

## 付録 C

# 教育用プロセッサ KAPPA3-RV32I の仕様 Ver. 0.1

### C.1 概要

この文書は教育用プロセッサ KAPPA3-RV32I(Kyushu Advanced Program for Processor Architecture Ver. 3 -RV32I) の仕様について記したものである。ここで述べるプロセッサは、教育用として最低限必要な機能を持ち、かつ学生が理解しやすい構造を採ることを目標とする。ベースとなるアーキテクチャとして設計を容易にし、かつパイプラインによる実装への発展を理解させるため、汎用レジスタを持ったロードストア型アーキテクチャとしてオープンなアーキテクチャ RISC-V(RV32I) を採用している。RISC-V では語長や命令セットの種類によっていくつかのバリエーションを持つが、ここでは 32 ビットの整数演算命令のみをサポートする RV32I を用いる。他の一般的な RISC(Reduced Instruction Set Computer) プロセッサと同様に、全ての算術論理演算はレジスタ・レジスタ間もしくはレジスタ・即値の間で行われる。以下に主な仕様を述べる。

語長:     • 32 ビットを 1 語とする。

- バイトアドレッシング — 1 バイト (8 ビット) 単位でメモリ番地がついている。つまり、1 語は 4 バイトからなる。
- リトルエンディアン — 4 バイトの下位のバイトが先 (下位アドレス) にくる形式のこと。つまり 32'h12345678 (Verilog-HDL の表記) という 32 ビットの値を 0 番地から始まる 4 バイト (つまり 0 番地から 3 番地) に書き込むと 0 番地の値は下位 8 ビットの 8'h78 となり 1 番地の値は次の 8 ビットの 8'h56、最後の 3 番地は 8'h12 となる。
- 全てのアクセスは整列化されていなければならない。4 バイトのワード (語) に対するアクセスは必ず 4 の倍数のアドレスに対して行われなければならない。2 バイトのハーフワード (半語) に対するアクセスは必ず 2 の倍数のアドレスに対して行われなければならない。

内部メモリ:   FPGA チップ内に 64K バイトの内部メモリを持つ。アドレス空間は 0x10000000–0x1000FFFF である。ただしあらかじめ設計された記述を与えるので学生は設計する必要はない。

タイマー:   プロセッサとは独立に動作する 64 ビットのタイマーカウンタを持つ。タイマーのカウント値が指定された値に一致した時に割り込み要求が発生する。

割り込み:   単純な割り込み機能を持つ。割り込み要求に応じて通常のプログラムの実行を中断して別のプログラムの実行を行う。

KAPPA3-RV32I は RISC-V の RV32I の仕様に準拠したものであるが、そのままでは学生実験の題材

として複雑すぎるので、KAPPA3-RV32I から割り込み関係の機能を削除して簡単化したプロセッサである KAPPA3-LIGHT を用意した。本実験ではまずこの KAPPA3-LIGHT の設計および動作確認を行う。引き続き、KAPPA3-RV32I を用いたソフトウェア開発の演習を行う。以降では KAPPA3-RV32I の仕様について述べるが割り込みと CSR(後述) に関する記述以外は KAPPA3-LIGHT も同様である。

## C.2 命令フォーマット

全ての命令は 32 ビット (1 語) 固定長であり、次の 6 種類の形式を持つ。なお、命令の形式に関わらず最下位 7 ビットはオペコード (opcode) と呼ばれるフィールドで命令の種類を表す。

表 C.1 命令フォーマットの種類

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

R-type: 主にレジスタ・レジスタ演算命令に使用される。 $R_d$  はデスティネーションレジスタと呼ばれる。演算結果もしくはロードした値を格納するレジスタを指定する。 $R_{s1}, R_{s2}$  はソースレジスタと呼ばれる。演算命令の場合にはその名の通りソースとなる値を格納しているレジスタを指定する。残りの 3 ビットの funct3 フィールドと 7 ビットの funct7 フィールドで命令の詳細な機能を指定する。

I-type: レジスタ・即値演算、レジスタ間接ジャンプ (JALR)、およびロード命令に使用される。 $R_d$  はデスティネーションレジスタと呼ばれる。通常は演算結果を格納するレジスタを指定するが、ジャンプ命令の場合には戻り値 (現在の PC の値) を格納するレジスタの指定に用いる。 $R_{s1}$  はソースレジスタと呼ばれる。演算に用いられるレジスタを指定する。ジャンプ命令およびロード命令の場合にはアドレスを表すベースレジスタとして用いられる。上位 12 ビットの即値は演算命令の場合は演算に用いる値として用いられる。ロード命令およびジャンプ命令の場合にはアドレスのオフセットとして用いられる。

S-type: ストア命令で用いられる。 $R_{s1}$  レジスタはアドレス計算のベースレジスタとして用いられる。 $R_{s2}$  レジスタはストアする値を表している。ロード命令と同じく即値は 12 ビットで表されるが 2 つのフィールドにまたがっている。これは全ての命令のなかで  $R_{s1}, R_{s2}, R_d$  レジスタの指定フィールドを同一にするための工夫である。

B-type: 分岐命令で用いられる。S-type に似ているが、即値のエンコーディングが異なっている。これは分岐先アドレスが偶数であることから最下位ビット (0 ビット) が不要であることから即値を 1 ビットずらして用いるための工夫である。ただし、即値の 12 ビットと 11 ビット以外は S-type と同一のエンコーディングになっている。

U-type: 上位ロード命令 (LUI, AUIPC) で用いられる。 $R_d$  フィールドとオペコード以外の 20 ビットを即値として用いている。ただし、31 ビット目から 12 ビット目までの上位 20 ビットを表していることに注意。

J-type: ジャンプ命令で用いられる。こちらも U-type と同様に  $R_d$  フィールドとオペコード以外の 20 ビットを即値として用いているがエンコーディングが異なる。こちらは即値の 10 ビット目から 1 ビット目

までが I-type と同一になるように工夫されている．なお，ジャンプ先のアドレスは偶数なので最下位ビットは常に 0 となるので指定しない．

## C.3 アドレッシングモード

アドレッシングモードとはメモリ上の位置 (メモリ番地) を指定する方法のことである．KAPPA3-RV32I では大まかには 1 種類のアドレッシングモードしかサポートしない．これは ‘指定されたレジスタ (ベースレジスタ) の値’ + ‘オフセット’ の形で与えられる．ただしベースレジスタの種類とオフセットの形式で以下の 4 種類がある．

- I-type:  $R_{s1}$  がベースレジスタとして用いられる．12 ビットの即値は符号拡張されてベースレジスタの値と加算される．
- S-type: I-type と同じく  $R_{s1}$  がベースレジスタとして用いられ，12 ビットの即値は符号拡張されてベースレジスタの値と加算される．ただし，即値のフィールドが I-type と異なっている．
- B-type: 命令中には明示されていないが，PC (プログラムカウンタ) がベースレジスタとして用いられる．さらに 12 ビットの即値は符号拡張されたあとで 2 倍されてから PC の値に加算される．
- J-type: こちらも PC をベースレジスタとして使用する．J-type の即値は 20 ビットであるが，S-type と同様に符号拡張されたあとで 2 倍してから PC に加算される．

このようにロード・ストア命令では汎用レジスタをベースレジスタに用い，分岐・ジャンプ命令では PC をベースレジスタに用いるようになっている．ただし，JALR 命令だけは例外で  $R_{s1}$  フィールドで指定された汎用レジスタがベースレジスタとして用いられる．この命令のみが現在の PC と無関係なアドレスにジャンプすることができる．他の分岐・ジャンプ命令が PC 相対アドレスを用いている理由は，プログラムがどこに配置されても命令中の分岐先アドレスの指定を書き換える必要がないようにするためである．このようにプログラムの配置位置によって内容を書き換える必要のないコードを PIC (position independent code) コードと呼ぶ．PIC コードを用いることでプログラム開発で用いられるリンカ・ローダの処理が大幅に簡単化される．

## C.4 命令セット

### C.4.1 即値ロードとジャンプ命令

表 C.2 命令セット (1)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[31:12]										rd		0110111		LUI
imm[31:12]										rd		0010111		AUIPC
imm[20 10:1 11 19:12]										rd		1101111		JAL
imm[11:0]						rs1		000		rd		1100111		JALR
imm[12 10:5]				rs2		rs1		000		imm[4:1 11]		1100011		BEQ
imm[12 10:5]				rs2		rs1		001		imm[4:1 11]		1100011		BNE
imm[12 10:5]				rs2		rs1		100		imm[4:1 11]		1100011		BLT
imm[12 10:5]				rs2		rs1		101		imm[4:1 11]		1100011		BGE
imm[12 10:5]				rs2		rs1		110		imm[4:1 11]		1100011		BLTU
imm[12 10:5]				rs2		rs1		111		imm[4:1 11]		1100011		BGEU

LUI(Load Upper Immediate) : U-Type

レジスタの上位 20 ビットに即値をロードする．下位 12 ビットは常に 0 となる．

AUIPC(Add Upper Immediate to PC) : U-type

PC に即値を足す．ただし，即値は上位 20 ビットを指定したもの．下位 12 ビットは 0 を足すものとみなす．

JAL(Jump And Link) : J-type

次の PC の値 (自分自身のアドレス +4) を  $R_d$  に入れ，指定されたアドレスにジャンプする． $R_d$  は戻り先のアドレスとして用いられる．

JALR(Jump And Link Register) : I-type

次の PC の値 (自分自身のアドレス +4) を  $R_d$  に入れ， $R_{s1} + imm$  のアドレスにジャンプする． $R_d$  は戻り先のアドレスとして用いられる．

