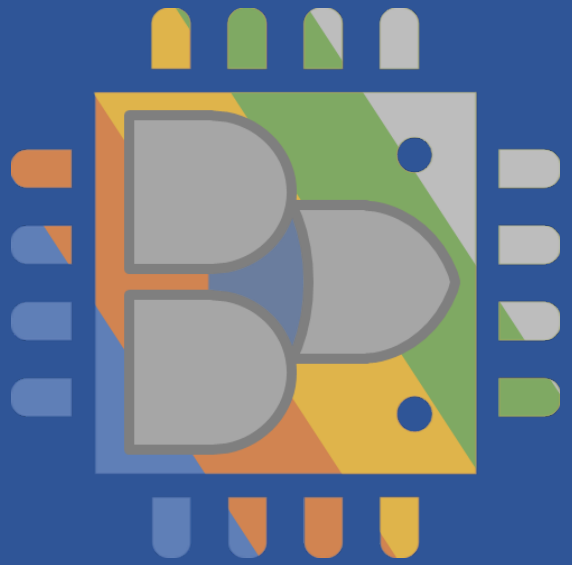


計算機システム設計論(8)
最小構成CPU

担当：　西　宏章

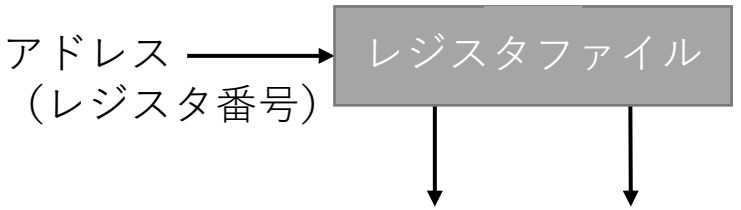
国立情報科学専攻
論理工学講座

マイクロコントローラの設計

- 非常に簡単なプロセッサを設計し、若干の拡張をします
  - マイクロコントローラの設計は、ハードウェア設計の基本が詰まっています
- プロセッサの構成要素
  - レジスタ:　変数記憶領域
  - プログラムカウンタ:　どこを実行しているかを示す
  - ALU:　計算を実際に行う
  - メモリ:　プログラムや計算結果を格納する
- これらの概念がつかめていない人は、今のうちに解決すること
- 計算して格納するという仕組み
  - レジスタは変数の記憶領域で、この変数の値をALUで計算するという構造
  - つまり、この2つがループになって構成されている
    - 組み合わせ回路の構成と同じであると気付くであろう
  - これらを制御する周辺回路群は後で設計するとして、ALUとレジスタファイルはどのように構成すればよいか？

レジスタファイルの構成

- レジスタファイルはFFの集合体
- 計算をする前提であればオペランド２つ
- つまり、2つの値を出力
  - アドレス込みでaポート、bポートとして設計
  - Register Arrayはメモリとして構成
    - レジスタ1つあたりの大きさを16bitとする(1990年代)
    - 性能を上げる場合はビット幅を増やせばよい
  - 書き込む口とアドレスもある
  - ここではレジスタの数を4とする



Complete it!

```
module rega(input logic [1:0] arad, …);
  logic [15:0] regar [3:0];
  always_ff @(posedge clk)
    if(en) regar[wad] <= in; // 書き込む側
  assign outa = regar[arad]; // 読み出す側 (aポート)
  assign outb = regar[brad]; // 読み出す側 (bポート)
endmodule
```

Think it!

リセットは？

リセットはどうするべきだろうか？

・必要？　・不必要？

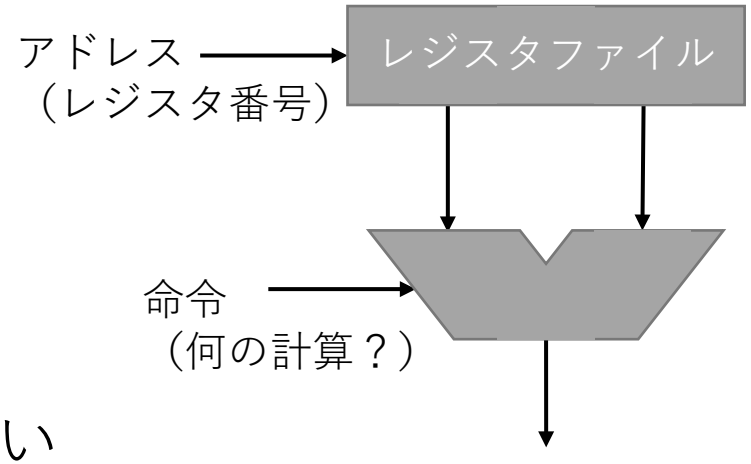
リセット付きFFは実は大きい

無しはおよそ7NANDゲートサイズ

有りはおよそ12NANDゲートサイズ

レジスタファイルとALUの構成

- ALUは命令の指令を受けて計算
  - レジスタファイルの2つの値を使って計算
  - 何の計算をするかを入力とする
  - もちろん、出力もある
- 計算要素としてよく利用される命令
  - 加減乗除算だが除算は組み合わせ回路で合成できない
  - ビット演算・ビット処理・ビットテスト・論理演算
  - 各種シフト




```
module alu(input [15:0] a, b,
  input [2:0] op, output [15:0] result);
  always_comb
    enum(ADD, SUB, AND, OR, XOR, SFL, SRR, THRU) OPTY;
    case(op)
    …
    endcase
endmodule
```

Complete it!

スルーって何？

Think it!

スルーさせると何がうれしいのだろうか？
・スルー必要？　・スルー不必要？
直接0や1などの値を作る例もあるが何がうれしいのだろうか？

シフタ

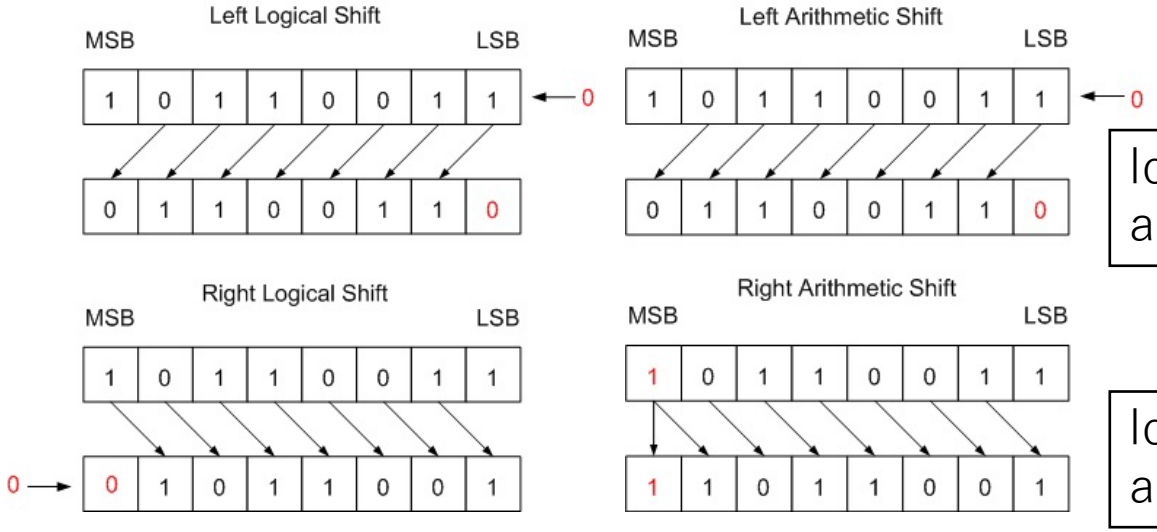
- シフタの動作
  - シフタの動作

N[2:0]	Y[7:0]
000	D[7:0]
001	{D[7:0],1'b0}
010	{D[7:0],2'b00}
011	{D[7:0],3'b000}
100	{D[7:0],4'b0000}
101	{D[7:0],5'b00000}
110	{D[7:0],6'b000000}
111	{D[7:0],7'b0000000}

- バレルシフタ
  - 例えばデータD[7:0]をシフト数(N[2:0])だけ位置をずらす演算を効率よく実装

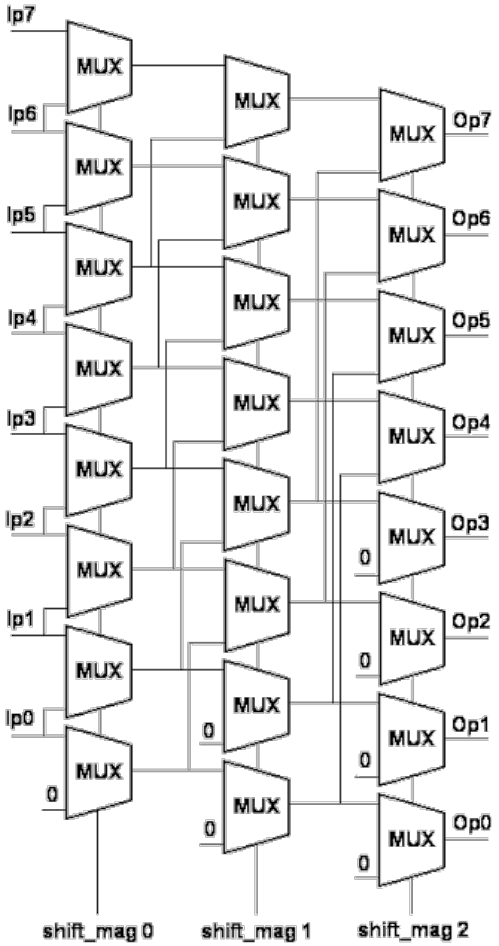
掛け算の構成に利用され、2進数の掛け算は組み合わせ回路で実現できる

- 論理シフタと算術シフタ



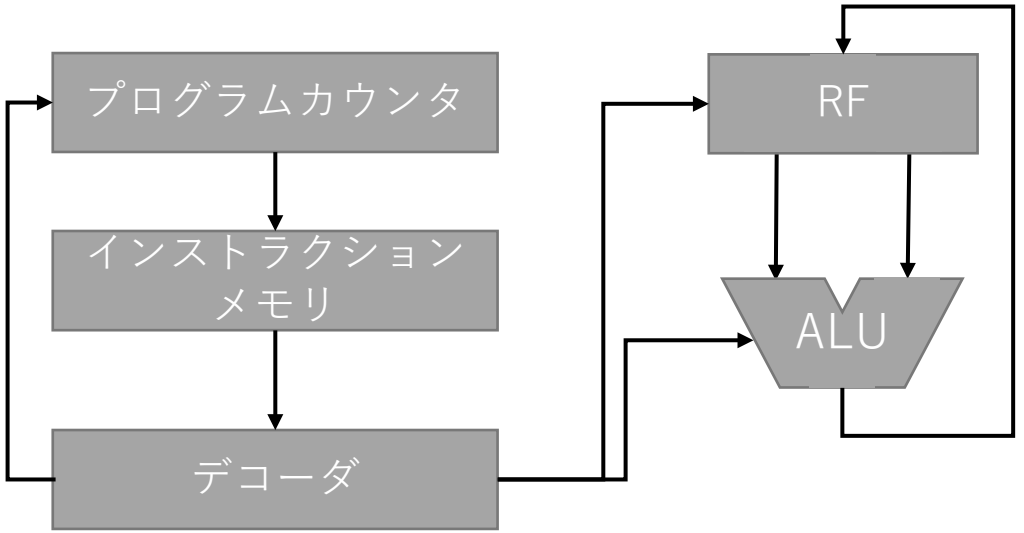

```
logic [15:0] a, b, c; //符号付でもOK
assign a = b << c; //論理左シフト
```

```
logic signed [15:0] a, b, c; //符号付
assign a = b >>> c; //算術右シフト
```



レジスタファイルとALUを制御する

- 最もシンプルな構造では
  - 本来はデコーダも不要
  - プログラムカウンタ(PC)は「どこ」を実行しているかのアドレスを管理
  - インストラクションメモリ(IM)はプログラムを管理、PCが差す命令を出力
  - デコーダはその命令にあった制御



- ジャンプ命令や分岐命令はないが
  これでもチューリングマシンとしては動作

```
module imem(input [7:0] pc,
  output [?:0] o);
  always_comb
    case(pc)
    8'h00: o = ?'b000_…;
    8'h01: …
    endcase
endmodule
```

Complete it!

```
module pc(inpupt halt,
  output logic [?:0] pc, …);
  always_ff @(posedge clk or
    posedge rst)
    if(rst) pc <= 0;
    else if(!halt) pc <= pc + 1;
endmodule
```

Think it!

これだと電卓としても…？

このままでは電卓に使うことすらできない
電卓として使えるようにするにはどうすればよいか？
デコーダーは？
命令を小さくくできるため効率が良い
では、デコーダのデメリットは？

インストラクションとデコーダ

- ここではデコーダは配線のみとする
- 現状でどれだけの制御線を操る必要があるか？(ビット幅は参考)
  - レジスタファイル
    - Aポートにどのレジスタの値を出すかのアドレス情報(2bit)A
    - Bポートにどのレジスタの値を出すかのアドレス情報(2bit)B
    - ALUの演算結果をレジスタに書き戻す際のアドレス情報(2bit)Wと書き込み許可(1bit)E
  - ALU
    - ALUへのコマンド(3bit)O
  - PC
    - 動作を止めるhalt(1bit)h
- 現状で10bit幅なので、そのままインストラクションメモリに格納
  - 例えば、11'bh_E_WW_OOO_AA_BB などとする
  - レジスタ0番とレジスタ1番に対してALUでコマンド番号1番の計算を行ってレジスタ2番に答えを描き戻すという命令は、11'b0_1_10_001_00_01になる

Machine Language(機械語)

- このようにCPUが直接理解可能な命令を機械語と呼ぶ
- 機械語とニーモニック
  - 機械語は人間には理解しにくいため、分かりやすい形に変換したのがニーモニック
  - アセンブラはニーモニック(アセンブラコード)を機械語に変換、逆が逆アセンブラ
  - 11'b0_0 _10_001_01_00 は、コマンド1番が足し算とすると
    ADD r2=r0, r1などと記載
  - このように、ニーモニックも含めて自由に設計できる
- ここまでの内容で、次の点をクリアするとフィボナッチ数列をひたすら出力する計算機が構成できます
  - ただし、値の初期化だけ現状でできません。ここだけズルをして解決します。
  - フィボナッチ数列とは、0, 1, 1, 2, 3, 5, 8, 13, …と、$a_{n+2} = a_{n+1} + a_n$となる数列
  - ちなみに一般解は、$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$

フィボナッチ生成器

- プログラム実行開始時に、r0に0を、r1に1を入れておく（これがズル）
- 計算手順
  - $a_{n+2} = a_{n+1} + a_n$において、a_n=r0, a_{n+1}=r1, a_{n+2}=r2とする
    従って、r2 := r0+r1を計算する
  - nを一つ進める。つまり、r0 := r1, r1 := r2とする(順番大事)
  - これを繰り返す
- 実際に設計する
  - ここでALUの何もしないスルー命令の良さがわかる
  - 動作を確認する
- ズルを解決する
  - 今のままでは5+10といった任意の数の足し算すら計算できない

演習問題（8）

- r0やr1をリセット時に初期化するのではなく、指定したレジスタを0や1にする専用命令 SET1, SET0を持つようなアーキテクチャを構成しなさい
  - ALUに0や1を出力する専用命令を構築する
- これに加えて、足し算と引き算が計算できるALUとしなさい
  - つまり4つの命令を備えること
- このALUを用いて、適切なレジスタファイルを構成し、ペラン数を順に求める計算機を構築しなさい。なお、レジスタは16bitとする
  - ペラン数：

$$P(0) = 0, P(1) = 2, P(2) = 3$$
$$P(n) = P(n-2) + P(n-3)\ for\ n > 2$$

    - n-頂点の閉路グラフにおける異なる極大独立集合の個数はn番目のペラン数となる
    - ペラン疑素数などは面白い

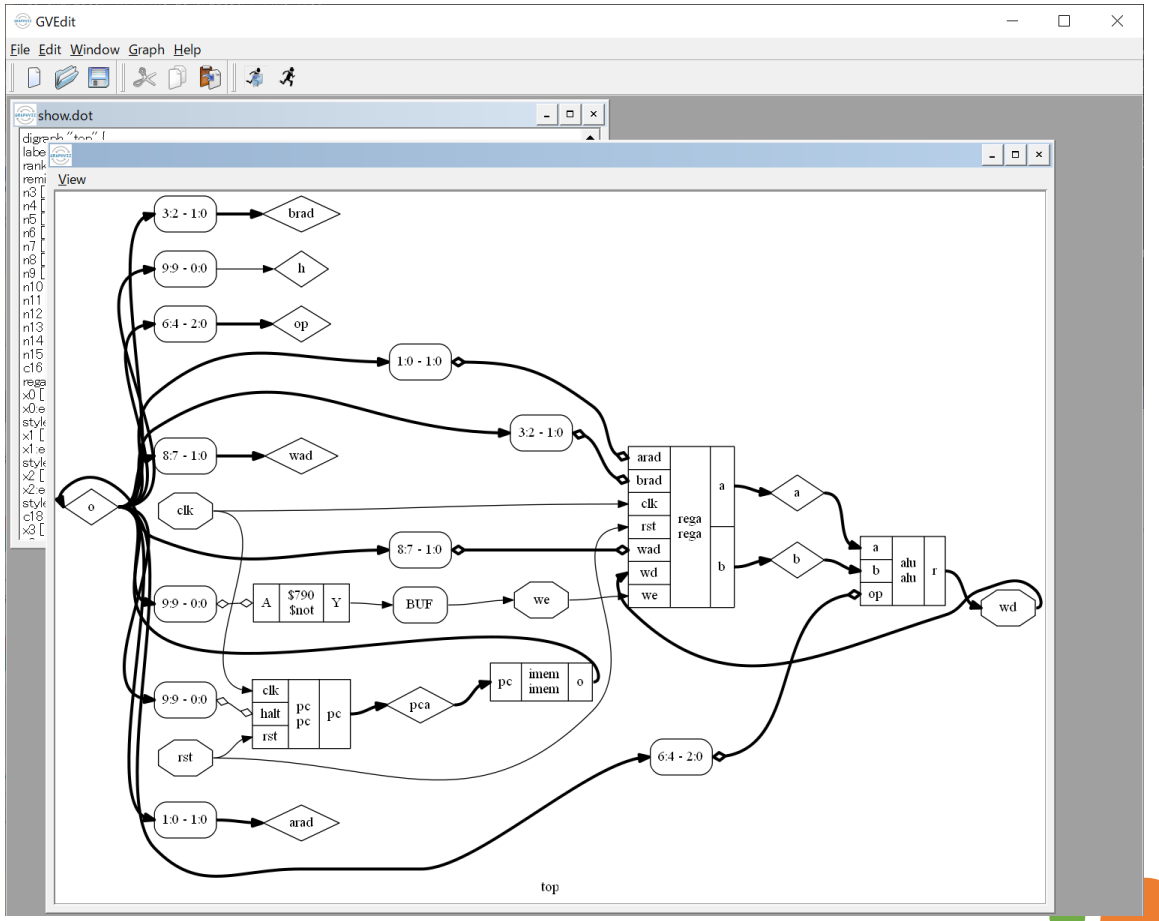
演習問題（８）

- 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（８）」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)
  - これらのフォーマットに従っていないレポート答案は受け取らない
  - 提出締め切りはLMSを確認すること

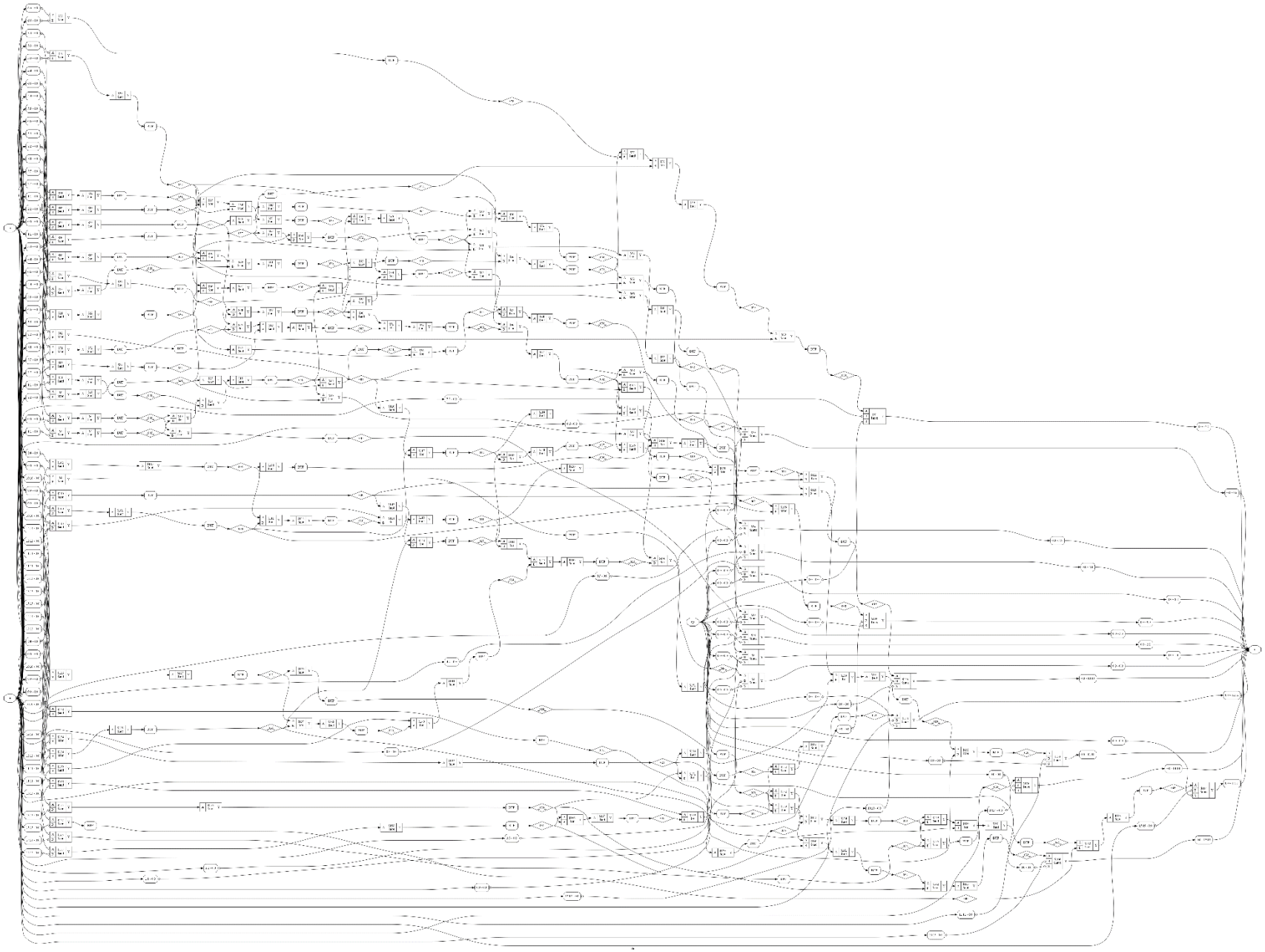
参考

- フィボナッチ計算機（既に立派な名前がついてしまっていますが）をyosysで合成してみました
  - MacOS版のxdotにはバグがあるのか、生成されたdotファイルが見れませんでしたので、別途ビューアを指定してください
    - Chip area for top module '¥top': 23055.000000
      - Chip area for module '¥top': 16.000000
      - Chip area for module '¥rega': 17072.000000
      - Chip area for module '¥pc': 1700.000000
      - Chip area for module '¥imem': 922.000000
      - Chip area for module '¥alu': 3345.000000

    となりました
    - 右はtopモジュールです
  - 次のページに、見るに堪えないALUの合成結果を貼り付けます
    - 足し算器がしっかりと合成されています
  - 全体合成は数秒で完了します



