論理設計入門
演習ノート

清水尚彦

利用テキスト
清水尚彦著
コンピュータ設計の基礎知識
ハードウェア・アーキテクチャ・コンパイラの設計と実装
２００３年　共立出版

演習概要

- **システムLSI開発概論**

  要求仕様の策定から回路実装まで

- **論理回路基礎演習**
  - **組合せ回路**
  - **順序回路と状態遷移図**
  - **表現形式**
- **システムLSI開発演習**

組込みシステムにおける ハードウェア開発

- シリコン集積度の向上により、ハードウェア化の要求が高まる
- 組込みシステムへの要求が高度化するとソフトウェアだけでは対応できない
  - 日本システムハウス協会の調査によると組込みシステムの約半数のプロジェクトではハードウェアも開発している
- 高速なＣＰＵは大消費電力
  - 同じことをハードウェアで実現すれば省電力
  - ＣＰＵのクロックを下げ、電源電圧を低下させたい
    - 消費電力は $P=CV^2F$ なので、電源電圧低下はメリット大　電源低下による遅延の増大はクロック低下で補償

ハードウェア開発の敷居低下

- 再構成可能ＬＳＩ（ＦＰＧＡなど）の普及
  - ハードウェアでありながら、ユーザが書換え可能
  - 同一品種を大量生産するので、少量多品種の開発においてコストパフォーマンスが高い
    - 携帯基地局、ルータ、ＤＶＤレコーダ、液晶ディスプレイなど多くの製品に使われている
- ハードウェア記述言語による開発
- 論理シミュレーションの高速化
- ハードウェア・ソフトウェア協調設計の普及

設計の基本
エンジニアリングチョイス

- アーキテクトは複数の実現手法を組合せ最適設計を行う
- ソフトだけ、ハードだけでは競合力のある設計にはならない
- 組込みソフトウェア技術者はハードウェアの仕様書を読みこなす能力が要求される
  - ハードウェアの仕組みを深く知ることで仕様書の行間が見えてくる

ディジタル回路設計技術

- **大規模化への対応のため、ハードウェア記述言語が必須となっている**
  - EDA（Electric Design Automation）
  - LSI CAD（Computer Aided Design）
- **記述言語としては、次のものが多用される**
  - Verilog（Verify Logic**の略**）
    - **論理検証に特化して設計された言語**
  - VHDL（VHSIC Description Language）
    - **米国国防総省主導による仕様記述言語**
  - SystemC
    - C**言語ベースの検証言語**
  - SFL（Structured Functional description Language）
    - NTT**がコンピュータ開発のために設計した言語**

データ系と制御系

- 論理設計の手順として、データ系と制御系を分離設計することが要求される
  - LSI面積縮小、配線遅延低減、消費電力低減など多くの要求事項はデータ系の設計制約となる
  - データ系は転送経路ごとにデータ幅の配線もしくは回路が要求されるのに対し、制御系は通常1本の信号でデータ系の制御を行う

設計の流れ

- 要求分析
  - 何を作るかを明確にする
- アーキテクチャ設計
  - どう作るかを記述する
- システム設計
  - アーキテクチャをテクノロジにマッピング
- 検証
  - 論理シミュレーションによりコンピュータ上で確認
- 実装・実機テスト
  - 本演習では(FPGA: Field Programmable Gate Array)に回路を実現する

論理設計入門 演習ノートテキスト P.107

要求分析

開発システムを分析し、仕様詳細化

何を作るかを明確にしよう

– 分析不十分な開発は後工程への影響大

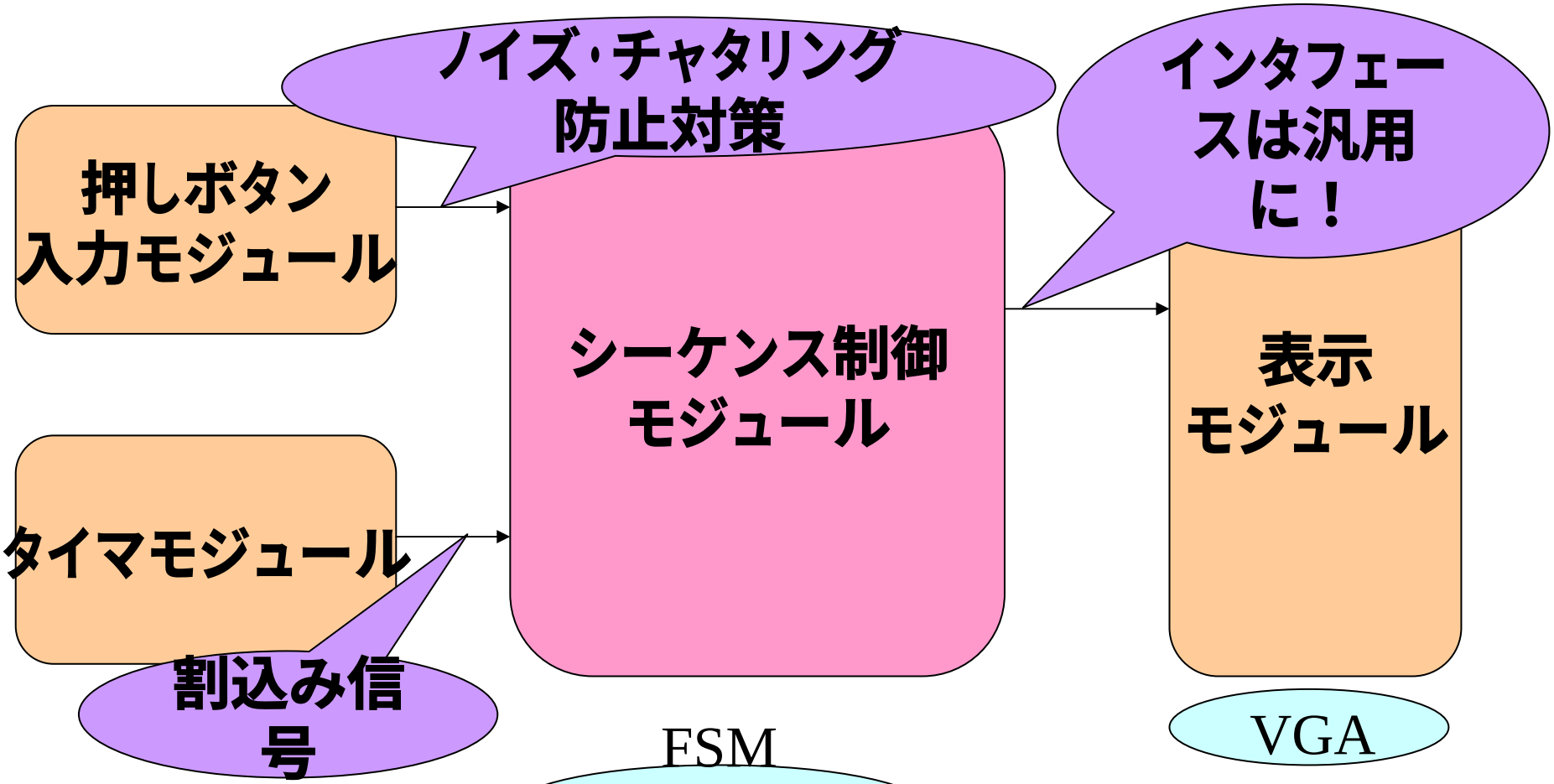
1. システムに必要な動作を抽出
   - 分析の開始時点では、過度の詳細化は避ける
   - 何を作るかを記述し、どう作るかは設計に回す
2. システムで発生するイベントを抽出
   - イベントをリストアップし、視覚化する

アーキテクチャ設計

- 要求分析結果に対して「どのように作るか」を設計

1. システム構成要素抽出
2. システム入出力、ユーザインタフェース定義
3. 構成要素間インタフェース定義

構成要素のモジュール分割と要素技術の選択の検討例



システム設計ならびに論理エントリ

- データの流れを整理しデータパスを設計
- 制御の流れを整理し状態遷移図を設計
- データパス上のデータ移動タイミングをタイミングチャートを作成し確認

- ハードウェア記述言語を用いて各要素を記述

組み合わせ回路のＨＤＬ記述

- 記述言語によって記述方法は異なる

Verilog

```
module comb (b , a , f );
input b;
input a;
output f;
   assign f = a&b;
endmodule
```

組合せ回路は
ブール代数式で記述

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity comb is  port(
      f: out std_logic;
      b: in std_logic;
      a: in std_logic);
end comb;
architecture RTL of comb is
begin
 f <= a and b;
end RTL;
```

SFL