ここで  $imm$  は即値フィールドで指定された即値である． $imm$  は符号付き 12 ビット整数として扱われる．

BEQ(Branch Equal) : B-type

$R_{s1} = R_{s2}$  の時に現在の PC の値 (自分自身のアドレス) に即値を加えたアドレスにジャンプする．即値は符号付き 13 ビット整数として扱われる．このフォーマットの即値のエンコーディングは複雑なので注意すること．分岐条件のみが異なる命令として BNE(Branch Not Equal:  $R_{s1} \neq R_{s2}$  の時に分岐する)，BLT(Branch Less Than:  $R_{s1} < R_{s2}$  の時に分岐する)，BGE(Branch Greater Than or Equal:  $R_{s1} \geq R_{s2}$  の時に分岐する)，BLTU(Branch Less Than Unsigned: 符号なし整数と見なして  $R_{s1} < R_{s2}$  の時に分岐する) BGEU(Branch Greater Than or Equal Unsigned: 符号なし整数と見なして  $R_{s1} \geq R_{s2}$  の時に分岐する) がある．

## C.4.2 ロード・ストア命令

表 C.3 命令セット (2)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[11:0]						rs1	000			rd	0000011			LB
imm[11:0]						rs1	001			rd	0000011			LH
imm[11:0]						rs1	010			rd	0000011			LW
imm[11:0]						rs1	100			rd	0000011			LBU
imm[11:0]						rs1	101			rd	0000011			LHU
imm[11:5]				rs2		rs1	000			imm[4:0]	0100011			SB
imm[11:5]				rs2		rs1	001			imm[4:0]	0100011			SH
imm[11:5]				rs2		rs1	010			imm[4:0]	0100011			SW

LB(Load Byte) : I-type

1 バイトの値をメモリから読み出す．読み出すアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．読み出された値は符号拡張されて $R_d$  に入る．

LH(Load Half word) : I-type

2 バイト (half word) の値をメモリから読み出す．読み出すアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．読み出された値は符号拡張されて $R_d$  に入る．アドレスは偶数でなければならない．

LW(Load Word) : I-type

4 バイト (word) の値をメモリから読み出す．読み出すアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．読み出された値は  $R_d$  に入る．アドレスは 4 の倍数でなければならない．

LBU(Load Byte Unsigned) : I-type

1 バイトの値をメモリから読み出す．読み出すアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．読み出された値はそのまま  $R_d$  に入る．上位 24 ビットには 0 が入る．

LHU(Load Half word Unsigned) : I-type

2 バイト (half word) の値をメモリから読み出す．読み出すアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．読み出された値はそのまま  $R_d$  に入る．上位 16 ビットには 0 が入る．

SB(Store Byte) : S-type

1 バイトの値をメモリに書き込む．書き込むアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．書き込む値は  $R_{s2}$  を用いる．

SH(Store Half word) : S-type

2 バイト (half word) の値をメモリに書き込む．書き込むアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．書き込む値は  $R_{s2}$  を用いる．アドレスは偶数でなければならない．

SW(Store Word) : S-type

4 バイト (word) の値をメモリに書き込む．書き込むアドレスは  $R_{s1} + imm$  で指定する．ここで  $imm$  は即値フィールドで指定された 12 ビットの符号付き整数である．書き込む値は  $R_{s2}$  を用いる．アドレスは 4 の倍数でなければならない．

ロード命令は LB, LH, LW, LBU, LHU の 5 種類が存在する．この内，2 文字目が 'B' の命令 (LB, LBU) はバイト (Byte: 8 ビット) 単位のアクセスを行う．2 文字目が 'H' の命令 (LH, LHU) はハーフワード (Half Word: 16 ビット) 単位のアクセスを行う．2 文字目が 'W' の命令はワード (Word: 32 ビット) 単位のアクセスを行う．3 文字目に 'U' のついた命令 (LBU, LHU) は読み出された値を符号なし数とみなして扱う．それ以外の 2 文字の命令 (LB, LH, LW) は読み出された対を符号付き数とみなして扱う．符号の有りなしは 8 ビット/16 ビットの値を符号拡張するかどうかに影響する．今，8 ビットで読み出した値が  $8'b1111.1111$  だとする\*1．これを符号なし数とみなすと 10 進数では 255 となる．一方符号付き数とみなすと 10 進数では -1 となる．それを 32 ビットに拡張すると，それぞれ  $32'b0000.0000.0000.1111.1111$  と  $32'b1111.1111.1111.1111.1111$  になる．

ストア命令もロード命令と同様にアクセスする単位に応じて SB, SH, SW の 3 種類が存在する．ただし，書き込む場合にはビット拡張を行わないので符号の有りなしの区別はない．

KAPPA3-RV32I では命令語長が 32 ビットなのでメモリアクセスも 32 ビット単位で行えると効率がよい．しかし，ロード/ストア命令において 8 ビット/16 ビット単位のメモリアクセスが発生した場合に少し考慮が必要となる．まず，簡単なロード命令から考える．今， $32'h8765.4321$  番地に対してバイト (8 ビット) のロード (読み出し) を行うと仮定する．メモリの番地は 1 バイトごとに割り振られているが，実際には 32 ビット (=4 バイト) がひとかたまりになっているので，アクセスする範囲は  $32'h8765.4320$  番地から  $32'h8765.4323$  番地までの 4 バイトとなる (図 C.1) ．

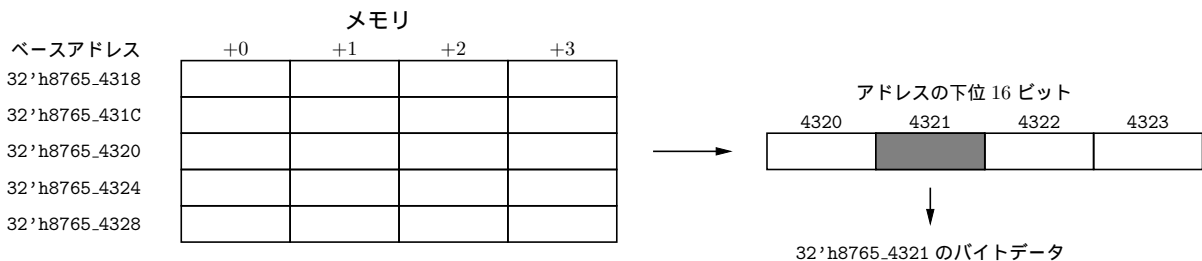


図 C.1 バイト単位のロードの例

このうち，必要なのは 2 番めのバイトだけなので，LB 命令では一旦 32 ビットのデータを読み出し，そのなかの 8 ビット分を切り出す処理を行う必要がある．16 ビット単位のロードの場合も同様の処理を行えばよい．

少し工夫が必要なのがストア命令である．上記の例と同様に  $32'h8765.4321$  番地にバイト単位のストアを行うことを考える．この場合， $32'h8756.4320$  番地や  $32'h8756.4322$  番地の内容を書き換えてはいけない．愚直には一旦， $32'h8765.4320$  番地から  $32'h8765.4323$  番地までの 4 バイトを一時的な保管場所に読み出し，その中の 2 バイト目だけを書き換えて，もとの場所へ書き戻すやり方が考えられるが，すると 1 回ストア命令を実行するために 1 回の読み出しと 1 回の書込みが発生するため効率が悪い．そこで，メモリ側に工夫をして，書き込みを行うバイトを指定するビットマスクを用意する．具体的には wrbits という 4 ビットの入力信号線をメモリに追加する．メモリの書込みが発生したときにはこの wrbits が 1 になっているバイトのみ書き込みを行うものとする．先程の例では 2 バイト目のみ書き込むので  $wrbits = 4'b0010$  となる．普通に 4

\*1 Verilog-HDL では数値表記中の . は無視されることに注意．ここでは 4 ビットの区切り文字として用いている．

バイト (32 ビット) すべてに書き込む場合には `wrbits = 4'b1111` とすればよい。

## C.4.3 即値演算命令

表 C.4 命令セット (3)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
imm[11:0]						rs1		000		rd		0010011		ADDI
imm[11:0]						rs1		010		rd		0010011		SLTI
imm[11:0]						rs1		011		rd		0010011		SLTIU
imm[11:0]						rs1		100		rd		0010011		XORI
imm[11:0]						rs1		110		rd		0010011		ORI
imm[11:0]						rs1		111		rd		0010011		ANDI
0000000				shamt		rs1		001		rd		0010011		SLLI
0000000				shamt		rs1		101		rd		0010011		SRLI
0100000				shamt		rs1		101		rd		0010011		SRAI

ADDI(ADD Immediate) : I-type

$R_{s1} + imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.

SLTI(Set Less Than Immediate) : I-type

$R_{s1} < imm$  なら  $R_d$  に 1 を入れ, そうでなければ 0 を入れる.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.

SLTIU(Set Less Than Immediate Unsigned) : I-type

$R_{s1} < imm$  なら  $R_d$  に 1 を入れ, そうでなければ 0 を入れる.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである. ただし, 大小比較は符号なし整数と見なしで行う.

XORI(XOR Immediate) : I-type

$R_{s1} \oplus imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\oplus$  演算はビットごとに排他的論理和を計算する.

ORI(OR Immediate) : I-type

$R_{s1} \vee imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\vee$  演算はビットごとに論理和を計算する.

ANDI(AND Immediate) : I-type

$R_{s1} \wedge imm$  の計算を行い,  $R_d$  に格納する.  $imm$  は即値フィールドで指定された 12 ビット符号付き整数を 32 ビットに拡張したものである.  $\wedge$  演算はビットごとに論理積を計算する.

SLLI(Shift Left Logical Immediate) : I-type

$R_{s1}$  の値を左に (最上位ビットの方向に) 論理シフトした結果を  $R_d$  に格納する. シフト量は最大で 31 ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (shamt) を用いる. SLLI の場合には上位 7 ビットは常に 0 となっている. 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと.

