第2章 命令：コンピュータの言葉(2)

大阪大学 大学院 情報科学研究科
今井 正治

E-mail: arch-2014@vlsilab.ics.es.osaka-u.ac.jp

講義内容

- 条件判定用の命令
- コンピュータ・ハードウェア内での手続きのサポート
- 人との情報交換
- 32ビットの即値およびアドレスに対するMIPSのアドレッシング方式
- 並列処理と命令：同期

条件分岐命令

- 等しいときに分岐
  - 命令：branch on equal
  - 記法：`beq $s1, $s2, L`
  - 動作：`if ( $s1 = $s2 ) go to L`
- 等しくないときに分岐
  - 命令：branch on not equal
  - 記法：`bne $s1, $s2, L`
  - 動作：`if ( $s1 != $s2 ) go to L`

if-then-else文のコンパイル

- Cの文

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

- MIPSのアセンブリ言語のコード

```
      bne  $s3, $s4, Else
      add  $s0, $s1, $s2
      j    Exit
Else: sub  $s0, $s1, $s2
Exit:
```

レジスタの割当て
$s0 ⇔ f
$s1 ⇔ g
$s2 ⇔ h
$s3 ⇔ i
$s4 ⇔ j

whileループのコンパイル

- Cの文

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

- MIPSのアセンブリ言語のコード

```
Loop: sll  $t1, $s3, 2     # i*4
      add  $t1, $t1, $s6   # save[i]のアドレス
      lw   $t0, 0($t1)     # save[i]の値
      bne  $t0, $s5, Exit  # save[i]≠kなら終了
      addi $s3, $s3, 1     # i += 1
      j    Loop
Exit:
```


条件分岐補助命令(1)

- より小さいかの判定(レジスタの内容の比較)
  - 命令: set on less than
  - 記法: `slt $s1, $s2, $s3`
  - 動作:

```
if ( $s2 < $s3 )
        $s1 = 1;
    else
        $s1 = 0;
```

  - 注: beq, bne 命令と組み合せて使用

条件分岐補助命令(2)

- より小さいかの判定(即値との比較)
  - 命令: set on less than immediate
  - 記法: `slt $s1, $s2, imm`
  - 動作:

```
if ( $s2 < imm )
        $s1 = 1;
    else
        $s1 = 0;
```

  - 注: beq, bne 命令と組み合せて使用

条件分岐補助命令(3)

- より小さいかの判定(符号なし数の比較)
  - 命令: set on less than unsigned
  - 記法: `sltu $s1, $s2, $s3`
  - 動作:

```
if ( $s2 < $s3 )
        $s1 = 1;
    else
        $s1 = 0;
```

  - 注: 後続の beq, bne 命令と組み合せて使用

符号付き数と符号なし数の比較

- $s0 = -1_{10}
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$
  を符号なし数とみなすと $4{,}294{,}967{,}295_{10}$
- $s1 = 1_{10}
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$
- `slt  $t0, $s0, $s1` # compare signed
  - `$t0 = 1`
- `sltu $t1, $s0, $s1` # compare unsigned
  - `$t1 = 0`

配列の境界チェック

- レジスタの割当て
  - `$s1`: 配列のインデックス
  - `$t2`: 配列の大きさ(`>0`)
- 配列のインデックスの検査
  - `$s1`≥`$t2` または `$s1<0` の場合，
    インデックス境界違反が発生
  - 検査方法

```
sltu $t0, $s1, $t2
beq  $t0, $zero, IndexOutOfBounds
```


case文(switch文)のサポート

- ジャンプ・アドレス表(jump address table)から該当する処理のアドレスをレジスタにロードし、そのレジスタの値を用いてジャンプする
- ジャンプ・レジスタ(jump register)命令
  - レジスタで指定されるアドレスに無条件でジャンプ

```
lw    $t0, 0($s3)
jr    $t0
```


講義内容(1)

- 条件判定用の命令
- コンピュータ・ハードウェア内での手続きのサポート
- 人との情報交換
- 32ビットの即値およびアドレスに対するMIPSのアドレッシング方式
- 並列処理と命令:同期

