SUZAKU
スターターキットガイド
(FPGA 開発編)

SZ130-SIL
SZ410-SIL

Version 2.4.5-8d87fa8
2009/07/17

株式会社アットマークテクノ [http://www.atmark-techno.com]

SUZAKU 公式サイト [http://suzaku.atmark-techno.com]

SUZAKU
スターターキットガイド
(FPGA 開発編)

株式会社アットマークテクノ

060-0035 札幌市中央区北 5 条東 2 丁目 AFT ビル 6F
TEL 011-207-6550 FAX 011-207-6570

製作著作 © 2008 Atmark Techno, Inc.

Version 2.4.5-8d87fa8
2009/07/17

前書き

1. はじめに

この度は、「SUZAKU スターターキット」をお買い上げいただきありがとうございます。

本スターターキットは、FPGA 搭載ボード"SUZAKU"を初めて手に取る方にもお使いいただけるよう、第一歩を踏み出すために必要な機材をセットにした学習用キットです。

"SUZAKU"は FPGA(Field Programmable Gate Array)を搭載した組み込み機器開発ボードです。FPGA とは簡単にいうとプログラミングすることができる LSI のことで、さまざまな設計データを送り込んで再構築させることが可能なデバイスです。

この FPGA は近年、より大規模化・低価格化してきています。現在では容易に入手できる FPGA ひとつで、内部にプロセッサと複数の必要な周辺回路を同時に構成するといったことが可能となっています。例えば UART がいくつも欲しい、GPIO ポートが大量に必要だ、画像処理を高速に行うための回路を投入したい、さらに、プロセッサを 2 つ持ちたいといった場合ですら、回路規模が許す限り自由に構成することが可能なのです。

"SUZAKU"は、この FPGA の利点を最大限に生かすべく誕生した小型 FPGA ボードです。

"SUZAKU"の特徴を以下に挙げます。

- 固定された外部インターフェースとして、Ethernet と RS-232C を持っています。
- マイクロコンピュータボードとして動作するために必要な要素であるクロック、DRAM、フラッシュメモリ、Ethernet MAC/Phy、RS-232C ドライバ/レシーバが、基板上に実装されています。
- 電源は+3.3V 単一入力です。内部に FPGA 用の電源である 2.5V、1.2V を作る回路が組み込まれています。また、FPGA を再コンフィギュレーション可能にするための回路が組み込まれています。
- 基板の外周に沿って 86 個(SZ310-U00 は 70 個)の空きピンが備えられています。これらはすべて FPGA の I/O ピンに結線されており、外部デバイスや装置との接続のため自由に使用することができます。
- FPGA の中ではソフトプロセッサ(MicroBlaze)もしくはハードプロセッサ(PowerPC)が動いています。
- フラッシュメモリの中には、OS(Linux)、Ethernet などのデバイスドライバ、アプリケーション群が書き込まれており、電源を入れるだけでこれらを利用することができるようになっています。
- 高機能である Linux を使用しながら、同時にリアルタイム処理を行うような用途向けに構成することも可能です。
- 基板上には SDRAM が 2 枚実装されており、これらを FPGA 内に構成した 2 つの CPU から独立して使用させることができるため、片方で Linux を、他方でリアルタイム OS を動作させる、といった使い方ができます。[1]

[1]SDRAM が 2 枚実装されているのは SZ130-U00 および SZ410-U00 です。

以上のように"SUZAKU"は、FPGA が持つ柔軟性と、Linux が持つ高機能性、豊富なソフトウェア資産等これらの利点を同時に享受することができるプラットフォームです。これらの特徴を利用することにより、旧来の開発手法に比べて開発期間を短縮し、コストダウンを実現することができます。

"SUZAKU"上での開発作業の流れは、

1. FPGA 開発
2. ソフトウェア開発

の 2 段階に大きく分けることができます。本書ではこのうち 1. FPGA 開発について、実際に"SUZAKU スターターキット"を使用しながら解説していきます。2. ソフトウェア開発については、本書と対となる"SUZAKU スターターキットガイド(Linux 開発編)"をご参照ください。

本書を足掛かりとして、SUZAKU 開発者のスペシャリストを目指していただければ幸いです。

1.1. 対象となる読者

本書は SUZAKU の FPGA 開発者向けに書かれた入門書です。SUZAKU の FPGA には初めからプロセッサが搭載されており、ディジタル回路、プロセッサ、バス、メモリ等様々な要素が絡み合ってきます。このため、どこで何が行われているのか分からない、やりたいことがあっても、それを実現するためにはどこをどうすればよいのか分からないという方もおられると思います。

本書では、どこから手をつけていいか分からない方、SUZAKU をはじめて使う方、SUZAKU での FPGA 開発方法について丁寧で分かりやすい説明を望む方を対象としています。

1.2. 対象 CD-ROM

本書は付属 CD-ROM 20080620 以降を対象としております。これ以前の CD-ROM の場合は最新版のデータ一式を SUZAKU 公式サイトのダウンロードページ [http://suzaku.atmark-techno.com/downloads/all]からダウンロードするか、v2.4.0 以前のバージョンのスターターキットガイドをご参照ください。

1.3. 本書の構成

本書では、SUZAKU スターターキットを使用してスロットマシンを製作しながら、SUZAKU の使い方について解説してきます。内容は 3 部構成となっています。

第 1 部では SUZAKU で FPGA 開発を行うために必要な知識や準備について説明をします。SUZAKU および LED/SW ボードについて(第 1、2、3 章)と、作業の前に必要な準備と簡単な SUZAKU の使い方(第 4、5、6 章)を説明します。

第 2 部では Xilinx の ISE というツールを用い、プロセッサを含まない FPGA の開発を実際に SUZAKU スターターキットを用いて体験します。まず、単色 LED を 1 つだけ点灯する簡単な回路の作成から始まり(第 7 章)、組み合わせ回路、順次回路の説明をし(第 8 章)、これらを踏まえて、スロットマシンの要素となる回路(単色 LED 順次点灯回路、7 セグメント LED デコーダ回路、ダイナミック点灯回路)を作成していきます(第 9 章)。

第 3 部では Xilinx の EDK というツールを用い、プロセッサを含んだ FPGA の開発を実際に SUZAKU スターターキットを用いて体験します。まずは EDK がどのようなツールであるのかを説明し、SUZAKU のデフォルトの構成を説明します(第 10 章)。その後、ISE で作成した回路を IP コアにして SUZAKU と接続し、スロットマシンを完成させます(第 11 章)。最後に、こんなこともやってみよう、という例を示します(第 12 章)。

スロットマシンを最後までつくり上げる頃には、SUZAKU の効果的な使い方を学んでいただけるのではないかと思います。

1.4. 表記について

本書は SUZAKU-S(SZ010-U00、SZ030-U00、SZ130-U00)、SUZAKU-V(SZ310-U00、SZ410-U00)を対象に書かれています。内容によってはすべての SUZAKU に当てはまらない場合がございます。当てはまらないものがある場合は以下の記号で対象となる SUZAKU を示します。

SZ010 SZ030 SZ130 SZ310 SZ410

また、これ以降型番の-U00 を省略して表記します。

本書では以下のようにフォントを使っています。

フォント例	説明
本文中のフォント	本文
SUZAKU %	プロンプトの文字列
std_logic_vector	VHDL 記述
#include	C 言語記述

2. 注意事項

2.1. 安全に関する注意事項

SUZAKU スターターキットを安全にご使用いただくために、特に以下の点にご注意くださいますようお願いいたします。

本製品には一般電子機器用(ＯＡ機器・通信機器・計測機器・工作機械等)に製造された半導体部品を使用していますので、その誤作動や故障が直接生命を脅かしたり、身体・財産等に危害を及ぼす恐れのある装置(医療機器・交通機器・燃焼制御・安全装置等)に組み込んで使用したりしないでください。また、半導体部品を使用した製品は、外来ノイズやサージにより誤作動したり故障したりする可能性があります。ご使用になる場合は万一誤作動、故障した場合においても生命・身体・財産等が侵害されることのないよう、装置としての安全設計(リミットスイッチやヒューズ・ブレーカ等の保護回路の設置、装置の多重化等)に万全を期されますようお願い申しあげます。

発熱により高温になる部品があります。周囲温度や取り扱いによってはやけどの恐れがあります。電源が入っている状態および電源切断後しばらくは本製品に触れないようお願い申しあげます。

2.2. 保証に関する注意事項

- 製品保証範囲について

- 付属品(ソフトウェアを含みます)を使用し、取扱説明書、各注意事項に基づく正常なご使用に限り有効です。万一正常なご使用のもと製品が故障した場合は、初期不良保証期間内であれば新品交換をさせていただきます。
- 保証対象外になる場合
- 次のような場合の故障・損傷は、保証期間内であっても保証対象外になります。
- 1. 取扱説明書記載の使用方法、または注意に反したお取り扱いによる場合
  2. 改造・調整や部品交換による場合。または正規のものを使用していないか、あるいは過去に使用されていた場合
  3. お客様のお手元に渡った後の輸送、移動時の落下等お取り扱いの不備による場合
  4. 火災・地震・水害・落雷・その他の天災、公害や異常電圧による場合
  5. アダプタ・ケーブル等の付属品について、同梱のものを使用していない場合
  6. 付属品がすべて揃っていない場合
- 免責事項
- 弊社に故意または重大な過失があった場合を除き、製品の使用および、故障、修理によって発生するいかなる損害についても、一切の責任を負わないものとします。

本製品は購入時の初期不良以外の保証を行っておりません。保証期間は商品到着後 2 週間です。本製品をご購入しましたらお手数でも必ず動作確認を行ってからご使用ください。本製品に対して注意事項を守らずに発生した故障につきましては保証対象外となります。

2.3. 取り扱い上の注意事項

劣化、破損、誤動作、発煙、発火の原因となることがあります。取り扱い時には以下のような点にご注意ください。

- 入力電源

  5V+5%以上の電圧を入力する、極性を間違う等しないでください。また、SUZAKU の＋ 3.3V 外部入力(CON6)に電源を供給しないでください。

- インターフェース

  各インターフェース(外部 I/O、RS-232C、Ethernet、JTAG)には規定以外の信号を接続しないでください。また、信号の極性、入出力方向を間違わないでください。

- 本製品の改造

  本製品について改造を行った場合は保証対象外となりますので、十分にご注意ください。(※コネクタ非搭載箇所へのコネクタの増設を除く。)

  コネクタを増設する際にはマスキングを行い、周囲の部品に半田くず、半田ボール等付着しない様十分にご注意ください。

なお、改造を行う場合は、改造前の動作確認を必ず行うようお願いします。

- FPGA プログラム

  周辺回路(ボード上の部品も含む)と信号の衝突(同じ信号に 2 つのデバイスから出力する)を起こすような FPGA プログラムを行わないでください。FPGA のプログラムを間違わないでください。

- 電源の投入

  本ボードや周辺回路に電源が入っている状態では絶対に FPGA I/O、JTAG 用コネクタの着脱を行わないでください。

- 静電気

  本ボードには CMOS デバイスを使用していますので、ご使用になるまでは帯電防止対策のされている、出荷時のパッケージ等にて保管してください。

- ラッチアップ

  電源および入出力からの過大なノイズやサージ、電源電圧の急激な変動等で、使用している CMOS デバイスがラッチアップを起こす可能性があります。いったんラッチアップ状態となると、電源を切断しないかぎりこの状態が維持されるため、デバイスの破損につながることがあります。ノイズの影響を受けやすい入出力ラインには保護回路を入れる、ノイズ源となる装置と共通の電源を使用しない等の対策をとることをお勧めします。

- 衝撃、振動

  落下や衝突などの強い衝撃や、強い振動、遠心力を与えないでください。振動部や回転部などへの搭載はしないでください。

- 高温低温、多湿

  極度に高温や低温になる環境や、湿度が高い環境では使用しないでください。

- 塵埃

  塵埃の多い環境では使用しないでください。

2.4. FPGA 使用に関しての注意事項

本製品に含まれる FPGA プロジェクト(付属のドキュメント等も含む)は、現状のまま(AS IS)提供されるものであり、特定の目的に適合することや、その信頼性、正確性を保証するものではありません。また、本製品の使用による結果について、なんら保証するものではありません。

本製品は、ベンダのツール(Xilinx 製 EDK、ISE やその他ベンダツール)やベンダの IP コアを利用し、FPGA プロジェクトの構築、コンパイル、コンフィギュレーションデータの生成を行っておりますが、これらツールに関しての販売、サポート、保証等は行っておりません。

目次

図目次

表目次

例目次

1.SUZAKU について

初めに"SUZAKU"がどのようなボードであるのか簡単に説明します。SUZAKU の詳細については本書内いたるところにちりばめられていますので、ここでは概要をつかんでください。

1.1. SUZAKU の特徴

SUZAKU(朱雀)は FPGA をベースとしたボードコンピュータです。

FPGA 上にプロセッサと周辺ペリフェラルコアを構成し、オペレーティングシステムとして Linux を採用しています。

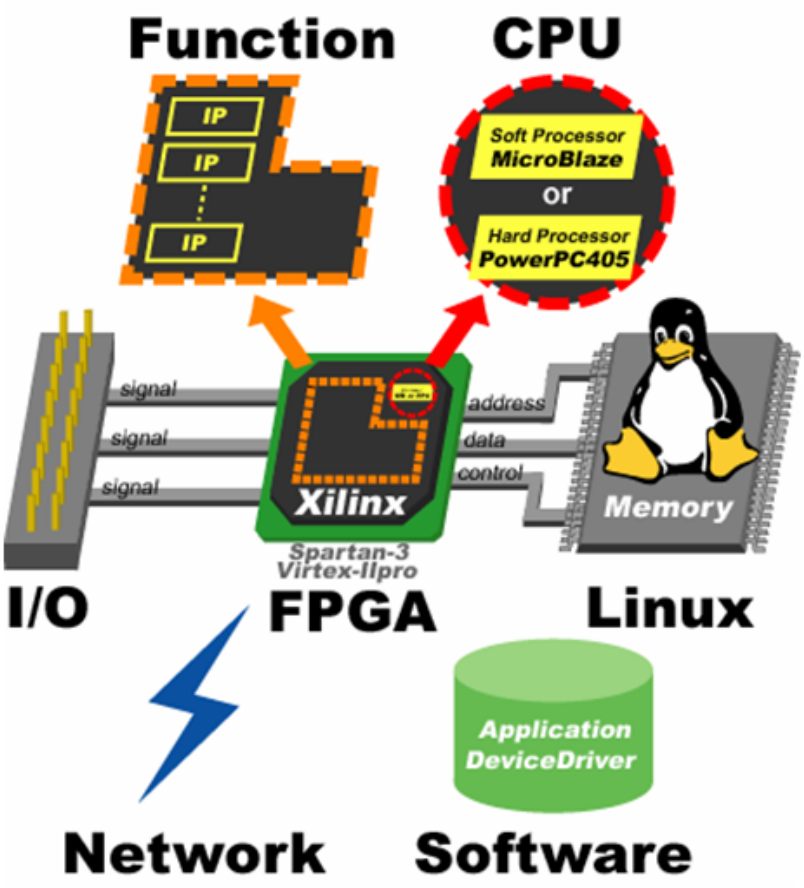


図 1.1. SUZAKU とは

FPGA & Function　　Xilinx の FPGA を採用し、大規模で柔軟な拡張をすることが出来ます。SZ010、SZ030 は Spartan-3、SZ130 は Spartan-3E、SZ310 は Virtex-II Pro、SZ410 は Virtex-4 FX を搭載しています。FPGA の内部は Xilinx やサードパーティ各社から供給される IP(Intellectual Property)を使用することで、必要な機能を容易に追加することが出来ます。また、ユーザによってもカスタマイズが可能です。

CPU	SZ010、SZ030、SZ130 は低コストで資産継承性の高いソフトプロセッサの MicroBlaze を採用し、SZ310、SZ410 は高性能で実績の高いハードプロセッサの PowerPC を採用しています。
Linux & Software	Linux を標準のオペレーティングシステムとして採用しているので、アプリケーションソフトウェアの開発には GNU のアセンブラや C コンパイラ等を使用することができます。SZ010、SZ030、SZ130 は MMU 不要の μClinux、SZ310、SZ410 は標準的な Linux に対応しています。デバイスドライバから各種サーバソフトウェアまで、オープンソースで開発された Linux 対応の豊富なソフトウェア資産を活用することが出来ます。
I/O	基板外周に SZ010、SZ030、SZ130、SZ410 は 86 ピン、SZ310 は 70 ピンのユーザが自由に使える外部 I/O を実装しています。例えば、GPIO や UART の数を増やし、外部 I/O ピンに割り当てるなどのカスタマイズが簡単に行えます。
Network	ボードには LAN(10BASE-T/100BASE-TX)が実装されています。LAN コントローラデバイスドライバ、各種プロトコルが最初から用意されているので、簡単にネットワークに接続できます。

MicroBlaze

MicroBlaze は Xilinx が提供する 32 ビット RISC コアです。MicroBlaze のブロック図を以下に示します。

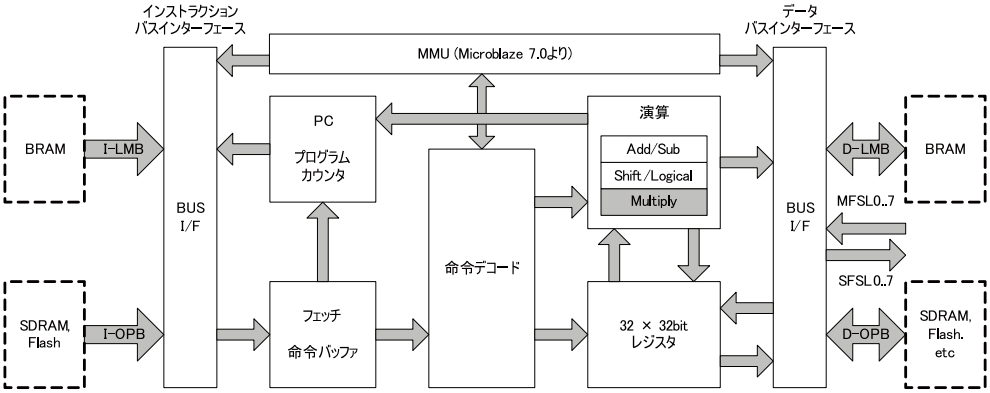


図 1.2. MicroBlaze ブロック図

主な機能

- 32bitRISC プロセッサ
- 32bit 固定長命令
- 32 個の汎用 32bit レジスタ
- MMU なし(MicroBlaze7.0 から MMU あり)
- 命令キャッシュとデータキャッシュ
- ハードウェア乗算器
- ハードウェアデバッグロジック対応
- ペリフェラルバス OPB(MicroBlaze7.0 から PLB)

1.2. 仕様

SUZAKU の主な仕様を以下に示します。

表 1.1. SUZAKU の仕様

	SUZAKU-S			SUZAKU-V	
モデル	SZ010	SZ030	SZ130	SZ310	SZ410
FPGA デバイス	Xilinx Spartan-3 (XC3S400)	Xilinx Spartan-3 (XC3S1000)	Xilinx Spartan-3E (XC3S1200E)	Xilinx Virtex-II Pro (XC2VP4)	Xilinx Virtex-4 FX (XC4VFX12)
CPU コア	MicroBlaze			PowerPC405	
CPU クロック	51.6096MHz			265.4208MHz	350MHz
水晶発振器周波数	3.6864MHz				100MHz
DRAM	16MB		16MB×2	32MB	32MB×2
フラッシュメモリ	4MB	8MB	8MB (SPI)	8MB	8MB (SPI)
Ethernet	10BASE-T/100BASE-TX				
拡張 I/O ピン	86			70	86
シリアル	1ch(UART : 115.2kbps)				
タイマ	2ch(OPB Timer : 1ch は OS で使用)			PowerPC 内蔵タイマ	
コンフィギュレーション	TE7720		SPI フラッシュメモリ	TE7720	SPI フラッシュメモリ
基板サイズ	72×47mm				
電源	電圧：＋ 3.3V±3%				
リセット機能	ソフトウェアリセット				
標準 OS	μ Clinux			Linux	

1.3. 全体ブロック図

SUZAKU の全体ブロック図について説明をします。IP コアについてはここでは説明しません。「10.2.5. IP コア」をご参照ください。

1.3.1. SZ010、SZ030

SZ010 および SZ030 の全体ブロック図を以下に示します。

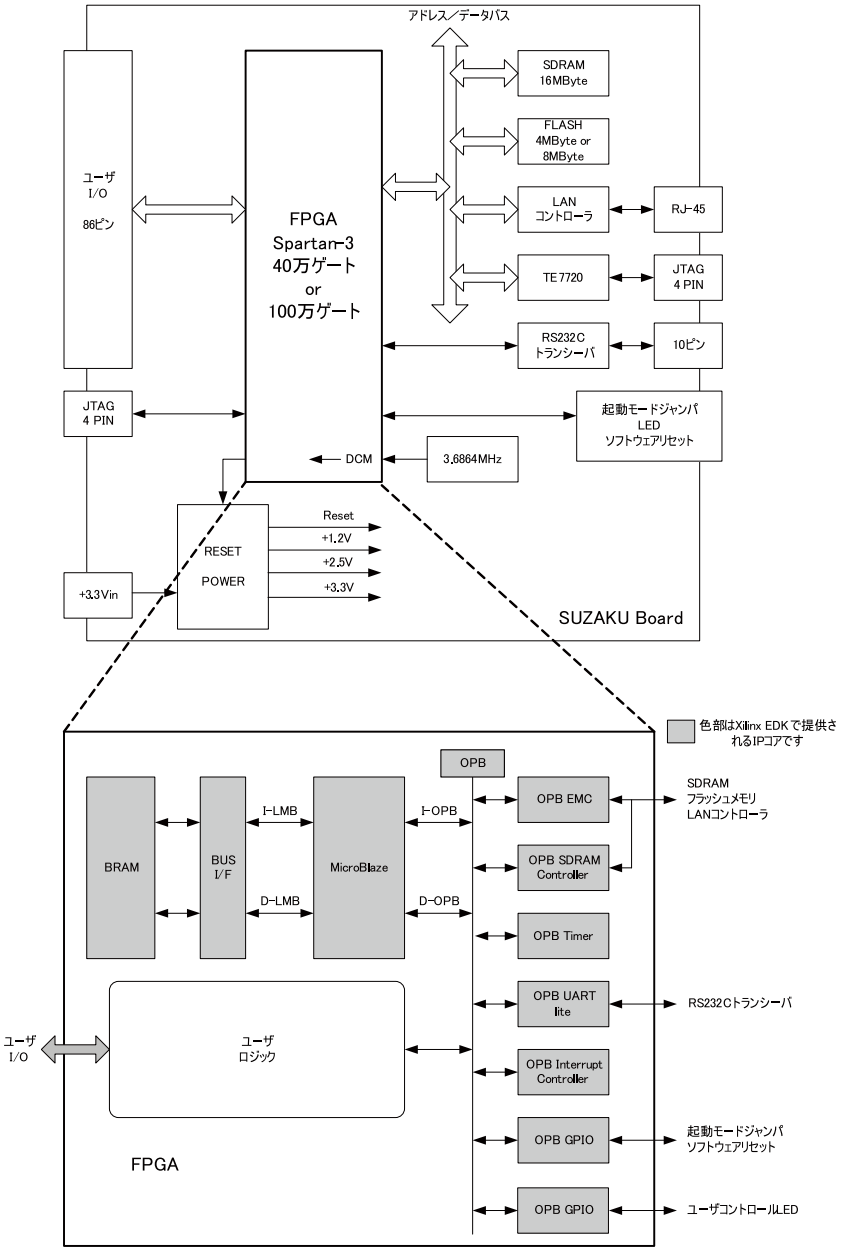


図 1.3. SZ010, SZ030 の全体ブロック図

1.3.1.1. プロセッサ

FPGA 内部で MicroBlaze を使用しています。

1.3.1.2. バス

3 種類のバスで構成しています。

FPGA 内部 LMB(Local Memory Bus)	MicroBlaze と BRAM(FPGA 内部メモリ)を接続する専用バス
FPGA 内部 OPB(On-Chip Peripheral Bus)	複数のペリフェラル IP コアを接続するバス
FPGA 外部バス	OPB-EMC 及び OPB-SDRAM Controller を介し、外部メモリデバイスなどを接続するバス

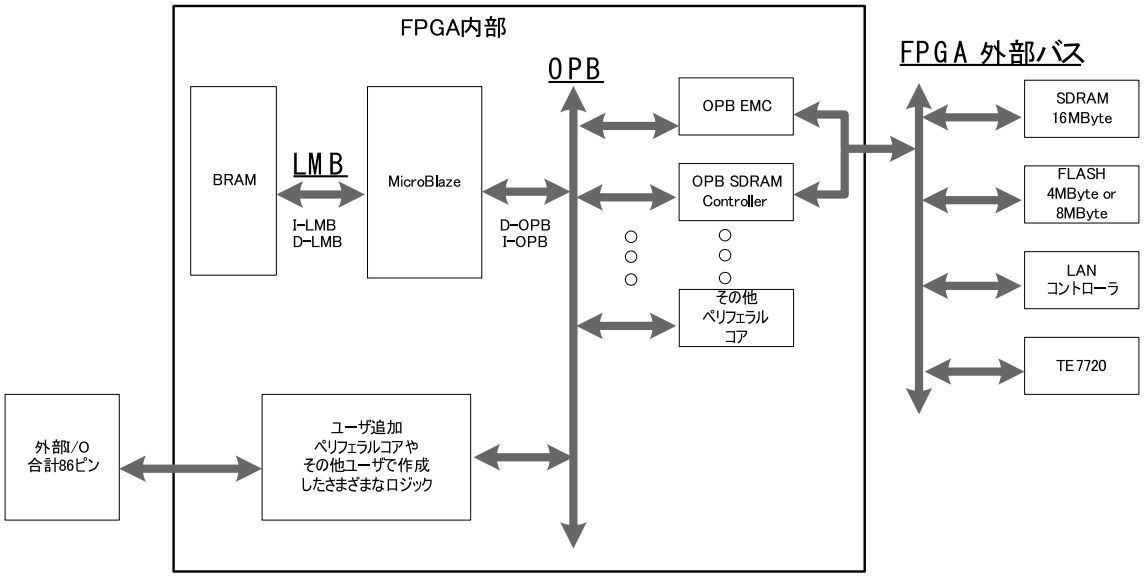


図 1.4. SZ010, SZ030 のバス

1.3.1.3. メモリ

3 種類のメモリで構成しています。

FPGA 内部 BRAM (デフォルト 8KByte)	ブートプログラム用として使用しています。起動完了後は、先頭の 32Byte(割り込みベクタ領域)以外であれば、ユーザプログラムで使用することもできます。
FPGA 外部フラッシュメモリ	SZ010 は 4MByte、SZ030 は 8MByte を実装しています。ブートローダ Hermit や Linux、FPGA コンフィギュレーションデータなどの保存に使用しています。OPB-EMC を使用し、OPB と接続しています。
FPGA 外部 SDRAM 16MByte	Linux のメインメモリとして使用しています。OPB-SDRAM Controller を使用し、OPB と接続しています。

1.3.1.4. シリアルコンソール

OS 用シリアルコンソールに OPB-UART lite を使用しています。OPB-UART lite は RS-232C トランシーバを介し SUZAKU CON1 に接続しています。RS-232C トランシーバは、4 チャンネルタイプのものを実装しています。

1.3.1.5. LAN

LAN コントローラに、LAN91C111(メーカ：SMSC)を実装しています。LAN91C111 は OPB-EMC を使用し OPB と接続しています。

1.3.1.6. FPGA コンフィギュレーション

FPGA コンフィギュレーション IC に TE7720(メーカ：東京エレクトロンデバイス)を実装しています。TE7720 の詳細については「6.2.2. LBPlayer2 で書き換える」をご参照ください。

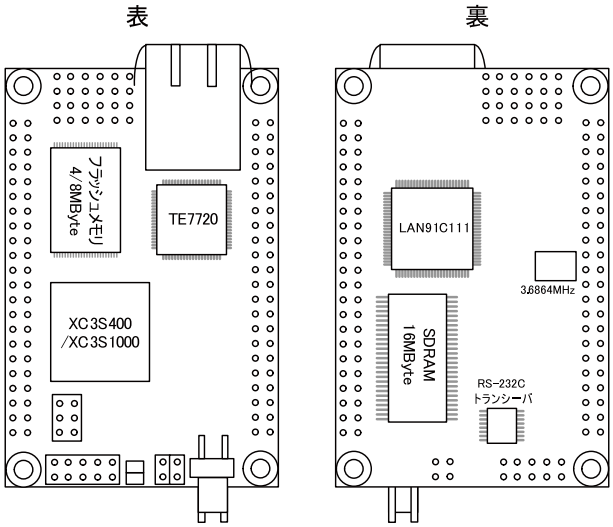


図 1.5. SZ010, SZ030 の主要部品配置図

1.3.2. SZ130

SZ130 の全体のブロック図は以下のとおりです。

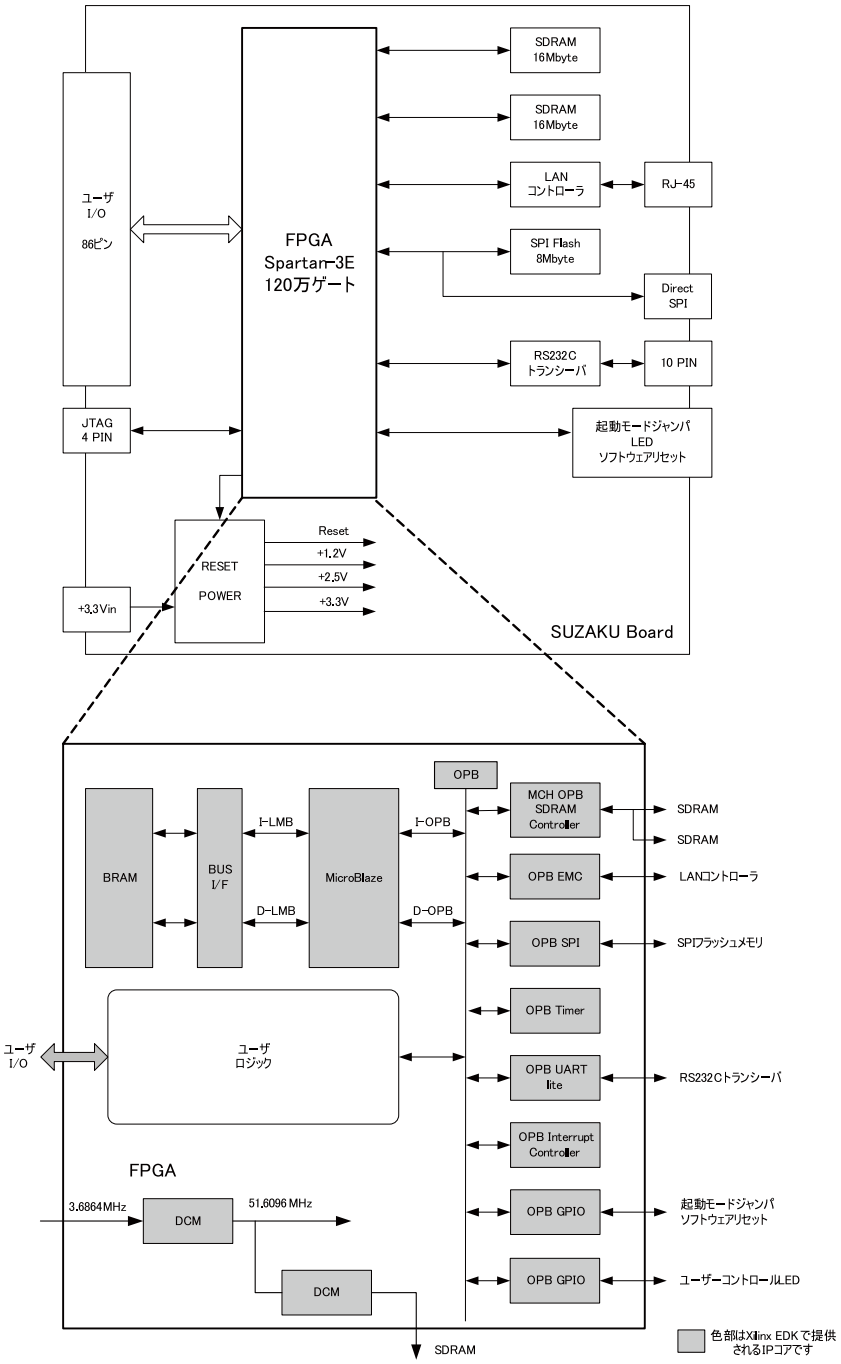


図 1.6. SZ130 の全体ブロック図

1.3.2.1. プロセッサ

FPGA 内部で MicroBlaze を使用しています。

1.3.2.1.1. バス

3 種類のバスで構成しています。

FPGA 内部 LMB(Local Memory Bus)	MicroBlaze と BRAM(FPGA 内部メモリ)を接続する専用バス

FPGA 内部 OPB(On-Chip Peripheral Bus)	複数のペリフェラル IP コアを接続するバス
FPGA 外部バス	OPB-EMC 及び OPB-SDRAM Controller を介し、外部メモリデバイスなどを接続するバス

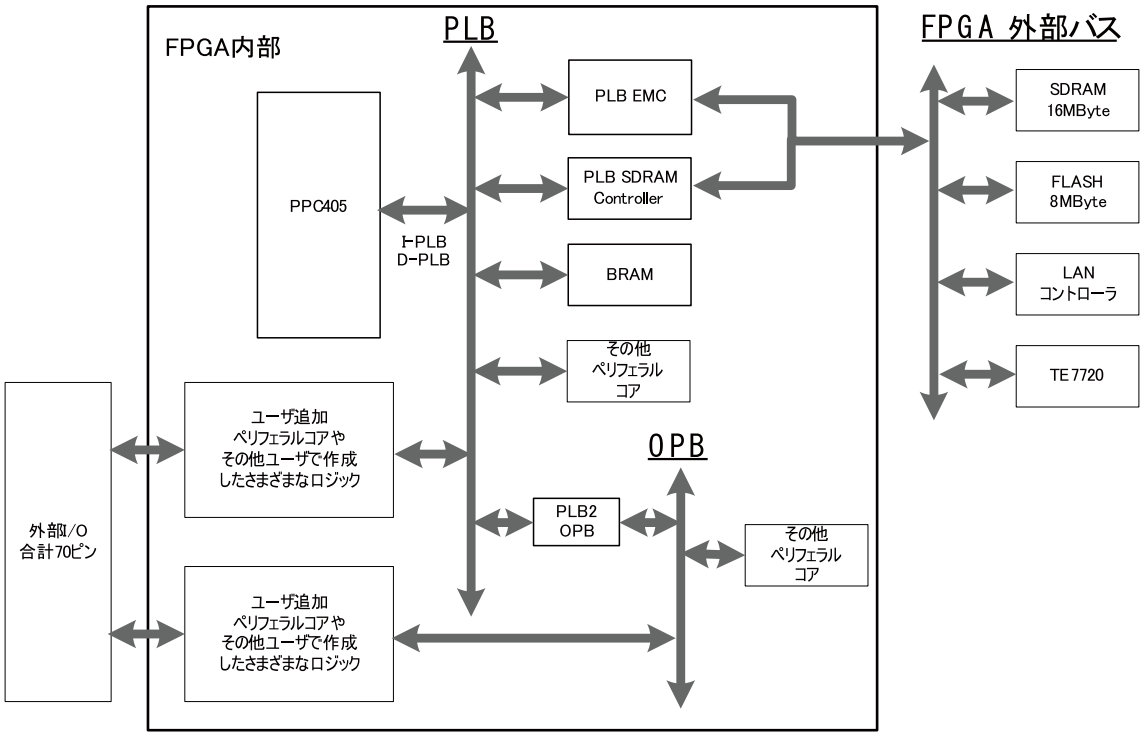


図 1.7. SZ130 のバス

1.3.2.2. メモリ

3 種類のメモリで構成しています。

FPGA 内部 BRAM (デフォルト 8KByte)	ブートプログラム用として使用しています。起動完了後は、先頭の 32Byte(割り込みベクタ領域)以外であれば、ユーザプログラムで使用することもできます。
FPGA 外部 SPI フラッシュメモリ	8MByte を実装しています。ブートローダ Hermit や Linux、FPGA コンフィグデータなどの保存に使用しています。OPB-SPI を使用し、OPB と接続しています。
FPGA 外部 SDRAM 16MByte×2	Linux のメインメモリとして使用しています。OPB-SDRAM Controller を使用し、OPB と接続しています。2 枚の SDRAM の信号線は、完全に 2 つに分離して、FPGA と接続されています。

1.3.2.3. シリアルコンソール

OS 用シリアルコンソールに OPB-UART lite を使用しています。OPB-UART lite は RS-232C トランシーバを介し、SUZAKU CON1 に接続しています。RS-232C トランシーバは、4 チャンネルタイプのものを実装しています。

1.3.2.4. LAN

LAN コントローラは LAN9115(メーカ：SMSC)を実装しています。LAN9115 は OPB-EMC を使用し OPB と接続しています。

1.3.2.5. FPGA コンフィギュレーション

SPI コンフィギュレーションを採用しています。SPI フラッシュメモリは M25P64(メーカ：ST マイクロエレクトロニクス)を実装しています。SPI フラッシュメモリの詳細については「6.2.3. SPI Writer で書き換える」をご参照ください。

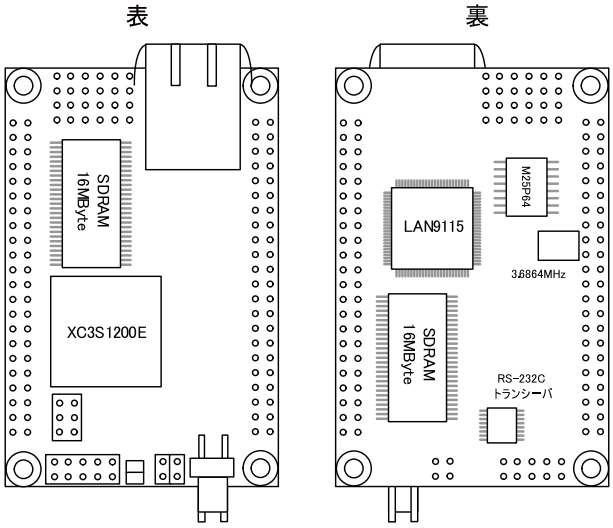


図 1.8. SZ130 の主要部品配置図

1.3.3. SZ310

SZ310 の全体のブロック図を以下に示します。

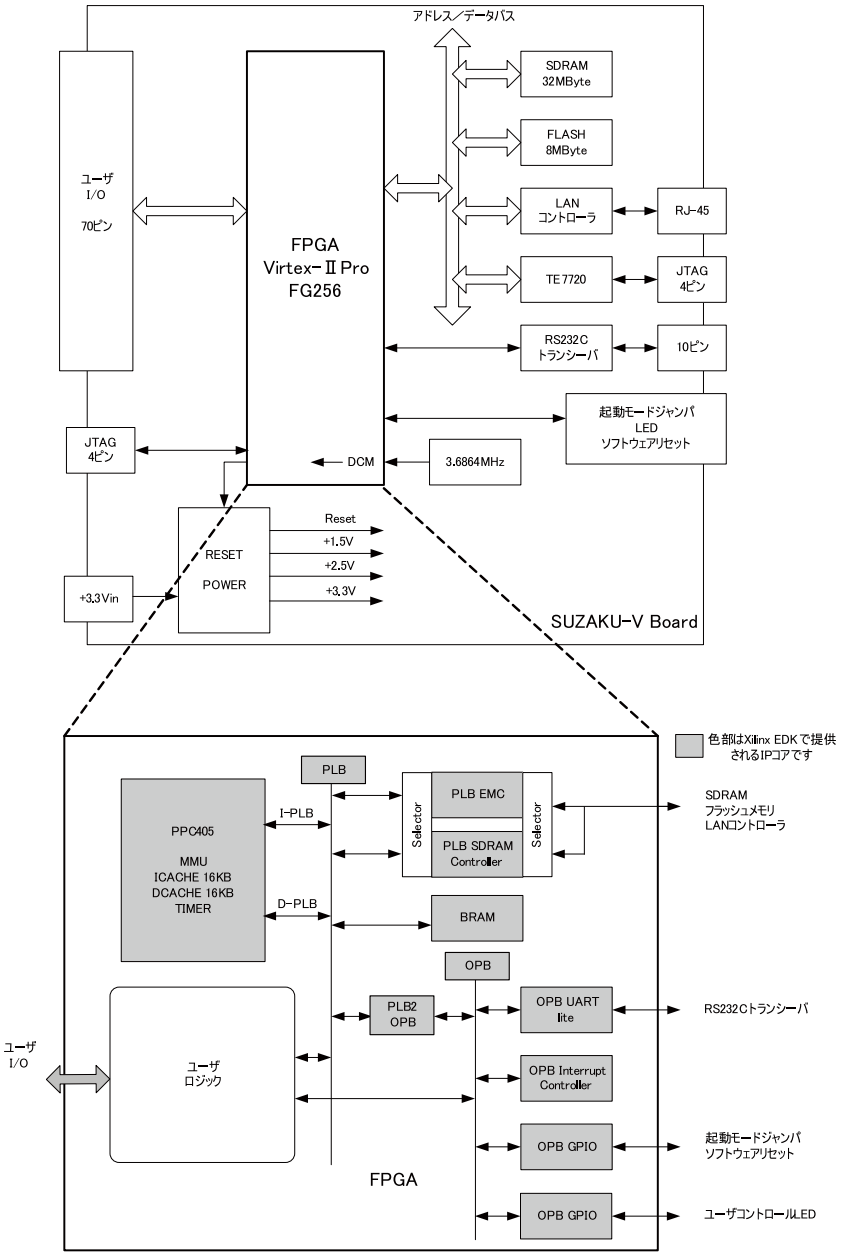


図 1.9. SZ310 の全体ブロック図

1.3.3.1. プロセッサ

FPGA 内部で PowerPC405 を使用しています。

1.3.3.2. バス

3 種類のバスで構成しています。

FPGA 内部 PLB(Processor Local Bus)	PowerPC405 と BRAM、PLB-SDRAM Controller などのペリフェラル IP コアを接続するバス
FPGA 内部 OPB(On-Chip Peripheral Bus)	OPB-UART lite、OPB-INTC などのペリフェラル IP コアを接続するバス

FPGA 外部バス　　PLB-EMC 及び、PLB-SDRAM Controller を介し、外部メモリデバイスなどを接続するバス

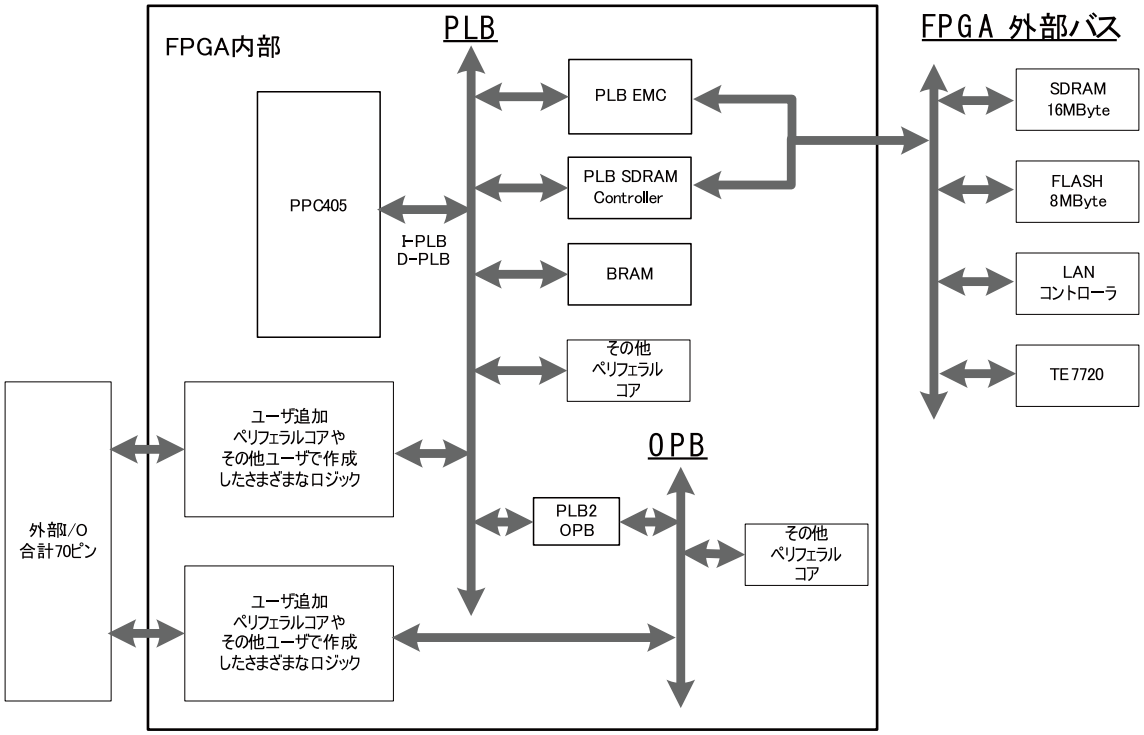


図 1.10. SZ310 のバス

1.3.3.3. メモリ

3 種類のメモリで構成しています。

FPGA 内部 BRAM (デフォルト 16KByte)	ブートプログラム用として使用しています。ブート完了後は、ユーザプログラムで使用することもできます。
FPGA 外部フラッシュメモリ	8MByte を実装しています。ブートローダ Hermit や Linux、FPGA コンフィグデータなどの保存に使用しています。PLB-EMC と接続しています。
FPGA 外部 SDRAM 32MByte	Linux のメインメモリとして使用しています。PLB-SDRAM を使用し、PLB と接続しています。

1.3.3.4. シリアルコンソール

OS 用シリアルコンソールに OPB-UART lite を使用しています。OPB-UART lite は RS-232C トランシーバを介し、SUZAKU CON1 に接続しています。RS-232C トランシーバは、4 チャンネルタイプのものを使用しています。

1.3.3.5. LAN

LAN コントローラに、FPGA 外部に LAN91C111(メーカ：SMSC)を実装しています。

LAN91C111 は、PLB-EMC を使用し、PLB と接続しています。

1.3.3.6. FPGA コンフィギュレーション

FPGA コンフィギュレーション IC に TE7720(メーカ：東京エレクトロンデバイス)を実装しています。TE7720 の詳細 については「6.2.2. LBPlayer2 で書き換える」をご参照ください。

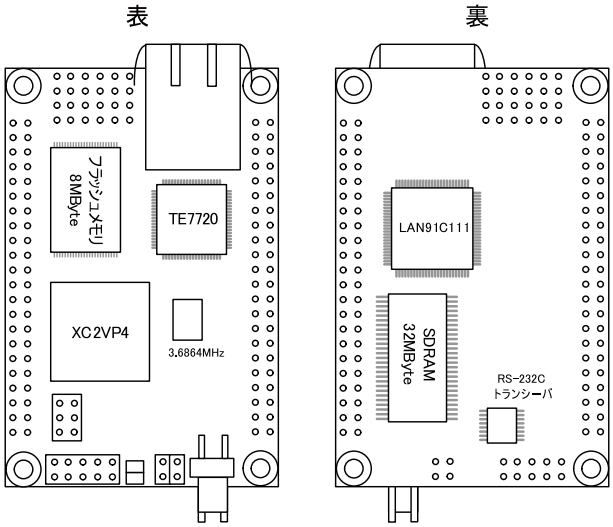


図 1.11. SZ310 の主要部品配置図

1.3.4. SZ410

SZ410 の全体のブロック図を以下に示します。

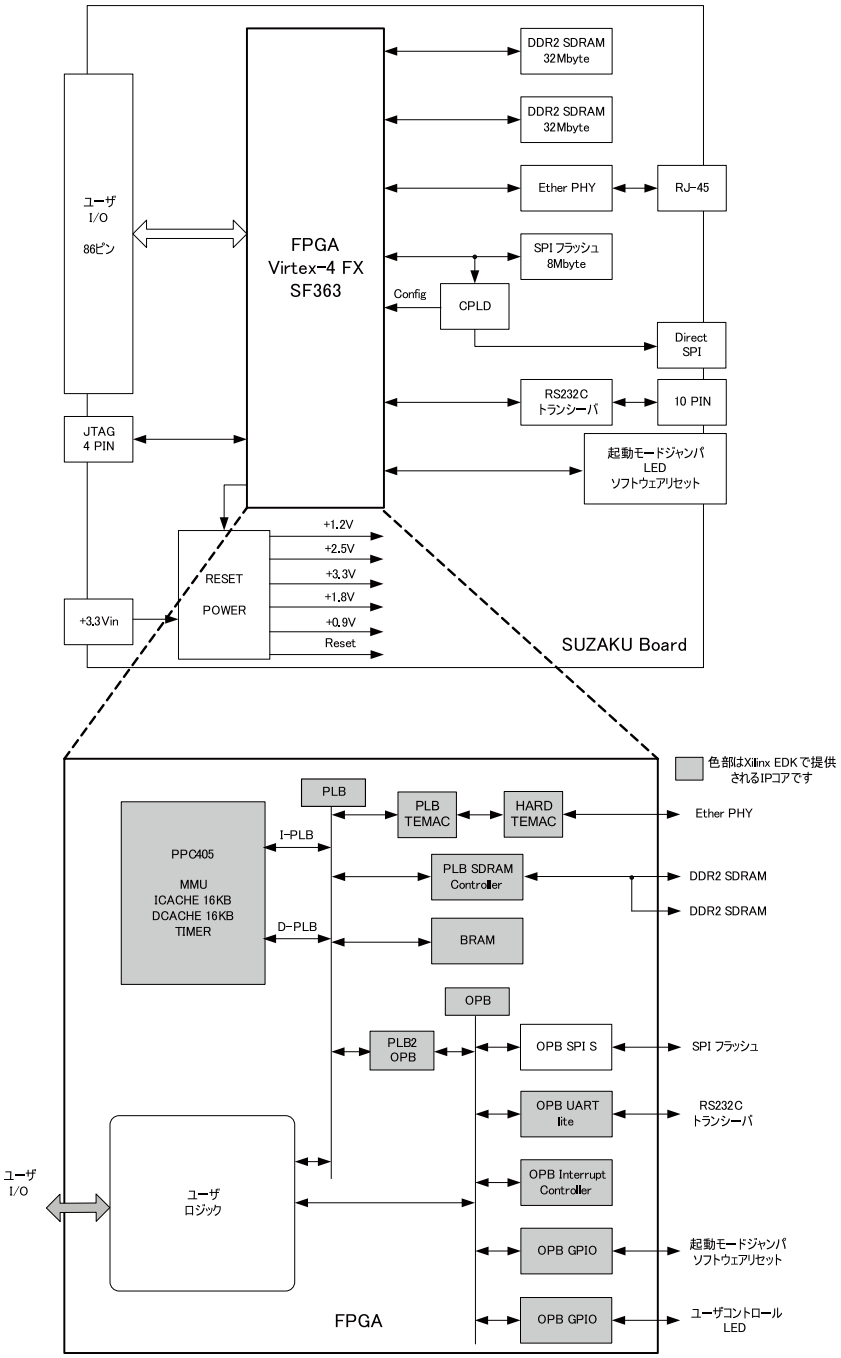


図 1.12. SZ410 の全体ブロック図(2008/1/18 以降)

1.3.4.1. プロセッサ

FPGA 内部で PowerPC405 を使用しています。

1.3.4.2. バス

4 種類のバスで構成しています。

FPGA 内部 PLB(Processor Local Bus)	PowerPC405 と BRAM、MPMC などのペリフェラル IP コアを接続するバス

OCM(On-Chip Memory Bus)	PowerPC405 と TEMAC の FIFO を接続するバス
DCR(Device Controll Register Bus)	PowerPC405 と TEMAC を接続するバス
FPGA 外部バス	PLB-DDR2 Controller 等を介し、外部メモリデバイスなどを接続するバス

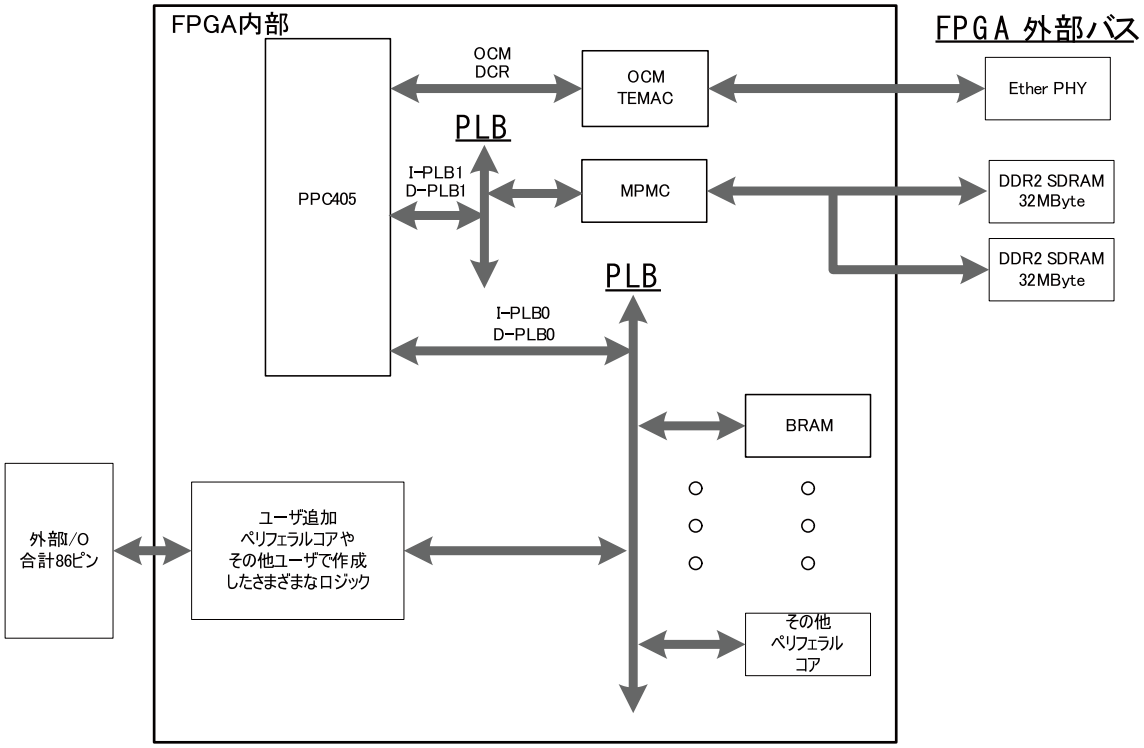


図 1.13. SZ410 のバス(2008/1/18 以降)

1.3.4.3. メモリ

3 種類のメモリで構成しています。

FPGA 内部 BRAM (デフォルト 16KByte)	ブートプログラム用として使用しています。ブート完了後は、ユーザプログラムで使用することもできます。
FPGA 外部 SPI フラッシュメモリ	8MByte を実装しています。ブートローダ Hermit や Linux、FPGA コンフィグデータなどの保存に使用しています。OPB-SPI を使用し、OPB と接続しています。
FPGA 外部 DDR2 SDRAM 32MByte×2	Linux のメインメモリとして使用しています。PLB-DDR2 Controller と接続しています。2 枚の DDR2 SDRAM の信号線は完全に 2 つに分離して、FPGA と接続されています。

1.3.4.4. シリアルコンソール

OS 用シリアルコンソールに OPB-UART lite を使用しています。OPB-UART lite は RS-232C トランシーバを介し、SUZAKU CON1 に接続しています。RS-232C トランシーバは、4 チャンネルタイプのものを使用しています。

1.3.4.5. LAN

Virtex-4 FX 内蔵の TEMAC(Tri-Mode Ether MAC)と 10BASE-T/100BASE-TX の Ethrer PHY(メーカ：SMSC)を使用しています

1.3.4.6. FPGA コンフィギュレーション

CPLD を使用した SPI コンフィギュレーションを採用しています。SPI フラッシュメモリは M25P64(メーカ：ST マイク ロエレクトロニクス)を実装しています。SPI フラッシュメモリの詳細については「6.2.3. SPI Writer で書き換える」をご参照ください。

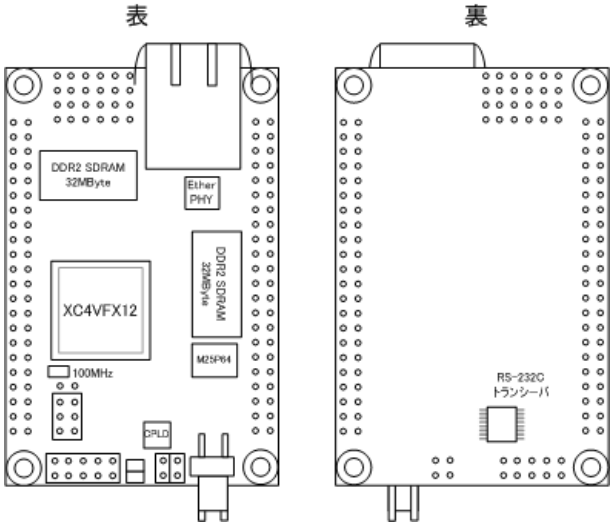
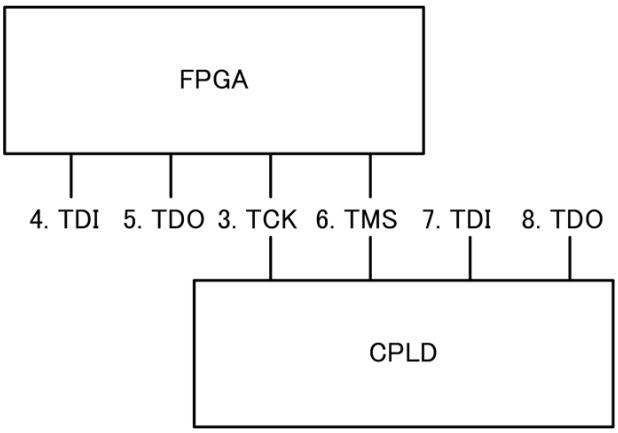
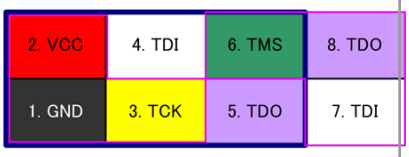


図 1.14. SZ410 の主要部品配置図

SZ410 の JTAG(CON7)コネクタの横のスルーホールは？

SZ410 の JTAG(CON7)コネクタの横のスルーホールは CPLD 書き込み用のスルーホールです。FPGA と CPLD の JTAG ピンは以下の図のように TCK、TMS を共用する形で配線されています。CPLD を書き換えることはもちろん出来ますが、書き換えに失敗してしまうと SUZAKU が動くようになるまでの復旧が大変なので、どうしてもということがないかぎりは、CPLD を書き換えないようにしてください。





SZ410 CPLD と SPI フラッシュの採用理由

CPLD と SPI フラッシュメモリを採用したのは、シリアルビットストリームである SPI はそのままマスターシリアルに置き換えることが容易であり、また 8mm×6mm の SPI フラッシュメモリと 5mm×5mm の CPLD (XC2C32QFP32 ピン)を使用すると、基板上の設置面積が小さくてすむからです。

1.4. メモリマップ

1.4.1. SZ010、SZ030

SZ010 および SZ030 のメモリマップは以下のとおりです。

表 1.2. SZ010、SZ030 のメモリマップ

Start Address	End Address	ペリフェラル	デバイス
0x0000 0000	0x0000 1FFF	BRAM	BlockRAM 8KByte
0x0000 2000	0x7FFF FFFF	Reserved	
0x8000 0000	0x80FF FFFF	OPB-SDRAM Controller	SDRAM 16MByte
0x8100 0000	0xFEFF FFFF	Free	
0xFF00 0000	0xFF7F FFFF	OPB-EMC	Flash 4MByte or 8MByte
0xFF80 0000	0xFFDF FFFF	Free	
0xFFE0 0000	0xFFEF FFFF	OPB-EMC	LAN コントローラ
0xFFF0 0000	0xFFFF 0FFF	Free	
0xFFFF 1000	0xFFFF 10FF	OPB-Timer	
0xFFFF 1100	0xFFFF 1FFF	Free	
0xFFFF 2000	0xFFFF 20FF	OPB-UART lite	RS-232C
0xFFFF 2100	0xFFFF 2FFF	Free	
0xFFFF 3000	0xFFFF 30FF	OPB-Interrupt Controller	
0xFFFF 3100	0xFFFF 9FFF	Free	
0xFFFF A000	0xFFFF A1FF	OPB-GPIO	ブートモードジャンパ ソフトウェアリセット
0xFFFF A200	0xFFFF A3FF	OPB-GPIO	LED
0xFFFF A400	0xFFFF FFFF	Free	

1.4.2. SZ130

SZ130 のメモリマップは以下のとおりです。

表 1.3. SZ130 のメモリマップ

Start Address	End Address	ペリフェラル	デバイス
0x0000 0000	0x0000 1FFF	BRAM	BlockRAM 8KByte
0x0000 2000	0x7FFF FFFF	Reserved	
0x8000 0000	0x81FF FFFF	OPB-SDRAM Controller	SDRAM 32MByte
0x8200 0000	0xFEFF FFFF	Free	
0xFF00 0000	0xFF00 01FF	OPB-SPI	SPI Flash 8MByte
0xFF00 0200	0xFFDF FFFF	Free	
0xFFE0 0000	0xFFE0 FFFF	OPB-EMC	LAN コントローラ
0xFFE1 0000	0xFFFF 0FFF	Free	
0xFFFF 1000	0xFFFF 10FF	OPB-Timer	
0xFFFF 1100	0xFFFF 1FFF	Free	
0xFFFF 2000	0xFFFF 20FF	OPB-UART lite	RS-232C
0xFFFF 2100	0xFFFF 2FFF	Free	
0xFFFF 3000	0xFFFF 30FF	OPB-Interrupt Controller	
0xFFFF 3100	0xFFFF 9FFF	Free	
0xFFFF A000	0xFFFF A1FF	OPB-GPIO	ブートモードジャンパ ソフトウェアリセット
0xFFFF A200	0xFFFF A3FF	OPB-GPIO	LED
0xFFFF A400	0xFFFF FFFF	Free	

1.4.3. SZ310

SZ310 のメモリマップは以下のとおりです。

表 1.4. SZ310 のメモリマップ

Start Address	End Address	ペリフェラル	デバイス
0x0000 0000	0x01FF FFFF	PLB-SDRAM Controller	SDRAM 32MByte
0x0200 0000	0xEFFF FFFF	Free	
0xF000 0000	0xF07F FFFF	PLB-EMC	Flash 8MByte
0xF080 0000	0xF0DF FFFF	Free	
0xF0E0 0000	0xF0EF FFFF	PLB-EMC	LAN コントローラ
0xF0F0 0000	0xF0FF 1FFF	Free	
0xF0FF 2000	0xF0FF 20FF	OPB-UART lite	RS-232C
0xF0FF 2100	0xF0FF 2FFF	Free	
0xF0FF 3000	0xF0FF 30FF	OPB-Interrupt Controller	
0xF0FF 3100	0xF0FF 9FFF	Free	
0xF0FF A000	0xF0FF A1FF	OPB-GPIO	ブートモードジャンパ ソフトウェアリセット
0xF0FF A200	0xF0FF A3FF	OPB-GPIO	LED
0xF0FF A400	0xFFFF BFFF	Free	

Start Address	End Address	ペリフェラル	デバイス
0xFFFF C000	0xFFFF FFFF	BRAM	BlockRAM 16KByte

1.4.4. SZ410

SZ410 のメモリマップは以下のとおりです。(2008/1/18 以降)

表 1.5. SZ410 のメモリマップ

Start Address	End Address	ペリフェラル	デバイス
0x0000 0000	0x03FF FFFF	MPMC	DDR2 SDRAM 64MByte
0x0400 0000	0xF0EF FFFF	Free	
0xF0FF 0000	0xF0FF 01FF	XPS-SPI	SPI Flash 8MByte
0xF0FF 0200	0xF0FF 1FFF	Free	
0xF0FF 2000	0xF0FF 20FF	XPS-UART lite	RS-232C
0xF0FF 2100	0xF0FF 2FFF	Free	
0xF0FF 3000	0xF0FF 30FF	XPS-Interrupt Controller	
0xF0FF 3100	0xF0FF 9FFF	Free	
0xF0FF A000	0xF0FF A1FF	XPS-GPIO	ブートモードジャンパ ソフトウェアリセット
0xF0FF A200	0xF0FF A3FF	XPS-GPIO	LED
0xF0FF A400	0xFFFF BFFF	Free	
0xFFFF C000	0xFFFF FFFF	BRAM	BlockRAM 16KByte

表 1.6. SZ410 の DCR メモリマップ

Start Address	End Address	ペリフェラル	デバイス
0x000	0x007	OCM-TEMAC	Ethernet PHY

2.LED/SW ボードについて

SUZAKU スターターキットは"SUZAKU+LED/SW ボード"で構成されます。LED/SW ボードは SUZAKU の学習用ボードとして生み出されました。LED/SW ボードについて回路図をみながら簡単に説明します。詳細については「7. ISE の使い方」で実際に LED/SW ボードに触りながら説明します。

2.1. 回路説明

以下の回路図が LED/SW ボードの回路図です。回路図及び部品表は付属 CD-ROM の

`"\suzaku-starter-kit\doc"`に収録されているので詳細はそちらをご参照ください。

LED/SW ボードには単色 LED が 4 つ(D1、D2、D3、D4)、押しボタンスイッチが 3 つ(SW1、SW2、SW3)、ロータリコードスイッチが 1 つ(SW4)、7 セグメント LED が 3 つ(LED1、LED2、LED3)、シリアルポートが 1 つ実装されており、それぞれ CON2 から SUZAKU と接続するようになっています。安定した＋ 3.3V を得るため AC アダプタ 5V から 3 端子レギュレータで＋ 3.3V を作っています。この＋ 3.3V は CON2、CON3 から SUZAKU 側へ供給されます。

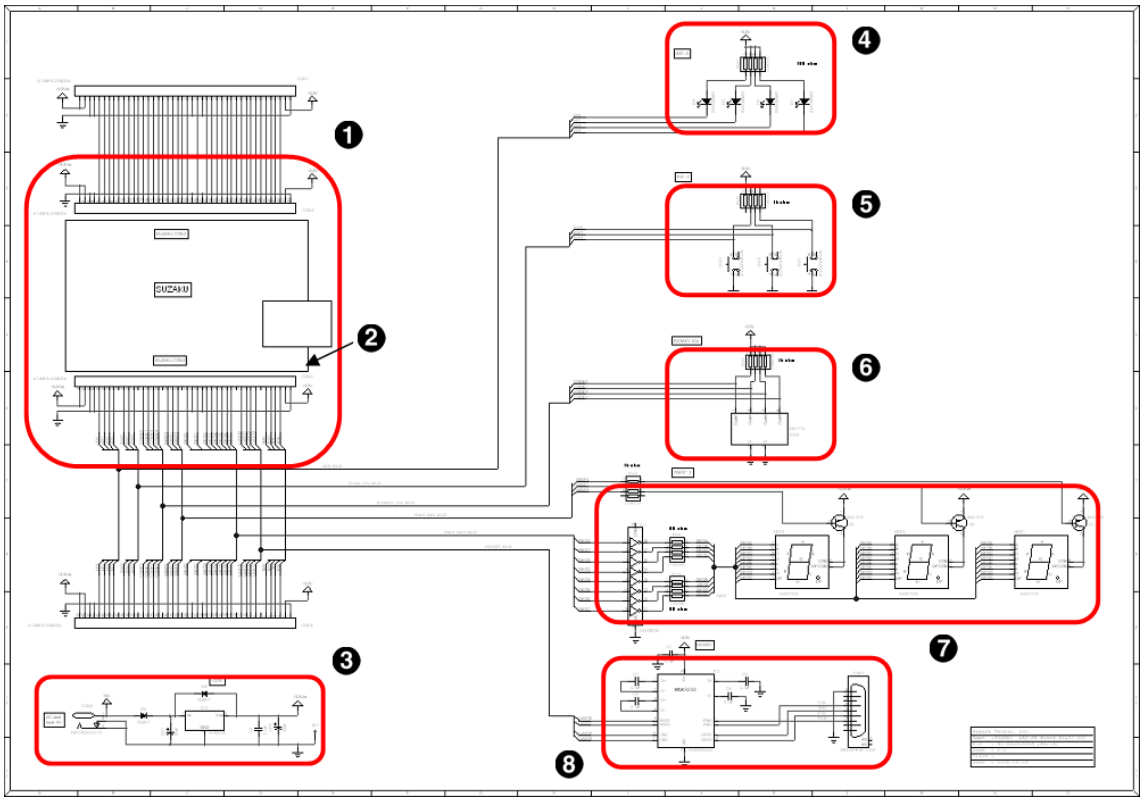


図 2.1. LED/SW 回路図(縮小版)

❶ SUZAKU

❷ CON2 のピンアサインについては「2.2. ピンアサイン」をご参照ください。

❸ 3 端子レギュレータ

❹ LED×4

❺ 押しボタンスイッチ×3

❻ ロータリコードスイッチ

❼ 7 セグメント LED

❽ シリアルポート

2.2. ピンアサイン

LED/SW ボードを使用する際に必要となるピンアサインを以下に示します。

その他の SUZAKU ＋ LED/SW ボードのピンアサインについては「13. SUZAKU ＋ LED/SW ボードのピンアサイン」をご参照ください。

表 2.1. クロック、リセット信号 ピンアサイン

番号	信号名	I/O	機能	FPGA 接続先			
				SZ010 SZ030	SZ130	SZ310	SZ410
	SYS_CLK	I	クロック信号	T9	U10	C8	Y6
	SYS_RST	I	リセット信号	F5	D3	A8	U3

表 2.2. 機能用ピンアサイン(CON2)

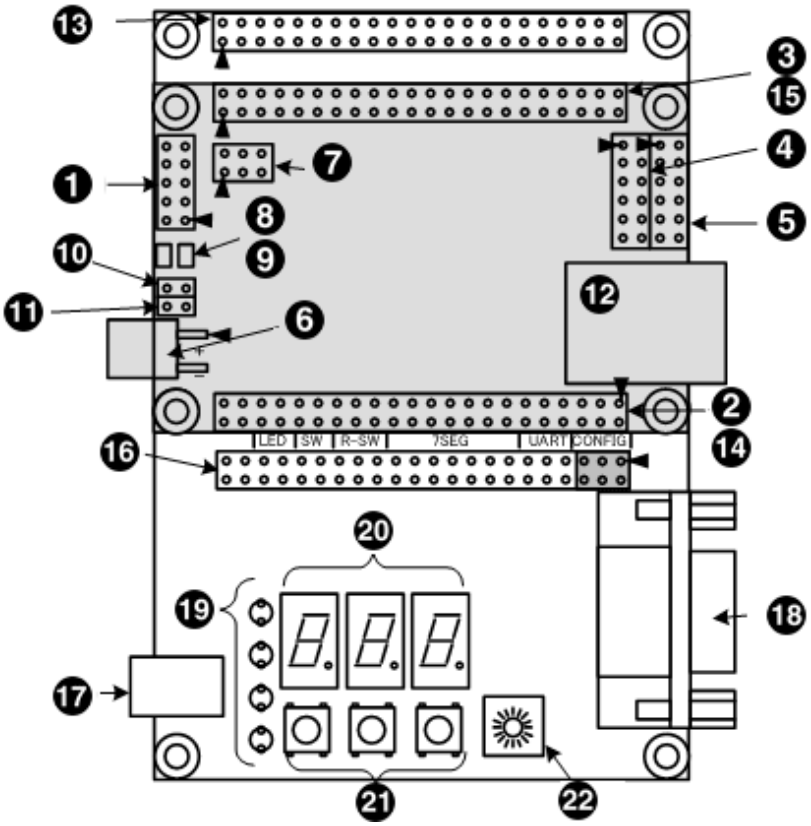
番号	信号名	I/O	機能	FPGA 接続先			
				SZ010 SZ030	SZ130	SZ310	SZ410
8	UART3	I	RTS	A7	N4	E14	D15
9	UART2	O	TXD	A3	M6	E13	E15
10	UART1	O	CTS	D5	M5	F12	F15
11	UART0	I	RXD	B4	M3	F13	P4
13	SEG7	O	セグメント DP	C5	L5	F15	P1
14	SEG6	O	セグメント G	B5	L6	F16	P2
15	SEG5	O	セグメント F	E6	L4	G13	L2
16	SEG4	O	セグメント E	D6	L3	G14	M2
17	SEG3	O	セグメント D	C6	L2	G15	N2
18	SEG2	O	セグメント C	B6	L1	G16	N3
20	SEG1	O	セグメント B	A8	C9	N9	Y7
22	SEG0	O	セグメント A	B8	D9	P9	W7
24	nSEL2	O	7 セグメント LED3 選択	D7	K6	H13	N5
25	nSEL1	O	7 セグメント LED2 選択	C7	K4	H14	M3
26	nSEL0	O	7 セグメント LED1 選択	B7	K3	H15	M4
28	nCODE3	I	ロータリコード スイッチ 23	C8	J1	J16	H5
29	nCODE2	I	ロータリコード スイッチ 22	A9	F9	J15	E2
30	nCODE1	I	ロータリコード スイッチ 21	A12	E9	J14	D2
31	nCODE0	I	ロータリコード スイッチ 20	C10	A10	J13	U9
33	nSW2	I	押しボタン スイッチ SW3	A14	D11	K16	L1
34	nSW1	I	押しボタン スイッチ SW2	B14	C11	K15	M1
35	nSW0	I	押しボタン スイッチ SW1	A13	F11	K14	G4
37	nLE0	O	単色 LED(緑)D1	B12	E12	L16	G2
38	nLE1	O	単色 LED(緑)D2	C12	F12	L15	F2
39	nLE2	O	単色 LED(緑)D3	D11	B11	L14	F1
40	nLE3	O	単色 LED(緑)D4	E11	A11	L13	E1

3.SUZAKU+LED/SW ボードの構成

SUZAKU+LED/SW ボードのコネクタの配置やジャンパの設定等について説明します。これらは実際に作業するために必要不可欠な情報です。誤挿入や誤配線等間違った使い方をすると、SUZAKU や LED/SW ボードが壊れてしまう可能性があります。しっかりとご確認ください。

3.1. 各種インターフェースの配置

SUZAKU には FPGA やメモリ、Ethernet コントローラ等が実装され、Linux が動作します。LED/SW ボードには LED やスイッチ、RS-232C トランシーバが実装されています。電源は LED/SW ボード側から供給します(SUZAKU 側の電源コネクタは使用しません)。各種インターフェースの配置は下図のようになっています。



♟のマークは基板上の 1 番ピンの位置を示します

図 3.1. 各種インターフェースの配置

表 3.1. SUZAKU のコネクタ配置

	図番	部品番号	説明
SUZAKU	1	CON1	RS-232C コネクタ
	2	CON2	外部 I/O、フラッシュメモリ用コネクタ (LED/SW CON2 と接続)
	3	CON3	外部 I/O コネクタ(LED/SW CON3 と接続)
	4	CON4	外部 I/O コネクタ
	5	CON5	外部 I/O コネクタ
	6	CON6	電源入力＋ 3.3V (LED/SW 接続時は絶対に使用しないでください)
	7	CON7	FPGA JTAG 用コネクタ
	8	D1	ユーザーコントロール LED(赤)
	9	D3	パワー ON LED(緑)
	10	JP1	起動モードジャンパ
	11	JP2	FPGA プログラム用ジャンパ
	12	L2	Ethernet 10BASE-T/100BASE-TX コネクタ

表 3.2. LED/SW のコネクタ配置

	図番	部品番号	説明
LED/SW	13	CON1	テスト拡張用コネクタ (CON3 と同じピンアサインで配線接続されています)
	14	CON2	SUZAKU 接続コネクタ(SUZAKU CON2 と接続)
	15	CON3	SUZAKU 接続コネクタ(SUZAKU CON3 と接続)
	16	CON4	テスト拡張用コネクタ (CON2 と同じピンアサインで配線接続されています)
	17	CON6	＋ 5V 入力コネクタ
	18	CON7	RS-232C コネクタ
	19	D1 - 4	単色 LED(緑) “Low”レベルで点灯
	20	LED1 - 3	7 セグメント LED　“High”レベルで点灯
	21	SW1 - 3	押しボタンスイッチ　押下で”Low”レベル
	22	SW4	ロータリコードスイッチ　選択時”Low”レベル

4.電源を入れる前に

SUZAKU スターターキットに電源を入れる前に、開発をするために必要なものやインストールする必要のあるソフトウェア、開発環境についての説明をします。

4.1. 必要なもの

SUZAKU スターターキットの場合は、以下のものが収められています。ご確認ください。

それ以外の場合は以下のものが必要となります。足りないものをそろえて下さい。

①SUZAKU[1]
②LED/SW ボード [2]
③CD-ROM
④AC アダプタ 5V
⑤D-sub9 ピン-10 ピン変換ケーブル
⑥D-sub9 ピンクロスケーブル
⑦スペーサ×4
⑧ネジ×4
⑨ジャンパプラグ×2

[1] 本書対応の SUZAKU は SUZAKU-S(SZ010、SZ030、SZ130)、SUZAKU-V(SZ310、SZ410)です。
[2]SZ410 をお使いの場合は、SZ410 対応品の LED/SW ボード(SIL00-U01)をお使いください。

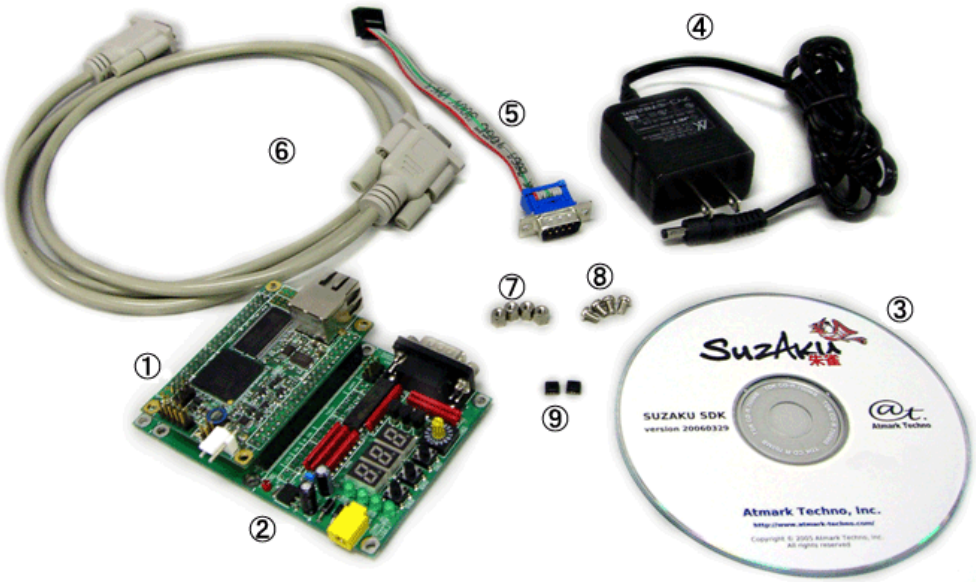


図 4.1. SUZAKU スターターキット(SZ130)

4.2. 開発環境

SUZAKU スターターキットの開発環境として必要なソフトウェアおよびハードウェアは以下のとおりです。

作業用 PC	Windows2000 または、WindowsXP が動作し、シリアルポート(1 ポート)、及びパラレルポート(1 ポート)を持つ PC を準備してください。
Xilinx Parallel CableⅢ、Ⅳ またはそれ相当品 [1]	Parallel CableⅢ、Ⅳ またはそれ相当のものを準備してください。Parallel CableⅣ 使用の場合は Parallel Cable Fly Leads も別途必要となります。
シリアル通信用ソフト	Tera Term(Pro)等のシリアル通信用ソフトをインストールしてください。Tera Term(Pro)はフリーソフトウェアのターミナルエミュレータで、シリアル通信等を行うことができます。Tera Term(Pro)には UTF-8 に対応しているバージョンもあります。
Xilinx ISE[1]	Xilinx ISE10.1i 以降(Foundation(有償版)、WebPACK(無償版)どちらでも可)を準備し、インストールしてください。無償版の ISE WebPACK は、Xilinx のホームページ [http://www.xilinx.co.jp/]からダウンロードできます。インストール後ソフトウェアアップデートをしてください。
Xilinx EDK[1]	Xilinx EDK10.1i 以降(Embedded Development Kit)を準備し、インストールしてください。60 日間評価版の EDK は、Xilinx のホームページ [http://www.xilinx.co.jp/]からダウンロードできます。インストール後ソフトウェアアップデートをしてください。
LBPlayer2 SZ010 SZ030 SZ310	LBPlayer2 をインストールしてください。SZ010、SZ030、SZ310 のフラッシュメモリに書き込む際に使用します。 付属 CD-ROM の`"\suzku\tools\LBPlay2_Release108.zip\Lbplayer2.lzh"`に収録されています。また、東京エレクトロンデバイスのホームページ [http://www.teldevice.co.jp/]から最新版をダウンロードすることが出来ます。インストール方法については解凍したフォルダ内の `readme.txt` 等をご参照ください。
SPI Writer SZ130 SZ410	SPI Writer をインストールしてください。SZ130、SZ410 のフラッシュメモリに書き込む際に使用します。付属 CD-ROM の`"\suzaku\tools\spi_writer-yyyymmdd.zip"`に収録されています。また、SUZAKU 公式サイトのダウンロードページ [http://suzaku.atmark-techno.com/downloads/all]から最新版をダウンロードすることが出来ます。インストール方法については解凍したフォルダ内の `spi_writer_manual_ja-x.x.x.pdf` をご参照ください。
ダウンローダ Hermit	ダウンローダ Hermit をインストールしてください。付属 CD-ROM の`"\suzaku\bootloader\hermit-at-win-x.x.x.zip"`に収録されています。また、SUZAKU 公式サイトのダウンロードページ [http://suzaku.atmark-techno.com/downloads/all]から最新版をダウンロードすることが出来ます。

[1] Xilinx 製品の詳細については、Xilinx のホームページ [http://www.xilinx.co.jp/]をご覧になられるか、Xilinx 代理店にお問い合わせください。

4.3. 付属 CD-ROM について

付属 CD-ROM には、開発に必要となる FPGA プロジェクト、ソフトウェア、開発環境、イメージファイル、各種マニュアルが収められています。これらは不具合解決や機能増強等のアップグレードを行うことがあります。下記サイトに最新版がございますのでダウンロードしてお使いください。

開発に関するファイル	http://suzaku.atmark-techno.com/downloads/all
各種マニュアル	http://suzaku.atmark-techno.com/downloads/docs

4.4. 組み立て

LED/SW ボードに SUZAKU を接続します(SUZAKU スターターキットの場合は出荷時に接続されています)。接続する際、方向、位置に十分ご注意ください。間違った方向、位置に接続して電源を投入した場合、電源がショートして壊れる可能性があります。SUZAKU スターターキットでは CON2 の 19 番ピンに逆挿し防止対策が施されています。

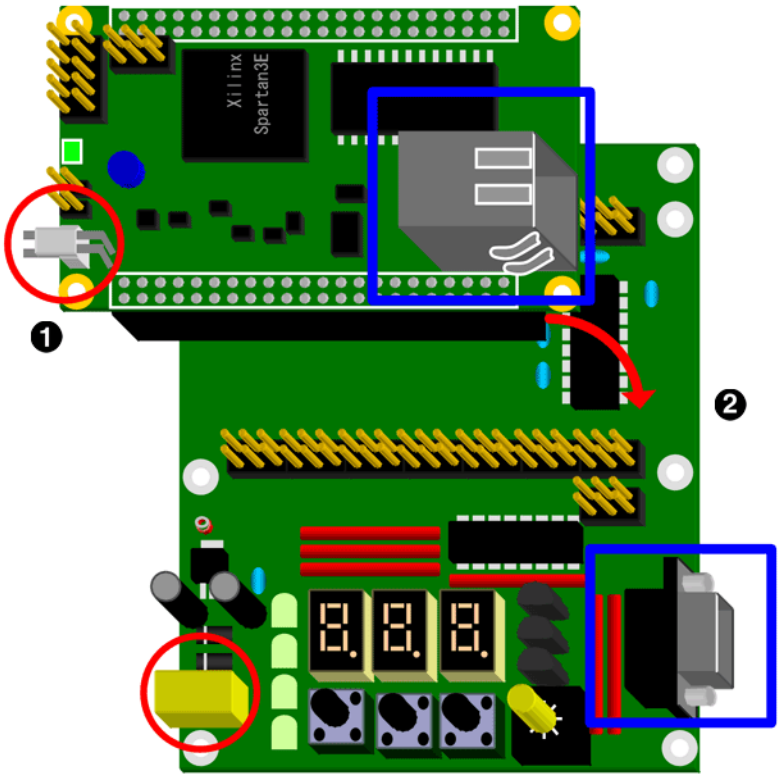


図 4.2. SUZAKU と LED/SW ボード接続

❶ 逆差し注意！

❷ コネクタをきっちりあわせる

もし、CON2、CON3 にコネクタが接続されていない場合、取り付け面と位置に注意し、コネクタを半田付けしてください。コネクタは 40〜44 ピンのものをご用意ください。CON2 の 41〜44 ピン、CON3 の 41〜44 ピンにはコネクタを接続しなくても動作いたしますので、コネクタが 44 ピンに足りない場合は、1 ピン側によせて半田付けしてください。

半田付けする際は、マスキングをし、周囲の部品に半田くず、半田ボール等付着しない様十分にご注意ください。

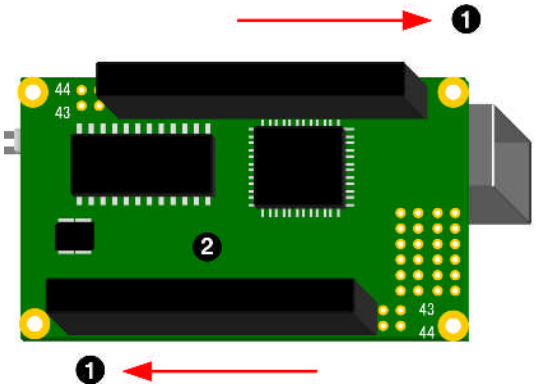


図 4.3. コネクタの半田付け

❶ 1 ピン側によせる

❷ 半田付けする面に注意

足を取り付けます。4 ヶ所にスペーサを取り付け、ネジ締めしてください。

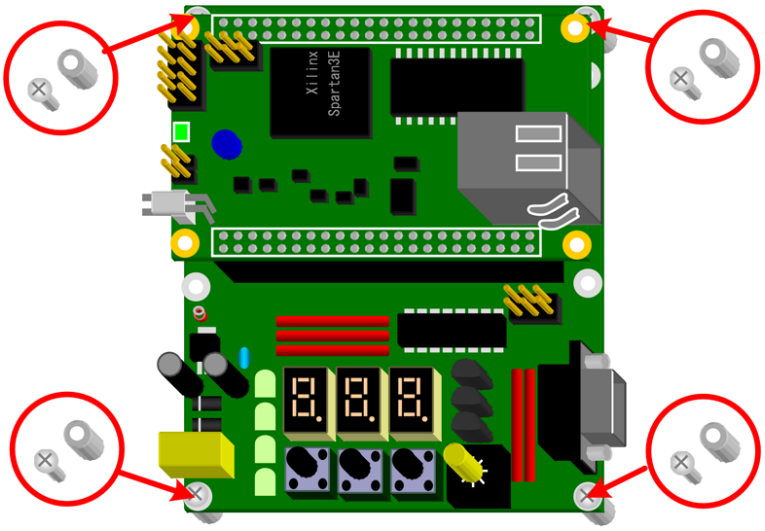


図 4.4. スペーサ取り付け

SZ410 をお使いの場合、SZ410 対応品シールが貼ってある LED/SW ボードをお使いください。 SZ410 は他の SUZAKU に比べ消費電流が多いため SZ410 対応品でない LED/SW ボードでは正常動作しないことがあります。

5.SUZAKU+LED/SW ボードを動かす

電源を入れて出荷状態の SUZAKU スターターキット(SUZAKU ＋ LED/SW ボード)を動かします。出荷状態では、フラッシュメモリに Linux が OS として入り、FPGA に今回最終目標とするスロットマシンが入っています。SUZAKU がどのような動きをするのか実際に体験してください。

SUZAKU はジャンパによりブートローダモード、オートブートモード、FPGA コンフィギュレーション待ちの 3 つの状態に設定できます。JP1 は起動モードジャンパ、JP2 は FPGA プログラム用ジャンパです。

ここではブートローダモードとオートブートモードで SUZAKU スターターキットを動かします。

表 5.1. ジャンパの設定と起動時の動作

JP1	JP2	起動時の動作	起動モード
ショート	オープン	ファーストブートローダ BBoot を起動	ブートローダモード
オープン	オープン	Linux カーネルを起動	オートブートモード
-	ショート	何も起動しません	FPGA コンフィギュレーション待ち

フラッシュメモリの中身が SUZAKU スターターキット出荷状態以外の場合、image リージョンおよび fpga リージョンを書き換える必要があります。Linux のイメージファイルは付属 CD-ROM の`"\suzaku-starter-kit\image\image-sz***-sil.bin"`に収録されています。FPGA ファイルは付属 CD-ROM の `"\suzaku-starter-kit\fpga\x.x\sz***\sz***-add_slot-yyyymmdd.zip"`の default_bit_file フォルダの中に収録されています。また、SUZAKU 公式サイト[http://suzaku.atmark-techno.com/series/stk/download]よりダウンロードすることも出来ます。書き換える方法については、「6. SUZAKU を書き換える」をご参照ください。

5.1. 接続方法

D-Sub9 ピン-10 ピン変換ケーブルを SUZAKU CON1 に、LAN ケーブルを SUZAKU L2 に接続してください。

SUZAKU CON1 に D-Sub9 ピン-10 ピン変換ケーブルを接続する際にはコネクタの白い三角マークと SUZAKU 基板上の白い三角マークを合わせるように接続します。コネクタの向きを反対に接続すると、機器を破損する恐れがありますので十分にご注意ください。

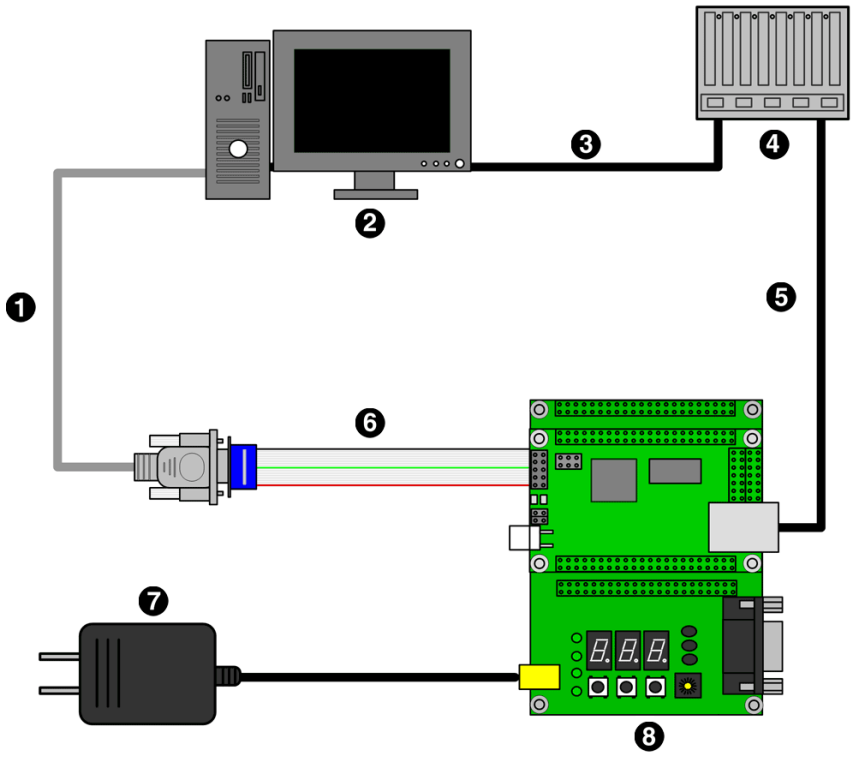


図 5.1. SUZAKU+LED/SW ボード配線

❶ D-Sub9 ピンクロスケーブル

❷ 作業用 PC

❸ LAN ケーブル

❹ HUB

❺ LAN ケーブル

❻ D-Sub9 ピン-10 ピン変換ケーブル (コネクタの向きに注意)

❼ AC アダプタ 5V

❽ SUZAKU ＋ LED/SW ボード

5.2. シリアル通信ソフトウェア

SUZAKU はシリアルポートをコンソールとして使用します。SUZAKU のコンソールから出力される情報を読み取ったり、SUZAKU のコンソールに情報を送ったりするには、シリアル通信ソフトウェアが必要です。ここでは Tera Term を使用した例を示します。

シリアル通信ソフトウェアを立ち上げ、シリアル通信の設定を行ってください。

・Baud rate	115200
・Data	8bit
・Parity	none
・Stop	1bit
・Flow control	none

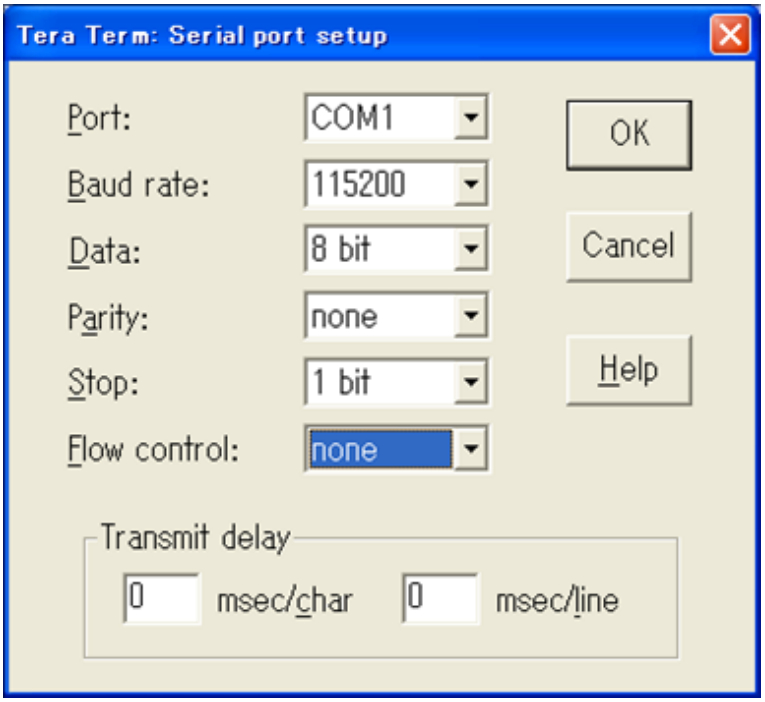


図 5.2. シリアルポート(Tera Term)の設定

5.3. ブートローダモードでスロットマシンを動かす

まず、ブートローダモードでスロットマシンを動かします。

JP1 にジャンパプラグをさしてショートさせてください。

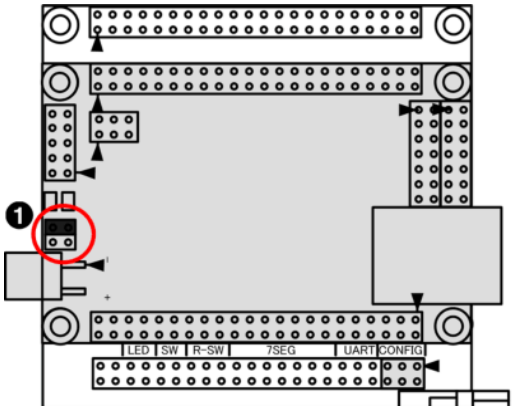


図 5.3. ブートローダモード ジャンパの設定

❶　JP1 ショート，JP2 オープン

5.3.1. 電源について

LED/SW CON6 から AC アダプタ 5V で電源を供給します。

SUZAKU CON6 からは絶対に電源を供給しないでください。電源がショートし、機器を破損する可能性があります。また、改造等により電源を外部から供給等行わないでください。SUZAKU と LED/SW ボードは、電源シーケンスの関係から、お互いに電源を供給し合うような形になっているので、機器を破損する可能性があります。

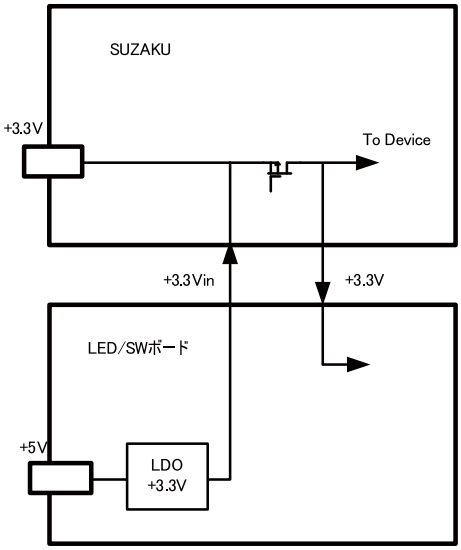


図 5.4. 電源系統

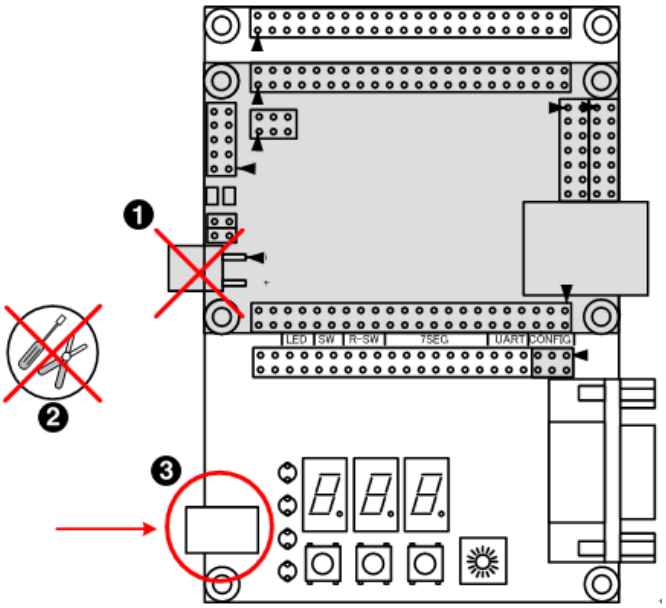


図 5.5. 電源ケーブル接続の諸注意

❶　絶対に入力しない

❷　電源の改造禁止

❸　電源入力はここから

5.3.2. スロットマシン起動

電源が供給されるとシリアル通信用ソフトウェアの画面に以下のメッセージが表示され、スロットマシンを動かすことができるようになります。

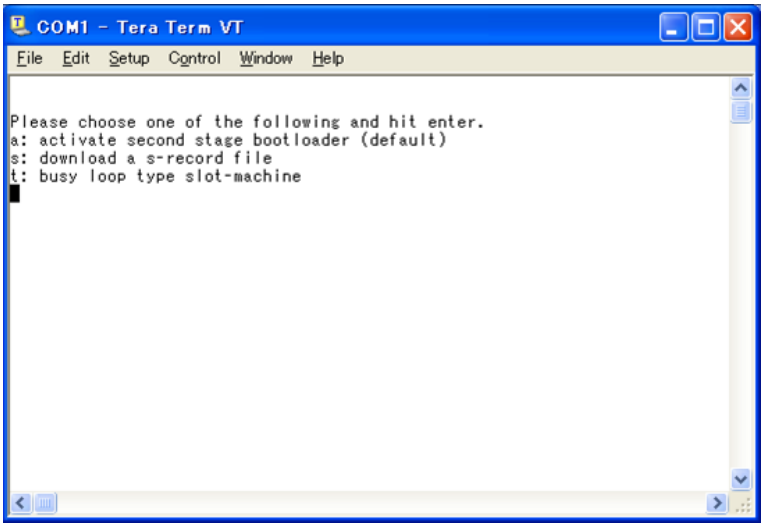


図 5.6. スロットマシンの起動

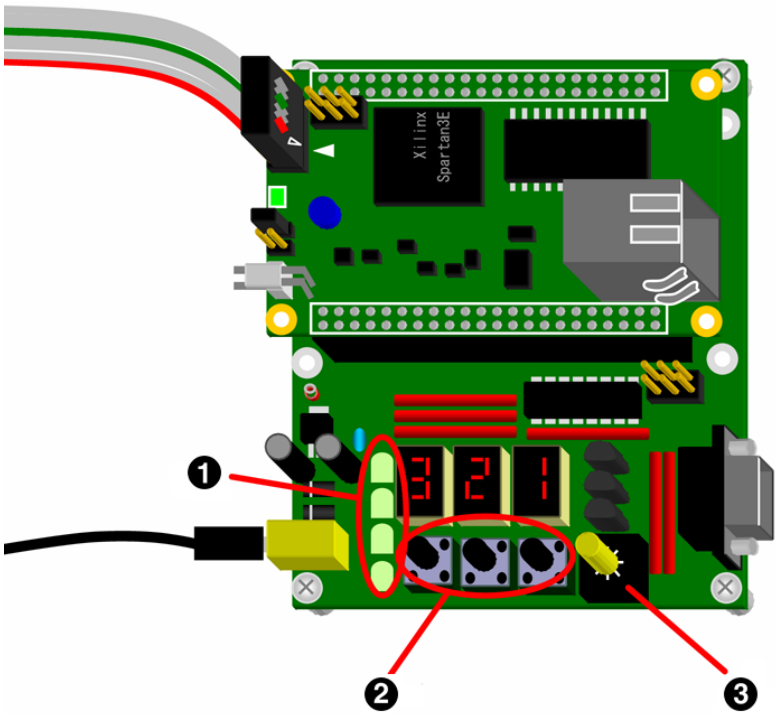


図 5.7. スロットマシンを動かしてみよう

❶ 数字がそろうと順次点灯

❷ 2 つ以上一緒に押すとスロートの回転が始まる

1 つ押すと、それぞれ対応する 7 セグメント LED の回転が止まる

❸ 0→1→2 とまわすと、数字の回転が速くなる

5.4. オートブートモードで Linux を動かす

次にオートブートモードで Linux を動かします。

JP1、JP2 がオープンになっていることを確認してください。

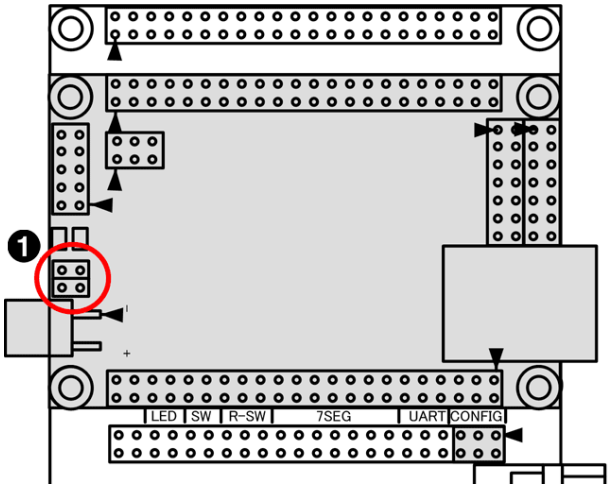


図 5.8. オートブートモード ジャンパの設定

❶ JP1、JP2 オープン

5.4.1. Linux の起動

シリアル通信用ソフトウェアが起動されていることを確認してから AC アダプタ 5V を接続し、電源を供給してください。シリアル通信用ソフトウェアの画面に Linux の起動ログが表示されます。

例 5.1. SUZAKU の起動ログ(SZ130 の場合)

```
Linux version 2.4.32-uc0 (build@sv-build) (gcc version 3.4.1 ( Xilinx EDK 9.1
Build EDK_J.19 121007 )) #1 2008 年 3 月 26 日 水曜日 19:40:27 JST
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
CPU: MICROBLAZE
Kernel command line:
Console: xmbserial on UARTLite
Calibrating delay loop... 25.65 BogoMIPS
Memory: 32MB = 32MB total
Memory: 29448KB available (957K code, 2001K data, 44K init)
Dentry cache hash table entries: 4096 (order: 3, 32768 bytes)
Inode cache hash table entries: 2048 (order: 2, 16384 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
```

```
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Microblaze UARTlite serial driver version 1.00
ttyS0 at 0xffff2000 (irq = 1) is a Microblaze UARTlite
ttyS1 at 0xffffa600 (irq = 3) is a Microblaze UARTlite
Starting kswapd
xgpio #0 at 0xFFFFA000 mapped to 0xFFFFA000
Xilinx GPIO registered
sil7segc (1.0.1): 7seg-LED Driver of SUZAKU I/O Board -LED/SW- for CGI demo.
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
eth0: LAN9115 (rev 1150001) at ffe00000 IRQ 2
Suzaku MTD mappings:
  Flash 0x800000 at 0xff000000
flash: Found an alies 0x800000 for the chip at 0x0, ST M25P64 device detect.
Creating 7 MTD partitions on "flash":
0x00000000-0x00800000 : "Flash/All"
0x00000000-0x00100000 : "Flash/FPGA"
0x00100000-0x00120000 : "Flash/Bootloader"
0x007f0000-0x00800000 : "Flash/Config"
0x00120000-0x007f0000 : "Flash/Image"
0x00120000-0x00420000 : "Flash/Kernel"
0x00420000-0x007f0000 : "Flash/User"
FLASH partition type: spi
uclinux[mtd]: RAM probe address=0x80125a5c size=0x1bf000
uclinux[mtd]: root filesystem index=7
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 4096)
VFS: Mounted root (romfs filesystem) readonly.
Freeing init memory: 44K
Mounting proc:
Mounting var:
Populating /var:
Running local start scripts.
Mounting /etc/config:
Populating /etc/config:
flatfsd: Created 4 configuration files (149 bytes)
Setting hostname:
Setting up interface lo:
Starting DHCP client:
Starting inetd:
Starting thttpd:

SUZAKU-S.SZ130-SIL login:
```

5.4.2. ログイン

表示されている SUZAKU のログインプロンプトから root ユーザでログインします。パスワードの初期設定は**"root"**です。

表 5.2. SUZAKU 初期設定時のユーザとパスワード

ユーザ名	パスワード	権限
root	root	特権ユーザ

5.4.3. ネットワークの設定

出荷状態の SUZAKU は DHCP で IP を取得するように設定されています。お使いの環境に DHCP サーバがない場合は固定 IP を割り当てる必要があります。以下のコマンドを入力し、固定 IP を割り当ててください。以下の例の 192.168.11.234 の部分には適当な IP アドレスを入力してください。固定 IP を割り当てる時は SUZAKU 上の特権ユーザで実行してください。

例 5.2. 固定 IP アドレスの割り当て

```
# ifconfig eth0 down
# ifconfig eth0 192.168.11.234
```

ネットワークの設定は以下のコマンドで表示されます。

例 5.3. ネットワークの設定の表示

```
#ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:11:0C:12:34:56
          inet addr:192.168.11.234  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:114 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
```

5.4.4. ウェブ

出荷状態の SUZAKU では、thttpd という小さな HTTP サーバが起動しています。先ほど確認した IP アドレス（例では 192.168.11.234）にお使いのウェブブラウザでアクセスすることで、動作確認ができます。"http://IP アドレス"にアクセスしてください。

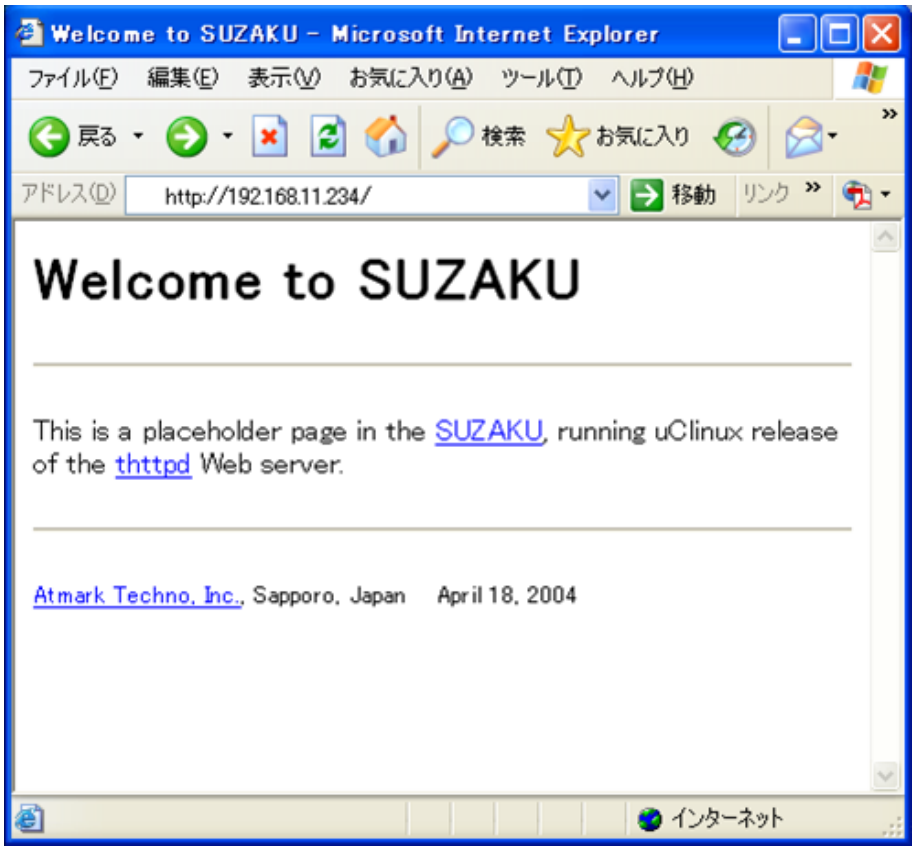


図 5.9. SUZAKU Web Page

さらに 7 セグメント LED を制御できる CGI が入っています。"http://IP アドレス/7seg-led-control.cgi"にアクセスしてください。

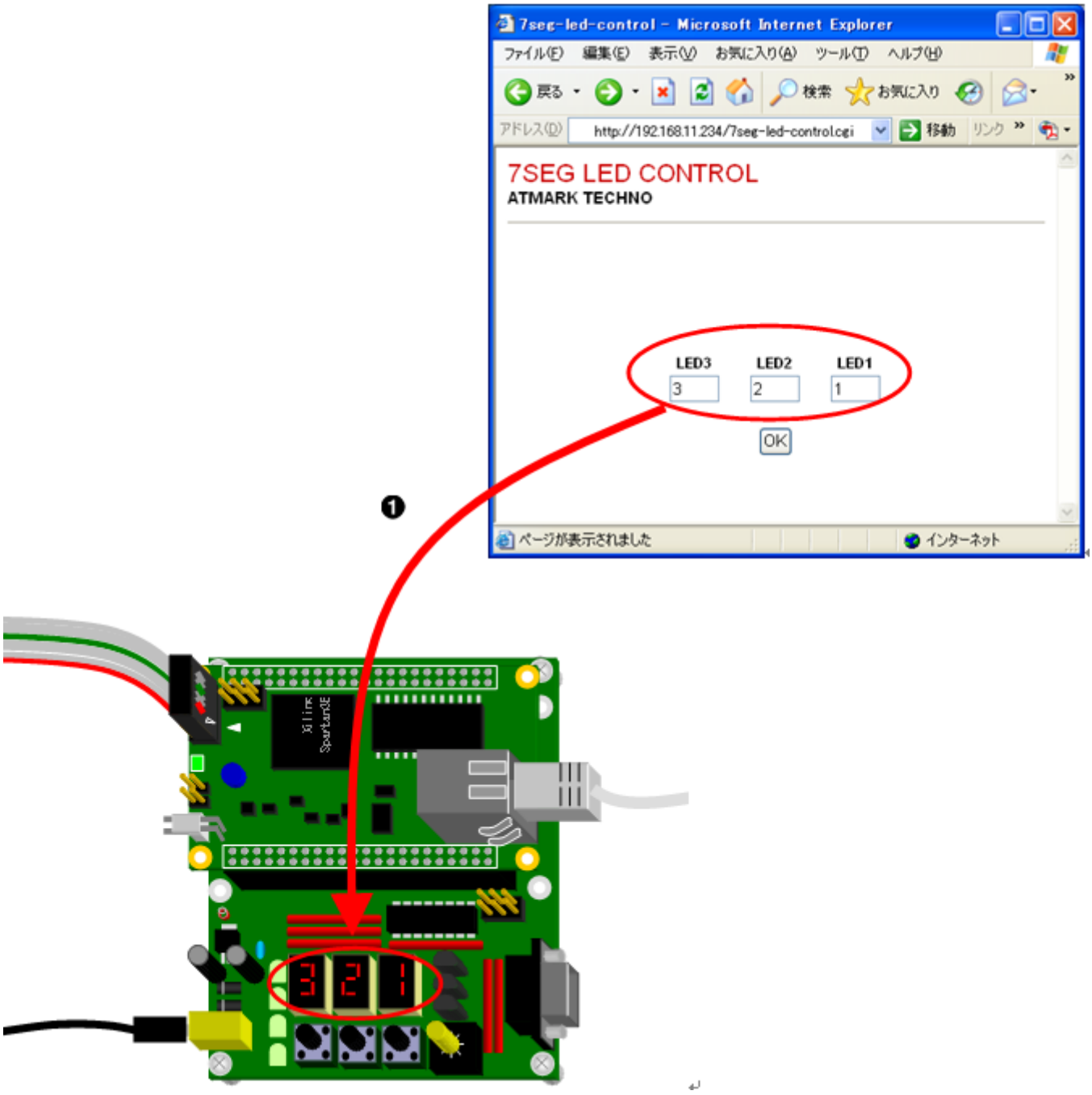


図 5.10. CGI を動かしてみる

❶　1~F(16 進数)の数字を設定して[OK]をクリックすると、7 セグメント LED に設定した数字が表示される。

5.4.5. 終了方法

SUZAKU スターターキットには電源ボタンがありません。終了するには、電源を切断する必要があります。AC アダプタをコンセントから抜いて終了してください。

5.5. SUZAKU のブートシーケンス

SUZAKU スターターキットを動かしてみましたがいかがだったでしょうか。SUZAKU のブートシーケンスについて説明をします。

①FPGA コンフィギュレーション
SUZAKU に電源を投入すると、まず FPGA にフラッシュメモリの中の FPGA リージョン(BRAM 内 BBoot を含む)をコンフィギュレーションします。(フラッシュメモリのリージョンについては「6. SUZAKU を書き換える」で説明をします。)

②BBoot
プログラムをスタートさせるリセットベクタのアドレス(MicroBlaze: 0x00000000 番地、PowerPC: 0xFFFFFFFC 番地)に SUZAKU では FPGA の BRAM を割り当てているので、コンフィギュレーション終了後、FPGA の BRAM 内の BBoot が動作します。

③Hermit
BBoot はフラッシュメモリの中のブートローダリージョン(Hermit)を SDRAM にコピーします。コピー終了後ブートローダ Hermit の先頭アドレスにジャンプするようになっており、Hermit が起動します。

④Linux
Hermit はフラッシュメモリの中のイメージリージョン(Linux)を SDRAM にコピーします。コピー終了後 Linux の先頭アドレスにジャンプするようになっており、Linux が起動します。Linux が起動後は Linux が SDRAM の全領域を使用し、Hermit は必要ないので上書きしてしまいます。

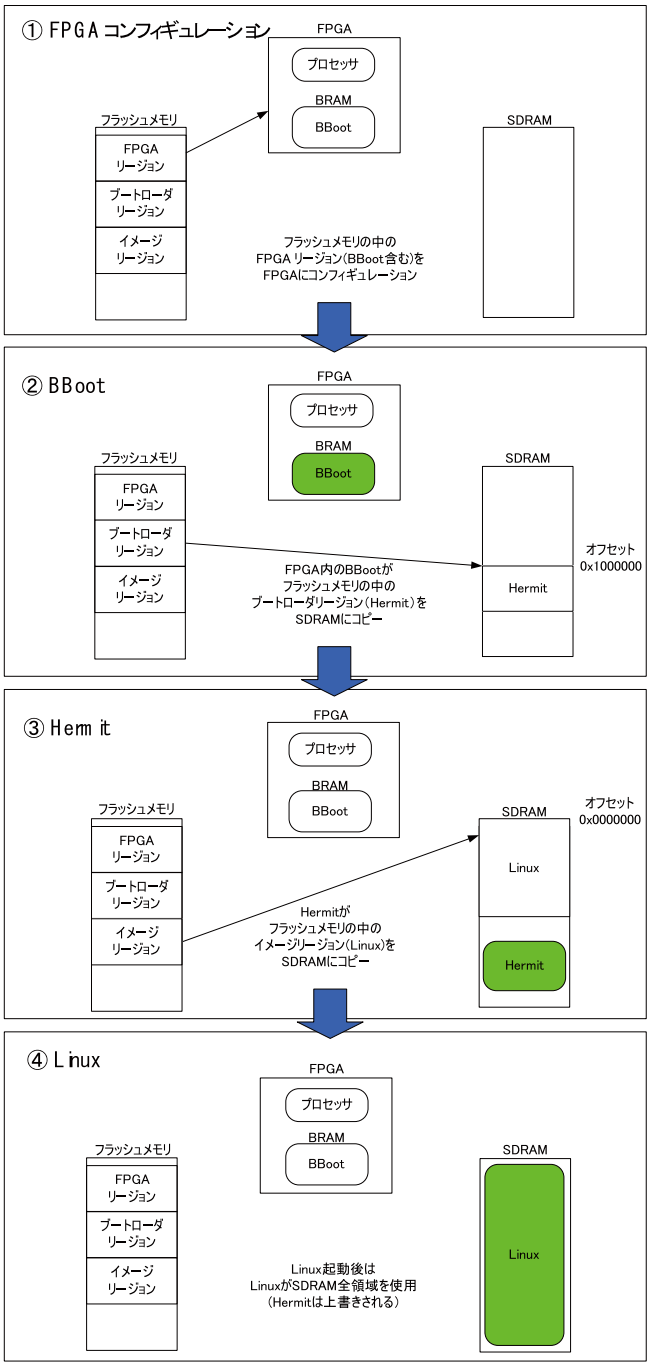


図 5.11. 2 段階ブート

SUZAKU スターターキットの BBoot は以下のようなフローで動作します。先ほどの動きを思い出して確認してみてください。

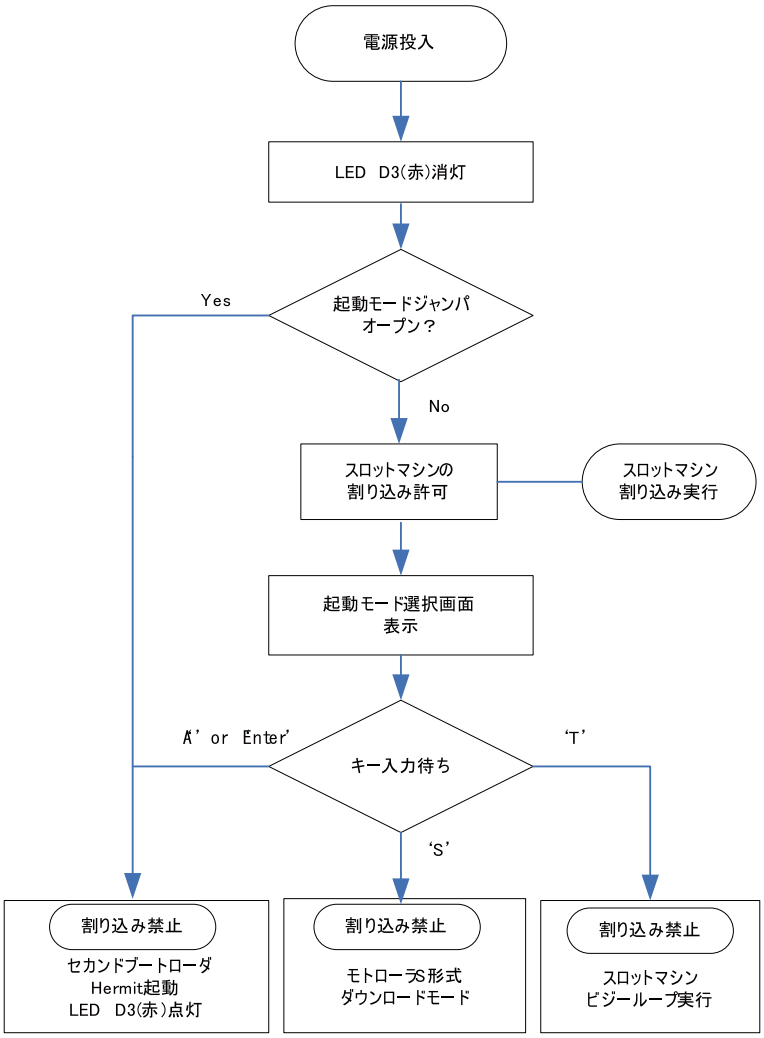


図 5.12. スターターキットの BBoot のフロー

2 段階ブート

SUZAKU のブートローダには以下の 2 つがあり、2 段階でブートします。

- ファーストブートローダ BBoot

  (FPGA の BRAM 内)

- セカンドブートローダ Hermit

2 段階のブートを使用しないことも可能ですが、SUZAKU ではあえて 2 段階でブートをしています。これには何らかの原因でフラッシュメモリの内容が書き換えられても、最低限のブートローダまでは動作させておきたかったという事と、ブートローダにより使用される BRAM のメモリ容量を最低限に抑えたかったという事の 2 つの理由があります。何らかの原因でフラッシュメモリの内容が書き換えられても、FPGA であれば JTAG からコンフィギュレーションデータをプログラムできるので、BBoot を復旧させることができ、フラッシュメモリの書き直しを行うことが出来ます。また、ブートローダは圧縮されたプログラムイメージを展開させたりするので、多くのメモリが必要となります。FPGA の BRAM は貴重なリソースなので、起動時にしか実行されないブートローダのために多くの容量を消費することを避け、セカンドブートローダ Hermit にこれらの機能を持たせることにしました。

6.SUZAKU を書き換える

SUZAKU で開発するためには SUZAKU を書き換える作業が必須となります。ここでは SUZAKU の書き換えかたを説明します。SUZAKU を書き換える = フラッシュメモリを書き換えることになります。フラッシュメモリには SUZAKU の基礎となる様々なデータが書き込まれています。

フラッシュメモリは大きく、FPGA リージョン、ブートローダリージョン、イメージリージョン、コンフィグリージョンという領域に分割し、データを書き込んでいます。

- FPGA リージョン ：SUZAKU の FPGA コンフィギュレーションデータが書き込まれています。
- ブートローダリージョン ： ブートローダ Hermit のデータが書き込まれています。
- イメージリージョン ： Linux のカーネルやユーザランドが書き込まれています。
- コンフィグリージョン ： ネットワークの設定やパスワードが書き込まれています。

これらの領域はそれぞれ個別に書き換えることが出来ます。

通常 FPGA のコンフィギュレーションデータは FPGA メーカから出されているシリアル ROM に記憶し、FPGA に書き込みますが、SUZAKU は TE7720（「6.2.2. LBPlayer2 で書き換える」参照）や、Spartan3E の機能や CPLD を使って市販のフラッシュメモリの FPGA リージョンに記憶して書き込んでいます。これとは別に、一時的に JTAG から FPGA にコンフィギュレーションデータを書き込む方法もあります。この場合、電源を落とすとコンフィギュレーションデータは消えてしまうのですが、フラッシュメモリを書き換えるよりも速いので、デバッグ時には大変有効です。

6.1. フラッシュメモリマップ

フラッシュメモリのリージョンの区分は、製品毎に異なります。

6.1.1. SZ130

SZ130 のフラッシュメモリマップは以下のとおりです。

表 6.1. フラッシュメモリマップ（SZ130 Flash:8MB）

アドレス	リージョン	サイズ	説明
0x00000000 | 0x000FFFFF	FPGA	1MB	FPGA
0x00100000 | 0x0011FFFF	ブートローダ	128KB	ブートローダ Hermit
0x00120000 | 0x007EFFFF	イメージ	約 6.81MB	Linux カーネル・ユーザーランド
0x007F0000 | 0x007FFFFF	コンフィグ	64KB	コンフィグ

6.1.2. SZ010

SZ010 のフラッシュメモリマップは以下のとおりです。

表 6.2. フラッシュメモリマップ（SZ010 : 4MB）

アドレス	リージョン	サイズ	説明
0x00000000 | 0x0007FFFF	FPGA	512KB	FPGA
0x00080000 | 0x0009FFFF	ブートローダ	128KB	ブートローダ Hermit
0x000A0000 | 0x003EFFFF	イメージ	約 3.31MB	Linux カーネル・ユーザーランド
0x003F0000 | 0x003FFFFF	コンフィグ	64KB	コンフィグ

6.1.3. SZ030, SZ310

SZ030、SZ310 のフラッシュメモリマップは以下のとおりです。

表 6.3. フラッシュメモリマップ（SZ030, SZ310 : 8MB）

アドレス	リージョン	サイズ	説明
0x00000000 | 0x0000FFFF	フリー 1	64KB	
0x00010000 | 0x0007FFFF	フリー 2	448KB	
0x00080000 | 0x000FFFFF	FPGA	512KB	FPGA
0x00100000 | 0x0011FFFF	ブートローダ	128KB	ブートローダ Hermit
0x00120000 | 0x007EFFFF	イメージ	約 6.81MB	Linux カーネル・ユーザーランド
0x007F0000 | 0x007FFFFF	コンフィグ	64KB	コンフィグ

6.1.4. SZ410

SZ410 のフラッシュメモリマップは以下のとおりです。

表 6.4. フラッシュメモリマップ（SZ410 : 8MB）

アドレス	リージョン	サイズ	説明
0x00000000 | 0x000FFFFF	FPGA	1MB	FPGA
0x00100000 | 0x0011FFFF	ブートローダ	128KB	ブートローダ Hermit
0x00120000 | 0x007EFFFF	イメージ	約 6.81MB	Linux カーネル・ユーザーランド
0x007F0000 | 0x007FFFFF	コンフィグ	64KB	コンフィグ

以下に SUZAKU のそれぞれのリージョンの書き換えかたと使用ファイル、書き換えるときのフラッシュメモリの状態を示します。

表 6.5. SUZAKU の書き換えかた [1]

		書き込み方法	使用ファイル	フラッシュメモリの状態
FPGA		iMPACT	bit ファイル	何も書き込まれていなくて良い
フラッシュメモリ	FPGA リージョン	LBPlayer2	mcs ファイル	何も書き込まれていなくて良い
		SPI Writer	bit ファイル	何も書き込まれていなくて良い
		ダウンローダ Hermit	bin ファイル	FPGA、ブートローダリージョンに正常に書き込まれている
	ブートローダリージョン	ダウンローダ Hermit	bin ファイル	FPGA、ブートローダリージョンに正常に書き込まれている
		BBoot モトローラ S 形式	srec ファイル	FPGA リージョンに正常に書き込まれている
	イメージリージョン	ダウンローダ Hermit	bin ファイル	FPGA、ブートローダリージョンに正常に書き込まれている
		NetFlash	bin ファイル	FPGA、ブートローダ、イメージリージョンに正常に書き込まれている
	コンフィグリージョン	ダウンローダ Hermit	bin ファイル	FPGA、ブートローダリージョンに正常に書き込まれている
		NetFlash	bin ファイル	FPGA、ブートローダ、イメージリージョンに正常に書き込まれている

[1] 灰色になっているところは本書では説明しません。「SUZAKU ソフトウェアマニュアル」をご参照ください。

6.2. FPGA の書き換えかた

SUZAKU の FPGA にコンフィギュレーションデータを書き込むには、iMPACT を使って JTAG で FPGA に直接書き込む方法、LBPlayer2 もしくは SPI Writer を使ってフラッシュメモリの FPGA リージョンに記憶させて書き込む方法、ダウンローダ Hermit でフラッシュメモリの FPGA リージョンに記憶させて書き込む方法があります。ここでは iMPACT と LBPlayer2、SPI Writer での書き換えかたについて説明いたします。ダウンローダ Hermit の使い方については「6.4.1. ダウンローダ Hermit で書き換える」を参考にして下さい。

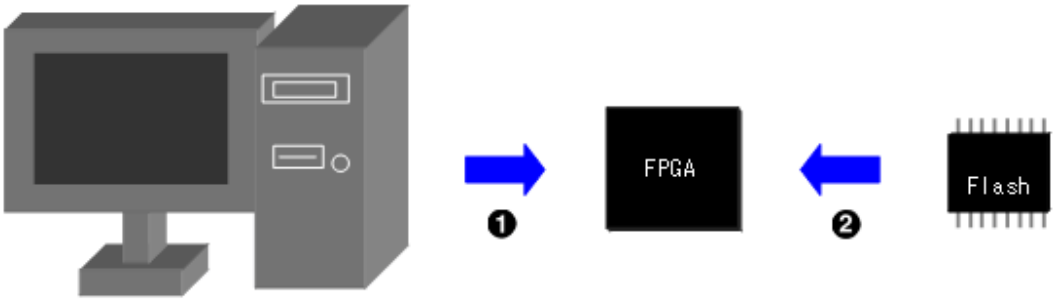


図 6.1. FPGA の書き込み

❶ JTAG で書き込み

❷ フラッシュメモリで書き込み

6.2.1. iMPACT で書き換える

iMPACT を使ってコンフィギュレーションデータを書き込む方法を説明します。iMPACT は ISE 付属のツールです。iMPACT で書き込むと、Xilinx の FPGA が SRAM ベースのためコンフィギュレーションは速いですが、電源を切るたびにコンフィギュレーションし直さなければなりません。

6.2.1.1. 書き込み準備

まず、SUZAKU JP2 にジャンパプラグをさし、ショートさせてください。JP2 をショートさせると、電源投入時 FPGA に対し、フラッシュメモリからのコンフィギュレーションデータの書き込みを停止させることができます。

SUZAKU CON7 に JTAG のダウンロードケーブル(Xilinx Parallel CableⅢ または Ⅳ）を接続し、LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。SUZAKU D3 のパワー ON LED(緑)が点灯しているか確認してください。

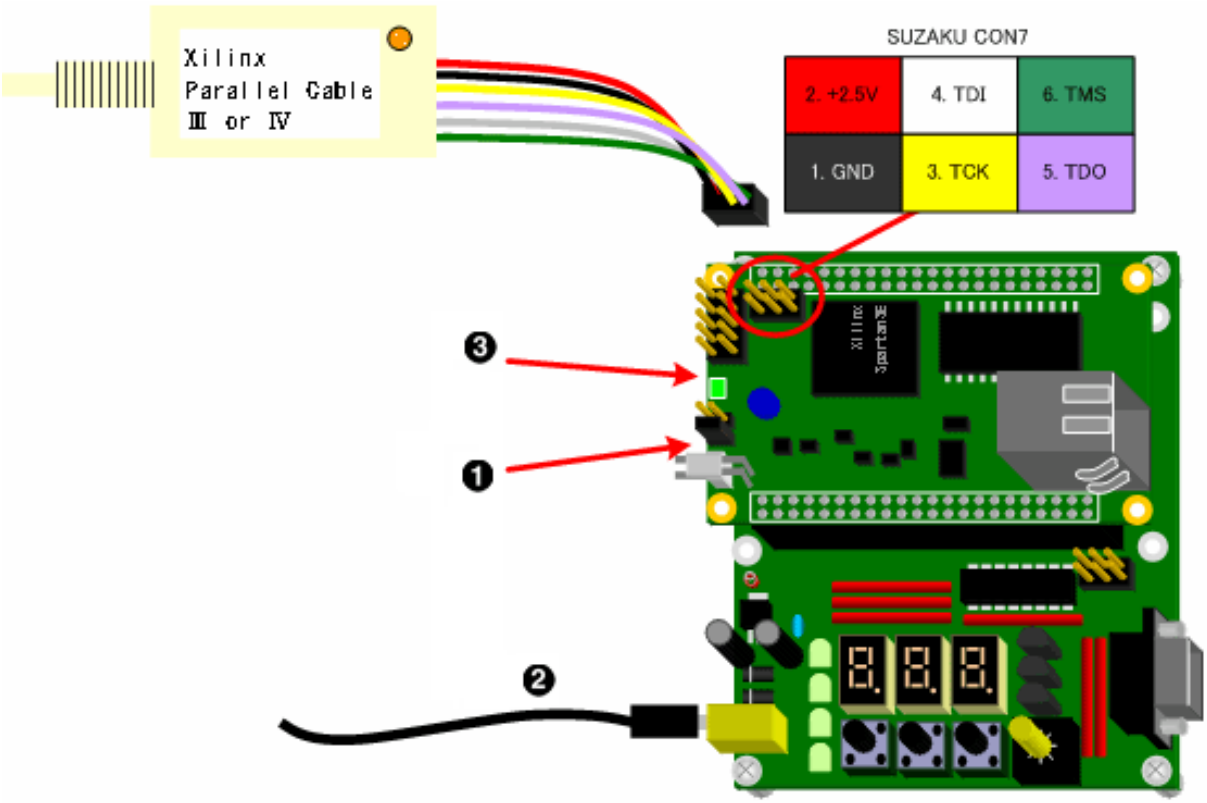


図 6.2. iMPACT 書き込み準備

❶　JP2 をショート

❷　LED/SW CON6 に AC アダプタ 5V

❸　パワー ON LED(緑)

6.2.1.2. iMPACT 立ち上げから書き込み

iMPACT を起動してください。iMPACT は[スタートメニュー]→[すべてのプログラム]→[Xilinx ISE Design Suite x.x]→[ISE]→[アクセサリ]→[iMPACT]から起動できます。

[create a new project (ipf)]にチェックを入れ、[OK]をクリックしてください。

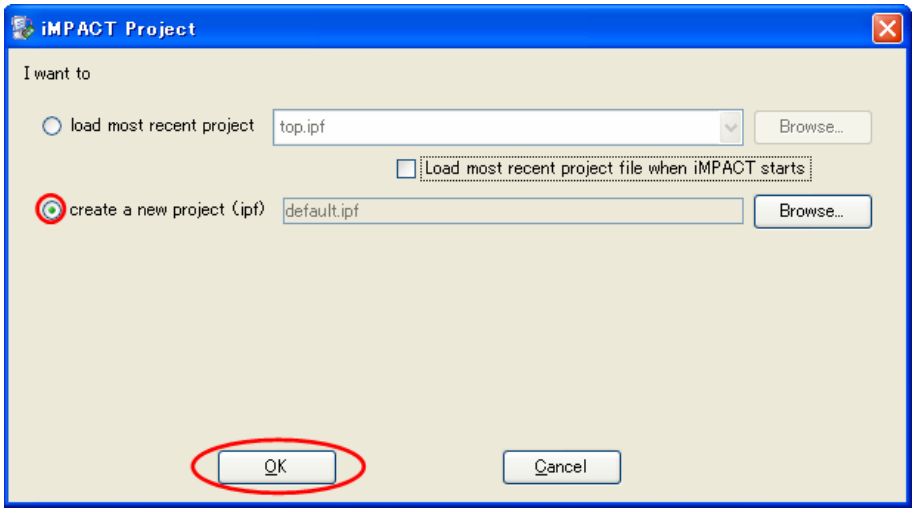


図 6.3. iMPACT 起動

[Configure devices using Boundary-Scan (JTAG)]にチェックを入れ、[Finish]をクリックしてください。

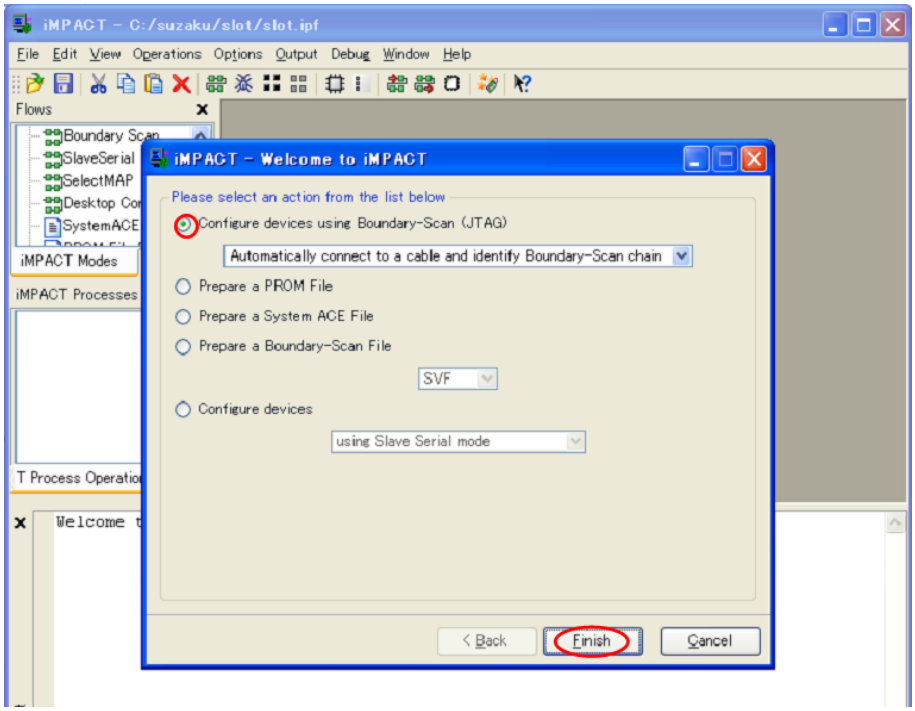


図 6.4. iMPACT 設定画面

書き込むデータを選択し、[Open]をクリックしてください。今回書き込むのは bit ファイルです。

SUZAKU のデフォルトの bit ファイルは付属 CD-ROM の`"\suzaku\fpga\x.x\sz***\sz***-yyyymmdd.zip"`を展開したフォルダの中の`"default_bit_file"`に収録されています。また、スロットマシンの bit ファイル(スターターキット出荷時の bit ファイル）は付属 CD-ROM の`"\suzaku-starter-kit\fpga\x.x\sz***\sz***-add_slot-yyyymmdd.zip"`を展開したフォルダの中の

`"default_bit_file"`に収録されています。

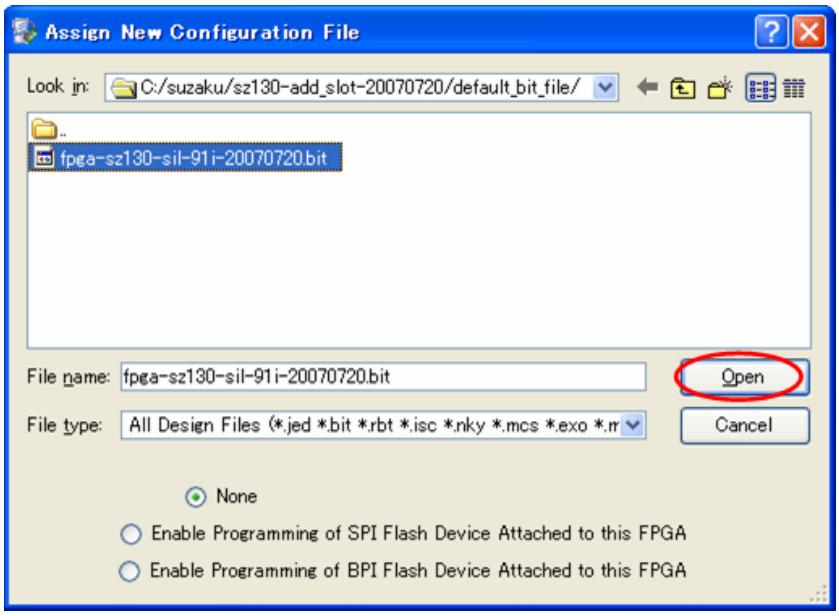


図 6.5. bit ファイル選択

何も変更せず[OK]をクリックして下さい。

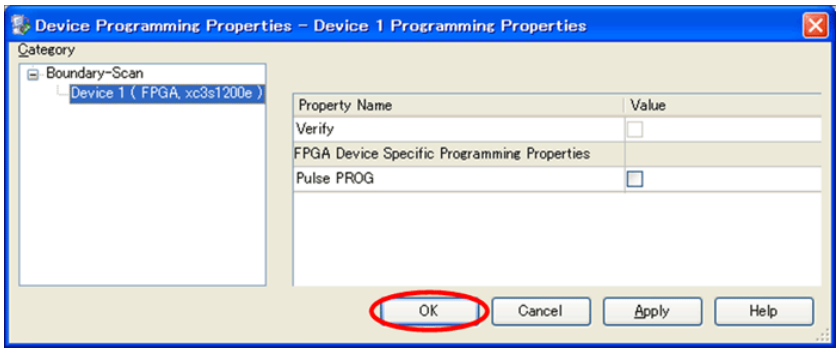


図 6.6. Program 設定

この時、Warning がでることがありますが[OK]をクリックして下さい。SZ310、SZ410 の場合はこの前に Add Virtex-・・・というウィンドウが立ち上がりますが、何も変更せず[OK]をクリックして下さい。

図 6.7. WARNING:iMPACT:2257

接続ミスがなく、SUZAKU の電源が入っていれば FPGA デバイスが発見されます。発見されなかった場合は接続を見直し、[File]→[New]をクリックし、[create a new project(ipf)]にチェックを入れてやり直してください。

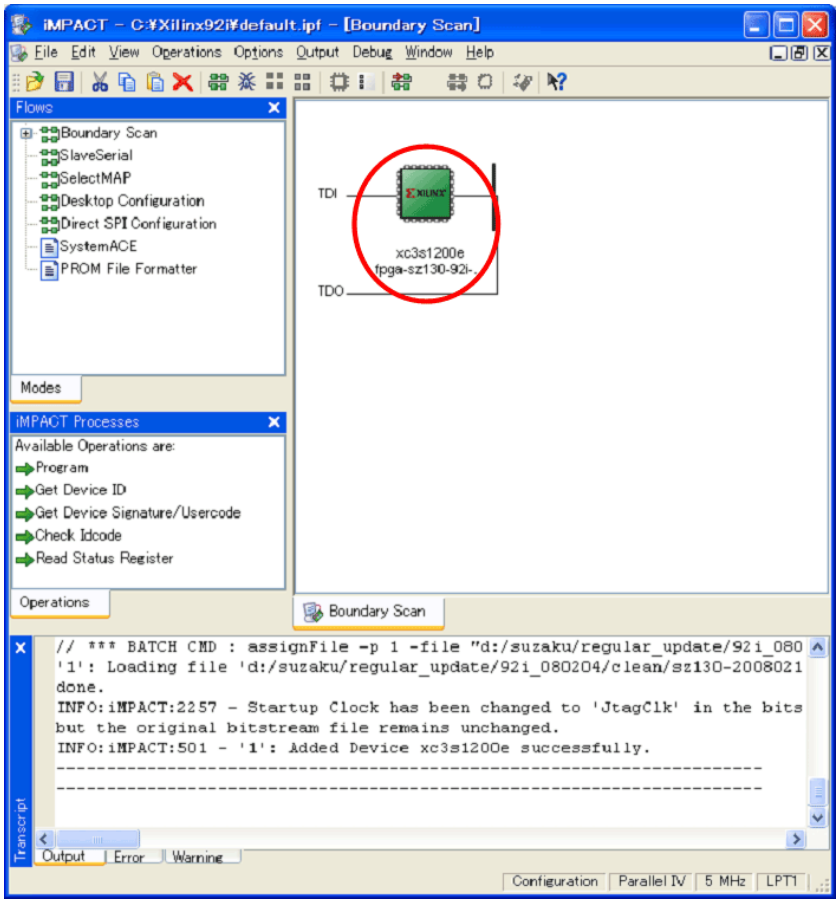


図 6.8. FPGA デバイス発見(SZ130 の場合)