```
module comb {
 input a,b;
 output f;
   f=a&b;
}
```

レジスタのＨＤＬ記述

- 記述言語によって記述方法は異なる

Verilog

```
module regi (m_clock , a , f );
 input m_clock;
 reg r;
 input a;
 output f;
   assign f = r;
always @(posedge m_clock  )
  begin
  r <= a;
end
endmodule
```

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity regi is
        port(m_clock: in std_logic;
        f: out std_logic; a: in std_logic);
end regi;
architecture RTL of regi is
   signal r: std_logic;
begin
  f <= r;
p_0: process(m_clock)
begin
if m_clock'event and m_clock='1' then
        r <= a;
end if;
end process;
end RTL;
```

SFL

```
module regi {
 input a;
 output f;
 reg r;
 par {
     r := a;
       f = r;
       }
}
```

特定の記述からレジスタを推定

状態遷移のＨＤＬ記述 (抜粋)

- 記述言語によって記述方法は異なる

Verilog

```
parameter _state_sg__st0 = 0;
 parameter _state_sg__st1 = 1;
 parameter _state_sg__st2 = 2;
…
assign _net_2 = ((_stage_sg_state_reg)==(_state_sg__st2))&_stage_sg;
assign _net_1 = ((_stage_sg_state_reg)==(_state_sg__st1))&_stage_sg;
assign _net_0 = ((_stage_sg_state_reg)==(_state_sg__st0))&_stage_sg;
…
always @(posedge m_clock or posedge p_reset)
  begin
if (p_reset)
        _stage_sg_state_reg <= _state_sg__st0;
else if (_net_2)
        _stage_sg_state_reg <= _state_sg__st0;
else if (_net_1)
        _stage_sg_state_reg <= _state_sg__st2;
else if (_net_0)
        _stage_sg_state_reg <= _state_sg__st1;
end
```

VHDL

```
constant v_state_sg_v_st0: std_logic_vector(1 downto 0) := 0;
 constant v_state_sg_v_st1: std_logic_vector(1 downto 0) := 1;
 constant v_state_sg_v_st2: std_logic_vector(1 downto 0) := 2;
…
v_net_5 <= (v_net_4) and v_stage_sg;
 v_net_4 <= '1' when ((v_stage_sg_state_reg) = (v_state_sg_v_st2))
else '0' ;
 v_net_3 <= (v_net_2) and v_stage_sg;
 v_net_2 <= '1' when ((v_stage_sg_state_reg) = (v_state_sg_v_st1))
else '0' ;
 v_net_1 <= (v_net_0) and v_stage_sg;
 v_net_0 <= '1' when ((v_stage_sg_state_reg) = (v_state_sg_v_st0))
else '0' ;
p_0: process(m_clock)
begin
if m_clock'event and m_clock='1' then
  if p_reset='1' then
       v_stage_sg_state_reg <= v_state_sg_v_st0;
  elsif (v_net_5)='1' then
       v_stage_sg_state_reg <= v_state_sg_v_st0;
  elsif (v_net_3)='1' then
       v_stage_sg_state_reg <= v_state_sg_v_st2;
  elsif (v_net_1)='1' then
       v_stage_sg_state_reg <= v_state_sg_v_st1;
  else  v_stage_sg_state_reg <= "00";
  end if;
end if;
end process;
```

SFL

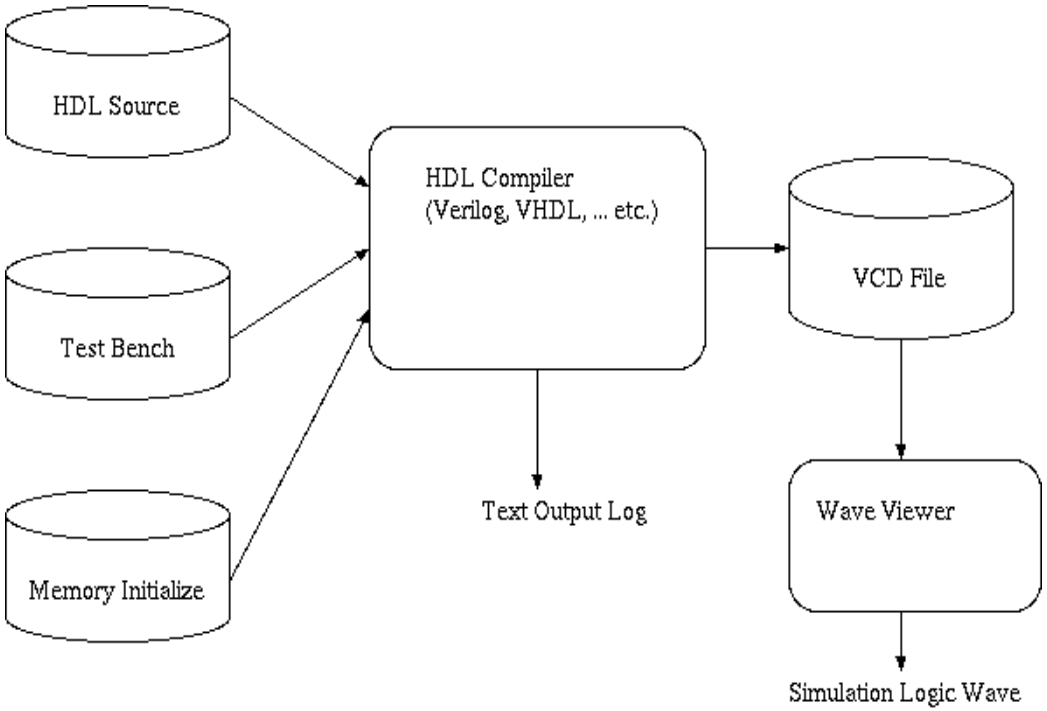
```
module sttest {
 stage_name sg { task t1(); }

 stage sg {
  state_name st0,st1,st2;
  first_state st0;
  state st0 goto st1;
  state st1 goto st2;
  state st2 goto st0;
  }
}
```

