はじめに

本書をお手に取って頂いてありがとうございます. 本書は Plan9 のアセンブラの解説書です.

Plan 9 のプログラミング環境といえば, ミドルボタンが必要なエディタ Acme からはじまり stdio.h を include しない Hello World の C 言語コンパイラ 8c, このご時世に動的リンクが存在しないライブラリ環境 e.t.c. と特異なことをご存じの方もいるでしょう. その例に漏れず, 同じアセンブラといえど Plan 9 のアセンブラは gas や NASM と異なる文化を持っています.

Plan 9 のマシン依存部, pbs(PC Partition Boot Selector) や 9load などの古式ゆかしき x86 向けブートローダ, 9front に最近入った UEFI 対応のローダなど様々な低レイヤの面白い箇所にこのアセンブラは登場します.

本書ではいわゆる x86 32bit (i386) 向けの 8a, x86 64bit (amd64) 向けの 6a をメインにこのアセンブラの独特の文法や作法などを紹介します. 底本として Rob Pike 著 A Manual for the Plan 9 assembler を引きつつ実際のコードを提示してこれを解説します. x86/64 自体の命令セット・アーキテクチャについては本書では触れません.

6a は昨今 Go 言語のコンパイラを含むビルドシステムにて用いられており, この解説文章として「Go アセンブラ入門」(http://golang-jp.org/doc/asm) が日本語で読める資料として存在します. またその他にも @oraccha さんの Plan 9 日記 (http://d.hatena.ne.jp/oraccha/) にもいくつか記事が存在します. 併せてこのあたりをご参照いただくと良いかと思います.

目次

第 1 章

Plan 9 におけるアセンブラ

1.1 Plan 9 でサポートされるアーキテクチャ

Plan 9 では MIPS, SPARC, i960, amd 29000, Motorola 68020, 68000, ppc, alpha, arm, i386, amd64 など様々なアークテクチャをサポートしています[*1].

コンパイラなどプログラムのビルドに関わるツールは**「アーキテクチャを示す 1 文字 + ツールの種類 1 文字」**という規則で命名されています. アーキテクチャを示す 1 文字には主に数字 (たまにアルファベット) が用いられます[*2]. それぞれの文字が対応するアーキテクチャを以下に列挙します.

- 0: MIPS 3000 (リトルエンディアン), 消滅
- 1: Motorola MC68000
- 2: Motorola MC68020
- 5: ARM(リトルエンディアン)
- 6: amd64 (かつては i960 用だった@ Plan 9 本家)
- 7: Alpha, 消滅
- 8: Intel i386, i486, Pentium, e.t.c
- 9: 64bit PowerPC, AMD Am29000, 消滅
- k: Sun SPARC
- q: PowerPC
- v: MIPS 3000 (ビッグエンディアン)

消滅 と末尾に付記したアーキテクチャは, 各種ドキュメントにその記録があるものの 9front や Plan 9 4th Edition ではソースコード (src/cmd/ の下) からコンパイラ等が消えてしまっているものです.

実際のところ, 9front カーネルがサポートしているアーキテクチャ/ターゲットボードは以下の 11 種類です. すでに 4th Edition の時点で SPARC や Alpha, MIPS のサポートはなくなり, 代わりに Raspberry Pi など新しめのボードのサポートが追加されているのが見て取れます.

- bcm: ARM (5a) Broadcom (Raspberry Pi) 用

[*1] ここに並べた大半は 「かつてはしていました」になってしまいましたが...

[*2] アスタリスク代わりに 'O' で代替表記することもあります

- kw: ARM (5a) Marvell Kirkwood (Sheevaplug など) 用
- mtx: PowerPC (qa)
- omap: ARM (5a)
- pc : i386 (8a)
- pc64: amd64 (6a)
- ppc: PowerPC (qa)
- sgi: MIPS (va)
- teg2: ARM (5a) NVIDIA Tegra2 (Compulab Trim-Slice シリーズ用)
- xen: i386 (8a) Xen 仮想化環境用
- zynq: ARM (5a) Zynq-7000 用

これらのターゲットに併せて適切なプレフィクスをもつツールを選択する必要があります.

1.2 Plan 9 プログラミング環境でのアセンブラ

プログラミング環境というと大げさですが, プログラマが書いたコードを実行形式に落とし込むためのツールとして, 他のプラットフォームのそれと同じく C 言語のコンパイラ, アセンブラ, ローダ (リンカ) がそれぞれ用意されています.

先に述べたとおりこれらのツールは**「アーキテクチャを示す 1 文字 + ツールの種類 1 文字」**の名前が付けられています. ツールの種類 1 文字としてコンパイラは **c**, アセンブラは **a**, そして ローダは **l** を用います. この規則に従い, i386 では C コンパイラは **8c**, アセンブラは **8a**, ローダは **8l** となります.

