RUECHIP2 CPU
制御／観測 リファレンスマニュアル

著者
中谷嵩之 (立命館大学大学院)
矢野正治 (京都大学大学院)
大迫 裕樹 (関西学院大学大学院)
中野 和香子 (関西学院大学大学院)
神原 弘之 (京都高度技術研究所)

2019 年 11 月 2 日
Ver.0.55

目次

第 1 章　RUECHIP2 CPU の概要 5

第 2 章　KR-CHIP 教育用ボードの仕様 7
2.1 KR-CHIP 教育用ボードのディスプレイ 8
2.2 KR-CHIP 教育用ボードのスイッチ 9
2.3 KR-CHIP 教育用ボードのコネクタ 14

第 3 章　RUECHIP2 CPU の仕様 17
3.1 RUECHIP2 CPU のメモリマップ 18
3.2 RUECHIP2 CPU 制御コマンド 19

第 4 章　プログラムの入力と実行 23
4.1 例題 23
4.2 入力 23
4.3 実行 25

第 5 章　パイプラインの観察 31
5.1 パイプラインレジスタの観察 31
5.2 フォワーディングの観察 33
5.3 分岐の観察 34

第 6 章　RUECHIP2 命令セット 37
6.1 命令形式 38
6.2 オペコードの割り当て 39
6.3 ロード命令 42
6.4 ストア命令 45
6.5 ALU イミディエト命令 48
6.6 3 オペランドレジスタタイプ命令 52
6.7 シフト命令 57
6.8 乗算／除算命令 60
6.9 ジャンプ命令 65
6.10 分岐命令 67
6.11 特殊命令 71
6.12 コプロセッサ命令 72

第 7 章 サンプルプログラムの実行 75
7.1 マーチングテストプログラム 75
7.2 KR-CHIP 用ソフトウェア環境 krchip の起動 75
7.3 サンプルプログラムのコンパイルとボードへの送信 78
7.4 サンプルプログラムの実行 81

参考文献 85

第 1 章

RUECHIP2 CPU の概要

RUECHIP2 CPU は、「パターソン＆ヘネシー コンピュータの構成と設計」で解説されている MIPS32 命令セットに基づく５段パイプラインプロセッサである. RUECHIP2 CPU には、「パイプライン」モードと,「シーケンシャル」モードの２つのプログラム実行のモードがある.

「パイライン」モードでは、命令の解釈実行が「パイプライン処理」で行われる. パイプラインレジスタの値の変化を観測することで、５段パイプラインプロセッサの動作原理ならびにパイプライン処理により生ずるデータ依存と制御依存について、深く学ぶことができる.

「シーケンシャル」モードでは, 命令の解釈実行が逐次的に１命令毎に行われる. 「シーケンシャル」モードで, 入力した機械語プログラムを実行することにより, MIPS32 命令セットの各命令の動作を容易に理解することができる.

第 2 章

KR-CHIP 教育用ボードの仕様

KR-CHIP 教育用ボードは, 以下の 3 種類の CPU の実行制御と内部動作の観測を行うために開発された.

- KUECHIP-3F (16bit アキュムレータ方式)
- KUE-CHIP2 (8bit アキュムレータ方式)
- RUECHIP2 (32bit 5 段パイプライン方式)

本リファレンスマニュアルでは, RUECHIP2 CPU (32bit 5 段パイプライン方式) の実行制御／観測の方法を解説する.

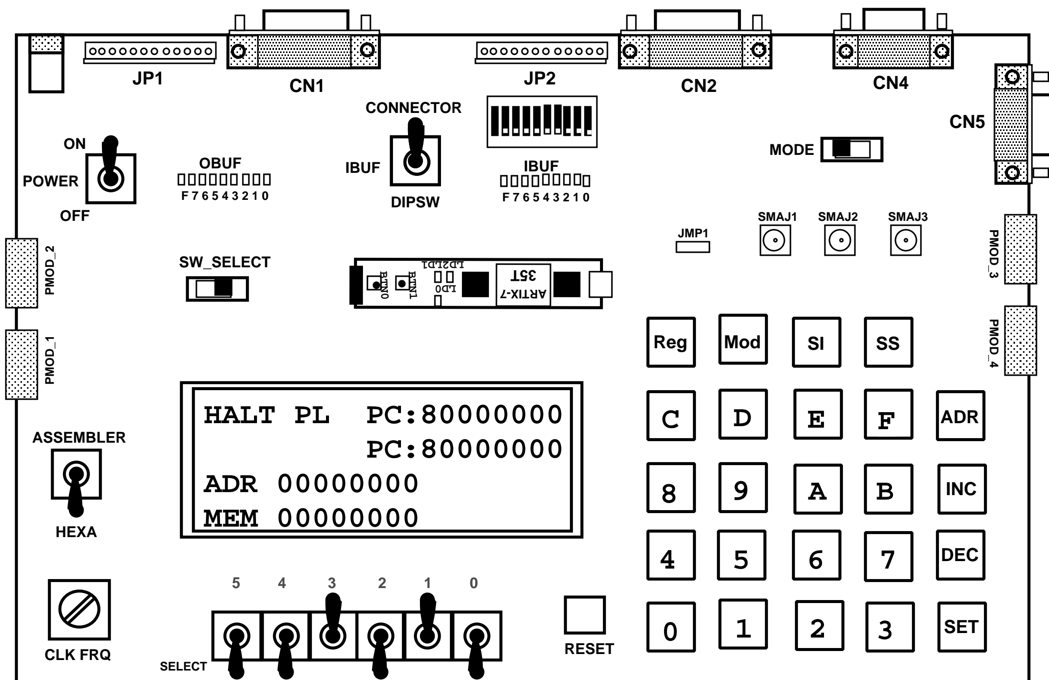


図 2.1　KR-CHIP 教育用ボード

図 2.1 に KR-CHIP 教育用ボードの外観を示す.

2.1　KR-CHIP 教育用ボードのディスプレイ

本章では, 蛍光表示管: VFD の機能について説明する.

ボードの起動時は, 図 2.2 のような画面が 1 秒間表示される.

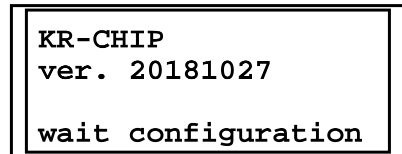


図 2.2　ボード起動時の画面例

数秒後に, 図 2.3 のような画面が表示される. 一行目は, 左から実行状態 (実行中: RUN, 停止中: HALT), 現在のモード (パイプライン: PL, シーケンシャル: SC), 32bit レジスタ：PC（Program Counter）の値を 8 桁の 16 進数で表示する, 二行目は, 6bit のトグルスイッチ：SEL_SW で指定した RUECHIP2 CPU 内部のレジスタの値を 8 桁の 16 進数、もしくは（逆アセンブルした時の）アセンブリ命令で表示する. (観測可能なレジスタについては 2.2.13 節を参照) 三行目は, RUECHIP2 CPU のメモリアドレスを、8 桁の 16 進数で表示する. 四行目は, 上段に表示されたメモリアドレスの中身を、8 桁の 16 進数もしくは（逆アセンブルした時の）アセンブリ記述で表示する. 起動時のメモリアドレスの中身は, 毎回値が異なるため図 2.3 の値とは異なる.

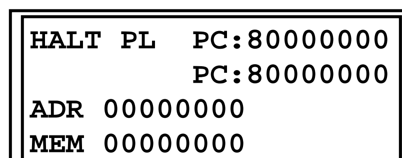


図 2.3　起動完了後に表示される画面

図 2.4 の画面が出力されているときは, キーボードからの入力を受け付けない状態である. RUECHIP2 CPU を搭載する FPGA ボード: Cmod A7 上の LED LD0 が青色に点灯している場合 (LED の詳細については?? 節を参照) に下記の画面が出力された場合には電源を付け直す必要がある. LD0 が緑色または赤色に点灯している場合には PC との接続モードとなる. 詳細は 7 章に記載する.

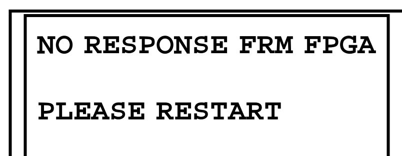


図 2.4　PC との接続モード

2.2　KR-CHIP 教育用ボードのスイッチ

本章では, 以下に示すスイッチの機能について説明する.

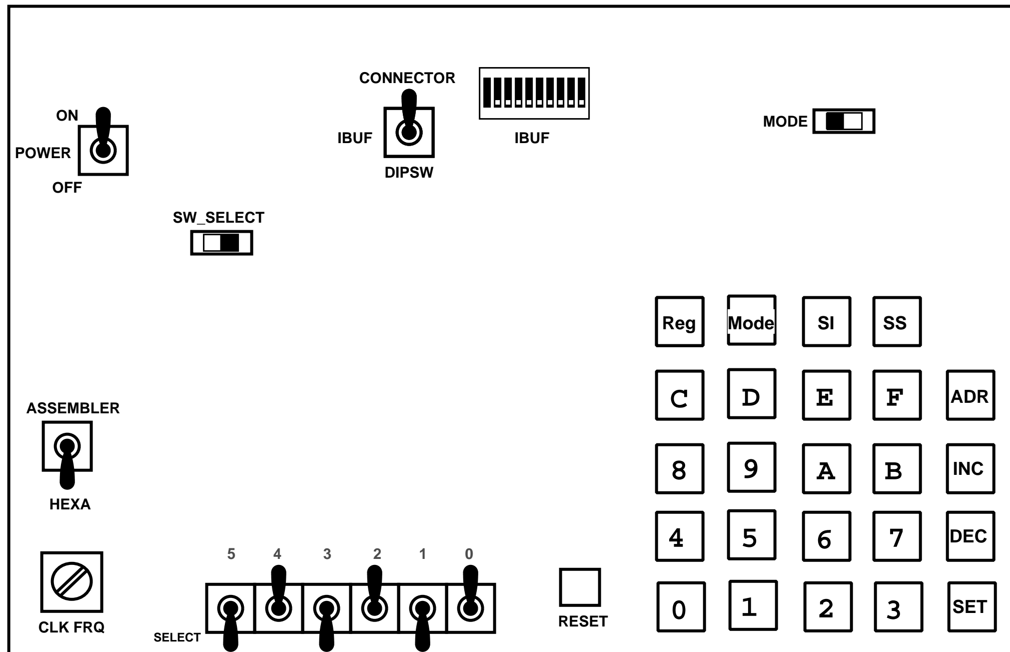
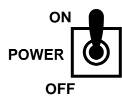


図 2.5　KR-CHIP 教育用ボードのスイッチ

2.2.1　電源スイッチ：POWER



KR-CHIP 教育用ボードの電源のオン／オフを行う. 奥側に倒すと電源がオンになり, 蛍光表示管: VFD が点灯する. VFD は, 電源スイッチをオフにしても電源コードを抜くまでは画面はそのまま表示され続ける.

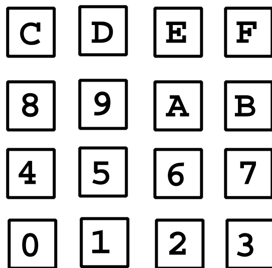
2.2.2　リセットボタン: RESET

RUECHIP2 CPU 内部のすべてのレジスタ, カウンタ, フラグ, その他のフリップフロップの値を 0 にリセットし, またフラグを 0 にリセットする. メモリの内容はそのまま保持される.

2.2.3　表示切替スイッチ：DISP

KR-CHIP 教育用ボードの蛍光表示管：四行目の表示を切替える．DISP がオフ（手前側、LED は消灯）の時、32bit の 16 進数で表示される．DISP がオン（奥側、LED は点灯）の時、逆アセンブルした時のアセンブリ記述で表示される．

2.2.4　16 進数入力キーボード



メモリ領域の編集, あるいは RUECHIP2 CPU のレジスタ (PC と汎用レジスタ：R01～R31) の値を書換えを行う際, 32bit の値を入力します．32 bit の値の入力には 8 回のプッシュが必要で, プッシュする度に下位から上位に入力された値が左シフトする．入力された内容は蛍光表示管: 四行目に表示される．

2.2.5　メモリ編集ボタン：SET

メモリ領域の値を 16 進数キーボードで入力した値に書換える．

2.2.6　メモリアドレス・デクリメントボタン：DEC

メモリ領域を編集する際, アドレスの値を 4 つデクリメントする．

2.2.7　メモリアドレス・インクリメントボタン：INC

メモリ領域を編集する際, アドレスの値を 4 つインクリメントする．

2.2.8 メモリアドレス書換えボタン：ADR

メモリのアドレスを 16 進数キーボードで入力した値に変更する．

2.2.9 プログラム実行モード切換えボタン：Mod

プログラムを「パイプライン」モードで実行するか，「シーケンシャル」モードで実行するかを選択する．Mode ボタンを押す度に，「パイプライン」モードと「シーケンシャル」モードに交互に切り替わる．蛍光表示管：一行目に「PL」が表示されていると「パイプライン」モード，「SC」が表示されていると「シーケンシャル」モードとなる．

2.2.10 プログラム実行／停止ボタン：SS

RUECHIP2 CPU がプログラムの実行が停止している「HALT」状態で「SS」ボタンを押すと，プログラムの解釈実行が開始され，プログラムが連続して実行される「RUN」状態になる．「RUN」状態で「SS」ボタンを押すと，「HALT」状態に移行しプログラムの実行を停止する．

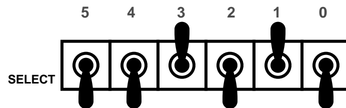
2.2.11 1 命令実行ボタン：SI

「パイプライン」モードで「SI」ボタンを押すと１クロックだけ命令の解釈実行が「パイプライン処理」で行われる．「パイプライン」モードで１命令の解釈実行を完了するには、「SI」ボタンを５回押すことが必要である．「シーケンシャル」モードでは、１命令だけ「逐次的に」命令の解釈実行が行われる．

2.2.12 レジスタ書込みボタン：Reg

6 bit トグル SW_SEL で選択した RUECHIP2 内部の 32bit レジスタ（PC もしくは汎用レジスタ R01〜R31）の値が, 16 進数キーボードで入力した値で書換えられる.

2.2.13　観測レジスタ切替スイッチ：SELECT (6bit)



観測／書換えを行う RUECHIP2 CPU 内部のレジスタ（もしくはフォワーディング出力、分岐の成立 or 不成立と分岐先候補のアドレス）を指定する．スイッチの値と観測対象の関係を表 2.1 に示す．◎のついたレジスタは、Reg ボタンを押すことで、16 進キーボードで指定した値で書換えることができる.

SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象
000000	PC ◎	000001	GPR:$1 ◎	000010	GPR:$2 ◎	000011	GPR:$3 ◎
000100	GPR:$4 ◎	000101	GPR:$5 ◎	000110	GPR:$6 ◎	000111	GPR:$7 ◎
001000	GPR:$8 ◎	001001	GPR:$9 ◎	001010	GPR:$10 ◎	001011	GPR:$11 ◎
001100	GPR:$12 ◎	001101	GPR:$13 ◎	001110	GPR:$14 ◎	001111	GPR:$15 ◎
010000	GPR:$16 ◎	010001	GPR:$17 ◎	010010	GPR:$18 ◎	010011	GPR:$19 ◎
010100	GPR:$20 ◎	010101	GPR:$20 ◎	010110	GPR:$22 ◎	010111	GPR:$23 ◎
011000	GPR:$24 ◎	011001	GPR:$25 ◎	011010	GPR:$26 ◎	011011	GPR:$27 ◎
011100	GPR:$28 ◎	011101	GPR:$29 ◎	011110	GPR:$30 ◎	011111	GPR:$31 ◎
100000	IF PC	100001	IF IR	100010	ID PC	100011	ID IR
100100	ID Rs	100101	ID Rt	100110	EX PC	100111	EX IR
101000	EX C	101001	EX HI	101010	EX LO	101011	EX SMDR
101100	MEM PC	101101	MEM IR	101110	MEM C	101111	EPC
110000	JMP FLAG	110001	JMPADDR	110010	Fw RS ADR	110011	FW RT ADR
110100	FW EX FLG	110101	FW EX RD	110110	FW EX DAT	110111	FW MEM FLG
111000	FW MEM RD	111001	FW MEM DAT	111010	FW WB FLG	111011	FW WB RD
111100	FW WB DAT	111101	CAUSE	111110		111111	

表 2.1　観測／書換え可能なレジスタの指定（※ GPR：汎用レジスタ）

2.2.14　動作クロック周波数切替スイッチ：CLK FRQ

RUECHIP2 CPU の動作クロック周波数を切替えを行う．CLKFRQ の目盛りと動作周波数の関係を表 2.2 に示す.