SRLI(Shift Right Logical Immediate) : I-type

$R_{s1}$  の値を右に (最下位ビットの方向に) 論理シフトした結果を  $R_d$  に格納する. シフト量は最大で 31



ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (shamt) を用いる。SRLI の場合には上位 7 ビットは常に 0 となっている。論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと。

SRAI(Shift Right Arithmetic Immediate) : I-type

$R_{s1}$  の値を右に (最下位ビットの方向に) 算術シフトした結果を  $R_d$  に格納する。シフト量は最大で 31 ビット (32 ビットシフトしたらなにも残らない) なので即値フィールドの下位 5 ビット (shamt) を用いる。SRAI の場合には上位 7 ビットは常に 0100000 となっている。算術シフトとはシフトによって空いたビットに最上位ビットの値を入れるシフトのこと。つまり、符号付き整数と見なした時にシフトによって符号が変化しない。

## C.4.4 レジスタ演算命令

表 C.5 命令セット (4)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
0000000				rs2		rs1		000		rd		0110011		ADD
0100000				rs2		rs1		000		rd		0110011		SUB
0000000				rs2		rs1		001		rd		0110011		SLL
0000000				rs2		rs1		010		rd		0110011		SLT
0000000				rs2		rs1		011		rd		0110011		SLTU
0000000				rs2		rs1		100		rd		0110011		XOR
0000000				rs2		rs1		101		rd		0110011		SRL
0100000				rs2		rs1		101		rd		0110011		SRA
0000000				rs2		rs1		110		rd		0110011		OR
0000000				rs2		rs1		111		rd		0110011		AND

ADD(ADD) : R-type

$R_{s1} + R_{s2}$  の結果を  $R_d$  に格納する .

SUB(SUB) : R-type

$R_{s1} - R_{s2}$  の結果を  $R_d$  に格納する .

SLL(Shift Left Logical) : R-Type

$R_{s1}$  の値を左に  $R_{s2}$  だけ論理シフトを行う . 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと .

SLT(Set Less Than) : R-type

$R_{s1} < R_{s2}$  の時  $R_d$  に 1 を入れ , そうでない時に 0 を入れる . 大小比較は符号付き整数と見なして行う .

SLTU(Set Less Than Unsigned) : R-Type

$R_{s1} < R_{s2}$  の時  $R_d$  に 1 を入れ , そうでない時に 0 を入れる . 大小比較は符号無し整数と見なして行う .

XOR(XOR) : R-type

$R_{s1} \oplus R_{s2}$  の結果を  $R_d$  に格納する .  $\oplus$  演算はビットごとの排他的論理和 .

SRL(Shift Right Logical) : R-Type

$R_{s1}$  の値を右に  $R_{s2}$  だけ論理シフトを行う . 論理シフトとはシフトによって空いたビットに 0 を入れるシフトのこと .

SRA(Shift Right Arighmetic) : R-Type

$R_{s1}$  の値を右に  $R_{s2}$  だけ算術シフトを行う . 算術シフトとはシフトによって空いたビットに最上位ビットの値を入れるシフトのこと . つまり , 符号付き整数と見なした時にシフトによって符号が変化しない .

OR(OR) : R-type

$R_{s1} \vee R_{s2}$  の結果を  $R_d$  に格納する .  $\vee$  演算はビットごとの論理和 .

AND(AND) : R-type

$R_{s1} \wedge R_{s2}$  の結果を  $R_d$  に格納する .  $\wedge$  演算はビットごとの論理積 .

## C.5 特権命令と割り込み処理

ここでは KAPPA3-RV32I の特権命令と割り込み処理について述べる．KAPPA3-LIGHT ではこの機能は実装しない．RISC-V では以下の 4 つの特権レベルを仮定している．

- マシンモード
- ユーザモード
- スーパーバイザモード
- ハイパーバイザモード

KAPPA3-RV32I ではこのうちのマシンモードのみを実装する．マシンモードではすべての特権命令が実行可能であり，セキュリティ的には脆弱だが特権レベルの管理や特権レベルの移動がないので実装は単純となる．

RISC-V の特権命令を表 C.6 に示す．ただし，マシンモード以外のモードで用いる命令は省いている．

表 C.6 命令セット (3)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	命令
0011000				00010		00000		000		00000		1110011		MRET
csr						rs1		001		rd		1110011		CSRRW
csr						rs1		010		rd		1110011		CSRRS
csr						rs1		011		rd		1110011		CSRRC
csr						zimm		101		rd		1110011		CSRRWI
csr						zimm		110		rd		1110011		CSRRSI
csr						zimm		111		rd		1110011		CSRRCI

MRET(Machine mode RETurn): R-Type

マシンモードにおける割り込み処理からの復帰．具体的な動作は後述．

CSR RW(CSR Read and Write): I-Type

CSR の読み込み & 書き込み．

CSR RS(CSR Read and Set): I-Type

CSR の読み込み & ビットセット．

CSR RC(CSR Read and Clear): I-Type

CSR の読み込み & ビットクリア．

CSR RWI(CSR Read and Write Immediate): I-Type

CSR の読み込み & 即値書き込み．

CSR RSI(CSR Read and Set Immediate): I-Type

CSR の読み込み & 即値ビットセット．

CSR RCI(CSR Read and Clear Immediate): I-Type

CSR の読み込み & 即値ビットクリア．

CSR で始まる命令は CSR(コントロール・ステータス・レジスタ) のアクセス命令であり，厳密には特権命令ではないが特権レベルの処理に関係が深いのでここに含める．実際，MRET と CSR 系の命令のオPCODEは同じ 110011 であり，同一の命令グループとして設計されている．

表 C.7 KAPPA3-RV32I の CSR

アドレス	ニーモニック	意味	備考
0x300	MSTATUS	全般の状態	MSTATUS[3] と MSTATUS[7] のみ意味を持つ。 MIE[7] のみ意味を持つ。
0x304	MIE	割り込み許可ビットマスク	
0x305	MTVEC	割り込みテーブルアドレス	
0x340	MSCRATCH	割り込みハンドラが使う一時レジスタ	
0x341	MEPC	割り込み時の復帰アドレス	
0x342	MCAUSE	割り込み原因	常に 32'h8000_0007
0x344	MIP	割り込み待ち状態	MIP[7] のみ意味を持つ。

RISC-V の割り込み・例外は以下の通り。

- アクセスフォールト例外
- ブレークポイント例外
- 環境呼び出し例外
- 不正命令例外
- 非整列化アドレス例外
- ソフトウェア割り込み
- タイマー割り込み
- 外部割り込み

KAPPA3-RV32I では簡単化のためにマシンモードのタイマー割り込みのみを扱うものとする。

### C.5.1 CSR と CSR 操作命令

CSR(Control Status Register) は多数の 32 ビットレジスタでおもに特権命令や割り込みに関する制御のために用いられる。命令セット上では 12 ビットのアドレスで指定されたレジスタファイルであるが、 $2^{12} = 4096$  個全てに意味のあるレジスタが実装されているわけではなく、また、特権モードや浮動小数点演算の実装の有無などで意味を持たないものも含まれる。KAPPA3-RV32I で実装する CSR を表 C.7 に示す。ニーモニックは特定のアドレスのレジスタにつけた呼び名である。以降はこのニーモニックを用いて CSR レジスタを参照する。たとえば CSR[0x300] は MSTATUS である。また MSTATUS[3] は MSTATUS レジスタの 3 ビット目を表す (Verilog-HDL 表記)。

MSTATUS は 32 ビット長であるが、今回は MSTATUS.MPIE = MSTATUS[7] と MSTATUS.MIE = MSTATUS[3] の 2 つのビットしか使用しない。実際には 2 ビットのみをレジスタとして実装して残りは読み出しに対しては常に 0 を返す。MSTATUS.MIE はマシンモードにおいて割り込みを許可する時 1 にセットするフラグである。MSTATUS.MPIE はマシンモードにおける割り込みハンドラ中で MSTATUS.MIE をクリアする時に元の値を保存しておくためのレジスタである。

MIE は個々の割り込み要因の許可/不許可 (enable の e) を表すビットマスクであるが、今回はマシンモードのタイマーしか実装しないので、MIE.MTIE = MIE[7] のみ実装する。他のビットは常に 0 を返す。

MIP は個々の割り込みが処理待ちかどうか (pending の p) を表すビットマスクであるが、MIE と同様にマシンモードのタイマーしか実装しないので、MIP.MTIP = MIP[7] のみ実装する。他のビットは常に 0 を返す。

MIP.MTIP に対する書き込みは許可されない。

MTVEC は割り込みが起こった時にジャンプするアドレスを保持する。正確には  $MTVEC[1:0] = 2'b00$  の時は MTVEC で示されたアドレスにジャンプし、 $MTVEC[1:0] = 2'b01$  の時は MTVEC の下位 2 ビットをクリアした値をベースアドレスとして、そこから  $MCAUSE \times 4$  のアドレスにジャンプするベクタモードもあるが KAPPA3-RV32I ではベクタモードを実装しない。

MSCRATCH は割り込みハンドラが処理の最初に 1 つの汎用レジスタの値を保存するために用いる。同時に前もって割り込みハンドラが使用するスタック領域の先頭 (底) アドレスを保持しておく。こうすることで、他の汎用レジスタを割り込みハンドラ用のスタック領域に退避させることができる。

MEPC は割り込みが起こった時の PC の値を保持する。これは割り込みハンドラからの復帰命令 MRET 時に復帰先アドレスとして用いられる。

MCAUSE は割り込み原因を表す。マシンモードのタイマー割り込みは  $32'h8000\_0007$  である。

CSR は通常のメモリとは異なるメモリ空間にマップされているため独自のアクセス命令が用意されている。すべての命令は I-type で、 $R_{s1}$ ,  $R_d$ ,  $Imm(12 \text{ ビット})$  のフィールドを持つ。このうち  $Imm$  は CSR のアドレスとして用いられる。 $R_{s1}$  は通常はソースレジスタを表すが、即値系の命令 `csrrwi`, `csrrsi`, `csrrci` では 5 ビットの即値 ( $zimm$ ) として用いられる。値は上位 27 ビットに 0 が拡張されてから用いられる。

