2022年4月8日

ハードウエア設計論

講義内容

ハードウエア設計論イントロ

VerilogHDLのインストール

池田　誠

TA3名：Wang、正田、福田が担当します

分からないという心の叫びはSLACKにどうぞ

質問は随時はSLACK　#2022s-ハードウエア設計論

講義資料等：　http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/

ユーザ名：HWD2022

パスワード：HardWareDesign

ハードウエア設計論

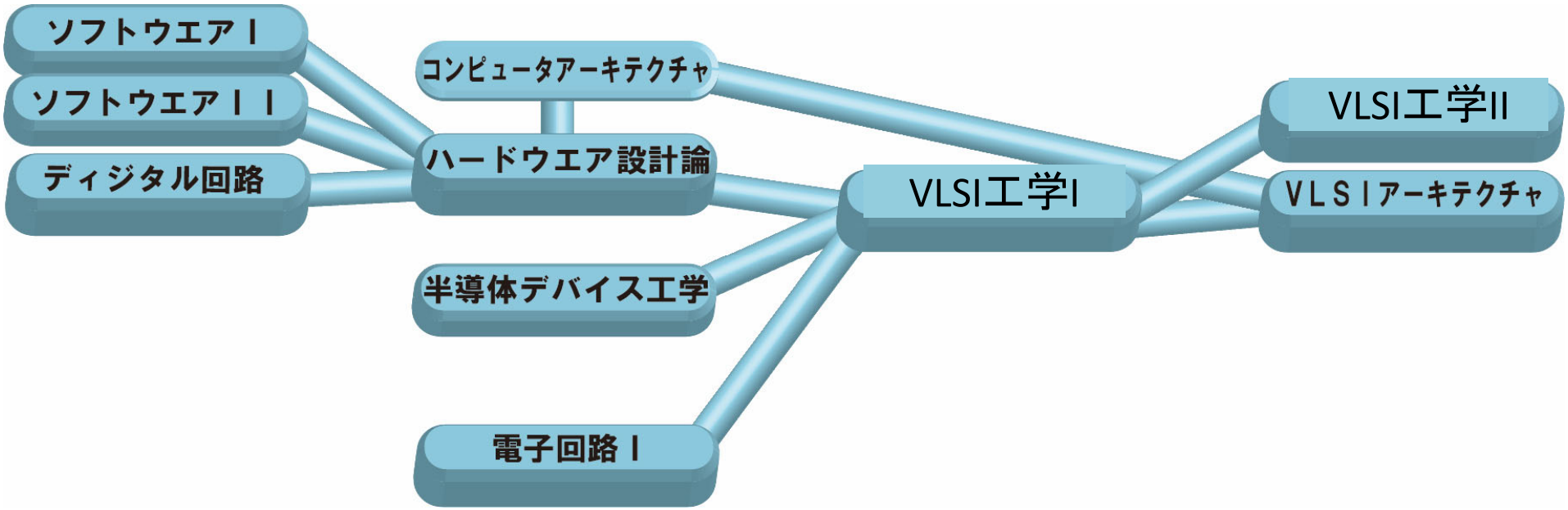
1	4月8日	イントロ：VerilogHDLの導入
2	4月15日	VerilogHDL..1 基本文法
3	4月22日	VerilogHDL..2 基本文法
4	5月6日	VerilogHDL..3 簡単な演習
5	5月13日	VerilogHDL..4 演習
6	5月20日	エミュレータを用いたハードウエア検証
7	5月27日	ハードウエアとテスト
8	6月3日	ハードウエアの自動設計
9	6月10日	C言語ベース設計
10	6月17日	休講の可能性
11	6月24日	ハードウェアとソフトウェア
12	7月1日	ハードウェアによる高速化
13	7月8日	組込みシステムとSoC設計

ハードウエア設計論の講義内容

- 現在電気を使うほとんどの機器に人知れず電子回路特にディジタル制御回路が含まれている。本講義では、それら組み込み機器のハードウエア設計を
  - 設計記述言語 VerilogHDLの習得を介した理解
  - システムレベル設計記述を介した理解
  - 記述言語からの設計の流れの理解
  - 設計検証・ハードウエアのテストの理解

を目指すものである

講義の体系


ソフトウエアⅠ
ソフトウエアⅠⅠ
ディジタル回路
コンピュータアーキテクチャ
ハードウエア設計論
半導体デバイス工学
電子回路Ⅰ
VLSI工学I
VLSI工学II
VLSIアーキテクチャ

