5.6 アセンブリ生成

この節では，前の節の説明に従って生成した仮想マシンコードを入力とし，現実の計算機アーキテクチャの一つである MIPS のアセンブリコードを生成する方法について説明する．説明にあたり，MIPS に関する基本的な知識は仮定する．たとえば：

```
li $v0, 1
li $v1, 2
add $v0, $v0, $v1
li $v1, 3
mul $v0, $v0, $v1
```

は，$(1+2)\times 3$ を計算し，その結果を v0 レジスタに格納する命令列であることが理解でき，また：

```
    .data
ARR:
    .word 1, 2, 3, 4, 5
    .text
    la $v2, ARR
    lw $v0, 1($v2)
    mul $v1, $v0, $v0
    sw $v1, 3($v2)
```

は，メモリ中の ARR ラベルの付けられた番地から順に並んでいる長さ 5 の 32 ビット整数配列の中から，2 番目の値を取り出し，その値の二乗を同じ配列の 4 番目に上書きする，ということが理解できるものとする．もしこれらを理解できないようであれば，[?] を読みなおして復習すること．

5.6.1 MIPS の呼出し規約

現実のハードウェアが備えている物理レジスタの個数は有限である．そのため，関数（とくに再帰関数）を用いるプログラムではプログラム中のすべての計算を物理レジスタだけで実行することは困難であり，一般に，何らかの方法に則ってプログラム中の各変数の値を格納する場所をメモリ中のどこかへ確保する必要がある．

たとえば，次のような ML[4] のプログラム[1]：

```
let rec f a = a + 1
and g b = f b
in g 0
```

[1]読みやすさのため，ML[4] のシンタックスではなく，同等の言語機能による OCaml のシンタックスを使って説明する．

を素朴に実行するだけであれば，プログラム全体で,「関数 f の引数 a を格納する場所」「関数 f を呼び出した結果（返り値）を格納する場所」「関数 g の引数 b を格納する場所」「関数 g の返り値を格納する場所」の計 4 ワード分を確保すれば十分である．

しかし，そのような素朴な方法だと，たとえば：

```
let rec fact = fun n -¿ if n > 0 then n * fact (n + (-1)) else 1
in fact 10
```

のような再帰関数 fact においては，複数の異なる (再帰) 呼出しの引数や返り値の格納場所が同じ領域を使うことになり，実行がおかしくなってしまう．具体的には，関数呼出し fact 9 の直後には，その返り値に対し 10 を掛ける必要があるが，関数呼出し fact 10 の引数である値 10 は，fact の引数を格納するただ一つの場所に置かれていたため，すでに値 9 で上書きされ失われてしまっている（正確には，さらに $8, 7, \ldots, 0$ で上書きされているが，ともかく 10 という値はもはや存在しない）．

結局のところ，関数定義単位で必要な領域のサイズを求め，その総和分を確保するだけでは不十分であり，関数呼出し毎に異なる場所を確保する必要があることが分かる．そこで，関数機能を持つプログラミング言語（関数型言語に限らず，たとえば C 言語なども含む）の処理系では，通常，呼出し/リターンの制御構造に対応できるよう，各関数呼出しの実行に必要となるメモリ領域を管理するためのデータ構造として，LIFO (last-in first-out) で管理するスタックを用いる．また，このスタックは，呼出し側が関数呼出しの次に実行するべき命令のアドレス（**リターンアドレス** (*return address*) と呼ばれる）を保存しておくためにも用いられる（詳細は後述）．

原理的には，ここまでに説明したように各関数の実行で必要となるすべての引数・返り値・局所変数（ならびにリターンアドレス）のための場所をスタック上に必ず確保することにすれば，再帰関数を含むようなプログラムであっても正しく実行できるが，それだけだと，プログラム実行中に計算されるすべての値をメモリ領域にいちいち格納するため，レジスタを有効活用できておらず実行性能はあまり良くない．たとえば，返り値を格納する場所をスタック上のある場所に定めてしまうと,（とくにレジスタで計算を行う RISC 系のハードウェアにおいては）呼び出された側が最後にスタックのその場所に返り値をストアしたすぐ後に，呼出し側が同じ場所からレジスタへロードするということが頻繁に起こる．この実行オーバヘッドは，たとえば「関数呼出しの返り値は必ず v0 レジスタを使って受け渡しする」と決めておけば避けることができる．まったく同様に，引数の受け渡しについてもレジスタの使い方に関し何らかの約束事を決めておけば，実行効率を良くすることができる．

