

- (1) 無料のSlackアプリをインストールを済ませておく
 (2) その後、通知したURLにアクセスし、下にスライド、



- (3) 下にある、「アカウントを作成する」をクリック、



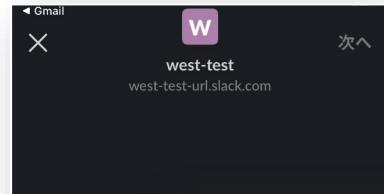
- (4) keio.jpのメールアドレスを入力後、「アカウントを作成」をクリック、



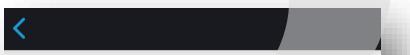
- (5) keio.jpのgmailに届くメールにある「ここをクリックして続行」を押し、



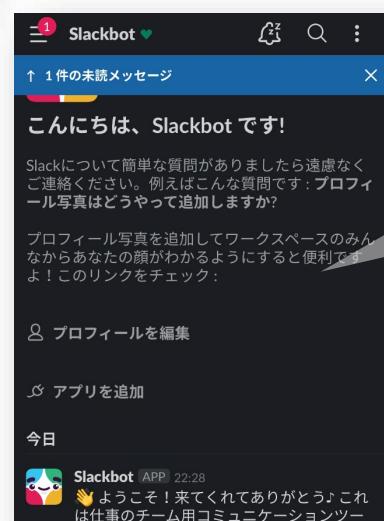
- (6) 名前に続き、パスワードを入力、



- (7) 「同意する」を押すと、



- (8) ワークスペースが利用できるようになります



keio.jpアカウントを用いたSlackによる授業ワークスペースへの参加の仕方

理工学部システムデザイン工学科 西

ユーザー向けサービス利

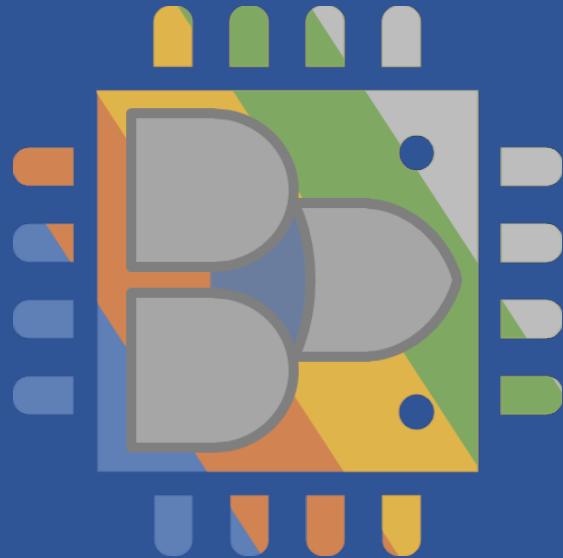
発効日：2018年4月20日

本ユーザー向けサービス利用規約(以下「ユーザ

主たる最初に

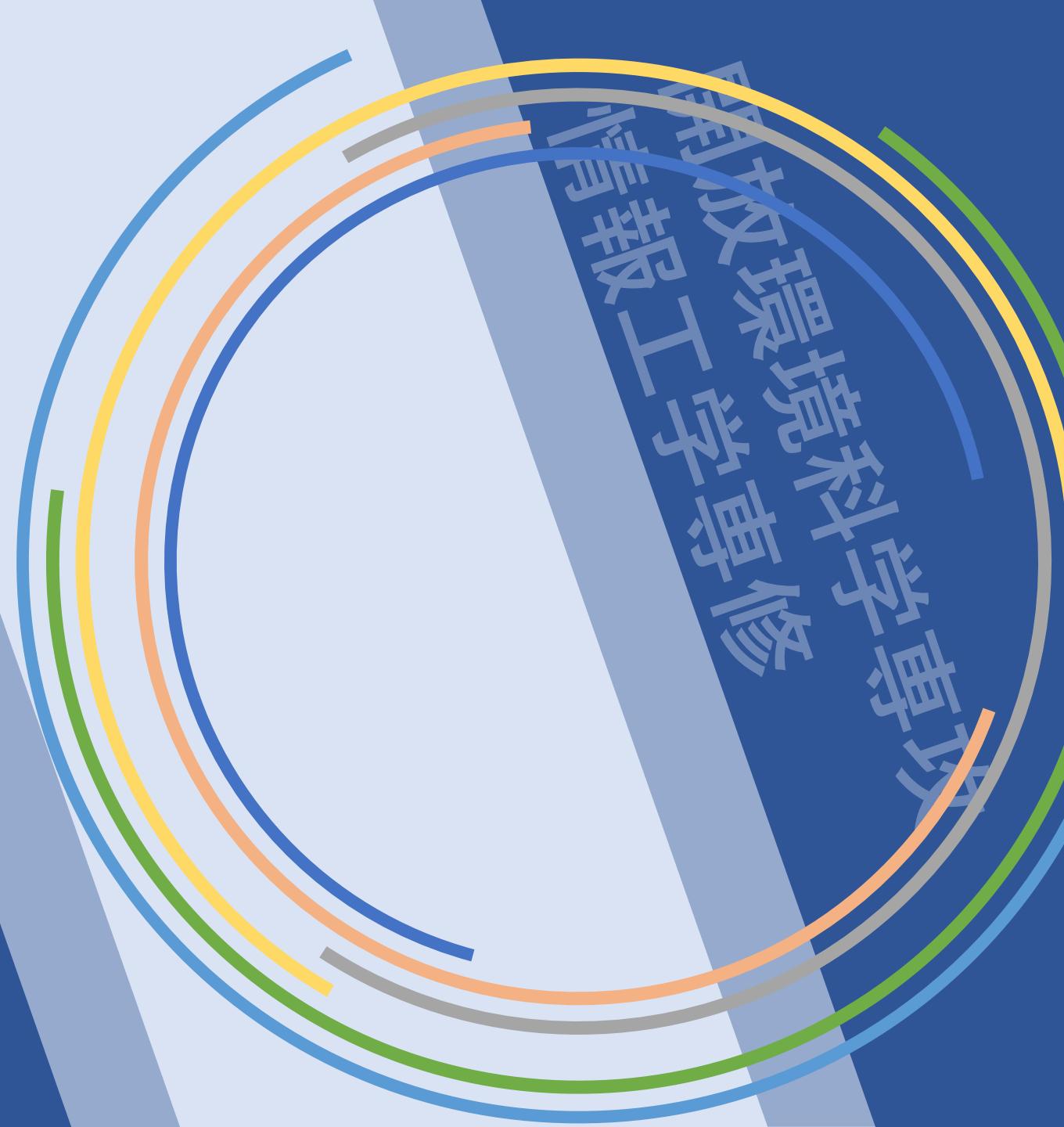
「同意する」を選択することで、ユーザー向けサービス利用規約、プライバシーポリシー、およびCookieポリシーを理解し同意るものとします。

同意する



計算機システム 設計論(1) ガイダンス

担当： 西 宏章





この授業の目指すところ

- ハードウェア記述言語を用いた論理回路設計の基本から応用を学ぶ
- 本来はFPGA実習基板を用いたハードウェア実習を含む、計算・通信機器設計について学ぶが、オンラインでも取得可能となるように再編集している
- 簡単なCPUおよび通信システム（ただし通信システムは相互連携が必要なため授業時間の関係から論理回路設計までは行わない）のハードウェア記述について学ぶ
 - 大学院の授業で基礎から学ぶことについては意見もあると思うが、当該授業の履修者が様々なバックグラウンドを持つ学生で構成されていることからも、「これまで行ってきた研究を具現化するためハードウェア設計について学びたい」「改めてプロセッサを含む論理回路設計を学びたい」というニーズに答えることは重要と考える
 - 実際にハードウェアを動作させるところまで進めることを目指すが、昨今の状況から論理回路シミュレータを用いた動作確認を最終目標とする





前提とする知識や能力

- プログラミング言語
 - 一般的なソフトウェアプログラミング言語の知識であればなんでもよい
 - ただし、それに固執すると学習上の妨げにもなる
 - オブジェクト指向言語の概念はある程度役に立つ
- 計算機やデジタル回路実験等の内容
 - こちらはかなり重要な基礎知識となる
 - これらに習熟していない学生の受け皿となっていることも理解している
- エラーや理不尽さに負けない努力と根性
 - ソフトウェア工学の発展からみれば、明らかに王道を外れた言語仕様
 - ハードウェア技術者は「結局0/1だし、合成できればよい」という楽観主義
- 妥協をする心
 - ソフトウェアと違い、「リソース」に響く
 - 動けばよいではない。クロックサイクル数やタイミング・利用記憶素子・回路規模・消費電力などを検討、設計に盛り込まなければならぬ
 - 全てにおいてよい結果を得ることはあきらめなければならない





授業の実施について

- 実施形態
 - 講義と演習課題がある
 - 全ての講義と演習はITCの計算機室で行う
- 評価
 - 演習課題の総合得点 + 最終課題で評価
 - 演習課題の総合得点30%（あれば） + 最終課題70%のウェイト分配
- この授業を履修することにより習得できる具体的な内容
 - Verilogハードウェア記述言語を用いた論理回路設計
 - 基礎的なプロセッサや通信機器、周辺回路設計に関する知識
 - ハードウェア構成の概念
 - これらを総じた計算機システムの設計に関する知識およびその理論体系
- 質問
 - west@sd.keio.ac.jp まで





Verilogってなに？

- こういう時代もあった
- 2018年からランキングに入っていない
 - ソフトウェア設計言語に絞られた様子
 - 実際、世界的に見れば「特殊能力」
 - 必要悪として知っているべき
- FPGA設計などで利用機会は増大
 - FPGAの利用メリットが明確であれば
 - SW/HWどちらが適するかの見極め能力は重要
- 食いっぱぐれない
 - 転職nendo転職エージェント
「おすすめの食いっぱぐれない仕事」
 - 1. エンジニア
 - 2. コンサルタント
 - 3. 設備・電力系技師

The screenshot shows a news article from Myナビニュース titled "2017年に学ぶべき年収の高いプログラミング言語トップ15" (Top 15 programming languages to learn in 2017 for high salary). The article is dated November 30, 2016, and is written by 后藤大地. It includes a logo for fossBytes and a list of 15 languages.

2017年に学ぶべき年収の高いプログラミング言語トップ15

後藤大地 [2016/11/30]

fossBytesに11月28日(米国時間)に掲載された記事「29 Highest Paying Programming Languages You Need To Learn In 2017 | fossBytes」が、年収の観点から、2017年に学習をするべきプログラミング言語を紹介した。同記事は、Paysa.comによる「Silicon Valley's Most Valuable Skills」の内容に基づいている。

ランキングトップ15の言語は以下のとおり。

1. Verilog
2. Scala
3. Scheme
4. Objective-C
5. R
6. Perl
7. Go
8. Python
9. C++
10. C
11. Ruby
12. LaTeX
13. Java
14. MATLAB
15. Flex

Hello, we are fossBytes.





そうじゃなくってVerilogってなに？

7

- ・ハードウェア設計はソフトウェア設計と何が違うのか？
 - ・ハードウェア設計は**並列度の向上**が容易
 - ・資源(面積・電力・発熱量・コスト)には限りがある
 - ・ある与えられた処理に特化した物理的実態を実装すること
 - ・ソフトウェア設計はソフトウェア設計を許すハードウェアにより可能
 - ・ハードウェアがなければ、ソフトウェアはただの紙
- ・自由度は高い・高すぎる
 - ・CPUが持つボトルネックは破壊できる
 - ・ノイマンアーキテクチャ・ハーバードアーキテクチャに従う必要はない
- ・最先端の真似事は基本意味がない
 - ・ソフトウェア設計のエキスパートでもWindowsを一人で設計するのは無謀
 - ・芸術品ともいえるインテルのプロセッサに勝ってやるぜ！は同様に無謀
 - ・それでも理解し、ある用途に「特化した」オリジナルを設計できることは重要





ハードウェア設計の実際

- 時代はスピードを追い求め、面倒なハードウェア設計をなんとか楽にしようと様々な技術や方法論が古くから提案され続けてきた
 - スケマティック記述CAD, 基板設計CAD(この講義ではおそらく扱わない)
 - ハードウェア記述言語
 - シミュレーションCAD
 - コンプライアンスCAD
- これらを身に着けなければ、商用ハードウェア実装は夢となるほど大変
- CADの手助けなしでも努力をすればできなくも無いが、その規模であれば実用上無意味
 - 1970年代までは、ANDなどのゲートを紙に書いて設計し、学校の体育館を貸しきって紙を並べてデバッグしたという話を聞いたことがある
 - 生産性向上のニーズとプロセッサ性能の向上により、ソフトウェアが持つ高い記述性・生産性を取り入れたハードウェア記述言語が提案、利用されるようになった
 - ハードウェア記述言語はいくつか提案されている





ハードウェア記述言語の種類

- Verilog

- この講義で習得する対象言語
- 利点：日本でメジャー、習得しやすく、タイプ量が少ないためお勧め
- 欠点：あいまいな点があり、言語としての完成度は他に比べると劣る
 - とはいっても、SystemVerilogの登場で無敵になった

- VHDL

- verilogと肩を並べて著名なハードウェア記述言語
- 利点：アメリカ国防省の標準仕様記述言語でありメジャー
- 欠点：タイプ量が多く、型に厳しい。

- System C

- 流行った時期もあった。C言語とほぼ同じ言語仕様をもつ
- 利点：Cさえ取得していれば誰でもハードウェア設計を始めることができる
- 欠点：タイミングの概念が曖昧でハードウェアを知る人は不満が残る

- その他

- 様々なハードウェア記述言語が提案されている。和製ではParthenonが著名





ハードウェア論理設計の手順

10

1. 何らかの方法で設計を行う
 2. 論理シミュレーションを行い動作を確認
 3. 論理合成を行い回路情報に変換
 - ANDやOR, FFといった情報になる
 4. 合成レポートを見て制約(スピード, 規模)を満足するか確認
 5. 仮負荷シミュレーションを行い動作やスペックを確認
 6. 配置配線(STA: Static Timing Analysis もここで行う)
 7. 実負荷シミュレーションを行い動作やスペックを確認
 8. テストベクトルの作成
 9. メーカに製造を依頼する(要するに製造開始でテープアウトと呼ばれる)
 10. 完成
- 赤：シミュレータで行う範囲
- 青：FPGAのCADで行う範囲



製造・実装対象

11

- 汎用LSI (Micro-controller, CPU, DSP)
- 機能LSI (System LSI, 専用LSI, System on Chip)
- Custom LSI (Gate Array : SOGとも言った, Standard Cell)
- Programmable Logic Deice (PLD/CPLD)
- Field Programmable Gate Array (FPGA)
- CPLDとFPGA

	CPLD	FPGA	
プログラム素子	EEPROMセル	SRAMセル	アンチヒューズ
プロセス(μm)	0.18～0.25	0.09～0.18	0.18～0.22
ゲート規模	小規模	大規模	中規模
再書き込み	可能	可能	不可
基本構造			



Verilogのすすめ

- ・この講義では習得のハードルの低さからVerilogを習得する
- ・本格的なハードウェア記述言語で業界標準の一つ
 - ・国内ではVerilogの方がより人気がある
- ・Verilogの言語仕様すべてを理解する必要はなく、厳選しつつ、授業で習得するVerilog文法項目でなんでも設計できるようにする
- ・最近の流れはSystemVerilog
 - ・とはいえ、Verilogの取得が前提
 - ・Verilog-HDLと同じ情報量で記述量を削減するための工夫が施されている
 - ・もうVHDLの出る幕はない
 - ・テストベンチは数分の1
 - ・今年からSystemVerilogも一緒に扱います！





デジタル回路の基礎

13

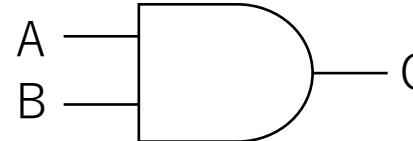
- 論理
 - すべて2進数で表現する
 - 0 = LOW = 0V(電圧の低い方という意味)
 - 1 = HIGH = 5V, 3.3Vなど
- MIL記号法
 - 米国軍の仕様記述ルール、MILITRAYが由来





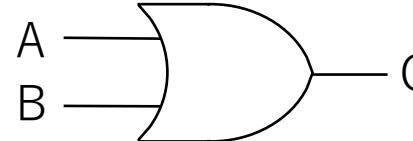
MIL記号法と真理値表(真偽値表)

AND(AND2)



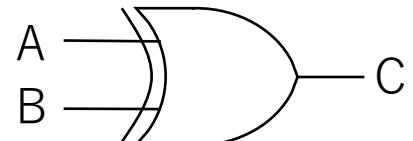
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

OR(OR2)



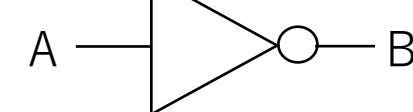
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

XOR(XOR2)



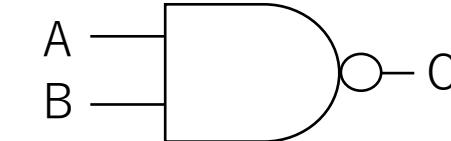
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

INV(NOT)



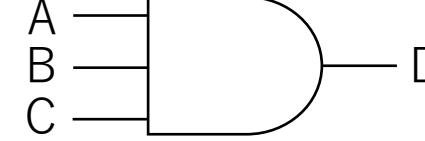
A	B
0	1
1	0

NAND(NAND2)



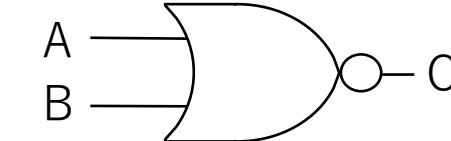
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

AND(AND3)



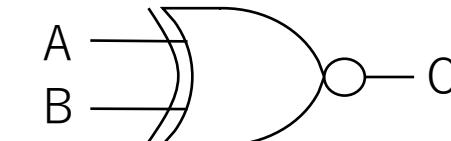
A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

NOR(NOR2)



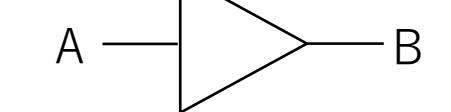
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

XNOR(XNOR2)



A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Buffer



A	B
0	0
1	1

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

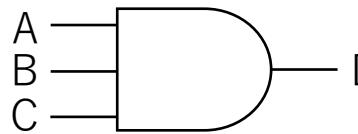




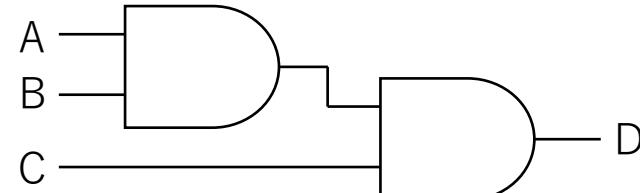
回路の基礎（問題1）

- 多入力回路は2入力回路の組み合わせで必ず変換できる

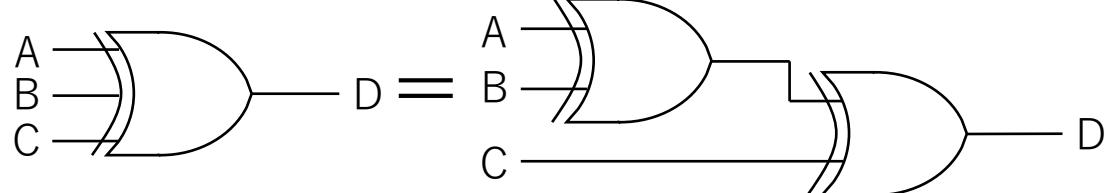
AND(AND3)



AND(AND2)

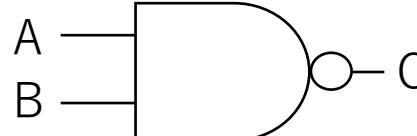


XOR(XOR3)



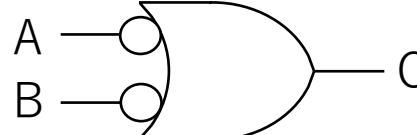
- 2通りの記述ができる（○は取る、なければ付ける、ANDとORは交換する）

NAND(NAND2)

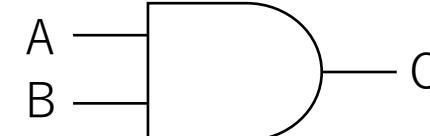


||

NAND(OR2b2)

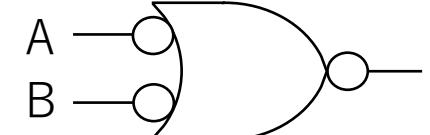


AND(AND2)

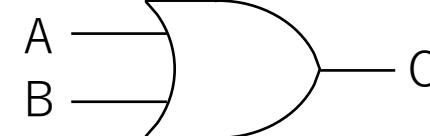


||

AND(NOR2b2)

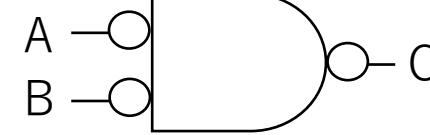


OR(OR2)

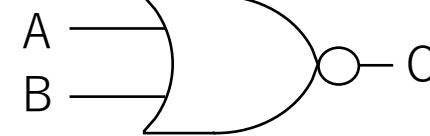


||

OR(NAND2b2)

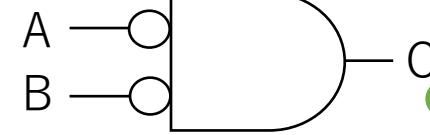


NOR(NOR2)



||

NOR(AND2b2)

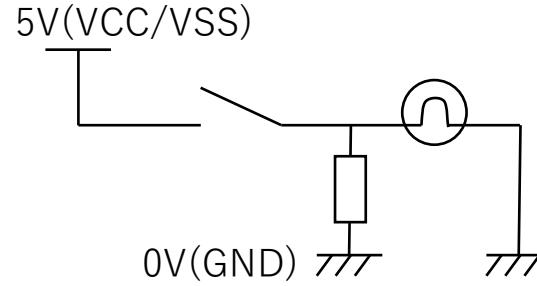
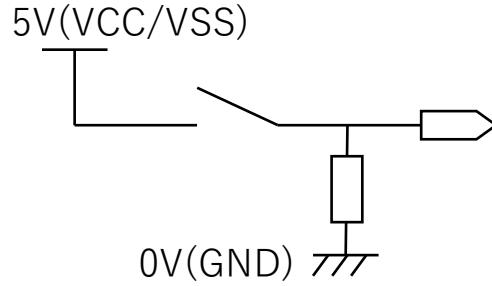




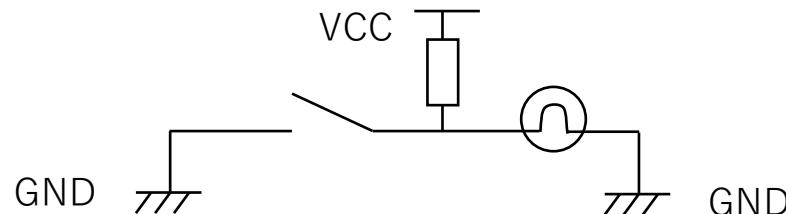
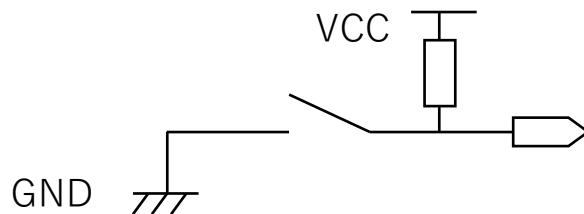
Active High(正論理)とActive Low(負論理)

16

- ボタンを押すと5Vになる、5Vにするとランプがつくなどは正論理

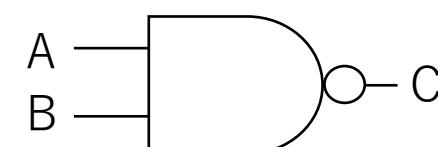


- 逆は負論理で、押すと0Vになり、0Vにするとランプがつく

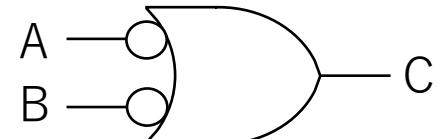


- は負論理を意味するために利用する（同じNANDでも使い分ける）

- 2つの正論理入力のAND条件で負論理の制御を行う



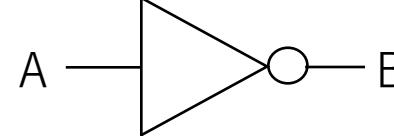
- 2つの負論理入力のOR条件で正論理の制御を行う





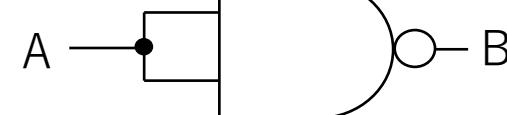
NANDによる置き換え

INV(NOT)



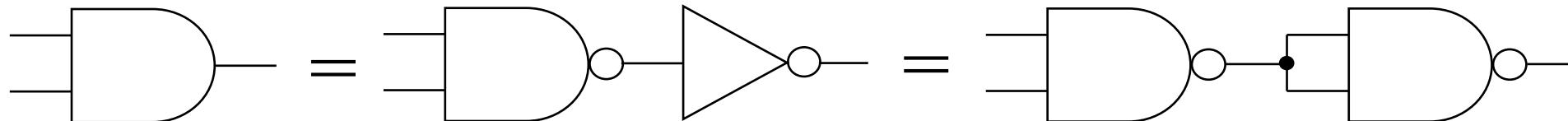
A	B
0	1
1	0

NAND(NAND2)

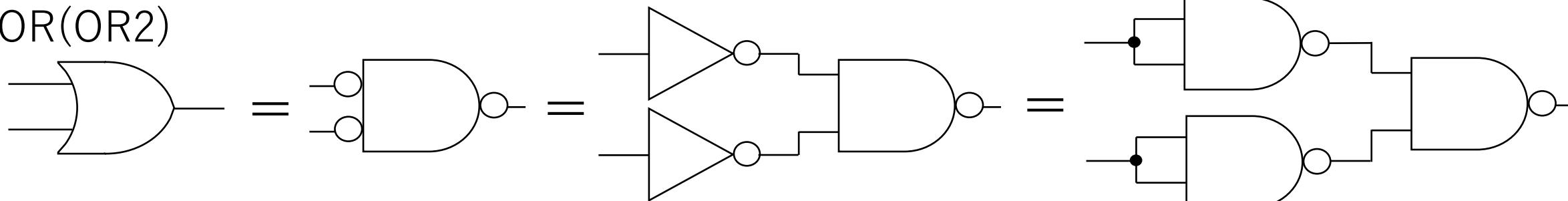


A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

AND(AND2)



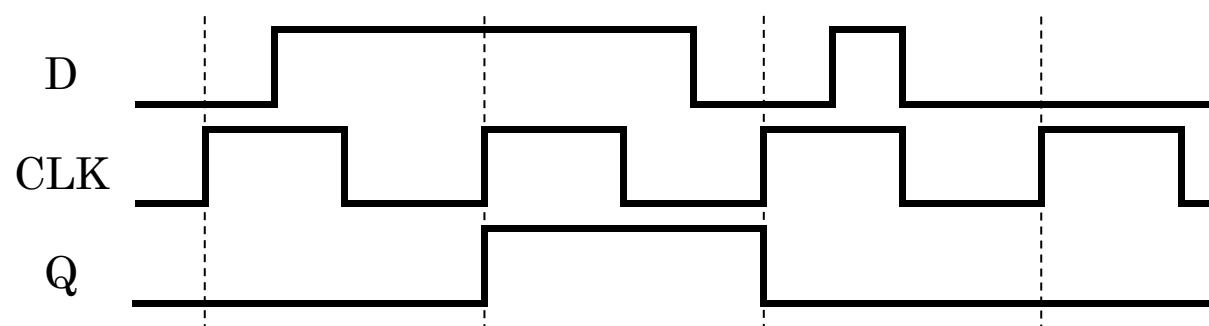
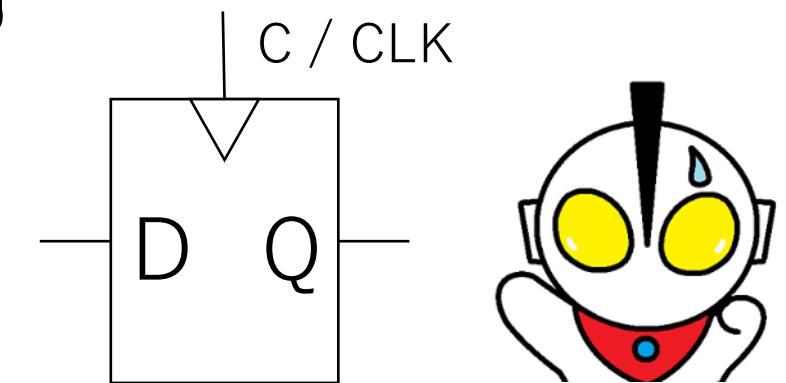
OR(OR2)





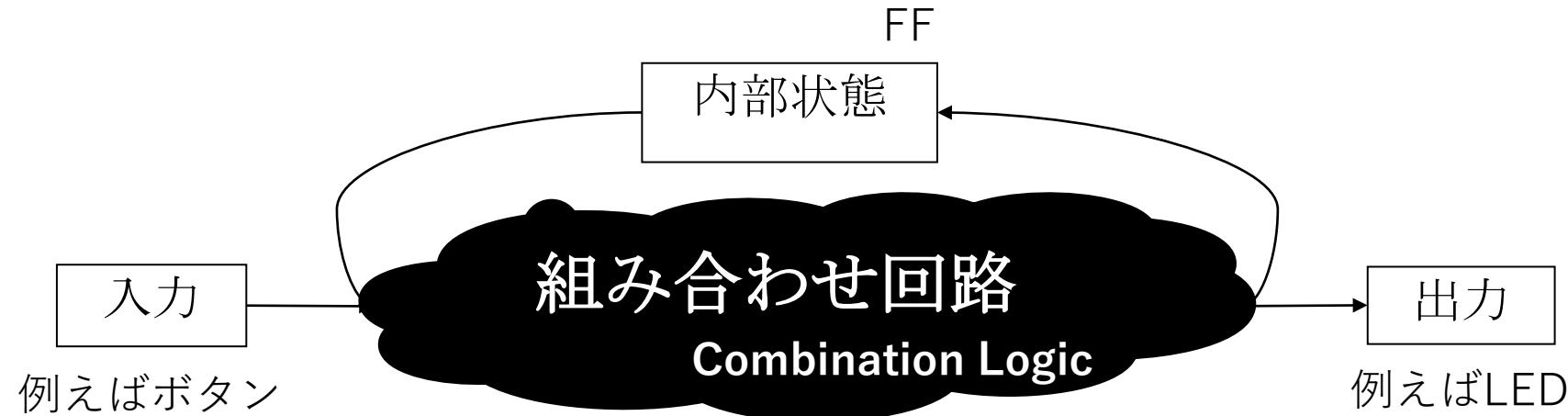
記憶素子

- ANDなどで構成される組み合わせ回路では「記憶」動作が実現できない
- 「記憶」する動作を行う回路を記憶素子と呼ぶ
- 様々な記憶素子が存在するが、この実験ではD-FFのみ用いる
 - したがって、FFといえば暗黙の了解でD-FFを指す
- FFのMIL記号法
 - ウルトラマンっぽい形で、DとCが入力、Qは出力
 - Final FantasyとDragon Quest
 - Cの立ち上がり時のDの入力をラッチしてQから出力する（0次ホールド）





順序回路



- 例えば、自動販売機の制御回路は、
 - 入力はお金
 - 出力はジュースを出すモーター
 - 内部状態は「いくらお金が投入されたか」
- すなわち、内部状態（過去の投入金額はいくらか）と入力（今いくら投入されたか）を勘案して、次の内部状態と出力（ジュースを出すかどうか）を判断する回路を設計すればよい

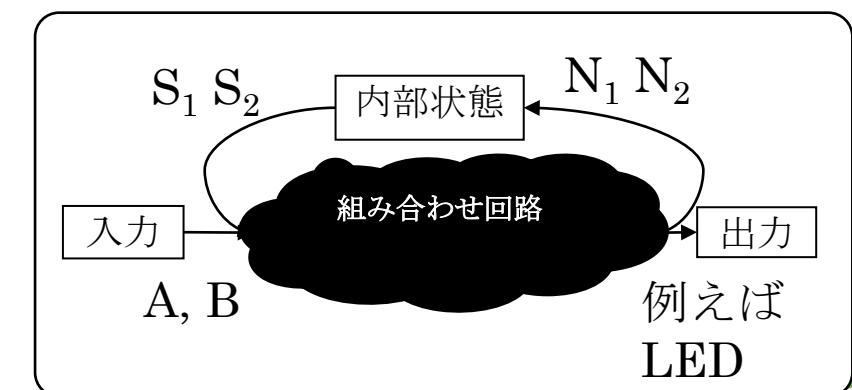
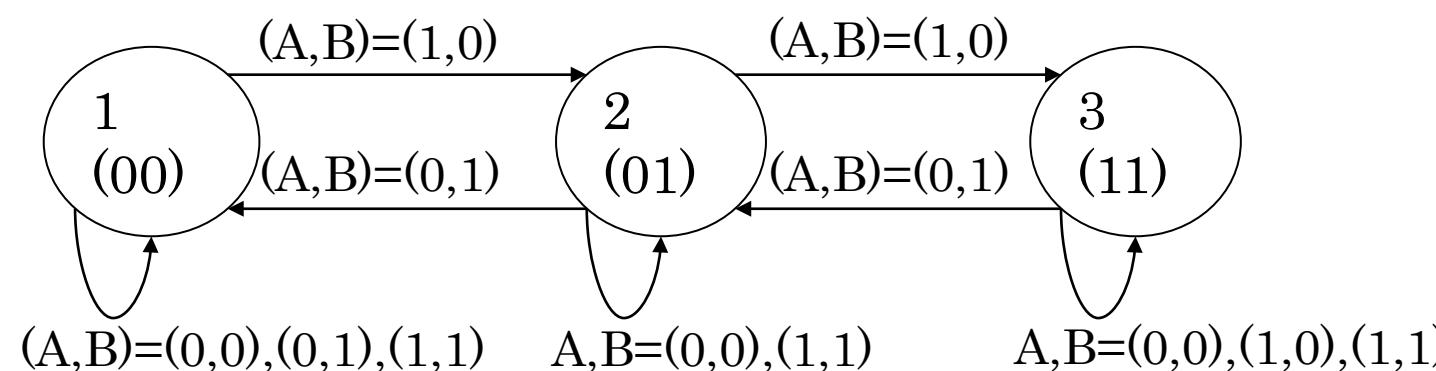
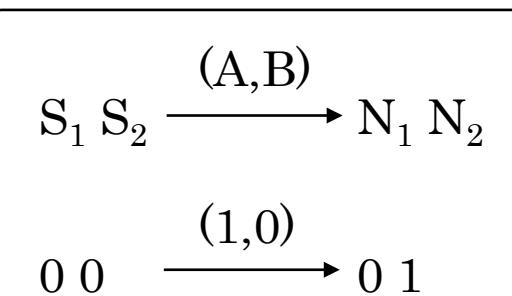
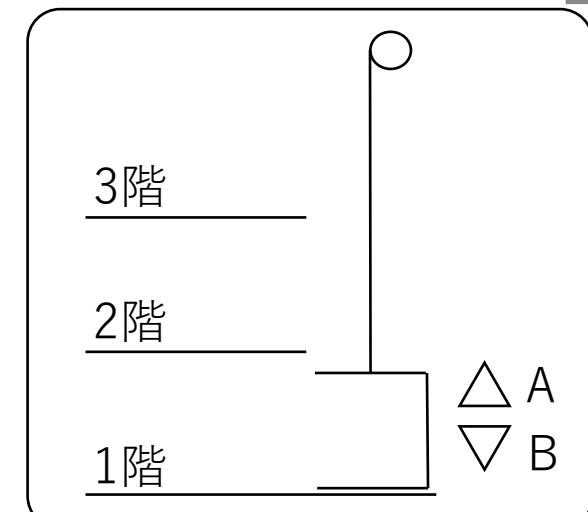




