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] の 2 , 3 , 4 章，本指導書の 1 , 2 , 3 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 , 4 章，本指導書の 4 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 , 4 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．さらに，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 6 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）
- 実験 9 （C プログラムの動作実験 9，整数乗算命令 mult の追加）… 発展課題

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 と 6 章では，第 3 週目，第 4 週目に行う RISC プロセッサの設計「上級編」,「応用編」の実験課題について説明する．7 章では，実験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，濱口毅
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: 2018 年 9 月 10 日
第 1.40 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] 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．

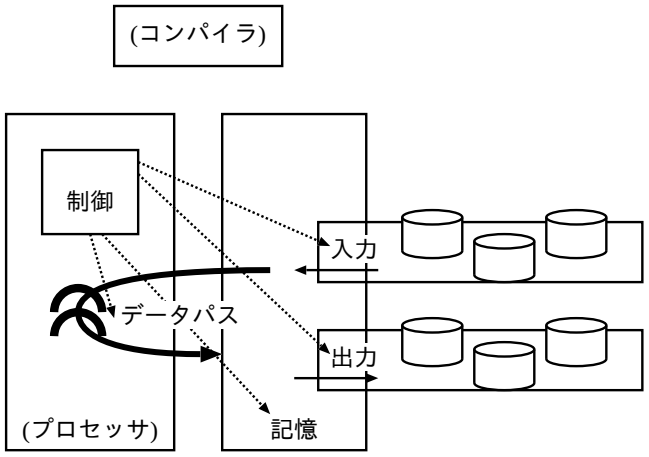


図 1: コンピュータの標準的な構成（参考文献 [7] 図 1.5 より）

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] 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: 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`

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．

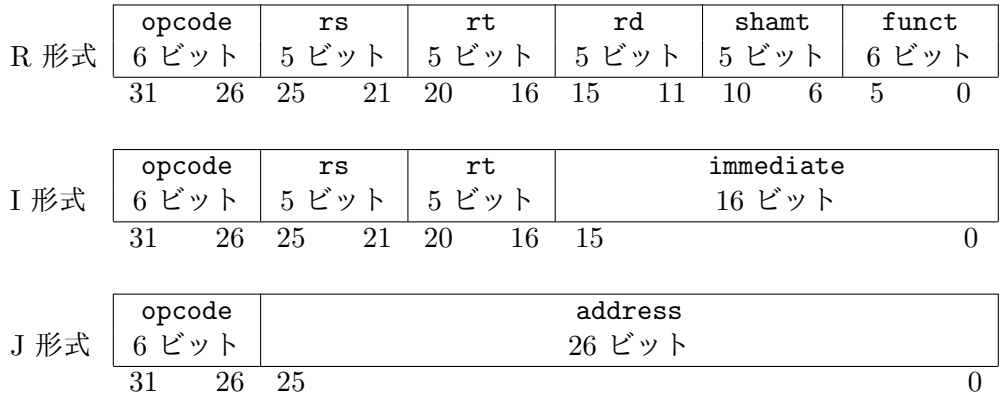


図 2: MIPS の命令のフィールド構成

R 形式の命令は 6 個のフィールド opcode, rs, rt, rd, shamt, funct から構成される．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．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．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，2 個のフィールド opcode, address から構成されている．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．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．

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．

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`

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，

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

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 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

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．

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

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

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．

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

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

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

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 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概要

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を設計する．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・サイクルで実行される**シングルサイクル方式**である．参考文献 [7] のシングルサイクルプロセッサの構成に基づき，RISC プロセッサの一つである MIPS の簡略化版であり，約 40 種類の命令を実行可能である．

2.2　設計環境の準備

プロセッサの設計は，ICE の Linux マシンにインストールされた Altera 社の EDA ツールを使用する．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　実験用コンピュータの構成

本実験では，図 3 に示す，Altera 社の DE2-115 ボードを使用する．設計したプロセッサは，DE2-115 ボードに搭載された FPGA で動作させる．

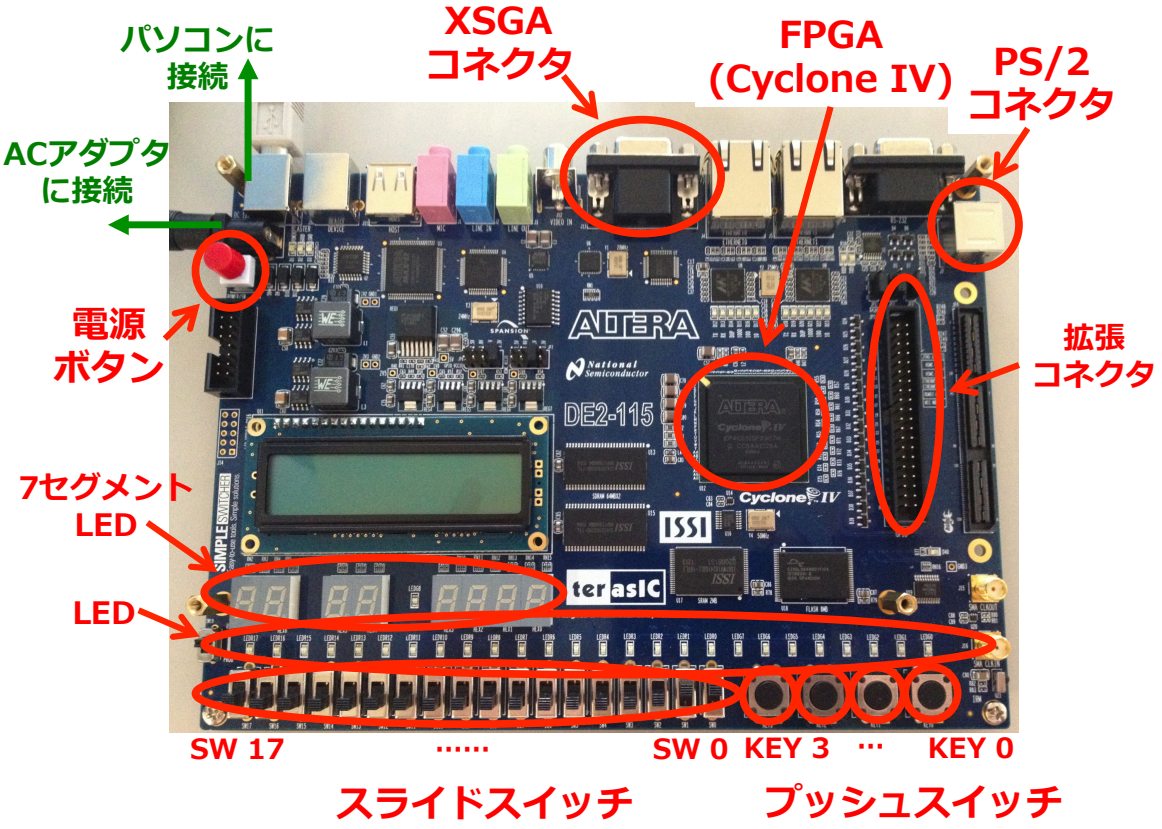


図 3: Altera 社 DE2-115 ボード

DE2-115 ボードに，液晶ディスプレイ，キーボードを接続して，動作実験用のコンピュータを構成する．図 4 に，動作実験用コンピュータの全体構成を示す．このコンピュータの構

成は，図 1 に示した，一般的なコンピュータと対応している．すなわち，キーボードと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，MIPS をターゲットプロセッサとする gcc を使用する．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，設計するプロセッサのマシン・コードに変換する．

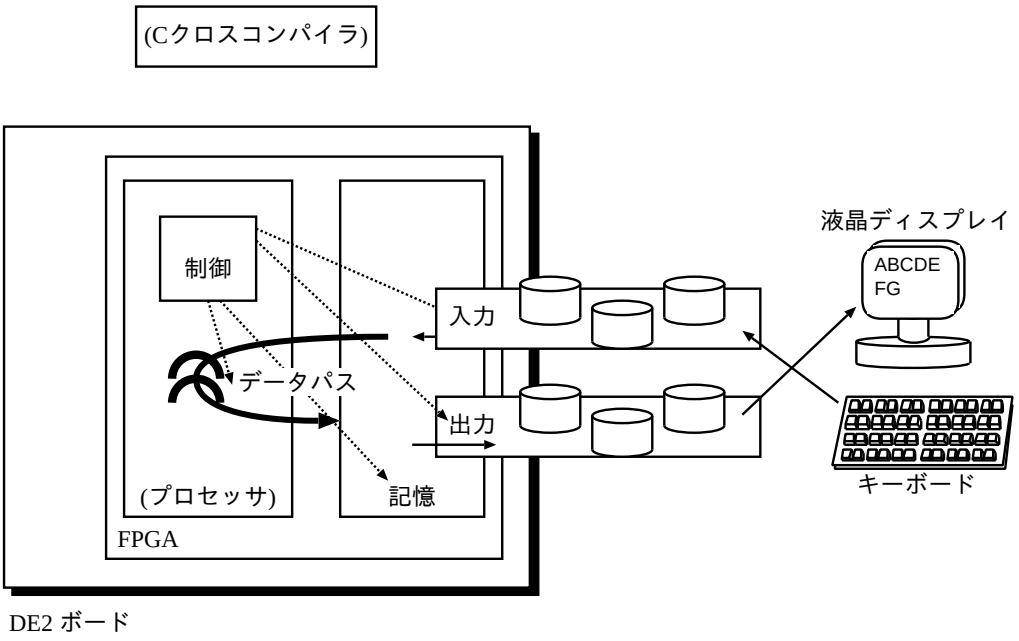


図 4: 動作実験用コンピュータの全体構成

動作実験用コンピュータの内部構成を図 5 に示す．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 内では，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．FPGA 上の CPU で動作するプログラムからディスプレイやキーボードを制御する場合には，それぞれ，キャラクタディスプレイインタフェース API とキーボードインタフェース API を使用する．

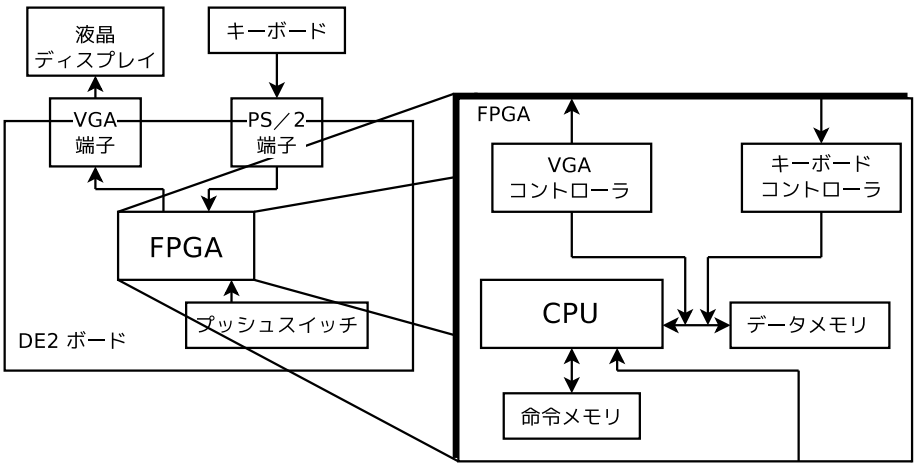


図 5: 動作実験用コンピュータの内部構成

2.4 実験の流れ

本実験では，以下の実験を通して，未完成のプロセッサに対して，徐々に機能を追加し，繰返し処理，分岐処理，関数呼出し，ポインタ，配列，符号なし整数の加減算ができるプロセッサを完成させる．最後に，完成させたプロセッサを使い，素数計算とステッピングモーター制御の実験を行う．

プロセッサの設計，追加実装

- 即値符号なし整数加算命令，ストア・ワード命令を正しく実行できるようにする（実験 1-1，1-2）
- ジャンプ命令を正しく実行できるようにする（実験 2-1，2-2）
- C クロスコンパイラを使いはじめる（実験 3）
- 即値符号なし・セット・オン・レス・ザン命令，ブランチ・ノット・イコール命令，ロード・ワード命令を正しく実行できるようにする（実験 4-1，4-2）
- ジャンプ・アンド・リンク命令を正しく実行できるようにする（実験 5-1，5-2）
- ジャンプ・レジスタ命令を正しく実行できるようにする（実験 6-1，6-2）

C 言語プログラミング

- 文字列をディスプレイに出力する my_print() 関数と，キーボードからの入力を受け付ける my_scan() 関数を使用して，素数計算プログラムを作成する（実験 7）
- ステッピングモータ制御プログラムを作成する（実験 8）

2.5 未完成なプロセッサの概要

実験の最初に提供される未完成なプロセッサは， 8 個の命令と，2 個のハードウェアモジュールが未実装の状態である．実装済の命令と未実装の命令は，以下の通りである．

実装済みの命令

- R 形式の命令：ADD, ADDU, SUB, SUBU, AND, OR, NOR, XOR, SLL, SRL, SLLV, SRLV, SRA, SLTU, JALR, SLR
- I 形式の命令：ADDI, ANDI, ORI, XORI, BEQ, BGEZ, BLEZ, BGTZ, BLTZ, BGEZAL, BLTZAL, SLTI, LUI

未実装の命令（本実験で追加実装する命令）

- R 形式の命令：JR
- I 形式の命令：LW, SW, ADDIU, BNE, SLTIU
- J 形式の命令：J, JAL

2.6 各実験の作業手順

プロセッサの各実験は，下記の 1 から 5 の手順で行う．

1. cross_compile.sh を使用して，C 言語のソース・プログラムを対象プロセッサで動作する MIPS マシン・コードに変換する（実験 3 以降）
2. bin2v を使用して，MIPS マシン・コードから命令メモリ（図 5）のメモリイメージファイルに変換する
3. Quartus II を使用して，プロセッサと周辺回路を論理合成する
4. Quartus II を使用して，ストリーム・アウトプット・ファイル（ sof ファイル）を DE2-115 ボードにダウンロードする
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プロセッサでプログラムを実行する

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，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．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．

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

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）

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．

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．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．

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．

本実験で使用する MIPS マシン・コード `print_B.bin` と，プロセッサの Verilog HDL 記述一式（本実験を通して完成させる未完成なプロセッサ）は，実験 Web ページからダウンロードできる．

本実験では，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つプロセッサを論理合成するで，

変換元のマシン・コードを実行するプロセッサを実現できる． MIPS マシン・コードの例として print_B.bin を使用する．また，変換プログラムとして bin2v を使用する．

2.2 節に従って EDA ツールの環境設定を行った後，「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 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．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．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．

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 が実装されたプロセッサ）で実行する，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

次に，プロセッサの命令メモリに格納される命令列を確認する．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．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 ビットアドレスと，そこに格納される 32 ビット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，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 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．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．

