第2章 命令：コンピュータの言葉（3）

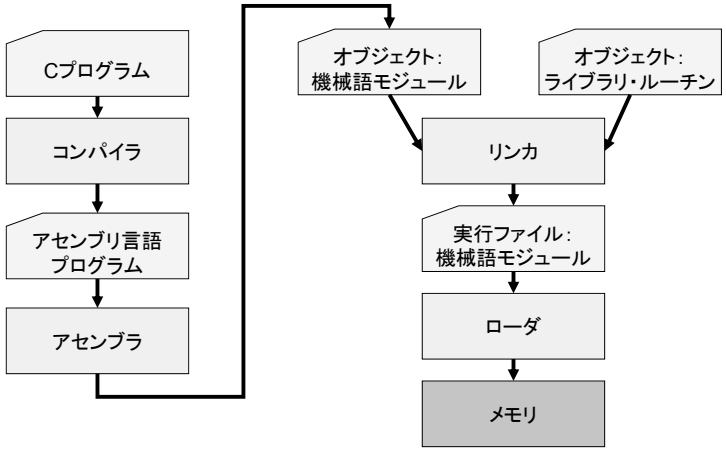
大阪大学 大学院 情報科学研究科
今井 正治

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

講義内容

- プログラムの翻訳と起動
- Cプログラムの包括的な例題解説
- 配列とポインタの対比
- ARMの命令セット
- x86の命令セット
- 誤信と落とし穴

Cプログラムの翻訳階層



C言語プログラムの翻訳過程（1）

- コンパイラ
  - C言語プログラムをアセンブリ言語（assembly language）のプログラムに翻訳
  - アセンブリ言語は，機械語のシンボル（記号）による表現
- アセンブラ
  - アセンブリ言語のプログラムをオブジェクト・モジュールに翻訳
  - 擬似命令（pseudo instruction）の利用が可能
    - 例：`move $s0, $t1 ≡ add $s0, $zero, $t1`
  - シンボル表（symbol table）を作成

C言語プログラムの翻訳過程(2)

- オブジェクト・ファイル(object file)の構成要素
  - オブジェクト・ファイル・ヘッダ(object file header)
  - テキスト・セグメント(text segment)
  - 動的データ・セグメント(dynamic data segment)
  - リロケーション情報(relocation information)
  - シンボル表(symbol table)
  - デバッグ情報(debug information)

C言語プログラムの翻訳過程(3)

- リンカ(linker), リンク・エディタ(link editor), リンケージ・エディタ(linkage editor)
  1. コード・モジュールおよびデータ・モジュールをメモリ中に置く
  2. データおよび命令のラベルのアドレスを判定する
  3. 内部および外部の参照先を解析する
  - 絶対アドレスによる参照はすべて実際のロケーションを反映するように再配置(relocation)する必要がある
  - リンカの出力は実行ファイル(executable file)

C言語プログラムの翻訳過程(4)

- ローダ(loader)(Unixの場合)
  1. 実行ファイルのヘッダを読んで, テキスト・セグメントとデータ・セグメントの大きさを判定する.
  2. テキスト・セグメントとデータ・セグメントを保持するのに十分な大きさのアドレス空間を確保する.
  3. 実行ファイルから命令とデータをメモリにコピーする.
  4. メイン・プログラムに渡されるパラメータが存在すれば, スタック上にコピーする.
  5. マシンのレジスタを初期化し, スタック・ポインタに最初の空きロケーションを設定する.

C言語プログラムの翻訳過程(5)

6. 開始(start-up)ルーチンにジャンプする.
   開始ルーチンはスタックからプログラムのパラメータをレジスタにコピーして, プログラムのマイン・ルーチンを呼び出す.
   メイン・ルーチンから制御が戻ると, 開始ルーチンは終了(exit)システム・コール(system call)を用いてプログラムを終了させる.