順序回路の例題

20

- エレベーター（もどき）の制御回路を作る
 - テキストの仕様を確認する
 - 実際にはエレベータは瞬間移動しない
 - 移動中の状態を表現すれば本当のエレベータも設計できる
- 状態遷移図(State Transition Graph)を作る
 - 状態数はいくつか？を考え、すべての状態遷移を記述する
 - 各状態に2進数で値を割り振る
 - 仕様にあるすべての状態遷移を記述する
 - 全ての遷移が含まれているか確認する（不足はつぎ足す）

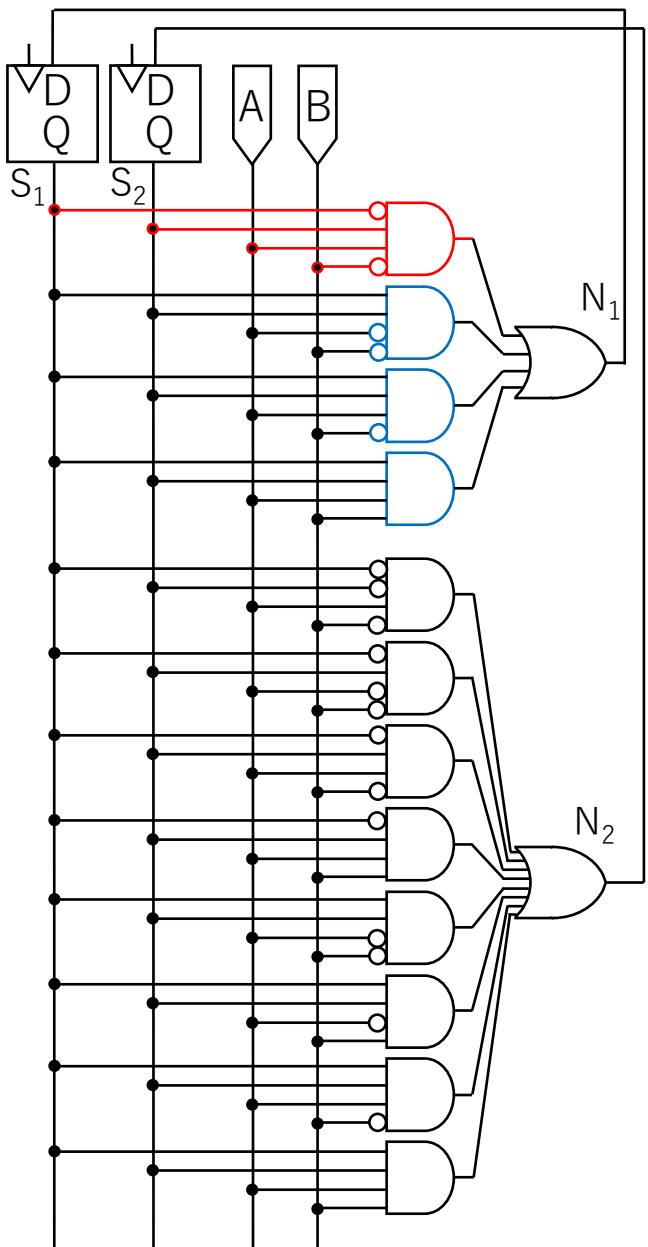




状態遷移図から真理値表を書く

21

S1	S2	A	B	N1	N2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	0	1
1	0	0	0	-	-
1	0	0	1	-	-
1	0	1	0	-	-
1	0	1	1	-	-
1	1	0	0	1	1
1	1	0	1	0	1
1	1	1	0	1	1
1	1	1	1	1	1



- 真理値表をそのまま回路にする
 - 1つの回路の出力は1つなので、2つの回路を個別に設計する
 - 真理値表を横に見たらAND
 - 真理値表を縦に見たらOR
- 一応回路は設計できるが・・・
 - こんな回路は設計してはいけない
 - 回路点数が多い!
 - 重量増（エレベータなのに）
 - 消費電力大（エコの時代なのに）
 - 基板大（ダウンサイジングなのに）
 - 入力数が多い
 - 回路が高価、特に8入力ORは大変
 - 配線も多い
 - 60本！
 - 実装ミスや故障がしやすい





回路を圧縮してシンプルにする

22

- もう、カルノー図は忘れよう。
 - CADが圧縮してくれる。だから、CADが圧縮しやすいようにわかるように記述
 - アーキテクチャ最適化は人間の仕事
 - 記述合成上の最適化はCADの仕事
- 74デバイスも忘れよう
 - もう使わない
 - FPGAによる設計
 - PCに回路を入力し、USBでFPGA基板とPCを接続、回路を焼き込む
 - FPGAの基本動作や仕組みはテキストを参照





FPGAとは

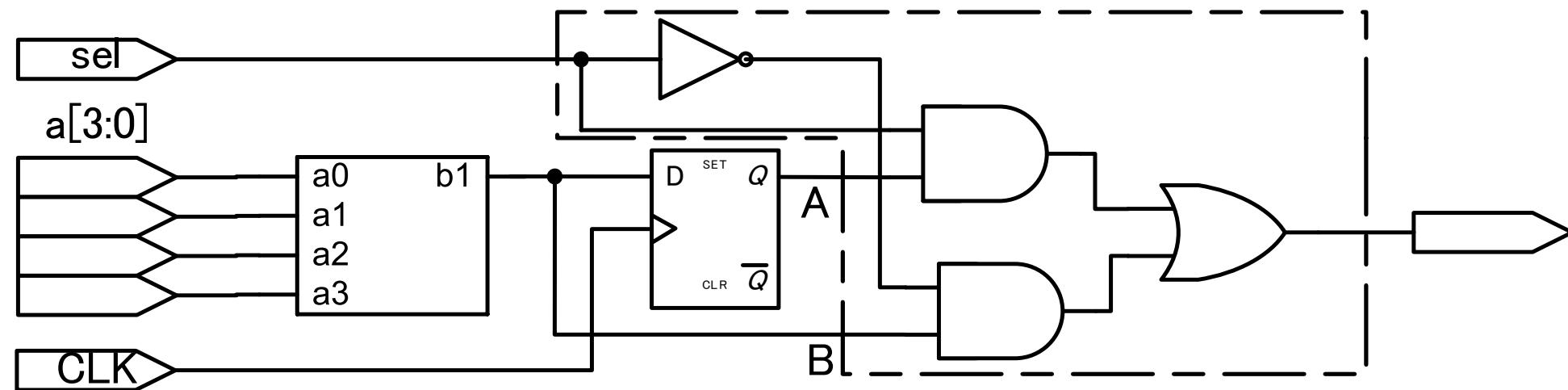
- FPGA(Field Programmable Gate Array)は内部の論理構成を自由に変えることができる(reconfigurable)デバイスの一種
 - FPGAは内部にAND, OR等、自由に論理ブロックを構成できる
 - さらにその論理ブロック間を自由に配線できる
 - FPGAはこの自由な論理ブロックおよび配線を実現するためにメモリ(SRAM: Static Random Access Memory)および半導体スイッチを利用する
- 真理値表はメモリである
 - 先に設計し圧縮した回路は、圧縮しなければN1およびN2について4入力1出力の真理値表が示す内容に従う回路である。 真理値表をアドレスが4bit、出力が1bitのメモリとそのデータの中身と考えれば、このメモリは設計した回路と等価である
 - 特にFPGAではこのメモリ要素をLUT (Look Up Table)と呼ぶ
 - これを自由に配線で接続できれば、どのような組み合わせ回路も設計可能であることを意味する





FPGAの動作原理

- FFも含めて自由に設計できるようにするため、FFをメモリの出力に配置し、その利用を選択できるようとする。次の構造に着目しよう
 - a0からa3はアドレス、その結果としてのb1を得ることで真理値表と等価となる
 - selが1の時、 b1がFFを介したAを通って出力される
 - selが0の時、 b1は直接Bを通って出力される
 - つまり、 FFの利用がselで選択できる。これを基本構造とする





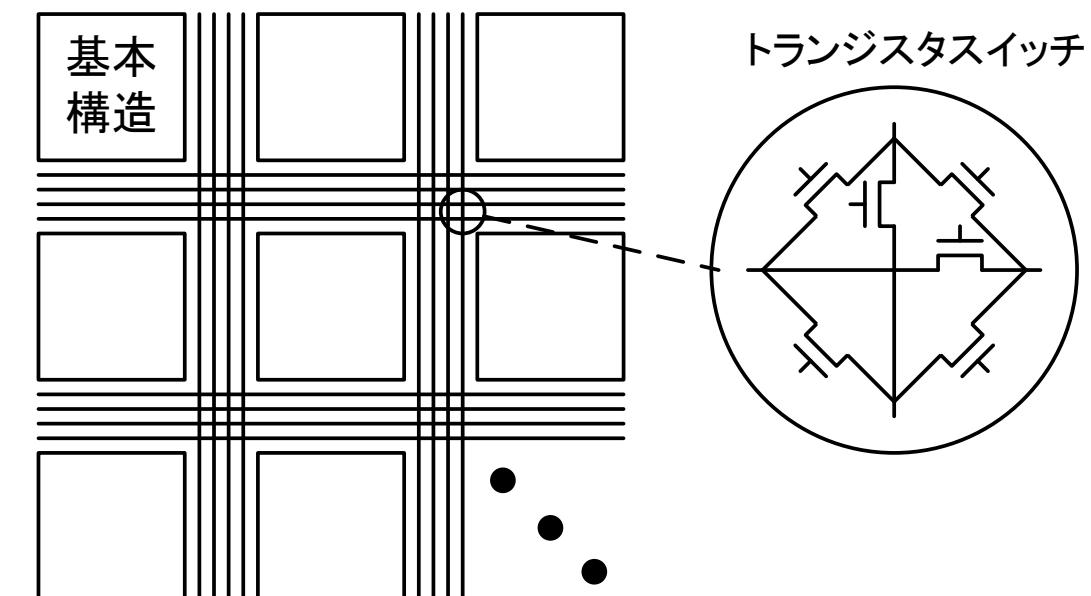
FPGAの動作原理

- FPGAは基本構造の海である

- 基本構造を大量に敷き詰め、かつそれらの間をすべて配線しておき、半導体スイッチでON/OFFできれば、基本的にはどのようなデジタル回路でも設計可能となる
- 配線は、各基本構造の周りに縦横に張り巡らされた配線の集合(バス)で実現
- バスの各交点に半導体スイッチ素子を配置、このON/OFF情報もSRAMで与える

- FPGAはSRAMで動作が決定する

- 回路もSRAM、配線もSRAM
- CADを用いてSRAMの内容を決定
- これをFPGAにアップロードすればよい





FPGAの可能性

- 制御応用
 - 電源ONと同時に動作
 - パソコンのようにOSのロードやデバイス認識などで待つことがない
 - 必ず遅延を守る
 - 決められた作業を決められたタイミングで行う
 - パソコンのようにどのような作業が割り込んでも、設計された以上に遅くならない
 - 制御では低遅延・低ジッターが必要でアルゴリズムがシンプルであることから、FPGA応用が進んでいる
- ホビー
 - FPGAは価格も落ち着き、既にホビーでも広く用いられている
 - ファミコンをFPGA上に実装する
 - https://pgate1.at-ninja.jp/NES_on_FPGA/
 - スーパーファミコンをFPGA上に実装する
 - https://pgate1.at-ninja.jp/SNES_on_FPGA/
 - シンセサイザーの実装例も多数あり、ニコニコ技術部、作ってみたでも定番

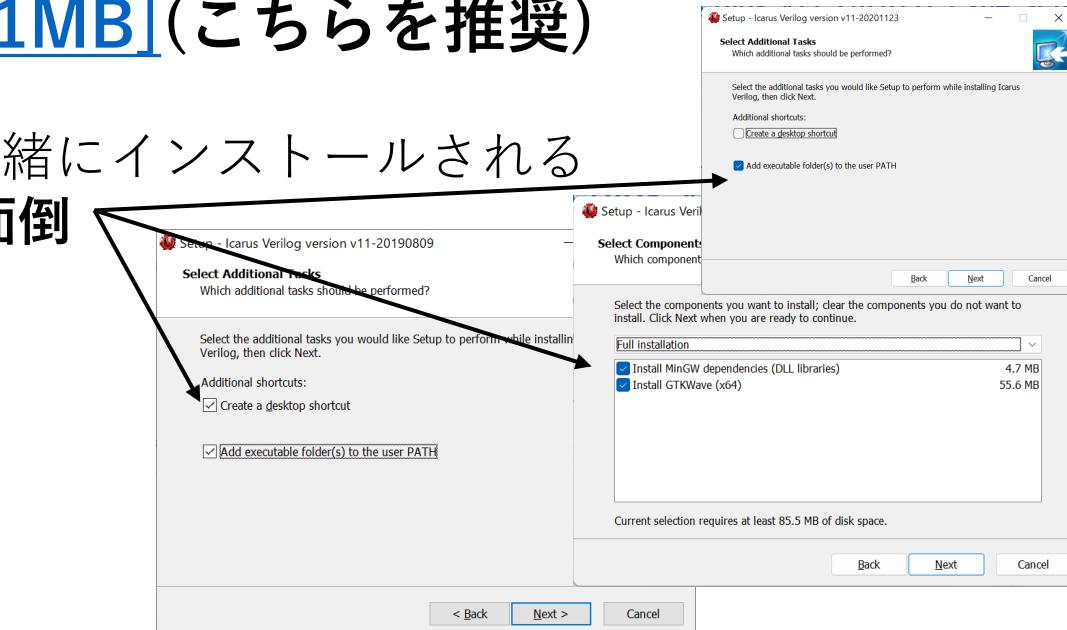




Icarus Verilog

27

- フリーのVerilogシミュレータ
 - 最新版は[iverilog-v11-20210204-x64_setup.exe \[44.1MB\]](#)(ただし問題あり)
 - [**iverilog-v11-20201123-x64_setup.exe \[18.1MB\]**](#)(こちらを推奨)
Windowsでは(Google先生に聞くとよい)
 - <http://bleyer.org/icarus/> 最新版ではgtkwaveも一緒にインストールされる
 - インストール時にPATH設定をチェックしないと面倒
 - オプション -g2012でSystemVerilogに対応
 - 各自自分のPCにインストールすること
 - MacOSでは(Google大先生に聞くとよい)
 - まずは、homebrewを入れる
 - \$xcode-select –install
 - /usr/bin/ruby -e "\$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/master/install>)"
 - 次に、Xquartzを入れる
 - <https://www.xquartz.org/>
 - \$brew cask install gtkwave
 - \$brew install icarus-verilog



- Macでgtkwaveをコマンドラインから起動するとエラーになるが次の方法で起動できる
 - Launchpadのgtkwave.appを実行する
 - vcdファイルを作成したら、open ファイル名.vcd
もしくは、Finderでvcdファイルをクリックする

ただし、この方法だと古いバージョンが入るのでお勧めしない。次のページを参照





- LinuxやMacOS環境では(Windows WSLなども含む)
 - sudo apt upgrade もしくは sudo yum update # Ubuntu系とCentOS系で違います
 - 次の方法はお手軽ですがバージョンが古くalways_combが使えません
 - sudo apt install iverilog # Centosでは、aptはyumに置き換えてください
 - 頑張るならこっち(環境依存・インストールパッケージ依存)
上記aptで入れてしまっていたら、 sudo apt remove iverilog
 - git clone <https://github.com/steveicarus/iverilog>
 - cd iverilog
 - sudo apt install build-essential
 - sudo apt install autoconf gperf flex bison
 - autoconf; ./configure; make
 - sudo make install
 - これでインストール完了です。iverilog -vでバージョンが11.0ならOK
 - 残りを入れましょう
 - sudo apt install gtkwave
 - sudo apt install make
 - Windows WSLでは、さらにXサーバのインストールが必要です
 - Windows上にXming([Xming-6-9-0-31-setup.exe](#))を導入





演習問題（提出自由）

- 演習

- Icrus Verilogの環境を構築しなさい
- iverilogとコマンドラインに入力すると次のメッセージが出ることを確認しなさい

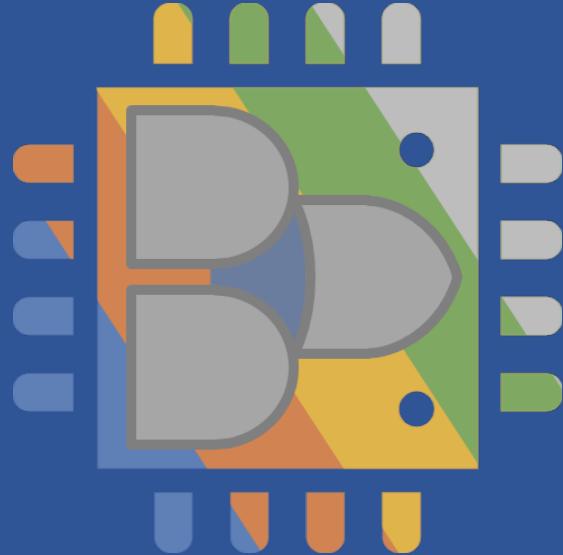
```
>iverilog (入力 以降 > をプロンプトとする)
```

```
iverilog: no source files.
```

```
Usage: iverilog [-EiSuvV] [-B base] [-c cmdfile|-f cmdfile]
                 [-g1995|-g2001|-g2005|-g2005-sv|-g2009|-g2012] [-g<feature>]
                 [-D macro[=defn]] [-I includedir]
                 [-M [mode=]depfile] [-m module]
                 [-N file] [-o filename] [-p flag=value]
                 [-s topmodule] [-t target] [-T min|typ|max]
                 [-W class] [-y dir] [-Y suf] [-I file] source_file(s)
```

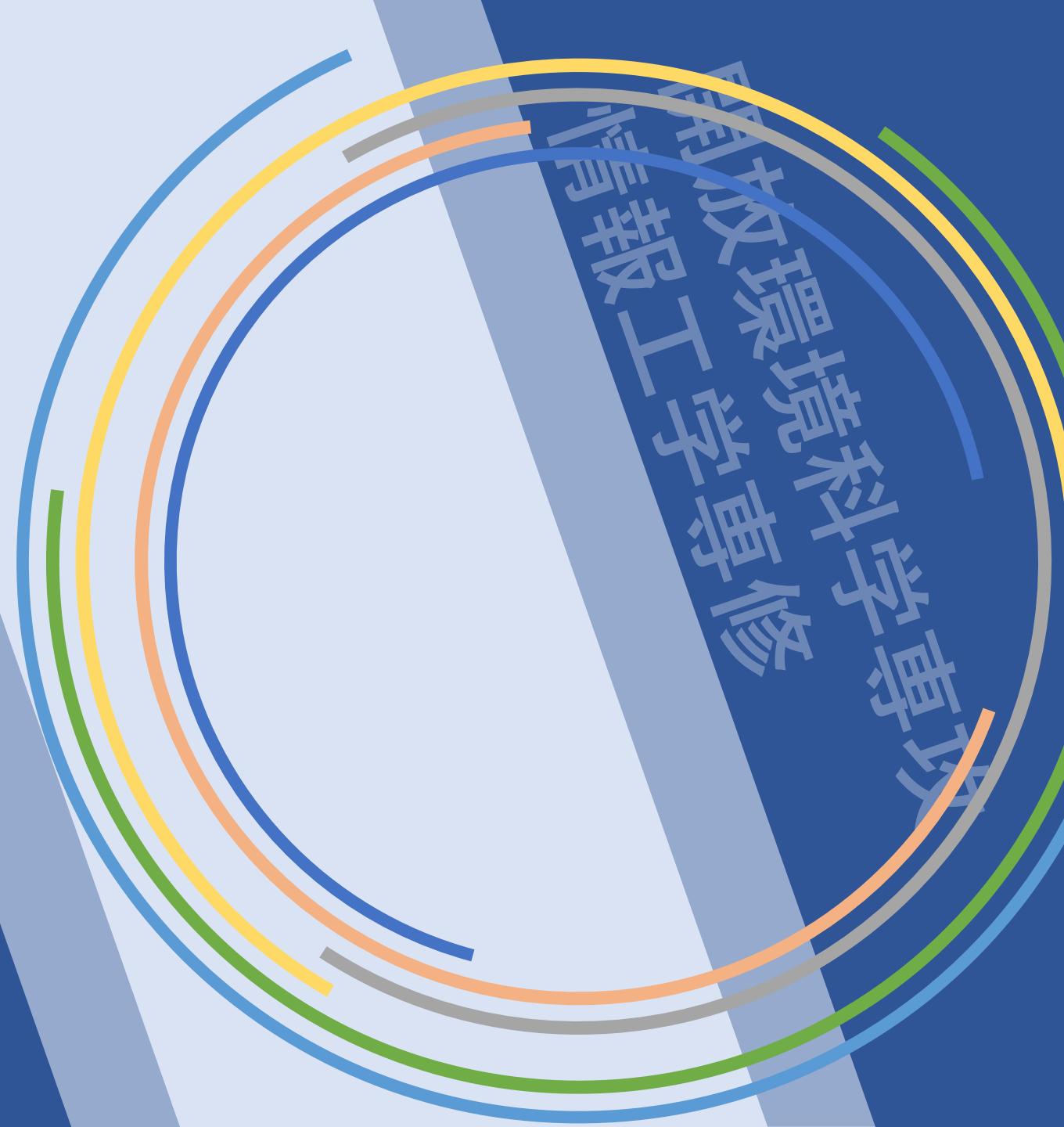
```
See the man page for details.
```





計算機システム 設計論(2) Verilogの基礎

担当： 西 宏章



```
counter(CLK, RES);
input CLK, RES;
output [3:0] Q;

wire CLK, RES;
reg [3:0] Q;

always @ (posedge CLK)
  if (RES == 1'b0)
    Q = 4'd0;
  else
    Q = Q + 4'd1;
end
```

Verilog-HDL記述の基本

- Verilogの文法を回路の観点で学ぼう



今後の予定

1. イントロダクション
2. Verilog-HDLの基本
3. 回路の決まりパターン
4. ステートマシーン
5. プロセッサの構成
6. 通信システムの構成
7. その他の話題
8. コンテスト





ハードウェア記述言語の記述スタイル

33

- behavior 記述
 - 回路の振る舞いを記述し合成を意図しない記述で、やりたい放題
 - プログラム言語として使うイメージで主にシミュレーション記述に用いる
- RTL 記述:
 - Register Transfer Level 記述し合成を意図する
 - 制約があるが、設計はこの記述スタイルになる
- Gate回路記述
 - primitive記述とも呼ぶ。Verilogでは、Verilogネットリストとも呼ぶ
 - 論理の結合をverilogで書いたイメージ

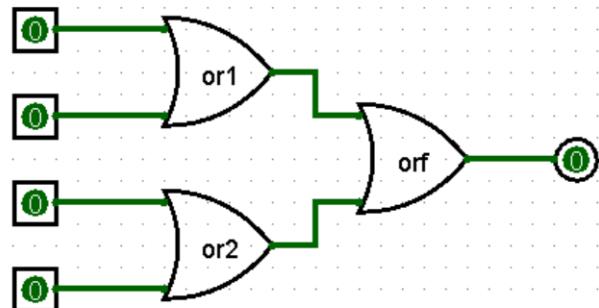


各記述スタイルの違い

Gate記述

```
module or(a, b, o);
  input a, b;
  output o;
  assign o = a | b;
endmodule
```

```
module org(o, a, b, c, d);
  input a, b, c, d;
  output o;
  or or1(a, b, x);
  or or2(c, d, y);
  or orf(x, y, o);
endmodule
```



RTL記述

```
module dcnt(r, l, en, clk, rst);
  input en, clk, rst;
  input [7:0] l;
  output [7:0] r;
  logic [7:0] r;
  always @(posedge clk) begin
    if (rst) r <= 0;
    else begin
      if (en) r <= l;
      else r <= r + 1;
    end
  end
endmodule
```

Behavior記述

```
module dcnt(r, l, en, clk, rst);
  input en, clk, rst;
  input [7:0] l;
  output [7:0] r;
  logic [7:0] r;
  always @(posedge clk) begin
    if (rst) r <= 0;
    else begin
      if (en) r <= l;
      else #10 r <= r + 1;
    end
    $display(r);
  end
endmodule
```





Verilogという言語の特徴

- verilog はC言語やその原型といわれているpascalに似た文法を持つ
 - 混乱しなければsyntaxの学習そのものは非常に楽で常識が通じる
 - 例えば
 - 文の最後は;で終わります.
 - begin end で構文を作ります. (C言語は{}だが、pascal は begin end でブロックを構成)
 - コメントは, /* */ や, // が利用できる
 - 一方で型はずぼら
- module endmoduleで構造を記述
 - 違いは後程するが、簡単な説明として大構造がmodule
 - 小構造がbegin-end block
- 信号線
 - 変数という概念は一度捨てる
 - 変数は回路では配線およびレジスタ (FlipFlop)を意味する
 - レジスタの出力の配線と考えたほうがわかりやすい





信号線の接続

- 信号線の宣言の仕方
 - 普通の信号線(1bit幅)は宣言無しに利用できる
 - バス(複数bitの信号線で束)はwire宣言を用いるがここではlogicの利用を推奨
 - logic [3:0] a; // 4bit幅のaを宣言
 - logic [3:0] a, b; // 4bit幅のaとbを宣言
 - 入出力は入出力時に幅を宣言すればよい
 - input [3:0] a; // 4bit幅のaを入力として宣言
 - output [3:0] b; // 4bit幅のbを出力として宣言
- ある信号線を他の信号線に接続するにはassign文を用いる
 - assign y = a; // 信号aと信号yを繋ぐ
- バスからある線を取り出す場合は配列として扱う
 - logic [7:0] a;
 - logic [2:0] b;
 - assign b = a[5:3];





1. Verilog-HDLで記述されたファイルを用意する。通常拡張子は .v
2. DOS窓などシェルを開き、次のコマンドを実行してコンパイルする
`iverilog -g2012 verilog_file_1.v verilog_file_2.v verilog_file_3.v ...`
3. エラーがなければa.outという中間ファイルが生成される
 - 中間ファイルは可読形式
 - `iverilog -g2012 -o out_file verilog_file_1.v verilog_file_2.v verilog_file_3.v ...`として実行すると、out_fileなる中間ファイルが生成される
 - `iverilog -h` でヘルプが表示される
4. 次のコマンドでシミュレーションを実行できる
 - `vvp` 中間ファイル
 - 結果が画面に表示される。結果をファイルに残したい場合は
`vvp 中間ファイル > 結果ファイル` として、リダイレクトする
 - `vvp -h` でヘルプが表示される





Gtkwaveの使い方

1. テストベンチに次の記述を追加してシミュレーションを行う

- \$dumpfile("ファイル.vcd"); 波形表示用の出力ファイル（VCDファイル）を指定
- \$dumpvars(0, モジュール名); 第1引数：ダンプ開始時刻 第2引数：観測対象

<例>

```
$dumpfile("rsIt.vcd");
$dumpvars(0, adder);
```

シミュレーションの結果rsIt.vcdファイルが生成され、モジュールadder内の信号変化が保存される。モジュールまたは観測対象信号を指定できる

2. 次のコマンドを実行する。

- gtkwave rsIt.vcd
- 波形表示ウィンドウが表示され、メニューからSearch → Signal Search Hierarchyを選択、サブウィンドウで階層を展開して観測したい信号を指定、Appendボタンをクリックする
- タイムスケールを調整してシミュレーション結果を観測する





簡単なオペレーション

39





覚えるべきキーワード

40

- これだけで全て記述・設計可能(主にSystemVerilogを扱う)
 - 青の文法は合成には関係せずシミュレーションに関係する
 - module ~ endmodule
 - always @(...) begin ~ end, これに類するalways*, priority, unique(各full_case, parallel_caseに相当)
 - if(...) ~ else ~ (begin ~ end を伴う場合もある), priority, unique, 比較の方法
 - case ~ endcase (begin ~ end を伴う場合もある), priority, unique(priority, uniqueはivで未サポート)
 - input, output
 - wire, reg, logic (今後はlogicを推奨)
 - assign
 - 算術演算やシフト命令といったC言語由来の計算演算子, 比較演算子など(SVは++, --, +=なども利用可能)
 - 数の表記方法
 - [...], { ... }
 - initial begin ~ end
 - #数字
 - \$display, \$monitor
 - \$shm_open, \$shm_probe, \$dumpvars, \$dumpfile
 - \$finish
 - \$readmemb, \$readmemh





Verilogキーワードに見る特徴

- goto などの実行制御文がない
 - ハードウェア記述は基本的に実行順序は記述せず、回路の結合を記述する
 - とはいえ、順序記述は理解しやすいため順序で考えてよい場所がある
- always ブロックの中は上から下に評価され順序が存在する
 - ただし評価されるだけで実行という概念は本質を表していない。そのalwaysブロックの中の表記と同じ組み合わせ回路が合成される。実行という概念は捨てる
- 複数のalwaysブロック間は全て同時に動作する
 - ソースの後ろが先に評価されることも当然ありえる
- あくまでも回路を記述する
 - C言語などは、ある程度動けばどのように書いてもよいという側面がある
 - verilogなどハードウェア記述言語による設計は、動作するだけではなく、実態を設計しているため、書き方により回路の規模や動作速度が決定する
 - アーキテクチャとはよく表現した言葉で、建築構造物は例えば箱でよいが、それでは住むことはできないし、逆にごちゃごちゃしていても住みにくい





moduleを用いた簡単な回路例

- module文で機能モジュールを構成
 - より小さな構成要素はbegin endで構成されるブロック
 - ブロックが0個以上寄せ集めてモジュールを構成
 - 0個以上のモジュールを寄せ集めて、上位階層のモジュールを構成
 - 階層的に構築して全体を設計
- シンプルかつ完全な記述の例

```
module test(a, b);
    output b;
    input a;
    assign b = a;
endmodule
```

- この回路の意味を考えてみよう（既に簡単なオペレーションで紹介済みです）



- 一般的なverilogのテキストと違い、文法は必要最低限とし、記述に対する回路との対応を考えて説明する
- 回路と記述の対応をまずマスターすること
 - 回路を見れば記述できる、逆に記述すると回路が見えることが必要
 - これができるれば必要リソースや遅延が感覚的に判断できるようになる
 - 結果的に高品質な回路が設計できるようになる
- プログラミング言語として学んだ上で回路を設計すると、動作仕様は満たすが、目も当てられない回路が出来上がる場合がよくある

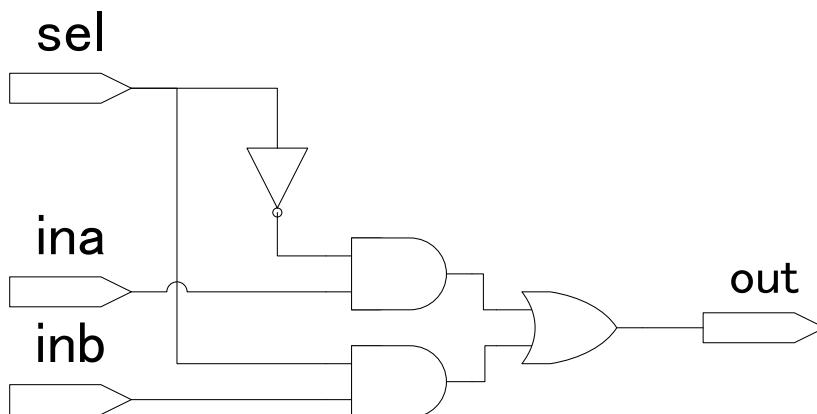




セレクタ

- セレクタは、論理回路で構成されたスイッチ
selが0であれば, ina の内容を, selが1であればinbの内容をoutに出力

回路記述(Schematic)



Verilog-HDLによる
Gate記述

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
assign out = (~sel & ina) |
(sel & inb);
endmodule
```

Verilog-HDLによる
RTL記述

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
logic out;
always_comb begin
//always @(sel or ina or inb) begin
if(sel) out = inb;
else out = ina;
end
endmodule
```





セレクタ記述のポイント

45

- これらは合成すると同じ回路が生成される
 - つまり見通しの立ちやすく、理解しやすいalwaysが優れているとわかる
- if文はセレクタを合成する
- if(sel) は if(sel != 0) と同じ、つまり、if(sel == 1)ということ
- if (条件) begin … endが基本
 - 条件に合致したら評価
 - 動作内容記述が1行の場合はbegin…endを省略できる
 - if (条件) begin … end else begin … endなど、C言語と同じ構造で記述
 - 入れ子も当然可能
- ここで、**module, input, output, always, if** の使い方をマスターすること





演習問題（1）

- 演習問題(1-1)
 - 次の記述に間違いがあれば訂正しなさい
 - input a, b;
 - module test(x, c, f);
 - assign y = s;
 - wire m[3:0], p;
 - output z, [1:0] w
 - assign s[3:0] = a [4:2];
 - end module;
 - output {2:0} f;
 - 次の値は何ビットのデータで、10進数の値は何か答えなさい
 - 5'b10
 - 'b1010
 - 8'h2c
 - 'h2c
 - 'd18





演習問題（1）

- 演習問題(1-2)

- Icrus Verilogの環境を構築しなさい（既に構築しているはずです）
- 次のサンプルプログラムを test.vとして記述し、iverilog test.vとしてエラーが出ないことを確認しなさい
- 別途editorが必要で、メモ帳やnotepad++、vimがお勧めである。
- この記述の意味を常識的に考えなさい。
 - aがHigh(1)のときbはどうなるか？
 - aがLow(0)のときbはどうなるか？

```
module test(a, b);
    output b;
    input a;
    assign b = a;
endmodule
```

- 注意事項

- レポートをMicrosoft Wordファイルで作成、keio.jpへ提出すること
- A4 2枚以内で作成し、最初にタイトルを「演習問題（2）」として記載し、名前と学籍番号も忘れずに記載すること
- 提出締め切りはLMSを確認すること





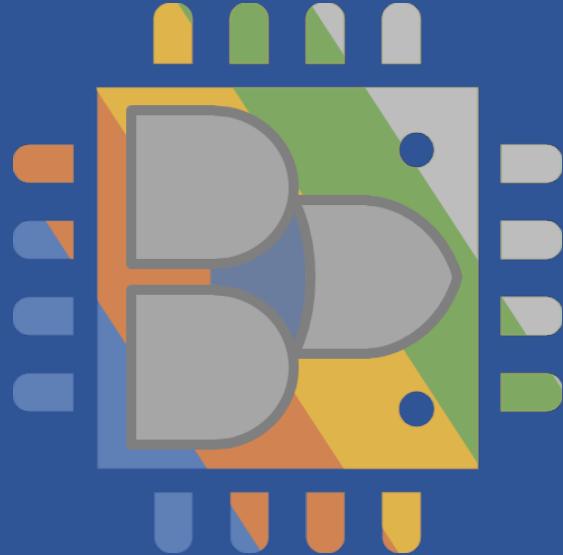
全てのレポートに供するする注意事項

48

- コードが貼り付けられていないレポートは点数を獲得できません
- コードはテキスト情報ではりつけてください
 - 画像で貼り付けている場合は、こちらで実行確認ができないため、コードを貼り付けていないレポートと同様に採点されます
- 実行を証明するように、gtkwaveの画面も貼り付けてください
- pdfを作成したら提出する前に、pdfを自身で確認してください
 - コードをドラッグ、コピーしてメモ帳などに張り付けたとき、きちんとテキストが保存されていることを確認してください
 - 採点ではpdfからコードをpdf2text|verilog-filter で自動抽出しますので、この動作に支障が出た場合は減点となります