つまり，本実験で設計するプロセッサでは，プログラムカウンタ PC=(0x0040 h3 h2 h1 h0) が指す命令は，命令メモリでは，アドレスの上位 0x0040 を除き，かつ，アドレスを右に 2 ビットシフト（(0x h3 h2 h1 h0) << 2 ）したアドレスに格納される．例えば，プロセッサの PC が 0x0040002c を指している場合，上位の 0x0040 を除き，2 ビット右シフトした命令メモリのアドレス 0x000b 番地に格納された命令が実行される．

```
<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

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 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．RAMRDAT, 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 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．

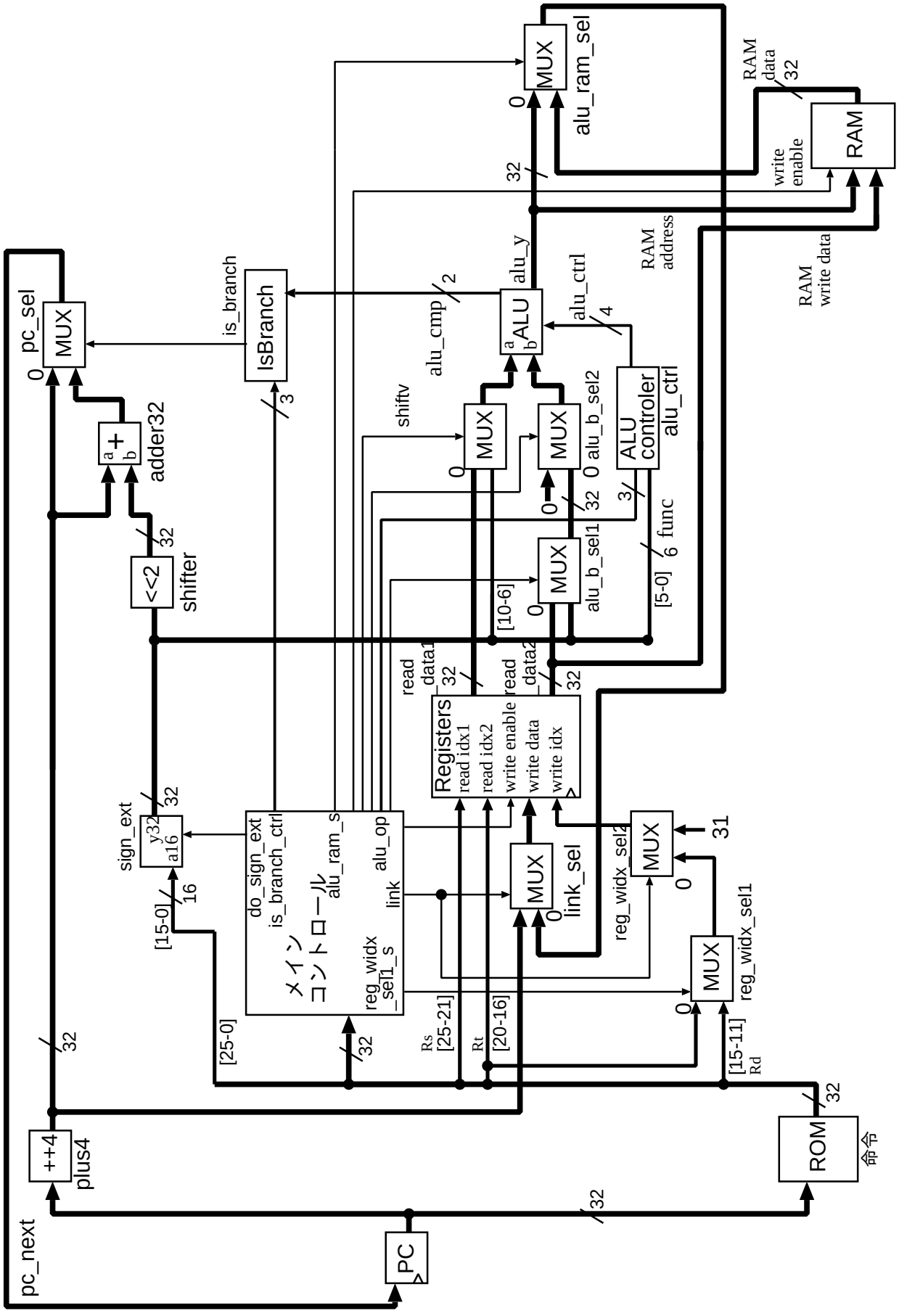


図 7: 動作実験 1-1 のプロセッサのブロック図

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）
RAMRDAT	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）

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， addiu 命令と sw 命令が未実装なプロセッサに対して追加設計を行い，両命令が正しく実行されるプロセッサを完成させる．

3.2.1 addiu 命令のためのメイン制御回路の追加設計

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対して追加設計を行う．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．

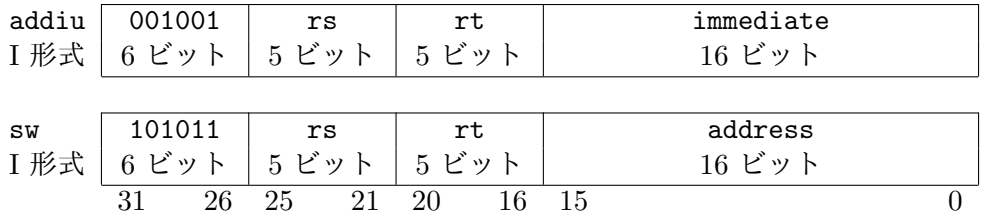
実験 1-2 動作実験 1-1 の addiu 命令と sw 命令が未実装なプロセッサについて，追加設計を行い，両命令を正しく実行するプロセッサを完成させなさい（追加設計 1）．また，そのプロセッサと動作実験 1-1 の `print_B.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 1-2）．本実験は，3 章を参考に，下記の 1, 2, 3, 4, 5 の手順で行うこと．

- 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



- 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 [7] p.246）．MUX は，2 入力 1 出力の Multiplexer（選択回路）である．選択信号が 0 の時に，2 つの入力信号のうち，0 のラベルが付けられている信号が出力される．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．

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．

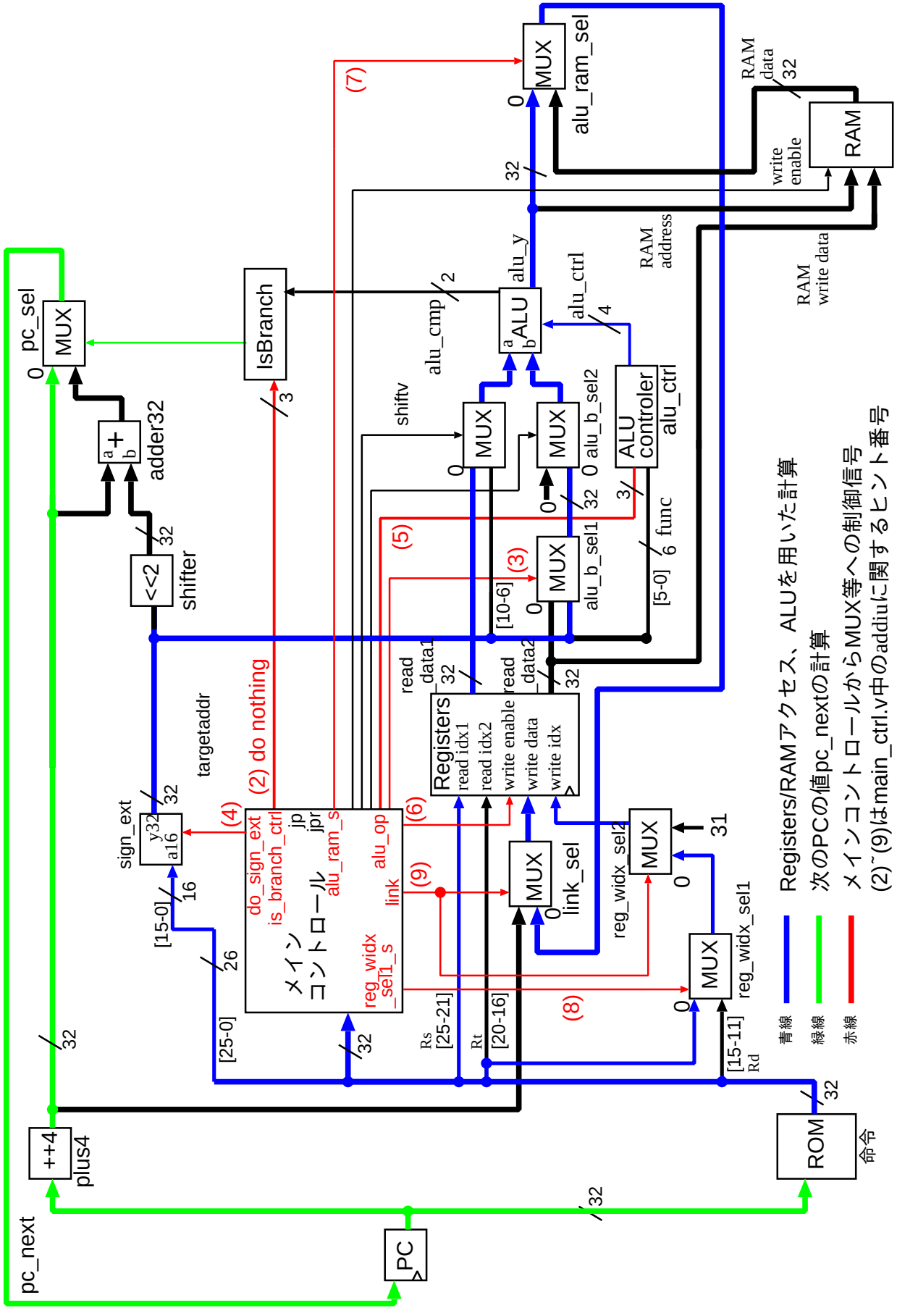


図 8: addiu 命令実行時のプロセッサ内の信号の流れ

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 [7] pp.247–259）.
   - 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「// 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 [7] p.246）.
   - 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「// 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 化するときに，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 [7] pp.250–259）.
   - 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「 // 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] とは，制御コードが少し異なる.

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 [7] pp.250–259）.
- 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「// 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 [7] pp.250–259）.
   - 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 [7] pp.250–259）.
   - 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 [7] pp.250–259）.
   - 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

次に，sw 命令の追加設計を行う．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「// 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 [7] pp.247–259）．
   - 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「// 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

   - 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．

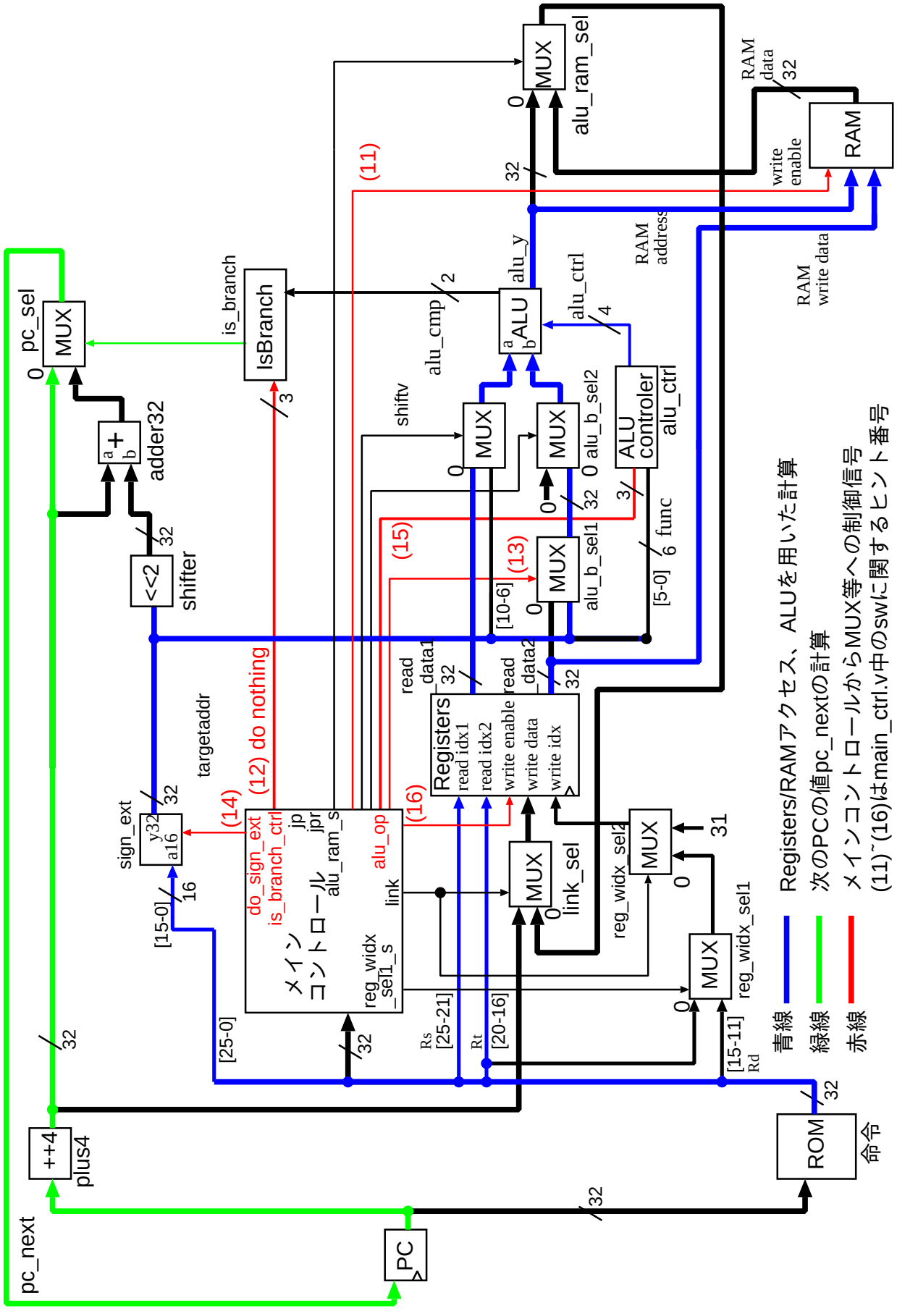


図 9: sw 命令実行時のプロセッサ内の信号の流れ

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 [7] p.246）.
   - 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「// 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 [7] pp.250–259）.
   - 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「 // 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] 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 [7] pp.250–259）.
   - 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「// 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

追加設計後のプロセッサ，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．さらに，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．

3.2.4 FGPA を用いた回路実現

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 で観察した結果と比較する．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 を両方とも 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.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

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で確認する．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」

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．

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．

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．

本動作実験で使用する MIPS マシン・コード print_B_while.bin は，実験 Web ページからダウンロードできる．

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

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 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．

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．

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

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

case ブロック内の各行は， 3.1.2 節で説明した通り，本実験で設計するプロセッサにおける，命令メモリの 10 ビットのアドレスと，そこに格納される 32 ビット命令の機械語の記述である．

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

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

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．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）．**

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，正しく動作しないはずである）．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．

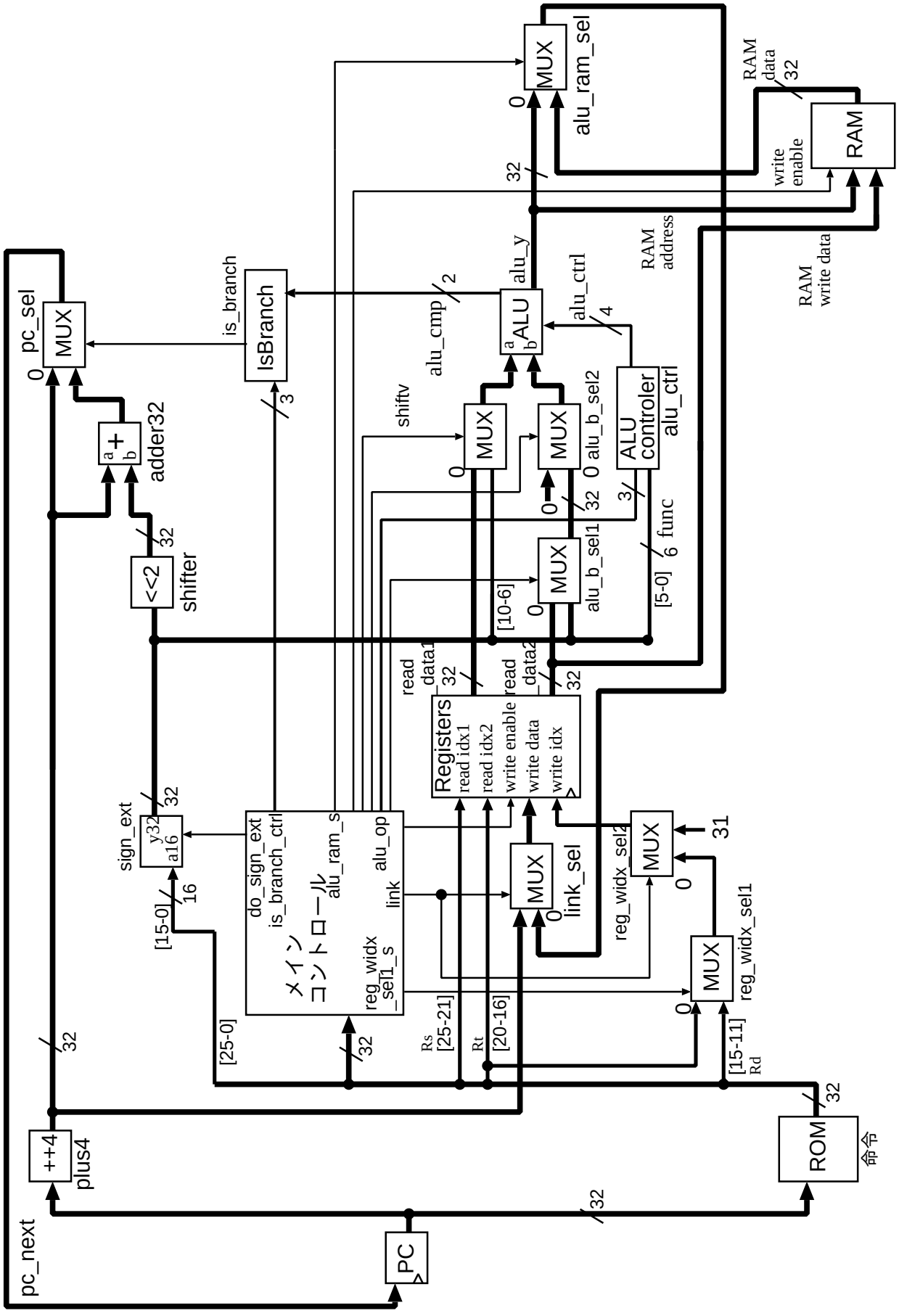


図 11: 動作実験 2-1 のプロセッサのブロック図

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．

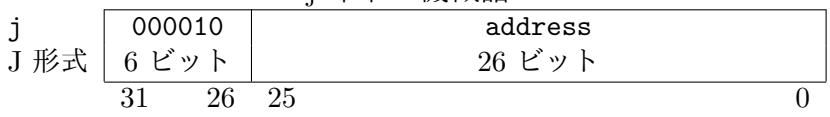
実験 2-2 動作実験 2-1 の j 命令が未実装なプロセッサについて，追加設計を行い，j 命令を正しく実行するプロセッサを完成させなさい（追加設計 2）．さらに，そのプロセッサと動作実験 2-1 の `print_B_while.bin` を FPGA 上に実現し，その動作を確認せよ（動作実験 2-2）．本実験は，4 章を参考に下記の 1, 2, 3, 4 の手順で行いなさい．

- 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



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

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．

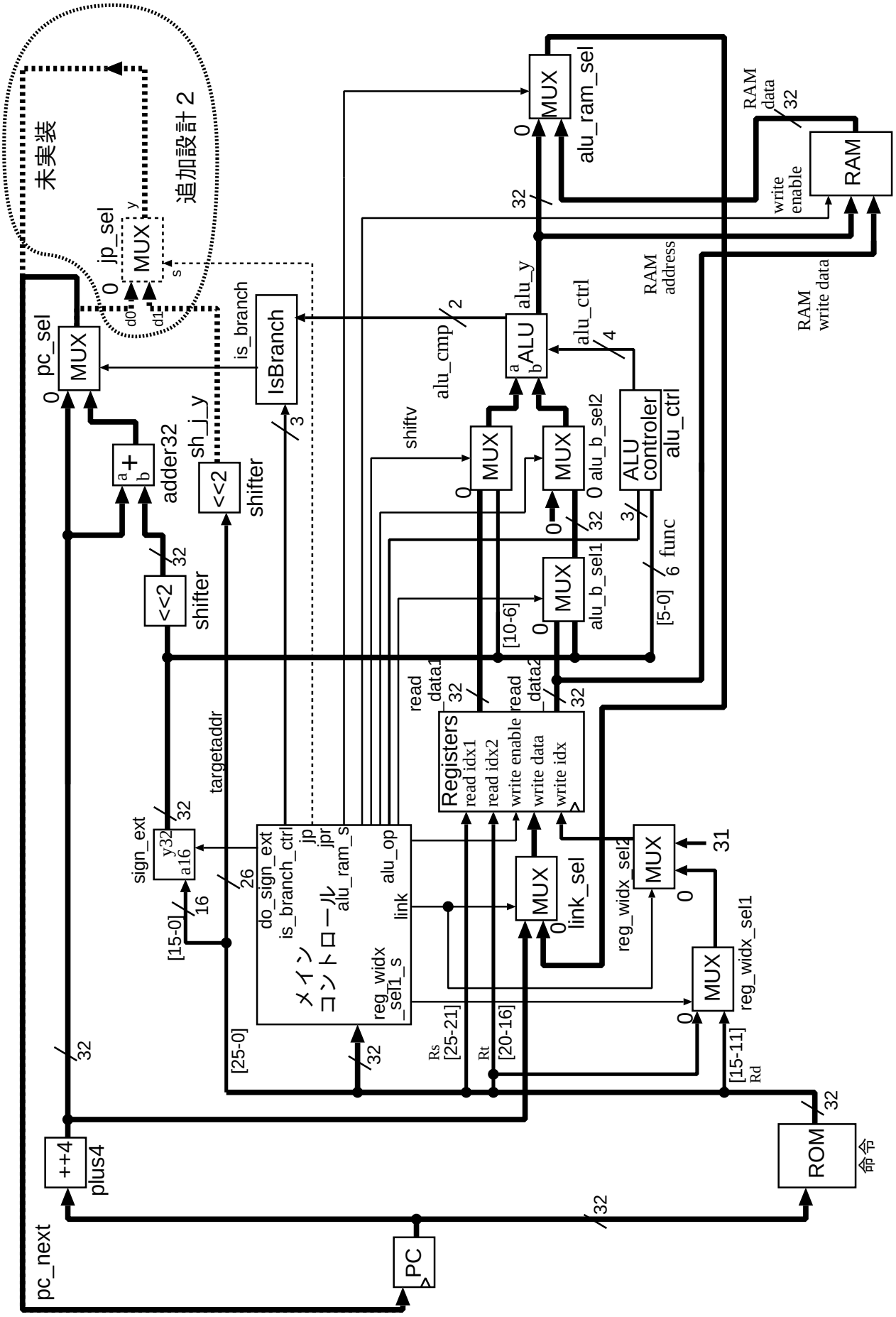


図 12: j 命令のためのジャンプ・セレクト・モジュールを含むプロセッサ

- 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．

4.2.2 j 命令のためのメイン制御回路の追加設計

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 [7] pp.247–259）．

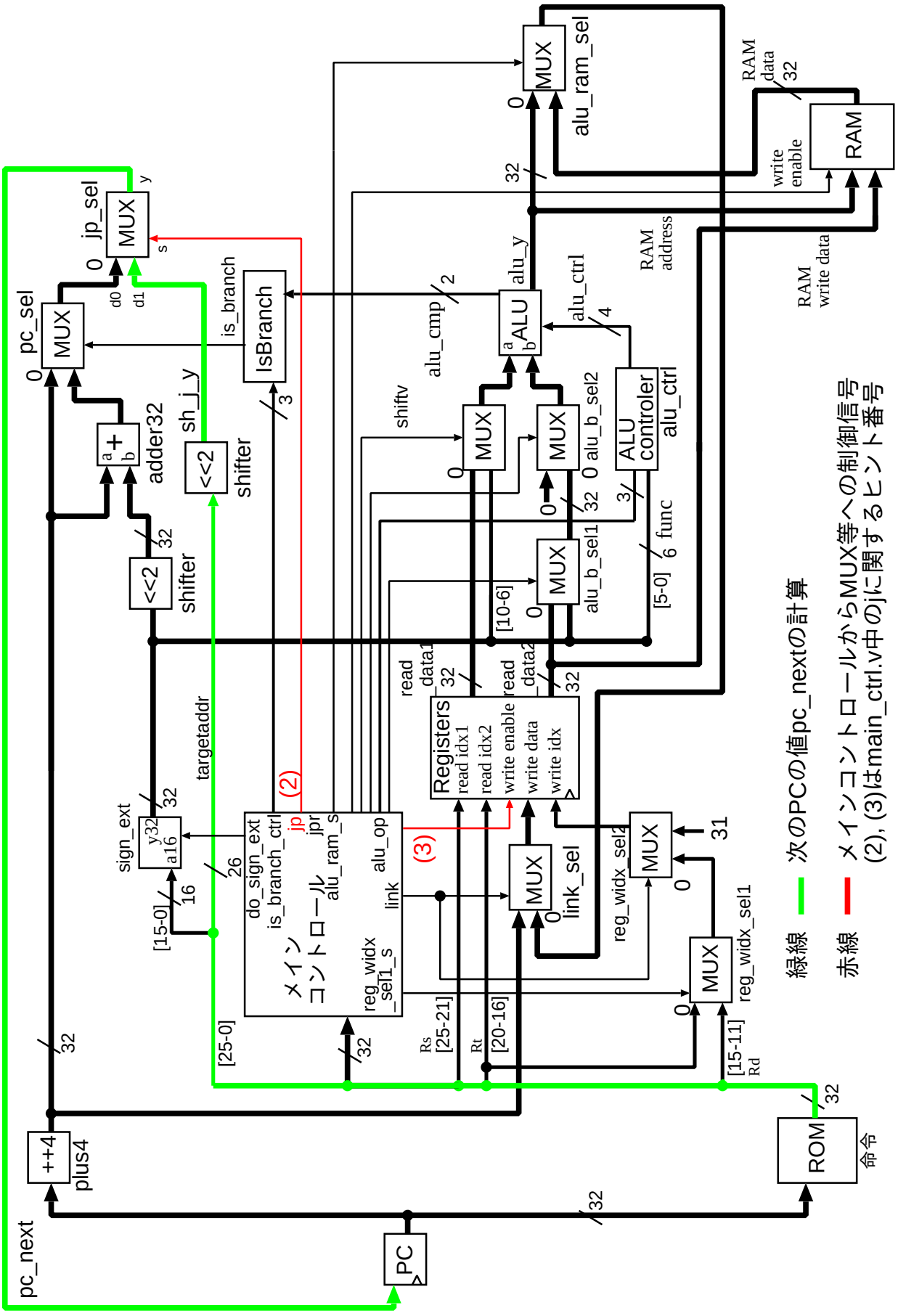


図 13: j 命令実行時のプロセッサ内の信号の流れ

- 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「// 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 [7] pp.250–259）.
   - 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「// 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.

4.2.3 論理合成

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

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）.**

4.3 C クロスコンパイラを用いたマシン・コード生成と実験 3

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

4.4 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）

本実験では，FPGA を搭載した実験基板を使用し，プロセッサを FPGA 上に実現してその動作を確認する．本動作実験では，即値符号なし・セット・オン・レス・ザン命令（set on less than immediate unsigned：`sltiu`）とブランチ・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 3 において，`sltiu`，`bne`，`lw` が正しく動くプロセッサを完成させた後，動作確認の際の比較に用いる．

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．

本動作実験で使用する C プログラム `print_all_char.c` は，実験 Web ページからダウンロードできる．

4.4.1 クロスコンパイル

本実験では，C プログラムの例としてディスプレイに 61 種類の文字を表示する図 15 の `print_all_char.c` を使用する．

クロスコンパイルには，「`cross_compile.sh`」を使用する．2.2 節に示した環境設定を行ったのち，「`cross_compile.sh print_all_char.c`」で，`print_all_char.c` から MIPS マシン･コードを作成せよ．クロスコンパイルにより，MIPS マシン･コード `print_all_char.bin` が得られる．