デバイスが緑色(灰色の場合は一回クリックしてください。)であることを確認し、Program をダブルクリックしてください。書き込みが始まります。

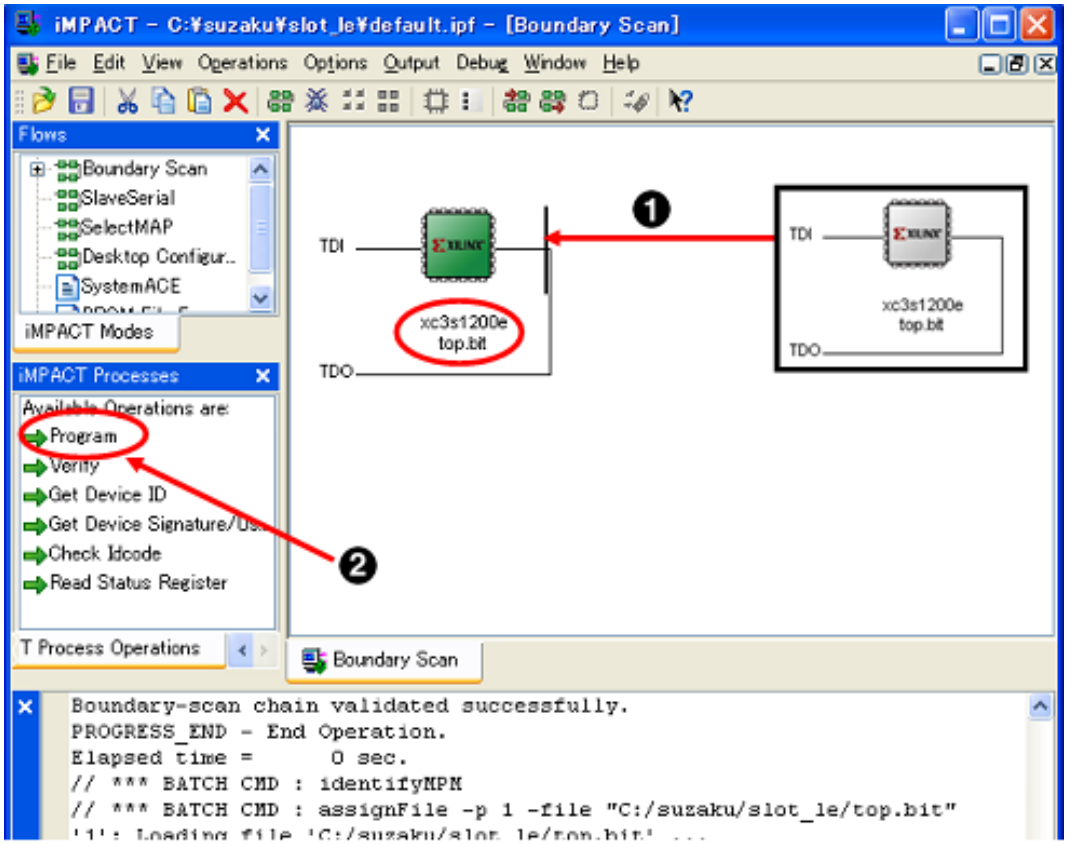


図 6.9. デバイス選択

❶　クリックすると緑色になる

❷　Program をダブルクリック

Program Succeeded と表示されれば、書き込み成功です。接続状態によっては書き込みに失敗することもありますので、失敗した場合は接続状態を確認し、再度書き込んでください。

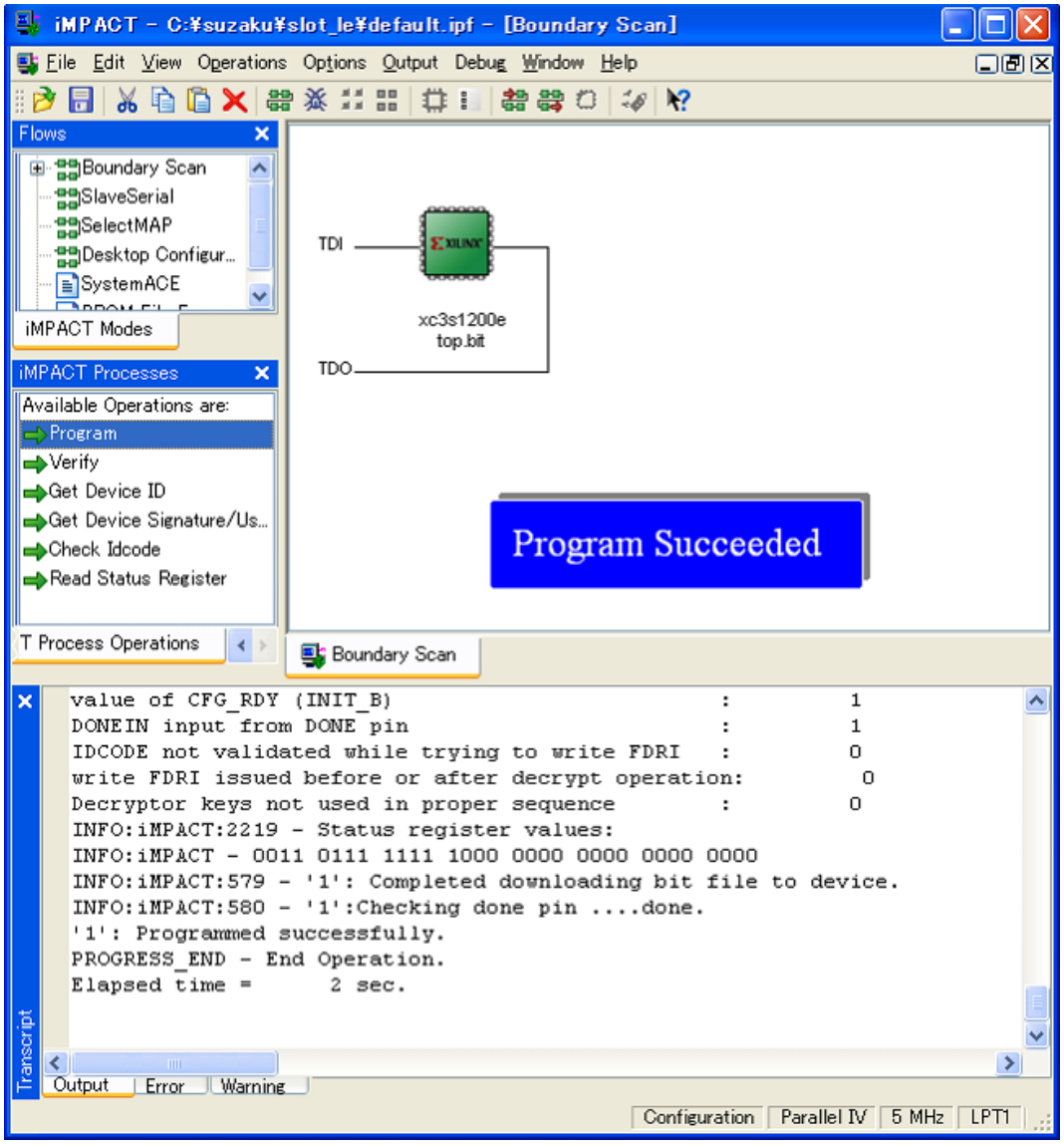


図 6.10. コンフィギュレーションデータ書き込み成功

6.2.1.3. iMPACT で書き換える 手順まとめ

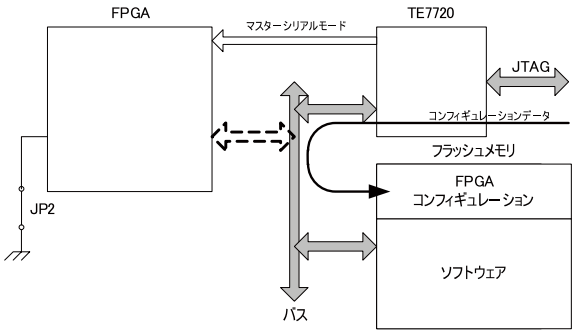
1. SUZAKU JP2 にジャンパプラグをさしてショートさせる
2. SUZAKU CON7 に JTAG ダウンロードケーブルを接続する
3. LED/SW CON6 に AC アダプタ 5V を接続し、電源投入
4. SUZAKU D3 のパワー ON LED(緑)が点灯していることを確認
5. iMPACT を立ち上げ、コンフィギュレーションデータ書き込み
6. 動作確認

※電源を切ると、書き込んだ内容は失われます。

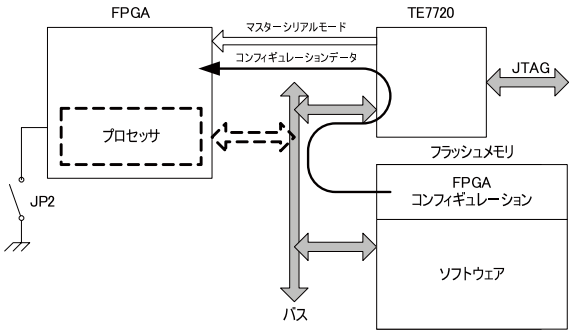
6.2.2. LBPlayer2 で書き換える

SZ010 SZ030 SZ310

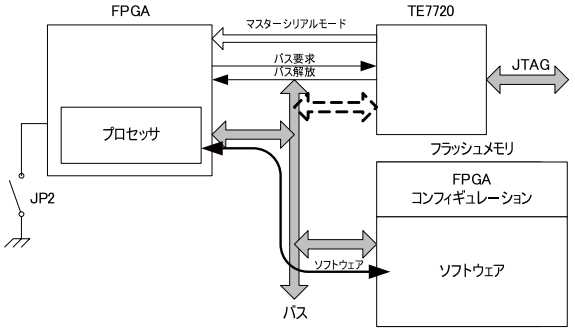
LBPlayer2 を使って書き換える方法を説明します。SZ010、SZ030、SZ310 は FPGA にコンフィギュレーションデータを書き込むデバイスとして TE7720(メーカ：東京エレクトロンデバイス)を実装しています。TE7720 は、JTAG から送られてくるデータをフラッシュメモリに記憶させ、再起動時にそのデータを読み込み、FPGA に書き込む IC です。TE7720 については東京エレクトロンデバイスのホームページ [http://www.teldevice.co.jp/]から詳細資料をダウンロードできます。



JTAGからTE7720経由でフラッシュメモリに書き込み



電源投入時フラッシュメモリからTE7720経由でFPGAに書き込み



FPGAの書き込み完了後、プロセッサがフラッシュメモリを使用

図 6.11. TE7720 の書き込み

6.2.2.1. 書き込み準備

SUZAKU JP2 にジャンパプラグをさし、ショートさせてください。JP2 をショートさせると、電源投入時 FPGA に対し、フラッシュメモリからの書き込みを停止させることができます。停止させないと書き込み不良等を起こしてしまいます。

LED/SW CON4 に JTAG のダウンロードケーブル(Xilinx Parallel Cable Ⅲ または Ⅳ)を接続し、LED/SW CON6 に AC アダプタ 5 V を接続し、電源を投入してください。SUZAKU D3 のパワー ON LED(緑)が点灯しているか確認してください。

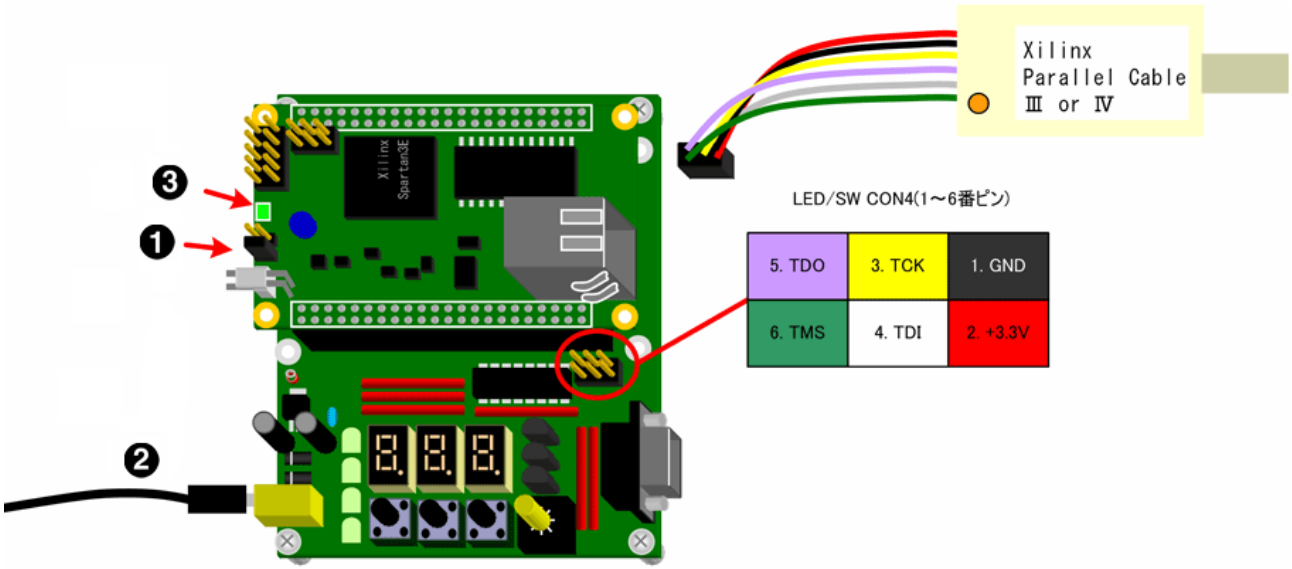


図 6.12. LBplayer2 書き込み準備

❶　JP2 をショート

❷　LED/SW CON5 に AC アダプタ 5V

❸　パワー ON LED(緑)

6.2.2.2. bit ファイルから mcs ファイルを作る

LBplayer2 で書き込めるファイルは mcs ファイルです。mcs ファイルは iMPACT で bit ファイルから変換して作成することが出来ます。mcs ファイルに変換する bit ファイルを準備してください。

SUZAKU のデフォルトの mcs ファイルは付属 CD-ROM の`"\suzaku\fpga\x.x\sz***\sz***-yyyymmdd.zip"`を展開したフォルダの中の`"default_bit_file"`に収録されています。また、スロットマシンの mcs ファイル(スターターキット出荷時の mcs ファイル）は付属 CD-ROM の`"\suzaku-starter-kit\fpga\x.x\sz***\sz***-add_slot-yyyymmdd.zip"`を展開したフォルダの中の

`"default_bit_file"`に収録されています。

iMPACT を起動してください。iMPACT は、[スタートメニュー]→[すべてのプログラム]→[Xilinx ISE Design Suite x.x]→[ISE]→[アクセサリ]→[iMPACT]から起動できます。

[create a new project (ipf)]にチェックを入れ、[OK]をクリックしてください。

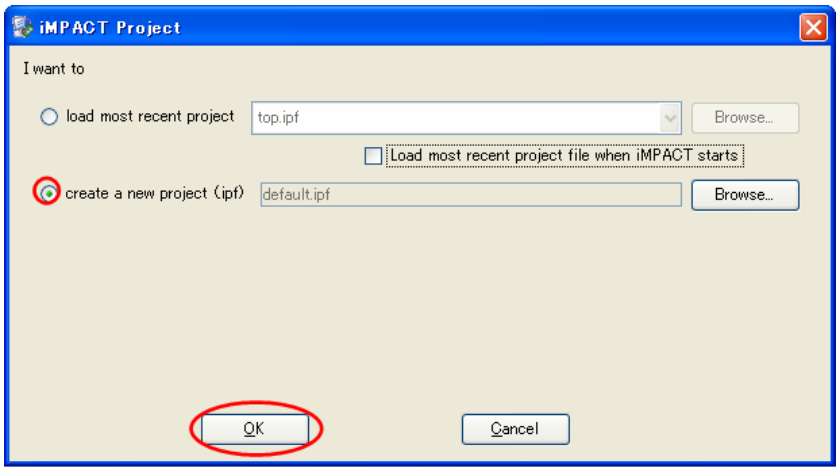


図 6.13. TE7720 iMPACT 起動

[Prepare a PROM File]を選択し、[Next]をクリックして下さい。

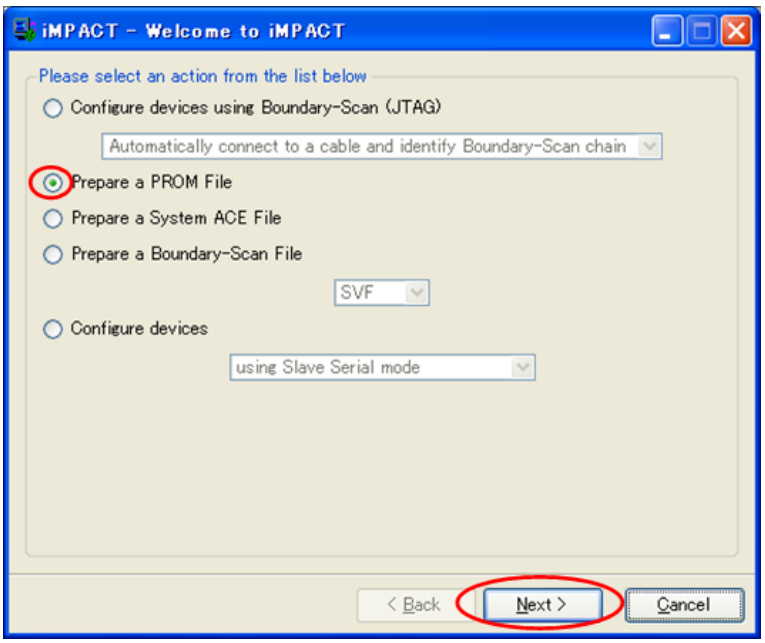


図 6.14. TE7720 iMPACT 立ち上げ

[Xilinx PROM]、[MCS]をチェックし、[PROM File Name]に作成する mcs ファイルの名前を入力し、[Location]に mcs ファイルの保存先を設定し、[Next]をクリックして下さい。

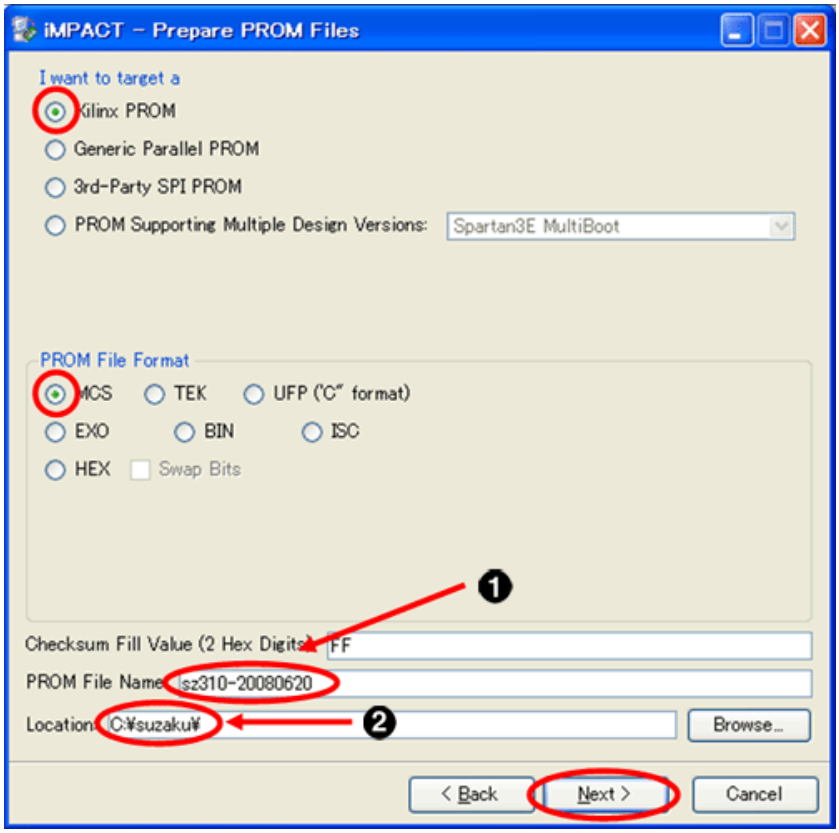


図 6.15. TE7720 iMPACT 設定

❶　ファイル名を入力

❷　保存先を入力

[Select a PROM]で[xc18v]→[xc18v04]を選択し、[Add]をクリックして下さい。

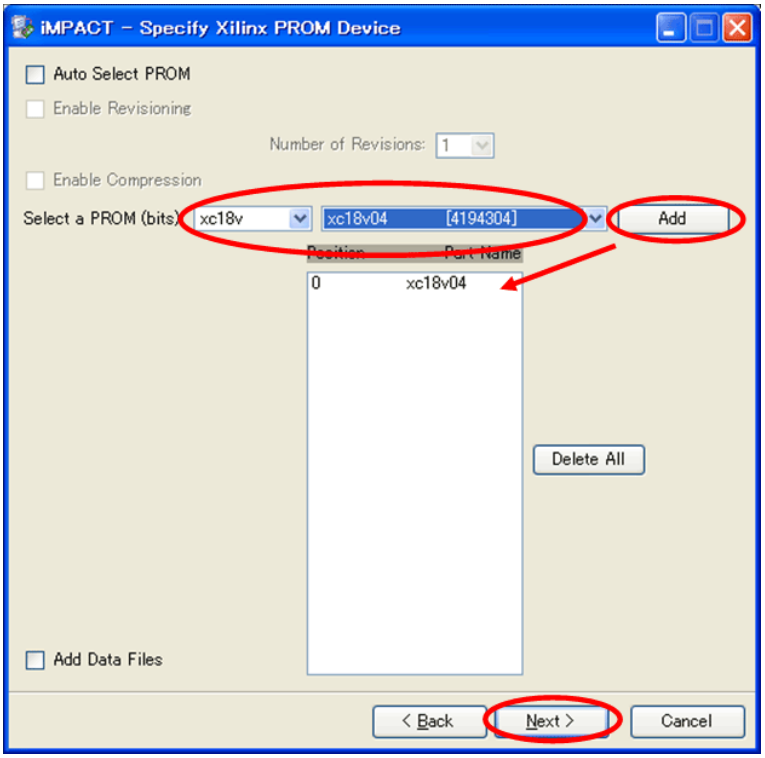


図 6.16. PROM の選択

確認画面が表示されます。間違いがなければ[Finish]をクリックして下さい。

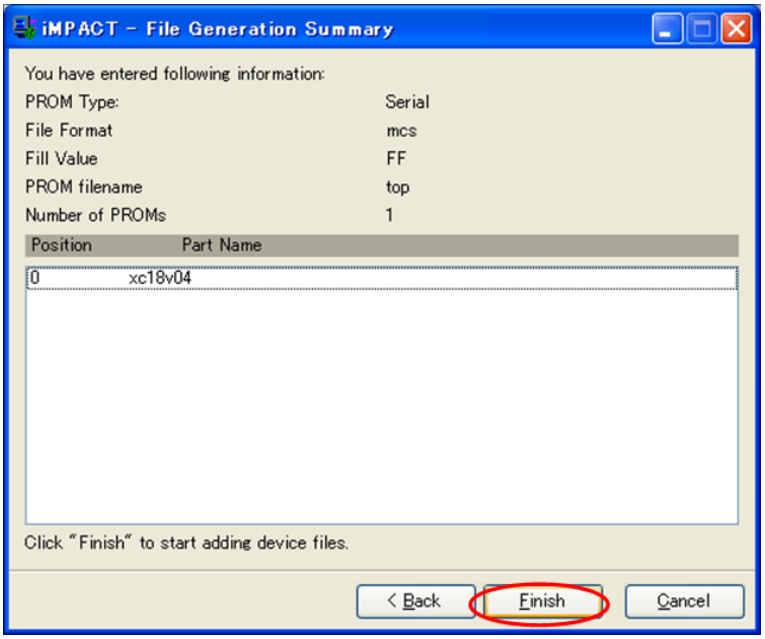


図 6.17. TE7720 確認画面

次の画面が表示されるので、[OK]をクリックして下さい。

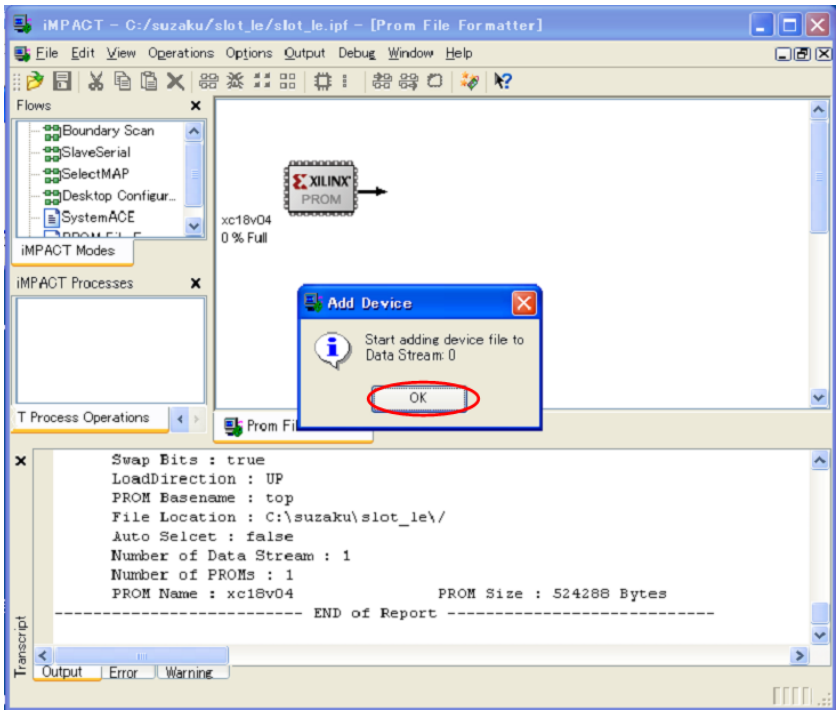


図 6.18. TE7720 デバイスファイル追加

mcs に変換する bit ファイルを選択し、[開く]をクリックして下さい。

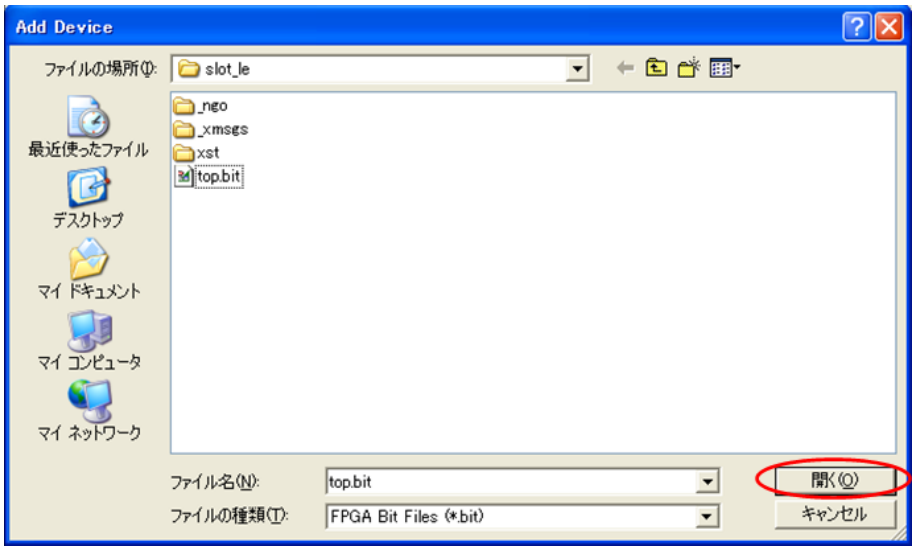


図 6.19. TE7720 bit ファイルを開く

他のデバイスを追加するか聞かれるので、[No]をクリックして下さい。

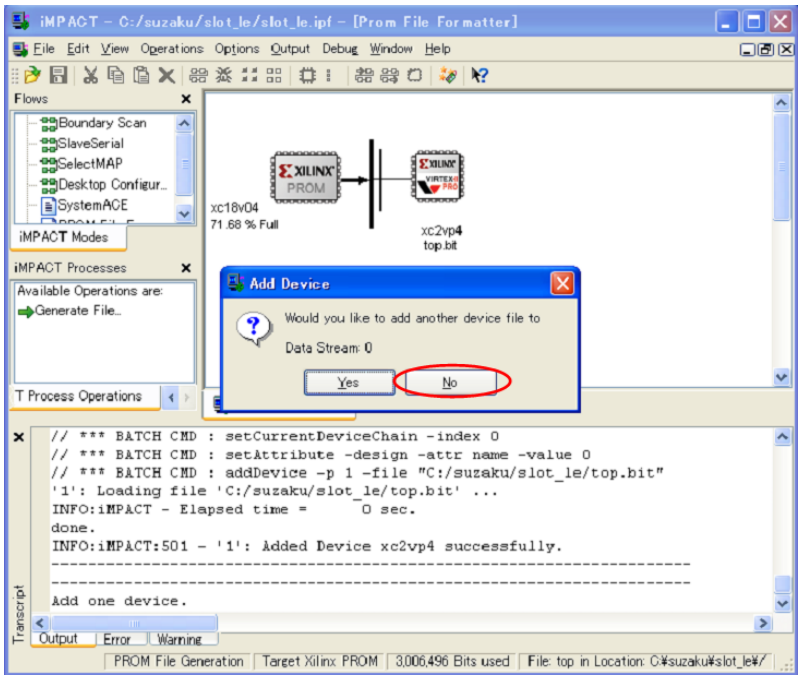


図 6.20. TE7720 デバイスファイルさらに追加

次の画面が表示されるので[OK]をクリックして下さい。

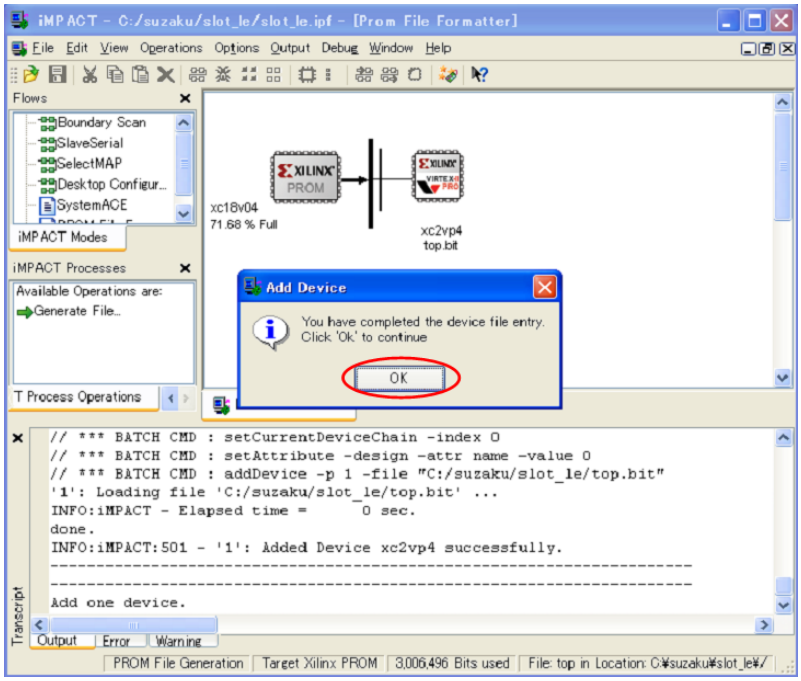


図 6.21. TE7720 準備完了

[Generate File...]をダブルクリックして下さい。PROM File Generation Succeeded と表示されたら、mcs ファイル作成完了です。

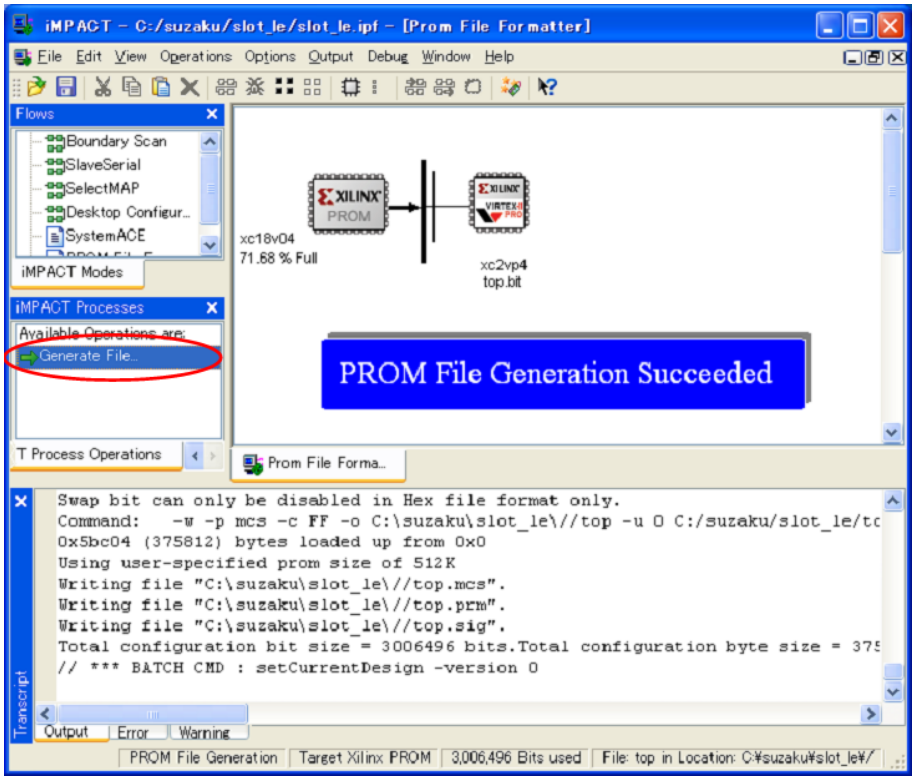


図 6.22. mcs ファイル出来上がり

iMPACT のバッチモード

mcs ファイルを作成するのに毎回 GUI で同じ作業を繰り返すのは面倒だと思ったことはないでしょうか。そんな人にはバッチファイルを作ることをお勧めします。以下に bit ファイルから mcs ファイルを作成する iMPACT のバッチファイルの一例を示します。以下の内容をテキストエディタ等で編集し、任意の名前をつけて保存してください。ここでは mcs.cmd とします。

```
setMode -pff
setSubmode -pffserial
addPromDevice -p 1 -name xc18V04
addDesign -version 0 -name 0
addDeviceChain -index 0
addDevice -p 1 -file top.bit
generate -format mcs -fillvalue FF -output top
quit
```

作成した `mcs.cmd` と同じフォルダに bit ファイル(`top.bit`)を置き、コマンドプロンプトを立ち上げてそのフォルダに移動し、以下のコマンドを実行して下さい。mcs ファイル(`top.mcs`)が出来上がります。iMPACT のバッチモードのコマンド詳細については iMPACT のヘルプをご参照ください。

```
> impact -batch mcs.cmd
```

6.2.2.3. LBPlayer2 立ち上げから書き込み

LBPlayer2 のフォルダを開いてください。"device.def"と"lbplay2.exe"の 2 つのファイルがあることを確認して、mcs ファイルを LBPlayer2 のフォルダの下にコピーしてください。

図 6.23. mcs ファイルコピー

❶　確認

❷　mcs ファイルを同じフォルダにおく

コマンドプロンプトを開き、LBPlayer2 のフォルダに移動し、以下のコマンドを実行してください。ドライバのエラーが出た場合の対処方法については後述の Tips をご参照ください。

```
> lbplay2 -deb [ファイル名].mcs
```

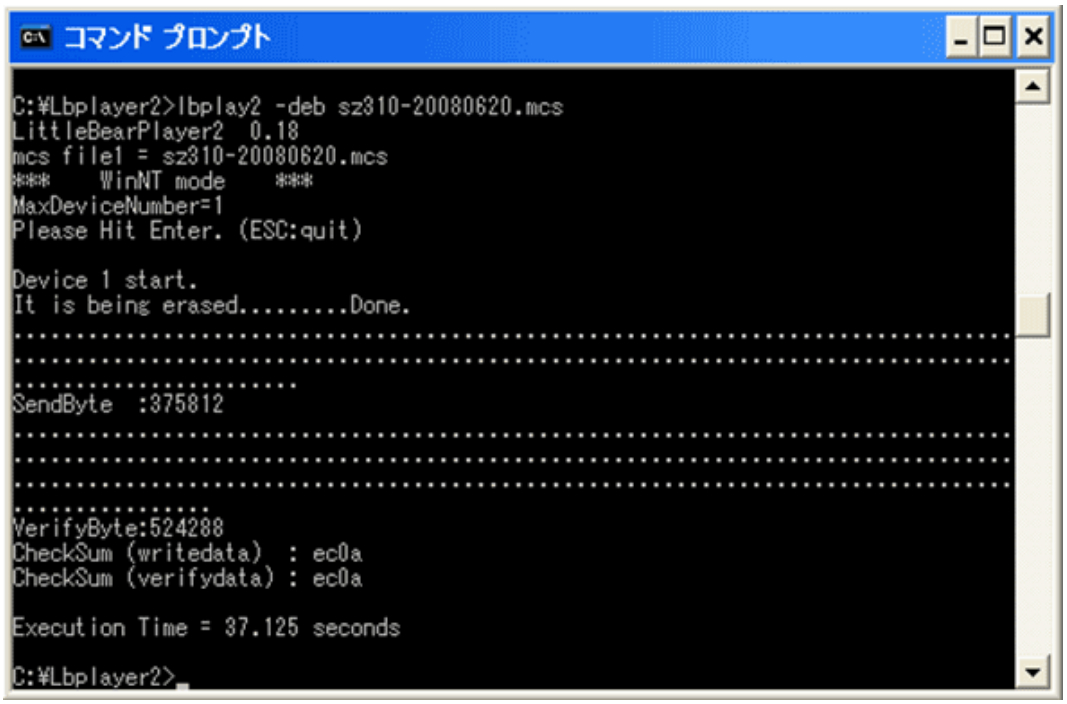


図 6.24. LBPlay2 実行

エラーが出なければ、書き込み完了です。書き込めたと思っても、エラーが出ていることがあるので、CheckSum 等をよく確認してください。

何らかの原因でエラーが発生した場合は、SUZAKU を動作させず、再び書き込みを行ってください。

LED/SW CON6 から AC アダプタ 5V を抜いて電源を切り、JP2 のジャンパプラグと LED/SW CON4 のダウンロードケーブルをはずしてください。再び LED/SW CON6 に AC アダプタ 5V を接続し、電源を再投入してください。

6.2.2.4. LBPlayer2 で書き換える 手順まとめ

1. SUZAKU JP2 にジャンパプラグをさしてショートさせる
2. LED/SW CON4 にダウンロードケーブルを接続する
3. LED/SW CON6 に AC アダプタ 5V を接続し、電源投入
4. SUZAKU D3 のパワー ON LED(緑)が点灯していることを確認
5. iMPACT を立ち上げ、mcs ファイルを作成
6. コマンドプロンプトを立ち上げ、LBPlayer2 でコンフィギュレーションデータを書き込む
7. LED/SW CON6 の AC アダプタ 5V をはずし、電源を切る
8. LED/SW CON4 のダウンロードケーブルをはずす
9. SUZAKU JP2 のジャンパプラグをはずす
10. LED/SW CON6 に AC アダプタ 5V を接続し、電源再投入
11. 動作確認

※電源を切っても、コンフィギュレーション内容は失われません。

書き込めたけど動かない？！

SZ310 のプロジェクトを ISE 8.1i で作っている場合、bit ファイルがうまく生成されません。コマンドプロンプト等を立ち上げ、プロジェクトフォルダに移動し、以下のコマンドで新たに bit ファイルを生成してください。(ここではプロジェクトフォルダを"sz310"、bit ファイルを"top.bit"、新しく生成する bit ファイルを"top_new.bit"としています)

```
> data2mem -bm sz310\implementation\xps_proj_bd.bmm
 -bt top.bit
 -bd sz310\ppc405_i\code\executable.elf
 tag bram -o b top_new.bit
```

LBPlayer2 ERROR

LBPlbay2 で書き込む際、

```
ERROR: Please check %windir%\system32\drivers\windrvr6.sys.
```

というエラーが発生する場合があります。

付属 CD-ROM の"\suzku\tools\LBPlay2_Release108.zip"を展開してください。展開後のフォルダの中に"Release205.zip"が入っているので、これをさらに展開してください。

展開後のフォルダの中にある"windrvr6.inf"、"windrvr6.sys を同じ名前のファイルがないことを確認し、Administrator 権限ユーザで以下のフォルダにコピーしてください。もし同じ名前のファイルがあった場合はバージョンを確認し、新しければコピーしてください。

- WindowsNT/2000 の場合 C:\WINNT\system32\drivers
- WindowsXP の場合 C:\WINDOWS\system32\drivers

コマンドプロンプトを立ち上げ、wdreg.exe のあるフォルダに移動し、以下のコマンドを実行してください。

```
> wdreg -inf [Windows インストールディレクトリ]\system32\drivers
\windrvr6.inf install
```

```
install: completed successfully
```

と表示されます。これでドライバがインストールされ、エラーが出なくなります。

6.2.3. SPI Writer で書き換える

SPI Writer を使って書き換える方法を説明します。

6.2.3.1. SZ130

SZ130

SZ130 の場合、コンフィギュレーションデータを M25P64(メーカ：ST マイクロエレクトロニクス)という SPI フラッシュメモリに記憶させ、再起動時に Spartan3E の SPI モードで FPGA にデータを書き込みます。SPI フラッシュメモリは SZ130 の裏面に実装されています。

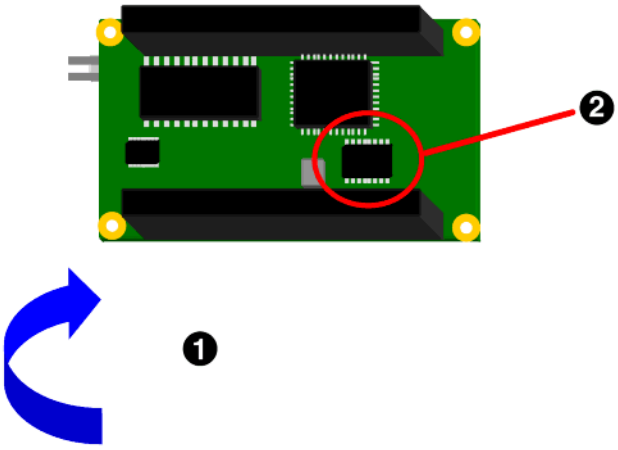


図 6.25. SZ130 の SPI フラッシュメモリの所在

❶ 裏

❷ SPI フラッシュメモリ

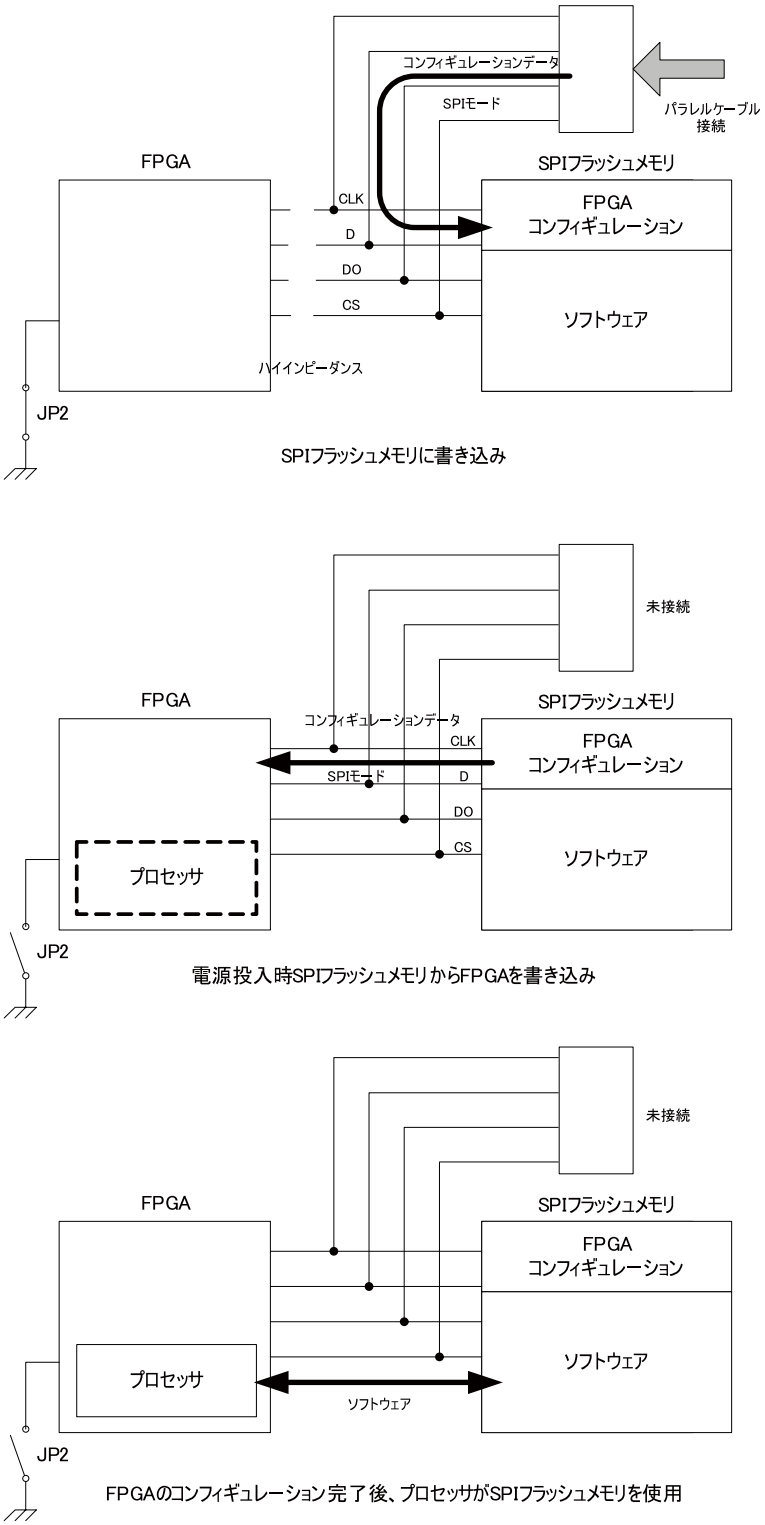


図 6.26. SPI モードの書き込み(SZ130)

6.2.3.2. SZ410

SZ410

SZ410 の場合、コンフィギュレーションデータを CPLD で M25P64(メーカ：ST マイクロエレクトロニクス)という SPI フラッシュメモリに記憶させ、再起動時に CPLD で SPI フラッシュメモリにコンフィギュレーションデータを書き込んでいます。CPLD および SPI フラッシュメモリは下図の位置に配置されています。

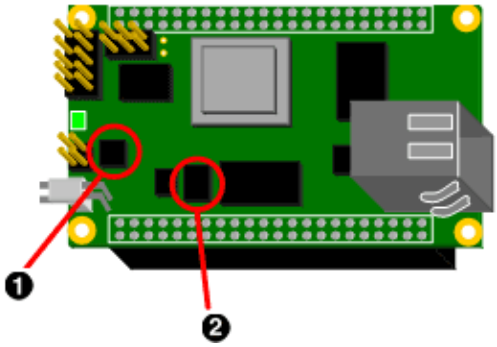


図 6.27. SZ410 の CPLD および SPI フラッシュメモリの所在

❶ CPLD

❷ SPI フラッシュメモリ

コンフィギュレーション時の CPLD の動作は以下の通りとなります。

1. JP2 をショートしてコンフィギュレーションデータを書き込むと、パラレルケーブルからのコンフィギュレーションデータを SPI フラッシュメモリにスルーで書き込む。

2. JP2 をオープンにして電源を投入すると、SPI のビットストリームをマスターシリアルに変換してコンフィギュレーションする。

3. コンフィギュレーション後は SPI フラッシュメモリのデータ線を開放し、SPI フラッシュメモリに FPGA の制御を渡す。

SUZAKU に実装している CPLD で使用している VHDL コードは、Xilinx の XAPP800 というドキュメントを元に作成しています。

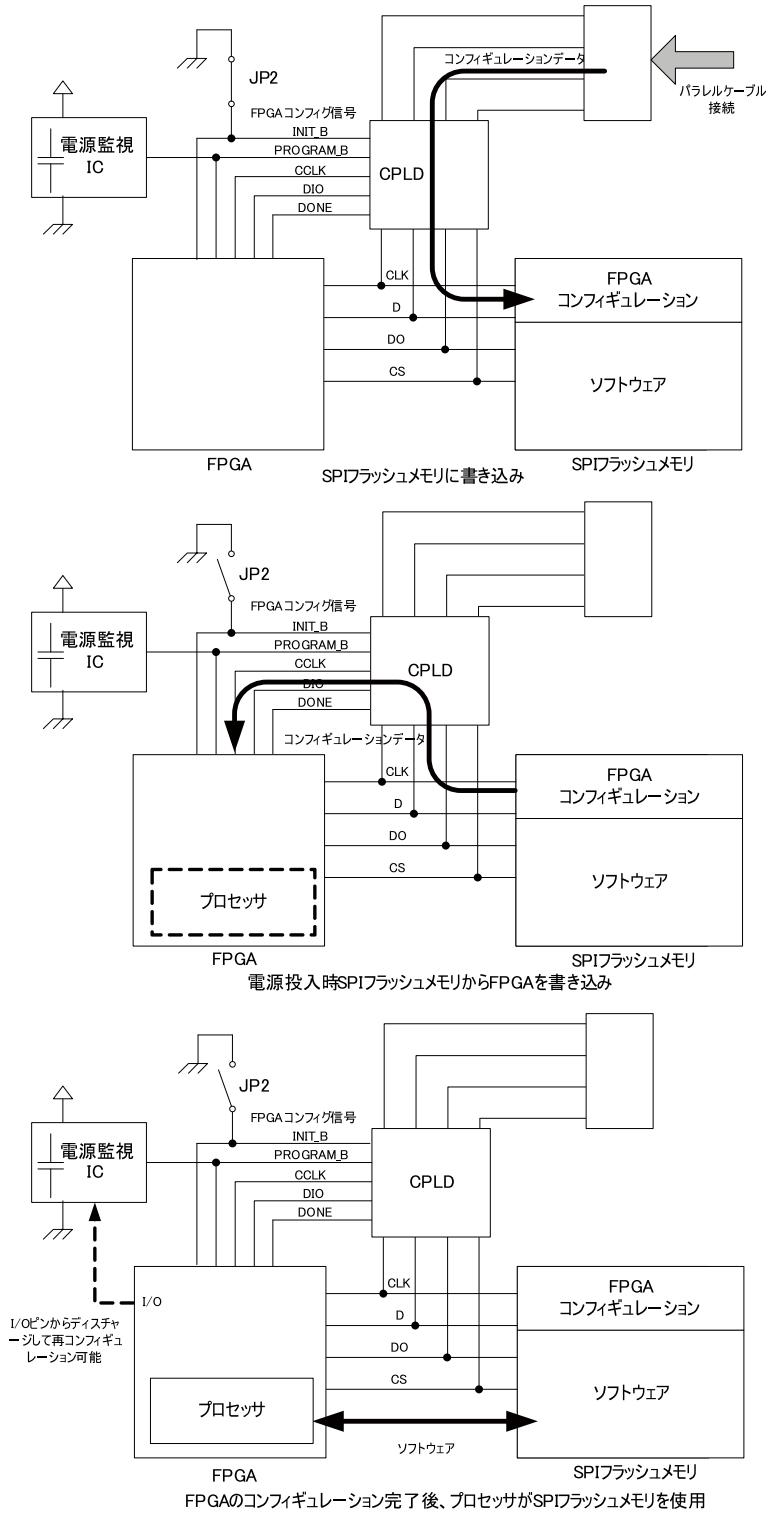


図 6.28. CPLD による書き込み(SZ410)

6.2.3.3. 書き込み準備

まず、SUZAKU JP2 にジャンパプラグをさし、ショートさせてください。JP2 をショートさせると、電源投入時 FPGA に対し、フラッシュメモリからのコンフィギュレーションを停止させることができます。コンフィギュレーションを停止させないと書き込み不良等を起こしてしまいます。

LED/SW CON4 に JTAG のダウンロードケーブル(Xilinx Parallel CableⅢ または Ⅳ）を接続し、LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。SUZAKU D3 のパワー ON LED(緑)が点灯しているか確認してください。

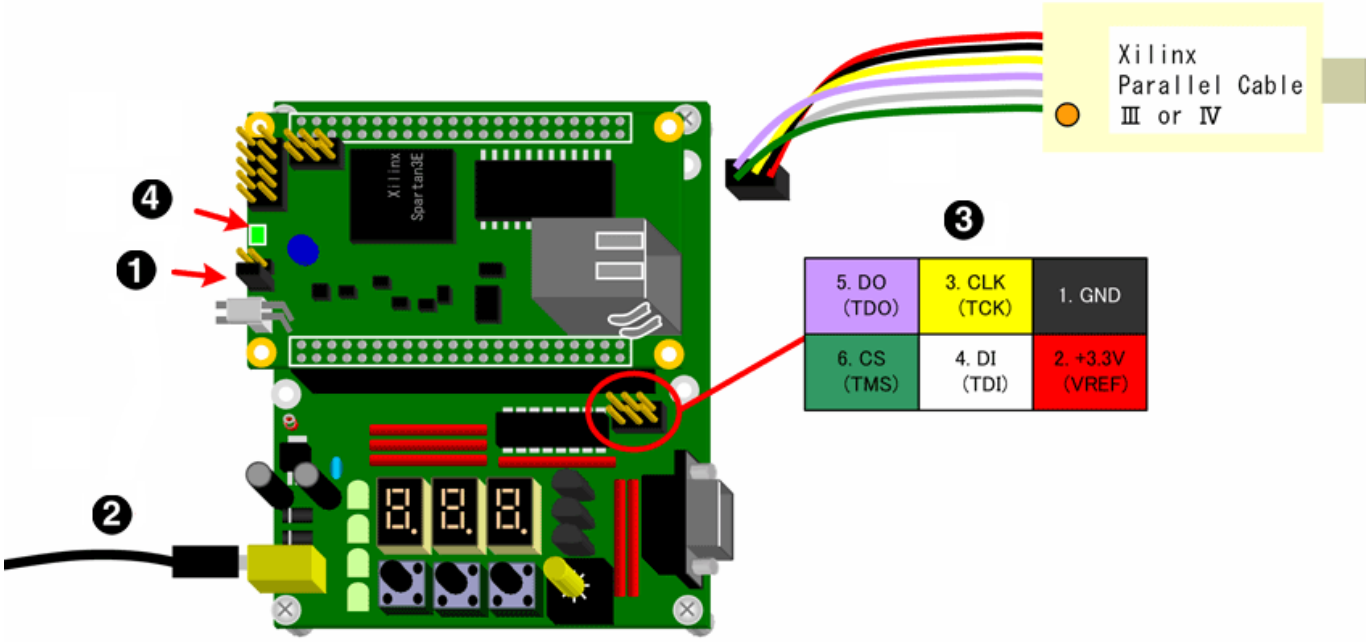


図 6.29. SPI Writer 書き込み準備

❶　JP2 をショート

❷　LED/SW CON6 に AC アダプタ 5V

❸　LED/SW CON4(1 - 6 番ピン)

❹　パワー ON LED(緑)

6.2.3.4. SPI Writer 立ち上げから書き込み

SPI Writer を立ち上げ、[...]をクリックして下さい。ファイル選択画面が立ち上がります。

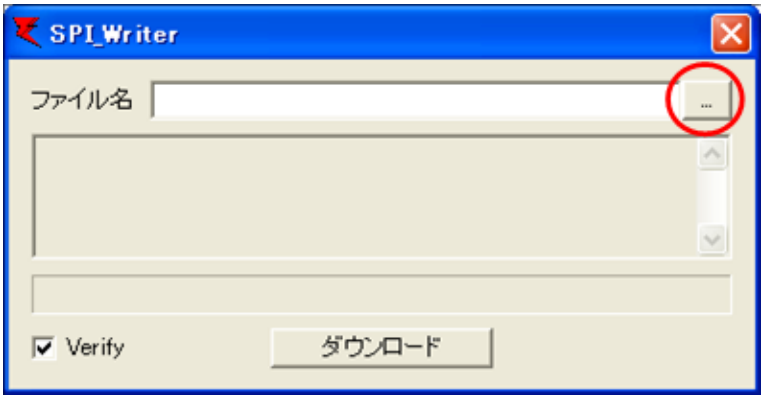


図 6.30. SPI_Writer

書き込む bit ファイルを選択し、[開く]をクリックしてください。SPI Writer で書き込めるファイルはbit ファイルです。

SUZAKU のデフォルトの bit ファイルは付属 CD-ROM の`"\suzaku\fpga\x.x\sz***\sz***-yyyymmdd.zip"`を展開したフォルダの中の`"default_bit_file"`に収録されています。また、スロットマシンの bit ファイル(スターターキット出荷時の bit ファイル）は付属 CD-ROM の`"\suzaku-starter-kit\fpga\x.x\sz***\sz***-add_slot-yyyymmdd.zip"`を展開したフォルダの中の

`"default_bit_file"`に収録されています。

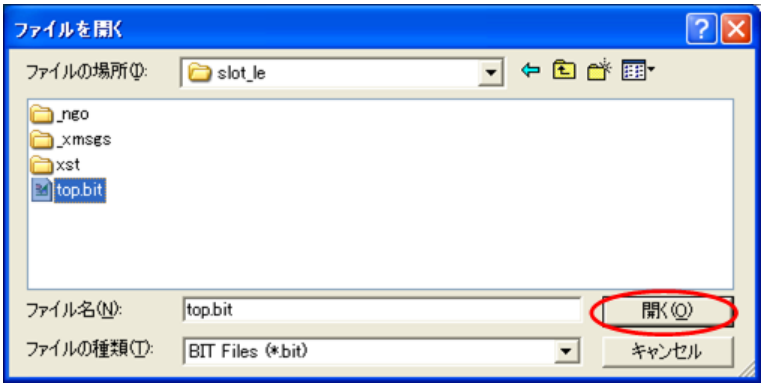


図 6.31. bit ファイル選択

SPI Writer は書き込みたい bit ファイルをドラッグ＆ドロップで選択することもできます。

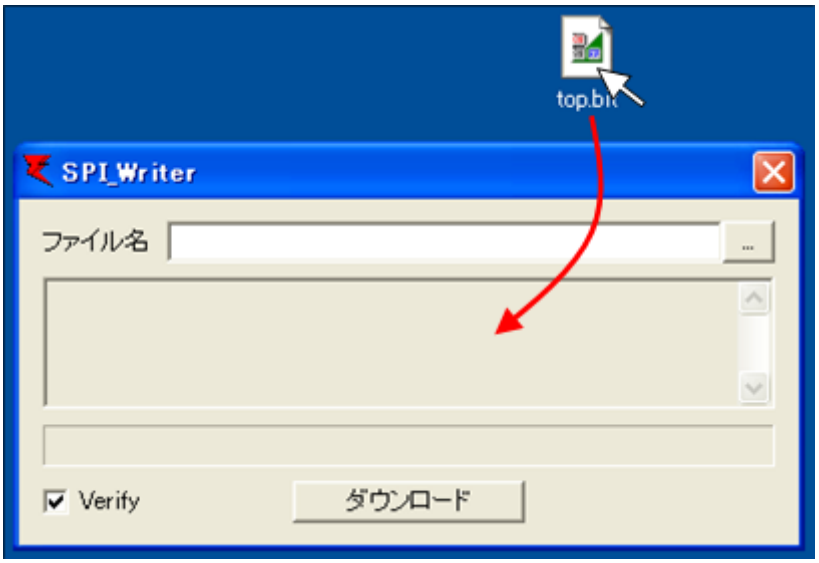


図 6.32. ドラッグ＆ドロップ

これで書き込み準備完了です。ダウンロードをクリックしてください。Verify を必要としない場合は、チェックボタンをはずしてください。

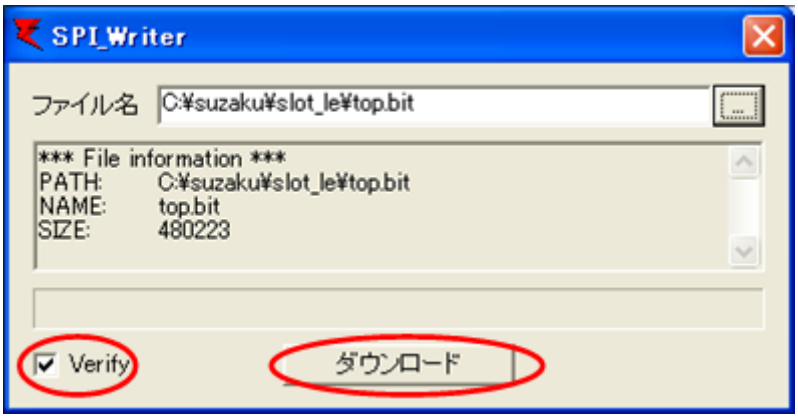


図 6.33. 書き込み準備完了

書き込みを開始してもいいか確認画面が表示されるので[OK]をクリックしてください。

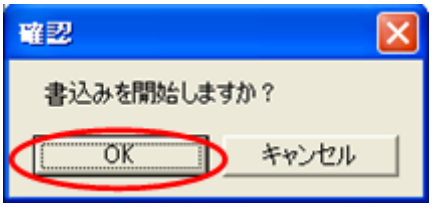


図 6.34. 書き込み確認画面

コンフィギュレーションデータが SPI フラッシュメモリに書き込まれます。ここで"Please check windrvr.sys"というエラーが発生した場合は後述の Tips を参照してください。

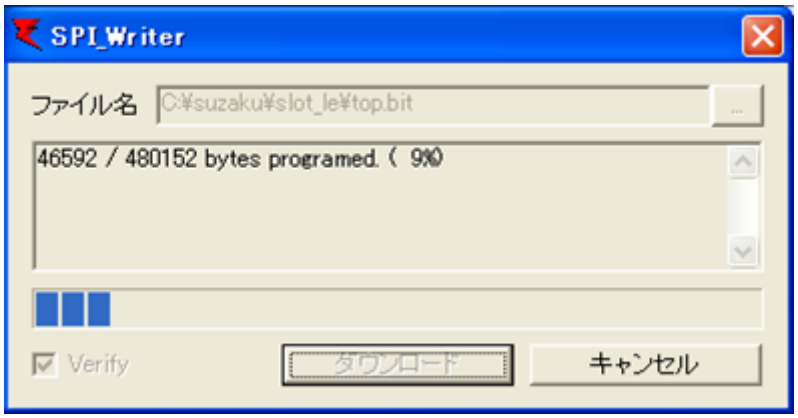


図 6.35. 書き込み中

以下の画面のように"Download has been completed!"と表示されたら書き込み終了です。

何らかの原因でエラーを起こした場合は、SUZAKU を動作させず、再び書き込みを行ってください。

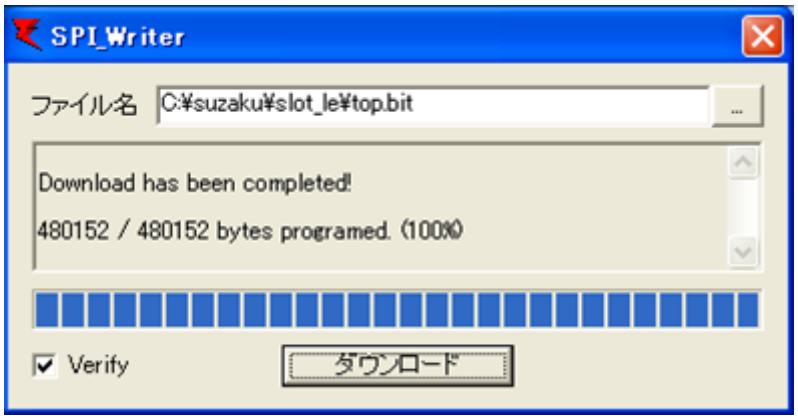


図 6.36. 書き込み終了

LED/SW CON6 から AC アダプタ 5V を抜いて電源を切り、JP2 のジャンパプラグと LED/SW CON4 のダウンロードケーブルをはずしてください。再び LED/SW CON6 に AC アダプタ 5V を接続し、電源を再投入してください。

6.2.3.5. SPI Writer で書き換える 手順まとめ

1. SUZAKU JP2 にジャンパプラグをさしてショートさせる
2. LED/SW CON4 にダウンロードケーブルを接続する
3. LED/SW CON6 に AC アダプタ 5V を接続し、電源投入
4. SUZAKU D3 のパワー ON LED(緑)が点灯していることを確認
5. SPI_Writer を立ち上げ、SPI フラッシュメモリにコンフィギュレーションデータを書き込む
6. LED/SW CON6 の AC アダプタ 5V をはずし、電源を切る
7. LED/SW CON4 のダウンロードケーブルをはずす

8. SUZAKU JP2 のジャンパプラグをはずす
9. LED/SW CON6 に AC アダプタ 5V を接続し、電源再投入
10. 動作確認

※電源を切っても、コンフィギュレーション内容は失われません。

SPI Writer とは

SPI Writer は SPI フラッシュメモリの先頭から 1MByte まで消去し、コンフィギュレーションデータを書き込む SUZAKU の SPI フラッシュメモリ専用の書き込みツールです。

SUZAKU は SPI フラッシュメモリにソフトウェアのデータやその他データを保存しており、これらのデータを壊さないために専用ツールで書き込みます。

SPI フラッシュメモリの書き込みツールとしては iMPACT の DirectSPI もあります。ただし、DirectSPI は SPI フラッシュメモリのデータを全消去して、コンフィギュレーションデータを書き込むツールであるため、SUZAKU の SPI フラッシュメモリ書き込み用として使うには注意が必要となります。

SPI Writer ERROR

SPI Writer で書き込む際、以下のエラーが出ることがあります。この場合ドライバのインストールが必要となります。

図 6.37. エラー表示

SPI Writer のフォルダの中に `wdreg.exe`、`difxapi.dll`、`wd811.cat`、`windrvr6.inf`、`windrvr6.sys` の 5 つのファイルがあることを確認してください。

コマンドプロンプトを立ち上げ、SPI Writer のフォルダに移動し、Administrator 権限ユーザで以下のコマンドを実行してください。

```
> wdreg -inf windrvr6.inf install
```

以下のようなログが表示されます。これでドライバがインストールされ、エラーが出なくなります。

```
> Installing a signed driver package
LOG Event: 1, ENTER:  DriverPackageInstallA
LOG Event: 1, ENTER:  DriverPackageInstallW
LOG Event: 1, Looking for Model Section [DeviceList]...
LOG Event: 1, windrvr6.inf: checking signature with catalog
'C:\spi_writer-20070119\wd811.cat' ...
LOG Event: 1, Driver package 'windrvr6.inf' is Authenticode
signed.
LOG Event: 1, Copied 'windrvr6.inf' to driver store...
LOG Event: 1, Copied 'wd811.cat' to driver store...
LOG Event: 1, Commiting queue...
LOG Event: 1, Copied file: 'C:\spi_writer-20070119\.
\windrvr6.sys' -> 'C:\WINDOWS\system32\DRVSTORE
\windrvr6_45AF516B2C99AB8FE1C0F3A3CBE523C199AE6F2B\.
\windrvr6.sys'.
LOG Event: 1, Installing INF file "C:\WINDOWS
\system32\DRVSTORE
\windrvr6_45AF516B2C99AB8FE1C0F3A3CBE523C199AE6F2B
\windrvr6.inf" of Type 6.
LOG Event: 1, Looking for Model Section [DeviceList]...
LOG Event: 1, Installing devices with Id "*WINDRVR6" using
INF "C:\WINDOWS\system32\DRVSTORE
\windrvr6_45AF516B2C99AB8FE1C0F3A3CBE523C199AE6F2B
\windrvrinstall: completed successfully
```

6.3. ブートローダ Hermit の書き換えかた

ブートローダ Hermit を書き換えるには BBoot で書き換える方法と、ダウンローダ Hermit で書き換える方法の 2 通りがあります。ここでは BBoot で書き換える方法を説明します。ダウンローダ Hermit の使い方については「6.4.1. ダウンローダ Hermit で書き換える」を参考にして下さい。

6.3.1. BBoot で書き換える

BBoot でブートローダ Hermit を書き換える方法を説明します。この際、Hermit のイメージデータはモトローラ S 形式のものを使用します。

6.3.1.1. 準備から書き込み

まず、JP1 にジャンパプラグをさし、ショートさせてください。JP1 をショートさせるとブートローダモードになります。

SUZAKU CON1 に方向に気をつけてシリアルケーブルを接続し、シリアル通信ソフトウェアを起動してください。(「5.2. シリアル通信ソフトウェア」参照)。LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。SUZAKU D3 のパワー ON LED(緑)が点灯しているか確認してください。

シリアル通信ソフトウェアの画面に以下のようなメッセージが表示されます。

SUZAKU スターターキット以外の場合は、電源投入直後に"z"キーを長押しすると、以下のようなメッセージが表示されます。何も立ち上がらない場合は、「6.2. FPGA の書き換えかた」を参照して FPGA を書き込んでください。

```
Please choose one of the following and hit enter.
a: active second stage bootloader (default)
s: download a s-record file
t: busy loop type slot-machine
```

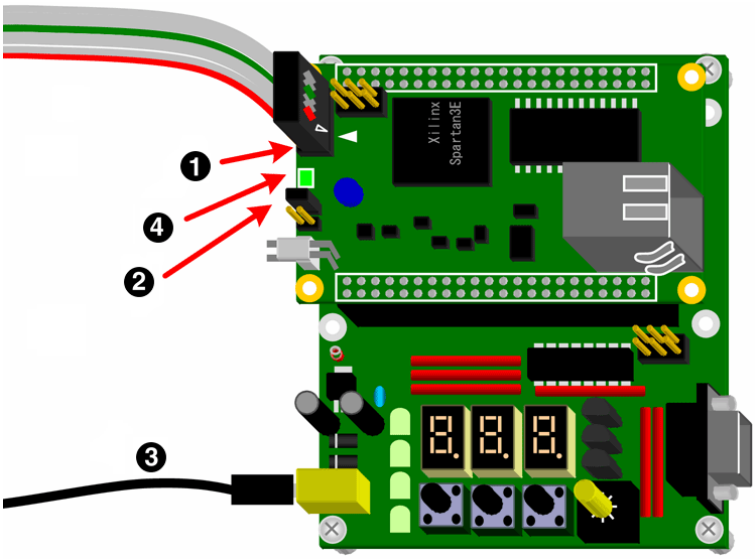


図 6.38. モトローラ S 形式書き換え準備

❶　三角印を合わせてシリアルケーブルを接続

❷ JP1 をショート

❸ LED/SW CON6 に AC アダプタ 5V

❹ パワー ON(緑)

"s"キーを押してください。モトローラ S 形式ダウンロードモードになります。以下のようなメッセージが表示されます。

```
Start sending S-Record!!
```

書き込むモトローラ S 形式のファイルを選択してください。書き込むファイルは **srec ファイル**です。

ブートローダ Hermit のファイルは付属 CD-ROM の`"\suzaku\bootloader\s-record"`に収録されています。

- loader-suzaku-microblaze-vx.x.x-4M.srec ： SZ010 用
- loader-suzaku-microblaze-vx.x.x-8M.srec ： SZ030、SZ130 用
- loader-suzaku-powerpc-vx.x.x-8M.srec ： SZ310、SZ410 用

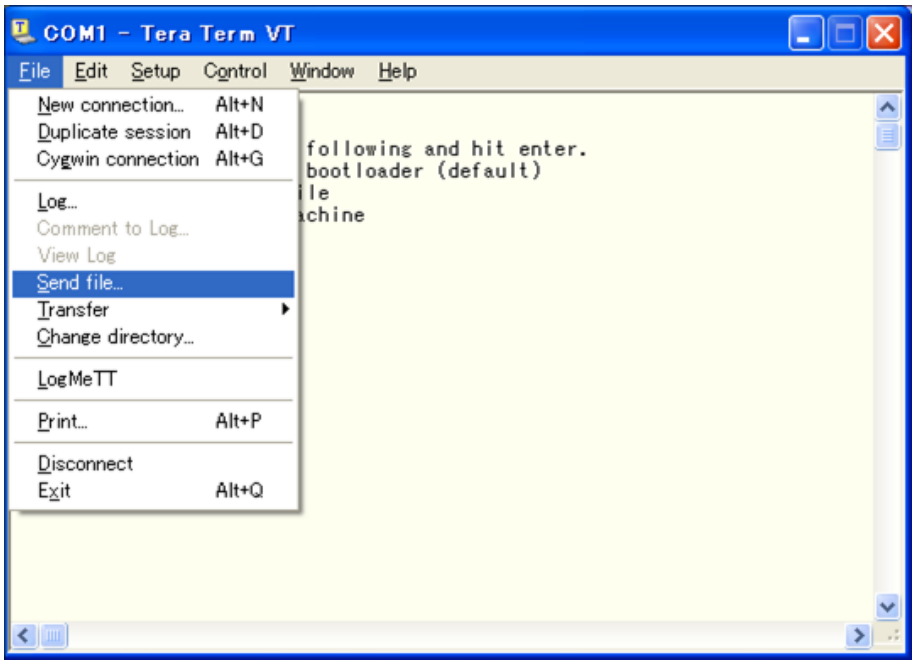


図 6.39. srec ファイルを送る

書き込みが始まると以下のような画面になります。

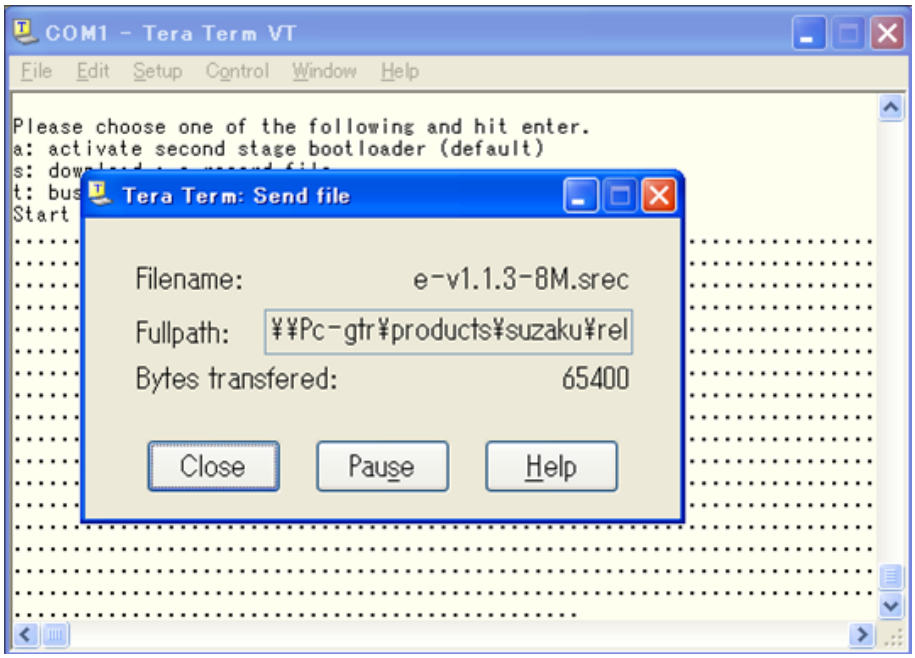


図 6.40. srec ファイル書き込み中

以下のように表示されたら書き込み完了です。電源を再投入してください。今書き込んだブートローダ Hermit が起動します。

```
Erasing SPI...
Programming SPI...
done.
Original checksum: 00XXXXXX
 Program checksum: 00XXXXXX
Reboot.

Please choose one of the following and hit enter.
a: active second stage bootloader (default)
s: download a s-record file
t: busy loop type slot-machine
```

6.3.1.2. BBoot で書き換える 手順まとめ

1. SUZAKU JP1 にジャンパプラグをさしてショートさせる
2. LED/SW CON1 にシリアルケーブルを接続する
3. シリアル通信ソフトウェアを立ち上げる
4. LED/SW CON6 に AC アダプタ 5V を接続し、電源投入
5. SUZAKU D3 のパワー ON LED(緑)が点灯していることを確認
6. シリアル通信ソフトウェアの画面を確認し、"s"と入力し、モトローラ S 形式ダウンロードモードにする
7. srec ファイルを選択し、書き込む

8. 動作確認

6.4. Linux の書き換えかた

Linux のイメージを書き換えるにはダウンローダ Hermit で書き換える方法、NetFlash で書き換える方法の 2 通りがあります。ここではダウンローダ Hermit で書き換える方法について説明します。NetFlash の使い方については「SUZAKU ソフトウェアマニュアル」をご参照ください。

6.4.1. ダウンローダ Hermit で書き換える

ダウンローダ Hermit で Linux のイメージを書き換える方法を説明します。

6.4.1.1. 書き込み準備

まず、JP1 にジャンパプラグをさし、ショートさせてください。JP1 をショートさせるとブートローダモードになります。

SUZAKU CON1 に方向に気をつけてシリアルケーブルを接続し、シリアル通信ソフトウェアを起動してください。(「5.2. シリアル通信ソフトウェア」参照)。LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。SUZAKU D3 のパワー ON LED(緑)が点灯しているか確認してください。

シリアル通信ソフトウェアの画面に以下のようなメッセージが表示されます。

```
Please choose one of the following and hit enter.
a: active second stage bootloader (default)
s: download a s-record file
t: busy loop type slot-machine
```

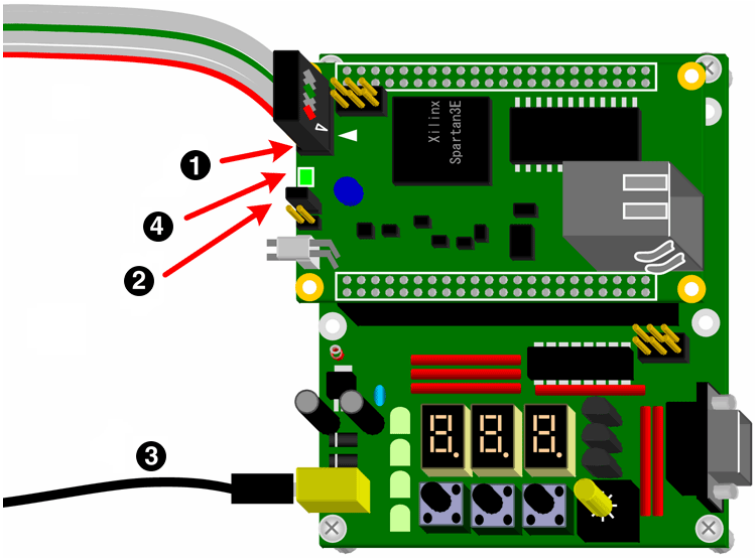


図 6.41. Linux 書き換え準備

❶　三角印を合わせてシリアルケーブルを接続

❷　JP1 をショート

❸　LED/SW CON6 に AC アダプタ 5V

❹　パワー ON(緑)

"a"もしくは"Enter"キーを押してください。ブートローダ Hermit が立ち上がります。以下のようなメッセージが表示されます。もし立ち上がらない場合は「6.3. ブートローダ Hermit の書き換えかた」を参照してブートローダ Hermit を書き込んでください。

```
Hermit-At v1.1.15 (suzaku/microblaze) compiled at 19:28:48, Feb 16 2008
hermit>
```

ブートローダ Hermit が立ち上がったのを確認したら、シリアルポートをシリアル通信ソフトから切断します。

[File]→[Disconnect]を選択してください。

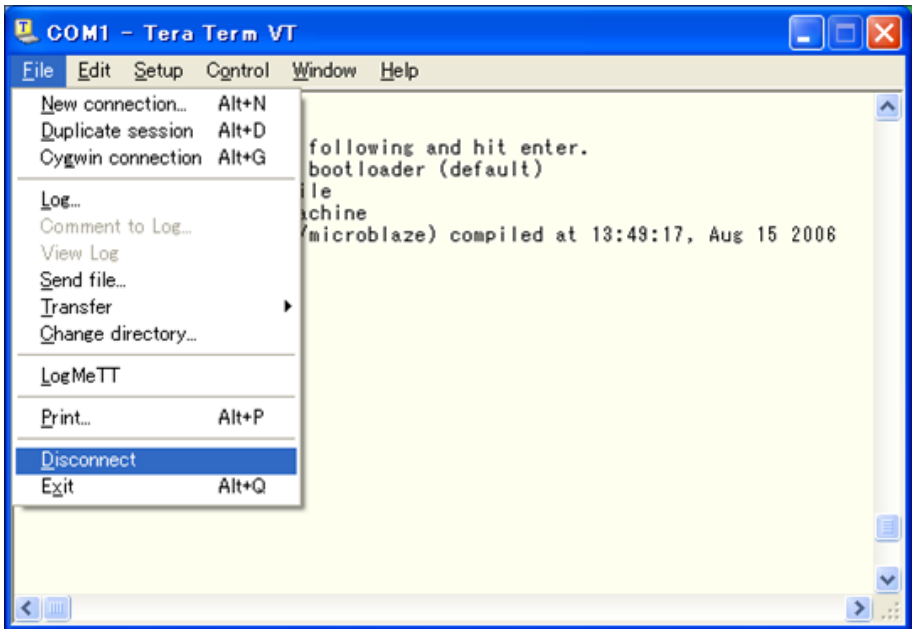


図 6.42. シリアルポートを切断

6.4.1.2. ダウンローダ Hermit 立ち上げから書き込み

ダウンローダ Hermit を起動してください。[Download]ボタンをクリックすると Download 画面が表示されます。[Serial Port] に、SUZAKU と接続しているシリアルポートを設定し、[Image]に書き込むイメージファイルを指定してください。ダウンローダ Hermit では **bin ファイル**を書き込みます。

SUZAKU のデフォルトの bin ファイルは付属 CD-ROM の`"\suzaku\image\image-sz***.bin"`に収録されています。また、スロットマシンの bin ファイル（スターターキット出荷時の bin ファイル）は付属 CD-ROM の`"\suzaku-starter-kit\image\image-sz***-sil.bin"`に収録されています。

[Region]には、書き込むリージョンまたは、アドレスを指定します。Linux はイメージリージョンに書き込みます。[image]を選択してください。

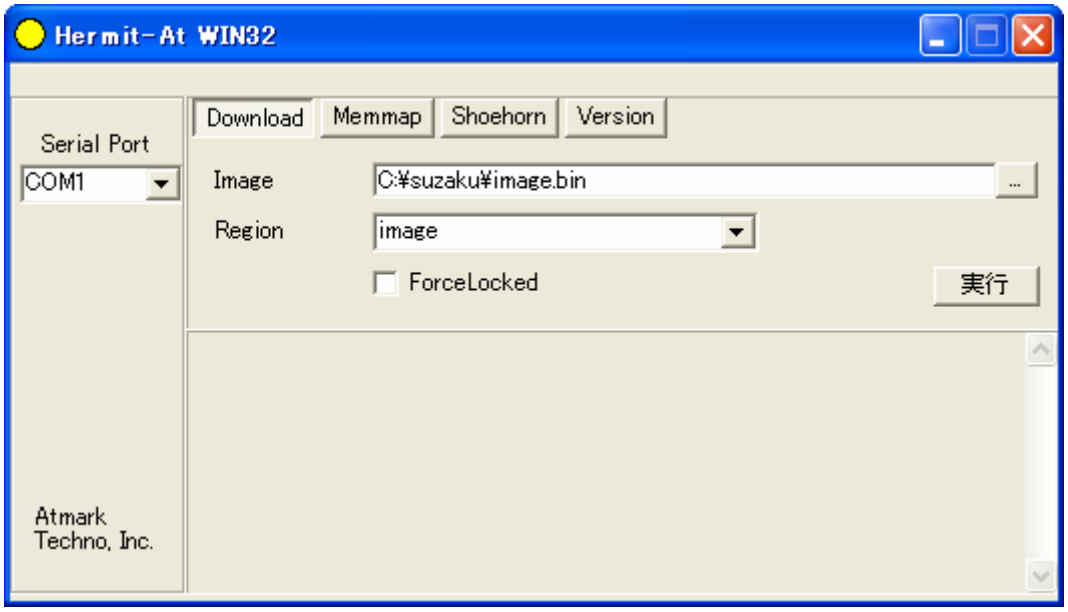


図 6.43. Download 画面

[実行]をクリックしてください。フラッシュメモリへの書き込みが始まります。書き込み中は、進捗状況が表示されます。

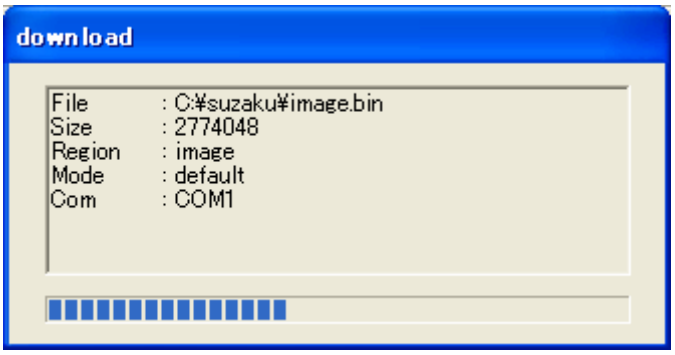


図 6.44. 書き込み進捗ダイアログ

書き込みが終了すると、書き込み終了画面が表示されます。"Download COMPLETE"と表示されたら書き込み成功です。

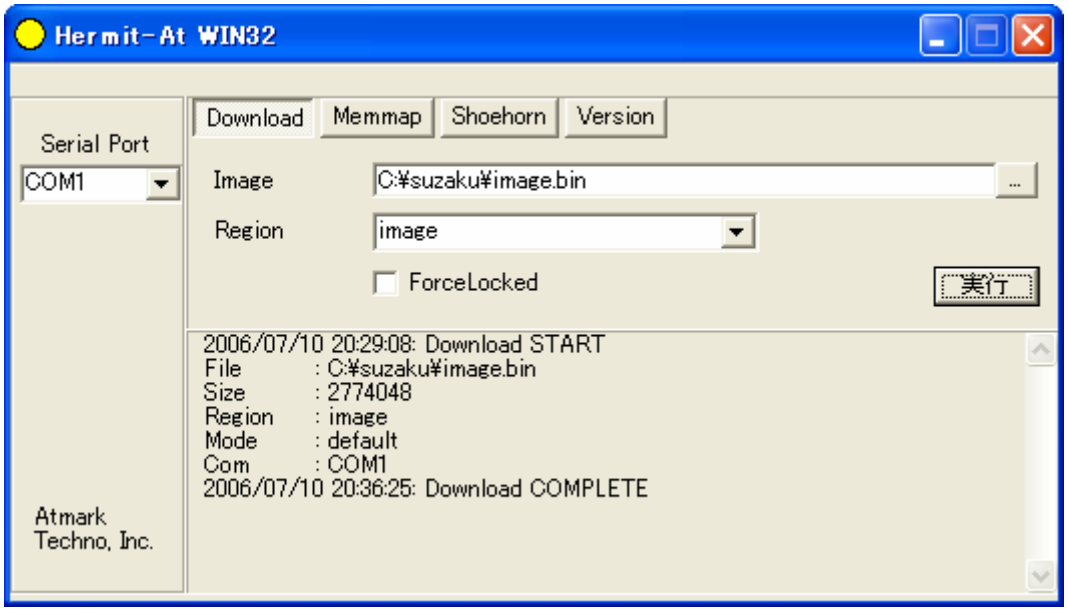


図 6.45. 書き込み終了

6.4.1.3. FPGA とブートローダ Hermit を書き換える

ダウンローダ Hermit では FPGA およびブートローダ Hermit も書き換えることが出来ます。これらの bin ファイルは Linux の bin ファイルと同じフォルダに収録されています。

- FPGA

  fpga-sz***-x.x- yyyymmdd.bin ： デフォルト

- fpga-sz***-sil-x.x- yyyymmdd.bin ： スロットマシン

- ブートローダ Hermit

  loader-suzaku-microblaze-vx.x.x.bin ： SZ010、SZ030、SZ130 用

- loader-suzaku-powerpc-vx.x.x.bin ： SZ310、SZ410 用

これらを書き換える場合は[Region]に[fpga]、[bootloader]をそれぞれ選択してください。また、書き込む際は、[ForceLocked]をチェックする必要があるのでご注意ください。チェックしないと、警告が表示され書き込みが始まりません。

6.4.1.4. ダウンローダ Hermit で書き換える 手順まとめ

1. SUZAKU JP1 にジャンパプラグをさしてショートさせる
2. LED/SW CON1 にシリアルケーブルを接続する
3. シリアル通信ソフトウェアを立ち上げる
4. LED/SW CON6 に AC アダプタ 5V を接続し、電源投入
5. SUZAKU D3 のパワー ON LED(緑)が点灯していることを確認
6. シリアル通信ソフトウェアの画面を確認し、"a"もしくは"Enter"キーを入力
7. ブートローダ Hermit モードになったのを確認し、シリアルポートを切断
8. ダウンローダ Hermit を起動し、イメージファイルを書き込む
9. 動作確認

FPGA の bin ファイルの作り方

FPGA の bit ファイルはヘッダ部にデバイス情報や日付情報が入っただけのバイナリデータファイルです。bin ファイルとの違いはヘッダ部だけなので、この部分を削除すれば bin ファイルを作ることが出来ます。バイナリエディタで bit ファイルを開き、ヘッダ部のデータ FFFFFFFFh の手前までを削除し、拡張子を bin に変更して保存してください。ダウンローダ Hermit で FPGA リージョンに書き込める FPGA の bin ファイルの出来上がりです。

パラレルポートがなくても・・・

お使いの PC にパラレルポートがないという方もたくさんおられると思いますが、実はパラレルポートがなくても、

- シリアルポート
- USB ポート
- Platform Cable USB

があれば、問題なく SUZAKU を書き換えることができます。BBoot とブートローダ Hermit さえ動けば、シリアルポートから FPGA リージョン、ブートローダリージョン、イメージリージョンの全てを書き換えることが出来るからです。BBoot、ブートローダ Hermit が壊れてしまった時のみ、シリアルポートからでは直せないので、USB ポートと Platform Cable USB（もしくはパラレルポートと Xilinx Parallel Cable Ⅲ/Ⅳ）が必要となります。BBoot とブートローダ Hermit は以下の手順で修復できます。

1. JP2 をショートし、Platform Cable USB と iMPACT で FPGA に BBoot(bit ファイル)をコンフィギュレーション
2. BBoot が起動するので、"s"キーを押し、モトローラ S 形式でブートローダ Hermit(srec ファイル)をフラッシュメモリのブートローダリージョンに書き込む(スターターキット以外の場合は、"s"キーの前に"z"キー長押しが必要)
3. 再度 Platform Cable USB と iMPACT で FPGA に BBoot(bit ファイル)をコンフィギュレーション
4. BBoot が起動するので、"a"キーを押し、ダウンローダ Hermit でフラッシュメモリの FPGA リージョンに BBoot(bin ファイル)を書き込む

7.ISE の使い方

FPGA 側から SUZAKU の開発をするためには、ISE(Integrated Software Environment)の使い方を知ることが必要不可欠です。ISE は Xilinx が提供する FPGA の統合型設計環境です。GUI 統合ツール Project Navigator で、FPGA に必要な論理合成、配置配線、bit ファイルの書き込みのツールなど、トータルな開発環境を提供しています。

ここでは LED/SW ボードの単色 LED(D1)を点灯させると共に ISE の使い方を説明します。ISE の使い方の詳細は ISE のヘルプ、マニュアル等を参照してください。ISE には日本語のヘルプ、マニュアル等も用意されています。

なお、本書では ISE において以下の手順で作業を行います。

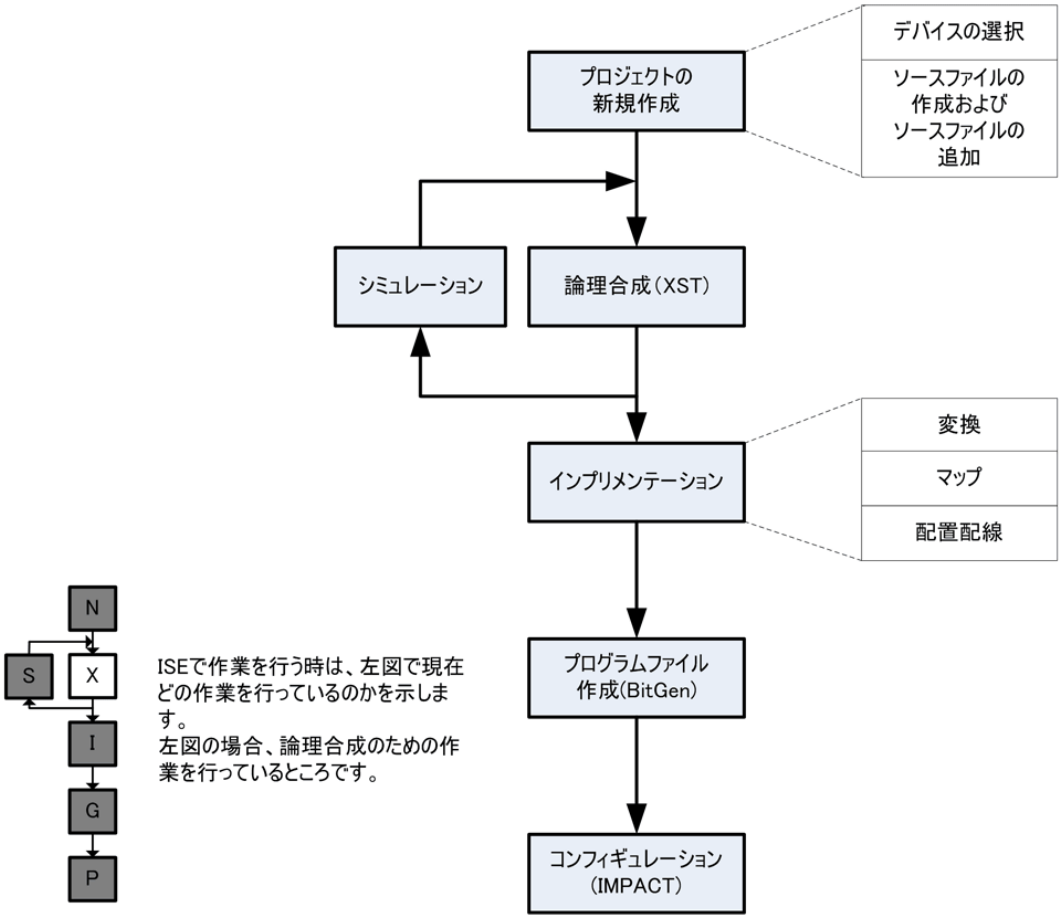


図 7.1. 本書での ISE 開発フロー

7.1. 単色 LED を点灯させる

ISE を使って、LED/SW ボードに実装されている単色 LED(D1)を点灯させてみます。

7.1.1. 単色 LED 周辺回路

単色 LED 周辺回路は下図のようになっています。それぞれ 180Ω の抵抗で 3.3V にプルアップされています。FPGA から"Low"を出力すると、単色 LED が点灯し、"High"を出力すると、単色 LED が消灯します。

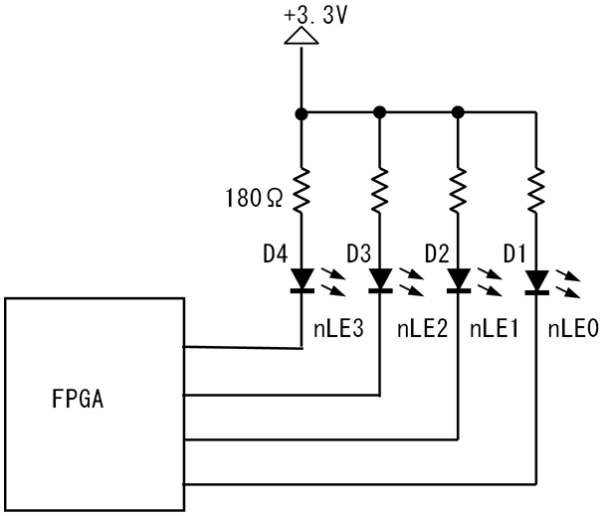


図 7.2. 単色 LED 周辺回路

FPGA の入出力について

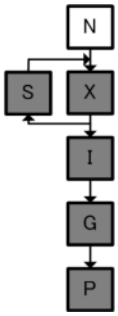
SUZAKU の FPGA の I/O ピンは CMOS ＋ 3.3V に設定されています。FPGA からは"Low"で 0.4V 以下、"High"で 2.9V 以上が出力されます。FPGA へは 0.8V 以下で"Low"、2.0V 以上で"High"が入力されます。ただし、ディジタル入力定格は-0.3V 縲鰒 3.6V なので、それを超えて入力しないでください。

表 7.1. FPGA 入力、出力

	Low(V)	High(V)
出力	OUT <= 0.4	2.9 <= OUT
入力	-0.3 < IN <= 0.8	2.0 <= IN <=3.6

7.2. プロジェクトの新規作成

7.2.1. プロジェクト作成



Project Navigator を起動してください。Project Navigator は、`"\ISE` のインストールフォルダ`\bin\nt\_impact.exe"`から起動できます。もしくは、[スタートメニュー]→[すべてのプログラム]→[Xilinx ISE Design Suite x.x]→[ISE]→[Project Navigator]から起動できます。
[File]→[New Project]をクリックしてください。

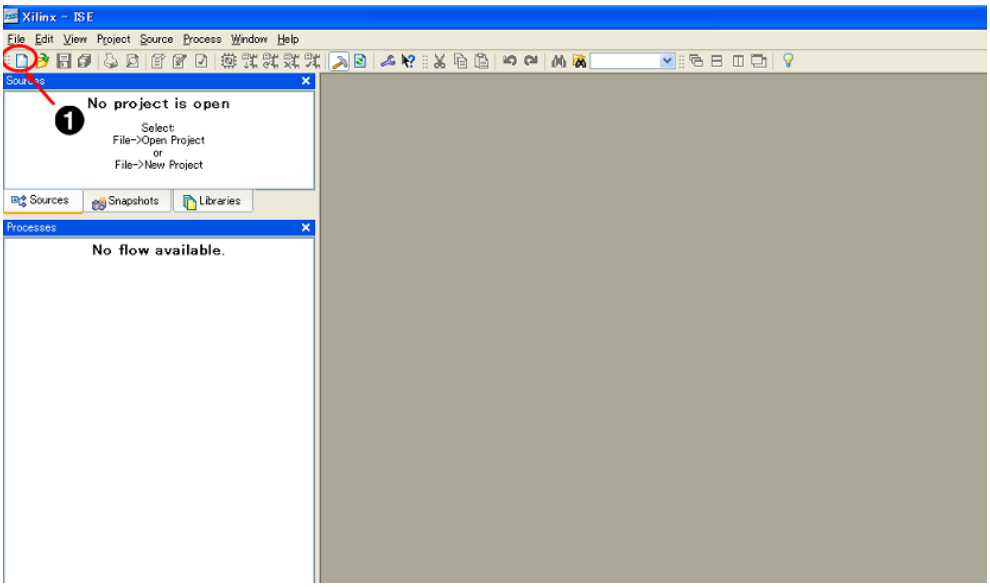


図 7.3. Project Navigator 起動

❶　[File]→[New Project]をクリック

New Project Wizard が表示されます。[Project Location]の[…]をクリックし、プロジェクトのディレクトリパスを指定します。ここでは C:\suzaku とします。[Project Name]に プロジェクト名を入力します。 slot_le と入力し、[Top-Level Source Type]が[HDL]となっていることを確認し、[Next]をクリックしてください。

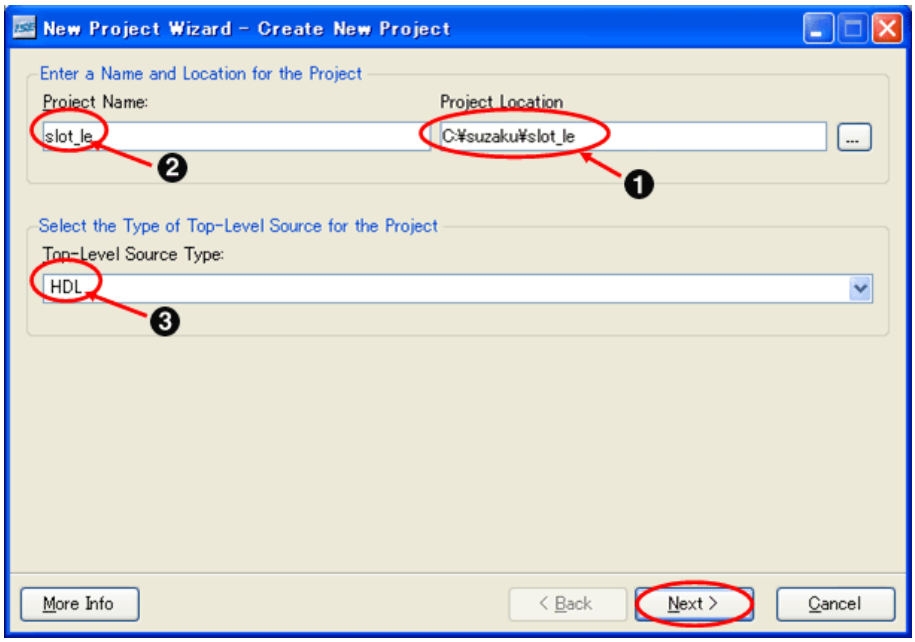


図 7.4. プロジェクトの新規作成

❶ ディレクトリパスを指定,`C:\suzaku` と入力

❷ slot_le と入力

❸ 確認

7.2.2. デバイスの選択

SUZAKU に実装されている FPGA デバイスを選択します。お使いの SUZAKU の型式の設定にし、[Next]をクリックしてください。

型式	SZ010	SZ030	SZ130	SZ310	SZ410
Product Category	All				
Family	Spartan3		Spartan3E	Virtex2P	Virtex4
Device	XC3S400	XC3S1000	XC3S1200E	XC2VP4	XC4VFX12
Package	FT256		FG320	FG256	SF363
Speed	-4			-5	-10
Synthesis Tool	XST(VHDL/Verilog)				
Simulator	ISE Simulator(VHDL/Verilog)				

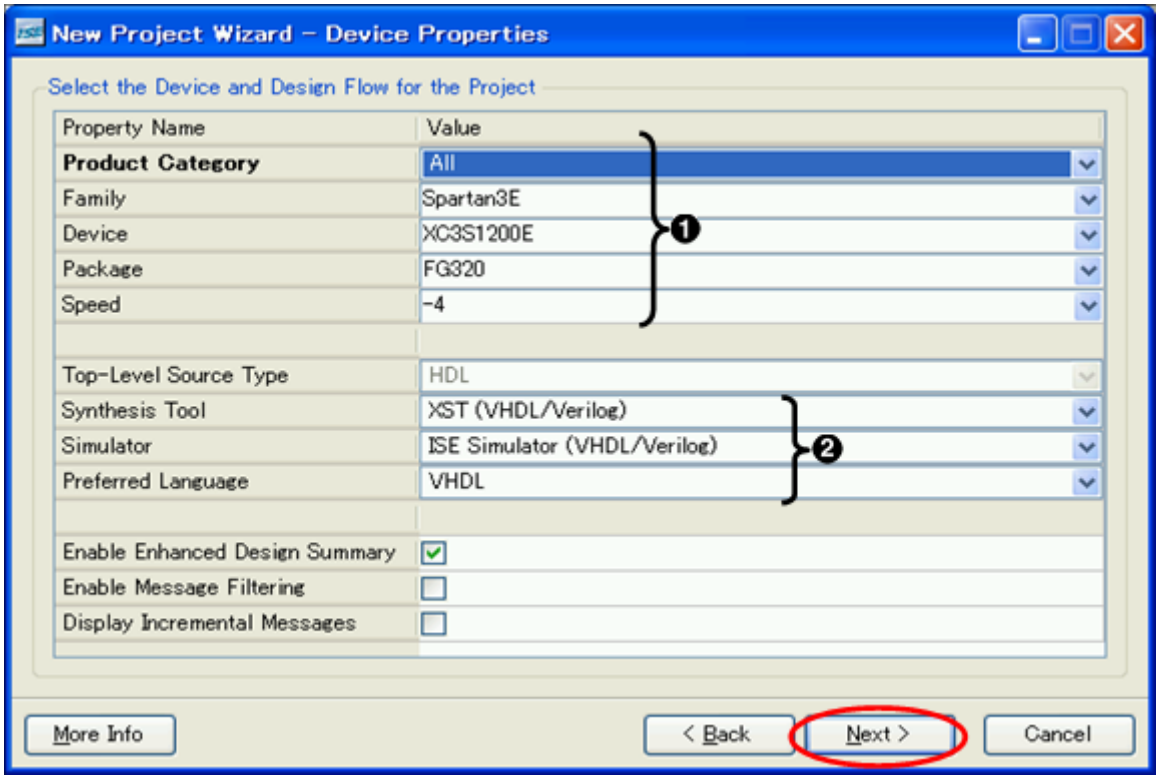


図 7.5. デバイスの選択(SZ130 の場合)

❶　デバイスの選択

❷　デザインフローの選択

7.2.3. ソースファイル作成

[New Source]をクリックしてください。

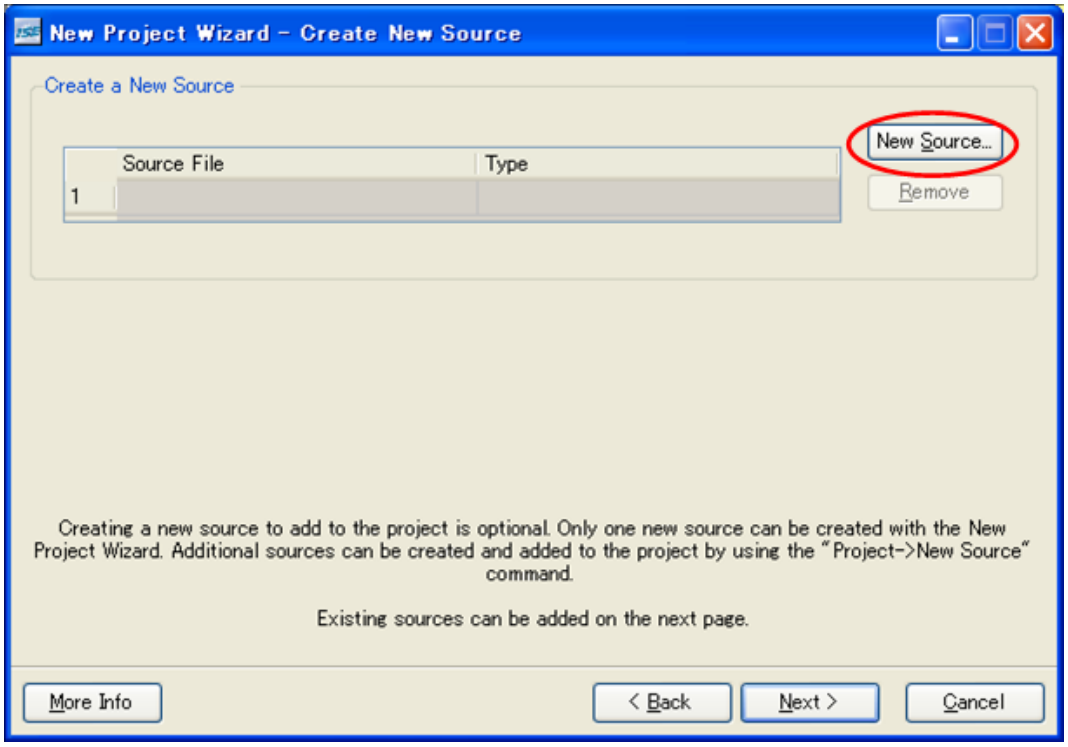


図 7.6. New Source 作成

[VHDL Module]を選択し、[File name]に top と入力し、[Next]をクリックしてください。VHDL ソースファイルが作成されます。

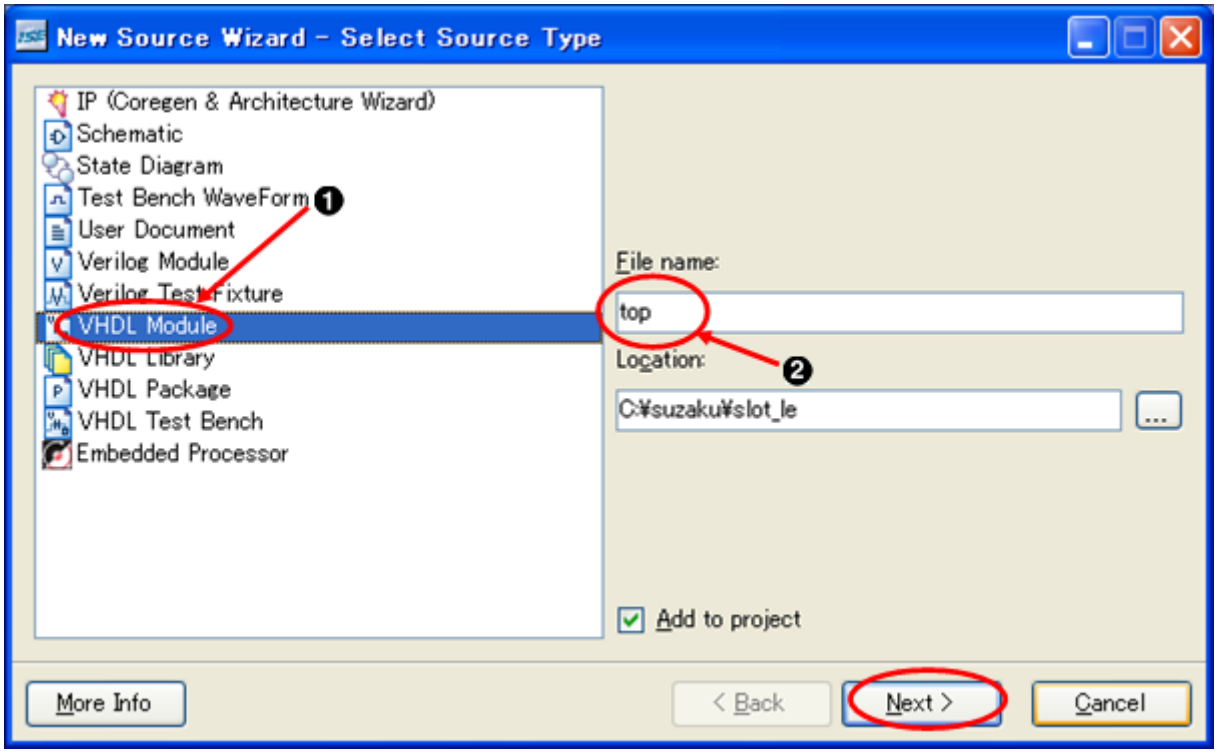


図 7.7. VHDL ソースファイル作成

❶　VHDL Module をクリック

❷　top と入力

[Architecture Name]を入力してください。何でも良いのですが、SUZAKU では IMP(implement の意味)としています。変更したら [Next]をクリックしてください。

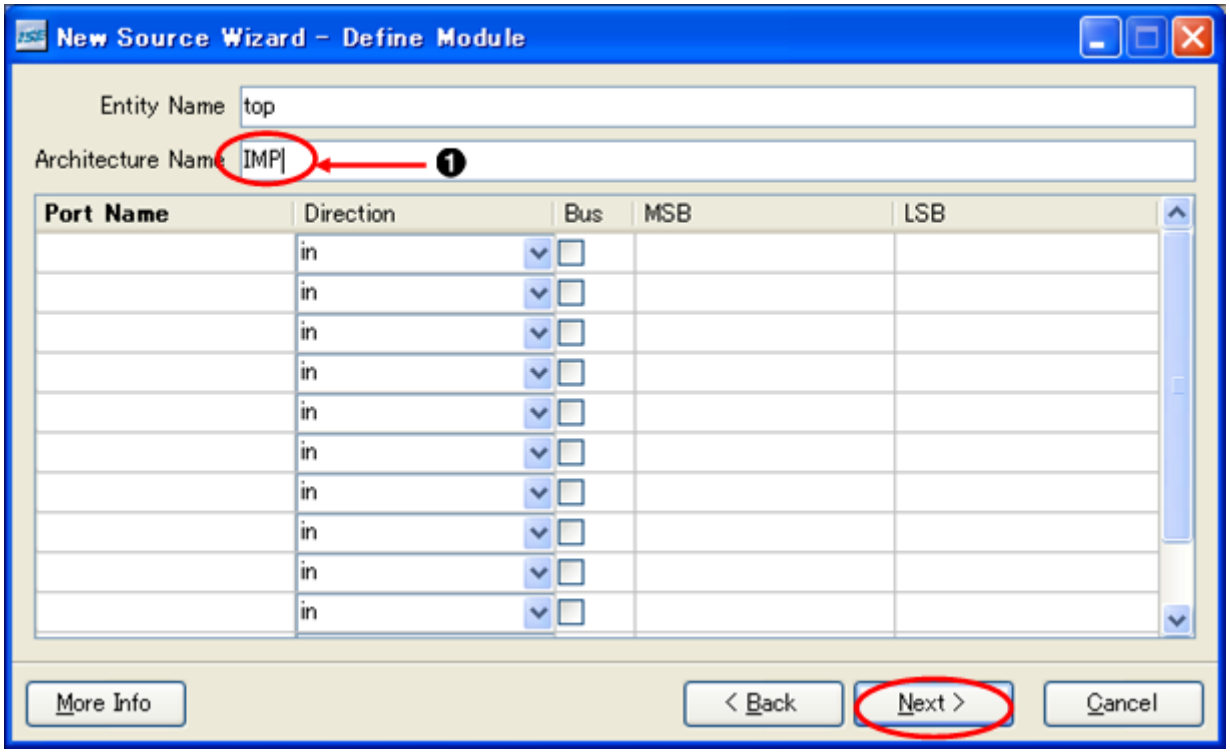


図 7.8. アーキテクチャ名定義

❶ IMP に変更

今作った VHDL ソースファイルの設定が表示されます。設定に間違いがないか確認をし、[Finish]をクリックしてください。

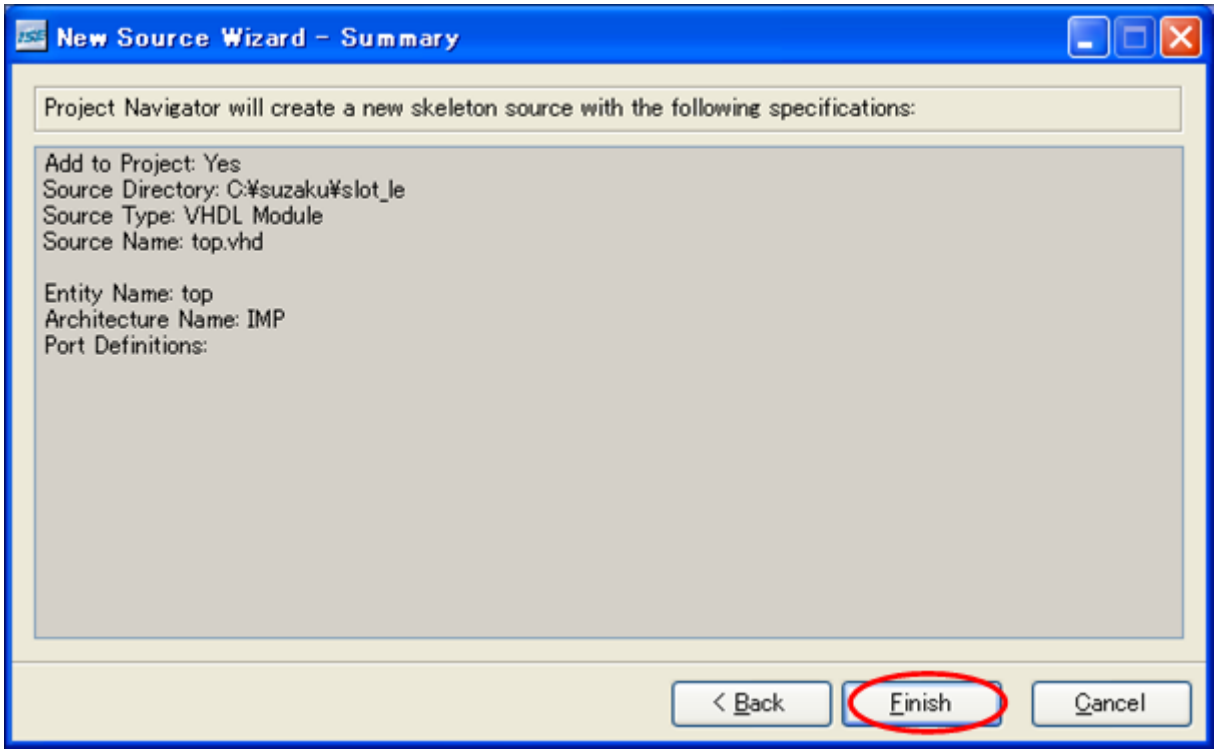


図 7.9. ソースファイル作成確認画面

以下の画面が出るまで[Next]をクリックして下さい。内容を確認し、[Finish]をクリックしてください。

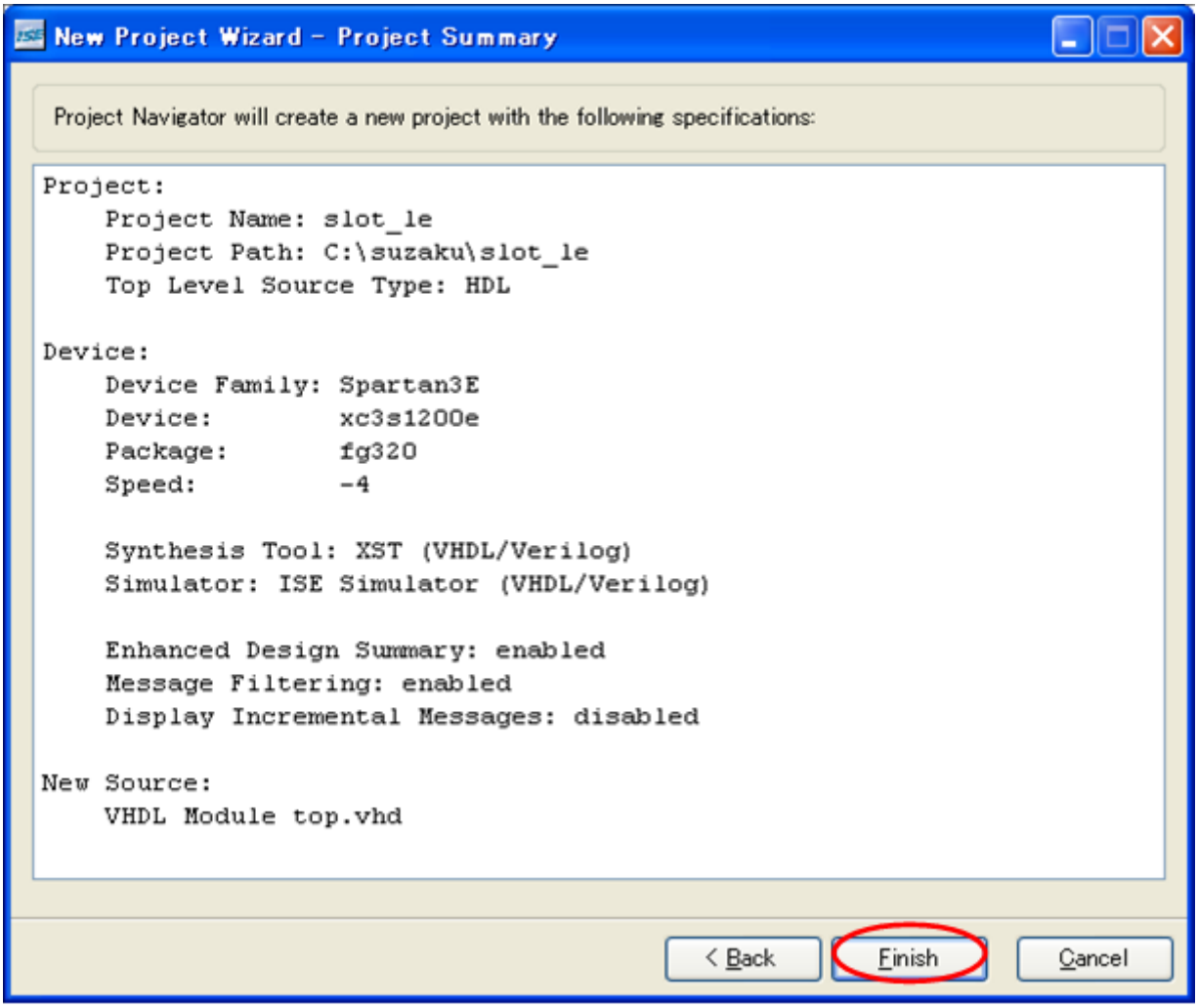


図 7.10. 最終確認画面(SZ130 の場合)

以上で新規プロジェクトおよび VHDL ソースファイルが出来上がります。

top-IMP(top.vhd)をダブルクリックしてください。`top.vhd` が開きます。

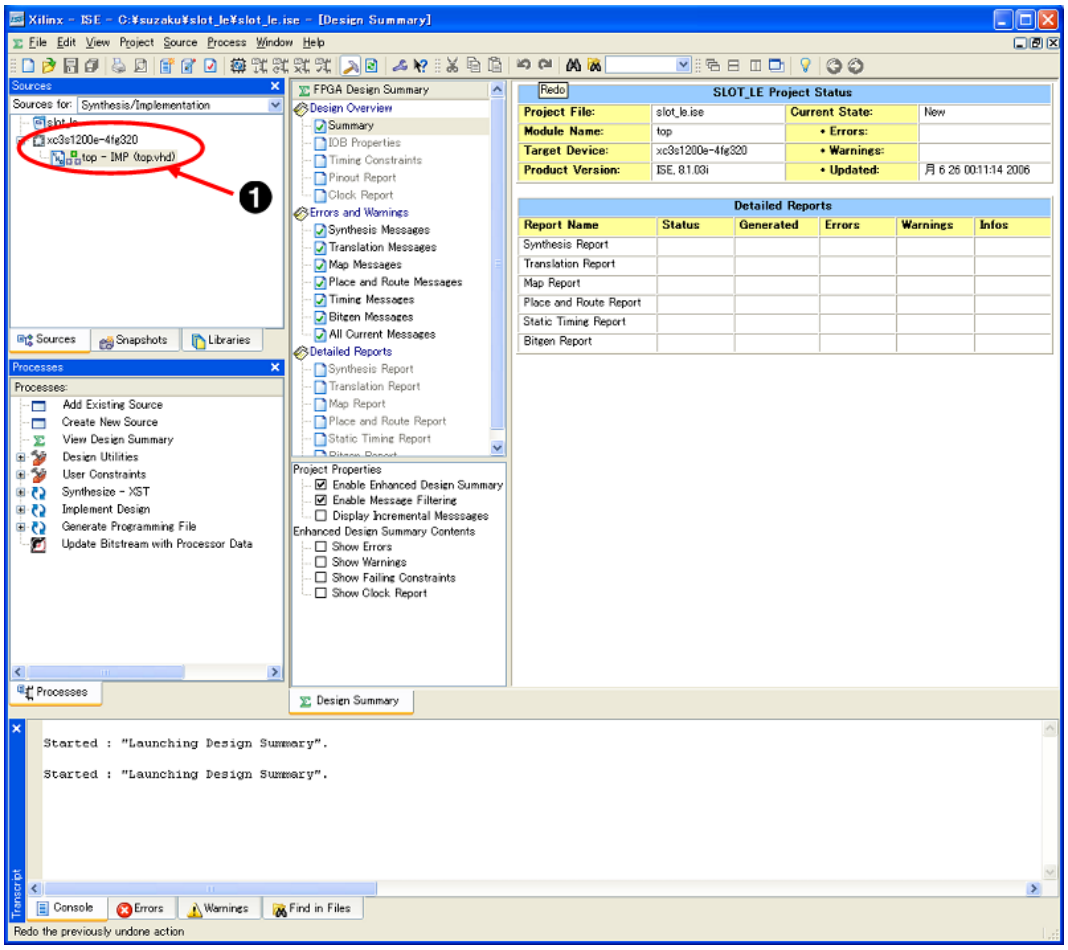
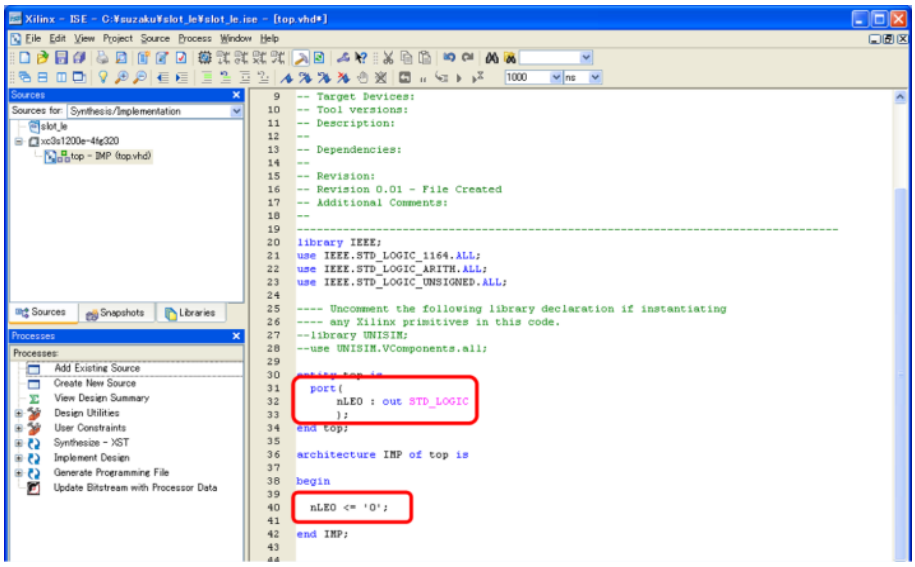


