シングルサイクル RISC プロセッサの設計

実験概要

本実験では，標準的な**シングルサイクル RISC（Reduced Instruction Set Computer）プロセッサ**の設計を行う．実験を通して，プロセッサの命令セット・アーキテクチャとその実現方式についての理解を深める．また，命令実行中のプロセッサの内部状態を観察することにより，C 言語プログラム中のループや条件分岐，関数呼び出しが，プロセッサの命令実行レベルでどのように処理されるのかについての理解を深める．

実験スケジュール

本実験は全 5 週で，以下のようなスケジュールで行う（第 5 週目は進度に合わせた調整日）．

第 1 週（予習：参考文献 7，8 の 2，3，5 章，本指導書の 1，2，3，5 章） シングルサイクル RISC プロセッサの設計と動作実験を行う．Verilog HDL で記述された，**即値符号なし整数加算命令**（add immediate unsigned：addiu）と**ストア・ワード命令**（store word：sw）が未実装なプロセッサについて，追加設計を行い，両命令が正しく動くプロセッサを完成させる．また，ディスプレイに文字を出力する簡単な機械語のマシン・コードを，追加設計前と後のプロセッサで実際に動作させる．

- 実験 1-1 （マシン・コードの動作実験 1-1 （ディスプレイへの文字出力））
- 実験 1-2 （プロセッサの追加設計 1 （addiu 命令，sw 命令）と動作実験 1-2）

第 2 週（予習：参考文献 7 の 2，3，5 章，本指導書の 4，5 章） プロセッサの設計と動作実験を行う．**ジャンプ命令**（jump：j）と**即値符号なし・セット・オン・レス・ザン命令**（set on less than immediate unsigned：sltiu），**ブランチ・オン・ノット・イコール命令**（branch on not equal：bne），**ロード・ワード命令**（load word：lw）が未実装な，第 1 週に完成させたプロセッサについて，さらに追加設計を行い，これらの命令が正しく動くプロセッサを完成させる．また，C プログラムから本実験をとおして完成させるプロセッサ用のマシン・コードを生成するクロスコンパイルの実験を行う．さらに，ディスプレイに繰り返し文字を出力するマシン・コードと C プログラムを，それぞれ追加設計前と後のプロセッサで実際に動作させる．

- 実験 2-1 （マシン・コードの動作実験 2-1，ディスプレイへの繰り返し文字出力 1）
- 実験 2-2 （追加設計 2（j 命令）と動作実験 2-2）
- 実験 3 （C クロスコンパイラを用いたマシン・コード生成）
- 実験 4-1 （C プログラムの動作実験 4-1，ディスプレイへの繰り返し文字出力 2）
- 実験 4-2 （追加設計 3（sltiu 命令，bne 命令，lw 命令）と C プログラムの動作実験 4-2）

第 3 週（予習：参考文献 7 の 2，3，5 章，本指導書の 5 章） プロセッサの設計と動作実験を行う．**ジャンプ・アンド・リンク命令**（jump and link：jal）と**ジャンプ・レジスタ命令**（jump register：jr）が未実装な，第 2 週に完成させたプロセッサについて，さらに追加設計を行い，両命令が正しく動くプロセッサを完成させる．また，ディスプレイに文字列を出力する関数と，キーボードからの文字列入力を受ける関数を含む C プログラムを，それぞれ追加設計前と後のプロセッサで実際に動作させる．

- 実験 5-1 （C プログラムの動作実験 5-1，関数呼出・ディスプレイへの文字列出力関数）
- 実験 5-2 （追加設計 4（jal 命令）と C プログラムの動作実験 5-2）
- 実験 6-1 （C プログラムの動作実験 6-1，関数からの復帰・キーボードからの文字列入力を受ける関数）
- 実験 6-2 （追加設計 5（jr 命令）と C プログラムの動作実験 6-2）

第 4 週（予習：本指導書の 5 章） 素数計算を行う C プログラムと，ステッピングモータを制御する C プログラムを作成し，第 3 週に完成させたプロセッサで実際に動作させる．

- 実験 7 （C プログラムの動作実験 7，素数計算）
- 実験 8 （C プログラムの動作実験 8，ステッピングモータの制御）

指導書の構成

1 章ではプロセッサの命令セットアーキテクチャについて述べ，2 章では本実験で設計するプロセッサと動作実験用コンピュータについて説明する．3 章，4 章では，第 1 週目，第 2 週目に行うシングルサイクル RISC プロセッサの設計「基礎編」,「中級編」についてそれぞれ説明する．5 章では本実験で実施する実験について，6 章ではレポートについて説明する．

実験の進め方

実験は，2〜3 人 1 組（各班 2 組で構成）で実施する．組ごとに，実験機器を共有しながら，全ての実験を進める．

実験課題目次

目 次

作成者: 中村一博，小尻智子
改訂者: 大野誠寛，松原豊
協力者: 平野靖，北坂孝幸，高田広章，冨山宏之，大下弘，土井富雄，
小川泰弘，濱口毅，出口大輔，村上靖明，後藤正之，柴田誠也，
高瀬英希，鬼頭信貴，大野真司，尾野紀博，小幡耕大，中村悟，
長瀬哲也，北川哲，島崎亮，安藤友樹
最終更新日: 2014 年 9 月 16 日
第 1.20 版

1　はじめに

現代の生活では，多種多様な電子機器が身の回りに存在しており，それら多くの機器にプロセッサ（processor），CPU（中央演算処理装置；Central Processing Unit）が搭載されている．

パソコンやゲーム機，携帯電話のみならず，各種家庭電化製品，音声・画像・映像機器，自動車，航空機，鉄道，船舶，ロボット等においてもディジタル化が進み，制御，データ処理等の用途で CPU は不可欠なものとなってきている．

1.1　コンピュータの標準的な構成

本節では参考文献 [7,8,9] に基づき，標準的なコンピュータの構成について述べる．

コンピュータを構成するすべての構成要素は，図 1 に示される 5 つの古典的な構成要素，**入力**，**出力**，**記憶**，**データパス**，**制御**のいずれかに概念的に分類される．

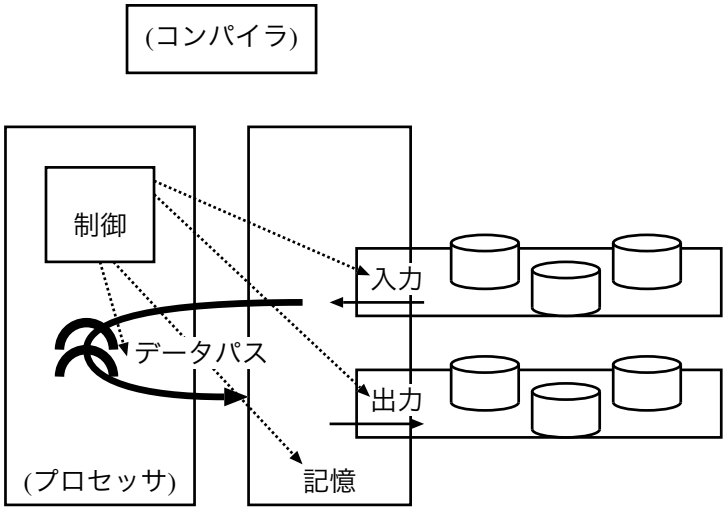


図 1: コンピュータの標準的な構成（参考文献 [7] 図 1.5，参考文献 [9] 図 1.4 より）．

この構成は，コンピュータのハードウェア実現において採用される種々のハードウェア技術に依存しない，現在および過去のほぼすべてのコンピュータに共通する標準的な構成である．ここで**プロセッサ**は，データパスと制御を合わせたものである．プロセッサは，記憶装置から**命令** (instruction) とデータを取り出す．入力装置は，データを記憶装置に書き込む．出力装置は，記憶装置からデータを読み出す．データパスは，プロセッサ内でデータを処理または保持する．制御装置は，データパス，記憶装置，入力装置，出力装置に，動作を指定する制御信号を送る．

プロセッサは，記憶装置から取り出された命令の指示どおりに動作する．その命令はプロセッサが直接理解できる 2 進数（バイナリ）形式の**機械語**（machine language）である必要がある．**コンパイラ**は，人間が理解しやすい**高水準プログラミング言語**（high-level programming language）で書かれた**プログラム**を，機械語をシンボル（記号）で表現する**アセンブリ言語**（assembly language）に変換する．**アセンブラ**（assembler）は，シンボル形式のアセンブリ言語を機械語の命令を連らねたバイナリ形式の**マシン・コード**（machine code）に変換する．

1.2 命令セット・アーキテクチャ

本節では参考文献 [7,8,9] に基づき，命令セット・アーキテクチャについて述べる．プロセッサの言葉である**命令**の語彙を**命令セット**（instruction set）という．人が機械語でプロセッサに適切な指示を出すためには，少なくともプロセッサの命令について理解していなければならない．このプロセッサの命令のような，正しく動作する機械語プログラムを書くためにプログラマが知っていなければならない事柄すべてを要素とする，プロセッサのハードウェアと機械語との間の抽象的なインタフェースを，プロセッサの**命令セット・アーキテクチャ**（instruction set architecture）という．

この抽象的な命令セット・アーキテクチャにより，プロセッサの機能と，その機能を実際に実行するプロセッサのハードウェアを独立に考えることが可能になる．プロセッサのハードウェアは，命令セット・アーキテクチャが論理回路として設計され，集積回路技術によりハードウェアの形で実現されたものである．機械語プログラムは，プロセッサを論理回路レベルで考えるまでもなく，プロセッサの命令，レジスタ，メモリ容量などの命令セット・アーキテクチャに基づいて書くことが可能である．

以下では，多くのメーカーの製品に組み込まれ，広く普及している命令セットの 1 つである MIPS の命令セットを例とし，MIPS 命令セットの主要部分のサブセットについて述べる．具体的には，MIPS の算術論理演算命令，メモリ参照命令，条件分岐命令，ジャンプと手続きサポート用の命令について述べる．

1.2.1 データの格納場所

プロセッサは命令の指示どおりに，データに対する演算や条件判定，データ転送などの処理を行う．処理対象のデータが収められている場所として，(1) プロセッサに直接組み込まれている記憶領域である**レジスタ**（register），(2) メモリ，(3) 命令（定数または即値）の 3 ヶ所がある．また，処理結果のデータが収められる場所として，(1) レジスタ，(2) メモリの 2 ヶ所がある．

(1) レジスタ

レジスタはプロセッサハードウェアの基本構成要素であり，命令セット・アーキテクチャの主要な要素である．レジスタは高速にアクセス可能なデータの一時的な格納場所であり，MIPS には 1 本 32 ビットのレジスタが 32 本ある．レジスタ 1 本のビット幅は 1 **語**（word）と呼ばれ，語は一つの単位として頻繁に用いられる．MIPS では 1 語は 32 ビットである．

MIPS では，算術演算は必ずレジスタを介して行われる．メモリにある演算対象のデータは，演算前にレジスタに移されていなければならない．メモリとレジスタ間でデータを転送する必要があるときには，**データ転送命令**（data transfer instruction）が用いられる．一般に，メモリからレジスタへデータを転送するデータ転送命令は**ロード**（load）命令と呼ばれる．また，レジスタからメモリへデータを転送する命令は**ストア**（store）命令と呼ばれる．

レジスタの表記には，人が書くときの形とプロセッサが読むときの形がある．人が書くときの形はアセンブリ言語，プロセッサが読むときの形は機械語である．表 1 に主要なレジスタのアセンブリ言語と機械語による表記を示す．アセンブリ言語では，シンボル `$s0`，`$s1`，...，`$s7` により，それぞれ 16 番目から 23 番目までのレジスタを表す．また，シンボル `$t0`，`$t1`，...，`$t7` により，それぞれ 8 番目から 15 番目のレジスタを表す．シンボル `$zero` は 0 番目，`$ra` は 31 番目のレジスタを表す．0 番目のレジスタには定数 0 が収められている．31 番目のレジスタは，手続き呼出の戻りアドレスを収めるのに用いられる．機械語では，2 進数でレジスタ番号を書き，それによりバイナリ形式でレジスタを表記する．

表 1: 主なレジスタ

レジスタ番号	アセンブリ言語	機械語	備考
0	`$zero`	`00000`	定数 0
8 から 15	`$t0, $t1, ..., $t7`	`01000` から `01111`	一時変数
16 から 23	`$s0, $s1, ..., $s7`	`10000` から `10111`	一時変数
31	`$ra`	`11111`	戻りアドレス

(2) メモリ

メモリは多くのデータを記憶することができる場所である．MIPS では，メモリはデータ転送命令によってのみアクセスされ，メモリ内の語にアクセスするにはその**アドレス**（address）を指定する必要がある．

アドレスは 0 から始まり，MIPS ではデータ 8 ビット単位，すなち**バイト**（byte）単位でアドレスを表すバイト・アドレス方式が採用されている．MIPS では，1 語が 4 バイトであることから，バイト・アドレス を 0 から 4 刻みに 0, 4, 8, 12, ... と進めていくことにより，順番に並んだ語の第 1 バイト目を指すことができる．例えば，3 番目の語のバイト・アドレスは 8 である．MIPS では，メモリ内の語にアクセスするとき，この 4 の倍数のアドレス，語アドレスが用いられる．

MIPS のデータ転送命令では，**ベース相対アドレッシング**（base addressing）が採用されており，**オフセット**（offset）と**ベース・アドレス**（base address）の和が，アクセスする語のアドレスとなる．オフセットはデータ転送命令中に直接書かれた定数で，プログラムにおける配列をメモリに記憶する際のインデックスに対応する．また，ベースアドレスについては，ベースアドレスを収めた**ベース・レジスタ**（base register）がデータ転送命令中で指定される．ベースアドレスは配列の開始アドレスに対応する．

メモリ・アドレスは，アセンブリ言語ではオフセットとベース・レジスタを並べ「オフセット（ベース・レジスタ)」のように書かれる．例えば，オフセットが `8`，ベース・レジスタが `$t0` の場合，メモリ・アドレスは `8($t0)` と書かれる．機械語では，オフセットとベース・レジスタがそれぞれ 2 進数で表記される．上記の例の場合，オフセットは `0000000000001000`，ベース・レジスタは `01000` と書かれる．

(3) 命令（定数または即値）

定数を命令内に直接書くことにより，定数のメモリからのロードがなくなり，処理が高速になる．命令の処理対象データの在処や処理結果データの格納先を表す**オペランド**（披演算子; operand）の 1 つを定数とした命令を即値命令という．MIPS では，即値の算術演算や論理演算，条件判定命令など即値命令が多数用意されている．

1.2.2 命令セットの概要

MIPS 命令セットの命令を大まかに分類すると，**算術演算命令**，**論理演算命令**，**データ転送命令**，**条件判定命令**，**条件分岐命令**，**ジャンプと手続きサポートのための命令**に分けられる．表 2 に，MIPS 命令セットの主要部分のサブセットをその機能区分ごとに示す．略号はその命令のアセンブリ言語でのシンボル表記である．算術演算命令と論理演算命令は，2 つのレジスタまたは，1 つのレジスタと命令内に収められているデータに対して演算を行い，その結果をレジスタに格納する命令である．データ転送命令は，メモリとレジスタ間でデータを転送する命令である．条件分岐命令は，条件が成立するときに，プログラムの実行の流れを命令内で指示される方へ分岐させる命令である．ジャンプ命令は，無条件に，プログラムの実行の流れを命令内で指示される方へ分岐させる命令である．手続きサポートのための

表 2: MIPS の主要な命令

区分	命令	略号	機能の概要
算術演算	add	`add`	整数加算
	add unsigned	`addu`	符号なし整数加算
	subtract	`sub`	整数減算
	subtract unsigned	`subu`	符号なし整数減算
	shift right arithmetic	`sra`	算術右シフト
	add immediate	`addi`	即値整数加算
	add immediate unsigned	`addiu`	即値符号なし整数加算
論理演算	and	`and`	ビット単位 AND
	or	`or`	ビット単位 OR
	nor	`nor`	ビット単位 NOR
	xor	`xor`	ビット単位 XOR
	shift left logical	`sll`	論理左シフト
	shift right logical	`srl`	論理右シフト
	shift left logical variable	`sllv`	論理左変数シフト
	shift right logical variable	`srlv`	論理右変数シフト
	and immediate	`andi`	即値ビット単位 AND
	or immediate	`ori`	即値ビット単位 OR
	xor immediate	`xori`	即値ビット単位 XOR
データ転送	load word	`lw`	メモリからレジスタへ転送
	store word	`sw`	レジスタからメモリへ転送
	load upper immediate	`lui`	定数をレジスタの上位へ転送
条件分岐	branch on not equal	`bne`	等しくないときに分岐
	branch on equal	`beq`	等しいときに分岐
	branch on greater than or equal to zero	`bgez`	>= のときに分岐
	branch on less than or equal to zero	`blez`	<= 0 のときに分岐
	branch on greater than zero	`bgtz`	> 0 のときに分岐
	branch on less than zero	`bltz`	< 0 のときに分岐
条件判定	set on less than	`slt`	< のとき 1 をセット
	set on less than unsigned	`sltu`	符号なし `slt`
	set on less than immediate	`slti`	即値 `slt`
	set on less than immediate unsigned	`sltiu`	符号なし即値 `slt`
ジャンプ	jump	`j`	ジャンプ
手続きサポート (ジャンプ)	jump and link	`jal`	PC 値をレジスタに退避しジャンプ
	jump register	`jr`	レジスタに退避させていた PC 値を戻す
	jump and link register	`jalr`	`jal` と `jr`
手続きサポート (条件分岐)	branch on greater than or equal to zero and link	`bgezal`	`bgez` と `jal`
	branch on less than zero and link	`bltzal`	`bltz` と `jal`

