DASM — DLX Assembler for SECONDS —

永見 康一[1]

1996/08/16 (Revision 1.2)

[1]e-mail:nagami@isl.ntt.jp, tel:0468-59-3599

更新履歴

Revision 1.2 初版

目次

はじめに

本書は、 DLX プロセッサの命令セットに対応したアセンブラ dasm 用の、アセンブリ言語マニュアルです。 dasm はアセンブリ言語で記述されたプログラム・ファイルを入力とし、プログラムのメモリ・イメージを表現する SECONDS スクリプトを出力します。また、プログラム内のアドレス情報 (ラベルとアドレスとの対応関係) を出力することができます。これにより、 DLX プロセッサの SECONDS シミュレーション環境を用いたプログラムの実行シミュレーションを容易に行なうことができます。

dasm の起動形式は次のようになっています。

% dasm *[options] filename*

filename には、プログラム・ファイルを指定します。 *[options]* には、次の各オプションを指定できます。

オプション	内容	省略値
-a *address-file*	アドレス情報ファイルを指定します。	標準エラー出力
-o *output-file*	出力ファイルを指定します。	標準出力
-h	起動形式のヘルプを表示します。	

1 アセンブリ・プログラムの概要

まず最初に、簡単なアセンブリ・プログラムを示し、各構文要素の概要を説明します。

```
行   ソース
 1:              .data 0xffff00 = "/SYS/M00"[0xffff00] *1
 2: darea: *2    .space 0
 3: from:        .word 0x000000ff *3
 4: to:          .space 4
 5: end:         .word 0
 6:
 7:              .text 0 = "/SYS/M00"[0] *4
 8:              lhi $gp, (darea>>16) *5                     #
 9:              .  $gp |= (darea & 0xffff) *6               # $gp = darea *7
10:              .  $v0 *8 = (%w)$gp[(%o)from]               # $v0 = *from
11:              .  nop                                      # load delay slot
12:              .  (to - darea) *9 [$gp] = (%b)$v0          # *to = (char)$v0
13:              .  (to - darea)[$gp] = (%h)$v0              # *to = (short)$v0
14:              .  (to - darea)[$gp] = (%w)$v0              # *to = (int)$v0
15:
16:              .  $v0 = 1                                  # $v0 = 1
17:              .  $gp[(%o)end] = $v0                       # *to = $v0
18:
19: inf:         j inf
20:              .  nop                                      # slot
```

***1** データ・セグメントの開始 1 - 5 行で一つのデータセグメントを定義しています。データ・セグメントでは、プログラムで使用するデータの定義や領域の確保を記述します。 → 2 節

***2** ラベル データ・セグメントにおけるデータ定義や領域確保のアセンブラ指令、あるいはテキスト・セグメントにおける命令ニーモニックや拡張表記による命令の記述行にはラベルをつけることができます。

ラベルに使える文字はアルファベット、数字、'_', '', '$' です。ただし先頭の文字に数字を用いることはできません。そして、ラベルには一つのアドレス値が対応します。対応するアドレスとは、データ定義・領域確保のアセンブラ指令の場合、そのデータ領域の先頭アドレスであり、命令記述行の場合その命令が格納されるアドレスです。なお、大きさ 0 の領域確保を用いることにより、同一のアドレスに複数のラベルをつけることもできます (2 行目)。これは、例えばデータセグメントの開始アドレスに特に ラベルを付けたい場合などに有効です。また、ラベルは定数式の中で使うことができ、その評価値は

ラベル ラベルをつけた位置の絶対アドレス値。

(%o) ラベル ラベルをつけた位置の、セグメント内オフセット。

(%s) ラベル ラベルをつけたデータ領域の大きさ (バイト単位)。テキスト・セグメント内のラベルの場合は常に 0。

のように、プリフィックス ((%o), (%s)) によってさまざまに変化します。 → 4 節

***3** アセンブラ指令 データ・セグメント内でデータ定義や領域確保を記述するためには、アセンブラ指令を用います。 → 2 節

***4** テキスト・セグメントの開始 7 - 20 行で一つのテキストセグメントを定義しています。テキスト・セグメントでは、プログラムのコード部分を命令ニーモニックや拡張表記などを用いて記述します。 → 2,5 節

***5** 命令ニーモニック DLX の命令と一対一対応する表記です。 → 5 節

***6** 拡張表記 命令ニーモニックを、より簡潔かつ直観的に記述するための表記です。 → 5 節

***7** コメント '#' から行末まではコメントとなり、 dasm はこの部分を無視します。

***8** レジスタ DLX が備える 32 本のレジスタを指定する表記です。レジスタ番号に基づく表記と、 MIPS の慣習にならったニックネーム表記があります。 →3 節

***9** 定数式 アセンブラ指令、命令ニーモニックおよび拡張表記において、定数を指定する部分には C 言語に準じた算術演算式が使えます。 → 4節

2 アセンブラ指令 (assembler directive)