以下に各命令の動作を示す。

- CSRRW CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値を CSR に書き込む。
- CSRRS CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値とのビットワイズ OR を CSR に書き込む。
- CSRRC CSR の値を  $R_d$  に読み出し、 $R_{s1}$  の値の反転値とのビットワイズ AND を CSR に書き込む。
- CSRRWI CSR の値を  $R_d$  に読み出し、即値の値を CSR に書き込む。
- CSRRSI CSR の値を  $R_d$  に読み出し、即値の値とのビットワイズ OR を CSR に書き込む。
- CSRRCI CSR の値を  $R_d$  に読み出し、即値の値の反転値とのビットワイズ AND を CSR に書き込む。

即値系の命令では即値が 5 ビットしかないため CSR の上位 27 ビットに値を設定することはできない。

## C.5.2 割り込み処理

RISC-V では  $MSTATUS.MIE = 1 \wedge MIE.MTIE = 1 \wedge MIP.MTIP = 1$  の時に割り込み処理が行われる。具体的には以下の処理を行う。

- $MEPC \leq PC$
- MCAUSE に原因を設定。ここではマシンモードのタイマー割り込みに固定。
- $MIP.MTIP \leq 0$
- $MSTATUS.MPIE \leq MSTATUS.MIE$
- $MSTATUS.MIE \leq 0$
- $PC \leq MTVEC$

PC に MTVEC の値が設定されることで次の命令実行時には割り込み処理ルーチンに処理が移行する。割り込み処理ルーチンでは処理の最後に MRET 命令でもとのプログラムに復帰する。その際の具体的な処理は以下の通り。

- $PC \leq MEPC$
- $MSTATUS.MIE \leq MSTATUS.MPIE$

割り込みからの復帰 MRET 命令を実装するためには DE フェイズ中で MRET 命令かどうかの判断を行い、MRET 命令の場合に WB フェイズでは上記のように PC の値の復帰などを行った後に IF フェイズに遷移する。

### C.5.3 タイマー

タイマーは命令実行とは独立して時刻をカウントするハードウェアで、RISC-V からは通常のメモリ空間にマップされたアドレスを介してアクセスする。ここでは 2 つの 64 ビットレジスタ `mtime` と `mtimecmp` を用意する。`mtime` はリセット時に 0 に初期化され、その後 1 サイクル (システムクロック) ごとに 1 つ値が増やされる読み出しのみのレジスタである。`mtimecmp` は読み書き可能なレジスタで、この値と `mtime` の値が一致した時にタイマー割り込みが発生する。具体的には `MIP.MTIP` が 1 になる。

## C.6 構成

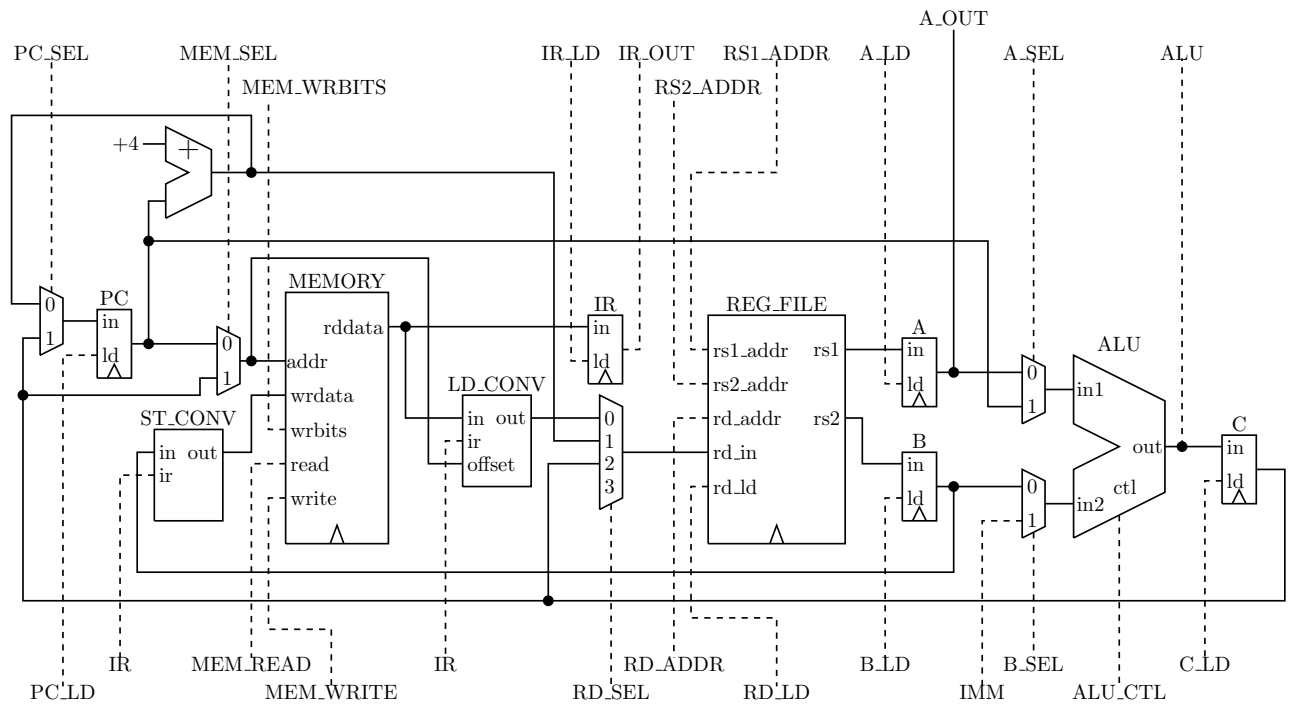


図 C.2 KAPPA3-RV32I の構成

図 C.2 に KAPPA3-RV32I の論理的な構成を示す。KAPPA3-RV32I は大きく分けて以下の部品から構成される。なお、下方および上方に伸びている破線はコントローラへの入出力である。またフリップフロップ (レジスタ) に対するクロック信号とリセット信号は省略している。詳細は後述するが、デバッグ用の各レジスタの値を観測したり設定したりする信号線も省略している。また、CSR も省略している。CSR に対する入力としては A\_OUT およびいくつかの制御入力を用いられる。CSR の出力は CSR\_OUT として RD\_SEL の入力に用いられる。

**PC** プログラムカウンタ。次に実行すべき命令のアドレスを保持する。汎用の 32 ビットレジスタを用いる。PC に関するコントロールは以下の通り。

- PC\_LD: PC に値を書き込む時 1 にする。

**PC\_SEL** PC への入力を選択するセクタ (マルチプレクサ) コントロールは以下の通り。

- 0: PC + 4 を用いる。
- 1: C レジスタの値を用いる。

**MEMORY** メモリ。命令プログラムおよびデータを格納する。ADDR にアクセスするメモリアドレスを指定する。DATA に書き込む値を指定する。読み出された値は OUT に出力される。メモリに関するコントロールは以下の通り。

- MEM\_READ: メモリの値を読み出す時に 1 にする。
- MEM\_WRITE: メモリに値を書き込む時に 1 にする。
- MEM\_WRBITS: メモリに書き込む対象を指定するビットマスク

MEM\_SEL メモリのアドレスを選択するセクタコントロールは以下の通り。

- 0: PC の値を用いる。
- 1: C レジスタの値を用いる。

IR 現在実行中の命令を保持するレジスタ。汎用の 32 ビットレジスタを用いる。読み出された値は IR\_OUT に出力される。IR に関するコントロールは以下の通り。

- IR\_LD: IR に値を書き込む時に 1 にする。

REG\_FILE:レジスタファイル データを一時的に格納しておくために、R0 — R31 の 32 本の 32 ビットレジスタを持つ。このような同種の汎用レジスタの集まりをレジスタファイルと呼ぶ。2 つの読み出しポート (RS1 と RS2) と 1 つの書き込みポート (DATA) を持つ。

- RS1\_ADDR: RS1 で読み出すレジスタ番号を指定する。
- RS2\_ADDR: RS2 で読み出すレジスタ番号を指定する。
- RD\_ADDR: 書き込むレジスタ番号を指定する。
- RD\_LD: レジスタに書き込む時に 1 にする。

R0 レジスタは特殊なレジスタで読み出し結果は常に 0 となり、書き込みはなにも行われない (エラーにもならない)。

RD\_SEL レジスタファイルに書き込む入力を選択するセクタコントロールは以下の通り。

- 0: メモリの出力を用いる。
- 1: PC レジスタの値を用いる。
- 2: C レジスタの値を用いる。
- 3: CSR の出力を用いる (KAPPA3-RV32I のみ)。

A, B レジスタファイルから読み出した値を保存しておく 2 つの 32 ビットレジスタ。A レジスタは  $R_{s1}$  フィールドで指定されたレジスタの値を B レジスタは  $R_{s2}$  フィールドで指定されたレジスタの値を格納する。A, B レジスタに関するコントロールは以下の通り。

- A\_LD: A レジスタに値を書き込む時 1 にする。
- B\_LD: B レジスタに値を書き込む時 1 にする。

なお、A レジスタの値 (A\_OUT) は CSR への入力にも用いられる。

ALU 演算を行う中心部分である。ALU 自体は記憶を持たない組み合わせ回路である。ALU は 2 つの 32 ビットの入力と 32 ビットの出力を持つ。ALU に関するコントロールは以下の通り

- ALU\_CTL: ALU で行う演算を指定する。詳細は AppendixC.9.5 参照。

ASEL ALU の入力 1(in1) の入力を選択するセクタ。コントロールは以下の通り。

- 0: A レジスタの値を用いる。
- 1: PC レジスタの値を用いる。

BSEL ALU の入力 2(in2) の入力を選択するセクタ。コントロールは以下の通り。

- 0: B レジスタの値を用いる。
- 1: 即値 (IMM) を用いる。

C ALU の演算結果を保存しておく 32 ビットレジスタ。C\_LD を 1 にすると ALU の値を書き込む。

CSR CSR(Control Status Register)。図 C.2 では省略されている。詳細は C.5.1 節を参照。アクセス対象を指定する CSR\_ADDR と入力値 CSR\_IN、操作 (Write—Set—Clear) を指定する CSR\_OP の入力と出力 CSR\_OUT を持つ。

- CSR\_ADDR は IMM に接続する。
- CSR\_IN は場合によって A\_OUT か RS1\_ADDR を接続する。
- CSR\_OP はコントローラで生成する。



- CSR\_OUT は RD\_SEL セレクタの入力に接続する .