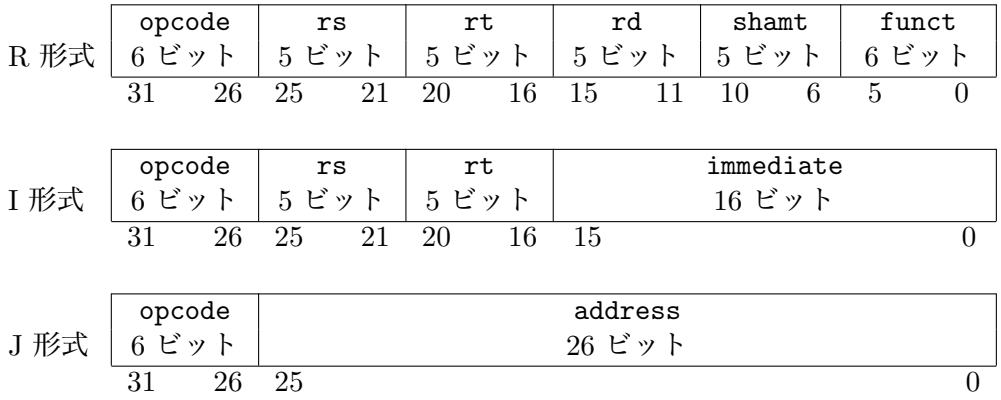


図 2: MIPS の命令のフィールド構成

命令は，プログラムの実行の流れを手続きの方へ分岐させる命令，手続きから元のプログラムの実行の流れに戻す命令である．

1.2.3 命令の表現

MIPS の 1 命令の長さは 32 ビットである．命令はプロセッサ・ハードウェアにも人にも理解しやすいように長さ数ビットの**フィールド**（field）から構成されている．フィールドの枠取りは**命令形式**と呼ばれ，MIPS の主な命令形式には (1) **R 形式**，(2) **I 形式**，(3) **J 形式**の 3 種類がある．図 2 にこれらの命令形式のフィールド構成を示す．R 形式の命令は 6 個のフィールド `opcode`, `rs`, `rt`, `rd`, `shamt`, `funct` から構成され，I 形式の命令は 4 個のフィールド `opcode`, `rs`, `rt`, `immediate` から構成されている．J 形式の命令は，2 個のフィールド `opcode`, `address` から構成されている．

R 形式の命令は，レジスタに収められているデータに対して演算を行い，その結果をレジスタに収める命令である．また，I 形式の命令は，即値およびデータ転送用の命令であり，レジスタに収められているデータと命令内に書かれているデータを元に処理を行う．その結果は，レジスタまたはメモリに収められる．J 形式の命令は，ジャンプおよび分岐用の命令であり，命令内に書かれているアドレスを元にジャンプおよび分岐処理を行う．

全ての命令形式において，命令の 26 ビット目から 31 ビット目までの 6 ビットは，命令の形式および操作の種類を表す `opcode` で，**命令操作コード**（opcode; オペコード）と呼ばれる．R 形式と I 形式にある `rs` フィールドは第 1 ソース・オペランドと呼ばれ，1 つ目のソース・オペランドのレジスタ，即ち操作対象データの在処を表す．R 形式にある `rd` フィールドはデスティネーション・オペランドと呼ばれ，デスティネーション・オペランドのレジスタ，即ち操作結果データの格納先を表す．R 形式と I 形式にある `rt` フィールドは R 形式では第 2 ソース・オペランドと呼ばれ，2 つ目のソース・オペランドのレジスタ，即ち操作対象データの在処を表す．I 形式では，`rt` フィールドはデスティネーション・オペランドのレジスタで，操作結果データの格納先を表す．I 形式の `immediate` フィールドは，定数または即値のオペランドで，ここにデータやアドレスが直接書かれる．J 形式の `address` フィールドも，定数または即値のオペランドで，ここにアドレスが直接書かれる．R 形式の `funct` フィールドには，R 形式の命令の機能が書かれる．R 形式の `shamt` フィールドは `shift amount` の略であり，語中のビットをシフト（shift）する命令のとき利用され，ここにシフトするビット数が書かれる．

以降では，各命令形式の命令の，アセンブリ言語と機械語について述べる．

表 3: R 形式の主要な命令

区分	命令	略号	機能の概要
算術演算	add	`add`	整数加算
	add unsigned	`addu`	符号なし整数加算
	subtract	`sub`	整数減算
	subtract unsigned	`subu`	符号なし整数減算
	shift right arithmetic	`sra`	算術右シフト
論理演算	and	`and`	ビット単位 AND
	or	`or`	ビット単位 OR
	nor	`nor`	ビット単位 NOR
	xor	`xor`	ビット単位 XOR
	shift left logical	`sll`	論理左シフト
	shift right logical	`srl`	論理右シフト
	shift left logical variable	`sllv`	論理左変数シフト
	shift right logical variable	`srlv`	論理右変数シフト
条件判定	set on less than	`slt`	$<$ のとき 1 をセット
	set on less than unsigned	`sltu`	符号なし `slt`
手続きサポート（ジャンプ）	jump register	`jr`	レジスタに退避させていた PC 値を戻す
	jump and link register	`jalr`	`jal` と `jr`

R 形式の命令

R 形式の命令には，算術演算命令，論理演算命令，条件判定命令，手続きサポートのための命令がある．表 3 に R 形式の主要な命令を示す．

アセンブリ言語では，R 形式の各フィールドがシンボルで表される．表 4 に R 形式の主要な命令のアセンブリ言語の例を示す．例えば，アセンブリ言語で整数の減算（subtract）は次のように書かれる．

```
sub $s1, $s2, $s3
```

`sub` は減算命令の名前 subtract の略号，`$s1`，`$s2`，`$s3` はオペランドのレジスタであり，`$s1` はデスティネーション・オペランド，`$s2` は第 1 ソース・オペランド，`$s3` は第 2 ソース・オペランドのレジスタである．この命令の意味は次のとおりである．

```
$s1 = $s2 - $s3
```

この命令によりレジスタ `$s1` に `$s2 - $s3` の結果が格納される．

機械語では，R 形式の `opcode` フィールドは全ての命令で同じであり，0 である．`funct` フィールドは個々の命令に応じて異なり，この値により行う演算が指定される．表 5 に，R 形式の主要な命令の機械語の例を示す．例えば，`sub` を意味する `funct` フィールドの値は 34 である．デスティネーション・オペランド，第 1 ソース・オペランド，第 2 ソース・オペランドの値は使用されるレジスタに応じて異なる．この例では，`$s1` がデスティネーション・オペランド，`$s2` が第 1 ソース・オペランド，`$s3` が第 2 ソース・オペランドのレジスタであり，`$s1`，`$s2`，`$s3` のレジスタ番号である 17，18，19 がそれぞれ `rd`，`rs`，`rt` フィールドの値となっている．

表 4: R 形式の主要な命令のアセンブリ言語の例

区分	命令の例	意味	説明
算術演算	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	3 オペランド，整数加算
	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	3 オペランド，符号なし整数加算
	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	3 オペランド，整数減算
	`subu $s1,$s2,$s3`	`$s1 = $s2 - $s3`	3 オペランド，符号なし整数減算
	`sra $s1,$s2,shamt`	`$s1 = $s2 >> shamt`	定数 `shamt` 分の算術右シフト
論理演算	`and $s1,$s2,$s3`	`$s1 = $s2 AND $s3`	3 オペランド，ビット単位 AND
	`or $s1,$s2,$s3`	`$s1 = $s2 OR $s3`	3 オペランド，ビット単位 OR
	`nor $s1,$s2,$s3`	`$s1 = $s2 NOR $s3`	3 オペランド，ビット単位 NOR
	`xor $s1,$s2,$s3`	`$s1 = $s2 XOR $s3`	3 オペランド，ビット単位 XOR
	`sll $s1,$s2,shamt`	`$s1 = $s2 << shamt`	定数 `shamt` 分の論理左シフト
	`srl $s1,$s2,shamt`	`$s1 = $s2 >> shamt`	定数 `shamt` 分の論理右シフト
	`sllv $s1,$s2,$s3`	`$s1 = $s2 << $s3`	定数 `$s3` 分の論理左シフト
	`srlv $s1,$s2,$s3`	`$s1 = $s2 >> $s3`	定数 `$s3` 分の論理右シフト
条件判定	`slt $s1,$s2,$s3`	`if($s2<$s3) $s1=1 else $1=0`	`$s2` と `$s3` を比較
	`sltu $s1,$s2,$s3`	`if($s2<$s3) $s1=1 else $1=0`	符号なし数値`$s2` と `$s3` を比較
手続きサポート (ジャンプ)	`jr $s1`	`goto $s1`	PC を`$s1` に設定 手続きからの戻り用
	`jalr $s1,$s2`	`goto $s1 + $s2`	PC を`$s1 + $s2` に設定

表 5: R 形式の主要な命令の機械語の例

命令	例	op	rs	rt	rd	shamt	funct
add	`add $s1,$s2,$s3`	0	18	19	17	0	32
addu	`addu $s1,$s2,$s3`	0	18	19	17	0	33
sub	`sub $s1,$s2,$s3`	0	18	19	17	0	34
subu	`subu $s1,$s2,$s3`	0	18	19	17	0	35
sra	`sra $s1,$s2,10`	0	0	18	17	10	3
and	`and $s1,$s2,$s3`	0	18	19	17	0	36
or	`or $s1,$s2,$s3`	0	18	19	17	0	37
nor	`nor $s1,$s2,$s3`	0	18	19	17	0	39
xor	`xor $s1,$s2,$s3`	0	18	19	17	0	38
sll	`sll $s1,$s2,10`	0	0	18	17	10	0
srl	`srl $s1,$s2,10`	0	0	18	17	0	2
sllv	`sllv $s1,$s2,$s3`	0	19	18	17	0	4
srlv	`srlv $s1,$s2,$s3`	0	19	18	17	0	6
slt	`slt $s1,$s2,$s3`	0	18	19	17	0	42
sltu	`sltu $s1,$s2,$s3`	0	18	19	17	0	43
jr	`jr $s1`	0	17	0	0	0	8
jalr	`jalr $s1,$s2`	0	17	0	18	0	9

表 6: I 形式の主要な命令

区分	命令	略号	機能の概要
算術演算	add immediate	addi	即値整数加算
	add immediate unsigned	addiu	即値符号なし整数加算
論理演算	and	andi	即値ビット単位 AND
	or	ori	即値ビット単位 OR
	xor	xori	即値ビット単位 XOR
データ転送	load word	lw	メモリからレジスタへ転送
	store word	sw	レジスタからメモリへ転送
	load upper immediate	lui	定数をレジスタの上位へ転送
条件分岐	branch on not equal	bne	等しくないときに分岐
	branch on equal	beq	等しいときに分岐
	branch on greater than or equal to zero	bgez	>= のときに分岐
	branch on less than or equal to zero	blez	<= 0 のときに分岐
	branch on greater than zero	bgtz	> 0 のときに分岐
	branch on less than zero	bltz	< 0 のときに分岐
条件判定	set on less than immediate	slti	即値 slt
	set on less than immediate unsigned	sltiu	符号なし即値 slt
手続きサポート (条件分岐)	branch on greater than or equal to zero and link	bgezal	bgez と jal
	branch on less than zero and link	bltzal	bltz と jal

I 形式の命令

I 形式の命令には，算術演算命令，論理演算命令，データ転送命令，条件分岐命令，条件判定命令，手続きサポートのための命令がある．表 6 に，I 形式の主要な命令を示す．

I 形式の命令のアセンブリ言語も，R 形式の命令と同様に各フィールドがシンボルを用いて表される．表 7 に I 形式の主要な命令のアセンブリ語の例を示す．表中の imm は immediate フィールドを表している．例えば，アセンブリ言語でメモリからレジスタへの値の転送 (load word) は次のように書かれる．

```
lw $s1, immediate($s2)
```

lw はレジスタへの値の転送命令の名前 load word の略号，$s1，$s2 はオペランドのレジスタである．$s1 はデスティネーション・オペランド，$s2 は第 1 ソース・オペランドのレジスタである．転送する元のメモリのアドレスが immediate($s2) で指定されている．immediate($s2) は$s2 と immediate の和で，値の入っているメモリのアドレスを表すベース相対アドレシングの表記である．sw もベース相対アドレシングの命令である．アドレシング形式には，その他にレジスタ・アドレシング，即値アドレシング，PC 相対アドレシング，擬似直接アドレシングがある．I 形式の命令では，addi，addiu などが即値アドレシング，bne，bqe などが PC 相対アドレシングである．

機械語では，opcode フィールドは命令に応じて異なり，ほとんどの命令は opcode フィールドの値によって識別できる．表 8 に I 形式の主要な命令の機械語の例を示す．例えば，lw を意味する opcode フィールドの値は 35 である．デスティネーション・オペランド，第 1 ソース・オペランドの値は使用されるレジスタに応じて異なる．lw の例では，$s1 がデスティネーション・オペランド，$s2 が第 1 ソース・オペランドであるため，rt，rs のフィールドの値はそれぞれ 17, 18 となる．bgez，bltz，bgezal，bltzal の opcode フィールドは全て 1 であり，条件の種類を rt フィールドの値で識別する．

表 7: I 形式の主要な命令のアセンブリ語の例

区分	命令	意味	備考
算術演算	addi $s1,$s2,imm	$s1 = $s2 + imm	imm を加算
	addiu $s1,$s2,imm	$s1 = $s2 + imm	符号なし imm を加算
論理演算	andi $s1,$s2,imm	$s1 = $s2 AND imm	ビット単位$s2,imm AND
	ori $s1,$s2,imm	$s1 = $s2 OR imm	ビット単位$s2,imm OR
	xori $s1,$s2,imm	$s1 = $s2 XOR imm	ビット単位$s2,imm XOR
データ転送	lw $s1,imm($s2)	$s1 = メモリ ($s2+imm)	メモリ ($s2+imm) から レジスタ$s1 へ転送
	sw $s1,imm($s2)	メモリ ($s2+imm)=$s1	レジスタ$s1 からメモリ ($s2+imm) へ転送
	lui $s1,imm	$s1=imm * 2^{16}	imm を$s1 の上位 16 ビットへ 転送
条件分岐	bne $s1,$s2,imm	if($s1!=$s2) goto (PC+4)+imm*4	$s1 と$s2 が等しくないときに PC は (PC+4)+imm*4
	beq $s1,$s2,imm	if($s1==$s2) goto (PC+4)+imm*4	$s1 と$s2 が等しいときに PC は (PC+4)+imm*4
	bgez $s1,imm	if($s1>=0) goto (PC+4)+imm*4	$s1 が 0 以上のときに PC は (PC+4)+imm*4
	blez $s1,imm	if($s1<=0) goto (PC+4)+imm*4	$s1 が 0 以下のときに PC は (PC+4)+imm*4
	bgtz $s1,imm	if($s1>0) goto (PC+4)+imm*4	$s1 が 0 より大きいときに PC は (PC+4)+imm*4
	bltz $s1,imm	if($s1<0) goto (PC+4)+imm*4	$s1 が 0 より小さいときに PC は (PC+4)+imm*4
条件判定	slti $s1,$s2,imm	if($s2<imm) $s1=1 else $s1=0	$s2 と imm を比較
	sltiu $s1,$s2,imm	if($s2<imm) $s1=1 else $s1=0	符号なし数値$s2 と imm を 比較
手続きサポート (条件分岐)	bgezal $s1,imm	if($s1>=0+$ra) goto (PC+4)+imm*4	戻り番地以上であれば PC は (PC+4)+imm*4
	bltzal $s1,imm	if($s1<0+$ra) goto (PC+4)+imm*4	戻り番地より小さければ PC は (PC+4)+imm*4

表 8: I 形式の主要な命令の機械語の例

命令	例	op	rs	rt	immediate
addi	addi $s1, $s2, 100	8	18	17	100
addiu	addiu $s1, $s2, 100	9	18	17	100
andi	andi $s1, $s2, 100	12	18	17	100
ori	ori $s1, $s2, 100	13	18	17	100
xori	xori $s1, $s2, 100	14	18	17	100
lw	lw $s1, 100($s2)	35	18	17	100
sw	sw $s1, 100($s2)	43	18	17	100
lui	lui $s1, 100	15	0	17	100
bne	bne $s1, $s2, 100	5	18	17	100
beq	beq $s1, $s2, 100	4	18	17	100
bgez	bgez $s1, 100	1	17	1	100
blez	blez $s1, 100	6	17	0	100
bgtz	bgtz $s1, 100	7	17	0	100
bltz	bltz $s1, 100	1	17	0	100
slti	slti $s1, $s2, 100	10	18	17	100
sltiu	sltiu $s1, $s2, 100	11	18	17	100
bgezal	bgezal $s1, 100	1	17	17	100
bltzal	bltzal $s1, 100	1	17	16	100