ALUの合成結果（見るに堪えない）



Gitを使う

- アプリケーション開発に関する共有ストレージと、オンラインの強力な共同バージョン管理サービスです
  - https://www.sejuku.net/blog/77097こちらが、比較的新しくて参考になるサイトかと思います。実際はもっとシンプルにできます。
  - https://git-scm.com/downloadsこちらのサイトでgitをインストールしてください。
- この授業で必須となる簡単な使い方
  - まず簡単にファイル一式を手に入れるには次のようにします。
    コマンドラインにgit clone https://github.com/keioNishi/lec-compsysと入力
    - 公開レポジトリなので、説明サイトにあるような、アカウント登録も、特に必要ないはずです。
  - lec-compsysというフォルダが作成され、テキストで紹介しているコードも含めてすべて取得できます。
  - こちらでgitの内容を更新した際には、フォルダの中で git pullとすれば、自動的に変更された部分だけ取得できます。とても便利です。

Makeを使う

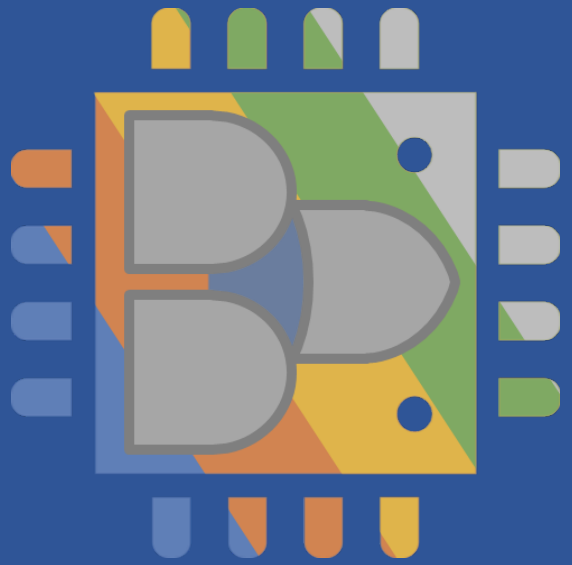
- ファイルの更新日付を利用し、依存関係から必要なコマンドを自動実行します
  - Makefileの書き方や、makeの使い方で検索するといかに便利かがわかります
- 皆さんが簡単に実行できるようにMakefileを提供しています
  - 極めて基本的な書き方になっているため、すぐに中身を理解できると思います（その代わり、冗長かつ面倒な書き方です。本来はもっとシンプルに書けます）
- Windowsでは、次の通りインストールしてください(macは入っている)
  - 次の3つについて「Binaries」をダウンロード
    - http://gnuwin32.sourceforge.net/packages/make.htm
    - http://gnuwin32.sourceforge.net/packages/libintl.htm
    - http://gnuwin32.sourceforge.net/packages/libiconv.htm
  - 全て展開してbinディレクトリ内のmake.exe、libintl3.dll、libiconv2.dllなどをC:¥Windows¥System32 の中に移動
  - 64bitの場合は C:¥Windows¥SysWOW64 へ移動
- 移動しなかった残りのファイルはすべて削除して結構です。

Makefileの作り方

次のファイルの例に従えば、
make all、もしくは単にmakeとすると、シミュレーションができます
make showとすると、波形がみれます
make synとすると、合成します。当然ですが、sw.ysの作成と添付が必要です

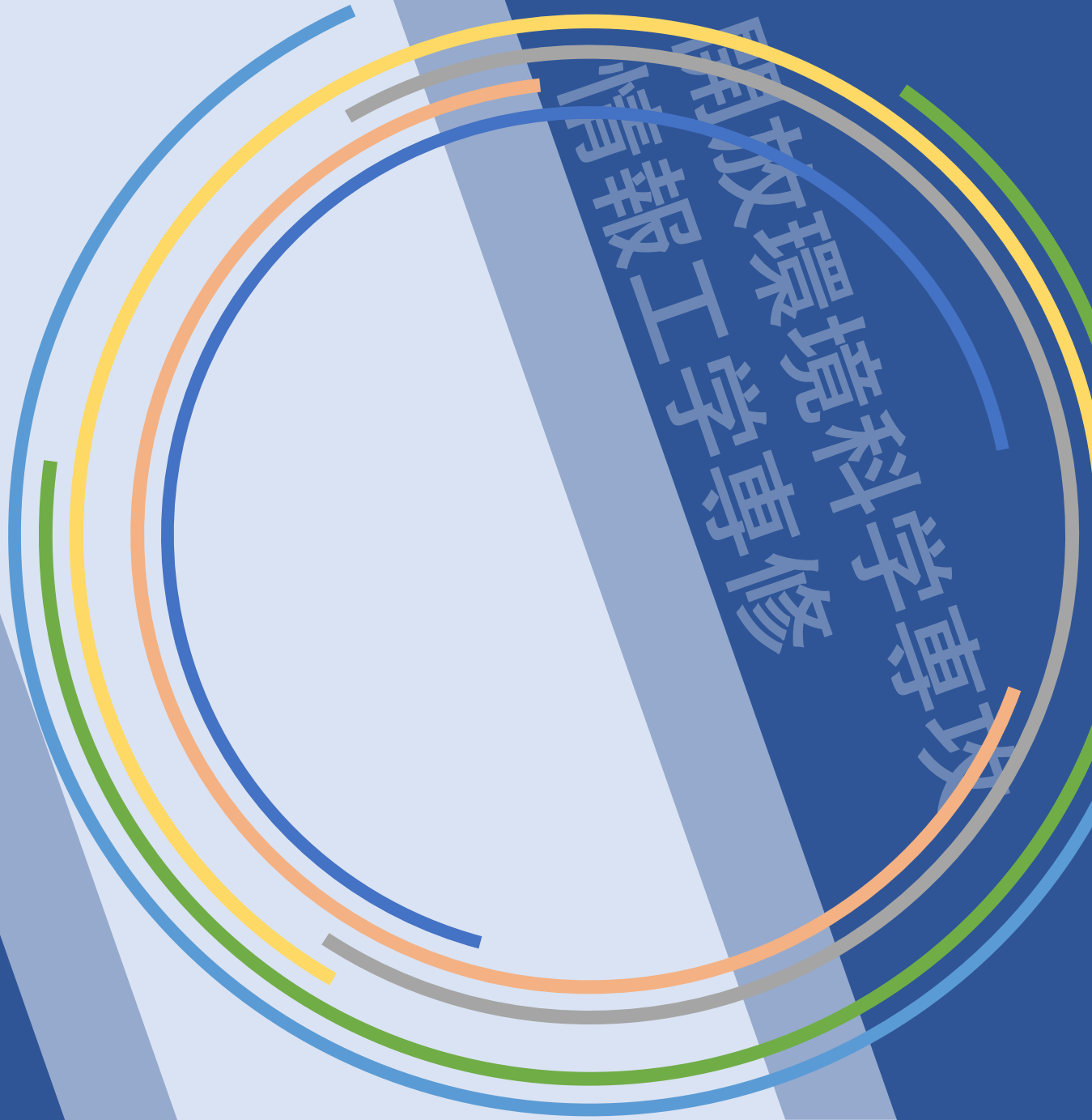
Makefileの例

```
all:
	iverilog -g2012 ソースコード群
	vvp a.out
show:
	gtkwave file.vcd
syn:
	yosys file.ys
synsim:
	iverilog -gspecify –Ttyp synth.v test.v ../osu018_stdcells.v
	vvp a.out
```



計算機システム設計論(9)
値の代入

担当：　西　宏章

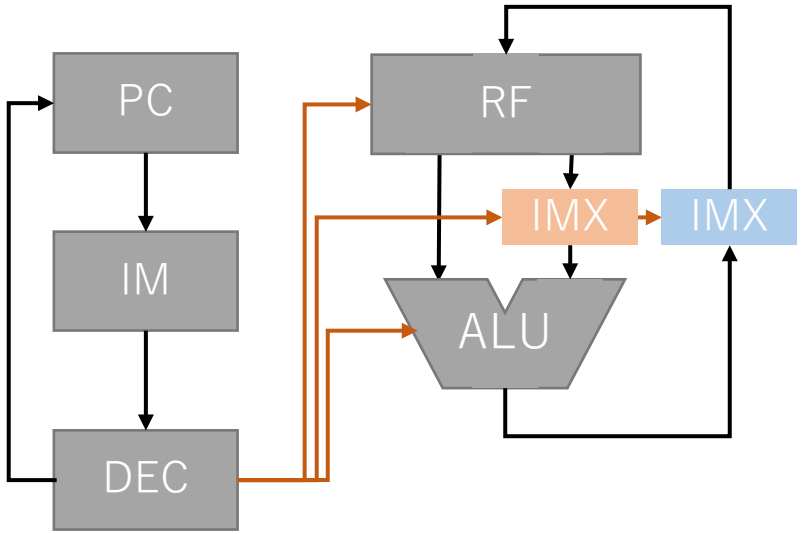


レジスタへの値の代入

- 基本として次の2つの方法が考えられる
  - インストラクションメモリのある番地に値を書き込んでおいて、これを読み出す
    - 美しくはないが、効率の良いやりかた
  - 直値代入命令を構成する
    - 考慮が必要
- メモリの内容を参照する
  - 何の値がどこにあるかを別に管理しなければならない
  - メモリのビット幅がそのまま利用できる
- 直地代入命令を利用する
  - 値が直接記載されていてわかりやすい
  - インストラクションメモリのサイズを浪費する・無駄が多い
- 共通する課題
  - 先ほどのデータサイクルのどこに値を入れるのか？
  - ALUの前ならば値で演算できるがパスが長くなり動作が遅くなるvice versa

直値代入命令

- ここではLoad Immediateを構成する
  - インストラクションメモリにデータ幅16bitを
    さらに追加するのは無駄(11+16=27bit命令になる)
    - では半分づつ書けばよいという実装があるのでそれに倣う
    - 他にも2サイクルの命令にするという実装もよくある
  - LIH:上位8bitのみ入れ替え、LIL:下位8bitのみ入れ替え
    - 演算前で直値演算も可能にすると IMX の位置が考えられる
      - 8bit直地I、LI(L)、HL(H)とすると、20'bh_LH_IIIIIIII_WW_OOO_BB_AA、結構大変
    - 計算後で代入するならば、IMX の位置が考えられる(利用価値は低くなる)
  - そろそろ、データと同じ16bitにしてデコーダを整理してみよう
    - あらゆることができるようにすると、縮約はかなり面倒
    - ここでは、例として、あまり沢山表現することは考えずシンプルに縮約する



Complete it!

```
module lidx(input [15:0] i, input ? li, hl, …);
  always_comb
  // if でも case でも li(LI命令である) (HL)LIHかLILかで
  // output oを生成する回路を構成する
endmodule
```

Think it!

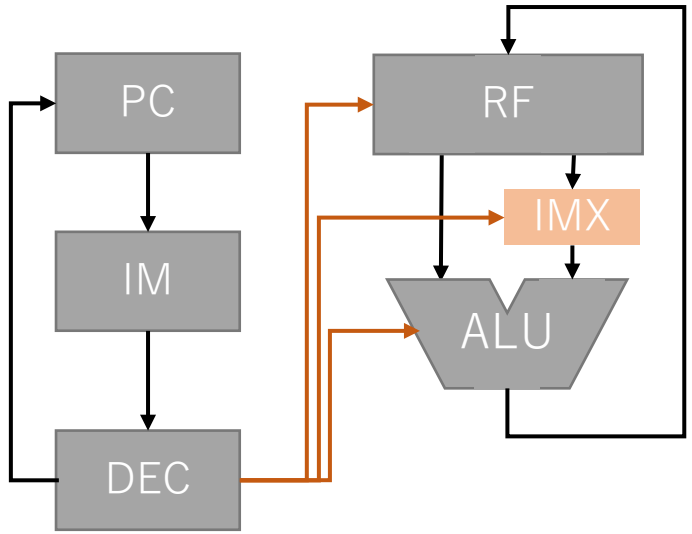
どちらの方針か？

・正解はないが、問題・改善点はあるはず
  ・なぜ演算前の方がよいのだろうか？
  ・演算前で半分だけの意味はどれだけあるか？
・デコーダの構成を考えよう

デコーダの一例

- 制御線の数に対して命令のbit幅を削減するために命令をエンコードする
  - これをデコードするのがデコーダ
  - 今後の仕様を満足することは考えず、今回の例を示す

```
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 0 0 0 * * * * * * * * * * * * 0 ; NOP
0 0 0 0 * * * * * * * * * * * * 1 ; HALT (1)
0 0 0 1 * * rw> op----> a-> b-> ; CAL rw=ra,rb
0 0 1 0 * * rw> im----------> ; LI rw,(s)im
0 1 0 0 rw> b-> im----------> ; LIL rw,rb,im
0 1 0 1 rw> b-> im----------> ; LIH rw,rb,im
1 0 0 oprw> a-> im----------> ; ADD/SUB rw=ra,im
```



- ここまでで、電卓プログラムの作成と実行が可能になる。
  （演習問題レポート無し）足し算を計算するプログラムを構成して実行しよう

Complete it!

ここまでで実現可能な命令のニーモニックを考えてみよう
・演算(演算では単純にレジスタ間のデータ移動もある)
・直地代入
・停止

Think it!

割り算はどうする？値は？
・この講義では触れないがclkを使って同期的に次々と割り算を実行する
数の表現はどうする？
・符号はともかく、固定・浮動小数点は？

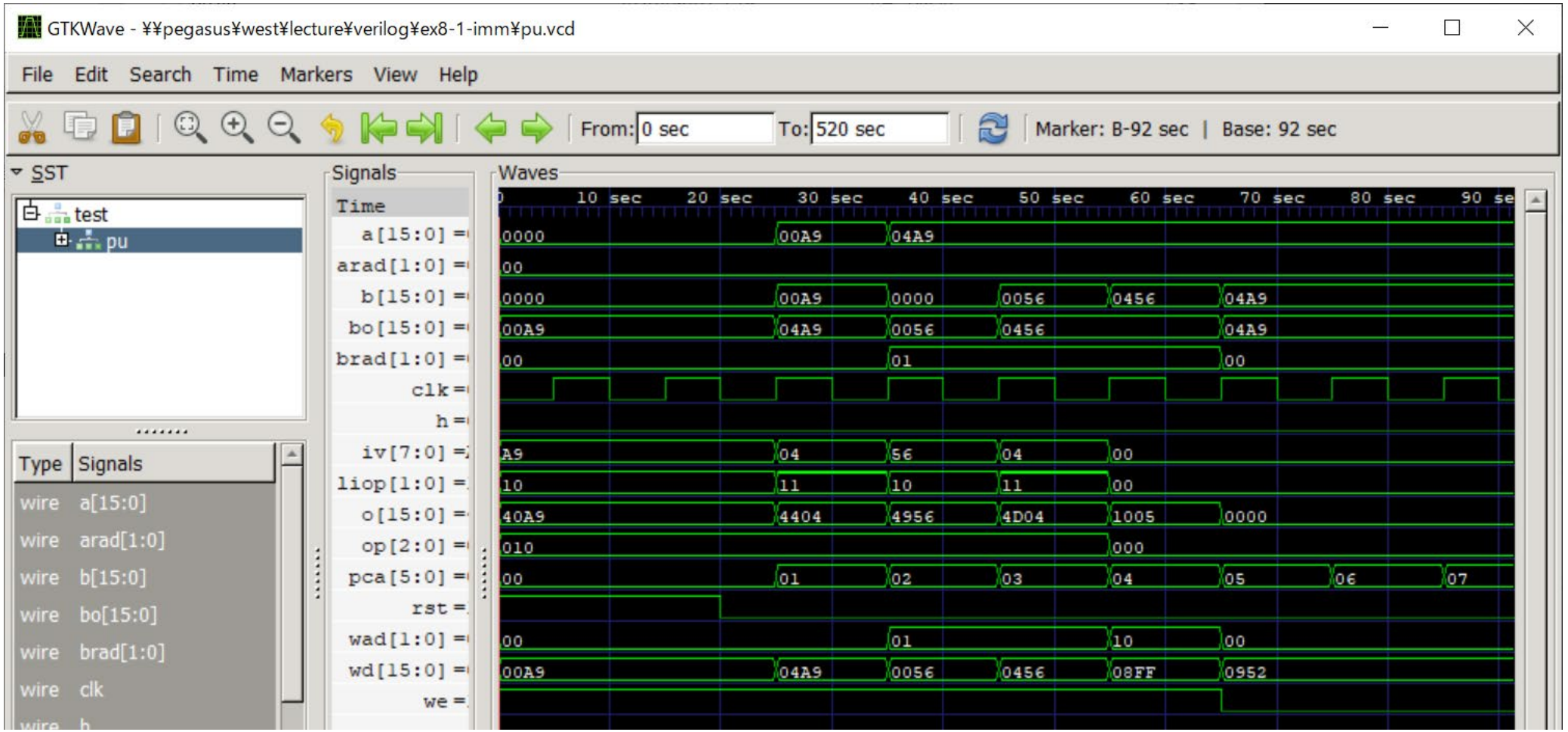
実行例


```
5'h00: o = 16'b0101_00_00_10101001; // LIL r0, r0, 8'b10101001
5'h01: o = 16'b0101_00_01_00000100; // LIH r0, r0, 8'b00000100
5'h02: o = 16'b0101_00_00_01010110; // LIL r1, r1, 8'b01010110
7'h03: o = 16'b0101_00_01_00000100; // LIH r1, r1, 8'b00000100
5'h04: o = 16'b0001_00_10_0000_00_01; // add r2=r0,r1
```

16’b0000010010101001 = 16’h04A9

16’b0000010001010110 = 16’h0456

16’b0000100011111111 = 16’h08FF

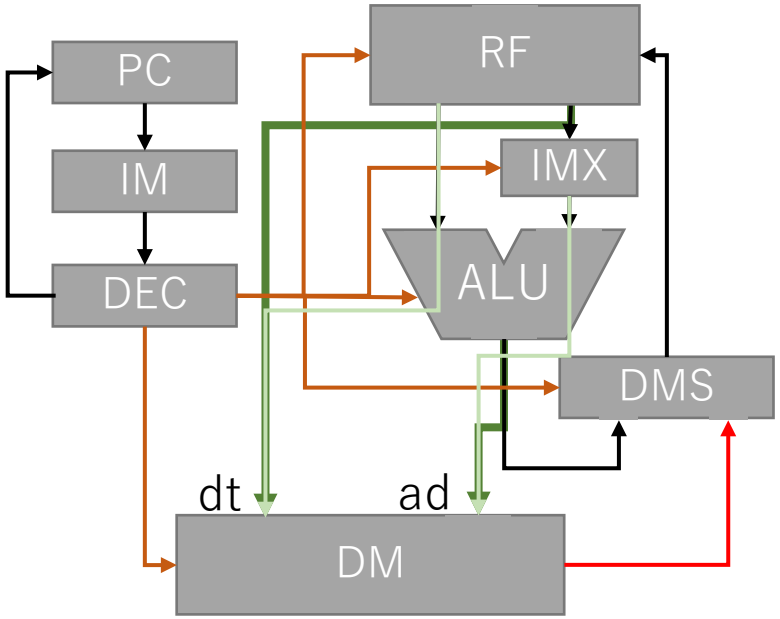


`define と parameter

- define
  - `define SIZE 15 などとして使う
  - 指定はmoduleの外
  - グローバルな定義
- `include “defs.vh” などとして、他のファイルを読み込むことができる
- parameter
  - parameter STEP=10; などとして使う
  - モジュール内で定義、 ；が必要
  - モジュールをインスタンス化する際に、上位階層でparameter値の設定ができる
    - パラメタライズ記述

データメモリ内容の参照

- データメモリからの読み出しと書き込み
  - メモリに必要な信号線は
    - 読み出しアドレス・書き込みアドレス
      これを同時にしたいか？という議論
    - 書き込みイネーブル
    - 書き込みデータ(こちらはinput)
    - 読み出しデータ(こちらはoutput)
  - これらを考慮すると図のようなパスが検討される
    - インデックスアクセスにしたいため、少しマニアックな設計を検討する
  - これで計算して答えを保存するフィボナッチ生成器ができる



Complete it!

```
module dm(input [7:0] ad, input rw
  input [15:0] di, output [15:0] do, clk);
  logic [15:0] dm [255:0]; // 好きに決めてよい
  always_ff @(posedge clk)
    // 設計しよう
assign do = dm[ad]; // 読み出し側データは垂れ流し
endmodule
```

同様に、メモリ参照込みのニーモニックを構成してみよう

Think it!

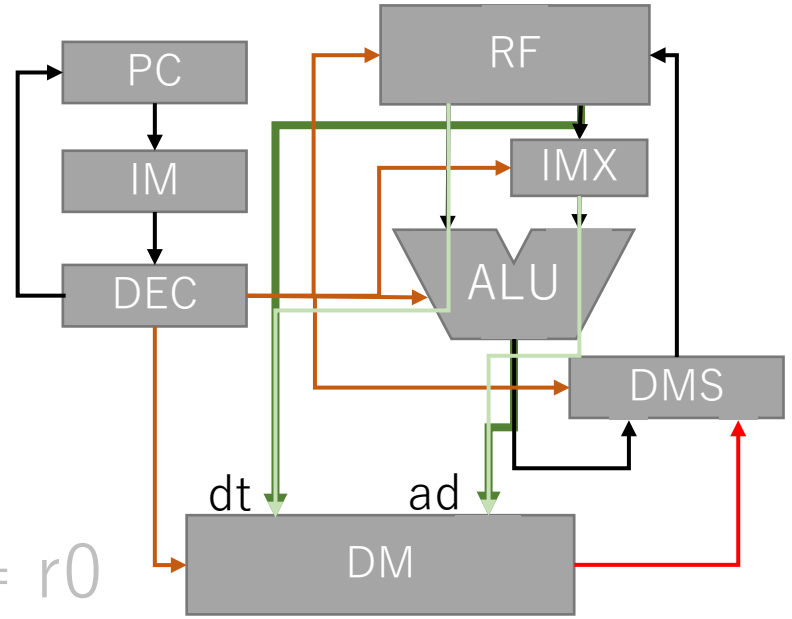
メモリ配線と構造

・アドレスとデータそれぞれaポート、bポートどちらにするべきか？上記はaポートがデータ、bポートがアドレスであるが、これはどういう方針で決められているだろうか？

・そもそもインストラクションとデータメモリを分ける意味は？

ここまでででできること

- 次のニーモニックの動作を図で追いかける
- LI r0 = 0x0003 // 実際はLIHとLILで記述
- LOAD_MEMORY/LM r0 = [r1]
  ここでは番地を[]で表す。 LM r0 = [0x0003]
- ADD r0 = r1, r2 ; AND r0 = r1, r2 ; MV r1 = r0
- STORE_MEMORY/SM [r0] = r1 ; SM [0x0003] = r0
- 次のような変わった命令も可能
- r1 = addr_of(SM [0x0003] = r0) ; r1 = addr_of(SM [r0] = r2)
  - メモリに書き込みつつ参照変数を演算
  - r1 =addr_of(SM [r0+0x0002] = r2)もOK
- LM r1 = [addr_of(SM [0x0003] = r0)]
  - メモリに入っていた値を読みつつ書き出し
  - LM r1 = addr_of(SM [0x0003] = r0)]もOK



Think it!

