情報科学実験C テキスト

updated on October 4, 2022

時限

秋・冬学期　月曜 4・5 限

指導教員

矢内　直人 (吹田・A305)

平井　健士 (吹田・B501)

山下 恭佑 (吹田・A302)

連絡先アドレス

lab-c-staff@ics.es.osaka-u.ac.jp

Web ページ

http://expweb.ics.es.osaka-u.ac.jp/lab-c/wiki.cgi

※本テキストは適宜更新される

Contents

Chapter 1

情報科学実験Cの概要

目的と内容

情報科学実験 C (以下，実験 C) は，ハードウェア設計 (特にハードウェア記述言語を用いた CPU 設計) と，その実装についての基礎知識，技術を習得することが目的である．
　その目的を達成するため，実験 C は以下の 2 つの実験から構成される．

- 実験 1：8 bit CPU (C-Processor) の設計と波形シミュレーションによる動作確認
- 実験 2：8 bit CPU (C-Processor) の FPGA ボードへの実装

なお，本実験を進めるに当たって「計算機援用工学 A」を同時に履修することを推奨する．
実験 2 で利用する FPGA ボードは，学籍番号下 2 桁の番号のものを利用すること．
下 3 桁が 700 台の場合は，90+下 2 桁のものを利用すること．

スケジュール

スケジュールは表 1.1 の通りである．別日に時間外実験は設定しない．

Table 1.1: 実験 C のスケジュール

日付	授業回数	実験内容 (対応するテキスト章番号)	課題
10/3(月) 4・5 限	1	[実験 1] 機能部品の設計と作成 (5 章)	必須
10/17(月) 4・5 限	2	[実験 1] データパスの作成 (5 章)	必須
10/24(月) 4・5 限	3	[実験 1] 制御部の設計 (5 章)	必須
10/31(月) 4・5 限	4	[実験 1] 制御部の設計 (5 章)	必須
11/14(月) 4・5 限	5	[実験 1] 制御部の作成 (5 章)	必須
11/21(月) 4・5 限	6	[実験 1] CPU 全体の動作テスト (5 章)	必須
11/28(月) 4・5 限	7	[実験 1] CPU 全体の動作テスト (5 章)	必須
11/29(火) 4・5 限	8	[実験 1] アセンブラプログラムの作成 (5 章)	必須
12/5(月) 4・5 限	9	[実験 1] 拡張課題 (5 章)	
12/12(月) 4・5 限	10	[実験 2] FPGA ボードへの CPU の実装 (6 章)	必須
12/19(月) 4・5 限	11	[実験 2] アセンブラプログラムの作成 (6 章)	必須
12/26(月) 4・5 限	12	[実験 2] CPU の改良 (6 章)	必須
1/16(月) 4・5 限	13	[実験 2] CPU の改良 (6 章)	必須
1/23(月) 4・5 限	14	[実験 2] アセンブラプログラムの作成 (6 章)	必須
1/30(月) 4・5 限	15	[実験 2] 拡張課題	
12/9(金) 17:00		中間レポート提出期限	
2/3(金) 17:00		最終レポート提出期限	

実施場所と出席確認

1.1 実施場所

・1 回目からは，状況に応じて対面で行う．
・状況に応じて対面もしくは自宅で FPGA ボードを用いた実装とする．

担当教員と TA (Teaching Assistant)

担当教員と TA は以下の通り．

- 教員：矢内 直人 (藤原研，准教授)
- 教員：平井 健士 (若宮研，助教)
- 教員：山下 恭佑 (藤原研，助教)
- TA：猪谷 幸永 (藤原研，博士前期課程 2 年)
- TA：加道 ちひろ (藤原研，博士前期課程 1 年)
- TA：小宮 千佳 (藤原研，博士前期課程 1 年)
- TA：山口 雄翔 (藤原研，博士前期課程 1 年)
- TA：小田 怜 (若宮研，博士前期課程 1 年)
- TA：藤岡 杏奈 (若宮研，博士前期課程 1 年)

課題

実験 C では，ほとんどの週で課題が設定されている．教員または TA がその週の課題を終了したかどうかを判定する．その週の課題が終了していれば，実験時間内であっても終了，帰宅してもよい．レポート作成のために，各課題の解答が必要になるので，課題が終了しても解答は保存しておくこと．

授業終了時刻より早く課題が終わり，合格の確認を受けた場合は，次週以降の課題に取りかかってもよい．次週以降の課題の確認は優先度が低く，授業時間内かつ教員・TA の手が空いている時のみ行う．

オンライン時での課題確認は，Zoom のブレイクアウトルームで行う．ブレイクアウトルームを 6 つ用意し，各部屋を TA が担当する．課題が完成したら，Zoom から TA に伝えること．

レポート

9 回目終了後に中間レポートを，15 回目終了後に最終レポートを提出してもらう．締切は下記に示す通りである．2 つともレポートを提出することが単位取得の必須要件である．なお，レポートのコピーが発覚した場合は，提供者も含めて即不合格にする場合がある．レポートは CLE から提出すること．

- 中間レポート：12/9 17:00 まで
- 最終レポート：2/3 17:00 まで[1]

成績評価

成績は，課題，宿題，中間レポート，最終レポートの出来および出席により決める．指定した時間内に課題が完了しなかった場合は減点する．中間レポート，最終レポート提出が遅れた場合は，大幅に減点する．拡張課題に取り組んだ場合は加点する．

質問

実験時間内に解らないことがあったら，教員または TA に質問すること．実験時間外に質問がある場合は，`lab-c-staff@ics.es.osaka-u.ac.jp` 宛にメールをすること．

[1] 飛び級希望の学生の場合，最終レポートの締切が早くなる可能性があるのでアナウンスに注意すること．

予習および自宅での学習方法

別日に時間外実験は設定しないため，入念に予習し，授業時間内に終わるようにすること．実験 1 で用いる ModelSim は”Starter Edition”，実験 2 で用いる Quartus Prime は”Lite Edition”と呼ばれる無償版が提供されている．

その他の補足事項

・設計記述やプログラムは，上書き保存で 1 つのファイルを更新していくだけではなく，時々バージョンの区切りをつけてコピーを保存すること．特に試験的な改変を加える前には一度現行のファイルセットをコピーして残しておくことを推奨する．・実験で作成したデータのバックアップや保存場所の選定などの管理は各個人の責任で行うこと．データ消失などのトラブルに対してはレポート提出期限延長などの救済措置は一切行わない．・実験 2 の実装で使用する FPGA ボードは各個人に 1 つ割り当てる．FPGA ボードが不調であった場合は，必ず申し出ること．勝手な判断で FPGA ボードを取り替えないようにすること．

Chapter 2

実験1・実験2で用いるCPUの仕様

本テキストでは，参考用 CPU である Tiny-Processor を例題として CPU の設計について解説する．実験 C では，Tiny-Processor の設計方法を参考にしながら，課題用 CPU である C-Processor を設計する．本章では，Tiny-Processor および C-Processor の仕様を述べる．

2.1 対象とする計算機システム

実験 C で設計する簡易計算機システムの全体図を図 2.1 に示す．実際に設計するのはプロセッサ (CPU) の内部であるが，システム全体についてもおおまかに把握しておいて欲しい．特に CPU と ROM/RAM をつなぐ信号については，今後の設計やシミュレーションでもよく登場する．

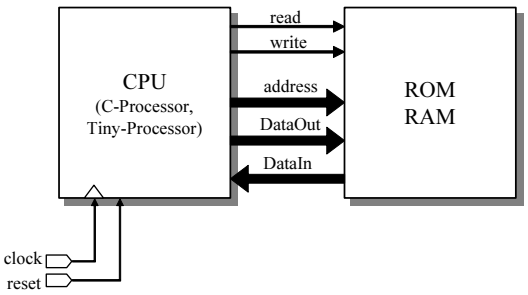


Figure 2.1: 設計する計算機システム

この計算機システムは以下のような手順で動作する．

1. reset 信号によりシステムリセットがかかり，命令ポインタ (IP: Instruction Pointer) が 0x0000 に初期化される．

2. プログラムが ROM に格納されているので，IP の現在値を Address 信号に乗せてメモリに伝え，ROM の該当アドレスのデータを読み出す．これを命令の**フェッチ (fetch)** という．

3. 読み出した命令は CPU 内で解釈され，命令に応じた処理 (演算・データアクセス・分岐等) が行われる

4. IP の値を新たに設定し，2. に戻って以降の命令も同じように処理する

特に変わったことは無い一般的なストアドプログラム方式の CPU の動作と思ってよい．実験 1・実験 2 では，ROM と RAM については区別せず同じアドレス空間で扱うことにしている．

ROM 上にロードしておくプログラムは機械語プログラムであり，0 と 1 の羅列である (実際には 16 進数で書く)．普段は C 言語等の高級言語でプログラムすることが多いだろうが，高級言語のコンパイラは最終的に機械語プログラムを吐き出し，それを実行時にメモリにロードしている．今回は，コンパイラなしで直接機械語プログラムを書くことになる．計算機がプログラムによってどのように動いているのか理解する機会だと思って取り組んで欲しい．また，逆にコンパイラがとても便利なツールだということを実感できるだろう．

2.2 参考用 CPU Tiny-Processor の仕様

まず，参考用として配布する Tiny-Processor の仕様について解説する．基本的な仕様は以下のとおりである．

- 1 word を 8 bit とする．
- アドレス空間は 2 word で指定される (すなわち 0x0000 から 0xFFFF [1]まで 65536 アドレス)．アドレス空間の前半 (0x0000 から 0x7FFF) を ROM 領域，後半 (0x8000 から 0xFFFF) を RAM 領域とする．
- 1 命令 (オペコード) は 1 word で表され，命令によっては引数 (オペランド) を複数持つ．
- レジスタでは，2 の補数表現による整数値を扱う (つまり [-128，127])．
- 動作開始直後，IP は制御部により 0x0000 の値にリセットされる．

Tiny-Processor の機能を解説するために，以下のレジスタを用いる．これらのレジスタは実装でも用いられているが，これらで全てというわけではなく，実装には他にも中間処理用のレジスタ等が存在する．

Table 2.1: Tiny-Processor のレジスタ

名前	用途	データサイズ	説明
A	アキュムレータ	8 bit	演算結果やデータを一時的に記憶する
B	アキュムレータ	8 bit	演算結果やデータを一時的に記憶する
Z	ゼロフラグ	1 bit	演算結果が 0 の場合にセットされる
IX	インデックスポインタ	16 bit	メモリアクセスの際のアドレス指定に用いる
IP	インストラクションポインタ	16 bit	次に実行すべき命令が格納されているメインメモリ上のアドレスを指す
IR	インストラクションレジスタ	16 bit	現在実行中の命令を格納する

表 2.2 が Tiny-Processor の命令セットである．課題で作成する C-Processor の命令セットと混同しないように注意すること．本命令セットでは基本的な算術演算とデータ転送をサポートしている．演算は演算ユニット (ALU) で実装されており，加算，インクリメント，デクリメントを行うことができる．メモリにアクセスする場合は，まず，SETIXH と SETIXL でアドレスの値を IX にセットした後， LDDA，LDDB，STDA 命令で IX の指す箇所に対する読み書きを行う．なお，表 2.2 中，‘memh(l)’, ‘id’ はオペランドとして記述した 1 word の定数を表す．‘memh(l)’ ではアドレス (の一部) を，‘id’ ではレジスタにロードする値を指定している．[IX] は「アドレス IX の (指す先の) 中身」を意味する．IX 自身と混同しないよう注意．また，(Z) はゼロフラグの操作を示す．

Table 2.2: Tiny-Processor の命令セット

命令	オペランド	コード									動作
アドレス指定系											
SETIXH	id	0	0	1	0	0	0	1	0	22	IP ← IP+2, IXH ← id
SETIXL	id	0	0	1	0	0	0	0	1	21	IP ← IP+2, IXL ← id
ロード系											
LDIA	id	0	0	1	0	1	0	0	0	28	IP ← IP+2, A ← id
LDIB	id	0	0	1	0	0	1	0	0	24	IP ← IP+2, B ← id
LDDA		0	0	0	1	1	0	0	0	18	IP ← IP+1, A ← [IX]
LDDB		0	0	0	1	0	1	0	0	14	IP ← IP+1, B ← [IX]
ストア系											
STDA		0	1	0	0	0	0	0	0	40	IP ← IP+1, [IX] ← A
演算系											
ADDA		1	1	0	0	1	0	0	1	C9	IP ← IP+1, A ← A + B (Z)
ADDB		1	1	0	0	0	1	0	1	C5	IP ← IP+1, B ← A + B (Z)
INCA		1	1	0	0	1	0	1	0	CA	IP ← IP+1, A ← A + 1 (Z)
DECA		1	1	0	0	1	0	1	1	CB	IP ← IP+1, A ← A - 1 (Z)
実行制御系											
JP	memh, meml	1	0	0	0	0	0	0	0	80	IP ← memh, meml
JPZ	memh, meml	1	0	0	1	0	0	0	0	90	if (Z = 1) IP ← memh, meml else IP ← IP+3

[1] “0xFFFF” という記述の先頭の “0x” は「16 進数で」という意味である．

2.3 課題用 CPU C-Processor の仕様

次に，本実験で設計する C-Processor の仕様について解説する．C-Processor はユーザから見れば，命令セットのみが Tiny-Processor と異なり，データサイズ，数値表現，インストラクションポインタの初期状態，メモリマップは Tiny-Processor と同じである．それらは Tiny-Processor の仕様の項を参考にすること．レジスタについては，表 2.1 のレジスタに加え，キャリーフラグレジスタ C (1 bit) を追加する．

C-Processor の命令セットを表 2.3 に示す．Tiny-Processor の命令セットに比べると，命令が増えている．なお，Tiny-Processor と共通の命令でもオペコードが異なることに注意．つまり，Tiny-Processor のプログラムは，そのまま C-Processor で動かすことはできない．機械語プログラムを書く際は，混同しないように十分に注意すること．なお，表 2.3 中，'memh(l)', 'id' はオペランドとして記述した 1 word の定数を表す．'memh(l)' ではアドレス (の一部) を，'id' ではレジスタにロードする値を指定している．[IX] は「アドレス IX の (指す先の) 中身」を意味する．IX 自身と混同しないよう注意．また，(Z) はゼロフラグの操作，(C, Z) はキャリーフラグとゼロフラグの操作を示す．

Table 2.3: C-Processor の命令セット

命令	オペランド	サイズ	コード									動作
アドレス指定系												
SETIXH	mem	1 word	1	1	0	1	0	0	0	0	d0	IP ← IP+2, IXH ← mem
SETIXL	mem	1 word	1	1	0	1	0	0	0	1	d1	IP ← IP+2, IXL ← mem
ロード系												
LDIA	id	1 word	1	1	0	1	1	0	0	0	d8	IP ← IP+2, A ← id
LDIB	id	1 word	1	1	0	1	1	0	0	1	d9	IP ← IP+2, B ← id
LDDA			1	1	1	0	0	0	0	0	e0	IP ← IP+1, A ← [IX]
LDDB			1	1	1	0	0	0	0	1	e1	IP ← IP+1, B ← [IX]
ストア系												
STDA			1	1	1	1	0	0	0	0	f0	IP ← IP+1, [IX] ← A
STDB			1	1	1	1	0	1	0	0	f4	IP ← IP+1, [IX] ← B
STDI	id	1 word	1	1	1	1	1	0	0	0	f8	IP ← IP+2, [IX] ← id
演算系												
ADDA			1	0	0	0	0	0	0	0	80	IP ← IP+1, A ← A + B (C, Z)
SUBA			1	0	0	0	0	0	0	1	81	IP ← IP+1, A ← A - B (C, Z)
ANDA			1	0	0	0	0	0	1	0	82	IP ← IP+1, A ← A ∧ B (Z)
ORA			1	0	0	0	0	0	1	1	83	IP ← IP+1, A ← A ∨ B (Z)
NOTA			1	0	0	0	0	1	0	0	84	IP ← IP+1, A ← ¬ A (Z)
INCA			1	0	0	0	0	1	0	1	85	IP ← IP+1, A ← A + 1 (C, Z)
DECA			1	0	0	0	0	1	1	0	86	IP ← IP+1, A ← A - 1 (C, Z)
ADDB			1	0	0	1	0	0	0	0	90	IP ← IP+1, B ← A + B (C, Z)
SUBB			1	0	0	1	0	0	0	1	91	IP ← IP+1, B ← A - B (C, Z)
ANDB			1	0	0	1	0	0	1	0	92	IP ← IP+1, B ← A ∧ B (Z)
ORB			1	0	0	1	0	0	1	1	93	IP ← IP+1, B ← A ∨ B (Z)
NOTB			1	0	0	1	1	0	0	0	98	IP ← IP+1, B ← ¬ B (Z)
INCB			1	0	0	1	1	0	0	1	99	IP ← IP+1, B ← B + 1 (C, Z)
DECB			1	0	0	1	1	0	1	0	9a	IP ← IP+1, B ← B - 1 (C, Z)
CMP			1	0	1	0	0	0	0	1	a1	IP ← IP+1, Z = 1, if A = B (Z)
実行制御系												
NOP			0	0	0	0	0	0	0	0	00	IP ← IP+1
JP	mem	2 word	0	1	1	0	0	0	0	0	60	IP ← mem
JPC	mem	2 word	0	1	0	0	0	0	0	0	40	if (C = 1) IP ← mem else IP ← IP+3
JPZ	mem	2 word	0	1	0	1	0	0	0	0	50	if (Z = 1) IP ← mem else IP ← IP+3

Chapter 3

回路シミュレーションの基礎

「設計中のプロセッサが正しく出来ているか？」という検証作業やデバッグ作業は，一般に回路シミュレータを用いて信号の波形を観測しながら行う．実験 C では，回路シミュレータとして Mentor Graphics 社製の ModelSim を使用する．ModelSim は設計現場でも広く使われている標準的な CAD ツールである．本章では，具体的なプロセッサ設計の解説に入る前に，ModelSim の使い方と ModelSim を使った回路の検証 (テスト) 方法，ならびにデバッグ方法について説明する．実験 C 初回で ModelSim を用いたシミュレーション技法の基礎を習得してほしい．

本章で使用するファイル

本章では下記のファイルを用いる．事前にホームページからダウンロードしておくこと．なお，ファイル名のうち `201xxxxx` は最終更新日の日付が入る．ホームページにあるファイルは適宜更新されることがあるため，最新版を使うこと．

- `Lab1-Gate-VHDL-201xxxxx.zip`
  Gate の VHDL 設計ファイル
- `Lab1-TP-VHDL-201xxxxx.zip`
  Tiny-Processor の VHDL 設計ファイル

3.1 回路シミュレーションとテストベンチ

具体的なシミュレーション方法の解説に入る前に，テストベンチについて説明する．クロック生成回路や発振回路などを除き，一般的には回路は単体では動作しない．回路の外部からクロックや入力信号などを適切なタイミングで与えることで，回路は決められたタイミングで動作し，期待される値を出力する．実際の回路の検証と同様に，設計した回路 (対象回路と言う) を回路シミュレータを用いて検証する場合も，対象回路の外側から適切なタイミングで適切な入力を与えるような (自立して動作する) ダミーの回路が用意となる．通常このようなダミーの回路は「テストベンチ」と呼ばれる．テストベンチは，対象回路を 1 つの部品として使用し，対象回路の全出力を入力，全入力に対して出力を行うような回路となる (図 3.1 参照)．実験 C においても，テストベンチを回路シミュレータ内で動作させることで，対象回路の検証を行う．

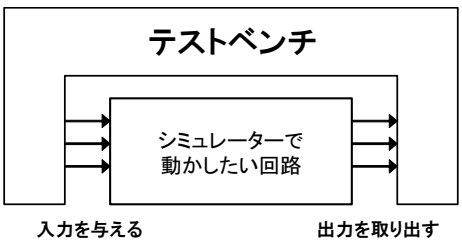


Figure 3.1: テストベンチの概念

3.2 ModelSim による回路シミュレーション

実験 C では，MentorGraphics 社製の ModelSim を回路シミュレータとして使用する．本節では，ModelSim の使い方を，論理演算を行う回路 `Gate.vhd` とそのテストベンチ `TestGate.vhd` を用いて説明する．下記の記述に従い，実際に操作を進めてほしい．なお，記述中，Menu → ··· という表記は，ウインドウ上部のメニューバーから該当する項目を選択する操作を表している．

準備

まず，ホームページより，Gate の設計が含まれているアーカイブ `Lab1-Gate-VHDL-201xxxxx.zip` をダウンロードし展開せよ．アーカイブに含まれるファイルのうち，`Gate.vhd` は，入力 A，入力 B に対して，AND，OR，NAND，NOR，XOR をとったものを出力する回路である．`TestGate.vhd` は，対象回路 Gate.vhd の入力 A，B に対して，100 ns 毎に 00 → 01 → 10 → 11 → 00 を与え，停止するテストベンチである．動作の詳細については，それぞれのファイルの記述を参照すること．

ModelSim の起動

デスクトップにある ModelSim のショートカットをダブルクリックし，ModelSim を起動する．ModelSim が起動すると，important Information ウィンドウが開くことがある．このウィンドウから ModelSim の新機能等を見ることができるが，本実験では関係ないので，Close ボタンで閉じる．このウィンドウを起動するたびに表示させたくないときは，Don't show this dialog again をチェックして閉じる．

ディレクトリの指定

次に，VHDL ファイルのあるディレクトリを指定する．ここで指定したディレクトリが作業ディレクトリとなる．Menu → File → Change Directory で Choose folder ウィンドウを開き，VHDL ファイルがあるディレクトリを選択する．ここで，パス名に日本語名を含むような場合 (例えば「デスクトップ」) はエラーが発生するので注意．**忘れがちな作業であるが，これを行わないとファイルが認識されず回路シミュレーションがうまくいかない**．

ライブラリの作成

Menu → File → New → Library で Create a New Library ウィンドウを開き，ライブラリ名を入力する．オプションは変更しなくてよい．Library Name と Library Physical name と 2 箇所入力フィールドがあるが，どちらも同じ名前にしておいた方が後で混乱しないだろう．ここではライブラリ名をデフォルトのまま work と入力すること．成功すると，Choose a Directory で指定したディレクトリの下に，ここで指定したライブラリ名のディレクトリが作成される．

回路の設計

新規に VHDL ファイルを設計を作成する場合は，Menu → File → New → Source → VHDL でファイルを開き，回路を VHDL 言語で記述していく．ファイルを保存するときは，Menu → File → Save で保存する．既に設計したファイル，あるいは設計途中のファイルを開くときは，メインウィンドウの Menu → File → Open で開きたいファイルを指定する．ここでは `Gate.vhd` と `TestGate.vhd` を開いてみよう．

回路のコンパイル

回路シミュレーションを行うには，設計した回路をコンパイルする必要がある．Menu → Compile → Compile で，Compile Source Files ウィンドウを開き，回路シミュレーションするファイルを全て選択してコンパイルする．複数のファイルを選択する場合には，Ctrl+クリックで追加選択が可能である．この際にコンパイルエラーが発生した場合は，回路記述に誤りがあることがほとんどであるので，設計内容に誤りが無いかを確認すること．ここでは，図 3.2 のように，`Gate.vhd` と `TestGate.vhd` の両方を選択して Compile ボタンを押してコンパイルする．

シミュレーション回路の選択

Menu → Simulate → Start Simulation で Start Simulation ウィンドウ (図 3.3) を開き，Design タグから Create a New Library で指定したライブラリのポイントを展開し，テストベンチを選択の後，OK ボタンを押す．この際，**Enable optimization チェックボックスのチェックを外すこと**．ここでは，work ライブラリにある testgate を選択して，OK ボタンを押す．

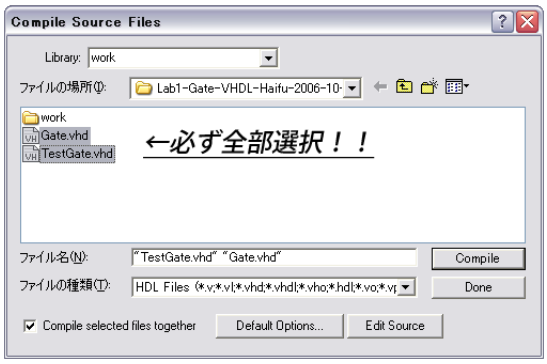


Figure 3.2: Compile Source Files

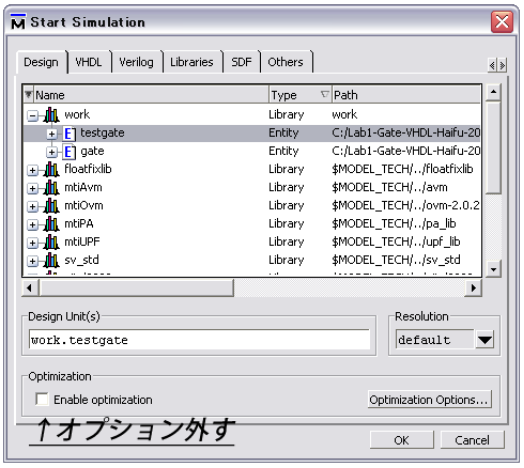


Figure 3.3: Start Simulation

波形観測ピンの登録

図 3.4 に Start Simulation 後のシミュレーション画面の一例を示す．シミュレーション画面は，下記の 4 つのビューウィンドウで構成される．各ビューウィンドウは Menu → View から Structure，Memory，Objects，Wave，Transcript 等，所望のウインドウを選択することで，表示/非表示を切替え可能である．