表 9: J 形式の主要な命令

区分	命令	略号	機能の概要
ジャンプ	jump	j	ジャンプ
手続きサポート (ジャンプ)	jump and link	jal	PC 値をレジスタに退避しジャンプ

J 形式の命令

J 形式の命令には，ジャンプ命令と手続きサポート命令がある．表 9 に，J 形式の主要な命令を示す．

J 形式のアセンブリ言語も，R 形式や I 形式の命令と同様に各フィールドがシンボルを用いて表される．表 10 に J 形式の主要な命令のアセンブリ語の例を示す．例えば，アセンブリ言語でジャンプ j（jump）は次のように書かれる．

```
j address
```

j はジャンプの略号であり，address はジャンプ先のアドレスを指定する値である．j 命令では擬似直接アドレッシングでジャンプ先を決定する．擬似直接アドレッシングとは，命令中の 26 ビットと PC の上位ビットを連結したものがジャンプ先のアドレスとなるアドレッシング形式である．jal 命令も擬似直接アドレッシングでジャンプを行う命令である．

J 形式の命令は opcode フィールドの値で識別される．表 11 に，J 形式の主要な命令の機械語の例を示す．j 命令，jal 命令の opcode フィールドの値はそれぞれ 2, 3 である．address フィールドにはジャンプ先の 26 ビット分のアドレスが入る．

表 10: J 形式の主要な命令のアセンブリ語の例

区分	命令	意味	備考
ジャンプ	`j address`	`goto address * 4`	PC を address*4 に
手続きサポート（ジャンプ）	`jal address`	`$ra=PC+4` `goto address * 4`	次の命令番地を$ra へ PC を address*4 に

表 11: J 形式の主要な命令の機械語の例

命令	例	op	address
j	j 100	2	100
jal	jal 100	3	100

1.3 命令セット・アーキテクチャの実現方式

プロセッサの命令セット・アーキテクチャは，1.2 節で述べたように，プロセッサハードウェアと機械語との間の抽象的なインタフェースである．プロセッサハードウェアは，命令セット・アーキテクチャが論理回路として設計され，ハードウェアの形で実現されたものである．

命令セット・アーキテクチャを実現する方式には，単純なものから複雑でより高性能なものまで様々なものがある．全ての命令の実行が 1 クロック・サイクルを要するシングルサイクルの実現方式や，複数サイクルを要するマルチサイクルの実現方式，シングルサイクル方式をパイプライン化した実現方式などがある．さらに，パイプライン方式でフォワーディング機構を有するものや，分岐予測機構を有するもの，複数命令発行，投機実行の機構を有するもの等もある．

MIPS のような RISC（Reduced Instruction Set Computer）プロセッサは，制御命令の数を減らし，複雑な処理を単純な命令の組み合わせで行うことで，回路を単純化し演算速度の向上を図っている．一方，CISC（Complex Instruction Set Computer）プロセッサは，1 つの命令で複雑な処理を一気に行うことができるように設計されている．ソフトウェア側で指定する命令を減らせる利点がある反面，CPU の仕組みが複雑になり，高速化しにくいという欠点もある．近年の CISC CPU は，パイプライン等の RISC 技術を取り入れ，RISC と CISC の長所を併せ持った CPU となっている．

2　シングルサイクル RISC プロセッサの設計「導入編」

2.1　設計するプロセッサの命令セット・アーキテクチャとその実現方式

本実験では，**1.2 節に示した命令セット・アーキテクチャ**を実現するプロセッサの設計を行う．命令セットアーキテクチャの実現方式は，全ての命令の実行が 1 クロック・サイクルを要する**シングルサイクル方式**とする．

本実験で設計するプロセッサは，参考文献 [7,8,9] のシングルサイクルプロセッサの構成に基づいており，よく知られた RISC プロセッサの一つである MIPS の命令セットのサブセットに対応している．

2.2　プロセッサ設計の準備

本実験では，ICE の Linux マシン上で， Altera 社の EDA ツールを使用して，プロセッサの設計を行う．**計算機と EDA ツールの環境設定方法は，前の実験「EDA ツールを用いた論理回路設計」と全く同じである．3.1 節を参考にして，環境設定を行うこと．**

2.3　プロセッサの動作実験用コンピュータの構成

設計したプロセッサの動作実験は，Altera DE2-115 ボード上に実現する，プロセッサの動作実験用コンピュータを用いて行う．このコンピュータの CPU として，設計した MIPS 型 CPU を使用する．この動作実験用コンピュータは，図 1 に示した一般的なコンピュータをハードウェア実現したものである．図 3 に DE2-115 ボード上に実現するコンピュータの構成を示す．この構成は，5 つの構成要素からなる図 1 の構成に基づいており，キーボードと

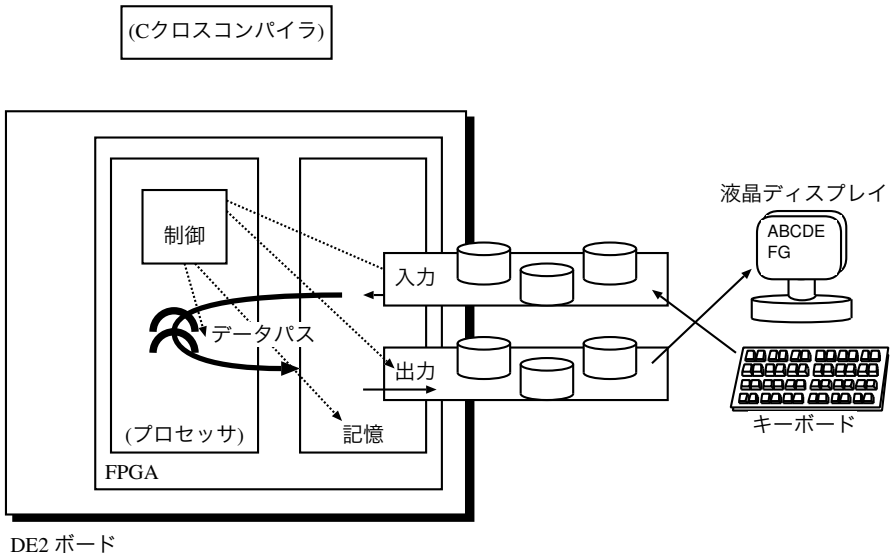


図 3: プロセッサの動作実験用コンピュータの構成

液晶ディスプレイがそれぞれ，(1) 入力装置，(2) 出力装置，DE2-115 ボード上の FPGA 内の回路が (3) 記憶装置，(4) 制御装置，(5) データパスを実現している．C クロスコンパイラは，プロセッサの設計を行う Linux マシン上で動作し，C 言語で書かれたプログラムを本実験で設計する MIPS 型 CPU のマシン・コードに変換する．

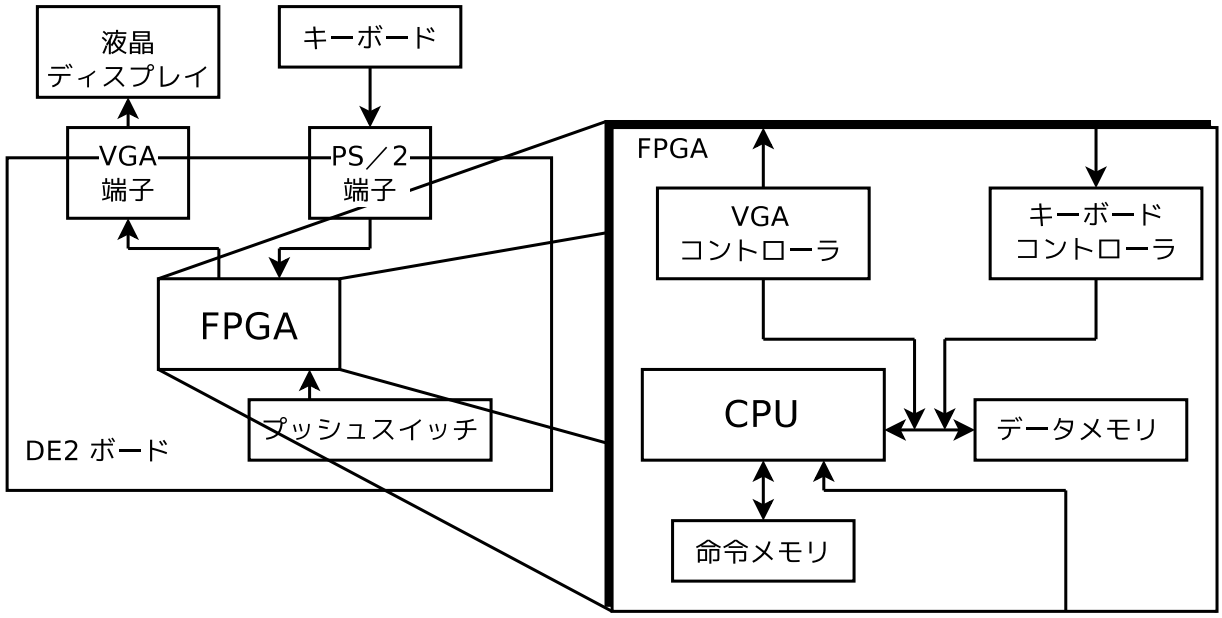


図 4: コンピュータの内部構成

図 4 にコンピュータの内部構成を示す．液晶ディスプレイとキーボードが，DE2-115 ボードの VGA 端子と PS/2 端子を介して DE2-115 ボード上の FPGA に接続されている．この FPGA 内には，CPU や記憶装置（命令メモリとデータメモリ），VGA コントローラ，キーボードコントローラが実現される．

2.4 動作実験の手順

DE2-115 ボードを用いたプロセッサの動作実験は，下記の 1 から 5 の手順で行う．手順 5 のスライドスイッチの操作では，CPU の動作クロック周波数を表 12 に基づき設定する．表 12 は，図 5 に示す DE2-115 ボードのスライドスイッチ SW1，SW0 の設定値と CPU の動作クロック周波数の関係を表している．なお，スライドスイッチは上げると 1，下げると 0 が FPGA に入力される．CPU の動作クロック周波数は 2[Hz], 200[Hz], 1000[Hz], 手動クロックの中から選択でき，手動クロックを選択した場合は，KEY3 を押す毎にクロックパルスが CPU に 1 つ送られる．CPU を 1 クロックずつ動作させる必要がある実験では手動クロックを選択する．DE2-115 ボードの 7 セグメント LED にはプロセッサの PC の値が表示される．なお，KEY2 を押すと，CPU 及び周辺回路がリセットされる．

1. MIPS 型 CPU で実行するプログラムのコンパイル（クロスコンパイル）
2. MIPS マシン・コードから命令メモリ (図 4) のメモリイメージファイルへの変換
3. CPU ならびに周辺回路の論理合成
4. DE2-115 ボードへのダウンロード
5. DE2-115 ボードのスライドスイッチ，プッシュスイッチを操作して CPU でプログラムを実行

表 12: スライドスイッチ SW1，SW0 による CPU の動作クロック周波数の設定

(SW1, SW0)	クロック周波数 [Hz]
(0, 0)	2
(0, 1)	200
(1, 0)	1000
(1, 1)	手動クロック

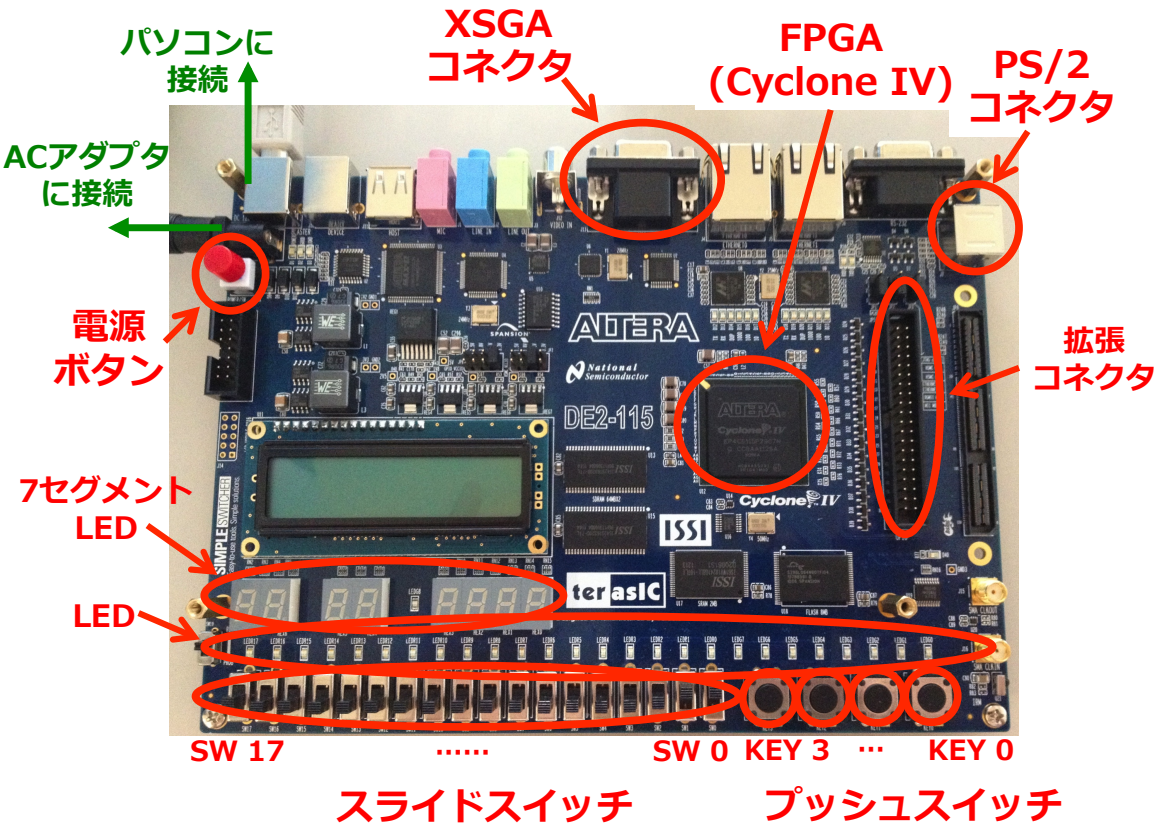


図 5: Altera DE2-115 ボード

3　シングルサイクル RISC プロセッサの設計「基礎編」

本実験では，シングルサイクル RISC プロセッサの設計と動作実験を行う．第 1 週目の実験では，プロセッサの動作実験と，**即値符号なし整数加算命令（addiu）**と**ストア・ワード命令（sw）**についてのプロセッサの追加設計を行う．

3.1　マシン・コードの動作実験 1-1（ディスプレイへの文字出力）

マシン・コードの動作実験 1-1 では，ディスプレイに文字 'B' を 1 つ表示する MIPS マシン・コード print_B.bin と，それを実行するプロセッサを FPGA 上に実現しその動作を確認する．この実験を通じて，プロセッサの動作実験の各手順の理解を目指す．本動作実験では，即値符号なし整数加算命令（add immediate unsigned：addiu）とストア・ワード命令（store word：sw）が未実装なプロセッサにおいて，それらの命令を含む簡単な機械語のマシン・コードを実行すると，どのような動作をするかを観察する．本実験で観察した結果は，

次のプロセッサの追加設計 1 において，addiu と sw が正しく動くプロセッサを完成させた後，動作比較の対象として用いる．

3.1.1 MIPS マシン・コードからのメモリ・イメージファイルの作成

本実験では，まず，MIPS マシン・コードをプロセッサの命令メモリのメモリ・イメージファイルに変換する．この変換により，QuartusII で論理合成可能なメモリ・イメージファイルが得られる．このメモリ・イメージを命令メモリに持ったプロセッサを論理合成することにより，変換元のマシン・コードを実行するプロセッサが得られる． MIPS マシン・コードの例として print_B.bin [1] を使用する．また変換には，変換プログラム bin2v を使用する．

最初に，2.2 節に従って EDA ツールの環境設定を行ったのち，「bin2v print_B.bin」**で MIPS マシン語プログラムからメモリ・イメージファイルを作成する．**この変換により，論理合成用のメモリ・イメージファイル rom8x1024_DE2.mif が得られる．また，機能レベルシミュレーション用の Verilog HDL 記述 rom8x1024_sim.v も同時に生成される．メモリ・イメージファイル rom8x1024_DE2.mif は，論理合成の際に QuartusII によって読まれるファイルである．また，機能レベルシミュレーション用の Verilog HDL 記述 rom8x1024_sim.v は，機能レベルシミュレーションとプロセッサが実行する命令列を確認する際に使用する．