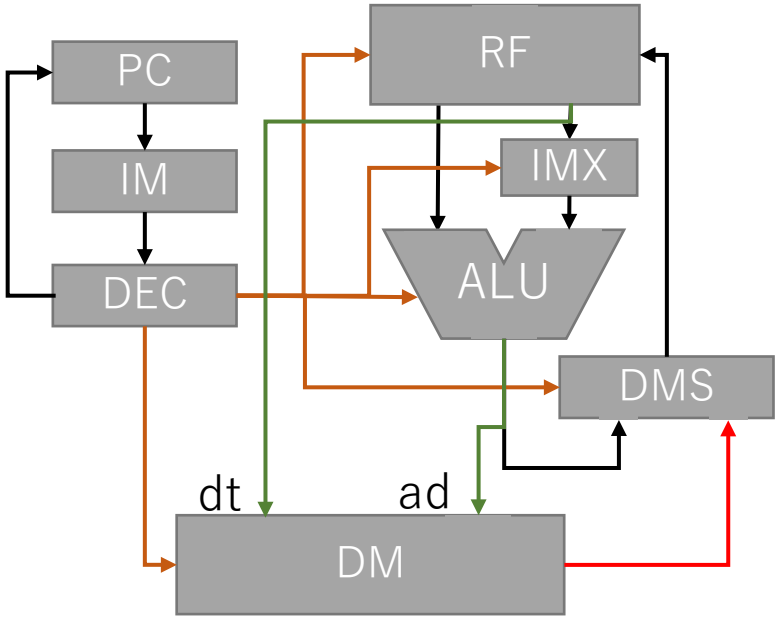
直値の利用

実際には直値で0x03というのを与えることは、上位8bitか下位8bitを設定できるだけなのでできない。どうすればできるようになるだろうか。上位が0であるという限定ならば解決できそう。

直値を捨てるという考えも全く問題ない。

インデックスアクセス

- ある場所を起点にして、そこから相対的に場所を指定する手法
  - メモリのアクセスは、ある基準となるアドレスからいくつ先のメモリをアクセスするケースが多い
  - C言語でa[3]は、*(a + 3)
- LM r0 = [r1+r2]
- LM r0 = [r1+0x0002]



（演習問題）
上記ニーモニックの動作を確認しよう

Think it!

データ側にALU？
・工夫によりレジスタと直値によるインデックスアクセスは可能になった
・レジスタとレジスタとの演算結果をアドレスとした書き込みはどうすればよいだろうか？
・この時機械語の語長はさらに増えてしまうが、何か工夫はできないだろうか？
　・RF書き込みアドレスの再利用・専用レジスタ
それがインデックス**レジスタ**の由来
・データ側で演算するメリットは？

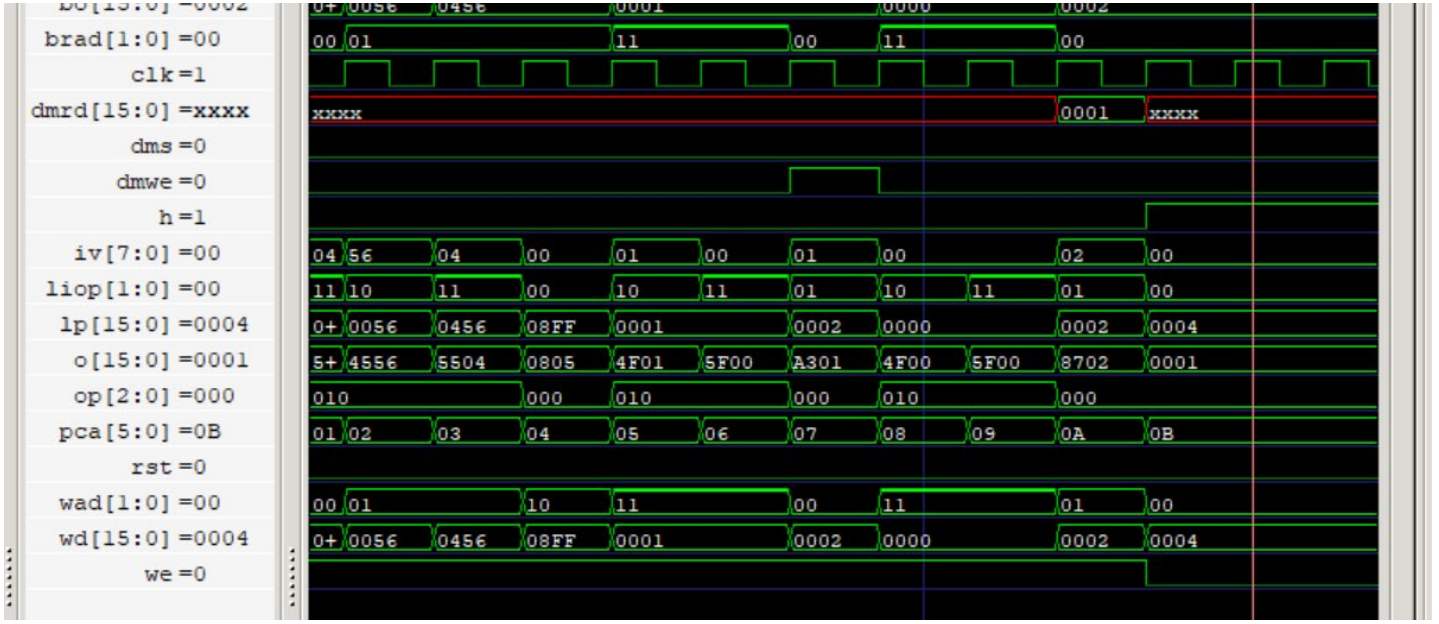
Complete it!

インデックスレジスタ込みの回路を構成し
実際に動作を確認しよう

デコーダの拡張と実行例


```
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 * * * 0 ; NOP
0 0 0 0 0 0 0 0 0 0 0 0 * * * 1 ; HALT
0 0 0 0 1 0 rw> im----------> ; LI rw,(s)im
0 0 0 1 0 0 b-> im----------> ; SM [(s)im]=rb
0 0 1 0 0 0 rw> op----> a-> b-> ; CAL rw=ra,rb
0 1 0 0 rw> b-> im----------> ; LIL rw,rb,im
0 1 0 1 rw> b-> im----------> ; LIH rw,rb,im
1 0 0 0 0 0 0 0 op----> a-> b-> ; SM [ra]=rb / SM [ra] = [ra op rb]
1 0 0 1 a-> b-> im----------> ; SM [ra + (s)im]=rb
1 1 0 0 rw> 0 0 im----------> ; LM rw=[im]
1 1 0 1 rw> F 0 op----> a-> b-> ; LM rw=[ra op rb]
1 1 1 0 rw> a-> im----------> ; LM rw=[ra + (s)im]
1 1 1 # rw> a-> im----------> ; CAL rw=ra,im (#=0:ADD/1:SUB only)
```

```
6'h00: 16'b0000_00_00_0_000_00_00; //NOP
6'h01: 16'b0100_00_00_1010_1001; //LIL r0,r0,8'b10101001
6'h02: 16'b0101_00_00_0000_0100; //LIH r0,r0,8'b00000100
6'h03: 16'b0100_01_01_0101_0110; //LIL r1,r1,8'b01010110
6'h04: 16'b0101_01_01_0000_0100; //LIH r1,r1,8'b00000100
6'h05: 16'b0010_00_10_0000_00_01; //ADD r2:=r0,r1
6'h06: 16'b0100_11_11_0000_0001; //LIL r3,r3,8'b00000001
6'h07: 16'b0101_11_11_0000_0000; //LIH r3,r3,8'b00000000
6'h08: 16'b1001_11_10_0000_0001; //SM [r3 + 00000001]=r2
6'h09: 16'b0100_11_11_0000_0000; //LIL r3,r3,8'b00000000
6'h0a: 16'b0101_11_11_0000_0000; //LIH r3,r3,8'b00000000
6'h0b: 16'b1110_01_11_0000_0010; //LM r1=[r3+00000010]
6'h0c: 16'b0000_00_00_0000_0001; //HALT
```



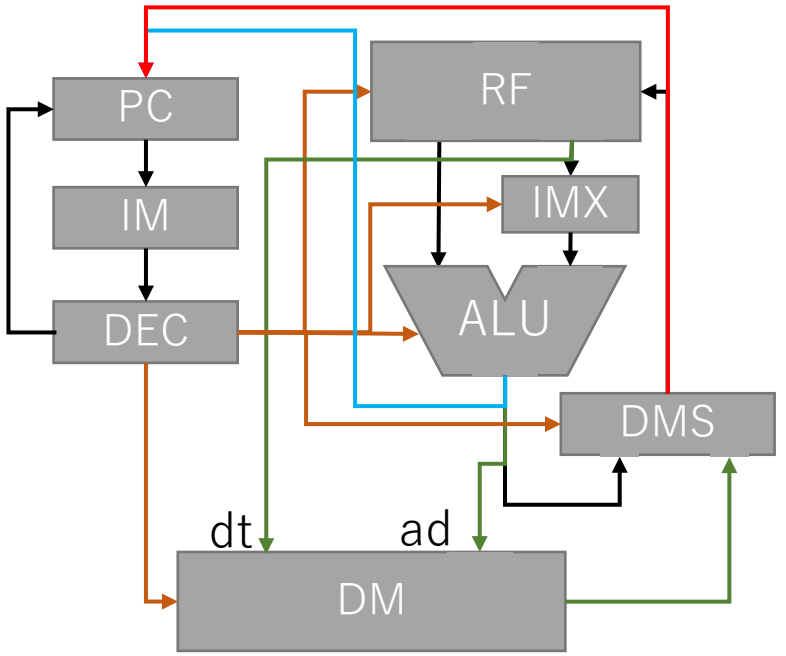
拡張を検討しよう

Think it!

- 0や1で変数を初期化することはよくある
  - ならば上位0で下位だけ直値は便利？
- 同様に直値との直接演算があると便利？
  - 例えば1足す・引く等の汎用演算など
- 直値は符号付として符号拡張して扱うべき？
  - 足し算だけで進む、戻るが自由にできる。
- 様々工夫してみよう

ジャンプ命令

- PCは単純に1増やすだけ
  →変更できれば実行順序を自由に変更可能
- そのようなパスを加える
  - どこに加えるかは自由
  - どこに加えるかで実現できる命令も速さも変わる
    - 青のパスならば計算結果で飛ぶぐらいまでできる
    - 赤のパスならばメモリの値を読み出した先に飛ぶまでできる
- 次のようなニーモニックを考える
  - JP r0 ; JP r0 + r1
  - JP 0x0003 ; JP r0 + 0x0010
  - JP [r0] ; JP [r0 + r1]
  - JP [r0 + 0x0010]



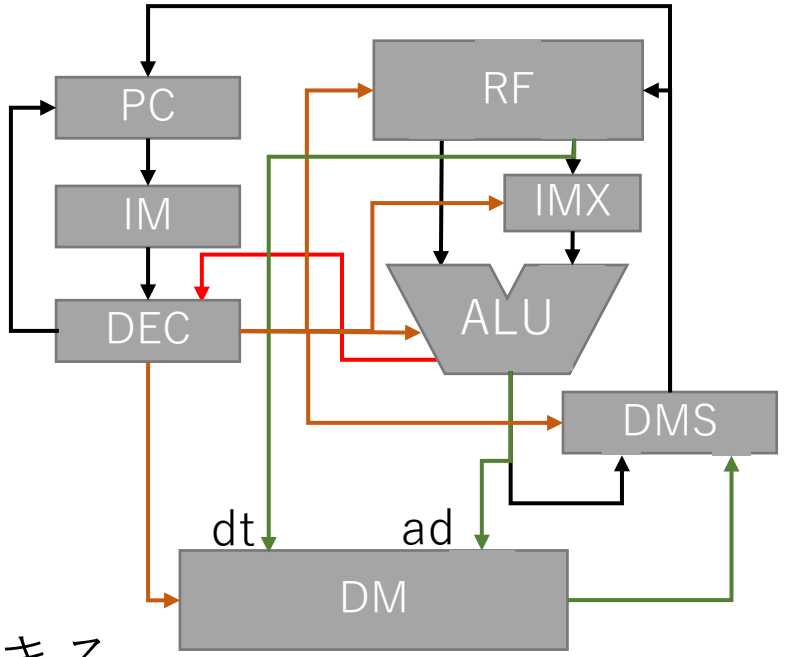
ジャンプ命令を実装する　Complete it!

自由度が高すぎて…　Think it!

CPUの設計は決められたものではなく、各個人が必要に応じて実験や制御のコントローラを実装するならば好きに記述してよい
ただし、実際にはコンパイラといった便利なツールがある方が楽なことが多いため、独自CPUはあまりお勧めできない
とはいえ、これらの構成の考え方は重要

比較命令

- 分岐がなければプログラムは書けない
  - ifとjumpがあれば何でも(面倒だが)実装可能
    - for, while, do-while, while, foreach, repeat全てifで変換可能
- 分岐は条件付きジャンプとして実装される
- では条件とは？
  - 直前の計算結果がどうか？によって動作を変える
    - 例えばA>Bならジャンプする A==0ならジャンプするなど
  - 引き算の計算結果が0か繰り下がりがあったかで判断できる
    - 引き算だけ行って答えを書き戻さないのもあり
  - SUB r0 = r1, r2
  - COMPARE/CMP r0, r1
  - JUMP_ON_ZERO/JZ r0
  - JZ r0+r1



Think it!

高級言語の変遷

BASICやFOTRANなど初期の高級言語は機械語と近い表現形式を持っていた。If(){}といった構造がなく、プログラムには行番号があり、if ～ then goto ～といった分岐ジャンプで表現されていた。その後構造化により設計しやすくなった。一方でマシン語との乖離は大きくなった

ステータスレジスタ(SR)

- 条件に何があればよいか
  - == 引き算結果がゼロ ZEROフラグ
  - ＞ 引き算結果が繰り下がらない　ボローた立たない
  - ＜ 引き算結果が繰り下がる　ボローが立つ
- 様々なSRのフラグ
  - ZF: Zero Flag = 演算結果がゼロ
  - CF: Carry Flag = 演算結果が溢れた
  - SF: Sign Flag = 演算結果が負
  - PF: Parity Flag = 演算結果のXORがゼロ
  - IE: Interrupt Enable=割り込み許可
  - OF: Overflow=符号付演算であふれ発生
- FFで同期させる
  - これで、次の命令に影響させることができる

```
logic [16:0] so, sa, sb; // 17bit化
assign sa = {1'b0, a} // {a[15],a}
assign sb = {1'b0, b}
always_comb
  unique case(op)
  4'b1: so = a + b; // ADD
  4'b2: so = a – b;
… //ここを押すとサンプルが手に入る↑
  {o,f} = {a,1'b0} >> b; // SR
… シフタは工夫が必要
  4'hf: so = a;
  endcase
always_ff @(posedge clk) begin
  if(so == 0) zf <= 1'b1; else zf  <= 1'b0;
  if(so[16]) cf <= 1'b1; else cf <= 1'b0;
end
```

Complete it!

分岐命令を実装しよう
ZFとCFを作成、DECが指令を出す際に利用してジャンプするかどうかを切り替える

Think it!

より使いやすく
ステータスレジスタ(状態レジスタ)を更新する命令と更新しない命令を定義すると使いやすい

演習問題（9）

- 既に構築したペラン数を演算する処理装置(PU)を、次の点で拡張したPUを実装しなさい
  - 直地代入、外部メモリ、デコーダ、ジャンプ、比較命令、条件ジャンプ
  - また、プログラムを書いていると今のプログラムカウンタ+3番地に飛びたいといった相対ジャンプが欲しくなるであろう。そこで、相対ジャンプを実装しなさい
- そのうえで、
  - ペラン数を16個計算し、総和を求めるプログラムを実行しなさい
  - 各ペラン数を順にメモリ上に書き込みなさい。
    - 0番地に1番目ペラン数、1番地に2番目ペラン数とし、16番目まで書き込む
  - 最後に、17番目に総和を書き込みなさい
- ポイント
  - 条件ジャンプを用いて、16個の計算と総和を求めるループを構成していることが必須です。同じ命令列を16個並べるなどは、これを満たしません。

演習問題（９）

- 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 で作成し、最初にタイトルを「演習問題（８）」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
    - コードは、設計したところのみテキスト情報としてそのまま張り付けてくれればＯＫです
    - 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)
    - なお、枚数は常識的なページ数であれば問題ありません
  - これらのフォーマットに従っていないレポート答案は受け取らない
  - 提出締め切りはLMSを確認すること

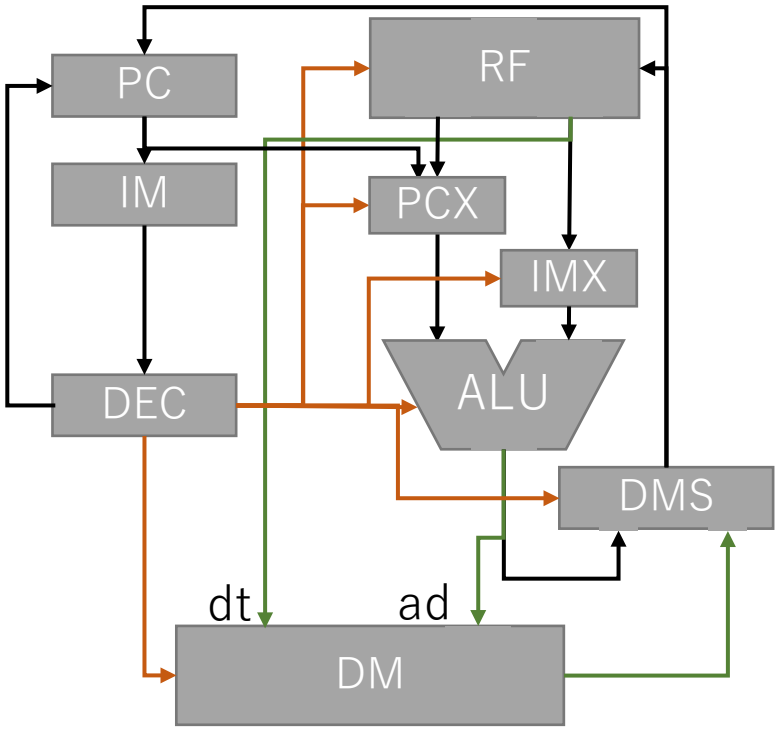
参考実装例

- 右のようなデコーダを構成
  - まだ拡張性は十分に残してある
- imは(s)imつまり、符号付
- opはALUへの命令
- ffは条件に用いる状態
  - 00:UC(無条件) 01:ZE(ゼロ)
  - 10:CA(キャリー) 11:SG(符号)
- pは条件でポジティブかネガティブか
  - 0:N(!=) 1:P(==)
  - 例えば、JP [r1]は、
    - p=P ff=NC、p=N ff=NC どちらでもよい
    - BR PZ [r1]つまり、ffがZEで、pが1の場合、
      ゼロフラグが立っていたら、という意味になる
    - BR NC [r1]つまり、ffがCAで、pが0の場合、
      キャリーフラフが立っていなかったら、という意味になる

```
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 * * * 0 ; NOP (0)
0 0 0 0 0 0 0 0 0 0 0 0 * * * 1 ; HALT (1)
0 0 0 0 0 0 0 1 op----> a-> b-> ; EVA CAL ra,rb/CMP ra,rb
0 0 0 0 0 1 rw> im----------> ; LI rw,(s)im
0 0 0 0 1 0 b-> im----------> ; SM [(s)im]=rb
0 0 0 1 0(p f f)op----> a-> b-> ; JP/BR pf [ra op rb] (ff:NC,Z,C,S)
0 0 0 1 1(p f f)im----------> ; JP/BR pf [(s)im]
0 0 1 0 0(p f f)im----------> ; JP/BR pf [PC + (s)im]
0 0 1 0 1 * rw> op----> a-> b-> ; CAL rw=ra,rb
0 1 0 0 rw> b-> im----------> ; LIL rw,rb,im
0 1 0 1 rw> b-> im----------> ; LIH rw,rb,im
0 1 1 0 rw> 0 0 0 0 0 0 0 S C Z ; LI rw,SR S:sign C:carry Z:zero
0 1 1 0 0 0 1 * op----> a-> b-> ; SM [ra]=rb / SM [ra] = [ra op rb]
1 0 0 0 rw> 0 0 im----------> ; LM rw=[im]
1 0 0 1 a-> b-> im----------> ; SM [ra + (s)im]=rb
1 0 1 0 rw> * 0 op----> a-> b-> ; LM rw=[ra op rb]
1 0 1 1 rw> a-> im----------> ; LM rw=[ra + (s)im]
1 1 0 # rw> a-> im----------> ; CAL rw=ra,im (#=0:ADD/1:SUB only)
1 1 1 a->(p f f)im----------> ; JP/BR pf [ra + (s)im]
```

参考実装例

- 右のようなコマンドも表現できる
- 実装は下記の図で十分
  - PC+直値にジャンプするためにはセレクタを追加する必要がある
    RFのAポートからALUの間に、PCを投入するセレクタを作成するとよい




```
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 * rw> 1 1 1 1 0 0 b-> ; MV rw=rb
0 0 0 0 1 0 rw> 0 0 0 0 0 0 0 0 ; RESET rw (LI rw=0)
1 1 0 0 rw> rw> 0 0 0 0 0 0 0 1 ; INC rw
1 1 0 1 rw> rw> 0 0 0 0 0 0 0 1 ; DEC rw
1 1 0 0 rw> ra> 0 0 0 0 0 0 0 1 ; INC rw=ra (rw = ra+1)
1 1 0 1 rw> ra> 0 0 0 0 0 0 0 1 ; DEC rw=ra (rw = ra-1)
0 0 0 0 0 0 1 0 0 0 0 1 a-> b-> ; CMP ra,rb (EVA SUB ra,rb)
1 0 0 0 rw> a-> 0 0 0 0 0 0 0 0 ; LM rw=[ra]
1 0 0 1 a-> b-> 0 0 0 0 0 0 0 0 ; SM [ra]=rb
0 1 1 0 0 0 1 * 1 1 1 1 a-> b-> ; SM [ra]=rb
```

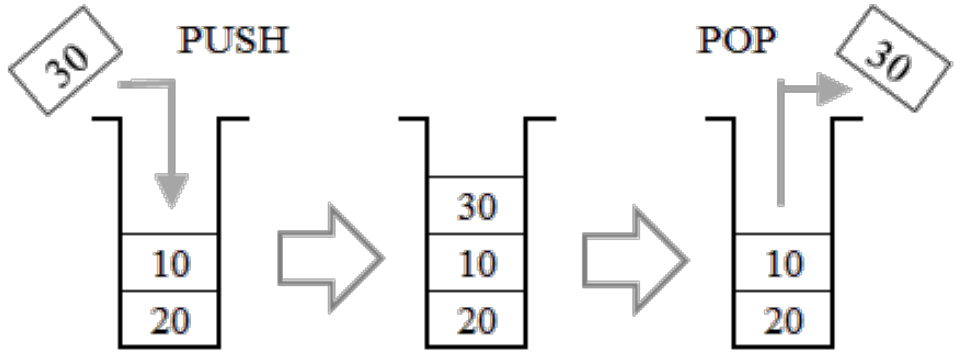
- 時間がない、面倒だ、難しいという人は、参考実装例から始めよう

できてしまうのだから仕方がない

- これらの例は「あくまでも一例」です
  - 自由な発想を期待します。動けばよいですし、最低限の命令を揃えればOKです
  - ０００００００１ op----> a-> b-> ; EVA CAL ra,rb (F=0)/CMP ra,rb
    - EVAはALUで計算した結果をRFに書き戻さないという命令です。最も重要なのは引き算で大小比較や等値判断に利用します。一般に比較命令CMPとして実装されています。他は無意味かもしれないが、できてしまうのです
    - ここではCMPはEVAの一態様で、EVA SUB ra, rb = CMP ra, rbとなります
  - ０１１０００１＊ op----> a-> b-> ; SM [ra]=rb / SM [ra] = [ra op rb]
    - RFのbポートがメモリのdtに入り、aポートとbポートのALUの演算結果をadに入れることができてしまいます。意味がない！と考えてSM [ra]=rbだけ実装するという方針は正解です。
    - 中には、行列などの初期化で、132+0番地に0、132+1番地に1、132+2番地に2などと書き込みたい時に効率化できるかも。PUのアーキテクチャ設計は自由度が高く、市販PUもそれぞれメーカが知恵を絞って、どういう命令をそろえるべきかで取捨選択した結果でしかありません。
  - ０１１０ rw> ０００００００ S C Z ; LI rw,SR S:sign C:carry Z:zero
    - ステータスレジスタ(SR) の値をレジスタに書き込む命令です。今回の仕様では必要ありませんが、CALLや割り込みでは欲しくなります。この場合はさらに逆に書き込む命令も必要ですね
    - さらにALUの状態レジスタ更新を阻害する（更新せずに保持する）工夫も必要ですね
  - 自由に、遊び心ももって、デコーダを設計してみてください

スタック

- メモリのあるアドレスをスタックレジスタsrで指しておく
  - 値やアドレスをスタックに詰めていく
  - PUSH 値をスタックに積む
  - POP 値をスタックから取り出す
  - 実際にはスタックの底はアドレス上位
- srを1足す1引くという特別なパスが必要
  - これをALUなどでも使いまわしたくなるので、++や--などの命令が存在
  - PUSHとPOPでは順番が逆となることに注意
    - PUSH: SM [sp] = r1; sp--
    - POP: sp++; LM r1 = [sp]
- 関数呼び出しも可能
    - CALL/CAL: SM [sp] = pc; sp--
    - RETERN/RET: sp++; LM pc = [sp]



スタックを実装しよう
レジスタファイルのr3をspと見做す形でも構わない

Complete it!