以上のようなスタックとレジスタの使い方に関する約束事は，レジスタの種類や個数，関数の呼出し/リターンに用いることのできるジャンプ命令の詳細な振舞いなどに依存するため，通常，計算機アーキテクチャ毎に定められている．関数呼出しに関するそのような約束事は，一般に**呼出し規約** (*calling convention*) と呼ばれる．

なお，本講義で作成する ML^4 コンパイラは MIPS アセンブリをターゲットとしているため，本来であれば MIPS の呼出し規約に厳密に従うべきところだが，説明を簡単にするため，少し簡略化した独自の呼出し規約を用いている．より本格的なプログラミング言語や言語処

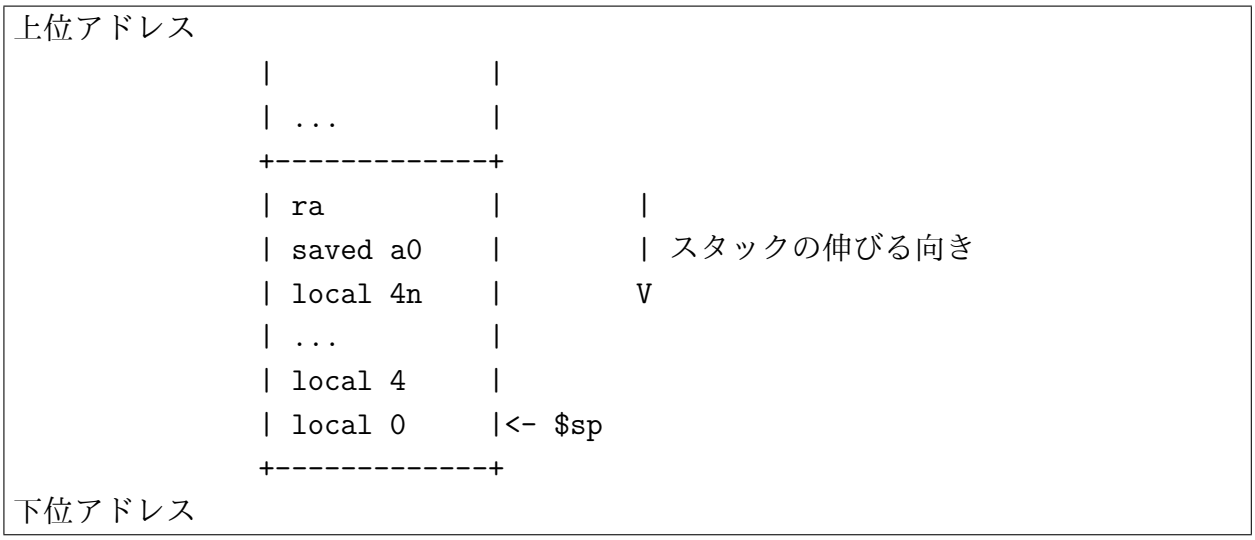


図 5.1: フレームの中身.

理系との互換性を気にする場合には，この資料に書かれている説明を理解した後に各自でさらに勉強して欲しい（「MIPS calling convention」などと検索すれば，たくさんの情報を得られる）.

ML^4 コンパイラにおける呼出し規約　まず，関数の返り値については先程触れたように，v0 レジスタを介して受け渡すことにする．また，ML^4 言語におけるすべての関数は引数を 1 つしか受け取らないため，その唯一の引数は a0 レジスタを介して受け渡すことにする.

各関数を実行するために必要なスタック上の連続領域（**フレーム** (*frame*) と呼ぶ）は，図 5.1 に示すように，スタックの一番上（メモリアドレスは下位の方）に確保される.

フレーム中に含まれる各データの説明は以下のとおりである.

リターンアドレス（ra）: このフレームを使っている関数を呼び出した関数が，その関数呼出しの次に実行する命令のアドレスを退避するための場所．さらに別の関数を呼ぶことがなければ必ずしも退避する必要はないが，簡単化のため，必ずフレームの一番上に退避することにしている.