- 最も左のビューウィンドウは下部のタブによって複数のウィンドウを選択するマルチビューウィンドウとなる．初期状態では図のように Structure が表示されている．タブのうち Memory を選択するとメモリの状態を表示できる．
- 中間のビューウィンドウは Objects ビューアである．左の Structure でインスタンス (コンポーネント) を選択すると，そのインスタンスの持つ信号が，この Objects ビューアにリストアップされる．
- 右のビューウィンドウは Wave ビューアである．Objects ビューアにリストアップされた信号を選択し，ドラッグ＆ドロップすることで，Wave ビューアに登録され波形が表示される．プロセッサなどの大きな回路を設計した場合は，使用しているコンポーネントの内部の信号線の変化を波形観測しなければならないことがある．このような場合は，Structure ビューアのコンポーネントツリーを展開し，中で呼び出しているコンポーネントを選択する．その後は，同様に Objects ビューアから Wave ビューアに信号をドラッグ＆ドロップする．
- 下のビューウィンドウは Transcript ウィンドウである．メッセージログが表示される．熟練者はコンソールとして使うこともするが，本実験ではメッセージログを参照するのみに用いる．
- 上部のアイコンの並んでいる部分に，Restart，Run，Run-all ショートカットボタン，時間設定のできるボックス等がある．これらを使うと素早くシミュレーションを進められて便利である．

今回は，Structure ビューアで testgate を選択すると，Objects に 7 つの信号が表示されるので，これを全て選択して Wave にドラッグ＆ドロップする．

シミュレーションの実行

シミュレーション画面上部　(図 3.4 上部の囲み)　から波形シミュレーションを行う時間 (run length) を設定する．run length のデフォルト値は 100 ns である．ここでは，500 ns あれば十分なので，run length を 500 ns に変更する．次に，Run アイコンをクリックして，シミュレーションを開始する．run length で指定した時間分のシミュレーションが行われ，登録したピンの波形が Wave ビューアに表示される．もう一度最初からシミュレーションしたい場合は，Menu → Simulate → Run → Restart，または Restart アイコンをクリックしてリセットしてから，再度 Run アイコンを押す．

波形の観察

波形の表示を整えるためには，Wave ビューアを選択した状態で，Menu → Wave → Zoom 中の機能を使うか，もしくは虫眼鏡のアイコンを使う．信号名の横には，ある時刻 (クリックすると出現する黄色い縦線の時刻) での，その信号の値が表示される．信号名が表示されている部分で右クリックして出現するポップアップメニュー

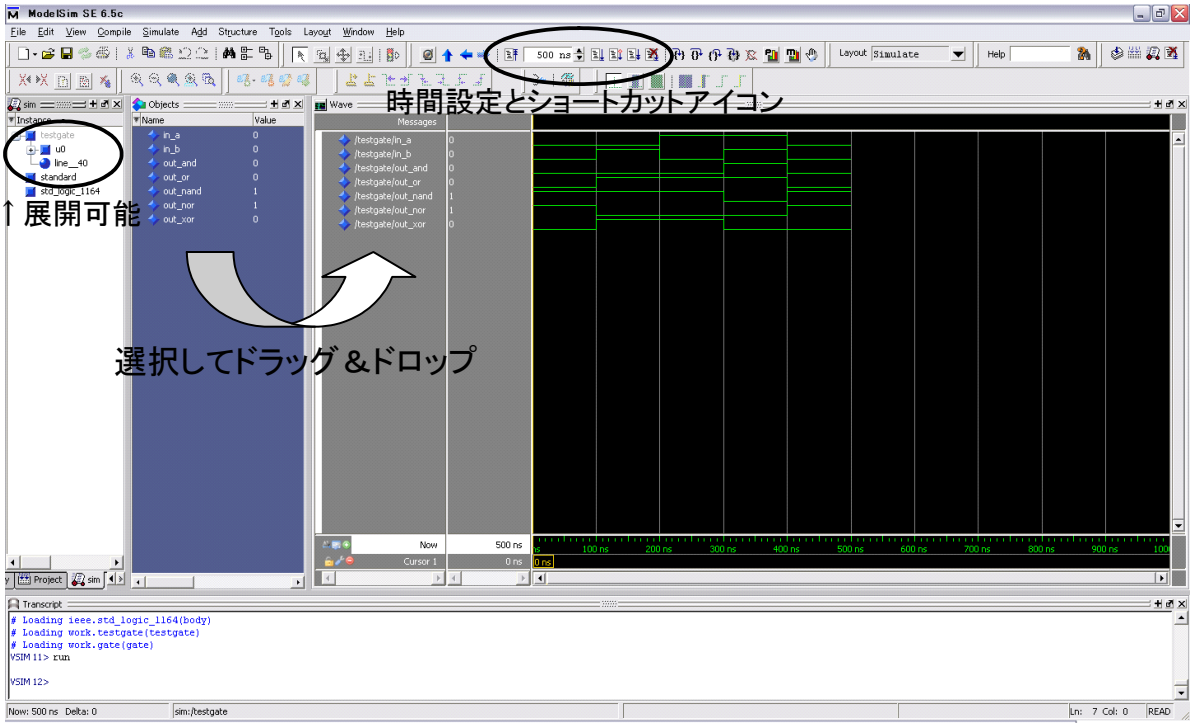


Figure 3.4: シミュレーション画面

から Radix を選択し，2 進，8 進，10 進等の信号の表現法を変更できる．Wave ビューアで波形をスクロールさせながら，それぞれの信号値が所望のタイミングで変化しているかを確認する．正しく動作していないときは，元の VHDL ファイルの記述を変更して，再度シミュレーションを行う．

シミュレーションの終了

大抵の場合は，シミュレーション画面のまま Wave の位置にソースファイルを開いて編集，コンパイルすることも可能だが，支障が生じる場合は一度シミュレーションを終了させる．シミュレーションの終了は，Menu → Simulate → End Simulation を選択する．これで，ウィンドウ構成が Start Simulation を行う前の段階に戻る．この場合，Wave に登録していた信号リストは初期化されるので注意．

3.3 ModelSim による CPU のシミュレーション

次に，CPU のシミュレーション方法について述べる．まず，ホームページより，Tiny-Processor とメモリ内容が含まれているアーカイブ `Lab1-TP-VHDL-201xxxxx.zip` をダウンロードし，展開せよ．本アーカイブは，Tiny-Processor を構成するファイル (本体 `TinyProcessor.vhd`，データパス部 `DataPath.vhd`，制御部 `Control.vhd`，基礎部品部 `Resource.vhd`) および Tiny-Processor のテストベンチ `TestProcessor.vhd`，メモリ `memory.txt`，およびその他のファイル `TestAlu.vhd`，`TestMux.vhd` からなる[1]．

回路シミュレーション

3.3 節と同様に，Tiny-Processor のシミュレーションを行ってみよう．Menu → File → Change Directory による「ディレクトリの指定」を忘れないようにすること．回路のコンパイルには，`TinyProcessor.vhd`，`DataPath.vhd`，`Control.vhd`，`Resource.vhd`，`TestProcessor.vhd` を指定する (選択がめんどうであれば全ての VHDL ファイルをコンパイルしてもかまわない)．シミュレーション回路の選択時には，work ライブラリにある testprocessor を選択する．なお，この時，Enable optimization チェックボックスを外すことに注意．

[1]CPU のシミュレーションでは，CPU 本体だけでなくメモリ (ROM/RAM) もテストベンチに含める必要がある．さらにメモリ上におかれる，命令列やデータなども用意する必要がある．メモリに格納されている情報 (命令列やデータ) はテストベンチに直接書いても構わないが，その場合，メモリの内容を変更するたびに (回路構造は変化していないにも関わらず) テストベンチの再コンパイルが必要になる．そこで，配布しているテストベンチでは，メモリに格納する情報は，テストベンチとは独立のファイル (memory.txt) として記述し，テストベンチ動作中にファイルから情報を読み込むようにしている．こうすることで，メモリの内容を書き換えても，テストベンチを再コンパイルする必要がなくなる．Tiny-Processor のテストベンチの動作の詳細については，TestProcessor.vhd の記述を参照すること．

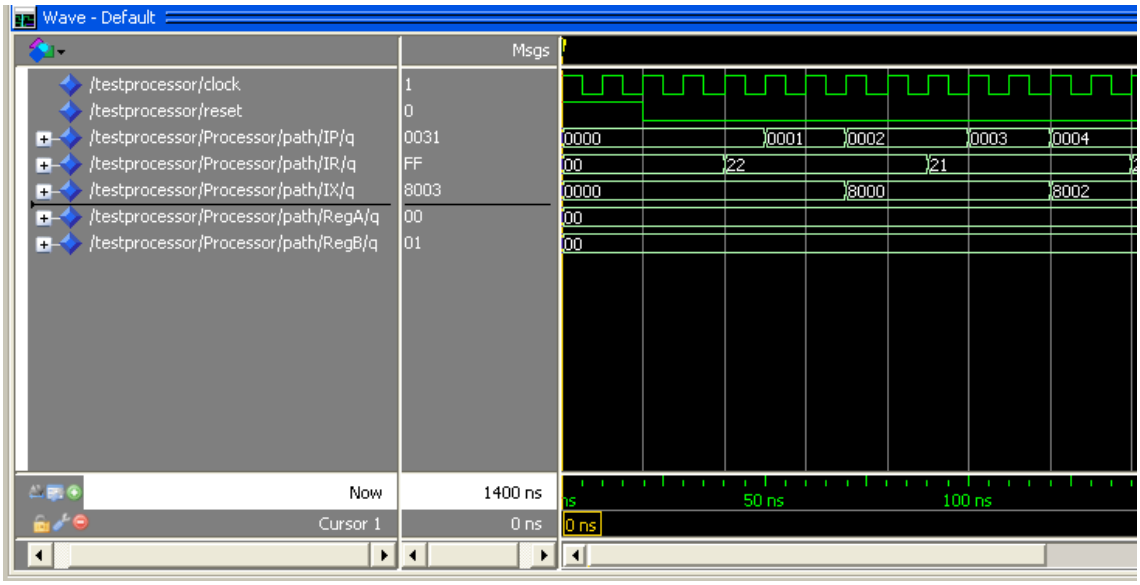


Figure 3.5: Tiny-Processor の波形シミュレーション画面

図 3.5 に波形観測の例を示す．この例では波形観測ピンとして以下の信号を指定している．また，信号を 16 進数で表示している．

- testprocessor/Processor/clock (クロック信号)
- testprocessor/Processor/reset (リセット信号)
- testprocessor/Processor/path/IP/q (IP レジスタの出力)
- testprocessor/Processor/path/IR/q (IR レジスタの出力)
- testprocessor/Processor/path/IX/q (IX レジスタの出力)
- testprocessor/Processor/path/RegA/q (A レジスタの出力)
- testprocessor/Processor/path/RegB/q (B レジスタの出力)

Run アイコンを押すとシミュレーションが順次実行される．ここでは，`memory.txt` に書かれた内容の機械語プログラムが動作しているはずである．表 2.2 に示される Tiny-Processor の命令セットと，レジスタの値の変化を見ながら，所望の動作をしているか確かめてみよう．なお，CPU のメモリ内容は，左のマルチビューウインドウ下部のタブから Memory タブを選択し，ウィンドウに表示されたメモリコンポーネントをダブルクリックすることで表示できる．データ表示はデフォルトでは 2 進数表示であるが，右クリック→ Properties で Data Radix を 10 進や 16 進に変更することで変更可能である．

メモリの記述法

テストベンチのメモリファイル `memory.txt` の記述方法を以下に解説する．`memory.txt` では，最初にアドレスを記述し，空白を空け，次に命令のコードやオペランドを記述する．ハイフン 2 つから後ろはコメントである．**ファイル中に空行が含まれると正しく動作しないので注意すること**．

以下に配布している `memory.txt` の冒頭部分を示す．

```
0000    22  -- SETIXH
0001    80  --
0002    21  -- SETIXL
0003    02  --
0004    28  -- LDIA
0005    44  --
0006    24  -- LDIB
0007    66  --
0008    21  -- SETIXL
0009    01  --
000a    18  -- LDDA
```

命令コードについては，表 2.2 に示される Tiny-Processor の命令セットを参考にすること．このプログラムの動作を命令セットの仕様に沿ってレジスタを用いて表すと以下のようになる．

IXH	←	0x80	(0000–0001 番地)
IXL	←	0x02	(0002–0003 番地)
A	←	0x44	(0004–0005 番地)
B	←	0x66	(0006–0007 番地)
IXL	←	0x01	(0008–0009 番地)
A	←	[IX]	(000A 番地)

Tiny-Processor ではアドレスが 16 bit であるにも関わらず，データバスは 8 bit で設計されている．そのため，IX レジスタへのアドレス設定は，SETIXH と SETIXL の 2 命令に分けて行う．アドレスの微変更に対しては，IX レジスタの下位 8 bit のみを変更することで余分な SETIXH 命令を省くことができる．LD 系命令は即値ロード LDIA，LDIB (レジスタにオペランド値を入れる) とメモリロード LDDA，LDDB の両方をサポートしている．よく混同するので注意すること．

サンプルプログラムと命令セットを眺めながら，この命令セットを実行するためにどのようなアーキテクチャにすればよいかを考えてみるとよい．Tiny-Processor の実装例は，5 章で追って解説する．

Chapter 4

実験1：CPUの設計(Tiny-Processor)

本章では Tiny-Processor の設計手順を述べる．ここに書かれている内容を参考に，C-Processor を設計し，シミュレーションによる動作確認を行うまでが実験 1 である．なお実験を進めるにあたって必要な知識が付録 A と付録 B に書かれているので，実験 1 に入る前にひととおり読んでおくこと．

本章で使用するファイル

実験 1 では下記のファイルを用いる．事前にホームページからダウンロードしておくこと．なお，ファイル名のうち 201xxxxx は最終更新日の日付が入る．ホームページにあるファイルは適宜更新されることがあるため，最新版を使うこと．

- `Lab1-TP-VHDL-20xxxxxx.zip`
  TinyProcessor の VHDL 設計ファイル．3 章で用いたものと同じ．
- `Lab1-Test-VHDL-20xxxxxx.zip`
  C-Processor DataPath 部および Control 部のテストベンチの VHDL 設計ファイル

4.1 CPU 設計の手順 (実験 1 の流れ)

プロセッサ設計は，おおまかに以下のような手順を経る．

1. 命令セットの決定：
   設計するプロセッサの仕様，命令セットを決定する．アプリケーションプログラムを設計対象のプロセッサで効率よく実行するために，どのような命令を実装すればよいかを検討する．例えば，乗算を多用するアプリケーションプログラムであれば，乗算を 1 命令で実行するような乗算命令を実装しておけば，アプリケーションプログラムの実行にかかる時間は減少する．なお，実験 C で対象とする Tiny-Processor および C-Processor の命令セットは予め与えられている．(2 章)

2. リソース (機能部品) の決定：
   仕様で定めた命令の機能 (動作) を，どの部品を使って実現するかを考え，プロセッサで使う部品を決定する．例えば，加算命令を ALU の加算機能を使って実装するのか，それとも，加算専用の加算器を用いて実装するのか等を決定する．(4.2 節)

3. データパス (アーキテクチャ) の決定：
   プロセッサで使用する機能部品の接続関係，および外部 (RAM や ROM，外部入出力装置など) とのインターフェースを定める．インターフェース (入出力) の仕様は，プロセッサの外部で使用するコンポーネントとのデータの送受信等を考慮して決定する．

4. 制御部の設計：
   プロセッサのデータパスを制御する制御部は，以下の手順で設計する．

   (a) 各命令の状態遷移を作成し，各状態での信号の出力を決定する．(4.5 節)
   (b) 外部出力信号の出力値が同じ遷移をするような命令の状態遷移を統合する．(4.7 節)
   (c) ハザード対策をすべき信号線をすべて取りだし，その信号線のみを扱う Moore 型の状態機械 (外部出力信号用オートマトン) を作成する．(4.8 節)

(d) 外部出力信号用オートマトンを，ハザードに注意しながらジョンソンカウンタ群に分解する．(4.10 節)

(e) 外部出力信号用ジョンソンカウンタ群を制御しながら，残りの信号線も制御する内部制御信号用の状態機械 (内部制御信号用オートマトン) を作成する．(4.13 節)

(f) IR レジスタ値などを参照して，内部制御信号用オートマトンを簡単化する．内部制御信号はハザードを考慮しなくてよい．(4.13 節)

(g) 内部制御信号用オートマトンをカウンタ群に分解する．内部制御信号用のカウンタはハザードが発生してもかまわないが，実験 C では外部出力用と同様にジョンソンカウンタを用いることにする．(4.13 節)

(h) 外部出力信号用ジョンソンカウンタと内部制御用ジョンソンカウンタの出力値を参考にしながら，制御信号のデコード論理を作成する．(4.14 節)

5. 動作の確認：
与えられた仕様を満たしていることを確認する．最終的に複数のジョンソンカウンタ群に分解した制御部の HDL 記述を作成し，シミュレータを用いて動作を確認する．

以降では，Tiny-Processor の作成を例として，プロセッサ設計の流れを説明する．Tiny-Processor は，C-Processor のごく一部の命令のみを実装している．したがって，Tiny-Processor の HDL 記述に対して変更を加えて行けば，C-Processor が完成する．

4.2 機能部品の選定と設計

命令セットの仕様が決まったら，命令セット仕様を実現するために，どのような機能部品 (リソース) を使用するかを決定する必要がある．具体的には,「命令の動作をどの機能部品を使って実現するか」を考える．例えば，ADDA の命令 (IP ← IP+1，A ← A+B (Z)) の場合，以下のようなことを検討する．

- “IP ← IP+1” の部分の機能を,
  - レジスタと加算器で実現するのか，レジスタと ALU で実現するのか,
  - 加算機能付きレジスタを使うのか,
- “A ← A+B (Z)” の部分の機能を,
  - 加算器で実現するのか,
  - ALU で実現するのか,
- “IP ← IP+1” と “A ← A+B (Z)” の加算機能は同じ部品を使うか,

また，“A ← A+B (Z)” からは，少なくともレジスタ A とレジスタ B の 2 つのレジスタが必要になることがわかる．ここで注意して欲しいのは，命令の動作には登場しなくても，一時的に値を保持するためのレジスタなどが必要になる場合もある点である．

Tiny-Processor の命令セット仕様を実現するため，本実験では，図 4.1 に示すようなデータパスを用いる．Tiny-Processor では，8 bit レジスタ，1 bit レジスタ，16 bit カウンタ，ALU，および 2 入力から 1 つを選択して出力する 8 bit/16 bit マルチプレクサ (セレクタ) を使用する．機能部品の仕様は以下の通りである．

- 16 bit/8 bit/1 bit レジスタ：16 bit, 8 bit，もしくは 1 bit のデータ入出力，クロック入力，データロード用の 1 bit 幅の信号線 load を持つ (16 bit レジスタは，上位 8 bit のみに値をとりこむ信号線 loadh，下位 8 bit のみに値を取り込む信号線 loadl を持つ)．load のデフォルト値は L であり，値が H の時，クロックの立ち上がりに同期してデータを読み込む.
- 16 bit カウンタ：16 bit の入出力，クロック入力，ロード信号 load，クリア信号 clear，2 種類のインクリメント信号 inc，inc2 のための 1 bit 幅の信号線を持つ．各信号のデフォルト値は L である．load 信号が H のときにクロックの立ち上がりに同期して入力値をカウンタに読み込み，clear が H であるとカウンタはリセットされる．また inc が H のときはカウンタが 1 つインクリメントされ，inc2 が H であればカウンタ値は 2 つインクリメントされる.
- ALU：2 系統のデータ入力 (各 8 bit)，1 系統の計算結果の出力 (8 bit)，キャリー出力，ゼロフラグ (1 bit)，どのような演算を行うかを規定する ALU モード (2 bit) の信号線 modeALU を持つ．modeALU のデフォルト値は LL である．modeALU の値と演算内容の対応を表 4.1 に示す.
- マルチプレクサ：2 系統の入力 (16 bit/8 bit)，1 系統の出力 (16 bit/8 bit)，どの入力を出力するかを選択する 1 bit の信号線を持つ.

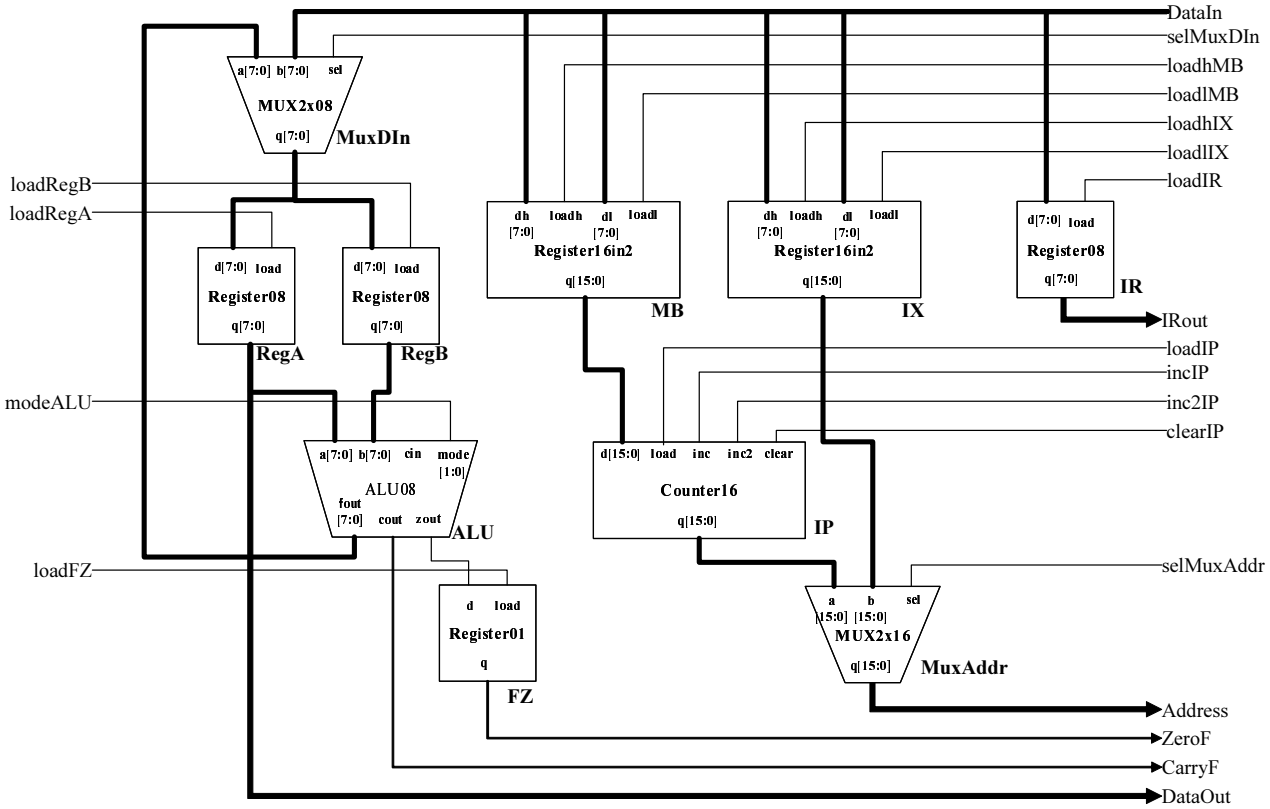


Figure 4.1: Tiny-Processor のデータパス

Table 4.1: Tiny-Processor における ALU の modeALU 信号に対応する機能

modeALU	演算内容
LL	¬ A
LH	A + B
HL	A + 1
HH	A − 1

4.3 データパスの決定

各命令の動作を実現する機能部品が決定したら，次に，それら機能部品の接続を決定する．図 4.1 に示した Tiny-Processor のデータパスに従って記述する．データパスにおけるレジスタ IR は命令を格納する 8 bit レジスタ，レジスタ RegA，RegB は 8 bit のデータレジスタ，レジスタ MB はインストラクションポインタ IP (16 bit カウンタ) にアドレス値を設定するために用いられる 16 bit レジスタ (上位・下位 8 bit を独立に読み込む) である．また，レジスタ IX は RAM 上のアドレスを保持する 16 bit レジスタであり，FZ はゼロフラグとして用いられる 1 bit レジスタである．MuxDIn，MuxAddr はそれぞれ 8 bit，16 bit の 2 入力 1 出力のマルチプレクサであり，sel が L の時に q に a が出力され，sel が H の時に q に b が出力される．

プロセッサのデータパスの構造は，各命令の動作を実現するデータの流れ，つまりどの機能部品の出力値をどの機能部品に入れればよいか，を考えて設計する．ADDA 命令 (ADDA： IP ← IP+1，A ← A+B (Z)) の場合は図 4.2，STDA 命令 (STDA： IP ← IP+1，[IX] ← A) の場合は図 4.3 のようなデータの流れになる．次に，これらの各命令におけるデータの流れを全て重ね合わせる．このとき，同じ機能部品の入力または同じ信号線に 2 つ以上のデータが流れてくるような場合は，その直前にマルチプレクサを挿入して，データが衝突しないようにする．Tiny-Processor では MuxAddr，MuxDIn がデータ衝突回避のために挿入されている．マルチプレクサはデータの衝突回避のために後から挿入するので，最初はマルチプレクサを用意せずに機能部品だけを並べて考え始めるとよい．

