８ビットＣＰＵ (KUE-CHIP2) の設計

澤田　宏

NTT コミュニケーション科学研究所

はじめに

演習の題材

パルテノン研究会では，現在，第 4 回 ASIC デザインコンテストが行われています．その規定課題の一つに，例年と同様に，８ビットマイクロプロセッサ KUE-CHIP2 を実現するというものがあります． KUE-CHIP2 は，教育用に開発されたもので，単純な命令セット・アーキテクチャを持っています．また，その動作が理解しやすいように，内部レジスタの値を観測・制御できたり，1 命令毎・1 フェーズ毎に動作を止めることができるようになっています．ただし，この課題では，これらの観測機能等は実現しなくてよいことになっています．

今回のパルテノン講習会では，例年と同様に，この KUE-CHIP2 を題材として，８ビット CPU の設計演習を行います．本演習では，わかりやすさ，簡潔さを第一に考えた SFL 記述を完成させます．したがって，ごく平凡な KUE-CHIP2 となりますが，この演習での内容を基にして，それぞれの素晴らしい KUE-CHIP2 を作って頂きたいと思います．また今年から，実際に設計した KUE-CHIP2 を FPGA を用いて動作させることも行います．これには ASTEM が開発した KUE-CHIP2 教育用ボードと KUE-CHIP2 daughter ボードを用います．

本文の最後に， ASIC デザインコンテストの規定課題の資料（以下，規定課題の資料と呼ぶ）を載せます．ここには KUE-CHIP2 の仕様が詳しく書かれています．本演習の参考資料として用いてください．

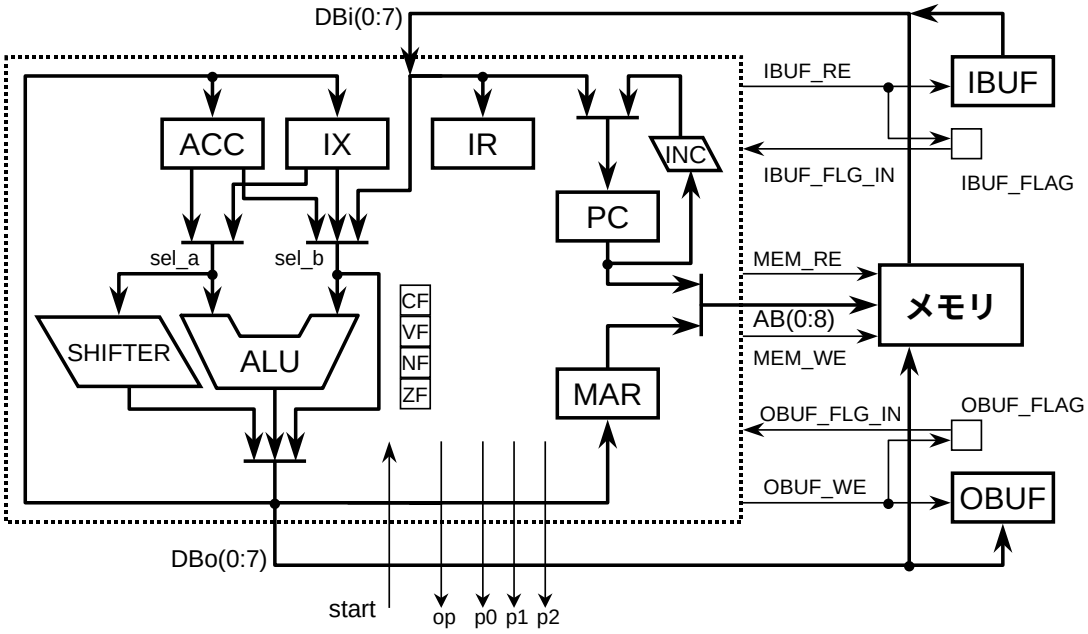
なお，本演習で使うパルテノンは， SunOS 用のバージョン 2.3.0.6 を想定しています．バージョン 2.3 のマニュアルとしては， PARTHENON/CQ 版にある「はじめての PARTHENON」およびこれを HTML 化したもの
(http://www.kecl.ntt.co.jp/car/parthe/html/manual.htm)
があります．

演習の進め方

本演習は以下のように進めていきたいと思います．

1. KUE-CHIP2 の構造と動作
   今回設計する KUE-CHIP2 のアーキテクチャの詳細を決定します．
2. モジュール kueshift の設計
   シフト命令を処理する部分を SFL のモジュールとして設計します．
3. モジュール kuealu の設計
   算術論理演算命令を処理する部分を SFL のモジュールとして設計します．
4. KUE-CHIP2 の SFL 記述
   2, 3 で設計したモジュールをサブモジュールとして用い， KUE-CHIP2 全体の SFL 記述を作成します．
5. KUE-CHIP2 の動作確認
   シミュレータ SECONDS を用いて， KUE-CHIP2 の命令セットからなるプログラムを動かしてみます．
6. FPGA を用いた実現
   ALTERA 社の FPGA をターゲットとして論理合成を行い， KUE-CHIP2 daughter ボードに搭載した FPGA に回路データをダウンロードします．これを KUE-CHIP2 教育用ボードと接続して，設計した FPGA 版 KUE-CHIP2 を動作させてみます．
7. ASIC をターゲットとした論理合成
   ASIC (DEMO 社 demo ライブラリ) をターゲットとして論理合成を行い，設計した KUE-CHIP2 の面積や動作速度を見てみます．

図 1: ブロック図



1 KUE-CHIP2 の構造と動作

今回の ASIC デザインコンテストでは， KUE-CHIP2 の命令セットのみが規定されていて，外部端子・内部の構造・各命令の動作のタイミングなどは，設計者が自由に決めることができます．そこで本実習で SFL 記述を始める前に，これらの詳細をここで決定します．

構成要素

図 1に今回設計する KUE-CHIP2 のブロック図を示します．この図には， KUE-CHIP2 がどのようなハードウェア資源から構成されているかということが書いてあります．

KUE-CHIP2 は ACC (アキュームレータ) と IX (インデックスレジスタ) という 2 つの 8 ビットレジスタを持ちます．これらのレジスタに対しては，加減算や論理演算を施したり，その値をシフトをすることもできます．そのために，加減算や論理演算を行う ALU と，シフトを行う SHIFTER を用意します．また，演算結果を示すフラグとしては， CF (桁上げフラグ)， VF (桁あふれフラグ)， NF (ネガティブフラグ)， ZF (零フラグ) が必要です．

KUE-CHIP2 のプログラムはメモリの 0 番地から 255 番地に置くことができるので，プログラムの実行位置を示す PC (プログラムカウンタ) は 8 ビット幅のものとなります．また，プログラムの逐次実行と共に PC の値を 1 ずつ増やしていくために， INC という 8 ビットのインクリメンタを用意します．

KUE-CHIP2 の命令には， 1 バイト (= 8 ビット) のものと 2 バイトのものがあります．いずれも， 1 バイト目は命令の種類を示し， 2 バイト目にはアドレス情報が格納されています．ここでは，メモリから読み出した命令の 1 語目を格納するために， 8 ビットの IR (インストラクションレジスタ) を用意します．また，命令の 2 語目から生成される実効アドレスを格納するために， 8 ビットの MAR (メモリアドレスレジスタ) を用意します．

外部端子

次に，外部端子について述べます．これらも図 1に示しています．

start は， KUE-CHIP2 の動作を開始させます． op は KUE-CHIP2 が動作中に 1 を出力します．また， p0, p1, p2 はそれぞれ，表 1 に示すフェーズ ph0, ph1, ph2 を実行中に 1 を出力します．

KUE-CHIP2 のプログラムは 0 番地から 255 番地までの間（プログラム領域）にしか置けませんが，データならば， 256 番地から 511 番地の間（データ領域）に置いてもアクセスすることができます．そのため，アドレスバス AB の幅は 8 ビットではなく 9 ビットになっています． AB に値を供給する PC や MAR は 8 ビットですが， AB の 9 ビット目（プログラム領域かデータ領域かを指定する）は，制御系から状況に応じて 0 か 1 が指定されます．

DBi はメモリや入力バッファ IBUF の内容を読み込むための入力データバスです．また， DBo はメモリや出力バッファ OBUF にデータを書き込むための出力データバスです．これらは共に 8 ビットの幅を持ちます．

メモリを制御する外部端子としては， MEM_RE と MEM_WE があります． MEM_RE により，メモリから DBi に値を転送します．また， MEM_WE により， DBo の値をメモリに書き込みます．

表 1: フェーズ表

略記号	命令コード	ph0	ph1	ph2
NOP	00000---	IR := MEM(0,PC) PC := INC(PC)	**何もしない**	
HLT	00001---		**停止する**	
OUT	00010---		OBUF := ACC, OBUF_FLAG := 1	
IN	00011---		ACC := IBUF, IBUF_FLAG := 0	
SRCF	0010v---		CF := v	
BRANCH	0011cccc --------		PC := MEM(0,PC) {**条件成立**} PC := INC(PC) {**条件不成立**}	
SHIFT	0100asss		A := SHIFTER(A)	
LD_REG	0110a00b		A := B	
AL_REG	1mmma00b		A := ALU(A, B)	
LD_IMM	0110a01- --------		A := MEM(0,PC) PC := INC(PC)	
AL_IMM	1mmma01- --------		A := ALU(A, MEM(0,PC)) PC := INC(PC)	
LD_MA	0110a1xr --------			A := MEM(r,MAR)
ST_MA	0111a1xr --------		MAR := MEM(0,PC) {+IX} PC := INC(PC)	MEM(r,MAR) := A
AL_MA	1mmma1xr --------			A := ALU(A, MEM(r,MAR))

表 2: アドレスモード

略記号	命令コード		アドレスモード	実効アドレス
_REG		b: 0	ACC	
		b: 1	IX	
_IMM			**即値アドレス**	
_MA	x: 0	r: 0	**絶対アドレス（プログラム領域）**	0 || (MAR := **2語目**)
		r: 1	**絶対アドレス（データ領域）**	1 || (MAR := **2語目**)
	x: 1	r: 0	**インデックス修飾アドレス（プログラム領域）**	0 || (MAR := **2語目**+IX)
		r: 1	**インデックス修飾アドレス（データ領域）**	1 || (MAR := **2語目**+IX)

IBUF_FLAG や OBUF_FLAG は，入力バッファ IBUF や出力バッファ OBUF に値が書き込まれた後，その値が読み出されたかどうかを示すフラグです．IBUF_FLAG や OBUF_FLAG の値は，IBUF_FLG_IN や OBUF_FLG_IN を通じて KUE-CHIP2 に入力され，分岐命令の条件判断に利用されます．

IBUF や OBUF とそれらのフラグを制御する外部端子としては，IBUF_RE と OBUF_WE があります．IBUF_RE は，IBUF の値を DBi に転送すると同時に，IBUF_FLAG を 0 にします．IBUF_RE と MEM_RE は同時にアクティブになってはいけません．OBUF_WE は，DBo の値を OBUF に書き込むと同時に，OBUF_FLAG を 1 にします．