## C.7 フェイズ

KAPPA3-RV32I は多くのプロセッサと同様に複数クロックの動作で一つの命令の実行を行う。ここでは 1 クロックで行う動作を「フェイズ」と呼ぶことにする。KAPPA3-RV32I で単純化のため、全ての命令に対して同一のフェイズ構成を用いる。

IF(Instruction Fetch): 命令読み出しフェイズ。

以下の処理を行う。

1. PC が指すアドレスのメモリの値を IR へ代入する。

DE(DEcode): 命令解析フェイズ。

以下の処理を行う。

1. IR 中の  $R_{s1}$  フィールドで示されたレジスタの値を A レジスタに入れる。
2. IR 中の  $R_{s2}$  フィールドで示されたレジスタの値を B レジスタに入れる。

EX(EXecute): 実行フェイズ。

以下の処理を行う。

1. 演算命令の場合は演算を行い、結果を C レジスタに入れる。
2. ロード命令、ストア命令、ジャンプ命令の場合はアドレス計算を行い、結果を C レジスタに入れる。
3. ストア命令の場合には書き込む値 ( $R_d$ ) を C レジスタに入れる。

WB(Write Back): メモリ/ライトバックフェイズ。

以下の処理を行う。

1. 演算命令の場合は C レジスタの値を  $R_d$  で指定されたレジスタに書き込む。
2. ロード命令の場合はメモリから値を読み出し  $R_d$  で指定されたレジスタに書き込む。
3. ストア命令の場合は B レジスタの値を C レジスタで指定されたメモリのアドレスに書き込む。
4. ジャンプ命令の場合には C レジスタの値を PC レジスタに入れる。
5. 分岐命令の場合には分岐条件を調べる。条件が成り立っていたら C レジスタの値を PC レジスタに入れる。
6. ジャンプ命令、分岐命令以外の場合には PC の値を 4 加算する (32 ビット分)。

IR(Interrupt): 割り込みフェイズ。

割り込み要求があった時に実行されるフェイズ。以下の処理を行う。

1.  $MEPC \leftarrow PC$  復帰用の PC アドレスを保存
2. MCASE に原因を設定。ここではマシンモードのタイマー割り込みに固定。
3.  $MIP.MTIP \leftarrow 0$  タイマー割り込み要求をクリアする。
4.  $MSTATUS.MPIE \leftarrow MSTATUS.MIE$  割り込み許可状態を保存する。
5.  $MSTATUS.MIE \leftarrow 0$  新たなタイマー割り込みを禁止する。
6.  $PC \leftarrow MTVEC$  割り込みハンドラのアドレスを PC に設定する。

各フェイズのうち、IF および DE フェイズは命令の種類に関わらず同一の処理を行う。残りのフェイズでは命令に応じて異なる処理を行う。

通常は ALU の演算は EX フェイズでのみ実行されるが、BEQ(条件分岐命令) では分岐先アドレスの計算を EX フェイズで行い、分岐条件の判断を WB フェイズで行っている。

CSR 系の命令は CSR で主な処理を行う。ALU は用いられない。

通常は IF → DE → EX → WB → IF ... を繰り返すが、割り込み要求があった場合には、WB から IF へ

表 C.8 フェイズ表

命令	IF	DE	EX	WB
ADD 系	$IR \leftarrow \text{mem}[PC]$	$A \leftarrow \text{reg}[R_{s1}]$  $B \leftarrow \text{reg}[R_{s2}]$	$C \leftarrow A + B$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
ADDI 系			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
Load 系			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow \text{mem}[C]$ $PC \leftarrow PC + 4$
Store 系			$C \leftarrow A + \text{S\_imm}$	$\text{mem}[C] \leftarrow B$ $PC \leftarrow PC + 4$
LUI			$C \leftarrow \text{U\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
AUIPC			$C \leftarrow PC + \text{U\_imm}$	$\text{reg}[R_d] \leftarrow C$ $PC \leftarrow PC + 4$
JAL			$C \leftarrow PC + \text{J\_imm}$	$\text{reg}[R_d] \leftarrow PC + 4$ $PC \leftarrow C$
JALR			$C \leftarrow A + \text{I\_imm}$	$\text{reg}[R_d] \leftarrow PC + 4$ $PC \leftarrow C$
BEQ			$C \leftarrow PC + \text{B\_imm}$	$\text{if } (A == B) \text{ } PC \leftarrow C$
MRET				$PC \leftarrow \text{MEPC}$ $\text{MSTATUS.MIE} \leftarrow \text{MSTATUS.MPIE}$
CSR RW				$\text{reg}[R_d] \leftarrow \text{CSR}[\text{I\_imm}]$ $\text{CSR}[\text{I\_imm}] \leftarrow A$ $PC \leftarrow PC + 4$
CSR RWI				$\text{reg}[R_d] \leftarrow \text{CSR}[\text{I\_imm}]$ $\text{CSR}[\text{I\_imm}] \leftarrow \text{zimm}$ $PC \leftarrow PC + 4$

移行する前に IR フェイズを実行する (KAPPA3-RV32I のみ) . IR フェイズで PC の値が変更されるため , 割り込みハンドラへ処理が移ることになる .

## C.8 KAPPA3-LIGHT 用の入力・表示モジュール

ここでは KAPPA3-LIGHT の動作を制御したり，内部の状態を観測するための FPGA ボードの仕様について述べる．FPGA ボードそのものの説明は付録 B を参照のこと．

### C.8.1 モード

KAPPA3-LIGHT には 2 つのモード — 入力モードと動作モード — がある．入力モードは KAPPA3-LIGHT 上のメモリやレジスタに値を書き込むためのモードであり，DIP スイッチの DIP\_A-0 を on にすることで入力モードに切り替えることができる．動作モードはプログラムを実行するためのモードで，1 フェイズごとに停止させたり，1 命令ごとに停止させることも可能である．動作モードには DIP\_A-0 を off にすることで切り替えることができる．DIP スイッチは上が on，下が off である．具体的には入力モードと動作モードによってプッシュ SW の右側の 1 列の動作のみが異なる．

### C.8.2 スイッチおよび LED

clock クロックの周波数を調節するロータリースイッチ．クロックが速すぎるとキー入力時に誤動作するのでクロック周期に同期した LED の点滅が目で分かる程度の速度に設定しておくこと．逆にクロックが遅すぎるとキーを押しても反応しない．通常は C ~ E の目盛りで使用する．

reset リセットスイッチ．回路中の reset 信号が直結している．メモリ以外の全レジスタを 0 に初期化する．  
 プッシュスイッチ 0 ~ F までの 16 個の数字キーと 'clear'，'+', '-', '=', の 4 個のキーからなる．数字キーは入力バッファに値を入力するために用いられる．右側の 4 つのキーはモードに応じて異なる働きをする (表 C.9)．

表 C.9 キーの機能

入力モード		動作モード	
'clear'	クリア	—	未使用
‘+’	アドレスを 4 増加	SP	実行 / 停止
‘-’	アドレスを 4 減少	SI	フェイズごとに停止
‘=’	書き込み	SS	命令ごとに停止

クリアキーが押されたときには入力バッファの値が 0 にクリアされる．KAPPA3-LIGHT のレジスタ・メモリには影響しない．‘+’ および ‘-’ はメモリのアクセスに対するアドレス (具体的には MAR の値) を加算もしくは減算するためのものである．KAPPA3 は 32 ビット (4 バイト) が一語なので 4 つ単位で増減する．‘=’ キーが押されたときには入力バッファの値が対象のレジスタ・メモリに書き込まれる．  
 DIP\_A-0 入力モードと動作モードを切り替えるのに用いる．off の時に動作モード．on の時に入力モードとなる．

HEX\_A メモリおよび内部レジスタの値を観測したり，値を入力するときに対象の要素を指定する (表 C.10)

HEX\_B 汎用レジスタを選択するのみ用いる (表 C.10)．ただし，このスイッチが意味を持つのは HEX\_A で汎用レジスタを指定しているときに限る．

表 C.10 ロータリー SW の意味





HEX_A	HEX_B	意味
0	—	PC
1	—	IR
2	—	A
3	—	B
4	—	C
5	—	未使用
6	offset	reg[offset]]
7	offset	reg[16 + offset]
8	—	MAR
9	—	memory

RK-A ~ RK-H 入力バッファの値を表示する．

7SEG-A ~ 7SEG-H レジスタ・メモリの内容を表示する．左側の 4 つのグループは現在表示している対象を示す．右側の 4 つのグループは実際の値を示す．そのため一度に 8 つの対象しか表示することができないため，HEX\_A および HEX\_B で指定された対象によって異なる表示内容となる．





HEX\_A の値が 0, 1, 2, 3 の時は表 C.11 の様な表示となる。

表 C.11 ページ 1

左側の 7SEG-LED の表示内容	意味	右側の 7SEG-LED の表示内容
	PC	PC の値
	IR	IR の値
	AREG	A レジスタの値
	BREG	B レジスタの値