使い方も特に UNIX 系の OS での gcc や cc の使い方とは変わらず, リスト 1.1 や リスト 1.2 に記載したようにオブジェクトファイル (標準で ***.O** に出力される) に落とし込み, 次にローダ (リンカ) で実行ファイルとして出力するといった流れです.

リスト 1.1: コンパイラでのコンパイル

```
/* コンパイラの実行: hello.c からオブジェクトファイル hello.8 を生成する */
% 8c  hello.c

/* ローダの実行: hello.8 から実行ファイル hello_world を生成する */
% 8l  - o  hello_world  hello.8
```

リスト 1.2: アセンブラの利用例

```
/* アセンブラの実行: hello.S からオブジェクトファイル hello.S.8 を生成する */
$ 8a  hello.S

/* ローダの実行: hello.S.8 から実行ファイル hello_world を生成する */
$ 8l  -o hello_world  hello.S.8
```

特異なところとしては他のプラットフォームではリンカと呼称する役割のツール 8l がローダと呼ばれるところでしょうか. 役割としてはリンカと同じくオブジェクトファイルまたはライブラリのリンク, そして実行ファイルの形成と出力を行います. ローダという名前は, 各オブジェクトファイル内のコードを実行ファイル内へとロード, 適切に配置していくという処理を元に付けられています.

ローダはこのように実行形式を形作るにあたり以下の様なことも行っています. まずローダでは分岐を畳んだり, 部分的にコピーしてこれを消したり, 不到達コードを排除したりといった最適化を行う場合があります. また NOP 疑似命令を消し飛ばすといったことも行います. これは Plan 9 での NOP 疑似命令 (機械語での NOP にあらず) は「何もしない」ではなく「命令がない」ことを示すために定義されていることに起因します.

第 2 章

Plan 9 アセンブラのお作法

この章では, Plan 9 のアセンブラのお作法について見ていきます. 基本的には gas や NASM と同じ立ち位置のツールなのでそれらと役割やできることに大きな差があるわけではありません. なのでアセンブラとは, といったお話はしません. また細かい文法や命令セットに踏み込むと切りがないため, ここでは Plan 9 アセンブラに特有と思われる部分や面白そうな部分に絞って見ていきます.

それぞれのアーキテクチャで使えるレジスタ, 定義されているオペランド, およびそれらの使い方はもちろんそれぞれに異なります. Plan 9 のアセンブラではこれらのアーキテクチャの差異を部分的に抽象化, ある程度共通化しおよそ同じような記述で利用できるようにしています. このために記法の共通化や, 疑似命令, 疑似レジスタなどを定義しています.

2.1 共通のお作法

命令やレジスタ, アドレッシングについては後述しますが以下に列挙するような共通のお作法が存在します.

1. 予約済みのシンボルはすべて大文字
2. ラベルやユーザ定義のシンボルは基本的に小文字
3. 即値は **$** (ドルマーク) を先頭に付ける
4. コメントは C 言語での記法を用いる
5. **;** (セミコロン) は区切り文字として利用
6. データの転送方向は基本的に左から右

レジスタや各アーキテクチャでサポートされるニーモニックなどの予約語は全て大文字で定義されています. たとえば疑似命令の MOV や レジスタ R0, AX などは小文字での表記はできません. 一方でラベルやサブルーチンの名前, 外部参照や局所参照に用いられるシンボルは小文字で命名します.

即値は先頭に **$** (ドルマーク) を付与し '$100' といった形で用います. この場合は 10 進数ですが '$0x56' のように 16 進数による記述も可能です. また即値内では四則演算や AND, OR などによる計算を行うこともできます (リスト 2.1). まためったに使われませんが浮動小数点もサポートしています.

リスト 2.1: 演算子の利用例 (src/9/pc64/l.s)

```
/* 第 1 オペランド: 即値として即値同士の計算式を記述 */
        ADDL    $(16*1024), CX   /* qemu puts multiboot data after the kernel */

/* 第 1 オペランド: もちろんマクロ同士も可 */
        MOVQ    $(CPUOMACH+MACHSIZE), SP

/* 第 1 オペランド: もちろんマクロ同士も可. OR や シフト演算も利用可 */
        LONG    $(SEGL|SEGG|SEGP|(0xF<<16)|SEGPL(0)|SEGEXEC|SEGR)

/* 第 1 オペランド: NOT の結果を即値として計算可 */
        ANDL    $~(BY2PG-1), DI
```

この例にもありますが, コメントは gas のそれと同じく C 言語の "/* ... */" または "// ..." の両方を使うことができます. 一方で gas の "#" (ハッシュマーク), NASM の ";" (セミコロン) といった記法はサポートしていません. 後者のセミコロンは Plan 9 アセンブラでは C 言語のそれと同じく文あるいは式の区切り文字として用います.

gas は AT&T 記法, NASM は インテル記法をベースにしています. Plan 9 では AT&T 記法つまり「ニーモニック 転送元 転送先」の並びを基本としています. 前章で述べたとおり, 様々なアーキテクチャ向けのアセンブラが存在しますが一部の命令を除きこれを保つようアセンブラで命令の抽象化などが行われています.