擬似命令（pseudo instruction）

MIPSの擬似命令	記述例	形式	意味
move	move $rd, $rs	R	$rd = $rs
multiply	mult $rd, $rs, $rt	R	$rd = $rs * $rt
multiply immediate	multi $rd, $rs, imm	I	$rd = $rs * imm
load immediate	li $rd, imm	I	$rd = imm
branch less than	blt $rs, $rt, addr	I	if $rs<$rt goto addr
branch less than or equal	ble $rs, $rt, addr	I	if $rs≤$rt goto addr
branch greater than	bgt $rs, $rt, addr	I	if $rs>$rt goto addr
branch greater than or equal	bge $rs, $rt, addr	I	if $rs≥$rt goto addr

講義内容

- プログラムの翻訳と起動
- Cプログラムの包括的な例題解説
- 配列とポインタの対比
- ARMの命令セット
- x86の命令セット
- 誤信と落とし穴

Cプログラムからアセンブリ・コードへの翻訳手順

1. Cプログラム中の変数にレジスタを割付ける
2. 手続き本体用のコードを生成する
3. 呼出し側（caller）と被呼出し側（callee）の両方で使用されるレジスタを退避する

swap手続きの翻訳（1）

- Cプログラム

```
void swap( int v[], int k )
{
    int temp;
    temp   = v[k];
    v[k]   = v[k+1];
    v[k+1] = temp;
}
```

- 変数へのレジスタの割付け
  - 変数 v:　　$a0
  - 変数 k:　　$a1
  - 変数 temp:　$t0

swap手続きの翻訳(2)

- 手続き本体のコードの生成

```
swap: sll  $t1, $a1, 2       # $t1  ← k*4
      add  $t1, $a0, $t1     # $t1  ← v+k*4
      lw   $t0, 0($t1)       # $t0  ← v[k]
      lw   $t2, 4($t1)       # $t2  ← v[k+1]
      sw   $t2, 0($t1)       # v[k] ← $t2
      sw   $t0, 4($t1)       # v[k+1] ← $t0
      jr   $ra               # return
```

- レジスタの退避と復帰
  - この例ではなし

sort手続きの翻訳(1)

- Cプログラム

```
void sort( int v[], int n )
{
    int i, j;
    for( i=0; i<n; i+=1 ) {
        for( j=i-1; j>=0 && v[j]>v[j+1]; j-=1 )
            { swap(v, j); }
    }
}
```

- 変数へのレジスタの割付け
  - 変数 v: $a0
  - 変数 k: $a1
  - 変数 i: $s0
  - 変数 j: $s1

sort手続きの翻訳(2)

- 最初のfor loop

```
for( i=0; i<n; i+=1 )
```

- 対応するアセンブリ・コード

```
        move  $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $a1     # if i≥n then $t0 ← 0
        beq   $t0, $zero, exit1 # if i≥n then go to exit1
              ...
              (最初のforループの本体)
              ...
        addi  $s0, s0, 1        # i += 1
        j     for1tst           # jump to the top of loop
exit1:
```


sort手続きの翻訳(3)

- 2番目のfor loop

```
for( j=i-1; j<=0 && v[j]>v[j+1]; i-=1 )
```

- 対応するアセンブリ・コード

```
        addi  $s1, $s0, -1      # j = i-1
for2tst: slti $t0, $s1, 0       # if j<0 then $t0 = 1
        bne   $t0, $zero, exit2 # if i<0 then go to exit2
        sll   $t1, $s1, 2       # $t1 = j*4
        add   $t2, $a0, $t1     # $t2 = v + (j*4)
        lw    $t3, 0($t2)       # $t3 = v[j]
        lw    $t4, 4($t2)       # $t4 = v[j+1]
        slt   $t0, $t4, $t3     # $t4 ≥ $t3 then $t0 = 0
        beq   $t0, $zero, exit2 # $t4 ≥ $t3 then go to exit2
              (2番目のforループの本体)
        addi  $s1, s1, -1       # i += 1
        j     for2tst           # jump to the top of loop
exit2:
```

sort手続きの翻訳(4)

- 2番目のfor loopの本体

```
swap(v, j);
```

- 対応するアセンブリ・コード

```
jal   swap  # call swap
```

- sortからのパラメータの引渡し
  - swap にパラメータを引渡すために、レジスタ$a0, $a1を退避

```
move  $s2, $a0    # $a0 を $s2 に退避
move  $s3, $a1    # $a1 を $s3 に退避
```

  - swap にパラメータを引渡す

```
move  $a0, $s2    # swap にパラメータ v を引渡す
move  $a1, $s1    # swap にパラメータ j を引渡す
```


sort手続きの翻訳(5)