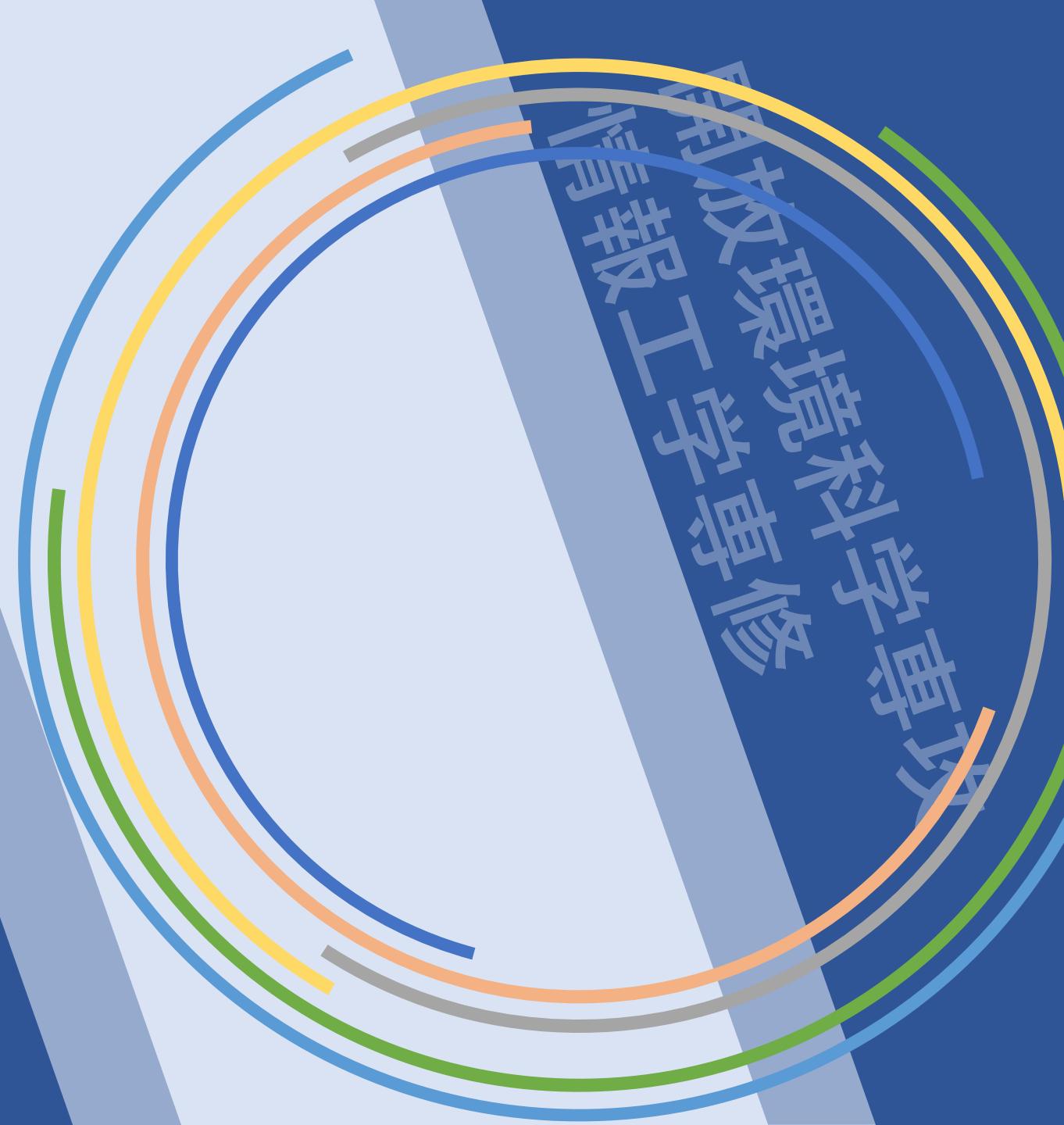
以上、よろしくお願いします





計算機システム 設計論(3) Combination Logic

担当： 西 宏章





moduleとblockの記述

50

- moduleは回路のあるモジュール(ブロックの集合体)を構成する
- 通常、入力と出力を伴う
 - ただし、テストモジュールは入出力が通常存在しない
- alwaysは回路ブロックを構成する
 - alwaysの中はC言語などと同様に評価され、その評価結果に等価な組み合わせ回路が生成される
 - つまり、値が確定しないalwaysはそもそもダメ！すべての値が確定すること！
- センシティリスト(信号線名をorで繋げたリスト)は該当するalwaysブロックが評価される条件
 - ただし、この意味が有効であるのはシミュレーションのみ、合成時は意味が薄れる
 - この違いが、論理シミュレーションと合成後の動作が異なるという問題を生み出す
 - センシティリストは面倒だ、そこでSystemVerilogの登場
 - alwaysブロックの入力と考えられる信号線を全てセンシティリストに記載
 - always @(*) やalways @*を用いる、さらにalways_combを用いる(推奨)





セレクタ回路記述

良くない記述

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
assign out = (~sel & ina) |
(sel & inb);
endmodule
```

古い記述スタイル

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
reg out;
always@(ina or inb or sel) begin
  if(sel) out = inb;
  else out = ina;
end
endmodule
```

一般に利用できるスタイル

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
reg out;
always@(*) begin
  if(sel) out = inb;
  else out = ina;
end
endmodule
```

```
module sel (ina, inb, sel, out);
input ina, inb, sel;
output out;
reg out;
always@* begin
  if(sel) out = inb;
  else out = ina;
end
endmodule
```

SystemVerilogスタイル

```
module sel (
  input ina, inb, sel, output out);
  logic out;
  always_comb begin
    if(sel) out = inb;
    else out = ina;
  end
endmodule
```





組み合わせ回路記述のルール (超重要)

52

- 合成を意図しないなら、どう書いてもよい
- もし合成を意図するなら、全ての状態を記述すること**

```
always_comb begin  
    if(a == 0) b = 1;  
end
```

- この記述は回路合成するとラッチが生成されエラーになる
- なぜか？ $a=0$ のとき $b=1$ は良いが、 $a=1$ のときはどうなるか？
 - 値が確定しない。すると仕方なく、「 $a=1$ の時は、以前の b を保存するんだ」と解釈し LATCH (ラッチという回路) を生成する。ライブラリにラッチが含まれていない場合、エラーとなる
- 次のどちらかの方法ですべての状態の値を確定させること
- 簡単な場合は全条件を記述する左、複雑な always はデフォルト代入の右がお勧め

```
always_comb begin  
    if(a == 0) b = 1;  
    else b = 0;  
end
```

```
always_comb begin  
    b = 0;  
    if(a == 0) b = 1;  
end
```

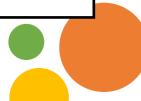
レポートでは恐ろしいぐらいにこのミスが出てくる





- 次のコードはselがセンシティブリストに含まれていない
 - 合成すると先ほどと同じselが生成される(警告メッセージが出力されるであろう)
 - シミュレーションするとselが変化した時にoutが変化しない記述になる
 - 例えば、 $ina = 1, inb = 0$ のとき、selを変化させても初期値から変化しないという現象が発生する可能性がある。このような不完全な記述の動作は処理系に依存する
 - 記載の意味そのままに捉えた回路は標準の回路では実現できないため合成できない**
(and, or, xorなどの論理ゲートやFF, Latchといったレジスタでは安定して実現できない)
 - 合成できる記述を心掛ける
 - Latchを使いたい時はalways_latchを使う
- behavior 記述はこの限りではない
 - 合成しない前提であるため何をしてよい
 - ただしあまり無茶をすると処理系で結果が異なる
 - 並列プログラムが処理系で処理の終了時刻が変化するのと同じ

```
module selmodoki(ina, inb, out, sel)
input ina, inb, sel;
output out;
reg out;
always @(ina or inb) begin
  if(sel) out = inb;
  else out = ina;
end
endmodule
```





alwaysによる組み合わせ回路合成について

- 以前は組み合わせ回路合成にfunctionを使うという文化があった
 - オワコンfunctionだが、void型を返すfunctionが仕様として加わった（神）
 - 右下の例をみると、void functionが使えることがわかる！function復活？
- always_comb, always_ff(always @(posedge)), always_latchを使おう
 - always_combは実行時一度実行される。
 - always @*はセンシティブリストのみ、つまり無駄に評価されて遅い
 - always_combは中身でセンシティブリストを構築。always *はfunctionなどに従う
 - always_combは複数のalways_combでの値代入をエラーで避ける！（神）**
 - そういう意味で、always_ffを使うこともお勧めする

```
module test; //できてしまう
logic [1:0] a, b;
always @@
  a = b;
always @@
  a = b+1'b1;
endmodule
```

```
module test; //エラーになる
logic [1:0] a, b;
always_comb
  a = b;
always_comb
  a = b+1'b1;
endmodule
```

```
always_comb begin
  step_a();
  step_b();
end
function void function a();
endfunction
function void function b();
endfunction
```

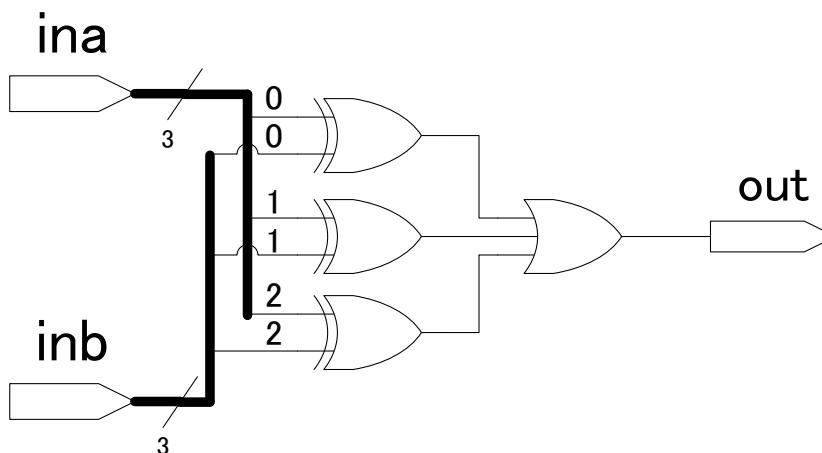




コンパレータの構成

- コンパレータは、inaの内容とinbの内容を比較する回路
- 例えばina, inb共に3bitであった場合のコンパレータは次の通り
 - 合成するとすべて同じ回路が生成される

回路記述(Schematic)



論理式による記述

```
module compeq(ina, inb, out);
    input [2:0] ina, inb;
    output out;
    logic out;
    always_comb begin
        out = (ina != inb);
    end
endmodule
```

alwaysによる記述

```
module compeq(ina, inb, out);
    input [2:0] ina, inb;
    output out;
    logic out;
    always_comb begin
        if(ina == inb) out = 1'b1;
        else out = 1'b0;
    end
endmodule
```





足し算回路の構成

- 足し算回路の基本は、桁ごとのシンプルな足し算回路をつなげること
 - 足す数・足される数・下の桁からの繰り上がりから、答えと繰り上げを計算する回路、すなわち、1bit加算器(フルアダ一 full adderと呼ばれる)を構成
 - Full adderをn個並べてnビット加算器全体を構成
- このようにして構成した加算器をリップルキャリーアダ一(ripple-carry adder)と呼ぶ
 - 論理回路の段数が多く T_{pd} が大きくなることから実行速度が遅い
- 回路規模を増やしても速く計算したい場合はキャリールックアヘッド(carry look ahead adder)と呼ぶ加算器を構成

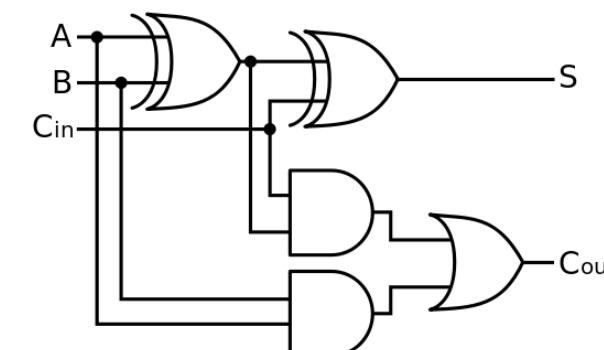




Full adder

- $s(\text{答え}) = a(\text{足される数}) + b(\text{足す数}) + ci(\text{繰り上がり});$
 - S はその位の答え s と繰り上がり co で構成される
 - 等価な動作をする回路は次の通り
 - s が1になるのは、 a, b, ci の組み合わせで1の数が奇数個の時
 - co が1になるのは、
 $(a, b, ci) = P_1(0, 1, 1), P_2(1, 0, 1), P_3(1, 1, 0), P_4(1, 1, 1)$ の時
 カルノー図で圧縮してもよいが、 P_1 とハミング距離が1なのは、
 P_4 で共通項は $b \& ci$ 、同様に P_2 は P_4 で $a \& ci$ 、 P_3 も P_4 で $a \& b$ 、
 以上で P_1 から P_4 のすべてを網羅していることから
 $co = a \& b | a \& ci | b \& ci$ となる

```
module faddr (a, b, ci, s, co);
  input a, b, ci;
  output s, co;
  assign s = a ^ b ^ ci;
  assign co = a & b | a & ci | b & ci;
endmodule
```



a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

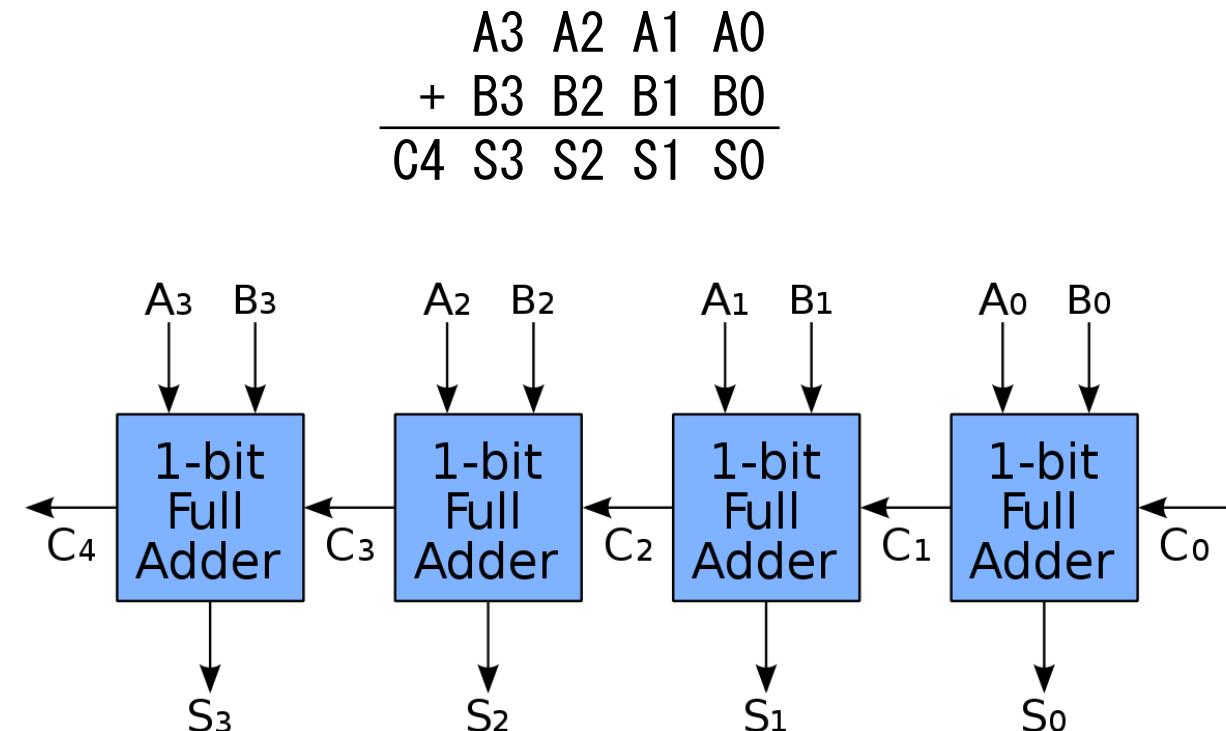




Ripple Carry Adder

- full adderを並べることで何ビットでも2進数加算器が構成できる
 - 次の記述のように回路図の階層構造と同様、moduleを他のmoduleで利用できる
 - moduleを参照する場合は、あらたに名称(ここではf0, f1, f2, f3など)を付ける
 - moduleはクラスで、オブジェクト指向のインスタンス化と類似

```
module ripple_adder(a, b, s, co);
    input [3:0] a, b;
    output [3:0] s;
    output co;
    logic [3:0] c;
    assign c[0] = 0;
    faddr f0(a[0], b[0], c[0], s[0], c[1]);
    faddr f1(a[1], b[1], c[1], s[1], c[2]);
    faddr f2(a[2], b[2], c[2], s[2], c[3]);
    faddr f3(a[3], b[3], c[3], s[3], co);
endmodule
```





引き算

- 足し算ができれば引き算もできる
 - 補数表現を使えば足し算で引き算が計算できる
 - $5-2 = 5+(-2)$
 - -2、すなわち補数は、ビット反転して1を加えることで求める
 - 4'b0010 (2)
 - 4'b1101 ($4'hB=11$)ビット反転する
 - 4'b1110 ($4'hE=-2$)1加える
 - $4'b0101(5)+4'b1110(-2)=5'b10011$
 - 4ビットでは、 $4'b0011=3$ となり、 $5-2=3$ が計算できる
- 補数を求める回路は、ビット反転(Inveter)と足し算回路で構成できる
 - 結局足し算・引き算回路は、足し算だけで構成できる

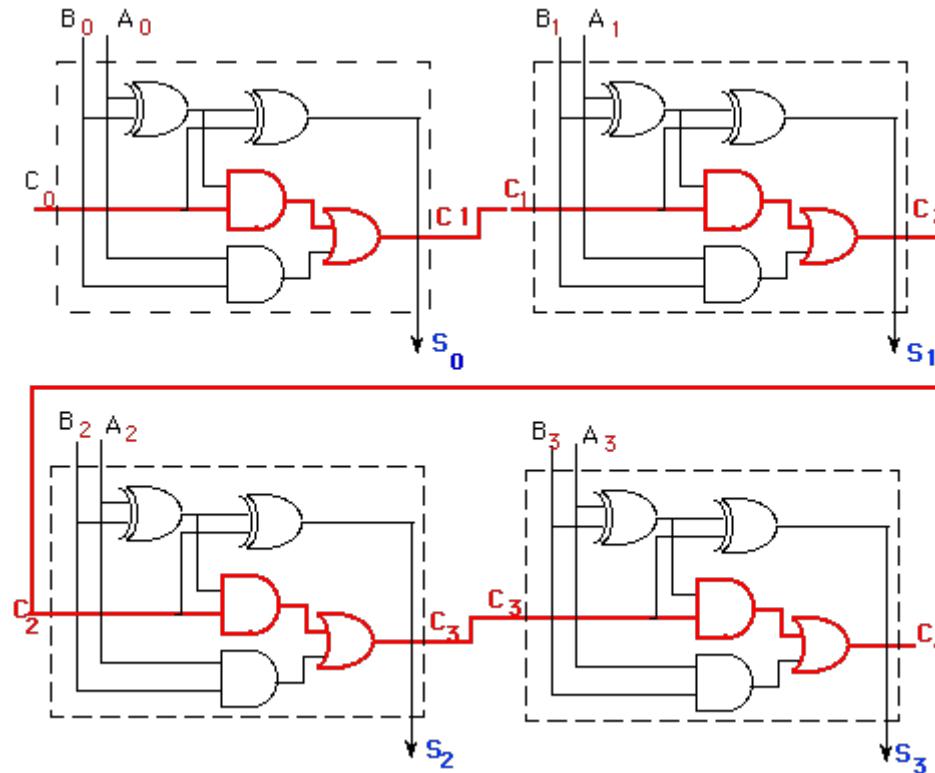




Ripple Carry Adderの問題点

60

- 構成した足し算器で $c[4]$ を求める論理段数（クリティカルパス）を調べる



- クリティカルパスが長い = 回路の動作遅延が長い = 計算能力が低下する





Carry Look Ahead

61

- carryだけならば早く計算できるため、carryだけ別に計算して計算能力を向上させるのがcarry look ahead

- carryはaとb両方が1か、どちらかが1でcarryが1のときに1

書き直すと $Cout = C_{i+1} = (A_i \& B_i) | (A_i \wedge B_i) \& C_i$

- ここで、 $G_i = A_i \& B_i$ および $P_i = A_i \wedge B_i$ とすると

- 次の式を得る $C_{i+1} = G_i + (P_i \& C_i) \cdots (1)$

これを再帰的に当てはめる

- $C_1 = G_0 + P_0 \& C_0$ ($i = 0$ とした)

- $C_2 = G_1 + P_1 \& C_1 = G_1 + P_1 \& (G_0 + P_0 \& C_0) = G_1 + (P_1 \& G_0) + (P_1 \& P_0 \& C_0)$

- $C_3 = G_2 + (P_2 \& G_1) + (P_2 \& P_1 \& G_0) + (P_2 \& P_1 \& P_0 \& C_0)$

- $C_4 = G_3 + (P_3 \& G_2) + (P_3 \& P_2 \& G_1) + (P_3 \& P_2 \& P_1 \& G_0) + (P_3 \& P_2 \& P_1 \& P_0 \& C_0)$

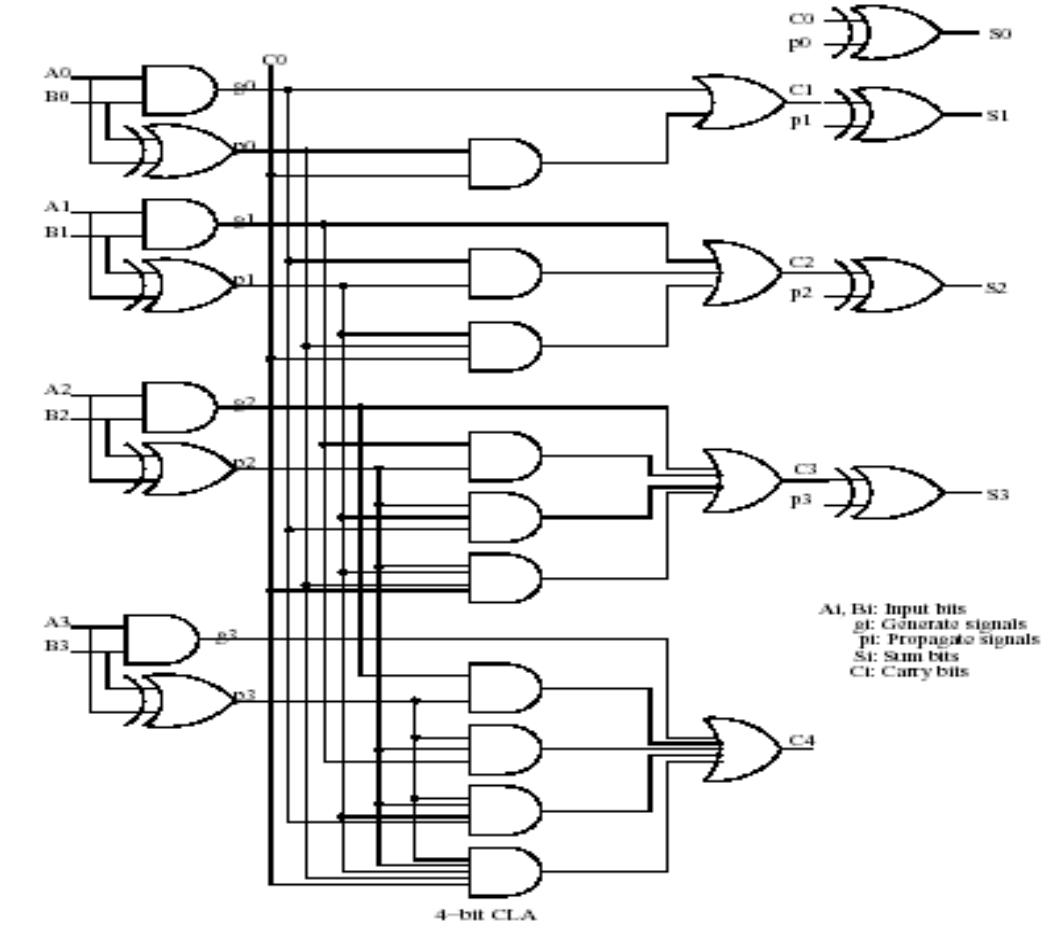
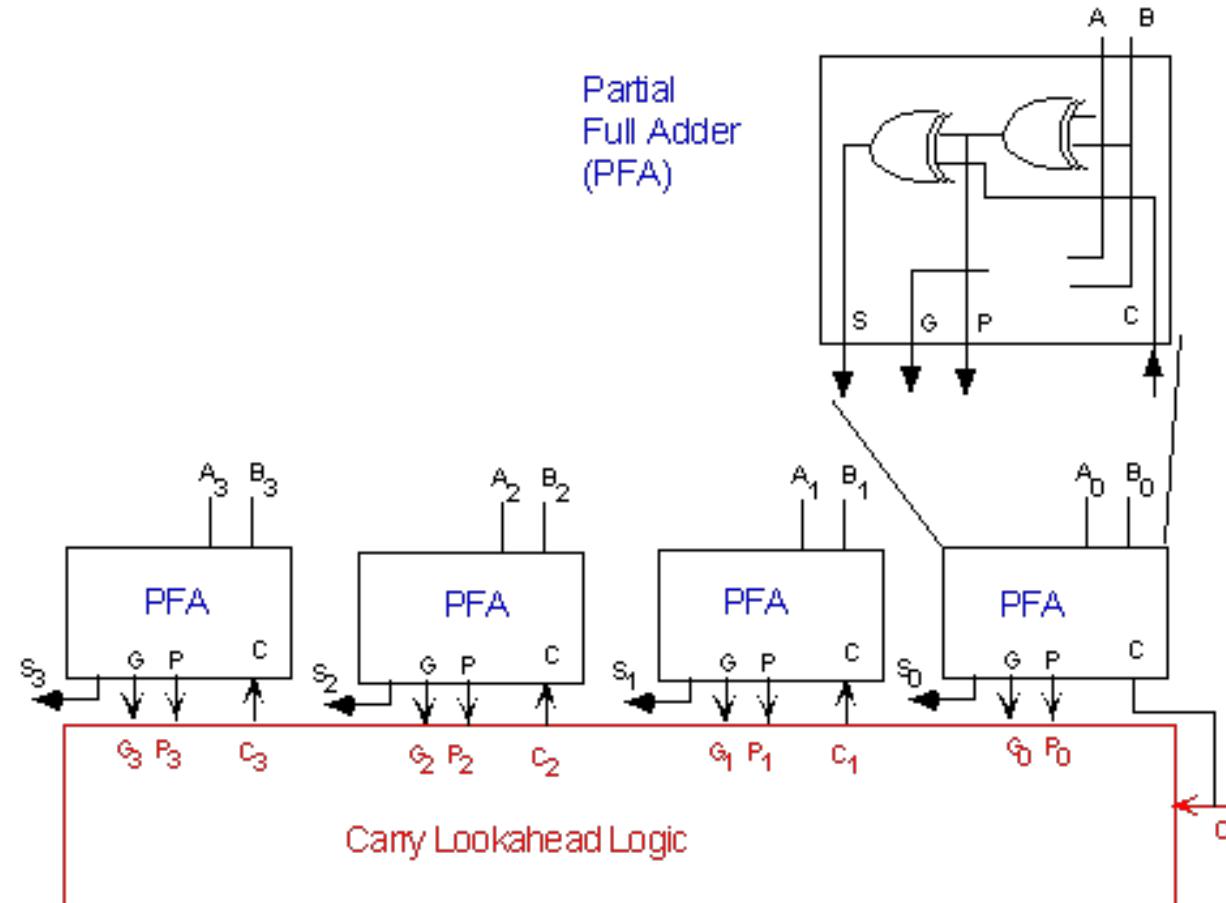
a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





Carry Look Aheadの構造

62

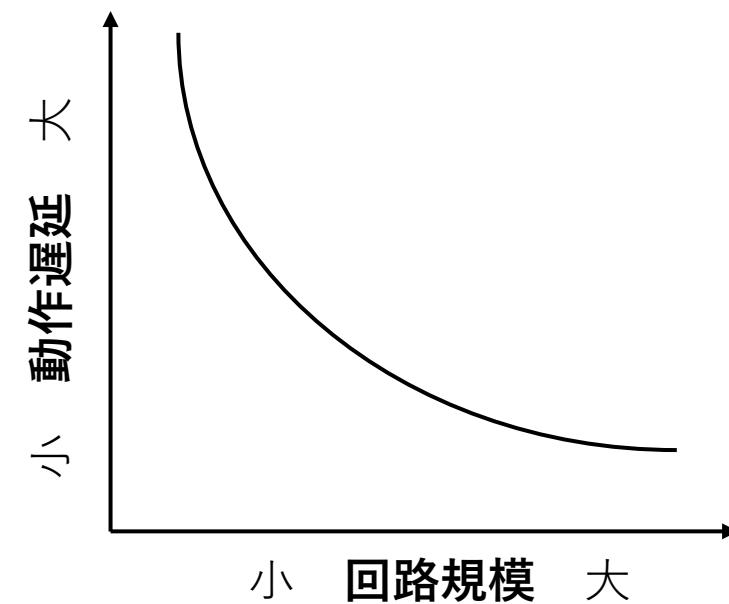




小さい回路が速いという迷信

63

- まず小さい回路が構成できる時点で、元の回路は無駄であったということ
 - その小さい回路があるという状況において、
 - その回路をさらに速く(遅延を短く)したいとするとどうするか？
- 方針1：伝搬遅延を短くするために並列化する→回路規模増大
- 方針2：CMOSならばドライブ能力を上げるために分割する→回路規模増大
-
- 結局、低遅延化は回路規模を増大させる方向に進む
 - 右の図でどこを狙うかは設計者次第
 - 線上のどの点も「最適設計」
 - これをいちいち回路を変更しては大変
 - verilogで高次元設計し合成系に任せればよい





足し算回路も同じ、Verilogに任せよう

64

- つまり
 - 回路規模が小さく遅いRipple Carry Adder
 - 回路規模が大きく速いCarry Look Ahead
 - 構成を混合させて、回路規模と遅延のバランスを考える必要がある
- これが難しい
 - 設計者は与えられた遅延内で、できるだけ回路規模の小さな回路を設計したい
 - 他の設計に変更があると、要求遅延も変化するため決め打ちで設計できない
- ここにVerilogの利点がある
 - Verilogでは+と書くだけ！ assign s = a+b;
 - どのように混ぜるかは合成CADに任せる
 - 設計変更コストも、遅延の設定の手間も縮約できる





フリップフロップの構成

- 組み合わせ回路はまずはこの程度にして次に記憶素子の記述を学ぶ
 - 記憶素子を覚えれば、基本的には何でも記述ができるようになる
- Dフリップフロップの記述は次の通り
- SystemVerilogでは次が推奨だが得はしていない
- 次の記述と区別すること
- これらは次の記述と同じ

```
logic q;  
always @(posedge clk) begin  
    q <= d;  
end
```

```
logic q;  
always_ff @(posedge clk) begin  
    q <= d;  
end
```

```
logic q;  
always @(d) begin // always @(*), always_comb  
    q = d;  
end
```

```
logic q;  
assign q=d;
```





Dフリップフロップの意味

- 記述内容をそのまま理解すると次の通り
 - クロック(clk)が立ち上がったとき、すなわち、0から1へ変化した瞬間(これをposedgeと呼ぶ)のみalwaysの中を評価する
 - alwaysが評価されると、出力qが入力dに等しくなり、それ以外はなにもしない
 - ここで、=ではなく、<=と記載されているが、ここでは「同じ」と考えてください
 - FFを合成する場合は、<=と表記します。違いは改めて説明します
 - 合成するとフリップフロップが生成される
- このように、alwaysやalways_ffでposedge clkを指定するとFFを合成
 - 自由に記述するとエラーになる場合もあるので、以降FFを用いた回路の高機能化について説明する