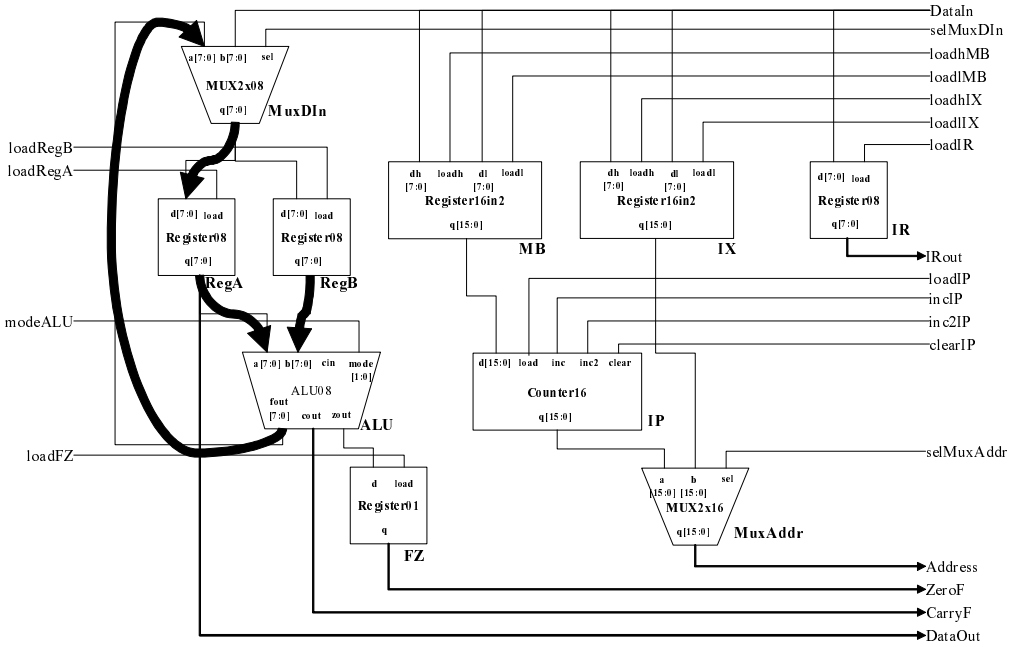


Figure 4.2: Tiny-Processor における ADDA 命令のデータの流れ

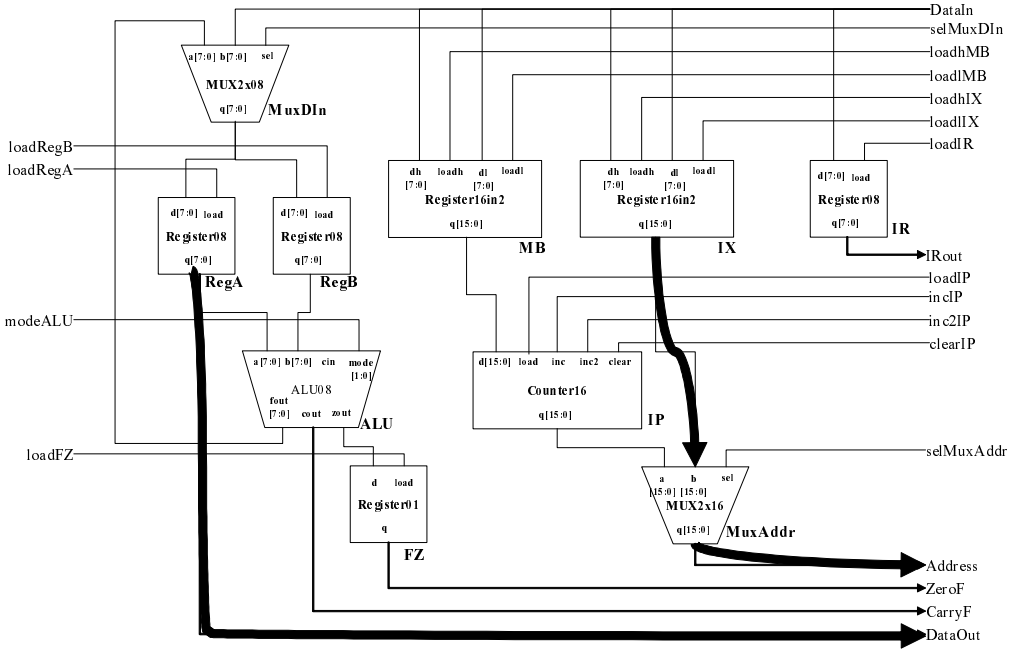


Figure 4.3: Tiny-Processor における STDA 命令のデータの流れ

4.4 メモリの書き込み・読み込み

データパス部が決定したら，データの流れを制御する制御部の設計へと進む．ここで重要になってくるのが，外部にある機能部品とのタイミングである．例えば，外部記憶として高速なメモリを使用する場合と，低速なメモリを使用する場合では，プロセッサから出力するアドレス情報やデータのタイミングがまったく異なってくる．実験 1 では，実験 2 で使用する FPGA ボードにプロセッサを搭載することを前提として進めている．そこで，

制御部の設計へ進む前に，実験 2 で使用する FPGA ボードで使われる部品，特にメモリ関係について説明する．

実験 2 で使用する FPGA ボードで用いられている ROM，RAM のタイミングチャートを図 4.4 に示す．図 4.4 上は，ROM と RAM の読み出しのタイミングチャートである．指定されたアドレスにアドレス値が指定され，/OE (output enable) が enable になると，アドレスに対応したデータが出力される．/OE が enable でないとき出力データ値はハイインピーダンスである．図 4.4 下は RAM の書込みのタイミングチャートである．アドレス値とデータ値が指定されその間に，/WE (write enable) が enable になると，そのデータ値が指定アドレスに書込まれる．

したがって，読み出しのときは，アドレスデータが確定してからそれに対応する有効なデータが出力されるのに十分長い時間 T_1 が必要であることと，データが書き込まれるのに十分長い時間が必要であること (すなわち，アドレス値とデータ値が確定してから，T_2 時間待って，/WE 信号を T_3 時間保持しなければいけない．また，その間ではアドレス値とデータ値が変化してはいけない) ことに注意する．これらの理由により，動作クロック 20 MHz では ROM と RAM の読み出しには 2 クロックを要し，RAM の書き込みは 3 クロックを必要とする．したがって，ROM と RAM の読み出しには，2 クロック間はアドレスと/OE 信号を保持しておき，後半の 1 クロックで出力データをレジスタに取り込むこととなる．一方，RAM の書き込みにおいては 3 クロック間アドレス値とデータ値を保持しておき，真ん中の 1 クロックで/WE 信号を enable にする必要がある．また，その 2 クロック，3 クロックの間は，アドレス，データ，write 信号，read 信号には，ハザードを発生させてはならない．

Tiny-Processor の制御部は，データパス部の制御だけでなく，メモリの読み込み・書き込みの制御 (/WE, /OE) も行わなければならない．また Tiny-Processor のデータパスでは，メモリとのデータの入出力は DataIn と DataOut となっているが，メモリの入出力は共通である．したがって，トライステートバッファを使用することで，メモリからの読み込み時に DataOut をハイインピーダンスにして，データの衝突を避けなければならない (図 4.5).

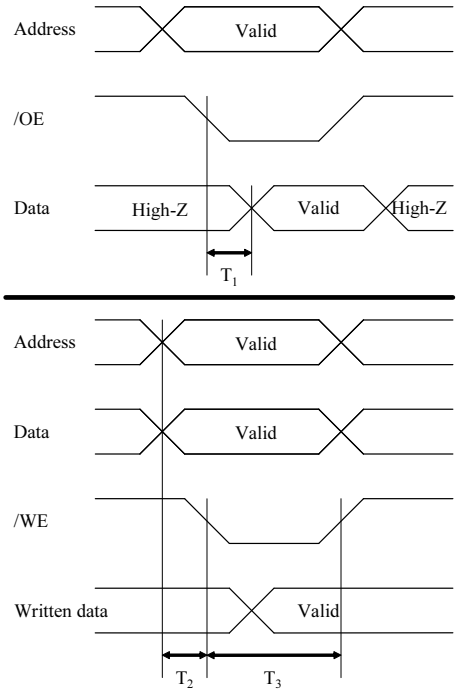


Figure 4.4: メモリアクセスのタイミング

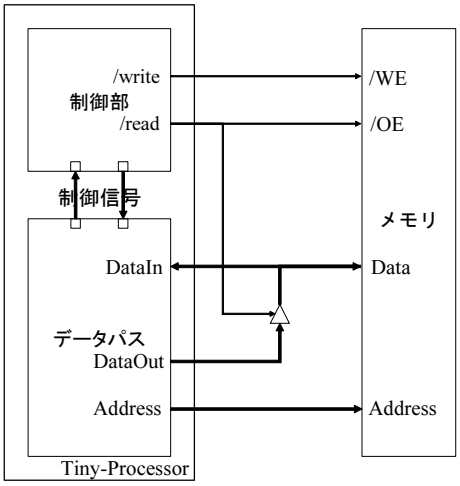


Figure 4.5: Tiny-Processor とメモリの接続関係

4.5 制御部の設計

制御部とデータパス部の関係の全体イメージを図 4.6 に示す．制御部は順序回路であり，回路の状態を記憶して内部／外部信号を決定する．内部信号とはデータパスに含まれるレジスタのロード信号や，マルチプレクサの選択信号のことで，これらを決定することでデータパスの特定部分にデータが流すことができる．外部信号はメモリのアクセスに使用する信号で，後述するがハザード対策が必要な信号のことである．

制御信号のデフォルト値の定義

制御部が出力する信号線を列挙し，そのデフォルト値を定義する．以下の表 4.2 において，信号名は/read，/write を除いて，“各機能部品における信号名+データパス上の部品名” という規則になっている．ここでハザードに

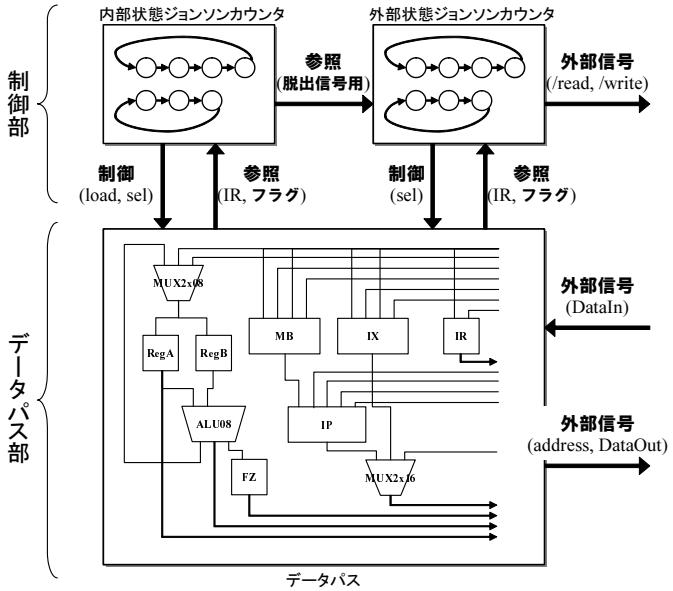


Figure 4.6: 制御部とデータパス部の関係 (CPU の構造)

Table 4.2: Tiny-Processor における制御信号のデフォルト値とハザード

信号名	デフォルト値	ハザード対策
selMuxDIn	L	
loadhMB	L	
loadlMB	L	
loadhIX	L	
loadlIX	L	
loadIR	L	
loadIP	L	
incIP	L	
inc2IP	L	
clearIP	L	
selMuxAddress	L	要
loadRegA	L	
loadRegB	L	
modeALU	LL	
cinALU	L	
loadFZ	L	
/read	H	要
/write	H	要

気をつけるべき信号線は，外部素子であるメモリの読み書きに関わる信号，selMuxAdrress, /read，/write である．なお，/read, /write など信号が負論理 (デフォルト値が H で L がアクティブ．Active Low とも呼ぶ) である場合，一般的にそれを明示するために，信号名の前にスラッシュをつけて表記することがある．

4.6 命令毎のデータパスの制御

LDDA 命令の実行に関するデータパスの制御について考える．LDDA 命令は，メモリの IX 番地に格納されているデータを読み出して，RegA にデータを格納する命令である．まず LDDA 命令を完了するまでに必要な一連の動作を，1 クロックで実現可能な動作にまで分解する．レジスタやメモリ以外の素子では，クロックを超えて値を保持できないので，基本的には，[レジスタ／メモリ] から [レジスタ／メモリ] までの動作に分解して考える．この場合以下の 3 つの動作に分けられる．

- レジスタ IX の値を Address バスに出力する

- データをメモリから読み込み DataIn に流し，MulDIn を通って RegA の入力 d まで流す.
- 入力 d に流れてきたデータをレジスタ RegA に取り込む.

次に考えるべきことは,「これらの動作を実現するために必要な制御信号と変化のタイミング」である．まず,1 クロック単位に分解された動作を実現するのに必要な制御信号を抜き出し，その制御信号をどのように変化させるかを決定する．この場合は以下のようになる.

- レジスタ IX を Address バスに出力するために selMuxAddr を H にする
- メモリからデータを読み込むために/read を L にして，RegA の入力まで値を届けるために selMuxDIn を H にする.
- レジスタ RegA に読み込むために，loadRegA を H にする

Tiny-Processor では selMuxAddr のデフォルト値を L としているので，selMuxAddr を H に変化させる必要がある．もし selMuxAddr のデフォルト値が H だった場合は，最初の状態では信号を変化させる必要がないので，何もしなくて良いことになる．信号のデフォルト値を H か L にするかによって，信号の制御の複雑さが大きく変わってくるので，信号のデフォルト値をどちらにするか注意して決める必要がある．基本的には，命令セット全体を見渡してみて，その信号が H か L のどちらで使われることが多そうであるかを考えるとよい.

4.7　命令毎のデータパスの制御のタイミング

次に，これらの制御信号を制御するタイミングについて考える．前述したように,「メモリからの読み込みは 2 サイクル必要なこと」や「ハードウェアではデータがぶつからない限り並列に動作させられる」を考えながら,上の動作をどのタイミングですればよいかを考える．この場合は,

1. selMuxAddr = H，/read = L
2. selMuxAddr = H，/read = L，loadRegA = H

となる．読み出し速度の早いメモリを使用している場合は，loadRegA = H は第 1 サイクルでも構わないが，実験 2 で使用する FPGA ボードに搭載されているメモリは，読み込みに 2 サイクルかかる．したがって，第 1 サイクル目で selMuxAddr = H として IX に格納されているメモリアドレスをアドレスバスにセットし，/read = L で読み出しを開始する．そして，第 2 サイクル目でメモリから出力が安定するので，DataIn に流れてきたデータを loadRegA = H で RegA に取り込む.

次に，各サイクルを 1 つの状態として表すことで，その命令における信号の状態遷移が得られる．Tiny-Processoer の全命令の信号状態遷移を書くと図 4.7 のようになる．フェッチ (命令の読み込み) は全ての命令に共通なので分離して書いてある．各状態の下には，その状態においてデフォルト値から変化させなければならない信号線を書いている．ここで，Fetch に 3 状態とっていることに注意して欲しい．IR に load するために前述のように 2 状態 (2 クロック分) で可能である．しかし IR の値 (つまり読み込んだ命令) によって状態の遷移先を決定しなければならないので，IR の値による分岐は 1 状態遷移した後でしか行えない．また，JPZ 命令は分岐が成立する場合と成立しない場合で，2 つの状態遷移がありうる.

図 4.7 を観察すると，ADDA，ADDB，INCA，DECA 命令の信号状態遷移は，ALU へ与える信号 (modeALU) だけが異なっていることがわかる．そこで，これらの状態を統合し，命令が読み込まれている IR レジスタの値によって，modeALU にどの値を出力するかを決定するようにすれば状態数が減らせる．簡単にできる範囲で統合した Tiny-Processor の状態遷移を図 4.8 に示す．状態の下にある四角には，統合した状態で共通してデフォルトから変化させなければならない信号と個別にデフォルトから変化させなければならない信号を書いている．どの状態とどの状態とを統合するかの一つの基準は，ハザード対策が必要な外部出力信号の制御が同じように遷移していく命令を統合するとよい．図 4.8 では SETIXH, SETIXL, LDIA, LDIB 命令を統合していないが，ハザード対策が必要な信号の制御は同じであるので，これらを統合すればさらに状態数を減らすことも可能である.

4.8　ハザード対策が必要な信号用の制御

次に selMuxAddr，/read, /write のようにハザードに気をつけるべき信号線を専門に扱うジョンソンカウンタを設計する．以降，これらのジョンソンカウンタを「外部出力信号用ジョンソンカウンタ」と呼ぶ.

外部出力信号用ジョンソンカウンタの制御を考えるために，信号状態遷移 (図 4.8) から，ハザード対策が必要な外部出力信号線の値の組み合わせに着目して，外部出力信号用の新たな状態を定義する (図 4.9)．外部出力信号の値の組み合わせが同じであれば，同じ外部出力信号用の状態とする．ただし，状態 stda0 と stda2 の外部出力の組み合わせは同じであるが，ハザード対策を容易にするため，別状態 s3，s5 としている．図 4.9 では状態の下にある四角には，その状態でデフォルトから変化させなければならない外部出力信号線を抜き出しており,その上に外部出力信号線の値の組み合わせと値の組合せに対して割り当てた状態名を書いている.

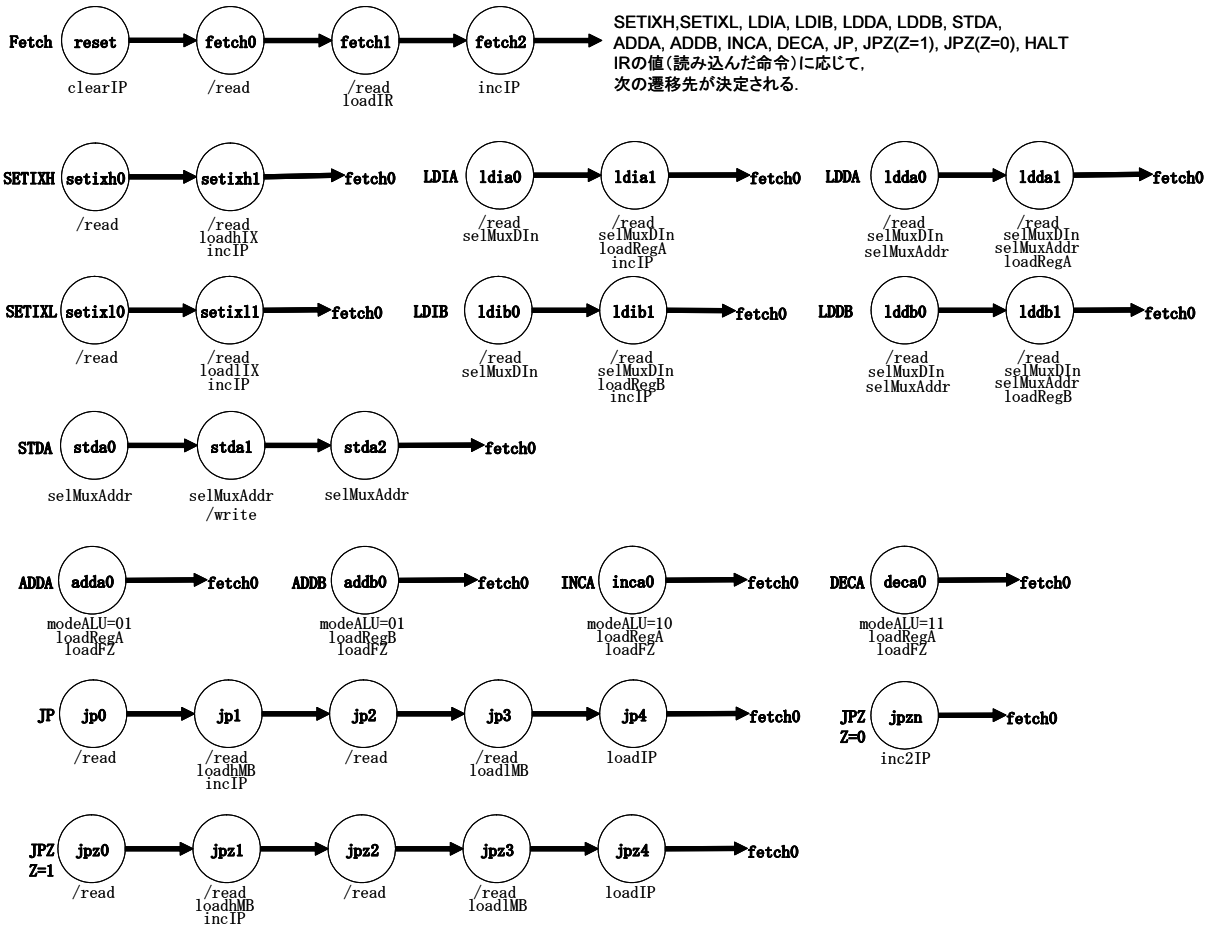


Figure 4.7: Tiny-Processor の全命令の信号状態遷移

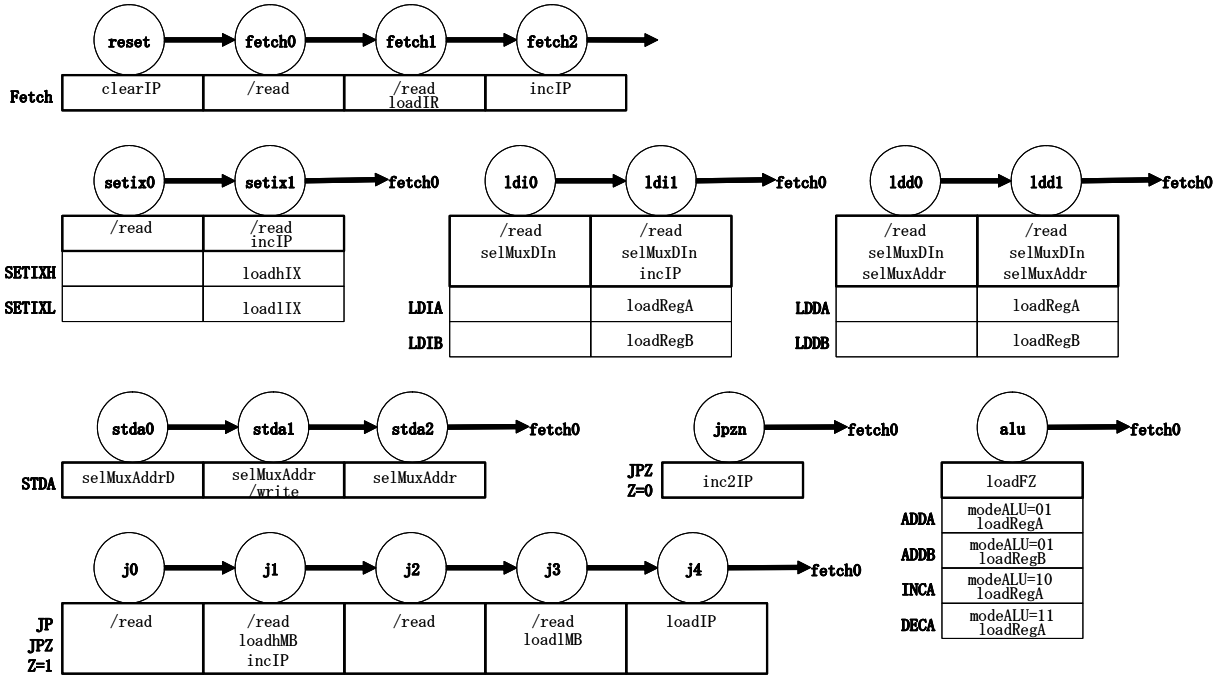


Figure 4.8: Tiny-Processor の統合された信号状態遷移

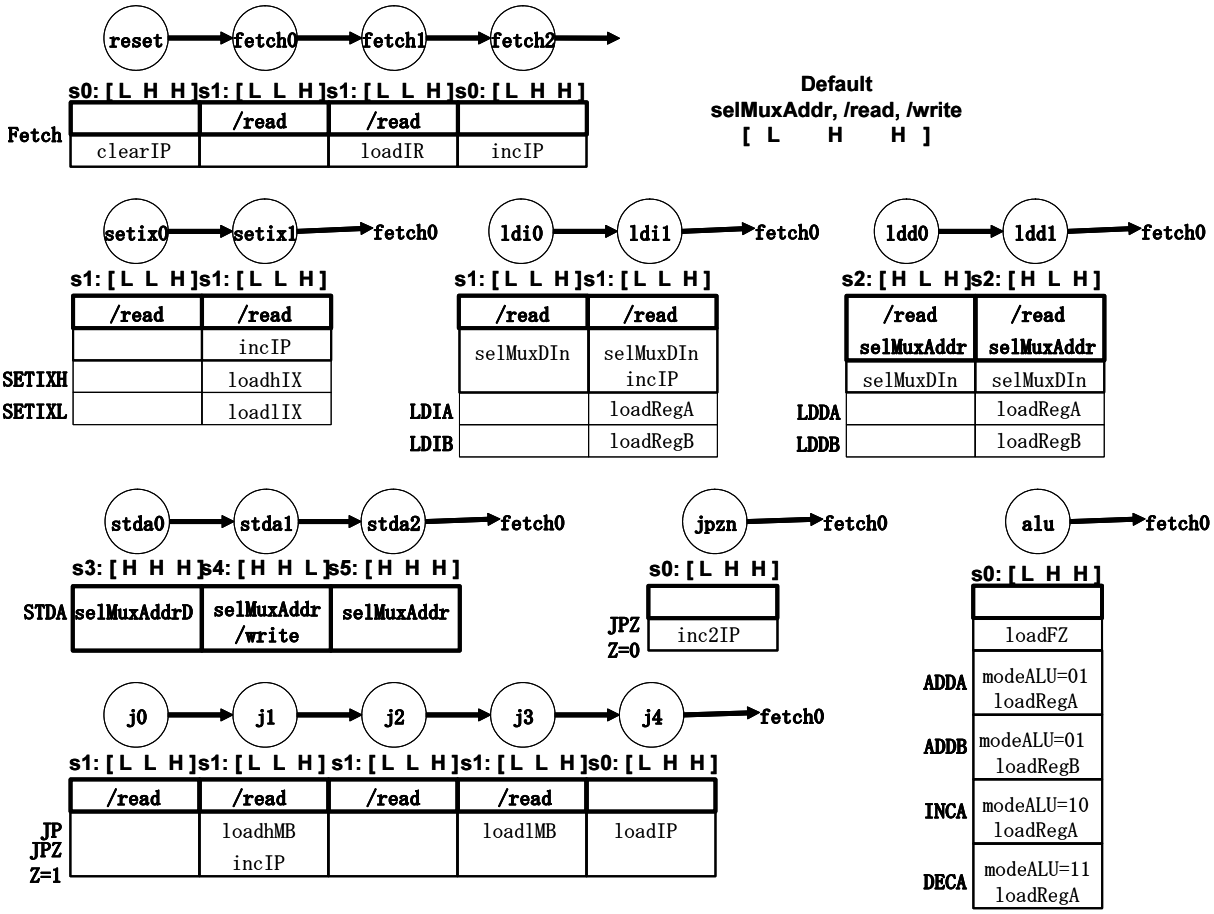


Figure 4.9: Tiny-Processor の外部出力信号用状態遷移

4.9 外部出力信号用状態機械

次に，外部出力信号用状態遷移を 1 つの状態機械 (外部出力信号用状態機械) として統合する．図 4.9 を外部出力信号用状態の遷移に着目して 1 つにまとめると図 4.10 のようになる．(このとき信号制御用状態名も書いておくと後で参照しやすくなる．)