assembler directive		
directive		**function**
`.data`	a_d = "*mem-name*"[o]	begin data segment from a_d
`.byte`	$b_1, b_2, \cdots, b_n$	define and allocate byte sequence
`.half`	$h_1, h_2, \cdots, h_n$	define and allocate half word sequence
`.word`	$w_1, w_2, \cdots, w_n$	define and allocate word sequence
`.space`	n	allocate n byte area
`.align`	n	align deta address with 2^n byte boundary
`.text`	a_t = "*mem-name*"[o]	begin text segment from a_t

`.data` a_d, `.text` a_t 指令により、それぞれデータ・セグメントとコード (テキスト)・セグメントの開始を記述できます。引数の a_d, a_t はともに定数式で、これにより各セグメントの開始アドレスを指定します。 a_t はワード境界、すなわち 4 の倍数である必要があります。また、データ・セグメント、コード・セグメントは任意の順序で複数記述することができますが、アドレス 0 から始まるコード・セグメントが必須であり、プログラムの実行はこのセグメントを起点とします。セグメントの内容は、 *mem-name* で表される SFL のメモリ・ファシリティの o 番地から順に格納されていきます。したがって、 *mem-name* はバイト単位の `mem` ファシリティである必要があります。

.byte $b_1, b_2, \cdots, b_n$ 指令は、データ・セグメントにおいて n 個のバイト列データを定義します。引数が複数の場合‘,’によって区切ります。各定数はバイト値として評価され、現在のデータ・セグメントに順次配置されていきます。.half 指令、. word 指令も同様で、それぞれハーフ・ワード列データ、ワード列データを定義・配置します。ただしこれらの場合、各データは、それぞれハーフ・ワード境界およびワード境界に配置されます。これを、データの自動整列化と呼びます。

.space n 指令は、未定義データ領域を確保します。引数 n は、確保する領域の大きさをバイト単位で表します。

.align n 指令は、データの整列化を行ないます。すなわち、後に続く .byte, .half, .word, .space などの領域確保指令の開始アドレスを、強制的に 2^n バイト境界に移動します。例外として $n = 0$ の場合、以降そのデータ・セグメントが終るまで、データの自動整列化機能が抑制されます。

3 レジスタ (register)

register			register		
#	**alias**	**what for**	#	**alias**	**what for**
0	$r0, $0	always zero	16	$r16, $s0	temporary (to be saved)
1	$r1, $at	reserved	17	$r17, $s1	temporary (to be saved)
2	$r2, $v0	return value	18	$r18, $s2	temporary (to be saved)
3	$r3, $v1	return value	19	$r19, $s3	temporary (to be saved)
4	$r4, $a0	1st argument	20	$r20, $s4	temporary (to be saved)
5	$r5, $a1	2nd argument	21	$r21, $s5	temporary (to be saved)
6	$r6, $a2	3rd argument	22	$r22, $s6	temporary (to be saved)
7	$r7, $a3	4th argument	23	$r23, $s7	temporary (to be saved)
8	$r8, $t0	temporary (not to be saved)	24	$r24, $t8	temporary (not to be saved)
9	$r9, $t1	temporary (not to be saved)	25	$r25, $t9	temporary (not to be saved)
10	$r10, $t2	temporary (not to be saved)	26	$r26, $k0	reserved (for kernel)
11	$r11, $t3	temporary (not to be saved)	27	$r27, $k1	reserved (for kernel)
12	$r12, $t4	temporary (not to be saved)	28	$r28, $gp	global pointer
13	$r13, $t5	temporary (not to be saved)	29	$r29, $sp	stack pointer
14	$r14, $t6	temporary (not to be saved)	30	$r30, $fp	frame pointer
15	$r15, $t7	temporary (not to be saved)	31	$r31, $ra	return address