図 7.11. 新規プロジェクト、ソースファイルのテンプレート作成完了

❶　top(top.vhd)をダブルクリック

7.3. ソースファイル作成

テンプレートが自動生成されています。以下のように単色 LED への出力信号の定義と単色 LED を点灯させる文を追加してください。



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity top is
  port (
    nLE0 : out  STD_LOGIC   ❶
  );
end top;

architecture IMP of top is
begin
  nLE0 <= '0';              ❷
end IMP;
```

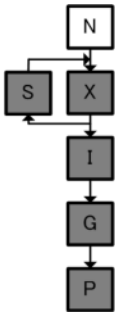
図 7.12. ソースコード入力

❶　出力信号の定義

❷　単色 LED を点灯させる

追加できたら、[File]→[Save]をクリックして保存してください。

7.4. 論理合成



トップモジュールtop - IMP(top.vhd)を選択し、Synthesize をダブルクリックしてください。Synthesize をダブルクリックすると文法チェックが始まります。ソースコードに間違いがなければ Synthesize の横にチェックマークもしくは警告マークが付きます。もしエラーマークになった場合はログをチェックし、ソースコードを見直してください。ソースコードを修正して保存するとマークが疑問符 に変わるので、再び Synthesize をダブルクリックし、エラーマークがなくなるまで繰り返してください。

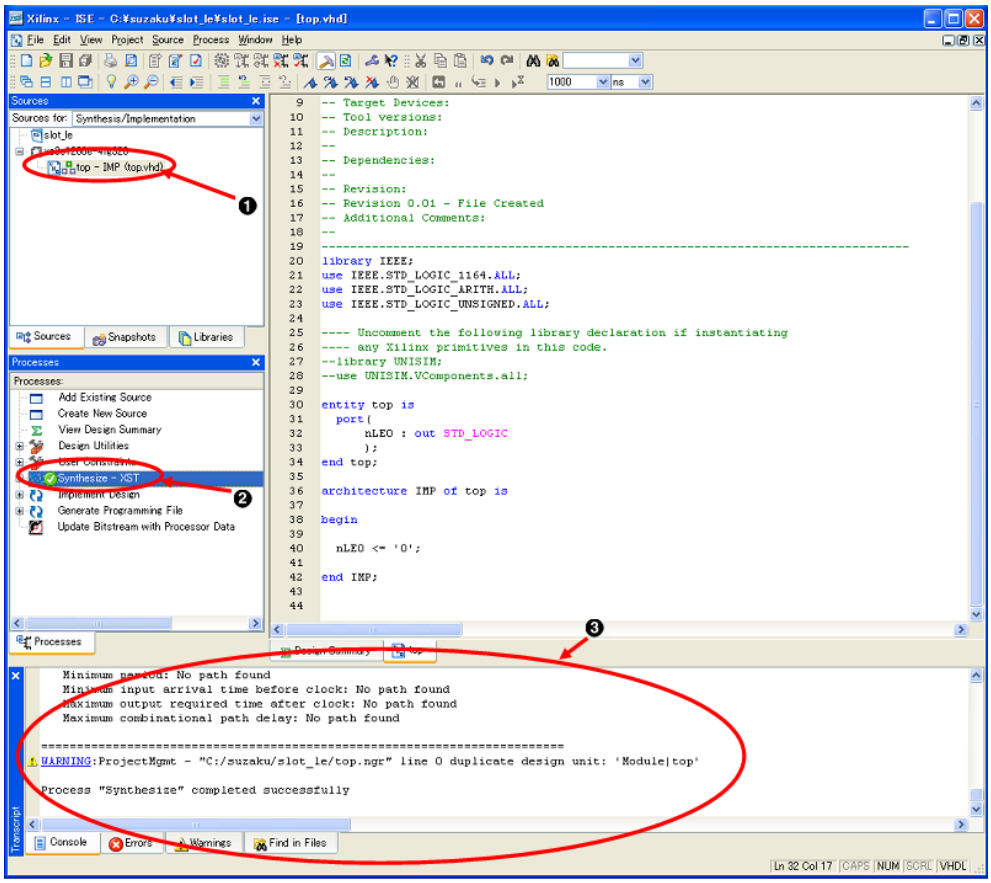


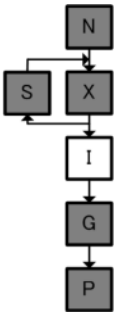
図 7.13. 文法チェック

❶ トップモジュールを選択

❷ Synthesize をダブルクリック

❸ ログが表示される

7.5. インプリメンテーション



User Constraints の横の ⊞をクリックして開き、

Floorplan IO - Pre-Synthesis をダブルクリックしてください。

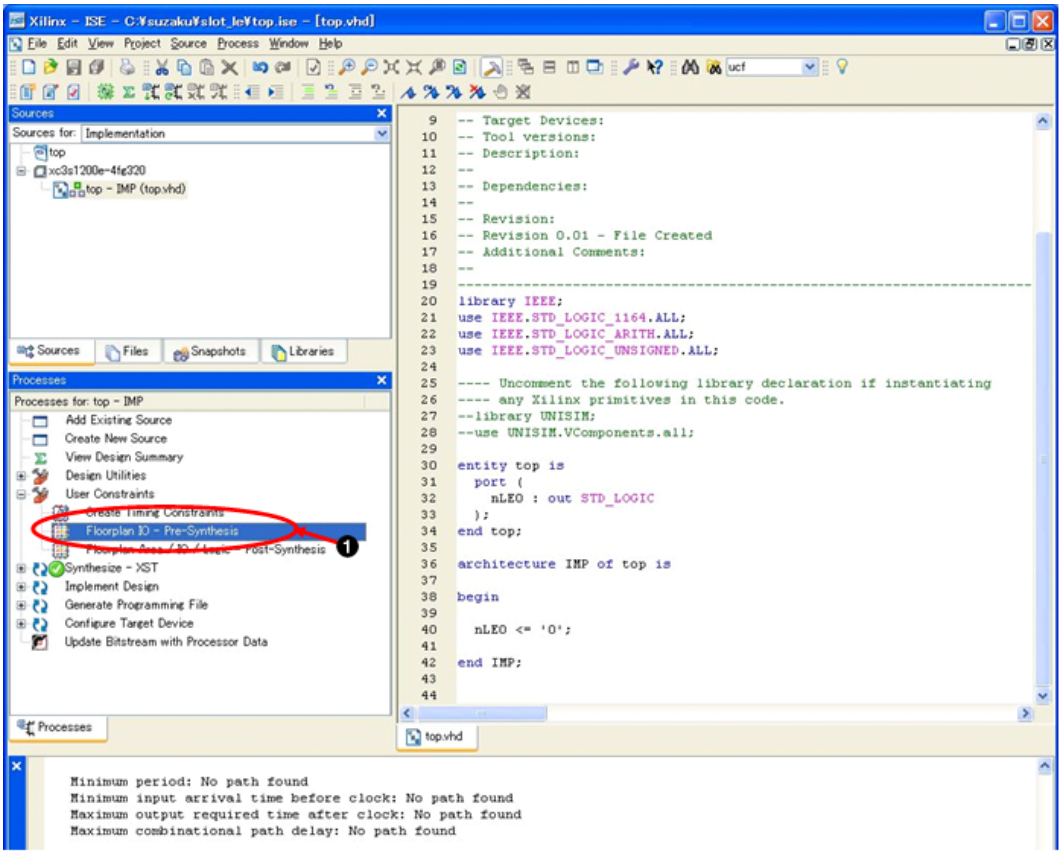


図 7.14. PACE を立ち上げる

❶　Floorplan IO - Pre-Synthesis をダブルクリック

ucf ファイルを追加してもいいかという質問をされるので[Yes]をクリックしてください。

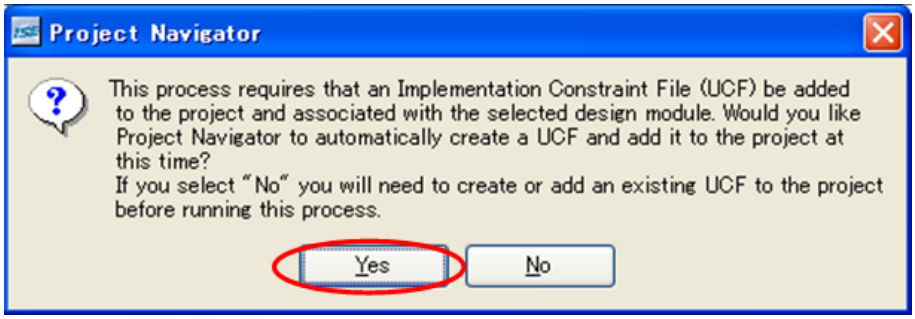


図 7.15. ucf ファイル作成確認

PACE というピンアサインを設定できるソフトが立ち上がります。「表 2.2. 機能用ピンアサイン(CON2)」を参照し、D1 の単色 LED とつながっている FPGA のピンを割り当てます。LOC にピンアサインを入力し、I/O Std.を LVCMOS33 としてください。

表 7.2. nLE0 ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nLE0	B12	E12	L16	G2

[File]→[Save]をクリックして保存し、PACE を閉じてください。

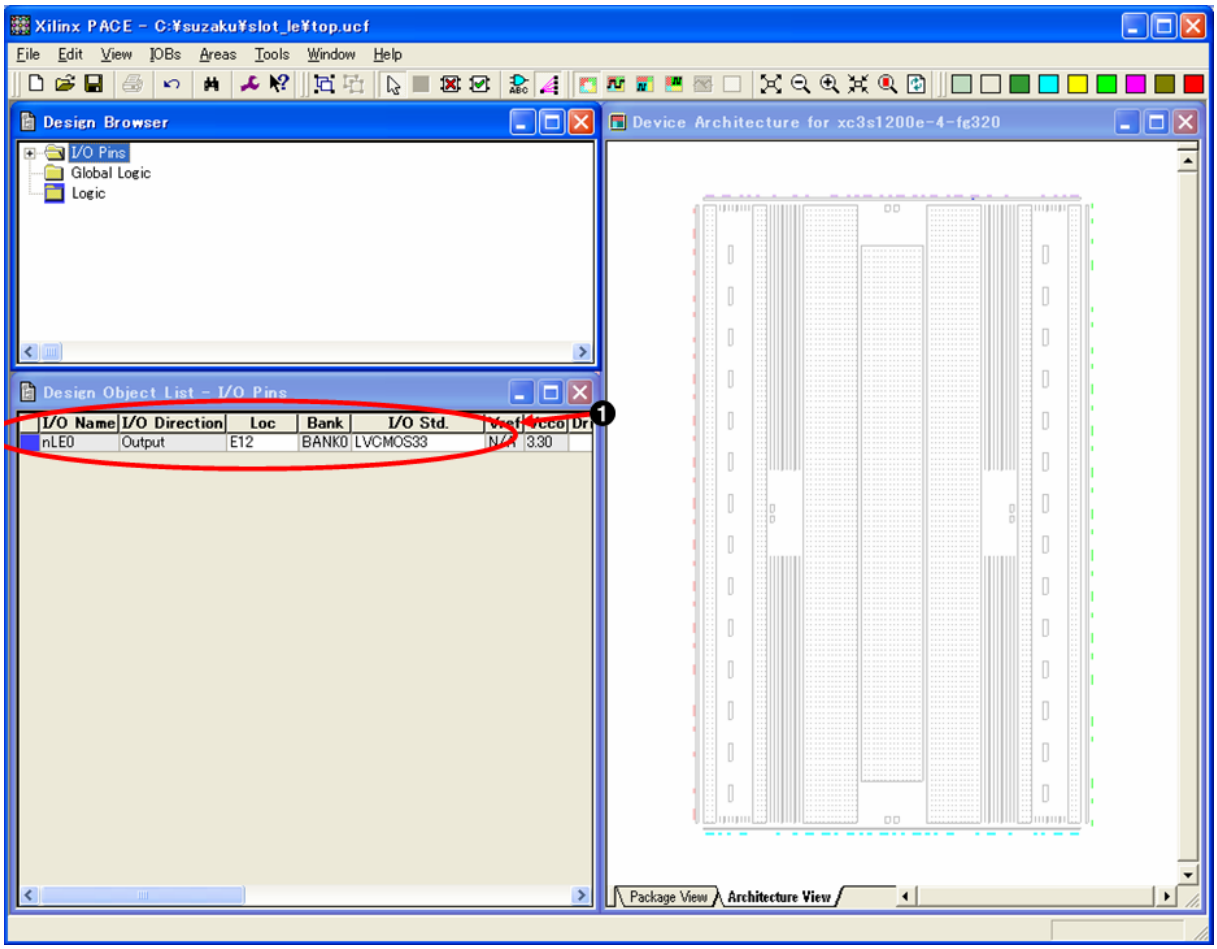
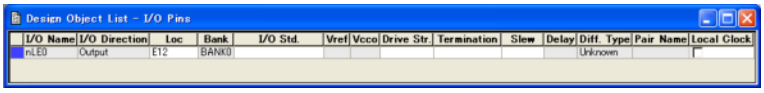


図 7.16. PACE によるピンアサイン(SZ130 の場合)

❶　Loc にピンアサインを入力し、I/O Std.を LVCMOS33 に設定

I/O ピンのカスタマイズ

最近の FPGA は I/O のカスタマイズも自由に行うことが出来ます。PACE の Design Object List -I/O Pins ウィンドウで、数 10kΩ の Pull Up、Pull Down をつけたり、電流制限をつけたり(オーバーシュートやアンダーシュートの設定)、スルーレートの High、Low の設定、I/O の属性の変更(SUZAKU では I/O 電圧が 3.3V のため、属性は C-MOS のみ)等が出来ます。



Project Navigator に戻って top-IMP(top.vhd)の横の⊞をクリックして開いてください。ピンアサインのファイル"`top.ucf`"が出来上がっています。今回は PACE でピンアサインをしましたが、Text で編集することもできます。`top.ucf` をクリックすると、Processes のウィンドウに top.ucf のプロセスが表示されるので Edit Constraints(Text)をダブルクリックしてください。ピンアサインが Text で表示されます。

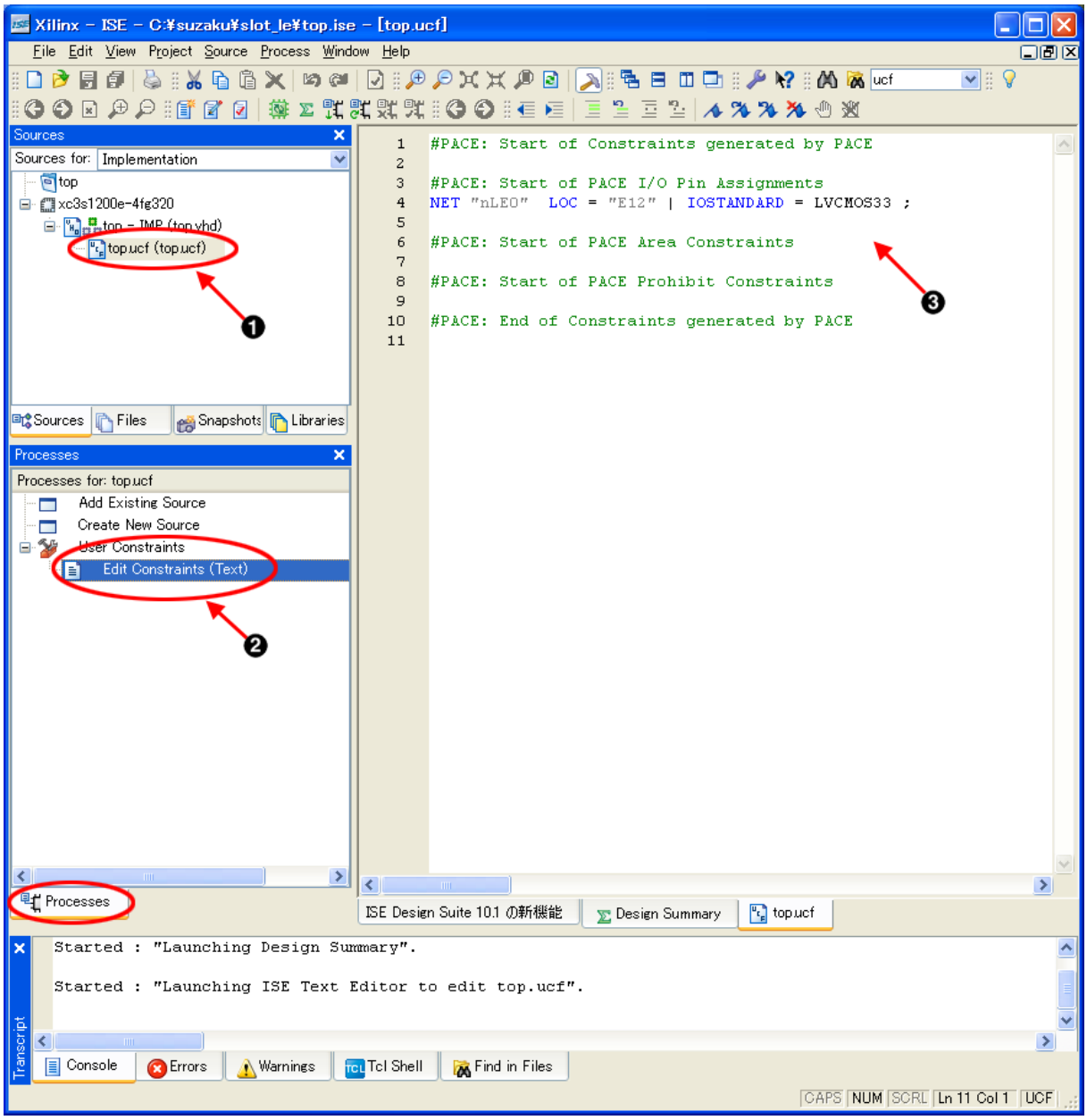


図 7.17. ピンアサインファイルの確認(SZ130 の場合)

❶　top_ucf(top.ucf)をクリック

❷　Edit Constraints(Text)をダブルクリック

❸　自動生成されたソースコード

top_IMP(top.vhd)をクリックし、Implement Design をダブルクリックしてください。残りのインプリメントが始まります。

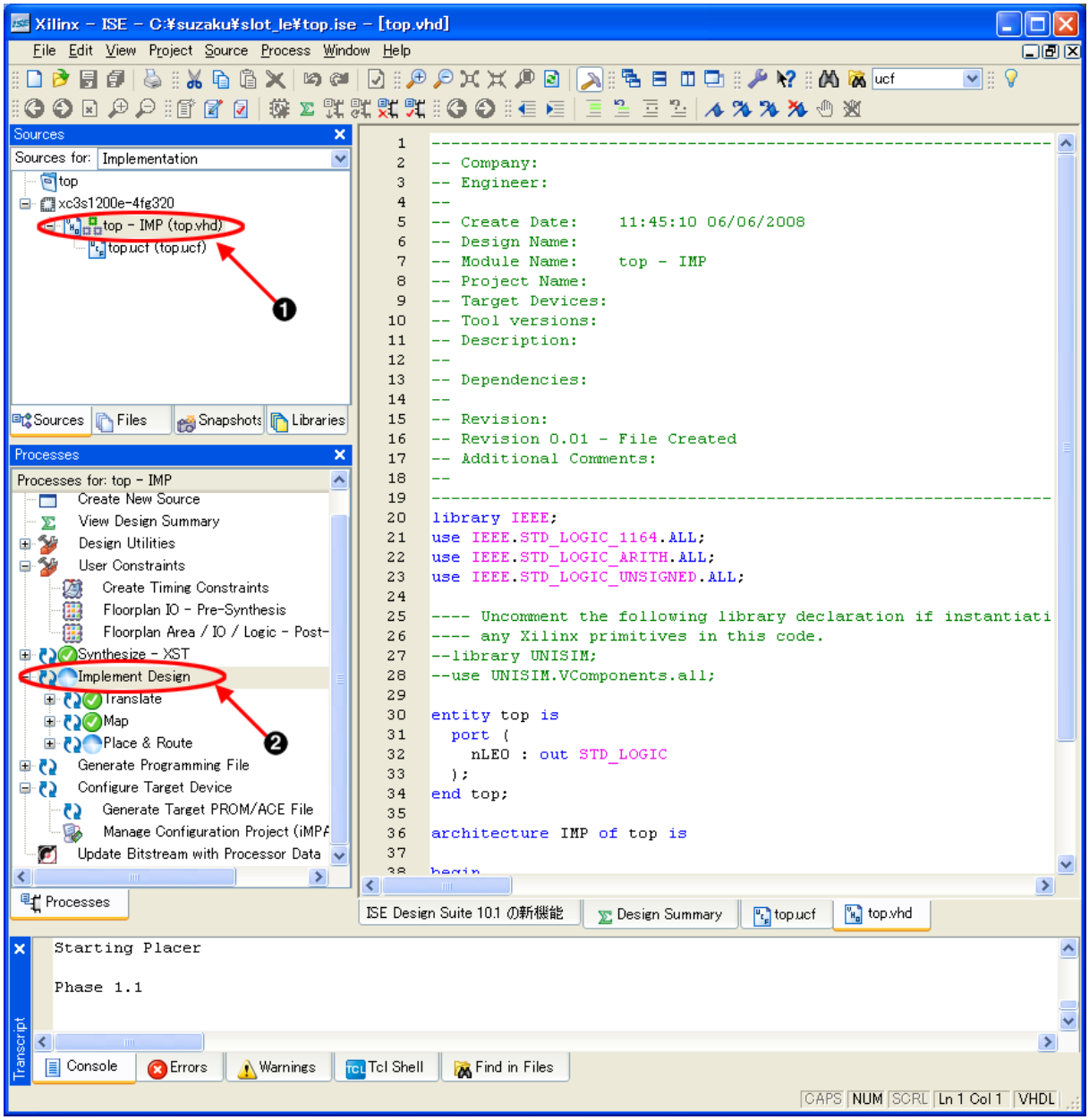


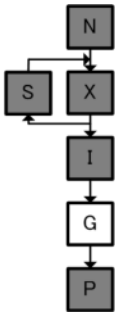
図 7.18. インプリメント

❶ top_IMP(top.vhd)をクリック

❷ Implement Design をダブルクリック

7.6. プログラムファイル作成

Generate Programming File をダブルクリックします。エラーがなければ、`top.bit` という FPGA コンフィギュレーション用の bit ファイルが生成されます。



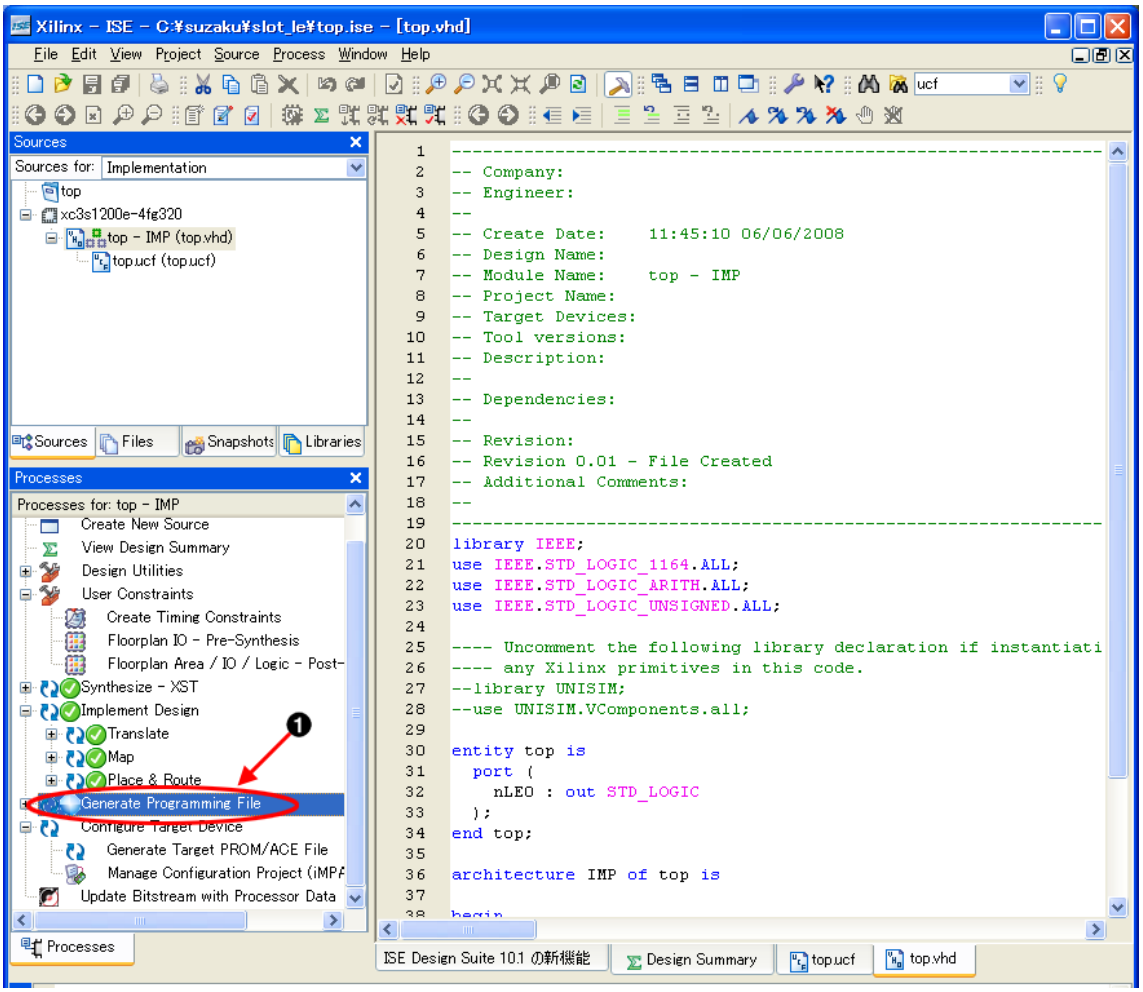
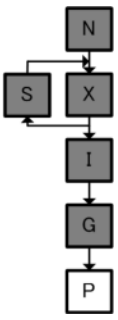


図 7.19. bit ファイル作成

❶ Generate Programming File をダブルクリック

7.7. コンフィギュレーション

FPGA に作成した `top.bit` を書き込みます。



7.7.1. JTAG でコンフィギュレーション

iMPACT で top.bit を書き込みます。iMPACT での FPGA の書き換え方については「6.2.1. iMPACT で書き換える」を参照してください。iMPACT は Manage Configuration Project (iMPACT)をダブルクリックすることでも起動することが出来ます。

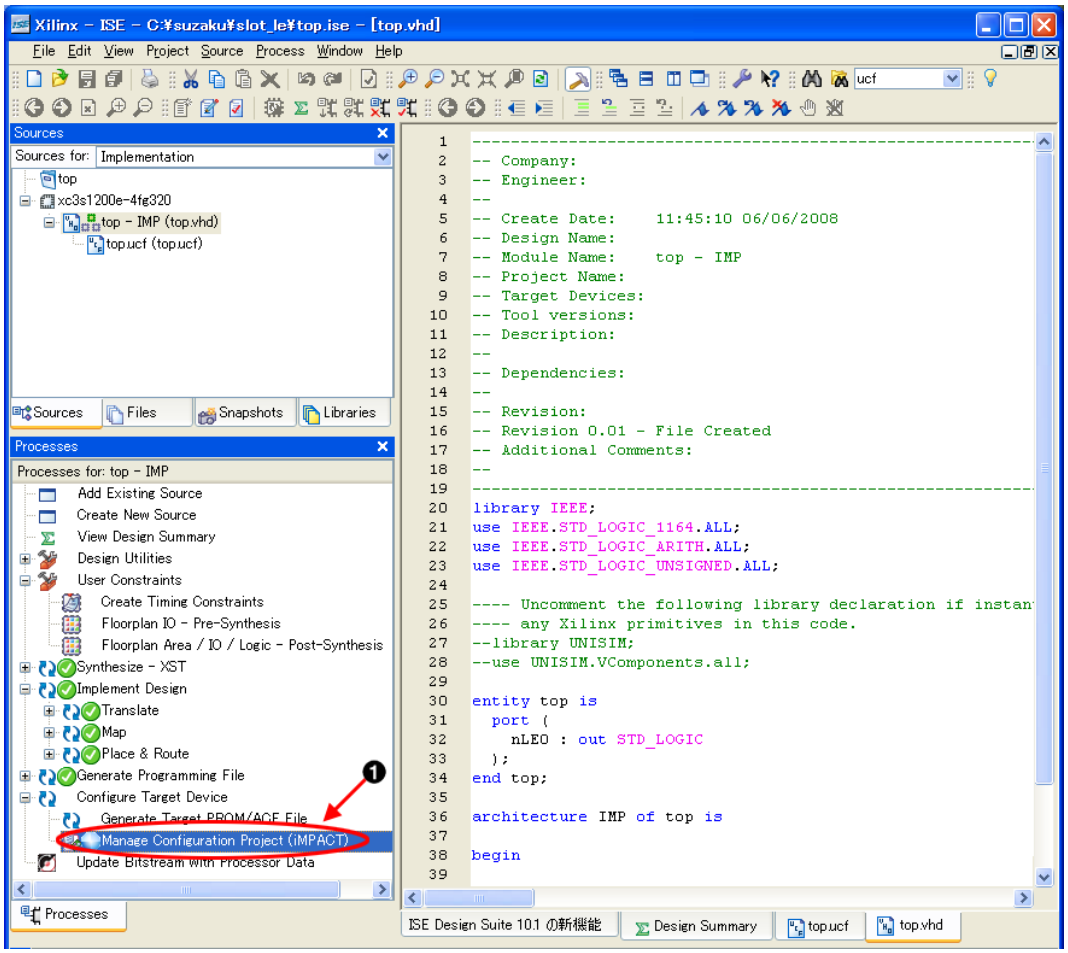


図 7.20. iMPACT 立ち上げ

❶　Manage Configuration Project (iMPACT)をダブルクリック

単色 LED(D1)が光ったでしょうか？

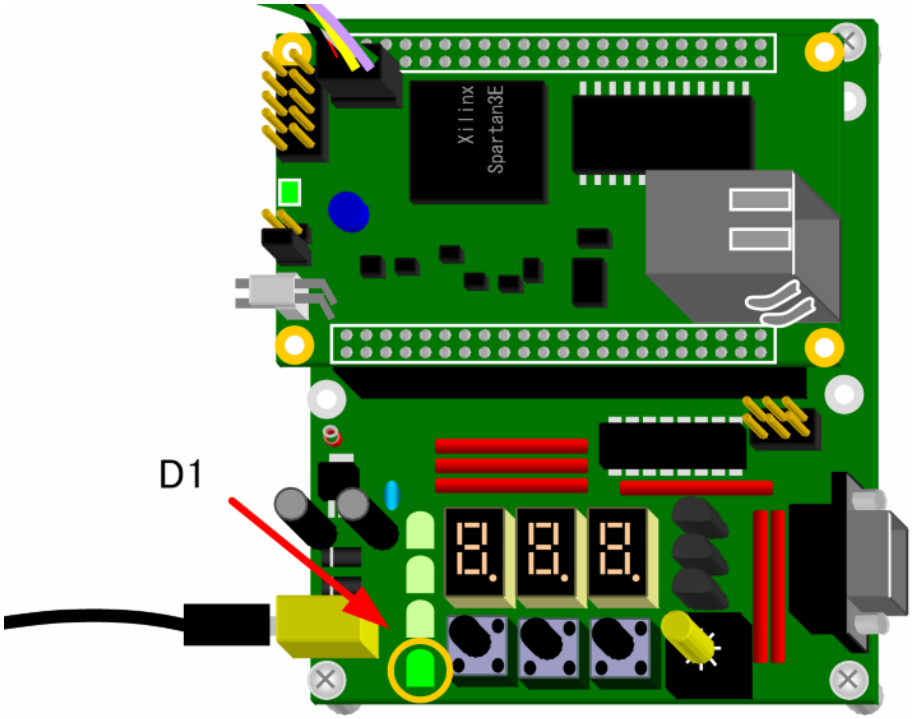


図 7.21. 単色 LED(D1)点灯

7.7.2. フラッシュメモリに保存してコンフィギュレーション

一回電源を切ってもう一度電源を入れてみてください。フラッシュメモリに保存されているデータがコンフィギュレーションされるので、スロットマシンの状態に戻ったと思います。これは Xilinx の FPGA が SRAM ベースのためです。内部の回路内容を保持させるにはフラッシュメモリにコンフィギュレーションデータを書き込む必要があります。フラッシュメモリへの書き込み方法については SZ010, SZ030, SZ130 の場合は「6.2.2. LBPlayer2 で書き換える」を、SZ130、SZ410 の場合は「6.2.3. SPI Writer で書き換える」を参照してください。フラッシュメモリに書き込むと、電源を切ってもコンフィギュレーションデータが失われません。

7.8. 空きピン処理

D1 を点灯させたとき、D2、D3、D4 が少し光っているのに気がついたでしょうか？(SZ310、SZ410 だとほとんど光りません)

これは空きピンの処理の仕方によります。

Generate Programming File を右クリックしてメニューを出し、Properties を選択してください。

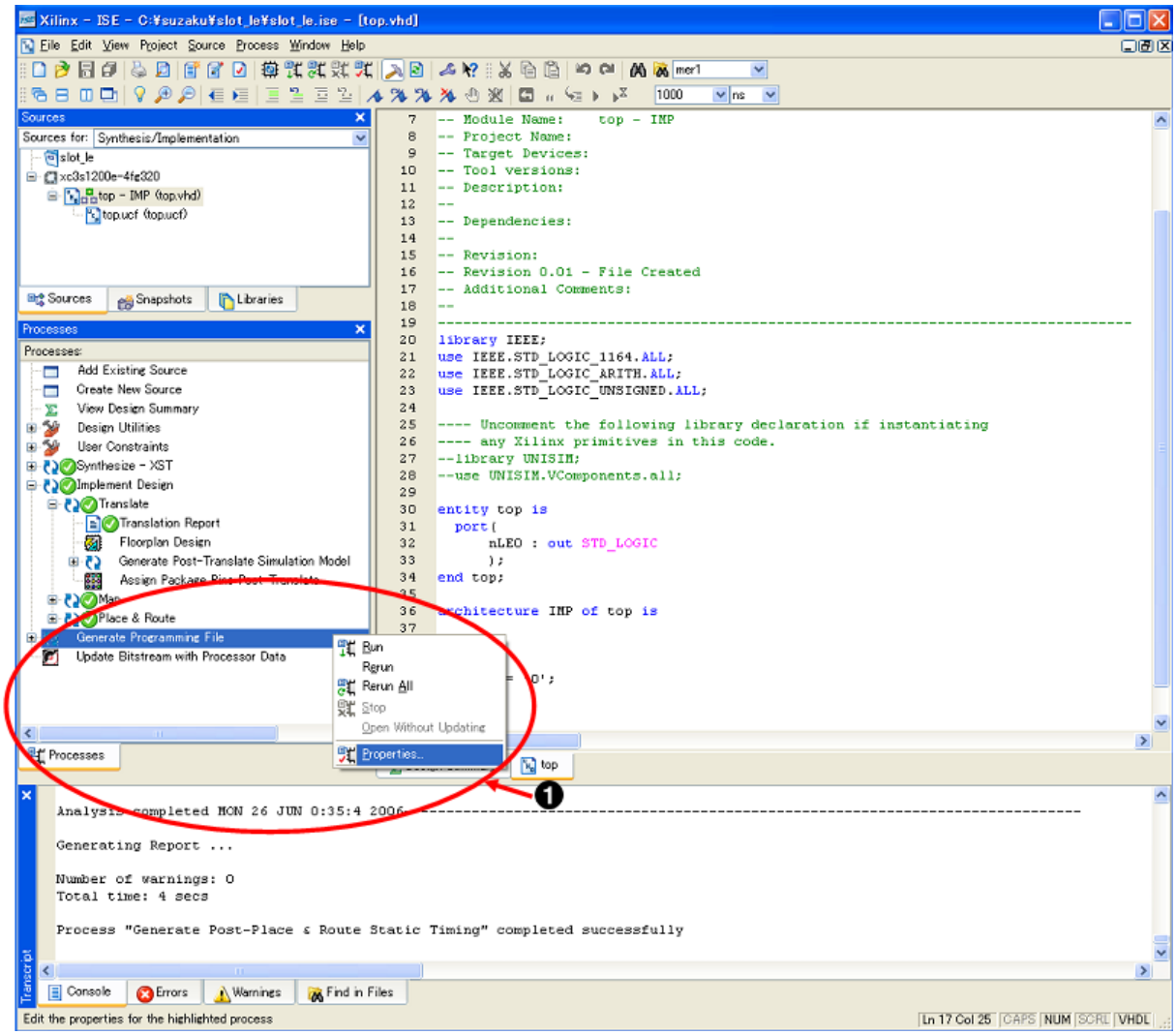


図 7.22. 空きピン処理の設定画面の出し方

❶　右クリックして、メニューを出し、Properties を選択

[Configuration Options]を選ぶと次の画面が出てきます。ここで空きピンの終端処理を設定できます。この中に[Unused IOB Pins]というのがありますが、これが空きピン処理の設定になります。初期設定で、[Pull Down]になっています。このため「図 7.24. 少し光る理由」のように D2、D3、D4 に電流が少し流れて LED が光ってしまいます。

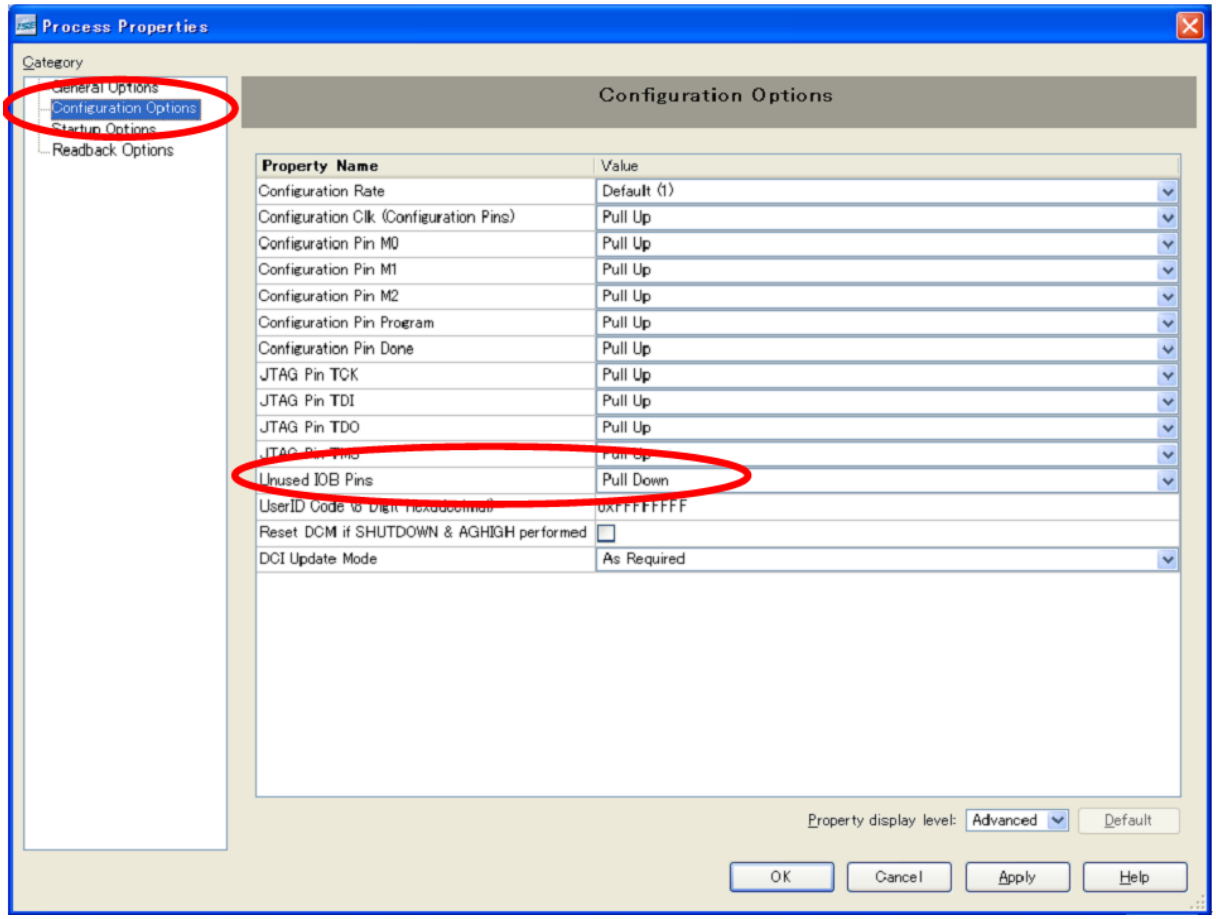


図 7.23. 空きピン処理設定

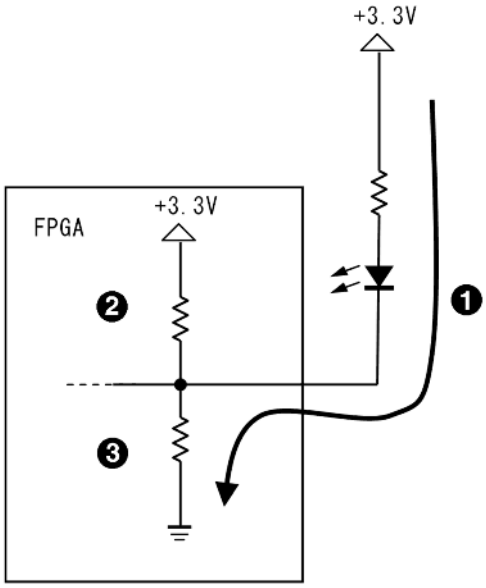


図 7.24. 少し光る理由

❶　電流が少し流れる

❷　ウィークプルアップ

❸　ウィークプルダウン

設定を[Pull Up]にすると、D2、D3、D4 は光らなくなりますが、ここの設定では空きピンの終端処理を個別に設定することが出来ないため、SUZAKU が動かなくなってしまう可能性があります。SUZAKU には RESET 回路があり、RESET 回路につながっているピンは外部で Pull Down されています。このピンの終端処理は Low かハイインピーダンスにしておかなければならず、High や Pull Up、Float に変更した場合、リセットがかかってしまうことがあります。また、空きピンから電圧が出力されているのはあまり良い状態ではありません。よって今回は D2、D3、D4 に信号を定義することで、この問題に対処します。`top.vhd`、`top.ucf` に D2、D3、D4 の信号の記述を加えてください。

例 7.1. 信号の記述を追記(top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity top is

  port (
    nLE0 : out  STD_LOGIC;

    nLE1 : out  STD_LOGIC;
    nLE2 : out  STD_LOGIC;
    nLE3 : out  STD_LOGIC

  );

end top;

architecture IMP of top is
begin

  nLE0 <= '0';

  nLE1 <= '1';
  nLE2 <= '1';
  nLE3 <= '1';

end IMP;
```

表 7.3. ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nLE1	C12	F12	L15	F2
nLE2	D11	B11	L14	F1
nLE3	E11	A11	L13	E1

8.VHDL によるロジック設計

本書を読むために必要となる最低限の VHDL の記述方法とロジック設計について説明します。VHDL の詳細やロジック設計については、世の中に詳しい書物が多数ありますのでそちらをご参照ください。

8.1. VHDL の基本構造

まず VHDL の記述方法を説明します。

VHDL の基本構造は

- ライブラリ宣言とパッケージ呼び出し
- エンティティ(entity)
- アーキテクチャ(architecture)

からなります。

ライブラリ宣言とパッケージ呼び出しで、各種演算子や関数などを定義したパッケージを呼び出し、エンティティに外部とのインターフェースを記述し、アーキテクチャに内部回路の構造や動作を記述します。

VHDL では予約語も含めて大文字と小文字を区別しません。例えば Port は port とかいても PORT とかいても同じに扱われます。予約語については後述の Tips を参照してください。もし、ISE 付属のテキストエディタを使っている場合、予約語は青に色が変わって表示されます。また、--で始めるとその行末までがコメントになります。

例 8.1. VHDL 基本構造

```
--ライブラリ宣言とパッケージ呼び出し　❶
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--エンティティ(入出力の宣言)
entity slot is
  Port (　❷
    --ここに入出力ピンの宣言を書く
);
end slot;

--アーキテクチャ(回路本体)
architecture IMP of slot is

  --内部信号等の各種宣言を記述する

begin
  --ここに回路を記述する
end IMP;
```

❶ -- の後はコメント文

❷ Port も port も PORT も同じ

8.2. ライブラリ宣言とパッケージ呼び出し

ISE で VHDL ソースコードを自動生成すると、パッケージが 3 つ呼び出されます。それぞれの用途は下表の通りです。他にも様々なパッケージがあるので、必要に応じて呼び出してください。ライブラリは自分で作成することも出来ます。

表 8.1. ライブラリとパッケージ

ライブラリ	パッケージ	用途
IEEE	std_logic_1164	基本関数
	std_logic_arith	算術演算
	std_logic_unsigned	符号なし演算

8.3. エンティティ(entity)

エンティティ内ではポートの宣言を行います。外部とのインターフェースについて定義する部分がエンティティになります。ISE で VHDL ソースコードを自動生成すると、エンティティ名はファイル名と同じ名前になります。

例 8.2. entity 記述

```
entity slot is --entity エンティティ名 is
  Port (
    SYS_CLK : in std_logic;
    nLE : out std_logic_vector(0 to 2);
    nSW : in  std_logic_vector(0 to 2) --最後に ; は不要
  );
end slot; --end エンティティ名;
```

8.3.1. 信号の定義

信号は以下の形式で宣言します。

例 8.3. 信号の定義

```
ポート信号名：入出力方向 データタイプ名;
```

8.3.2. 入出力方向

入出力方向には in、out、inout 等を記述します。

表 8.2. 入出力方向

in	入力であることを指定
out	出力であることを指定(内部で再利用できない)
inout	入出力であることを指定

VHDL では、出力ポート信号を内部に参照できません。内部で参照したいときは、内部参照用に内部信号を用います。内部信号の宣言については「8.4. アーキテクチャ(architecture)」の内部信号の定義の項を参照してください。

8.3.3. データタイプ

データタイプには色々ありますが、よく使うのは std_logic と std_logic_vector です。std_logic で１ビットの信号を定義し、std_logic_vector(0 to n) で n ＋ 1 ビット幅の信号を定義します。

nLE : out std_logic_vector(0 to 2) とすると 3 ビットの幅を持った出力信号を定義することができます。nLE(0)、nLE(1)、nLE(2)とすることで、それぞれのビットを切り出すことができます。

to を使って定義すると、MSB 側がビット 0 になります。(downto とすると LSB 側がビット 0 になります。本書では IBM の CoreConnect にあわせてバスを定義するため(「9.1.7. バスのビットラベルについて」参照)、to を使います。)

表 8.3. データタイプ

std_logic	IEEE ライブラリで定義
std_logic_vector	std_logic のベクタ・タイプ
integer	整数型(32 ビット)

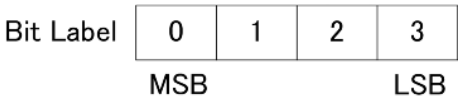


図 8.1. to を使って定義

8.4. アーキテクチャ(architecture)

回路の構造や動作などをここに記述をします。アーキテクチャ名は任意ですが、SUZAKU では IMP としています。

例 8.4. architecture 記述

```
architecture IMP of slot is --architecture アーキテクチャ名 of エンティティ名 is
--内部信号の定義
  signal count : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1);
  signal count_led : STD_LOGIC;

begin
--ここに同時処理文を記述する
  nLE <= not le; --信号代入文                    ❶
  process(SYS_CLK) --プロセス文
  begin
    if SYS_CLK'event and SYS_CLK = '1' then      ❶
      if SYS_RST = '1' then
        count <= (others => '0');
      else
        count <= count + 1;
      end if;
    end if;
  end process;
  end IMP; --end アーキテクチャ名;
```

❶ 同時に処理される

8.4.1. 内部信号の定義

内部で使用する信号は architecture と begin の間に記述します。信号の宣言には signal を用い、以下の形式で宣言します。データタイプ名はエンティティの信号宣言と同じですので、「8.3. エンティティ(entity)」のデータタイプの項をご参照ください。

例 8.5. 内部信号定義

```
signal 信号名 : データタイプ名 ;
```

8.4.2. 同時処理文

begin 縲ｲ nd の間に直接記述された回路を同時処理文といいます。同時処理文ではそれぞれが他の同時処理文と関係なく動作し、並列に処理されます。信号代入文、プロセス文などの回路を記述します。

8.4.3. 信号代入文

A<=B;とすると、A に B が代入されます。

8.4.4. プロセス文

プロセス文は以下の形で記述します。

例 8.6. プロセス文

```
process(センシティビティリスト)
begin
 ・
 ・
 ・
end process;
```

センシティビティリストに記述した値のどれかが変化すると、中に記述した文が上から実行されていきます。最終行まで実行すると上に戻り、次にこれらの信号が変化するまで動作を停止します。

VHDL 予約語

abs, access, after, alias, all, and ,architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, constant, disconnect, downto, else, elsif, end, entity, exit, file, for, function, generate, generic, guarded, if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod, nand, new, next, nor, not, null, of, on, open, or, others, out, package, port, postponed, process, pure, range, record, register, reject, rem, report, return, rol, ror, select, severity, shared, signal, sla, sll, sra, srl,

subtype, then, to, transport, type, unaffected, units, until, use, variable, wait, when, while, with, xnor, xor

8.5. 組み合わせ回路(not、and、or)

ここからは少しロジック設計について説明します。

"not"、"and"、"or"などの基本論理ゲートを組み合わせて作られるものを組み合わせ回路といい、クロックを必要とせずに現在の入力だけで出力が決まります。押しボタンスイッチと単色 LED を使って基本論理ゲートの動作を確認します。

8.5.1. 押しボタンスイッチ周辺回路

組み合わせロジックの入力として、押しボタンスイッチを利用します。押しボタンスイッチは下記のような回路になっています。単色 LED の周辺回路は「図 7.2. 単色 LED 周辺回路」を参照してください。

ボタンを押していないと"High"が FPGA に入力され、ボタンを押していると"Low"が FPGA に入力されます。

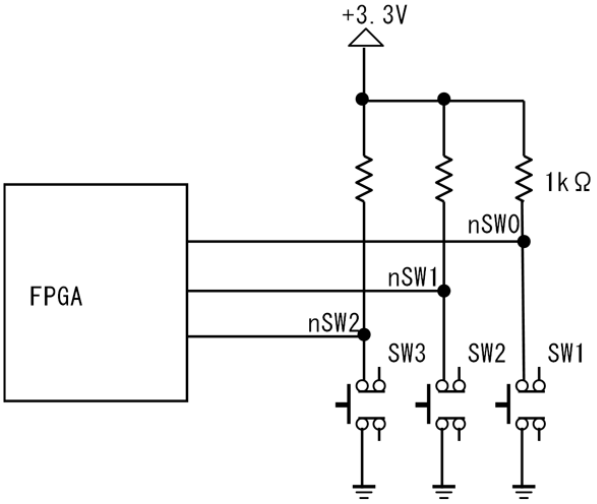


図 8.2. 押しボタンスイッチ周辺回路

8.5.2. not、and、or を使う

信号には正論理、負論理の 2 種類の表現があります。例えば、単色 LED を LE という信号名で定義し、"High"("1")で点灯した場合は正論理、nLE という信号名で定義し、"Low"("0")で点灯した場合は、負論理となります。(nLE の n は負論理だということを明言するために使います)

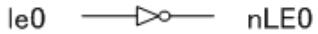
LED/SW ボードには正論理、負論理の信号が混在しているので、分かりやすくするために負論理の信号は FPGA 内部で反転させて正論理として扱うようにします。

8.5.2.1. not

負論理から正論理(正論理から負論理)は次の一文で記述できます。

例 8.7. not 記述

```
nLE0 <= not le0;
```



le0	nLE0
0	1
1	0

図 8.3. not 回路と真理値表

8.5.2.2. and

SW1(信号名：sw0)と SW2(信号名：sw1)を両方押したら D1(信号名：le0)が点灯するというのは以下の一文で記述できます。

例 8.8. and 記述

```
le0 <= sw0 and sw1;
```

sw0	sw1	le0
0	0	0
0	1	0
1	0	0
1	1	1

図 8.4. and 回路と真理値表

8.5.2.3. or

SW 1 (信号名：sw0)か SW2(信号名：sw1)のどちらか一方でも押されたら D1(信号名：le0)が点灯するというのは以下の一文で記述できます。

例 8.9. or 記述

```
le0 <= sw0 or sw1;
```

sw0	sw1	le0
0	0	0
0	1	1
1	0	1
1	1	1

図 8.5. or 回路と真理値表

8.5.2.4. not, and, or の top.vhd

先ほど単色 LED(D1)を光らせたプロジェクトを変更して試してみてください。

例 8.10. not、and、or(top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--エンティティ(入出力の宣言)
entity top is
  Port (
    nLE0 : out STD_LOGIC; --単色 LED(D1)への出力信号(負論理)
    nSW0 : in STD_LOGIC; --スイッチ(SW1)からの入力信号(負論理)
    nSW1 : in STD_LOGIC --スイッチ(SW2)からの入力信号(負論理)
  );
end top;

--アーキテクチャ(回路本体)
architecture IMP of top is

  signal le0 : STD_LOGIC; --単色 LED 内部信号(正論理)
  signal sw0 : STD_LOGIC; --スイッチ(SW1)内部信号(正論理)
  signal sw1 : STD_LOGIC; --スイッチ(SW2)内部信号(正論理)

begin

  sw0 <= not nSW0; --not 回路で入力前に正論理にする
  sw1 <= not nSW1; --not 回路で入力前に正論理にする

  le0 <= sw0 and sw1; --and 回路(両方押したら LED が光る)
  --le0 <= sw0 or sw1; --or 回路(どちらか一方でも押したら LED が光る)

  nLE0 <= not le0; --not 回路で出力前に負論理にする

end IMP;
```

8.5.2.5. not, and, or のピンアサイン

ピンアサインは以下になります。

表 8.4. not、and、or のピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nLE0	B12	E12	L16	G2
nSW0	A13	F11	K14	G4
nSW1	B14	C11	K15	M1

8.6. 順序回路

その時点の入力だけでなく、過去の入力信号にも依存する回路を順序回路といいます。値を保持する、そのまま出力する、といったことができます。

順序回路は基本的に同期設計により成り立ちます。非同期設計は現在の状況に応じて物事が動き、次に何が起こるか分からなくなるので、順序回路には向きません。もし非同期信号を使いたい場合は、通常 1 回クロックに同期させてから使います。

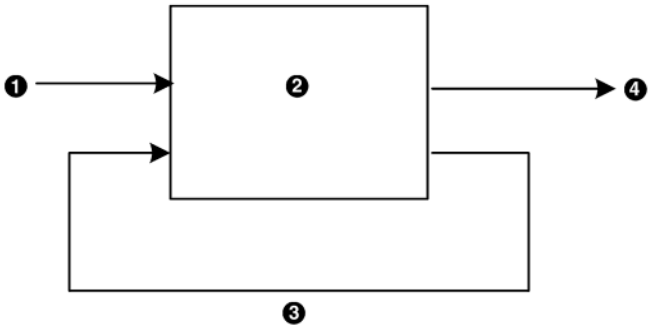


図 8.6. 順序回路の概念図

❶　入力

❷　順序回路

❸　出力は過去の入力にも依存

❹　出力

8.6.1. D-FF(D 型フリップフロップ)

順序回路で重要なのは D-FF です。

クロックの立ち上がりでデータを保持し、次のクロックで保持したデータを出力します。クロックの立ち上がり以外でデータが変化しても出力は変化しません。

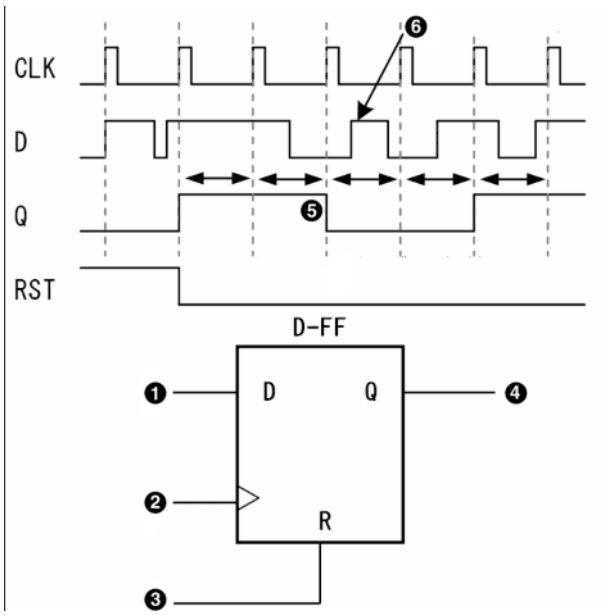


図 8.7. D-FF の動作

❶ 入力

❷ CLK クロック

❸ RST リセット

❹ 出力

❺ 次のクロックまでデータ保持

❻ ここでは出力は変化しない

8.6.2. 同期設計

回路は入力信号の時間差によって動作が決まります。非同期設計では ns 単位で時間差を作ってしまうことがあり、タイミング設計が非常に困難です。論理合成、配置配線による信号の遅延はツールの種類やバージョンに依存します。また、温度やデバイスの個体差によっても信号が遅延します。これらのすべての遅延を非同期設計で押さえ込むのは至難の業です。押さえ込むのに失敗すると、タイミング不良を起こし、次に何が起こるのか分からなくなってしまいます。

それにひきかえ同期設計はタイミング設計が簡単になります。同期設計では、同期用クロックの周期時間よりもゲートや配線による遅延やセットアップなどの積算時間ほうが短ければ、回路が設計通りに動作することが保証されています。FPGA は内部にクロック専用線を複数もっていて、これらのクロック専用線は他の線に比べて Delay が少なく、信号が速く到達することが保証されています。

よって、一般的に FPGA ではこのクロック専用線を用い、同期設計を行います。

同期回路は組み合わせ回路と D-FF とで成り立っています。組み合わせ回路の規模を小さくすることで、遅延は少なくなり速い回路を作ることができます。どこに D-FF を入れ、組み合わせ回路の規模どう小さくするかで、全体の最高動作周波数が決まってきます。

8.6.3. カウンタ

順序回路の基本的な例としてカウンタを上げます。カウンタはクロックにあわせて数値をインクリメント(デクリメント)します。カウンタの回路は以下のように記述できます。

例 8.11. カウンタ記述

```
process(SYS_CLK) --クロック信号に変化があると実行
begin
  if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
    if SYS_RST = '1' then --リセットされたら(同期リセット)
      count <= (others => '0'); --カウンタ初期化
    else --その他は
      count <= count + 1; --カウント値をインクリメント
    end if;
  end if;
end process;
```

8.6.3.1. クロックの記述

クロックの立ち上がりエッジに同期させたい場合以下のように記述します。 SYS_CLK = '0'とすると立下りエッジに同期させることができます。

例 8.12. クロックの立ち上がりエッジに同期

```
if SYS_CLK'event and SYS_CLK='1' then
```

8.6.3.2. リセットの記述

クロックの記述の下にリセットを記述すると同期リセット、上に記述すると非同期リセットになります。

SUZAKU には電源監視 IC が実装されており、電源投入時にリセットがかかるようになっています。このリセット信号を用いて、信号の初期化を行います。VHDL ではこの様に外部からのリセットで初期化する方法の他に内部信号定義の時に初期化する方法もあります。

例 8.13. 同期リセット

```
if SYS_CLK'event and SYS_CLK = '1' then
  if SYS_RST = '1' then
```

8.6.3.3. if 文

if 文は以下の形式で記述します。

例 8.14. if 文

```
if 条件 then
  順次処理文
elsif 条件 then
  順次処理文
else
  順次処理文
end if;
```

8.6.3.4. other で初期化

others は残りすべてという意味で、others=>'0'とすると、残っているビットすべてに 0 が代入されます。

例 8.15. other で初期化

```
count <= (others => '0');
```

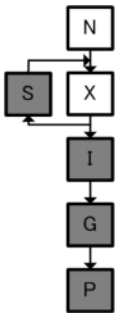
8.7. ISE Simulator の使い方

HDL でコーディングを行ったら、PC 上でシミュレーションを行います。シミュレーションはロジック設計で重要な作業です。実際にデバイスに書き込んでからでは、各信号の挙動をとても検証しづらいですが、PC 上のシミュレーションでは、信号の挙動が明快にわかります。シミュレーションで自分の考えていた通りに動作しているか確認してから、実際のデバイスに書き込みます。

最も基本の順序回路であるカウンタの動きを確認すると共に、ISE に含まれているシミュレータ ISE Simulator の使い方を説明します。

8.7.1. プロジェクトの新規作成

プロジェクトを新規作成してください。プロジェクト名は slot_counter とし、[New Source]で `slot_counter.vhd` とし、新規ソースコードを作ってください。



8.7.1.1. slot_counter.vhd

カウンタ回路を記述してください。今回は、4 ビットカウンタのシミュレーションを行います。4 ビットカウンタでは 0 から 15 まで数えることができます。

記述できたら Synthesize をダブルクリックして文法に間違いがないかチェックしてください。

例 8.16. カウンタ(slot_counter.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity slot_counter is
  generic (
    C_CNT_WIDTH : integer := 4 --カウンタのビット幅
  );
  Port (
    SYS_CLK : in STD_LOGIC; --クロック信号
    SYS_RST : in STD_LOGIC; --リセット信号
    count : out STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1) --カウンタ値
  );
end slot_counter;

architecture IMP of slot_counter is
--内部信号の定義
  signal count_w : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1); --カウンタ値内部用
begin

  process(SYS_CLK) --クロック信号に変化があると実行
  begin
    if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
      if SYS_RST = '1' then --リセットされたら(同期リセット)
        count_w <= (others => '0'); --カウンタ初期化
      else  --その他は
        count_w <= count_w + 1; --カウント値をインクリメント
      end if;
    end if;
  end process;

  count <= count_w; --カウンタ値を外部に出力

end IMP;
```

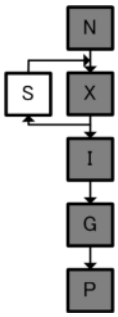
8.7.1.2. generic 文について

バスの幅などのパラメータを渡す時などに使います。記述形式はポート文とほぼ同じですが、情報を渡すだけなので、"in"や"out"などの方向の指定はありません。

例 8.17. generic 文

```
generic (
  信号名 : データタイプ名 := 初期値
);
```

8.7.2. テストベンチの新規作成



カウンタの動作をシミュレーションで確認します。
[Project]→[New Source]をクリックしてください。
[Test Bench WaveForm]を選択し、[File name]にファイル名を入力し、[Next]をクリックしてください。ここではファイル名を slot_counter_tb とします。

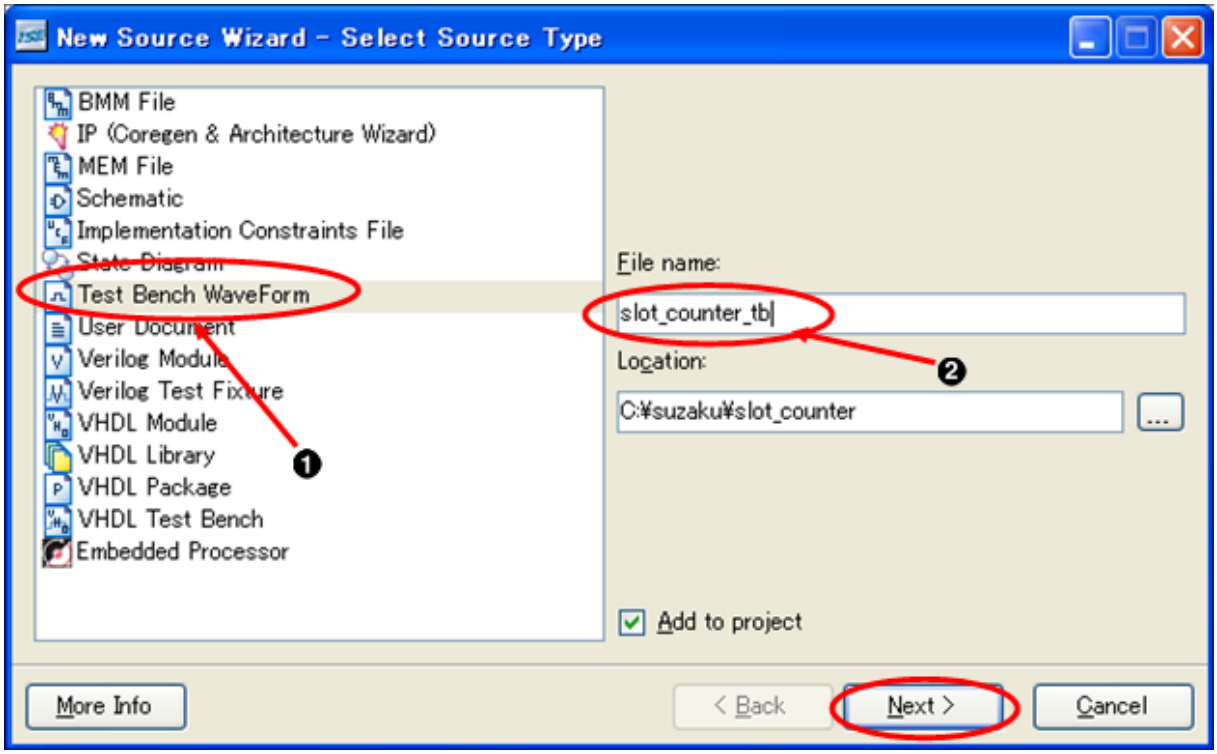


図 8.8. テストベンチ作成

❶　Test Bench WaveForm を選択

❷　slot_counter_tb と入力

次の画面が出るまで[Next]および[Finish]をクリックしてください。クロック波形を作成します。[Initial Length of Test Bench] を 10000 に変更して[Finish]をクリックしてください。[Initial Length of Test Bench]を変更すると、シミュレーション時間を変更することができます。他にも色々設定を変えることができますが、今回はカウンタの動きを見たいだけなので変更しません。

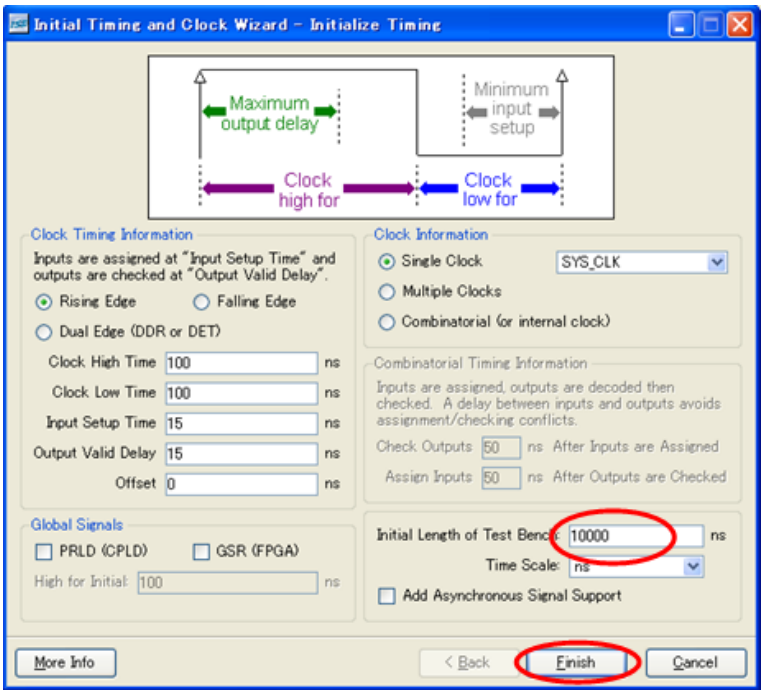


図 8.9. クロック波形作成

次の画面が表示されます。

Source ウィンドウの Source for:を[Behavioral Simulation]に変更してください。

リセット信号を入力しないと信号が初期化されないので、リセット信号を入力します。クロックが細かくて見にくいので キーを押して適当な大きさに拡大してください。水色のセルをクリックすると信号の"High"、"Low"を切り替えることができます。図のように SYS_RST の信号を生成してください。100ns で信号を立ち上げ、500ns で信号を立ち下げています。

[File]→[Save]をクリックし保存してください。

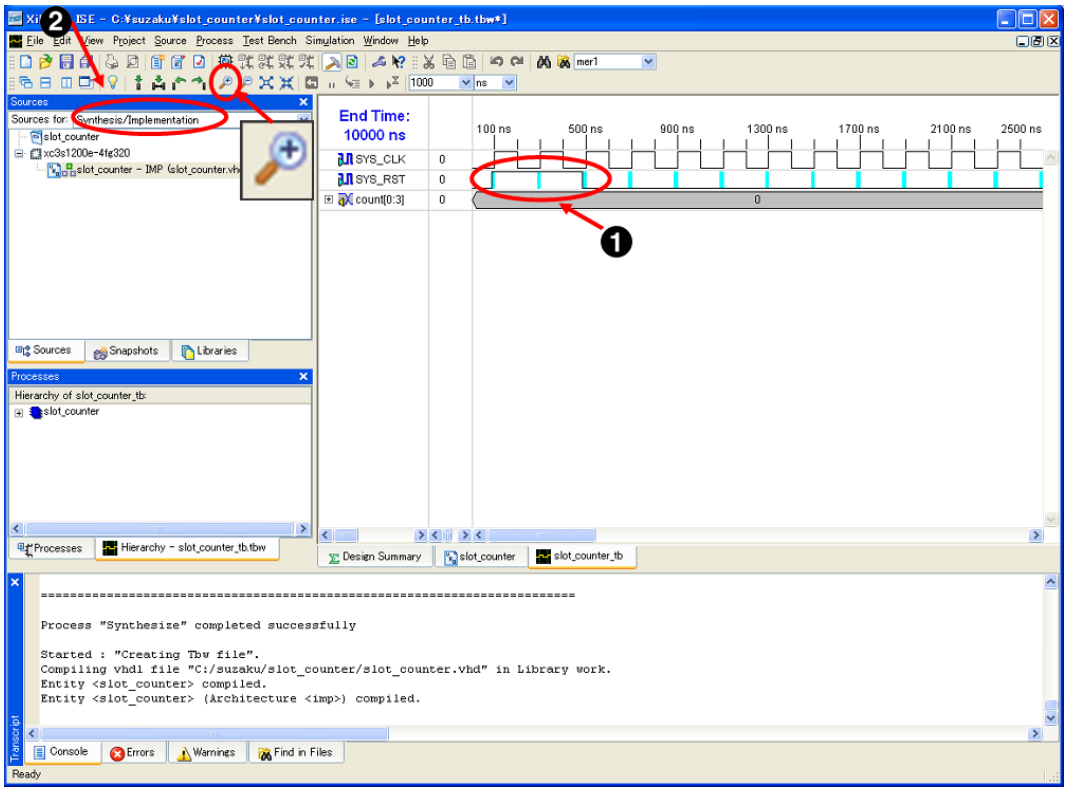


図 8.10. リセット波形生成

❶　リセット信号生成

❷　Behavioral Simulation に変更

シミュレーションの設定をします。[Simulate Behavioral Model]の上で右クリックをし、メニューで[Properties…]を選択してください。Process Properties が表示されるので、Simulation Run Time を 10000ns に変更し、[OK]をクリックしてください。

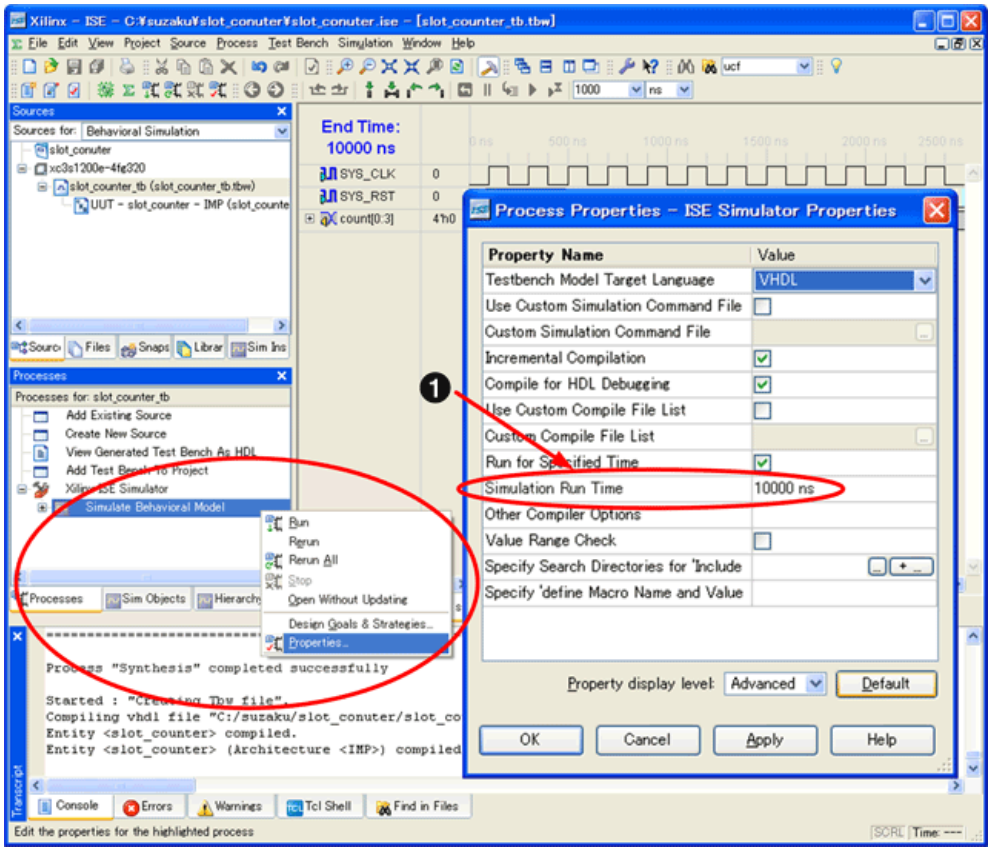


図 8.11. シミュレーション設定

❶　10000ns に変更

8.7.3. シミュレーション実行

Processes タブをクリックしてください。View Generated Test Bench As HDL をダブルクリックすると、自動生成されたテストベンチを見ることができます。Xilinx ISE Simulator の⊞をクリックして開き、Simulate Behavioral Model をダブルクリックしてください。シミュレーション結果が表示されます。

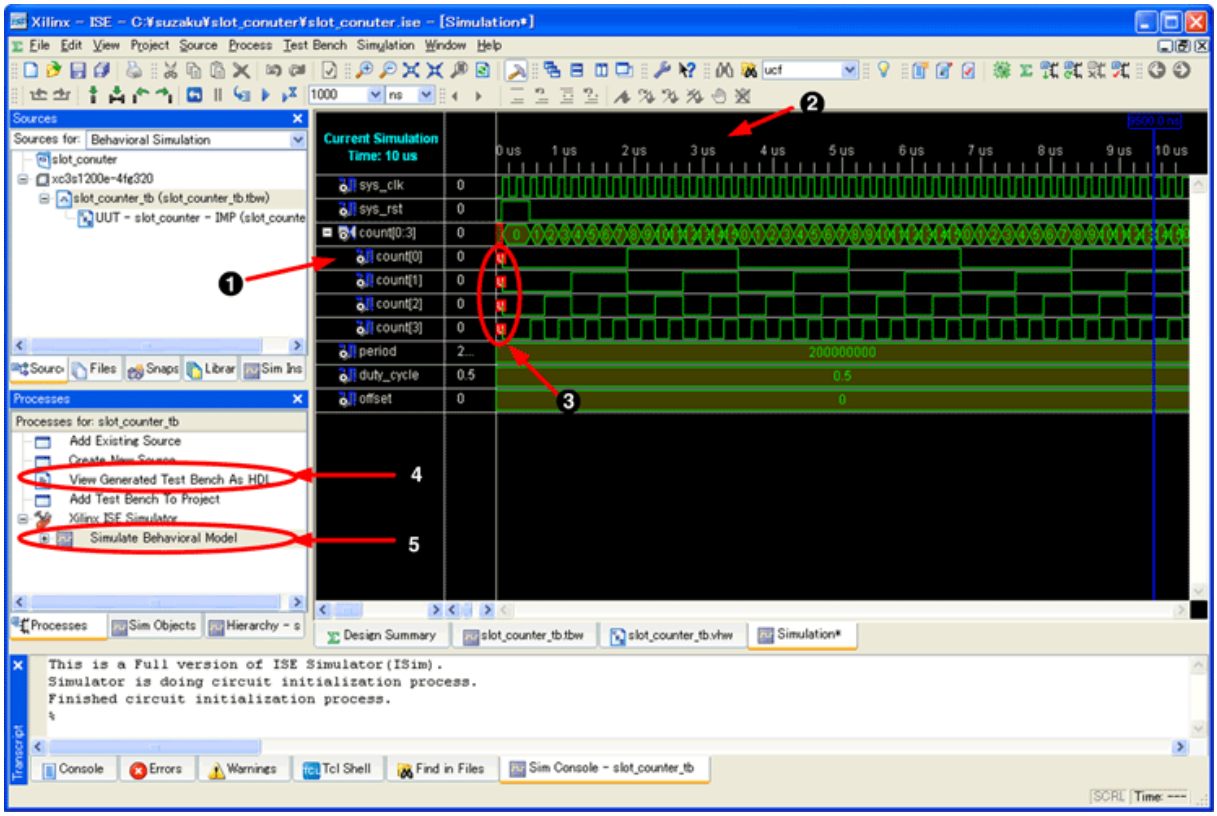


図 8.12. シミュレーション結果

❶ 0 - 15 のカウントを繰り返す

❷ ‘1’、’0’を繰り返す

❸ 初めは値が不定

❹ View Generated Test Bench As HDL をダブルクリック

❺ Simulate Behavioral Model をダブルクリック

クロックについて	sys_clk の波形を見てください。クロックは"1"、"0"を繰り返します。
リセットについて	count の波形の一番初めは赤い線で u と書かれています。これは値が不定という意味です。sys_rst が"High"になると、初期化されて値が決定します。
カウンタの波形チェック	sys_clk の立ち上がりのタイミングごとにカウントアップしているのが分かります。 count[3]の波形の周期は sys_clk の倍、count[2]の波形の周期は count[3]の倍、count[1]の波形の周期は count[2]の倍・・・となっています。これを分周といいます。

VHDL による回路設計について、この後は必要に応じて説明していきます。

9.FPGA 入門 スロットマシン製作

ここからは、本格的にロジック設計を行います。

「5.3. ブートローダモードでスロットマシンを動かす」で動かしたスロットマシンは下図の構成で作られています。これと同じスロットマシンを製作していきます。スロットマシンの機能を実現するには色々な方法が考えられるのですが、SUZAKU のデフォルトにスロットマシンの IP コアを接続し、ソフトウェアで制御することによりスロットマシンを製作しています。

本章ではまず、下図の右側の IP コアの中身を製作します。左側のソフトウェアについては後の章で説明します。

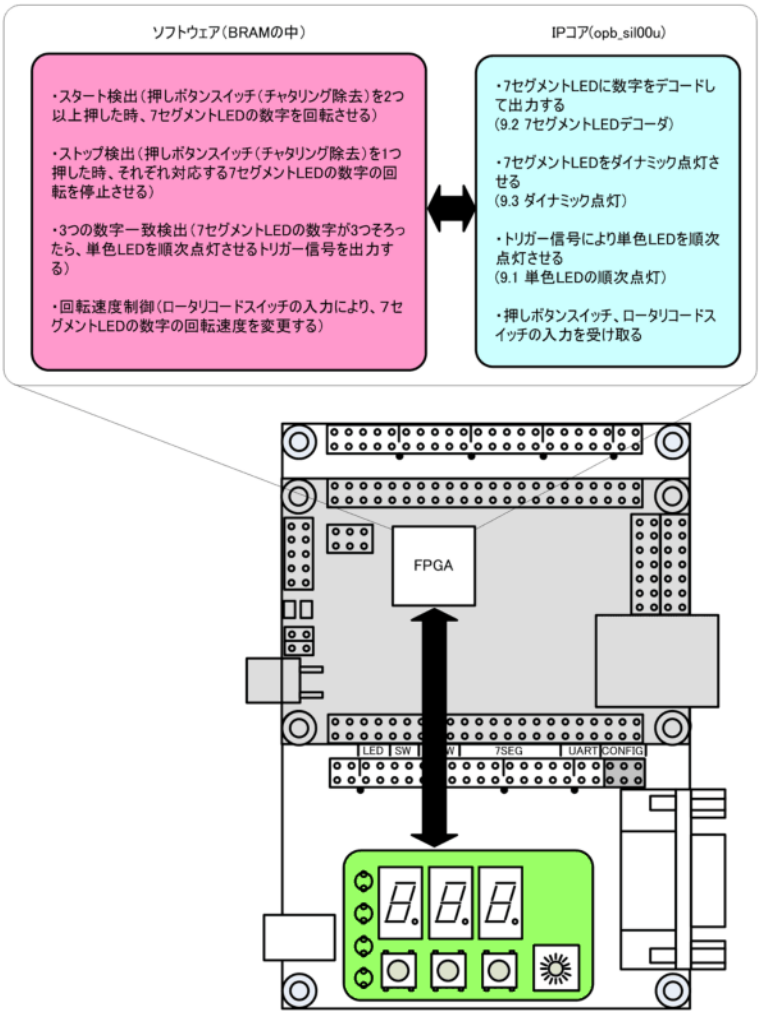


図 9.1. スロットマシンの構成

9.1. 単色 LED の順次点灯

まず、"トリガー信号により単色 LED を順次点灯させる"の単色 LED の順次点灯の部分を製作します。

スロットが当たった時に、当たった！という感じを出すために、単色 LED を順次点灯(D1→D2→D3→D4→D1→・・・・)させます。

SZ010,SZ030,SZ130,SZ310 のクロックは 3.6864MHz、SZ410 のクロックは 100MHz となっています。このクロックをタイミング信号として単色 LED を順次点灯させると速すぎるので、目に見えるくらいの速さのタイミング信号をカウンタで作ります。カウンタは先ほどシミュレーションの時に作った回路をそのまま使います。シミュレーションはビット幅 4 ビットで行います。シミュレーション後はカウンタのビット幅を SZ010、SZ030、SZ130、SZ310 の場合は 19 ビットに、SZ410 の場合は 23 ビットとします。カウンタの最上位ビット count(0)の値は SZ010、SZ030、SZ130、SZ310 の場合は 2^{19}=524288 カウントごとに(約 7Hz)、SZ410 の場合は 2^{23}=8388608 カウントごとに(約 12Hz)"0"、"1"を繰り返します。

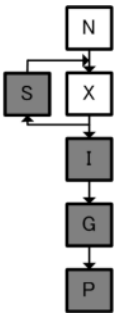
単色 LED を順次点灯させるのに、シフトレジスタを用います。シフトレジスタをシフトさせる一番簡単な条件は、タイミング信号が"0"または"1"の時、常にシフトすることです。カウンタの最上位ビットから出力される"0"、"1"はデューティー比 50:50 になっていて、このままだと、一番簡単な条件でシフトレジスタを作った場合、同じレベルの間は常にシフトし続けてしまうので使えません。このため count(0)の値が"0"から"1"になる時のエッジを検出し、1 クロックだけ"1"を出力するタイミング信号を作ります。

9.1.1. 単色 LED 周辺回路

単色 LED 周辺回路は「図 7.2. 単色 LED 周辺回路」をご参照ください。

9.1.2. プロジェクト新規作成、論理合成

プロジェクトを新規作成してください。
プロジェクト名は le_seq_blink とし、new Source で top.vhd を作ってください。
top - IMP(top.vhd)を右クリックしてメニューを出し、[New Source...]を選択してください。



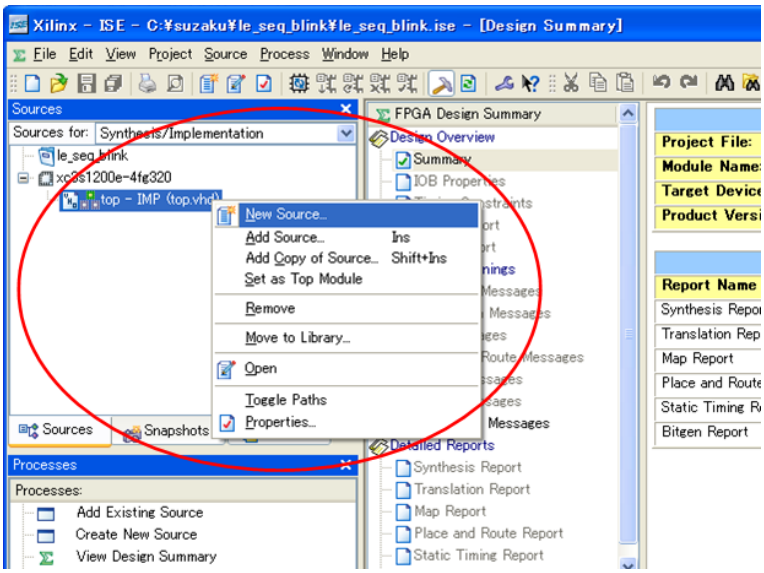


図 9.2. New Source の追加

New Source Wizard が立ち上がるので、[VHDL Module]を選択し、[File name]に le_seq_blink と入力し、新しいソースファイルを作ってください。

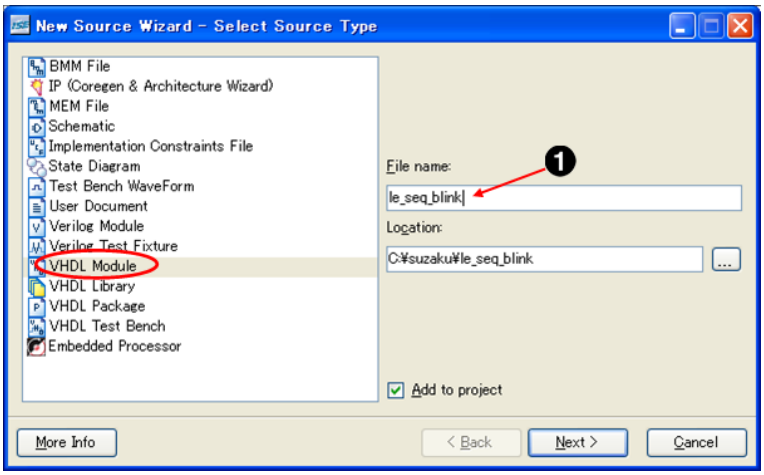


図 9.3. New Source 名前入力

❶　le_seq_blink と入力

top - IMP(top.vhd)を右クリックしてメニューを出し、[Add Copy of Source...]を選択してください。

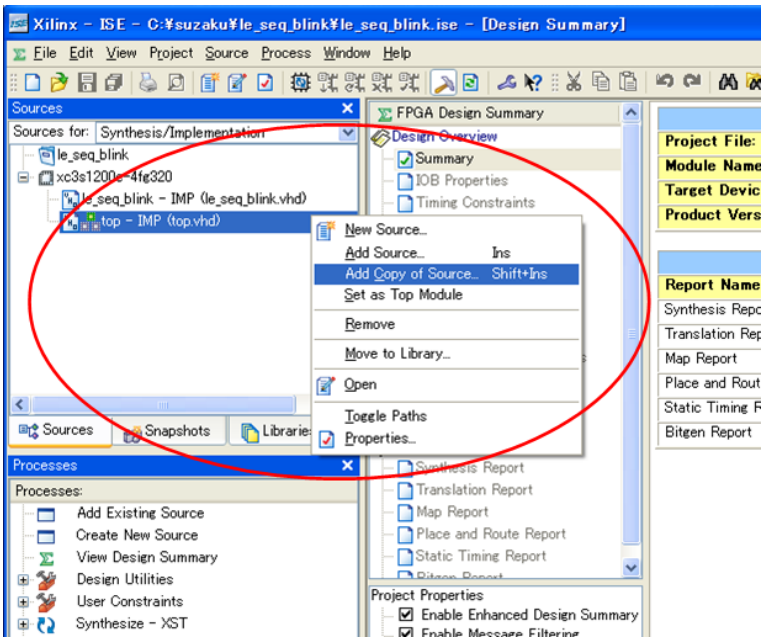


図 9.4. 既存のソースファイル追加

先ほどシミュレーションで作った `slot_counter.vhd` を選択してください。

下図が表示されるので、[OK]をクリックしてください。プロジェクトに `slot_counter.vhd` が追加されます。

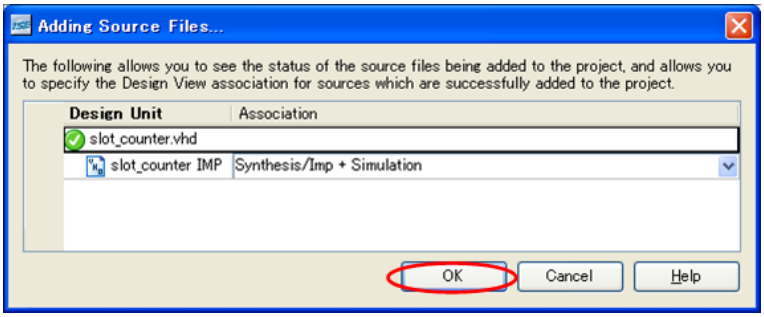


図 9.5. 既存のソースファイル追加時の確認

9.1.2.1. le_seq_blink.vhd

単色 LED を順次点灯させる回路を記述します。記述できたら保存して、le_seq_blink-IMP(le_seq_blink.vhd)を選択し、Check Syntax をダブルクリックして、文法チェックをしてください。

例 9.1. 単色 LED 順次点灯(le_seq_blink.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity le_seq_blink is
  Port (
```

```
    SYS_CLK   : in  STD_LOGIC;      --クロック信号
    SYS_RST   : in  STD_LOGIC;      --リセット信号
    le_timing : in  STD_LOGIC;      --単色 LED 順次点灯のタイミング信号
    le        : out STD_LOGIC_VECTOR(0 to 3) --単色 LED 出力信号
  );
end le_seq_blink;

architecture IMP of le_seq_blink is
--内部信号の定義
  signal le_w       : STD_LOGIC_VECTOR(0 to 3); --単色 LED 内部信号
  signal le_tim     : STD_LOGIC; --単色 LED 順次点灯のタイミング内部信号
  signal le_tim_reg : STD_LOGIC; --単色 LED 順次点灯タイミング信号の 1 クロック前の値
begin

  process(SYS_CLK) --クロック信号に変化があると実行
  begin
    if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
      if SYS_RST = '1' then --リセットされたら(同期リセット)
        le_tim_reg <= '0'; --初期化
      else
        le_tim_reg <= le_timing; --1 クロック前の値を保持
      end if;
    end if;
  end process;

  le_tim <= le_timing and (not le_tim_reg); --エッジ検出

  process(SYS_CLK) --クロック信号に変化があると実行
  begin
    if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
      if SYS_RST = '1' then --リセットされたら(同期リセット)
        le_w <= "0001"; --はじめに D1 を光らせる
      else
        if le_tim = '1' then --タイミング信号の値が'1'になったら
          le_w <= le_w(1 to 3) & le_w(0); --1bit 左にシフト
        end if;
      end if;
    end if;
  end process;

  le <= le_w; --外部に出力

end IMP;
```

9.1.2.2. top.vhd

`top.vhd` を上位階層として slot_counter と le_seq_blink の回路を呼び出します。

カウンタのビット幅をシミュレーション用に一旦 4 ビットに設定します。実際のビット数でシミュレーションを行ってもいいのですが、LED が順次点灯する様子を見るのに、非常に長い時間かかります。今回はエッジ検出の様子とシフトレジスタの様子を確認したいだけなので 4 ビットにします。

記述できたら top-IMP(top.vhd)を選択し、Synthesize をダブルクリックして、文法チェックをしてください。

文法チェックが終わったら、top-IMP(top.vhd)の上で右クリックしメニューを出し、[Set as Top Module]を選択してください。

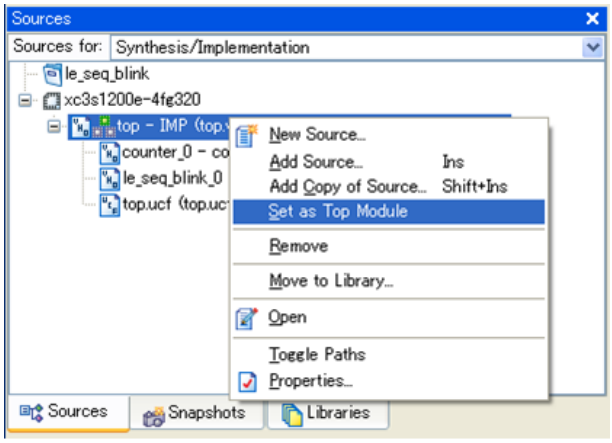


図 9.6. 上位階層に設定

例 9.2. 単色 LED 順次点灯(top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity top is
  generic (
    C_CNT_WIDTH : integer := 4 --カウンタのビット幅(シミュレーション用)
--    C_CNT_WIDTH : integer := 19 --カウンタのビット幅(SZ410 の時は 23)
  );
  Port (
    SYS_CLK    : in STD_LOGIC;  --クロック信号
    SYS_RST    : in STD_LOGIC;  --リセット信号
    nLE : out  STD_LOGIC_VECTOR(0 to 3) --単色 LED 出力信号(負論理)
  );
end top;

architecture IMP of top is
  signal count : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1);
  signal le    : STD_LOGIC_VECTOR(0 to 3); --単色 LED 内部信号

  component slot_counter
    generic (
      C_CNT_WIDTH : integer := C_CNT_WIDTH --カウンタのビット幅
    );
    Port (
      SYS_CLK : in STD_LOGIC;  --クロック信号
      SYS_RST : in STD_LOGIC;  --リセット信号
      count : out STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1) --カウンタ値
    );
  end component;

  component le_seq_blink
    Port (
```

```
      SYS_CLK   : in STD_LOGIC;  --クロック信号
      SYS_RST   : in STD_LOGIC;  --リセット信号
      le_timing : in  STD_LOGIC; --単色 LED 順次点灯のタイミング信号
      le        : out  STD_LOGIC_VECTOR(0 to 3) --単色 LED 出力信号
    );
  end component;

begin

  slot_counter_0 : slot_counter
    Port map(
      SYS_CLK => SYS_CLK,
      SYS_RST => SYS_RST,
      count   => count
    );

  le_seq_blink_0 : le_seq_blink
    Port map(
      SYS_CLK   => SYS_CLK,
      SYS_RST   => SYS_RST,
      le_timing => count(0), --カウンタの最上位ビットを接続
      le        => le
    );

  nLE <= not le; --外部に出力

end IMP;
```

9.1.2.3. コンポーネント文について

上位のメイン回路から下位回路を呼び出すためには、エンティティ文の中でコンポーネントとして定義します。

例 9.3. component 文

```
  component コンポーネント名
    Port (
      信号名 : 入出力方向 データタイプ
    );
  end component;
```

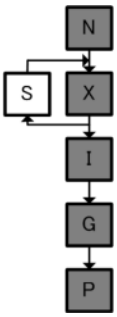
コンポーネントの定義が終わったら、アーキテクチャ文の begin の下で呼び出します。

下位回路のポートと信号は port map で結合します。ラベル名は、このコンポーネントにつけられる名前で、そのアーキテクチャ内でユニークな名前でなければいけません。

例 9.4. port map 文

```
ラベル名 : コンポーネント名
  Port map (
    ポート名=>信号名
  );
```

9.1.3. シミュレーション



単色 LED の順次点灯のシミュレーションを行います。シミュレーションの詳細は「8.7. ISE Simulator の使い方」を参照してください。

[Project]→[New Source]でテストベンチ(Test Bench WaveForm)を新規作成してください。ここではファイル名を le_seq_blink_tb とします。上位階層のファイルを聞かれるで、"top"を選択してください。また、[Initial Length of Test Bench]は今回も 10000ns に変更してください。

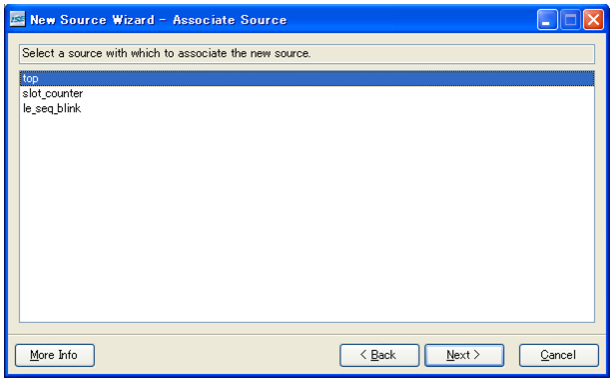


図 9.7. 上位階層選択

SYS_RST の信号を前回と同様(100ns で立ち上げ、500ns 立ち下げ)に生成して保存してください。

Sources ウィンドウの[Sources for:]を[Befavioral Simulation]に変更し、le_seq_blink_tb を選択し、Process タブをクリックし、[Simulate Behavioral Model]をダブルクリックしてください。シミュレーションが実行されます。

このままでは確認したい波形を全てみれていないので、他の信号も追加します。Sources ウィンドウの Sim Instances タブをクリックし、le_seg_blink_0 を選択してください。選択できたら、Processes ウィンドウの SimObjects タブをクリックしてください。信号がいくつか表示されますが、このうち le_tim や le_tim_reg、le_timing、le を追加してください。追加できたらシミュレーション時間を 10000ns に変更し、[Simulation]→[Restart]、[Simulation]→[Run for Specified Time]をクリックしてください。

エッジ検出やシフトの様子、LED の順次点灯の様子を確認してください。

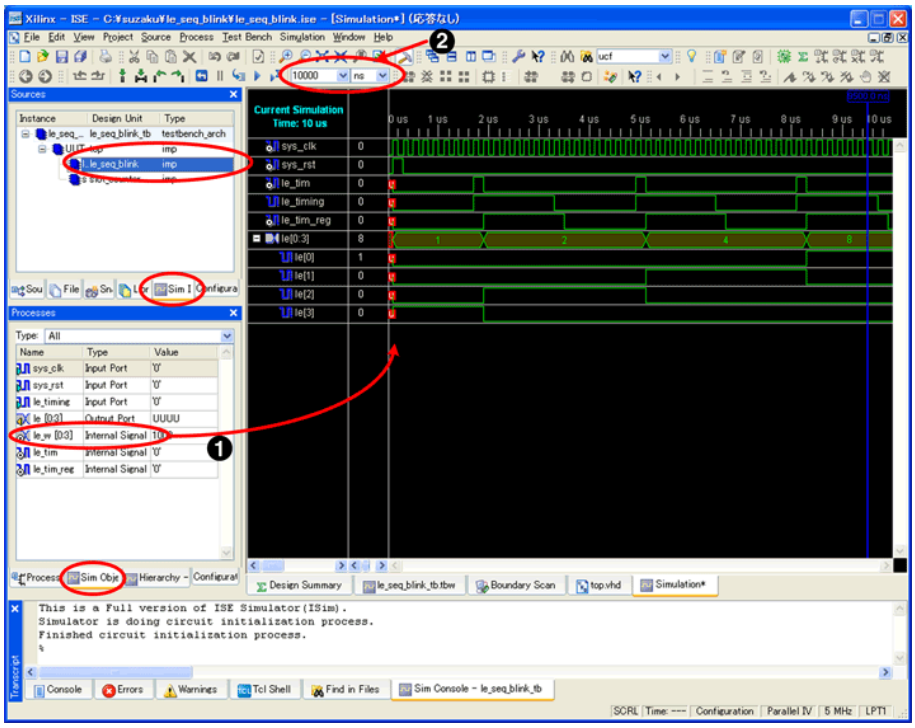


図 9.8. 見たい信号を追加

❶　見たい信号をドラッグ&ドロップで追加

❷　10000ns に変更

9.1.3.1. タイミング信号生成(エッジ検出)について

カウンタの最上位ビットの前回の値を保持し、その値と今回の最上位ビットの値が違ったならば信号を出力します。

例 9.5. エッジ検出

```
process(SYS_CLK)
begin
  if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
    if SYS_RST = '1' then --リセットされたら(同期リセット)
      le_tim_reg <= '0'; --初期化
    else
      le_tim_reg <= le_timing; --1 クロック前の値を保持
    end if;
  end if;
end process;

le_tim <= le_timing and (not le_tim_reg); --エッジ検出
```

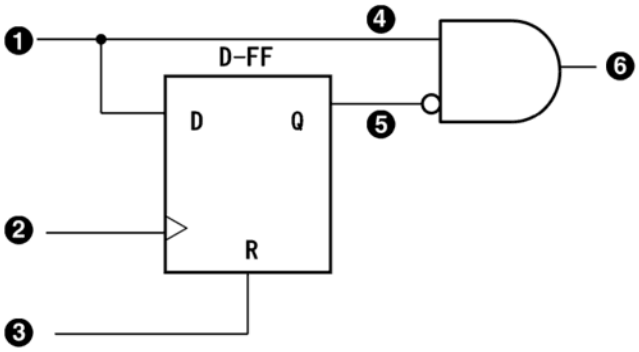


図 9.9. エッジ検出回路

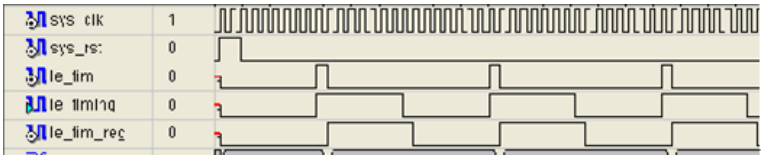


図 9.10. エッジ検出の波形

❶ 入力

❷ CLK クロック

❸ RST クロック

❹ 今回のデータ

❺ 前回のデータ

❻ 出力

count(0)	count(1)	count(2)	count(3)
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

図 9.11. 最上位ビットの動作

最上位ビットに注目してみて下さい。カウンタは最大値までカウントしたら 0 にもどってカウントし続けます。

9.1.3.2. シフトレジスタについて

シフトレジスタは、記憶しているデータの桁を左右にシフトさせることができるレジスタです。左にシフトするには以下の記述をします。

例 9.6. シフトレジスタ

```
process(SYS_CLK) --クロック信号に変化があると実行
begin
  if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
    if SYS_RST = '1' then --リセットされたら(同期リセット)
      le <= "0001";
    else
      if le_tim = '1' then --タイミング信号の値が'1'になったら
        le <= le(1 to 3) & le(0); --1bit 左にシフト
      end if;
    end if;
  end if;
end process;
```

9.1.3.3. &について

&を使うと bit を連結することができます。

例 9.7. bit 連結

```
le <= le(1 to 3) & le(0);
```

(1 to 3) で、1 ビット目から 3 ビット目までを切り出すことができます。(to で幅を設定している場合は to、downto で幅を設定している場合は downto で切り出す)イベントが発生するたびに最上位ビットを最下位に連結させることにより、"1"の値を順に左にシフトさせます。

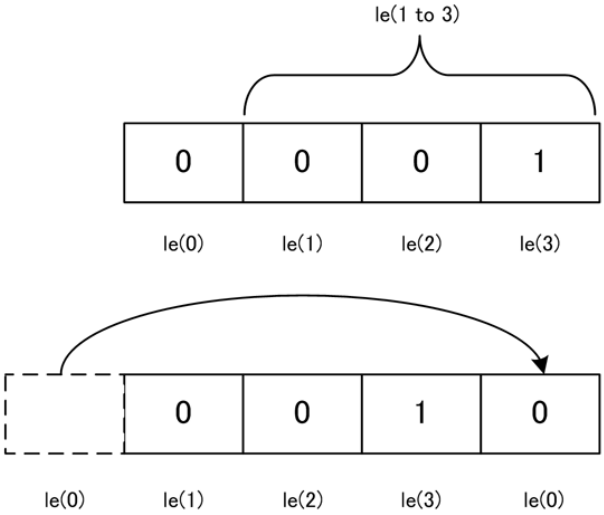


図 9.12. bit 連結

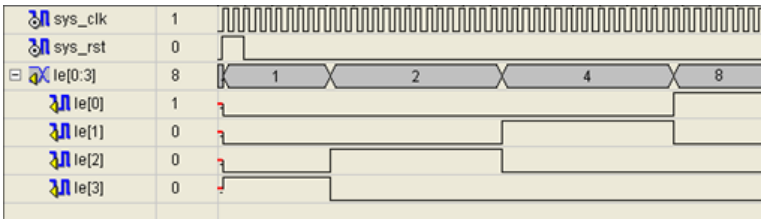


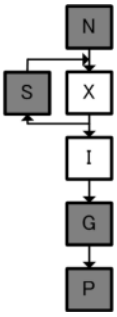
図 9.13. シフトレジスタの波形

9.1.4. 再度論理合成

シミュレーション用に 4 ビットにしていた"top.vhd"のカウンタのビット幅を 19 ビットにし、Synthesize をダブルクリックして文法チェックしてください。

9.1.5. インプリメンテーション

単色 LED への出力信号を STD_LOGIC_VECTOR(0 to n)で定義しました。to で定義すると MSB 側が 0 になります。この場合信号を入出力する前に、信号の MSB と LSB をひっくり返さなければいけません。VHDL のソースでひっくり返してもいいのですが、最後にピンアサインでひっくり返します。例えば nLE0 は nLE<3>、nLE1 は nLE<2>、nLE2 は nLE<1>、nLE3 は nLE<0>にピンアサインします。



ピンアサインができたら、Implement Design をダブルクリックしてください。

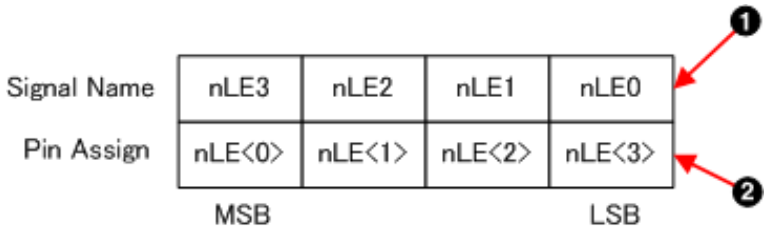


図 9.14. ピンアサインでひっくり返す

❶　ピンアサインでひっくり返す

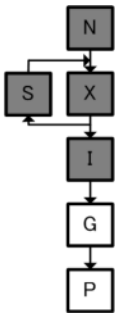
❷　バスの定義は MSB 側がビット 0

表 9.1. 単色 LED 順次点灯ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
SYS_CLK	T9	U10	C8	Y6
SYS_RST	F5	D3	A8	U3
nLE<0>	E11	A11	L13	E1
nLE<1>	D11	B11	L14	F1
nLE<2>	C12	F12	L15	F2
nLE<3>	B12	E12	L16	G2

9.1.6. プログラムファイル作成、コンフィギュレーション

Generate Programming File をダブルクリックし、bit ファイルを作ってください。
iMPACT を立ち上げ、コンフィギュレーションしてください。



単色 LED が D1→D2→D3→D4→D1→・・・・と順次点灯するのを確認してください。

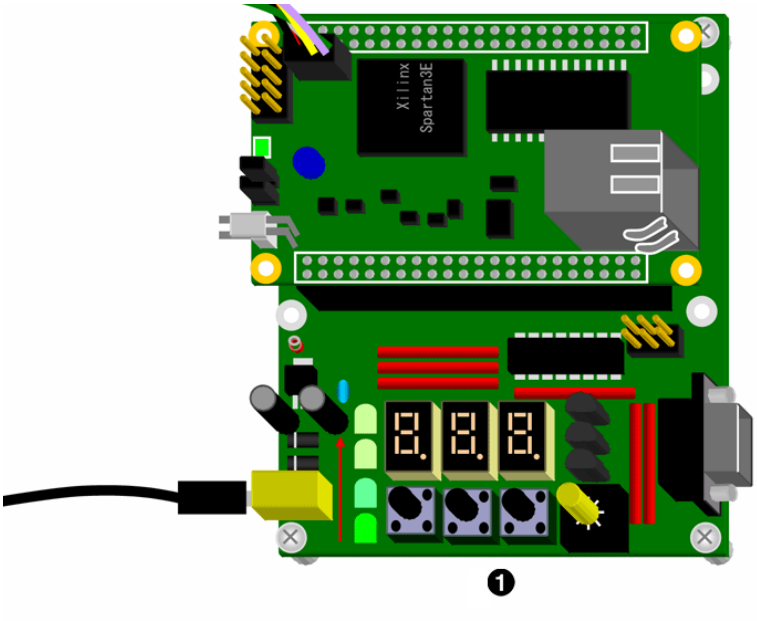


図 9.15. 単色 LED 順次点灯

❶　D1→D2→D3→D4→D1→…と順次点灯

9.1.7. バスのビットラベルについて

MicroBlaze、PowerPC はバスアーキテクチャとして IBM の CoreConnect を採用しています。CoreConnect のバスおよびレジスタビットの命名規則で MSB 側がビット 0 に定義されています。このため、LSB 側がビット 0 に定義されている外部デバイスと比べビットラベルが逆になります。LED/SW ボードも一般の外部デバイスと同じく、LSB 側をビット 0 に定義しています。

本書内の VHDL ソースコード内でバスの定義を行う場合、IBM の CoreConnect に合わせて MSB 側をビット 0 にしています。これを外部デバイスと接続するために、FPGA のピンアサインの設定で MSB と LSB をひっくり返しています。

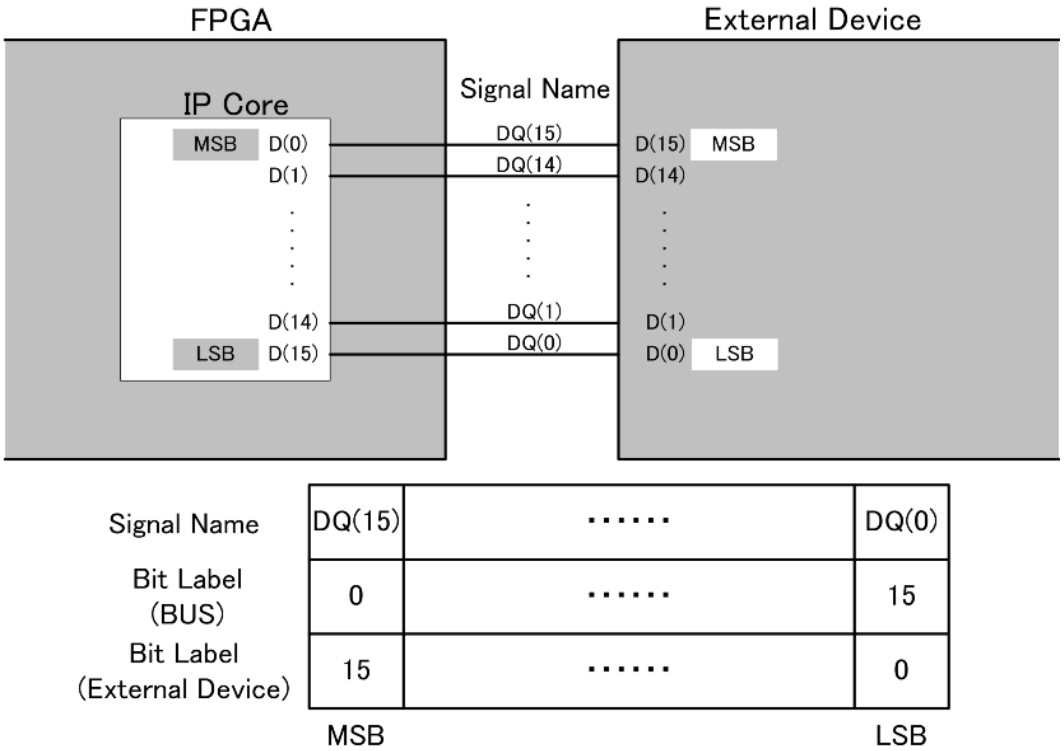


図 9.16. CoreConnect のビットラベルと信号

9.2. 7 セグメント LED デコーダ

7 セグメント LED に数字を表示させて回転させます。まずは数字を表示するために 7 セグメント LED のデコーダを作ります。デコーダを作っただけでは数字が表示できているかどうか分からないので、ここではロータリコードスイッチからの入力を 7 セグメント LED に表示する回路を作ります。

9.2.1. ロータリコードスイッチ周辺回路

LED/SW ボードに実装されているロータリコードスイッチは 4 ビットで 0 ～ F までの数字を表現できます。それぞれ 1kΩ の抵抗で 3.3V にプルアップされています。負論理なので内部で正論理にして使います。正論理にした場合のそれぞれの"High"(" 1 ")、"Low"("0")は"表 10-1 ロータリコードスイッチ(正論理)"のようになります。

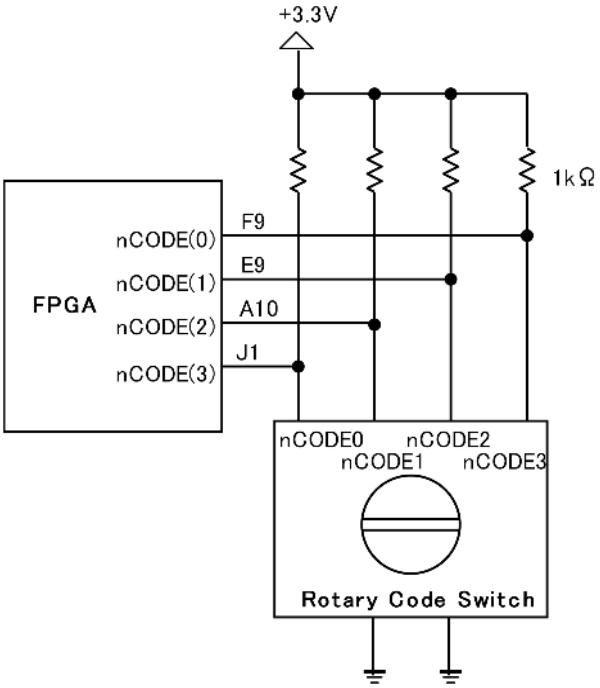


図 9.17. ロータリコードスイッチ周辺回路とピンアサイン

表 9.2. ロータリコードスイッチ(正論理)

数字	CODE3	CODE2	CODE1	CODE0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
b	1	0	1	1
C	1	1	0	0
d	1	1	0	1
E	1	1	1	0
F	1	1	1	1

9.2.2. 7 セグメント LED 周辺回路

7 セグメント LED のセグメントは下図のように配置されていて、A~G までの各発光ダイオードの適当なものだけを光らすと数字を表示することができます。「表 9.3. 7 セグメント LED デコーダ(正論理)」と照らし合わせてどう光らせれば数字になるか確認してみてください。

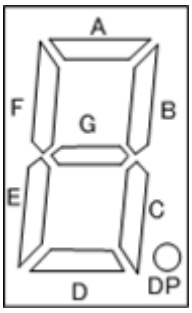


図 9.18. セグメントの配置

表 9.3. 7 セグメント LED デコーダ(正論理)

数字	DP(SEG7)	G(SEG6)	F(SEG5)	E(SEG4)	D(SEG3)	C(SEG2)	B(SEG1)	A(SEG0)
0	0	0	1	1	1	1	1	1
1	0	0	0	0	0	1	1	0
2	0	1	0	1	1	0	1	1
3	0	1	0	0	1	1	1	1
4	0	1	1	0	0	1	1	0
5	0	1	1	0	1	1	0	1
6	0	1	1	1	1	1	0	1
7	0	0	1	0	0	1	1	1
8	0	1	1	1	1	1	1	1
9	0	1	1	0	1	1	1	1
A	0	1	1	1	0	1	1	1
B	0	1	1	1	1	1	0	0
C	0	0	1	1	1	0	0	1
D	0	1	0	1	1	1	1	0
E	0	1	1	1	1	0	0	1
F	0	1	1	1	0	0	0	1

LED/SW ボードには 7 セグメント LED が 3 つ実装されていて、Q1 に"Low"を入力すると LED 1、Q2 に"Low"を入力すると LED2、Q3 に"Low"を入力すると LED3 を扱うことができます。Q1、Q2、Q3 を同時に"Low"にすることで、全部を光らすこともできますが、同じ数字しか表示することはできません。異なる数字を表示したいときはダイナミック点灯という手法を用います。(「9.3. ダイナミック点灯」 参照)

Q1 ～ Q3 のセレクト信号は負論理、7 セグメント LED は正論理です。7 セグメント LED が正論理なのは電流を増やすために、バッファとしてインバータが 7 セグメント LED の前に実装されているためです。

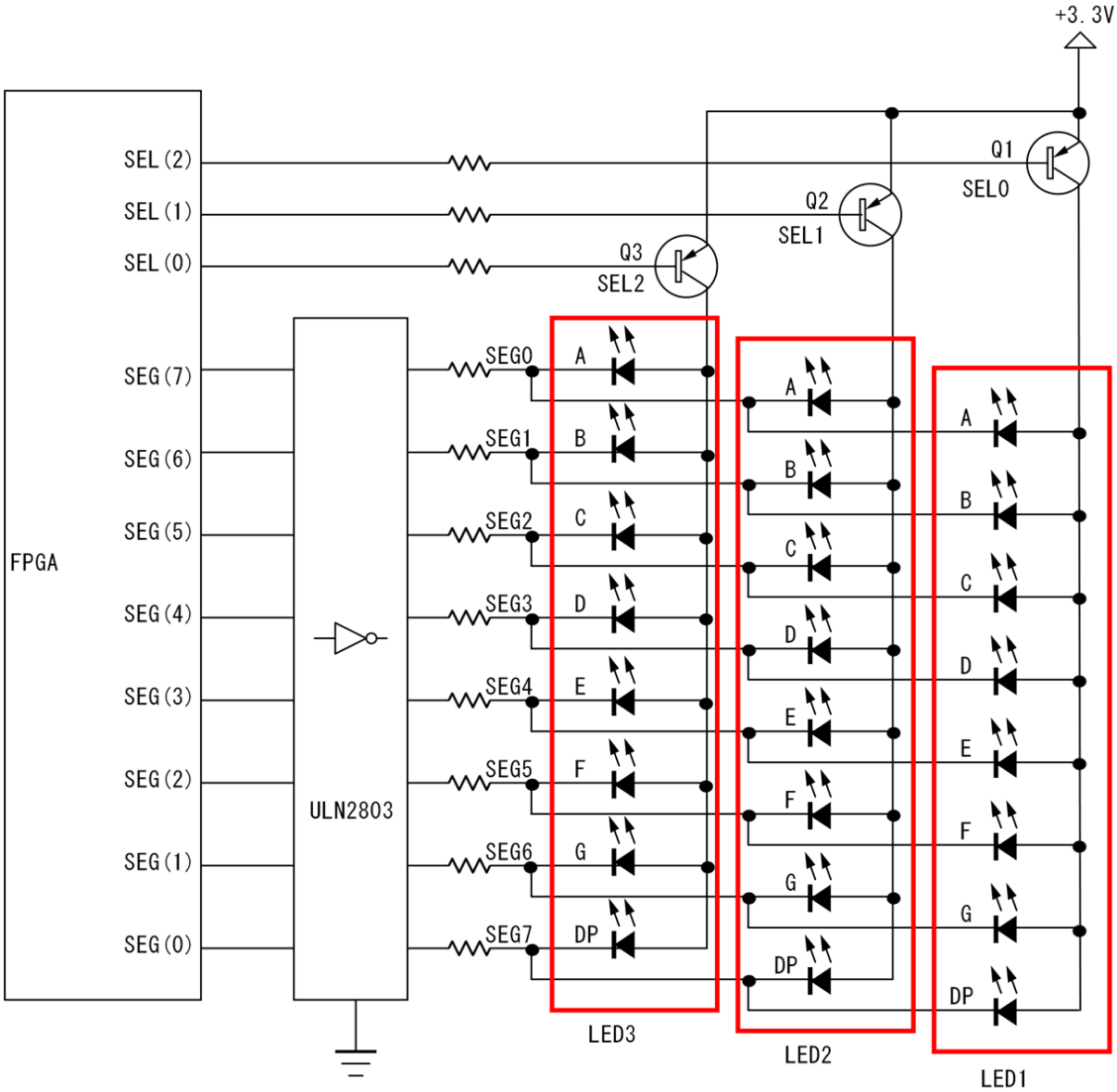
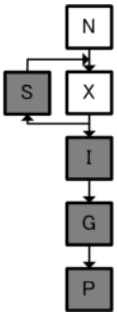


図 9.19. 7 セグメント LED 周辺回路

9.2.3. プロジェクト新規作成、論理合成

プロジェクトを新規作成してください。
プロジェクト名は seg7_decoder とし、new Source で top.vhd を作ってください。
top - IMP(top.vhd)を右クリックしてメニューを出し、[New Source...]を選択し、
`seg7_decoder.vhd` を新規作成してください。



9.2.3.1. seg7_decoder.vhd

7 セグメント LED のデコーダ回路を記述してください。記述できたら、seg7_decoder-IMP(seg7_decoder.vhd)を選択し、Check Syntax をダブルクリックして、文法チェックをしてください。

例 9.8. 7 セグメント LED デコーダ(seg7_decoder.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity seg7_decoder is
  Port (
    SEG : out STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
    seg_data : in STD_LOGIC_VECTOR(0 to 3) --4bit バイナリコード
  );
end seg7_decoder;

architecture IMP of seg7_decoder is
begin
  --デコーダ記述
  process(seg_data)
  begin
    case seg_data is
      when "0000" => SEG <= "00111111"; --0
      when "0001" => SEG <= "00000110"; --1
      when "0010" => SEG <= "01011011"; --2
      when "0011" => SEG <= "01001111"; --3
      when "0100" => SEG <= "01100110"; --4
      when "0101" => SEG <= "01101101"; --5
      when "0110" => SEG <= "01111101"; --6
      when "0111" => SEG <= "00100111"; --7
      when "1000" => SEG <= "01111111"; --8
      when "1001" => SEG <= "01101111"; --9
      when "1010" => SEG <= "01110111"; --A
      when "1011" => SEG <= "01111100"; --b
      when "1100" => SEG <= "00111001"; --C
      when "1101" => SEG <= "01011110"; --d
      when "1110" => SEG <= "01111001"; --E
      when "1111" => SEG <= "01110001"; --F
      when others => SEG <= "XXXXXXXX"; --0,1 以外(X,Z,U 等)の場合
    end case;
  end process;
end IMP;
```

9.2.3.2. case 文について

case 文は次の書式で記述します。「例 9.8. 7 セグメント LED デコーダ(seg7_decoder.vhd)」をみると `when others => SEG <= "XXXXXXXX"`という記述があります。std_logic_vector は'0'、'1'の他に'X'や'Y'、'U'などの値を持っているため、残り全てを記述するために"XXXXXXXX"(出力不定)としています。

例 9.9. case 文

```
case 式 is
  when 値 => 順次処理文
  when others => 順次処理文
end case;
```

9.2.3.3. top.vhd

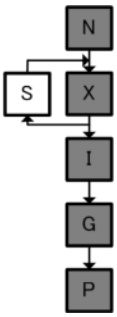
`top.vhd` を上位階層として seg7_decoder の回路を呼び出します。記述できたら top-IMP(top.vhd) を選択し、Synthesize をダブルクリックして、文法チェックをしてください。

文法チェックが終わったら、top-IMP(top.vhd)の上で右クリックしメニューを出し、[Set as Top Module]を選択し、`top.vhd` を上位階層としてください。

例 9.10. 7 セグメント LED デコーダ(top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity top is
  Port (
    nCODE : in STD_LOGIC_VECTOR(0 to 3); --ロータリコードスイッチからの入力信号(負論理)
    SEG : out STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号(正論理)
    nSEL : out STD_LOGIC_VECTOR(0 to 2) --7 セグメント LED セレクト信号(負論理)
    );
  end top;

architecture IMP of top is
  signal code : STD_LOGIC_VECTOR(0 to 3); --ロータリコードスイッチ内部信号(正論理)
  signal sel : STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト内部信号(正論理)

  component seg7_decoder
    Port (
      SEG : out STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
      seg_data : in STD_LOGIC_VECTOR(0 to 3) --4bit バイナリコード
    );
  end component;

begin
  seg7_decoder_0 : seg7_decoder
    Port map(
      SEG => SEG,
      seg_data => code
    );
  sel <= "001"; --7 セグメント LED1 を点灯させる
  nSEL <= not sel; --負論理にして出力
  code <= not nCODE; --正論理にして入力

end IMP;
```

9.2.4. シミュレーション



デコーダのシミュレーションを行います。シミュレーションの詳細は「8.7. ISE Simulator の使い方」を参照してください。

[Project]→[New Source]でテストベンチ(Test Bench WaveForm)を新規作成してください。ここではファイル名を seg7_decoder_tb とします。上位階層のファイルを聞かれるで、"top"を選択してください。、[Initial Length of Test Bench]は 2000ns に変更してください。

ロータリコードスイッチの値を設定します。nCODE の波形の上で左クリックすると Set Value というウィンドウが出てきます。[Pattern Wizard]をクリックして下さい。Pattern Wizard が立ち上がります。Pattern Type を[Count Down]、Number of Cycle を[16]、Initial Value を[15]、Terminal Value を[0]、Decrement By を[1]、Count Every を[1]にして[OK]をクリックして下さい。15 から 0 までの波形が自動で生成されます。うまく生成されたら保存をして下さい。

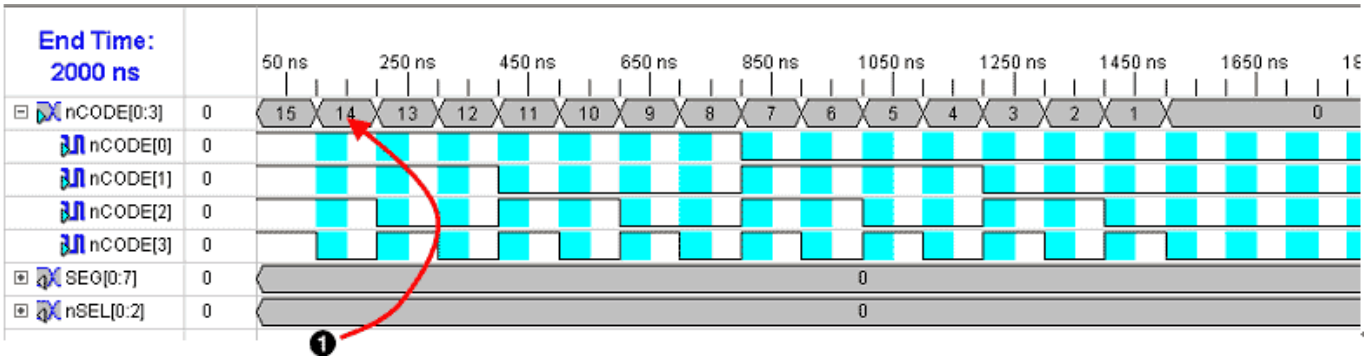


図 9.20. 波形生成

❶　左クリック

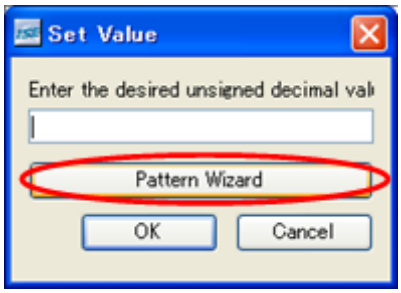


図 9.21. Set Value

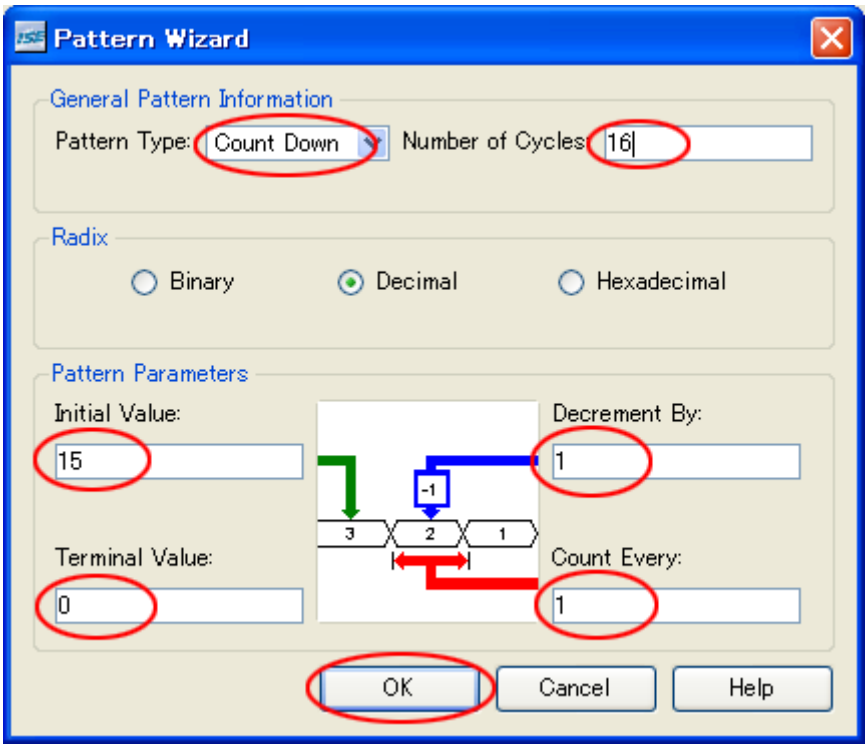


図 9.22. Pattern Wizard

Sources ウィンドウの[Sources for:]を[Befavioral Simulation]に変更し、seg7_decoder_tb を選択し、Process タブをクリックし、[Simulate Behavioral Model]をダブルクリックしてください。シミュレーションが実行されます。

ロータリコードスイッチ nCODE は負論理の信号で `top.vhd` で正論理の信号 code にしています。負論理のままでは何がデコードされているのか分かりにくいので、正論理の信号 code を追加します。追加できたら[Simulation]→[Restart]、[Simulation]→[Run All]をクリックしてください。

ロータリコードスイッチの値がきちんとデコードされて 7 セグメント LED に渡されているのか確認します。それぞれ値が Decimal で表示されています。このままではデコードされた結果が何か分からないので、波形の上で右クリックし、表示を Hexadecimal に変更してください。

「表 9.2. ロータリコードスイッチ(正論理)」、「表 9.3. 7 セグメント LED デコーダ(正論理)」を参照してデコードされた結果が正しいか確認してください。

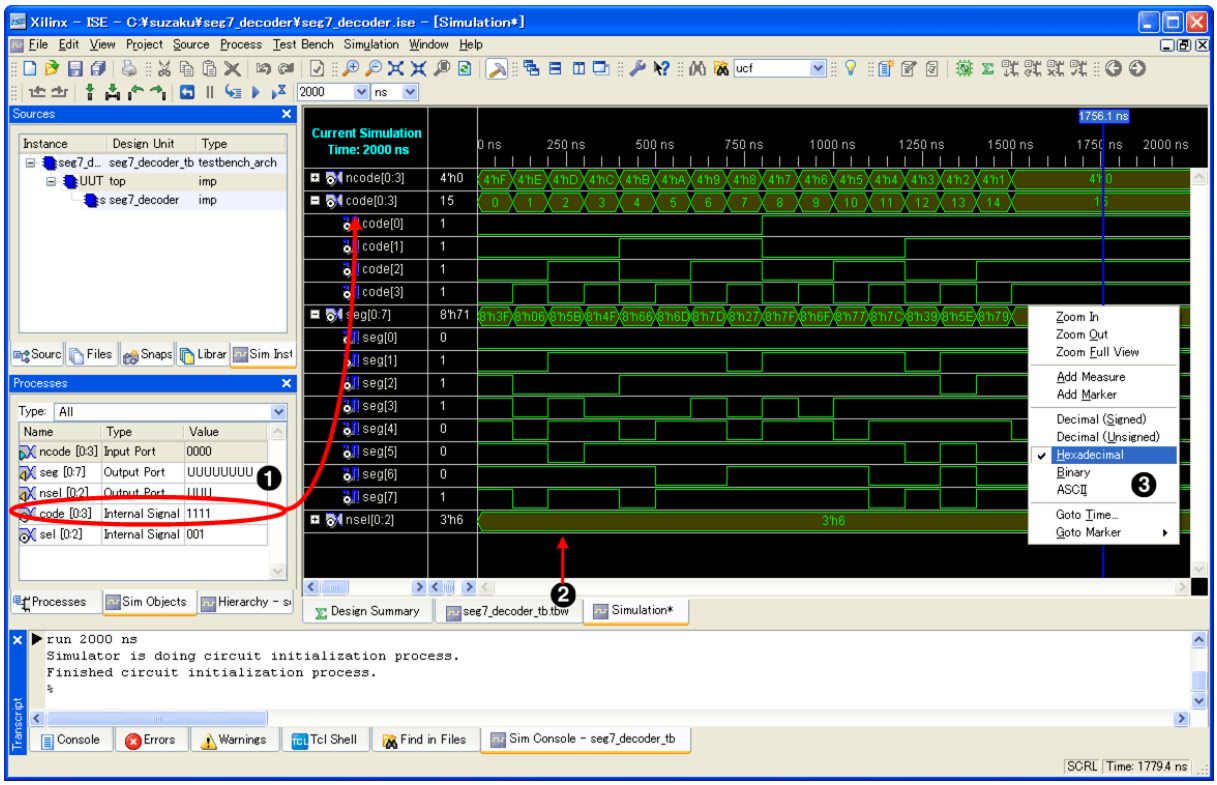


図 9.23. デコーダシミュレーション結果

❶　code[0:3]を追加

❷　code が 0 の時 seg は 00111111

❸　右クリックして Hexadecimal に変更

9.2.5. インプリメンテーション

信号を STD_LOGIC_VECTOR(0 to n)で定義しているので、MSB 側がビット 0 になります。信号は最後にピンアサインでひっくり返しています。例えば nCODE0 は nCODE<3>、nCODE1 は nCODE<2>、nCODE2 は nCODE<1>、nCODE3 は nCODE<0>にピンアサインします。ピンアサインができたら、Implement Design をダブルクリックしてください。

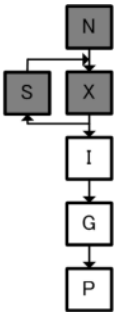


表 9.4. 7 セグメント LED デコーダ ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nCODE<0>	C8	J1	J16	H5
nCODE<1>	A9	F9	J15	E2
nCODE<2>	A12	E9	J14	D2
nCODE<3>	C10	A10	J13	U9
SEG<0>	C5	L5	F15	P1
SEG<1>	B5	L6	F16	P2
SEG<2>	E6	L4	G13	L2
SEG<3>	D6	L3	G14	M2
SEG<4>	C6	L2	G15	N2
SEG<5>	B6	L1	G16	N3
SEG<6>	A8	C9	N9	Y7
SEG<7>	B8	D9	P9	W7
nSEL<0>	D7	K6	H13	N5
nSEL<1>	C7	K4	H14	M3
nSEL<2>	B7	K3	H15	M4

9.2.6. プログラムファイル作成、コンフィギュレーション

Generate Programming File をダブルクリックし、bit ファイルを作ってください。

iMPACT を立ち上げ、コンフィギュレーションしてください。

ロータリコードスイッチをまわすと、対応する数字が 7 セグメント LED(LED1)に表示されます。

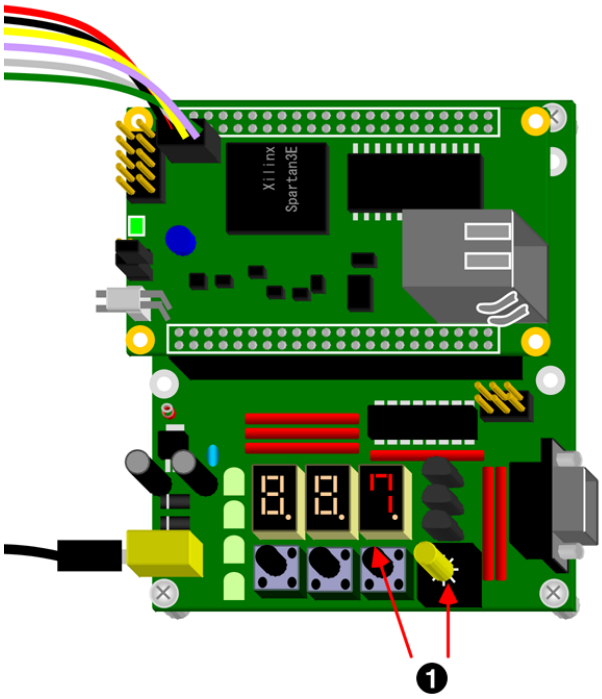


図 9.24. 7 セグメント LED デコーダ

❶ ロータリコードスイッチをまわすと対応する数字が LED1 に表示される

9.3. ダイナミック点灯

3 つの 7 セグメント LED をダイナミック点灯させます。

ダイナミック点灯とは配線の本数を減らすための手法です。複数の 7 セグメント LED に同じデータ線を接続しています。7 セグメント LED を順次点灯することにより、複数の 7 セグメント LED が同時に点灯しているように見せます。

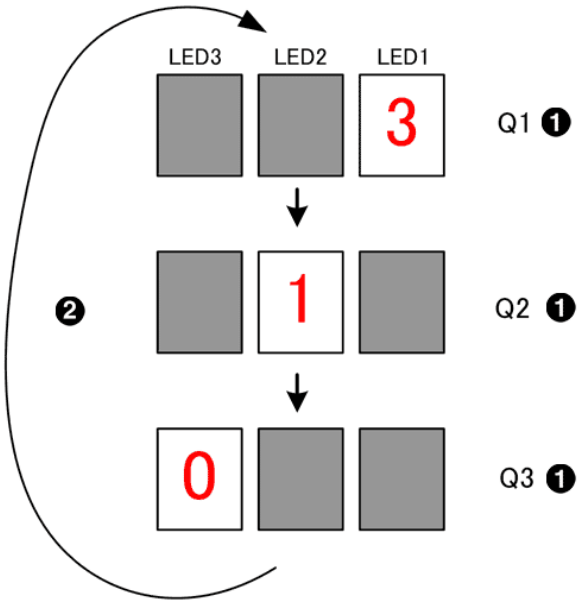


図 9.25. 7 セグメント LED ダイナミック点灯

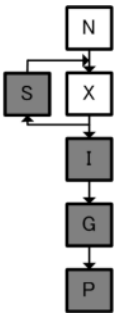
❶ 選択

❷ 繰り返す

9.3.1. 7 セグメント LED 周辺回路

ダイナミック点灯に必要となる 7 セグメント LED 周辺回路は「図 9.19. 7 セグメント LED 周辺回路」を参照してください。

9.3.2. プロジェクト新規作成、論理合成



プロジェクトを新規作成してください。

プロジェクト名は dynamic_ctrl とし、new Source で `top.vhd` を作ってください。

top - IMP(top.vhd)を右クリックしてメニューを出し、[New Source...]を選択し、`dynamic_ctrl.vhd` を新規作成してください。

top-IMP(top.vhd)を右クリックしてメニューを出し、[Add Copy of Source...]を選択し、`slot_counter.vhd, seg7_decoder.vhd` を追加してください。

top-IMP(top.vhd)の上で右クリックしメニューを出し、[Set as Top Module]を選択し、`top.vhd` を上位階層としてください。

9.3.2.1. dynamic_ctrl.vhd

ダイナミック点灯させる回路を記述してください。記述できたら、dynamic_ctrl-IMP(dynamic_ctrl.vhd)を選択し、Check Syntax をダブルクリックして、文法チェックをしてください。

例 9.11. ダイナミック点灯(dynamic_ctrl.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity dynamic_ctrl is
  Port (
    SYS_CLK     : in  STD_LOGIC;                --クロック信号
    SYS_RST     : in  STD_LOGIC;                --リセット信号
    nSEL        : out STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号(負論理)
    seg7_timing : in  STD_LOGIC;                --ダイナミック点灯タイミング信号
    seg_in1     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED1 の値
    seg_in2     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED2 の値
    seg_in3     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED3 の値
    seg_data    : out STD_LOGIC_VECTOR(0 to 3)  --4bit バイナリコード
  );
end dynamic_ctrl;

architecture IMP of dynamic_ctrl is

  signal sel      : STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号(正論理)
  signal seg7_tim : STD_LOGIC;                --ダイナミック点灯タイミング信号
  signal seg7_tim_reg : STD_LOGIC;            --1 クロック前の値

begin
  process(SYS_CLK)
  begin
    if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
      if SYS_RST = '1' then --リセットされたら(同期リセット)
        seg7_tim_reg <= '0'; --初期化
      else
        seg7_tim_reg <= seg7_timing; --値を保持
      end if;
    end if;
  end process;

  seg7_tim <= seg7_timing and (not seg7_tim_reg); --エッジ検出

  process(SYS_CLK) --クロック信号に変化があると実行
  begin
    if SYS_CLK'event and SYS_CLK = '1' then --クロックの立ち上がりに同期
      if SYS_RST = '1' then --リセットされたら(同期リセット)
        sel <= "001"; --はじめに 7 セグメント LED 1 を光らせる
      else
        if seg7_tim = '1' then --7 セグ用タイミング信号の値が'1'になったら
          sel <= sel(1 to 2) & sel(0); --1bit 左にシフト
        end if;
      end if;
    end if;
  end process;

  --セレクト信号により代入する数字を変え,外部に出力
  seg_data <= seg_in1 when sel = "001" else seg_in2 when sel = "010" else seg_in3;
  nSEL <= not sel; --負論理に直して出力

end IMP;
```

9.3.2.2. top.vhd

今回は約 1kHz で表示する 7 セグメント LED を切り替えます。そのためにカウンタの 8 ビット目のエッジを取ります。top.vhd を上位階層として slot_counter、seg7_decoder、dynamic_ctrl の回路を呼び出します。記述できたら top-IMP(top.vhd)を選択し、Synthesize をダブルクリックして、文法チェックをしてください。

例 9.12. ダイナミック点灯(top.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity top is
  generic (
    C_CNT_WIDTH : integer:= 19 --カウンタのビット幅 (SZ410 の場合は 23)
  );
  Port (
    SYS_CLK : in  STD_LOGIC;                --クロック信号
    SYS_RST : in  STD_LOGIC;                --リセット信号
    nSEL    : out STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号(負論理)
    SEG     : out STD_LOGIC_VECTOR(0 to 7)  --7 セグメント LED への出力信号(正論理)
  );
end top;

architecture IMP of top is
  signal seg_in1  : STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED1 の値
  signal seg_in2  : STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED2 の値
  signal seg_in3  : STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED3 の値
  signal seg_data : STD_LOGIC_VECTOR(0 to 3);  --4bit バイナリコード
  signal count    : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1);  --カウンタ値

  component slot_counter
    generic (
      C_CNT_WIDTH : integer := C_CNT_WIDTH  --カウンタのビット幅
    );
    Port (
      SYS_CLK : in STD_LOGIC;  --クロック信号
      SYS_RST : in STD_LOGIC;  --リセット信号
      count     : out STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1) --カウンタ値
    );
  end component;

  component dynamic_ctrl
    Port (
      SYS_CLK  : in  STD_LOGIC;  --クロック信号
      SYS_RST  : in  STD_LOGIC;  --リセット信号
      nSEL     : out STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号(負論理)
      seg7_timing : in  STD_LOGIC;  --ダイナミック点灯タイミング信号
      seg_in1  : in  STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED1 の値
      seg_in2  : in  STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED2 の値
      seg_in3  : in  STD_LOGIC_VECTOR(0 to 3);  --7 セグメント LED3 の値
      seg_data : out STD_LOGIC_VECTOR(0 to 3)  --4bit バイナリコード
    );
```

```
  end component;

  component seg7_decoder
    Port (
      SEG      : out STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
      seg_data : in  STD_LOGIC_VECTOR(0 to 3)  --4bit バイナリコード
    );
  end component;

begin
  slot_counter_0 : slot_counter
    Port map(
      SYS_CLK => SYS_CLK,
      SYS_RST => SYS_RST,
      count => count
    );

  dynamic_ctrl_0 : dynamic_ctrl
    Port map(
      SYS_CLK  => SYS_CLK,
      SYS_RST  => SYS_RST,
      nSEL     => nSEL,
      seg7_timing => count(8),  --8 ビット目
      seg_in1  => seg_in1,
      seg_in2  => seg_in2,
      seg_in3  => seg_in3,
      seg_data => seg_data
    );

  seg7_decoder_0 : seg7_decoder
    Port map(
      SEG      => SEG,
      seg_data => seg_data
    );

  seg_in1 <= "0000";  --0 を表示
  seg_in2 <= "0001";  --1 を表示
  seg_in3 <= "0011";  --3 を表示

end IMP;
```

9.3.3. シミュレーション

シミュレーションの説明はいたしませんので、各自シミュレーションしてみてください。シミュレーションの詳細は「8.7. ISE Simulator の使い方」を参照してください。下図のシミュレーション結果は、カウンタの値を 12 にしてシミュレーションを行った結果です。

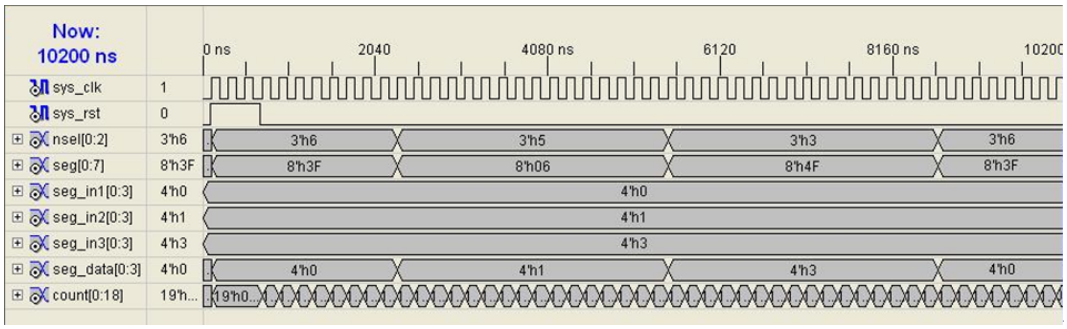


図 9.26. ダイナミック点灯シミュレーション結果

9.3.4. インプリメンテーション

「表 2.1. クロック、リセット信号 ピンアサイン」、「表 2.2. 機能用ピンアサイン(CON2)」を参照しピンアサインしてください。今回ここにはピンアサインを載せないので、各自考えてください。どうしても分からない場合は付属 CD-ROM の `"\suzaku-starter-kit\fpga"` の中の`"dynamic_ctrl.zip"`を展開し、その中にある`"top.ucf"`を参照してください。ピンアサインができたら、Implement Design をダブルクリックしてください。

9.3.5. プログラムファイル作成、コンフィギュレーション

Generate Programming File をダブルクリックし、bit ファイルを作ってください。

iMPACT を立ち上げ、コンフィギュレーションしてください。

7 セグメント LED に"3"、"1"、"0"と表示されます。他の数字も表示してみてください。

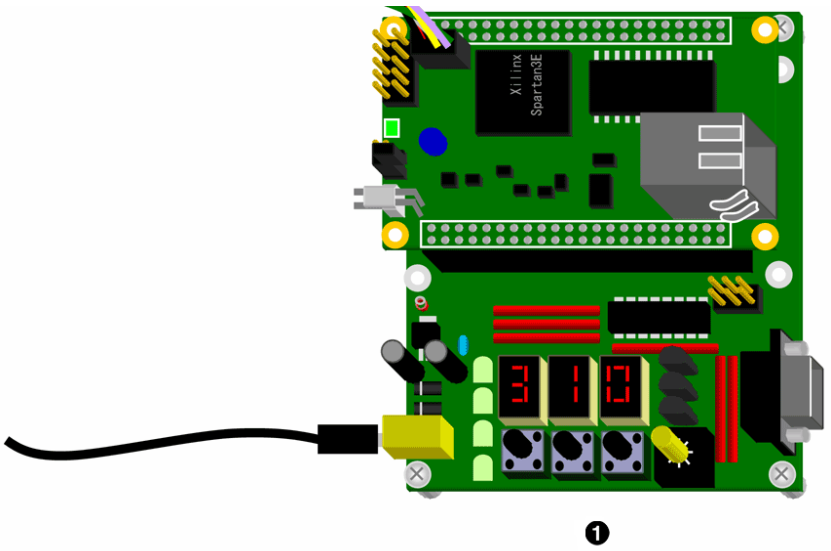


図 9.27. ダイナミック点灯

❶　ダイナミック点灯で数字が表示されます

10.EDK の使い方

SUZAKU のデフォルトは EDK(Embedded Development Kit)で構築されており、SUZAKU を使いこなすには EDK を使えなければいけません。EDK は Xilinx が提供する組み込み機器開発環境で、プロセッサや周辺ペリフェラルのソースコードやライブラリが登録されており、それらを GUI 環境下で構築、設定できます。ユーザで作った IP(Intellectual Property)コアも登録することができます。また、ソフトウェアのコンパイラやライブラリが登録されており、C 言語による開発も行うことができます。ISE の機能を取り込んでいるので、論理合成、マッピングを行うことができ、生成されたバイナリファイルを任意の BRAM にいれて、コンフィギュレーションファイルの生成を行うこともできます。

ここでは一旦スロットマシンと離れ、EDK の使い方を説明します。EDK の使い方の詳細は EDK のヘルプ等を参照してください。EDK には日本語のマニュアルも用意されています。

なお、本書では EDK において以下の手順で作業を行います。

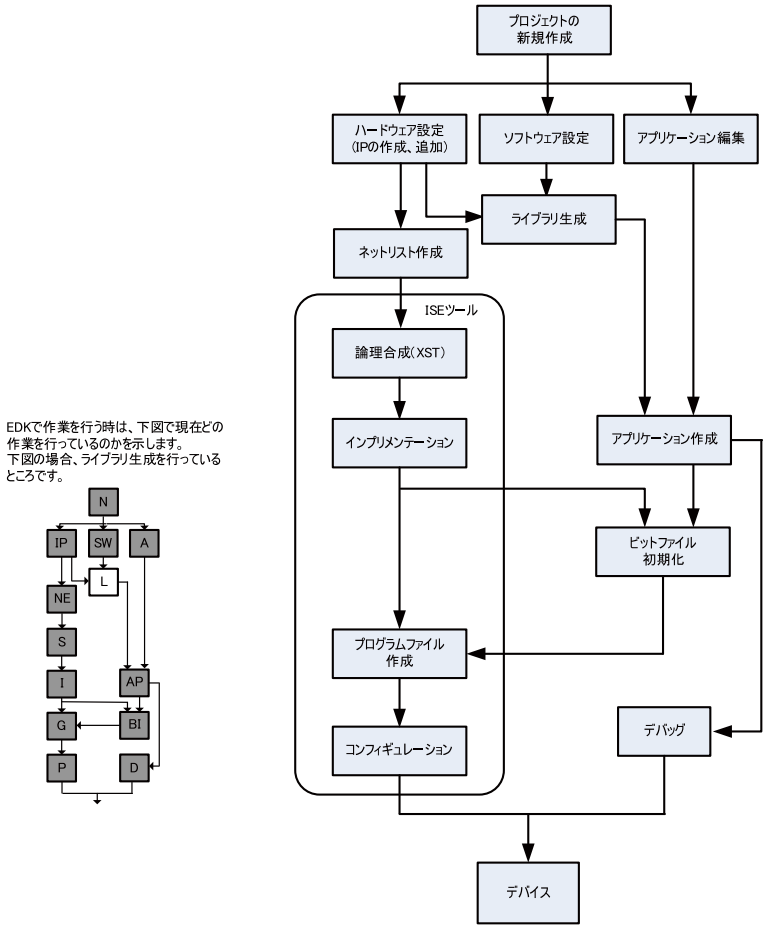


図 10.1. 本書での EDK 開発フロー

10.1. BSB ではじめての MicroBlaze & PowerPC

SZ010 SZ030 SZ130 SZ310 SZ410

まずは EDK に慣れるため、何もない状態からプロセッサが動くプロジェクトを作成します。EDK には BSB(Base System Builder)というウィザードが用意されています。BSB を用いることで、プロセッサが動くプロジェクトを簡単に作ることができます。ここでは BSB を使ってシリアル通信用ソフトウェアの画面に Hello SUZAKU と表示するプロジェクトを以下のような構成で作ります。

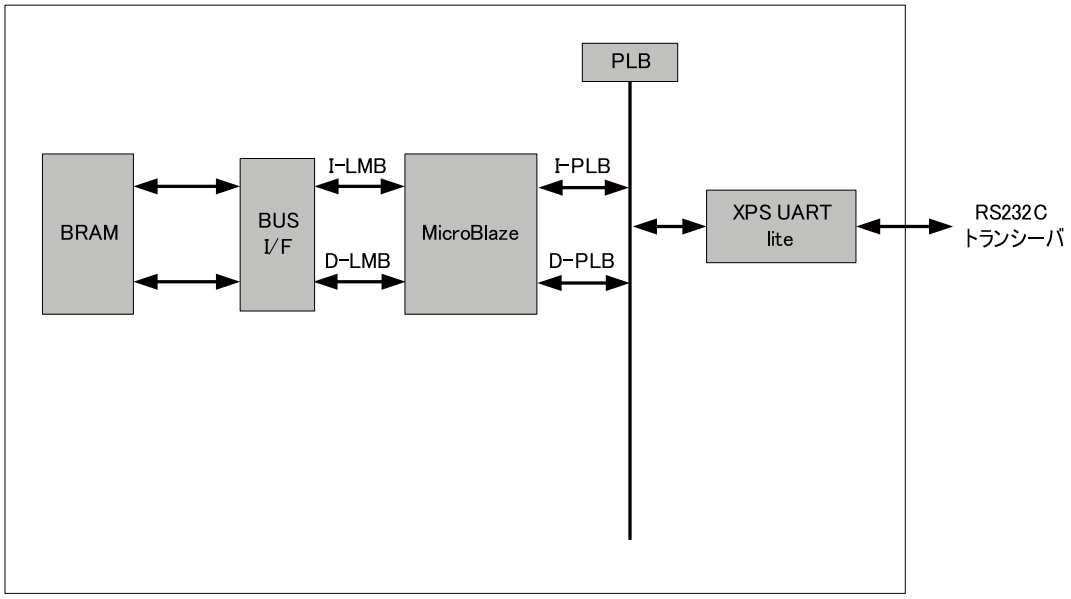


図 10.2. Hello SUZAKU プロジェクト(MicroBlaze)

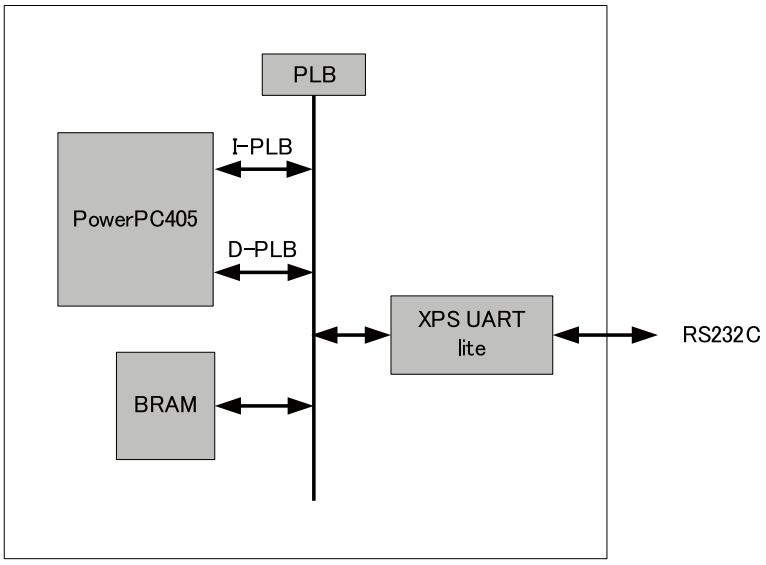
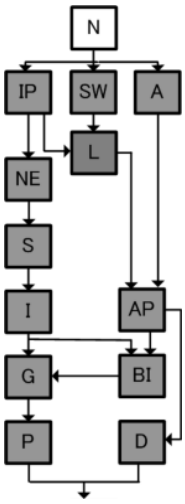


図 10.3. Hello SUZAKU プロジェクト(PowerPC)

10.1.1. BSB



Platform Studio を起動してください。Platform Studio は"EDK のインストールフォルダ\bin\nt_xps.exe"から起動できます。もしくは、[スタートメニュー]→[全てのプログラム]→[Xilinx ISE Design Suite X.X]→[EDK]→[Xilinx Platform Studio]から起動できます。
以下のような図が表示されるので、[Base System Builder wizard]を選択して[OK]をクリックして下さい。もし表示されなかった場合は[File]→[New Project...]をクリックして下さい。

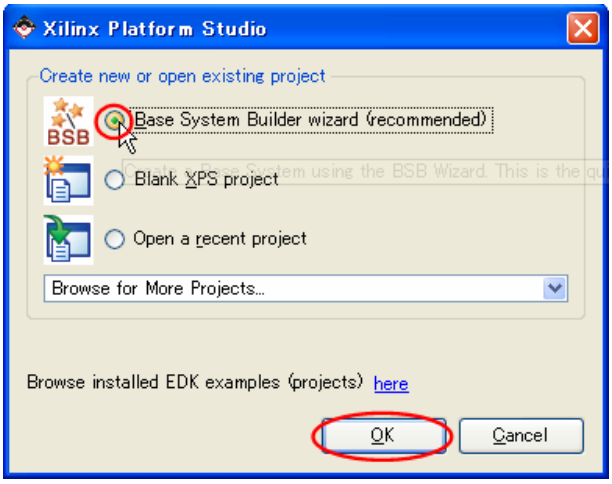


図 10.4. BSB 選択

プロジェクトファイルの保存場所を聞かれます。ここでは"C:\suzaku\sz***\system.xmp"(***は型式)とします。設定ができたら、[OK]をクリックして下さい。

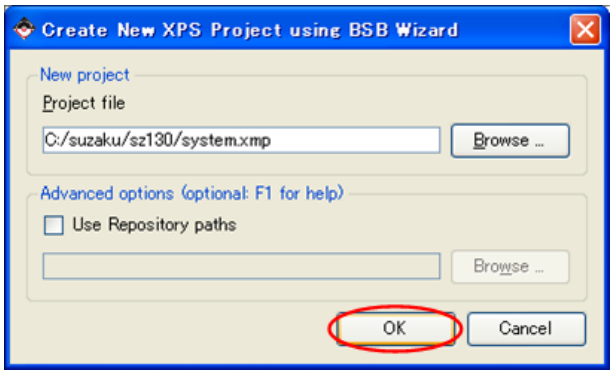


図 10.5. BSB ファイル保存

新しいデザインをはじめるので、[I would like to create a new design]を選択し、[Next]をクリックして下さい。

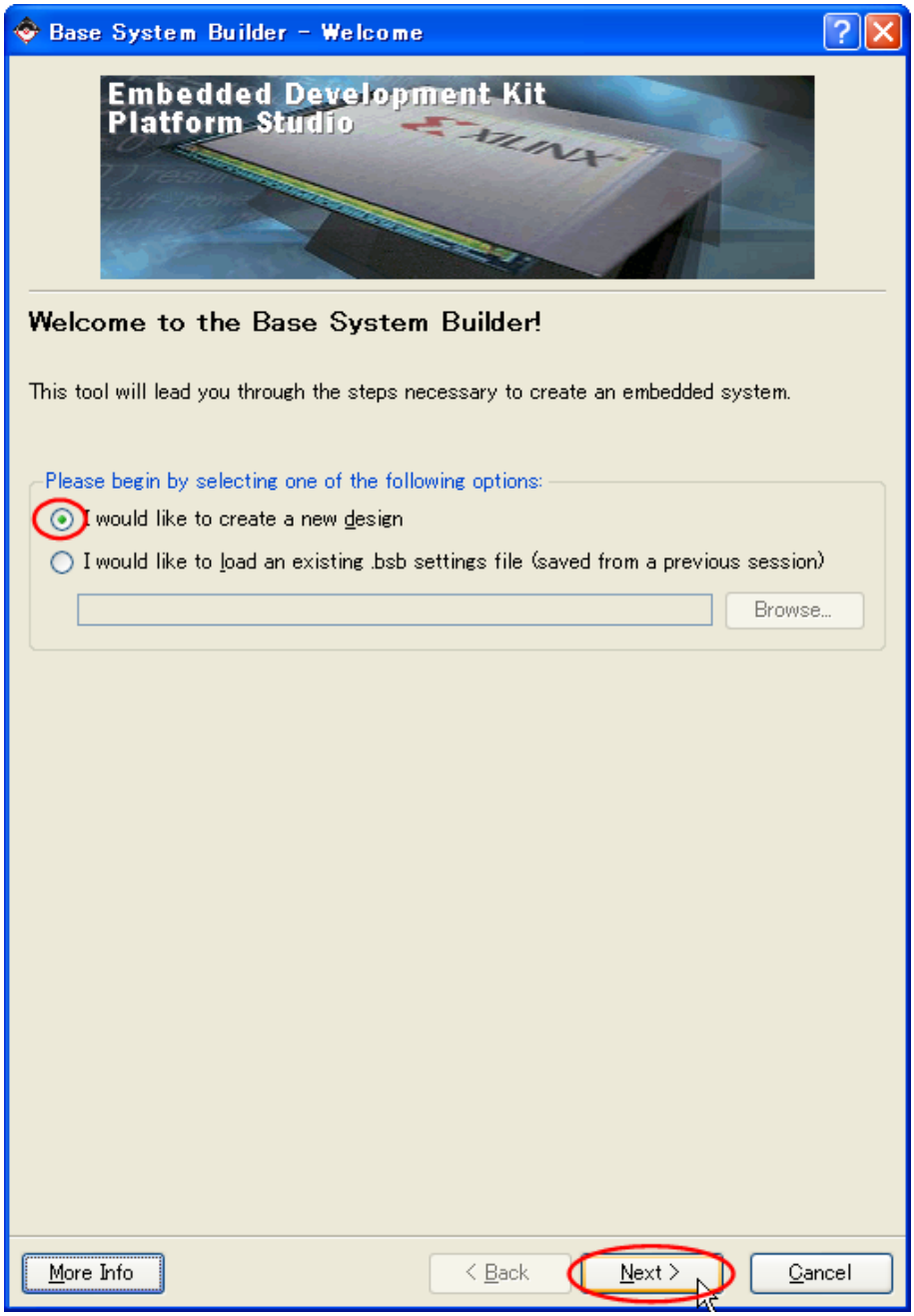


図 10.6. 新しいデザインをはじめる

ターゲットとなるボードの選択を行います。1 から全てカスタムで作るので、[I would like to create a system for a custom board]を選択し、[Next]をクリックして下さい。