退避された a0 レジスタ（saved a0）: このフレームを使っている関数が実引数を a0 レジスタにセットして他の関数を呼び出す際，すでに a0 レジスタに入っている自分自身のパラメータを退避するための場所．関数呼出し以降にパラメータを使わないのであれば必ずしも退避する必要はない（が，次節のコード生成では，簡単化のため必ず退避することにしている）.

局所変数（local 0〜4n）: 関数本体で let により束縛する局所変数の値を格納するための場所．$\mathcal{VT}$ で $\mathbf{local}(0)$ を関数からの返り値を格納する場所として使用したので，それに則って，局所変数の個数が N 個ならば局所変数を格納するための領域が $4N + 4$ バイトになっている.

なお，sp レジスタ（スタックポインタ）は，常にスタック最上部（フレームの一番下）を指すようにする．したがって，i 番目 $(1 \leq i \leq N)$ の局所変数が格納されている領域はアドレス$sp+$4i となる．また，saved a0 のアドレスは$sp+$4$N+4$，リターンアドレスのアドレスは$sp+$4$N+8$ となる．

以下は，上記のスタックレイアウトに基づいて関数呼出しを行う手順の概要である（詳細な手順については次節を参照）．

1. 呼出し側は，a0 レジスタの値を（自身の）saved a0 に退避してから，関数呼出しの実引数を a0 レジスタにセットする．
2. jal あるいは jalr 命令によって呼び出される関数の先頭へジャンプする．呼出し側の次の命令のアドレスは ra レジスタにセットされる．
3. （以下，呼び出された関数側で実行）sp レジスタを必要なだけ下げる．
4. ra レジスタに入っているリターンアドレスを，スタックの所定の位置に退避する．
5. 関数本体を実行し，求まった返り値を v0 レジスタにセットする．
6. スタック中のリターンアドレスをレジスタに戻す．
7. 3 で下げたのと同じ分だけ sp レジスタを上げることで，スタックからフレームを取り除く．
8. jr 命令によって 6 で取り出したリターンアドレスへリターンする．
9. （呼出し側で実行）v0 レジスタから返り値を取り出し，さらに，退避しておいた saved a0 を a0 レジスタに戻す．

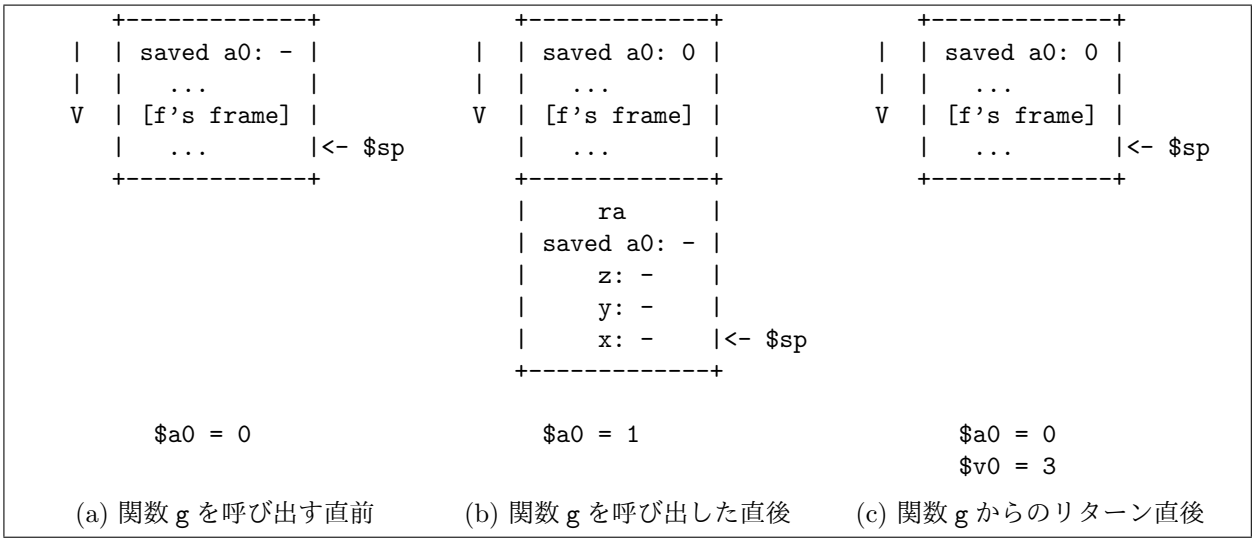
最後に，以下の簡単な OCaml コード：

```
let rec f a = g (a+1)
and g b = let x = b + b in
          let y = x * x in
          let z = y - 1 in
            z
in f 0
```