本実験で使用する MIPS マシン・コード print_B.bin は，正しいプロセッサ（即値符号なし整数加算命令 addiu と，ストア・ワード命令 sw が実装済みのプロセッサ）で動作させると，次の 1, 2, 3 のような動作をする命令列を含んだ，バイナリファイルである．

1. データメモリ（RAM）の 0x0300 番地に 0 を格納
   ```
   addiu $s2, $s0, 0x0300
   sw $s0, 0x0000($s2)
   ```
2. RAM の 0x0304 番地に 2 を格納
   ```
   addiu $s3, $s0, 0x0304
   addiu $s2, $s0, 0x0002
   sw $s2, 0x0000($s3)
   ```
3. RAM の 0x0300 番地に 1 を上書き
   ```
   addiu $s3, $s0, 0x0300
   addiu $s2, $s0, 0x0001
   sw $s2, 0x0000($s3)
   ```

3.1.2 命令メモリに格納される命令列の確認

本実験では，次に，プロセッサの命令メモリに格納される命令列の確認を行う．この確認には，bin2v により生成された機能レベルシミュレーション用の Verilog HDL 記述 rom8x1024_sim.v を使用する．図 6 に rom8x1024_sim.v の一部を示す．図 6 の case ブロック内の各行は，本実験で設計するプロセッサにおける，命令メモリの 10-bit アドレスとそこに格納される 32-bit 命令の機械語が記述されている．各行の // 以降のコメント部には，その行に記述されているアドレスと命令に関するコメントが記述されている．コメント部には，左から，実際の MIPS の命令メモリにおけるアドレス，命令名，命令の内容が記述されている．ここで，シンボル REG[0], ..., REG[31] は，レジスタ 0 番から 31 番，すなわち $zero, ..., $ra

```
<省略>
case (word_addr)
 <省略>
 10'h00b: data = 32'h24020300; // 0040002c: ADDIU, REG[2]<=REG[0]+768(=0x00000300);     ここが PC=0x002c の命令
 10'h00c: data = 32'hac400000; // 00400030: SW, RAM[REG[2]+0]<=REG[0];
 10'h00d: data = 32'h24030304; // 00400034: ADDIU, REG[3]<=REG[0]+772(=0x00000304);
 10'h00e: data = 32'h24020002; // 00400038: ADDIU, REG[2]<=REG[0]+2(=0x00000002);
 10'h00f: data = 32'hac620000; // 0040003c: SW, RAM[REG[3]+0]<=REG[2];
 10'h010: data = 32'h24030300; // 00400040: ADDIU, REG[3]<=REG[0]+768(=0x00000300);
 10'h011: data = 32'h24020001; // 00400044: ADDIU, REG[2]<=REG[0]+1(=0x00000001);
 10'h012: data = 32'hac620000; // 00400048: SW, RAM[REG[3]+0]<=REG[2];
 <省略>
endcase
<省略>
```

図 6: rom8x1024_sim.v の一部

を表す (表 1)．また，シンボル RAM[w] は，データメモリの w 番地を表す．
例えば，図 6 の case ブロック内の最初の記述は，本実験で設計するプロセッサの命令メモリの 0x00b 番地に機械語 0x24020300 が格納されることを表している．また，この命令は実際の MIPS では 0x0040002c に格納され，命令名は addiu，レジスタ 2 番にレジスタ 0 番（値は常に 0）+768 の結果をセットする命令であることを表している．なお，本実験で設計するプロセッサのプログラムカウンタ PC=(0x h3 h2 h1 h0) が指す命令は，本プロセッサの命令メモリでは，アドレスを右に 2-bit シフトした ((0x h3 h2 h1 h0) >> 2) 番地に格納されている．例えば，本実験で設計するプロセッサの PC=0x002c が指す命令は，本プロセッサの命令メモリの (0x002c) >> 2，即ち 0x000b 番地に格納されている．また，本実験で設計するプロセッサのプログラムカウンタ PC=(0x h3 h2 h1 h0) が指す命令は，実際の MIPS の命令メモリでは，アドレスの上位に 0x0040 を付加した (0x0040 h3 h2 h1 h0) 番地に格納されている．

print_B.bin から生成された rom8x1024_sim.v, または，図 6 の Verilog HDL 記述を解析し，次の 1, 2, 3, 4, 5 に答えよ．なお，addiu は即値符号なし整数加算命令，sw はレジスタの値をメモリに転送するストア・ワード命令，レジスタ 0 番の値は常に 0 である．

1. プロセッサが PC=0x002c の命令を実行することにより，レジスタ REG[2] の値がいくらになるかを予想せよ．
2. プロセッサが PC=0x0030 の命令を実行することにより，RAM の 768 (0x00000300) 番地の値がいくらになるかを予想せよ．
3. プロセッサが PC=0x0034 番地の命令を実行することにより，REG[3] の値いくらになるかを予想せよ．
4. プロセッサが PC=0x003c の命令を実行すると，RAM の何番地の値が変化し，変化後の値はいくらかを予想せよ．
5. プロセッサが PC=0x0048 の命令を実行すると，RAM の何番地の値が変化し，変化前，変化後の値はそれぞれいくらかを予想せよ．

3.1.3 論理合成

本実験では，次に，addiu 命令と sw 命令が未実装なプロセッサならびに命令メモリ，その他周辺回路の論理合成を行う．論理合成には，bin2v により生成された論理合成用のメモリ・イ

[1] http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k01_addiu_sw/print_B.bin

メージファイル rom8x1024_DE2.mif とプロセッサの Verilog HDL 記述一式 mips_de2-115.tar.gz [2] を使用する．

「tar xvfz ./mips_de2-115.tar.gz」で mips_de2-115.tar.gz を展開し，プロセッサのソース一式を得る．プロセッサのソース一式と rom8x1024_DE2.mif を，Quartus II を使用して論理合成すると FPGA にダウンロード可能なストリーム・アウト・ファイル DE2_115_Default.sof が得られる．

本実験を通じで完成させる未完成なプロセッサの Verilog HDL 記述一式が，ディレクトリ mips_de2-115 のサブディレクトリ MIPS に展開される．新たに，プロセッサのメイン制御回路の Verilog HDL 記述 main_ctrl.v も，同じサブディレクトリ mips_de2-115/MIPS 内に存在する．**メモリ・イメージファイル rom8x1024_DE2.mif をディレクトリ mips_de2-115 にコピーし，ディレクトリ mips_de2-115 に cd して，「quartus_sh --flow compile DE2_115_Default」で論理合成を行う．**論理合成には，計算機の性能により 5 分から 20 分程度の時間がかかる．論理合成が完了すると，ディレクトリ mips_de2-115 内に FPGA にダウンロード可能なプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof が生成される．

3.1.4 FPGA を用いた回路実現

本実験では，次に，addiu 命令と sw 命令が未実装なプロセッサの実際の動作を観察する．観察した結果は，次のプロセッサの追加設計 1 において，addiu と sw が正しく動くプロセッサを完成させた後，動作比較の対象として用いる．ここでは，論理合成により生成されたプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof を， Quartus II を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させる．DE2-115 ボード上のプッシュスイッチ KEY2, KEY3 は，それぞれプロセッサをリセットするためのスイッチ，クロックパルスを生成するためのスイッチである．

DE2-115 ボードのスライドスイッチ SW0, SW1 をともに 1（上）にして，プロセッサへのクロック供給を手動モードにする．プロセッサへのクロック供給が手動モードの時，KEY3 を 1 回押すと，プロセッサにクロックパルスが 1 つ送られ，プロセッサは PC の指している命令メモリの命令を 1 つ実行する．なお，本実験で設計するプロセッサは，命令メモリの 0x0000 番地の命令から実行を開始する．

今回プロセッサが実行するマシン・コード print_B.bin はディスプレイ下部に文字 'B' を 1 つ表示するプログラムである．**KEY3 を数回押しプロセッサにクロックパルスを送り，プロセッサに PC=0x0000 番地から PC=0x0048 番地までの命令を実行させ，ディスプレイ下部に文字 'B' が 1 つ表示されるかどうかを確認せよ．**この時，ディスプレイ上部にはプロセッサ内部の主な信号線の現在の値が表示されている．一方，ディスプレイ下部に文字は全く表示されないはずである．

図 7 に動作実験 1-1 のプロセッサのブロック図を示す．ディスプレイ上の信号線名とブロック図中の信号線名は，似た名前のもの同士が対応している．例えば，ディスプレイ上の表示 PC, ALUY がブロック図中のプログラムカウンタ PC, ALU の出力 alu_y にそれぞれ対応している．ALUY の表示の後の A, CTRL, B, COMP は，それぞれブロック図中の ALU の入力 a, alu_ctrl, b, 出力 alu_cmp に対応している．COMP の表示の後の REGD1, IDX, REGD2, IDX は，それぞれブロック図中のレジスタファイル Registers の出力 read_data1, 入力 read idx1, 出力 read_data2, 入力 read idx2 に対応している．その後の REGWRITED, IDX, WEN は，それぞれ Registers の入力 write data, write idx, write enable に対応している．RAMDATA, ADDR, WDATA, WEN は，それぞれブロック図中のデータメモリ RAM の出力 RAM data, 入力 RAM

[2] http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k01_addiu_sw/mips_de2-115.tar.gz

address, RAM write data, write enable に対応している．これらの対応関係をまとめると表 13 のようになる．ブロック図中の線の幅はビット幅と対応しており，一番細い線は 1-bit の線，一番太い線は 32-bit の配線を表している．また，ブロック図左下の ROM が，命令メモリである．プロセッサはここから命令を読み，命令毎に決められた処理を行う．ブロック図右下の RAM は，データメモリである． **3.1.2 節の 1, 2, 3, 4, 5 で予想した結果と同じように動作するかどうかを確認せよ（予想と異なり，正しくない動作のはずである）．** この結果から，プロセッサが，addiu 命令と sw 命令を正しく実行できていないことが分かる．

次の実験 1-2 では，プロセッサの追加設計を行い，プロセッサ内部で行われるデータ転送や演算などを制御するメイン制御回路を，これらの命令に対応したものにする．

表 13: ディスプレイに表示される信号線名とブロック図中の信号線との対応関係

ディスプレイに表示される信号線名	ブロック図中の信号線
PC	PC（プログラムカウンタの現在の値）
ALUY	alu_y（ALU の演算結果出力）
A	a（ALU への入力）
CTRL	alu_ctrl（ALU への制御用入力）
B	b（ALU への入力）
COMP	alu_cmp（ALU での比較結果出力）
REGD1	read data1（レジスタファイル Registers の出力）
REGD1 の後の IDX	read idx1（Registers への入力）
REGD2	read data2（Registers の出力）
REGD2 の後の IDX	read idx2（Registers への入力）
REGWRITED	write data（Registers への入力 ）
REGWRITED の後の IDX	write idx（Registers への入力）
REGWRITED の後の WEN	write eneble（Registers に対する書込許可制御入力）
RAMDATA	RAM data（データメモリ RAM からの出力）
ADDR	RAM address（RAM へのアクセスアドレス入力）
WDATA	RAM write data（RAM への書込データ入力）
WDATA の後の WEN	write enable（RAM に対する書込許可制御入力）

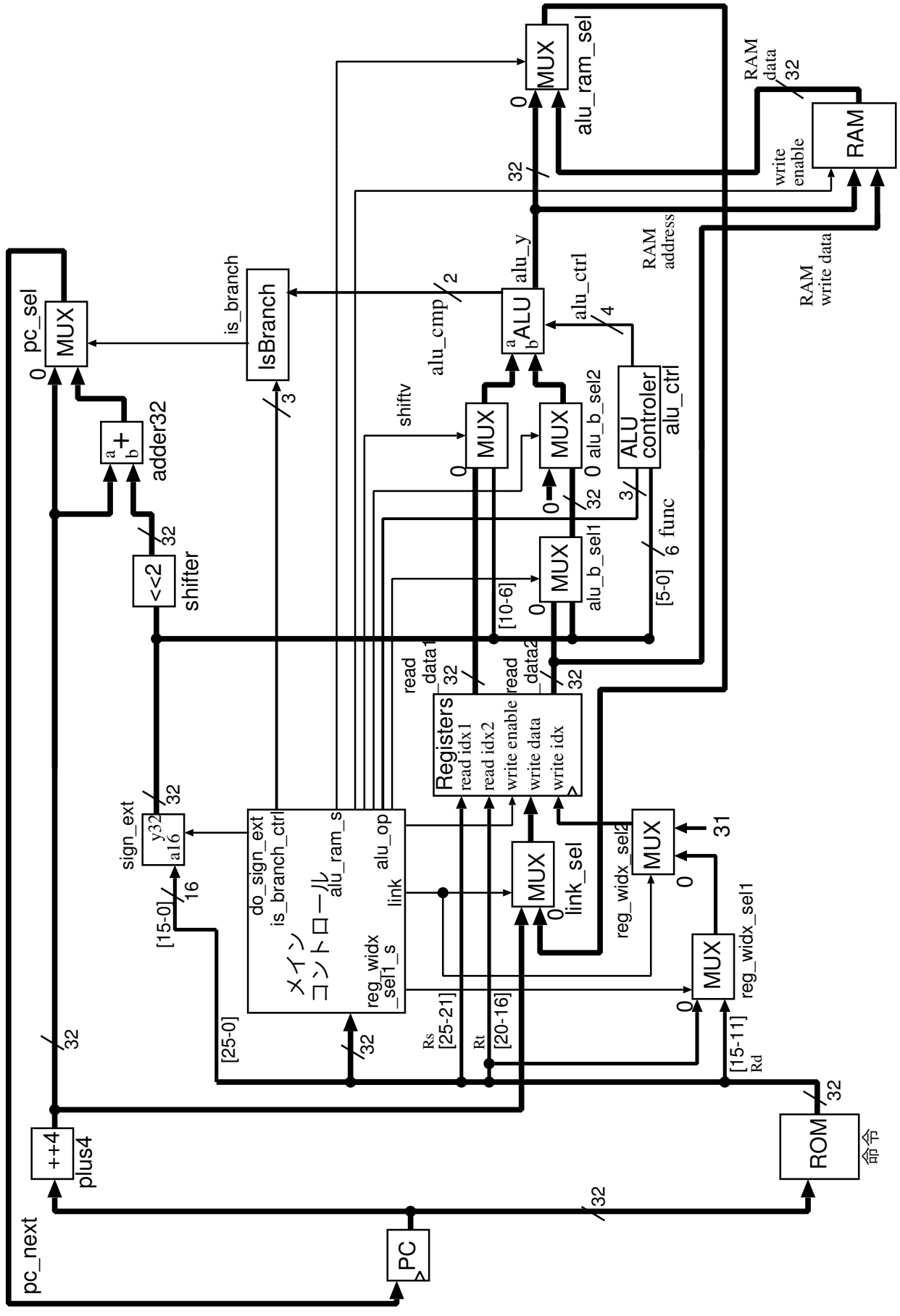


図 7: 動作実験 1-1 のプロセッサのブロック図

3.2 プロセッサの追加設計1（addiu 命令，sw 命令）と動作実験 1-2

本実験では，まず，プロセッサの追加設計と動作実験を行い，プロセッサの追加設計の手順とプロセッサの動作の理解を目指す．ここでは，addiu 命令と sw 命令が未実装なプロセッサを例とし，追加設計を行い，両命令が正しく実行されるプロセッサを完成させる．さらに，プロセッサを実際に動作させて観察する．

3.2.1 addiu 命令のためのメイン制御回路の追加設計

本実験では，動作実験 1 で動作を確認した addiu 命令と sw 命令が未実装なプロセッサについて，追加設計を行う．ここでは，動作実験 1 で使用したプロセッサのメイン制御回路の Verilog HDL 記述 main_ctrl.v を使用する．main_ctrl.v は，動作実験 1 でプロセッサの Verilog HDL 記述一式 mips_de2-115.tar.gz を展開した際に作成されたディレクトリ mips_de2-115 のサブディレクトリ MIPS にある．**ソースファイル main_ctrl.v 中のコメント，追加設計 1 のヒント (1)〜(9) の周辺を，下記の 1, 2, 3, 4, 5, 6, 7, 8, 9 の手順で適切なものに変更せよ．**

0. addiu 命令について

   - addiu 命令は，命令の rs フィールドで指定されるレジスタの値と命令に直接書かれている値 immediate を，符号なし整数加算し，結果を命令の rt フィールドで指定されるレジスタに格納する命令である．
   - addiu 命令実行時のプロセッサ内の信号の流れを図 8 に示す．青線（濃い灰色の線），緑線（薄い灰色の線）とラベル付けされた信号線が，addiu 命令の実行に関わっている．以下では，信号の流れがブロック図のようになるように，赤線（(2)〜(9) の番号付きの線）とラベル付けされた制御信号を適切に設定する．なお，**制御信号に付いた (2)〜(9) の番号と，ヒントの番号の間には対応関係がある．**
   - addiu 命令は符号拡張された immediate と rs の符号なし整数加算を行う．
   - ブロック図中の sign_ext は符号拡張モジュールである（参考文献 [9] pp.285，または参考文献 [8] pp.270）．また，MUX は，2 入力 1 出力の Multiplexer, 選択回路を表しており，その 2 つの入力信号のうち，0 のラベルが付けられている方が，選択信号が 0 の時に出力される信号である．ALU (Arithmetic and Logic Unit; 算術論理演算ユニット) は，加算や減算，シフト，AND, OR などの演算を行うものである．