CLKFRQ	動作クロック	CLKFRQ	動作クロック	CLKFRQ	動作クロック	CLKFRQ	動作クロック
0	10 MHz	1	333 kHz	2	100 kHz	3	33 kHz
4	10 kHz	5	3 kHz	6	1 kHz	7	333 Hz
8	100 Hz	9	33 Hz	10	10 Hz	11	3.3 Hz
12	1 Hz	13	0.33 Hz	14	0.10 Hz	15	0.033 Hz

表 2.2 動作クロック周波数の設定

2.2.15 **制御／観測** CPU **の選択：**SW_SELECT (**スライド** SW)

KUE-CHIP2, KUECHIP-3F, RUECHIP のどの CPU を制御／観測するかを切り替えるためのスイッチである．RUECHIP2 CPU の動作を制御／観測するときには, 右側に設定する．左側に設定すると, KUE-CHIP2, KUECHIP-3F が制御／観測の対象になる．本スイッチの設定を変更した時は POWER スイッチで電源をオン／オフし, KR-CHIP 教育用ボードを再起動する必要がある．

2.2.16 R8C29 **マイコンのファームウェア書換え：**MODE (**スライド** SW)

RUECHIP2 CPU を制御／観測するとき, MODE スイッチを左側に設定する．KR-CHIP 教育用ボード上の R8C29 マイコン (U04) のプログラムを書換える時だけ, 右側にする．

2.3　KR-CHIP 教育用ボードのコネクタ

以下では, KR-CHIP 教育用ボードのコネクタの機能について説明する.

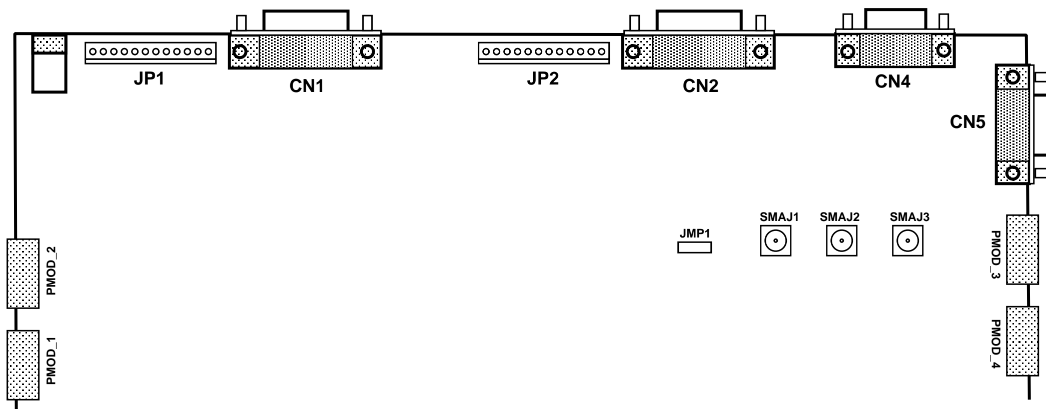
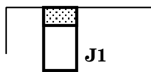


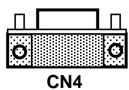
図 2.6　KR-CHIP 教育用ボードのコネクタ

2.3.1　電源コネクタ：J1



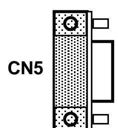
電圧 5.0V, 電流容量 1.8A の直流安定化電源に接続する. 極性はセンター＋ (プラス) で, センター (ピン) の直径は 2.1mm である.

2.3.2　RS232C コネクタ：CN4



PC とのシリアル通信により、RUECHIP2 CPU のプログラムの書き込み／命令実行の制御／ CPU の内部観測を行うときに利用する. 通信の方法は 7 章で説明する.

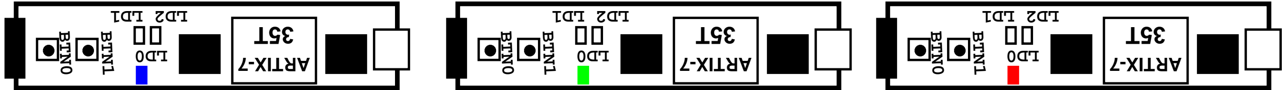
2.3.3　RS232C コネクタ：CN5



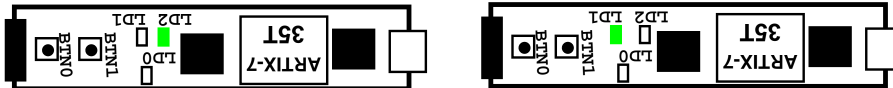
KR-CHIP 教育用ボード上の R8C29 マイコンの制御プログラムを書き換える際に, RS232C ケーブル (クロス) で Windows PC と接続する.

2.3.4 Cmod A7 上の LED

LD0 が青色に点灯しているときは通常の実行モードとなり, 緑または赤色に点灯しているときは PC との接続モードとなる.



LD1 と LD2 はクロック周波数に同期して交互に点滅する.



第 3 章

RUECHIP2 CPU の仕様

ここでは RUECHIP2 CPU のメモリマップ, また制御コマンドを示す.

3.1 RUECHIP2 CPU **のメモリマップ**

MIPS のメモリマップを図 3.1 に示す．MIPS は仮想アドレス空間内のアドレスを物理アドレスに変換することで, CPU の物理メモリ空間を論理的に拡張する．MIPS では, 通常仮想アドレス空間で 4 種類のセグメントが使用できる．KR-CHIP に含まれる MIPS のメモリマップを図 3.2 に示す．
KR-CHIP に含まれる MIPS は, そのうち kseg0, kseg1 のみをサポートしている．kseg0, kseg1 は通常の MIPS の仕様で以下の様になっている．

- kseg0: kseg0 は物理アドレス空間の最初の 512MB に直接マップされている．これらの参照はキャッシュメモリを使用する．実行可能コードと一部のカーネルデータに使用される．
- kseg1: kseg1 も kseg0 と同様に物理アドレス空間の最初の 512MB に直接マップされている．しかし参照にはキャッシュを使用しない．kseg1 は通常入出力レジスタ, ROM およびディスクバッファに使用する．

KR-CHIP に含まれる MIPS は SRAM が 512B のため, kseg0 においては x8000_0000〜x8008_0000, kseg1 においては 0xA000_0000〜0xA008_000 のみが有効である．物理アドレスは 0x0000_000〜0x0008_0000 が有効である．

RUECHIP2 CPU がリセットされると, PC は 0x8000_0000 番地に非同期的にセットされるそのため, 開始直後プログラムは 0x8000_0000(物理メモリ 0x0000_0000) から読み出されキャッシュされる．

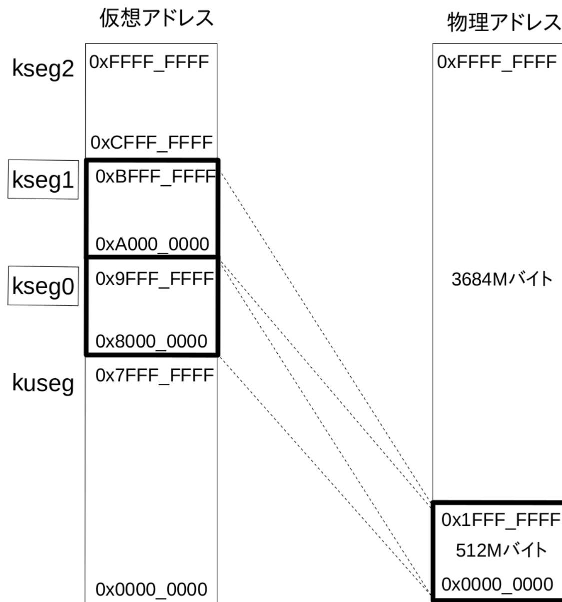


図 3.1 MIPS のメモリマップ

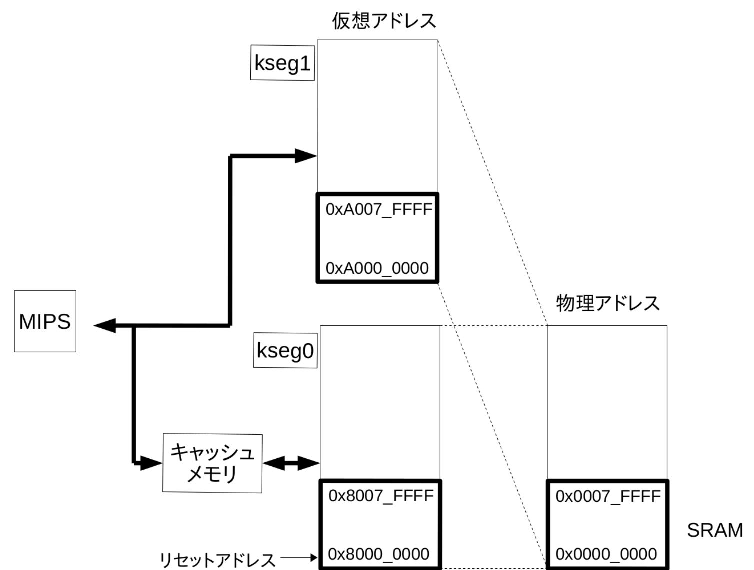


図 3.2 RUECHIP2 CPU のメモリマップ

また、CPU のメモリアクセスにおけるバイト順序は 32bit のリトルエンディアンである．そのため, メモリ内でのデータは表 3.1 の様に配置される．

表 3.1 メモリ内のデータの配置

bit 31-24 23-16 15-8 7-0

ワード アドレス	バイトアドレス				C 記述の例
	値				
0x0008	11	10	9	8	int i = 0x12345678
	0x12	0x34	0x56	0x78	
0x0004	7	6	5	4	unsigned short us[2] = { 0x1234, 0x5678 }
	0x56	0x78	0x12	0x34	
0x0000	3	2	1	0	unsigned char uc[4] = { 0x12, 0x34, 0x56, 0x78 }
	0x78	0x56	0x34	0x12	

3.2 RUECHIP2 CPU 制御コマンド

RUECHIP2 CPU は, その動作と内部レジスタの観測／書換えを, RS232C のシリアル通信で制御する. RUECHIP2 CPU 自主的に通信を開始することはなく, シリアル通信で送られてきたコマンドに従って動作する. KR-CHIP 教育ボードのほかに, Windows PC の Hyperterminal, Teraterm 等のターミナルソフトウェア経由からコマンドを入力することで, 動作を制御することができる.

3.2.1 RS232C 通信の仕様

ボーレート：9600bps
ストップビット：1bit
パリティ：なし
フロー制御：なし

3.2.2 RS232C 通信のコマンド

メモリ読み出しコマンド：'R'AAAAAAAA

メモリのアドレス A（32bit）に格納されている値（32bit）を読み出し, 出力する. 出力は 16 進数表記の 8 文字である. アドレスの下位 2bit は無視される.

メモリ書き込みコマンド：'W'AAAAAAAA':'DDDDDDDD

メモリのアドレス A（32bit）にデータ D（32bit）を書き込む. 読み出しと同様に, アドレスの下位 2bit を無視する. 書き込み結果（成功・不成功）等は出力されない. 書き込み結果を確かめたい場合は, メモり読み出しコマンドを使って確認する.

レジスタ読み出しコマンド：'r'AA

レジスタ番号 A（8bit）に格納されている値（32bit）を読み出し, 出力する. 出力は 16 進数表記の 8 文字である. レジスタ番号は以下のようになっている（番号は 16 進数）.
00：PC（プログラムカウンタ）
01〜1F：GPR（汎用レジスタ：$01〜$31）

20：IF ステージ PC, 21：IF ステージ IR
22：ID ステージ PC, 23：ID ステージ IR, 24：ID ステージ Rs, 25：ID ステージ Rt
26：EX ステージ PC, 27：EX ステージ IR, 28：EX ステージ C
29：EX ステージ HI, 2A：EX ステージ LO, 2B：EX ステージ SMDR
2C：MEM ステージ PC, 2D：MEM ステージ IR, 2E：MEM ステージ C
2F：EPC, 30：分岐の成立 (1)・不成立 (0), 31：分岐先候補アドレス
32：ID ステージ RS アドレス, 33：ID ステージ RT アドレス,
34：EX ステージフォワーディングフラグ, 35：EX ステージ RD, 36：EX ステージフォワーディングデータ
37：MEM ステージフォワーディングフラグ, 38：MEM ステージ RD, 39：MEM ステージフォワーディングデータ
3A：WB ステージフォワーディングフラグ, 3B：WB ステージ RD, 3C：WB ステージフォワーディングデータ

レジスタ書き込みコマンド：'w'AA:'DDDDDDDD

レジスタ番号 A（8bit）にデータ D（32bit）を書き込む. 書き込み結果（成功・不成功）等は出力されない. 書き込み結果を確かめたい場合は, レジスタ読み出しコマンドを使って確認する. レジスタ番号は以下のようになっている（番号は 16 進数）.
00：PC（プログラムカウンタ）
01〜1F：GPR（汎用レジスタ

イネーブルパルス設定コマンド：'P'DDDDDDDD

イネーブルパルスの出力間隔を D に変更する. 出力間隔は (50,000,000 / (D *128))Hz で与えられる. D が 390625（16 進数で 0005F5E1）で約 1Hz. D が 0 のとき, イネーブルパルスの出力を停止する. D が 0 以外になったとき, 自動的に出力を開始する.

イネーブルパルス 1 回出力コマンド：'p'

イネーブルパルスが 1 回だけプロセッサに送られ, プロセッサは命令の解釈実行を 1 クロックだけ実行する. 1 クロックが出力されるまで, プロセッサの動作の制御が停止する (一瞬のみでプロセッサの動作には影響しない) ため, 他のコマンドの受け取りや, 入出力の状態の更新が停止する. PC のターミナルソフトウェアから連続でコマンドを送る際などは注意が必要. イネーブルパルス設定コマンドで定期的にパルスを送っている間でも実行できるが, 推奨できない. パルスを停止してから実行すべきである.

リセットコマンド：'C'

RUECHIP1 プロセッサ（の内部レジスタ）を非同期的にリセットする. 各レジスタの値は 0x0000 0000 でクリアされ, PC は 0x8000 0000 に戻る. イネーブルパルスの設定や, メモリ内容はそのまま維持されるため注意が必要である (リセット直後から動き出すこともありえる).

シングルサイクルモード設定コマンド：'S'

RUECHIP2 CPU の実行モードを,
・「パイプライン」モードの時は「シーケンシャル」モードに変更
・「シーケンシャル」モードの時は「パイプライン」モードに変更
する. パワー ON（もしくは再コンフィグレーション）直後は, 「パイプライン」モードに設定されている. このコ

マンドにより，「シーケンシャル」モードに変更された時には”01”が，「パイプライン」モードに変更された時には”00”が出力される．出力は 16 進数表記の 2 文字ですので注意が必要である．

シリアル入力コマンド：'I'DD

シリアル通信で, D を入力します．受け取った値をバッファに書き込めた場合, ”01”を返す．書き込めない場合, ”00”を返す．一度バッファに値を書き込むと, RUECHIP1 プロセッサから読み出されない限り, 書き込めない．書き込んだ値を RUECHIP1 プロセッサから使用する場合, ライブラリの API を用いて読み出す必要がある．

第 4 章

プログラムの入力と実行

本章では, 実際に RUECHIP2 CPU 上でプログラムを実行し, その挙動を観察する.