状態レジスタと状態変数の値を定義し
遷移条件を見て状態レジスタを更新

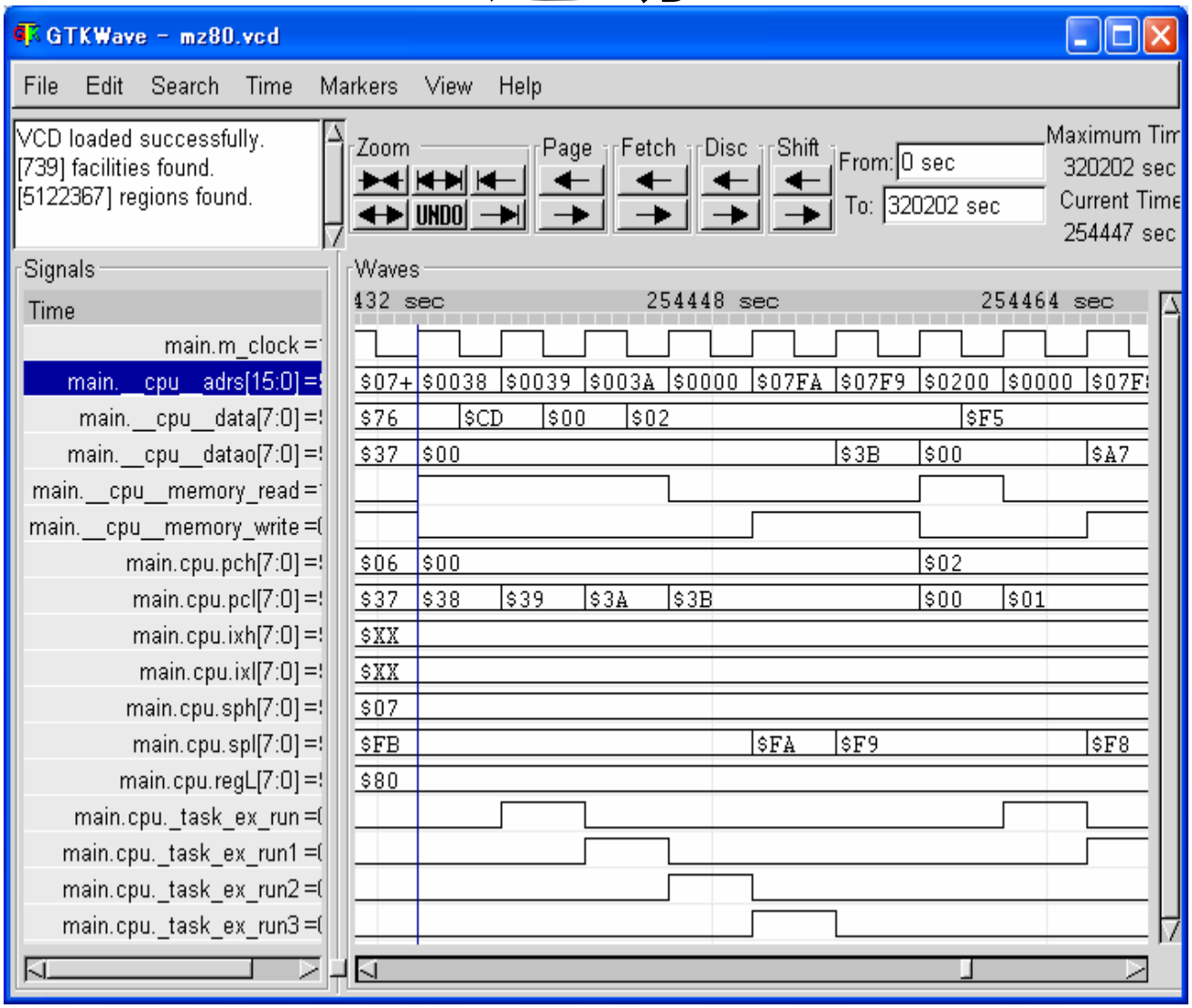
論理シミュレーションによる検証

- HDLコンパイラにHDLソースコード、検証スクリプト(テストベンチ)、(必要に応じて)メモリ初期化ファイルを与え、シミュレーション実施
  - 波形出力有無
  - テキストログ選択
  - 停止条件