手続き(procedure)呼び出し

- 手続き呼び出しの手順
  1. 呼び出される手続き(callee)からアクセスできる場所にパラメータを置く
  2. 手続きに制御を渡す
  3. 手続きの実行に必要なメモリ資源を確保する
  4. 必要が処理を実行する
  5. 呼び出し側(caller)からアクセスできる場所に結果を置く
  6. 制御を手続きを呼び出した位置(呼び出し命令の次の命令)に戻す

パラメータと結果の値の受渡し

- パラメータ数が少ない場合(4個以下)
  - 引数レジスタ
    `$a0, $a1, $a2, $a3`
  - 結果の値を返すレジスタ
    `$v0, $v1`
  - 戻りアドレスを格納するレジスタ
    `$ra`
- パラメータ数が多い場合(5個以上)
  - スタック上にパラメータを格納

手続き呼び出しと復帰

- 手続き呼び出し(jump and link)
  - 命令: `jal [address]`
  - 動作: `$ra = PC + 4` (次の命令のアドレス)
    `PC = [address]`
- 手続きからの復帰(jump register)
  - 命令: `jr $ra`
  - 動作: `PC = $ra`
- 注: PC(Program Counter)は命令のアドレスを格納するレジスタ

他の手続きを呼び出さない手続きの例

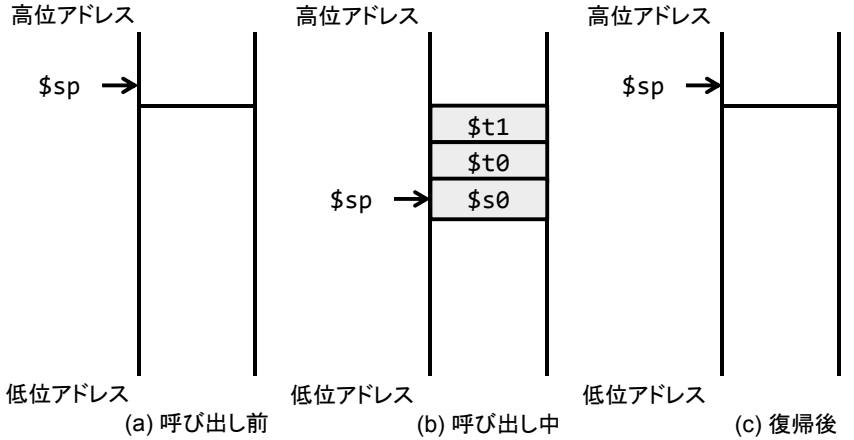
```
int leaf( int g, int h, int i, int j )
{
      int f;

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

関数 leaf のアセンブリ言語プログラム

```
leaf:  addi $sp, $sp, -12      # reserve 3 words space
       sw   $t1, 8($sp)        # save $t1
       sw   $t0, 4($sp)        # save $t0        作業用レジスタの退避
       sw   $s0, 0($sp)        # save $s0
       add  $t0, $a0, $a1      # $t0 = g + h
       add  $t1, $a2, $a3      # $t1 = i + j     関数内での演算
       sub  $s0, $t0, $t1      # f = $t0 - $t1
       add  $v0, $s0, $zero    # return f
       lw   $s0, 0($sp)        # restore $s0
       lw   $t0, 4($sp)        # restore $t0     作業用レジスタの復元
       lw   $t1, 8($sp)        # restore $t1
       addi $sp, $sp, 12       # release space
       jr   $ra                # go back to caller
```


手続き呼び出しの前後のスタックの状態



再帰関数（recursive function）の例

```
int fact( int n )
{
    if (n < 1)   return (1);
         else    return (n * fact(n-1));
}
```


関数 fact のアセンブリ言語プログラム(1)

```
fact:  addi $sp, $sp, -8       # reserve 2 words space
       sw   $ra, 4($sp)        # push return address
       sw   $a0, 0($sp)        # push n
       slti $t0, $a0, 1        # check if n < 1
       beq  $t0, $zero, L1     # if n>= 1 goto L1
       addi $v0, $zero, 1      # $v0 = 1
       addi $sp, $sp, 8        # discard 2 words space
       jr   $ra                # return $v0
```

関数 fact のアセンブリ言語プログラム(2)

