MIPS算術論理演算命令とレジスタ

- 算術論理演算命令は３オペランド
  - add, sub, mult, div, and, or,...
- オペランドの順序は固定
  - 結果は第１オペランドに格納される
  - `add $1,$2,$3 ; $1=$2+$3`
- 転送系は２オペランド
  - 例 `lw $s1,100($2)`
- 条件分岐系は３オペランド
  - 例 `beq $1,$2,25`
- 絶対ジャンプ系は１オペランド
  - 例 `j 2500`

- レジスタ
  - CPUの内部にあるデータの格納場所
  - メモリと比べて２桁以上高速に参照できる
  - 限られた数しかない（MIPSは32個）
- MIPSのレジスタ名とレジスタ番号

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

レジスタを使ったアセンブリ・プログラムの例

- Cステートメント（C言語のプログラム，コード）

```
f = ( g + h ) - ( i + j );
```

  - 変数とレジスタの割当　f：$s0, g:$s1, h:$s2, i:$s3, j:$s4
    - 一時格納場所として$t0と$t1を使う．
    - t は temporal （一時的）の t を意味する

- MIPSコード（アセンブリ，プログラム）

```
add $t0,$s1,$s2 # $t0 = f+g
add $t1,$s3,$s4 # $t1 = i+j
sub $s0,$t0,$t1 # f = $t0 - $t1
```

- 配列はどのように実現するか？

問題のパターン

C 言語のプログラム

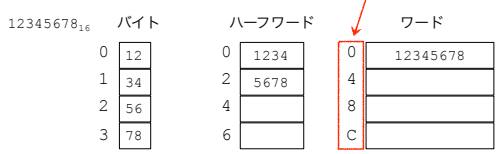
変数とレジスタの割当て

MIPSコード
適切なコメントを書くこと
正しいコメントは評価する．

MIPSのメモリ

- 巨大な１次元の配列
  - 4番地の内容 Memory[4]
- MIPSはバイトアドレッシング
  - 2^{32}バイトあるいは2^{30}ワード

ワード単位でメモリを見れば，
4番地ごとに見える



データ転送命令

- データ転送命令
  - メモリとレジスタの間でデータを転送する命令
- ロード命令とストア命令
  - オペランドはメモリアドレス
  - `100($s3)`
    - レジスタ$s3の内容に100を加えた値をアドレスとする
- ロード命令　`lw $s0,100($s3)`
  - メモリからレジスタへデータを転送する命令
- ストア命令　`sw $s0,100($s3)`
  - レジスタからメモリへデータを転送する命令

- `lw`と`sw`の使い方
  - レジスタ`$s3`：`100`
- `100`番地の内容を`$t0`に格納する
  - `lw $t0,0($s3)# 100+0`
- `104`番地の内容を`$t0`に格納する
  - `lw $t0,4($s3)# 100+4`
- `$t0`の内容を`100`番地に格納する
  - `sw $t0,0($s3)# 100+0`
- `$t0`の内容を`104`番地に格納する
  - `sw $t0,4($s3)# 100+4`

C言語からアセンブリ言語

- Cステートメント　a = b + c
- 整数型変数a，b，cはメモリ上にあるとする
  - aは100番地，bは104番地，cは108番地
- レジスタ$s3には100が格納されている．
- レジスタ$zeroは0が格納されている．

- MIPSコード（解答例1，○）

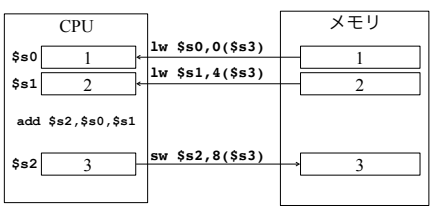
```
lw $s0,4($s3)    # $s0 = b
lw $s1,8($s3)    # $s1 = c
add $s2,$s0,$s1 # $s2= b+c
sw $s2,0($s3)    # a = $s2
```

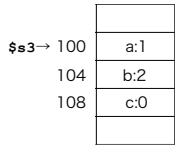
- MIPSコード（解答例2，△）

```
lw $s0,104($zero) # $s0 = b
lw $s1,108($zero) # $s1 = c
add $s2,$s0,$s1   # $s2= b+c
sw $s2,100($zero) # a = $s2
```