```
logic q;  
always @(posedge clk) begin  
    q <= d;  
end
```



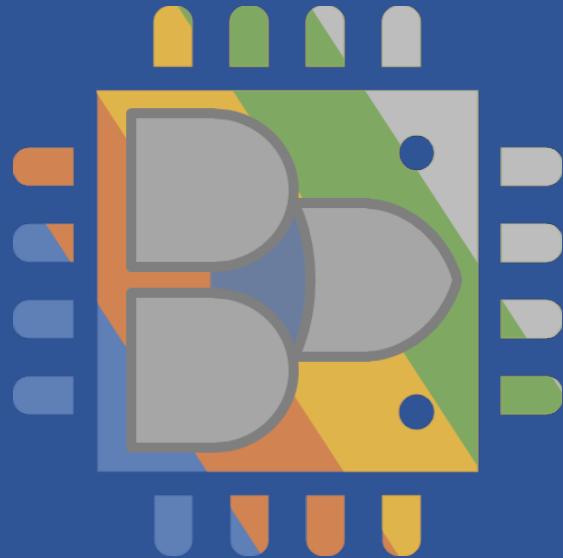


演習問題（3）

67

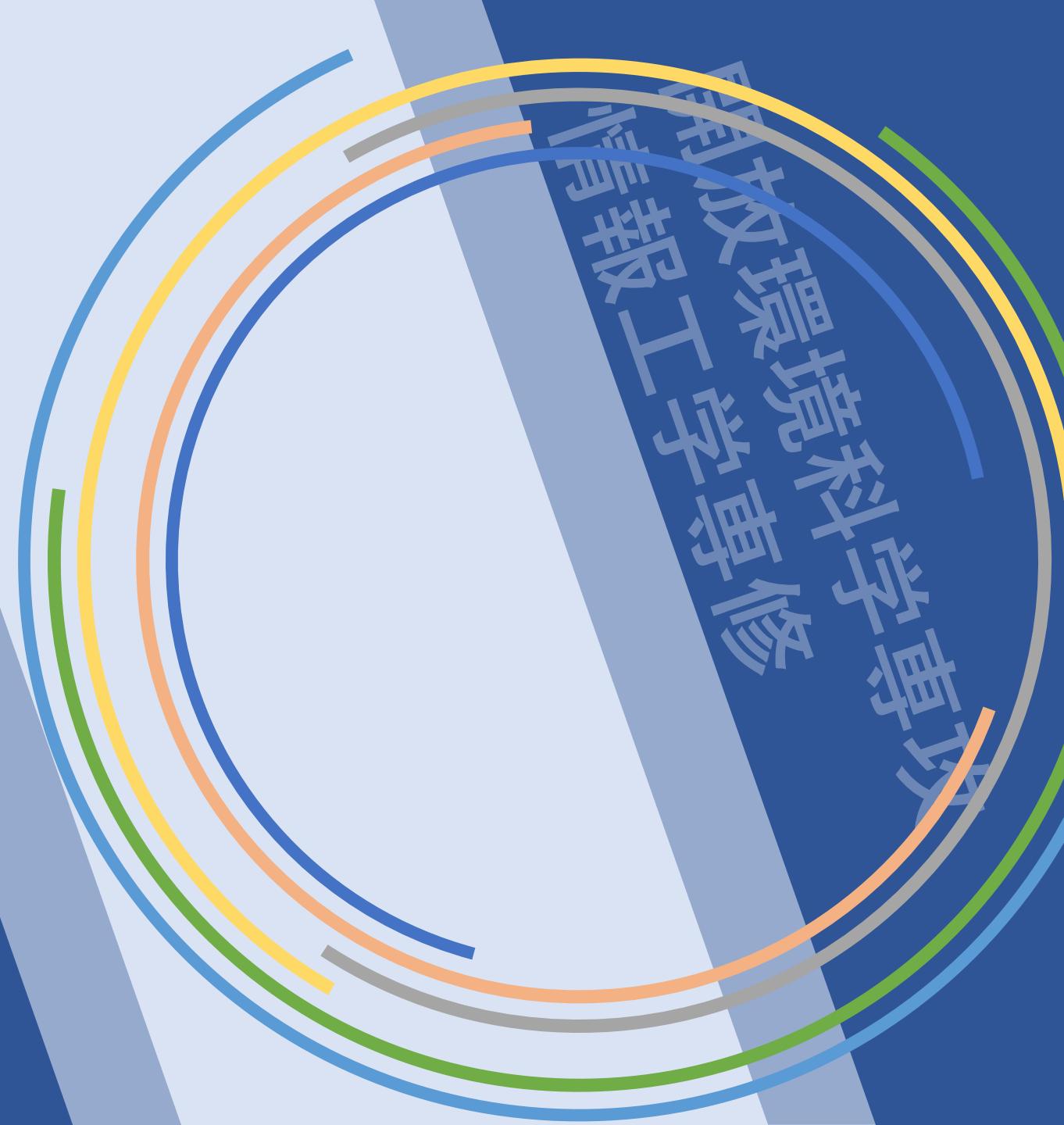
- 演習問題(2-1)
 - 1bit Full-adderを用いて4bitのRipple-carry Adderを構成し、動作を確認しなさい
 - このFull-adderを用いて、引き算演算子を用いず、補数を使った足し算器のみによる引き算器を構成し、動作を確認しなさい
- 演習問題(2-2)
 - 3bitのaおよびbがあるとき、 $a > b$ や $a >= b$ を求める回路はどのような回路になるか、実際に求めてその回路規模を体感しなさい
- 注意事項
 - レポートをMicrosoft Wordファイルで作成、LMSへ提出すること
 - A4 2枚以内で作成し、最初にタイトルを「演習問題（3）」として記載し、名前と学籍番号も忘れずに記載すること
 - Verilogソースコードとgtkwaveなどシミュレーションの画面を含めること
 - 提出締め切りはLMSを確認すること





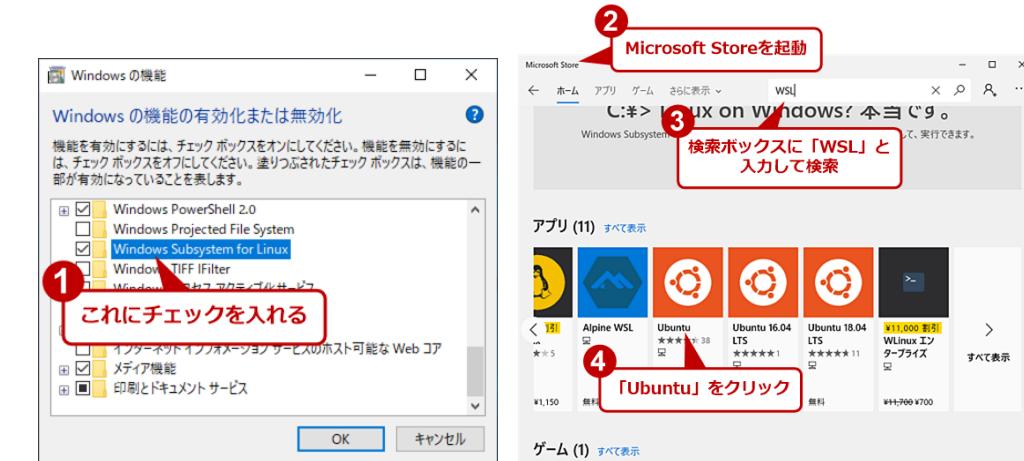
計算機システム 設計論(4) Register

担当： 西 宏章



余談ですが

- Windowsを利用して
• やっぱりLinuxのコマンドが使いたいなあ
- 方法1
 - WSLを導入する。おすすめ度B
 - apt install iverilogで導入すると古いので一部文法が使えない。最新をコンパイル
 - DISPLAY環境変数を設定しないとダメ。 gtkwaveはwindowsの方が綺麗
- 方法2
 - Cygwinを入れる。おすすめ度C
- 方法3
 - busyboxを入れる。おすすめ度B <http://frippery.org/busybox/>
 - --install で必要なバイナリがすべて入る（単純に山のようなコピーができる）
 - Windowsとして使える。これで十分。shも使える。
- 方法4
 - git bashを入れる。おすすめ度A <https://gitforwindows.org/>
 - 後々利用しますがgitを入れるついでに入るbashと周辺コマンド群
 - ほぼネイティブなbash環境を提供、ジョブコントロールも一部可能で便利





フリップフロップの構成

- 組み合わせ回路はまずはこの程度にして次に記憶素子の記述を学ぶ
 - 記憶素子を覚えれば、基本的には何でも記述ができるようになる
- Dフリップフロップの記述は次の通り
- SystemVerilogでは次が推奨だが得はしていない
- 次の記述と区別すること
- これらは次の記述と同じ

```
logic q;  
always @(posedge clk) begin  
    q <= d;  
end
```

```
logic q;  
always_ff @(posedge clk) begin  
    q <= d;  
end
```

```
logic q;  
always @(d) begin // always @(*), always_comb  
    q = d;  
end
```

```
logic q;  
assign q=d;
```





Dフリップフロップの意味

- 記述内容をそのまま理解すると次の通り
 - クロック(clk)が立ち上がったとき、すなわち、0から1へ変化した瞬間(これをposedgeと呼ぶ)のみalwaysの中を評価する
 - alwaysが評価されると、出力qが入力dに等しくなり、それ以外はなにもしない
 - ここで、=ではなく、<=と記載されているが、ここでは「同じ」と考えてください
 - FFを合成する場合は、<=と表記します。違いは改めて説明します
 - 合成するとフリップフロップが生成される
- このように、alwaysやalways_ffでposedge clkを指定するとFFを合成
 - 自由に記述するとエラーになる場合もあるので、以降FFを用いた回路の高機能化について説明する

```
logic q;  
always @(posedge clk) begin  
    q <= d;  
end
```





同期リセット付きフリップフロップ

72

- FFを高機能化する
 - 同期リセット付きフリップフロップ

```
logic q;  
always @(posedge clk) begin  
    if(rst) begin  
        q <= 1'b0;  
    end else begin  
        q <= d;  
    end  
end
```

```
logic q;  
always @(posedge clk) begin  
    if(rst) q <= 1'b0;  
    else q <= d;  
end
```

- rst信号線がhighになった場合でも、クロックが立ち上がらないと0クリアされない
- このように、「意味上・解釈上の動作」が「実際の回路の動作」と一致していれば、その回路に合成される
- コンパイラが解釈しやすいように記載するべき
 - あまり気にしなくてもよいが、回路の種類が限られているということは、意味上・解釈上の動作もそれによって制限されるということに気を付ける
 - だから、回路を知らないと、存在する・しないが判断できない





非同期リセット付きフリップフロップ

73

- 次のようにposedge rstを追加する
 - 記述の意味を解釈すると同じであることがわかる

```
logic q;  
always @(posedge clk or posedge rst) begin  
    if(rst) begin  
        q <= 1'b0;  
    end else begin  
        q <= d;  
    end  
end
```

- 次のように記述すると合成できない
 - シミュレーションはできてしまう
 - 意味としては、clkとrstという2つのクロックがあり、両方で反応するFF
 - そんな回路は普通準備されていない

```
logic q;  
always @(posedge clk or posedge rst) begin  
    q <= d;  
end
```





メモリ

- さらに高度なFFの応用回路としてメモリを設計する

- メモリを構成するには、配列を利用

```
logic [4:0] mem [3:0];
```

- これが3:0つまり、4個並んだ構造となる

```
logic [4:0] mem;
```

- Verilogの添え字は全て「個数」であることに注意

アドレス幅が3:0の4bitで、16個と思わないように（これは無駄であり得ない）

- アクセスの仕方

- 2番地の内容を読み出す

```
logic [4:0] out;  
assign out = mem[2];
```

- 3番地に内容を書き込む

```
logic [4:0] in;  
always @(posedge clk) begin  
    mem[3] <= in;  
end
```



メモリ記述

- 完全なメモリ記述は（完全Dual Portメモリ）次の通り

- アドレス・データ

- 読み込みアドレス ra
- 書き込みアドレス wa
- 読み込み値 rd
- 書き込み値 wd
- 書き込み we

```
assign rd = mem[ra];
always @(posedge clk) begin
    if(we) mem[wa] <= wd;
end
```

- より一般的な（半導体デバイスとして販売している）メモリ

- ピン数が多くなるため、省略されている
- アドレス線共通化
- データ線を双方向として共通化
- アドレスとデータ線を共通化

などなど、ピンネックを解消する工夫が施されている





カウンタ

- ・ カウンタとは0から順に数え上げるなど、値を変化させるハードウェア
 - ・ シンプルに毎クロック0から7まで数えて7の次に0に戻る (wraparound) 場合
 - ・ 型を守っていれば、中は何を書いてもよい
- ・ このカウンタにイネーブル線をつける

```
logic [2:0] count;  
always @(posedge clk) begin  
    if(rst) count <= 0;  
    else count <= count + 1;  
end
```

```
logic [2:0] count;  
always @(posedge clk) begin  
    if(rst) count <= 0;  
    else  
        if(en) count <= count + 1;  
end
```





アップダウンカウンタ

- ・ イネーブルつきアップダウンカウンタ
 - ・ udという信号がhighのときup動作
 - ・ udがlowのときdown動作
- ・ プリセッタブルダウンカウンタ(タイマ)
 - ・ 今度はラップラウンドなし
 - ・ enがlowのときvalでタイマの値を与える
 - ・ enがhighでダウンカウントを開始0で停止

```
always @(posedge clk) begin
    if(rst) count <= 0;
    else
        if(en) begin
            if(up) count <= count + 1;
            else count <= count - 1;
        end
    end
```