4.1 例題

本章では以下の例題を扱う. これは 0 〜 N-1 までの総和を計算するプログラムである.

```
LW    $5,0x0($28)
ADDIU $3,$0,0x1
ADDIU $4,$0,0x0
ADDU  $4,$4,$3
ADDIU $3,$3,0x1
SLT   $2,$3,$5
BNE   $2,$0,0xFFFC
SW    $4,0x4($28)
```

図 4.1 プログラム

4.2 入力

それでは実際にプログラムを入力する. 以下では, RUECHIP2 CPU を非同期リセットした際, PC に設定されるアドレス：0x80000000 からプログラムを入力することにする.

はじめに, プログラムを入力するアドレスの設定を行う. RUECHIP2 制御/観測ボード上の 16 進数入力キーボード (16 進キーボード) から 8000 0000 と入力する.

[8] [0] [0] [0] [0] [0] [0] [0]

ディスプレイの四行目に, 入力した値が反映される.

```
HALT PL   PC:80000000
          PC:80000000
ADR 00000000
input> 80000000
```

値を確認し, アドレス書き換えボタン (ADR キー) を押す.

ADR

ディスプレイの三行目の ADR に値が反映され, メモリの 0x80000000 に入力が行うことができるようになる.

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
MEM 00000000
```

次に, メモリにプログラムを書き込む. 最初の命令である `LW $5,0x0($28)` は, 16 進数で 0x8F850000 で表される. 16 進キーボードから 8F850000 を入力する.

8 F 8 5 0 0 0 0

ディスプレイの四行目に入力した値が表示される.

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
input> 8F850000
```

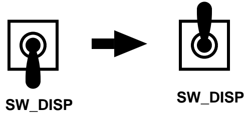
確認後, メモリ編集ボタン (SET キー) を押す.

SET

0x80000000 番地の値が書き換えられ, ディスプレイの四行目にその値が表示される.

表示切り替えスイッチ (SW_DISP) で, 入力した値とその逆アセンブルの表示切り替えが可能.

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
MEM 8F850000
```




```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
MEM 8F850000
```

→

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
LW $5,0x0($28)
```

ここまでで最初の命令をメモリに書き込むことが出来た.
次の命令を入力する. アドレスインクリメントボタン (INC キー) を押す.

INC

ディスプレイの ADR が 80000004 となって, ここに書き込むことが出来るようになる.
(同様に, DEC キーを押すことで アドレスを戻すことができる).

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000004
MEM 00000000
```

`ADDIU $3,$0,0x1` は, 16 進数で 0x24030001 で表される. 先程と同じ手順で入力する.

2 4 0 3 0 0 0 1 SET

続く命令も同様に入力を行う. 以下の表 4.2 に全命令の番地と, 入力する 16 進数の値を示す.

4.3 実行

本章では, 入力したプログラムを非パイプラインモードで実行し, その結果を観察する.
プログラム実行前に, 初期設定を行う.
始めにプログラム実行モードの設定を行う. ディスプレイの一行目を確認する. 起動直後は PC PL PC:80000000 となっている.

```
80000000: 8F850000  LW    $5,0x0($28)
80000004: 24030001  ADDIU $3,$0,0x1
80000008: 24040000  ADDIU $4,$0,0x0
8000000c: 00832021  ADDU  $4,$4,$3
80000010: 24630001  ADDIU $3,$3,0x1
80000014: 0065102A  SLT   $2,$3,$5
80000018: 1440FFFC  BNE   $2,$0,0xFFFC
8000001c: AF840004  SW    $4,0x4($28)
```

図 4.2 プログラムのアドレスと値

```
HALT PL   PC:80000000
          PC:80000000
ADR 8000001C
MEM AF840004
```

PC 横の文字は実行モードを表しており, P はパイプラインモードで実行することを示している. 実行モード切り替えボタン (Mod キー) を押し,

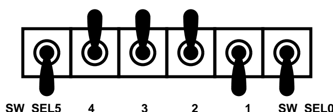
Mod

非パイプラインモードに切り替える. PC 横の文字が SC となる.

```
HALT SC   PC:80000000
          PC:80000000
ADR 8000001C
MEM AF840004
```

本章で入力したプログラムは, レジスタ $28 が示すアドレスから値を取得する. そこで, レジスタ $28 の値を設定と, 該当するメモリアドレスの編集を予め行う.

まずレジスタの設定を行う. 観測レジスタ切り替えスイッチ (SW_SEL) を左から 011100 に設定する.



ディスプレイ二行目に R28 と表示される.

次に, 16 進キーボードから A0000020 を入力し, レジスタ書き込みボタン (Reg キー) を押す.

```
HALT SC   PC:80000000
        R28:00000000
ADR 8000001C
MEM AF840004
```

A 0 0 0 0 0 2 0 Reg

レジスタ $28 に値が設定され, ディスプレイに表示される.

```
HALT SC   PC:80000000
        R28:A0000020
ADR 8000001C
MEM AF840004
```

次にメモリに値を書き込む. プログラム入力と同じ手順である. 書き込みを行うアドレスを設定する. 16 進キーボードから A0000020 を入力し, Adr キーを押す.

A 0 0 0 0 0 2 0 Adr

```
HALT SC   PC:80000000
        R28:A0000020
ADR A0000020
MEM 00000000
```

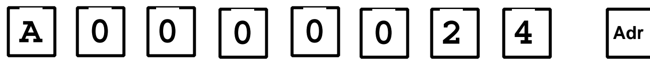
次に, 書き込む値を設定する. 本章のプログラムは, ここで入力する値を N として, 0 〜 N-1 までの総和を計算する. 入力する値は任意の値で構わないが, 例として 00000004 を入力し, SET キーを押す.
ここまでで実行前の設定は完了.

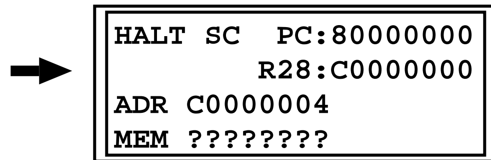
0 0 0 0 0 0 0 4 SET

➡

```
HALT SC   PC:80000000
        R28:A0000020
ADR A0000020
MEM 00000004
```

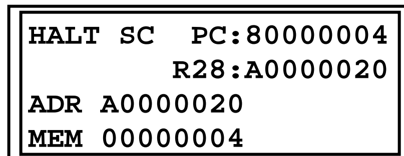
なお, 本プログラムは最後の命令 `SW $4,0x4($28)` で, 計算した値をメモリに格納する. プログラム実行前に A0000024 にアドレスを設定し, 値を観察しながら実行すると良い.



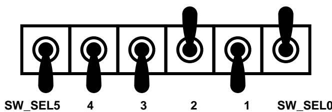


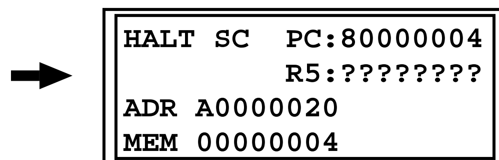
プログラムを実行する. 1 命令/1 サイクル実行ボタン (SI キー) を押す.

80000000 番地の命令が実行され, PC の値が 80000004 となる.



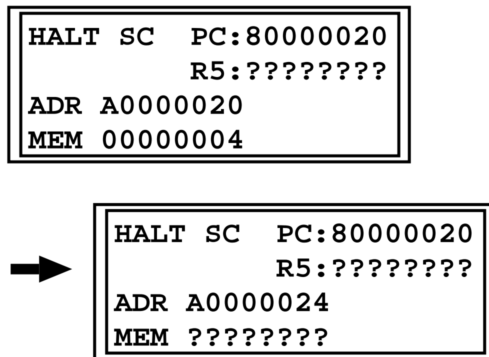
SW_SEL を 000101 に設定し, レジスタ $5 の値を確認する.





以降, 同様に SI キーを押して命令を実行, 適宜レジスタを観察する. 最後に PC の値が 80000020 となれば実行完了.

メモリ A0000024 番地を確認し, 正しい値が格納されていることを確認する.



「シーケンシャル」モード実行時の RUECHIP2 CPU のブロックダイアグラムを以下の図 4.3 に示す.

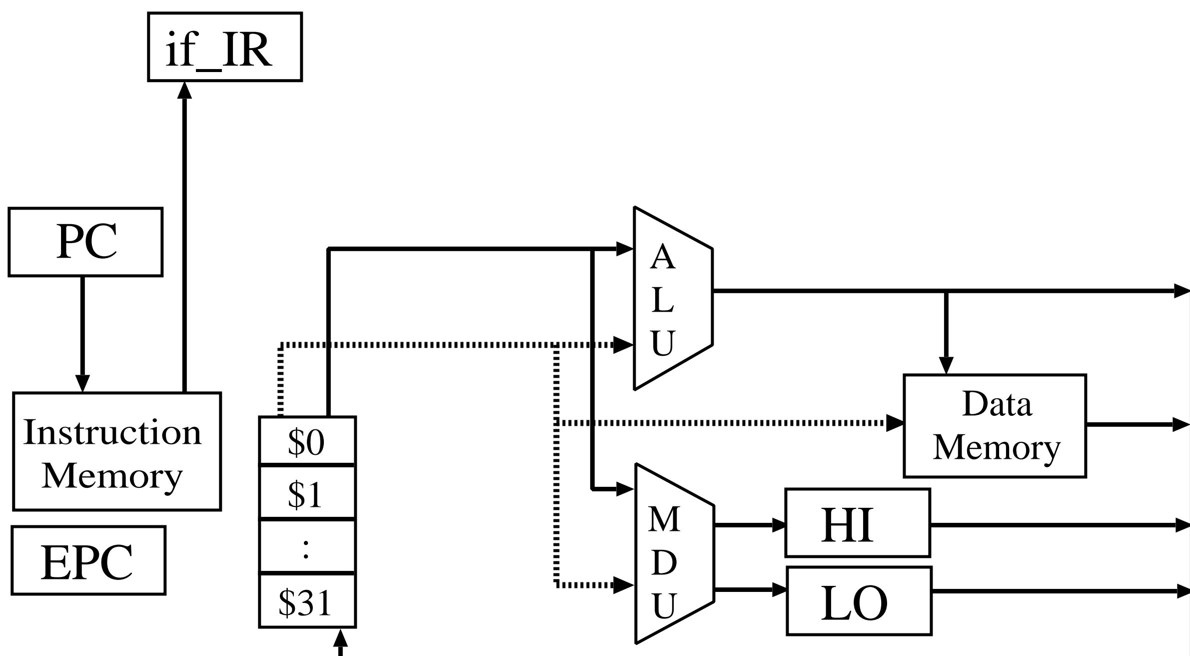


図 4.3　「シーケンシャル」モード実行時のブロックダイアグラム

「シーケンシャル」モードで観測／書換えを行う RUECHIP2 CPU 内部のレジスタと SW_SEL スイッチの値の関係を表 4.1 に示す. ◎のついたレジスタは, Reg ボタンを押すことで, 16 進キーボードで指定した値で書換えることができる.

SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象
000000	PC ◎	000001	GPR:$1 ◎	000010	GPR:$2 ◎	000011	GPR:$3 ◎
000100	GPR:$4 ◎	000101	GPR:$5 ◎	000110	GPR:$6 ◎	000111	GPR:$7 ◎
001000	GPR:$8 ◎	001001	GPR:$9 ◎	001010	GPR:$10 ◎	001011	GPR:$11 ◎
001100	GPR:$12 ◎	001101	GPR:$13 ◎	001110	GPR:$14 ◎	001111	GPR:$15 ◎
010000	GPR:$16 ◎	010001	GPR:$17 ◎	010010	GPR:$18 ◎	010011	GPR:$19 ◎
010100	GPR:$20 ◎	010101	GPR:$20 ◎	010110	GPR:$22 ◎	010111	GPR:$23 ◎
011000	GPR:$24 ◎	011001	GPR:$25 ◎	011010	GPR:$26 ◎	011011	GPR:$27 ◎
011100	GPR:$28 ◎	011101	GPR:$29 ◎	011110	GPR:$30 ◎	011111	GPR:$31 ◎

表 4.1　「シーケンシャル」モードで観測／書換え可能なレジスタの指定（※ GPR：汎用レジスタ）

第 5 章

パイプラインの観察

本章では, パイプラインモードでプログラムを実行し, その挙動を観察する.

5.1 パイプラインレジスタの観察

はじめに以下の命令を入力する.

```
80000000: 24030001  ADDIU $3,$0,0x1
80000004: 00000000  SLL   $0,$0,0x0
80000008: 00000000  SLL   $0,$0,0x0
8000000c: 00000000  SLL   $0,$0,0x0
80000010: 00000000  SLL   $0,$0,0x0
```

図 5.1　プログラムのアドレスと値

これはレジスタ $3 に値 0x1 を設定するだけのプログラムです. 80000004 番地以降の SLL $0,$0,0x0(NOP 命令) で不確定な動作を消去している. 以降の例題では明記しないが, 必要であれば最後の命令以降に NOP を書き込む.

プログラムの入力を行う. (入力に関して, 詳細な説明は “第 3 章 プログラムの入力と実行” を参照. ここでは手順を追うにとどめる).

プログラムを書き込む位置を設定する. KR-CHIP の 16 進キーボードから 80000000 を入力し, ADR キーを押す. ディスプレイの三行目に値が反映される.

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
MEM 00000000
```

設定したメモリアドレス (0x80000000) にプログラムを書き込む. 16 進キーボードから 24030001 (ADDIU $3,$0,0x1) を入力し, SET キーを押す. ディスプレイの四行目に入力した値, もしくはその逆アセンブリが表示される (SW_DISP スイッチで切り替え可能) .

INC キーを押してアドレスを進め, 同様に入力を続ける.

SET

INC

プログラムの入力が完了したら, プログラムの実行モードを設定する. 本章ではプログラムをパイプラインモードで動作させる.

プログラム実行モードをパイプラインモード, プログラムカウンタを 0x80000000 に設定する. ディスプレイの一行目が HALT PL PC:80000000 となっていることを確認する. (※起動直後はこの状態になっている)

```
HALT PL  PC:80000000
         PC:80000000
ADR 8000001C
MEM AF840004
```

なっていないようであれば, プログラム実行モードは Mod キーで切り替える.

Mod

また, プログラムカウンタは SW_SEL スイッチを左から 100000 とし, 16 進キーボードから 80000000 を入力, 最後に Reg キーを押すことで設定する.

最後に, 観測のためレジスタ $3 (SW_SEL スイッチは 000101) の値を 0x00000001 以外の値（例えば 0x00000000）に設定する.

以上で設定は完了.

```
HALT PL  PC:80000000
         R3:00000000
ADR 8000001C
MEM AF840004
```

SI キーを押してプログラムを実行する.

SI

レジスタ $3 の値を観察する. また, 各パイプラインレジスタを観察することができる. 例として, SW_SEL を 100010 に設定することで IF PC (IF ステージでのプログラムカウンタ) の値を観察できる. 10011 で IF IR (IF ステージでの 命令レジスタ) を観察できる.

その他, 観察可能なパイプラインレジスタを表 5.1 に示す.

再度 SI キーを押し, 再度各パイプラインレジスタの値を観察し, その挙動を確認する. 先ほど確認した IF PC(100010), IF IR(100011) に加えて, ID PC (100100), ID IR(100101), ID Rs(101010), ID Rt(101100) 等を観測してみる.