ロード・ストアアーキテクチャ

- データをメモリからレジスタにロードする．
- レジスタを使って計算する
- 計算結果をレジスタからメモリにストアする





配列とload命令とstore命令

配列A[]

■ A[]のベースアドレス$s3

$s3→	A[0]
+4	A[1]
+8	A[2]
+12	A[3]
+16	A[4]
+20	A[5]
+24	A[6]
+28	A[7]
+32	**A[8]**
+36	A[9]
+40	A[10]
+44	A[11]
+48	**A[12]**

load命令

■ Cステートメント

```
g = h + A[8];
```

▶ g：$s1，h：$s2

■ MIPSコード

```
lw $t0,32($s3)
add $s1,$s2,$t0
```

store命令

■ Cステートメント

```
A[12]=h+A[8];
```

▶ h：$s2

■ MIPSコード

```
lw $t0,32($s3)
add $t0,$s2,$t0
sw $t0,48($s3)
```

配列のインデックスが変数の場合

■ Cステートメント

```
g = h + A[i];
```

▶ 配列A[] のベースアドレス：$s3

▶ g:$s1，h:$s2，i:$s4

■ 配列A[i]のアドレス&A[i]を計算する

▶ &A[i] は &A[0]+4*i である．

■ iを4倍する方法

▶ iを4回加算する i+i+i+i

▶ iの２倍を２回加算する (i+i)+(i+i)

```
add $t1,$s4,$s4 # $t1=i+i=2*i
add $t1,$t1,$t1 # $t1=2*i+2*i
```

■ MIPSコード

```
add $t1,$s4,$s4 # $t1=2*i
add $t1,$t1,$t1 # $t1=4*i
add $t1,$t1,$s3 # $t1=&A[0]+4*i
lw $t0,0($t1)   # $t0=A[i]
add $s1,$s2,$t0
```

■ ４倍　教科書ではシフトを使う　i<<2

■ ５倍

```
add $t1,$s4,$s4 # ２倍
add $t1,$t1,$t1 # ４倍
add $t1,$t1,$s4 # ５倍
add $t1,$t1,$t1 # １０倍
```

即値（そくち，immediate）のオペランド

■ C言語のプログラムで使われる小さな定数

```
i = i + 1 ;
```

▶ i:$s3

定数の表

$s1→	1
	2
	3
	4
	…

■ アセンブリ言語に変換

▶ 定数の表を使う方法

```
lw $t0, 8($s1)
add $s3,$s3,$t0 # $s3=$s3+3
```

▶ 定数の表を用意しないといけない．

▶ 小さな定数はよく使われるので，その度にlw命令を使うのは遅くなる．

■ 命令セットの設計方針

▶ 一般的な場合を高速化せよ．

▶ 小さければ小さい程になる．

■ 即値命令（immediate）

```
addi $s3,$s3,1  # i=i+1
addi $s3,$s3,-1 # i=i-1
```

▶ subiはないよ！

■ 即値オペランド

▶ 16ビットの符号付整数

▶ 値の範囲： -2^{15}〜$2^{15}-1$

▶ -32,768 〜 32,767

符号拡張

- 即値命令（immediate）
  `addi $s3,$s3,1  # i=i+1`
  - レジスタは32ビット
  - 即値オペランドは16ビット
- 32ビットと16ビットの加算
  - 16ビットの整数を32ビットに変換する
    - 符号拡張
- 符号拡張
  - 符号ビットの写しで拡張する
- 8ビットを符号拡張して16ビットに変換する
  - (00001111)→
    (0000000000001111)
  - (10000010)→
    (1111111110000010)
- ビット数が変わっても，数ならば，正数は正数，負数は負数に変換しないといけないので，極めて自然な考え方である．
- 数でないので，論理命令は符号拡張しない．