```
#define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
#define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)
main()
{
  unsigned int k;
  for (k = 0; k <= 60; k++) {
    EXTIO_PRINT_STROKE = (unsigned int)0x00000000;
    EXTIO_PRINT_ASCII = k;
    EXTIO_PRINT_STROKE = (unsigned int)0x00000001;
  }
}
```

図 15: print_all_char.c

4.4.2 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，MIPS マシン・コード print_all_char.bin を使用する．また変換には，変換プログラム bin2v を使用する.「bin2v print_all_char.bin」で，MIPS マシン・コード print_all_char.bin からメモリ・イメージファイルを作成せよ．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命令メモリの 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_all_char.bin は，正しいプロセッサ（sltiu 命令，bne 命令，lw 命令が実装済みのプロセッサ）で動作させると，以下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 REG[30] 番地に 0 を格納
   `sw $s0, 0x0000($s30)`
2. PC=0x040006c 番地の命令にジャンプ
   `j 0x040006c`
3. RAM の REG[30] 番地の値と 61 を比較
   `lw $s2, 0x0000($s30)`
   `sltiu $2, $2, 0x0000003d`
4. 比較結果が 1 なら $\mathrm{PC} = \mathrm{PC} + 4 - 17 * 4$，そうでなければ $\mathrm{PC} = \mathrm{PC} + 4$

4.4.3 命令メモリに格納される命令列の確認

本実験では，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 16 に rom8x1024_sim.v の一部を示す．図 16 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 命令の機械語の記述である．また，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説明が記述されている．コメント部には，実際の MIPS の命令メモリにおけるアドレスと，命令名，命令の意味が記述されている．命令の意味の記述では，シンボル REG[0], REG[1], . . . , REG[31] により，レジスタ 0 番から 31 番，$s0, . . . , $s31 を表す．また，シンボル RAM[w] により，データメモリの w 番地を表す．

図 16 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 0x01e 番地に機械語 0x1440ffef 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 0x00400078 に格納され，命令名は bne，レジスタ 2 番とレジスタ 0 番の値が等