なぜスタックの底は上位？

Think it!

スタックの底はアドレス上位になっているのは、安全性を考慮した結果。なぜかを考えよう

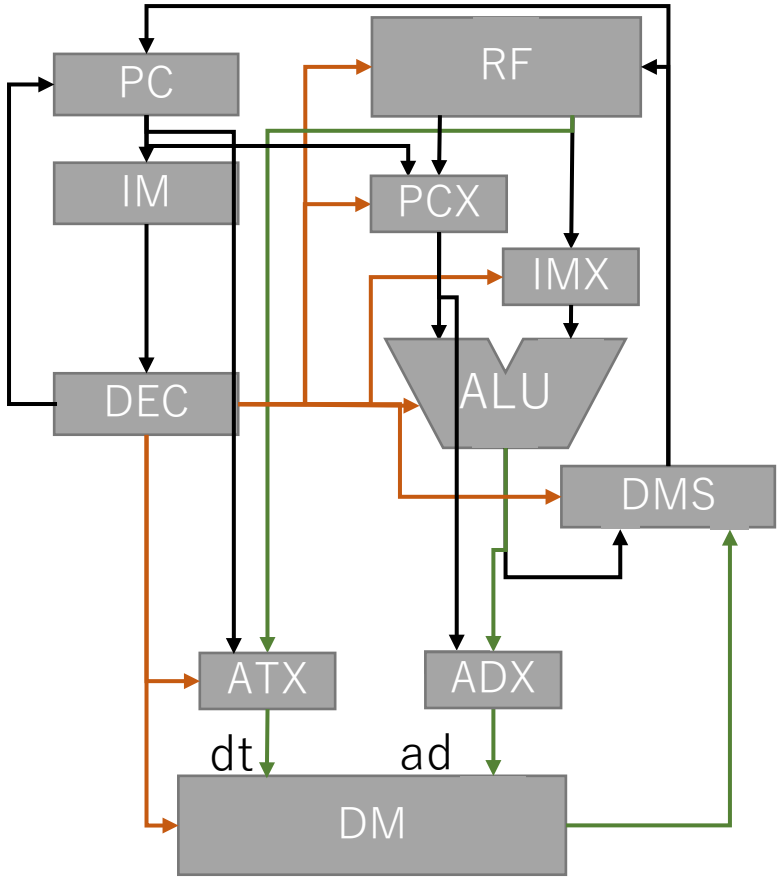
関数呼び出しするなら？

関数呼び出しするためには、様々な情報をスタックに詰めなければならない。なぜか？何か？を考えよう

参考実装例

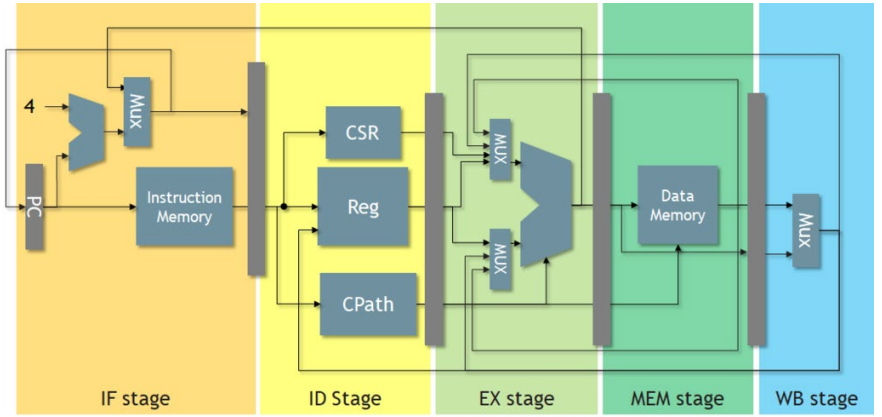
- ALUを用いて+1や-1を実現する
  - 計算順序があるため、新たにセレクタ ADX を設ける
  - PUSH: SM [sp] = r1; sp—
    - 後から引くため、ADXはALUからの出力ではなく PCXの出力を利用、ALUで-1を計算して書き込む
  - POP: sp++; LM r1 = [sp]
    - 先に引くため、ADXはALUからの出力を利用
  - CALL
    - 後から引くためPCXの出力を利用、PCXはPCを選択
  - RET
    - 先に引くため、ACXはALUからの出力を利用 PCXはPCを選択

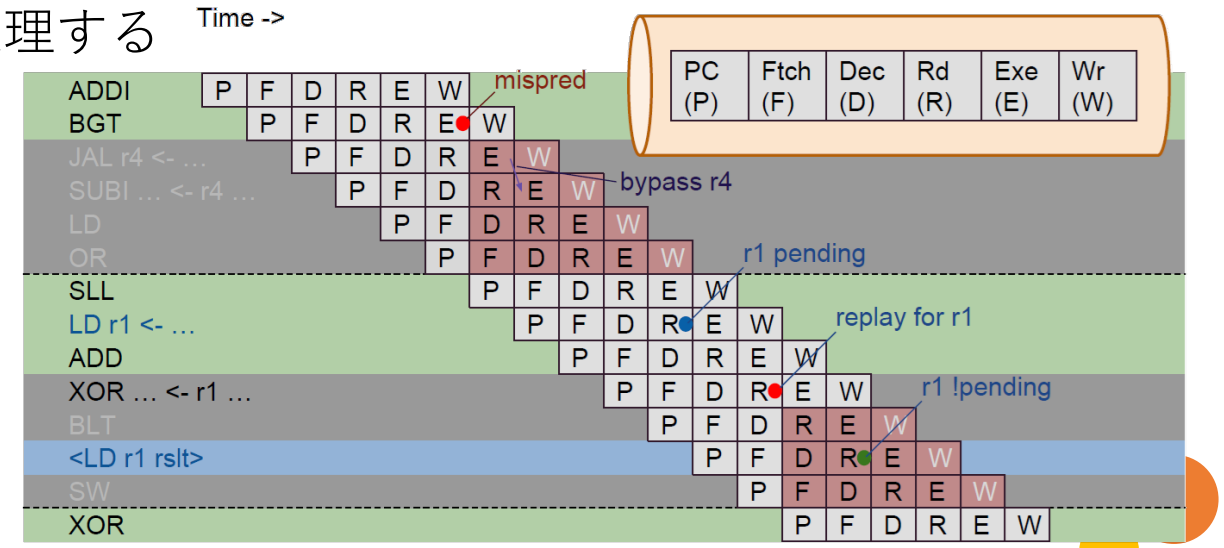
```
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 0 a-> b-> ; PUSH rb to ra
0 0 0 0 0 0 0 0 0 1 0 1 a-> b-> ; POP rb from ra
0 0 0 0 0 0 0 0 0 1 1 0 a-> * * ; CALL PC to ra
0 0 0 0 0 0 0 0 0 1 1 1 a-> * * ; RET PC to ra
```



割り込み・パイプライン

- 割り込み
  - 外部から割込信号でレジスタを設定、その間状態フラグは更新せず、halt状態
    - 緊急停止に近い。一般的に割り込みは割り込みジャンプを指す
- 割り込みジャンプ
  - 外部から割込信号でSRとPCをPUSHしてジャンプ
  - RETがきたら、PC、SRを戻して復帰
- パイプライン
  - さらに効率よく処理し性能を上げる手法
    - 各パスをFFで分断しクロックサイクル毎に処理する
  - 並列化により性能向上
    - 同じクロックサイクルで複数の仕事ができる
    - 処理遅延は増大する
    - 図はRISC-Vのパイプライン
  - 採用はアプリケーション次第





Memory Mapped I/O

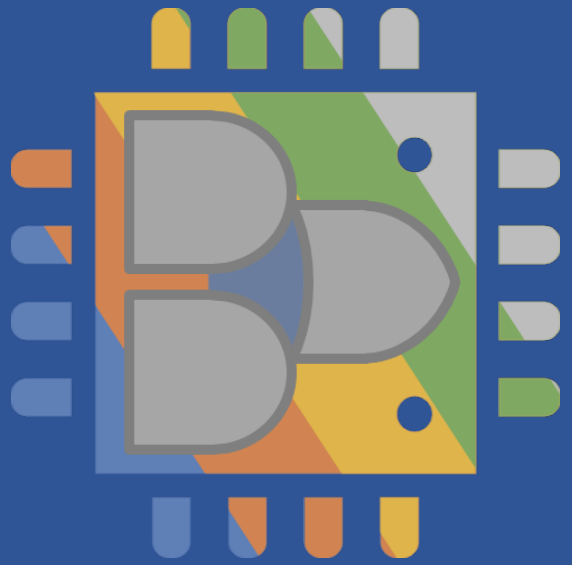
- 通常は外部にキーボードやスクリーン、モータやセンサなど、各種インタフェースを接続する
- どのように実現するのか？
- メモリの一部として割り当ててしまう
  - 例えば
    - 0x0000から0x0FFFまで実際にメモリが実装されており、
    - 0x1000をキーボードの入力文字(読み込みのみ意味がある)
    - 0x1001をモータへの出力ON/OFF（書き込みのみ意味がある）
  - などとして、アドレスデコーダ(AND)で選択、自由に入出力を組み込む
  - case文で実装できる

Memory Mapped I/Oの欠点 Think it!

ではMemory Mapped I/Oの欠点は？
他にどのような方法があるだろうか？
　ダイレクトレジスタ・ダイレクトパスなど

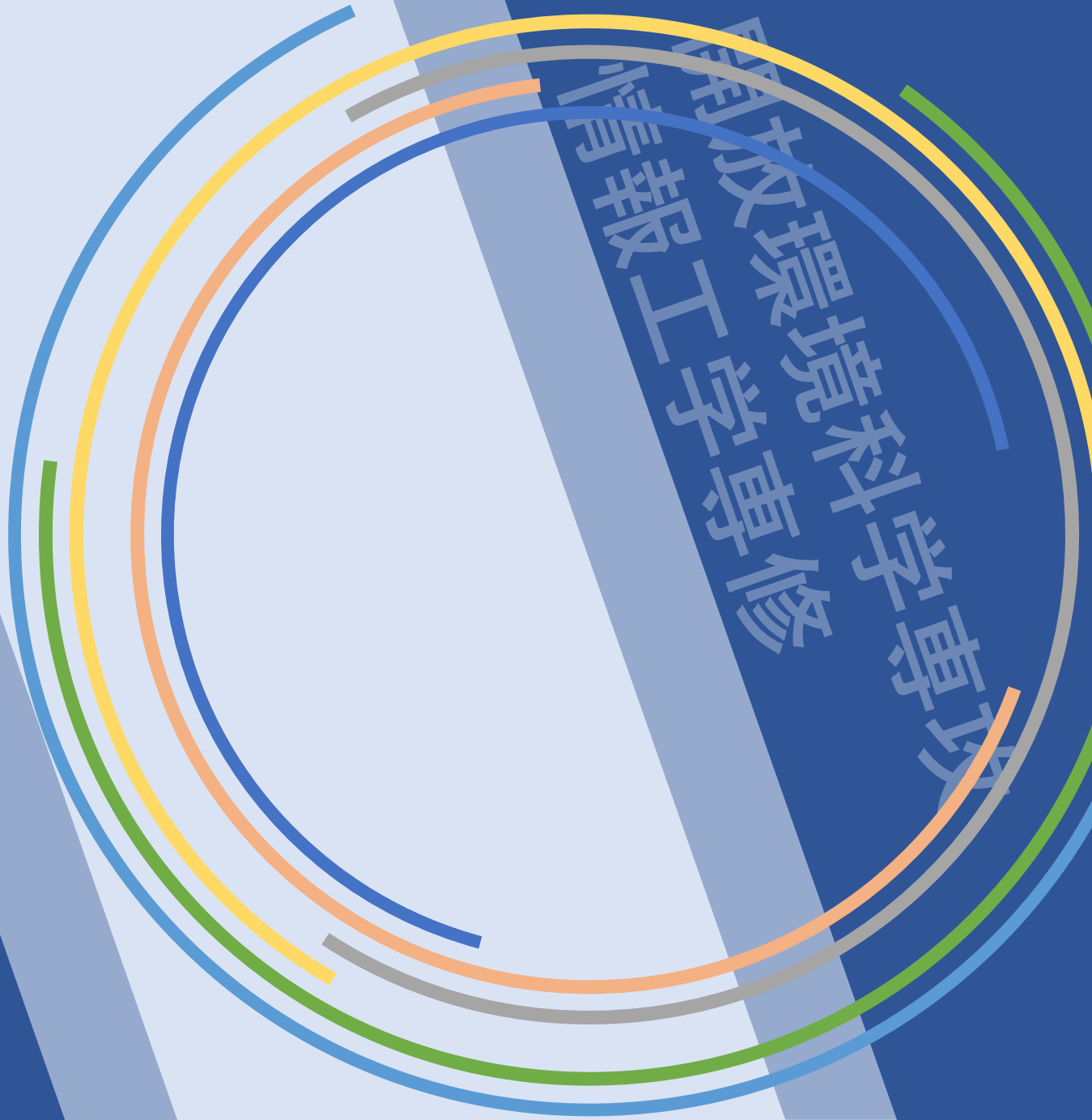
RISCとCISC

- 今回命令は固定長で、LIの実装の通り長いLIを無理やりLIHとLILに分けて詰め込んだ
  - これが可変長、つまり、LIを命令を2か所使うマルチワード命令とすれば問題なく実装可能
  - 一方で、可変長のデコーダは実装が面倒で、特にパイプライン実装が面倒
- CISC (Complex Instruction Set Computer)
  - 初めは分類など無く全て CISC
  - メモリが高価でその効率利用が強く求められた結果、命令に応じて語長を定め複雑な命令セットを準備することが善であった(当時のこの完成系が68000系プロセッサ、あまりにも美しい機械語)
- RISC (Reduced Instruction Set Computer) の登場
  - 1974年、IBMのジョンコックは「命令の20%がコンピュータの80%の処理を行う」と提唱、これを基に1990年、ヘネシーやパターソンによりRISCが考案
  - 命令の20%を如何に速く実行するかを追求、限られた命令を備え、複雑な動作は基本動作に分解し(Load Store アーキテクチャ)、語長をそろえてパイプライン化した
  - どうせコンパイラを利用するのだからユーザの設計コストに影響しないとした
  - この試みは大成功を収め、一時期はRISC一色に染まった
- 実際には
  - CISCは生き残っています
    - スーパスカラーやキャッシュなどが必須となりバランスが重要になった
    - RISCとて結局ニーズに答えるうち複雑度が増大した
    - 結局どれだけリソースをつぎ込めるかの勝負になり、最大手インテルが猛然とプロセッサ能力を高めた
  - インテルがRISCとCISCのいいとこどりでうまく逃げ切った



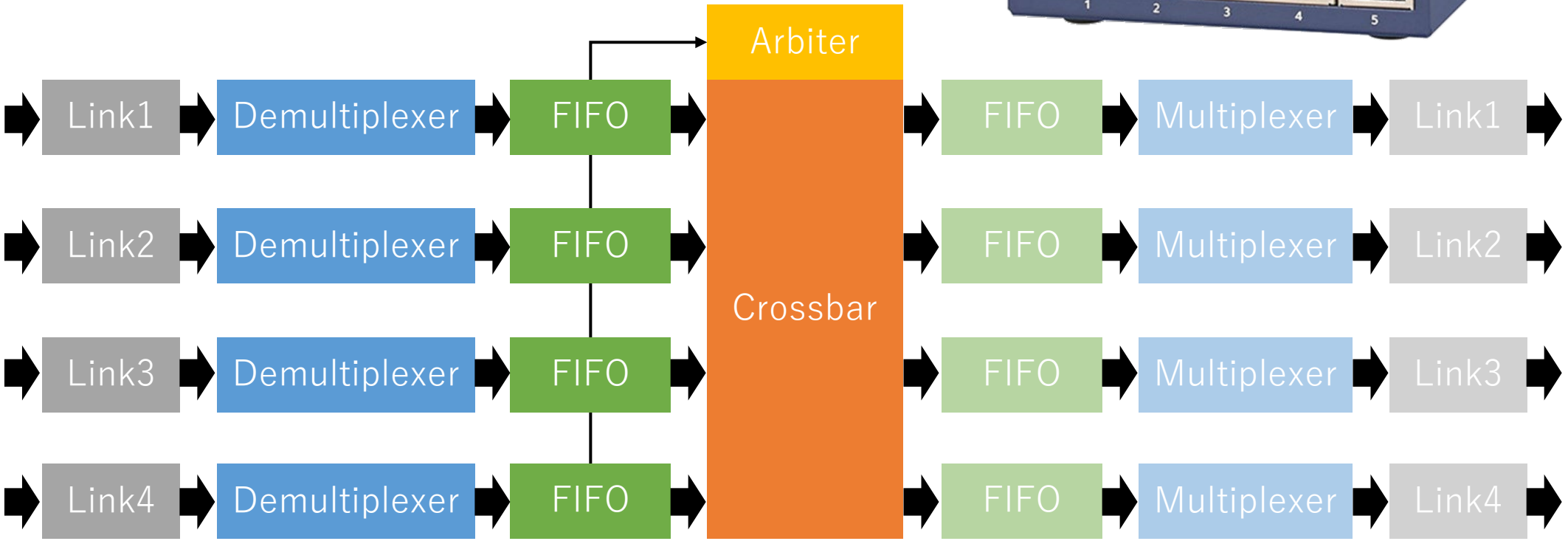
計算機システム設計論(10)
スタック
通信機器システム

担当：　西　宏章



スイッチ（最終課題）

- 計算機結合網の中心的ハードウェア

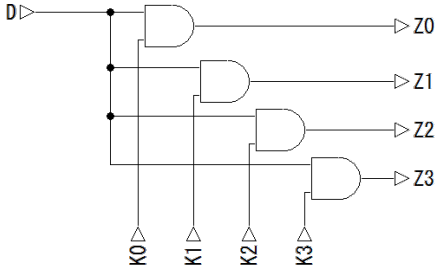


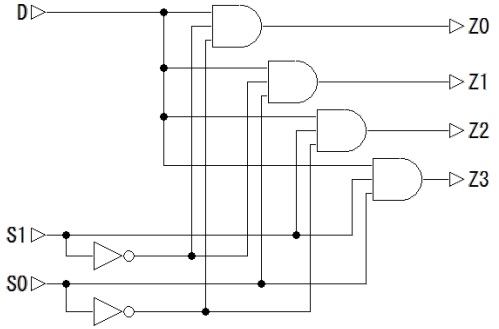
信号線入力　ビットレート ビット幅変換　VOQ 待ち行列　交換器と 調停器　VIQ 無い場合もあり　ビットレート ビット幅変換　信号線出力

高ビットレート 小ビット幅 ← 低ビットレート 大ビット幅 → 高ビットレート 小ビット幅

マルチプレクサ・デマルチプレクサ

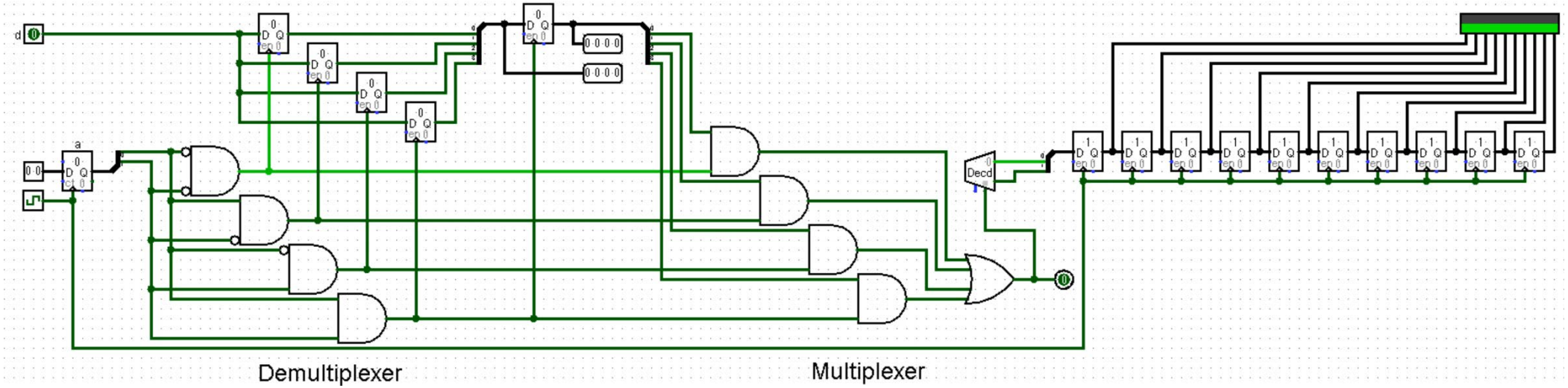
- 実は学習済みでシリアルパラレル変換の応用でシフトレジスタを利用
  - 本来のマルチプレクサ・デマルチプレクサは次の通り





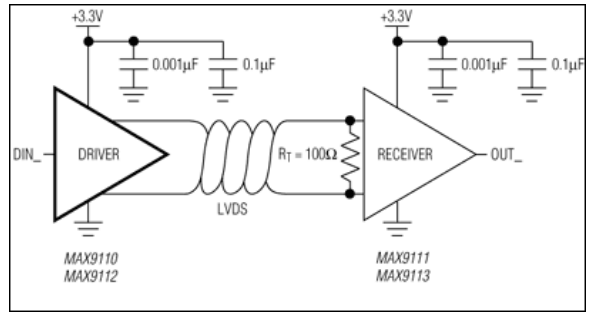
1をデマルチプレクスしたのが
デコーダ（下のDecd)
デコーダの出力で信号を選択すると
データセレクタになる

  - よく混乱されているが、実際にはシフトレジスタを利用するのが正解
    - 動作上の意味は一緒、 SerDes(Serializer/Deserializer)の方が一般的
    - なお、実機では例えばEthernet(IEEE 802.3)に適合するため様々な変換が行われる



交流符号化(ACカップリング)とDCバランス

- Cカット、つまりコンデンサで分断された通信路による低ノイズ高速通信路構築技術、時代はシリアル→差動→Cカットと推移
- なぜ通信できなくなるのか？
  - 長距離伝送すると、規準がわからない
  - 規準を伝えようとすると信号線が2本必要(GNDとシグナル)
    - この2本でノイズの入り方が違うとダメ
  - 電圧ではなく電流の向きで伝えよう(2本で+と-を入れ替える)
    - 差動はある程度うまくいったが、高速化に伴い信号線2本は無駄という判断になる
  - さらにクロックも必要、ところがクロックが別線なのでずれてしまう
- 信号線一本で通信しよう
  - ではどうやって基準を伝えるの？
    - 例えばずっと1やずっと0という信号を伝えることは不可能
    - ならば、常に0か1かパラパラ変化し続けていればよい、これがACコーディング
    - クロックも一緒に送ることができる
  - では0と1が適当にパラパラしていればよいのか？
    - そういうわけではない、0と1のバランスがとれていなければ、徐々に電圧が動いてしまう
    - ずっと0になるようにDCバランスを考える