```
always @(posedge clk) begin
    if(rst) count <= 0;
    else
        if(en) begin
            if(count != 0) count <= count - 1;
        end
        else
            count <= val;
    end
end
```

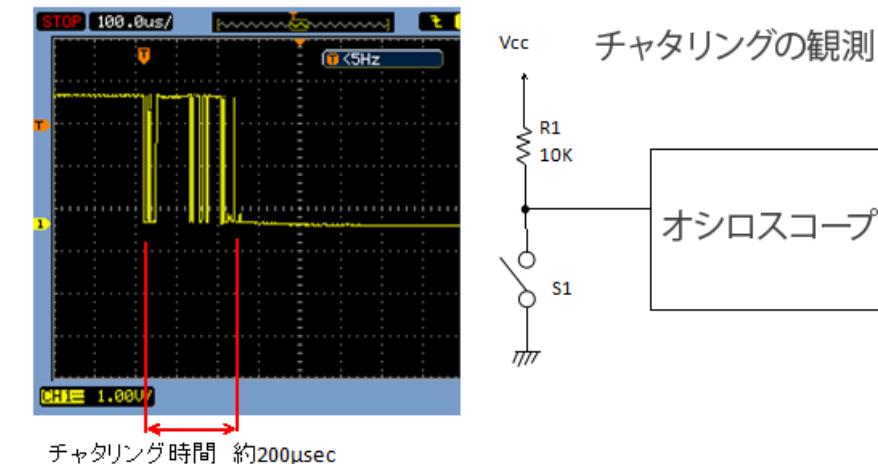




チャタリング防止回路

78

- トグルスイッチや押しボタンスイッチなど、メカニカルスイッチや、リレーはチャタリングと呼ばれる、複数回のON/OFF現象が微小時間に発生
- 原因は接点近接時の放電やバウンド
- デジタル回路の動作速度はチャタリングの周波数よりも高くON/OFFが見える
- 意図と異なる動作となる



- カウンタを用いたチャタリング防止
 - ONを検出したときにカウンタを戻す
 - カウンタがwraparoundするまで次のONを受け付けない
 - 実際にはカウンタを用いるのはかなり大げさであるが、動作速度が速い場合で、直接スイッチ入力を扱う場合は必要となる
 - 例えば、チャタリング時間は数十msに及ぶこともある
 - 一般には良いスイッチを使うことで、1ms未満に抑えることができる

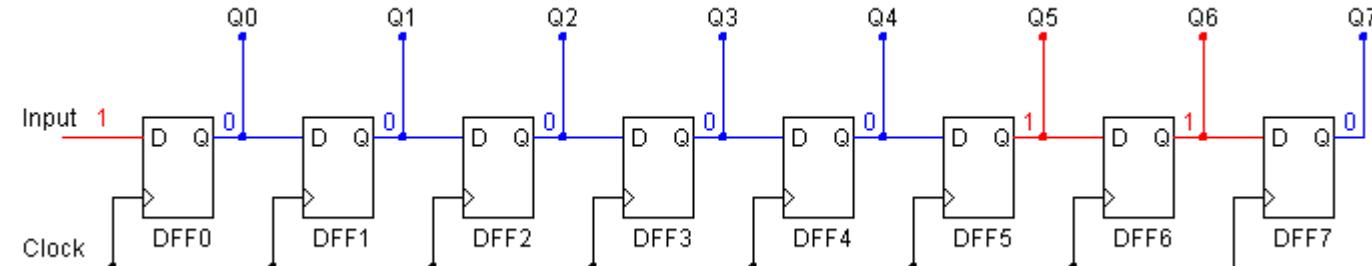




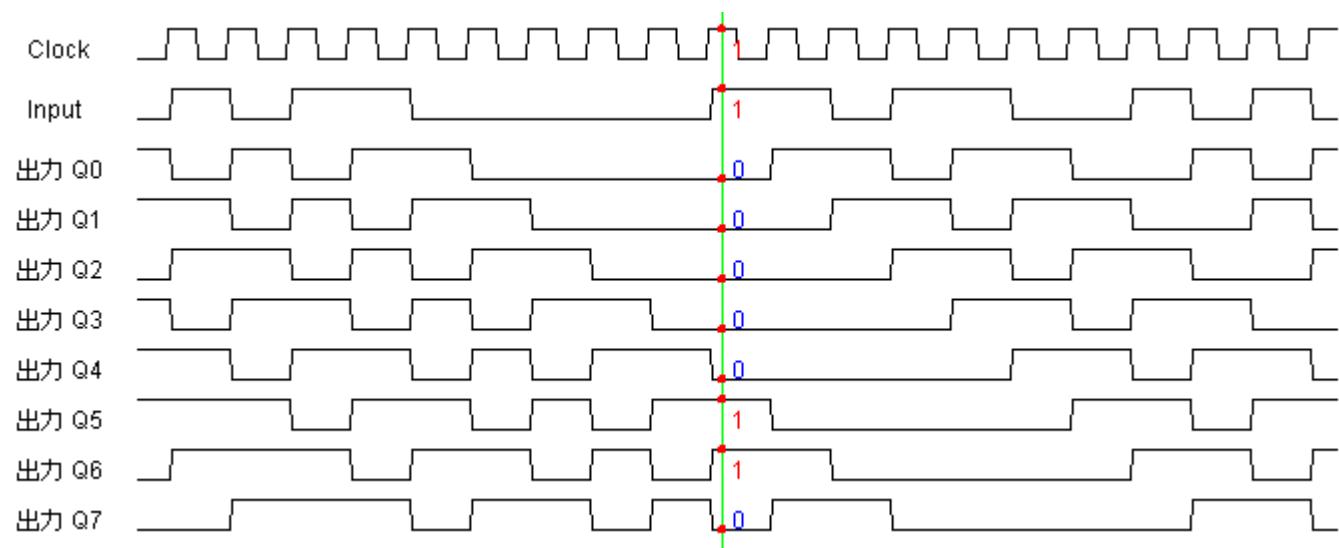
シフトレジスタ(バーレルシフタではない)

79

- シリアル・パラレル変換や、パラレル・シリアル変換などで用いる
- D-FFを順に接続して並べる



- 入力を変化させると、その変化が順に伝搬する



- 結果的にシリアル信号をパラレル信号に変換できる
 - この例では、01100000という信号が再現されている





シフトレジスタの記述

80

- 記述はシンプル

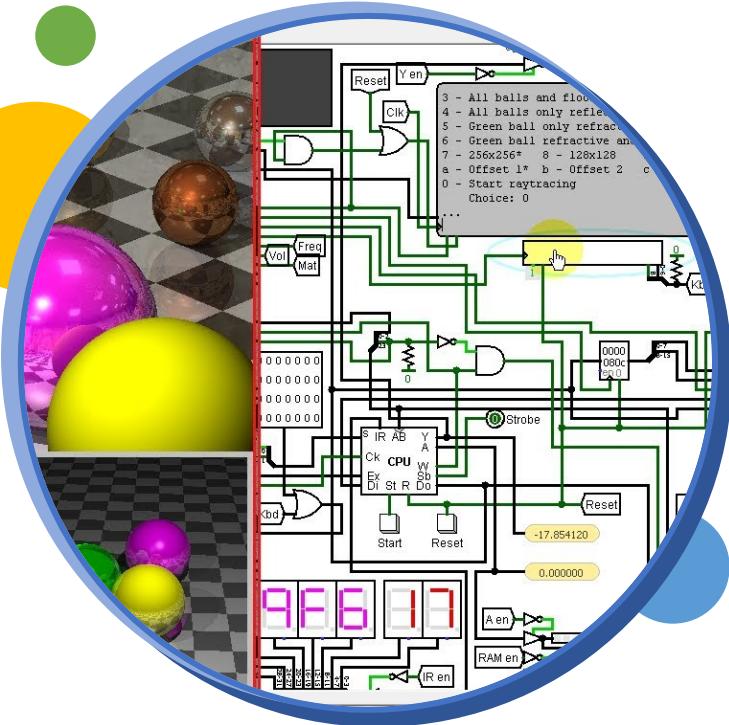
```
logic [7:0] sr;
always @(posedge clk) begin
    if(rst) sr <= 0;
    else begin
        sr <= {sr,in};
    end
```

- Logisimで動作を確認しよう



Logisimによる実習

- 回路設計・シミュレータを使おう





Logisimによる実習

- ・今日はオンライン実験であることを考慮し、FPGAの実機を使わず、シミュレータによる実習を行う
- ・Logisimによる実習を行う
 - ・グラフィカルな論理回路設計・シミュレータ
 - ・信号レベルの(0/1)や(Low/High)により配線の色が変わるために理解しやすい
 - ・階層設計が可能で、頑張れば複雑な回路も設計できる(ファミコンも作れる)
- ・欠点
 - ・標準では波形入出力ができるないが、テキスト形式のLogファイルを介して外部ツールを用いて確認できる

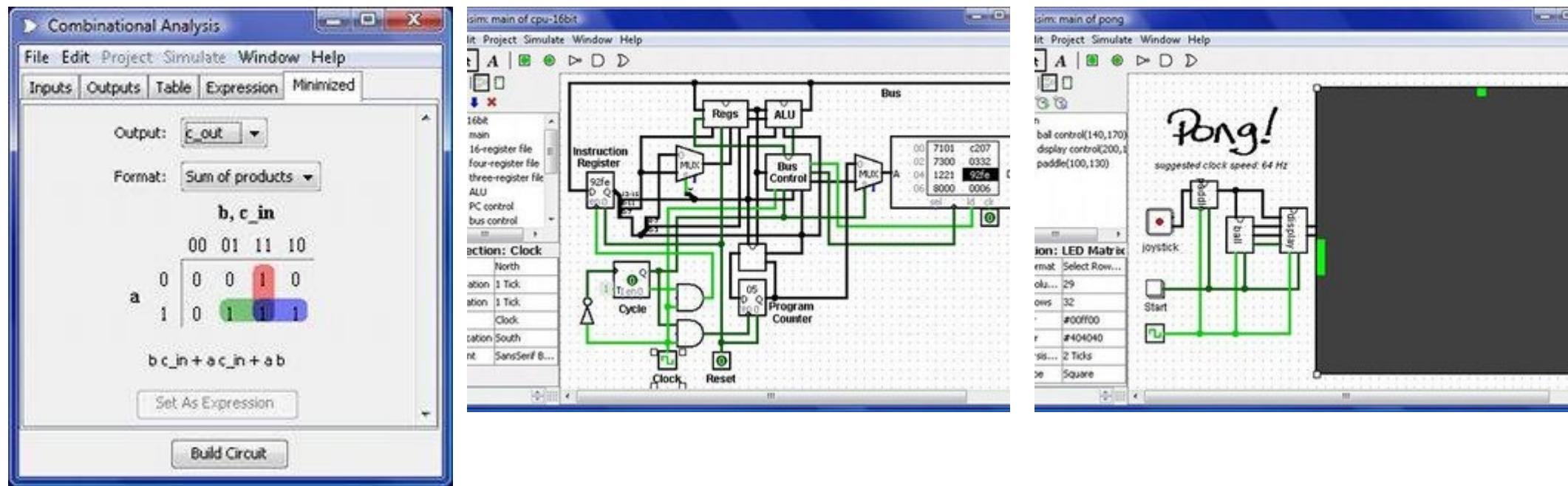




Logisimのインストール

- 各個人のPCにLogisimをインストールする

- フリーウェアであり、Windows/Mac/Linuxに対応しており、Javaで記述されているためRaspberry-Piでも動作する
- <http://sourceforge.net/projects/circuit/>
- 現在最新版は2.7
- <https://sourceforge.net/projects/circuit/files/2.7.x/2.7.1/>





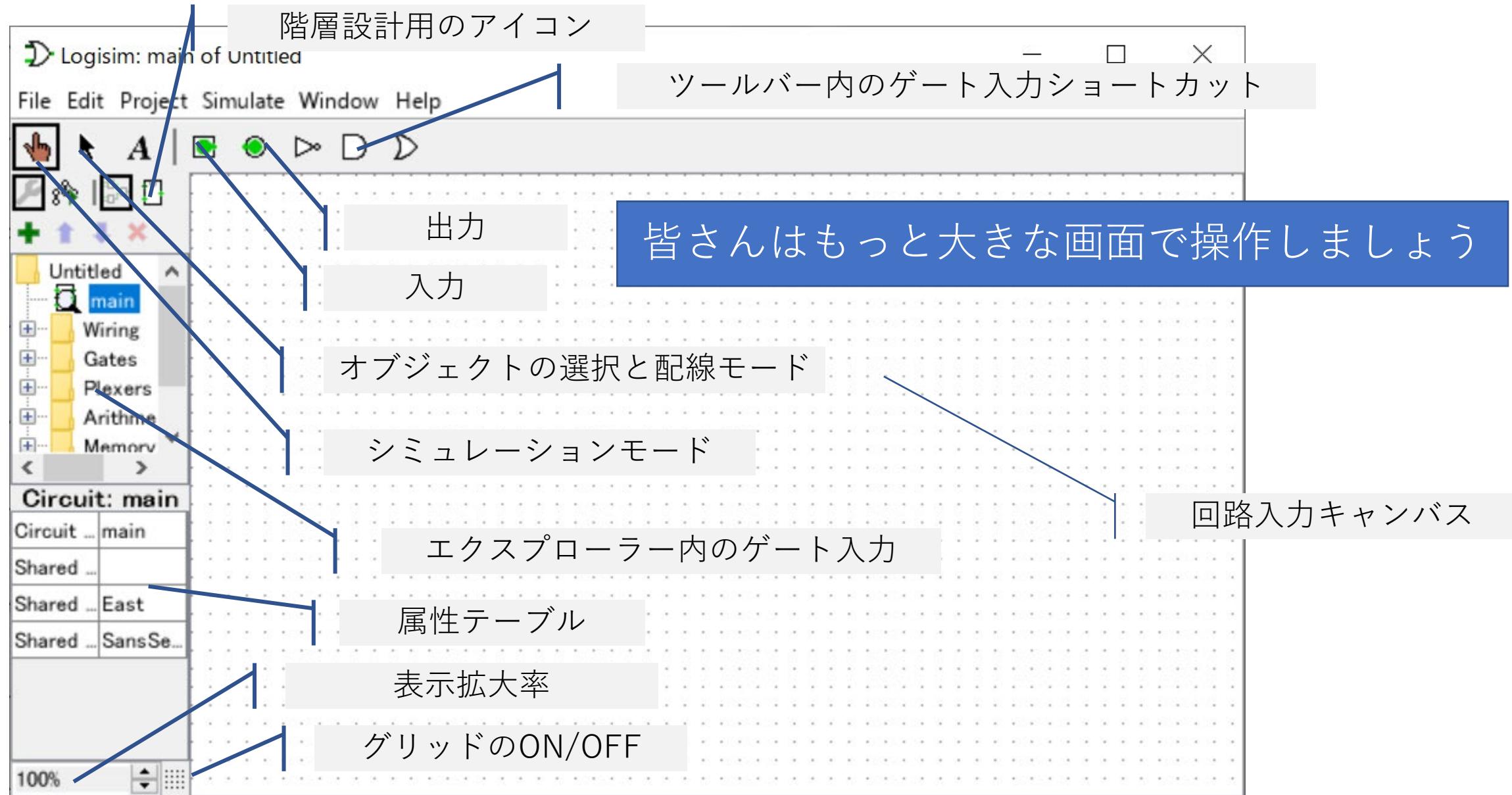
起動のしかた

- Windows, Macとともにダウンロードしたバイナリを動作させるだけ
- Linuxやバイナリが動作しない場合は、Javaの動作環境をインストール
 - `java -jar logisim-generic-2.7.1.jar`
とコマンドラインに入力
Javaのインストールはこちらを参考に
https://java.com/ja/download/help/download_options.xml



メニュー

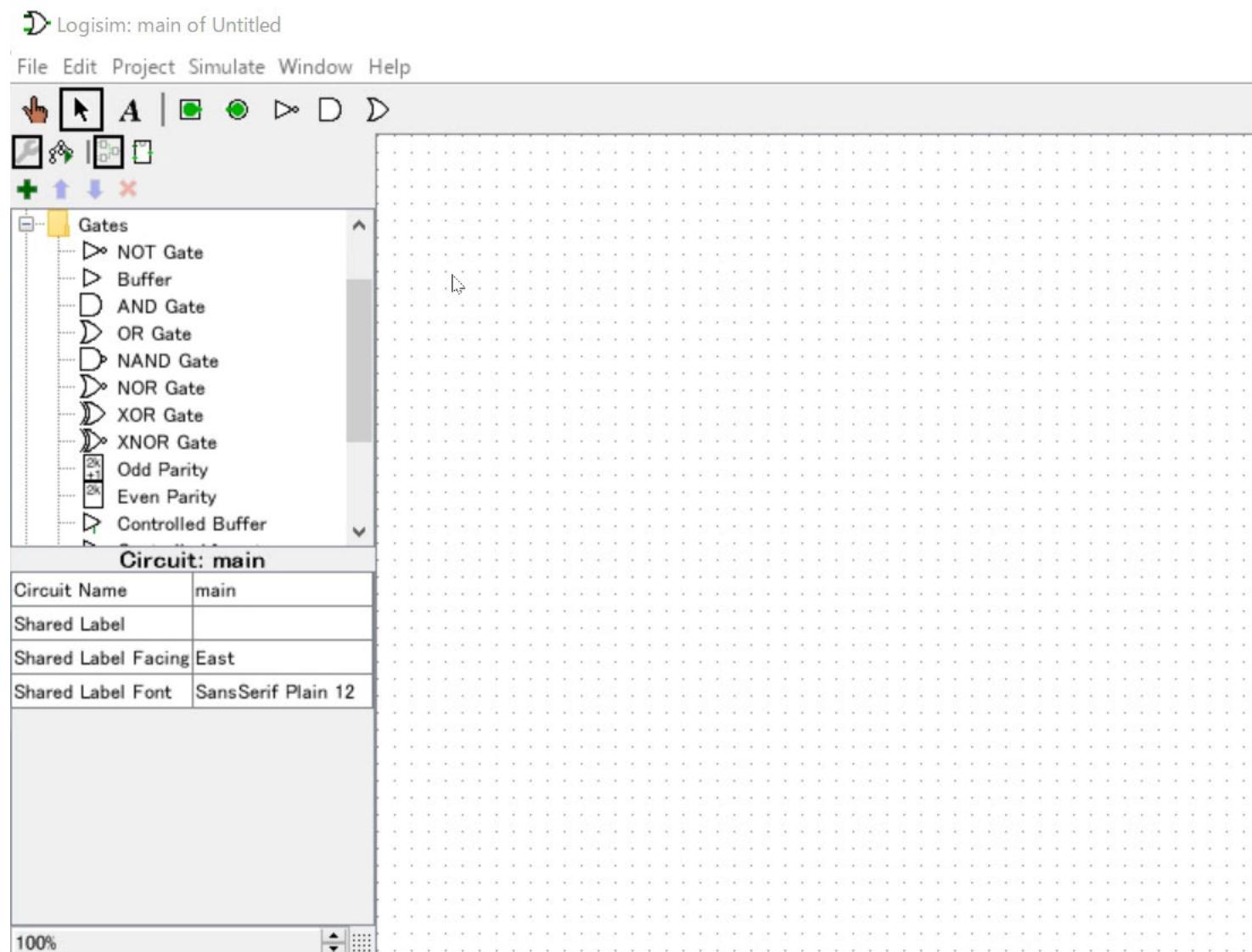
85





回路の置き方

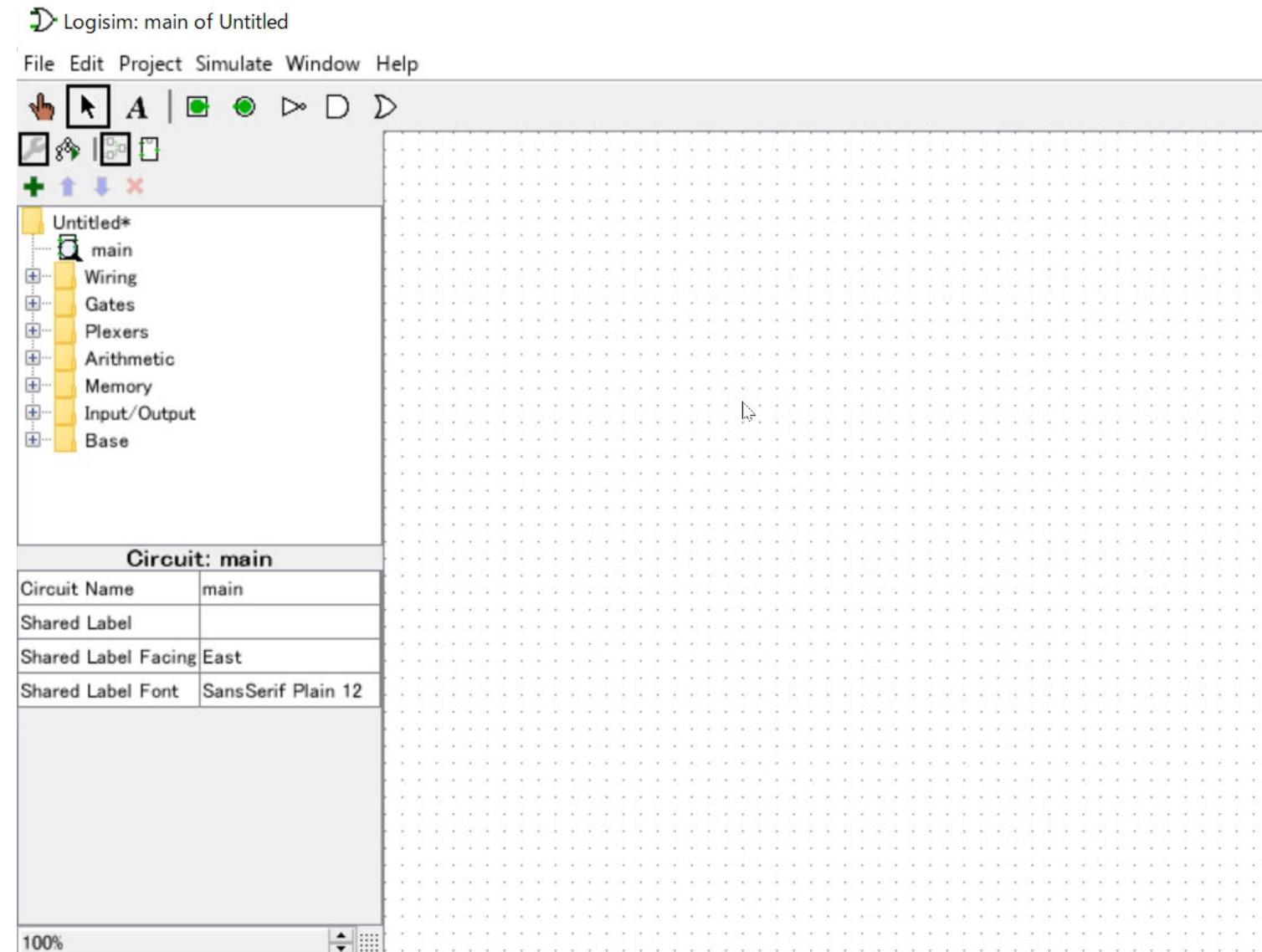
- ・ツールバーにあるゲート入力ショートカットを利用
- ・もしくはエクスプローラのゲートから利用
- ・画面に回路を配置したら
 - ・入力数を選択
 - ・名前を付けて分かりやすく
 - ・Facingで向きを変える
 - ・Negateで入力を反転する





NAND回路の動作を確認する

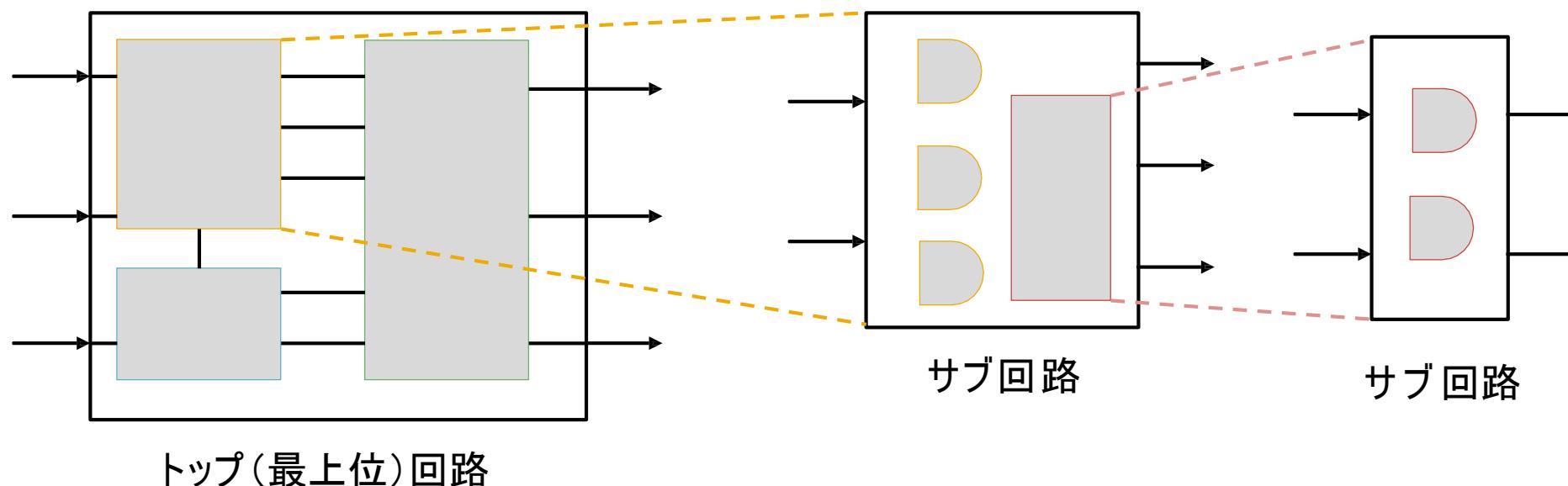
- NAND回路を2入力で配置
- 入力2つ(ina, inb)と、出力1つ(out)を配置
- これらを配線で結ぶ
配線はつながっていれば途中で描き足してもよい
- 指アイコンをクリック
- 入力をクリックすると、0/1が反転する
- これに応じて出力も0/1変化
- NANDの真理値表を確認





階層設計

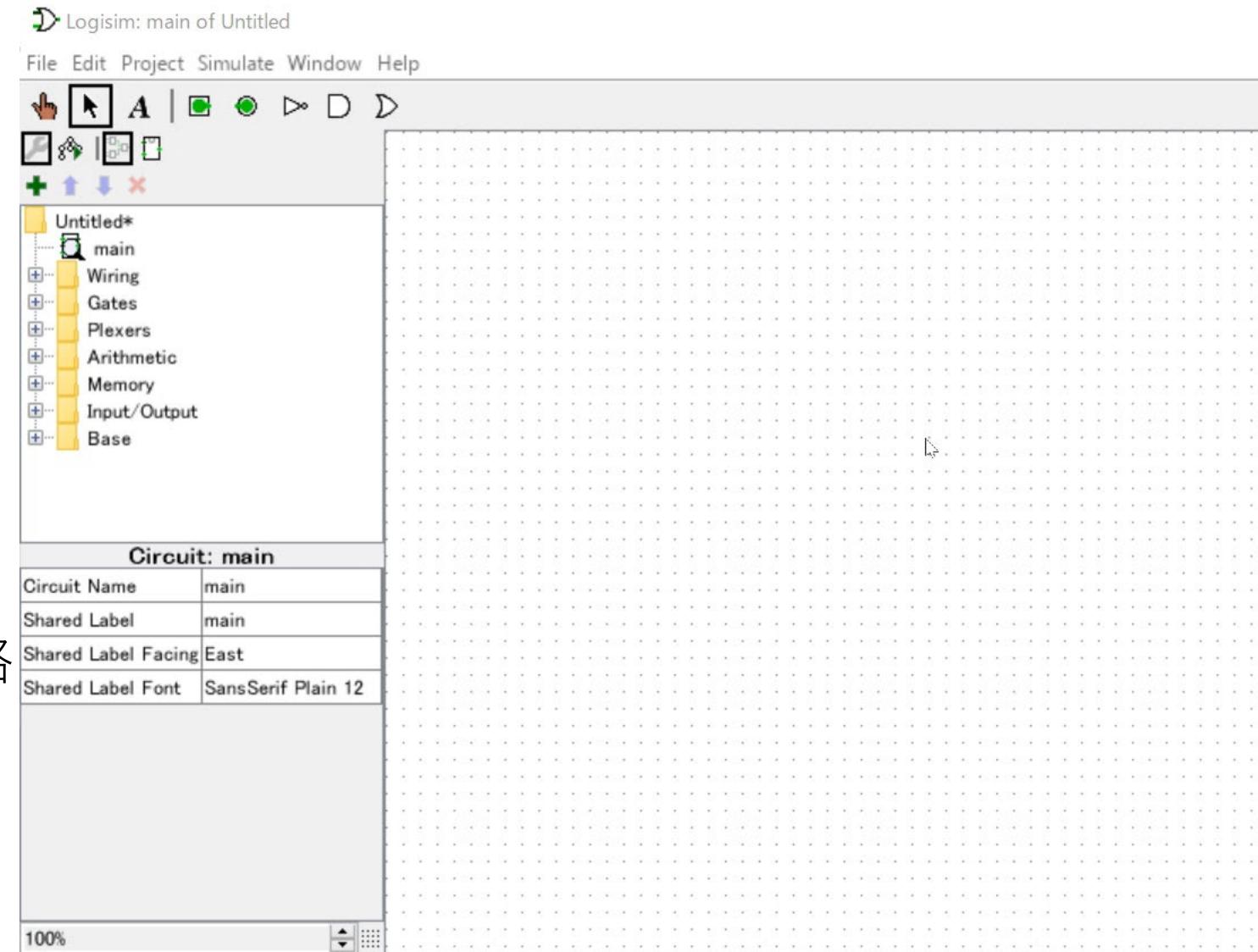
- 回路を1枚の回路図の中に詰め込んで設計すると見通しが悪い
 - 同じ機能の回路を複数利用する場合も面倒
- プログラミングで関数を用いるのと同様、回路図を階層化する
 - 機能ごと、ブロックごとに回路を設計し、それを新しい一つの回路とする





階層設計の例

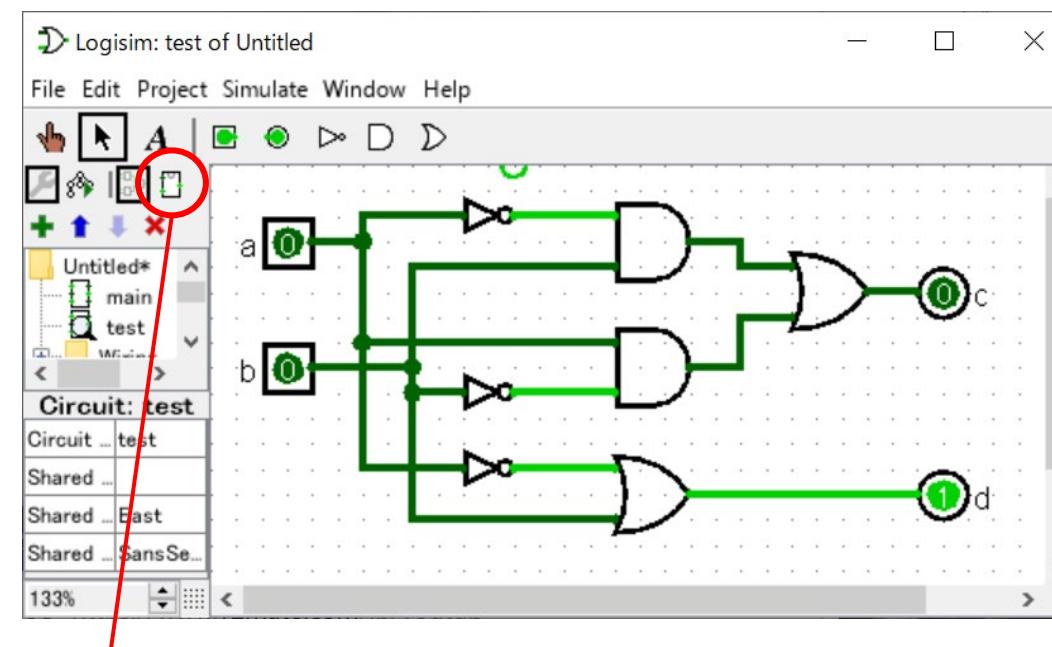
- 最初に+を押して階層を作るとか、回路を入力してからProject->Add Circuitを選択
 - 回路全体の名前を付ける
- 回路にはラベルの付いた入出力が必ず必要
 - 階層の入出力になる
- 別の回路図をダブルクリック
 - 虫眼鏡マークが今見ている回路
- 別の回路を選択し配置できる
 - 何個でも配置できる



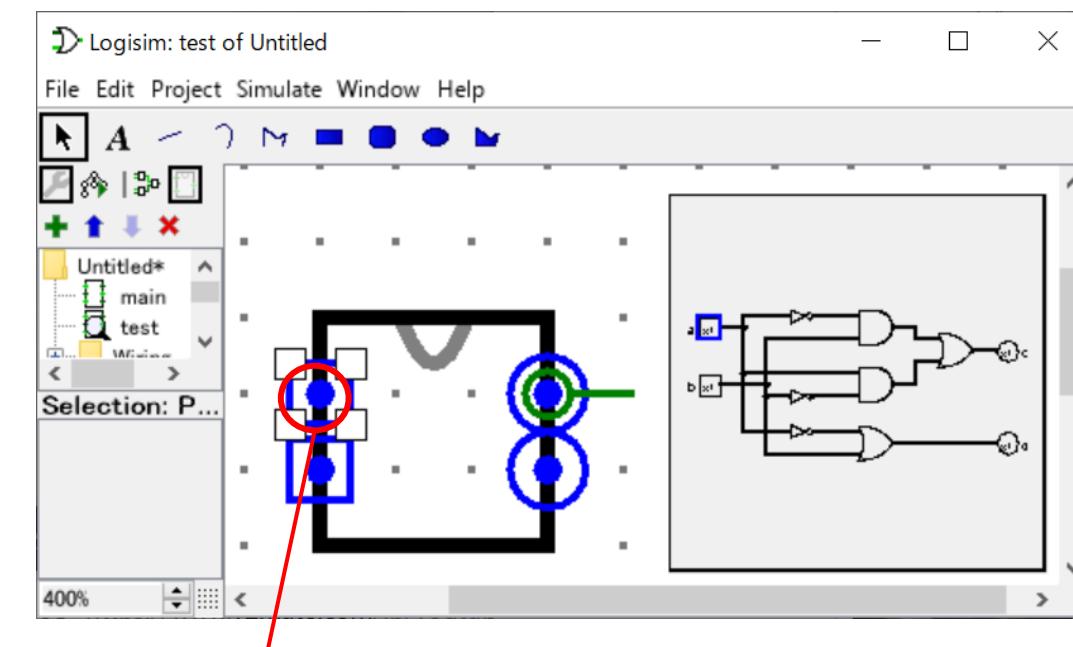
階層設計の例(回路の配置)

90

- ・階層設計が終わったら、必ず配線に名前を付けておきます
 - ・a, b, c, dとつけ、testというモジュールにしています
 - ・名前がないと、上位層で参照したときに何の配線かわからなくなります
 - ・ここで、階層モデルの絵柄の修正や、ピンの場所の修正などができます



ここを押すと、階層モデルの編集ができます
普通は何もしなくて結構です

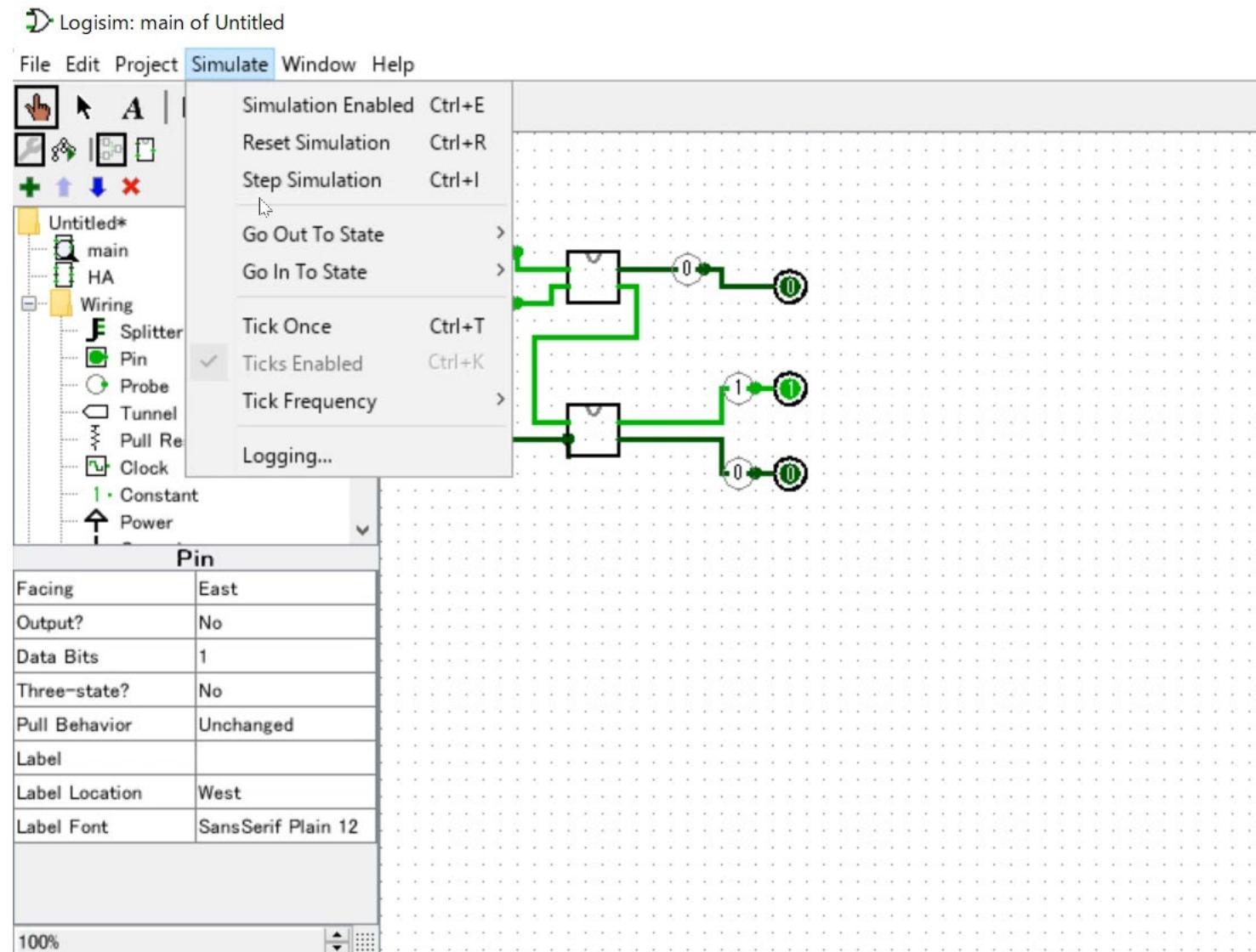


ピンを選ぶと、右に回路が現れて、どの信号線かわかります
ピンは移動できますし、ここでお絵かきもできます



クロックを含む動作の確認

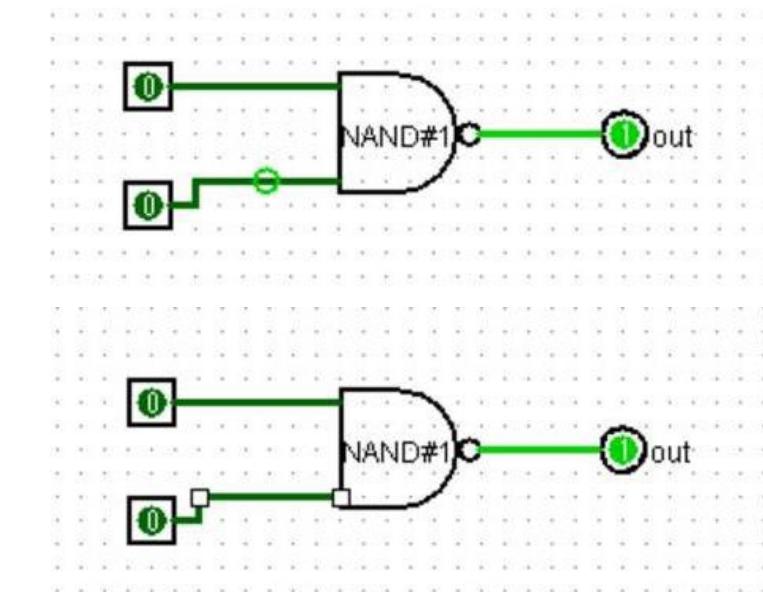
- クロックを配置する
- SimulationタブのStart Simulationを選択すると動作する



設計上の注意

92

- 入力されていない入力は無視される
 - 例えば5入力のANDを配置し、2つだけ使えば、2入力のANDを利用したのと同じ
- WYSIWYGである
 - 回路図の絵として見える動作を行う。
 - 描き方はどうでもよい
- 配線の途中から線を引き出すには○のカーソルから描き始める
- 配線を移動するには配線をクリックしてセグメントを選択し移動する
 - 移動しても切れない（ただし勝手にくっつく）
- ロジックの黒丸●はバス（配線の束）、青丸●は1本の配線を意味する



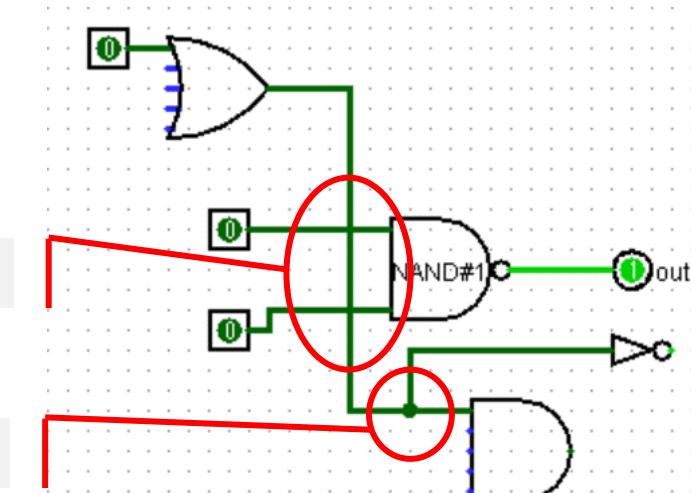
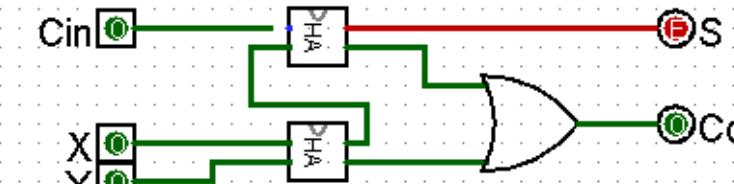
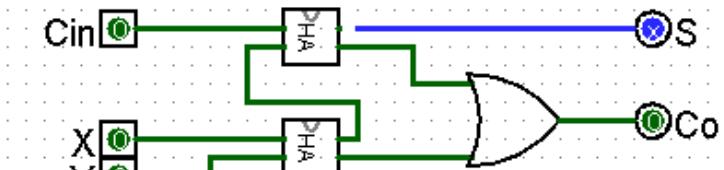


設計上の注意

- 配線の色が**青い**場合は、出力が浮いている
 - つまり接続できていない
- 配線の色が**赤い**場合は、入力が浮いているかショートしている
 - エラーの場合は階層の中も確認すること
- 配線はクロスしてもショートしていない。
●があるとショートしている

ショートしていない

ショートしている

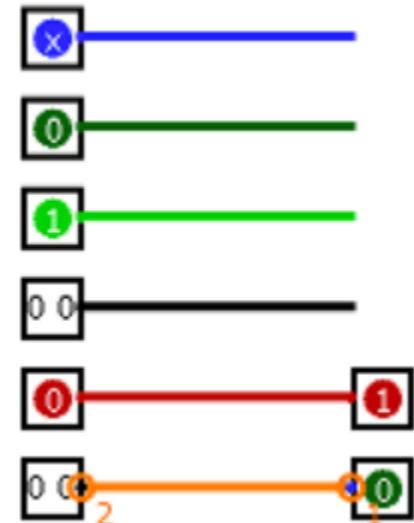




配線の色について(詳細)

- 既に説明済みですが、より詳しくは次の通り

- 灰色：未配線で配線のbit幅が不明
 - 何も繋がっていない配線はbit幅が不明のため灰色となる
- 青色：1bit幅のシングルの配線で、ドライバがつながっていない
 - 誰も配線をドライブしないため不明の値Xとなる
- 暗緑色：配線は正常で1bit幅、0, Low, 0V, GNDの値を持つ
- 明緑色：配線は正常で1bit幅、High, 5V, GNDの値を持つ
- 黒色：配線は正常で、複数bit幅のバスである
 - なお、一部のバスが未配線であることを許容する
- 赤色：エラーです
 - 例えば、出力がぶつかっている状態などにより適切な値が得られないことを意味します
- 橙色：ビット幅が一致していない
 - この場合、数でビット幅が示される





Logisim追加マニュアル

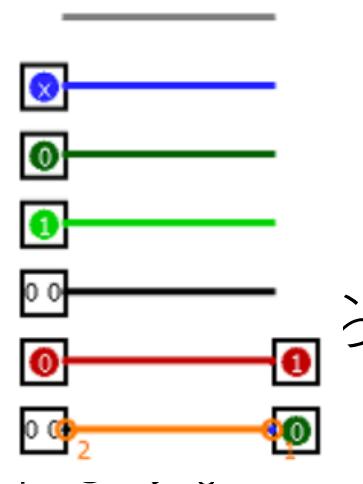
95

- Logisimのオペレーションについて質問を頂いた中で、「ここは知っている方がよいだろう」という内容を集め、まとめて紹介させていただきます。

(1) 配線の色について

- 既に説明済みですが、詳細は次の通りです。

- 灰色：未配線で配線のbit幅が不明です。何も繋がっていない配線はbit幅が不明のため灰色となります
- 青色：1bit幅のシングルの配線で、ドライバがつながっていません。誰も配線をドライブしないため不明の値Xとなります
- 暗緑色：配線は正常で1bit幅、0, Low, 0V, GNDの値を持ちます
- 明緑色：配線は正常で1bit幅、High, 5V, GNDの値を持ちます
- 黒色：配線は正常で、複数bit幅のバスです。なお、一部のバスが未許容とを許容します
- 赤色：エラーです。例えば、出力がぶつかっている状態などにより遅れることを意味します
- 橙色：ビット幅が一致していません。この場合、数でビット幅が示さ





ライブラリ(1)

- 次のようなライブラリがある
 - スプリッタ  **Splitter** バスから配線を引き出したり、逆に配線を束ねてバスにするときに使います
 - Fan Out(ファンアウト)で分岐数、Bit Width Inでそのうちの有効数、また、Bit Width Inで指定した数のそれぞれのbit幅を入力します
 - ピン  **Pin** 入出力のピンを指定します。□が入力、○が出力です。
 - スリーステート(0, 1, Z表現、Zはハイインピーダンス)やプルアップ・ダウン(抵抗を介してVCCやGNDに接続)などが指定ができます
 - プローブ  **Probe** 配線の持つ値や状態を表示します。基数を変更でき、デバッグに利用できます
 - トンネル  **Tunnel** 同じ名前のトンネル同士は配線が描かれていなくても繋がっていると判断されます
 - 効果的に使うと回路図が見やすくなります
 - プル・レジスタ  **Pull Resistor** 配線が誰もドライブしていない場合、インピーダンスにならず、プルアップならば1、プルダウンならば0にドライブされます。一般的なプルアップ、プルダウンと同じです
 - クロック出力  **Clock** します。周波数はSimualte->Tick Frequencyで、また、Tick Enabledでクロックを投入しますTick Onceでクロックを一回だけ入れることができ、クロックステップ毎に回路の動作を確認できます
 - コンスタント  **Constant** 定数値を生成します
 - パワー・グラウンド  **Power/Ground** 配線を0か1にします
 - トランジスタ  **Transistor** MOS-FETではないため、極性があり、>の方向に電流が流れます。その他はMOSと同じです
 - トライステートゲート  **Transmission Gate** スイッチと同じで、0/1を通すか、切り離すか(ハイインピーダンス)を選べます
 - ビット拡張  **Bit Extender** 異なるbit幅のバスを繋げるときの未配線部分の処理を指定できます。
- ゲートは特に説明不要であろう





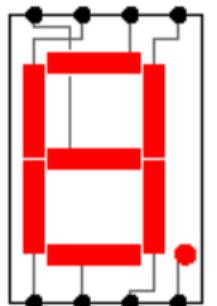
ライブラリ(2)

- コントロールドバッファ (シリーズステートバッファ)  Controlled Buffer/Inverter 本来、トライステートゲートは双方方向であるが、このシミュレータでは向きがあるため上記トライステートゲートと同じに動作になる
- マルチプレクサ  Multiplexer 複数の配線から一つの配線を選択します。本数やバス幅を選択できる
- デマルチプレクサ  Demultiplexer 逆に1つの配線を複数の配線のどれかに選択して接続する
- デコーダ  Decoder 選択した配線を1にします。残りの配線は0かZ(ハイインピーダンス)にできる
- プライオリティエンコーダ  Priority Encoder 一般にいう、固定優先順位アービタとして機能する
 - 複数の入力のうち、1つだけHighの場合はその番号を、複数Highの場合は優先順位が高い、つまり大きな番号が選ばれる
- データセレクタ  Bit Selector 任意の組み合わせの、配線の束をバスから作り出す。バスはビットマップで指定する
- 演算器  Adder  Multiplier 演算を行う。割り算は直接不可
 Subtractor  Divider
- ネゲータとコンパレータ  Negator bitの反転および、バス同士の大小等号比較を行う
 Comparator
- シフタ  Shifter バスの中身を右や左に指定bit幅だけ移動させる
- ビットアダー  Bit Adder 入力に含まれる1の数を数えて出力する
- ビットファインダー  Bit Finder バスの内容から、最上位もしくは最下位から順に探して最初に0もしくは1が現れる場所を求める
- フリップフロップはANDやOR同様、流石にここでは扱わない
- レジスタ  Register 複数のD-FFを束ねてバスをラッチ(クロック立ち上がりで入力値を固定出力)する。
1bitならば、D-FFとほぼ同じ
- カウンタ  Counter 好きな大きさのアップ・ダウンカウンタが構成できる
- シフトレジスタ  Shift Register 指定個数FFが並んでつながった構造をもち、クロックが入るごとにbitがずれて記憶される



ライブラリ(3)

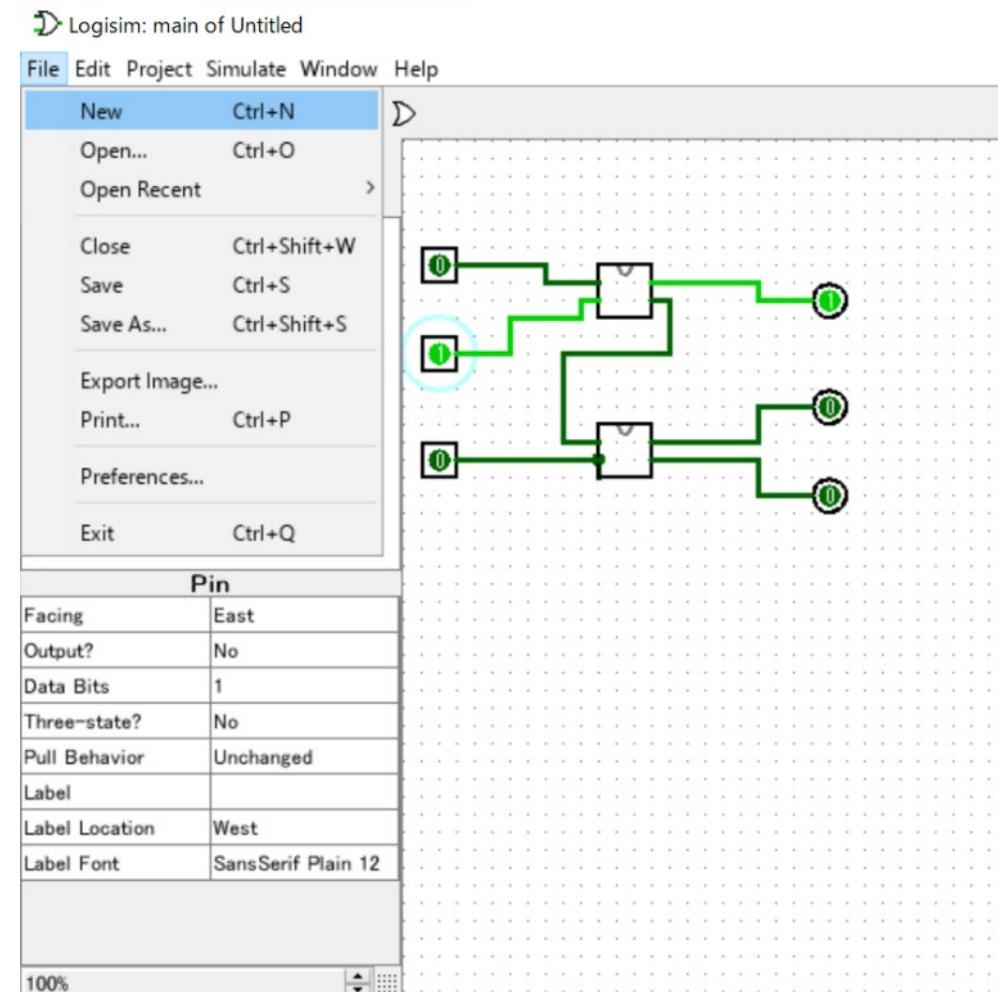
- 亂数生成器  Random 亂数を生成する
- RAMとROMでメモリを構成する。中身は専用のエディタで記述する
- ボタン  Button ONで1をOFFで0を出力する理想的なボタンでチャタリングなし
- ジョイスティック  Joystick デジタルですが多値がとれるためアナログ的に動作する
- キーボード  Keyboard キーボード入力により7bitアスキーワードを受け付ける
- LED  LED 正論理負論理の切り替えや発色の変更が可能。ただし3色LEDはない
- 7セグメントディスプレイ(ナナセグ)  7-Segment Display 一般的な数字表現デバイス
 - 右のような配線となっている
- 16進数値ディスプレイ  Hex Digit Display 4bitの入力を0-9, A-Fで表記
- LEDマトリクス  LED Matrix 縦横のLEDの数や色の変更ができる
- キャラクタディスプレイ  Character Display ターミナル画面
 - アスキーコードとトリガ線で制御する。サイズや色が変更できる
- その他についてはマニュアルを参照のこと





回路図の保存と読み込み

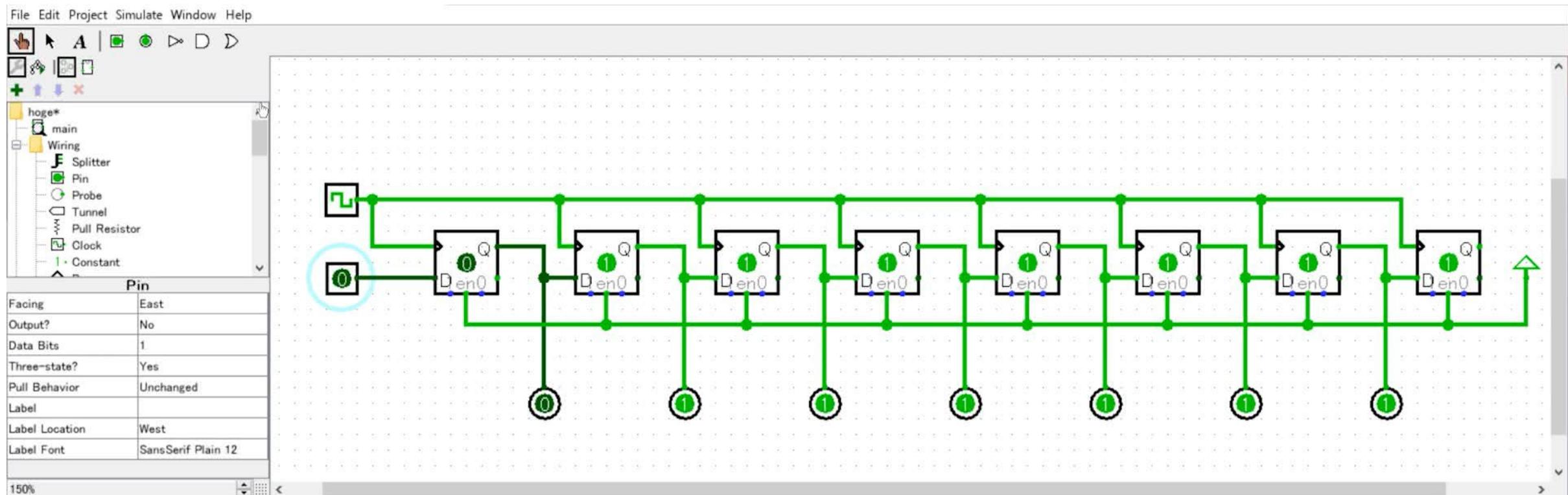
- 保存
 - FileタブのSaveやSave asを利用
- 読み込み
 - FileタブのOpenを利用
- こまめに保存してトラブルに備えよう！



回路シミュレータでシフトレジスタを作る

100

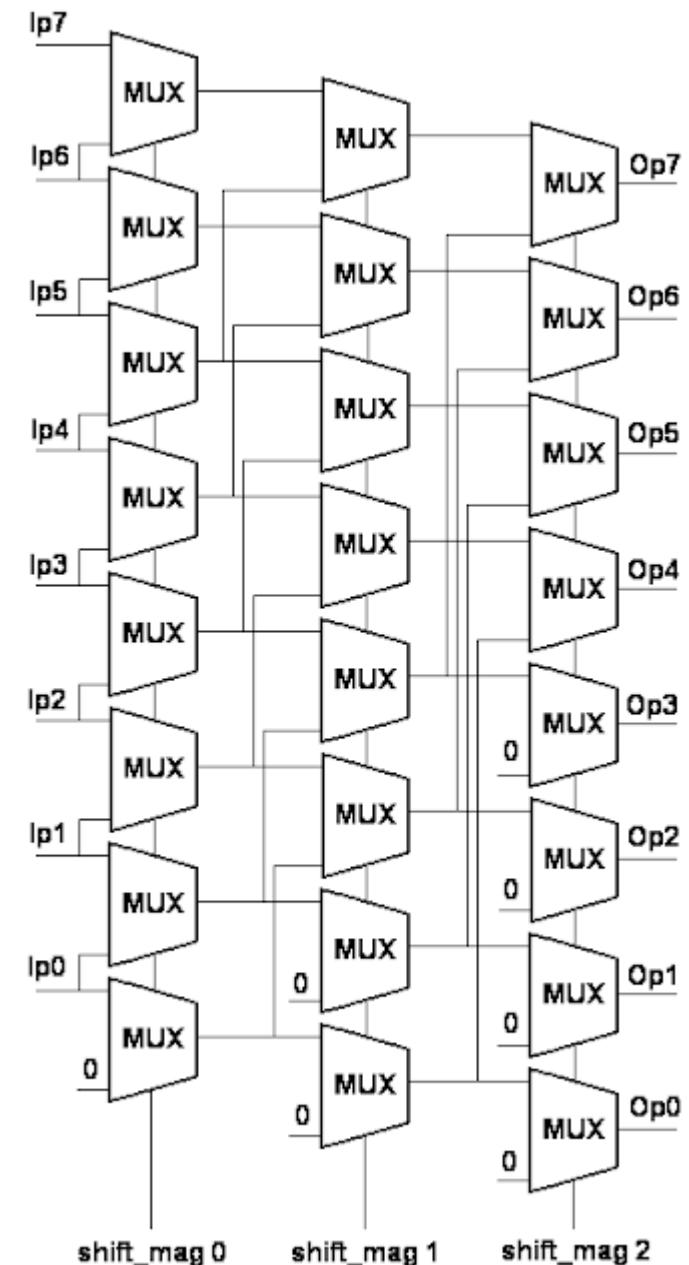
- 実際に動作させてみると、シリアルの入力が伝搬して伝わり、パラレル出力されていることがわかる



ではバレルシフタ

- 入力に対して
 - 1ビットずらすshift_mag0
 - 2ビットずらすshift_mag1
 - 4ビットずらすshift_mag2
 - これらを組み合わせて任意の7bitまでずらす
- 例えば、5ずらす場合、
 - $5 = 3'b101$
 - つまり、shift_mag0とshift_mag2がHIGHとなり、1ビットと4ビットのずらしが同時に起きるため、結果的に5ビットずれる
- この回路も覚える必要はない
verilogのシフタはバレルシフタを構成する