4.10 状態機械のジョンソンカウンタ群への分割

複数のジョンソンカウンタをうまく制御することにより，複雑な状態機械を作ることができる．ここでは，与えられた状態機械を実現する複数のジョンソンカウンタを導出する方法の一つを紹介する．状態機械の一つの状態は，一つのジョンソンカウンタのみが担当する．残りのジョンソンカウンタは各待機状態でストップしている．状態遷移により担当するジョンソンカウンタが変われば，その新たなジョンソンカウンタが待機状態から脱出する．前のジョンソンカウンタは逆に待機状態に入る．一つの状態機械をこのようなジョンソンカウンタ群に分解するには，以下の規則に従えばよい．

- 遷移先が複数存在する状態 (但し自己ループのある状態は遷移先とは見なさない) を分岐状態と呼ぶ.
- 複数の状態から遷移してくる状態 (但し自己ループがある状態は遷移元とは見なさない) を合流状態と呼ぶ.
- 合流状態から分岐状態までの状態遷移の系列を一つのジョンソンカウンタに対応させる.
- 分岐状態の次状態から合流状態の手前までの状態遷移の系列を一つのジョンソンカウンタに対応させる.
- ただし，始点の合流状態が自状態遷移系列の最後の状態から合流するのであれば，始点の合流状態のみ別のジョンソンカウンタに対応させる.

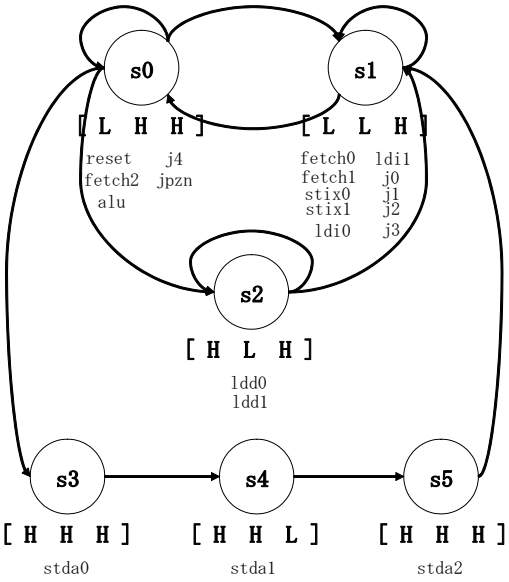


Figure 4.10: Tiny-Processor の外部出力信号用状態機械

4.11 外部出力信号用状態機械のジョンソンカウンタでの実現

図 4.10 をジョンソンカウンタ群で実現すると図 4.11 になる．すべてのジョンソンカウンタが待機状態 (wJCextA, wJCextB，wJCextC) のときのみ，selMuxAddr，/read，/write はデフォルト値をとなり，その他は状態に依存した値を出力する．また，ある一つのジョンソンカウンタが待機状態でない場合は，他のジョンソンカウンタはすべて待機状態になるようにする．各待機状態のループからの脱出用の信号線をそれぞれ cJCextA，cJCextB, cJCextC とし，デフォルト値を L とする．図 4.11 では，状態名の下に，その状態でのジョンソンカウンタの出力値を，丸の下にはそのときの外部出力の値を書いている．

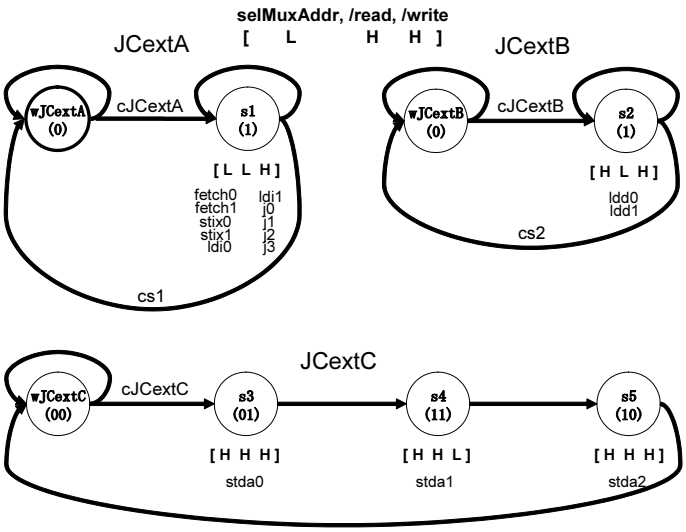


Figure 4.11: Tiny-Processor の外部出力ジョンソンカウンタ

ところで，4.10 節に忠実に従うならば s0 に相当するジョンソンカウンタが必要に思うだろう．確かに s0 を実現するジョンソンカウンタを用意しても構わない．しかし，上で制御信号のデフォルトの値を決定し，デフォルトのときの値の組み合わせが s0 であることを思い出して欲しい.「デフォルトとは信号値を変化させない場合」と考え,「信号値を変化させない場合」とは全てのジョンソンカウンタが待機状態にある場合と考えることで，s0 (デフォルト出力の状態) を扱うジョンソンカウンタを別途設ける必要はないことが分かる．

4.12 外部出力信号のデコード

前節までで作成したジョンソンカウンタの出力値 (状態) によって外部出力信号を制御するデコード論理について考える．図 4.11 を見ると selMuxAddr 信号は，状態 s2，s3，s4，s5 のときに H を出力し，それ以外の状態では L であることが判る．状態 s2，s3，s4，s5 はそれぞれ，

- s2: ジョンソンカウンタ JCextB の出力が 1
- s3: ジョンソンカウンタ JCextC の出力が 01
- s4: ジョンソンカウンタ JCextC の出力が 11
- s5: ジョンソンカウンタ JCextC の出力が 10

に対応するので，ジョンソンカウンタ JCextB の出力が 1, JCextC の出力が 01,11,10 のときに selMuxAddr を 1 にすればよい．従って selMuxAddr のデコード論理は以下のようになる．

```
selMuxAddr <= '1' when qJCextB = '1' else
              '1' when qJCextC = "01" else
              '1' when qJCextC = "11" else
              '1' when qJCextC = "10" else
              '0';
```

/read，/write についても同様にデコード論理を決定する．このとき，/read，/write 信号は負論理である点に注意すること．/read，/write 信号のデコード論理は以下のようになる．

```
read <= '0' when qJCextA = '1' else
        '0' when qJCextB = '1' else
        '1';
write <= '0' when qJCextC = "11" else
         '1';
```

ここで重要なのは，**デフォルト値から変化させる条件を書くこと**である．ちなみに，外部出力信号が変化する状態とそのときのジョンソンカウンタの出力値を表としてまとめると表 4.3 のようになる．

Table 4.3: Tiny-Processor における外部出力信号のデコード論理表

信号名	デフォルト	出力	状態	条件
selMuxAddr	L	H	s2	qJCextB = '1'
			s3	qJCextC = "01"
			s4	qJCextC = "11"
			s5	qJCextC = "10"
read	H	L	s1	qJCextA = '1'
			s2	qJCextB = '1'
write	H	L	s4	qJCextC = "11"

4.13 内部制御信号の制御

外部出力信号用は外部出力信号用ジョンソンカウンタが担当するので，外部出力以外の制御信号である内部制御信号について考える．まず，最初に作成した状態遷移 (図 4.8) から，外部出力信号線を取り除く．このままだと外部出力信号用ジョンソンカウンタは待機状態のまま動作しなくなるので，外部出力を変化させたいところ，つまり，外部出力信号用ジョンソンカウンタが自己ループから脱出し，次の状態へ遷移して欲しいところに外部出力信号用ジョンソンカウンタの遷移条件 (脱出用信号) を新たに制御すべき信号として追加した状態遷移を作成する (図 4.12)．図 4.12 では，外部出力信号用ジョンソンカウンタの脱出用信号をどこで出したらよいかを判りやすくするために，各状態の下にその状態でとるべき外部出力信号用ジョンソンカウンタの状態を示している．

状態 j3 → j4 → fetch0 と遷移する場合について考える．図 4.8 を見てみると，外部出力信号/read は，j3 で L，j4 で H，j5 で L と変化させる必要がある．外部出力信号の制御は外部出力信号用ジョンソンカウンタが行うので，/read 信号を L → H → L と変化させるためには，j3 → j4 → fetch0 の遷移に応じて外部出力信号用

ジョンソンカウンタの状態を s1 → s0 → s1 と変化させる必要がある．図 4.11 を見ると，状態 s1 は外部出力信号用ジョンソンカウンタ JCextA で実現されており，状態 s1 には自己ループがある．したがって，外部出力信号用ジョンソンカウンタ JCextA を s1 → wJCextA(=s0) → s1 と変化させるには，状態が j3 のときに cs1 を H にすることで外部出力信号用ジョンソンカウンタ JCextA が状態 s1 から s0 へ遷移し，j4 のときに cCJextA を 1 にすることで状態 s1 へ遷移させる．図 4.12 を見ると，命令が格納されているレジスタ IR の値を利用して制御信号をデコードすれば，もっと状態数が減らせることがわかる．外部出力用の状態機械はハザード対策のため，IR の値を遷移条件に使用できなかったが，内部制御信号では IR の値により現在実行中の命令が区別できるので，例えば，alu0 と jpnz を一つの状態にまとめることが可能である．同様に setix0 と ldi0 と ldd0，および setix1 と ldi1 と ldd1 も同一の状態にまとめることができる．外部出力信号用ジョンソンカウンタの制御信号も他の制御信号と同等に扱い，IR の値によって外部出力信号用ジョンソンカウンタの制御信号をデコードすることで，外部出力の状態が異なっていたとしても 1 つの内部制御信号の状態として表すことができる．図 4.13 は状態を統合した状態遷移である．図 4.14 は図 4.13 の状態遷移を一つにまとめた内部制御信号用状態機械である．

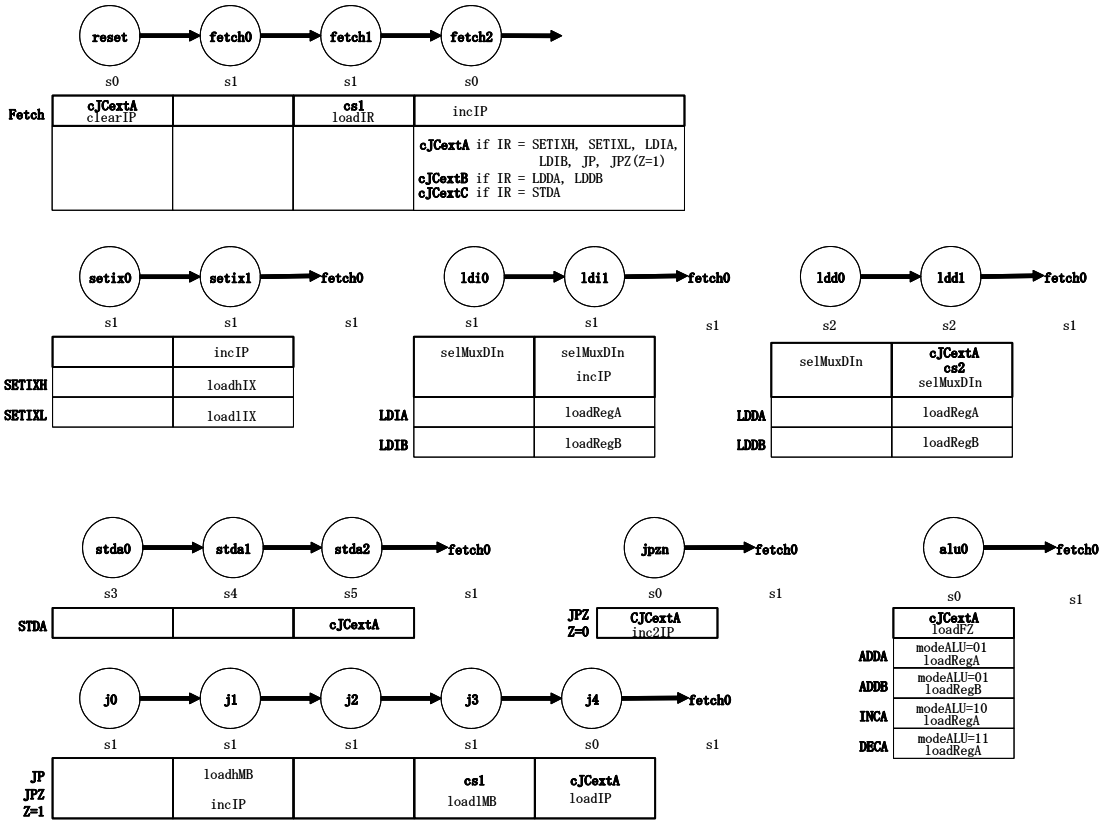


Figure 4.12: Tiny-Processor の内部信号の状態遷移

図 4.14 をジョンソンカウンタ群に分解した結果を図 4.15 に示す．ジョンソンカウンタ JCintC は，担当する状態数が偶数であるため，待機状態を含めて状態数を偶数にするためにダミー状態が入っている．

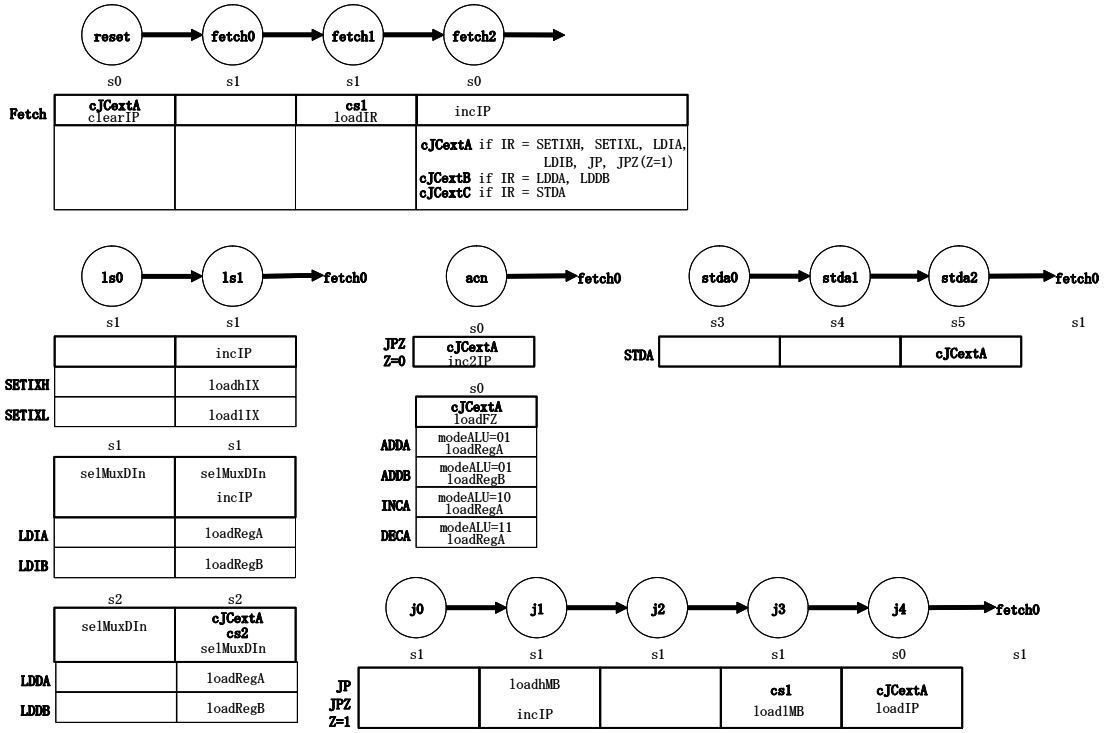


Figure 4.13: Tiny-Processor の統合された内部信号の状態遷移

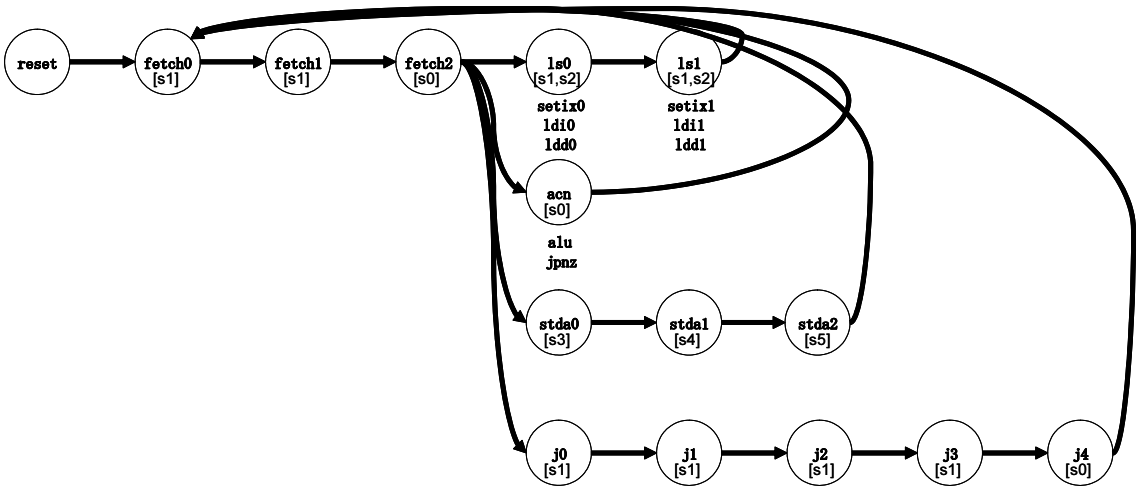


Figure 4.14: Tiny-Processor の統合された内部制御信号用状態機械

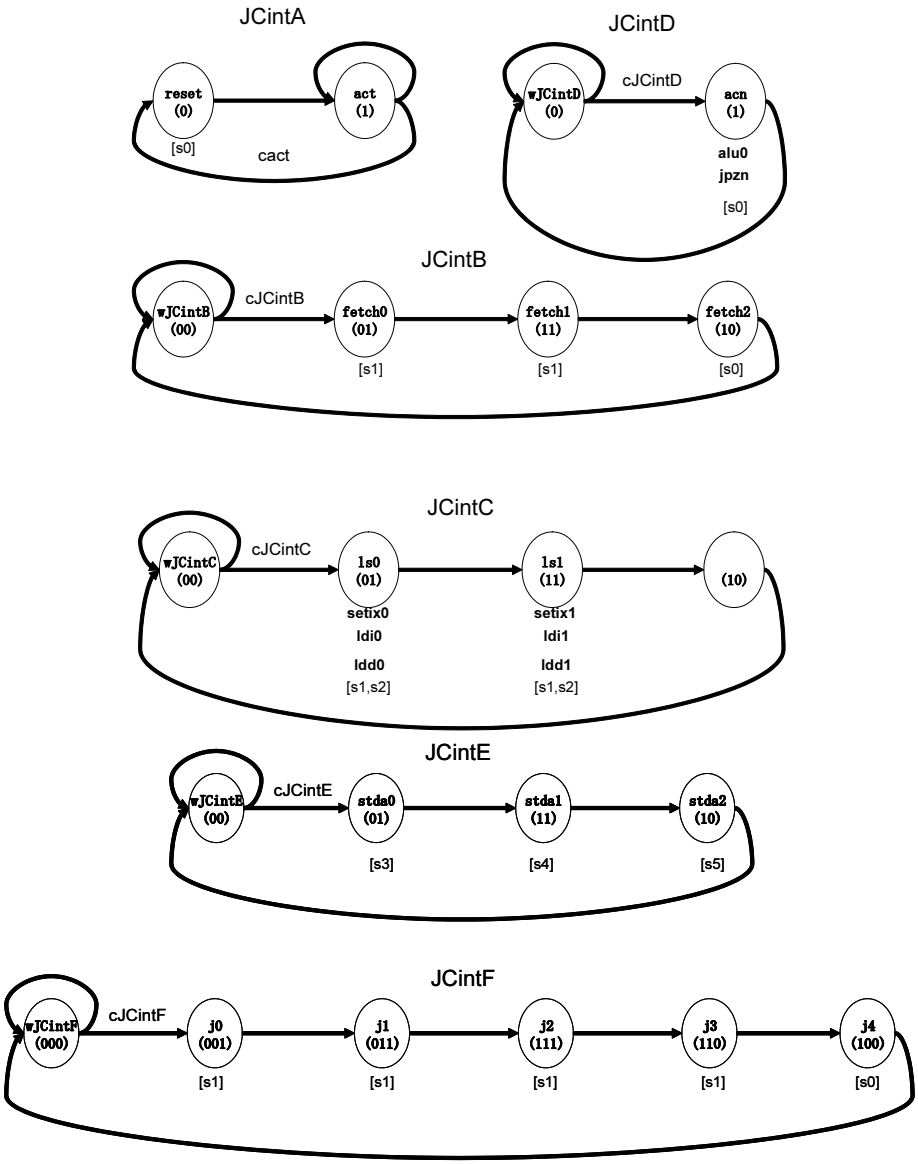


Figure 4.15: Tiny-Processor の内部信号ジョンソンカウンタ

4.14 内部制御信号のデコード論理

内部制御信号のデコードについて考える．まず，外部出力信号用ジョンソンカウンタの脱出条件の考え方について説明する．図 4.13 の各状態の下に，その状態に対応する外部出力信号用ジョンソンカウンタの状態を示している．図 4.13 において，前後の状態で外部出力信号用ジョンソンカウンタの状態が変化しているところでは，外部出力信号用ジョンソンカウンタの脱出条件を考える必要がある．例えば，STDA 命令について考える．STDA 命令の動作をフェッチから考えると，fetch0 → fetch1 → fetch2 → stda0 → stda1 → stda2 → fetch0 と状態遷移する．このとき，外部出力信号用状態は，s1 → s1 → s0 → s3 → s4 → s5 → s1 と変化する．したがって fetch2 から stda0 へ遷移するときに，ジョンソンカウンタ cJCextC をループから脱出させるために，cJCextC を 1 にする必要がある．cJCextC3 を 1 にする条件は，JCintB の出力が 10 で，かつ IR の値が 01000000 のときつまり，

```
cJCextC <= '1' when qJCintB = "10" and
                    irout(7 downto 4) = "0100" else
           '0';
```