2.2 命令

個々のアーキテクチャのニーモニックを取り上げると切りがないため, ここでは Plan9 アセンブラでの疑似命令およびアセンブラにてサポートされない命令の実装方法に絞って記述します.

2.2.1 疑似命令

Plan 9 のアセンブラコード中に良く出てくる疑似命令として, ここでは MOV, BYTE (およびその仲間), DATA, GLOBL の 4 種を取り上げます.

MOV

MOV はレジスタ, アドレスを対象とした値の転送を行うための疑似命令です. アーキテクチャにより即値, レジスタ, アドレスを転送元, 転送先とした場合のオペレーションは異なります. Plan 9 アセンブラでは MOV 疑似命令によりこれらを抽象化し **MOVx src, dst** との記法で共通に行えるようにしています. 先にも述べたとおり AT&T 記法をベースとしているため「転送元 to 転送先」の順序で記述します. MOV の末尾にはサフィクスとして転送するサイズを表す 1 文字を付記します. 1 バイトは **B** (Byte), 2 バイトは **W** (Word),4 バイトは **L** (Long), 8 バイトは **Q** (Quad) を用います. なおレジスタのビット幅と MOV 疑似命令で指定される幅が一致しない場合は, より狭い方のサイズの転送と見なして MOV の幅が自動的に変わります. 例として リスト 2.2 における MOVW の利用例では, 32bit 幅のレジスタである AX に対して 2 バイト (WORD) の転送を行おうとしています. この場合, AX の下 2

バイト (AL 分) のみが転送されます. 転送先には下 2 バイトが反映される一方で上 2 バイトは 0 クリアされます.

リスト 2.2: MOV の利用例 (src/9/pc/l.s)

```
# MOVW の利用例 : AX から DS に下 2 バイト転送
        MOVW    AX, DS

# MOVL の利用例 : DX から CR0 に 4 バイト転送
        MOVL    DX, CR0                         /* turn on paging */
```

BYTE, WORD, LONG, QUAD

これらの命令はそれぞれ 1, 2, 4, 8 バイトのデータを命令列の中に配置する疑似命令です. 単純にあるメモリの位置に値を格納したい場合は前述の MOV, データセクション中であれば後述の DATA を使います. この命令はプログラム中のこの場所に値を配置することを目的としています. 主な用途としては各種テーブル, たとえば割り込みベクタや GDT, マルチブートヘッダなどの構造を定義するために用いられます (リスト 2.3). また別の使い方として後述するように, ニーモニックが未実装の命令を呼び出すために用いることがあります.

リスト 2.3: LONG によるマルチブートヘッダの実装 (src/9/pc/l.s)

```
/*
 * Must be 4-byte aligned.
 */
TEXT _multibootheader(SB), $0
        LONG    $0x1BADB002                     /* magic */
        LONG    $0x00010003                     /* flags */
        LONG    $-(0x1BADB002 + 0x00010003)     /* checksum */
        LONG    $_multibootheader-KZERO(SB)     /* header_addr */
        LONG    $_startKADDR-KZERO(SB)          /* load_addr */
        LONG    $edata-KZERO(SB)                /* load_end_addr */
        LONG    $end-KZERO(SB)                  /* bss_end_addr */
        LONG    $_multibootentry-KZERO(SB)      /* entry_addr */
        LONG    $0                              /* mode_type */
        LONG    $0                              /* width */
        LONG    $0                              /* height */
        LONG    $0                              /* depth */
```

DATA

DATA はデータセクションに値を置く疑似命令です. 「**DATA 名前/バイト数, 値**」といった記法で利用することができます.「**DATA array+0(SB)/4, $"abc\z"**」 と書いた場合は, "main" という文字列データの配置された領域をデータセクションに確保し, array というシンボルを付与して参照, アクセスできるようにしています. スラッシュ (/) の後ろに指定できるバイト数の上限は 8 までとなっています. 実際の利用例をリスト 2.4 に引用します. この

コードは libc が 各プログラムを main 関数経由で起動する部分および終了後の処理を記述しています. プログラムが終了すると最終的に exits 関数を呼び終了処理を行いますが, この時に引数としてシンボル $_exits により示される "main\z" の格納された領域のポインタを渡しています.