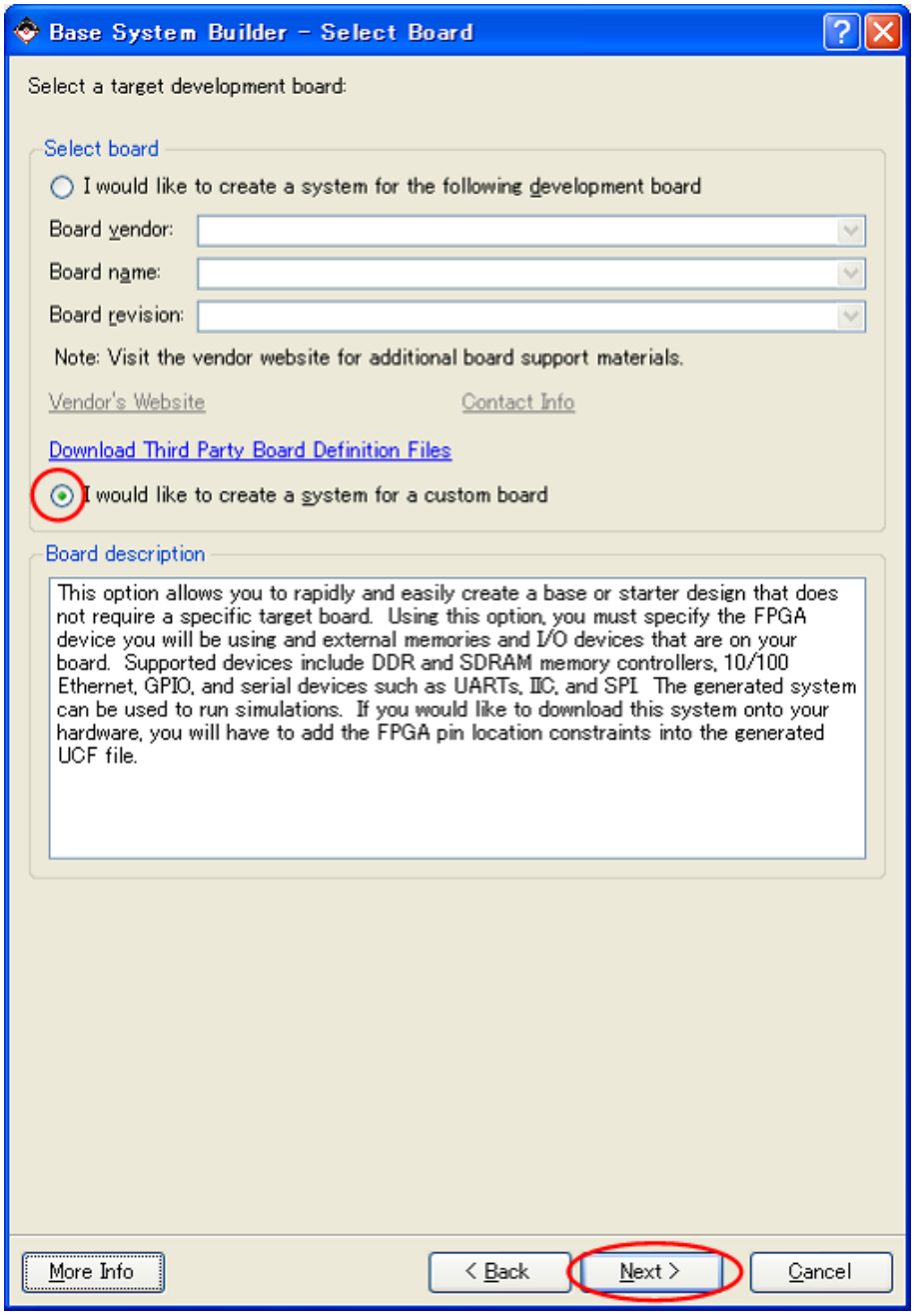


図 10.7. ターゲットボードの選択

FPGA とプロセッサの選択画面が表示されます。FPGA は、お使いの SUZAKU の型式の設定にしてください。SZ010,SZ030,SZ130 の場合 CPU は MicroBlaze を選択してください。SZ310,SZ410 の場合はどちらを選択してもかまいません。選択できたら、[Next]をクリックしてください。

型式	SZ010	SZ030	SZ130	SZ310	SZ410
Architecture	spartan3		spartan3e	virtex2p	virtex4
Device	xc3s400	xc3s1000	xc3s1200e	xc2vp4	xc4vfx12
Package	ft256		fg320	fg256	sf363
Speed grade	-4			-5	-10

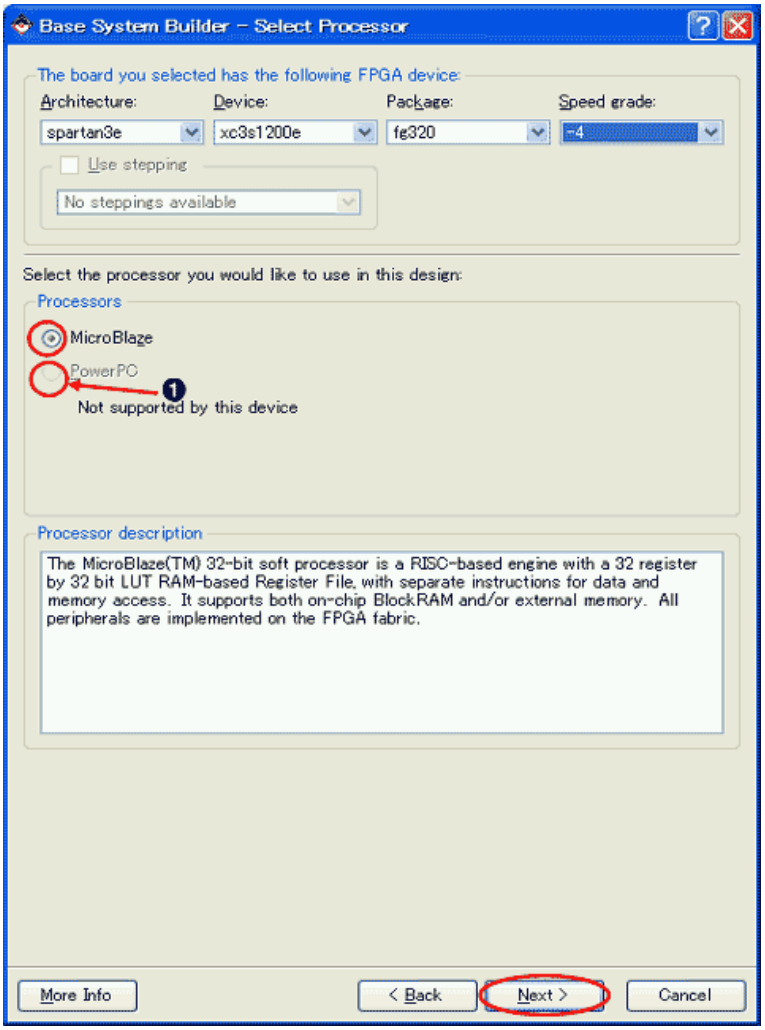


図 10.8. FPGA とプロセッサの設定

❶　SZ310,SZ410 の場合 PowerPC でも可

MicroBlaze もしくは PowerPC の設定を行います。

10.1.1.1. MicroBlaze の場合の設定

SZ010,SZ030,SZ130,SZ310 のクロックは 3.6864MHz なので、Reference clock frequency を 3.6864MHz に設定してください。Processor-Bus clock frequence は SUZAKU のデフォルトにならって、51.61MHz とします。(この値は SZ010 でクロックに何も制約をかけなかった場合の最大値です。)

SZ410 のクロックは 100MHz なので、Reference clock frequency を 100MHz とし、Processor-Bus clock frequence を 100MHz としてください。

Reset polarity は Active High に設定してください。デバッガは今回使わないので No debug をチェックしてください。設定ができたら[Next]をクリックして下さい。

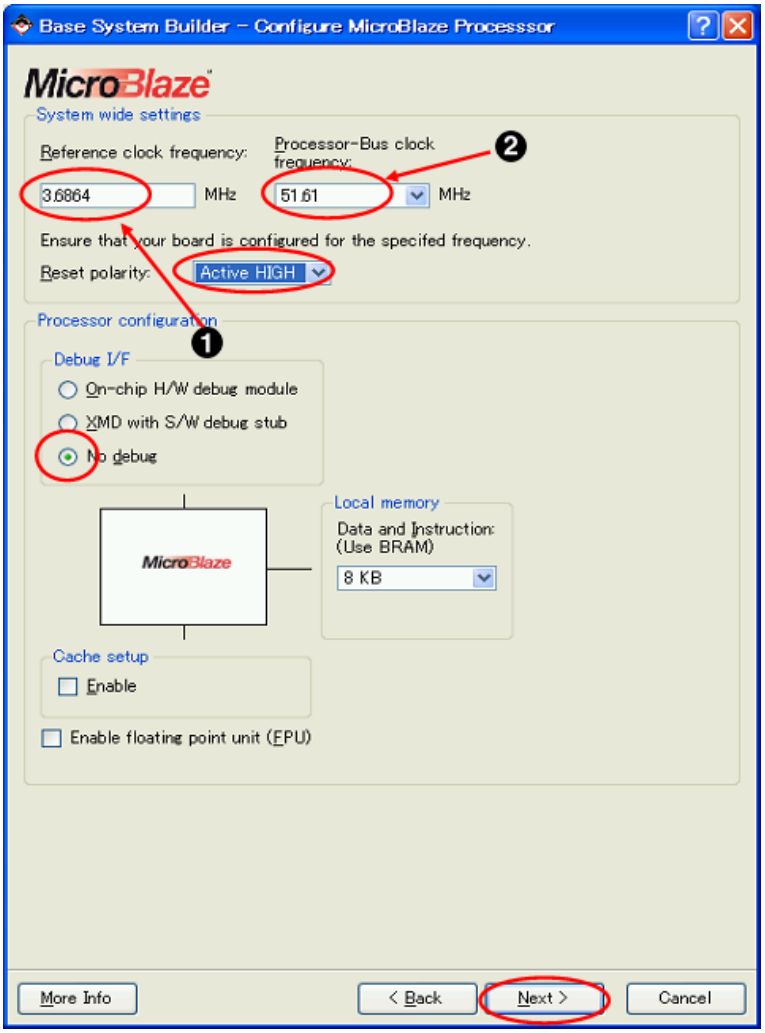


図 10.9. MicroBlaze の設定

❶　SZ010, SZ030, SZ130, SZ310 の場合 3.6864MHz、SZ410 の場合 100MHz

❷　SZ010, SZ030, SZ130, SZ310 の場合 51.61MHz、SZ410 の場合 100MHz

10.1.1.2. PowerPC の場合の設定

SZ310 SZ410

SZ310 の場合クロックは 3.6864MHz なので、Reference clock frequency を 3.6864MHz に設定してください。Processor-Bus clock frequence は 29.49MHz とし、Bus clock frequency は 29.49MHz とします。

SZ410 の場合クロックは 100MHz なので、Reference clock frequency を 100MHz に設定してください。Processor-Bus clock frequence は 300MHz とし、Bus clock frequency は 100MHz とします。

Reset polarity は Active High に設定してください。デバッガは今回使わないので No debug をチェックしてください。設定ができたら[Next]をクリックして下さい。

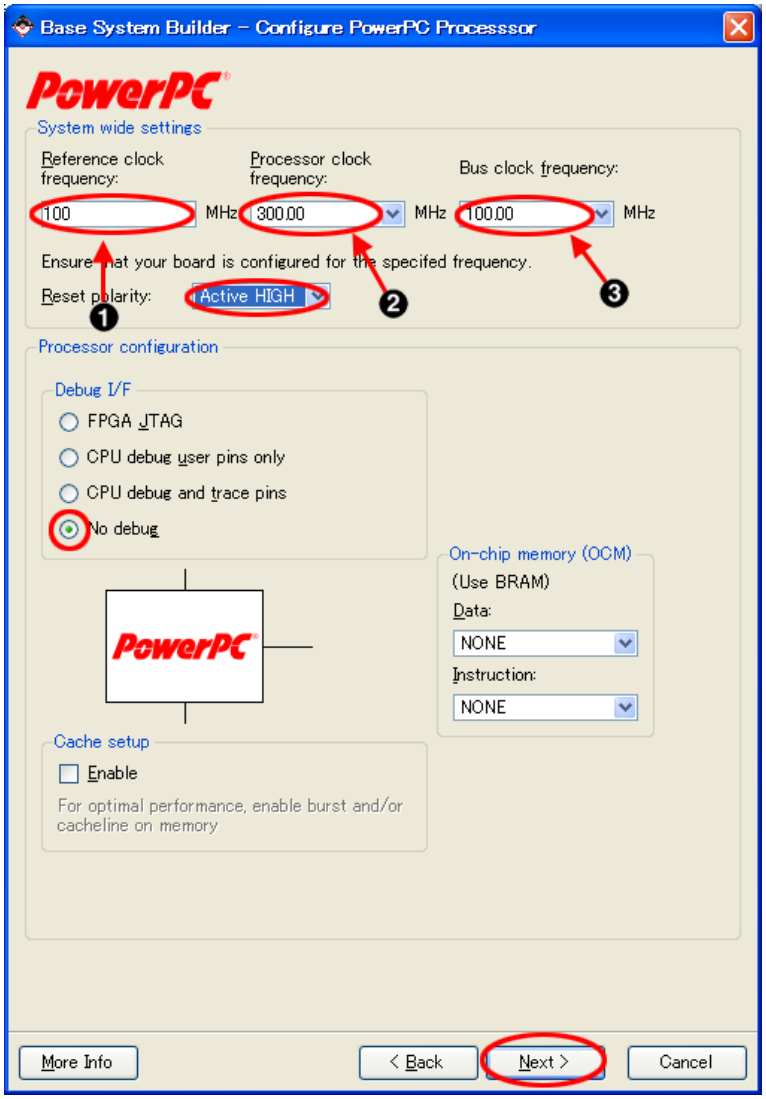


図 10.10. PowerPC の設定

❶　SZ310 の場合 3.6864MHz、SZ410 の場合 100MHz

❷　SZ310 の場合 29.49MHz、SZ410 の場合 300MHz

❸　SZ310 の場合 29.49MHz、SZ410 の場合 100MHz

I/O デバイスのコアの追加画面が表示されます。シリアルを使うので、UART のコアを追加します。[Add Device]をクリックして下さい。Add Device ウィンドウが出るので、IO Interface Type で[UART]を選択し、[OK]をクリックして下さい。

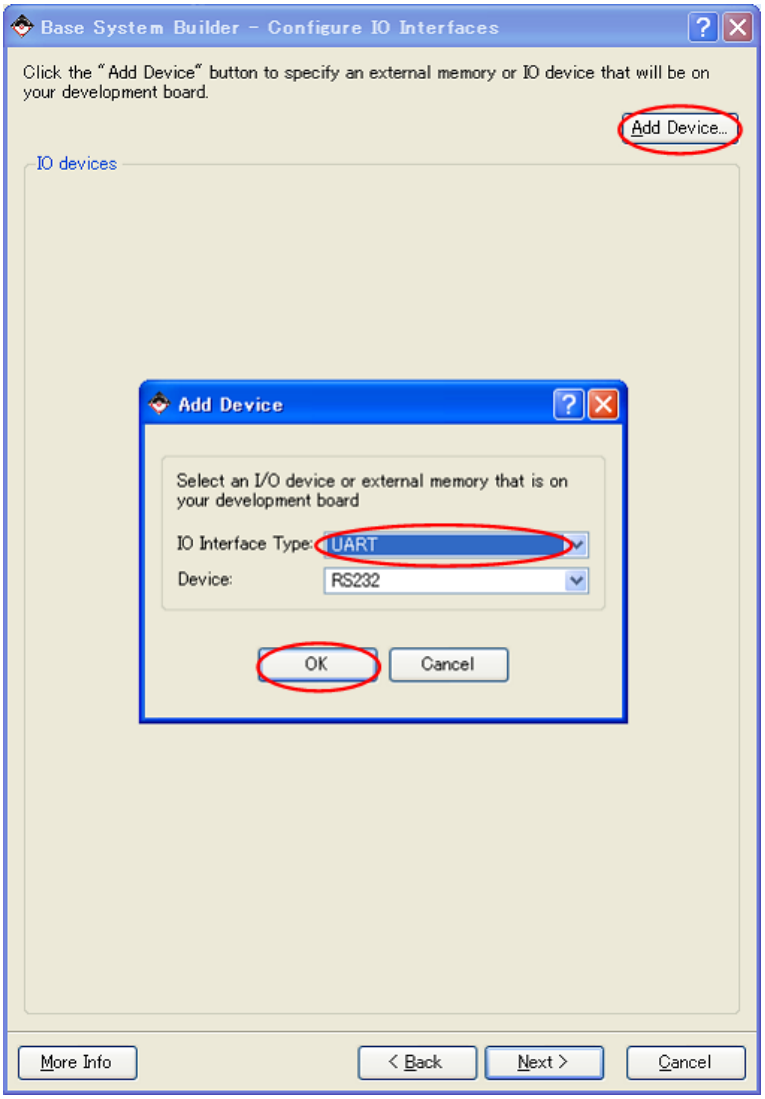


図 10.11. I/O デバイスの選択

以下のように UART が追加されます。Baudrate を[115200]に変更し、Use Parity のチェックをはずし、[Next]をクリックして下さい。

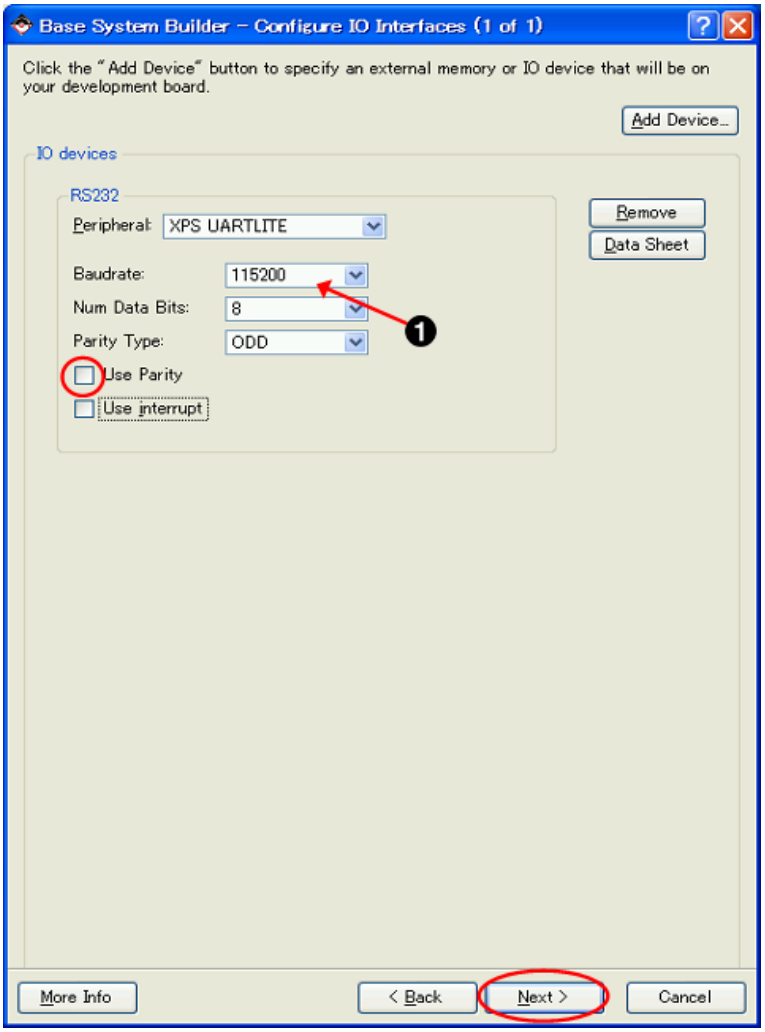


図 10.12. I/O デバイスの選択追加

❶　115200 に変更

ここでは周辺回路を追加できますが、何も追加しません。[Next]をクリックして下さい。PowerPC の場合は Momory size を 16KB に変更し、[Next]をクリックして下さい。

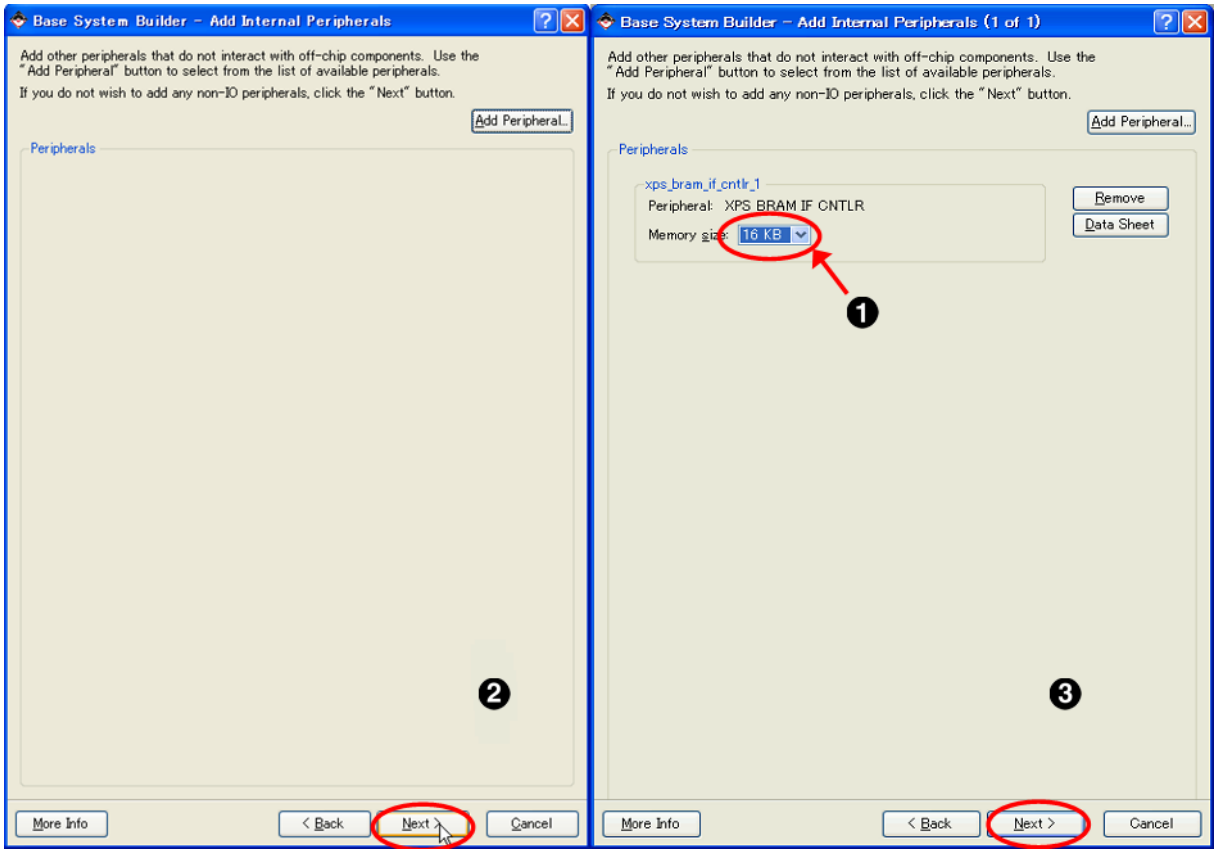


図 10.13. 周辺回路の選択追加

❶ 16KB

❷ MicroBlaze の場合

❸ PowerPC の場合

ソフトウェアの標準入出力とサンプルアプリケーションの選択画面が表示されます。今回は必要ないので、両方ともチェックをはずし、[Next]をクリックして下さい。

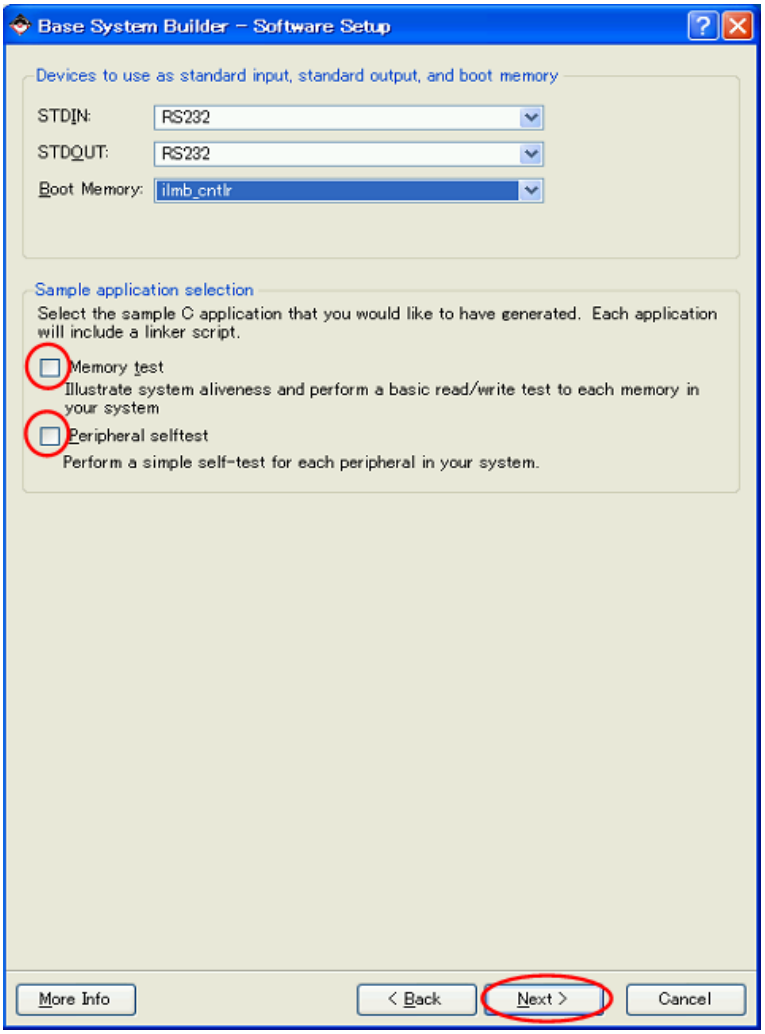


図 10.14. ソフトウェアに関する設定

これまでに設定した項目が下図のように表示されます。PowerPC の場合は BRAM の Base Addr が 0xFFFFC000 であることを確認し、[Generate]をクリックしてください。

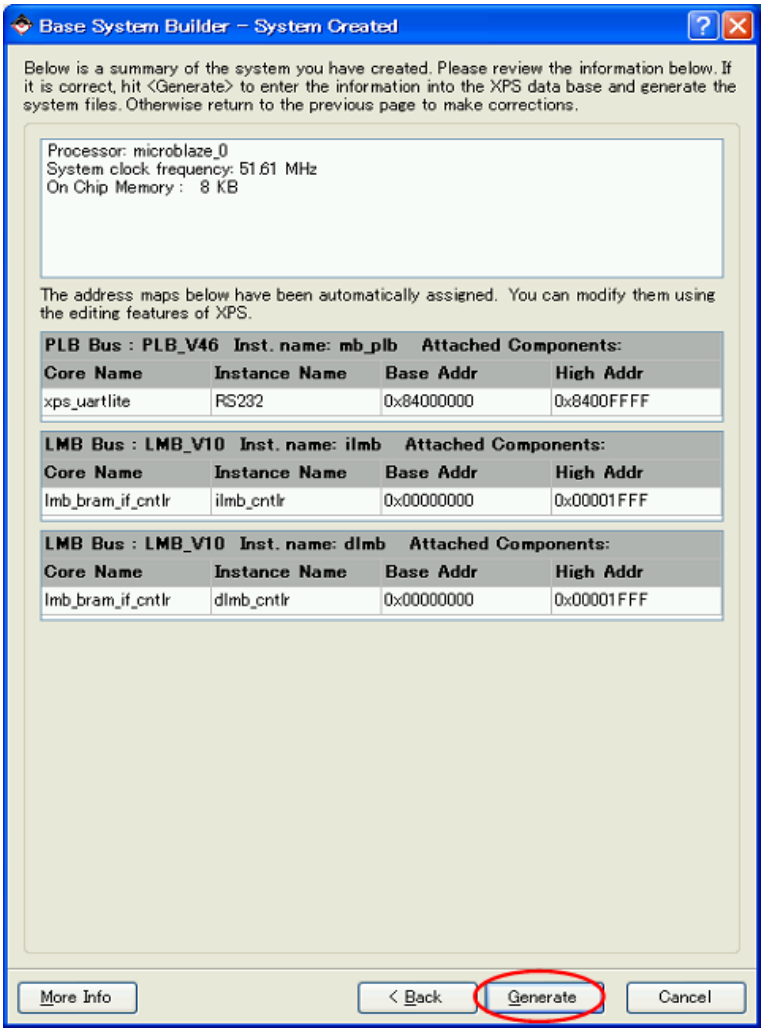


図 10.15. 設定の確認(MicroBlaze)

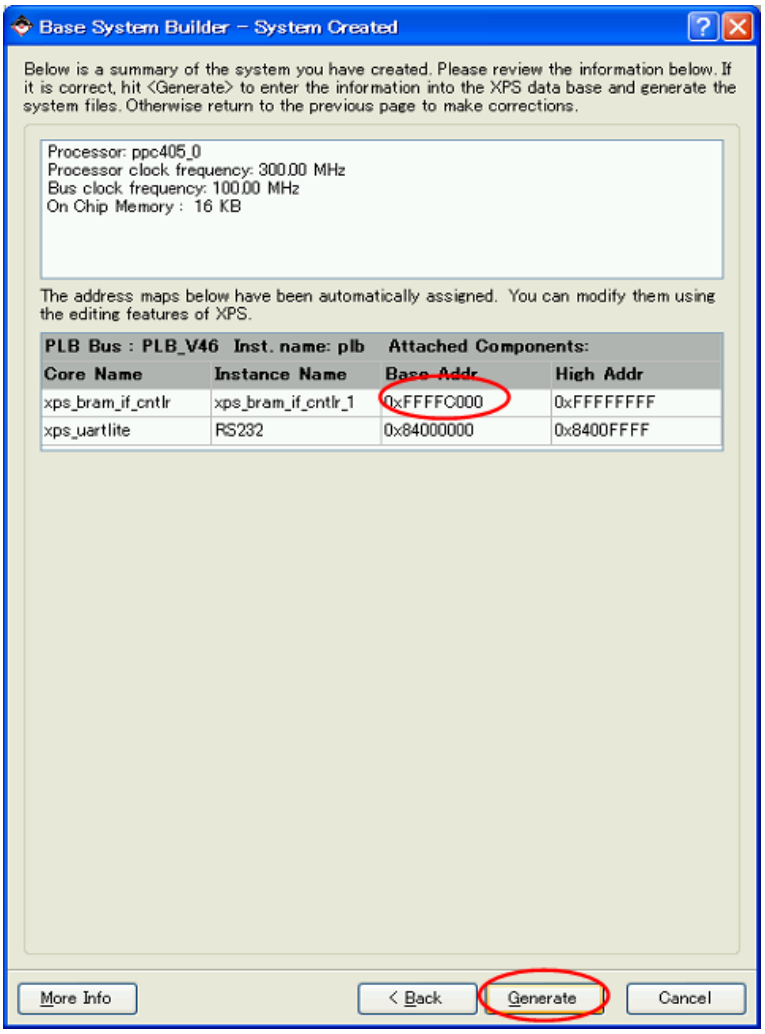


図 10.16. 設定の確認(PowerPC)

以下のように生成されたファイルが表示されます。生成されたファイルを確認し、[Finish]をクリックして下さい。

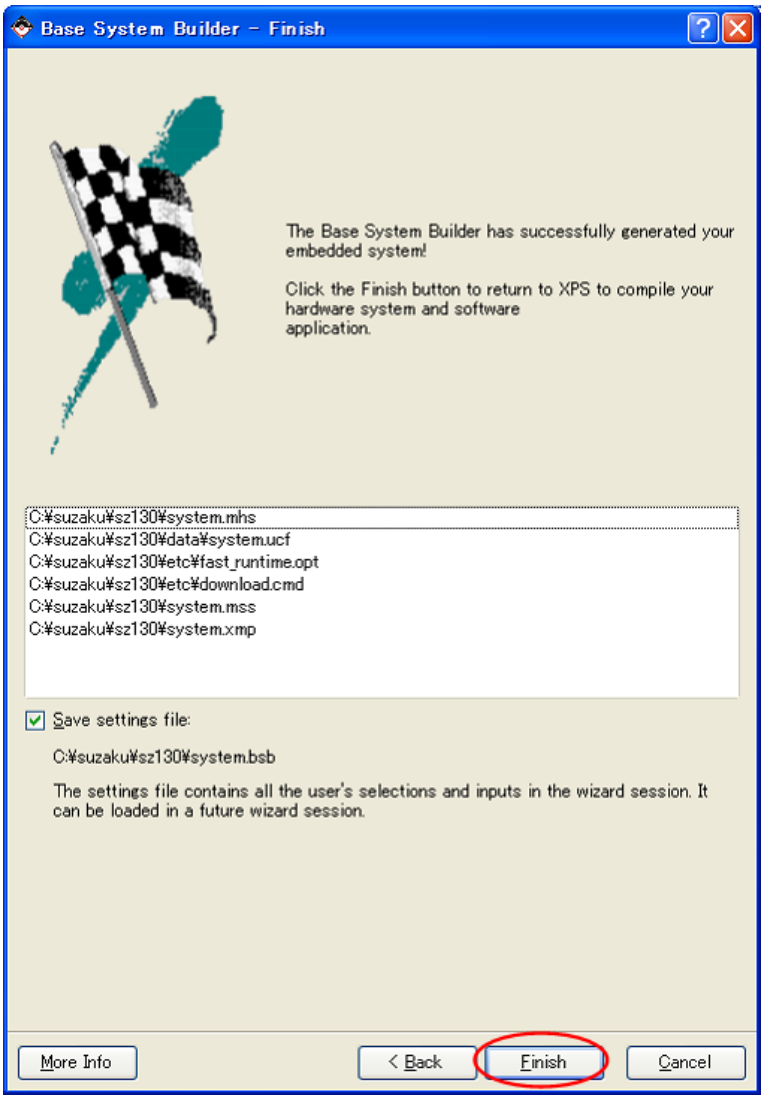


図 10.17. システムの生成完了

以下のウィンドウが立ち上がるので、[OK]をクリックして下さい。

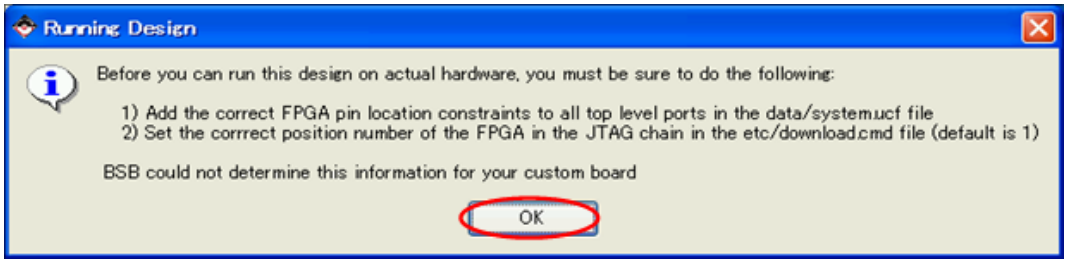


図 10.18. XPS に戻る

以下のような構成で自動生成されます。

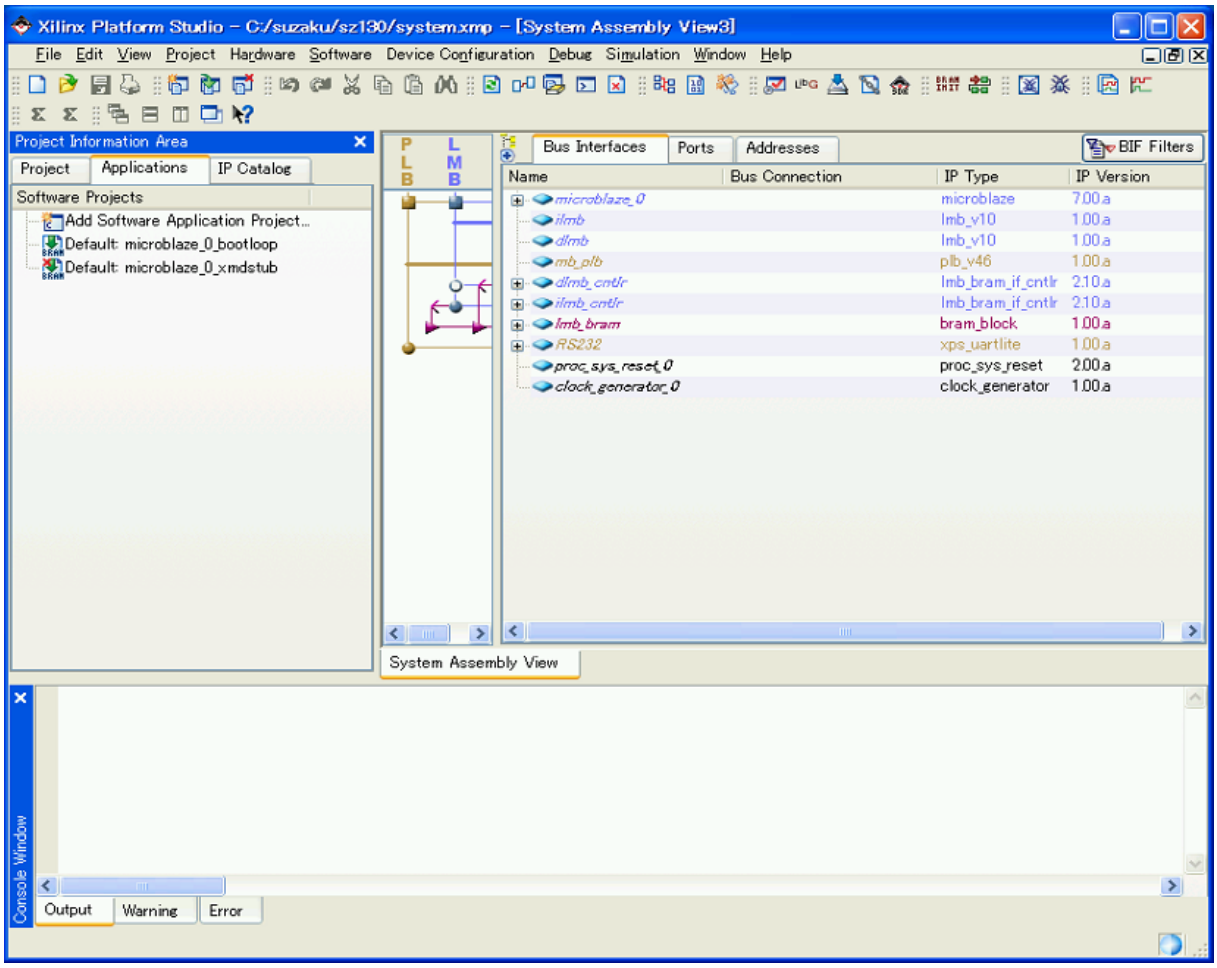


図 10.19. XPS の表示

DCM について

DCM は遅延ロックループ(DLL)、デジタル周波数合成(DFS)、位相シフト(Phase Shifter)、ステータスロジックの 4 つのユニットで構成されていて、これらは独立、または互いに関連して動作します。

DLL：クロック出力信号の伝搬遅延がゼロになるようにスキュー調整を行う。2 逓倍クロック、クロック分周、1/4 位相シフト出力を生成できる。

DFS：自分で設定した 2 つの整数により、逓倍、分周したクロックを生成できる。

Phase Shifter：CLKIN 入力に対するクロック出力の位相関係を制御する。

ステータスロジック：DCM の現在の状態を出力する。

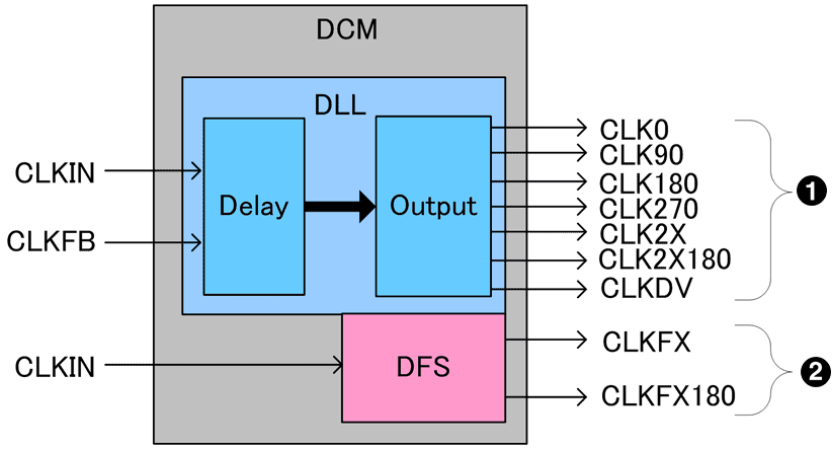


図 10.20. DCM の一部

❶ DLL 機能

❷ DFS 機能

3.6864MHz を 14 倍して 51.6096MHz(PowerPC の場合は 29.49MHz)にするには DFS の機能を使います。DLL、DFS の CLKIN へ入力できるクロック周波数はそれぞれ以下の表のとおりです。DLL の機能を使うと、3.6864MHz は範囲外になり入力できなくなってしまうので、DFS の機能だけ使うようにします。

表 10.1. 入力できるクロック周波数

型式	SZ010	SZ030	SZ130	SZ310	SZ410
DLL(MHZ)	18~280		5~200	32~150	24~180
DFS(MHz)	1~280		0.2~333	1~210	

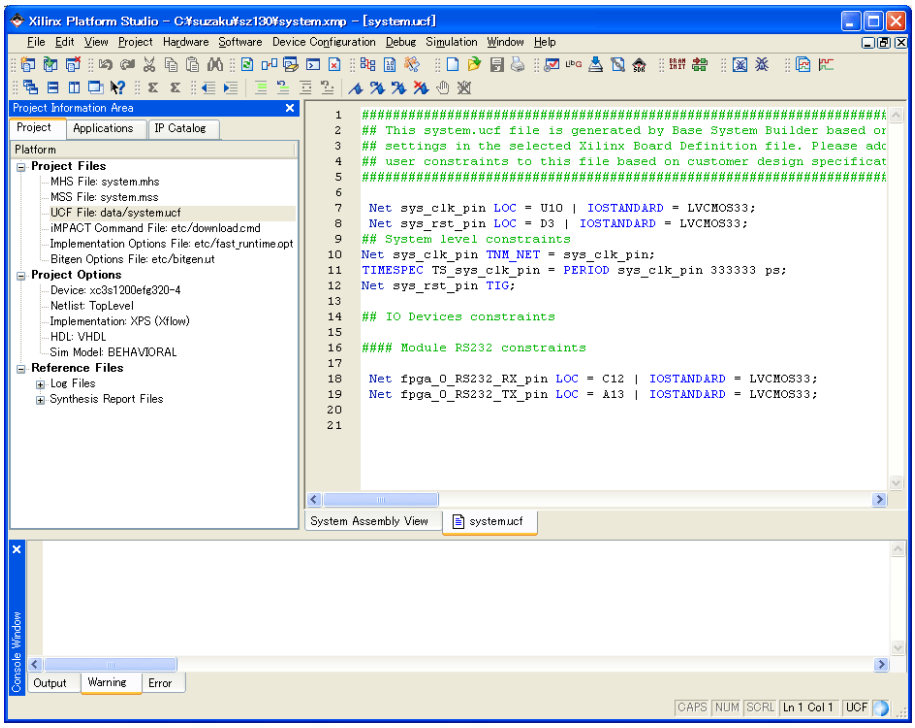
10.1.2. XPS ハードウェア設定

10.1.2.1. ピンの設定

Project タブをクリックし、UCF File: `data/system.ucf` をダブルクリックして開いてください。ピンアサインを設定します。sys_clk_pin、sys_rst_pin、fpga_0_RS232_RX_pin、fpga_0_RS232_TX_pin をそれぞれピンアサインしてください。それぞれコメントアウトした記述があると思います。記述できたら[File]→[Save]をクリックし、保存してください。

表 10.2. ピンアサイン(system.ucf)

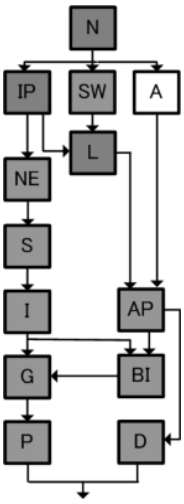
	SZ010 SZ030	SZ130	SZ310	SZ410
sys_clk_pin	T9	U10	C8	Y6
sys_rst_pin	F5	D3	A8	U3
fpga_0_RS232_RX_pin	E2	C12	C10	Y4
fpga_0_RS232_TX_pin	E4	A13	C9	U4




```
Net sys_clk_pin LOC=U10 | IOSTANDARD = LVCMOS33 ;
Net sys_rst_pin LOC=D3 | IOSTANDARD = LVCMOS33 ;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 333333 ps;
Net sys_rst_pin TIG;

## IO Devices constraints

#### Module RS232 constraints

Net fpga_0_RS232_RX_pin LOC=C12 | IOSTANDARD = LVCMOS33 ;
Net fpga_0_RS232_TX_pin LOC=A13 | IOSTANDARD = LVCMOS33 ;
```

図 10.21. SZ130 の場合のピンアサイン(system.ucf)

10.1.3. XPS アプリケーション作成



Hello SUZAKU と表示するアプリケーションを作成します。
Applications のタブをクリックしてください。
Add Software Application Project を右クリックし、Add Software Application Project をクリックして下さい。ウィンドウが立ち上がるので、Project Name に hello-suzaku と入力し、[OK]をクリックして下さい。

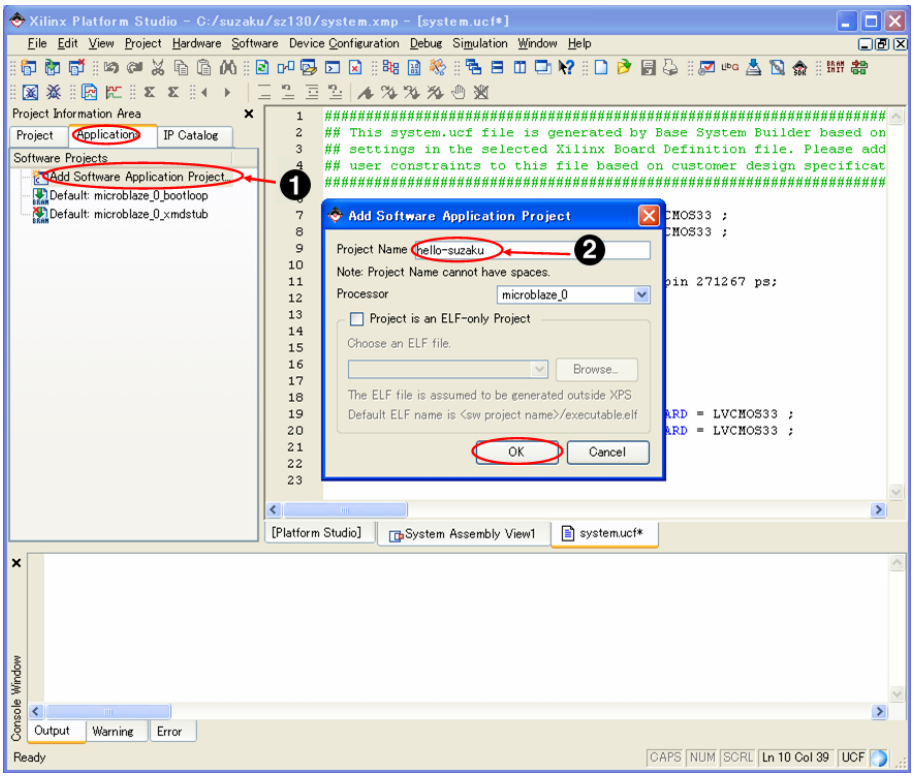


図 10.22. hello-suzaku 作成

❶ 右クリックして、Add Software Application Project を選択

❷ プロジェクトの名前入力

Project: hello-suzaku ができるので、Sources の上で右クリックし、Add New files を選択しください。その場に直接プログラムを置いてもいいのですが、ファイルがたくさんになると分かりにくくなるので、フォルダを 1 つ作成します。hello-suzaku というフォルダを作成し、その中に `main.c` というファイルを作ってください。

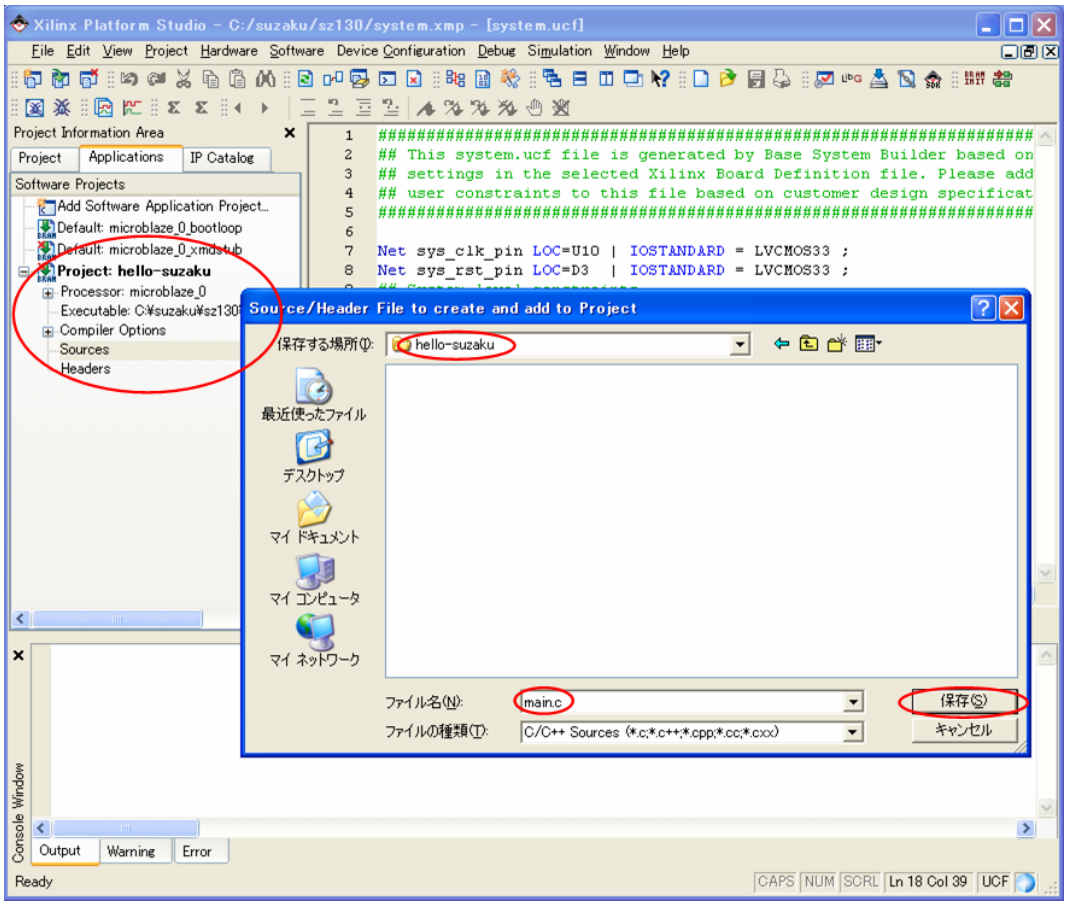


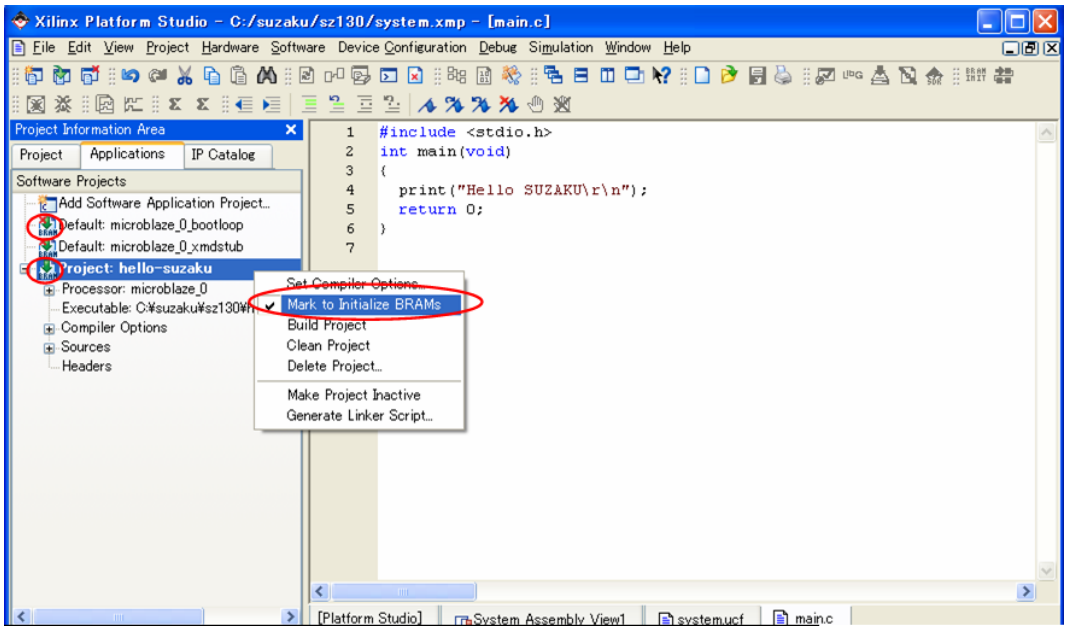
図 10.23. main.c 作成

Sources の下に追加されるので `main.c` をダブルクリックして開き、Hello SUZAKU と表示するプログラムを記述してください。記述できたら[File]→[Save]を選択し、保存してください。

Project : hello-suzaku を右クリックして、Mark to Initialize BRAMs をクリックして下さい。

チェックマークがつき、Project : hello-suzaku の横のアイコンが BRAM に変わります。これで hello-suzaku が BRAM に初期値として書き込まれるようになります。

microblaze_0_bootloop は書き込みません。Project : BBoot の横のアイコンが BRAM であることを確認してください。




```
#include <stdio.h>
int main(void)
{
  print("Hello SUZAKU\r\n");
  return 0;
}
```

図 10.24. Hello SUZAKU のソースコード(main.c)

PowerPC の場合はリンカースクリプトの設定が必要となります。Project:hello-suzaku の部分をダブルクリックして下さい。Use Default Linker Script をチェックし、Program Start Address に 0xFFFFC000 と入力して[OK]をクリックして下さい。0xFFFFC000 は BRAM の Base Address で、プログラムが BRAM から始まるように設定されます。

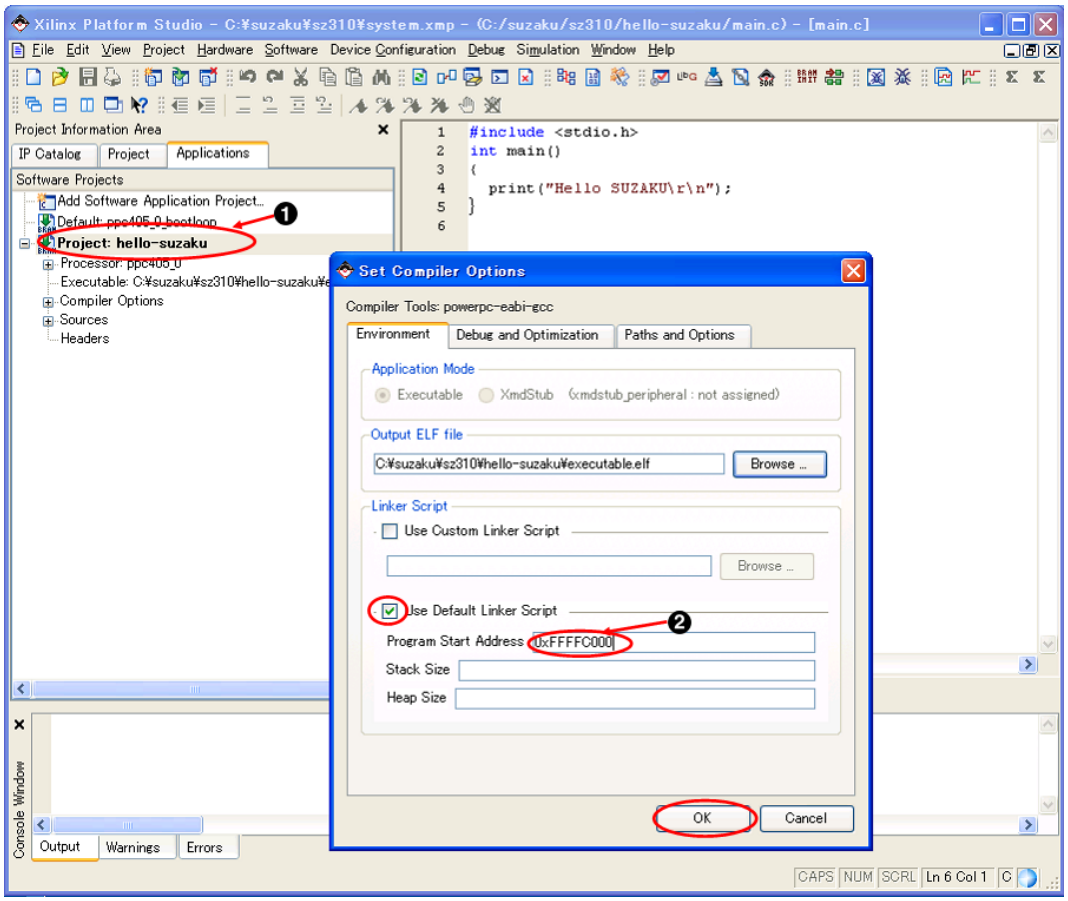


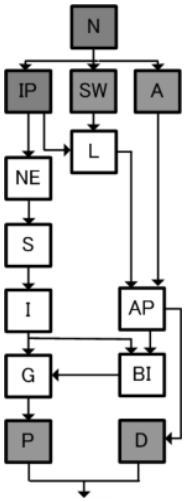
図 10.25. リンカースクリプトの設定(PowerPC)

❶　ダブルクリック

❷　0xFFFFC000 と入力

10.1.4. プログラムファイルを作成してコンフィギュレーション

[Device Configuration]→[Update Bitstream] をクリックしてください。bit ファイルが生成されます。



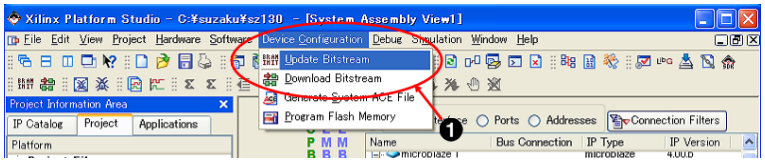
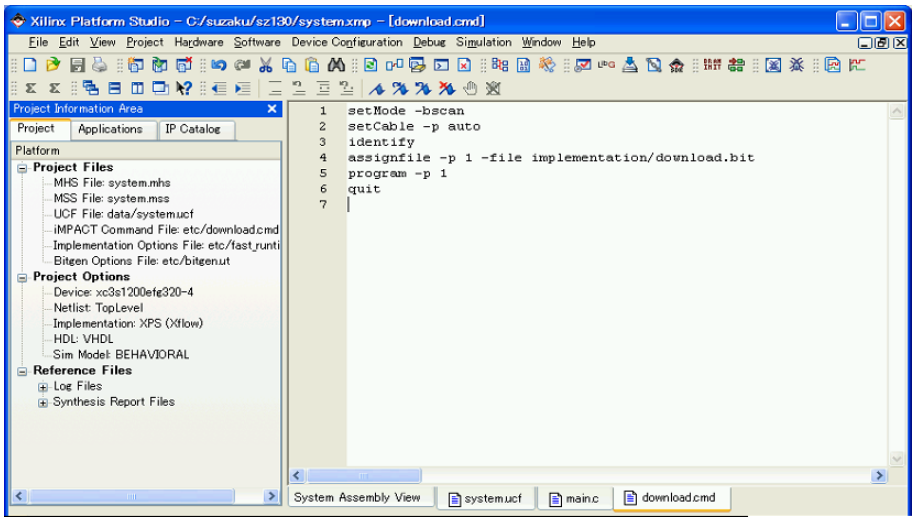


図 10.26. bit ファイル作成

❶　[Device Configuration]→[Update Bitstream]を選択

Project タブをクリックし、iMPACT Command File etc/download.cmd をダブルクリックして開いてください。ダウンロードの設定を変更し保存してください。




```
setMode -bscan
setCable -p auto
identify
assignfile -p 1 -file implementation/download.bit
program -p 1
quit
```

図 10.27. download.cmd の変更

シリアル通信用ソフトウェアを立ち上げ、シリアル通信の設定を行ってください。

(「5.2. シリアル通信ソフトウェア」参照)

SUZAKU JP2 をショートし、SUZAKU CON7 にダウンロードケーブルを接続してください。

LED/SW CON7 にシリアルケーブルを接続してください。

最後に LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。

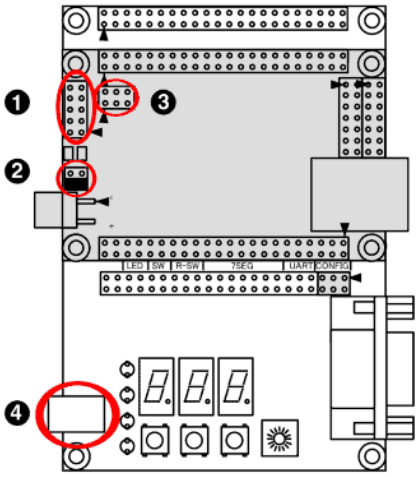


図 10.28. ジャンパの設定等

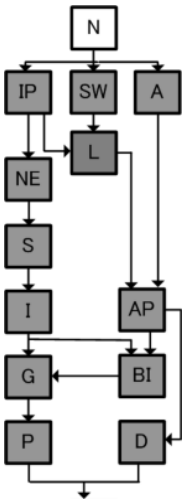
❶ シリアルケーブル接続

❷ JP2 ショート

❸ ダウンロードケーブル接続

❹ 電源投入

[Device Configuration]→[Download Bitstream]をクリックしてください。バッチモードの iMPACT を使用して FPGA に bit ファイルがコンフィギュレーションされます。
シリアル通信ソフトウェアに下図のように表示されたら成功です。



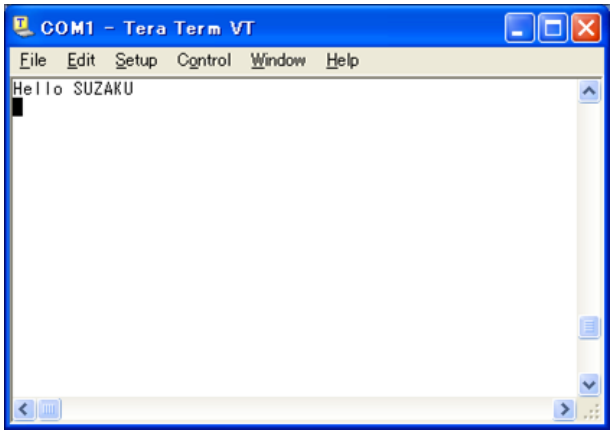


図 10.29. 書き込み成功例

フラッシュメモリに書き込む bit ファイル

このままの設定だと、フラッシュメモリに書き込める bit ファイルを生成することが出来ません。フラッシュメモリに書き込む bit ファイルを作る場合は `bitgen.ut` ファイルを開き、-g StartUpCLK:JTAGCLK を-g StartUpClk:CCLK に変更してください。

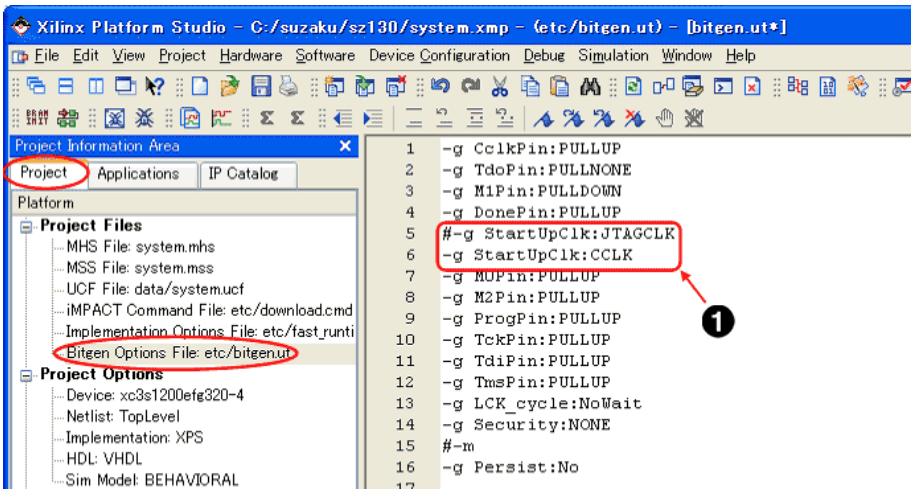


図 10.30. bitgen.ut の変更

❶ #-g StartUpClk:JTAGCLK

-g StartUpClk:CCLK

かなり簡単に MicroBlaze を動かすことが出来ました。EDK の感触はつかめたでしょうか。今つくった Hello SUZAKU プロジェクトを踏まえて次の SUZAKU のデフォルトを見てみてください。

10.2. SUZAKU のデフォルト

SUZAKU のデフォルトを説明します。

付属 CD-ROM の`"\suzaku\fpga_proj\x.x\sz***\sz***-yyyymmdd.zip"`[1]をハードディスクに展開してください。ここでは展開後のフォルダを`"C:\suzaku"`の下にコピーします。`"C:\suzaku\sz***-yyyymmdd"`の中の`"xps_proj.xmp"`をダブルクリックして開いてください。Platform Studio が起動し、SUZAKU のデフォルトが開きます。Platform Studio は`"EDK` のインストールフォルダ`\bin\nt\_xps.exe"`もしくは[スタートメニュー]→[すべてのプログラム]→[Xilinx ISE Design Suite X.X]→[EDK]→[Xilinx Platform Studio]から起動できます。

[1]SZ410 の場合は 20080118 以降のプロジェクトを使用してください

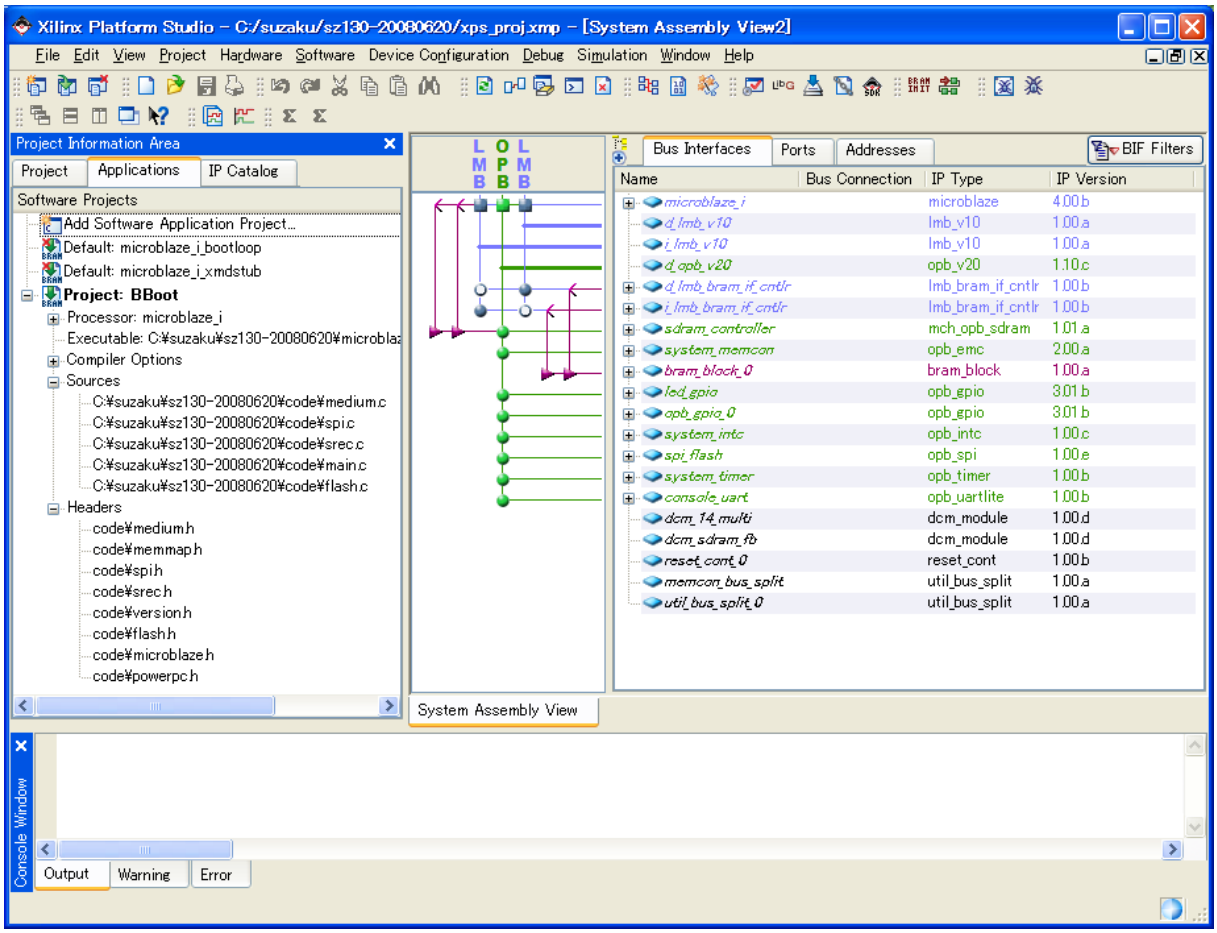


図 10.31. XPS 起動

10.2.1. SZ010, SZ030 の構成

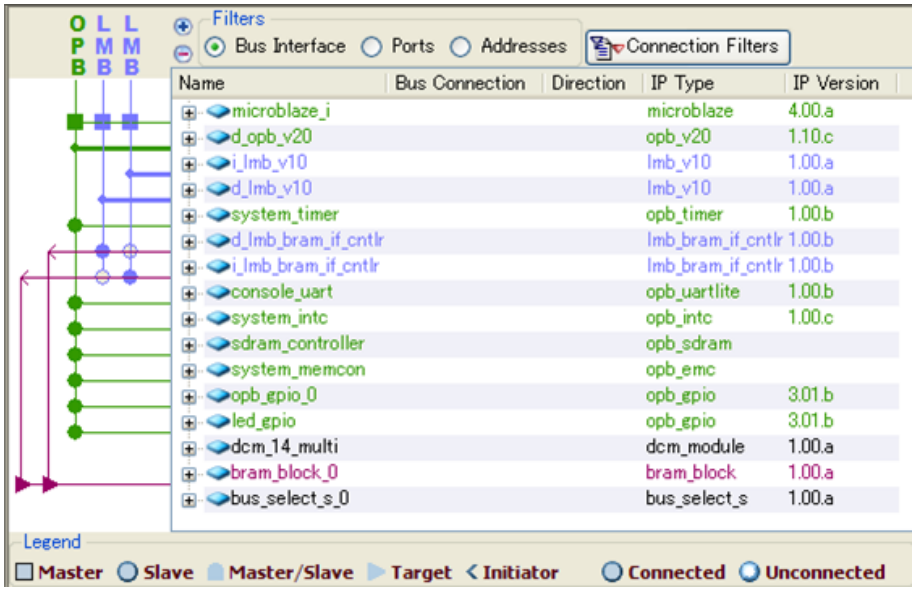


図 10.32. SZ010、SZ030 のデフォルト(EDK)

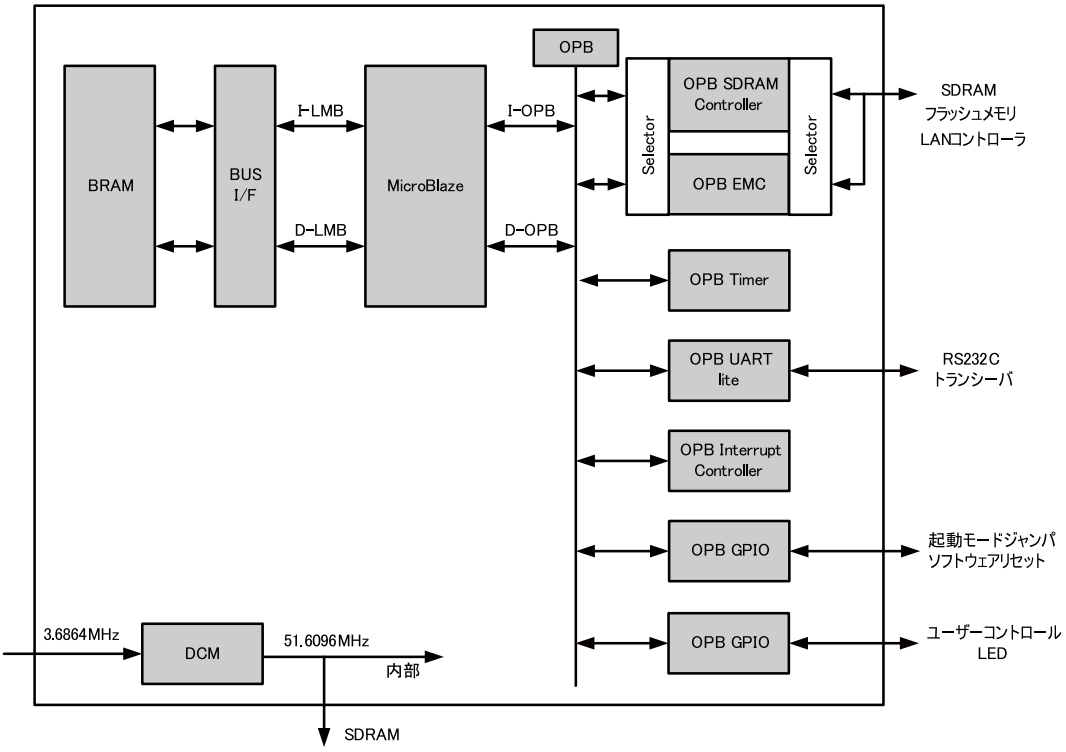


図 10.33. SZ010、SZ030 デフォルトのブロック図

10.2.2. SZ130 の構成

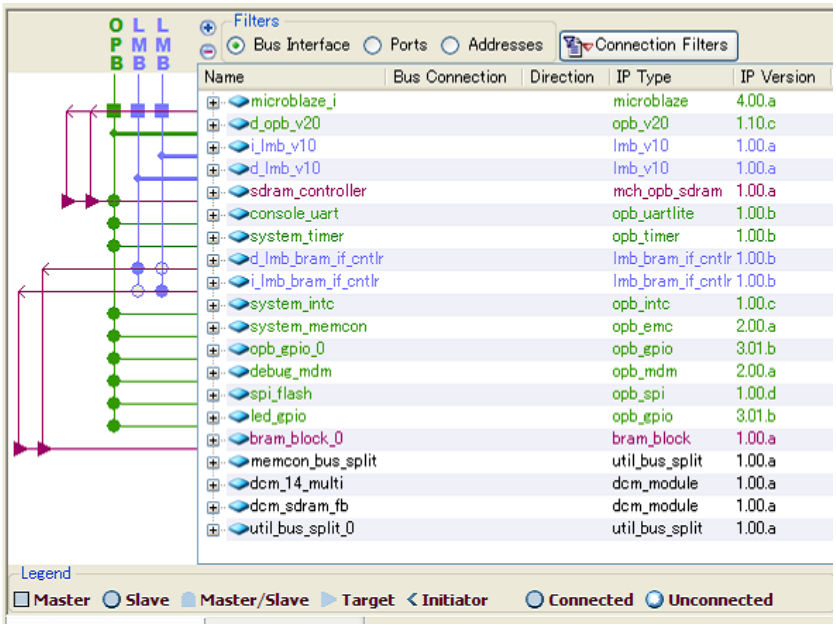


図 10.34. SZ130 のデフォルト(EDK)

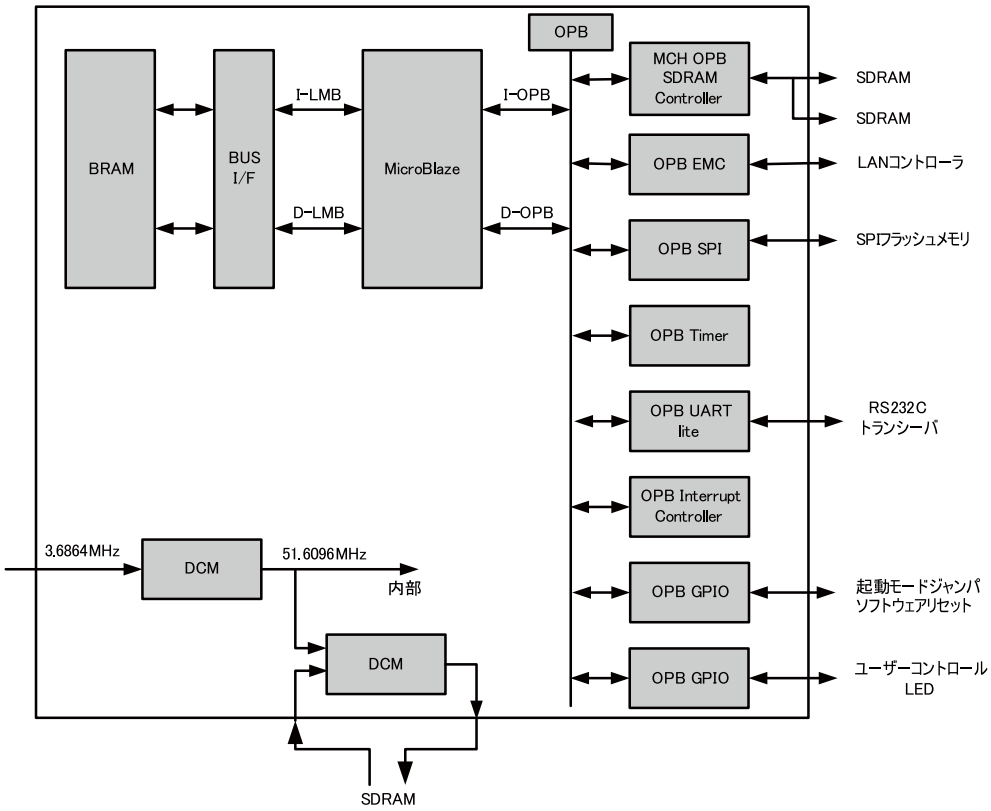


図 10.35. SZ130 デフォルトのブロック図

10.2.3. SZ310 の構成

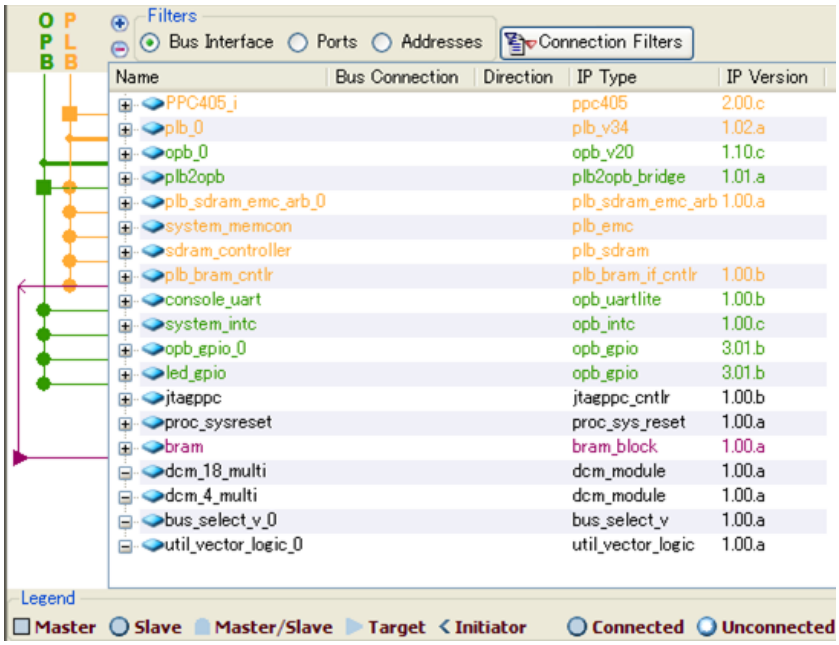


図 10.36. SZ310 のデフォルト(EDK)

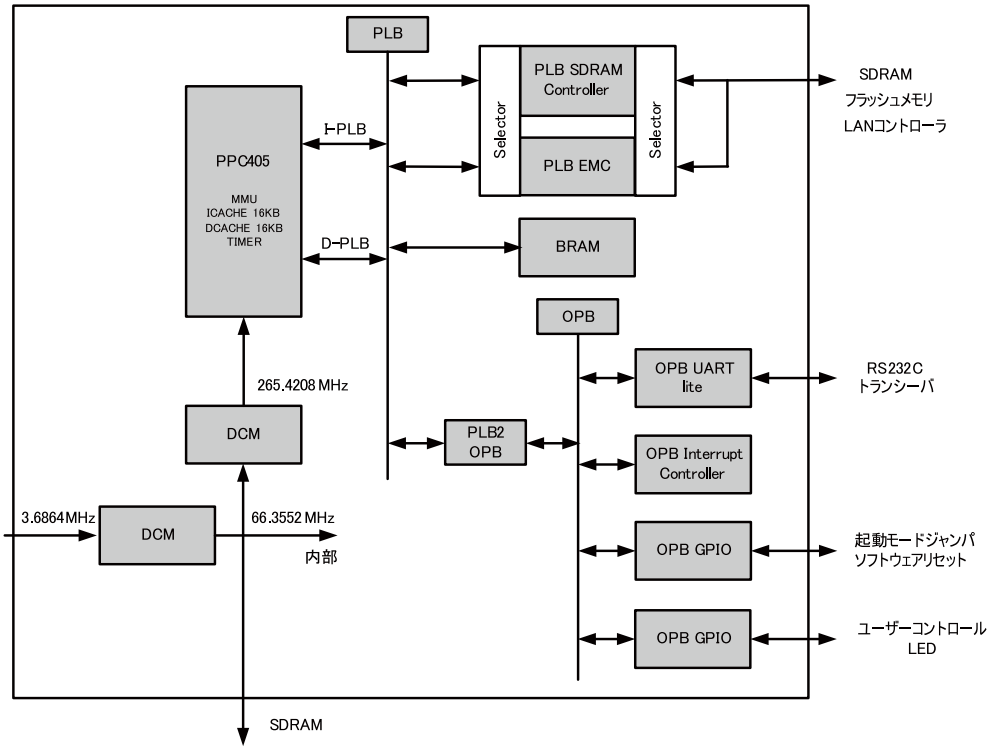


図 10.37. SZ310 デフォルトのブロック図

10.2.4. SZ410 の構成

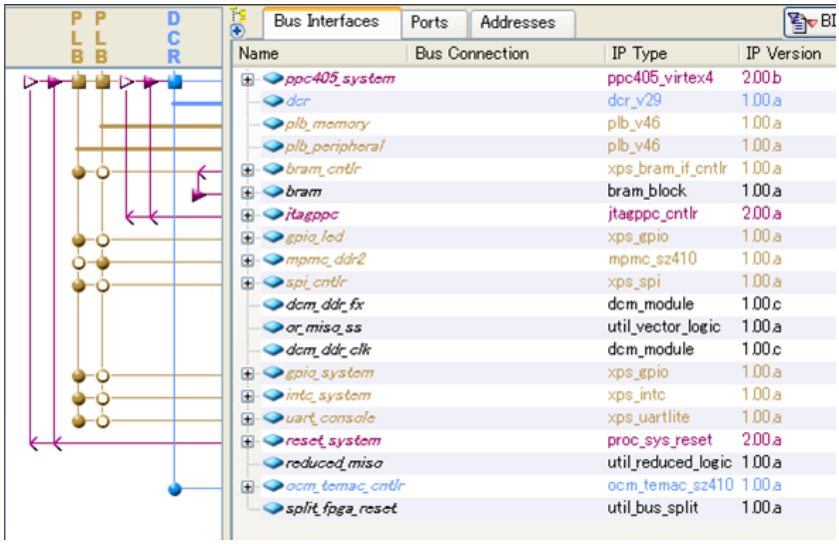


図 10.38. SZ410 のデフォルト(EDK)(2008/1/18 以降)

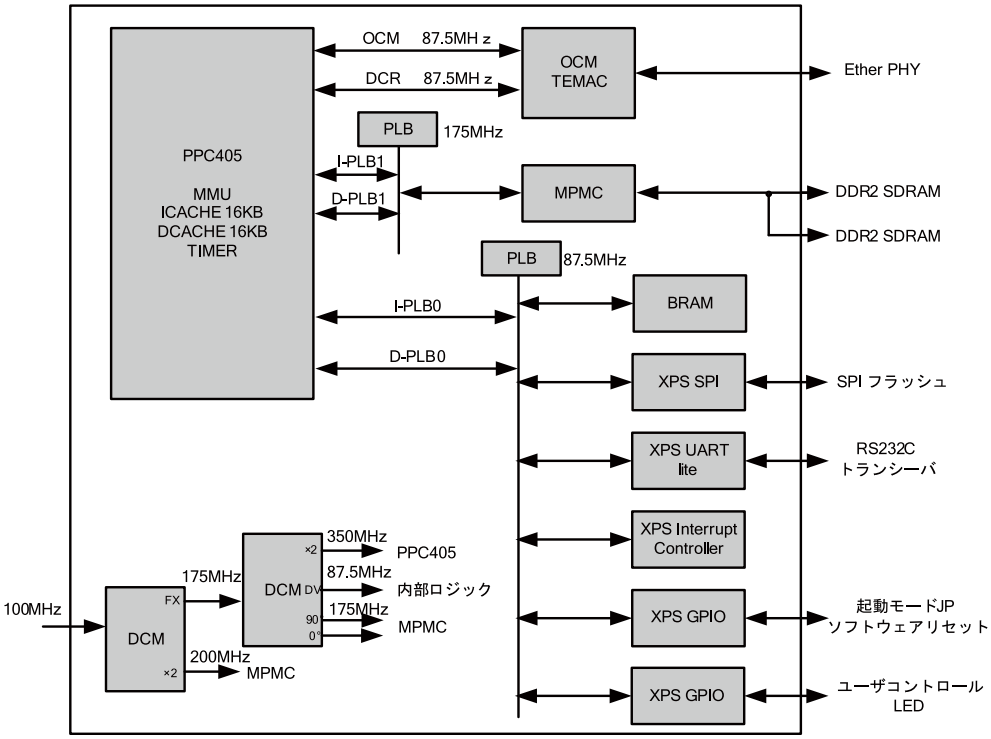


図 10.39. SZ410 デフォルトのブロック図(2008/1/18 以降)

10.2.5. IP コア

SUZAKU の FPGA で使用している IP コアについて説明します。SUZAKU はプロセッサ、バス、FPGA 内部メモリ、外部メモリコントローラ、LAN コントローラ、割り込みコントローラ、タイマ、シリアル、GPIO、クロックの IP コアを使用しています。

10.2.5.1. プロセッサ (microblaze, ppc405)

SZ010,SZ030,SZ130 ではソフトプロセッサの MicroBlaze を、SZ310,SZ410 ではハードプロセッサの PowerPC を使用しています。

MicroBlaze	• 32bitRISC プロセッサ • 32bit 固定長命令 • 32 個の汎用 32bit レジスタ • MMU なし • 命令キャッシュとデータキャッシュ • ハードウェア乗算器 • ハードウェアデバッグロジック対応 • ペリフェラルバス OPB(CoreConnect)
PowerPC	• 32bitRISC プロセッサ • 32bit 固定長命令 • 32 個の汎用 32bit レジスタ • MMU あり • 命令キャッシュとデータキャッシュ • ハードウェア乗算器 • ハードウェアデバッグロジック対応 • ペリフェラルバス PLB(CoreConnect)

10.2.5.2. バス (lmb_v10, opb_v20, plb_v34, plb_v46, dcr_v29, ocm_v10)

SZ010,SZ030,SZ130 は LMB と OPB で構成しています。LMB は MicroBlaze と BRAM を接続する専用バスです。OPB はシリアルやタイマなどの IP コアの接続に使用しています。

SZ310 は OPB と PLB_v3.4 で構成しています。OPB はシリアルやタイマなどの IP コアの接続に使用しています。PLB は PowerPC と BRAM の接続および SDRAM Controller、PLB EMC の接続に使用しています。なお、OPB と PLB はブリッジにより接続しています。

SZ410 は PLB_v4.6 と DCR、OCM で構成しています。PLB は PowerPC と BRAM の接続および MPMC やシリアルなどの IP コアの接続に使用しています。DCR は PowerPC と TEMAC の接続、OCM は PowerPC と TEMAC の FIFO の接続に使用しています。

10.2.5.3. FPGA 内部メモリ BRAM (bram_block)

FPGA は内部メモリに BRAM を持っています。SZ010,SZ030,SZ130 ではプロセッサのプログラムを置く容量を 8KByte に、SZ310,SZ410 では 16KByte に設定しています。これは設定できる最小の容量となっているので、適宜増やしてお使いください。ただ、プロセッサのプログラムを置く容量を多く設定しすぎるとキャッシュやその他のユーザが設定できる RAM や FIFO の容量が減ってしまうため、分配の際には注意が必要となります。

BRAM には FPGA のコンフィギュレーション時に初期値を書き込むことが出来ます。SUZAKU では BRAM に初期値としてブートローダ BBoot(「11.9.6. BBoot 編集」参照)を書き込んでいます。BBoot が終わり、次のプログラム(ブートローダ Hermit や Linux)が起動した後は、SZ010,SZ030,SZ130 では、最初の 32Byte 以外、SZ310,SZ410 では全領域をユーザプログラムが使用することも可能です。

BRAM とバスを接続するために SZ010,SZ030,SZ130 では lmb_bram_if_cntlr、SZ310 では plb_bram_if_cntlr、SZ410 では xps_bram_if_cntlr というコントローラを使用しています。

10.2.5.4. 外部メモリコントローラ (opb_emc, opb_sdram, mch_opb_sdram, plb_emc, plb_sdram, mpmc)

外部メモリコントローラとして、SZ010,SZ030,SZ130 では OPB EMC と OPB SDRAM、SZ310 では PLB EMC と PLB SDRAM を使用しています。OPB EMC、PLB EMC はフラッシュメモリと LAN コントローラとの接続に使用し、OPB SDRAM、PLB SDRAM は SDRAM との接続に使用しています。SZ410 では MPMC を DDR2 との接続に使用しています。

10.2.5.5. LAN コントローラ (hard_temac) SZ410

SZ410 ではハード的に内蔵されている LAN コントローラ、hard_temac を使用しています。

10.2.5.6. 割り込みコントローラ (opb_intc, xps_intc)

割り込みコントローラに OPB | XPS Interrupt Controller を使用しています。最大 32 本の割り込み入力が可能で、それぞれの入力に対し、属性(ハイレベル/ローレベル/立ち上がりエッジ/立下りエッジ)の指定が出来ます。EDK は割り込みコントローラに接続されている IP コアの割り込み線の属性を見て、割り込みの受け付け回路を最適になるように自動生成してくれます。

10.2.5.7. タイマ (opb_timer)

SZ010,SZ030,SZ130 では OPB TIMER を、SZ310,SZ410 では PowerPC の内部タイマを使用しています。

10.2.5.8. シリアル (opb_ uart_lite, xps_uart_lite)

シリアルに OPB | XPS UART lite を使用しています。UART lite は送受信 16MByte ずつの FIFO を持っており、送信 FIFO が空になった時と、受信 FIFO にデータが入ってきた時に割り込みを発生します。UART liteはハードウェアフロー制御信号やモデム制御信号を持っていません。

CON1(RS232C 用 10 ピンヘッダ)には、TXD、RXD、RTS、CTS をピンアサインしています。そのうち UART lite で使用している信号は TXD、RXD のみで、その他の信号は未使用となっています。これらの未使用の信号は GPIO やユーザロジックを接続してフロー制御したり別の UART lite を接続して、2 ポート目の UART とすることも可能です。

10.2.5.9. GPIO (opb_gpio, xps_gpio)

D1、D3 の LED の点灯や JP の判別に OPB | XPS GPIO を使用しています。

10.2.5.10. クロック (dcm_module)

SZ010,SZ030,SZ130 では SUZAKU のクロック 3.6864MHz を DCM で 14 逓倍し 51.6096MHｚにして、SDRAM や内部バス、MicroBlaze に供給しています。SZ310 では SUZAKU のクロック 3.6864MHz を DCM で 18 逓倍し 66.3552MHz にして SDRAM や内部バスに供給し、さらにこれを 4 逓倍して 265.4208MHz にして PowerPC に供給しています。SZ410 では SUZAKU のクロック 100MHz を 7/4 倍および 2 倍して 175MHz および 200MHｚにして MPMC に供給しています。さらに、175MHz を 2 分周して 87.5MHz にして内部バスなどに供給し、175MHz を 2 倍して 350MHz にして PowerPC に供給しています。

10.2.5.11. SPI (opb_spi, xps_xpi)

SZ130 SZ410

SZ130,SZ410 では OPB | XPS SPI を使用しています。SZ130 の FPGA (Spartan-3E)は SPI モードを使うことが出来ます。SPI モードは市販の SPI フラッシュメモリからコンフィギュレーションするモードです。SZ410 では CPLD による SPI コンフィギュレーションを採用しています。

10.2.5.12. ブリッジ (plb2opb_bridge)

SZ310

SZ310 では OPB をブリッジを使用して PLB に接続しています。OPB に追加する IP コアはこのブリッジで設定された BaseAddress と HighAddress の間にアドレスを設定する必要があります。デフォルトでは BaseAddress が 0xF0F00000、HighAddress が 0xF0FFFFFF に設定されています。このアドレスは C_RNG0_BASEADDR と C_RNG0_HIGHADDR の値により変更することが出来ます。もし仮にこのアドレス内からはみ出したところに IP コアのアドレスを設定してコンフィギュレーションしても動かないだけでエラーは出ませんのでご注意ください。

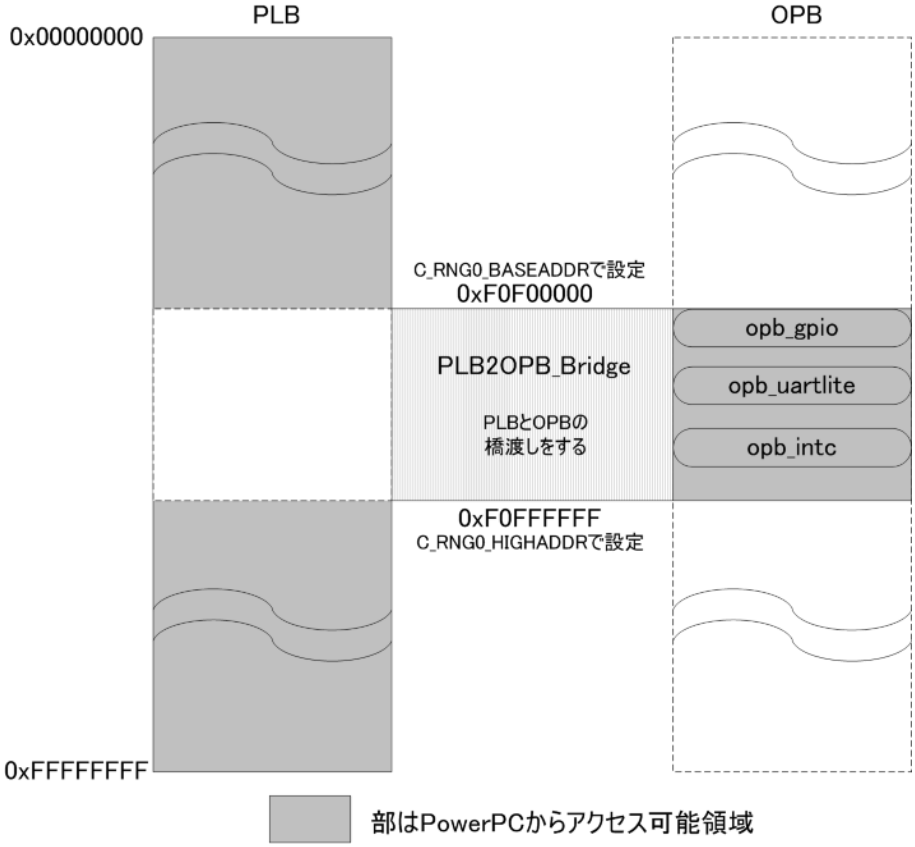
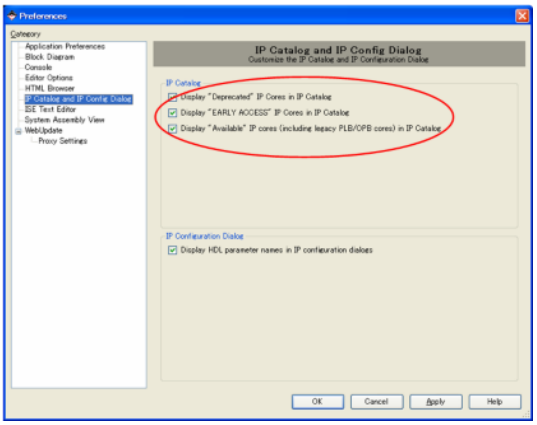
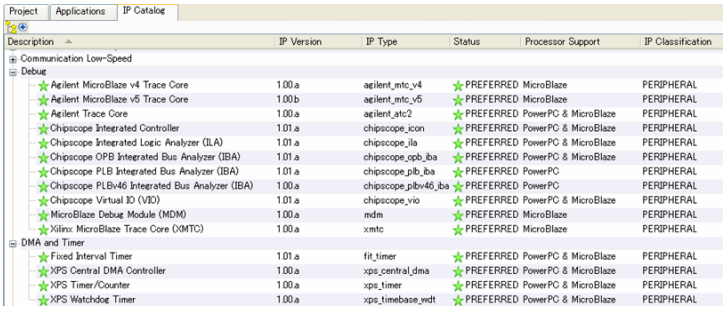

PLB
OPB
0x00000000
C_RNG0_BASEADDRで設定
0xF0F00000
opb_gpio
PLB2OPB_Bridge
opb_uartlite
PLBとOPBの
橋渡しをする
opb_intc
0xF0FFFFFF
C_RNG0_HIGHADDRで設定
0xFFFFFFFF
部はPowerPCからアクセス可能領域

図 10.40. ブリッジ

全 IP 表示

EDK のデフォルトセッティングの場合、IP Catalog に Status が PREFERRED のものしか表示されません。他の Status の IP を表示したい場合は[Edit]→[Preferences]をクリックし、Category で IP Catalog and IP Catalog Dialog を選択し、IP Catalog で Display にチェックしてください。





Project | Applications | IP Catalog

Description	IP Version	IP Type	Status	Processor Support	IP Classification
Communication Low-Speed					
Debug					
Agilent MicroBlaze v4 Trace Core	1.00.a	agilent_mtc_v4	PREFERRED	MicroBlaze	PERIPHERAL
Agilent MicroBlaze v5 Trace Core	1.00.b	agilent_mtc_v5	PREFERRED	MicroBlaze	PERIPHERAL
Agilent Trace Core	1.00.a	agilent_atc2	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
Chipscope Integrated Controller	1.01.a	chipscope_icon	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
Chipscope Integrated Logic Analyzer (ILA)	1.01.a	chipscope_ila	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
Chipscope OPB Integrated Bus Analyzer (IBA)	1.01.a	chipscope_opb_iba	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
Chipscope PLB Integrated Bus Analyzer (IBA)	1.01.a	chipscope_plb_iba	PREFERRED	PowerPC	PERIPHERAL
Chipscope PLBv46 Integrated Bus Analyzer (IBA)	1.00.a	chipscope_plbv46_iba	PREFERRED	PowerPC	PERIPHERAL
Chipscope Virtual IO (VIO)	1.01.a	chipscope_vio	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
MicroBlaze Debug Module (MDM)	1.00.a	mdm	PREFERRED	MicroBlaze	PERIPHERAL
Xilinx MicroBlaze Trace Core (XMTC)	1.00.a	xmtc	PREFERRED	MicroBlaze	PERIPHERAL
DMA and Timer					
Fixed Interval Timer	1.01.a	fit_timer	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
XPS Central DMA Controller	1.00.a	xps_central_dma	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
XPS Timer/Counter	1.00.a	xps_timer	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
XPS Watchdog Timer	1.00.a	xps_timebase_wdt	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL

Project | Applications | IP Catalog

Description	IP Version	IP Type	Status	Processor Support	IP Classification
Interprocessor Communication					
XPS Mailbox	1.00.a	xps_mailbox	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
XPS Mutex	1.00.a	xps_mutex	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
Memory and Memory Controller					
Block RAM (BRAM) Block	1.00.a	bram_block	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
LMB BRAM Controller	1.00.b	lmb_bram_if_cntlr	DEPRECATED	MicroBlaze	PERIPHERAL
LMB BRAM Controller	2.00.a	lmb_bram_if_cntlr	DEPRECATED	MicroBlaze	PERIPHERAL
LMB BRAM Controller	2.10.a	lmb_bram_if_cntlr	PREFERRED	MicroBlaze	PERIPHERAL
Multi-Port Memory Controller	3.00.a	mpmc	PREFERRED	PowerPC & MicroBlaze	PERIPHERAL
OPB BRAM Controller	1.00.a	opb_bram_if_cntlr	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL
OPB DDR SDRAM Controller	2.00.a	opb_ddr	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB DDR SDRAM Controller	2.00.b	opb_ddr	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB DDR SDRAM Controller	2.00.c	opb_ddr	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL
OPB External Memory Controller	2.00.a	opb_emc	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR SDRAM Controller	1.00.a	mch_opb_ddr	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR SDRAM Controller	1.00.b	mch_opb_ddr	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR SDRAM Controller	1.00.c	mch_opb_ddr	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR2 SDRAM Controller	1.00.a	mch_opb_ddr2	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR2 SDRAM Controller	1.01.a	mch_opb_ddr2	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel DDR2 SDRAM Controller	1.02.a	mch_opb_ddr2	EARLY_ACCESS	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel External Memory Controller	1.00.a	mch_opb_emc	DEPRECATED	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel External Memory Controller	1.01.a	mch_opb_emc	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL
OPB Multi-Channel SDRAM Controller	1.00.a	mch_opb_sdram	AVAILABLE	PowerPC & MicroBlaze	PERIPHERAL

10.3. GPIO の追加

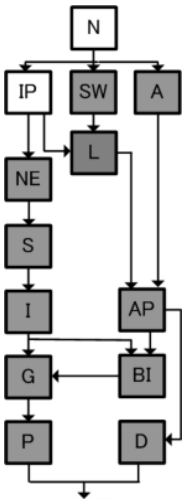
ISE で単色 LED(D1)を点灯させましたが、EDK でも単色 LED を点灯させてみます。

10.3.1. GPIO の接続

SUZAKU のデフォルトに GPIO を追加して、単色 LED(D1)を点灯させるアプリケーションを製作します。GPIO は SZ010,SZ030,SZ130,SZ310 の場合 OPB バスに、SZ410 の場合 PLB バスに接続し、GPIO の出力を単色 LED に接続します。

先ほど"C:\suzaku"の下にコピーしたフォルダの名前を"sz***-add_uart_gpio"に変更して作業を進めます。

"C:\suzaku\sz***-add_uart_gpio\xps_proj.xmp"をダブルクリックして開いてください。



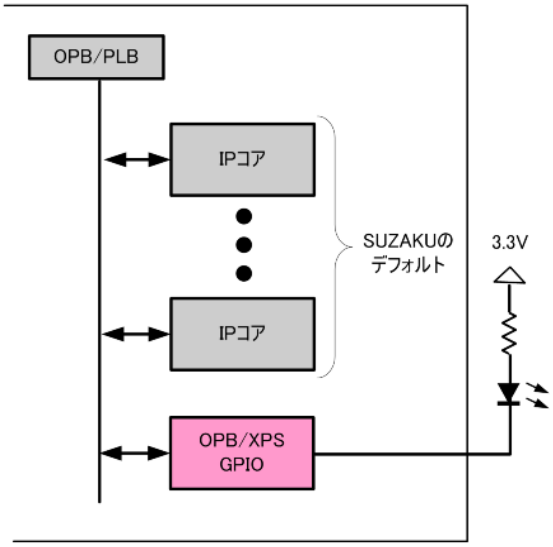


図 10.41. GPIO を追加して LED を点灯

10.3.2. ハードウェア設定

まずはハードウェアの設定を行います。

ここでの設定は Project タブの Project Files→MHS File: xps_proj.mhs に反映されます。

10.3.2.1. IP コア追加

IP コアを追加します。IP Catalog のタブをクリックしてください。IP Catalog には EDK に登録されている IP コアやユーザが登録した IP コアの一覧が表示されます。ここから使いたい IP コアを選択し、追加することができます。

General Purpose IO の中にある SZ010,SZ030,SZ130.SZ310 の場合 opb_gpio を SZ410 の場合 xps_gpio を右クリックしてメニューを出し、Add IP を選択してください。IP コアが追加されます。

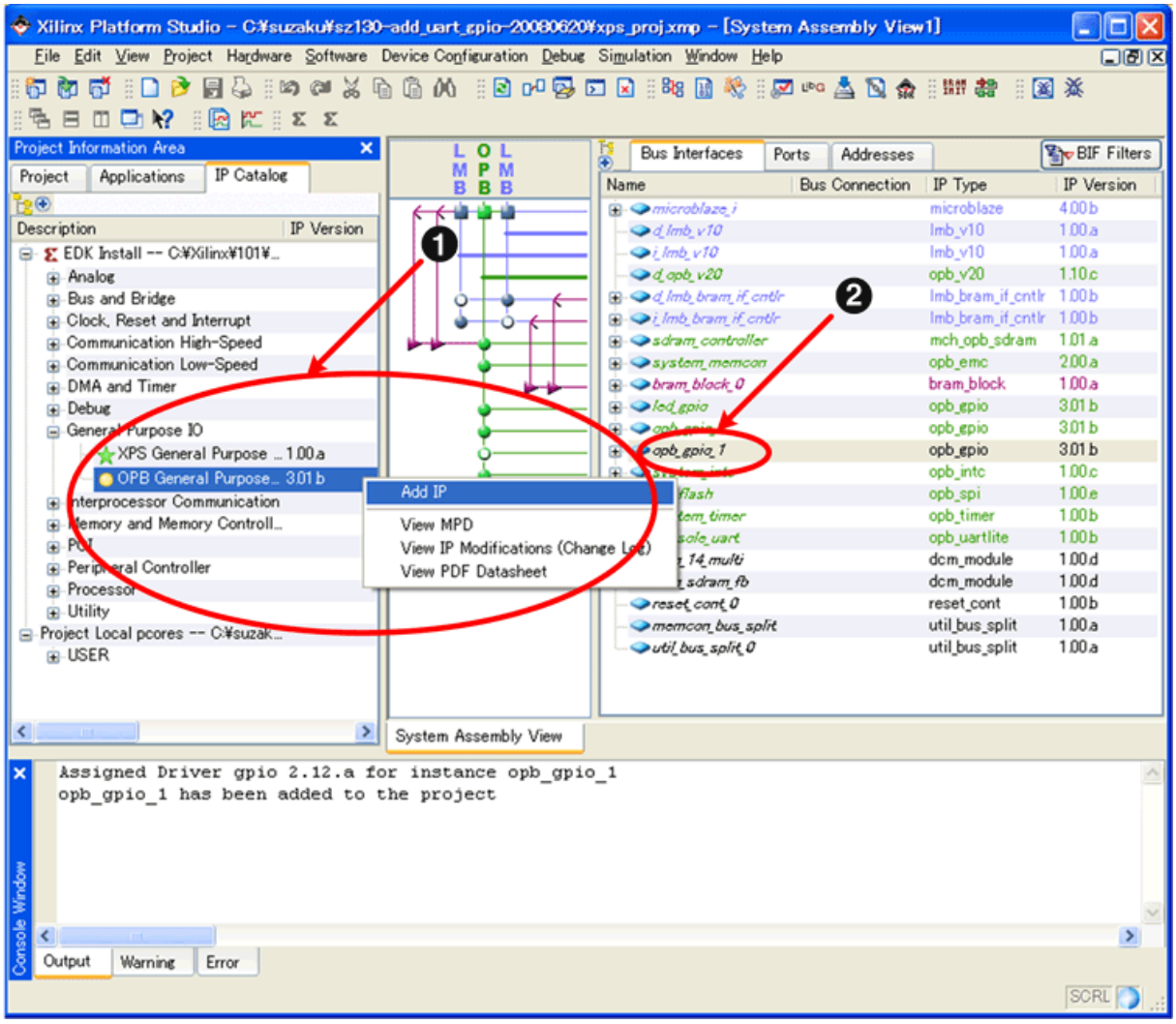


図 10.42. opb/xps_gpio の追加

❶ opb | xps_gpio を右クリックしてメニューを出し Add IP を選択

❷ IP コアが追加される

10.3.2.2. バスに接続

バスに GPIO を接続します。SZ010,SZ030.SZ130.SZ310 の場合は OPB バス、SZ410 の場合は PLB バス(plb_peripheral)に接続します。Bus Interface を選択し、追加した GPIO の横の丸をクリックしてください。○→●

これでバスに GPIO が接続されます。

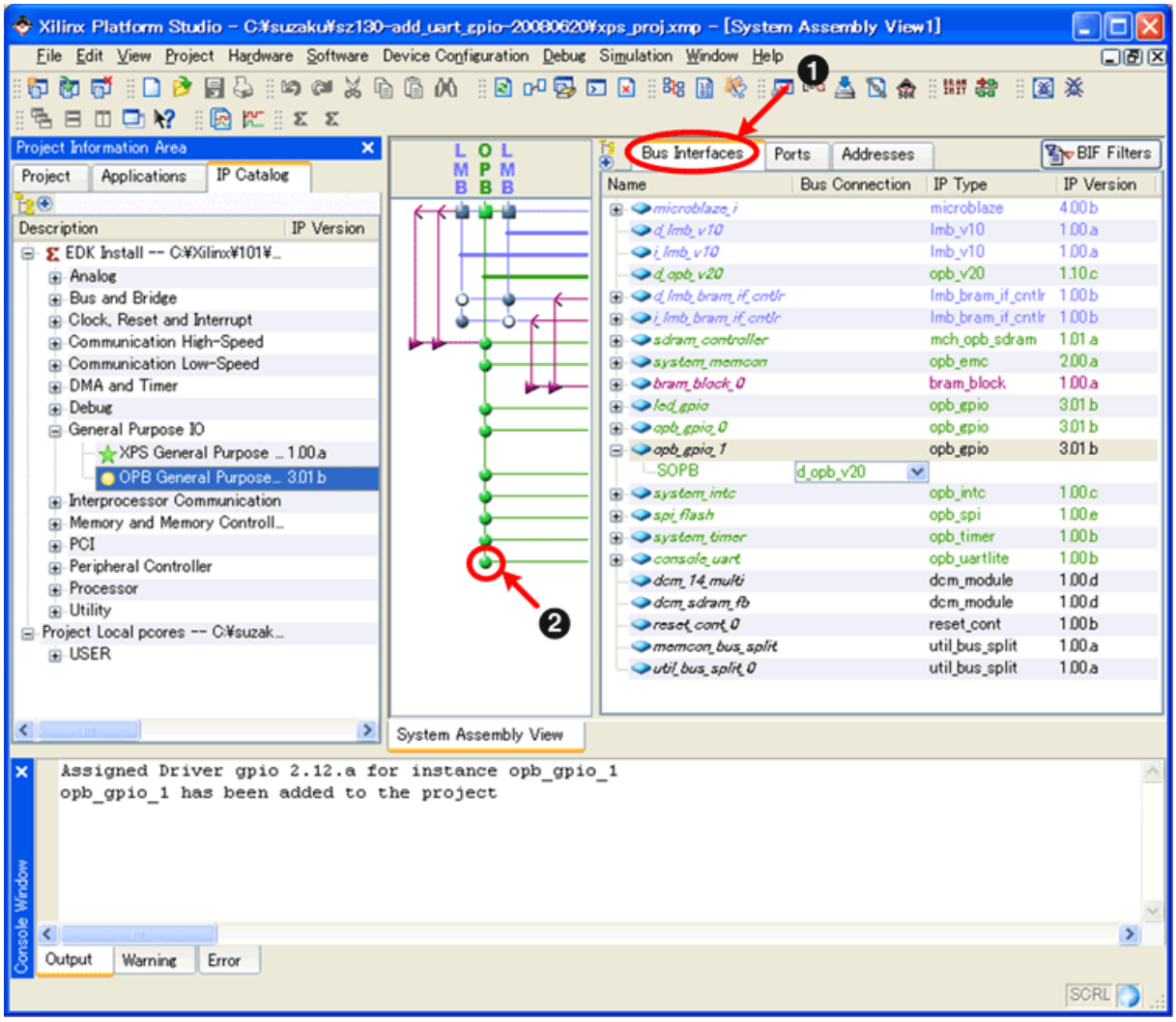


図 10.43. バスに接続

❶　Bus Interface を選択

❷　白丸をクリック○→●

10.3.2.3. IP コアの設定

IP コアはさまざまな設定をすることができます。今回追加した GPIO では GPIO の本数、出力の属性、プロセッサから制御する際の BaseAddress などを設定することができます。

GPIO を右クリックしてメニューを出し、[Configure IP]を選択してください。

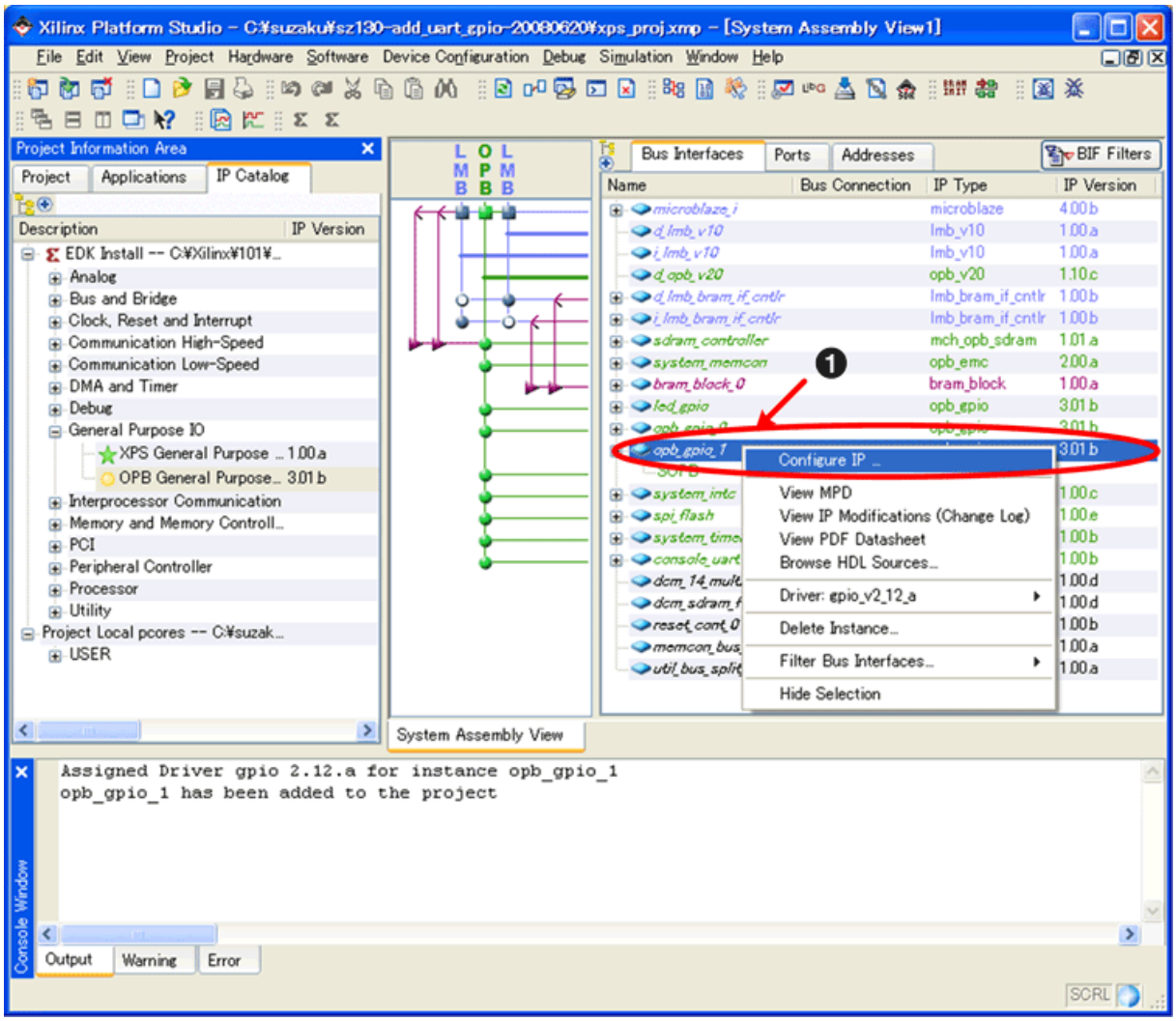


図 10.44. Configure IP

❶　右クリックしてメニューを出し、Configure IP を選択

単色 LED を 1 つだけ光らすので、バスの幅は 1 ビットにします。

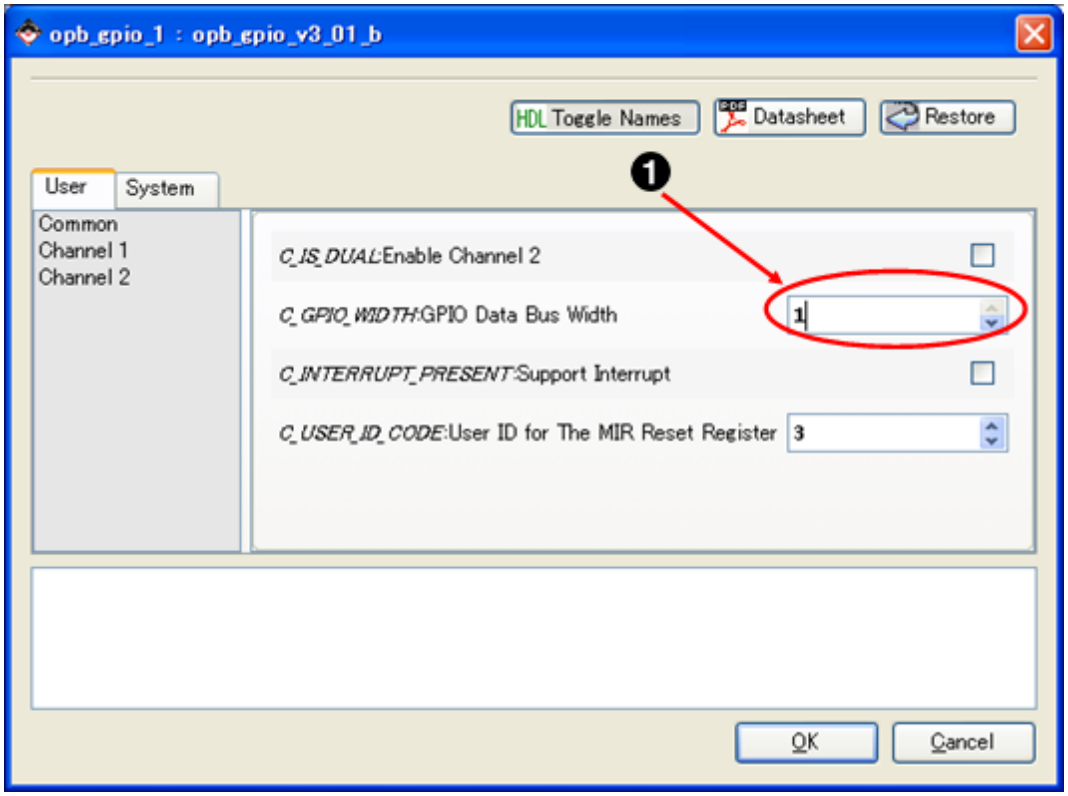


図 10.45. バス幅の設定

❶ 1 にする

Channel1 を選択し、Bi-directional を[FALSE]にしてください。

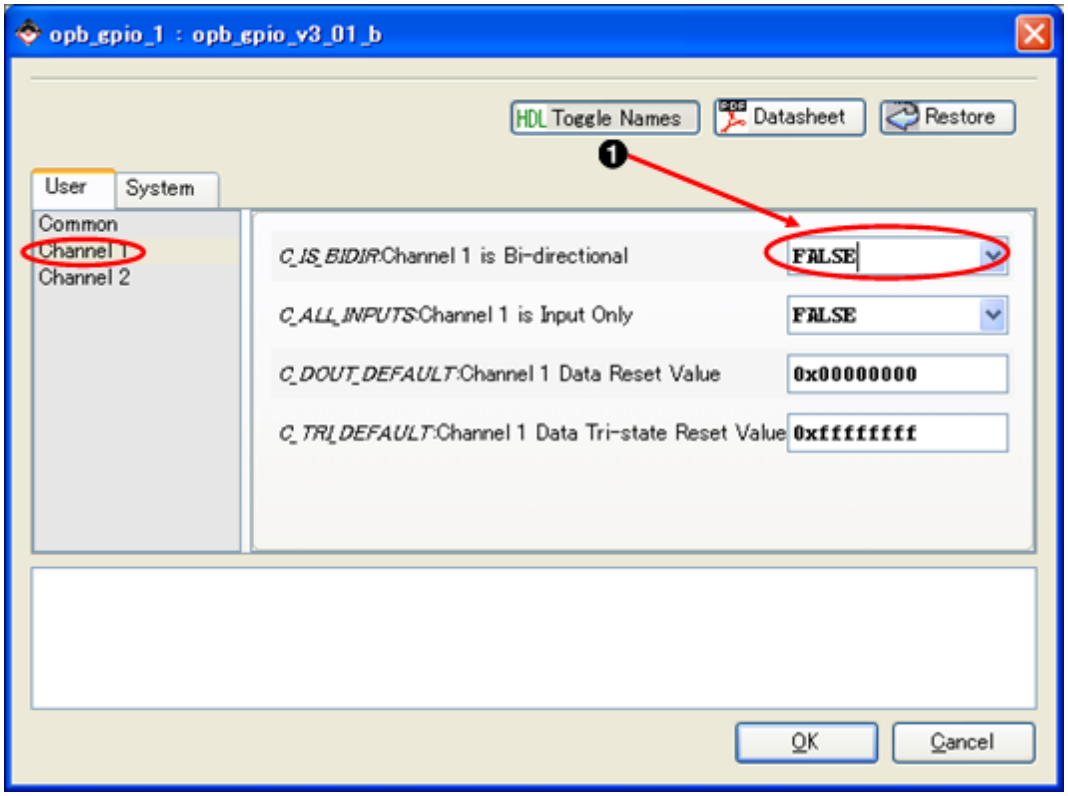


図 10.46. その他設定変更

❶　FALSE に変更

System タブをクリックし、[Base Address]、[High Address]にそれぞれメモリアドレスを入力して、[OK]をクリックしてください。メモリアドレスは SUZAKU のメモリマップで Free と書いてあるところに割り当てます(「1.4. メモリマップ」参照)。メモリアドレスは同じバスにつながっている周辺回路を CPU が見分けるために使用する大事な値です。この値が任意に決められていることで、CPU や他のコアが通信できるようになります。

表 10.3. GPIO メモリアドレス

	SZ010、SZ030、SZ130	SZ310、SZ410
Base Address	0xFFFFA400	0xF0FFA400
High Address	0xFFFFA5FF	0xF0FFA5FF

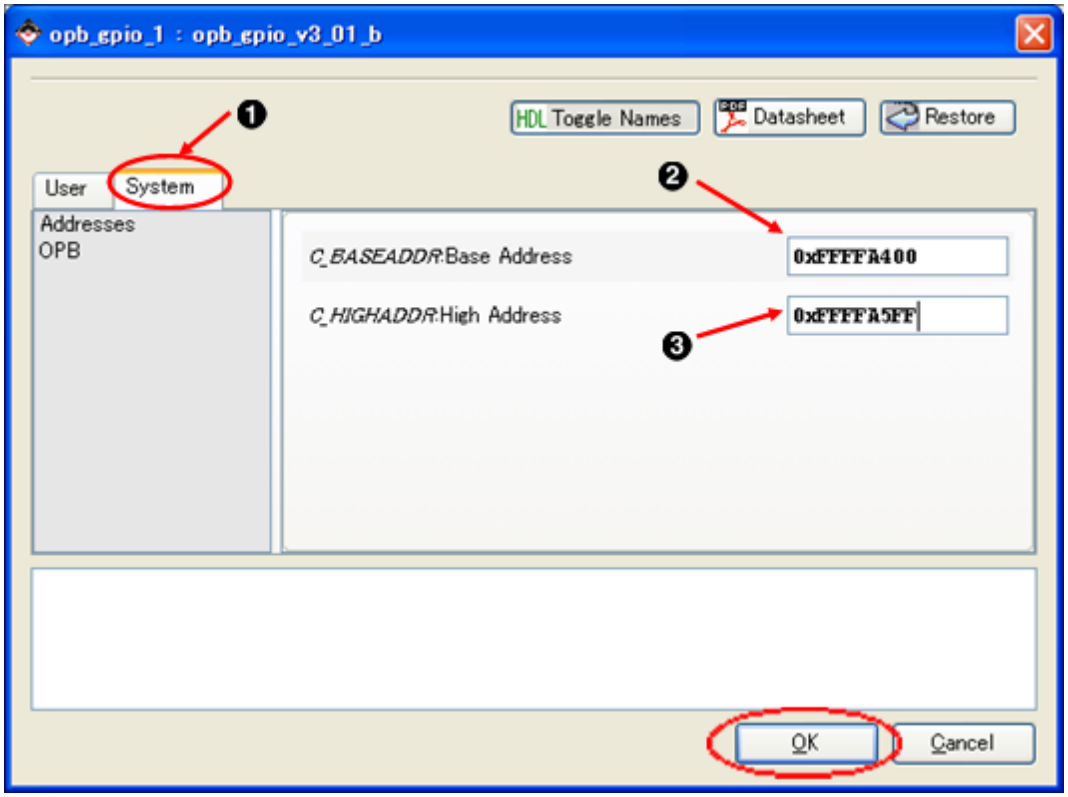


図 10.47. メモリアドレス設定

❶ System タブをクリック

❷ Base Address を入力

❸ High Address を入力

IP コアの詳細を知りたい時は、[Datasheet]をクリックしてください。データシートが表示されます。

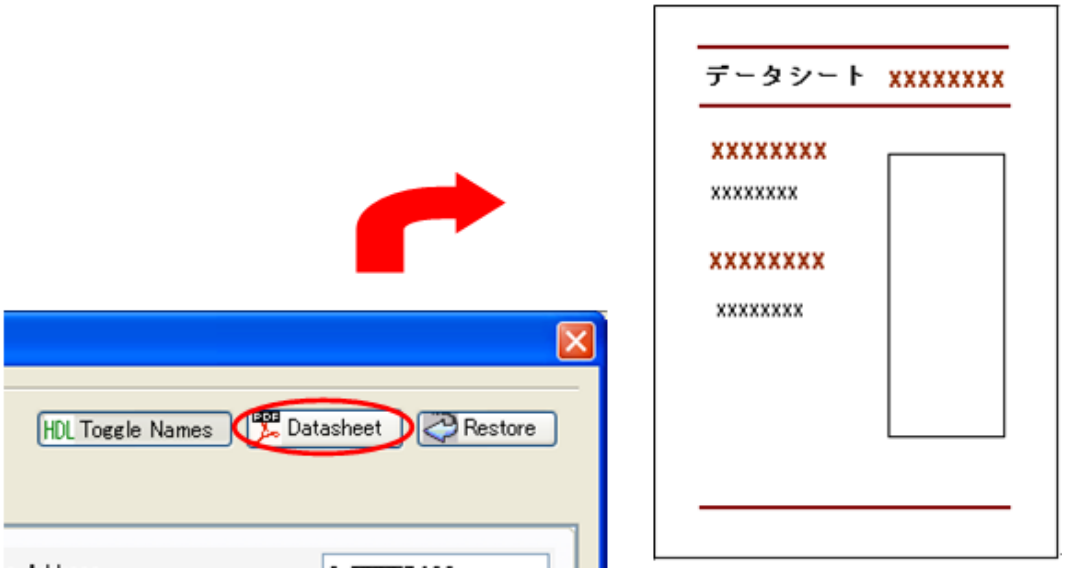


図 10.48. データシートの出し方

設定が出来たら[OK]をクリックして下さい。

10.3.2.4. メモリマップ確認

Addresses を選択し、BaseAddress と High Address と Size に間違いがないか確認してください。

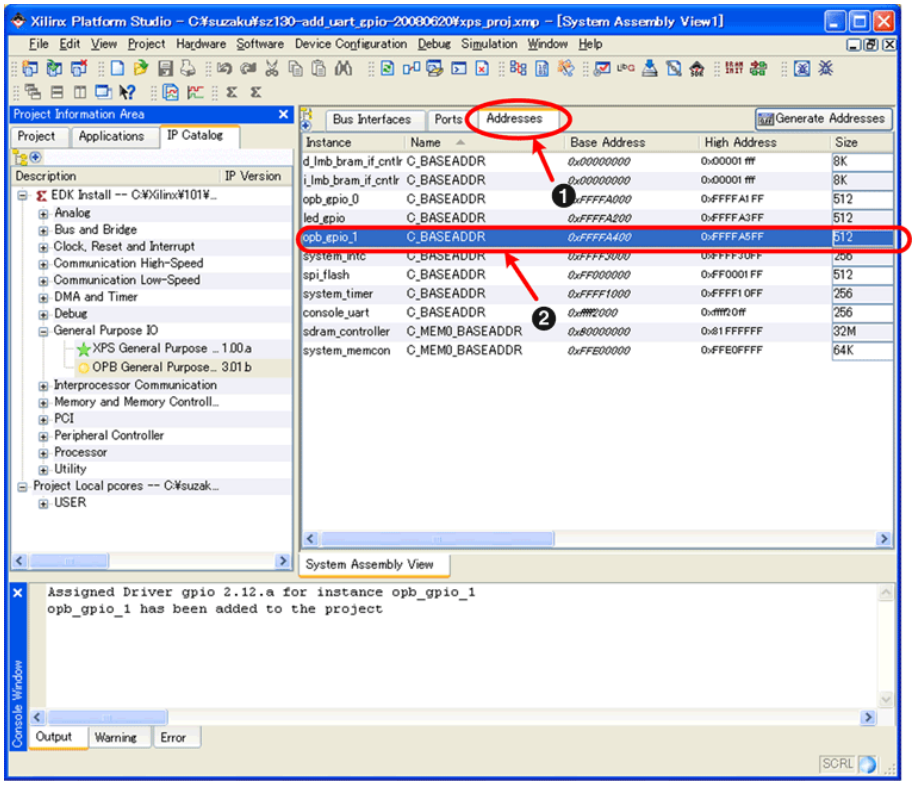


図 10.49. メモリマップ確認

❶　Addresses を選択

❷　メモリマップを確認

10.3.2.5. 信号の定義

IP コアのバス以外への接続の指定をします。External Ports に登録すると、FPGA 外部信号を定義することができ、それ以外は内部信号になります。

Ports を選択し、GPIO の⊞をクリックして開いてください。GPIO_d_out の Net の部分をクリックすると、Net 名を入力することが出来ます。Net 名は何でも良いのですが、ここでは nLE と入力してください。欄外をクリックすると確定します。

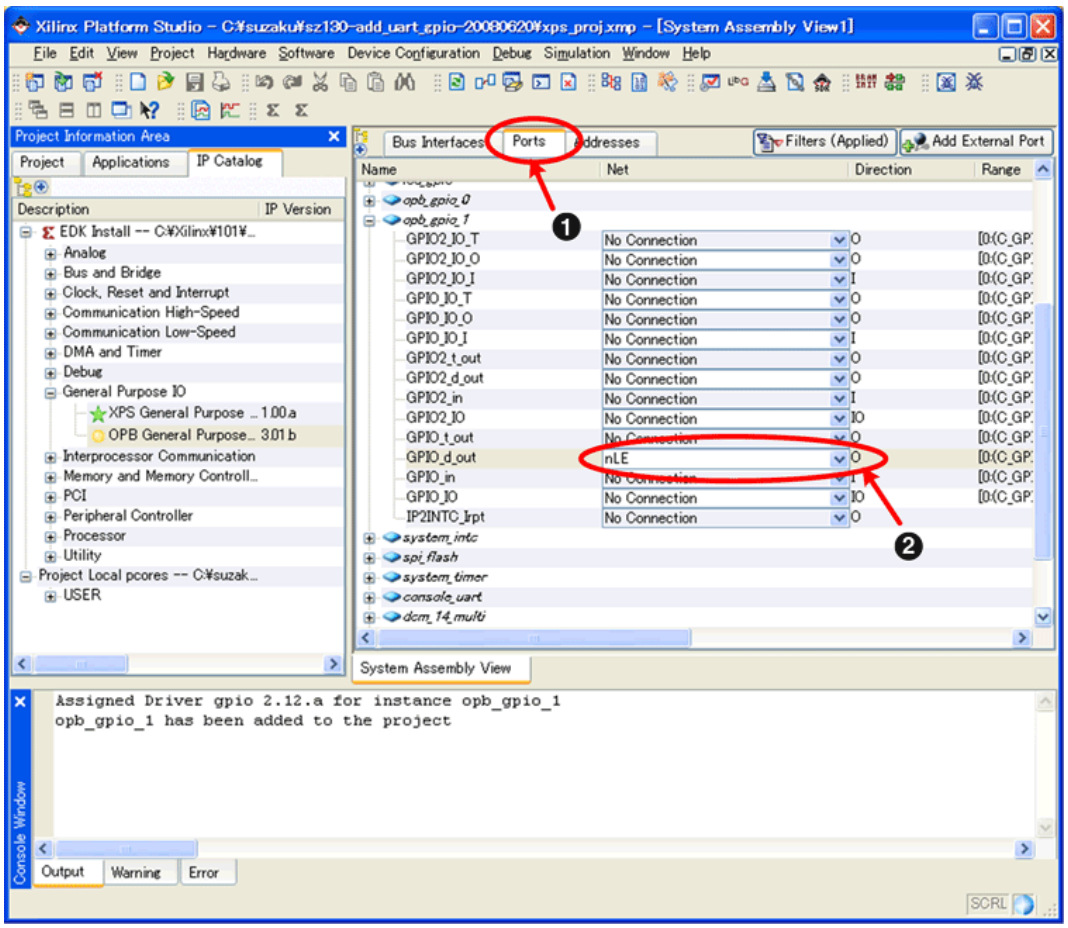


図 10.50. Net 名入力

❶　Ports を選択

❷　クリックし、nLE とネット名を入力し、確定させる

もう一度 nLE の Net の部分をクリックし、今度はをクリックし、[Make External]を選択し、確定させてください。

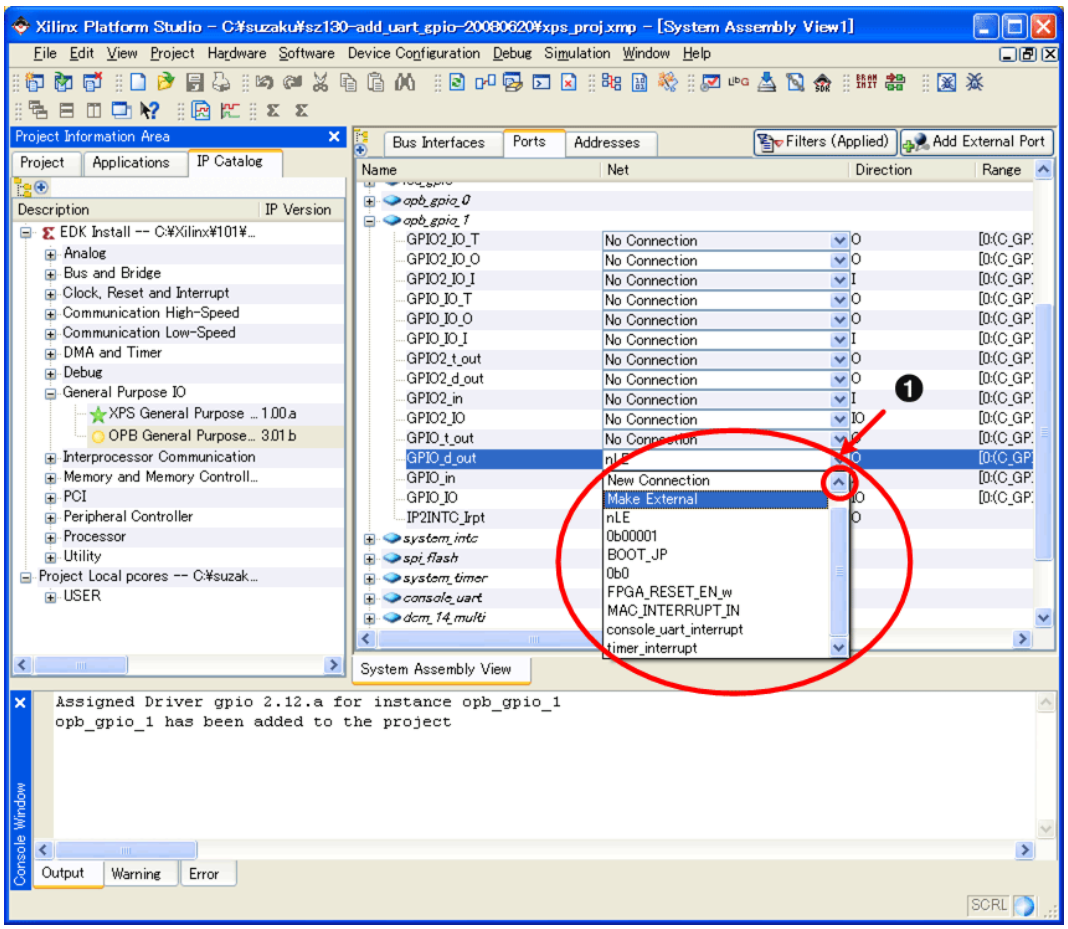


図 10.51. 外部信号にする

❶　Make External を選択し、確定させる

External Ports の⊞をクリックして開いてください。External Ports にはシステムの外部に出力する信号が定義されています。この中に Name：nLE_pin という信号が生成されているのを確認してください。

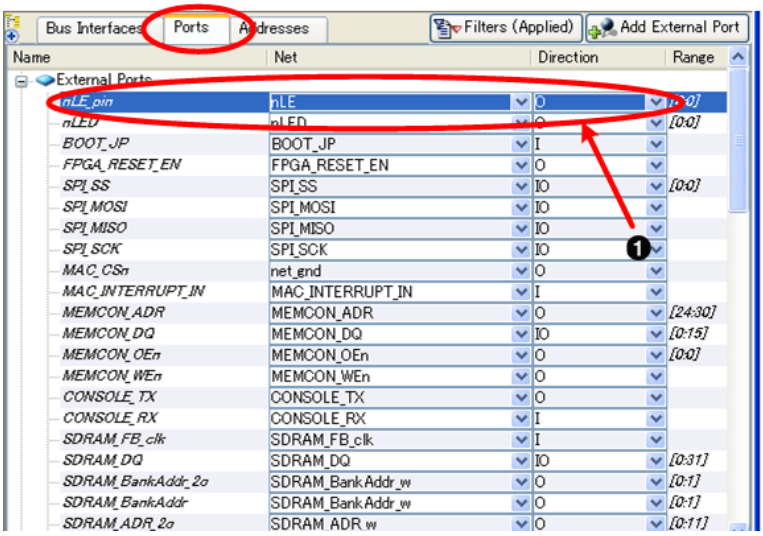


図 10.52. 信号名変更

❶　nLE_pin が生成されているのを確認

ここまでに行った作業はすべて mhs ファイルに反映されます。mhs ファイルはハードウェアの構成や設定を記述するファイルです。Project タブをクリックし、Project Files→MHS File: `xps_proj.mhs` をダブルクリックして開いてください。