の実行中，関数 f から関数 g を呼び出す前後のスタックの状態を図 5.2 に示す．

現実の呼出し規約に関する余談

一般のプログラミング言語では，多引数関数を定義できたり，引数の数が固定ではない（可変長引数と呼ばれる）関数を定義できたりする．また，関数本体の実行中に使用するメモリサイズを正確に求めることが難しいこともある（たとえ



(a) 関数 g を呼び出す直前　(b) 関数 g を呼び出した直後　(c) 関数 g からのリターン直後

図 5.2: ML^4 のスタックレイアウト

ば，スタック上に任意サイズのメモリ領域を確保できる C 言語の `alloca` 関数を使用した場合など）．そのような場合，物理レジスタに収まりきらない引数をスタックに置く必要がある．それに加え，実行中に位置が一定ではない `sp` レジスタからの相対アドレスを用いてスタック中のパラメータおよび局所変数へアクセスしようとすると，コンパイル時に求める必要のある相対アドレスの計算が複雑になる．

そこで，現実のコンパイラにおいては，`sp` レジスタとは別に，関数フレームの `sp` とは反対側（典型的には `ra` の入っているあたり）を常に指し続ける `fp` レジスタを別に用意しておき，パラメータや局所変数へのアクセスには `fp` レジスタからの相対アドレスを用いるのが一般的である．ただし，リターンアドレスと同様に，呼出し側の `fp` レジスタの値もスタック中に退避する手間がさらに必要となる．

5.6.2　アセンブリ生成

以上を踏まえて，$\mathcal{V}$ のプログラムを MIPS アセンブリに変換する関数 Asm の定義を図 5.3 に示す．なお，以下の説明では MIPS アセンブリ中の命令について逐一説明はしないので，必要であればリファレンス [] を参照されたい．

オペランドの変換は $Asm_r(op)$ で行う．この変換では「op に格納されている値をレジスタ r にロードする」MIPS アセンブリが生成される．各ケースの説明は以下の通りである．

$Asm_r(\mathbf{param}(1))$: $\mathbf{param}(1)$（関数の第一引数）をレジスタ r にロードする．現在の $\mathcal{V}$ では一引数関数のみが定義できるため $\mathbf{param}(1)$ についてのみ定義されている．関数の引数は関数呼び出し規約からレジスタ `$a0` に格納されているので，この内容を r にロードするために `move` 命令を用いている．

（$[\![op]\!]$ は $\mathcal{V}$ の op に対応する MIPS の命令，$[\![l]\!]$ はラベル名 l を MIPS アセンブリ内のラベルとして解釈できるようにした表現である．ただし，$[\![l_{main}]\!] = \texttt{main}$ とする．）

Definition of $Asm_r(op)$

$$
\begin{array}{rcl}
Asm_r(\mathbf{param}(1)) & = & \texttt{move}\ \ r, \$\texttt{a0} \\
Asm_r(\mathbf{local}(n)) & = & \texttt{lw}\ \ r, n(\$\texttt{sp}) \\
Asm_r(\mathbf{labimm}(l)) & = & \texttt{la}\ \ r, [\![l]\!] \\
Asm_r(\mathbf{imm}(n)) & = & \texttt{li}\ \ r, n
\end{array}
$$