命令の分類

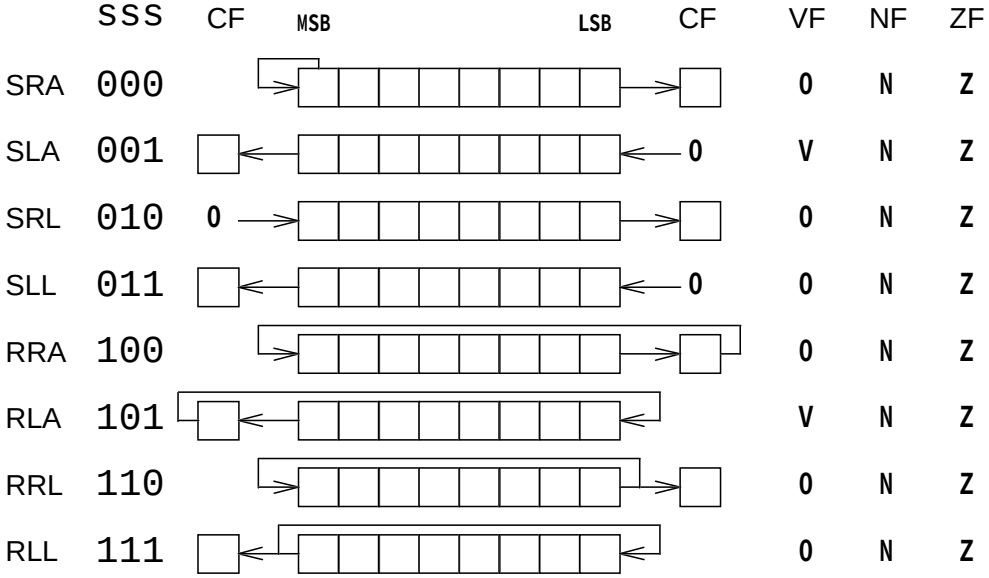
次に，図 1に示されたハードウェア資源（レジスタ，メモリ，モジュール，端子など）を，どのタイミングでどのように利用するかということ決めていきます．その前に，表 1に示すように，KUE-CHIP2 の命令 (詳細は規定課題の資料を参照してください) を分類します．

NOP (何もしない), HLT (停止), OUT (出力バッファに書き込む), IN (入力バッファを読み込む) 命令はそれぞれ独自のものとして分類します．RCF, SCF (桁上げフラグリセット・セット) 命令は 1 つにまとめて SRCF とします．Bcc (分岐) 命令は条件によって様々なものがありますが，これらをすべて 1 つにまとめて BRANCH とします．Ssm (シフト) 命令と Rsm (巡回シフト) 命令もすべてを 1 つにまとめて SHIFT とします．

LD (ロード), ST (ストア) 命令は，アドレスの指定方法により，LD_REG, LD_IMM, LD_MA, ST_MA と細分化して考えます．ここで，_REG (register) はレジスタ指定，_IMM (immediate) は即値指定，_MA (memory address) は絶対アドレスとインデックス修飾アドレス指定を表します．

SBC, ADC (桁上げつき減算・加算), SUB, ADD (桁上げなし減算・加算), EOR, OR, AND (論理演算), CMP (比較) 命令はすべて ALU を使用する命令なので 1 つにまとめて考えますが，LD, ST 命令と同様にアドレスの指定方法がいろいろあるので，AL_REG, AL_IMM, AL_MA と分類します．

図 2: シフト命令



アドレス・モード

先ほど，命令を分類した際に，_REG, _IMM, _MA と，アドレスの指定方法によっても分類しました．ここでは，それらのアドレスの指定方法について詳しく見ていきます．

KUE-CHIP2 は表 2に示すようなアドレスモードを持ちます．_REG では，命令コードの b によって指定されたレジスタの値が対象となります．_IMM では，命令コードの 2 語目の値そのものが対象となります．_MA では，実効アドレスが指すメモリの番地が対象となります．

_MA では，実効アドレスを生成して MAR に書き込みますが，その方法が命令コードの x や r によって異なります． x が 0 であれば，命令コードの 2 語目をそのまま MAR に書き込みますが， x が 1 であれば，命令コードの 2 語目に IX の値を加えた値を MAR に書き込みます．また，命令コードの r は，メモリのプログラム領域 (0) かデータ領域 (1) のどちらにアクセスするかということを示します．つまり， r がそのまま実効アドレスの 9 ビット目になります． KUE-CHIP2 のメモリ空間は 512 バイトで， 0 番地から 255 番地はプログラム領域， 256 番地から 511 番地はデータ領域と区別されています．

各命令の動作

以上で命令の分類が終わりましたので，各命令の動作を決めていきます．表 1に示すように，今回の設計では ph0, ph1, ph2 という 3 つのフェーズを用意します．各フェーズの動作はすべて 1 クロック内に終了します． ph0 では，メモリから読み出した命令コードを IR に格納します．この動作は命令の種類によらず同じです．なお， MEM(0,PC) は，アドレスの 9 ビット目が 0 （プログラム領域）で，残りのアドレス 8 ビットが PC の値であるようなメモリの番地を意味します． ph1 と ph2 では，命令によって動作が異なってきます． ph1 でその動作を終える命令もかなりあります．それでは，各命令の動作について見ていきましょう．

NOP では何もしません． HLT では停止します．

OUT では， OBUF に ACC の値を書き込み， OBUF_FLAG を 1 にします． IN では， IBUF の値を ACC に読み込み， IBUF_FLAG を 0 にします．

SRCF では， CF （桁上げフラグ）に命令コードの v の値を書き込みます．

BRANCH では，命令コードの cccc が示す条件を満たしていれば， 2 語目が示すアドレスに分岐します． cccc と分岐条件の対応については，規定課題の資料を見てください．

SHIFT では， SHIFTER を用いて，命令コードの sss が示すようにシフトを行います． sss が示すシフトの種類ついては，図 2を見てください．なお，命令コードの a または b は， ph1 や ph2 の A または B が ACC か IX のどちらであるかを示します． a が 0 ならば A は ACC， 1 ならば A は IX とします． b に関しても同様です．

LD_REG では，レジスタ B の値をレジスタ A に書き込みます．

AL_REG では，レジスタ A, B に対し算術論理演算を行い，結果をレジスタ A に書き込みます．

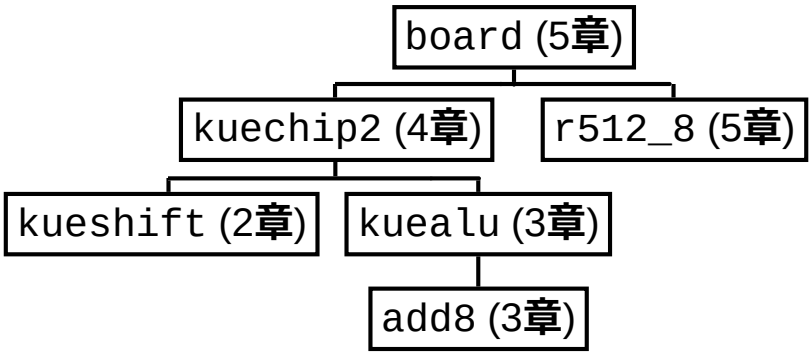
LD_IMM では， 2 語目そのものの値をレジスタ A に書き込みます．

図 3: ALU の動作

略記号	命令	mmm	命令の内容	CF	VF	NF	ZF
SBC	SuB with Carry	000	A := A + ^B + ^CF	^C	V	N	Z
ADC	ADd with Carry	001	A := A + B + CF	C	V	N	Z
SUB	SUBtraction	010	A := A + ^B + 0b1	-	V	N	Z
ADD	ADDition	011	A := A + B	-	V	N	Z
EOR	Exclusive OR	100	A := A @ B	-	0	N	Z
OR	OR	101	A := A | B	-	0	N	Z
AND	AND	110	A := A & B	-	0	N	Z
CMP	CoMPare	111	A + ^B + 0b1	-	V	N	Z

C: MSB(最上位ビット)からの桁上げが起こったときに1となり、その他の場合は0となる
V: MSBからの桁上げとMSBへの桁上げのどちらか1つのみ起こったときに1となり、その他の場合は0となる
N: 結果においてMSBが1であれば1となり、その他の場合は0となる
Z: 結果においてすべてのビットが0であれば1となり、その他の場合は0となる

図 4: モジュール構成図



AL_IMM では，レジスタ A と 2 語目そのものの値に対し算術論理演算を行い，結果をレジスタ A に書き込みます．

_MA では，まず ph1 で実効アドレスの 9 ビット目以外を MAR に書き込みます．そして ph2 においては， LD_MA では，実効アドレスが示すメモリの番地の値をレジスタ A に書き込みます． ST_MA では，実効アドレスが示すメモリの番地にレジスタ A の値を書き込みます． AL_MA では，レジスタ A の値と実効アドレスが示すメモリの番地の値に対し算術論理演算を行い，結果をレジスタ A に書き込みます．

AL では， ALU を用いて，命令コードの mmm が示す算術論理演算を行います． mmm と ALU の動作の対応は，図 3を見てください．なお，シフト命令や算術論理演算命令が行われた際には，その結果に従って CF, VF, NF, ZF の値を更新します．

以上が今回設計する KUE-CHIP2 のアーキテクチャの詳細です．次の章からは，これを SFL で表現していきます．まず 2 章では，シフト命令を処理する SHIFTER を， SFL のモジュール kueshift として設計します． 3 章では，算術論理演算命令を行う ALU を， SFL のモジュール kuealu として設計します． 4 章では，これらのモジュールをそれぞれサブモジュール shifter, alu として用いて， KUE-CHIP2 全体の SFL 記述を完成させます．図 4にこれから設計するモジュール間の関係を示します．

2　モジュール kueshift の設計

まずは，KUE-CHIP2 のシフト命令を処理する回路を SFL で記述します．SFL では，ある機能を持つひとかたまりの回路をモジュールと呼びます．ここでは，シフト命令を処理する回路をモジュール kueshift として設計します．あるモジュールは別のモジュールの中で，サブモジュールとして利用できます．すなわち，階層化設計を行うことができます．モジュール kueshift は，後で KUE-CHIP2 全体を設計するときに，サブモジュール shifter として利用されます．

SFL 記述

リスト 2.1 にモジュール kueshift の SFL 記述の一例を示します．図 2のシフト命令の説明と照らし合わせれば，SFL 記述の意味が良く分かると思います．

5 行目から 7 行目では，外部端子の定義を行っています．キーワード instrin，input，output は，それぞれ，制御入力端子，データ入力端子，データ出力端子を示します．制御入力端子は，外部からそのモジュールに仕事を依頼するためのものです． kueshift は， do という制御入力端子を外部から起動することで，シフト処理を行うものとします．データ入力端子としては，シフトの種類を示す mode<3> と，シフトされるデータ in<8> と，シフト前の CF の値 ci を持つものとします．ここで， mode は， 8 種類のシフト命令のうちの 1 つを指定するものですが，表 1や図 2の sss に対応させることにします．データ出力端子としては，シフト結果を表す out<8> と，シフト後の各フラグを表す co，vo，no，zo を用意します．

13 行目から 40 行目には，制御入力端子 do が起動されたときに行う動作が記述されています．ここで， == は一致判定を， in<7> や in<7:1> などはビット切り出しを， || はビットの連結を， /| は桁方向の or を表しています．KUE-CHIP2 のシフト命令のシフトの量はすべて 1 ビットです．シフトの際に入り込んでくる値は，一時的に edgebit というデータ内部端子に保持するものとします． 10 行目の sel はデータ内部端子を示します． 25 行目から 28 行目は right 方向のシフトの動作， 29 行目から 32 行目は left 方向のシフトの動作です． 34 行目から 37 行目は，桁あふれフラグ VF に関する記述です． SLA 命令と RLA 命令において，シフト前とシフト後で最上位ビットが異なるときに，桁あふれとみなされます．