```
L1:   addi $a0, $a0, -1        # set argument n-1
      jal  fact                # call fact with n-1
      lw   $a0, 0($sp)         # restore n
      lw   $ra, 4($sp)         # restore return address
      addi $sp, $sp, 8         # discard 2 words space
      mult $v0, $a0, $v0       # $v0 = n * fact(n-1)
      jr   $ra                 # return $v0
```

（注） mult は、乗算命令

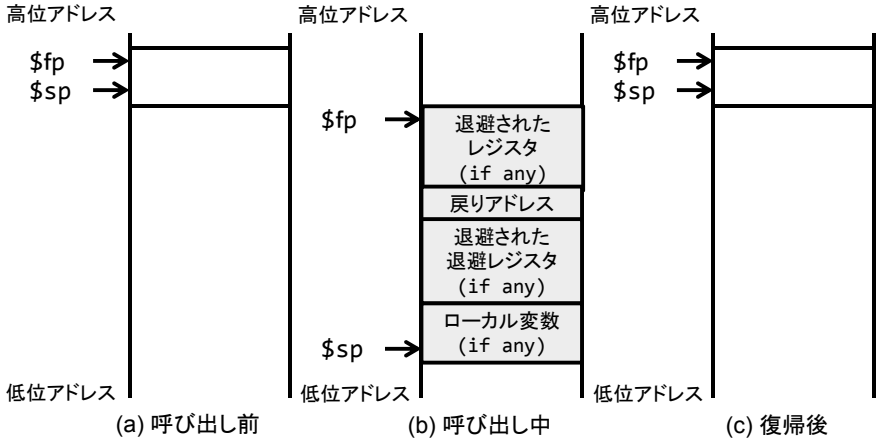
C言語での変数の記憶クラスとMIPSでのサポート方法

- C言語での変数の記憶クラス
  - 自動変数（automatic variable）
    - 手続きの内部だけで存在する
    - 手続きの終了に伴い廃棄される
  - 静的変数（static variable）
    - 手続きの開始，終了に関わらず存在する
    - 全ての手続きの「外」で宣言された変数
    - 予約語 static を用いて宣言された変数
- MIPSで静的変数を管理するためのレジスタ
  - $gp (global pointer)

手続き呼び出しの前に保存される情報と保存されない情報

保存される情報		保存されない情報	
退避レジスタ	$s0 - $s7	一時レジスタ	$t0 - $t9
スタック・ポインタ・レジスタ	$sp	引数レジスタ	$a0 - $a3
戻りアドレス・レジスタ	$ra	戻り値レジスタ	$v0 - $v1
スタック・ポインタより上のスタックの内容		スタック・ポインタより下のスタックの内容	

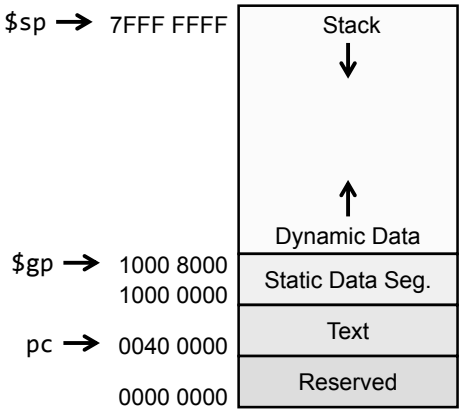
新しいデータ用のスペースのスタック上での割当て



$fp： frame pointer

新しいデータ用のスペースのヒープ上での割当て

- テキストセグメント（text segment）機械語コードを格納するメモリ領域
- 静的データセグメント（static data segment）
- ヒープ（heap）動的データを管理
  - malloc( )
  - free( )



MIPSのレジスタ規約

名前	レジスタ番号	用途	呼出し時退避?
$zero	0	定数0	該当せず
$v0-$v1	2-3	結果および式の評価のための値	No
$a0-$a3	4-7	引数	No
$t0-$t7	8-15	一時	No
$s0-$s7	16-23	レジスタ変数	Yes
$t8-$t9	24-25	予備の一時	No
$gp	28	グローバル・ポインタ	Yes
$sp	29	スタック・ポインタ	Yes
$fp	30	フレーム・ポインタ	Yes
$ra	31	戻りアドレス	Yes

講義内容（1）

- 条件判定用の命令
- コンピュータ・ハードウェア内での手続きのサポート
- 人との情報交換
- 32ビットの即値およびアドレスに対するMIPSのアドレッシング方式
- 並列処理と命令：同期

文字コード ASCII (1)

	000	001	010	011	100	101	110	111
00000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
00001	BS	HT	LF	VT	FF	CR	SO	SI
00010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
00011	CAN	EM	SUB	ESC	FSP	GSP	RSP	USP
00100	space	!	”	#	$	%	&	’
00101	(	)	*	+	,	-	.	/
00110	0	1	2	3	4	5	6	7
00111	8	9	:	;	<	=	>	?

文字コード ASCII (2)

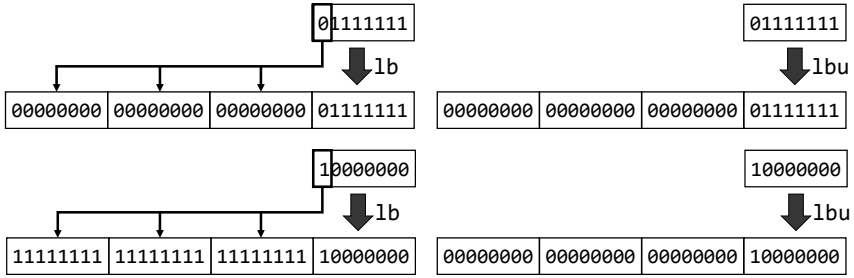