Definition of $Asm_n(i)$

$$
\begin{array}{rcl}
Asm_n(\mathbf{local}(ofs) \leftarrow op) & = & Asm_{\$\texttt{t0}}(op) \\
 & & \texttt{sw}\ \ \$\texttt{t0}, ofs(\$\texttt{sp}) \\
Asm_n(\mathbf{local}(ofs) \leftarrow op(op_1, op_2)) & = & Asm_{\$\texttt{t0}}(op_1) \\
 & & Asm_{\$\texttt{t1}}(op_2) \\
 & & [\![op]\!]\ \ \$\texttt{t0}, \$\texttt{t0}, \$\texttt{t1} \\
 & & \texttt{sw}\ \ \$\texttt{t0}, ofs(\$\texttt{sp}) \\
Asm_n(l :)) & = & [\![l]\!]: \\
Asm_n(\mathbf{if}\ op\ \mathbf{then\ goto}\ l) & = & Asm_{\$\texttt{t0}}(op) \\
 & & \texttt{bgtz}\ \ \$\texttt{t0}, [\![l]\!] \\
Asm_n(\mathbf{goto}\ l) & = & \texttt{j}\ \ [\![l]\!] \\
Asm_n(\mathbf{local}(ofs) \leftarrow \mathbf{call}\ op_f(op_1)) & = & Save_{n+4}(\$\texttt{a0}) \\
 & & Asm_{\$\texttt{a0}}(op_1) \\
 & & Asm_{\$\texttt{t0}}(op_f) \\
 & & \texttt{jalr}\ \ \$\texttt{ra}, \$\texttt{t0} \\
 & & \texttt{sw}\ \ \$\texttt{v0}, ofs(\$\texttt{sp}) \\
 & & Restore_{n+4}(\$\texttt{a0}) \\
Asm_n(\mathbf{return}(op)) & = & Asm_{\$\texttt{v0}}(op) \\
 & & Epilogue(n) \\
 & & \texttt{jr}\ \ \$\texttt{ra}
\end{array}
$$

Definition of $Asm(d)$

$$
\begin{array}{rcl}
Asm((l \mid i_1 \ldots i_m \mid n)) & = & [\![l]\!]: \\
 & & Prologue(n) \\
 & & Asm_n(i_1) \\
 & & \ldots \\
 & & Asm_n(i_m)
\end{array}
$$

Definition of $Asm(P)$

$$
\begin{array}{rcl}
Asm((d_1 \ldots d_m \mid i_1 \ldots i_n \mid k)) & = & \texttt{.text} \\
 & & \texttt{.globl main} \\
 & & Asm(d_1) \\
 & & \ldots \\
 & & Asm(d_m) \\
 & & [\![l_{main}]\!]: \\
 & & Prologue(k) \\
 & & Asm_k(i_1) \\
 & & \ldots \\
 & & Asm_k(i_n)
\end{array}
$$

図 5.3: アセンブリ生成の定義.

（$[\![op]\!]$ は $\mathcal{V}$ の op に対応する MIPS の命令名である.）

Definition of $Prologue(n)$

$$Prologue(n) \quad = \quad \begin{array}{l} \texttt{addiu} \quad \texttt{\$sp}, \texttt{\$sp}, -n-8 \\ Save_{n+8}(\texttt{\$ra}) \end{array}$$

Definition of $Epilogue(n)$

$$Epilogue(n) \quad = \quad \begin{array}{l} Restore_{n+8}(\texttt{\$ra}) \\ \texttt{addiu} \quad \texttt{\$sp}, \texttt{\$sp}, n+8 \end{array}$$

Definition of $Save_n(r)$

$$Save_n(r) \quad = \quad \texttt{sw} \quad r, n(\texttt{\$sp})$$

Definition of $Restore_n(r)$

$$Restore_n(r) \quad = \quad \texttt{lw} \quad r, n(\texttt{\$sp})$$

図 5.4: アセンブリ生成用補助関数の定義.

$Asm_r(\mathbf{local}(n))$: $\mathbf{local}(n)$ に格納されている値をレジスタ r にロードする．$\mathbf{local}(n)$ は関数呼び出し規約から $n(\texttt{\$sp})$ に格納されているので，これをレジスタ r にロードするために `lw` 命令を用いる.

$Asm_r(\mathbf{labimm}(l))$: コード中のラベル l のアドレスを r にロードする.[2]このために `la` 命令を用いている．$[\![l]\!]$ はラベル l を MIPS 内で解釈できる記号に変換したものである.

$Asm_r(\mathbf{imm}(n))$: 整数定数 n をレジスタ r にロードする．これは `li` 命令を用いて実装することができる.