リスト 2.1 の説明は以上です． kueshift の SFL 記述は， kueshift.sfl というファイルに格納することにします．

リスト 2.1: SFL 記述 (kueshift.sfl)

```
 1  /** kueshift: shifter for KUE-CHIP2 **/
 2
 3  module kueshift {
 4      /** external pins **/
 5      instrin     do;
 6      input       mode<3>, in<8>, ci;
 7      output      out<8>, co, vo, no, zo;
 8
 9      /** contents **/
10      sel         edgebit;
11
12      /** operations of instrin **/
13      instruct do par {
14          any { /* edgebit */
15              mode == 0b000 : edgebit = in<7>;
16              mode == 0b001 : edgebit = 0b0;
17              mode == 0b010 : edgebit = 0b0;
18              mode == 0b011 : edgebit = 0b0;
19              mode == 0b100 : edgebit = ci;
20              mode == 0b101 : edgebit = ci;
21              mode == 0b110 : edgebit = in<0>;
22              mode == 0b111 : edgebit = in<7>;
23          }
24          any { /* out, co */
25              mode<0> == 0b0 : par { /* right */
26                  out = edgebit || in<7:1>;
27                  co = in<0>;
28              }
29              mode<0> == 0b1 : par { /* left */
30                  out = in<6:0> || edgebit;
31                  co = in<7>;
32              }
33          }
34          any { /* vo */
35              mode<1:0> == 0b01 : vo = out<7> @ in<7>; /* SLA, RLA */
36              else : vo = 0b0;
37          }
38          no = out<7>;
39          zo = ^(/| out);
40      }
41  }
```

SECONDS を用いてシミュレーション

モジュール kueshift の SFL 記述が完成したら，SECONDS を用いてその動作を確かめてみます．リスト 2.2 に，そのための SECONDS のコマンド列を用意しましたので，これを利用してください．3 行目では SFL 記述を読み込み，4 行目ではシミュレーションイメージを構築しています．6 行目から 8 行目では，レポートの書式を設定しています．report do でレポートを表示させます．set で端子に値を設定します．

リスト 2.2: SECONDS へのコマンド列 (kueshift.sec)

```
 1  # simulation data for kueshift
 2
 3  sflread kueshift.sfl
 4  autoinstall kueshift
 5
 6  rpt_add ext \
 7  "do=%B mode=%B in=%B ci=%B    out=%B cvnz=%B%B%B%B\n" \
 8  do mode in ci out co vo no zo
 9
10  # sleep
11  report do
12  set mode 000; set in 00001111; set ci 1; report do
13
14  # execute
15  set do 1
16  set mode 000; set in 00001111; set ci 1; report do
17  set mode 001; set in 00001111; set ci 1; report do
18  set mode 010; set in 00001111; set ci 1; report do
19  set mode 011; set in 00001111; set ci 1; report do
20  set mode 100; set in 00001111; set ci 1; report do
21  set mode 101; set in 00001111; set ci 1; report do
22  set mode 110; set in 00001111; set ci 1; report do
23  set mode 111; set in 00001111; set ci 1; report do
24
25  set mode 000; set in 11110000; set ci 1; report do
26  set mode 001; set in 11110000; set ci 1; report do
27  set mode 010; set in 11110000; set ci 1; report do
28  set mode 011; set in 11110000; set ci 1; report do
29  set mode 100; set in 11110000; set ci 1; report do
30  set mode 101; set in 11110000; set ci 1; report do
31  set mode 110; set in 11110000; set ci 1; report do
32  set mode 111; set in 11110000; set ci 1; report do
```

リスト 2.2 の内容を，kueshift.sec というファイルに保存して，コマンドプロンプトで，

```
% seconds < kueshift.sec
```

としてみてください．SFL 記述が正しければ，リスト 2.3 のような結果となるはずです．ここで，何も設定していないときの制御入力端子 do の値は 0 であることがわかります．このときはまだ do は起動されていません．従って，いくらデータ入力端子に値を設定しても，データ出力端子には結果が出力されません．一方，set コマンドで do に 1 を設定すると，結果がきちんと出力されました．これは，do が起動されて，シフト処理を行ったからです．また，値を設定しなければ，制御入力端子の場合はその値が 0 となっていますが，データ入力端子の場合はその値が z（不定）となっていることに注意してください．この違いは，一般に，制御端子とデータ端子にあてはまります．リスト 2.2 を適当に変更して，いろいろな値で動作を確認してみてください．

リスト 2.3: SECONDS の実行結果

```
do=0 mode=zzz in=zzzzzzzz ci=z    out=zzzzzzzz cvnz=zzzz
do=0 mode=000 in=00001111 ci=1    out=zzzzzzzz cvnz=zzzz
do=1 mode=000 in=00001111 ci=1    out=00000111 cvnz=1000
do=1 mode=001 in=00001111 ci=1    out=00011110 cvnz=0000
do=1 mode=010 in=00001111 ci=1    out=00000111 cvnz=1000
do=1 mode=011 in=00001111 ci=1    out=00011110 cvnz=0000
do=1 mode=100 in=00001111 ci=1    out=10000111 cvnz=1010
do=1 mode=101 in=00001111 ci=1    out=00011111 cvnz=0000
do=1 mode=110 in=00001111 ci=1    out=10000111 cvnz=1010
do=1 mode=111 in=00001111 ci=1    out=00011110 cvnz=0000
do=1 mode=000 in=11110000 ci=1    out=11111000 cvnz=0010
do=1 mode=001 in=11110000 ci=1    out=11100000 cvnz=1010
do=1 mode=010 in=11110000 ci=1    out=01111000 cvnz=0000
do=1 mode=011 in=11110000 ci=1    out=11100000 cvnz=1010
do=1 mode=100 in=11110000 ci=1    out=11111000 cvnz=0010
do=1 mode=101 in=11110000 ci=1    out=11100001 cvnz=1010
do=1 mode=110 in=11110000 ci=1    out=01111000 cvnz=0000
do=1 mode=111 in=11110000 ci=1    out=11100001 cvnz=1010
```

3　モジュール kuealu の設計

次は，KUE-CHIP2 の算術論理演算命令を処理する回路を，SFL のモジュール kuealu として設計します．さきほどの kueshift と同様に，モジュール kuealu は，KUE-CHIP2 全体を設計するときに，サブモジュール alu として利用されます．

KUE-CHIP2 の算術論理演算命令には，図 3に示すように，加算命令 (ADD, ADC)，減算命令 (SUB, SBC)，EOR, OR, AND 命令，比較命令 (CMP) があります．まず，EOR, OR, AND 命令については，SFL の演算子 @，|，& を使って簡単に処理できます．その他の命令では，8 ビットの加算あるいは減算を行う必要があります．KUE-CHIP2 では負の数を 2 の補数で表すことになっていますので，減算は減数の全ビットを反転させて加算を行うことで実現できます．ただしこの際に，桁上げを考慮しない場合 (SUB, CMP) はさらに 1 を加え，桁上げを考慮する場合 (SBC) はさらに桁上げフラグ CF の否定を加えます．従って，加算器さえ用意すればその他の命令は処理できます．

以上の考えに基づき，加算を行うモジュール add8 を定義し，モジュール kuealu の内部で，サブモジュールとして用いることにします．すなわち階層化設計を行うわけです．SFL 記述をリスト 3.1 に示します．これは，大きく分けて 3 つの部分，モジュール add8 の定義部，add8 のインタフェースの宣言部，モジュール kuealu の定義部から成ります．

モジュール add8 の定義部 (3 行目から 33 行目)

ここでは，モジュール add8 がどのようなものから構成され，どのように動作するかが記述されています．

5 行目から 7 行目には，外部端子の定義が行われています．制御入力端子としては kueshift と同様に do を持ち，do の起動で加算を行うものとします．データ入力端子としては，a<8>，b<8>，ci を持ち，データ出力端子としては，out<8>，co，vo を持つものとします．ここで，out は a + b + ci の下位 8 ビットを，co は a + b + ci の 8 ビット目からの桁上げを出力するものとします．また，vo は桁あふれを示すものですが，a + b + ci の 8 ビット目からの桁上げと 8 ビット目への桁上げの排他的論理和を出力するものとします．

13 行目から 32 行目には，制御入力端子 do による動作が記述されています．多ビットの加算器の構成法には，桁上げ伝播方式のものや桁上げ先見方式のものなど，いろいろな種類のものがあります．ここでは，わかりやすいように，桁上げ伝播方式で構成しました．これは，1 ビットの全加算器を直列につないだようなものです．桁上げ伝播加算器は簡単な構成なのですが，直列に計算を行うので遅延時間が大きくなります．SFL 記述中のデータ内部端子 c0，...，c7 が各ビットの桁上げを保持します．

add8 のインタフェースの宣言部 (35 行目から 42 行目)

モジュール add8 は，モジュール kuealu の中でサブモジュールとして使用されます．そのためには，add8 のインタフェースの宣言を行う必要があります．モジュールの定義部がキーワード module で始まるのに対し，インタフェースの宣言部はキーワード declare で始まります．

インタフェースの宣言部では，そのモジュールが外部からどのように見えるかということが記述されます．従って，まず，外部端子の定義を行います．これは，モジュールの定義部のものと同じです．次に，必要であれは，制御入力端子の仮引数の定義をここで行います．制御入力端子 do は加算を行いますが，そのためには a<8>，b<8>，ci の値が必要ですので，これらを制御入力端子 do の仮引数とします．

モジュール kuealu の定義部 (44 行目から 79 行目)

以上で，モジュール add8 に関する記述が終わりましたので，モジュール kuealu の定義を行います．まずは，外部端子の定義を行います．制御入力端子 do は，kuealu に算術論理演算を依頼するためのものです．データ入力端子としては，算術論理演算の種類を示す mode<3>，被演算データ a<8> と b<8>，演算前の CF の値 ci を用意します．ここで，mode は，8 種類の算術論理演算命令のうちの 1 つを指定するものですが，表 1や図 3の mmm に対応させることにします．また，データ出力端子としては，演算結果を表す out<8> と，演算後の各フラグを表す co，vo，no，zo を用意します．

51 行目では，さきほどの設計したモジュール add8 をサブモジュール名 add として用いることを宣言しています．

54 行目から 78 行目が，制御入力端子 do による動作の記述です．66 行目などでは add.co として，サブモジュール add のデータ出力端子 co を参照しています．56 行目などでは，サブモジュール add の制御入力端子 do が起動されています．制御入力端子は，() をそのあとに書くことで起動されます．引数があれば，() の中に書きます．モジュール add8 の do は，41 行目に示すような仮引数を持ちますので，例えば，56 行目の add.do(a，^b，^ci) という記述は，

```
add.a = a; add.b = ^b; add.ci = ^ci; add.do();
```

ということを意味します．なお，add.do(a，^b，^ci).out という表現は，制御入力端子の起動とサブモジュールの外部端子の参照を 1 文にまとめたものです．