	000	001	010	011	100	101	110	111
01000	@	A	B	C	D	E	F	G
01001	H	I	J	K	L	M	N	O
01010	P	Q	R	S	T	U	V	W
01011	X	Y	Z	[	¥	]	^	_
01100	`	a	b	c	d	e	f	g
01101	h	i	j	k	l	m	n	o
01110	p	q	r	s	t	u	v	w
01111	x	y	z	{	|	}	~	DEL

データ転送命令

- メモリからレジスタへのバイトデータ転送
  - 命令: load byte
  - 記法: `lb $s1, 100($s2)`
  - 動作: `$s1 = Mem[$s2+100]`
- レジスタからメモリへのバイトデータ転送
  - 命令: store byte
  - 記法: `sb $s1, 100($s2)`
  - 動作: `Mem[$s2+100] = $s1`

符号拡張(sign extention)

- 符号付きロード命令
  - `lb` load byte (符号付ロード命令)
  - `lbu` load byte unsigned (符号なしロード命令)



文字列(string)

- 文字列の長さは通常は可変
- 文字列の長さを表す方法
  1. 文字列の最初の文字を文字列の長さを示すために使用
  2. 付随する変数に文字列の長さを保持する (文字列を構造体として扱う)
  3. 文字列の最後の文字に文字列の終端を示す特定の文字を使用(C言語では null = 0)

ユニコード(Unicode)の取り扱い

- 半語(half word)で1つの文字を表すコード体系
- ユニコードを扱うためのデータ転送命令
  - `lh` load half
  - `lhu` load half unsigned
  - `sh` store half
  - データはレジスタ(32ビット)の右側の16ビットに格納

講義内容(1)

- 条件判定用の命令
- コンピュータ・ハードウェア内での手続きのサポート
- 人との情報交換
- 32ビットの即値およびアドレスに対するMIPSのアドレッシング方式
- 並列処理と命令:同期

アドレッシング・モード(addressing mode)(1)

- オペランドを指定する方法
- 指定対象
  - レジスタ(general purpose register)
  - 主記憶(main memory)
  - 特殊レジスタ(special register)
- 暗に(implicit)指定する方法
  - 命令によって一意に決まる
  - 命令の一部で指定される

アドレッシング・モード(addressing mode)(2)

- 陽に(explicit)指定する方法
  - 汎用レジスタの番号
  - 主記憶のアドレスを指定
- 実効アドレス(effective address)
  - (主にメモリアクセスの場合に)実際にアクセスされるメモリのアドレス

MIPSのアドレッシング・モード(1)

- 即値アドレッシング(immediate addressing)
  - オペランドは命令中で指定した定数
- レジスタ・アドレッシング(register addressing)
  - オペランドはレジスタ
- ベース相対アドレッシング(base addressing)
  ディスプレースメント・アドレッシング
  (displacement addressing)
  - 命令中で指定した定数とレジスタの内容の和

MIPSのアドレッシング・モード(2)

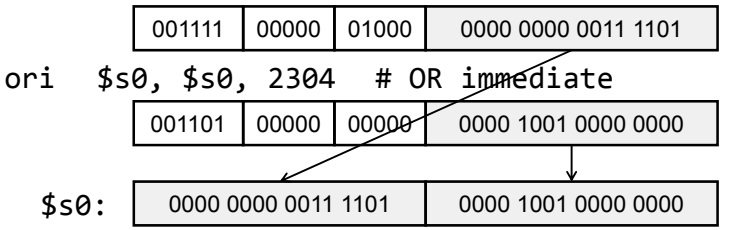
- PC相対アドレッシング
  (PC-relative addressing)
  - 命令中で指定した定数とPCの値の和
- 擬似直接アドレッシング
  (pseudo direct addressing)
  - PCの上位4ビット(bit 32 – bit 29), 命令中の26ビットの即値, "00" を連結

即値アドレッシング
(immediate addressing)

- 命令語の一部(即値)をオペランドとして用いる

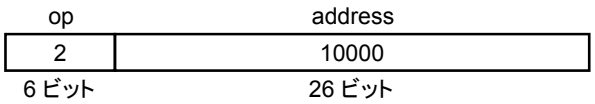
- 例: 32ビットの定数のロード