以降, Si キーを押して同様に実行, 観察を行う. EX C (101110), MEM C(110010), そして レジスタ $3 と値を

SW_SEL	観測対象	SW_SEL	観測対象
100000	PC	100001	
100010	IF PC	100011	IF IR
100100	ID PC	100101	ID IR
100110	EX PC	100111	EX IR
101000	MEM PC	101001	MEM IR
101010	ID RS	101011	ID RsAddr
101100	ID Rt	101101	ID RtAddr
101110	EX C	101111	EX SMDR
110000	EX LO	110001	EX HI
110010	MEM C	110011	EPC
110100	BR Cond	110101	BR Addr
110110	EX FwdD	110111	EX FwdF/Rd
111000	MEM FwdD	111001	MEM FwdF/Rd
111010	WB FwdD	111011	WB FwdF/Rd

表 5.1 パイプラインレジスタの観測

追っていくと良い. また, RUECHIP2 CPU は IF, ID, EX, MEM, WB の 5 段パイプラインとなっていることに注意して, 表 5.1 を参照しながら適切なパイプラインレジスタを観察し, 各データがレジスタを流れていく様子を観察する. SI キーを合計 5 回押し, プログラムカウンタの値が 0x80000014 になると, レジスタ $03 に値が書き込まれ, 命令 `ADDIU $3,$0,0x1` の実行が完了する.

5.2 フォワーディングの観察

本節ではフォワーディングの観察を行う. 図 5.2 のプログラムを入力, 実行する.

```
80000000: 24030001  ADDIU $3,$0,0x1
80000004: 24040002  ADDIU $4,$0,0x2
80000008: 00832021  ADDU  $4,$4,$3
```

図 5.2 プログラムのアドレスと値

特に以下のことに注意する.

実行を進め, プログラムカウンタが 0x80000010 に達した時点で, 2 行目の命令の EX ステージが実行される. EX C (101110) を確認する.

この時, 同時に 3 行目の命令の ID ステージが実行される. ID ステージでは, EX ステージの計算に必要となる値が取得される. この命令では $3, $4 レジスタからの値を取得しているが, 1, 2 行目の命令で, $3, $4 レジスタへの書き込みを行っている. 前節での観察から, 現時点で両レジスタの値がどうなっているか予測した上で, 両レジスタを観察する. 次に, 両レジスタの値が取得される ID RS(101010), ID Rt(101100) を観察する.

最後に, EX FwdD(110110) を観察する.

このときプロセッサで何が起こっているのか考察せよ. 章末のプロセッサダイアグラム図を合わせて参照すると

良い. プロセッサはパイプラインを壊さないように実行を進める.

5.3 分岐の観察

本節では分岐の観察を行う. 図 5.3 のプログラムを入力する. このプログラムは前章のプログラムと全く同一のものである. 必要に応じて前章を参照する. このプログラムは　0 〜N-1 の総和を計算するプログラムである. レジスタ $28 が示すメモリアドレスから N の値を取得し, その次のアドレスに結果を代入する.

```
80000000: 8F850000  LW    $5,0x0($28)
80000004: 24030001  ADDIU $3,$0,0x1
80000008: 24040000  ADDIU $4,$0,0x0
8000000c: 00832021  ADDU  $4,$4,$3
80000010: 24630001  ADDIU $3,$3,0x1
80000014: 0065102A  SLT   $2,$3,$5
80000018: 1440FFFC  BNE   $2,$0,0xFFFC
8000001c: AF840004  SW    $4,0x4($28)
```

図 5.3　プログラムのアドレスと値

プログラムの実行を行う. メモリ 0xA0000020 に N の値 (4 など), レジスタ $28 に 0xA0000020 を設定し, 実行する. メモリアドレス 0x8000001C の命令で, 0xA0000024 に総和が代入される. その直前の 0x80000018 は分岐命令で, この命令を実行することで分岐が生じる. 0x8000001c の命令はいつ実行されるか, 0xC0000024 を観察してその実行タイミングを観察せよ.

次に, 図 5.4 を入力する. 前例とほぼ同一ですが, メモリアドレス 0x8000001C に NOP 命令が挿入されている.

```
80000000: 8F850000  LW    $5,0x0($28)
80000004: 24030001  ADDIU $3,$0,0x1
80000008: 24040000  ADDIU $4,$0,0x0
8000000c: 00832021  ADDU  $4,$4,$3
80000010: 24630001  ADDIU $3,$3,0x1
80000014: 0065102A  SLT   $2,$3,$5
80000018: 1440FFFC  BNE   $2,$0,0xFFFC
8000001c: 00000000  SLL   $0,$0,0x0
80000020: AF840004  SW    $4,0x4($28)
```

図 5.4　プログラムのアドレスと値

これはメモリ 0xA000024 に N の値 (4 など), レジスタ $28 に 0xA0000024 を設定し, 同様に実行します. メモリアドレス 0xA0000028 の値に注意して実行を行う.

パイプラインモードで観測可能なレジスタの一覧を以下の表 5.2 に示す．◎のついたレジスタは, Reg ボタンを押すことで 16 進キーボードで指定した値で上書きすることができる．

SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象	SW_SEL	観測対象
000000	GPR:$0	000001	GPR:$1 ◎	000010	GPR:$2 ◎	000011	GPR:$3 ◎
000100	GPR:$4 ◎	000101	GPR:$5 ◎	000110	GPR:$6 ◎	000111	GPR:$7 ◎
001000	GPR:$8 ◎	001001	GPR:$9 ◎	001010	GPR:$10 ◎	001011	GPR:$11 ◎
001100	GPR:$12 ◎	001101	GPR:$13 ◎	001110	GPR:$14 ◎	001111	GPR:$15 ◎
010000	GPR:$16 ◎	010001	GPR:$17 ◎	010010	GPR:$18 ◎	010011	GPR:$19 ◎
010100	GPR:$20 ◎	010101	GPR:$20 ◎	010110	GPR:$22 ◎	010111	GPR:$23 ◎
011000	GPR:$24 ◎	011001	GPR:$25 ◎	011010	GPR:$26 ◎	011011	GPR:$27 ◎
011100	GPR:$28 ◎	011101	GPR:$29 ◎	011110	GPR:$30 ◎	011111	GPR:$31 ◎
100000	PC ◎	100001		100010	IF PC	100011	IF IR
100100	ID PC	100101	ID IR	100110	EX PC	100111	EX IR
101000	MEM PC	101001	MEM IR	101010	ID RS	101011	ID RsAddr
101100	ID Rt	101101	ID RtAddr	101110	EX C	101111	EX SMDR
110000	EX LO	110001	EX HI	110010	MEM C	110011	EPC
110100	BR Cond	110101	BR Addr	110110	EX FwdD	110111	EX FwdF/Rd
111000	MEM FwdD	111001	MEM FwdF/Rd	111010	WB FwdD	111011	WB FwdF/Rd

表 5.2　パイプラインレジスタの観測（※ GPR：汎用レジスタ）

最後に,「パイプライン」モード実行時の RUECHIP2 CPU のブロックダイアグラムを図 5.3 に示す．またこのブロックダイアグラムのフォワーディング機能について拡大した図を図 5.3 にに示す．

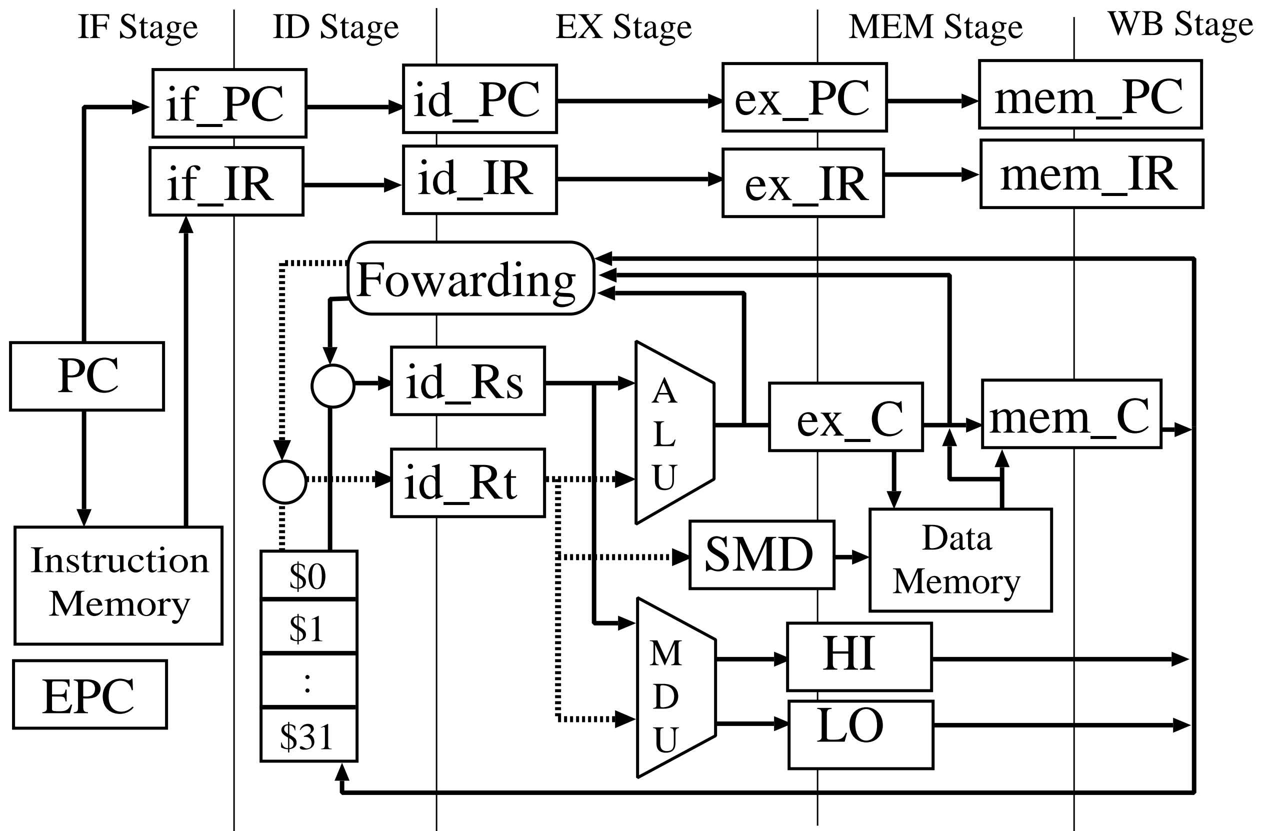


図 5.5　「パイプライン」モード実行時のブロックダイアグラム

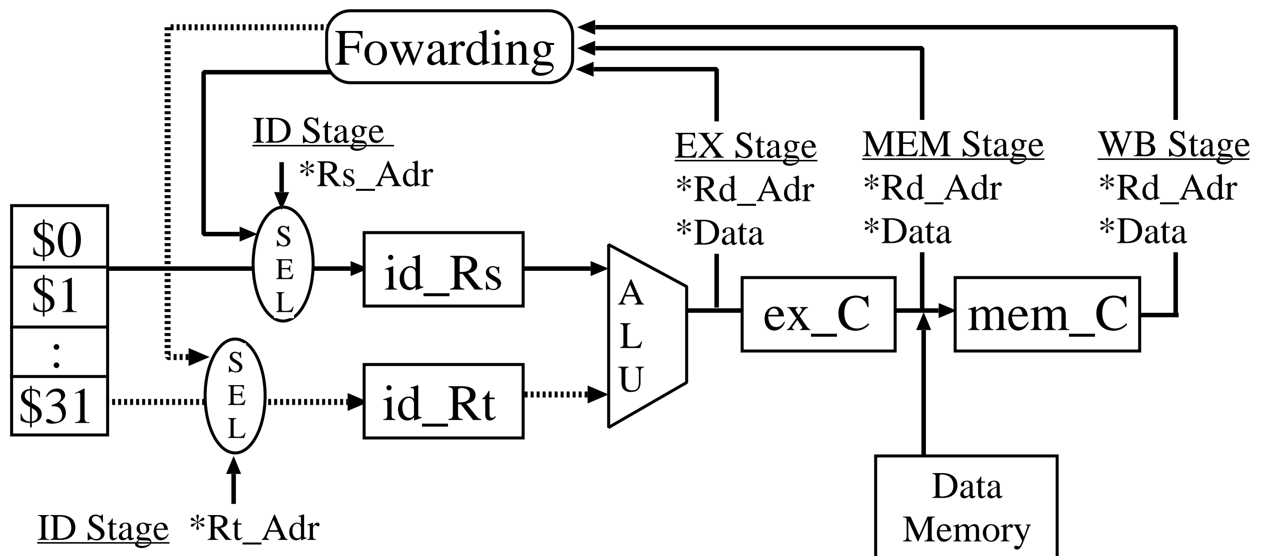


図 5.6　「パイプライン」モード実行時のフォワーディング機能の拡大図

第 6 章

RUECHIP2 命令セット

この章では, RUECHIP2 の命令セットの

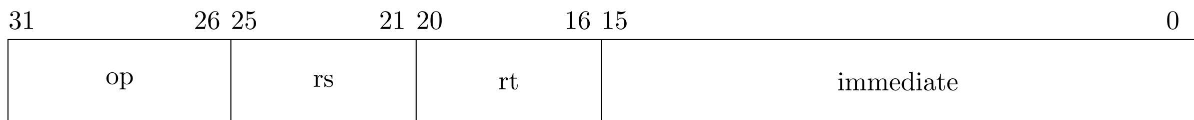
- 命令の形式
- オペコードの割り当て
- ロード命令
- ストア命令
- ALU 即値命令
- 3 オペランド・レジスタタイプ命令
- 乗除算命令
- ジャンプ命令
- 分岐命令
- コプロセッサ命令

を取り上げる.

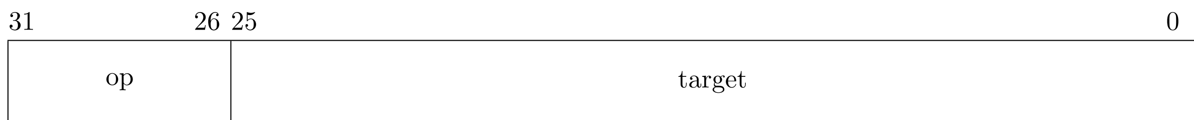
6.1 命令形式

すべての命令は，ワード境界に位置合わせされた単一ワード（32 ビット）で構成される．命令形式は次の 3 種類しかない．

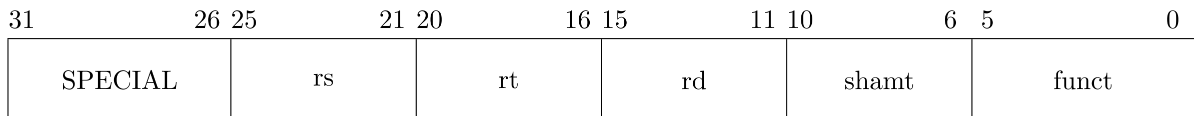
○ I タイプ（イミディエト）



○ J タイプ（ジャンプ）



○ R タイプ（レジスタ）



v

ただし，

op	6 ビットの命令コード
rs	5 ビットのソースレジスタ指定子
rt	5 ビットのターゲット（ソース／デスティネーション）レジスタまたは分岐条件
immediate	16 ビットのイミディエト，分岐ディスプレースメント，またはアドレス・ディスプレースメント
target	26 ビットの無条件分岐ターゲットアドレス
rd	5 ビットのデスティネーション・レジスタ指定子
shamt	5 ビットのシフト量
funct	6 ビットの機能フィールド

6.2　オペコードの割り当て

表 6.1　Opcode

IR[27..26]	00	01	10	11
IR[31..28]				
0000	*SPECIAL*	*BCOND*	J	JAL
0001	BEQ	BNE	BLEZ	BGTZ
0010	ADDI	ADDIU	SLTI	SLTIU
0011	ANDI	ORI	XORI	LUI
0100	*COP0*			
0101	<u>ABSI</u>			
0110				
0111				
1000	LB	LH	LWL	LW
1001	LBU	LHU	LWR	
1010	SB	SH	SWL	SW
1011			SWR	
1100				
1101				
1110				
1111				

表 6.2　*BCOND*

IR[17..16]	00	01	10	11
IR[20..18]				
000	BLTZ	BGEZ		
001				
010				
011				
100	BLTZAL	BGEZAL		
101				
110				
111				

表 6.3 *SPECIAL*

IR[1..0]	00	01	10	11
IR[5..2]				
0000	SLL		SRL	SRA
0001	SLLV		SRLV	SRAV
0010	JR	JALR		
0011	SYSCALL	BREAK		
0100	MFHI	MTHI	MFLO	MTLO
0101				
0110	MULT	MULTU	DIV	DIVU
0111	MAC			
1000	ADD	ADDU	SUB	SUBU
1001	AND	OR	XOR	NOR
1010			SLT	SLTU
1011				
1100				
1101				
1110				
1111				

表 6.4 *COPz*

IR[24..23]	00	01	10	11
IR[25,22,21,16]				
0,0,0,*	MF	MT		
0,0,1,0				
0,0,1,1				
0,1,0,0				
0,1,0,1				
0,1,1,0				
0,1,1,1				
1,*,*,*	*CO*			

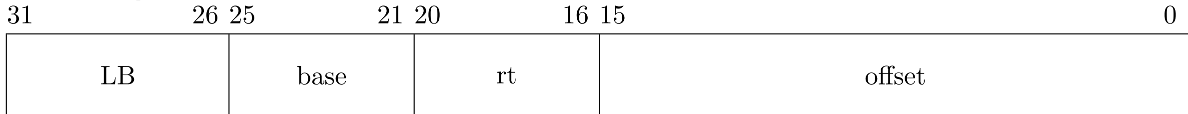
表 6.5　*COP0*

IR[1..0]	00	01	10	11
IR[4..2]				
000				
001				
010				
011				
100	RFE			
101				
110				
111				

6.3 ロード命令

6.3.1 LB 命令

バイトのロード



31 … 26	25 … 21	20 … 16	15 … 0
LB	base	rt	offset

意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. 有効アドレスが指定されたメモリ位置のバイトの内容は符号拡張されて汎用レジスタ rt にロードされる.