- 手続き呼出し間でのレジスタの退避
  - 戻り番地　　$ra
  - sort 手続きで使用されているレジスタ
    $s0, $s1, $s2, $s3
  - スタック上に領域を確保してレジスタを退避

```
addi  $sp, $sp, -20     # レジスタ5個分の領域を確保
sw    $ra, 16($sp)      # 戻り番地を退避
sw    $s3, 12($sp)      # レジスタ$s3を退避
sw    $s2, 8($sp)       # レジスタ$s2を退避
sw    $s1, 4($sp)       # レジスタ$s1を退避
sw    $s0, 0($sp)       # レジスタ$s0を退避
```


sort手続きの翻訳(6)

- レジスタの復元
  - 戻り番地　　$ra
  - sort 手続きで使用されていたレジスタ
    $s0, $s1, $s2, $s3
  - スタックからレジスタを復元し, 呼出し元に戻る

```
lw    $s0, 0($sp)       # $s0 を復元
lw    $s1, 4($sp)       # $s1 を復元
lw    $s2, 8($sp)       # $s2 を復元
lw    $s3, 12($sp)      # $s3 を復元
lw    $ra, 16($sp)      # 戻り番地を復元
addi  $sp, $sp, 20      # レジスタ5個分の領域を開放
jr    $ra               # 呼出し元に戻る
```


講義内容

- プログラムの翻訳と起動
- Cプログラムの包括的な例題解説
- 配列とポインタの対比
- ARMの命令セット
- x86の命令セット
- 誤信と落とし穴

配列を用いたプログラムの例

- 配列を用いたプログラム

```
clear1(int array[], int size)
{
   int i;
   for (i=0; i < size; i += 1)
       array[i] = 0;
}
```

- レジスタの割付け
  - $a0　arrayのアドレス
  - $a1　sizeの値
  - $t0　i

配列版のアセンブリ・コード

```
       move  $t0, $zero        # $t0 = 0
loop1: sll   $t1, $t0, 2       # $t1 = $t0 * 4
       add   $t2, $a0, $t1     # $t2 = array[i]のアドレス
       sw    $zero, 0($t2)     # array[i] = 0
       addi  $t0, $t0, 1        # i = i + 1
       slt   $t3, $t0, $a1     # if(i<size) $t3 = 1
       bne   $t3, $zero, loop1 # if(i<size) go to loop1
```


ポインタを用いたプログラムの例

- ポインタを用いたプログラム

```
clear2(int *array, int size)
{
   int *p;
   for (p = &array[0]; p < &array[size]; p = p + 1)
       *p = 0;
}
```

- レジスタの割付け
  - $a0　arrayのアドレス
  - $a1　sizeの値
  - $t0　p

ポインタ版のアセンブリ・コード（1）

- 最適化前

```
       move  $t0, $a0          # p = &array
loop2: sw    $zero, 0($t0)     # memory[p] = 0
       addi  $t0, $t0, 4       # p = p + 4
       sll   $t1, $a1, 2       # $t1 = size * 4
       add   $t2, $a0, $t1     # $t2 = array[size]のアドレス
       slt   $t3, $t0, $t2     # if(i<size) $t3 = 1
       bne   $t3, $zero, loop2 # if(i<size) go to loop1
```

ポインタ版のアセンブリ・コード(2)

□ 最適化後

```
       move  $t0, $a0          # p = &array
       sll   $t1, $a1, 2       # $t1 = size * 4
       add   $t2, $a0, $t1     # $t2 = array[size]のアドレス
loop2: sw    $zero, 0($t0)     # memory[p] = 0
       addi  $t0, $t0, 4       # p = p + 4
       slt   $t3, $t0, $t2     # if(i<size) $t3 = 1
       bne   $t3, $zero, loop2 # if(i<size) go to loop1
```


講義内容

- □ プログラムの翻訳と起動
- □ Cプログラムの包括的な例題解説
- □ 配列とポインタの対比
- □ ARMの命令セット
- □ x86の命令セット
- □ 誤信と落とし穴

ARMとMIPSの命令セットの類似点

比較項目	ARM	MIPS
発表年	1985	1985
命令長(ビット)	32	32
アドレス空間(サイズ, モデル)	32ビット, フラット	32ビット, フラット
データ整列化	整列化	整列化
データ・アドレッシング・モード数	9	3
整数レジスタ(数, モデル, サイズ)	16 GPR × 32ビット	32 GPR × 32ビット
入出力	メモリ・マップ方式	メモリ・マップ方式