1. 追加設計 1 のヒント (1)：I 形式の命令 addiu の追加，命令コードの定義

   - addiu の命令操作コードが「　　」であることから，**記述「　　」を「　　」に変更する．**

2. 追加設計 1 のヒント (2)：I 形式の命令 addiu の追加，is_branch モジュールへの制御信号の記述

   - is_branch は条件分岐用のモジュールである（図 8，参考文献 [9] pp.295-299, pp.287，または参考文献 [8] pp.280-284, pp.271）．

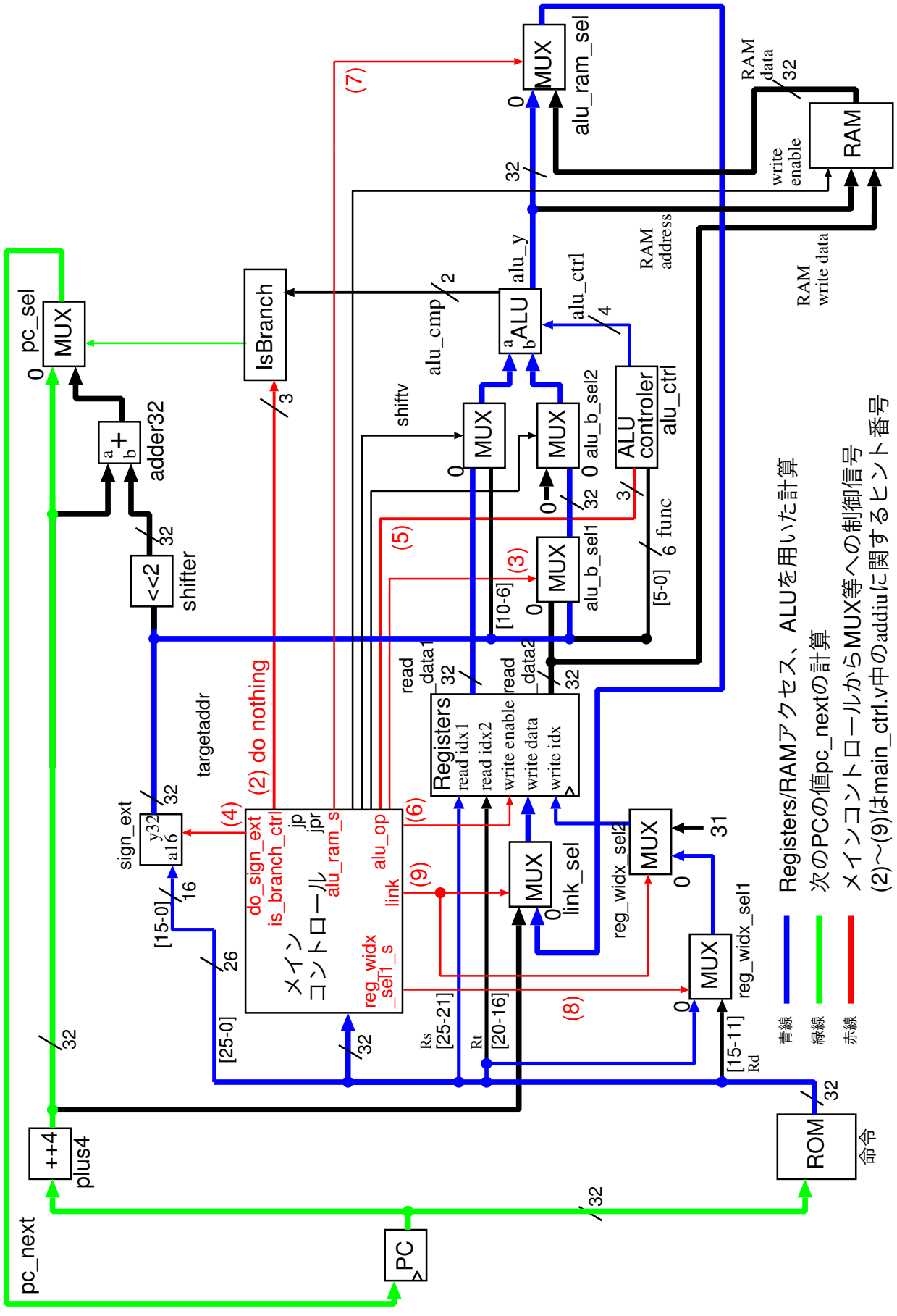


図 8: addiu 命令実行時のプロセッサ内の信号の流れ

- addiu 命令は beq (branch on equal) 命令などの条件分岐命令ではないので，is_branch への制御信号としては「　　」が適切である（ソース中の is_branch に関するコメント (ヒントの数十行上あたり)「// is_branch_d0 // 分岐判定モジュール is_branch の制御信号 // 3'b000, ==, EQ // ... ＜省略＞... // 3'b110, do nothing」より）.
- **記述「　　」を「　　」に変更する.**

3. 追加設計 1 のヒント (3)：I 形式の命令 addiu の追加，ALU の入力ポート B へ流すデータを選択するセレクト信号の記述

   - ALU の B ポートに，命令に直接書かれている値 immediate（命令 [15:0]）を転送するには，セレクタ alu_b_sel1 のセレクト信号を 1'b0 にするのがよいか，1'b1 にするのがよいかを考える.
   - 「　　」にするのがよいことから (図 8 より)，**記述「　　」を「　　」に変更する.**

4. 追加設計 1 のヒント (4)：I 形式の命令 addiu の追加，符号拡張を行うかどうかの制御

   - sign_ext は符号拡張モジュールである（図 8，参考文献 [9] pp.285，または参考文献 [8] pp.270）.
   - sign_ext への制御信号としては「　　」が適切である（ソース中の do_sign_ext に関するコメント (ヒントの数行上あたり)「// do_sign_ext // 符号拡張モジュール sign_ext の制御信号// do_sign_ext == 1'b0：16-bit データを 32-bit 化するとき符号拡張を行わない// do_sign_ext == 1'b1：16-bit データを 32-bit 化するとき符号拡張を行う」より）.
   - **記述「　　」を「　　」に変更する.**

5. 追加設計 1 のヒント (5)：I 形式の命令 addiu の追加，加算を行う制御信号の記述

   - alu_op は ALU 制御モジュール alu_ctrler への制御信号である（図 8，参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）.
   - addiu 命令は ALU に加算を行わせる命令なので，制御信号 alu_op の値として「　　」が適切である（ソース中の alu_op に関するコメント (ヒントの数行上あたり)「 // alu_op // ALU 制御モジュール alu_ctrler の制御信号// 3'b000, ALU に加算を行わせる// 3'b001, ALU に LUI の処理を行わせる// 3'b010, ALU に R 形式の命令に対して，R 形式の機能コードに応じた演算を行わせる// 3'b011, ALU に AND 演算を行わせる// 3'b100, ALU に OR 演算を行わせる// 3'b101, ALU に XOR 演算を行わせる// 3'b110, ALU に SLT の処理を行わせる// 3'b111, ALU に SLTU の処理を行わせる」より）.
   - **記述「　　」を「　　」に変更する.** 参考文献 [7,8,9] とは制御コードがやや異なる.

6. 追加設計 1 のヒント (6)：I 形式の命令 addiu の追加，レジスタファイルへの制御信号の記述

- reg_write_enable はレジスタファイル registers の書き込み制御信号である（図 8, 参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）.
- addiu 命令は演算結果をレジスタに書き込む命令なので，制御信号 reg_write_enable の値として「　　」が適切である（ソース中の reg_write_enable に関するコメント (ヒントの数行上あたり)「// reg_write_enable // レジスタファイル registers の書き込み制御信号// reg_write_enable == 1'b0:書き込みを行わない// reg_write_enable == 1'b1：書き込みを行う」より）.
- **記述「　　」を「　　」に変更する.**

7. 追加設計 1 のヒント (7)：I 形式の命令 addiu の追加，レジスタファイルの方へ流すデータを選択するセレクト信号の記述
   - alu_ram_sel_s はセレクタ alu_ram_sel モジュールのセレクト信号である（図 8，参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）.
   - ALU から出てくる演算結果をレジスタに転送するには，alu_ram_sel のセレクト信号を 1'b0 にするのがよいか，1'b1 にするのがよいかを考える.
   - 「　　」にするのがよいことから (図 8 より)，**記述「　　」を「　　」に変更する.**
8. 追加設計 1 のヒント (8)：I 形式の命令 addiu の追加，レジスタファイルの write_idx へ流すデータを選択するセレクト信号の記述 (1)
   - reg_widx_sel1_s はセレクタ reg_widx_sel1 モジュールのセレクト信号である（図 8, 参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）.
   - レジスタファイルのデータ書き込み先インデックス write_idx に，命令の rt（命令 [20:16]）を転送するには，reg_widx_sel1 のセレクト信号を 1'b0 にするのがよいか，1'b1 にするのがよいかを考える.
   - 「　　」にするのがよいことから (図 8 より)，**記述「　　」を「　　」に変更する.**
9. 追加設計 1 のヒント (9)：I 形式の命令 addiu の追加，レジスタファイルの write_idx へ流すデータを選択するセレクト信号の記述 (2)
   - link はセレクタ reg_widx_sel2 モジュールのセレクト信号である（図 8，参考文献 [9] pp.101，または参考文献 [7] pp.71）.
   - レジスタファイルのデータ書き込み先インデックス write_idx に，命令の rt（命令 [20:16]）を転送するには，reg_widx_sel2 のセレクト信号を 1'b0 にするのがよいか，1'b1 にするのがよいかを考える.
   - 「　　」にするのがよいことから（図 8 より），**記述「　　」を「　　」に変更する.**

3.2.2 `sw` 命令のためのメイン制御回路の追加設計

本実験では，次に，sw 命令についての追加設計を行う．ここでも，プロセッサのメイン制御回路の Verilog HDL 記述 `main_ctrl.v` を使用する．`main_ctrl.v` は，addiu 命令につ

いての追加設計を行った後のものを使用する．**ソースファイル** `main_ctrl.v` **中のコメント，追加設計 1 のヒント (10)〜(16) の周辺を，下記の 1, 2, 3, 4, 5, 6, 7 の手順で適切に変更せよ．**

0. sw 命令について

   - sw 命令は，命令の rt フィールドで指定されるレジスタの値をメモリに転送する命令である．命令の rs フィールドで指定されるレジスタの値と命令に直接書かれている値 immediate との和が，転送先のメモリのアドレスとなる．
   - sw 命令実行時のプロセッサ内の信号の流れを図 9 に示す．青線（濃い灰色），緑線（薄い灰色）とラベル付けされた信号線が sw 命令の実行に関わっている．以下では，信号の流れがブロック図のようになるように，赤線（(11)〜(16) の番号）とラベル付けされた制御信号を適切に設定する．制御信号に付いた (11)〜(16) の番号と，ヒントの番号の間には対応関係がある．

1. 追加設計 1 のヒント (10)：I 形式の命令 sw の追加，命令コードの定義

   - sw の命令操作コードが「　　」であることから，**記述「　　」を「　　」に変更する．**

2. 追加設計 1 のヒント (11)：I 形式の命令 sw の追加，RAM への制御信号の記述

   - ram_write_enable はメモリの書き込み制御信号である（図 9）．
   - sw 命令はレジスタの値をメモリに書き込む命令なので，制御信号 ram_write_enable の値として「　　」が適切である（ソース中の ram_write_enable に関するコメント (ヒントの数行上あたり)「// ram_write_enable // RAM の書き込み制御信号// ram_write_enable == 1'b0：書き込みを行わない// ram_write_enable == 1'b1：書き込みを行う」より）．
   - **記述「　　」を「　　」に変更する．**

3. 追加設計 1 のヒント (12)：I 形式の命令 sw の追加，is_branch モジュールへの制御信号の記述

   - is_branch は条件分岐用のモジュールである（図 9，参考文献 [9] pp.295-299, pp.287, または参考文献 [8] pp.280-284, pp.271）．
   - sw 命令は beq (branch on equal) 命令などの条件分岐命令ではないので，is_branch への制御信号としては「　　」が適切である（ソース中の is_branch に関するコメント (ヒントの数十行上あたり)「// is_branch_d0 // 分岐判定モジュール is_branch の制御信号 // 3'b000, ==, EQ // ...＜省略＞... // 3'b110, do nothing」より）．
   - **記述「　　」を「　　」に変更する．**

4. 追加設計 1 のヒント (13)：I 形式の命令 sw の追加，ALU の入力ポート B へ流すデータを選択するセレクト信号の記述

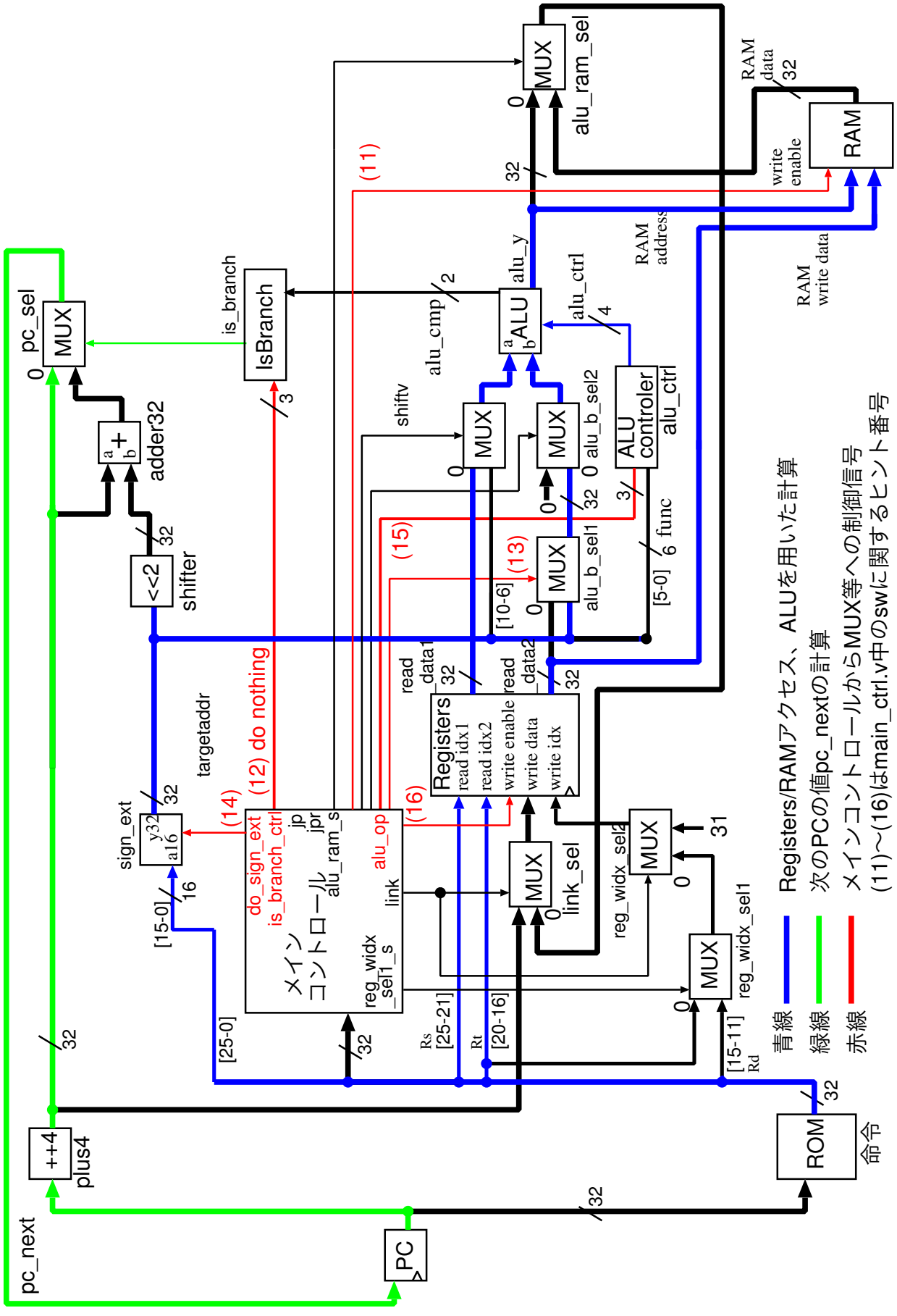


図 9: sw 命令実行時のプロセッサ内の信号の流れ

- ALU の B ポートに，命令に直接書かれている値 address（命令 [15:0]）を転送するには，セレクタ alu_b_sel1 のセレクト信号を 1'b0 にするのがよいか，1'b1 にするのがよいかを考える．
- 「　　」にするのがよいことから（図 9 より），**記述「　　」を「　　」に変更する．**

