Output	PIN_AF30	
HEX1[6]	Output	PIN_AD27	
HEX2[0]	Output	PIN_AB23	7 セグメント LED HEX2（負論理）
HEX2[1]	Output	PIN_AE29	
HEX2[2]	Output	PIN_AD29	
HEX2[3]	Output	PIN_AC28	
HEX2[4]	Output	PIN_AD30	
HEX2[5]	Output	PIN_AC29	
HEX2[6]	Output	PIN_AC30	
HEX3[0]	Output	PIN_AD26	7 セグメント LED HEX3（負論理）
HEX3[1]	Output	PIN_AC27	
HEX3[2]	Output	PIN_AD25	
HEX3[3]	Output	PIN_AC25	
HEX3[4]	Output	PIN_AB28	
HEX3[5]	Output	PIN_AB25	
HEX3[6]	Output	PIN_AB22	
HEX4[0]	Output	PIN_AA24	7 セグメント LED HEX4（負論理）
HEX4[1]	Output	PIN_Y23	
HEX4[2]	Output	PIN_Y24	
HEX4[3]	Output	PIN_W22	
HEX4[4]	Output	PIN_W24	
HEX4[5]	Output	PIN_V23	
HEX4[6]	Output	PIN_W25	
HEX5[0]	Output	PIN_V25	7 セグメント LED HEX5（負論理）
HEX5[1]	Output	PIN_AA28	
HEX5[2]	Output	PIN_Y27	
HEX5[3]	Output	PIN_AB27	
HEX5[4]	Output	PIN_AB26	
HEX5[5]	Output	PIN_AA26	
HEX5[6]	Output	PIN_AA25	

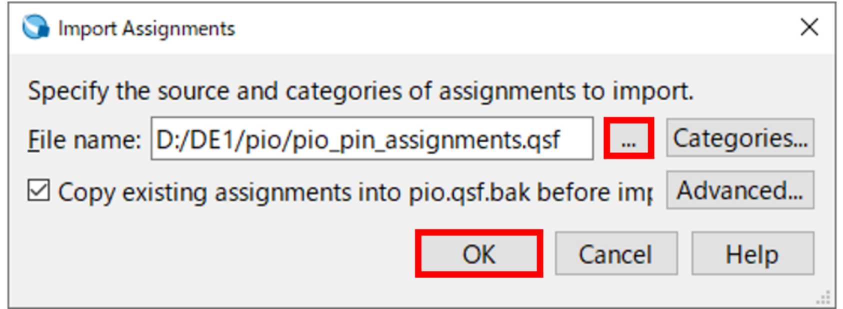


図 2.39　取り込むピンアサイン設定ファイルの指定

ピンアサイン設定の取り込み後，メニューの「Assignments」→「Pin Planner」から Pin Planner を起動して，図 2.40 のようにピンが割り当てられたことを確認してください．

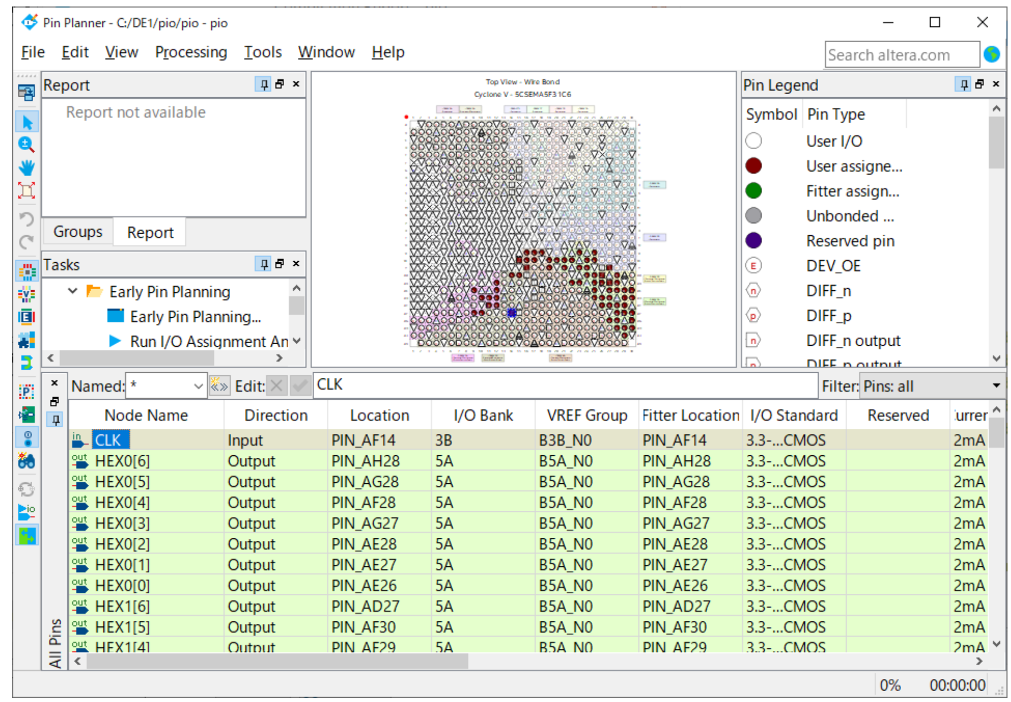


図 2.40　ピンアサイン設定取り込み後の Pin Planner

ピンを割り当てた後，「Tasks」ペインの「Compile Design」（図 2.41）をダブルクリックして回路をコンパイルします．

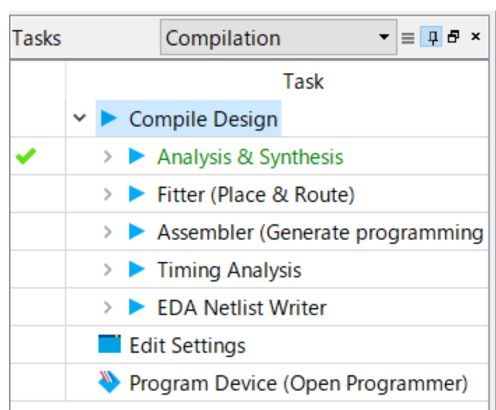


図 2.41　Compile Design

2.2.6　FPGA へのプログラミング

コンパイルしたハードウェアデザインを FPGA にプログラムします．

DE1-SoC に実装されている FPGA の Cyclone V には，ARM マイコンと FPGA の 2 つの部分が存在します．そのため，2 つのうち FPGA の部分にハードウェアデザインをプログラムするように指定する操作が必要となります．

まず DE1-SoC を PC に接続します．図 2.42 のように，USB Blaster 端子に USB ケーブルを接続し，電源 DC ジャックに AC アダプタを接続します．USB ケーブルの他端は PC に，AC アダプタの電源プラグは電源タップのコンセントに接続します．接続後，赤色の電源スイッチを押して，電源を入れます．

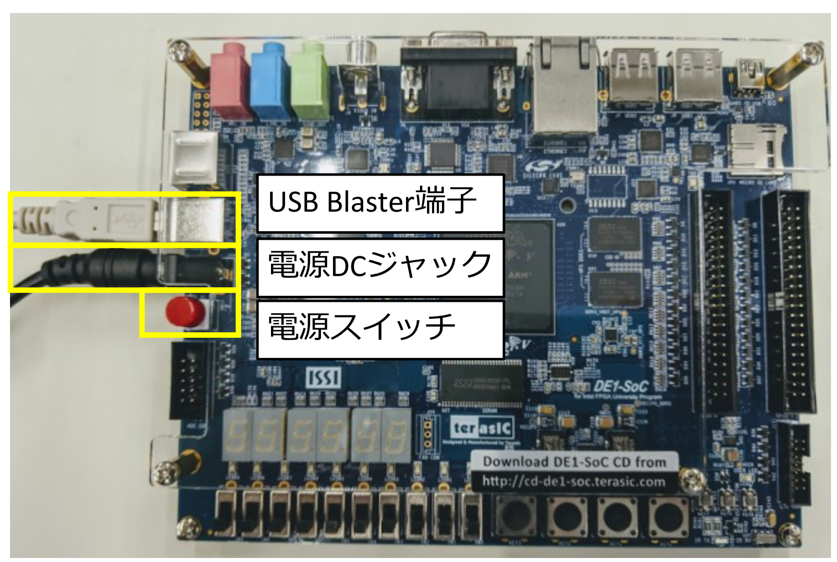


図 2.42　DE1-SoC の各端子および電源スイッチの配置

電源を入れたら，Programmer を使用して FPGA にハードウェアデザインをプログラムします．「Tasks」ペインの「Program Device (Open Programmer)」（図 2.43）をダブルクリックして，Programmer を起動します．

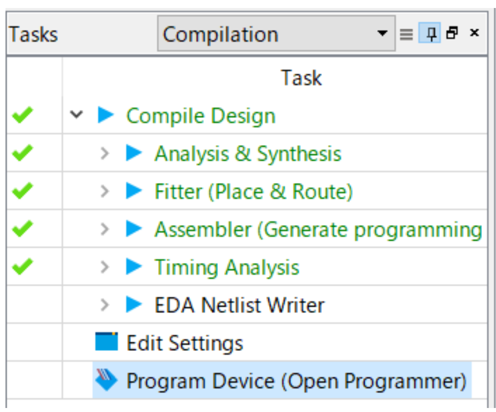


図 2.43　Program Device (Open Programmer)

Programmer を起動すると，図 2.44 の画面が表示されます．「Hardware Setup...」をクリックして，ハードウェア設定画面を表示します．

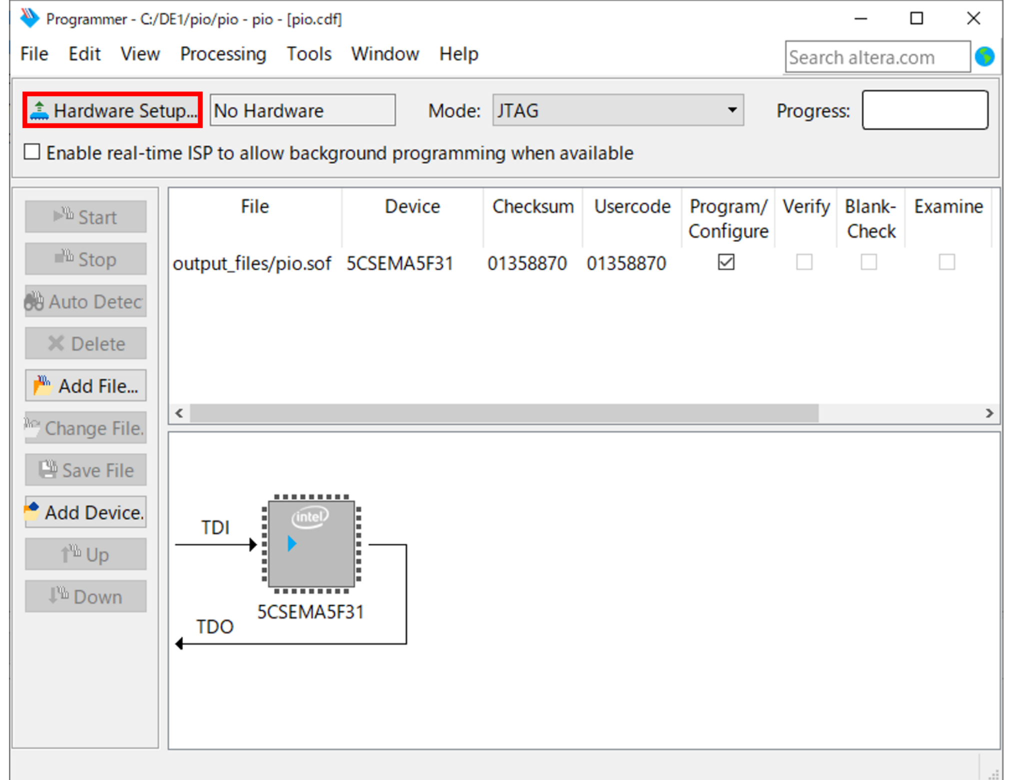


図 2.44　Programmer の画面

ハードウェア設定画面において,「Currently selected hardware」を「DE-SoC」に設定します（図 2.45）.

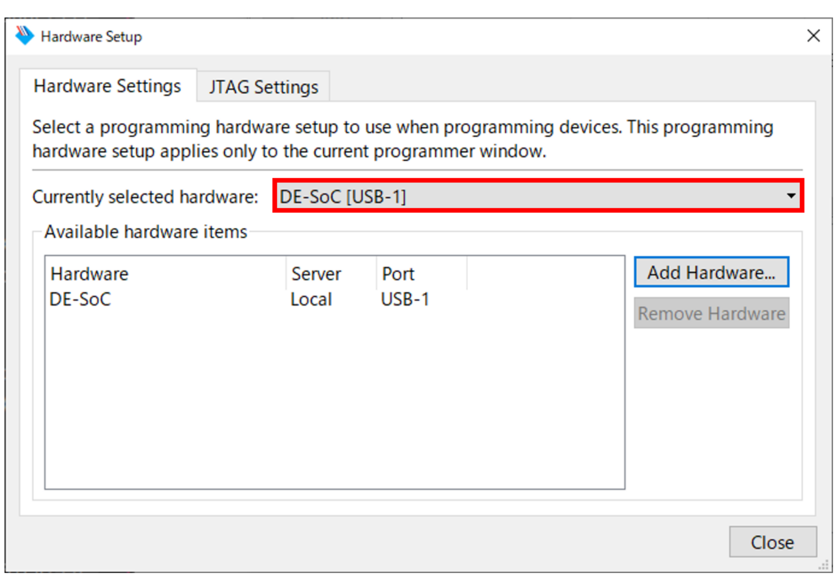


図 2.45　「Currently selected hardware」の設定

「Start」でハードウェアデザインをプログラムしたいところですが，FPGA デバイスの構造が誤って認識されており，失敗してしまいます．そこで，これを修正します．左側の「Auto Detect」ボタン（図 2.46）をクリックします．

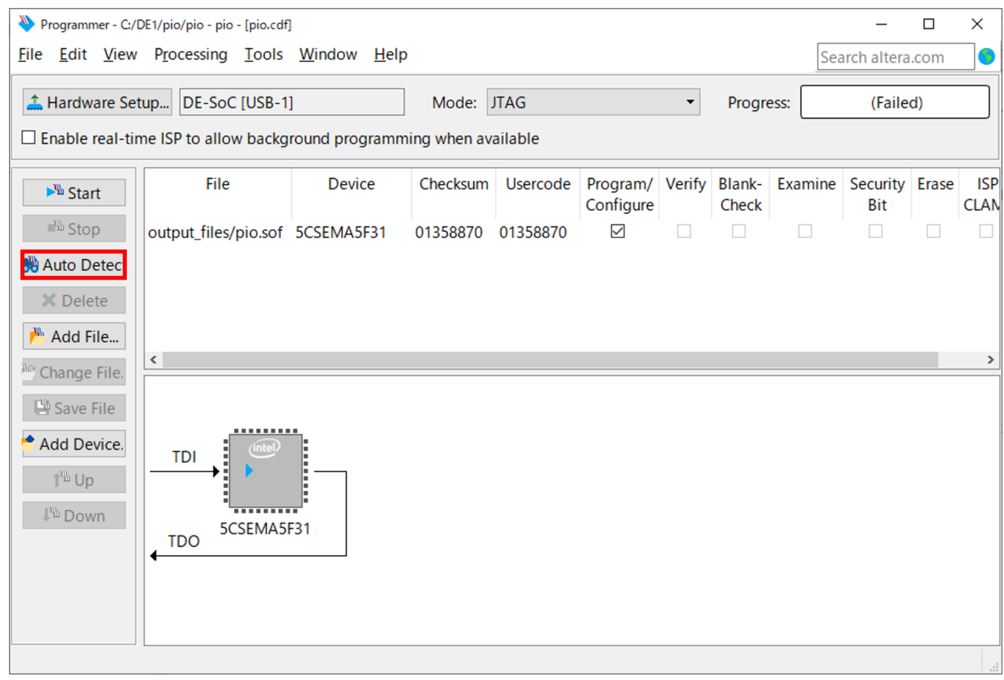


図 2.46　「Auto Detect」ボタン

「Auto Detect」ボタンをクリックすると，図 2.47 に示すデバイス選択画面が表示されます．「5CSEMA5」を選択して「OK」をクリックします．

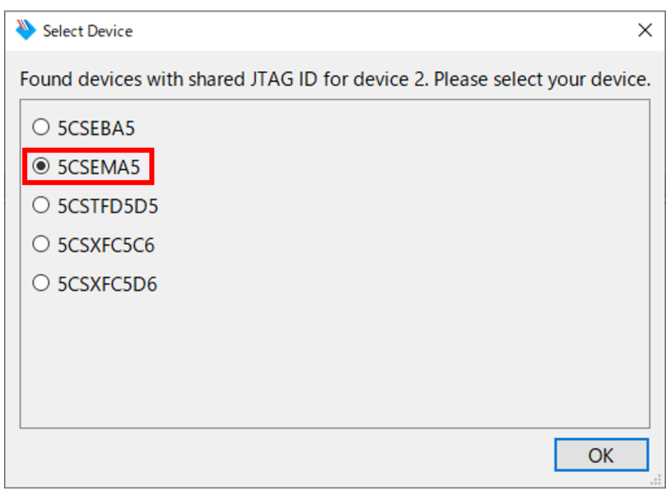


図 2.47　デバイス選択画面

続いて，検出されたデバイスが設定と一致しないため，更新するか確認する画面が表示されます（図 2.48）．「Yes」をクリックして更新します．

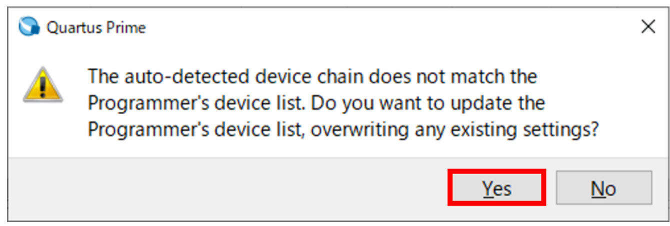


図 2.48　デバイス設定更新の確認

デバイスの自動検出後，2 つのデバイス「SOCVHPS」と「5CSEMA5」が表示されます（図 2.49）．「5CSEMA5」を選択して，「Change File...」ボタンをクリックします．

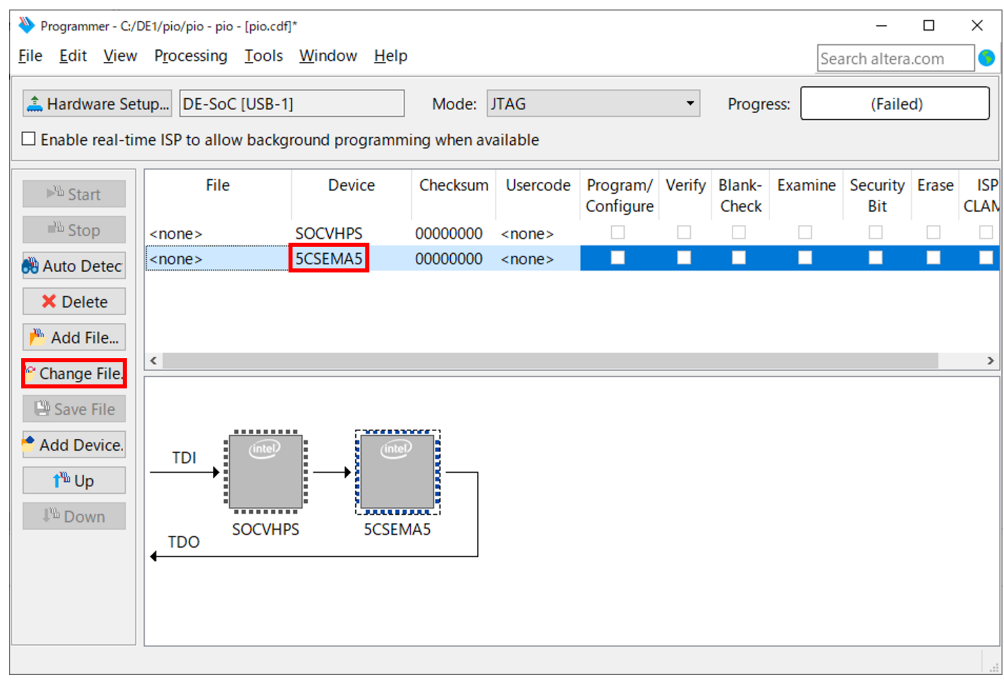


図 2.49　デバイス自動検出後の画面

「Change File...」ボタンをクリックするとファイル選択画面が表示されるので，「output_files\pio.sof」（プロジェクトのフォルダを起点とした相対パス）を選択します．sof ファイルの選択後，FPGA デバイス 5CSEMA5F31 が正しく認識されます．これを確認して「Program/Configure」をチェックします．チェック後，「Start」をクリックしてハードウェアデザインを FPGA にプログラムします（図 2.50）．

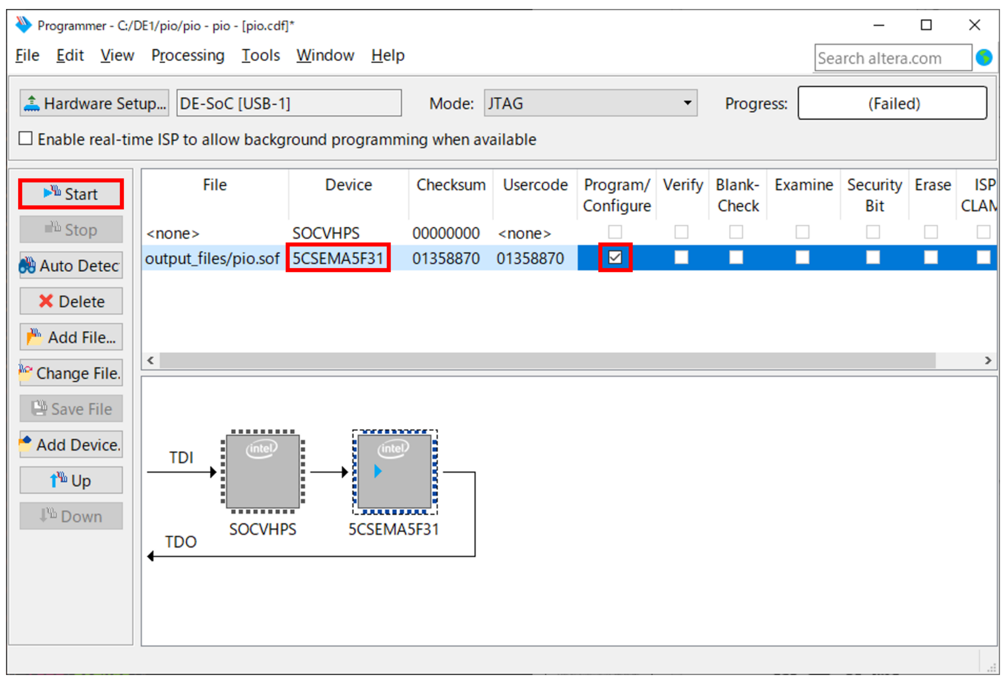


図 2.50　sof ファイル選択後の画面

正常にプログラムできると，「Progress」欄に「Successful」（成功）と表示されます（図 2.51）．「Failed」

（失敗）となる場合は，手順を確認して再度実行してください．

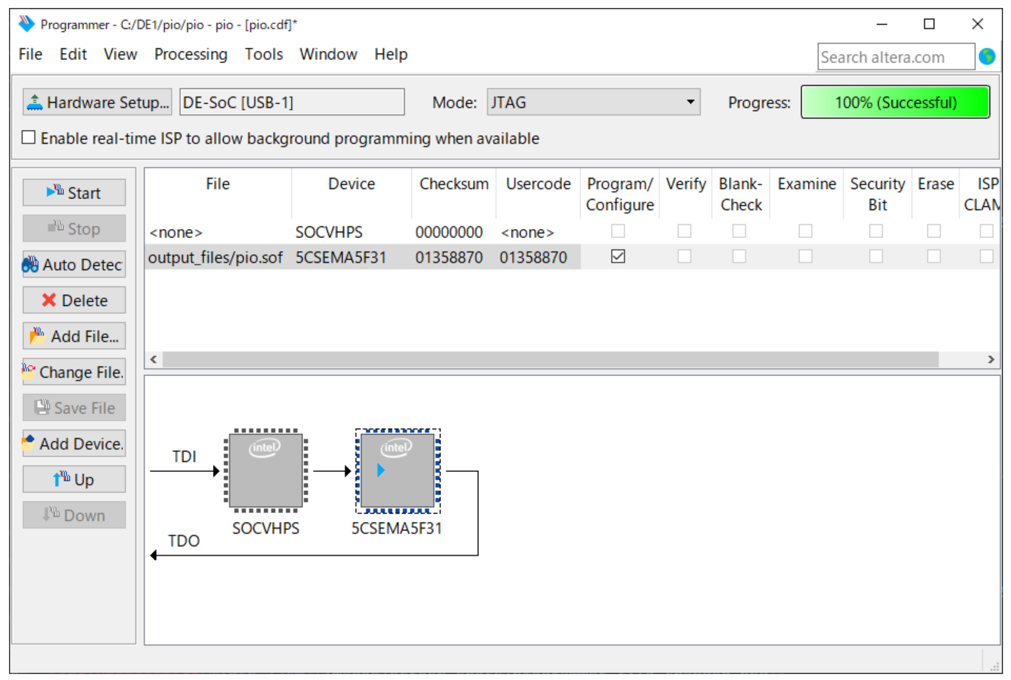


図 2.51　プログラミング成功時の画面

以上の手順で，ハードウェアの構築および FPGA へのプログラムが完了しました．続いて，構築したハードウェア上で実行する組込みソフトウェアを開発していきます．

2.3　組込みソフトウェアの開発

この節では，前節で構築したハードウェア上で動作する組込みソフトウェアの開発手順を示します．図 1.2（p. 3）のように，組込みソフトウェアの開発では **Nios II SBT** を使用します．標準出力に 1 行出力するだけの簡単な「Hello World」アプリケーションの作成を通して，プロジェクトの作り方，プログラムの実行やデバッグの方法を学びます．

2.3.1　Nios II SBT の起動

Platform Designer と同様に，Nios II SBT も Quartus Prime のメニューから起動できます．「Tools」→「Nios II Software Build Tools for Eclipse」（図 2.52）を選択すると，Nios II SBT（Eclipse ベースの統合開発環境）が起動します．

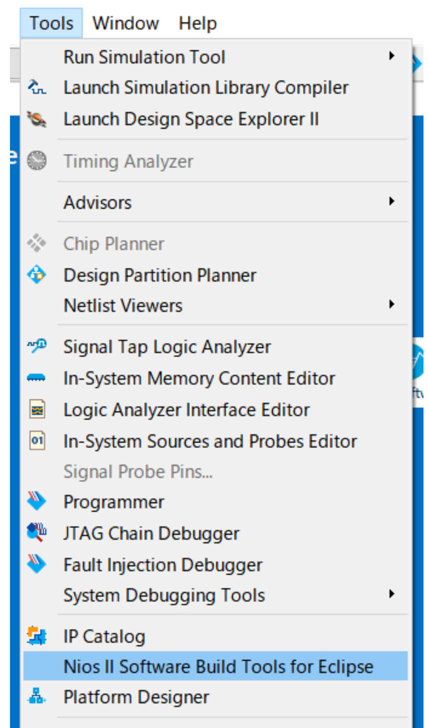


図 2.52　Nios II Software Build Tools for Eclipse の選択

起動時には，ワークスペース（作業フォルダ）をどこにするか質問されます（図 2.53）*2．ワークスペースは，ハードウェア構成を大きく変更するたびに新しく作るようにします．ここではハードウェアデザインのフォルダ `D:\DE1\pio` を指定して「OK」をクリックします．

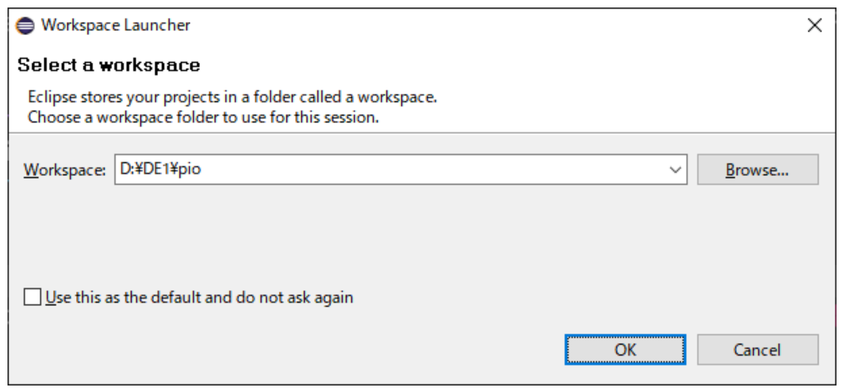


図 2.53　ワークスペースの設定

2.3.2　プロジェクトの作成

テンプレートを使用して組込みソフトウェアのプロジェクトを作成します．メニューから「File」→「New」→「Nios II Application and BSP from Template」を選択します（図 2.54）．

*2 Eclipse の設定が，指定したフォルダ以下に保存されます．なお，起動時のワークスペース設定後にワークスペースを変更したい場合は，メニューから「File」→「Switch Workspace」→「Other...」を選択します．

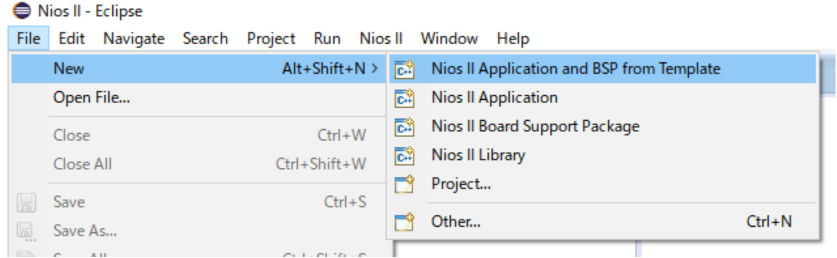


図 2.54　Nios II Application and BSP from Template の選択

メニュー項目を選択すると，「Nios II Application and BSP from Template」ウィンドウ（図 2.55）が表示されます．表 2.5 のように設定します．

表 2.5　プロジェクト作成時の設定

項目	値
SOPC Information File name	`D:\DE1\pio\nios2_pio.sopcinfo`
Project name	`uart1`
Project template	Hello World Small

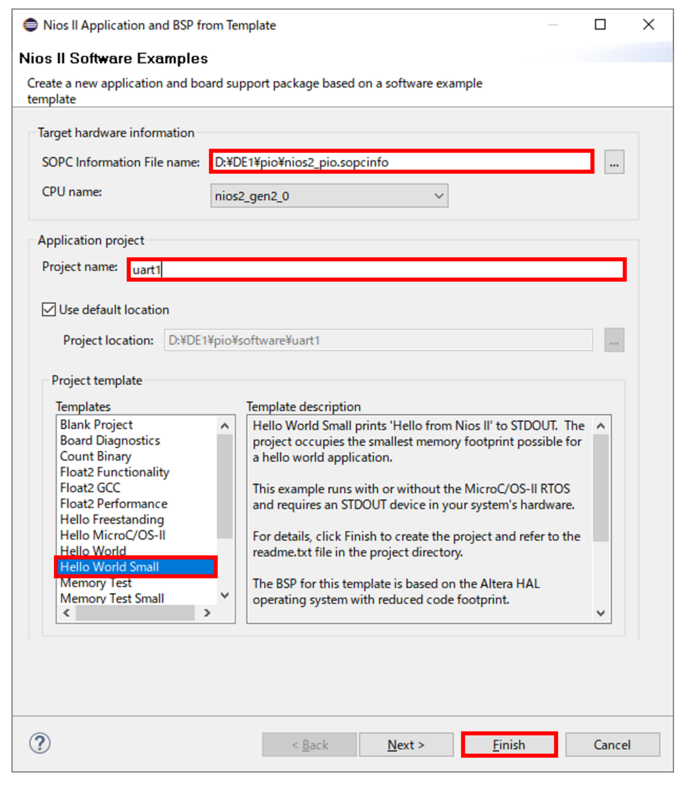


図 2.55　Nios II Application and BSP from Template ウィンドウ

プロジェクト作成時に指定する `*.sopcinfo` ファイルは，マイコンシステムに関する情報をまとめたファイルです．Platform Designer で HDL を生成する際に生成されます．このファイルを基にして，組込みソフトウェア開発で用いられるヘッダファイルやライブラリが生成されます．

プロジェクトテンプレートとして「Hello World Small」を選択するのは，小さなメモリ空間にプログラムを格納する必要があるためです．このテンプレートを使用すると，簡易仕様の関数を集めたライブ

ラリが用意され，プログラムが小さくなります．

プロジェクトを作成すると，左側の「Project Explorer」ペインに以下の 2 つのプロジェクトが表示されます．

- **uart1**：アプリケーションプロジェクト
- **uart1_bsp**：アプリケーションプロジェクトで使用するライブラリのプロジェクト

「Project Explorer」ペイン内の uart1 をダブルクリックすると，プロジェクトを構成するファイルの一覧が表示されます．その中から hello_world_small.c をダブルクリックして開くと，main 関数を含む C 言語ソースコードが表示されます（図 2.56）．このプログラムは，以下のように非常に単純な処理を行います．

1. alt_putstr 関数を用いて，標準出力（JTAG UART 送信）に "Hello from Nios II!\n" と出力する．
2. 何もしない無限ループ．

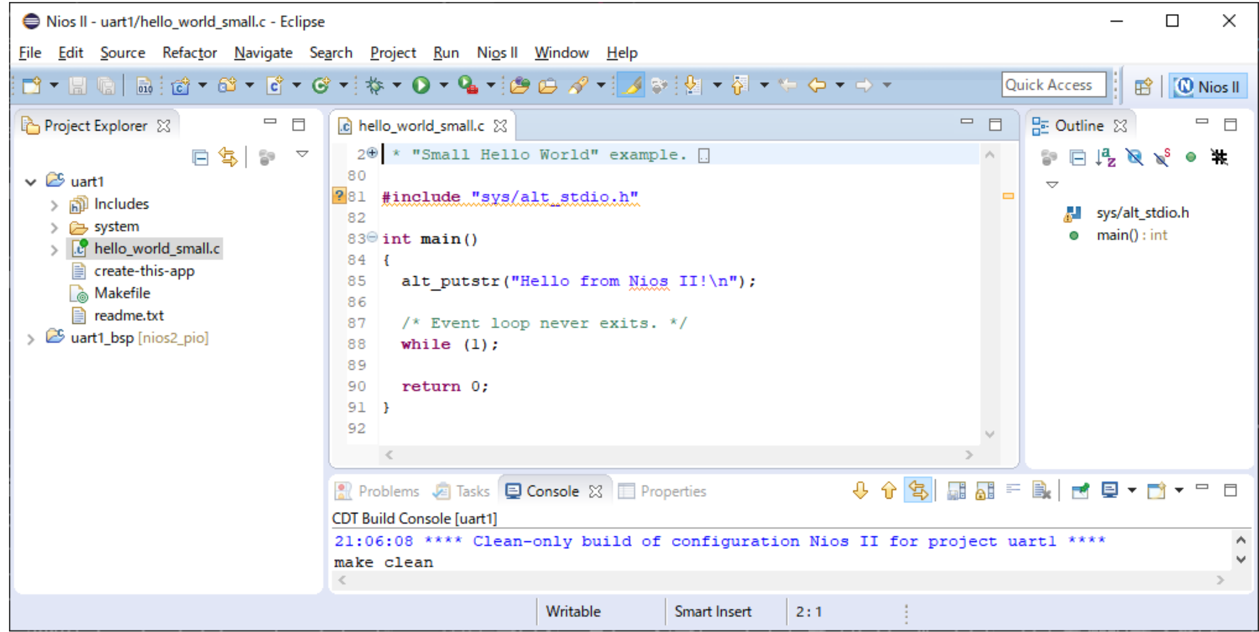


図 2.56　hello_world_small.c

2.3.3　プロジェクトのビルドとプログラムの実行

このままプロジェクトをビルドしてみましょう．「Project Explorer」ペインにおいて，アプリケーションプロジェクト uart1 の上で右クリックし，表示されるメニューから「Build Project」を選択すると，ビルドが開始されます（図 2.57）．

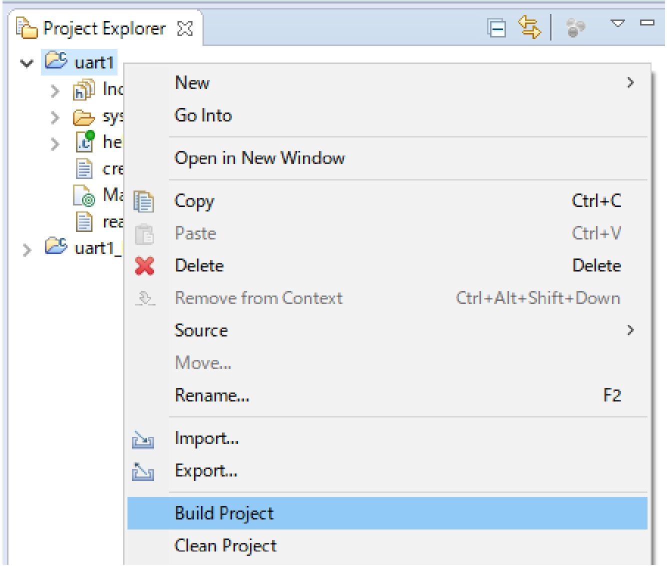


図 2.57　Build Project

図 2.58 のように，下部の「Console」タブに「build complete」というメッセージが表示されれば，ビルド成功です．

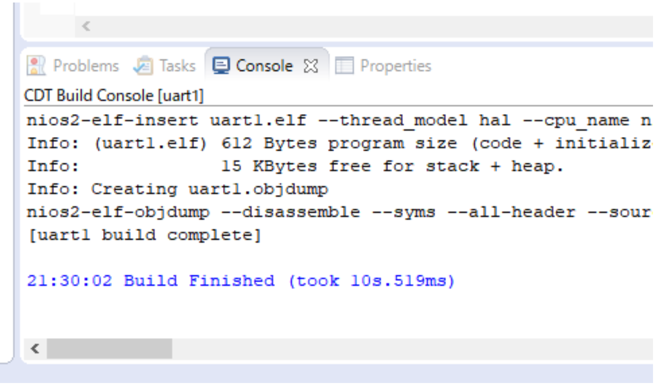


図 2.58　ビルド成功時のメッセージ

ビルドが成功したら，DE1-SoC 上でプログラムを実行してみます．「Project Explorer」ペインの `uart1` の上で右クリックし，表示されるメニューから「Run As」→「3 Nios II Hardware」を選択します（図 2.59）．

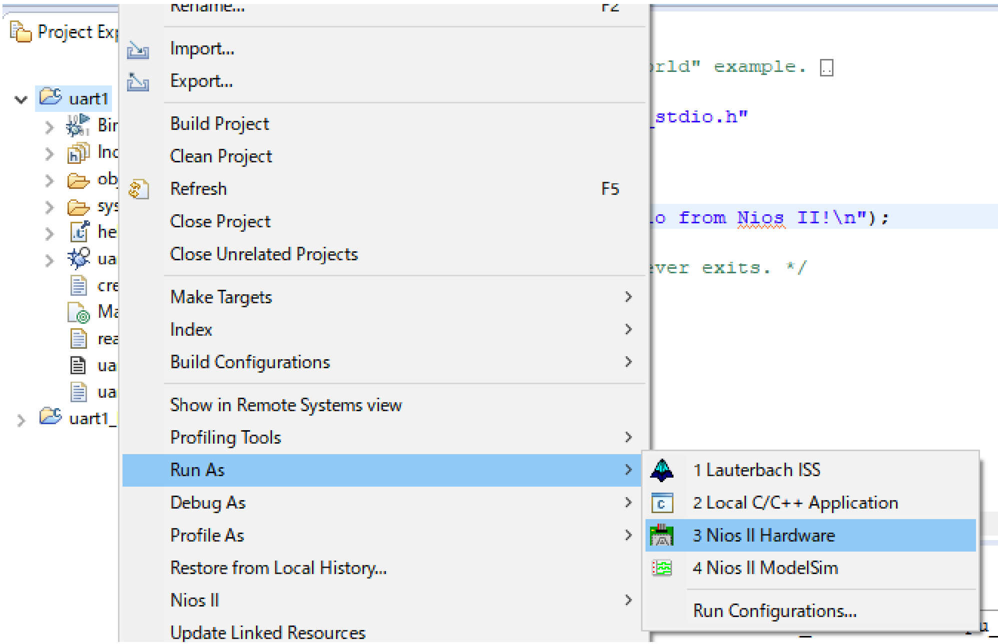


図 2.59　Run As Nios II Hardware

プログラムがすぐに実行されず，「Run Configurations」画面（図 2.60）が表示された場合は，システム ID チェックを無効化する必要があります．「Target Connection」タブの「System ID checks」欄にある 2 項目（Ignore mismatched system ID，Ignore mismatched system timestamp）にチェックします．また，「Processors」欄に何も表示されていない場合は，電源が入った DE1-SoC が PC に接続されているのを確認してから「Refresh Connections」ボタンをクリックして，FPGA デバイスを認識させます．その後，「Apply」ボタン，「Run」ボタンを順にクリックすると，プログラムが実行されます．

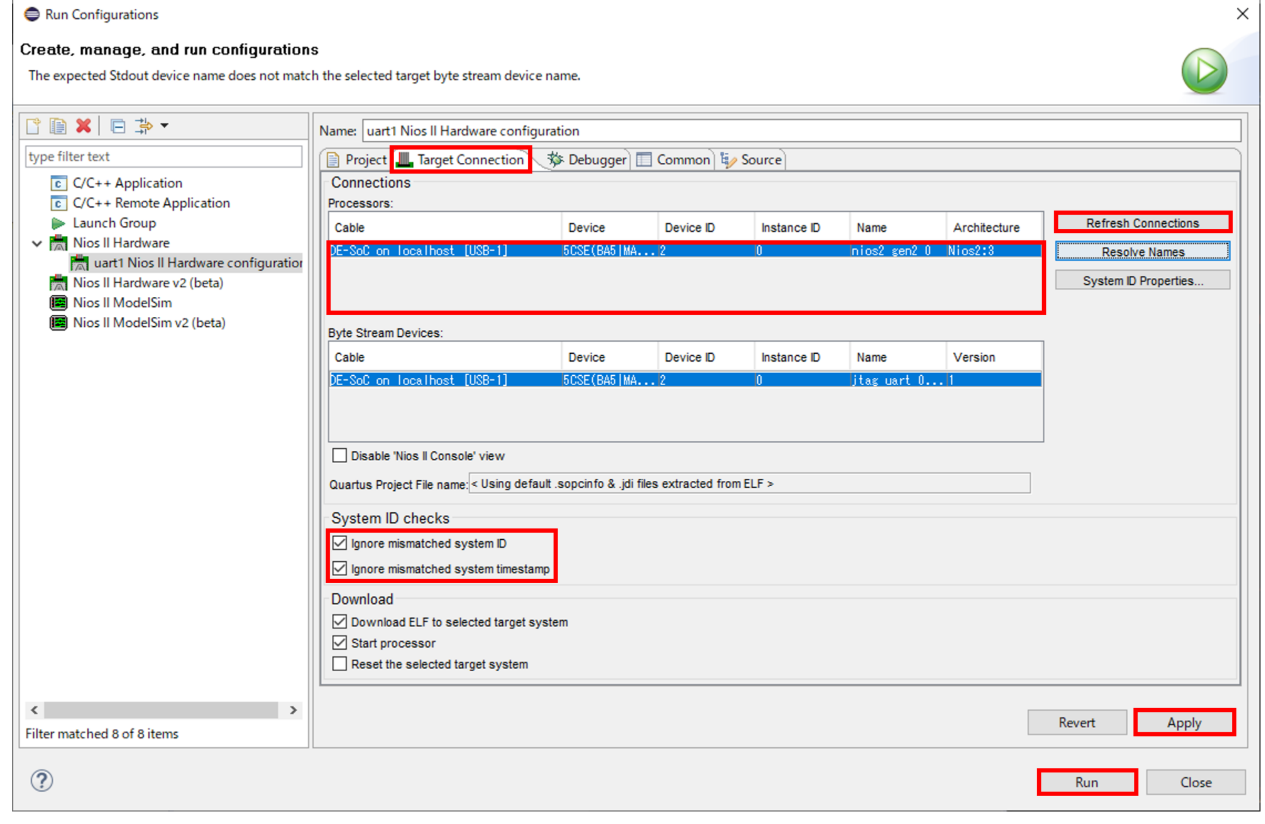


図 2.60　Run Configurations 画面

プログラムが正常に実行されると，下部が「Nios II Console」タブに切り替わり，図 2.61 のように表示されます．メッセージの表示後は，メインループ（無限ループ）内の処理が繰り返されています．プログラムを停止させるには，右側にある赤い四角形の実行停止ボタンをクリックします．

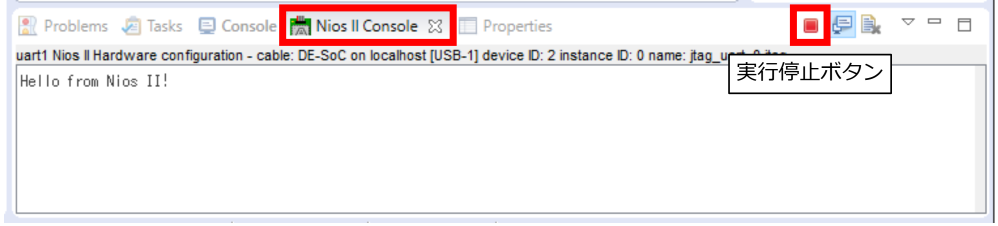


図 2.61　「`Hello from Nios II!`」が表示された Nios II Console

2.3.4　alt 標準入出力関数によるデバッグ

JTAG UART をマイコンシステムに加えると，`alt` 標準入出力関数が使えるようになります．この関数を使用して「`printf` デバッグ」が行えます．

`alt` 標準入出力関数を使用するには，`sys/alt_stdio.h` をインクルードします．右側「Outline」ペイン（図 2.62）の「`sys/alt_stdio.h`」をダブルクリックすると，その内容を確認できます．

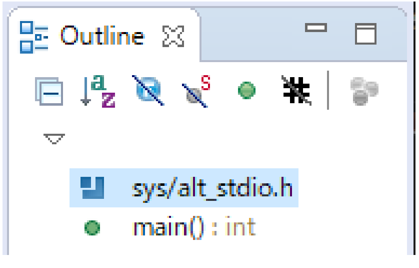


図 2.62　Outline ペイン

`sys/alt_stdio.h` では，標準入出力関数に似た 4 つの関数が定義されています（図 2.63）．これらのソースファイルは，ライブラリプロジェクト `uart1_bsp` の `HAL\inc` フォルダおよび `HAL\src` フォルダに格納されています（図 2.64）．

hello_world_small.c　alt_stdio.h

```
50 extern "C" {
51 #endif
52
53 int alt_getchar();
54 int alt_putchar(int c);
55 int alt_putstr(const char* str);
56 void alt_printf(const char *fmt, ...);
57 #ifdef ALT_SEMIHOSTING
```

Outline

- _ALT_STDIO_H_
- stdarg.h
- alt_getchar() : int
- alt_putchar(int) : int
- alt_putstr(const char*) : int
- alt_printf(const char*, ...) : void
- alt_putcharbuf(int) : int

図 2.63　`sys/alt_stdio.h`

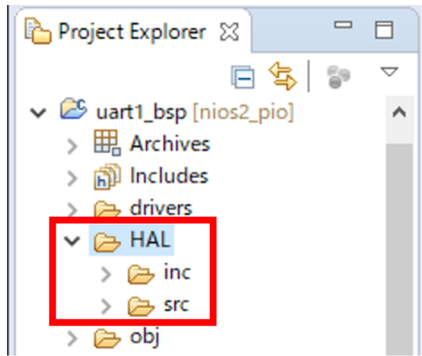


図 2.64　ライブラリのソースファイルの格納先

出力関数

出力関数として使用頻度が高いのは，`alt_putstr()` 関数と `alt_printf()` 関数です．

■ alt_putstr()

標準出力に文字列を出力します．

プロトタイプ `int alt_putstr(const char* str);`

引数

- `str`：出力する文字列

戻り値 正常に実行できれば非負の整数．エラーが発生すれば`-1`.

■ alt_printf()

指定された書式に変換して出力します．

プロトタイプ `void alt_printf(const char *fmt, ...);`

引数

- `fmt`：フォーマット文字列．簡易版のため，`%c`，`%s`，`%x` のみ使用可能．
- `...`：各変換指定と対応する値

戻り値 なし

入力関数

入力関数については，Nios II Console から入力された 1 文字を返す `alt_getchar()` 関数しか用意されていません．しかし，改行文字が文字コード `0x0A`（LF，`'\n'`）として取り込まれることを利用すると，リスト 2.2 のように，入力された 1 行分の文字列を返す関数 `my_getstr()` を作ることができます．

リスト 2.2　入力された 1 行分の文字列を返す関数

```
#include <stddef.h>
#include <sys/alt_stdio.h>

/**
 * @brief 標準入力から1行取り込む。
 * @param buf 格納先バッファ。
 * @param buf_size 格納先バッファの大きさ。
 * @return 取り込んだ文字列の先頭を示すポインタ。
 */
char* my_getstr(char* buf, size_t buf_size) {
  char c;
  size_t i;

  for (i = 0; i < buf_size - 1; i++) {
    // 1文字取り込む
    c = alt_getchar();
    // 改行コードならば取り込み終了
    if (c == '\n') {
      break;
    }

    buf[i] = c;
  }

  // 文字列を終端させる
  buf[i] = '\0';

  return buf;
}
```

課題 2.1 alt 標準入出力関数（uart2）

alt_printf() および my_getstr() を使用して，入力された名前に応じてあいさつを返すプログラムを作成してください．プログラムを作成する際は，2.3.2 項（p. 36）の手順で，課題名と同じ名前のプロジェクト uart2 を作成してください．

■ 入出力例

```
[uart2]
Name:
Muko Yutaka
-> [1] Hello, Muko Yutaka.

Name:
Amagasaki Taro
-> [2] Hello, Amagasaki Taro.

...

Name:
Habatan
-> [9] Hello, Habatan.

Name:
Harotore-kun
-> [a] Hello, Harotore-kun.
```

2.3.5　Nios II Debug パースペクティブによるデバッグ

Nios II SBT には，Nios II Debug というデバッグ用のパースペクティブ（作業用の画面）が用意されています．この項では，Nios II Debug パースペクティブを利用したデバッグの方法を紹介します．

Nios II Debug パースペクティブへの移行

以下の手順で，Nios II Debug パースペクティブに移行します．左側の「Project Exproler」ペインにおいて，アプリケーションプロジェクト名の上で右クリックします．表示されたメニューから「Debug As」→「2 Nios II Hardware」を選択します（図 2.65）．

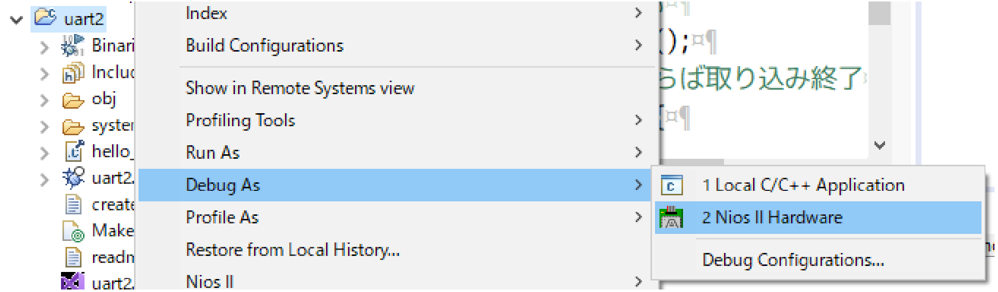


図 2.65　Nios II Debug パースペクティブへの移行

選択後，Nios II Debug パースペクティブに移行するか確認する画面（図 2.66）が表示されますので，「Yes」をクリックします．

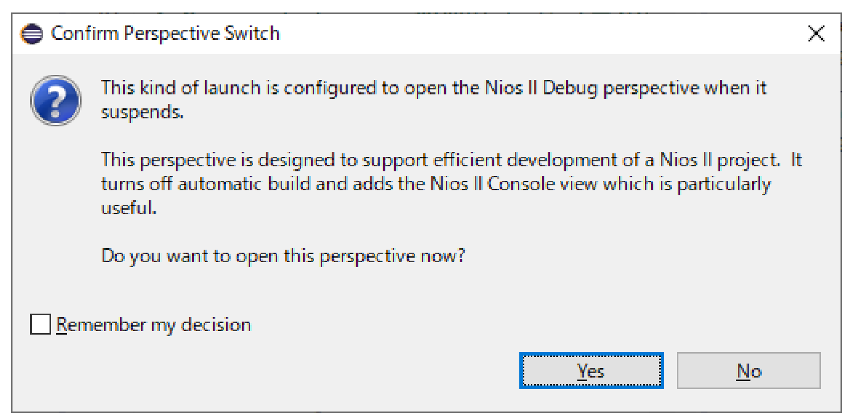


図 2.66　Nios II Debug パースペクティブへの移行確認

Nios II Debug パースペクティブに移行すると，画面が図 2.67 のように変わります．

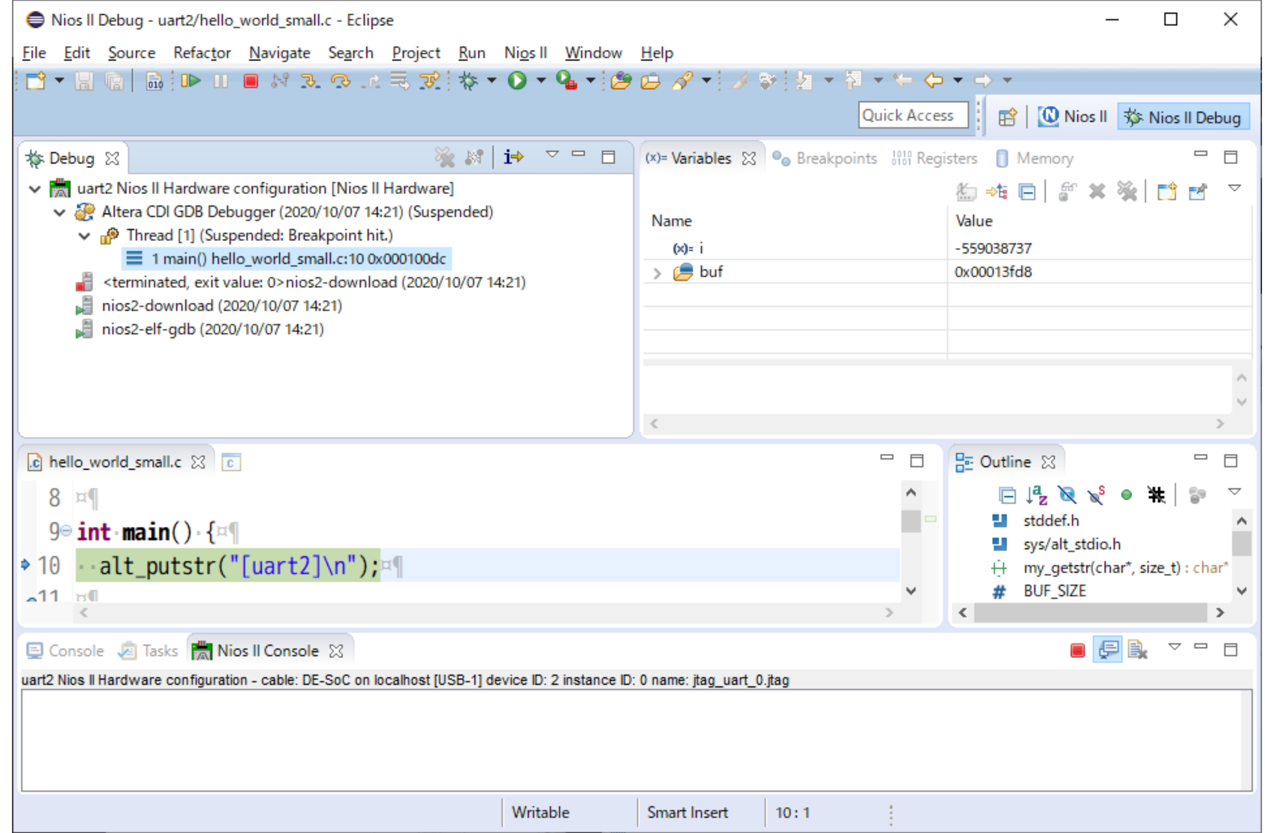


図 2.67　Nios II Debug パースペクティブ

元のパースペクティブに戻りたいときは，上部のツールバーから赤い四角形の「Terminate」ボタンをクリックしてプログラムを停止した後，右上の「Nios II」ボタンをクリックします（図 2.68）．

図 2.68　元のパースペクティブに戻るための手順

ブレークポイントの設定とプログラムの実行

ブレークポイント（デバッグ中にプログラムの実行を一時停止させる行）を設定するには，中央左側のソースコードペインにおいて，行番号をダブルクリックします（図 2.69）．

hello_world_small.c　alt_putstr.c

```
 7 #define BUF_SIZE 32
 8
 9 int main() {
10   alt_putstr("[uart2]\n");
```

① 行番号をダブルクリック

```
12   int i;
```

```
10   alt_putstr("[uart2]\n");
```

② ブレークポイントが設定される

図 2.69　ブレークポイントの設定

ブレークポイントまでプログラムを実行するには，上部のツールバーの「Resume」ボタンをクリックします（図 2.70）．

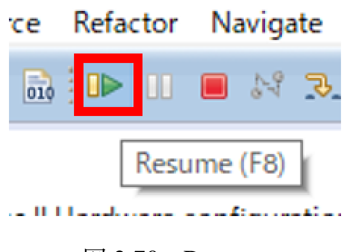


図 2.70　Resume

プログラムを最初から実行し直したい場合には，ツールバーの虫のボタンのプルダウンメニューから，アプリケーションプロジェクト名を選択します（図 2.71）．

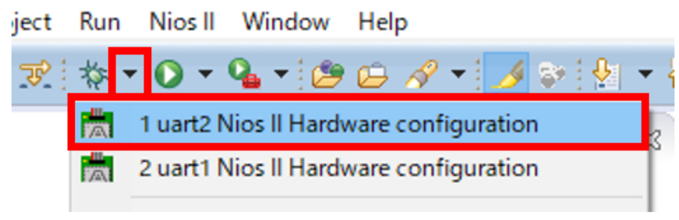


図 2.71　プログラムの再実行

プログラムのステップ実行

プログラムを 1 行ずつ実行するステップ実行は，上部のツールバー（図 2.72）から行えます．

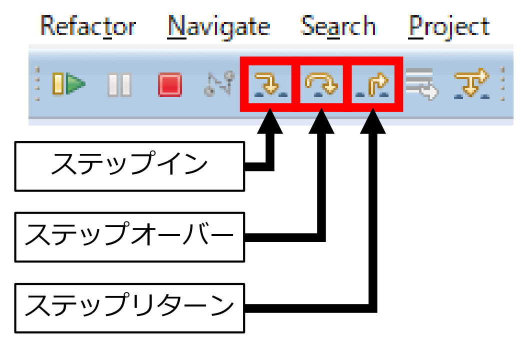


図 2.72　ステップ実行で使用するボタン

ステップ実行の各操作について，以下にまとめます [7].

