

Visual Synthesizer := $\frac{Sonitus}{Lucis \times Colorum}$
Lucis et Colorum Sonitus

Iori yoneji

2011年1月27日

written with p^ATEX2e

第Ⅰ部

アブストラクト

この論文は、

- 実際に自分がオープンソフトウェア/ハードウェア、オープンノウハウについての公開を通して、これらの活動への、読者の参加容易性の向上

を図るとともに、

- オープンソフトウェア/ハードウェア、オープンノウハウによる恩恵の民間での享受可能性
- ソースコードを実際に読み、書くことによって、ソフトウェアとハードウェアの製作について実例を挙げ、現状の開発環境の問題点と実地でのプロセス
- パーソナルファブリケーションとオープンソースをとりまく社会とビジネスモデルの現状と今後の予測
- 本研究で開発したハードウェア、およびソフトウェアの新規性、および課題

を明らかにする

ことを目的とする。

今回の製作対象の、外から見た仕様について

- 学生が買える、廉価なカメラモジュールで、室内等の明るさや色などを元にパラメータを作成する。
- パラメータを元にスピーカから音が鳴る。

と定める。

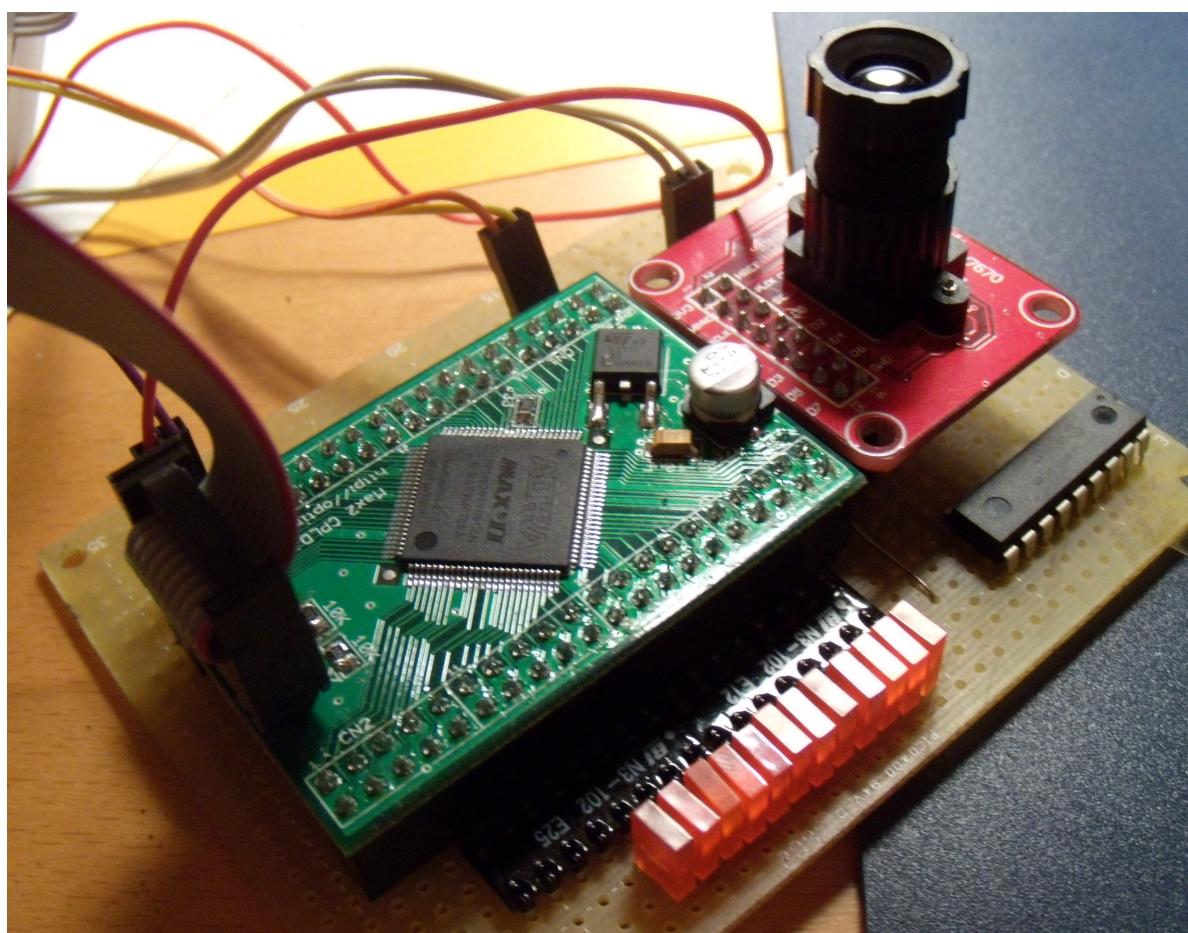
目次

第 I 部 アブストラクト	1
第 II 部 哲学	5
1 前口上 -お目汚し-	5
2 実用性 -音楽的要素がなくても-	5
2.1 こういった、直接の役に立たない物作りの土壤	6
2.2 キーワードとしての オープンソース ラピット・プロトタイピング パーソナル・ファブリケーション	7
第 III 部 内部仕様について	10
3 最初期の仕様	10
3.1 カメラ -目玉-	10
3.2 プロセッサ -大脳-	10
3.3 CPLD -脊髄脳髄その他-	10
3.4 SRAM -海馬。中期記憶-	10
4 現在の仕様	12
4.1 SRAM の破棄	12
第 IV 部 開発環境	14
5 オープンソースデスクトップ環境において	14
5.1 ARM マイコンのコンパイラ	15
5.2 AlteraCPLD の開発環境	16
5.3 その他	16
6 Windows 環境において	17
6.1 ARM の開発環境	17
6.2 AlteraCPLD の開発環境	17
7 git と github について	17
第 V 部 Verilog を書く	21

8	モジュールの構成	21
9	clkdivider(分周器)	22
10	データ抽出レイヤ	22
10.1	ピクセル選択モジュール	22
10.2	ピクセルデータフェッチ, 連結モジュール	26
11	データ転送レイヤ	27
12	シミュレーション	27
12.1	シミュレータの意義	27
12.2	シミュレーションモデル	30
第 VI 部 I2C セットアップ関数の開発		33
13	データ送出関数	33
13.1	動作	33
13.2	問題点	34
14	シーケンシャルデータ受信関数	34
14.1	バイトリード	34
15	OV7670 全リード関数	35
16	OV7670 バイトライト関数	35
第 VII 部 SCCB での OV7670 コンフィギュレーション		36
17	OV7670 の注意点	36
18	OV7670 用のレジスタセットアップデータの整理	36
第 VIII 部 独自シリアル転送プロトコル		39
19	ransmicta	40
20	レシーバ	40
第 IX 部 関数値渡しの謎		40
第 X 部 ノウハウの共有		47

21	ブログ	48
22	SNS	48
23	フォーラム	48
24	メーリングリスト	48
第 XI 部 この論文に使われている技術		48
25	ヴァージョン管理	48
26	TeX	48
第 XII 部 社会に於けるオープンソースの立場		48
27	ビジネスモデル	48
28	ホスティングサービス	48
29	近年のクオリティの向上、幅広い層の参画	48
第 XIII 部 90 億総ソースコード・リテラシーの時代へ		48
第 XIV 部 文献集		48
30	データシート	48
30.1	半導体-英語	48
31	お世話になった個人のサイト（必ずしもサイトのデータというわけではなく直接お話を伺った場合を含む、敬称略）	49
32	公式サイト	49
33	解説書など	49
34	よむものリスト	49
34.1	センサ	50
34.2	規格	50

図1 OV7670とMAX II



第 II 部 哲学

1 前口上 -お目汚し-

過去に様々な形において、音楽と映像の関連づけが行われてきた。
映画中の音楽であったり…といった面もあるが、それはむしろ視聴者への心理的な関連付けを両者が並列して行っているということである。
ここで私が言いたいのは、むしろビジュアライザやイコライザの類についてである。
つらつら長く書いてしまいそうで怖い。意味の希薄な前口上を冗長に書いて見たところでなにもない。当然だ。つまるところ、と結論を急いでしまおう。詰まるところ、私の言いたいのは、iTunes や Rhythmbox のようなビジュアライザのような音楽のデータの（ごく一部をランダムに用いた）画像の自動生成システムを、なんの価値もなく、ただいたずらに PC に負担をかけるばかりで、あきるほど見てきた。しかし、今までにおいて、その逆のハードウェアを作ってみようという試みは寡聞にして聞かない。
音楽は、

- 作業中、考え中、読書中の類のときの BGM として。またトーク番組の BGM として
- その音楽を楽しむ。
- その曲を覚えてなにか自分で歌って踊って演奏してみたいなことをする。

といった用途が考えられる訳だけれども、とにもかくにも、再生中のディスプレイを凝視することなんて（むしろそのウィンドウが開かれていることだって）、普通に考えて、ないことなのだ。

ところが、である。なにか外を向いてみる、とかその類の、目で見ているもの、顔を向けている対象に特定の意味と意志をフォーカスしていないときには、その風景が音楽化 (soundlize) されても、困らない。.. んじゃないかなあと考えたわけである。

崩して言ってみる。
飽きているからかも知れないけれど、音楽が（機械的に）映像化されているのをみても、何とも思わないか、ムッとする。遅い PC を使って作業しているとなおさらだ。
けれど、外を見たときとかみたいな、Not Watch nor Look, But see な状況の時、それが音楽化されてみたら、ちょっと楽しいかもしれない。
そういうことを企んだのだった。

2 実用性 -音楽的要素がなくても-

音楽的要素が無くしても、視覚情報を音声に変換するという試みは、例えば後天的に目の見えなくなってしまった人への、わずかな手助けとなる可能性を秘めている。
音を位置と色で対応付けることができれば、ある程度の視覚補助になろう。もし、その目的で特化させたものを作るに至れば（今回の開発では到底無理そうだが）、日常的な交通の用に供することすら出来るかもしれない。

2.1 こういった、直接の役に立たない物作りの土壤

物作りとは、役に立つと万人が思うものを作ることのみであろうか。

ある日、東京工業大学の大岡山キャンパスで開かれた Make: Tokyo Meeting に行ってきましたが、半球のボールにたくさんつまみがついていて、大人の科学の付録のシンセとマイコンの接続をわざわざ信号でなくてサーボモータでつまみを動かすことで音作りを視覚化したり、テスラコイルで音楽を再生してみたり、そういった、電気、光、音の融合したような分野での趣味の工作と研究の成果が大小様々に配置されては製作者(高専生、大学生、院生を中心として高校生も含む)がそばにいて、お互い、または見学者と話しているといった雰囲気であった。それに触発もされて、私はこうしてこの文章を書いている。

幸い、先述のものを個人で、実現している例はまだない様なので良かった。

Make:や、筑波大学主催の、産学共同で行われた prosume といった企画は、どういう目的なのだろうか。
そのままでは、役にたたないかも知れないものを作る必要があるのだろうか。

引用開始***参考 1

PROSUME 2010 は、個人でもここまでできる！をコンセプトとした、クリエーターのためのイベントです。電子工作、機械工作、科学実験、クラフト、音響、映像パフォーマンスなど、基本的に個人やサークル等で開発、作成したものであれば何でも出展可能です。またこれらクリエーターを応援してくれる企業の出展も募集します。

Prosume(プロシューム)とは、produce(生産) + consume(消費)を組み合せた造語です。1980年に Alvin Toffler 氏の「第三の波」の中で、生産者であり、消費者であるという意味の、Prosumer が提唱されたのが最初とされています。消費者の個別のニーズに応じた製品は、消費者自身が作り出すというもので、それを可能にするのが従来技術の低コスト化と、身近になった最新技術です。キーワードは、オープンソース、ラピッド・プロトタイピング、そしてパーソナル・ファブリケーションです。

これは、prosume に実際に出展した人との会話を通して得た、私の解釈であるが、”産業資本主義の末期的構造”として一般に問題提起されている構造-すなわち生産者(企業)というブラックボックスから与えられたものを「欲しい！」という動機のみで消費し、”需要が供給を生み出しているのではなく、「開発された量産製品」の存在が需要を生み出している”に対する一つの答えだと思っている。