DLX には 32 本の整数レジスタがあり、0 ∼ 31 までの番号が付いています。0 番レジスタは特殊なレジスタで、常に値 0 として参照されます。その他の 31 本のレジスタは通常の 32 ビット整数レジスタとして用いることができます。

アセンブリ・プログラム内では $r0, $r1, ··· , $r31 として記述できますが、MIPS にならって各レジスタをニックネームで記述することもできます。表の alias の欄を参照して下さい。このニックネームは、MIPS が各レジスタに割り当てている役割に基づくものです。各役割については what for の欄を参照して下さい。

4 定数式 (constant expression)

アセンブリ・プログラム内の

1. アセンブラ指令 の引数
2. 命令記述における即値オペランド

の箇所において、定数式を記述することができます。

定数式の構文は、次の BNF 表記で定義されます。終端記号は 終端記号、非終端記号は 非終端記号 のように表記しており、一部正規表現を用いている部分もあります。また、式の評価は常に 32 ビットの数として行なわれます。

```
定数式      ::= 定数項
              | ( 定数式 )
              | 単項演算子 定数式
              | 定数式 二項演算子 定数式
定数項      ::= ラベル
              | (%o) ラベル
              | (%s) ラベル
              | 16 進定数
              | 10 進定数
              | 8 進定数
              | 2 進定数
ラベル      ::= [a-zA-Z_$] [a-zA-Z_$0-9]*
16 進定数   ::= 0x [0-9a-f]+
10 進定数   ::= 0 | [1-9] [0-9]*
8 進定数    ::= 0o [0-7]+
2 進定数    ::= 0b [01]+
単項演算子  ::= - | ~ | ++ | [++] | (++) | -- | [--] | (--)
二項演算子  ::= * | (*) | [*] | / | (/) | [/] | % | (%) | [%]
              | + | (+) | [+] | - | (-) | [-]
              | << | (>>) | (>>)
              | == | (!=)
              | < | (<) | [<] | > | (>) | [>]
              | <= | (<=) | [<=] | >= | (>=) | [>=]
              | & | ^ | |
```

演算子の優先順位

演算子	演算	優先度	結合
-	負符号	0	右
~	ビット反転	0	右
++ (++)	符号つき +1	0	右
[++]	符号なし +1	0	右
-- (--)	符号つき -1	0	右
[--]	符号なし -1	0	右
* (*)	符号つき乗算	1	左
[*]	符号なし乗算	1	左
/ (/)	符号つき除算	1	左
[/]	符号なし除算	1	左
% (%)	符号つき剰余	1	左
[%]	符号なし剰余	1	左
+ (+)	符号つき加算	2	左
[+]	符号なし加算	2	左
- (-)	符号つき減算	2	左
[-]	符号なし減算	2	左

演算子の優先順位

演算子	演算	優先度	結合
<<	左シフト	3	左
>>	論理右シフト	3	左
(>>)	算術右シフト	3	左
< (<)	符号つき小なり	4	左
[<]	符号なし小なり	4	左
> (>)	符号つき大なり	4	左
[>]	符号なし大なり	4	左
<= (<=)	符号つき以下	4	左
[<=]	符号なし以下	4	左
>= (>=)	符号つき以上	4	左
[>=]	符号なし以上	4	左
==	等値	5	左
!=	不等値	5	左
&	論理積	6	左
^	排他的論理和	7	左
|	論理和	8	左

5 命令セット (instruction set)

この節では、 dasm がサポートする DLX の命令セットを列挙します。命令セットは

1. 分岐命令
2. アクセス命令
3. 比較命令
4. 算術論理演算命令
5. シフト命令

の各グループに分類し、表にまとめました。表の各欄は次のような意味を持ちます。

instruction アセンブリ・プログラムで記述するための命令ニーモニックです。オペランド部分は次の表記を用いています。

表記	意味
$r_1, r_2, \cdots$	レジスタを指定します (→ 3 節)。
i	定数式を記述できます。最終的な結果は 16 ビットにキャストされます。
i_a	アドレスを表す定数式を記述します。

alias 各命令ニーモニックを、より簡潔で直観的な表記で記述するための拡張表記です。命令ニーモニックの代替表記として、アセンブリ・プログラム中で記述することができます。拡張表記はすべて . で始まることに注意して下さい。