- **ステップイン**（ショートカット F5）：現在の行から呼び出された最初の関数の最初の文で停止する．
- **ステップオーバー**（ショートカット F6）：現在の文とそれが呼び出すすべての関数を実行し，次の文の前に停止する（または，途中の最初のブレークポイント）．
- **ステップリターン**[*3]（ショートカット F7）：(1) 現在の関数を最後まで実行し，(2) 呼び出し元での関数呼び出しの次の文の前で停止する（または，途中の最初のブレークポイント）．

最適化の抑止

プログラムをステップ実行したとき，不自然な前後の動きが見られる場合は，コンパイラによる最適化が原因かもしれません．「Hello World Small」テンプレートを使用した場合，既定の設定では，コンパイラはプログラムが小さくなるように最適化を行います．最適化が行われると，プログラムが省略されたり並び替えられたりして，ソースコードとは異なる順序で実行される可能性があります．

プログラムが必ずソースコードのとおりの順序で進むようにするには，最適化を行わない設定にします．設定は，プログラムを停止してから，Nios II 開発用の通常のパースペクティブに移行して行います．左側の「Project Exproler」ペインにおいて，アプリケーションプロジェクト名の上で右クリックし，表示されるメニューの最下部にある「Properties」を選択します（図 2.73）．

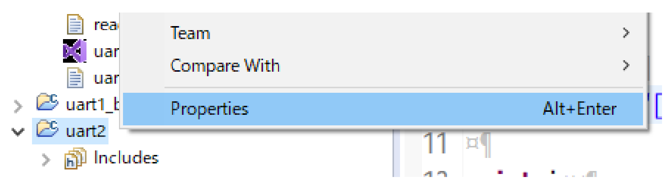


図 2.73　Properties

続いて表示されるプロジェクトの設定画面において，左側の「Nios II Application Properties」を選択します．そして，「Optimization level」を「Off」に設定した後，「OK」をクリックして設定画面を閉じます（図 2.74）．

[*3] **ステップアウト**ともいいます．

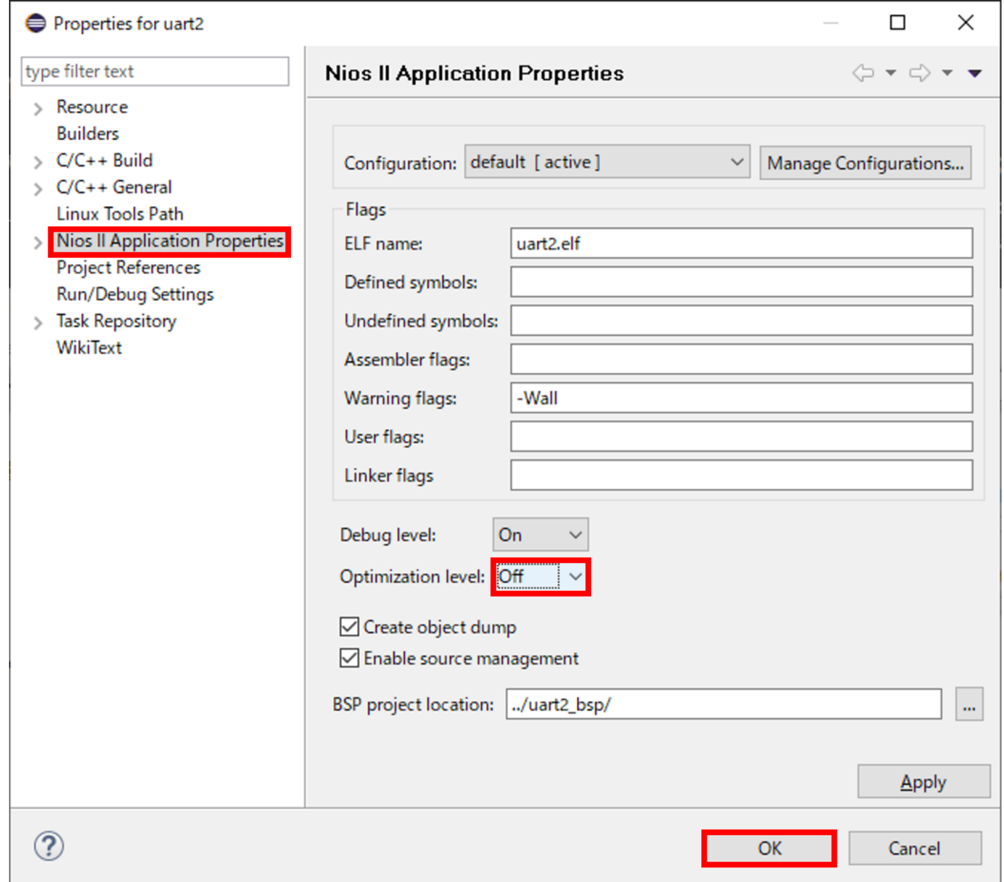


図 2.74　最適化を抑止する設定

設定後，再度 Nios II Debug パースペクティブに移行すると，最適化しない設定でビルドされたプログラムが実行されます．再びステップ実行してみて，プログラムが正しい順序で実行されることを確認してください．

マイコンの状態の確認

Nios II Debug パースペクティブの右上のペインでは，マイコンの現在の状態を確認できます．ペイン内のタブを切り替えることで，変数の内容（図 2.75），ブレークポイントの一覧（図 2.76），レジスタの内容（図 2.77），メモリの内容（図 2.78）を見ることができます．

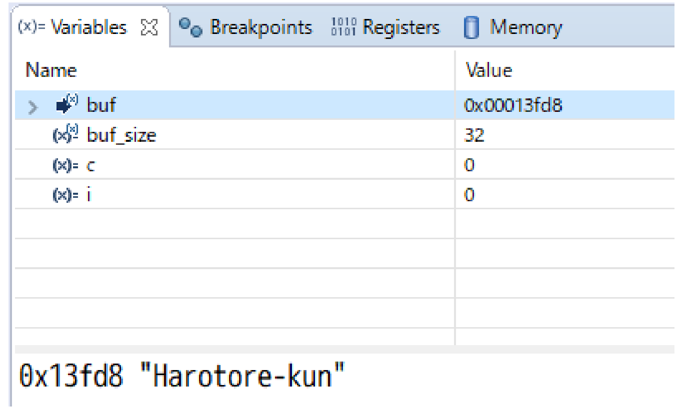


図 2.75　変数の内容の確認

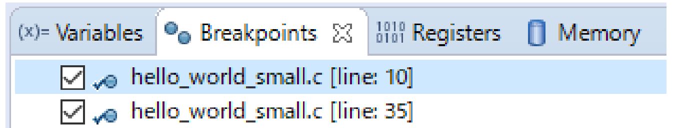


図 2.76　ブレークポイントの一覧

Name	Value
Main	
zero	0
at	3735928559
r2	6
r3	32
r4	81880
r5	32
r6	66918

図 2.77　レジスタの内容の確認

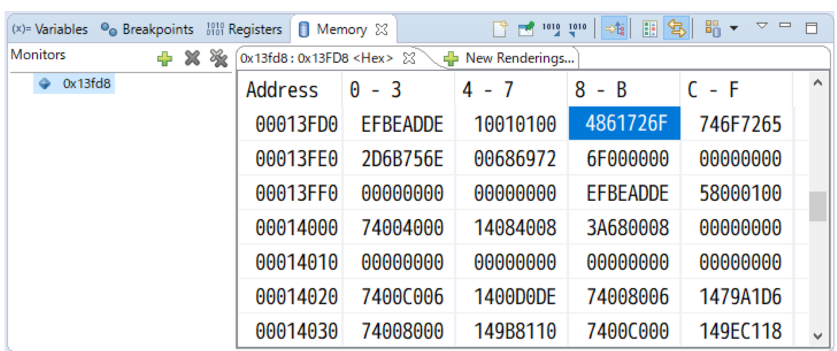


図 2.78　メモリの内容の確認

2.4　LED の制御

この節では，PIO を使用して，DE1-SoC に実装されている LED を制御する方法について学習します．

2.4.1　DE1-SoC の LED 回路

DE1-SoC に実装されている LED の回路図を図 2.79 に示します．回路は正論理であり，接続されているピンの電圧が H レベルであるときに LED が点灯します．

図 2.79　DE1-SoC の LED 回路 [6]

2.2 節では，10 ビットの出力用 PIO（先頭アドレス `0x30000`）を LED 制御レジスタとしてマイコンシステムに組み込み，各ビットに対して LED と接続されているピンを割り当てました．このレジスタのビットと LED との対応を図 2.80 に示します．ビットを 1 に設定すると対応する LED が点灯し，0 に設定すると消灯します．

ビット	7	6	5	4	3	2	1	0
LED	LEDR7	LEDR6	LEDR5	LEDR4	LEDR3	LEDR2	LEDR1	LEDR0
ビット	15	14	13	12	11	10	9	8
LED	未使用	未使用	未使用	未使用	未使用	未使用	LEDR9	LEDR8

図 2.80　LED 制御レジスタのビットと LED との対応

2.4.2　LED 制御プログラムの作成

例題 2.1　LED の点滅（led1）

LEDR0〜LEDR9 を 1 秒間隔で点滅させる（「500 ms 点灯，500 ms 消灯」を繰り返す）プログラムを作成してください．

Nios II SBT のメニューから「File」→「New」→「Nios II Application and BSP from Template」を選択し，表 2.6 に示す設定で例題 2.1 用のプロジェクトを作成します．

表 2.6　例題 2.1 用プロジェクト作成時の設定

項目	値
SOPC Information File name	`D:\DE1\pio\nios2_pio.sopcinfo`
Project name	`led1`
Project template	Hello World Small

プロジェクトを作成したら，リスト 2.3 に示す C 言語プログラムを `hello_world_small.c` に入力し

ます．入力後，ビルド，実行すると，DE1-SoC の LEDR0〜LEDR9 が 1 秒間隔で点滅します．

リスト 2.3　例題 2.1 の C 言語プログラム

```
#include <unistd.h>

#include "sys/alt_stdio.h"
#include "altera_avalon_pio_regs.h"

#include "system.h"

void wait_ms(int ms);

int main() {
  alt_putstr("[led1]\n");

  while (1) {
    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0x3FF);
    wait_ms(500);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0x000);
    wait_ms(500);
  }

  return 0;
}

/**
 * @brief 指定された時間だけ待機する。
 * @param ms 待機時間（ms単位）。
 */
void wait_ms(int ms) {
  int i;
  for (i = 0; i < ms; i++) {
    usleep(1000);
  }
}
```

レジスタへの書き込み

PIO で作成したレジスタへの書き込みには，ヘッダファイル altera_avalon_pio_regs.h で定義されている IOWR_ALTERA_AVALON_PIO_DATA() マクロの使用が推奨されています [8]．このマクロを使用すると，ビット幅を考慮した複雑な型キャストが不要になるため，可読性の向上や記述ミスの防止に

つながります[*4].

マクロ定義 #define IOWR_ALTERA_AVALON_PIO_DATA(base, data) ...

必要なヘッダファイル altera_avalon_pio_regs.h

引数

- base：PIO の先頭アドレス
- data：書き込むデータ

例題 2.1 において，LEDR0〜LEDR9 をすべて点灯させるためには，LED 回路が正論理であることから，LED 制御レジスタの全ビットを 1 にします．すなわち，LED 制御レジスタに対して $(11\ 1111\ 1111)_2$ = 0x3FF を書き込みます．

アドレスの指定

マイコンシステムの構成設定時に設定した各 IP の先頭アドレスは，ヘッダファイル system.h の中にマクロとして定義されており，名前で指定することができます．マクロ名は，大文字に変換された IP コアの名前の末尾に「_BASE」を加えたものになります．LED 用 PIO の場合，IP コア名が led だったため，その先頭アドレスを示すマクロ名は LED_BASE となります．

このマクロを IOWR_ALTERA_AVALON_PIO_DATA() の第 1 引数に置くことで，どのレジスタを書き換えるかが明確になり，ソースコードの可読性が向上します．今後の課題で別の出力装置の制御を行う際にも，この書き方を積極的に利用してください．

指定した時間の待機

LED を一定時間点灯・消灯させるのに必要な待機には，ヘッダファイル unistd.h で定義されている usleep() 関数を利用できます．usleep() 関数を実行すると，指定された時間（μs 単位）だけ待機します．ただし，例題 2.1 の要件では待機時間の数値が非常に大きくなるため，ms 単位で待機時間を指定できる wait_ms() 関数を作成して使用しています．

[*4] 複雑な型キャストとは，例えば (volatile unsigned long *) のようなもの．レジスタの大きさやデータバス幅が異なる別のマイコンにプログラムを移植する場合は，この型キャストを全箇所書き換える必要が生じる可能性もある．その際には記述ミスが起こりやすい．

課題 2.2　LED の制御（led2）

手順に従って，下の図のように LED を点灯させるプログラムを作成してください．

(1) 下の図の空欄を埋めて，LED の点灯パターンと対応する 16 進数の値を求めてください．
(2) (1) で求めた値を利用して，制御プログラムを作成してください．

番号	9	8	7	6	5	4	3	2	1	0
LED		■		■		■		■		■
ビット										1

16 進数　0x ______

200 ms 経過 ↓　↑ 200 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED	■		■		■		■		■	
ビット										0

16 進数　0x ______

課題 2.3　LED の制御（led3）

下の図のように LED を点灯させるプログラムを作成してください．

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

⋮

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

⋮

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

番号	9	8	7	6	5	4	3	2	1	0
LED										

↓ 50 ms 経過

（最初に戻る）

■ **ヒント**　ビットシフト演算およびループを使うと，簡潔に書けます．

2.5　スライドスイッチからの入力

この節では，DE1-SoC に実装されているスライドスイッチからデータを入力する方法について学習します．

2.5.1　DE1-SoC のスライドスイッチ回路

DE1-SoC に実装されているスライドスイッチの接続図を図 2.81 に示します．FPGA への入力電圧は，スイッチが下側にあるときに L レベル，上側にあるときに H レベルとなります．

図 2.81　DE1-SoC のスライドスイッチの接続図 [6]

2.2 節で構成したスライドスイッチレジスタ（10 ビットの入力用 PIO）のビットとスイッチとの対応を図 2.82 に示します．スイッチが下側にあるときに対応するビットが 0 に，上側にあるときに対応するビットが 1 になります．

ビット	7	6	5	4	3	2	1	0
スイッチ	SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
ビット	15	14	13	12	11	10	9	8
スイッチ	未使用	未使用	未使用	未使用	未使用	未使用	SW9	SW8

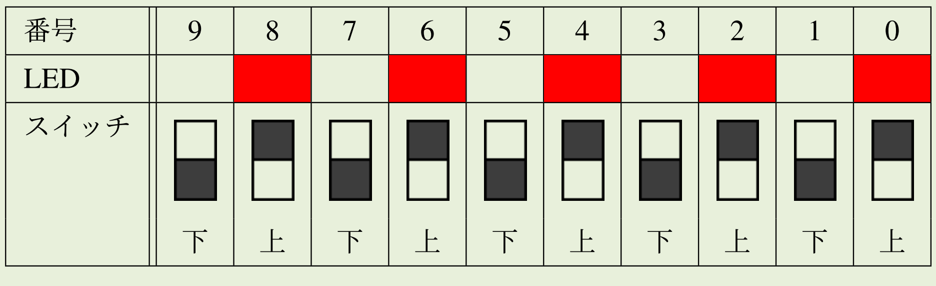
図 2.82　スライドスイッチレジスタのビットとスイッチとの対応

2.5.2　スライドスイッチ入力を使用するプログラムの作成

例題 2.2　スライドスイッチ入力：レジスタの値の読み込み（slide_sw1）

各スライドスイッチとその上にある LED について，スイッチを上げると LED を点灯させ，スイッチを下げると LED を消灯させるプログラムを作成してください．

■ 動作例



番号	9	8	7	6	5	4	3	2	1	0
LED										
スイッチ	下	上	下	上	下	上	下	上	下	上

Nios II SBT のメニューから「File」→「New」→「Nios II Application and BSP from Template」を選択し，表 2.7 に示す設定で例題 2.2 用のプロジェクトを作成します．

表 2.7　例題 2.2 用プロジェクト作成時の設定

項目	値
SOPC Information File name	`D:\DE1\pio\nios2_pio.sopcinfo`
Project name	`slide_sw1`
Project template	Hello World Small

続いてリスト 2.4 に示す C 言語プログラムを `hello_world_small.c` に入力します．入力後，ビルド，実行して動作を確認してください．

リスト 2.4 例題 2.2 の C 言語プログラム

```
#include <stdint.h>

#include "sys/alt_stdio.h"
#include "altera_avalon_pio_regs.h"

#include "system.h"

int main() {
  alt_putstr("[slide_sw1]\n");

  uint16_t pattern;
  while (1) {
    pattern = IORD_ALTERA_AVALON_PIO_DATA(SLIDE_SW_BASE);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, pattern);
  }

  return 0;
}
```

レジスタの値の読み込み

PIO で作成したレジスタの値の読み込みには，ヘッダファイル `altera_avalon_pio_regs.h` で定義されている `IORD_ALTERA_AVALON_PIO_DATA()` マクロを使用します．`IOWR_ALTERA_AVALON_PIO_DATA()` と同様に，このマクロを使用すると，可読性の向上や記述ミスの防止につながります．

マクロ定義 `#define IORD_ALTERA_AVALON_PIO_DATA(base) ...`

必要なヘッダファイル `altera_avalon_pio_regs.h`

引数

- base：PIO の先頭アドレス

戻り値 指定されたアドレスの値（32 ビット幅）

例題 2.2 では，`IORD_ALTERA_AVALON_PIO_DATA()` の戻り値を `uint16_t` 型の変数 `pattern` で受け取っています．`uint16_t` は，ヘッダファイル `stdint.h` で定義される 16 ビット符号なし整数の型であり，マイコンごとに異なる `int` 型等のビット幅に影響されずに，値のビット幅を明示できます．`uint16_t` の他には，`int8_t`，`uint8_t`，`int16_t`，`int32_t`，`uint32_t` 等の型が用意されています．今回はスイッチの状態をそのまま LED の点灯パターンに反映させればよいので，`pattern` を加工せず `IORD_ALTERA_AVALON_PIO_DATA()` に渡しています．

例題 2.3　スライドスイッチ入力：スイッチの状態の判定と分岐（slide_sw2）

SW0 を上げると LEDR0〜LEDR9 を点灯させ，SW0 を下げると LEDR0〜LEDR9 を消灯させるプログラムを作成してください．

■ 動作例

番号	9	8	7	6	5	4	3	2	1	0
LED										
スイッチ	下	下	下	下	下	下	下	下	下	下

番号	9	8	7	6	5	4	3	2	1	0
LED										
スイッチ	下	下	下	下	下	下	下	下	下	上

リスト 2.5　例題 2.3 の C 言語プログラム

```
1  #include <stdint.h>
2
3  #include "sys/alt_stdio.h"
4  #include "altera_avalon_pio_regs.h"
5  #include "system.h"
6
7  int main() {
8    alt_putstr("[slide_sw2]\n");
9
10   uint16_t slide_sw;
11   uint16_t led_pattern;
12   while (1) {
13     slide_sw = IORD_ALTERA_AVALON_PIO_DATA(SLIDE_SW_BASE);
14     if (slide_sw & 0x001) {
15       led_pattern = 0x3FF;
16     } else {
17       led_pattern = 0x000;
18     }
19
20     IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, led_pattern);
21   }
22
```

```
23    return 0;
24  }
```

スイッチの状態の判定と分岐

スイッチの状態を判定して処理を分岐させる場合は，条件式としてレジスタから読み取った値にビット演算 &（AND）を行った結果を用います．今回は SW0 の状態を判定したいので，10 ビットのうち 0 ビット目のみ 1 とした `0x001` をマスクビットとして用います．SW0 が上側になっているとき，`slide_sw & 0x001` は 0 より大きい値となり，真と判定されます．一方で SW0 が下側になっているときは，`slide_sw & 0x001` は 0 となり，偽と判定されます．以上の判定の例を図 2.83 に示します．

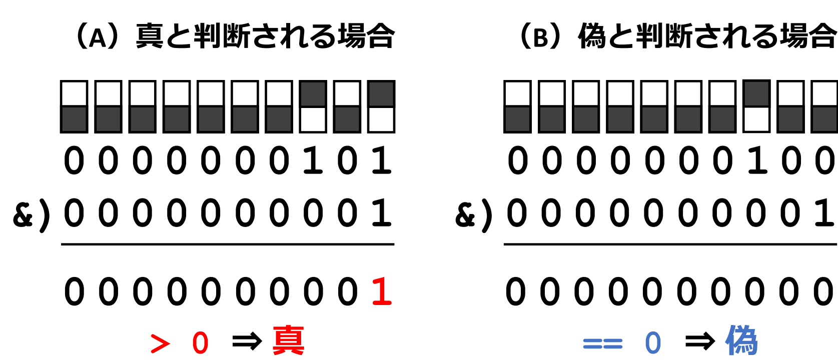


図 2.83　スイッチの状態の判定例

課題 2.4　スライドスイッチ入力：論理の反転（slide_sw3）

例題 2.2 とは逆に，スイッチを下げるとその上の LED を点灯させ，スイッチを上げるとその上の LED を消灯させるプログラムを作成してください．

■ **ヒント**　変数 `pattern` に代入する値のみ変更すると，最小限の変更で作成できます．

課題 2.5　スライドスイッチ入力：複数のスイッチの状態の判定（slide_sw4）

手順に従って，以下の仕様を満たすプログラムを作成してください．

■ 仕様

- SW0 を上げると，LEDR0〜LEDR4 を点灯させる．SW0 を下げると，LEDR0〜LEDR4 を消灯させる．
- SW9 を上げると，LEDR5〜LEDR9 を点灯させる．SW9 を下げると，LEDR5〜LEDR9 を消灯させる．
- 以上の 2 つを同時に実行できる．

■ 手順

(1) SW0，SW9 が上がっているか判定するためのマスクビットを，それぞれ 2 進数と 16 進数で表してください．
(2) LEDR0〜LEDR4，LEDR5〜LEDR9 を点灯させるために，それぞれ LED 制御レジスタに代入する値を 16 進数で表してください．
(3) (1)，(2) で求めた値を利用して，制御プログラムを作成してください．

(1) スイッチが上がっているか判定するためのマスクビット

■ SW0

2 進数 ________________　16 進数 `0x`________

■ SW9

2 進数 ________________　16 進数 `0x`________

(2) LED 制御レジスタに代入する値

■ LEDR0〜LEDR4

番号	9	8	7	6	5	4	3	2	1	0
LED										
ビット										

16 進数 `0x`________

■ LEDR5〜LEDR9

番号	9	8	7	6	5	4	3	2	1	0
LED										
ビット										

16 進数 `0x`________

2.6　ボタンスイッチからの入力

この節では，DE1-SoC に実装されているボタンスイッチからデータを入力する方法について学習します．

2.6.1　DE1-SoC のボタンスイッチ回路

DE1-SoC に実装されているボタンスイッチの回路図を図 2.84 に示します．ボタンスイッチにはシュミットトリガ回路が接続されており，図 2.85 のようにチャタリング除去された負論理の信号が，FPGA に入力されます．

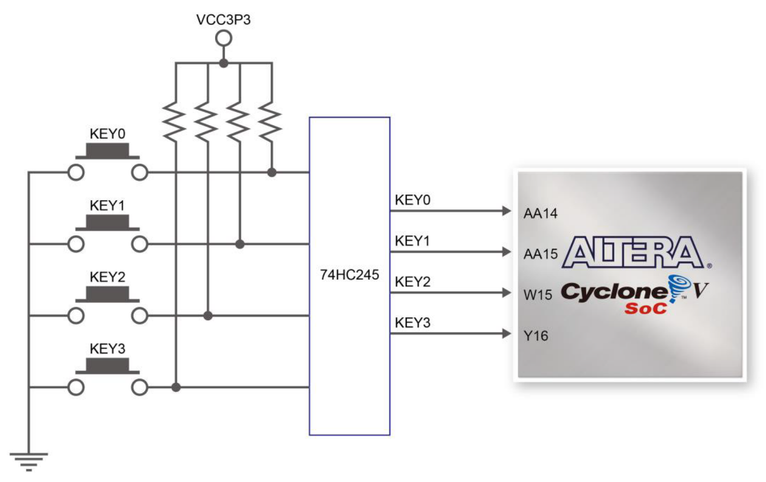


図 2.84　DE1-SoC のボタンスイッチ回路 [6]

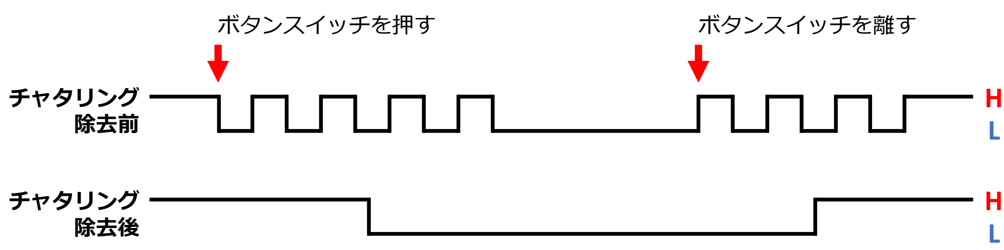


図 2.85　ボタンスイッチのチャタリング除去前後の電圧

2.2 節では，ボタンスイッチのうち右端の KEY0 をリセットスイッチとして使用し，残りの KEY1〜KEY3 を入力装置として使用できるように構成しました．また，KEY1〜KEY3 については，C 言語プログラムから扱いやすいように，論理を反転して正論理の信号が入力されるようにしています（p. 26 のリスト 2.1 を参照）．ボタンスイッチレジスタ（3 ビットの入力用 PIO）のビットとスイッチとの対応は，図 2.86 のとおりです．スイッチが押されていないときに対応するビットが 0 に，押されているときに対応するビットが 1 になります．

ビット	7	6	5	4	3	2	1	0
スイッチ	未使用	未使用	未使用	未使用	未使用	KEY3	KEY2	KEY1

図 2.86　ボタンスイッチレジスタのビットとスイッチとの対応

2.6.2　ボタンスイッチ入力を使用するプログラムの作成

例題 2.4　ボタンスイッチ入力：LED の点灯位置の制御（button_sw1）

次の仕様を満たすプログラムを作成してください．

- 初期状態では，LEDR0 を点灯させる．
- 右端の LED（LEDR0）が点灯していないとき，KEY1 を押すと点灯位置が 1 つ右に動く．
- 左端の LED（LEDR9）が点灯していないとき，KEY3 を押すと点灯位置が 1 つ左に動く．

リスト 2.6　例題 2.4 の C 言語プログラム

```
#include <stdint.h>
#include <unistd.h>

#include "sys/alt_stdio.h"
#include "altera_avalon_pio_regs.h"
#include "system.h"

// ボタンスイッチの数
#define N_BUTTON_SW 3

// ボタンスイッチの位置の型
typedef enum {
  KEY1, KEY2, KEY3
} ButtonSWPosition;

void LED_update(int pos);
void ButtonSW_update_pressed(uint8_t prev, uint8_t current, uint8_t* pressed);
void wait_ms(int ms);

int main() {
  alt_putstr("[button_sw1]\n");

  // LEDの点灯位置
  int pos = 0;
  // LEDの状態を初期化する
  LED_update(pos);

  // 前回調べたボタンスイッチレジスタの値
  uint8_t prev_button_sw = 0x0;
  // ボタンスイッチレジスタの値
  uint8_t button_sw;
  // ボタンスイッチ押下フラグの配列
  uint8_t button_sw_pressed[N_BUTTON_SW];

  while (1) {
    // ボタンスイッチレジスタの値を読み込む
    button_sw = IORD_ALTERA_AVALON_PIO_DATA(BUTTON_SW_BASE);
    // ボタンスイッチ押下フラグを更新する
```

```
39      ButtonSW_update_pressed(prev_button_sw, button_sw, button_sw_pressed);
40
41      if (button_sw_pressed[KEY1]) {
42        // KEY1が押されたとき
43        if (pos > 0) {
44          // 点灯位置が右端でなければ、点灯位置を1つ右に動かす
45          pos--;
46          LED_update(pos);
47        }
48      } else if (button_sw_pressed[KEY3]) {
49        // KEY3が押されたとき
50        if (pos < 9) {
51          // 点灯位置が左端でなければ、点灯位置を1つ左に動かす
52          pos++;
53          LED_update(pos);
54        }
55      }
56
57      // ボタンスイッチレジスタの値を記録する
58      prev_button_sw = button_sw;
59
60      wait_ms(10);
61    }
62
63    return 0;
64  }
65
66  /**
67   * @brief LEDの状態を更新する。
68   * @param pos 点灯させる位置。
69   */
70  void LED_update(int pos) {
71    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0x001 << pos);
72  }
73
74  /**
75   * @brief ボタンスイッチが押されたか調べ、押下フラグを更新する。
76   * @param prev 前回取得したボタンスイッチレジスタの値。
77   * @param current 今回取得したボタンスイッチレジスタの値。
78   * @param pressed ボタンスイッチ押下フラグの配列。
79   */
80  void ButtonSW_update_pressed(uint8_t prev, uint8_t current, uint8_t* pressed) {
81    int i;
82    for (i = 0; i < N_BUTTON_SW; i++) {
83      // マスクビット
84      uint8_t mask = 0x1 << i;
85
86      // 前回取得した値からi番目のビットを取り出す
87      uint8_t prev_masked = prev & mask;
88      // 今回取得した値からi番目のビットを取り出す
89      uint8_t masked = current & mask;
```

```
90
91      if ((masked != prev_masked) && (masked != 0x0)) {
92        // ビットが0→1に変化していたら（立ち上がりが発生したら）
93        // ボタンが押されたと判断する
94        pressed[i] = 1;
95      } else {
96        pressed[i] = 0;
97      }
98    }
99  }
100
101 /**
102  * @brief 指定された時間だけ待機する。
103  * @param ms 待機時間（ms単位）。
104  */
105 void wait_ms(int ms) {
106   int i;
107   for (i = 0; i < ms; i++) {
108     usleep(1000);
109   }
110 }
```

エッジ検出とチャタリング除去

ボタンスイッチを使用する際には，スイッチが押された瞬間に対応する処理を実行する場合が多くなります．この動作に必要なのが**エッジ検出**です．

例題 2.4 では，関数 `ButtonSW_update_pressed()` においてエッジ検出を行っています．この処理の概要を図 2.87 に示します．構成したハードウェアにおいては，ボタンスイッチから入力される信号が正論理であるため，スイッチが押された瞬間にボタンスイッチレジスタの対応するビットが `0` から 1 に変化します．この変化を検出するため，一定時間ごとにボタンスイッチレジスタの値を調べて，前回調べた値と比較します．ビットが `0` から 1 に変化した場合はボタンスイッチ押下フラグを 1（真）に設定し，そうでない場合は `0`（偽）に設定します．このようにボタンスイッチ押下フラグを設定したうえで，メインループにおいてフラグが真の場合を条件とする分岐を記述することで，スイッチが押された瞬間に対応する処理を実行できます．レジスタの値を調べる際の時間間隔は，チャタリングが収まるまでよりも十分に長く，かつ反応速度が遅すぎない程度（一般的には数 ms から数十 ms）とします．今回は 10 ms ごとにレジスタの値を調べるため，`main()` 関数にあるメインループの最後で `wait_ms(10)` を呼び出しています．

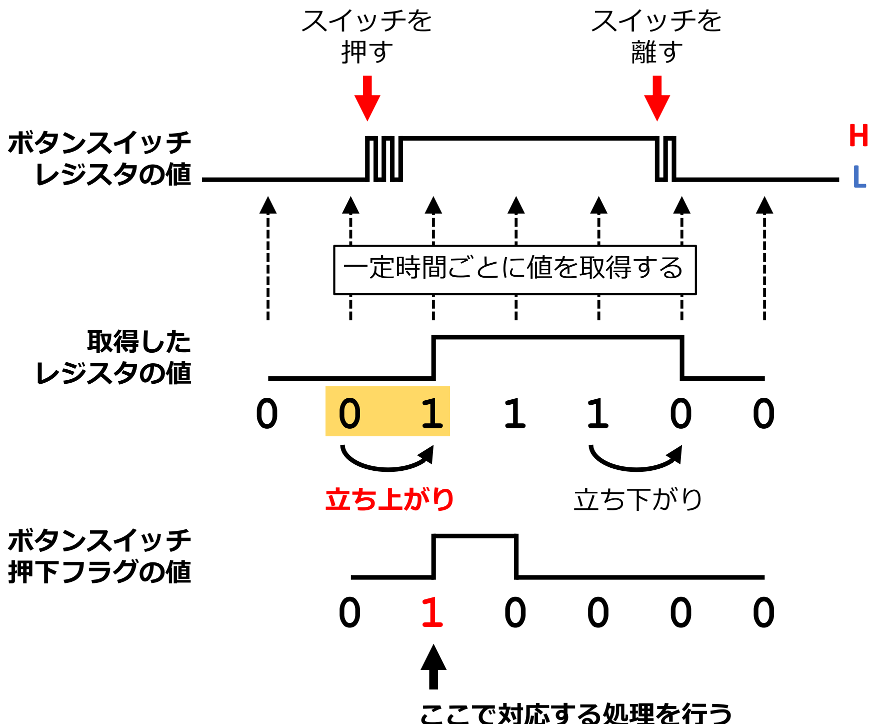


図 2.87　ボタンスイッチ入力のエッジ検出の概要

課題 2.6　レベルメータ（button_sw3）

次の仕様を満たすレベルメータのプログラムを作成してください．

- ボタンスイッチを使用して「レベル」を設定できる．
  - レベルは 1〜10 の 10 段階とする．初期状態ではレベル 1 とする．
  - レベルが 2 以上のとき，KEY1 を押すとレベルが 1 減少する．
  - レベルが 10 未満のとき，KEY3 を押すとレベルが 1 増加する．
- 下の図のように，現在のレベルを LEDR0〜LEDR9 に表示する．

■ レベル 1

番号	9	8	7	6	5	4	3	2	1	0
LED										

■ レベル 2

番号	9	8	7	6	5	4	3	2	1	0
LED										

■ レベル 3

番号	9	8	7	6	5	4	3	2	1	0
LED										

⋮

■ レベル 9

番号	9	8	7	6	5	4	3	2	1	0
LED										

■ レベル 10

番号	9	8	7	6	5	4	3	2	1	0
LED										

■ ヒント　例題 2.4（p. 63）を基にして作ります．関数 `LED_update()` で書き込む LED 点灯パターンを，図に合わせて変更してください．

2.7　7 セグメント LED の制御

この節では，DE1-SoC に実装されている 7 セグメント LED を制御する方法について学習します．

2.7.1　DE1-SoC の 7 セグメント LED 回路

DE1-SoC に実装されている 7 セグメント LED の回路図を図 2.88 に示します．DE1-SoC には 6 個の 7 セグメント LED が実装されており，各桁の DP を除くセグメントが，それぞれ独立に FPGA のピンと接続されています．したがって，これらの 7 セグメント LED を，単純なスタティック点灯方式で制御できます．

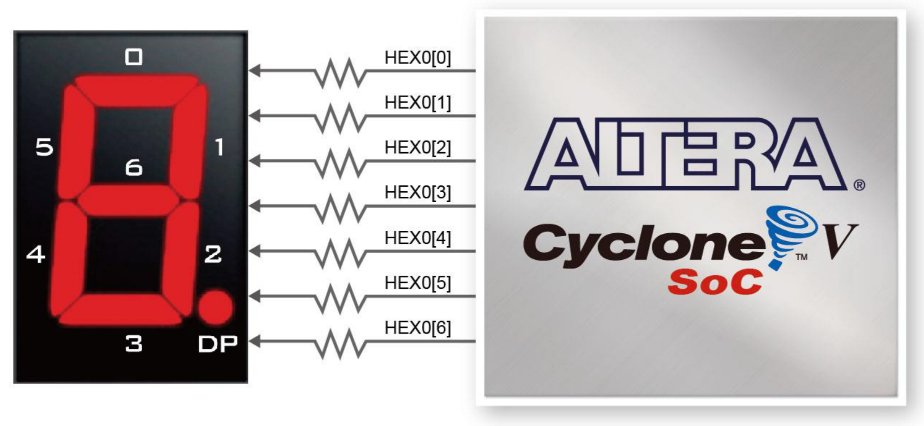


図 2.88　DE1-SoC の 7 セグメント LED「HEX0」の回路 [6]

この 7 セグメント LED はアノードコモンであり，各セグメントは L レベルで点灯，H レベルで消灯します．2.2 節で構成した各桁の制御用レジスタ（7 ビットの出力用 PIO）のビットとセグメントとの対応は，図 2.89 のとおりです．ビットを 0 に設定すると対応するセグメントが点灯し，1 に設定すると消灯します．

ビット	7	6	5	4	3	2	1	0
セグメント	未使用	6	5	4	3	2	1	0

図 2.89　7 セグメント LED の各桁制御レジスタのビットとセグメントとの対応

数字の点灯パターンの例を表 2.8 に示します．C 言語のプログラムでは，16 進数の値を各桁の制御レジスタに代入して，数字を点灯させます．

表 2.8　7 セグメント LED の点灯パターン例

数字	0	1	2	3	4
点灯パターン					
ビット列	100 0000	111 1001	010 0100	011 0000	001 1001
16 進数	0x40	0x79	0x24	0x30	0x19

数字	5	6	7	8	9
点灯パターン					
ビット列	001 0010	000 0010	111 1000	000 0000	001 1000
16 進数	0x12	0x02	0x78	0x00	0x18

2.7.2　7 セグメント LED 制御プログラムの作成

例題 2.5　7 セグメント LED への数値の表示（seven_seg1）

以下の仕様を満たすプログラムを作成してください．

- 7 セグメント LED に現在の数値を表示する．初期値は 0 とする．
- 現在の数値が 1 以上のとき，KEY3 を押すと数値が 1 減少する．
- 現在の数値が 0 のとき，KEY3 を押すと数値が 999999 になる．
- 現在の数値が 999998 以下のとき，KEY1 を押すと数値が 1 増加する．
- 現在の数値が 999999 のとき，KEY1 を押すと数値が 0 になる．
- KEY2 を押すと数値が 0 になる．

リスト 2.7　例題 2.5 の C 言語プログラム

```
1  #include <stdint.h>
2  #include <unistd.h>
3
4  #include "sys/alt_stdio.h"
5  #include "altera_avalon_pio_regs.h"
6  #include "system.h"
7
8  // ボタンスイッチの数
9  #define N_BUTTON_SW 3
10
11 // ボタンスイッチの位置の型
```

```
12 typedef enum {
13   KEY1, KEY2, KEY3
14 } ButtonSWPosition;
15
16 void SevenSeg_update(int num);
17 void ButtonSW_update_pressed(uint8_t prev, uint8_t current, uint8_t*
     pressed);
18 void wait_ms(int ms);
19
20 int main() {
21   alt_putstr("[seven_seg1]\n");
22
23   // 現在の数値
24   int num = 0;
25   // 7セグメントLEDの状態を初期化する
26   SevenSeg_update(num);
27
28   // 前回調べたボタンスイッチレジスタの値
29   uint8_t prev_button_sw = 0x0;
30   // ボタンスイッチレジスタの値
31   uint8_t button_sw;
32   // ボタンスイッチ押下フラグの配列
33   uint8_t button_sw_pressed[N_BUTTON_SW];
34
35   while (1) {
36     // ボタンスイッチレジスタの値を読み込む
37     button_sw = IORD_ALTERA_AVALON_PIO_DATA(BUTTON_SW_BASE);
38     // ボタンスイッチ押下フラグを更新する
39     ButtonSW_update_pressed(prev_button_sw, button_sw,
     button_sw_pressed);
40
41     if (button_sw_pressed[KEY2]) {
42       // KEY2が押されたとき
43       num = 0;
44       SevenSeg_update(num);
45     } else if (button_sw_pressed[KEY3]) {
46       // KEY3が押されたとき
47       if (num >= 1) {
48         num--;
49       } else {
50         num = 999999;
```

```
51        }
52
53        SevenSeg_update(num);
54      } else if (button_sw_pressed[KEY1]) {
55        // KEY1が押されたとき
56        if (num <= 999998) {
57          num++;
58        } else {
59          num = 0;
60        }
61
62        SevenSeg_update(num);
63      }
64
65      // ボタンスイッチレジスタの値を記録する
66      prev_button_sw = button_sw;
67
68      wait_ms(10);
69    }
70
71    return 0;
72  }
73
74  /**
75   * @brief 7セグメントLEDの状態を更新する。
76   * @param num 表示する数値。
77   */
78  void SevenSeg_update(int num) {
79    // 7セグメントLEDの点灯パターン
80    static const uint8_t SevenSegPattern[] = {
81      // [0],  [1],  [2],  [3],  [4],  [5],  [6],  [7],  [8],  [9]
82        0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x78, 0x00, 0x18
83    };
84
85    // 100000の位の数
86    int num_100000  = num / 100000;
87    int rest_100000 = num % 100000;
88
89    // 10000の位の数
90    int num_10000  = rest_100000 / 10000;
91    int rest_10000 = rest_100000 % 10000;
```

```
92
93    // 1000の位の数
94    int num_1000  = rest_10000 / 1000;
95    int rest_1000 = rest_10000 % 1000;
96
97    // 100の位の数
98    int num_100  = rest_1000 / 100;
99    int rest_100 = rest_1000 % 100;
100
101   // 10の位の数
102   int num_10 = rest_100 / 10;
103   // 1の位の数
104   int num_1  = rest_100 % 10;
105
106   IOWR_ALTERA_AVALON_PIO_DATA(HEX_0_BASE, SevenSegPattern[num_1]);
107   IOWR_ALTERA_AVALON_PIO_DATA(HEX_1_BASE, SevenSegPattern[num_10]);
108   IOWR_ALTERA_AVALON_PIO_DATA(HEX_2_BASE, SevenSegPattern[num_100]);
109   IOWR_ALTERA_AVALON_PIO_DATA(HEX_3_BASE, SevenSegPattern[num_1000]);
110   IOWR_ALTERA_AVALON_PIO_DATA(HEX_4_BASE, SevenSegPattern[num_10000]);
111   IOWR_ALTERA_AVALON_PIO_DATA(HEX_5_BASE, SevenSegPattern[num_100000]);
112 }
113
114 /**
115  * @brief ボタンスイッチが押されたか調べ、押下フラグを更新する。
116  * @param prev 前回取得したボタンスイッチレジスタの値。
117  * @param current 今回取得したボタンスイッチレジスタの値。
118  * @param pressed ボタンスイッチ押下フラグの配列。
119  */
120 void ButtonSW_update_pressed(uint8_t prev, uint8_t current, uint8_t*
       pressed) {
121   int i;
122   for (i = 0; i < N_BUTTON_SW; i++) {
123     // マスクビット
124     uint8_t mask = 0x1 << i;
125
126     // 前回取得した値からi番目のビットを取り出す
127     uint8_t prev_masked = prev & mask;
128     // 今回取得した値からi番目のビットを取り出す
129     uint8_t masked = current & mask;
130
131     if ((masked != prev_masked) && (masked != 0x0)) {
```

```
132        // ビットが0→1に変化していたら（立ち上がりが発生したら）
133        // ボタンが押されたと判断する
134        pressed[i] = 1;
135      } else {
136        pressed[i] = 0;
137      }
138    }
139  }
140
141  /**
142   * @brief 指定された時間だけ待機する。
143   * @param ms 待機時間（ms単位）。
144   */
145  void wait_ms(int ms) {
146    int i;
147    for (i = 0; i < ms; i++) {
148      usleep(1000);
149    }
150  }
```

7 セグメント LED への数値の表示

C 言語による 7 セグメント LED の制御では，int 型の数値の 7 セグメント LED への表示をよく行います．例題 2.5 では，関数 SevenSeg_update() がその処理を一手に担っており，この関数に数値を渡して呼び出すと，その数値が 7 セグメント LED に表示されます．

int 型の数値を 7 セグメント LED に表示する際のポイントは，数値の各桁の数を取り出すことです．0〜999999 までの数値について各桁の数を取り出すには，表 2.9 のように，最上位の桁から 10^n で割った商を求めていきます．

表 2.9　0〜999999 までの数値の各桁の数を取り出す手順

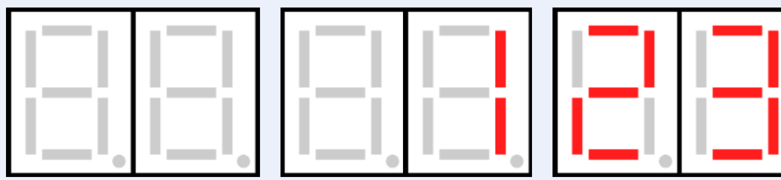
順番	計算		求められるもの
(1)	数値を 100000 で割った商	→	100000 の位の数
(2)	(1) の余りを 10000 で割った商	→	10000 の位の数
(3)	(2) の余りを 1000 で割った商	→	1000 の位の数
(4)	(3) の余りを 100 で割った商	→	100 の位の数
(5)	(4) の余りを 10 で割った商	→	10 の位の数
(6)	(5) の余り	→	1 の位の数

関数 SevenSeg_update() では，以上の手順で各桁の数を取り出した後，各桁の制御用レジスタに，取り出した数と対応する点灯パターンを書き込みます．点灯パターンはあらかじめ定数配列 SevenSegPattern に格納しておき，添字として数を指定して取り出せるようにしています．

課題 2.7 数値の右詰め表示（seven_seg2）

例題 2.5 のプログラムを変更して，下の図のように，先頭の 0 を表示せず右詰めで数値を表示するようにします．手順に従ってプログラムを作成してください．

■ 動作例（数値 123 を表示する場合）



■ 手順

(1) 各セグメントが負論理で点灯することを踏まえて，すべてのセグメントを消灯させるために各桁の制御レジスタに代入する値を 16 進数で表してください．

(2) 表示する数値の変数 num を使用して各桁を点灯させる条件を考え，C 言語の条件式で表してください．ただし，1 の位は常に点灯させます．

(3) (1)，(2) の回答を利用して，制御プログラムを作成してください．

(1) すべてのセグメントを消灯させるために各桁の制御レジスタに代入する値

2 進数 ______________________ 16 進数 0x__________

(2) 各桁を点灯させる条件

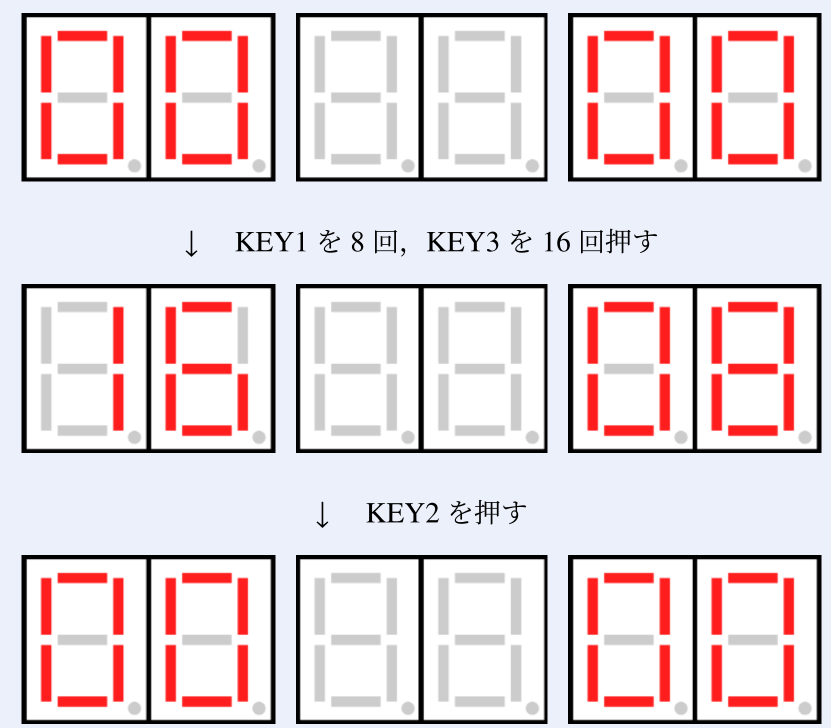
桁	点灯させる条件
1 の位	（常に点灯）
10 の位	
100 の位	
1000 の位	
10000 の位	
100000 の位	

課題 2.8 ボタンスイッチが押された回数の表示（seven_seg3）

以下の仕様を満たすプログラムを作成してください．

- KEY1，KEY3 が押された回数をそれぞれ数えて，7 セグメント LED に表示する．
- KEY1，KEY3 ともに，99 回押された後さらに押されたら，押された回数を 0 に戻す．
- KEY2 を押すと，KEY1 が押された回数および KEY3 が押された回数の両方を 0 に戻す．
- KEY1 が押された回数を HEX0 と HEX1 に，KEY3 が押された回数を HEX4 と HEX5 に，それぞれ 2 桁で表示する．
- HEX2 と HEX3 は常に消灯させる．

■ 動作例



■ ヒント KEY1 が押された回数，KEY3 が押された回数を格納する変数をそれぞれ用意して，ボタンスイッチが押されるたびに 1 ずつ増加させます．関数 `SevenSeg_update()` の引数を 2 つに増やし，これらの変数を受け取れるようにすると，見通しよく実装できます．KEY2 と KEY3 の消灯には，課題 2.7（1）の結果を利用します．

2.8 まとめ

この章では，ソフトコアプロセッサ Nios II を使用する制御システムのハードウェアを構築する方法，および IP コア「PIO（Parallel I/O）」を使用して，Nios II の基本的な入出力制御を実装する方法について学習しました．要点を以下にまとめます．

- Platform Designer を使用し，以下の手順で Nios II および周辺回路を構成する．
  1. Nios II，オンチップメモリ，JTAG UART，およびアプリケーションで使用する IP コアをマイコンシステムに追加する．
  2. 配線，Nios II のベクタ設定，ポート出力設定，アドレスの割り当てを行う．
  3. Verilog コードを生成し，ハードウェアデザインプロジェクトに追加する．
- Nios II SBT を用いて，組込みソフトウェアの開発を行う．
  - 「Hello World Small」テンプレートを用いて，アプリケーションプロジェクトおよびライブラリプロジェクト（`*_bsp`）を作成する．
  - `alt` 標準入出力関数および Nios II Debug パースペクティブを用いてデバッグを行う．
- パラレルポートの IP コア「PIO（Parallel I/O）」を用いて制御レジスタを作成し，基本的な入出力制御を行う．
  - レジスタへの書き込みには `IOWR_ALTERA_AVALON_PIO_DATA()` マクロを使用する．
  - レジスタの値の読み込みには `IORD_ALTERA_AVALON_PIO_DATA()` マクロを使用する．

第 3 章

実践的なハードウェア設計

第 2 章では，Nios II および PIO（Parallel I/O）を含む必要最小限のマイコンシステムを構築して，DE1-SoC に実装されている様々な入出力装置（LED，スイッチ，7 セグメント LED）を制御することができました．しかし，第 2 章で製作したシステムでは，通常のマイコンを使用するシステムと同様の機能しか使用していません．そのため，これまでの実習では，FPGA およびソフトコアプロセッサを使用することの利点を実感しにくかったのではないでしょうか．

ソフトコアプロセッサを使用した制御システム構築には，最小限のシステムを構成できることの他にも，様々な利点があります．ひとつは，FPGA の外部に部品を追加せずに，独自のハードウェアを追加できることです．独自のハードウェアを使うことにより，通常のマイコンを使用するシステムと比べて，動作が速くなったり，組込みソフトウェアを記述しやすくなったりすることが期待できます．また，FPGA には，内部の信号を観測できるロジックアナライザも追加できます．テストベンチとともにこのロジックアナライザを活用すると，システムの動作検証およびハードウェアの改良が容易になります．この 2 つの機能は，通常のマイコンを使用する制御システムにはない大きな利点です．

この章では，以上の 2 つの機能を活用した実践的なハードウェア設計に取り組みます．3.1 節では，2.7.2 項においてソフトウェアで実装した 7 セグメント LED への数値の表示処理を具体例として，ソフトウェアで記述した処理をハードウェアに移行します．3.2 節では，FPGA にロジックアナライザを組み込み，その観測結果を利用してトグルスイッチのチャタリング除去回路を設計します．

3.1　ソフトウェアで記述した処理のハードウェア化

この節では，ソフトウェアで記述した処理をハードウェアに移行する方法について学習します．題材として，7 セグメント LED への数値表示処理を取り上げます．

3.1.1　トレードオフを考慮したシステム設計

FPGA とマイコンを組み合わせた制御システムを設計する際には，必要な処理をどちらのデバイスに実装するか，選択する必要があります．それでは，どのような基準で適切なデバイスを選択すればよいのでしょうか．

表 3.1 に，FPGA 開発とマイコン開発の比較結果をまとめます．

まず性能について，FPGA を使うと，一部の処理をマイコンよりも高速に実行できる可能性があります．これは，マイコンが処理に対応する命令を 1 つずつメモリから読み出して実行するのに対して，FPGA には処理が回路として組み込まれるためです．

次に回路の柔軟性について，FPGA 開発では，HDL の記述や IP コアの追加によって回路を自在に変更することができます．一方で，マイコンの回路は一切変更できません（そのため，一般にマイコンには，使用の有無と関係なく非常に多くのペリフェラル（周辺回路）が搭載されています）．

プログラムの記述については，FPGA 開発（HDL）よりもマイコン開発（C 言語等）の方が柔軟です．FPGA 開発では，ハードウェアへの変換を意識してコードを書く必要があります．また，マイコンからどのように使われるかも想定しながら，ハードウェアを設計しなければなりません．そのため，マイコン開発と比べて記述に制約があります．

消費電力については，マイコンよりも FPGA の方が少なくなります．これは，マイコンでは汎用性のある大規模な回路を高いクロック周波数で動作させる必要があるのに対して，FPGA では専用の回路を低いクロック周波数で動作させればよいためです．

最後に，開発効率は，FPGA 開発よりもマイコン開発の方が高いといえます．FPGA 開発でハードウェアを変更する際には，コンパイルに長い時間を要し，また変更に合わせてマイコンのプログラムも修正する必要があります．マイコン開発では，多くの場合組込みソフトウェアの変更だけで済み，ソフトウェアのコンパイル時間もハードウェアのコンパイルと比較して短くなります．