GLOBL

GLOBL は シンボルをグローバルにする疑似命令です. 「**GLOBL 名前, バイト数**」 という記法で利用されます. 基本的には前述の DATA 疑似命令の直後に配置され, シンボルが指す領域を外部から参照できるものとして宣言します. この宣言には第 2 引数にてサイズを付与することができます. 通常はシンボルにて参照できる領域が DATA 疑似命令により事前に初期化されていることを期待しますが, そうでない場合は指定されたサイズに従って 0 クリアします. リスト 2.4 の例では, DATA では "main" と 4 バイト分の初期値を埋めています. ここにかぶせる形で GLOBL では 5 バイトとしています. これにより, DATA 疑似命令側で "\z" を追加しなくても NULL 終端するようにしています.

リスト 2.4: DATA および GLOBL の利用例 (src/libc/386/main9.s)

```
#define NPRIVATES       16

TEXT    _main(SB), 1, $(8+NPRIVATES*4)
        MOVL    AX, _tos(SB)
        LEAL    8(SP), AX
        MOVL    AX, _privates(SB)
        MOVL    $NPRIVATES, _nprivates(SB)
        MOVL    inargc-4(FP), AX
        MOVL    AX, 0(SP)
        LEAL    inargv+0(FP), AX
        MOVL    AX, 4(SP)
        CALL    main(SB)

loop:
        MOVL    $_exits<>(SB), AX    /* "main" の書かれた領域のアドレスを AX に */
        MOVL    AX, 0(SP)            /* アドレスをスタックに push */
        CALL    exits(SB)            /* exits 関数の呼び出し */
        JMP     loop

DATA    _exits<>+0(SB)/4, $"main"    /* 局所シンボル _exits<> を定義，初期値設定 */
GLOBL   _exits<>+0(SB), $5           /* _exits<> の領域を外部からアクセス可能に */
```

これらの他に TEXT なる疑似命令も存在しますが, これについてはサブルーチンの節にて後述します.

*1 **"\z"** は NULL 終端 ("\0") を意味します

2.2.2 サポートされていないニーモニックの利用

アセンブラでサポートされているアーキテクチャ依存の命令は, その一覧がそれぞれのコンパイラのソースディレクトリ直下に置かれている **O.out.h** なるファイルに列挙されています. i386 の場合は **src/cmd/8c/8.out.h** になります. 実際のバイナリとしての命令との対応表は Optab 構造体として各ローダにて定義されています. i386 の場合は **src/cmd/8l/optab.c** をご参照ください.

この一覧は, アーキテクチャリファレンスに完全には追従はできておらずニーモニックとして定義されていない命令が存在します. このような場合に Plan 9 では先に述べた BYTE 疑似命令を使ってオペコードのバイナリを手で打って未サポートの命令を呼び出すといったテクニックを多用しています. リスト 2.5 にこの例を記載します.

リスト 2.5: touser での 未実装命令 SYSRETQ の実装 (src/9/pc64/l.s)

```
# touser での SYSRETQ の実装
TEXT touser(SB), 1, $-4
        CLI
        SWAPGS

        MOVL    $0, RMACH
        MOVL    $0, RUSER

        MOVQ    $(UTZERO+0x28), CX              /* ip */
        MOVL    $0x200, R11                     /* flags */
        MOVQ    RARG, SP                        /* sp */

        BYTE $0x48; SYSRET                      /* SYSRETQ */
```

この例では最終行の **BYTE $0x48; SYSRET** という表記でニーモニックが定義されていない SYSRETQ を実装しています. SYSRET は 6l でサポートされているニーモニックでオペコードとしては **0x0f 0x07** なるバイナリ列です. 一方 SYSRETQ は **0x48 0x0f 0x07** というバイト列としてアーキテクチャマニュアルに記載されていますが 6a のニーモニックとしては定義されておらず変換表にも載っていません. ここで BYTE 疑似命令を用います.

先に述べたとおり BYTE などの疑似命令は命令列の中に値を置くための命令です. 最終的な成果物のバイナリでは, CPU がそれと解釈できるバイナリになっていれば未実装命令であれ実行させることができます. この例では SYSRETQ のバイナリ列だと思わせるにに必要な **0x48** を BYTE を用いて SYSRET 命令の前に挿入し実質的に SYSRETQ を呼び出すようにしています.

2.3 レジスタ (疑似レジスタ)

レジスタの構成および名前はアーキテクチャごとに固有のため, 詳細については各アーキテクチャのマニュアルに説明を譲ります. ここでは Plan 9 にて用意されている疑似レジスタについて触れます.

疑似レジスタは, どのアーキテクチャでも同じ名前で同じ役割をもち同じ使い方ができるようにア

センブラおよびローダが提供する擬似的なレジスタです. 種類としては **SP, FP, PC, SB** の 4 つが利用可能となっています.

アーキテクチャによっては, 一部は別名で存在していたりまた一部はそもそも存在しなかったりと様々です. この部分をローダとアセンブラが隠蔽して擬似的に等価な機能を提供するようになっています. たとえば x86 には FP がありませんが ebp が存在しています.

それぞれの機能を以下に解説します.