となる．stda0 から stda1 へ変化するところでも外部出力信号用状態は変化しているが，このときは脱出などの制御は考えなくてよい．なぜならば，ひとつ前の状態に遷移したときに，JCextC はすでにループから抜け出しており，外部から独立に 1 サイクルごとに状態を変化させはじめているので，stda0 から stda1 へ変化するところで，JCextC は s3 から s4 へと自然と変化する．stda2 から fetch0 へ戻るときは，外部出力信号用状態が s5 から s1 へ変化するので，脱出条件 cJCextA の制御を考える必要があり，ジョンソンカウンタ JCintE の出力が 10 のときに cJCextA を 1 にする必要がある．また，他のジョンソンカウンタ JCintA, JCintC, JCintD, JCintF を考慮すると cJCextA のデコード論理は，

```
cJCextA <= '1' when qJCintA = '0' else
           '1' when qJCintB = "10" and irout(7 downto 4) = "0010" else
           '1' when qJCintB = "10" and irout(7 downto 4) = "1000" else
           '1' when qJCintB = "10" and irout(7 downto 4) = "1001" and ZeroF = '1' else
           '1' when qJCintC = "11" and irout(7 downto 4) = "0001" else
           '1' when qJCintD = '1' else
           '1' when qJCintE = "10" else
           '1' when qJCintF = "100" else
           '0';
```

となる．外部出力信号用ジョンソンカウンタの脱出信号のデコード論理表を書くと表 4.4 の様になる．「IR = 命令名」の部分は，irout のバイナリ値についての条件式 (つまり，irout(x downto x) = "xxxxxxxx" というような) に置き換えて読んで欲しい．

内部制御信号用のジョンソンカウンタの脱出条件について同様に考えることができ，内部制御信号用ジョンソンカウンタの脱出信号のデコード論理は表 4.5 の様になる．また，ジョンソンカウンタの脱出条件以外の制御信号のデコード論理も，外部出力信号のデコードと同じ用に考えればよい．ジョンソンカウンタの脱出信号以外の制御信号のデコード論理は表 4.6 の様になる．

Table 4.4: Tiny-Processor における外部出力信号用ジョンソンカウンタの脱出信号のデコード論理表

信号名	デフォルト値	出力値	内部状態	条件
cJCextA	L	H	reset	qJCintA='0'
			fetch2	qJCintB="10" **and** IR = (SETIXH, SETIXL, LDIA, LDIB, JP, JPZ(Z=1))
			ls1	qJCintC="11" **and** IR = (LDDA, LDDB)
			acn	qJCintD='1'
			stda2	qJCintE="10"
			j4	qJCintF="100"
cJCextB	L	H	fetch2	qJCintB="10" **and** IR = (LDDA, LDDB)
cJCextC	L	H	fetch2	qJCintB="10" **and** IR = STDA
cs1	L	H	fetch1	qJCintB="11"
			j3	qJCintF="110"
cs2	L	H	ls1	qJCintC="11" **and** IR = (LDDA, LDDB)

Table 4.5: Tiny-Processor における内部信号用ジョンソンカウンタの脱出信号のデコード論理表

信号名	デフォルト値	出力値	内部状態	条件
cact	L			ソフトウエアリセットを考えないので，常に L
cJCintB	L	H	reset	qJCintA='0'
			ls1	qJCintC="11"
			acn	qJCintD='1'
			stda2	qJCintE="10"
			j4	qJCintF="100"
cJCintC	L	H	fetch2	qJCintB="10" **and** IR=(SETIXH, SETIXL, LDIA, LDIB, LDDA, LDDB)
cJCintD	L	H	fetch2	qJCintB="10" **and** IR=(ADDA, ADDB, INCA, DECA, JPZ(Z=0))
cJCintE	L	H	fetch2	qJCintB="10" **and** IR=(STDA)
cJCintF	L	H	fetch2	qJCintB="10" **and** IR=(JP, JPZ(Z=1))

Table 4.6: Tiny-Processor におけるジョンソンカウンタの脱出信号以外の制御信号のデコード論理表

信号名	デフォルト値	出力値	内部状態	条件
clearIP	L	H	reset	qJCintA='0'
loadIR	L	H	fetch1	qJCintB="11"
modeALU	LL	**	acn	qJCintD='1' **and** IR = (ADDA, ADDB, INCA, DECA)
loadFZ	L	H	acn	qJCintD='1' **and** IR = (ADDA, ADDB, INCA, DECA)
loadhMB	L	H	j1	qJCintF="011"
loadlMB	L	H	j3	qJCintF="110"
loadIP	L	H	j4	qJCintF="100"
loadhIX	L	H	ls1	qJCintC="11" **and** IR = SETIXH
loadlIX	L	H	ls1	qJCintC="11" **and** IR = SETIXL
loadRegA	L	H	ls1	qJCintC="11" **and** IR = (LDIA, LDDA)
			acn	qJCintD='1' **and** IR = (ADDA, INCA, DECA)
loadRegB	L	H	ls1	qJCintC="11" **and** IR = (LDIB, LDDB)
			acn	qJCintD='1' **and** IR = ADDB
incIP	L	H	fetch2	qJCintB="10"
			ls1	qJCintC="11" **and** IR = (SETIXH, SETIXL, LDIA, LDIB)
			j1	qJCintF="011"
inc2IP	L	H	acn	qJCintD='1' **and** IR = JPZ(Z=0)
selMuxDIn	L	H	ls0	qJCintC="01" **and** IR = (LDIA, LDIB, LDDA, LDDB)
			ls1	qJCintC="11" **and** IR = (LDIA, LDIB, LDDA, LDDB)

(modeALU の ** には，各命令に対応する mode 信号の値が入る)

課題 (必須ではない)

まず，3.2 節および 3.3 節の内容に従って，ModelSim による論理演算回路 Gate および参考用 CPU Tiny-Processor の回路シミュレーションを行うこと．その後，以下の課題に取り組むこと．VHDL 記述の大まかな構造については付録 C を参考にし，詳細は各自ウェブで検索すること．ModelSim の使用法および ModelSim 上での波形の見方の習得が本課題の目的である．

1. `Gate.vhd` に入力 A，B の否定を出力する信号 OUT_NOTA, OUT_NOTB を追加せよ．ModelSim の波形ビューア上で正しく実装されていることを示せ．`TestGate.vhd` 中で Gate を呼び出している部分も変更しなければならないことに注意 (確認はしない).
2. 下記の動作仕様を満たした，Tiny-Processor 上で動作する機械語プログラムを作成せよ．ModelSim 上で波形シミュレーションを行いながら，プログラムが正しく動作していることを説明せよ．説明に際しては，少なくとも IP，IR，IX，レジスタ A，レジスタ B の出力 q の変化を示しながら，正しくメモリ読み出し，加算されている様子を説明すること．IP，IX，IR については 16 進表示で，レジスタ A，レジスタ B は 10 進表示に設定すること．また，最終的に正しいデータが書き出されているかどうかは，Memory ビューアによって示すこと．メモリ表示は 10 進表示にしておくこと．

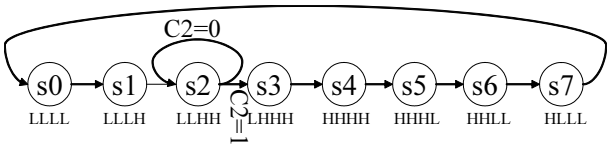
   プログラムの仕様

   - メモリ上の 8000 番地と 8001 番地に格納されている 2 数を加算した結果をレジスタ B (RegB) へ格納する．
   - メモリ上の 8002 番地の値に 1 を加算してメモリ上の 9000 番地へ書き出す.

以上のことが，何を意味するかわからない場合は，もう一度テキストを読み返し，それでもわからなければ質問すること．後半の課題については，`memory.txt` 中に空行が含まれるとテストベンチが動作しないことに注意せよ．

課題 (必須ではない)

1. 付録 A と付録 B を一通り読み理解すること (確認は行わない).
2. 次の状態遷移を実現するジョンソンカウンタを作成したい．挿入する回路の真理値表，論理式を求めよ．



Chapter 5

実験1：CPUの設計(C-Processor)

本章では C-Processor を設計し，シミュレーションによる動作確認を行うまでが実験 1 である．なお実験を進めるにあたって必要な知識が付録 A と付録 B に書かれているので，実験 1 に入る前にひととおり読んでおくこと．
4 章で説明された以下の様な設計手順を行い，C-Processor を作成する．

- データパスの決定と実装
- 制御部の設計 (外部出力信号，内部制御信号，その他の信号の 3 つの信号制御)
- C-Processor の実装

5.1 データパスの決定と実装

実験 1 で作成する CPU である C-Processor のデータパスを図 5.1 に示す．C-Processor では，値をそのままメモリに格納する STDI 命令，RegB の値をメモリに格納する STDB 命令，加算以外の演算命令，Carry フラグを利用する JPC 命令などが存在する．これらを実現するために，Tiny-Processor に存在しない部品や ALU に演算命令を追加する必要がある．そこで，MuxDOut として利用されている 4 入力マルチプレクサの作成と ALU に演算処理の追加を行う．

課題 1 (10/3)

1. 4.3 節以降を読み C-Processor の命令セットに含まれる全ての命令について，図 4.2 や図 4.3 のようなデータの流れの説明ができるようにすること．
2. `Resource.vhd` 内の Mux2x08 を参考に，8 bit の 4 入力 (a，b，c，d) から 1 つを選択して q へ出力するマルチプレクサ Mux4x08 を VHDL で記述せよ．`TestMux.vhd` を参考にテストベンチを作成し，設計したマルチプレクサが正しく動作することを ModelSim 上で示せ．
3. 下に示す modeALU 信号を用いて C-Processor 用の ALU を VHDL で記述し，`TestALU.vhd` を参考にテストベンチを作成し，設計した ALU が正しく動作することを ModelSim 上で示せ．

2，3 ともにチェックの際には，自分で作成したテストベンチで検証が十分であることを説明できるようにしておくこと (つまり，入力パターンを十分に調べ尽くしていること)．
ALU の記述で VHDL の ‘+’ や ‘−’ の算術演算子を使ってはいけない．これらの演算子は論理合成によって加算器になるため，せっかく ALU で演算器を共有設計した意味がなくなってしまう． HDL で算術演算子を書くことは,「論理合成に任せて演算器を生成しても良い」という意思表示だと思うこと．

modeALU 信号割り当て

LLLL	A+B	LHLL	not A	HLLL	not B
LLLH	A-B	LHLH	A+1	HLLH	B + 1
LLHL	A and B	LHHL	A - 1	HLHL	B - 1
LLHH	A or B	LHHH	未使用	HLHH	未使用

ヒント：減算は加算で実現可能．減算のために新たに加算器を用意する必要はない．負数はどうすれば作れるのかを思い出すこと．ALU の入力 cin は，キャリー入力だが，これを 1 にすることで演算結果に 1 を足すことができる．

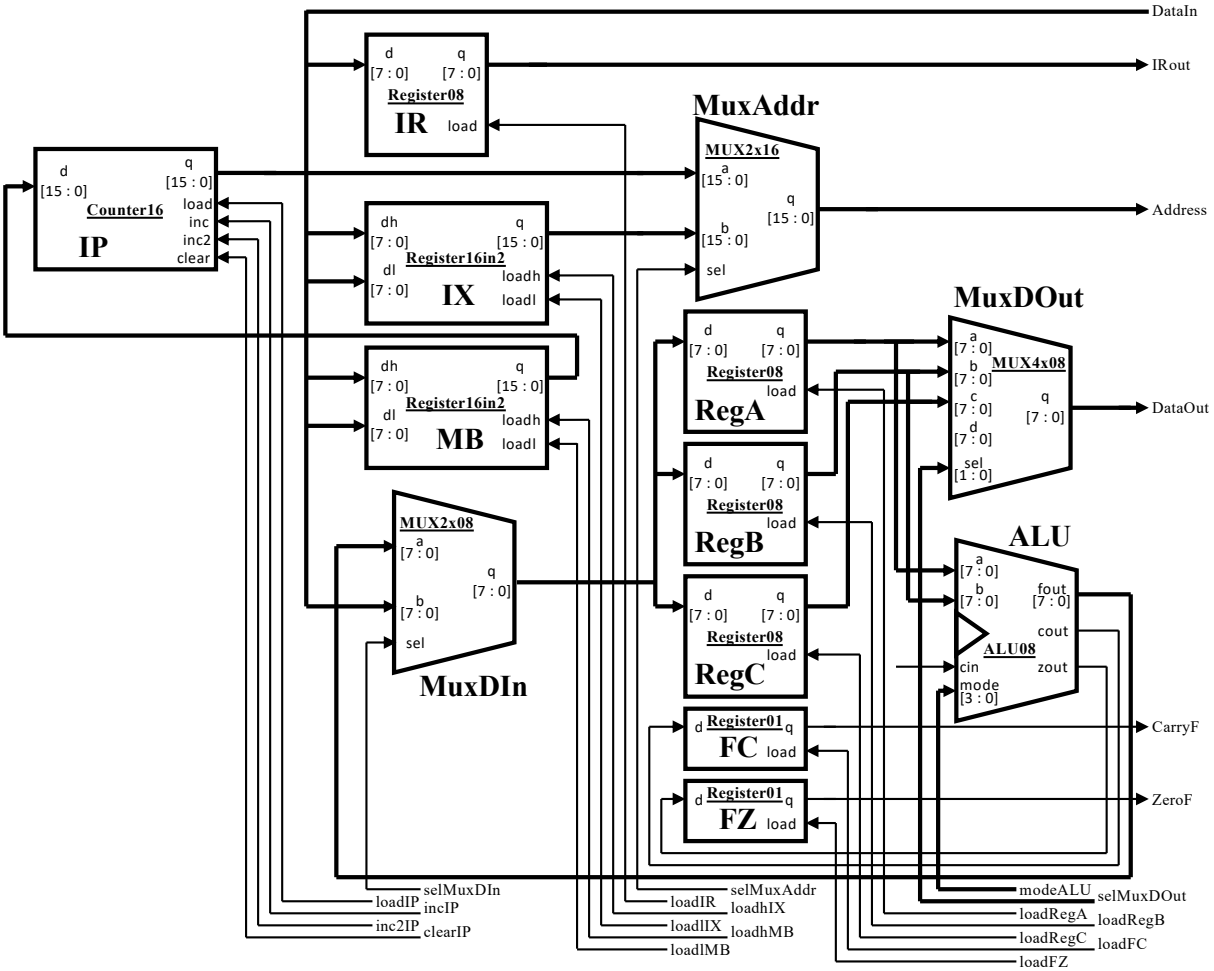


Figure 5.1: C-Processor のデータパス

課題 2 (10/17)

1. データパス部を VHDL で実装せよ．(Tiny-Processor の `DataPath.vhd` の記述を修正すればよい)
2. テストベンチ `TestDataPath.vhd` によりデータパス部を単体検証せよ．各自追加したコンポーネントや信号の命名が配布データと違っている可能性が高いため，テストベンチ中の信号が自分のデータパスの信号名と違う場合は，テストベンチ中の信号名を適切な名前に (一斉) 置換せよ．`TestDataPath.jpg` のような波形が出力されることを確認せよ．

ヒント： アーキテクチャ記述内で新たな信号を増やした場合は，宣言部 (エンティティ記述もしくは内部信号宣言) も修正する必要があることに気をつけよ．これを忘れると次週以降のバグの元となる．
また，シミュレーションに使用するファイルは，`TestDataPath.vhd`，`DataPath.vhd`，`Resource.vhd` だけでよい (ALU や MUX は `Resource.vhd` に含めておくという前提で)．無駄なファイルを含めると，正しくシミュレーションできない可能性がある．

5.2 制御部の設計

全体の制御信号：ここでは，制御部全体の設計を行う．まず初めに，4.5 節から 4.7 節で説明されている様に，命令全体を対象にデータパスの制御をどの様に行うかを設計する．

課題 3 (10/24)

1. データパスの制御，メモリアクセスに必要な制御信号を抜き出し，各制御信号のデフォルト値を決定し，ハザード対策を行わなければならない制御信号を示せ (表 4.2 に相当).
2. C-Processor の全命令の信号状態遷移を作成せよ (図 4.7 に相当).

- 「メモリアクセスに必要な制御信号」とは，メモリへの入力信号である Address，DataOut，/read，/write に関係のある信号である．表 4.2 では，一見関係の無さそうな selMuxAddress がハザード対策要となっているが，マルチプレクサは組合せ回路であり，入力の変化が出力に影響するためである．つまり，入力である選択信号 selMuxAddress がハザードでブレると，出力の Address にブレが及んでしまうのである．

外部出力信号制御：次に，外部出力信号の制御を行うために 4.8 節から 4.12 節で説明されている様に，状態遷移図やジョンソンカウンタを利用して外部出力信号の制御について設計を行う．

課題 3 (10/24)

1. C-Processor の外部出力信号用状態遷移を作成し，ハザード対策が必要な外部出力信号線の組み合わせに対して状態を割り当てよ (図 4.9 に相当).
2. 外部出力信号用状態遷移を 1 つにまとめた外部出力信号用状態機械を作成せよ (図 4.10 に相当) .
3. 外部出力信号用状態機械を複数の外部出力信号用ジョンソンカウンタに分解せよ (図 4.11 に相当) .
4. 外部出力信号のデコード論理表を作成せよ (表 4.3 に相当).

内部信号制御：内部信号の制御を行うために 4.13 節と 4.14 節を行うことで，内部信号の制御部分を設計する．

課題 4 (10/31)

1. 外部出力信号を除いて，外部出力信号制御用ジョンソンカウンタの脱出信号を追加した，内部制御信号用状態遷移を作成せよ (図 4.13 に相当).
2. 内部制御信号用状態機械図を作成せよ (図 4.14 に相当).
3. 内部制御信号用状態機械をジョンソンカウンタに分解せよ (図 4.15 に相当).

その他の信号制御：ジョンソンカウンタがある状態で待機状態になっている場合に，その状態から脱出するための脱出信号を作成する必要がある．合わせて，外部出力信号，内部出力信号以外の制御信号も設計する．

課題 4 (10/31)

1. 外部出力信号用ジョンソンカウンタの脱出信号のデコード論理表を作成せよ．(表 4.4 に相当)
2. 内部制御信号用ジョンソンカウンタの脱出信号のデコード論理表を作成せよ．(表 4.5 に相当)
3. ジョンソンカウンタの脱出信号以外の制御信号のデコード論理表を作成せよ．(表 4.6 に相当)

制御部の実装：これまで作成したジョンソンカウンタやデコード理論を制御部に VHDL で実装する．

課題 5 (11/14)

課題のチェックはソースコードのチェックで行う．

1. 外部出力信号用ジョンソンカウンタ，外部出力信号のデコード論理を VHDL として記述せよ．
2. 内部制御信号用ジョンソンカウンタを VHDL として記述せよ．

ヒント： いずれも Tiny-Processor の `Control.vhd` の該当箇所に追加して記述すればよい．

5.3 C-Processor の実装

これまで行ってきた，制御部とデータパスを利用して，C-Processor を実際に動作できることろまで実装を行う．

課題 6 (11/21)

1. C-Processor を VHDL で記述せよ．アドレス指定系命令，ロード系命令，ストア系命令の全命令 (表 2.3 参照) が全ての場合に正しく動作するかプログラム (メモリイメージ) を書いて確認せよ．全ての動作が確認できたら，動作確認を受けること．実際に動作確認する命令は，下記の命令とする．ただし，プログラムではこの順番で実行する必要はない．

   チェック時に動作確認する命令：SETIXH，SETIXL，LDIA，LDDB，STDA，STDI

   ヒント： 課題 4 でのデータパス部の検証と同様に，制御部のテストベンチ `TestControl.vhd` を用いて制御部の検証をしても良い．

課題 7 (11/28)

1. 演算系命令，実行制御系命令の全命令 (表 2.3 参照) が動作することを確認せよ．条件付き分岐命令は成立する場合としない場合の両方，演算に関しては，フラグが 1 から 0 へ，0 から 1 へ変化する場合と変化しない場合を含めて，全ての場合に正しく動作するかプログラム (メモリイメージ) を書いて確認せよ．

   全ての動作が確認できたら，TA による動作確認を受けること．実際に TA により動作確認する命令は，下記の命令とする．ただし，プログラムではこの順番で実行する必要はない．また，TA による動作確認の際には，全ての場合を列挙しなくてよい．

   チェック時に動作確認する命令：ADDA，SUBB，ANDA，ORB，INCA，DECB，NOTA，CMP，NOP，JP，JPC，JPZ

   なお，ModelSim でシミュレーションを実行して，Memory List からメモリイメージを参照できない場合，Simulation → Start Simulation から，Others タブ→ Profiler の Enable Memory Profiling にチェックを入れること．

課題 8 (11/29)

1. 負数を含む 8 bit の 2 整数の乗算プログラムを書き，正しく計算されることを示せ．チェックでは,「正しく計算できている」と十分確認できるように，必要な信号を ModelSim 波形ウィンドウに含めて教員・TA に説明を行うこと．
   - 結果が 8 bit を超えるような乗算は対象としない．
   - 計算させる値は，動作確認時に次の 3 組を指定する．(1) 2 番目の数が 0，(2) 両方とも負数，(3) その他の数字．
   - 計算させる値は，2 つともあらかじめメモリに入れるようにして，そこから読み込むようにせよ．また，計算結果はメモリの特定のアドレス (任意) に書き込むようにせよ．
   - 1 回の実行で 3 組とも計算しても，1 組ずつ 3 回の実行で行っても構わない．

始める前に必ず行うこと

必ず拡張前の C-Processor のファイル一式 (フォルダごと) をコピーして別に保存しておくこと．実験 2 では，拡張前の通常版 C-Processor を使うように．

a. プロセッサ命令セットの拡張 (難易度：標準〜難しい)

C-Processor の命令セットに新たな命令を自由に追加せよ．例えば，C-Processor にはシフト演算などが含まれていない．CASL 等を参考にして決めてもよいだろう．

1. まず，追加する命令の仕様を決めよ．(命令コード・オペランド有無・動作)
2. その動作を達成するためのレジスタや信号を加えてデータパスを変更せよ．場合によってはコンポーネント自体を新たに書く必要がある．
3. 同様に制御信号についても追加，変更を行うこと．
4. 新たな命令を含むプログラムを作成し，シミュレーションによって正しく実装されているかを検証せよ．

b. 簡易アセンブラの作成 (難易度：易しい〜標準)

アセンブリ言語で書かれたプログラムを入力すると，`memory.txt` のようなメモリイメージを出力するようなアセンブラを開発せよ．

1. 入力となるアセンブリ言語の仕様 (命令セット・書き方) を決めよ．始めはアセンブリ命令を機械語と 1 対 1 に対応づけるように決めるのが良いだろう．
2. 実装言語を決めてアセンブラをプログラミングせよ．実装に用いる言語は自由である．プログラミング環境の整備は各自で行うこと．入出力を手軽に扱える Perl あたりが楽であろう．
3. アセンブリ言語を用いてプログラムし，出力されたメモリイメージを C-Processor 上でシミュレーションして検証せよ．
4. 複数の機械語命令に対応するアセンブリ命令なども取り扱えると便利だろう．(SETIX とか)

c. 演算器の改善 (難易度：難しい)

C-Processor の ALU 内の加算器は，桁上げ (Carry) を次の全加算器に順繰りに送っていく実装となっている．これは，波のように桁上げが伝播していくことから，Ripple Carry Adder (RCA) と呼ばれる．しかし，入力のビット幅が大きくなると，RCA では桁上げが伝播する経路がクリティカルパスとなり，低速になってしまう．そこで，桁上げ先見回路を追加した加算器に置き換えることを考える．

1. Carry Look-ahead Adder (CLA) では，RCA より少ないゲート遅延で桁上げ伝播を行うことが可能である．CLA について復習・調査せよ．(調べた文献・ウェブページはレポートに参考文献として引用元の情報を書くこと)
2. CLA を VHDL で実装し，演算器を入れ替えよ．
3. シミュレーションによって正しく動作していることを確認せよ．
4. 他にも加算器高速化の手法は多数ある．調べてみるとよいだろう．

5.4 デバッグのヒント

CPU 設計および ModelSim によるシミュレーション時によくあるミス，デバッグ時のチェック事項をいくつか上げておくので，参考にすること．

1. ModelSim によるシミュレーションがうまくいかない
   - VHDL ファイルのあるディレクトリを指定する
   - File → Change Directory でディレクトリを指定しているか (3.2 節参照)
   - シミュレーション実行時に Enable optimization チェックボックスを外しているか (3.2 節参照)
   - `memory.txt` に空行を含んでいないか (3.3 節参照)
   - コンポーネントの階層は展開できる．内部の信号まで確認しなければ，発見が難しいバグもある．
2. エラーメッセージが出て，コンパイルが通らない
   - タイプミス，構文ミスをよく確認する．
   - 名前の勘違いなどをもう一度見直す．
   - コンポーネントインスタンス化の際，ポートの不足やビット幅の不一致などがないか信号の宣言を確認する．
3. コンパイルは通るが，波形に赤い線が出る
   - コンパイル時に必要なファイルを含めていないため，部分的にブラックボックスになっている場合がある．コンパイル時は C-Processor 本体, Datapath, Resource, Control, テストベンチの VHDL ファイルをすべて指定しなければならない．
4. コンパイルは通るが動作がおかしい．
   - まずきちんと実行されない命令がどれなのかを特定すること．
   - ジョンソンカウンタのカウント値を Wave ウィンドウに含め，カウンタがきちんと遷移しているかを確かめる．
5. ジョンソンカウンタもきちんと動いているはず，だが正しく動かない
   - 残るはデコード論理の誤り．デコード論理をもう一度きちんと見直すこと．コピーペーストミス，タイプミス等些細なミスが多い．and, or を含む条件文が意図通りに評価されているか括弧付けなどを確かめる．
6. それでもうまく動かない
   - プログラム側を確認すること．自分の書いたプログラム (メモリイメージ) が正しく呼び出されているか確認．
   - 同じ命令でも命令コードは Tiny-Processor とは異なることに注意する．

中間レポート

提出方法

12/9 17:00 までに CLE から提出すること．レポートのファイル形式は PDF 形式とし，ファイル名は「学籍番号下 4 桁.pdf」とすること (例えば学籍番号が 09B10099 であれば，0099.pdf) .