例 10.1. GPIO を追加した mhs ファイルの例(SZ130)

```
PARAMETER VERSION = 2.1.0

PORT clk_in = clk_in, DIR = I, SIGIS = CLK
PORT SYS_Rst = SYS_Rst, DIR = I
# 中略
PORT nLE_pin = nLE, DIR = O, VEC = [0:0]          #外部への出力信号定義

# 中略

BEGIN opb_gpio                                    #IP コア名
 PARAMETER INSTANCE = opb_gpio_1                  #インスタンス名(Name のところ)
 PARAMETER HW_VER = 3.01.b                        #バージョン
 PARAMETER C_GPIO_WIDTH = 1                       #1 ビットに設定
 PARAMETER C_IS_BIDIR = 0                         #GPIO_IO を Input として使用しないので 0
 PARAMETER C_BASEADDR = 0xFFFFA400                #メモリマップの設定(BASE ADDRESS)
 PARAMETER C_HIGHADDR = 0xFFFFA5FF                #メモリマップの設定(HIGH ADDRESS)
 BUS_INTERFACE SOPB = d_opb_v20                   #バスに接続
 PORT GPIO_d_out = nLE                            #External Port に接続
END
```

10.3.2.6. ピンアサイン

Project タブをクリックし、UCF File: `data/xps_proj.ucf` をダブルクリックして開いてください。ピンアサイン設定のファイルが開きます。単色 LED(D1)を点灯させるため、FPGA に信号を割り当てま

す。先ほど External Ports の nLE_pin の定義の部分で、Range が[0:0]と設定されていました。Range を設定した場合は、バス幅が 1 ビットでも nLE_pin<0>のように記述します。記述できたら[File]→[Save]をクリックし、保存してください。

表 10.4. nLE_pin<0> ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nLE_pin<0>	B12	E12	L16	G2

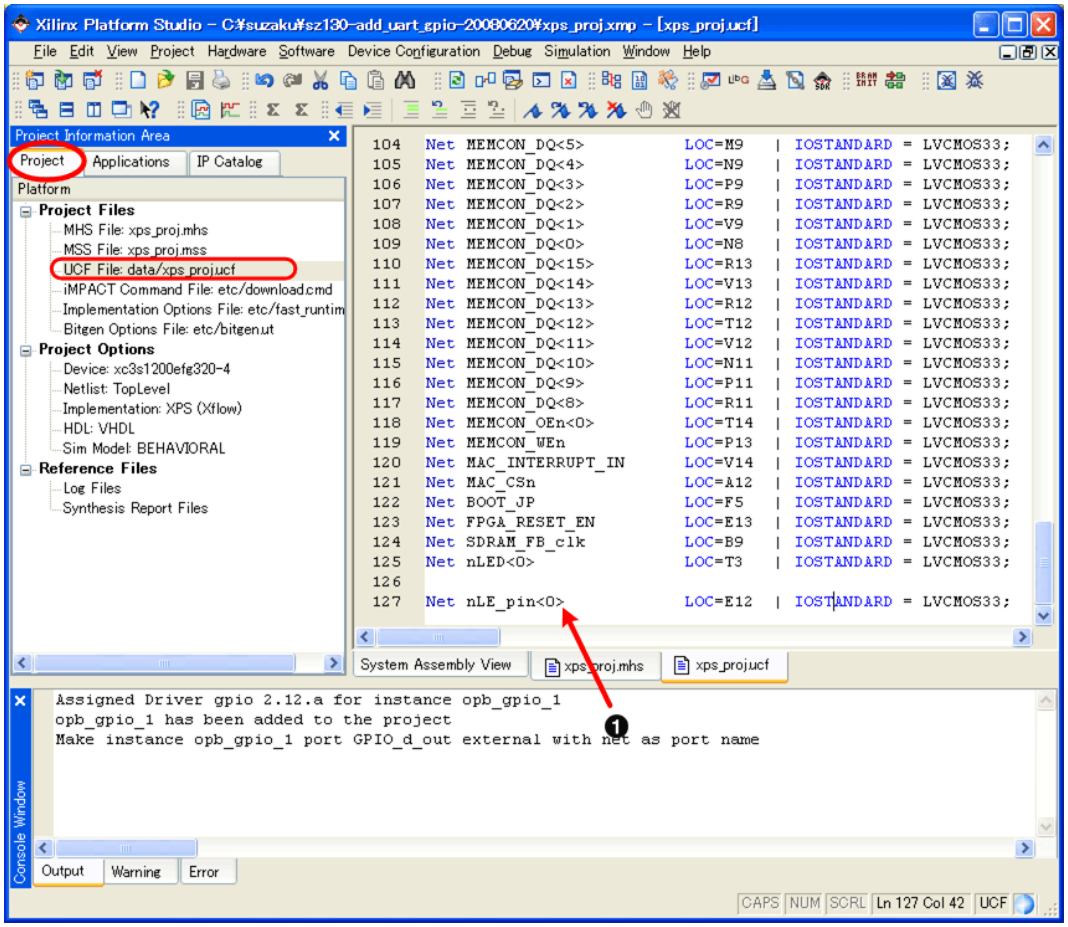


図 10.53. GPIO(xps_proj.ucf)

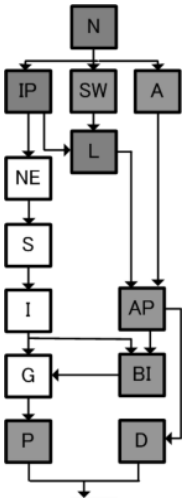
❶　ピンアサインを追記

10.3.2.7. IP コア追加 作業まとめ

1. IP コアを SUZAKU のデフォルトに追加
2. OPB バスに接続
3. 各種設定変更
4. アドレスを設定
5. 入出力信号の接続

10.3.3. ネットリスト, プログラムファイル(Hard のみ) 作成

[Hardware]→[Generate Netlist]をクリックして下さい。ネットリストが生成されます。



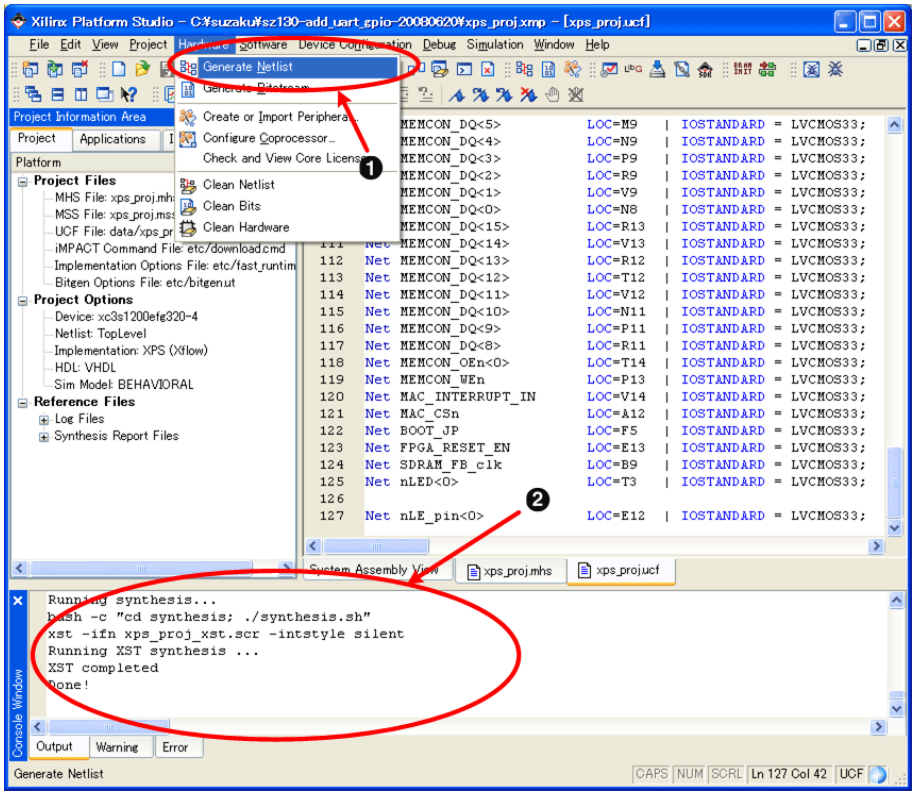


図 10.54. ネットリスト作成

❶　[Hardware]→[Generate Netlist]を選択

❷　ログが表示される

[Hardware]→[Generate Bitstream]をクリックして下さい。ソフトウェアを含まない bit ファイルが生成されます。エラーが出た場合は、今までの工程を見直してみてください。

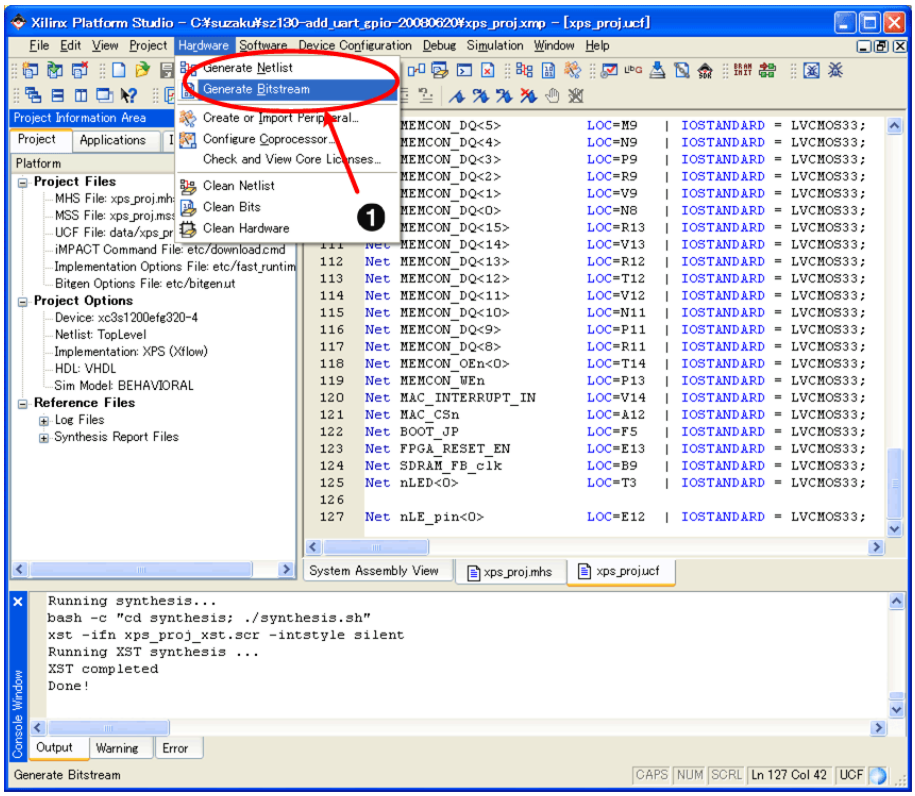


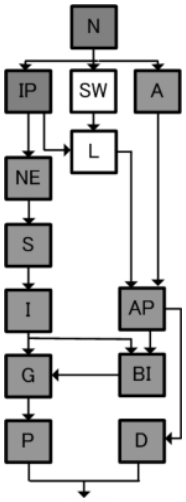
図 10.55. bit ファイル(Hard)作成

❶　[Hardware]→[Generate Bitstream]を選択

10.3.4. ソフトウェア設定

10.3.4.1. ライブラリ, ドライバ設定

[Software]→[Software Platform Settings]をクリックしてください。ここではプロセッサ、OS、およびライブラリの設定を変更することが出来ます。Drivers をクリックすると、Driver の設定が表示されます。
追加した GPIO の Driver を generic に変更し、OK をクリックしてください。



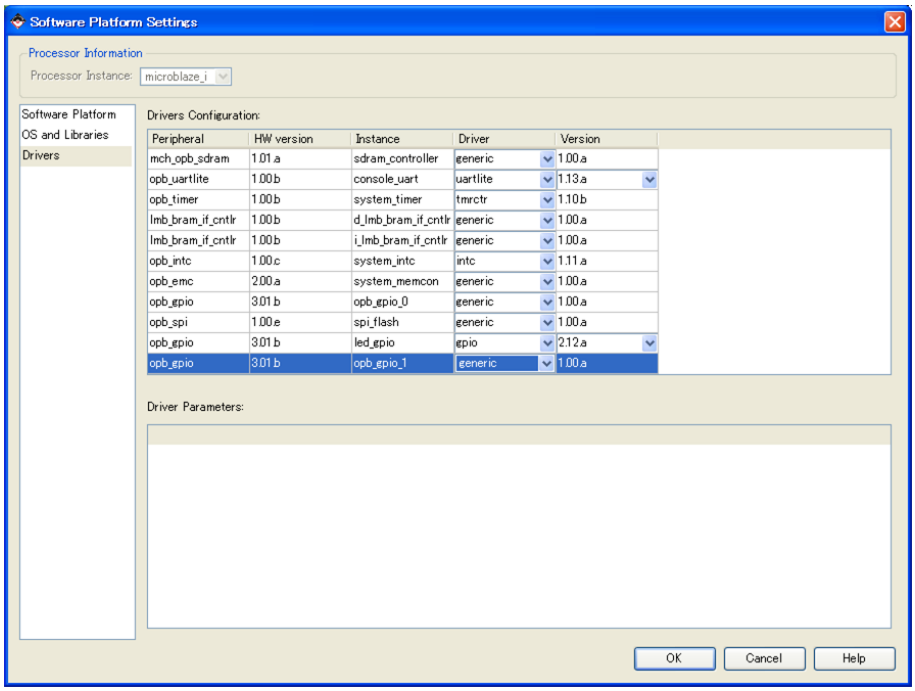


図 10.56. GPIO Driver 設定

ここで設定した内容は mss ファイルに反映されます。mss ファイルはライブラリやドライバの設定を記述するファイルです。Project タブをクリックし、Project Files→MSS File: xps_proj.mss をダブルクリックして開いてください。

例 10.2. GPIO の設定を追加した mss ファイルの例(SZ130)

```
BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = opb_gpio_1
END
```

Bug Fix

Software Platform Settings を変更すると、MSS ファイルに以下のような記述が入り、エラーが出ることがあります。この記述は必要ないので、削除してください。

```
PARAMETER int_handler = , int_port = MAC_INTERRUPT_IN
```

10.3.4.2. ライブラリ, ドライバ生成

[Software]→[Generate Libraries and BSPs] をクリックして下さい。

ライブラリや様々な設定を定義したヘッダファイルが出来上がります。

xparametaers.h を開いてください。xparameters.h にはシステムのアドレスマップが定義されます。

先ほど設定した GPIO の BASEADDR と HIGHADDR が自動で定義されています。後ほど BASEADDR の定義を使います。

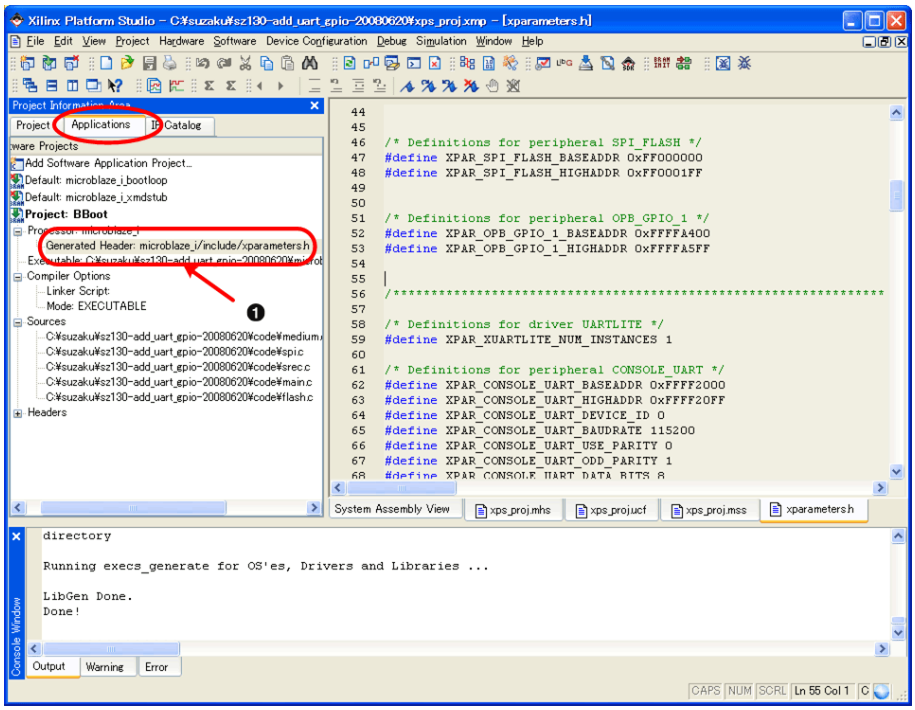
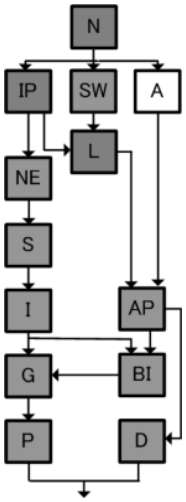


図 10.57. xparameters.h

❶　xparameters.h をダブルクリックして開く

10.3.5. アプリケーション編集

単色 LED を光らせるアプリケーションを作成します。
Applications タブをクリックしてください。Add Software Application Project を右クリックし、Add Software Application Project をクリックして下さい。



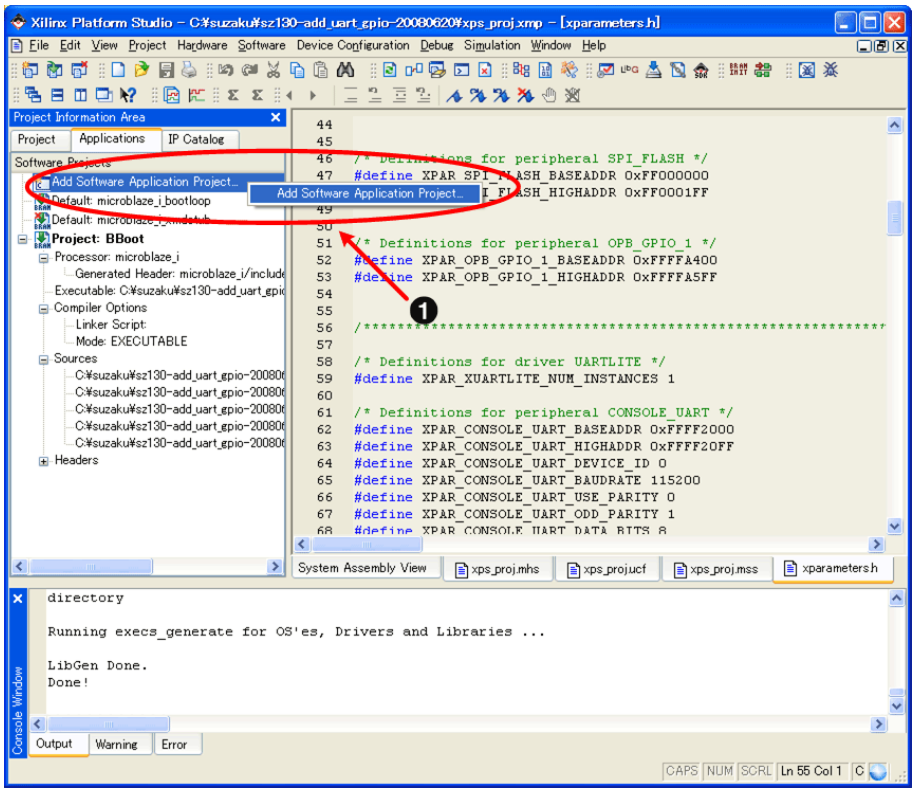


図 10.58. アプリケーション作成

❶ 右クリックして、Add Software Application Project を選択

下図が立ち上がるので、アプリケーションのプロジェクトの名前を入力します。ここでは hello-led とします。

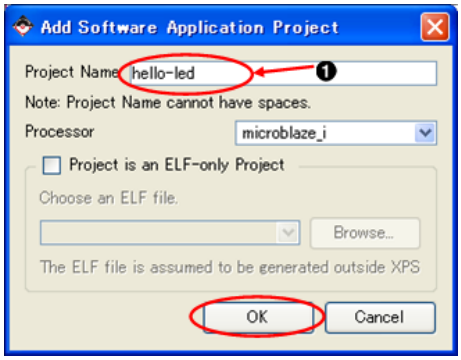


図 10.59. アプリケーションのプロジェクト名

❶ プロジェクトの名前入力

Project : hello-led が出来上がります。Sources を右クリックしメニューの Add New File をクリックして下さい。

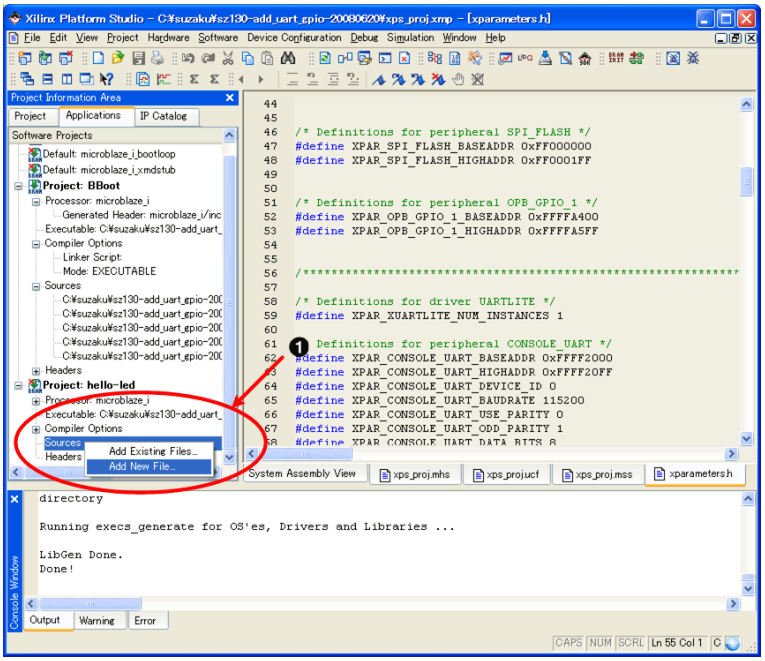


図 10.60. New File 作成

❶　hello-led の Sources を右クリックしてメニューを出し、Add New File...をクリック

フォルダを作成し、作成したフォルダを開いてください。ここでは hello-led というフォルダを作成します。ファイル名に `main.c` と入力し、[保存]をクリックして下さい。

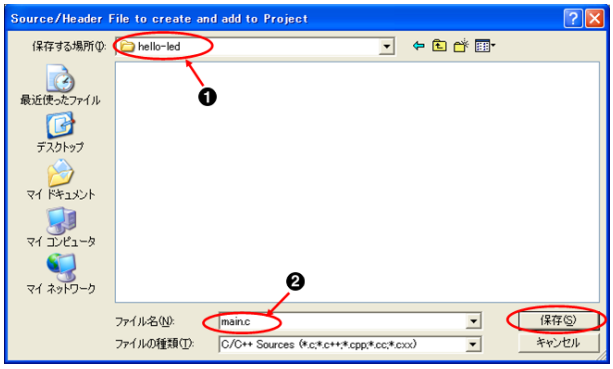


図 10.61. main.c 作成

❶　新しいフォルダ作成

❷　main.c と入力

Sources に `main.c` が作成されます。ダブルクリックしてファイルを開いてください。

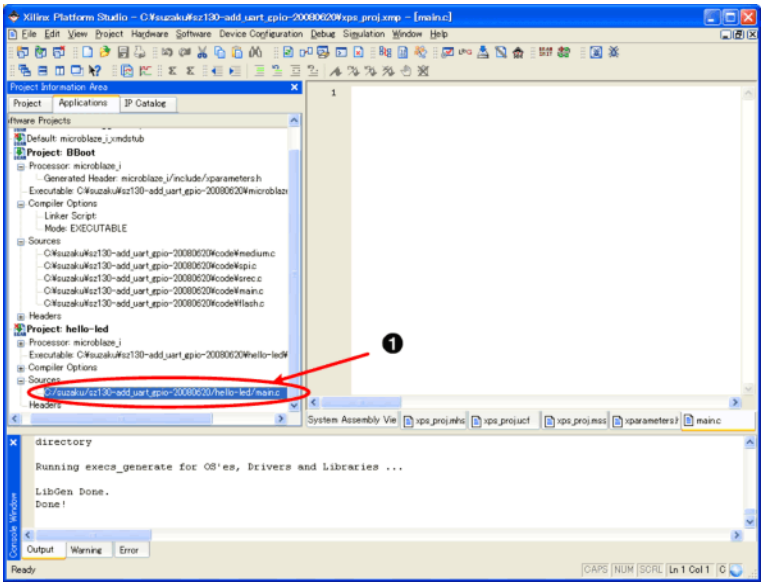


図 10.62. main.c を開く

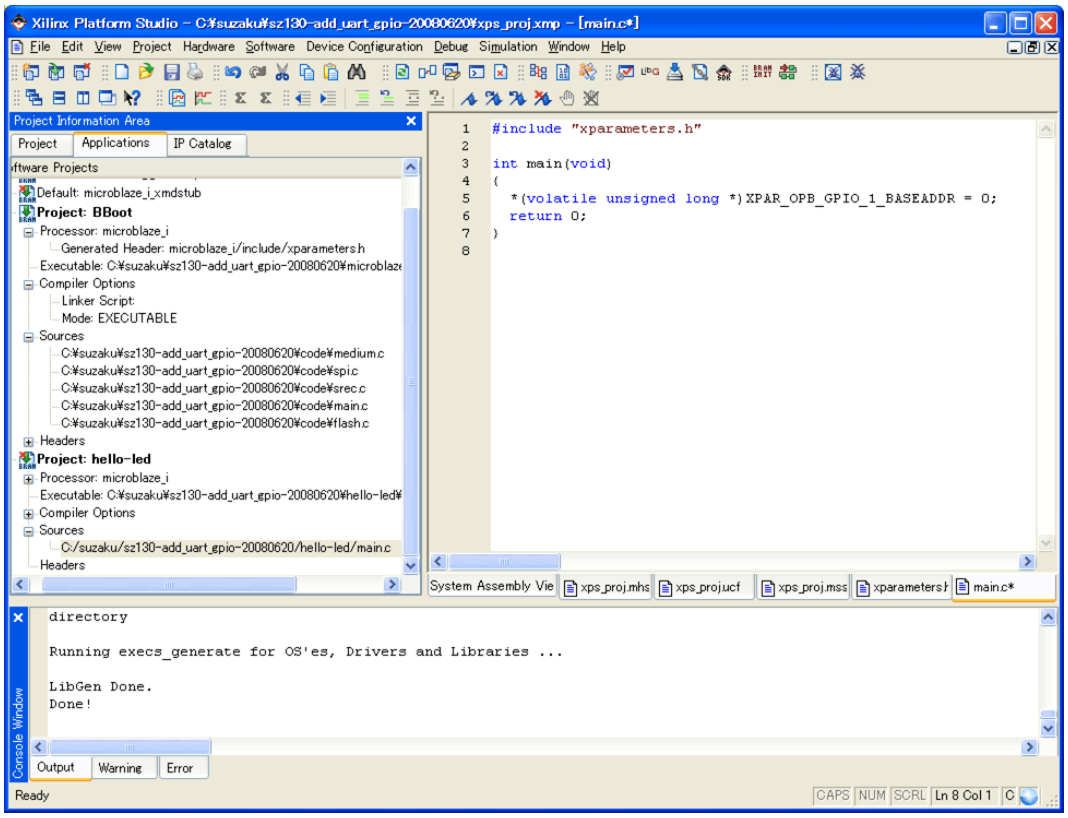
❶ main.c をダブルクリック

単色 LED を点灯するソースコードを記述します。[GPIO の BASEADDR]には先ほど `xparameters.h` で確認した GPIO の BASEADDER の define を入れてください。

インスタンス名を変更していなければ、

SZ010,SZ030,SZ130,SZ310 の場合は XPAR_OPB_GPIO_1_BASEADDR、

SZ410 の場合は XPAR_XPS_GPIO_0_BASEADDR となります。




```
#include "xparameters.h"

int main(void)
{
  *(volatile unsigned long *)[GPIO  BASEADDR] = 0;
  return 0;
}
```

図 10.63. 単色 LED 点灯のソースコード(main.c)

記述できたら[File]→[Save]を選択し、保存してください。

Project : hello-led を右クリックして、Mark to Initialize BRAMs をクリックして下さい。

チェックマークがつき、Project : hello-led の横のアイコンが  に変わります。これで hello-led が BRAM に初期値として書き込まれるようになります。

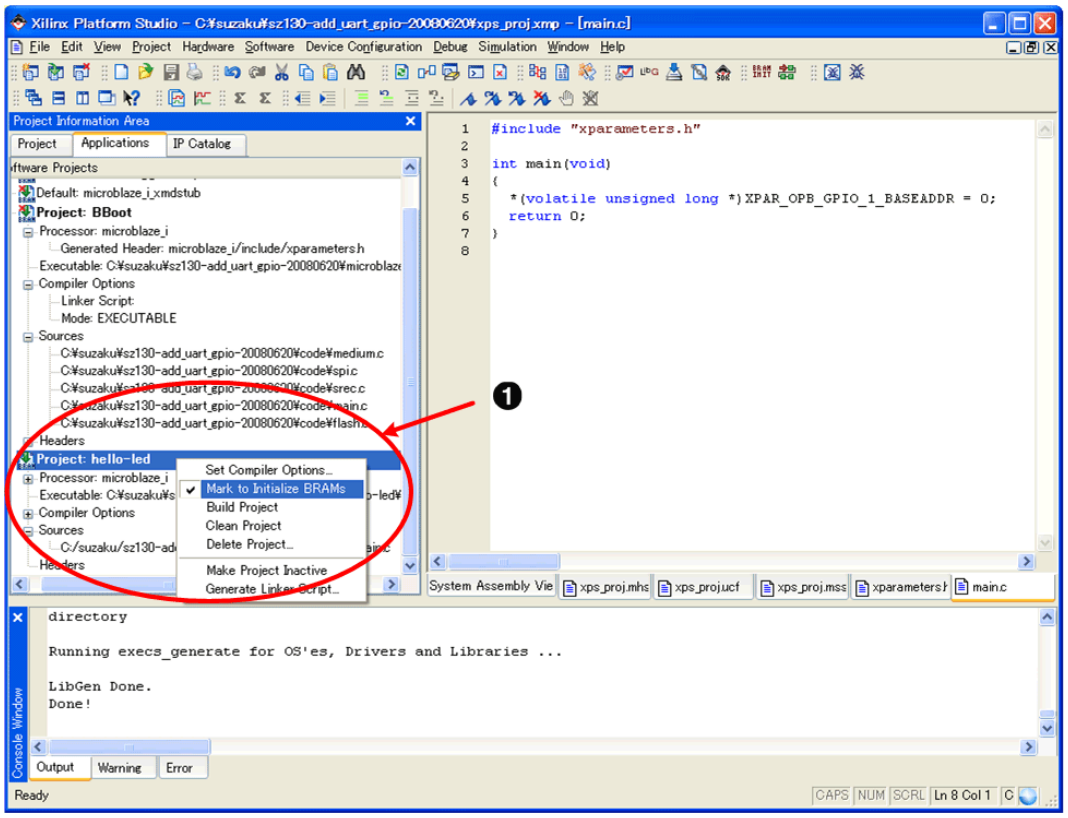


図 10.64. hello-led を書き込むように設定

❶　選択されるとチェックマークがつき、左のアイコンから✕マークがなくなる

今回 BBoot(「11.9.6. BBoot 編集」参照)は書き込みません。Project : BBoot の横のアイコンが  であることを確認してください。

10.3.5.1. リンカースクリプト設定

SZ310 SZ410

SZ310,SZ410 の場合リンカースクリプトの設定が必要です。

Project : hello-led を右クリックして、Set Compiler Options をクリックして下さい。

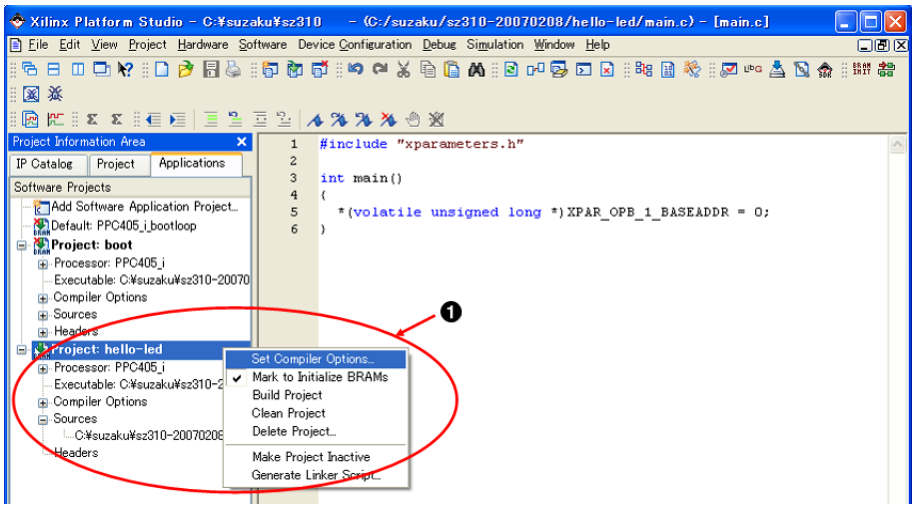


図 10.65. リンカースクリプト設定

❶ 右クリックしてメニューを出し、Set Compiler Options をクリック

Use Default Linker Script をチェックし、Program Start Address に 0xFFFFC000 と入力して[OK]をクリックして下さい。0xFFFFC000 は BRAM の Base Address で、プログラムが BRAM から始まるように設定されます。

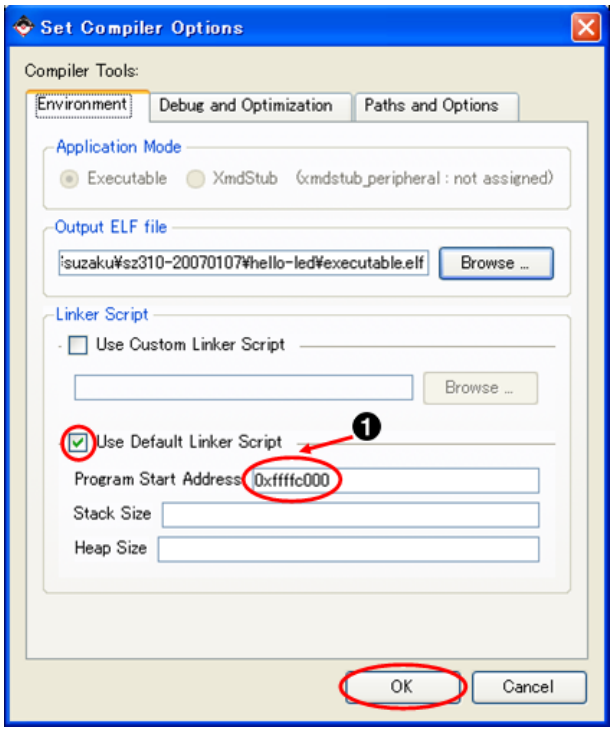
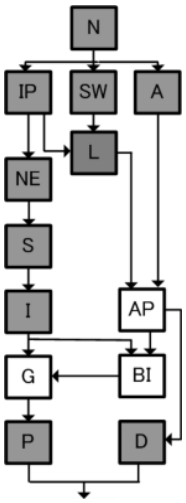


図 10.66. スタートアドレス設定

❶ 0xFFFFC000 と入力

10.3.6. アプリケーション生成

[Software]→[Build All User Applications]をクリックして下さい。コンパイラが起動され、各アプリケーションのプログラムソースの設定が読み込まれます。エラーがなければ executable.elf が"C:\suzaku\sz***-add_uart_gpio\microblaze_i|ppc405-i|ppc405_system\code"の下に出来上がります。



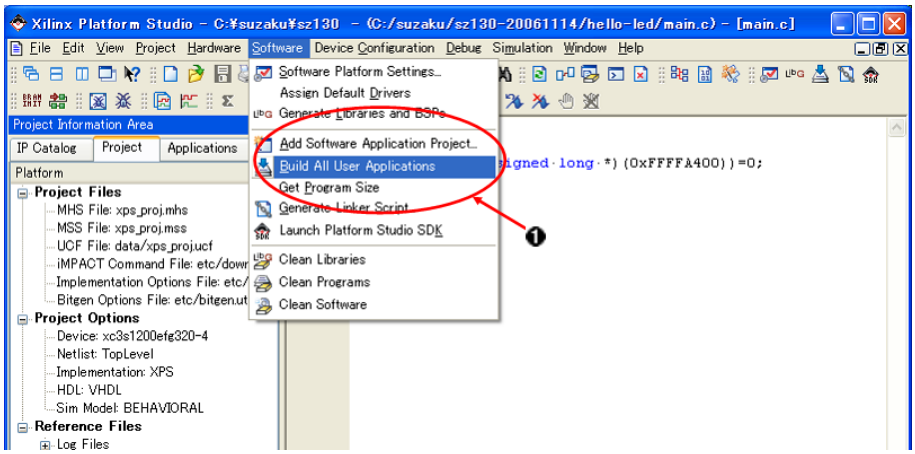


図 10.67. elf ファイル作成

❶　[Software]→[Build All User Applications]を選択

10.3.7. プログラムファイル作成

ハードウェアでつくった bit ファイルの中にアプリケーションを書き込みます。

[Device Configuration]→[Update Bitstream]をクリックしてください。bit ファイルが生成されます。

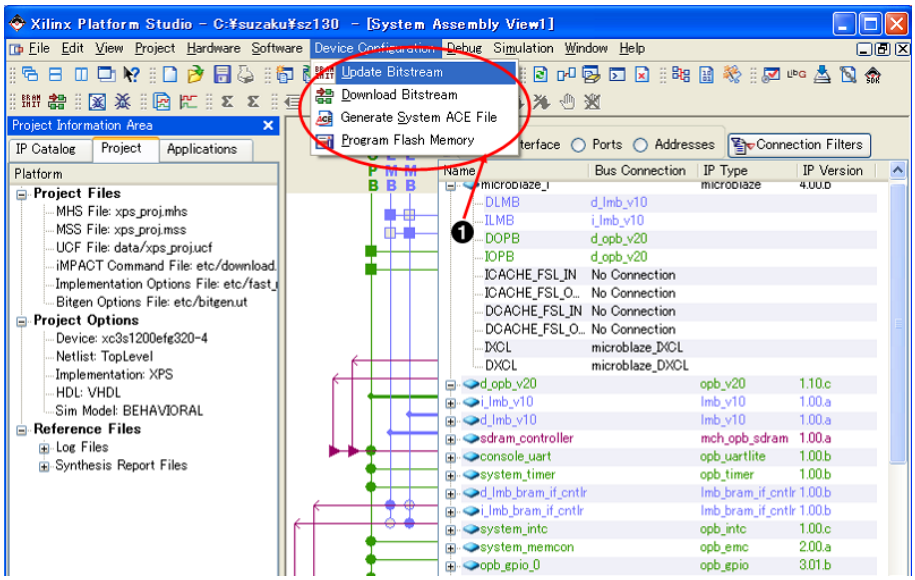


図 10.68. bit ファイル作成

❶　[Device Configuration]→[Update Bitstream]を選択

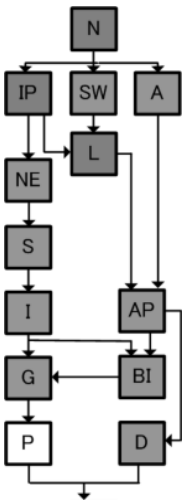
変な ERROR がでたら

エラーを修正した後、Update Bitstream や Build All User Application を実行すると、エラーを修正できてないようなエラーが出ることがありま

す。そんな時は[Hardware]→[Clean Hardware]や[Software]→[Clean Software]を実行し、クリーンしてみてください。うまくいくことがあります。

10.3.8. コンフィギュレーション

bit ファイルを JTAG でコンフィギュレーションします。SUZAKU JP2 をショートさせ、SUZAKU CON7 にダウンロードケーブルを接続し、LES/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。



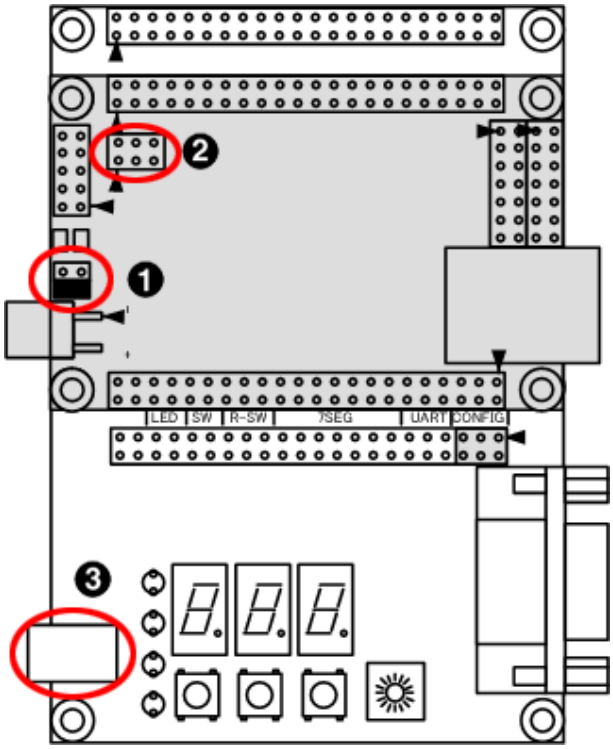


図 10.69. ジャンパの設定等

❶ JP2 ショート

❷ ダウンロードケーブル接続

❸ 電源投入

[Device Configuration]→[Download Bitstream]をクリックしてください。バッチモードの iMPACT を使用して FPGA に bit ファイルがコンフィギュレーションされます。

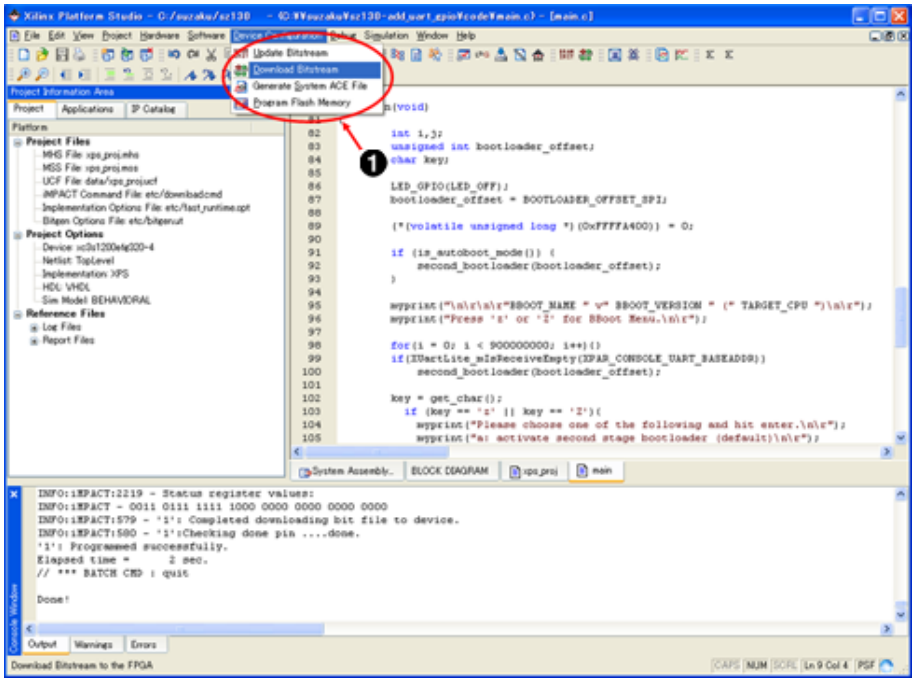


図 10.70. コンフィギュレーション

❶　[Device Configuration]→[Download Bitstream]を選択

単色 LED(D1)が光ったでしょうか？

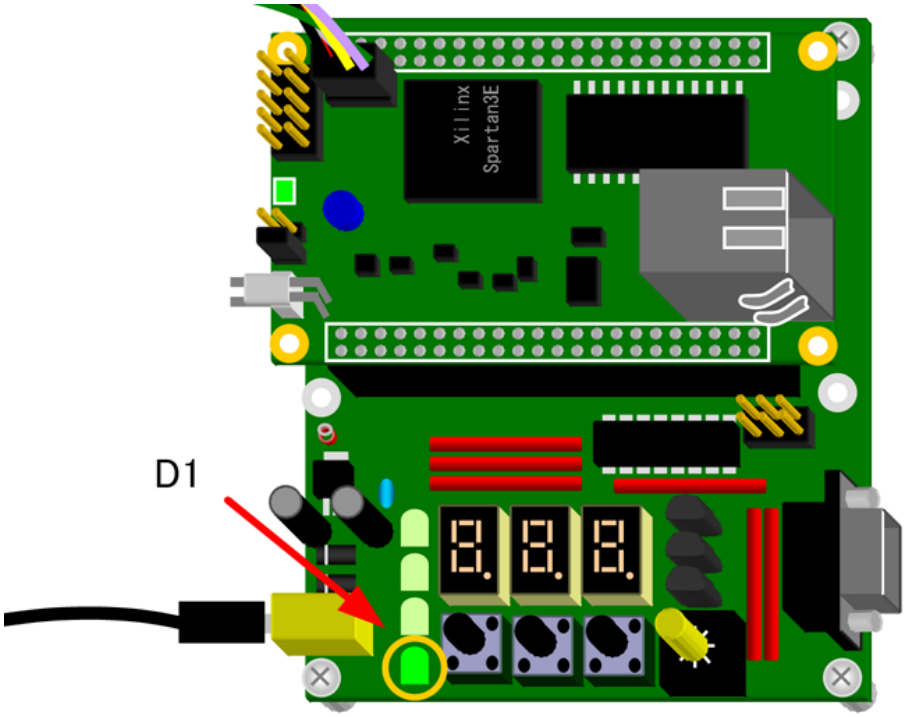


図 10.71. 単色 LED(D1)点灯

フラッシュメモリに書き込む場合は、"C:\suzaku\sz***-add_uart_gpio\implementation"フォルダ内に download.bit が出来上がっているので、これを使って下さい。

10.3.9. 空きピン処理

ISE の時と同様、D2、D3、D4 が少し光っています(SZ310,SZ410 ではほとんど光りません)。EDK で空きピン処理をしたい場合、bitgen.ut ファイルを編集します。ここに UnusedPin の設定を記述することで、終端処理を設定することが出来ます。デフォルトでこのピンは PullDown に設定されているため、SUZAKU のデフォルトでは明記していません。この設定を PullUp にすることで D2、D3、D4 は光らなくなります。ただ、空きピンから電圧が出力されているのはあまり良い状態ではないので、今回も D2、D3、D4 に信号を定義することで対処します。

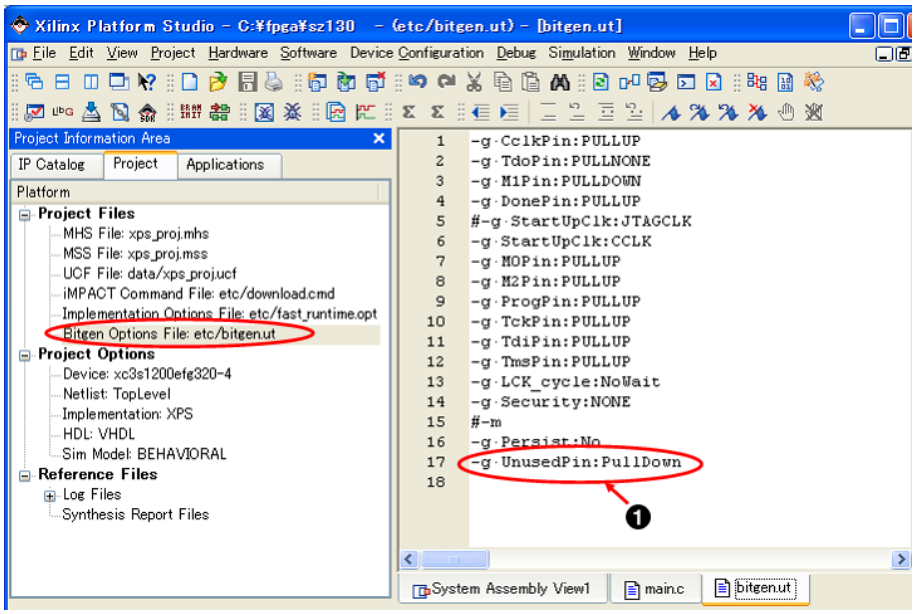


図 10.72. Bitgen のオプション設定

❶　-g UnusedPin:PullDown

Add External Port をクリックすると、External Ports に信号が追加されるので 3 つ追加してください。Name に適当な名前を記述し(ここでは nLE2_pin, nLE3_pin, nLE4_pin と記述)、Direction を O に設定し、Range に[0:0](Range は設定しなくてもかまいません。設定しなかった場合はピンアサインで<0>は不要)と記述し、Net に net_vcc と記述してください。Net に net_vcc と記述すると、'1'が出力されます。

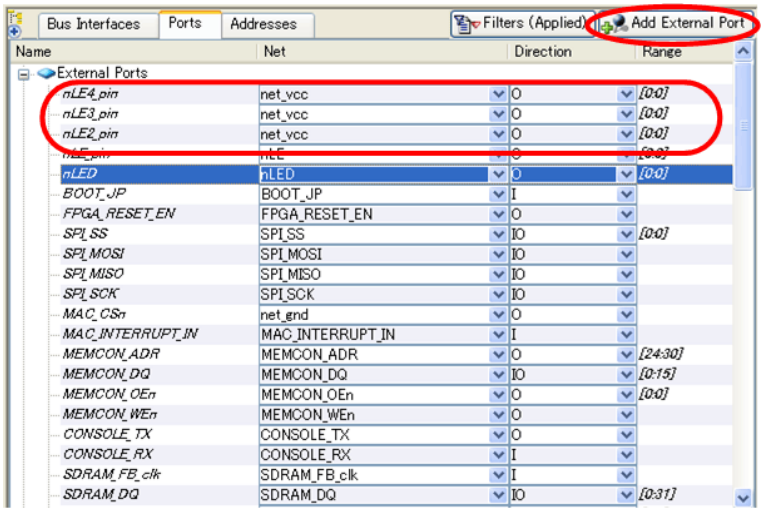


図 10.73. EDK での空きピンの処理

`xps_proj.ucf` を開き、ピンアサインを追記してください。

表 10.5. ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nLE2_pin<0>	C12	F12	L15	F2
nLE3_pin<0>	D11	B11	L14	F1
nLE4_pin<0>	E11	A11	L13	E1

Flat View

以下のような表示になって元に戻したいと思ったことはないでしょうか。右クリックしてメニューを出して Flat View のチェックをはずせば、元の表示に戻すことができます。

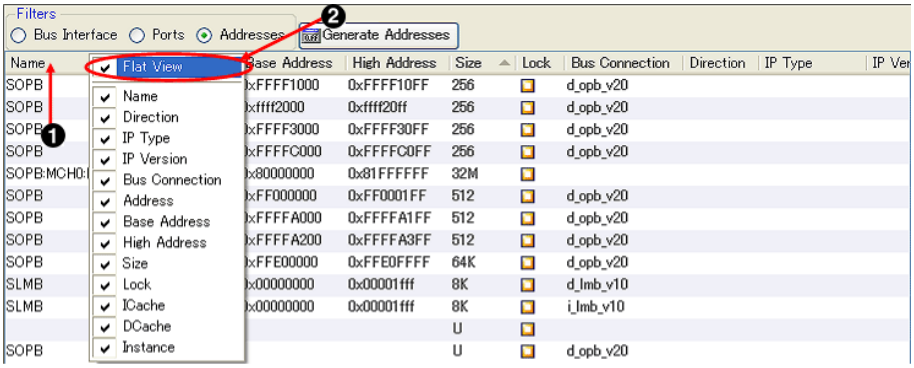


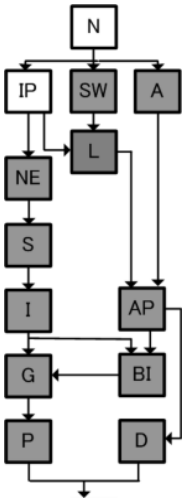
図 10.74. Flat View

❶ このあたりを右クリックしてメニューを出す

❷ Flat View のチェックをはずす

10.4. UART の追加

UART を追加し、BRAM の中のアプリケーションに受信した文字をそのまま送信するソースコードを追加します。



10.4.1. ハードウェア設定

10.4.1.1. IP コアの追加

IP Catalog のタブをクリックし、Communication Low-Speed を開いてください。SZ010、SZ030、SZ130、SZ310 の場合 opb_uartlite を、SZ410 の場合 xps_uartlite を右クリックしてメニューを出し、Add IP を選択してください。IP コアが追加されます。

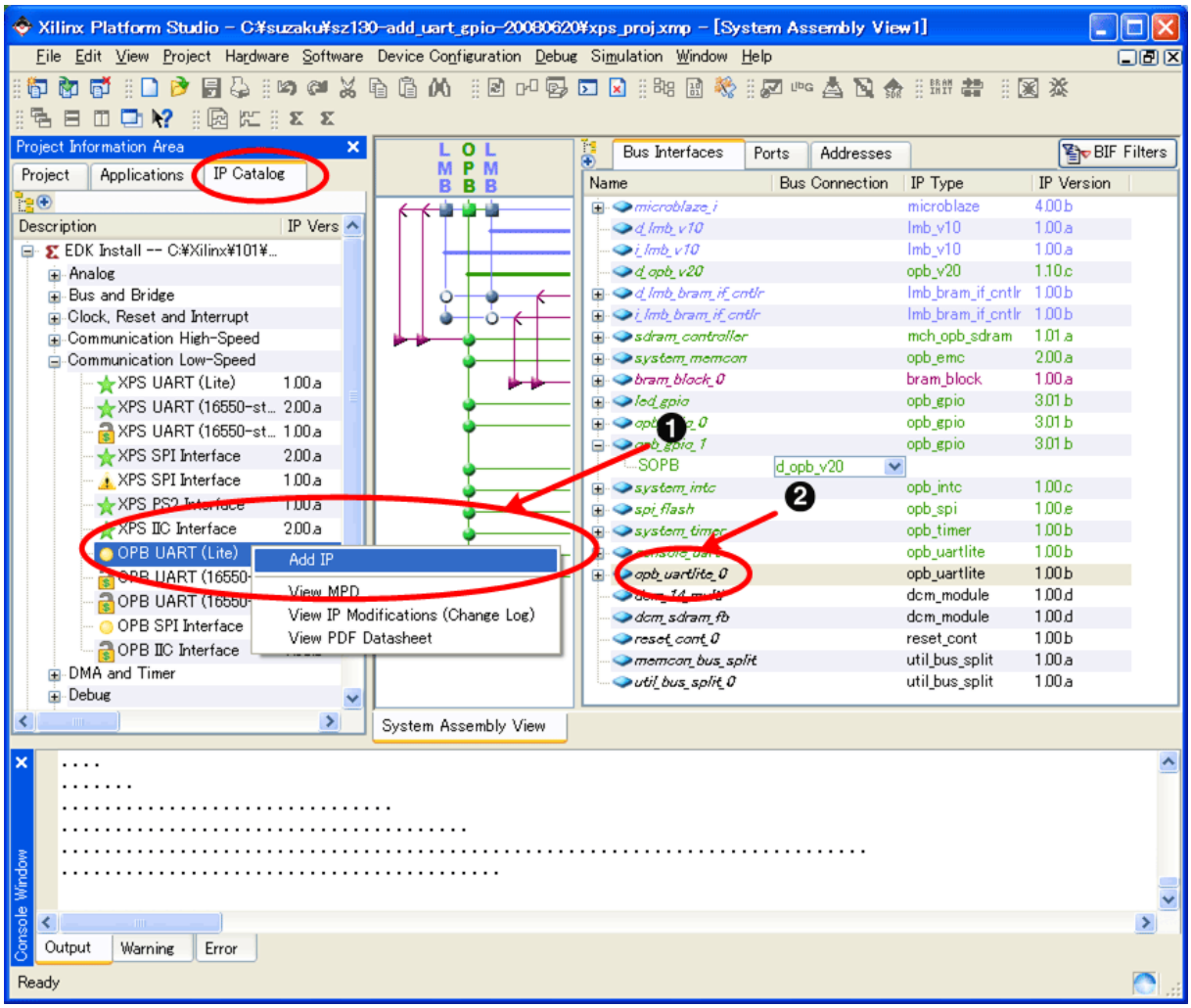


図 10.75. opb/xps_uartlite の追加

❶ opb | xps_uartlite を右クリックしてメニューを出し Add IP を選択

❷ IP コアが追加される

10.4.1.2. バスに接続

SZ010,SZ030,SZ130,SZ310 の場合 OPB バスに、SZ410 の場合 PLB バス(plb_peripheral)に接続します。

Bus Interface を選択し、追加した UART lite の横の丸をクリックしてください。○→●

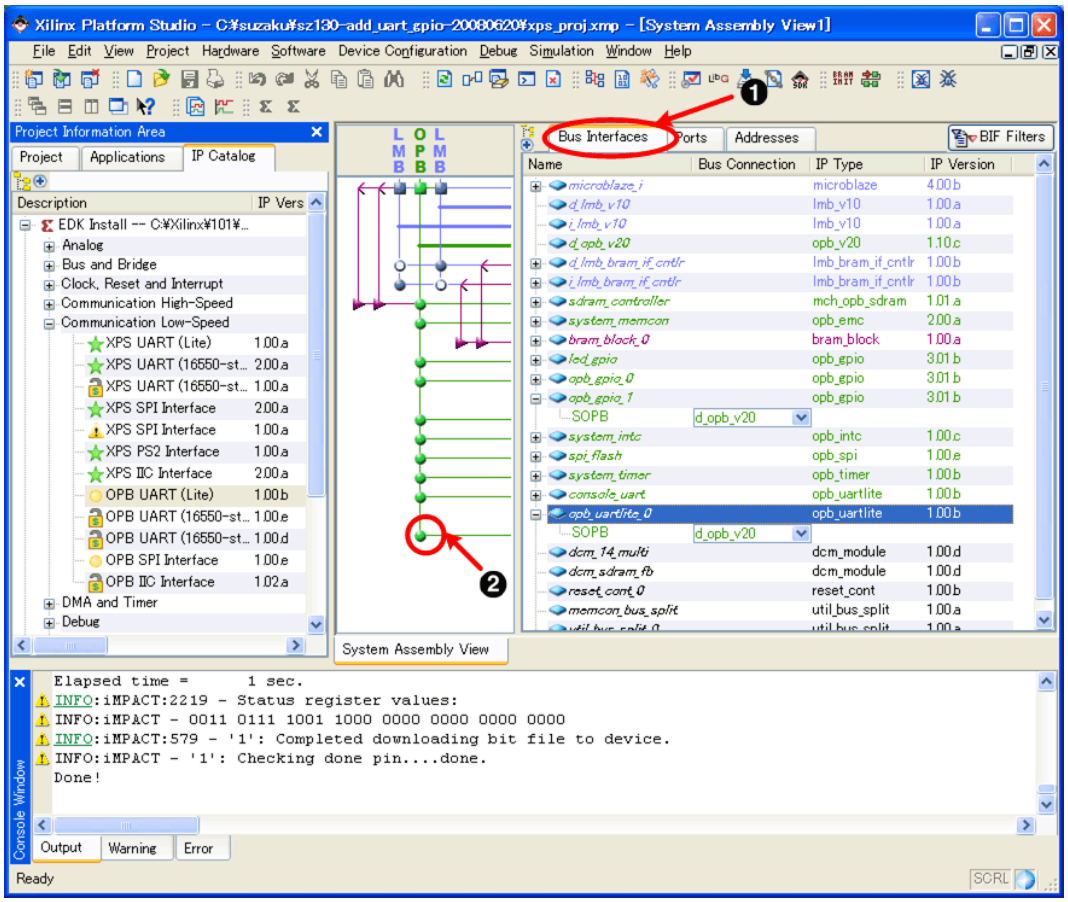


図 10.76. バスに接続

❶　Bus Interface を選択

❷　白丸をクリック○→●

10.4.1.3. IP コアの設定

UART lite を右クリックしてメニューを出し、Configure IP を選択してください。

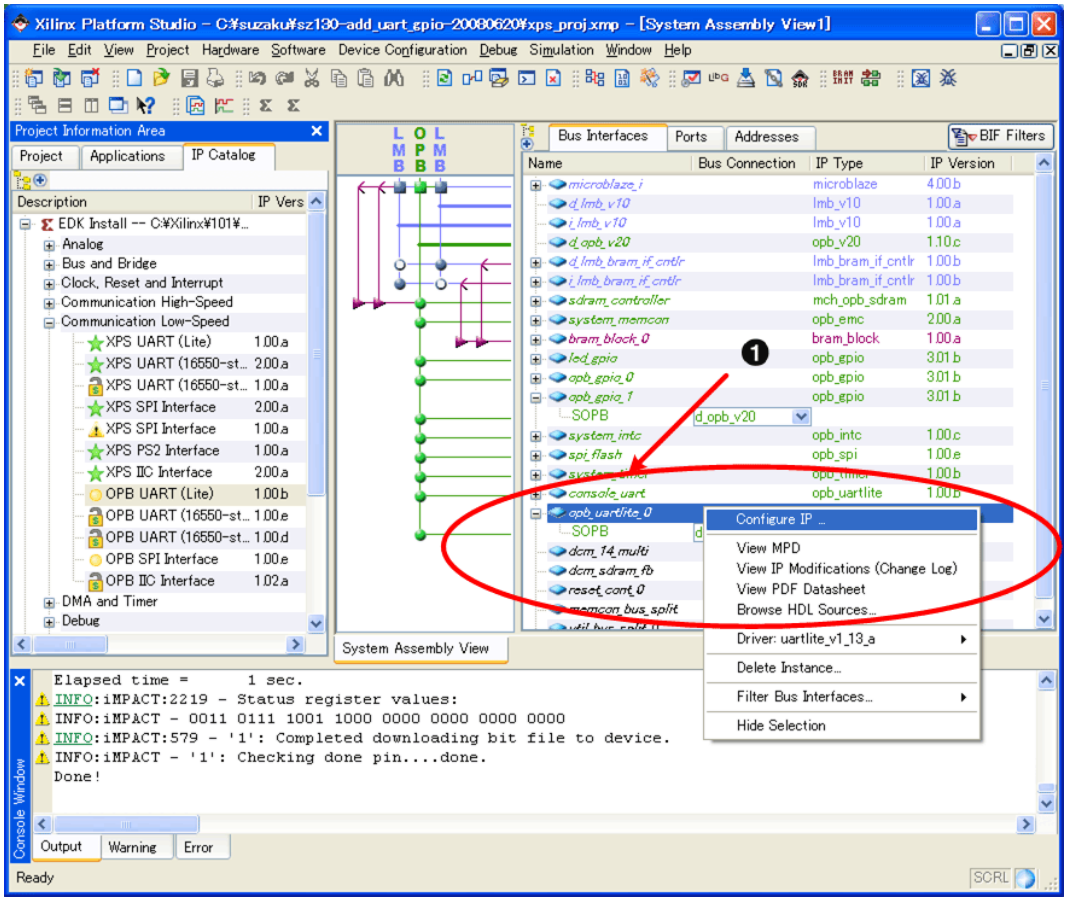


図 10.77. Configure IP

❶　右クリックしてメニューを出し、Configure IP を選択

以下の設定にしてください。

UART Lite Baud Rate	115200
Number of Data Bits in a Serial Frame	8
Use Parity	FALSE
Parity Type	ODD

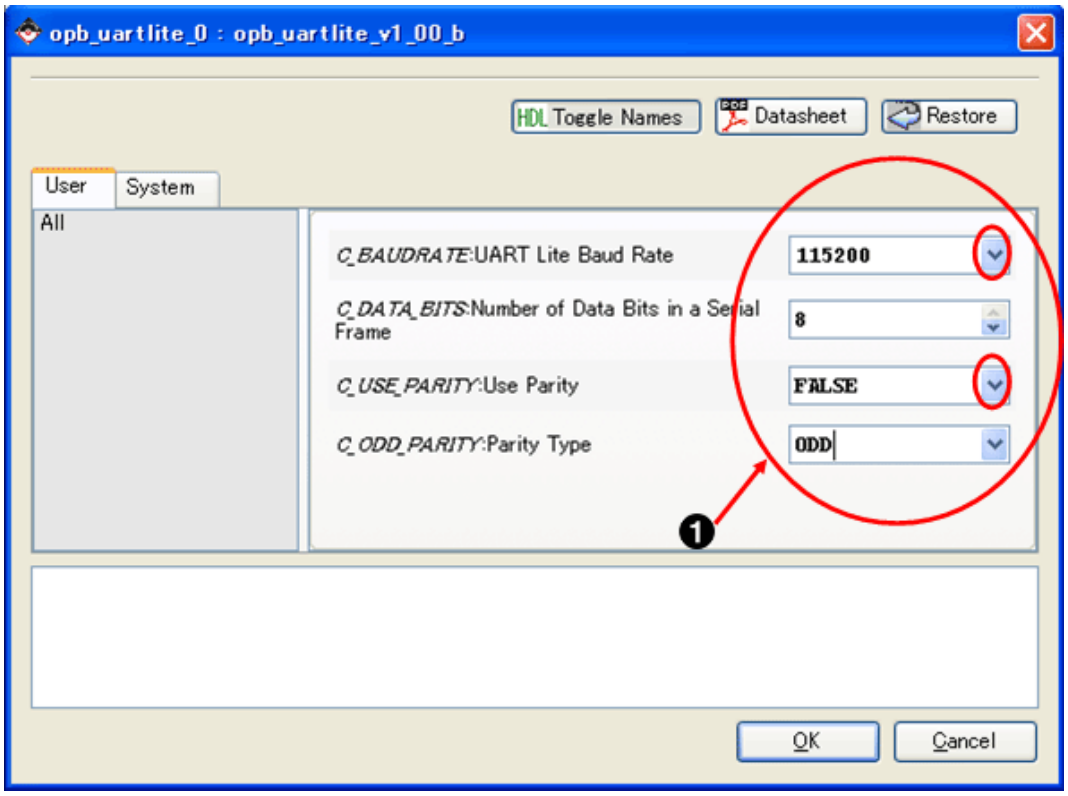


図 10.78. UART 設定変更

❶ 設定変更

[System]タブをクリックし、[Base Address]、[High Address]に入力してください。メモリアドレスは SUZAKU のメモリマップで Free と書いてあるところに割り当てます。

（「1.4. メモリマップ」参照）

表 10.6. UART メモリアドレス

	SZ010、SZ030、SZ130	SZ310、SZ410
Base Address	0xFFFFA600	0xF0FFA600
High Address	0xFFFFA6FF	0xF0FFA6FF

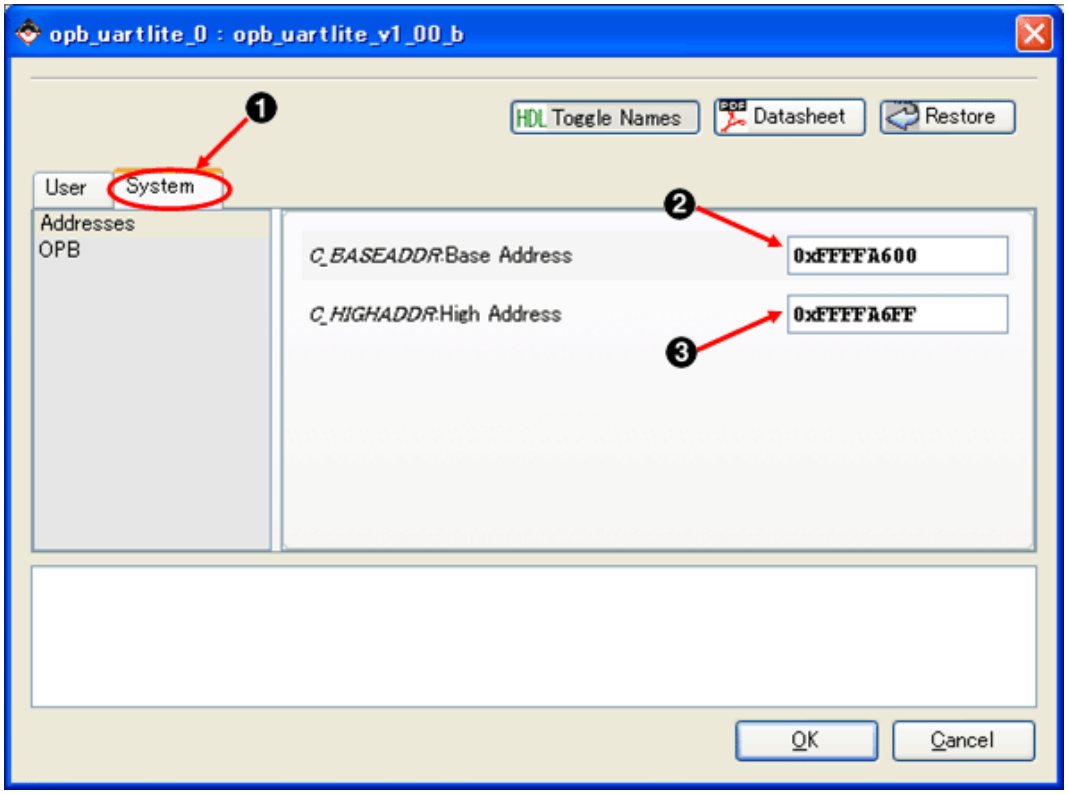


図 10.79. メモリアドレス設定

❶ System タブをクリック

❷ Base Address を入力

❸ High Address を入力

OPB | PLB をクリックし、[OPB Clock Frequency]もしくは[Clock Frequency of PLB Slave]にクロック周波数を入力して、[OK]をクリックしてください。

表 10.7. OPB Clock Frequency

	SZ010、SZ030、SZ130	SZ310	SZ410
OPB Clock Frequency	51609600Hz	66355200Hz	87500000Hz

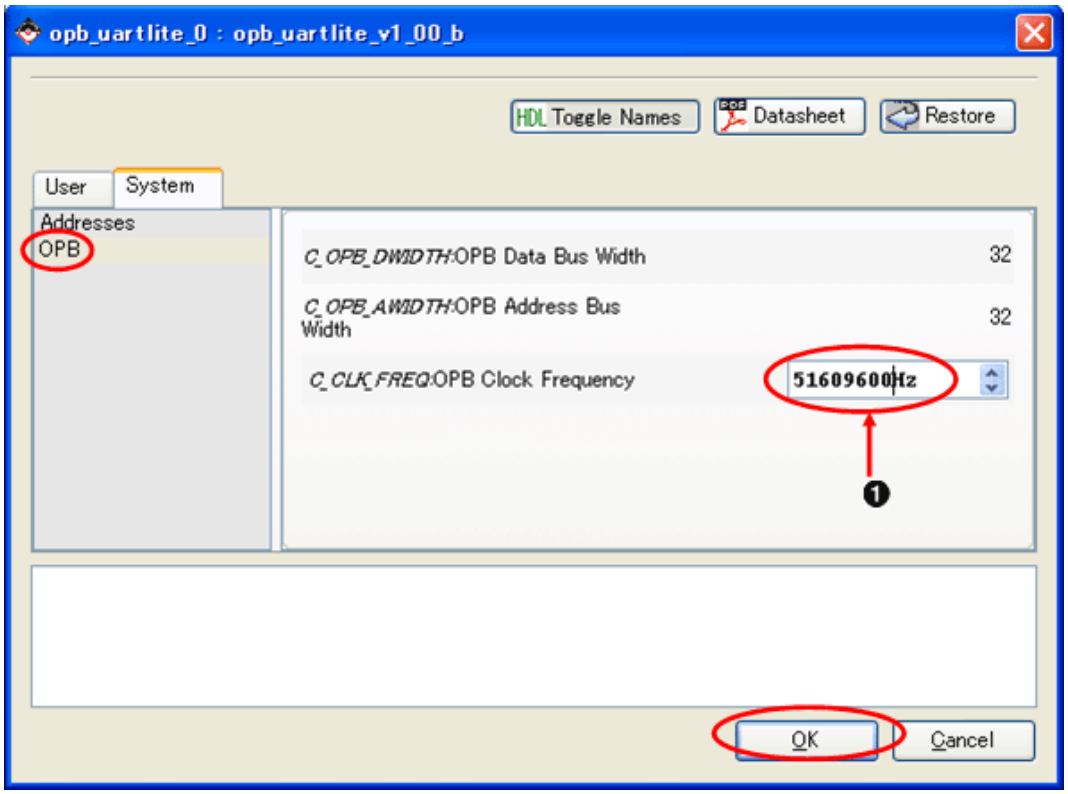


図 10.80. クロック周波数の設定

❶　クロック周波数入力

10.4.1.4. メモリマップ確認

Addresses を選択し、BaseAddress と High Address と Size に間違いがないか確認してください。

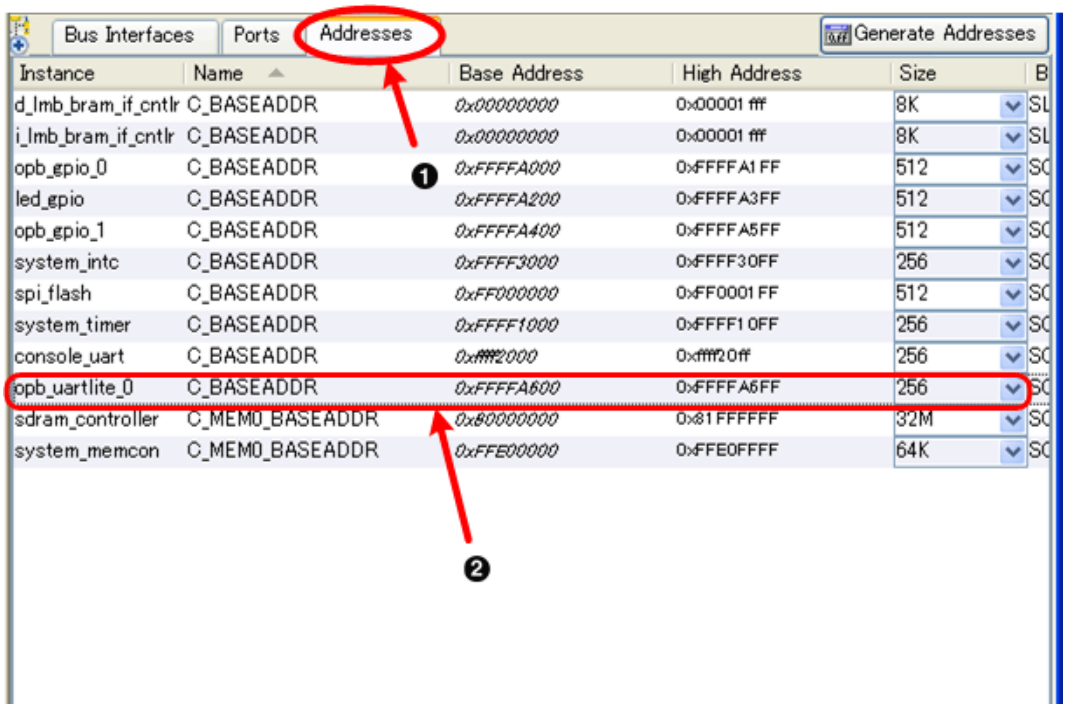


図 10.81. メモリマップ確認

❶　Addresses を選択

❷　メモリマップを確認

10.4.1.5. 信号の定義

Ports を選択してください。UART lite の RX と TX の Net に名前をつけ、その後 Make External を選択して確定してください。

External Ports に信号が定義されているか確認してください。

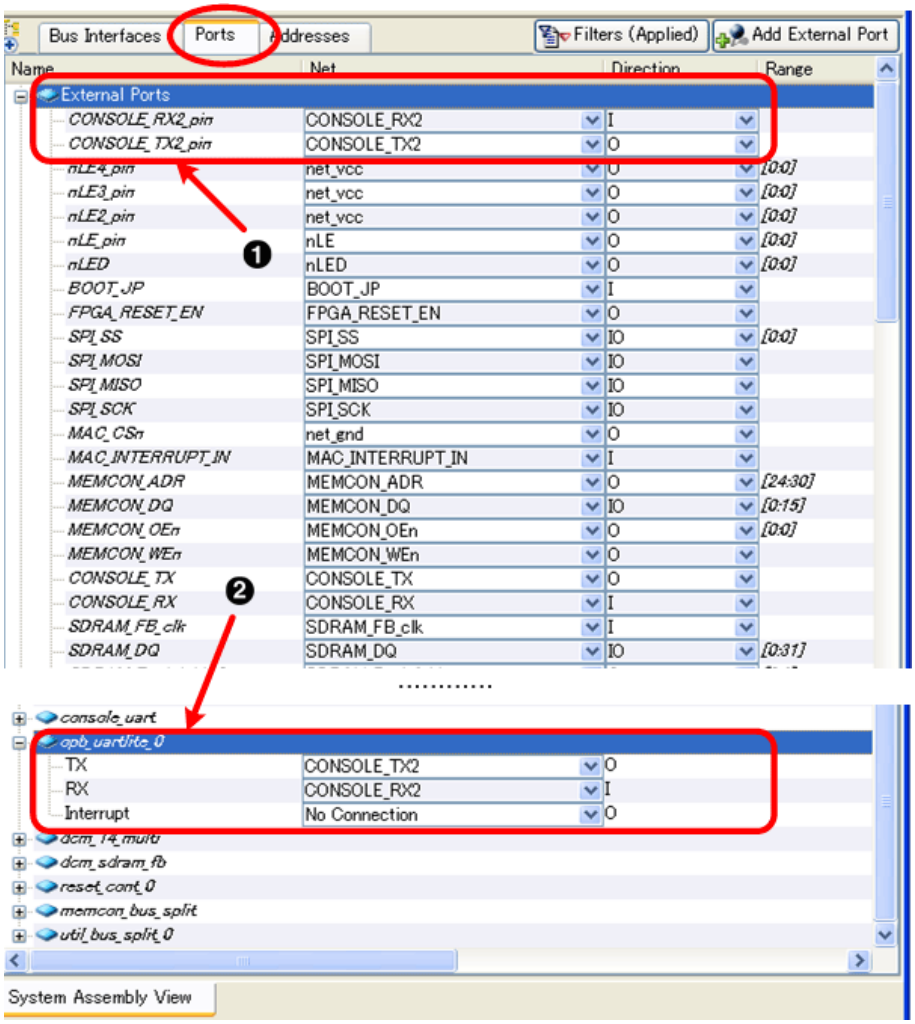


図 10.82. 信号の定義

❶　CONSOLE_RX2_pin, CONSOLE_TX2_pin が定義されているのを確認

❷　名前を入力し、確定

CONSOLE_RX2

CONSOLE_TX2

↓

Make External を選択し、確定

10.4.1.6. ピンアサイン

Project タブをクリックし、UCF ファイルを開き、増えた 2 ピンを追加し、保存してください。

表 10.8. CONSOLE ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
CONSOLE_RX2_pin	B4	M3	F13	P4
CONSOLE_TX2_pin	A3	M6	E13	E15

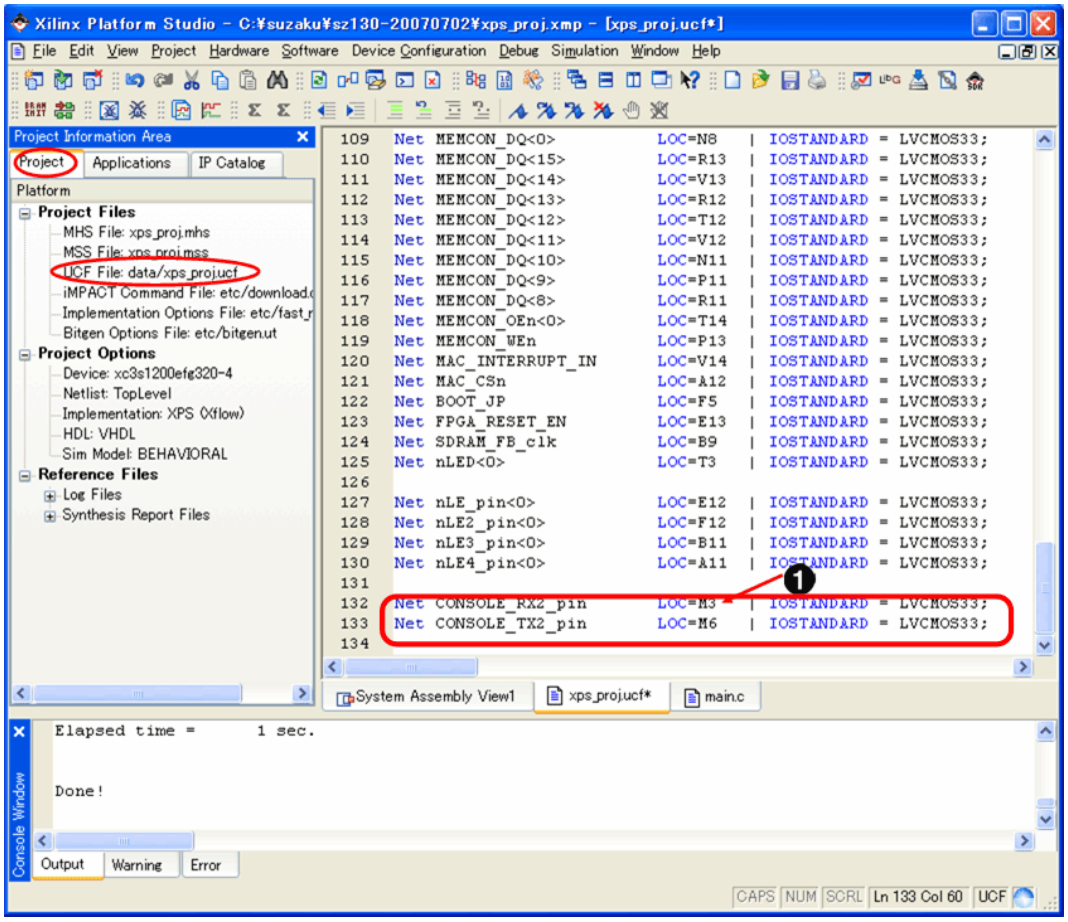
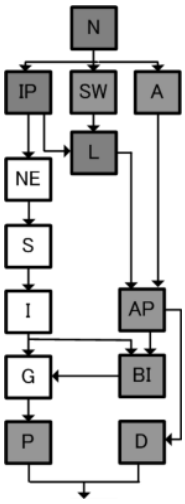


図 10.83. UART(xps_proj.ucf)

❶　ピンアサイン追記

10.4.2. ネットリスト, プログラムファイル(Hard のみ) 作成

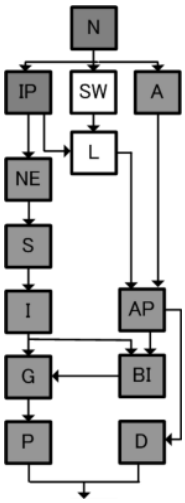


[Hardware]→[Generate Netlist]をクリックして下さい。ネットリストが生成されます。

[Hardware]→[Generate Bitstream]をクリックして下さい。ソフトウェアを含まない bit ファイルが生成されます。エラーが出た場合は、今までの工程を見直してみてください。

10.4.3. ソフトウェア設定

10.4.3.1. ライブラリ, ドライバ設定



[Software]→[Software Platform Settings]をクリックしてください。
追加した opb_uartlite の Driver を generic に変更し、OK をクリックしてください。

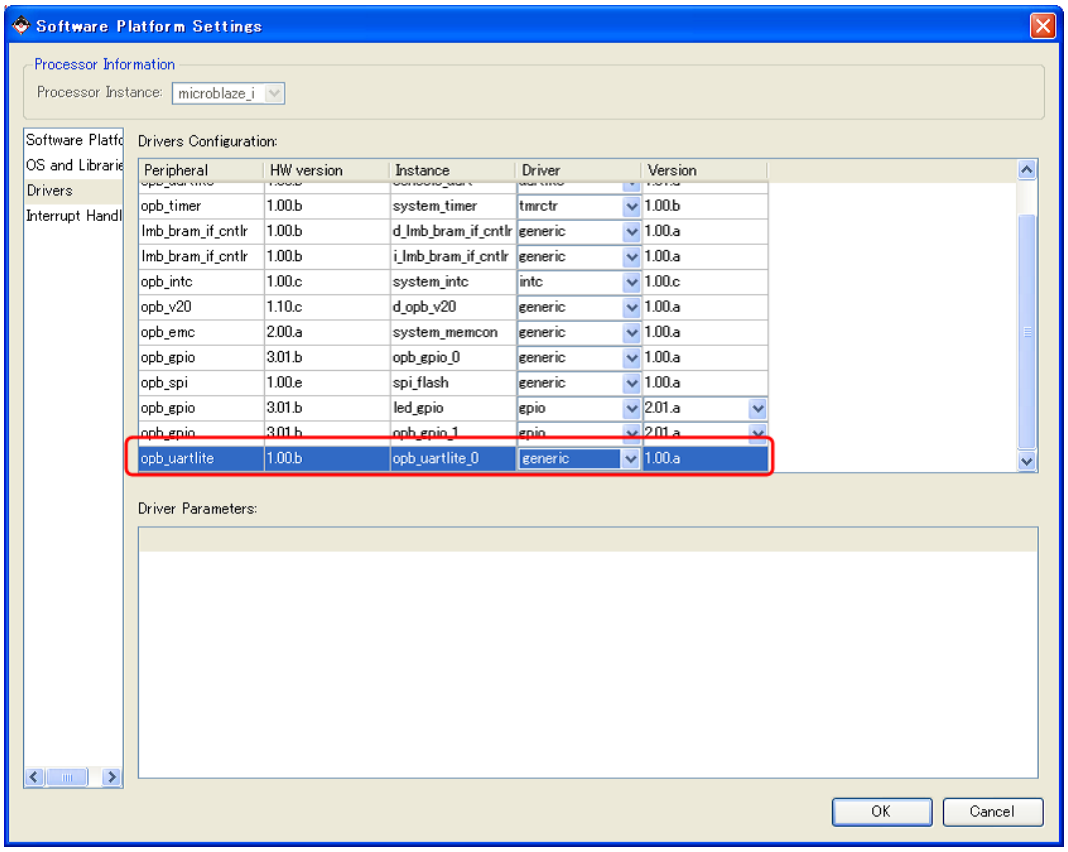


図 10.84. UART Driver 設定

10.4.3.2. ライブラリ, ドライバ生成

[Software]→[Generate Libraries and BSPs]をクリックして下さい。

ライブラリと様々な設定を定義したヘッダファイルが出来上がります。

`xparametaers.h` を開いてください。`xparameters.h` にはシステムのアドレスマップが定義されます。

先ほど設定した UART の BASEADDR と HIGHADDR も自動で定義されています。

例 10.3. xparameters.h の定義の例

```
/* Definitions for peripheral OPB_UARTLITE_0 */
#define XPAR_OPB_UARTLITE_0_BASEADDR 0xFFFFA600
#define XPAR_OPB_UARTLITE_0_HIGHADDR 0xFFFFA6FF
```

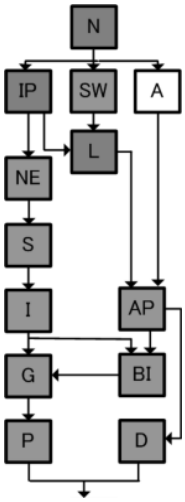
ソフトウェアに関するファイルは"C:\suzaku\sz***-add_uart_gpio\microblaze_i | ppc405_i | ppc405_system"の下に収められます。このフォルダの下の"include\xuartlite_l.h"を開いてください。UART を扱うことのできる関数等が定義されています。以下の関数を使います。

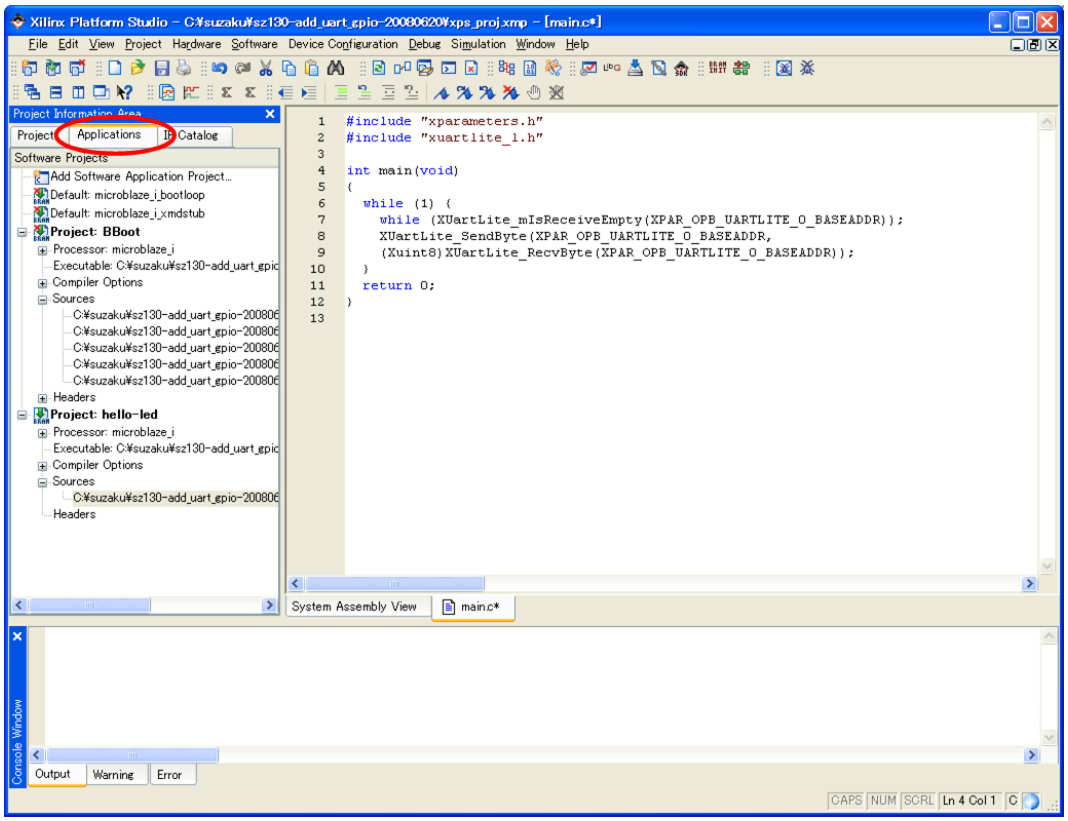
例 10.4. xuartlite_l.h に定義されている関数

```
/* 受信 FIFO のデータの有無をチェックする */
#define XUartLite_mIsReceiveEmpty(BaseAddress) \
  ((XUartLite_mGetStatusReg((BaseAddress)) & XUL_SR_RX_FIFO_VALID_DATA) != \
    XUL_SR_RX_FIFO_VALID_DATA)

/* 送信 FIFO に 1Byte 分 Write する */
void XUartLite_SendByte(Xuint32 BaseAddress, Xuint8 Data);

/* 受信 FIFO のデータを 1Byte 分 Read する */
Xuint8 XUartLite_RecvByte(Xuint32 BaseAddress);
```

10.4.4. アプリケーション編集

Applications タブをクリックし、先ほど作成した hello-led の main.c を開いてください。uartlite のヘッダファイルの定義を追加し、先ほど書いた単色 LED を点灯する一文を消し、受信した文字をそのまま送信するコードを追加し、保存してください。
[UART lite の BASEADDR]には `xparameters.h` で確認した UART lite の BASEADDR を入力してください。
インスタンス名を変更していなければ、SZ010,SZ030,SZ130,SZ310 の場合 XPAR_OPB_UARTLITE_0_BASEADDR、SZ410 の場合 XPAR_XPS_UARTLITE_0_BASEADDR となります。



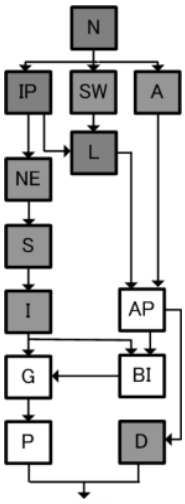



```
#include "xparameters.h"
#include "xuartlite_l.h"
int main(void)
{
  while (1) {
    while(XUartLite_mIsReceiveEmpty([UART lite  BASEADDR]));
    XUartLite_SendByte([UART lite  BASEADDR],
    (Xuint8)XUartLite_RecvByte([UART lite  BASEADDR]));
  }
  return 0;
}
```

図 10.85. 送受信ソースコード追加(main.c)

10.4.5. アプリケーション生成

[Software]→[Build All User Applications]をクリックして下さい。コンパイラが起動され、各ソフトウェア アプリケーションのプログラム ソースの設定が読み込まれます。エラーがなければ `executable.elf` が出来上がります。



10.4.6. プログラムファイル作成

ハードウェアでつくった bit ファイルの中にソフトウェアを書き込みます。 [Device Configuration]→[Update Bitstream]をクリックしてください。bit ファイルが生成されます。エラーがでたら間違いを修正して再び[Update Bitstream]をクリックしてください。bit ファイルは `download.bit` という名前で`"C:\suzaku\sz***-add_uart_gpio\implementation"`フォルダに出来上がります。

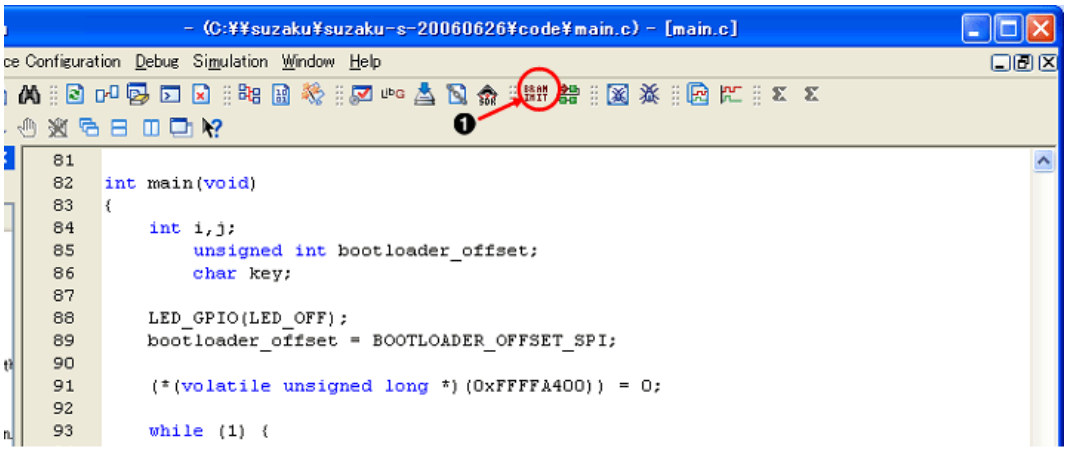


図 10.86. bit ファイルの作成

❶ Update Bitstream

10.4.7. コンフィギュレーション

シリアル通信用ソフトウェアを立ち上げ、シリアル通信の設定を行ってください。(「5.2. シリアル通信ソフトウェア」 参照)

SUZAKU JP2 をショートし、SUZAKU CON7 にダウンロードケーブルを接続してください。

LED/SW CON7 にシリアルケーブルを接続してください。

最後に LED/SW CON6 に AC アダプタ 5V を接続し、電源を投入してください。

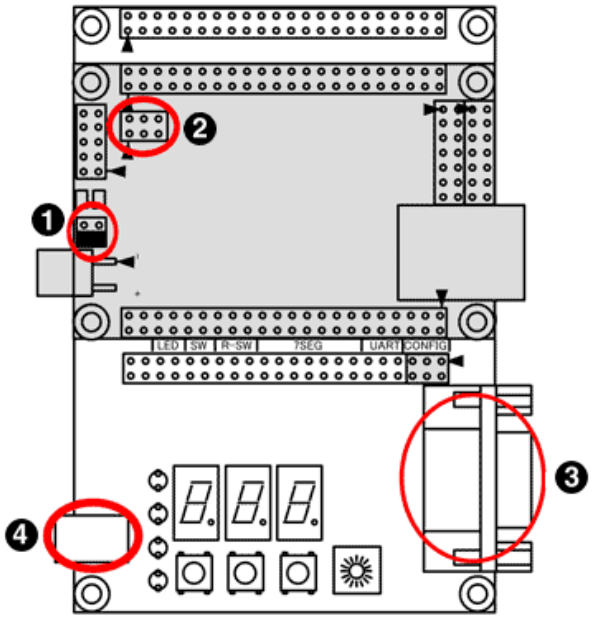


図 10.87. ジャンパの設定等

❶ JP2 ショート

❷ ダウンロードケーブル接続

❸ シリアルケーブル接続

❹ 電源投入

[Device Configuration]→[Download Bitstream]をクリックしてください。バッチモードの iMPACT を使用して FPGA に bit ファイルがコンフィギュレーションされます。

キーボードから何か文字を打ち込んでください。打ち込んだ文字がそのまま送信されてコンソールに表示されます。

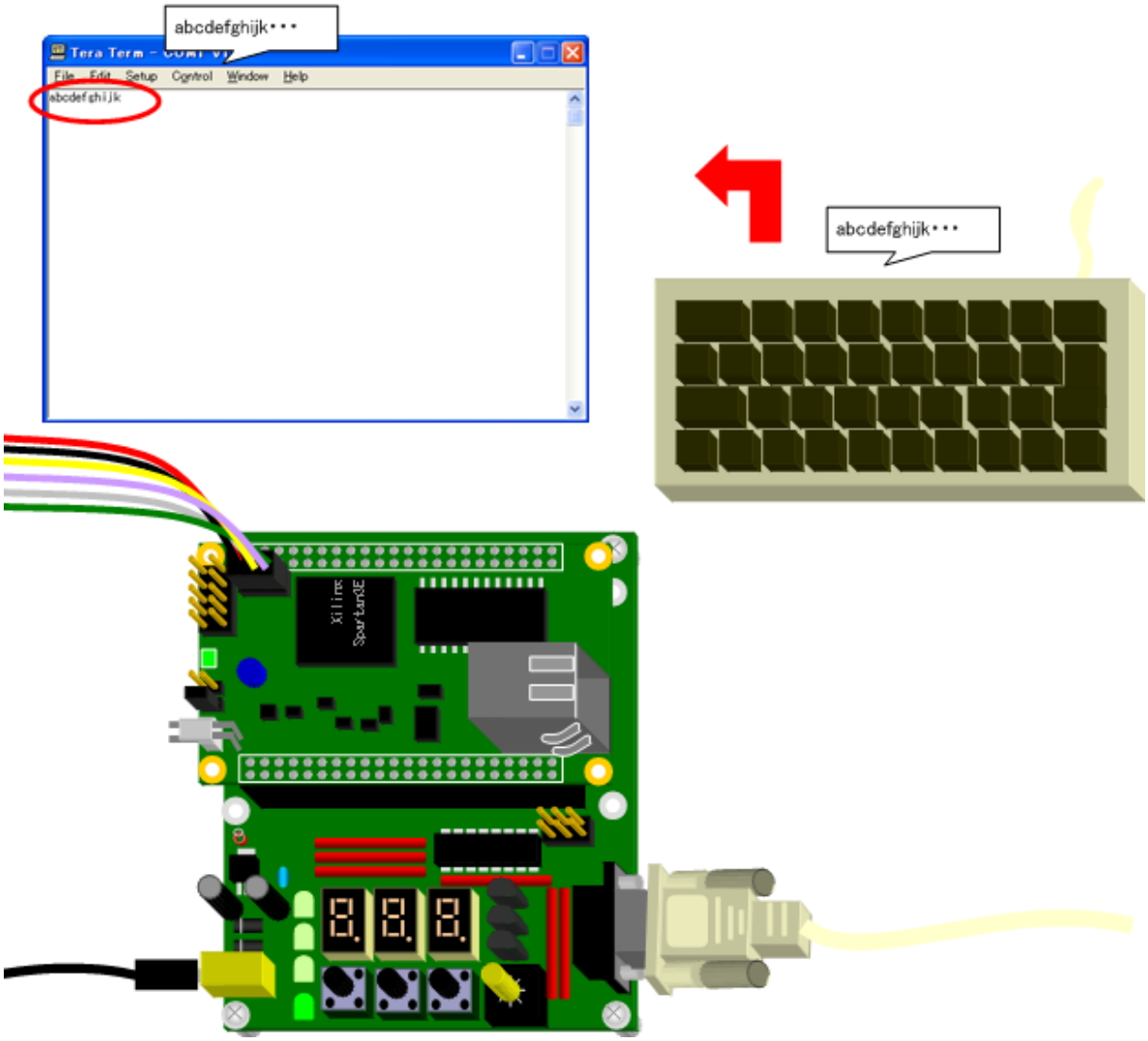


図 10.88. シリアル通信 動作確認

11.スロットマシンのコアを CPU で制御する

ここからはまた、スロットマシンに戻ります。「9. FPGA 入門 スロットマシン製作」で作った回路を少し改造してコアにして、SUZAKU のデフォルトの回路に接続し、スロットマシンを作り上げます。作業手順は以下の通りです。

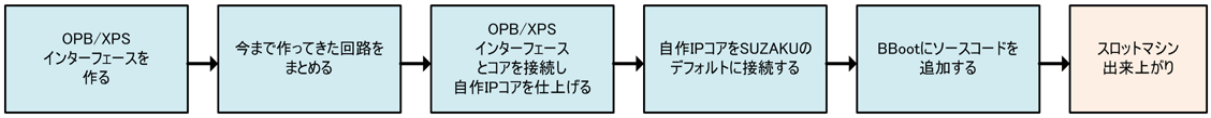


図 11.1. スロットマシンへの道のり

付属 CD-ROM の`"\suzaku\fpga_proj\x.x\sz***\sz***-yyyymmdd.zip"`をハードディスクに展開してください。ここでは展開後のフォルダを`"C:\suzaku"`の下にコピーして作業を進めます。

`"C:\suzaku\sz***- yyyymmdd"`の中の`"xps_proj.xmp"`をダブルクリックして開いてください。

Xilinx Platform Studio が起動し、SUZAKU のデフォルトが開きます。

11.1. スロットマシンのコアの構成 (OPB)

SZ410 をお使いの場合は「11.5. スロットマシンのコアの構成 (XPS)」へ進んでください。

スロットマシンのコアを"opb_sil00"という名前で下図のように製作します。

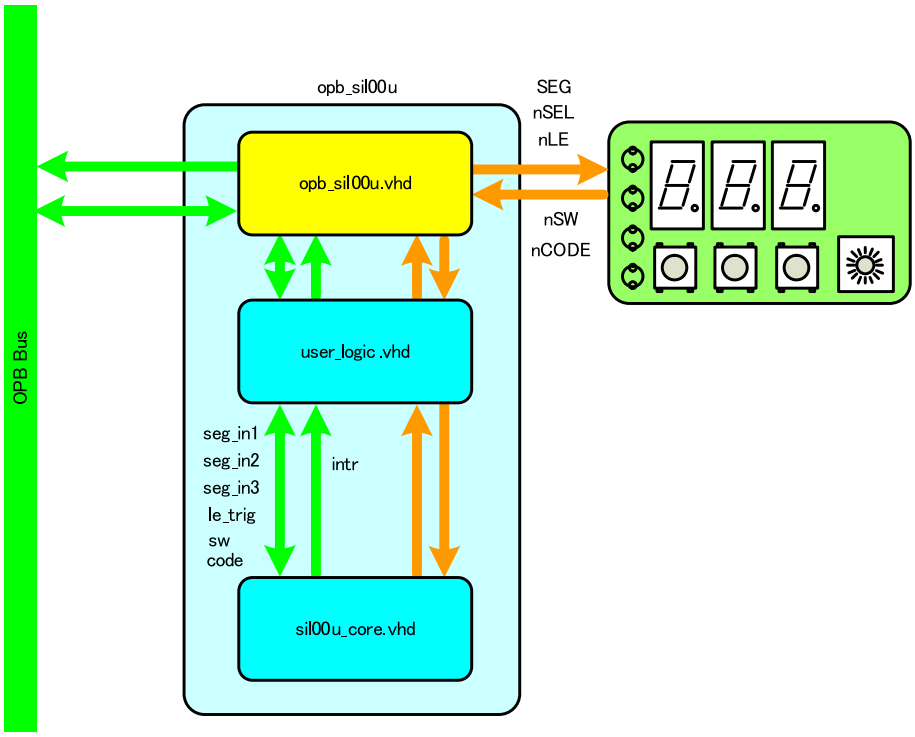
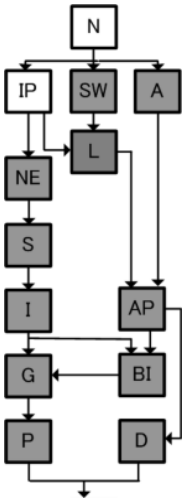

opb_sil00u
SEG
nSEL
nLE
opb_sil00u.vhd
nSW
nCODE
OPB Bus
user_logic.vhd
seg_in1
seg_in2
seg_in3
le_trig
sw
code
intr
sil00u_core.vhd

図 11.2. 自作 IP コア

11.2. ウィザードを使って OPB インターフェースをつくる



OPB バスに接続するインターフェースをつくります。EDK にはインターフェースを作るウィザードが用意されているので、簡単にインターフェースをつくることが出来ます。スロットマシンのコアを CPU から制御するためには、バスに接続しなければいけません。

[Hardware]→[Create or Import Peripheral...]をクリックしてください。

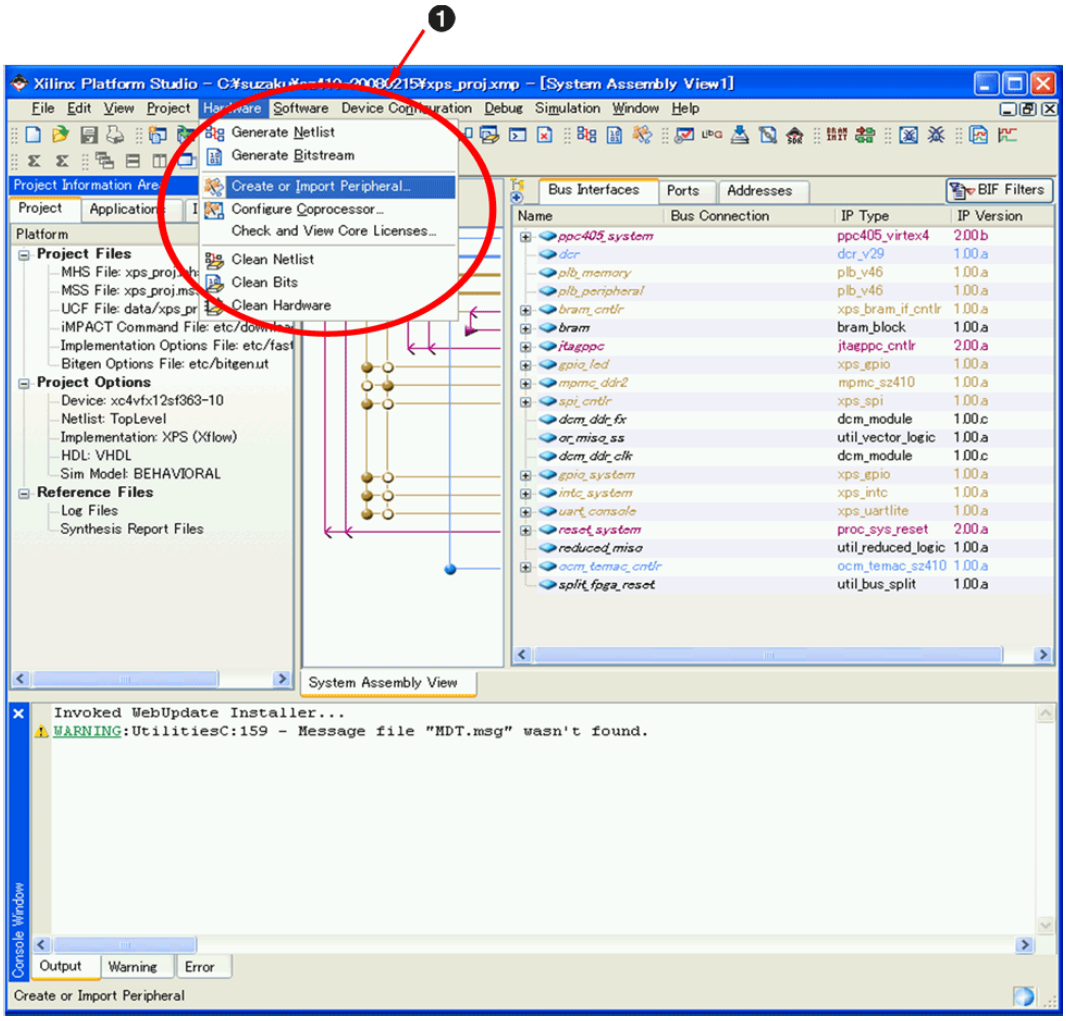


図 11.3. Create and Import Peripheral Wizard の起動のさせ方

❶　[Hardware]→[Create or Import Peripheral...]をクリック

Create and Import Peripheral Wizard が立ち上がります。[Next]をクリックして下さい。

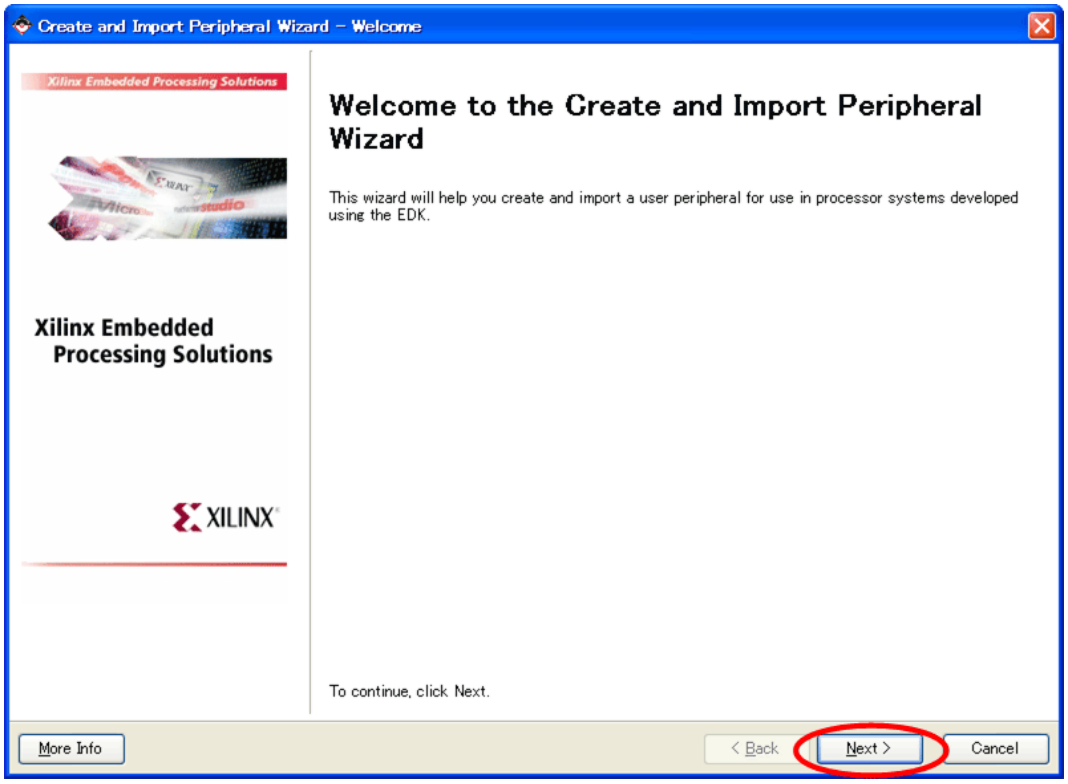


図 11.4. Create and Import Peripheral Wizard

Create and Import Peripheral Wizard が立ち上がります。新規で作るので Select flow の[Create templates for a new peripheral]をチェックして[Next]をクリックしてください。

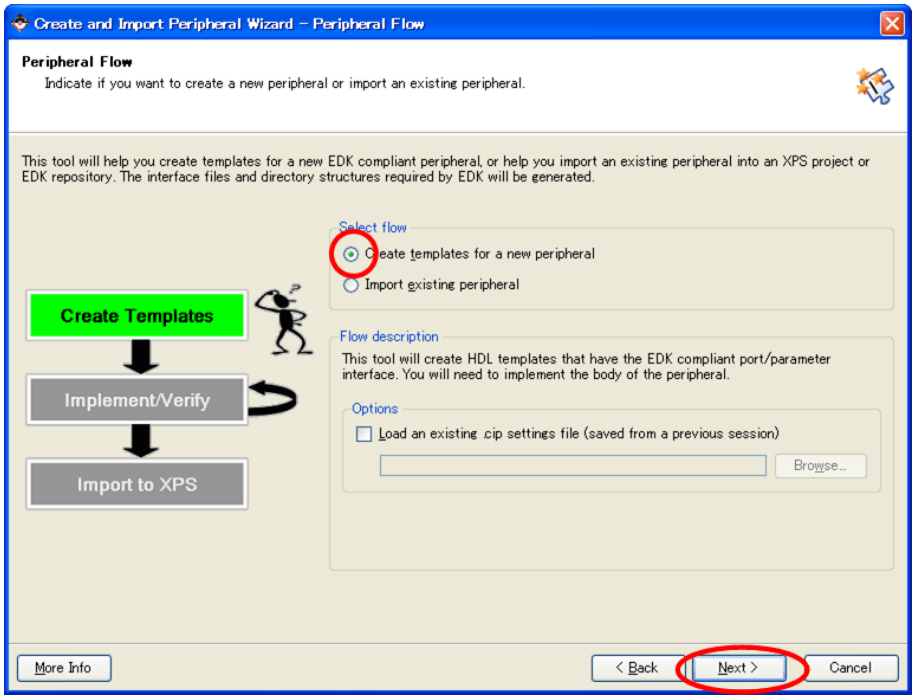


図 11.5. Peripheral Flow

コアを生成する場所を指定します。何も変更せずそのまま[NEXT]をクリックしてください。

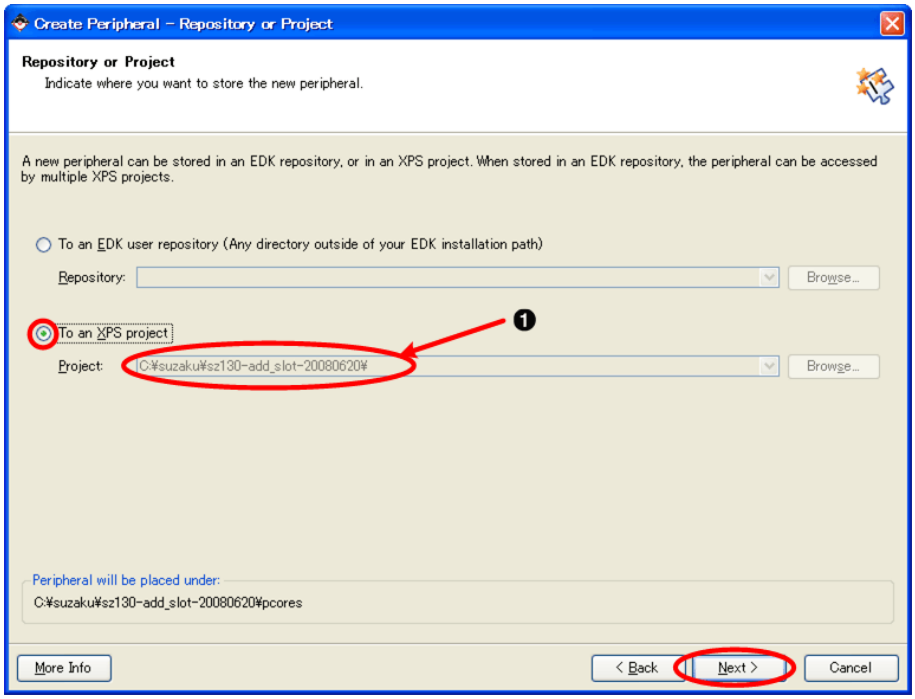


図 11.6. コアの生成場所の指定

❶　C:\suzaku\sz***- yyyymmdd \

コアに名前をつけます。[Name]に名前を入力してください。opb_sil00 とします。

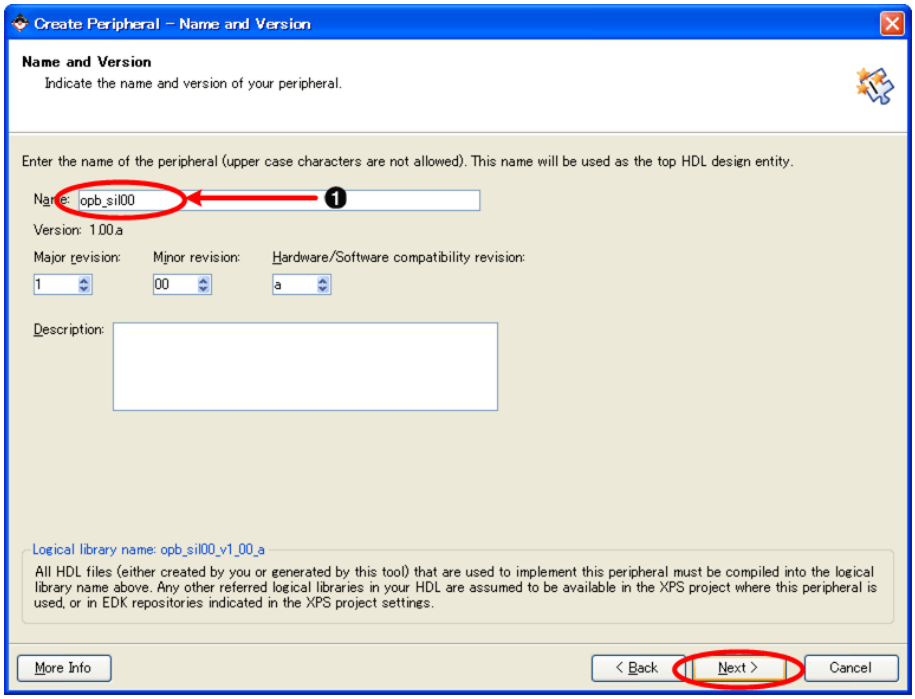


図 11.7. コアの名前

❶　opb_sil00 と入力

バスを選択します。OPB につなぐので[On-chip Peripheral Bus(OPB)]を選択して[Next]をクリックしてください。

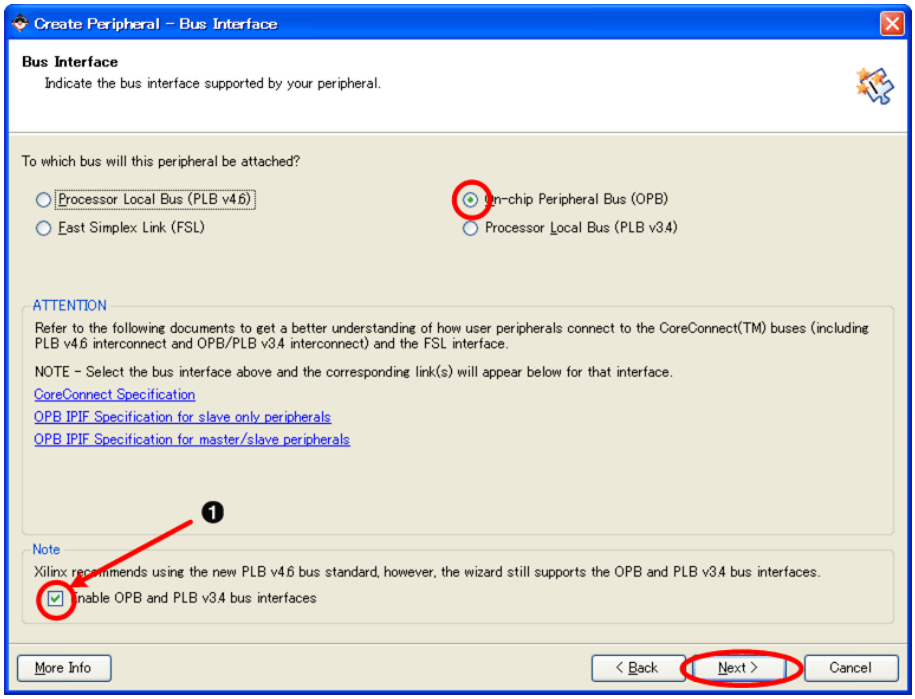


図 11.8. バスの選択

❶　チェックしないと OPB は表示されません.

IPIF にはアドレスのデコード、バイト調整などの基本的な機能に加えて、ペリフェラルの作成を大幅に簡略化するオプションの機能が備わっています。選択した項目により OPB ペリフェラルテンプレートが生成されます。

今回は[User logic interrupt support]、[User logic S/W register support]を選択し、[Next]をクリックしてください。割り込みのユーザーテンプレート、ソフトウェアアクセス可能レジスタが生成されるユーザーテンプレートが追加されます。

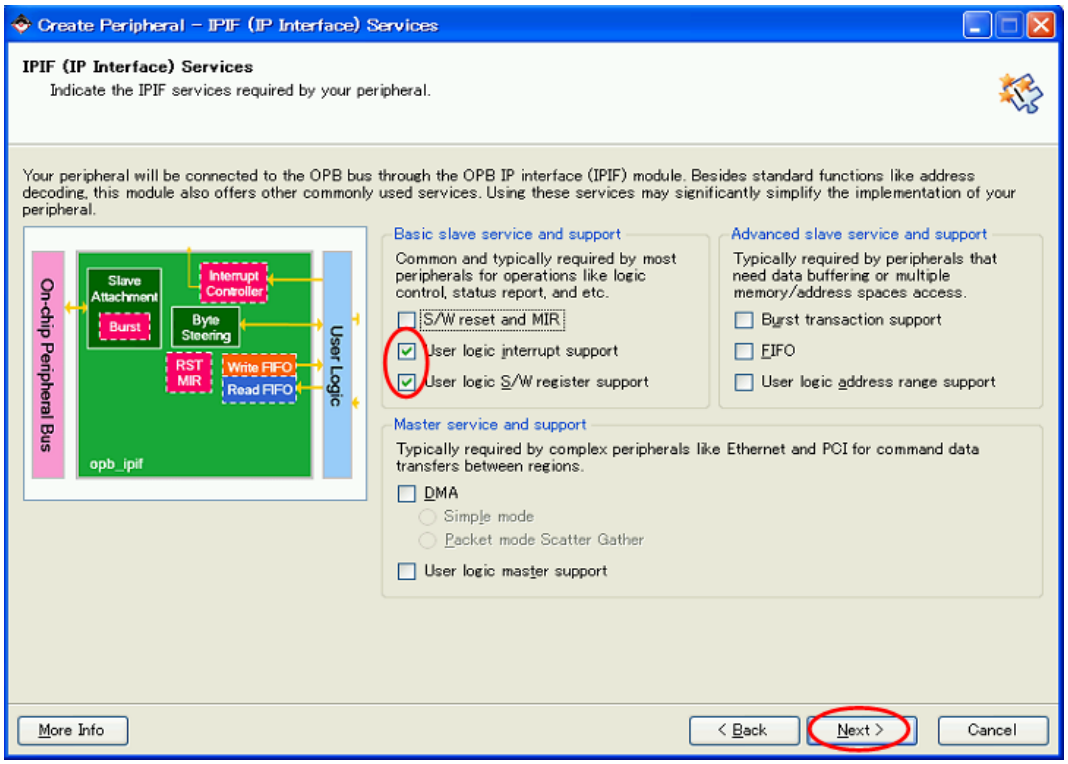


図 11.9. テンプレート追加

割り込みの設定をします。割り込みに使用するカウンタの Duty が 50%なのでエッジ取り込みにし、今回は立ち上がりエッジを使用します。[Use Device ISC(interrupt source controller)]のチェックボタンをはずし、Interrupt capture mode を[Rising Edge Detect]に設定して、[Next]をクリックして下さい。

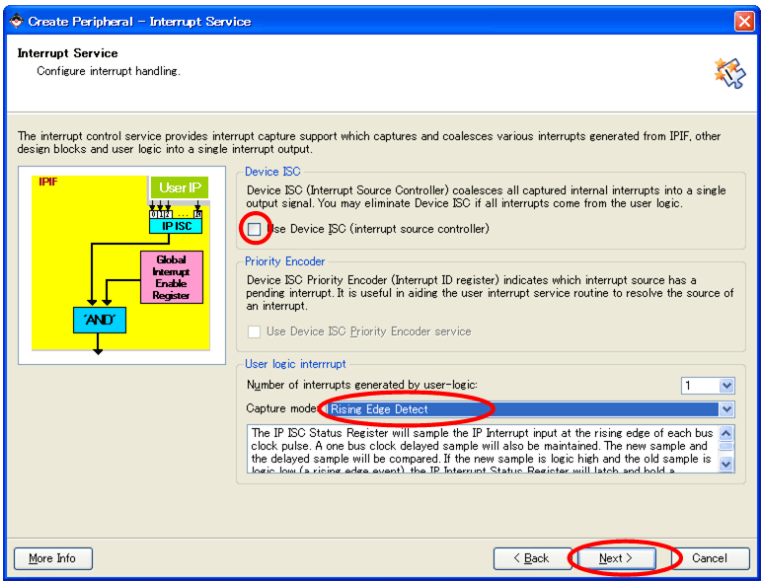


図 11.10. Interrupt 設定

ソフトウェアアクセス可能レジスタ数と、サイズ(バイト、ハーフワード、ワード、ダブルワード)を指定します。今回必要となるレジスタは、読み込み/書き込みのレジスタが 4 つ(7 セグメント LED の値を設定するレジスタ 3 つ、単色 LED のトリガ信号を設定するレジスタ)、読み込みのレジスタが 2 つ(押しボタンスイッチの情報をやり取りするレジスタ、ロータリコードスイッチの情報をやり取りするレジスタ)の合計 6 つです。サイズは 4 ビットでいいのですが、最小単位がバイトなので、8 ビットとします。

[Number of software accessible resiters]を 6 にし、[Data width of each register]を 8bit にし、[Next]をクリックしてください。

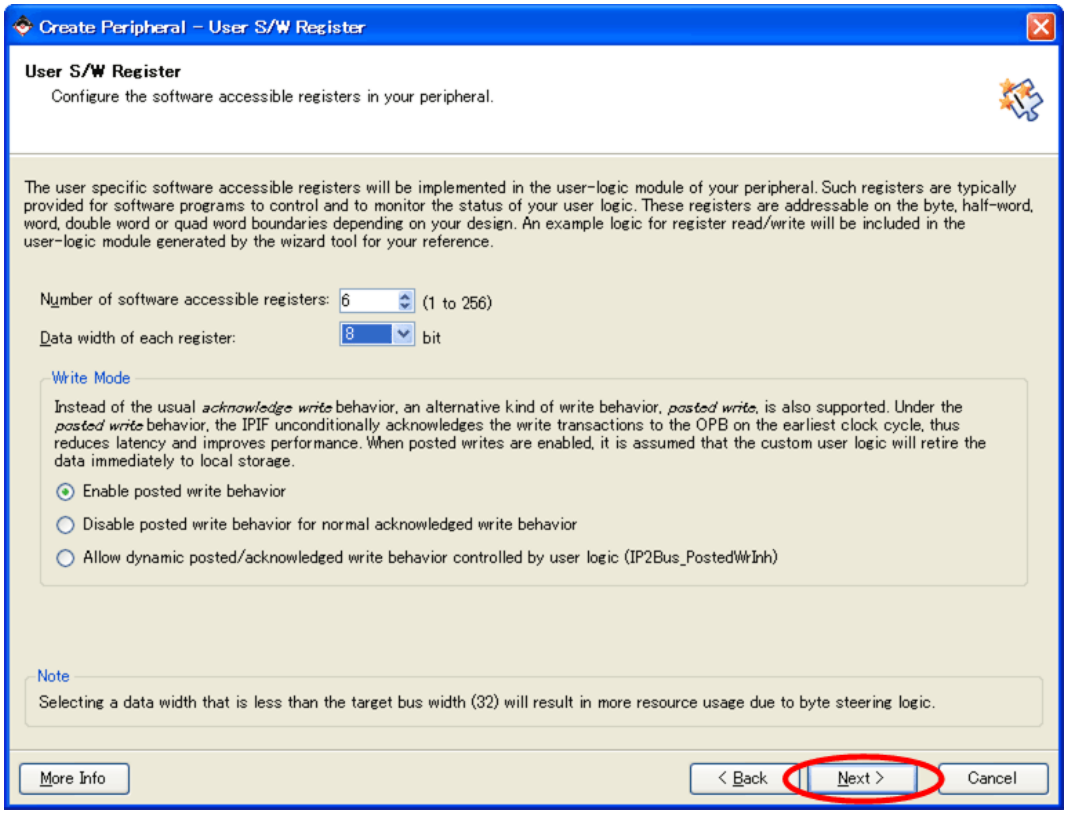


図 11.11. レジスタ数とバス幅指定

IPIC を設定します。すでにいくつか ON になっていますが、IPIF Services ページで指定した機能をインプリメントするために必要なものに自動でチェックされています。このまま[Next]をクリックしてください。

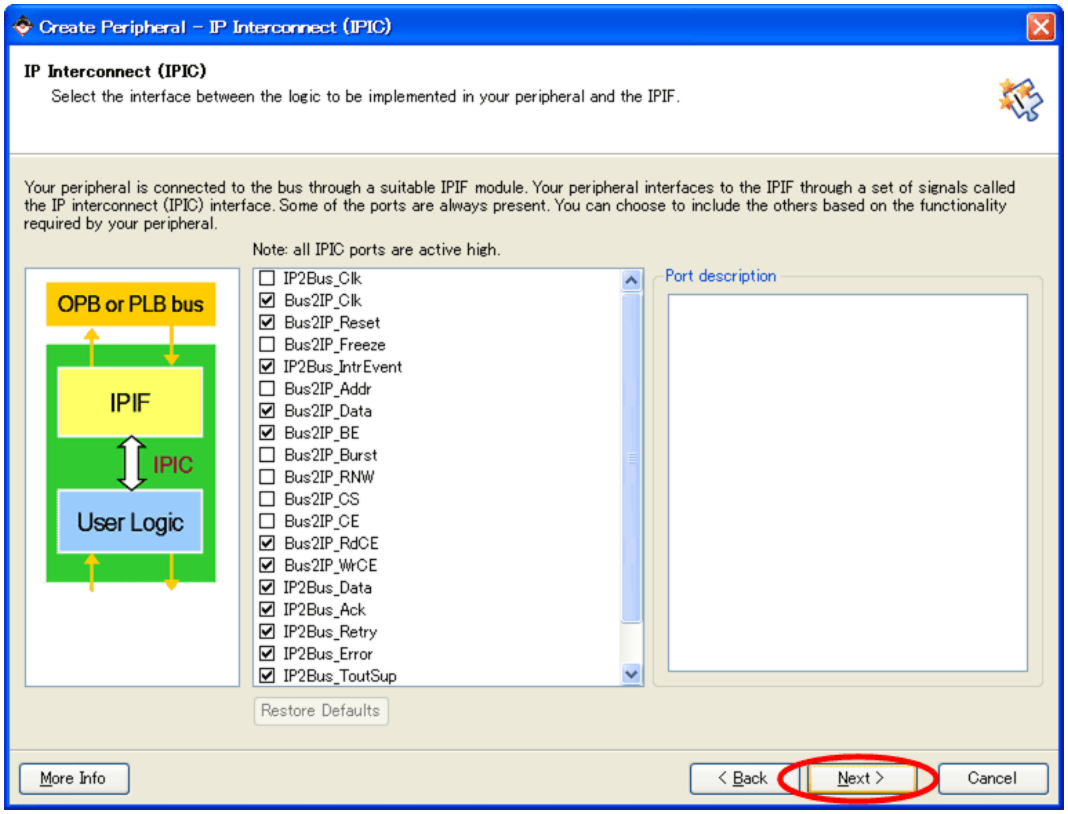


図 11.12. IPIC 設定

ここを ON にすると、カスタムロジックおよび機能のシミュレーションに使用するサポートファイルを生成できますが、今回は使いません。そのまま[Next]をクリックしてください。

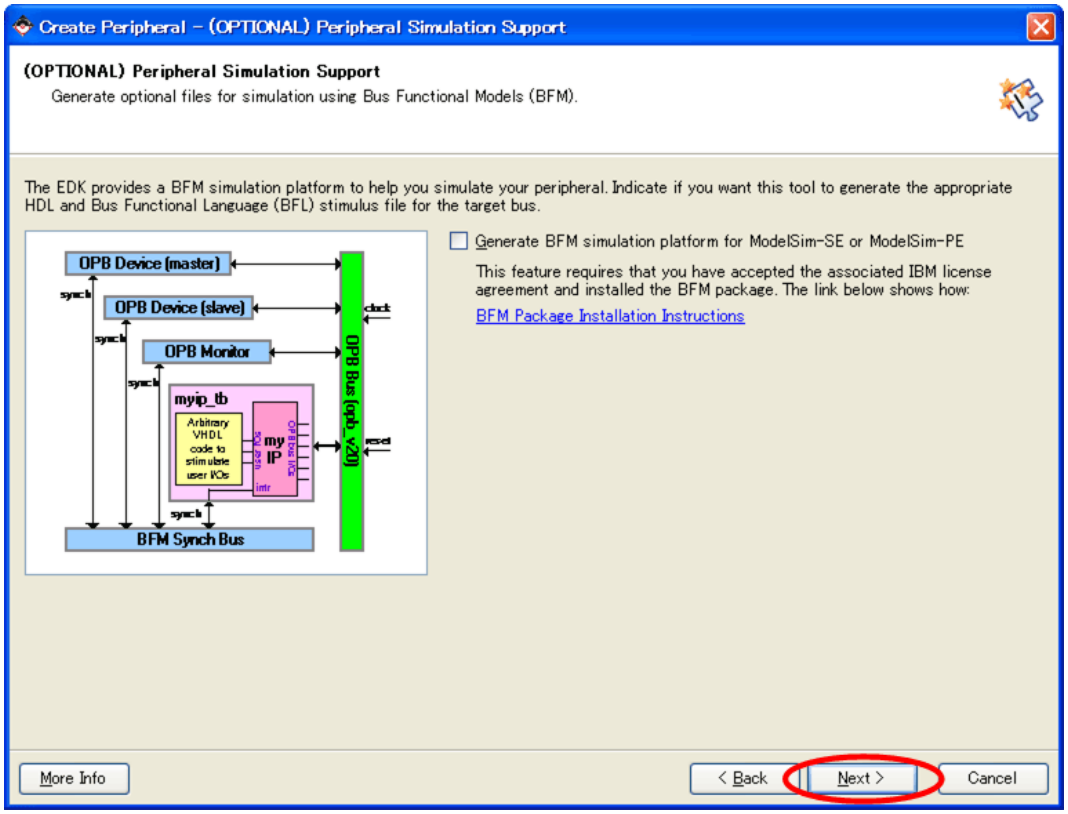


図 11.13. サポートファイル生成確認

下図の項目をチェックし、[Next]をクリックしてください。

ソフトフェアドライバテンプレートファイルとドライバディレクトリ構成が作成されます。

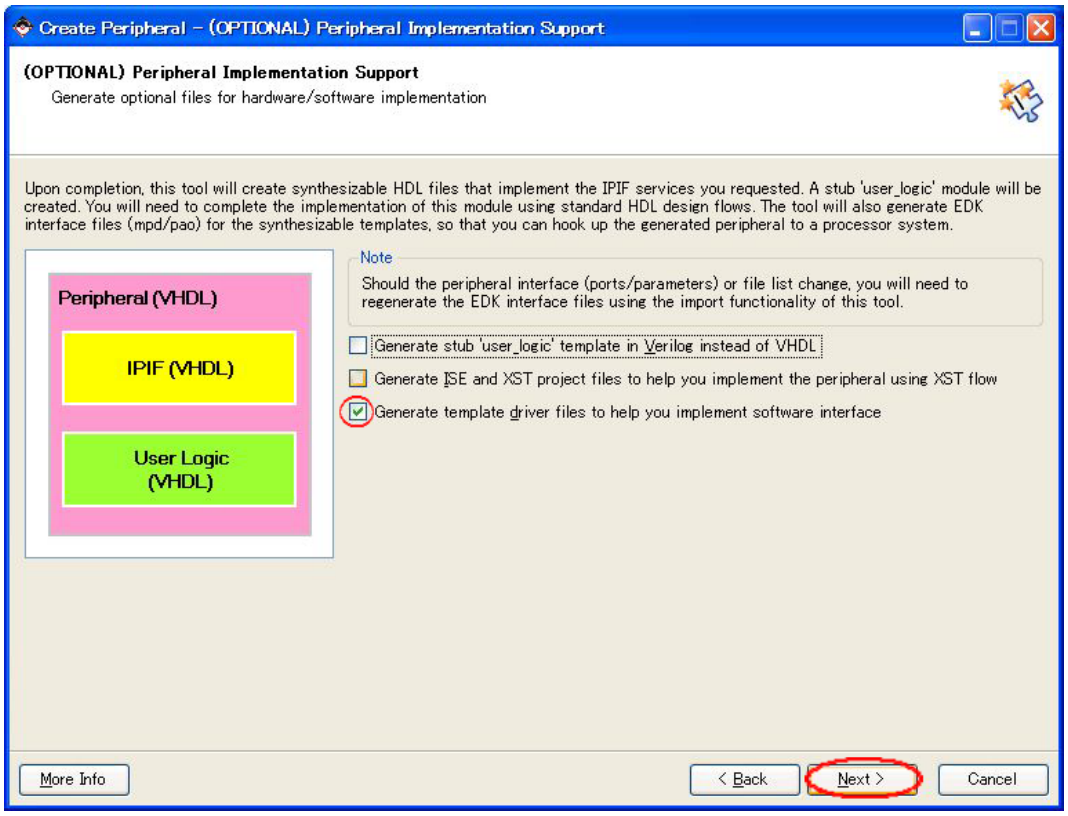


図 11.14. オプション設定

以上で終了です。[Finish]をクリックしてください。

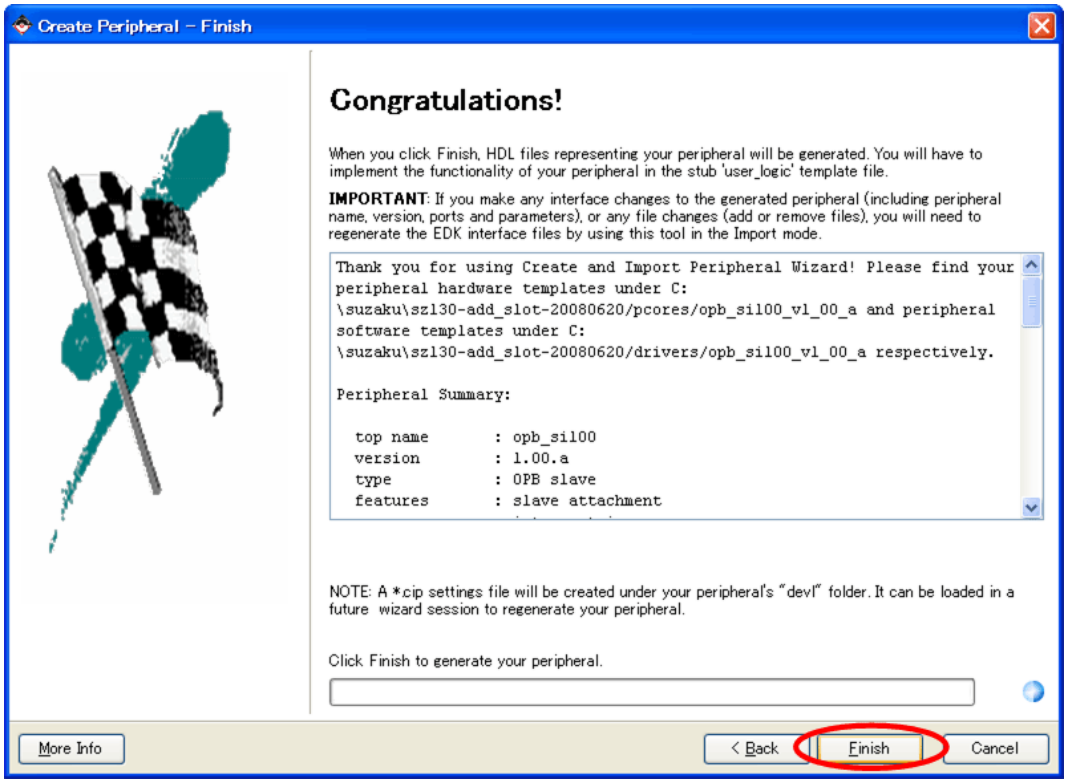


図 11.15. 終了

"C:\suzaku\sz***-yyyymmdd\pcores"の下に OPB バスに接続するインターフェースの雛形が生成されます。

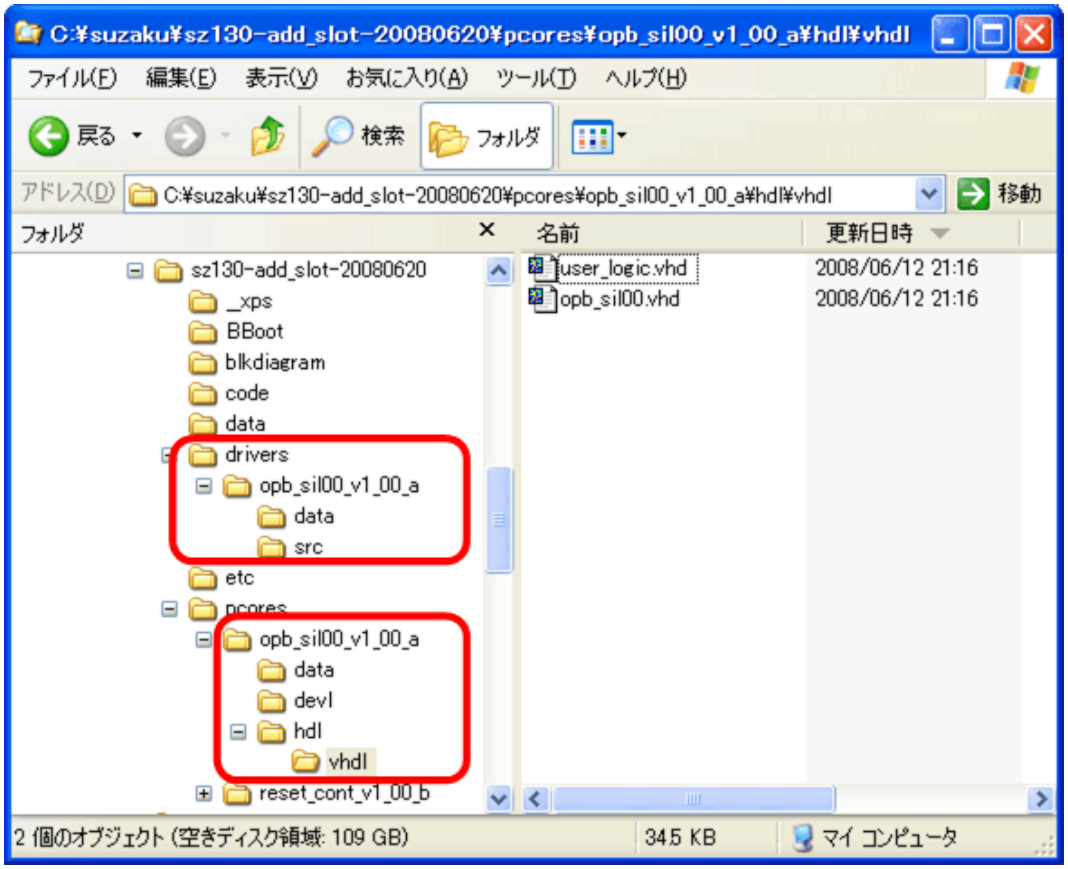


図 11.16. フォルダ構成

11.3. 今まで作ってきた回路をまとめる (OPB)

下図の仕様で今まで作ってきた回路を OPB バスに接続できるようにまとめます。`sil00u_core.vhd` をテキストエディタ等で新規作成してください。

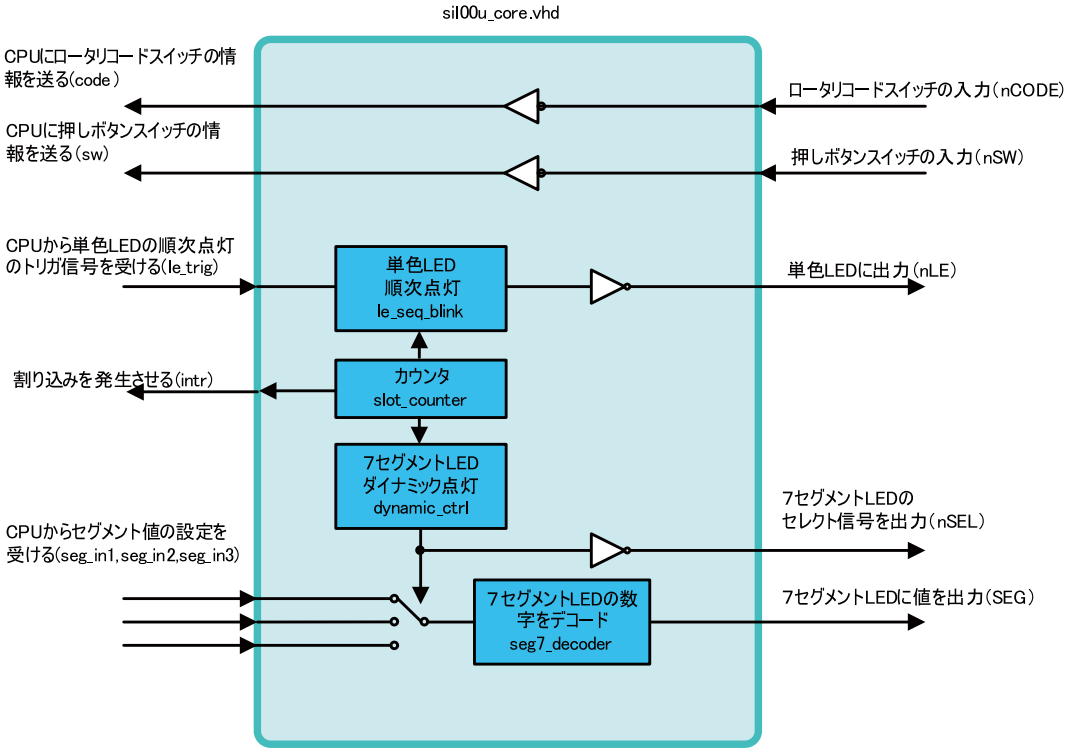


図 11.17. 自作 IP コア(ソフト版)の仕様

11.3.1. sil00u_core.vhd

SZ010、SZ030、SZ130 ではバスクロックが 51.6096MHz、SZ310 では 66.3552MHz、SZ410 では 87.5MHz になっています。カウンタのビット数を 4 ビット増やし 23 ビットにします。sil00u_core.vhd を上位階層として今まで作った回路、slot_counter、le_seq_blink、seg7_decoder、dynamic_ctrl 回路を呼び出します。また、押しボタンスイッチ、ロータリコードスイッチ、割り込みの信号の定義をします。

例 11.1. コア(sil00u_core.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sil00u_core is
  generic (
    C_CNT_WIDTH : integer := 23  --カウンタのビット幅
  );

  Port (
    SYS_CLK : in  STD_LOGIC;                --クロック信号      ❶
    SYS_RST : in  STD_LOGIC;                --リセット信号

    -- External
    SEG     : out STD_LOGIC_VECTOR(0 to 7); --7 セグ LED にダイナミック点灯で値を出力
    nSEL    : out STD_LOGIC_VECTOR(0 to 2); --7 セグ LED にセレクトをを出力
    nLE     : out STD_LOGIC_VECTOR(0 to 3); --単色 LED に順次点灯を出力
```

```
    nSW     : in  STD_LOGIC_VECTOR(0 to 2); --押しボタンスイッチを入力
    nCODE   : in  STD_LOGIC_VECTOR(0 to 3); --ロータリコードスイッチを入力

    -- Register Write
    seg_in1 : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    seg_in2 : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    seg_in3 : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    le_trig : in  STD_LOGIC;                --CPU から単色 LED の順次点灯のトリガ信号の設
定を受ける

    -- Register Read
    sw      : out STD_LOGIC_VECTOR(0 to 2); --CPU に押しボタンスイッチの情報を送る
    code    : out STD_LOGIC_VECTOR(0 to 3); --CPU にロータリコードスイッチの情報を送る
    intr    : out STD_LOGIC                 --カウンタの出力を割り込みコントローラに送る
  );
end sil00u_core;

architecture IMP of sil00u_core is
  signal count    : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1);
  signal le       : STD_LOGIC_VECTOR(0 to 3);
  signal le_t     : STD_LOGIC_VECTOR(0 to 3);
  signal seg_data : STD_LOGIC_VECTOR(0 to 3);

  component slot_counter                                  ❷
    generic (
      C_CNT_WIDTH : integer := C_CNT_WIDTH
    );
    Port (
      SYS_CLK : in STD_LOGIC;                             --クロック信号
      SYS_RST : in STD_LOGIC;                             --リセット信号
      count   : out STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1) --カウンタ値
    );
  end component;

  component le_seq_blink
    Port (
      SYS_CLK   : in STD_LOGIC;                   --クロック信号
      SYS_RST   : in STD_LOGIC;                   --リセット信号
      le        : out  STD_LOGIC_VECTOR(0 to 3);  --単色 LED 出力信号
      le_timing : in  STD_LOGIC                   --タイミング信号
    );
  end component;

  component dynamic_ctrl
    Port (
      SYS_CLK     : in STD_LOGIC;                 --クロック信号
      SYS_RST     : in STD_LOGIC;                 --リセット信号
      nSEL        : out STD_LOGIC_VECTOR(0 to 2); --77 セグメント LED セレクト信号(負論理)
      seg7_timing : in  STD_LOGIC;                --7 セグタイミング信号
      seg_in1     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED1 の値
      seg_in2     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED2 の値
      seg_in3     : in  STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED3 の値
      seg_data    : out  STD_LOGIC_VECTOR(0 to 3) --4 ビットバイナリコード
    );
  end component;
```

```
  component seg7_decoder
    Port (
      SEG         : out STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
      seg_data    : in STD_LOGIC_VECTOR(0 to 3)   --4 ビットバイナリコード
    );
  end component;

begin
  slot_counter_0 : slot_counter
    Port map(
      SYS_CLK   => SYS_CLK,
      SYS_RST   => SYS_RST,
      count     => count
    );

  le_seq_blink_0 : le_seq_blink
    Port map(
      SYS_CLK   => SYS_CLK,                                  ❸
      SYS_RST   => SYS_RST,
      le        => le,
      le_timing => count(0)
    );

  dynamic_ctrl_0 : dynamic_ctrl
    Port map(
      SYS_CLK     => SYS_CLK,
      SYS_RST     => SYS_RST,
      nSEL        => nSEL,
      seg7_timing => count(8),
      seg_in1     => seg_in1,
      seg_in2     => seg_in2,
      seg_in3     => seg_in3,
      seg_data    => seg_data
    );

  seg7_decoder_0 : seg7_decoder
    Port map(
      SEG       => SEG,
      seg_data  => seg_data
    );
--トリガ信号が'1'の時順次点灯
  le_t  <= le and "1111" when le_trig = '1' else "0000";
  nLE   <= not le_t;      --外部に出力
  sw    <= not nSW;       --正論理にして入力
  code  <= not nCODE;     --正論理にして入力
  intr  <= count(4);      --カウンタの出力を割り込みコントローラに送る
end IMP;
```

❶　sil00u_core.vhd の入出力を定義

❷　4 つの回路のコンポーネント宣言

❸　4 つの回路のインスタンス

11.4. OPB インターフェースとコアを接続し、自作 IP コアを仕上げる

今まとめた回路(sil00u_core.vhd、slot_counter.vhd、dynamic_ctrl.vhd、

seg7_decoder.vhd、le_seq_blink.vhd)を

"C:\suzaku\sz***-yyyymmdd\pcores\opb_sil00_v1_00_a\hdl\vhdl"にコピーしてください。

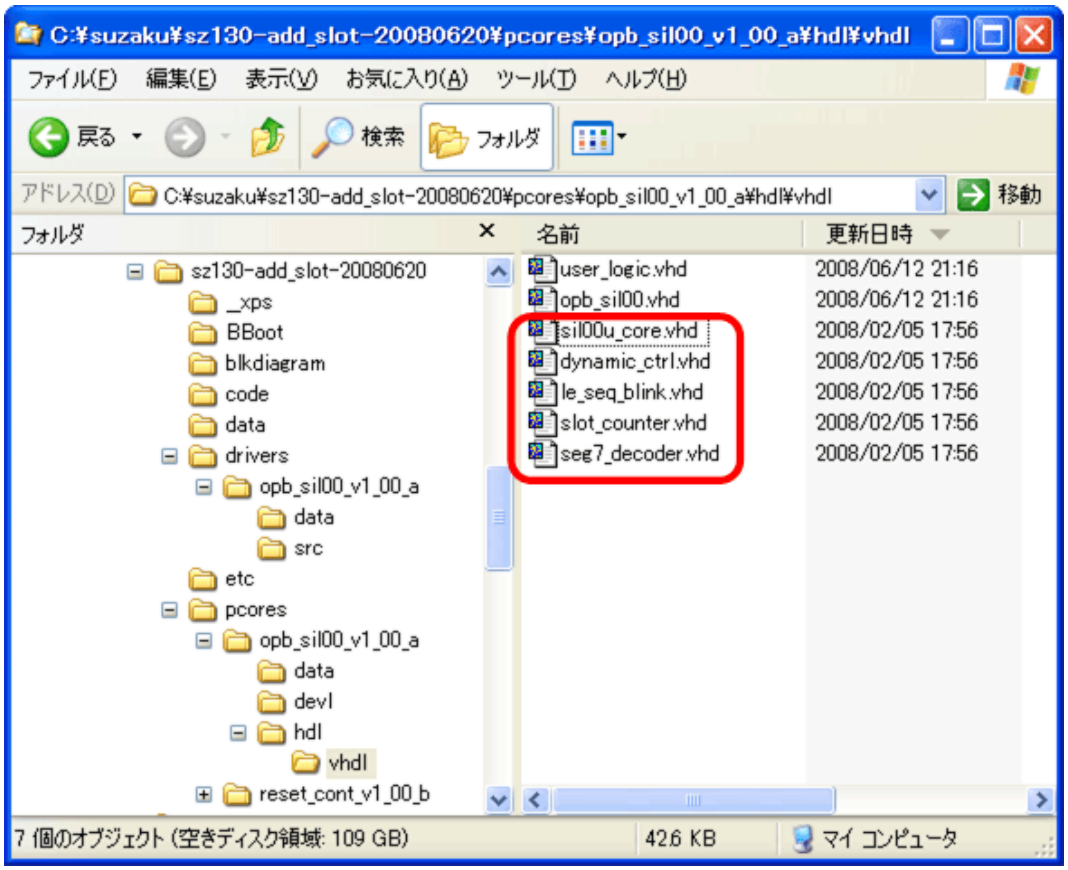


図 11.18. コアをコピー

11.4.1. user_logic.vhd

user_logic.vhd を開いてください。自動生成されたコードを編集していきます。user_logic を上位階層として、sil00u_core 回路を呼び出すソースコードを追加します。押しボタンスイッチ、ロータリコードスイッチは読み込むだけで書き込みは出来ません。この 2 つのために新たに、読み込み/書き込みレジスタではなく、読み込みレジスタを定義しています。ソースコードを追加するところには、大体--USER xxx added here とコメントが入っているので、目印にしてください。

例 11.2. opb_sil00(user_logic.vhd)

```
------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
------------------------------------------------------------------------------
```

```
--中略
-- DO NOT EDIT BELOW THIS LINE --------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE --------------------

 library opb_sil00_v1_00_a;      --ライブラリとして呼び出す
 use opb_sil00_v1_00_a.all;

------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_DWIDTH                        : integer              := 8;
    C_NUM_CE                        : integer              := 6;
    C_IP_INTR_NUM                   : integer              := 1
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------

    SEG   : out STD_LOGIC_VECTOR(0 to 7);    -- 7 セグメント LED 出力
    nSEL  : out STD_LOGIC_VECTOR(0 to 2);    -- セレクト出力
    nLE   : out STD_LOGIC_VECTOR(0 to 3);    -- 単色 LED 出力
    nSW   : in STD_LOGIC_VECTOR(0 to 2);     -- スイッチ入力
    nCODE : in STD_LOGIC_VECTOR(0 to 3);     -- ロータリ SW 入力
    intr  : out STD_LOGIC;

    -- ADD USER PORTS ABOVE THIS LINE ------------------
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk                      : in  std_logic;
    Bus2IP_Reset                    : in  std_logic;
    Bus2IP_Data                     : in  std_logic_vector(0 to C_DWIDTH-1);
    Bus2IP_BE                       : in  std_logic_vector(0 to C_DWIDTH/8-1);
    Bus2IP_RdCE                     : in  std_logic_vector(0 to C_NUM_CE-1);
    Bus2IP_WrCE                     : in  std_logic_vector(0 to C_NUM_CE-1);
    IP2Bus_Data                     : out std_logic_vector(0 to C_DWIDTH-1);
    IP2Bus_Ack                      : out std_logic;
    IP2Bus_Retry                    : out std_logic;
    IP2Bus_Error                    : out std_logic;
    IP2Bus_ToutSup                  : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
```

```
end entity user_logic;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------
architecture IMP of user_logic is

  signal slv_reg_r4  : std_logic_vector(0 to C_DWIDTH-1);    --押しボタンスイッチ用
  signal slv_reg_r5  : std_logic_vector(0 to C_DWIDTH-1);    --ロータリコードスイッチ用

  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
  signal slv_reg0     : std_logic_vector(0 to C_DWIDTH-1);   --7 セグメント LED1 用
  signal slv_reg1     : std_logic_vector(0 to C_DWIDTH-1);   --7 セグメント LED2 用
  signal slv_reg2     : std_logic_vector(0 to C_DWIDTH-1);   --7 セグメント LED3 用
  signal slv_reg3     : std_logic_vector(0 to C_DWIDTH-1);   --単色 LED トリガー用
  signal slv_reg4     : std_logic_vector(0 to C_DWIDTH-1);
  signal slv_reg5     : std_logic_vector(0 to C_DWIDTH-1);
  signal slv_reg_write_select    : std_logic_vector(0 to 5);
  signal slv_reg_read_select     : std_logic_vector(0 to 5);
  signal slv_ip2bus_data         : std_logic_vector(0 to C_DWIDTH-1);
  signal slv_read_ack            : std_logic;
  signal slv_write_ack           : std_logic;

  ------------------------------------------
  -- Signals for user logic interrupt example
  ------------------------------------------
  signal interrupt    : std_logic_vector(0 to C_IP_INTR_NUM-1); --割り込み用
begin

 -- 下位モジュール呼び出し sil00u_core インスタンス
 sil00u_core_0 : entity opb_sil00_v1_00_a.sil00u_core
  PORT MAP(
    SYS_CLK => Bus2IP_Clk,
    SYS_RST => Bus2IP_Reset,

    -- External
    SEG   => SEG,
    nSEL  => nSEL,
    nLE   => nLE,
    nSW   => nSW,
    nCODE => nCODE,

    -- R/W
    seg_in1 => slv_reg0(4 to 7),
    seg_in2 => slv_reg1(4 to 7),
    seg_in3 => slv_reg2(4 to 7),
    le_trig => slv_reg3(7),

    -- Read
    sw   => slv_reg_r4(5 to 7),
    code => slv_reg_r5(4 to 7),
    intr => interrupt(0)
  );

--中略
  slv_reg_write_select <= Bus2IP_WrCE(0 to 5);
```

```
  slv_reg_read_select  <= Bus2IP_RdCE(0 to 5);
  slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2)
                          or Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or Bus2IP_WrCE(5);
  slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2)
                          or Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or Bus2IP_RdCE(5);
  -- implement slave model register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
        slv_reg0 <= (others => '0');
        slv_reg1 <= (others => '0');
        slv_reg2 <= (others => '0');
        slv_reg3 <= (others => '0');
        slv_reg4 <= (others => '0');
        slv_reg5 <= (others => '0');
      else
        case slv_reg_write_select is
          when "100000" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                 slv_reg0(byte_index*8 to byte_index*8+7)
                   <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "010000" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg1(byte_index*8 to byte_index*8+7)
                  <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "001000" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg2(byte_index*8 to byte_index*8+7)
                  <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "000100" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg3(byte_index*8 to byte_index*8+7)
                  <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "000010" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg4(byte_index*8 to byte_index*8+7)
                  <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
          when "000001" =>
            for byte_index in 0 to (C_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg5(byte_index*8 to byte_index*8+7)
                  <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
```

```
              end if;
            end loop;
          when others => null;
        end case;
      end if;
    end if;
  end process SLAVE_REG_WRITE_PROC;
  -- implement slave model register read mux

-- CPU からのレジスタ読み込み
 -- SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0,
 -- slv_reg1, slv_reg2, slv_reg3, slv_reg4, slv_reg5 ) is
  SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1,
                                 slv_reg2, slv_reg3, slv_reg_r4, slv_reg_r5 ) is

  begin
    case slv_reg_read_select is
      when "100000" => slv_ip2bus_data <= slv_reg0;
      when "010000" => slv_ip2bus_data <= slv_reg1;
      when "001000" => slv_ip2bus_data <= slv_reg2;
      when "000100" => slv_ip2bus_data <= slv_reg3;

      -- when "000010" => slv_ip2bus_data <= slv_reg4;
      -- when "000001" => slv_ip2bus_data <= slv_reg5;
      when "000010"    => slv_ip2bus_data <= slv_reg_r4;
      when "000001"    => slv_ip2bus_data <= slv_reg_r5;

      when others => slv_ip2bus_data <= (others => '0');
    end case;
  end process SLAVE_REG_READ_PROC;
  ------------------------------------------
  -- Example code to generate user logic interrupts
  --
  -- Note:
  -- The example code presented here is to show you one way of generating
  -- interrupts from the user logic. This code snippet infers a counter
  -- and generate the interrupts whenever the counter rollover (the counter
  -- will rollover ~21 sec @50Mhz).