  をテストベンチで記述

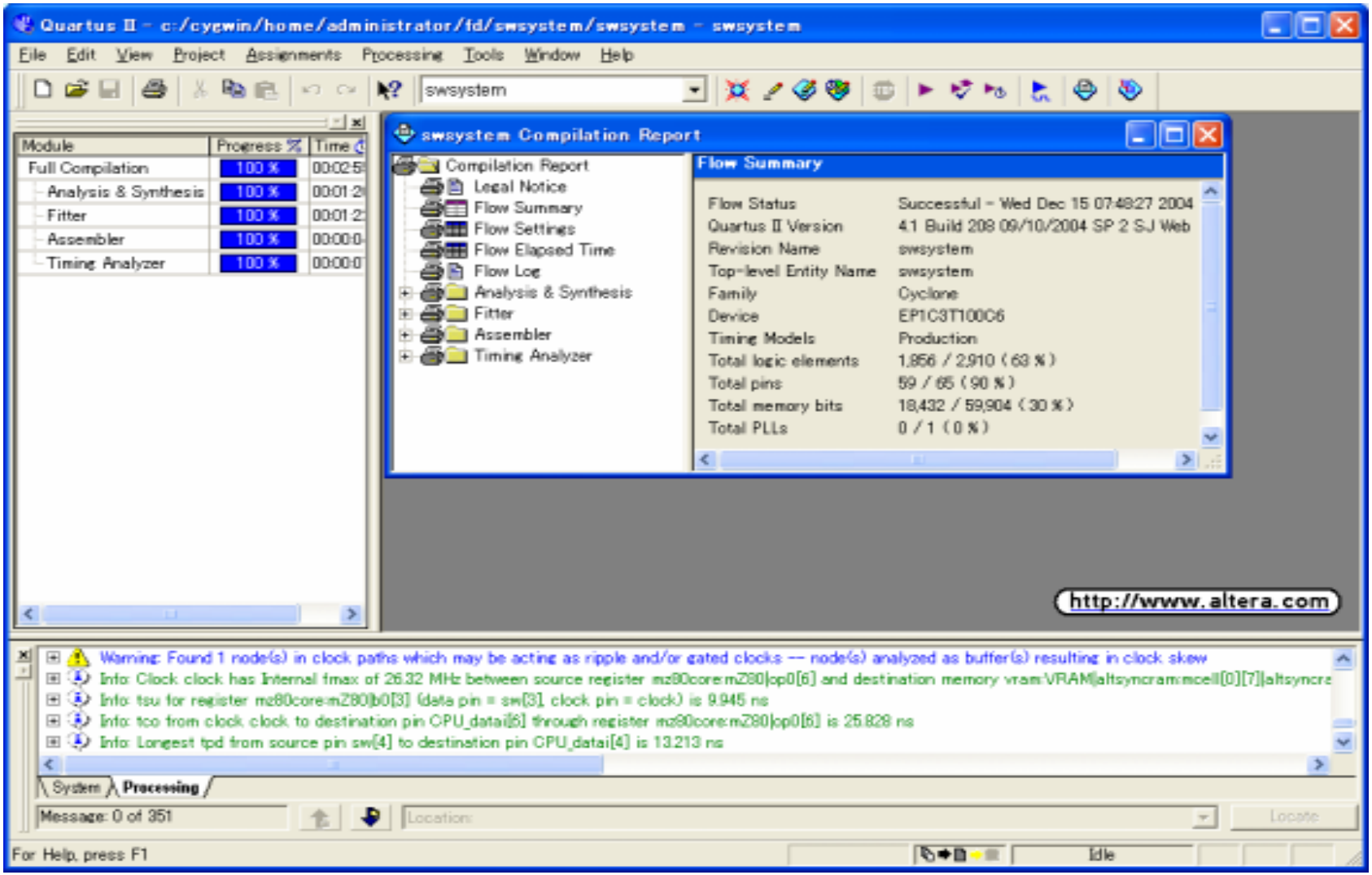


CPU連動シミュレーション

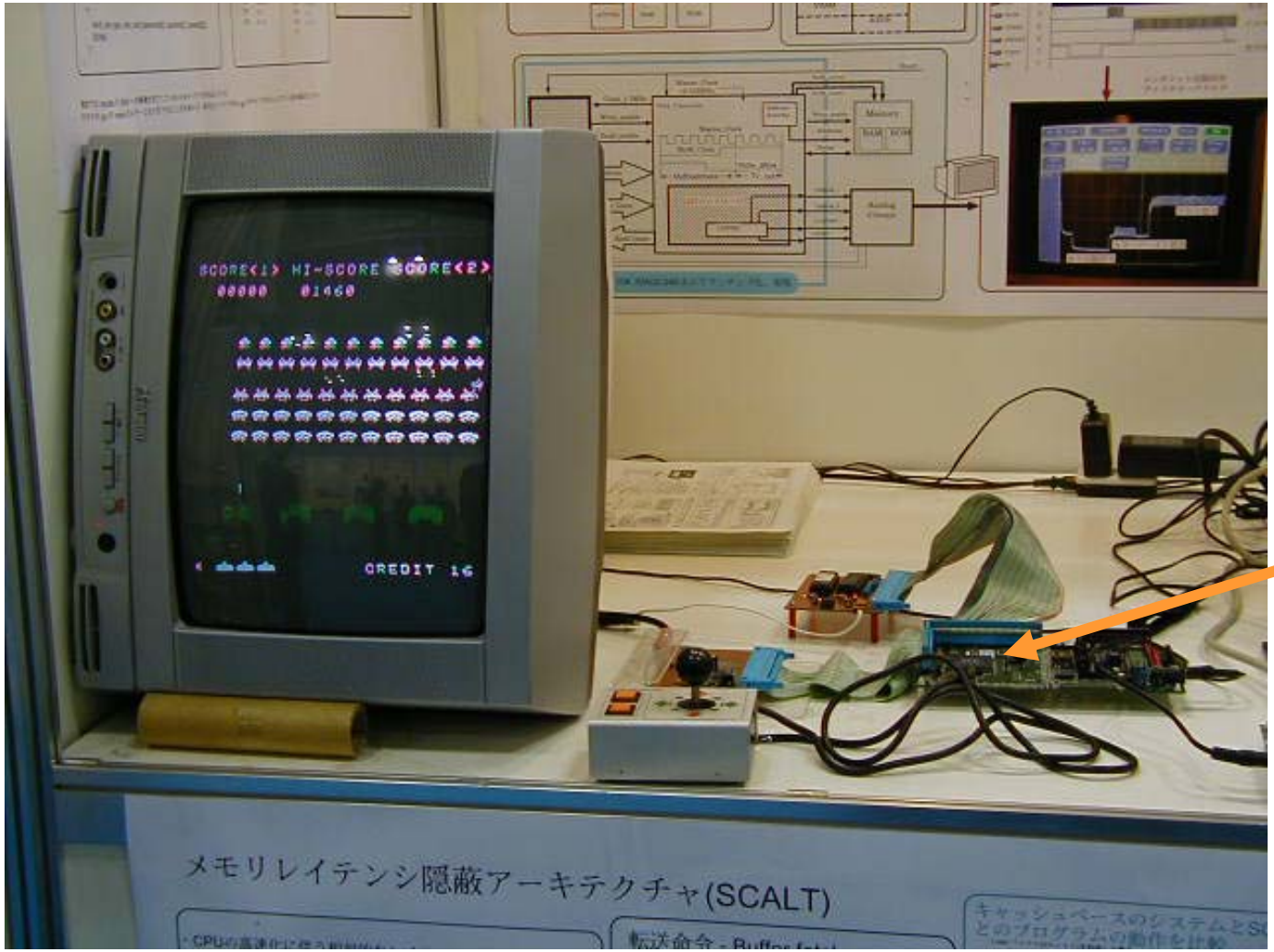


検証用のソフトを作成し、CPU連動シミュレーションを実施する

A社FPGAへのマッピング



FPGAを用いた電子回路実装



写真のシステムは㈱タイトーよりプログラムROMの提供を受けています

演習概要

- システムLSI開発概論
- 論理回路基礎演習
  - 組合せ回路
  - 順序回路と状態遷移図
  - 表現形式
- システムLSI開発演習

組合せ回路と順序回路

- 組合せ回路は入力に対して一意に出力が決定される

- 順序回路は初期状態、入力の履歴とその時点の入力値が出力を決定する
  - 回路が状態を持つと呼ぶ
  - 状態を保持するために記憶素子が用いられる
    - フリップフロップ (FF)・レジスタ・ラッチ・メモリ
    - flip flop, register, latch, memory

演習概要

- システムLSI開発概論
- 論理回路基礎演習
  - 組合せ回路
  - 順序回路と状態遷移図
  - 表現形式
- システムLSI開発演習

ブール変数とブール代数

- **ブール変数**：
  1または0の2値の値域を有する変数
- **ブール変数の値域に閉じる演算体系をブール代数と呼ぶ**
- **ブール代数演算**：
  論理積、論理和、否定

基本論理演算

- 組合せ回路に用いる基本論理演算
  - テキストP．91図4．3
    - 論理積（&） 入力が全て1のときに出力が1
    - 論理和（|） 入力のいずれか1つが1のときに出力が1
    - 否定（^） 1－入力値を出力
    - 排他的論理和（@） 入力信号中の1の数が奇数のとき1
  - 演算を表す回路記号（MIL記号）
  - 演算を表す演算子

演習問題　1

- **次のブール代数式において入力変数**a,b,c**のすべての値に対する出力変数**f**の値を表で表せ**

- (1) f = a & b
- (2) f = a & (b | c)
- (3) f = a | (^b & c)

(1)**の回答例**

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

入力信号
(ブール代数式右辺)

出力信号
(同左辺)

表形式でブール代数演算の入力と出力を記述した表を真理値表と呼ぶ

演習問題　2

次の真理値表があらわすブール代数式を導出せよ

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
*	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0

f(2)　出力1の行に注目

f(4)

f(5)

入力した値が出力変数に影響しない Don’t Care と呼ぶ

f=f(2)|f(4)|f(5)

テキストP．93の手続きを参照

ブール代数の定理

- **交換則**
  - A&B = B&A
  - A|B = B|A
- **分配則**
  - A|(B&C) = (A|B)&(A|C)
  - A&(B|C) = (A&B)|(A&C)
- **結合則**
  - (A&B)&C = A&(B&C)
  - (A|B)|C = A|(B|C)

ブール代数の定理 (続き)

- **吸収定理**
  - A&(A|B) = A
  - A|(A&B) = A
- **ド・モルガンの定理**
  - $\overline{f(\&,|,A,B,C,\cdots)} = f(|,\&,\overline{A},\overline{B},\overline{C},\cdots)$
  - **例**:
    - $\overline{A\&B \mid C} = (\overline{A} \mid \overline{B}) \ \& \ \overline{C}$

演習問題　3

- 二つの吸収定理の左辺のブール代数式の真理値表を作成し、吸収定理を確認せよ

- ド･モルガンの定理の例の右辺と左辺のそれぞれの真理値表を作成し、定理を確認せよ

演習問題　4

- 2桁の二進数と1桁の二進数を加算する。
  真理値表とブール代数式を作成せよ
  - 入力1：　A,B　　(Aが上位Bが下位のビット)
  - 入力2：　C
  - 出力：　S,T,U　　(Sが最上位、Uが最下位ビット)

  S,T,Uのそれぞれに対して独立して回路を作成
  真理値表は3出力をまとめて記述
  ＡＢＣ：ＳＴＵ

演習概要

- システムLSI開発概論
- 論理回路基礎演習
  - 組合せ回路
  - 順序回路と状態遷移図
  - 表現形式
- システムLSI開発演習

フリップフロップの動作

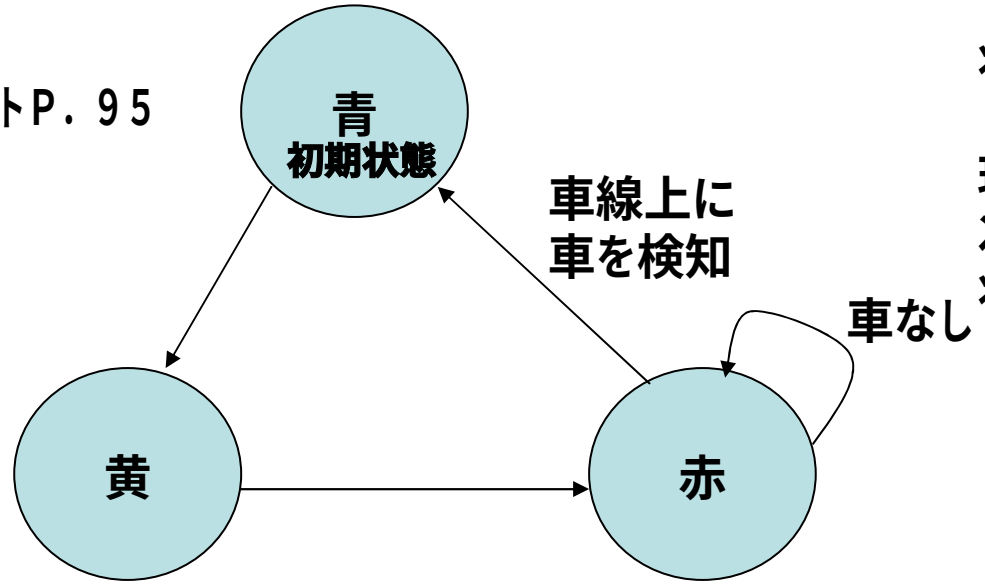
- クロック信号の立ち上がり（0から1への遷移）の直前のデータ入力（D）の値が記憶され、出力（Q）に出力される。
- フリップフロップは論理回路の状態ならびに情報を記憶する基本素子として用いる
  - 記憶のために使われるフリップフロップをレジスタと呼ぶ
- 記憶は正帰還を持つ回路、もしくは電荷により保たれる

テキストP．79、94

状態遷移図と状態遷移表

- 状態は○で表し、遷移を矢印で示す。条件がある遷移は矢印上に条件を示す

テキストP．９５



状態と遷移条件に名前をつける
(各名前に2進数を割り当て)
現状態、遷移条件を左に、
次状態、出力信号を右に
状態遷移図を表形式に記述

状態遷移の初期状態は
青とする。

状態をフリップフロップで記憶 (状態変数)