```
logic [7:0] d;
logic [2:0] s;
assign o = d<<s;
```



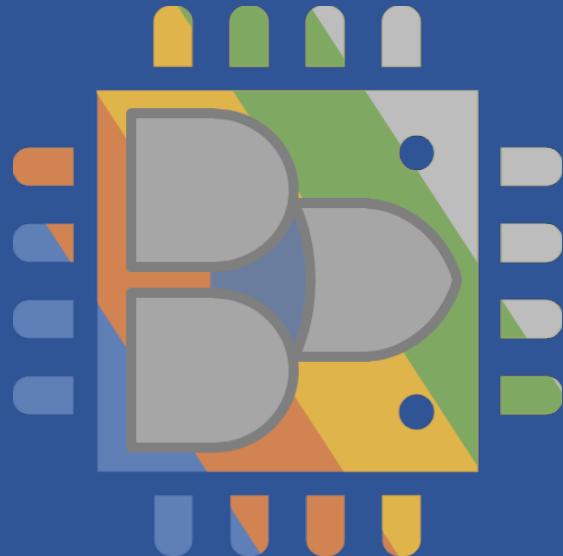


演習問題（4）

102

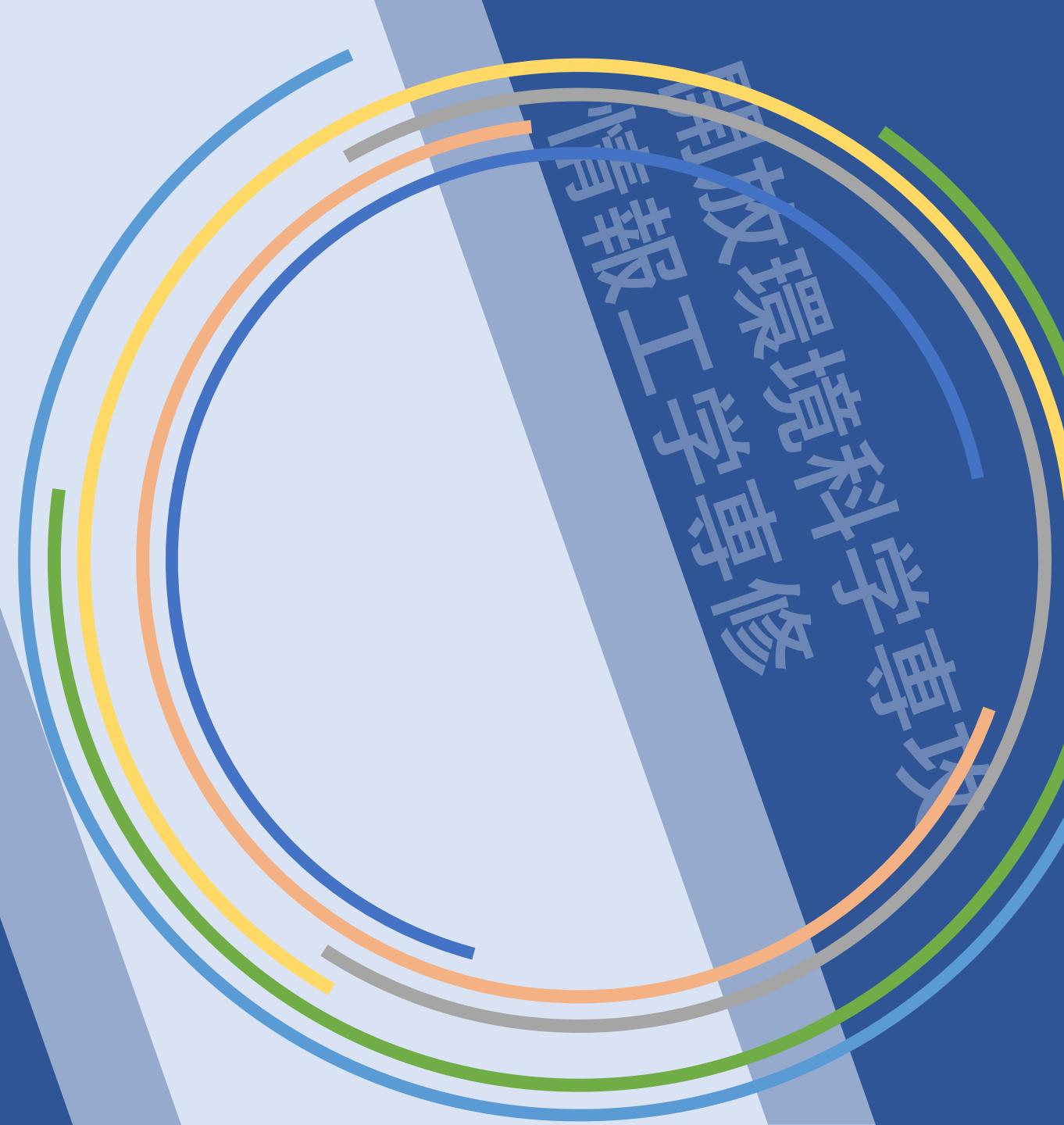
- 次の仕様を持つ8bitのプリセッタブルダウンカウンタを記述しなさい
 - enが1のときカウンタが動作する
 - enが0のとき8bit入力vの値でカウンタが初期化される
 - カウンタの値が0になると、3クロック分出力bが1になる(beep出力)
- 注意事項
 - レポートをMicrosoft Wordファイルで作成、LMSへ提出すること
 - A4 2枚以内で作成し、最初にタイトルを「演習問題（3）」として記載し、名前と学籍番号も忘れずに記載すること
 - まずverilogソースを添付、シミュレーションを行い、動作している証拠としてgtkwaveの画面をキャプチャしてWordに張り付けなさい
 - 設計で利用した配線は全てgtkwave上で表示しておくこと(clk, rstなどすべて)
 - verilogソースコードをテキスト情報として貼り付けること（画像ではない）
 - 提出締め切りはLMSを確認すること





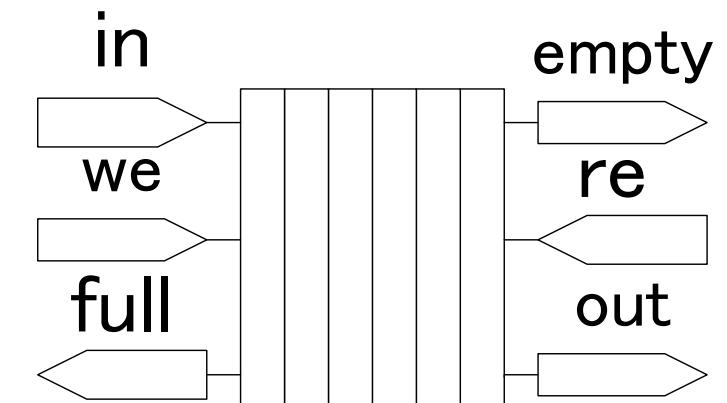
計算機システム 設計論(5) State Machine

担当： 西 宏章





- FIFOの用途
 - 流れを扱う設計
 - 異なるタイミングで動作を行うモジュールの結合
- さまざまな場面で必要となる基本論理構造
 - 2つの異なる回路があり、どちらも10クロックに5回情報を処理して次に伝える
 - ひとつは5クロック作業して5クロック休む
 - もうひとつは1クロック作業して1クロック休む
 - この場合、直接つないでも正しく動作しない
 - FIFOを用いてバッファを構成し、伝える必要がある
 - 実際のデジタルシステムではFIFOが大量に利用
- 構成イメージ
 - 入力のin、出力のout、入力に対してFIFOが一杯であることを意味するfull、出力に対してデータが空であることを意味するempty、書き込みを行うwe、読み出しを行うre

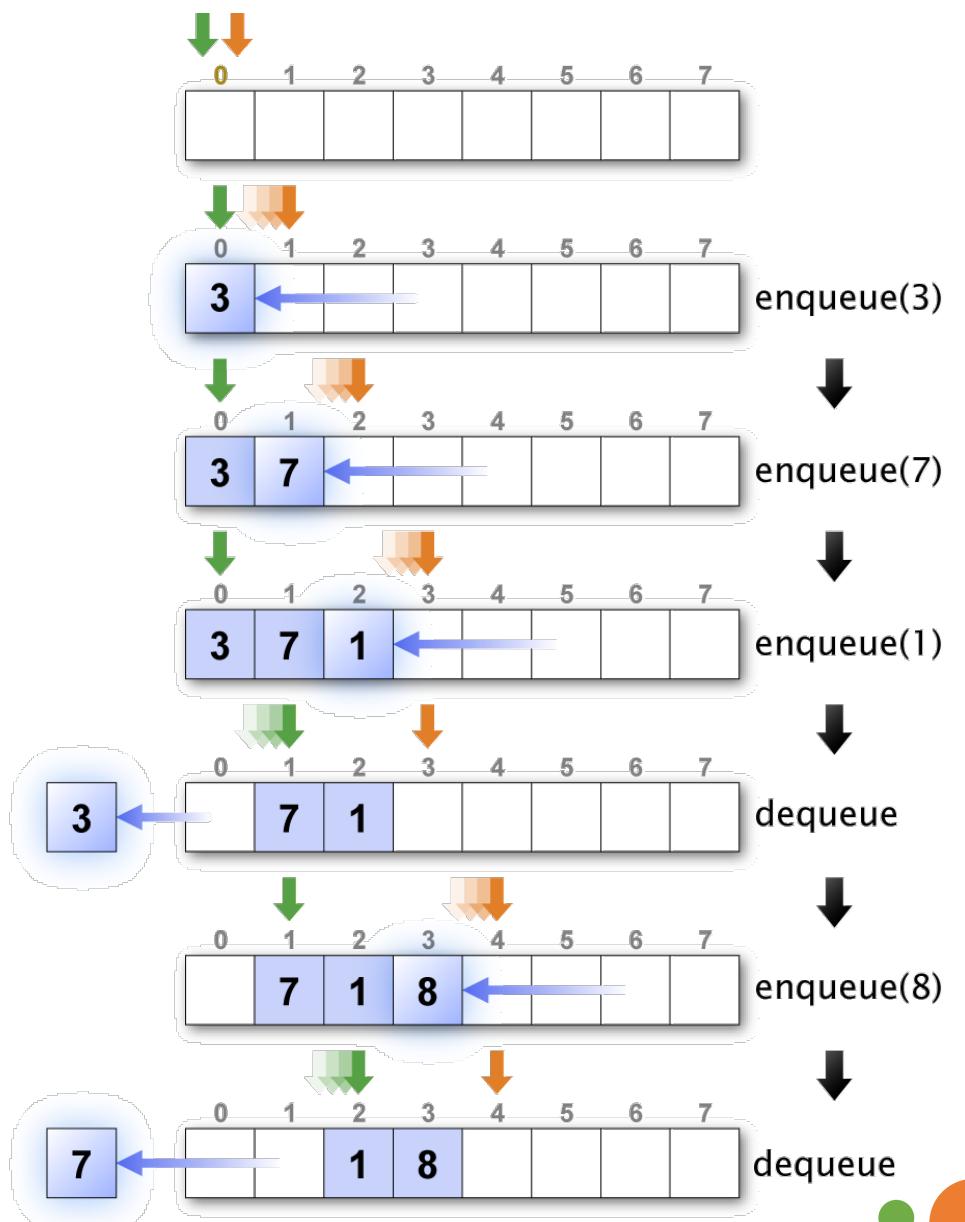
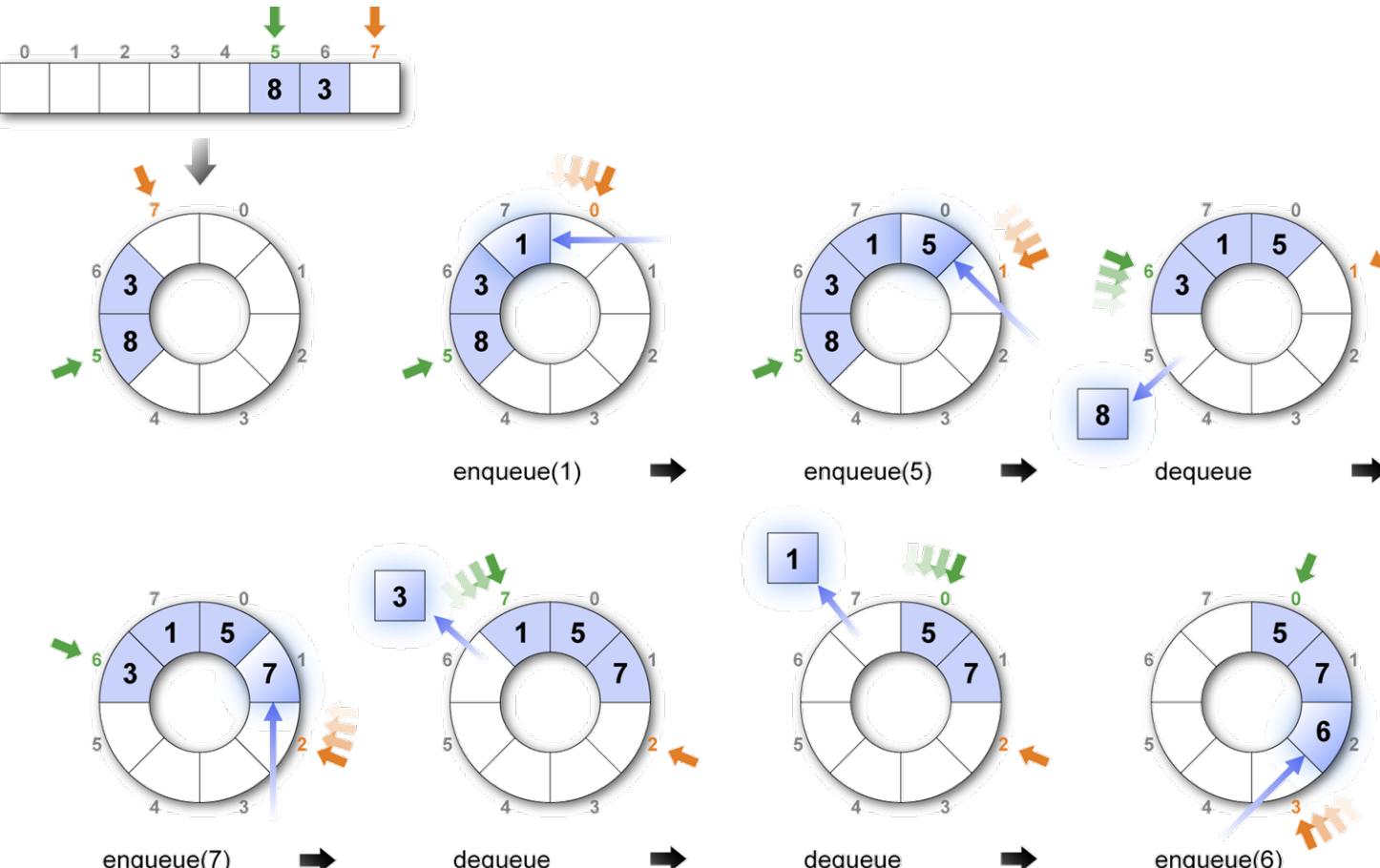




サイクリックバッファ(リングバッファ)

105

・動作内容





FIFOの動作と構成

106

- 書き込み側

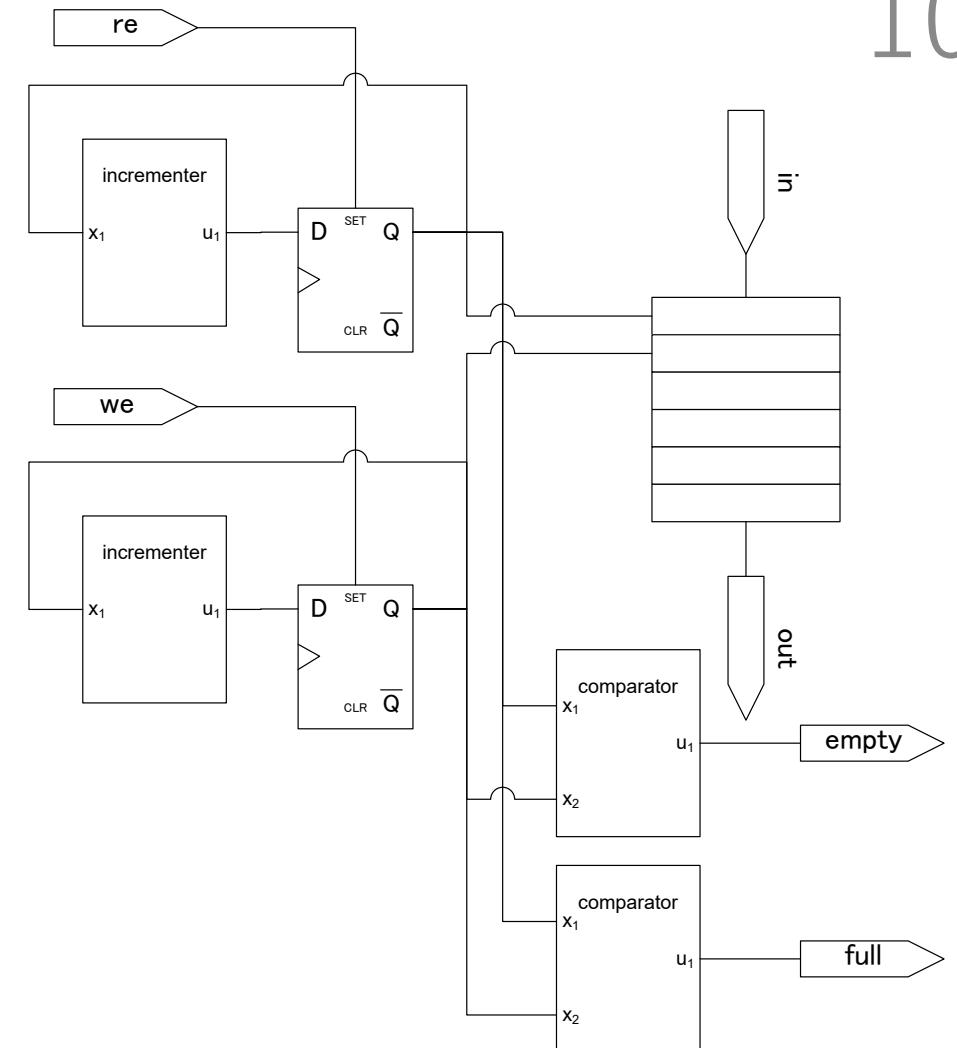
- fullがhighの場合、FIFOが一杯であるため書き込み不可でfullがlowになるまで待つ
- fullがhighでweをhighとするのはルール違反であるが、そのままwraparoundを許す場合と、書けないようにブロックする場合がある

- 読み出し側

- emptyがhighの場合、FIFOが空であるため読み出し不可でemptyがlowまで待つ
- emptyがhighでreをhighとするのはルール違反であるが、同様に2つのケースがある

- 動作

- 読み出しそうと書き込み用の2つのカウンタがあり、読み出しあくまでも書き込みが行われるたびに1増える。カウンタの値が一緒であればempty、書き込みアドレス+1と読み出しあドレスが等しければfull





FIFOの記述

```
module fifo(in, we, full, out, re, empty, clk, rst);
    input [7:0] in;
    input we;
    output full;
    output [7:0] out;
    input re;
    output empty;
    input clk, rst;

    logic [7:0] mem[15:0];
    logic [3:0] head, tail, headi; //ここに注意
    logic empty, full;
    always @(posedge clk) begin
        if(rst) begin
            head <= 0;
            tail <= 0;
        end else begin
            if(we) head <= head + 1;
            if(re) tail <= tail + 1;
        end
    end
end
```

```
always @ (posedge clk)
    if(we) mem[head] <= in;
    assign out = mem[tail]; // データは常に出ている
always_comb begin
    if(head == tail) empty = 1'b1;
    else empty = 1'b0;
    headi = head+1; //ここに注意
    if(tail == headi) full = 1'b1; //ここに注意
    else full = 1'b0;
end
endmodule
```

- **if(tail == head+1) full = 1'b1;**
や
if(head+1 == tail) full = 1'b1;
と記述すると間違いであり、正しく動作しない場合がある
 - head+1はbit幅が拡張され1つ増えており、比較しても一致しない
 - $7+1 == 0$ であるべきが、そうはならなくなる
- 比較を行う際には、ビット幅に気を付けること**



動作内容の続き

108

- headとtailはC言語でいう2のポインタであり、FIFOメモリであるmemのどこかを指す。headはmemの書き込み場所を、tailはmemの読み出し場所を表す。動作はheadをtailが追いかけるイメージである。
- 非同期式FIFO(入力と出力で異なるクロックドメインのクロックで動作するFIFO)は機器間接続ではしばしば必要となるが、これを設計するのはかなり高度な内容を含み、正しく動作する非同期式FIFOを論理レベルだけで設計することは困難
 - 二つのクロックで動作するFFが存在しないため難しい
 - 非同期式FIFOの構成法として遅延書き込み型, bit拡張型, 反射型などがある
- FPGAの利用では、FIFOやメモリ記述はライブラリが準備されており、これを明確に指定して利用すること。実装効率がよい

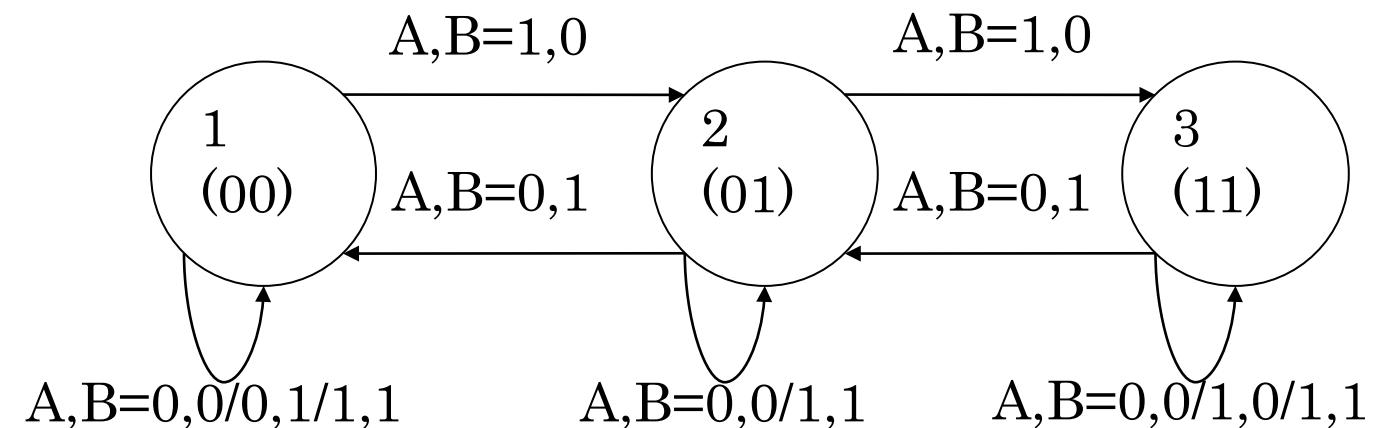




ステートマシン

109

- 順序回路とも呼ばれシーケンスを記述する上で重要な構成要素
 - ソフトウェアによるプログラム同様、まずA、次にB、もし○○であればC、そうでなければAに戻るといった、アルゴリズムの順序を記述できる
 - パラレルではなく、シリアルを実現
- ステートマシンは、次のように記述する
 - 最初に示したアップダウンカウンタを例にとる





ステートマシンの実装例

110

- 2つの部分から構成される
 - 現在の状態を管理するFF
 - 内部状態 (state) を保存
 - 計算された次の状態に遷移し続ける
 - 次の状態を計算する組み合わせ回路
 - Combination logic
 - 現在の状態(state)と入力a、bから、次の状態(nextstate)を計算する
 - case文の合成はこだわりが必要
 - 2'b10は利用していないが、合成時don't careとして扱えるか？扱えば回路が小さくなる
 - このためのfull_case
 - 2'b00, 2'b01, 2'b11の状態を同時に判断できるのか？出来ればpriority encoder回路が不要
 - このためのparallel_case

```
logic [1:0] state, nextstate // 3状態なので2bit
always @(posedge clk) begin
    if(rst) state <= 2'b00;
    state <= nextstate;
end
always_comb begin
    nextstate = state;
    if(({a,b} == 2'b00) || ({a,b} == 2'b11))
        nextstate = state;
    else
        case(state)
            // synopsys full_case parallel_case
            2'b00: if(a) nextstate = 2'b01;
            2'b01:begin
                if(a) nextstate = 2'b11;
                if(b) nextstate = 2'b00;
            end
            2'b11: if(b) nextstate = 2'b01;
        endcase
    end
end
```





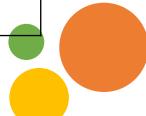
ちょっとまってコメントに意味があるの？

111

- ・業界標準CADベンダー Synopsysがやむなく取り入れたルール
 - ・合成のとき困るけど、言語仕様は変えられないためやむなくコメントへ
 - ・コメントがコメントではない！でも、業界標準になってしまった
- ・SystemVerilogはこの状況に終止符を打った
 - ・priorityとuniqueを文法に追加（ただし、直観的な対応が微妙でiv未サポート）

full_case parallel_case	SystemVerilog
case()	case()
case() // full_case	priority case()
case() // parallel_case	unique case() … default
case() // full_case parallel_case	unique case()

書き方	意味
priority if	if-else_ifのどこか、少なくとも1つがマッチすることを保証し、何もマッチしないとエラーになる。else文があると無意味
unique if	if-else_ifが複数にマッチしないことを保証、するとエラーになる





Verilog Attribute (Verilog 2001より)

112

- Verilog attributeによる解決法もある
(* 属性値指定 *) と記述する/* */はコメント
- 合成における最適化を行わない
 - (* don't_touch = “{true|false}” *)
- FULL_CASE
 - (* full_case *) (* full_case = “{true|false}” *)
- PARALLEL_CASE
 - (* parallel_case *) (* parallel_case = “{true|false}” *)
- OPTIMIZE
 - (* optimize *) 最適化の制御
 - (* state_variable *) ステートマシンの状態値に対する最適化
 - (* ripple_adder *) リップルキャリーの指定
 - (* mealy_fsm *) ステートマシンがミーリーの機械であることの指定
 - (* clock_line *) クロック線であることの指定
- 他にも様々な拡張が実装されている

```
(* full_case=1, parallel_case=0 *)
case (SELECT)
  3'b001 : Q1 <= A[0];
  3'b011 : Q1 <= A[1];
  3'b110 : Q1 <= A[2];
endcase
```

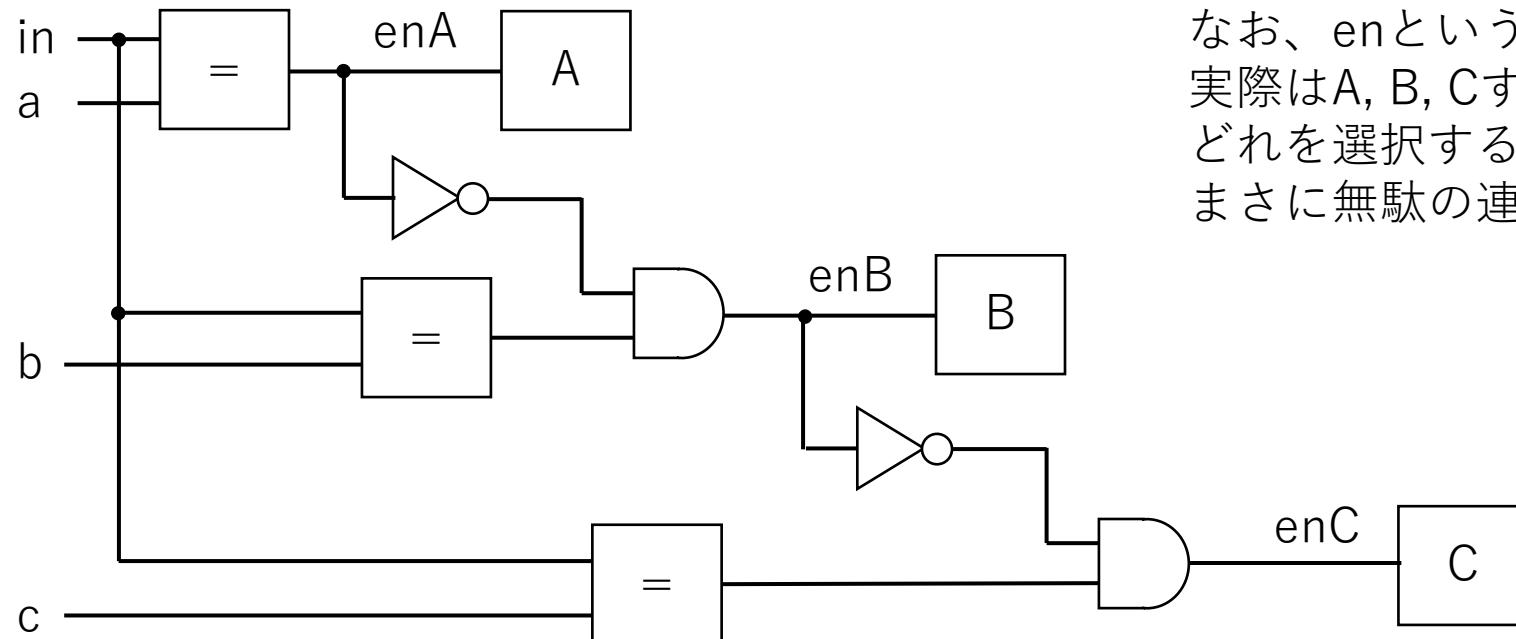




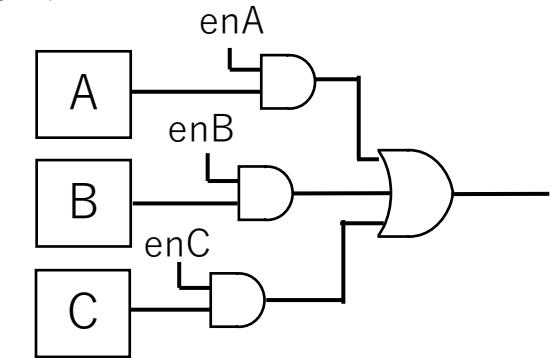
プライオリティエンコーダ

113

- 各ifは、それ以前のifが成立していないという条件のもと判断されるが、これらが全て比較器とセレクタとなり、遅延が伝搬する
 - 遅く大きくなる
- これを解決するのが、caseとpriority, unique (iv未サポート)
 - ifはプライオリティエンコーダを用いて順に調べる構文
 - caseは書き方で並列であったり不完全な形でも調べられる構文



なお、enというenabler信号で記載しているが、
実際はA, B, Cすべての処理が行われたうえで、
どれを選択するかというセレクタが動作して一つ選ばれる
まさに無駄の連続がハードウェア





ステートを名称で表記する

114

- enumを使うと状態を名称で表記できる
 - enum logic[1:0] {READ, WRITE, SLEEP, NOP} cmode;と定義すると、
 - cmd = NOP;
 - if() cmd = READ;などと記述できる
- 最適化が気になる場合は、さらに次のように直接割り当てる数を指定
 - enum logic[1:0] {READ = 'b10, WRITE = 'b01, SLEEP = 'b11, NOP = 'b00}

```
enum logic[2:0] {READY, WAIT, EXEC} state;  
always@(posedge clk or posedge rst)  
  if(rst) state <= READY;  
  else state <= nstate;  
end
```

```
always_comb  
  unique case (state)  
    READY: if(run) state = WAIT;  
    WAIT: if(start) state = EXEC;  
    EXEC: if(stop) state = WAIT;  
      else if (done) state = READY;  
  endcase  
end
```





One Hot State Machine

115

- ステートマシンは遅い
 - ステートのFFの値からある状態を得るためにデコーダが必要
 - 3'b110をstate READYとすると、この状態には1 and 1 and 0を確認するロジックが必要
 - このデコーダをなくして高速化
- 各ステートを表現するFFをそれぞれ1つずつ準備する
 - 10状態では通常4個のFFでよいが、10個のFFを準備
 - 必ずどこか1ビットだけHIGHとなるように状態を決定する
 - ここでは001, 010, 100で1か所のみ
 - caseの書き方を工夫してOne Hot化する

```
module test(input logic a, b, clk);
    typedef enum logic[2:0] {ONE = 3'b001, TWO = 3'b010,
    THREE = 3'b100} myst;
    myst state, nstate;
    always @(posedge clk)
        if(rst) state <= ONE;
        else state <= nstate;