製品を開発する際には，これらの要素に関するトレードオフを考慮して，適切にシステムを設計する必要があります．

表 3.1　FPGA 開発とマイコン開発の比較（○：他方より有利，△：他方より不利，×：他方より大きく不利）．文献 [10, 11] を参考にして作成した．

デバイス	性能	回路の柔軟性	プログラム記述の柔軟性	消費電力	開発効率
FPGA	○	○	△	○	△
マイコン	△	×	○	△	○

7 セグメント LED への数値表示処理のハードウェア化

2.7.2 項で実装した 7 セグメント LED への数値表示処理（`SevenSeg_update()` 関数）は，プログラムの行数が少ない割には負荷の高い処理となっています．これは，数値から各桁の数字を取り出すのに除算（割り算）を多く行っているためです．システムで使用している Nios II/e コアには除算命令が含まれていないため，除算はソフトウェア的に多くの段階を踏んで実行されます．すなわち，C 言語では 1 つの演算子（`/`，`%`）で表現されていた部分が，機械語としては長いプログラムに変換されてしまいます．

例えばこのような処理は，C 言語ではなく HDL で記述すること（＝ハードウェア化）によって，低負荷化，高速化できます．

3.1.2　7 セグメント LED への数値表示処理のハードウェア仕様

7 セグメント LED への数値表示処理を実際にハードウェア化する前に，仕様を決めておきます．3.1.1 項で述べたように，HDL での記述（FPGA 開発）と C 言語での記述（マイコン開発）とを比較すると，様々な要素にトレードオフの関係があります．例えば，7 セグメント LED への数値表示処理をハードウェア化すると，高速化と引き換えに，7 セグメント LED に数値以外の情報を表示するのが難しくなります[*1]．また，制御レジスタを不適切に設計すると，C 言語プログラムからその値を正しく設定するの

[*1] 不可能ではありませんが，仕様がとても複雑になるでしょう．

が難しくなります．以上の例のように，ハードウェア化の際には，様々な要素を考慮して仕様を決めなければなりません．

今回は，以下に示す仕様で，7 セグメント LED への数値表示処理をハードウェア化してみます．

仕様

- 7 セグメント LED の用途を数値の表示に限定する（⇒他の点灯パターンを考慮する必要がなく，処理が簡潔になる）．
- 先頭の 0 の表示の有無を設定できる（→ p. 74 の課題 2.7）．
- 以下に示す，2 つの制御レジスタを用意する．
  - **数値レジスタ**：7 セグメント LED に表示する数値を設定するレジスタ．20 ビット長で，`0`〜`999999` の値を格納することを想定する．
  - **ゼロ埋め制御レジスタ**：先頭の 0 の表示の有無を制御する，1 ビット長のレジスタ．
    - ∗ `0` を設定→先頭の 0 を表示する．
      - · 例：数値レジスタに `123` を設定→ `000123` を表示
    - ∗ 1 を設定→先頭の 0 を表示しない．
      - · 例：数値レジスタに `123` を設定→`␣␣␣123` を表示
- 1000000〜1048575（20 ビットで表せる数の最大値）を表示しようとした場合，100000 の位を「A」と表示する．

3.1.3　7 セグメント LED への数値表示処理の実装

それでは，7 セグメント LED への数値表示処理を行うハードウェアを実装していきましょう．最初に，先頭の 0 が常に表示される，単純な数値表示処理を実装します．実装にあたっては，処理が正しいことを確認するため，テストベンチの記述および論理シミュレーションも行います．論理シミュレーションでは，ModelSim を単独で使用します．

ModelSim プロジェクトの作成

まず，ModelSim のプロジェクトを作成します．

デスクトップ上にある ModelSim のショートカットから，ModelSim を起動します．メニューから「File」→「New」→「Project...」を選択します（図 3.1）．現在開いているプロジェクトを閉じるか質問された場合は，「はい」ボタンをクリックして閉じます．

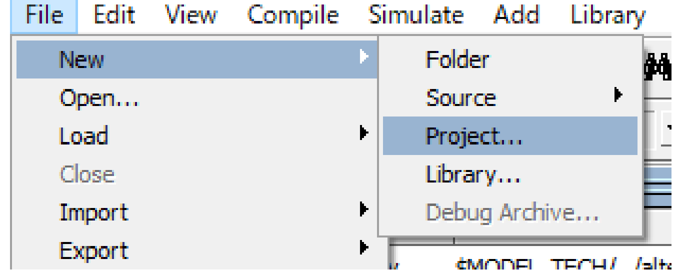


図 3.1　新規プロジェクト作成の選択

続いて，「Create Project」画面（図 3.2）が表示されます．表 3.2 のように設定して「OK」ボタンをクリックすると，プロジェクトが作成されます．その際にフォルダを作成するか質問された場合は，「はい」ボタンをクリックしてフォルダを作成します．

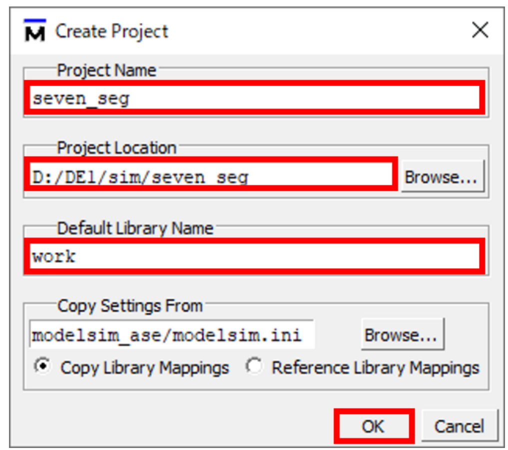


図 3.2　Create Project

表 3.2　プロジェクト作成時の設定

項目	設定値
プロジェクト名（Project Name）	seven_seg
プロジェクトの場所（Project Location）	D:/DE1/sim/seven_seg
既定のライブラリ名（Default Library Name）	work

モジュールのインターフェースの整備

ここから Verilog コードを記述していきます．最初に，7 セグメント LED への数値表示処理を担当する Verilog モジュール（モジュール名：seven_seg_decoder）のインターフェースを表 3.3 のように整備して，最低限の入出力ができるようにします．この時点では，入力信号と関係なく 7 セグメント LED を常に消灯させておきます．

表 3.3　seven_seg_decoder モジュールのインターフェース

ポート名	方向	ビット幅	信号の意味
num	入力	20	7 セグメント LED に表示する数値
zero_suppress	入力	1	先頭の 0 を非表示にするか
hex0	出力	7	HEX0（1 の位）に出力する信号
hex1	出力	7	HEX1（10 の位）に出力する信号
hex2	出力	7	HEX2（100 の位）に出力する信号
hex3	出力	7	HEX3（1000 の位）に出力する信号
hex4	出力	7	HEX4（10000 の位）に出力する信号
hex5	出力	7	HEX5（100000 の位）に出力する信号

Quartus Prime 等のエディタを使用して以下の Verilog コードを記述し，`D:\DE1\sim\seven_seg\seven_seg_decoder.v` として保存します．

リスト 3.1　モジュールのインターフェースの整備（`seven_seg_decoder.v`）

```
1  // 7セグメントLEDへの数値表示処理を担当するモジュール
2  module seven_seg_decoder(
3    // 7セグメントLEDに表示する数値
4    input [19:0] num,
5    // 先頭の0を非表示にするか
6    input zero_suppress,
7
8    // HEX0（1の位）に出力する信号
9    output [6:0] hex0,
10   // HEX1（10の位）に出力する信号
11   output [6:0] hex1,
12   // HEX2（100の位）に出力する信号
13   output [6:0] hex2,
14   // HEX3（1000の位）に出力する信号
15   output [6:0] hex3,
16   // HEX4（10000の位）に出力する信号
17   output [6:0] hex4,
18   // HEX5（100000の位）に出力する信号
19   output [6:0] hex5
20 );
21
22 // 全消灯
23 parameter PATTERN_OFF = 7'b111_1111;
24
25 // すべての桁を全消灯する
26 assign hex0 = PATTERN_OFF;
27 assign hex1 = PATTERN_OFF;
28 assign hex2 = PATTERN_OFF;
29 assign hex3 = PATTERN_OFF;
30 assign hex4 = PATTERN_OFF;
31 assign hex5 = PATTERN_OFF;
32
33 endmodule
```

コードの記述後，ソースファイルを ModelSim プロジェクトに追加します．「Add items to the Project」画面（図 3.3）において，「Add Existing File」をクリックします．

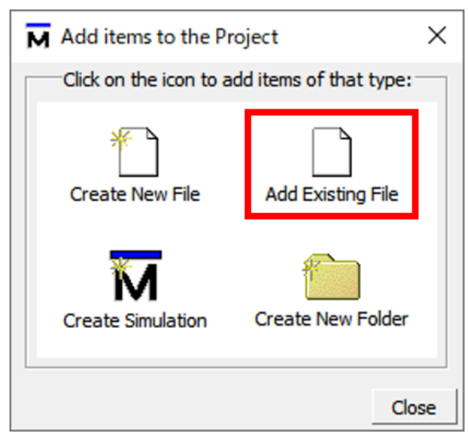


図 3.3　Add items to the Project

続いて表示される「Add file to Project」画面（図 3.4）において，「Browse...」ボタンをクリックしてファイル選択画面を表示し，`D:\DE1\sim\seven_seg\seven_seg_decoder.v` を選択します．「OK」ボタンをクリックすると，ソースファイルがプロジェクトに追加されます．

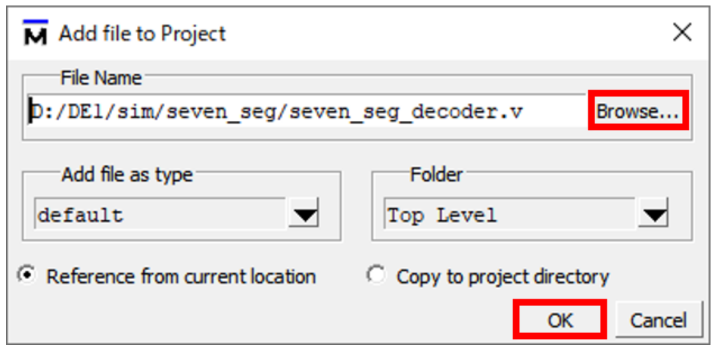


図 3.4　Add file to Project

テストベンチの追加

今回は，入力および望ましい動作（テストケース）が記述されたテストベンチを先に作成し，それらを満たすようにコードを記述するという手順で，ハードウェアを実装していきます．このような手順での開発は，**テスト駆動開発**（Test-Driven Development；TDD）と呼ばれています．

テストケースとして，表 3.4 に示す入出力の組合せを考えます．このテストでは，入力する数値の桁数を変えて，各桁数において先頭の 0 が表示されることを確認します．また，0〜9 の数字がいずれかのテストケースで出現するようにしてあります．さらに，1000000 以上の数値を入力した場合に 100000 の位が「A」となるかも確認します．

表 3.4　7 セグメント LED への数値表示処理のテストケース．hex0〜hex5 列は，その桁に表示されるべき字を示す．

番号	num	zero_suppress	hex5	hex4	hex3	hex2	hex1	hex0
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	1
3	23	0	0	0	0	0	2	3
4	456	0	0	0	0	4	5	6
5	7890	0	0	0	7	8	9	0
6	54321	0	0	5	4	3	2	1
7	987654	0	9	8	7	6	5	4
8	1048575	0	A	4	8	5	7	5

以上のテストケースが含まれるテストベンチファイル seven_seg_decoder_test.v（リスト 3.2）を配布します．このファイルを D:\DE1\sim\seven_seg\ 内にコピーし，「Add items to the Project」画面の「Add Existing File」を使用してプロジェクトに追加してください．ファイルを追加したら，「Add items to the Project」画面を，「Close」ボタンをクリックして閉じます．

リスト 3.2　7 セグメント LED への数値表示処理のテストベンチ（seven_seg_decoder_test.v）

```
1  `timescale 1 ms / 1 ms
2  
3  // テストベンチの最上位階層
4  // テストベンチなので、入出力ポートはない
5  module seven_seg_decoder_test;
6  
7  // 経過時間の単位
8  parameter STEP = 10;
9  
10 // 点灯パターン
11 parameter PATTERN_0 = 7'b100_0000;
12 parameter PATTERN_1 = 7'b111_1001;
13 parameter PATTERN_2 = 7'b010_0100;
14 parameter PATTERN_3 = 7'b011_0000;
15 parameter PATTERN_4 = 7'b001_1001;
16 parameter PATTERN_5 = 7'b001_0010;
17 parameter PATTERN_6 = 7'b000_0010;
18 parameter PATTERN_7 = 7'b111_1000;
19 parameter PATTERN_8 = 7'b000_0000;
20 parameter PATTERN_9 = 7'b001_1000;
21 parameter PATTERN_A = 7'b000_1000;
22 
23 // 全消灯
24 parameter PATTERN_OFF = 7'b111_1111;
25 
26 // 実際に出力された信号
27 wire [6:0] actual_hex0;
28 wire [6:0] actual_hex1;
29 wire [6:0] actual_hex2;
30 wire [6:0] actual_hex3;
31 wire [6:0] actual_hex4;
```

```
32 wire [6:0] actual_hex5;
33
34 // 7セグメントLEDに表示する数値
35 reg [19:0] num;
36
37 // 出力信号の期待値
38 reg [6:0] expected_hex0;
39 reg [6:0] expected_hex1;
40 reg [6:0] expected_hex2;
41 reg [6:0] expected_hex3;
42 reg [6:0] expected_hex4;
43 reg [6:0] expected_hex5;
44
45 // 各桁で、実際の出力信号が期待値と一致していたか
46 // 0: 一致していなかった
47 // 1: 一致していた
48 wire [5:0] matched;
49
50 // 7セグメントLEDデコーダモジュールの実体
51 seven_seg_decoder d0 (
52   .num (num),
53   .zero_suppress(1'b0),
54   .hex0 (actual_hex0),
55   .hex1 (actual_hex1),
56   .hex2 (actual_hex2),
57   .hex3 (actual_hex3),
58   .hex4 (actual_hex4),
59   .hex5 (actual_hex5)
60 );
61
62 // 各桁について、出力信号が期待値と一致していたか調べる
63 assign matched[0] = (actual_hex0 == expected_hex0);
64 assign matched[1] = (actual_hex1 == expected_hex1);
65 assign matched[2] = (actual_hex2 == expected_hex2);
66 assign matched[3] = (actual_hex3 == expected_hex3);
67 assign matched[4] = (actual_hex4 == expected_hex4);
68 assign matched[5] = (actual_hex5 == expected_hex5);
69
70 initial begin
71   num <= 20'd0;
72   expected_hex0 <= PATTERN_0;
73   expected_hex1 <= PATTERN_0;
74   expected_hex2 <= PATTERN_0;
75   expected_hex3 <= PATTERN_0;
76   expected_hex4 <= PATTERN_0;
77   expected_hex5 <= PATTERN_0;
78
79   #STEP
80   num <= 20'd1;
81   expected_hex0 <= PATTERN_1;
82   expected_hex1 <= PATTERN_0;
```

```
83     expected_hex2 <= PATTERN_0;
84     expected_hex3 <= PATTERN_0;
85     expected_hex4 <= PATTERN_0;
86     expected_hex5 <= PATTERN_0;
87
88     #STEP
89     num <= 20'd23;
90     expected_hex0 <= PATTERN_3;
91     expected_hex1 <= PATTERN_2;
92     expected_hex2 <= PATTERN_0;
93     expected_hex3 <= PATTERN_0;
94     expected_hex4 <= PATTERN_0;
95     expected_hex5 <= PATTERN_0;
96
97     #STEP
98     num <= 20'd456;
99     expected_hex0 <= PATTERN_6;
100    expected_hex1 <= PATTERN_5;
101    expected_hex2 <= PATTERN_4;
102    expected_hex3 <= PATTERN_0;
103    expected_hex4 <= PATTERN_0;
104    expected_hex5 <= PATTERN_0;
105
106    #STEP
107    num <= 20'd7890;
108    expected_hex0 <= PATTERN_0;
109    expected_hex1 <= PATTERN_9;
110    expected_hex2 <= PATTERN_8;
111    expected_hex3 <= PATTERN_7;
112    expected_hex4 <= PATTERN_0;
113    expected_hex5 <= PATTERN_0;
114
115    #STEP
116    num <= 20'd54321;
117    expected_hex0 <= PATTERN_1;
118    expected_hex1 <= PATTERN_2;
119    expected_hex2 <= PATTERN_3;
120    expected_hex3 <= PATTERN_4;
121    expected_hex4 <= PATTERN_5;
122    expected_hex5 <= PATTERN_0;
123
124    #STEP
125    num <= 20'd987654;
126    expected_hex0 <= PATTERN_4;
127    expected_hex1 <= PATTERN_5;
128    expected_hex2 <= PATTERN_6;
129    expected_hex3 <= PATTERN_7;
130    expected_hex4 <= PATTERN_8;
131    expected_hex5 <= PATTERN_9;
132
133    #STEP
```

```
134    num <= 20'd1048575;
135    expected_hex0 <= PATTERN_5;
136    expected_hex1 <= PATTERN_7;
137    expected_hex2 <= PATTERN_5;
138    expected_hex3 <= PATTERN_8;
139    expected_hex4 <= PATTERN_4;
140    expected_hex5 <= PATTERN_A;
141
142    #STEP
143    $stop;
144  end
145
146  endmodule
```

このテストベンチでは，各桁の出力信号と期待値との一致を示す（一致していれば 1，一致していなければ 0），matched という信号線を定義してあります．この信号線を確認することで，正常動作か否かの判断を簡単に行えます．

論理シミュレーション（実装前）

モジュールを実装する前に，一度論理シミュレーションを実行しておきます．現段階では，モジュールおよびテストベンチのコンパイルは成功し，出力信号は期待どおりにならないはずです．論理シミュレーションを実行して，この 2 つを確認します．

まず，プロジェクトに追加した 2 つのソースファイルをコンパイルします．コンパイル前は，プロジェクトのファイル一覧の「Status」欄が「?」となっています（図 3.5）．メニューから「Compile」→「Compile All」（図 3.6）を選択すると，ソースファイルがすべてコンパイルされます．コンパイルが成功すると，「Status」欄にチェックマークがつきます（図 3.7）．コンパイルに失敗した場合は，ソースコードを修正し，再度コンパイルしてください．

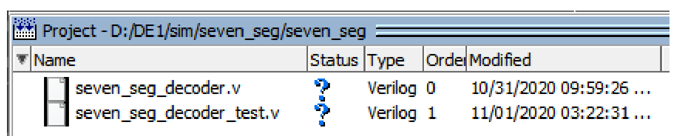


図 3.5　プロジェクトのファイル一覧（コンパイル前）

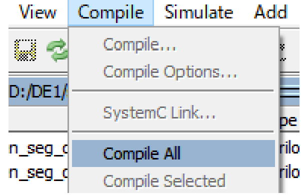


図 3.6　Compile All

図 3.7　プロジェクトのファイル一覧（コンパイル後）

コンパイル後，メニューから「Simulate」→「Start Simulation...」（図 3.8）を選択します．続いて表示される「Start Simulation」画面の「Design」タブにおいて，「`work`」「`seven_seg_decoder_test`」を選択して，「OK」ボタンをクリックします（図 3.9）．その後，ModelSim の画面が「Simulate」レイアウト（図 3.10）に変わります．

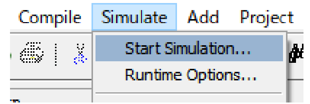


図 3.8　Start Simulation

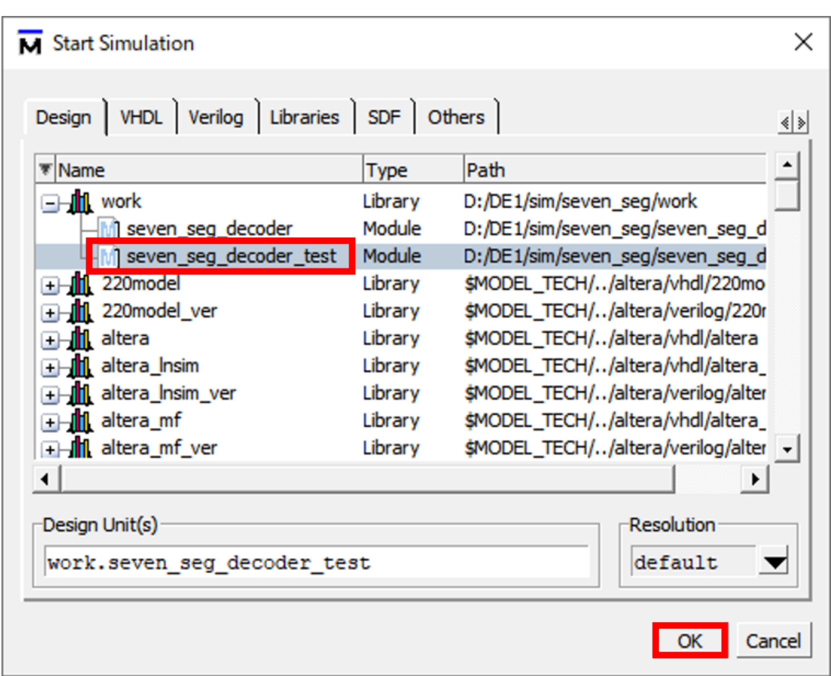


図 3.9　論理シミュレーションで使用するモジュールの選択

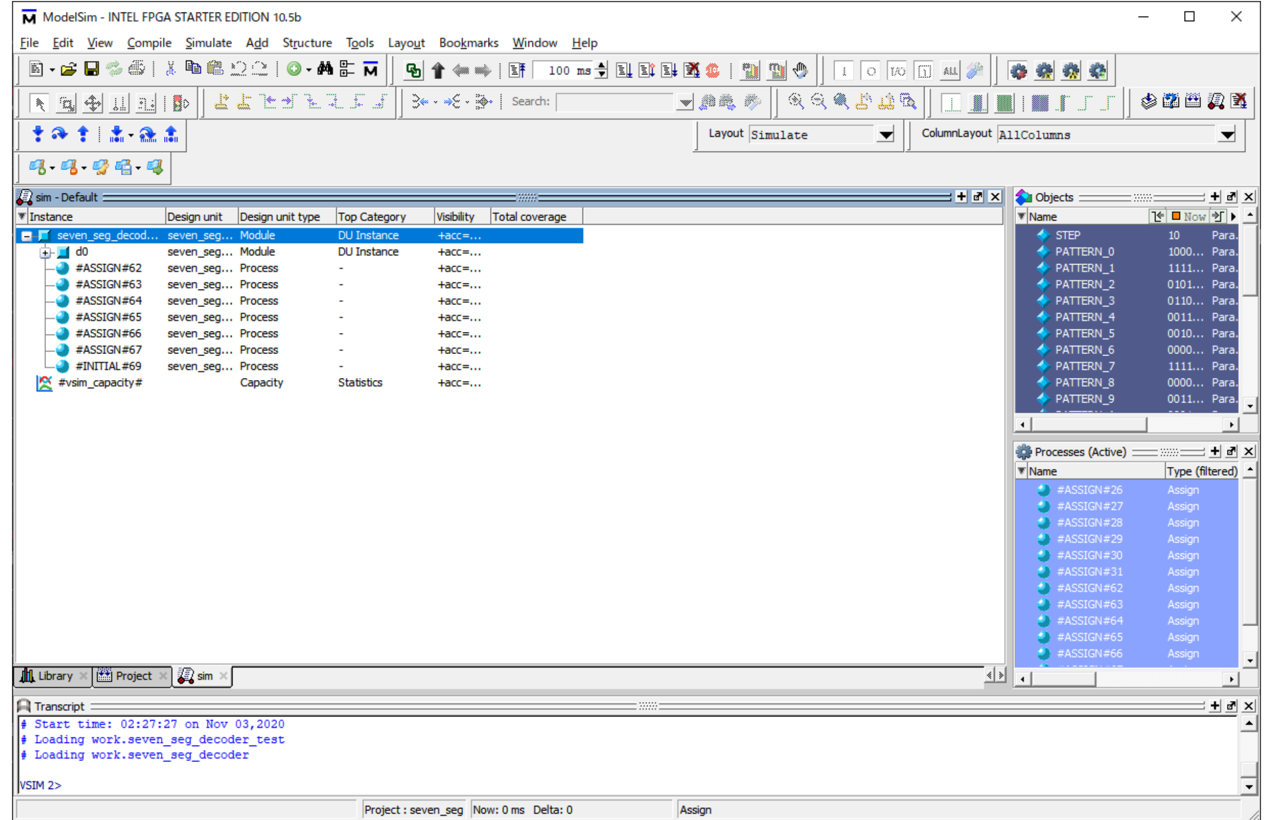


図 3.10　Simulate レイアウト

論理シミュレーションでは出力波形を観察したいので，出力波形を表示します．メニューから「View」→「Wave」（図 3.11）を選択すると，出力波形を表示する Wave ペインが現れます．そのままでは Wave ペインの幅が狭すぎるため，図 3.12 のように画面レイアウトを調整してください．

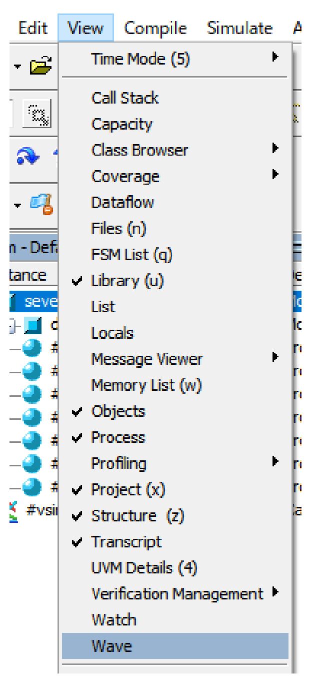


図 3.11　Wave ペインの表示

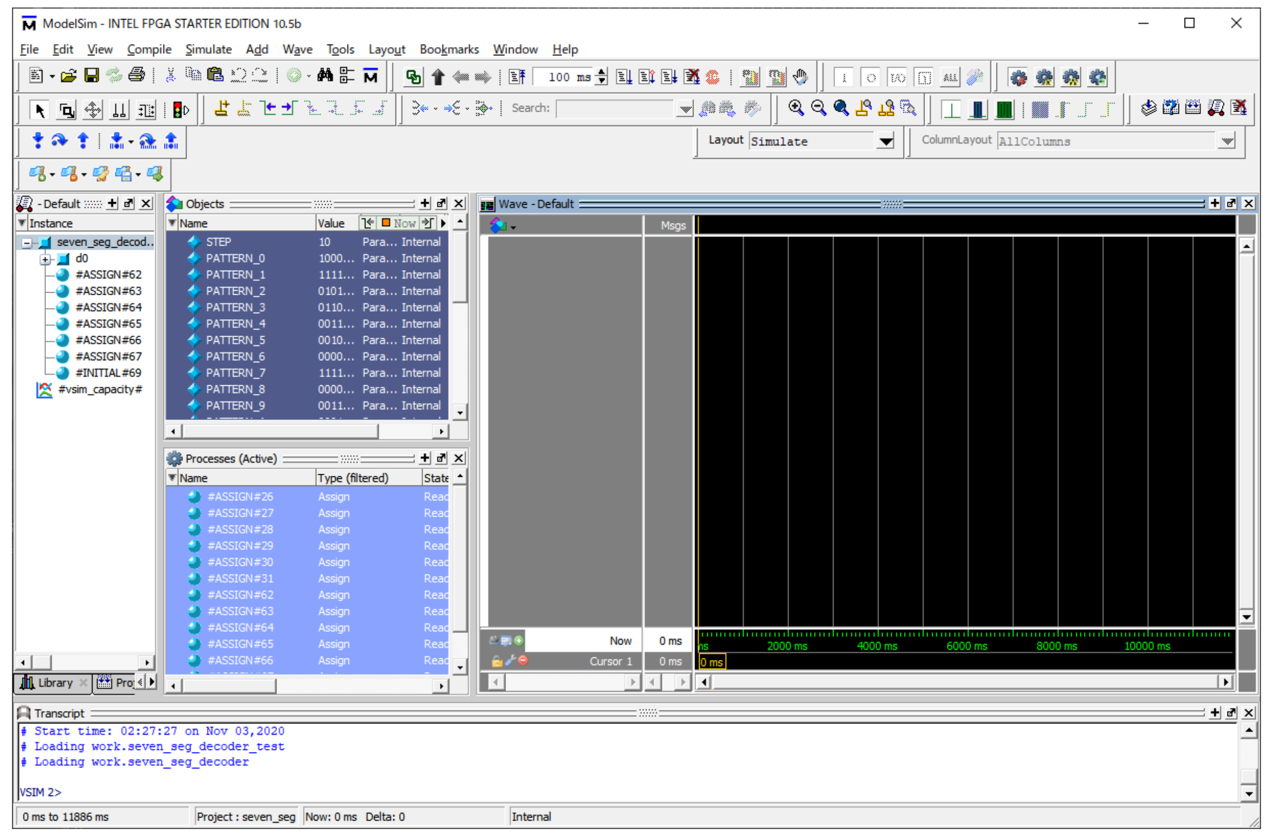


図 3.12　レイアウト調整後の ModelSim 画面

続いて，観察する信号を選択します．Objects ペインから Wave ペインに，以下の信号をドラッグ&ドロップします．

- num
- matched
- actual_hex0
- actual_hex1
- actual_hex2
- actual_hex3
- actual_hex4
- actual_hex5

信号を追加した直後は，信号名が長く表示されていて見づらいため，Wave ペイン下部の「Toggle leaf names <-> full names」ボタンをクリックして，短い名前で表示するようにします．以上を行った後の画面を，図 3.13 に示します．

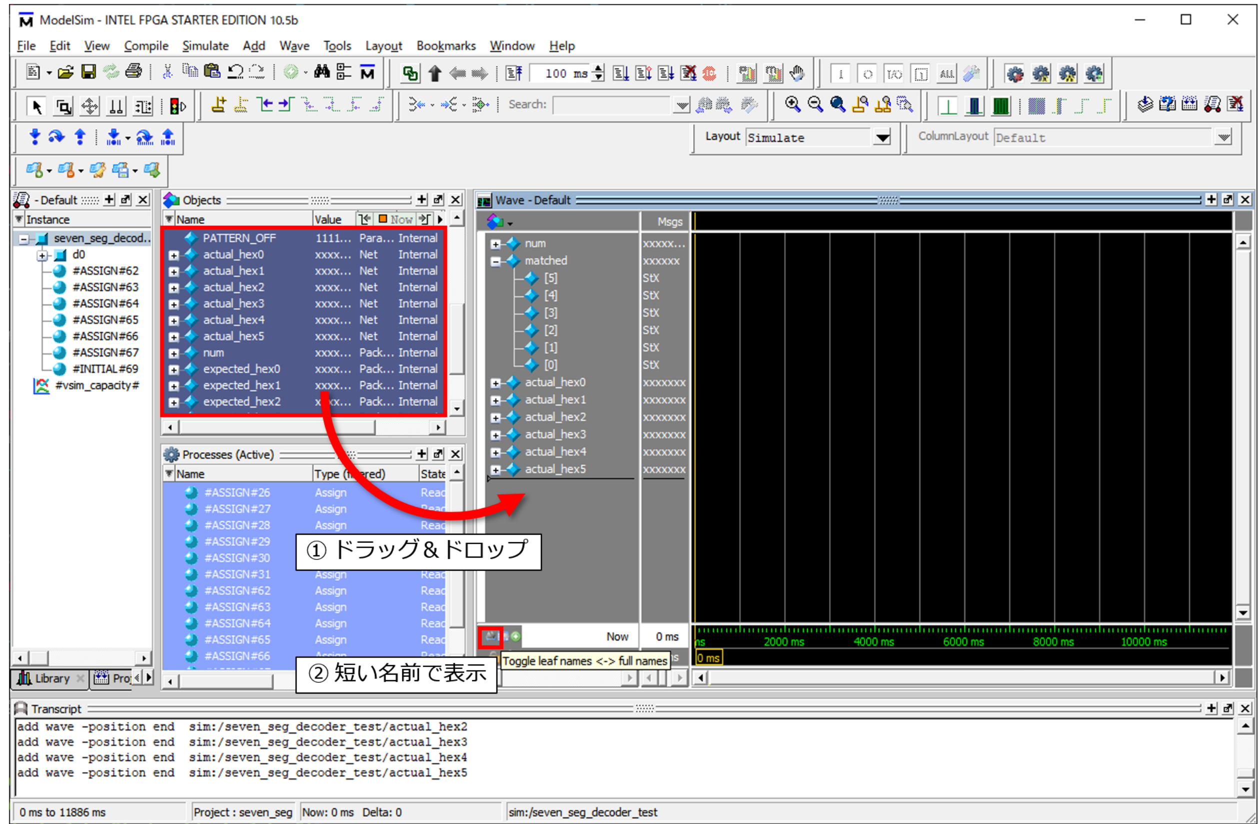


図 3.13　観察する信号の選択

以上で論理シミュレーションの設定が完了したので，上部のツールバーの「Run -All」ボタン（図 3.14）をクリックして，シミュレーションを実行します．シミュレーションの終了後，Wave ペインにおいて「Wave」タブを選択し，ツールバーの「Zoom Full」ボタンをクリックすると，図 3.15 のように出力波形が表示されます．

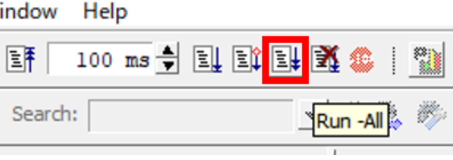


図 3.14　Run -All

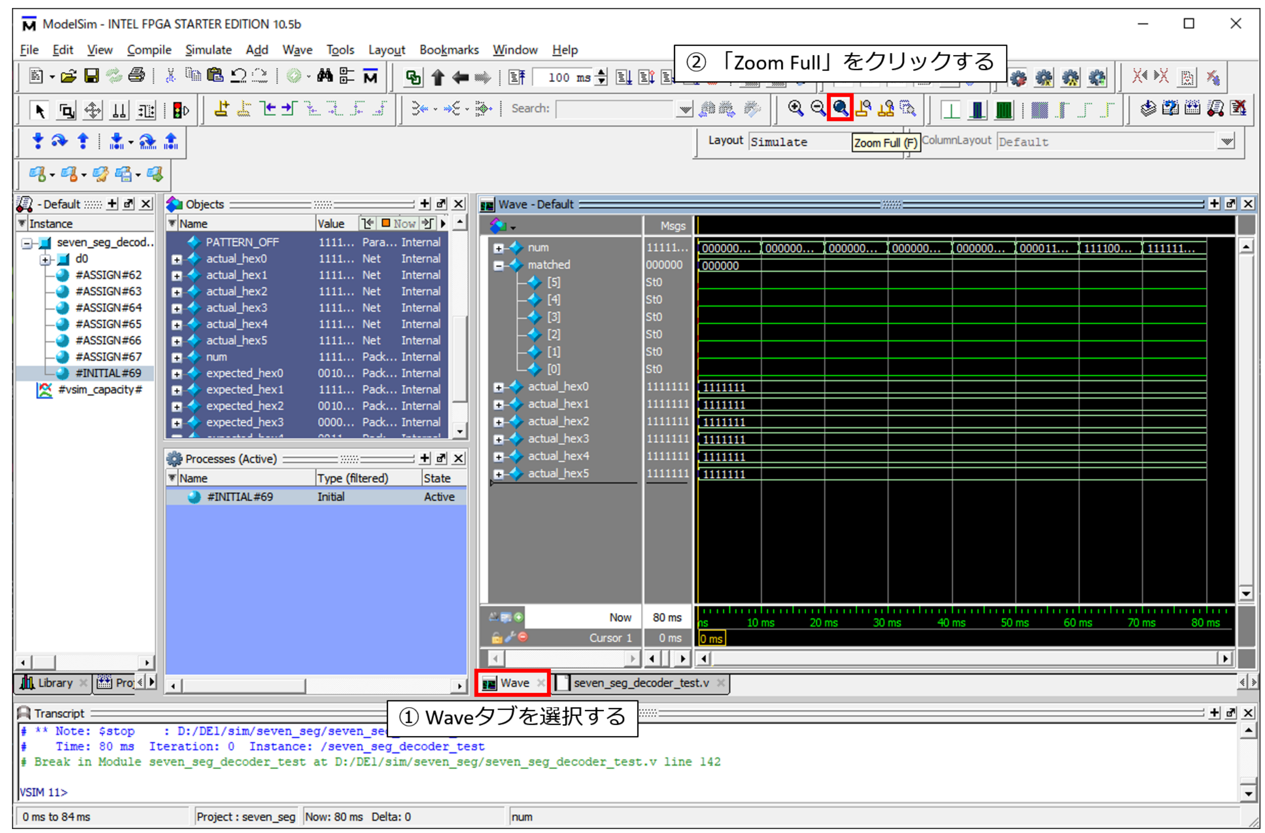


図 3.15　出力波形の表示

出力波形を拡大して図 3.16 に示します．出力信号と期待値との一致を示す matched は，常に 000000（いずれの桁も出力信号が期待値と異なる）となっています．現段階ではすべての桁を常に消灯させているため，これは正常です．これから，matched の全ビットが 1 となる（出力信号が期待値と一致する）ことを目標として，Verilog コードを記述していきます．

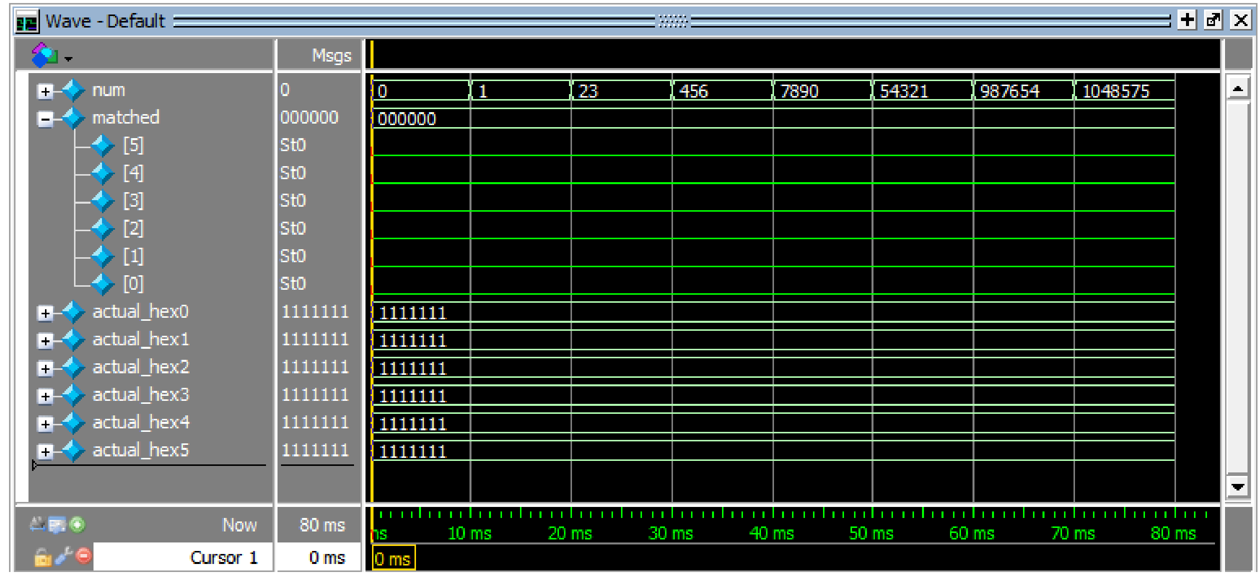


図 3.16　モジュール実装前の出力波形

なお，7 セグメント LED に表示する数値を示す num の値は，ビット数が多いため一部のみ表示されています．この値を符号なしの 10 進数として表示すると，どのテストケースかが分かりやすくなりま

す．Wave ペインの num の上で右クリックしてメニューを表示し，「Radix」→「Unsigned」を選択すると，値の表現が変わります（図 3.17）.

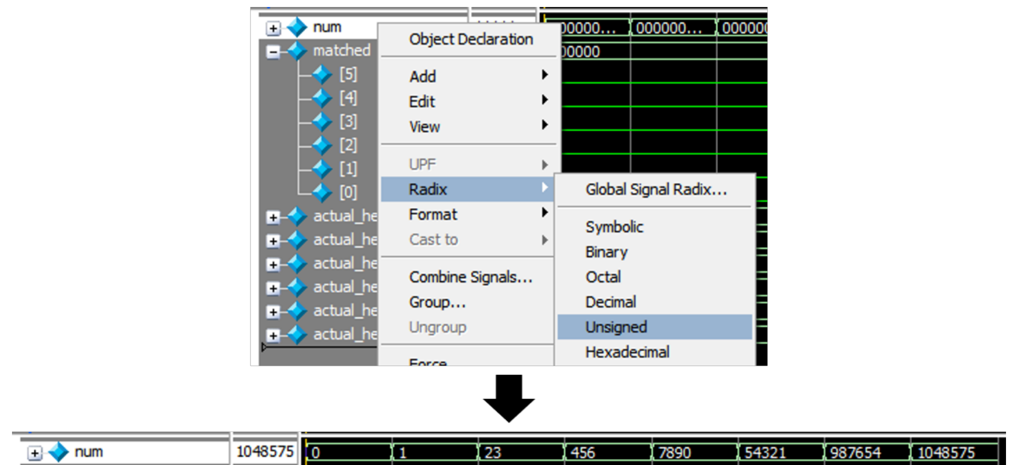


図 3.17　符号なし 10 進数での表示

モジュールの実装

論理シミュレーションの後，7 セグメント LED への数値表示処理を Verilog で実装します．ポート宣言が記述されている seven_seg_decoder.v を，リスト 3.3 のように書き換えてください．

リスト 3.3　7 セグメント LED への数値表示の実装（seven_seg_decoder.v）

```
1  // 7セグメントLEDへの数値表示処理を担当するモジュール
2  module seven_seg_decoder(
3    // 7セグメントLEDに表示する数値
4    input [19:0] num,
5    // 先頭の0を非表示にするか
6    input zero_suppress,
7
8    // HEX0（1の位）に出力する信号
9    output [6:0] hex0,
10   // HEX1（10の位）に出力する信号
11   output [6:0] hex1,
12   // HEX2（100の位）に出力する信号
13   output [6:0] hex2,
14   // HEX3（1000の位）に出力する信号
15   output [6:0] hex3,
16   // HEX4（10000の位）に出力する信号
17   output [6:0] hex4,
18   // HEX5（100000の位）に出力する信号
19   output [6:0] hex5
20 );
21
22 // 点灯パターン
23 parameter PATTERN_0 = 7'b100_0000;
24 parameter PATTERN_1 = 7'b111_1001;
25 parameter PATTERN_2 = 7'b010_0100;
26 parameter PATTERN_3 = 7'b011_0000;
```

```
27 parameter PATTERN_4 = 7'b001_1001;
28 parameter PATTERN_5 = 7'b001_0010;
29 parameter PATTERN_6 = 7'b000_0010;
30 parameter PATTERN_7 = 7'b111_1000;
31 parameter PATTERN_8 = 7'b000_0000;
32 parameter PATTERN_9 = 7'b001_1000;
33 parameter PATTERN_A = 7'b000_1000;
34
35 // 全消灯
36 parameter PATTERN_OFF = 7'b111_1111;
37
38 // 数値に対応する点灯パターンを返す
39 function [6:0] decode;
40   // 数値
41   input [3:0] n;
42
43   case (n)
44     4'd0:    decode = PATTERN_0;
45     4'd1:    decode = PATTERN_1;
46     4'd2:    decode = PATTERN_2;
47     4'd3:    decode = PATTERN_3;
48     4'd4:    decode = PATTERN_4;
49     4'd5:    decode = PATTERN_5;
50     4'd6:    decode = PATTERN_6;
51     4'd7:    decode = PATTERN_7;
52     4'd8:    decode = PATTERN_8;
53     4'd9:    decode = PATTERN_9;
54     default: decode = PATTERN_A;
55   endcase
56 endfunction
57
58 // 数値を100000で割った商
59 wire [19:0] quotient_100000;
60 // 数値を100000で割った余り
61 wire [19:0] remainder_100000;
62 // 100000の位の数
63 wire [3:0]  num_100000;
64 // 100000の位の数を取り出した後の残り
65 wire [16:0] rest_100000;
66
67 // 数値を10000で割った商
68 wire [16:0] quotient_10000;
69 // 数値を10000で割った余り
70 wire [16:0] remainder_10000;
71 // 10000の位の数
72 wire [3:0]  num_10000;
73 // 10000の位の数を取り出した後の残り
74 wire [13:0] rest_10000;
75
76 // 数値を1000で割った商
77 wire [13:0] quotient_1000;
```

```
78  // 数値を1000で割った余り
79  wire [13:0] remainder_1000;
80  // 1000の位の数
81  wire [3:0] num_1000;
82  // 1000の位の数を取り出した後の残り
83  wire [9:0] rest_1000;
84
85  // 数値を100で割った商
86  wire [9:0] quotient_100;
87  // 数値を100で割った余り
88  wire [9:0] remainder_100;
89  // 100の位の数
90  wire [3:0] num_100;
91  // 100の位の数を取り出した後の残り
92  wire [6:0] rest_100;
93
94  // 数値を10で割った商
95  wire [6:0] quotient_10;
96  // 数値を10で割った余り
97  wire [6:0] remainder_10;
98  // 10の位の数
99  wire [3:0] num_10;
100 // 1の位の数
101 wire [3:0] num_1;
102
103 // 100000の位の数を取り出す
104 assign quotient_100000  = num / 17'd100000;
105 assign remainder_100000 = num % 17'd100000;
106 assign num_100000  = quotient_100000[3:0];
107 assign rest_100000 = remainder_100000[16:0];
108
109 // 10000の位の数を取り出す
110 assign quotient_10000  = rest_100000 / 14'd10000;
111 assign remainder_10000 = rest_100000 % 14'd10000;
112 assign num_10000  = quotient_10000[3:0];
113 assign rest_10000 = remainder_10000[13:0];
114
115 // 1000の位の数を取り出す
116 assign quotient_1000  = rest_10000 / 10'd1000;
117 assign remainder_1000 = rest_10000 % 10'd1000;
118 assign num_1000  = quotient_1000[3:0];
119 assign rest_1000 = remainder_1000[9:0];
120
121 // 100の位の数を取り出す
122 assign quotient_100  = rest_1000 / 7'd100;
123 assign remainder_100 = rest_1000 % 7'd100;
124 assign num_100  = quotient_100[3:0];
125 assign rest_100 = remainder_100[3:0];
126
127 // 10の位、1の位の数を取り出す
128 assign quotient_10  = rest_100 / 4'd10;
```

```
129 assign remainder_10 = rest_100 % 4'd10;
130 assign num_10 = quotient_10[3:0];
131 assign num_1  = remainder_10[3:0];
132
133 // 各桁の数と対応する点灯パターンを出力する
134 assign hex0 = decode(num_1);
135 assign hex1 = decode(num_10);
136 assign hex2 = decode(num_100);
137 assign hex3 = decode(num_1000);
138 assign hex4 = decode(num_10000);
139 assign hex5 = decode(num_100000);
140
141 endmodule
```

このモジュールには，入力された数値の各桁の数を取り出す処理と，各桁の数を 7 セグメント LED の点灯パターンにデコードする処理が含まれています．前者について，C 言語での記述（p. 69 のリスト 2.7）と比較すると，Verilog コードは，コンパイル時の警告防止のため信号線のビット幅を明示している点が特徴的です．後者は，関数 decode に実装されています．

論理シミュレーション（実装後）

Verilog コードの記述後，論理シミュレーションによる動作確認を行います．

まず，ソースコードの変更をシミュレーションに反映させるため，ソースファイルを再コンパイルします．メニューから「Compile」→「Compile All」を選択すると，ソースコードを記述した seven_seg_decoder.v が再コンパイルされます．

再コンパイル後，論理シミュレーションを最初からやり直します．上部のツールバーの「Restart」ボタン（図 3.18）をクリックすると，「Restart」画面（図 3.19）が表示されます．設定を何も変更せず「OK」ボタンをクリックすると，時刻および結果が初期化されます．この状態でツールバーの「Run -All」ボタン（図 3.18）をクリックすると，変更後のモジュールの論理シミュレーションが実行されます．

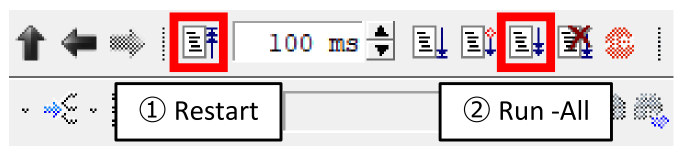


図 3.18　「Restart」ボタンと「Run -All」ボタン

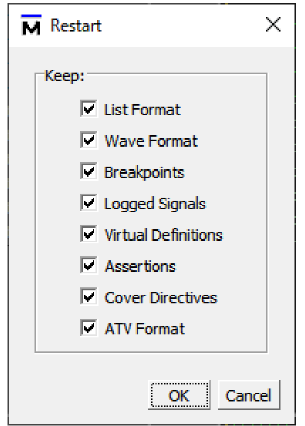


図 3.19　Restart 画面

論理シミュレーションの結果を図 3.20 に示します．出力信号が期待どおりになっていたかを示す matched に注目すると，ビット 2〜ビット 5（1000 の位〜100000 の位と対応）は常に 1 となっており，期待どおりに信号が出力されていたことが分かります．一方で，ビット 0（1 の位）およびビット 1（10 の位）は時折 0 となっており，その時間では出力信号が正しくありません．例えば，テストケース 3（表示したい数値が 23）では，10 の位に 1000000（「0」の点灯パターン），1 の位に 1111000（「7」の点灯パターン）が出力されており，期待値（10 の位に「2」，1 の位に「3」）とは異なります．この結果から，1 の位および 10 の位について，数字の取り出しもしくはデコード処理に問題があると推測できます．

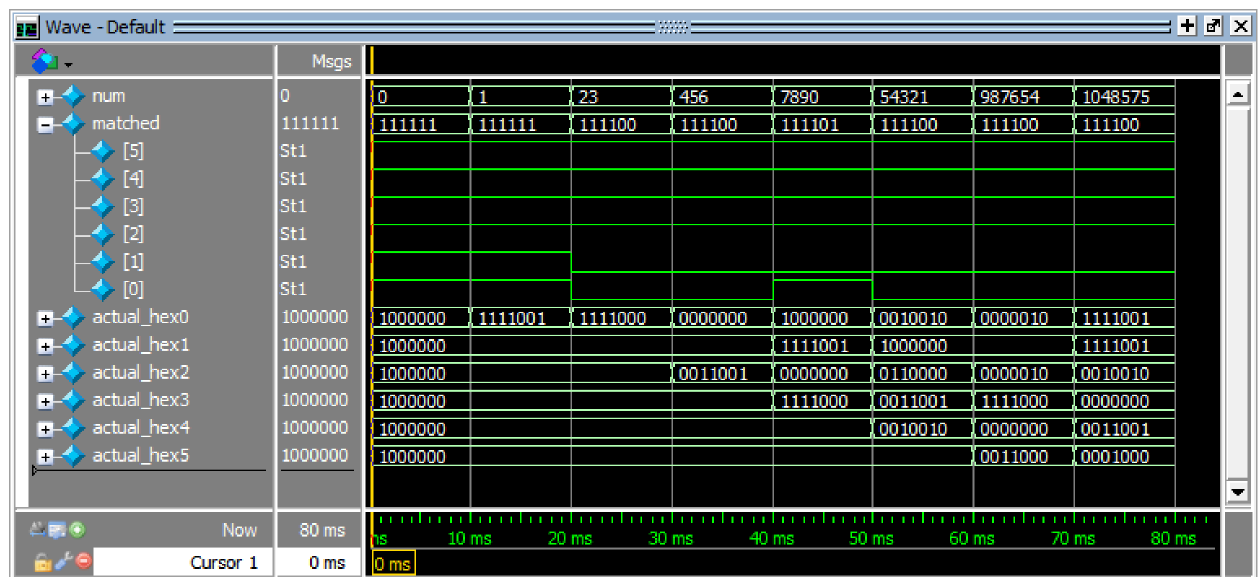


図 3.20　モジュール実装後の出力波形

モジュールの修正

実は，リスト 3.3 の Verilog コードには誤りがありました．その場所は，105 行目の

```
assign rest_100 = remainder_100[3:0];
```

です．rest_100 は，100 の位の数を取り出した後の残りの数が設定される信号線ですが，この記述では remainder_100（100 で割った余り）のうち 4 ビット分しか使用していません．remainder_100 がとり得る範囲は 0〜99 であり，その表現には 7 ビットが必要です．そのため，上位 3 ビットが必要となる，数値の下 2 桁が 15〜99 の場合に期待どおりの出力が得られなかったと考えられます．

問題を修正するには，remainder_100 から 7 ビット分を取り出すようにします．すなわち，105 行目を

```
assign rest_100 = remainder_100[6:0];
```

に変更します．

以上のように変更したら，再度論理シミュレーションによる動作確認を行います．ソースファイルの再コンパイル（メニューの「Compile」→「Compile All」）および論理シミュレーションのやり直し（ツールバーの「Restart」，「Run -All」）を行い，出力波形を確認してください．

ソースコード修正後の出力波形は図 3.21 のようになります．問題を修正したことで，すべての時間において matched の全ビットが 1 になりました．これは，すべてのテストケースについて，期待どおりの信号が出力されたことを示します．したがって，以上で 7 セグメント LED に対する基本的な数値表示処理を正しく実装できました．

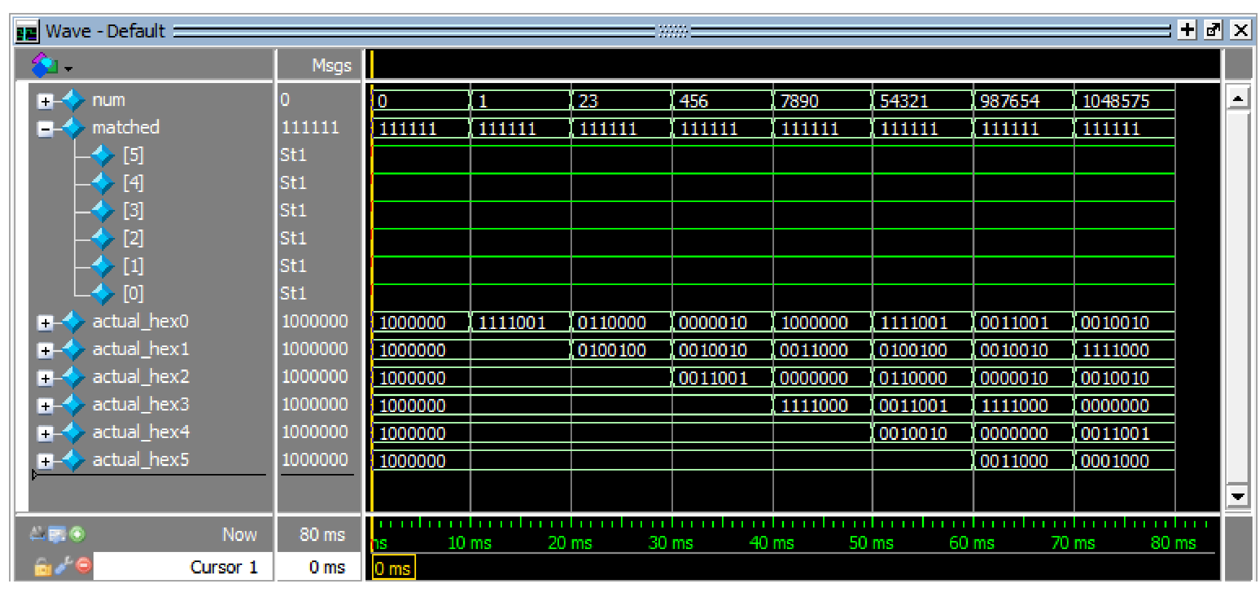


図 3.21　モジュール修正後の出力波形

3.1.4　ゼロ埋め制御機能の実装

基本的な数値表示処理を実装できたので，続いて先頭の 0 の表示の有無を制御する機能を追加します．

テストベンチの追加

この機能追加でも，最初にテストベンチを作成します．先頭の 0 を非表示にした場合のテストケースを表 3.5 に示します．表 3.4 と比較すると，zero_suppress が 1 である点，小さな数値を指定した際に先頭の 0 が消灯されることを期待している点が異なります．

表 3.5　先頭の 0 の非表示に関するテストケース．hex0〜hex5 列は，その桁に表示されるべき字を示す．

番号	num	zero_suppress	hex5	hex4	hex3	hex2	hex1	hex0
1	0	1	消灯	消灯	消灯	消灯	消灯	0
2	1	1	消灯	消灯	消灯	消灯	消灯	1
3	23	1	消灯	消灯	消灯	消灯	2	3
4	456	1	消灯	消灯	消灯	4	5	6
5	7890	1	消灯	消灯	7	8	9	0
6	54321	1	消灯	5	4	3	2	1
7	987654	1	9	8	7	6	5	4
8	1048575	1	A	4	8	5	7	5