レポート内容

以下の内容について報告すること.

1. 設計した C-Processor のデータパスアーキテクチャ図を示し，Tiny-Processor のデータパスからの変更点について説明せよ.
2. STDI 命令のデータの流れについて，1. で示した図上にわかるように矢印を書き込んだ上で説明せよ.
3. C-Processor の外部出力信号用ジョンソンカウンタと内部制御信号用ジョンソンカウンタを示せ.
4. C-Processor の設計にあたって工夫した点やアピールする点があれば，章立てて報告すること. 拡張課題についても同様.
5. 感想 (C-Processor の設計は易しかった／難しかったか？ソフトウェアプログラミングとの違いは感じたか？疑問に思ったこと，授業でもっとフォローして欲しいこと，など)

※レポートのコピーが発覚した場合は，提供者も含めて即不合格にする場合がある.

Chapter 6

実験2：CPUのFPGAボードへの実装

実験 1 では，Tiny-Processor を拡張して C-Processor を設計した．さらに，シミュレータ ModelSim を用いて，設計した C-Processor が正しく動作するかどうか，計算機上での検証を行った．実験 2 では，C-Processor の実機への実装を行う．実際に LSI チップを製作することは時間やコストの制約上困難なので，FPGA を用いて C-Processor を実装する．FPGA とは何度も回路データを書き込み使用することのできるデバイスであり，LSI チップ実装に比べて速度の面で劣るが，早期テストやプロトタイピング (試作) などによく用いられる．

本章で使用するファイル

実験 2 では下記のファイルを用いる．事前にホームページからダウンロードしておくこと．なお，ファイル名のうち xxxxxx は最終更新の日付が入る．ホームページにあるファイルは適宜更新されることがあるため，最新版を使うこと．

- `de0cv-lab2-20211001.zip`
  TinyProcessor の Quartus Prime プロジェクトファイル．

6.1 CPU のFPGA ボードへの実装の手順 (実験 2 の流れ)

実験 2 では，実験 1 で設計した C-Processor を FPGA ボード上に実装し，FPGA ボード上でアセンブラプログラムを実行させる．その手順は以下の通りである．

1. VHDL で書かれた C-Processor の回路データ (`Control.vhd`, `DataPath.vhd`, `Resource.vhd`, `CProcessor.vhd`), および CPU 周辺回路と ROM と RAM (`CPUCircuit.vhd`, `rom1p.vhd`, `ram1p.vhd`)，アセンブラプログラムが記述されているファイル (`rom.mif`) を Quartus Prime でコンパイルする．これは，使用するデバイスの情報，デバイスのピンとの対応関係の情報，C-Processor および CPU 周辺回路データを埋め込み，回路情報ファイル (以下，コンフィギュレーションファイル) を作成することである (`de0cv-lab2.sof` ファイルを作成する)．その後，コンフィグレーションファイル (`de0cv-lab2.sof` ファイル) を FPGA ボードにダウンロードする．6.4 節参照.
2. FPGA ボードのリセットボタンを押して，プログラムの動作を開始させる．
3. アセンブラプログラムを書き換える (`rom.mif` を書き換える) 場合は，Quartus に含まれる mif ファイルエディタを利用するか，テキストエディタで変更を行う．

実験 2 で使用する FPGA ボードは簡単な入出力装置を備えている．実験 1 では，メモリ上に置かれているデータに対して乗算を行い，結果をメモリ上あるいはレジスタ内に格納していた．実験 2 では，外部入力装置から値を入力し (取り込み)，演算した結果を外部出力装置に出力する (表示する)．FPGA ボードの詳細な仕様については 6.2 節を，CPU 周辺回路の仕様については 6.3 節を参照すること．

6.2 FPGA ボードの仕様と接続・設定

実験 2 では設計した回路を，FPGA デバイスと入出力装置を備えた FPGA ボードである DE0-CV に実装する．本節では FPGA ボード DE0-CV の説明を行う．FPGA ボード DE0-CV は，FPGA デバイスとして ALTERA 社製 CycloneV デバイスファミリ 5CEBA4F23C7 を搭載し，外部にメモリ，シリアル，スイッチ，LED などを

持つ．FPGA ボード DE0-CV の外観を図 6.1 に示す．以降，FPGA ボード DE0-CV の主な周辺装置，PC との接続方法，設定方法の概要の説明を行う．詳細は DE0-CV ユーザーズマニュアルを参照すること [1]．

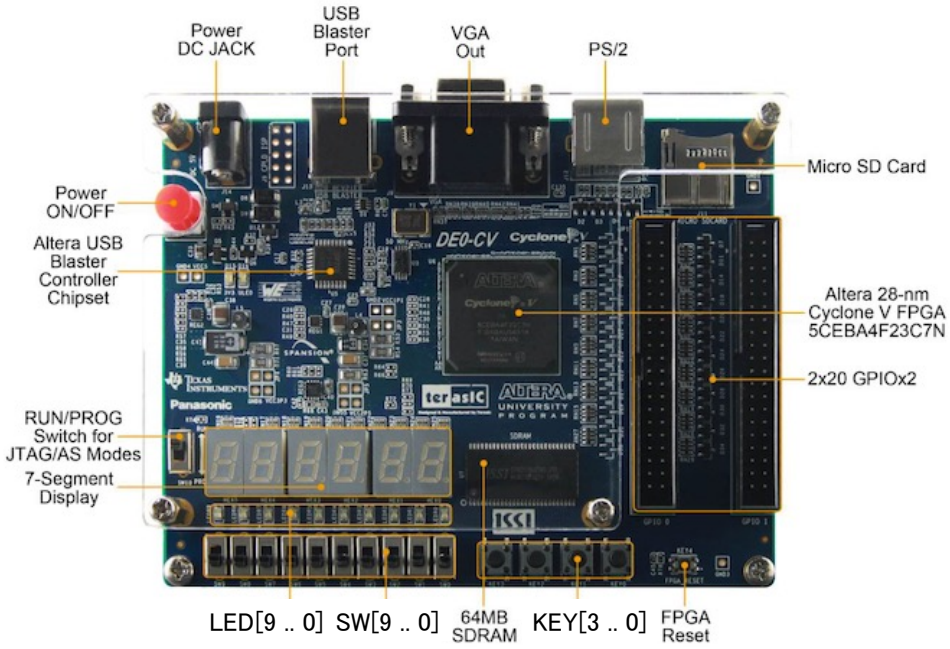


Figure 6.1: FPGA ボード DE0-CV の外観

リセットスイッチ (FPGA Reset)

押下すると，リセット信号 (L) が FPGA (CycloneV) に入力される．

プッシュボタン (KEY[3 .. 0])

CPUCircuit では，ボタンが押されると 1，押されていないと 0 となる．

スイッチ (SW[9 .. 0])

10 個のスイッチがあり，下側が 0，上側が 1 を表している．

LED (LED[0 .. 0])，7 セグメント LED

LED と 7 セグメント LED が装着されている．いずれも正論理で動作する．すなわち，L 出力時は消灯し，H 出力時は点灯する．

メモリ

DE0-CV には 64MB の SDRAM メモリが搭載されているが，実験 C では利用しない．

6.3 CPU 周辺回路の仕様

CPU の動作確認のために，CPU と FPGA ボード中の入出力装置を接続する回路 (以下，CPU 周辺回路) を用意している．(CLE からダウンロードできる VHDL ファイルのうち，`CPUCircuit.vhd`，`ram1p.vhd`，`rom1p.vhd` に相当する)．

[1] ユーザーズマニュアルは `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921&PartNo=4` からダウンロードできる．ただし，マニュアルには，メモリの I/O についてはほとんど書かれていないので注意．

CPU 周辺回路では，外部入出力装置をメモリアドレスの一部に割り当て，そのアドレスに書き込む (読み込む) ことで外部入出力装置への出力 (入力) を実現する．外部入出力装置のアドレス割り当ては表 6.1 の通りである．また，結線されている外部入出力装置とその用途は表 6.2 の通りである．FPGA ボードに装着されている外部入出力装置のうち表 6.2 に書かれていないものを使用したい場合は，CPU 周辺回路 (`CPUCircuit.vhd`) を書きかえることで，使用可能となる．

Table 6.1: メモリアドレスの割当

アドレス	用途
0x0000-0x0400	ROM
0x0401-0x7FFF	非使用
0x8000-0x8100	RAM
0x8101-0xFFFD	非使用
0xFFFE	プッシュボタン入力用
0xFFFF	スイッチ入力用 (SW[7 .. 0])

Table 6.2: 配布 CPU 周辺回路で結線されている入出力装置とその用途

入出力装置	信号名	用途
スイッチ SW7–SW0	SW[7..0]	入力用
プッシュボタン KEY3–KEY0	KEY[3..0]	入力用
スイッチ SW9–SW8	SW[9..8]	信号表示用
7 セグメント LED HEX5–HEX4	HEX5 – HEX4	Address の下 8 bit を表示
7 セグメント LED HEX3–HEX0	HEX3 – HEX0	信号表示用 SW[9..8] = ”00” → Address(16bit) を表示 SW[9..8] = ”01” →IRout (8 bit)，DataOut (8 bit) を表示 SW[9..8] = ”10” →Aout (8 bit)，Bout (8 bit) を表示
LED LEDR0	LEDR[0]	ZeroF 信号を表示
LED LEDR1	LEDR[1]	CarryF 信号を表示
リセットスイッチ	RESET	reset 入力用

図 6.2 に Processor (CPU) と外部デバイスとの結線図 (CPU 周辺回路の結線図) を示す．以降，CPU 周辺回路の各モジュールの説明を行う．動作の詳細については，`CPUCircuit.vhd` のソースコードを参照すること．

KeyEnc4

KeyEnc4 は 4 bit のエンコーダである．プッシュボタンの入力に対してチャタリングを除去し，入力を 4 bit にまとめる．プシュボタンからの入力は正論理 (ボタン押下時に 1，開放時に 0) として CPU へ入力される．チャタリングの除去には 15 クロックを要する．

DecSeg

DecSeg は 4 bit の数値データを 7 セグメント LED の on/off 情報に変換するデコーダである．

InputCtrl と SelInput

アドレス値によって，入力装置から読み込むか，ROM/RAM から読み込むかを切替える．アドレス値が 0xFFFE であればプッシュボタンから入力，0xFFFF であればスイッチから入力，それ以外であれば ROM/RAM から読み込むように切り替える．

SelOutput

スイッチ SW[9..8] で 7 セグメント LED に表示する情報を変更する SW[9..8] が ”00 ”であれば，アドレス値を，SW[9..8] が ”01 ”であれば DataOut と IR のレジスタ値 IRout を，SW[9..8] が ”10 ”であればレジスタ A の値 RegA とレジスタ B の値 RegB を出力する．

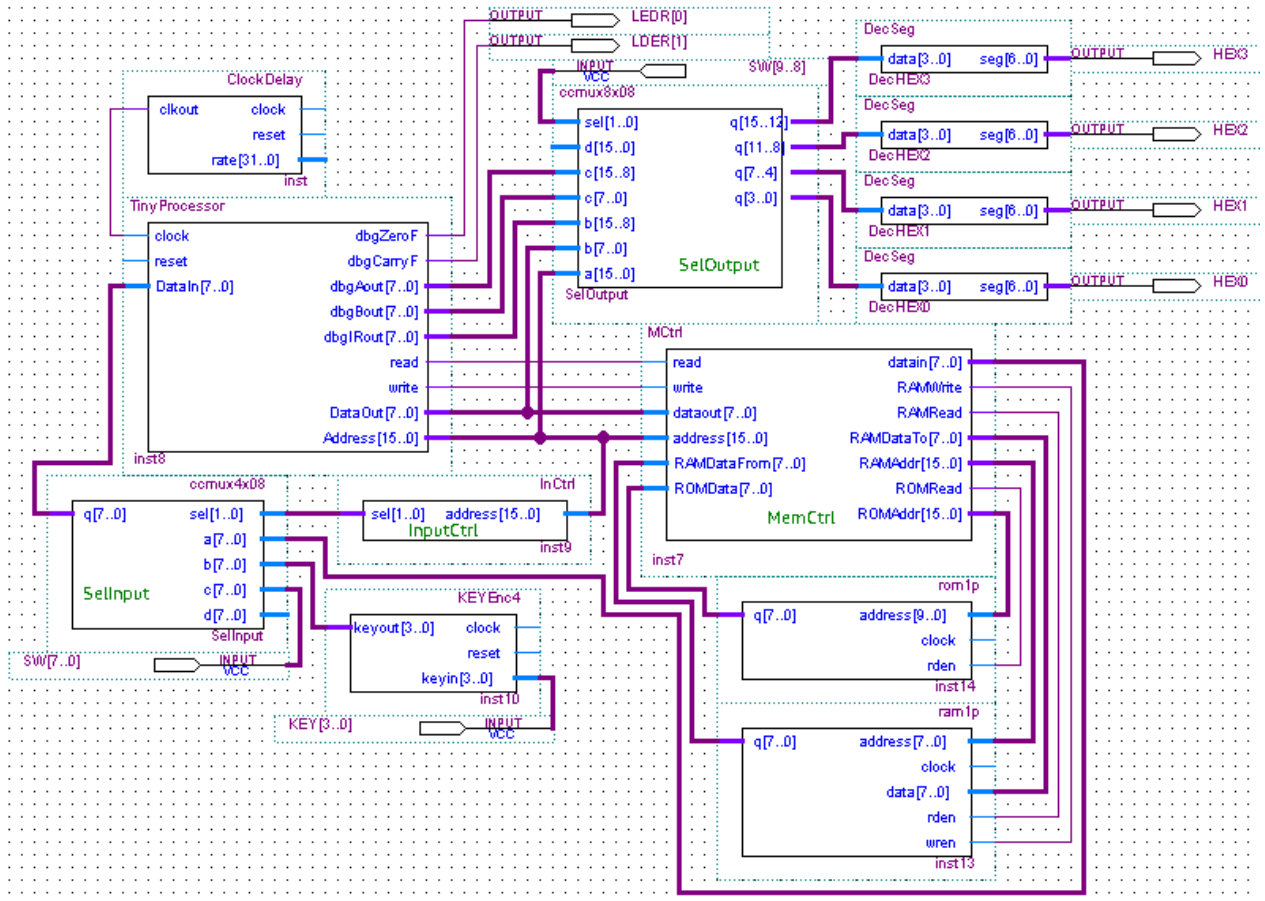


Figure 6.2: CPU と外部デバイスとの結線図

MemCtrl

アドレス値によって，ROM から読み込むか，RAM から読み込む/書き込むかを切替える．すなわち，アドレス値が 0x0000–0x0400 であれば ROM から読み込み，0x8000–0x8100 であれば RAM から読み込む/書き込む．ROM/RAM の CS (Chip Select，負論理) とも接続されている．

ROM/RAM

アセンブラプログラムが保存されている ROM である．ROM の内容は，`rom.mif` の内容が反映される．

ClockDelay

DE0-CV のクロック周波数は 50MHz である．`CPUCircuit.vhd` に含まれる信号線 `clkdelayrate` の値を変更することによって，動作周波数を 50MHz/`clkdelayrate` に変更することができる．

6.4 Quartus Prime による CPU および CPU 周辺回路の FPGA ボードへの実装

実験 C では Altera 社の CAD ソフトウェア「Quartus Prime」を用いて，使用するデバイスに関する情報や，回路の入出力とデバイスのピンとの対応関係等の情報が入ったファイル (.sof という拡張子を持つ) を作成する．また，Quartus Prime を用いてこの.sof ファイルを FPGA ボードにプログラム (直感的には，プログラムというよりも.sof ファイルの FPGA ボードへのダウンロード) することにより，VHDL 回路を FPGA に実装する．以降，de0cv-lab2.sof を作成し，FPGA ボードにダウンロードする方法を説明する．前節と同様に，実際に動作しながら読み進めてほしい．なお，`de0cv-lab2-20xxxxxx.zip` を展開後に現れる quartus ディレクトリに本節で用いるプロジェクトファイルが入っている．

Quartus Prime の起動，プロジェクトファイルの読み込み

まず，Quartus Prime Lite を起動，File→Open Project を選び，Quartus プロジェクトファイルである `de0cv-lab2.qpf` を選ぶ．あるいは，`de0cv-lab2.qpf` をダブルクリックしてもよい．なお，Quartus Prime を起動すると，Qurtus Prime の Update を行うかどうかを訪ねるウィンドウが開く場合がある．その場合は「No」を選択する．

回路のコンパイル

Processing → StartCompilation で回路をコンパイルする．数分でコンパイルが終了し，エラーがなければ，`de0cv-lab2.sof` ファイルが生成される．

プログラマ設定

Quartus Prime で設計やコンパイルを行った回路コンフィギュレーションデータを FPGA ボード DE0-CV へダウンロードする場合，プログラマの設定が必要である．なお，本節で述べる設定はプロジェクトなどに依存しないので，1 回設定すればよい．まず，DE0-CV を USB ケーブルで PC に接続する．次に，Tools から Programmer を選び，Programmer ウィンドウ内の Hardware Setup の右側に表示されるハードウェアが，USB-Blaster でない場合は，Hardware Setup ボタンを押して，Harware Setup ウインドウを開く．Hardware Setup ウィンドウの Currently selected hardware で，USB-Blaster を選択し，close で Hardware Setup ウインドウを閉じる．

DE0-CV を PC に接続した際に，ドライバの設定画面が出てきた場合は，その PC にドライバが設定されていない．ドライバの設定には，管理者権限が必要なため，TA ないしは教員を呼ぶこと．

また，Programmer を起動した際に，“Instration of JTAG server as a windows service failed, ...” というエラーウインドウが表示されることがある．これは，Quartus Prime に USB Blaster(DE0-CV に内蔵) がうまく設定されていないことを意味する．設定には管理者権限が必要なため，この場合も，TA ないしは教員を呼ぶこと．

FPGA ボードの接続と準備

パソコンと FPGA ボード DE0-CV を，USB ケーブルを介して接続する．そして，FPGA ボードに電源を接続し，電源を投入する．FPGA ボードの設定については 6.2 節を参照すること．FPGA へのプログラムの前に，**RUN/PROG Switch for JTAB/AS Modes が RUN 側に設定されていることを確認すること**．

FPGA へのプログラム

`de0cv-lab2.sof` ファイルを FPGA ボード DE0-CV へ書き込む (プログラミングする) ことで，設計した回路が FPGA 上で動作するようになる．Quartus Prime の Tools から Programmer を選ぶと図 6.3 のようなウィンドウが開くので，FPGA に書き込みたいコンフィギュレーションファイル (`de0cv-lab2.sof`) を選択し，Program/Configure にチェックを入れ，Start ボタンを押して FPGA へ回路データを書き込む．書き込みが成功すれば，ウインドウ中の Progress 部分に Successful と表示される．

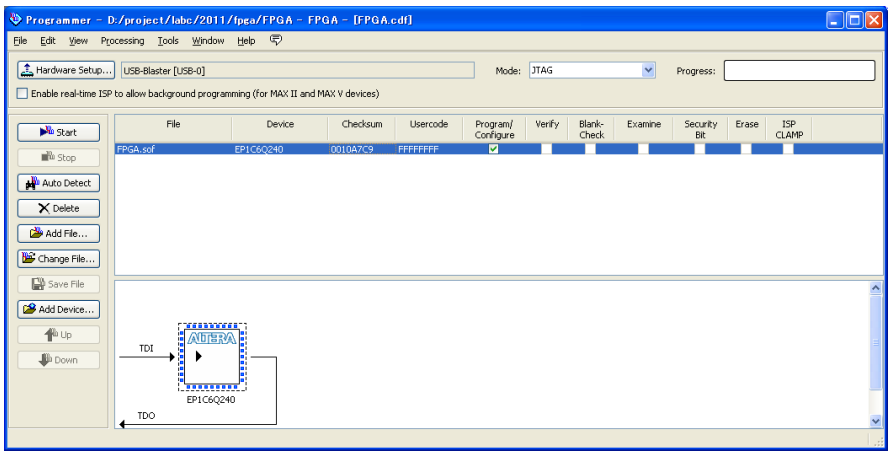


Figure 6.3: Programmer ウインドウ

実行中の ROM，RAM の内容の確認

Programmer で Successful となり，FPGA ボード上でプログラムが実行されている時に，Tools→In-System Memory Content Editor を起動することで，実行中の ROM や RAM の内容を確認することが可能となる (図 6.4 参照)．起動すると，図 6.5(1) のようなウインドウが表示される．Instance ID の列にある rom を選択し，右クリックすると図 6.5(2) のようになり，そこで，Read Data from In-System Memory を選択すると，その時点での内容を取得し，図 6.5(3) の様に rom の内容を表示する．図 6.5(4) の様に，ram を選択すると ram の内容が取得できる．その際に，Continuously Read Data from In-System Memory を選択すると，その間は，常時読み込みを行う様になり，プログラムの内容に沿って ram の変更がリアルタイムで変更される．

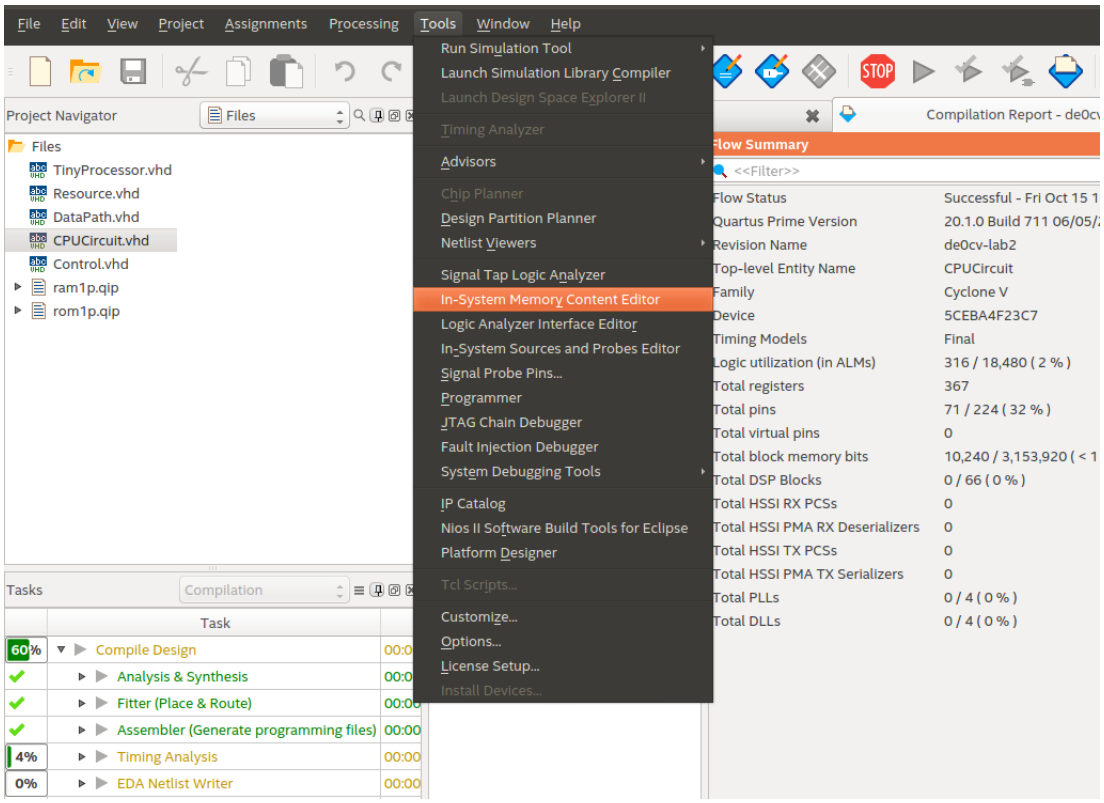


Figure 6.4: In-System Memory Content Editor の起動

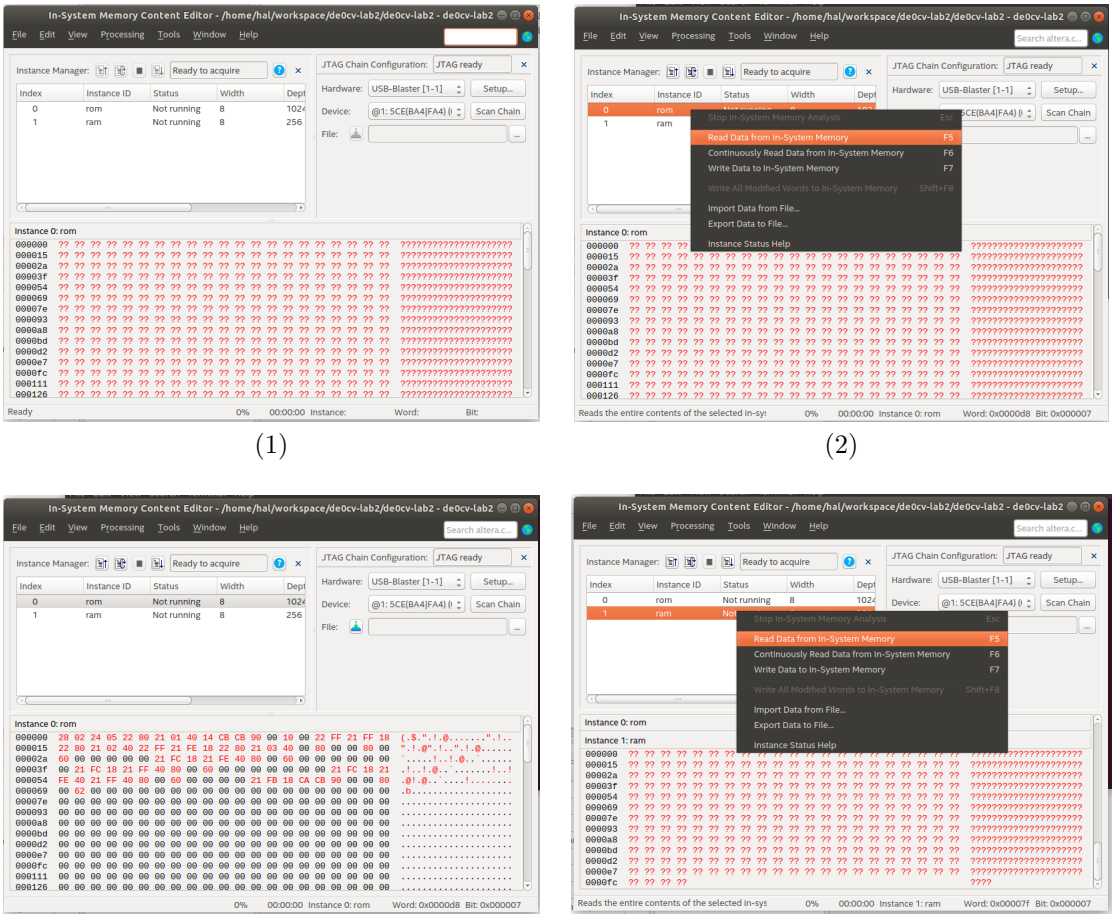


Figure 6.5: In-System Memory Content Editor

課題 10 (12/12)