本日の出欠は･･･

正式にはとりません･･ただ進捗確認のために以下の課題を行ってください。４月１５日までに終えてください。

WEBから課題１（課題１－１、課題１－２）を提出する。

使用する環境 / 本日の課題

- Ubuntu20.04(学科PCのバージョン) + iVerilog + gtkwave
- **困った場合には、、演習などのファイルは http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/ にあります**
- **本日の課題**
  - **Verilogのインストールと動作確認**
  - **簡単なVerilogの入力と動作確認**
  - **結果のUPLOAD（出欠を兼ねる）**
    upload出来ない場合には→手を挙げて申し出るor
    オンラインの場合チャットで連絡
    （電気系以外の学科の方は学籍番号　氏名をチャットでお知らせください）

使用する環境

- iVerilog + gtkwave
- **Ubuntu（Windows, MAC）のインストール方法は次ページ以降のとおり**
  - **Windows:インストールパスおよびユーザ名に日本語（多バイト文字）、スペースを絶対に入れないこと。**
- **講義の出欠を兼ねた演習課題の提出をしてもらいます。**
- **本日の課題**
  - **Verilogのインストールと動作確認**
    https://iverilog.fandom.com/wiki/Installation_Guide
    を参照ください
  - **簡単なVerilogの入力と動作確認**
  - **結果のUPLOAD　upload出来ない場合には**
    →チャットで連絡
    （電気系以外の学科の方は学籍番号　氏名をチャットでお知らせください）

Verilogのインストール: Ubuntu

- Linux(Ubuntu)を起動してログインしターミナルを開く
  % **sudo apt-get install verilog**
  % **sudo apt-get install nvidia-settings**
  % **sudo apt-get install gtkwave**
- パスワードを聞かれるので、自分のパスワードを入力
- インストールの実施の有無を聞かれるので **y**

Verilogのインストール: Windowsの場合

- http://bleyer.org/icarus/iverilog-v11-20210204-x64_setup.exe
  http://bleyer.org/icarus/にアクセスして、違うバージョン、OSを選択してもよい

  をダウンロードしてインストール
  ユーザ名や、インストール先のディレクトリー名にスペースや2byte文字を入れてはならない
  NG : C:¥Program Files¥iVerilog
  NG : C:¥ハードウエア設計論¥iVerilog
  OK : C:¥iVerilog
  OK : C:¥Tools¥iVerilog

  OK : ユーザ名　mikeda
  NG : ユーザ名　Makoto Ikeda
  NG : ユーザ名　池田
- インストールが終了したら、インストールの確認
  - コマンドプロンプトを開いて「iverilog」とコマンド入力。Usageが表示されれば第一段階OK
  - ただし iverilog　test.vと打って出力が出ない場合には上述のスペース、2バイト文字の制約に引っかかっている可能性が大
  - コマンドプロンプトを開いて「gtkwave」とコマンド入力。GUIが起動すればOKです。

Verilogのインストール: MACの場合（更新）

MACの場合開発環境をインストールする必要があります。すでに意識的に（or無意識に）インストールしているはいいのですが、そうではない場合には、Homebrew, xcode, xquarts, switchなどのパッケージをインストールする必要があります。

- Brewのインストール
  $ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
  passwordを聞かれたらMacのパスワードを打ってしばらく待つ
- xcodeのインストール
  https://developer.apple.com/jp/xcode/resources/ にアクセスしXcode11をインストール
- XQuartsのインストール（gtkwaveがエラーを出す場合）
  brew cask install xquartz
- Switchのインストール
  cpan install Switch (もしくは sudo cpan install Switch)
- iVerilogのインストール
  brew install icarus-Verilog
- gtkwaveのインストール
  brew cask install gtkwave

- gtkwaveと打って起動せず/Applications/gtkwave.app/Contents/Resources/bin/gtkwaveと打つと起動する場合には、以下の設定が必要（本当はインストールが正常なら不要なはず）
  PATHに/Applications/gtkwave.app/Contents/Resources/binを追加するか、aliasを設定する
  - SHELLとしてbashを使用している場合(zshの場合には ~/.zshrc)
    ~/.bashrcに
    alias gktwave = /Applications/gtkwave.app/Contents/Resources/bin/gtkwave
    もしくは
    export “PATH=/Applications/gtkwave.app/Contents/Resources/bin/:$PATH”
    を追加する。
  - SHELLとしてzshを使用している場合

不幸にしてcommand not foundになった場合 ~/.bashrc (or ~/.zshrc) を
/usr/bin/open ~/.***** で開いて編集しなおしてください