```
<省略>
case (word_addr)
 <省略>
  10'h00b: data = 32'hafc00000; // 0040002c: SW, RAM[REG[30]+0]<=REG[0];    ここが PC=0x002c の命令
  10'h00c: data = 32'h0810001b; // 00400030: J, PC<=0x0010001b*4(=0x0040006c);
  10'h00d: data = 32'h00000000; // 00400034: SLL, REG[0]<=REG[0]<<0;
  10'h00e: data = 32'h24020300; // 00400038: ADDIU, REG[2]<=REG[0]+768(=0x00000300);
  10'h00f: data = 32'hac400000; // 0040003c: SW, RAM[REG[2]+0]<=REG[0];
  10'h010: data = 32'h24030304; // 00400040: ADDIU, REG[3]<=REG[0]+772(=0x00000304);
  10'h011: data = 32'h8fc20000; // 00400044: LW, REG[2]<=RAM[REG[30]+0];    ここが 命令メモリ 0x011 の命令
  10'h012: data = 32'h00000000; // 00400048: SLL, REG[0]<=REG[0]<<0;
  10'h013: data = 32'hac620000; // 0040004c: SW, RAM[REG[3]+0]<=REG[2];
  10'h014: data = 32'h24030300; // 00400050: ADDIU, REG[3]<=REG[0]+768(=0x00000300);
  10'h015: data = 32'h24020001; // 00400054: ADDIU, REG[2]<=REG[0]+1(=0x00000001);
  10'h016: data = 32'hac620000; // 00400058: SW, RAM[REG[3]+0]<=REG[2];
  10'h017: data = 32'h8fc20000; // 0040005c: LW, REG[2]<=RAM[REG[30]+0];
  10'h018: data = 32'h00000000; // 00400060: SLL, REG[0]<=REG[0]<<0;
  10'h019: data = 32'h24420001; // 00400064: ADDIU, REG[2]<=REG[2]+1(=0x00000001);
  10'h01a: data = 32'hafc20000; // 00400068: SW, RAM[REG[30]+0]<=REG[2];
  10'h01b: data = 32'h8fc20000; // 0040006c: LW, REG[2]<=RAM[REG[30]+0];
  10'h01c: data = 32'h00000000; // 00400070: SLL, REG[0]<=REG[0]<<0;
   10'h01d: data = 32'h2c42003d; // 00400074: SLTIU, REG[2]<=(REG[2]<61(=0x0000003d))?1:0;      ここが 命令メモ
リ 0x01d の命令
  10'h01e: data = 32'h1440ffef; // 00400078: BNE, PC<=(REG[2] != REG[0])?PC+4+65519*4:PC+4;
 <省略>
endcase
<省略>
```

図 16: rom8x1024_sim.v の一部

しくなければ，PC に $\mathrm{PC}+4+65519*4(\mathrm{PC}+(1-17)*4)$ をセットし，等しければ PC に $\mathrm{PC}+4$ をセットする命令であることを表している．

print_all_char.bin から生成された *rom8x1024_sim.v* または，図 *16* の *Verilog HDL* 記述を解析し，以下の *1,2* について答えよ． なお，sltiu は即値符号なし・セット・オン・レス・ザン命令，bne はブランチ・オン・ノット・イコール命令，lw はロード・ワード命令である．

1. プロセッサが最初に PC=0x0074 番地の命令を実行した直後のレジスタ 2 番目の値を予想せよ．

2. プロセッサが最初に PC=0x0078 番地の命令を実行した直後の PC の値を予想せよ．

4.4.4 論理合成

本実験では，sltiu 命令と bne 命令，lw 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 2-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．

命令メモリのメモリ･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．なお，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等の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．

4.4.5 FPGA を用いた回路実現

本実験では，sltiu 命令と bne 命令，lw 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 3 において，sltiu，bne，lw が正しく動

くプロセッサを完成させた後，動作確認の際の比較に用いる．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させよ．

今回プロセッサが実行するマシン・コード print_all_char.bin はディスプレイに 61 種類の文字を表示するプログラムである．*KEY3* **を連打して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** *30* **個程度の命令を実行させ，ディスプレイ下部に文字が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 つも表示されないはずである．

図 17 に動作実験 4-1 のプロセッサのブロック図を示す．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．各信号線は， 図 17 の名前の似た信号線と，それぞれ対応している．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 の命令メモリに格納される命令列の確認の，1, 2 で予想した結果と同じ正しい動作かどうかを確認せよ．予想と異なる正しくない動作のはずである．

プロセッサが，sltiu 命令と bne 命令，lw 命令を正しく実行できていないことが分かる．これらの命令を正しく実行するために，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しているメイン制御回路を，これらの命令を適切に処理できるものにする必要がある．

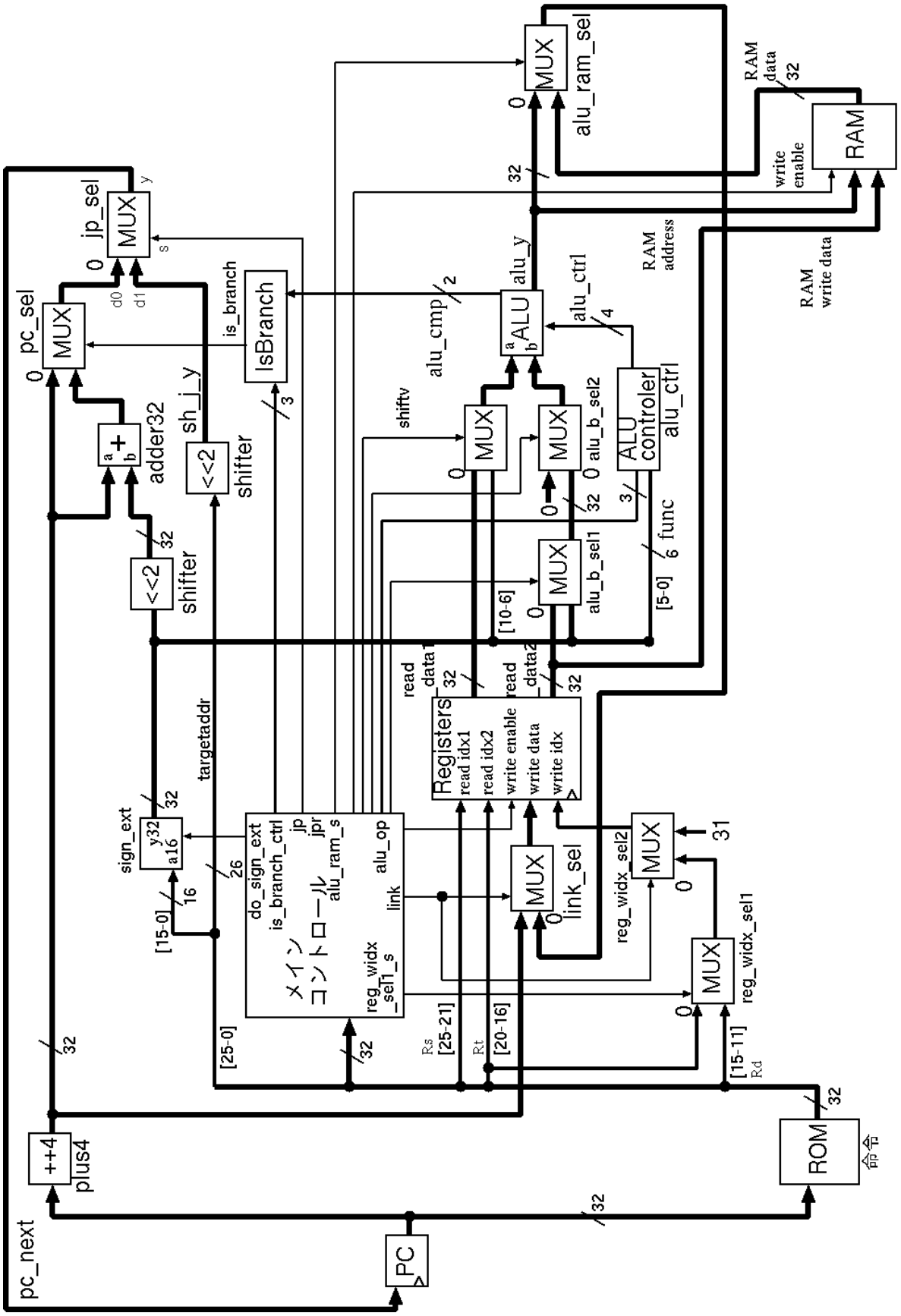


図 17: 動作実験 4-1 のプロセッサのブロック図

4.5 プロセッサの追加設計 3 （ sltiu 命令，bne 命令，lw 命令）と C プログラムの動作実験 4-2

本実験では，プロセッサの追加設計と動作実験を行う．これにより，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．本実験では，sltiu 命令と bne 命令，lw 命令が未実装なプロセッサを例とし，追加設計を行い，これらの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．

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）．さらに，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

4.5.1 sltiu 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 4-1 で動作を確認した sltiu 命令と bne 命令，lw 命令が未実装なプロセッサについて，追加設計を行う．

本実験では，動作実験 4-1 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，ディレクトリ mips_de2-115 内のサブディレクトリ MIPS 内にある．

ソースファイル main_ctrl.v 中のコメント，追加設計 *3* のヒント *(1)*〜*(8)* の周辺を，適切に変更せよ．

- sltiu 命令実行時のプロセッサ内の信号の流れを図 18 に示す．ブロック図中の青（濃い灰色）と緑（薄い灰色）の線で書かれた信号が sltiu 命令の実行に関わっている．信号の流れがブロック図のようになるように，ブロック図中の赤で書かれた制御信号を適切に設定する．

4.5.2 bne 命令のためのメイン制御回路の追加設計

本実験では，bne 命令についての追加設計を行う．

本実験では，動作実験 4-1 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，ディレクトリ mips_de2-115 内のサブディレクトリ MIPS 内にある．

ソースファイル main_ctrl.v 中のコメント，追加設計 *3* のヒント *(9)*〜*(13)* の周辺を，適切に変更せよ．

- bne 命令実行時のプロセッサ内の信号の流れを図 19 に示す．ブロック図中の青（濃い灰色）と緑（薄い灰色）の線で書かれた信号が bne 命令の実行に関わっている．信号の流れがブロック図のようになるように，ブロック図中の赤で書かれた制御信号を適切に設定する．

4.5.3 lw 命令のためのメイン制御回路の追加設計

本実験では，lw 命令についての追加設計を行う．

本実験では，動作実験 4-1 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，ディレクトリ mips_de2-115 内のサブディレクトリ MIPS 内にある．

ソースファイル main_ctrl.v 中のコメント，追加設計 *3* のヒント *(14)*〜*(21)* の周辺を，適切に変更せよ．

- lw 命令実行時のプロセッサ内の信号の流れを図 20 に示す．ブロック図中の青（濃い灰色）と緑（薄い灰色）の線で書かれた信号が lw 命令の実行に関わっている．信号の流れがブロック図のようになるように，ブロック図中の赤で書かれた制御信号を適切に設定する．

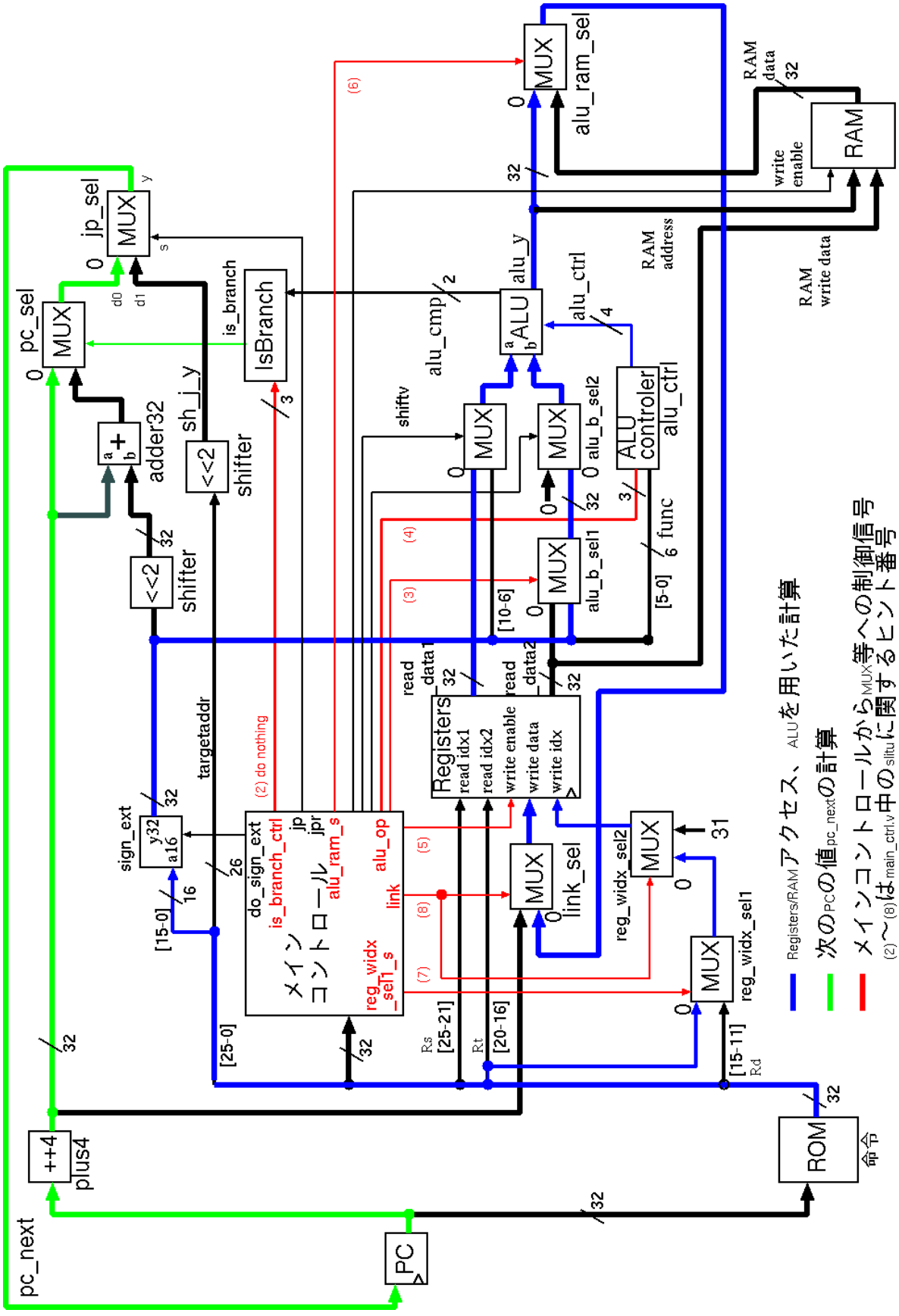


図 18: `sltiu` 命令実行時のプロセッサ内の信号の流れ

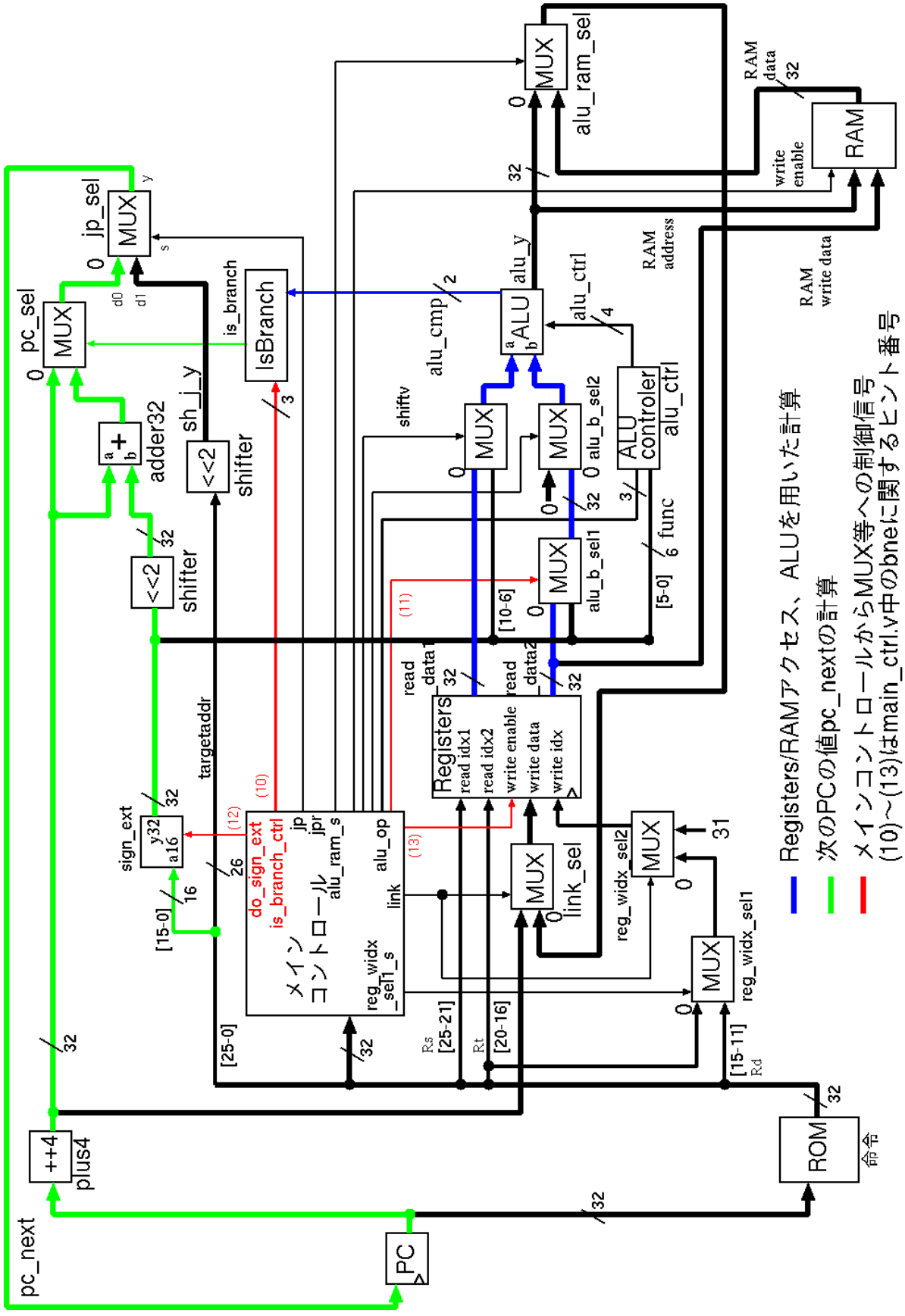


図 19: bne 命令実行時のプロセッサ内の信号の流れ

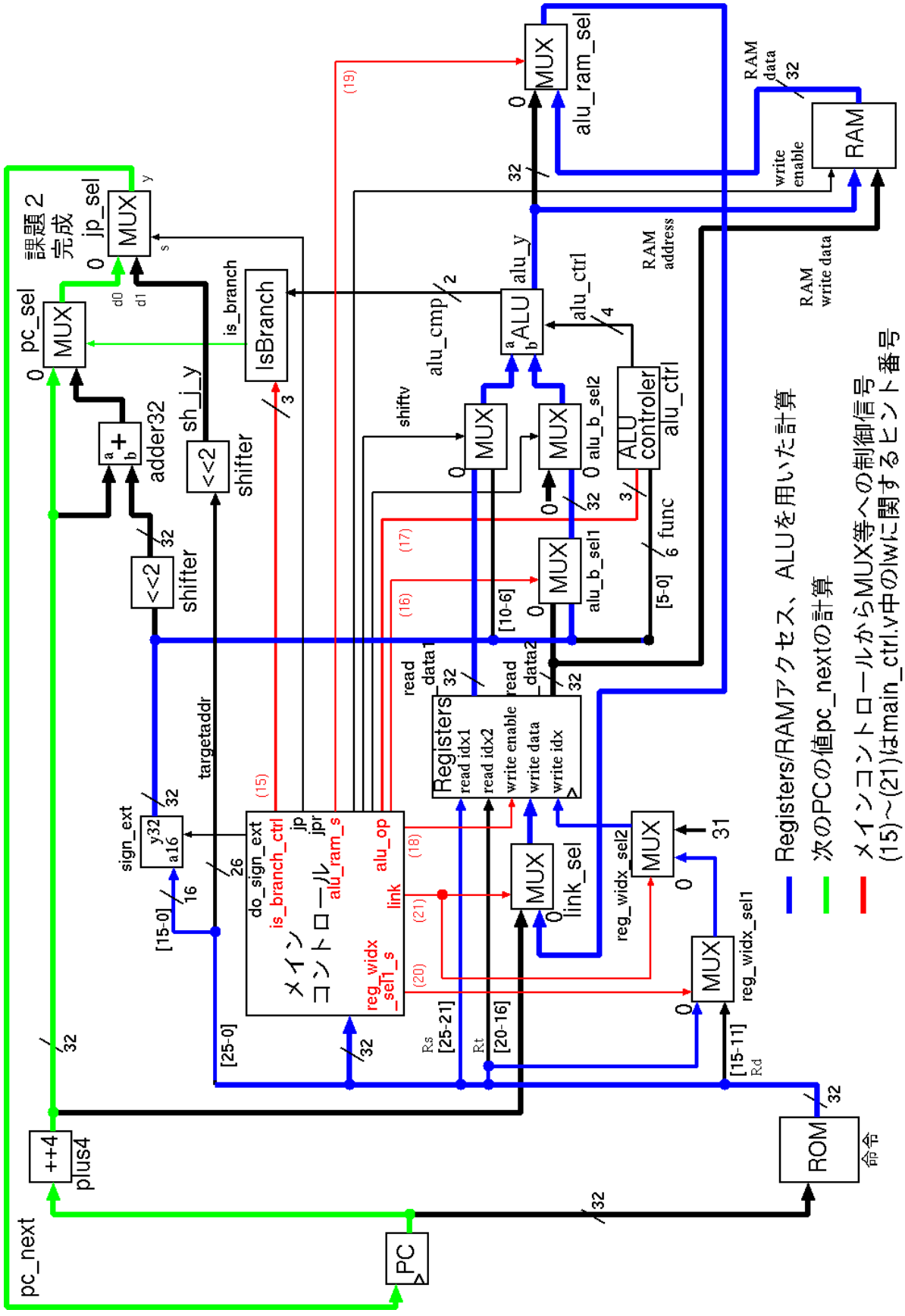


図 20: lw 命令実行時のプロセッサ内の信号の流れ

4.5.4 論理合成

本実験では，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 4-1 で使用したその他プロセッサの Verilog HDL 記述一式，print_all_char.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 内の，サブディレクトリ MIPS 内に置く．

更に，**ディレクトリ mips_de2-115 に cd し，print_all_char.bin の rom8x1024_DE2.mif がそこにあるのを確認して，「quartus_sh --flow compile DE2_115_Default」で論理合成を行う．**

なお，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等の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．

4.5.5 FPGA を用いた回路実現

本実験では，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 4-1 で観察した結果との比較を行う．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて *DE2-115* ボード上の *FPGA* にダウンロードし，動作させよ．

プロセッサが実行するマシン・コード print_all_char.bin はディスプレイ下部に 61 種類の文字を表示するプログラムである．*KEY3* を連打して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 *60* 個程度の命令を実行させ，ディスプレイ下部に異なる文字が *2* つ表示されるかどうかを確認せよ． 異なる文字が 2 つ表示されるはずである．

また，動作実験 *4-1* で確認された，動作実験 *4-1* の *3* の *1, 2* 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． 動作実験 4-1 の 3 の 1, 2 で予想した結果と同じ動作になったはずである．

完成版の sof を動かしてみたい場合は，実験 Web ページから DE2_115_Default.k04.sof をダウンロードできる

5　シングルサイクル RISC プロセッサの設計「上級編」

第 3 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・アンド・リンク命令（jal）とジャンプ・レジスタ命令（jr）について追加設計を行う．

5.1　C プログラムの動作実験 5-1（関数呼出し・ディスプレイへの文字列出力関数）

本実験では，FPGA を搭載した実験基板を使用し，プロセッサを FPGA 上に実現してその動作を確認する．本動作実験では，ジャンプ・アンド・リンク命令（jump and link : jal）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 4 において，jal が正しく動くプロセッサを完成させた後，動作確認の際の比較に用いる．

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．

本動作実験で使用する C プログラム my_print.c は，実験 Web ページからダウンロードできる．

5.1.1　クロスコンパイル

本実験では，C プログラムの例として文字列を表示する図 21 の my_print.c を使用する．

クロスコンパイルには,「cross_compile.sh」を使用する．2.2 節に示した環境設定を行ったのち,「cross_compile.sh my_print.c」で，my_print.c から MIPS マシン・コードを作成せよ．クロスコンパイルにより，MIPS マシン・コード my_print.bin が得られる．