8B10Bの例

- 8bitの情報を10bitの情報にして、必ず5bit以下の0/1の連続にしかならないようにエンコードし、さらに特殊コードも表現する手法

Table C-1: **Valid Data Characters**

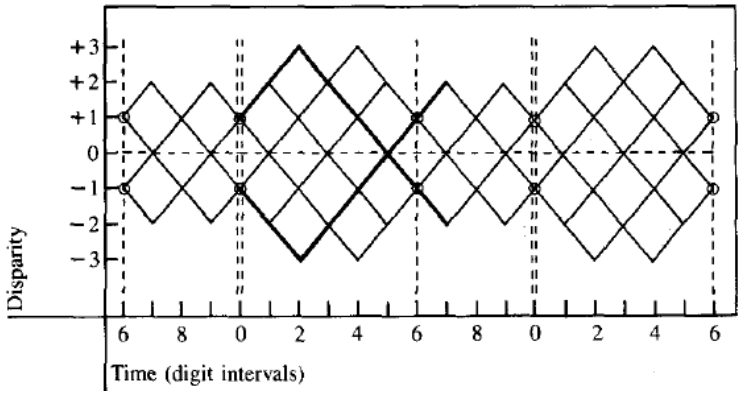
Data Byte Name	Bits HGF EDCBA	Current RD – abcdei fghj	Current RD + abcdei fghj
D0.0	000 00000	100111 0100	011000 1011
D1.0	000 00001	011101 0100	100010 1011
D2.0	000 00010	101101 0100	010010 1011
D3.0	000 00011	110001 1011	110001 0100
D4.0	000 00100	110101 0100	001010 1011
D5.0	000 00101	101001 1011	101001 0100
D6.0	000 00110	011001 1011	011001 0100
D7.0	000 00111	111000 1011	000111 0100
D8.0	000 01000	111001 0100	000110 1011
D9.0	000 01001	100101 1011	100101 0100
...			
D31.7	111 11111	101011 0001	010100 1110

Table C-2: **Valid Control K Characters**

Special Code Name	Bits HGF EDCBA	Current RD – abcdei fghj	Current RD + abcdei fghj
K28.0	000 11100	001111 0100	110000 1011
K28.1	001 11100	001111 1001	110000 0110
K28.2	010 11100	001111 0101	110000 1010
K28.3	011 11100	001111 0011	110000 1100
K28.4	100 11100	001111 0010	110000 1101
K28.5	101 11100	001111 1010	110000 0101
K28.6	110 11100	001111 0110	110000 1001
K28.7[1]	111 11100	001111 1000	110000 0111
K23.7	111 10111	111010 1000	000101 0111
K27.7	111 11011	110110 1000	001001 0111
K29.7	111 11101	101110 1000	010001 0111
K30.7	111 11110	011110 1000	100001 0111

0101010011011001110000010101101110100101001001110100110010011101001100111001

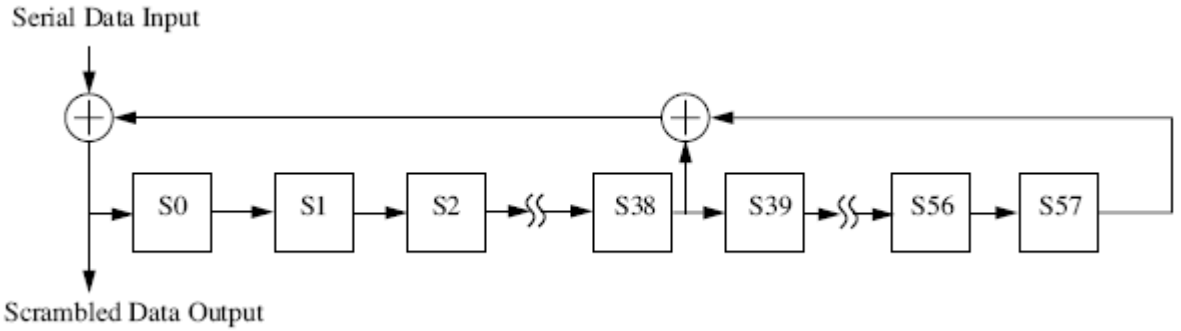
??.?　K28.5 D11.3　D16.2　D31.3　D0.0

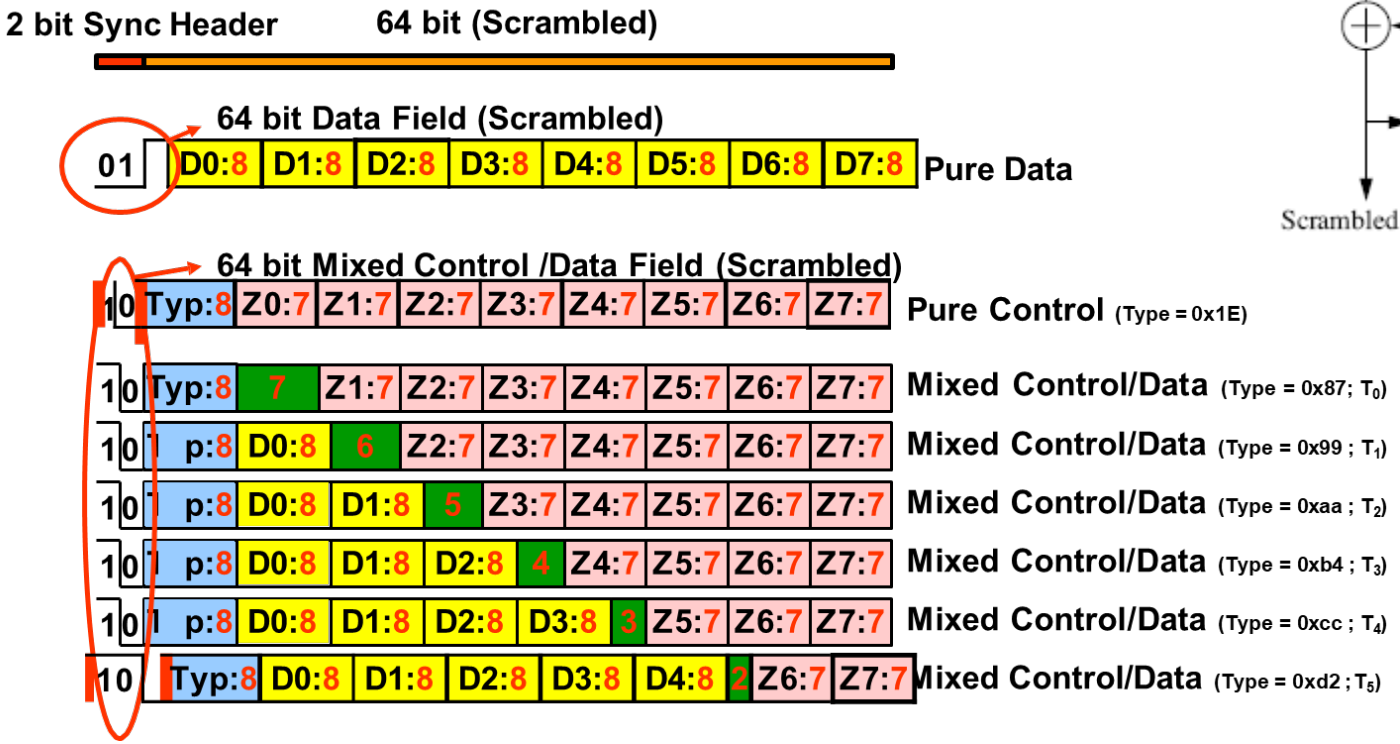


- さらに自己同期型

64B66B

- 8B10Bだと20%も無駄、64B66Bならば3%、128B130Bもある
  - たった3%足すだけで交流符号化できてDCバランスがとれるのか？
- そもそも厳密に守ることはやめてしまった
  - 確率的に0や1が並んでしまうことはあるけど、エラーの確率よりも低ければよい
  - $X^{58}+X^{39}+1$を用いて、CRC演算を行う





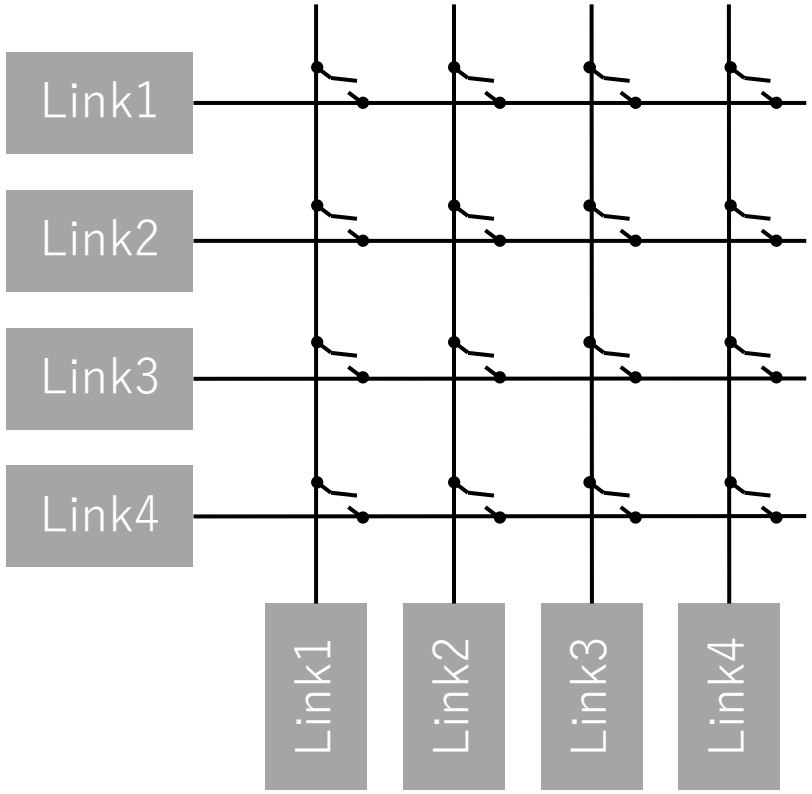
データを詰め込むため、かなり変態、狂気の沙汰な変換を行う

8B10Bと64B66Bなど新しい方式の比較

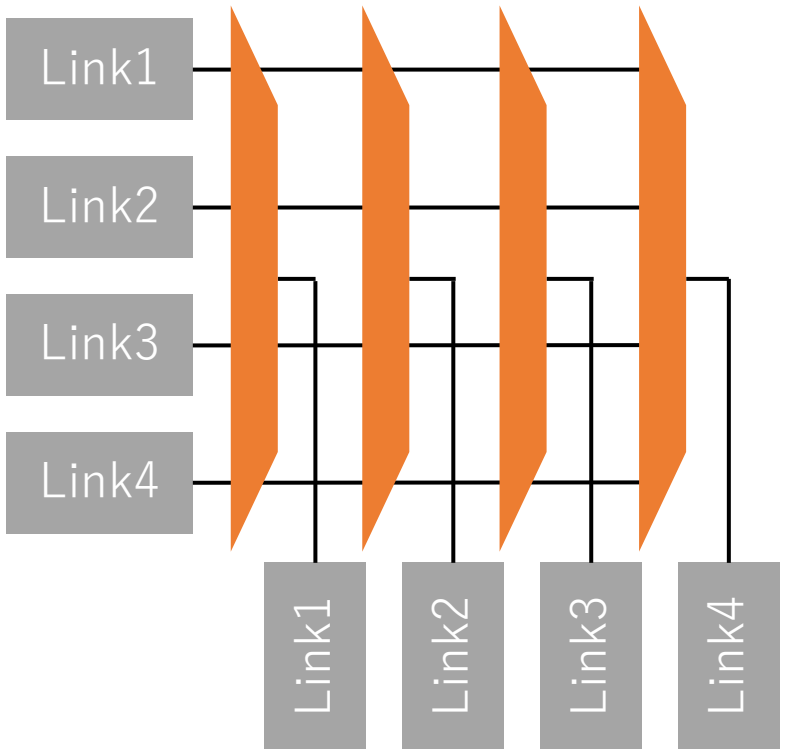
	8B10Bなど テーブル変換	64B66Bなど スクランブル変換
ランレングス（連続数）	最大5	確率的にはいくらでも
DCバランス	完璧	保証はしていないが、確率的にはバランスする
ビット同期クロック抽出	完璧	保証はしていないが、ともかく66bit毎見ていれば、いつかは同期する
ワード同期	Kキャラクタを使う	同期ヘッダで判断する
制御文字	Kキャラクタを使う	コントロールコードを用いる

Crossbar

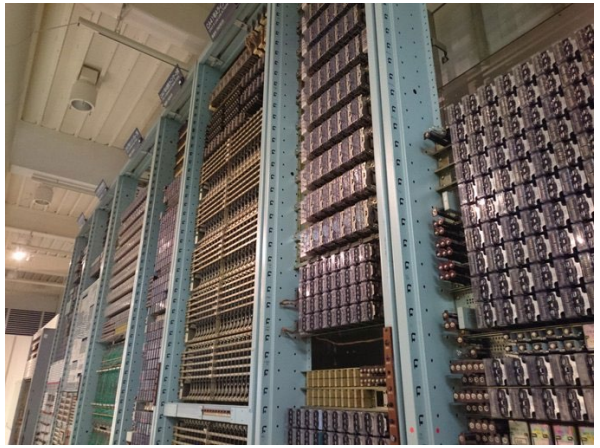
- 各接点にスイッチがあり全入力と全出力を自由に結合
  - どのような接続も実現
  - 可能な限りすべての接続を同時に実現＝ノンブロッキング網
  - 実はユニキャスト以外にマルチキャスト・ブロードキャストも可能



メカニカルなクロスバ



データセレクタで構成したクロスバ



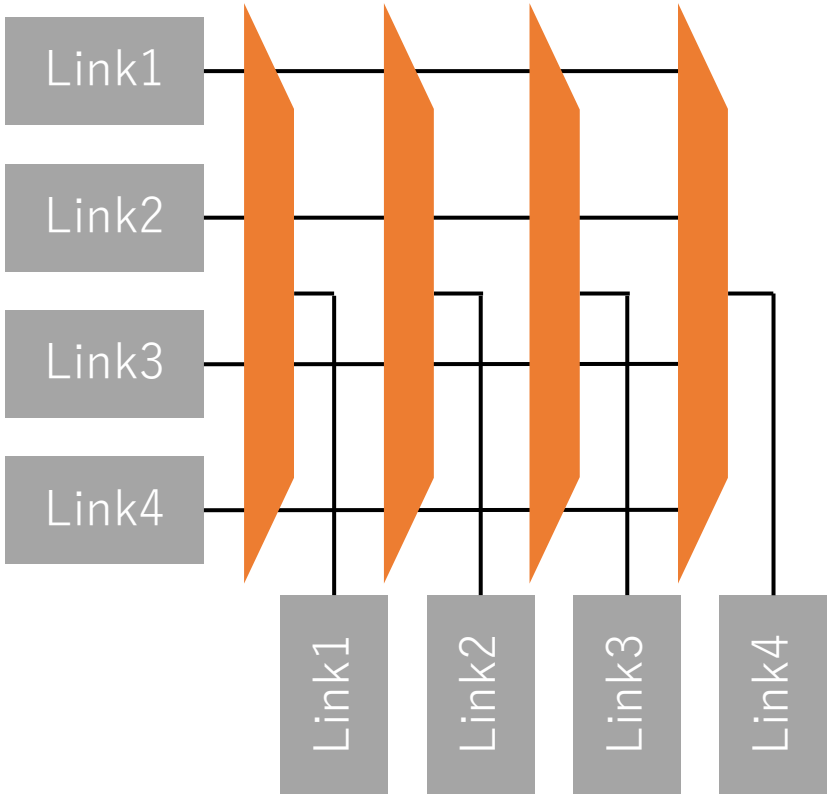
クロスバーの記述例

- クロスバーモジュール
  - 今回は、 4x4スイッチを構成する
- 実装においてマルチプレクサ、デマルチプレクサは省いて考える
- 以下次の定義を用いる

```
`define PKTW 9
`define PORT 3 // 3:0 // Number of Ports
```

- 参考例

```
module cb(input [`PKTW:0] i0, i1, i2, i3, output [`PKTW:0] o0,
o1, o2, o3, input [`PORT:0] d0, d1, d2, d3);
        cbsel cbsel0(i0, i1, i2, i3, o0, d0);
        cbsel cbsel1(i0, i1, i2, i3, o1, d1);
        cbsel cbsel2(i0, i1, i2, i3, o2, d2);
        cbsel cbsel3(i0, i1, i2, i3, o3, d3);
endmodule
```

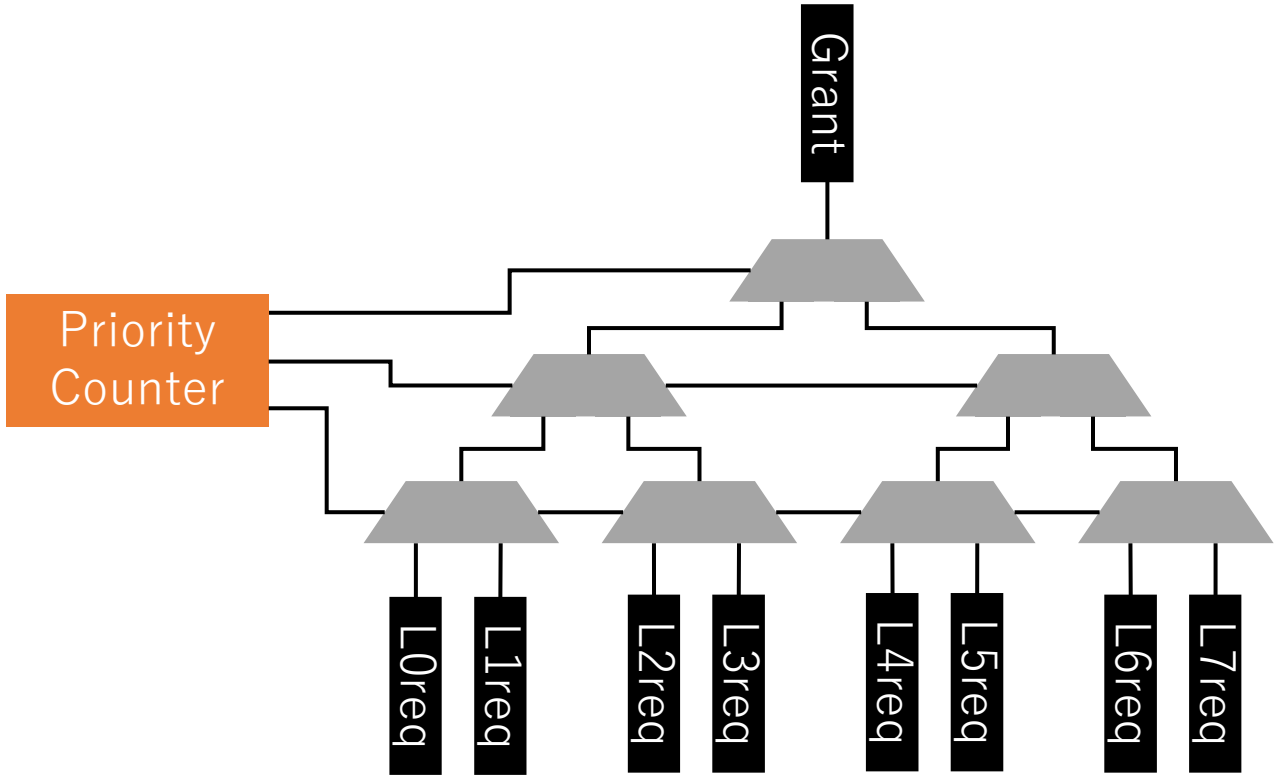


Arbiter

- 複数のリンクが同じ出力を要求すると競合(出線競合)するがこれを解決
  - 各リンクは、各入力からデータが入ると要求(Request)をアービタに出す
  - アービタから送信許可(Grant)を受け取って初めてデータを送信できる
  - どのようにGrantを生成するかが、Arbiterの設計
- Fixed Arbiter
  - 要求を固定優先順位で決める。シンプルだが公平ではない
  - この場合特にプライオリティエンコーダと呼ぶ(ifの実装でも説明済みでifで書ける)
- Round-Robin Arbiter
  - 優先順位を順番に回す。公平なアービタの一つ
  - Fixed Arbiterをリンク数だけ構成し、Grantを出すたびに切り替えることで実現
    - 他にも様々な手法がある
- 4x4しかないので、0123, 1230, 2301, 3012の4パターンで優先順位が決まる回路を構成し、これを順番に利用するという回路で十分

(余談) Tree Arbiter

- 1995年提案・実装したRound-Robin Arbiter構築方式
  - Priority CounterはGrantを出すたびにカウントを1増やすWrap-Roundカウンタ
    - データセレクタが木構造で連結しておりデータセレクタの選択がPriority Counterにより決定
    - カウンタの値と最優先順位、さらにそれ以降の優先順位も順番に遷移するため公平



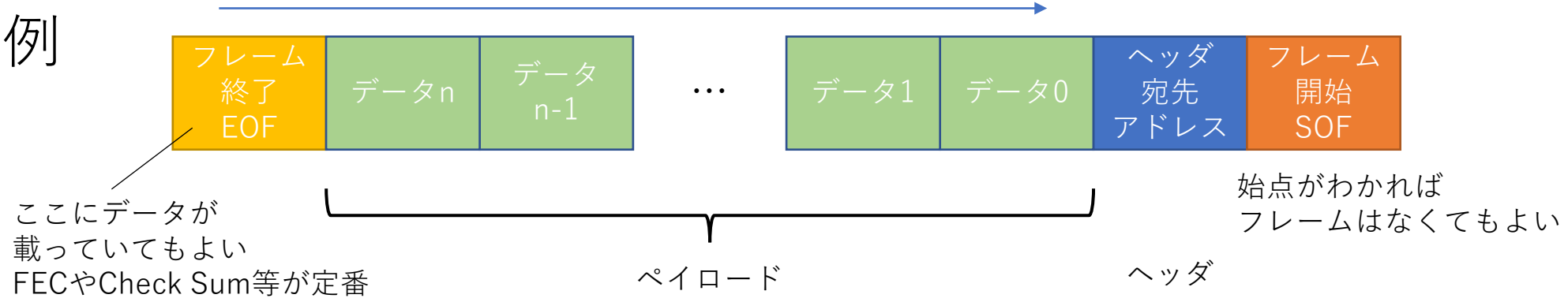
Think it!

2^nではない場合は？

確かに2^nの時完全に公平になる
それ以外は準公平である。なぜだろうか？

パケット様式

- ネットワークはスイッチ設計とパケット設計が重要
  - CPU設計と命令設計と同じ
- Packetを基本としてパケットを細かくしたのがFlit、パケットを纏めるのがフレーム、パケットやフレームの集合体がストリーム
- 典型的な例



- フレームの識別
  - 8B10Bなど交流符号化のSOF/EOFを利用
  - 1bit/2bit追加して対応
    - 1bit:Valid/Invalid
    - 2bit:Special Character + SOF/EOF

Think it!