description 各命令の機能の簡潔な説明です。

behavior 各命令の機能を形式的に定義します。次のような表記を用いています。

表記	意味
$a \Leftarrow b$	a に b の値を転送する
$\circ v$	v の値を符号つき整数とみなした数
$\bullet v$	v の値を符号なし整数とみなした数
$\langle v^{\circ n} \rangle$	v の値を、 n ビットに符号拡張した n ビットの数
$\langle v^{\bullet n} \rangle$	v の値を、 n ビットにゼロ拡張した n ビットの数
$M\left[a :_{\times} n\right]$	メモリアドレス a から始まる n バイトを連結した、 $8 \times n$ ビットの数
$v_{\lvert h..l \rvert}$	v の値の第 h ビットから第 l ビットまでを切り出した数
$a\#b$	a と b をビット連結した数
pc	プログラムカウンタ

なお、 dasm がサポートする命令セットは、原則として DLX の原典である文献 [1] に記されている命令セットから、浮動小数点命令を除いたものですが、種々の理由により一部変更を加えた命令や追加された命令があります。これらの変更については変更点を明記します。

5.1 分岐 (jump and branch)

jump and branch			
instruction	**alias**	**description**	**behavior**
`j` i_a	`.goto` i_a	relative jump	$pc \Leftarrow i_a$
`jr` r_1 `$r31`	`.goto` r_1 `.return`	absolute jump	$pc \Leftarrow r_1$
`jal` i_a	`.call` i_a	relative jump and link	$\$r31 \Leftarrow pc + 4; pc \Leftarrow i_a$
`jalr` r_1	`.call` r_1	absolute jump and link	$\$r31 \Leftarrow pc + 4; pc \Leftarrow r_1$
`beqz` r_1 , i_a		branch if equal zero	if$(r_1 = 0) pc \Leftarrow i_a$
`bnez` r_1 , i_a		branch if equal zero	if$(r_1 \neq 0) pc \Leftarrow i_a$
~~`bfpt`~~			*(not implemented)*
~~`bfpf`~~			*(not implemented)*
~~`rfe`~~		return from exception	*(not implemented)*
~~`trap`~~			*(not implemented)*

`j`，`jal` 命令のニーモニック形式は absolute jump ですが、命令コードの形式では即値として i_a と $pc+4$ との距離を指定するので、実際は relative jump です。また、 `beqz`，`bnez` 命令も同様に relative jump です。そのため、ジャンプの相対距離に関して、

`j`，`jal` 命令 $(pc + 4 - 2^{25} \leq i_a < pc + 4 + 2^{25})$
`beqz`，`bnez` 命令 $(pc + 4 - 2^{15} \leq i_a < pc + 4 + 2^{15})$

という制約がつきます。 dasm はこの制約を検査します。

変更事項 このグループで変更・追加された命令はありません。

5.2 アクセス (load and store)

load and store			
instruction	**alias**	**description**	**behavior**
`lb` r_1, $i(r_2)$	$.r_1$ = (%b) $i[r_2]$ $.r_1$ = (%b) $r_2[i]$	load signed byte	$r_1 \Leftarrow \left\langle M\left[\left\langle i^{\circ 32}\right\rangle + r_2 :_{\times} 1\right]^{\circ 32}\right\rangle$
`lbu` r_1, $i(r_2)$	$.r_1$ = (%bu) $i[r_2]$ $.r_1$ = (%bu) $r_2[i]$	load unsigned byte	$r_1 \Leftarrow \left\langle M\left[\left\langle i^{\circ 32}\right\rangle + r_2 :_{\times} 1\right]^{\bullet 32}\right\rangle$
`lh` r_1, $i(r_2)$	$.r_1$ = (%h) $i[r_2]$ $.r_1$ = (%h) $r_2[i]$	load signed half word	$r_1 \Leftarrow \left\langle M\left[\left\langle i^{\circ 32}\right\rangle + r_2 :_{\times} 2\right]^{\circ 32}\right\rangle$
`lhu` r_1, $i(r_2)$	$.r_1$ = (%hu) $i[r_2]$ $.r_1$ = (%hu) $r_2[i]$	load unsigned half word	$r_1 \Leftarrow \left\langle M\left[\left\langle i^{\circ 32}\right\rangle + r_2 :_{\times} 2\right]^{\bullet 32}\right\rangle$
`lw` r_1, $i(r_2)$	$.r_1$ = (%w) $i[r_2]$ $.r_1$ = (%w) $r_2[i]$	load word	$r_1 \Leftarrow M\left[\left\langle i^{\circ 32}\right\rangle + r_2 :_{\times} 4\right]$
~~lf~~		load SP float	*(not implemented)*
~~ld~~		load DP float	*(not implemented)*
`sb` $i(r_1)$, r_2	$.i[r_1]$ = (%b) r_2 $.r_1[i]$ = (%b) r_2	store byte	$M\left[\left\langle i^{\circ 32}\right\rangle + r_1 :_{\times} 1\right] \Leftarrow r_{2\|7..0\|}$
`sh` $i(r_1)$, r_2	$.i[r_1]$ = (%h) r_2 $.r_1[i]$ = (%h) r_2	store half word	$M\left[\left\langle i^{\circ 32}\right\rangle + r_1 :_{\times} 2\right] \Leftarrow r_{2\|15..0\|}$
`sw` $i(r_1)$, r_2	$.i[r_1]$ = (%w) r_2 $.r_1[i]$ = (%w) r_2 $.i[r_1]$ = r_2 $.r_1[i]$ = r_2	store word	$M\left[\left\langle i^{\circ 32}\right\rangle + r_1 :_{\times} 4\right] \Leftarrow r_2$