SP (Stack Pointer)

現在のスタックの先頭位置のポインタを格納するレジスタです.i386 では ESP に相当し, PUSH/POP 操作時に増減します. 一時領域として, ポインタを引き渡して値を取得するタイプの命令向けにテンポラリな値の格納場所として用いられる場合もあります. リスト 2.6 の例では x87 FPU から制御レジスタとステータスレジスタの値を読み出してくるためにスタックの先頭を初期化して引き渡しています. また サブルーチンを呼び出す際に引数を格納するためにも用いられます.

リスト 2.6: getfcr での SP の利用例 (src/libc/386/getfcr.c)

```
TEXT    getfcr(SB), $4
        MOVW    AX, 0(SP)
        WAIT
        FSTCW   0(SP)         /* 一時領域としてスタックの先頭を利用 */
        MOVW    0(SP), AX
        XORB    $0x3f,AX
        RET
```

FP (Frame Pointer)

現在のフレームの先頭位置のアドレスを格納します. 後述するアドレッシングの節でも取り上げるように **0(FP)**, **4(FP)** といった記法を用いて, 現在のフレームにスタック経由で引き渡される引数を参照することができます. リスト 2.7 は outsl 関数にて, フレームポインタから引き渡されるはずの 3 つの引数 (ポート番号, アドレス, カウント数) を取り出している例です. コード中に **port, address, count** といった変数のような名前が登場しますが, こちらについては後述する「アドレッシング」をご参照ください.

リスト 2.7: outsl での FP の利用例 (src/9/pc/l.s)

```
TEXT outsl(SB), $0
        MOVL    port+0(FP), DX     /* 第 1 引数: ポート番号 */
        MOVL    address+4(FP), SI  /* 第 2 引数: アドレス   */
        MOVL    count+8(FP), CX    /* 第 3 引数: カウント数 */
        CLD
        REP;    OUTSL
        RET
```

PC (Program Counter)

PC は現在実行中の命令のアドレスを格納するレジスタです. i386 では EIP がこれに対応します. 2a などでは BRA (BRanch Always) 命令にて特殊な使い方ができますが, i386/amd64 ではあまり活用されていないため詳細は省略します.

SB (Static Base)

SB は現在のプログラムのアドレス空間の先頭アドレスを格納するレジスタです. これまでに挙げたアセンブラのコードの例にもちらほらとでてきました. あるプログラムのメモリ空間を考えると, 定義したサブルーチンやテキストセクションあるいはデータセクション中の値の場所 (アドレス) はそのプログラムのアドレス空間の先頭から数えてあるオフセット離れた位置に存在するメモリ上の一定サイズの領域と考えることができます. アセンブラでは便宜のために, このそれぞれに名前を付けることができます. 実体として「先頭 + ある名前に対応するオフセットの位置にあるサブルーチン」,「先頭 + ある名前に対応するオフセットの位置にある値」という考え方をするため, 先頭を意味する **SB** との表記が付いて回ります.

リスト 2.8: SB の利用例 (src/9/pc/l.s)

```
/*
 * サブルーチンのシンボルを定義する場合:
 * SB から数えてここの位置に bios32call なるシンボルを作り,
 * サブルーチンとして呼び出せるようにする
 */
TEXT bios32call(SB), $0
    ....
```

2.4 アドレッシング

Plan 9 のアセンブラでは MC68020 (2a) のアドレッシング記法をベースに, これを全アーキテクチャでも利用するようにしています. ここでは i386, amd64 のコードに登場する主要なアドレッシング記法に絞って記載します.

レジスタ: **REG** (**例**: AX, BX, PC)

物理, 疑似問わずレジスタに格納されている値を参照する場合はひらで書きます

即値: **$con** (**例**: $100, $(100-80), $0xFFFFFFFF)

即値を表現する場合は, 値の先頭に **$** (ドルマーク) を付与します. なお先にも述べたとおり 10 進数以外にも 16 進数や括弧で囲んだ計算式を書くことも可能です. また浮動小数点も記述できます.

オフセット付き間接参照: **o(REG)** (**例**: 0(AX), 4(AX))

レジスタに格納されている値をアドレスとして参照し, そこにオフセット o を足した場所を参照する記法です. MOV 疑似命令の対象として例の様に書くことで, 転送元として AX に格納されたアドレスから 0, 4 バイト目の領域に存在する値を取得したり, 転送先としてその位置に値を転送するといったことができます.

引数参照: **o(FP)** (**例**: 0(FP), 4(FP))

これは前述のオフセット付き間接参照の一部ですが, 特にサブルーチン内で引数を取得するために多用されます. この具体例は リスト 2.7 にて挙げたとおりです.