Wormhole Routing

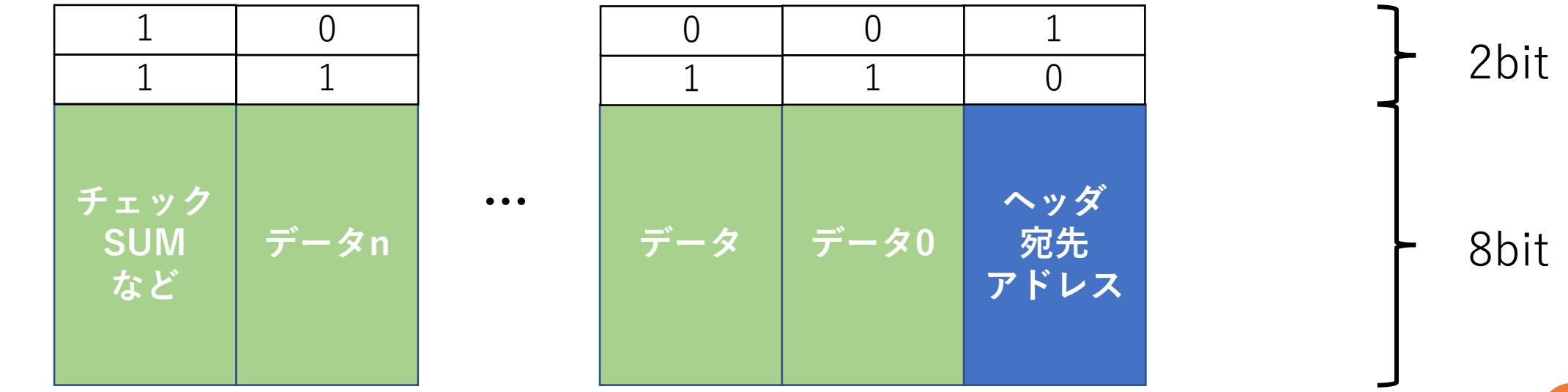
フリット毎にすべての情報が含まれてているか、ヘッダをコピーするかして、パケットを分割し、複数のスイッチにまたがってルーティングするこのような効率化手法が存在するが、他にどのような手法があるだろうか？

Flit構成の例

- 今回はSOF,EOFは用いず2bitの識別子を加えて次のようにFlitを定義する

```
`define HEAD 2'b10 // Flow bit type
`define BODY 2'b01 // Flow bit type
`define TAIL 2'b11 // Flow bit type
`define EMPT 2'b00 // Flow bit type
`define FLOWBH 9 // Flow Bit High
`define FLOWBL 8 // Flow Bit Low
```

- フレーム識別子を用いて、ヘッダ、ペイロード、トレイラ、ギャップを判断する
- ヘッダには例えば宛先ポート番号を記載する



ルーティング

- ソースルーティング(Source Routing)
  - 首切りルーティング(Decapitation Routing)とも呼ばれ、ソースつまり送り元が全てのパケット配信経路を指定する
  - 例
    - 送り元は、テーブルを引いて宛先に到達するに必要な情報として、3,1,4を得る
      - これを用いて、それぞれ、3, 1, 4と記載したヘッダを3つ作成してパケットを構成する
    - 先頭のヘッダに3と記述されているので、先頭を取り除いて3番ポートから出力する
    - 次のスイッチでは、先頭に1と記述されているので、同様に1番ポートから出力する
    - 次のスイッチでは、先頭に4と記述されているので、同様に4番ポートから出力する
    - つまりパケットはどんどん短くなる
- テーブルルーティング(Table Routing)
  - Destination Routingとも呼ばれ、各スイッチが宛先番号でルーティングテーブルを検索し、どこから出力するかを決定する
- いずれもルーティングテーブルをあらかじめ設定しておく必要がある
  - ここでルーティングプロトコルの話になるが、この授業の範疇ではない

マルチキャスト・ブロードキャスト

- 実はクロスバーは1対多対応
  - 1対1のユニキャスト、1対多のマルチキャスト、1対全のブロードキャストが可能
- つまり、1番ポートの入力パケットが、1, 3, 4番ポートへ出力したいといった1対多のスイッチングも可能
  - この時パケットの内容はコピーされる
- この時、クロスバのアービタがデッドロックしないように気を付ける
  - 例えば、
    - 1番ポートの入力パケットが1, 2, 3, 4番を要求、
    - 同時に2番ポートの入力パケットも1, 2, 3, 4番を要求
    - 1番ポートの入力パケットが1, 2番を獲得済みで、3, 4番の解放待ち
    - 2番ポートの入力パケットが3, 4番を獲得済みで、1, 2番の解放待ち
  - 回避方法
    - 全てのアービタの優先順位を揃えて、切り替えタイミングを工夫する
      - シンプルだが公平性と効率が落ちる可能性がある
    - 獲得できたところだけ転送し、分割や繰り返して転送する（普通は選択しない）
      - 遅延が大きくなる、特殊なFIFOにする必要がある、といった問題がある

最終演習問題

- 4x4のスイッチを構成し、実際にパケットを流して動作を確認
  - 必ず複数の異なるパケット長の動作を確認すること
    - また、バイト文字を送って、正しくバイト文字が受け取れることを確認すること
  - 必ずリンク数は4以上にすること
  - 必ず出線競合状態を確認し、アービタで解決できていることを確認
  - フレームのフォーマットや宛先の情報などは自由に設計してよい
  - 必要最低限の実装でよく、アービタも自由な設計でよい
  - yosysの合成結果を示しなさい（最後の参考例に倣って回路規模だけでよい）
- 発展演習問題（次のどれかに挑戦）
  - SerDesの実装（見ずらいから、あまり期待していない拡張）
  - ルーティングテーブルの実装
  - 3つ以上のスイッチをつなぎ合わせて動作することを確認
  - デッドロックが発生することを確認（つまりフロー制御を実装すること）
  - マルチキャストサポート
- スーパー演習問題（次のどれかに挑戦）
  - デッドロックを回避する機構を搭載（例えば仮想チャネル）
  - CPUと連携して動作させる
    - CPUのMemory Mapped I/Oでパケットを順番に書く、データメモリの内容をパケットにして送るなど

最終演習問題

- 注意事項
  - レポートをwordファイルとコードのzipファイルで作成、LMSへ提出すること
    - 説明文をMicrosoft Wordファイルで作成する
    - **verilogソースには全て、コメントで先頭に学籍番号・ローマ字氏名を記入しておくこと**
    - **ソースプログラムとMakefileをzip圧縮**して、別途LMSで提出
      - 当方がmakeでコンパイルし、実行、合成できるように、Makefileを入れておくこと
      - wordファイルとzipファイルは提出場所が違いますのでご注意ください
    - zipには、**課題の確認項目を動作するテストモジュールおよびテストベクトルを添付**
    - 適切にvcdファイルが生成できるようにすること
    - シミュレーションを行い、動作の証拠としてgtkwaveの画面をキャプチャしWordに張り付ける
    - **合成できるようにしておくこと**
  - 標準問題への回答と動作を基準に採点する
    - 挑戦問題にトライした場合は、何にトライしたか、どのように設計したかを明記すること
  - wordファイルの最初にタイトルを「最終演習問題」とし、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

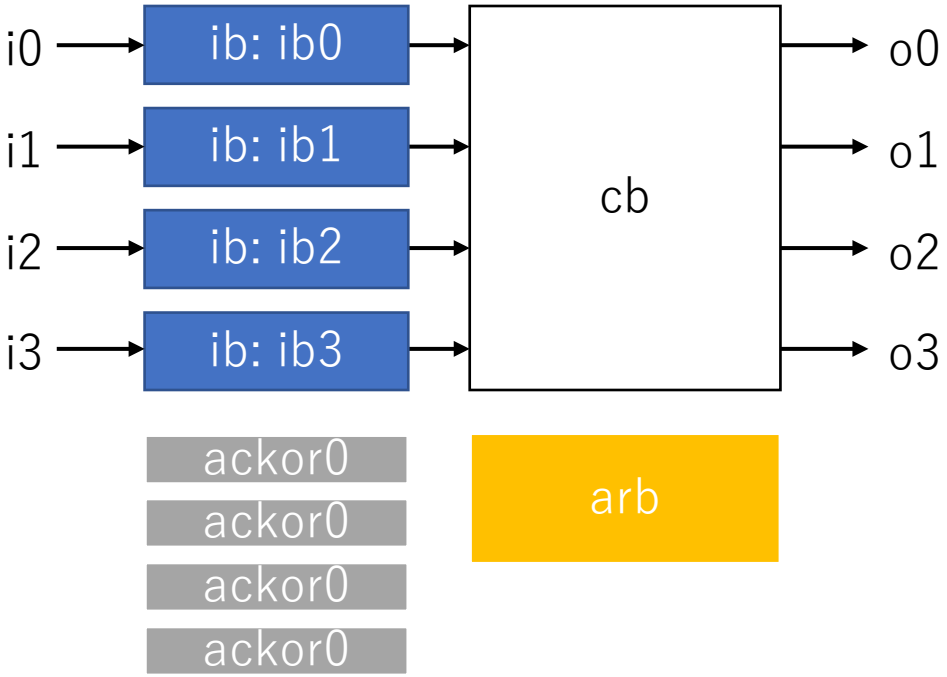
最終演習課題を首尾よく進めるためのヒント

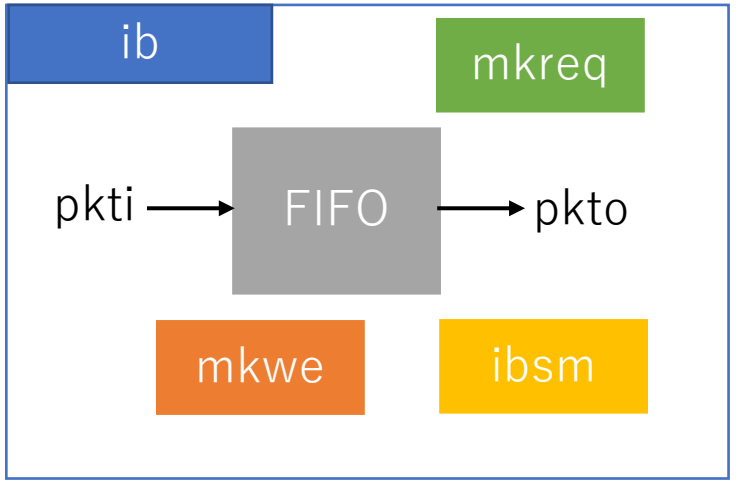
- 次の前提に従うと比較的楽に設計できる
  - パケットはデータ8ビット、フレーム指定2ビットとする
    - Packet Bodyが4つの場合で、宛先が3の場合
      9'b00_xxxxxxxx　　// データが何もない状態を指す。xはなんでもよい
      9'b11_00000011　　// ヘッダフレーム、LSBの2ビットに宛先が記載されている
      9'b01_dddddddd　　// ペイロード、dは何でもよい。ここでは文字コード
      …
      9'b01_dddddddd
      9'b10_dddddddd　　// トレイラーフレーム、ペイロードの一部でよい。
      場合によってはFECやチェックサム等
- まずはノーマルに、最低限仕様で作ること！
- タイミング問題が厄介
  - タイミング設計をちゃんとやらないから
    - Cのプログラムとの違いはここ！スイッチ設計が最もハードウェア設計の醍醐味が理解できる
    - どのクロックで何をするかのチャートをきちんとと考える。これをさぼると大変になるだけ
- スイッチの性能にこだわると、タイミング設計がかなり面倒になる
  - こだわらずに、確実に処理を刻むとよい
  - HPC向けスイッチはこのタイミング処理を限界まで詰めている

最終演習課題を首尾よく進めるためのヒント

- 面倒くさがるとはまる
  - 各モジュールについてテストモジュールを書いて、簡単な動作チェックをすること
  - スイッチ設計は、沢山の信号が入り乱れるので、くみ上げると厄介
- スイッチ設計はループが発生しやすい
  - シミュレーションが固まったらループ発生を疑う
    - CPUは完全同期で相互依存が少ないためループが発生しにくい
  - vvpの実行が固まったら
    - Ctrl-Cで停止する
    - finishと入力して終了する
      - helpとするとvvpのコマンドヘルプが出る
    - gtkwaveで動いているところまでの波形を確認する
  - ループの原因を追究する、回避するには
    - 多くはrequest、ackのループ
    - このループのどこかをFFでラッチする

モジュール構成図（詳細配線は省略）

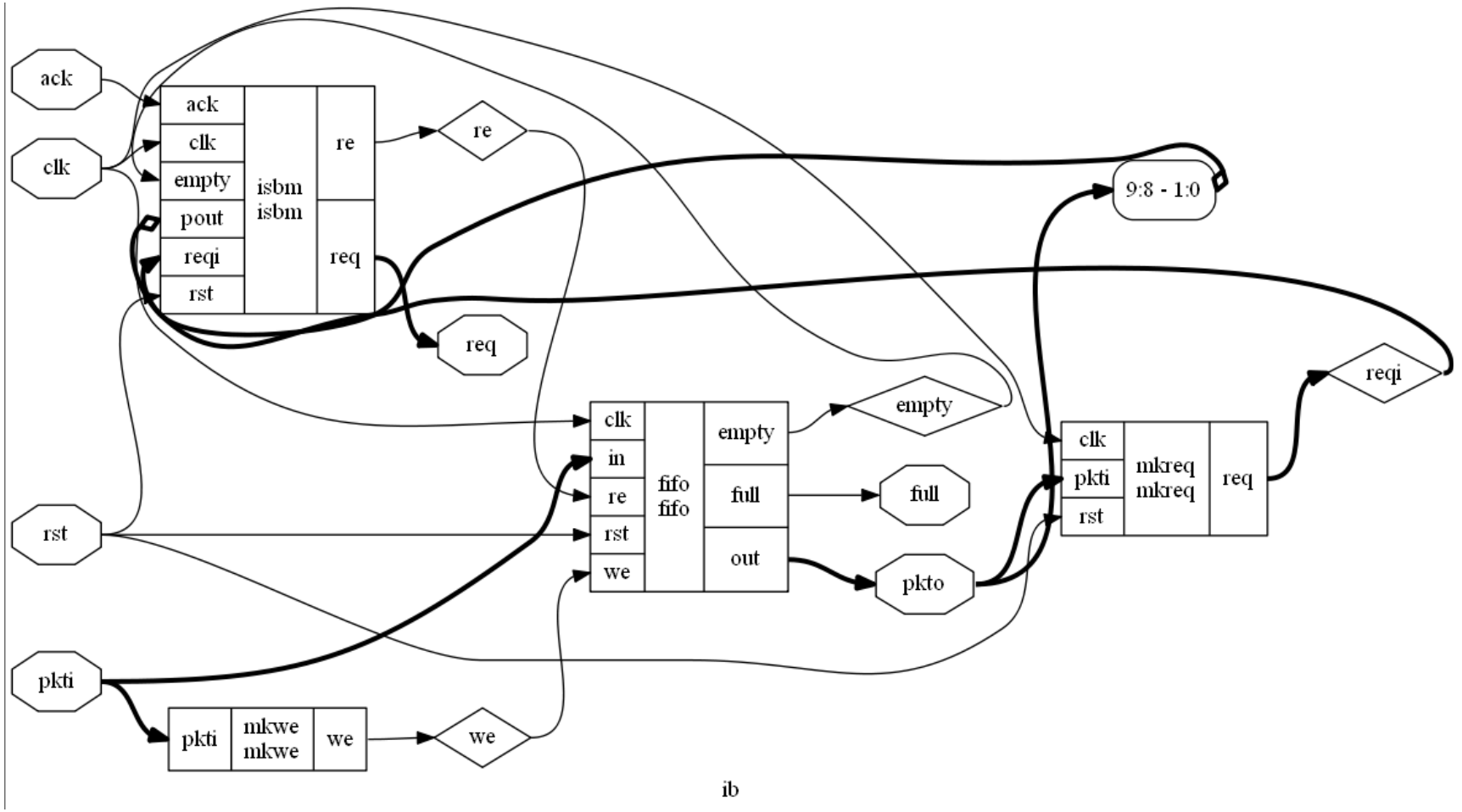




ib: Input Buffer 入力にあるパケット制御モジュール、ほぼすべての制御を行う
cb: Crossbar クロスバー、中身は4:1のセレクタであるcbselが4つ
arb: Arbiter アービタ、手前と出口で配線の付け替えが必要となることを忘れないように
ackor: Or of Acknowledge ACKがあったことをORを取って1つの信号線にする。マルチキャストの際は工夫必要
fifo: 授業の通りのfifoだが、サイズの調整とemptyの時出力が0になるように調整
mkwe: fifoのweを生成する。ハンドシェイク(フロー制御)する場合は工夫必要
ibsm: すべての要、fifoの出力を見ながら、Req、Ackの管理や、reの制御などを行う、なにをするにも工夫必要
mkreq: リクエストの生成、2ビットの宛先から4ビットを作るデコーダ。マルチキャストしないので必要

ibの合成結果

- ibにあるステートマシンのタイミング設計は慎重に行うこと
  - 「初期（何もなし） → リクエスト出してACK待ち → 転送 → 初期」など
  - リクエストは一度出したら落とさない！転送が終わったら必ず落とす！



ib

トップモジュールの参考例

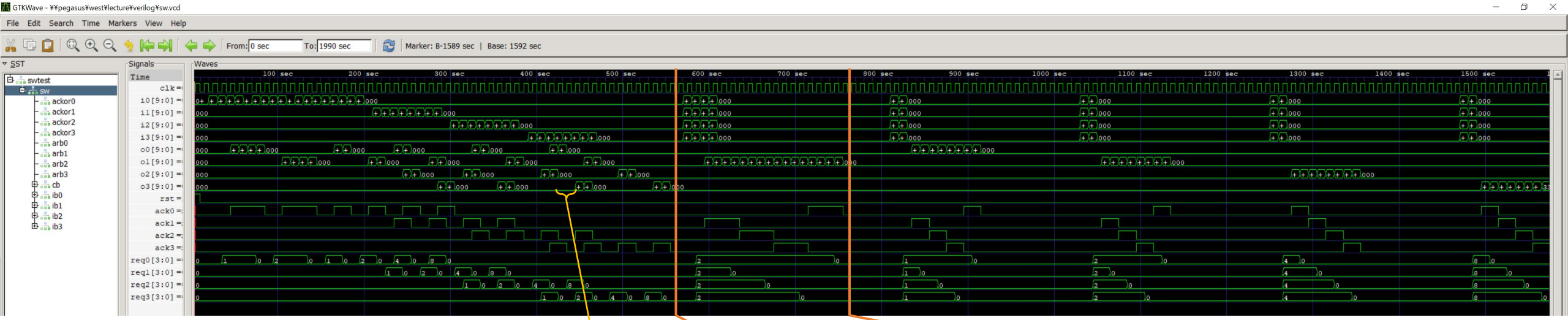
<<< sw.v >>>

```
`include "sw.vh"

module sw(input [`PKTW:0] i0, i1, i2, i3, output [`PKTW:0] o0, o1, o2, o3, input clk,
rst);
        logic [`PKTW:0] co0, co1, co2, co3;
        logic [`PORT:0] req0, req1, req2, req3;
        ib ib0(i0, co0, req0, ack0, full0, clk, rst);
        ib ib1(i1, co1, req1, ack1, full1, clk, rst);
        ib ib2(i2, co2, req2, ack2, full2, clk, rst);
        ib ib3(i3, co3, req3, ack3, full3, clk, rst);
        ackor ackor0(ack00, ack10, ack20, ack30, ack0);
        ackor ackor1(ack01, ack11, ack21, ack31, ack1);
        ackor ackor2(ack02, ack12, ack22, ack32, ack2);
        ackor ackor3(ack03, ack13, ack23, ack33, ack3);
        arb arb0(req0[0], req1[0], req2[0], req3[0], ack00, ack01, ack02, ack03, clk, rst);
        arb arb1(req0[1], req1[1], req2[1], req3[1], ack10, ack11, ack12, ack13, clk, rst);
        arb arb2(req0[2], req1[2], req2[2], req3[2], ack20, ack21, ack22, ack23, clk, rst);
        arb arb3(req0[3], req1[3], req2[3], req3[3], ack30, ack31, ack32, ack33, clk, rst);
        cb cb(co0, co1, co2, co3, o0, o1, o2, o3,
                {ack03, ack02, ack01, ack00}, {ack13, ack12, ack11, ack10},
                {ack23, ack22, ack21, ack20}, {ack33, ack32, ack31, ack30});
endmodule
```

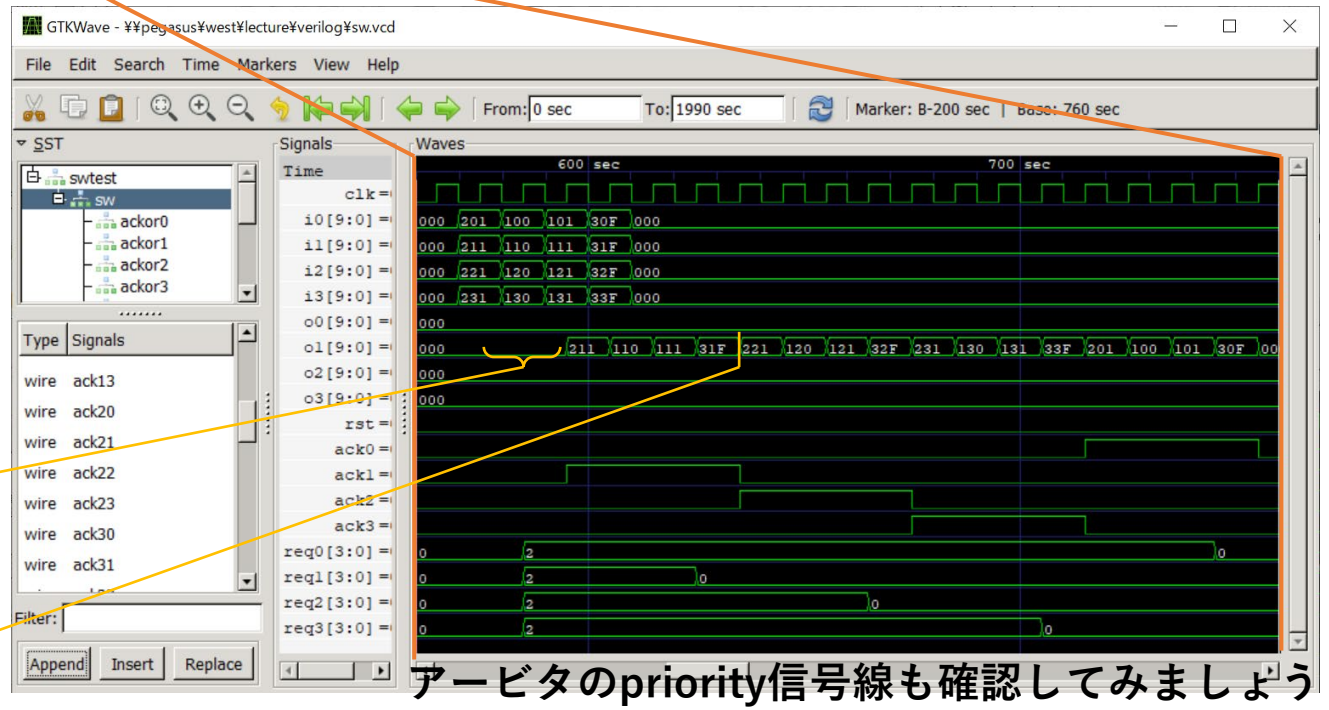
<<< sw.vh >>>

```
`define HEAD 2'b10 // Flow bit type
`define BODY 2'b01 // Flow bit type
`define TAIL 2'b11 // Flow bit type
`define EMPT 2'b00 // Flow bit type

`define ASSERT 1'b1
`define NEGATE 1'b0

`define PKTW 9
`define FIFOL 15 // 16:0 FIFO Length
`define FIFOLB 3 // 3:0  FIFO Length Bit

`define FLOWBH 9 // Flow Bit High
`define FLOWBL 8 // Flow Bit Low

`define PORT 3 // 3:0 // Number of Ports
```