 ------------------------------------------
  -- INTR_PROC : process( Bus2IP_Clk ) is
  --  constant COUNT_SIZE : integer := 30;
  --  constant ALL_ONES : std_logic_vector(0 to COUNT_SIZE-1) := (others => '1');
  --  variable counter : std_logic_vector(0 to COUNT_SIZE-1);
  -- begin
  --
  --   if ( Bus2IP_Clk'event and Bus2IP_Clk = '1' ) then
  --     if ( Bus2IP_Reset = '1' ) then
  --       counter := (others => '0');
  --       interrupt <= (others => '0');
  --     else
  --       counter := counter + 1;
  --         if ( counter = ALL_ONES ) then
  --           interrupt <= (others => '1');
  --         else
  --           interrupt <= (others => '0');
  --         end if;
  --       end if;
```

```
  --     end if;
  --
  --   end process INTR_PROC;

  IP2Bus_IntrEvent <= interrupt;  -- 割り込みを counter からの出力に接続

  ------------------------------------------
  -- Example code to drive IP to Bus signals
  ------------------------------------------
  IP2Bus_Data        <= slv_ip2bus_data;
  IP2Bus_Ack         <= slv_write_ack or slv_read_ack;
  IP2Bus_Error       <= '0';
  IP2Bus_Retry       <= '0';
  IP2Bus_ToutSup     <= '0';

end IMP;
```

11.4.1.1. ライブラリ

ライブラリはデザインデータの集まりで、パッケージ宣言、エンティティ宣言、アーキテクチャ宣言などで構成されます。ライブラリを使用すると、コンポーネント宣言を省略することができます。

```
library opb_sil00_v1_00_a;
use opb_sil00_v1_00_a.all;
```

11.4.2. opb_sil00.vhd

`opb_sil00.vhd` を開いてください。自動生成されたコードを編集していきます。opb_sil00 を上位階層として、user_logic 回路を呼び出すコードを追加します。

例 11.3. opb_sil00(opb_sil00.vhd)

```
------------------------------------------------------------------------------
-- opb_sil00.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- 中略
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
use proc_common_v2_00_a.ipif_pkg.all;
library opb_ipif_v3_01_c;
use opb_ipif_v3_01_c.all;

library opb_sil00_v1_00_a;
use opb_sil00_v1_00_a.all;

------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity opb_sil00 is
```

```
  generic
  (
    --中略
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------

    SEG   : out  STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
    nSEL  : out  STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号
    nLE   : out  STD_LOGIC_VECTOR(0 to 3); --単色 LED への出力信号
    nSW   : in STD_LOGIC_VECTOR(0 to 2);   --押しボタンスイッチからの入力信号
    nCODE : in STD_LOGIC_VECTOR(0 to 3);   --ロータリコードスイッチからの入力信号

    -- ADD USER PORTS ABOVE THIS LINE ------------------
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add to or delete
    OPB_Clk                        : in  std_logic;
    OPB_Rst                        : in  std_logic;
    Sl_DBus                        : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sl_errAck                      : out std_logic;
    Sl_retry                       : out std_logic;
    Sl_toutSup                     : out std_logic;
    Sl_xferAck                     : out std_logic;
    OPB_ABus                       : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE                         : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus                       : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW                        : in  std_logic;
    OPB_select                     : in  std_logic;
    OPB_seqAddr                    : in  std_logic;
    IP2INTC_Irpt                   : out std_logic
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  attribute SIGIS : string;
  attribute SIGIS of OPB_Clk       : signal is "Clk";
  attribute SIGIS of OPB_Rst       : signal is "Rst";
  attribute SIGIS of IP2INTC_Irpt  : signal is "INTR_LEVEL_HIGH";

end entity opb_sil00;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------
architecture IMP of opb_sil00 is
--中略
begin
  ------------------------------------------
  -- instantiate the OPB IPIF
  ------------------------------------------
  OPB_IPIF_I : entity opb_ipif_v3_01_c.opb_ipif
    generic map
    (
--中略
    )
    port map
    (
--中略
    );
```

```
  ------------------------------------------
  -- instantiate the User Logic
  ------------------------------------------
  USER_LOGIC_I : entity opb_sil00_v1_00_a.user_logic
    generic map
    (
--中略
    )
    port map
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------

      SEG => SEG,
      nSEL => nSEL,
      nLE => nLE,
      nSW => nSW,
      nCODE => nCODE,

      -- MAP USER PORTS ABOVE THIS LINE ------------------
      Bus2IP_Clk                     => iBus2IP_Clk,
      Bus2IP_Reset                   => iBus2IP_Reset,
      Bus2IP_Data                    => uBus2IP_Data,
      Bus2IP_BE                      => uBus2IP_BE,
      Bus2IP_RdCE                    => uBus2IP_RdCE,
      Bus2IP_WrCE                    => uBus2IP_WrCE,
      IP2Bus_Data                    => uIP2Bus_Data,
      IP2Bus_Ack                     => iIP2Bus_Ack,
      IP2Bus_Retry                   => iIP2Bus_Retry,
      IP2Bus_Error                   => iIP2Bus_Error,
      IP2Bus_ToutSup                 => iIP2Bus_ToutSup
    );
--中略
end IMP;
```

11.4.3. opb_sil00_v2_1_0.mpd

"C:\suzaku\sz***- yyyymmdd \pcores\opb_sil00_v1_00_a\data"を開いてください。

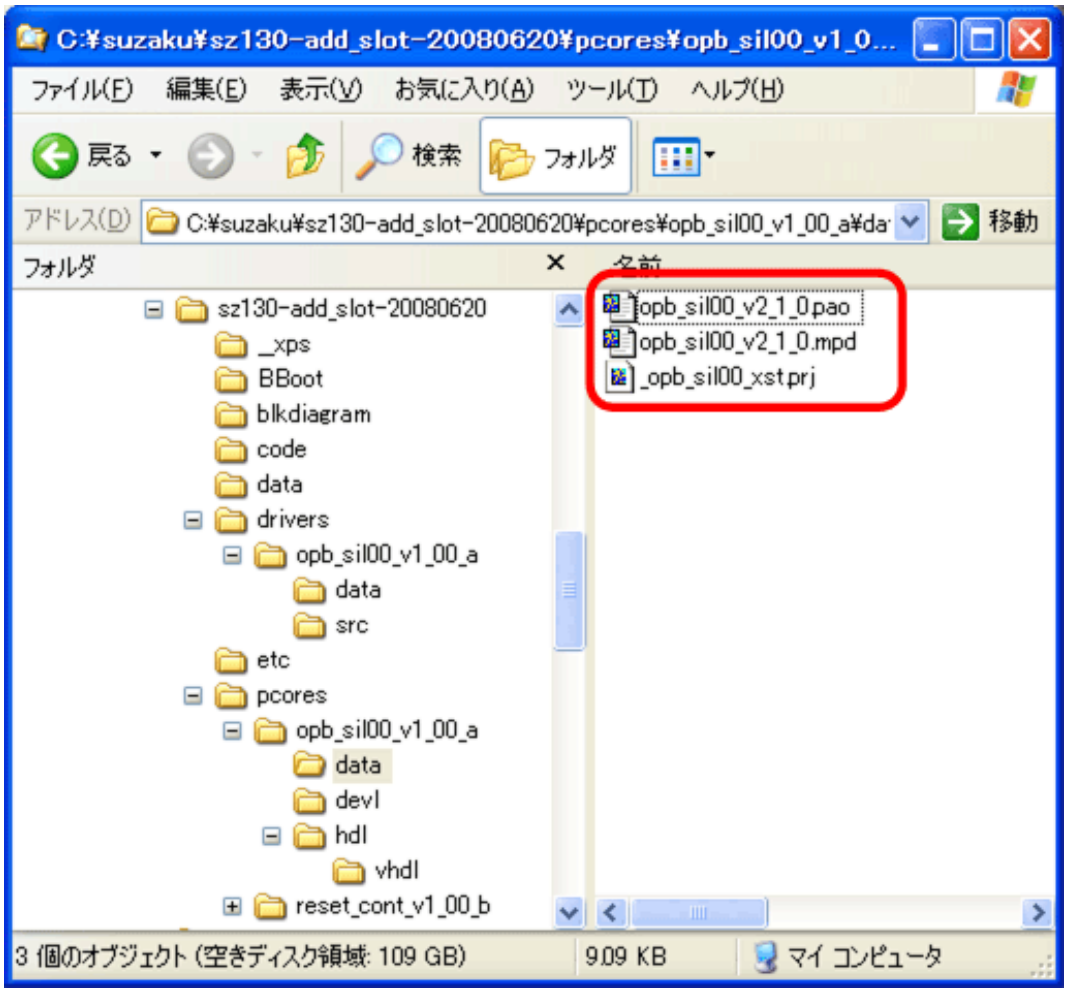


図 11.19. フォルダ構成

`opb_sil00_v2_1_0.mpd` を編集します。mpd(Microprocessor Peripheral Definition)ファイルでは信号の入出力の方向やビット幅等を定義できます。7 セグメント LED、7 セグメント LED セレクト、単色 LED、押しボタンスイッチ、ロータリコードスイッチの信号を外部と接続できるように定義します。

以下の文を一番下に追加してください。

例 11.4. opb_sil00_v2_1_0.mpd

```
PORT SEG    = "", DIR = O, VEC = [0:7]
PORT nSEL   = "", DIR = O, VEC = [0:2]
PORT nLE    = "", DIR = O, VEC = [0:3]
PORT nSW    = "", DIR = I, VEC = [0:2]
PORT nCODE  = "", DIR = I, VEC = [0:3]
```

11.4.4. opb_sil00_v2_1_0.pao

`opb_sil00_v2_1_0.pao` を編集します。pao(Peripheral Analyze Order)ファイルはペリフェラルのコンパイル(構成およびシミュレーション用)に必要な HDL ファイルと、その解析順を指定します。自分で書いたソースコードを追加します。

以下の文を一番下に追加してください。

例 11.5. opb_sil00_v2_1_0.pao

```
lib opb_sil00_v1_00_a sil00u_core vhdl
lib opb_sil00_v1_00_a slot_counter vhdl
lib opb_sil00_v1_00_a le_seq_blink vhdl
lib opb_sil00_v1_00_a seg7_decoder vhdl
lib opb_sil00_v1_00_a dynamic_ctrl vhdl
```

11.4.5. opb_sil00.c

"C:\suzaku\sz***- yyyymmdd \drivers\opb_sil00_v1_00_a\src\opb_sil00.c"を編集します。SUZAKU では stdio を使用しておらず、xil_printf( )は生成されません。

OPB_SIL00_Intr_DefaultHandler 関数の中に記述されている xil_printf( )の行をコメントアウトしてください。

例 11.6. opb_sil00.c

```
void OPB_SIL00_Intr_DefaultHandler(void * baseaddr_p)
{
  Xuint32 baseaddr;
  Xuint32 IntrStatus;
  Xuint32 IpStatus;

  baseaddr = (Xuint32) baseaddr_p;

  /*
   * Get status from Device Interrupt Status Register.
   */
  IntrStatus = OPB_SIL00_mReadReg(baseaddr, OPB_SIL00_INTR_DISR_OFFSET);

//  xil_printf("Device Interrupt! DISR value : 0x%08x \n\r", IntrStatus);

  /*
   * Verify the source of the interrupt is the user logic and clear the interrupt
   * source by toggle write baca to the IP ISR register.
   */
  if ( (IntrStatus & INTR_IPIR_MASK) == INTR_IPIR_MASK )
  {
//    xil_printf("User logic interrupt! \n\r");
    IpStatus = OPB_SIL00_mReadReg(baseaddr, OPB_SIL00_INTR_ISR_OFFSET);
    OPB_SIL00_mWriteReg(baseaddr, OPB_SIL00_INTR_ISR_OFFSET, IpStatus);
  }
}
```

これで自作 IP コアの完成です。

11.5. スロットマシンのコアの構成 (XPS)

SZ410 以外をお使いの場合は「11.9. 自作 IP コアの追加」へ進んでください。

スロットマシンのコアを"xps_sil00"という名前で下図のように製作します。

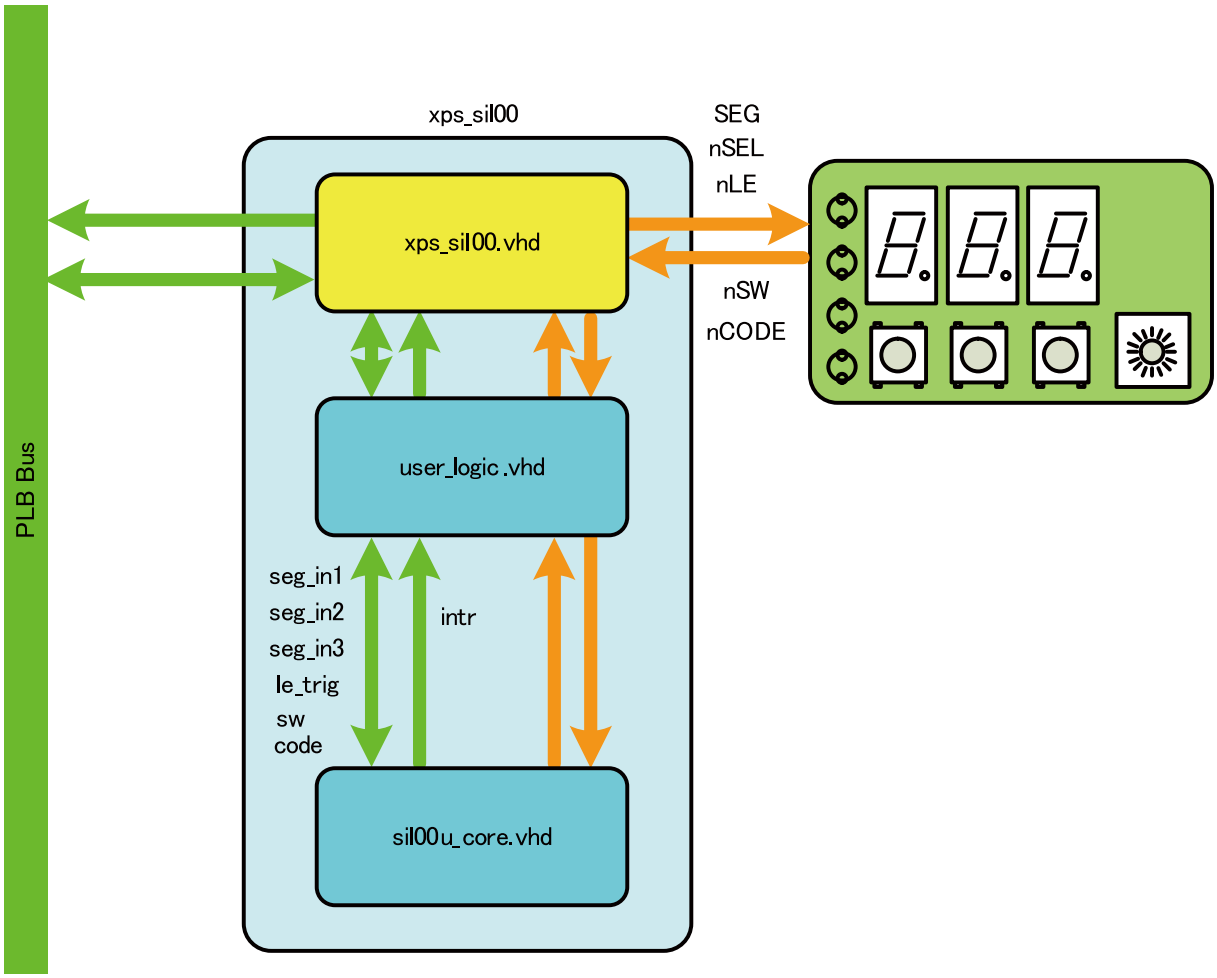
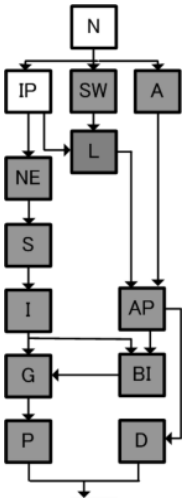


図 11.20. 自作 IP コア

11.6. ウィザードを使って XPS インターフェースをつくる



XPS バスに接続するインターフェースをつくります。EDK にはインターフェースを作るウィザードが用意されているので、簡単にインターフェースをつくることが出来ます。スロットマシンのコアを CPU から制御するためには、バスに接続しなければいけません。

[Hardware]→[Create or Import Peripheral...]をクリックしてください。

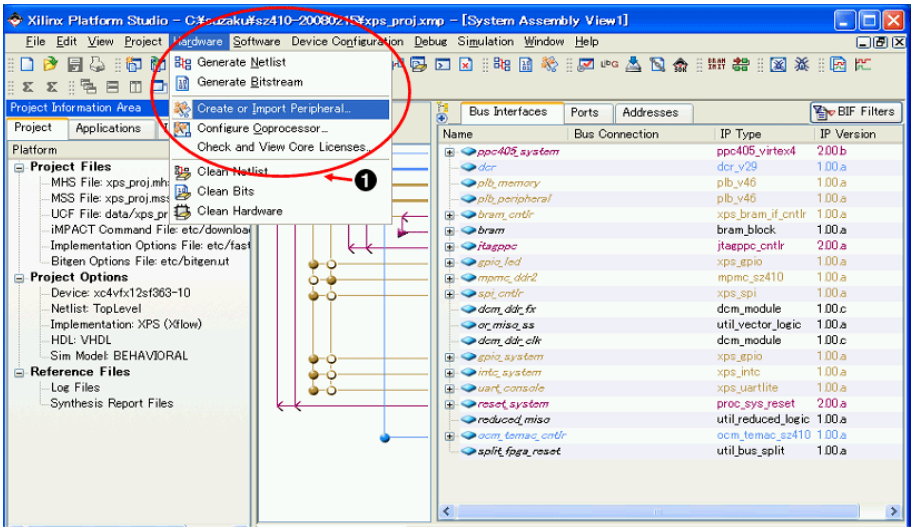


図 11.21. Create and Import Peripheral Wizard の起動のさせ方

❶　[Hardware]→[Create or Import Peripheral...]をクリック

Create and Import Peripheral Wizard が立ち上がります。[Next]をクリックして下さい。

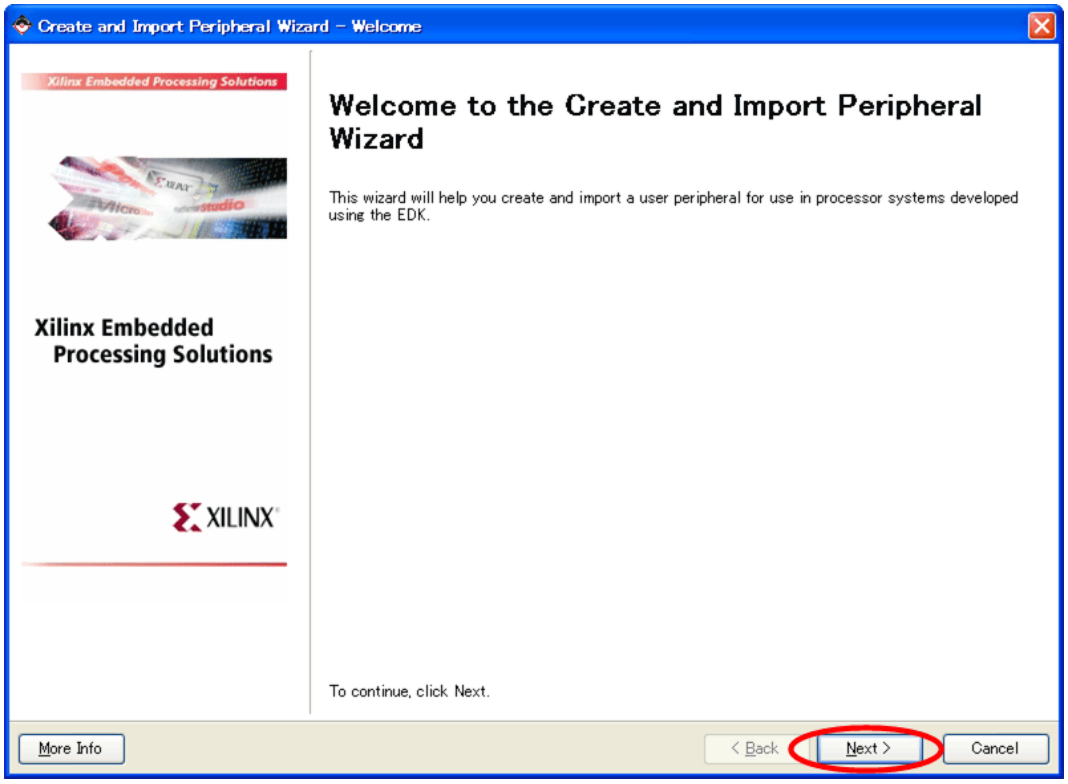


図 11.22. Create and Import Peripheral Wizard 起動画面

Create and Import Peripheral Wizard が立ち上がります。新規で作るので Select flow の[Create templates for a new peripheral]をチェックして[Next]をクリックしてください。

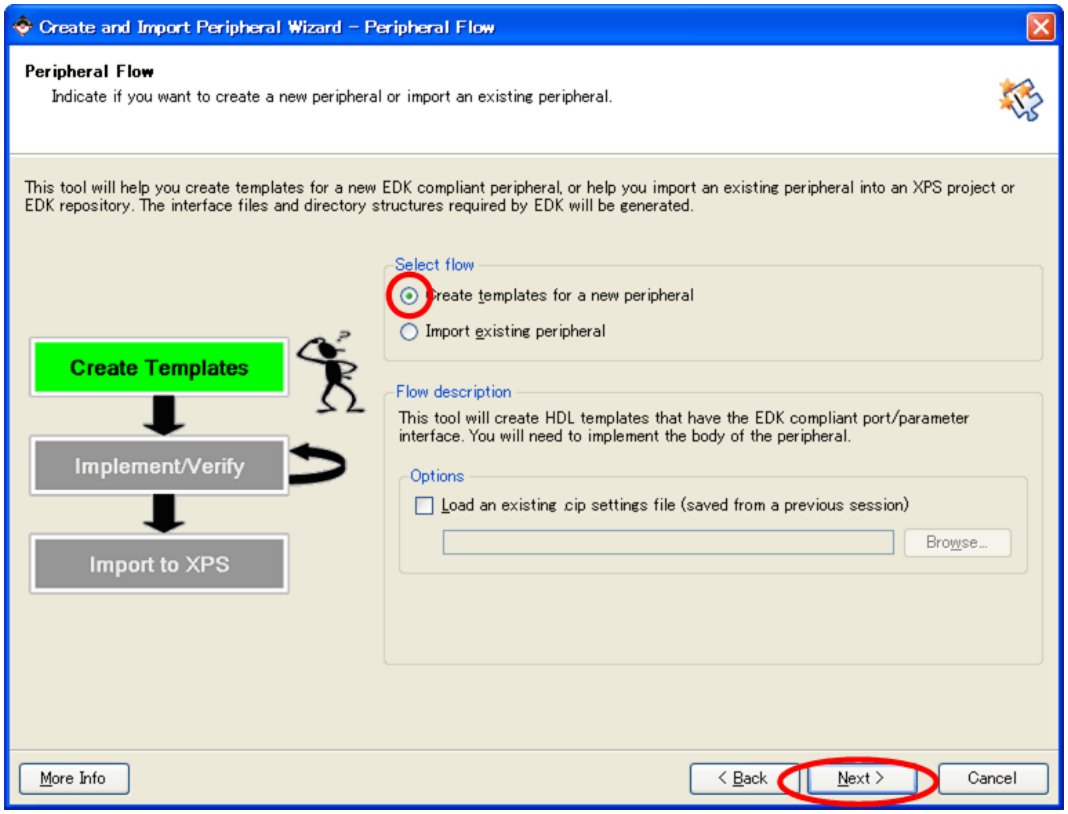


図 11.23. Peripheral Flow

コアを生成する場所を指定します。[To an XPS project]をチェックし、[Next]をクリックして下さい。

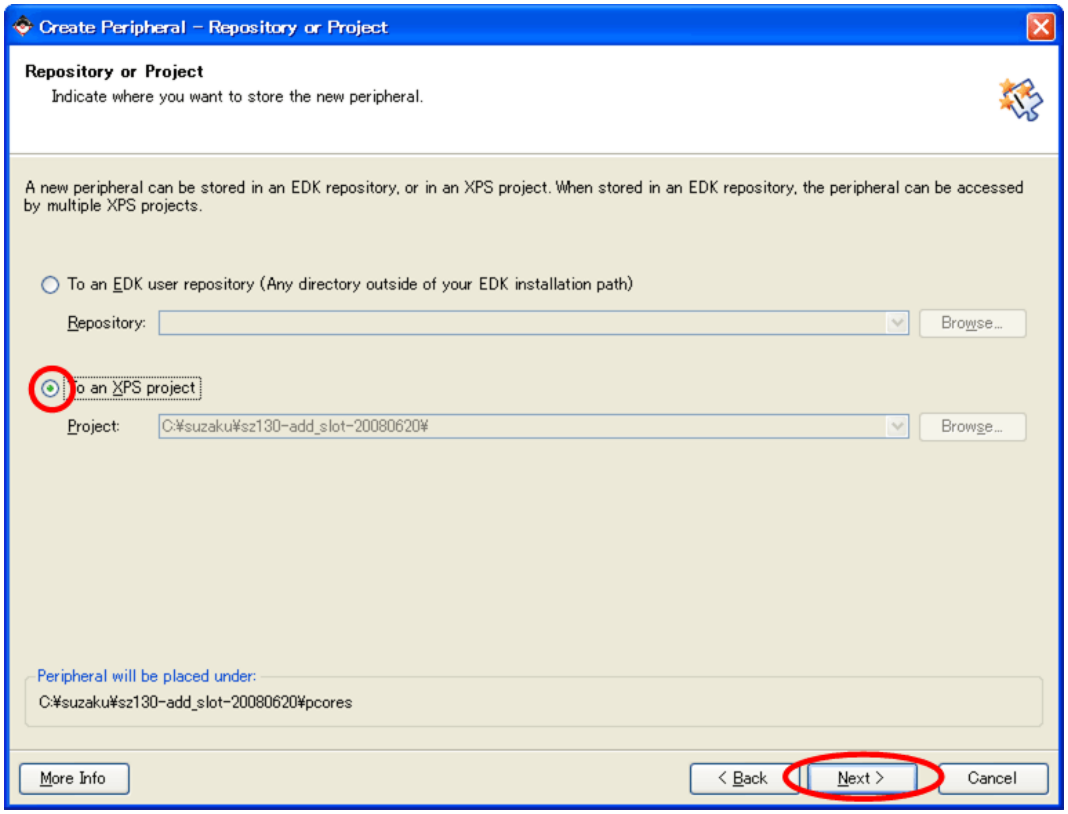


図 11.24. コアの生成場所の指定

コアに名前をつけます。[Name]に名前を入力してください。xps_sil00 とします。

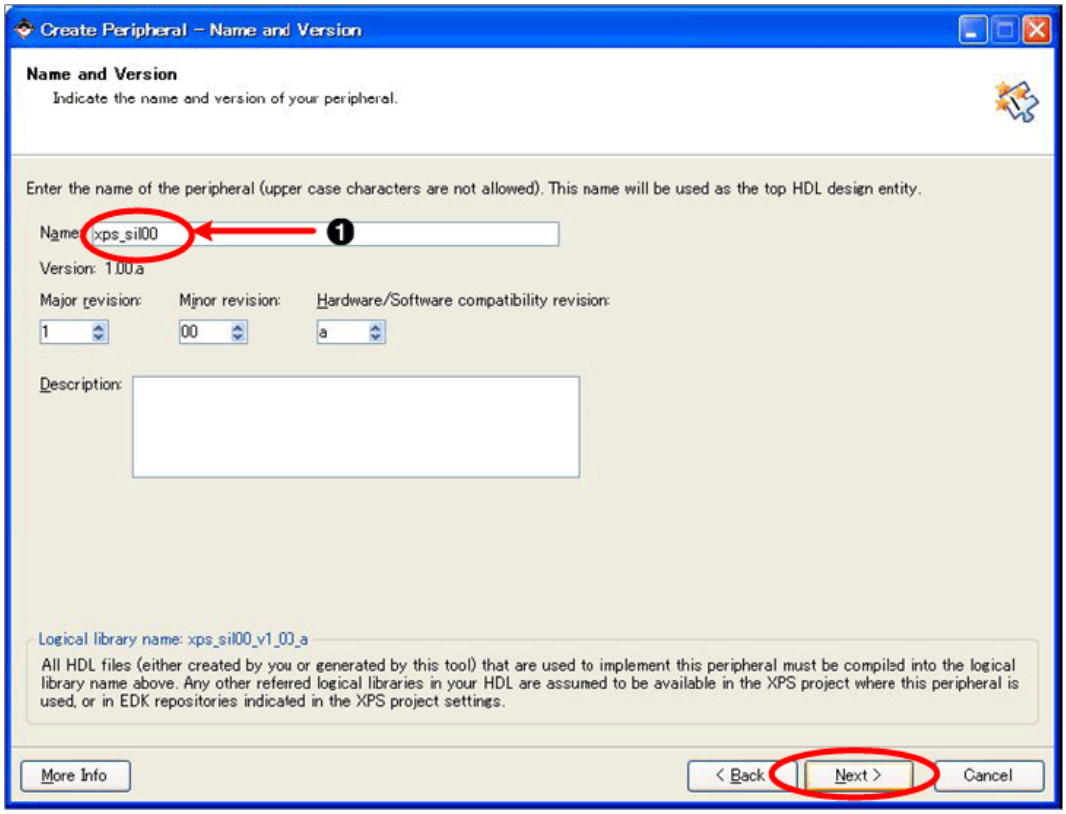


図 11.25. コアの名前

❶　xps_sil00 と入力

バスを選択します。[Processor Local Bus(PLB v4.6)]を選択して[Next]をクリックしてください。

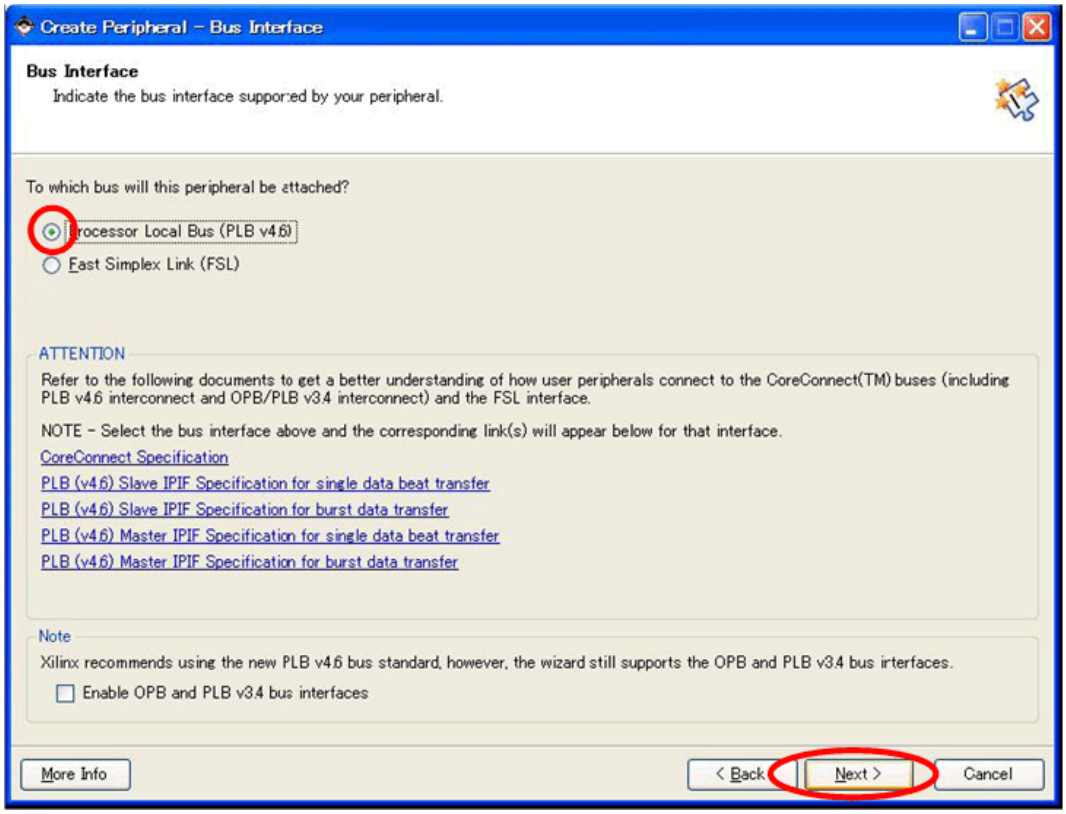


図 11.26. バスの選択

IPIF にはアドレスのデコード、バイト調整などの基本的な機能に加えて、ペリフェラルの作成を大幅に簡略化するオプションの機能が備わっています。選択した項目により PLB ペリフェラルテンプレートが生成されます。

今回は[Interrupt control]、[User logic software register]を選択し、[Next]をクリックしてください。割り込みのユーザーテンプレート、ソフトウェアアクセス可能レジスタが生成されるユーザーテンプレートが追加されます。

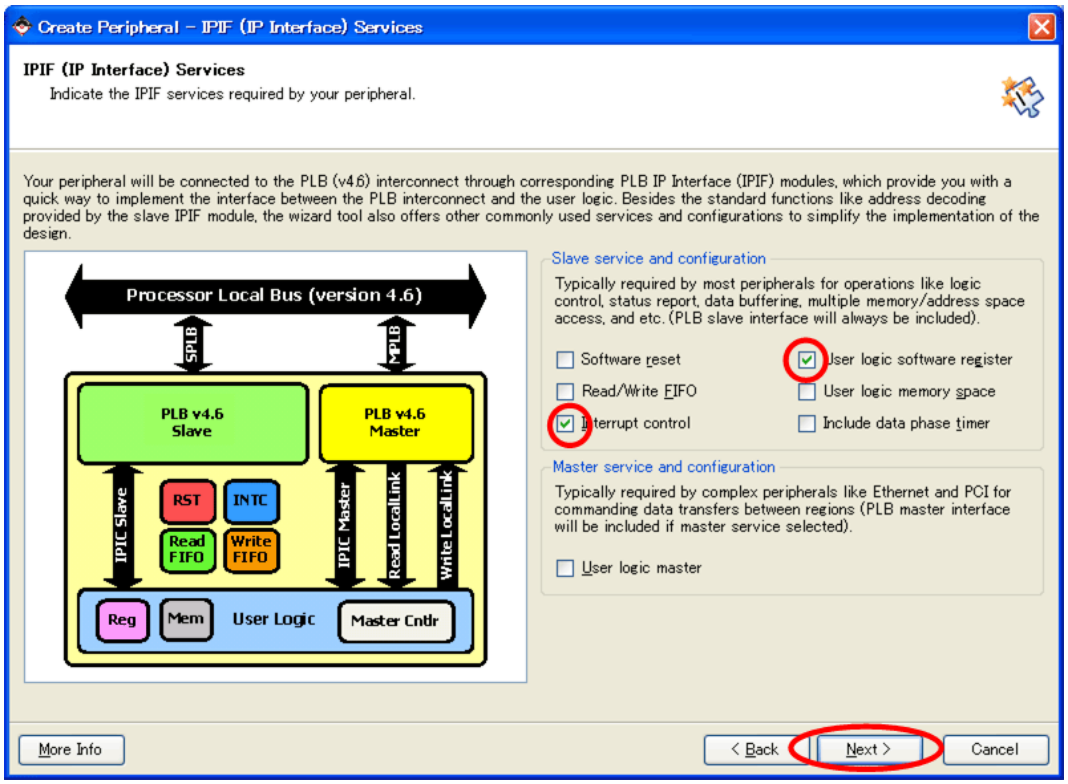


図 11.27. テンプレート追加

Slave Interface の設定ができます。ここでは何も変更せず[Next]をクリックしてください。

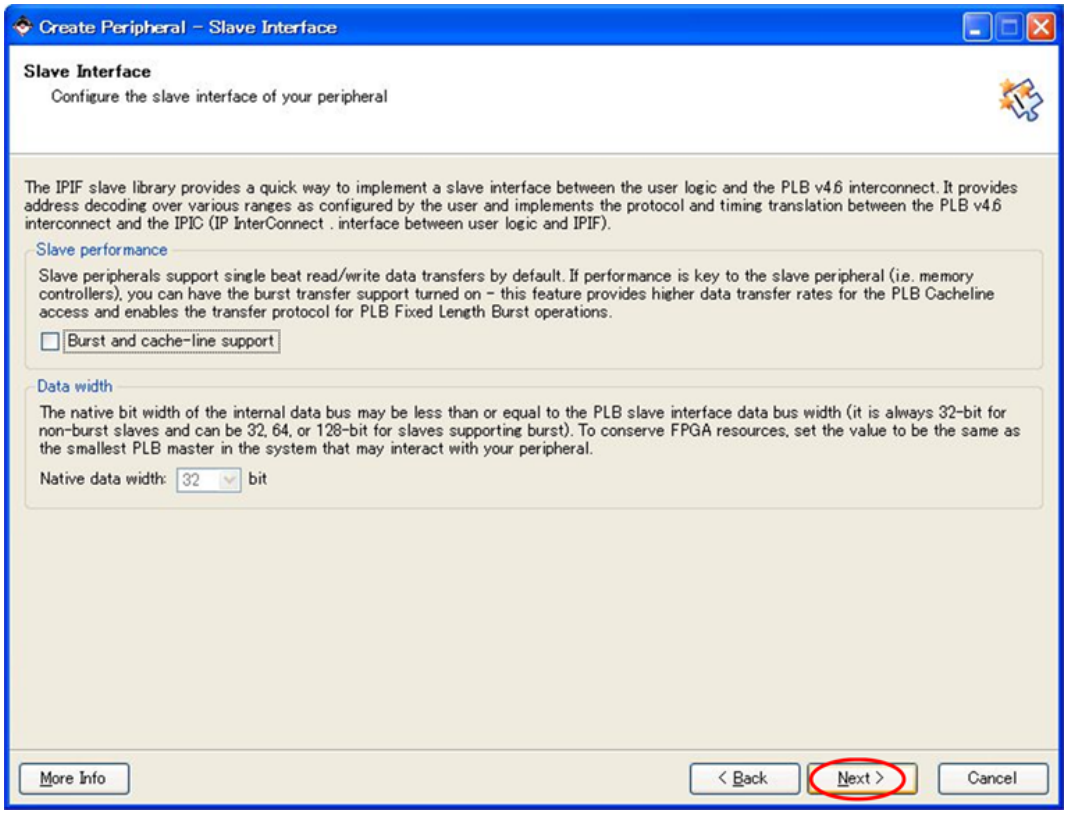


図 11.28. Slave Interface の設定

割り込みの設定をします。割り込みに使用するカウンタの Duty が 50%なのでエッジ取り込みにし、今回は立ち上がりエッジを使用します。[Use Device ISC(interrupt source controller)]のチェックボタンをはずし、 Interrupt capture mode を[Rising Edge Detect]に設定して、[Next]をクリックして下さい。

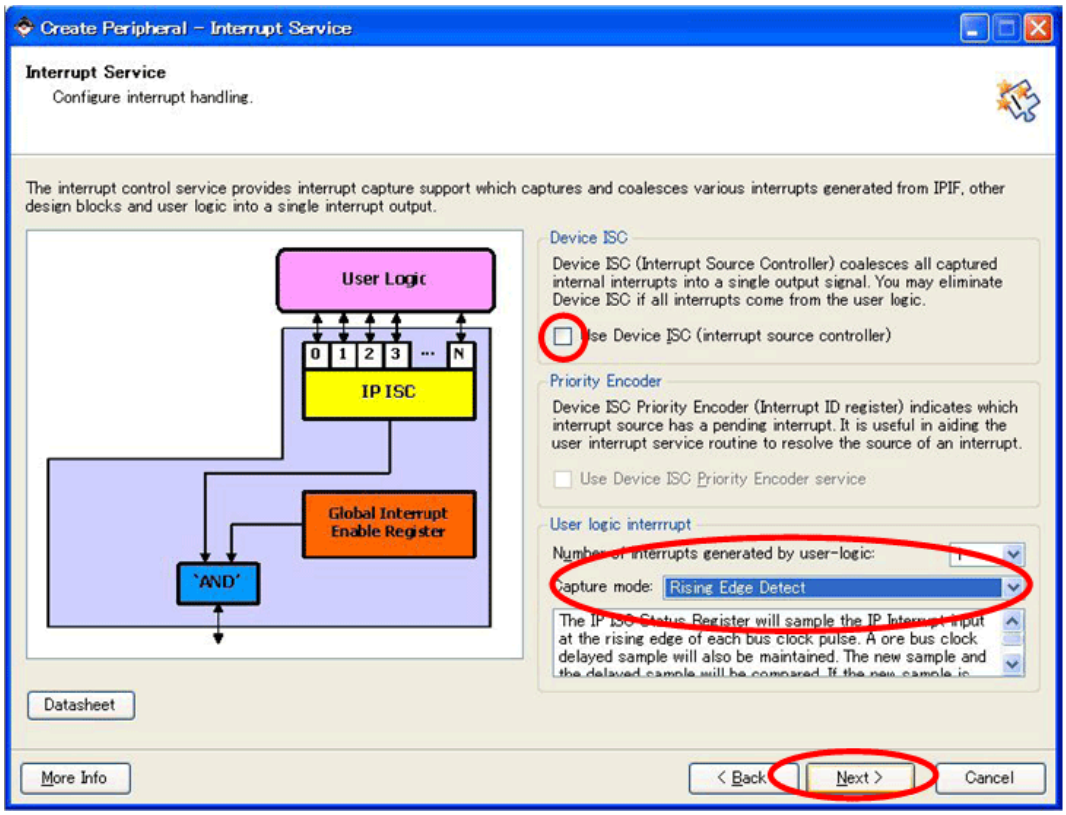


図 11.29. Interrupt 設定

ソフトウェアアクセス可能レジスタ数を指定します。今回必要となるのは書き込み/読み込みレジスタ1 つ(7 セグメント LED の値を設定する 3byte と単色 LED のトリガ信号を設定する 1byte)と読み込みレジスタ 1 つ(押しボタンスイッチの情報をやり取りする 1byte、ロータリコードスイッチの情報をやり取りする 1byte)です。

[Number of software accessible resiters]を 2 にし、[Next]をクリックしてください。

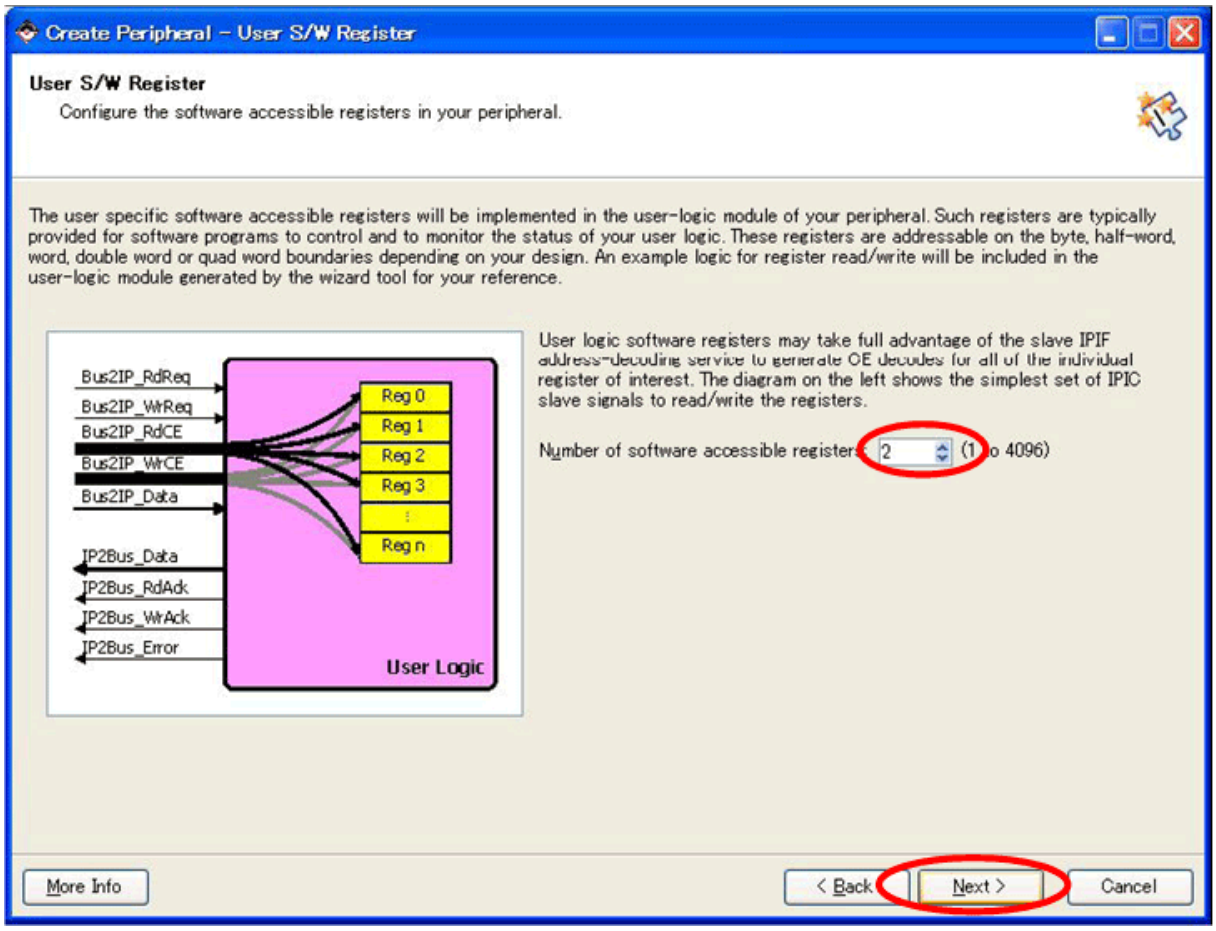


図 11.30. レジスタ数指定

IPIC を設定します。すでにいくつか ON になっていますが、IPIF Services ページで指定した機能をインプリメントするために必要なものに自動でチェックされています。このまま[Next]をクリックしてください。

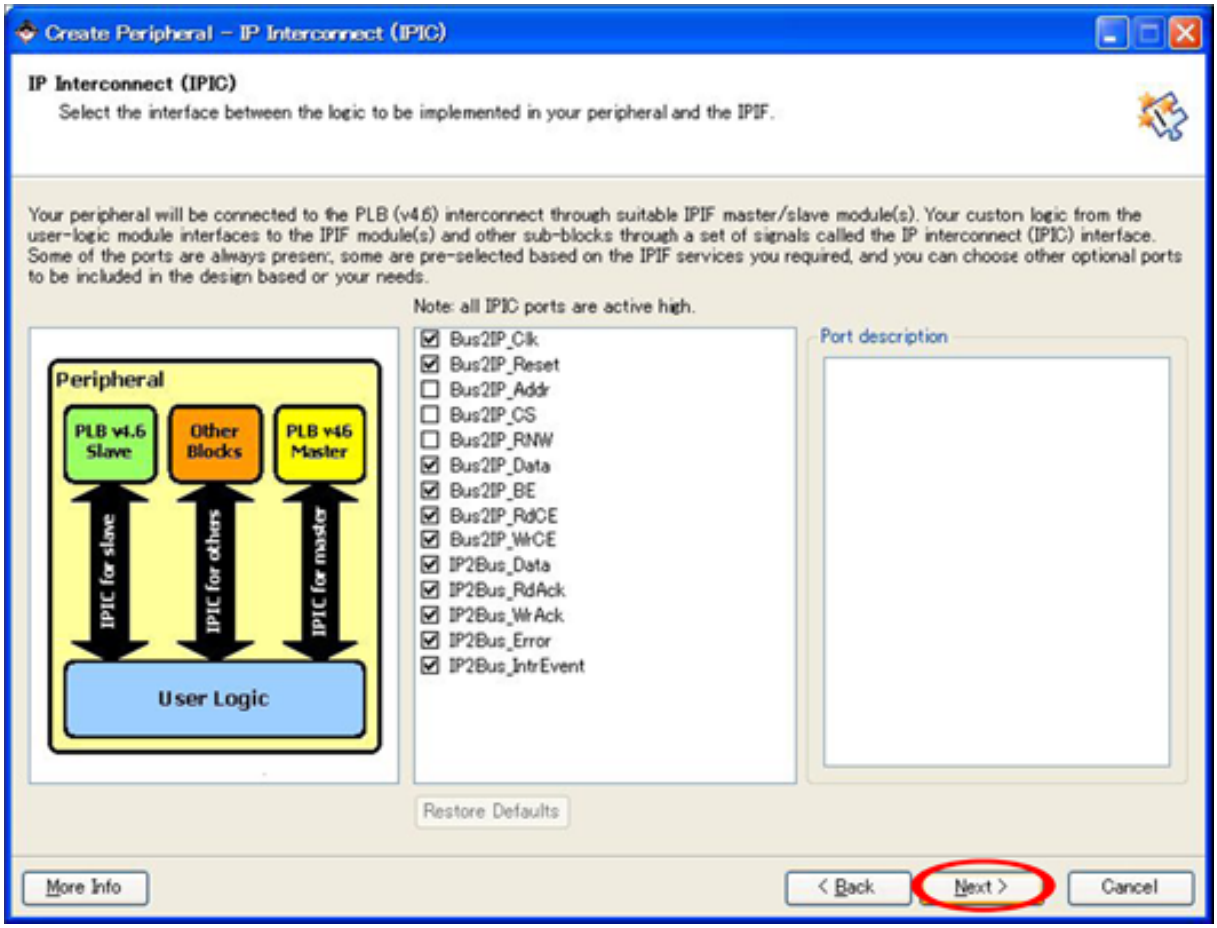


図 11.31. IPIC 設定

ここを ON にすると、カスタムロジックおよび機能のシミュレーションに使用するサポートファイルを生成できますが、今回は使いません。そのまま[Next]をクリックしてください。

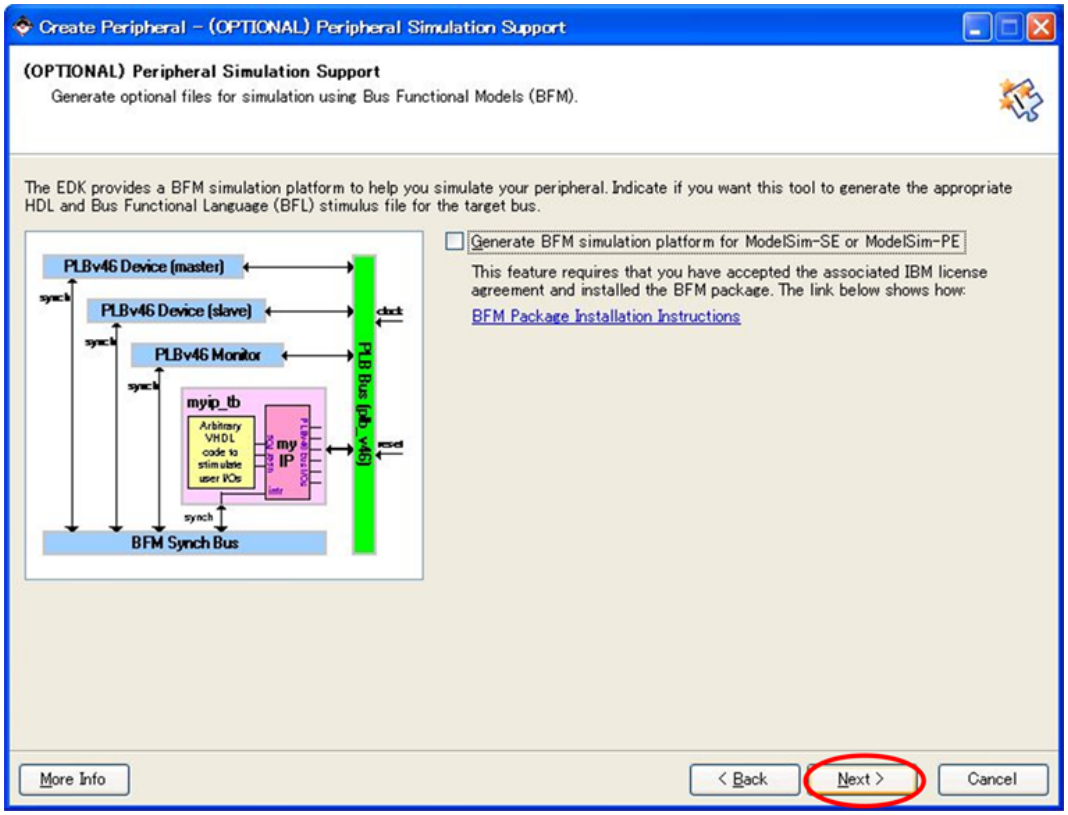


図 11.32. サポートファイル生成確認

下図の項目をチェックし、[Next]をクリックしてください。

ソフトフェアドライバテンプレートファイルとドライバディレクトリ構成が作成されます。

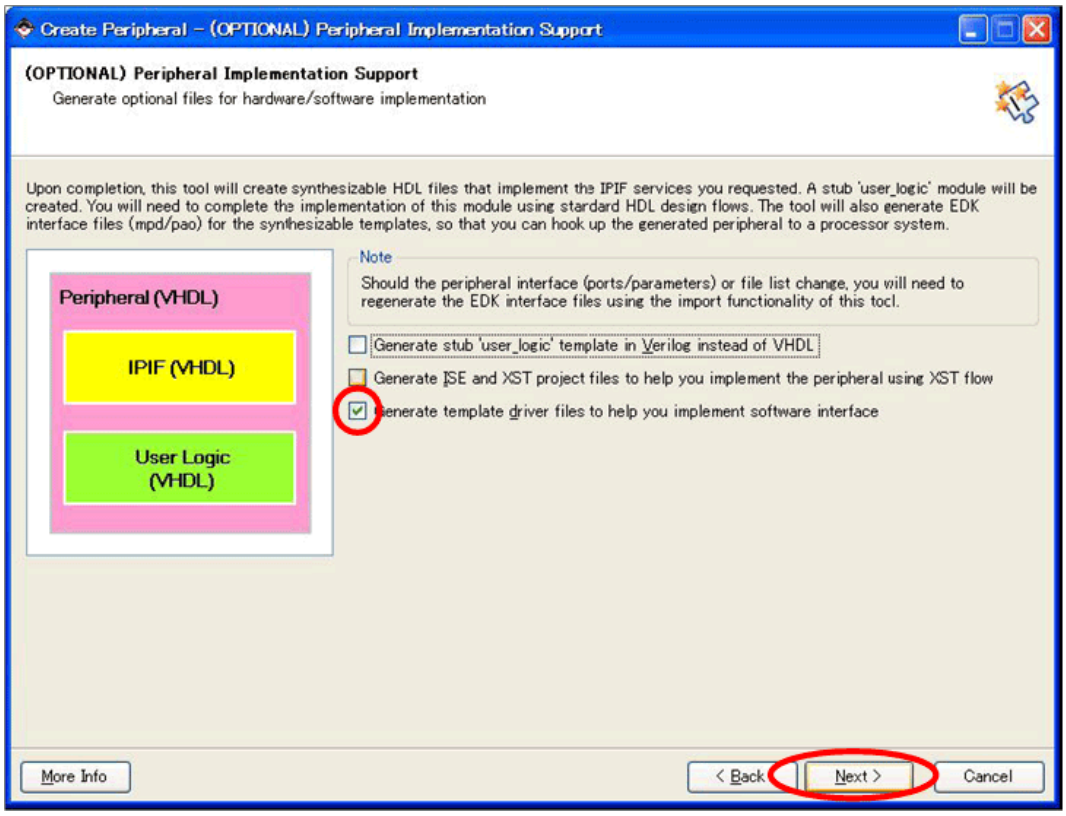


図 11.33. オプション設定

以上で終了です。[Finish]をクリックしてください。

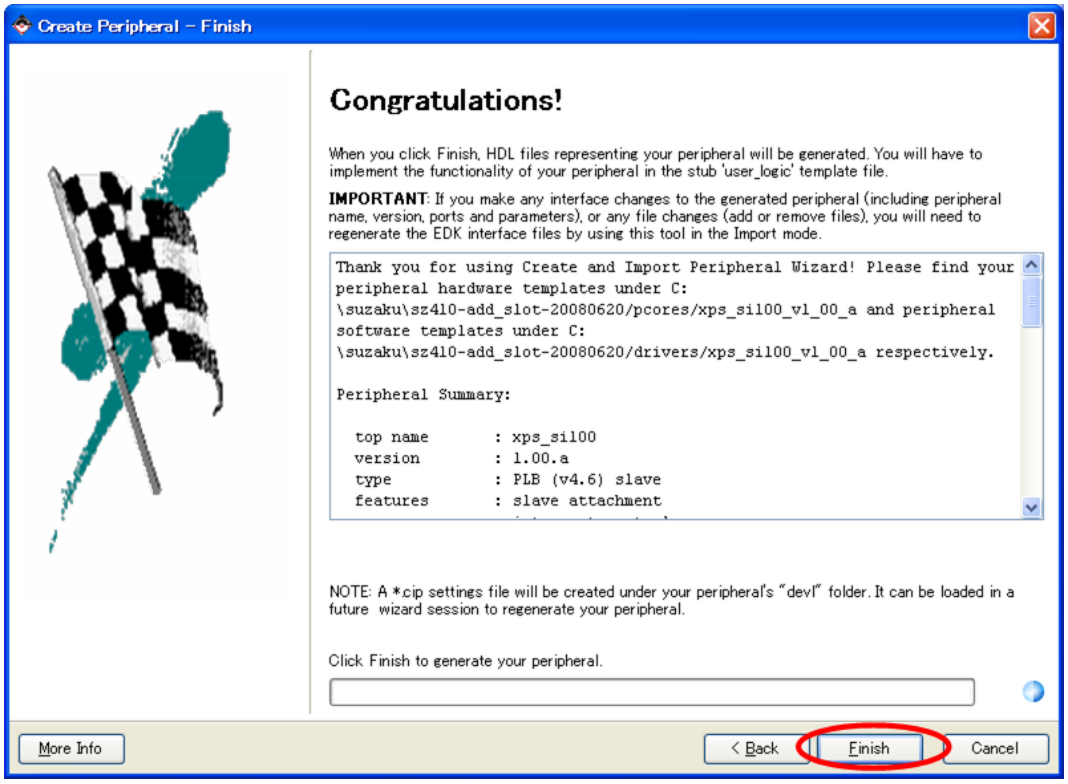


図 11.34. 終了

"C:\suzaku\sz***- yyyymmdd \pcores"の下に PLB バスに接続するインターフェースの雛形が生成されます。

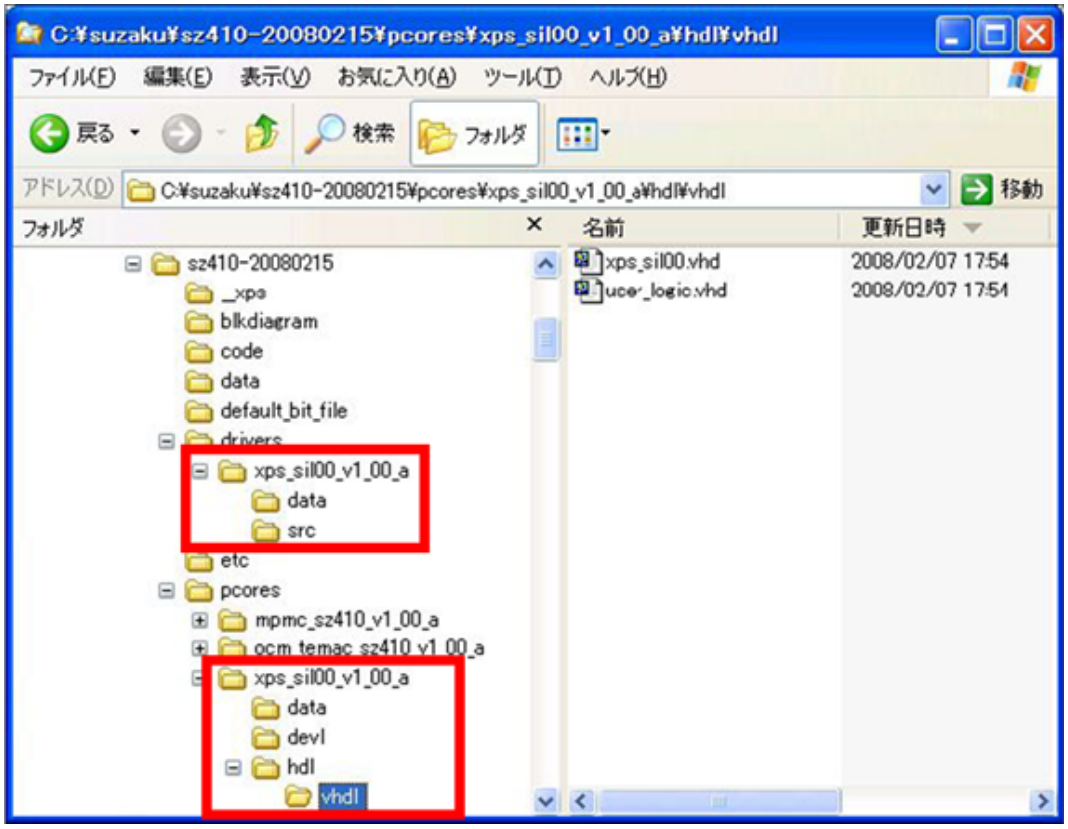


図 11.35. フォルダ構成

11.7. 今まで作ってきた回路をまとめる (XPS)

下図の仕様で今まで作ってきた回路を PLB バスに接続できるようにまとめます。`sil00u_core.vhd` をテキストエディタ等で新規作成してください。

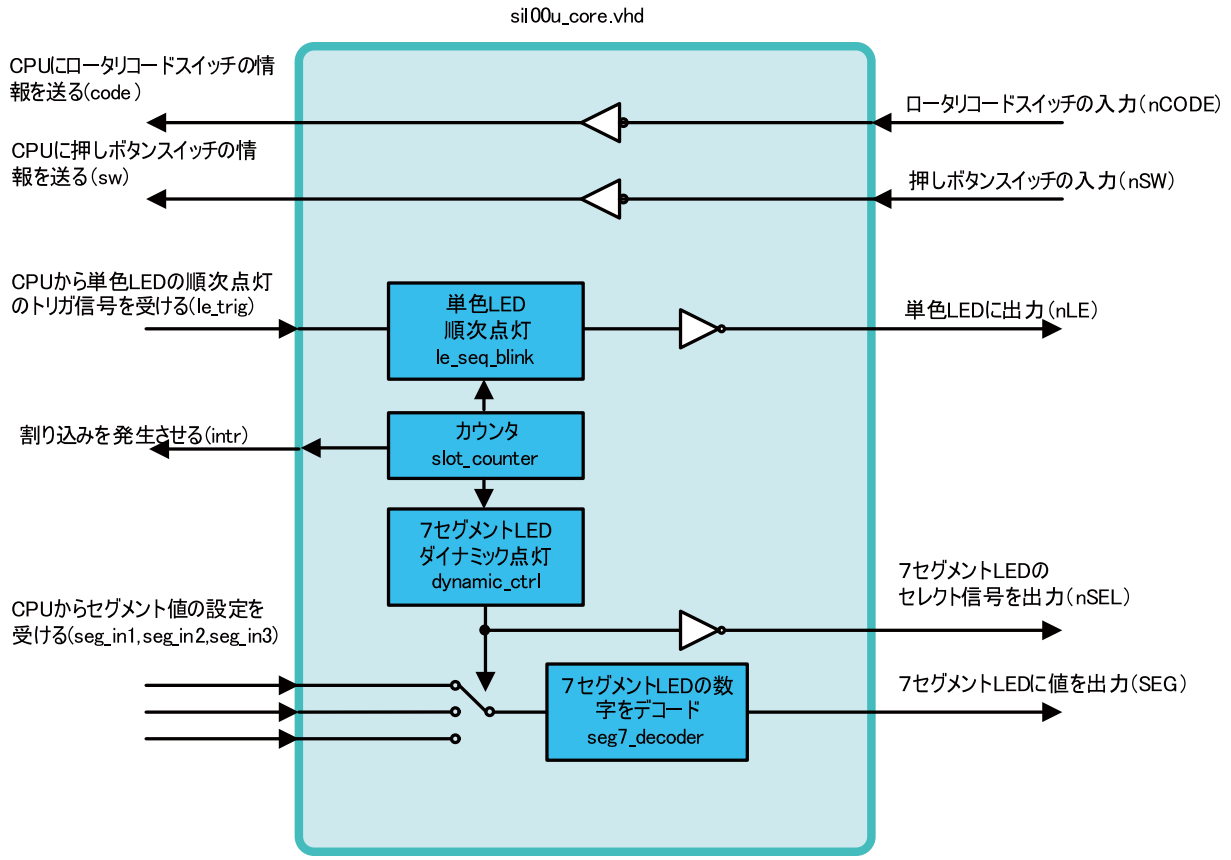


図 11.36. 自作 IP コア(ソフト版)の仕様

11.7.1. sil00u_core.vhd

SZ010、SZ030、SZ130 ではバスクロックが 51.6096MHz、SZ310 では 66.3552MHz、SZ410 では 87.5MHz になっています。カウンタのビット数を 4 ビット増やし 23 ビットにします。`sil00u_core.vhd` を上位階層として今まで作った回路、slot_counter、le_seq_blink、seg7_decoder、dynamic_ctrl 回路を呼び出します。また、押しボタンスイッチ、ロータリコードスイッチ、割り込みの信号の定義をします。

例 11.7. コア(sil00u_core.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sil00u_core is
  generic (
    C_CNT_WIDTH : integer := 23    --カウンタのビット幅
  );

  Port (
    SYS_CLK  : in  STD_LOGIC; --クロック信号          ❶
    SYS_RST  : in  STD_LOGIC; --リセット信号

    -- External
```

```
    SEG      : out STD_LOGIC_VECTOR(0 to 7); --7 セグ LED にダイナミック点灯で値を出力
    nSEL     : out STD_LOGIC_VECTOR(0 to 2); --7 セグ LED にセレクトをを出力
    nLE      : out STD_LOGIC_VECTOR(0 to 3); --単色 LED に順次点灯を出力
    nSW      : in  STD_LOGIC_VECTOR(0 to 2); --押しボタンスイッチを入力
    nCODE    : in  STD_LOGIC_VECTOR(0 to 3); --ロータリコードスイッチを入力

    -- Register Write
    seg_in1  : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    seg_in2  : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    seg_in3  : in  STD_LOGIC_VECTOR(0 to 3); --CPU からセグメント値の設定を受ける
    le_trig  : in  STD_LOGIC; --CPU から単色 LED の順次点灯のトリガ信号の設定を受ける

    -- Register Read
    sw       : out STD_LOGIC_VECTOR(0 to 2); --CPU に押しボタンスイッチの情報を送る
    code     : out STD_LOGIC_VECTOR(0 to 3); --CPU にロータリコードスイッチの情報を送る
    intr     : out STD_LOGIC  --カウンタの出力を割り込みコントローラに送る
  );
end sil00u_core;

architecture IMP of sil00u_core is
  signal count  : STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1);
  signal le     : STD_LOGIC_VECTOR(0 to 3);
  signal le_t   : STD_LOGIC_VECTOR(0 to 3);
  signal seg_data : STD_LOGIC_VECTOR(0 to 3);

  component slot_counter
    generic (
      C_CNT_WIDTH : integer := C_CNT_WIDTH
    );
    Port (
      SYS_CLK   : in  STD_LOGIC; --クロック信号
      SYS_RST   : in  STD_LOGIC; --リセット信号
      count     : out STD_LOGIC_VECTOR(0 to C_CNT_WIDTH-1) --カウンタ値
    );
  end component;

  component le_seq_blink                                  ❷
    Port (
      SYS_CLK   : in  STD_LOGIC;                 --クロック信号
      SYS_RST   : in  STD_LOGIC;                 --リセット信号
      le        : out STD_LOGIC_VECTOR(0 to 3); --単色 LED 出力信号
      le_timing : in  STD_LOGIC                  --タイミング信号
    );
  end component;

  component dynamic_ctrl
    Port (
      SYS_CLK   : in   STD_LOGIC;                 --クロック信号
      SYS_RST   : in   STD_LOGIC;                 --リセット信号
      nSEL      : out  STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号(負論理)
      seg7_timing : in STD_LOGIC;                 --7 セグタイミング信号
      seg_in1   : in   STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED1 の値
      seg_in2   : in   STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED2 の値
      seg_in3   : in   STD_LOGIC_VECTOR(0 to 3); --7 セグメント LED3 の値
      seg_data  : out  STD_LOGIC_VECTOR(0 to 3)  --4 ビットバイナリコード
```

```
    );
  end component;

  component seg7_decoder
    Port (
      SEG       : out  STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
      seg_data  : in   STD_LOGIC_VECTOR(0 to 3)  --4 ビットバイナリコード
    );
  end component;

begin
  slot_counter_0 : slot_counter                                    ❸
    Port map(
      SYS_CLK => SYS_CLK,
      SYS_RST => SYS_RST,
      count => count
    );

  le_seq_blink_0 : le_seq_blink
    Port map(
      SYS_CLK => SYS_CLK,
      SYS_RST => SYS_RST,
      le => le,
      le_timing => count(0)
    );

  dynamic_ctrl_0 : dynamic_ctrl
    Port map(
      SYS_CLK => SYS_CLK,
      SYS_RST => SYS_RST,
      nSEL => nSEL,
      seg7_timing => count(8),
      seg_in1 => seg_in1,
      seg_in2 => seg_in2,
      seg_in3 => seg_in3,
      seg_data => seg_data
    );

  seg7_decoder_0 : seg7_decoder
    Port map(
      SEG => SEG,
      seg_data => seg_data
    );
--トリガ信号が'1'の時順次点灯
  le_t <= le and "1111" when le_trig = '1' else "0000";
  nLE  <= not le_t;  --外部に出力
  sw   <= not nSW;   --正論理にして入力
  code <= not nCODE; --正論理にして入力
  intr <= count(4);  --カウンタの出力を割り込みコントローラに送る
end IMP;
```

❶　sil00u_core.vhd の入出力を定義

❸　4 つの回路のコンポーネント宣言

❷　4 つの回路のインスタンス

11.8. XPS インターフェースとコアを接続し、自作 IP コアを仕上げる

今まとめた回路(sil00u_core.vhd、slot_counter.vhd、

dynamic_ctrl.vhd、seg7_decoder.vhd、le_seq_blink.vhd)を

"C:\suzaku\sz***-yyyymmdd\pcores\xps_sil00_v1_00_a\hdl\vhdl"にコピーしてください。

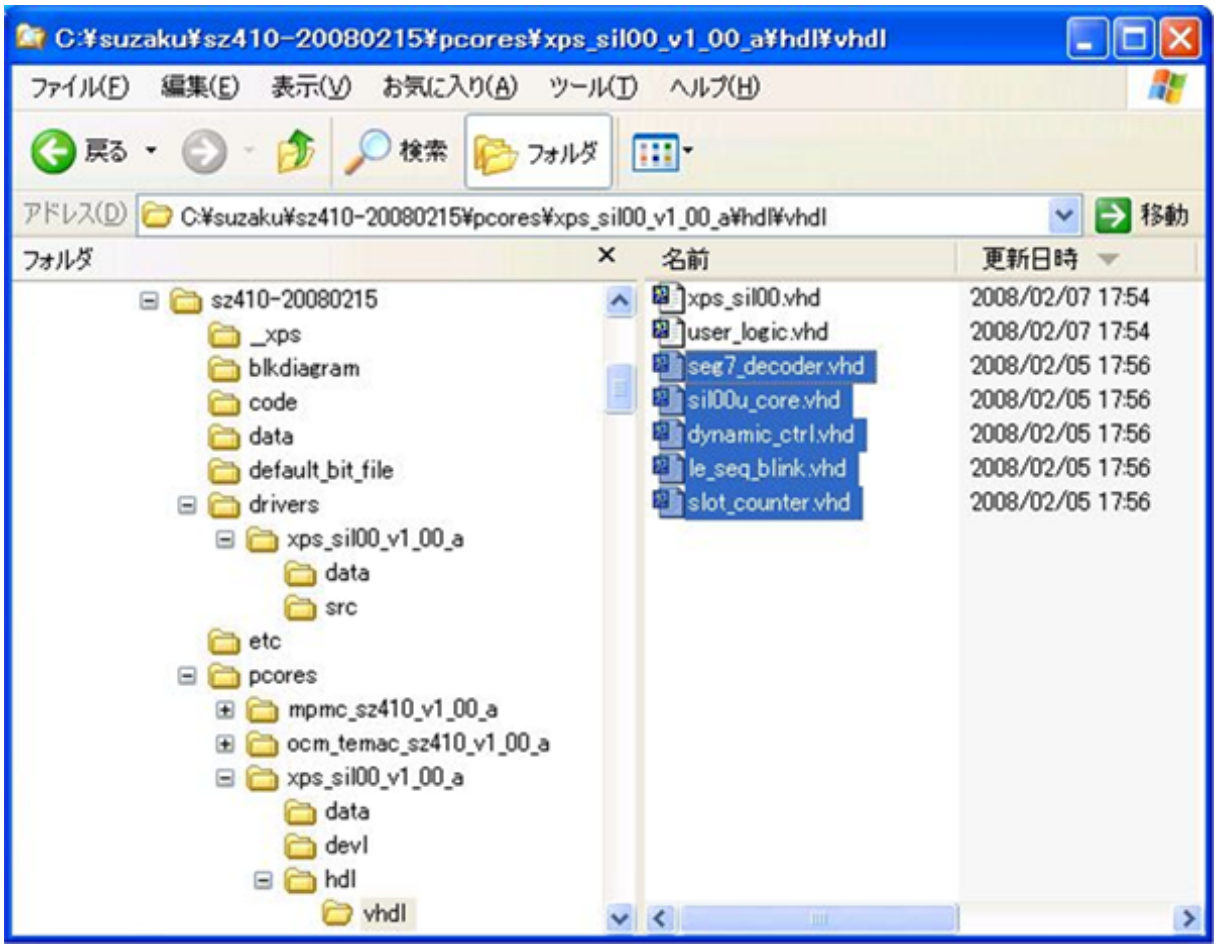


図 11.37. コアをコピー

11.8.1. user_logic.vhd

user_logic.vhd を開いてください。自動生成されたコードを編集していきます。user_logic を上位階層として、sil00u_core 回路を呼び出すソースコードを追加します。押しボタンスイッチ、ロータリコードスイッチは読み込むだけで書き込みは出来ません。この 2 つのために新たに、読み込み/書き込みレジスタではなく、読み込みレジスタを定義しています。ソースコードを追加するところには、大体--USER xxx added here とコメントが入っているので、目印にしてください。

例 11.8. xps_sil00(user_logic.vhd)

```
------------------------------------------------------------------------------
-- user_logic.vhd - entity/architecture pair
```

```
------------------------------------------------------------------------------
--中略
-- DO NOT EDIT BELOW THIS LINE --------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;

-- DO NOT EDIT ABOVE THIS LINE --------------------

 library xps_sil00_v1_00_a;      --ライブラリとして呼び出す
 use xps_sil00_v1_00_a.all;

--中略
------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_SLV_DWIDTH   : integer              := 32;
    C_NUM_REG      : integer              := 2;
    C_NUM_INTR     : integer              := 1
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------

    SEG         : out   STD_LOGIC_VECTOR(0 to 7);   -- 7 セグメント LED 出力
    nSEL        : out   STD_LOGIC_VECTOR(0 to 2);   -- セレクト出力
    nLE         : out   STD_LOGIC_VECTOR(0 to 3);   -- 単色 LED 出力
    nSW         : in    STD_LOGIC_VECTOR(0 to 2);   -- スイッチ入力
    nCODE       : in    STD_LOGIC_VECTOR(0 to 3);   -- ロータリ SW 入力

    -- ADD USER PORTS ABOVE THIS LINE ------------------
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk          : in    std_logic;
    Bus2IP_Clk          : in    std_logic;
    Bus2IP_Reset        : in    std_logic;
    Bus2IP_Data         : in    std_logic_vector(0 to C_SLV_DWIDTH-1);
    Bus2IP_BE           : in    std_logic_vector(0 to C_SLV_DWIDTH/8-1);
    Bus2IP_RdCE         : in    std_logic_vector(0 to C_NUM_REG-1);
    Bus2IP_WrCE         : in    std_logic_vector(0 to C_NUM_REG-1);
    IP2Bus_Data         : out   std_logic_vector(0 to C_SLV_DWIDTH-1);
    IP2Bus_RdAck        : out   std_logic;
    IP2Bus_WrAck        : out   std_logic;
    IP2Bus_Error        : out   std_logic;
    IP2Bus_IntrEvent    : out   std_logic_vector(0 to C_NUM_INTR-1)
```

```
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );

  attribute SIGIS                   : string;
  attribute SIGIS of Bus2IP_Clk     : signal is "CLK";
  attribute SIGIS of Bus2IP_Reset   : signal is "RST";

end entity user_logic;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
  signal slv_reg0               : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg1               : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_reg_write_sel      : std_logic_vector(0 to 1);
  signal slv_reg_read_sel       : std_logic_vector(0 to 1);
  signal slv_ip2bus_data        : std_logic_vector(0 to C_SLV_DWIDTH-1);
  signal slv_read_ack           : std_logic;
  signal slv_write_ack          : std_logic;

  ------------------------------------------
  -- Signals for user logic interrupt example
  ------------------------------------------
  signal intr_conuter    : std_logic_vector(0 to C_NUM_INTR-1); --割り込み用

begin

 -- 下位モジュール呼び出し sil00u_core インスタンス
 sil00u_core_0 : entity xps_sil00_v1_00_a.sil00u_core
  PORT MAP(
    SYS_CLK => Bus2IP_Clk,
    SYS_RST => Bus2IP_Reset,

    -- External
    SEG   => SEG,
    nSEL  => nSEL,
    nLE   => nLE,
    nSW   => nSW,
    nCODE => nCODE,

    -- R/W
    seg_in1 => slv_reg0(4 to 7),
    seg_in2 => slv_reg0(12 to 15),
    seg_in3 => slv_reg0(20 to 23),
    le_trig => slv_reg0(31),

    -- Register Read
    sw      => slv_reg1(5 to 7),
    code    => slv_reg1(12 to 15),
    -- intrrupt
    intr    => intr_counter(0)
  );
```

```
--中略
  slv_reg_write_select <= Bus2IP_WrCE(0 to 5);
  slv_reg_read_select  <= Bus2IP_RdCE(0 to 5);
  slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2)
                           or Bus2IP_WrCE(3) or Bus2IP_WrCE(4) or Bus2IP_WrCE(5);
  slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2)
                           or Bus2IP_RdCE(3) or Bus2IP_RdCE(4) or Bus2IP_RdCE(5);
  -- implement slave model register(s)
  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
  begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
        slv_reg0 <= (others => '0');
  --      slv_reg1 <= (others => '0');

      else
        case slv_reg_write_select is
          when "10" =>
            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                 slv_reg0(byte_index*8 to byte_index*8+7)
                   <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
              end if;
            end loop;
  --        when "01" =>
  --          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
  --            if ( Bus2IP_BE(byte_index) = '1' ) then
  --              slv_reg1(byte_index*8 to byte_index*8+7)
  --                <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
  --            end if;
  --          end loop;
          when others => null;
        end case;
      end if;
    end if;
  end process SLAVE_REG_WRITE_PROC;

  -- implement slave model software accessible register(s) read mux
  -- CPU からのレジスタ読み込み
  SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1 ) is

  begin
    case slv_reg_read_select is
      when "10" => slv_ip2bus_data <= slv_reg0;
      when "01" => slv_ip2bus_data <= slv_reg1;
      when others => slv_ip2bus_data <= (others => '0');
    end case;
  end process SLAVE_REG_READ_PROC;

  ------------------------------------------
  -- Example code to generate user logic interrupts
  --
  -- Note:
  -- The example code presented here is to show you one way of generating
  -- interrupts from the user logic. This code snippet infers a counter
  -- and generate the interrupts whenever the counter rollover (the counter
  -- will rollover ~21 sec @50Mhz).
  ------------------------------------------
```

```
   --  INTR_PROC : process( Bus2IP_Clk ) is
  --   constant COUNT_SIZE : integer := 30;
  --   constant ALL_ONES : std_logic_vector(0 to COUNT_SIZE-1):=(others => '1');
  --   variable counter  : std_logic_vector(0 to COUNT_SIZE-1);
  --  begin
  --
  --    if ( Bus2IP_Clk'event and Bus2IP_Clk = '1' ) then
  --      if ( Bus2IP_Reset = '1' ) then
  --        counter := (others => '0');
  --        intr_counter <= (others => '0');
  --      else
  --        counter := counter + 1;
  --        if ( counter = ALL_ONES ) then
  --          intr_counter <= (others => '1');
  --        else
  --          intr_counter <= (others => '0');
  --        end if;
  --      end if;
  --    end if;
  --
  --  end process INTR_PROC;

  IP2Bus_IntrEvent <= intr_counter;  --割り込みを counter からの出力に接続

  ------------------------------------------
  -- Example code to drive IP to Bus signals
  ------------------------------------------
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else (others => '0');
  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';

end IMP;
```

11.8.2. xps_sil00.vhd

`xps_sil00.vhd` を開いてください。自動生成されたコードを編集していきます。xps_sil00 を上位階層として、user_logic 回路を呼び出すコードを追加します。

例 11.9. xps_sil00(xps_sil00.vhd)

```
-----------------------------------------------------------------------------
-- xps_sil00.vhd - entity/architecture pair
-----------------------------------------------------------------------------
-- 中略
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
use proc_common_v2_00_a.ipif_pkg.all;

library interrupt_control_v2_00_a;
use interrupt_control_v2_00_a.interrupt_control;
```

```
library plbv46_slave_single_v1_00_a;
use plbv46_slave_single_v1_00_a.plbv46_slave_single;

library xps_sil00_v1_00_a;
use xps_sil00_v1_00_a.user_logic;

------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
entity xps_sil00 is
  generic
  (
    --中略
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------

    SEG         : out   STD_LOGIC_VECTOR(0 to 7); --7 セグメント LED への出力信号
    nSEL        : out   STD_LOGIC_VECTOR(0 to 2); --7 セグメント LED セレクト信号
    nLE         : out   STD_LOGIC_VECTOR(0 to 3); --単色 LED への出力信号
    nSW         : in    STD_LOGIC_VECTOR(0 to 2); --押しボタンスイッチからの入力信号
    nCODE       : in    STD_LOGIC_VECTOR(0 to 3); --ロータリコードスイッチからの入力信号

    -- ADD USER PORTS ABOVE THIS LINE ------------------
    -- DO NOT EDIT BELOW THIS LINE ----------------------
    -- Bus protocol ports, do not add to or delete
    --中略
    -- DO NOT EDIT ABOVE THIS LINE ----------------------
  );
  attribute SIGIS : string;
  attribute SIGIS of SPLB_Clk       : signal is "Clk";
  attribute SIGIS of SPLB_Rst       : signal is "Rst";
  attribute SIGIS of IP2INTC_Irpt  : signal is "INTR_LEVEL_HIGH";

end entity xps_sil00;
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------
architecture IMP of xps_sil00 is
--中略
begin
  ------------------------------------------
  -- instantiate plbv46_slave_single
  ------------------------------------------
  PLBV46_SLAVE_SINGLE_I : entity plbv46_slave_single_v1_00_a.plbv46_slave_single
    generic map
    (
--中略
    )
    port map
    (
--中略
    );

  ------------------------------------------
```

```
  -- instantiate interrupt_control
  ------------------------------------------
  INTERRUPT_CONTROL_I : entity interrupt_control_v2_00_a.interrupt_control
    generic map
    (
--中略
    )
    port map
    (
--中略
    );

  ------------------------------------------
  -- instantiate the User Logic
  ------------------------------------------
  USER_LOGIC_I : entity xps_sil00_v1_00_a.user_logic
    generic map
    (
--中略
    )
    port map
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------
      SEG   => SEG,
      nSEL  => nSEL,
      nLE   => nLE,
      nSW   => nSW,
      nCODE => nCODE,
      -- MAP USER PORTS ABOVE THIS LINE ------------------
--中略
    );
--中略
end IMP;
```

11.8.3. xps_sil00_v2_1_0.mpd

"C:\suzaku\sz***- yyyymmdd \pcores\xps_sil00_v1_00_a\data"を開いてください。

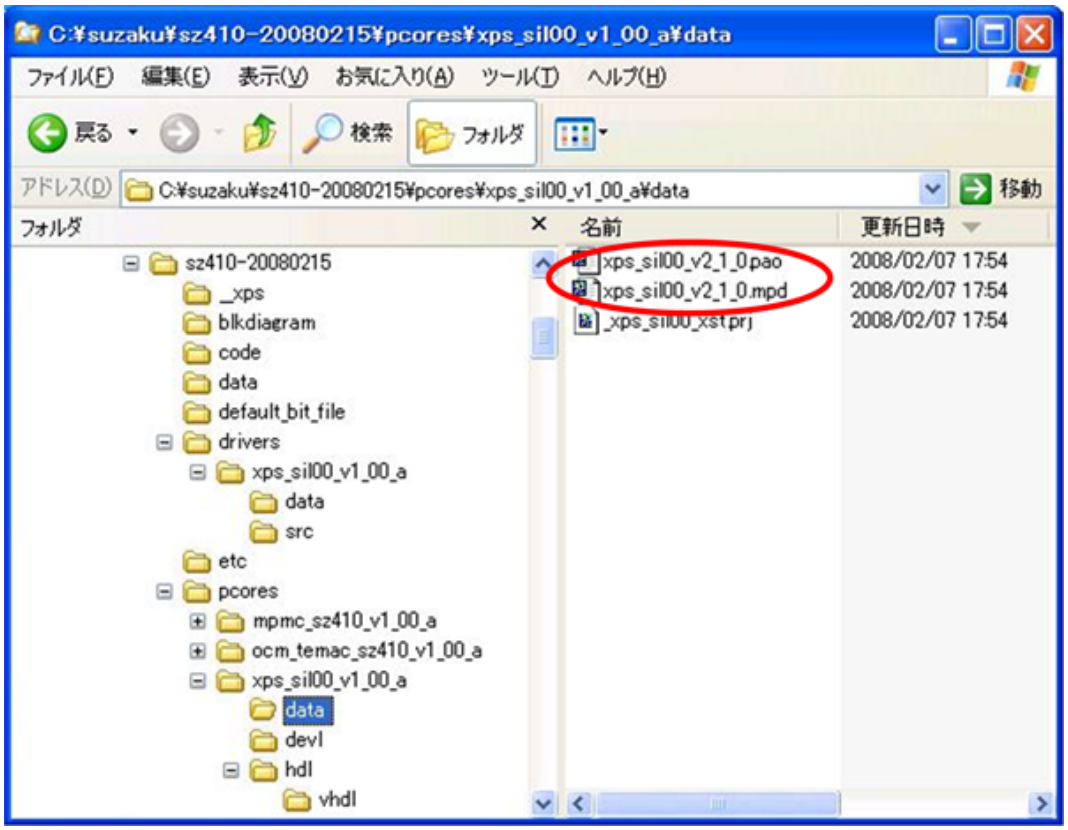


図 11.38. フォルダ構成

xps_sil00_v2_1_0.mpd を編集します。mpd(Microprocessor Peripheral Definition)ファイルでは信号の入出力の方向やビット幅等を定義できます。7 セグメント LED、7 セグメント LED セレクト、単色 LED、押しボタンスイッチ、ロータリコードスイッチの信号を外部と接続できるように定義します。

以下の文を一番下に追加してください。

例 11.10. xps_sil00_v2_1_0.mpd

```
PORT SEG  = "", DIR = O, VEC = [0:7]
PORT nSEL  = "", DIR = O, VEC = [0:2]
PORT nLE   = "", DIR = O, VEC = [0:3]
PORT nSW   = "", DIR = I, VEC = [0:2]
PORT nCODE = "", DIR = I, VEC = [0:3]
```

11.8.4. xps_sil00_v2_1_0.pao

xps_sil00_v2_1_0.pao を編集します。pao(Peripheral Analyze Order)ファイルはペリフェラルのコンパイル(構成およびシミュレーション用)に必要な HDL ファイルと、その解析順を指定します。自分で書いたソースコードを追加します。

以下の文を一番下に追加してください。

例 11.11. xps_sil00_v2_1_0.pao

```
lib xps_sil00_v1_00_a sil00u_core vhdl
lib xps_sil00_v1_00_a slot_counter vhdl
lib xps_sil00_v1_00_a le_seq_blink vhdl
lib xps_sil00_v1_00_a seg7_decoder vhdl
lib xps_sil00_v1_00_a dynamic_ctrl vhdl
```

11.8.5. xps_sil00.c

"C:\suzaku\sz***-yyyymmdd\drivers\xps_sil00_v1_00_a\src\xps_sil00.c"を編集します。SUZAKU では stdio を使用しておらず、xil_printf() は生成されません。XPS_SIL00_Intr_DefaultHandler 関数の中に記述されている xil_printf()の行をコメントアウトしてください。

例 11.12. xps_sil00.c

```
void XPS_SIL00_Intr_DefaultHandler(void * baseaddr_p)
{
  Xuint32 baseaddr;
  Xuint32 IntrStatus;
  Xuint32 IpStatus;

  baseaddr = (Xuint32) baseaddr_p;

  /*
   * Get status from Device Interrupt Status Register.
   */
  IntrStatus = XPS_SIL00_mReadReg(baseaddr, XPS_SIL00_INTR_DISR_OFFSET);

//  xil_printf("Device Interrupt! DISR value : 0x%08x \n\r", IntrStatus);

  /*
   * Verify the source of the interrupt is the user logic and clear the interrupt
   * source by toggle write baca to the IP ISR register.
   */
  if ( (IntrStatus & INTR_IPIR_MASK) == INTR_IPIR_MASK )
  {
//    xil_printf("User logic interrupt! \n\r");
    IpStatus = XPS_SIL00_mReadReg(baseaddr, XPS_SIL00_INTR_ISR_OFFSET);
    XPS_SIL00_mWriteReg(baseaddr, XPS_SIL00_INTR_ISR_OFFSET, IpStatus);
  }
}
```

これで自作 IP コアの完成です。

11.9. 自作 IP コアの追加

EDK で、自作 IP コアを SUZAKU のデフォルトのプロジェクトに追加します。

まず、SUZAKU のデフォルトに追加した時のブロック図を見てください。

自作 IP コアは、OPB | PLB と接続され、外部(押しボタンスイッチや単色 LED)とつながります。

11.9.1. SZ010、SZ030 の場合

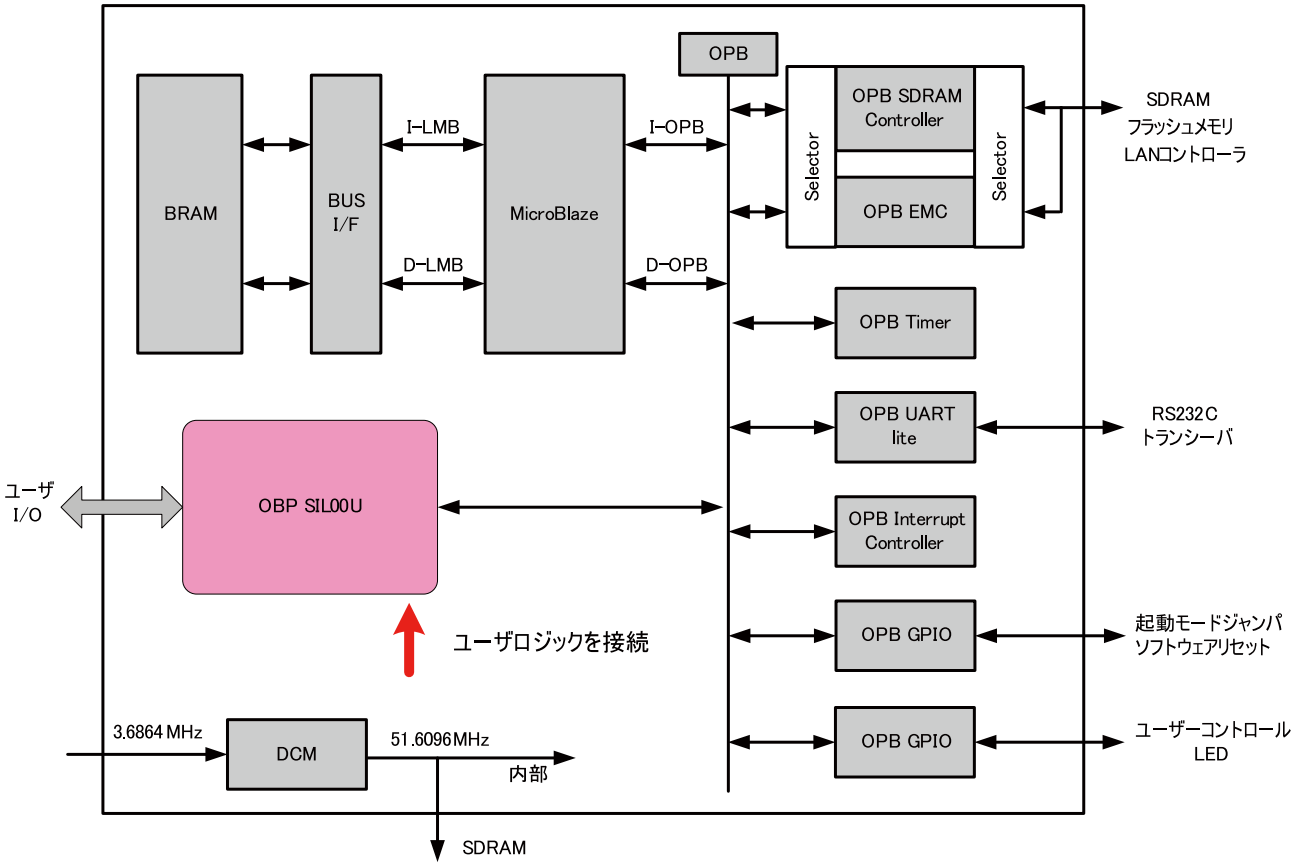


図 11.39. SZ010、SZ030 のデフォルトに自作 IP コアを追加

11.9.2. SZ130 の場合

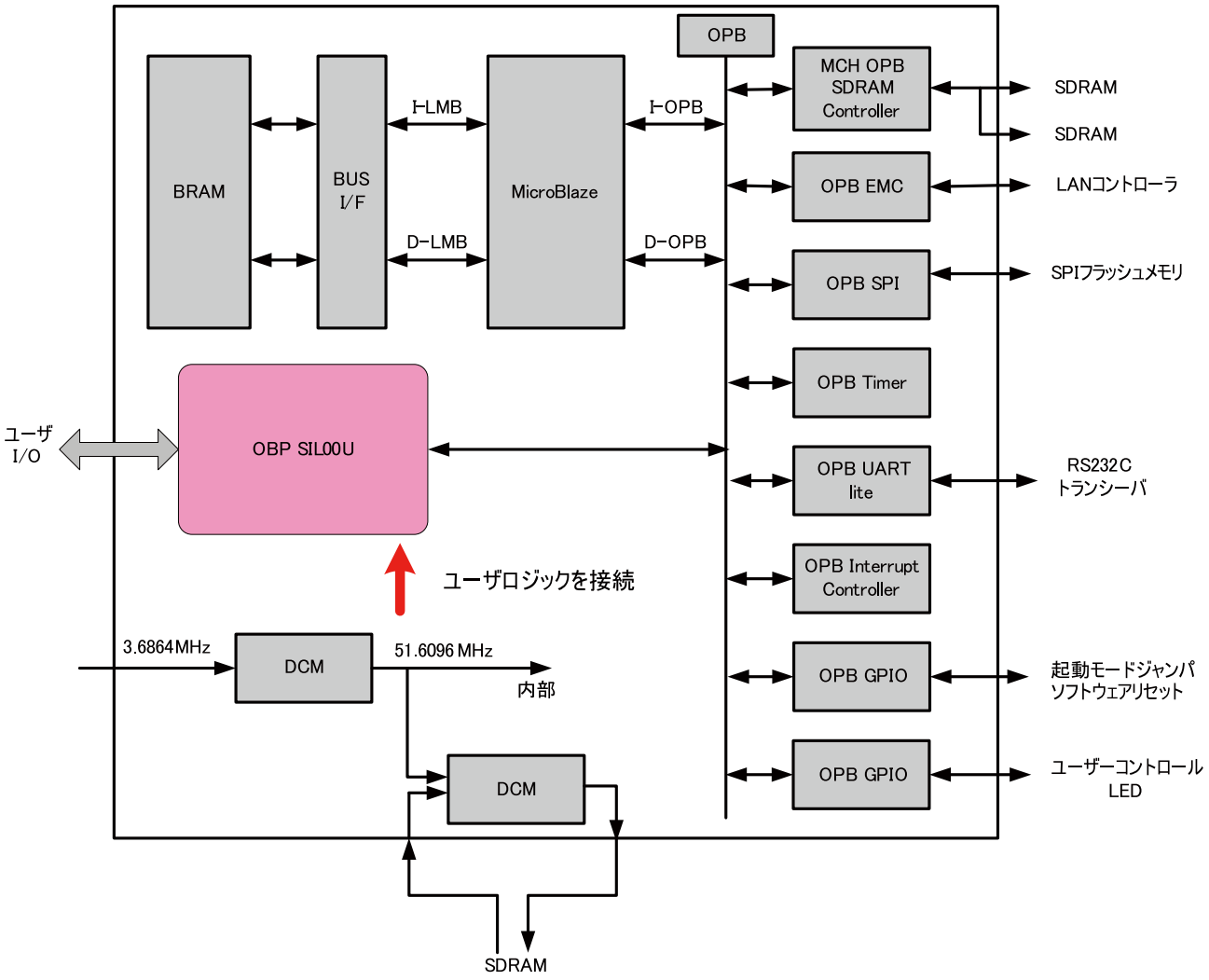


図 11.40. SZ130 のデフォルトに自作 IP コアを追加

11.9.3. SZ310 の場合

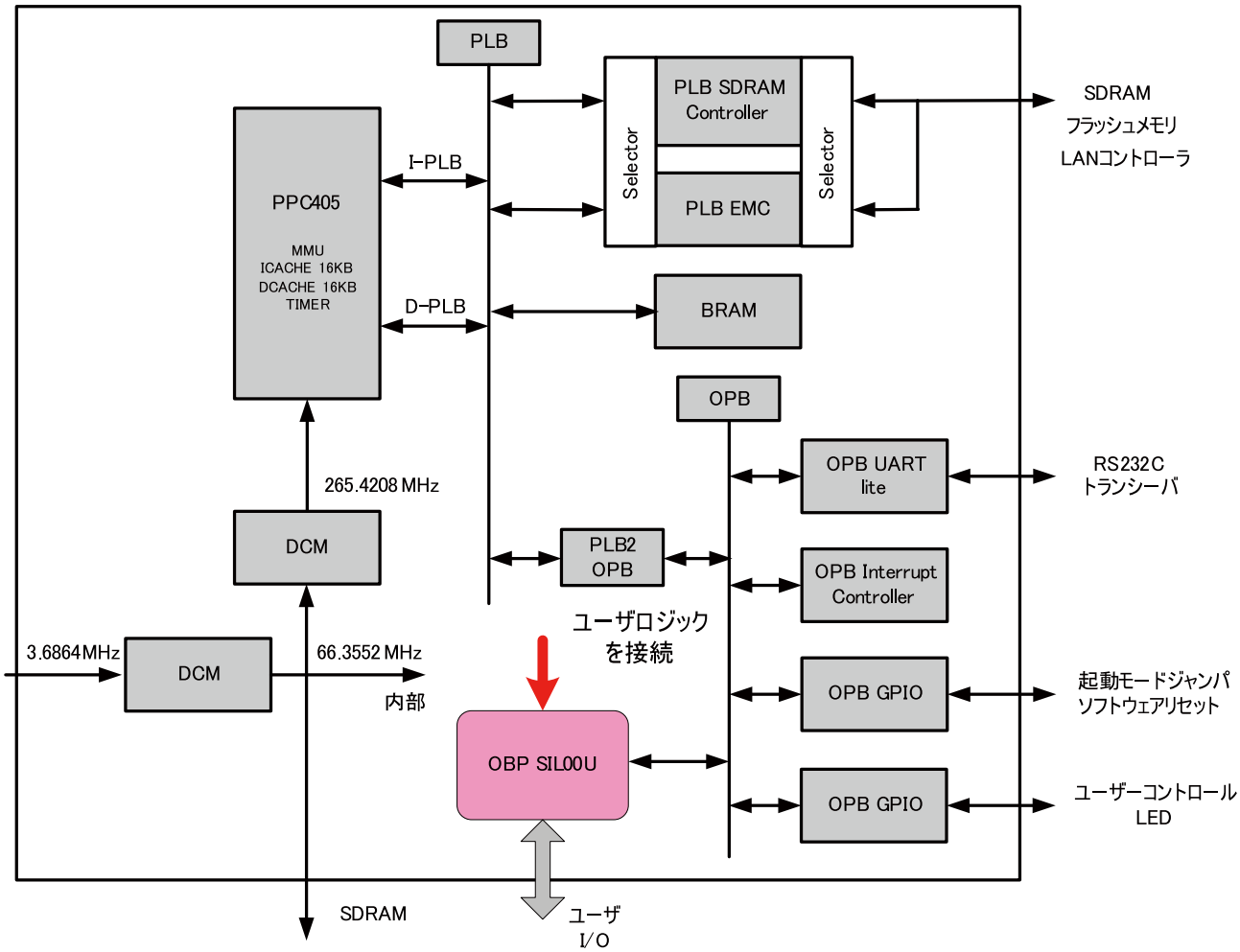


図 11.41. SZ310 のデフォルトに自作 IP コアを追加

11.9.4. SZ410 の場合

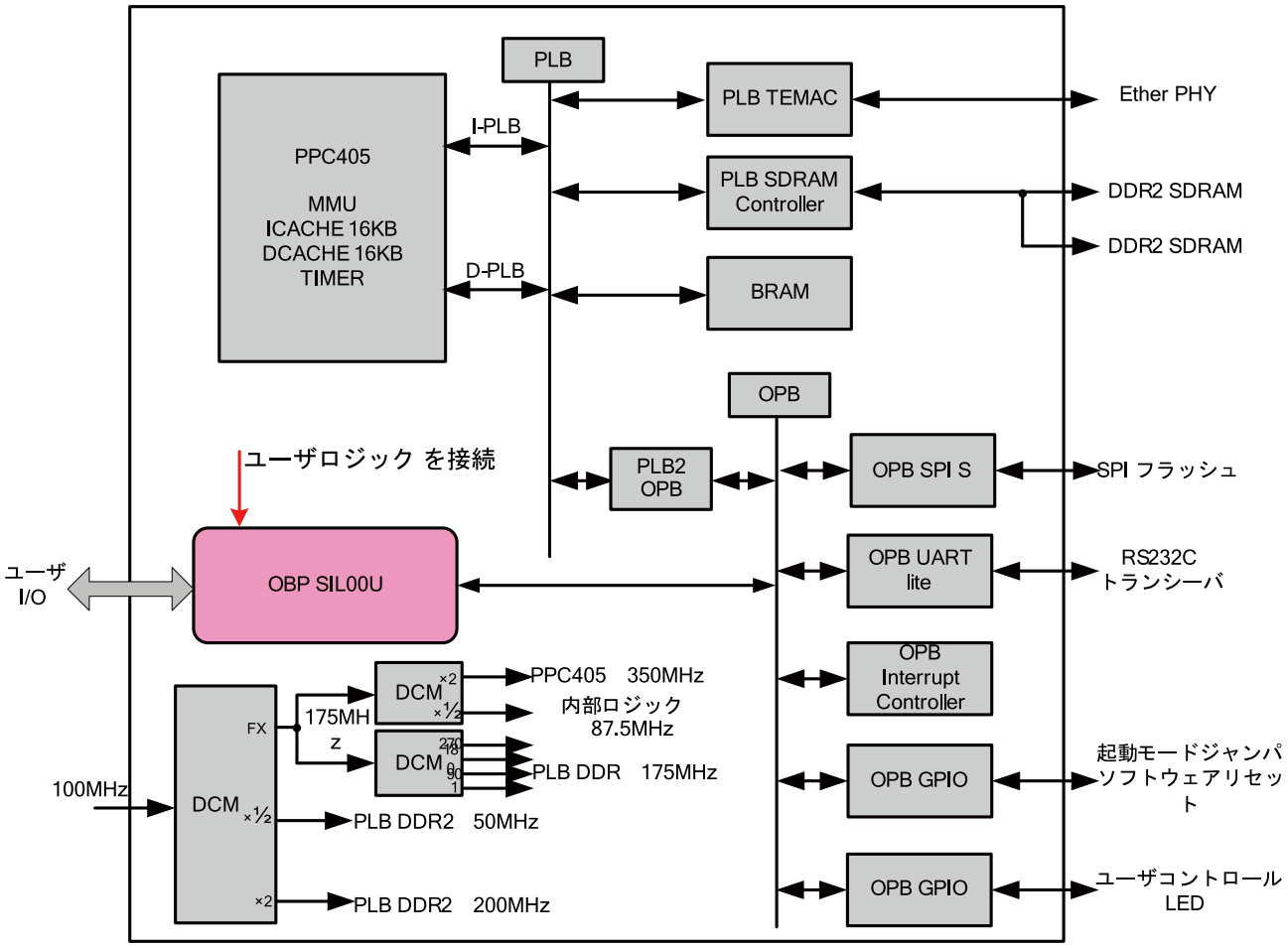


図 11.42. SZ410 のデフォルトに自作 IP コアを追加

11.9.5. ハードウェア設定

自作コアが、EDK に読み込まれているか確認します。

EDK に自作コアが作成されたことを伝えるため、[Project]→[Rescan User Repositories]をクリックしてください。

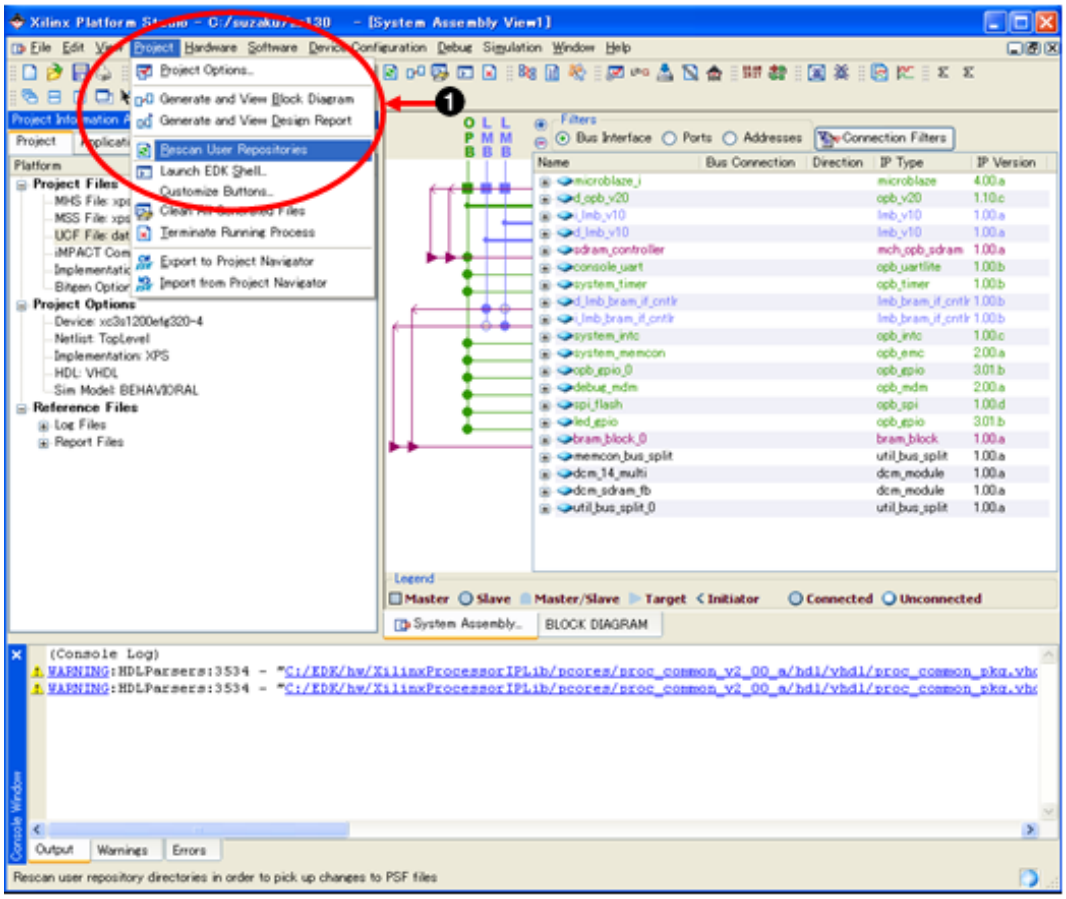


図 11.43. 自作 IP コア読み込み

❶　[Project]→[Rescan User Repositories]をクリック

IP Catalog の Project Local pcores に自分のコア opb_sil00 | xps_sil00 が追加されます。

もしうまく追加されなかった場合は、一回 Xilinx Platform Studio を閉じて、再起動し、`xps_proj.xmp` を開き直してください。

11.9.5.1. IP コアの追加

opb_sil00 | xps_sil00 を右クリックして出てくるメニューの Add IP を選択してください。

opb_sil00 | xps_sil00 が追加されます。

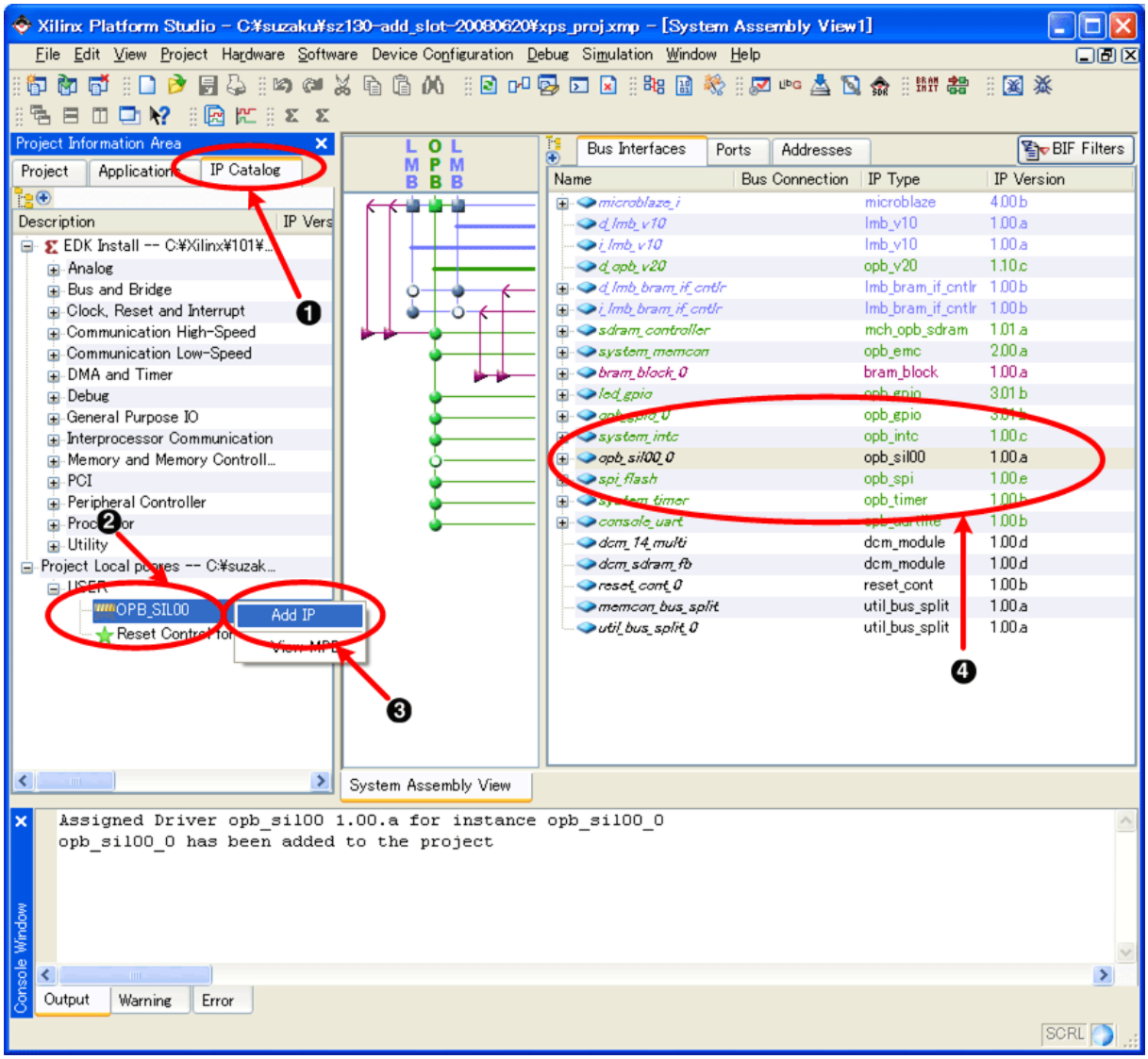


図 11.44. 自作 IP コア追加

❶ IP Catalog のタブをクリック

❷ Project Repository に自作コアが追加されている

❸ 右クリックしてメニューを出し、Add IP を選択

❹ IP コアが追加される

11.9.5.2. バスに接続

Bus Interface を選択し、opb_sil00_0 | xps_sil00_0 の横の丸をクリックしてください。○→●

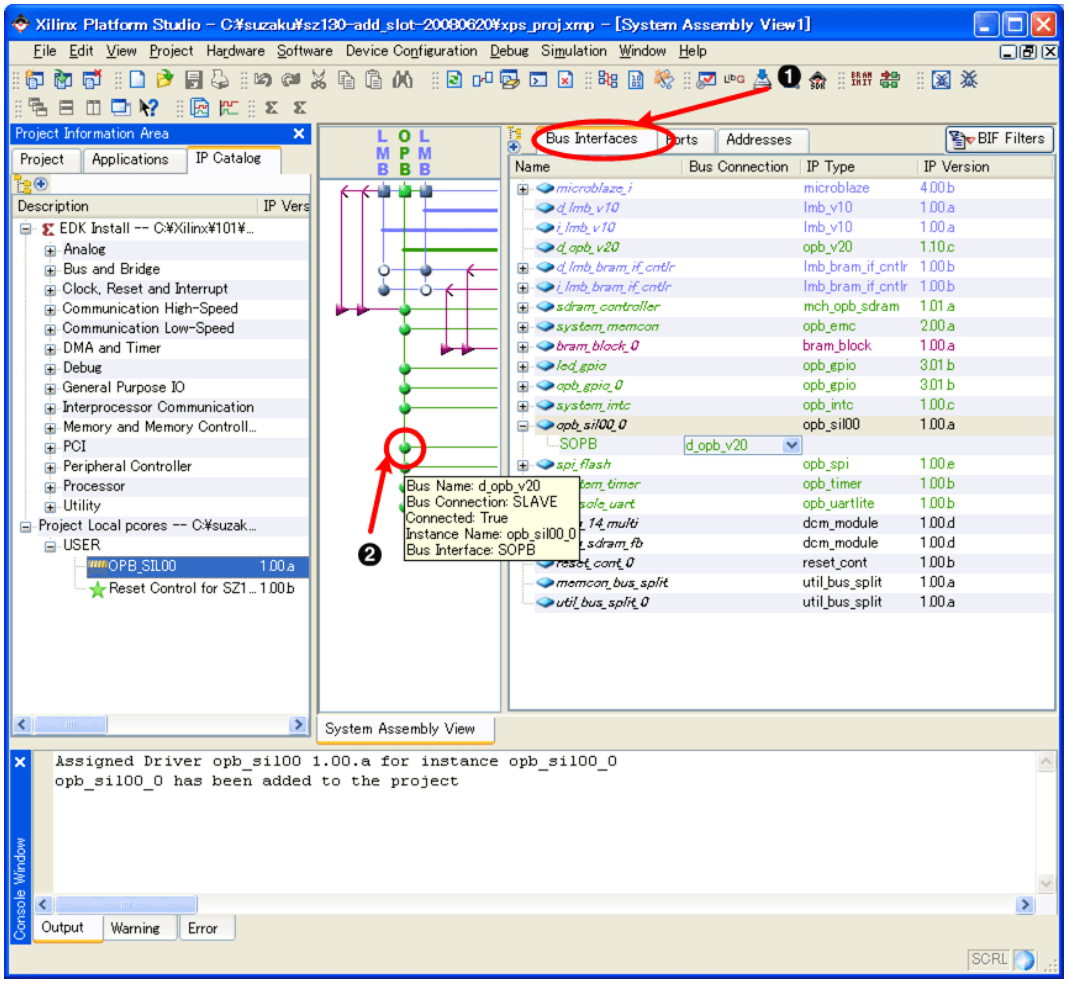


図 11.45. OPB バスに接続(SZ010,SZ030,SZ130,SZ310)

❶ Bus Interface を選択

❷ 白丸をクリック○→●

SZ410 の場合は PLB バス(plb_peripheral)につなぎ、xps_sil00_0 から sil_cntlr にインスタンス名を変更してください。

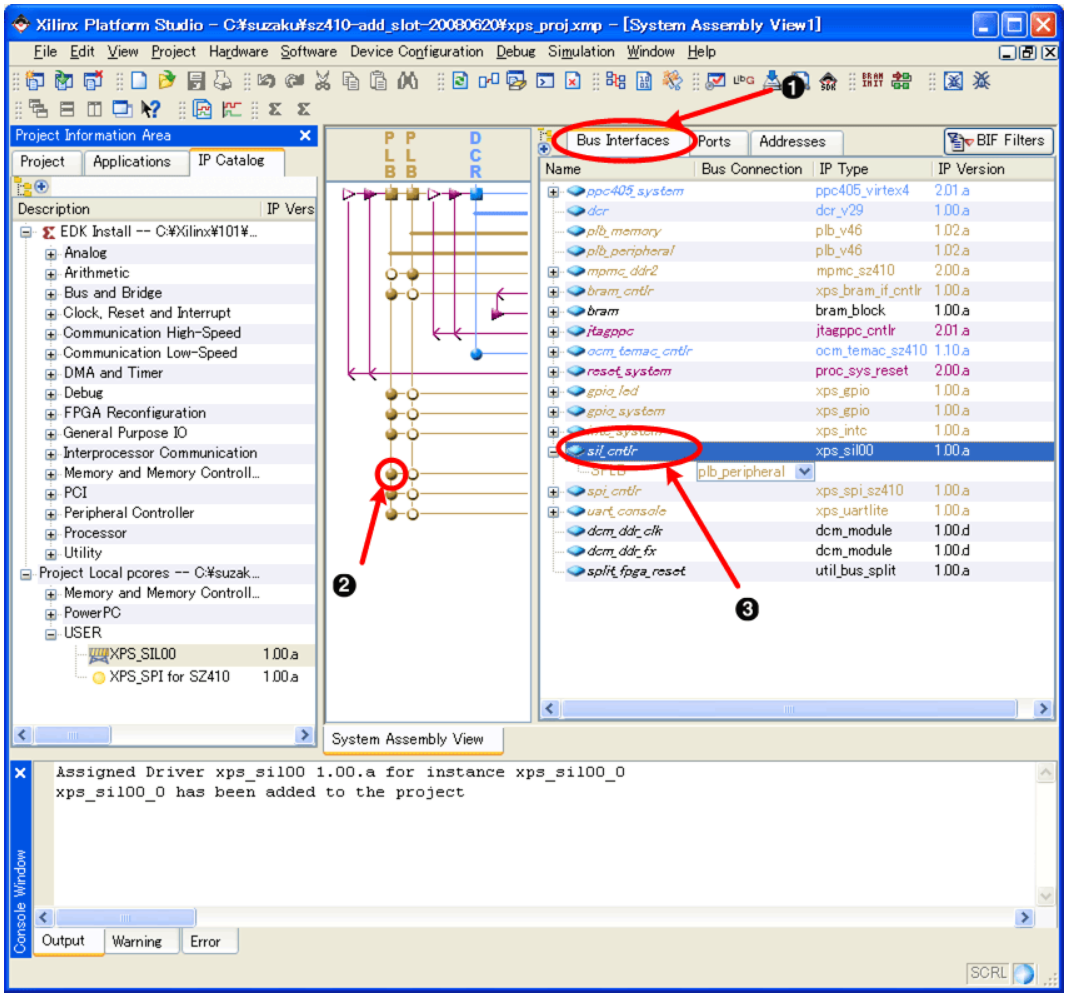


図 11.46. PLB バスに接続(SZ410)

❶ Bus Interface を選択

❷ 白丸をクリック○ →●

❸ sil_cntlr にインスタンス名変更

11.9.5.3. IP コアの設定

opb_sil00_0 | sil_cntlr を右クリックし、メニューの Configure IP...を選択してください。

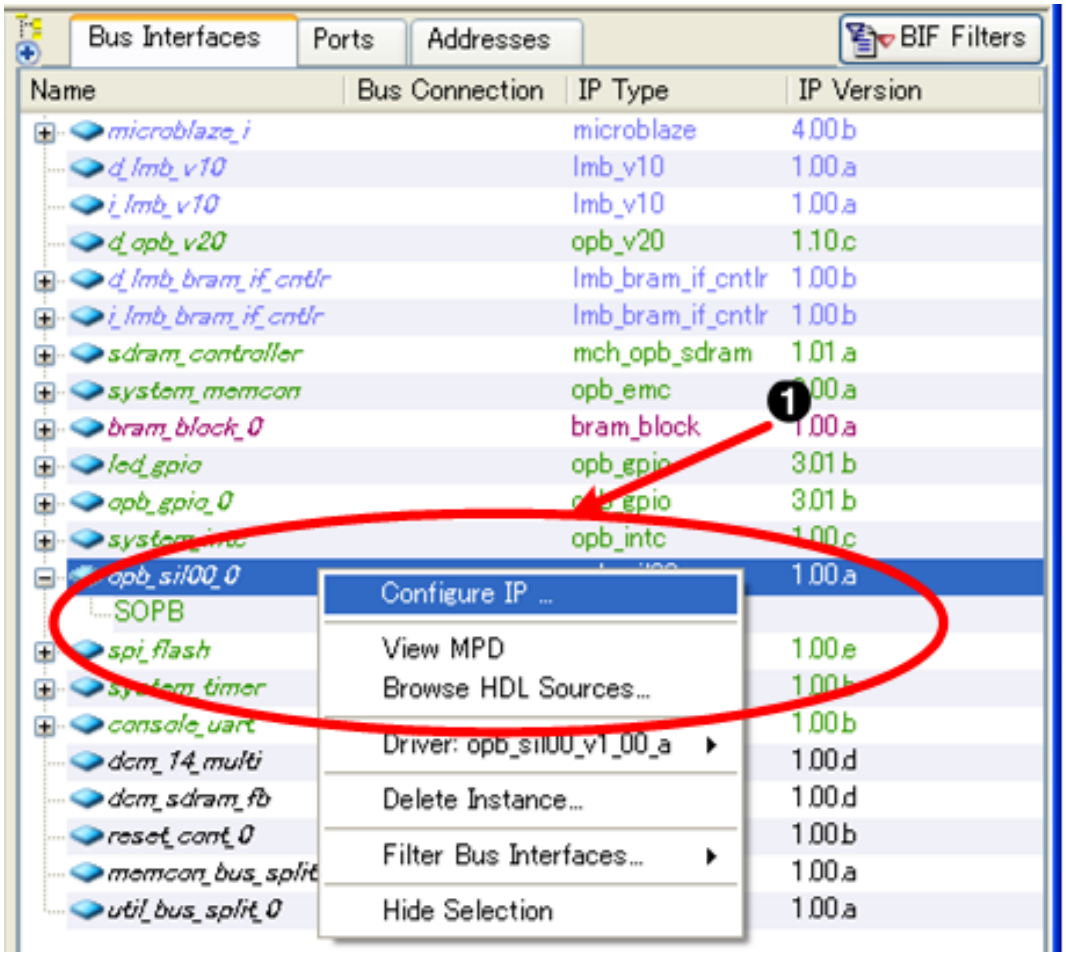


図 11.47. アドレス設定画面呼び出し

❶　右クリックしてメニューを出し Configure IP を選択

メモリアドレスを設定します。[C_BASEADDR]、[C_HIGHADDR]にメモリアドレスを入力し、[OK]をクリックして下さい。メモリアドレスは SUZAKU のメモリマップで Free と書いてあるところに割り当てます。(「1.4. メモリマップ」参照)