アセンブラ文法

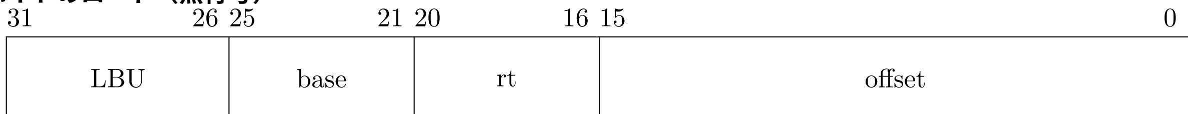
LB rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.2 LBU 命令

バイトのロード（無符号）



31 … 26	25 … 21	20 … 16	15 … 0
LBU	base	rt	offset

意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. 有効アドレスが指定されたメモリ位置のバイトの内容はゼロ拡張されて汎用レジスタ rt にロードされる.

アセンブラ文法

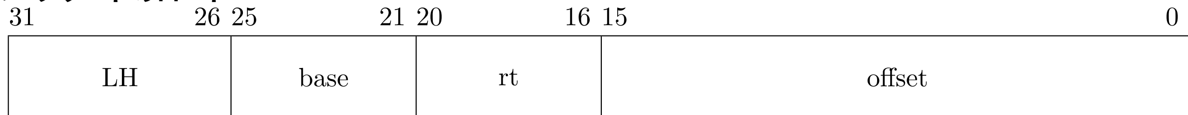
LBU rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.3　LH 命令

ハーフワードのロード



31　　　26	25　　　21	20　　　16	15　　　　　0
LH	base	rt	offset

意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. 有効アドレスが指定されたメモリ位置のハーフワードの内容は符号拡張されて汎用レジスタ rt にロードされる.

有効アドレスの最下位ビットが 0 でない場合には, アドレスエラー例外が発生する.

アセンブラ文法

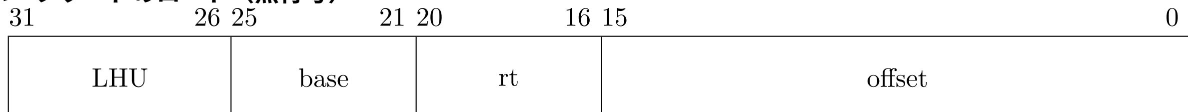
LH　rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.4　LHU 命令

ハーフワードのロード（無符号）



31　　　26	25　　　21	20　　　16	15　　　　　0
LHU	base	rt	offset

意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. 有効アドレスが指定されたメモリ位置のハーフワードの内容はゼロ拡張されて汎用レジスタ rt にロードされる.

有効アドレスの最下位ビットが 0 でない場合には, アドレスエラー例外が発生する.

アセンブラ文法

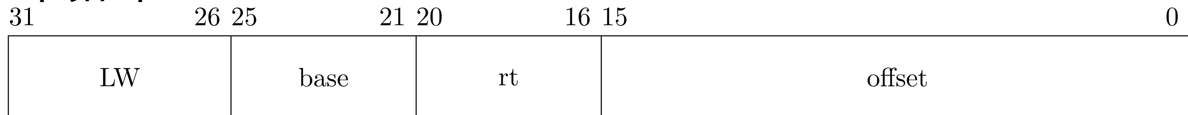
LHU　rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.5　LW 命令

ワードのロード



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. 有効アドレスが指定されたメモリ位置のワードの内容は汎用レジスタ rt にロードされる.

有効アドレスの下位 2 ビットが 0 でない場合には, アドレスエラー例外が発生する.

アセンブラ文法

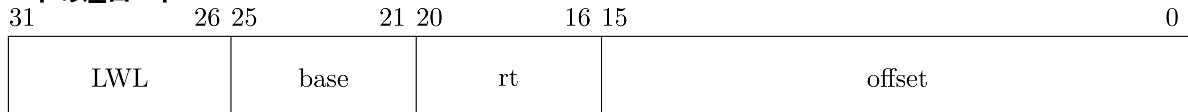
LW　rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.6　LWL 命令

ワードの左ロード



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. アドレス指定されたバイトがワードの最左端バイトになるように, アドレス指定されたワードを左にシフトする. メモリからのバイトをレジスタ rt の内容とマージし, 結果をレジスタ rt にロードする.

アセンブラ文法

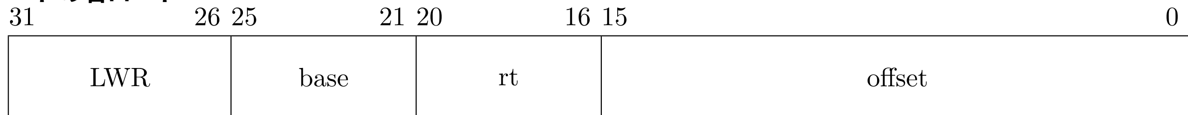
LWL　rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.3.7　LWR 命令

ワードの右ロード



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. アドレス指定されたバイトがワードの最右端バイトになるように, アドレス指定されたワードを右にシフトする. メモリからのバイトをレジスタ rt の内容とマージし, 結果をレジスタ rt にロードする.

アセンブラ文法

LWR　rt,offset(base)

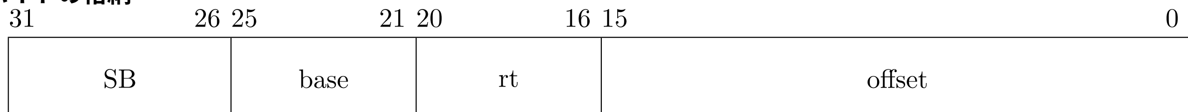
例外

- バスエラー例外
- アドレスエラー例外

6.4　ストア命令

6.4.1　SB 命令

バイトの格納



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. レジスタ rt の内容の最下位バイトは有効アドレスに格納される.

アセンブラ文法

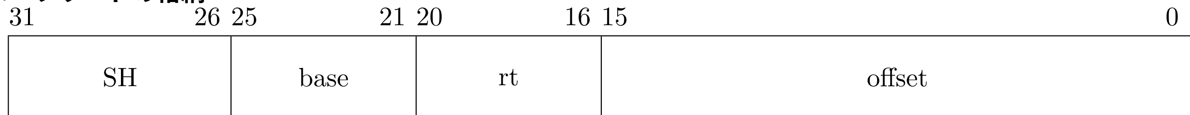
SB　rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.4.2 SH 命令

ハーフワードの格納



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. レジスタ rt の内容の最下位ハーフワードは有効アドレスに格納される.

有効アドレスの最下位ビットが 0 でない場合には, アドレスエラー例外が発生する.

アセンブラ文法

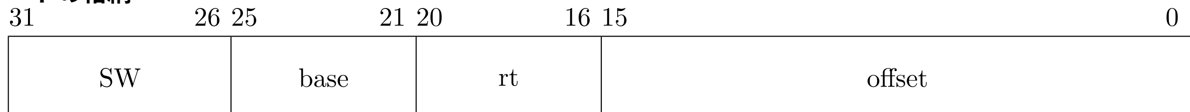
p SH rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.4.3 SW 命令

ワードの格納



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. レジスタ rt の内容が有効アドレスの指定するメモリ位置に格納される.

有効アドレスの下位 2 ビットが 0 でない場合には, アドレスエラー例外が発生する.

アセンブラ文法

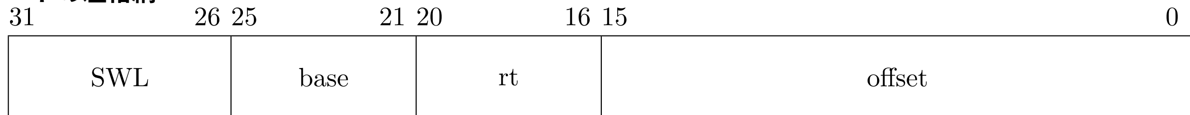
SW rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.4.4 SWL 命令

ワードの左格納



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. ワードの最左端バイトがアドレス指定されたバイトの位置になるように, レジスタ rt の内容を左にシフトする. 元のデータが記憶されている複数バイトを, アドレス指定されたバイト位置の対応する複数バイトにストアする.

アセンブラ文法

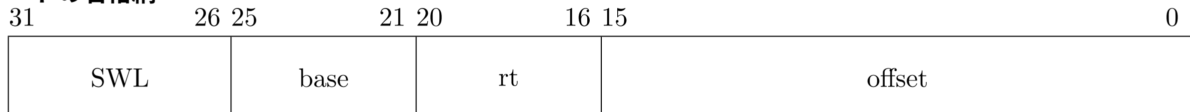
SWL rt,offset(base)

例外

- バスエラー例外
- アドレスエラー例外

6.4.5 SWL 命令

ワードの右格納



意味

16 ビット offset を符号拡張して汎用レジスタ base の内容に加算し, 32 ビット無符号有効アドレスを生成する. ワードの最右端バイトがアドレス指定されたバイトの位置になるように, レジスタ rt の内容を右にシフトする. 元のデータが記憶されている複数バイトを, アドレス指定されたバイト位置の対応する複数バイトにストアする.

アセンブラ文法

SWL rt,offset(base)

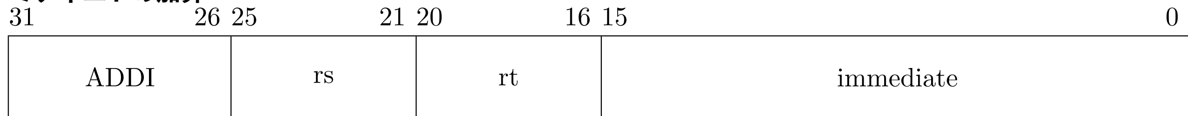
例外

- バスエラー例外
- アドレスエラー例外

6.5　ALU イミディエト命令

6.5.1　ADDI 命令

イミディエトの加算



意味

16 ビットイミディエトを符号拡張して汎用レジスタ rs の内容に加算し, 32 ビットの結果を生成する. この結果は汎用レジスタ rt 内に格納される. キャリーアウトの上位 2 ビットが異なる場合（2 の補数のオーバフロー）には, オーバフロー例外が発生する.

アセンブラ文法

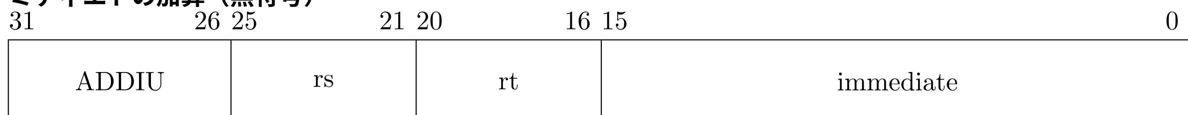
ADDI　rt,rs,immediate

例外

- オーバーフロー例外

6.5.2　ADDIU 命令

イミディエトの加算（無符号）



意味

16 ビットイミディエトを符号拡張して汎用レジスタ rs の内容に加算し, 32 ビットの結果を生成する. この結果は汎用レジスタ rt 内に格納される. どんな場合でもオーバフロー例外は発生しない.

アセンブラ文法

ADDIU　rt,rs,immediate

例外

- なし

6.5.3　SLTI 命令

イミディエト未満のセット

31　　26	25　　21	20　　16	15　　0
SLTI	rs	rt	immediate

意味

16 ビットイミディエトを符号拡張して汎用レジスタ rs の内容と比較する．符号付き 32 ビット整数として両方の数量を考慮しながら, rs が符号拡張付きイミディエト未満の場合には結果は 1 になり, そうでなければ結果は 0 になる．結果は汎用レジスタ rt に格納される．

どんな場合でも, オーバフロー例外は発生しない．比較はその比較演算の間に利用される減算がオーバフローをおこしても有効である．

アセンブラ文法

SLTI　rt,rs,immediate

例外

- なし

6.5.4　SLTIU 命令

イミディエト未満のセット（無符号）

31　　26	25　　21	20　　16	15　　0
SLTIU	rs	rt	immediate

意味

16 ビットイミディエトを符号拡張して汎用レジスタ rs の内容と比較する．無符号 32 ビット整数として両方の数を考慮しながら, rs が符号拡張付きイミディエト未満の場合には結果は 1 になり, そうでなければ結果は 0 になる．結果は汎用レジスタ rt に格納される．

どんな場合でも, オーバフロー例外は発生しない．比較はその比較演算の間に利用される減算がオーバフローをおこしても有効である．

アセンブラ文法

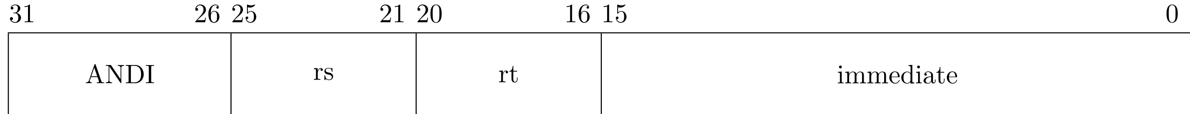
SLTIU　rt,rs,immediate

例外

- なし

6.5.5　ANDI 命令

イミディエトの論理積



意味

16 ビットのイミディエトをゼロ拡張して, 汎用レジスタ rs の内容とのビットごとの論理積をとり合成する．この結果は汎用レジスタ rt 内に格納される.

アセンブラ文法

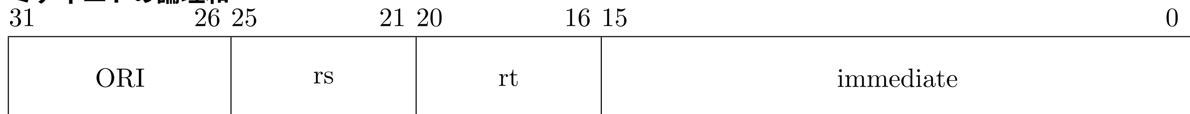
ANDI　rt,rs,immediate

例外

- なし

6.5.6　ORI 命令

イミディエトの論理和



意味

16 ビットのイミディエトをゼロ拡張して, 汎用レジスタ rs の内容とのビットごとの論理和をとり合成する．この結果は汎用レジスタ rt 内に格納される.