具体的には、こんなものがあったら良いなというものを作り、そこから情報やアイディアを(消費者として、かつ製作者として)シェアすることにより、他人の作った物より斬新なもの、より複雑なものを目指すことが容易になり、PROSUME にあげられた”従来技術の低コスト化”と、情報技術の高度な発達によってそれが加速されたということである。

もし、一般的の「消費者であり生産者ではない」人への応用が可能になるものをそこから「純粋な生産ブラックボックス」である企業が見いだし、量産し、人類の生活をわずかでも向上できれば、”Prosumer”として本望なのではないか。

2.2 キーワードとしての オープンソース ラピット・プロトタイピング パーソナル・ファブリケーション

「第三の波」が 1980 年に書かれていることは非常に衝撃的である。なぜか？それは、上にあげた様々な革命がほとんど無名の赤ん坊でしかなかった時代だからだ。

2.2.1 オープンソースとは一体何か

参考 2 として、最も普及しているオープンソースライセンス及び哲学のうちの特に歴史があるものとして、GPL(GNU General Public License) の原文のリンクを示す。

これが何を示しているのか。引用開始

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice;

拙訳-入手した (GPL でライセンスされた) ソースコードの (一字一句正しい) 原本をどんな形でも配布できる。そのとき、GPL ライセンスされたソフトウェアであることを、明らかに分かるように示さなければならない

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

参考 3 にしめすのは sourceforge 社 (オープンソース文化のハブとして活躍し続けるプログラム開発支援やテクノロジ系ニュースコミュニティのホスト会社として世界的に有名) にホストされている OpenSourceGroup-Japan による非公式訳である。引用開始

- a) 作品には、あなたが作品を改変したことと、改変に関連した日時を記述した告知を目立つよう載せなければならない。 b) 作品には、それが本許諾書と、下記第 7 項に従って追加された条

件すべての下で公開されていることを記述した告知を目立つように載せなければならない。この条件は、上記第4項における「告知をすべてそのまま保全」するための条項を改変する。c) 作品の大部分を、総体として、コピーを所有するに至った人全員に、本許諾書の下でライセンスしなければならない。そこで、本許諾書は、本許諾書第7項に基づく適用可能な追加的条項のすべてとともに、作品全体に、すなわちその大部分に、それらがどのようにパッケージされているかに関わらず適用されることになる。本許諾書は、これ以外のやり方には作品をライセンスする許可を与えないが、あなたが本許諾書以外で別途許可を得ていた場合には、それによって得られた許可まで無効とするものではない。d) 改変された作品が対話的なユーザインターフェースを有する場合、それらのインターフェースは『適切な法的告知』を表示しなければならない。ただし、『プログラム』に元々『適切な法的告知』を表示しない対話的なインターフェースがある場合、あなたの作品で表示するようにする必要はない。

ここで特徴的なのはコピーされ、改変された work(作品) も GPL でライセンスされることであるが、オープンソースして重要なのはそこではない。

だれかが先行して何かをオープンソースとして作った場合、それをいかなる形でも利用でき、その成果をオープンにすれば誰かに利用してもらえるのだ。

これは開発期間の短縮及びモチベーションの向上につながる。

また、ソースコードを作品と呼んでいるのも、Prosume や Make:的な感性に置ける、芸術としての技術的作品という側面を強く押し出しているとも言えよう。

しかし、この GPL というライセンスすらも、1989 年になって初版が策定されるのだ。Alvin Toffler 氏すごい。

オープンソースとソースコード文学に関しての議論だけ進めてたくさんもの論文がかけよう。しかし、ここでは割愛させていただく。

少なくとも、ここで述べたことが、”Prosumer” という「第三の波」を生み出す土壤になっていることは明らかである。

2.2.2 ラピット・プロトタイピング

rapid prototyping とは高速な試作である。企業単位でしか買えないような大仰な装置を使って試作品を時間をかけてデバッグして世に送り出していくようでは、個人では当然できない上に、大企業が圧倒的に有利で、また、設計時間も長くなってしまう。しかし、計算機科学の驚くべき進化によって、またはコンテンツ産業の貪欲な大規模化によって、たかだかウェブサイトを閲覧する程度のつもりで個人が購入したコンピュータが一昔前のスーパーコンピュータの様な圧倒的な速度を誇っている。このことは、設計工程に関して、廉価な PC で、短期間で CAD を使用して、アナログ回路、論理回路、コンピュータソフトウェア、機械、模型など多くのものをシミュレートし、設計できる様にした。

このことは個人の休日の工作のできる幅を押し広げつづけている。

2.2.3 パーソナルファブリケーション

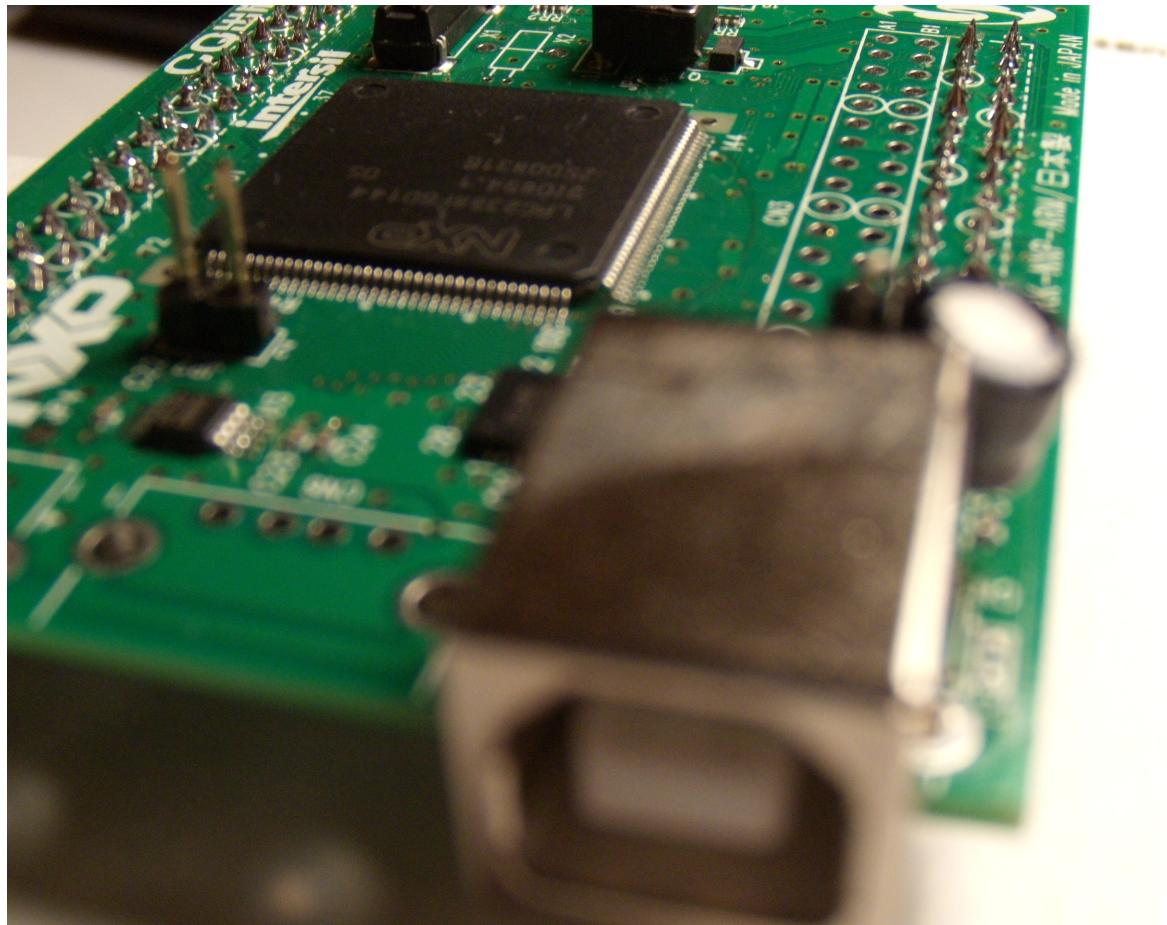
この単語は個人でのものづくりをさしている。これは、上で述べてきたことが一般化し、個人の製作能力を飛躍的に上昇させ、様々なことを個人にとって可能にしてきたという事の当然の帰結として、出てきた概念で

ある。個人のアプリケーションの持つ可能性が無視できなくなってきたとき、”Prosumer”的な先駆者が自然に発生し、あとから「第三の波」がやってきたといって過言ではないだろう。

当然の事ながらインターネットの高速化、リッチソフトウェア化、利用人口の増大は、その「波」を明らかに加速させている。

今回の純粋な技術的なヒントの半分ほどはインターネットを介して入ってきてる。特に、半導体のデータシートを個人が入手できるというのは革命である。

図2 接写ターゲット基板



第 III 部

内部仕様について

3 最初期の仕様

以下に全体構成を示す。(中心となるチップ / メーカ / ベンダ)

- カメラ (OV7670 / Omnipixel / aitendo) 2980 円
- マイクロプロセッサ (LPC2388 / NXP semiconductor / CQ 出版) 1980 円 (本込み)
- CPLD (EPM570T100C5 / Altera / optimize) 1600 円 (未実装)
- SRAM (CY7C1041DV33 / Cypress / 秋月電子通商) 500 円

上記の構成の理由について述べる。

3.1 カメラ -目玉-

映像を電気信号に変える素子。

3.2 プロセッサ -大脳-

ワンチップ・コンピュータ。プログラムを実行でき、Flash EPROM がプログラム領域なので、いくらでも書き換えができる。

3.3 CPLD -脊髄脳髄その他-

プログラマブルなロジック IC 群。プロセッサとおおきくちがうのは、”手順”をプログラムするのではなく、回路そのものを組み込むことである。動作のシミュレーションができる。クロックに同期しない動作が可能であるので、低速であっても、メモリ操作などができる。後述の HBE や LBE を素早く制御するのに、上記のプロセッサでは役不足である。

3.4 SRAM -海馬。中期記憶-

SDRAM はコマンドがある。RAM に一度カメラからのコンスタントな情報をプールするのは CPLD になるので、実質制御は CPLD になろう。すると、ロジック IC でしか処理出来ない速度で、SRAM 以上の複雑なコマンドを送るのは、初めての RAM 制御としては酷であると判断した。

よって SRAM を使用する。

SDRAM は価格が比較的安く、入手性も高いが、今回は簡単に実現できる事を優先させていただく。

さらに、なぜこのチップを選択したかというと、実はカメラのデータアウトプットと深い関係がある。
カメラのデータバス幅は 8bit である。しかし、1pixel のデータは 16bit、そう、1 データ集合を 2 回に分けて送信してくるのだ!!

そこで、16bit 幅の普通の RAM で簡単に制御しようとすると、実際のカメラからのデータの 2 倍のデータ

容量が必要となる。

必要となる RAM 容量 =

アドレス長 × 1 アドレスに於けるデータ幅 = *vertical* × *horizontal* × 2 × 8bit

16bit 幅の RAM を同じ方法で (1 回受信し 8 シフトして次回受信分と足すという大変な作業無しで) 扱うときの必要とされる RAM 容量 =

アドレス長 × 1 アドレスに於けるデータ幅 = *vertical* × *horizontal* × 2 × 16bit

となり、2 倍の RAM 容量が必要なのは自明だ。

しかもこの方式では、プロセッサが RAM から値をとるという簡単な行為のために 2cycle も浪費しなければならない。残念である上に、連結処理に (Green が上位 bits と下位 bits に分かれているため) 時間を割く必要がある。手間も割く。しかし、この SRAM のデータシートを読むと 16bit の SRAM でありながら、下位 bit、上位 bit のゲートの開け閉めが任意に行えるため、上位 bit に関する値だけ取り込みたい！といったあと、下位 bit だけまた同じアドレスに取り込みたい！といったわがままが効くのだ。

しかも、上位も下位も開けておけば 16bit いっぺんに読み込み書き込み可能なのである。

It's fits into just my needs!! 引用開始。資料 2*****

To write to the device, take Chip Enable (\overline{CE}) and Write Enable (\overline{WE}) inputs LOW. If Byte Low Enable (\overline{BLE}) is LOW, then data from IO pins (IO0 through IO7) is written into the location specified on the address pins (A0 through A15). If Byte High Enable (\overline{BHE}) is LOW, then data from IO pins (IO8 through IO15) is written into the location specified on the address pins (A0 through A15).