HEX\_A の値が 6, 7, 8, 9 で HEX\_B の値が 0 の時は表 C.12 の様な表示となる。

表 C.12 ページ 2

左側の 7SEG-LED の表示内容	意味	右側の 7SEG-LED の表示内容
	CREG	C レジスタの値
	R00	レジスタファイルの値 (この例では reg[0])
	MAR	MAR の値
	MDR	mem[MAR] の値

2 行目の R00 は実際には HEX\_A, HEX\_B の値に応じて異なる表示となる。入力モードでは現在の書込み対象の 7SEG-LED がゆっくりと点滅する。テンキーを押すと MU500-RK 側の 7SEG-LED に値が入力されるので、‘=’ キーを押すと対象のレジスタ・メモリに値が書き込まれる。メモリに値を書き込むときにはまず MAR にメモリアドレスを設定し、引き続き、書き込み対象をメモリにして値を入力する。‘+’ キーと ‘-’ キーは MAR の値を 4 単位で増減するので、連続したアドレスに値を書き込むときには使用すると効率がよい。

## C.9 各部の仕様

### C.9.1 汎用の 32 ビットレジスタ

汎用の 32 ビットレジスタのテンプレート

```
module reg32(input          clock,    // クロック
             input          reset,    // リセット

             input [31:0]    in,      // 書き込みデータ
             input          ld,      // 書き込み制御信号

             output reg [31:0] out,    // 出力

             input          dbg_mode; // デバッグモード
             input [31:0]    dbg_in,  // デバッグモードの書き込みデータ
             input          dbg_ld);  // デバッグモードの書き込み制御信号

    ...
endmodule
```

**構成** PC, IR, A, B, C で用いられる汎用の 32 ビットレジスタ。

**動作** 以下の優先順位に従って処理を行う。

- reset が 0 なら内部の値を 32'b0 にする。
- dbg\_mode が 1 かつ dbg\_ld が 1 なら dbg\_in の値を書き込む。
- dbg\_mode が 0 かつ ld が 1 なら in の値を書き込む。

この記述は public/reg32.v にある。

## C.9.2 レジスタファイル

### レジスタファイルのテンプレート

```

module regfile(input          clock,    // クロック信号 (立ち上がりエッジ)
               input          reset,    // リセット信号 (0 でリセット)

               input [4:0]     rs1_addr, // RS1 のレジスタ番号
               input [4:0]     rs2_addr, // RS2 のレジスタ番号
               input [4:0]     rd_addr,  // RD のレジスタ番号

               input [31:0]     rd_in,   // RD に書き込むデータ
               input            rd_ld,   // RD の書き込み制御信号

               output [31:0]     rs1_out, // RS1 の出力
               output [31:0]     rs2_out, // RS2 の出力

               input            dbg_mode; // デバッグモード
               input [31:0]     dbg_in,  // デバッグモードの書き込みデータ
               input [4:0]     dbg_addr, // デバッグモードのレジスタ番号
               input            dbg_ld,  // デバッグモードの書き込み制御信号
               output [31:0]     dbg_out); // デバッグモードの出力

    ...
endmodule

```

**構成** 32 個の 32 ビットの汎用レジスタを持つ。ただし、 $R_0$  は読み出すと常に 0 を返し、書き込みを行っても値は変化しないダミーのレジスタである。

**動作** 以下の優先順位に従って処理を行う。

- `dbg_addr` で指定されたレジスタの値を `dbg_out` に出力する。ただし、`dbg_out` はレジスタではないので `assign` 文で作ること。
- `rs1_addr` で指定されたレジスタの値を `rs1_out` に出力する。ただし、`rs1_out` はレジスタではないので `assign` 文で作ること。
- `rs2_addr` で指定されたレジスタの値を `rs2_out` に出力する。ただし、`rs2_out` はレジスタではないので `assign` 文で作ること。
- `reset` が 0 なら全てのレジスタの値を 0 にする。
- `dbg_mode` が 1 かつ `dbg_ld` が 1 なら `dbg_addr` で指定されたレジスタに `dbg_in` の値を書き込む。
- `dbg_mode` が 0 かつ `rd_ld` が 1 なら `rd_addr` で指定されたレジスタに `rd_in` の値を書き込む。

この記述は `public/regfile.v` にある。



## C.9.3 STCONV

STCONV のテンプレート

```
module stconv(input [31:0] in, // 入力
              input [31:0] ir, // IR の値
              output [31:0] out); // 出力
    ...
endmodule
```

**構成** メモリに書き込む 32 ビットのデータを変換する組み合わせ回路。ストア命令 (sb, sh, sw) で用いられる。

**動作**

- sb 命令: in の下位 8 ビットの値を適切な位置にコピーする。例えば, 32'h0001 番地に書き込む場合には {16'b0, in[7:0], 8'b0} という風に in[7:0] を 8 ビット左にシフトする必要がある。ただし, 実際には mem\_wrbits で書き込む対象を指定するので, 書き込まない部分は 0 である必要はない。そこで, sb 命令の場合には書き込む番地に関わらず, {in[7:0], in[7:0], in[7:0], in[7:0]} (または {4{in[7:]}}) を用いればよい。
- sh 命令: in の下位 16 ビットの値を適切な位置にコピーする。sb 命令の場合と同様に考えること。
- sw 命令: in をそのまま out に入れればよい。

### C.9.4 LDCONV

#### LDCONV のテンプレート

```
module ldconv(input [31:0] in,      // 入力
              input [31:0] ir,      // IR の値
              input [1:0]  offset, // メモリアドレスの下位 2 ビット
              output [31:0] out);   // 出力
...
endmodule
```

**構成** メモリから読み出された 32 ビットのデータを変換する組み合わせ回路。ロード命令 (lb, lbu, lh, lhu, lw) で用いられる。

- 動作**
- lb 命令: 該当の番地を含む 4 バイトの値が in に入っているので、そこから該当のバイトデータを取り出す。さらに符号拡張を行う。
  - lbu 命令: 該当の番地を含む 4 バイトの値が in に入っているので、そこから該当のバイトデータを取り出す。ここでは上位ビットには 0 を入れる。
  - lh 命令: 該当の番地を含む 4 バイトの値が in に入っているので、そこから該当の 16 ビット (ハーフワード) データを取り出す。さらに符号拡張を行う。
  - lhu 命令: 該当の番地を含む 4 バイトの値が in に入っているので、そこから該当の 16 ビット (ハーフワード) データを取り出す。ここでは上位ビットには 0 を入れる。
  - lw 命令: in をそのまま out に入れればよい。

## C.9.5 ALU

## ALU のテンプレート

```

module alu(input [31:0] in1, // 入力 1
           input [31:0] in2, // 入力 2
           input [ 3:0] ctl, // 機能コントロール信号
           output [31:0] out); // 出力

    ...

endmodule

```

構成 純粋な組み合わせ回路として実装する。

動作 ● ALU の動作：ctl の値に従い，表 C.13 のように動作する。

表 C.13 ALU の動作

ctl	動作	備考
0000	$out \leftarrow in2$	LUI 用
0010	$out \leftarrow in1 == in2$	等価比較，BEQ 用
0011	$out \leftarrow in1 \neq in2$	非等価比較，BNE 用
0100	$out \leftarrow in1 < in2$	小なり比較，BLT, SLT 用
0101	$out \leftarrow in1 \geq in2$	大なり比較，BGE
0110	$out \leftarrow in1 < in2$	符号なし小なり比較，BLTU, SLTU 用
0111	$out \leftarrow in1 \geq in2$	符号なし大なり比較，BGEU 用
1000	$out \leftarrow in1 + in2$	ADD 用，分岐先アドレス計算用
1001	$out \leftarrow in1 - in2$	SUB 用
1010	$out \leftarrow in1 \oplus in2$	XOR 用
1011	$out \leftarrow in1 \vee in2$	OR 用
1100	$out \leftarrow in1 \wedge in2$	AND 用
1101	$out \leftarrow in1 \text{ shift left logical } in2$	SLL 用
1110	$out \leftarrow in1 \text{ shift right logical } in2$	SRL 用
1111	$out \leftarrow in1 \text{ shift right arithmetic } in2$	SRA 用

- $==, !=, <, \geq$  の結果は 32'b1 か 32'b0 になる。
- $\oplus$  はビットごとの排他的論理和 (XOR) を表す。Verilog-HDL の演算子は  $\wedge$ 。
- $\vee$  はビットごとの論理和 (OR) を表す。Verilog-HDL の演算子は  $|$ 。
- $\wedge$  はビットごとの論理積 (AND) を表す。Verilog-HDL の演算子は  $\&$ 。
- shift left logical は in1 の値を in2 の値の数だけ左にシフトする。最下位ビットには 0 が入る。
- shift right logical は in1 の値を in2 の値の数だけ右にシフトする。最上位ビットには 0 が入る。
- shift right arithmetic は in1 の値を in2 の値の数だけ右にシフトする。ただし，最上位ビットは昔の値のまま変更しない。これは元の数を 2 の補数表現の符号付き整数と見なした時に，右シフト動作が 2 で割ることと等価になるようにする工夫である。そのためこのシフトは

arithmetic(算術的) と呼ばれる .

この記述は `public/alu.v` にある .