レジスタ・レジスタ命令の対応関係

命令名	ARM	MIPS
Add	add	addu, addiu
Add (trap if overflow)	adds, swivs	add
Subtract	sub	subu
Subtract (trap if overflow)	subs, swivs	sub
Multiply	mul	mult, multu
Divide	-	div, divu
And	and	and
Or	or	or
Xor	eor	xor
Load high part register	-	lui
Shift left logical	lsl	sllv, sll
Shift right logical	lsr	srlv, srl
Shift right arithmetic	asr	srav, sra
Compare	cmp, cmn, tst, teq	slt/I, slt/iu

データ転送命令の対応関係

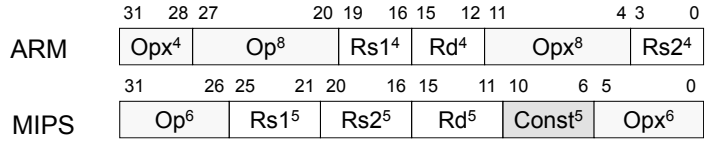
命令名	ARM	MIPS
Load byte signed	ldrsb	lb
Load byte unsigned	ldrb	lbu
Load halfword signed	ldrsh	lh
Load halfword unsigned	ldrh	lhu
Load word	ldr	lw
Store byte	strb	sb
Store halfword	strh	sh
Store word	str	sw
Read, write special register	mrs, msr	move
Atomic exchange	swp, swpb	ll, sc

アドレッシング・モードの比較

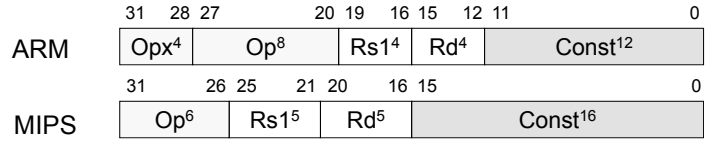
アドレッシング・モード	ARM	MIPS
レジスタ・オペランド	×	×
即値オペランド	×	×
レジスタ＋オフセット（ディスプレースメントまたはベース相対）	×	×
レジスタ＋レジスタ（インデックス修飾）	×	-
レジスタ＋スケール付きレジスタ（スケール付き）	×	-
レジスタ＋オフセットかつレジスタ更新	×	-
レジスタ+レジスタかつレジスタ更新	×	-
自動インクリメント，自動ディクリメント	×	-
PC相対データ	×	-

命令形式（1）

□ レジスタ・レジスタ

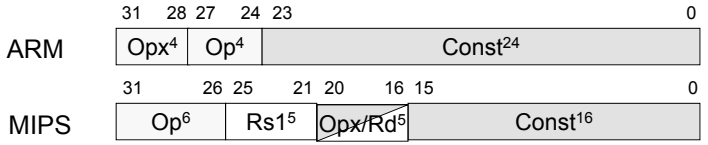


□ データ転送

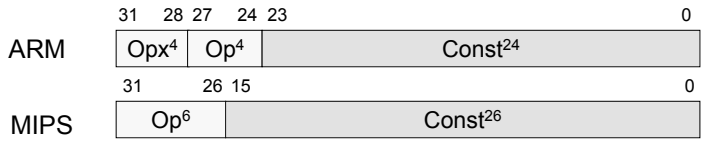


命令形式（2）

□ 分岐



□ ジャンプ/コール



ARMに固有の機能

- MIPSにはない，算術命令と論理命令
- ARMは0レジスタを持たない
- 多倍長算術演算のサポート
- 即値フィールドの取り扱い
  - 下位8ビットを32ビットに符号拡張
  - 上位4ビットで指定される値を2倍したビット数だけ右に回転
- 算術命令および論理命令の第2レジスタに対する，演算前のシフトオプション
  - 左論理シフト、右論理シフト、右算術シフト、右ローテート
- レジスタ・グループの保存・復元命令（block loads and stores）
  - 16本からなるレジスタ・ファイルのうちの任意のレジスタを命令の即値で指定して保存，復元が可能
  - 手続き呼出し・復帰時のレジスタの退避・復元，メモリブロックのコピーに使用