名前付き引数参照: **name+o(FP)** (**例**: address+4(FP))

フレームポインタから引っ張った引数に対して名前を付ける際に用いられる記法です. この例は同じく リスト 2.7 でも紹介しました. 実体は '+' より後ろの **4(FP)** と記述している部分, つまり "第 2 引数である" と記述している部分です. name の部分には人間が読む際にヒント情報となるように名前が付けられるようになっており, これを用いて port, address などと引数としての意味を提示しています. 次の自動シンボルと異なり, この名前は当該行以外からは参照できません.

自動シンボル: **name+o(SB), name(SB)** (**例**: address+4(FP), touser(SB), array+1(SB))

前述の引数の例と酷似していますが, こちらは SB からのオフセットを取る自動シンボル (automatic symbol) とよばれる記法です. SB 疑似レジスタの項でも述べた通り, TEXT や GLOBL で定義されたシンボルの位置から指定されたオフセット離れた領域にある値やサブルーチンを参照するために用いられます. 前述の引数参照とは異なり, この名前はグローバルに参照することができます. 例として TEXT での定義およびその呼び出しの記法をリスト 2.9 に, GLOBL との組合せを リスト 2.10 に記載します. 例にもあるようにオフセットが 0 の場合は **+0** との表記を省略可能です. また滅多登場しませんがにありませんが, マイナス方向にオフセットを取る場合 "-1" とすることも可能です.

リスト 2.9: サブルーチン rmode16 を定義した場合 (src/boot/pc/l.s)

```
/* サブルーチン rmode16 を定義 */
TEXT rmode16(SB), $0
    ...

/* rmode16 を呼び出し: オフセット 0 のため "+0" は省略可能 */
    CALL rmode16(SB)
```

リスト 2.10: 自動シンボルとして array を定義した場合

```
    /* データセクションの任意の位置に array として 4 バイトのデータをセット */
    DATA    array+0(SB)/4, $"abc\z"

    /* array として確保された領域を外部から参照できる様にする（4 バイト分）*/
    GLOBL   array(SB), $4

    /* array の先頭 0 バイト目から 1 バイトを AX に格納する（'a' だけが入る）*/
    MOVB    array(SB), AX

    /* array の先頭 2 バイト目から 1 バイトを AX に格納する（'b' だけが入る）*/
    MOVB    array+1(SB), AX
```

局所シンボル: name<>+o(SB) (**例**: _exit<>+0(SB))

局所シンボル (local symbol) はほぼ自動シンボルと同じような用途で用いられます. 記法上はシンボルの末尾に **"<>"** が付与されているという違いがあります. 機能上は「局所」の名前が示すとおりこのシンボルのスコープはこれが定義されているファイルの中のみに限定されるという違いがあります. 実装上はローダがこの "<>" の位置に, 重複しない任意の整数を勝手に割り当てるといったことを行い, これによりさもファイルごとに固有であるかのように見せています (たとえば _exit<>+0(SB) は _exit99+0(SB) のように変換される). GLOBL を用いた例については前述の リスト 2.4 における _exits<>+0(SB) の定義・利用が参考となります. TEXT による局所シンボルとしてのサブルーチン定義の例をリスト 2.11 に記載します.

リスト 2.11: 局所シンボル _protected<> の定義 (src/9/pc64/l.s)

```
/* 同一ファイル内からしか参照できない _protected<>(SB) の定義 */
TEXT _protected<>(SB), 1, $-4
    ....
```

アドレス参照: **$name(SB), $name+o(SB), $name<>+o(SB)** (**例**: $apmjumpstruct+0(SB))

アドレス参照は指定されたシンボルのアドレスを取得する記法です. 前述の自動シンボルではシンボルの位置に格納されている値を取得できました. この記法ではシンボルの指すアドレスを取得します. 局所シンボルのアドレス参照の例は リスト 2.4 にて登場していますが, 該当部分だけを リスト 2.12 に抜粋します. この例では **__exits<>+0(SB)** なるデータセクションに確保された領域の局所シンボルを定義していました.「アドレス参照」の部分でこれを参照しています. この例では, **"main"** の文字列が格納されている領域の先頭アドレスを取得, これを AX に転送しています.

リスト 2.12: 局所シンボルのアドレス参照 _exits<>(SB)

```
loop:
        MOVL    $_exits<>(SB), AX    /* アドレス参照 */
        MOVL    AX, 0(SP)
        CALL    exits(SB)
        JMP     loop

DATA    _exits<>+0(SB)/4, $"main"
GLOBL   _exits<>+0(SB), $5
```

2.5 プロシージャの定義

2.5.1 サブルーチンの定義

サブルーチンやテーブルのエントリーポイントを宣言するためには **TEXT** 疑似命令を用います. これはシンボルを定義するものであり JMP 命令で対象となるラベルとは区別されます.

TEXT は通常 2 つ引数を取り, 第 1 引数に名前, 第 2 引数にサイズを指定できます. 第 1 引数の名前には CALL 命令などの対象として利用するシンボルを指定します. 第 2 引数のサイズはこのサブルーチンを呼ぶ際に, スタック中に自動的に確保してほしいバイト数を指定します.