現状態と入力信号から出力および次状態を計算

状態遷移表と状態遷移マシン

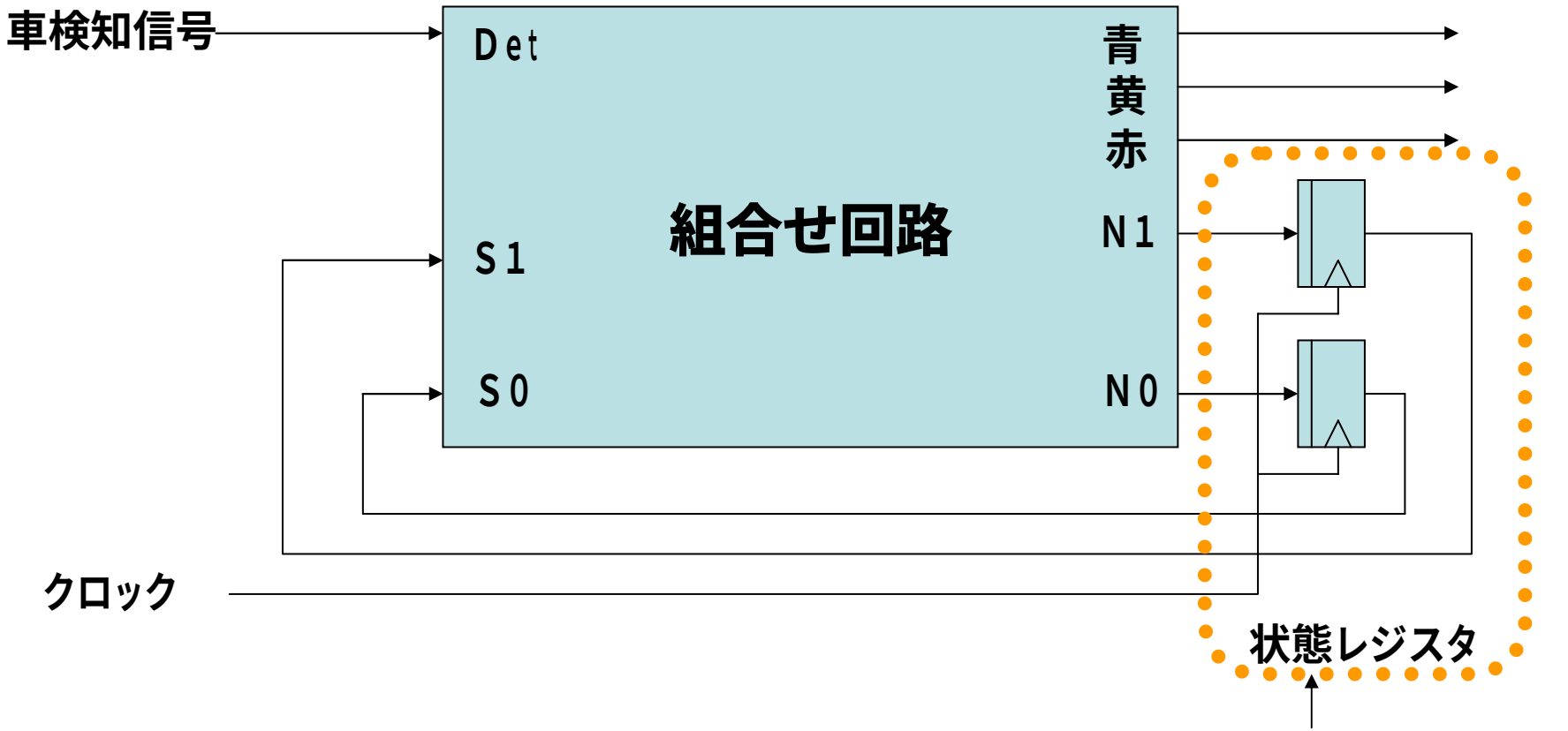
- 現状態を表す信号 S1, S0
- 次状態を表す信号 N1, N0　　テキストP．95，96
- 車の検知信号 Det
- 現状態をＦＦに保持
- 組合せ回路が次状態計算

状態変数への数値割当

状態名	割当	
青	0	0
黄	0	1
赤	1	0

	現状態		検知	次状態		
	S1	S0	Det	N1	N0	
青	0	0	＊	0	1	黄
黄	0	1	＊	1	0	赤
赤	1	0	0	1	0	赤
赤	1	0	1	0	0	青

状態遷移機械
(ＦＳＭ: Finite State Machine)



演習問題　5

本ＦＳＭの組合せ回路の出力をブール代数式で表せ

演習問題　6
押しボタン式信号制御回路を作成

- 使用する信号は次の通り
  - 自動車用信号はB,Y,R
  - 歩行者用信号はWb,Wr
  - 押しボタンはP
- 回路の状態は自分で分析し、状態遷移図を作成した上で状態数から必要なだけの状態信号を作成すること
  - 状態遷移表で参照しない入力はDon’t care(＊)とする
  - 状態遷移図が分岐するときには遷移表には分岐先ごと別々の行を作成する

状態遷移図で表せる回路

- 定型的な回路構成により、組み合わせ回路とフリップフロップを用いて実現可能

- ただし、入力や出力もしくは状態を記憶するフリップフロップの接続形態はさまざまなバリエーションが考えられる
- 例
  - 出力はすべてフリップフロップから出す
  - 入力はいったんフリップフロップに記憶する

状態遷移図では表しにくい回路

- 複数の部分に分割して考えると小さな状態遷移であっても、全体をまとめると状態数が膨大になるもの
  - 小さなブロックに分割して、ブロックごとの状態遷移を用いて設計する
  - 全体の動作に関しては別途検証が必要
- 多段階に処理が必要なもの
  - 処理Aの後で処理B、ついで処理Cを行う
  - 性能向上のため処理Aが終わったら、次のデータへの処理Aを引き続き実行する
    - パイプライン処理と呼ぶ

演習概要

- システムLSI開発概論
- 論理回路基礎演習
  - 組合せ回路
  - 順序回路と状態遷移図
  - 表現形式
- システムLSI開発演習

整数表現（テキストP．１０８）

- 数学用語の整数とコンピュータ用語の整数は意味が異なる
  - 数学の整数は無限の値域を持つが、コンピュータは有限値域
  - よく使われる整数表現に符号なし二進数（4．2式）と、2の補数表記符号付二進数（4．3式）がある

演習問題　7

- テキストP．１１０　例題4．2（1）を4．2式より導出せよ
- 4ビットの2の補数表記の二進数において、
  4+4はいくらになるか？
- C言語のchar型は8ビットの2の補数表記の整数である。char型の変数a,bがそれぞれ100であったとして、c=a+bはいくらになるか？

右のリストを実行したと
仮定して結果を想像せよ

```
#include <stdio.h>
int main () {
 char a,b,c;
 a=100; b=100; c=a+b;
 printf("a=%d, b=%d, c=%d¥n",a,b,c);
 }
```

基数表現

- 基数をKとすると数値の値は
  - $V = \Sigma di \times K^{(i-f)}$
    - ここでfは小数点位置を表す
  - 10進以上の基数を使用する場合に、桁を表す記号として0～9、A、B、C・・・とアルファベットを用いる

テキストP．113

コンピュータでよく用いられる基数

- コンピュータの世界では2のべき乗の基数がよく用いられる
  - 2進数：1桁が1ビットを表す
    - 0, 1
  - 8進数：1桁が3ビットを表す
    - 0, 1, 2, 3, 4, 5, 6, 7
  - １６進数：1桁が4ビットを表す
    - 0～9, A, B, C, D, E, F
- 1010 1111 1101 0010 0111 0101 0000 0001
- A F D 2 7 5 0 1

演習問題　8

- １２．３４を５進数、８進数として値を求めよ
- 自分の名前のイニシャルをローマ字で表記し、２桁の３６進数の整数とみなして値を求めよ

実数の表現

テキストP．112

- 数学の意味する実数を表現できるコンピュータは存在しない
- 値域を限定した近似値を浮動小数点数(floating point number)として導入する
  - 仮数部および指数部の二つの数値で数を表す
  - 1.234×10^{1}
    仮数部　　指数部
    - 対するものに固定小数点(fixed point number)がある
  - 比較的小さな指数部を用いても大きな値域を得ることができる

IEEE　７５４浮動小数点フォーマット

- **符号** (sign)**ビット**S
  - **全体の符号を決めるビット：0で正、1で負**
- **指数**(exponent)**部**E
  - **2のべき乗の指数を表す。精度によって決められる定数を加えた符号なし2進数**
- **仮数**(Mantissa)**部**M
  - **小数点以下の数値を表す。ただし、正規化された浮動小数点数では仮数部には1が暗黙のうちに加えられる。**

１０進数からの変換

- Sを決定する
- 符号を外した数値を2進数に変換
- 1.xxxxxx ×2^Kと表記しなおす
- xxxxxx部分からMを決定、KからEを決定する

- KはEそのものではないことに注意
  - テキスト(4．5)式を参照

演習問題　9

- **次の短精度浮動小数点数はいくらか？**
  - 1 01111111 000 0000 0000 0000 0000　0000
- P.124**演習**4.5 (1),(2)**を短精度浮動小数点で表せ**

演習概要

- システムLSI開発概論
- 論理回路基礎演習
  - 組合せ回路
  - 順序回路と状態遷移図
  - 表現形式
- システムLSI開発演習

3桁表示の電卓を設計してみよう

演習における設計の流れ

- 要求分析
  - 何を作るかを明確にする
- アーキテクチャ設計
  - どう作るかを記述する
- システム設計
  - アーキテクチャをテクノロジにマッピング