アセンブリ言語から機械語への変換

- アセンブリ言語
  `add $t0, $s1, $s2`
- レジスタ名をレジスタ番号に変換する
  `add $8, $17, $18`
  - `$t0：8, $s1：17, $s2：18`
- `add`命令のOPコードを調べる
  - add　0=(000000)$_2$
  - func　32=(100000)$_2$
- 3つのレジスタ・オペランド
  - rd: destination 結果を格納する
  - rs: source 計算に使うレジスタ
  - rt: t is next to s. 計算に使うレジスタ
- 機械語
  - 32ビット長
  - 6つのフィールド

6	5	5	5	5	6
op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

000000 10001 10010 01000 0000 100000

- 0000 0010 0011 0010 0100 0000 0010 0000=(0232402)$_{16}$

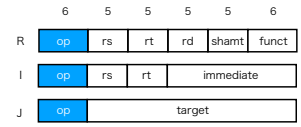
アセンブリ言語から機械語への変換

- アセンブリ言語
  `add $t0, $s1, $s2`
- レジスタ名をレジスタ番号に変換する
  `add $8, $17, $18`
  - `$t0：8, $s1：17, $s2：18`
- `add`命令のOPコードを調べる
  - add　0=(000000)$_2$
  - func　32=(100000)$_2$
- 機械語
  - 32ビット長
  - 6つのフィールド

op	rs	rt	rd	shamt	funct
op	rs	rt	rd	shamt	funct
0	17	18	8	0	32
000000	10001	10010	01000	0000	100000

MIPS命令セット

■ 3つの形式（R，I，J）



■ op　6ビットのオペコード

■ rs　第１ソースオペランドのレジスタの指定

■ rt　第２ソースオペランドのレジスタの指定

■ rd　デスティネーションオペランドのレジスタの指定

■ immediate　16ビットのイミディエト
分岐ディスプレイスメント
アドレスディスプレイスメント

■ target　26ビットの無条件分岐ターゲットアドレス（j命令）

■ shamt　5ビットのシフト量

■ funct　6ビットの機能フィールド

MIPS命令の符号化

命令	形式	op	rs	rt	rd	shamt	funct	address
add	R	0	レジスタ	レジスタ	レジスタ	0	32	適用せず
sub	R	0	レジスタ	レジスタ	レジスタ	0	34	適用せず
addi	I	8	レジスタ	レジスタ	適用せず	適用せず	適用せず	定数
lw	I	35	レジスタ	レジスタ	適用せず	適用せず	適用せず	アドレス
sw	I	43	レジスタ	レジスタ	適用せず	適用せず	適用せず	アドレス

■ レジスタ（5ビット）　0から31までのレジスタ番号

■ アドレス（16ビット）

■ 適用せず　該当命令形式にない

■ R形式について

- add命令とsub命令のopフィールドの値は共に0で同じである．
- functが32ならばadd，34ならばsubである．

C言語からアセンブリ言語，アセンブリ言語から機械語

■ Cステートメント

```
A[300]=h+A[300]
```

- 配列A[]のベースアドレス：$t1
- h：$s2

■ アセンブリ言語（レジスタ名）

```
lw $t0,1200($t1)
add $t0,$s2,$t0
sw $t0,1200($t1)
```

- $t1：9，$t0：8，$s2：18

■ アセンブリ言語（レジスタ番号）

```
lw $8,1200($9)
add $8,$18,$8
sw $8,1200($9)
```

■ アセンブリ言語から機械語

op	rs	rt	rd	shamt	funct
			address		
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

■ 機械語を2進数に変換

op	rs	rt	rd	shamt	funct
			address		
100011	01001	01000	0000010010110000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000010010110000		

■ 16進数に変換

8d2804b0

02484020

ad2804b0

論理演算とMIPS命令

論理演算	Cの演算子	Javaの演算子	MIPS命令
左シフト	<<	<<	sll
右シフト	>>	>>>	srl
ビット単位のAND	&	&	and,andi
ビット単位のOR	|	|	or,ori
ビット単位のNOT	~	~	nor

■ sll : shift left logical 論理左シフト

■ srl : shift right logical 論理右シフト

■ nor : not or

■ javaの “>>” は算術右シフトである.