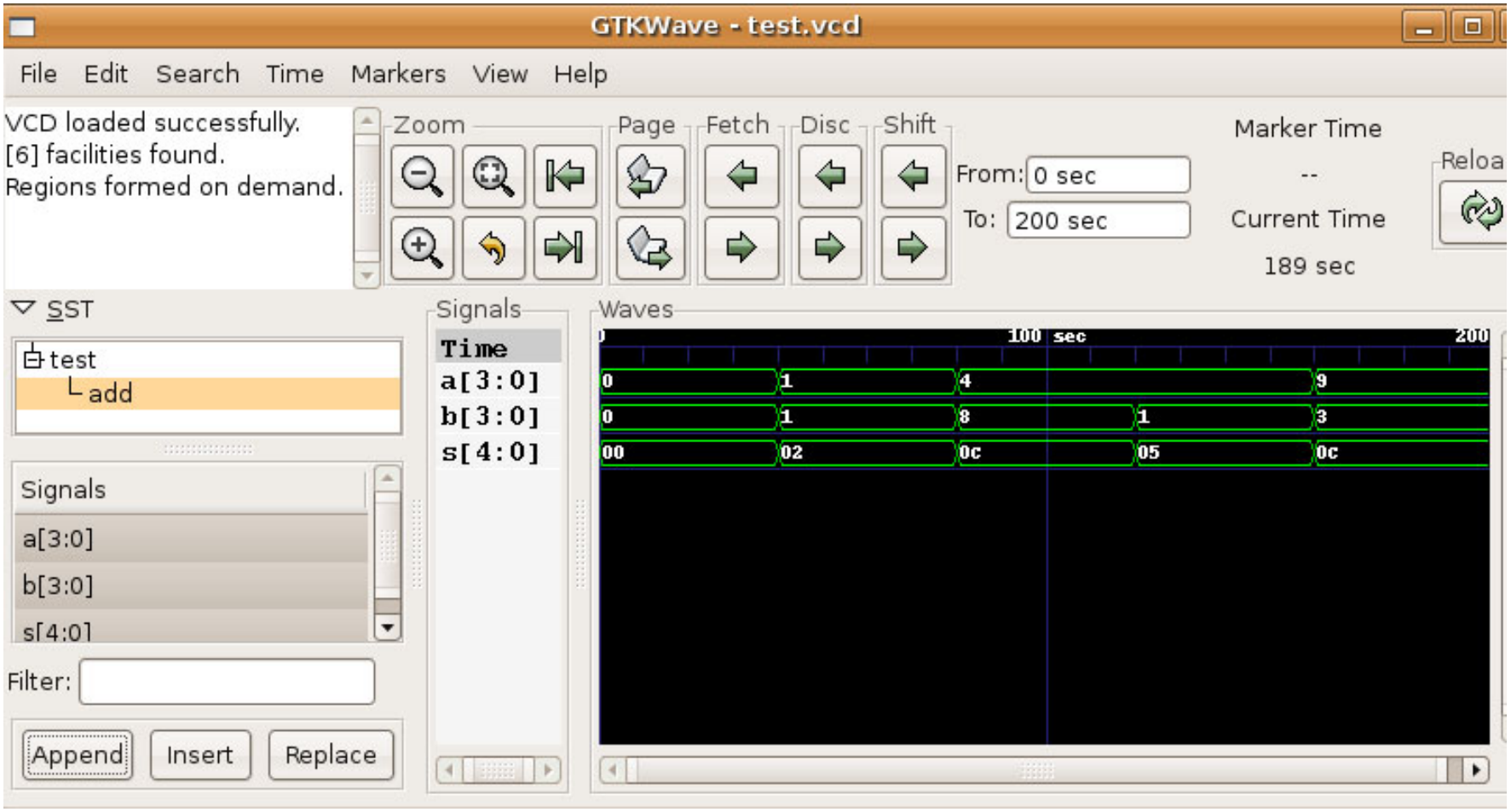
Verilogの実行確認

- http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/test.v をダウンロード
  % **iverilog　test.v**
  % **./a.out**
- 以下のように表示されればOK

```
0   0000 + 0000 = 00000   ( 0 +  0 =  0)
40   0001 + 0001 = 00010   ( 1 +  1 =  2)
80   0100 + 1000 = 01100   ( 4 +  8 = 12)
120   0100 + 0001 = 00101   ( 4 +  1 =  5)
160   1001 + 0011 = 01100   ( 9 +  3 = 12)
200   1101 + 1101 = 11010   (13 + 13 = 26)
```