To read from the device, take Chip Enable (\overline{CE}) and OutputEnable (\overline{OE}) LOW while forcing the Write Enable (\overline{WE}) HIGH. If Byte Low Enable (\overline{BLE}) is LOW, then data from the memory location specified by the address pins appear on IO0 to IO7. If Byte High Enable (\overline{BHE}) is LOW, then data from memory appears on IO8 to IO15. See the “Truth Table” on page 9 for a complete description of read and write modes.

引用終了。また、文中に参照しろ！と書いてある Truth Table(真理表)

Truth Table

\overline{CE}	\overline{WE}	\overline{OE}	\overline{BHE}	\overline{BLE}	Input/Output	Mode
H	X	X	X	X	HighZ	Deselect
						PowerDown
L	X	X	H	H	HighZ	Output Disabled
L	H	L	L	L	Data Out (IO0-IO15)	Read
L	H	L	H	L	Data Out (IO0-IO7); IO8-IO15 in High Z	Read

L	H	L	L	H	Data Out (IO8-IO15); Read IO0-IO7 in HighZ	
L	H	H	L	L	HighZ	Output Disabled
L	H	H	H	L	HighZ	Output Disabled
L	H	H	L	H	HighZ	Output Disabled
L	L	X	L	L	Data In(IO0-IO15)	Write
L	L	X	H	L	Data In(IO0-IO7); IO8-IO15 in High Z	Write
L	L	X	L	H	Data In(IO8-IO15); IO0-IO7 in High Z	Write

4 現在の仕様

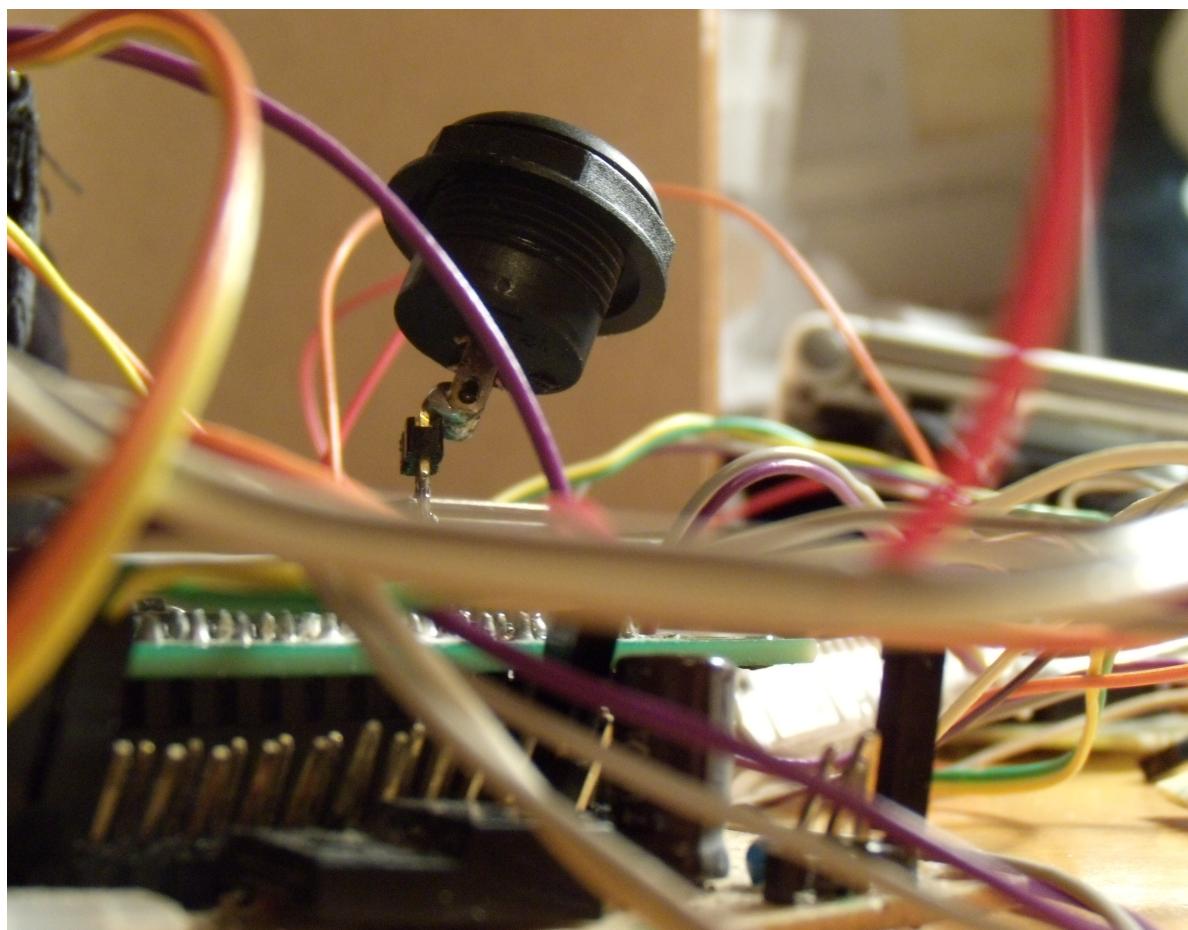
4.1 SRAM の破棄

SRAM を使用することに関して以下のデメリットがあった:

- SRAM 自体のサイズは小さいが、変換基板が無用に大きく、配置が手間である
- ピン数が多いために、ユニバーサル基板上に配置し、ピンを通じてコードで配線をすると、安定性が著しく低下する
- データ入出力層が SRAM に対してマイコン及び CPLD につながることとなり、短絡防止の保護回路を別に必要とする

また、後述する通り、作成に伴って SRAM を必要としなくても（信号データを一度 RAM 展開しなくても）データ抽出ができるようになったため、いらなくなつた。

図3 うつくしいもの



第 IV 部

開発環境

この開発環境の整備が、ファストプロトタイピングや、とくにパーソナルファブリケーションの大きなネックとなる。これが改善されなければ、今後の組み込みにおける、オープンソース環境での開発において大きな損失を招くであろう。

以下に、どのような開発環境の整備が簡易にでき、また、何が難しいか、不可能かを述べる。

5 オープンソースデスクトップ環境において

このサブセクション内で、一般に、ソフトウェアと言った場合、オープンソースであることとする。

また、デスクトップ環境とは、いわゆる PC での環境のことを指し、ここではホストと同意義とする。

『ホスト』は組み込み開発の『ターゲット』に対する用語とし、組み込み開発の母艦というニュアンスを含む。

プロプライエタリ (私企業や個人、政府が保有し、OSS でない) ソフトウェアの場合は配布企業とプロプライエタリであることを明示する。

オープンソースデスクトップ環境 (以降 OSSDE) として、以下の 3 つの OS を対象とした。

- Debian GNU/Linux(Debian project)
- OpenSuSE (OpenSuSE project)
- OpenBSD (OpenBSD)

また、オープンソースではないデスクトップ環境 (以降 non-OSSDE) として、以下の 2 つの OS を対象とした。

- Windows 7 リリース候補版 (マイクロソフト)
- Windows XP professional(マイクロソフト)

OSSDE として使用した OS はいずれも UNIX 互換プラットフォームであり、強力なコンソールを備える (その他にも十分に便利な X windows system を備える)。対して、windows では、コマンドプロンプトの機能は貧弱であるといわざるを得ない。

これは Cygwin(後述) などといったソフトウェアを利用することにより補完されるものの、パッケージ管理などといった大きな OSSDE 側のアドヴァンテージを解消することはできない。

パッケージ管理システムとは、UNIX互換の環境などにおいて存在する機能で、OS(正確には開発/配布者)ごとに存在する。

たとえば Debian の場合、`dpkg` というソフトウェアがパッケージを管理し、インストール、依存関係の確認、リムーブ、バージなど管理を行う。パッケージとは、ソフトウェアについて、作者による定義ファイルやバイナリファイル、場合によってはソースコードなどインストールに必要な情報が入った圧縮ファイルである。一般にパッケージ管理システムはこれだけの機能しかないわけではない。

Debian では `apt` というソフトウェア群が、あるソフトウェアの導入に伴って、依存関係の洗い出しを行い、必要なソフトウェアを自動でインストールし、不要になれば自動的に削除する。

これにより、さまざまな(開発が私企業でなく、個人が参加しているため本当にさまざまな)ライブラリが存在するにもかかわらず、使用者はライブラリを意識することなくただ目的のソフトウェアにありつけるし、開発者としても依存するライブラリとそのヴァージョンについて定義しておけば、利用者は必ずこのライブラリを持っているとみなせるので、自分が開発したいソフトウェアのみに専念することができる。

今回の組み込みに対する開発では、ホスト側にとっては一利用者に過ぎないことになるが、これらの OSSDE の特徴は、まさに `prosume` という風潮に適したシステムだと言える。

また、強力なコンソールを備えている事は、開発において、自動化したい部分をスクリプト化することで能率をあげることもでき、また、ウィンドウを利用してマウス操作でディレクトリの移動やソフトウェアの起動を行うよりも素早く、なおかつ開発に関係の無いことに患わされることが無いことを意味する。

これについてはさらに後述する。

まず、開発環境の導入の最初の一歩はインストールである。開発環境のインストールについて、比較する。

5.1 ARMマイコンのコンパイラ

一般に、ARMマイコンにおいて、もっとも安定しているコンパイラは `arm-none-eabi` であるとされている。

これは CodeSourcery 社によって開発されているもので、GUI(グラフィカルユーザインターフェース: ウィンドウとアイコンによって提供される環境)を備えた製品と、それに対するサポートを販売しているが、コンソールで扱うツールキットに関しては無料で提供している。このツールキットは、フロントエンドが GNU toolchain であり、これは非常に有名かつ有用なツールキットであり、オープンソースソフトウェアである。

また、ほかの ARM コンパイラとして、以下のものと比較対象にした。自分のソースコードにもっとも馴染んだものが `arm-none-eabi` であり、大きな性能差を感じたわけではないことをここに記しておく。

- arm-elf-eabi ビルドに失敗。
- arm-eabi
- arm-elf

上 2 つは、Linux 上でのみ確認できた。また、arm-elf については OpenBSD のみで試してみた。

5.2 AlteraCPLD の開発環境

Altera の開発環境、Quartus II は windows,Linux どちらでも動作する。過去のバージョンでは Solaris で動作するものもあるようだ。

逆に言えば、マイコンと違って、CPLD の開発環境は完全なブラックボックスであるので、windows のみのサポートでは不満があったのだと思われる。

Quartus II にかぎらず、Altera や Xilinx などの、PLD メーカがサポートしている Linux ディストリビューションは、Redhat 系と言われるものの中でも、RedHatEnterpriseLinux(RHEL) と CentOS(RHEL からサポートとロゴなどの著作権を除いた OSS な OS)、および OpenSuSE のみである。

ところが、Debian で動作させることができが一般に可能であり、使う上で、特に問題があるわけではない。唯一ライブラリの問題があるが、古いライブラリを持ってきて展開すれば使用できる。

また、OpenSuSE では、書き込み機である USB-Blaster の認識が不安定で、一般的な問題ではないようだが、私が使った上では、まったく使い物にならなかった。むしろ、サポート対象ではないはずの Debian での使用の方が快適である。それには以下のようにする。

```
#echo /dev/usbfs      /proc/bus/usb    usbfs    devmode=0666      0      0 >> /etc/fstab
```

5.3 その他

5.3.1 Make

UNIX 系 OS でのコマンドラインによる快適な操作と相まって、Make というのはすばらしいソフトウェアである。

Make というのは、いくつかのソースファイルがあった場合に、差分のみコンパイルすることができて、高速にソフトウェアをビルドできる。また、オプションによって動作を変えることが出来るため、役に立つ。