■ andi/ori の i は immediate を意味する

シフト演算

左シフト

■ 1ビット左シフト 2倍

■ 2ビット左シフト 4倍

■ nビット左シフト 2^n倍

```
  00110010
0← 01100100 ←0
0← 11001000 ←0
1← 10010000 ←0
1← 00100000 ←0
0← 01000000 ←0
0← 10000000 ←0
1← 00000000 ←0
0← 00000000 ←0
```

右シフト

■ 1ビット右シフト 半分？

▶ 半分にはならない

```
  00110010
0→ 00011001 →0
0→ 00001100 →1
0→ 00000110 →0
0→ 00000011 →0
0→ 00000001 →1
0→ 00000000 →1
0→ 00000000 →0
0→ 00000000 →0
```

使い方

■ $s4の4倍を計算する方法

▶ 2ビット左シフト

```
sll $t1,$s4,2
```

ー 1命令で速い

▶ 2倍＋2倍

```
add $t1,$s4,$s4
add $t1,$t1,$t1
```

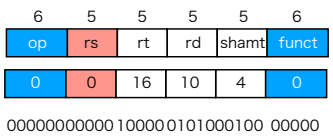
MIPSの論理演算命令

■ R形式 `sll $t2,$s0,4`

`sll $10,$16,4`

▶ shamt:shift amount

▶ $(00105100)_{16}$



00000000000 10000 01010 00100 00000

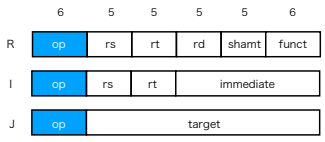
■ R形式 `and $t0,$t1,$t2`

■ I形式 `andi $t0,$t1,3328`$_{10}$

■ R形式 `or $t0,$t1,$t2`

■ I形式 `ori $t0,$t1,15360`$_{10}$

■ R形式 `nor $t0,$t1,$t3`



nor命令によるnot演算

- `not $s0 = nor $s0,$so,$zero`
- `nor $s0,$zero,$s1 # $s0=!$s1`
  `nor $s0,$s1,$zero # $s0=!$s1`
- NOR演算

X	Y	X nor Y
0	0	1
0	1	0
1	0	0
1	1	0

- Yがゼロなら，not X = X nor 0

X	Y	X nor Y
0	0	1
1	0	0

- Xでもゼロなら，not Y = 0 nor Y

X	Y	X nor Y
0	0	1
0	1	0

AND演算の使い方（1）

- ワード中の特定のビットを取り出す

X	Y	X and Y	備考
0	0	0	0になる
0	1	0	Xの値
1	0	0	0になる
1	1	1	Xの値

- `$t2`の下から３バイト目を取り出す．
- マスクパターン　$t1=(00000F00)16
  - 残したいビットに該当するビットを1にする
- `and $t0,$t2,$t2`

```
$t2  0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0000 1111 0000 0000
$t0  0000 0000 0000 0000 0000 1101 0000 0000
```

- `srl $t0,$t0,8`

```
$t0  0000 0000 0000 0000 0000 0000 0000 1101
```

AND演算の使い方（2）

- 特定のビットを0にする

X	Y	X and Y	備考
0	0	0	0になる
0	1	0	Xの値
1	0	0	0になる
1	1	1	Xの値

- マスクパターン　$t1=(000000FF)16
  - 0にしたいビットの該当するビットを0にする
    それ以外のビットは１にする
- `and $t0,$t2,$t2`

```
$t2  1010 1100 0101 1111 0011 1001 1100 1101
$t1  0000 0000 0000 0000 0000 0000 1111 1111
$t0  0000 0000 0000 0000 0000 0000 1100 1101
```

OR演算の使い方

■ 特定のビットを1にする

X	Y	X or Y	備考
0	0	0	Xの値
0	1	1	1になる
1	0	1	Xの値
1	1	1	1になる

■ パターン　$t1=(000000FF)16

▶ 1にしたいビットの該当するビットを1にする
それ以外のビットは0にする

■ `or $t0,$t2,$t2`

```
$t2  1010 1100 0101 1111 0011 1001 1100 1101
$t1  0000 0000 0000 0000 0000 0000 1111 1111
$t0  1010 1100 0101 1111 0011 1001 1111 1111
```