```
#define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
#define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)

void my_print();

main()
{
  unsigned int string[64];

  string[0] = 'H';
  string[1] = 'E';
  string[2] = 'L';
  string[3] = 'L';
  string[4] = 'O';
  string[5] = '!';
  string[6] = '!';
  string[7] = '\0';

  my_print(string);

  string[0] = 'B';
  string[1] = '\0';
}

void my_print(str)
     unsigned int *str;
{
  while (*str != '\0') {
    EXTIO_PRINT_STROKE = (unsigned int)0x00000000;

    if ((*str >= 'A') && (*str <= 'Z')) {
      EXTIO_PRINT_ASCII = *str - 'A' + 1;
    } else if ((*str >= 'a') && (*str <= 'z')) {
      EXTIO_PRINT_ASCII = *str - 'a' + 1;
    } else if ((*str >= '0') && (*str <= '9')) {
      EXTIO_PRINT_ASCII = *str - '0' + 48;
    } else {
      if (*str == '@') {
EXTIO_PRINT_ASCII = (unsigned int)0;
      } else if (*str == '[') {
EXTIO_PRINT_ASCII = (unsigned int)27;
      } else if (*str == ']') {
EXTIO_PRINT_ASCII = (unsigned int)29;
      } else if ((*str >= ' ') && (*str <= '/')) {
EXTIO_PRINT_ASCII = *str - ' ' + 32;
      } else if (*str == '?') {
EXTIO_PRINT_ASCII = (unsigned int)58;
      } else if (*str == '=') {
EXTIO_PRINT_ASCII = (unsigned int)59;
      } else if (*str == ';') {
EXTIO_PRINT_ASCII = (unsigned int)60;
      } else if (*str == ':') {
EXTIO_PRINT_ASCII = (unsigned int)61;
      } else {
EXTIO_PRINT_ASCII = (unsigned int)0x00000000;
      }
    }

    EXTIO_PRINT_STROKE = (unsigned int)0x00000001;
    str++;
  }
}
```

図 21: my_print.c

5.1.2 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，MIPS マシン・コード my_print.bin を使用する．また変換には，変換プログラム bin2v を使用する.「bin2v my_print.bin」で，MIPS マシン・コード my_print.bin からメモリ・イメージファイルを作成せよ．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命令メモリの 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 my_print.bin は，正しいプロセッサ（ジャンプ&リンク命令 jal が実装済みのプロセッサ）で動作させると，以下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 $s30+16 番地に 'H' を格納
   ```
   addiu $s2, $s0, 0x0048
   sw $s2, 0x0010($s30)
   ```
2. データメモリ（RAM）の $s30+20 番地に 'E' を格納
   ```
   addiu $s2, $s0, 0x0045
   sw $s2, 0x0014($s30)
   ```
3. データメモリ（RAM）の $s30+24 番地に 'L' を格納
   ```
   addiu $s2, $s0, 0x004c
   sw $s2, 0x0018($s30)
   ```
4. データメモリ（RAM）の $s30+28 番地に 'L' を格納
   ```
   addiu $s2, $s0, 0x004c
   sw $s2, 0x001c($s30)
   ```
5. データメモリ（RAM）の $s30+32 番地に 'O' を格納
   ```
   addiu $s2, $s0, 0x004f
   sw $s2, 0x0020($s30)
   ```
6. データメモリ（RAM）の $s30+36 番地に '!' を格納
   ```
   addiu $s2, $s0, 0x0021
   sw $s2, 0x0024($s30)
   ```
7. データメモリ（RAM）の $s30+40 番地に '!' を格納
   ```
   addiu $s2, $s0, 0x0021
   sw $s2, 0x0028($s30)
   ```
8. データメモリ（RAM）の $s30+44 番地に '\0' を格納
   ```
   sw $s0, 0x002c($s30)
   ```
9. PC = 0x04000a0 番地の命令にジャンプ&リンク
   ```
   jal 0x04000a0
   ```

5.1.3 命令メモリに格納される命令列の確認

本実験では，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 22 に rom8x1024_sim.v の一部を示す．図 22 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 命令の機械語の記述である．また，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説明が記述されている．コメント部には，実際の MIPS の命令メモリにおけるアドレスと，命令名，命令の意味が記述されている．命令の意味の記述では，シ