リスト 3.1 の説明は以上です．これらの SFL 記述は，kuealu.sfl というファイルに格納することにします．

リスト 3.1: SFL 記述 (kuealu.sfl)

```
/** kuealu: ALU for KUE-CHIP2 **/

module add8 {
    /** external pins **/
    instrin     do;
    input       a<8>, b<8>, ci;
    output      out<8>, co, vo;

    /** elements **/
    sel         c0, c1, c2, c3, c4, c5, c6, c7;

    /** operations of instrin **/
    instruct do par {
        c0 = (a<0> & b<0>) | (b<0> & ci) | (ci & a<0>);
        c1 = (a<1> & b<1>) | (b<1> & c0) | (c0 & a<1>);
        c2 = (a<2> & b<2>) | (b<2> & c1) | (c1 & a<2>);
        c3 = (a<3> & b<3>) | (b<3> & c2) | (c2 & a<3>);
        c4 = (a<4> & b<4>) | (b<4> & c3) | (c3 & a<4>);
        c5 = (a<5> & b<5>) | (b<5> & c4) | (c4 & a<5>);
        c6 = (a<6> & b<6>) | (b<6> & c5) | (c5 & a<6>);
        c7 = (a<7> & b<7>) | (b<7> & c6) | (c6 & a<7>);
        out = (a<7> @ b<7> @ c6) ||
              (a<6> @ b<6> @ c5) ||
              (a<5> @ b<5> @ c4) ||
              (a<4> @ b<4> @ c3) ||
              (a<3> @ b<3> @ c2) ||
              (a<2> @ b<2> @ c1) ||
              (a<1> @ b<1> @ c0) ||
              (a<0> @ b<0> @ ci);
        co = c7;
        vo = c7 @ c6;
    }
}

declare add8 {
    /** external pins **/
    instrin     do;
    input       a<8>, b<8>, ci;
    output      out<8>, co, vo;
    /** arguments of instrin **/
    instr_arg   do(a, b, ci);
}

module kuealu {
    /** external pins **/
    instrin     do;
    input       mode<3>, a<8>, b<8>, ci;
    output      out<8>, co, vo, no, zo;

    /** elements **/
    add8        add;

    /** operations of instrin **/
    instruct do par {
        any { /* out */
            mode == 0b000 : out = add.do(a, ^b, ^ci).out; /* SBC */
            mode == 0b001 : out = add.do(a, b, ci).out;   /* ADC */
            mode == 0b010 : out = add.do(a, ^b, 0b1).out; /* SUB */
            mode == 0b011 : out = add.do(a, b, 0b0).out;  /* ADD */
            mode == 0b100 : out = a @ b;                  /* EOR */
            mode == 0b101 : out = a | b;                  /* OR  */
            mode == 0b110 : out = a & b;                  /* AND */
            mode == 0b111 : out = add.do(a, ^b, 0b1).out; /* CMP */
        }
        any { /* co */
            mode == 0b000 : co = ^add.co; /* SBC */
            mode == 0b001 : co = add.co;  /* ADC */
            else : co = ci; /* SUB, ADD, EOR, OR, AND, CMP */
        }
        any { /* vo */
            mode == 0b100 : vo = 0b0; /* EOR */
            mode == 0b101 : vo = 0b0; /* OR  */
            mode == 0b110 : vo = 0b0; /* AND */
            else : vo = add.vo; /* SBC, ADC, SUB, ADD, CMP */
        }
        no = out<7>;
        zo = ^(/| out);
    }
}
```

SECONDS を用いてシミュレーション

kuealu の記述が完成したら，SECONDS を用いてその動作を確かめてみます．リスト 3.2 に，そのための SECONDS のコマンド列を用意しましたので，これを利用してください．

リスト 3.2: SECONDS へのコマンド列 (kuealu.sec)

```
 1  # simulation data for kuealu
 2
 3  sflread kuealu.sfl
 4  autoinstall kuealu
 5
 6  rpt_add ext \
 7  "do=%B mode=%B a=%X b=%X ci=%B out=%X cvnz=%B%B%B%B  " \
 8  do mode a b ci out co vo no zo
 9
10  rpt_add add \
11  "add[do=%B a=%X b=%X ci=%B cv=%B%B]\n" \
12  add.do add.a add.b add.ci add.co add.vo
13
14  # execute
15  set do 1
16
17  # xor, or, and
18  set mode 100; set a X33; set b X07; set ci 0; report do
19  set mode 101; set a X33; set b X07; set ci 0; report do
20  set mode 110; set a X33; set b X07; set ci 0; report do
21
22  # sub, add
23  set mode 010; set a X33; set b X07; set ci 0; report do
24  set mode 010; set a X07; set b X33; set ci 1; report do
25  set mode 011; set a X33; set b X07; set ci 0; report do
26  set mode 011; set a X07; set b X33; set ci 1; report do
27
28  # sbc, adc
29  set mode 000; set a X33; set b X07; set ci 0; report do
30  set mode 000; set a X07; set b X33; set ci 1; report do
31  set mode 001; set a X33; set b X07; set ci 0; report do
32  set mode 001; set a X07; set b X33; set ci 1; report do
```

リスト 3.2 の内容を，kuealu.sec というファイルに保存して，コマンドプロンプトで，

```
% seconds < kuealu.sec
```

としてみてください．SFL 記述が正しければ，リスト 3.3 のような結果となるはずです．ここには，kuealu の外部端子だけではなく，add の外部端子の値も表示しています．命令の種類が EOR, OR, AND の時は，加算を行う必要がないため，add が動作していないことがわかります．また，加算器 add で減算を行っているとき，add.b の値や桁上げフラグの値がどうなっているかに着目してください．

リスト 3.2 を適当に変更して，いろいろな値で動作を確認してみてください．

リスト 3.3: SECONDS の実行結果

```
do=1 mode=100 a=33 b=07 ci=0 out=34 cvnz=0000  add[do=0 a=zz b=zz ci=z cv=zz]
do=1 mode=101 a=33 b=07 ci=0 out=37 cvnz=0000  add[do=0 a=zz b=zz ci=z cv=zz]
do=1 mode=110 a=33 b=07 ci=0 out=03 cvnz=0000  add[do=0 a=zz b=zz ci=z cv=zz]
do=1 mode=010 a=33 b=07 ci=0 out=2c cvnz=0000  add[do=1 a=33 b=f8 ci=1 cv=10]
do=1 mode=010 a=07 b=33 ci=1 out=d4 cvnz=1010  add[do=1 a=07 b=cc ci=1 cv=00]
do=1 mode=011 a=33 b=07 ci=0 out=3a cvnz=0000  add[do=1 a=33 b=07 ci=0 cv=00]
do=1 mode=011 a=07 b=33 ci=1 out=3a cvnz=1000  add[do=1 a=07 b=33 ci=0 cv=00]
do=1 mode=000 a=33 b=07 ci=0 out=2c cvnz=0000  add[do=1 a=33 b=f8 ci=1 cv=10]
do=1 mode=000 a=07 b=33 ci=1 out=d3 cvnz=1010  add[do=1 a=07 b=cc ci=0 cv=00]
do=1 mode=001 a=33 b=07 ci=0 out=3a cvnz=0000  add[do=1 a=33 b=07 ci=0 cv=00]
do=1 mode=001 a=07 b=33 ci=1 out=3b cvnz=0000  add[do=1 a=07 b=33 ci=1 cv=00]
```

4 KUE-CHIP2 の SFL 記述

いよいよ KUE-CHIP2 全体を，SFL のモジュール kuechip2 として記述します．リスト 4.1 に一例を示します．

ファイルのインクルード（3 行目から 5 行目）

モジュール kuechip2 では，これまでに設計したモジュール kueshift と kuealu をサブモジュールとして用います．それらの SFL 記述は，必ずしもここにインクルードする必要はないのですが，インクルードすると論理合成の際に便利です．%i と空白に続けて "と "でファイル名を囲むことで，カレントディレクトリから見たパス名のファイルをインクルードします．

図 1のブロック図には，INC という 8 ビットのインクリメンタがあります．8 ビットのインクリメンタはまだ設計していないのですが，PARTHENON 標準ライブラリの中に inc8 という 8 ビットのインクリメンタが存在しますので，これを用いることにします．PARTHENON 標準ライブラリは，$PARTHENON/sfl_lib.dir というディレクトリに，多くのファイルとして格納されています．%i と空白に続けて <と> でファイル名を囲むことで，このディレクトリにあるファイルをインクルードします．

inc8.h というファイルの中身は，リスト 4.2 のようになっています．inc8 は，機能回路として定義されています (12 行目から 17 行目)．機能回路はモジュールのようなものですが，論理合成されない，使える演算子と構成要素が拡張されているなどの点でモジュールとは異なっています．機能回路の記述は，circuit で始めます．inc8 などの PARTHENON 標準ライブラリは，すでに論理合成されてそのネットリストが用意されていますので，機能回路で定義されているわけです．

インタフェースの宣言（7 行目から 23 行目）

モジュール kuechip2 では，これまでに設計したモジュール kueshift，kuealu や PARTHENON 標準ライブラリの inc8 を，それぞれサブモジュール shifter，alu，inc として用います．そのためにはインタフェースの宣言が必要です．kueshift，kuealu については，ここで行っています．また inc8 については，inc8.h（リスト 4.2) の 5 行目から 10 行目で宣言されています．なお，kueshift の制御入力端子 do の仮引数と kuealu の制御入力端子 do の仮引数は，共にそれらのすべてのデータ入力端子とします．

マクロ定義（25 行目から 38 行目）

SFL では，%d を用いてマクロ定義を行います．たとえば，NOP という文字列は，(ir<7:3> == 0b00000) という文字列に置き換えられます．ここではマクロ定義を用いて，表 1のフェーズ表に従って命令の分類を行っています．

外部端子の定義（42 行目から 48 行目）

ここからは，モジュール kuechip2 の定義部の記述となります．まずは，外部端子を記述します．図 1のブロック図に従って，外部端子を書き下せばよいのですが，どれを制御端子にしてどれをデータ端子にするかということを決める必要があります．

まず入力端子に関してですが，kuechip2 を起動するための start を制御入力端子とし，メモリや IBUF の値を読み込むための dbi<8>，IBUF_FLAG, OBUF_FLAG を読み込むための ibuf_flg_in，obuf_flg_in をデータ入力端子とします．出力端子に関しては，メモリの値の読み出しを依頼する mem_re，メモリへ値の書き込みを依頼する mem_we，IBUF の値の読み出しを依頼する ibuf_re，OBUF へ値の書き込みを依頼する obuf_we は制御出力端子にします．また，KUE-CHIP2 の動作状態を示す op，p0，p1，p2 も制御出力端子とします．そして，メモリや OBUF に値を書き込むための dbo<8>，メモリのアドレスを指定するための ab<9> はデータ出力端子とします．

構成要素の定義（51 行目から 59 行目）

次は kuechip2 の構成要素を記述します．まずレジスタに関しては，図 1に示したように，acc，ix，cf，vf，nf，zf，pc，ir，mar を用意します．KUE-CHIP2 のレジスタはリセット信号でリセットされるので，レジスタとしては reg_wr (register with reset) を用いることにします．そのほかに，図 1に示されているものとしては，モジュール kueshift, kuealu を，それぞれサブモジュール shifter，alu として用い，PARTHENON 標準ライブラリの inc8 を，サブモジュール inc として用います．また，データ内部端子として，sel_b を用意します．

そのほかに，図 1にはありませんが，SFL 記述を簡潔にするためのいくつかの構成要素を定義します．bcond_calc，bcond_status は，BRANCH 命令を処理するために設けた制御内部端子です．exec_alu は，alu を用いて算術論理演算命令を処理するための制御内部端子です．制御内部端子は，制御入力端子や制御出力端子と同様に，起動されなければその値は 0 で，起動されるとその値が 1 になります．また，動作を関連づけることもできます．これらの働きについてはもう少し後で説明します．

制御端子の仮引数の定義（62 行目から 65 行目）

制御端子のうち，制御出力端子と制御内部端子の仮引数の定義は，モジュールの定義部で行います．制御出力端子と制御内部端子は，これらを持つモジュールによって起動されるからです．

mem_re は ab を仮引数として持つものとします．メモリの内容を読みだすためには，アドレスを与えなければならないからです． mem_we については，アドレスだけではなく，メモリに書き込むべきデータの値も与えなければならないので， ab と dbo を仮引数とします． obuf_we については，書き込むべきデータの値だけを与えれば良いので，dbo を仮引数とします．

exec_alu は sel_b を仮引数として持つものとします．図 1に示すように， sel_b は alu の入力の 1 つとなるからです．

ステージとタスクと仮引数の定義（68 行目）

SFL には，ステージという概念があります．制御端子は 1 クロックで終了する動作を行うのに対し，ステージは複数クロックにまたがる一連の動作を行います．すなわち，ステージは状態を持ちますが，制御端子は状態を持ちません．

ステージを複数個用意すると，別々の動作主体による並列動作をうまく記述することができます．しかし，ここではステージを一つだけ用意し，その唯一のステージの名前を all とします．ステージ all は，表 1に示されたフェーズ表に従って，各命令の動作を制御するものとします．

ステージにはタスクというものが存在し，これが起動されることでステージが動作すると考えられています．ステージ all にもタスクが必要です． t という名前のタスクを用意します．