論理演算の使い方

■ 1ワードは4バイト，1文字は1バイト

■ "ABC"から1文字`C`を取り出し，小文字の`c`に変換する

```
$$t2="ABC"      0100 0001 0100 0010 0100 0011 0000 0000
8ビット右シフト  0000 0000 0100 0001 0100 0010 0100 0011
AND演算         0000 0000 0000 0000 0000 0000 1111 1111
                0000 0000 0000 0000 0000 0000 0100 0011
OR演算          0000 0000 0000 0000 0000 0000 0010 0000
大文字Cから小文字c 0000 0000 0000 0000 0000 0000 0110 0011
```

ASCII文字コード

文字	コード 10進	16進	文字	コード 10進	16進	文字	コード 10進	16進	文字	コード 10進	16進	文字	コード 10進	16進	文字	コード 10進	16進	文字	コード 10進	16進
NUL	0	0x00	DLE	16	0x10	SP	32	0x20	0	48	0x30	@	64	0x40	P	80	0x50	`	96	0x60	p	112	0x70
SOH	1	0x01	DC1	17	0x11	!	33	0x21	1	49	0x31	A	65	0x41	Q	81	0x51	a	97	0x61	q	113	0x71
STX	2	0x02	DC2	18	0x12	"	34	0x22	2	50	0x32	B	66	0x42	R	82	0x52	b	98	0x62	r	114	0x72
ETX	3	0x03	DC3	19	0x13	#	35	0x23	3	51	0x33	C	67	0x43	S	83	0x53	c	99	0x63	s	115	0x73
EOT	4	0x04	DC4	20	0x14	$	36	0x24	4	52	0x34	D	68	0x44	T	84	0x54	d	100	0x64	t	116	0x74
ENQ	5	0x05	NAK	21	0x15	%	37	0x25	5	53	0x35	E	69	0x45	U	85	0x55	e	101	0x65	u	117	0x75
ACK	6	0x06	SYN	22	0x16	&	38	0x26	6	54	0x36	F	70	0x46	V	86	0x56	f	102	0x66	v	118	0x76
BEL	7	0x07	ETB	23	0x17	'	39	0x27	7	55	0x37	G	71	0x47	W	87	0x57	g	103	0x67	w	119	0x77
BS	8	0x08	CAN	24	0x18	(	40	0x28	8	56	0x38	H	72	0x48	X	88	0x58	h	104	0x68	x	120	0x78
HT	9	0x09	EM	25	0x19	)	41	0x29	9	57	0x39	I	73	0x49	Y	89	0x59	i	105	0x69	y	121	0x79
NL*	10	0x0a	SUB	26	0x1a	*	42	0x2a	:	58	0x3a	J	74	0x4a	Z	90	0x5a	j	106	0x6a	z	122	0x7a
VT	11	0x0b	ESC	27	0x1b	+	43	0x2b	;	59	0x3b	K	75	0x4b	[	91	0x5b	k	107	0x6b	{	123	0x7b
NP	12	0x0c	FS	28	0x1c	,	44	0x2c	<	60	0x3c	L	76	0x4c	\	92	0x5c	l	108	0x6c	|	124	0x7c
CR	13	0x0d	GS	29	0x1d	-	45	0x2d	=	61	0x3d	M	77	0x4d	]	93	0x5d	m	109	0x6d	}	125	0x7d
SO	14	0x0e	RS	30	0x1e	.	46	0x2e	>	62	0x3e	N	78	0x4e	^	94	0x5e	n	110	0x6e	~	126	0x7e
SI	15	0x0f	US	31	0x1f	/	47	0x2f	?	63	0x3f	O	79	0x4f	_	95	0x5f	o	111	0x6f	DEL	127	0x7f