命令の変換を行う関数 $Asm_n(i)$ は，$\mathcal{V}$ の命令 i を，局所変数用に n バイトを使う関数の内部にあると仮定して実行する MIPS の命令列を生成する．この n はフレーム内に格納されている値にアクセスする際に，そのアドレスの $\texttt{\$sp}$ からのオフセットを計算するために用いられる．各ケースの説明は以下の通りである.

$Asm_n(\mathbf{local}(ofs) \leftarrow op)$: この命令は「op に格納されている値を $\mathbf{local}(ofs)$ に格納する」ように動作する．そのためにまず op の値を求め，一時レジスタ $\texttt{\$t0}$ に格納する命令を生成し（$Asm_{\texttt{\$t0}}(op)$）その後 $\texttt{\$t0}$ に格納されているアドレスからレジスタ r に値をロードする命令 $\texttt{sw} \quad \texttt{\$t0}, ofs(\texttt{\$sp})$ を生成する.

$\mathbf{local}(ofs) \leftarrow op(op_1, op_2)$: op_1 に格納されている値をレジスタ $\texttt{\$t0}$ に（$Asm_{\texttt{\$t0}}(op_1)$），$op_2$ に格納されている値をレジスタ $\texttt{\$t1}$ に（$Asm_{\texttt{\$t0}}(op_1)$）それぞれロードする．その上で，レジスタ $\texttt{\$t0}$ の値とレジスタ $\texttt{\$t1}$ の値を引数として演算子 op によって計算し，その結

[2]このようにレジスタにコード中のアドレスをロードすることで，コード中の「場所」を値として保持することが可能となる．これは高階関数の実装で必要になる.

果を $t0 にロード（$[\![op]\!]$ $t0, $t0, $t1）する．定義を簡潔にするために，演算子 op に対応する MIPS の命令を $[\![op]\!]$ で表し，具体的に使わなければならない命令を $[\![-]\!]$ の定義の中に押し込めている．(例えば $[\![+]\!] = \texttt{addu}, [\![-]\!] = \texttt{mulou}$ とすればよい.）最後にレジスタ $t0 の値を **local**($ofs$) にストア（sw $t0, ofs($sp)）している．ofs バイト目のローカル変数のアドレスが $sp + ofs であることに注意せよ．

l : : ラベル $[\![l]\!]$ を生成（$[\![l]\!]$:）している．ここで $[\![l]\!]$ はラベル l を MIPS アセンブリ内でラベルとして解釈できる識別子に変換したものである．この変換は一対一対応でさえあればどのように定義しても良いが，メインのプログラムを表すラベル l_{main} は，MIPS アセンブリ内のエントリポイント（プログラムの実行時に最初に制御が移される場所）を表す main というラベル名に変換する必要がある．

Asm_n(**if** op **then goto** l) : まず op に格納されている値をレジスタ $t0 に格納する．その上で，レジスタ $t0 が true を表す非ゼロ値であれば $[\![l]\!]$ にジャンプ（bgtz $t0, $[\![l]\!]$）する．

Asm_n(**goto** l) : 無条件でラベル $[\![l]\!]$ にジャンプ（j $[\![l]\!]$）する．

Asm_n(**local**(ofs) $\leftarrow$ **call** $op_f(op_1)$) : 関数呼び出しを行う際には，関数呼び出し規約に従ってレジスタの内容を退避・復帰したり，引数をセットしたり，返り値を取得したりしなければならない．今回のコンパイラにおいては，関数の呼び出し側では，3 ページで説明した通り，(1) レジスタ a0 の値の退避，(2) レジスタ v0 に格納されている返り値の取得，(3) 退避しておいたレジスタ a0 の値の復帰を行う必要がある．レジスタ値の退避・復帰を行う命令列は他のケースでも使用するので，それぞれテンプレ化して図 5.4 に「アドレス $sp + n にレジスタ r の内容を退避する命令列 $Save_n(r)$」と「アドレス $sp + n に退避したレジスタ r の内容を復帰する命令列 $Restore_n(r)$」として定義してある．$Save$ と $Restore$ を使うと，関数呼び出し前に実行されるべき命令列は以下の通りとなる．