アセンブラ文法

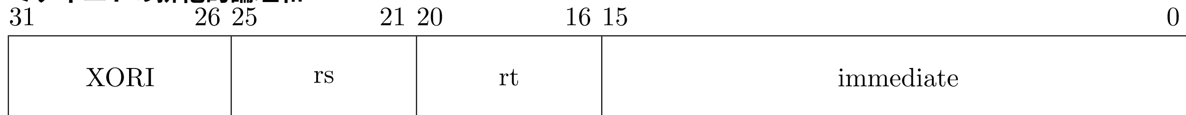
ORI　rt,rs,immediate

例外

- なし

6.5.7 XORI 命令

イミディエトの排他的論理和



意味

16 ビットのイミディエトをゼロ拡張して, 汎用レジスタ rs の内容とのビットごとの排他的論理和をとり合成する. この結果は汎用レジスタ rt 内に格納される.

アセンブラ文法

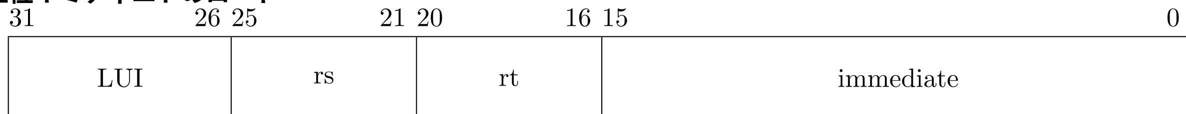
XORI rt,rs,immediate

例外

- なし

6.5.8 LUI 命令

上位イミディエトのロード



意味

16 ビットイミディエトを 16 ビット左詰めして 0 の 16 ビットと合成する. この結果を汎用レジスタ rt に格納する.

アセンブラ文法

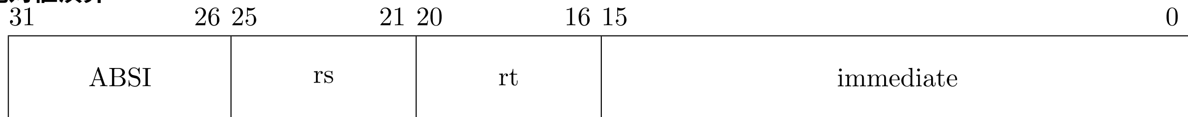
LUI rt,rs,immediate

例外

- なし

6.5.9　ABSI 命令

絶対値演算



意味

汎用レジスタ rs の値（2 の補数）を絶対値に変換して, 汎用レジスタ rt に代入する．immediate の値は実行結果に影響しない．

アセンブラ文法

ABSI　rt,rs,immediate

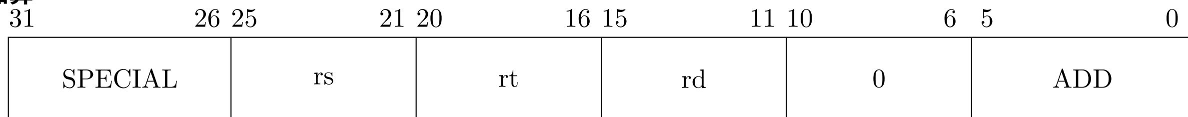
例外

- なし

6.6　3 オペランドレジスタタイプ命令

6.6.1　ADD 命令

加算



意味

汎用レジスタ rs の内容と汎用レジスタ rt の内容を加算して 32 ビットの結果を生成する．この結果は汎用レジスタ rd 内に格納される．キャリーアウトの上位 2 ビットが異なる場合（2 の補数のオーバフロー）には, オーバフロー例外が発生する．

アセンブラ文法

ADD　rd,rs,rt

例外

- オーバフロー例外

6.6.2　ADDU 命令

加算（無符号）

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	ADDU

意味

汎用レジスタ rs の内容と汎用レジスタ rt の内容を加算して 32 ビットの結果を生成する．この結果は汎用レジスタ rd 内に格納される．どんな場合でもオーバフロー例外は発生しない．

アセンブラ文法

ADDU　rd,rs,rt

例外

- なし

6.6.3　SUB 命令

減算

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	SUB

意味

汎用レジスタ rs の内容から汎用レジスタ rt の内容を減算して 32 ビットの結果を生成する．この結果は汎用レジスタ rd 内に格納される．キャリーアウトの上位 2 ビットが異なる場合（2 の補数のオーバフロー）には, オーバフロー例外が発生する．

アセンブラ文法

SUB　rd,rs,rt

例外

- オーバフロー例外

6.6.4 SUBU 命令

減算（無符号）

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	SUBU

意味

汎用レジスタ rs の内容から汎用レジスタ rt の内容を減算して 32 ビットの結果を生成する．この結果は汎用レジスタ rd 内に格納される．どんな場合でもオーバフロー例外は発生しない．

アセンブラ文法

SUBU rd,rs,rt

例外

- なし

6.6.5 SLT 命令

未満へのセット

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	SLT

意味

汎用レジスタ rt の内容を汎用レジスタ rs の内容と比較する．符号付き 32 ビット整数として両方の数量を考慮して, 汎用レジスタ rs の内容が汎用レジスタ rt の内容未満の場合は結果は 1 になる．そうでない場合には結果は 0 になる．この結果は汎用レジスタ rd に格納される．

どんな場合でも, オーバフロー例外は発生しない．比較はその比較演算の間に利用される減算がオーバフローをおこしても有効である．

アセンブラ文法

SLT rd,rs,rt

例外

- なし

6.6.6　SLTU 命令

未満へのセット（無符号）

31　26	25　21	20　16	15　11	10　6	5　0
SPECIAL	rs	rt	rd	0	SLTU

意味

汎用レジスタ rt の内容を汎用レジスタ rs の内容と比較する．無符号 32 ビット整数として両方の数量を考慮して, 汎用レジスタ rs の内容が汎用レジスタ rt の内容未満の場合は結果は 1 になる．そうでない場合には結果は 0 になる．この結果は汎用レジスタ rd に格納される．

どんな場合でも, オーバフロー例外は発生しない．比較はその比較演算の間に利用される減算がオーバフローをおこしても有効である．

アセンブラ文法

SLTU　rd,rs,rt

例外

- なし

6.6.7　AND 命令

論理積

31　26	25　21	20　16	15　11	10　6	5　0
SPECIAL	rs	rt	rd	0	AND

意味

汎用レジスタ rs の内容と汎用レジスタ rt の内容のビットごとの論理積をとり合成する．この結果は汎用レジスタ rd 内に格納される．

アセンブラ文法

AND　rd,rs,rt

例外

- なし

6.6.8 OR 命令

論理和

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	OR

意味

汎用レジスタ rs の内容と汎用レジスタ rt の内容のビットごとの論理和をとり合成する．この結果は汎用レジスタ rd 内に格納される．

アセンブラ文法

OR rd,rs,rt

例外

- なし

6.6.9 XOR 命令

排他的論理和

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	XOR

意味

汎用レジスタ rs の内容と汎用レジスタ rt の内容のビットごとの排他的論理和をとり合成する．この結果は汎用レジスタ rd 内に格納される．

アセンブラ文法

XOR rd,rs,rt

例外

- なし

6.6.10　NOR 命令

否定論理和

31　　26	25　　21	20　　16	15　　11	10　　6	5　　0
SPECIAL	rs	rt	rd	0	NOR

意味

汎用レジスタの rs の内容と汎用レジスタ rt の内容のビットごとの否定論理和をとり合成する．その結果は汎用レジスタ rd に格納される．

アセンブラ文法

NOR　rd,rs,rt

例外

- なし

6.7　シフト命令

6.7.1　SLL 命令

論理左シフト

31　　26	25　　21	20　　16	15　　11	10　　6	5　　0
SPECIAL	0	rt	rd	shamt	SLL

意味

0 を下位ビットに挿入して，汎用レジスタ rt の内容を shamt ビットだけ左にシフトする．32 ビットの結果はレジスタ rd に格納される．

アセンブラ文法

SLL　rd,rt,shamt

例外

- なし

6.7.2　SRL 命令

論理右シフト

31　　26	25　　21	20　　16	15　　11	10　　6	5　　0
SPECIAL	0	rt	rd	shamt	SRL

意味

0 を上位ビットに挿入して, 汎用レジスタ rt の内容を shamt ビットだけ右にシフトする. 32 ビットの結果はレジスタ rd に格納される.

アセンブラ文法

SRL　rd,shamt

例外

- なし

6.7.3　SRA 命令

算術右シフト

31　　26	25　　21	20　　16	15　　11	10　　6	5　　0
SPECIAL	0	rt	rd	shamt	SRA

意味

上位ビットを符号拡張して汎用レジスタ rt の内容を shamt ビットだけ右にシフトする. 32 ビットの結果はレジスタ rd に格納される.

アセンブラ文法

SRA　rd,rt,shamt

例外

- なし

6.7.4　SLLV 命令

論理変数左シフト

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL	rs	rt	rd	0	SLLV

意味

0 を下位ビットに挿入して, 汎用レジスタ rt の内容を汎用レジスタ rs の内容の下位 5 ビットが指定するビット数だけ左にシフトする. 32 ビットの結果はレジスタ rd に格納される.

アセンブラ文法

SLLV　rd,rt,rs

例外

- なし

6.7.5　SRLV 命令

論理変数右シフト

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL	rs	rt	rd	0	SRLV

意味

0 を上位ビットに挿入して, 汎用レジスタ rt の内容を汎用レジスタ rs の内容の下位 5 ビットが指定するビット数だけ右にシフトする. 32 ビットの結果はレジスタ rd に格納される.

アセンブラ文法

SRLV　rd,rt,rs

例外

- なし

6.7.6 SRAV 命令

算術変数右シフト

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	rt	rd	0	SRAV

意味

上位ビットを符号拡張して汎用レジスタ rt の内容を汎用レジスタ rs の内容の下位 5 ビットが指定するビット数だけ右にシフトする．32 ビットの結果はレジスタ rd に格納される．

アセンブラ文法

SRAV rd,rt,rs

例外

- なし

6.8 乗算／除算命令

6.8.1 MULT 命令

乗算

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	MULT

意味

汎用レジスタの rs の内容を汎用レジスタ rt の内容で乗算する．ただし, 両方のオペランドを 32 ビットの 2 の補数の値として扱う．どんな場合でも, オーバフロー例外は発生しない．

オペレーションが完了したときに, ダブルワードの結果の下位ワードを特殊レジスタ LO にロードし, ダブルワードの結果の上位ワードを特殊レジスタ HI にロードする．MFHI 命令と MFLO 命令はインターロックされていて, オペレーションが完了する前にそれを読みとろうとしても, オペレーション終了まではその命令を遅らせるようになっている．

乗算オペレーションは R3000 内の独立した別の実行ユニットによって実行される．乗算オペレーションが開始された後も, 他の命令の実行は平行して継続される．乗算ユニットと除算ユニットは, 命令が実行できなくなるキャッシュミスのミッシングやその他の遅延サイクルの間も実行を継続する．

アセンブラ文法

MULT rs,rt

例外

- なし

6.8.2　MULTU 命令

乗算（無符号）

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	MULTU

意味

汎用レジスタの rs の内容を汎用レジスタ rt の内容で乗算する．ただし, 両方のオペランドを 32 ビットの無符号値として扱う．どんな場合でも, オーバフロー例外は発生しない.

オペレーションが完了したときに, ダブルワードの結果の下位ワードを特殊レジスタ LO にロードし, ダブルワードの結果の上位ワードを特殊レジスタ HI にロードする．MFHI 命令と MFLO 命令はインターロックされていて, オペレーションが完了する前にそれを読みとろうとしても, オペレーション終了まではその命令を遅らせるようになっている.

乗算オペレーションは R3000 内の独立した別の実行ユニットによって実行される．乗算オペレーションが開始された後も, 他の命令の実行は平行して継続される．乗算ユニットと除算ユニットは, 命令が実行できなくなるキャッシュミスのミッシングやその他の遅延サイクルの間も実行を継続する.

アセンブラ文法

MULTU　rs,rt

例外

- なし

6.8.3　DIV 命令

除算

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	DIV

意味

汎用レジスタの rs の内容を汎用レジスタ rt の内容で除算する．ただし, 両方のオペランドを 32 ビットの 2 の補数の値として扱う．どんな場合でも, オーバフロー例外は発生しない.

オペレーションが完了したときに, ダブルワードの結果の商ワードを特殊レジスタ LO にロードし, ダブルワードの結果の剰余ワードを特殊レジスタ HI にロードする．MFHI 命令と MFLO 命令はインターロックされていて,

オペレーションが完了する前にそれを読みとろうとしても, オペレーション終了まではその命令を遅らせるようになっている.

除算オペレーションは R3000 内の独立した別の実行ユニットによって実行される. 除算オペレーションが開始された後も, 他の命令の実行は平行して継続される. 乗算ユニットと除算ユニットは, 命令が実行できなくなるキャッシュミスのミッシングやその他の遅延サイクルの間も実行を継続する.

アセンブラ文法

DIV rs,rt

例外

- なし

6.8.4 DIVU 命令

除算（無符号）

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	DIVU

意味

汎用レジスタの rs の内容を汎用レジスタ rt の内容で除算する. ただし, 両方のオペランドを 32 ビットの符号なしの値として扱う. どんな場合でも, オーバフロー例外は発生しない.

オペレーションが完了したときに, ダブルワードの結果の商ワードを特殊レジスタ LO にロードし, ダブルワードの結果の剰余ワードを特殊レジスタ HI にロードする. MFHI 命令と MFLO 命令はインターロックされていて, オペレーションが完了する前にそれを読みとろうとしても, オペレーション終了まではその命令を遅らせるようになっている.

除算オペレーションは R3000 内の独立した別の実行ユニットによって実行される. 除算オペレーションが開始された後も, 他の命令の実行は平行して継続される. 乗算ユニットと除算ユニットは, 命令が実行できなくなるキャッシュミスのミッシングやその他の遅延サイクルの間も実行を継続する.

アセンブラ文法

DIVU rs,rt

例外

- なし

6.8.5　MFHI 命令

HI からの転送

31–26	25–16	15–11	10–6	5–0
SPECIAL	0	rd	0	MFHI

意味

特殊レジスタ HI の内容を汎用レジスタ rd にロードする．

アセンブラ文法

MFHI　rd

例外

- なし

6.8.6　MFLO 命令

LO からの転送

31–26	25–16	15–11	10–6	5–0
SPECIAL	0	rd	0	MFLO

意味

特殊レジスタ LO の内容を汎用レジスタ rd にロードする．

アセンブラ文法

MFLO　rd

例外

- なし

6.8.7　MTHI 命令

HI への転送

31–26	25–21	20–6	5–0
SPECIAL	rs	0	MTHI

意味

汎用レジスタ rs の内容を特殊レジスタ HI にロードする.

MTHI オペレーションが MULT, MULTU, DIV あるいは DIVU 命令の後にしかも MFLO, MFHI, MTLO あるいは MTHI 命令のどれかに先立って実行される場合には, 特殊レジスタ LO の内容は定義されない.

アセンブラ文法

MTHI rs

例外

- なし

6.8.8 MTLO 命令

LO への転送

31–26	25–21	20–6	5–0
SPECIAL	rs	0	MTLO

意味

汎用レジスタ rs の内容を特殊レジスタ LO にロードする.

MTHI オペレーションが MULT, MULTU, DIV あるいは DIVU 命令の後にしかも MFLO, MFHI, MTLO あるいは MTHI 命令のどれかに先立って実行される場合には, 特殊レジスタ HI の内容は定義されない.

アセンブラ文法

MTLO rs

例外

- なし

6.8.9 MAC 命令

積和演算

31–26	25–21	20–16	15–6	5–0
SPECIAL	rs	rt	0	MAC

意味

汎用レジスタの rs の内容を汎用レジスタ rt の内容で積和演算を行なう. ただし, 両方のオペランドを 32 ビットの 2 の補数の値として扱う. どんな場合でも, オーバフロー例外は発生しない.