要求分析

開発システムを分析し、仕様詳細化

何を作るかを明確にしよう

– 分析不十分な開発は後工程への影響大

1. システムに必要な動作を抽出
   - 分析の開始時点では、過度の詳細化は避ける
   - 何を作るかを記述し、どう作るかは設計に回す
2. システムで発生するイベントを抽出
   - イベントをリストアップし、視覚化する

機能の洗い出し (要求分析) 例 (電卓のキー入力)

- 初期状態から数字キーが入力されたらLの位置に入力数値を表示
- 数字キーが表示されている状態でさらに数字キーが入力されたら表示されている数字を左に移動し、Lの位置に入力数値を表示

- L桁に数値を表示する場合、L桁の出力線に接続するレジスタ (フリップフロップ) に出力値を書き込む。

アーキテクチャ設計

- 要求分析結果に対して「どのように作るか」を設計

1. システム構成要素抽出
2. システム入出力、ユーザインタフェース定義
3. 構成要素間インタフェース定義

１０進加算のアーキテクチャ設計

- 使用可能な資源
  - ４ビット入力２本、桁上げ入力、５ビット出力を有する２進加算器を１つ使用
  - 入力データを選択するデータセレクタ (マルチプレクサ)
    - 入力データの数は前回の要求分析結果より判断
  - 出力データを一時保持するレジスタ (フリップフロップ)

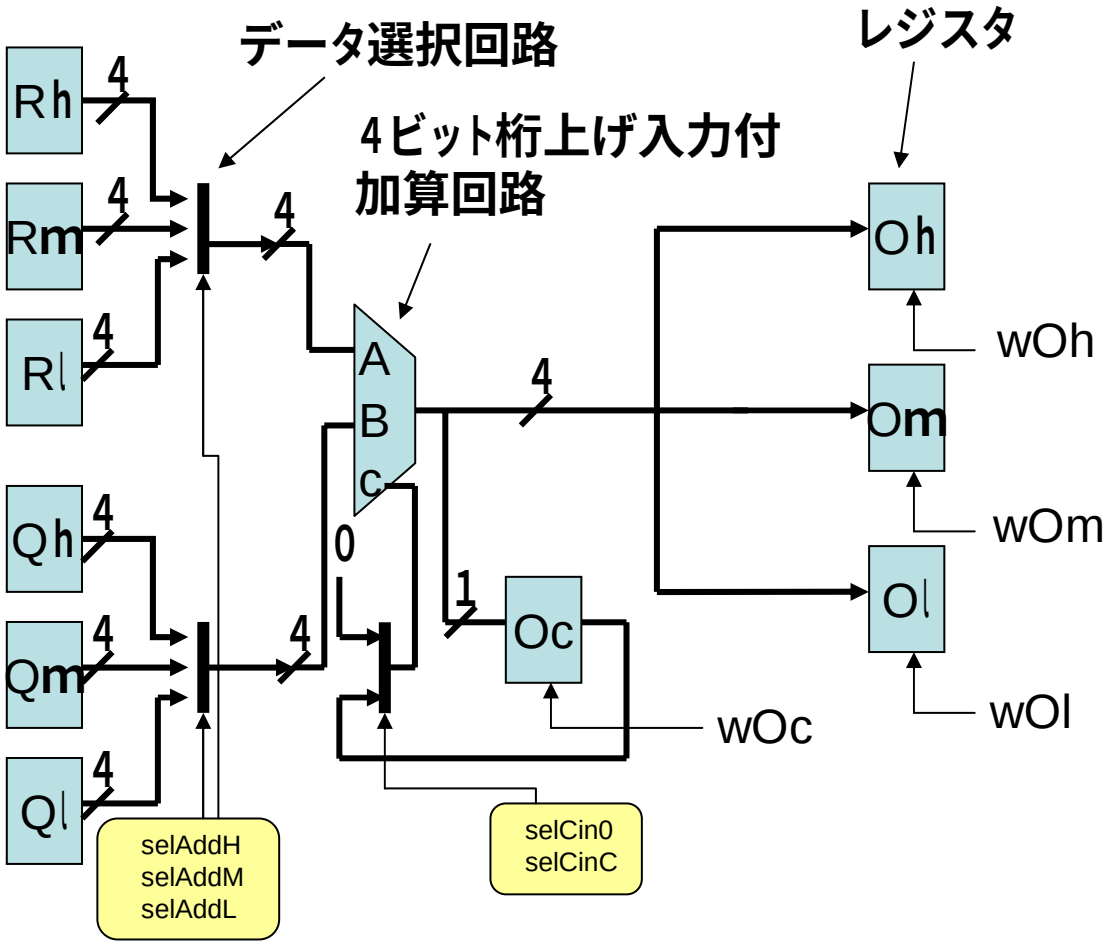
資源として不足があれば、随時追加を検討すること

１０進電卓のシステム設計

- データの流れを整理しデータパスを設計
- 制御の流れを整理し状態遷移図を設計
- データパス上のデータ移動タイミングをタイミングチャートを作成し確認

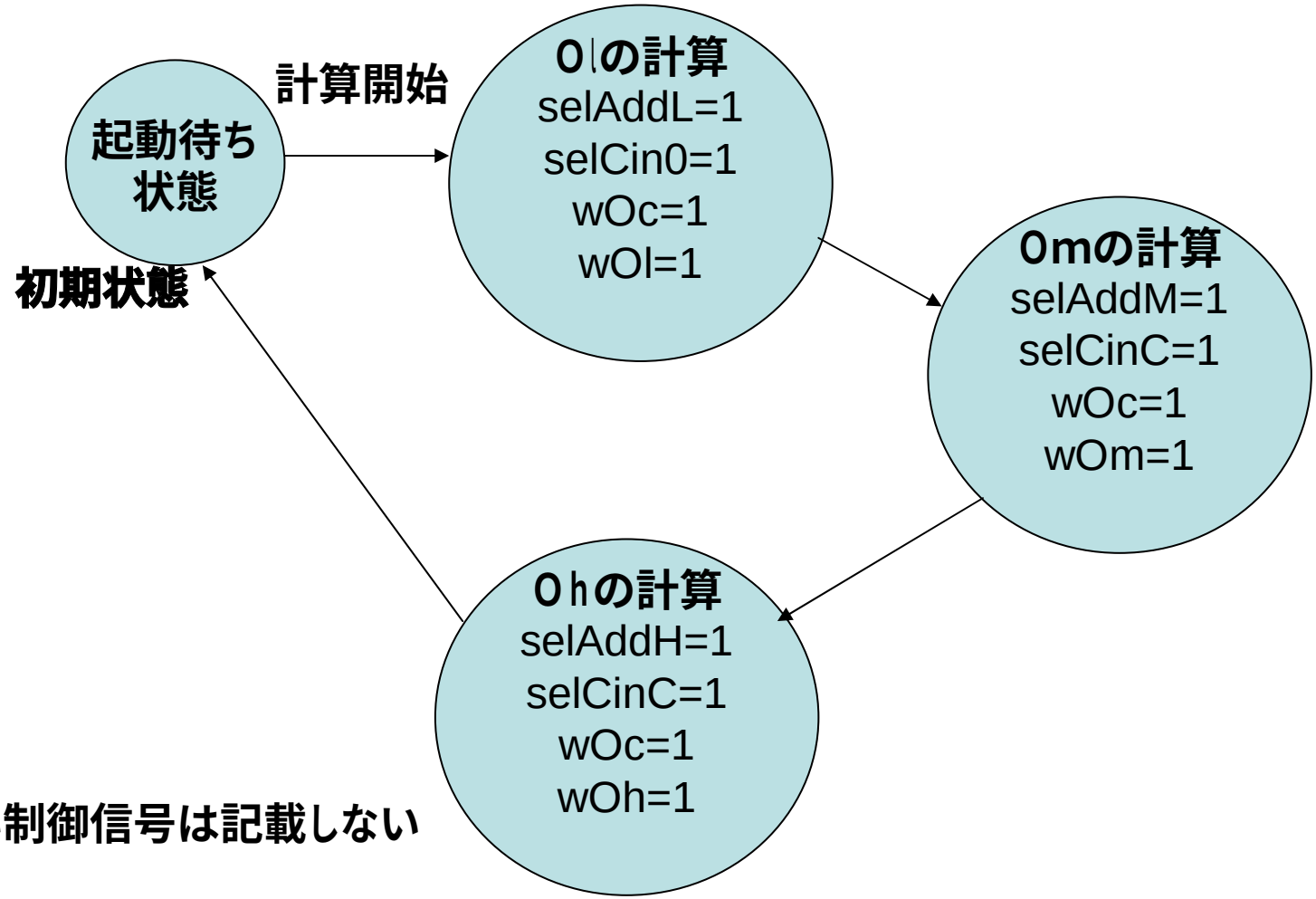
- 資源抽出が終了した時点で、主要な資源とデータの転送路を結ぶブロック図を作成
  - ブロック図の書き方はテキストP129を参照