以上のテストケースが含まれるテストベンチファイル seven_seg_decoder_zero_suppress_test.v（リスト 3.4）を配布します．このファイルを D:\DE1\sim\seven_seg\ 内にコピーしてください．その後，左側の「Project」タブ内をクリックし，続いてメニューから「Project」→「Add to Project」→「Existing File...」を選択してプロジェクトに追加します．

リスト 3.4　先頭の 0 の非表示に関するテストベンチ（seven_seg_decoder_zero_suppress_test.v）

```
1  `timescale 1 ms / 1 ms
2  
3  // 先頭の0の非表示に関するテストベンチの最上位階層
4  // テストベンチなので、入出力ポートはない
5  module seven_seg_decoder_zero_suppress_test;
6  
7  // 経過時間の単位
8  parameter STEP = 10;
9  
10 // 点灯パターン
11 parameter PATTERN_0 = 7'b100_0000;
12 parameter PATTERN_1 = 7'b111_1001;
13 parameter PATTERN_2 = 7'b010_0100;
14 parameter PATTERN_3 = 7'b011_0000;
15 parameter PATTERN_4 = 7'b001_1001;
16 parameter PATTERN_5 = 7'b001_0010;
17 parameter PATTERN_6 = 7'b000_0010;
18 parameter PATTERN_7 = 7'b111_1000;
19 parameter PATTERN_8 = 7'b000_0000;
20 parameter PATTERN_9 = 7'b001_1000;
21 parameter PATTERN_A = 7'b000_1000;
22 
23 // 全消灯
24 parameter PATTERN_OFF = 7'b111_1111;
25 
```

```
26 // 実際に出力された信号
27 wire [6:0] actual_hex0;
28 wire [6:0] actual_hex1;
29 wire [6:0] actual_hex2;
30 wire [6:0] actual_hex3;
31 wire [6:0] actual_hex4;
32 wire [6:0] actual_hex5;
33
34 // 7セグメントLEDに表示する数値
35 reg [19:0] num;
36
37 // 出力信号の期待値
38 reg [6:0] expected_hex0;
39 reg [6:0] expected_hex1;
40 reg [6:0] expected_hex2;
41 reg [6:0] expected_hex3;
42 reg [6:0] expected_hex4;
43 reg [6:0] expected_hex5;
44
45 // 各桁で、実際の出力信号が期待値と一致していたか
46 // 0: 一致していなかった
47 // 1: 一致していた
48 wire [5:0] matched;
49
50 // 7セグメントLEDデコーダモジュールの実体
51 seven_seg_decoder d0 (
52   .num (num),
53   .zero_suppress(1'b1),
54   .hex0 (actual_hex0),
55   .hex1 (actual_hex1),
56   .hex2 (actual_hex2),
57   .hex3 (actual_hex3),
58   .hex4 (actual_hex4),
59   .hex5 (actual_hex5)
60 );
61
62 // 各桁について、出力信号が期待値と一致していたか調べる
63 assign matched[0] = (actual_hex0 == expected_hex0);
64 assign matched[1] = (actual_hex1 == expected_hex1);
65 assign matched[2] = (actual_hex2 == expected_hex2);
66 assign matched[3] = (actual_hex3 == expected_hex3);
67 assign matched[4] = (actual_hex4 == expected_hex4);
68 assign matched[5] = (actual_hex5 == expected_hex5);
69
70 initial begin
71   num <= 20'd0;
72   expected_hex0 <= PATTERN_0;
73   expected_hex1 <= PATTERN_OFF;
74   expected_hex2 <= PATTERN_OFF;
75   expected_hex3 <= PATTERN_OFF;
76   expected_hex4 <= PATTERN_OFF;
```

```
77      expected_hex5 <= PATTERN_OFF;
78
79      #STEP
80      num <= 20'd1;
81      expected_hex0 <= PATTERN_1;
82      expected_hex1 <= PATTERN_OFF;
83      expected_hex2 <= PATTERN_OFF;
84      expected_hex3 <= PATTERN_OFF;
85      expected_hex4 <= PATTERN_OFF;
86      expected_hex5 <= PATTERN_OFF;
87
88      #STEP
89      num <= 20'd23;
90      expected_hex0 <= PATTERN_3;
91      expected_hex1 <= PATTERN_2;
92      expected_hex2 <= PATTERN_OFF;
93      expected_hex3 <= PATTERN_OFF;
94      expected_hex4 <= PATTERN_OFF;
95      expected_hex5 <= PATTERN_OFF;
96
97      #STEP
98      num <= 20'd456;
99      expected_hex0 <= PATTERN_6;
100     expected_hex1 <= PATTERN_5;
101     expected_hex2 <= PATTERN_4;
102     expected_hex3 <= PATTERN_OFF;
103     expected_hex4 <= PATTERN_OFF;
104     expected_hex5 <= PATTERN_OFF;
105
106     #STEP
107     num <= 20'd7890;
108     expected_hex0 <= PATTERN_0;
109     expected_hex1 <= PATTERN_9;
110     expected_hex2 <= PATTERN_8;
111     expected_hex3 <= PATTERN_7;
112     expected_hex4 <= PATTERN_OFF;
113     expected_hex5 <= PATTERN_OFF;
114
115     #STEP
116     num <= 20'd54321;
117     expected_hex0 <= PATTERN_1;
118     expected_hex1 <= PATTERN_2;
119     expected_hex2 <= PATTERN_3;
120     expected_hex3 <= PATTERN_4;
121     expected_hex4 <= PATTERN_5;
122     expected_hex5 <= PATTERN_OFF;
123
124     #STEP
125     num <= 20'd987654;
126     expected_hex0 <= PATTERN_4;
127     expected_hex1 <= PATTERN_5;
```

```
128    expected_hex2 <= PATTERN_6;
129    expected_hex3 <= PATTERN_7;
130    expected_hex4 <= PATTERN_8;
131    expected_hex5 <= PATTERN_9;
132
133    #STEP
134    num <= 20'd1048575;
135    expected_hex0 <= PATTERN_5;
136    expected_hex1 <= PATTERN_7;
137    expected_hex2 <= PATTERN_5;
138    expected_hex3 <= PATTERN_8;
139    expected_hex4 <= PATTERN_4;
140    expected_hex5 <= PATTERN_A;
141
142    #STEP
143    $stop;
144  end
145
146  endmodule
```

論理シミュレーション（実装前）

モジュールを実装する前に，一度論理シミュレーションを実行します．現段階では，表 3.5 の「消灯」の部分について，出力信号が期待値と異なるはずです．

まず，メニューから「Compile」→「Compile All」を選択して，プロジェクトに追加した seven_seg_decoder_zero_suppress_test.v をコンパイルします．コンパイルが成功すると，「Status」欄が「?」からチェックマークに変化します．

コンパイル後, メニューから「Simulate」→「Start Simulation...」を選択します．続いて表示される「Start Simulation」画面の「Design」タブにおいて，「work」「seven_seg_decoder_zero_suppress_test」を選択して，「OK」ボタンをクリックします（図 3.22）．

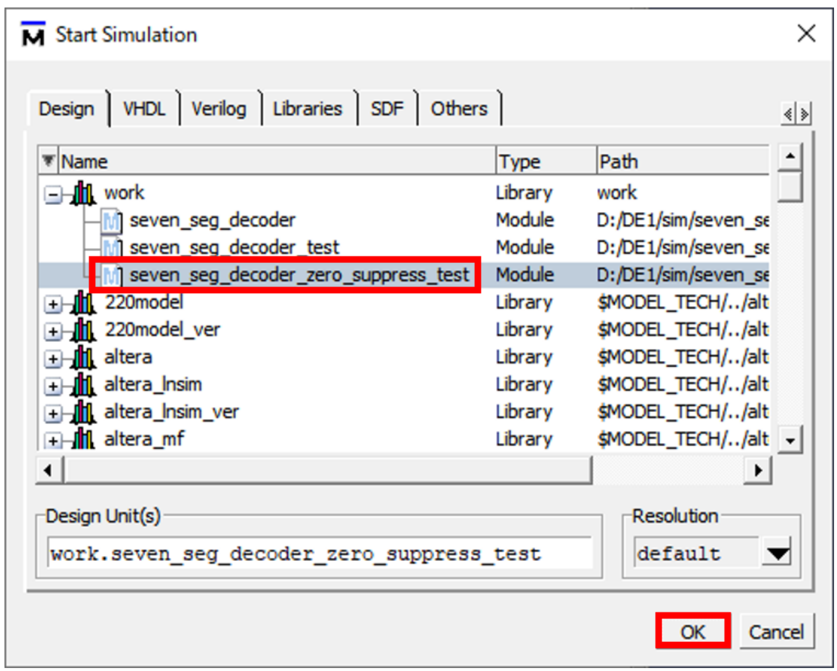


図 3.22　論理シミュレーションで使用するモジュールの選択

画面のレイアウトが変わったら，観察する信号を選択します．Objects ペインから Wave ペインに，以下の信号をドラッグ&ドロップします．

- num
- matched
- actual_hex0
- actual_hex1
- actual_hex2
- actual_hex3
- actual_hex4
- actual_hex5

信号を追加した直後は，信号名が長く表示されていて見づらいため，Wave ペイン下部の「Toggle leaf names <-> full names」ボタンをクリックして，短い名前で表示するようにします．また，7 セグメント LED に表示する数値を示す num を符号なし 10 進数として表示するため，num の上で右クリックしてメニューを表示し，「Radix」→「Unsigned」を選択します．以上を行った後の画面を，図 3.23 に示します．

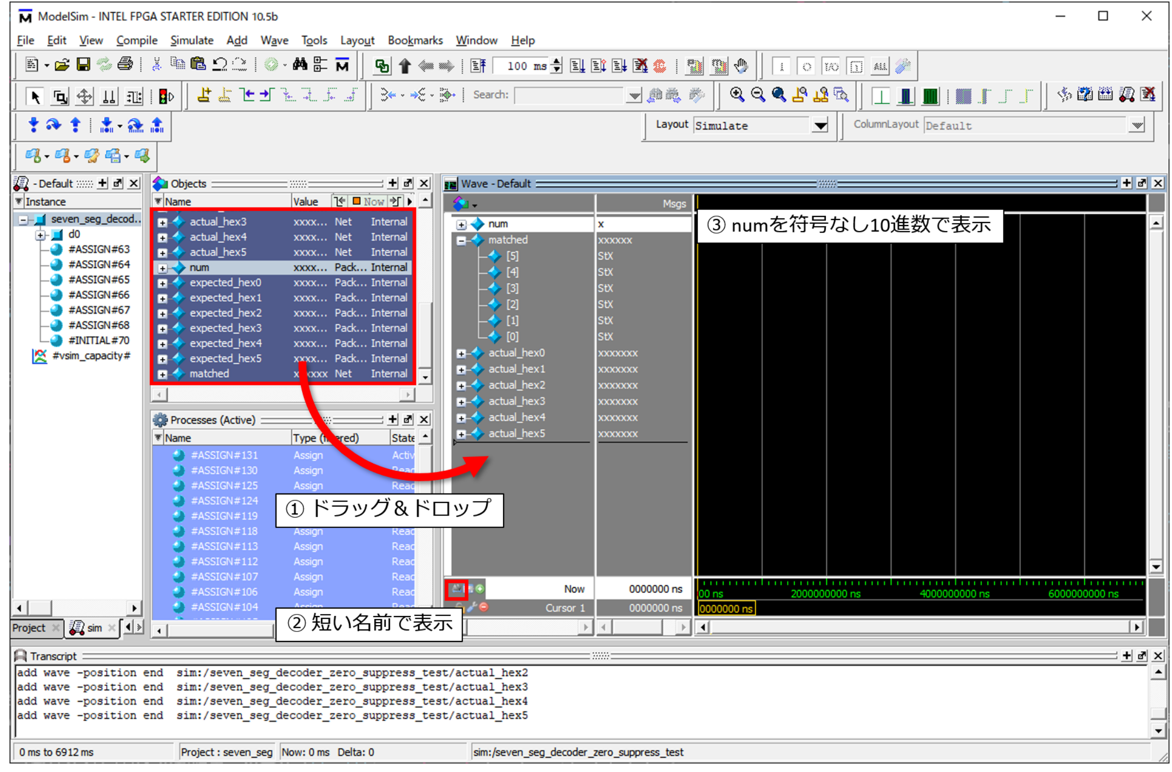


図 3.23　観察する信号の選択

以上で論理シミュレーションの設定が完了したので，上部のツールバーの「Run -All」ボタンをクリックして，シミュレーションを実行します．シミュレーションの終了後，Wave ペインにおいて「Wave」タブを選択し，ツールバーの「Zoom Full」ボタンをクリックすると，出力波形が表示されます．

出力波形を拡大して図 3.24 に示します．出力信号と期待値との一致を示す `matched` は，消灯を期待している箇所のみ `0`（期待値と異なる）となっています．今後，この箇所を 1 とする（出力信号が期待値と一致する）ことを目標として，Verilog コードを記述していきます．

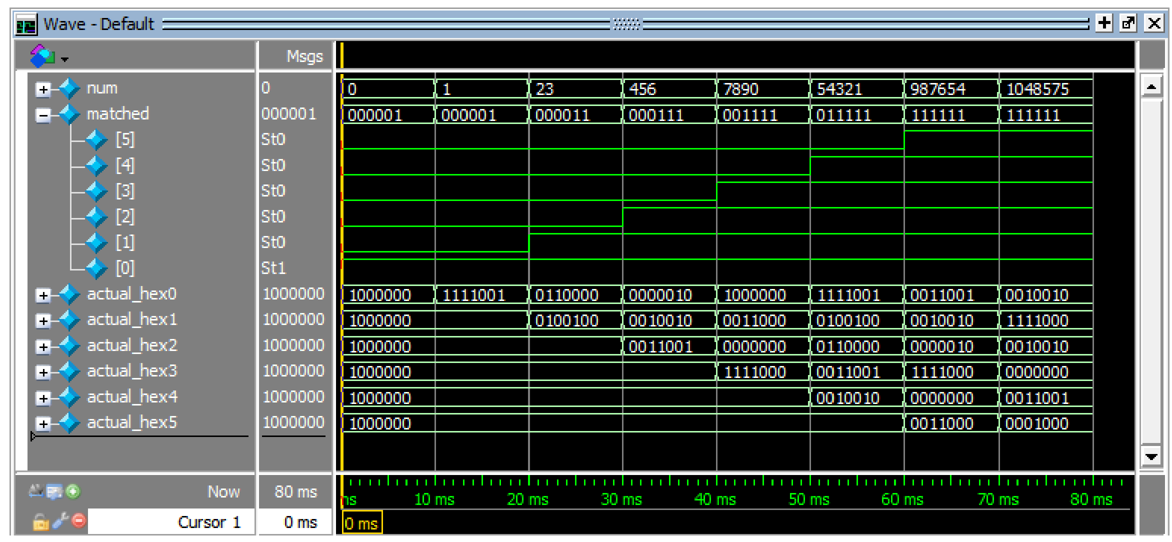


図 3.24　ゼロ埋め制御機能実装前の出力波形

ゼロ埋め制御機能の実装

論理シミュレーションの後，先頭の 0 の表示の有無を制御する機能を実装します．
以下のすべてを満たす場合に各桁を消灯させることで，要件が満たされます．

- 先頭の 0 を非表示にすることを示す信号 zero_suppress が 1 である．
- 1 の位（常時点灯）以外について，7 セグメント LED に表示する数値 num が，各桁の表示に必要な最小値未満である．
  - 10 の位：10 未満
  - 100 の位：100 未満
  - 1000 の位：1000 未満
  - 10000 の位：10000 未満
  - 100000 の位：100000 未満

この処理には共通点が多いため，以下の関数 decode_or_turn_off にまとめます．

```
// 数値を点灯させるか、消灯させる
function [6:0] decode_or_turn_off;
  // 一桁の数
  input [3:0] digit;
  // 先頭の0を非表示にするか
  input zero_suppress;
  // 7セグメントLEDに表示する数値
  input [19:0] num;
  // 表示/非表示の閾値
  input [19:0] threshold;

  if (zero_suppress) begin
    // 先頭の0を非表示にする場合
    decode_or_turn_off = (num < threshold) ? PATTERN_OFF : decode(digit);
  end else begin
    // 先頭の0を表示する場合
    decode_or_turn_off = decode(digit);
  end
endfunction
```

また，末尾の代入文で作成した関数を呼び出すようにします．

```
// 各桁の数と対応する点灯パターンを出力する
assign hex0 = decode(num_1);
assign hex1 = decode_or_turn_off(num_10, zero_suppress, num, 10);
assign hex2 = decode_or_turn_off(num_100, zero_suppress, num, 100);
assign hex3 = decode_or_turn_off(num_1000, zero_suppress, num, 1000);
assign hex4 = decode_or_turn_off(num_10000, zero_suppress, num, 10000);
assign hex5 = decode_or_turn_off(num_100000, zero_suppress, num, 100000);
```

以上の変更を反映した seven_seg_decoder.v を，リスト 3.5 に示します．

リスト 3.5　7 セグメント LED への数値表示処理の完成版（seven_seg_decoder.v）

```
1  // 7セグメントLEDへの数値表示処理を担当するモジュール
2  module seven_seg_decoder(
3    // 7セグメントLEDに表示する数値
4    input [19:0] num,
5    // 先頭の0を非表示にするか
6    input zero_suppress,
7
8    // HEX0（1の位）に出力する信号
9    output [6:0] hex0,
10   // HEX1（10の位）に出力する信号
11   output [6:0] hex1,
12   // HEX2（100の位）に出力する信号
13   output [6:0] hex2,
14   // HEX3（1000の位）に出力する信号
15   output [6:0] hex3,
16   // HEX4（10000の位）に出力する信号
17   output [6:0] hex4,
18   // HEX5（100000の位）に出力する信号
19   output [6:0] hex5
20 );
21
22 // 点灯パターン
23 parameter PATTERN_0 = 7'b100_0000;
24 parameter PATTERN_1 = 7'b111_1001;
25 parameter PATTERN_2 = 7'b010_0100;
26 parameter PATTERN_3 = 7'b011_0000;
27 parameter PATTERN_4 = 7'b001_1001;
28 parameter PATTERN_5 = 7'b001_0010;
29 parameter PATTERN_6 = 7'b000_0010;
30 parameter PATTERN_7 = 7'b111_1000;
31 parameter PATTERN_8 = 7'b000_0000;
32 parameter PATTERN_9 = 7'b001_1000;
33 parameter PATTERN_A = 7'b000_1000;
34
35 // 全消灯
36 parameter PATTERN_OFF = 7'b111_1111;
37
38 // 数値に対応する点灯パターンを返す
39 function [6:0] decode;
40   // 数値
41   input [3:0] n;
42
43   case (n)
44     4'd0:    decode = PATTERN_0;
45     4'd1:    decode = PATTERN_1;
46     4'd2:    decode = PATTERN_2;
47     4'd3:    decode = PATTERN_3;
48     4'd4:    decode = PATTERN_4;
49     4'd5:    decode = PATTERN_5;
50     4'd6:    decode = PATTERN_6;
```

```
51       4'd7:     decode = PATTERN_7;
52       4'd8:     decode = PATTERN_8;
53       4'd9:     decode = PATTERN_9;
54       default: decode = PATTERN_A;
55     endcase
56   endfunction
57
58   // 数値を点灯させるか、消灯させる
59   function [6:0] decode_or_turn_off;
60     // 一桁の数
61     input [3:0] digit;
62     // 先頭の0を非表示にするか
63     input zero_suppress;
64     // 7セグメントLEDに表示する数値
65     input [19:0] num;
66     // 表示/非表示の閾値
67     input [19:0] threshold;
68
69     if (zero_suppress) begin
70       // 先頭の0を非表示にする場合
71       decode_or_turn_off = (num < threshold) ? PATTERN_OFF : decode(digit);
72     end else begin
73       // 先頭の0を表示する場合
74       decode_or_turn_off = decode(digit);
75     end
76   endfunction
77
78   // 数値を100000で割った商
79   wire [19:0] quotient_100000;
80   // 数値を100000で割った余り
81   wire [19:0] remainder_100000;
82   // 100000の位の数
83   wire [3:0]  num_100000;
84   // 100000の位の数を取り出した後の残り
85   wire [16:0] rest_100000;
86
87   // 数値を10000で割った商
88   wire [16:0] quotient_10000;
89   // 数値を10000で割った余り
90   wire [16:0] remainder_10000;
91   // 10000の位の数
92   wire [3:0]  num_10000;
93   // 10000の位の数を取り出した後の残り
94   wire [13:0] rest_10000;
95
96   // 数値を1000で割った商
97   wire [13:0] quotient_1000;
98   // 数値を1000で割った余り
99   wire [13:0] remainder_1000;
100  // 1000の位の数
101  wire [3:0] num_1000;
```

```
102 // 1000の位の数を取り出した後の残り
103 wire [9:0] rest_1000;
104
105 // 数値を100で割った商
106 wire [9:0] quotient_100;
107 // 数値を100で割った余り
108 wire [9:0] remainder_100;
109 // 100の位の数
110 wire [3:0] num_100;
111 // 100の位の数を取り出した後の残り
112 wire [6:0] rest_100;
113
114 // 数値を10で割った商
115 wire [6:0] quotient_10;
116 // 数値を10で割った余り
117 wire [6:0] remainder_10;
118 // 10の位の数
119 wire [3:0] num_10;
120 // 1の位の数
121 wire [3:0] num_1;
122
123 // 100000の位の数を取り出す
124 assign quotient_100000  = num / 17'd100000;
125 assign remainder_100000 = num % 17'd100000;
126 assign num_100000  = quotient_100000[3:0];
127 assign rest_100000 = remainder_100000[16:0];
128
129 // 10000の位の数を取り出す
130 assign quotient_10000  = rest_100000 / 14'd10000;
131 assign remainder_10000 = rest_100000 % 14'd10000;
132 assign num_10000  = quotient_10000[3:0];
133 assign rest_10000 = remainder_10000[13:0];
134
135 // 1000の位の数を取り出す
136 assign quotient_1000  = rest_10000 / 10'd1000;
137 assign remainder_1000 = rest_10000 % 10'd1000;
138 assign num_1000  = quotient_1000[3:0];
139 assign rest_1000 = remainder_1000[9:0];
140
141 // 100の位の数を取り出す
142 assign quotient_100  = rest_1000 / 7'd100;
143 assign remainder_100 = rest_1000 % 7'd100;
144 assign num_100  = quotient_100[3:0];
145 assign rest_100 = remainder_100[6:0];
146
147 // 10の位、1の位の数を取り出す
148 assign quotient_10  = rest_100 / 4'd10;
149 assign remainder_10 = rest_100 % 4'd10;
150 assign num_10 = quotient_10[3:0];
151 assign num_1  = remainder_10[3:0];
152
```

```
153 // 各桁の数と対応する点灯パターンを出力する
154 assign hex0 = decode(num_1);
155 assign hex1 = decode_or_turn_off(num_10, zero_suppress, num, 10);
156 assign hex2 = decode_or_turn_off(num_100, zero_suppress, num, 100);
157 assign hex3 = decode_or_turn_off(num_1000, zero_suppress, num, 1000);
158 assign hex4 = decode_or_turn_off(num_10000, zero_suppress, num, 10000);
159 assign hex5 = decode_or_turn_off(num_100000, zero_suppress, num, 100000);
160
161 endmodule
```

論理シミュレーション（実装後）

ゼロ埋め制御機能の実装後，論理シミュレーションによる動作確認を行います．ソースファイルの再コンパイル（メニューの「Compile」→「Compile All」）および論理シミュレーションのやり直し（ツールバーの「Restart」,「Run -All」）を行い，出力波形を確認してください．

機能実装後の出力波形を図 3.25 に示します．消灯を期待した部分も含めて，すべての時間において matched の全ビットが 1 になっており，出力信号が期待値どおりであることが分かります．念のため，通常の数値表示処理の論理シミュレーション（seven_seg_decoder_test）も実行して，そちらでも出力信号が期待値どおりとなる（すなわち，追加した機能によって影響されない）ことを確認してください．以上で 7 セグメント LED への数値表示処理をすべて実装できました．

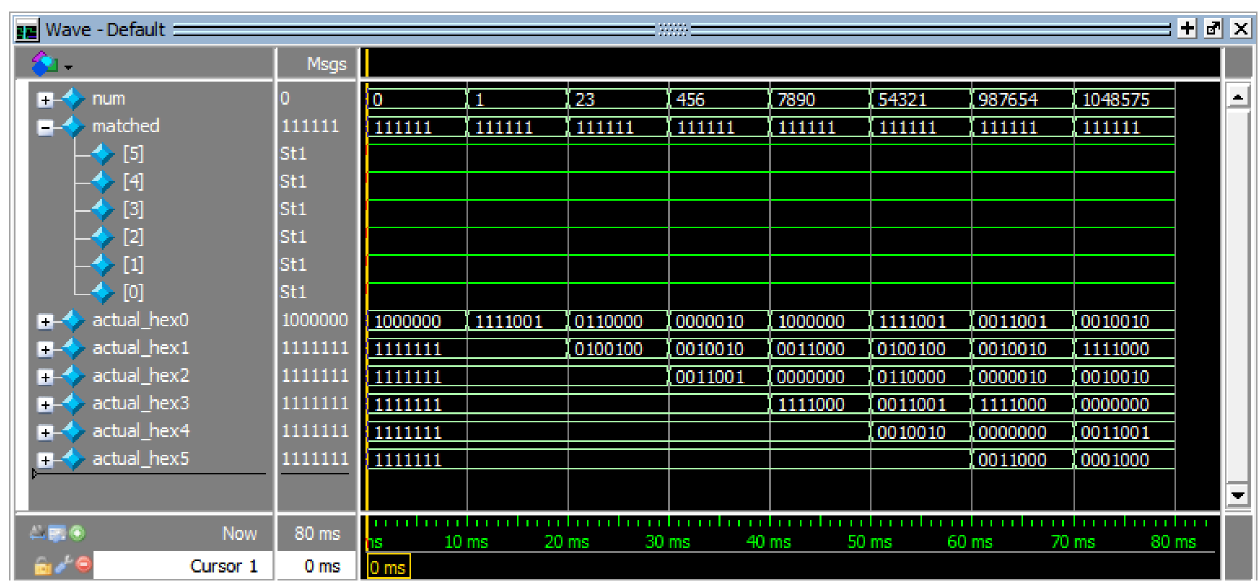


図 3.25　ゼロ埋め制御機能実装後の出力波形

3.1.5　Nios II からの利用

この項では，実装した 7 セグメント LED への数値表示処理を Nios II から利用する方法について解説します．

マイコンシステムの構成

マイコンシステムを構成する際には，数値レジスタ用およびゼロ埋め制御レジスタ用の PIO を追加します．設定内容を表 3.6 に示します．

表 3.6　7 セグメント LED への数値表示処理用の PIO の設定

名称例	ビット幅	方向	用途
seven_seg_num	20	Output（出力）	数値レジスタ
seven_seg_zero_suppress	1	Output（出力）	ゼロ埋め制御レジスタ

Verilog コードの記述

Verilog コードにおいて，wire を介して seven_seg_decoder モジュールとマイコンシステムを接続します．記述方法の概要をリスト 3.6 に示します．

リスト 3.6　seven_seg_decoder モジュールとマイコンシステムの接続

```
// 7セグメントLEDへの数値表示処理：数値レジスタ
wire [19:0] seven_seg_num;
// 7セグメントLEDへの数値表示処理：ゼロ埋め制御レジスタ
wire seven_seg_zero_suppress;

// 7セグメントLEDへの数値表示処理
seven_seg_decoder d0 (
  .num           (seven_seg_num),
  .zero_suppress (seven_seg_zero_suppress),

  .hex0          (HEX0),
  // ...
  .hex5          (HEX5)
);

// Nios IIマイコンシステム
nios2_system u0 (
  // ...

  .seven_seg_num_export           (seven_seg_num),
  .seven_seg_zero_suppress_export (seven_seg_zero_suppress),

  // ...
);
```

C 言語コードの記述

C 言語の組込みプログラムからは，IOWR_ALTERA_AVALON_PIO_DATA() マクロを使用して各レジスタに値を書き込むことで，7 セグメント LED を制御できます．記述例をリスト 3.7 に示します．

リスト 3.7　C 言語による 7 セグメント LED 制御の記述例

```
#include "altera_avalon_pio_regs.h"

// 7セグメントLEDに「123456」を表示する
IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_NUM_BASE , 123456);

// 先頭のゼロを表示するように設定する
IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_ZERO_SUPPRESS_BASE, 0);

// 先頭のゼロを表示しないように設定する
IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_ZERO_SUPPRESS_BASE, 1);
```

3.2　ロジックアナライザを活用したチャタリング除去回路の設計

この節では，FPGA にロジックアナライザを組み込んで内部信号を観察する方法について学習します．また，その観測結果を利用して，トグルスイッチのチャタリング除去回路を設計します．

3.2.1　ロジックアナライザ Signal Tap の概要

ロジックアナライザはデジタル回路のデバッグ用の計測器で，オシロスコープと同様に波形を観測できます．オシロスコープと比べると，以下に示す特徴があります [8].

- 扱う数値が 0 か 1 に限られる（多ビット信号は 10 進数，16 進数等で表示することも可能）.
- チャンネル数が多い.
- 複雑なトリガ条件（計測開始・終了条件）を設定できる.

Intel 社の FPGA には，Signal Tap というロジックアナライザ機能を組み込むことができます．Signal Tap は Quartus Prime に統合されており，特別な装置を用意することなく入出力端子の電圧や内部信号を観察できます．Signal Tap を使用する際は，画面での測定条件の設定，および Signal Tap 用の回路も含めた論理合成が必要です.

Signal Tap を使用した信号測定の手順は，以下のとおりです.

1. 計測する信号を指定する.
2. サンプリングクロックとして使用する信号と記録するデータ数を指定する.
3. トリガソースとトリガ条件（エッジ等）を設定する.
4. 測定用の回路と被測定回路とを一緒に論理合成する.
5. トリガ発生を待ち，信号を記録する.
6. 取り込んだデータをグラフ化して表示する.

3.2.2　実習の概要

今回の実習では，Signal Tap を使用して，図 3.26 に示す追加基板上のトグルスイッチから出力された信号を観察します.

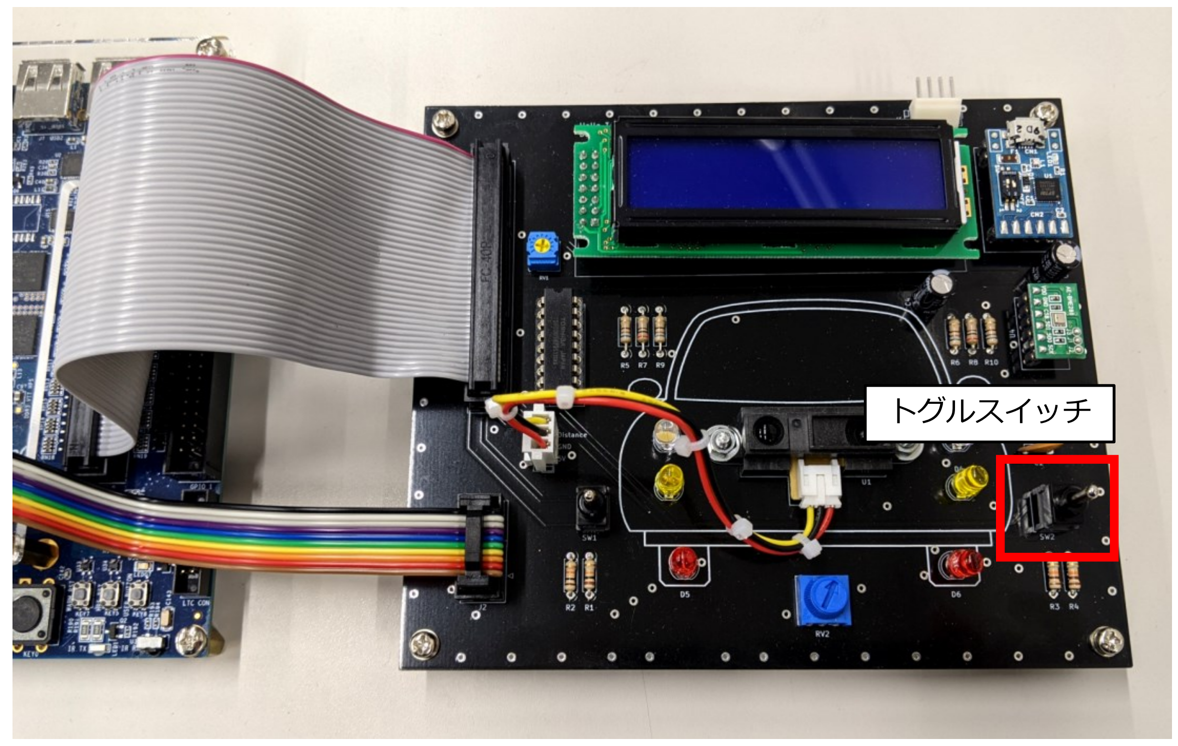


図 3.26　追加基板上のトグルスイッチ

トグルスイッチ周辺の回路図を図 3.27 に示します．このトグルスイッチには 3 つの位置があり，2 つの端子から状態と対応した信号を出力します．レバーが中央にあるときは，プルダウン抵抗の働きにより，両端子ともに L レベルを出力します．レバーを上または下に傾けると，傾けた方向と反対側の端子が共通端子と接続され，H レベルを出力します．チャタリング除去回路は設けられていないため，Signal Tap で出力信号を観察すると，レバーを動かした際に出力信号のチャタリング（図 3.28）を観察できます．

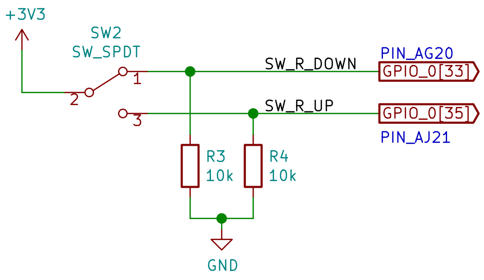


図 3.27　トグルスイッチ周辺の回路図

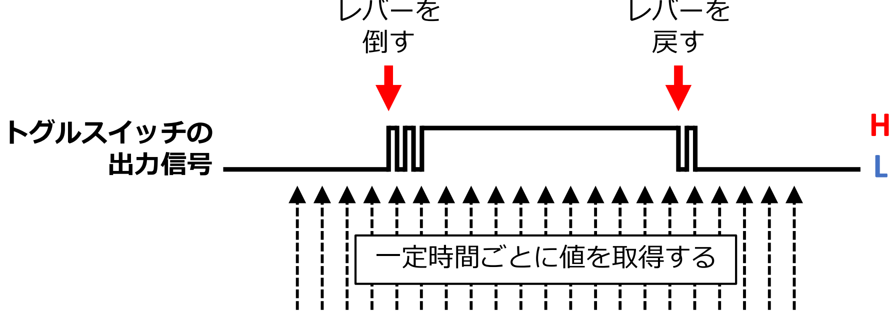


図 3.28　トグルスイッチの出力信号のチャタリング

トグルスイッチの出力信号の観察後は，その結果を基にして，チャタリング除去回路を設計します．最後に，設計したチャタリング除去回路を通した後のトグルスイッチの出力信号を Signal Tap で観察して，安定した出力が得られることを確認します．

3.2.3　波形観察用プロジェクトの作成

最初に，Signal Tap を用いて波形を観察する際に使用する Quartus Prime のプロジェクトを作成します．

プロジェクトの作成

Quartus Prime を起動して，表 3.7 の設定でプロジェクトを作成してください．プロジェクト作成の手順については，2.2.2 項（p. 9～）を参照してください．

表 3.7　波形観察用プロジェクト作成時の設定

項目	設定値
作業フォルダ（working directory）	`D:\DE1\sig_tap`
プロジェクト名（name of this project）	`sig_tap`
最上位階層名（name of the top-level design entity）	`sig_tap`
開発キット（Development Kit）	DE1-SoC Board

ハードウェアの設計

プロジェクトを作成したら，観察対象の信号やサンプリング用のクロックを含む回路を Verilog で記述します．

今回の観察では，サンプリング用クロックとして 100 kHz のクロックを用意します．これは，数百 μs から数 ms の時間で発生するチャタリング現象を観察しやすくするための工夫です．DE1-SoC に実装された 50 MHz のクロックそのままではサンプリング間隔が非常に短くなり，測定データ全体の確認が困難になります（記録できる時間よりも長い間チャタリングが発生する場合もあります）．

次に示すリスト 3.8 を記述して，`D:\DE1\sig_tap\sig_tap.v` として保存してください．

リスト 3.8　波形観察用ハードウェアの実装（`sig_tap.v`）

```
1  module sig_tap (
2    // 50 MHzクロック
3    input CLK,
4    // リセット信号（負論理）
5    input RESET_N,
6
7    // 右トグルスイッチ上側
8    input TOGGLE_SW_R_UP,
9    // 右トグルスイッチ下側
10   input TOGGLE_SW_R_DOWN,
11
```

```
12   // 100 kHzサンプリングクロック
13   output reg CLK_100KHZ
14 );
15
16 // 100 kHzクロックの最大カウント値
17 parameter CLK_100KHZ_MAX_COUNT = 8'd249;
18
19 // 100 kHzクロックのカウンタ（8'd0～8'd249）
20 reg [7:0] clk_100khz_count;
21
22 // 100 kHzクロック
23 always @(posedge CLK or negedge RESET_N) begin
24   if (!RESET_N) begin
25     clk_100khz_count <= 8'd0;
26     CLK_100KHZ <= 1'b0;
27   end else if (clk_100khz_count == CLK_100KHZ_MAX_COUNT) begin
28     clk_100khz_count <= 8'd0;
29     CLK_100KHZ <= ~CLK_100KHZ;
30   end else begin
31     clk_100khz_count <= clk_100khz_count + 8'd1;
32   end
33 end
34
35 endmodule
```

Verilog コードの記述後，左側にある「Tasks」ペインの「Analysis & Synthesis」（図 2.37，p. 28）をダブルクリックして，論理合成を行います．論理合成が成功したら，メニューから「Assignments」→「Pin Planner」を選択して Pin Planner を起動し，表 3.8 のようにピンアサインを行います．

表 3.8 波形観察用プロジェクトにおけるピンの割り当て

Node Name	Direction	Location	I/O Standard	割り当て先の機能
CLK	Input	PIN_AF14	3.3-V LVCMOS	50 MHz クロック
RESET_N	Input	PIN_AA14	3.3-V LVCMOS	ボタンスイッチ KEY0（負論理）
TOGGLE_SW_R_DOWN	Input	PIN_AG20	3.3-V LVCMOS	右トグルスイッチ下側
TOGGLE_SW_R_UP	Input	PIN_AJ21	3.3-V LVCMOS	右トグルスイッチ上側
CLK_100KHZ	Output	PIN_V16	3.3-V LVCMOS	LEDR0

ピンを割り当てた後は，「Tasks」ペインの「Compile Design」（図 2.41，p. 30）をダブルクリックして回路をコンパイルしてください．

3.2.4 Signal Tap を用いたチャタリング現象の観察

ハードウェアの設計後，Signal Tap を使用してトグルスイッチの出力信号を観察します．

Signal Tap の起動

メニューから「Tools」→「Signal Tap Logic Analyzer」(図 3.29) を選択します. 選択すると Signal Tap が起動し, 図 3.30 に示す画面が表示されます.

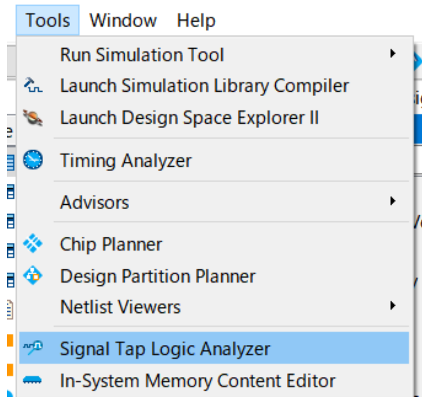


図 3.29　Signal Tap Logic Analyzer

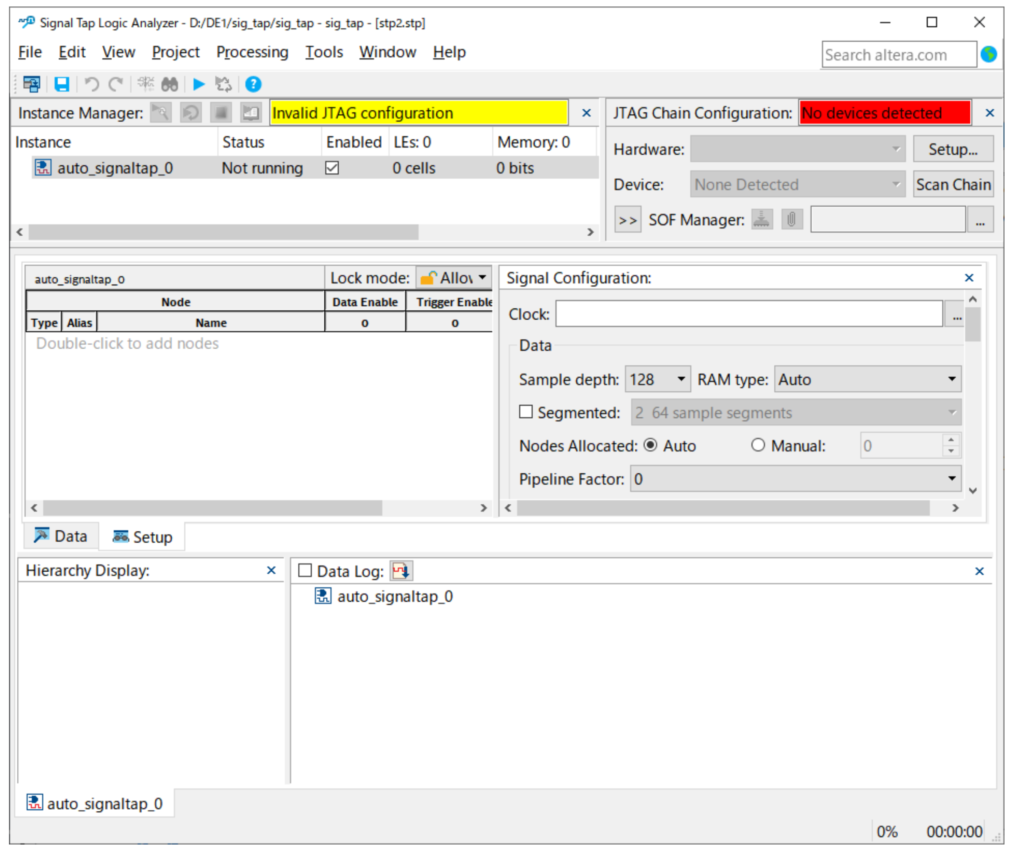


図 3.30　Signal Tap の画面

観察する信号の選択

最初に, Signal Tap を用いて観察する信号を選択します. 中央左側ペインの「Double-click to add nodes」と表示されている部分 (図 3.31) をダブルクリックします.

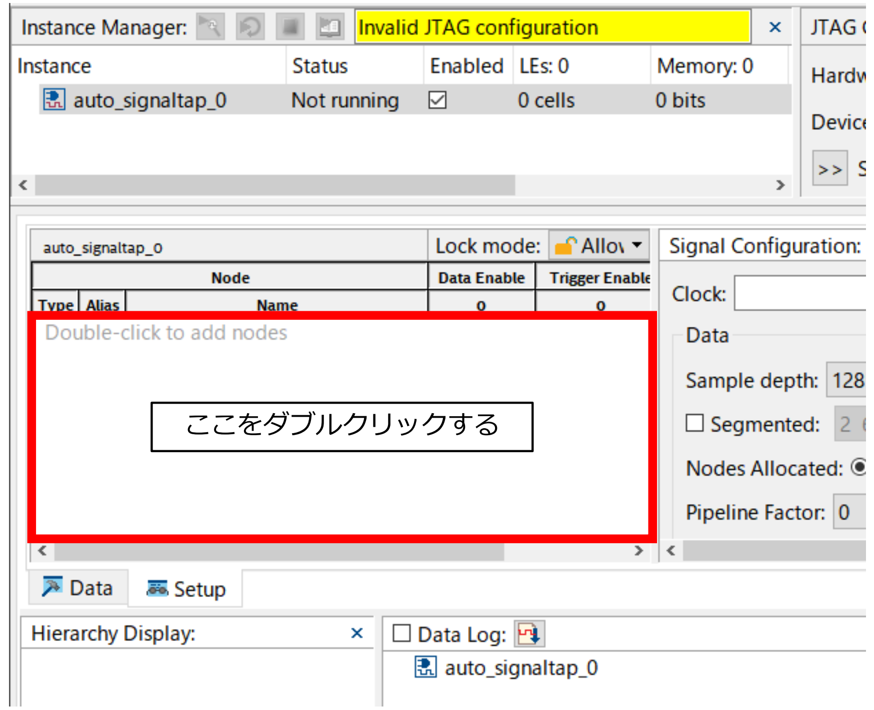


図 3.31　観察する信号の選択のためにダブルクリックする部分

ダブルクリックすると，観察したい信号を指定する「Node Finder」画面（図 3.32）が表示されます．ここでは観察する信号を選択しやすくするため，先にフィルタの設定を行います．画面右上の「∀」ボタンをクリックすると画面が広がり，「Options」欄が現れます．その中の「Filter」を「Signal Tap: pre-synthesis」に設定して，上部の「List」ボタンをクリックすると，Verilog で記述した信号が一覧表示されます．一覧表示された信号の中から以下の 2 つの信号を選択して「>」ボタンをクリックし，右側の「Nodes Found」欄に追加します．

- `TOGGLE_SW_R_DOWN`
- `TOGGLE_SW_R_UP`

その後「Insert」ボタンをクリックすると，「Nodes Found」欄に表示された信号が観察対象として追加されます（図 3.33）．観察対象の信号が追加されたことを確認後，「Close」ボタンをクリックして「Node Finder」画面を閉じます．

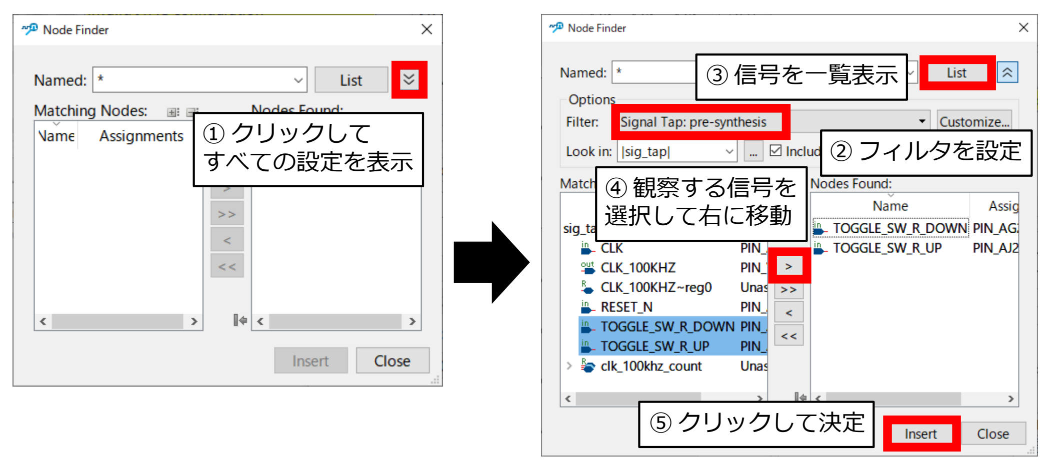


図 3.32　観察する信号の選択

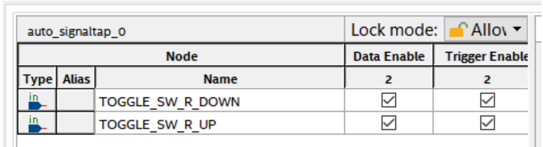


図 3.33　観察対象として追加された信号

サンプリングの設定

続いて，中央右側の「Signal Configuration」ペイン（図 3.34）において，信号のサンプリングについての設定を行います．「Clock」欄右側の「...」ボタンをクリックして，サンプリング時に使用するクロックを指定します．「Node Finder」画面が現れるので，信号指定時と同様にして `CLK_100KHZ` を指定します．また，「Sample Depth」（取得するサンプル数）を 4 K（40.96 ms 分）に設定します．

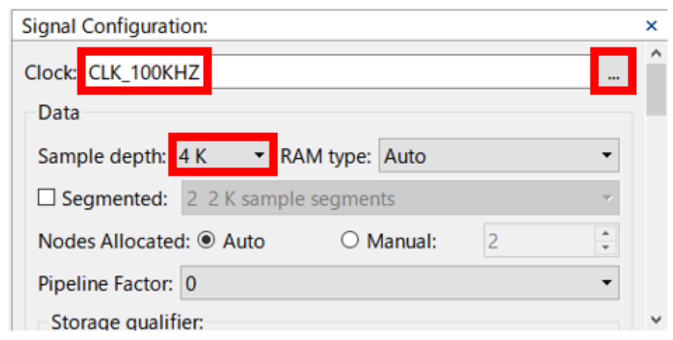


図 3.34　サンプリングの設定

トリガ条件の設定

今回は，トグルスイッチのレバーの位置の変化をトリガとして，出力信号を観察します．レバーの位置が変化する際は `TOGGLE_SW_R_UP` または `TOGGLE_SW_R_DOWN` が変化しますので，この変化をトリガ条件として設定します．

まず，「Trigger Conditions」列の見出しを「Basic OR」に設定します（図 3.35）．

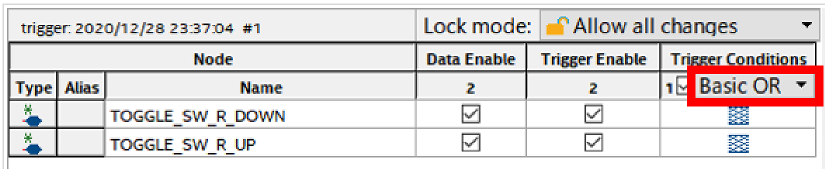


図 3.35　Basic OR

続いて，中央左側ペインの各行について，「Trigger Conditions」列で右クリックし，表示されたメニューから「Either Edge」（立上り，立下りのいずれか）を選択します（図 3.36）．

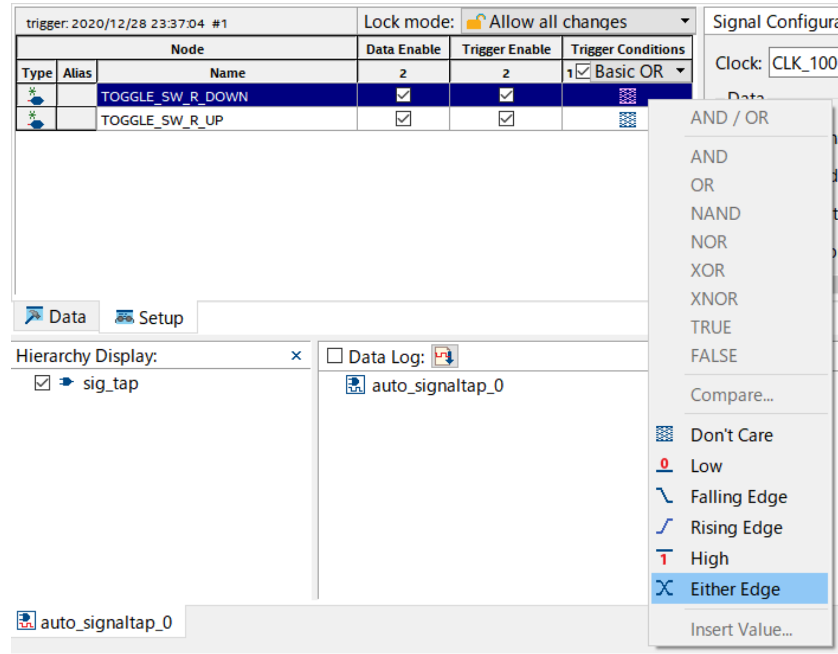


図 3.36　Either Edge（立上り，立下りのいずれか）

「Trigger Conditions」が図 3.37 のように設定されれば，トリガ条件の設定は完了です．

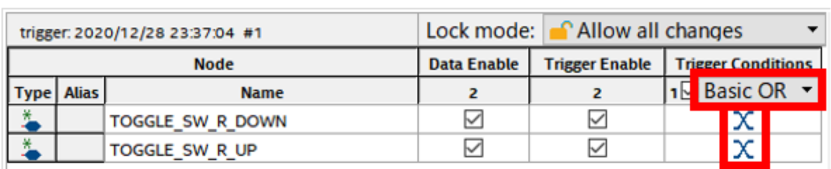


図 3.37　トリガ条件の設定

以上で，波形観察前の設定は完了です．

論理合成および FPGA へのプログラミング

設定後，Signal Tap と被測定回路とを合わせて論理合成する必要があります．ツールバーから「Start Compilation」（図 3.38）をクリックして，プロジェクトをコンパイルします．

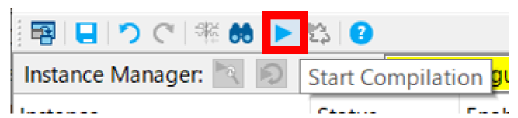
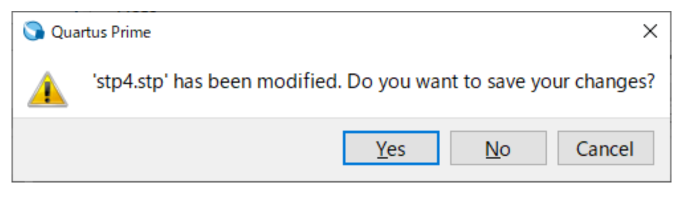
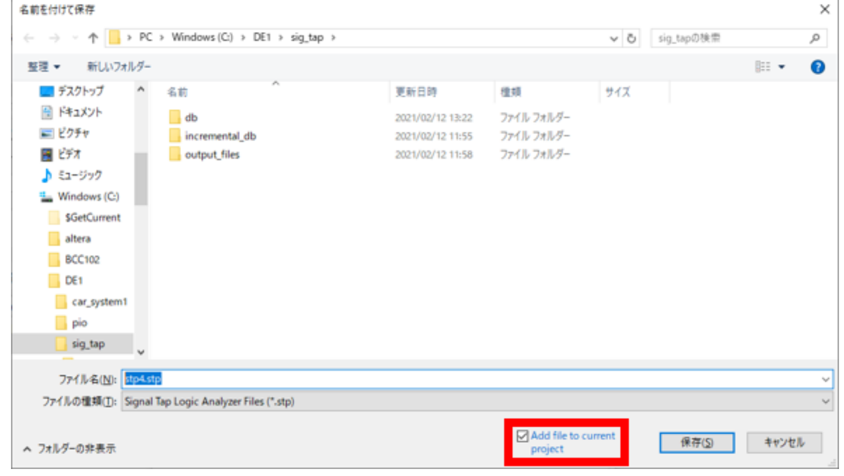


図 3.38　Start Compilation

初回は，図 3.39 のように，Signal Tap の設定ファイルの保存に関するダイアログがいくつか表示されますが，既定の設定のまま「Yes」や「保存」をクリックして次へ進み，保存とプロジェクトへの追加を行います．





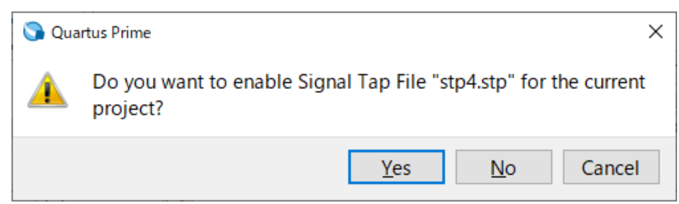


図 3.39　Signal Tap の設定ファイルの保存に関するダイアログ

コンパイルが終了したら，Quartus Prime の画面に切り替えてハードウェアデザインを FPGA に書き込みます．2.2.6 項（p. 31〜）の手順で，Programmer を起動して FPGA へのプログラミングを行ってください．プログラミングの完了後，再び Signal Tap の画面に戻ります．

ハードウェアの指定

Signal Tap の画面右上の「JTAG Chain Configuration」ペインに「No devices detected」（デバイスが検出されません）と表示される場合（図 3.40）は，信号の観察前に使用するハードウェアを指定する必要があります．

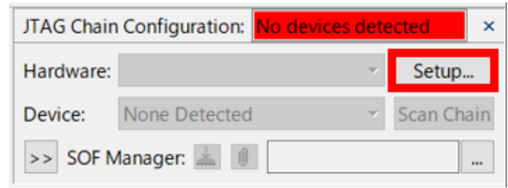


図 3.40　No devices detected（デバイスが検出されません）

このペインの「Setup」ボタンをクリックして，ハードウェア設定画面を表示します．ハードウェア設定画面では，「Currently selected hardware」を「DE-SoC」に設定します（図 3.41）．

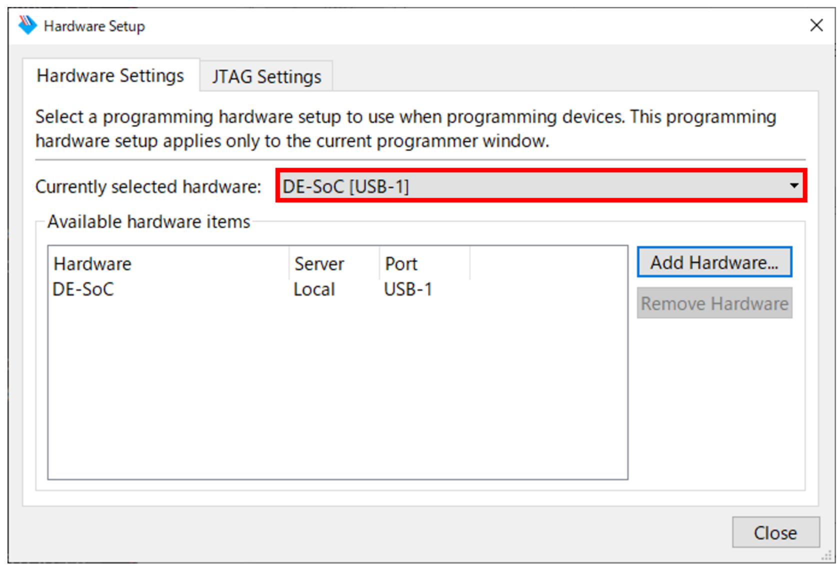


図 3.41　「Currently selected hardware」の設定

ハードウェアの指定後,「JTAG Ready」（図 3.42）と表示されるのを確認してください.