制御端子による動作（71 行目から 100 行目）

次に，制御端子による動作を記述します．制御端子のうち，制御入力端子と制御内部端子による動作の定義は，これらを持つモジュールの定義部で行われます．一方，制御出力端子による動作の定義は，その制御出力端子を持つモジュールの外，つまりそのモジュールをサブモジュールとして使用するモジュールで行われます．従って，ここでは行われません．

制御入力端子 start が起動されると， generate という SFL のキーワードにより，ステージ all のタスク t が起動され， KUE-CHIP2 が動き出します．

制御内部端子 bcond_calc は，命令コードとフラグの値に従って， BRANCH 命令の分岐条件を判定します．分岐条件の詳細については，規定課題の資料を見てください．もし条件が成り立てば，制御内部端子 bcond_status を起動して 1 にします．条件が成り立たなければ， bcond_status は 0 のままです．

制御内部端子 exec_alu は， alu を用いて算術論理演算命令を行い，フラグを更新します．算術論理演算命令の第 1 オペランドは， ir<3> が 1 か 0 かで， ix か acc となります．命令の第 2 オペランドは， exec_alu の仮引数 sel_b に与えられた値となります．演算結果は， ir<3> が 1 か 0 かで， ix か acc に格納されます（比較命令 CMP 以外）．

ステージの動作（103 行目から 205 行目）

最後に，ステージ all の動作を記述します．ステージは generate で起動され，起動された次のクロックから動作を開始し， finish というキーワードで動作を終了します．したがって，ステージ all は制御入力端子 start によって動作を開始し， HLT 命令により動作を終了します．

ステージ all は 3 つの状態を持つものとし，表 1に示した 3 つのフェーズ ph0，ph1，ph2 をそれぞれの状態に対応させます．状態 ph0 では，命令の種類によらず動作は一定ですが，状態 ph1，ph2 での動作は命令の種類によって異ってきます．表 1と見比べることで， SFL 記述の意味がわかりやすくなると思います．

状態を持つステージでは，どれかの 1 つの状態のみが有効になっています．ステージが動作中であっても，有効でない状態の動作は行われません．初めに有効な状態（初期状態）は，キーワード first_state で指定します．有効な状態の変更（状態遷移）は，キーワード goto によって行います．

制御端子の起動

kuechip2 の SFL 記述には，制御端子の起動がたくさん現れています．制御端子は，その後に () を付けることで起動します．制御端子は，制御入力端子，制御出力端子，制御内部端子に分けられますが，ここでは，それぞれの起動について説明します．

まず，制御入力端子についてですが，サブモジュールの制御入力端子は，これを使用している外側のモジュールが起動します．例えば 145 行目では，サブモジュール shifter の制御入力端子 do を， ir<2:0>，ix，cf を引数として起動しています．これにより， kueshift の do の仮引数の定義（13 行目）に従って， shifter のデータ入力端子 mode，in，ci に，それぞれ， ir<2:0>，ix，cf の値が転送されます．また， shifter の do が 1 になり，それに対応づけられた動作が行われます．この動作は，モジュール kueshift の定義部に記述されています．このように，制御入力端子を用いて，あるモジュールからそこで使用しているサブモジュールに仕事を依頼します．

次に，制御出力端子について見てみます．例えば 109 行目では，制御出力端子 mem_re を， 0b0 || pc を引数として起動しています．これにより， 62 行目の仮引数の定義に従って，データ出力端子 ab に 0b0 || pc の値が転送されます．また， mem_re が 1 になり，それに対応づけられた動作が行われます．この動作とは，メモリの ab 番地の内容を dbi に転送することですが，これはモジュール kuechip2 の外側で規定されます．このように，制御出力端子を用いて，あるモジュールから外側のモジュールに仕事を依頼します．

最後に制御内部端子について見てみます．例えば 161 行目では， ix を引数として制御内部端子 exec_alu を起動しています．これにより， 65 行目の仮引数の定義に従って，データ内部端子 sel_b に ix の値が転送されます．また， exec_alu が 1 になり，それに対応づけられた動作が行われます．この動作は 90 行目から 100 行目に書かれています．制御内部端子は，制御入力端子や制御出力端子のように別のモジュールに仕事を依頼するものではなく，一連のまとまった動作を記述するためのものです．これを用いることで SFL 記述を簡潔にすることができます．

リスト 4.1 の説明は以上です．これらの SFL 記述は， kuechip2.sfl というファイルに格納するものとします．

リスト 4.1: SFL 記述 (kuechip2.sfl)

```
 1  /** kuechip2: The main module of KUE-CHIP2 **/
 2
 3  %i "kueshift.sfl"
 4  %i "kuealu.sfl"
 5  %i <inc8.h>
 6
 7  declare kueshift {
 8      /** external pins **/
 9      instrin     do;
10      input       mode<3>, in<8>, ci;
11      output      out<8>, co, vo, no, zo;
12      /** arguments of instrin **/
13      instr_arg   do(mode, in, ci);
14  }
15
16  declare kuealu {
17      /** external pins **/
18      instrin     do;
19      input       mode<3>, a<8>, b<8>, ci;
20      output      out<8>, co, vo, no, zo;
21      /** arguments of instrin **/
22      instr_arg   do(mode, a, b, ci);
23  }
24
25  %d NOP    (ir<7:3> == 0b00000)
26  %d HLT    ((ir<7:3> == 0b00001) | (ir<7:4> == 0b0101))
27  %d OUT    (ir<7:3> == 0b00010)
28  %d IN     (ir<7:3> == 0b00011)
29  %d SRCF   (ir<7:4> == 0b0010)
30  %d BRANCH (ir<7:4> == 0b0011)
31  %d SHIFT  (ir<7:4> == 0b0100)
32  %d LD_REG ((ir<7:4> == 0b0110) & (ir<2:1> == 0b00))
33  %d AL_REG ((ir<7> == 0b1)      & (ir<2:1> == 0b00))
34  %d LD_IMM ((ir<7:4> == 0b0110) & (ir<2:1> == 0b01))
35  %d AL_IMM ((ir<7> == 0b1)      & (ir<2:1> == 0b01))
36  %d LD_MA  ((ir<7:4> == 0b0110) & (ir<2> == 0b1))
37  %d ST_MA  ((ir<7:4> == 0b0111) & (ir<2> == 0b1))
38  %d AL_MA  ((ir<7> == 0b1)      & (ir<2> == 0b1))
39
40  module kuechip2 {
41      /** external pins **/
42      instrin     start;
43      input       dbi<8>;
44      input       ibuf_flg_in, obuf_flg_in;
45      instrout    mem_we, mem_re;
46      instrout    ibuf_re, obuf_we;
47      instrout    op, p0, p1, p2;
48      output      dbo<8>, ab<9>;
49
50      /** elements **/
51      reg_wr      acc<8>, ix<8>;
52      reg_wr      cf, vf, nf, zf;
53      reg_wr      pc<8>, ir<8>, mar<8>;
54      kuealu      alu;
55      kueshift    shifter;
56      inc8        inc;
57      sel         sel_b<8>;
58      instrself   bcond_calc, bcond_status;
59      instrself   exec_alu;
60
61      /** arguments of instrout and instrself **/
62      instr_arg   mem_re(ab);
63      instr_arg   mem_we(ab, dbo);
64      instr_arg   obuf_we(dbo);
```

```
65      instr_arg   exec_alu(sel_b);
66
67      /** stages and tasks and their arguments **/
68      stage_name  all { task t(); }
69
70      /** operations of instrin and instrself **/
71      instruct start generate all.t();
72      instruct bcond_calc if (
73          ( ir<3:0> == 0b0000 ) |
74          ( ( ir<3:0> == 0b1000 ) & vf ) |
75          ( ( ir<3:0> == 0b0001 ) & ^zf ) |
76          ( ( ir<3:0> == 0b1001 ) & zf ) |
77          ( ( ir<3:0> == 0b0010 ) & ^nf ) |
78          ( ( ir<3:0> == 0b1010 ) & nf ) |
79          ( ( ir<3:0> == 0b0011 ) & ^(nf | zf) ) |
80          ( ( ir<3:0> == 0b1011 ) & nf | zf ) |
81          ( ( ir<3:0> == 0b0100 ) & ^ibuf_flg_in ) |
82          ( ( ir<3:0> == 0b1100 ) & obuf_flg_in ) |
83          ( ( ir<3:0> == 0b0101 ) & ^cf ) |
84          ( ( ir<3:0> == 0b1101 ) & cf ) |
85          ( ( ir<3:0> == 0b0110 ) & ^(vf @ nf) ) |
86          ( ( ir<3:0> == 0b1110 ) & vf @ nf ) |
87          ( ( ir<3:0> == 0b0111 ) & ^((vf @ nf) | zf) ) |
88          ( ( ir<3:0> == 0b1111 ) & (vf @ nf) | zf )
89                              ) bcond_status();
90      instruct exec_alu par {
91          any {
92              ir<3> : alu.do(ir<6:4>, ix, sel_b, cf);
93              else : alu.do(ir<6:4>, acc, sel_b, cf);
94          }
95          if ( ^(ir<6:4> == 0b111) ) any { /* except CMP */
96              ir<3> : ix := alu.out;
97              else : acc := alu.out;
98          }
99          cf := alu.co; vf := alu.vo; nf := alu.no; zf := alu.zo;
100     }
101
102     /** operations of stages **/
103     stage all {
104         state_name ph0, ph1, ph2;
105         first_state ph0;
106         op();
107         state ph0 par {
108             p0();
109             ir := mem_re(0b0 || pc).dbi;
110             pc := inc.do(pc).out;
111             goto ph1;
112         }
113         state ph1 par {
114             p1();
115             any {
116                 NOP : goto ph0;
117                 HLT : par {
118                     goto ph0;
119                     finish;
120                 }
121                 OUT : par {
122                     obuf_we(acc);
123                     goto ph0;
124                 }
125                 IN : par {
126                     acc := ibuf_re().dbi;
127                     goto ph2;
128                 }
129                 SRCF : par {
130                     cf := ir<3>;
131                     goto ph0;
132                 }
133                 BRANCH : par {
134                     bcond_calc();
135                     mem_re(0b0 || pc);
136                     inc.do(pc);
137                     any {
138                         bcond_status : pc := dbi;
139                         else : pc := inc.out;
140                     }
141                     goto ph0;
142                 }
143                 SHIFT : par {
144                     any {
145                         ir<3> : ix := shifter.do(ir<2:0>, ix, cf).out;
146                         else : acc := shifter.do(ir<2:0>, acc, cf).out;
147                     }
```

```
148                     cf := shifter.co; vf := shifter.vo;
149                     nf := shifter.no; zf := shifter.zo;
150                     goto ph0;
151                 }
152                 LD_REG : par {
153                     any {
154                         ir<0> & ^ir<3> : acc := ix;
155                         ^ir<0> & ir<3> : ix := acc;
156                     }
157                     goto ph0;
158                 }
159                 AL_REG : par {
160                     any {
161                         ir<0> : exec_alu(ix);
162                         else : exec_alu(acc);
163                     }
164                     goto ph0;
165                 }
166                 LD_IMM : par {
167                     any {
168                         ir<3> : ix := mem_re(0b0 || pc).dbi;
169                         else : acc := mem_re(0b0 || pc).dbi;
170                     }
171                     pc := inc.do(pc).out;
172                     goto ph0;
173                 }
174                 AL_IMM : par {
175                     exec_alu(mem_re(0b0 || pc).dbi);
176                     pc := inc.do(pc).out;
177                     goto ph0;
178                 }
179                 LD_MA | ST_MA | AL_MA : par {
180                     mem_re(0b0 || pc);
181                     any {
182                         ir<1> : mar := alu.do(0b011, ix, dbi, 0b0).out;
183                         else : mar := dbi;
184                     }
185                     pc := inc.do(pc).out;
186                     goto ph2;
187                 }
188             }
189         }
190         state ph2 par {
191             p2();
192             any {
193                 LD_MA : any {
194                     ir<3> : ix := mem_re(ir<0> || mar).dbi;
195                     else : acc := mem_re(ir<0> || mar).dbi;
196                 }
197                 ST_MA : any {
198                     ir<3> : mem_we(ir<0> || mar, ix);
199                     else : mem_we(ir<0> || mar, acc);
200                 }
201                 AL_MA : exec_alu(mem_re(ir<0> || mar).dbi);
202             }
203             goto ph0;
204         }
205     }
206 }
```