練習課題
2進12ビット加算器



ブロック図の回路を12ビット加算器として動作させる状態遷移図を作成せよ
レジスタRとQにはあらかじめ値が入っており、レジスタOに計算結果を格納
レジスタへの入力とレジスタの書き込み信号は同一状態内で発行する

状態遷移図の設計

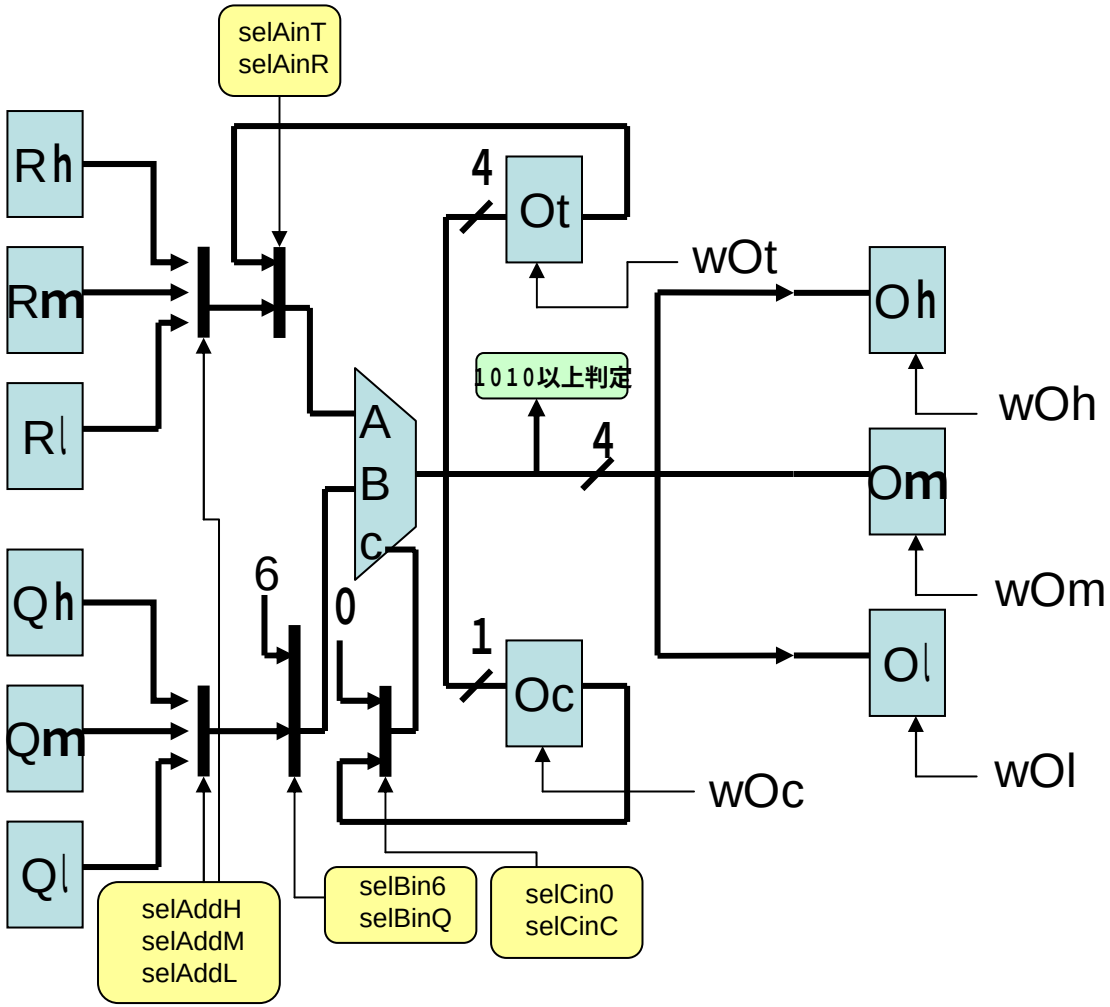


起動しない制御信号は記載しない

１０進計算の基礎知識

- 2進加算器を用いて１０進計算を行う
  - 8＋2＝１０となって欲しいが、2進加算器では
  - １０００＋００１０＝１０１０　となる。これを補正するには、加算結果にさらに6（０１１０）を加える
  - １０１０＋０１１０＝<u>１</u> <u>００００</u>
  - 加算結果が１０１０以上の場合、および、桁上げが発生した場合には6を加える
    - 8＋8＝16　１０００＋１０００＝１ ０００
    - これに6を加えると １ ０１１０ ➡16