条件分岐命令 conditional branch

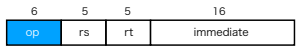
`beq $t1,$t2,L1`

- `$t1`と`$t2`が等しければ`L1`に分岐し，そうでなければ次の命令を実行する．
- `($t1-$t2=0)`ならば`L1`に分岐する．

`bne $t1,$t2,L1`

- `$t1`と`$t2`が等しくなければ`L1`に分岐し，そうでなければ次の命令を実行する．
- `($t1-$t2≠0)`ならば`L1`に分岐する．

`beq/bne`はI形式，相対アドレス



- beqのopコード　$8_{10}=(001000)_2$

```
if(R[rs]==R[rt])
PC=PC+4+BranchAddr
```

- bneのopコード　$9_{10}=(001001)_2$

```
if(R[rs]!=R[rt])
PC=PC+4+BranchAddr
```

- BranchAddr=
  `{14{immediate[15]},immediate,2'b0}`
  - 4倍，`immediate<<2`

if-else文

- Cステートメント

```
if(i == j ) f = g + h ;
else f = g - h ;
```

  - `f-j:$s0-$s4`

- MIPSコード（`bne`を使う）

```
      bne $s3,$s4,Else
      add $s0,$s1,$s2
      j Exit  # Exitへジャンプ
Else: sub $s0,$s1,$s2
Exit:
```

- `j`命令　ジャンプ命令
  - 無条件分岐　unconditional branch

- MIPSコード（`beq`を使う）

```
      beq $s3,$s4,Then
      sub $s0,$s1,$s2
      j Exit  # Exitへジャンプ
Then: add $s0,$s1,$s2
Exit:
```

While文のコンパイル

- Cステートメント

```
while ( save[i]==k)
    i += 1;
```

  - 配列`save[]`のベースアドレス：`$s6`
  - `f-j:$s0-$s4`

- Cステートメント
  - `while`文から`if`文と`goto`文

```
Loop:if(save[i]!=k) goto Exit;
     i=i+1;
     goto Loop
Exit:
```

- MIPSコード

```
Loop: sll $t1,$s3,2
      add $t1,$t1,$s6
      lw $t0,0($t1)
      bne $t0,$s5,Exit
      addi $s3,$s3,1
      j Loop
Exit:
```

- MIPSコード（最適化）

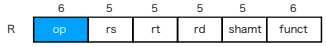
```
      beq $zero,$zero,Entry
Loop: addi $s3,$s3,1
Entry:sll $t1,$s3,2
      add $t1,$t1,$s6
      lw $t0,0($t1)
      beq $t0,$s5,Loop
Exit:
```

slt(set on less than) 命令

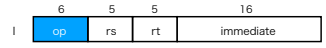
■ slt $t0,$s3,$s4
- $s3<$s4 のとき，$t0に１を設定
- そうでないとき，$t0に０を設定
  $s3 >= $s4のとき，$t0に０を設定

■ slti $t0,$s2,10
- $s2<10 のとき，$t0に１を設定
- そうでないとき，$t0に０を設定
  $s2 >= 10のとき，$t0に０を設定

■ slt
- R形式　0/2a
- R[rd]=(R[rs]<R[rt])? 1: 0



■ slti
- I 形式　a
- R[rd]=(R[rs]<SignExtImm)? 1: 0



整数値の条件判定

■ sltとbeq/bneの組合せで，すべての条件判定（６通り）を生成することができる．
- $zeroは値０のレジスタ．

実際のコンパイラでは$t0ではなくて$atが使われる．

```
■ beq $s1,$s2,L1    # $s1  ==  $s2 のときL1
■ bne $s1,$s2,L1    # $s1  !=  $s2 のときL1
■ slt $t0,$s1,$s2
  bne $t0,$zero,L1 # $s1  <   $s2 のときL1
■ slt $t0,$s1,$s2          否定          bne → beq
  beq $t0,$zero,L1 # $s1  >=  $s2 のときL1
■ slt $t0,$s2,$s1
  bne $t0,$zero,L1 # $s1  >   $s2 のときL1
■ slt $t0,$s2,$s1          否定          bne → beq
  beq $t0,$zero,L1 # $s1  =<  $s2 のときL1
```

$s1と$s2を入れ替えた

jr(jump register)命令

■ jr $t0
- $t0の内容のアドレスにジャンプする．

■ $s0は，０から3の値をとる．
- ジャンプ・アドレス表のアドレス：$t4

```
sll $t1,$s0,2
add $t1,$t1,$t4
lw $t0,0($t1)
jr $t0
```