```
lui   $s0, 61          # Load upper immediate
ori   $s0, $s0, 2304   # OR immediate
```



ジャンプ命令でのアドレッシング

- ジャンプ命令(J形式)
  - 例: `j  10000  # 10000番地にジャンプ`



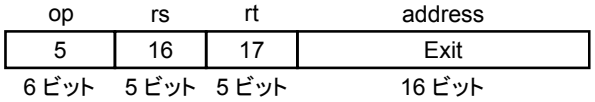
  - ジャンプ先の可能なアドレスはワードアドレス
    - PCの上位4ビットに即値および "00" を連接することにより32ビットのアドレスを得る
  - ジャンプ命令およびジャンプアンドリンク(jal)命令では、J形式を適用している

ジャンプ命令についての補足

- アドレッシング範囲の限界
  - ジャンプ命令では，PCの上位4ビットをそのまま用い下位28ビットのみを置き換えるので，ジャンプ命令で指定可能なアドレスの範囲は2^{28} バイト
  - プログラムを256 MBの境界を越えて配置出来ない
- 256 MBの境界を越えてジャンプする方法
  - レジスタジャンプ（jr）命令と他の命令の組合わせ

条件分岐命令でのアドレッシング

- 条件分岐命令（I形式）
  - 例: `bne $s0, $s1, Exit`



  - PC相対アドレッシング（PC-relative addressing）
    - PCの値 ＝ 現在のPCの値 + address
    - address は語アドレスを表し，符号付数として扱う（アドレス計算では符号拡張が必要）
    - 分岐先のアドレスは $(PC + 4) \pm 2^{17}$ の範囲
    - 命令実行中のPCの値は，その命令のアドレス＋4となっている

機械語での分岐オフセットの表記（1）

- C言語プログラム

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

- 対応するアセンブリ･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:
```