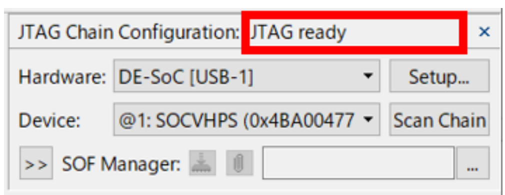


図 3.42　JTAG Ready

信号の観察

それでは，追加基板上のトグルスイッチのレバーを動かした際の信号の変化を観察していきましょう.

まず，追加基板上の右トグルスイッチのレバーを中央に戻します．続いて,「Instance Manager」ペインの状況欄に「Ready to acquire」（取得の準備完了）と表示されているのを確認してから,「auto_signaltap_0」を選択して「Run Analysis」（解析実行）ボタンをクリックします（図 3.43）.

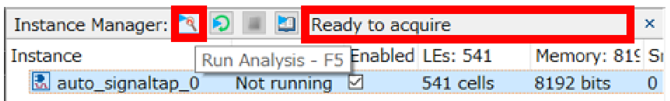


図 3.43　信号の取得開始

状況欄に緑色の背景で「Acquisition in progress」（取得中）と表示されたら（図 3.44），右トグルスイッチのレバーを上に動かします.

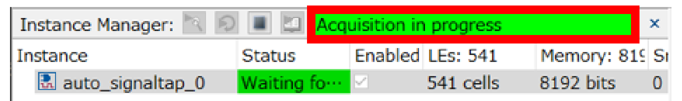


図 3.44　信号取得中

レバーを動かすと変化した信号が取り込まれ，図 3.45 のようにタイミングチャートが表示されます.

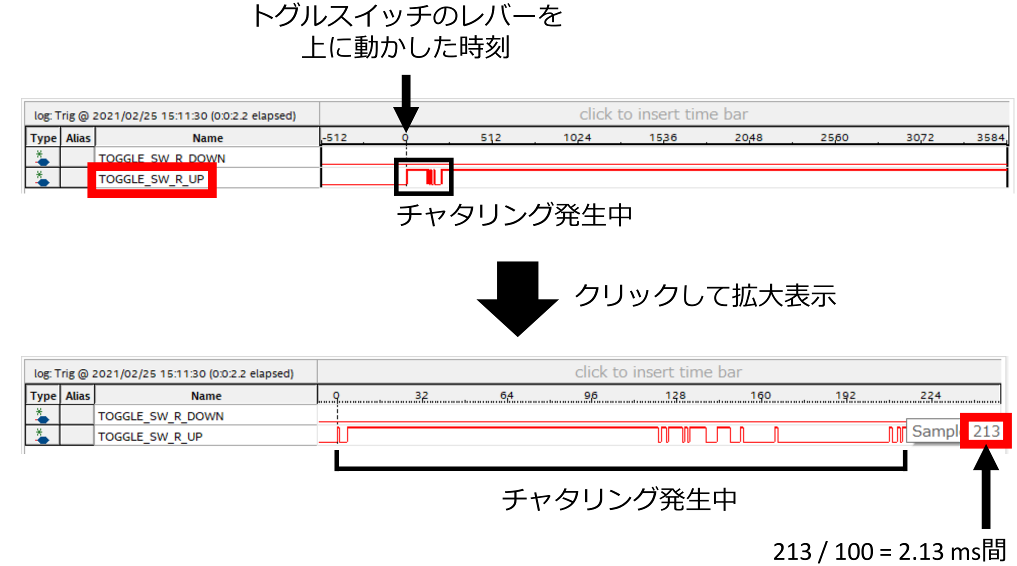


図 3.45　トグルスイッチのレバーを上に動かした際の信号の変化

図 3.45 より，レバーを上に動かしたとき TOGGLE_SW_R_UP が L から H に変化したこと，またその前後でチャタリングが発生したことが分かります．この操作を 5 回程度繰り返して，チャタリングの発生時間（チャタリングが発生していた時間の末尾のクロック数/100，ms 単位）の最大値を求めてください．同様に，右トグルスイッチのレバーを中央から下に動かしたときの TOGGLE_SW_R_DOWN の変化を 5 回程度観測して，チャタリングの発生時間 [ms] の最大値を求めてください．

レバー位置の変化	変化した信号	チャタリング発生クロック数	チャタリング発生時間
中央→上	TOGGLE_SW_R_UP		ms
中央→下	TOGGLE_SW_R_DOWN		ms

3.2.5　チャタリング除去回路の設計

この項では，チャタリング現象の観測結果を利用して，チャタリング除去回路を設計します．

シフトレジスタによるチャタリングの除去

今回は，シフトレジスタを利用して信号のチャタリングを除去します．

シフトレジスタによるチャタリング除去の原理を図 3.46 に示します．トグルスイッチのレバーの位置が変化してチャタリングが発生している間，出力信号は 0 と 1 との間を不規則に行き来します．その後チャタリングが落ち着くと，出力信号は 0 または 1 のままとなり安定します．この性質より，数ビットのシフトレジスタで信号を連続して取り込み，すべてのビットが 1 となったかを AND ゲートで調べることで，出力信号が 1 で安定したか判断することができます．

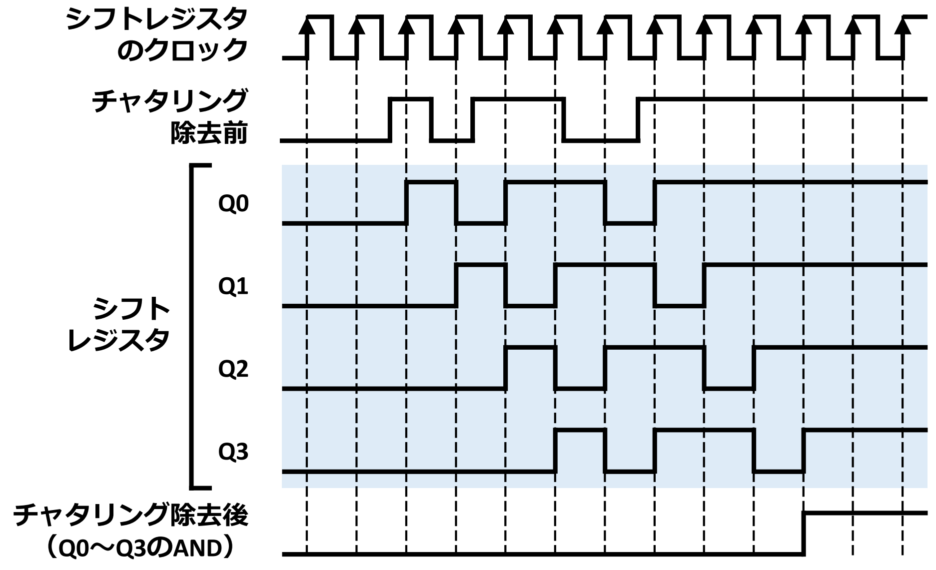


図 3.46　シフトレジスタによるチャタリング除去

前項の観察実験において，トグルスイッチの出力信号のチャタリングが数 ms 間にわたって発生することが確認できたかと思います．そのため，チャタリングが落ち着くまでの時間を 20 ms 程度と想定しておけば，概ね問題ないでしょう．そこで，今回は 200 Hz（5 ms 周期）クロックで 4 ビットのシフトレジスタに信号を連続して取り込み，出力信号が 1 で安定したか判断することにします．

チャタリング除去回路を含むモジュールの実装

以上のチャタリング除去回路を含めたトグルスイッチのモジュール `toggle_sw` を Verilog で実装します．このモジュールのインターフェースは，表 3.9 のように設定することとします．

表 3.9　`toggle_sw` モジュールのインターフェース

ポート名	方向	ビット幅	信号の役割	論理/値の定義
`clk`	入力	1	50 MHz クロック	
`reset_n`	入力	1	リセット	負論理
`toggle_sw_up`	入力	1	トグルスイッチ上側	正論理
`toggle_sw_down`	入力	1	トグルスイッチ下側	正論理
`out`	出力	2	出力値	`2'b00`：中央 `2'b01`：上側 `2'b10`：下側

次に示すリスト 3.9 を記述して `D:\DE1\sig_tap\toggle_sw.v` に保存し，プロジェクトに追加してください．

リスト 3.9　トグルスイッチモジュールの実装（`toggle_sw.v`）

```
1  // トグルスイッチ入力のモジュール
2  module toggle_sw (
3    // 50 MHzクロック
4    input clk,
```

```
5     // リセット (負論理)
6     input reset_n,
7
8     // トグルスイッチ上側
9     input toggle_sw_up,
10    // トグルスイッチ下側
11    input toggle_sw_down,
12
13    // 出力値
14    // * 2'b00: 中央
15    // * 2'b01: 上側
16    // * 2'b10: 下側
17    output [1:0] out
18  );
19
20  // 200 Hzクロックのカウンタ (18'd0-18'd249_999)
21  reg [17:0] clk_200hz_count;
22
23  // 200 Hzクロックの有効信号
24  wire clk_200hz_enable;
25  assign clk_200hz_enable = (clk_200hz_count == 18'd249_999);
26
27  // 読み取った値:上側
28  reg [3:0] read_values_up;
29  // 読み取った値:下側
30  reg [3:0] read_values_down;
31
32  // 200 Hzクロック
33  always @(posedge clk or negedge reset_n) begin
34    if (!reset_n) begin
35      clk_200hz_count <= 18'd0;
36    end else if (clk_200hz_enable) begin
37      clk_200hz_count <= 18'd0;
38    end else begin
39      clk_200hz_count <= clk_200hz_count + 18'd1;
40    end
41  end
42
43  // 5 ms周期またはリセット時に行う処理
44  always @(posedge clk or negedge reset_n) begin
45    if (!reset_n) begin
46      read_values_up <= 4'b0000;
47      read_values_down <= 4'b0000;
48    end else if (clk_200hz_enable) begin
49      read_values_up <= { read_values_up[2:0], toggle_sw_up };
50      read_values_down <= { read_values_down[2:0], toggle_sw_down };
51    end
52  end
53
54  // チャタリングを除去して、状態を示す値を返す
55  //
```

```
56 // 戻り値:
57 // * 2'b00: 中央
58 // * 2'b01: 上側
59 // * 2'b10: 下側
60 function [1:0] debounce;
61   input [3:0] read_values_up;
62   input [3:0] read_values_down;
63
64   case ({ read_values_up, read_values_down })
65     8'b1111_0000: debounce = 2'b01;
66     8'b0000_1111: debounce = 2'b10;
67     default:      debounce = 2'b00;
68   endcase
69 endfunction
70
71 assign out = debounce(read_values_up, read_values_down);
72
73 endmodule
```

モジュール toggle_sw を作成したら，これを最上位階層 sig_tap から呼び出します．ファイル sig_tap.v をリスト 3.10 のように書き換えてください．

リスト 3.10　波形観察用ハードウェア（toggle_sw モジュール使用）の実装（sig_tap.v）

```
1 module sig_tap (
2   // 50 MHzクロック
3   input CLK,
4   // リセット信号（負論理）
5   input RESET_N,
6
7   // 右トグルスイッチ上側
8   input TOGGLE_SW_R_UP,
9   // 右トグルスイッチ下側
10  input TOGGLE_SW_R_DOWN,
11
12  // 100 kHzサンプリングクロック
13  output reg CLK_100KHZ,
14
15  // チャタリング除去後のトグルスイッチの出力値
16  output [1:0] TOGGLE_SW_R_DEBOUNCED
17 );
18
19 // 100 kHzクロックの最大カウント値
20 parameter CLK_100KHZ_MAX_COUNT = 8'd249;
21
22 // 100 kHzクロックのカウンタ（8'd0～8'd249）
23 reg [7:0] clk_100khz_count;
24
25 // 100 kHzクロック
26 always @(posedge CLK or negedge RESET_N) begin
27   if (!RESET_N) begin
28     clk_100khz_count <= 8'd0;
```

```
29      CLK_100KHZ <= 1'b0;
30    end else if (clk_100khz_count == CLK_100KHZ_MAX_COUNT) begin
31      clk_100khz_count <= 8'd0;
32      CLK_100KHZ <= ~CLK_100KHZ;
33    end else begin
34      clk_100khz_count <= clk_100khz_count + 8'd1;
35    end
36  end
37
38  // トグルスイッチのチャタリング除去
39  toggle_sw toggle_sw_r(
40    .clk(CLK),
41    .reset_n(RESET_N),
42    .toggle_sw_up(TOGGLE_SW_R_UP),
43    .toggle_sw_down(TOGGLE_SW_R_DOWN),
44    .out(TOGGLE_SW_R_DEBOUNCED)
45  );
46
47  endmodule
```

Verilog コードの記述後，左側にある「Tasks」ペインの「Analysis & Synthesis」（図 2.37，p. 28）をダブルクリックして，論理合成を行います．論理合成が成功したら，メニューから「Assignments」→「Pin Planner」を選択して Pin Planner を起動し，表 3.10 のようにピンアサインを行います．

表 3.10　波形観察用プロジェクト（toggle_sw モジュール使用）におけるピンの割り当て

Node Name	Direction	Location	I/O Standard	割り当て先の機能
CLK	Input	PIN_AF14	3.3-V LVCMOS	50 MHz クロック
RESET_N	Input	PIN_AA14	3.3-V LVCMOS	ボタンスイッチ KEY0（負論理）
TOGGLE_SW_R_DOWN	Input	PIN_AG20	3.3-V LVCMOS	右トグルスイッチ下側
TOGGLE_SW_R_UP	Input	PIN_AJ21	3.3-V LVCMOS	右トグルスイッチ上側
CLK_100KHZ	Output	PIN_V16	3.3-V LVCMOS	LEDR0
TOGGLE_SW_R_DEBOUNCED[0]	Output	PIN_W16	3.3-V LVCMOS	LEDR1
TOGGLE_SW_R_DEBOUNCED[1]	Output	PIN_V17	3.3-V LVCMOS	LEDR2

ピンを割り当てた後は，「Tasks」ペインの「Compile Design」（図 2.41，p. 30）をダブルクリックして回路をコンパイルしてください．

論理シミュレーション

参考までに，チャタリング除去回路のテストベンチと論理シミュレーション結果を示します．

リスト 3.11 にテストベンチのソースコードを示します．

リスト 3.11　トグルスイッチのチャタリング除去回路のテストベンチ（toggle_sw_test.v）

```
1  `timescale 1 ns / 1 ns
2
3  // トグルスイッチのテストベンチの最上位階層
4  // テストベンチなので、入出力ポートはない
5  module toggle_sw_test;
```

```
6
7   // 経過時間の単位
8   parameter STEP = 10;
9
10  // クロックのシミュレーション用最大カウント値
11  parameter CLK_200HZ_MAX_COUNT = 18'd3;
12
13  reg clk;
14  reg reset_n;
15
16  reg toggle_sw_up;
17  reg toggle_sw_down;
18
19  wire [1:0] out;
20
21  // トグルスイッチのチャタリング除去回路の実体
22  toggle_sw #(
23    .CLK_200HZ_MAX_COUNT(CLK_200HZ_MAX_COUNT)
24  ) t0 (
25    .clk(clk),
26    .reset_n(reset_n),
27    .toggle_sw_up(toggle_sw_up),
28    .toggle_sw_down(toggle_sw_down),
29    .out(out)
30  );
31
32  // クロック生成
33  always #(STEP / 2) begin
34    clk = ~clk;
35  end
36
37  initial begin
38    clk <= 1'b0;
39    reset_n <= 1'b1;
40
41    toggle_sw_up <= 1'b0;
42    toggle_sw_down <= 1'b0;
43
44    /* リセット */
45
46    // 1 ns
47    #(STEP / 10) reset_n <= 1'b0;
48
49    // 3 ns
50    #(2 * STEP / 10) reset_n <= 1'b1;
51
52    /* レバー上側のテスト */
53
54    // 93 ns
55    #(9 * STEP) toggle_sw_up <= 1'b1;
56
```

```
57    // 123 ns
58    #(3 * STEP) toggle_sw_up <= 1'b0;
59
60    // 153 ns
61    #(3 * STEP) toggle_sw_up <= 1'b1;
62
63    // 233 ns
64    #(8 * STEP) toggle_sw_up <= 1'b0;
65
66    // 303 ns
67    #(7 * STEP) toggle_sw_up <= 1'b1;
68
69    // 583 ns
70    #(28 * STEP) toggle_sw_up <= 1'b0;
71
72    // 613 ns
73    #(3 * STEP) toggle_sw_up <= 1'b1;
74
75    // 673 ns
76    #(6 * STEP) toggle_sw_up <= 1'b0;
77
78    // 703 ns
79    #(3 * STEP) toggle_sw_up <= 1'b1;
80
81    // 733 ns
82    #(3 * STEP) toggle_sw_up <= 1'b0;
83
84    /* レバー下側のテスト */
85
86    // 1013 ns
87    #(28 * STEP) toggle_sw_down <= 1'b1;
88
89    // 1043 ns
90    #(3 * STEP) toggle_sw_down <= 1'b0;
91
92    // 1073 ns
93    #(3 * STEP) toggle_sw_down <= 1'b1;
94
95    // 1153 ns
96    #(8 * STEP) toggle_sw_down <= 1'b0;
97
98    // 1123 ns
99    #(7 * STEP) toggle_sw_down <= 1'b1;
100
101   // 1403 ns
102   #(28 * STEP) toggle_sw_down <= 1'b0;
103
104   // 1433 ns
105   #(3 * STEP) toggle_sw_down <= 1'b1;
106
107   // 1493 ns
```

```
108   #(6 * STEP) toggle_sw_down <= 1'b0;
109
110   // 1523 ns
111   #(3 * STEP) toggle_sw_down <= 1'b1;
112
113   // 1553 ns
114   #(3 * STEP) toggle_sw_down <= 1'b0;
115
116   /* レバー上下同時High（通常は起こり得ない）のテスト */
117
118   // 2013 ns
119   #(58 * STEP) toggle_sw_up <= 1'b1;
120
121   // 2043 ns
122   #(3 * STEP) toggle_sw_down <= 1'b1;
123
124   // 2233 ns
125   #(19 * STEP) toggle_sw_down <= 1'b0;
126
127   // 2273 ns
128   #(4 * STEP) toggle_sw_up <= 1'b0;
129
130   #(50 * STEP) $stop;
131 end
132
133 endmodule
```

図 3.47 にチャタリング除去回路の論理シミュレーション結果を示します．図 3.47 より，トグルスイッチからの入力信号のチャタリングが除去され，安定した信号が出力されることを確認できます．また，上側と下側の入力信号が同時に 1（H レベル）になるという通常起こり得ない状況において，安全のため出力が上側・下側ともに 0（L レベル）なることも分かります．

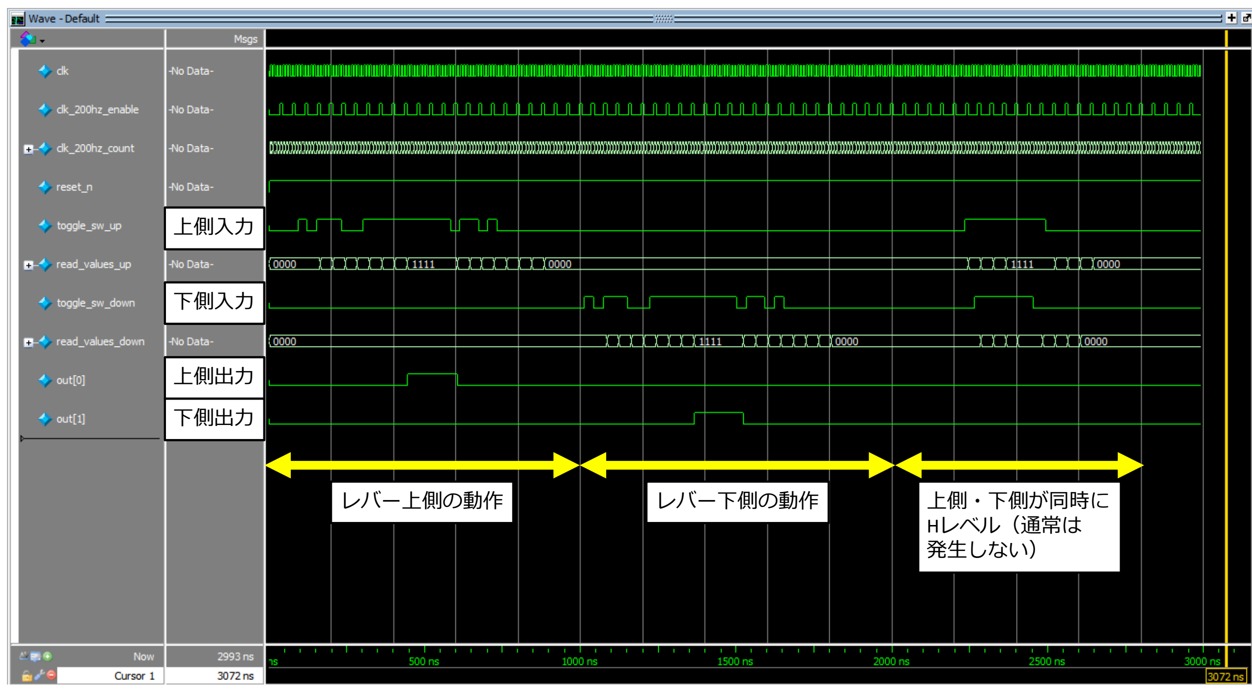


図 3.47　トグルスイッチのチャタリング除去回路の論理シミュレーション結果

3.2.6　チャタリング除去回路の動作確認

この項では，設計したチャタリング除去回路の動作確認を行います．動作確認では，Signal Tap を使用して，トグルスイッチからの入力信号および `toggle_sw` モジュールの出力信号を確認します．

観察する信号の追加

最初に，前項で追加した，`toggle_sw` モジュールの出力が接続されているポート `TOGGLE_SW_R_DEBOUNCED` を観察対象として追加します．

メニューから「Tools」→「Signal Tap Logic Analyzer」（図 3.29）を選択して Signal Tap を起動します．中央左側ペインの「Setup」タブを選択して設定画面に戻り，表の空白部分をダブルクリックして「Node Finder」画面を表示します（図 3.48）．

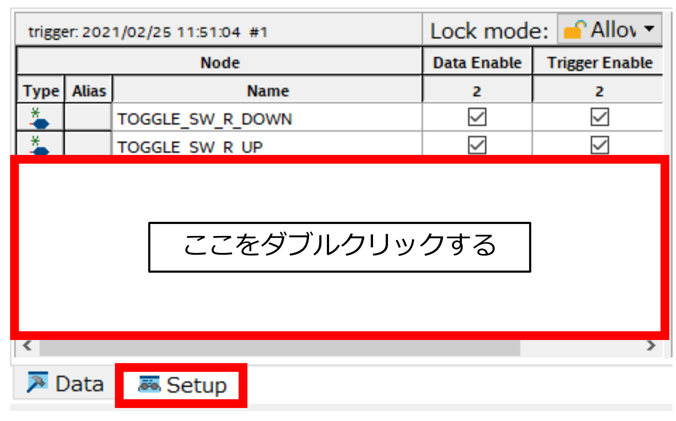


図 3.48　観察対象追加のためにダブルクリックする部分

「Node Finder」画面が表示されたら，「Filter」を「Signal Tap: pre-synthesis」に設定後，上部の「List」

ボタンをクリックして，信号を一覧表示します．その中から「TOGGLE_SW_R_DEBOUNCED」を選択して「>」ボタンをクリックし，右側の「Nodes Found」欄に追加します（図 3.49）．その後「Insert」ボタンをクリックすると，「Nodes Found」欄に表示された信号が観察対象として追加されます（図 3.50）．観察対象の信号が追加されたことを確認後，「Close」ボタンをクリックして「Node Finder」画面を閉じます．

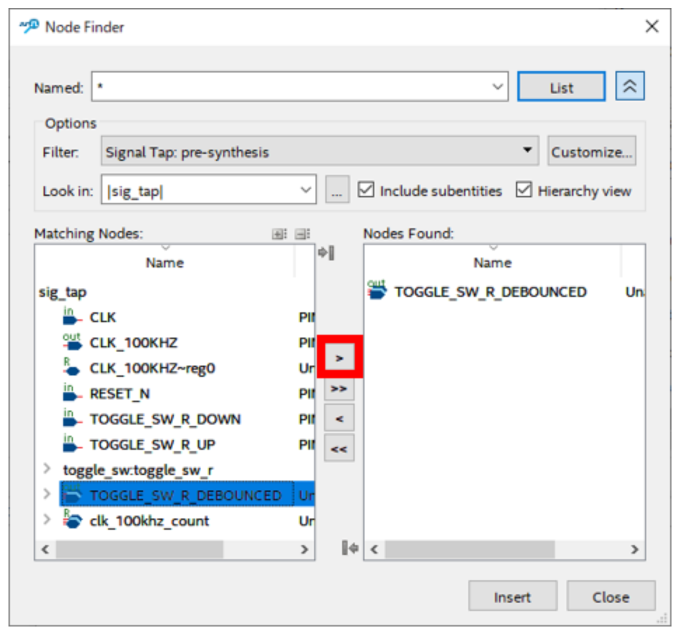


図 3.49　観察対象として追加する信号の選択

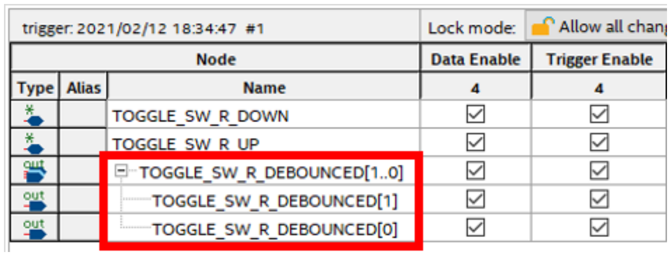


図 3.50　観察対象として追加された信号

トリガ条件の設定については，前回の観察時から変える必要はありません．

論理合成および FPGA へのプログラミング

設定後，Signal Tap と被測定回路とを合わせて論理合成します．ツールバーから「Start Compilation」をクリックして，プロジェクトをコンパイルします．

コンパイルが終了したら，Quartus Prime の画面に切り替えてハードウェアデザインを FPGA に書き込みます．2.2.6 項（p. 31〜）の手順で，Programmer を起動して FPGA へのプログラミングを行ってください．プログラミングの完了後，再び Signal Tap の画面に戻ります．

信号の観察

Signal Tap の準備ができたので，追加基板上のトグルスイッチのレバーを動かして，その際の信号の変化を観察します．

まず，追加基板上の右トグルスイッチのレバーを中央に戻します．続いて，「Instance Manager」

ペインの状況欄に「Ready to acquire」（取得の準備完了）と表示されているのを確認してから，「auto_signaltap_0」を選択して「Run Analysis」（解析実行）ボタンをクリックします（図 3.43）．状況欄に緑色の背景で「Acquisition in progress」（取得中）と表示されたら（図 3.44），右トグルスイッチのレバーを上に動かします．レバーを動かすと変化した信号が取り込まれ，図 3.51 のようにタイミングチャートが表示されます．

図 3.51　トグルスイッチのレバーを上に動かした際の信号の変化（チャタリング除去回路あり）

図 3.51 に示すように，トグルスイッチからの入力信号にチャタリングが発生していても，対応する出力信号 TOGGLE_SW_R_DEBOUNCED[0] は影響されません．この出力信号は，200 Hz クロックの立上り 4 回分連続で入力信号が 1 となったとき，初めて 0 から 1 に変化します．結果として，出力信号は処理に用いやすい単純な波形となります．

同様にして，トグルスイッチのレバーを中央から下に動かした際の出力信号も観察してみましょう．観察結果の例を図 3.52 に示します．この場合は，対応する出力信号 TOGGLE_SW_R_DEBOUNCED[1] が単純な波形で 0 から 1 に変化することを確認できます．

図 3.52　トグルスイッチのレバーを下に動かした際の信号の変化（チャタリング除去回路あり）

以上のように，今回設計した toggle_sw モジュールは，シフトレジスタを利用して入力信号のチャタリングを除去し，単純な波形の信号を出力します．シフトレジスタへのデータ蓄積の分だけ出力信号には遅延が発生するため，実際の設計では，チャタリングの発生時間やシステムの時間要件を考慮しながら，シフトレジスタのビット幅やクロック周波数を決めることになるでしょう．

3.2.7　制御への応用：ウィンカー回路の製作

チャタリング除去回路の導入によって，トグルスイッチは単純な波形の信号を出力するようになりました．このような波形になれば，出力信号のエッジ検出によってレバー位置の変化を読み取り，制御に利用することができます．ここでは，応用例として，追加基板上のウィンカー LED（図 3.53）の制御にトグルスイッチを利用してみましょう．

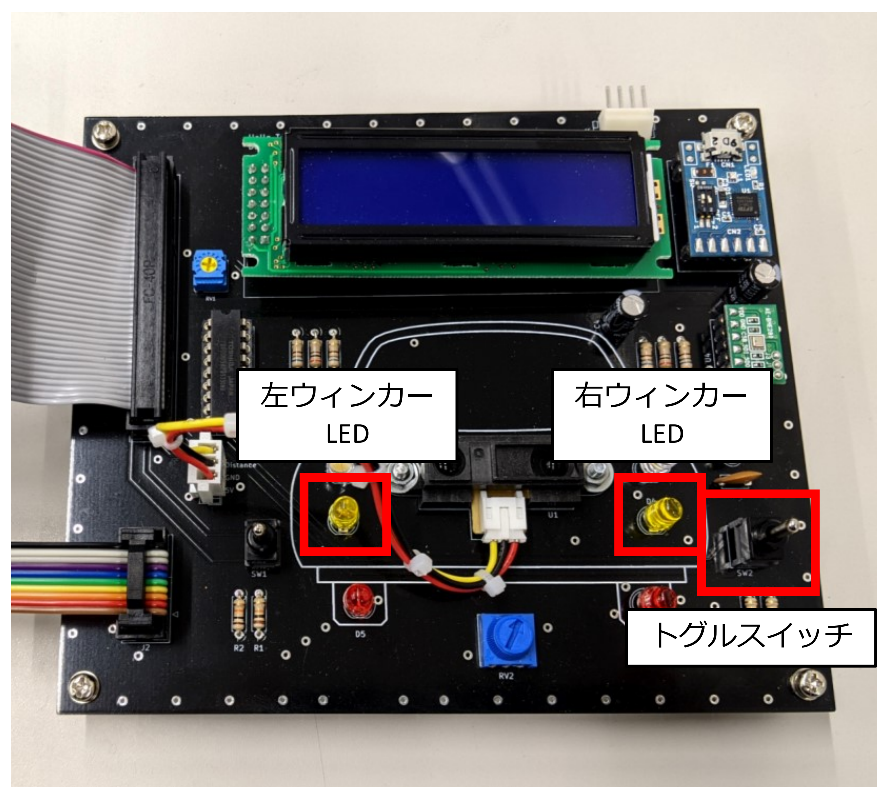


図 3.53　追加基板上のウィンカー LED とトグルスイッチ

ウィンカー回路の仕様

以下に示す仕様でウィンカー回路を製作します．

仕様

- 右トグルスイッチによってウィンカー LED の点灯状態を切り替えられる．
  - レバー中央：消灯
  - レバー上側：左ウィンカー LED が点滅，右ウィンカー LED が消灯
  - レバー下側：右ウィンカー LED が点滅，左ウィンカー LED が消灯
- ウィンカー LED が点滅する際の周期は 800 ms（400 ms 点灯，400 ms 消灯）とする．
- オンスタート機能（右トグルスイッチのレバーが端に動いた瞬間に，点灯状態から点滅を開始する機能）を持つ．

モジュール設計の方針

個々のウィンカーの点滅動作は単純ですが，ウィンカー回路全体では動作が意外に複雑です．以下に示す方針でモジュールを設計していきます．

■ LED の点滅　ウィンカー LED は正論理です．したがって，1（H レベル）を出力すると点灯し，`0`（L レベル）を出力すると消灯します．

LED の一定周期での点滅は，分周回路で実現できます．DE1-SoC のクロック周波数は 50 MHz，LED

の点滅周期は 800 ms = 0.8 s ですから，分周比 N は次のようになります．

$$N = \frac{50 \times 10^6\,\mathrm{Hz}}{\frac{1}{0.8\,\mathrm{s}}} = \frac{50 \times 10^6\,\mathrm{Hz}}{1.25\,\mathrm{Hz}} = 40,000,000$$

点灯時間と消灯時間が等しいため，デューティ比は 50% です．そのため，分周比の半分の 20,000,000 進カウンタを用意し，最大値までカウントしたときに出力信号の論理を反転するようにします．

トグルスイッチのレバーが端にあるときのみ点滅させるには，トグルスイッチの出力信号をイネーブル信号 `enable` として受け取り，それが 1 のときのみカウントを行うようにします．イネーブル信号が `0` のときは，カウントを止め，出力信号を `0` に設定して LED を消灯させます．

■ オンスタート機能　オンスタート機能（右トグルスイッチのレバーが端に動いた瞬間に，点灯状態から点滅を開始する機能）を実現するには，イネーブル信号のエッジ検出を行います．レバーが端にあるとき，`toggle_sw` の出力の対応するビットは 1 になるため，イネーブル信号の立ち上がり（`0` → 1）を検出します．検出時には，カウントを `0` から開始するとともに出力信号を 1 に設定して，LED の点滅が点灯状態から始まるようにします．このようにすれば，トグルスイッチのレバーが端に動いた瞬間から LED が 400 ms 点灯することが保証されます．

以上の動作を示すタイミングチャートを図 3.54 に示します．`prev_enable` は，エッジ検出のため，1 回前のクロックにおけるイネーブル信号を記憶しておくレジスタです．イネーブル信号 `enable` が 1，かつ 1 回前のイネーブル信号 `prev_enable` が `0` のとき，イネーブル信号が立ち上がったと判断します．このとき，カウント値のレジスタおよび出力信号のレジスタに値をノンブロッキング代入します．結果として，イネーブル信号の立ち上がり検出時の次のクロックから LED の点滅が始まります．

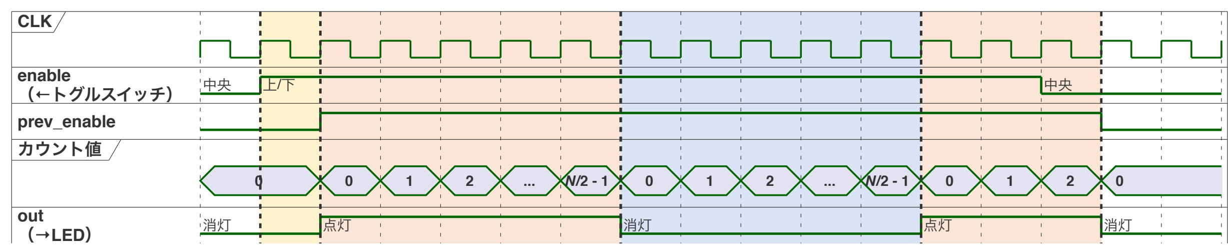


図 3.54　ウィンカー回路のタイミングチャート

■ 階層設計　左右のウィンカーを比較すると，トグルスイッチから入力されるイネーブル信号が異なるだけで，動作は同じです．そこで，個々のウィンカーの動作を 1 つのモジュールで実装して，最上位階層が 2 個（左右）の実体（インスタンス）を持つように構成します．この構成にすると，必要なコードの量を減らすことができます．

ウィンカー回路全体の構成を図 3.55 に示します．最上位階層の名前は `blinker_lr`，個々のウィンカーのモジュール名は `blinker` とします．左右のウィンカーのインスタンス名は，それぞれ `blinker_l`，`blinker_r` とします．トグルスイッチ上側の出力信号 `out[0]` を左ウィンカー `blinker_l` のイネーブル信号として，またトグルスイッチ下側の出力信号 `out[1]` を右ウィンカー `blinker_r` のイネーブル信号として，それぞれ入力します．

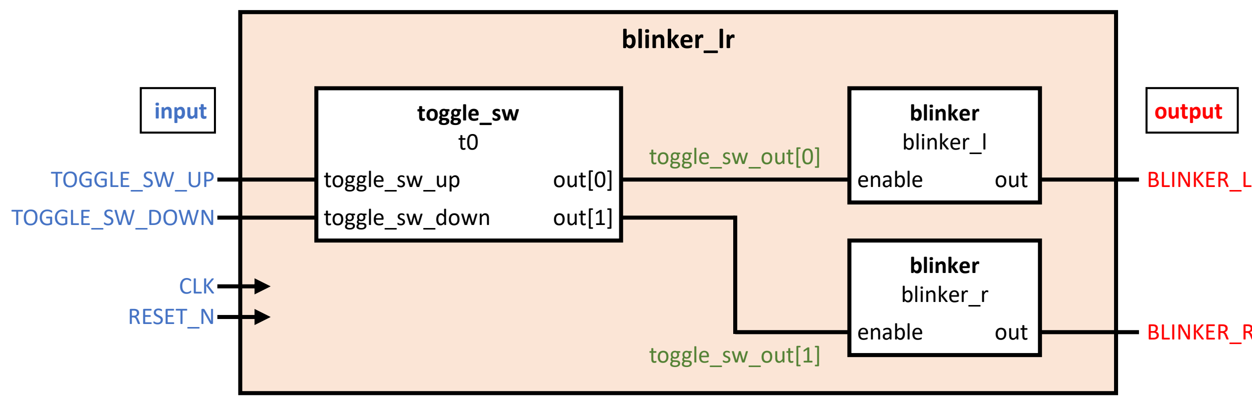


図 3.55　ウィンカー回路の構成

プロジェクトの作成

表 3.11 の設定で，Quartus Prime のプロジェクトを作成してください．プロジェクト作成の手順については，2.2.2 項（p. 9〜）を参照してください．

表 3.11　ウィンカー回路用プロジェクト作成時の設定

項目	設定値
作業フォルダ（working directory）	`D:\DE1\blinker_lr`
プロジェクト名（name of this project）	`blinker_lr`
最上位階層名（name of the top-level design entity）	`blinker_lr`
開発キット（Development Kit）	DE1-SoC Board

プロジェクトが作成されたら，今回作成したトグルスイッチのモジュールをプロジェクトに追加します．Windows のエクスプローラーを使用して，`D:\DE1\sig_tap\toggle_sw.v` を，プロジェクトのフォルダ `D:\DE1\blinker_lr` にコピーします．コピー後，メニューから「Add/Remove files in Project...」を選択して，`toggle_sw.v` をプロジェクトに追加してください．

ウィンカーの実装

仕様に従い，個々のウィンカーのモジュールと最上位階層を実装します．

■個々のウィンカーのモジュール　個々のウィンカーのモジュール `blinker` を実装します．このモジュールのインターフェースは，表 3.12 のように設定することとします．

表 3.12　`blinker` モジュールのインターフェース

ポート名	方向	ビット幅	信号の役割	論理
`clk`	入力	1	50 MHz クロック	
`reset_n`	入力	1	リセット	負論理
`enable`	入力	1	点滅イネーブル	正論理
`out`	出力	1	ウィンカー LED への出力	正論理

次に示すリスト 3.12 を記述して D:\DE1\blinker_lr\blinker.v に保存し，プロジェクトに追加してください．

リスト 3.12　個々のウィンカーのモジュールの実装（blinker.v）

```
1  // ウィンカーのモジュール
2  module blinker (
3    // 50 MHzクロック
4    input clk,
5    // リセット（負論理）
6    input reset_n,
7    // 点滅イネーブル
8    input enable,
9    // 出力
10   output reg out
11 );
12
13 // 分周比
14 // 周期0.8 s => 周波数1.25 Hz
15 // => 分周比50 M / 1.25 = 40,000,000
16 // デューティ比50%より半分 => 20,000,000
17 parameter PRESCALE_RATIO = 25'd20_000_000;
18
19 reg [24:0] prescale_count;
20 wire prescale_en;
21
22 // 1回前の点滅イネーブル
23 reg prev_enable;
24
25 assign prescale_en =
26   (prescale_count == PRESCALE_RATIO - 25'd1);
27
28 always @(posedge clk or negedge reset_n) begin
29   if (!reset_n) begin
30     prescale_count <= 25'd0;
31     prev_enable <= 1'b0;
32     out <= 1'b0;
33   end else begin
34     if (enable) begin
35       // 点滅有効
36
37       if (!prev_enable) begin
38         // 立ち上がり（0→1）を検出 => 点灯状態から開始
39         prescale_count <= 25'd0;
40         out <= 1'b1;
41       end else if (prescale_en) begin
42         // 半周期経過 => 論理反転
43         prescale_count <= 25'd0;
44         out <= ~out;
45       end else begin
46         prescale_count <= prescale_count + 25'd1;
47       end
```

```
48      end else begin
49        // 点滅無効 => 消灯、カウンタリセット
50        prescale_count <= 25'd0;
51        out <= 1'b0;
52      end
53
54      // 次回に備えて、現在の点滅イネーブル値を記憶する
55      prev_enable <= enable;
56    end
57  end
58
59  endmodule
```

■**最上位階層**　トグルスイッチと個々のウィンカーを接続する最上位階層 blinker_lr を実装します．最上位階層のインターフェースは，表 3.13 のように設定することとします．

表 3.13　blinker_lr モジュールのインターフェース

ポート名	方向	ビット幅	信号の役割	論理
CLK	入力	1	50 MHz クロック	
RESET_N	入力	1	リセット	負論理
TOGGLE_SW_UP	入力	1	右トグルスイッチ上側	正論理
TOGGLE_SW_DOWN	入力	1	右トグルスイッチ下側	正論理
BLINKER_L	出力	1	左ウィンカー	正論理
BLINKER_R	出力	1	右ウィンカー	正論理

次に示すリスト 3.13 を記述して D:\DE1\blinker_lr\blinker_lr.v に保存し，プロジェクトに追加してください．

リスト 3.13　ウィンカー回路の最上位階層の実装（blinker_lr.v）

```
1   // 左右ウィンカーのモジュール
2   module blinker_lr (
3     // 50 MHzクロック
4     input CLK,
5     // リセット（負論理）
6     input RESET_N,
7
8     // トグルスイッチ上側
9     input TOGGLE_SW_UP,
10    // トグルスイッチ下側
11    input TOGGLE_SW_DOWN,
12    // 左ウィンカー
13    output BLINKER_L,
14    // 右ウィンカー
15    output BLINKER_R
16  );
17
```

```
18 wire [1:0] toggle_sw_out;
19
20 // トグルスイッチの実体
21 // 出力値
22 // * 2'b00: 中央
23 // * 2'b01: 上側
24 // * 2'b10: 下側
25 toggle_sw t0 (
26   .clk (CLK),
27   .reset_n (RESET_N),
28   .toggle_sw_up (TOGGLE_SW_UP),
29   .toggle_sw_down (TOGGLE_SW_DOWN),
30   .out (toggle_sw_out)
31 );
32
33 // 左ウィンカーの実体
34 blinker blinker_l (
35   .clk (CLK),
36   .reset_n (RESET_N),
37   .enable (toggle_sw_out[0]),
38   .out (BLINKER_L)
39 );
40
41 // 右ウィンカーの実体
42 blinker blinker_r (
43   .clk (CLK),
44   .reset_n (RESET_N),
45   .enable (toggle_sw_out[1]),
46   .out (BLINKER_R)
47 );
48
49 endmodule
```

論理合成，ピンの割り当て，コンパイル

2 つのモジュールの実装後，左側にある「Tasks」ペインの「Analysis & Synthesis」（図 2.37，p. 28）をダブルクリックして，論理合成を行います．論理合成が成功したら，メニューから「Assignments」→「Pin Planner」を選択して Pin Planner を起動し，表 3.14 のようにピンアサインを行います．

表 3.14　ウィンカー回路のピンの割り当て

Node Name	Direction	Location	I/O Standard	割り当て先の機能
CLK	Input	PIN_AF14	3.3-V LVCMOS	50 MHz クロック
RESET_N	Input	PIN_AA14	3.3-V LVCMOS	ボタンスイッチ KEY0（負論理）
TOGGLE_SW_DOWN	Input	PIN_AG20	3.3-V LVCMOS	右トグルスイッチ下側
TOGGLE_SW_UP	Input	PIN_AJ21	3.3-V LVCMOS	右トグルスイッチ上側
BLINKER_L	Output	PIN_AF21	3.3-V LVCMOS	左ウィンカー LED
BLINKER_R	Output	PIN_AK21	3.3-V LVCMOS	右ウィンカー LED

ピンを割り当てた後は，「Tasks」ペインの「Compile Design」（図 2.41，p. 30）をダブルクリックして回路をコンパイルしてください．

プログラミングと動作確認

コンパイルが終了したら，2.2.6 項（p. 31～）の手順で，Programmer を起動して FPGA へのプログラミングを行います．

回路を書き込んだら，右トグルスイッチを操作して，ウィンカー回路が仕様どおりに動作するか確認します．特に，ウィンカー LED の点滅が起こる条件，点滅の周期，およびオンスタート機能が働いているかに注意して，動作確認を行ってください．

3.2.8　Nios II からの利用

この項では，追加基板上のトグルスイッチを Nios II から利用する方法について解説します．

マイコンシステムの構成

マイコンシステムを構成する際には，`toggle_sw` モジュールからの出力を入力するための PIO を追加します．設定内容を表 3.15 に示します．

表 3.15　トグルスイッチ用の PIO の設定

名称例	ビット幅	方向	用途
toggle_sw_l	2	Input（入力）	左トグルスイッチ
toggle_sw_r	2	Input（入力）	右トグルスイッチ

Verilog コードの記述

Verilog コードにおいて，`wire` を介して `toggle_sw` モジュールとマイコンシステムを接続します．記述方法の概要をリスト 3.14 に示します．

リスト 3.14　`toggle_sw` モジュールとマイコンシステムの接続

```
module some_system (
  // 50 MHzクロック
  input CLK,
  // リセット（負論理）
  input RESET_N,

  // 左トグルスイッチ上側
  input TOGGLE_SW_L_UP,
  // 左トグルスイッチ下側
  input TOGGLE_SW_L_DOWN,

  // 右トグルスイッチ上側
  input TOGGLE_SW_R_UP,
  // 右トグルスイッチ下側
  input TOGGLE_SW_R_DOWN,
);
```

```
// 左トグルスイッチのチャタリング除去後の値
wire [1:0] toggle_sw_l_value;
// 右トグルスイッチのチャタリング除去後の値
wire [1:0] toggle_sw_r_value;

// 左トグルスイッチのチャタリング除去
toggle_sw toggle_sw_l(
  .clk(CLK),
  .reset_n(RESET_N),
  .toggle_sw_up(TOGGLE_SW_L_UP),
  .toggle_sw_down(TOGGLE_SW_L_DOWN),
  .out(toggle_sw_l_value)
);

// 右トグルスイッチのチャタリング除去
toggle_sw toggle_sw_r(
  .clk(CLK),
  .reset_n(RESET_N),
  .toggle_sw_up(TOGGLE_SW_R_UP),
  .toggle_sw_down(TOGGLE_SW_R_DOWN),
  .out(toggle_sw_r_value)
);

// Nios IIマイコンシステム
nios2_system u0 (
  // ...

  .toggle_sw_l_export (toggle_sw_l_value),
  .toggle_sw_r_export (toggle_sw_r_value),

  // ...
);
```

C 言語コードの記述

C 言語の組込みプログラムからは，IORD_ALTERA_AVALON_PIO_DATA() マクロを使用して，チャタリング除去されたトグルスイッチの信号を読み込めます．その際に，マクロを使用してレバーの位置と対応する値に名前を付けておくと，コードが読みやすくなります．記述例をリスト 3.15 に示します．

リスト 3.15　C 言語によるトグルスイッチ入力の記述例

```
#include "altera_avalon_pio_regs.h"

/** トグルスイッチのレバーが中央にある。 */
#define TOGGLE_SW_CENTER 0x0
/** トグルスイッチのレバーが上側にある。 */
#define TOGGLE_SW_UP      (0x1 << 0)
/** トグルスイッチのレバーが下側にある。 */
#define TOGGLE_SW_DOWN    (0x1 << 1)
```

```
// 左トグルスイッチの値を読み込む
uint8_t toggle_sw_l = IORD_ALTERA_AVALON_PIO_DATA(TOGGLE_SW_L_BASE);

// 右トグルスイッチの値を読み込む
uint8_t toggle_sw_r = IORD_ALTERA_AVALON_PIO_DATA(TOGGLE_SW_R_BASE);

if (toggle_sw_l == TOGGLE_SW_UP) {
  // 左トグルスイッチのレバーが上側にあるときの処理
} else if (toggle_sw_l == TOGGLE_SW_DOWN) {
  // 左トグルスイッチのレバーが下側にあるときの処理
}

if (toggle_sw_r == TOGGLE_SW_UP) {
  // 右トグルスイッチのレバーが上側にあるときの処理
} else if (toggle_sw_r == TOGGLE_SW_DOWN) {
  // 右トグルスイッチのレバーが下側にあるときの処理
}
```

3.3　まとめ

この章では，ソフトウェアで記述した処理のハードウェア化，および FPGA に組み込んだロジックアナライザを活用した回路設計に取り組みました．要点を以下にまとめます．

- FPGA 開発とマイコン開発にはそれぞれ長所と短所がある．製品を開発する際には，各要素に関するトレードオフを考慮して，適切にシステムを設計する必要がある．
- テスト駆動開発では，最初に失敗するテストコードを書き，続いてテストが成功するようにコードを書く．さらに，テストが成功する状態を維持したまま，内部の構造を改善する（リファクタリング）．この手順を実行することで，無駄がなく確実に動作するコードを書くことができる．
- FPGA にはロジックアナライザ機能を組み込むことができる．この機能を利用することで，実際の入出力波形を確認しながらハードウェア開発を進めていくことができる．

次の章では，この章で開発した `seven_seg_decoder` モジュールおよび `toggle_sw` モジュールも利用して，より複雑な制御システムの構築に取り組みます．

第 4 章

IP コアを活用した制御システム構築

これまでの章では，Parallel I/O（PIO）のみを利用して，Verilog で記述したハードウェアを制御してきました．しかし，センサからのデータ入力，厳密なタイミング制御，機器との通信など，より複雑な制御を PIO だけで行うのは非現実的です．このような制御を短い期間で実装するには，通常のマイコン開発と同様に，専用の機能を活用することが必要になります．

近年の FPGA 開発では，機能別に用意された回路部品の設計情報である **IP コア**の活用が広まっています．現在は，無償のものも含む多くの IP コアが流通しており，デジタル回路におけるロジック IC のように，便利な機能を手軽に利用できます．この実習で使用している Nios II EDS にも，無償で使用可能な膨大な数の IP コアが同梱されています．

そこでこの章では，IP コアを活用して，製品を意識した制御システムを構築する課題に取り組みます．これまでの章よりも課題の規模が大きくなるため，IP コアの活用方法に加えて，大規模な組込み開発を見通しよく行う方法についても解説します．

4.1　製作する制御システムの仕様

この章の課題で製作する制御システムは，簡易車載システムです．このシステムは，実習で使用してきた DE1-SoC ボードと，それに接続する追加基板から構成されます（図 4.1）．追加基板には，トグルスイッチ，LED，キャラクタ LCD，および各種センサが実装されています（図 4.2）．追加基板の回路図を図 4.3 に示します．これらの装置を利用して，自動車の電装品やインストルメントパネル（計器盤）の機能を持つ組込みシステムを構築していきます．

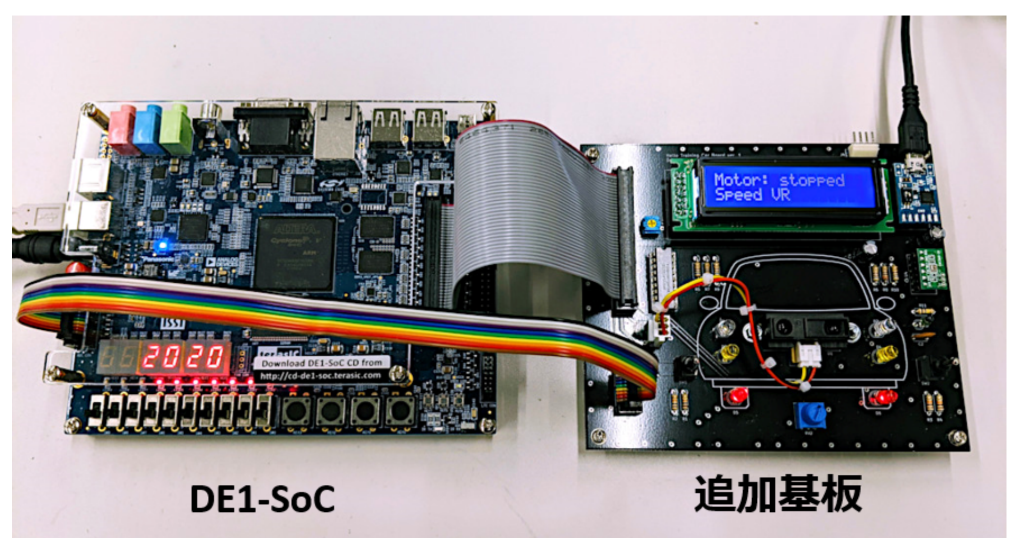
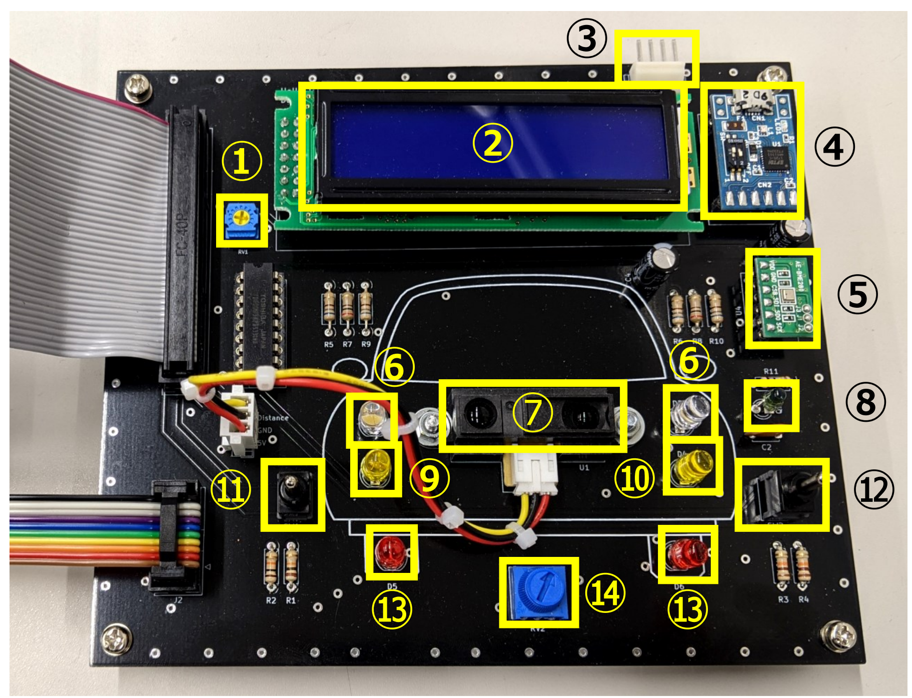


図 4.1　簡易車載システムを構成する基板



① LCDコントラスト調整用可変抵抗器	② キャラクタLCDモジュール	③ シリアル通信用コネクタ
④ USBシリアル通信モジュール（対PC用）	⑤ 温湿度・気圧センサモジュール	⑥ ヘッドライトLED
⑦ 測距センサ	⑧ 照度センサ（フォトトランジスタ）	⑨ 左ウィンカーLED
⑩ 右ウィンカーLED	⑪ 左トグルスイッチ	⑫ 右トグルスイッチ
⑬ ブレーキLED	⑭ モータ速度調整用可変抵抗器	