## C.9.6 CSR

## CSR のテンプレート

```

module csr(input          clock,      // クロック
            input         reset,      // リセット
            input [11:0]  addr,       // アドレス
            input [31:0]  in,         // 入力
            input [1:0]   op,         // 操作命令
            output [31:0] out,        // 出力

            input         mie_in      // MSTATUS.MIE への入力
            input         mie_ld      // MSTATUS.MIE の書込み制御信号
            output        mie_out     // MSTATUS.MIE の出力

            input         mpie_in     // MSTATUS.MPIE への入力
            input         mpie_ld     // MSTATUS.MPIE の書込み制御信号
            output        mpie_out    // MSTATUS.MPIE の出力

            input         mtie_in     // MIE.MTIE への入力
            input         mtie_ld     // MIE.MTIE の書込み制御信号
            output        mtie_out    // MIE.MTIE の出力

            input [31:0]  mepc_in     // MEPC への入力
            input         mepc_ld     // MEPC の書込み制御信号
            output [31:0] mepc_out    // MEPC の出力

            input [31:0]  mcause_in   // MCAUSE への入力
            input         mcause_ld   // MCAUSE の書込み制御信号
            output [31:0] mcause_out  // MCAUSE の出力

            input         mtip_in     // MIP.MTIP への入力
            input         mtip_ld     // MIP.MTIP の書込み制御信号
            output        mtip_out    // MIP.MTIP の出力

);
    ...
endmodule

```

CSR は一種のレジスタファイルであるが、個々のレジスタが特殊な意味を持っている。そこで、命令実行で用いられるインターフェイスとは別に、割り込み処理で個別に参照されるインターフェイスの 2 種類を用意している。具体的には以下の信号線は CSR 命令実行中に用いられる。

- addr CSR レジスタのアドレスを指定する。

- in CSR 命令における入力値 .
- op CSR 命令の種類 . ここでは以下の符号化を用いるものとする .

表 C.14 op の符号化

op	意味
00	nop . なにもしない .
01	write . in の値をそのまま書き込む .
10	set . in の値との論理和を取る .
11	clear . in をビット単位で反転させた値との論理積を取る .

- out CSR 命令で指定されたレジスタの値 .

in は通常は  $R_{s1}$  で指定されたレジスタの値だが , CSRRWI のような即値系の CSR 命令の場合は rs1 の値を 5 ビットの即値とみなして上位 27 ビットに 0 を入れたものを使用することに注意 (ただしそれは CSR モジュールの仕事ではない) . op の値によっては in との論理演算が必要となるが , 単純な AND や OR 演算なので ALU を用いずに CSR モジュール内で処理する .

それ以外の信号線は XXX\_in , XXX\_ld , XXX\_out の 3 つ組となっており , それぞれ XXX\_in が入力 , XXX\_ld が書き込み制御信号 , XXX\_out が出力となっている . これは該当するレジスタの入力 , 書き込み制御信号 , 出力と直結する . CSR の各レジスタは CSR 系の命令以外にもハードウェア割り込みに関係して直接アクセスされる可能性があることに注意 .

KAPPA3-LIGHT では CSR を用いない .

### C.9.7 フェイズジェネレータ

フェイズジェネレータのテンプレート

```
module phasegen(input      clock,      // クロック
                input      reset,      // リセット
                input      run,        // run 信号
                input      step_phase, // フェイズ単位実行信号
                input      step_inst,   // n 命令単位実行信号
                output [3:0] cstate,    // フェイズ出力
                output      running);   // 実行中を示す信号

...

endmodule
```

フェイズジェネレータは正確にはコントローラの一部であるが、KAPPA3-LIGHT ではデバッグ用にフェイズ毎や命令毎に実行を停止させる機能を持たせるため、フェイズ遷移を行うモジュールを独立させている。その結果、コントローラ内部に記憶を持たない純粋な組み合わせ回路となっている。

cstate はフェイズを表す 4 ビットの信号線で以下のような符号化を行うものとする。

表 C.15 cstate の符号化

フェイズ	cstate
IF	4'b0001
DE	4'b0010
EX	4'b0100
WB	4'b1000

フェイズジェネレータが正しく動いている限り、cstate の値は上記の 4 つ以外にはならないはずであるが、安全のため、不正な値の場合には次のクロックで IF フェイズに遷移することが望ましい。

**構成** 4 ビットの phase 信号の値を保持するレジスタ (各 phase を reg 宣言すればよい) と、次のような内部状態を保持するためのレジスタを持つ。ここでいう内部状態とはフェイズとは無関係なので注意すること。この内部状態は 4 つあるので 4 状態を保持するためには最低 2 ビットのレジスタが必要である。どの状態をどの符号に割り当てるかは自由である。

表 C.16 フェイズジェネレータの内部状態

内部状態	意味
STOP	停止状態
RUN	通常の実行モード
STEP_INST	命令毎に停止するモード
STEP_PHASE	フェイズ毎に停止するモード

**動作** 以下の優先順位に従って処理を行う。

- reset が 0 のとき phase を IF にし、内部状態を Stop にする。

- 内部状態に従って以下の動作をする

STOP — run が 1 のとき RUN へ

- step\_inst が 1 のとき STEP\_INST へ
- step\_phase が 1 のとき STEP\_PHASE へ
- それ以外では状態は変化しない

RUN — run が 1 のときは次状態を STOP にする .

- それ以外は phase を 1 つ進ませ , 次状態は RUN のまま .

STEP\_INST — phase が WB の時は phase を IF にして状態を STOP にする .

- それ以外は phase を 1 つ進ませ , 次状態は STEP\_INST のまま .

STEP\_PHASE — phase を 1 つ進ませる .

次状態は必ず STOP

尚 , running 信号は内部状態が STOP 以外の時に 1 となる . 純粋な組み合わせ回路として作成すること .



## C.9.8 コントローラ

コントローラのテンプレート

```

module controller(input [3:0]  cstate,      // フェイズ信号
                  input [31:0] ir,         // IR レジスタの値
                  input [31:0] addr,       // メモリアドレス
                  input [31:0] alu_out,    // ALU の出力

                  output        pc_sel,    // PC の入力選択
                  output        pc_ld,     // PC の書き込み制御
                  output        mem_sel,    // メモリアドレスの入力選択
                  output        mem_read,  // メモリの読み込み制御
                  output        mem_write, // メモリの書き込み制御
                  output [3:0]  mem_wrbits, // メモリの書き込みビットマスク
                  output        ir_ld,     // IR レジスタの書き込み制御
                  output [4:0]  rs1_addr,  // RS1 アドレス
                  output [4:0]  rs2_addr,  // RS2 アドレス
                  output [4:0]  rd_addr,   // RD アドレス
                  output [1:0]  rd_sel,    // RD の入力選択
                  output        rd_ld,     // RD の書き込み制御
                  output        a_ld,      // A レジスタの書き込み制御
                  output        b_ld,      // B レジスタの書き込み制御
                  output        a_sel,     // ALU の入力 1 の入力選択
                  output        b_sel,     // ALU の入力 2 の入力選択
                  output [31:0] imm,       // 即値
                  output [3:0]  alu_ctl,   // ALU の機能コード
                  output        c_ld);     // C レジスタの書き込み制御

...

endmodule

```

**構成** このモジュールは組み合わせ回路として設計すること。

**動作** 他のモジュールを制御する信号を生成する。詳細は以下の通り。

PC の制御 (pc\_sel, pc\_ld) PC の値が変化するのは以下の場合。

- JAL 命令および JALR 命令でかつ cstate が WB。この場合には C レジスタの値を代入する。
- BEQ 命令などの条件分岐命令でかつ cstate が WB。この場合には分岐条件を満たした時だけ C レジスタの値を代入する。分岐条件の結果は alu\_out が持っている。
- 上記以外の命令でかつ cstate が WB。この場合には常に 4 を加算する。

メモリの制御 (mem\_sel, mem\_read, mem\_write, mem\_wrbits) メモリアドレスを指定するのは以下の場合。

- cstate が IF。この場合には PC の値を選ぶ。
- ロード命令およびストア命令で cstate が WB。この場合には C レジスタの値を選ぶ。

メモリの内容を読み出すのは以下の場合．

- ロード命令で `cstate` が WB．

メモリの内容が変化するのは以下の場合．

- ストア命令で `cstate` が WB．

メモリに書き込むビットマスクは以下のように設定する．RV32I では基本的に 1 語 (32 ビット=4 バイト) 単位でメモリアクセスを行うが，`sb` 命令と `sh` 命令の場合にはそれぞれ 8 ビット=1 バイト，16 ビット=2 バイト単位でアクセスする必要がある．そのため，4 バイトのうち，書き換えるバイトを指定するために `mem_wrbits` という信号線を用意する．これは 4 ビットの信号線でそれぞれのビットが 0~3 バイト目に対応している．例えば 0 バイト目のみ書き込む場合には 4'b0001 と指定する．4 バイトすべてに書き込む場合には 4'b1111 を用いる．

- `sb` 命令の場合，メモリアドレスの下位 2 ビットに応じて 0, 1, 2, 3 のどれか一つのビットのみ 1 とする．
- `sh` 命令の場合，メモリアドレスの下位 2 ビットが 2'00 か 2'10 かに応じて 0 バイトめと 1 バイトめか，2 バイトと 3 バイトめビットを 1 にする．
- `sw` 命令の場合，すべてのバイトに書き込むので 4'b1111 となる．

IR の制御 (`ir_ld`) IR の内容が変化するのは以下の場合．

- `cstate` が IF．

レジスタファイルの制御その 1(`rs1_addr`, `rs2_addr`, `rd_addr`) RV32I では  $r_{s1}$ ,  $r_{s2}$ ,  $r_d$  のフィールドが全ての命令形式で同一なのでこれをそのまま `rs1_addr`, `rs2_addr`, `rd_addr` に用いればよい．

レジスタファイルの制御その 2(`a_ld`, `b_ld`) レジスタファイルの内容を A レジスタ，B レジスタに読み出すのは以下の場合．

- `cstate` が DE．

命令によっては A レジスタや B レジスタの値を用いない場合があり，その場合にはここで読み出すことが無駄になるが，条件判断を行う論理回路を作るほうが無駄なので無条件に読み出すほうが論理回路は簡単になる．

レジスタファイルの制御その 3(`rd_sel`, `rd_ld`) レジスタファイルの内容が変化するのは以下の場合．

- 演算命令で `cstate` が WB．この場合は C レジスタの値を書き込む．
- ロード命令で `cstate` が WB．この場合はメモリの出力の値を書き込む．
- ジャンプ命令で `cstate` が WB．この場合は PC レジスタの値を書き込む．