オペレーションが完了したときに, ダブルワードの結果の下位ワードを特殊レジスタ LO にロードし, ダブルワードの結果の上位ワードを特殊レジスタ HI にロードする．MFHI 命令と MFLO 命令はインターロックされていて, オペレーションが完了する前にそれを読みとろうとしても, オペレーション終了まではその命令を遅らせるようになっている．

積和演算オペレーションは R3000 内の独立した別の実行ユニットによって実行される．積和演算オペレーションが開始された後も, 他の命令の実行は平行して継続される．乗算ユニットと除算ユニットは, 命令が実行できなくなるキャッシュミスのミッシングやその他の遅延サイクルの間も実行を継続する．

アセンブラ文法

MAC　rs,rt

例外

- なし

6.9　ジャンプ命令

6.9.1　J 命令

ジャンプ

意味

26 ビットターゲット・アドレスを 2 ビット左詰めして現在のプログラム・カウンタの上位 4 ビットと合成する．プログラムは計算されたアドレスに 1 命令の遅れで無条件に分岐する．

アセンブラ文法

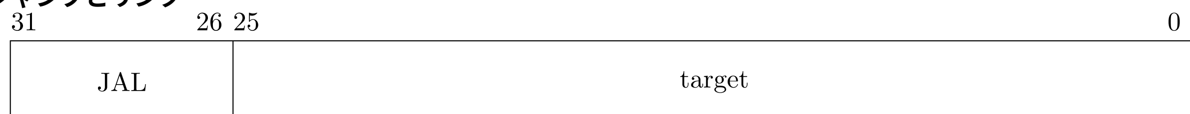
J　target

例外

- なし

6.9.2　JAL 命令

ジャンプとリンク



意味

26 ビットターゲット・アドレスを 2 ビット左詰めして現在のプログラム・カウンタの上位 4 ビットと合成し, さらにプログラムを計算されたアドレスに 1 命令の遅れで無条件に分岐する. 遅延スロットの後の命令のアドレスはリンクレジスタ r31 に格納される.

アセンブラ文法

JAL target

例外

- なし

6.9.3 JR 命令

ジャンプレジスタ

31–26	25–21	20–6	5–0
SPECIAL	rs	0	JR

意味

プログラムは 1 命令遅れで汎用レジスタ rs にあるアドレスに無条件で分岐する.

アセンブラ文法

JR rs

例外

- アドレスエラー例外

6.9.4 JALR 命令

ジャンプのリンクのレジスタ

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL	rs	0	rd	0	JALR

意味

プログラムは 1 命令遅れで汎用レジスタ rs にあるアドレスに無条件で分岐する. 遅延スロットの後の命令のアドレスは汎用レジスタ rd 内に格納される. アセンブリ言語命令で省略してある場合, rd のデフォルト値は 31 である.

アセンブラ文法

JALR　rs
JALR　rd,rs

例外

- アドレスエラー例外

6.10　分岐命令

6.10.1　BEQ 命令

等号での分岐

31 … 26	25 … 21	20 … 16	15 … 0
BEQ	rs	rt	offset

意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する．汎用レジスタ rs の内容と汎用レジスタ rt の内容を比較する．二つのレジスタが等しい場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する．

アセンブラ文法

BEQ　rs,rt.offset

例外

- なし

6.10.2　BNE 命令

不等号での分岐

31 … 26	25 … 21	20 … 16	15 … 0
BNE	rs	rt	offset

意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する．汎用レジスタ rs の内容と汎用レジスタ rt の内容を比較する．二つのレジスタが等しくない場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する．

アセンブラ文法

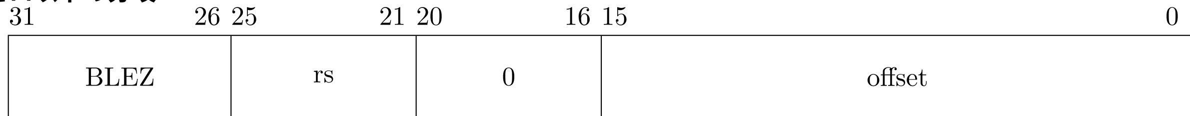
BNE rs,rt,offset

例外

- なし

6.10.3 BLEZ 命令

ゼロ以下の分岐



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する. 汎用レジスタ rs の内容を 0 と比較する. 汎用レジスタ rs の内容の符号ビットがセットされているかまたはその内容が 0 に等しい場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する.

アセンブラ文法

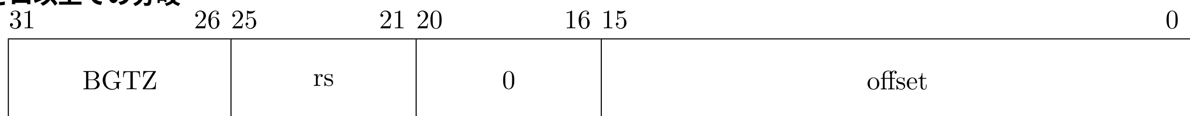
BLEZ rs,offset

例外

- なし

6.10.4 BGTZ 命令

ゼロ以上での分岐



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する. 汎用レジスタ rs の内容を 0 と比較する. 汎用レジスタ rs の内容の符号ビットがクリアされていてかつその内容が 0 でない場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する.

アセンブラ文法

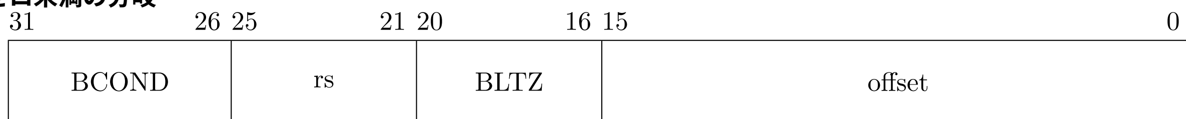
BGTZ　rs,offset

例外

- なし

6.10.5　BLTZ 命令

ゼロ未満の分岐



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する. 汎用レジスタ rs の内容を 0 と比較する. 汎用レジスタ rs の内容の符号ビットがセットされている場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する.

アセンブラ文法

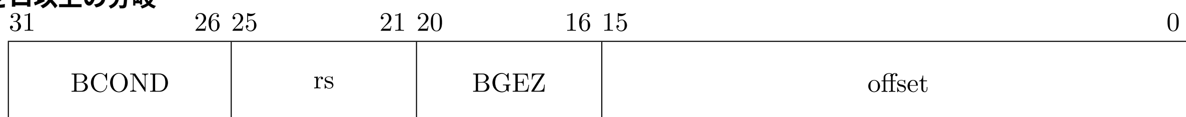
BLTZ　rs,offset

例外

- なし

6.10.6　BGEZ 命令

ゼロ以上の分岐



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する. 汎用レジスタ rs の内容の符号ビットがクリアされている場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する.

アセンブラ文法

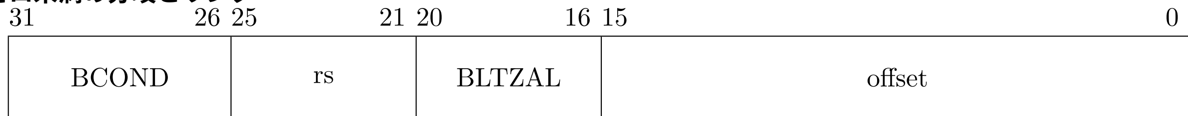
BGEZ　rs,offset

例外

- なし

6.10.7　BLTZAL 命令

ゼロ未満の分岐とリンク



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する．遅延スロットの後の命令のアドレスは無条件でリンクレジスタ r31 に格納される．汎用レジスタ rs の内容の符号ビットがセットされている場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する．

アセンブラ文法

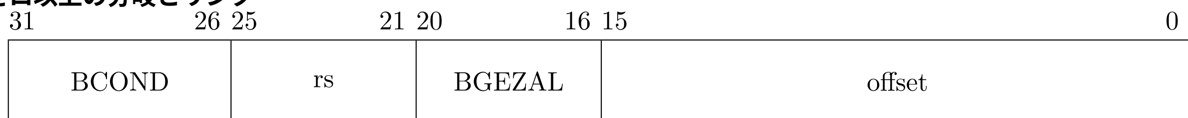
BLTZAL　rs,offset

例外

- なし

6.10.8　BGEZAL 命令

ゼロ以上の分岐とリンク



意味

分岐ターゲット・アドレスを遅延スロット内の命令のアドレスと 16 ビットのオフセットの和から計算し, 2 ビット左詰めして 32 ビットに符号拡張する．遅延スロットの後の命令のアドレスは無条件でリンクレジスタ r31 に格納される．汎用レジスタ rs の内容の符号ビットがクリアされている場合には, プログラムは 1 命令の遅れでターゲット・アドレスに分岐する．

アセンブラ文法

BGEZAL　rs,offset

例外

- なし

6.11　特殊命令

6.11.1　SYSCALL 命令

システムコール

31　　　26	25　　　　　　　　　　6	5　　　0
SPECIAL	0	SYSCALL

意味

システムコール・トラップが発生して, 制御権を即座に無条件で例外ハンドラに移動する.

アセンブラ文法

SYSCALL

例外

- システムコール例外

6.11.2　BREAK 命令

ブレーク

31　　　26	25　　　　　　　　　　6	5　　　0
SPECIAL	code	BREAK

意味

ブレークポイント・トラップが発生して, 即座に無条件で制御権を例外ハンドラに移す.

コードフィールドはソフトウェア・パラメータとして使用することが可能であるが, 命令ハンドラがこれを取り出すためには命令のあるメモリワードの内容をロードしなければならない.

アセンブラ文法

BREAK

例外

- ブレークポイント・トラップ `

6.12 コプロセッサ命令

6.12.1 MTC0 命令

システム制御コプロセッサへの転送

31–26	25–21	20–16	15–11	10–0
COP0	MT	rt	rd	0

意味

汎用レジスタ rt の内容をシステム制御コプロセッサ (CP0) のコプロセッサレジスタ rd にロードする.

アセンブラ文法

MTC0 rt,rd

例外

- コプロセッサ不使用例外

6.12.2 MFC0 命令

システム制御コプロセッサからの転送

31–26	25–21	20–16	15–11	10–0
COP0	MF	rt	rd	0

意味

システムの制御コプロセッサ (CP0) のコプロセッサレジスタ rd の内容を汎用レジスタ rt にロードする.

アセンブラ文法

MFC0 rt,rd

例外

- コプロセッサ不使用例外

6.12.3 RFE 命令

例外からの復元

31–26	25	24–5	4–0
COP0	CO	0	RFE

意味

状態レジスタ (SR) の以前の割り込みマスクとカーネルユーザモード・ビット（IEp と KUp）を, 対応する現在の状態ビット（IEc と KUc）内に復元して, 旧状態ビット（IEo と KUo）を, 対応する状態ビット（IEp と KUp）内に復元する. 旧状態ビットは変更されないままである.

普通, RFE 命令は PC を復元させるために JR（無条件分岐レジスタ）命令の遅延スロットの後に続く.

アセンブラ文法

RFE

例外

- コプロセッサ不使用例外

第 7 章

サンプルプログラムの実行

RUECHIP2 CPU を搭載する FPGA 基板: Cmod A7 モジュールは, KR-CHIP 教育用ボード上のキーボードとディスプレイに接続しているが, PC と Cmod A7 モジュールを接続し通信することも可能である．この章では, C で書かれたプログラムを KR-CHIP 用ソフトウェア環境 (kr-chip) でコンパイルし, RUECHIP2 で実行する方法を, マーチングプログラムを例に取り具体的にを示す.

7.1　マーチングテストプログラム

この章の終わりに示す図 7.4 のプログラムが例にとるマーチングテストを行うプログラムである．マーチングテストは正しくメモリへの読み書きが行えること確認するテストである．下のプログラムでは, メモリに書き込んだ値と, 読み出した値が一致していない場合に実行が止まるようになっており, 一致している場合は無限ループするようになっている.

テストの結果は A0001F00 番地に書き込まれる．書き込み時 4, 読み出し時 5, テストするメモリ帯域すべてに対して, 書き込みと読み出しの値が一致していた場合は 3 を書き込み左にシフトした値になる．よって 2 桁目以降に 3 が増えていくことを確認できればマーチングテストの成功を確認できる.

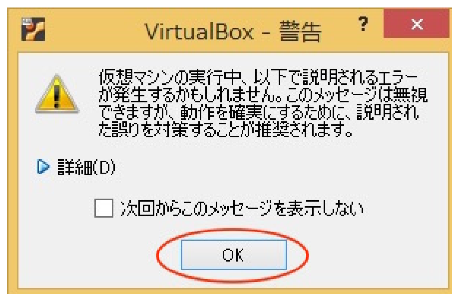
7.2　KR-CHIP 用ソフトウェア環境 krchip の起動

サンプルプログラムのコンパイルは KR-CHIP 用ソフトウェア環境 krchip で行う．以下で krchip 環境の起動の方法を示す．krchip のインストール方法は KRCHIP ボードのリファレンスマニュアルで解説している.

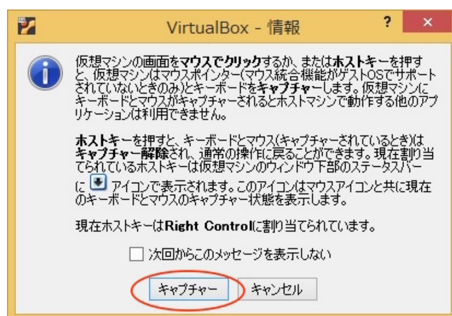
1. Virtual Box ソフトウェアを起動する.

2.「Oracle VM VirtualBox マネージャー」の「kr-chip」が選択されている状態で，「起動 (T)」ボタンをクリックする．

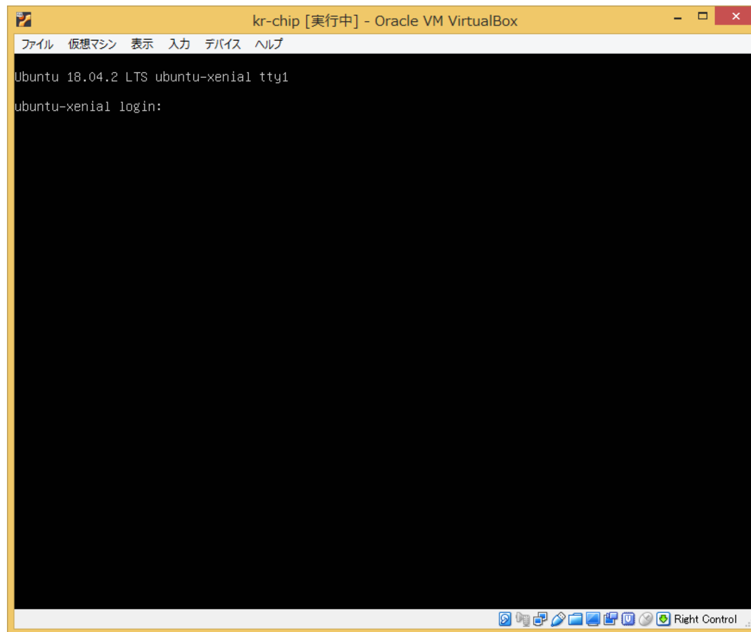
3.「VirtualBox - 警告」ウィンドウが表示されたら「 OK 」ボタンをクリックする．



4.「 VirtualBox - 情報」ウィンドウが表示されたら「キャプチャー」ボタンをクリックする．

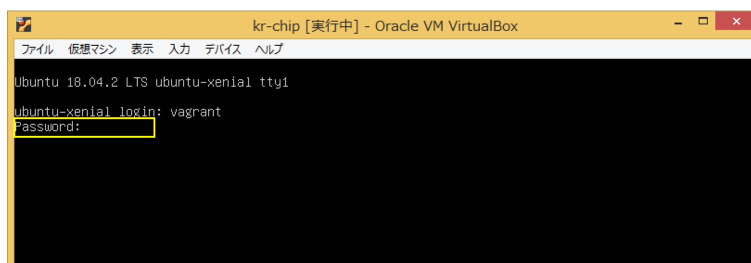


以下の「kr-chip [実行中] 」ウィンドウが表示されたら，KR-CHIP 用の Ubuntu Linux 環境の起動に成功している．



5. ログイン名に「vagrant」を入力してリターンする．

6. パスワードも「vagrant」と入力しリターンする．



7. ログインに成功すると，以下のメッセージが表示される．

```
kr-chip [実行中] - Oracle VM VirtualBox
ファイル  仮想マシン  表示  入力  デバイス  ヘルプ

Ubuntu 18.04.2 LTS ubuntu-xenial tty1

ubuntu-xenial login: vagrant
Password:
Last login: Tue Jun  4 01:35:14 UTC 2019 on tty1
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-50-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue Jun  4 01:38:44 UTC 2019

  System load:  0.18              Processes:             96
  Usage of /:   63.1% of 9.63GB   Users logged in:       0
  Memory usage: 16%               IP address for enp0s3: 10.0.2.15
  Swap usage:   0%

 * Ubuntu's Kubernetes 1.14 distributions can bypass Docker and use containerd
   directly, see https://bit.ly/ubuntu-containerd or try it now with

     snap install microk8s --classic

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

57 packages can be updated.
0 updates are security updates.


vagrant@ubuntu-xenial:~$
                                                    Right Control
```

7.3　サンプルプログラムのコンパイルとボードへの送信

7.3.1　PC と RUECHIP2 CPU 間の通信

サンプルプログラムの krchip 環境でのコンパイル, またボードと通信し実行ファイルをボードに送る方法を示す.