図 4.2　追加基板上に実装された装置

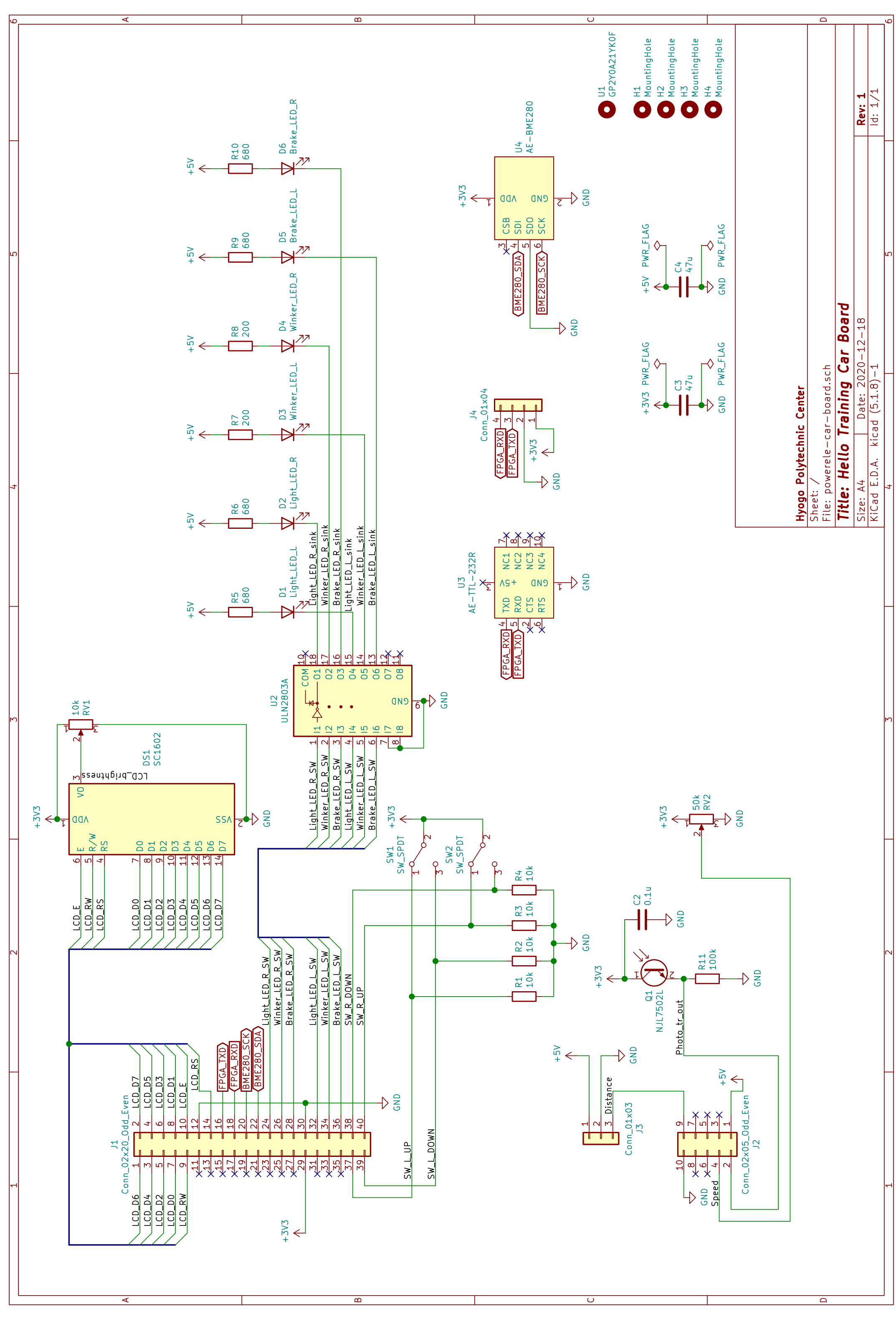


図 4.3　追加基板の回路図

以下では，システムに搭載する各機能の仕様について述べます．

4.1.1　モータ制御

- 車輪を模したモータ（別の基板に接続）を制御できる．
  - KEY2 を押すことでモータの回転/停止を切り替えられる．
  - 速度調整用可変抵抗を用いてモータの回転速度を制御できる．
- モータの制御にはシリアル通信（UART）を使用する．
- モータの停止中は左右のブレーキ LED を点灯させ，回転中は消灯させる．

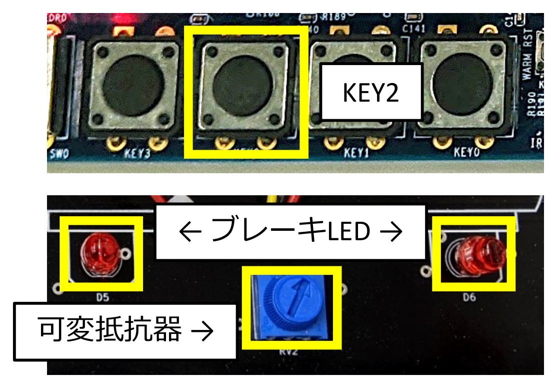


図 4.4　モータ制御関連の装置

4.1.2　ヘッドライト

- 左トグルスイッチによってヘッドライト LED の点灯状態（点灯/消灯）を切り替えられる．
  - レバー下側：消灯
  - レバー上側：点灯
  - レバー中央：自動点灯（周囲が明るいとき消灯，暗いとき点灯）
- 左右のヘッドライト LED の点灯状態が等しい．

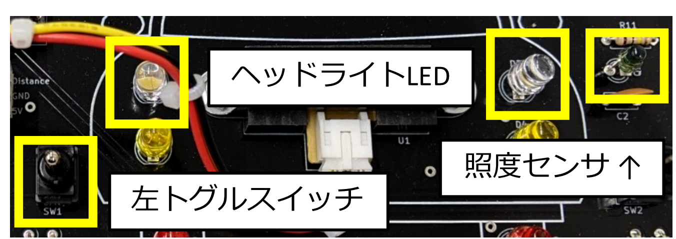


図 4.5　ヘッドライト制御関連の装置

4.1.3　ウィンカー

- 右トグルスイッチによってウィンカー LED の点灯状態を切り替えられる．
  - レバー中央：消灯
  - レバー上側：左ウィンカー LED が点滅，右ウィンカー LED が消灯

– レバー下側：右ウィンカー LED が点滅，左ウィンカー LED が消灯

- ウィンカー LED が点滅する際の周期は 800 ms（400 ms 点灯，400 ms 消灯）とする．
- オンスタート機能（右トグルスイッチのレバーが端に動いた瞬間に，点灯状態から点滅を開始する機能）を持つ．

図 4.6　ウィンカー制御関連の装置

4.1.4　メータ

- 以下の項目の値を 6 桁の 7 セグメント LED に表示できる．
  1. モータの回転速度設定（%）
  2. 速度調整用可変抵抗の A/D 変換値
  3. 測距センサの A/D 変換値
  4. 照度センサの A/D 変換値
- 各項目について，最大値に対する現在の値の割合を LEDR に表示できる（レベルメータ）．
- KEY1（次の項目）および KEY3（前の項目）を利用して表示項目を切り替えられる．
  – 最初の項目が選択されているときに KEY3 を押すと，最後の項目に切り替わる．
  – 最後の項目が選択されているときに KEY1 を押すと，最初の項目に切り替わる．
- 数値を 7 セグメント LED に表示する際，先頭のゼロを省略する．

図 4.7　メータ制御関連の装置

4.1.5　文字表示器

- 2 行分のキャラクタ LCD モジュールに現在の状態を表示できる（図 4.8）．
  - 1 行目にはモータの状態（停止中/回転中）を表示する．
    - 停止中：「`Motor: stopped`」
    - 回転中：「`Motor: running`」
  - 2 行目には，メータに表示されている項目を表示する．各項目に対応する文字列を表 4.1 に示す．

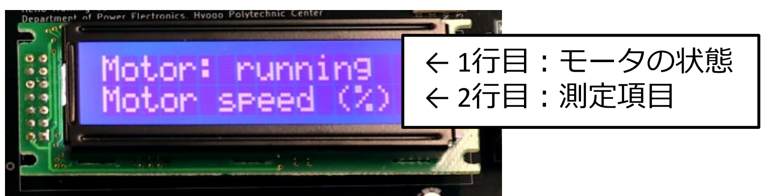


図 4.8　キャラクタ LCD に表示する文字列

表 4.1　キャラクタ LCD に表示する文字列：測定項目

番号	測定項目	文字列
0	モータの回転速度設定（%）	`Motor speed (%)`
1	速度調整用可変抵抗の A/D 変換値	`Speed VR`
2	測距センサの A/D 変換値	`Distance`
3	照度センサの A/D 変換値	`Luminance`

4.2　ハードウェアの構築

最初に，Quartus Prime および Platform Designer を使用して簡易車載システムのハードウェアを構築します．今回のハードウェアの構成を図 4.9 に示します．仕様に挙げた機能のうちウィンカーについては，3.2.7 項で製作したウィンカー回路（`blinker_lr` モジュール）が仕様を満たすためそのまま使用し，マイコンシステムとは独立に動作するようにします．

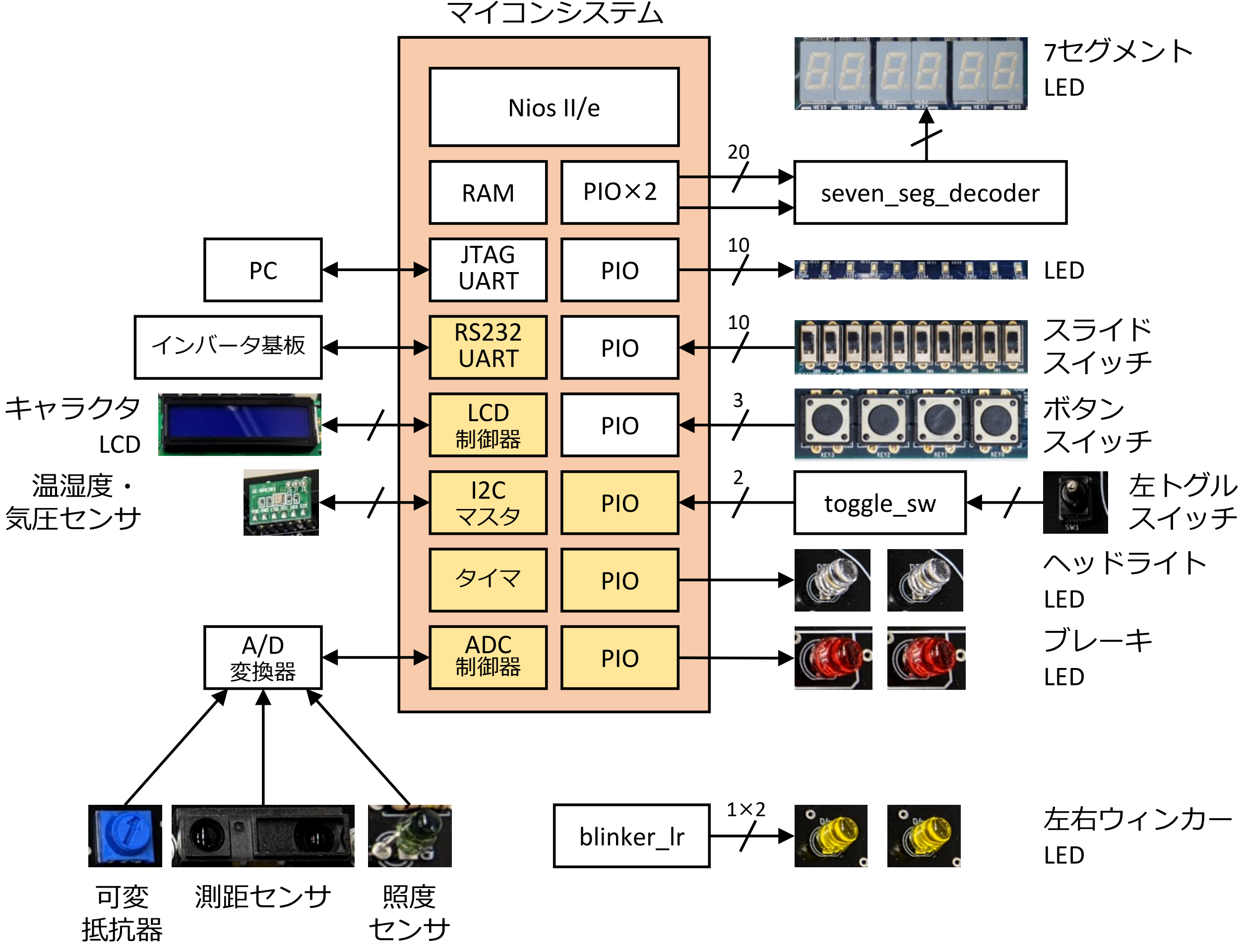


図 4.9　簡易車載システムのハードウェア構成

4.2.1　プロジェクトの作成

まず Quartus Prime のプロジェクトを作成します．Quartus Prime を起動して，表 4.2 の設定でプロジェクトを作成してください．プロジェクト作成の手順については，2.2.2 項（p. 9〜）を参照してください．

表 4.2　簡易車載システムのプロジェクト作成時の設定

項目	設定値
作業フォルダ（working directory）	`D:\DE1\car_system`
プロジェクト名（name of this project）	`car_system`
最上位階層名（name of the top-level design entity）	`car_system`
開発キット（Development Kit）	DE1-SoC Board

4.2.2　Nios II および周辺回路の構成

ハードウェアデザインプロジェクトの作成後，Platform Designer を用いて，Nios II および周辺回路を構成します．

既存のマイコンシステム構成の再利用

簡易車載システムのマイコンシステムには，第 2 章で構成したものと共通する IP コアが多く含まれます．そのため，今回は第 2 章で作成したマイコンシステムを再利用して，効率良くマイコンシステムを構成してみましょう．

Quartus Prime のメニュー「Tools」→「Platform Designer」から Platform Designer を起動します．Platform Designer の起動後，メニューから「File」→「Open...」を選択して，第 2 章で作成したマイコンシステムの構成設定 `D:\DE1\pio\nios2_pio.qsys` を開きます．このファイルが開かれたら，メニューから「File」→「Save As...」を選択します．ファイル保存のダイアログで `D:\DE1\car_system` に移動して，`nios2_car_system.qsys` という名前で保存してください．以降は，ここで保存した `nios2_car_system.qsys` に変更を加えていきます．

7 セグメント LED 用 PIO の追加

簡易車載システムの 7 セグメント LED の制御には，3.1 節で作成したモジュール `seven_seg` を使用します．3.1.5 項で述べたように，このモジュールを使用する際には数値レジスタ用およびゼロ埋め制御レジスタ用の PIO を追加します．

レジスタ用 PIO の追加の前に，第 2 章で追加した PIO を削除しておく必要があります．`hex_0` ～`hex_5` を選択した後，右クリックで表示されるメニューから「Remove」を選択すると（図 4.10），6 つの PIO をまとめて削除できます．

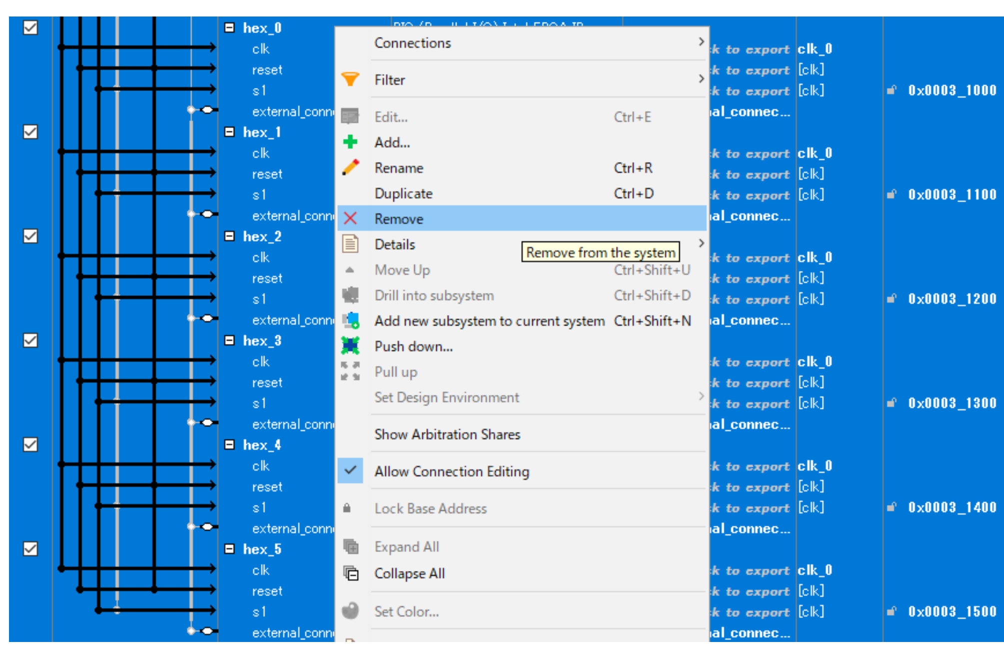


図 4.10　7 セグメント LED 用 PIO の削除

PIO の削除後，p. 16 の手順に従って，表 4.3 の設定で PIO を追加してください．

表 4.3　7 セグメント LED への数値表示処理用の PIO の設定

名称	ビット幅	方向	用途
seven_seg_num	20	Output（出力）	数値レジスタ
seven_seg_zero_suppress	1	Output（出力）	ゼロ埋め制御レジスタ

トグルスイッチ入力用および各種 LED 制御用 PIO の追加

追加基板上のトグルスイッチ入力用および各種 LED の制御用の PIO を追加します．p. 16 の手順に従って，表 4.4 の設定で PIO を追加してください．

表 4.4　トグルスイッチ入力用および各種 LED 制御用 PIO の設定

名称	ビット幅	方向	用途
toggle_sw_l	2	Input（入力）	左トグルスイッチ
light_led	1	Output（出力）	ヘッドライト LED
brake_led	1	Output（出力）	ブレーキ LED

キャラクタ LCD コントローラの追加

追加基板上のキャラクタ LCD モジュールを制御するためのコントローラを追加します．「IP Catalog」ペインにおいて「University Program」→「Audio & Video」→「16x2 Character Display」を選択した後，「Add」ボタンをクリックして（図 4.11），キャラクタ LCD コントローラを追加します．

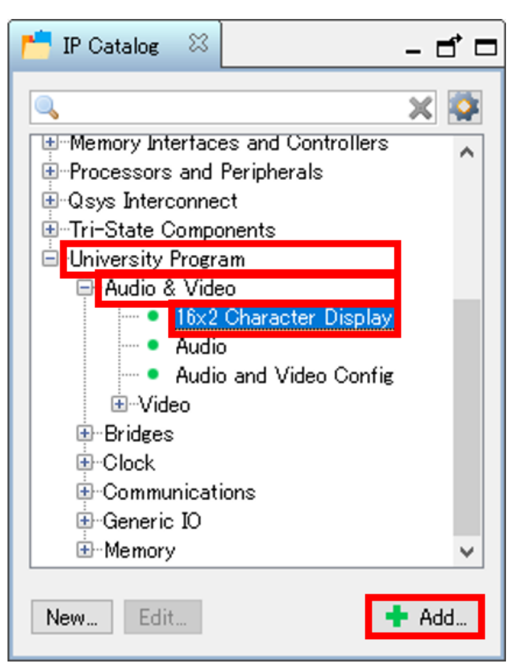


図 4.11　キャラクタ LCD コントローラの選択

続いてキャラクタ LCD コントローラの設定画面（図 4.12）が表示されます．この画面では，キャラクタ LCD のカーソル表示について設定できます．カーソルを非表示にするため，「Display Cursor」を「None」に設定して「Finish」ボタンをクリックしてください．

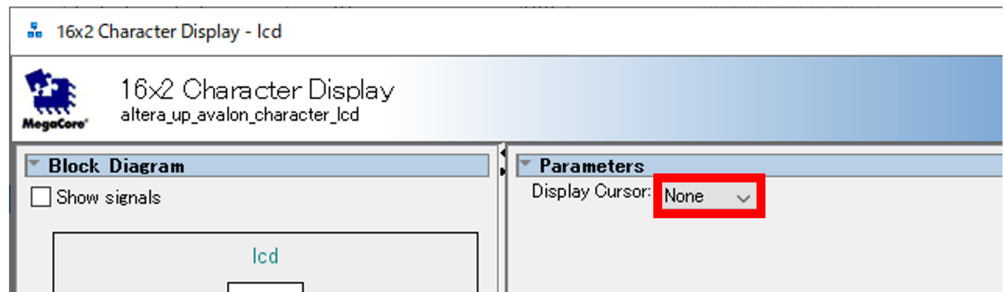


図 4.12　キャラクタ LCD コントローラの設定

追加後は，この IP コアを「lcd」に改名します．

A/D 変換器コントローラの追加

追加基板上の可変抵抗器や各種センサが出力する電圧を A/D 変換するための A/D 変換器コントローラを追加します．「IP Catalog」ペインにおいて「University Program」→「Generic IO」→「ADC Controller for DE-series Boards」を選択した後，「Add」ボタンをクリックして（図 4.13），A/D 変換器コントローラを追加します．

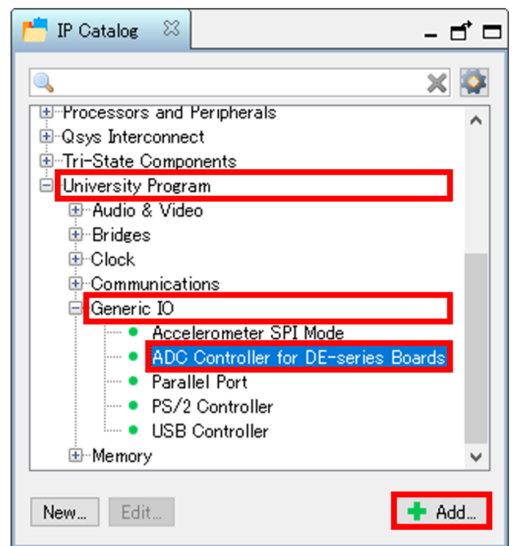


図 4.13　A/D 変換器コントローラの選択

続いて A/D 変換器コントローラの設定画面（図 4.14）が表示されます．A/D 変換器が実装されている基板の種類を選択する必要があるため，「DE-Series Board」を「DE1-SoC」に設定して「Finish」ボタンをクリックしてください．

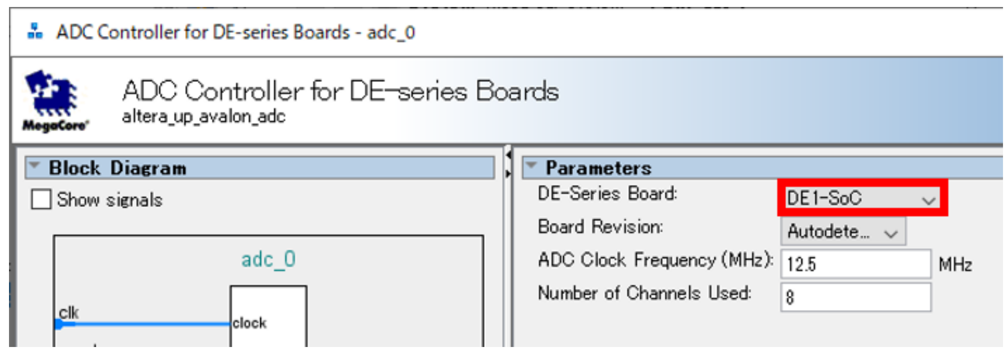


図 4.14　A/D 変換器コントローラの設定

追加後は，この IP コアを「adc」に改名します．

RS232 UART の追加

PC，あるいは 3 相ブラシレス DC モータのインバータ基板とのシリアル通信を行えるようにするため，UART の IP コアを追加します．「IP Catalog」ペインにおいて「University Program」→「Communications」→「RS232 UART」を選択した後，「Add」ボタンをクリックして（図 4.15），この IP コアを追加します．

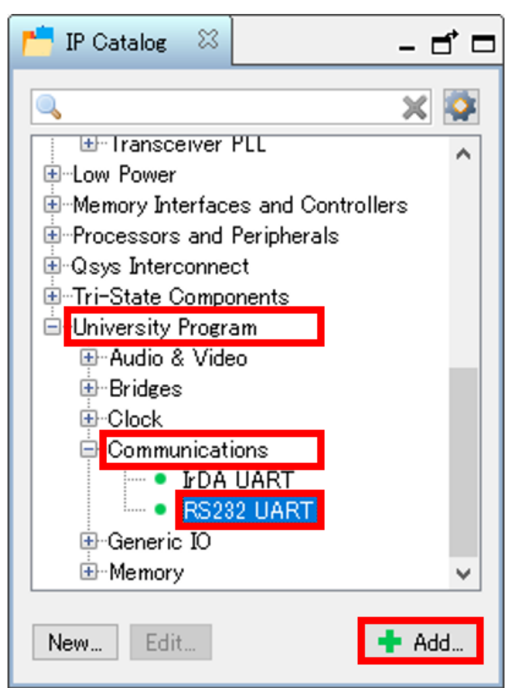


図 4.15　RS232 UART の選択

続いて RS232 UART の設定画面（図 4.16）が表示されます．今回はシリアル通信の設定を，ボーレート 38 400 bps，パリティなし，データ長 8 ビット，ストップビット長 1 ビットに設定します．「Baud Rate (bps)」を 38400 に設定してください．残りの項目は既定値のままにしますが，「Data Format」欄の各項目が省略表示されているため，ドロップダウンリストを展開して値が図 4.16 の説明どおりになっているか確認してください．

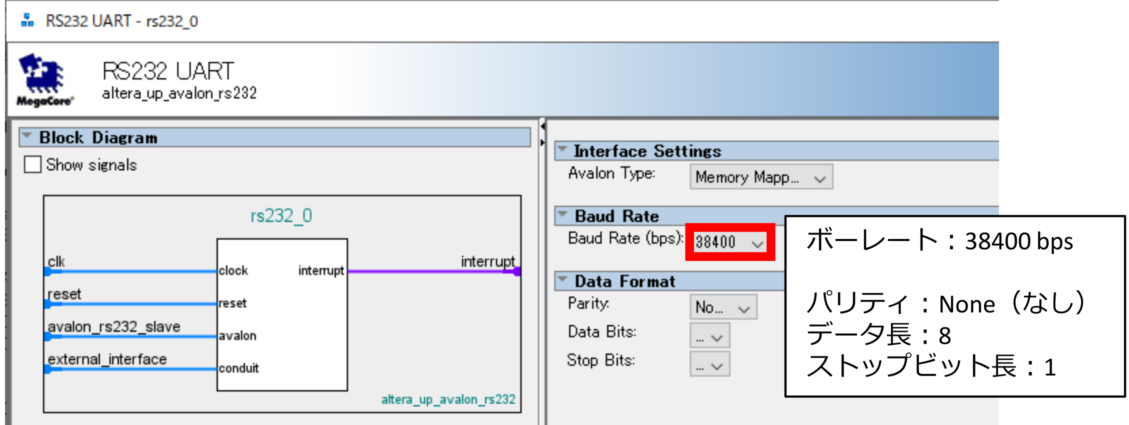


図 4.16　RS232 UART の設定

追加後は，この IP コアを「uart」に改名します．

タイマの追加

今回製作する簡易車載システムでは，時間をより正確に計測するため，タイマの IP コアを追加します．「IP Catalog」ペインにおいて「Processors and Peripherals」→「Peripherals」→「Interval Timer Intel FPGA IP」を選択した後，「Add」ボタンをクリックして（図 4.17），この IP コアを追加します．

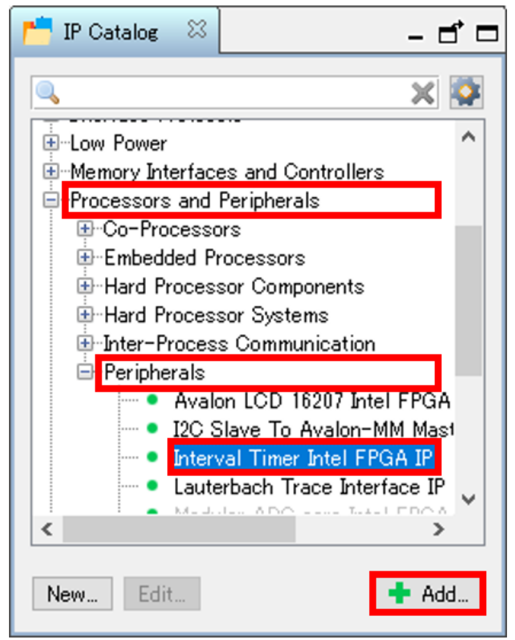


図 4.17　タイマの選択

続いてタイマの設定画面（図 4.18）が表示されます．この画面では，タイマの周期やカウンタのビット幅の設定のほか，動作の細かな調整が行えます．今回は既定値（周期 1 ms，32 ビットのカウンタ）のまま使用しますが，用途によっては調整が必要になるでしょう．

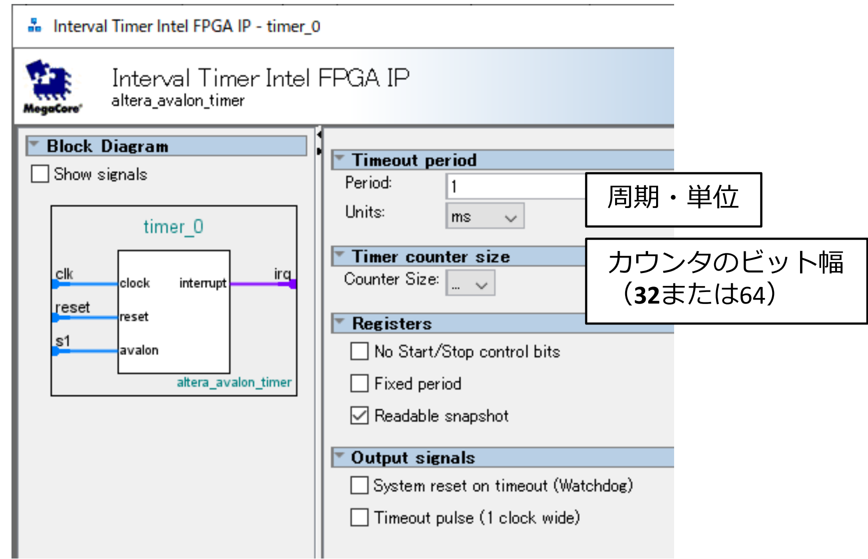


図 4.18　タイマの設定

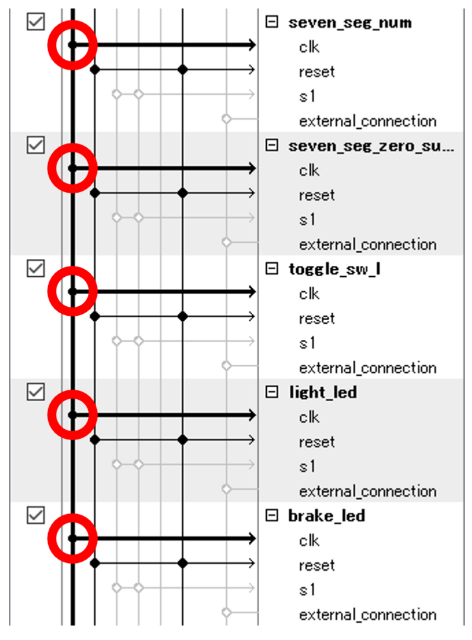
追加後は，この IP コアを「timer_1ms」に改名します．
以上で IP コアの追加は完了です．

IP コア間の配線

以上の作業で追加した IP コアに対して，システムの動作に必要な配線を行います．

最初に，clk_0 からのリセット信号を各 IP コアに接続します．メニューから「System」→「Create Global Reset Network」を選択してリセット信号の自動配線を行います．

次に，clk_0 からのクロック信号を各 IP コアに接続します．clk_0 の出力 clk と各 IP コアの入力 clk（または clock）との交点「○」をクリックして「●」に変化させると，両者が接続された状態になります．追加したすべての IP コアに対して，この作業を行ってください．図 4.19 に示す状態になれば，クロック信号の配線は完了です．



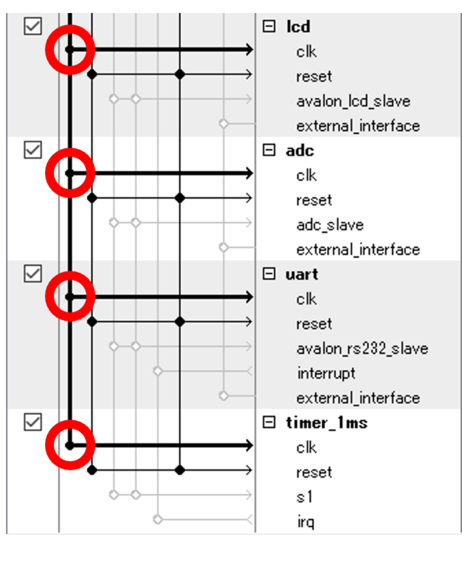


図 4.19　クロック信号の配線

続いて，追加した IP コアを Nios II から制御できるようにするため，`data_master`（データバス）と接続します．図 4.20 に示す状態になれば，データバスとの接続は完了です．

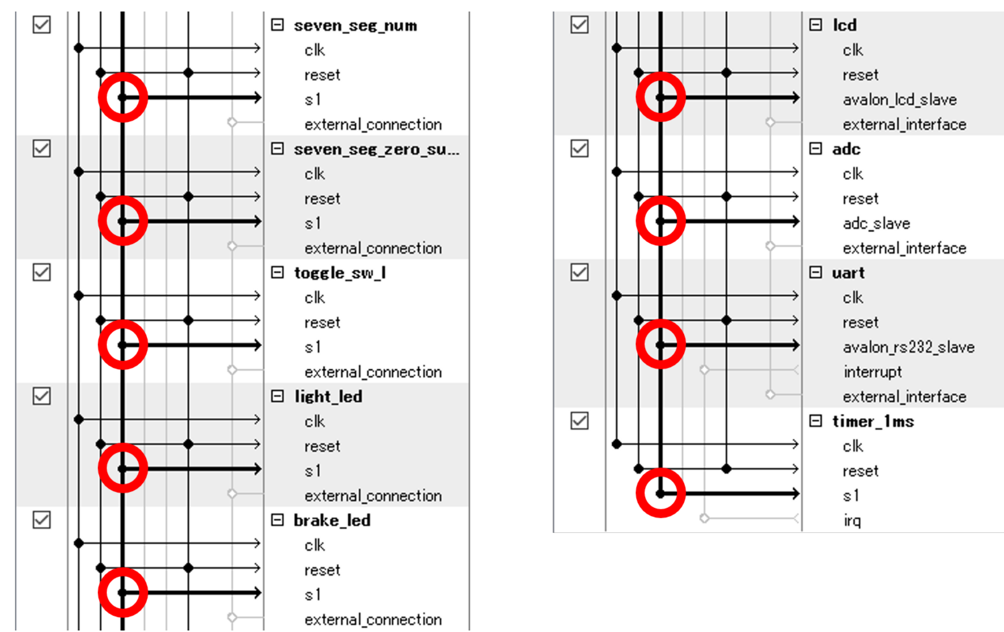


図 4.20　IP コアとデータバスの接続

最後に，RS232 UART およびタイマの IRQ を Nios II に接続します．本実習ではこれらの IRQ を使用しませんが，警告を止めるために必要です．これらの IP コアの「IRQ」列の印をダブルクリックすると，Nios II と接続されます（図 4.21）．

図 4.21　IRQ と Nios II の接続

ポート出力の設定

追加した IP コアを Verilog コードから参照できるようにするため，ポートを外部に出力する設定を行います．追加した各 IP コア（タイマを除く）について，「Conduit」と表示されている行の「Export」列の部分をダブルクリックします．ダブルクリックすると，出力するポートの名前を設定できるようになるので，既定値のまま Enter を押して確定させます．完了後の状態を図 4.22 に示します．

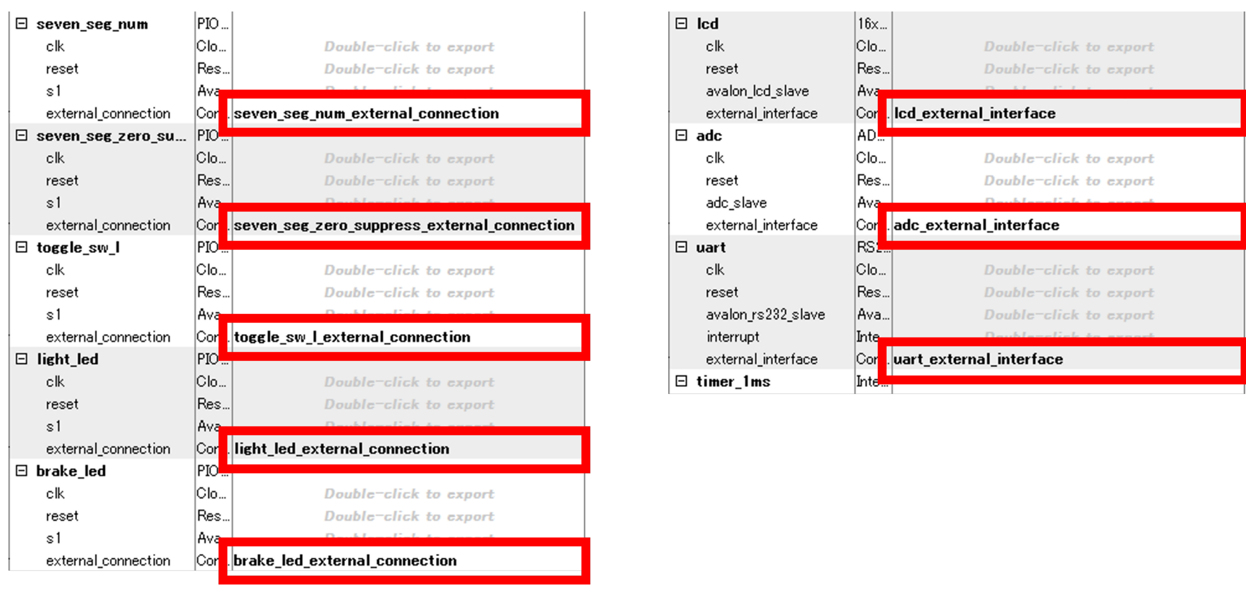


図 4.22　ポート出力設定完了後の状態

オンチップメモリ容量の変更

シリアル通信の処理で使用する入出力関数がメモリを多く使用するため，オンチップメモリの容量を増やします．「onchip_memory_2_0」の上で右クリックし，表示されるメニューから「Edit...」を選択します．設定画面が表示されたら，「Total memory size」を 65536 バイト（64 KB）に設定します（図 4.23）．

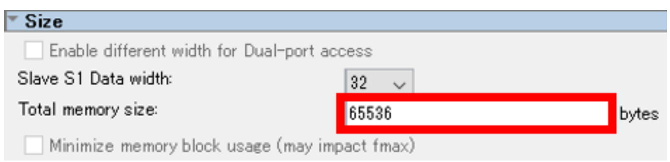


図 4.23　オンチップメモリ容量の変更

アドレスの割り当て

追加した IP コアにアドレスを割り当てます．各 IP コアについて「Base」列のアドレス部分をダブルクリックして先頭アドレスを入力し，Enter を押して確定させます．表 4.5 に設定する値を示します．設定後，「Address Map」タブの内容が図 4.24 のようになったことを確認してください．

表 4.5　各 IP コアの先頭アドレス

IP コア名	信号名	先頭アドレス
seven_seg_num	s1	0x30300
seven_seg_zero_suppress	s1	0x30400
toggle_sw_l	s1	0x30500
light_led	s1	0x30600
brake_led	s1	0x30700
lcd	avalon_lcd_slave	0x31000
adc	adc_slave	0x32000
uart	avalon_rs232_slave	0x33000
timer_1ms	s1	0x40000

System Contents　Address Map　Interconnect Requirements

System: nios2_car_system　Path: timer_1ms.s1

	nios2_gen2_0.data_master	nios2_gen2_0.instruction_master
button_sw.s1	0x0003_0200 - 0x0003_020f	
jtag_uart_0.avalon_jtag_slave	0x0002_0000 - 0x0002_0007	
led.s1	0x0003_0000 - 0x0003_000f	
nios2_gen2_0.debug_mem_slave	0x0000_0800 - 0x0000_0fff	0x0000_0800 - 0x0000_0fff
onchip_memory2_0.s1	0x0001_0000 - 0x0001_3fff	0x0001_0000 - 0x0001_3fff
slide_sw.s1		0x0003_0100 - 0x0003_010f
seven_seg_num.s1	0x0003_0300 - 0x0003_030f	
seven_seg_zero_suppress.s1	0x0003_0400 - 0x0003_040f	
toggle_sw_l.s1	0x0003_0500 - 0x0003_050f	
light_led.s1	0x0003_0600 - 0x0003_060f	
brake_led.s1	0x0003_0700 - 0x0003_070f	
lcd.avalon_lcd_slave	0x0003_1000 - 0x0003_1001	
adc.adc_slave	0x0003_2000 - 0x0003_201f	
uart.avalon_rs232_slave	0x0003_3000 - 0x0003_3007	
timer_1ms.s1	0x0004_0000 - 0x0004_001f	

図 4.24　アドレス割り当て後の Address Map

以上の手順で，Nios II および周辺回路を構成できました．下部の「Messages」ペインにエラーおよび警告が出ていないことを確認して，次の Verilog コードの生成に進んでください．

Verilog コードの生成

構成した Nios II および周辺回路の Verilog コードを生成し，ハードウェアデザインのプロジェクトに追加できるようにします．

メニューから「Generate」→「Generate HDL...」を選択します．続いて HDL 生成の設定画面が表示されますが，ここではそのまま「Generate」ボタンをクリックします．その後，「Save changes to 〜?」と，これまでの構成設定を保存するか確認された場合は，「Save」をクリックして保存します．ファイル名は「`nios2_car_system.qsys`」としてください．

構成設定の保存後，Verilog コードの生成が始まります．正常にコードが生成されればダイアログに「Generate Completed」（生成完了）と表示されますので，「Close」をクリックして閉じます．

Nios II および周辺回路のプロジェクトへの追加

Quartus Prime に戻って，生成したコードをハードウェアデザインのプロジェクトに追加します．

メニューから「Add/Remove files in Project...」を選択します．続いて表示される画面において，「...」ボタンをクリックしてファイル選択画面を表示し，「nios2_car_system\synthesis\nios2_car_system.qip」（プロジェクトのフォルダを起点とした相対パス）を追加します（図 4.25）．

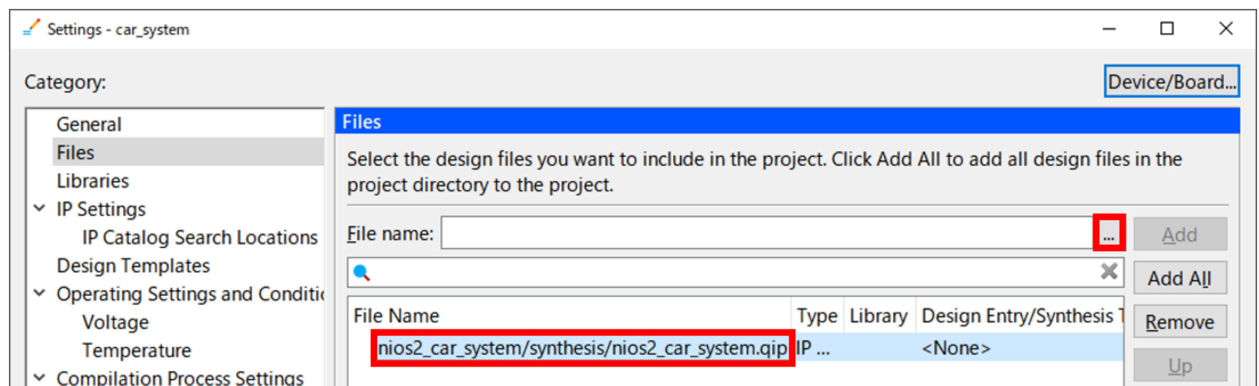


図 4.25　nios2_car_system.qip の追加

4.2.3　最上位階層の作成と論理合成

前項で構成したマイコンシステムをプロジェクトに追加します．また，今回は第 3 章で作成した seven_seg_decoder モジュールおよび toggle_sw モジュールもプロジェクトに追加して使用します．これらのモジュールを最上位階層から呼び出し，システム全体の回路を論理合成します．

第 3 章で作成したモジュールの追加

Windows のエクスプローラーを使用して，第 3 章で作成したモジュールの Verilog ファイル（表 4.6）を，プロジェクトのフォルダ D:\DE1\car_system にコピーします．

表 4.6　第 3 章で作成したモジュールの Verilog ファイル

モジュール	パス	内容
seven_seg_decoder	D:\DE1\seven_seg\seven_seg_decoder.v	7 セグメント LED への数値表示処理
toggle_sw	D:\DE1\blinker_lr\toggle_sw.v	トグルスイッチのチャタリング除去
blinker	D:\DE1\blinker_lr\blinker.v	個々のウィンカー
blinker_lr	D:\DE1\blinker_lr\blinker_lr.v	左右のウィンカー

コピー後，メニューから「Add/Remove files in Project...」を選択します．続いて表示される画面において，「...」ボタンをクリックしてファイル選択画面を表示し，コピーした 2 つの Verilog ファイルを追加します（図 4.26）．

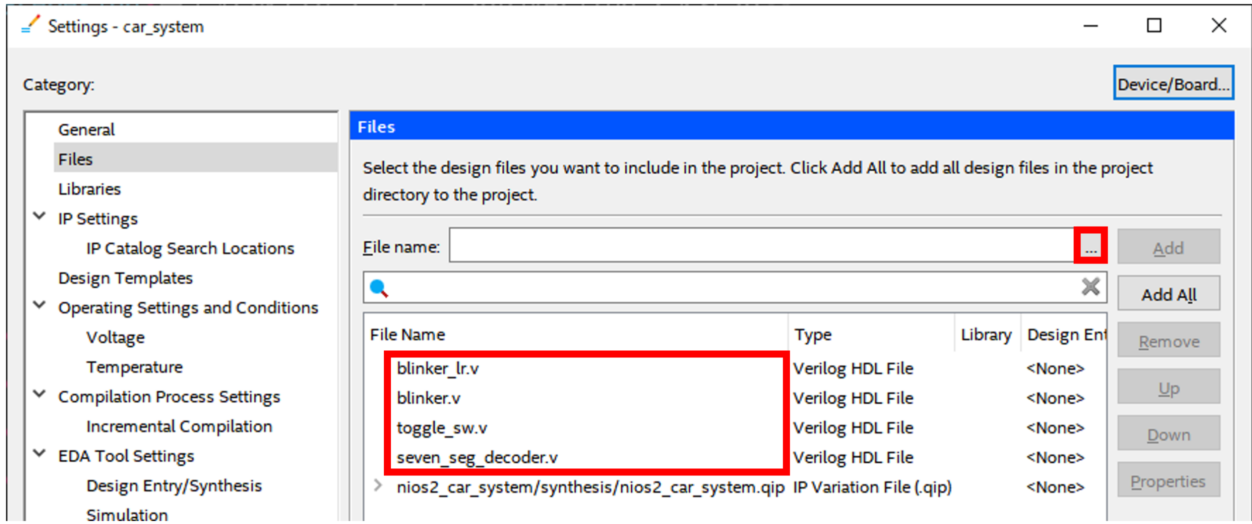


図 4.26　第 3 章で作成したモジュールの追加

最上位階層の作成

最上位階層のモジュール名はプロジェクト名と同じ名前の「car_system」とし，これを Verilog ファイル car_system.v に作成します．このモジュールでは，クロック，リセットおよび入出力装置用のポートを定義し，追加したモジュールと接続します．リスト 4.1 の内容を car_system.v に入力してください．Nios II および周辺回路をまとめたモジュール nios2_car_system については，Platform Designer のメニュー「Generate」→「Show Instantiation Template...」から表示できる記述例（図 4.27）をコピー&ペーストすると，簡単に記述できます．

リスト 4.1　ハードウェアデザインの最上位階層（car_system.v）

```
// 簡易車載システムの最上位階層
module car_system (
  // 50 MHzクロック
  input CLK,
  // リセット（負論理）
  input RESET_N,

  // スライドスイッチ
  input [9:0] SW,
  // ボタンスイッチ（負論理）
  input [3:1] KEY_N,

  // 左トグルスイッチ上側
  input TOGGLE_SW_L_UP,
  // 左トグルスイッチ下側
  input TOGGLE_SW_L_DOWN,

  // 右トグルスイッチ上側
  input TOGGLE_SW_R_UP,
```

```
20    // 右トグルスイッチ下側
21    input TOGGLE_SW_R_DOWN,
22
23    // LED
24    output [9:0] LEDR,
25
26    // 7セグメントLED
27    output [6:0] HEX0,
28    output [6:0] HEX1,
29    output [6:0] HEX2,
30    output [6:0] HEX3,
31    output [6:0] HEX4,
32    output [6:0] HEX5,
33
34    // ヘッドライトLED
35    output LIGHT_LED_L,
36    output LIGHT_LED_R,
37
38    // ウィンカーLED
39    output BLINKER_LED_L,
40    output BLINKER_LED_R,
41
42    // ブレーキLED
43    output BRAKE_LED_L,
44    output BRAKE_LED_R,
45
46    // A/D変換器
47    output ADC_CS_N,
48    input ADC_DOUT,
49    output ADC_DIN,
50    output ADC_SCLK,
51
52    // LCD
53    inout [7:0] LCD_DATA,
54    output LCD_EN,
55    output LCD_RS,
56    output LCD_RW,
57
58    // UART
59    input UART_RXD,
60    output UART_TXD
```

```
61 );
62 
63 // ボタンスイッチの値を反転して正論理に変える
64 wire [2:0] key;
65 assign key = ~KEY_N[3:1];
66 
67 // 7セグメントLED
68 wire [19:0] seven_seg_num;
69 wire seven_seg_zero_suppress;
70 
71 // 左トグルスイッチの入力値
72 wire [1:0] toggle_sw_l_value;
73 
74 // ヘッドライトLEDの出力値
75 wire light_led_value;
76 // ブレーキLEDの出力値
77 wire brake_led_value;
78 
79 // 7セグメントLEDデコーダ
80 seven_seg_decoder seven_seg_decoder_0 (
81   .num           (seven_seg_num),
82   .zero_suppress (seven_seg_zero_suppress),
83   .hex0          (HEX0),
84   .hex1          (HEX1),
85   .hex2          (HEX2),
86   .hex3          (HEX3),
87   .hex4          (HEX4),
88   .hex5          (HEX5)
89 );
90 
91 // 左右ウィンカー
92 blinker_lr b0 (
93   .CLK            (CLK),
94   .RESET_N        (RESET_N),
95   .TOGGLE_SW_UP   (TOGGLE_SW_R_UP),
96   .TOGGLE_SW_DOWN (TOGGLE_SW_R_DOWN),
97   .BLINKER_L      (BLINKER_LED_L),
98   .BLINKER_R      (BLINKER_LED_R)
99 );
100 
101 // 左トグルスイッチ
```

```
102 toggle_sw toggle_sw_l (
103   .clk (CLK),
104   .reset_n (RESET_N),
105   .toggle_sw_up (TOGGLE_SW_L_UP),
106   .toggle_sw_down (TOGGLE_SW_L_DOWN),
107   .out (toggle_sw_l_value)
108 );
109
110 // Nios IIマイコンシステム
111 nios2_car_system u0 (
112   .adc_external_interface_sclk (ADC_SCLK),
113   .adc_external_interface_cs_n (ADC_CS_N),
114   .adc_external_interface_dout (ADC_DOUT),
115   .adc_external_interface_din  (ADC_DIN),
116
117   .brake_led_external_connection_export (brake_led_value),
118   .button_sw_external_connection_export (key),
119   .clk_clk                              (CLK),
120
121   .lcd_external_interface_DATA (LCD_DATA),
122   .lcd_external_interface_ON   (),
123   .lcd_external_interface_BLON (),
124   .lcd_external_interface_EN   (LCD_EN),
125   .lcd_external_interface_RS   (LCD_RS),
126   .lcd_external_interface_RW   (LCD_RW),
127
128   .led_external_connection_export       (LEDR),
129   .light_led_external_connection_export (light_led_value),
130   .reset_reset_n                        (RESET_N),
131
132   .seven_seg_num_external_connection_export           (seven_seg_num),
133   .seven_seg_zero_suppress_external_connection_export (
        seven_seg_zero_suppress),
134
135   .slide_sw_external_connection_export    (SW),
136   .toggle_sw_l_external_connection_export (toggle_sw_l_value),
137
138   .uart_external_interface_RXD (UART_RXD),
139   .uart_external_interface_TXD (UART_TXD)
140 );
141
```

```
142 assign LIGHT_LED_L = light_led_value;
143 assign LIGHT_LED_R = light_led_value;
144
145 assign BRAKE_LED_L = brake_led_value;
146 assign BRAKE_LED_R = brake_led_value;
147
148 endmodule
```

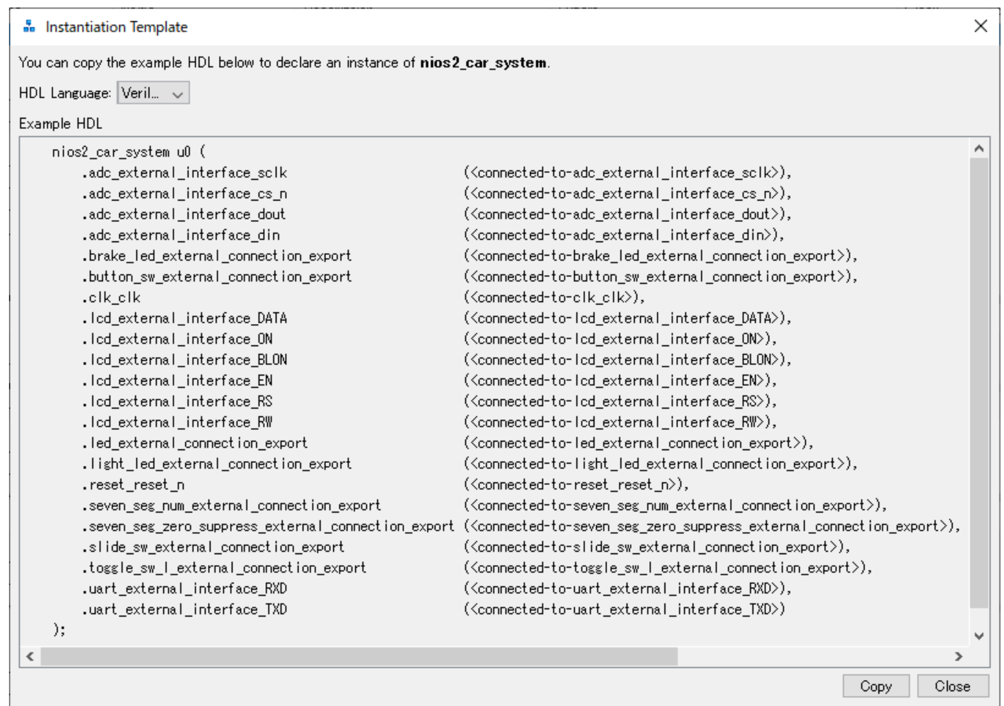


図 4.27　モジュール `nios2_car_system` の記述例のコピー

論理合成

最上位階層の作成後，左側にある「Tasks」ペインの「Analysis & Synthesis」をダブルクリックして，論理合成を行います．論理合成に失敗した場合は，最上位階層のソースコードを確認し，修正してください．

4.2.4　ピンアサイン，コンパイル，プログラミング

論理合成を行った後，ピンアサインとコンパイルを行い，FPGA へプログラミングできるようにします．

ピンアサインで必要になる，信号とピンとの対応を，表 4.7 および表 4.8 に示します．これらを手作業で割り当てることもできますが，量が非常に膨大でありミスしやすいため，今回もピンアサイン設定ファイルの取り込みを行います．

まず，Windows のエクスプローラーを使用して，指導員が配布したピンアサイン設定ファイル `car_system_pin_assignment.qsf` を `D:\DE1\car_system\` 内にコピーします．続いて，Quartus Prime のメニューから「Assignments」→「Import Assignments...」を選択して，`D:\DE1\car_system\`

`car_system_pin_assignment.qsf` を取り込みます．取り込み後，メニューの「Assignments」→「Pin Planner」から Pin Planner を起動して，ピンが割り当てられていることを確認してください．