5.3.2 git

git というのはバージョン管理システムである。Apple の TimeCapsule などは有名だが、同じように、一里塚過ぎたら（過ぎなくともかまわないが）Commit することによって、あらたなバージョンとして登録される。もし、前のバージョンに戻したい場合も、またブランチ（枝分かれ）する場合も、破壊的なことを一切せずに要求をこなすことができる。これは OSS にとって非常に大きな意味を持つ。なぜなら OSS では、ある人が創っているソフトウェアを元に、別のソフトウェアを派生させる、ということが日常的に行われる。そのためにソースコードを公開しているといつても過言ではない。バグフィックスであれば元のソフトウェアに取り込まれるかもしれないし、別の方針をこれから歩んでいこうということもある。たとえば便利さやリリースの速さを優先したり、セキュリティを優先したりするためにブランチしたプロジェクトなど枚挙に暇がない。

他にも、たとえば操作ミスや思い直した事があって論文の身に何かあっても、落ち着いて過去のバージョンのファイルを取り出すことができる。

6 Windows 環境において

6.1 ARM の開発環境

- IAR workspace(nonOSS:IAR systems)

IAR systems の製品であるが、ライブラリの名前が嫌で、しかも見た目がすきじゃなかった。使い辛い。

使用可能な規模に制限がある。

6.2 AlteraCPLD の開発環境

Quartus II をなんら問題なく使うことができるが、フォントが汚い。これは windows に起因する問題で修正ができる。

このように、windows で使う開発環境として問題になることはほとんどなく、素晴らしいように見えるが、git を使うにも、ちゃんとしたコマンドラインを使うにも、不思議なほど手間がかかる。

7 git と github について

git というのは、分散バージョン管理システムである。つまり、そのリポジトリが存在する場所が 1箇所に限らないということである。それではバージョン間で矛盾しないの

か？それは、中央サーバにただ一つのマスタリポジトリを置くことによって解決される。つまり、git にはサーバにリポジトリをアップ（以降 push とよぶ）する機能がついている。そこで、無料で利用できる、データの損失の心配の無いクラウドに push できればこれほど安心なことはない。

さて、その要求を満たすサービスがある。それは github という。ではどうやって使うのかというと、rsa 暗号鍵を用意した後、サイトにアクセスし、リポジトリ作成をリクエストする。

新しいリポジトリの作成

プロジェクト名

説明 (optional)

ホームページのURL (optional)

メモ

注意: すでにGitHub上にある別の! プッシュしたい場合は、ここで新 fork してください。

このリポジトリにアクセスできるのは誰ですか? (あとから変更することもできます)

誰でも (公開リポジトリについてさらに学ぶ)
 プランをアップグレードすれば、非公開のリポジトリも作れるようになります!

リポジトリを作る

Blog サポート 研修 求人情報 ショップ 連絡先 API ステータス
© 2011 GitHub Inc. All rights reserved. 利用規約 プライバシーポリシー セキュリティ

Powered by the
Cloud Computer
rockspace

日本語 English Deutsch Français Português (BR) Русский 中文 その他すべての言語を見る ➔

その後、以下のようにコマンドを実行する。

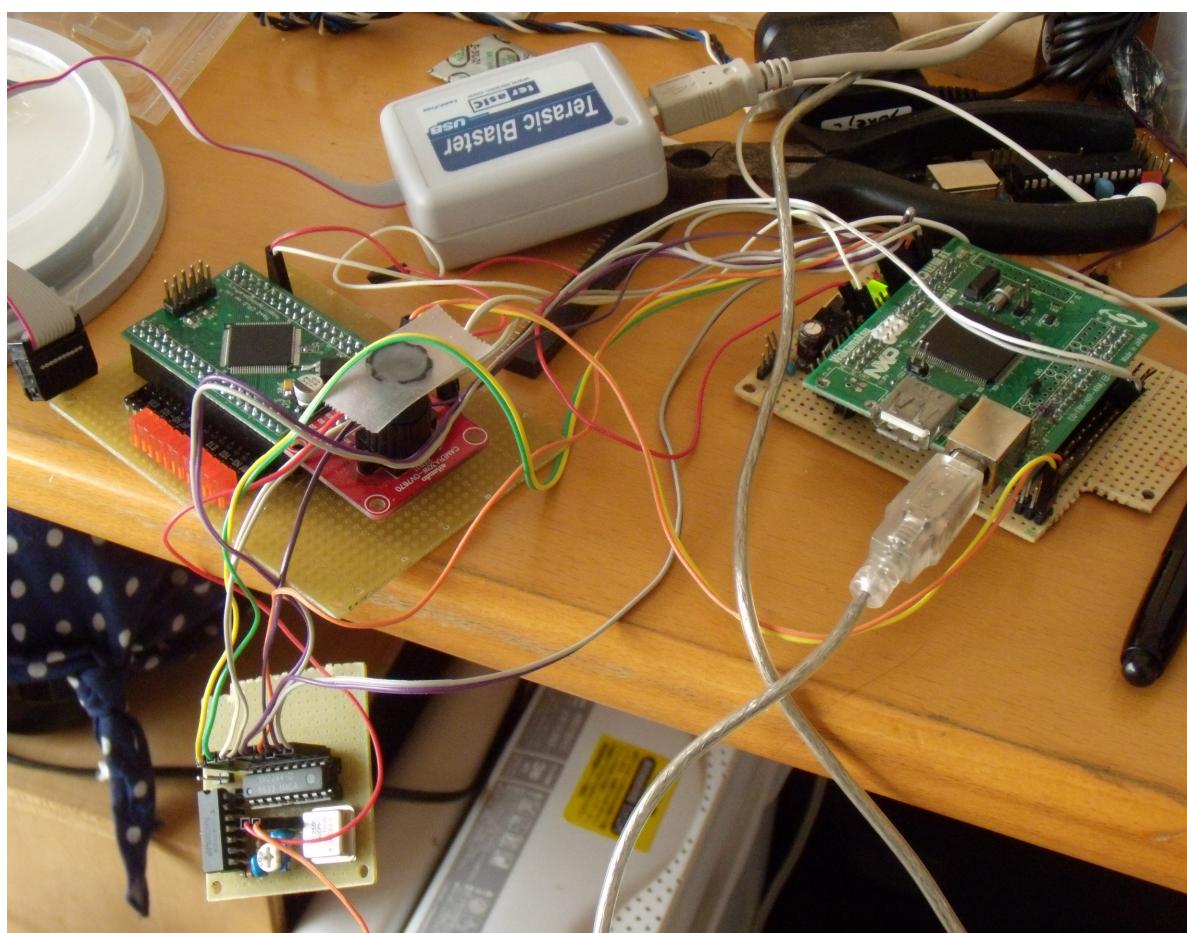
```
Debian% cd 論文
Debian% gedit textest.tex&
[1] 16628
Debian% ls
Makepdf  textest.aux  textest.dvi  textest.log  textest.out  textest.pdf  textest.tex
textest.tex~  textest.toc
Debian% git init
Initialized empty Git repository in /home/ioriveur/論文/.git/
Debian% git add ./
Debian% git commit -a
```

```
[master (root-commit) 10c4ed0] First Commit
 8 files changed, 950 insertions(+), 0 deletions(-)
 create mode 100755 Makepdf
 create mode 100644 textest.aux
 create mode 100644 textest.dvi
 create mode 100644 textest.log
 create mode 100644 textest.out
 create mode 100644 textest.pdf
 create mode 100644 textest.tex
 create mode 100644 textest.tex~
 create mode 100644 textest.toc
Debian% git remote add origin git@github.com:iori-yja/Report.git
Debian% git push origin master
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 110.86 KiB, done.
Total 11 (delta 2), reused 0 (delta 0)
To git@github.com:iori-yja/Report.git
 * [new branch]      master -> master
```

気がつくべきなのは、これは、既に公開され、オープンソースソフトウェアとなったということだ。

ソフトウェアを創れるならば、OSSとして育てていくことはまったくコストでは無いということが分かっていただけたと思う。

図4 混迷を極める開発室



第 V 部

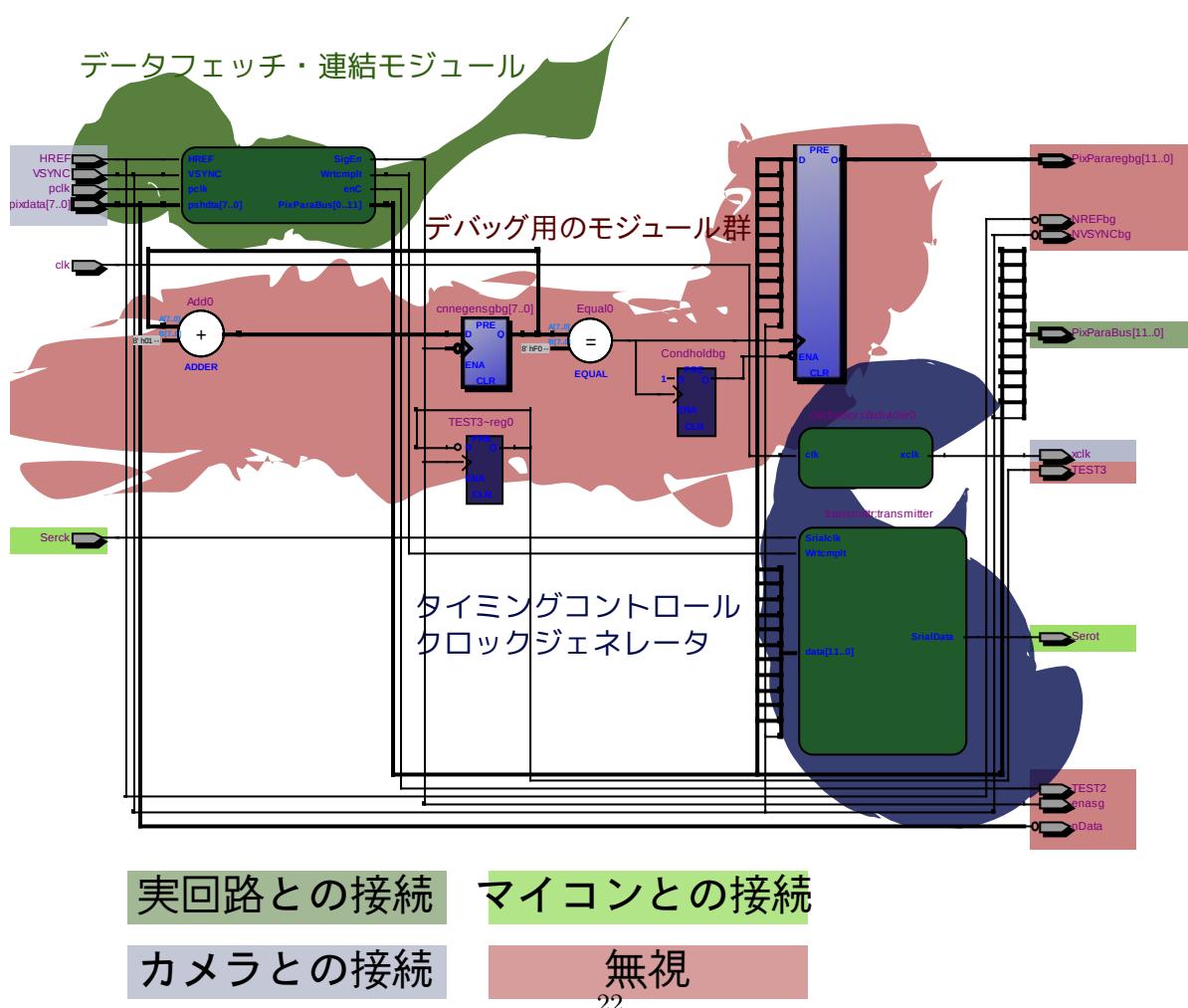
Verilog を書く

Verilog を書くのは、この論文がきっかけであった。
であるから、問題が起きるとしたらここだろうなという気がしていたが、まさにそのような事態はおきました。この場を借りて、辛抱強く待ってくださった本村先生に感謝の意をささげます。