1. 6.4 節の手順に従って，Tiny-Processor および周辺回路，ROM 一式を FPGA ボードにダウンロードし，`rom.mif` に書かれているプログラムが正しく動作することを確認せよ．(チェックは行わない)
2. 以下の手順により，C-Processor を FPGA ボードに実装し，動作を確認せよ．
   (a) Tiny-Processor 用の FPGA プロジェクトファイルをディレクトリごとコピーして，C-Processor 用の新しいディレクトリを作成せよ．
   (b) CPU 周辺回路 (`CPUCircuit.vhd`) と接続可能なように，実験 1 で作成した C-Processor の記述を変更せよ．(ヒント参照)
   (c) Tiny-Processor 用の `rom.mif` の内容を C-Processor 用の命令コードに書き換えよ．
   (d) Quartus により回路をコンパイルした後，`de0cv-lab2.sof` を FPGA ボードにダウンロードせよ．
   (e) チェックではプログラムが正しく動作することを示せ．

ヒント

- CPU 周辺回路 (`CPUCircuit.vhd`) は CPU の clock，reset，read，write，Address，Dataout，dbgIRout，dbgAout，dbgBout，dbgCarryF，dbgZeroF と接続する．実験 1 で作成した C-Processor の出力ポートには dbgIRout，dbgZeroF，dbgCarryF，dbgAout，dbgBout がないため，そのまま CPU 周辺回路と接続できない．配布した Tiny-Processor 用 VHDL ファイル (`TinyProcessor.vhd` および `DataPath.vhd`) 内の記述・コメント (for debug on FPGA と書かれている部分) を参考にして，C-Processor 用 VHDL ファイルを修正することにより，C-Processor の出力ポートを追加すること．

課題 11 (12/19)

1. FPGA ボード上に実装した C-Processor で，下記の仕様のプログラムを作成し動作させよ．ここに書かれていない仕様は自由に決めて良い．

乗算 (8 bit)

- スイッチを使って 8 bit の数値を設定した後，プッシュボタンを押すことで取り込めば，8 bit の数を 1 つ入力できる．これを使ってまず 8 bit の 2 数 A，B を入力する (負数を含む)．
- その 8 bit の 2 数の積 $A \times B$ を RAM に保存せよ．ここでは，結果が 8 bit を超えるような演算は対象としない．
- 実際に計算させる値は，動作確認時に次の 3 組指定する．(1) 2 番目の数が 0，(2) 両方とも負数，(3) その他の数字．途中でリセットをかけてもよい．
- プッシュボタンは入力ごとに別々のものを使っても，同じものを使ってもよい．

課題 12 (12/26)，課題 13 (1/16), 課題 14 (1/23)

1. C-Processor の命令セットの拡張を行う，次にあげる 4 種類の命令，論理シフト命令，算術シフト命令，移動命令，乗算命令のうち 2 種類以上を追加する．

命令	オペランド	サイズ	コード									動作
論理シフト												
SLL	num	1 word	0	1	1	1	0	0	0	0	70	IP ← IP+2, A ← A << num (C, Z)
SRL	num	1 word	0	1	1	1	0	0	0	1	71	IP ← IP+2, A ← A >> num (C, Z)
算術シフト												
SLA	num	1 word	0	1	1	1	0	0	1	0	72	IP ← IP+2, A ← A << num (C, Z)
SRA	num	1 word	0	1	1	1	0	0	1	1	73	IP ← IP+2, A ← A >> num (C, Z)
データの移動												
MOVAB			0	1	1	1	0	1	0	0	74	IP ← IP+1, B ← A
MOVBA			0	1	1	1	0	1	0	1	75	IP ← IP+1, A ← B
乗算												
MULA			0	1	1	1	0	1	1	0	76	IP ← IP+1, A ← A × B (C, Z)
MULB			0	1	1	1	0	1	1	1	77	IP ← IP+1, B ← A × B (C, Z)

6.5 プログラミングのヒント

以降の課題では，FPGA ボードのメモリ上で動作するような機械語プログラムを作成する．実験 1 で行ったシミュレーション用のメモリイメージのプログラミングと少し異なる点が出てくる．FPGA ボード上にプログラムするにあたって陥りやすい誤りや，デバッグのテクニックについて少しヒントを書いておくので参考にして欲しい．

デバッグ方法のヒント

- バス，レジスタの値の表示

  LED にはアドレスバス (Address)，データバス (DataOut)，IR レジスタの値，A レジスタの値，B レジスタの値，ゼロフラグの値，キャリーフラグの値を表示できる．動作の詳細は 6.3 節を参照のこと．これらの値を確認することで，命令がきちんと順番に行われているか確認できる．また，アドレスバスには，メモリへの読み書きを行う際には IX レジスタの値が，それ以外の場合は IP レジスタの値が表示されるので，ロード・ストアの際の IX 値やジャンプ系命令で IP が変更されていることも確認できる．

- PC 上のシミュレーションでの検証

  まず，ModelSim でシミュレーションしておくと結果的に発見の難しいバグに悩まなくて済む可能性がある CPUCircuit および ROM をテストする簡単なテストベンチを用意しているので，シミュレーションによる検証をしたい場合には実験 C のホームページからテストベンチをダウンロードし，用いること．テストベンチでは，`rom.mif` に記述してある命令がそのままシミュレーションできるが，ボタンを押すタイミングやスイッチの値を決定するタイミングはテストベンチ内で，記述する必要がある点に注意すること．

改変の難点

ModelSim 用のメモリイメージプログラムも行頭に番地をつけなければならないため，かなり面倒だったと思うが，ボードのメモリ用プログラムもアドレスの整合を取るのがかなり面倒になる．すなわち，命令を挿入した場合に以降のアドレスがずれるため，ジャンプ系の命令の飛び先の番地が変わってしまい，その都度直さなくてはならない．これを解消するには以下のような方法を取ると良い．

- まずテキスト形式でしっかりとプログラム全体を記述し，間違いが無いことを確認してから，バイナリ形式に書き下すことで修正回数を最低限に抑える．行き当たりばったりで書かない
- バイナリコードを書く際は，ジャンプ系命令の飛び先 (つまりアセンブリ言語で言うところのラベルが貼られている箇所) の手前に NOP (命令コード 0x00) をいくつかはさんでおく．そうすると命令の挿入に対応しやすい
- 簡易アセンブラを正しく実装できた場合 (拡張課題) は，アセンブリプログラムから簡単にバイナリ形式を得られる

課題 15 (1/30) 拡張課題

FPGA ボード上に実装した C-Processor で，下記の仕様のプログラムを作成し動作させよ．ここに書かれていない仕様は自由に決めて良い

a. 最大公約数を求めるプログラムの作成 (8bit)(難易度：易しい〜難しい)

- スイッチを使って 8 bit の数値を設定した後，プッシュボタンを押すことで取り込めば，8 bit の数を 1 つ入力できる．これを使ってまず 8 bit の 2 数 A，B を入力する (負数を含む).
- その 8 bit の 2 数の最大公約数を RAM に保存せよ．ここでは，結果が 8 bit を超えるような演算は対象としない.
- A と B ともに正の整数とする.
- プッシュボタンは入力ごとに別々のものを使っても，同じものを使ってもよい.

a. 最小公倍数を求めるプログラムの作成 (8bit) (難易度：易しい〜難しい)

FPGA ボード上に実装した C-Processor で，下記の仕様のプログラムを作成し動作させよ．ここに書かれていない仕様は自由に決めて良い.

- スイッチを使って 8 bit の数値を設定した後，プッシュボタンを押すことで取り込めば，8 bit の数を 1 つ入力できる．これを使ってまず 8 bit の 2 数 A，B を入力する (負数を含む).
- その 8 bit の 2 数の最大公約数を RAM に保存せよ．ここでは，結果が 8 bit を超えるような演算は対象としない.
- A と B ともに正の整数とする.
- プッシュボタンは入力ごとに別々のものを使っても，同じものを使ってもよい.

b. 演算器の改善 (難易度：標準〜難しい)

C-Processor の ALU 内の加算器は，桁上げ (Carry) を次の全加算器に順繰りに送っていく実装となっている．これは，波のように桁上げが伝播していくことから，Ripple Carry Adder (RCA) と呼ばれる．しかし，入力のビット幅が大きくなると，RCA では桁上げが伝播する経路がクリティカルパスとなり，低速になってしまう．そこで，桁上げ先見回路を追加した加算器に置き換えることを考える.

1. CLA を VHDL で実装し，演算器を入れ替えよ.
2. FPGA 上で正しく動作していることを確認せよ.

最終レポート

提出方法

2/3 17:00 までに CLE から提出すること．レポートのファイル形式は PDF 形式とし，ファイル名は「学籍番号下 4 桁.pdf」とすること．なお，飛び級希望の学生の場合，最終レポートの締切が早くなる可能性があるため，アナウンスに注意すること．

レポート内容

以下の内容について報告すること．

1. 課題 11 で作成したプログラムの動作仕様について説明せよ．
2. 課題 12，課題 13, 課題 14 で追加した命令について，どのように実装したかを説明せよ．
3. 実験 2 の C-Processor の実装にあたって工夫した点やアピールする点があれば，章立てて報告すること．拡張課題についても同様に報告せよ．
4. 拡張課題（課題 15）で作成したプログラムの動作仕様について説明せよ．

レポートに必要な情報・データ・画面のスナップショット等は実験最終回までに USB メモリなど手元にコピーするなどしておくこと．レポート作成のために時間外に実験室を開けることはしない．また，レポートのコピーが発覚した場合は，提供者も含めて即不合格にする場合がある．

Appendix A

計算機システムとハードウェア

本章では,「計算機アーキテクチャ」や「論理設計」等の今までの講義から，CPU 制作に必要な基礎知識の復習を行う．実験 C の第 2 回目の授業までに一通り読んでおくこと．ここでは詳細には触れないので，過去の授業の教科書や資料等の該当箇所も復習しておくとなお良い．また，授業内容と関連のある背景知識についても少し触れておく．

A.1　プロセッサと命令セット

計算機は**機械語命令**によって **CPU** (Central Processing Unit: 中央演算装置) に指示を与え，計算を行わせることで動作する．機械語命令とは命令本体を表す部分 (命令コード) と命令の引数を表す部分 (オペランド) からなるビット列である．この命令を並べたものが**プログラム**になる．ストアドプログラム方式の計算機では，プログラムはメモリ上に置かれ，逐次読み込まれて実行されていく．

機械語命令は人間にとっては理解しづらく，直観的に動作を把握しづらい．これを解消するために，まず開発されたのがアセンブリ言語である．アセンブリ言語は，機械語命令をその動作を表す略語 (ニモニック) で置き換えることによって，人間にとってわかりやすいものにしている．しかし，プログラムが長大になると，アセンブリ言語では機能の把握が難しくなる．また，アセンブラ命令は基本的には機械語命令と 1 対 1 で対応するため，計算機が変わると命令セットは全く違うものになってしまう．つまり移植性が乏しい．このため，より自然言語に近い，C 言語のような高級言語が開発された．高級言語で書かれたプログラムを機械語で書かれたプログラムに翻訳する作業がコンパイルである．

実験 C で実装するプロセッサは簡易な **RISC** (Reduced Instruction Set Computer) プロセッサである．RISC とは命令の数を極力減らして，回路を単純化することによって高速化を測る設計思想のことを言う．例えば，実験 C 課題中にもあるが，積の計算はそれに特化した命令を持つこと無しに，和の演算をプログラムで繰り返し行うことによって実現する．そのため設計中には積算器は含まれていない．

A.2　順序回路

順序回路とは，状態を記憶することのできる回路のことであり，図 A.1 に示すように**組合せ回路**と**記憶 (レジスタ・メモリ)** によって構成される．CPU も順序回路の 1 つである．

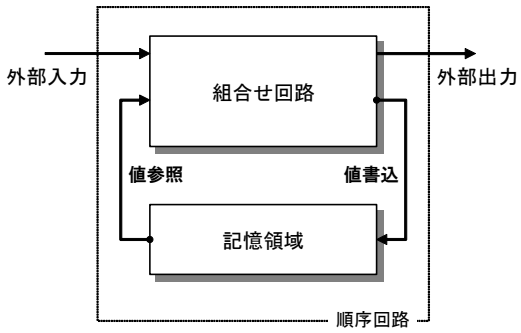


Figure A.1: 順序回路

組合せ回路とは，入力が変化すると回路内部の論理ゲートを値が伝播して，出力が変化するような回路であ

る．入力が変化した後，出力が変化するまでには，一般には複数の論理ゲートを通過する必要があり，**遅延**が発生する．記憶とは，値を保存できる機構のことで，レジスタやメモリがそれに該当する．

順序回路では，組合せ回路で計算した値を記憶領域に保存し，またその記憶の値を計算に利用する．値を記憶することが可能なので，過去の動作履歴を憶えておくことができる．つまり，回路は記憶領域の**状態**を遷移させることで動作する．有限状態機械を回路として実装すると順序回路になる．

組合せ回路において，出力に至るまでに通過するゲートの数や種類は異なるため，各出力の値が更新されるタイミングはバラバラである．そのため，レジスタに値を記憶するタイミングもバラバラになってしまい，データの一貫性を保つのに不都合が生じる．典型的な順序回路では**クロック**を用いてデータ取り込みタイミングを同期し，この問題を解消している．同期式順序回路においては，毎クロックの立ち上がり (あるいは立ち下がり) のタイミングで，記憶素子は値を取り込む．このため，極端に遅延が大きい経路 (**クリティカル・パス**) が存在すると，クロックはその遅延に合わせて周波数を低く (つまり，クロックの頻度をゆっくりに) 設定しなければならなくなる．クリティカル・パスの遅延をいかに小さくするかが順序回路の高速動作化の要点となる．

A.3 パイプライン

クリティカル・パスを分割して，クロック周波数を上げるための一般的な方法として，**パイプライン**化がよく用いられる．パイプラインは「長い処理を 1 クロックで行うよりも複数クロックに分割し，1 クロック間での処理を短く揃えよう」という考えに基づいている．

図 A.2 にパイプライン化の模式図を示す．例では，長いクリティカル・パスを持つ組合せ回路 C を，3 つの

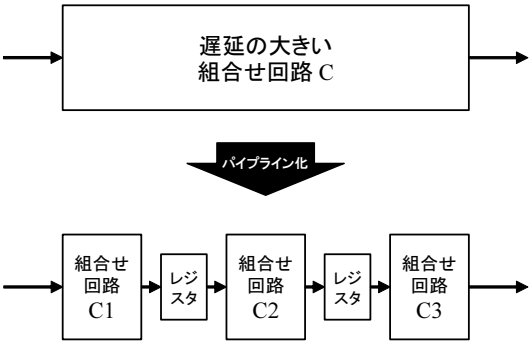


Figure A.2: パイプライン化

回路 C1, C2, C3 に分割し，各クロックごとに値をレジスタに保存して次の回路に渡している．この例におけるパイプライン化によるタスクの処理過程の時間変化を図 A.3 に示す．縦の点線がクロックの同期タイミングだと思って欲しい．もし，組合せ回路 C の遅延を等分して回路 C1, C2, C3 に分割できたとする．一見するとクロッ

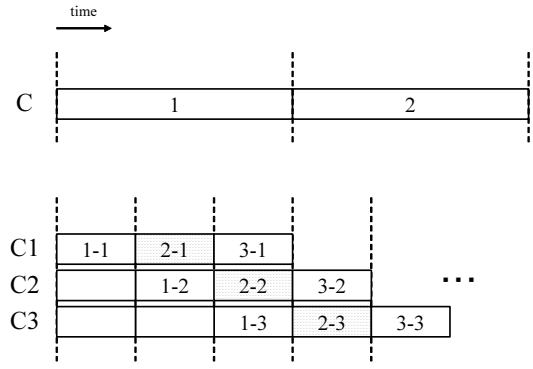


Figure A.3: パイプラインのタスク処理過程

ク速度は 3 倍になったが，1 つのタスクに 3 クロックかかるため，クロック速度は速くなっても合計の処理速度は変わっていないように思える．しかし，パイプライン方式ではその名の通り，次々に入力を入れることができる．つまり，最初のタスクが C1 を終えて，C2 で処理されている間に C1 で次のタスクを処理することができる．単位時間当たりの処理量である**スループット**を考えてみると違いがよくわかるだろう．このようにして処理を高速化することが可能なパイプラインだが，手放しにどんどん細切れにすれば速くなるというわけではない．

処理を分割することで，**ハザード**と呼ばれる不都合が生じ，入力を一旦停止 (**ストール**) しなければならなくなるためである．ハザードやストールについては実験 3 で演習および考察してもらう．

A.4 有限状態機械

有限状態機械 (Finite State Machine: FSM) とは，状態の遷移によって (システムの) 動作を表すモデルである．FSM をハードウェアで実装する場合，状態をレジスタに記憶させ，次クロックにおけるレジスタの値を現在のレジスタの値と入力信号の値から計算する．その際，FSM 上での 1 回の遷移は，ハードウェアにおける 1 クロックサイクルに対応付けられる．これはまさに順序回路である．そのため，制御系等の順序回路を作成する場合には，FSM によって仕様モデリングを行ってからハードウェア実装することが多い．UML 等の仕様分析手法にも状態遷移モデルを記述するステートチャートが採用されている．

図 A.4 にストップ付 8 進カウンタの FSM の例を示す．このカウンタでは入力である stop 信号が 0 のときはカウントアップし，stop 信号を 1 にすることによってカウントアップを停止する．このように現状態と入力によって次状態が決定される FSM はミーリ (Mealy) 型と呼ばれる．

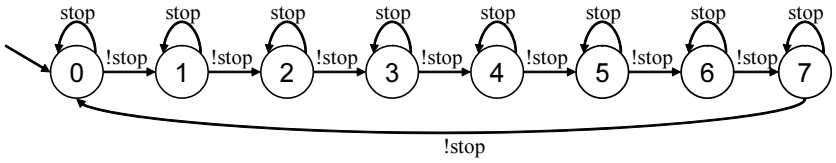


Figure A.4: ストップ付カウンタの FSM

A.5 論理合成と HDL

その昔は回路設計の大半を論理ゲート (AND・OR などの) と配線で記述していた．つまり，グラフィックエディタなどでゲートとそれらの間のワイヤを書くことで設計を行っていた．しかし，現在ではチップに集積可能な回路規模が増大しており，市販の LSI では 1 チップに数十〜数百万ゲートが含まれるため，これらをマニュアルで設計することは至難となる．そこで，開発されたのが**論理合成**技術である．

論理合成とは，ハードウェア記述言語 (Hardware Description Language: **HDL**) で書かれた記述を論理ゲート回路に変換する技術のことを言う．HDL では，レジスタとその接続部分の論理のみを記述すればよく，ゲート回路よりも人間が理解しやすい形でハードウェアを設計できる．また，再利用性も高まる．このレジスタと接続の論理を記述する記述抽象度のことをレジスタ転送レベル (Register Transfer Level: **RTL**) と呼ぶ．

HDL においては **VHDL** と **Verilog** の 2 種類が主たる言語となっており，市場流通している設計によく用いられている．実験 C では，VHDL を用いて CPU の RTL 設計を行う．

A.6 ハードウェア設計の流れ

ハードウェア (FPGA に実装する場合) は以下のような工程を経て製作される．

1. 仕様より HDL 記述を作成
2. 記述した HDL 記述を**シミュレーション**により検証・デバッグ
3. 検証済 HDL 記述が完成
4. 完成した HDL 記述を論理合成
5. 論理合成された回路データを FPGA にダウンロード
6. FPGA 上でプログラムを動かす

この工程をソフトウェア設計と比較してみる．2. の工程は，プログラムを書き，それに適当な入力を与えて走らせてバグを取る段階にあたるだろう．また，4. はプログラムから実行ファイルを生成するためのコンパイルに対応すると考えればよい (厳密には少し違う)．

シミュレーションや論理合成を行ってくれるツールの事を CAD (Computer-Aided Design) ツールと呼ぶ．一般的には CAD といえば，製図などを計算機上で行うツールのことを指すが，計算機科学分野においては，設計自動化に貢献するツールのことを CAD ツールと呼ぶ．実験 C では主に Mentor Graphics 社のツールを用い

る．2. の工程 (シミュレーション) で使用するツールが ModelSim，4. の工程 (論理合成) で使用するツールが Precision というツールである．ModelSim には “Student Edition” と呼ばれる無償版もあるので，各自でもダウンロードして利用できる．

A.7 FPGA

ハードウェアは，最終的にチップの形で製品として流通するのが一般的である．設計をチップにするには，論理合成の後，配置配線を行い，フォトマスクを製造し，そのマスクを用いてシリコン上へイオン打ち込みを行って半導体を形成するという過程を経る．プロセス技術が進歩して，配線を微細に行うことができるようになった反面，フォトマスク製造も難しくなり，複雑な半導体製品のフォトマスク製造費用は数千万〜数億円にのぼると言われている．

チップ形式でのハードウェア実装は大規模な市場流通に向く．なぜなら，初期コストは高価だが，大量生産で製品単価は安く抑えられるためである．しかし，その初期コストは莫大であり，企業単位でなければ気軽に利用できるわけではない．また，チップ実装すると後戻りできないため，試しに実機を動かしてみるというようなテストはできない．

それに対して，**FPGA** (Field Programmable Gate Array) と呼ばれるデバイスが近年急速に利用されるようになった．FPGA は内部の書換え可能デバイスに回路をダウンロードして動作させる．もちろん，チップ実装よりも動作速度や取り扱える回路規模の面で劣るが，FPGA で回路を動かすことで早期に実機でのテストを行うことができる (プロトタイピングと呼ばれる)．さらに，もともとそれほど速度や回路面積の要求が厳しくないシステムの場合や，素早く実装したり頻繁にアップデートや設定を変えたい性質の機器 (ネットワーク系の機器に多い) では，チップ実装でなく FPGA で実装が最適な場面もある．本実験のような学習用途にも向いている．

図 A.5 に，FPGA の構成要素であるロジックセルの概念図を示す．基本的なロジックセルは任意の組合せ回

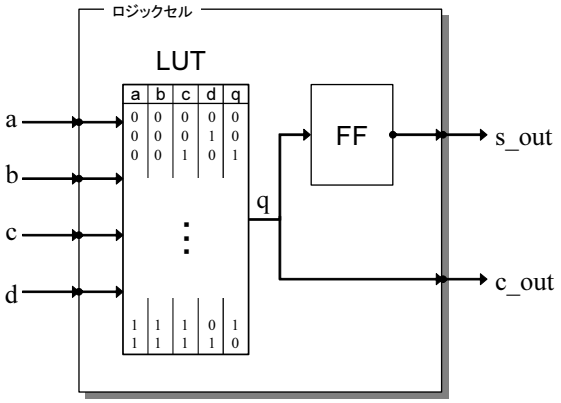


Figure A.5: FPGA のロジックセル

路を実現できる LUT (Look-Up Table) と，記憶素子であるフリップフロップ (FF) からなる．LUT はロジックセルの組合せ回路部分にあたる．この例では，入力 a, b, c, d のパターンによって，出力 q を 0 か 1 のどちらにするかを表として憶えていると考えればよい．この表を書き換えて，ロジックセル間の配線を変えることで任意の組合せ回路を実現可能である．4 入力 1 出力の LUT が用いられることが多い．また，FF によって値を記憶できるため，順序回路も動作させることができる．FF も配線次第で多ビットレジスタにすることが可能であり，任意のレジスタが実現可能である．