機械語での分岐オフセットの表記（2）

- アセンブリ結果

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

より遠くへの分岐

- 近傍への条件分岐

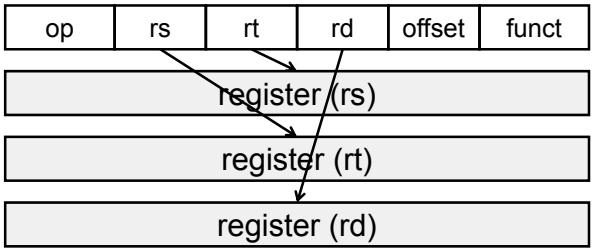
```
beq  $s0, $s1, L1
```

- より遠くへの条件分岐は，条件分岐命令とジャンプ命令を組合わせて実現する

```
    bne  $s0, $s1, L2
    j    L1
L2:
```


レジスタ・アドレッシング（register addressing）

- オペランドにレジスタを指定

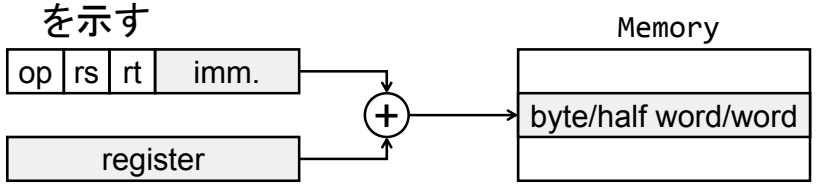


- 例：加算命令

```
Add  $t0, $a0, $a1   # $t0 = $a0 + $a1
```


ベース相対アドレッシング（base addressing）

- 命令中に指定した定数とレジスタの内容の和によってオペランドが格納されているメモリの位置を示す

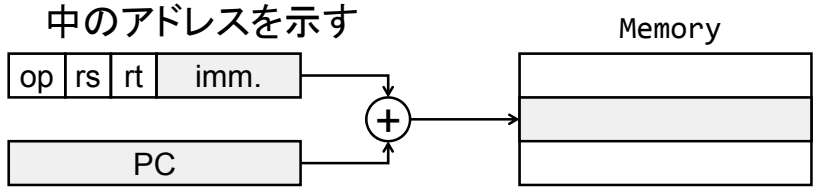


- 例：

```
lw  $t0, 32($s0)    # Load word from
                    # $s0 + 32
```


PC相対アドレッシング（PC-relative addressing）

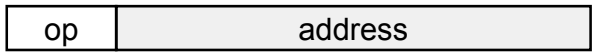
- PCと命令中に指定した定数の和によってメモリ中のアドレスを示す



  - PCには実行中の命令のアドレス＋4（次に実行される予定の命令のアドレス）が格納されている
  - imm.には，分岐先のアドレスーPCの値を格納

擬似直接アドレッシング（pseudo direct addressing）

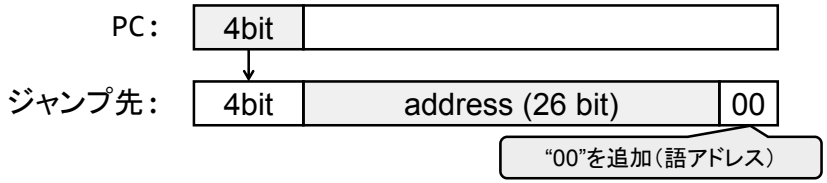
- 命令中の即値をメモリ・アドレスとして用いる



- 例：ジャンプ命令

```
j     10000            # Jump to 10000
```

- MIPSでの疑似直接アドレッシング



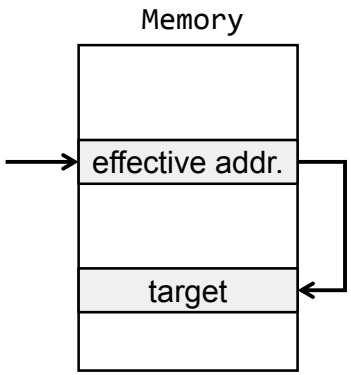
条件分岐命令

- 例：`bne $s0, $s1, Exit`

- 分岐先アドレス
  - PC = PC + Exit
  - Exit は語アドレスであり，符号付き数
  - 分岐範囲は $(\mathrm{PC}+4) \pm 2^{17}$
  - 注：PC+4 は、次の命令のアドレス

間接アドレッシング（indirect addressing）

- 指定されたアドレスに格納されているメモリ中のデータを実効アドレスとして，再度メモリにアクセスする
- CISC命令
- MIPSの命令セットにはない



機械語とアセンブリ言語ニモニックの対応(1)

OP (31:29)	Op(28:26)							
	000	001	010	011	100	101	110	111
000	R形式	Bitz/gez	jump	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010	TLB	FIPt						
011								
100	lb	lh	lwl	lw	lbu	lhu	lwr	
101	sb	sh	swl	sw			swr	
110	llw	lwcl						
111	scw	swcl						