表 11.1. 自作 IP のメモリアドレス

	SZ010、SZ030 SZ130	SZ310、SZ410
Base Address	0xFFFFD000	0xF0FFD000
High Address	0xFFFFD1FF	0xF0FFD1FF

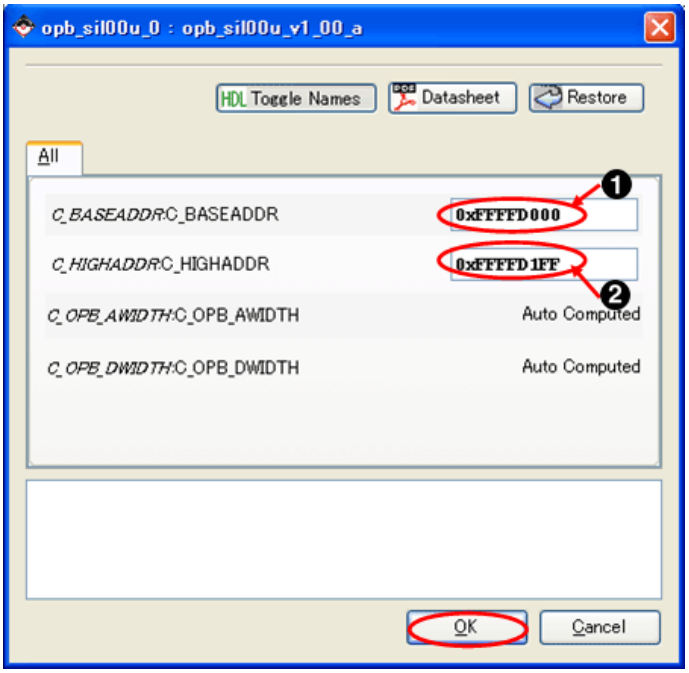


図 11.48. アドレス設定

❶　BASEADDR を入力

❷　HIGHADDR を入力

11.9.5.4. メモリマップ確認

Addresses を選択し、opb_sil00_0 | sil_cntlr の BaseAddress と High Address と Size に間違いがないか、確認してください。

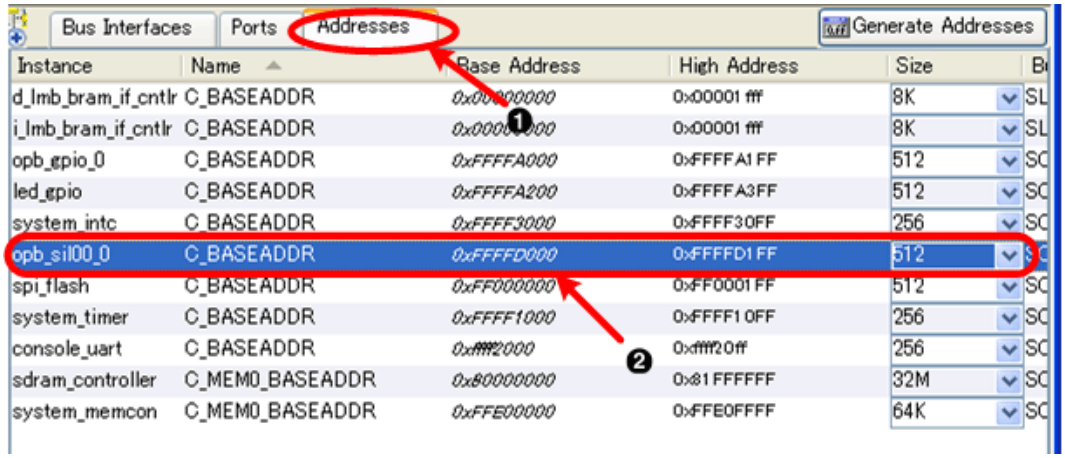


図 11.49. メモリマップ確認

❶　Addresses を選択

❷　メモリマップを確認

11.9.5.5. 信号の定義

Ports を選択し、opb_sil00_0 の をクリックして開いてください。mpd ファイルで自分で設定した信号線＋自動生成された割り込み線が出来上がっていると思います。

SEG の Net の部分をクリックし、Net 名を SEG と入力し、欄外をクリックし確定させてください。

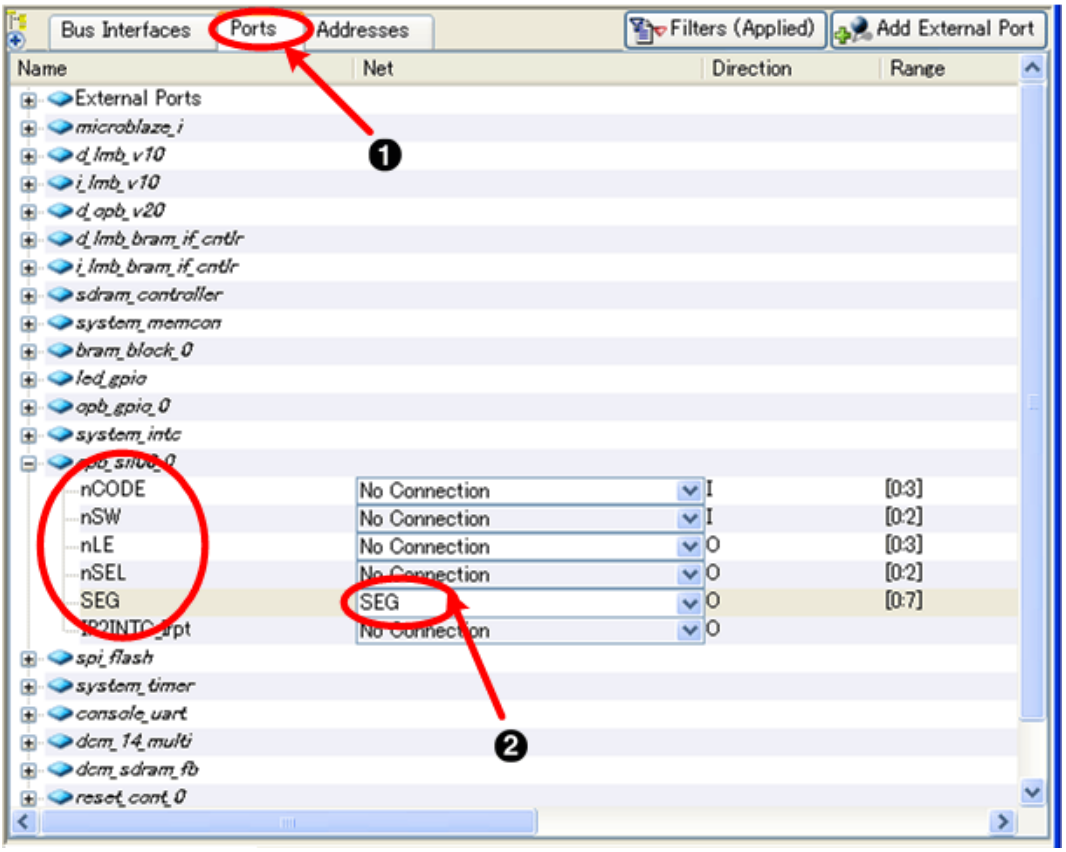


図 11.50. NET 名入力

❶　Ports を選択

❷　クリックし、SEG と入力し、確定させる

もう一度 SEG の Net をクリックし、今度は をクリックし、[Make External]を選択し、欄外をクリックして確定させてください。

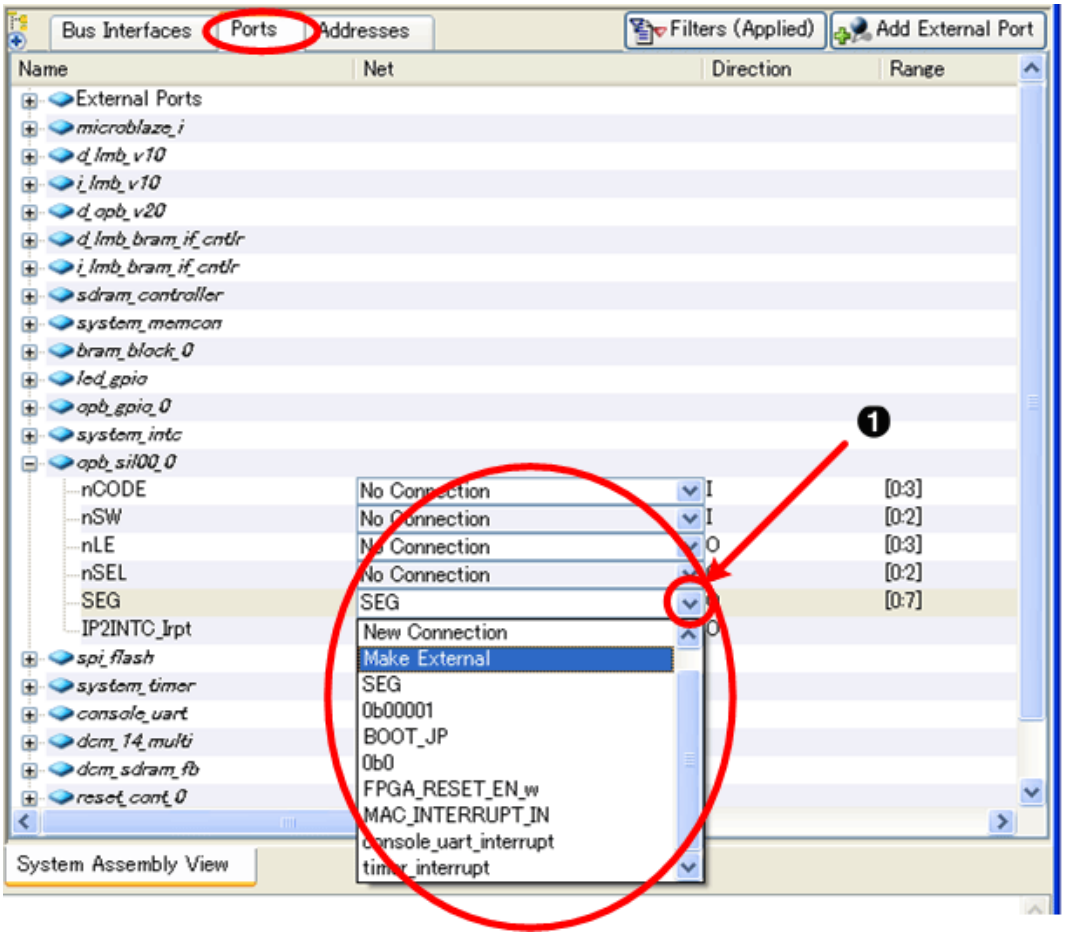


図 11.51. 外部信号にする

❶　Make External を選択し、確定させる

External Ports の⊞をクリックして開いてください。Name:SEG_pin という信号が出来上がっています。変えなくても良いのですが、後々の読みやすさのため信号名を変更します。SEG_pin をクリックし、名前を SEG に変更してください。これで、外部出力信号 SEG が定義されます。

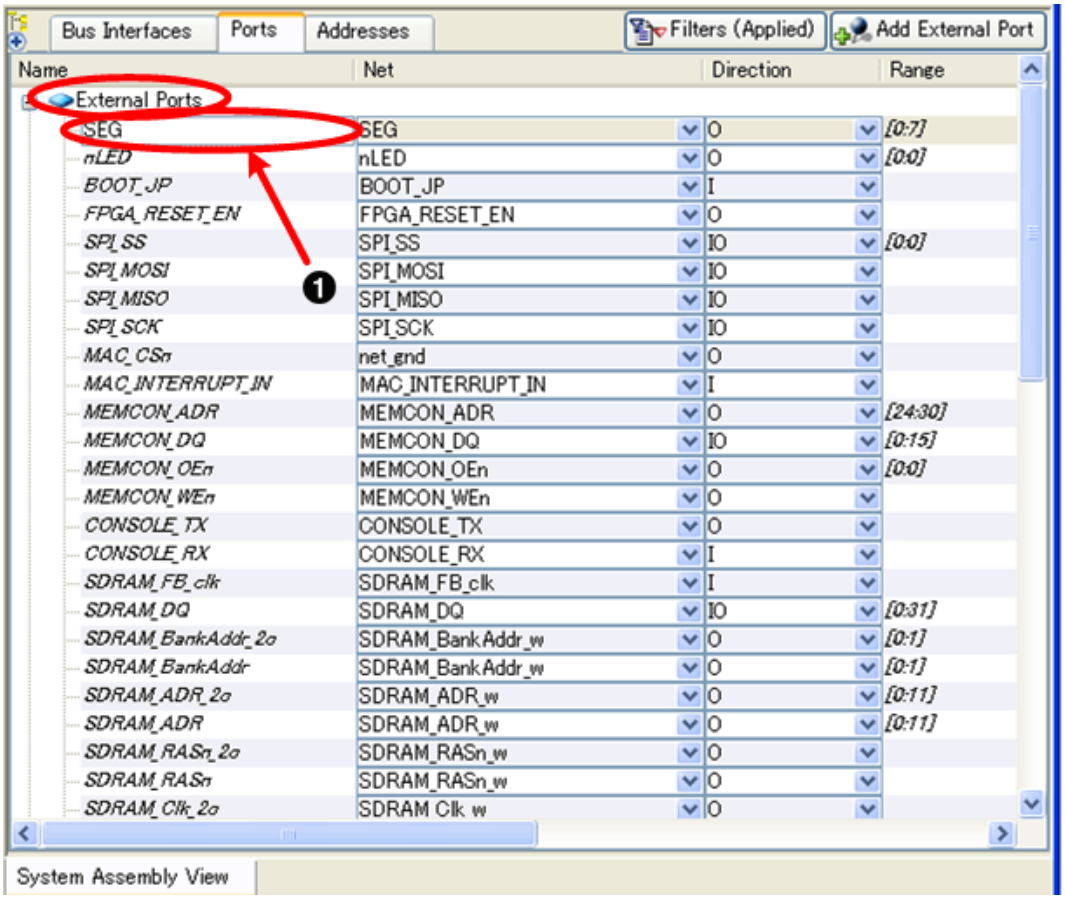


図 11.52. 出力信号定義

❶　クリックし、SEG_pin→SEG に変更

nSEL、nLE、nSW、nCODE も SEG と同様の操作を行ってください。

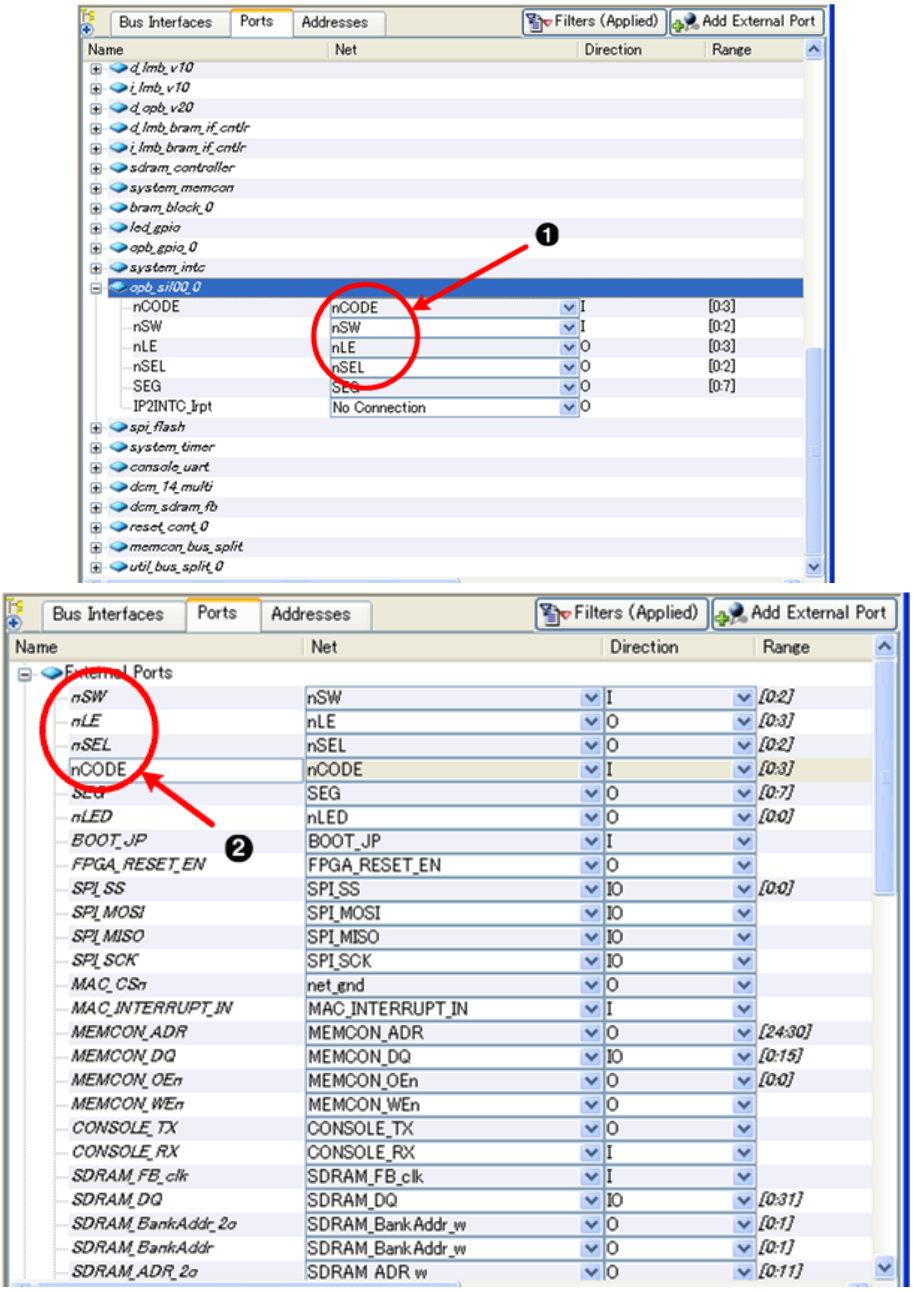


図 11.53. 残り出力信号定義

❶　名前を入力し、確定

↓

Make External を選択し、確定

❷　名前を変更

MicroBlaze および PowerPC は外部から 1 つの割り込みしか受けることができません。SUZAKU では、複数の割り込み要因があるため、割り込みコントローラを用いて、割り込み処理を行います。割り込みが発生すると、割り込みハンドラが呼び出され、優先順位が高いものから割り込みが処理されます。

IP2INTC_Irpt の Net に sil_intr と入力してください。、system_inc(SZ410 の場合は intc_system)の Intr の Net をクリックしてください。割り込み信号の設定ウィンドが立ち上がります。sil_intr を追加してください。これで自作コアの割り込みが割り込みコントローラに接続されます。右側に書いたものほど割り込みの優先順位が高くなります。

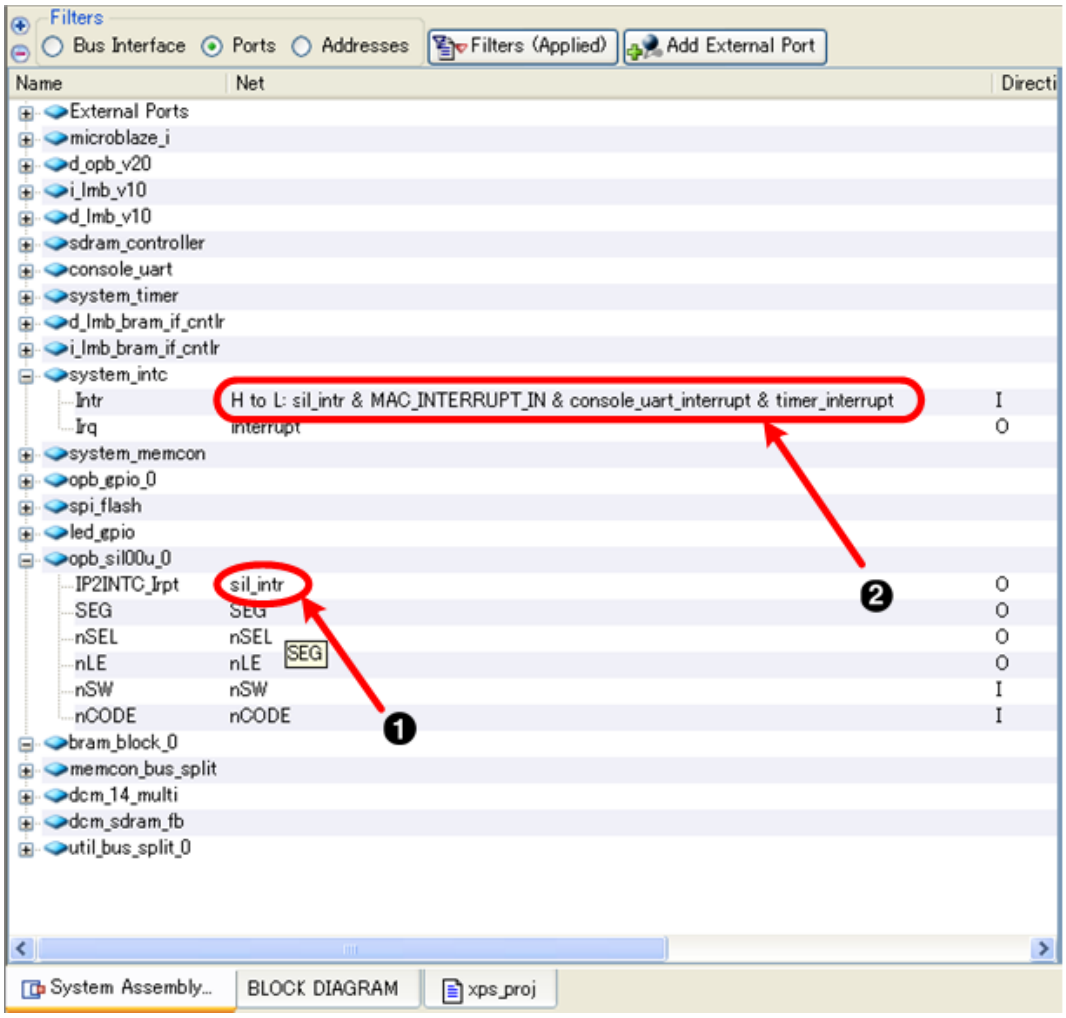


図 11.54. 割り込みコントローラ

❶　sil_intr と入力し、確定させる

❷　sil_intr & を追記する

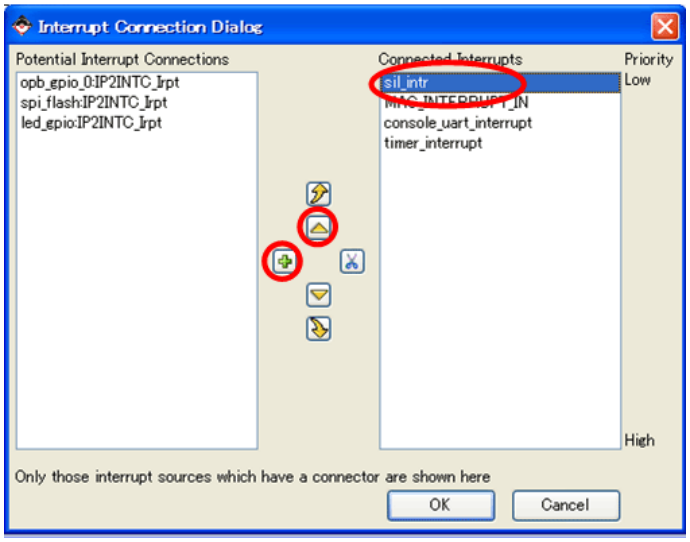


図 11.55. 割り込み信号の順番の設定

11.9.5.6. MPMC 編集

SZ410

SZ410 の場合 MPMC の編集をします。mpmc_ddr2 の上で右クリックをし、[Configure IP ...]を選択してください。

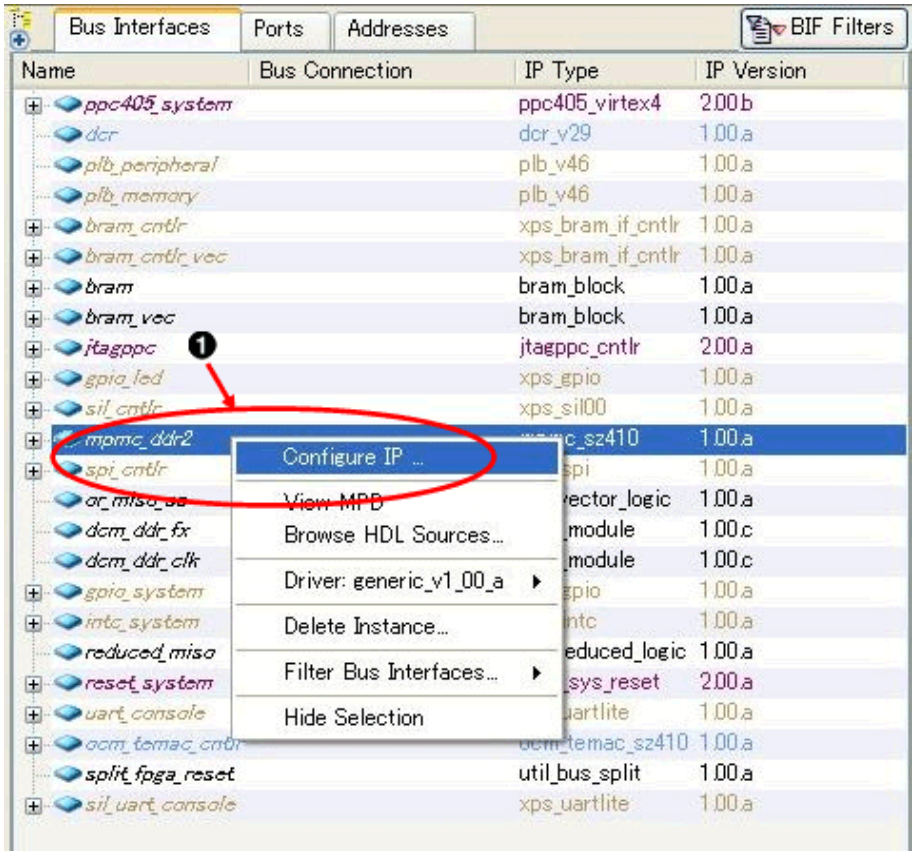


図 11.56. MPMC 編集

❶　右クリックしてメニューを出し、[Configure_IP...]を選択

以下の画面が立ち上がります。[Advanced]→[Data Path Configuration]をクリックし、Read FIFO Config と Write FIFO Config を[SRL]に変更してください。後で BBoot の編集をした時に BRAM の容量が足りなくなってしまうため、シフトレジスタを使用します。変更できたら[OK]をクリックして下さい。

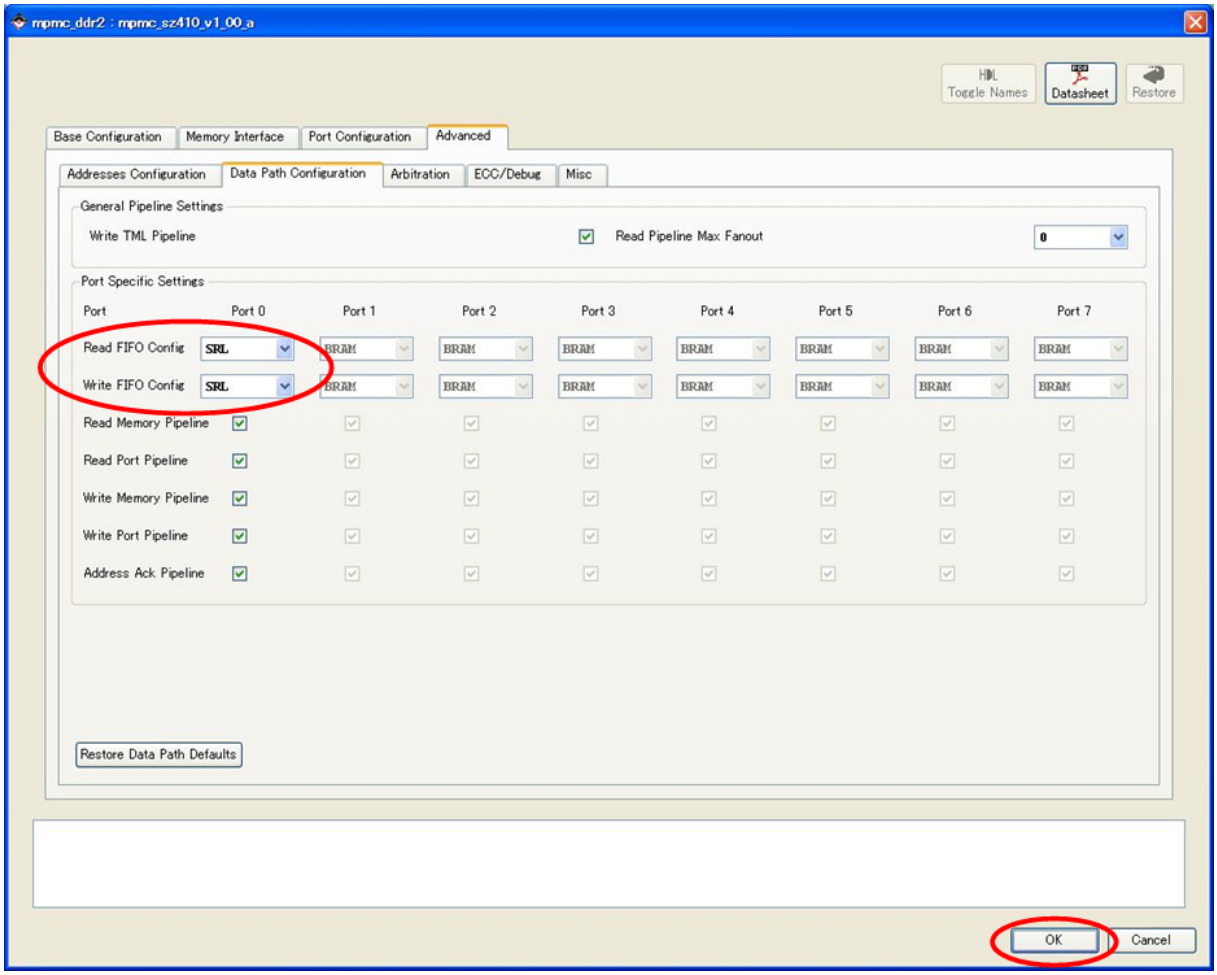


図 11.57. FIFO Config 変更

11.9.5.7. 割り込み

SZ310 SZ410

SZ310,SZ410 はハードプロセッサの PowerPC なので、設定が SZ010,SZ030,SZ130 と違います。PowerPC では、EVPR(例外ベクタプレフィックスレジスタ)に割り込みベクタをセットする必要があります。EVPR は上位 16bit のみが使用され、下位 16bit は無視されるレジスタです。デフォルトのプロジェクトでは、boot セクションが 0xFFFFFFFC にあり、BRAM を 0xFFFFC000 - 0xFFFFFFFF に割り当てています。EVPR に割り込みベクタをセットするためには、0xFFFF0000 まで拡張しないといけないのですが、これだと BRAM の容量が 64KByte になってしまいます。SZ310 に採用している Virtex-II Pro は 48KByte の BRAM しか内蔵していないため容量が足りません。SZ410 では足りないことはありませんが、もったいないので xFFFFC000 - 0xFFFFFFFF の他に、もうひとつ BRAM を用意し、0xFFFF0000 - 0xFFFF3FFF にセットし、ここに割り込みベクタを割り当てるようにします。また、text セクションが 15KByte になっていますので、rodata、data、bss なども、0xFFFF0000 - 0xFFFF3FFF 側に割り当てて、必要容量を分散することにします。

SZ310 の場合 plb_bram_if_cntlr を追加して、PLB に接続し、bram_block を追加して、

plb_bram_if_cntlr に接続してください。

SZ410 の場合 xps_bram_cntlr を追加してインスタンス名を bram_cntlr_vec に変更して、

PLB(plb_peripheral)に接続し、bram_block を追加してインスタンス名を bram_vec に変更し、bram_cntlr_vec に接続してください。

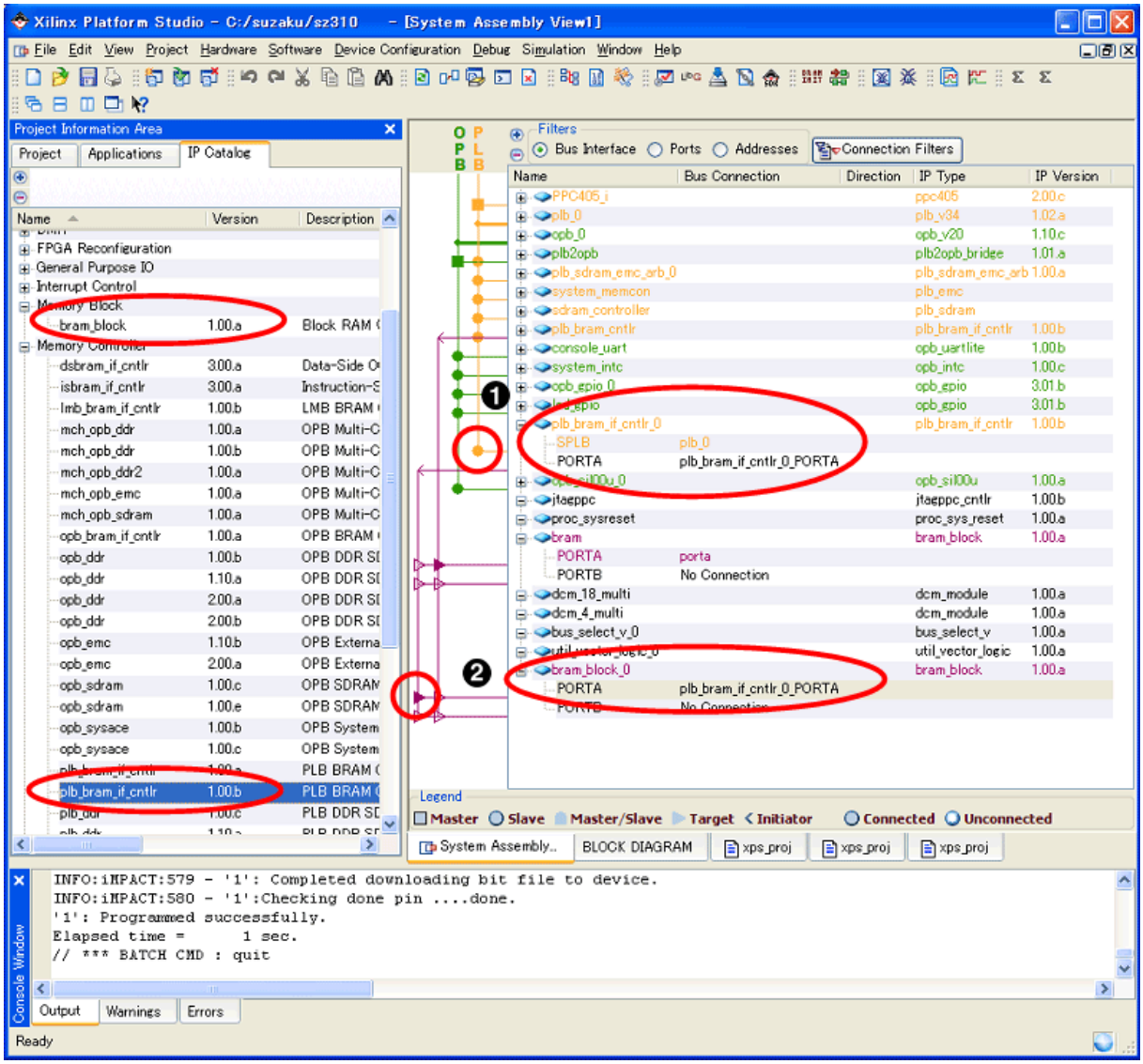


図 11.58. 割り込み設定(SZ310)

❶ plb_bram_if_cntlr を追加し、PLB に接続する

❷ bram_block を追加し、plb_bram_if_contlr_0 に接続

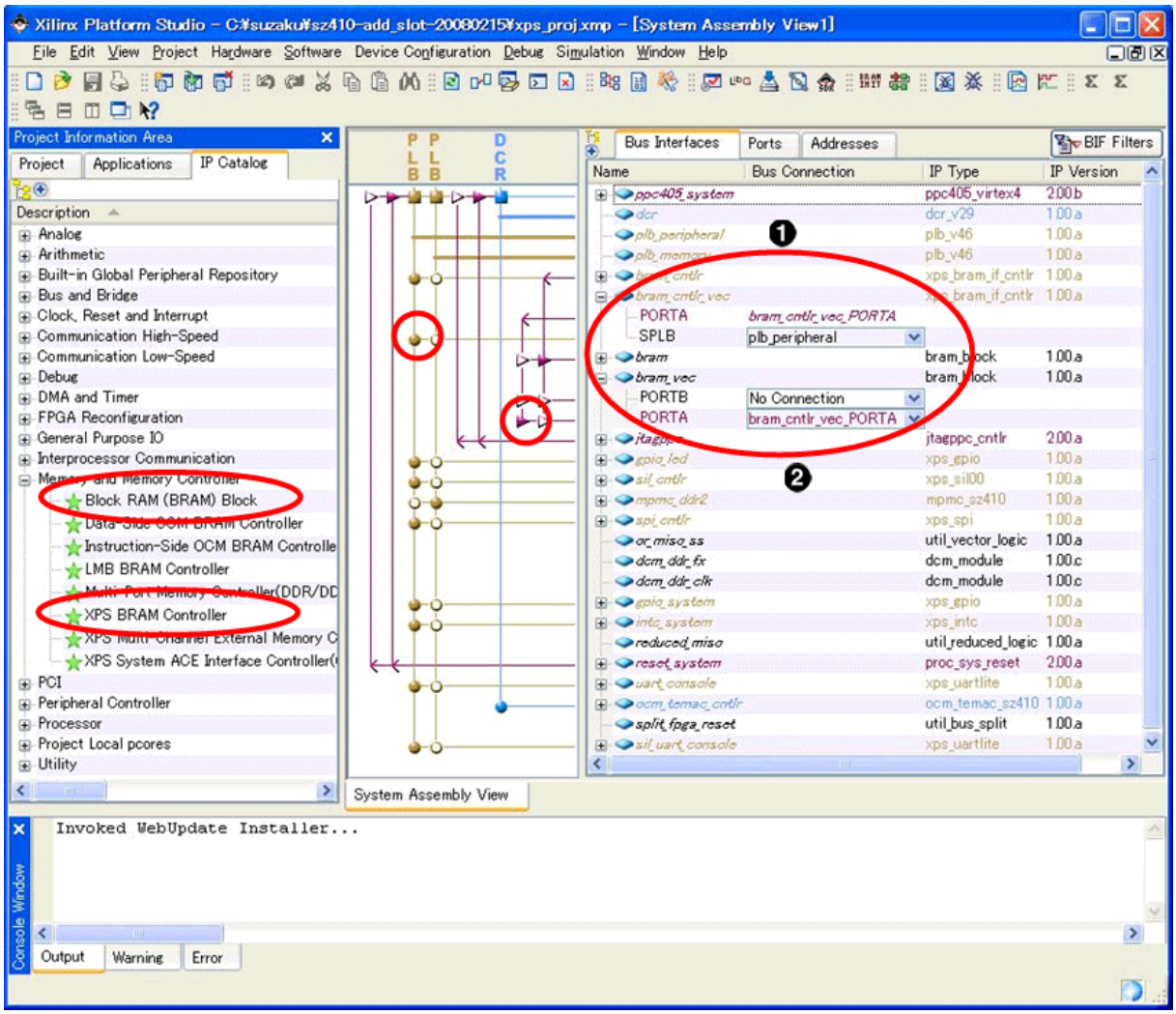


図 11.59. 割り込み設定(SZ410)

❶　xps_bram_if_cntlr を追加し PLB に接続インスタンス名を bram_cntlr_vec に変更

❷　bram_block を追加し、インスタンス名を bram_vec に変更 bram_cntlr_vec に接続する

plb_bram_if_cntlr_0/bram_cntrl_vec の設定画面を開き、Base Address に[0xFFFF0000]、High Address に[0xFFFF3FFF]と入力し、[OK]をクリックして下さい。

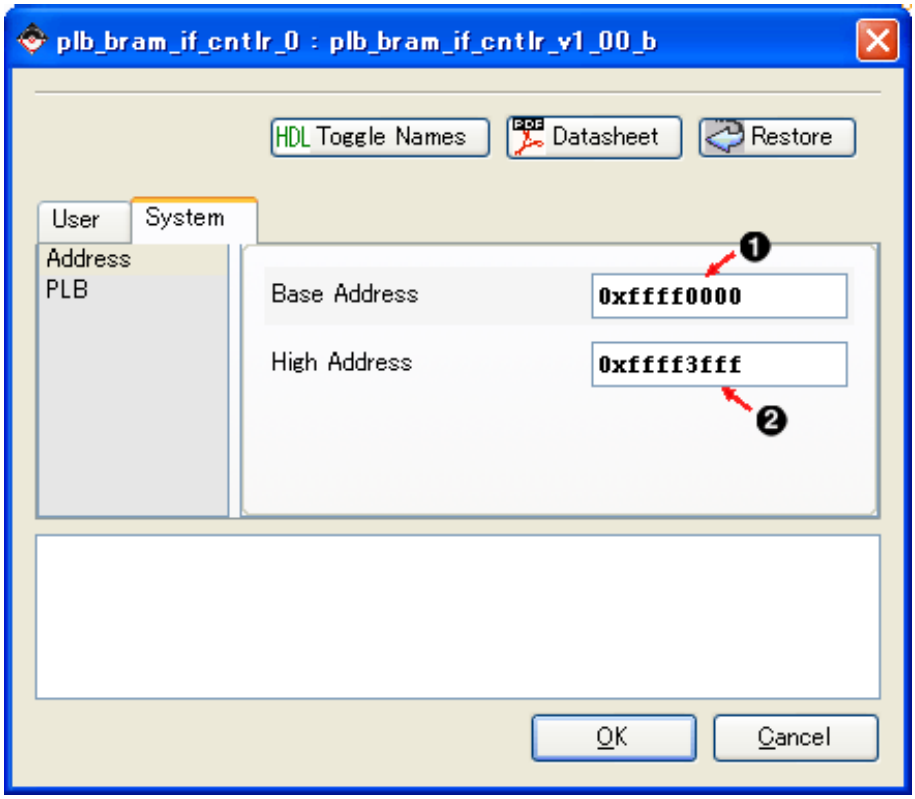


図 11.60. 割り込み設定(アドレス変更)

❶　0xFFFF0000 と入力

❷　0xFFFF3FFF と入力

リンカースクリプトを作ります。[Software]→[Generate Linker Script...]を選択してください。

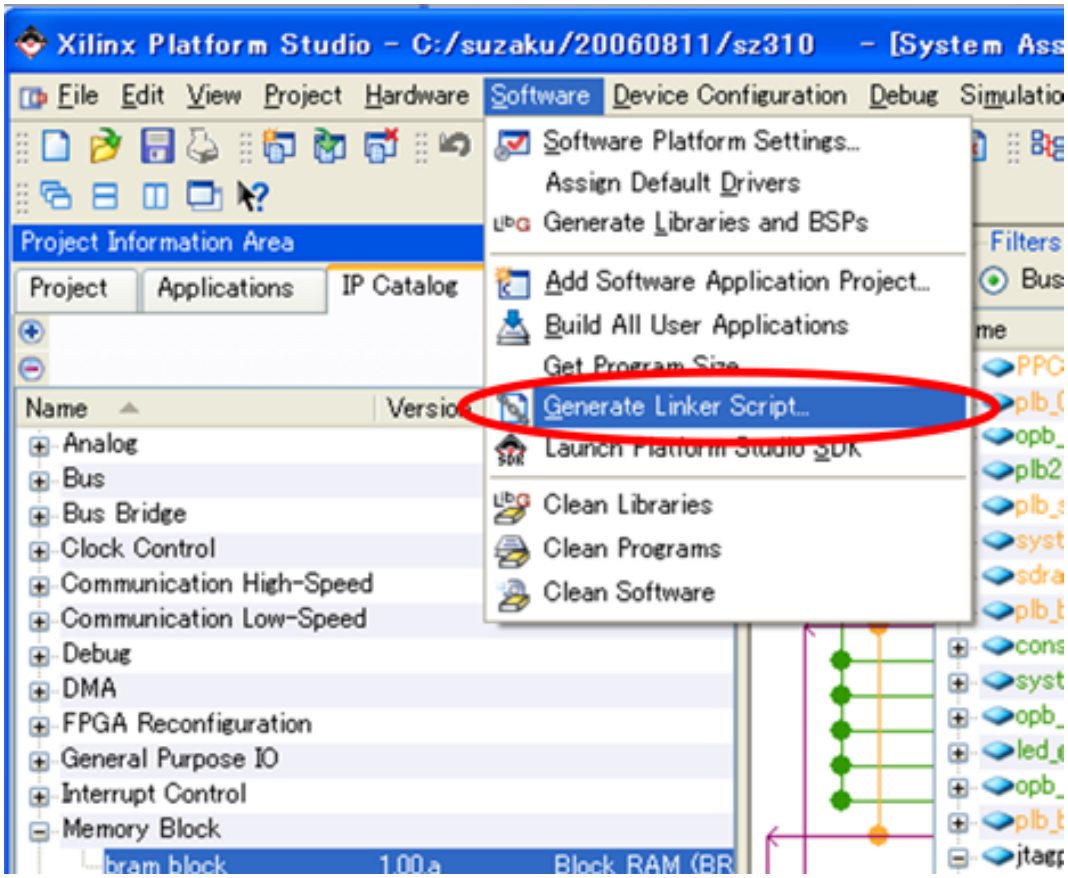


図 11.61. 割り込み設定(リンカースクリプト)

[Sections View]の Memory の部分を編集し、[Generate]をクリックして下さい。

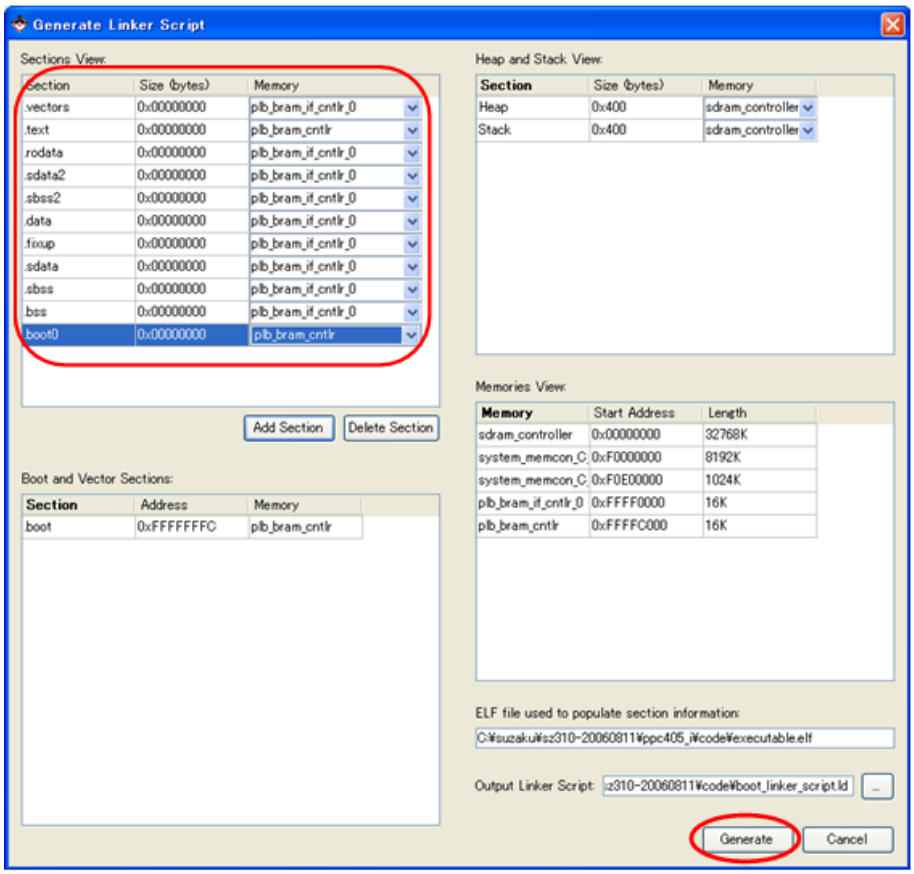


図 11.62. 割り込み設定(リンカースクリプト設定：SZ310)

Section	Memory
.vecotrs	plb_bram_if_cntlr_0
.text	plb_bram_cntlr
.rodata	plb_bram_if_cntlr_0
.sdata2	plb_bram_if_cntlr_0
.sbss2	plb_bram_if_cntlr_0
.data	plb_bram_if_cntlr_0
.fixup	plb_bram_if_cntlr_0
.sdata	plb_bram_if_cntlr_0
.sbss	plb_bram_if_cntlr_0
.bss	plb_bram_if_cntlr_0
.boot0	plb_bram_cntlr

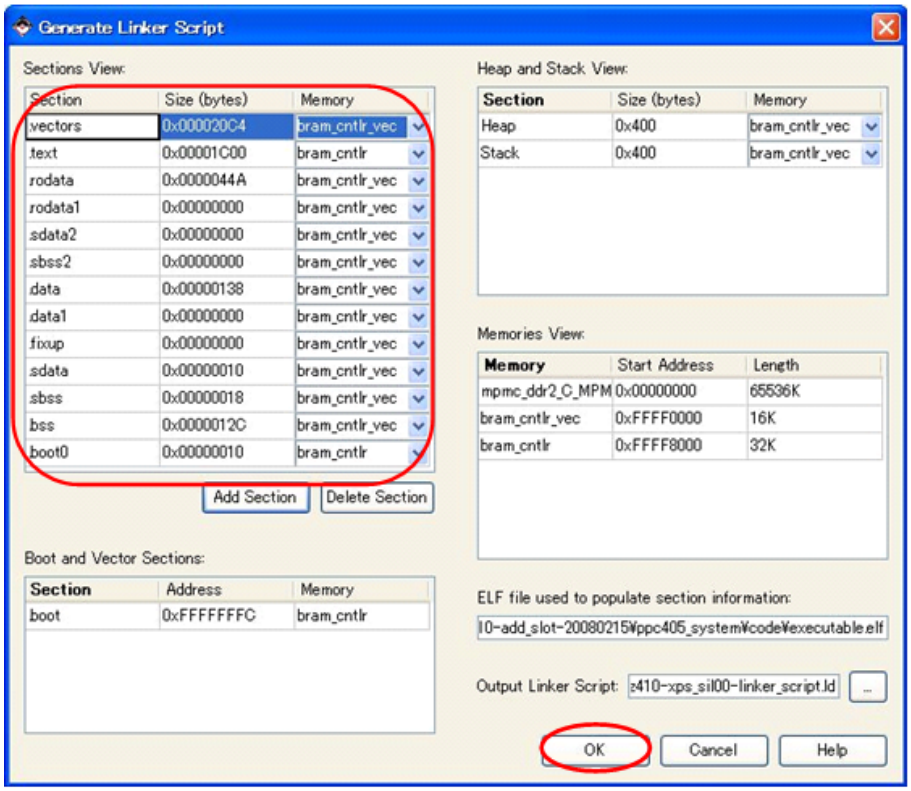


図 11.63. 割り込み設定(リンカースクリプト設定：SZ410)

Section	Memory
.vecotrs	bram_cntlr_vec
.text	bram_cntlr
.rodata	bram_cntlr_vec
.rodata1	bram_cntlr_vec
.sdata2	bram_cntlr_vec
.sbss2	bram_cntlr_vec
.data	bram_cntlr_vec
.data1	bram_cntlr_vec
.fixup	bram_cntlr_vec
.sdata	bram_cntlr_vec
.sbss	bram_cntlr_vec
.bss	bram_cntlr_vec
.boot0	bram_cntlr

11.9.5.8. ピンアサイン

Project Files の UCF File:`data/xps_proj.ucf` をダブルクリックしてください。ピンアサインのファイルが開きます。

ピンアサインを追加入力し、保存してください。

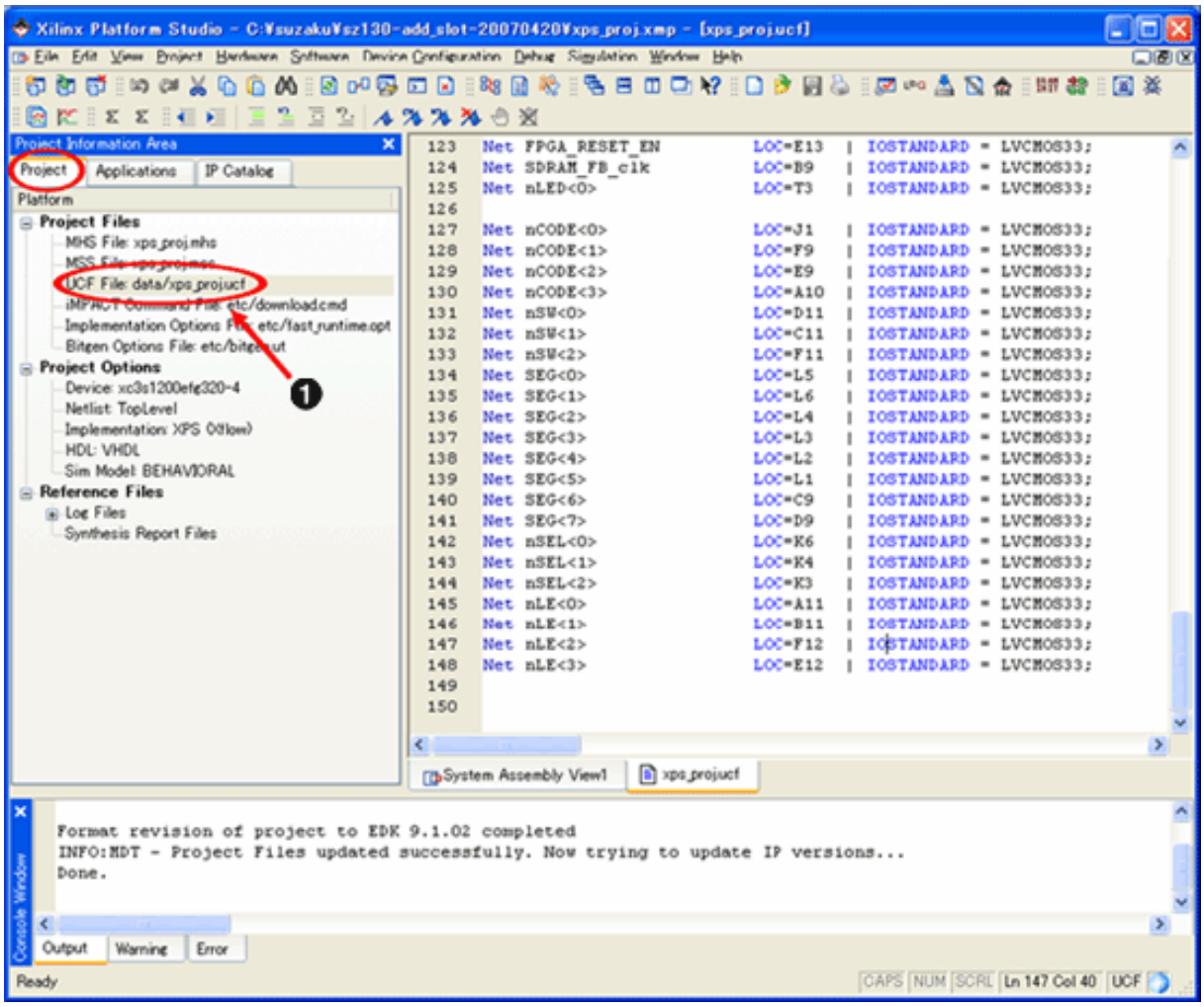


図 11.64. 自作 IP コア(xps_proj.ucf)

❶　UCF File: data/xps_proj.ucf をダブルクリック

表 11.2. 自作 IP コア ピンアサイン

	SZ010 SZ030	SZ130	SZ310	SZ410
nCODE<0>	C8	J1	J16	H5
nCODE<1>	A9	F9	J15	E2
nCODE<2>	A12	E9	J14	D2
nCODE<3>	C10	A10	J13	U9
nSW<0>	A14	D11	K16	L1
nSW<1>	B14	C11	K15	M1
nSW<2>	A13	F11	K14	G4
SEG<0>	C5	L5	F15	P1
SEG<1>	B5	L6	F16	P2
SEG<2>	E6	L4	G13	L2
SEG<3>	D6	L3	G14	M2
SEG<4>	C6	L2	G15	N2
SEG<5>	B6	L1	G16	N3
SEG<6>	A8	C9	N9	Y7
SEG<7>	B8	D9	P9	W7
nSEL<0>	D7	K6	H13	N5
nSEL<1>	C7	K4	H14	M3
nSEL<2>	B7	K3	H15	M4
nLE<0>	E11	A11	L13	E1
nLE<1>	D11	B11	L14	F1
nLE<2>	C12	F12	L15	F2
nLE<3>	B12	E12	L16	G2

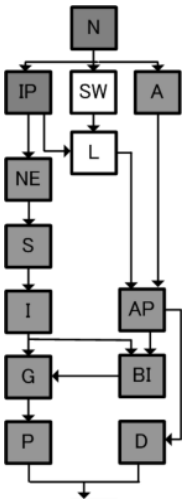
以上で自作コアの追加は終わりです。

11.9.5.9. 自作 IP コア追加 作業まとめ

1. IP コアが EDK に読み込まれているか確認
2. IP コアを SUZAKU のデフォルトに追加
3. OPB に接続
4. アドレスを設定
5. 入出力信号の接続

11.9.6. BBoot 編集

今回は、SUZAKU のデフォルトに入っている BBoot に手を加えて、アプリケーションを作成します。
BBoot とは、FPGA の BRAM に配置され、電源を投入すると最初に動作するプログラムで、ブートローダ Hermit の起動や書き換えを行うものです。
初めに BBoot のプロジェクトにスロットマシンのサンプルソフトを追加します。
そして、`main.c` に変更を加えていきます。
BBoot に追加するスロットマシンのソフトウェアの仕様を下記に示します。



- スタート検出(押しボタンスイッチ(チャタリング除去)を 2 つ以上押した時、7 セグメント LED の数字を回転させる)
- ストップ検出(押しボタンスイッチ(チャタリング除去)を 1 つ押した時、それぞれ対応する 7 セグメント LED の数字の回転を停止させる)
- 3 つの数字一致検出(7 セグメント LED の数字が 3 つそろったら、単色 LED を順次点灯させるトリガー信号を出力する)
- 回転速度制御(ロータリコードスイッチの入力により、7 セグメント LED の数字の回転速度を変更する)
- 電源投入時は自作 IP コアからの割り込みでタイミングをはかり、コンソールから"t"の文字を受けたら、ビジーループでタイミングをはかるモードに変更

スロットマシンのソフトウェアは、自作 IP コアからの割り込みでタイミングをはかるモードと、ビジーループでタイミングをはかるモードを実装しています。

組み込みソフトウェアにとって、割り込みは重要機能のひとつです。

「11.11. ソフトウェアのデバッグ」でデバッカを起動し、サンプルソフトウェアの流れを確認しますので、是非、詳細動作を確認してください。

11.9.6.1.

SUZAKU に電源投入後、最初に動くデフォルトの SUZAKU のブートローダ BBoot は、以下のフローで実行されます。

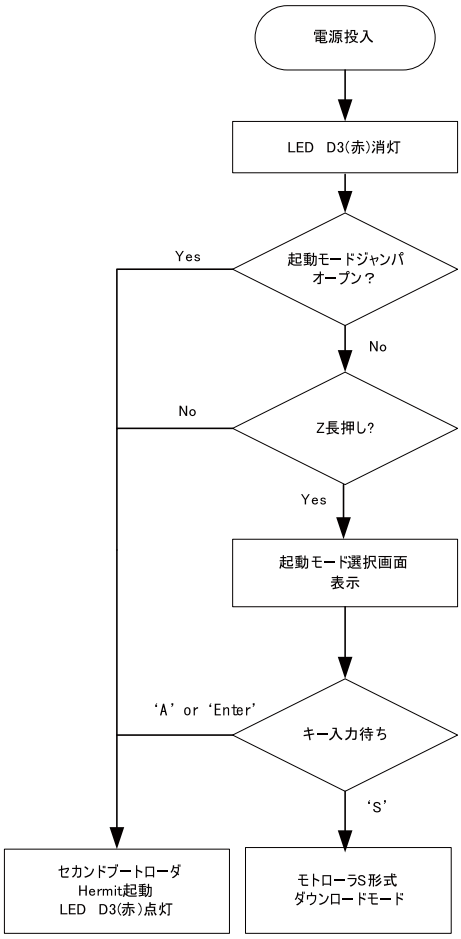
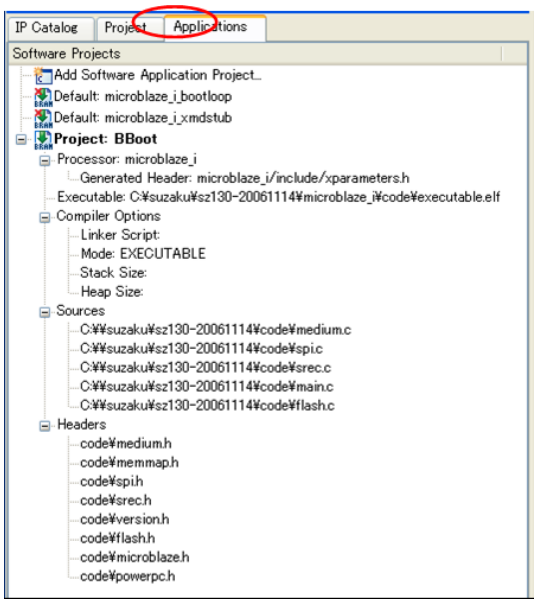


図 11.65. BBoot のフロー

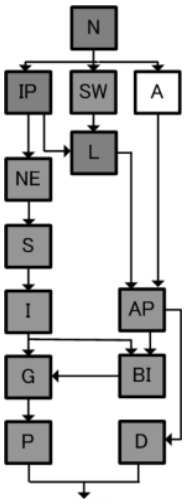
Applications のタブをクリックしてください。BBoot は下記のソースファイル等で構成されます。



medium.c	UART での送受信
spi.c	SPI フラッシュメモリに対する Read、Write(SZ130,SZ410 で使用)
srec.c	モトローラ S 形式のファイルの受信
flash.c	フラッシュメモリに対する Read、Write(SZ010,SZ030,SZ310 で使用)
main.c	ブートモードのチェック、コマンドの受信、ブートローダ Hermit のコピーとジャンプ、モトローラ S 形式のファイルの受信 (EDK では、ベクタやリンカがデフォルトで設定されているため、main 関数を書けば、main 関数からプログラムがスタートするようにコンパイルしてくれます。)

図 11.66. BBoot の構成

下図のように、BBoot にスロットマシンのソフトウェアを追加します。スロットマシンはビジーループモードと、割り込みモードの 2 通りで追加します。



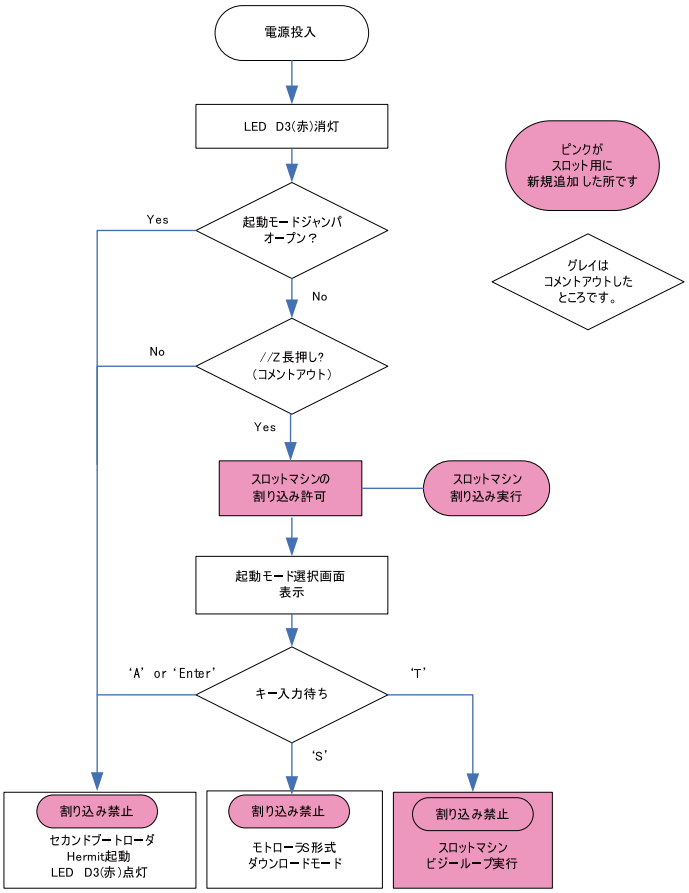


図 11.67. スロットマシンのフロー

11.9.6.2. ライブラリ, ドライバ生成

Generate Libraries and Drivers LibGをクリックして下さい。

ライブラリと様々な設定を定義したヘッダファイルが出来上がります。

`xparametaers.h` を開いてください。`xparameters.h` にはシステムのアドレスマップが定義されます。

自作 IP コアの BASEADDR と HIGHADDR も自動で定義されています。

例 11.13. xparameters.h の定義の例

```
/* Definitions for driver OPB_SIL00 */
#define XPAR_OPB_SIL00_NUM_INSTANCES 1

/* Definitions for peripheral OPB_SIL00_0 */
#define XPAR_OPB_SIL00_0_DEVICE_ID 0
#define XPAR_OPB_SIL00_0_BASEADDR 0xFFFFD000
#define XPAR_OPB_SIL00_0_HIGHADDR 0xFFFFD1FF
```

ソフトウェアに関するファイルは

"C:\suzaku\sz***-yyyymmdd\microblaze_i | ppc405_i | ppc405_system"の下に収められます。このフォルダの下の"\include\opb_sil00.h | xps_sil00.h"を開いてください。自作 IP コアを扱うことのできる関数等が定義されています。

11.9.6.3. プロジェクトにソースファイル追加

"C\suzaku\sz***-yyyymmdd\code"フォルダに slot.c,interrupt.c,slot.h,interrupt.h が入っているのを確認してください。入っていない場合は付属 CD-ROM からコピーしてください。

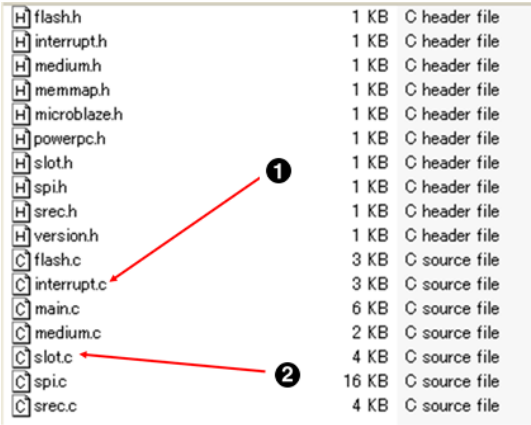


図 11.68. ソースファイル確認

❶
- 割り込みの初期化をし、許可をする
- 割り込みの禁止をする
- 割り込みが発生した時、スロットマシンを動作させる

  (割り込みハンドラ:timer_interrupt_handler)

❷
- スタート検出(押しボタンスイッチ(チャタリング除去)を 2 つ以上押した時、7 セグメント LED の数字を回転させる)
- ストップ検出(押しボタンスイッチ(チャタリング除去)を 1 つ押した時、それぞれ対応する 7 セグメント LED の数字の回転を停止させる)
- 3 つの数字一致検出(7 セグメント LED の数字が 3 つそろったら、単色 LED を順次点灯させるトリガー信号を出力する)
- 回転速度制御(ロータリコードスイッチの入力により、7 セグメント LED の数字の回転速度を変更する)

Applications の Sources を右クリックし、メニューの Add Existing Files...を選択し、Sources に `slot.c`、`interrupt.c` を追加してください。ソースは`"c:\suzaku\sz***-yyyymmdd\code"`の下に収められています。

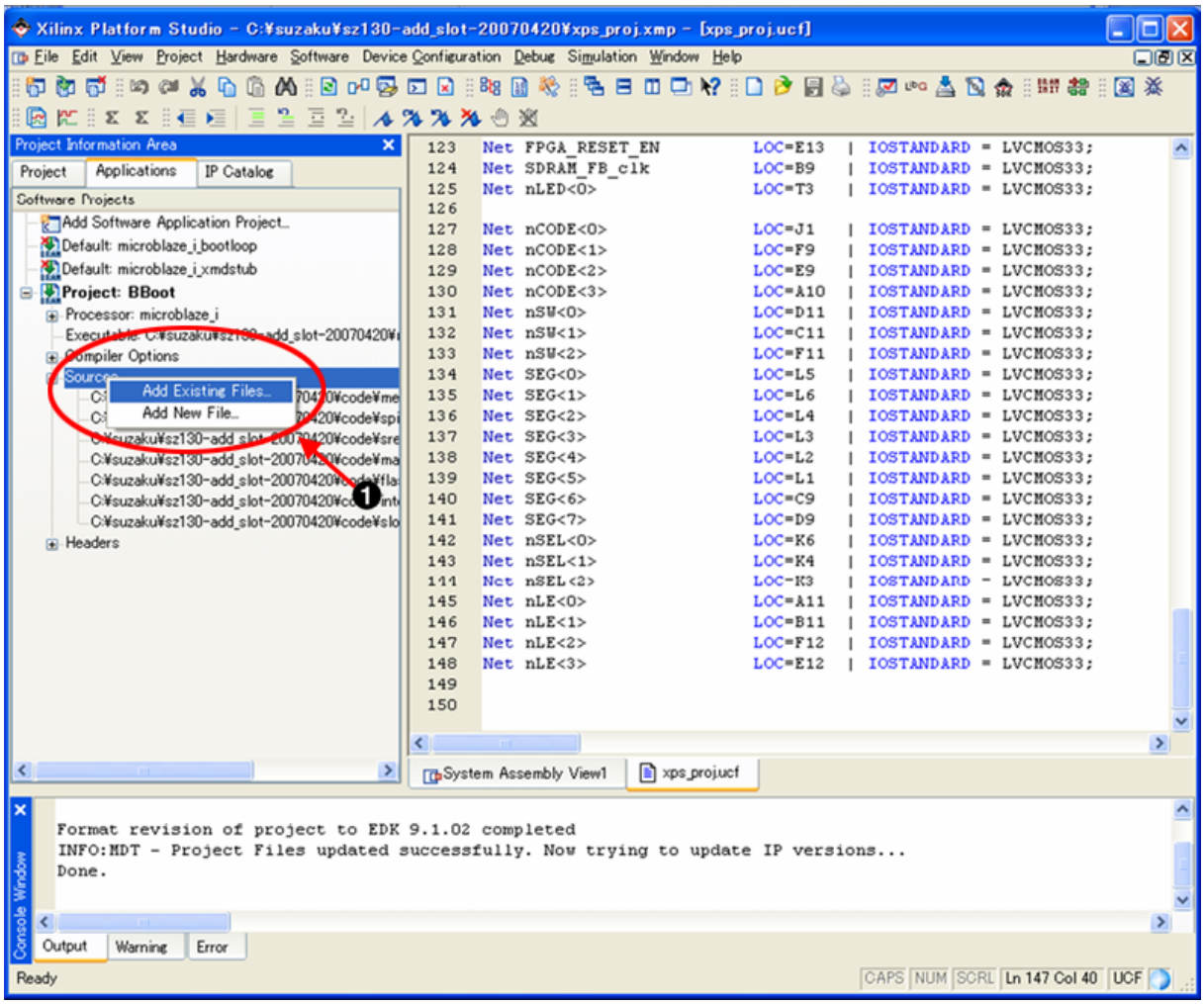


図 11.69. ソースファイル追加

❶　右クリックをしてメニューを出し Add Existing Files...を選択

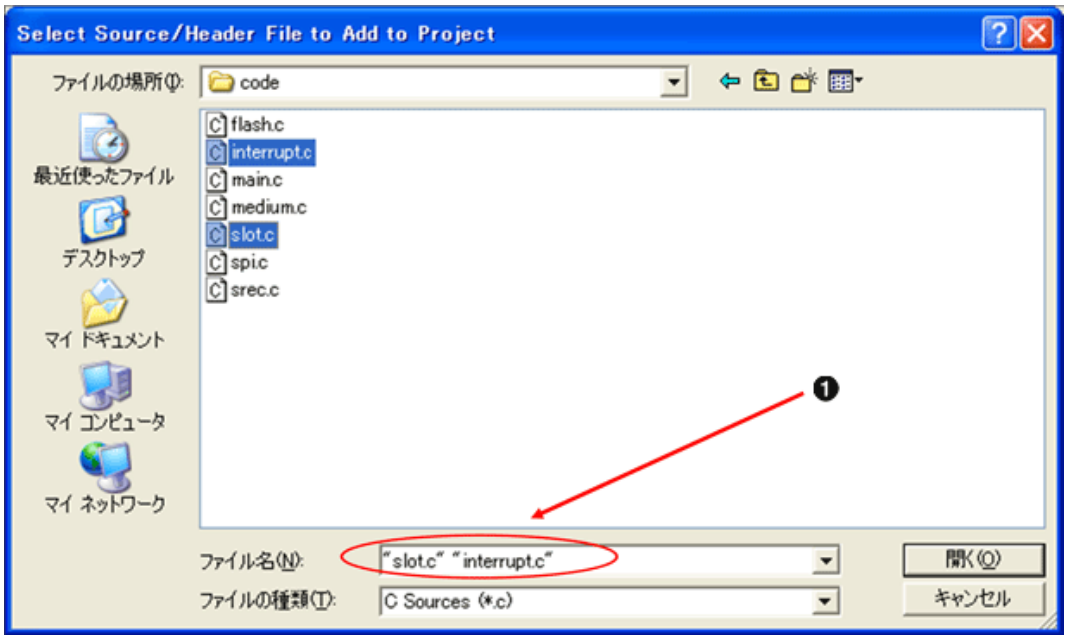


図 11.70. ソースファイル選択

❶　“interrupt.c”、”slot.c”を追加

Applications の Headers を右クリックし、メニューの Add Existing Files...を選択し、Headers に `slot.h`、`interrupt.h` を追加してください。

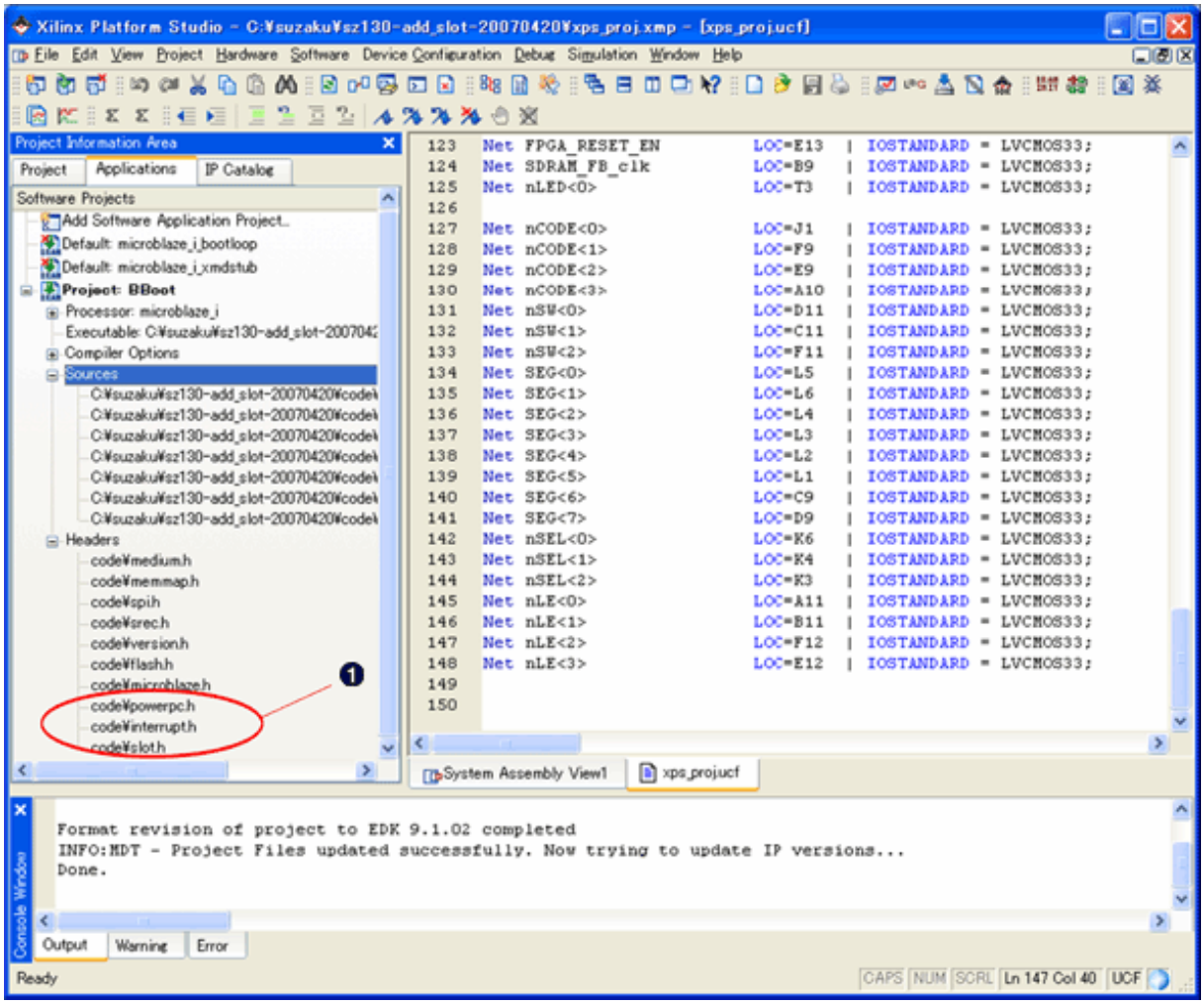


図 11.71. ヘッダファイル追加

❶　`slot.h`、`interrupt.h` 追加

11.9.6.4. main.c ソースコード編集

Applications タブをクリックしてください。Project:BBoot の Sources の `main.c` をダブルクリックして開き、太字の部分を確認してください。

例 11.14. 自作 IP コア(main.c)

```
#include <ctype.h>
#include <xuartlite_l.h>
#include "version.h"
#include "memmap.h"
#include "srec.h"
#include "medium.h"
#include "spi.h"
#include "flash.h"
```

```
#ifdef XPAR_OCM_TEMAC_SZ410_0_BASEADDR
#include "xpseudo_asm.h"
#endif

#ifdef  XPAR_OPB_SIL00U_0_BASEADDR    ❶
#define XPAR_OPB_SIL00_0_BASEADDR   XPAR_OPB_SIL00U_0_BASEADDR
#elif   XPAR_SIL_CNTLR_BASEADDR
#define XPAR_OPB_SIL00_0_BASEADDR   XPAR_SIL_CNTLR_BASEADDR
#endif

#define LED_GPIO(v)    (*(volatile unsigned long *)(LED_REGISTER_BASEADDR) = (v)
#define LED_ON                    (0)
#define LED_OFF                   (1)

#define MAX_BUFFER_SIZE         (128)

#ifdef   SPI_REGISTER_BASEADDR
#define BOOTLOADER_OFFSET_SPI  (0x00100000)
#else
#define FLASH_4MiB (0x16)
#define FLASH_8MiB (0x17)
#define BOOTLOADER_OFFSET_4MiB_FLASH (0x00080000)
#define BOOTLOADER_OFFSET_8MiB_FLASH (0x00100000)
#endif


// 中略


int main(void)
{
    unsigned int bootloader_offset;
    char      key;

    LED_GPIO(LED_OFF);


// 中略



    if (get_bootloader_offset(&bootloader_offset) < 0)
        goto halt;

    if (is_autoboot_mode()) {
        second_bootloader(bootloader_offset);
    }

#ifdef XPAR_OPB_SIL00_0_BASEADDR
        interrupt_init();       //割り込み許可

#else

  myprint("\r\n\r\n"BBOOT_NAME " v" BBOOT_VERSION " (" TARGET_CPU ")\r\n");
  myprint("Press 'z' or 'Z' for BBoot Menu.\r\n");

  /* busy loop to wait getting a char 'z' or 'Z' */
```

```
  busy_wait(BBOOTMENU_WAITTIME);
  if (XUartLite_mIsReceiveEmpty(XPAR_CONSOLE_UART_BASEADDR) ||
                          ((key = get_char()) != 0 && key != 'z' && key != 'Z'))
      second_bootloader(bootloader_offset);
#endif

    /* clear for long time pushing */
    clear_rx_fifo();
    myprint("\r\n\r\nPlease choose one of the following and hit enter.\r\n");
    myprint("a: activate second stage bootloader (default)\r\n");
    myprint("s: download a s-record file\r\n");

#ifdef XPAR_OPB_SIL00_0_BASEADDR
    myprint("t: busy loop type slot-machine\r\n");
#endif

    while (1) {
        key = get_char();
        switch (key) {
            case 'a':    /* activate second stage bootloader */
            case 'A':
            case '\r':
            case '\n':

#ifdef XPAR_OPB_SIL00_0_BASEADDR
                interrupt_clean();      //割り込み禁止
#endif

                second_bootloader(bootloader_offset);
                break;
            case 's':
            case 'S':

#ifdef XPAR_OPB_SIL00_0_BASEADDR
                interrupt_clean();      //割り込み禁止
#endif

                myprint("Start sending S-Record!!\r\n");
                download();
                break;

#ifdef XPAR_OPB_SIL00_0_BASEADDR
            case 't':
            case 'T':                                   ❷
                interrupt_clean(); //割り込み禁止
                myprint("busy loop type slot-machine\r\n");
                while (1) {
                     busy_wait(10000);
                     slot();
                }
#endif

#ifdef UART2_BASEADDR
            case 'u':
            case 'U':
                myprint("Check Uart2\n\r");
```

```
                while (XUartLite_mIsReceiveEmpty(UART2_BASEADDR));
                    XUartLite_SendByte(UART2_BASEADDR,
                            (Xuint8)XUartLite_RecvByte(UART2_BASEADDR));
                    myprint("Done\n\r");
                break;
#endif
            default:
                myprint("Invalid selection.\r\n");
            case 'z':
            case 'Z':
                clear_rx_fifo();
                break;
            }
        }
halt:
    myprint("Halting...\r\n");
    return 0;

}
```

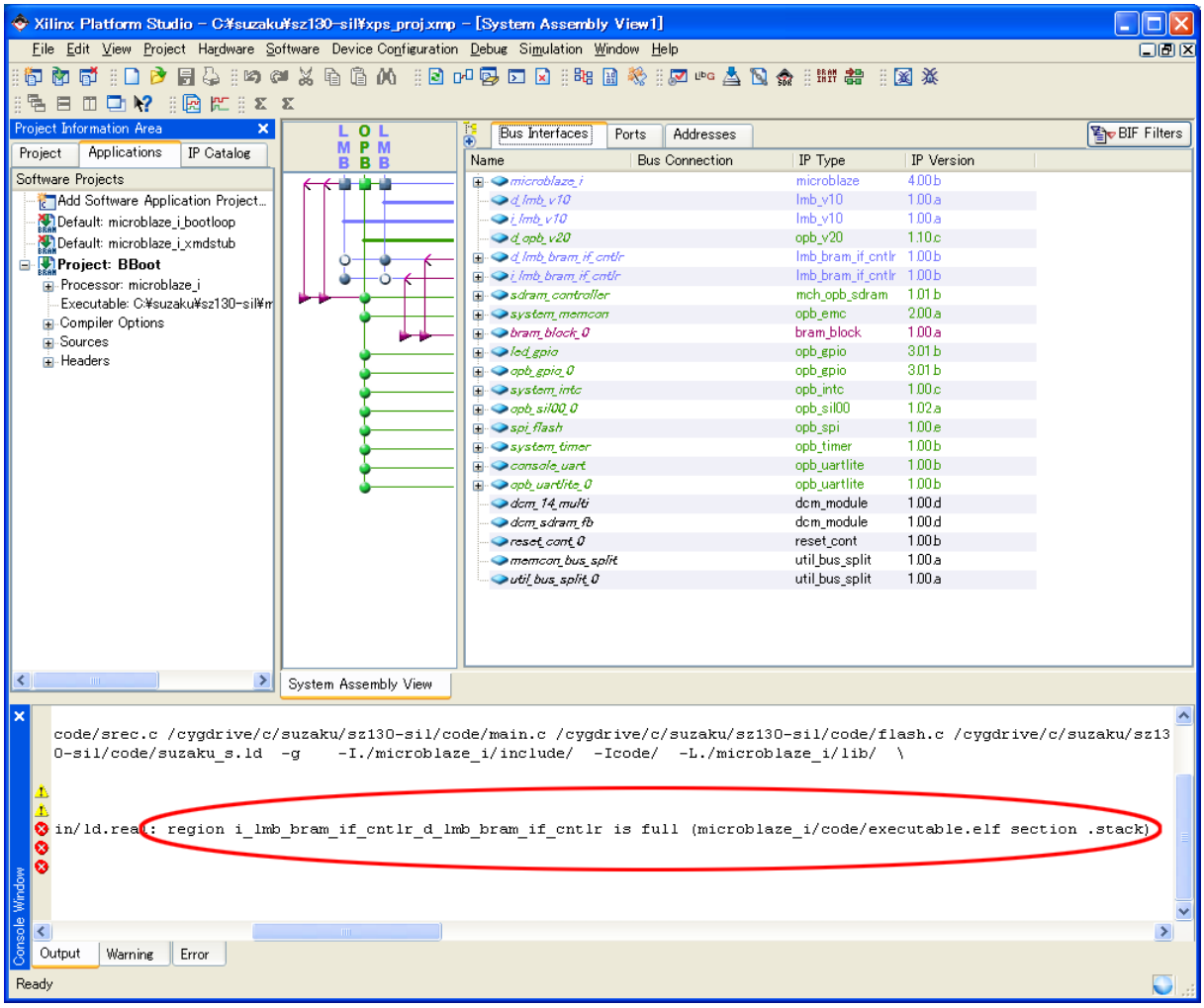
❶　XPAR_OPB_SIL00_0_BASEADDR が `xparameters.h` に定義されているとスロットマシンのプログラムが呼び出されます。自分の `xparameters.h` の定義と違う場合は適宜変更してください。

❷　コンソールから'T'が入力されたらビジーループでスロットマシンを実行

11.9.7. BRAM の容量を増やす

ソフトウェアのサイズが BRAM の容量を越えた場合は、BRAM の容量を増やす必要があります。デフォルトの BRAM の容量は、SZ010,SZ030,SZ130 が 8kByte、SZ310,SZ410 が 16kByte です。

「11.9.11. プログラムファイル作成」で以下のようなエラーメッセージが表示されたら、BRAM の容量不足です。



SZ010,SZ030,SZ130 の場合 d_lmb_bram_if_cntlr を右クリックしてメニューを出し、Configure IP を選択してください。[LMB BRAM High Address]を 0x00003FFF に変更し、[OK]をクリックして下さい。

i_lmb_bram_if_cntlr も同様に 0x00003FFF に変更してください。これで BRAM 容量が 16KByte に変更されます。

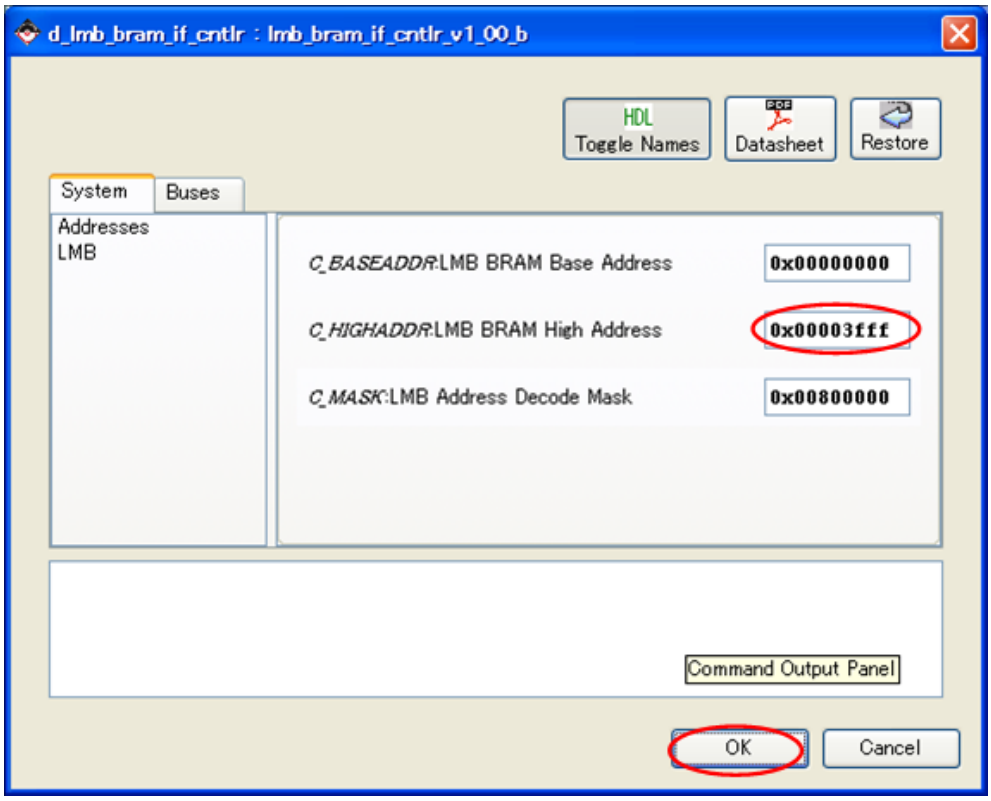


図 11.72. 8KByte → 16KByte に変更(d_lmb_bram_if_cntlr)

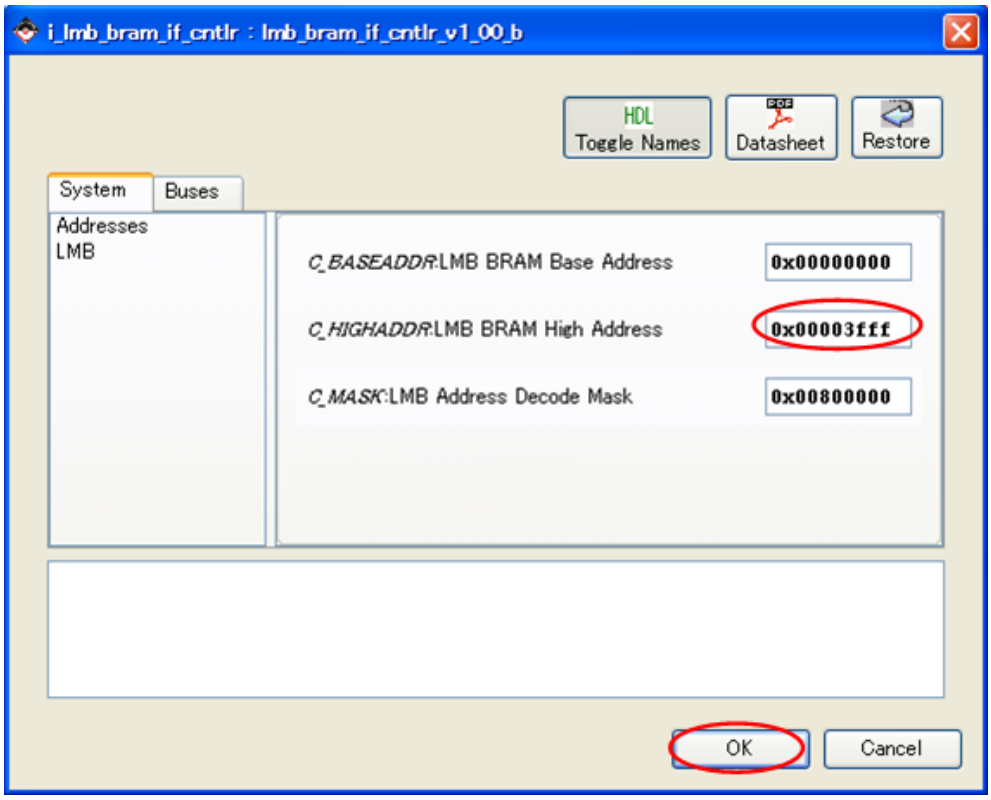


図 11.73. 8KByte → 16KByte に変更(i_lmb_bram_if_cntlr)

SZ310,SZ410 の場合 plb_bram_cntlr | bram_cntlr を右クリックしてメニューを出し、Configure IP を選択してください。[Base Address]を 0xFFFF8000 に変更し、[OK]をクリックして下さい。

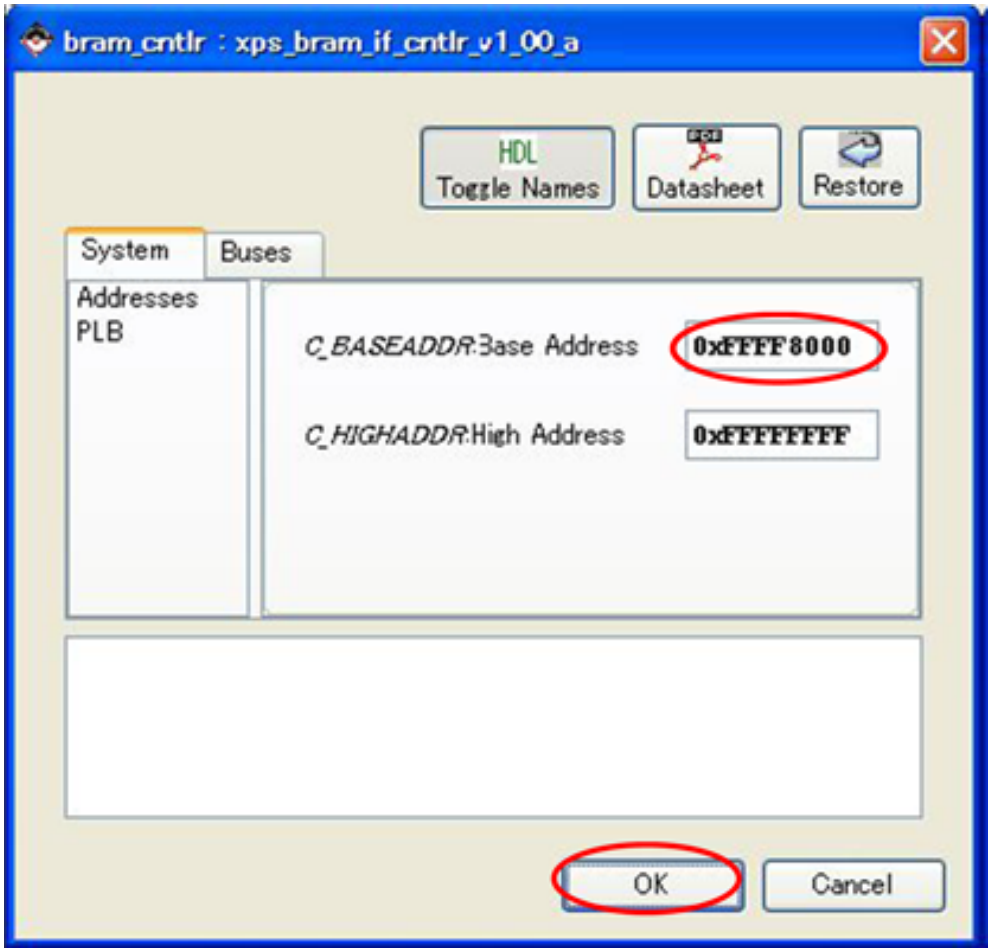


図 11.74. 16KByte → 32KByte に変更

11.9.8. リンカスクリプトを更新

リンカスクリプトを更新するために、以下の作業を行います。

まずは古いリンカスクリプトを削除します。

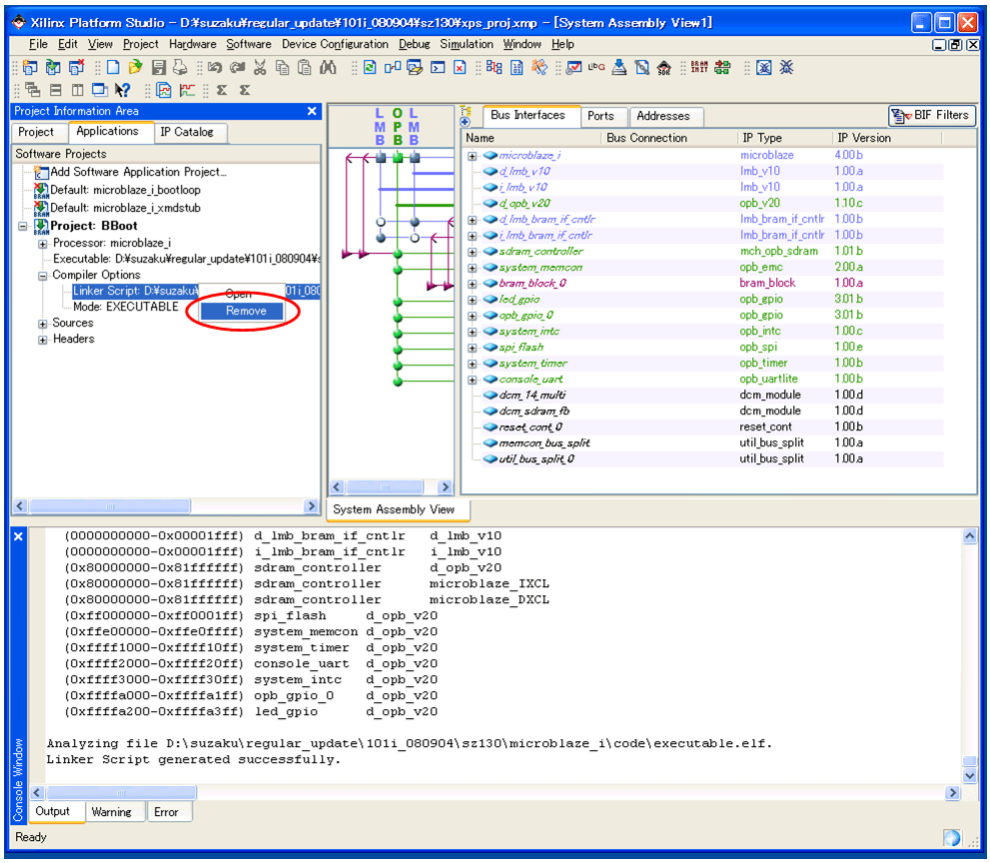


図 11.75. 古いリンカスクリプトの削除

Compiler Options の下の Linker Script をダブルクリックしてください、「図 11.76. 新しいスクリプトを選択」が出てきます。

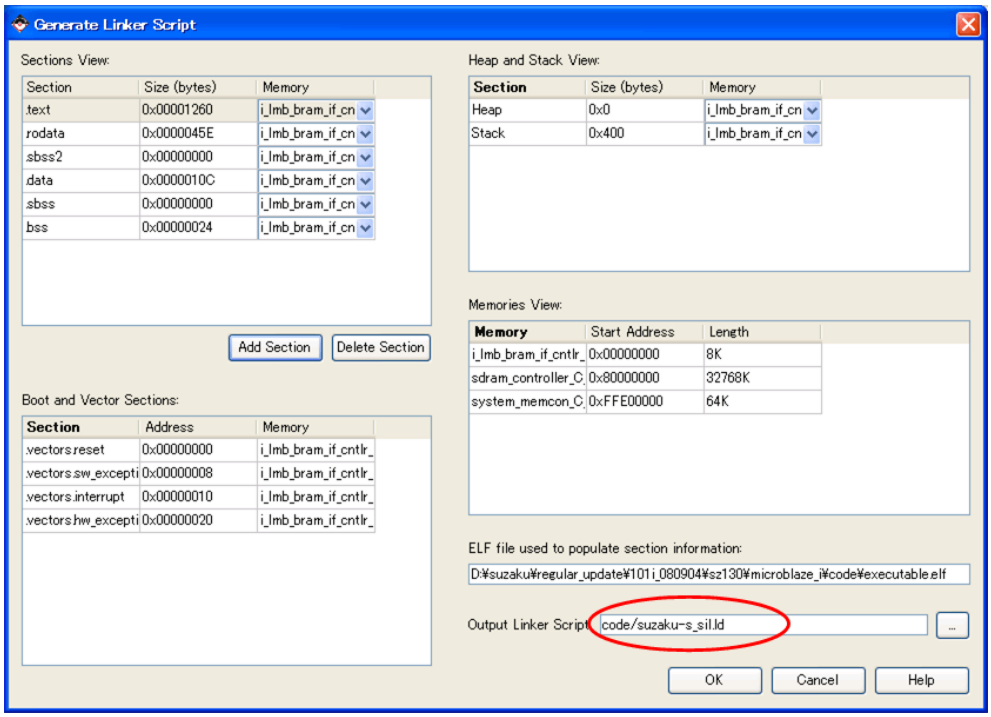
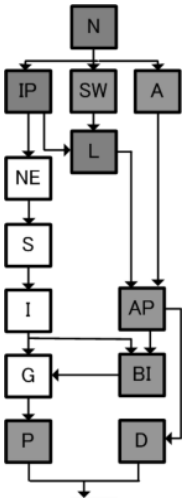


図 11.76. 新しいスクリプトを選択

Output Linker Script の後方に新しいリンカスクリプトファイル名を入力して、OK をクリックしてください。

11.9.9. ネットリスト, プログラムファイル(Hard のみ) 作成



これで自作 IP コアの追加が終了しました。自作 IP コアに間違いがないかチェックします。

[Hardware]→[Generate Netlist]をクリックして下さい。ネットリストが生成されます。

[Hardware]→[Generate Bitstream]をクリックして下さい。ソフトウェアを含まない bit ファイルが生成されます。

もし自作 IP コアにエラーがある場合、`synthesis/opb_sil00_0_wrapper_xst.srp | xps_sil00_0_wrapper_xst.srp` にログが表示されるので、これを開いてエラーを確認し、修正を行ってください。

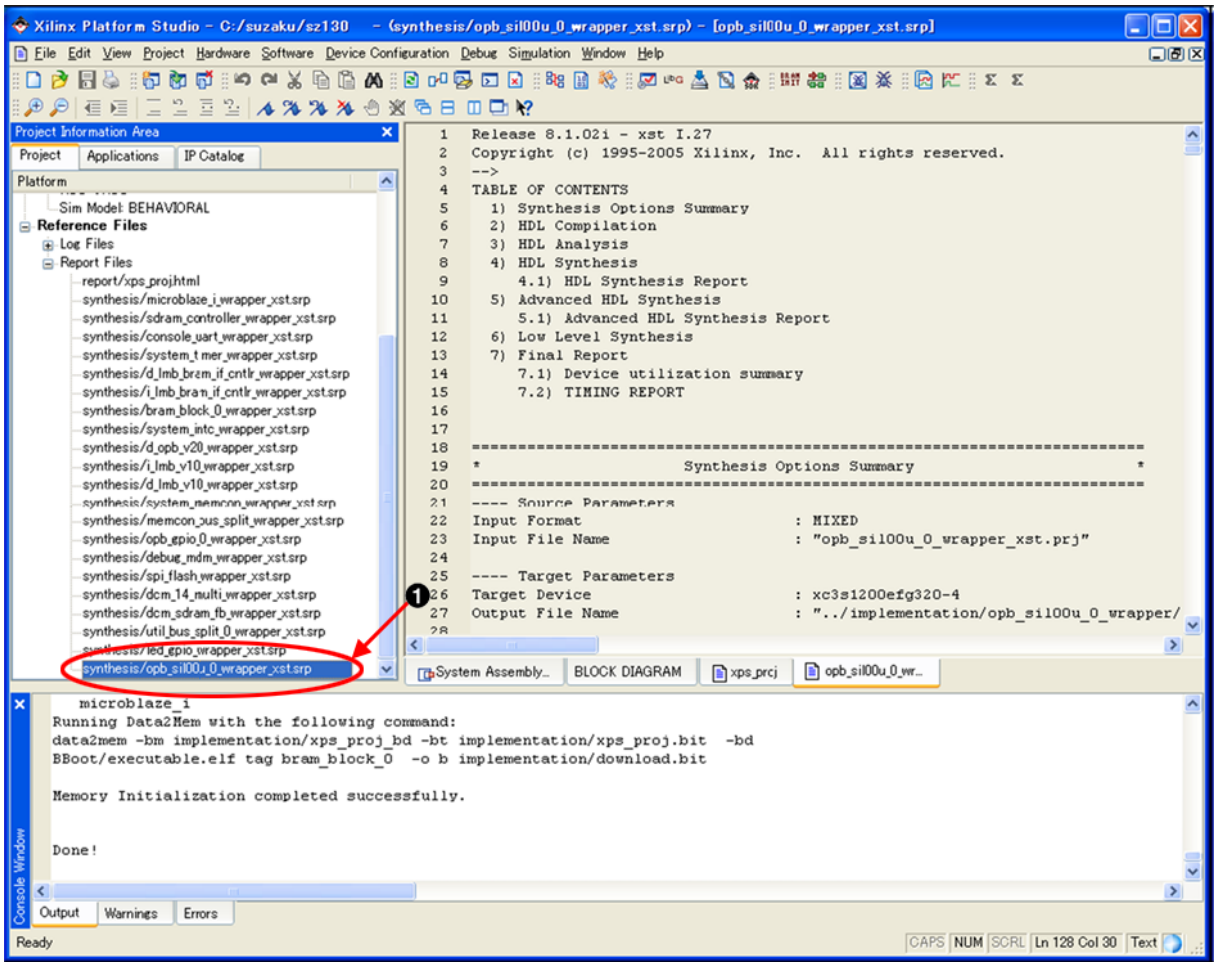


図 11.77. エラーレポート

❶ “synthesis/opb_sil00_0_wrapper_xst.srp| xps_sil00_0_wrapper_xst.srp”に自作コアのログが格納される

11.9.10. アプリケーション生成

[Software]→[Build All User Applications]をクリックして下さい。コンパイラが起動され、各ソフトウェア アプリケーションのプログラム ソースの設定が読み込まれます。 エラーがなければ `executable.elf` が出来上がります。

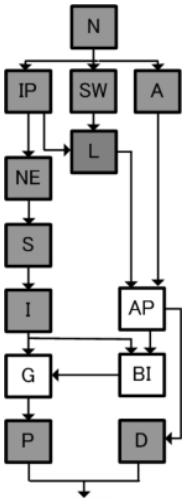
11.9.11. プログラムファイル作成

ハードウェアでつくった bit ファイルの中にソフトウェアを書き込みます。

[Device Configuration]→[Update Bitstream]をクリックしてください。bit ファイルが生成されます。エラーがでたら間違いを修正して再び[Update Bitstream]をクリックしてください。

11.9.12. コンフィギュレーション

JP1,JP2 をショートし、JTAG のコネクタを接続し、シリアルケーブルを向きに注意して接続してください。シリアル通信用ソフトウェアを立ち上げ、AC アダプタ 5V を接続して電源を入れてください。



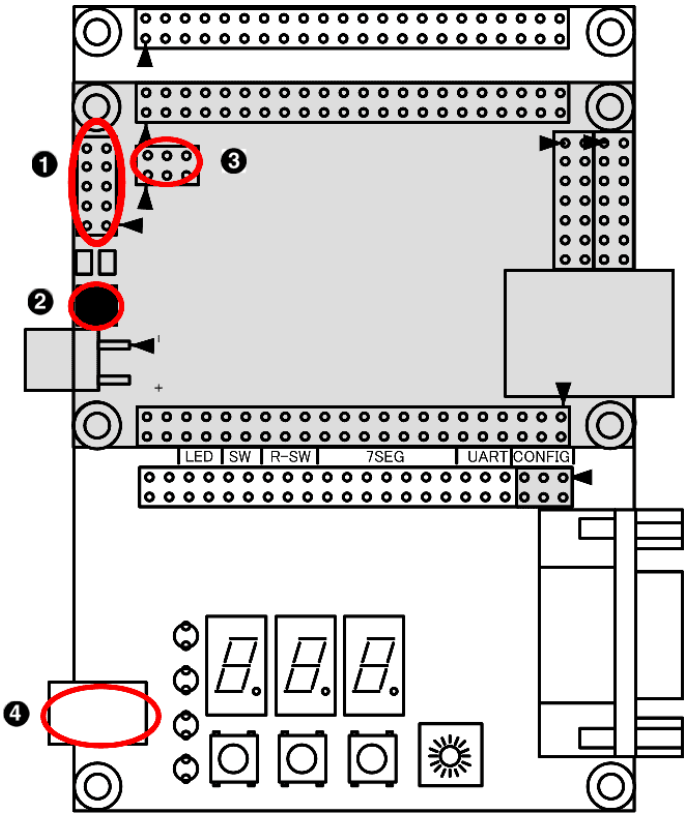


図 11.78. ジャンパの設定等

❶　シリアルケーブル接続

❷　JP1、JP2 ショート

❸　ダウンロードケーブル接続

❹　電源投入

[Device Configuration]→[Download Bitstream]をクリックしてください。バッチモードの iMPACT を使用して FPGA に bit ファイルがコンフィギュレーションされます。

11.10. スロットマシン完成

以上でスロットマシンの完成です！

11.10.1. スロットマシン動作確認

スロットが割り込みモードで動きます。色々触って動きを確認してみてください。

下図のように表示されるので、"T"を押してください。

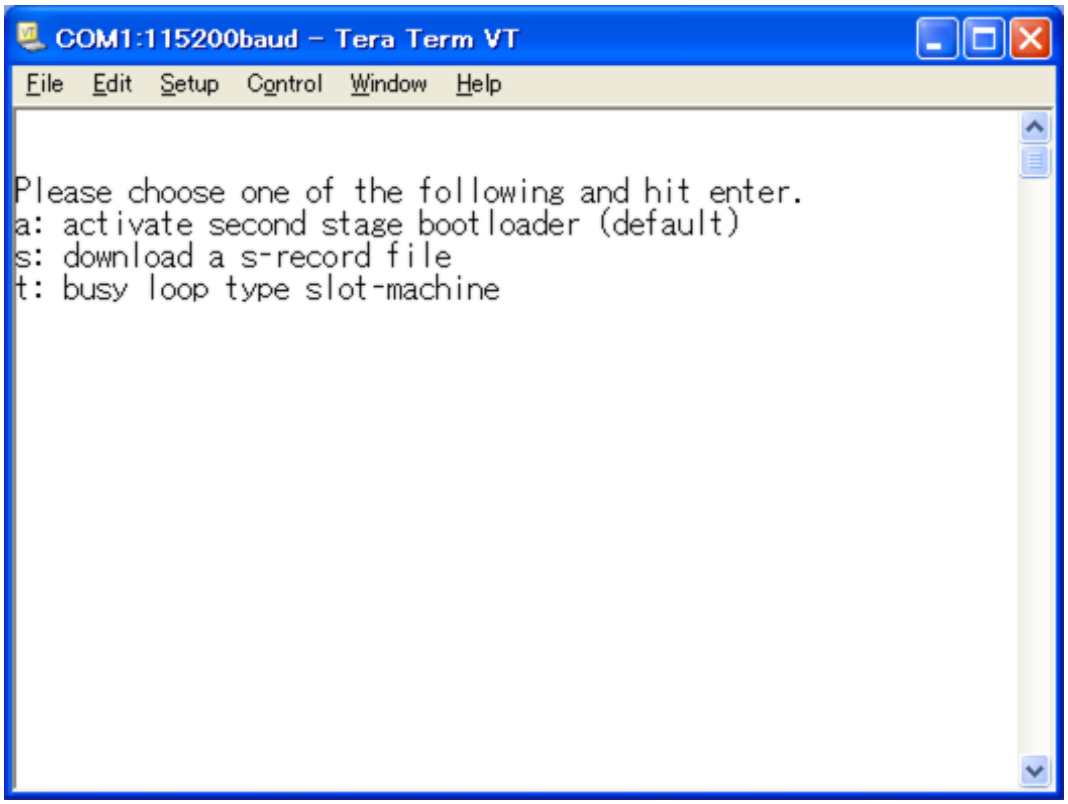


図 11.79. スロットマシン実行画面 1

スロットマシンがビジーループモードで動きます。スロットを色々触って動きを確かめてみてください。

スロットマシンの動きにはほとんど変わりはありませんが大きな違いが一つあります。さきほどまではシリアルコンソールでキー入力を受け付けていましたが、受け付けなくなっていると思います。割り込みでは同時に平行して複数の作業を行うことができます。割り込みが使えると、できる作業の幅がビジーループに比べ格段に増えます。

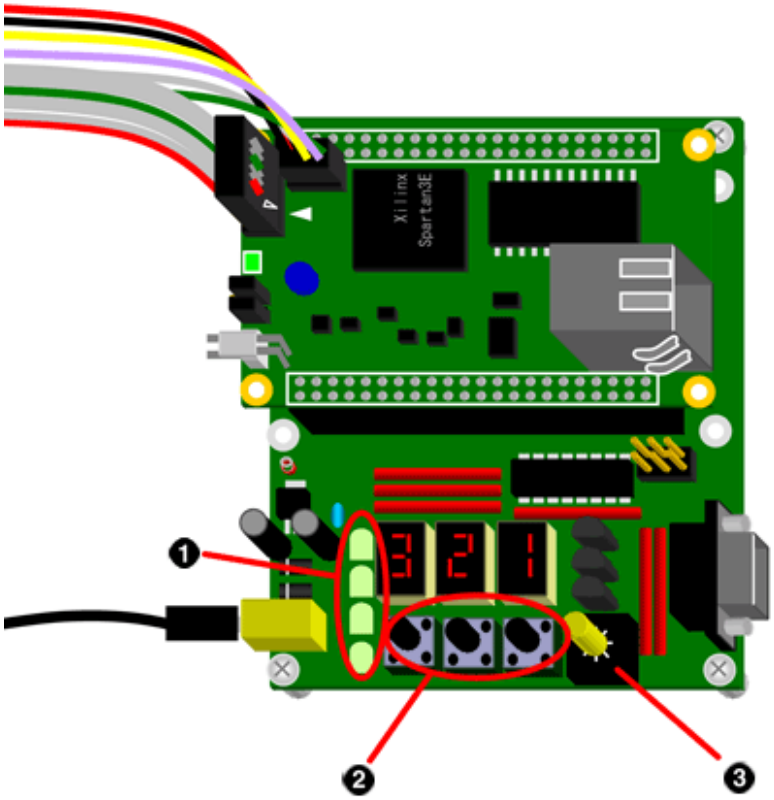


図 11.80. スロットマシン完成

❶　数字がそろうと順次点灯

❷　2 つ以上一緒に押すとスロートの回転が始まる

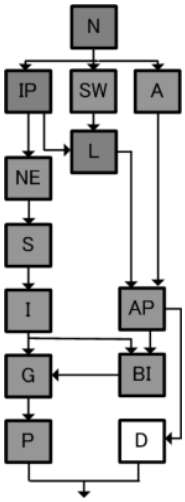
　　1 つ押すと、それぞれ対応する 7 セグメント LED の回転が止まる

❸　0→1→2 とまわすと、数字の回転が速くなる

11.11. ソフトウェアのデバッグ

EDK にはプロセッサと通信し、ソフトウェアをデバッグする機能がついています。
ソフトウェアの処理の流れや変数の確認、メモリのダンプ、CPU レジスタのダンプなどが行え、ソフトウェアの開発にとても便利です。
このデバッグ機能を用いてスロットマシンのプログラム"interrupt.c"、"slot.c"の動作を確認します。
今回は、下記の 2 つの動作をステップ実行で確認します。

- 割込みが発生したときの流れ
- スロットの動作



11.11.1. ソフトウェアデバッグ用に FPGA プロジェクトを更新

SZ010 SZ030 SZ130

まずは、デバッグが行えるようプロジェクトを更新します。SZ310、SZ410 の場合はプロジェクトを更新しなくてもデバッグを行えるので、次のデバッガの設定に進んでください。

microblaze_i を右クリックしてメニューを出し、Configure IP を選択してください。Debug タブをクリックし、[Enable MicroBlaze Debug Module Interface]をチェックしてください。デバッグロジックがイネーブルになります。次にブレークポイント数を設定します。最大 8 まで設定することが出来ます。ここでは、[Number of PC Breakpoints]を 2 にします。設定できたら[OK]をクリックして下さい。

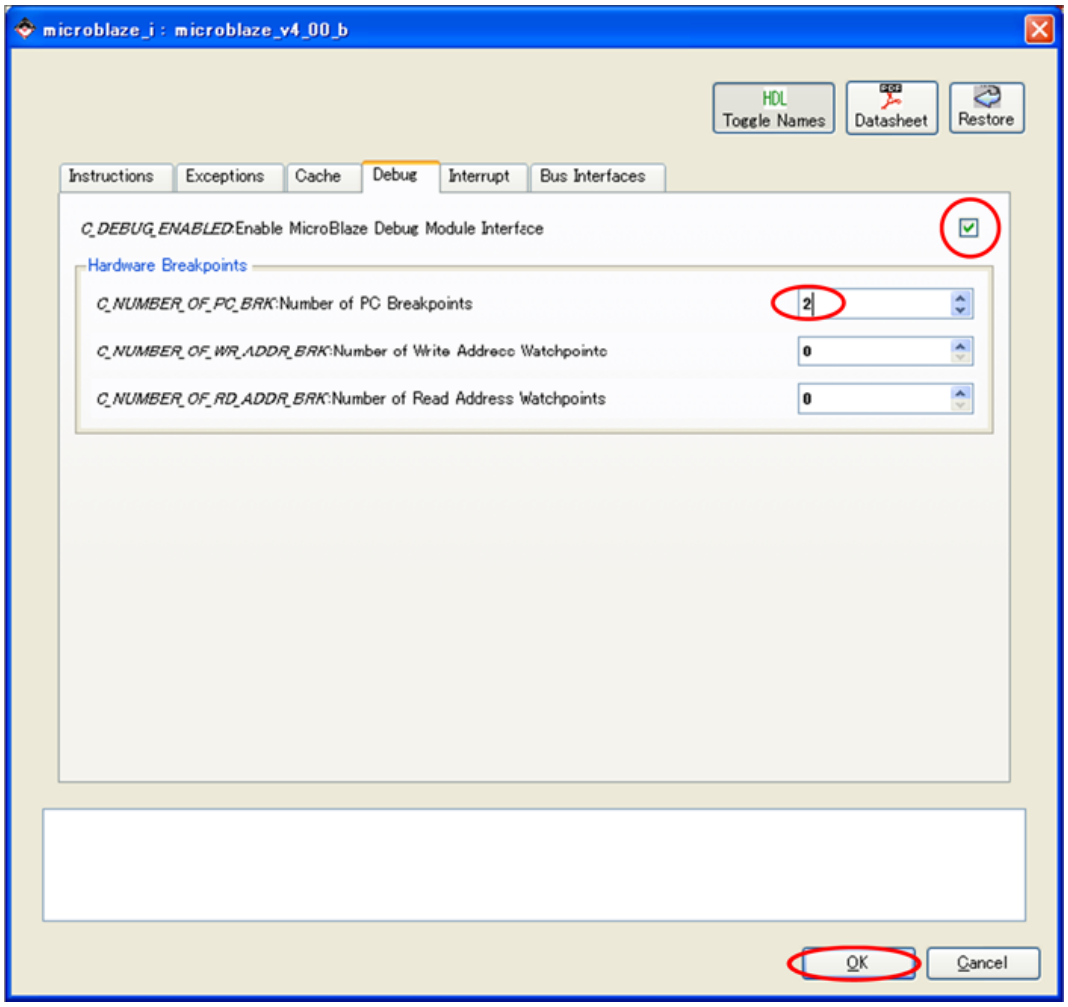


図 11.81. MicroBlaze のデバッグ設定

今回は FSL を使わないのですが、勝手に FSL の設定をされてエラーが出ることがあります。

エラーが出たら、MHS ファイルを開き、PARAMETER C_FSL_DATA_SIZE = 32 に変更してください。

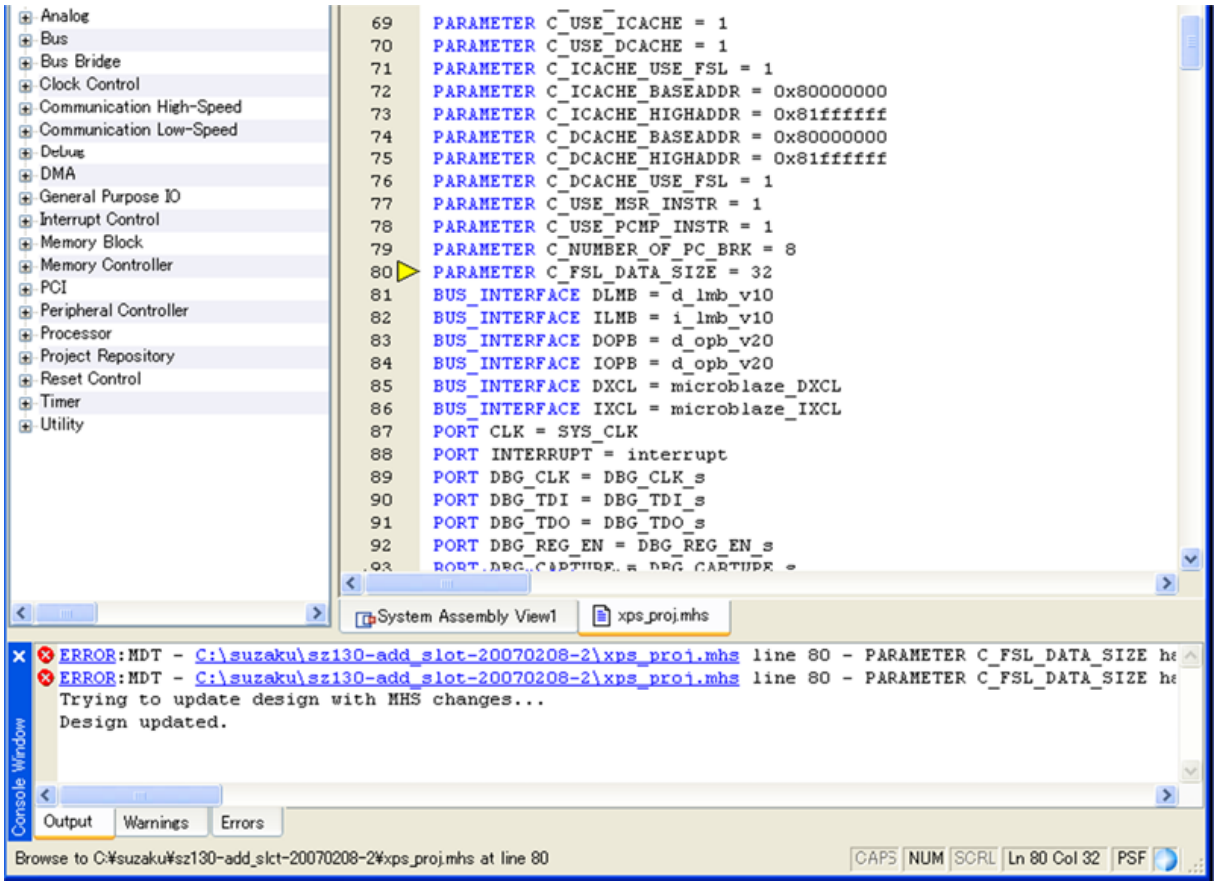


図 11.82. FSL のエラー

IP Catalog のタブをクリックし、Debug→opb_mdm **2.00a** を追加し、OPB バスに接続してください。

現在 SUZAKU のデフォルトの microblaze のバージョンは 4.00b で、opb_mdm2.00a 以降のバージョンのコア(opb_mdm3.00a や mdm1.00a)は microblaze 4.00b に対応していません。バージョンにご注意ください。

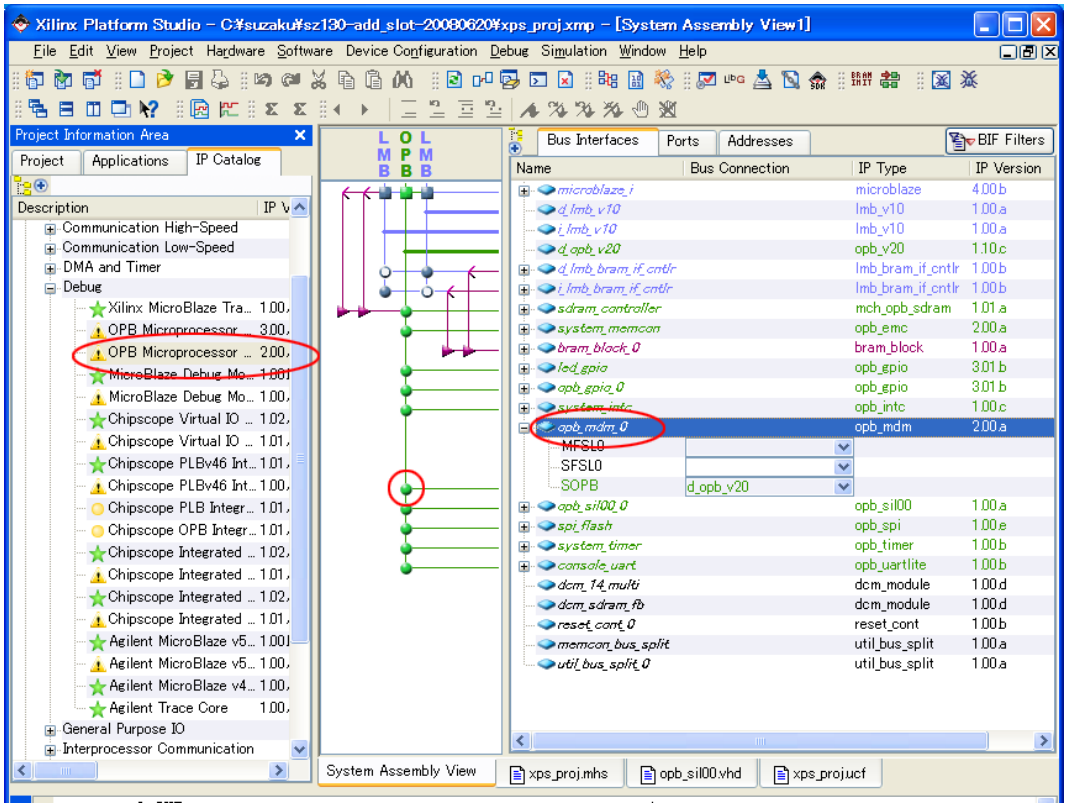


図 11.83. opb-mdm を追加してバスに接続

opb_mdm_0 を右クリックしてメニューを出し、Configure IP を選択してください。[System]タブをクリックし、[Base Address]に 0xFFFFE000、[High Address]に 0xFFFFE0FF と入力し、[OK]をクリックして下さい。アドレスは SUZAKU のメモリマップの Free のところならばどこでも構いません。(「1.4. メモリマップ」参照)

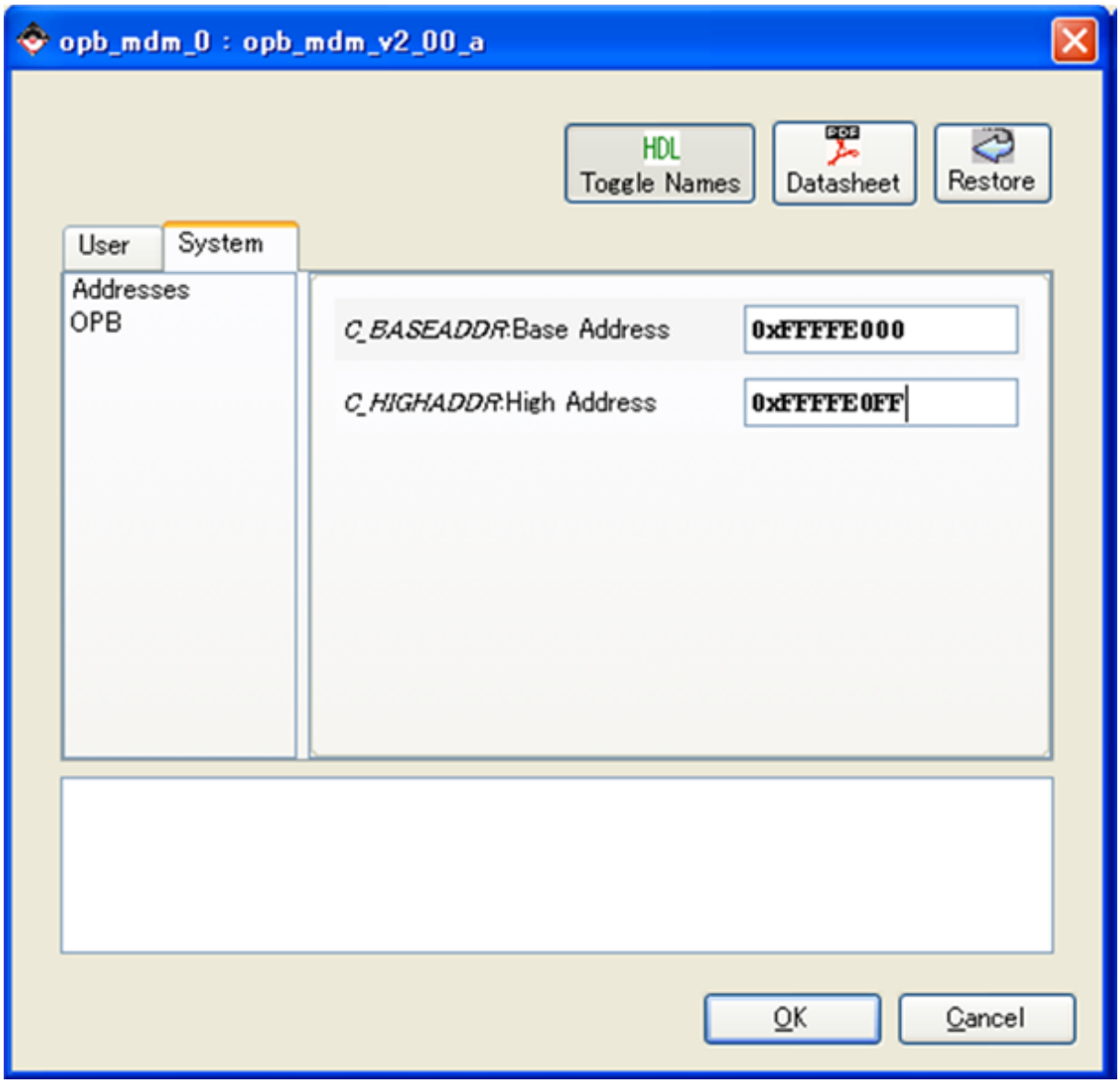


図 11.84. デバッガのアドレス設定

今回はデバッグのため、プログラムを最適化しないでコンパイルします。そのため、BRAM の容量が足りなくなります。そこで BRAM の容量を 8KByte から 16KByte に増やします。すでに増やしている場合は増やす必要はありません。

d_lmb_bram_if_cntlr を右クリックしてメニューを出し、Configure IP を選択してください。[LMB BRAM High Address]を 0x00003FFF に変更し、[OK]をクリックして下さい。i_lmb_bram_if_cntlr も同様に 0x00003FFF に変更してください。これで BRAM 容量が 16KByte に変更されます。

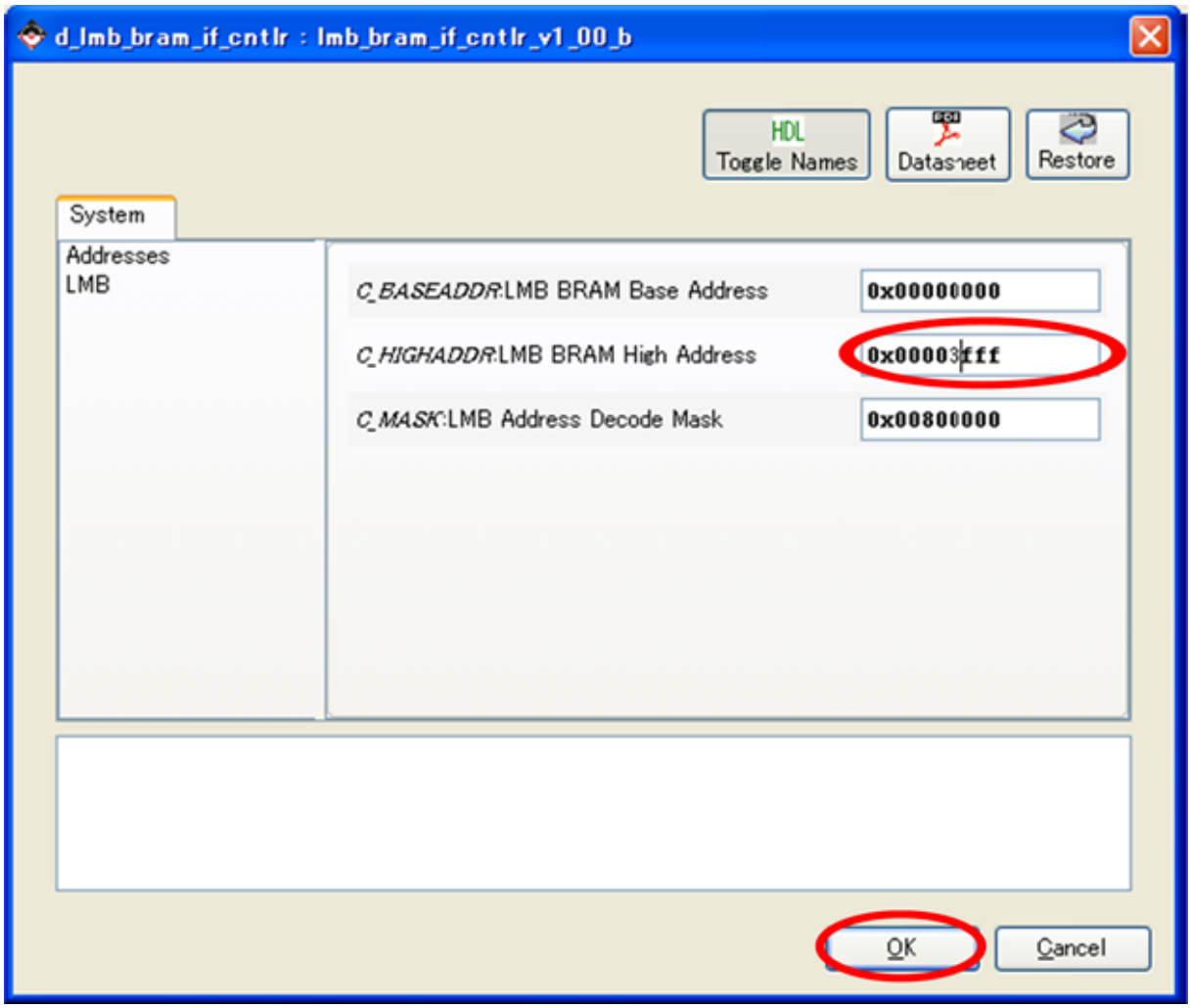


図 11.85. 8KByte → 16KByte に変更

11.11.2. デバッガの設定

Applications タブをクリックし、[Project: BBoot]→[Compiler Option]をダブルクリックして開いてください。以下のような画面が表示されます。Debug and Optimization タブをクリックし、最適化をしないので、Optimization Level を[No Optimization]にして下さい。[Generate Debug Symbols]、[Create Symbols for Debugging] がチェックされているのを確認し、[OK]をクリックしてください。

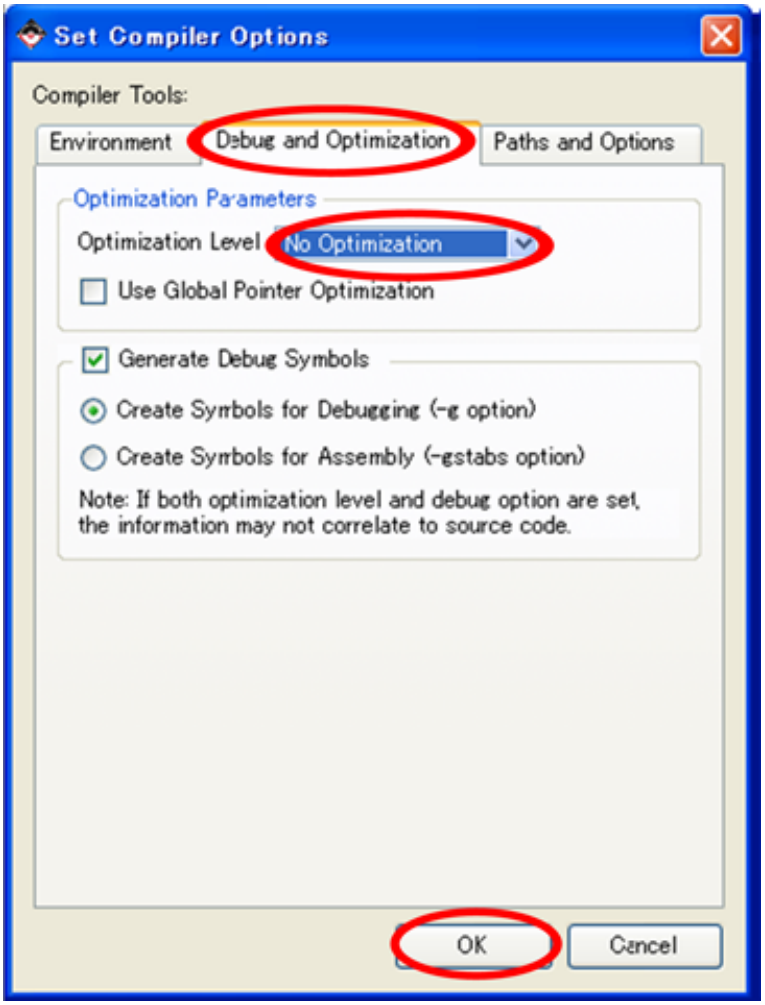


図 11.86. コンパイラオプション

[Debug] → [XMD Debug Options] をクリックしてください。

[Connection Type]を[Hardware]、[JTAG Cable]の Type を[Auto]にし、[Auto Discover JTAG Chain Definition]をチェックしてください。[JTAG Cable]の Type はお使いのケーブルを選んでいただいてもかまいません。設定できたら[OK]をクリックし、設定を保存してください。

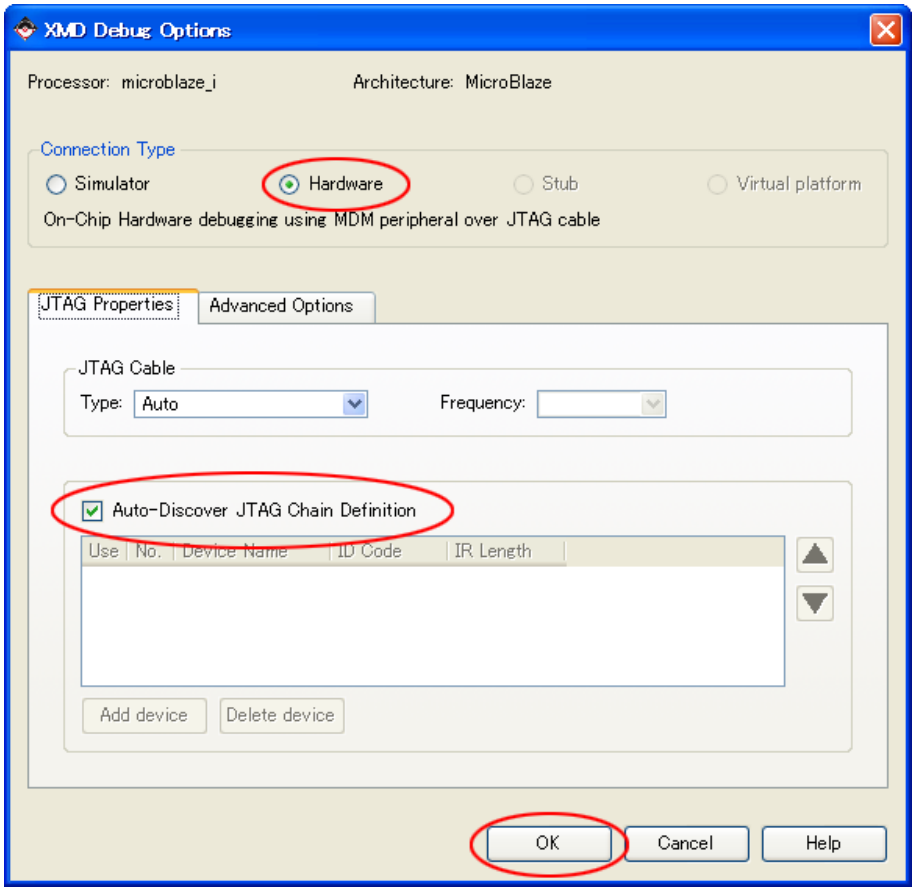


図 11.87. デバッグオプション

以上でプロジェクトの更新は終わりです。

[Device Configuration]→[Update Bitstream]をクリックしてください。

SZ130 の場合、次のようなエラーが出る場合があります。

```
ERROR:Place:249 - Automatic clock placement failed.  Please attempt to analyze the
Global clocking required for thisdesign and either lock the clock placement or area
coords="" locate the logic driven by the clocks so that that the clocks may be
placed in such a way that all logic driven by them may be routed.  The main
restriction on clock placement is that only one clock output signal for any
Primary / Secondary pair of clocks may enter any region.  For further information
see the "Using Global Clock Networks" section in the V-II User Guide (Chapter 2:
Design Considerations)
Phase 4.30 (Checksum:26259fc) REAL time: 53 secs

#中略
```

```
PAR done!
ERROR:Xflow - Program par returned error code 31. Aborting flow execution...
```

この場合、MHS ファイルを開き、SPI_SCK の信号定義部分に"SIGIS = CLK"と追記してください。

```
PORT SPI_SCK = SPI_SCK, DIR = IO, SIGIS = CLK
```

11.11.3. XMD の起動

XMD とは、プロセッサ(MicroBlaze や PowerPC)と PC のデバッグアプリケーションの仲立ちをしてくれるものです。XMD とデバッグアプリケーションは、TCP 経由でやり取りをしますので、ネットワーク経由でも接続できます。