変更事項　このグループで変更・追加された命令はありません。

5.3 比較 (compare operation)

compare operations			
instruction	**alias**	**description**	**behavior**
`seqi` r_1, r_2, i	`.`r_1 `=` r_2 `==` i	if equal	$r_1 \Leftarrow r_2 = i$
`snei` r_1, r_2, i	`.`r_1 `=` r_2 `!=` i	if not equal	$r_1 \Leftarrow r_2 \neq i$
`slti` r_1, r_2, i	`.`r_1 `=` r_2 `(<)` i `.`r_1 `=` r_2 `<` i	if signed less than	$r_1 \Leftarrow \circ r_2 < \circ \left\langle i^{\circ 32} \right\rangle$
`sltui` r_1, r_2, i	`.`r_1 `=` r_2 `[<]` i	if unsigned less than	$r_1 \Leftarrow \bullet r_2 < \bullet \left\langle i^{\circ 32} \right\rangle$
`sgti` r_1, r_2, i	`.`r_1 `=` r_2 `(>)` i `.`r_1 `=` r_2 `>` i	if signed greater than	$r_1 \Leftarrow \circ r_2 > \circ \left\langle i^{\circ 32} \right\rangle$
`sgtui` r_1, r_2, i	`.`r_1 `=` r_2 `[>]` i	if unsigned greater than	$r_1 \Leftarrow \bullet r_2 > \bullet \left\langle i^{\circ 32} \right\rangle$
`slei` r_1, r_2, i	`.`r_1 `=` r_2 `(<=)` i `.`r_1 `=` r_2 `<=` i	if signed less than or equal	$r_1 \Leftarrow \circ r_2 \leq \circ \left\langle i^{\circ 32} \right\rangle$
`sleui` r_1, r_2, i	`.`r_1 `=` r_2 `[<=]` i	if unsigned less than or equal	$r_1 \Leftarrow \bullet r_2 \leq \bullet \left\langle i^{\circ 32} \right\rangle$
`sgei` r_1, r_2, i	`.`r_1 `=` r_2 `(>=)` i `.`r_1 `=` r_2 `>=` i	if signed greater than or equal	$r_1 \Leftarrow \circ r_2 \geq \circ \left\langle i^{\circ 32} \right\rangle$
`sgeui` r_1, r_2, i	`.`r_1 `=` r_2 `[>=]` i	if unsigned greater than or equal	$r_1 \Leftarrow \bullet r_2 \geq \bullet \left\langle i^{\circ 32} \right\rangle$
`seq` r_1, r_2, r_3	`.`r_1 `=` r_2 `==` r_3	if equal	$r_1 \Leftarrow r_2 = r_3$
`sne` r_1, r_2, r_3	`.`r_1 `=` r_2 `!=` r_3	if not equal	$r_1 \Leftarrow r_2 \neq r_3$
`slt` r_1, r_2, r_3	`.`r_1 `=` r_2 `(<)` r_3 `.`r_1 `=` r_2 `<` r_3	if signed less than	$r_1 \Leftarrow \circ r_2 < \circ r_3$
`sltu` r_1, r_2, r_3	`.`r_1 `=` r_2 `[<]` r_3	if unsigned less than	$r_1 \Leftarrow \bullet r_2 < \bullet r_3$
`sgt` r_1, r_2, r_3	`.`r_1 `=` r_2 `(>)` r_3 `.`r_1 `=` r_2 `>` r_3	if signed greater than	$r_1 \Leftarrow \circ r_2 > \circ r_3$
`sgtu` r_1, r_2, r_3	`.`r_1 `=` r_2 `[>]` r_3	if unsigned greater than	$r_1 \Leftarrow \bullet r_2 > \bullet r_3$
`sle` r_1, r_2, r_3	`.`r_1 `=` r_2 `(<=)` r_3 `.`r_1 `=` r_2 `<=` r_3	if signed less than or equal	$r_1 \Leftarrow \circ r_2 \leq \circ r_3$
`sleu` r_1, r_2, r_3	`.`r_1 `=` r_2 `[<=]` r_3	if unsigned less than or equal	$r_1 \Leftarrow \bullet r_2 \leq \bullet r_3$
`sge` r_1, r_2, r_3	`.`r_1 `=` r_2 `(>=)` r_3 `.`r_1 `=` r_2 `>=` r_3	if signed greater than or equal	$r_1 \Leftarrow \circ r_2 \geq \circ r_3$
`sgeu` r_1, r_2, r_3	`.`r_1 `=` r_2 `[>=]` r_3	if unsigned greater than or equal	$r_1 \Leftarrow \bullet r_2 \geq \bullet r_3$