5. 追加設計 1 のヒント (14)：I 形式の命令 sw の追加，符号拡張を行う制御信号の記述

   - sign_ext は符号拡張モジュールである（図 9，参考文献 [9] pp.285，または参考文献 [8] p.270）．
   - sw 命令はアドレス計算のために，符号拡張された address と rs の符号なし整数加算を行う．
   - sign_ext への制御信号としては「　　」が適切である（ソース中の do_sign_ext に関するコメント (ヒントの数行上あたり)「// do_sign_ext // 符号拡張モジュール sign_ext の制御信号// do_sign_ext == 1'b0：16-bit データを 32-bit 化するとき符号拡張を行わない// do_sign_ext == 1'b1：16-bit データを 32-bit 化するとき符号拡張を行う」より）．
   - **記述「　　」を「　　」に変更する．**

6. 追加設計 1 のヒント (15)：I 形式の命令 sw の追加，加算を行う制御信号の記述

   - alu_op は ALU 制御モジュール alu_ctrler への制御信号である（図 9，参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）．
   - sw 命令は ALU に加算を行わせる命令なので，制御信号 alu_op の値として「　　」が適切である（ソース中の alu_op に関するコメント (ヒントの数行上あたり)「// alu_op // ALU 制御モジュール alu_ctrler の制御信号// 3'b000, ALU に加算を行わせる// 3'b001, ALU に LUI の処理を行わせる// 3'b010, ALU に R 形式の命令に対して，R 形式の機能コードに応じた演算を行わせる// 3'b011, ALU に AND 演算を行わせる// 3'b100, ALU に OR 演算を行わせる// 3'b101, ALU に XOR 演算を行わせる// 3'b110, ALU に SLT の処理を行わせる// 3'b111, ALU に SLTU の処理を行わせる」より）．
   - **記述「　　」を「　　」に変更する．**
   - 参考文献 [7,8,9] とは制御コードがやや異なる．

7. 追加設計 1 のヒント (16)：I 形式の命令 sw の追加，レジスタファイルへの制御信号の記述

   - reg_write_enable はレジスタファイル registers の書き込み制御信号である（図 9，文献 [9] pp.290-299，または参考文献 [8] pp.274-284）．
   - sw 命令はレジスタに値を書き込まない命令なので，制御信号 reg_write_enable の値として「　　」が適切である（ソース中の reg_write_enable に関するコメント (ヒントの数行上あたり)「// reg_write_enable // レジスタファイル registers の書き込み制御信号// reg_write_enable == 1'b0：書き込みを行わない// reg_write_enable == 1'b1：書き込みを行う」より）．

- **記述「　　」を「　　」に変更する.**

3.2.3　論理合成

本実験では，次に，追加設計後のプロセッサならびに命令メモリ，その他周辺回路の論理合成を行う．論理合成には，追加設計後の main_ctrl.v と，動作実験 1-1 で使用したその他プロセッサの Verilog HDL 記述一式，print_B.bin から生成したメモリ・イメージファイル rom8x1024_DE2.mif を使用する．

追加設計後の main_ctrl.v **を，プロセッサなど一式のディレクトリ** mips_de2-115 **の，サブディレクトリ** MIPS **に置く．更に，ディレクトリ** mips_de2-115 **に移動** (cd) **し，**pring_B.bin **の** rom8x1024_DE2.mif **があることを確認し，「**quartus_sh --flow compile DE2_115_Default**」で論理合成を行う．** 論理合成が完了すると，ディレクトリ mips_de2-115 内に FPGA にダウンロード可能なプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof が生成される．

3.2.4　FGPA を用いた回路実現

本実験では，次に，追加設計後のプロセッサの実際の動作を観察し，動作実験 1-1 で観察した結果との比較を行う．ここでは，論理合成により生成されたプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof を， Quartus II を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させる．

スライドスイッチ SW0, SW1 をともに 1 にし，ロセッサへのクロック供給を手動モードにする．プロセッサが実行するマシン・コード print_B.bin は，ディスプレイ下部に文字 'B' を 1 つ表示するプログラムである．**KEY3 を数回押しプロセッサにクロックパルスを送り，プロセッサに PC=0x0000 番地から PC=0x0048 番地までの命令を実行させ，ディスプレイ下部に文字 'B' が 1 つ表示されるかどうかを確認せよ（文字 'B' が 1 つ表示されるはずである．** さらに，**動作実験 1-1 で確認された，3.1.2 節の 1, 2, 3, 4, 5 で予想した結果と異なる動作について，その動作に変化がないかどうかを確認せよ（3.1.2 節の 1, 2, 3, 4, 5 で予想した結果と同じ動作になったはずである）.**

3.2.5　プロセッサの機能レベルシミュレーション

本実験では，最後に，追加設計後のプロセッサの動作を機能レベルシミュレーションにより確認する．機能レベルシミュレーションには，動作実験 1 で print_B.bin から生成した機能レベルシミュレーション用の命令メモリの Verilog HDL 記述 rom8x1024_sim.v と，追加設計後のプロセッサの Verilog HDL 記述一式を使用する．

機能レベルシミュレーションを行う前に，プロセッサのトップレベル記述mips_de2-115/MIPS/cpu.v をシミュレーション用の記述に変更し，機能レベルシミュレーション用のソースにしておく必要がある．**下記の 1, 2, 3, 4, 5, 6 の手順で，ソース** mips_de2-115/MIPS/cpu.v **の記述を変更せよ.**

1. cpu.v の 70 行目周辺，動作実験用の include 文をコメントアウトする．
2. cpu.v の 65 行目周辺，機能レベルシミュレーション用 include を有効にする．

3. cpu.v の 320 行目周辺，動作実験用の ROM の実体化を数行コメントアウトする．
4. cpu.v の 315 行目周辺，機能レベルシミュレーション用の ROM の実体化を有効にする．
5. cpu.v の 340 行目周辺，動作実験用の RAM の実体化を数行コメントアウトする．
6. cpu.v の 335 行目周辺，機能レベルシミュレーション用の RAM の実体化を有効にする．

cpu.v の変更後，rom8x1024_sim.v をディレクトリ mips_de2-115/MIPS にコピーし，ディレクトリ mips_de2-115/MIPS に cd して，EDA ツールを用いた論理回路設計の 3.2 節を参考に「vsim test_cpu.v」により機能レベルシミュレーションを行う．なお，**機能レベルシミュレーション後，次の実験課題で再び論理合成が行えるように，cpu.v の記述を元にもどしておくこと．**

4 シングルサイクル RISC プロセッサの設計「中級編」

本実験では，シングルサイクル RISC プロセッサの設計と動作実験を行う．第 2 週目の実験では，プロセッサの動作実験と，プロセッサの**ジャンプ命令（j）**と**即値符号なし・セット・オン・レス・ザン命令（sltiu）**，**ブランチ・オン・ノットイコール命令（bne）**，と**ロード・ワード（lw）**についての追加設計を行う．また，クロスコンパイラを用いたプログラム開発についての実験も行う．以下では，ジャンプ命令についての動作実験 2-1 ならびに追加設計 2，動作実験 2-2 について述べる

4.1 マシン・コードの動作実験 2-1（文字の繰り返し出力 1）

プロセッサの動作実験 2-1 では，ディスプレイに文字 'B' を繰り返し表示する MIPS マシン・コード print_B_while.bin と，それを実行するプロセッサとして追加設計 1 で完成させたプロセッサを FPGA 上に実現し，その動作を確認する．

本動作実験では，実験 1-2 で完成させたジャンプ命令（jump：j）が未実装なプロセッサにおいて，その命令を含む簡単な機械語のマシン・コードを実行すると，どのような動作をするかを観察する．本実験で観察した結果は，次のプロセッサの追加設計 2 において，j が正しく動くプロセッサを完成させた後，動作比較の対象として用いる．

4.1.1 MIPS マシン・コードからのメモリ・イメージファイルの作成

本実験では，まず，MIPS マシン・コードを命令メモリのメモリ・イメージファイルに変換する．ここでは，MIPS マシン・コードの例として print_B_while.bin [3] を使用する．変換には，変換プログラム bin2v を使用する．**EDA ツールの環境設定を行ったのち，「bin2v print_B_while.bin」で， MIPS マシン語プログラムからメモリ・イメージファイルを作成する．**この変換により，論理合成用のメモリ・イメージファイル rom8x1024_DE2.mif と，機能レベルシミュレーション用の Verilog HDL 記述 rom8x1024_sim.v が得られる．

なお，本実験で使用する MIPS マシン・コード print_B_while.bin は，正しいプロセッサ（ジャンプ命令 j が実装済みのプロセッサ）で動作させると，次の 1, 2, 3, 4 のような動作をする命令列を含んだ，バイナリファイルである．

[3] http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k02_j/print_B_while.bin

1. データメモリ（RAM）の 0x0300 番地に 0 を格納
   ```
   addiu $s2, $s0, 0x0300
   sw $s0, 0x0000($s2)
   ```

2. RAM の 0x0304 番地に 2 を格納
   ```
   addiu $s3, $s0, 0x0304
   addiu $s2, $s0, 0x0002
   sw $s2, 0x0000($s3)
   ```

3. RAM の 0x0300 番地に 1 を上書き
   ```
   addiu $s3, $s0, 0x0300
   addiu $s2, $s0, 0x0001
   sw $s2, 0x0000($s3)
   ```

4. PC = 0x040002c 番地の命令にジャンプ
   ```
   j 0x040002c
   ```

4.1.2 命令メモリに格納される命令列の確認

本実験では，次に，命令メモリに格納される命令列の確認を行う．この確認には，bin2v により生成された機能レベルシミュレーション用の Verilog HDL 記述 rom8x1024_sim.v を使用する．図 10 に rom8x1024_sim.v の一部を示す．

```
<省略>
case (word_addr)
 <省略>
 10'h00b: data = 32'h24020300; // 0040002c: ADDIU, REG[2]<=REG[0]+768(=0x00000300);     ここが PC=0x002c の命令
 10'h00c: data = 32'hac400000; // 00400030: SW, RAM[REG[2]+0]<=REG[0];
 10'h00d: data = 32'h24030304; // 00400034: ADDIU, REG[3]<=REG[0]+772(=0x00000304);
 10'h00e: data = 32'h24020002; // 00400038: ADDIU, REG[2]<=REG[0]+2(=0x00000002);
 10'h00f: data = 32'hac620000; // 0040003c: SW, RAM[REG[3]+0]<=REG[2];
 10'h010: data = 32'h24030300; // 00400040: ADDIU, REG[3]<=REG[0]+768(=0x00000300);
 10'h011: data = 32'h24020001; // 00400044: ADDIU, REG[2]<=REG[0]+1(=0x00000001);
 10'h012: data = 32'hac620000; // 00400048: SW, RAM[REG[3]+0]<=REG[2];
 10'h013: data = 32'h0810000b; // 0040004c: J, PC<=0x0010000b*4(=0x0040002c);     ここが 命令メモリ 0x013 の命令
 <省略>
endcase
<省略>
```

図 10: rom8x1024_sim.v の一部

図 10 の case ブロック内の最後の記述は，本実験で設計するプロセッサの命令メモリの 0x013 番地に機械語 0x0810000b が格納されることを表している．また，この命令は実際の MIPS では 0x0040004c に格納され，命令名は j，PC に 0x040002c をセットする命令であることを表している．

print_B_while.bin から生成された rom8x1024_sim.v または，図 10 の Verilog HDL 記述を解析し，以下の 1 について答えよ．なお，j はジャンプ命令である．

1. プロセッサが PC=0x004c の命令を実行することにより，PC に格納される値と，それが表す命令メモリの番地を予想せよ．

4.1.3 論理合成

本実験では，次に，j 命令が未実装なプロセッサならびに命令メモリ，その他周辺回路の論理合成を行う．論理合成には，bin2v により生成された論理合成用のメモリ・イメージファイル rom8x1024_DE2.mif と実験 1-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．**メモリ・イメージファイル** rom8x1024_DE2.mif **をディレクトリ** mips_de2-115 **にコピーし，ディレクトリ** mips_de2-115 **に** cd **して，「**quartus_sh --flow compile DE2_115_Default**」で論理合成を行う．**論理合成が完了すると，ディレクトリ mips_de2-115 内に FPGA にダウンロード可能なプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof が生成される．

4.1.4 FPGA を用いた回路実現

本実験では，次に，j 命令が未実装なプロセッサの実際の動作を観察する．観察した結果は，次のプロセッサの追加設計 2 において，j が正しく動くプロセッサを完成させた後，動作比較の対象として用いる．ここでは，論理合成により生成されたプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof を， Quartus II を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させる．**スライドスイッチ SW0, SW1 をともに 1 にし，プロセッサへのクロック供給を手動モードにする．**今回プロセッサが実行するマシン・コード print_B_while.bin はディスプレイ下部に文字 'B' を繰り返し表示するプログラムである．**KEY3 を数回押しクロックパルスを送り，プロセッサに PC=0x0000 番地の命令から 25 個程度の命令を実行させ，ディスプレイ下部に文字 'B' が繰り返し表示されるかどうかを確認せよ（ディスプレイ下部に文字は 1 つしか表示されないはずである．**ディスプレイ上部にはプロセッサ内部の主な信号線の現在の値が表示されている．

図 11 に動作実験 2-1 のプロセッサのブロック図を示す．

4.1.2 節の 1 で予想した結果と同じ正しい動作かどうかを確認せよ（予想と異なり，正しくない動作のはずである）．この結果から，プロセッサが，j 命令を正しく実行できていないことが分かる．次の実験 2-2 では，プロセッサの追加設計を行い，プロセッサ内部で行われるデータ転送や演算などを制御するメイン制御回路を，これらの命令に対応したものにする．

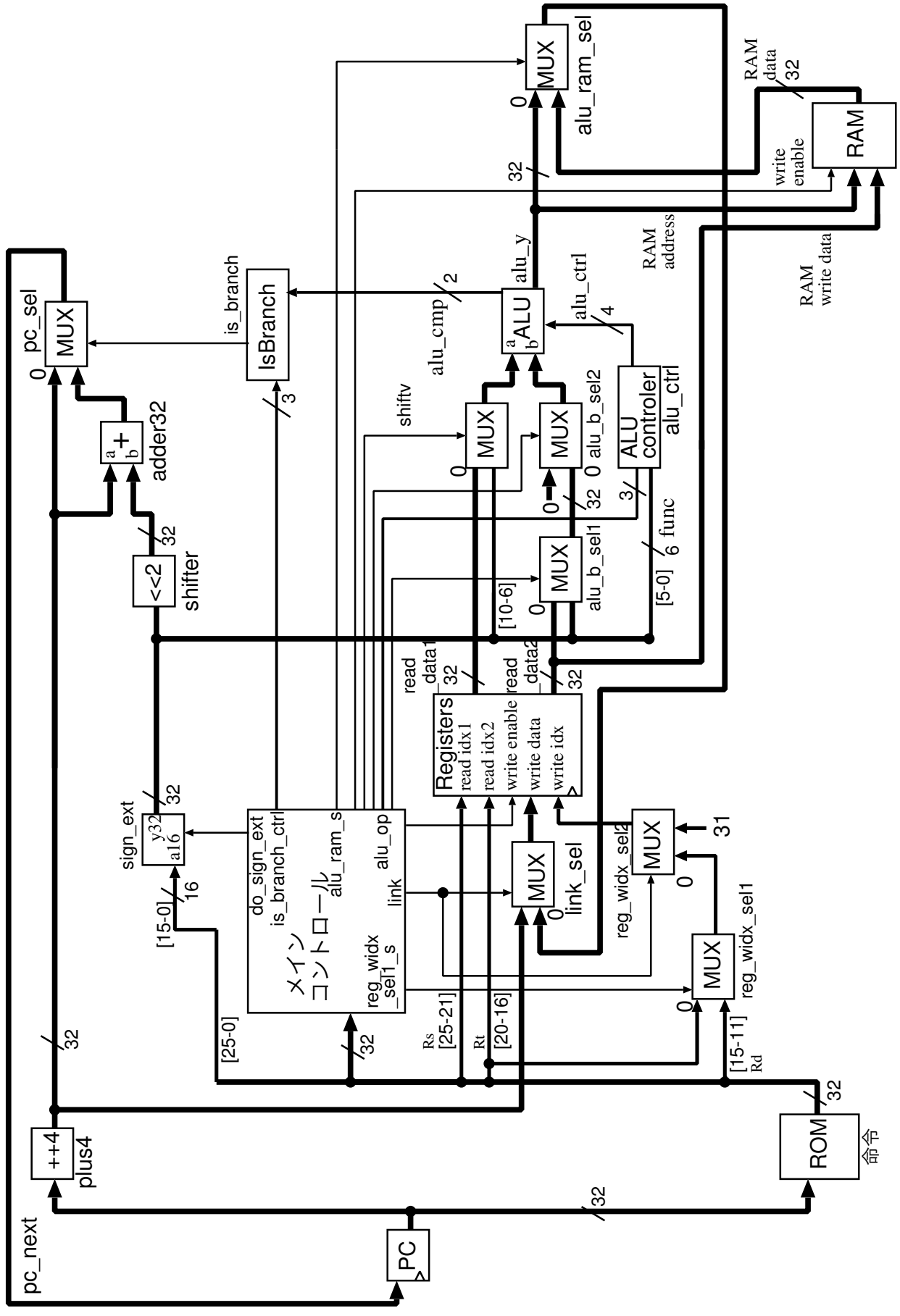


図 11: 動作実験 2-1 のプロセッサのブロック図

4.2 プロセッサの追加設計 2（j 命令）と動作実験 2-2

本実験では，プロセッサの追加設計と動作実験を行う．ここでは，j 命令が未実装なプロセッサを例とし，追加設計を行い，j 命令が正しく実行されるプロセッサを完成させる．また，その動作を実際に動作させて観察する．