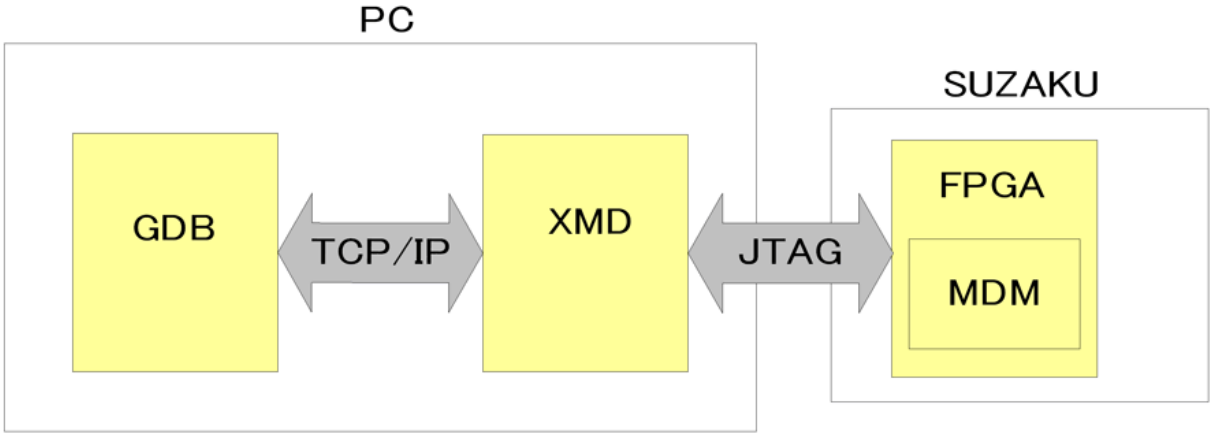


図 11.88. XMD の接続

[Device Configuration]→[Download Bitstream]をクリックして下さい。FPGA にデバッグ機能付きのスロットマシンのコンフィギュレーションデータがダウンロードされます。

[Debug]→[Launch XMD...]をクリックして下さい。

コマンドプロンプトが立ち上がり、以下のように表示されます。

例 11.15. XMD の起動ログ(SZ130 の場合)

```
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 8.2.02 Build EDK_Im_Sp2.4
Copyright (c) 1995-2005 Xilinx, Inc.  All rights reserved.

XMD%
Loading XMP File..
Processor(s) in System ::

Microblaze(1) : microblaze_i
Address Map for Processor microblaze_i
  (0x00000000-0x00001fff) d_lmb_bram_if_cntlr   d_lmb_v10
  (0x00000000-0x00001fff) i_lmb_bram_if_cntlr   i_lmb_v10
  (0x80000000-0x81ffffff) sdram_controller      d_opb_v20
  (0x80000000-0x81ffffff) sdram_controller      microblaze_IXCL
```

```
  (0x80000000-0x81ffffff) sdram_controller      microblaze_DXCL
  (0xff000000-0xff0001ff) spi_flash     d_opb_v20
  (0xffe00000-0xffe0ffff) system_memcon d_opb_v20
  (0xffff1000-0xffff10ff) system_timer  d_opb_v20
  (0xffff2000-0xffff20ff) console_uart  d_opb_v20
  (0xffff3000-0xffff30ff) system_intc   d_opb_v20
  (0xffffa000-0xffffa1ff) opb_gpio_0    d_opb_v20
  (0xffffa200-0xffffa3ff) led_gpio      d_opb_v20
  (0xffffa600-0xffffa6ff) opb_uartlite_0        d_opb_v20
  (0xffffd000-0xffffd1ff) opb_sil00_0   d_opb_v20
  (0xffffe000-0xffffe0ff) opb_mdm_0     d_opb_v20

Connecting to cable (Parallel Port - LPT1).
Checking cable driver.
 Driver windrvr6.sys version = 7.0.0.0. LPT base address = 0378h.
 ECP base address = 0778h.
Cable connection established.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       01c2e093           6        XC3S1200E
Assuming, Device No: 1 contains the MicroBlaze system
Connected to the JTAG MicroProcessor Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
--------------------------------------
Version..........................4.00.b
No of PC Breakpoints...............2
No of Read Addr/Data Watchpoints...0
No of Write Addr/Data Watchpoints..0
Instruction Cache Support..........on
Instruction Cache Base Address.....0x80000000
Instruction Cache High Address.....0x81ffffff
Data Cache Support.................on
Data Cache Base Address............0x80000000
Data Cache High Address............0x81ffffff
Exceptions  Support................off
FPU  Support.......................off
FSL DCache Support.................on
FSL ICache Support.................on
Hard Divider Support...............on
Hard Multiplier Support............on
Barrel Shifter Support.............on
MSR clr/set Instruction Support....on
Compare Instruction Support........on
JTAG MDM Connected to MicroBlaze 1
Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD%
```

以上のようにコンソールに表示がでたら、起動成功です。

11.11.4. GDB を起動し、ソフトウェアをスタートさせる

GDB はソフトウェアデバッグのユーザインターフェースになります。まずは GDB を起動します。

[Debug]→[Launch Software Debugger]をクリックして下さい。以下の画面が立ち上がります。

ここではデフォルト設定のままとします。そのまま[OK]をクリックして下さい。立ち上がらない場合は[File]→[Target Settings...]から以下の画面を立ち上げることが出来ます。

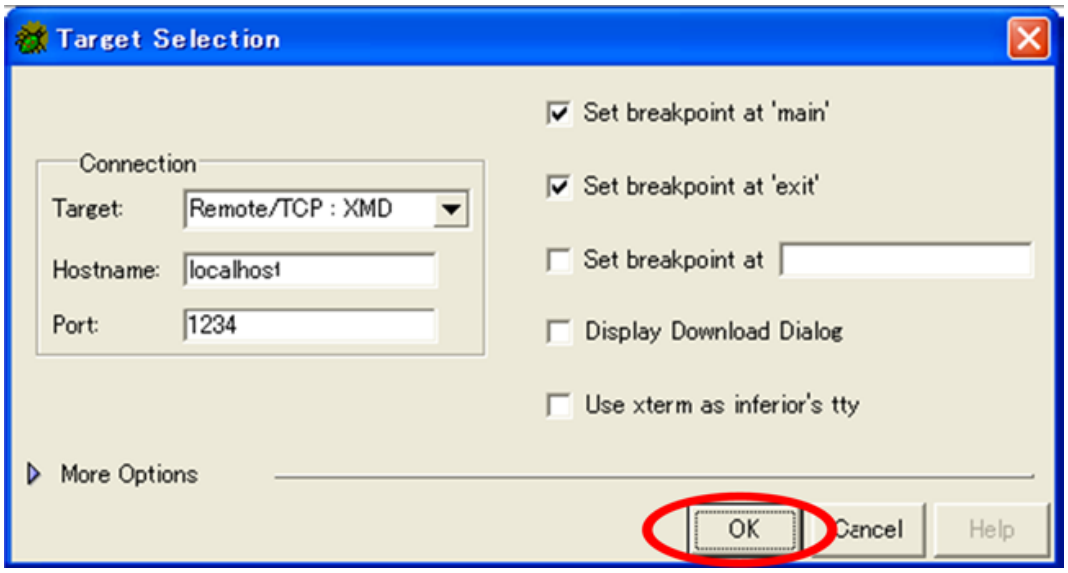


図 11.89. デバッグ設定

以下の画面が立ち上がります。[Run]→[Run]をクリックして下さい。main 関数で Break します。LED_GPIO(LED_OFF)が緑色にハイライトされるとおもいます。

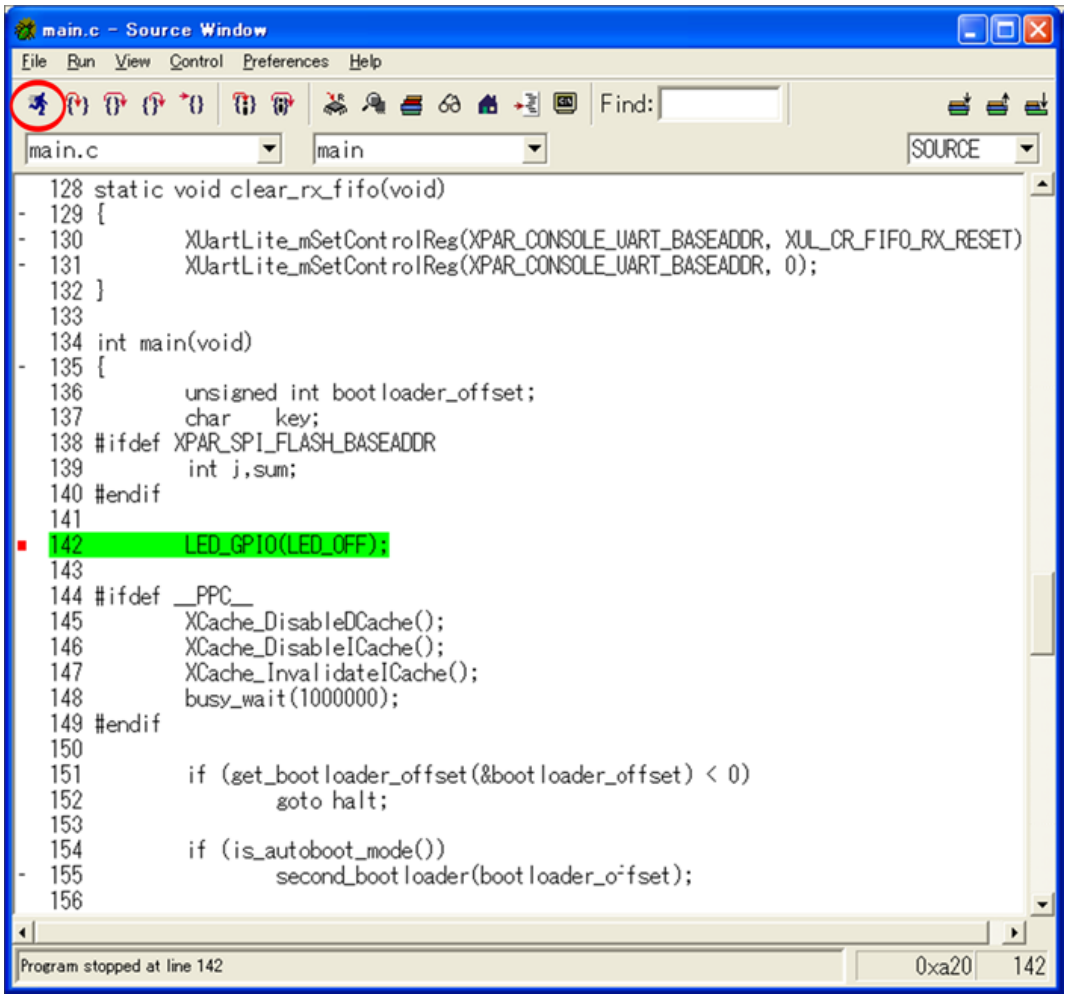


図 11.90. main で Break

11.11.5. ステップ実行で割り込みの流れをみる

[View]→[Breakpoints]をクリックして下さい。現在の Breakpoint が表示されます。"main"と"exit"に Breakpoint が設定されています。Breakpoint は割り込みベクタだけでいいので、[Global]→[Remove All]をクリックし、Breakpoint を消去してください。

MicroBlaze の割り込みベクタは 0x10、PowerPC の割り込みベクタは 0xFFFF0500 です。ここに Breakpoint を設定します。

XMD で以下のコマンドを実行してください。

例 11.16. Breakpoint 設定(SZ010, SZ030, SZ130 の場合)

```
XMD% bps 0x10 hw
```

例 11.17. Breakpoint 設定(SZ310, SZ410 の場合)

```
XMD% bps 0xFFFF0500 hw
```

XMD コマンド

コマンド	使用例	説明
bps <address/function> <hw/sw>	bps 0x10 sw bps main hw	address または function の開始部分にハードウェアまたはソフトウェアブレークポイントを設定
bpr <address/function/all>	bpr 0x10 bpr main bpr all	ブレークポイントを削除
bpl	bpl	現在のブレークポイントを表示

[Control]→[Continue]をクリックしてください。割り込みハンドラ 0x10(PowerPC の場合は 0xFFFF0500)で Break します。ここから[Step]で順に動きを見ます。どのように割り込みが入るのかが分かります。

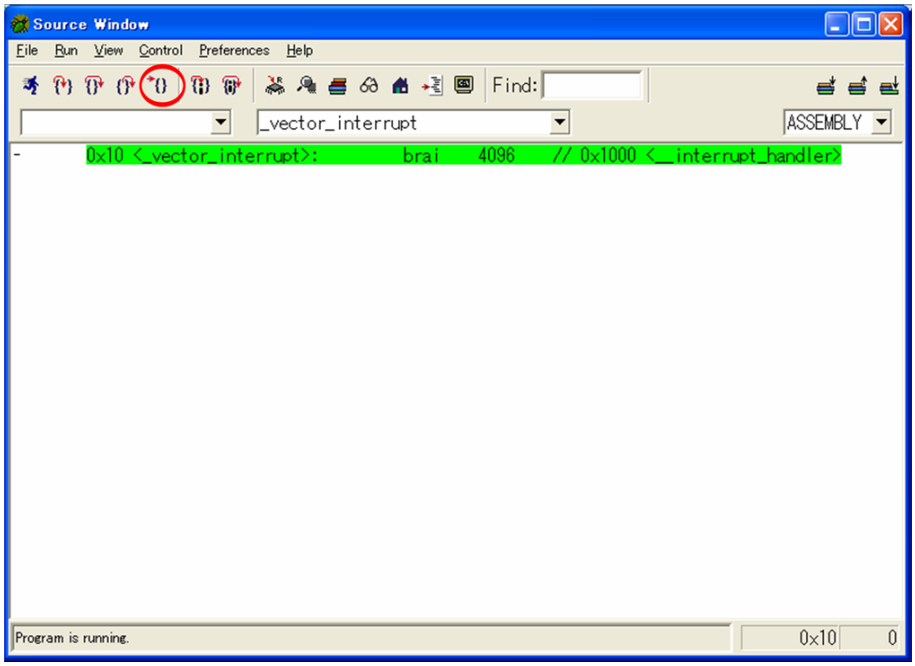


図 11.91. 0x10 で Break

MicroBlaze の場合、割込みが入ると 割込みベクタ 0x10 にジャンプします。

PowerPC の場合、割り込みが入ると割り込みベクタ 0xFFFF0500 にジャンプします。

これらのアドレスは、FPGA 内部の BRAM の領域です。MicroBlaze の場合、割込みハンドラ __interrupt_handler( )へのジャンプ命令が記述されています。

PowerPC の場合、XIntc_DeviceInterruptHandler( )へのジャンプ命令が記述されています。

[Control]→[Step]を押してください。

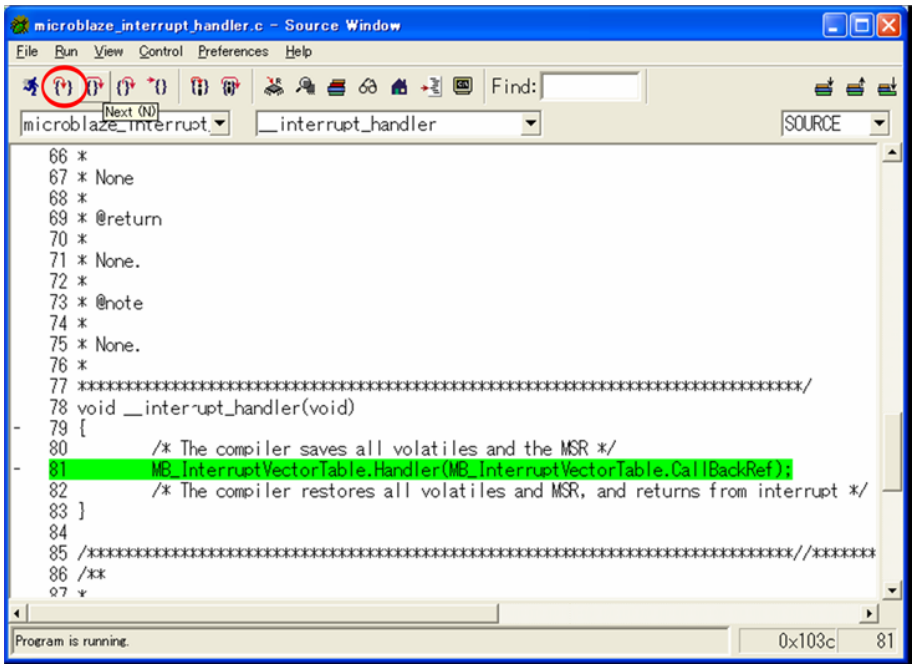
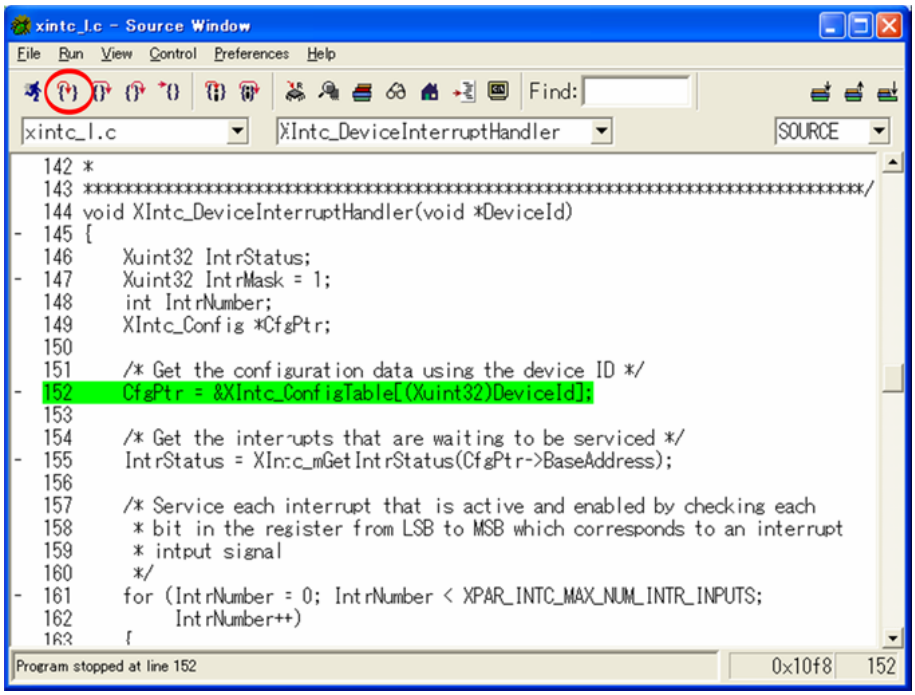


図 11.92. _interrupt_handler( )で Break

__interrupt_handler( )には、PORT INTERRUPT に接続されているコア(SUZAKU では OPB-INTC)の割込みハンドラ(XIntc_DeviceInterruptHandler( ))へジャンプする命令が記述されています。

何度か[Step]を押してください。



XIntc_DeviceInterruptHandler( )には、割り込みコントローラに接続されているコア達の中から、実際に割込みを発生させたコアを見つけ、そのコアの割込みハンドラへのジャンプする命令が記述されています。

OPB-SIL00 の場合"Default Handler"ではなく、timer_interrupt_handler を使用しています。

図 11.93. XIntc_DeviceInterruptHandler( )で Break

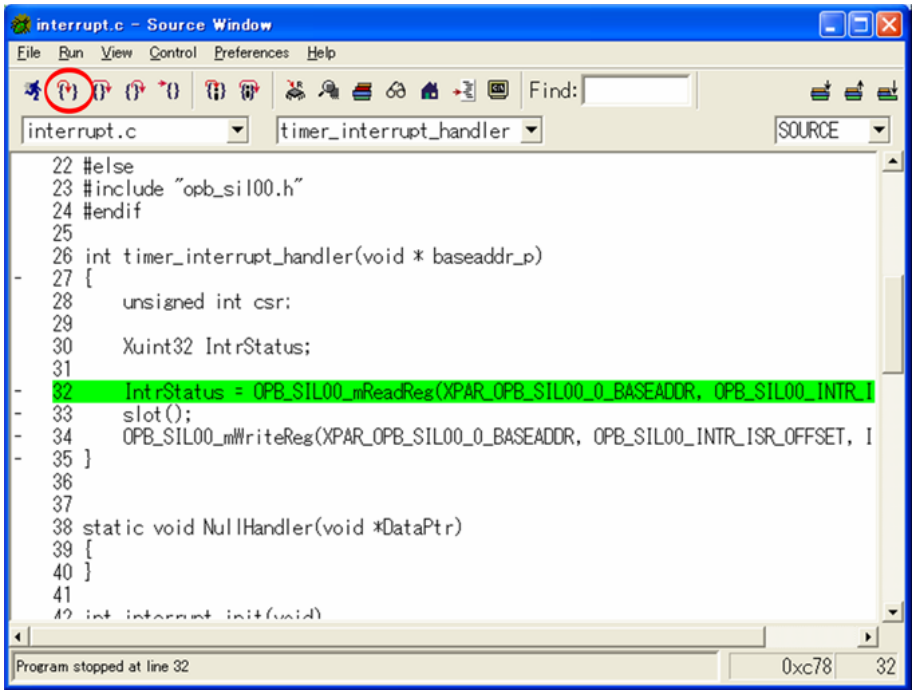


図 11.94. timer_interrupt_handler( )で Break

11.11.6. slot.c の動作を確認してみる

`slot.c` にジャンプしてきました。

[View]→[Loval Variables]をクリックして下さい。ローカル変数の一覧が表示されます。

[View]→[Stack]をクリックして下さい。現在スタックしている関数の一覧が表示されます。

ステップ実行やその他メニューの[View]で開くことのできる各種情報を元に、ソースコードと合わせてプログラムの流れやローカル変数を参照し、スロットの動作を確認してください。

かなりのステップ実行をしなくてはいけませんが、7 セグメント LED に 3→2→1 と数字が表示される様子を見ることが出来ます。また、`slot.c` の一通りの動作が終わると、スタックにしたがって、関数が戻っていく様子を見ることが出来ます。

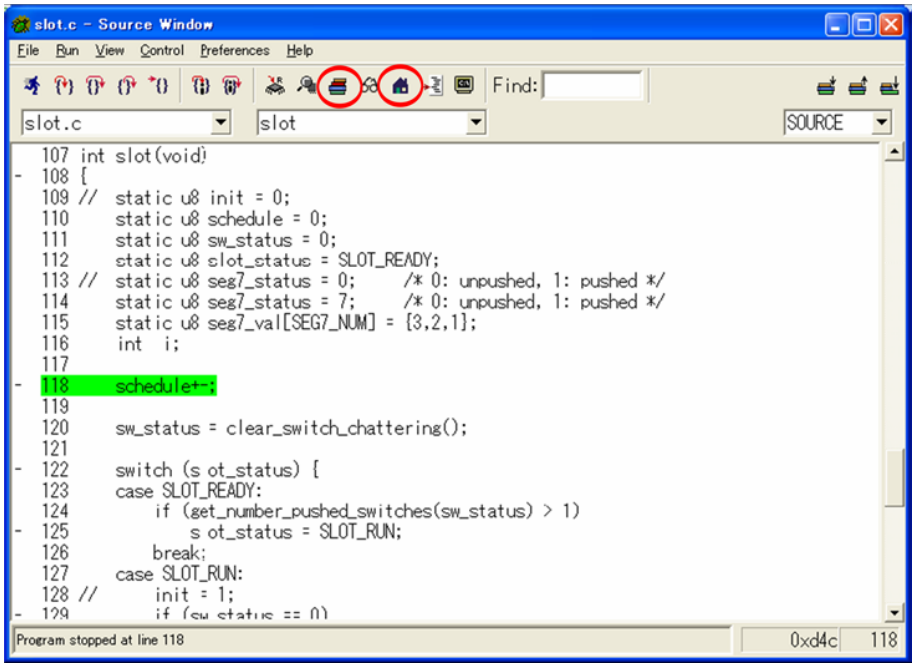


図 11.95. slot( )で Break

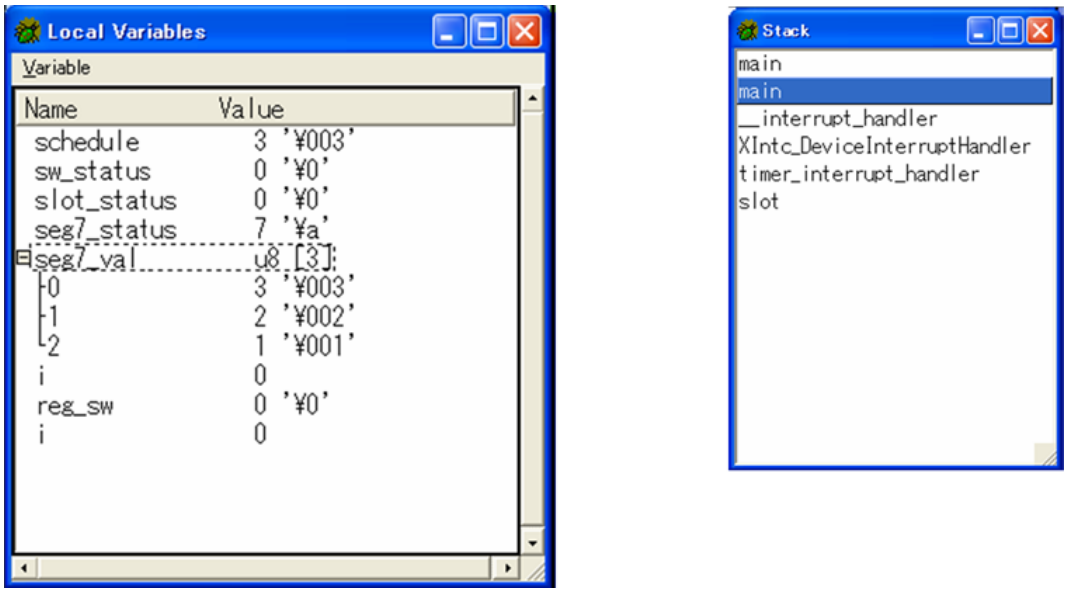


図 11.96. ローカル変数やスタックの一覧

12.こんなこともやってみよう

最後にこんなこともできますというのを紹介します。[1] SZ010 SZ030 SZ130 SZ310

12.1. EDK を ISE のサブモジュールとして読み込む

ここでは EDK を ISE のサブモジュールとして読み込む方法を説明します。

SUZAKU は EDK だけで構築されています。EDK だけで作業しても良いのですが、EDK だけで作業すると、自分で書いたロジックを追加したい時に、EDK に読み込めるようにしなければいけません。自分のロジックのほかにロジック追加用の設定ファイル(mpd ファイル、pao ファイル)を書かなければいけなかったり、Xilinx の命名規則にのっとっていなければならなかったりします。EDK に読み込めるようにしておけば、再利用しやすいというメリットはあるのですが、少々面倒です。

EDK と ISE は連携しており、EDK で構築生成させたネットリストを ISE でサブモジュールとして読み込ませることができます。この機能を使えば、EDK で GPIO や簡単な OPB インターフェースだけを用意しておき、EDK の External 設定することで、ISE でこれらの信号をサブモジュールの入出力として取り扱えるようになり、ISE で作りこんだ自分のロジックへ容易に組み込むことができます。

さらに、自分のロジックのみに変更があった場合、開発フローの ISE のみを実行するだけですむので、コンパイル時間を半減させることができます。また、ISE は、配置配線ツールや制約ツール、タイミング解析ツールなど GUI で設定することができ分かりやすいです。

ISE と EDK を連携させるには、

1. EDK から ISE のプロジェクトを生成させる

   (EDK で Export to Project Navigator を実行する)

2. ISE に EDK を取り込む

の 2 通りがあります。1 の方法は今後サポートされなくなる予定なので、ここでは、2.の"ISE に EDK を取り込む方法"を紹介します。

2006 年 4 月以前の SUZAKU は EDK を ISE のサブモジュールとして読み込んでプロジェクトを作成していました。この 2 の方法により、EDK だけで構築されている SUZAKU のプロジェクトを、昔の EDK を ISE のサブモジュールとして読み込んだプロジェクトにつくりかえることができます。

12.1.1. EDK で作業

例として SUZAKU のデフォルトの EDK のプロジェクトを ISE に取り込みます。

付属 CD-ROM の`"\suzaku\fpga_proj\x.x\sz***\sz***-yyyymmdd.zip"`をハードディスクに展開してください。

適当な名前のフォルダを作り、フォルダの下に、展開した EDK フォルダをコピーしてください。ここでは`"C:\suzaku\suzaku-ise\sz***-yyyymmdd"`として作業を進めます。

[1] SZ410 に関しては現在動作確認中につき少々お待ちください。

"C:\suzaku\suzaku-ise\sz***-yyyymmdd"の中の"xps_proj.xmp"をダブルクリックして開いてください。

Xilinx Platform Studio が起動し、SUZAKU のデフォルトが開きます。

[Hardware]→[Generate Netlist]をクリックし、ネットリストを作成してください。

ネットリストを作成すると、"C:\suzaku\suzaku-ise\sz***-yyyymmdd\hdl"に

xps_proj_stub.vhd というファイルが出来上がります。

12.1.2. EDK から ISE へ移行

この xps_proj_stub.vhd を"C:\suzaku\suzaku-ise"にコピーしてください。この際、この名前のままだと少し分かりづらいので、ファイルの名前を top.vhd に変更してください。(もちろん変えなくても良いです。。)

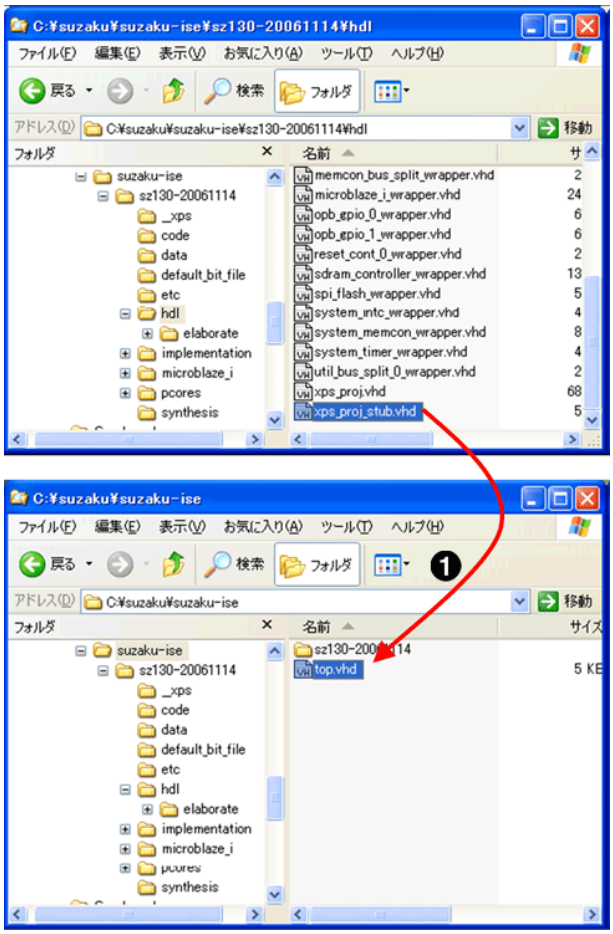


図 12.1. xps_proj_stub.vhd をコピー

❶　xps_proj_stub.vhd をコピーして名前を top.vhd に変更

12.1.3. ISE で作業

Project Navigator を起動してください。[File]→[New Project]をクリックしてください。

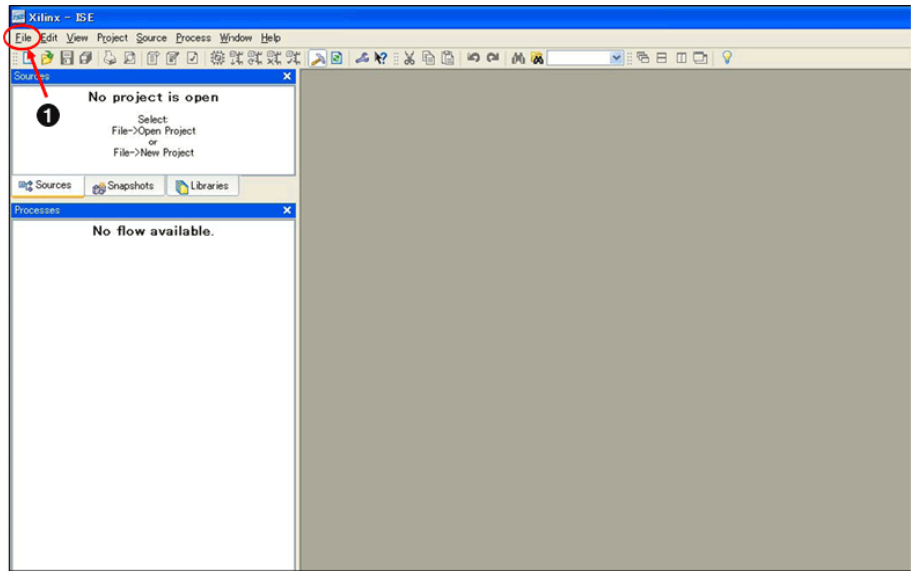


図 12.2. Project Navigator 起動

❶　[File]→[New Project]をクリック

New Project Wizard が表示されます。[Project Location]の[…]をクリックし、プロジェクトのディレクトリパスを指定します。ここでは `C:\suzaku\suzaku-ise` となります。[Project Name]に プロジェクト名を入力します。 top と入力し、[Top-Level Source Type]が[HDL]となっていることを確認し、[Next]をクリックしてください。

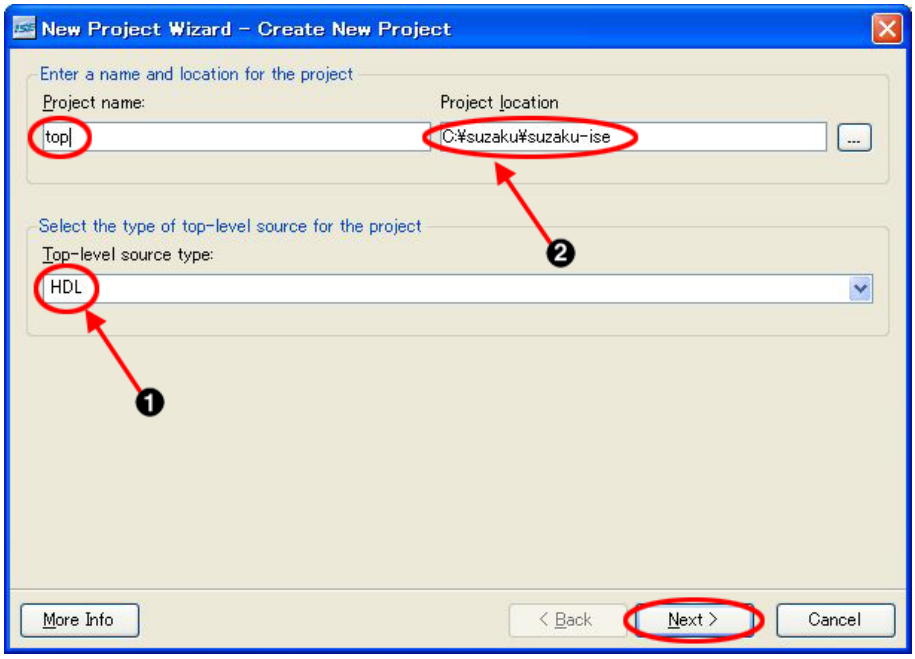


図 12.3. プロジェクトの新規作成

❶　確認

❷　ディレクトリパスを指定

SUZAKU に実装されている FPGA デバイスを選択します。お使いの SUZAKU の型式の設定にし、[Next]をクリックしてください。

型式	SZ010	SZ030	SZ130	SZ310	SZ410
Product Category	ALL				
Family	Spartan3		Spartan3E	Virtex2P	Virtex4
Device	XC3S400	XC3S1000	XC3S1200E	XC2VP4	XC4VFX12
Package	FT256		FG320	FG256	SF363
Speed	-4			-5	-10
Synthesis Tool	XST(VHDL/Verilog)				
Simulator	ISE Simulator(VHDL/Verilog)				

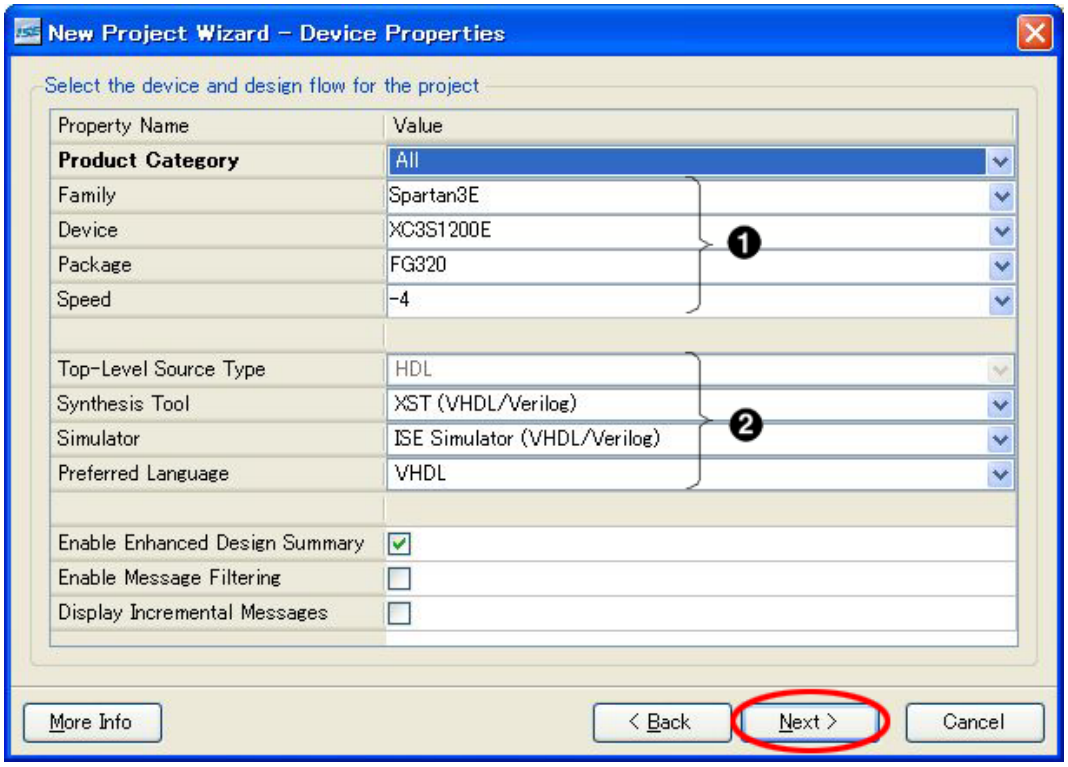


図 12.4. デバイスの選択(SZ130 の場合)

❶　デバイスの選択

❷　デザインフローの選択

後は設定の変更をしないので、[Next]、[Finish]をクリックし、ウィザードを終了してください。

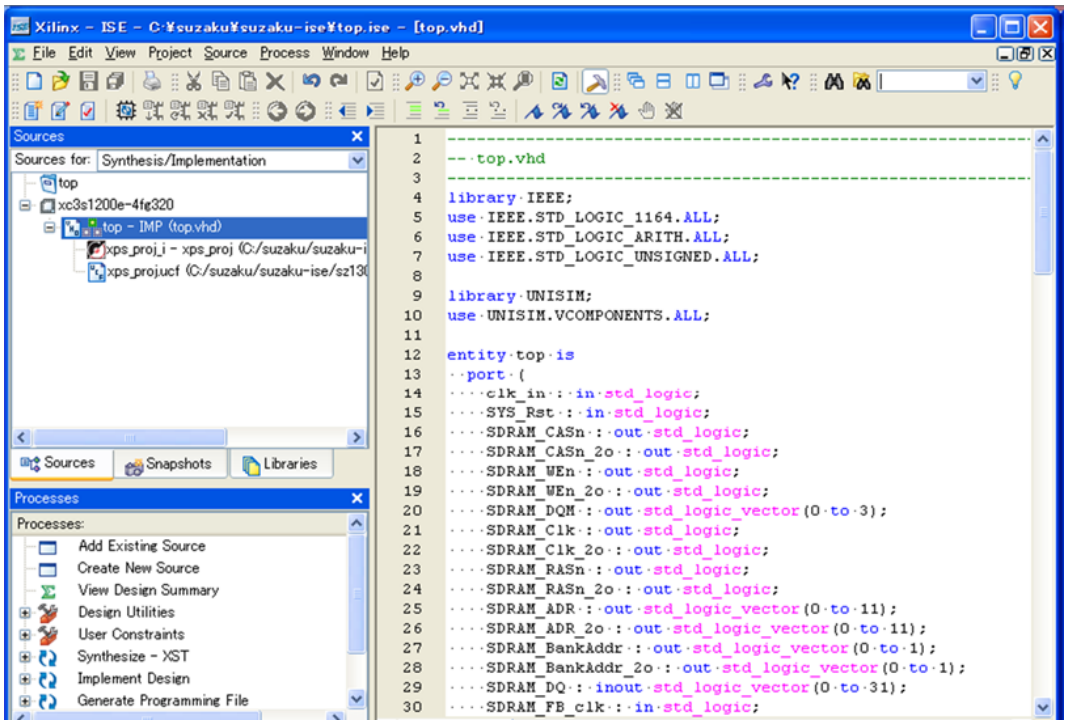
`top.vhd` と `xps_proj.xmp` と `xps_proj.ucf` をプロジェクトに追加します。[Project]→[Add Source]をクリックしてファイルを追加してください。`xps_proj.xmp` は`"C:\suzaku\suzaku-ise\sz***- yyyymmdd"`の下に `xps_proj.ucf` は

`"C:\suzaku\suzaku-ise\sz***- yyyymmdd \data"`の下にあります。

top -IMP(top.vhd)をダブルクリックして開いてください。ライブラリの追加を行います。また、少し分かりづらいので、STRUCTURE となっているアーキテクチャ名を IMP に変更し(2 箇所)、

xps_proj_stub となっているエンティティ名を top に変更します(3 箇所)。（ライブラリの追加は必須です。名前の変更は必須ではありません。。）

変更ができたら[File]→[Save]をクリックし、保存してください。




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;      -- ライブラリ追加
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

entity top is
  port (
    --中略
  );
end top;

architecture IMP of top is
  component xps_proj is
  port (
    --中略
  );
  end component;
begin
  xps_proj_i : xps_proj
    port map (
    --中略
    );
end architecture IMP;
```

図 12.5. ソースコード入力

Generate Programming File をダブルクリックして下さい。ソフトウェアを含まない bit ファイルが生成されます。

Update Bitstream with Processor Data をダブルクリックして下さい。ハードウェアでつくった bit ファイルの中にアプリケーションを書き込みます。top_download.bit が出来上がります。

SZ310 のプロジェクトを ISE 8.1i で作っている場合、ハードウェアでつくった bit ファイルの中にアプリケーションがうまく書き込まれず、top_download.bit がきちんと生成されないことがあります。コマンドプロンプト等を立ち上げ、`"C:\suzaku\suzaku-ise"`に移動し、以下のコマンドで新たに bit ファイルを生成してください。

```
data2mem -bm sz310-yyyymmdd\implementation\xps_proj_bd.bmm
 -bt top.bit
 -bd sz310- yyyymmdd \ppc405_i\code\executable.elf
 tag bram -o b top_new.bit
```

iMPACT を立ち上げ、`top_download.bit(top_new.bit)`を書き込んでください。SUZAKU のデフォルトが書き込まれます。

EDK で External の信号を追加削除した場合は、ネットリストを作成し直し、新たに生成された

xps_proj_stub.vhd を参考に、`top.vhd` の xps_proj の component 宣言および、そのインスタンスに変更を加えてください。

ISE はコンパイル前に EDK に変更がないか確認をします。EDK で mhs ファイルや mss ファイル、ソフトウェアを変更した場合、自動的にバックエンドで EDK を動作させコンパイルを実行してくれます。

12.2. IP コア(ハード版)

先ほどソフトウェアで実現したスロットマシンの機能をハードウェアに置き換え、ソフトの負担を減らすことができます。

実はこちらの方法のほうが SUZAKU らしいやり方といえます。slot.vhd の中身の説明はしませんので、各自見て考えてみてください。ソフト版と違うのはカウンタのみで、他はほぼ同じ作りになっています。

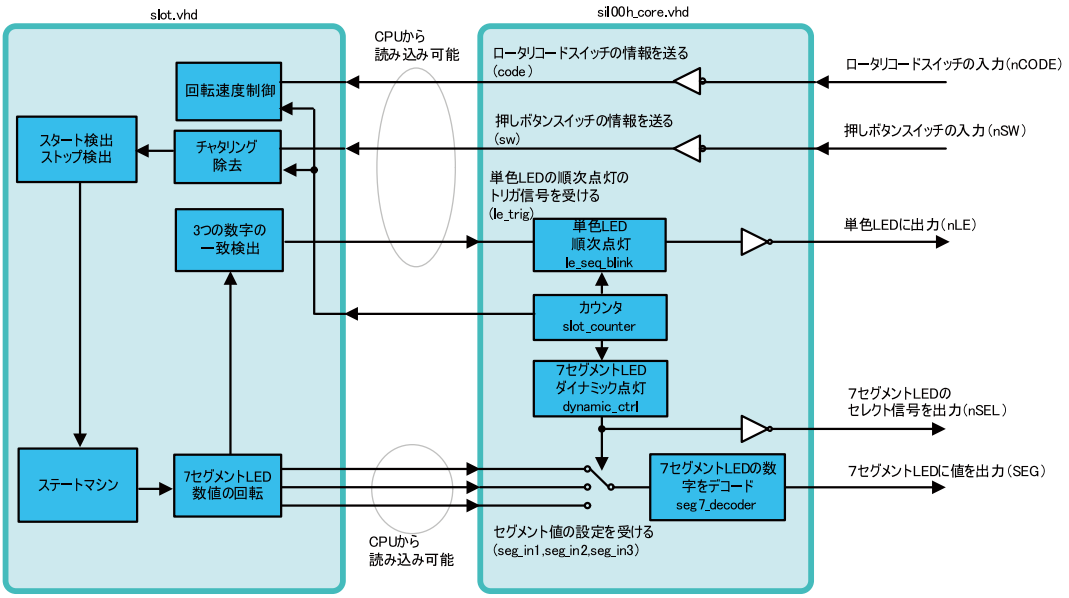


図 12.6. Create and Import Peripheral Wizard の起動のさせ方

`"C\suzaku\sz***- yyyymmdd"`をコピーしてその場にペーストし、名前を変更してください。

ここでは`"C\suzaku\sz***-h- yyyymmdd"`として作業を進めます。

SZ010, SZ030, SZ130, SZ310 の場合、付属 CD-ROM の

`"suzaku-starter-kit\fpga\opb_sil00h_vx_xx_x.zip"`をハードディスクに展開してください。SZ410 の場合、付属 CD-ROM の `"suzaku-starter-kit\fpga\xps_sil00h_vx_xx_x.zip"`をハードディスクに展開してください。

展開後のフォルダ"opb_sil00h_vx_xx_x | xps_sil00h_vx_xx_x"を

"C\suzaku\sz***-h- yyyymmdd \pcores"にコピーしてください。

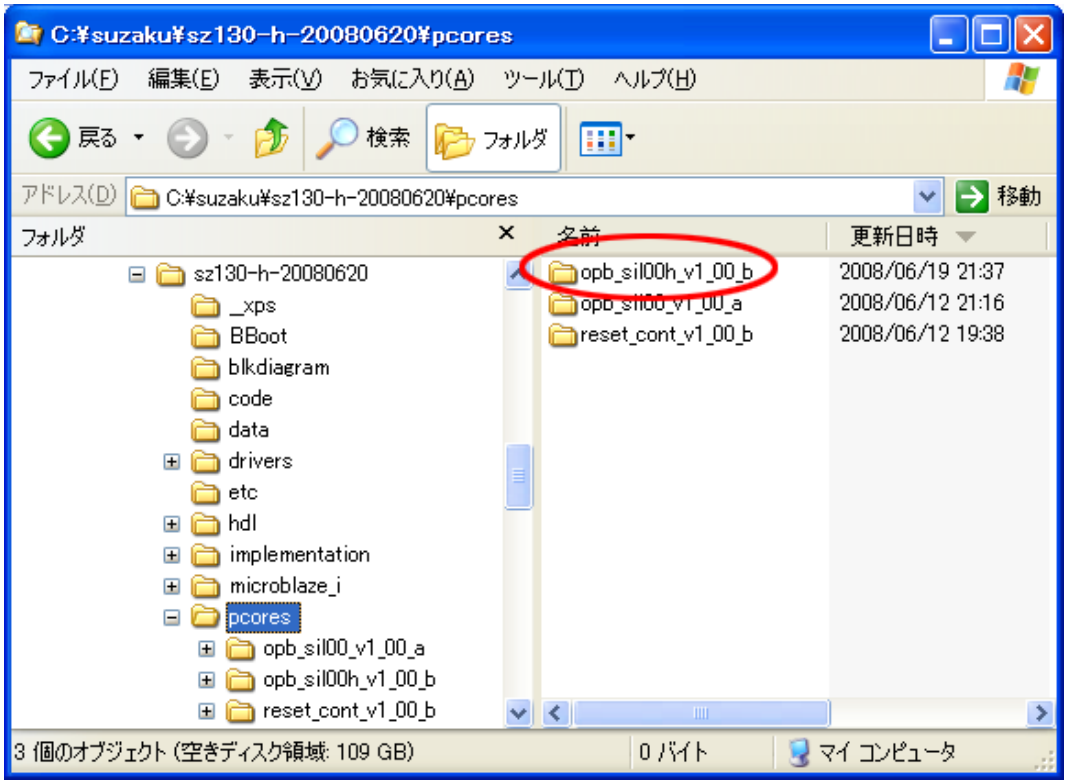


図 12.7. IP コア(ハード版)追加

"C:\suzaku\szz***-h- yyyymmdd"の中の"xps_proj.xmp"を開いてください。

IP Catalog の Project Repository に opb_sil00h があるのを確認してください。

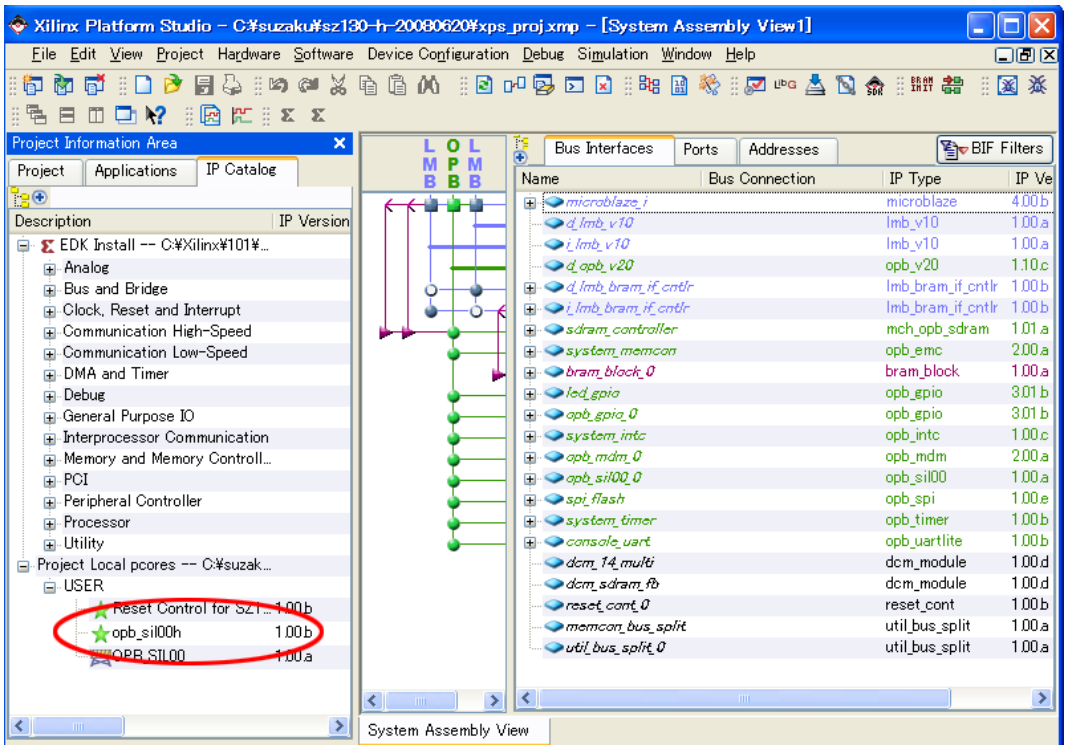


図 12.8. IP コア(ハード版)追加確認

自作 IP コア(ソフト版)opb_sil00 を IP コア(ハード版)opb_sil00h に置き換えます。opb_sil00h は割り込みを使用していないので、割り込みに関する記述を削除します。

まず、ハードウェアの変更をします。

Project の MSS File: xps_proj.mss をダブルクリックして開いてください。opb_sil00 のドライバの記述をしているところを探して削除し、保存してください。opb_sil00h ではドライバを使用しません。

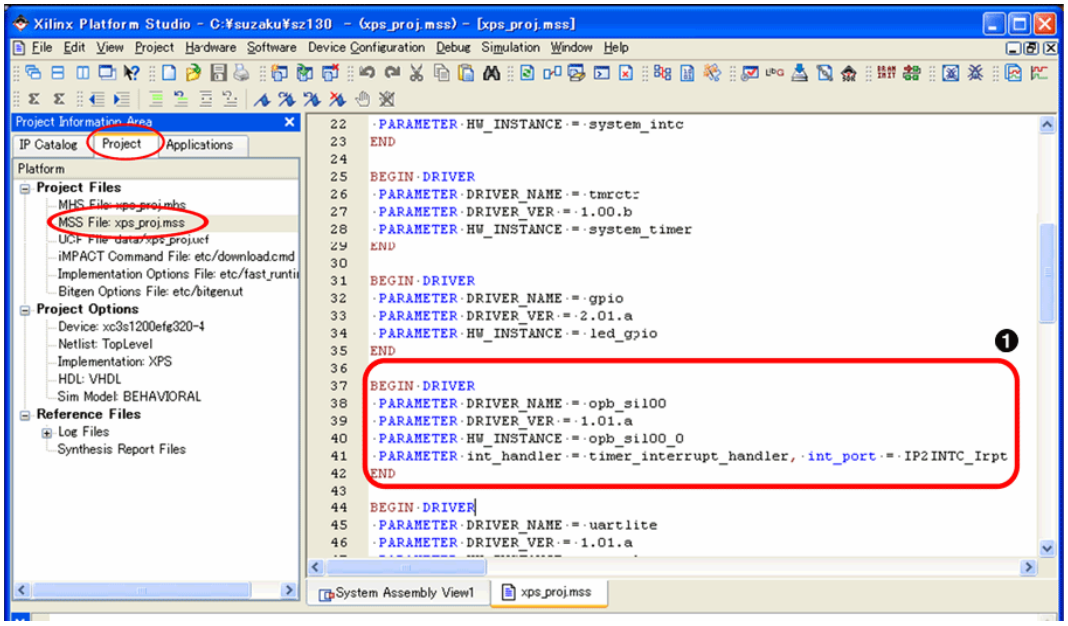


図 12.9. MSS File 変更

❶　削除

Project の MHS File: xps_proj.mhs をダブルクリックして開いてください。

opb_sil00 | xps_sil00 を記述しているところを探してこれを opb_sil00h | xps_sil00h の記述に変更します。その際、HW_VER があっているかも確認してください。

opb_intc | xps_intc を記述しているところを探して、割り込みの記述を削除します。

変更が終わったら保存してください。

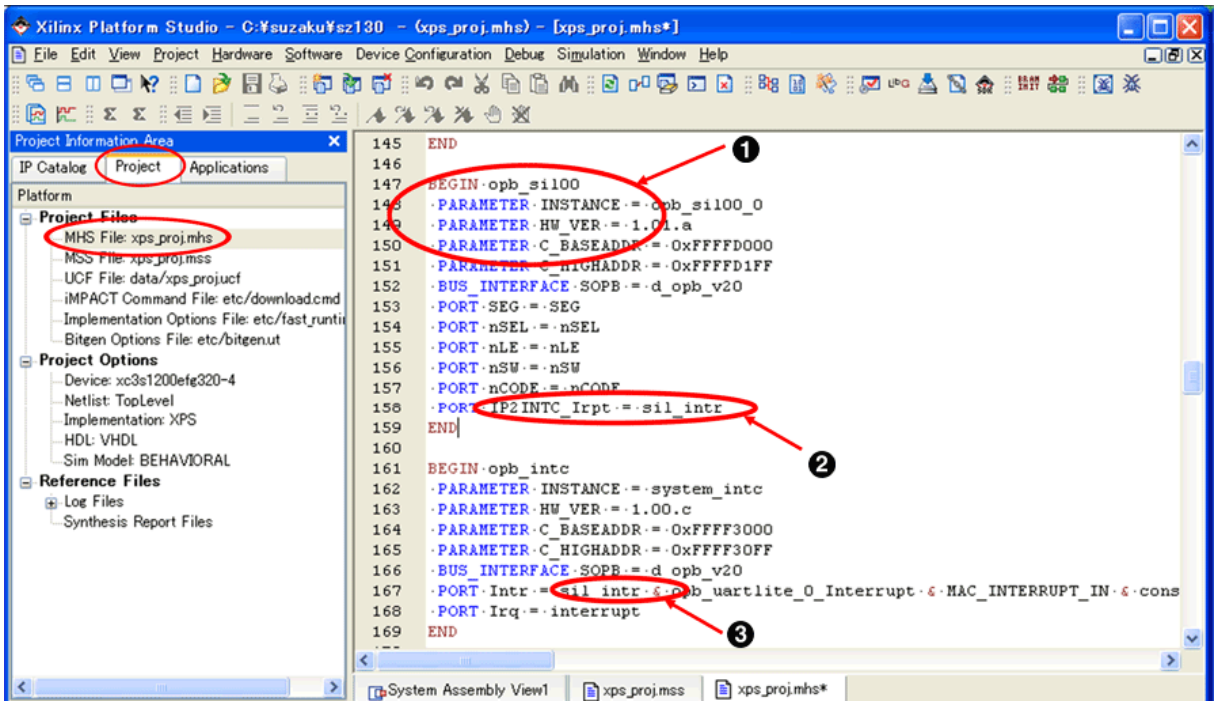


図 12.10. MHS File 変更

❶　Z010,SZ030,SZ130,SZ310 の場合

BEGIN opb_sil00h に変更

SZ410 の場合

BEGIN xps_sil00h に変更

HW_VER があっているか確認

❷　PORT IP2INTC_Irpt = sil_intr を削除

❸　sil_intr &を削除

以上で IP コアが置き換わりました。

IP Type が opb_sil00h | xps_sil00h に変更されます。

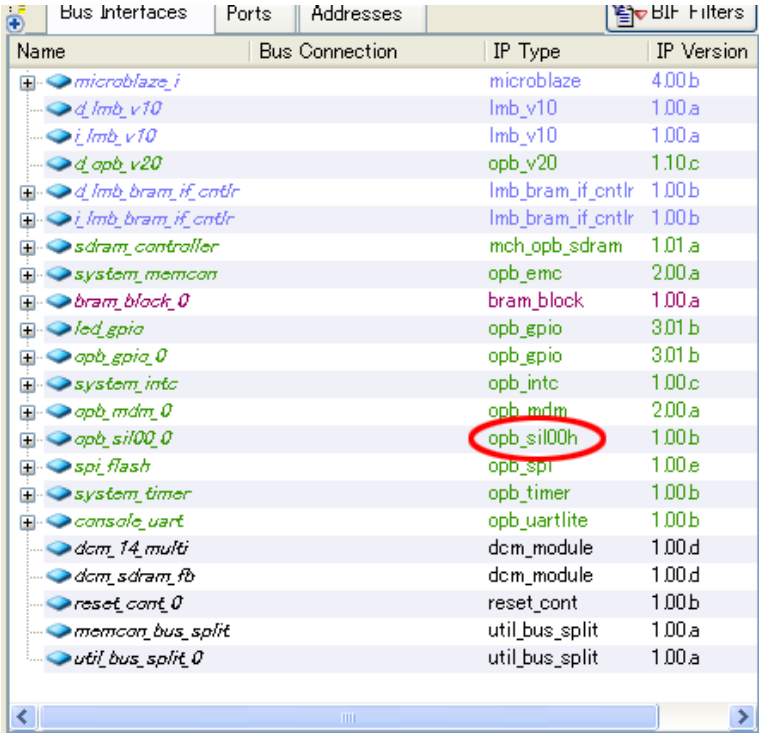


図 12.11. IP コア(ハード版)に置き換え

次にソフトウェアから割り込みの設定、記述を削除していきます。

Applications の Sources から `interrupt.c`、`slot.c`、Headers から `interrupt.h`、`slot.h` を削除してください。これらのファイルは使用しません。

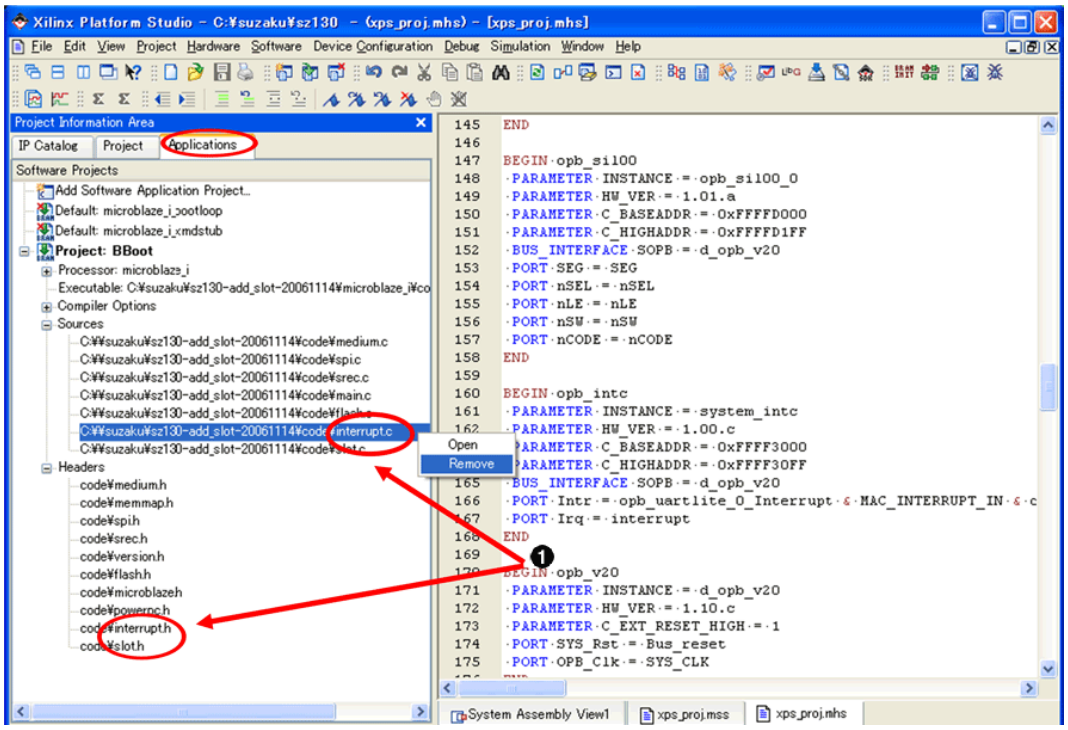


図 12.12. 不要なファイルの削除

❶　interrupt.c , slot.c , interrupt.h , slot.h を削除

これで変更は終了です。コンフィギュレーションしてください。数字の回転が少し速いですが、ソフト版とほぼ同じスロットマシンが動きます。

12.3. CGI で 7 セグメント LED をコントロール

自分で作ったスロットマシンの IP コアを CGI でコントロールします。`"C:\suzaku\sz***-yyyymmdd \implementation"`の中にある `download.bit` をつかって フラッシュメモリを書き換えてください。

フラッシュメモリの中に入っている Linux では最初から CGI が動作しています。(フラッシュメモリの中の Linux を書き換えてしまっている場合は、フラッシュメモリの image を書き直して下さい。image は付属 CD-ROM の`"\suzaku-starter-kit\image"`の中の `image-sz***-sil.bin` を使ってください。)

シリアル通信ソフトウェアを起動後、SUZAKU スターターキットの JP1、JP2 をオープンにして電源を投入してください。Linux が起動するので、ネットワークの設定をしてください。

IP アドレスを確認し、お使いのブラウザで`"http://IP アドレス/7seg-led-control.cgi"`にアクセスしてください。スロットマシンの 7 セグメント LED の回路がブラウザから制御できます。

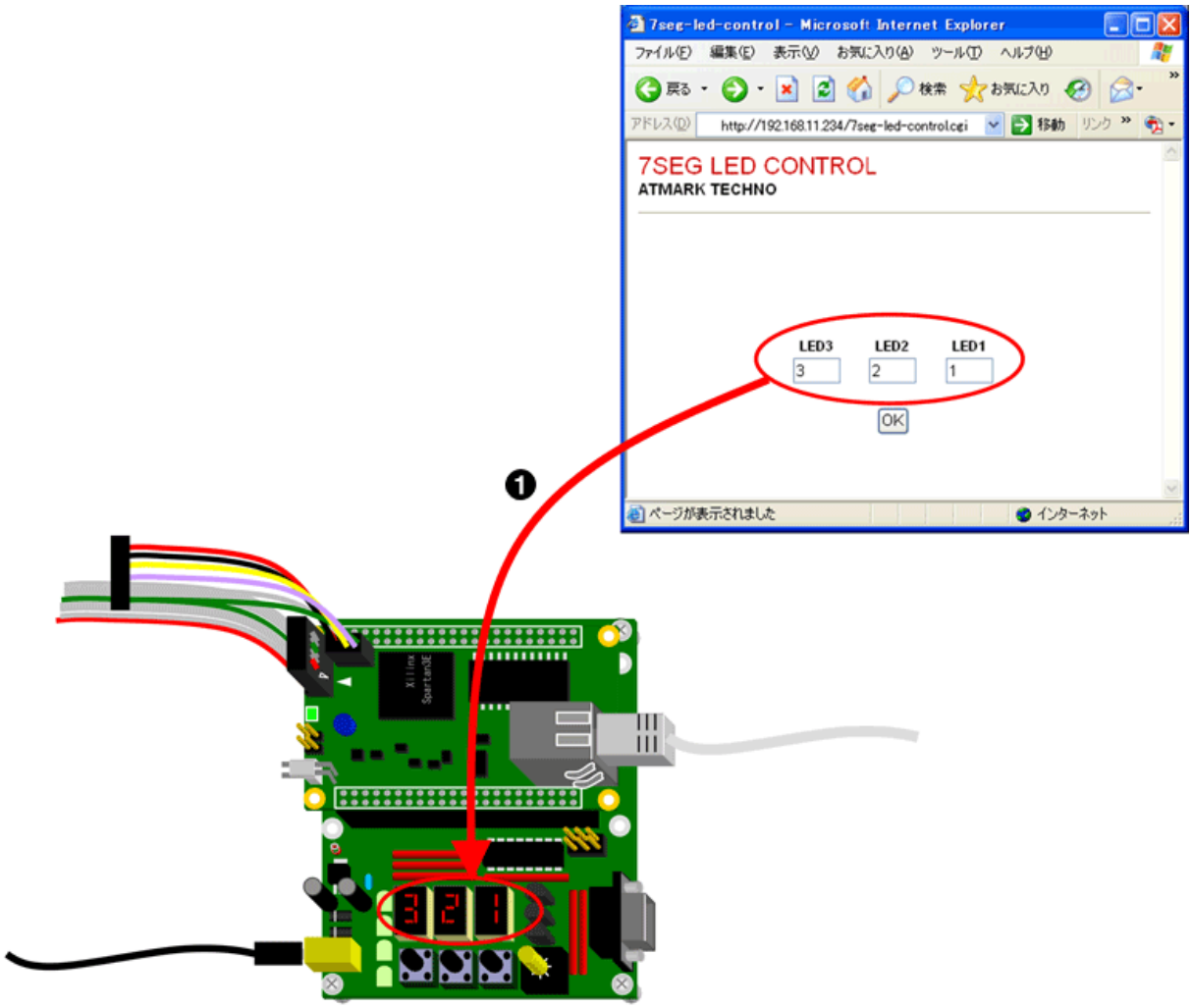


図 12.13. 自作のコアをコントロール

❶　1~F(16 進数)の数字を設定して[OK]をクリックすると、7 セグメント LED に設定した数字が表示される

これは、以下のソースコードで CGI を作成することにより実現しています。以下のソースコードだけでは CGI を作成することはできませんので、ご注意ください。

CGI の作成方法やコンパイル方法、フラッシュメモリに書き込むためのデータの作成方法等ついては、"SUZAKU　スターターキットガイド(Linux 開発編)"、"SUZAKU　ソフトウェアマニュアル"、"uClinux-dist 開発者ガイド"を参照してください。

12.3.1. CGI で 7 セグメント LED をコントロール(7seg-led-control.c)

例 12.1. CGI で 7 セグメント LED をコントロール(7seg-led-control.c)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define PROGRAM_NAME             "7seg-led-control"
#define CGI_PATH PROGRAM_NAME    ".cgi"

#define DEV_NAME                 "/dev/sil7segc"

#define FORM_OK_BUTTON           "ok_button"
#define FORM_LED1_TEXT_BOX       "led1"
#define FORM_LED2_TEXT_BOX       "led2"
#define FORM_LED3_TEXT_BOX       "led3"

static void print_content_type(void)
{
  printf("Content-Type:text/html\n\n");
}

static void print_style_sheet(void)
{
  printf("<style type=\"text/css\">\n\n");

  printf("body {\n");
  printf("margin: 0 0 0 0;\n");
  printf("padding: 10px 10px 10px 10px;\n");
  printf("font-family: Arial, sans-serif;\n");
  printf("background: #ffffff;\n");
  printf("}\n\n");

  printf("h1 {\n");
  printf("margin: 0 0 0 0;\n");
  printf("padding: 0 0 0 0;\n");
  printf("color: #cc0000;\n");
  printf("font-weight: normal;\n");
  printf("}\n\n");

  printf("h2 {\n");
  printf("margin: 0 0 0 0;\n");
  printf("padding: 0 0 0 0;\n");
  printf("font-size: 14px;\n");
  printf("}\n\n");
```

```
  printf("hr {\n");
  printf("height: 1px;\n");
  printf("background-color: #999999;\n");
  printf("border: none;\n");
  printf("margin: 5px 0 70px 0;\n");
  printf("}\n\n");

  printf(".leds {\n");
  printf("font-size: 12px;\n");
  printf("font-weight: bold;\n");
  printf("line-height: 20px;\n");
  printf("}\n\n");

  printf("</style>\n\n");
}

static void print_html_head(void)
{

                      \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">
\n");
  printf("<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"ja\" xml:lang=
\"ja\">\n\n");

  printf("<head>\n\n");

  printf("<meta http-equiv=\"content-type\" content=\"text/html; charset=utf-8\"/>
\n\n");
  printf("<title>%s</title>\n\n", PROGRAM_NAME);

  print_style_sheet();

  printf("</head>\n\n");

  printf("<body>\n\n");
}

static void print_html_tail(void)
{
  printf("</body>\n\n");
  printf("</html>\n");
}

static void display_page(int fd)
{
  unsigned char leds[3];

  read(fd, leds, 3);

  print_content_type();

  print_html_head();

  printf("<h1>7SEG LED CONTROL</h1>\n");
  printf("<h2>ATMARK TECHNO</h2>\n\n");

  printf("<hr />\n\n");
```

```
  printf("<form action=\"%s\" method=\"get\">\n\n", CGI_PATH);

  printf("<table border=\"0\" cellpadding=\"10\" cellspacing=\"0\" width=\"200px\"
                                                        align=\"center\" class=\"leds
\">\n");

  printf("<tr>\n");

  printf("<td align=\"center\">");

  printf("LED3<br />\n");
  printf("<input type=\"text\" name=\"%s\" value=\"%x\" size=\"1\" maxlength=\"1
                                                         \"/>\n",FORM_LED3_TEXT_BOX,
leds[2]);

  printf("</td>\n<td align=\"center\">");

  printf("LED2<br />\n");
  printf("<input type=\"text\" name=\"%s\" value=\"%x\" size=\"1\" maxlength=\"1
                                                        \" />\n",FORM_LED2_TEXT_BOX,
leds[1]);

  printf("</td>\n<td align=\"center\">");

  printf("LED1<br />\n");
  printf("<input type=\"text\" name=\"%s\" value=\"%x\" size=\"1\" maxlength=\"1
                                                        \" />\n",FORM_LED1_TEXT_BOX,
leds[0]);

  printf("</td>\n");

  printf("</tr><tr>\n");

  printf("<td colspan=\"3\" align=\"center\">\n");

  printf("<input type=\"submit\" value=\"OK\" name=\"%s\" />\n", FORM_OK_BUTTON);

  printf("</td>\n");

  printf("</tr>\n");

  printf("</table>\n\n");

  printf("</form>\n\n");

  print_html_tail();
}

static unsigned int get_query_pair_hex_value(char *query, char *query_pair_name)
{
  char *pair_start, *pair_value;
  unsigned int hex_value = 0;

  pair_start = strstr(query, query_pair_name);
  if (pair_start) {
    pair_value = strchr(pair_start, '=') + 1;
    if (pair_value) {
```

```
      sscanf(pair_value, "%#37;x", &hex_value);
    }
  }
  return hex_value;
}

static void handle_query(int fd)
{
  char *query;
  unsigned char leds[3];

  query = getenv("QUERY_STRING");
  if (!query) {
    return;
  }

  if (!strstr(query, FORM_OK_BUTTON)) {
    return;
  }
  leds[0] = (unsigned char) get_query_pair_hex_value(query, FORM_LED1_TEXT_BOX);
  leds[1] = (unsigned char) get_query_pair_hex_value(query, FORM_LED2_TEXT_BOX);
  leds[2] = (unsigned char) get_query_pair_hex_value(query, FORM_LED3_TEXT_BOX);
  write(fd, leds, 3);
}

int main(int argc, char *argv[])
{
  int fd;
  fd = open(DEV_NAME, O_RDWR);
  handle_query(fd);
  display_page(fd);
  close(fd);

  exit(EXIT_SUCCESS);
}
```

12.4. SDK を使ってデバッグ

Eclipse ベースの SDK(Software Develpment Kit)でデバッグします。「11.11. ソフトウェアのデバッグ」と基本的に出来ることは同じです。

「11.1. スロットマシンのコアの構成 (OPB)」の作業まで行ったプロジェクトを開き、[Software]→[Launch Platform Studio SDK]をクリックして下さい。

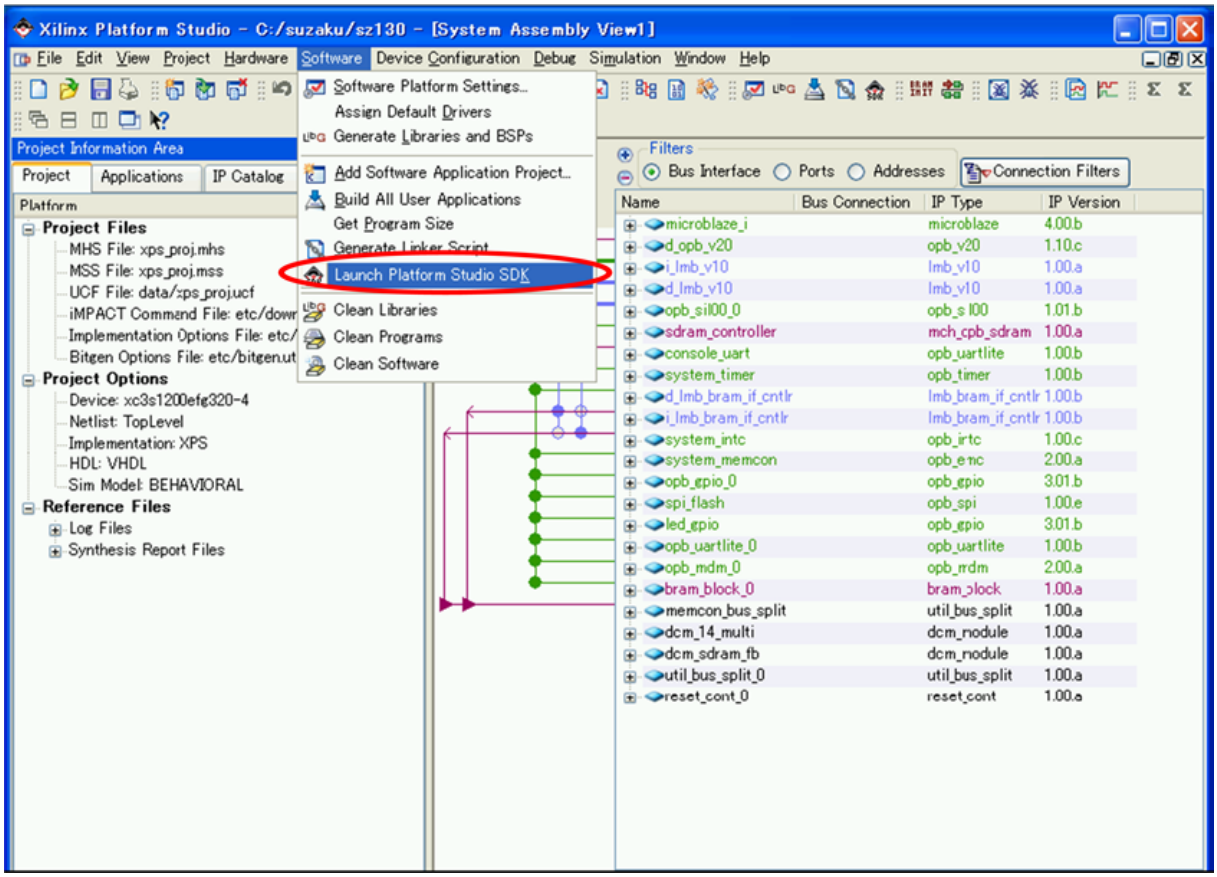


図 12.14. SDK 起動

以下のウィザードが立ち上がるので[Import XPS Application Projects]を選択し、[Next]をクリックして下さい。

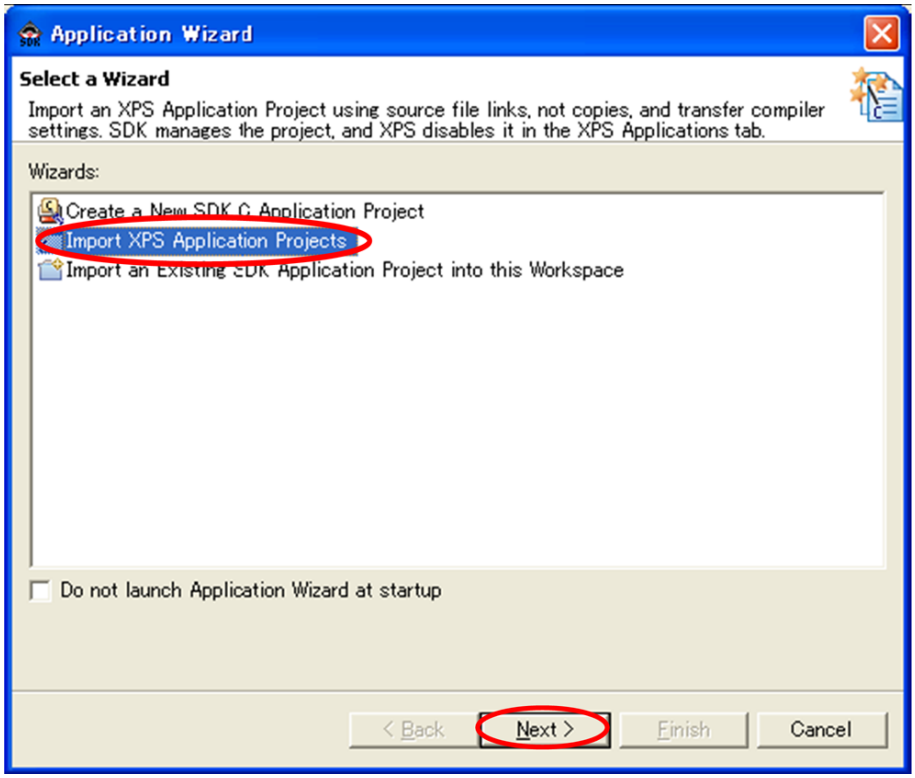


図 12.15. アプリケーションのインポート

BBoot にチェックをし、[Finish]をクリックして下さい。

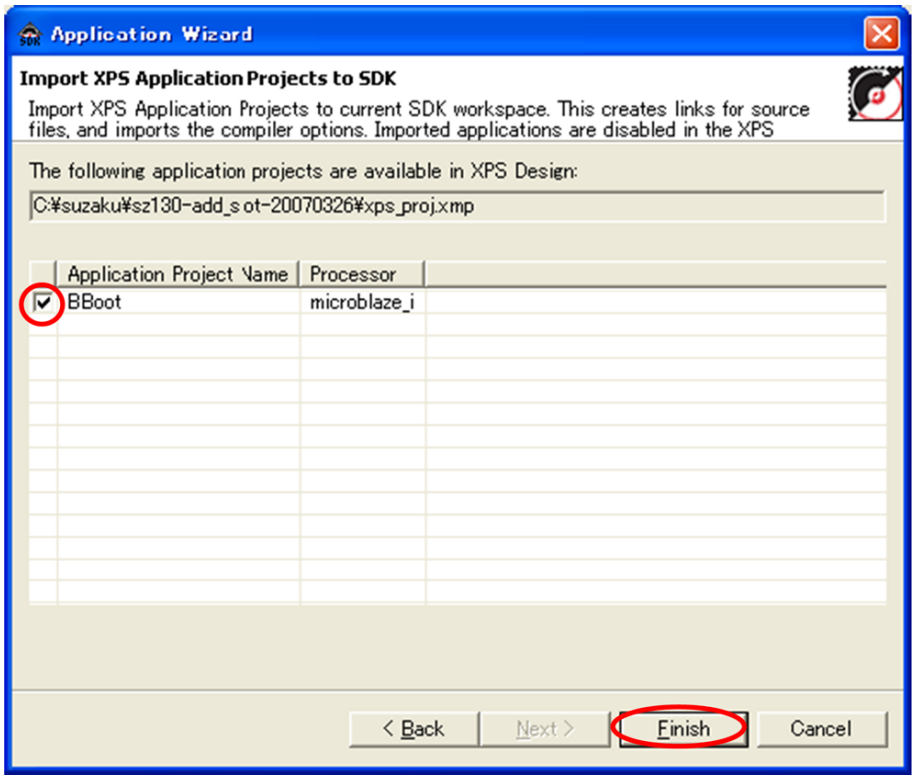


図 12.16. SDK 起動の際の注意

以下の画面が表示されます。良ければ[OK]をクリックして下さい。

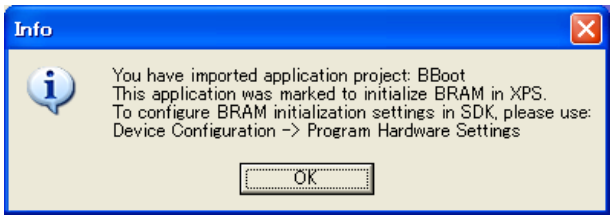


図 12.17. BBoot 利用の際の注意

BBoot がインポートされます。インポート後自動的に Build され、BBoot.elf が作成されます。(SDK の設定によっては Build されません)BBoot の上で右クリックをし、メニューの[Propeties]を選択してください。

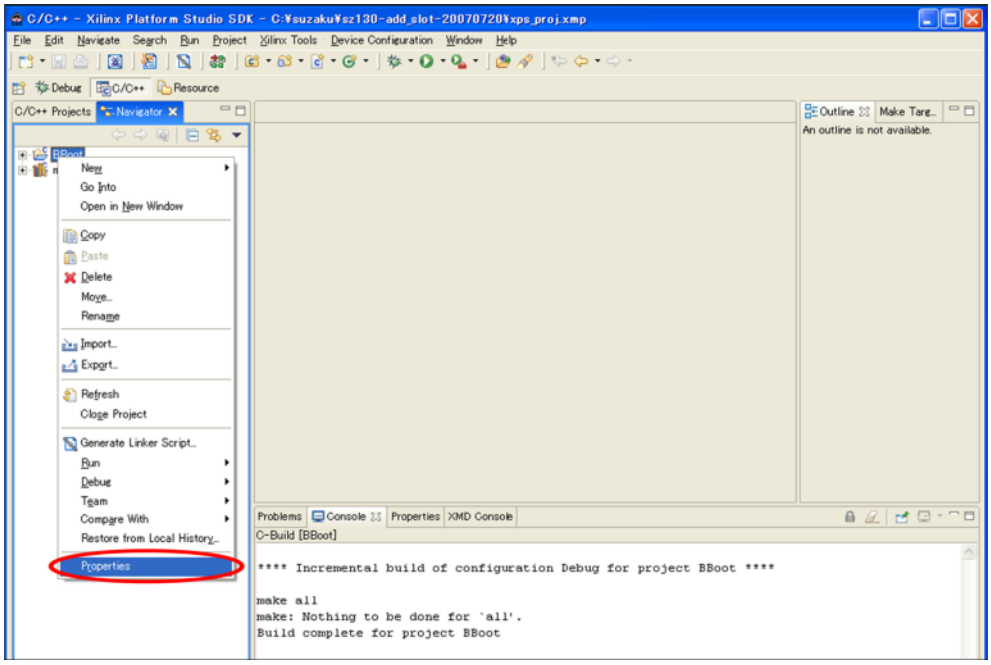


図 12.18. Build 設定

C/C++ Build を選択し、Configuration Settings の Tool Settings タブの Debug and Optimization を選択し、Optimization Level を[No Optimization]に設定して[OK]をクリックして下さい。

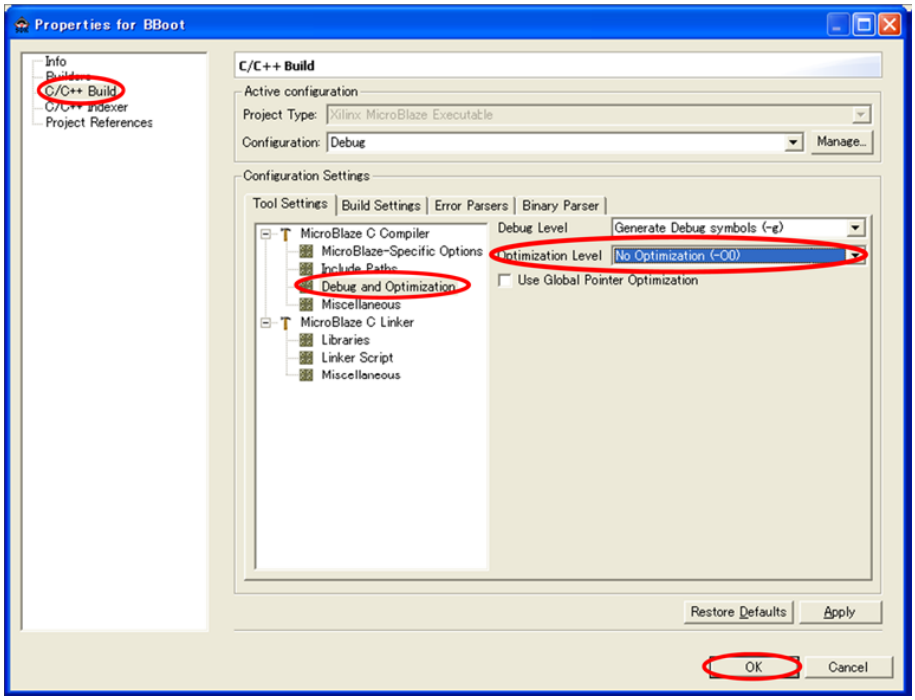


図 12.19. Optimization Level 設定

変更があった場合は自動的に Build されます。自動的に Build されない場合は BBoot の上で右クリックをし、メニューで Build Project を選択し、Build して下さい。

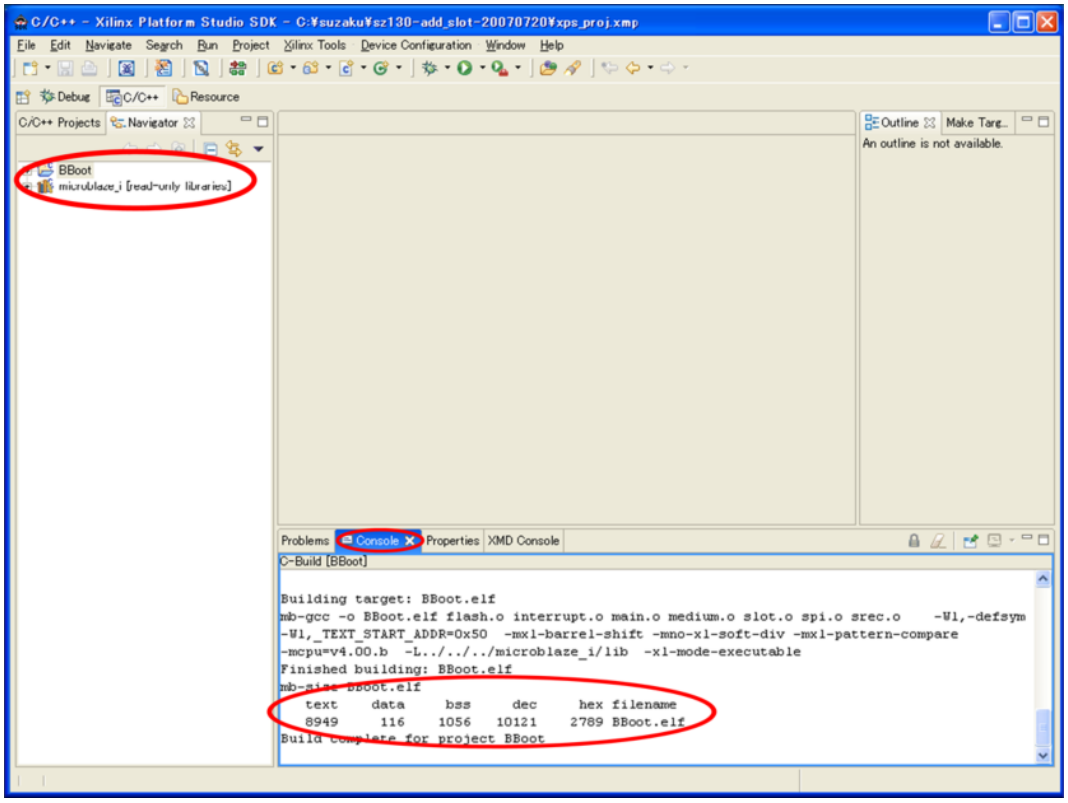


図 12.20. BBoot.elf 作成

[Device Configuration]→[Bitstream Settings]をクリックし、Initialization ELF に先ほど作成された `BBoot.elf` ファイルを指定し、[Save]をクリックして下さい。

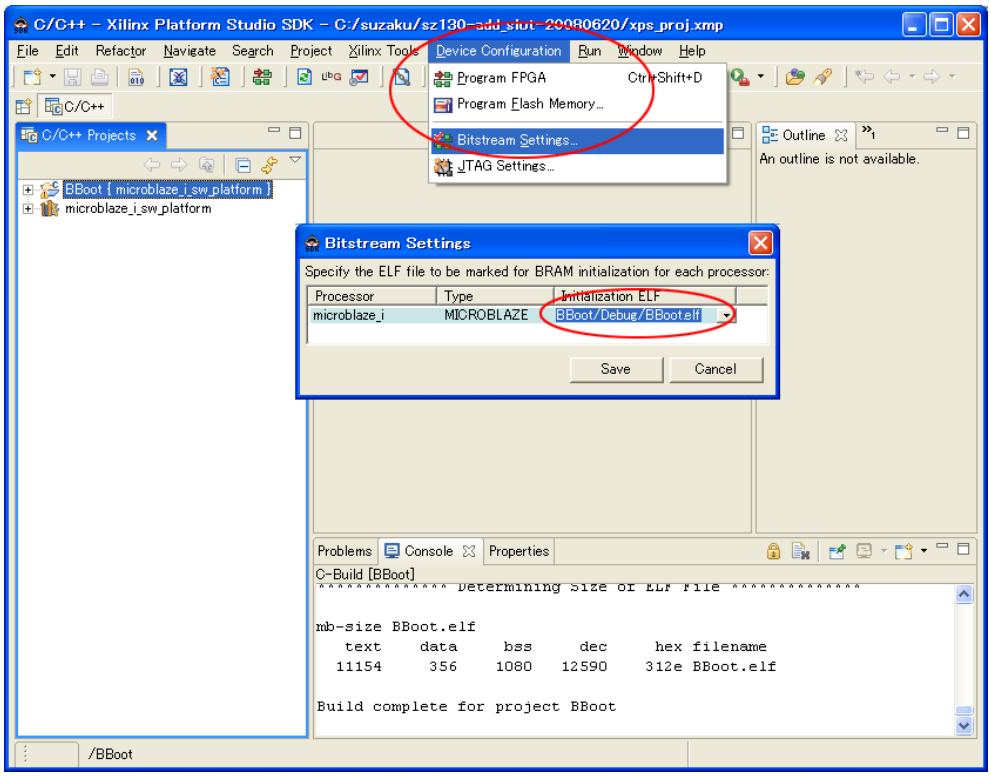


図 12.21. Program Hardware Setting

[Device Configuration]→[Program FPGA]をクリックして下さい。FPGA にデバッグ機能付きのスロットマシンのコンフィギュレーションデータがダウンロードされます。

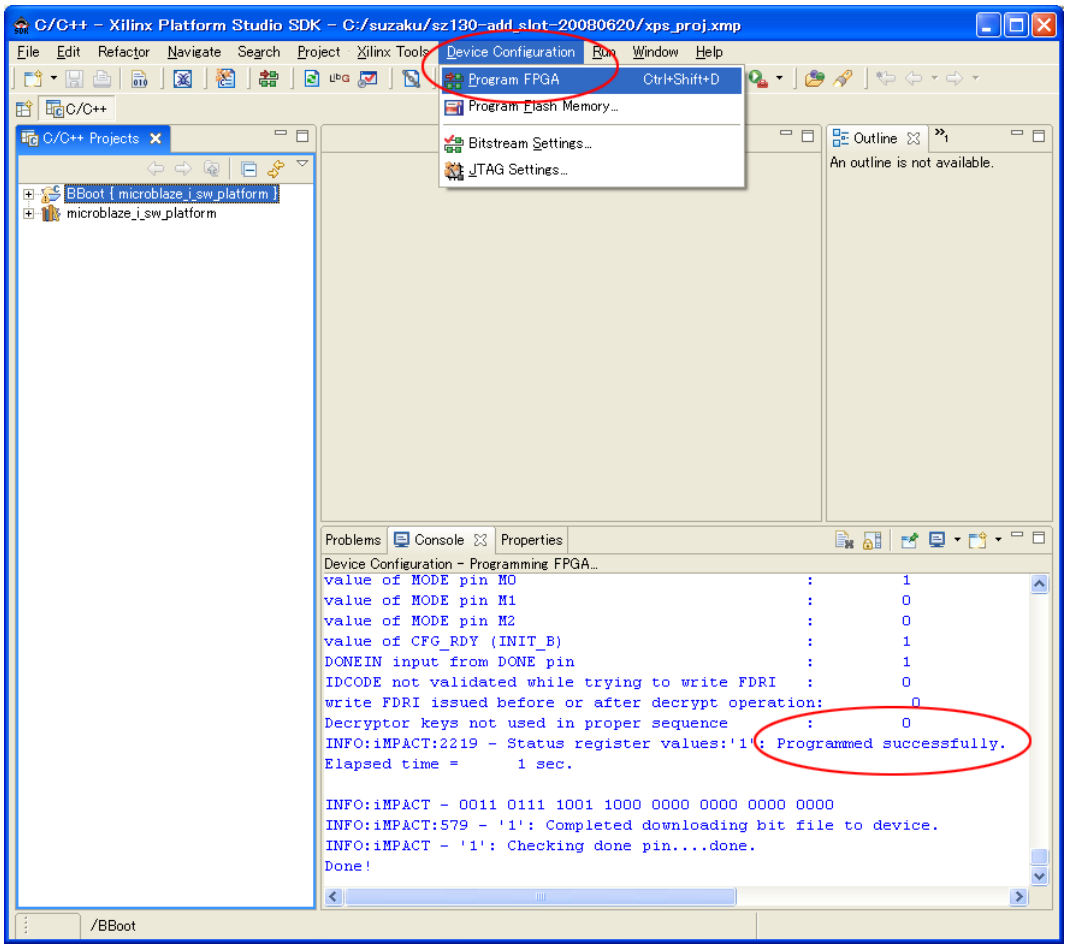


図 12.22. コンフィギュレーションデータダウンロード

BreakPoint を設定します。今回は BreakPoint を timer_interrupt_handler に設定します。`interrupt.c` を開き、int timer_interrupt_handler と書いてある行を探し、行の数字の横でダブルクリックして下さい。BreakPoint が設定されます。

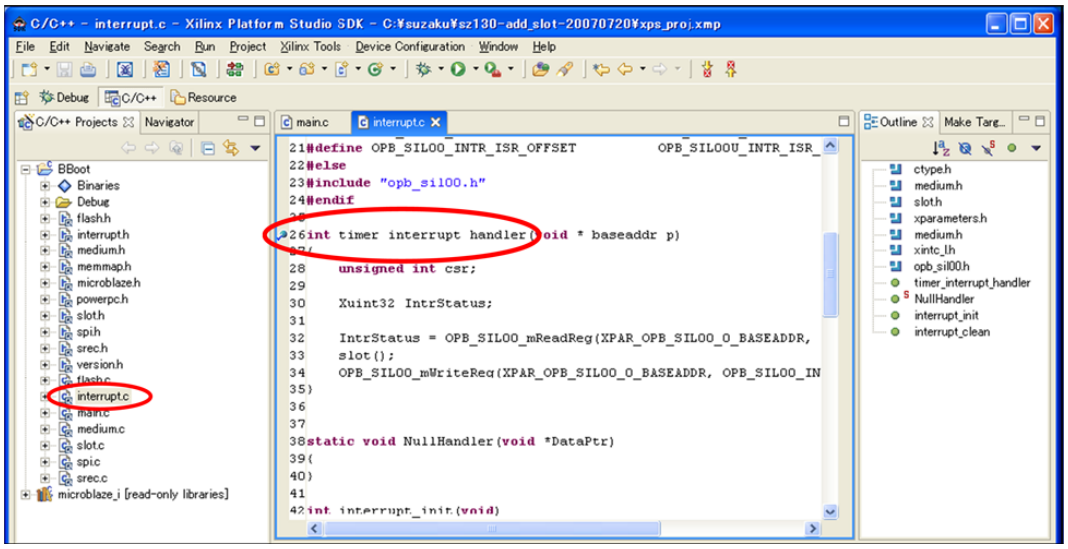


図 12.23. BreakPoint 設定

[Run]→[Debug]をクリックして下さい。

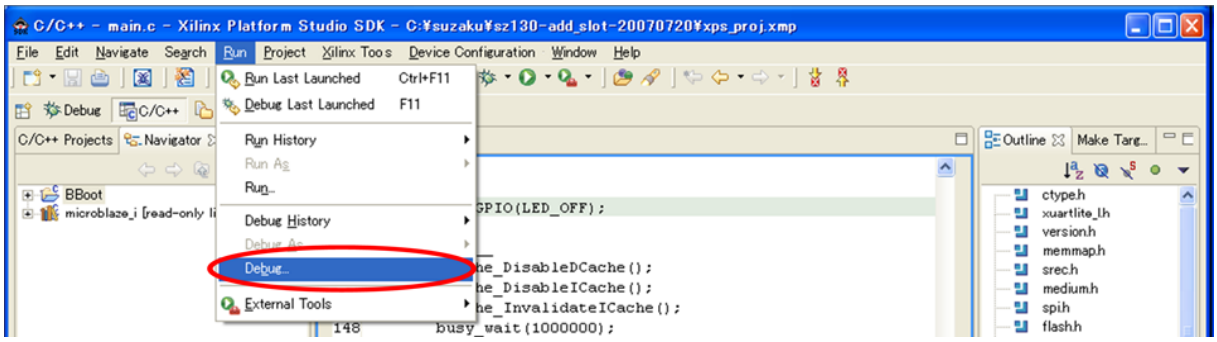


図 12.24. デバッガの設定

以下の画面が表示されるので、[New]をクリックして下さい。BBoot が追加され、Project : BBoot、C/C++Application : Debug\BBoot と設定されます。

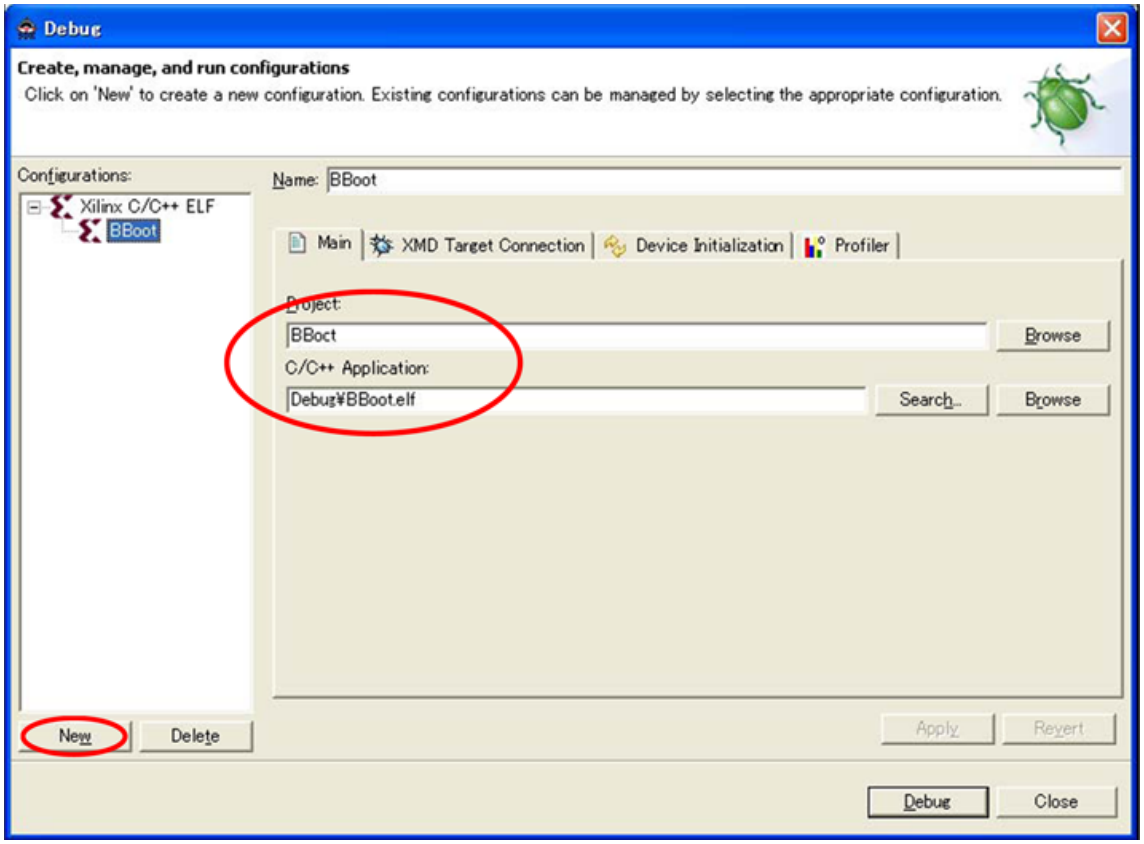


図 12.25. BBoot を追加

[XMD Target Connection]タブをクリックして下さい。XMD Target Type、XMD Connect Command を確認し、[Debug]をクリックして下さい。

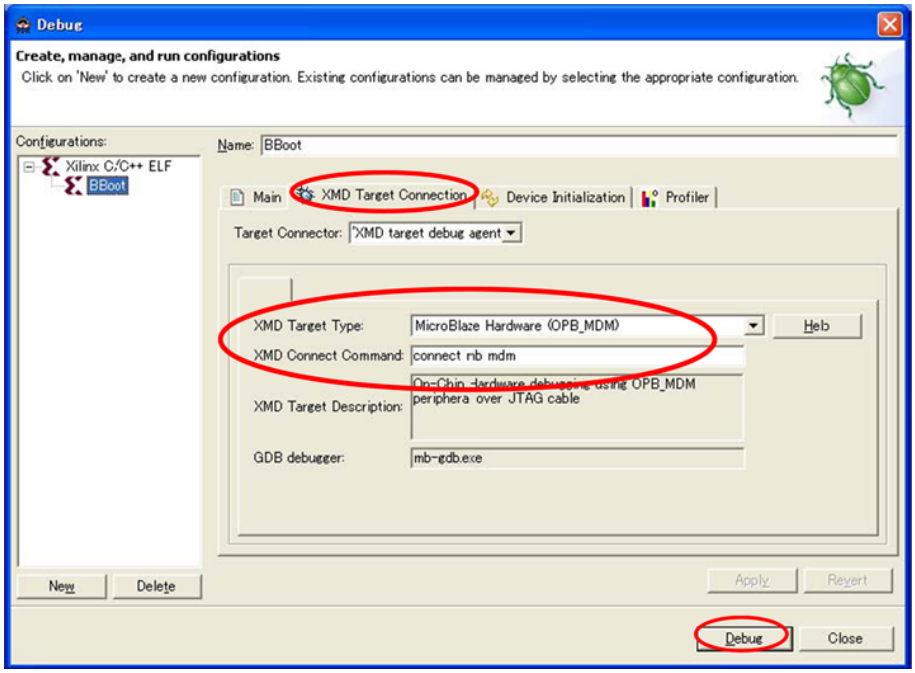


図 12.26. XMD の設定(Microblaze の場合)

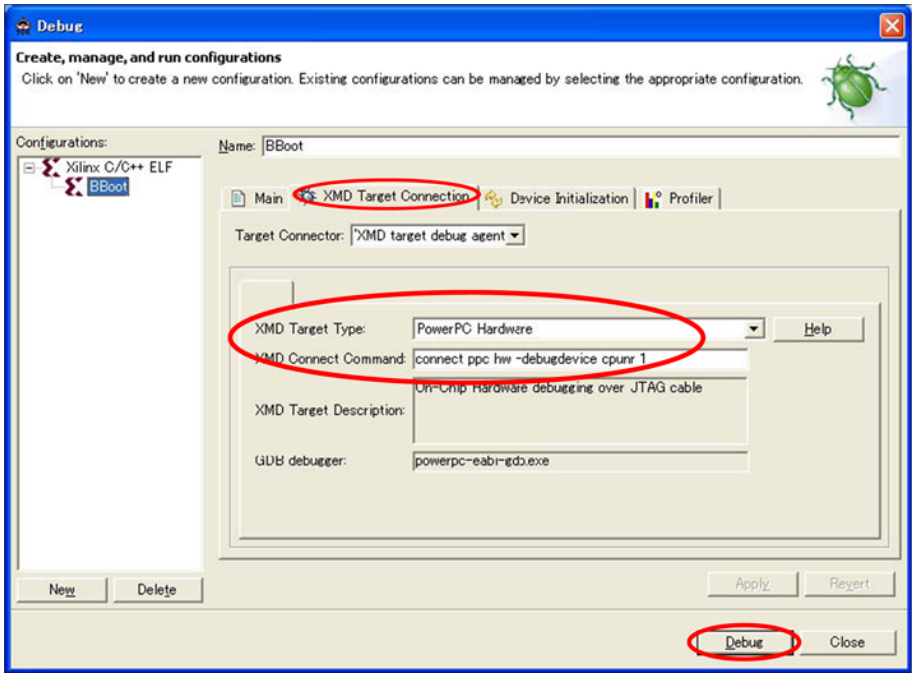


図 12.27. XMD の設定(PowerPC の場合)

XMD、GDB が立ち上がります。main 関数で Break し、以下のような画面になります。

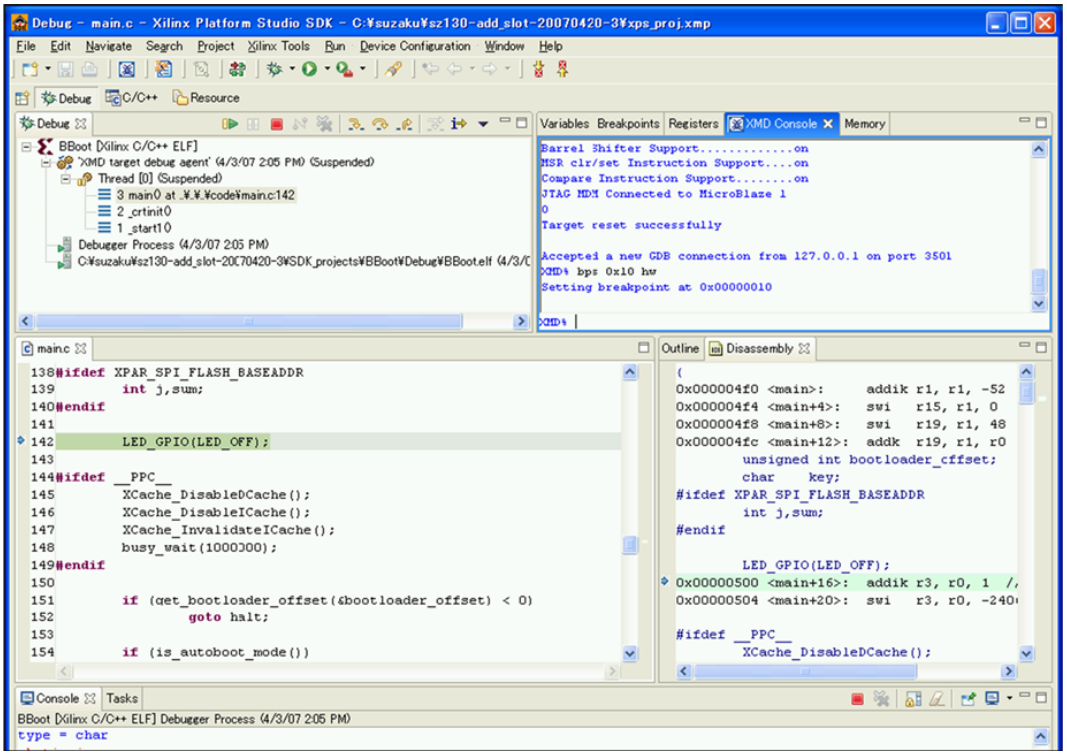


図 12.28. main 関数で Break

Resume をクリックして下さい。先ほど設定した BreakPoint で Break します。Step Into 等をクリックしてください。Instruction Stepping Mode をクリックすると、インストラクション単位でステップ実行できるようになります。

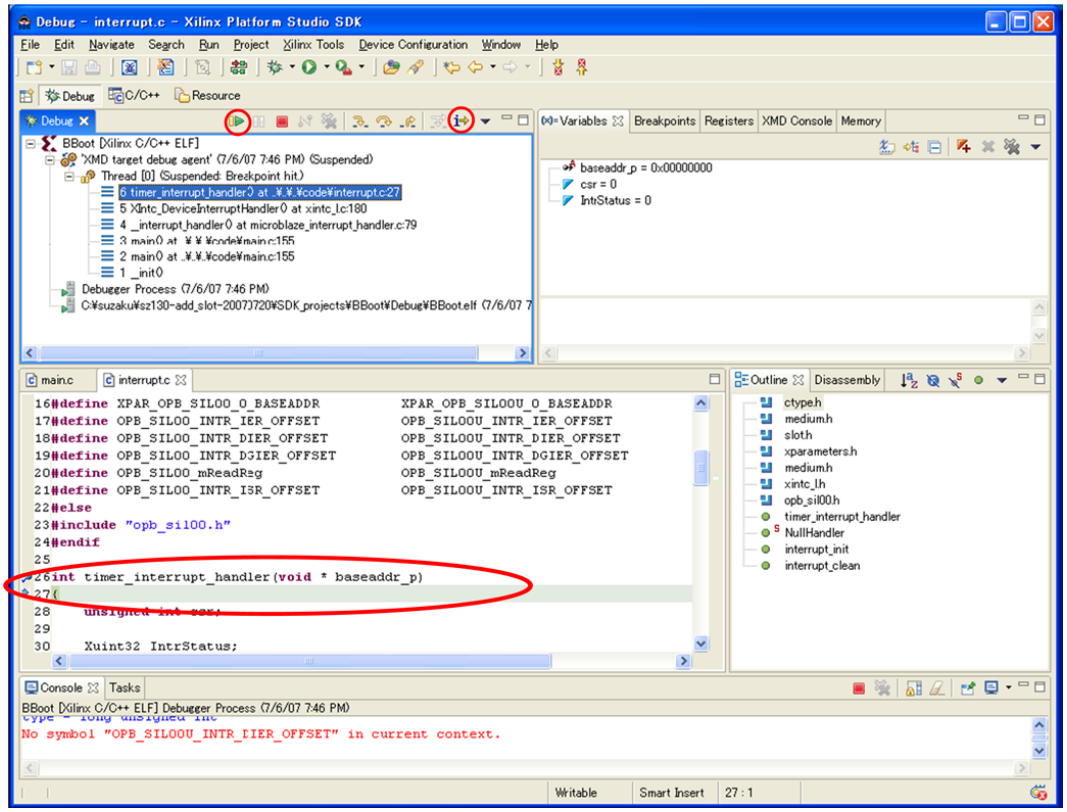


図 12.29. timer_interrupt_handler で Break

ローカル変数やスタックの一覧を確認してみてください。

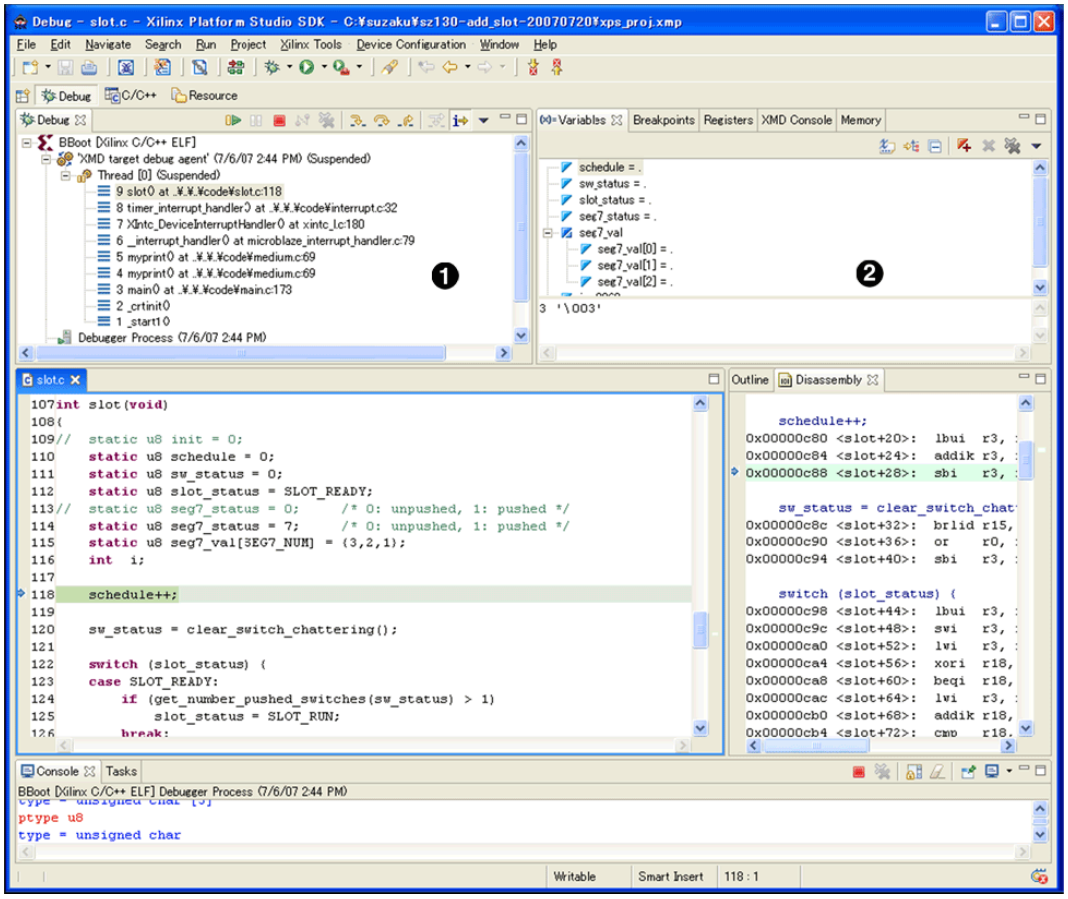


図 12.30. スタック一覧やローカル変数を確認

❶　スタック一覧

❷　ローカル変数

12.5. これから先は・・・

本書はこれで終わりです。FPGA 開発する上での基礎知識、ISE や EDK といった専用開発ツールの使い方、VHDL 言語の記述方法、FPGA に搭載されるプロセッサの使用方法、そして SUZAKU の効果的な使い方は身についたでしょうか。スターターキットを通して学んだことは、ほんの足掛かりにすぎません。ここからは自ら調べ、情報を仕入れ、勉強をし、アイデアを練り、情報を発信し、SUZAKU 開発者のスペシャリストを目指してください。

本書と対になる SUZAKU スターターキットガイド(Linux 開発編)では本書とは違った切り口で SUZAKU の開発を行うので、是非ご一読ください！

12.6. 最新版のダウンロード

本書で紹介いたしましたソースコードやファイルは、不具合解決や機能増強等のアップグレードを行うことがあります。下記サイトに最新版がございますのでダウンロードしてお使いください。

開発に関するファイル [http://suzaku.atmark-techno.com/downloads/all]

各種マニュアル [http://suzaku.atmark-techno.com/downloads/docs]

13.SUZAKU ＋ LED/SW ボードのピンアサイン

SUZAKU と LED/SW ボードの全ピンアサインを載せます。SUZAKU で新たに何かを開発する時などにご参照ください。

13.1. SUZAKU のピンアサイン

13.1.1. SUZAKU CON1 RS-232C

RS-232C コネクタです。レベルバッファを介して、FPGA と接続しています。ボード側で使用しているコネクタは、型式：A1-10PA-2.54DSA、メーカ：ヒロセ(相当品)です。

表 13.1. シリアルコンソールの設定

項目	設定
転送レート	115.2kbps
データ	8bit
パリティ	なし
ストップ bit	1bit
フロー制御	なし

表 13.2. SUZAKU CON1 RS-232C

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
1			空き				
2			空き				
3	RXD	I		E2	C12	C10	Y4
4	RTS	O		F4	B13	D9	V4
5	TXD	O		E4	A13	C9	U4
6	CTS	I		E1	D12	D10	V3
7			空き				
8			空き				
9	GND		グランド				
10	+3.3VOUT	O	内部ロジック用電源 出力+3.3V				

13.1.2. SUZAKU CON2 外部 I/O、フラッシュメモリ用コネクタ

外部 I/O 及びフラッシュメモリ用コネクタです。LED/SW ボードの CON2 とコネクタ接続します。

表 13.3. SUZAKU CON2 外部 I/O、フラッシュメモリ用コネクタ

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
1		グランド				
2	O	内部ロジック用電源出力+3.3V				
3	I	FPGA プログラム用	TCK	CLK	TCK	CLK
4	I	FPGA プログラム用	TDI	D	TDI	D
5	O	FPGA プログラム用	TDO	DO	TDO	DO
6	I	FPGA プログラム用	TMS	nCS	TMS	nCS
7	I/O	外部 I/O	A5	N5	E14	E14
8	I/O	外部 I/O	A7	N4	E15	D15
9	I/O	外部 I/O	A3	M6	E13	E15
10	I/O	外部 I/O	D5	M5	F12	F15
11	I/O	外部 I/O	B4	M3	F13	P4
12	I/O	外部 I/O	A4	M4	F14	P5
13	I/O	外部 I/O	C5	L5	F15	P1
14	I/O	外部 I/O	B5	L6	F16	P2
15	I/O	外部 I/O	E6	L4	G13	L2
16	I/O	外部 I/O	D6	L3	G14	M2
17	I/O	外部 I/O	C6	L2	G15	N2
18	I/O	外部 I/O	B6	L1	G16	N3
19		グランド or 誤挿入防止用				
20	I/O	外部 I/O　(GCLK)	A8	C9	N9	Y7
21		グランド				
22	I/O	外部 I/O　(GCLK)	B8	D9	P9	W7
23	I/O	外部 I/O	E7	K5	G12	N4
24	I/O	外部 I/O	D7	K6	H13	N5
25	I/O	外部 I/O	C7	K4	H14	M3
26	I/O	外部 I/O	B7	K3	H15	M4
27	I/O	外部 I/O	D8	J2	H16	H4
28	I/O	外部 I/O	C8	J1	J16	H5
29	I/O	外部 I/O	A9	F9	J15	E2
30	I/O	外部 I/O	A12	E9	J14	D2
31	I/O	外部 I/O	C10	A10	J13	U9
32	I/O	外部 I/O	D12	B10	K12	V10
33	I/O	外部 I/O	A14	D11	K16	L1
34	I/O	外部 I/O	B14	C11	K15	M1
35	I/O	外部 I/O	A13	F11	K14	G4
36	I/O	外部 I/O	B13	E11	K13	G5
37	I/O	外部 I/O	B12	E12	L16	G2
38	I/O	外部 I/O	C12	F12	L15	F2
39	I/O	外部 I/O	D11	B11	L14	F1

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
40	I/O	外部 I/O	E11	A11	L13	E1
41		グランド				
42		グランド				
43	I	電源入力+3.3V				
44	I	電源入力+3.3V				

13.1.3. SUZAKU CON3 外部 I/O コネクタ

外部 I/O コネクタです。LED/SW ボードの CON3 とコネクタ接続します。

表 13.4. SUZAKU CON3 外部 I/O コネクタ

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
1	I	電源入力+3.3V				
2	I	電源入力+3.3V				
3		グランド				
4		グランド				
5	I/O	外部 I/O	B11	B14	L12	K3
6	I/O	外部 I/O	C11	A14	M13	K2
7	I/O	外部 I/O	D10	D14	M16	K1
8	I/O	外部 I/O	E10	C14	N16	J2
9	I/O	外部 I/O	A10	B16	M15	H3
10	I/O	外部 I/O	B10	A16	M14	H2
11	I/O	外部 I/O	B16	C18	P15	L5
12	I/O	外部 I/O	C16	C17	P13	L4
13	I/O	外部 I/O	C15	D17	R14	K5
14	I/O	外部 I/O	D14	D16	P14	K4
15	I/O	外部 I/O	D15	F15	T15	J6
16	I/O	外部 I/O	D16	F14	T14	J5
17	I/O	外部 I/O	E13	G14	N12	H1
18	I/O	外部 I/O	E14	G13	P12	G1
19	I/O	外部 I/O	E15	F18	N11	F3
20	I/O	外部 I/O	E16	F17	M11	E3
21	I/O	外部 I/O	F12	G15	M10	C3
22	I/O	外部 I/O	F13	G16	N10	C2
23	I/O	外部 I/O　(GCLK)	C9	E10	R9	W5
24		グランド				
25	I/O	外部 I/O　(GCLK)	D9	D10	T9	Y5
26		グランド				
27	I/O	外部 I/O	F14	H14	P10	B2

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
28	I/O	外部 I/O	F15	H15	T8	C1
29	I/O	外部 I/O	G12	H16	R8	A3
30	I/O	外部 I/O	G13	H17	P8	B3
31	I/O	外部 I/O	G14	J12	N8	J4
32	I/O	外部 I/O	G15	J13	P7	J3
33	I/O	外部 I/O	H13	J15	N7	D4
34	I/O	外部 I/O	H14	J14	M7	D3
35	I/O	外部 I/O	H15	J17	M6	D5
36	I/O	外部 I/O	H16	J16	N6	E5
37	I/O	外部 I/O	P16	K15	P5	B4
38	I/O	外部 I/O	R16	K14	N5	C4
39	I/O	外部 I/O	K15	K13	T3	C6
40	I/O	外部 I/O	G16	K12	T2	C5
41						
42		未接続				
43	O	内部ロジック用電源出力+3.3V				
44		グランド				

13.1.4. SUZAKU CON4 外部 I/O コネクタ

外部 I/O コネクタです。コネクタは実装されていません。

表 13.5. SUZAKU CON4 外部 I/O コネクタ

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
1		空き				
2		空き				
3	I/O	外部 I/O	L15	L18	N8	B5
4	I/O	外部 I/O	L14	L17	P7	A5
5	I/O	外部 I/O	K12	L16	N7	A6
6	I/O	外部 I/O	L12	L15	M7	B6
7	I/O	外部 I/O	K14	N18	M6	D8
8	I/O	外部 I/O	K13	M18	N6	C8
9	I/O	外部 I/O	J14	M16	P5	M5
10	I/O	外部 I/O	J13	M15	N5	M6
11	I/O	外部 I/O	J16	P17	T3	C13
12	I/O	外部 I/O	K16	P18	T2	D13

13.1.5. SUZAKU CON5 外部 I/O コネクタ

外部 I/O コネクタです。コネクタは実装されていません。

表 13.6. SUZAKU CON5 外部 I/O コネクタ

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
1		グランド				
2	O	内部ロジック用電源出力 +3.3V				
3	I/O	外部 I/O	P15	M14	P4	F4
4	I/O	外部 I/O	P14	M13	R3	F5
5	I/O	外部 I/O	N16	R15	P3	F6
6	I/O	外部 I/O	N15	R16	P2	E6
7	I/O	外部 I/O	M14	R18	M10	D6
8	I/O	外部 I/O	N14	T18	N10	E7
9	I/O	外部 I/O	M16	U18	P10	D9
10	I/O	外部 I/O	M15	T17	T8	C9
11	I/O	外部 I/O	L13	T15	R8	C12
12	I/O	外部 I/O	M13	R14	P8	D12

13.1.6. SUZAKU CON6 電源入力+3.3V

SUZAKU と LED/SW ボードを接続時は使用しないでください。詳細は「5.3.1. 電源について」を参照してください。

13.1.7. SUZAKU CON7 FPGA JTAG 用コネクタ

FPGA JTAG 用コネクタです。コンフィギュレーションする時は SUZAKU JP1、JP2 をショートしてください。

詳細は「6.2. FPGA の書き換えかた」を参照してください。

表 13.7. SUZAKU CON7 FPGA JTAG 用コネクタ

番号	信号名	I/O	機能
1	GND		グランド
2	+2.5VOUT	O	内部ロジック用電源出力 +2.5V
3	TCK	I	JTAG
4	TDI	I	JTAG
5	TDO	O	JTAG
6	TMS	I	JTAG

2. +2.5V	4. TDI	6. TMS
1. GND	3. TCK	5. TDO

図 13.1. JTAG ピンアサイン

13.1.8. SUZAKU D1、D3 LED

ユーザーコントロール LED(赤)とパワー ON LED(緑)です。

表 13.8. SUZAKU D1、D3 LED

番号	I/O	機能	FPGA 接続ピン番号			
			SZ010 SZ030	SZ130	SZ310	SZ410
D1	O	ユーザーコントロール LED	G5	T3	A9	T4
D3	O	SUZAKU ボードに＋ 3.3V が供給されると点灯				

13.1.9. SUZAKU JP1,JP2 設定用ジャンパ

ジャンパによりオートブートモード、ブートローダモード、FPGA コンフィギュレーション待ちの 3 つの状態に設定します。

表 13.9. SUZAKU JP1、JP2 設定用ジャンパ

信号名	I/O	機能
JP1	I	起動モード設定用ジャンパです。オープンでオートブート(SUZAKU 起動時に Linux が自動的に起動)します。ショートでブートローダモード(ブートローダのみを起動した状態)になります。
JP2		FPGA に JTAG からコンフィギュレーションする時と、フラッシュメモリにコンフィギュレーションデータをダウンロードする時に使用するジャンパです。(本ジャンパをショートすると、電源再投入時 FPGA に対し、コンフィギュレーションを停止することができます)

13.1.10. SUZAKU L2 Ethernet 10BASE-T/100BASE-TX

ボード側で使用しているコネクタ型式/メーカは、J0026D21B/PULSE です。

表 13.10. SUZAKU L2 Ethernet 10BASE-T/100BASE-TX

番号	信号名	I/O	機能
1	TX+		差動ツイストペア出力+
2	TX-		差動ツイストペア出力-
3	RX+		差動ツイストペア入力+
4			75Ω 終端(4 番ピンと 5 番ピンはショートしています)
5			75Ω 終端(4 番ピンと 5 番ピンはショートしています)
6	RX-		差動ツイストペア入力-
7			75Ω 終端(7 番ピンと 8 番ピンはショートしています)
8			75Ω 終端(7 番ピンと 8 番ピンはショートしています)

13.2. LED/SW ボードのピンアサイン

13.2.1. LED/SW CON1 テスト拡張用コネクタ

CON3 と同じピンアサインで信号が配線接続されています。詳しくは CON3 を参照してください。

13.2.2. LED/SW CON2 SUZAKU 接続コネクタ

SUZAKU CON2 と接続します。

表 13.11. SUZAKU CON1 RS-232C

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
1	GND		グランド				
2	+3.3V	I	+3.3V SUZAKU 側から供給				
3	CONF_C			TCK	CLK	TCK	CLK
4	CONF_I			TDI	D	TDI	D
5	CONF_O			TDO	DO	TDO	DO
6	CONF_S			TMS	nCS	TMS	nCS
7	NC			A5	N5	E14	E14
8	UART3	I	RTS	A7	N4	E15	D15
9	UART2	O	TXD	A3	M6	E13	E15
10	UART1	O	CTS	D5	M5	F12	F15
11	UART0	I	RXD	B4	M3	F13	P4
12	NC			A4	M4	F14	P5
13	SEG7	O	セグメント DP "High"で点灯	C5	L5	F15	P1
14	SEG6	O	セグメント G "High"で点灯	B5	L6	F16	P2
15	SEG5	O	セグメント F "High"で点灯	E6	L4	G13	L2
16	SEG4	O	セグメント E "High"で点灯	D6	L3	G14	M2
17	SEG3	O	セグメント D "High"で点灯	C6	L2	G15	N2
18	SEG2	O	セグメント C "High"で点灯	B6	L1	G16	N3
19			誤挿入防止用				
20	SEG1	O	セグメント B "High"で点灯	A8	C9	N9	Y7
21	GND		グランド				
22	SEG0	O	セグメント A "High"で点灯	B8	D9	P9	W7
23	NC			E7	K5	G12	N4
24	nSEL2	O	7 セグメント LED3 "Low"でコモン選択	D7	K6	H13	N5
25	nSEL1	O	7 セグメント LED2 "Low"でコモン選択	C7	K4	H14	M3
26	nSEL0	O	7 セグメント LED1 "Low"でコモン選択	B7	K3	H15	M4
27	NC			D8	J2	H16	H4
28	nCODE3	I	ロータリスイッチ 4 ビット目 選択時"Low"	C8	J1	J16	H5

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
29	nCODE2	I	ロータリスイッチ 3 ビット目 選択時"Low"	A9	F9	J15	E2
30	nCODE1	I	ロータリスイッチ 2 ビット目 選択時"Low"	A12	E9	J14	D2
31	nCODE0	I	ロータリスイッチ 1 ビット目 選択時"Low"	C10	A10	J13	U9
32	NC			D12	B10	K12	V10
33	nSW2	I	押しボタンスイッチ SW3　押下で"Low"	A14	D11	K16	L1
34	nSW1	I	押しボタンスイッチ SW2　押下で"Low"	B14	C11	K15	M1
35	nSW0	I	押しボタンスイッチ SW1　押下で"Low"	A13	F11	K14	G4
36	NC			B13	E11	K13	G5
37	nLE0	O	単色 LED(緑) D1 "Low"で点灯	B12	E12	L16	G2
38	nLE1	O	単色 LED(緑) D2 "Low"で点灯	C12	F12	L15	F2
39	nLE2	O	単色 LED(緑) D3 "Low"で点灯	D11	B11	L14	F1
40	nLE3	O	単色 LED(緑) D4 "Low"で点灯	E11	A11	L13	E1
41	GND		グランド				
42	GND		グランド				
43	+3.3V	O	電源出力 +3.3V SUZAKU 側に供給				
44	+3.3V	O	電源出力 +3.3V SUZAKU 側に供給				

13.2.3. LED/SW CON3 SUZAKU 接続コネクタ

SUZAKU CON3 と接続します。

表 13.12. LED/SW CON3 SUZAKU 接続コネクタ

番号	信号名	I/O	機能
1	+3.3V	O	+3.3V SUZAKU 側に供給
2	+3.3V	O	+3.3V SUZAKU 側に供給
3	GND		グランド
4	GND		グランド
5			
6			
7			
8			

番号	信号名	I/O	機能
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24	GND		グランド
25			
26	GND		グランド
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43	+3.3V	I	+3.3V SUZAKU 側から供給
44	GND		グランド

13.2.4. LED/SW CON4 テスト拡張用コネクタ

CON2 と同じピンアサインで信号が配線接続されています。信号についての詳細は CON2 を参照してください。1~6 ピンにフラッシュメモリ書き込み用コネクタが実装されています。SUZAKU と接続時、フラッシュメモリにデータを書き込めます。書き込む時は SUZAKU JP1、JP2 をショートしてください。

詳細は「6.2. FPGA の書き換えかた」を参照してください。

表 13.13. LED/SW CON4 フラッシュメモリ書き込み用コネクタ

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
1	GND						
2	+3.3V						
3	CONF_C	I		TCK	CLK	TCK	CLK
4	CONF_I	I		TDI	D	TDI	D
5	CONF_O	O		TDO	DO	TDO	DO
6	CONF_S	I		TMS	nCS	TMS	nCS

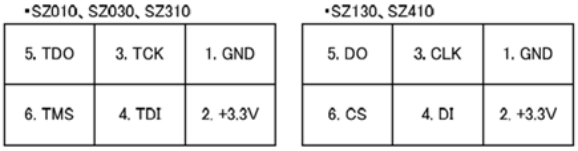


図 13.2. フラッシュメモリ書き込み　ピンアサイン

13.2.5. LED/SW CON6 ＋ 5V 入力コネクタ

＋ 5V±5%の電源を入力してください。AC アダプタ 5V は添付品をご使用ください。（＋ 5V 出力 EIAJ ＃2）

表 13.14. LED/SW CON6 ＋ 5V 入力コネクタ

番号	信号名	I/O	機能
1	＋ 5V	I	＋ 5V センタープラスピン
2	GND		グランド

図 13.3. ＋ 5V センタープラスピン

13.2.6. LED/SW CON7 RS-232C コネクタ

D-sub9 ピンが実装されています。

表 13.15. LED/SW CON7 RS-232C コネクタ

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
1							
2	UART0	I	RXD	B4	M3	F13	P4
3	UART2	O	TXD	A3	M6	E13	E15

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
4							
5	GND		グランド				
6							
7	UART3	O	RTS	A7	N4	E14	D15
8	UART1	I	CTS	D5	M5	F12	F15
9							

13.2.7. LED/SW 7 セグメント LED セレクタ

7 セグメント LED 選択用 PNP トランジスタが実装されています。"Low"を入力すると、それぞれに対応する 7 セグメント LED を選択することができます。

表 13.16. LED/SW 7 セグメント LED セレクタ

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
LED1	nSEL0	O	LED1 コモン "Low"で選択	B7	K3	H15	M4
LED2	nSEL1	O	LED2 コモン "Low"で選択	C7	K4	H14	M3
LED3	nSEL2	O	LED3 コモン "Low"で選択	D7	K6	H13	N5

13.2.8. LED/SW LED1 ～ 3 7 セグメント LED

7 セグメント LED が 3 つ実装されています。"High"を入力すると、それぞれに対応するセグメントを点灯させることができます。

表 13.17. LED/SW LED1~3 7 セグメント LED

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
A	SEG0	O	セグメント A "High"で点灯	B8	D9	P9	W7
B	SEG1	O	セグメント B "High"で点灯	A8	C9	N9	Y7
C	SEG2	O	セグメント C "High"で点灯	B6	L1	G16	N3
D	SEG3	O	セグメント D "High"で点灯	C6	L2	G15	N2
E	SEG4	O	セグメント E "High"で点灯	D6	L3	G14	M2
F	SEG5	O	セグメント F	E6	L4	G13	L2

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
			"High"で点灯				
G	SEG6	O	セグメント G "High"で点灯	B5	L6	F16	P2
DP	SEG7	O	セグメント DP "High"で点灯	C5	L5	F15	P1

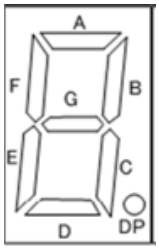


図 13.4. 7 セグメント LED

13.2.9. LED/SW D1 ～ 4 単色 LED(緑)

単色 LED（緑）が４つ実装されています。"Low"を入力すると点灯します。

表 13.18. LED/SW D1 ～ 4 単色 LED(緑)

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
D1	nLE0	O	単色 LED(緑)　D1 "Low"で点灯	B12	E12	L16	G2
D2	nLE1	O	単色 LED(緑)　D2 "Low"で点灯	C12	F12	L15	F2
D3	nLE2	O	単色 LED(緑)　D3 "Low"で点灯	D11	B11	L14	F1
D4	nLE3	O	単色 LED(緑)　D4 "Low"で点灯	E11	A11	L13	E1

13.2.10. LED/SW SW1 ～ 3 押しボタンスイッチ

押しボタンスイッチが 3 つ実装されています。押すと"Low"を出力します。

表 13.19. LED/SW SW1 ～ 3

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
SW1	nSW0	I	押しボタンスイッチ SW1　押下で"Low"	A13	F11	K14	G4
SW2	nSW1	I	押しボタンスイッチ SW2　押下で"Low"	B14	C11	K15	M1

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
SW3	nSW2	I	押しボタンスイッチ SW3　押下で"Low"	A14	D11	K16	L1

13.2.11. LED/SW SW4 ロータリコードスイッチ

ロータリコードスイッチが実装されています。選択時"Low"を出力します。

表 13.20. LED/SW SW4

番号	信号名	I/O	機能	FPGA 接続ピン番号			
				SZ010 SZ030	SZ130	SZ310	SZ410
SW4	nCODE0	I	ロータリコードスイッチ　2^0 選択で”Low”	C10	A10	J13	U9
	nCODE1	I	ロータリコードスイッチ　2^1 選択で”Low”	A12	E9	J14	D2
	nCODE2	I	ロータリコードスイッチ　2^2 選択で”Low”	A9	F9	J15	E2
	nCODE3	I	ロータリコードスイッチ　2^3 選択で”Low”	C8	J1	J16	H5

参考文献

[1] *「エンベデッド システム ツール リファレンス マニュアル」*. Xilinx 株式会社.

[2] *「開発システムリファレンスガイド」*. Xilinx 株式会社.

[3] *「Platform Studio Help」*. Xilinx 株式会社.

[4] *「ISE Help」*. Xilinx 株式会社.

[5] *「VHDL によるハードウェア設計入門」*. CQ 出版社. 長谷川裕恭 .

[6] *「デザインウェーブマガジン 2004 年 5 月号ソフト・マクロの CPU で Linux を動かす（前編）」*. CQ 出版社.

[7] *「改訂　初めてでも使える HDL 文法ガイド」*. CQ 出版社.

改訂履歴

バージョン	年月日	改訂内容
1.0.0	2006/07/14	・初版作成
1.0.1	2006/07/19	・誤記訂正
1.0.2	2006/07/24	・ピンアサイン訂正(CON4 の 9、10 ピン)
2.0.0	2006/08/11	・SZ010、SZ030、SZ310 対応のための全面変更(以下重要な変更のみ記) "JTAG Clock に変更する"の記載を消去 BBoot の変更、CD-ROM の内容変更 自作コアに割り込み機能追加
2.0.1	2006/08/18	・TE7720 の図の文字化けを修正
2.0.2	2006/08/23	・誤記訂正
2.0.3	2006/10/18	・Linux 編にあわせて章構成変更 ・保証に関する注意事項追記 ・半田付けの際の注意を追記 ・コネクタ説明箇所に JTAG、Flash メモリ書き込みについて追記 ・8.2i 対応のための内容を追記
2.1.2	2006/11/30	・構成を大幅変更(以下重要な変更のみ記) ダイナミック点灯、単色 LED 順次点灯、デコーダの VHDL 修正 SZ130 のメモリマップ　誤記訂正 SZ310 の構成図　誤記訂正 IP コアハード編の項 誤記訂正 EDK の項　ソフトウェアについて追記 EDK の項　IP コアの説明を追記 シミュレーションの記述を追記 SUZAKU の書き込み方を一つにまとめて記述 デバッグの項追加 参考文献を追加 EDK を ISE のサブモジュールにする方法を追加
2.1.3	2006/12/06	・メモリマップ誤記訂正 ・JTAG、SPI のピンアサインの図に色づけ ・その他誤記訂正 ・表紙デザイン改版
2.1.4	2007/01/19	・EDK の使い方の章に SZ310 のリンカースクリプトについて追記 ・EDK の使い方の章に BSB の使い方を追加
2.1.5	2007/02/16	・SUZAKU についての章に SUZAKU 全体ブロック図等追記 ・TIPS　パラレルポートがなくても追加 ・TIPS　FPGA の bin ファイルの作り方追加 ・TIPS　XMD コマンド追加

		・TIPS　MicroBlaze 追加 ・その他修正、構成変更
2.1.6	2007/03/16	・2 段階 Boot の図を修正、誤記訂正
2.1.7	2007/04/20	・BSB で MicroBlaze の章に PowerPC の内容を追記 ・デバッグの章の内容修正
2.1.8	2007/07/20	・ISE/EDK9.1i にて内容確認&内容修正 静的割り込みから動的割り込みに変更 ・UCF に CMOS 3.3V を明記 ・SDK を使ってデバッグの章を追加 ・その他文章訂正
2.2.0	2007/10/10	・SZ410 対応のための全面変更
2.2.1	2007/10/19	・TIPS　全 IP 表示追加
2.2.2	2007/12/14	・ISE/EDK9.2i にて内容確認&内容修正 9.2i 用 BSB の章追記 IPIF のキャプチャを 9.2i に変更 デバッグの章修正 9.2i で SZ130 でスロットのプログラムが 8KByte を超えるため 16KByte に BRAM を増やすよう内容追記 ・TIPS 目次追加 ・必要なものに Parallel Cable Fly Leads 追記
2.3.0	2008/02/15	・SZ410 のデフォルトプロジェクト変更にともなう内容修正 ・BBoot2.3 -> 2.4 にともなう内容修正
2.3.1	2008/03/14	・SZ410 のデフォルトプロジェクト変更にともなう内容修正 MPMC で BRAM のリソースを使用しないように変更
2.3.1	2008/04/21	・「6.3.1. BBoot で書き換える」修正 BBoot にファイルのチェックサム機能が新たに追加されたため内容修正
2.3.3	2008/06/20	・表紙に ISE/EDK の対応バージョン明記 ・誤記訂正
2.4.0	2008/06/20	・ISE/EDK10.1i 用に内容修正
2.4.1	2008/07/18	・誤記修正
2.4.2	2008/09/26	・タイトルを英語表記からカタカナ表記に ・BRAM を増やした場合のリンカスクリプト更新方法を追記
2.4.3	2009/01/09	・誤記修正
2.4.4	2009/03/19	・「2. LED/SW ボードについて」2 章のタイトル修正 ・参照先を記述する際の表記を統一 ・表記ゆれを修正
2.4.5	2009/07/17	・本文のレイアウト統一 ・文字化け修正 ・文字が読みにくい画像の差し替え

SUZAKU スターターキットガイド(FPGA 開発編)
Version 2.4.5-8d87fa8
2009/07/17

株式会社アットマークテクノ
060-0035 札幌市中央区北 5 条東 2 丁目 AFT ビル 6F TEL 011-207-6550 FAX 011-207-6570