8 モジュールの構成

このモジュールの全体像を示す。

図 5 モジュールの全体図:拡大できます



9 clkdivider(分周器)

まず初めに、もっとも簡単なモジュールを作成する。

今回、MAX II の基板に搭載した水晶発信器が 40MHz である。今回の用途ではカメラの動作周波数は動作定格内であれば遅い方がより信号バスがノイズに強くなるため、10MHz に分周してカメラのクロックソースに入力する。分周するメリットは、さらに、分周することによって、クロックのデューティ比を 50% に近くすることが可能である。

つまり、水晶発信器が必ずしも理想的な出力をしているとは限らないが、少なくとも 1 サイクルの長さはほとんどかわっていない、とするとき、分周器のクロック出力が H になっている時間と L になっている時間は、クロック入力のデューティ比に関わらず、どちらも入力 1 サイクル分であるから、出力波形は高速動作をするプロセサのクロックソースとして適したものとなる。

リスト 1 分周器

```
1 module clkdivider (input clk, output xclk);
2   reg [1:0] t_count = 0;
3   always@(posedge clk) t_count[0] <= ~t_count[0];
4   always@(posedge t_count[0]) t_count[1] <= ~t_count[1];
5   assign xclk = t_count[1];
6 endmodule
```

今回は、2 つのレジスタを設け、入力クロックの立ち上がりエッジに、t_count[0] を反転させる。すると、t_count が立ち上がるのは 2 入力クロックサイクルに 1 回となる。t_count[1] は t_count[0] が立ち上がりクロックになる時に反転するため、結果として 4 入力サイクルにつき 1 サイクルで blink することがわかる。
よってこのモジュールは 4:1 分周器になっている。

10 データ抽出レイヤ

10.1 ピクセル選択モジュール

リスト 2 特定のピクセルのサイクルのときに立ち上がる、一部省略

```
1 module TimingManager(
2   input VSYNC,
3   input HREF,
4   input pclk,
5   output reg Sig_En,
6   output enC);
```

```

7 reg [8:0] line = 0;
8 reg [8:0] foo = 0;
9 always @(posedge HREF or posedge VSYNC)begin
10   if(HREF == 1'b1)line <= line +1'b1;
11   else line = 8'h00;
12 end
13 assign enC = (line==9'h30 ) ? 1'b1 : 1'b0;
14
15 always @(posedge pclk)begin
16   if ( HREF&enC ) foo <= foo + 1'b1;
17   else foo <= 7'b0000000;
18   if(foo[0]&foo[8]&HREF)begin
19     Sig_En<=1'b1;
20     foo <= 7'b0000000;
21   end
22   else begin
23     Sig_En<=1'b0;
24   end
25 end
26 endmodule

```

このモジュールでは、カメラが生成する制御信号をすべて取り扱っていて、ほかのモジュールと分け合っていたりしないため、タイミング系で不具合が発生した場合はここに問題がある、と比較的迅速に問題の特定が出来る。

まず、簡単に OV7670 の制御信号について説明する。

CMOS カメラ OV7670 モジュールの外に出ている制御信号は

- VSYNC
- HREF
- pclk

のたった 3 つである。また、カメラ内に止まっている、普段引き出されないものに HSYNC があるが、基本的に使わずにすむようだ。

フレーム (1 つの画像) の最初の行の最初のピクセルを送出する前に、VSYNC が立ち上がり、3 行分の時間ホールドされる。その間、有効なデータは流れきていないが、なんの値が出ているのかは未だ不明であり、タダの不定値である。

その後、VSYNC が立ち下がり、実データが送出される条件が一つ揃うようになる。

次に、17 フレーム分の時間、制御信号のバスに変化は無い。その後、HREF が上がると、ようやく実データが送出され始める。1 行がおわると、一度 HREF が立ち下がり、144px 分やすむ。

その間、データシートによると、実データはキューに入れられて待っているらしい。

まず、このソースの目につくのは、pclk 立ち上がりエッジで起動する以下のカウンタ

リスト 3 ピクセルごとのカウンタ

```

1 always @(posedge pclk)begin
2     if ( HREF&enC ) foo <= foo + 1'b1;
3     else foo <= 7'b0000000;
4     if(foo[0]&foo[8]&HREF)begin
5         Sig_En<=1'b1;
6         foo <= 7'b0000000;
7     end
8     else begin
9         Sig_En<=1'b0;
10    end
11 end

```

であろう。しかし、このカウンタは、中の if 文の通り、HREF と enC が H でなければ動かない。また、if を満たさない場合は 0 にリセットされる。

HREF が L のとき、無効なデータがやってくる。だからこのデータを手っ取り早く見ないようにするためにには、カウンタをリセットして、0 のときに少なくともデータを見ない様にすればいい。

ただ、enC とはなんだろうか。これは、

リスト 4 行のカウンタ

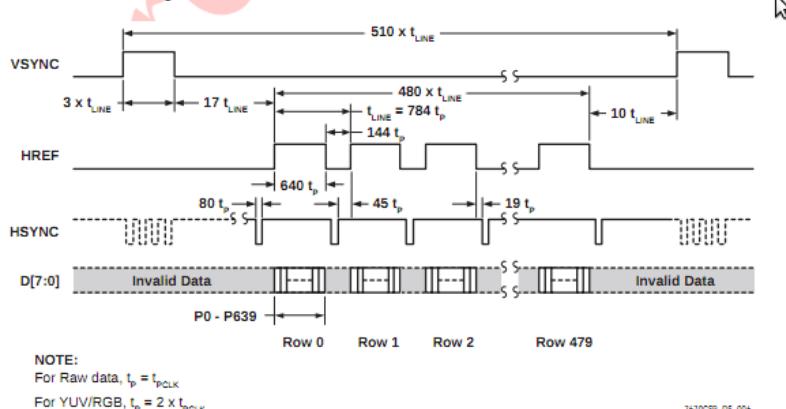
```

1 always @(posedge HREF or posedge VSYNC)begin
2     if(HREF == 1'b1)line <= line +1'b1;
3     else line = 8'h00;
4     end
5     assign enC = (line==9'h30) ? 1'b1 : 1'b0;

```

図 6 OV7670 のデータシートによるタイミングチャート

Figure 6 VGA Frame Timing



でアサインされているワイヤが enC であり、line(行数) の値によってドライブされる。つまり、特定の行の時のみピクセルカウンタが起動される様になっているわけだ。

実を言うと、この設計にたどり着くまで、もっと別の設計でやろうとしていた。最初は一つの always でやっていたが、implicit latch を生成しますよ！っていうワーニングや、always@文で呼び起こされたのに何も値をえないことがあります、といったワーニングが出ていて、調べても何の事だかよく分かっていなかった。

もし、この設計を思いついていなかったら、ずっと気持ち悪いままだったが、これによってワーニングが消えたどころか、使用するロジックエレメントの数もグッと減ったため、きっと裏でなにかあって、always@文が何のレジスタの値もえない可能性があると、無駄なラッチを生成することがある様だ。

ということがわかったが、具体的なバッドノウハウもリゾルブノウハウもほとんど出回っていないという状態であるため、今後のワーニングメッセージの改善や、教本、ノウハウ集として有用なブログなどの登場を期待する必要があると感じた。

このためには、半導体ベンダ側の努力のみならず、企業のような、ノウハウが閉鎖的にこもりがちな顧客だけ、という状態から脱して、それこそ手芸やバレエのごとく、幅広い一般人が FPGA や CPLD などを使うようになる必要も、当然あるだろう。どれくらい未来になるか分からないが、IT がもっと陳腐化すればいずれ来る未来であると思う。

10.2 ピクセルデータフェッチ, 連結モジュール

リスト 5 データコミッタ

```
1 module Dtacutcmmit(
2     input SigEn,
3     input [7:0] pshdta,
4     output WrCmplt,
5     output reg [11:0] popdta
6 );
7     assign WrCmplt = ~SigEn;
8     reg [3:0] redpx;
9     always @(posedge SigEn)begin
10         redpx [3:0] <= pshdta[3:0];
11     end
12     always @(negedge SigEn)begin
13         popdta[11:0]<= { redpx[3:0], pshdta[7:0] };
14     end
15 endmodule
```

このモジュールは、SigEn の両エッジで働く。これもだいぶ悩んだ末に落ち着いた形であって、SigEn と言うのはピクセル選択モジュールである TimingManager のものと同じである。これは、上で述べた変更によって pclk の立ち上がりエッジでのみ動作するようになったため、Sig_En のエッジ動作時には、ピクセルデータが流れている pshdta は必ず保証されている。これは上のタイミングチャートを見てほしい。pclk 立ち下がりエッジのときにデータ値が変化し、逆に立ち上がりエッジの前約 15ms 以上と後 8ms が保証されている。(ただし 24MHz 動作時、今回はさらに 2.4 倍遅い)

11 データ転送レイヤ

```
1 module transmittr (
2     input Srialclk,
3     input Wrtcmplt,
4     input [11:0]data ,
5     output SrialData
6 );
7 reg [11:0]Datareg;
8 reg Endflg;
9 wire Endtoken;
10 assign Endtoken = Endflg & ~Srialclk;
11 assign SrialData = Endtoken ? ~Datareg [0] : Datareg [0];
12 reg [3:0]ShftCount = 0;
13 always@( posedge Srialclk )begin
14     Datareg [11:0] <= { 1'b0, Datareg [11:1] };
15     ShftCount <= ShftCount + 4'h1;
16     if ( ShftCount == 4'hB )begin
17         Endflg <= 1'b1;
18         if( Wrtcmplt )begin
19             ShftCount <= 0;
20         end
21     end
22     else if ( ShftCount == 4'h0 )begin
23         Endflg <= 0;
24         Datareg [11:0] <= data [11:0];
25     end
26 end
27
28 endmodule
```

12 シミュレーション

12.1 シミュレータの意義

つぎに、モジュールが書けたら、マイコンでは実機で動かすことが一般的だが、PLD ではシミュレータにかけることが多い。

理由としては、まずマイコンよりもさらに可観測性が低いため、予期せぬ動作をした場合にデバッグが非常に大きな仕事となってしまうということ、C 言語でマイコンに対して書いたものとくらべ、人間にとって可読性が高いものではないため、まったく想定外のことを書いてしまっている可能性がより高いということ、シミュレーションを行う上では、マイコンの、特にペリフェラルでの動作と比べて Verilog の方が可視性があるということなどが挙げられるはずだ。

シミュレータは今回、Modelsim-altera starter edition(nonOSS:MentorGraphics 社)を用いた。

シミュレーションモデルも HDL を使って表記する。

論文の構成上、他のモジュールより後に書いたが、すべてのモジュールを書き終えてから回路検証をするのではなく、一つのモジュールを書きながらテストモジュールを書いて逐一懸賞をしていくことが何よりも大切だと思われる。

この方法であれば、検証対象モジュールとシミュレーションモデル、双方とも、バグを書いた時点できちんと分かる上、比較的見つけることが容易であり、最も効率がいいと思われる。

また、ModelSim でのコマンドは tcl であるため、セミコロンで連接することにより一気にビルドから波形を追加してシミュレーションまで終えてしまうことができる。

リスト 6 一気にシミュレーションまで行う 1 行スクリプト

```
1 vsim work.TEST;add wave -r /*;run 1600000;
```

もし、add wave しなければ、シミュレーションが行われた後もタイミングチャートをみることは出来ない。また、ModelSim の GUI は貧弱で、それのみに頼るには不足している印象を受けた。