PC と Cmod A7 はシリアル通信にて通信することができる. この通信は, 常に PC からコマンドを送信し, FPGA はそのコマンドに対して応答する. 送受信するコマンドは ASCII 文字列でのみ構成される. FPGA ボードはコマンド以外の文字列を受信すると, 受信した文字列を送り返す.

通信方法には, RS232C コネクタを利用する方法と microUSB を利用する方法があるが RUECHIP2 との通信には microUSB を用いて通信を行う. これらのモードは BTN1 により切り替え, LD0 により判別できる. BTN1 を押すと, LD0 が青→緑→赤→青…と順に切り替わる (図 7.1, 7.2, 7.3). LD0 が緑色に点灯しているときは RS232C コネクタと接続し, 赤色に点灯しているときは Cmod A7 上の microUSB ポートを利用し PC と接続する. BTN1 の隣のボタン BTN0 は FPGA のリセットボタンのため注意が必要である.

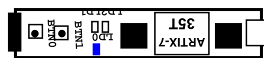


図 7.1　KR-CHIP 教育用ボード接続時

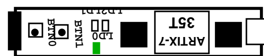


図 7.2　RS232C コネクタ接続時

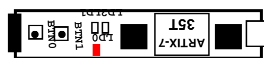


図 7.3　microUSB 接続時

7.3.2　サンプルプログラムのコンパイルとボードへの送信

microUSB ポートを用いた接続では, krchip 環境と Cmod A7 モジュールの USB コネクタを接続する. よって PC の USB ポートを krchip 環境で認識させる必要がある. 以下では krchip 環境と Cmod A7 モジュール間を接続, サンプルプログラムをコンパイルし, Cmod A7 モジュールに送信する手順を示す.

1. KR-CHIP 教育用ボードの電源コネクタに 5V AC アダプタを接続し, 電源スイッチ：SW_POWER を ON にする.
2. Cmod A7 モジュール上の LED：LD0 が青色で点灯していることを確認する.
3. PC と KR-CHIP 教育用ボード上の Cmod A7 を microUSB type-B ケーブルで接続する.

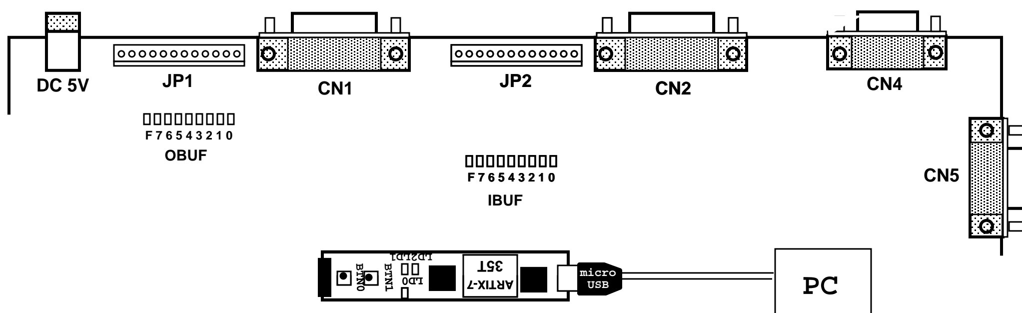
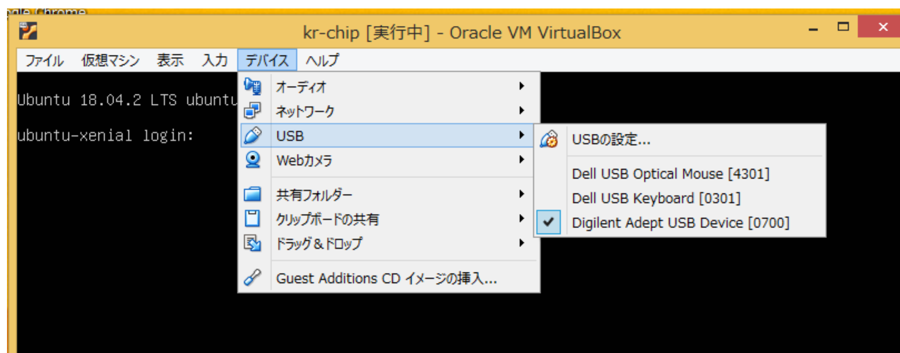
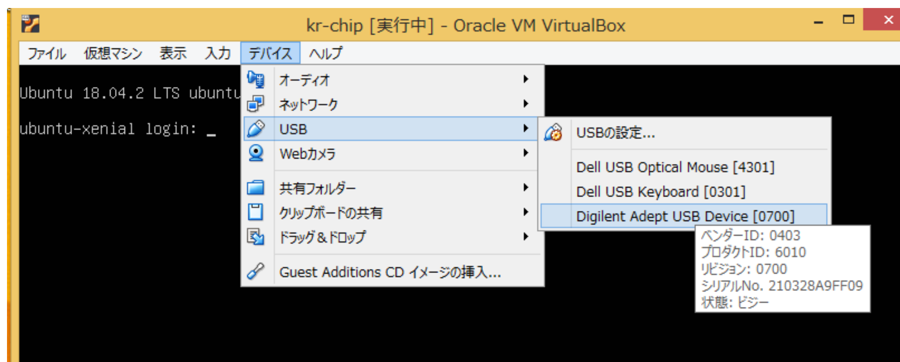


図 7.4　microUSB により PC と FPGA を接続

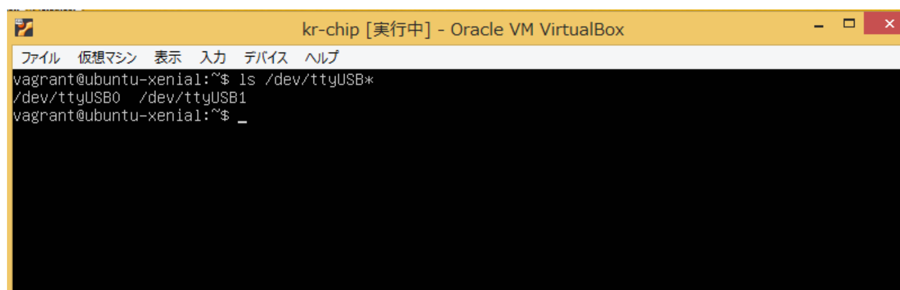
4. Cmod A7 モジュール上の BTN1 ボタンを 2 度押し, PC との接続モードに変更する (LD0 は青→緑→赤と変化).
5. メニューバーのデバイスから USB を選択.



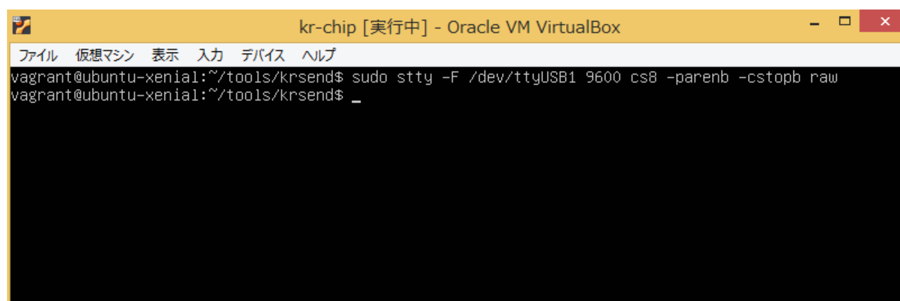
6.「Digilent Adept USB Device」を選択し USB デバイスを選択する



7. `$ ls /dev/` コマンドを使用し，RUECHIP2 CPU が何という名前のデバイスとして認識されているか確認する (例: /dev/ttyUSB1/).



8. `$sudo stty -F /dev/ttyUSB1 9600 cs8 -parenb -cstopb raw` コマンドを実行する．



9. `$sudo chmod o+rw /dev/ttyUSB1` コマンドを実行する．

10. サンプルプログラム marching_rue2.c があるディレクトリで`$perl marching_rue2.c > /dev/ttyUSB1` コマンドを実行し marching_rue2.c をコンパイルし，Cmod A7 モジュールに送信する．`krmake.pl` に

PATH が通っていない場合は, 相対パス, 絶対パスに変更する.

11. 書き込みが終了したことを確認する. (書き込み中は microUSB の上にある LED が点灯)
12. Cmod A7 上の BTN1 を 1 度押し, PC との接続モードに変更する (赤→青).
13. Reset ボタンを押し, KR-CHIP 教育用ボード上のディプレイの表示が図 7.5 のようになっていることを確認する.

```
HALT PL   PC:80000000
          PC:80000000
ADR 80000000
MEM 00000000
```

図 7.5　起動完了後に表示される画面

7.4　サンプルプログラムの実行

送信したプログラムの実行を行う.

マーチングテストはテストするメモリ帯域の全ての番地に書き込みと読み出しを行い, 値が一致している場合, メモリアドレスの A0001F00 番地に 3 が書き込まれ左シフトし次のループに入るようになっている.

よって以下の手順で確認する.

KR-CHIP 教育用ボードの 16 進キーボードから A0001F00 を入力し, ADR キーを押す. ディスプレイの三行目に値が反映される.

```
HALT PL   PC:80000000
          PC:80000000
ADR A0001F00
MEM 00000000
```

動作クロック周波数を切替えスイッチ CLKFRQ の目盛りを 1 に合わせる.

SS キーを押してプログラムを実行する.

SS

四行目に表示されるメモリの値の 2 桁目以降に 3 が書き込まれていく様が確認できたら実行プログラムは正しく動作していると言える.

```
RUN PL    PC:80000000
          PC:80000000
ADR A0001F00
MEM 00003334
```

```
#define S11     1
#define S12     5
#define S21     2
#define S22     6
#define S31     7
#define S32     4
#define S41     7
#define S42     3

/* F, G, H and I are basic MD5 functions. */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits. */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (8-(n))))

#define FF(a, b, c, d, x, s) { \
        (a) += F ((b), (c), (d)) + (x); \
        (a) = ROTATE_LEFT ((a), (s)); \
        (a) += (b); \
  }
#define GG(a, b, c, d, x, s) { \
        (a) += G ((b), (c), (d)) + (x); \
        (a) = ROTATE_LEFT ((a), (s)); \
        (a) += (b); \
  }
#define HH(a, b, c, d, x, s) { \
        (a) += H ((b), (c), (d)) + (x); \
        (a) = ROTATE_LEFT ((a), (s)); \
        (a) += (b); \
  }
#define II(a, b, c, d, x, s) { \
        (a) += I ((b), (c), (d)) + (x); \
        (a) = ROTATE_LEFT ((a), (s)); \
        (a) += (b); \
  }

#define SEED(x, y) ((x) ^ (~y ^ 0xD5AB))
#define SETRESULT(x) { *result = ((*result & 0xFFFFFFF0) | (x & 0x0000000F));}
void march_write( unsigned int* start, unsigned int* end, unsigned int seed );
int march_read(   unsigned int* start, unsigned int* end, unsigned int seed );
unsigned int simple_hash(unsigned int);
volatile unsigned int* result  = ( unsigned int* ) 0xA0001F00;
```

```
int main (void) {
     unsigned int* start_data  = ( unsigned int* ) 0xA0000a6C;
     unsigned int* end_data    = ( unsigned int* ) 0xA0001EFC;
     unsigned int* start_instr = ( unsigned int* ) 0x80000a6C;
     unsigned int* end_instr   = ( unsigned int* ) 0x80001EFC;

    unsigned int counter= 1;
    int flag= 0;
    SETRESULT(1);
     while ( 1 ) {
            march_write( start_data, end_data, counter );
            flag = march_read( start_data, end_data, counter );
            if ( flag ) { SETRESULT(2);; break; }

            march_write( start_instr, end_instr, counter );
            flag = march_read( start_instr, end_instr, counter );
            if ( flag ) { SETRESULT(2);; break; }

            counter++;
            SETRESULT(3);
            *result = *result << 4;
    }
    return 0;
}

void march_write( unsigned int* start, unsigned int* end, unsigned int seed )
   {
        unsigned int input;
        unsigned int value;
        unsigned int* addr;
        SETRESULT(4);
        for( addr = start ; addr < end ; addr += 1 ) {
            input = SEED((unsigned int)addr, seed);
            value = simple_hash(input);
            *addr = value;
        }
}

int march_read( unsigned int* start, unsigned int* end, unsigned int seed ) {
        unsigned int input;
        unsigned int value;
        unsigned int* addr;
        int flag = 0;

        SETRESULT(5);
        for(addr = start ; addr < end ; addr += 1){
            input = SEED((unsigned int)addr, seed);
            value = simple_hash(input);
            if( *addr != value ) {
                flag = 1;
            }
        }

 return flag;
}
```

```
unsigned int simple_hash(unsigned int input) {
        unsigned char div[4];
        unsigned char a,b,c,d;
        unsigned char x[]= {0x0F,0x11,0x03,0x50};
        unsigned int data=0;
        int i=0;
        for(i=0;i<4;i++)
        {       div[i] = (input >> 8*i) & 0xFF; }
        a = div[0]; b = div[1]; c = div[2]; d = div[3];

        /* Round 1 */
        FF(a, b, c, d, x[ 0], S11);     /*  1 */
        FF(d, a, b, c, x[ 1], S12);     /*  2 */
        FF(c, d, a, b, x[ 2], S11);     /*  3 */
        FF(b, c, d, a, x[ 3], S12);     /*  4 */

        /* Round 2 */
        GG(a, b, c, d, x[ 3], S21);     /*  5 */
        GG(d, a, b, c, x[ 1], S22);     /*  6 */
        GG(c, d, a, b, x[ 2], S21);     /*  7 */
        GG(b, c, d, a, x[ 0], S22);     /*  8 */

        /* Round 3 */
        HH(a, b, c, d, x[ 2], S31);     /*  9 */
        HH(d, a, b, c, x[ 3], S32);     /* 10 */
        HH(c, d, a, b, x[ 0], S31);     /* 11 */
        HH(b, c, d, a, x[ 1], S32);     /* 12 */

        /* Round 4 */
        II(a, b, c, d, x[ 1], S41);     /* 13 */
        II(d, a, b, c, x[ 0], S42);     /* 14 */
        II(c, d, a, b, x[ 3], S41);     /* 15 */
        II(b, c, d, a, x[ 2], S42);     /* 16 */

        div[0] += a;
        div[1] += b;
        div[2] += c;
        div[3] += d;

        for(i=0;i<4;i++)
                {       data += (div[i] << 8*i); }

        return data;
}
```

図 7.6 マーチングテストプログラム

参考文献

[1] Gerry Kane, mips RISC アーキテクチャー R2000/R3000 ー，共立出版, 1992.