Verilogの実行結果のGUIでの確認

- % **gtkwave　test.vcd**



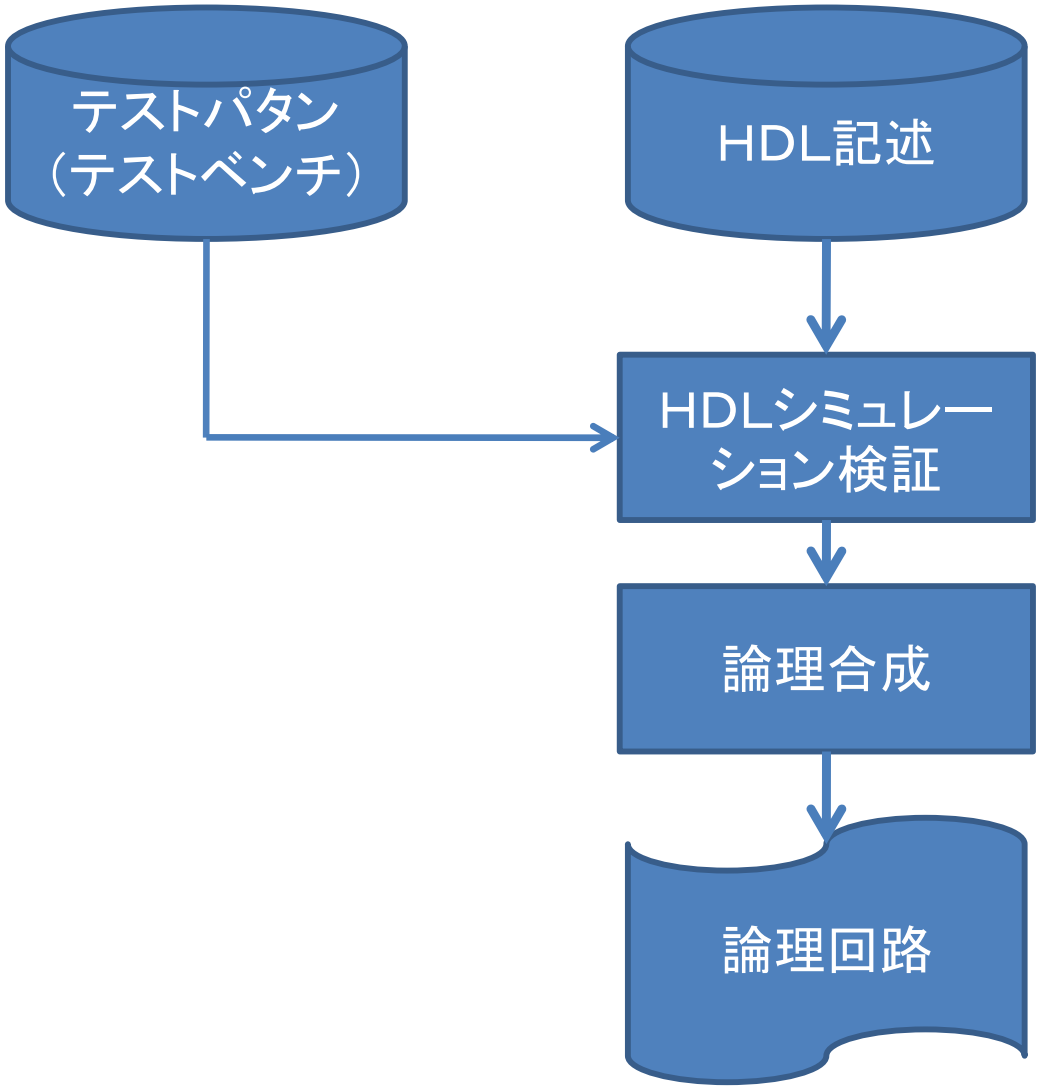
ハードウエア記述言語

言語	VHDL	VerilogHDL	UDL／I	SFL
開発開始時期	1981	1984	1987	1981
開発組織	IEEE	Cadence	電子協	NTT
言語仕様公表	1987	1985	1990	1985
論理シミュレータ	有	有	有	有
論理合成系	有	有	有	有
規格の見直し	1993,2000, 2002, 2008	IEEE 1364/ 1995,2001, 2005 SystemVerilog IEEE 1800-2005, IEEE/IEC 62530-2011	1992	なし