リスト 4.2: inc8.h

```
 1 /*****************************************
 2 * (C)Copyright by N.T.T 1993(unpublished) *
 3 * All rights are reserved.                *
 4 *****************************************/
 5 declare inc8 {
 6     input       in<8> ;
 7     output      out<8> ;
 8     instrin     do ;
 9     instr_arg do(in) ;
10 }
11
12 circuit inc8 {
13     input       in<8> ;
14     output      out<8> ;
15     instrin     do ;
16     instruct do out = in + 0b1 ;
17 }
```

5 KUE-CHIP2 の動作確認

KUE-CHIP2 全体の SFL 記述が完成したら，SECONDS を用いてその動作を確かめてみます．ここでは，KUE-CHIP2 の命令セットからなるプログラムを実際に動かしてみて，結果が正しいかどうかを見てみます．そのためには，KUE-CHIP2 本体とメモリなどが搭載されたボードのようなものを用意する必要がありますので，これを SFL のモジュール board として記述します．リスト 5.1 に一例を示します．

機能回路 r512_8

まず，KUE-CHIP2 が使用する 512 バイトのメモリを，機能回路 r512_8 として設計します．メモリは，機能回路でのみ構成要素として記述できます．

制御入力端子としては，読み出し，書き込みを行うための read，write を用意します．データ入力端子としては，アドレスを示す adrs<9>，書き込みデータを入力する din<8> を用意します．データ出力端子としては，読み出しデータを出力する dout<8> を用意します．

SFL では mem というキーワードでメモリを表します．ここでは，cell という名前の 512 バイト (8 ビットを 1 バイトとする) のメモリを宣言しています．あとは，制御入力端子 read と write の動作を記述します．cell[adrs] で，cell というメモリの adrs 番地を示します．

インタフェースの宣言

モジュール board では，モジュール kuechip2 と機能回路 r512_8 をサブモジュールとして使用しますので，ここでこれらのインタフェースの宣言を行う必要があります．モジュール kuechip2 は，制御入力端子 start を持ちますが，仮引数はありませんので，仮引数の定義に関しては何も書きません．従って，外部端子の定義のみを書きます．機能回路 r512_8 については，read の仮引数を adrs，write の仮引数を adrs と din とします．

モジュール board の定義部

以上で，kuechip2 とメモリ r512_8 の SFL 記述ができましたので，IBUF, IBUF_FLAG, OBUF, OBUF_FLAG と共に，これらに関係を持たせましょう．モジュール board がその役目を果たします．

モジュール board には外部端子を用意しません．SECONDS では，メモリに直接値を書き込んだり，kuechip2 の端子に値を設定できるからです．

従って，構成要素の定義から始めます．モジュール board は，モジュール kuechip2，機能回路 r512_8 を，サブモジュール名 cpu，mem として利用します．また，ibuf，ibuf_flag，obuf，obuf_flag は，reg_wr で実現することにします．

あとは，動作の記述です．初めのpar{ }で囲まれた部分には，常に行われる動作が記述されています．つまり，ibuf_flag，obuf_flag の値は，常に cpu の ibuf_flg_in，obuf_flg_in に供給されるということです．49 行目から 58 行目には，cpu の制御出力端子の動作の定義が行われています．このように，あるモジュールの制御出力端子の動作の定義は，そのモジュールをサブモジュールとして用いる外側のモジュールで行われます．

リスト 5.1 の説明は以上です．これらの SFL 記述は board.sfl というファイルに格納することにします．

リスト 5.1: 動作確認のための SFL 記述 (board.sfl)

```
 1  /** board: KUE-CHIP2 Educational Board **/
 2  /** This description is only for simulation using SECONDS **/
 3
 4  circuit r512_8 {
 5      /** external pins **/
 6      instrin     read, write;
 7      input       adrs<9>, din<8>;
 8      output      dout<8>;
 9      /** elements **/
10      mem         cell[512]<8>;
11      /** operations of instrin **/
12      instruct    read dout = cell[adrs];
13      instruct    write cell[adrs] := din;
14  }
15
16  declare kuechip2 {
17      /** external pins **/
18      instrin     start;
19      input       dbi<8>;
20      input       ibuf_flg_in, obuf_flg_in;
21      instrout    mem_we, mem_re;
22      instrout    ibuf_re, obuf_we;
23      output      dbo<8>, ab<9>;
24  }
25
26  declare r512_8 {
27      /** external pins **/
```

```
28      instrin     read, write;
29      input       adrs<9>, din<8>;
30      output      dout<8>;
31      /** arguments of instrin **/
32      instr_arg   read(adrs);
33      instr_arg   write(adrs, din);
34  }
35
36  module board {
37      /** elements **/
38      reg_wr      ibuf<8>, ibuf_flg, obuf<8>, obuf_flg;
39      kuechip2    cpu;
40      r512_8      mem;
41
42      /** operations in common **/
43      par {
44          cpu.ibuf_flg_in = ibuf_flg;
45          cpu.obuf_flg_in = obuf_flg;
46      }
47
48      /** operations of instrout **/
49      instruct cpu.mem_we mem.write(cpu.ab, cpu.dbo);
50      instruct cpu.mem_re cpu.dbi = mem.read(cpu.ab).dout;
51      instruct cpu.ibuf_re par {
52          cpu.dbi = ibuf;
53          ibuf_flg := 0b0;
54      }
55      instruct cpu.obuf_we par {
56          obuf := cpu.dbo;
57          obuf_flg := 0b1;
58      }
59  }
```

1 から N までの和を計算するプログラム

これで，KUE-CHIP2 のプログラムを SECONDS を用いて実行してみるための SFL 記述が整いました．

まず最初は，1 から N までの和を計算する KUE-CHIP2 のプログラムで動作を確認してみます．リスト 5.2 にアセンブルリストを示します．このプログラムは，X080 番地の値を N として，1 から N までの和を計算して，結果を X081 番地に格納するというものです．

リスト 5.2: 1 から N までの和を計算する KUE-CHIP2 のプログラム

```
 1  *** KUE-CHIP2 Assembler ver.1.0   by H.Ochi ***
 2
 3  * Sum from 1 to N
 4  * Programmed by Akira Uejima, May. 5, 1992
 5  * e-mail: uejima@mics.cs.ritsumei.ac.jp
 6
 7  * N
 8   80 :           N:      EQU             80H
 9  * Sum(result)
10   81 :           SUM:    EQU             81H
11
12   00 :   6C 80           LD      IX,     [N]
13   02 :   62 00           LD      ACC,    0
14   04 :   B1      LOOP:   ADD     ACC,    IX
15   05 :   AA 01           SUB     IX,     1
16   07 :   33 04           BP              LOOP
17   09 :   74 81           ST      ACC,    [SUM]
18   0B :   0F              HLT
19
20                          END
```

リスト 5.3 は，このプログラムを SECONDS で実行するためのコマンド列です．SECONDS には memset というメモリに値を書き込む便利なコマンドがあります．1 つめの memset では，1 から N までの和を計算する KUE-CHIP2 のプログラムを，X000 番地を始点としてメモリに書き込んでいます．2 つめの memset では，X080 番地に N の値を書き込んでいます．最後の行では，1 から N までの和が格納されている X081 番地の内容を表示しています．

リスト 5.3: SECONDS へのコマンド列

```
1  # sum of 1, 2, ..., n
2
3  sflread kuechip2.sfl
4  sflread board.sfl
5  autoinstall board
6
7  # program
8  memset mem/cell X000 \
```

```
 9  X6c X80 X62 X00 Xb1 Xaa X01 X33 X04 X74 \
10  X81 X0f
11
12  # n
13  memset mem/cell X080 \
14  X0a
15
16  # report
17  rpt_add mem \
18  "[%3T] (%B)%S re=%B we=%B ab=%X dbi=%X dbo=%X " \
19  cpu/all.t cpu/all cpu.mem_re cpu.mem_we cpu.ab cpu.dbi cpu.dbo
20
21  rpt_add reg \
22  "pc=%X ir=%X acc=%X ix=%X flag=%B%B%B%B %B\n" \
23  cpu/pc cpu/ir cpu/acc cpu/ix cpu/cf cpu/vf cpu/nf cpu/zf
24
25  # execution
26  set cpu/start 1; forward +75;
27
28  # show result
29  print "%X\n" mem/cell@X081
```

リスト 5.3 の内容を，sum1n.sec というファイルに保存して，コマンドプロンプトで，

```
% seconds < sum1n.sec
```

としてみてください．実行結果は，リスト 5.4 のようになるはずです．この場合，N が X0a (10) なので，X81 番地の内容は X37 (55) となります．

リスト 5.4: SECONDS の実行結果

```
[1  ] (1)ph0 re=1 we=0 ab=000 dbi=6c dbo=zz pc=00 ir=00 acc=00 ix=00 flag=0000
[2  ] (1)ph1 re=1 we=0 ab=001 dbi=80 dbo=zz pc=01 ir=6c acc=00 ix=00 flag=0000
[3  ] (1)ph2 re=1 we=0 ab=080 dbi=0a dbo=zz pc=02 ir=6c acc=00 ix=00 flag=0000
[4  ] (1)ph0 re=1 we=0 ab=002 dbi=62 dbo=zz pc=02 ir=6c acc=00 ix=0a flag=0000
[5  ] (1)ph1 re=1 we=0 ab=003 dbi=00 dbo=zz pc=03 ir=62 acc=00 ix=0a flag=0000
[6  ] (1)ph0 re=1 we=0 ab=004 dbi=b1 dbo=zz pc=04 ir=62 acc=00 ix=0a flag=0000
[7  ] (1)ph1 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=05 ir=b1 acc=00 ix=0a flag=0000
[8  ] (1)ph0 re=1 we=0 ab=005 dbi=aa dbo=zz pc=05 ir=b1 acc=0a ix=0a flag=0000
[9  ] (1)ph1 re=1 we=0 ab=006 dbi=01 dbo=zz pc=06 ir=aa acc=0a ix=0a flag=0000
[10 ] (1)ph0 re=1 we=0 ab=007 dbi=33 dbo=zz pc=07 ir=aa acc=0a ix=09 flag=0000
                        :
                        :
                        :
[60 ] (1)ph0 re=1 we=0 ab=004 dbi=b1 dbo=zz pc=04 ir=33 acc=36 ix=01 flag=0000
[61 ] (1)ph1 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=05 ir=b1 acc=36 ix=01 flag=0000
[62 ] (1)ph0 re=1 we=0 ab=005 dbi=aa dbo=zz pc=05 ir=b1 acc=37 ix=01 flag=0000
[63 ] (1)ph1 re=1 we=0 ab=006 dbi=01 dbo=zz pc=06 ir=aa acc=37 ix=01 flag=0000
[64 ] (1)ph0 re=1 we=0 ab=007 dbi=33 dbo=zz pc=07 ir=aa acc=37 ix=00 flag=0001
[65 ] (1)ph1 re=1 we=0 ab=008 dbi=04 dbo=zz pc=08 ir=33 acc=37 ix=00 flag=0001
[66 ] (1)ph0 re=1 we=0 ab=009 dbi=74 dbo=zz pc=09 ir=33 acc=37 ix=00 flag=0001
[67 ] (1)ph1 re=1 we=0 ab=00a dbi=81 dbo=zz pc=0a ir=74 acc=37 ix=00 flag=0001
[68 ] (1)ph2 re=0 we=1 ab=081 dbi=zz dbo=37 pc=0b ir=74 acc=37 ix=00 flag=0001
[69 ] (1)ph0 re=1 we=0 ab=00b dbi=0f dbo=zz pc=0b ir=74 acc=37 ix=00 flag=0001
[70 ] (1)ph1 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
[71 ] (0)ph0 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
[72 ] (0)ph0 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
[73 ] (0)ph0 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
[74 ] (0)ph0 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
[75 ] (0)ph0 re=0 we=0 ab=zzz dbi=zz dbo=zz pc=0c ir=0f acc=37 ix=00 flag=0001
37
```