```
<省略>
case (word_addr)
 <省略>
      10'h00b: data = 32'h03a0f021; // 0040002c: ADDU, REG[30]<=REG[29]+REG[0];     ここが PC=0x002c の命令
      10'h00c: data = 32'h24020048; // 00400030: ADDIU, REG[2]<=REG[0]+72(=0x00000048);
      10'h00d: data = 32'hafc20010; // 00400034: SW, RAM[REG[30]+16]<=REG[2];
      10'h00e: data = 32'h24020045; // 00400038: ADDIU, REG[2]<=REG[0]+69(=0x00000045);
      10'h00f: data = 32'hafc20014; // 0040003c: SW, RAM[REG[30]+20]<=REG[2];
      10'h010: data = 32'h2402004c; // 00400040: ADDIU, REG[2]<=REG[0]+76(=0x0000004c);
      10'h011: data = 32'hafc20018; // 00400044: SW, RAM[REG[30]+24]<=REG[2];
      10'h012: data = 32'h2402004c; // 00400048: ADDIU, REG[2]<=REG[0]+76(=0x0000004c);
      10'h013: data = 32'hafc2001c; // 0040004c: SW, RAM[REG[30]+28]<=REG[2];
      10'h014: data = 32'h2402004f; // 00400050: ADDIU, REG[2]<=REG[0]+79(=0x0000004f);
      10'h015: data = 32'hafc20020; // 00400054: SW, RAM[REG[30]+32]<=REG[2];
      10'h016: data = 32'h24020021; // 00400058: ADDIU, REG[2]<=REG[0]+33(=0x00000021);
      10'h017: data = 32'hafc20024; // 0040005c: SW, RAM[REG[30]+36]<=REG[2];
      10'h018: data = 32'h24020021; // 00400060: ADDIU, REG[2]<=REG[0]+33(=0x00000021);
      10'h019: data = 32'hafc20028; // 00400064: SW, RAM[REG[30]+40]<=REG[2];
      10'h01a: data = 32'hafc0002c; // 00400068: SW, RAM[REG[30]+44]<=REG[0];
      10'h01b: data = 32'h27c20010; // 0040006c: ADDIU, REG[2]<=REG[30]+16(=0x00000010);
      10'h01c: data = 32'h00402021; // 00400070: ADDU, REG[4]<=REG[2]+REG[0];
      10'h01d: data = 32'h0c100028; // 00400074: JAL, PC<=0x00100028*4(=0x004000a0); REG[31]<=PC+4     ここが 命
令メモリ 0x01d の命令
 <省略>
endcase
<省略>
```

図 22: rom8x1024_sim.v の一部

ンボル REG[0], REG[1], . . . , REG[31] により，レジスタ 0 番から 31 番，$s0, . . . , $s31 を表す．また，シンボル RAM[w] により，データメモリの w 番地を表す．

図 22 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 0x01d 番地に機械語 0x0c1000028 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 0x00400074 に格納され，命令名は jal，レジスタ 31 番に PC + 4 をセットし，PC に 0x00100028 ∗ 4 をセットする命令であることを表している．

my_print.bin から生成された rom8x1024_sim.v, または，図 *22* の *Verilog HDL* 記述を解析し，以下の *1, 2* について答えよ． なお，jal はジャンプ・アンド・リンク命令はである．

1. プロセッサが最初に PC=0x0074 番地の命令を実行した直後のレジスタ 31 番目の値を予想せよ．

2. プロセッサが最初に PC=0x0074 番地の命令を実行した直後の PC の値を予想せよ．

5.1.4 論理合成

本実験では，jal 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 4-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．

命令メモリのメモリ･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．なお，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等の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．

5.1.5 FPGA を用いた回路実現

本実験では，jal 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 4 において，jal が正しく動くプロセッサを完成させた後，動作確認の際の比較に用いる．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させよ．

今回プロセッサが実行するマシン･コード my_print.bin はディスプレイに文字列 "HELLO!!" を表示するプログラムである．*KEY3* を連打して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 *70* 個程度の命令を実行させ，ディスプレイ下部に文字列 "HELLO!!" の一部が表示されるかどうかを確認せよ． ディスプレイ下部に "HELLO!!" に含まれる文字は 1 つも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．各信号線は， 図 23 の名前の似た信号線と，それぞれ対応している．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 の命令メモリに格納される命令列の確認の，1, 2 で予想した結果と同じ正しい動作かどうかを確認せよ．予想と異なる正しくない動作のはずである．

プロセッサが，jal 命令を正しく実行できていないことが分かる．これらの命令を正しく実行するために，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しているメイン制御回路を，この命令を適切に処理できるものにする必要がある．

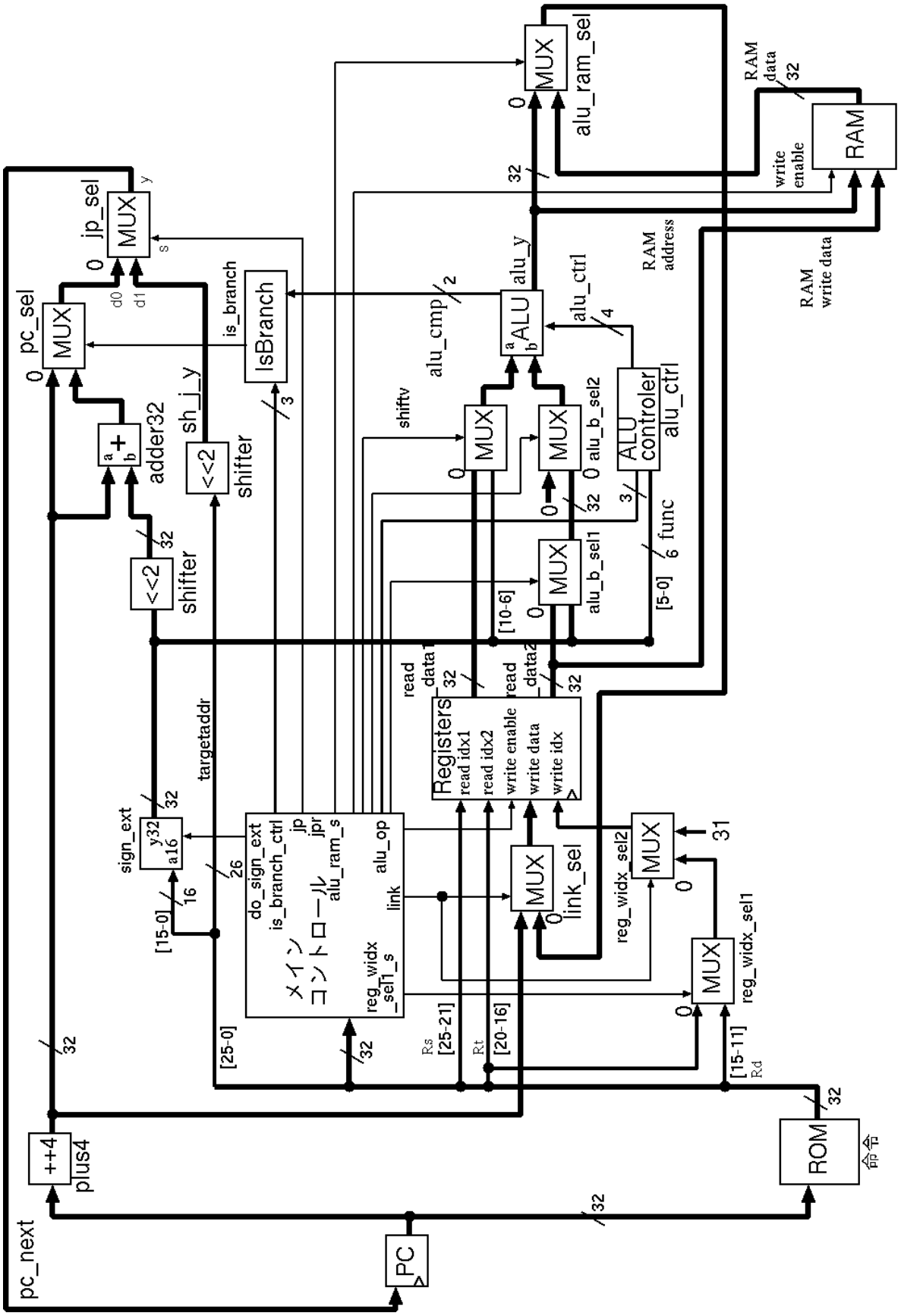


図 23: 動作実験 5-1 のプロセッサのブロック図

5.2 プロセッサの追加設計 4（jal 命令）と C プログラムの動作実験 5-2

本実験では，プロセッサの追加設計と動作実験を行う．これにより，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．本実験では，jal 命令が未実装なプロセッサを例とし，追加設計を行い，これらの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．

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）．さらに，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．

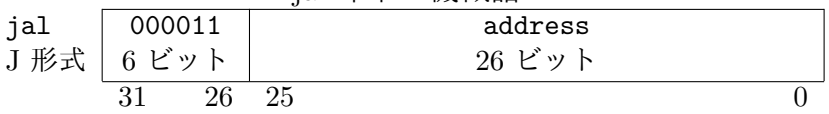
- プロセッサの追加設計 4 の手順
  1. jal 命令のためのメイン制御回路の追加設計
     プロセッサのメイン制御回路の追加設計を行い，プロセッサを完成させる．
- 動作実験 5-2 の手順
  2. 論理合成
     完成したプロセッサ，その他周辺回路の論理合成を行う．
  3. FPGA を用いた回路実現
     完成したプロセッサ等の回路を DE2-115 ボード上の FPGA にダウンロードし，実際の動作を観察する．

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 = address`
		`ra = PC + 4`
		（`ra` は 31 番目のレジスタ）

jal 命令の機械語



5.2.1 jal 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 5-1 で動作を確認した `jal` 命令が未実装なプロセッサについて，追加設計を行う．

本実験では，動作実験 5-1 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，ディレクトリ `mips_de2-115` 内のサブディレクトリ `MIPS` 内にある．

ソースファイル `main_ctrl.v` 中のコメント，追加設計 *4* のヒント *(1)*〜*(4)* の周辺を，適切に変更せよ．

- `jal` 命令実行時のプロセッサ内の信号の流れを図 24 に示す．ブロック図中の青（濃い灰色）と緑（薄い灰色）の線で書かれた信号が `jal` 命令の実行に関わっている．信号

の流れがブロック図のようになるように，ブロック図中の赤で書かれた制御信号を適切に設定する．

5.2.2 論理合成

本実験では，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 5-1 で使用したその他プロセッサの Verilog HDL 記述一式，my_print.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 内の，サブディレクトリ MIPS 内に置く．

更に，ディレクトリ mips_de2-115 に cd し，my_print.bin の rom8x1024_DE2.mif がそこにあるのを確認して,「quartus_sh --flow compile DE2_115_Default」で論理合成を行う．

なお，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等の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．

5.2.3 FPGA を用いた回路実現

本実験では，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 5-1 で観察した結果との比較を行う．本実験には，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 を使用する．**DE2_115_Default.sof を quartus_pgm を用いて *DE2-115* ボード上の *FPGA* にダウンロードし，動作させよ．**

プロセッサが実行するマシン・コード my_print.bin はディスプレイ下部に文字列を表示するプログラムである．**KEY3 を連打して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 *70* 個程度の命令を実行させ，ディスプレイ下部に文字列 "HELLO!!" の一部が表示されるかどうかを確認せよ．** 文字列 "HELLO!!" の一部が表示されるはずである．クロック供給モードを 1000Hz にセットすると数秒で文字列 "HELLO!!" が完全に表示される．

また，**動作実験 *5-1* で確認された，動作実験 *5-1* の *3* の *1, 2* 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．** 動作実験 5-1 の 3 の 1, 2 で予想した結果と同じ動作になったはずである．

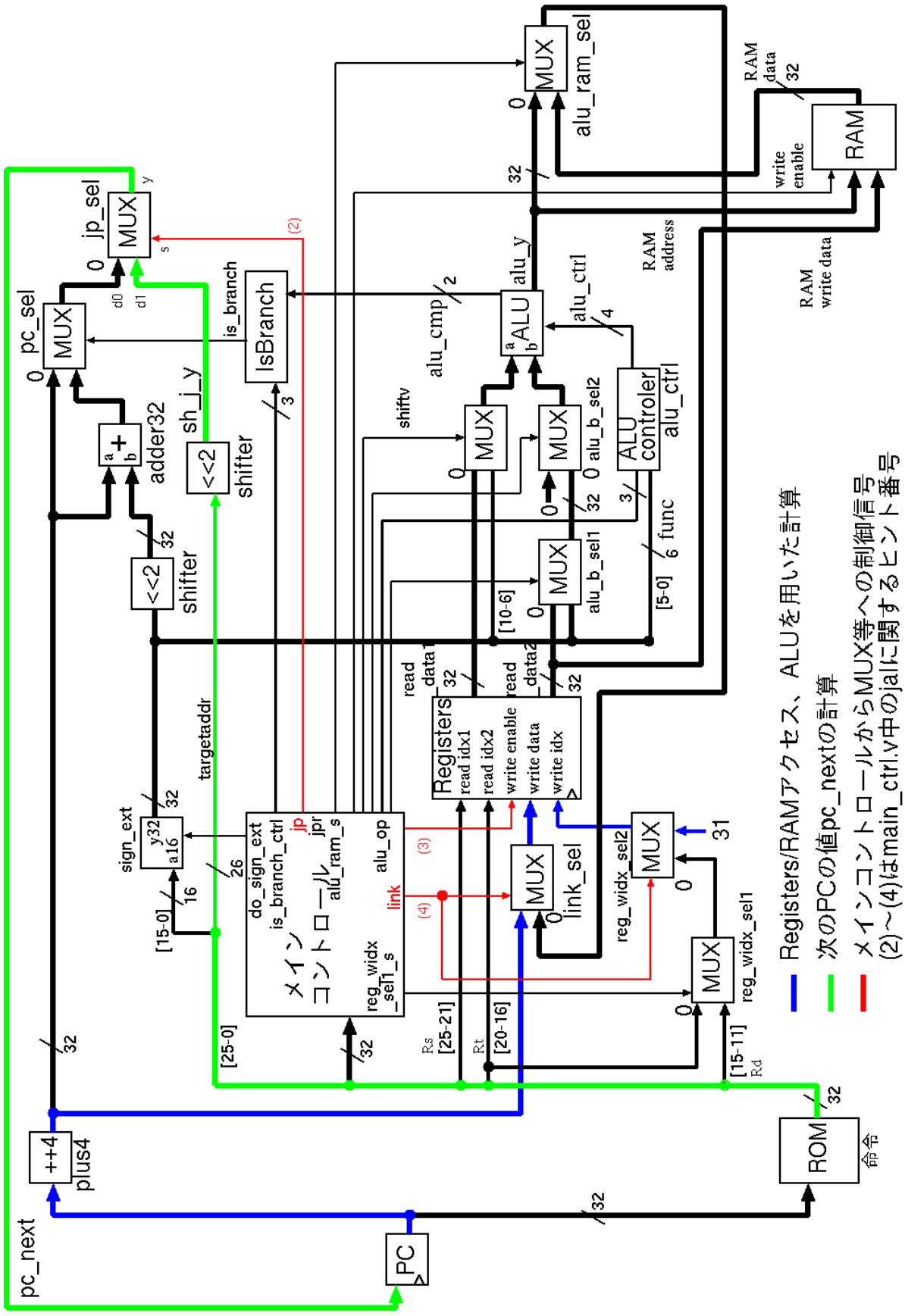


図 24: `jal` 命令実行時のプロセッサ内の信号の流れ

5.3 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）

本実験では，FPGA を搭載した実験基板を使用し，プロセッサを FPGA 上に実現してその動作を確認する．本動作実験では，ジャンプ・レジスタ命令（jump register：jr）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 5 において，jr が正しく動くプロセッサを完成させた後，動作確認の際の比較に用いる．

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．

本動作実験で使用する C プログラム my_scan.c は，実験 Web ページからダウンロードできる．

5.3.1 クロスコンパイル

本実験では，C プログラムの例としてキーボードからの文字列入力を受ける図25 の my_scan.c を使用する．

クロスコンパイルには，「cross_compile.sh」を使用する．2.2 節に示した環境設定を行ったのち，「cross_compile.sh my_scan.c」で，my_scan.c から MIPS マシン・コードを作成せよ．クロスコンパイルにより，MIPS マシン・コード my_scan.bin が得られる．

```
#define EXTIO_SCAN_ASCII (*(volatile unsigned int *)0x0310)
#define EXTIO_SCAN_REQ (*(volatile unsigned int *)0x030c)
#define EXTIO_SCAN_STROKE (*(volatile unsigned int *)0x0308)

#define SCAN_STRORING (unsigned int)0xffffffff

#define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
#define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)

void my_print();
void my_scan();

main()
{
  unsigned int string1[32];
  unsigned int string2[32];

  string1[0] = 'H';
  string1[1] = 'E';
  string1[2] = 'L';
  string1[3] = 'L';
  string1[4] = 'O';
  string1[5] = '!';
  string1[6] = '!';
  string1[7] = '\n';
  string1[8] = '\0';

  my_print(string1);

  <省略>
}

void my_scan(str)
     unsigned int *str;
{
  EXTIO_SCAN_STROKE = (unsigned int)0x00000000;
  EXTIO_SCAN_REQ = (unsigned int)0x00000001;
  EXTIO_SCAN_STROKE = (unsigned int)0x00000001;
  <省略>
}

void my_print(str)
     unsigned int *str;
{
  while (*str != '\0') {
    EXTIO_PRINT_STROKE = (unsigned int)0x00000000;
  <省略>
}
```

図 25: my_scan.c の一部

5.3.2 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，MIPS マシン・コード my_scan.bin を使用する．また変換には，変換プログラム bin2v を使用する.「bin2v my_scan.bin」で，MIPS マシン・コード my_scan.bin からメモリ・イメージファイルを作成せよ．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命令メモリの 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 my_scan.bin は，正しいプロセッサ（ジャンプレジスタ命令 jr が実装済みのプロセッサ）で動作させると，以下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. PC = 0x04004d8 番地の命令にジャンプ&リンク
   jal 0x04004d8
2. PC = REG[31] 番地の命令にジャンプレジスタ
   jr

5.3.3 命令メモリに格納される命令列の確認

本実験では，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 26 に rom8x1024_sim.v の一部を示す．図 26 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 命令の機械語の記述である．また，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説明が記述されている．コメント部には，実際の MIPS の命令メモリにおけるアドレスと，命令名，命令の意味が記述されている．命令の意味の記述では，シンボル REG[0], REG[1], . . . , REG[31] により，レジスタ 0 番から 31 番，$s0, . . . , $s31 を表す．また，シンボル RAM[w] により，データメモリの w 番地を表す．