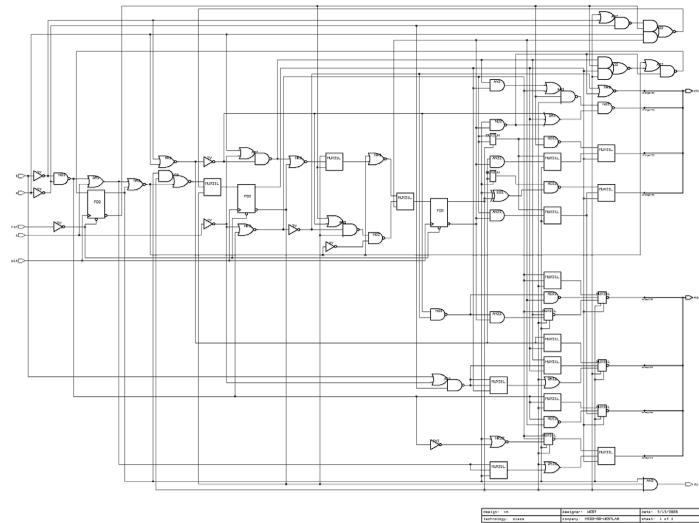
    always_comb
        unique case(1)
            state[0]:
                unique case({a,b})
                    2'b00: nstate = TWO;
                    2'b01: nstate = ONE;
                    2'b10: nstate = THREE;
                    2'b11: nstate = TWO;
                endcase
            state[2]:
                unique case({a,b})
                    2'b00: nstate = THREE;
                    2'b01: nstate = TWO;
                    2'b10: nstate = THREE;
                    2'b11: nstate = THREE;
                endcase
            endcase
        endmodule
```



演習問題（4）

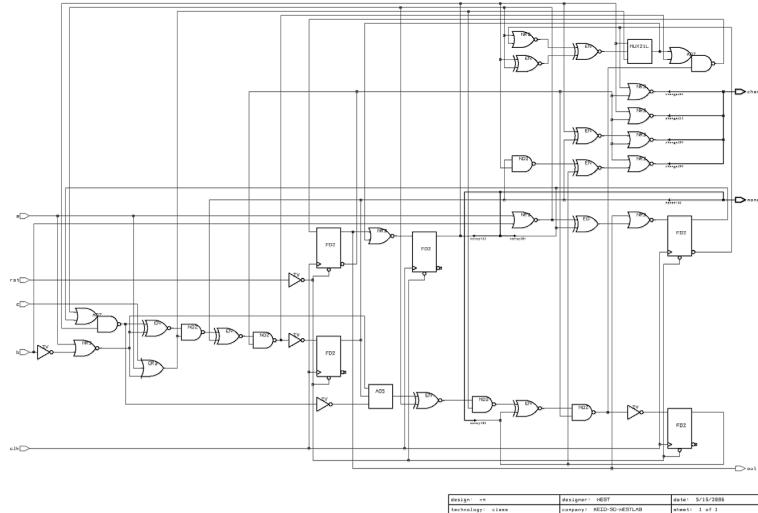
116

- ステートマシンの演習として以下の回路を設計しなさい
 - 50円の缶ジュースを購入する自動販売機のシーケンスを作成せよ
 - 硬貨は、10円、50円を受け付け、おつりの値も出力する
- どのように設計してもよいが、結局合成系が優秀
 - TSMC(台湾の超有名なファブ)で評価、要するにきちんと見極めることが大事



面積160平方 μm (配線領域込)

$T_{pd\max}$ は10.11ns最大動作周波数98.91MHz
真面目に設計(なのに小さくならなかった)



面積103平方 μm (配線領域込) 64%程度削減

$T_{pd\max}$ は9.05ns(90%程度へ向上)最大動作周波数110MHz
ズルをして足し算を使った(だけど小さくなった)





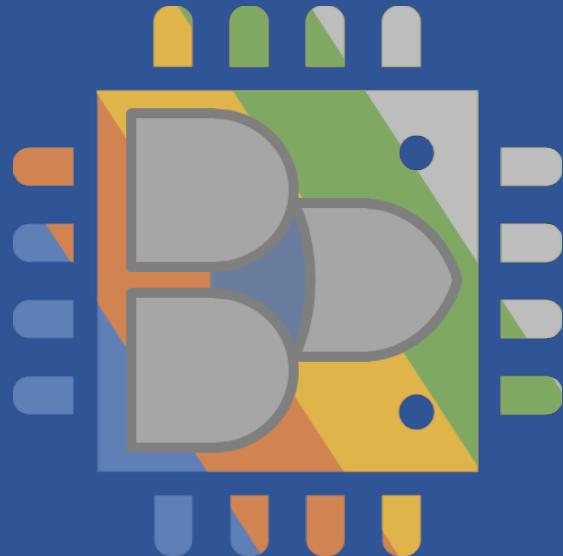
演習問題（5）

117

- 注意事項

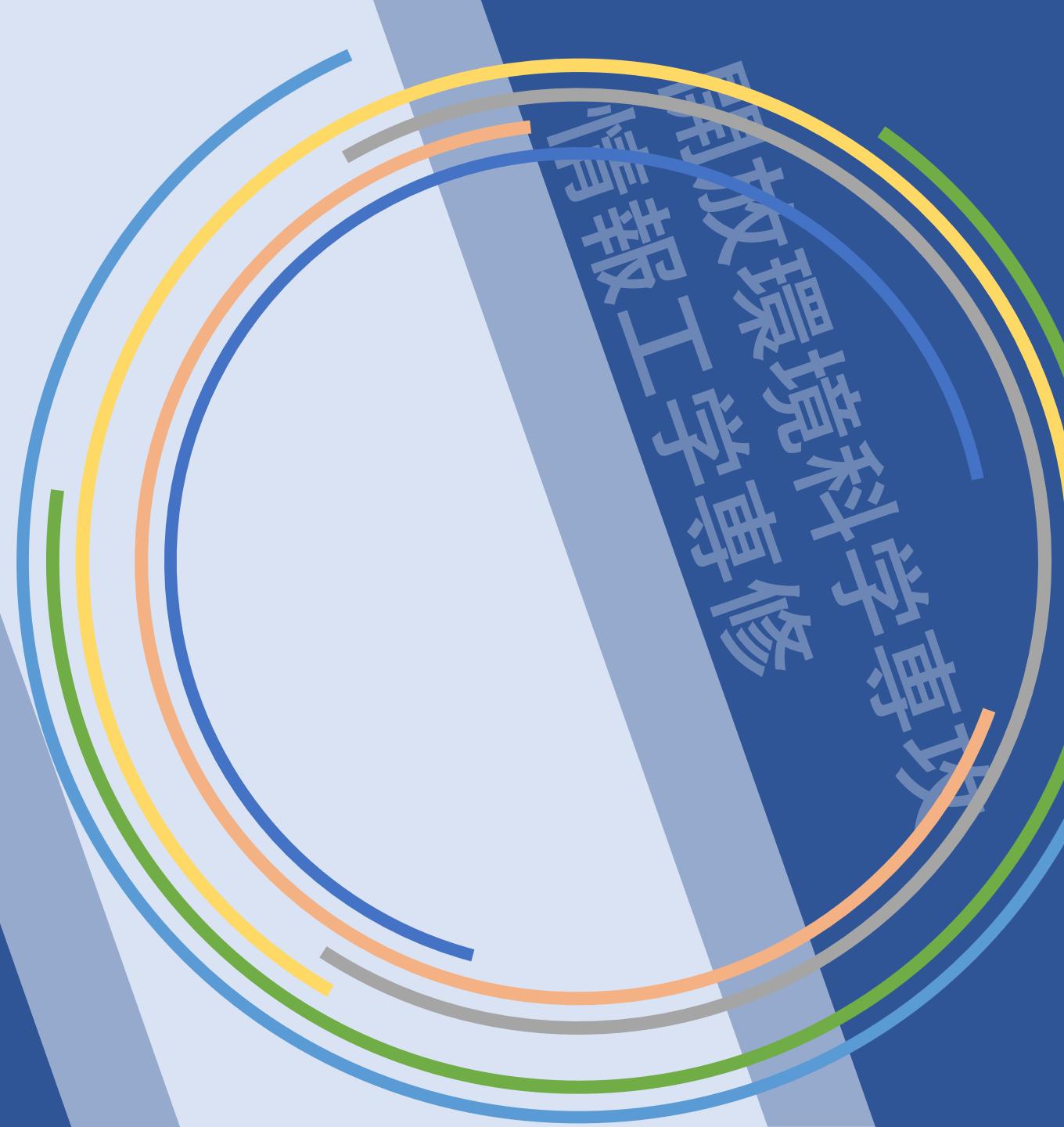
- レポートをMicrosoft Wordファイルで作成、LMSへ提出すること
- A4 2枚以内で作成し、最初にタイトルを「演習問題（5）」として記載し、名前と学籍番号も忘れずに記載すること
 - まずverilogソースを添付、シミュレーションを行い、動作している証拠としてgtkwaveの画面をキャプチャしてWordに張り付けなさい
 - 設計で利用した配線は全てgtkwave上で表示しておくこと(clk, rstなどすべて)
- verilogソースコードをテキスト情報として貼り付けること（画像ではない）
- 提出締め切りはLMSを確認すること





計算機システム 設計論(6) Verilog

担当： 西 宏章





Pitfall (落とし穴)

- 複数always文での1つのregへの代入

- 2箇所で代入されているのはだめ。回路的には出力衝突でショート
- always_comb, always_ffで防ぐことができる

```
always @(posedge clk)
  if(cond) dame <= in1;
always @(posedge clk)
  if(cond) dame <= in2;
```

- これは、 posedge clkに限定される話ではない
- 組み合わせ回路でも同様に問題になる
 - もちろん、 posedge clkと組み合わせ回路でも同じ
- よくある間違いは、 ステートマシンでの記述
 - ステートマシンは2つで構成される posedge clkでstate <= nstateとする
 - stateから、 nstateや他の信号線を制御する
 - posedge clk側で、 組み合わせ回路の初期化をしようとする
 - 初期化できるのはFFだけ！それもリセットだけ！
 - 組み合わせ回路は、 初期化されたFFの値を使って無難な制御をしていればよい
 - 配線は初期化できない！





Pitfall (落とし穴)

120

- やはり最多のは、組み合わせ回路の「全状態記述」
 - ここまで何度か説明してきましたが、毎年最も多いミスはここ
 - 組み合わせ回路は、必ず値が求まらないといけない
 - 複雑な組み合わせ回路はとにかくデフォルト代入を心掛けること
 - alwaysの後に全出力の値を確定させておくこと

```
always_comb begin  
    a = 0;  
    if(b = 0) a = 1;  
end
```

- ブロッキング代入 = とノンブロッキング代入 $<=$ の混合
 - 混ぜてはいけない
 - では、何が違うのか？

```
module test0(input a, output b, c, input clk);  
    logic a, b, c;  
    always @ (posedge clk) begin  
        b = a;  
        b <= c;  
    end  
endmodule
```





ブロッキング代入とノンブロッキング代入

121

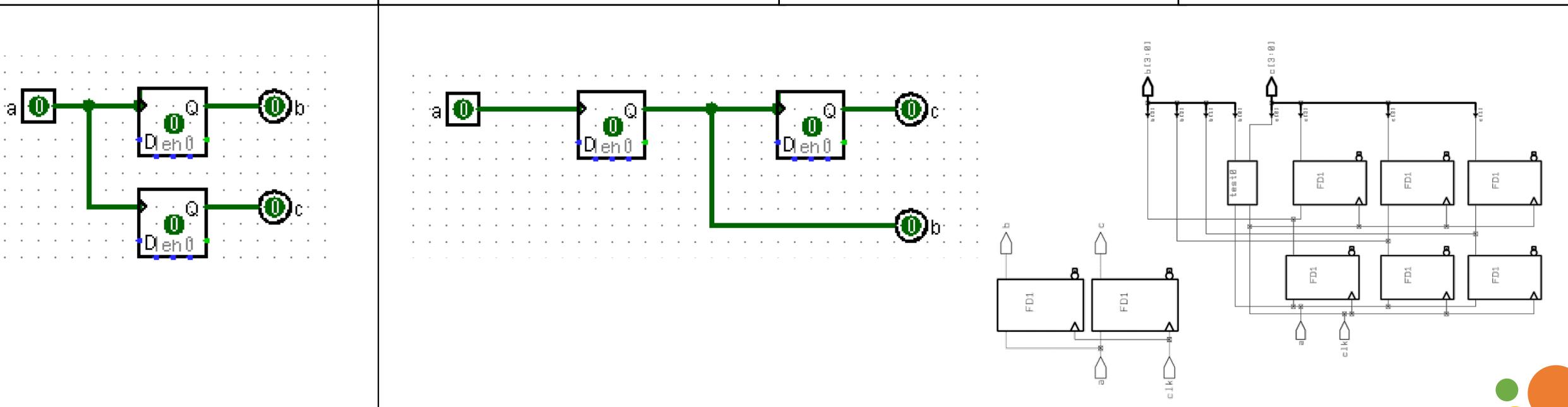
- 記述と回路合成結果をみてみよう

```
module test0(  
    input a,  
    output logic b, c,  
    input clk);  
always @(posedge clk) begin  
    b = a;  
    c = b;  
end  
endmodule
```

```
module test1(  
    input a,  
    output logic b, c,  
    input clk);  
always @(posedge clk) begin  
    c = b;  
    b = a;  
end  
endmodule
```

```
module test2(  
    input a,  
    output logic b, c,  
    input clk);  
always @(posedge clk) begin  
    b <= a;  
    c <= b;  
end  
endmodule
```

```
module test3(  
    input a,  
    output logic b, c,  
    input clk);  
always @(posedge clk) begin  
    c <= b;  
    b <= a;  
end  
endmodule
```





ブロッキング代入とノンブロッキング代入

122

- ・ソフトウェア設計者はブロッキングでよいと思う
 - $b = a; // b$ の値が a になる
 - $c = b; // c$ の値が b になるので、 $a=b=c$ になる
普通のプログラミング言語になれている人はこれが普通と思う
- ・ハードウェア設計者はブロッキングが変に思う
 - 頭の中はalways @(posedge clk)はFFを合成する
 - FF並べても、クロックごと同期して代入されるのは当たり前
 - シフトレジスタ参照
 - 厄介なのは、他のブロッキング全体の影響を受けるという点
 - ブロッキングで値が変化すると別のalwaysが反応して、また値を変えて…
 - 全て追いかける必要がある。変なところだけ並列記述
- ・ここに文化の違いが出てくる
 - ハードウェア構成を見ながら学ぶという立場のため
 - 組み合わせ回路はブロッキング = で
 - FFはノンブロッキング $<=$ で記述しよう、というポリシーを採用





Pitfall posedge は限定したほうが。。。

123

- posedge は何についても書けますが、その場合シミュレーションはできても合成できるかどうかはライブラリに左右される

- たとえば右の例は合成できません

```
always @(posedge clk or negedge clk) begin  
    out <= in; // 両エッジトリガは普通合成できない  
end
```

- 次も、clk でもrst でも反応する2相クロックの両立ち上がりエッジFFの動作になっているため合成できません

```
always @(posedge clk or posedge rst) begin  
    out <= in; // こんなセルみたことない  
end
```

- 次も、最初の条件が、同期リセットの pafun と非同期リセットの rst でできているため合成できません

```
always @(posedge clk or posedge rst) begin  
    if((rst == 1'b1) || (yabai == 1'b1)) // ここがダメ  
        out <= 1'b0;  
    else  
        out <= yabai;  
end
```





Pitfall clockラインへのゲート

124

- clock ラインにゲートを入れる gated clock というテクニックは普通は用いません
- 小電力設計や高度な高密度設計が必要なASIC設計ではいざ知らず、FPGAでは、利用すると遅延が伸びるなどの弊害が発生します





あらためて記述ポリシー

125

- verilog には多くの記述のスタイル、様々な文化圏が存在
 - SystemVerilogの登場でまとまりつつある
- 以下の点で説明を省略
 - taskは合成ならmodule で代用できる
 - function(void functionだけ意味がある。これはmoduleと同じ)
 - FPGA 設計で不要と思われる wired-or、wired-and、スリーステートも省略
 - 無駄な supply や(定数を代入すればよい)、双方向は面倒なので inout も省略





あらためて文法：値と演算子

126

- 値

- 10進:そのまま表記か、桁数'd
- 2進:桁数'b
- 16進:桁数'h
- この他にハイインピーダンス Z や不定 X がある
Z は合成には関係なく、X は don't care として利用できる

- 演算子

- Cに似ており目新しいのは連接演算子とリダクション演算子の動作ぐらい
優先順位もCと同じ
- 加算 +, 減算 -, 乗算 *, 除算 /, 剰余 % がある
- 除算 / および剰余 % は、オペランドが定数でなければ合成できない
- 演算結果が確定する場合は演算器は合成されない





- 関係演算子
 - 比較に使う。true は 1 を、false は 0 を返す。if構文や?演算子で用いる
 - <, <=, >, >=がある
- 等値演算子
 - ==, !=, ==?, !=? がある。関係演算子と同様に用いる
 - 比較演算でのワイルドカード (==?, !=?) が利用できる。casexと同じ
- 論理演算子
 - 否定 !, 論理 AND &&, 論理 OR || がある
- ビット演算子
 - ビット単位で演算を行う。バス同士では対応するビット間で演算される
 - 否定(単項)演算子 ~, AND &, OR |, XOR ^, XNOR ^~ もしくは ~^ がある
- リダクション演算子
 - オペランドの全ビットを演算して 1 ビットを返す
 - AND &, OR |, NAND ~&, NOR ~|, XOR ^, XNOR ^~ もしくは ~^ がある

```
if(v ==? 'hF??F) ...
else if (v !=? 'h0??0) ...
else if (v ==? 'b0000_????_0000_????) ...
else if (v !=? 'b1111_????_????_1111) ...
```





シフト・条件・連接演算子

128

- シフト演算子
 - 左シフト <<, 右シフト >>がある
 - signed指定による符号付の値については、<<<や>>>が使える。算術シフト。
- 条件演算子
 - ?(条件文) 設立時: 非設立時
という形で条件を記述する
 - 結果としてセレクタや入れ子にしてマルチプレクサを合成
 - 通常はif 構文を使ったほうが見やすい
- 連接演算子
 - 複数のビットオペランドを結合し、信号線を束ねてバスにすることができる
 - {と}で括り、, で並べる
 - {1'b1,1'b0,1'b0}は3'b100 と同じ
 - 繰り返し表現が可能で、同じ値を{1'b1,{2{1'b0}}}と書くことができる
 - 3bit 幅のバス a と4bit 幅のバス b があるとき、aの3bit目とbの4,3ビット目を連接する場合は{a[2],b[3:2]}と表記する





logicリテラル

129

- サイズ指定無しのlogicリテラルが用意されている。
 - 全ビット0の場合'0
 - 全ビット1の場合'1
 - 全ビットxの場合'x
 - 全ビットzの場合'z
- というように、全て同じスタイルでレジスタに定数を代入することが可能





2-State Types、Static/Automatic

130

- bitが利用できる
- 例えば、
 - bit clk;と宣言すれば、always #5 clk = ~clk; でclk = 0の初期化が不要(0/1だから)
- automatic宣言するとfunction/taskにおいて一時変数のように利用可能
 - でもどちらも教えてないので、スルー





function

131

- functionの例
 - 利用してもvoid functionか？
 - void functionは、alwaysの中で利用できるため、alwaysの中を関数化し効率化できる

```
module selE(in0, in1, sel, out);
    input in0, in1;
    output out;
    function selout; // バス出力の場合ビット幅も記述する
        input in0, in1, sel;
        begin
            if(sel) selout = in0;
            else selout = in1;
        end
    endfunction
    assign out = selout(in0, in1, sel);
endmodule
```





Stream Operator(<<)とinside演算子

132

- Stream Operator(<<)
 - ベクターのビット順を入れ替え反転させる。つまり、ビット列を右から読み出す
 - これを使えばエンディアン変換等が簡潔に記述できる
 - $\{<<\{4'b1110\}\}=4'b0111$ 、 $\{<<2\{4'b1110\}\}=4'b1011$ (右から2づつ取り出す)
- Inside演算子
 - insideはある信号のある範囲で値の集合の中にマッチするかを判定する演算子
 - 合成可能！
 - ifやcaseで利用可能

```
if(val inside {[0:100],254,255}) flag = 'b1;  
if(val inside {8'b00??_????,[64:100], 8'b1111_111?}) flag = 'b1;
```





Parameter を使った宣言

133

- 次の方法がある
 - モジュールの先頭で宣言
 - ポート宣言の前に #() で宣言
- デフォルト値を指定すること
 - 後者はポート幅変更も可能

```
module mdl (input clk, rst, output logic [7:0] o );
  parameter PRMT = 16;
  reg [(PRMT-1):0] r;
  ...
endmodule
```

```
module mdl #(parameter PRMT = 24)
  ( input clk, rst, output logic [7:0] LED );
  reg [(PRMT-1):0] r;
  ...
endmodule
```





- 設計は小さなモジュール毎に行い、各モジュールの動作検証を終えて動作が確認されている小さなモジュールを集めてより大きなものを構成する
 - Waterfall型ではなくAgile型の開発スタイルで
 - 各モジュールはそれぞれ独自のテスト環境を持たせる
 - テストモジュールは同一ファイルに入れると合成できないため、別のファイルに記述してverilog シミュレーションで一緒にロードする
 - 単純な and 回路のモジュール and(ina, inb, out)のテスト記述例は次の通り

```
'timescale 1ps/1ps
module test;
    logic ina, inb, out;
    and hogeand(ina, inb, out);
    always #10 ina = ~ina;
    initial begin
        $dumpfile("test.vcd");
        $dumpvars(0, hogeand);
        $dumplimit(1000000);
        $monitor($stime,, "ina:%b inb:%b out:%b", ina, inb, out);
```

```
    ina = 0;
    inb = 0;
    #50
        inb = 1;
    #50
        inb = 0;
    #50
        $finish;
    end
endmodule
```



- テストモジュールでの記述
 - テストモジュールに入力出力はない
 - timescale 1ps/1psでシミュレーションのレゾリューション指定
 - レゾリューションが ps に指定されているため、#10は10ps後の意味
 - ここでのalways文の記述はクロック入力でも重要でinが10psごとにHigh/Lowを繰り返す
 - \$dumpfile("test.vcd");
 - test.vcd というファイル名で波形データを保存する指令
 - \$dumpvars(0, hogeand);
 - hogeandモジュールから0階層(全階層)下るまで全ての信号線を波形抽出せよという指令
 - \$dumpvars(0, hogeand, 1, gege, 0, pafun, 3, kyain);と複数のモジュールを指定できる
 - \$dumplimit(1000000);
 - 生成する波形データのファイルの大きさの制限を指定
 - \$monitorはCとほぼ同じフォーマットで画面出力させる
 - 指定されている信号に変化があったときだけ出力
 - \$displayは評価されたときだけ値を出力(C言語同様のフォーマット指定ができる)
 - #数字 経過時間の指定で、相対時間
 - \$finishはシミュレーション終了を意味する





\$monitor, \$displayのフォーマット指定

136

- \$display("フォーマット", 引数1, 引数2, ...);
- \$monitor("フォーマット", 引数1, 引数2, ...);
- 同時に使うと便利なシステム変数
 - \$realtime
 - real型で実時間を表す
 - \$display("Time = %f", \$realtime);などとする
 - \$time
 - 64bit整数でtimescaleを積算すれば実時間
 - \$bitstoreal、\$realtobits
 - real型と64bitの変換
 - 実数計算を行う回路のデバッグで利用
 - \$itor、\$rtoi
 - real型とinteger型の変換
 - \$sin、\$cos、乱数など様々な関数がある

%n	改行
%t	タブ
%%	¥を出力
%"	"を出力
%ddd	ASCIIコード(8進)文字出力
%%	%を出力
%b %B	2進数
%d %D	10進数
%e %E	指数
%f %F	実数
%g %G	%e、%fの桁数が少ない方を出力
%h %H	16進数
%l %L	ライブラリ・バインド情報
%m %M	階層名
%o %O	8進数
%s %S	文字列
%t %T	時刻
%u %U	0,1の2値出力(X,Zは0)
%v %V	信号強度
%z %Z	0,1,X,Zの4値出力



- ・シミュレーションの効率化
 - ・たとえばtest.v と and のファイル and.v があったとき、合成は and.v に対してのみ行い、シミュレーションはverilog test.v and.vとして行う
- ・ファイル入出力
 - ・\$readmemb(ファイル名, メモリレジスタ名)
 - ・\$readmemh(同様)

ファイルのデータが2進か16進で使い分け、ファイルの中身をメモリに読み込む
 ファイルの中身は「@アドレス データ」もしくは、単に「データ」の行だけで構成

 - ・\$fopen(ファイル名)
 C同様にファイルディスクリプタの値を返すため、integer 等に保存して
 \$fdisplay(ファイルディスクリプタ, フォーマット, 変数リスト)で書き込む
 - ・全てinitial begin end の中で行うこと

```
--- file : readmemb ---   --- file : readmemh ---
00000001               01
00000010               02
00000100               04
00001000               08
00010000               10
00100000               20
01000000               40
10000000               80
```

```
integer fdo; // 最初に指定しておく
fdo = fopen("test.out"); // initial ブロック内で指定する
$fdisplay(fdo, "ina:%b inb:%b out:%b", ina, inb, out);
```





メモリ内容のダンプについて

138

- 次のように記述する

```
integer i; // 追加で必要
```

```
initial begin
```

```
    $dumpfile("memorydump.vcd");
```

```
    $dumpvars(0, 階層指定);
```

```
    for(i = 0; i < 2**アドレス幅; i++) // このfor節が追加で必要
```

```
        $dumpvars(1,mem.data[i]); // メモリのデータバスをそれぞれ指定
```

```
end
```





SystemVerilogの特徴

139

- 冗長記述の削除
 - ポート名とネット名が同じ時はポート名を省略できる
- 多次元配列
 - SystemVerilogは多次元配列をサポート logic [3:0][1:0][7:0] mem [255 : 0] ;
- 列挙データ型
 - parameterよりも使いやすい typedef enum reg {Red, Green, Yellow} signal;
- テストベンチ設計で恩恵
 - オブジェクト指向：struct・union・classをサポート
 - 何かが変化しながら、何かを変化させてとか簡単に書ける：join_any, join_none
- データ配列のコピーが可能(for文がいらない)
- C言語とのインターフェースが易化
- always_comb/always_ff/always_latchをサポート
 - clocking、ifやcaseにおけるuniqueやpriority(iv未サポート)、reg/wireを融合したlogic
 - **Implicit port connection** 神ですね
 - **SystemVerilog assertion**
 - Associated Array(文字などの参照配列), Dynamic Arrayとか何でもありか！
 - 亂数とかもあるし、2値設計もできて超高速シミュレーション！





ただし、過度な期待はしない

140

- always_combを使ってもラッチが生成される場合がある
 - 常に合成後のレポートでラッチがないことを確認？であれば意味ないかも
 - 処理系頑張ってください
- シミュレーション用の拡張が多め
 - 合成記述は比較的しっかり記述しないと
- コマンドラインでオプションを指定しないといけない
 - Icrusでは、iverilog -g2012とする
- 拡張子はsvにするべき
 - vだとこれまでの文法で記載されていると思われ、合成などでエラー頻発
- 曖昧さは仮想化さ！と言い放ち、言語仕様の拡張でカバーしてしまった
 - package = namespaceもサポート





Implicit port connection

141

- SystemVerilogではポートの接続方法の略記法が拡張されている
- ポートの接続には以下の方法がある
 - 順番を守って接続（今まで通り）
 - 名前を表記して、順番の入れ替わりを許す（拡張されている）
 - 完全に省略する（神）

```
module test1(); // 順番
    wire [3:0] command;
    wire [7:0] wdata;
    wire [7:0] rdata;
    host h(wdata, rdata);
    target t(rdata, wdata);
endmodule
module host(
    output logic [7:0] wdata,
    input logic [7:0] rdata);
    assign wdata = 8'h55;
endmodule
module target(
    output logic [7:0] rdata,
    input logic [7:0] wdata);
    assign rdata = wdata + 8'b1;
endmodule
```

```
module test1(); // 名前で
    wire [3:0] command;
    wire [7:0] wdata;
    wire [7:0] rdata;
    host h(.wdata(wdata), .rdata(rdata));
    target t(.rdata(rdata), .wdata(wdata));
endmodule
module host(
    output logic [7:0] wdata,
    input logic [7:0] rdata);
    assign wdata = 8'h55;
endmodule
module target(
    output logic [7:0] rdata,
    input logic [7:0] wdata);
    assign rdata = wdata + 8'b1;
endmodule
```

```
module test1(); // 同名なら略可
    wire [3:0] command;
    wire [7:0] wdata;
    wire [7:0] rdata;
    host h(.wdata, .rdata);
    target t(.rdata, .wdata);
endmodule
module host(
    output logic [7:0] wdata,
    input logic [7:0] rdata);
    assign wdata = 8'h55;
endmodule
module target(
    output logic [7:0] rdata,
    input logic [7:0] wdata);
    assign rdata = wdata + 8'b1;
endmodule
```

```
module test1(); // 神！ 差分でOK
    wire [3:0] command;
    wire [7:0] wdata;
    wire [7:0] rdata;
    host h.*;
    target t.*;
endmodule
module host(
    output logic [7:0] wdata,
    input logic [7:0] rdata);
    assign wdata = 8'h55;
endmodule
module target(
    output logic [7:0] rdata,
    input logic [7:0] wdata);
    assign rdata = wdata + 8'b1;
endmodule
```



Interface (神)

142

- module間接続を簡略化でき、バス幅や向きも別途決定できる
- ただし、Icarus Verilogでは実装が完全ではないため残念ながら省略

```
interface #(parameter WID=8) busif(input logic clk, rst);  
    logic[7:0] a;  
    logic[WID -1:0] wd;  
    modport mm(output a, wd);  
    modport ms(input a, wd, clk, rst);  
endinterface
```

```
module top(input logic clk, rst);  
    busif #(8) bm(.*);  
    busif #(16) bs(.*);  
    mst mst( .*, .bm(bm.mm), .bm(bs.ms) );  
    slave slv1( .bus(bm.ms) );  
    slave_sub slv2( .bus(bs.ms) );  
endmodule
```

```
module slave( busif.ms bus);  
    slave_sub sub(.bus);  
endmodule
```

```
module slave_sub (busif.ms bus);  
endmodule
```

```
module mst (clk, rst, busif.mm bus1, bu2);  
...  
endmodule
```

 martinwhitaker commented on 8 Oct 2017

Sorry, no, Icarus does not currently support interfaces in any useful way. It lets you define them, but that's all. Adding support for this would be a major piece of work.

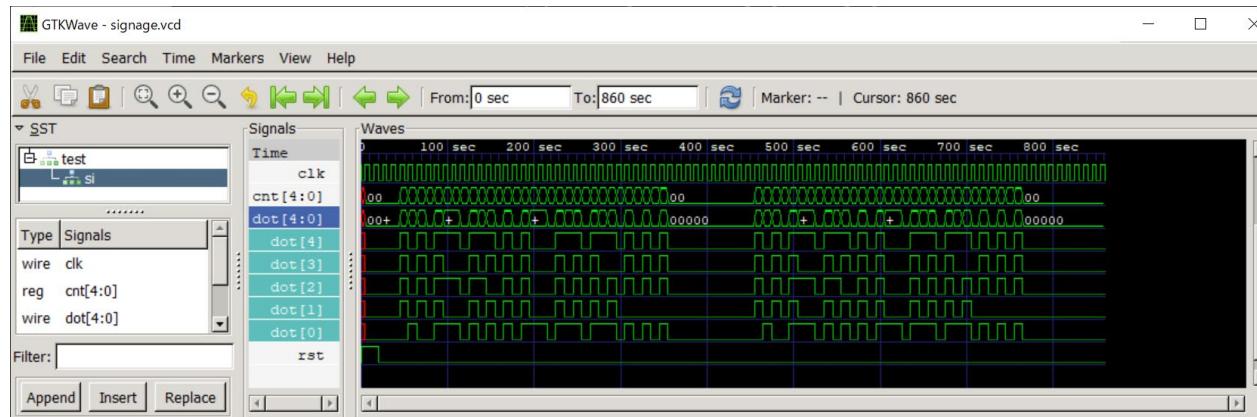




演習問題（6）

143

- 次の動作を行う回路を設計しなさい
 - LEDアレイによるサイネージを制御するコントローラ
 - 5×32 (5ビットアドレッシング) のメモリを準備
 - このメモリの中身を0番地から15番地まで順に読み出す
 - 表示開始信号stがhighになったら次々に読み出し
 - 全て読み終わったら停止、次にst信号線がhighになるまで待機
 - <http://mksd.jp/fingerfive.html>などを参考に5bitビットマップフォントで文章を作成
 - この内容を\$readmembや、\$readmemhなどを用いてメモリにダウンロード
 - 読み出した内容を出力、 gtkwave上で文字出力を確認（VERILOG!!!と書いた）





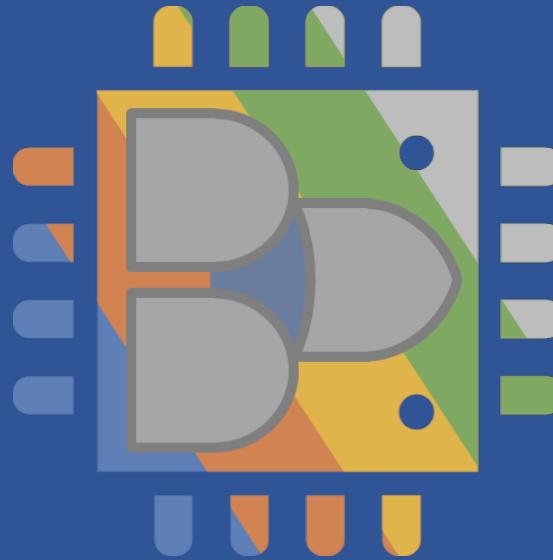
演習問題（6）

144

- 注意事項

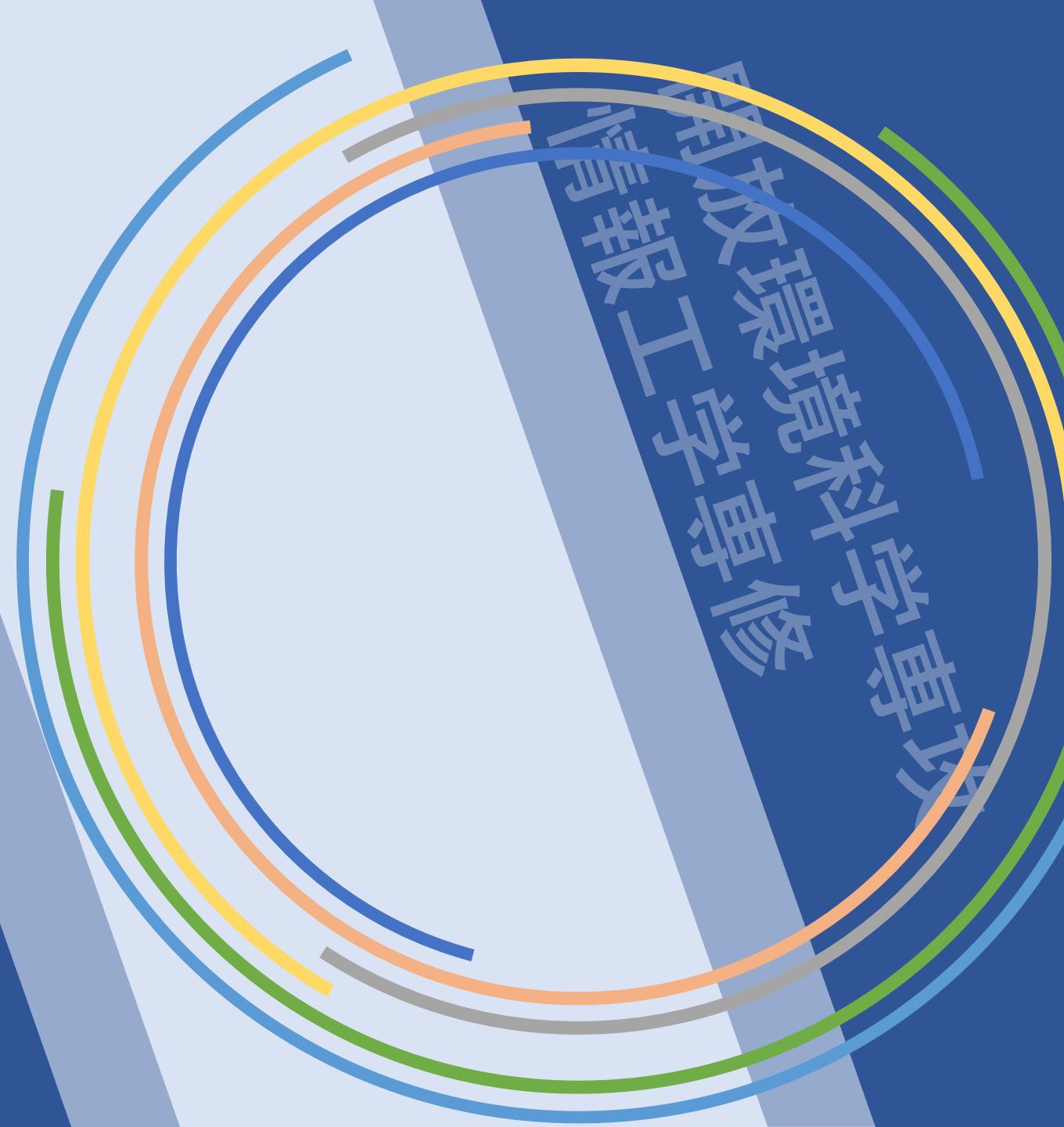
- レポートをMicrosoft Wordファイルで作成、LMSへ提出すること
- A4 2枚程度で作成し、最初にタイトルを「演習問題（5）」として記載し、名前と学籍番号も忘れずに記載すること
 - まずverilogソースを添付、シミュレーションを行い、動作している証拠としてgtkwaveの画面をキャプチャしてWordに張り付けなさい
 - 設計で利用した配線は全てgtkwave上で表示しておくこと(clk, rstなどすべて)
- verilogソースコードをテキスト情報として貼り付けること（画像ではない）
- 提出締め切りはLMSを確認すること





計算機システム 設計論(7) 論理合成

担当： 西 宏章





合成系のインストール

146

- フリーで合成の感覚を学びたい
 - あまり良いツールがないのですが、とりあえずならばyosys
 - <https://github.com/cliffordwolf/yosys.git>
 - ただし、SystemVerilogに部分対応
 - Verilog-2005は対応、SystemVerilogはかなり対応しているようであるが、微妙？
 - 対応状況はicarusよりも悪い?例えばenumは最新バージョン以外エラーになる
- インストール
 - Windowsならば、Downloadから<http://bygone.clairexen.net/yosys/nogit/win32/yosys-win32-mxebin-0.9.zip>を入手して展開
 - 面倒なので、iverilogと同じフォルダ(c:\$iverilog\$bin)にすべてコピーしておく
 - このテキストの例ならば、c:\$iverilog\$binに入っているはず。パスも通っているはず
- MacOSX/Linuxへのインストール
 - 厄介なので、次のページへ
 - gitに行けばすべて掲載されている
 - Linux, Cygwinでのコンパイル情報あり



- まずはソース一式を入手する