‥なぜハードウエア記述言語‥

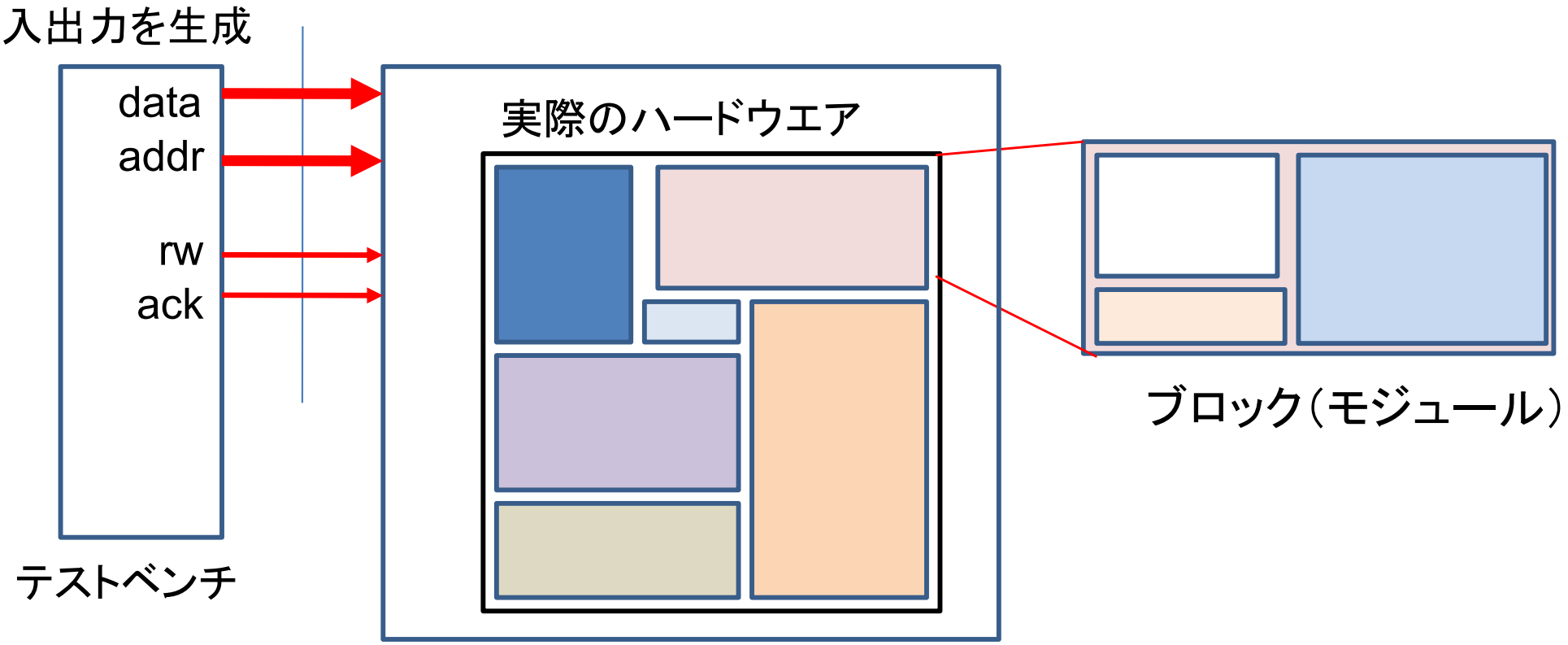
- 言語の文法そのものは、ほぼソフト
- 大きな違いは「時間」の概念
  - すべては同時に動作する（ステップごとではない）
  - 時間を制御することが出来る
  - 変数が、記憶（不揮発性）と揮発性（配線）に区別
- 並列ソフトウエアと本質的には同じ
  - VerilogHDLのオリジンはProlog(並列化志向の言語）

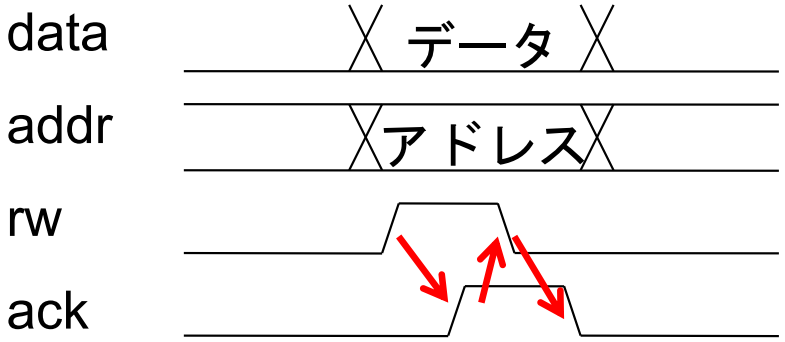
HDLによる設計



- 抽象度の高い記述による記述量の削減と設計効率向上
- ゲートレベル設計の自動化
- 設計の早期からのシミュレーションによる検証
- 制御回路の自動生成

HDLの基本





入出力信号とタイミング

VerilogHDLの基本構文

module　*モジュール名*　(*ポート名*, *ポート名*, ・・・　);
モジュールの入出力の宣言（全ポート名を宣言する）;
モジュール内信号の宣言（暗黙の定義は出来るだけ避ける）;
回路・機能の定義 ;
endmodule