図 7 ソースコードを開いた ModelSim Altera Starter Edition

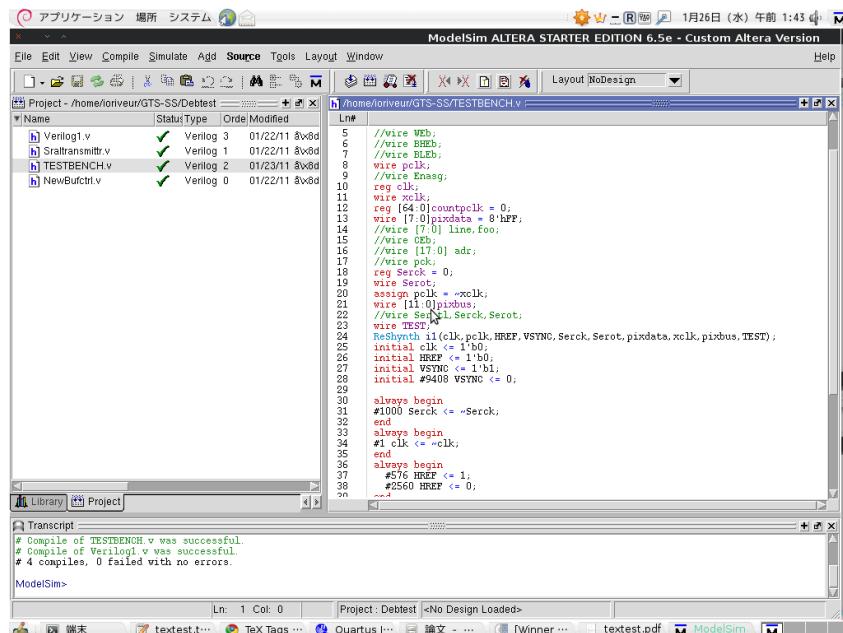
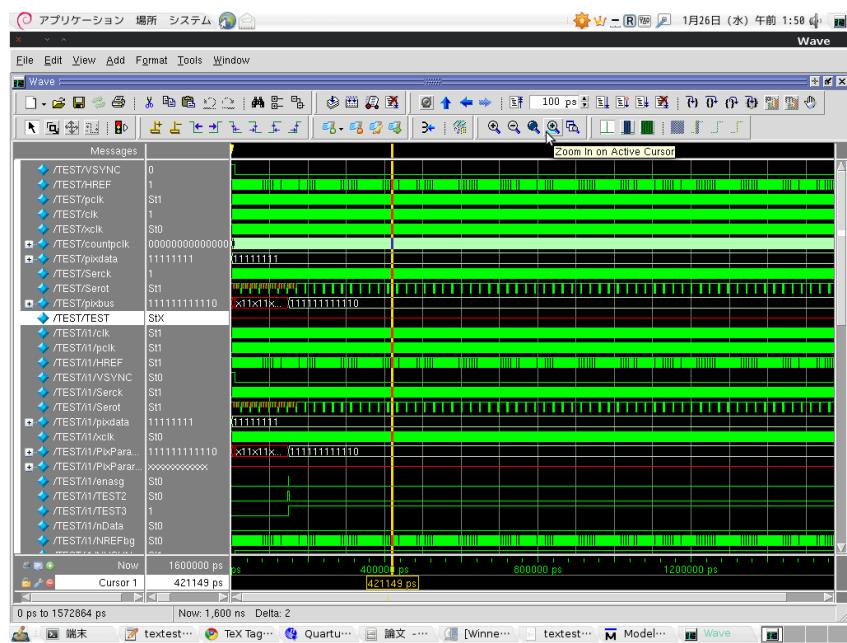


図8 シュミレーションを終え、タイミングチャートを表示する ModelSim Altera Starter Edition



12.2 シミュレーションモデル

他の、実回路で動かすための Verilog と違い、シミュレーションではさまざまな便利な命令を使うことが出来るが、私が思うに、これらを一まとめにして Verilog と呼んでいる現状が、現在の簡易的では無いが実装されるべき構文の、実回路での論理合成可能性がいまだに上がらないままである元凶であるように見える。

ちなみに、このシミュレーションモデルでは、コメントアウトした部分も、わざと掲載している。それは、消してしまうのではなく、コメントアウトすることによって、また後で検証対象モジュールのバスを観測したくなったときにすぐ戻せるようにするという技を明示的に示すためだ。

リスト 7 回路検証のためのモジュール. 実回路には組み込まれない

```
1 module TEST;
2 reg VSYNC;
3 reg HREF;
4 //wire WEB; これら辺は外付けの SRAMを使う予定だったころの名残りが残っている
5 //wire BHEb;
6 //wire BLEb; always @(posedge pclk)begin
7   if ( HREF&enC ) foo <= foo + 1'b1;
8   else foo <= 7'b0000000;
9   if(foo[0]&foo[8]&HREF)begin
10     Sig_En<=1'b1;
11     foo <= 7'b0000000;
12   end
13   else begin
14     Sig_En<=1'b0;
15   end
16 end
17 wire pclk;
18 //wire Enasg; これはピクセル選択モジュールのインプルシグナルの観測用
19 reg clk; //40MHzのマスタクロック
20 wire xclk;
21 reg [64:0] countpclk = 0;
22 wire [7:0] pixdata = 8'h46;
23 //wire [7:0] line,foo; バスの観測用
24 //wire CEB;
25 //wire [17:0] adr;
26 //wire pck;
27 reg Serck = 0;
28 wire Serot;
29 assign pclk = ~xclk;
30 wire [11:0] pixbus;
31 //wire Serctl,Serck,Serot;
32 wire TEST;
```

```

33 ReShynth i1(clk,pclk,HREF,VSYNC,Serck,Serot,pixdata,xclk,pixbus,TEST);
34 initial clk <= 1'b0;
35 initial HREF <= 1'b0;
36 initial VSYNC <= 1'b1;
37 initial #9408 VSYNC <= 0;
38
39 always begin
40 #1000 Serck <= ~Serck;
41 end
42 always begin
43 #1 clk <= ~clk; //マスタクロック生成
44 end
45 always begin
46 #576 HREF <= 1;
47 #2560 HREF <= 0;
48 end
49 always begin
50 #1589952 VSYNC <= 1;
51 #9408 VSYNC <= 0;
52 end
53 always @(negedge pclk)
54 if(VSYNC) countpclk <= 0;
55 else
56 countpclk <= countpclk + 1;
57 always @pixbus $display($time, "pix:%d countpclk is %d", pixbus,countpclk);
58 endmodule

```

このテスト用モジュールは、主に

- クロックソースの流し込み(マスタクロック, シリアルクロック, ピクセルクロック)
- カメラの制御信号の生成、流し込み
- バスデータの仮想的な提供
- 特定の観測しているワイヤに変化があった場合のモニタリング

を行っている。

たとえば

```

1 always begin
2 #576 HREF <= 1;
3 #2560 HREF <= 0;
4 end
5 always begin
6 #1589952 VSYNC <= 1;
7 #9408 VSYNC <= 0;
8 end

```

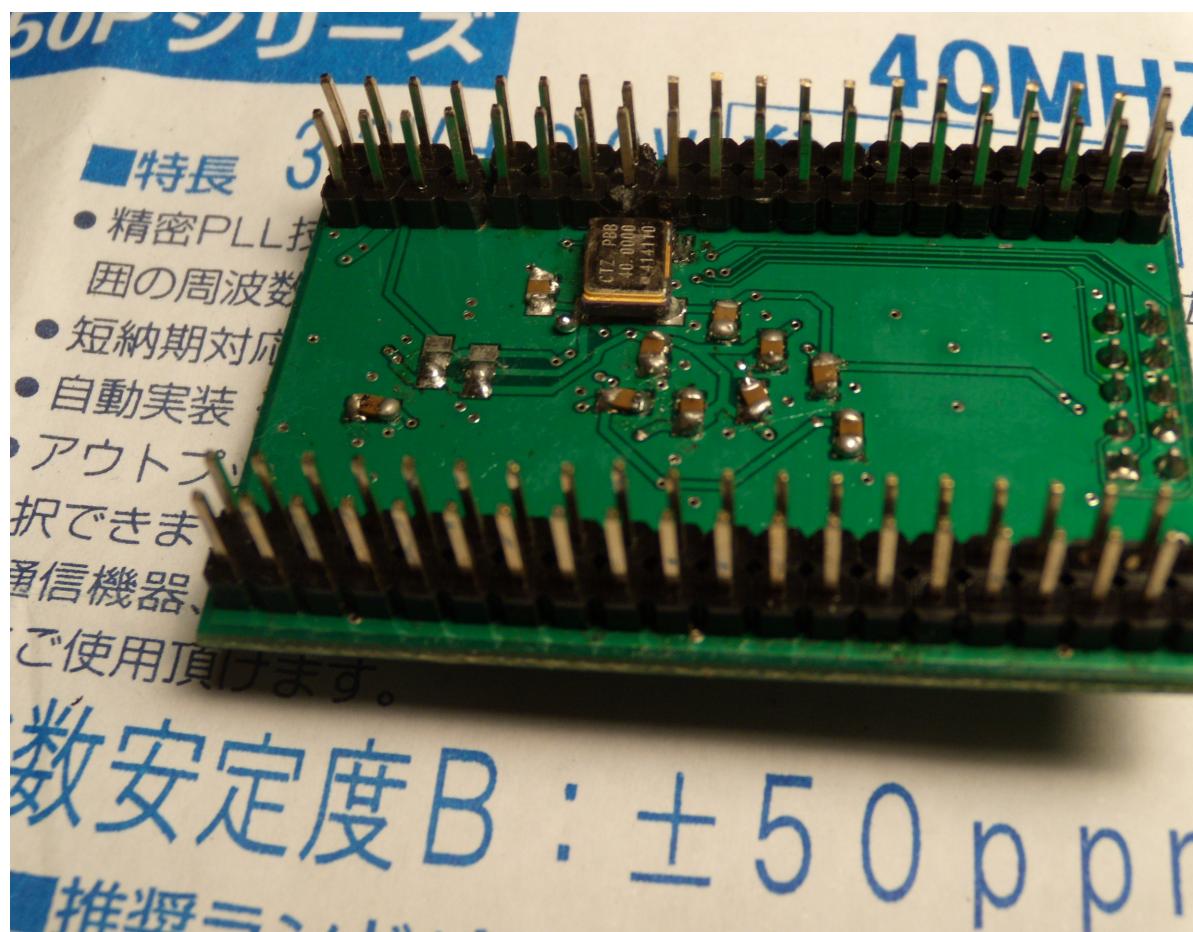
ここ下りでは、カメラから送られてくる制御信号を提供しているが、これはカメラのデータシート(

[http://aitendo2.sakura.ne.jp/aitendo_data/product_img2/product_img/camera/CAMERA30W-V7670/OV7670_DS_\(1.4\).pdf](http://aitendo2.sakura.ne.jp/aitendo_data/product_img2/product_img/camera/CAMERA30W-V7670/OV7670_DS_(1.4).pdf)

)を参考することによって比較的早くできあがっていた。

逆に、テストモジュールを書きづらいのは、仕様が決まってない自分の回路にたいしてであったりするため、侮れない。

図9 CPLD ボードの裏側の水晶発振子とそのデータシート



第 VI 部

I²C セットアップ関数の開発

I²C ドライバが必要だったため、作成した。割り込みを使わない *I²C* で必要となる動作が、2 番ポートにおいてすべてカヴァーされているため、ライブラリとして公開する。ただ、このコードを書いたのがずっと昔で、今回だましだまし使っていたので、とても汚いコードとなっている。

そんな汚いコードを公開するのか？という疑問もあるが、そもそもなにもないよりマシであるという考え方もある。

13 データ送出関数

```
1 void i2csender(int Continue, unsigned int Data, int Keta){  
2 //int i2cStatus;  
3     if(Continue==0){  
4         I22DAT = Data;  
5         I22CONSET |= 0x04;  
6         I22CONCLR = 0x08;  
7         while(I22STAT!=0x18 && I22STAT!=0x20);  
8         FIO2PIN1 = 0x2;  
9         if(I22STAT==0x20){  
10             printf("No such device%4x\n",Data);  
11             i2cErr = 0x22;  
12             return;  
13         }  
14     }else {  
15         for(;Keta != 0;Keta-=8){  
16             I22DAT = (Data & 0xFF);  
17             I22CONCLR = 0x28;  
18             while(I22STAT!=0x28);  
19             Data = Data >> 8;  
20         }  
21     }  
22 }
```

13.1 動作

この関数の動作は、ライトサイクルとして最初であった場合には、まず割り込みをクリアし、マスタモードに入らないようにセットアップし、また、”データ”としてアドレスを送出させるところから始まる。その後、デバイスから返答があるか、どこからも返答が無いと分かるまで待ち、終了処理を行う。