1. レジスタ $a0 をメモリ上のアドレス $sp + $n + 4 に退避 ($Save_{n+4}$($a0)) する．$a0 は今から行う関数呼び出しのための実引数で上書きされるからである．
2. op_1 に格納されている実引数を $a0 にロードする．
3. op_f に格納されているラベル (=コード上のアドレス) を $t0 にロードする．
4. jalr 命令を使って $t0 に格納されているラベルにジャンプする．jalr 命令の第一引数 $ra には，ジャンプ先からリターンするときに帰ってくるべきコード上のアドレス (=この命令の次の行) がセットされる[3]．

この次の行からは，この後呼び出された関数が実行されリターンした後に実行されるべき命令列が書いてある．

[3]なので，$ra はこの命令の実行前にどこかに退避されていなければならないが，これは関数定義のアセンブリ生成のところで説明する．

1. レジスタ $v0 に格納されているはずの（関数呼び出し規約を参照のこと）リターンされた値を **local**(ofs)，すなわち $sp + ofs に sw 命令を使ってストアする．
2. $sp + $n + 4$ に呼び出し前に退避しておいたレジスタ $a0 の内容を復帰させる ($Restore_{n+4}$($a0))．

以上の命令列が実際に正しく関数呼び出しを実行することを確認するためには，関数呼出し時の命令のみではなく，リターン命令 (Asm_n(**return**(op))) や関数定義側でどのような命令列が生成されるかも確認することがある．前者についてはすぐ，後者については後で $Asm(d)$ の定義を説明する際にそれぞれ説明する．

return(op) : op に格納されている値を呼び出し側に返さなければならない．関数呼び出し規約によれば，関数がリターンする前には以下の処理を行う必要がある: (1) 返り値をレジスタ $v0 にロード, (2) 関数の先頭でフレーム内に退避しておいた $ra の値を復帰, (3) $sp レジスタの値を先頭で下げた分だけ上げる (すなわち，現在のフレームに使っていたスタック上の領域を解放する), (4) jr 命令を用いて $ra に格納されたアドレスにリターンする．具体的には以下の命令列が生成される:

1. op に格納されている値を返り値を格納すべきレジスタ ($v0) にロード ($Asm_{\$v0}(op)$) する．
2. $ra の値の復帰とフレームの解放を行う．定義中では $Epilogue(n)$ でこの処理を行う命令を生成している．$Epilogue(n)$ はローカルな記憶領域のサイズが n のフレームを持つ関数呼び出しのリターン前の処理を行う命令列で，退避しておいた $ra の復帰 $Restore_{n+8}$($ra) と，$sp の値の更新を行う．
3. jr 命令で復帰した $ra にリターンする．

関数定義 ($l \mid i_1 \ldots i_m \mid n$) に対応する命令列 $Asm((l \mid i_1 \ldots i_m \mid n))$ は以下のアセンブリを生成する．

1. この関数のラベルを生成する ($[\![l]\!]$:)．
2. 関数本体の先頭で行わなければならない処理を行う命令列を生成する．関数呼び出し規約によれば以下の処理を行う必要がある:
   (a) レジスタ $sp の値を更新して今から使うフレームを確保する．
   (b) レジスタ $ra の値をフレーム内の所定の位置に退避する．

   以上の処理を行うための命令列を $Prologue(n)$ として図 5.4 に定義している．ここで n はこの関数内で使用する局所変数のための記憶領域のサイズである．$Prologue(n)$ は初めにフレームのサイズ分 $sp の値を addiu 命令を用いて減らす．フレームのサイズは，(局所変数用領域)+($ra 退避先用の領域 4 バイト)+($a0 退避先用の領域 4 バイト) なので $n + 8$ である．その後，$ra をフレーム内の所定の場所に退避している．

最後に，プログラム $(d_1 \ldots d_m \mid i_1 \ldots i_n \mid k)$ のアセンブリ生成の定義を説明しよう．まず先頭で，以降にかかれている情報がプログラムである旨を示す `.text` ディレクティブを生成している．その次の行には，アセンブリ中の `main` というラベル名がグローバルなラベル，すなわち外部から見える名前であることが宣言されている．その後 d_1 から d_m までのアセンブリを順番に生成した後に，$[\![l_{main}]\!]$: に続いて，メインのプログラムを実行するためのフレームの確保を $Prologue(k)$ で行い (k はフレームのサイズ)，命令列 $i_1 \ldots i_n$ に対応するアセンブリを生成する．