モジュールとは「機能」もしくは「構造」のまとまり。プログラミングの関数*のようなもの。全ての記述は module ～ endmoduleで囲う。
構文は セミコロン ; により閉じる
複数構文にまたがる場合には　begin ～ endで囲う。
入出力の定義は、 入力　 input, 出力 output, 双方向 inoutにより定義
例:

```
module   test ( inA, inB, outC );
  input  inA, inB;
  output  outC;
endmodule
```

*ただしmodule内に function, taskといった手続きを記述することが可能であるのでmoduleを「関数」と表現するのは適切ではない

VerilogHDLの基本構文

add4.v

すべての要素は module – endmodule
ではさむ

入出力の宣言
（変数の宣言、代入、プロセス文の前におかなくてはならない）

継続代入文
（プロセス文中（後述）に入れてはならない）

```
module add4(s,a,b);
  output      [4:0] s;
  input[3:0] a,b;
  assign  s=a+b;
endmodule
```

いざ実行：

```
%   iverilog  add4.v
%   ./a.out
```

？？？

テストベンチが無い
（＝入出力が与えられていないので何も動かない）

テストベンチを作成

- シミュレーションを行うためには設計回路に入力を与えるためのプログラムが必要
  - テストベンチ
- 同じverilogで記述する

テストベンチ

testadd41.v

```
module testadd4;
  wire          [4:0] s;
  reg           [3:0] a, b;
initial begin
    $monitor( “%t %b + %b = %b”, $time, a, b, s);
    a <= 0;   b <= 0;
    #40   a <= 1;   b <= 3;
    #40   a <= 4;   b <= 8;
    #40   a <= $random;   b <= $random;
    #40   a <= $random;   b <= $random;
    #40
    $finish;
end
add4   add ( s,a,b );
endmodule
```

システムタスク:
$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
$finish シミュレーション終了

被検証用モジュールの
呼び出し(インスタンス化宣言)
(関数呼び出しのようなもの)

インスタンス名(任意)

入出力信号名
(「定義順呼び出し」、「名前呼び出し」がある)ここは定義順呼び出し(moduleの宣言順に変数を記述)

モジュール名
module ***で定義された名前

テストベンチ

再度実行:

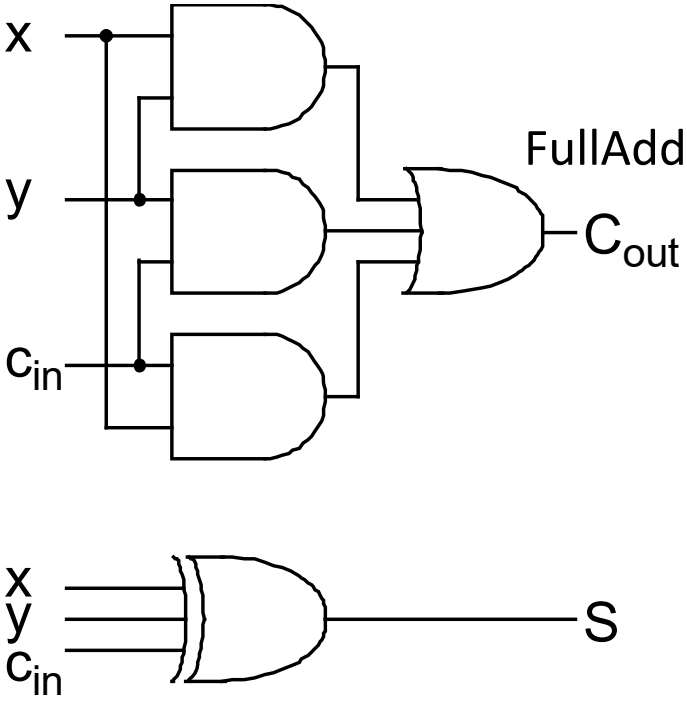
% **iverilog testadd41.v add4.v**

% **./a.out**

```
  0 0000 + 0000 = 00000
 40 0001 + 0011 = 00100
 80 0100 + 1000 = 01100
120 0100 + 0001 = 00101
160 1001 + 0011 = 01100
```