全体の動作と出線競合時の動作のイメージ



- このテストベクトルは公開します
  - 共通ベンチマークとしてください
- 追加機能があれば、それに対応するテストベクトルを作成してチェックしてください
- この設計例では、宛先切り替え時パケットとパケットの間が常に2サイクル空きます
  - 最速では0にできます。先読みアービトレーションというテクニックを使います
- この設計例では、常に2サイクルの遅延でパケットが出てきます
  - 最速では1サイクルにできます。ただし、厄介なのとクリティカルパスが長く非実用的です。通常はテーブルを引いたり色々するのでもっと長くなります
- この設計例では、出線競合時のパケット間ギャップは常に0です



アービタのpriority信号線も確認してみましょう

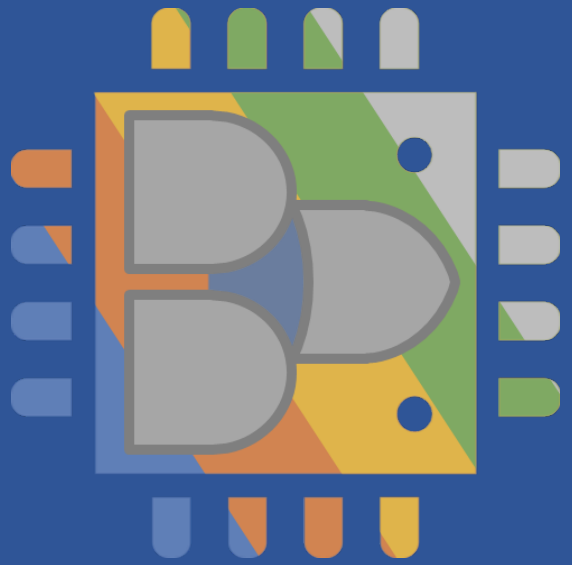
yosysによるswの合成結果の例

モジュール毎の回路規模

- ackor
  - 72.000000
- arb
  - 2356.000000
- cb
  - Unknown (0)
- cbsel
  - 1424.000000
- fifo
  - 35751.000000
- ib
  - Unknown (0)
- isbm
  - 822.000000
- mkreq
  - 1124.000000
- mkwe
  - 32.000000
- sw
  - Unknown (0)

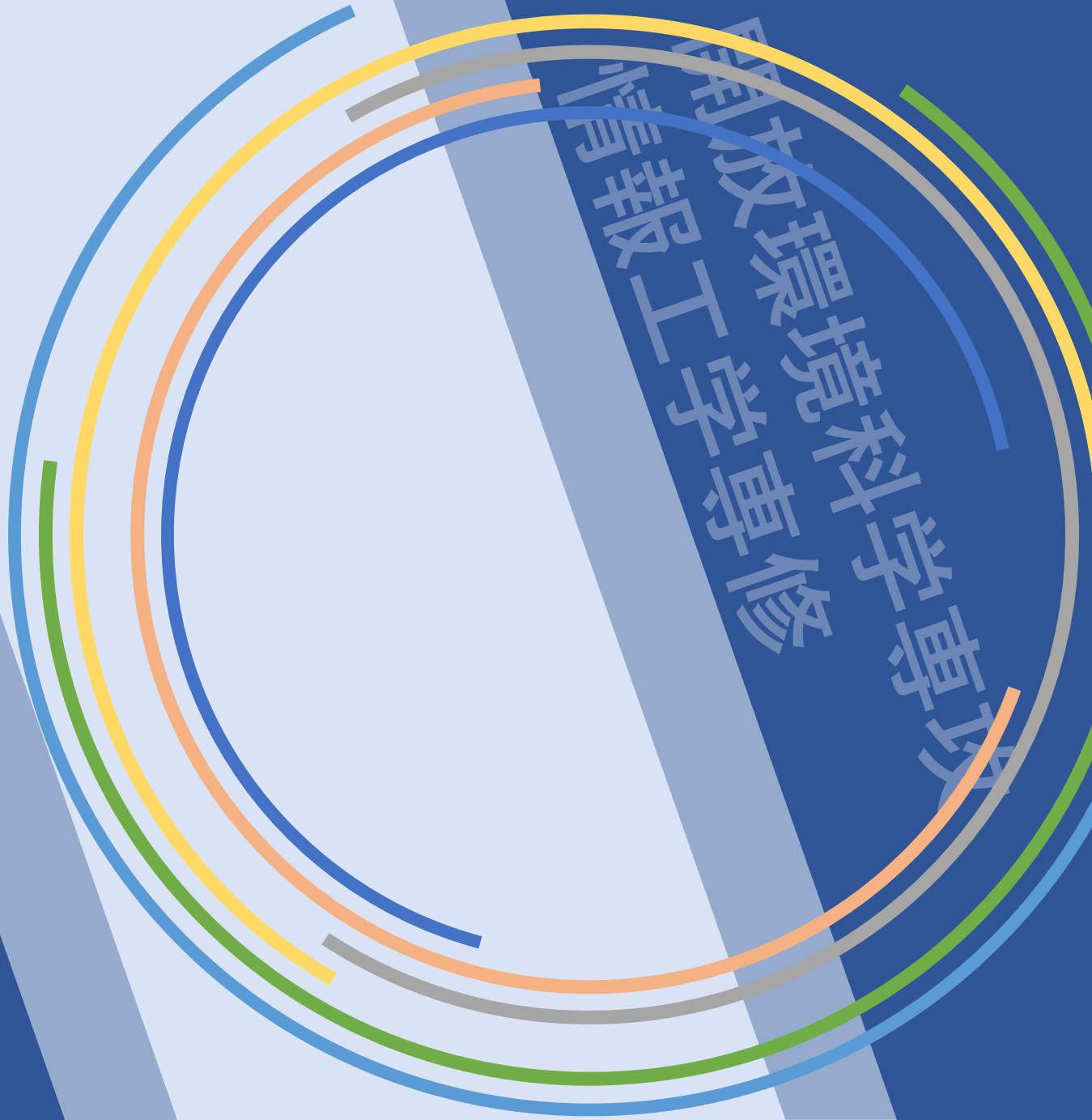
```
=== design hierarchy ===
   sw                              1
     ackor                         4
     arb                           4
     cb                            1
       cbsel                       4
     ib                            4
       fifo                        1
       isbm                        1
       mkreq                       1
       mkwe                        1
Number of wires:                 6546
Number of cells:                 3584
```

```
AND2X1                          76
    AOI21X1                     88
    AOI22X1                     80
    DFFPOSX1                   688
    DFFSR                       48
    INVX1                      816
    MUX2X1                     992
    NAND2X1                    160
    NAND3X1                     44
    NOR2X1                     212
    NOR3X1                      16
    OAI21X1                    220
    OAI22X1                     92
    OR2X1                       20
    XNOR2X1                     28
    XOR2X1                       4

Chip area for top module '¥sw': 166324.000000
```



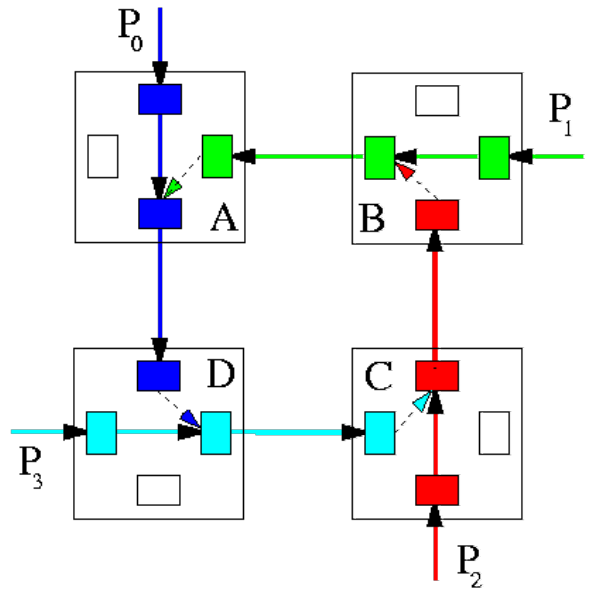
計算機システム設計論(11)
通信システム設計
最終課題

担当：　西　宏章



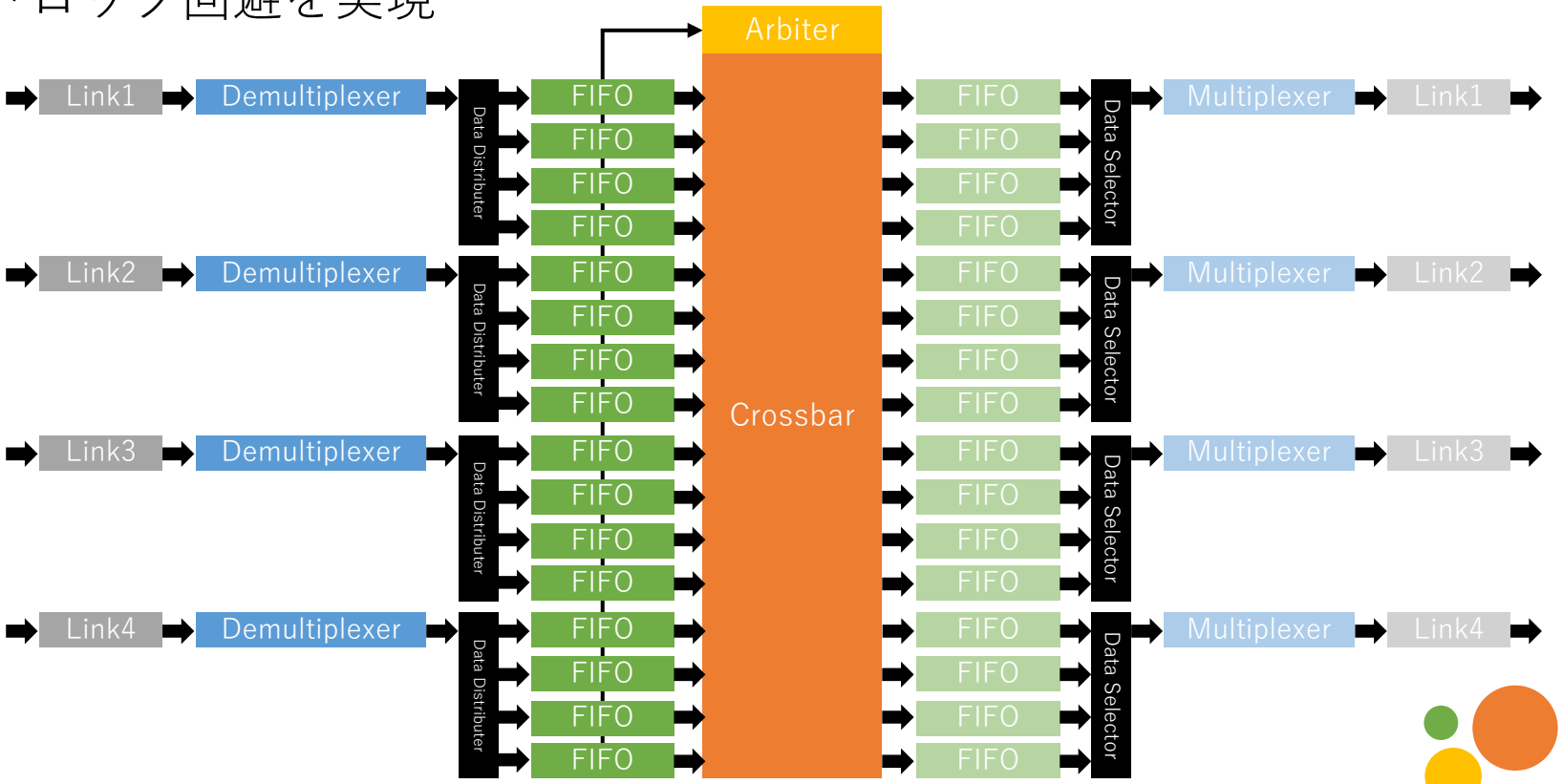
テーブル・仮想チャネル・デッドロック

- テーブルルーティング Destination Routing
  - 宛先アドレスからどのリンクへ出力するかを決めるルーティングテーブル
  - 一般的には宛先から一意に出力を決定するがマルチキャストも可能
    - デッドロック回避のため宛先と送り主の両方から決定する例もある
- 首切りルーティング Source Routing
  - 頭に「どのリンクから出るか」がホップ数分フリットとして加えられる
  - 各ルータ(スイッチ)で利用する度に捨てていく
  - ビットマップでマルチキャストも可能
- デッドロック
  - 右のように、それぞれ行きたい先が既に占有済みで、その依存関係が巡回している状態Cyclic Dependency
- デッドロック回避と仮想チャネル
  - 回避方法は様々あるが、優先順位によるパケット追い越しも含めて仮想チャネルが利用される



仮想チャネル

- チャネル
  - 一つのリンクに仮想的に複数のリンクがあるかのように扱える仕組み
  - 波長多重や時間多重など様々な方法で実現
  - 仮想チャネル
    - チャネルを仮想的にバッファ(FIFO)のみで実現
    - パケット順序の交換やデッドロック回避を実現

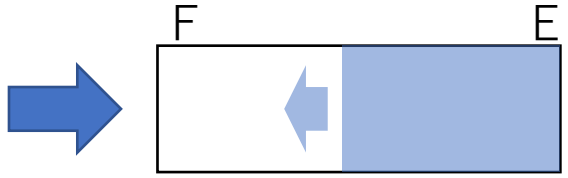


フロー制御

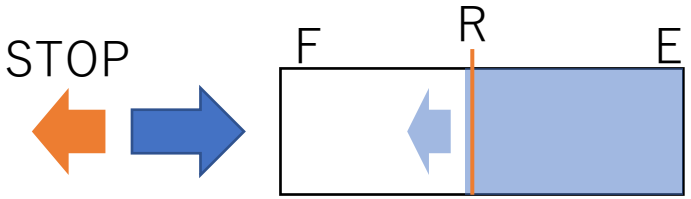
- スイッチにはバッファがあり、アービタによる調停結果により転送許可が得られなければずっとパケットをため込み続ける
  - そのうち、パケットバッファが溢れてしまい、データが壊れる
- インターネットはベストエフォートで、この点はある意味無視している
  - 一応、ICMPにはBackpressureがあり前のスイッチなどに転送停止要求を出せる
  - 一方で再送させて壊れてもよいように作られている
  - デッドロックしない（パケットを落としてしまうので）
- 一般にハイパフォーマンスコンピューティング向けネットワークでは、パケットがなくならないようにフロー制御を行う
  - 再送のコストが大きいため、ネットワークがパケットの完全性とFIFO性を保証
- フロー制御とは
  - パケットバッファが溢れないように、スイッチ間でデータの流れを制御すること
  - XON/XOFFなどシリアル通信でも存在する

GO/STOP制御

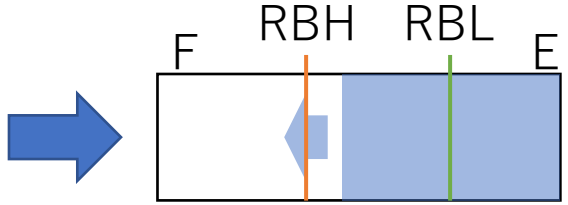
- パケットが入ってくるとバッファを埋めてゆく
  - このまま放置すると溢れてしまう



- そこで、あるレベルRを超えれば転送を停止するSTOPリクエストを出す
  - また、このレベルを下回れば転送を開始するGOリクエストを出す



- しかしながら次の問題がある
  - このままではRの境界を越えたか超えないかでGO/STOPのリクエストが出されるため、この境界付近でリクエストが頻発し、制御で通信帯域をとってしまい不安定になりやすい
  - そこでヒステリシス特性を持たせる
    - RBHを超えたらSTOPリクエスト
    - RBLを下回ったらGOリクエスト

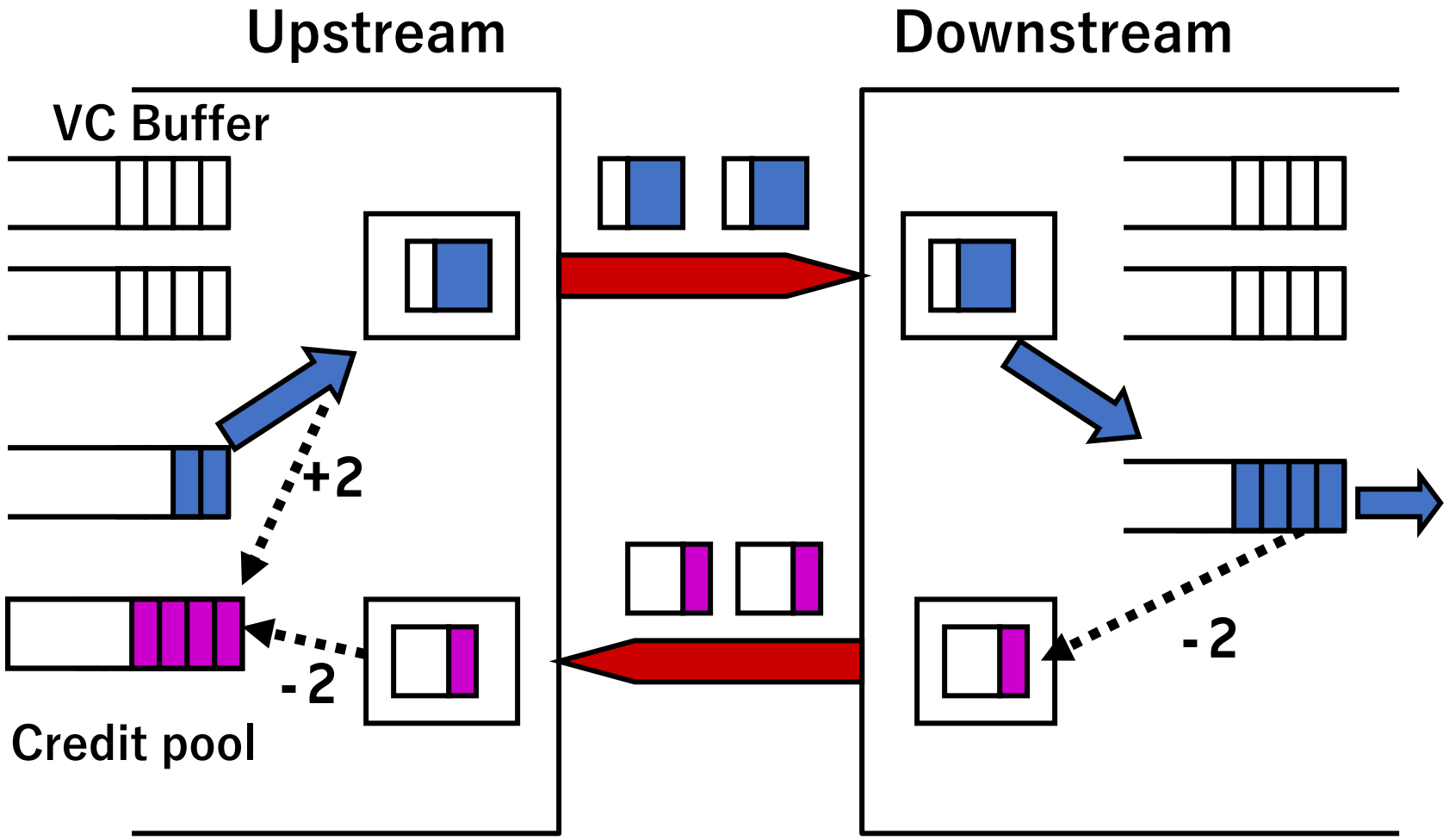


  - これらのスレッショルドの調整が必要でバッファ利用が非効率
    - 往復通信遅延や相手のGO/STOPリクエストを受け取ってから反応するまでの遅延を見込んで設定
- シリアルでの実装
  - 信号線を使うRTS(Request To Send)/CTS(Clear To Send)・Lazy Slack Buffer
  - フロー制御パケットを使うXON/XOFF

Credit制御

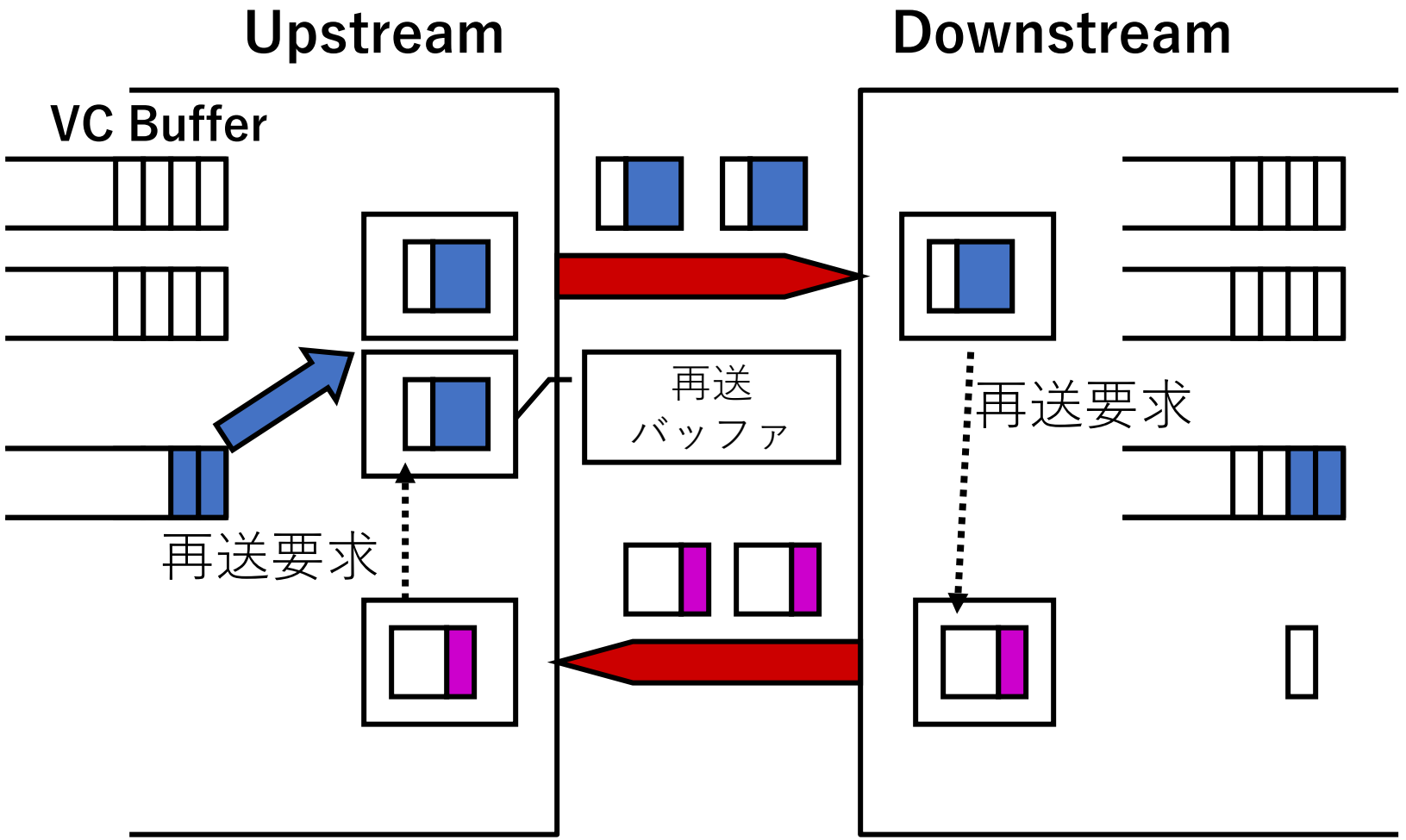
- GO/STOP制御の問題を解決
  - バッファは極力使いたい
  - 通信遅延や相手の処理遅延の見積もりを極力避けたい
- 基本的な考え方
  - GO/STOP制御は、データを受け取る方が管理する
  - Credit制御は、データを送る方が管理する
  - データ送信側はCredit(相手のバッファサイズに等しい数のコイン)を持っている
  - データを送るたびにコインを消費する　→　相手を溢れさせることはない
  - データ受信側は、パケットバッファからパケットを抜き出す（クロスバーを通る）度に、データ送信側コインを返す
  - これらの動作によりフロー制御を行う
- 欠点
  - フロー制御の通信コストが大きい
    - Piggybackによる効率化や、creditをまとめて返す仕組みによる効率化がある