この結果を見ると，1 から 10 までの和を求めるのに，70 クロック要しています．今回設計した KUE-CHIP2 では，各命令を実行するのに 2 あるいは 3 クロックを要し，命令実行のオーバラップは行っていません．各命令に必要なクロック数を減らしたり，各命令の実行をオーバラップさせることで，kuechip2 を高速化していく余地はまだまだあります．例えば，時間 61 では，mem_re，mem_we ともに起動されておらず，アドレスバス ab が使用されていません．このとき，ir の値は b1 なので，ADD ACC，IX という命令が実行されています．レジスタ間の加算なので，メモリにはアクセスしていないというわけです．設計次第では，この隙に，次の命令をメモリから ir に読み込むということもできます．

別のプログラムで動作確認

規定課題の資料には，KUE-CHIP2 のプログラムが 2 つ掲載されています．時間的に余裕のある方は，これを用いた動作確認も行ってみてください．

6 FPGA を用いた実現

KUE-CHIP2 全体の SFL 記述が終わり，SECONDS を用いたシミュレーションもうまくいくようでしたら，FPGA をターゲットとして論理合成を行い，実際に動作させてみましょう．

KUE-CHIP2 教育用ボードと daughter ボード

写真 1 に，今回設計した回路を FPGA で実現した様子を示します．手前の大きなボードが KUE-CHIP2 教育用ボードです．ここには，本物の KUE-CHIP2 やメモリが装着されています．KUE-CHIP2 教育用ボードには外部へのコネクタがあり，製作した回路をここに接続して，本物の KUE-CHIP2 と置き換えて動作させることができます．奥左側のボードが KUE-CHIP2 daughter ボードです．今回設計した回路をこの中の ALTERA 社の FPGA に実現します．奥右側は，FPGA の回路データをパソコンからダウンロードするための装置です．

論理合成の流れ

まず，今回作成した SFL 記述を，PARTHENON の論理合成プログラムを用いて，EDIF 形式のネットリストに変換します．そして，そのファイルをパソコン側に持って行き，ALTERA 社の MAX+plusII というプログラムを用いて，FPGA にダウンロードすべき回路データを作成します．

PARTHENON での論理合成

PARTHENON では，SFLEXP や OPT_MAP などのいくつかのプログラムが連携して論理合成を行います．1 つずつ実行していくこともできますが，auto というコマンドを用いると，一連のプログラムを自動的に起動することができます．これは以下のようにして用います．

```
% auto kuechip2 nld4 ALTERA altera
```

実行すると，しばらくの間いろいろなメッセージが出力されますが，最終的には，kuechip2.4th/kuechip2.nld という NLD 形式のファイルや kuechip2.edf という EDIF 形式のファイルが出来るはずです．この auto というコマンドの意味は，`kuechip2` という SFL のモジュールを論理合成して，最終的に kuechip2.4th/kuechip2.nld というネットリストを作成してください，その際に使用するセルライブラリは，ALTERA 社の altera というセルライブラリです，ということです．以上で，PARTHENON 側での論理合成は終了です．kuechip2.edf というファイルをパソコン側に持っていきます．

MAX+plusII での論理合成

MAX+plusII で行うことは，次の 2 つのプロジェクトに分れます．

プロジェクト名 kuechip2

ここでは，PARTHENON が出力した EDIF ファイル kuechip2.edf をコンパイルして，MAX+plusII で部品として利用できるようにします．手順を以下に示します．

1. File → Project → Name メニューで kuechip2.edf を選ぶ．
2. Assign → Device メニューで FLEX8000 の EPF8636ALC84-3 を選ぶ．MAX+plusII → Compiler を起動する．PARTHENON が出力した EDIF 形式のファイルを MAX+plusII が解釈できるようにするために，Interfaces → EDIF Netlist Reader Settings... を起動する．そこで，Vendor: Custom とし，Customize で，VCC: VDD, GND: VSS, Library Mapping Files: partenon.lmf とする．以上の準備が完了したら，Start ボタンを押してコンパイルを行う．
3. MAX+plusII → Hierarchy Display を起動し，その中の edf をダブルクリックして Text Editor を開く．File → Create Default Symbol を起動すると，Compiler が起動して kuechip2 の Graphic Editor でのシンボルができる．

プロジェクト名 kue2

ここでは，プロジェクト名 kuechip2 で作った部品に外部端子を接続して，FPGA にダウンロードする回路データを作成します．手順を以下に示します．外部端子を一つずつ定義する作業は，非常に時間のかかるものです．したがって本実習では，外部端子を定義したファイル (kue2.gdf, kue2.acf) をあらかじめ用意しておきました．

1. File → Project → Name メニューで kue2.gdf を選ぶ．
2. MAX+plusII → Hierarchy Display を起動し，その中の gdf をダブルクリックすると Graphic Editor が開く．Symbol → Enter Symbol メニューで，先ほど部品として登録した kuechip2 を読み込み．既に用意されている外部端子を kuechip2 につなぐ (図 5)．

図5では，いくつかの外部端子にNOTゲートが接続されています．本物のKUE-CHIP2では，いくつかの制御端子がlow active (普段の値は1で指示を伝えるときに0となる) として定義されています．一方，SFLの制御端子は，普段の値は0で指示を伝えるときに1となります．この違いを修正するために，NOTゲートを接続しています．

3. Assign → Deviceメニューで FLEX8000 の EPF8636ALC84-3 を選ぶ．MAX+plusII → Compilerを起動し，Interfaces → EDIF Netlist Reader Settings... で，プロジェクト名 kuechip2 の時と同様に設定し，Startボタンを押してコンパイルを行う．
4. MAX+plusII → Programmerを起動する．もしHardware Setupが出てきたら，正しく設定を行う．これらの設定は，Options → Hardware Setupで変更できる．コネクタ等を正しく接続してConfigureボタンを押すと，回路データがFPGAにダウンロードされる．

実際に動作させる

以上で，今回設計した回路がFPGA上に実現されましたので，本当に動作するかどうか確かめてみます．FPGA上のKUE-CHIP2は，KUE-CHIP2教育用ボードの中の本物のKUE-CHIP2と置き換えて動作させることができます．ただし今回設計したものは，制御・観測機能やメモリにプログラムを書き込む機能などを備えていません．従って，本物のKUE-CHIP2を使ってメモリにプログラムを書き込み，その後切り替えてFPGA上のKUE-CHIP2を動作させます．手順を以下に示します．

1. **各種トグルスイッチの初期設定:** POWERをON，CLKを中立 (ボード上と外部の双方のKUE-CHIP2にクロックを供給)，WRをDATA (8ビットトグルスイッチDATAから値を書き込む)，IMCをNORMAL (アドレスの下位9ビットはKUE-CHIP2が与える)，MEMをEXT (KUE-CHIP2の外部にあるメモリを用いる) とする．
2. **プログラムの書き込み:** KUE-CHIP2教育用ボード上のKUE-CHIP2を用いてプログラムを書き込むので，トグルスイッチCHIPをBOARDにする．トグルスイッチSELを0にして，メモリのプログラム領域の値を表示する．プログラムはアドレス0から始まるため，プッシュスイッチRESETを押してアドレスを0にする．メモリに値を書き込むためには，トグルスイッチDATAに書き込みたい値を設定し，プッシュスイッチSETを押す．アドレスを1つ進めるためには，プッシュスイッチADRINCを押す．1つ戻すためにはADRDECを用いる．
3. **入力データの書き込み:** データを書き込むアドレスに移動するため，トグルスイッチSELを8にしてメモリアドレスレジスタMARの値を表示し，トグルスイッチDATAに移動したいアドレスを設定してプッシュスイッチSETを押す．その後，プログラム領域にデータを書き込む場合はトグルスイッチSELを0に，データ領域に書き込む場合は1にする．値の書き込みはプログラムを書き込む場合と同様に行う．
4. **プログラムの実行:** トグルスイッチCHIPをEXTに切り替えて，KUE-CHIP2をFPGAで実現したものに置き換える．プッシュスイッチEXT．RST (不揮発性RAMの上側) を押してFPGA上のKUE-CHIP2をリセットする．プッシュスイッチSSを押すとプログラムの実行を開始する．動作中はLEDランプOPが点灯し，フェーズランプP0，P1，P2のどれかが，現在実行中のフェーズに従って点灯する．ロータリスイッチCLK-FRQを回してクロックスピードを変えることで，目で追って確認できる程にまで速度を落とすことができる．
5. **計算結果の確認:** トグルスイッチCHIPをBOARDに切り替える．入力データの書き込みのときと同様にしてメモリアドレスレジスタMARの値を書き換え，計算結果が格納されているアドレスに移動する．プログラム領域に計算結果が書き込まれている場合はトグルスイッチSELを0に，データ領域に書き込まれている場合は1にして，計算結果を確認する．

5章のKUE-CHIP2の動作確認にある「1からNまでの和を計算するプログラム」や規定課題の資料にある2つのプログラムなどを，上記の手順で実際に処理させてみてください．もしもうまく行かないときは，NOPやHLTなどの基本的な命令から確認していくと良いでしょう．