表 4.7　簡易車載システムのピン割り当て（その 1）

Node Name	Direction	Location	割り当て先の機能
CLK	Input	PIN_AF14	50 MHz クロック
RESET_N	Input	PIN_AA14	ボタンスイッチ KEY0（負論理）
SW[0]	Input	PIN_AB12	スライドスイッチ SW0
SW[1]	Input	PIN_AC12	スライドスイッチ SW1
SW[2]	Input	PIN_AF9	スライドスイッチ SW2
SW[3]	Input	PIN_AF10	スライドスイッチ SW3
SW[4]	Input	PIN_AD11	スライドスイッチ SW4
SW[5]	Input	PIN_AD12	スライドスイッチ SW5
SW[6]	Input	PIN_AE11	スライドスイッチ SW6
SW[7]	Input	PIN_AC9	スライドスイッチ SW7
SW[8]	Input	PIN_AD10	スライドスイッチ SW8
SW[9]	Input	PIN_AE12	スライドスイッチ SW9
KEY_N[1]	Input	PIN_AA15	ボタンスイッチ KEY1（負論理）
KEY_N[2]	Input	PIN_W15	ボタンスイッチ KEY2（負論理）
KEY_N[3]	Input	PIN_Y16	ボタンスイッチ KEY3（負論理）
LEDR[0]	Output	PIN_V16	LEDR0
LEDR[1]	Output	PIN_W16	LEDR1
LEDR[2]	Output	PIN_V17	LEDR2
LEDR[3]	Output	PIN_V18	LEDR3
LEDR[4]	Output	PIN_W17	LEDR4
LEDR[5]	Output	PIN_W19	LEDR5
LEDR[6]	Output	PIN_Y19	LEDR6
LEDR[7]	Output	PIN_W20	LEDR7
LEDR[8]	Output	PIN_W21	LEDR8
LEDR[9]	Output	PIN_Y21	LEDR9
HEX0[0]	Output	PIN_AE26	7 セグメント LED HEX0（負論理）
HEX0[1]	Output	PIN_AE27	
HEX0[2]	Output	PIN_AE28	
HEX0[3]	Output	PIN_AG27	
HEX0[4]	Output	PIN_AF28	
HEX0[5]	Output	PIN_AG28	
HEX0[6]	Output	PIN_AH28	
HEX1[0]	Output	PIN_AJ29	7 セグメント LED HEX1（負論理）
HEX1[1]	Output	PIN_AH29	
HEX1[2]	Output	PIN_AH30	
HEX1[3]	Output	PIN_AG30	
HEX1[4]	Output	PIN_AF29	
HEX1[5]	Output	PIN_AF30	
HEX1[6]	Output	PIN_AD27	
HEX2[0]	Output	PIN_AB23	7 セグメント LED HEX2（負論理）
HEX2[1]	Output	PIN_AE29	
HEX2[2]	Output	PIN_AD29	
HEX2[3]	Output	PIN_AC28	
HEX2[4]	Output	PIN_AD30	
HEX2[5]	Output	PIN_AC29	
HEX2[6]	Output	PIN_AC30	
HEX3[0]	Output	PIN_AD26	7 セグメント LED HEX3（負論理）
HEX3[1]	Output	PIN_AC27	
HEX3[2]	Output	PIN_AD25	
HEX3[3]	Output	PIN_AC25	
HEX3[4]	Output	PIN_AB28	
HEX3[5]	Output	PIN_AB25	
HEX3[6]	Output	PIN_AB22	
HEX4[0]	Output	PIN_AA24	7 セグメント LED HEX4（負論理）
HEX4[1]	Output	PIN_Y23	
HEX4[2]	Output	PIN_Y24	
HEX4[3]	Output	PIN_W22	
HEX4[4]	Output	PIN_W24	
HEX4[5]	Output	PIN_V23	
HEX4[6]	Output	PIN_W25	
HEX5[0]	Output	PIN_V25	7 セグメント LED HEX5（負論理）
HEX5[1]	Output	PIN_AA28	
HEX5[2]	Output	PIN_Y27	
HEX5[3]	Output	PIN_AB27	
HEX5[4]	Output	PIN_AB26	
HEX5[5]	Output	PIN_AA26	
HEX5[6]	Output	PIN_AA25	

表 4.8　簡易車載システムのピン割り当て（その 2）

Node Name	Direction	Location	割り当て先の機能
TOGGLE_SW_L_UP	Input	PIN_AF18	左トグルスイッチ上側
TOGGLE_SW_L_DOWN	Input	PIN_AG18	左トグルスイッチ下側
TOGGLE_SW_R_UP	Input	PIN_AJ21	右トグルスイッチ上側
TOGGLE_SW_R_DOWN	Input	PIN_AG20	右トグルスイッチ下側
LIGHT_LED_L	Output	PIN_AE19	左ヘッドライト LED
LIGHT_LED_R	Output	PIN_AJ20	右ヘッドライト LED
BLINKER_LED_L	Output	PIN_AF21	左ウィンカー LED
BLINKER_LED_R	Output	PIN_AK21	右ウィンカー LED
BRAKE_LED_L	Output	PIN_AG21	左ブレーキ LED
BRAKE_LED_R	Output	PIN_AD20	右ブレーキ LED
LCD_DATA[0]	Bidir	PIN_AK19	キャラクタ LCD
LCD_DATA[1]	Bidir	PIN_AJ19	
LCD_DATA[2]	Bidir	PIN_AK16	
LCD_DATA[3]	Bidir	PIN_AK18	
LCD_DATA[4]	Bidir	PIN_AD17	
LCD_DATA[5]	Bidir	PIN_Y18	
LCD_DATA[6]	Bidir	PIN_AC18	
LCD_DATA[7]	Bidir	PIN_Y17	
LCD_EN	Output	PIN_AJ16	
LCD_RS	Output	PIN_AH17	
LCD_RW	Output	PIN_AJ17	
ADC_CS_N	Output	PIN_AJ4	A/D 変換器
ADC_DIN	Output	PIN_AK4	
ADC_DOUT	Output	PIN_AK3	
ADC_SCLK	Output	PIN_AK2	
UART_RXD	Input	PIN_AG17	UART RXD
UART_TXD	Output	PIN_AE16	UART TXD

ピンを割り当てた後，「Tasks」ペインの「Compile Design」をダブルクリックして回路をコンパイルします．コンパイルが成功したら，2.2.6 項（p. 31〜）の手順に従って FPGA に回路をプログラムしてください．

4.2.5　ウィンカーの動作確認

簡易車載システムの機能のうち，ウィンカーについては，3.2.7 項で製作したウィンカー回路（`blinker_lr` モジュール）が担当しています．最上位階層から正しくモジュール呼び出しを行っていれば，仕様どおり動作することが期待されます．組込みソフトウェアの開発を行う前に，以下に再掲する仕様どおりにウィンカーが動作することを確認しましょう．

仕様

- 右トグルスイッチによってウィンカー LED の点灯状態を切り替えられる．
  - レバー中央：消灯
  - レバー上側：左ウィンカー LED が点滅，右ウィンカー LED が消灯
  - レバー下側：右ウィンカー LED が点滅，左ウィンカー LED が消灯
- ウィンカー LED が点滅する際の周期は 800 ms（400 ms 点灯，400 ms 消灯）とする．
- オンスタート機能（右トグルスイッチのレバーが端に動いた瞬間に，点灯状態から点滅を開始する機能）を持つ．

ウィンカーが正常に動作しない場合は，最上位階層の記述やピンの割り当てを再確認し，修正してください．

4.3　組込みソフトウェアの開発

この節では，4.1 節で示した仕様を満たすように，簡易車載システムの組込みソフトウェアを作成します．

4.3.1　コーディング規約

今回製作する制御システムは，これまでの課題と比べて大規模なものです．仕様も複雑であり，特に C 言語で記述するプログラムの量が非常に多くなります．これに伴い，用意する変数や関数なども多くなるため，プログラムを一定の書き方で記述しなければ可読性や保守性が低くなってしまいます．

そのため，大規模なシステムを製作する際には，プログラムの可読性や保守性の向上を目的として，あらかじめプログラミングのルール（**コーディング規約**）を決めておくことが一般的です．このルールに従う限り，多くの開発者が開発に参加しても一定以上の質でプログラムを記述できます．そこで，この章のシステム構築においても最低限のコーディング規約を定めておくことにします．

命名規則

変数や関数などの名前，すなわち識別子に関するルールを命名規則といいます．簡易車載システムでは，以下のように命名規則を定めます．

- 識別子は，原則として英数字をアンダーバー「_」で区切ったもの（snake_case）にする．

  - 例：uint8_t button_sw; uint8_t button_sw_read(void);
- マクロ定義（#define）や列挙型（enum）で宣言した定数は，大文字の英数字をアンダーバー「_」で区切ったもの（SCREAMING_SNAKE_CASE）にする．
  - 例：#define N_BUTTON_SW 3
- 構造体（struct）や列挙型（enum）には，typedef を用いて短い別名を付ける．この別名は，通常の文字を小文字に，単語の最初の文字を大文字にする（CamelCase）．
  - 例：typedef struct { /* ... */ } ADCValues;
  - 例：typedef enum { /* ... */ } ButtonSWPosition;
- 特定の装置や機能に関連する関数の名前には，一定の接頭辞（表 4.9）を付ける．

表 4.9 関数名に付ける接頭辞

装置・機能	実装場所	接頭辞
ボタンスイッチ	DE1-SoC	button_sw_
LEDR		ledr_
7 セグメント LED		seven_seg_
スライドスイッチ		slide_sw_
ブレーキ LED	追加基板	brake_led_
A/D 変換		car_adc_
キャラクタ LCD		car_lcd_
ヘッドライト LED		light_led_
トグルスイッチ		toggle_sw_
シリアル通信（UART）		uart_

ファイルの分割

大規模な開発では，一般にプログラムが非常に長くなります．従来どおり 1 つのソースファイルに長いプログラムを含めると，処理の理解や，目的とする部分の発見が難しくなります．このような状態は，機能追加や保守の際に大きなコストがかかる原因となります．

以上の問題の発生を防ぐため，大規模な開発では，適切な規模でファイルを分割することが一般的です．ファイルの分割の仕方には様々な考え方がありますが，今回は表 4.10 のように，装置・機能ごとにファイルを分割するようにします．

表 4.10　ファイルの分割

装置・機能	ソースファイル	ヘッダファイル
メイン処理	main.c	
共通の定義		common.h
A/D 変換	adc.c	adc.h
ボタンスイッチ	button_sw.c	button_sw.h
キャラクタ LCD	lcd.c	lcd.h
LED	led.c	led.h
7 セグメント LED	seven_seg.c	seven_seg.h
トグルスイッチ	toggle_sw.c	toggle_sw.h
シリアル通信（UART）	uart.c	uart.h

インクルードガード

プロジェクトを構成するファイルのうち，ヘッダファイルは複数のファイルから何度もインクルードされる可能性があり，その際には多重定義が問題となることがあります[*1]．これを防ぐため，ヘッダファイルにはリスト 4.2 のような**インクルードガード**という定型句を書くのが一般的です．

リスト 4.2　インクルードガード

```
#ifndef BUTTON_SW_H_
#define BUTTON_SW_H_

/* 定義や宣言 */

#endif /* BUTTON_SW_H_ */
```

インクルードガードは，マクロ定義を利用して多重定義を防ぎます．以下では，リスト 4.2 の例を用いて，インクルードガードの仕組みを説明します．初回の読み込みではマクロ BUTTON_SW_H が定義されていないため，プリプロセッサによって#ifndef〜#endif が展開され，マクロ BUTTON_SW_H の宣言後，様々な定義や宣言が行われます．2 回目以降の読み込みでは，既にマクロ BUTTON_SW_H が定義されているため，#ifndef〜#endif の展開は行われません．したがって，定義や宣言は初回読み込み時の 1 回だけ行われることになります．

インクルードガードで使用するマクロの名前については，ヘッダファイル名と合わせておく（定数の命名規則と合わせて SCREAMING_SNAKE_CASE にする，例：button_sw.h → BUTTON_SW_H_）のが一般的です．この実習で使用している Nios II SBT（Eclipse）は，新しいヘッダファイルの作成時にファイル名に合わせて自動で適切なインクルードガードを挿入してくれますので，それを利用します．

4.3.2　プロジェクトの作成

最初に，簡易車載システムの組込みソフトウェアのプロジェクトを作成します．

[*1] 例えば，ヘッダファイルの多重読み込みによって構造体を複数回定義しようとして，コンパイルエラーになるなど．

Quartus Prime のメニューから「Tools」→「Nios II Software Build Tools for Eclipse」を選択して Nios II SBT を起動します．起動時にワークスペース（作業フォルダ）の設定画面が表示された場合は，ハードウェアデザインのフォルダ `D:\DE1\car_system` を指定します．

Nios II SBT が起動したら，メニューから「File」→「New」→「Nios II Application and BSP from Template」を選択して，表 4.11 の設定でプロジェクトを作成します．

表 4.11　簡易車載システムの組込みソフトウェアプロジェクト作成時の設定

項目	値
SOPC Information File name	`D:\DE1\car_system\nios2_car_system.sopcinfo`
Project name	`car_system`
Project template	Hello World Small

4.3.3　BSP の設定

追加したタイマをシステムクロックとして使用するため，BSP（Board Support Package）の設定を変更します．設定には BSP Editor という画面を使用します．左側の「Project Explorer」ペイン内の `car_system` の上で右クリックし，表示されるメニューから「Nios II」→「BSP Editor...」（図 4.28）を選択すると，BSP Editor が表示されます．

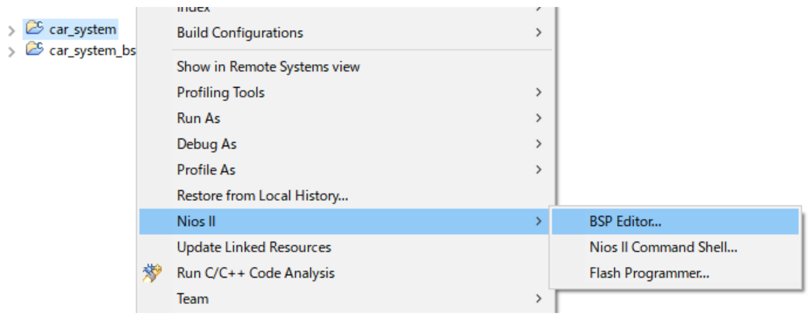


図 4.28　BSP Editor の表示

BSP Editor（図 4.29）が表示されたら，右側の sys_clk_timer 欄を `timer_1ms` に設定します．この設定により，`timer_1ms` を使用して周期的なイベント（アラーム）を発生させることが可能になります．設定後，「Generate」ボタンをクリックして BSP を生成します．BSP を生成したら，「Exit」ボタンをクリックして BSP Editor を終了します．

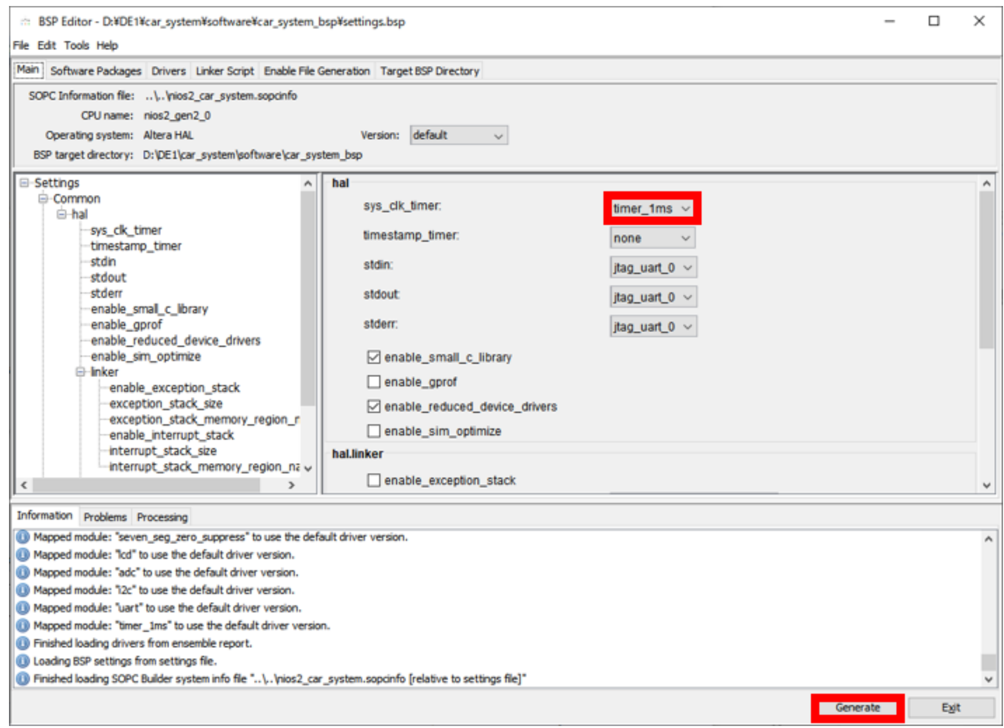


図 4.29　BSP の設定

BSP の設定変更後，アプリケーションプロジェクトをビルドできることを確認しましょう．「Project Explorer」ペインにおいて，`car_system` の上で右クリックし，表示されるメニューから「Build Project」を選択すると，ビルドが実行されます．

4.3.4　プログラムの枠組み

ここから C 言語のプログラムを書いていきます．

プログラムを見通しよく書くため，先にプログラムの枠組みを作ります．これはシステムのメイン処理に相当するため，表 4.10 に従って `main.c` に記述しましょう．作成したプロジェクトには `main()` 関数が含まれている `hello_world_small.c` がありますので，これを `main.c` に改名します．左側の「Project Explorer」ペイン内の `car_system` → `hello_world_small.c` の上で右クリックして，表示されるメニューから「Rename...」を選択します．続いて表示されるダイアログにおいて `main.c` に改名し，「OK」をクリックして確定させます（図 4.30）．

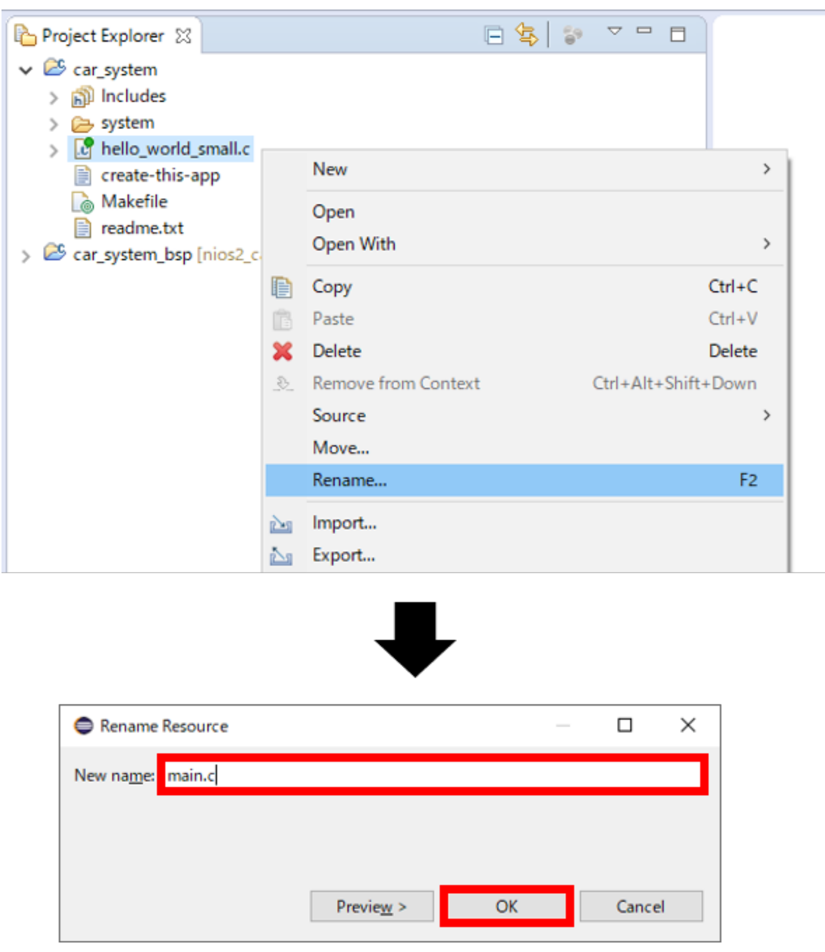


図 4.30　hello_world_small.c の main.c への改名

改名後，main.c の内容をリスト 4.3 のように書き換えてください．

リスト 4.3　main.c：プログラムの枠組み

```
1  #include "system.h"
2
3  #include <stdbool.h>
4  #include <stdint.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  #include "sys/alt_alarm.h"
9  #include "sys/alt_irq.h"
10 #include "sys/alt_stdio.h"
11
12 #include "altera_avalon_pio_regs.h"
13 #include "altera_up_avalon_adc.h"
14 #include "altera_up_avalon_character_lcd.h"
15 #include "altera_up_avalon_rs232.h"
16
17 /* プロトタイプ宣言 */
18 static bool car_system_setup(void);
19 static alt_u32 alarm_1ms_callback(void* context);
20
21 /* 周辺装置の変数宣言 */
```

```
22
23 // LCD
24 static alt_up_character_lcd_dev* lcd = NULL;
25 // A/D変換器
26 static alt_up_adc_dev* adc = NULL;
27 // RS232 UART
28 static alt_up_rs232_dev* uart = NULL;
29 // 1 msアラーム
30 static alt_alarm alarm_1ms;
31
32 /* イベントフラグの変数宣言 */
33
34 // 1 ms経過したか
35 static volatile bool elapsed_1ms = false;
36 // 200 ms経過したか
37 static volatile bool elapsed_200ms = false;
38 // 1 s経過したか
39 static volatile bool elapsed_1s = false;
40
41 /**
42  * @brief メイン処理。
43  * @retval 0      正常終了。
44  * @retval 0以外 異常終了。
45  */
46 int main() {
47   alt_putstr("[car_system]\n");
48
49   if (!car_system_setup()) {
50     // システムの初期設定に失敗した場合、処理を中止する
51     return 1;
52   }
53
54   // メインループ
55   while (true) {
56     // 繰り返し行う処理を書く
57   }
58
59   return 0;
60 }
61
62 /**
63  * @brief 簡易車載システムの初期設定を行う。
64  * @retval true  初期設定に成功した場合。
65  * @retval false 初期設定に失敗した場合。
66  */
67 static bool car_system_setup(void) {
68   // 1 msアラーム開始
69   if (alt_alarm_start(&alarm_1ms, alt_ticks_per_second() / 1000,
70                       alarm_1ms_callback, NULL) < 0) {
71     // 1 msアラームを開始できなかった場合
72     alt_putstr("alarm_1ms: could not start.\n");
```

```
73	    return false;
74	  }
75
76	  return true;
77	}
78
79	/**
80	 * @brief 1 msごとに実行する処理。
81	 * @param context 未使用。
82	 * @return 次に処理を実行するまでのカウント数（1 ms分）。
83	 */
84	static alt_u32 alarm_1ms_callback(void* context) {
85	  static int count_200ms = 0;
86	  static int count_1s = 0;
87
88	  // 1 ms経過フラグセット
89	  elapsed_1ms = true;
90
91	  if (count_200ms >= 199) {
92	    count_200ms = 0;
93
94	    // 200 ms経過フラグセット
95	    elapsed_200ms = true;
96	  } else {
97	    count_200ms++;
98	  }
99
100	  if (count_1s >= 999) {
101	    count_1s = 0;
102
103	    // 1000 ms = 1 s経過フラグセット
104	    elapsed_1s = true;
105	  } else {
106	    count_1s++;
107	  }
108
109	  // 1 ms分のカウント数を返す
110	  return alt_ticks_per_second() / 1000;
111	}
```

以下では各部分の詳細について説明します．

ヘッダファイルの取り込み

冒頭では，標準ライブラリのヘッダファイルに加えて，追加した IP コアに関連するヘッダファイルを取り込みます．詳細については，各機能の実装時に説明します．

今回の開発では，標準ライブラリから，真偽値を扱いやすくする `stdbool.h` を使うようにしてみました．このヘッダファイルは，真偽値を表す `bool` 型（整数型），真を表す `true`（= 1），偽を表す `false`（= 0）を定義します．これらのキーワードは C++ や Java など他の言語でも一般的に使われているため，扱い慣れておくと役立つでしょう．

周辺装置の変数宣言

LCD や A/D 変換器といった周辺装置の IP コアを使用するのに必要な構造体（あるいはそのポインタ）の変数を定義します．これらを使用する前には，専用の関数で初期化する必要があります．初期化の方法については，各機能の実装時に説明します．

なお，ファイル内部でのみ使用する関数やグローバル変数には `static` を付けて，有効範囲をファイル内に限定するのが安全です [12]．他のファイルから意図せず関数の呼び出しやグローバル変数の変更を行ってしまうことを防止できます．

初期設定

周辺装置やタイマ等の初期設定を `car_system_setup()` 関数にまとめて記述します．戻り値は初期設定に成功したかを示す真偽値とし，すべての設定が成功したときに真（`true`）を返すようにします．

この段階では，`car_system_setup()` 関数には後述するタイマの設定のみ記述しています．その他の周辺装置を使用するときに，必要に応じて追記していきます．

メインループ

メインループでは，一定時間が経過した，LCD に表示する内容が変わった，といったイベントごとに対応する処理を記述します．このようなプログラムを**イベント駆動型プログラム**といいます．今回はイベントに対応するフラグ変数（イベントフラグ）を用いてイベント発生の有無を表し，この変数の変化を監視して，対応する処理を記述します．以上の概要をリスト 4.4 に示します．

リスト 4.4　main.c：メインループの概要

```
// メインループ
while (true) {
  if (一定時間経過した) {
    // フラグクリア

    // 一定時間経過したときの処理
  }

  // ...

  if (LCDに表示する内容が変わった) {
    // フラグクリア

    // LCDの表示を更新する処理
  }

  // ...
}
```

時間経過の検出

簡易車載システムが以下の定期的な処理を行うように，タイマを設定します．

- 1 ms 経過時：トグルスイッチ入力やボタンスイッチ入力の変化の検出
- 200 ms 経過時：A/D 変換
- 1 s（1000 ms）経過時：温湿度・気圧モジュールからの値の取得

時間経過を正確に検出するため，`timer_1ms` の割り込みを利用します．1 ms ごとにコールバック関数 `alarm_1ms_callback` を呼び出すように設定し，この関数の中で，各時間の経過についてフラグを立てて通知します．1 ms 以上の計時にはカウンタを使用します．

Nios II の開発では，HAL API[*2]を使うことで，レジスタを操作することなくタイマを使用できます．タイマー関連の代表的な関数を以下に示します．

■ alt_alarm_start()

コールバック関数を登録して，タイマー動作を開始します．

プロトタイプ

```
int alt_alarm_start(alt_alarm* alarm, alt_u32 nticks,
                    alt_u32 (*callback) (void* context),
                    void* context);
```

引数

- `alarm`：アラーム構造体
- `nticks`：コールバック関数を最初に呼び出すまでのカウント数（tick）
- `callback`：時間経過時に呼び出すコールバック関数
- `context`：コールバック関数に引数として渡す値

戻り値 正常に実行できれば `0`．エラーが発生すれば負の値．

■ alt_alarm_stop()

タイマー動作を停止します．

プロトタイプ `void alt_alarm_stop(alt_alarm* alarm);`

引数

- `alarm`：アラーム構造体

戻り値 なし

■ alt_ticks_per_second()

1 秒あたりのカウント数（tick）を返します．

プロトタイプ `alt_u32 alt_ticks_per_second(void);`

引数 なし

戻り値 1 秒あたりのカウント数（tick）

[*2] Hardware Abstruction Layer Application Programming Interface．HAL API はハードウェアを抽象化して，扱いやすいインターフェースを提供します．HAL API を使用すると，ハードウェアの変更に強いプログラムを容易に記述できます．

4.3.5　ヘッドライトの制御

まずは簡単な処理から実装していきましょう．ヘッドライトの制御について，次の仕様を満たすように処理を実装します（自動点灯は，後に A/D 変換とともに実装します）．

仕様

- 左トグルスイッチによってヘッドライト LED の点灯状態（点灯/消灯）を切り替えられる．
  - レバー上側：点灯
  - その他の場所：消灯
- 左右のヘッドライト LED の点灯状態が等しい．

仕様のうち，「左右のヘッドライト LED の点灯状態が等しい」は，リスト 4.1 の以下の部分によって満たされています．

```
assign LIGHT_LED_L = light_led_value;
assign LIGHT_LED_R = light_led_value;
```

したがって，残りの項目「左トグルスイッチによってヘッドライト LED の点灯状態（点灯/消灯）を切り替えられる」を C 言語で実装します．

トグルスイッチの状態の読み取り

トグルスイッチの状態の読み取りについて，`toggle_sw.c` および `toggle_sw.h` に実装します．

まず，ヘッダファイル `toggle_sw.h` を作成します．「Project Explorer」ペイン内の `car_system` の上で右クリックして，表示されるメニューから「New」→「Header File」（図 4.31）を選択します．

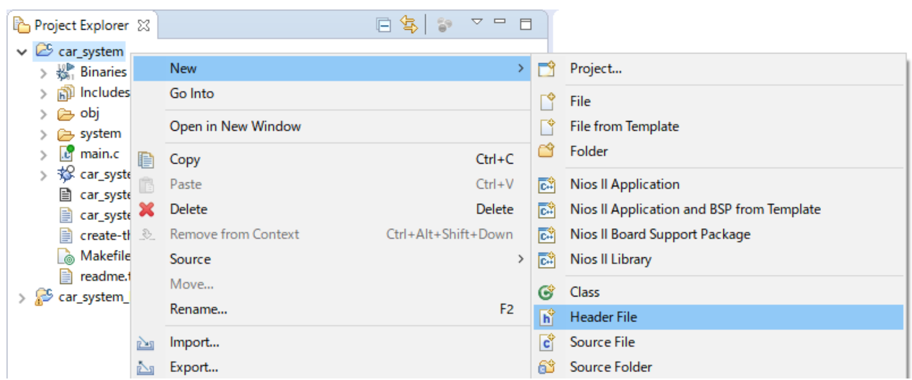


図 4.31　ヘッダファイルの作成

続いて表示される「New Header File」（新規ヘッダファイル）画面において，図 4.32 のように設定して「Finish」ボタンをクリックすると，インクルードガード等が自動的に挿入された `toggle_sw.h` が作成されます．

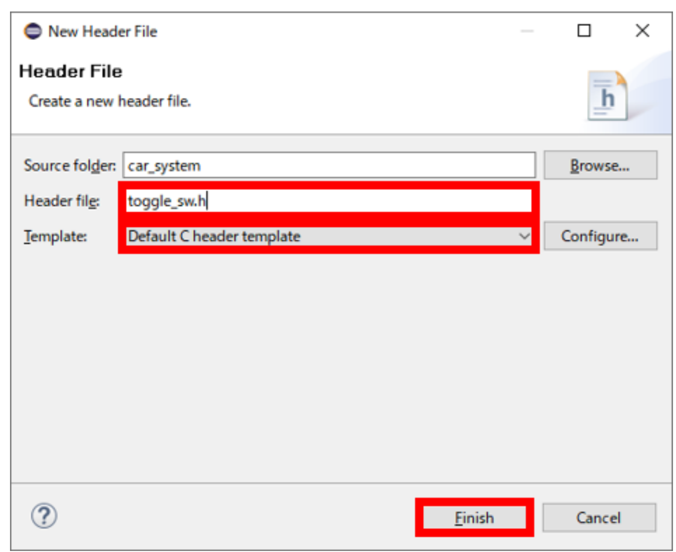


図 4.32　`toggle_sw.h` の作成設定

同様にして，ソースファイル `toggle_sw.c` も作成します．ソースファイルを作成する際には，右クリックによって表示されるメニューから「New」→「Source File」を選択します．続いて表示される設定画面では，図 4.33 のように設定します．

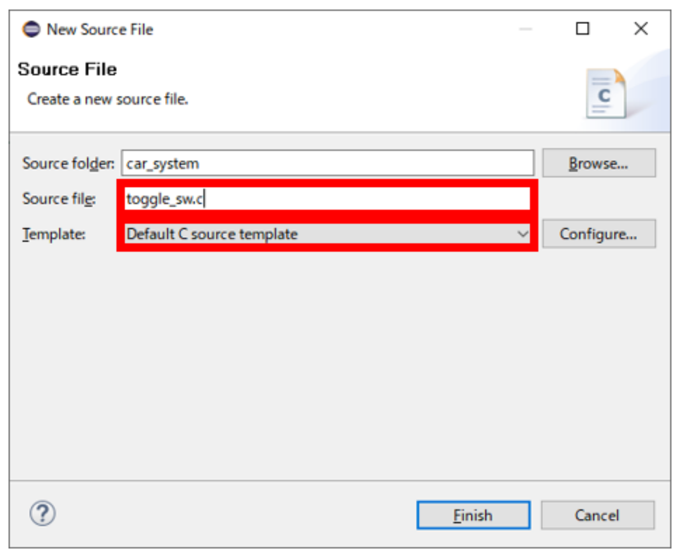


図 4.33　`toggle_sw.c` の作成設定

ファイルを作成したら，以下に示すソースコードを入力してください．

■ toggle_sw.h

リスト 4.5　toggle_sw.h

```
1  /** トグルスイッチ読み取り処理のヘッダ。 */
2
3  #ifndef TOGGLE_SW_H_
4  #define TOGGLE_SW_H_
5
6  #include <stdint.h>
7
```

```
8  /** トグルスイッチのレバーが中央にある。 */
9  #define TOGGLE_SW_CENTER 0x0
10 /** トグルスイッチのレバーが上側にある。 */
11 #define TOGGLE_SW_UP      (0x1 << 0)
12 /** トグルスイッチのレバーが下側にある。 */
13 #define TOGGLE_SW_DOWN    (0x1 << 1)
14
15 uint8_t toggle_sw_l_read(void);
16
17 #endif /* TOGGLE_SW_H_ */
```

■ **toggle_sw.c**

リスト 4.6　toggle_sw.c

```
1  /** トグルスイッチ読み取り処理。 */
2
3  #include "system.h"
4  #include "toggle_sw.h"
5
6  #include <stdint.h>
7
8  #include "altera_avalon_pio_regs.h"
9
10 /**
11  * @brief 左トグルスイッチの状態を読み取る。
12  * @return 左トグルスイッチの状態。
13  */
14 uint8_t toggle_sw_l_read(void) {
15   return IORD_ALTERA_AVALON_PIO_DATA(TOGGLE_SW_L_BASE);
16 }
```

■ **解説**　toggle_sw.c では，左トグルスイッチの状態を読み取る関数 toggle_sw_l_read() を定義します．3.2.8 項で述べたように，トグルスイッチの状態の読み取りには IORD_ALTERA_AVALON_PIO_DATA() マクロを使用します．

ヘッダファイルでは，表 4.12 に示すマクロを定義して，レバー位置を示す値に名前を付けます．このマクロを使用することで，トグルスイッチの状態を利用した制御を分かりやすく書くことができます．

表 4.12　トグルスイッチのレバー位置を示すマクロ

マクロ	値	レバー位置
TOGGLE_SW_CENTER	0x0	中央
TOGGLE_SW_UP	0x1	上側
TOGGLE_SW_DOWN	0x2	下側

ヘッドライトの制御

ヘッドライトの制御について，led.c および led.h に実装します．新規ファイルとして led.h と led.c を作成し，以下に示すソースコードを入力してください．

■led.h

リスト 4.7 led.h：ヘッドライト制御

```
/** LED制御処理のヘッダ。 */

#ifndef LED_H_
#define LED_H_

#include <stdbool.h>

void light_led_update(bool led_on);

#endif /* LED_H_ */
```

■led.c

リスト 4.8 led.c：ヘッドライト制御

```
/** LED制御処理。 */

#include "system.h"
#include "led.h"
#include "toggle_sw.h"

#include <stdbool.h>
#include <stdint.h>

#include "altera_avalon_pio_regs.h"

/**
 * @brief ヘッドライトLEDの状態を更新する。
 * @param led_on LEDを点灯させるか。
 */
void light_led_update(bool led_on) {
  IOWR_ALTERA_AVALON_PIO_DATA(LIGHT_LED_BASE, led_on ? 1 : 0);
}
```

■解説 追加基板上の LED はすべて正論理で動作するため，2.4 節の手法と同様に制御できます．`led.c` には，ヘッドライト LED の点灯状態を制御する `light_led_update()` という関数を定義しました．この関数は，`IOWR_ALTERA_AVALON_PIO_DATA()` マクロを呼んで，ヘッドライト LED への出力を制御します．引数として LED の点灯/消灯を示す `led_on` を設け，LED への出力が確実に `0` または 1 となるようにしています．今後，他の LED に対しても同様の関数を定義していきます．

main.c

`main.c` には，一定時間ごとに実行する処理として，左トグルスイッチの状態の監視およびヘッドライト LED の制御を追加します．トグルスイッチの操作頻度を考慮すると，実行周期はボタンスイッチと同様の数 ms〜数十 ms で問題ないでしょう．今回は，1 ms ごとに上記の処理を実行するように書いてみます．左トグルスイッチのレバーが上側にあるときのみヘッドライト LED を点灯させ，それ以外の場合は消灯させます．

■ ヘッダファイルの取り込み　冒頭に toggle_sw.h および led.h の取り込みを追加します．

```
#include "system.h"
#include "toggle_sw.h" // 追加
#include "led.h"       // 追加

#include <stdbool.h>
// ...
```

■ 1 ms 経過時の処理の追加　1 ms 経過時の処理を，関数 on_elapsed_1ms() として追加します．この関数において，左トグルスイッチの状態を読み取り，ヘッドライト LED の点灯/消灯を切り替えます．

```
/* プロトタイプ宣言 */
// ...
static void on_elapsed_1ms(void);

/* main関数等 */

/** 1 ms経過時の処理。 */
static void on_elapsed_1ms(void) {
  // 左トグルスイッチの状態
  uint8_t toggle_sw_l = toggle_sw_l_read();

  // ヘッドライトLEDを点灯させるか
  bool light_led_on;
  if (toggle_sw_l == TOGGLE_SW_UP) {
    light_led_on = true;
  } else {
    light_led_on = false;
  }

  light_led_update(light_led_on);
}
```

■ メインループ　1 ms 経過時に関数 on_elapsed_1ms() を呼ぶように，メインループを変更します．

```
  // メインループ
  while (true) {
    if (elapsed_1ms) {
      // 1 ms経過したとき

      // フラグクリア
      elapsed_1ms = false;
```

```
      on_elapsed_1ms();
    }
  }
```

動作確認

プログラムを実行して，図 4.34 のように，左トグルスイッチの操作に応じてヘッドライト LED の点灯/消灯が切り替わることを確認してください．

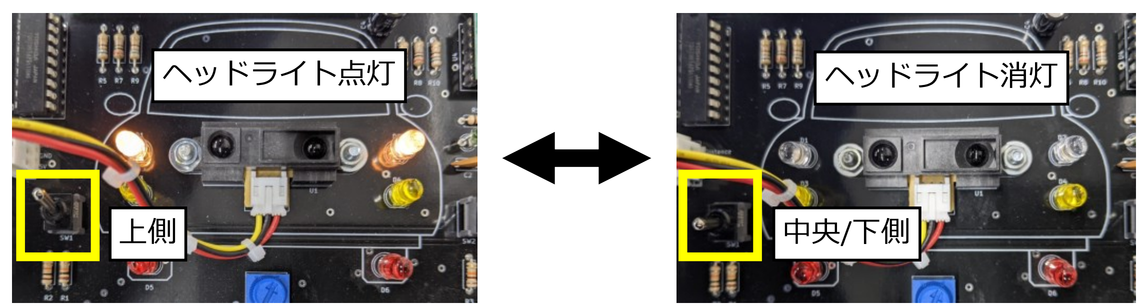


図 4.34　ヘッドライト制御の動作

4.3.6　キャラクタ LCD の制御

ここから，専用 IP コアを用いて周辺装置の制御を行っていきます．最初に，キャラクタ LCD を制御します．

IP コアの概要

キャラクタ LCD コントローラの IP コアは，HD44780 互換のキャラクタ LCD を制御できます．この IP コアは，LCD に対して 8 ビット幅のデータバスを用いてデータを送信し，ビジーフラグを使用しないディレイ方式で処理完了まで待ちます．LCD との通信処理は，ハードウェアとして実装されています．

ライブラリには LCD へのコマンド送信のための関数が用意されており，レジスタを意識せずに LCD を制御できます．これらの関数を使用する際には，`altera_up_avalon_character_lcd.h` をインクルードします．代表的な関数を以下に示します．

■ alt_up_character_lcd_open_dev()

キャラクタ LCD デバイスを開きます．

プロトタイプ

```
alt_up_character_lcd_dev* alt_up_character_lcd_open_dev(
  const char* name);
```

引数

- name：Platform Designer で設定したキャラクタ LCD の名前

戻り値 デバイスの構造体，または NULL（デバイスが見つからない場合）

■ alt_up_character_lcd_init()

表示をクリアすることにより，LCD を初期化します．

プロトタイプ

```
void alt_up_character_lcd_init(alt_up_character_lcd_dev *lcd);
```

引数

- lcd：LCD コントローラデバイスの構造体

戻り値 なし

■ alt_up_character_lcd_set_cursor_pos()

カーソル位置を設定します．

プロトタイプ

```
int alt_up_character_lcd_set_cursor_pos(alt_up_character_lcd_dev *lcd,
                                          unsigned x_pos,
                                          unsigned y_pos);
```

引数

- lcd：LCD コントローラデバイスの構造体
- x_pos：x 座標（0〜15）
- y_pos：y 座標（0：上の行，1：下の行）

戻り値 成功時は 0

■ alt_up_character_lcd_string()

NULL 終端文字列の文字を LCD に書き込みます．

プロトタイプ

```
void alt_up_character_lcd_string(alt_up_character_lcd_dev *lcd,
                                 const char *ptr);
```

引数

- lcd：LCD コントローラデバイスの構造体
- ptr：文字バッファへのポインタ

戻り値 なし

キャラクタ LCD 制御の実装方針

キャラクタ LCD と関連する仕様を以下に再掲します．

仕様

- 2 行分のキャラクタ LCD モジュールに現在の状態を表示できる．
  - 1 行目にはモータの状態（停止中/回転中）を表示する．
    - 停止中：「Motor: stopped」
    - 回転中：「Motor: running」
  - 2 行目には，メータに表示されている項目を表示する．各項目に対応する文字列を表 4.13 に示す．

- KEY1（次の項目）および KEY3（前の項目）を利用して表示項目を切り替えられる．
  - 最初の項目が選択されているときに KEY3 を押すと，最後の項目に切り替わる．
  - 最後の項目が選択されているときに KEY1 を押すと，最初の項目に切り替わる．

表 4.13 キャラクタ LCD に表示する文字列：測定項目（再掲）

番号	測定項目	文字列
0	モータの回転速度設定（%）	`Motor speed (%)`
1	速度調整用可変抵抗の A/D 変換値	`Speed VR`
2	測距センサの A/D 変換値	`Distance`
3	照度センサの A/D 変換値	`Luminance`

仕様から，2 種類の状態（モータが回転中かどうか，および表示中の測定項目）を表示する必要があることが分かります．そのため，`main.c` には，これらの状態を表す変数を用意します．また，LCD の表示更新にはある程度の時間（ms オーダー）が必要であるため，イベントフラグを使用して，状態変化時のみ LCD の表示を更新するようにしてみましょう．このようにすることで，機器の応答速度の低下を防止できます．

ボタンスイッチ入力については，2.6.2 項と同様にエッジ検出を行います．この処理は，`button_sw.c` と `button_sw.h` に記述します．

測定項目の表示は，LCD に加えて，後に 7 セグメント LED に対しても行います．そのため，関連する列挙型などを共通のヘッダファイル `common.h` に記述します．

共通のヘッダファイル

測定項目についての列挙型を，共通のヘッダファイル `common.h` に記述します．新規ファイルとして `common.h` を作成し，以下に示すソースコードを入力してください．

リスト 4.9 common.h

```
1  /** 共通ヘッダ。 */
2
3  #ifndef COMMON_H_
4  #define COMMON_H_
5
6  /** 測定項目を表す型。 */
7  typedef enum {
8    /** モータの回転速度設定 */
9    M_SPEED,
10   /** 速度調整用可変抵抗 */
11   M_SPEED_VR,
12   /** 距離 */
13   M_DISTANCE,
14   /** 明るさ */
15   M_LUMINANCE,
16   /** ダミー：測定項目の個数 */
17   N_MEASUREMENT_ITEMS
18 } MeasurementItem;
```

```
19
20 #endif /* COMMON_H_ */
```

定義した列挙子は int 型の整数値を持ちます（表 4.14）．先頭の M_SPEED が 0 となり，以降は 1 ずつ増加していきます．したがって，これらを 4 進カウンタがとる値と同等に扱うことができます．最後に追加した N_MEASUREMENT_ITEMS は測定項目の個数（4）と等しくなりますので，最後の項目の名前に依存せずに最大値を取り出すことができます．

表 4.14　列挙型 MeasurementItem の列挙子がとる整数値

列挙子	整数値	内容
M_SPEED	0	モータの回転速度設定（%）
M_SPEED_VR	1	速度調整用可変抵抗の A/D 変換値
M_DISTANCE	2	測距センサの A/D 変換値
M_LUMINANCE	3	照度センサの A/D 変換値
N_MEASUREMENT_ITEMS	4	測定項目の個数

キャラクタ LCD の制御

キャラクタ LCD の制御について，lcd.c および lcd.h に実装します．新規ファイルとして lcd.h と lcd.c を作成し，以下に示すソースコードを入力してください．

■**lcd.h**　lcd.h には，キャラクタ LCD にシステムの状態を表示する関数のプロトタイプ宣言を記述します．

リスト 4.10　lcd.h

```
1  /** キャラクタLCD制御処理のヘッダ。 */
2
3  #ifndef LCD_H_
4  #define LCD_H_
5
6  #include "altera_up_avalon_character_lcd.h"
7  #include <stdbool.h>
8
9  #include "common.h"
10
11 void car_lcd_print_motor_status(alt_up_character_lcd_dev* lcd,
12                                 bool motor_running);
13 void car_lcd_print_measurement_item(alt_up_character_lcd_dev* lcd,
14                                     MeasurementItem item);
15
16 #endif /* LCD_H_ */
```

■**lcd.c**　lcd.c には，2 種類の状態（モータが回転中かどうか，および表示中の測定項目）ごとに，キャラクタ LCD の表示更新処理を記述します．古い文字の表示が残らないように，各行の文字列には 16 文字まで空白を埋めておきます．表示中の測定項目については，配列を用いて表 4.13 のデータを用意しておくと，分かりやすく書けます．

リスト 4.11　lcd.c

```
/** キャラクタLCD制御処理。 */

#include "lcd.h"

#include "altera_up_avalon_character_lcd.h"
#include <stdbool.h>

/**
 * @brief LCDの1行目にモータの状態を表示する。
 * @param lcd LCDコントローラデバイスの構造体。
 * @param motor_running モータが回転中か。
 */
void car_lcd_print_motor_status(alt_up_character_lcd_dev* lcd,
                                bool motor_running) {
  const char* line = motor_running ? "Motor: running  " : "Motor: stopped  ";

  alt_up_character_lcd_set_cursor_pos(lcd, 0, 0);
  alt_up_character_lcd_string(lcd, line);
}

/**
 * @brief LCDの2行目に、表示中の測定項目を出力する。
 * @param lcd LCDコントローラデバイスの構造体。
 * @param item 表示中の測定項目。
 */
void car_lcd_print_measurement_item(alt_up_character_lcd_dev* lcd,
                                    MeasurementItem item) {
  // 測定項目と対応する文字列。16文字まで空白で埋める。
  static const char* MeasurementItemLine[] = {
    "Motor speed (%) ",
    "Speed VR        ",
    "Distance        ",
    "Luminance       "
  };
  const char* line = MeasurementItemLine[item];

  alt_up_character_lcd_set_cursor_pos(lcd, 0, 1);
  alt_up_character_lcd_string(lcd, line);
}
```

ボタンスイッチ入力

2.6.2 項と同様に，ボタンスイッチのエッジ検出処理を記述します．新規ファイルとして button_sw.h と button_sw.c を作成し，以下に示すソースコードを入力してください．

■ button_sw.h

リスト 4.12　button_sw.h

```
/** ボタンスイッチ入力処理のヘッダ。 */

```

```
3  #ifndef BUTTON_SW_H_
4  #define BUTTON_SW_H_
5
6  #include <stdbool.h>
7  #include <stdint.h>
8
9  // ボタンスイッチの数
10 #define N_BUTTON_SW 3
11
12 /** ボタンスイッチの位置の型。 */
13 typedef enum {
14   KEY1, KEY2, KEY3
15 } ButtonSWPosition;
16
17 uint8_t button_sw_read(void);
18 void button_sw_update_pressed(uint8_t prev, uint8_t current, bool* pressed);
19
20 #endif /* BUTTON_SW_H_ */
```

■ **button_sw.c**

リスト 4.13　button_sw.c

```
1  /** ボタンスイッチ入力処理。 */
2
3  #include "button_sw.h"
4
5  #include <stdbool.h>
6  #include <stdint.h>
7
8  #include "system.h"
9  #include "altera_avalon_pio_regs.h"
10
11 /**
12  * @brief ボタンスイッチの状態を読み取る。
13  * @return ボタンスイッチの状態。
14  */
15 uint8_t button_sw_read(void) {
16   return IORD_ALTERA_AVALON_PIO_DATA(BUTTON_SW_BASE);
17 }
18
19 /**
20  * @brief ボタンスイッチが押されたか調べ、押下フラグを更新する。
21  * @param prev 前回取得したボタンスイッチレジスタの値。
22  * @param current 今回取得したボタンスイッチレジスタの値。
23  * @param pressed ボタンスイッチ押下フラグの配列。
24  */
25 void button_sw_update_pressed(uint8_t prev, uint8_t current, bool* pressed) {
26   int i;
27   for (i = 0; i < N_BUTTON_SW; i++) {
28     // マスクビット
29     uint8_t mask = 0x1 << i;
```

```
30
31     // 前回取得した値からi番目のビットを取り出す
32     uint8_t prev_masked = prev & mask;
33     // 今回取得した値からi番目のビットを取り出す
34     uint8_t masked = current & mask;
35
36     if ((masked != prev_masked) && (masked != 0x0)) {
37       // ビットが0→1に変化していたら（立ち上がりが発生したら）
38       // ボタンが押されたと判断する
39       pressed[i] = true;
40     } else {
41       pressed[i] = false;
42     }
43   }
44 }
```

main.c

main.c には，キャラクタ LCD の初期化，ボタンスイッチの入力に対応する処理などを追加します．

■ヘッダファイルの取り込み 今回追加したヘッダファイルの取り込みを追加します．

```
#include "led.h"
#include "common.h"    // 追加
#include "button_sw.h" // 追加
#include "lcd.h"       // 追加

#include <stdbool.h>
// ...
```

■プロトタイプ宣言の追加 ボタンスイッチ入力に対応する処理を担う関数のプロトタイプ宣言を追加します．実装は後に行います．

```
/* プロトタイプ宣言 */
// ...
static void on_button_sw_press(bool* button_sw_pressed);
```

■グローバル変数の追加 新しいイベントフラグや，システムの状態を表す変数の宣言を追加します．

```
/* イベントフラグの変数宣言 */
// ...

// モータの状態が変化したか
static volatile bool motor_running_changed = false;
// 表示中の測定項目が変化したか
static volatile bool measurement_item_changed = false;
```

```
/* システムの状態を表す変数の宣言 */

// モータが回転中か
static bool motor_running = false;
// 表示中の測定項目
static MeasurementItem measurement_item = M_SPEED;
```

■キャラクタ LCD 初期化処理の追加 car_system_setup() 関数の冒頭に，キャラクタ LCD の初期化処理を追加します．

```
static bool car_system_setup(void) {
  // キャラクタLCDを初期化する
  lcd = alt_up_character_lcd_open_dev(LCD_NAME);
  if (lcd == NULL) {
    alt_putstr("lcd: open error\n");
    return false;
  }

  alt_up_character_lcd_init(lcd);

  // キャラクタLCDに初期状態を表示する
  car_lcd_print_motor_status(lcd, motor_running);
  car_lcd_print_measurement_item(lcd, measurement_item);

  // 1 msアラーム開始
  // ...
}
```

■ボタンスイッチ入力に対応する処理の追加 ボタンスイッチ入力に対応する処理を追加します．トグルスイッチ入力と同様に，1 ms 周期でエッジ検出を行って，ボタンスイッチが押されたときに対応する処理を記述します．

まず，関数 on_elapsed_1ms() を以下のように変更します．エッジ検出の後，ボタンスイッチ入力に対応する処理を実行します．そして最後に，次のエッジ検出の準備として，ボタンスイッチレジスタの値を記録します．なお，この関数が長くなり過ぎるのを防ぐため，ボタンスイッチ入力に対応する処理を別の関数に分けて記述するようにします．

```
/** 1 ms経過時の処理。 */
static void on_elapsed_1ms(void) {
  // 前回調べたボタンスイッチレジスタの値
  static uint8_t prev_button_sw = 0x0;

  // 左トグルスイッチの状態
  uint8_t toggle_sw_l = toggle_sw_l_read();

  // ボタンスイッチレジスタの値
  uint8_t button_sw = button_sw_read();
  // ボタンスイッチ押下フラグの配列
  bool button_sw_pressed[N_BUTTON_SW] = { 0 };
```

```
  // ボタンスイッチ押下フラグを更新する
  button_sw_update_pressed(prev_button_sw, button_sw, button_sw_pressed);

  // ヘッドライトLEDを点灯させるか
  bool light_led_on;
  if (toggle_sw_l == TOGGLE_SW_UP) {
    light_led_on = true;
  } else {
    light_led_on = false;
  }

  light_led_update(light_led_on);

  // ボタンスイッチ押下に対応する処理を実行する
  on_button_sw_press(button_sw_pressed);

  // ボタンスイッチレジスタの値を記録する
  prev_button_sw = button_sw;
}
```

続いて，関数 on_button_sw_press() を追加して，ボタンスイッチ入力に対応する処理を実装します．モータの回転/停止切り替え（KEY2 押下）では，モータが回転中かどうかを表すグローバル変数の値を反転します．反転後，対応するイベントフラグ motor_running_changed を真に設定して，変更を通知します．

測定項目の表示切り替え（KEY1：次，KEY3：前）は，カウンタの値を増減させる処理と同様です．状態遷移図を図 4.35 に示します．列挙型は int 型と同様に扱えるため，インクリメントやデクリメントによって，表示項目の変更を簡潔に表現できます．範囲外を表示しようとする操作に対しては，反対側の端の項目を表示するように設定しています．表示項目の変更後は，対応するイベントフラグ measurement_item_changed を真に設定して，変更を通知します．

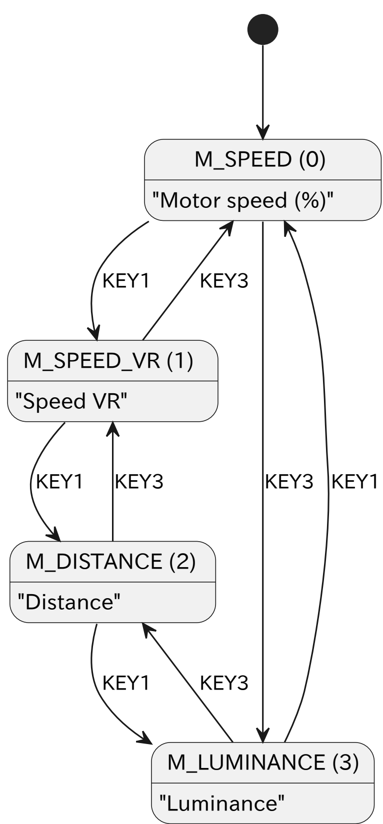


図 4.35　測定項目表示の状態遷移図

```
/**
 * @brief ボタンスイッチ入力に対応する処理。
 * @param button_sw_pressed ボタンスイッチ押下フラグ。
 */
static void on_button_sw_press(bool* button_sw_pressed) {
  if (button_sw_pressed[KEY2]) {
    // KEY2が押されたとき：モータ回転/停止を切り替える
    motor_running = !motor_running;
    motor_running_changed = true;
  } else if (button_sw_pressed[KEY1]) {
    // KEY1が押されたとき

    if (measurement_item >= N_MEASUREMENT_ITEMS - 1) {
      // 最後の測定項目 => 最初の測定項目を表示する
      measurement_item = 0;
    } else {
      // 次の測定項目を表示する
      measurement_item++;
    }

    // 表示中の測定項目が変化したことを通知する
    measurement_item_changed = true;
  } else if (button_sw_pressed[KEY3]) {
    // KEY3が押されたとき

    /* 課題のコードをここに書く */
```

```
  }
}
```

■ メインループ　メインループには，イベントの発生に対応する処理を追加します．モータの回転/停止の変化（motor_running_changed）と，表示する測定項目の変化（measurement_item_changed）のそれぞれについて，発生した場合にキャラクタ LCD の表示を更新する処理を記述します．その際には，**イベントフラグのクリア**を忘れずに記述するようにします．クリアを忘れると，発生時の処理を延々と繰り返してしまうので，注意してください．

```
// メインループ
while (true) {
  if (elapsed_1ms) {
    // 1 ms経過したとき
    // ...
  }

  if (motor_running_changed) {
    // モータの状態が変化したとき

    // フラグクリア
    motor_running_changed = false;

    car_lcd_print_motor_status(lcd, motor_running);
  }

  if (measurement_item_changed) {
    // 表示中の測定項目が変化したとき

    // フラグクリア
    measurement_item_changed = false;

    car_lcd_print_measurement_item(lcd, measurement_item);
  }
}
```

課題 4.1　キャラクタ LCD への状態表示の確認および KEY3 押下時処理の実装

コードを入力して，キャラクタ LCD への状態表示について動作確認してください．
表示する測定項目の切り替えについては，KEY1 押下時の処理（次の項目の表示）のみが実装されています．動作確認後，KEY1 押下時の処理を参考にして，KEY3 押下時の処理（前の項目の表示）を実装してください．

■ ヒント　最後の測定項目は，N_MEASUREMENT_ITEMS - 1 という式で取得できます．

4.3.7　センサと A/D 変換

センサとは，物理的，化学的な現象を電気信号やデータに変換して出力する装置です．電子機器においては，機器の内外の状態を表示したり，制御に必要なデータを収集したりするため，様々なセンサが

使用されます．

簡易車載システムの追加基板にも，いくつかのセンサが実装されています．このうち，アナログ出力のセンサを概説するとともに，A/D 変換によって測定値を取得して，周辺装置の制御に利用する方法についても説明します．

速度調整用可変抵抗器

モータ速度調整用の可変抵抗器が，追加基板の中央下部に実装されています．つまみの位置に応じて，0 V（左端）から 3.3 V（右端）まで，直線的にアナログ電圧を出力します．

測距センサ

測距センサは，自動ブレーキ（障害物が近づいたときに，自動でモータを停止する）に必要となる，対象物との距離を測定するセンサです．追加基板の中央に設置されています（GP2Y0A21YK，シャープ社製）．

このセンサは赤外光を発して，対象物からの反射光を受け取ります．反射光はレンズで集められて，受光素子（位置検出素子）上の 1 点に入射します．対象物との距離によって異なる入射位置を検出して，三角測量の原理によって対象物との距離を測定します [13]（図 4.36）．出力電圧は，概ね対象物との距離が近いほど高くなるという特性を持ちます [14]（図 4.37）．

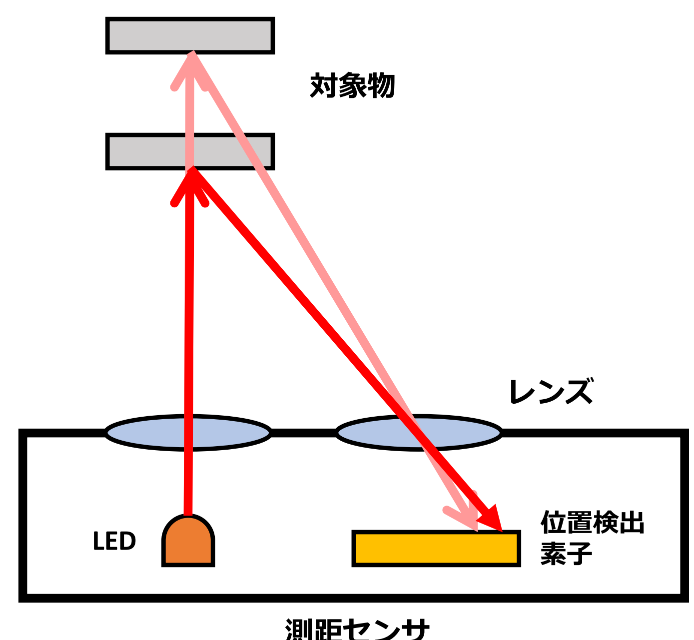


図 4.36　測距センサの測定原理．対象物との距離によって反射光の入射位置が異なることを利用して，距離を算出する．

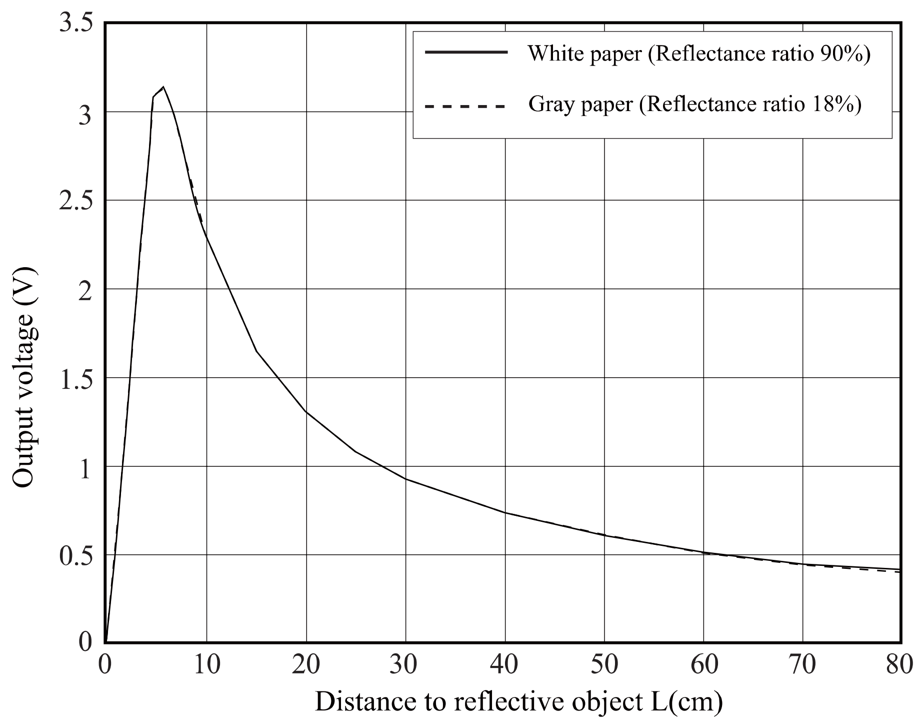


図 4.37　測距センサ GP2Y0A21YK の出力特性（文献 [14] から引用）．横軸：対象物との距離 [cm]，縦軸：出力電圧 [V].