ライトサイクルとして最初では無かった場合は、int 値を、下 8bit ずつ送出する。

13.2 問題点

これを FreeRTOS 上で動作させると、初回呼び出し時の 3 回に 2 回、しかも周期的に OS 全体を落とすという大きな問題を抱えている。

14 シーケンシャルデータ受信関数

```
1 int i2creader(int size, int Adr, int registernumber){
2     char Loopy;
3     upper:
4         I22DAT = (Adr + 1);
5         I22CONSET |= 0x04;
6         I22CONCLR = 0x08;
7         printf("Status is %x\n", I22STAT);
8         while(I22STAT != 0x40 && I22STAT != 0x48) printf(".");
9         if(I22STAT == 0x48){
10             printf("no acknowlege(%x)\n", I22STAT);
11             i2crestart(2);
12             goto upper;
13         }
14         while(registernumber != (size+1)){
15             I22CONSET |= 0x04;
16             I22CONCLR = 0x28;
17             while(I22STAT != 0x50 );
18             vTaskDelay(10 / portTICK_RATE_MS);
19             printf("Adress%x,\tData%x", registernumber++, I22DAT);
20         }
21         printf("done.....");
22         fflush(stdout);
23         I22CONCLR = 0x0C;
24 }
```

これは、SCCB レジスタに対して、シーケンシャルリードを 0 番地から最大値である 0xca までを一期に読み取る関数である。あまり美しいコードにはなっていない。

14.1 バイトリード

サイズを 1 にすれば、バイトリードとなる。本質的に、シーケンシャルリードとバイトリードに違いは無いのだ。

15 OV7670 全リード関数

OV7670 の SCCB レジスタをリードする関数である。

```
1 void SCCBread(int subadr, int size, int NumByte){
2     i2cstart(2);
3 //    vTaskDelay(300 / portTICK_RATE_MS);
4     i2cErr = 0;          //i want to read! b So, start.
5     i2csender(0,0x42,8); //Hey, ?
6     if(i2cErr==0x22){
7         printf("fatal");
8         i2cstop(2);
9         return;
10    }
11    i2csender(1,subadr,8);
12 //    printf("Adress read request at%4x OK\n\n", subadr); //0x03byte?
13    i2crestart(2);
14    i2creader(size,0x42,NumByte);
15    i2cstop(2);
16    printf("Debu331g\n");
17 }
```

一つ目のコメントアウトは、安全性のために最初いれておいたものだが、無くても動くらしいと分かったためコメントアウトした。i2cstart(*intport*) 関数はそのままなのでここでは割愛する。

二つ目のコメントアウトは、はずすと冗長な出力となる。

これをみるとわかるとおり、リクエストの送信とデータの受信を一括して行う。

16 OV7670 バイトライト関数

```
1 int SCCBwrite(int subadr, int size, int Data ){
2     i2cstart(2);
3     i2cErr = 0;
4     i2csender(0,0x42,8); //Hey, ?
5     if(i2cErr==0x22){
6         printf("fatal");
7         i2cstop(2);
8         return -1;
9     }
10    i2csender(1,subadr,8); //3byte?
11    i2csender(1,Data,size);
12    i2cstop(2);
13    return 0;
14 }
```

第 VII 部

SCCB での OV7670 コンフィギュレーション

17 OV7670 の注意点

SCCB というのは、 I^2C というシリアル通信のスーパセットで、実質 I^2C と同じ方法で通信が出来るとされている。ただし、実際にやってみるといくつか違いがあった。

まず、 I^2C 接続の EEPROM での実験をしめす。今回使用したのは、microchip 社製 24LC256 である。

これに、バイトライト、バイトリード、シーケンシャルリード、シーケンシャルライトを試したが、バイトライトの後に wait をはさむ必要が無く、シーケンシャルライトも問題なく行えた。

これに対して、OV7670 では、書き込みの後に wait を必要としている。データシートには記述が無いものの、実験によるとシーケンシャルライトは出来ないようだ。このことから、完全な I^2C メモリインターフェースを積んでいるというよりは、あくまで I^2C インタフェースをつかった通信が出来ますよ、という程度に考えておくべきだ。

また、ただの RAM のつもりでヴェリファイプログラムを書くべきではない。理由は後述する。

18 OV7670 用のレジスタセットアップデータの整理

<http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/drivers/media/video/ov7670.c>
ここに、OV7670 用の linux ドライバがあるが、非常に不思議な事に、データシートには always "0" などとかかれた場所や reserved 領域に書き込んだりしている。ともかく、これをどれほど信用すべきなのかんがえあぐねているところに、FPGA で遊んでみるさんが既に試していた。

リスト 8 ov7670_set_up.ttl

```
1 ; OV7670<81>@set up macro
2 ;
3 ; delay set
4 pause_ms =0
5 ; delay for debug
6 ;pause_ms =2000
7 ;
8 ; ADR=0x12 WDATA=0x04
```

```

9  sendln  'W1204'
10 sendln
11 mpause pause_ms
12 ;
13 ; ADR=0x40 WDATA=0xd0
14 sendln  'W40D0'
15 sendln
16 mpause pause_ms
17 ;
18 ; ADR=0x8c WDATA=0x02
19 sendln  'W8C02'
20 sendln
21 mpause pause_ms
22 sendln  'W703a'
23 sendln
24 mpause pause_ms
25 sendln  'W7135'
26 sendln
27 mpause pause_ms
28 sendln  'W7211'
29 sendln
30 mpause pause_ms
31 sendln  'W73f0'
32 sendln
33 mpause pause_ms
34 sendln  'Wa202'
35 sendln
36 mpause pause_ms
37 sendln  'W1500'
38 sendln
39 mpause pause_ms
40 sendln  'W7A20'
41 :
42 :
43 :
44 :

```

これは Teraterm かなんかのスクリプトで、このままでは使えないで、vim で整形し、そのまま SCCB レジスタに書き込む関数にした。

```

1  SCCBByteWrite(0x1204);
2  vTaskDelay(2 / portTICK_RATE_MS);
3  SCCBByteWrite(0x40D0);
4  vTaskDelay(2 / portTICK_RATE_MS);
5  SCCBByteWrite(0x8C02);
6  vTaskDelay(2 / portTICK_RATE_MS);
7  SCCBByteWrite(0x703a);

```

```

8 vTaskDelay(2 / portTICK_RATE_MS);
9 SCCBByteWrite(0x7211);
10 vTaskDelay(2 / portTICK_RATE_MS);
11 SCCBByteWrite(0x40D0);
12 vTaskDelay(2/portTICK_RATE_MS);
13 SCCBByteWrite(0x8C02);
14 :
15 :
16 :
17 :
18 :

```

しかし、その後、関数内に定数を置いてしまうことによる、保守性、可読性、コード密度の低さなどの問題があり、これを解決するために分離することとなった：

リスト 9 i2c.c の vi2c タスクの冒頭

```

1 printf("Setup begin[>  ");
2 fflush(stdout);
3 for(regpointer=0;regpointer!=129;){
4     if(SCCBByteWrite(OV7670[regpointer])==-1)continue;
5     regpointer++;
6     switch (regpointer%4){
7         case 0:
8             Loopy='|';
9             break;
10        case 1:
11            Loopy='/';
12            break;
13        case 2:
14            Loopy='-';
15            break;
16        case 3:
17            Loopy='\\';
18            break;
19    }
20    printf("\b\b\b=>%c",Loopy);
21    fflush(stdout);
22    vTaskDelay(10 / portTICK_RATE_MS);
23 }
24 printf("\rData transmition End!\n");
25 fflush(stdout);

```

とし、

リスト 10 OV7670userconfig.h の抜粋

```

1 const int OV7670 []={
2 0x1204,

```

```
3 0x40D0 ,  
4 0x8C02 ,  
5 0x703a ,  
6 0x7211 ,  
7 :  
8 :  
9 :  
10 :
```

というヘッダファイルを設けた。これにより、ループを回して書き込むようになったとのと、設定ファイルを別ファイル扱いにできたので、修正が容易になった。
また、これにより、二次的なメリットが得られた。

```
1 Hello, world!1:0  
2 203PDIRRegister Reset  
3 Setup begin  
4 [=====> \
```

今までのメッセージはデバッグを目的とした冗長な出力がなされていたため、上の様なメッセージととした。簡潔であり、また十分な情報量である。

ひとつのノウハウとしては、バグが存在する間は冗長な出力が好ましいが、安定して動かせるようになったら静かな方が好ましい。

また、ヴェリファイは今回行っていない。なぜなら、制御用レジスタにまざって、動的に変化するレジスタ、例えばアイリス値などがあり、また、述べた通り不思議な事だが、なぜかそのようなレジスタにも値を書き込むようになっている。このようなレジスタがすべて分かれれば良いものの、Reserved や Always zero とかかれたレジスタにも値を書き込んでいるため、これらがまた、動的に変化するレジスタであつたらデバッグの難度が非常に高いと言わざるを得ず、また、すでに十分に動いていると言う現状から、個人では把握できないものとして考えるべきと言える。

メーカーはデータシートなど、デバイスを使用するに当たってどうしても必要な情報は大口顧客などにのみ提供するのではなく、万人が使えるものにするよう努めるべきである。

第 VIII 部

独自シリアル転送プロトコル

19 トランスマッタ

20 レシーバ

第 IX 部

関数値渡しの謎

ARM での関数の値渡しに疑問を持ち研究を行った。

FreeRTOS 上でにあるプロセスのサブルーチンが孫ルーチンを呼ぶ部分をまず見てみるとする。

子ルーチン:(思ったより長くなつたので抜粋)

リスト 11 子ルーチン

```
1 .LVL64:  
2 .loc 1 264 0  
3 mov ip, #0  
4 .loc 1 265 0  
5 mov r0, ip  
6 mov r1, #66  
7 mov r2, #8  
8 .loc 1 264 0  
9 str ip, [r4, #0]  
10 .loc 1 265 0  
11 bl i2csender
```

リスト 12 孫ルーチン

```
1 i2csender:  
2 .LFB5:  
3 .loc 1 134 0  
4 .cfi_startproc  
5 @ Function supports interworking.  
6 @ args = 0, pretend = 0, frame = 0  
7 @ frame_needed = 0, uses_anonymous_args = 0  
8 .LVL22:  
9 .loc 1 136 0  
10 cmp r0, #0
```

```

11 .loc 1 134 0
12 stmfd sp!, {r3, r4, r5, r6, r7, lr}
13 .LCFI1:
14 .cfi_def_cfa_offset 24
15 .loc 1 134 0
16 mov r4, r1
17 .cfi_offset 14, -4
18 .cfi_offset 7, -8
19 .cfi_offset 6, -12
20 .cfi_offset 5, -16
21 .cfi_offset 4, -20
22 .cfi_offset 3, -24
23 mov r6, r2
24 .loc 1 136 0
25 beq .L58
26 .loc 1 148 0 discriminator 1
27 cmp r2, #0
28 .loc 1 149 0 discriminator 1
29 ldrne r5, .L71
30 .loc 1 150 0 discriminator 1
31 movne r7, #40
32 .loc 1 148 0 discriminator 1
33 beq .L57
34 .LVL23:
35 .L68:
36 .loc 1 149 0
37 and r1, r4, #255
38 str r1, [r5, #8]
39 .loc 1 150 0
40 str r7, [r5, #24]
41 .L63:
42 .loc 1 151 0 discriminator 1
43 ldr r3, [r5, #4]
44 cmp r3, #40
45 bne .L63
46 .loc 1 152 0
47 ldr r0, .L71+4
48 mov r1, r4
49 bl printf

```

ここで、孫ルーチンの C 言語での記述を見てみよう。

リスト 13 孫ルーチン:C 言語

```

1 void i2csender(int Continue, unsigned int Data, int place){
2 if(Continue==0){
3 I22DAT = Data;
4 I22CONSET |= 0x04;
5 I22CONCLR = 0x08;