リスト 2.13: TEXT を用いたエントリポイントの例: cas64 (src/libc/i386/atom.s)

```
/*
 * int cas64(u64int *p, u64int ov, u64int nv);
 */

/*
 * CMPXCHG64 (DI): 0000 1111 1100 0111 0000 1110,
 */
#define CMPXCHG64               BYTE $0x0F; BYTE $0xC7; BYTE $0x0F

TEXT    cas64+0(SB),0,$0
        MOVL    p+0(FP), DI
        MOVL    ov+0x4(FP), AX
        MOVL    ov+0x8(FP), DX
        MOVL    nv+0xc(FP), BX
        MOVL    nv+0x10(FP), CX
        LOCK
        CMPXCHG64
        JNE     fail
        MOVL    $1,AX
        RET
```

TEXT 疑似オペレーションは引数を 3 つ取る場合があります. このときは上記で述べた第 2 引数は第 3 引数に引き下がり, 代わりにビットフィールドを指定する引数が追加されます. このビットフィールドでは各ビットを立てることによりローダがこれを処理する際の挙動を変えることができます. 例として 1bit 目が立っていればプロファイリングの禁止, 2bit 目が立っていれば同一プログラム中に複数の同一 TEXT を定義できるといった塩梅です.

2.5.2 サブルーチンの呼び出し

前節にて定義したサブルーチンは TEXT 疑似命令の第 1 引数として指定したシンボルをもとに呼び出すことが可能です. アセンブラ内から呼び出す場合は CALL 疑似命令を用います. C 言語のコードからであれば関数として呼び出すことができます.

呼び出し規約はアーキテクチャにより様々ですが, 基本的には呼び出し側は SP に示されるスタックに引数を積んでサブルーチンを呼び出します. サブルーチン側は FP として示されるスタックに

それが渡されているものとして処理を行い, データレジスタまたは汎用レジスタのうち第 1 のものにアドレスなり即値なりを返値として転送して戻る, という良くある流れを取ります. なおサブルーチン側はレジスタは呼び出し側が保存しているものとして, 一部例外を除き好きに使って良いとされています ("caller saves").

2.6 その他

Plan 9 アセンブラでは **#define** や **#include** が使えます. たとえば amd64 のマシン依存部のコードでは リスト 2.14 の例にあるように先頭で各種 define によるマクロの定義されたヘッダファイルの取り込みや, 前述したアセンブラでサポートされていない命令の定義のために, BYTE 疑似命令によって構成したバイト列 (エンコードされた機械語命令) をマクロとして名前を付けて利用といったことを行っています.

リスト 2.14: #include と #define の利用 (src/9/pc64/l.s)

```
/* マクロとして DELAY を定義 */
#define DELAY           BYTE $0xEB; BYTE $0x00  /* JMP .+2 */

/* 以下の様にニーモニックであるかのように記述 */
TEXT _lme<>(SB), 1, $-4
        MOVL    SI, CR3                         /* load the mmu */
        DELAY

        ....
```

第 3 章

アーキテクチャ固有のお話

この章では, 前章で解説した一般的な Plan 9 アセンブラ共通のお作法から外れる箇所を i386, amd64 のそれぞれについて取り上げます.

3.1 i386

8a は x86 32bit (i386) アーキテクチャ向けのアセンブラです. 基本的にはこのアセンブラはプロテクトモードで動作するコードのアセンブル用であることが前提となっています. x86 固有のお話としてはレジスタそして命令セットにおける差分に分けられます.

3.1.1 利用可能なレジスタ

レジスタとしては SP, AX, BX, CX, DX, BP, DI, SI というシンボルでいわゆる x86 の汎用データレジスタにアクセスすることができます. 同様にセグメントレジスタ (CS, DS, ES, FS, GS, SS) や CR0, CR3 といった制御レジスタ, GDTR, IDTR に類される各種システムテーブルポインタレジスタへのアクセスもサポートされています.

コンパイラ, アセンブラそしてリンカで規定されるこれらのレジスタの利用方法は以下の通りです. スタックポインタは SP に保持されます. サブルーチン等の返値には AX を用います. 物理的なフレームポインタとして x86 には BP (ebp) が存在しますがそちらは使わず FP 利用します.