Credit ベースフロー制御の例



再送機構

- 実際には、再送機構とエラー訂正の天秤、もしくは両方の実装を考えることになる



最終演習問題

- 4x4のスイッチを構成し、実際にパケットを流して動作を確認
  - 必ず複数の異なるパケット長の動作を確認すること
    - また、バイト文字を送って、正しくバイト文字が受け取れることを確認すること
  - 必ずリンク数は4以上にすること
  - 必ず出線競合状態を確認し、アービタで解決できていることを確認
  - フレームのフォーマットや宛先の情報などは自由に設計してよい
  - 必要最低限の実装でよく、アービタも自由な設計でよい
  - yosysの合成結果や、合成後のシミュレーション結果も示しなさい
  - ハンドシェイクを実装（必須ではない）
    - スイッチを2つ連結して、ハンドシェイク動作を確認すること
    - 下流側（出力側）のハンドシェイク入力や混雑によりバッファが溢れそうになったら、上流側に余裕をもって伝えること
    - 余裕の程度は各自で設定してよい

最終演習問題

- 注意事項
  - レポートをwordファイルとコードのzipファイルそれぞれを作成、LMSへ提出すること
    - 説明文をMicrosoft Wordファイルで、A4 で作成する。枚数は規定しないがシンプルに。
    - ソースプログラムとMakefileをzip圧縮して、別途LMSで提出
      - 当方がmakeでコンパイルし、実行、合成できるように、Makefileを入れておくこと
    - wordファイルとzipファイルは提出場所が違いますのでご注意ください
    - Zipには、課題の確認項目を動作するテストモジュールおよびテストベクトルを添付
    - 適切にvcdファイルが生成できるようにすること
    - 動作の証拠としてgtkwaveの画面をキャプチャしWordに張り付けること
  - 標準問題への回答と動作を基準に採点
    - 挑戦問題にトライした場合は、何にトライしたか、どのように設計したかを明記すること
  - wordファイルの最初にタイトルを「最終演習問題」とし、名前と学籍番号も忘れずに記載すること
  - ソースコードの先頭にはコメントで学籍番号、ローマ字氏名を入れておくこと
  - 提出締め切りはLMSを確認すること

最終課題提出について

- Wordファイルとverilogソースのzipファイルは別提出です
- verilogソースには全て、コメントで先頭に学籍番号・ローマ字氏名を記入してください
- verilogソースのzipファイルにはMakefileを入れてください！
  - MacOSはXcodeで普通に入っているはずです
  - Windowsの人はhttp://gnuwin32.sourceforge.net/packages/make.htm
    こちらからインストールしてください
    「Windows コマンド make」でググればいろいろ情報がでてきます
- Makefileはいろいろ機能がありますが、シンプルに次でOKです

  コマンド:

  　　やること1

  　　やること2

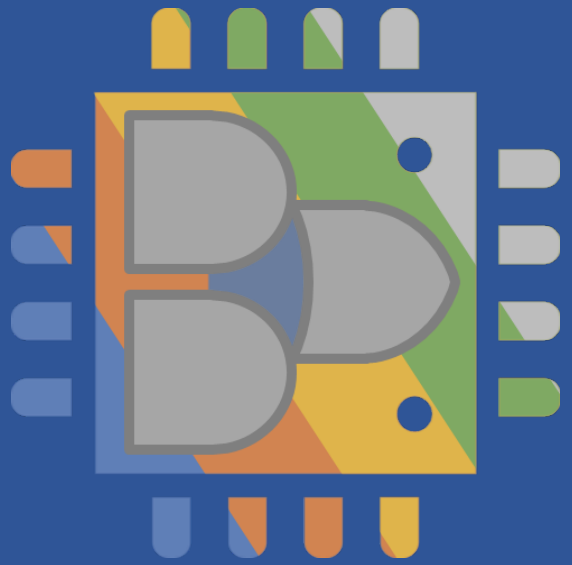
  先頭はタブです。以上です。次に例を示します

最終課題Makefileの作り方

次のファイルで、
make all、もしくは単にmakeとすると、シミュレーションができます
make showとすると、波形がみれます
make synとすると、合成します。当然ですが、sw.ysの作成と添付が必要です

Makefileの例

```
all:
	iverilog -g2012 sw.v ib.v ibsm.v fifo.v mkreq.v mkwe.v arb.v cb.v cbsel.v ackor.v swtest.v
	vvp a.out
show:
	gtkwave sw.vcd
syn:
	yosys sw.ys
synsim:
	iverilog -gspecify –Ttyp synth.v swtest.v ../osu018_stdcells.v
	vvp a.out
```



計算機システム設計論(12)
AD/DAコンバータ

担当：　西　宏章

電気電子情報工学科
西　宏章

最終課題提出について

- LMSをご覧ください
  - Wordファイルとverilogソースのzipファイルは別提出です
  - verilogソースには全て、コメントで先頭に学籍番号・ローマ字氏名を記入してください
- verilogソースのzipファイルにはMakefileを入れてください！
  - MacOSはXcodeで普通に入っているはずです
  - Windowsの人はhttp://gnuwin32.sourceforge.net/packages/make.htm
    こちらからインストールしてください
    「Windows コマンド make」でググればいろいろ情報がでてきます
- Makefileはいろいろ機能がありますが、シンプルに次でOKです

  コマンド:

  　　やること1

  　　やること2

  先頭はタブです。以上です。次に例を示します

最終課題Makefileの作り方

次のファイルで、
make all、もしくは単にmakeとすると、シミュレーションができます
make showとすると、波形がみれます
make synとすると、合成します。当然ですが、sw.ysの作成と添付が必要です

Makefileの例

```
all:
	iverilog -g2012 sw.v ib.v ibsm.v fifo.v mkreq.v mkwe.v arb.v cb.v cbsel.v ackor.v swtest.v
	vvp a.out
show:
	gtkwave sw.vcd
syn:
	yosys sw.ys
synsim:
	iverilog -gspecify –Ttyp synth.v swtest.v ../osu018_stdcells.v
	vvp a.out
```

レポートの採点について

- verilogソースコードとwordファイル別々に登録していただいています
  - verilogソースコードは、Makefileを探し出し自動実行して、その結果を目視で評価します。
  - 全員のverilogソースコードをコピペルナーで剽窃チェックします
    - verilogコードをityle-Verilog-formatterで成形し、
    - VerilatorでLintした結果
    - PyverilogによるLexer結果からReg, Wire, Ioportにある変数名を消去した結果

    以上のファイルを纏めてコピペルナーにかけます
  - Wordファイルはそのままコピペルナーにかけます
- オリジナリティを持ってください
  - 不幸にも、たまたま一致してしまったレポートは、「オリジナリティが欠如した」と判断され、そのまま採点されます。
  - ツールは剽窃チェッカーですが、やっていることはオリジナリティの定量的チェックと考えてください

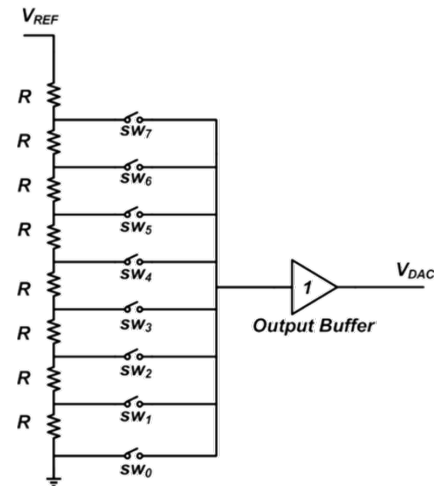
最終演習問題（参考）

- 注意事項
  - レポートをwordファイルとコードのzipファイルで作成、LMSへ提出すること
    - 説明文をMicrosoft Wordファイルで、A4 2枚程度で作成する
    - ソースプログラムとMakefile(次に説明、動画には含まれていません。動画では最終回で説明しています）をzip圧縮して、別途LMSで提出
    - wordファイルとzipファイルは提出場所が違いますのでご注意ください
    - Zipには、課題の確認項目を動作するテストモジュールおよびテストベクトルを添付
    - 当方がmakeでコンパイルし、実行、合成できるように、Makefileを入れておくこと
    - 適切にvcdファイルが生成できるようにすること
    - シミュレーションを行い、動作の証拠としてgtkwaveの画面をキャプチャしWordに張り付ける
  - 標準問題への回答と動作を基準に採点する
    - 挑戦問題にトライした場合は、何にトライしたか、どのように設計したかを明記すること
  - wordファイルの最初にタイトルを「最終演習問題」とし、名前と学籍番号も忘れずに記載すること
  - ソースコードの先頭にはコメントで学籍番号、ローマ字氏名を入れておくこと
  - 提出締め切りはLMSを確認すること

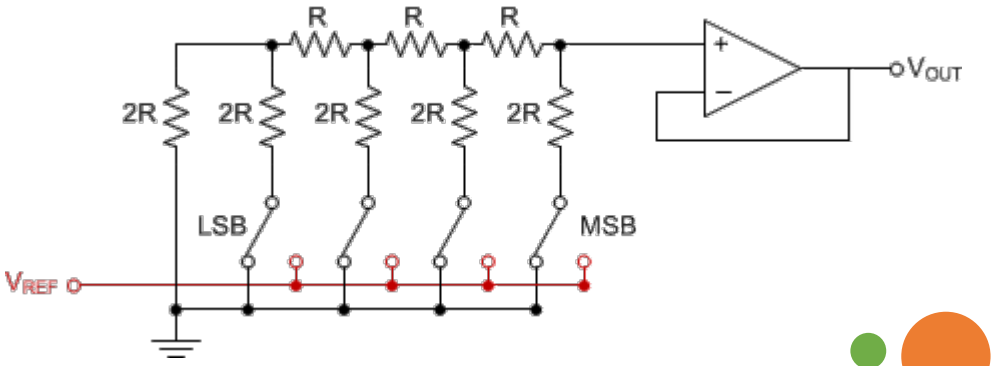
D/A Converter: DAC

- 前提としてDAC、ADCなどアナログは専用の変換ICを使うべき
- ここでは、簡易的に仕組みを説明
- D/Aコンバータ
  - デジタルをアナログに変換
  - デジタル値の0, 1, 2, 3…に対して、アナログ値0.1V, 0.2V, 0.3V…へ変換
- A/Dコンバータ
  - アナログをデジタルに変換
- DAC, ADC共に、一般的に10bit程度以上が利用されている
    - つまり、解像度は1024諧調以上
- サンプリングレートは様々

String DAC / Ladder DAC

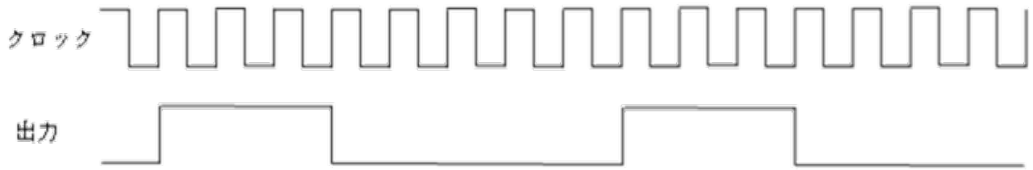


- ストリング DAC
  - スイッチの場所で抵抗値が変わる最も基本的発想
  - 抵抗器の数は解像度の増加に伴って指数関数的に増加
    - かつRを全て同じ抵抗値に揃えるのが困難で非現実的
- Ladder DAC
  - R-2R構造と呼ばれるバイナリ加重抵抗ラダー構造を利用して対応
    - 例えばLSBからみて3番目は、左は全てGNDから2Rと2Rの合成抵抗Rに対して直列Rで2R、さらにこれが2Rと並列になる、と見えるため、3番目に限らず、どこを見ても2RでGNDに繋がる
    - 右も同様に2RでGNDになる
    - すると、3番目は、左右に2RでGND、下は2RでV_{REF}に接続するため、$\frac{V_{REF}}{3R}$の電流が流れる
    - 左右等しいためそれぞれ、$\frac{V_{REF}}{6R}$流れる
    - 従って、n個あるk番目がONの時にオペアンプへ流れる電流は、$\frac{V_{REF}}{6 \cdot 2^n R} 2^k$となる
    - 重ね合わせにより、任意のスイッチS_kのON/OFFベクトルに対し、で$V_o = -\frac{V_s}{3 \cdot 2^n} \sum_{k=1}^{n} 2^k S_k$

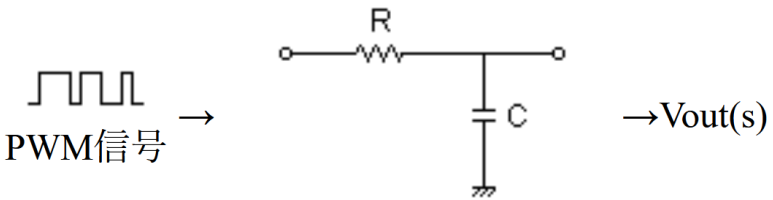


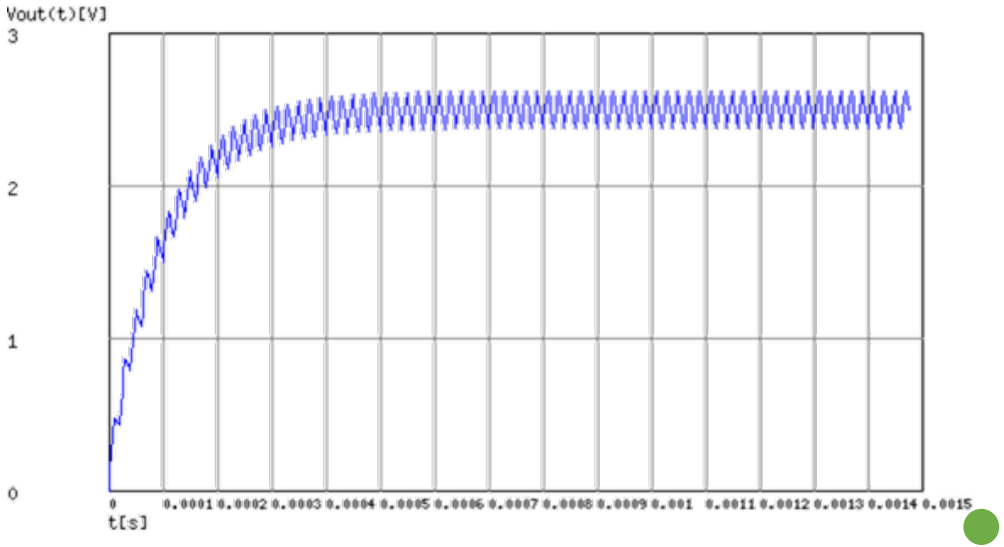
PWM DAC

- 完全デジタルかつシンプルにDACを構成するにはPWMとLPFを応用
  - FPGAやASICなどデジタルデバイスのI/Oは高速動作を目的としている
  - ドライブ能力(電圧や電流の許容量)が非常に小さく、浮遊容量も数十pF未満
    - さらにアンプを利用



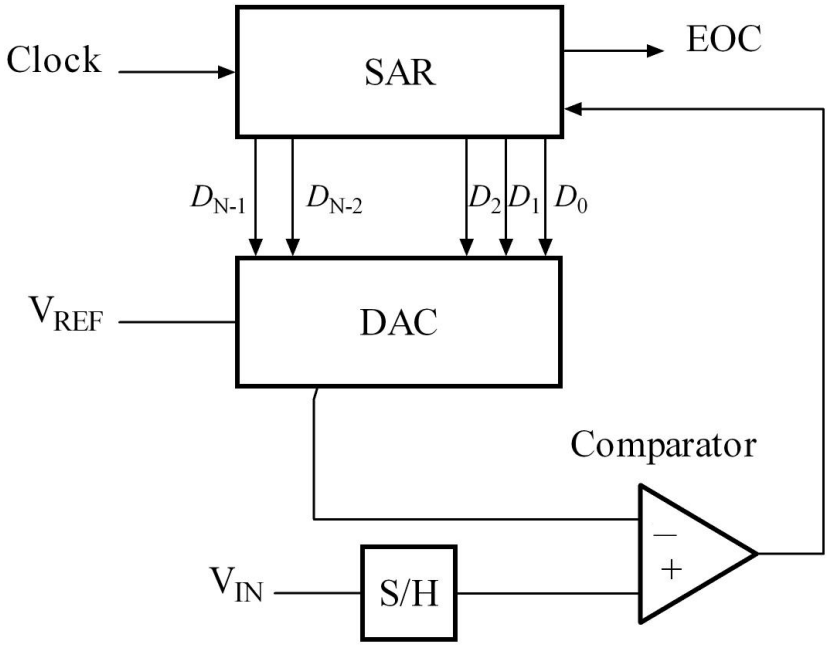
- PWM制御
  - デジタルデバイスは例えば0Vか5Vかしか出力できないが、その間の3Vを出したい
  - 3/5の時間、つまり3クロックの間5Vを出力し、2クロックの間0Vを出力する
  - これをCRローパスフィルタに入れる
    - 1KHz、50%Duty、5V信号、10KΩ、10nF をシミュレートすると、2.5Vへおよそ0.35ms で到達





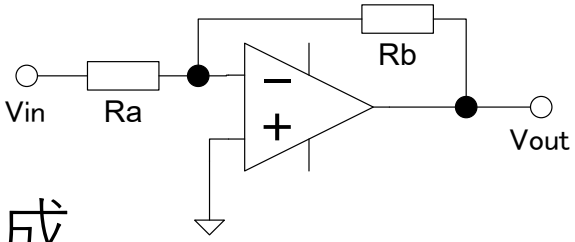
ADC 逐次比較型・フラッシュ型

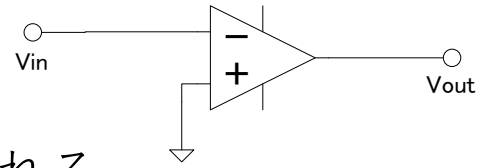
- 逐次比較型(successive approximation)
  - 標本回路(sample and hold circuit: S/H) により入力電圧を固定し、DA変換で既知電圧を作り出して逐次比較することで変換
  - もちろんであるが、2分岐法で比較することで、より高速に変換できる
    - それでも10bit ADCでは、10回比較が必要
- フラッシュ型
  - 比較器を複数用いてパラレルに行う方法
  - 256個ならべて8bitとなり規模が大きいが非常に高速に動作(10G超えも余裕)
    - 測定器向け

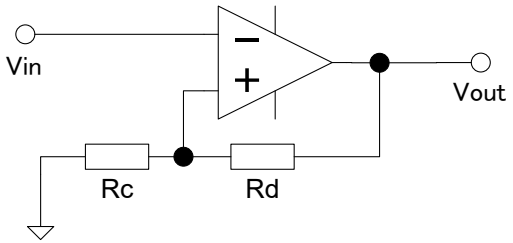


ADCにおける比較

- 一般的にはLM393などを用いる
- OPアンプを利用して反転増幅アンプを構成
  - 基準電圧Vrefを定めると、$V_{out} = (V_{ref} - V_{in})\frac{R_b}{R_a} + V_{ref}$
  - この回路の入出力特性は図1、R_b無限大では図2
  - つまり、ハイインピーダンスとして
    これで十分
    - V_{out}はV_{in}がV_{ref}を横切ると＋から－へ振り切れる
    - このままではノイズに弱いためヒステリシス特性を入れる
    - $V_{ref}LH = V_{ref} + \left((V_{out_{MAX}} - V_{ref}) \cdot \frac{R_c}{R_c+R_d}\right)$
    - $V_{ref}HL = V_{ref} + \left((V_{out_{MIN}} - V_{ref}) \cdot \frac{R_c}{R_c+R_d}\right)$
    - $V_{ref}LH - V_{ref}HL = (V_{out_{MAX}} - V_{out_{MIN}}) \cdot \frac{R_c}{R_c+R_d}$
  - ヒステリシスの係り具合を調整でき1bit ADCとして機能する







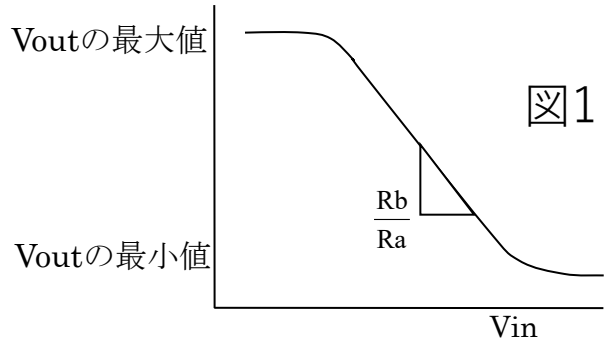


図1

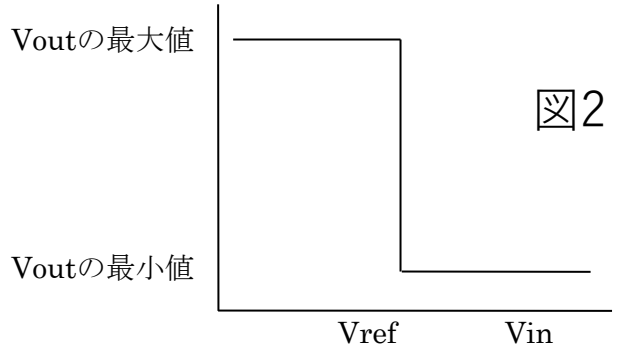
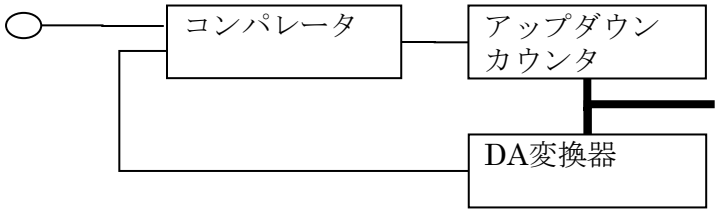
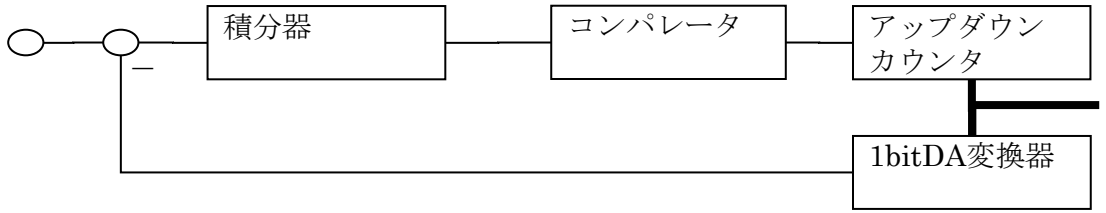


図2

ADC 積分型・シグマδ

- シグマデルタADC
  - 日本でデルタシグマ(ΔΣ)型と呼ばれ、日本人提案
  - 一言で言えば微分の積分によるADC
  - 十分に速い理想的な速度でサンプリングが行われたとする
    - サンプリングの結果一つ前のサンプリング値と今を比較
    - 値が１つ増えるか、値が１つ減るか、変わらないかのどれか
    - この3値について、値が変化しないということは、値の増減を繰り返すと表現すれば2値でよい
    - 値が増えたら１減ったら０として表現、ただし常に値が変化しないといけない
    - そこで積分器を入れて誤差を積み重ねる





最後に

- 以上で計算機システム設計論のすべての講義内容は終了となります

- ハードウェア設計について初めての方は一端がつかめたでしょうか？
- 既に学習済みの方は理解が深まったでしょうか？
  - 少なくとも、論理記述と回路の関係は深まったのではないかと期待しています

- PUやスイッチ設計を通して、計算機システム、とくにシステムという観点での理解は進んだでしょうか？

- 最終レポートへの取り組みをよろしくお願いします。
  - 質問などはいつでもSlackへお寄せください。