照度センサ

照度センサは，ヘッドライトの自動点灯に必要となる，周囲の明るさを測定するセンサです．追加基板の右側に実装されています（NJL7502L，新日本無線社製）．

NJL7502L は可視光に反応するフォトトランジスタです（図 4.38）．フォトトランジスタの外観は LED と同様（2 端子）ですが，動作はバイポーラトランジスタと似ています．すなわち，バイポーラトランジスタにおけるベース電流の代わりに，入射光の強さによってコレクタ電流が変化します（図 4.39）．

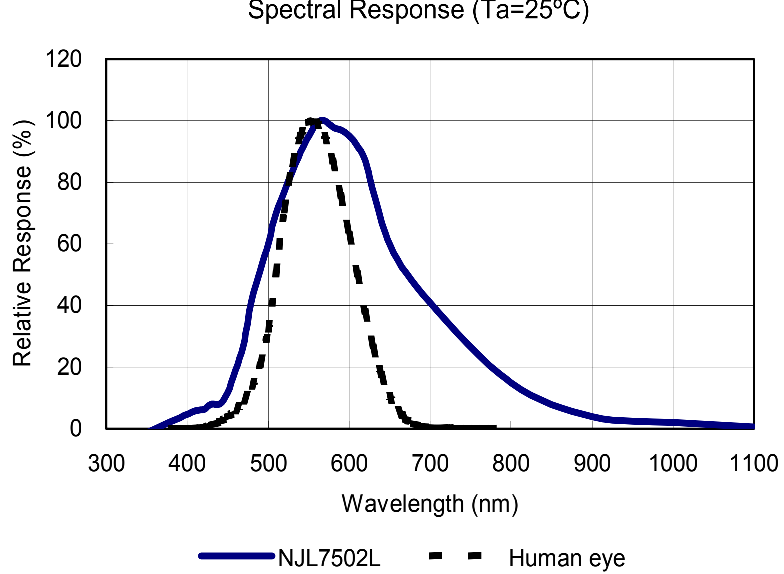


図 4.38　フォトトランジスタ NJL7502L のスペクトル応答（文献 [15] から引用）．横軸：波長 [nm]，縦軸：相対的な応答 [%].

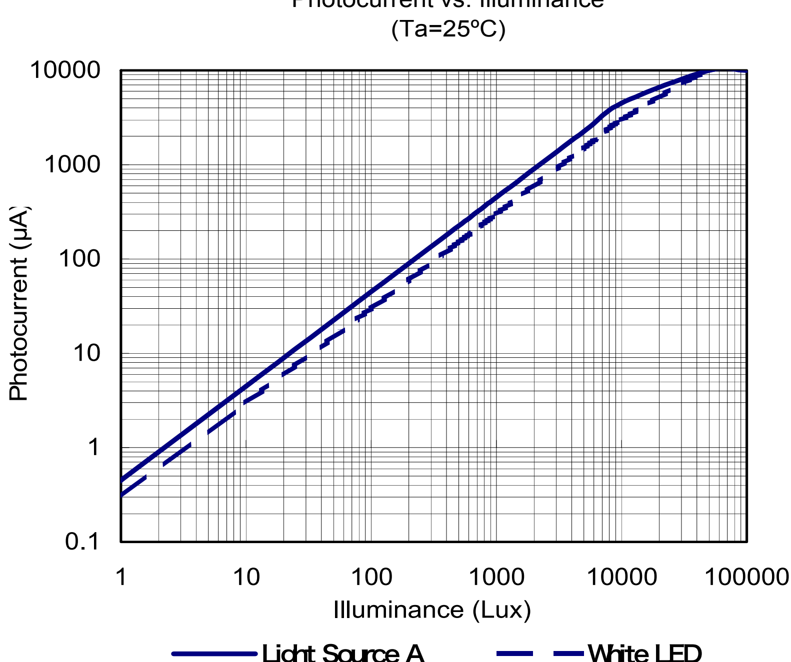


図 4.39　フォトトランジスタ NJL7502L の照度-光電流特性（文献 [15] から引用）．横軸：照度 [lux]，縦軸：光電流 [μA].

追加基板では，光電流を出力電圧に変換する回路として，コレクタ接地回路（図 4.40）を採用しています．コレクタ接地回路ではエミッタ側に抵抗を接続するため，入射光が強い（周囲が明るい）ほど出力電圧が高くなります．

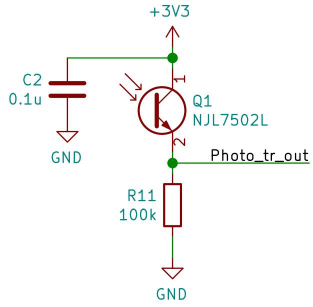


図 4.40　フォトトランジスタのコレクタ接地回路

DE1-SoC の A/D 変換器

通常，FPGA はデジタル信号のみを扱えますが，DE1-SoC には A/D 変換器 LTC2308 が実装されているため，A/D 変換が可能です．LTC2308 は，12 ビット（4096 段階）の分解能で，8 チャンネル（チャンネル 0〜チャンネル 7）分の A/D 変換を行えます．入力端子は DE1-SoC 左下の 2×5 ヘッダで，各チャンネルに 0〜4.096 V のアナログ電圧を入力できます．

追加基板上のセンサのアナログ出力は，表 4.15 に示すチャンネルに接続されています．いずれのセンサについても，出力電圧の最大値は 3.3 V 以下です．

表 4.15　追加基板上のセンサのアナログ出力が接続されているチャンネル

センサ	A/D 変換器のチャンネル
速度調整用可変抵抗器	2
測距センサ	7
照度センサ	0

IP コアの概要

マイコンシステムに組み込んだ A/D 変換器コントローラの IP コアは，DE シリーズに実装された A/D 変換器 IC と通信して，A/D 変換後のデジタル値を取得します．この IP コアのライブラリに用意されている関数を以下に示します．関数を使用する際には，`altera_up_avalon_adc.h` をインクルードします．

■ alt_up_adc_open_dev()

A/D 変換器デバイスを開きます．

プロトタイプ

```
alt_up_adc_dev* alt_up_adc_open_dev(const char* name);
```

引数

- `name`：Platform Designer で設定した A/D 変換器の名前

戻り値 デバイスの構造体，または `NULL`（デバイスが見つからない場合）

■ alt_up_adc_read()

指定されたチャンネルから値を読み取ります．

プロトタイプ

```
unsigned int alt_up_adc_read(alt_up_adc_dev* adc, unsigned channel);
```

引数

- `adc`：A/D 変換器デバイスの構造体
- `channel`：値を読み取るチャンネルの番号

戻り値 A/D 変換器から読み取った 12 ビット値

■ alt_up_adc_auto_enable()

チャンネルの自動変換を有効にします．

プロトタイプ

```
void alt_up_adc_auto_enable(alt_up_adc_dev* adc);
```

引数

- `adc`：A/D 変換器デバイスの構造体

戻り値 なし

A/D 変換関連処理の実装方針

A/D 変換関連処理については，まずメータ機能を実装することにします．関連する仕様を以下に再掲します．

仕様

- 以下の項目の値を 6 桁の 7 セグメント LED に表示できる．
    1. モータの回転速度設定（%）
    2. 速度調整用可変抵抗の A/D 変換値
    3. 測距センサの A/D 変換値
    4. 照度センサの A/D 変換値
- 各項目について，最大値に対する現在の値の割合を LEDR に表示できる（レベルメータ）．
- KEY1（次の項目）および KEY3（前の項目）を利用して表示項目を切り替えられる．
    - 最初の項目が選択されているときに KEY3 を押すと，最後の項目に切り替わる．
    - 最後の項目が選択されているときに KEY1 を押すと，最初の項目に切り替わる．
- 数値を 7 セグメント LED に表示する際，先頭のゼロを省略する．

測定値の急激な変化による影響を小さくするため，今回は 200 ms ごとに A/D 変換するようにしてみます．関連する関数が 4 種類の値をまとめて扱えるように，測定値をまとめる構造体 `ADCValues` を作成して，これを受け渡すようにします．

仕様より，A/D 変換に加えて，7 セグメント LED への数値表示およびレベルメータを実装します．7 セグメント LED については，3.1 節で実装した機能（レジスタに数値を代入するだけで表示可能，ゼロ埋め制御）を活用します．レベルメータの表示方法は，課題 2.6 と同様です．表示するレベルは 1〜10 の 10 段階で指定するようにします．A/D 変換で値を取得した直後に，モータの回転速度（1〜100%）やレベルメータに表示する値（1〜10）の計算も行っておくと，後に制御や表示の処理を簡潔に書けます．

A/D 変換および計算

A/D 変換および各種計算について，`adc.c` および `adc.h` に実装します．新規ファイルとして `adc.h` と `adc.c` を作成し，以下に示すソースコードを入力してください．

■ **`adc.h`**　`adc.h` には，A/D 変換後の値を格納する `ADCValues` 構造体，および A/D 変換を実行する関数のプロトタイプ宣言を記述します．`ADCValues` 構造体には，計算結果として，モータの回転速度（1%〜100%）やレベルメータ用のレベルの値も含めるようにしました．

リスト 4.14　adc.h

```
1  /** A/D変換処理のヘッダ。 */
2
3  #ifndef ADC_H_
4  #define ADC_H_
5
6  #include <stdint.h>
7
8  #include "altera_up_avalon_adc.h"
9
10 /** A/D変換で取得した値を示す型。 */
```

```
11 typedef struct {
12   /** 速度調整用可変抵抗 */
13   uint16_t speed_vr;
14   /** 距離 */
15   uint16_t distance;
16   /** 明るさ */
17   uint16_t luminance;
18 
19   /** モータの回転速度（%） */
20   int speed;
21 
22   /** モータの回転速度のレベル */
23   int speed_level;
24   /** 速度調整用可変抵抗のレベル */
25   int speed_vr_level;
26   /** 距離のレベル */
27   int distance_level;
28   /** 明るさのレベル */
29   int luminance_level;
30 } ADCValues;
31 
32 void adc_get_values(alt_up_adc_dev* adc, ADCValues* values);
33 
34 #endif /* ADC_H_ */
```

■**adc.c**　adc.c には，A/D 変換および計算（モータの回転速度，レベルメータ用のレベル）の処理を記述します．

リスト 4.15　adc.c

```
1  /** A/D変換処理。 */
2  
3  #include "adc.h"
4  
5  #include <stdint.h>
6  
7  #include "altera_up_avalon_adc.h"
8  
9  // 速度調整用可変抵抗を接続したチャンネル
10 #define ADC_CH_SPEED_VR 2
11 // 測距センサを接続したチャンネル
12 #define ADC_CH_DISTANCE 7
13 // 照度センサを接続したチャンネル
14 #define ADC_CH_LUMINANCE 0
15 
16 static uint16_t adc_get_value(alt_up_adc_dev* adc, uint8_t channel);
17 static int adc_value_to_level(unsigned int value, unsigned int max_value);
18 
19 /**
20  * @brief A/D変換で各種センサの値を得る。
21  * @param adc A/D変換器コントローラデバイス。
22  * @param values A/D変換で取得した値の構造体。
```

```
23  */
24 void adc_get_values(alt_up_adc_dev* adc, ADCValues* values) {
25   // 各センサについてA/D変換する
26   values->speed_vr = adc_get_value(adc, ADC_CH_SPEED_VR);
27   values->distance = adc_get_value(adc, ADC_CH_DISTANCE);
28   values->luminance = adc_get_value(adc, ADC_CH_LUMINANCE);
29
30   // モータの回転速度（1%〜100%）を計算する
31   int speed = values->speed_vr * 100 / 3300; // 0〜99
32   values->speed = 1 + speed;                 // 1〜100
33
34   // レベルを計算する
35   values->speed_level = adc_value_to_level(speed, 100);
36   values->speed_vr_level = adc_value_to_level(values->speed_vr, 3300);
37   values->distance_level = adc_value_to_level(values->distance, 3300);
38   values->luminance_level = adc_value_to_level(values->luminance, 3300);
39 }
40
41 /**
42  * @brief A/D変換で指定したチャンネルの値を得る。
43  * @param adc A/D変換器コントローラデバイス。
44  * @param channel チャンネル。
45  * @return A/D変換で得たデジタル値（0〜3299）。
46  */
47 static uint16_t adc_get_value(alt_up_adc_dev* adc, uint8_t channel) {
48   // 指定されたチャンネルの12ビット値を読み取る
49   uint16_t value = alt_up_adc_read(adc, channel) & 0x0FFF;
50
51   // 値を0〜3299に収める
52   if (value >= 3300) {
53     value = 3299;
54   }
55
56   return value;
57 }
58
59 /**
60  * @brief 値をレベルに変換する。
61  * @param value 変換する値（最小値0）。
62  * @param max_value 変換する値の最大値 + 1。
63  * @return レベル（1〜10）。
64  */
65 static int adc_value_to_level(unsigned int value, unsigned int max_value) {
66   int level = value * 10 / max_value;
67   if (level > 9) {
68     level = 9;
69   }
70
71   return level + 1;
72 }
```

関数 adc_get_value は各チャンネルの A/D 変換を担います．12 ビットの値を読み取り，変換後のデジタル値が 0〜3299 が収まるようにしています．

モータの回転速度（1%〜100%）やレベル（1〜10）の計算では，最大値を変更する比例計算が必要になります．一般に，0 以上 M_1 未満の値 x を 0 以上 M_2 未満の値 y に変換するには，次のように計算します．整数同士の除算が含まれるため，結果が常に 0 とならないように，除算は最後に行います．

$$y = x \times M_2 / M_1$$

7 セグメント LED の制御

7 セグメント LED の制御について，seven_seg.c および seven_seg.h に実装します．新規ファイルとして seven_seg.h と seven_seg.c を作成し，以下に示すソースコードを入力してください．

■ **seven_seg.h**　seven_seg.h には，7 セグメント LED を制御する関数のプロトタイプ宣言を記述します．

リスト 4.16　seven_seg.h

```
1  /** 7セグメントLED制御処理のヘッダ。 */
2
3  #ifndef SEVEN_SEG_H_
4  #define SEVEN_SEG_H_
5
6  #include <stdbool.h>
7  #include <stdint.h>
8
9  void seven_seg_set_zero_suppress(bool zero_suppress);
10 void seven_seg_set_num(int32_t num);
11
12 #endif /* SEVEN_SEG_H_ */
```

■ **seven_seg.c**　seven_seg.c には，関数の実装を記述します．表示したい数値から点灯パターンへのデコードとゼロ埋め制御は既にハードウェアとして実装されているため，値を範囲内に収めてレジスタへ代入するだけで制御できます．

リスト 4.17　seven_seg.c

```
1  /** 7セグメントLED制御処理。 */
2
3  #include "seven_seg.h"
4
5  #include <stdbool.h>
6  #include <stdint.h>
7
8  #include "system.h"
9  #include "altera_avalon_pio_regs.h"
10
11 /**
12  * @brief 7セグメントLEDの先頭の0表示の有無を設定する。
13  * @param zero_suppress 先頭の0を表示する => false、非表示 => true。
14  */
```

```
15 void seven_seg_set_zero_suppress(bool zero_suppress) {
16   IOWR_ALTERA_AVALON_PIO_DATA(
17       SEVEN_SEG_ZERO_SUPPRESS_BASE,
18       zero_suppress ? 1 : 0);
19 }
20
21 /**
22  * @brief 7セグメントLEDに表示する数値を設定する。
23  * @param num 7セグメントLEDに表示する数値。
24  */
25 void seven_seg_set_num(int32_t num) {
26   int num_clamped;
27
28   // 表示する数値を0～999999に収める
29   if (num < 0) {
30     num_clamped = 0;
31   } else if (num > 999999) {
32     num_clamped = 999999;
33   } else {
34     num_clamped = num;
35   }
36
37   IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_NUM_BASE, num_clamped);
38 }
```

レベルメータ

LEDR へのレベルメータ表示について，`led.c` および `led.h` に追記します．

■**`led.h`**　`led.h` には，LEDR へのレベルメータ表示の関数のプロトタイプ宣言を追加します．

リスト 4.18　`led.h`：LEDR へのレベルメータ表示

```
void light_led_update(bool led_on);
void ledr_set_level(int level); // 追加
```

■**`led.c`**　`led.c` には，LEDR へのレベルメータ表示の関数を追加します．レベルを 1～10 に制限して，レベル 1 のとき右端の LED のみが，レベル 10 のときすべての LED が点灯するように制御します．

リスト 4.19　`led.c`：LEDR へのレベルメータ表示

```
/**
 * @brief レベルメータに表示するレベルを設定する。
 * @param level レベル（1～10）。
 */
void ledr_set_level(int level) {
  uint16_t pattern = 0x000;
  int level_clamped;
  int i;

  // 表示するレベルを1～10に収める
  if (level < 1) {
```

```
    level_clamped = 1;
  } else if (level > 10) {
    level_clamped = 10;
  } else {
    level_clamped = level;
  }

  for (i = 0; i < level_clamped; i++) {
    pattern |= (0x1 << i);
  }

  IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, pattern);
}
```

main.c

main.c には，A/D 変換器や 7 セグメント LED の初期化，測定値の周辺装置への表示，および定期的に実行する処理を追加します．

■ ヘッダファイルの取り込み　今回追加したヘッダファイルの取り込みを追加します．

```
#include "lcd.h"
#include "adc.h"       // 追加
#include "seven_seg.h" // 追加

#include <stdbool.h>
// ...
```

■ プロトタイプ宣言の追加　定期的に実行する処理，および測定値の周辺装置への表示を行う関数のプロトタイプ宣言を追加します．実装は後に行います．

```
/* プロトタイプ宣言 */
// ...
static void on_elapsed_200ms(void);
static void display_value(void);
```

■ グローバル変数の追加　A/D 変換によって取得した値を格納する構造体変数の宣言を追加します．

```
/* システムの状態を表す変数の宣言 */

// ...

// A/D変換によって取得した値
static ADCValues adc_values = {0};
```

■ A/D 変換器および 7 セグメント LED 初期化処理の追加　car_system_setup() 関数の冒頭に，A/D 変換器および 7 セグメント LED を初期化する処理を追加します．

```
static bool car_system_setup(void) {
  // 7セグメントLEDの初期設定を行う
  seven_seg_set_zero_suppress(true);

  // A/D変換器を初期化する
  adc = alt_up_adc_open_dev(ADC_NAME);
  if (adc == NULL) {
    alt_putstr("adc: open error\n");
    return false;
  }

  alt_up_adc_auto_enable(adc);
  adc_get_values(adc, &adc_values);
  display_value();

  // キャラクタLCDを初期化する
  // ...
}
```

■ 測定値を周辺装置に表示する処理の追加　測定値を周辺装置に表示する関数 `display_value()` を追加します．この関数は，表示中の測定項目に対応する値を `adc_values` から取り出して，7 セグメント LED やレベルメータに出力します．

```
/** 測定値を周辺装置に表示する。 */
static void display_value(void) {
  int32_t seven_seg_value = 0; // 7セグメントLEDに表示する数値
  int level = 0;               // レベルメータに表示するレベル

  switch (measurement_item) {
  case M_SPEED:
    seven_seg_value = adc_values.speed;
    level = adc_values.speed_level;
    break;
  case M_SPEED_VR:
    seven_seg_value = adc_values.speed_vr;
    level = adc_values.speed_vr_level;
    break;
  case M_DISTANCE:
    seven_seg_value = adc_values.distance;
    level = adc_values.distance_level;
    break;
  case M_LUMINANCE:
    seven_seg_value = adc_values.luminance;
    level = adc_values.luminance_level;
    break;
  default:
    break;
  }

  seven_seg_set_num(seven_seg_value);
```

```
  ledr_set_level(level);
}
```

■ 200 ms 経過時の処理の追加　200 ms 経過時の処理を，関数 on_elapsed_200ms() として追加します．この関数では，A/D 変換および測定値の表示を実行します．

```
/** 200 ms経過時の処理。 */
static void on_elapsed_200ms(void) {
  adc_get_values(adc, &adc_values);
  display_value();
}
```

■ メインループ　メインループについては，まず 200 ms 経過時に関数 on_elapsed_200ms() を呼び出すように変更します．

```
    if (elapsed_1ms) {
      // 1 ms経過したとき
      // ...
    }

    if (elapsed_200ms) {
      // 200 ms経過したとき

      // フラグクリア
      elapsed_200ms = false;

      on_elapsed_200ms();
    }

    if (motor_running_changed) {
      // モータの状態が変化したとき
      // ...
    }
```

また，表示中の測定項目が変化したときに，関数 display_value() を呼び出して周辺装置の表示を更新するようにします．

```
    if (measurement_item_changed) {
      // 表示中の測定項目が変化したとき

      // フラグクリア
      measurement_item_changed = false;

      display_value(); // 追加
      car_lcd_print_measurement_item(lcd, measurement_item);
    }
```

動作確認

プログラムを実行し，以下の各項目について，7 セグメント LED およびレベルメータの表示が想定どおりに変化することを確認してください．

- `Motor speed (%)`（モータの回転速度），`Speed VR`（速度調整用可変抵抗）：速度調整用可変抵抗を右に回すと，値が大きくなる．
- `Distance`（測距センサ）：基板の上から手などを近づけると，値が大きくなる．
- `Luminance`（照度センサ）：手で覆うなどして照度センサの周囲を暗くすると，値が小さくなる．

4.3.8　シリアル通信によるモータ制御

速度調整用可変抵抗の出力電圧読み取りが可能になったので，車輪の回転をイメージしたモータ制御を実装していきましょう．関連する仕様を以下に再掲します．

仕様

- 車輪を模したモータ（別の基板に接続）を制御できる．
  - KEY2 を押すことでモータの回転/停止を切り替えられる．
  - 速度調整用可変抵抗を用いてモータの回転速度を制御できる．
- モータの制御にはシリアル通信（UART）を使用する．
- モータの停止中は左右のブレーキ LED を点灯させ，回転中は消灯させる．

制御するモータは，RX マイコンの課題「シリアル通信によるモータ制御」で使用したブラシレス DC モータとします．RX マイコンの課題では，インバータ基板上のマイコンに通信プロトコルを実装して，RS-232C シリアル通信経由でコマンドによるモータ制御ができるようになりました．今回はそのコマンドを利用して，簡易車載システムからシリアル通信によるモータ制御ができるようにすることを目指します（図 4.41）．

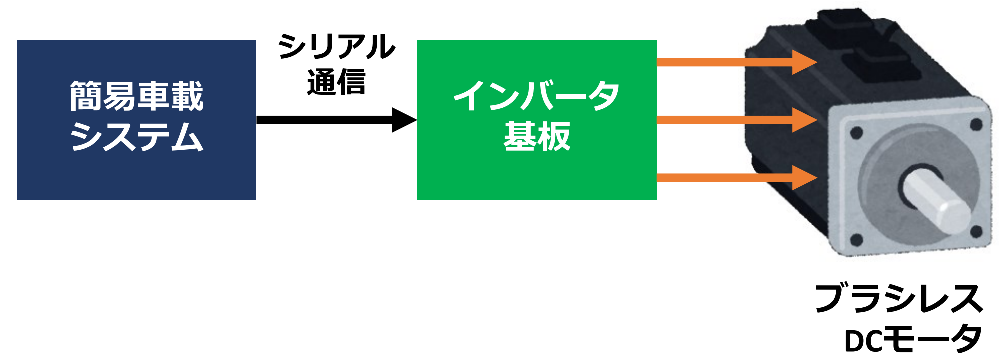


図 4.41　シリアル通信経由のモータ制御．簡易車載システムからインバータ基板に RS-232C シリアル通信経由でコマンドを送信し，ブラシレス DC モータを制御する．

通信プロトコル

インバータ基板との RS-232C シリアル通信の設定を表 4.16 に示します．これらは，マイコンシステムの構成時に IP コア `uart`（RS232 UART）に設定済みです．

表 4.16　インバータ基板との RS-232C シリアル通信の設定

項目	値
ボーレート	38 400 bps
データ長	8 ビット
パリティ	なし
ストップビット長	1 ビット
フロー制御	なし
改行コード	CR+LF

インバータ基板は，以下の通信コマンドを受信します．

- `set_speed` *NNN*<CR><LF>：モータの回転速度を設定する．
  - *NNN*：モータの回転速度（%）．
    - 3 桁の整数 `000`〜`100` で指定する．
    - `000` を指定するとモータが停止する．
    - `100` を指定するとモータが最大速度で回転する．
- `cw`<CR><LF>：モータの回転方向を時計回りに設定する．
- `ccw`<CR><LF>：モータの回転方向を半時計回りに設定する．

機器間の接続

シリアル通信を行えるように，以下の組合せで機器間を接続します．図 4.42 も参考にしてください．

- シリアル通信用コネクタの GND（CN4 ピン 2）↔ インバータ基板の GND（CN2 ピン 40）
- シリアル通信用コネクタの TXD（CN4 ピン 3）↔ インバータ基板の TXD0（CN2 ピン 14）
- USB シリアル通信モジュール ↔ PC の USB コネクタ

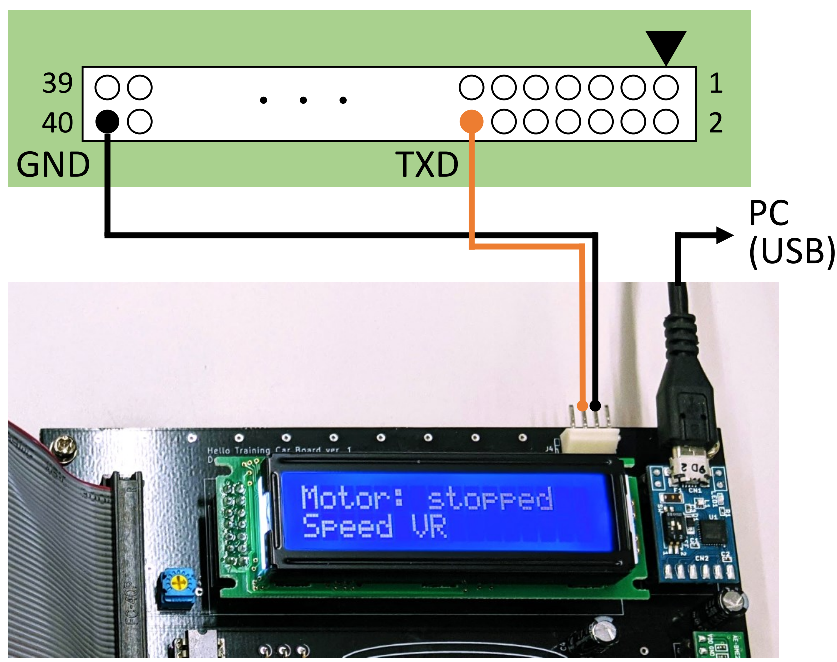


図 4.42 機器間の接続．図の上部のコネクタはインバータ基板の CN2 を示す．PC とは，USB シリアル通信モジュールのマイクロ USB 端子，USB ケーブルを用いて接続する．

IP コアの概要

IP コア「RS232 UART」は，構成設定に従って RS-232C シリアル通信を行います．この IP コアのライブラリに用意されている代表的な関数を以下に示します．関数を使用する際は，`altera_up_avalon_rs232.h` をインクルードします．

■ alt_up_rs232_open_dev()

RS232 デバイスを開きます．

プロトタイプ

```
alt_up_rs232_dev* alt_up_rs232_open_dev(const char* name);
```

引数

- name：Platform Designer で設定したデバイス名

戻り値 デバイスの構造体

■ alt_up_rs232_read_data()

RS232 UART コアから文字データを読み込みます．

プロトタイプ

```
int alt_up_rs232_read_data(alt_up_rs232_dev *rs232,
                           alt_u8 *data,
                           alt_u8 *parity_error);
```

引数

- `rs232`：RS232 デバイスの構造体
- `data`：読み込んだ文字列を格納する場所へのポインタ
- `parity_error`：パリティエラーの値（`0`：正常，`-1`：パリティエラー）を格納する場所

へのポインタ

戻り値 `0`：成功，`-1`：エラー

■alt_up_rs232_write_data()

RS232 UART コアへ文字データを書き込みます．

プロトタイプ

```
int alt_up_rs232_write_data(alt_up_rs232_dev *rs232, alt_u8 data);
```

引数

- rs232：RS232 デバイスの構造体
- data：RS232 UART コアへ転送する文字

戻り値 `0`：成功，`-1`：エラー

モータ制御の実装方針

簡易車載システムにおけるモータの状態（回転中/停止中）に合わせて，インバータ基板に対して set_speed コマンドを送信します．モータ回転中状態の場合は，可変抵抗の出力電圧の A/D 変換で求められた回転速度（1%～100%）を指定して，コマンドを送信します．一方でモータ停止中状態の場合は，回転速度を 0% としてコマンドを送信します．その際，1 回前の回転速度を記憶しておき，変化があった場合のみコマンドを送信すれば，通信量を減らすことができます．

回転速度指定変更に伴うコマンドの送信は，イベントフラグを使用してメインループの後の方で行うようにします．一般に，シリアル通信は時間を要する処理です．そのため，優先順位を低くし，かつ必要なときのみ行うようにすることで，機器の応答速度への影響を小さくします．また，そのイベントが発生したときにブレーキ LED の状態更新も行うようにします．

シリアル通信処理

シリアル通信の処理を uart.c および uart.h に実装します．新規ファイルとして uart.h と uart.c を作成し，以下に示すソースコードを入力してください．

■uart.h　uart.h には，シリアル通信を行う公開関数のプロトタイプ宣言を記述します．ここでは，モータ制御の各コマンドを送信する関数を用意して公開するようにします．

リスト 4.20　uart.h

```
1   /** シリアル通信処理のヘッダ。 */
2
3   #ifndef UART_H_
4   #define UART_H_
5
6   #include "altera_up_avalon_rs232.h"
7
8   void uart_set_speed(alt_up_rs232_dev* uart, int speed);
9   void uart_cw(alt_up_rs232_dev* uart);
10  void uart_ccw(alt_up_rs232_dev* uart);
11
12  #endif /* UART_H_ */
```

■**uart.c** uart.c において，シリアル通信で文字列を送信する関数，およびモータ制御の各コマンドを送信する関数を実装します．

リスト 4.21 uart.c

```
/** シリアル通信処理。 */

#include "uart.h"

#include <stdio.h>
#include "altera_up_avalon_rs232.h"

static void uart_write(alt_up_rs232_dev* uart, const char* s);

/**
 * @brief モータの回転速度を設定する。
 * @param uart RS232デバイスの構造体。
 * @param speed モータの回転速度（0～100）
 */
void uart_set_speed(alt_up_rs232_dev* uart, int speed) {
  int speed_clamped;

  // 値を0～100に収める
  if (speed < 0) {
    speed_clamped = 0;
  } else if (speed > 100) {
    speed_clamped = 100;
  } else {
    speed_clamped = speed;
  }

  // コマンドの文字列: "set_speed NNN\r\n"
  char command[20] = "";
  sprintf(command, "set_speed %03d\r\n", speed_clamped);

  uart_write(uart, command);
}

/**
 * @brief モータの回転方向を時計回りに設定する。
 * @param uart RS232デバイスの構造体。
 */
void uart_cw(alt_up_rs232_dev* uart) {
  /* 課題のコードをここに書く */
}

/**
 * @brief モータの回転方向を反時計回りに設定する。
 * @param uart RS232デバイスの構造体。
 */
void uart_ccw(alt_up_rs232_dev* uart) {
```

```
47    /* 課題のコードをここに書く */
48  }
49
50  /**
51   * @brief シリアル通信で文字列を送信する。
52   * @param uart RS232デバイスの構造体。
53   * @param s 送信する文字列。
54   */
55  static void uart_write(alt_up_rs232_dev* uart, const char* s) {
56    for (const char* p = s; *p != '\0'; p++) {
57      alt_up_rs232_write_data(uart, *p);
58    }
59  }
```

関数 uart_write() は，指定された文字列をシリアル通信で送信します．IP コアのライブラリに用意された関数 alt_up_rs232_write_data() は 1 文字の送信しかできないため，文字列の送信時にはやや不便です．そこで，文字列の各文字を順番に送信する関数 uart_write() を作成して，コマンド送信時にはこちらを利用するようにします．

関数 uart_set_speed() は，モータの回転速度を設定する set_speed コマンドを送信します．整数型の引数で指定された回転速度を 3 桁の整数で表すため，関数 sprintf() を使用します．sprintf() は，printf() と同様に書式指定文字列に従って出力を生成する関数で，出力を文字列に格納します．桁の不足分を 0 で埋めつつ 3 桁の整数を出力するため，書式指定文字列として"%03d"を指定します．

ブレーキ LED の制御

ブレーキ LED の制御について，led.c および led.h に実装します．

■led.h　led.h には，ブレーキ LED を制御する関数のプロトタイプ宣言を追加します．

```
void light_led_update(bool led_on);
void ledr_set_level(int level);
void brake_led_update(bool led_on); // 追加
```

■led.c　led.c において，関数 brake_led_update() を実装します．処理内容はヘッドライト LED の制御と同様です．

```
/**
 * @brief ブレーキLEDの状態を更新する。
 * @param led_on LEDを点灯させるか。
 */
void brake_led_update(bool led_on) {
  IOWR_ALTERA_AVALON_PIO_DATA(BRAKE_LED_BASE, led_on ? 1 : 0);
}
```

main.c

main.c に，実装方針で示した処理を記述します．

■ヘッダファイルの取り込み　今回追加したヘッダファイルの取り込みを追加します．

```
#include "seven_seg.h"
#include "uart.h"       // 追加

#include <stdbool.h>
// ...
```

■イベントフラグの変数宣言　モータの回転速度指定が変化したかを示すイベントフラグ motor_speed_changed を追加します．

```
/* イベントフラグの変数宣言 */
// ...

// モータの回転速度指定が変化したか
// 初期状態をtrueにして，起動時にモータ速度を送信するようにする
static volatile bool motor_speed_changed = true;
```

■システムの状態を表す変数の宣言　モータの回転速度（0〜100）を指定するグローバル変数 motor_speed を追加します．コマンド set_speed を送信する際には，この変数の値を引数として指定します．この変数を用意することで，回転速度指定の変化を検出できるようになります．この変数を変更する際には，イベントフラグ motor_speed_changed のセットも忘れずに行いましょう．

```
/* システムの状態を表す変数の宣言 */
// ...

// モータの回転速度（0〜100）
static int motor_speed = 0;
```

■RS232 UART デバイス初期化処理の追加　car_system_setup() 関数に，RS232 UART デバイスを初期化する処理を追加します．

```
static bool car_system_setup(void) {
  // ...

  // キャラクタLCDに初期状態を表示する
  // ...

  // RS232 UARTを初期化する
  uart = alt_up_rs232_open_dev(UART_NAME);
  if (uart == NULL) {
    alt_putstr("uart: open error\n");
    return false;
  }

  // 1 msアラーム開始
  // ...
}
```

■モータの状態変化時の処理の変更　メインループの「モータの状態が変化したとき」(motor_running_changed) の処理を変更します.

```
    if (motor_running_changed) {
      // モータの状態が変化したとき

      // フラグクリア
      motor_running_changed = false;

      // 状態に合わせてモータの回転速度を設定する
      // 回転中 => 可変抵抗で指定した回転速度
      // 停止中 => 0
      motor_speed = motor_running ? adc_values.speed : 0;
      motor_speed_changed = true;

      car_lcd_print_motor_status(lcd, motor_running);
    }
```

■モータ回転速度指定変化時の処理　メインループに, モータ回転速度指定変化時(motor_speed_changed) の処理を追加します. set_speed コマンドをシリアル通信で送信することに加えて, ブレーキ LED の点灯/消灯切り替えも行います. モータの状態が変化したとき (motor_running_changed) にもモータの回転速度を設定する処理が含まれるため, このイベントのすぐ後に追加するのが良いでしょう.

```
    if (motor_running_changed) {
      // モータの状態が変化したとき
      // ...
    }

    if (motor_speed_changed) {
      // モータの回転速度が変化したとき

      // フラグクリア
      motor_speed_changed = false;

      // ブレーキLED: 回転速度0のときのみ点灯
      brake_led_update(motor_speed == 0);

      uart_set_speed(uart, motor_speed);
    }

    if (measurement_item_changed) {
      // 表示中の測定項目が変化したとき
      // ...
    }
```

■200 ms ごとの処理の変更　200 ms ごとの処理を担う関数 on_elapsed_200ms() を変更します.

```
/** 200 ms経過時の処理。 */
static void on_elapsed_200ms(void) {
  adc_get_values(adc, &adc_values);
```

```
  display_value();

  if (motor_running && motor_speed != adc_values.speed) {
    // モータ回転中、かつ可変抵抗による指定回転速度が変化していた
    // ときのみモータ回転速度を更新する
    motor_speed = adc_values.speed;
    motor_speed_changed = true;
  }
}
```

動作確認

プログラムを実行して，以下に示す動作を確認します．

- システム起動直後に，コマンド `set_speed 000` が送信されてモータが停止する．
- KEY2 を押してモータ回転中状態に移行後，速度調整用可変抵抗を回すと，コマンド `set_speed` が送信されてモータの回転速度が変化する．
- モータ回転中に KEY2 を押してモータ停止中状態に移行すると，コマンド `set_speed 000` が送信されてモータが停止する．

最初は PC を送信先にして，コマンドが正しく送信されることを確認します．追加基板上の USB シリアル通信モジュールと PC を接続して Tera Term を起動し，コマンド `set_speed` が正しく表示されることを確かめてください．

コマンドが正しく送信されていたら，実際にブラシレス DC モータを制御できるか確認します．インバータ基板に直流 24 V の電圧を供給したうえで，CubeSuite+ を用いてシリアル通信によるモータ制御のプログラムをダウンロード，実行します．この状態で上記の操作を行い，モータを制御できることを確かめてください．

4.3.9　各種センサの制御への応用

前項までで，簡易車載システムに基本的な機能を実装できました．最後に，追加基板上のセンサを利用して，近年の自動車に搭載されている安全性向上のための機能を実装してみましょう．

ヘッドライトの自動点灯（オートライト）

近年の自動車には，周囲が暗いときにヘッドライトを自動点灯させる機能（オートライト）が搭載されています．2020 年 4 月以降に販売される新型の乗用車については，搭載が義務化されました．ヘッドライトを点灯させると自動車の視認性が高まるため，交通事故が起こりにくくなります．

自動点灯の原理は，自動車の前端に設置した照度センサで周囲の明るさを検出して，基準よりも暗い場合にヘッドライトを点灯させるというものです．実際の自動車では，木陰や立体交差の下で頻繁に点灯/消灯を繰り返さないようにするなど，より高度な制御が行われています [16].

■ オートライトの実装　ヘッドライトの仕様を以下に再掲します．

仕様

- 左トグルスイッチによってヘッドライト LED の点灯状態（点灯/消灯）を切り替えられる．
  - レバー下側：消灯
  - レバー上側：点灯
  - **レバー中央：自動点灯（周囲が明るいとき消灯，暗いとき点灯）**
- 左右のヘッドライト LED の点灯状態が等しい．

上記の仕様のうち，レバー位置が中央の場合に行う自動点灯処理を main.c に実装します．

まず，冒頭のヘッダファイルの取り込みの後に，自動点灯の基準となる照度センサの A/D 変換値を定義します．現段階では，仮の値として 1500 を設定しておきます．

```
// ...
#include "altera_up_avalon_rs232.h"

// 自動点灯の基準となる照度センサのA/D変換値
#define LUMINANCE_THRESHOLD 1500

/* プロトタイプ宣言 */
// ...
```

続いて，関数 on_elapsed_1ms() 内のヘッドライト LED を制御する部分に，レバー位置が中央の場合の処理を追加します．点灯条件を「周囲が基準よりも暗い」≡「照度センサの A/D 変換値が基準値よりも小さい」とすることで，オートライトを実現できます．

```
/** 1 ms経過時の処理。 */
static void on_elapsed_1ms(void) {
  // ...

  // ヘッドライトLEDを点灯させるか
  bool light_led_on;
  if (toggle_sw_l == TOGGLE_SW_UP) {
    light_led_on = true;
  } else if (toggle_sw_l == TOGGLE_SW_CENTER) {
    // 自動点灯：周囲が基準よりも暗いときにヘッドライトLEDを点灯させる
    light_led_on = (adc_values.luminance < LUMINANCE_THRESHOLD);
  } else {
    light_led_on = false;
  }

  light_led_update(light_led_on);

  // ボタンスイッチ押下に対応する処理を実行する
  // ...
}
```

■オートライトの動作確認　左トグルスイッチのレバー位置を中央にします．まず，追加基板右側の照度センサ（フォトトランジスタ）に光が当たっている状態で，ヘッドライト LED が点灯しない

ことを確認します．続いて，照度センサを手や部品皿などで覆って光が当たらないようにしたときに，ヘッドライト LED が点灯することを確認します．感度を調整したい場合は，冒頭で定義した定数 LUMINANCE_THRESHOLD の値を調整してください．

自動ブレーキ

自動ブレーキも，オートライトと並んで近年の多くの自動車に搭載されている，安全性を向上させるためのシステムです．正式には**衝突被害軽減ブレーキ**や**先進緊急ブレーキシステム**（Advanced Emergency Braking System；**AEBS**）と呼ばれ，導入の義務化が検討されています [17]．

自動ブレーキの原理は，自動車の「目」の役割を担うミリ波レーダーや前方カメラで捉えた情報から衝突の危険を判断し，状況に応じてドライバーに警告する，または自動でブレーキを作動させるというものです [17]．要素技術として，信号処理，画像処理，駆動系との連携等が関連するため，非常に高度な技術といえます．また，開発に当たっては，人工知能（AI）による推論も含む演算の高速化のため，FPGA が採用されています [18]．

■ 自動ブレーキの実装　今回は，簡易車載システムに単純な自動ブレーキ機能を実装してみます．仕様を以下に示します．

仕様

- モータの回転中，追加基板の上方約 10 cm まで障害物を近づけると，モータを停止させる．

追加基板と障害物との距離の測定には，追加基板中央の測距センサを使用します．図 4.37（p. 193）の出力特性から読み取ると，障害物との距離が 10 cm のときの測距センサの出力電圧は約 2.25 V（A/D 変換値では 2250）です．測距センサの A/D 変換値がこの基準値を上回ったときにモータを停止中状態にする（motor_running に false を代入する）ことで，自動ブレーキを実現できます．

まず，main.c 冒頭のヘッダファイルの取り込みの後に，自動ブレーキの基準となる測距センサの A/D 変換値を定義します．

```
// ...
#include "altera_up_avalon_rs232.h"

// 自動点灯の基準となる照度センサのA/D変換値
#define LUMINANCE_THRESHOLD 1500

// 自動ブレーキの基準となる測距センサのA/D変換値（追加）
#define DISTANCE_THRESHOLD 2250

/* プロトタイプ宣言 */
// ...
```

続いて，関数 on_elapsed_200ms() において，A/D 変換を行った後の処理を変更します．

```
/** 200 ms経過時の処理。 */
static void on_elapsed_200ms(void) {
  adc_get_values(adc, &adc_values);
  display_value();
```

```
  if (motor_running) {
    if (adc_values.distance >= DISTANCE_THRESHOLD) {
      // モータ回転中に障害物が基準よりも近づいていたとき、
      // モータを停止させる
      motor_running = false;
      motor_running_changed = true;
    } else if (motor_speed != adc_values.speed) {
      // モータ回転中、かつ可変抵抗による指定回転速度が変化していた
      // ときのみモータ回転速度を更新する
      motor_speed = adc_values.speed;
      motor_speed_changed = true;
    }
  }
}
```

■ 自動ブレーキの動作確認　インバータ基板のプログラムを動作させた状態で，KEY2 を押し，速度調整用可変抵抗を回して，モータを回転させます．この状態で追加基板に手を 10 cm 程度まで近づけると，モータが停止してブレーキ LED が点灯することを確認します．また，手を離した後で KEY2 を押すと，自動ブレーキ発動前と同様にモータが回転することを確認します．

4.4　応用課題

この節では，簡易車載システムの機能を使用して取り組む応用課題を提示します．前節まで構築してきた簡易車載システム本体とは異なり，細かな仕様は指定しません．取り組みやすい課題から，設計・実装に挑戦してみましょう．

課題 4.2　モータの回転方向の切り替え

シリアル通信コマンド `cw` および `ccw`（4.3.8 項，p. 204〜）を用いて，モータの回転開始時に回転方向を指定できるようにしてください．モータの回転方向の指示には，スライドスイッチを利用できます．

課題 4.3　ハザードランプ

左右のウィンカーを同時に点滅させるハザードランプの機能を実装してください．課題 4.2 と同様に，通常のウィンカー動作との切り替えにはスライドスイッチを利用できます．

■ ヒント　Verilog コードの変更のみで実装できます．`car_system.v`（リスト 4.1，p. 158）において，`blinker_lr` モジュールのインスタンス `b0` の入力信号にスライドスイッチの出力信号を含めるのが簡便です．また，トグルスイッチのチャタリング除去を行う `toggle_sw` モジュールを変更して，出力が `2'b11`（レバーが上側と下側の両方にあることに相当）となるようにすることが必要です．

課題 4.4 センサを利用したモータの速度調整

速度調整用可変抵抗器の代わりに測距センサまたは照度センサの値を利用して，モータの速度を設定してください．

■ ヒント センサの値をモータの回転速度設定値 1〜100% に変換して，シリアル通信の `set_speed` コマンドでインバータ基板に伝えます．安全のため，測距センサを用いる場合は対象物までの距離が離れている場合に，また照度センサを用いる場合は周囲が明るい場合に，モータが遅く回転するように値を変換するとよいでしょう．実装できると，モータの回転速度を直感的に設定できます．

課題 4.5　IoT ノード

最近話題の IoT（Internet of Things：モノのインターネット）システムに搭載されている機能の典型例として，機器の状態を表すデータを送信し，遠隔地で集約して可視化する（グラフなどで表す）というものがあります．そこで，簡易車載システムにもこのような機能を追加してみましょう．簡易車載システムに実装されているセンサから得られる値（モータの回転速度（%），測距センサの A/D 変換値，照度センサの A/D 変換値）を PC にシリアル通信で送信し，そのデータを基にしたグラフを描いてください．

■ヒント　1 秒ごと（イベントフラグ：`elapsed_1s`）に状態値をシリアル通信で送信します．モータの回転速度（%）はグローバル変数 `motor_speed` に，A/D 変換値はグローバル変数 `adc_values` に格納されています．

状態値は，カンマ（`,`）で区切って連結した文字列（CSV）で送信します．`sprintf()` 関数を使うと簡単に CSV に整形できます．

例：「通し番号 , モータの回転速度（%）, 距離 , 明るさ<CR><LF>」

```
1,45,398,2804
2,54,715,2698
3,78,1833,2717
4,0,2564,2833     ←自動ブレーキ発動
```

プログラムを実行したら Tera Term でデータを受信し，内容を CSV ファイル（*.csv）として保存します．保存した CSV ファイルを Excel で読み込むと，表として表示されます．Excel の機能でその表をグラフ化すれば，目標としていたグラフを描けます．

4.5　まとめ

この章では，簡易車載システムの開発を通して，周辺装置を制御する技術を学習しました．専用 IP コアも活用することで，キャラクタ LCD，センサ（測距センサ，照度センサ），RS-232C シリアル通信といった様々な周辺装置を制御することができました．また，大規模開発で必要となるコーディング規約（命名規則，ファイルの分割）の考え方や，状態遷移の実装についても触れました．これらには，様々な開発の場面において役立つポイントが多く含まれています．今後の業務においてもぜひ活用してみてください．

参考文献

[1] インテル：“Nios II プロセッサー”, https://www.intel.co.jp/content/www/jp/ja/products/programmable/processor/nios-ii.html（2020 年 9 月 16 日閲覧）.

[2] 堀内伸郎：“アルテラのソフトコア・プロセッサ Nios II の紹介”，TOPPERS プロジェクト，https://www.toppers.jp/press/altera-0704.pdf（2007）.

[3] 堀内伸郎：“FPGA に CPU を内蔵するいくつかのアプローチ”，MONOist，https://monoist.atmarkit.co.jp/mn/articles/1012/27/news104.html（2010）.

[4] マクニカ：“プロセッサ Nios II の概要”，YouTube，https://www.youtube.com/watch?v=HtSidb2dBEk（2020）.

[5] MathWorks：“IP コア”，https://jp.mathworks.com/discovery/ip-core-generation.html（2020 年 9 月 23 日閲覧）.

[6] Terasic: “DE1-SoC Board”, https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=836（2020 年 9 月 24 日閲覧）.

[7] JetBrains：“コードをステップ実行する - 公式ヘルプ | JetBrains Rider “，https://pleiades.io/help/rider/Stepping_Through_the_Program.html（2020 年 10 月 6 日閲覧）.

[8] 小林優：“【改訂 2 版】FPGA ボードで学ぶ組込みシステム開発入門［Intel FPGA 編］”，技術評論社（2018）.

[9] SESSAME WG 2：“組込みソフトウェア開発のための構造化モデリング”，翔泳社（2006）.

[10] 小松洋一：“FPGA の特徴とは？ 他デバイスと比較してみよう”，MONOist，https://monoist.atmarkit.co.jp/mn/articles/1808/02/news001.html（2018）.

[11] マクニカ：“これで分かる！！ プロセッサと FPGA と ASIC の違い”，組込み技術ラボ，https://emb.macnica.co.jp/articles/8204/（2019）.

[12] JPCERT コーディネーションセンター：“DCL19-C. 変数と関数の有効範囲を最小限にする”，https://www.jpcert.or.jp/sc-rules/c-dcl19-c.html（2020）.

[13] 川西信也：“人体検出用途小型薄型測距センサ”，シャープ技報，98 号，pp. 29–31，https://corporate.jp.sharp/rd/33/pdf/98_p29.pdf（2008）.

[14] シャープ：“GP2Y0A21YK”，https://jp.sharp/products/device/doc/opto/gp2y0a21yk_e.pdf（2021 年 4 月 12 日閲覧）.

[15] 新日本無線：“NJL7502L”，https://www.njr.co.jp/electronic_device/PDF/NJL7502L_J.pdf（2013）.

[16] 日産自動車：“インテリジェント オートライトシステム （フロントワイパー連動，薄暮感知機能付）”, https://www.nissan-global.com/JP/TECHNOLOGY/OVERVIEW/smart_auto_headlight_wiper.html（2021 年 4 月 25 日閲覧）.

[17] キーエンス：“自動ブレーキ義務化で何が変わる？ 新しい国際基準について”，https://www.

keyence.co.jp/ss/general/automotive-manufacturing/009/（2021 年 4 月 25 日閲覧）.

[18] 津田健二："FPGA が実現したスバルの新型レヴォーグの自動ブレーキ機能 ", https://news.mynavi.jp/article/car-electronics-137/（2020）.