4.2.1 j 命令のためのジャンプ・セレクト・モジュールの追加設計

本実験では，まず，動作実験 2-1 で動作を確認した j 命令が未実装なプロセッサに対して追加設計を行う．ここでは，プロセッサの最上位階層の Verilog HDL 記述 `cpu.v` を使用する．`cpu.v` は，ディレクトリ `mips_de2-115` のサブディレクトリ `MIPS` にある．**追加設計 2 のヒント (1)〜(4) の周辺を，下記の 1, 2, 3, 4 の手順で適切なものに変更せよ．**

- j 命令は,「命令の address フィールドに直接書かれている値」 × 4 を PC に格納する命令である．
- j 命令のためのジャンプ・セレクト・モジュールを含むプロセッサのブロック図を，図 12 に示す．破線で囲まれた，未実装，追加設計 2 と書かれた部分が j 命令のためのジャンプ・セレクト・モジュールである．MUX `jp_sel` は，2 入力 1 出力の Multiplexer, 選択回路であり，その 2 つの入力信号のうち，0 のラベルが付けられている方が，選択信号 `jp` が 0 の時に出力される信号である．以下では，このジャンプ・セレクト・モジュールをプロセッサの最上位階層の記述に追加する．

1. 追加設計 2 のヒント (1)：jp_sel の入出力ワイヤの宣言
   - **図 12 のワイヤ `jp_sel_d0`, `jp_sel_d1`, `jp_sel_s`, `jp_sel_y` に対応する，同名のワイヤを宣言する．**
2. 追加設計 2 のヒント (2)：32-bit, 32-bit 入力, 32-bit 出力のセレクタを実体化
   - **図 12 のモジュール `jp_sel` に対応する，同名のモジュールを実体化する．**
3. 追加設計 2 のヒント (3)：jp_sel の出力 jp_sel_y の pc_next への接続
   - モジュール `jp_sel` の出力 jp_sel_y を図 12 のように pc_next に接続する．
   - **古い接続 assign pc_next = pc_sel_y; は消去する．**
4. 追加設計 2 のヒント (4)：jp_sel の入力 jp_sel_d0, jp_sel_d1, jp_sel_s の接続
   - モジュール `jp_sel` の入力 jp_sel_d0, jp_sel_d1, jp_sel_s を，それぞれ図 12 のように pc_sel_y, sh_j_y, jp に接続する．

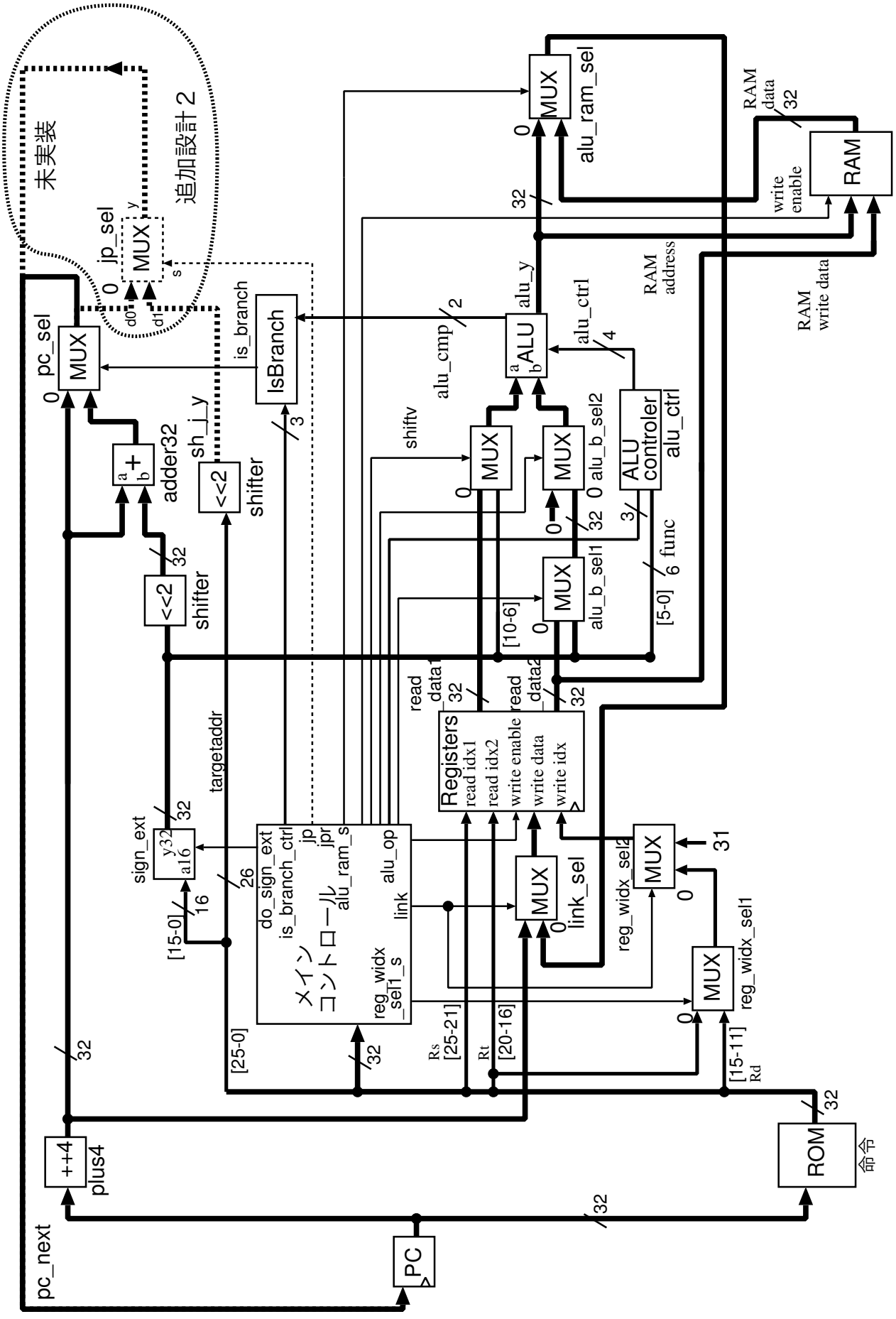


図 12: j 命令のためのジャンプ・セレクト・モジュールを含むプロセッサ

4.2.2 j 命令のためのメイン制御回路の追加設計

本実験では，次に，動作実験 2-1 で動作を確認した j 命令が未実装なプロセッサに対して，追加設計を行う．ここでは，プロセッサのメイン制御回路の Verilog HDL 記述 `main_ctrl.v` を使用する．`main_ctrl.v` は，ディレクトリ `mips_de2-115` のサブディレクトリ `MIPS` にある．**ソースファイル `main_ctrl.v` 中のコメント，追加設計 2 のヒント (1)〜(3) の周辺を，下記の 1, 2, 3 の手順で適切に変更せよ．**

- j 命令実行時のプロセッサ内の信号の流れを図 13 に示す．緑線（薄い灰色）とラベル付けされた信号線が j 命令の実行に関わっている．以下では，信号の流れがブロック図のようになるように，赤線（(2),(3) の番号付き）とラベル付けされた制御信号を適切に設定する．制御信号に付いた (2),(3) の番号と，ヒントの番号の間には対応関係がある．

1. 追加設計 2 のヒント (1)：J 形式の命令 j の追加，命令コードの定義
   - j の命令操作コードが「　　」であることから，**記述「　　」「　　」に変更する．**
2. 追加設計 2 のヒント (2)：J 形式の命令 j の追加，jp_sel モジュールへの制御信号の記述
   - jp_sel はジャンプ用のモジュールである（図 13，参考文献 [9] pp.295-299，pp.287，または参考文献 [8] pp.280-284, pp.271）．
   - j 命令はジャンプ命令なので，jp_sel への制御信号としては「　　」が適切である（ソース中の jp_sel に関するコメント (ヒントの数行上あたり)「// jump, J, JAL 用// MUX, jp_sel モジュールのセレクト信号// jp == 1'b0： jump しない場合の，次の PC の値を選択// jp == 1'b1： jump する場合の，次の PC の値を選択」より）．
   - **記述「　　」を「　　」に変更する．**
3. 追加設計 2 のヒント (3)：J 形式の命令 j の追加，レジスタファイルへの制御信号の記述
   - reg_write_enable はレジスタファイル registers の書き込み制御信号である（図 13，参考文献 [9] pp.290-299，または参考文献 [8] pp.274-284）．
   - j 命令は演算結果をレジスタに書き込む命令ではないので，制御信号 reg_write_enable の値として「　　」が適切である（ソース中の reg_write_enable に関するコメント (ヒントの数行上あたり)「// reg_write_enable // レジスタファイル registers の書き込み制御信号// reg_write_enable == 1'b0:書き込みを行わない// reg_write_enable == 1'b1：書き込みを行う」より）．
   - **記述「　　」を「　　」に変更する．**

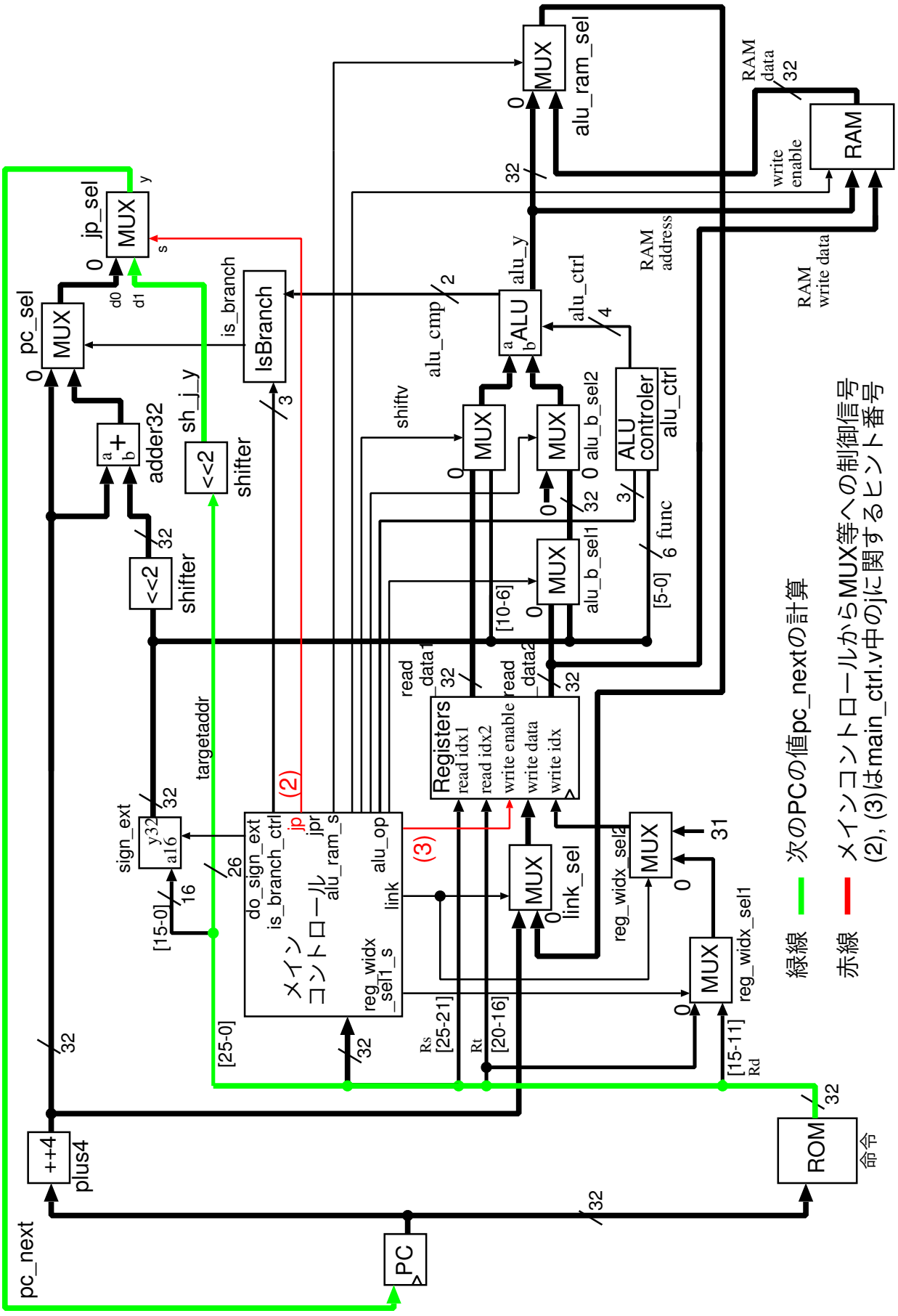


図 13: j 命令実行時のプロセッサ内の信号の流れ

4.2.3 論理合成

本実験では，次に，追加設計後のプロセッサならびに命令メモリ，その他周辺回路の論理合成を行う．論理合成には，追加設計後の main_ctrl.v と cpu.v，動作実験 2-1 で使用したその他プロセッサの Verilog HDL 記述一式，print_B_while.bin から生成したメモリ・イメージファイル rom8x1024_DE2.mif を使用する．

追加設計後の main_ctrl.v と cpu.v を，ディレクトリ mips_de2-115 のサブディレクトリ MIPS に置く．更に，ディレクトリ mips_de2-115 に cd し，pring_B_while.bin の rom8x1024_DE2.mif が，そこにあるのを確認してから，「quartus_sh --flow compile DE2_115_Default」で論理合成を行う．論理合成が完了すると，ディレクトリ mips_de2-115 内に FPGA にダウンロード可能なプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof が生成される．

4.2.4 FGPA を用いた回路実現

本実験では，次に，追加設計後のプロセッサの実際の動作を観察し，動作実験 2-1 で観察した結果との比較を行う．ここでは，論理合成により生成されたプロセッサなど回路一式のストリーム・アウト・ファイル DE2_115_Default.sof を，Quartus II を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させる．また，**スライドスイッチ SW0, SW1 をともに 1 にし，プロセッサへのクロック供給を手動モードにする．**プロセッサが実行するマシン・コード print_B_while.bin はディスプレイ下部に文字 'B' を繰り返し表示するプログラムである．**KEY3 を数回押しプロセッサにクロックパルスを送り，プロセッサに PC=0x0000 番地から 25 個程度の命令を実行させ，ディスプレイ下部に文字'B' が繰り返し表示されるかどうかを確認せよ（文字 'B' が繰り返し表示されるはずである．**また，**動作実験 2-1 で確認された，4.1.2 節の 1 で予想した結果と異なる動作について，その動作に変化がないかどうかを確認せよ（4.1.2 節の 1 で予想した結果と同じ動作になったはずである）．**

5 実験

実験 1-1 ディスプレイに文字 'B' を 1 つ表示する MIPS マシン・コード print_B.bin と，それを実行するプロセッサを FPGA 上に実現しその動作を確認せよ（動作実験 1-1）．本動作実験は，**3 章を参考に**，下記の 1, 2, 3, 4 の手順で行いなさい．

- 動作実験 1-1 の手順
  1. メモリイメージファイルの作成
     3.1.1 節を参考に，MIPS マシン・コード print_B.bin から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  2. 命令メモリに格納される命令列の確認
     3.1.2 節を参考に，命令メモリに格納される命令列を確認し，プロセッサの動作を予想する．
  3. 論理合成
     3.1.3 節を参考に，プロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     3.1.4 節を参考に，プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

なお，本動作実験で使用する MIPS マシン・コード print_B.bin と，プロセッサの Verilog HDL 記述一式は下記の URL からダウンロードできる．

MIPS マシン・コード:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k01_addiu_sw/print_B.bin
（実験 1-1 用の MIPS マシン・コード，バイナリファイル）

プロセッサの Verilog HDL 記述一式:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k01_addiu_sw/mips_de2-115.tar.gz
（**本実験をとおして完成させる未完成なプロセッサ**）

実験 1-1 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k01_addiu_sw_1_1/index.html