```
<省略>
case (word_addr)
 <省略>
      10'h00b: data = 32'h03a0f021; // 0040002c: ADDU, REG[30]<=REG[29]+REG[0];     ここが PC=0x002c の命令
<省略>
      10'h01f: data = 32'h0c100136; // 0040007c: JAL, PC<=0x00100136*4(=0x004004d8); REG[31]<=PC+4
<省略>
      10'h136: data = 32'h27bdfff8; // 004004d8: ADDIU, REG[29]<=REG[29]+65528(=0x0000fff8);
      10'h137: data = 32'hafbe0000; // 004004dc: SW, RAM[REG[29]+0]<=REG[30];
      10'h138: data = 32'h03a0f021; // 004004e0: ADDU, REG[30]<=REG[29]+REG[0];
      10'h139: data = 32'hafc40008; // 004004e4: SW, RAM[REG[30]+8]<=REG[4];
      10'h13a: data = 32'h081001f8; // 004004e8: J, PC<=0x001001f8*4(=0x004007e0);
<省略>
      10'h1f8: data = 32'h8fc20008; // 004007e0: LW, REG[2]<=RAM[REG[30]+8];
      10'h1f9: data = 32'h00000000; // 004007e4: SLL, REG[0]<=REG[0]<<0;
      10'h1fa: data = 32'h8c420000; // 004007e8: LW, REG[2]<=RAM[REG[2]+0];
      10'h1fb: data = 32'h00000000; // 004007ec: SLL, REG[0]<=REG[0]<<0;
      10'h1fc: data = 32'h1440ff3f; // 004007f0: BNE, PC<=(REG[2] != REG[0])?PC+4+65343*4:PC+4;
      10'h1fd: data = 32'h00000000; // 004007f4: SLL, REG[0]<=REG[0]<<0;
      10'h1fe: data = 32'h03c0e821; // 004007f8: ADDU, REG[29]<=REG[30]+REG[0];
      10'h1ff: data = 32'h8fbe0000; // 004007fc: LW, REG[30]<=RAM[REG[29]+0];
      10'h200: data = 32'h27bd0008; // 00400800: ADDIU, REG[29]<=REG[29]+8(=0x00000008);
      10'h201: data = 32'h03e00008; // 00400804: JR, PC<=REG[31];     ここが 命令メモリ 0x201 の命令
 <省略>
endcase
<省略>
```

図 26: rom8x1024_sim.v の一部

図 26 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 0x201 番地に機械語 0x03e00008 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 0x00400800 に格納され，命令名は jr，レジスタ 31 番に 退避されている値を PC にセットする命令であることを表している．

my_scan.bin から生成された rom8x1024_sim.v，または，図 *26* の *Verilog HDL* 記述を解析し，以下の *1, 2* について答えよ．

なお，jr はジャンプ・レジスタ命令はである．

1. プロセッサが最初に PC=0x007c 番地の命令を実行した直後のレジスタ 31 番目の値を予想せよ．

2. プロセッサが最初に PC=0x0804 番地の命令を実行した直後の PC の値を予想せよ．

5.3.4 論理合成

本実験では，jr 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 5-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．

命令メモリのメモリ･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．なお，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等の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．

5.3.5 FPGA を用いた回路実現

本実験では，jr 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 5 において，jr が正しく動くプロセッサを完成させた後，動作確認の際の比較に用いる．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させよ．

今回プロセッサが実行するマシン・コード my_scan.bin はキーボードからの文字列入力を受けるプログラムである．**プロセッサにクロックパルスを次々送り，プロセッサに PC=0x0000 番地からの命令を実行させ，ディスプレイ下部に文字列が *2* つ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列は 1 つしか表示されないはずである．

図 27 に動作実験 4-1 のプロセッサのブロック図を示す．

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．各信号線は，図 27 の名前の似た信号線と，それぞれ対応している．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 の命令メモリに格納される命令列の確認の，1, 2 で予想した結果と同じ正しい動作かどうかを確認せよ．予想と異なる正しくない動作のはずである．

プロセッサが，jr 命令を正しく実行できていないことが分かる．これらの命令を正しく実行するために，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しているメイン制御回路を，この命令を適切に処理できるものにする必要がある．

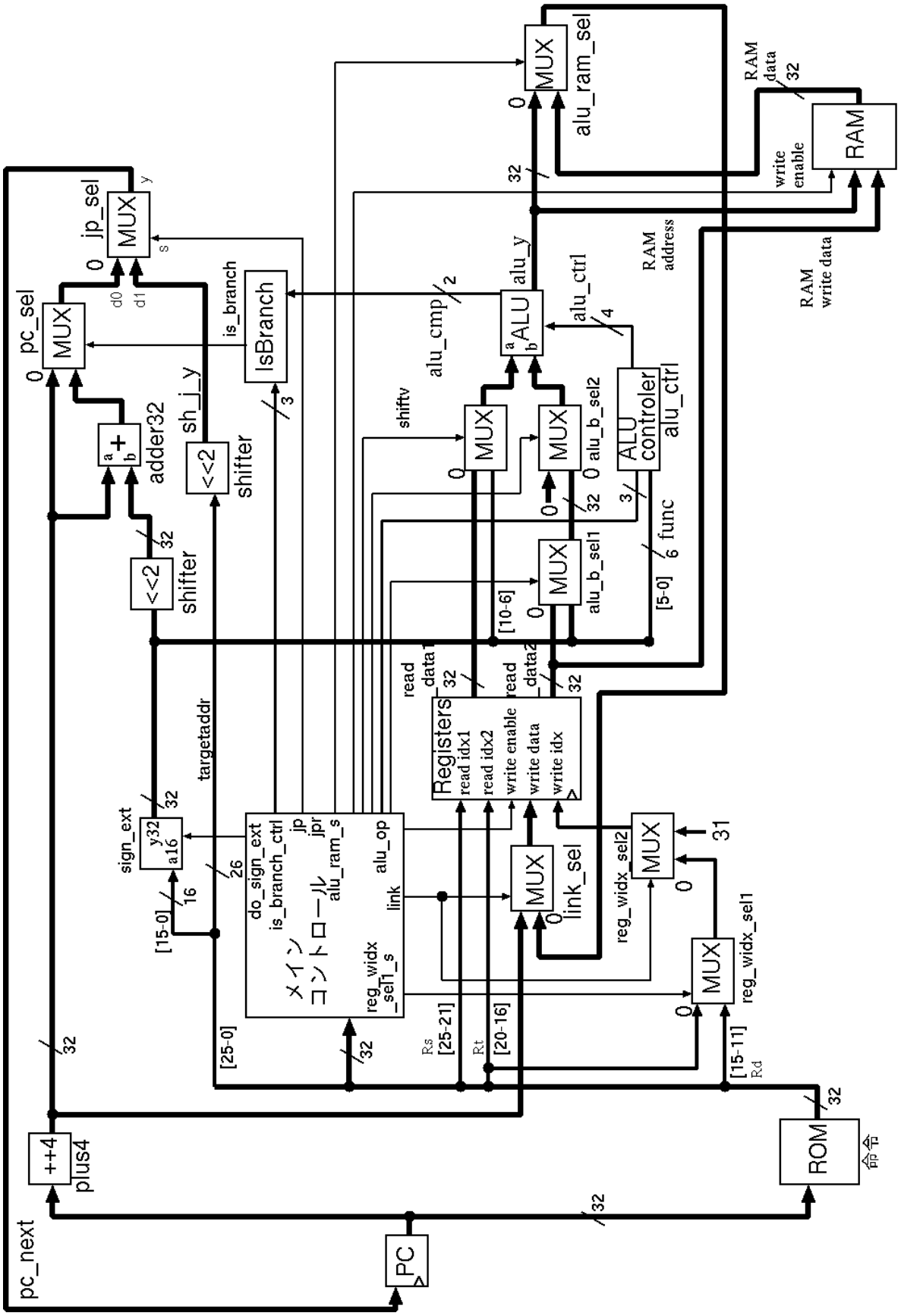


図 27: 動作実験 6-1 のプロセッサのブロック図

5.4 プロセッサの追加設計 5（jr 命令）と C プログラムの動作実験 6-2

本実験では，プロセッサの追加設計と動作実験を行う．これにより，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．

本実験では，jr 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．また，その動作を実際に動作させて観察する．

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）．さらに，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 R 形式	000000	11111	00000	00000	00000	001000
	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

5.4.1 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

本実験では，動作実験 6-1 で動作を確認した jr 命令が未実装なプロセッサについて，追加設計を行う．

本実験では，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 内のサブディレクトリ MIPS 内にある．

ソースファイル cpu.v 中のコメント，追加設計 *5* のヒント *(1)*〜*(4)* の周辺に，ジャンプ・レジスタ・セレクト・モジュールに関する記述を追加せよ．

- jr 命令は，レジスタ 31 番に 退避されている値を PC にセットする命令である．

- jr 命令のためのジャンプ・レジスタ・セレクト・モジュールを図 28 に示す．ブロック図中の破線で囲まれた，未実装，追加設計 5 と書かれた部分が jr 命令のためのジャンプ・レジスタ・セレクト・モジュールである．以下では，このジャンプ・レジスタ・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．

5.4.2 jr 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 6-1 で動作を確認した jr 命令が未実装なプロセッサについて，追加設計を行う．

本実験では，動作実験 6-1 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，ディレクトリ mips_de2-115 内のサブディレクトリ MIPS 内にある．

ソースファイル main_ctrl.v 中のコメント，追加設計 *5* のヒント *(1)*〜*(2)* の周辺を，適切に変更せよ．

- jr 命令実行時のプロセッサ内の信号の流れを図 29 に示す．ブロック図中の青（濃い灰色）と緑（薄い灰色）の線で書かれた信号が jr 命令の実行に関わっている．信号の流れがブロック図のようになるように，ブロック図中の赤で書かれた制御信号を適切に設定する．

5.4.3 論理合成

本実験では，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 6-1 で使用したその他プロセッサの Verilog HDL 記述一式，my_scan.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，プロセッサなど一式のディレクトリ mips_de2-115 内の，サブディレクトリ MIPS 内に置く．

更に，ディレクトリ mips_de2-115 に *cd* し，my_scan.bin の rom8x1024_DE2.mif がそこにあるのを確認して，「quartus_sh --flow compile DE2_115_Default」で論理合成を行う．

なお，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等の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．

5.4.4 FPGA を用いた回路実現

本実験では，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 6-1 で観察した結果との比較を行う．本実験には，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 を使用する．**DE2_115_Default.sof を quartus_pgm を用いて *DE2-115* ボード上の *FPGA* にダウンロードし，動作させよ．**

プロセッサが実行するマシン・コード my_scan.bin はキーボードからの文字列入力を受けるプログラムである．**プロセッサにクロックパルスを次々送り，プロセッサに PC=0x0000**

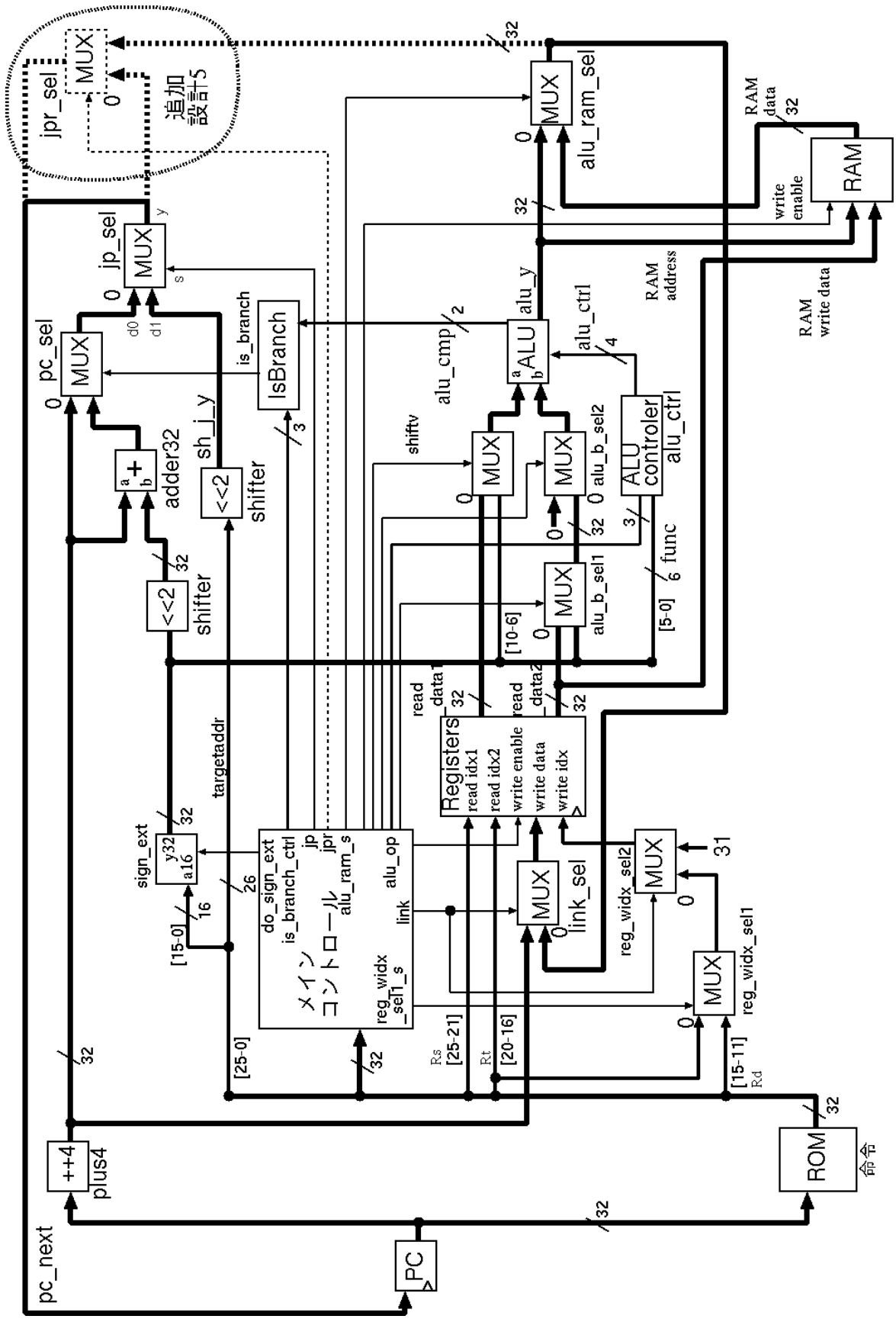


図 28: jr 命令のためのジャンプ・レジスタ・セレクト・モジュール

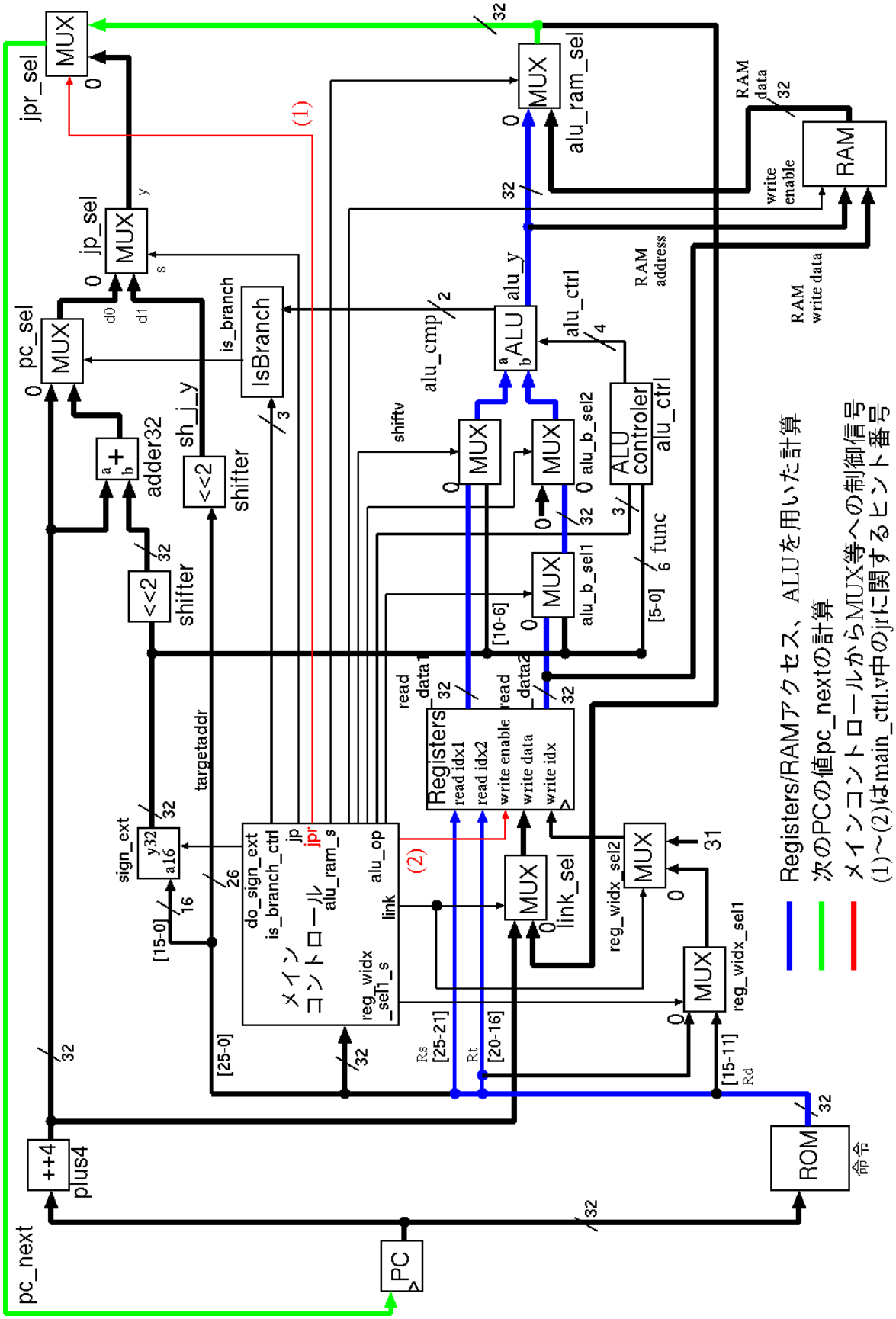


図 29: jr 命令実行時のプロセッサ内の信号の流れ

番地からの命令を実行させ，ディスプレイ下部に文字が *2* つ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が 2 つ表示されるずである．

また，動作実験 *6-1* で確認された，動作実験 *6-1* の *2* の *1, 2* 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．動作実験 6-1 の 2 の 1, 2 で予想した結果と同じ動作になったはずである．

6　シングルサイクル RISC プロセッサの設計「応用編」

第 4 週目の実験では，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．

6.1　C プログラムの動作実験 7（素数計算）

本実験では，3 以上「キーボード入力された数」以下の素数をディスプレイに次々表示する処理を，C プログラムとプロセッサにより行う．

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．

本動作実験で使用する C プログラム `sosuu.c` は，実験 Web ページからダウンロードできる．

6.1.1 クロスコンパイル

本実験では，図 30 の sosuu.c を使用する．

クロスコンパイルには,「cross_compile.sh」を使用する．2.2 節に示した環境設定を行ったのち,「cross_compile.sh sosuu.c」で，sosuu.c から MIPS マシン・コードを作成せよ．クロスコンパイルにより，MIPS マシン・コード sosuu.bin が得られる．

6.1.2 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，MIPS マシン・コード sosuu.bin を使用する．また変換には，変換プログラム bin2v を使用する.「bin2v sosuu.bin」で，MIPS マシン・コード sosuu.bin からメモリ・イメージファイルを作成せよ．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命令メモリの Verilog HDL 記述 rom8x1024_sim.v が得られる．

6.1.3 命令メモリに格納される命令列の確認

本実験では，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 31 に rom8x1024_sim.v の一部を示す．図 31 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 命令の機械語の記述である．また，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説明が記述されている．コメント部には，実際の MIPS の命令メモリにおけるアドレスと，命令名，命令の意味が記述されている．命令の意味の記述では，シンボル REG[0], REG[1], . . . , REG[31] により，レジスタ 0 番から 31 番，$s0, . . . , $s31 を表す．また，シンボル RAM[w] により，データメモリの w 番地を表す．

(a) 命令メモリの 0x082 番地の命令は，実験 *6-2* で作成したプロセッサでは未実装な命令である．これはどのような命令か，参考書 *[3]*（または，参考書 *[1]*）の表紙の次のページ，緑紙，*MIPS* リファレンスデータ，丸 *3* の表，参考書 *[7]* の *p.189,pp.226-227* の表を参考に，調査せよ（0x082 番地の命令についてのみよい）．なお，func=27(10) は func が *10* 進で *27* を表している．

(b) *(a)* の命令は，*C* ソース 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，sosuu_check() のどの記述に対応しているかを予想せよ．

6.1.4 論理合成

本実験では，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 6-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．

命令メモリのメモリ･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．なお，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

```
#define EXTIO_SCAN_ASCII (*(volatile unsigned int *)0x0310)
#define EXTIO_SCAN_REQ (*(volatile unsigned int *)0x030c)
#define EXTIO_SCAN_STROKE (*(volatile unsigned int *)0x0308)

#define SCAN_STRORING (unsigned int)0xffffffff

#define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
#define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)

#define TRUE  0x1
#define FALSE 0x0

unsigned int sosuu_check(unsigned int kouho);
unsigned int my_a2i();
void my_i2a();
void my_print();
void my_scan();

main() {
  unsigned int i;
  unsigned int k;
  unsigned int str1[16];
  unsigned int str2[16];