レジスタ - 即値形式のすべての命令において、16 ビットの即値が 符号拡張 されることに注意して下さい。

変更事項 オリジナルの DLX では、符号つき整数の比較命令 `s**`, `s**i` (`**` は `eq`, `ne`, `lt`, `gt`, `le`, `ge` のいずれか) のみ用意されていますが、新たに符号なし整数の比較命令 `s**u`, `s**ui` を追加しました。

5.4 算術論理 (ALU operation)

arithmetic and logical operations (register - immediate)				
instruction		**alias**	**description**	**behavior**
addi	r_1, r_2, i	.r_1 = r_2 (+) i	signed addition	$r_1 \Leftarrow \circ r_2 + \circ \left\langle i^{\circ 32} \right\rangle$
	r_1, r_2, i	.r_1 = r_2 + i		
	r_1, r_1, i	.r_1 (+)= i		
	r_1, r_1, i	.r_1 += i		
	r_1, r_1, 1	.(++)r_1		
	r_1, r_1, 1	.++r_1		
addui	r_1, r_2, i	.r_1 = r_2 [+] i	unsigned addition	$r_1 \Leftarrow \bullet r_2 + \bullet \left\langle i^{\circ 32} \right\rangle$
	r_1, r_1, i	.r_1 [+]= i		
	r_1, r_1, 1	.[++]r_1		
subi	r_1, r_2, i	.r_1 = r_2 (-) i	signed subtraction	$r_1 \Leftarrow \circ r_2 - \circ \left\langle i^{\circ 32} \right\rangle$
	r_1, r_2, i	.r_1 = r_2 - i		
	r_1, r_1, i	.r_1 (-)= i		
	r_1, r_1, i	.r_1 -= i		
	r_1, r_1, 1	.(--)r_1		
	r_1, r_1, 1	.--r_1		
subui	r_1, r_2, i	.r_1 = r_2 [-] i	unsigned subtruction	$r_1 \Leftarrow \bullet r_2 - \bullet \left\langle i^{\circ 32} \right\rangle$
	r_1, r_1, i	.r_1 [-]= i		
	r_1, r_1, 1	.[--]r_1		
andi	r_1, r_2, i	.r_1 = r_2 & i	logical AND	$r_1 \Leftarrow r_2 \& \left\langle i^{\bullet 32} \right\rangle$
	r_1, r_1, i	.r_1 &= i		
ori	r_1, r_2, i	.r_1 = r_2 | i	logical OR	$r_1 \Leftarrow r_2 \vert \left\langle i^{\bullet 32} \right\rangle$
	r_1, r_1, i	.r_1 |= i		
xori	r_1, r_2, i	.r_1 = r_2 ^ i	logical XOR	$r_1 \Leftarrow r_2 \hat{} \left\langle i^{\bullet 32} \right\rangle$
	r_1, r_1, i	.r_1 ^= i		
lhi	r_1, i		load high immediate	$r_1 \Leftarrow i_{\vert 15..0 \vert} \# 0x0000$