１０進加算器作成



演習問題　１０

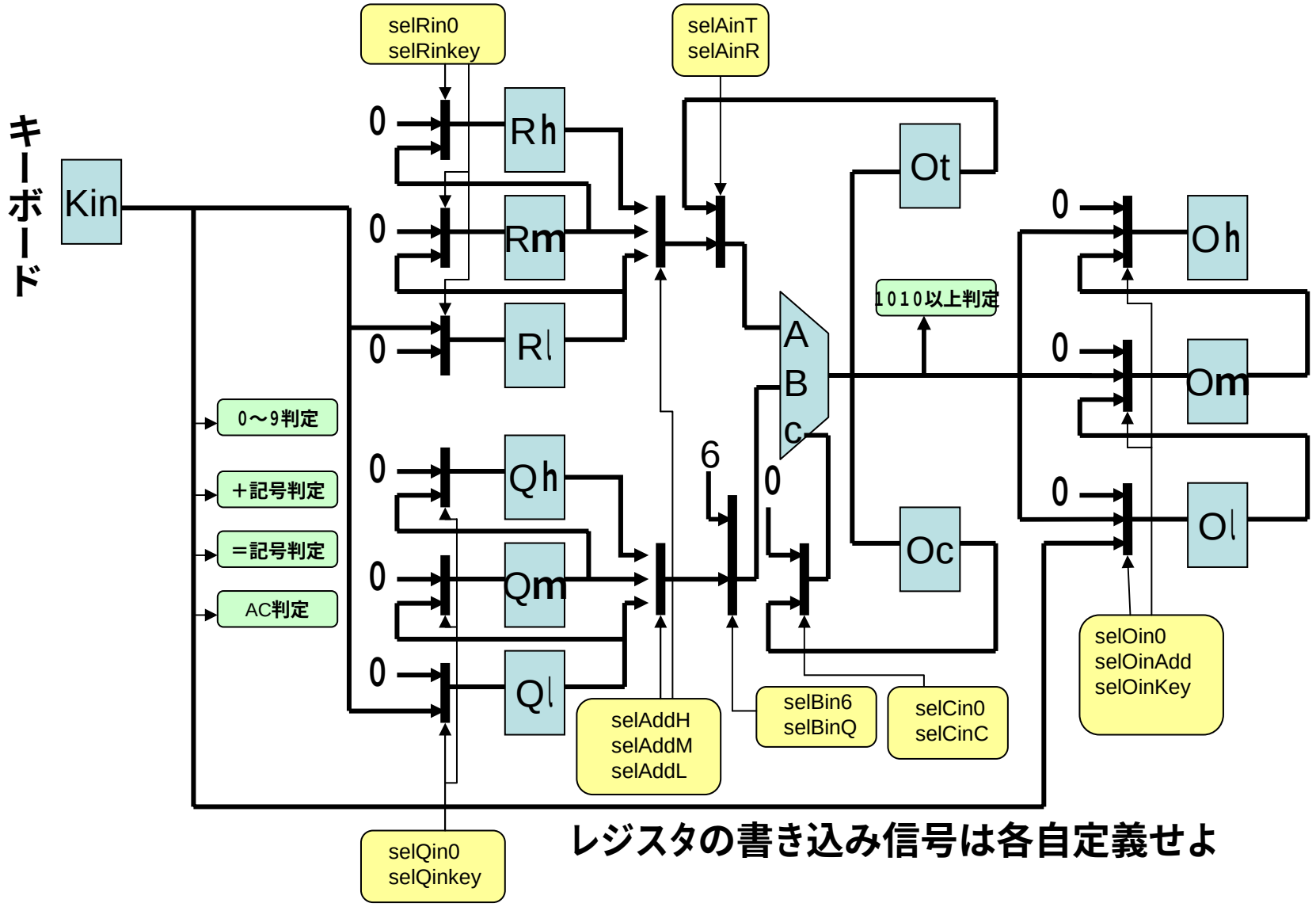
ブロック図の回路を１０進加算器として動作させる状態遷移図を作成せよ

レジスタRとQにはあらかじめ値が入っており、レジスタOに計算結果を格納

演習問題　１１

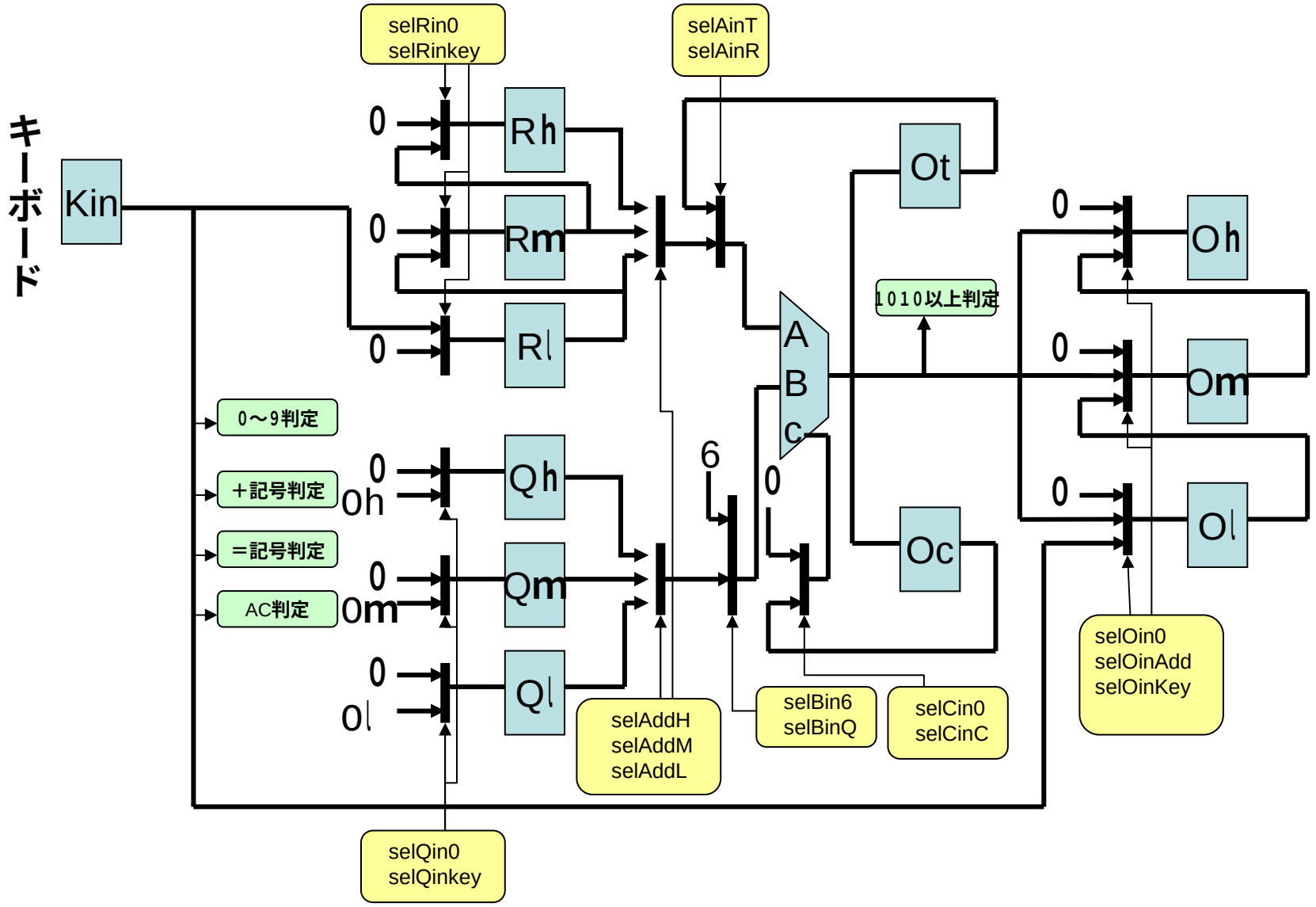
- 加算を行う3桁電卓を設計せよ
  - 4ビット入力および桁上げ入力を持つ加算器を用いるものとする
  - フリップフロップは必要な数用意すること
  - 入力は１０個の数字キーと＋、AC、＝キー
    - キーが押されたら、4ビットのデータが入力される
    - また、同時にキー入力の制御線が1となる
  - 出力は１０進3桁 (各桁4ビットの出力線とする)
- 要求分析、構成要素抽出を行った後、ブロック図、状態遷移図を作成せよ

電卓ブロック図参考例



レジスタの書き込み信号は各自定義せよ

電卓ブロック図参考例 (改良版)



まとめ

- システムＬＳＩ設計の考え方
  - 要求分析
  - アーキテクチャ設計
  - システム設計
- システムＬＳＩ設計のための基礎知識
  - 組合せ回路
  - 順序回路
  - 表現形式
- システムＬＳＩ設計演習

付録

- ハードウェア記述言語の講習を受けた後、各自設計した電卓をＨＤＬで実現してみよう。

- まずは2進１２ビット加算器の例題を見てから、設計の流れに沿って、検討しよう。

2進12ビット加算器HDL記述例

```
/* 2進12ビット加算器の例題 */
/* 例題ではレジスタはあらかじめ設定済みとしたが、
   ここでは、各レジスタ設定を外部から行う */
module add12 {
 input key<4>;
 instrin setRh, setRm, setRl, setQh, setQm, setQl;

 instrin doadd;

 output outH<4>, outM<4>, outL<4>;
 reg Rh<4>, Rm<4>, Rl<4>, Qh<4>, Qm<4>, Ql<4>, Oc;
 reg Oh<4>, Om<4>, Ol<4>;
 instrself selAddH, selAddM, selAddL, selCin0,
           selCinC;
 instrself wOc, wOh, w0m, wOl, exec_add;
 sel addinA<4>, addinB<4>, addinC, addOut<5>;

 stage_name exec {task t();}

 par { outH = Oh; outM = Om; outL = Ol; }

 instruct doadd generate exec.t();

 instruct setRh Rh := key;
 instruct setRm Rm := key;
 instruct setRl Rl := key;
 instruct setQh Qh := key;
 instruct setQm Qm := key;
 instruct setQl Ql := key;

 instruct selAddH par { addinA = Rh; addinB = Qh; }
 instruct selAddM par { addinA = Rm; addinB = Qm; }
 instruct selAddL par { addinA = Rl; addinB = Ql; }
 instruct selCin0 addinC = 0b0;
 instruct selCinC addinC = Oc;
 instruct wOc Oc := addOut<4>;
 instruct wOh Oh := addOut<3:0>;
 instruct wOm Om := addOut<3:0>;
 instruct wOl Ol := addOut<3:0>;
 instruct exec_add addOut = addinA + addinB + addinC;

 stage exec {
  state_name invoke_wait, calcL, calcM, calcH;
  first_state invoke_wait;

  state invoke_wait goto calcL;
  state calcL par { selAddL(); selCin0(); wOc(); wOl();
          exec_add(); goto calcM;}
  state calcM par { selAddM(); selCinC(); wOc(); wOm();
          exec_add(); goto calcH;}
  state calcH par { selAddH(); selCinC(); wOc(); wOh();
          exec_add(); goto invoke_wait; finish;}
  }
}
```

電卓を実現するSFL記述

- 電卓を実現するSFL記述の作成を行ってみよう

モジュール名: calculator
入力端子: key<4>
出力端子: outL<4>, outM<4>, outH<4>
制御入力端子: keyin、**制御出力端子**: keyreq
入力を要求するときにはkeyreq**を起動し、外部から**keyin**とともに**key**に入力データが到着するとする**

データパス設計

- ブロック図の各線に名前をつける
  - テキストP．１２９のCPUブロック図を参照
- セレクタの選択信号、レジスタの設定信号をモジュールの制御線（instrinもしくはinstrself）として宣言

制御系設計

- 状態遷移図の各ステートをステージ中に状態(state)として宣言。初期状態をfirst_stateとして宣言
- 各state記述においてその状態に出力する制御信号を起動する