  <省略>
}

/* unsigned int kouho の素数判定を行う関数 */
/* 素数なら TRUE を返す */
/* 素数でないなら FALSE を返す */
unsigned int sosuu_check(unsigned int kouho) {
  unsigned int t, tester, result;

  if ((kouho % 2) == 0) {
    /* kouho は偶数である == TRUE */
    return FALSE;
  } else {
    result = TRUE;
    for (tester = 3; tester < kouho/2; tester += 2) {
      /* kouho が本当に素数かどうかをチェック */
      if ((kouho % tester) == 0) {
        /* kouho は tester の倍数である */
        result = FALSE;
      }
    }
    return result;
  }
}

/* 文字列（数字） srt[] を unsigned int に変換する関数 */
/* unsigned int result を返す */
unsigned int my_a2i(str)
     unsigned int *str;
{
  unsigned int *str_tmp;
  unsigned int k;
  unsigned int result;

  <省略>
}

/* unsigned int i を文字列（数字）に変換して print する関数 */
void my_i2a(unsigned int i) {
  unsigned int counter;
  unsigned int s[4];

  <省略>
}

/* キーボードから入力された文字列を str[] に記憶する関数 */
void my_scan(str)
     unsigned int *str;
{
  <省略>
}
```

図 30: sosuu.c の一部

```
<省略>
      10'h00b: data = 32'h03a0f021; // 0040002c: ADDU, REG[30]<=REG[29]+REG[0];     ここが PC=0x002c の命令
<省略>
      10'h081: data = 32'h14400002; // 00400204: BNE, PC<=(REG[2] != REG[0])?PC+4+2*4:PC+4;
      10'h082: data = 32'h0062001b; // 00400208: R type, unknown. func=27(10)     ここが 命令メモリ 0x082 の命令
      10'h083: data = 32'h0007000d; // 0040020c: R type, unknown. func=13(10)
      10'h084: data = 32'h00001010; // 00400210: R type, unknown. func=16(10)
      10'h085: data = 32'h14400002; // 00400214: BNE, PC<=(REG[2] != REG[0])?PC+4+2*4:PC+4;
<省略>
```

図 31: rom8x1024_sim.v の一部

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等の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．

6.1.5 FPGA を用いた回路実現

本実験では，実験 6-2 で作成したプロセッサで sosuu.bin を実行するとどのような動作をするかを観察する．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて *DE2-115* ボード上の *FPGA* にダウンロードし，動作させよ．

今回プロセッサが実行するマシン・コード sosuu.bin は，3 以上「キーボードから入力された数」以下の素数を次々とディスプレイに表示するプログラムである．

(a) クロック供給モードを *1000Hz* に設定し，ディスプレイ下部に"HELLO\n"，"NUM=" と表示されたら，キーボードから 20\n を入力する（\n は改行）．*3* 以上 *20* 以下の素数が次々と表示されてほしいが，そうはならないはずである．その結果を観察せよ．

(b) 前項で観察された正しくない動作の原因は *6.1.3* 節の *(a),(b)* にある．この問題を解決する方法を 2 つ考えよ．

6.1.6 C プログラムの変更

本実験では，おそらく 2 つの解決法のうちの 1 つである，C プログラムの変更を行う．

3 から「キーボード入力された数」までの素数をディスプレイに表示する処理が正しく行えるように，sosuu.c を変更し，実際に動作させて動作を確認せよ．

C プログラムの変更は，次の 2 点に注意して行う．

1. switch 文は使用しない（今のところ，cross_compile.sh, 又は，bin2v, プロセッサが対応していない）．if 文を使うこと．

2. C プログラムのソースにおいて，main 関数の記述は，他のどの関数の記述よりも先に書く（今のところ，プロセッサが実行する順番が，記述順に依存している）．

6.2 C プログラムの動作実験 8（ステッピングモータの制御）

本実験では，キーボードからステッピングモータを制御する処理を，C プログラムとプロセッサにより行う．

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．

本動作実験で使用する C プログラム `motor.c` は，実験 Web ページからダウンロードできる．

6.2.1 クロスコンパイル

本実験では，図 32 の `motor.c` を使用する．

クロスコンパイルには,「`cross_compile.sh`」を使用する．2.2 節に示した環境設定を行ったのち,「`cross_compile.sh motor.c`」で，`motor.c` から MIPS マシン・コードを作成せよ．クロスコンパイルにより，MIPS マシン・コード `motor.bin` が得られる．

6.2.2 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，MIPS マシン・コード `motor.bin` を使用する．また変換には，変換プログラム `bin2v` を使用する.「`bin2v motor.bin`」で，MIPS マシン・コード `motor.bin` からメモリ・イメージファイルを作成せよ．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命令メモリの Verilog HDL 記述 `rom8x1024_sim.v` が得られる．

```
#define EXTIO_SCAN_ASCII (*(volatile unsigned int *)0x0310)
#define EXTIO_SCAN_REQ (*(volatile unsigned int *)0x030c)
#define EXTIO_SCAN_STROKE (*(volatile unsigned int *)0x0308)

#define SCAN_STRORING (unsigned int)0xffffffff

#define EXTIO_PRINT_STROKE (*(volatile unsigned int *) 0x0300)
#define EXTIO_PRINT_ASCII  (*(volatile unsigned int *) 0x0304)

#define TRUE    0x1
#define FALSE   0x0

#define GPIO0 (*(volatile unsigned int *) 0x0320)
#define COUNTER 5

void my_motor();
void ext_out();

main() {
  while(1){
    my_motor();
  }
}

void my_motor() {
  ext_out(8);
  ext_out(4);
  ext_out(2);
  ext_out(1);
}

void ext_out(unsigned int num) {
  unsigned int i;

  GPIO0 = num;
}
```

図 32: motor.c

6.2.3 論理合成

本実験では，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 6-2 で完成させたプロセッサの Verilog HDL 記述一式を使用する．

命令メモリのメモリ･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．なお，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等の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．

6.2.4 FPGA を用いた回路実現

本実験では，実験 6-2 で作成したプロセッサで motor.bin を実行するとどのような動作をするかを観察する．本実験には，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 を使用する．DE2_115_Default.sof を quartus_pgm を用いて DE2-115 ボード上の FPGA にダウンロードし，動作させよ．

今回プロセッサが実行するマシン・コード motor.bin は，DE2-115 ボードの汎用拡張 IO（GPIO0）に，4-bit のモーター制御信号を出力するプログラムである．

本実験では DE2-115 ボード以外に，情報工学実験第 1（ハードウェア）で使用したステッピングモータ駆動回路，ステッピングモータ，直流安定化電源を使用する．

(a) *DE2-115* ボードの *GPIO0* に接続されている *40* ピンケーブルの，ピン番号 *0,1,2,3* から（図 *33*）ステッピングモータ駆動回路のデータ入力ピン *0,1,2,3* 番にそれぞれ配線する．

(b) *DE2-115* ボードの *GPIO0* に接続されている *40* ピンケーブルのピン番号 *11*（*GND*）から（図 *33*）ステッピングモータ駆動回路の *GND* ピンに配線する．

(c) ステッピングモータ駆動回路とステッピングモータ間を基板上のコメントに従い配線する．

(d) 直流安定化電源とステッピングモータ間を基板上のコメントに従い配線する．

(e) *DE2-115* ボードのクロック供給モードを *1000Hz* に設定する．

(f) ステッピングモータが *DE2-115* ボードからの *1* 相励磁方式の制御信号により回転する．

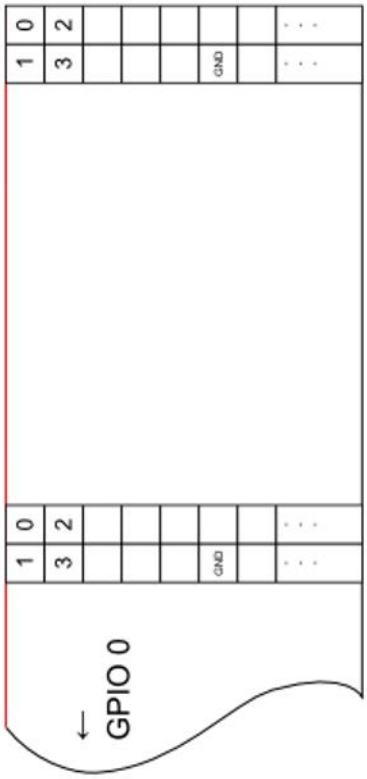


図 33: ステッピングモータ接続ピン番号

6.2.5 ステッピングモータ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せよ．

C プログラムの変更は，次の 2 点に注意して行う．

1. switch 文は使用しない（今のところ，cross_compile.sh, 又は，bin2v, プロセッサが対応していない）．if 文を使うこと．

2. C プログラムのソースにおいて，main 関数の記述は，他のどの関数の記述よりも先に書く（今のところ，プロセッサが実行する順番が，記述順に依存している）．

6.3 C プログラムの動作実験 9（整数乗算命令 mult の追加）

実験 9 整数乗算命令 mult、ならびにムーブ・フロム・Lo 命令 mflo のための bin2v の拡張とプロセッサ（cpu.v,main_ctrl.v,alu_ctrler.v）の拡張を、下記の手順で行いなさい。

- 実験 9 の手順
  1. 整数乗算命令 mult ならびにムーブ・フロム・Lo 命令 mflo のための bin2v の拡張
  2. 整数乗算命令 mult のためのプロセッサの拡張
  3. 動作実験

本動作実験で使用する C プログラム `mult.c` は，実験 Web ページからダウンロードできる．

6.3.1 整数乗算命令 mult ならびにムーブ・フロム・Lo 命令 mflo のための bin2v の拡張

本実験では、整数乗算命令 mult のための bin2v の拡張を行う。また、乗算結果が格納される 64-bit レジスタ Hi, Lo の下位 32-bit Lo をレジスタに転送する、ムーブ・フロム・Lo 命令 mflo についての拡張も同時に行う。bin2v の C 記述 bin2v.tar.gz は，実験用 Web サイトからダウンロード可能である。

bin2v.tar.gz を「`tar xvfz ./bin2v.tar.gz`」で展開し、bin2v のソース bin2v.c と Makefile を得る。

整数乗算命令 mult とムーブ・フロム・Lo 命令 mflo に関する拡張を行う。bin2v.c に図 34 に示すようななコメントを追加する（実験 9 のヒント（１）から（３））。

```
/*
実験 9 ヒント（1）: 整数乗算命令 mult, ムーブ・フロム・Lo 命令 mflo についての
コメント追加
‘define      R  6’b000000    R 形式 (add, addu, sub, subu, and, or, slt,
                                    jalr, jr, mult, mflo)
*/

/*
実験 9 ヒント（2）: 整数乗算命令 mult についてのコメント追加
MULT(op = 000000, func = 011000)
MULT      {Hi, Lo} <= REG[rs] * REG[rt];                MULT rs,rt
*/

/*
実験 9 ヒント（3）: ムーブ・フロム・Lo 命令 mflo についてのコメント追加
MFLO(op = 000000, func = 010010)
MFLO      REG[rd]  <= Lo;                               MFLO rd
*/
```

図 34: bin2v.c に追加するコメント

追加したコメントを参考に、MFLO 命令の func コードに関する define を追加する（実験 9 のヒント（４））。

追加したコメントを参考に、rom8x1024_sim.v 生成用の MULT 命令に関する記述を変更する（実験 9 のヒント（５））。追加したコメントを参考に、rom8x1024_sim.v 生成用の MFLO 命令に関する記述を追加する（実験 9 のヒント（６））。

追加したコメントを参考に、rom8x1024_DE2.mif 生成用の MULT 命令に関する記述を変更する（実験 9 のヒント（７））。追加したコメントを参考に、rom8x1024_DE2.mif 生成用の MFLO 命令に関する記述を追加する（実験 9 のヒント（８））。

bin2v.c を make し、mult, mflo 命令に対応した bin2v を得る。

6.3.2 整数乗算命令 mult のためのプロセッサの拡張

本実験では、実験 6-2 で完成させたプロセッサの Verilog HDL 記述 cpu.v, alu.v, main_ctrl.v, alu_ctrler.v について追加設計を行う。

整数乗算命令 mult ならびにムーブ・フロム・Lo 命令 mflo に関する拡張を行う。cpu.v に図 35 のようなコメントの変更を行う。

追加したコメントを参考に、図 36 のような alu.v の変更を行う（実験 9 のヒント（２）から（７））。

```
// 実験 9 のヒント（1）: ALU モジュールの実体化に関する記述の変更
//        clock, reset 信号の追加、乗算結果を保持するレジスタ hi と lo 用
// alu
//                    +----+
//      alu_a[31:0]->|    |
//      alu_b[31:0]->|    |->alu_y[31:0]
//    alu_ctrl[3:0]->|    |->alu_comp[1:0]
//                    +----+
alu  alua(alu_a, alu_b, alu_ctrl, alu_y, alu_comp);
を
// 実験 9 のヒント（1）: ALU モジュールの実体化に関する記述の変更
//        clock, reset 信号の追加、乗算結果を保持するレジスタ hi と lo 用
// alu
//                    +----+
//             clock->|    |
//             reset->|    |
//      alu_a[31:0]->|    |
//      alu_b[31:0]->|    |->alu_y[31:0]
//    alu_ctrl[3:0]->|    |->alu_comp[1:0]
//                    +----+
alu  alua(clock, reset, alu_a, alu_b, alu_ctrl, alu_y, alu_comp);
に変更
```

図 35: cpu.v の変更

main_ctrl.v に図 37 のようなコメントの追加を行う。

alu_ctrler.v に、次のような追加および変更を行う（実験 9 のヒント（１０）から（１２））。mult, mflo 命令用の ALU 制御コードの define を追加する（実験 9 のヒント（１０））。mult, mflo 命令用の ALU 制御コードについてのコメントを追加する（実験 9 のヒント（１１））。実行する命令が mult, mflo 命令のとき、mult, mflo 命令用の ALU 制御コードを生成する処理を行う記述を追加する（実験 9 のヒント（１２））。

6.3.3 動作実験

キーボードから入力された数の 2 乗を表示するプログラム mult.c を使用して、拡張された bin2v とプロセッサが正しく動くかどうかを確認せよ。

```
// 実験 9 のヒント（2）: コメントの追加 (1)
// mult(multiply)
// mflo(move from Lo)

// 実験 9 のヒント（3）: コメントの追加 (2)
// 1011,           mult
// 1100,           mflo

// 実験 9 のヒント（4）: mult, mflo 用の ALU 制御コードの define
‘define     ALU_MULT 4’b1011
‘define     ALU_MFLO 4’b1100

// 実験 9 のヒント（5）: ALU モジュールの入力ポートの拡張
//                       clock, reset 信号の追加、乗算結果を保持するレジス
タ hi と lo 用
module alu (alu_a, alu_b, alu_ctrl, alu_y, alu_comp);  // 入出力ポート
を
module alu (clock, reset, alu_a, alu_b, alu_ctrl, alu_y, alu_comp);  // 入
出力ポートに変更

// 実験 9 のヒント（6）: mult 命令実行時に alu_a * alu_b の結果を {hi, lo}
に格納する記述の追加
input       clock, reset;   // 入力 クロック, リセット
reg    [31:0] hi;              //上位
reg    [31:0] lo;              //下位
always @(posedge clock or negedge reset) begin
  if (reset == 1’b0) begin
    hi <= 32’h00000000;
    lo <= 32’h00000000;
  end else begin
    {hi, lo} <= (alu_ctrl == ‘ALU_MULT) ? alu_a * alu_b : {hi, lo};
  end
end
の追加

// 実験 9 のヒント（7）: mflo 命令実行時に {hi, lo} の lo を result に出力す
る記述の追加
‘ALU_MFLO: begin
  result <= lo;
end
```

図 36: alu.v　の変更

7　実験レポートについて

各実験について，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，表を交えてまとめよ．なお，実験 9 は発展課題なので，実施するかどうかは任意とする（実施してレポートに記載すれば，点数を加点する）．

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.

```
// 実験 9 のヒント（9）: mult, mflo 命令に関するコメントの追加
/* R 形式
MULT(op = 000000, func = 011000)
MULT       {Hi, Lo} <= REG[rs] * REG[rt];              MULT rs,rt

MFLO(op = 000000, func = 010010)
MFLO       REG[rd] <= Lo;                              MFLO rd
*/
```

図 37: main_ctrl.v へのコメントの追加

[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 5 版. 日経 BP 社, 2014.