FPGA 内部にはロジックセルが格子状に敷き詰められ互いに配線されている．配線も変更することが可能なので，比較的大規模な回路も実装できる．この構造を基本とし，演算器 (加算・積算) や RAM を付け加えたり，LUT を 6・8 入力にしたりとデバイスメーカー各社が拡張を施して製造・販売している．

Appendix B

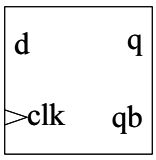
同期式回路の基本部品

本章では，同期式回路で用いる基本部品の説明を行う．実験 C の第 2 回目の授業までに一通り読んでおくこと．これらの部品の HDL 記述については，Tiny-Processor の `Resource.vhd` に書かれているので，必要に応じて参照すること．

B.1 D-フリップフロップ (D-FF)

順序回路の構成部品は，ゲート (組み合わせ回路) とフリップフロップ (FF) からなる．フリップフロップは，SR-FF，JK-FF，D-FF，T-FF 等があり，レベルトリガ式，立ち上がり (下がり) エッジトリガ式等多くの種類がある．

本実験ではフリップフロップとして，ホールドタイム (ラッチ時間) 0 の立ち上がりエッジトリガ式の D-FF を仮定する．D-FF のシンボルと動作表を図 B.1 に示す．clk が L から H に変化したときのみ，q の値が D の値に変化し，それ以外では q の値は変化しない．qb は常に q の否定を出力する．なお，図 B.1 ではリセットポートを省略している．



clk	d	q	qb
↑	H	H	L
↑	L	L	H
-	-	q	qb

Figure B.1: DFF

順序回路の状態は，フリップフロップの出力 q の値 (H または L) の組として表すことができる．n 個の状態を表すには最低 $log_2 n$ 個の D-FF が必要である．

B.2 1 bit レジスタ

図 B.2 に，D-FF を用いた 1 bit レジスタを示す．この構成の 1 bit レジスタは，load 信号が enable でクロックが立ち上がったときに，d の値を D-FF に読み込む．また，$load$ 信号が H でないときも，内部の D-FF は前の値 q を用いて，クロックが立ち上がるたびに値を更新する．

一般に，n bit レジスタは 1 bit レジスタを n 個結合させることで実現する．

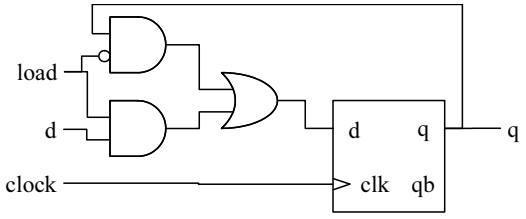


Figure B.2: 1 bit レジスタ

B.3　カウンタ

一つの処理を数クロックかけて行う回路は，自分自身が今どの状態にあるのかを把握 (記憶) する必要がある．これらの回路は，一般には何らかのカウンタ回路を搭載し，処理開始時に 0 に設定後，クロックが入力されるたびにカウントアップ，カウンタ回路から出力される値によって自分の現在の状態を把握する．プロセッサでは，カウンタ回路から出力される値を見て，現在どの状態にあり，どのような制御信号を内部にあるいは外部に出力し，次の状態へ遷移したらよいかを判断する．図 B.3 は D-FF を用いた一般的なカウンタ回路である．図 B.3 のカウンタ回路は，クロックが入力される度に，出力 $q_3q_2q_1q_0$ が，LLLL → LLLH → LLHL → LLHH → LHLL → LHLH → LHHL → LHHH → HLLL → $\cdots$ と変化する．

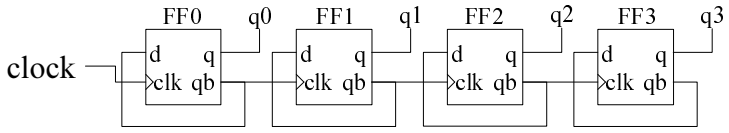


Figure B.3: 4 bit カウンタ

現在の回路の状態を記憶するためには，図 B.3 のようなカウンタ回路を使用すればよいが，クロックに同期しないで動作する部分回路を持つような回路の場合，図 B.3 のカウンタではハザードにより回路が誤動作する可能性がある．例えば，クロック単位では LLLH → LLHL へ状態が変化する場合であっても，実際のカウンタ回路の出力は，LLLH → (LLLL) → LLHL と変化する[1]．もし，カウンタの出力が LLLL のときのみに非同期メモリの書き込み信号が enable になるようなシステムの場合，カウンタの出力が LLLH → (LLLL) → LLHL と変化することで，メモリ書き込み信号が一瞬 enable になり (これがハザード)，意図しないところでメモリへの書き込み動作を開始してしまう．なお，メモリがクロックと同期して動作している場合は，ハザードが発生しても，次のクロックが立ち上がるまでには信号が安定しているので，正常に動作する (不正な書き込みは起きない)．

周辺機器も含めて単一のクロックで同期して動作するシステムはほとんど無く，通常は非同期の周辺機器や周波数が異なっている周辺機器が接続されている．これらの部分とのデータの受け渡しを制御をする場合は，該当する信号線にハザードが発生しないようプロセッサを設計する必要がある．この場合，図 B.3 のカウンタは使用できない．本実験では，次に述べるジョンソンカウンタを用いることで，ハザードを回避する．

B.4　ジョンソンカウンタ

ジョンソンカウンタは，ハザードが発生しない外部信号の生成によく用いられる．ジョンソンカウンタは，出力がハミング距離 1 で遷移するカウンタである．4 bit ジョンソンカウンタ (図 B.4) の出力 $q_3q_2q_1q_0$ は LLLL → LLLH → LLHH → LHHH → HHHH → HHHL → HHLL → HLLL → LLLL → $\cdots$ と変化する．ジョンソンカウンタはハミング距離 1 で遷移するため，状態遷移の途中でハザードが発生しない．ジョンソンカウンタは n 個の FF で，$2n$ 個の有効な状態 (偶数個であることに注意) を表すことができる．

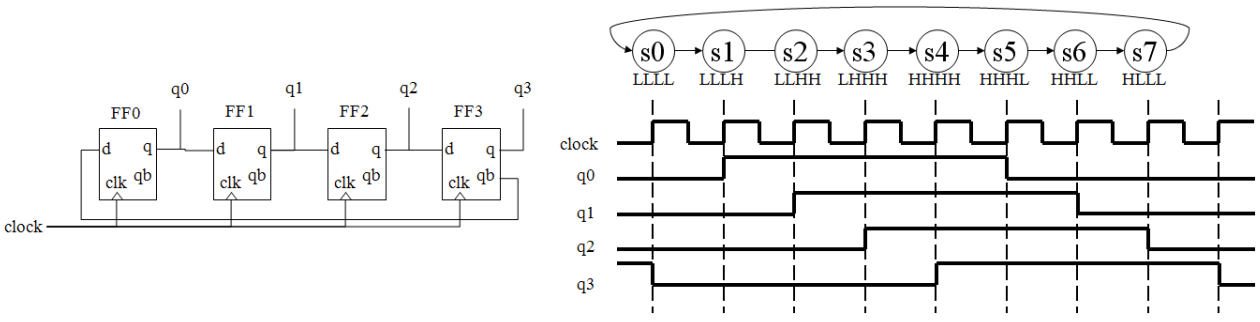


Figure B.4: 4 bit ジョンソンカウンタ

図 B.4 に示したカウンタ回路をそのまま状態の記憶に用いる場合，1 クロック毎に次の状態へ変化して行うため，「ある信号が H になったら次の状態へ進む」というような自己ループを含む状態遷移を実現することは難

[1] FF0 が q=H の状態でクロックが立ち上がると，FF0 の状態が q=L となる．このときカウンタ回路からの出力が一瞬 LLLL となる．その後，FF1 が FF0 の qb=H を読み込むことで FF1 の状態が q=H となり，出力が LLHL で安定する．

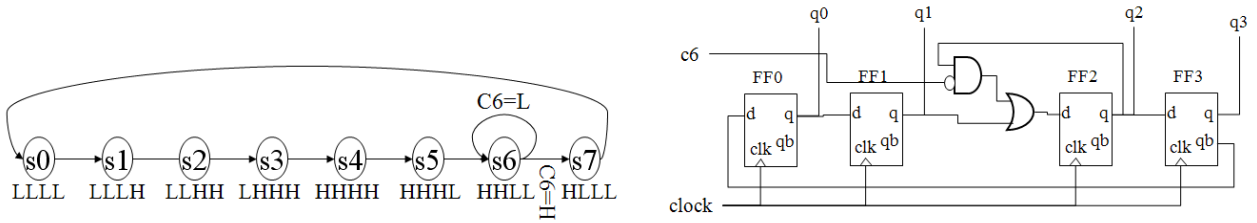


Figure B.5: 停止状態を持つ 4 bit ジョンソンカウンタと回路

しい．そこで，ジョンソンカウンタを構成しているフリップフロップの入力側に回路を追加して，ある条件になるまでカウンタを停止させることで，自己ループを持つような状態遷移も扱えるようにする．

図 B.5 左側の状態遷移は，状態 s6 で信号 $C6$ が L のときは自己ループを回って再び状態 s6 へ，$C6$ が H のときはループから脱出し状態 s7 へ遷移する状態遷移である．これを例に，ジョンソンカウンタによる自己ループを含む状態遷移の実現法を説明する．この例で，状態 s6，s7 のジョンソンカウンタの出力 $q_3q_2q_1q_0$ はそれぞれ HHLL，HLLL である．したがって，状態 s6 で信号 $C6$ が H のときのみ FF2 の出力 $q2$ が L になればよい．逆に言うと，$C6$ が L のとき FF2 の出力 $q2$ を H のままにしておくことで，ジョンソンカウンタを停止し，状態 s7 へ遷移しないように制御できる．そこで FF2 の手前に「状態 s6 で $C6$ が L のときに H を出力し，FF2 の d に与える回路」を挿入すれば良い．状態が s6 であるかどうかは，$q1$ と $q2$ の値がそれぞれ L と H になっているかどうかで判断ができる．FF2 の手前に挿入すべき回路の真理値表は表 B.1 のようになり，論理式で書くと，$d2 = q1 \vee (q2 \wedge \neg c6)$ となる．図 B.5 右側に挿入された回路を示す．

Table B.1: 挿入する回路の真理値表

q1	q2	c6	d2
L	L	L	L
L	L	H	L
L	H	L	H
L	H	H	L

q1	q2	c6	d2
H	L	L	H
H	L	H	H
H	H	L	H
H	H	H	H

B.5 レジスタ間のデータ転送タイミング

図 B.6 上のデータパスにおけるレジスタ間データ転送のタイミングチャートを図 B.6 下に示す．この例ではレジスタ A，B が加算器に接続されており，加算器の出力は，レジスタ A の出力とともにマルチプレクサに接続されている．このマルチプレクサは，信号線 $select$ の値が L のときレジスタ A の値を出力し，$select$ の値が H のときは加算器の値を出力する．

ここで， A ← A+B というレジスタ転送を考える．レジスタ A への A+B の値のデータ転送は図 B.6 左下のように，1 クロックで行える．$LoadA$ が H になった次のクロックの立ち上がりで，レジスタ A に A+B の値が取り込まれる．図 B.6 下を見ると，データ取り込みの直後に $select$ の値が L に変化するので，MUX の出力値が変化して誤動作してしまうのではないかと思うかもしれない．しかし実際には，クロックが立ち上がってから，$select$ や $LoadA$ が変化するまでには必ず遅延があるので，MUX の出力値が変化したときには，すでにデータの読み込みは終了しており (図 B.6 下右)，値を正常に取り込むことができる．ここで，T_{su} は，FF のセットアップタイムと呼ばれ，クロックが立ち上がる最低 T_{su} 前までに，入力データ (この例では MUX の出力値) が確定している必要がある．一方，T_{hold} は FF のホールドタイムと呼ばれ，クロック立ち上がり後，入力データを保持しなければならない時間を表す．実験で使う D-FF については，ホールドタイムは 0 である (よって，仮に FF の出力遅延時間が 0 であったとしても，図のように制御信号を変化させれば値を取り込むことができる)．

以上より，レジスタ間でのデータ転送は 1 クロックで行えると考えてよい．またレジスタ制御信号にハザードが出たとしても，クロックの立ち上がり時までに安定しているので，プロセッサ内部にあるレジスタの読み込み信号に関してはハザードの問題は一切考慮しなくてよい．

B.6 同期式順序回路としてのプロセッサ

ここでは，プロセッサをどのように同期式回路としてとらえられるか，について説明する．一般にプロセッサは図 B.7 に示すように，データパス部と制御部に分けることができる．制御部はデータパス部の (一部の) レジ

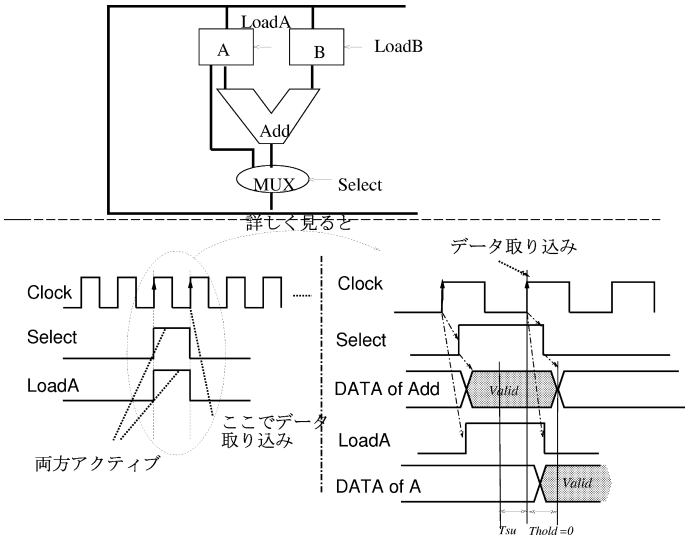


Figure B.6: 転送タイミング

スタ値を参考にして状態を変化させ，各状態で適切な信号値をデータパス側に与えることで，値を読みだしたり，演算したり，格納したりして，プロセッサ全体を動作させる．

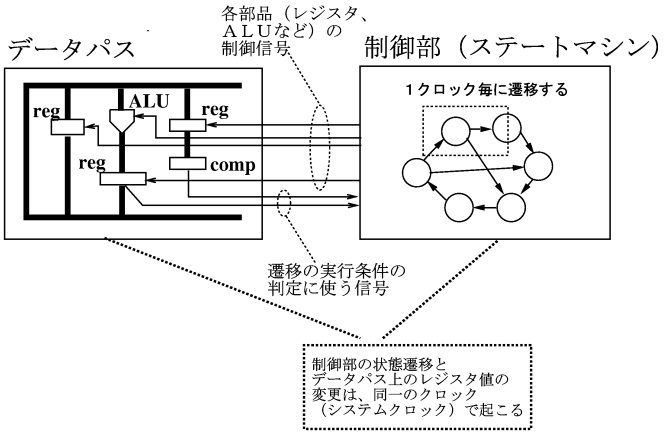


Figure B.7: プロセッサの内部構造

クロックの立ち上がり直後から，次の立ち上がりの直前までが，制御部の各状態に対応する．制御部がどの状態にあるかは，制御部内の状態レジスタ (カウンタ回路) の値により判別する．クロックが立ち上がる度に状態の遷移が起きる．制御部内の状態レジスタ，およびデータパス上のデータレジスタの値は，いずれもクロックの立ち上がりで設定される (値が変化するかどうかには関係なく，クロックの立ち上がり毎に必ず読み込まれる)．各 FF のデータ入力 (D 入力) は，組合せ論理回路の各部の遅延時間の差などにより，クロック立ち上がり直後は不安定になり，しばらくするすると安定する．その安定した値を，次のクロック立ち上がりで取込む．

以上が 1 クロックで同期式順序回路が行う動作である．いずれのレジスタについても，一旦クロックが立ち上がって値が設定される (読み込まれる) と，次のクロック立ち上がりまでは絶対に値が変わらない．また 1 状態中のレジスタ更新時に，複数のレジスタに対して値を変更することができる (1 回の遷移実行で，並列にレジスタ間の演算・データ転送を行える)．もちろんレジスタ間のデータ交換も，データバスの競合がない限りは，ソフトウェアのようにデータの一時待避用レジスタを使うことなく，1 クロックで行える．実際のハードウェアの設計では，可能な限り並列に演算・データ転送を行って高速化を図っている．

Appendix C

VHDLの簡易ガイド

ここではVHDLの書き方について，ポイントだけごくごく簡単に紹介する．詳細は，参考書やウェブを参照すること．

C.1 VHDL記述の構成

VHDL記述の一般的な構成は以下のようになる．

```
パッケージ呼び出し部

- - エンティティ記述部
entity エンティティ名 is
        ポート宣言
end エンティティ名;

- - アーキテクチャ記述部
architecture アーキテクチャ名 of エンティティ名 is
        内部信号・内部モジュール宣言
begin
        回路本体
end アーキテクチャ名;
```

Figure C.1: VHDLの全体構成

C.2 パッケージ呼び出し

本来は特殊なパッケージをインクルード際に記述する．現段階では，とりあえずおまじないと思って以下の記述があればよい．

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Figure C.2: パッケージ呼び出しの定型記述

C.3 エンティティ記述

まず始めに回路モジュールのインターフェース部分の宣言を行う．主にそのモジュールの入出力ポートを記述する．例として，図 C.3 に「8 bit 2入力／1出力マルチプレクサ」のエンティティ記述を挙げる．ここは回

路の内容を書いているわけではないことに注意．あくまで外から見てこのモジュール (エンティティ) がどのような入出力を持っているのかだけを書くところである．(図 C.4)

```
entity Mux2x08 is
    port (
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        sel : in std_logic;
        q : out std_logic_vector(7 downto 0)
    );
end Mux2x08;
```

Figure C.3: 8bit 2-1 マルチプレクサのエンティティ記述

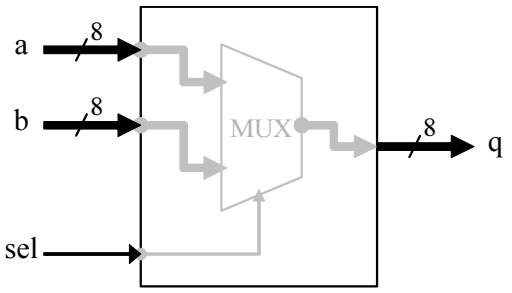


Figure C.4: エンティティ記述のイメージ

port ブロック中には各入出力信号の名前，**in**(入力)/**out**(出力)，信号の幅 (ビット数) を記述する．1 bit の信号の場合は “**std_logic**”，$n+1$ ビットの場合は “**std_logic_vector**(n **downto** 0)” と書く．“0 **to** 7” と書かずに (書いても良いのだが)，“7 **downto** 0” と書くのは「上位ビットが左」という意識のハードウェア設計の慣例による．

C.4 アーキテクチャ記述

次に，回路の中身であるアーキテクチャ記述を書く．ここでは，そのモジュールがどのような機能を持つか，具体的な論理を記述する．例として，図 C.5 に「8 bit 2 入力／1 出力マルチプレクサ」のアーキテクチャ記述を挙げる．

```
architecture logic of Mux2x08 is
begin
    q <= a when sel = '0' else
         b ;
end logic;
```

Figure C.5: 8bit 2-1 マルチプレクサのエンティティ記述

この記述は,「sel の値が 0 の場合は，a の値を q に入れ，それ以外の場合 (sel が 1 の場合) は b の値を q に入れよ」ということになる．ここでのポイントは以下の 2 点.

1. 代入記号が ‘<=’ である

   ‘=’ ではないことに注意．逆に条件比較は ‘==’ でなく ‘=’ である.

2. セレクタ (条件記述) であるが，if を使っていない

   “**if** ($sel = 0$) q <= a **else** q <= b” とソフトウェアプログラム的に書きたくなるが，これは許されない．**if** を使った記述は「記述を上から下へ実行する」的な書き方 (手続き的) であり，**when** を使った書き方

は「q の値をどうするか」つまり「q という信号に何を接続するか (もしくはレジスタに何を代入するか)」に着目した書き方 (宣言的) である．後者の方がハードウェアをイメージしやすいといえる．

ちなみに，**if** 文は **process** ブロックでのみ記述が許されているが，実験 C においては，FF とテストベンチ以外では process ブロックを使わない．よって，通常のソフトウェア・プログラミングのような **if** を使った条件分岐は基本的に書かない．

さらに，**when** を使った信号代入のみを使えば，**1 つの信号 (レジスタ) に関する代入文は 1 回しか現れない**．同じ信号に関する代入を複数箇所に書くことは，**process** ブロック以外では許されない．

内部信号

モジュール内部で使用する信号は，**begin** の前で宣言する．内部信号宣言の例を図 C.6 に挙げる．

```
signal result_adder : std_logic_vector(7 downto 0);
```

Figure C.6: 内部信号宣言の記述

内部信号の宣言には，先頭に **signal** というキーワードをつけて宣言する．その他はエンティティ宣言のポート宣言と同様である．

演算子

他にも表 C.1 のような演算子が用意されている．

Table C.1: VHDL の演算子

関係演算子		ビット演算子	
=	等しい	and	論理積
		or	論理和
		xor	排他的論理和
		nand	否定論理積
		nor	否定論理和

大小比較の演算子も用意されているが，実験 C では特に使う必要はないだろう．算術演算子ももちろん用意されている．しかし，実験 C では算術演算子は使用しないことにする．なぜなら，算術演算は論理合成によって演算回路に変換されるため，回路規模が増大し，遅延も増すためである．そもそも，そうならないよう実験 C では ALU を使った設計になっている．手軽に書けるからといって，高品質の回路ができるかというとそうではないところが，RTL と論理合成を用いた設計の難しいところである．

C.5 モジュールの呼び出し (インスタンス化)

エンティティおよびアーキテクチャ記述によって書かれたモジュールは，呼び出してインスタンス化することで実際に使用する．つまり，トップモジュール以外のモジュールは必ず呼び出してインスタンス化することになる．例として，図 C.7 に 8 bit 加算器をインスタンス化する記述を示す．(ALU の設計記述から抜粋)

まず，**begin** より前の内部信号を宣言する箇所に，**component** 宣言を書く．これは呼び出す下位モジュール (例の場合は RCAdder08) のエンティティ宣言にあたる．つまり，入出力のインターフェースの名前宣言にあたる．そして，実際に **begin** 以降でモジュールを呼び出す．ここでは “adder” という名前を与えてインスタンス化している．呼び出したモジュールの入出力は，呼び出し元の信号と信号代入文によって結線する (例の記述では内部信号の宣言は省く)． これでモジュールをインスタンス化し，そのモジュールに入力を与え，出力を引き出すことができる．インスタンス化のイメージ図を図 C.8) に示す．

C.6 Tips

- コメントは ‘- -’(ハイフン 2 つ) から行末まで
- 1 ファイルに複数のモジュールを記述しても良い

```
architecture logic of ALU08 is
component RCAdder08
    port (
        x : in std_logic_vector(7 downto 0);
        y : in std_logic_vector(7 downto 0);
        cin : in std_logic;
        s : out std_logic_vector(7 downto 0);
        c : out std_logic
    );
end component;

begin
adder : RCAdder08
    port map (
        x => inA,
        y => inB,
        cin => cin_tmp,
        s => result_adder,
        c => cout_tmp
    );
end logic;
```

Figure C.7: 8 bit 加算器のインスタンス化記述

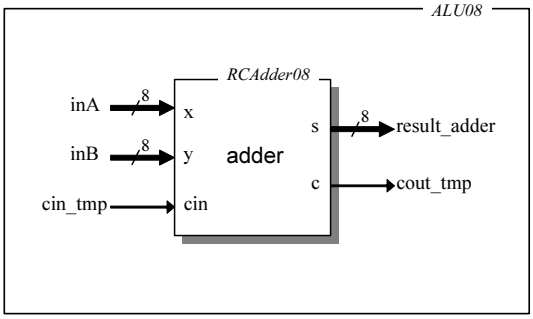


Figure C.8: インスタンス化記述のイメージ

- VHDL ファイルの拡張子は ".vhd" である

C.7 まとめ

HDL でハードウェアを設計する際のポイントは「ハードウェアを意識して書く」こと．ソフトウェアプログラミングのように「処理の流れ」を書くのではなく,「部品とそのつながりを書いている」という意識を持つ必要がある．論理合成ツールは優秀なので，ある程度乱暴なコードでも回路にしてくれるが，ハードウェア設計の意識を持って書いた記述の方が効率の良い回路になる．

もう一度，特に実験 C に限った重要なポイントをまとめる．

- **if** は使わない．**when** を使った代入文を使う
- +, −, ∗ などの算術演算子は使わない
- **entity**, **component** 宣言は，入出力のインターフェース部分を書く．中身は **architecture** 部に書く
- モジュールは呼び出して (インスタンス化して) 使う