名称	記号	定義	優先順位	名称	記号	定義	優先順位
算術演算	+	加算	3	論理演算	!	論理値のNOT	1
	-	減算	3		&&	論理値のAND	9
	*	乗算	2		||	論理値のOR	10
	/	除算	2	等号演算	==	論理等号	6
	%	剰余	2		!=	論理不等号	6
ビット演算	~	ビット毎の反転	1		===	ケース等号(X,Zも一致)	6
	&	ビット毎のAND	7		!==	ケース不等号(X,Zも不一致)	6
	|	ビット毎のOR	8	関係演算	<	小なり	5
	^	ビット毎のExOR	7		<=	小なりイコール	5
	~^	ビット毎のExNOR	7		>	大なり	5
リダクション演算(単項演算)	&	各桁ビットのAND	1		=>	大なりイコール	5
	~&	各桁ビットのNAND	1	シフト演算	<<	右オペランド分左シフト(空いたビットは0)	4
	|	各桁ビットのOR	1		>>	右オペランド分右シフト(空いたビットは0)	4
	~|	各桁ビットのNOR	1	条件演算	? :	条件 ? 真の場合 : 偽の場合	11
	^	各桁ビットのExOR	1				
	~^	各桁ビットのExNOR	1				

簡単な論理式を実現してみよう



左図の全加算器を実現してみる:

FullAdderFunction.v

```
module  FullAdderFunction ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;

assign   cout  = (x & y ) | ( x & cin )  | (y & cin ) ;
assign   s = x ^ y ^ cin ;
endmodule
```

論理式やプロセス文、手続き文を用いた記述を
「動作記述」と呼ぶ

Verilogで定義される論理値と定数

取りうる値:

0: Low (論理 0)

1: High (論理 1)

x: 不定値:0か1か不定であるがどちらかの値を取る

z: High Impedance:0でも1でもない(定義されない値)

定数:

<ビット幅>'<基数><数値>として表す

<基数>: b, B: 2進数、o,O: 8進数, d,D: 10進数, h,H: 16進数

(例)

表記	基数	ビット幅	二進数表記
8	10	32bit	0000.....01000
4'd5	10	4bit	0101
1'b0	2	1bit	0
16'h 0f0f	16	16bit	0000111100001111
4'bx	2	4bit	xxxx

Verilogで定義される型

ネット型: 配線を表す。信号の論理値は接続されるノードの値として決定される: 組み合わせ回路の「値」として用いる

→ 単なる配線であるため、何らかの演算結果が「接続」されているだけであり、代入操作としては接続、つまり assign 文のみが使用可能

レジスタ型: レジスタ(記憶素子:いわゆるプログラミング的な変数)。信号の論理値が保持される: 順序機械の状態として用いる

→ レベルを保持するラッチやフリップフロップに相当。always文, initial文, function, taskの中での手続き代入操作のみが可能。(assignは出来ない)

定義可能な型の種類

ネット型

- wire型：
  - 継続的代入されているときのみ値を保持する型。一般の配線と同様。通常は「組み合わせ論理部（組み合わせ回路）」を表現するのに使用
  - 1bitのwire型は定義を省略可能：ただしこの暗黙の定義は使わない方が賢明

レジスタ型

- reg型：
  - 任意ビット、記憶保持が可能な型（通常はFFなど状態、データを保存したいノードに対して使用）, signedを指定しない場合には符号なし
- integer型
  - 32bit幅の符号付き整数型
- real型(実数), time型(符号無し64ビット), realtime型(実数表記での時間)

- signed指定
  - reg signed [7:0] a;　　aを符号付きレジスタ型として定義
  - wire signed [7:0] a;　aを符号付きwire型として定義
- バス幅の定義
  - reg [7:0] aなど。a[0] からa[7]の8ビット幅として定義。降順
- アレイの定義
  - reg a[0:31]など。0から31番地までを確保。昇順

式・数の表現

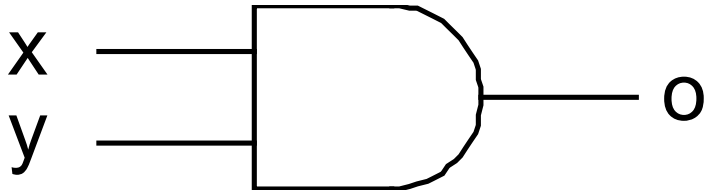
- 連接
  - {式1, 式2}：式1, 式2をつなげる {0101, 1100} → 01011100
  - {定数式 {式, 式} }：{}内を定数式の値だけ繰り返す { 5 {10} } → 1010101010
- レンジ式
  - [定数1:定数2]：a[6:3] → a[6], a[5], a[4], a[3]
  - [式+:定数]： a[P*8+:4] → P=0の時a[3:0], P=1の時 a[11:8]
  - [式-:定数]：a[P*8-:4] → P=1の時 a[8:5], P=2の時 a[16:13]