写真 1: KUE-CHIP2 教育用ボードと daughter ボード

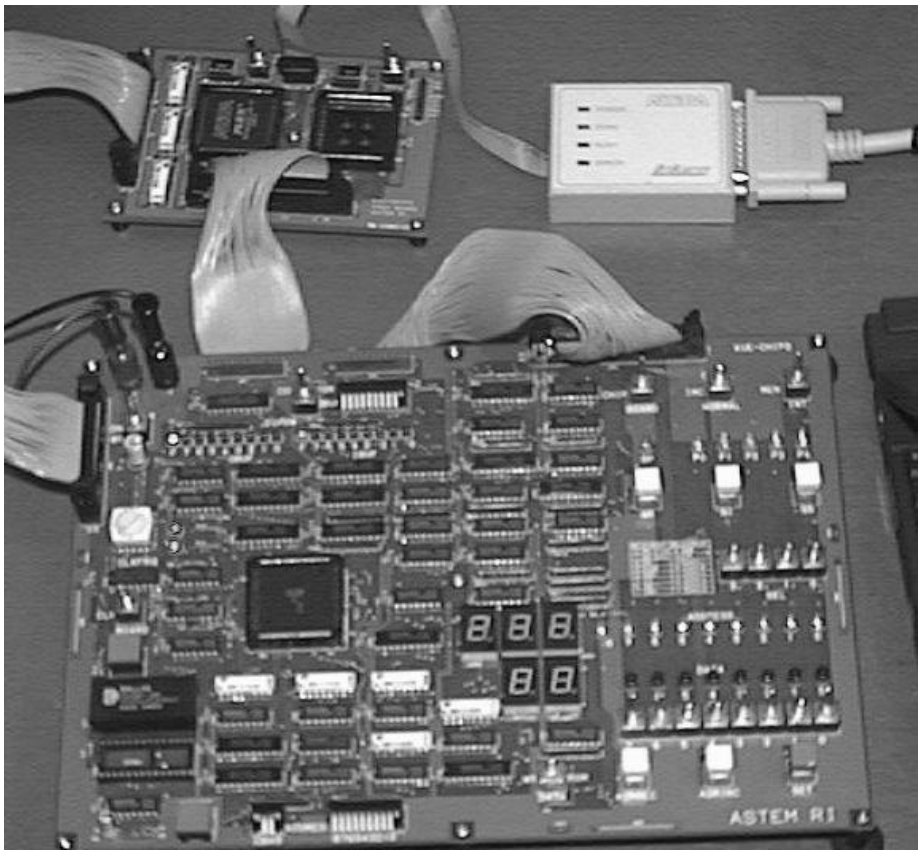
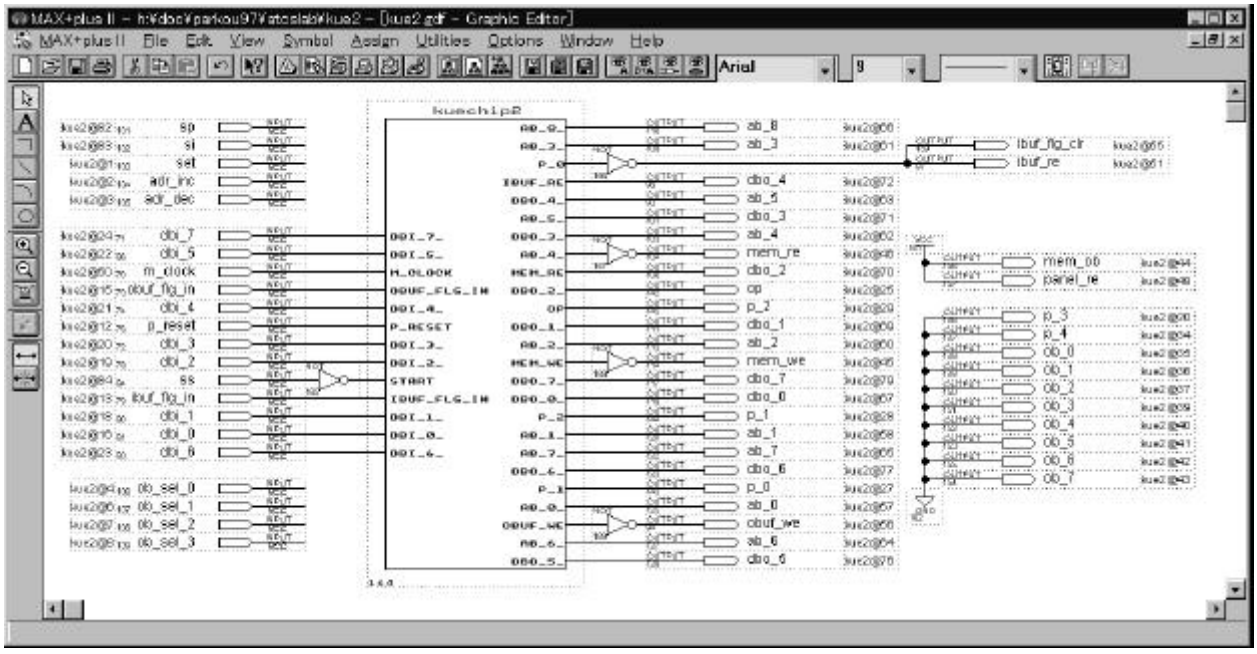


図 5: MAX+plusII Graphic Editor での外部端子の接続



7　ASIC をターゲットとした論理合成

最後に，ASIC をターゲットとして PARTHENON で論理合成を行います．ASIC をターゲットにすると，FPGA のように完成品をすぐに手にいれることは出来ません．そこで，OPT_MAP を用いて，面積や動作速度を見積もります．

auto コマンドで論理合成

FPGA をターゲットとした論理合成と同様に，auto コマンドを用いて論理合成を行います．ここでは，DEMO 社 demo ライブラリという ASIC 用のセルライブラリを用いますので，以下のように実行します．

```
% auto kuechip2 nld4 DEMO demo
```

出来上がった kuechip2 の面積と動作速度

論理合成が終わった kuechip2 は，どれくらいの面積で，どれくらいの動作速度なのかを見てみます．そのためには，以下のように実行します．

```
% opt_map kuechip2 kuechip2.4th $PARTHENON/cell_lib.dir/DEMO/demo/cell.dir
```

OPT_MAP が起動したあと，

```
OPT_MAP> move
        position   = /
        type       = NLD
        class_name = kuechip2
        power      = 3955.6
        area       = 521.93
        gates      = 1996
               :
OPT_MAP>
```

とすると，kuechip2 の面積や消費電力などがわかります．

次に，動作速度についてみてみます．外部にあるメモリの値を参照する場合としない場合とに分けて考えます．まず，以下のようにすることで，クロックが立ち上がってレジスタの値が変化し，その変化が回路全体に伝わるのに要する時間を求めることができます．これによると，m_clock が時刻 0 で立ち上がると（その他の入力端子は変化していない），時刻 4.9ns でレジスタ ir_reg5 の値が変化し終わり，時刻 46.6ns に oai--31_56.zn という端子が変化したのを最後に，それ以降は回路のどの部分も変化しないということがわかります．kuechip2.4th/kuechip2.nld を見ると，oai--31_56 の zn は零フラグ zf に接続されています．

```
OPT_MAP> scalc
    all statistics calculated
OPT_MAP> lcalc
    all load calculated
OPT_MAP> set /m_clock 0
    normal pin (kuechip2) m_clock  changed to source pin
OPT_MAP> dcalc
    all delay calculated
OPT_MAP> maxn /
    maximum rise delay path 1
        src  0  max 0.00000e+00
        nml  1  max 2.53030e+00  (/i       ) to (/z       )  /m_clock-buf0-0(bf3_i1)
        nml  2  max 4.90650e+00  (/m_clock ) to (/nout    )  /ir_reg5(dtr_reg)
        nml  3  max 6.92210e+00  (/i       ) to (/z       )  /ir_reg5-buf0-0(bf2_i1)
                      :
        nml 29  max 3.93413e+01  (/a2      ) to (/zn      )  /alu_oai-10(oa_i21)
        nml 30  max 4.07445e+01  (/a2      ) to (/zn      )  /alu_enor-1(xn1_i2)
        nml 31  max 4.21657e+01  (/a2      ) to (/z       )  /alu_eor-9(xo1_i2)
        nml 32  max 4.30537e+01  (/a2      ) to (/zn      )  /kuechip2_subm-1_nand--2_36(nd1_i2)
        nml 33  max 4.44177e+01  (/a1      ) to (/zn      )  /nand--4_28(nd1_i4)
        snk 34  max 4.65673e+01  (/a2      ) to (/zn      )  /oai--31_56(oa_i31)
OPT_MAP>
```

メモリの値を参照する場合はどうでしょか．まず，以下のようにして，m_clock が立ち上がってから mem_re や ab が変化し終わるまでの時間を求めます．これによると，m_clock が変化してからメモリ読み出しの準備が整うまでに 12.8ns かかることなどがわかります．

```
OPT_MAP> net /mem_re
    ------------------------------------------------------------------
    sink pin  mem_re
        maximum rise delay  1.27657e+01
        minimum rise delay  6.12230e+00
OPT_MAP> net /ab[0]
    ------------------------------------------------------------------
    sink pin  ab[0]
        maximum rise delay  8.12830e+00
        minimum rise delay  5.74430e+00
OPT_MAP> net /ab[8]
```

```
    ---------------------------------------------------------------
    sink pin  ab[8]
        maximum rise delay  7.86400e+00
        minimum rise delay  7.63830e+00
OPT_MAP>
```

そして， m_clock 以外の入力端子が変化してから，回路全体が変化し終わるまでの時間を求めます． init pin 0 ですべての外部入力端子に立ち上がりイベントを設定しています． unset で m_clock の立ち上がりイベントを消しています．この結果によると， m_clock 以外の入力端子が時刻 0 で立ち上がると，時刻 30.2ns に oai--31_56.zn という端子が変化して，それ以降は回路のどの部分も変化しないということがわかります．この場合も，零フラグ zf への経路が最大遅延を引き起こしています．

```
OPT_MAP> init pin 0
    top module pin initialized
        normal net    637
        in calc net     0
        source net     13
        inhibit net     0
        sink net       44
OPT_MAP> unset /m_clock
    source pin (kuechip2) m_clock  changed to normal pin
OPT_MAP> dcalc
    all delay calculated
OPT_MAP> maxn /
    maximum rise delay path 1
        src  0  max 0.00000e+00
        nml  1  max 6.36000e-01  (/a1      ) to (/zn      )  /sel_b_nand-2(nd1_i2)
        nml  2  max 1.36200e+00  (/a1      ) to (/zn      )  /sel_b_nand-29(nd1_i3)
        nml  3  max 1.98000e+00  (/a1      ) to (/zn      )  /sel-7_nand-6(nd1_i2)
                      :
        nml 21  max 2.29416e+01  (/a2      ) to (/zn      )  /alu_oai-10(oa_i21)
        nml 22  max 2.43448e+01  (/a2      ) to (/zn      )  /alu_enor-1(xn1_i2)
        nml 23  max 2.57660e+01  (/a2      ) to (/z       )  /alu_eor-9(xo1_i2)
        nml 24  max 2.66540e+01  (/a2      ) to (/zn      )  /kuechip2_subm-1_nand--2_36(nd1_i2)
        nml 25  max 2.80180e+01  (/a1      ) to (/zn      )  /nand--4_28(nd1_i4)
        snk 26  max 3.01676e+01  (/a2      ) to (/zn      )  /oai--31_56(oa_i31)
OPT_MAP>
```

仮に mem_re が変化してから dbi が変化するまで（メモリの値を読み出すまで）に 30.0ns かかるとすると， m_clock が変化してから zf にそれが伝わるまで 12.8ns+30.0ns+30.2ns=73.0ns かかるということになります．

デザインコンテストでは， KUE-CHIP2 のプログラムの実行クロック数を小さくすることも大切ですが，出来上がった KUE-CHIP2 の面積や動作周波数も重要な評価基準となるでしょう．今回は auto コマンドを一度実行しただけですが，動作速度などの設計条件をきちんと設定したり， OPT_MAP や ONSET に与えるコマンドを変更することで，論理合成結果を改善することができます．

実は，今回の KUE-CHIP2 の SFL 記述は，図 1のブロック図を忠実に反映しているわけではありません．図 1では，内部のセレクタとして， sel_a と sel_b が明示されていますが， SFL 記述では， sel_b しか明示していません．また， ACC や IX への入力は，一度 DBo にまとめられたものが供給されていますが， SFL 記述ではこのような書き方はしていません． SFL では， 1 つの転送先に対して複数の転送元がある場合，自動的にセレクタが挿入されます．例えば， acc に対しては， alu.out, dbi, shifter.out, ix からの転送があるので， 8 ビットの 4to1 セレクタが挿入されます．従って，現在のままの SFL 記述では，共有できるはずの多くのセレクタが共有されていません．図 1のブロック図を SFL 記述に忠実に反映させるだけでも， kuechip2 の面積は減少すると思われます．

また， alu の部分が最大遅延を引き起こしているようですが，これは，加算を行うモジュール add8 を桁上げ伝搬加算器で構成したからだと思われます． add8 を桁上げ先見加算器などに改良することで，最大遅延の値が減少することでしょう．

今回作成した SFL 記述には，そのほかにもいろいろと改良すべき点があります． SFL で設計すると，設計変更が容易に行えるので，いろんなアイデアをすぐに実現でき， SECONDS や論理合成プログラムですぐに評価することができます．今回の SFL 記述をたたき台として，すばらしい KUE-CHIP2 を設計していただきたいと思います．