変更事項　オリジナルの DLX では、レジスタ - 即値形式の論理演算命令 andi, xori, ori において、即値は 符号拡張 と定義されていますが、これを ゼロ拡張 に変更しました。理由は次の通りです。

文献 [1] には、lhi, addui の 2 命令によって、32 ビット定数をレジスタに格納できることを示唆する記述がありますが、 addui 命令の即値は符号拡張と定義されているため、実際は不可能です (ビット 15 が 1 である定数の場合)。そこで、論理演算命令の即値をゼロ拡張と定義することにより、 lhi, ori の 2 命令で 32 ビット定数のレジスタ格納を可能にしました。この即値拡張形式は、 DLX の源泉である MIPS の命令定義に準じています。

arithmetic and logical operations (register - register)				
instruction		**alias**	**description**	**behavior**
add	r_1, r_2, r_3	.r_1 = r_2 (+) r_3	signed addition	$r_1 \Leftarrow \circ r_2 + \circ r_3$
	r_1, r_2, r_3	.r_1 = r_2 + r_3		
	r_1, r_1, r_2	.r_1 (+)= r_2		
	r_1, r_1, r_2	.r_1 += r_2		
	r_1, r_2, $r0	.r_1 = r_2		
	$r0, $r0, $r0	.nop		
addu	r_1, r_2, r_3	.r_1 = r_2 [+] r_3	unsigned addition	$r_1 \Leftarrow \bullet r_2 + \bullet r_3$
	r_1, r_1, r_2	.r_1 [+]= r_2		
sub	r_1, r_2, r_3	.r_1 = r_2 (-) r_3	signed subtraction	$r_1 \Leftarrow \circ r_2 - \circ r_3$
	r_1, r_2, r_3	.r_1 = r_2 - r_3		
	r_1, r_1, r_2	.r_1 (-)= r_2		
	r_1, r_1, r_2	.r_1 -= r_2		
subu	r_1, r_2, r_3	.r_1 = r_2 [-] r_3	unsigned subtruction	$r_1 \Leftarrow \bullet r_2 - \bullet r_3$
	r_1, r_1, r_2	.r_1 [-]= r_2		
mult	r_1, r_2, r_3	.r_1 = r_2 (*) r_3	signed multiplication	$r_1 \Leftarrow \circ r_2 \times \circ r_3$
	r_1, r_1, r_2	.r_1 *= r_2		
	r_1, r_1, r_2	.r_1 (*)= r_2		
	r_1, r_1, r_2	.r_1 *= r_2		
multu	r_1, r_2, r_3	.r_1 = r_2 [*] r_3	unsigned multiplication	$r_1 \Leftarrow \bullet r_2 \times \bullet r_3$
	r_1, r_1, r_2	.r_1 [*]= r_2		
~~div~~			signed division	*(not implemented)*
~~divu~~			unsigned division	*(not implemented)*
and	r_1, r_2, r_3	.r_1 = r_2 & r_3	logical AND	$r_1 \Leftarrow r_2 \& r_3$
	r_1, r_1, r_2	.r_1 &= r_2		
or	r_1, r_2, r_3	.r_1 = $r_2 \mid r_3$	logical OR	$r_1 \Leftarrow r_2 \vert r_3$
	r_1, r_1, r_2	.r_1 $\vert$= r_2		
xor	r_1, r_2, r_3	.r_1 = r_2 ^ r_3	logical XOR	$r_1 \Leftarrow r_2$ ^ r_3
	r_1, r_1, r_2	.r_1 ^= r_2		

変更事項　整数乗算命令 mult，multu は、オペランドとして浮動小数点レジスタを指定すると定義されていますが、現時点では整数レジスタを指定するものと定義しています。また、文献 [1] では、記述箇所によってレジスタ - 即値形式の整数乗算命令 multi，multui が定義されているように記述されていますが、この命令は用意されていません。