```

```

6 printf( " requesting\x% x ",Data);
7 while(I22STAT!=0x18 && I22STAT!=0x20)printf( " . ");
8 if(I22STAT==0x20){
9 printf( " No such device\x4x\n ",Data);
10 i2cErr = 0x22;
11 return;
12 }
13 }else {
14 for(;place != 0;place-=8){
15 I22DAT = (Data & 0xFF);
16 I22CONCLR = 0x28;
17 while(I22STAT!=0x28);
18 printf( " Data\x4x\n ",Data);
19 Data = Data >> 8;
20 }
21 }
22 }

```

呼び出し部:引数が3つある。i2csender(0,0×42,8); 0,0×42(0d66),8を投げている。もう一度子ルーチンを見る。

リスト 14 子ルーチンさらに抜粋

```

1 .LVL64:
2 .loc 1 264 0
3 mov ip, #0
4 .loc 1 265 0
5 mov r0, ip @引数1つめ、なんでうえのmov ip, #0をへるのか不明過ぎる、mov r0, #0ではだめなのか
6 mov r1, #66 @引数2つめ
7 mov r2, #8 @引数3つめ
8 .loc 1 264 0
9 str ip, [r4, #0]
10 .loc 1 265 0
11 bl i2csender

```

ということでどうやらARMでは3つの引数を渡すときにr0~r2に置くということが分かった。本当だろうか。最初にあるstmdfd sp!, r3, r4, r5, r6, r7, lrと最後にあるldmfd sp!, r3, r4, r5, r6, r7, lrを見る。つまり、スタックにr3~r7とlrを積んで、作業スペースを確保しているのだろう。

では次に、printf()のような文字列引数の場合、どうやって渡してるんだろうという疑問が沸く。レジスタ1こ文字列が收まりきるとは限らないからだ。

リスト 15 printf呼び出し:アセンブリ

```

1 .LCFI1:
2 .cfi_def_cfa_offset 24
3 .loc 1 134 0

```

```

4  mov r4, r1
5  .cfi_offset 14, -4
6  .cfi_offset 7, -8
7  .cfi_offset 6, -12
8  .cfi_offset 5, -16
9  .cfi_offset 4, -20
10 .cfi_offset 3, -24
11 mov r6, r2 @第3引数をr6に退避(r2が0だと0フラグが立つ)
12 .loc 1 136 0
13 beq .L58 @0フラグ立ってたら分岐
14 .loc 1 148 0 discriminator 1
15 cmp r2, #0
16 .loc 1 149 0 discriminator 1
17 ldrne r5, .L71 @I22DATのアドレス
18 .loc 1 150 0 discriminator 1
19 movne r7, #40 @0を第3引数にしてないので、r7に40を入れる。
20 .loc 1 148 0 discriminator 1
21 beq .L57
22 .LVL23:
23 .L68:
24 .loc 1 149 0
25 and r1, r4, #255
26 str r1, [r5, #8] @I22DATにいれます
27 .loc 1 150 0
28 str r7, [r5, #24] @I22CONCLRに0x28をいれる
29 .L63:
30 .loc 1 151 0 discriminator 1
31 ldr r3, [r5, #4] @I22STATをロード
32 cmp r3, #40 @それって0x28?
33 bne .L63 @違う間ループ、3命令かな?(実は.locがようわかっていない)
34 .loc 1 152 0
35 ldr r0, .L71+4 ここが問題のprintf()にたいする引数渡し
36 mov r1, r4
37 b1 printf

```

さて、L71+4 が何をさすのだろうか。

リスト 16 ラベル L71

```

1 .L71:
2 .word -536346624
3 .word .LC3
4 .word .LC1
5 .word .LC2
6 .word i2cErr
7 .cfi_endproc

```

ふむ。+4 ということは、きっと.LC3 だろう。(4byte = 32bit, まえのが.word だから 32bit でアラインされている) .LC3 にはなにがはいっているのであろうか。

リスト 17 ラベル LC3

```
1 .LC3:  
2 .ascii " Data\%4x$\backslash012$\backslash000  
3 .space 3
```

これを見るとどうやら.ascii というアライン方法があるらしいが、そんなものは見たことが無い。

ということで、データシートを開く。

https://dl.dropbox.com/u/15570814/051020DDI0100HJ_v61.pdf しかし載ってない。

あれ? とにかく、.ascii っていうアライン方法で、“ ASCII encoded string ”ってやってやれば、(多分アセンブラー) 対応するバイナリに変換して LC3 に置いているのだろう。

ところで、\012\000 ってなんだろう。とりあえず、\n を\t に変えてもっかいコンパイルする。

.ascii " Data%4x\011\000 どうやら \n が \012 で、\t では \011 っぽい。しかし、なんのことかわからない。

では、PC ではどうなるんだろう。

.string "Hello,World!" おや?

movl \$.LC0, %edi

call puts

おい、そんな最適化いらぬぞ! ということで....

\t にしてみる。

.string "Hello,world!\t "

ほう。 \t とな。ふうむ。 \012\000 にあたりそうな ascii コードも \n になさそうだし、PC じゃふつうに\t って渡すし。

とにかく、これでは 8byte なため、r0 にわたして printf にリンク付き分岐するみたい。

printf(" Data%4x::::\n ",Data);

と言う風に変えてみた。

ところが、

リスト 18 ラベル LC3

```
1 .LC3:  
2 .ascii " Data\%4x::::1200  
3 .space 3  
4 .L71:  
5 .word -536346624  
6 .word .LC3  
7 .word .LC1
```

```
8 .word .LC2
9 .word i2cErr
10 .cfi_endproc
```

命令も

リスト 19 printf()呼び出し部

```
1 ldr r0, .L71+4
2 mov r1, r4
3 bl printf
```

変わらない。

もしや、.L71+4 には.LC3 のアドレスがはいってるのか？

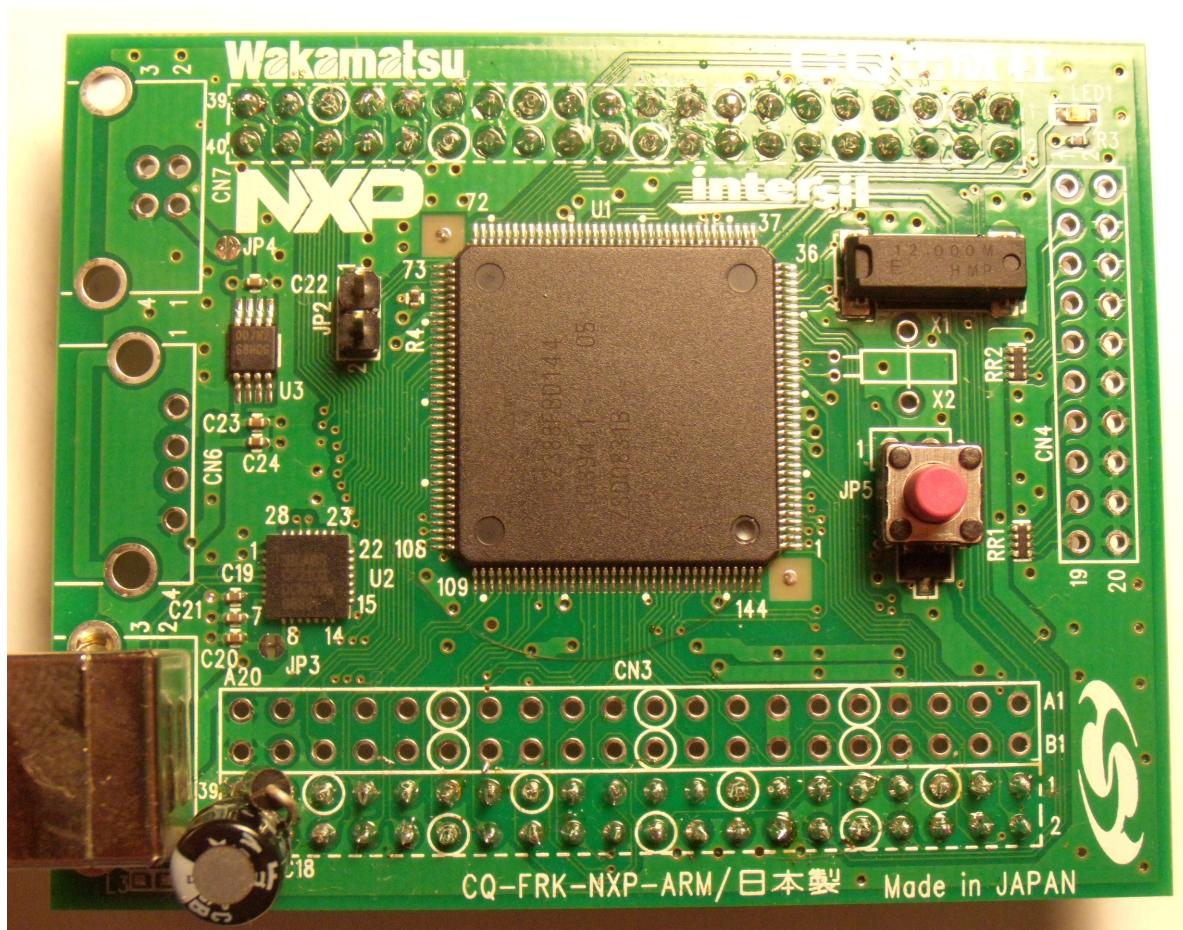
arm-none-eabi-objdump する

リスト 20 objdump で見た printf()呼び出し部

```
1 3c: e59f009c ldr r0, [pc, #156] ; e0 <i2csender+0xe0>
2 40: e1a01004 mov r1, r4
3 44: ebfffffe bl 0 <printf>
4 e0: 0000003c .word 0x0000003c
```

あれ？ 3c ってだれ？ 0x3c を r0 にロードしているのにもかかわらず、0x3c をなにがはいるかは未知であるはず。

図 10 ターゲットのマイコンボード



第 X 部

ノウハウの共有

21 ブログ

22 SNS

23 フォーラム

24 メーリングリスト

第 XI 部

この論文に使われている技術

25 ヴァージョン管理

26 TeX

第 XII 部

社会に於けるオープンソースの立場

27 ビジネスマodel

28 ホスティングサービス

29 近年のクオリティの向上、幅広い層の参画

第 XIII 部

90 億総ソースコード・リテラシィの時代へ

第 XIV 部

文献集

30 データシート

30.1 半導体-英語

49

資料 2 LPC23xx User Manual (UM10211)
http://ics.nxp.com/support/documents/microcontrollers/pdf/_user.manual.lpc23xx.pdf

(メーカーのサイト)

資料 3 CYC1041DV33.pdf
<http://www.cypress.com/?docID=18144> (メーカーのサイト)

31 お世話になった個人のサイト (必ずしもサイトのデータというわけではなく直接お話を伺った場合を含む、敬称略)

FPGA で遊んでみる <http://www.hmwr-lsi.co.jp/>
アルテラマスター P <http://www001.upp.so-net.ne.jp/syrius/>
後閑 哲也 <http://www.picfun.com/>

32 公式サイト

参考 1PROSUME 2010 <http://www.prosume.org/> (PROSUME 公式サイト)
参考 2The GNU General Public License - GNU Project - Free Software Foundation (FSF)
<http://www.gnu.org/licenses/gpl.html> (オープンソース哲学の総本山)
参考 3licenses/GNU_General_Public_License_version_3.0
Open Source Group Japan Wiki - SourceForge.JP
<http://sourceforge.jp/projects/opensource/>
wiki/licenses/FGNU_General_Public_License_version_3.0
ALTERA <http://altera.com/>

33 解説書など

資料 4 改訂・入門 Verilog HDL 記述 — 著者 小林 優 — 発行所 CQ 出版株式会社
資料 5 Interface 2009 年 5 月号 — 編集・出版 CQ 出版 ISSN:0387-9569
資料 6 Interface 2009 年 6 月号 — 編集・出版 CQ 出版 ISSN:0387-9569
ものづくり革命-パーソナル・ファブリケーションの夜明け- — 著 ニール・ガーシエンフェルド — 訳 糸川 洋 発行社 ソフトバンククリエイティブ ISBN-13: 978-4797333145

34 よむものリスト

需要と供給の関係の本具体的に例をあげる
第三の波さがす

34.1 センサ

34.2 規格

図 11 飲み干された RedBull は RedNull である