VerilogHDLの基本構文：構造記述と組み合わせ回路

回路の定義:
他のモジュールを呼び出す記述：**構造記述**
（ちょうど回路図を書いているようなもの）

モジュール名　　インスタンス名　（ポート）
モジュール名：呼び出すモジュールの名前
インスタンス名：任意（呼び出しの名前）:変数や他のインスタンス名と重複してはならない
ポート:
　定義順呼び出し:
　　ポート定義順に信号を記述
　名前呼び出し:　module (a, b, c); の場合
　　.a(sigA), .b(sigB), .c(sigC) と記述すること
で記述順は関係なくなる

```
module and2 (x, y, o);
  input  x, y;
  output   o;
  and   a1 ( o, x, y );
endmodule
```



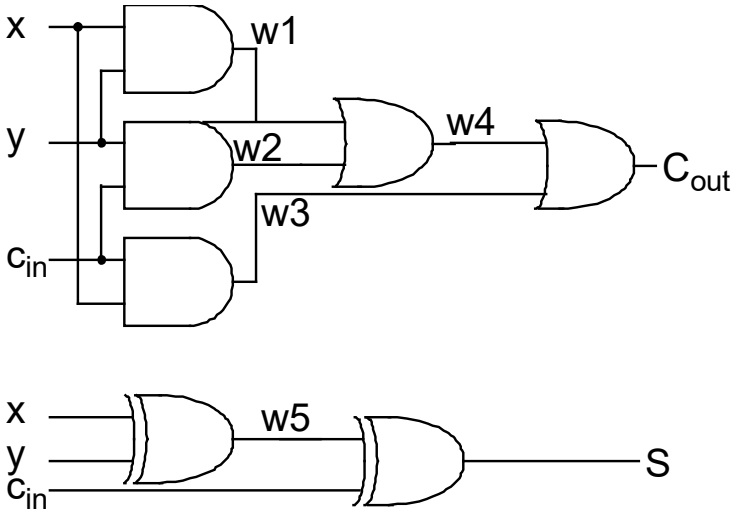
Verilogで定義されるプリミティブゲート

Verilog-HDLにあらかじめ組み込まれているゲート。module定義なしに使用可能。
通常、ポートは 出力、入力、イネーブルの順となっている

(プリミティブゲートも moduleとして定義されている)

種別	ゲート名	出力	入力	イネーブル	機能
1入力ゲート	buf, not	OUT	IN		
2入力ゲート	and, nand, nor, or, xor, xnor	OUT	IN1, IN2		
3state	bufif0, bufif1, notif0, notif1	OUT	DATA	CONTROL	buf: バッファ, not: 論理反転, if0: control=0で出力, if1: CONTROL=1で出力
switch	nmos, pmos, cmos, rnmos, rpmos, rcmos, tran, tranif0, tranif1, rtran, rtranif0, rtranif1				
pullup, pulldown	pullup, pulldown				

簡単な回路図を実現してみよう



左図の全加算器を構造記述により実現してみる:

FullAdderStructure.v

```
module  FullAdderStructure ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;
wire   w1, w2, w3, w4, w5;

and   a1 ( w1, x, y );
and   a2 ( w2, y, cin );
and   a3 ( w3, cin, x );
or     o1 ( w4, w1, w2 );
or     o2 ( cout, w4, w3 );
xor    x1 ( w5, x, y );
xor    x2 ( s, w5, cin );
endmodule
```

課題１

- 課題１－１：動作記述（論理式）バージョン
  module FullAdderFunction ()
  を完成させ、simfulladd.vを作成して、シミュレーション結果を確認する
- 課題１－２：構造記述バージョン
  module FullAdderStructure ()
  を完成させ、simfulladd.vを作成して、シミュレーション結果を確認する
- FullAdderFunction.v, FullAdderStructure.vを
  http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/
  から提出

simfulladd.v

課題１のテストベンチ

```
module simfulladd;
  wire          s, cout;
  reg           x, y, cin;
  initial begin
        $monitor( "%t In (x, y, cin) -> Out (s, cout): (%b, %b, %b) -> (%b, %b)", $time, x, y, cin, s, cout);
        x <= 0;   y <= 0; cin<=0;
        #40       x <= 1;   y <= 0; cin<=0;
        #40       x <= 0;   y <= 1; cin<=0;
                  ……. 入力全条件を検証すること
        #40       x <= 1;   y <= 1; cin<=1;
        #40
        $finish;
  end
  FullAdderFunction  add ( x, y, cin, cout, s );
endmodule
```

FullAdderStructureも同様