機械語とアセンブリ言語ニモニックの対応(2)

23-21 / 25-24	Op(31:26)=010000（TLB），rs(25:21)							
	000	001	010	011	100	101	110	111
00	mfc0		cfc0		mtc0		etc0	
01								
10								
11								

機械語とアセンブリ言語ニモニックの対応(3)

2-0 / 5-3	Op(31:26)=000000（R形式），funct(5:0)							
	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr			syscall	break		l
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

講義内容（1）

- 条件判定用の命令
- コンピュータ・ハードウェア内での手続きのサポート
- 人との情報交換
- 32ビットの即値およびアドレスに対するMIPSのアドレッシング方式
- 並列処理と命令：同期

データ競合（data race）の例

- タスクA

```
A1: lw   $s0, 0($s3)
A2: addi $s0, $s0, 1
A3: sw   $s0, 0($s3)
```

- タスクB

```
B1: lw   $s1, 0($s3)
B2: subi $s1, $s1, 1
B3: sw   $s1, 0($s3)
```

- 初期値
  - `mem[$s3] = 10`
- 実行順序1
  - A1, A2, A3, B1, B2, B3 の場合：mem[$s3] = 10
  - A1, B1, B2, B3, A2, A3 の場合：mem[$s3] = 11
  - A1, B1, A2, A3, B2, B3 の場合：mem[$s3] = 9

同期（synchronization）

- タスク間の協調動作
  - あるタスクが読む必要のある値に対し，他のタスクが新しい値を書き込もうとするような事態
  - 書き込みが完了し，その値を読んでも問題ないと判断できるようにするために，タスク間の同期が必要
- 同期を取るための手段
  - 相互排除（mutual exclusion）領域の形成
  - ロック（lock）
  - アンロック（unlock）

マルチ・プロセッサでの同期

- ハードウェア・プリミティブ（hardware primitive）
  - メモリ・ロケーションの不可分（atomic）な読み出しと変更の実行
- 不可分な交換（atomic exchange）
  不可分なスワップ（atomic swap）
- ロック動作の実現
  - 共有変数 x
  - $x = 0$ であれば，ロックは空いており，利用可
    $x = 1$ であれば，ロックは利用できない

ロック動作の実現

- ロック動作
  - レジスタ内に設定されている 1 を x の内容と不可分に交換
  - 交換後のレジスタの値が 1 であれば，他のプロセスがロックを使用中なので，利用できない
    x の値は 1 のまま
  - 交換後のレジスタの値が 0 であれば，利用可
    x の値は 1 となる
- アンロック動作
  - 利用が終了した場合には，x に 0 を格納する

不可分なメモリ操作の実現

- ll（load linked）命令
  - 指定されたメモリ・ロケーションの内容が，同じアドレスに対する sc 命令の実行前に変更されると，sc 命令の実行は不成功
- sc（store conditional）命令
  - レジスタの値をメモリに格納する
  - レジスタの値を，実行が成功すれば 1 に，失敗すれば 0 に変更する

ll命令とsc命令の使用例

```
try:  add   $t0, $zero, $s4  # 交換する値をコピー
      ll    $t1, 0($s1)      # load linked
      sc    $t0, 0($s1)      # store conditional
      beq   $t0, $zero, try  # ストアに失敗，やり直し
      add   $s4, $zero, $t1  # ロードした値を$s4に格納
```

- 実行後には，$s4の内容と$s1によって指定されるメモリ・ロケーションの内容が不可分に交換されている
- ll命令とsc命令の間に他のプロセッサが割りこんでメモリの値を変更してしまった場合には，sc命令の$t0に0が返される（sc命令の実行が失敗）

ll命令とsc命令についての補足

- 以下の場合にsc命令は不成功になる
  - 単一プロセッサで，ll命令とsc命令の間で，プロセッサがコンテキスト・スイッチを行った場合
  - ll命令の対象アドレスに別のプロセスがストアを試みた場合
  - ll命令発行後に例外が発生した場合
- ll命令とsc命令の間に挿入しても安全な命令は，レジスタ・レジスタ演算命令のみであり，それ以外はデッドロックが発生する可能性がある