ジャンプテーブル

$t4 →	L0 アドレス
$t4+4 →	L1 アドレス
$t4+8 →	L2 アドレス
$t4+12 →	L3 アドレス

■ ジャンプ先のコード

```
L0:add $s0,$s3,$s4
   j Exit
L1:add $s0,$s1,$s2
   j Exit
L2:sub $s0,$s1,$s2
   j Exit
L3:sub $s0,$s3,$s4
Exit:
```

switch-case文（1）

■ 次のCコードをMIPSコードに変換せよ．

- f–k：$s0–$s5

```
switch (k) {
   case 0: f = i + j;
           break;
   case 1: f = g + h;
           break;
   case 2: f = g - h;
           break;
   case 3: f = i - j;
           break;
}
```

■ case文に対応するMIPSコード

- case文はラベルに対応するので，case0からcase3を，それぞれラベルL0からL3に対応づける．
- kに応じてL0からL3にジャンプすれば良い

```
L0:add $s0,$s3,$s4
   j Exit
L1:add $s0,$s1,$s2
   j Exit
L2:sub $s0,$s1,$s2
   j Exit
L3:sub $s0,$s3,$s4
Exit:
```

switch-case文（2）

■ switch文に対応するMIPSコード

```
    slt $t3,$s5,$zero  # Test if k < 0
    bne $t3,$zero,Exit
    slti $t3,$s5,4     # Test if k < 4
    beq $t3,$zero,Exit # if k >=4,goto Exit
    sll $t1,$s5,2
    addi $t1,$t1,$t4
    lw $t0,0($t1)
    jr $t0
L0:add $s0,$s3,$s4
   j Exit
L1:add $s0,$s1,$s2
   j Exit
L2:sub $s0,$s1,$s2
   j Exit
L3:sub $s0,$s3,$s4
Exit:
```

ジャンプテーブル

$t4 →	L0 アドレス
$t4+4 →	L1 アドレス
$t4+8 →	L2 アドレス
$t4+12 →	L3 アドレス

関数（手続き）呼び出し

```
main{
  int x;
  …
  x =leaf_example(1,2,3,4);
  …
}
int leaf_example( int g, int h, int i, int j)
{
  int f;
  f = ( g + h ) - ( i + j );
  return f;
}
```

1．実引数をレジスタに設定
2．関数に制御を移す
3．関数に必要なメモリを確保
4．関数の実行
5．返り値を設定
6．制御を戻す

関数呼び出しの手順

■ 6つの手順

① 引数をレジスタ（$a0〜$a3）に置く

② 関数に制御を移す （jal命令，$ra）

③ 関数の実行に必要なメモリをスタック上に確保する

④ 関数の本体を実行する

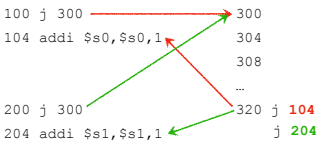
⑤ 返り値（関数値）をレジスタ（$v0）に置く

⑥ 制御を呼出し元に戻す（jr $ra #returnに相当）

■ レジスタの割当て

- $a0〜$a3：実引数（arguments）
- $v0〜$v1：返り値(values for results and expression evaluation)
- $ra：戻りアドレスアドレス（制御を戻すアドレス） (return address)

j命令では関数呼び出しに対応できない



■関数の開始アドレスは300

■100番地から関数を呼び出したとき，戻り番地は104番地になる．

- 320番地は　j 104　となる．

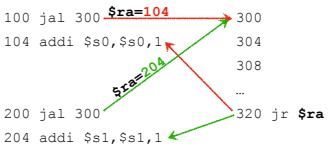
■200番地から関数を呼び出したとき，戻り番地は204番地になる．

- 320番地は　j 204　となる．

■関数を呼び出す番地で，戻り番地が異なるので，j命令は使えない．

- jr命令を使えばよいが，戻り番地をどのようにレジスタに設定するか？

jal命令



■**jal**命令 jump and link

- リンク　戻り番地を$raに設定する．

■100番地から関数を呼び出すと $ra=104

■200番地から関数を呼び出すと $ra=204

■制御を戻るには，jr $ra とすればよい．