```
$ git clone https://github.com/cliffordwolf/yosys.git
$ cd yosys
```

- コンパイル環境を整える

- MacOSの場合

- 既にHomebrewが入っているという前提、icarus verilogをgitから入れたらな大丈夫なはず
\$ brew tap Homebrew/bundle && brew bundle

- Linuxの場合はiverilogのインストールでgitから入れる方法をまずは選択

- \$sudo apt install tcl-dev libreadline-dev libffi-dev としておくこと。
CentOSは\$sudo yum install readline-devel libffi-devel

- コンパイルの準備

```
$ make config-clang; make config-gcc
```

- コンパイルし、インストールする。インストールは当然rootアカウント

```
$ make #相当時間かかるので覚悟すること
$ sudo make install
```

- 動作テストする

```
$make test
```





ついでにGraphvizをインストール

148

- 合成結果の確認に必要
- インストール
 - Windows
 - ダウンロード <https://graphviz.gitlab.io/download/>
 - ただし、最近のバージョンにはGUIが入らないという罠がある
 - 古いバージョンだが、K-LMSからインストールするか、諦めてdotコマンドを使う
 - インストールすると、 にgvedit.exeが現れる（最新版はgveditが含まれていない）
 - Dotを使う場合は、 dot -Tpdf -o show.pdf show.dot などとして、 pdfを作る
 - MacOS
 - % brew install graphviz でOK。なお、yosysを導入すると既に入っている可能性が高い
 - gveditではなく、xdotコマンドを利用
 - yosys内でshowコマンドによりxdotを使って直接表示されるのでとても便利
 - ただし終了はCtrl-Cで。exitコマンドを使うと、シェルごと持っていく
 - Linux
 - % apt install graphviz





Windowsの追加設定

149

- Windowsの場合、gveditを直接呼び出すことができないのでパスを通そう
 - まず を右クリック、システム->システム情報->(左)システムの詳細設定->環境変数->ユーザの環境変数の中のPath->編集->新規
C:\Program Files (x86)\Graphviz2.38\bin と記入->OKとしてPATHに加える
 - これで、コマンドラインから直接gvedit.exeが呼び出せる
 - showコマンドでdotファイルを作成後、gvedit dotファイル名とする
- MacOSやLinuxの方が便利
- シンプルな回路なら、オンラインサービスの方が良いかも？
<https://dreampuf.github.io/GraphvizOnline/>



- System Verilogにもそれなりに対応
 - Implicit port connectionは例外指定などは受け付けないなど、また未対応が多い
- それなりに様々な合成操作ができる
 - synopsys directiveをサポート
 - nomem2regによりFF array(memory)の合成を阻止できる
 - nolatchesでラッチの混入を阻止できる
- よくある合成系とよく似た操作
 - 下記コマンドを入力してもよいし、スクリプトにして実行してもよい
 - 例えば、以下をscr.vとし、yosys scr.vで実行(実はtclコマンド)

```
read_verilog -sv design.v
hierarchy
proc; opt
techmap; opt
write_verilog synth.v
show
```

```
read_verilog -sv design.v
hierarchy
proc; opt
techmap; opt
dfflibmap -liberty yosys/examples/cmos/cmos_cells.lib
abc -liberty yosys/examples/cmos/cmos_cells.lib; opt
write_verilog synth.v
show
```





典型的な合成スクリプト

151

- read_verilog -sv design.v # ファイル(ここでは一つだけ)読込、 svでSystemVerilog
- proc # High-level 記述をDFFとMUXに変換、 コメントは#で記述
- opt # ここで一度最適化
- memory # High-level メモリをDFFとMUXに変換
- opt # 最適化、 何かするとoptしたほうがよいようだ
- techmap # gate-levelネットリストに変換
- opt # さらに一度最適化
- dfflibmap -liberty cells.lib # レジスタをcellライブラリに割り当てる
- abc -liberty cells.lib # 残りをcellsライブラリに割り当てる
- opt # さらに最適化
- clean # 未使用cellやワイヤを削除
- opt # さらに最適化
- write_verilog synth.v # ネットリストを書き出し
- show # MacOSならこれでよい WindowsでPATHを通せば -viewer gvedit





簡便化するために

152

- synコマンドで自動的に実行してくれる
 - -top top_module トップモジュールを指定、-auto-top で自動検索
 - -flatten 階層を破壊してまっ平にする

- 以下の処理を一気に行ってくれる

begin:

hierarchy -check [-top <top> | -auto-top]

coarse:

proc; flatten (if -flatten); opt_expr; opt_clean; check; opt

wreduce; alumacc; share; opt

fsm; opt -fast

memory -nomap; opt_clean

fine:

opt -fast -full

memory_map; opt -full

techmap; opt -fast

abc -fast; opt -fast

check:

hierarchy -check; stat; check

- synth_xilinxなどもあり、FPGA向け合成が可能となっている





より本格的な合成

153

- その他の重要・便利そうなコマンド
 - uniquify # 同じモジュールを複数利用しているとき、コピーする(最適化を進める)
 - flatten # 階層を破壊してまっ平にする(最適化を進める)
 - verilog_defaults –add オプション # read_verilogのデフォルトオプションを指定
 - verilog_defins オプション # マクロの定義を行う –Dnameで定義、 –Unameで解除
 - write_edif ファイル名 # EDIF netlistを出力
 - その他SPICEモデルを含む様々なフォーマットのファイルを出力可能、 write_table等もある
 - abc
 - ABC toolと呼ばれるツールを用いてテクノロジのマッピングを行う
 - -script file yosys-abc専用のスクリプトファイルを指定する
 - syn同様、自動的にスクリプトは充てられる
 - -liberty file 指定されたCellライブラリでネットリストを生成する
 - -constr file 制約ファイルを指定する(後述)
 - -D <pico sec> 遅延制約を与える
 - -g type 利用するGateの種類を指定できる(マニュアル参照)





- -libertyと一緒に利用し、タイミング制約(timing constraints)を与える
 - constr fileは以下の2つの行の組み合わせで構成される
 - set_driving_cell <cell_name>
 - どのセルの入力かを指定
 - set_load <floating_point_number>
 - fF(フェムトファラッド)で各出力への容量を指定
 - 残念ながら、この形態では時間を用いて遅延制約を与えることは困難

• 実践的な合成スクリプトの例

```
read_verilog -sv design.v # 複数あるなら、全部指定すること。Iverilogと同じ。  
read_liberty -lib cells.lib # 不要かもしれない。tclでは、set lib cells.libとして、$libで参照できる  
proc  
  flatten # ただしこれを行うと回路図を見ることはほぼあきらめること  
synth -auto-top  
write_verilog synth.v  
dfflibmap -liberty cells.lib  
abc -D 5000 -liberty cells.lib  
stat -liberty cells.lib  
write_verilog gate.v
```

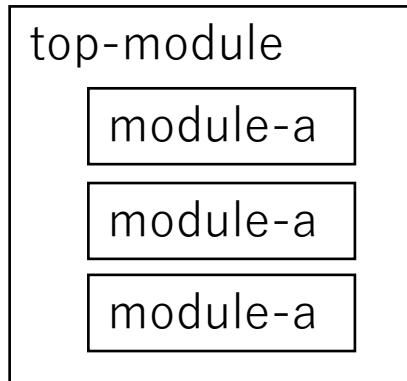




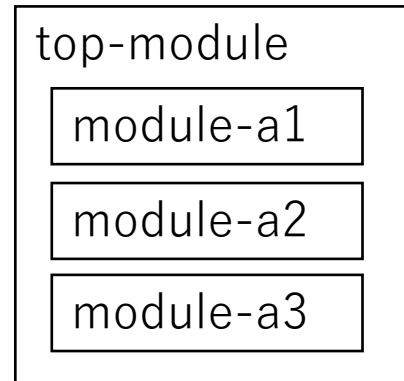
Uniquify / Flatten

155

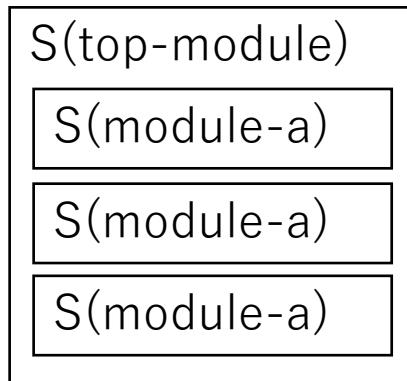
- Uniquify



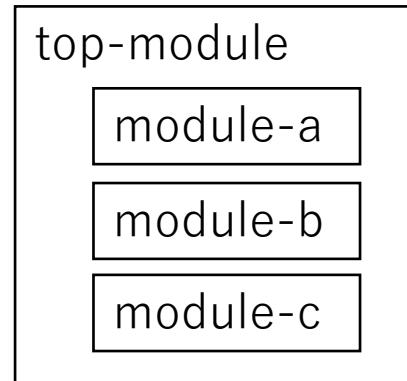
Uniquify



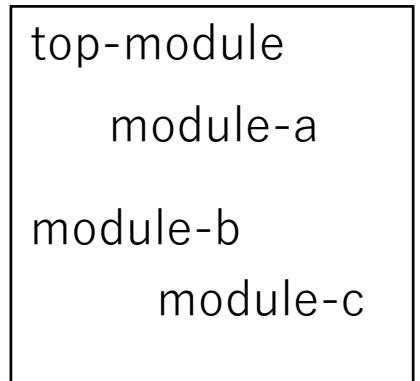
Synthesize



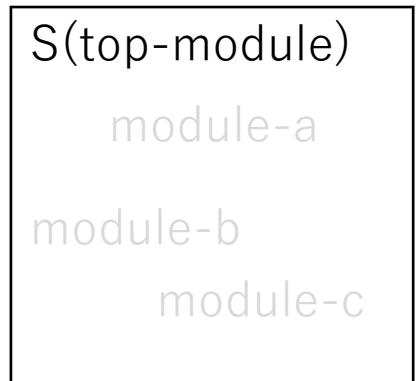
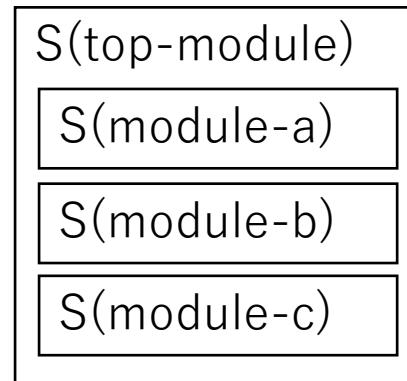
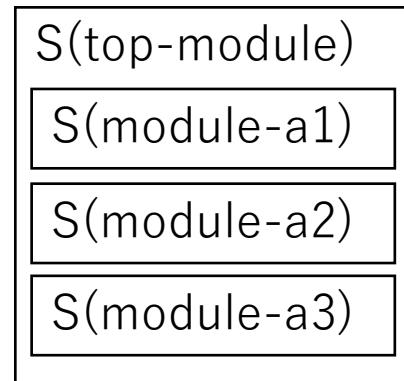
- Flatten



Flatten



Synthesize



FPGA設計、例えばVivadoでは、ungroup(IP展開), flatten(同じ)などの類似機能が存在する





ライブラリを使う

156

- 提供ライブラリ

- FreePKD45プロセスデザインキット(PDK)
 - 45nmテクノロジによるオープンソース、オープンアクセスベースPDK
 - 実際にASIC製作ができるクオリティで構成されたライブラリ、TSMCなどが製作請負可能
 - box.comの授業のフォルダ内、EDAの中にある(FreePDK-NangateOpenCellLibrary.zip)

- MOSIS_SCMOS(TSMC18n)ライブラリ

- Oklahoma State University (OSU) によるオープンなスタンダードセルライブラリ
- fast, typical, slowが一つで全て含まれており、iverilog -T typ,min,maxとして指定する
- box.comの授業のフォルダ内、EDAの中にある(osu018_stdcell.zip)

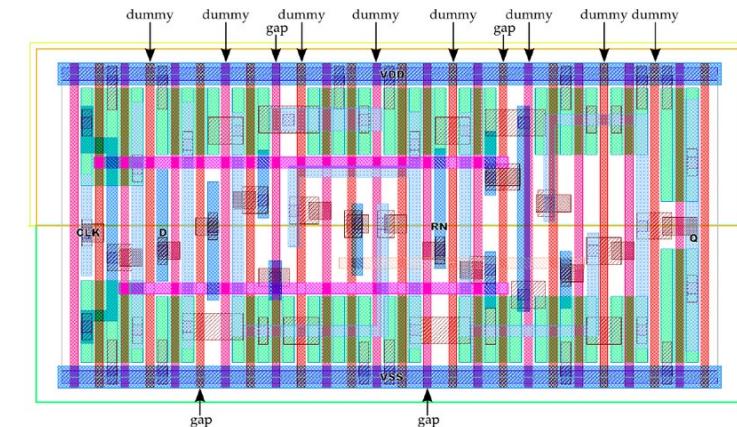
- 内容

- fast(min):最速タイミング制約・setupエラーの検出用
- typical(typ):通常性能による制約・利用価値は小さい
- slow(max):最悪性能値・マニュファクチャラーが保証する値
- 実際に製造する場合はfast, typical, slow全て通す

- 利用準備

- 展開してソースツリーの階層においておく

- 利用しやすいように、verilog/ex1 であれば、verilogの下に置くとよいy-liberty ..//xxxなどと指定できるため扱いやすい

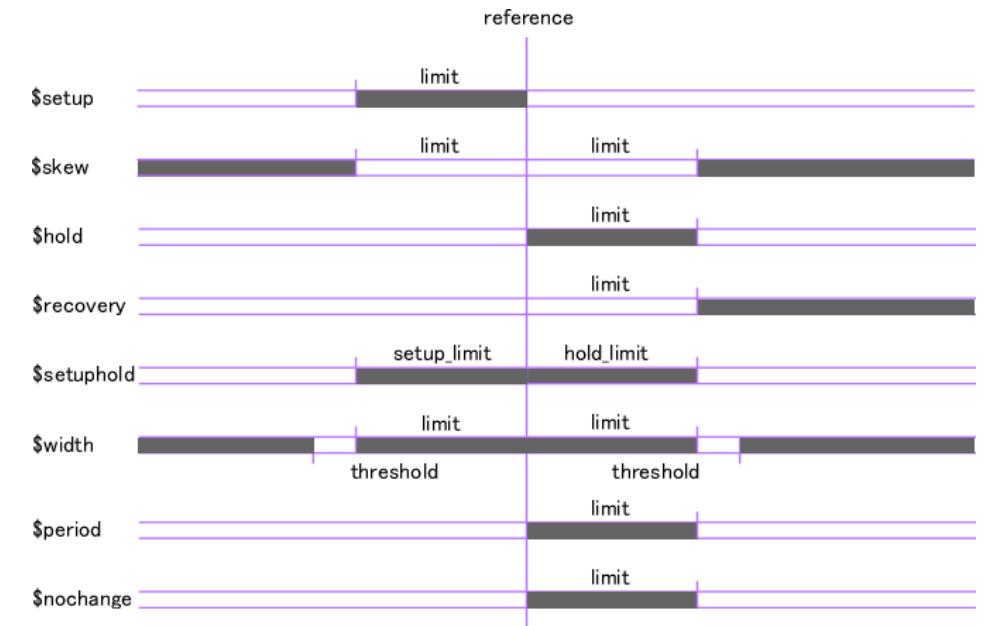




ライブラリを覗く

157

- 何が違うのか?
 - primitive – endprimitiveがある
 - RTLによるVerilogネットリスト記述よりもさらにプリミティブなセルレベル記述がある
 - 内容は真理値表
 - specify – endspecifyがある
 - module内のspecifyブロックでモジュール内の信号同士の間のディレイを規定
 - specify
(i => o) = (1.0, 1.2); //iからoへの遅延がrise,fallそれぞれ単位時間の1.0倍と1.2倍
 - endspecify
 - 使われていないが、右の表の値も設定できる
- 実はつまらなかった
 - FreePDKはfastもslowも全部同じであった
 - osu18はなかなかよさそう

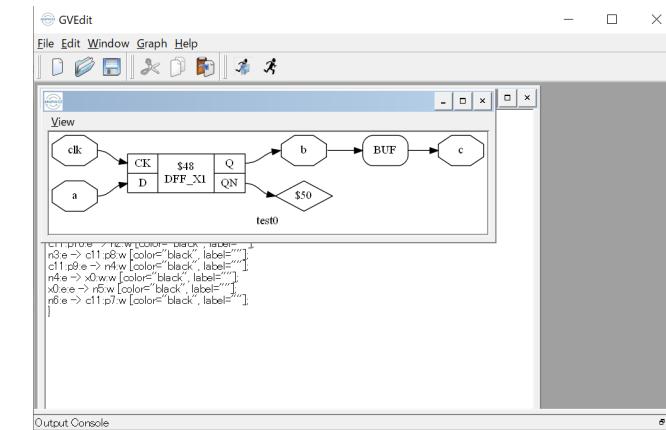




実際に合成してみよう

158

- 前回のtest0, test1の違いを確認
- 練習のため、typicalで評価
 - osu018_stdcells.libを使う
 - 合成すると、ブロッキング代入とノンブロッキング代入が結果として表れていることがわかる
 - テストモジュールを作成して仮負荷シミュレーションをやってみよう
 - Verilog出力したファイルと、テストモジュール、osu018_stdcells.vこの3つを同時にシミュレーションする
 - iverilog -gspecify -T (min,typ,max)として遅延付き指定とし、遅延モデルを選択する
- RTLレベル設計でのシミュレーションと何が違うか確認しよう
- dotファイルを確認してみよう
 - オンラインでも確認できます。<https://stamm-wilbrandt.de/GraphvizFiddle/#>





(base) west@ropross:/mnt/c/Users/west/lec-compsys/SYN\$

<https://youtu.be/kkaGaTns8TQ>

59

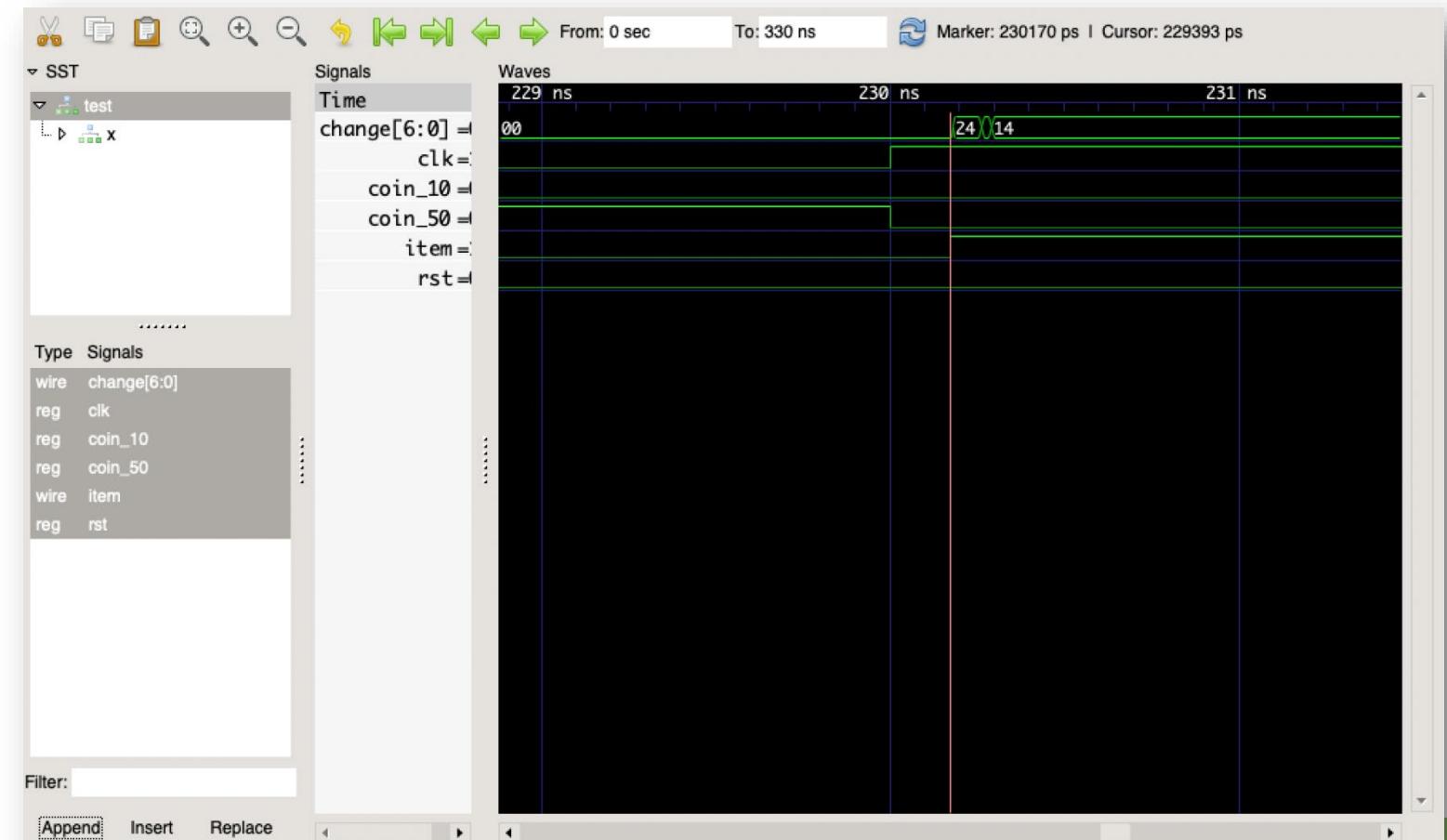




遅延付きシミュレーションの例

160

- 遅延付きシミュレーションによって、同時に変化していた信号がバラバラに動作するようになり、リップル（信号バタつき）が見えるようになる
 - 特にバスの値が変化しているのがわかる





合成と遅延つきシミュレーションの注意点

161

- 同じテストモジュールでテストするとエラーになった?
 - 通常それでよいが、設計によってはエラーになる可能性がある
 - 例えば次のようなエラーが発生する
error: reg hoge; cannot be driven by primitives or continuous assignment.
 - これは、テストモジュールで試験対象となる合成結果に含まれるoutput hogeという信号を入力として受けるときに、logic hogeと宣言していると発生する
 - 合成前はalwaysなどでhogeを出力しており問題はなかった
 - ところが合成するとassign hogeになってしまった
 - 結果としてlogicではなく、wireで受けなければならなくなつた
 - **テストモジュールは、試験対象となるモジュールの入力をlogicで、出力をwireで設計しておけば、合成前、合成後を問わず共通化できる**
- yosysの問題点
 - タイミング制約合成は一般的なサポートとしては行われていない
 - 時間制約を持たせた合成ができない
 - タイミング解析もできない（ここが一番まずい）
 - 通常はStatic Timing Analysis(STA)を生成して、バックアノテートして解析する





シミュレーションループ問題

162

- vvpでシミュレーションすると実行が固まり反応しなくなる
 - バグではありません。設計・記述がまずいのです
- 例えば次のようなモデルを考えます
 - これは合成可能です。ただし、実際に回路にすると「発振」します。
 - $a=1$ で最初 $c=0$ のとき $d=1$ になります。すると、 $b=1$ で $d=1$ なので、 $c=1$ になります。ところが、 $a=1$ で $c=1$ なので、 $d=0$ になります。すると、 $c=0$ になり…

```
module loop(input a, input b, output logic c);
    logic d;
    always_comb
        if(a == 1'b1 && c == 0) d = 1;
        else d = 0;
    always_comb
        if(b == 1'b1 && d == 1) c = 1;
        else c = 0;
endmodule
```

```
module test;
    reg a, b;
    wire c;
    loop loop(a, b, c);
initial begin
    a = 1;
    b = 1;
    #20
    $finish;
end
endmodule
```





シミュレーションループ問題の解決法

163

- 繰り返しますがシミュレータが悪いではありません
 - 商用のverilogシミュレータでも、ループは発生します。
 - ループを抜けられない場合は自動的にシミュレーションが停止
 - その後、デバッグモードに入リステップ実行でループを追いかけます
- 流石にiverilogはそこまではできませんが、ループを追いかけることはできます
 - あまり役に立たないかもしれない方法
 - vvpが固まったらCtrl-Cで停止してインタラクティブモードへ
 - push/pop、stepを利用してトレース(help参照)
 - シンプルかつ簡便な方法
 - iverilog -g2012 -pfileline=1 とします。
これでvvpのデバッグメッセージが出てきます
 - ソースの何行目を実行しているかを追跡できます
 - この場合にインタラクティブモードに入るとより詳しくレポートがでます
 - 面倒だがより強力な方法
 - \$displayを埋め込んで実行順番や配線の値を追う方法
 - ただし、iverilogによるinitial begin end にない\$displayの扱いに依存します。通常は、文法上も動作上も問題ありません。
 - 単純な修正方法は「設計方針がっているならば」FFを入れてループを切ること

```
loop.v:4: If statement.  
loop.v:5: Blocking assignment.  
loop.v:4: Event wait (@) statement.  
loop.v:7: If statement.  
loop.v:8: Blocking assignment.  
loop.v:7: Event wait (@) statement.  
loop.v:4: If statement.  
loop.v:4: Blocking assignment.  
loop.v:4: Event wait (@) statement.  
loop.v:7: If statement.  
loop.v:7: Blocking assignment.  
loop.v:7: Event wait (@) statement.  
loop.v:4: If statement.  
loop.v:5: Blocking assignment.  
loop.v:4: Event wait (@) statement.  
loop.v:7: If statement.  
loop.v:8: Blocking assignment.  
...
```





テストモジュール記述上の問題（重要）

164

- 実装にも依存しますが、テストモジュールは、logicを使わないう方が無難
 - yosysが合成でassingを使ってきますが、assignに対してlogicが上手く動作せず、iverilogがエラーになる場合があります
 - これは、iverilog+yosysの問題です
- テストモジュールは次のようにしてください

```
module test; // 名前はどうでもよい
reg in; // testに渡す(testの入力)はreg
wire ou; // testから受ける(testの出力)はwire
// login in, ou; としないこと
test test(in, ou);
begin
…// お馴染みのいろいろ
in = 1; // inをいろいろ変化させる
…// いろいろと試す
end
endmodule
```

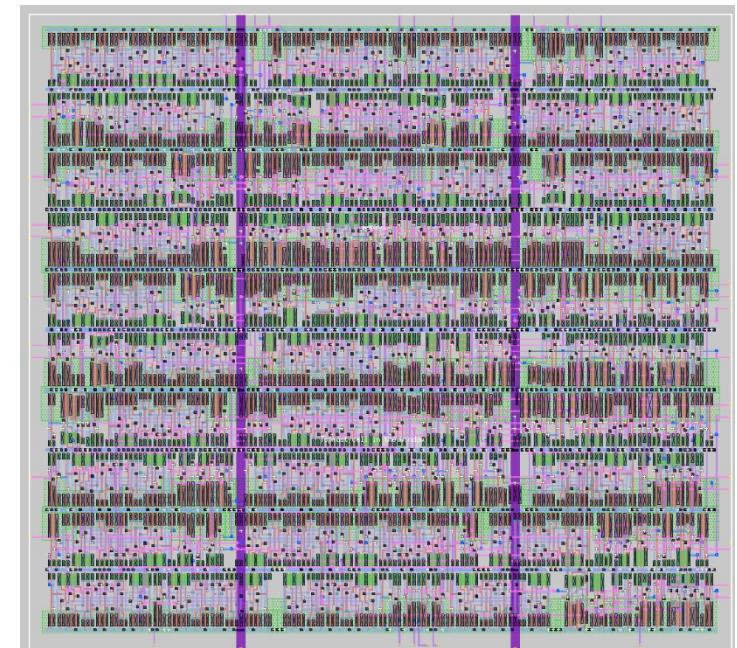
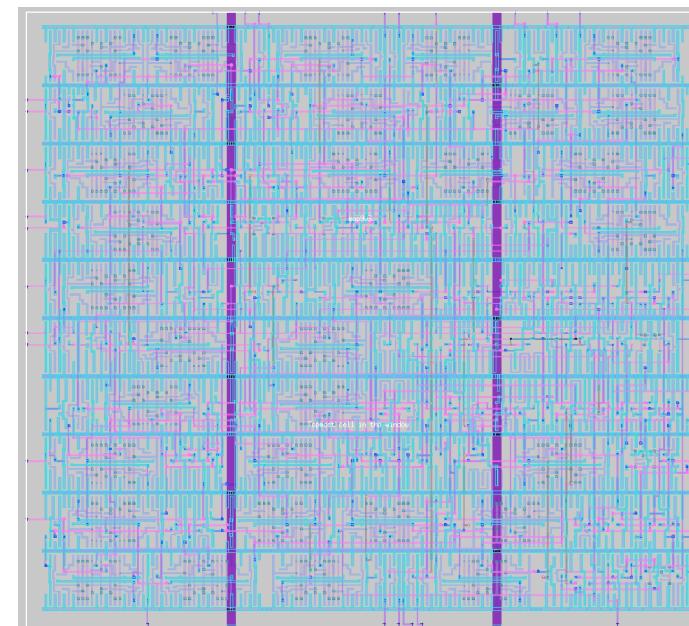
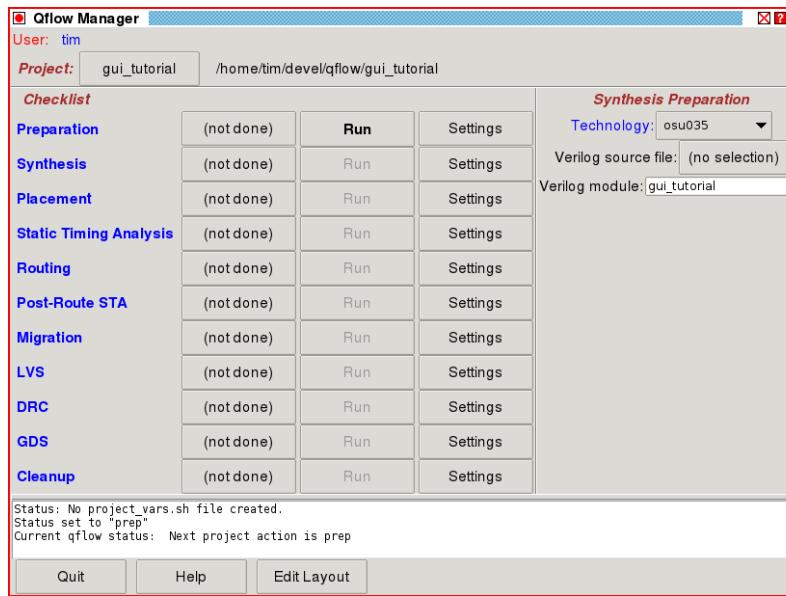




より本格的にやるには

- Qflow

- <http://opencircuitdesign.com/qflow/download.html>
- オープンソースだけで頑張っているEDAツールセット
- Linuxなら、それなりに合成から配置配線、spiceまでできる。現在Qflow1.3
- Verilogの合成、配置配線、spiceシミュレーションなどができる
- もちろん、商用と比較すればおもちゃ





演習問題（7-1）

166

次のソースコードは「合成する上で」問題のある記述である。問題の箇所を全て適切に修正しなさい。

なお、シミュレーションは動作し、合成も一応最後まで通りますが（設定やバージョンによっては通らない可能性もあります）、その生成結果をシミュレーションしようとすると、散々な結果になっています。

```
// 次のmoduleの合成上の誤りを直せ
module ex(input a, output logic [1:0] s,
output logic b, input clk, input rst);
    logic [1:0] ns;
    always @(posedge clk) begin
        if(rst) begin
            s <= 0;
            b <= 0;
        end else begin
            s <= ns;
        end
    end
// 1へ続く
```

```
// 1からの続き
always @* begin
    case(s)
        2'b00:begin
            b = 1;
            ns = 2'b01;
        end
        2'b01:begin
            b = 0;
            if(a) ns = 2'b10;
            else ns = 2'b11;
        end
    end
// 2へ続く
```

```
// 2からの続き
2'b11:begin
    if(a) b = 1;
    else b = 0;
    ns = 2'b00;
end
endcase
end
Endmodule
// 以上
```





演習問題（7-2）

167

- ・ 演習問題（5）のステートマシンを合成し評価せよ
 - ・ 実践的な合成スクリプトの例を用いて合成する
 - ・ gate.vとテストモジュールとtypicalライブラリでシミュレーションを行う
 - ・ verilogソースでtimescaleをきちんと与えること
 - ・ 何かしらセルライブラリを利用すること
 - ・ 実際に遅延していることは確認してください
- ・ 発展演習問題
 - ・ 演習問題（5）でも示した通り、足し算を用いて合成することもできる
 - ・ 現在の投入金額cなどを準備して $c+=10$ や、 $c+=50$ などとして計算する方法
 - ・ ステートマシンと足し算実装で、yosysの実力がどの程度か実際に見てみること





演習問題（7-1）と（7-2）

168

- 注意事項

- レポートをMicrosoft Wordファイルで作成、LMSへ提出すること
- A4 3枚以内で作成し、最初にタイトルを「演習問題（7）」として記載し、名前と学籍番号も忘れずに記載すること
 - 問題それぞれを一つのレポートで提出、演習問題（7-1）は修正後のコードを貼り付けて、「どこが悪いのか」「なぜ悪いのか」を記述しなさい。
 - 演習問題（7-2）は、まずverilogソースを添付、シミュレーションを行い、動作している証拠としてgtkwaveの画面をキャプチャしてWordに張り付けなさい
 - yosysのstatコマンドで出力されるエリアサイズ(回路規模)情報も記載すること
 - 演習問題（7-2）は、gveditやxdotの画面もキャプチャしてWordに張り付けること
 - 設計で利用した配線は全てgtkwave上で表示しておくこと(clk, rstなどすべて)
- verilogソースコードをテキスト情報として貼り付けること（画像ではない）
- 提出締め切りはLMSを確認すること