5.5 シフト (shift operation)

shift operations			
instruction	**alias**	**description**	**behavior**
`slli` r_1, r_2, i r_1, r_1, i	.r_1 = r_2 << i .r_1 <<= i	shift logical left	$r_1 \Leftarrow \bullet r_2 \ll \bullet \left\langle i^{\circ 32} \right\rangle$
`srli` r_1, r_2, i r_1, r_1, i	.r_1 = r_2 >> i .r_1 >>= i	shift logical right	$r_1 \Leftarrow \bullet r_2 \gg \bullet \left\langle i^{\circ 32} \right\rangle$
`srai` r_1, r_2, i r_1, r_1, i	.r_1 = r_2 (>>) i .r_1 (>>)= i	shift arithmetic right	$r_1 \Leftarrow \circ r_2 \gg \bullet \left\langle i^{\circ 32} \right\rangle$
`sll` r_1, r_2, r_3 r_1, r_1, r_2	.r_1 = r_2 << r_3 .r_1 <<= r_2	shift logical left	$r_1 \Leftarrow \bullet r_2 \ll \bullet r_3$
`srl` r_1, r_2, r_3 r_1, r_1, r_2	.r_1 = r_2 >> r_3 .r_1 >>= r_2	shift logical right	$r_1 \Leftarrow \bullet r_2 \gg \bullet r_3$
`sra` r_1, r_2, r_3 r_1, r_1, r_2	.r_1 = r_2 (>>) r_3 .r_1 (>>)= r_2	shift arithmetic right	$r_1 \Leftarrow \circ r_2 \gg \bullet r_3$

変更事項　このグループで変更・追加された命令はありません。

参考文献

[1] John L.Hennessy and David A.Patterson; 富田 眞治 ・ 村上 和彰 ・ 新實治男 訳: “コンピュータ・アーキテクチャ: 設計・実現・評価の定量的アプローチ”, 日経 BP 社, 1992

[2] David A.Patterson and John L.Hennessy and; 成田光彰 訳: “コンピュータの構成と設計 [下]”, 日経 BP 社, 1996

A アセンブリ・プログラム例

A.1 mem.asm

メモリのロードとストアを行なうだけの単純なプログラムです。領域 from に格納されているワード値を $v0 にロードし、領域 to に、順にバイト、ハーフ・ワード、ワードとしてストアしていきます。ストアが終了すると、領域 end に格納されている値を 0x00000001 に変えて、無限ループに入ります。

```
        .data 0x00ffff00 = "/SYS/M00"[0x00ffff00]
darea:  .space 0
from:   .word 0x000000ff
to:     .space 4
end:    .word 0

        .text 0 = "/SYS/M00"[0]
        lhi $gp, (darea>>16)           #
        . $gp |= (darea & 0xffff)      # $gp = darea
        . $v0 = (%w)$gp[(%o)from]      # $v0 = *from
        . nop                          # load delay slot
        . $gp[(%o)to] = (%b)$v0        # *to = (unsigned char)$v0
        . $gp[(%o)to] = (%h)$v0        # *to = (unsigned short)$v0
        . $gp[(%o)to] = (%w)$v0        # *to = (unsigned int)$v0

        . $v0 = 1                      # $v0 = 1
        . $gp[(%o)end] = $v0           # *to = $v0

inf:    . goto inf                     # infinite loop
        . nop                          # slot
```

アセンブル結果

```
memset /SYS/M00 16776960\
X00 X00 X00 Xff
memset /SYS/M00 16776968\
X00 X00 X00 X00
memset /SYS/M00 0\
X3f X9c X00 Xff\
X37 X9c Xff X00\
X8f X82 X00 X00\
X00 X00 X00 X20\
Xa3 X82 X00 X04\
Xa7 X82 X00 X04\
Xaf X82 X00 X04\
X34 X02 X00 X01\
Xaf X82 X00 X08\
X0b Xff Xff Xfc\
X00 X00 X00 X20
rpt_add from  "from [00ffff00]: %2X %2X %2X %2X\n"\
/SYS/M00@Xffff00 /SYS/M00@Xffff01 /SYS/M00@Xffff02 /SYS/M00@Xffff03
rpt_add to    "to   [00ffff04]: %2X %2X %2X %2X\n"\
/SYS/M00@Xffff04 /SYS/M00@Xffff05 /SYS/M00@Xffff06 /SYS/M00@Xffff07
rpt_add end   "end  [00ffff08]: %2X %2X %2X %2X\n"\
/SYS/M00@Xffff08 /SYS/M00@Xffff09 /SYS/M00@Xffff0a /SYS/M00@Xffff0b
```