詳細はフェイズ表を参照すること．

即値の生成 (`imm`) 命令形式に応じて以下の種類がある．

- `I_imm` : I-type の命令形式で用いられる即値．12 ビットの符号付き整数を 32 ビットの符号付き整数に符号拡張する．
- `S_imm` : S-type の命令形式で用いられる即値．12 ビットの符号付き整数を 32 ビットの符号付き整数に符号拡張する．`I_imm` との違いは即値のもととなるビット位置が異なる．
- `B_imm` : B-type の命令形式で用いられる即値．分岐先アドレスが奇数になることはないため，0 ビット目は常に 0 である．IR 中では 1 ビット目から 12 ビット目までの 12 ビットを指定する．その後 32 ビットの符号付き整数に符号拡張する．この形式は複雑なのでよく確認すること．
- `U_imm` : U-type の命令形式で用いられる即値．IR 中で指定された 20 ビットを上位 20 ビットに用いて下位 12 ビットを 0 とする．
- `J_imm` : J-type の命令形式で用いられる即値．分岐先アドレスが奇数になることはないため，

0 ビット目は常に 0 である。IR 中では 1 ビット目から 20 ビット目までの 20 ビットを指定する。その後 32 ビットの符号付き整数に符号拡張する。この形式は複雑なのでよく確認すること。

- 即値のシフト命令：大まかには I-type の命令だが、32 ビットの演算ではシフト量は最大で 32 なので I-type の即値フィールドのうち下位 5 ビットのみを用いる。上位のビットは `srl` 命令と `srla` 命令の区別に用いられる。

ALU の制御 (`a_sel`, `b_sel`, `alu_ctl`) `cstate` が EX の時、ALU は何らかの形で用いられている (フェイズ表 C.8 参照) ので、その内容に応じた機能コードを `alu_ctl` に出力する。同時に `a_sel`, `b_sel` にも適切な値を設定すること。さらに条件分岐命令では `cstate` が WB の時でも分岐条件の判断で ALU を用いる。

C レジスタの制御 (`cld`) C レジスタの内容が変化するのは以下の場合。

- `cstate` が EX。

### C.9.9 メモリ

#### メモリ

```
module memory(input      clock,          // クロック
               input [31:0] address,      // アドレス
               input      read,          // 読み出しイネーブル
               input      write,         // 書き込みイネーブル
               input [31:0] wrdata,       // 書き込みデータ
               input [3:0]  wrbits,       // 書き込みビットマスク
               output [31:0] rddata,      // 読み出しデータ

               input      dbg_mode,      // デバッグモード
               input [31:0] dbg_address,  // デバッグ用のアドレス
               input      dbg_read,      // デバッグ用の読み出しイネーブル
               input      dbg_write,     // デバッグ用の書き込みイネーブル
               input [31:0] dbg_in,      // デバッグ用の書き込みデータ
               output [31:0] dbg_out);   // デバッグ用の読み出しデータ

    ...
endmodule
```

この記述は public/memory.v にある . 内部で public/mem64dk.v をインスタンス化している . mem64kd.v の記述は Quartus 専用の特殊な記述である .

## C.9.10 KAPPA3-LIGHT コア

## KAPPA3-LIGHT コア

```

module kappa3_light_core(input  clock,                // クロック
                        input  clock2,               // clock を 2 分周したもの
                        input  reset,                // リセット
                        input  run,                  // 'run' 信号
                        input  step_phase,           // 'SP' 信号
                        input  step_inst,            // 'SI' 信号

                        input [31:0] dbg_in,          // デバッグ用書き込みデータ
                        input      dbg_pc_ld,        // PC のデバッグ用書き込みイ
ネーブル信号

                        input      dbg_ir_ld,        // IR のデバッグ用書き込みイ
ネーブル信号

                        input      dbg_reg_ld,       // REGFILE のデバッグ用書き込み
イネーブル信号

                        input [4:0]  dbg_reg_addr,   // REGFILE のデバッグ用アドレス
                        input      dbg_a_ld,        // A レジスタのデバッグ用書き込
み位ネーブル信号

                        input      dbg_b_ld,        // B レジスタのデバッグ用書き込
み位ネーブル信号

                        input      dbg_c_ld,        // C レジスタのデバッグ用書き込
み位ネーブル信号

                        input [31:0] dbg_mem_addr,   // デバッグ用のメモリアドレス
                        input      dbg_mem_read,    // デバッグ用のメモリの読み出し
信号

                        input      dbg_mem_write,   // デバッグ用のメモリの書き込み
信号

                        output [31:0] dbg_pc_out,    // PC のデバッグ出力
                        output [31:0] dbg_ir_out,    // IR のデバッグ出力
                        output [31:0] dbg_reg_out,    // REGFILE のデバッグ出力
                        output [31:0] dbg_a_out,     // A レジスタのデバッグ出力
                        output [31:0] dbg_b_out,     // B レジスタのデバッグ出力
                        output [31:0] dbg_c_out,     // C レジスタのデバッグ出力
                        output [31:0] dbg_mem_out    // メモリからの読み出しデータ

                        ...
endmodule

```

図 C.2 とほぼ同様の構成である。ただし、デバッグ用のインターフェイス信号 (先頭が dbg\_で始まる) が追加されている。書き込む値は共通で dbg\_in を用いる。書き込む対象は dbg\_xx\_ld で指定する。レジスタの

値は `dbg_xx_out` から取得する。ただし、メモリだけ読み出しが `dbg_mem_read` で書き込みが `dbg_mem_write` となっている。下位のモジュールは ALU、レジスタファイル、PC、IR、A、B、C、STCONV、LDCONV、フェイズジェネレータ、およびコントローラである。このうち、PC、IR、A、B、C は汎用の 32 ビットレジスタ `reg32` を用いる。CSR は含まれない。

`public/kappa3_light_core_dp.v` はこれらのうち、PC、IR、A、B、C、レジスタファイルおよびメモリをインスタンス化した記述である。ALU、STCONV、LDCONV、フェイズジェネレータおよびコントローラがないためプロセッサとしては動作しないが、内部の記憶素子をすべて含んでいるので後述のデバッガの動作確認に用いることができる。

なお、メモリのみ `clock` を用い、残りは `clock2` を用いている。これはメモリの読み出し、書き込みのタイミングを考慮したためである。

## C.9.11 KAPPA3-LIGHT 用トップモジュール

## KAPPA3-LIGHT 用トップモジュール

```

module kappa3_light(input      sys_clock, // システムクロック
                    input      reset,     // リセット
                    input      clock,     // CPU クロック
                    input      psw_a0,    // プッシュ SW-A0
                    input      psw_a1,    // プッシュ SW-A1
                    input      psw_a2,    // プッシュ SW-A2
                    input      psw_a3,    // プッシュ SW-A3
                    input      psw_a4,    // プッシュ SW-A4
                    input      psw_b0,    // プッシュ SW-B0
                    input      psw_b1,    // プッシュ SW-B1
                    input      psw_b2,    // プッシュ SW-B2
                    input      psw_b3,    // プッシュ SW-B3
                    input      psw_b4,    // プッシュ SW-B4
                    input      psw_c0,    // プッシュ SW-C0
                    input      psw_c1,    // プッシュ SW-C1
                    input      psw_c2,    // プッシュ SW-C2
                    input      psw_c3,    // プッシュ SW-C3
                    input      psw_c4,    // プッシュ SW-C4
                    input      psw_d0,    // プッシュ SW-D0
                    input      psw_d1,    // プッシュ SW-D1
                    input      psw_d2,    // プッシュ SW-D2
                    input      psw_d3,    // プッシュ SW-D3
                    input      psw_d4,    // プッシュ SW-D4
                    input [3:0] hex_a,    // ロータリー SW HEX_A
                    input [3:0] hex_b,    // ロータリー SW HEX_B
                    input [7:0] dip_a,    // DIP-SW DIP_A
                    input [7:0] dip_b,    // DIP-SW DIP_B
                    output [7:0] seg_x,    // MU500-RK のボード出力 (SEG_X)
                    output [3:0] sel_x,    // MU500-RK のボード出力 (SEL_X)
                    output [7:0] seg_y,    // MU500-RK のボード出力 (SEG_Y)
                    output [3:0] sel_y,    // MU500-RK のボード出力 (SEL_Y)
                    output [7:0] led_out, // MU500-RK のボード出力 (LED_OUT)
                    output [7:0] seg_a,    // MU500-7SEG のボード出力 (SEG_A)
                    output [7:0] seg_b,    // MU500-7SEG のボード出力 (SEG_B)
                    output [7:0] seg_c,    // MU500-7SEG のボード出力 (SEG_C)
                    output [7:0] seg_d,    // MU500-7SEG のボード出力 (SEG_D)
                    output [7:0] seg_e,    // MU500-7SEG のボード出力 (SEG_E)
                    output [7:0] seg_f,    // MU500-7SEG のボード出力 (SEG_F)
                    output [7:0] seg_g,    // MU500-7SEG のボード出力 (SEG_G)
                    output [7:0] seg_h,    // MU500-7SEG のボード出力 (SEG_H)
                    output [8:0] sel);     // MU500-7SEG のボード出力 (SEL)

```

...

KAPPA3-LIGHT コアと外部の入出力を接続する . public/kappa3\_light.v にある記述を用いること .



## 参考文献

- [1] 三菱電機マイコン機器ソフトウェア株式会社, “MU500-RX セット\_ユーザーズマニュアル Ver1.1.pdf”
- [2] 三菱電機マイコン機器ソフトウェア株式会社, “MU500-7SEG マニュアル Ver2.pdf”
- [3] 三菱電機マイコン機器ソフトウェア株式会社, “FPGA 設計ツール操作手順書 (RX 専用 QuartusII)\_V122.pdf”
- [4] 木村 真也, “Verilog-HDL 論理回路設計”, CQ 出版社, 2001.