MIPSにはないARMの算術命令と論理命令

名前	定義	ARM v.4	MIPS
Load immediate	Rd = imm	mov	addi $0
Not	Rd = ~(Rs1)	mvn	nor $0
Move	Rd = Rs1	mov	or $0
Rotate right	Rd = Rs1 >> i $Rd_{0\ldots i-1} = Rs_{31-i\ldots31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 – Rs1	rsb, rsc	
Support for multiword integer add	Carryout.Rd = Rd + Rs1 + Old CarryOut	abcs	-
Support for multiword integer sub	Carryout.Rd = Rd - Rs1 + Old CarryOut	abcs	-

講義内容

- プログラムの翻訳と起動
- Cプログラムの包括的な例題解説
- 配列とポインタの対比
- ARMの命令セット
- x86の命令セット
- 誤信と落とし穴

x86のレジスタ・セット

算術，論理，データ転送の命令のタイプ

ソース／デスティネーション・オペランド	第2のソース・オペランド
レジスタ	レジスタ
レジスタ	即値
レジスタ	メモリ
メモリ	レジスタ
メモリ	即値

■ 即値の長さは，8ビット，16ビット，32ビット

32ビットのアドレッシング・モード

- レジスタ間接
  - アドレスはレジスタ中
- ベース相対モード
  - ベース・レジスタ＋ディスプレースメント
- ベース相対＋スケール付きインデックス
  - ベース＋($2^{スケール}$ ×インデックス)
    スケールは0, 1, 2, 3
- ベース相対＋スケール付きインデックス
  ＋ディスプレースメント
  - ベース＋($2^{スケール}$ ×インデックス)＋ディスプレースメント
    スケールは0, 1, 2, 3

整数演算でのデータ・タイプ

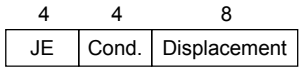
- データ・タイプ
  - 8ビット（バイト: byte），16ビット（ワード: word），
    32ビット（ダブルワード: double word）
- オペランドの長さの指定
  - デフォルト値：コード・セグメント・レジスタで指定
  - 命令に8ビットのプレフィックス（prefix）を付加

整数に対する操作

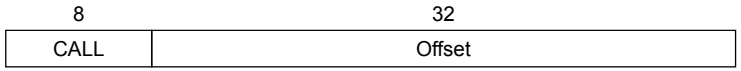
- データ転送命令
  - move, push, pop
- 算術演算命令および論理演算命令，条件判定および整数や小数の演算操作など
- フローの制御，条件分岐，無条件ジャンプ，手続き呼出しと手続きからの復帰
  - 条件分岐には，条件コード（condition code）ないしフラグ（flag）を使用
- 文字列に対する命令，文字列の転送や比較

命令タイプ(1)

- 命令長
  - 1バイト～15バイト
- 命令タイプの例
  - JE EIP＋ディスプレースメント

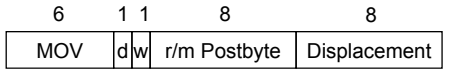


  - CALL

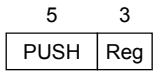


命令タイプ(2)

- 命令タイプの例(続き)
  - MOV EBX, [EDI+45]

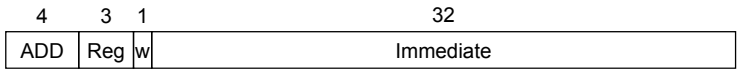


  - PUSH ESI

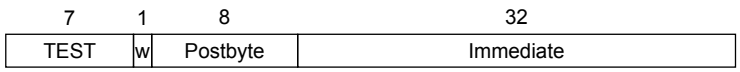


命令タイプ(3)

- 命令タイプの例(続き)
  - ADD EAX, #6765



  - TEST EDX, #42



講義内容

- プログラムの翻訳と起動
- Cプログラムの包括的な例題解説
- 配列とポインタの対比
- ARMの命令セット
- x86の命令セット
- 誤信と落とし穴

誤信と落とし穴(1)

- 誤信: 命令を強力にすれば性能が改善される
- 落とし穴: 最高の性能を実現するために, プログラムをアセンブリ言語で組むこと
- 誤信: バイナリ互換性の商業的な重要性は, 成功を収めた命令セットは変化しないことを意味する

誤信と落とし穴(2)

- 落とし穴: バイト・アドレッシング方式を用いるマシン上の一連の語のアドレスは1ずつ増えるのではないことを忘れること
- 落とし穴: 自動変数のへのポインタをそれが定義された手続きの外で使うこと