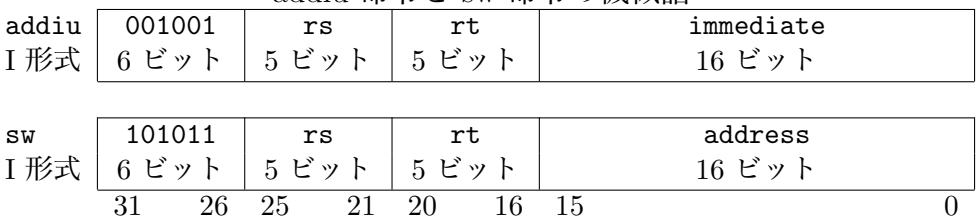
実験 1-2 動作実験 1-1 の addiu 命令と sw 命令が未実装なプロセッサについて，追加設計を行い，両命令を正しく実行するプロセッサを完成させなさい（追加設計 1）．また，そのプロセッサと動作実験 1-1 の `print_B.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 1-2）．本実験は，**3 章を参考に**，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- プロセッサの追加設計 1 の手順
  1. addiu 命令のためのメイン制御回路の追加設計
     3.2.1 節を参考に，プロセッサのメイン制御回路の追加設計を行う．
  2. sw 命令のためのメイン制御回路の追加設計
     3.2.2 節を参考に，プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 1-2 の手順
  3. 論理合成
     3.2.3 節を参考に，完成したプロセッサ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     3.2.4 節を参考に，完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．
  5. プロセッサの機能レベルシミュレーション
     3.2.5 節を参考に，プロセッサの動作を機能レベルシミュレーションにより確認する．

addiu 命令と sw 命令のアセンブリ言語

区分	命令	意味
算術演算	`addiu rt,rs,immediate`	`rt = rs + immediate`
データ転送	`sw rt,address(rs)`	メモリ `[rs + address] = rt`

addiu 命令と sw 命令の機械語



実験 1-2 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/`
`j2hard-mips/k01_addiu_sw_1_2/index.html`

実験 2-1 ディスプレイに文字 'B' を繰り返し表示する MIPS マシン・コード print_B_while.bin と，それを実行するプロセッサとして実験 1-2 で完成させたプロセッサを FPGA 上に実現しその動作を確認せよ（動作実験 2-1）．本動作実験は，**4 章を参考に**，下記の 1, 2, 3, 4 の手順で行いなさい．

- 動作実験 2-1 の手順
  1. メモリイメージファイルの作成
     4.1.1 節を参考に，MIPS マシン・コード print_B_while.bin から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  2. 命令メモリに格納される命令列の確認
     4.1.2 節を参考に，命令メモリに格納される命令列を確認し，プロセッサの動作を予想する．
  3. 論理合成
     4.1.3 節を参考に，実験 1-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     4.1.4 節を参考に，プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

なお，本動作実験で使用する MIPS マシン・コード print_B_while.bin は，下記の URL からダウンロードできる．

MIPS マシン・コード:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k02_j/print_B_while.bin
（実験 2-1 用の MIPS マシン・コード，バイナリファイル）

実験 2-1 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k02_j_2_1/index.html

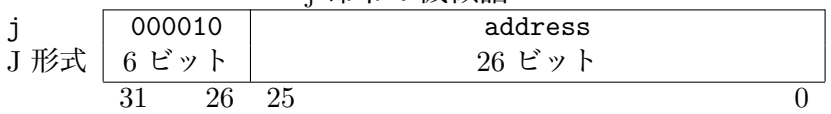
実験 2-2 動作実験 2-1 の j 命令が未実装なプロセッサについて，追加設計を行い，j 命令を正しく実行するプロセッサを完成させなさい（追加設計 2）．また，そのプロセッサと動作実験 2-1 の `print_B_while.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 2-2）．本実験は，**4 章を参考に**，下記の 1, 2, 3, 4 の手順で行いなさい．

- プロセッサの追加設計 2 の手順
  1. j 命令のためのジャンプ・セレクト・モジュールの追加設計
     4.2.1 節を参考に，プロセッサの最上位階層の記述に追加設計を行う．
  2. j 命令のためのメイン制御回路の追加設計
     4.2.2 節を参考に，プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 2-2 の手順
  3. 論理合成
     4.2.3 節を参考に，完成したプロセッサ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     4.2.4 節を参考に，完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

j 命令のアセンブリ言語

区分	命令	意味
ジャンプ	`j address`	`PC = address × 4`

j 命令の機械語



実験 2-2 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k02_j_2_2/index.html`

実験 3 図 14 に，MIPS 用にコンパイルすると，実験 1-1, 1-2 で使用した MIPS マシン・コード print_B.bin が得られる C 言語のソース print_B.c を示す．ソースの 1,2 行目は，それぞれ，プロセッサのデータメモリの 0x0300 番地と 0x0304 番地を指す define 文である．5 行目は，プロセッサのデータメモリの 0x0300 番地に 0x00000000 を格納する記述である．6,7 行目は，それぞれ，プロセッサのデータメモリの 0x0304, 0x0300 番地に 0x00000002, 0x00000001 を格納する記述である．

図 14 のソースから実験 1-1, 1-2 で使用したマシン・コードが生成されることをふまえ，**実験 2-1, 2-2 で使用した MIPS マシン・コード** print_B_while.bin **が生成される，元となった C 言語のソース** my_print_B_while.c **を作成せよ**（ヒント：print_B.c に 2 行追加）．

また，**作成した** my_print_B_while.c **を MIPS 用にクロスコンパイルし，MIPS マシン・コード** my_print_B_while.bin **を生成せよ．**クロスコンパイルには，「cross_compile.sh」を使用し，「cross_compile.sh my_print_B_while.c」で MIPS マシン・コードが得られる．更に，**生成された** my_print_B_while.bin **に対して，「**bin2v**」を使用し，メモリ・イメージファイル** rom8x1024_DE2.mif **を生成し，その内容が，実験 2-1 で使用したものと同じであるかどうかを確認せよ．**

```
1:    #define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
2:    #define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)
3:    main()
4:    {
5:      EXTIO_PRINT_STROKE = (unsigned int)0x00000000;
6:      EXTIO_PRINT_ASCII = (unsigned int)0x00000002;
7:      EXTIO_PRINT_STROKE = (unsigned int)0x00000001;
8:    }
```

図 14: print_B.c

実験 3 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/
j2hard-mips/k03_cross/index.html

実験 4-1 ディスプレイに 61 種類の文字を表示する C プログラム print_all_char.c と，それを実行するプロセッサとして実験 2-2 で完成させたプロセッサを FPGA 上に実現しその動作を確認せよ（動作実験 4-1）．本動作実験は，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- 動作実験 4-1 の手順
  1. クロスコンパイル
     C 言語プログラム print_all_char.c から，MIPS のマシン・コード print_all_char.bin を生成する．
  2. メモリイメージファイルの作成
     MIPS マシン・コード print_all_char.bin から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  3. 命令メモリに格納される命令列の確認
     命令メモリに格納される命令列を確認し，プロセッサの動作を予想する．
  4. 論理合成
     実験 2-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  5. FPGA を用いた回路実現
     プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

なお，本動作実験で使用する C プログラム print_all_char.c は，下記の URL からダウンロードできる．

C プログラム:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k04_sltiu_bne_lw/print_all_char.c
（実験 4-1 用の C プログラム）

実験 4-1 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k04_sltiu_bne_lw_4_1/index.html

実験 4-2 動作実験 4-1 の sltiu 命令と bne 命令，lw 命令が未実装なプロセッサについて，追加設計を行い，これら命令を正しく実行するプロセッサを完成させなさい（追加設計 3）．また，そのプロセッサと動作実験 4-1 の `print_all_char.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 4-2）．本実験は，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- プロセッサの追加設計 3 の手順
  1. sltiu 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行う．
  2. bne 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行う．
  3. lw 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 4-2 の手順
  4. 論理合成
     完成したプロセッサ，その他周辺回路の論理合成を行う．
  5. FPGA を用いた回路実現
     完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

sltiu 命令と bne 命令，lw 命令のアセンブリ言語

区分	命令	意味
条件判定	`sltiu rt,rs,immediate`	`rt = (rs < immediate) ? 1 : 0`
条件分岐	`bne rt,rs,address`	`PC = (rs ≠ rt) ? PC + 4 + address × 4 : PC + 4`
データ転送	`lw rt,address(rs)`	`rt = メモリ [rs + address]`

sltiu 命令と bne 命令，lw 命令の機械語

実験 4-2 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k04_sltiu_bne_lw_4_2/index.html`

実験 5-1 ディスプレイに文字列を表示する C プログラム my_print.c と，それを実行するプロセッサとして実験 4-2 で完成させたプロセッサを FPGA 上に実現しその動作を確認せよ（動作実験 5-1）．本動作実験は，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- 動作実験 5-1 の手順
  1. クロスコンパイル
     C 言語プログラム my_print.c から，MIPS のマシン・コード my_print.bin を生成する．
  2. メモリイメージファイルの作成
     MIPS マシン・コード my_print.bin から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  3. 命令メモリに格納される命令列の確認
     命令メモリに格納される命令列を確認し，プロセッサの動作を予想する．
  4. 論理合成
     実験 4-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  5. FPGA を用いた回路実現
     プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

なお，本動作実験で使用する C プログラム my_print.c は，下記の URL からダウンロードできる．

C プログラム:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k05_jal/my_print.c
（実験 5-1 用の C プログラム）

実験 5-1 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k05_jal_5_1/index.html

実験 5-2 動作実験 5-1 の jal 命令が未実装なプロセッサについて，追加設計を行い，jal 命令を正しく実行するプロセッサを完成させなさい（追加設計 4）．また，そのプロセッサと動作実験 5-1 の `my_print.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 5-2）．本実験は，下記の 1, 2, 3 の手順で行いなさい．

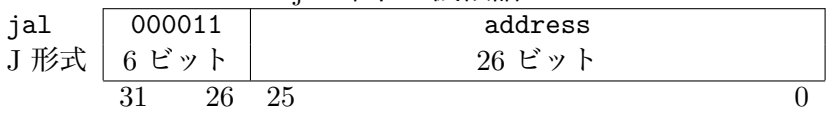
- プロセッサの追加設計 4 の手順
  1. jal 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 5-2 の手順
  2. 論理合成
     完成したプロセッサ，その他周辺回路の論理合成を行う．
  3. FPGA を用いた回路実現
     完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

jal 命令のアセンブリ言語

区分	命令	意味

手続きサポート（ジャンプ）	`jal address`	`PC = PC + 4`
		`ra = address × 4`
		（ra は 31 番目のレジスタ）

jal 命令の機械語



実験 5-2 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k05_jal_5_2/index.html`

実験 6-1 キーボードからの文字列入力を受けるC プログラム my_scan.c と，それを実行するプロセッサとして実験 5-2 で完成させたプロセッサを FPGA 上に実現しその動作を確認せよ（動作実験 6-1）．本動作実験は，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- 動作実験 6-1 の手順
  1. クロスコンパイル
     C 言語プログラム my_scan.c から，MIPS のマシン・コード my_scan.bin を生成する．
  2. メモリイメージファイルの作成
     MIPS マシン・コード my_scan.bin から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  3. 命令メモリに格納される命令列の確認
     命令メモリに格納される命令列を確認し，プロセッサの動作を予想する．
  4. 論理合成
     実験 5-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  5. FPGA を用いた回路実現
     プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

なお，本動作実験で使用する C プログラム my_scan.c は，下記の URL からダウンロードできる．

C プログラム:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k06_jr/my_scan.c
（実験 6-1 用の C プログラム）

実験 6-1 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k06_jr_6_1/index.html

実験 6-2 動作実験 6-1 の jr 命令が未実装なプロセッサについて，追加設計を行い，jr 命令を正しく実行するプロセッサを完成させなさい（追加設計 5）．また，そのプロセッサと動作実験 6-1 の `my_scan.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 6-2）．本実験は，下記の 1, 2, 3, 4 の手順で行いなさい．

- プロセッサの追加設計 5 の手順
  1. jr 命令のためのジャンプ・レジスタ・セレクト・モジュールの追加設計
     プロセッサの最上位階層の記述に追加設計を行う．
  2. jr 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 6-2 の手順
  3. 論理合成
     完成したプロセッサ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

jr 命令のアセンブリ言語

区分	命令	意味
手続きサポート（ジャンプ）	`jr rs`	`PC = ra`（`ra` は 31 番目のレジスタ）

jr 命令の機械語

`jr`	`000000`	`11111`	`00000`	`00000`	`00000`	`001000`
R 形式	6 ビット	5 ビット	5 ビット	5 ビット	5 ビット	6 ビット
	31 26	25 21	20 16	15 11	10 6	5 0

実験 6-2 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k06_jr_6_2/index.html`

実験 7 3 から「キーボードから入力された数」までの数のうち，素数であるもののみをディスプレイに次々と表示する処理を，C プログラムと実験 6-2 で完成させたプロセッサにより実現せよ．本実験は，下記の 1, 2, 3, 4, 5, 6 の手順で行いなさい．

- 実験 7 の手順
  1. クロスコンパイル
     sosuu.c から，MIPS のマシン・コード sosuu.bin を生成する．
  2. メモリイメージファイルの作成
     sosuu.bin から，メモリ・イメージファイルを作成する．
  3. 命令メモリに格納される命令列の確認
     (a) 命令メモリの 0x082 番地の命令は，実験 6-2 で完成させたプロセッサでは未実装な命令である．この命令はどのような命令か調査せよ．
     (b) 3 (a) の命令は，sosuu.c 中の関数 sosuu_check() の処理を行う命令の一つである．具体的に，どの記述に対応しているか予想せよ．
  4. 論理合成
     実験 6-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  5. FPGA を用いた回路実現
     プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．
     (a) HELLO, NUM= と表示されたら，キーボードから “20” と入力し，その結果を観察せよ．
     (b) 5 (a) で観察された正しくない動作の原因は，3(a),(b) のためである．この問題を解決する方法を 2 つ考えよ．
  6. C プログラム sosuu.c の変更（おそらく，2 つの解決法のうちの 1 つ）
     3 から「キーボードから入力された数」までの数のうち，素数であるもののみをディスプレイに次々と表示する処理が正しく行えるように sosuu.c を修正し，実際にその動作を確認する．

なお，本動作実験で使用する C プログラム sosuu.c は，下記の URL からダウンロードできる．

C プログラム:
http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k07_sosuu/sosuu.c
(実験 7 用の C プログラム)

実験 7 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k07_sosuu/index.html

実験 8 キーボードからステッピングモーターを制御する処理を，C プログラムと実験 6-2 で完成させたプロセッサにより実現せよ．本実験は，下記の 1, 2, 3, 4, 5 の手順で行いなさい．

- 実験 8 の手順

  1. クロスコンパイル
     C 言語プログラム `motor.c` から，MIPS のマシン・コード `motor.bin` を生成する．
  2. メモリイメージファイルの作成
     MIPS マシン・コード `motor.bin` から，プロセッサの命令メモリのメモリ・イメージファイルを作成する．
  3. 論理合成
     実験 6-2 で完成させたプロセッサならびに作成したメモリイメージ，その他周辺回路の論理合成を行う．
  4. FPGA を用いた回路実現
     プロセッサなど回路一式を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する（本プログラムはキーボードからの制御はできない)．
  5. モーター制御プログラムの作成
     キーボードからモーターを制御するプログラムを自由に作成し，実際にその動作を確認する．

なお，本動作実験で使用する C プログラム `motor.c` は，下記の URL からダウンロードできる．

C プログラム:
`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k08_motor/motor.c`
(実験 8 用の C プログラム)

実験 8 に関する追加の情報がある場合は下記の URL に掲載するので，その時はここを参照すること．

`http://www.ice.nuie.nagoya-u.ac.jp/jikken/hard/j2hard-mips/k08_motor/index.html`

6　実験レポートについて

実験 1，実験 2，実験 3，実験 4，実験 5，実験 6，実験 7，実験 8 について，実験の概要，使用機器ならびにソフトウェア，実験の手順，実験の各段階の説明，動作実験の結果，実験の考察を，文章ならびに図，表を交えてまとめなさい．

参考文献

[1] http://www.vdec.u-tokyo.ac.jp/ 東京大学大規模集積システム設計教育研究センター（VDEC）．

[2] VDEC 監修, 浅田邦博. ディジタル集積回路の設計と試作. 培風館, 2000.

[3] 深山正幸, 北川章夫, 秋田純一, 鈴木正國. HDL による VLSI 設計 – Verilog-HDL と VHDL による CPU 設計 –. 共立出版株式会社, 1999.

[4] 白石肇. わかりやすいシステム LSI 入門. オーム社, 1999.

[5] 桜井至. HDL によるデジタル設計の基礎. テクノプレス, 1997.

[6] James O. Hamblen and Michael D. Furman. Rapid Prototyping of Digital Systems. Kluwer Academic Publishers, 2000.

[7] パターソン&ヘネシー 著, 成田光彰 訳. コンピュータの構成と設計（上巻）第 3 版. 日経 BP 社, 1999.

[8] パターソン&ヘネシー 著, 成田光彰 訳. コンピュータの構成と設計（下巻）第 3 版. 日経 BP 社, 1999.

[9] パターソン&ヘネシー 著, 成田光彰 訳. コンピュータの構成と設計（上巻）第 4 版. 日経 BP 社, 2009.