リスト 3.1: AX を用いた返値の例: getcr0 (src/9/pc/l.s)

```
TEXT getcr0(SB), $0         /* CR0 - processor control */
        MOVL    CR0, AX
        RET
```

x86 的には正式な 32bit 幅レジスタの名称は **E** をプレフィクスとした名前 (e.g. EAX, ESP) ですが, 8a では定義されておらず使えません. ビット幅については第 2 章で述べたロード・ストアを担当する MOV 疑似命令のサフィクスをもって判断されます. リスト 3.2 に eax (8a では AX) から ebx (8a では BX) に値を格納する際の例を記載します. 特筆すべきは例 3 と例 4 が同じオペレーションであることです. MOV のサフィクスによる幅指定は下位からの幅であることに注意する必要

があります．AH などの上位にアクセスしたい場合は明示的にこのレジスタを指定する必要があります．

リスト 3.2: MOV のサフィクス切り替えによる

```
# 例 1. 32bit 幅の MOV : eax から ebx (32bit 全体)
MOVL    AX, BX

# 例 2. 16bit 幅の MOV : ax から bx (下 16bit)
MOVW    AX, BX

# 例 3. 8bit 幅の MOV : al から bl (下 8bit)
MOVB    AX, BX

# 例 4. 8bit 幅の MOV : al から bl (下 8bit)
MOVB    AL, BL

# 例 5. 8bit 幅の MOV : ah から bh (下位 16bit 幅中の上 8bit)
MOVB    AH, BH
```

これらの MOV とレジスタの柔軟な関係は基本的には汎用レジスタに限ったものでありその他の 32bit 幅が前提であるレジスタでは不正なものと見なされます．たとえば MOVB BP, DI といった格納は基本的には不正なものです．

3.1.2 命令セット

命令セットは一部を除き, Intel Software Developer's Manual に記載されたものを用いることができます．前章でも述べたとおりニーモニックは全て大文字である必要があります．"一部" の例外はロード、ストア命令および分岐命令です．先のレジスタの節でも MOV 疑似命令について触れましたが, Plan 9 のアセンブラではレジスタ, メモリを問わず値の移動には MOV 疑似命令を用います．とはいえ gas や NASM でも i386 では基本的には同様のニーモニックである mov 命令でまかなえているのでこのあたりは大きな差異ではないでしょう．

例外として分岐命令があります．x86 では JO , JNO といった分岐命令を定義していますがこれらを使うことはできません．8a では 2a つまり MC68020 向けの分岐命令のみがサポートされています．このため JO, JNO, JB, JNB, JZ, JNZ, JBE, JNBE, JS, JNS, JP, JNP, JL, JNL, JLE, JNLE といった x86 向けの分岐命令は JOS, JOC, JCS, JCC, JEQ, JNE, JLS, JHI, JMI, JPL, JPS, JPC, JLT, JGE, JLE, JGT と書く必要があります．

その他ニーモニック自体ではなく使い方に影響のあるものを以下に挙げます．JMP や CALL は x86 のそれと同じく用いることができますが, **JMP*** のように * (アスタリスク) が後置されている場合があります．これらは絶対番地へのジャンプに用いられます．リスト 3.3 は, カーネルを起動するにあたり絶対番地に飛ぶためにこれを用いている例です．このような低レイヤの処理でのみ用いられる傾向にあります．

リスト 3.3: アスタリスク付きの JMP の例 (src/9/pc/l.s)

```
TEXT _startKADDR(SB), $0
        MOVL    $_startPADDR(SB), AX
        ANDL    $~KZERO, AX
        JMP*    AX
```

x86 には指定回数分だけ第 1 オペランドの命令を繰り返し実行する rep 命令が存在します. gas ではリスト 3.4 上部のように利用されます. 8a でも REP, REPN はニーモニックとして定義されていますが, プレフィクス的に用いることはできません. 例の下部に上げたようにあたかも個別の独立した命令であるように書く必要があります. 多くの例では ; (セミコロン) で区切って後置しているかのように記述をすることが多いようです (リスト 3.5 に挙げる insb サブルーチンなど).

リスト 3.4: rep 命令の利用例: 6 回 movsb を繰り返す

```
/* gas の場合 */
  mov $6, %ecx
  rep movsb

/* 8a の場合 */
  MOV $6, CX
  REP; MOVSB
```

リスト 3.5: rep 命令の利用例: insb サブルーチン (src/9/pc/l.s)

```
TEXT insb(SB), $0
        MOVL    port+0(FP), DX
        MOVL    address+4(FP), DI
        MOVL    count+8(FP), CX
        CLD
        REP;    INSB            /* ここで INSB を CX 回繰り返す */
        RET
```

3.2 amd64

amd6 用のアセンブラ 6a も基本的なところは 8a と同じです. ここでは i386 との差異がある部分のみ記載します.

3.2.1 レジスタの種類における差異

レジスタの種類としては基本的には 8a, i386 のそれと同様のレジスタが 64bit 幅に拡充されたものとして利用できます. amd64 では 64bit 幅の汎用レジスタは rax, rbx という名前で定義されていますが, 6a ではこれらは存在せず 32bit 幅の場合と区別すること無く AX, BX といったシンボル

を用います.

6a では MMX および MMX および XMM の命令セットが追加されておりそれぞれのレジスタが M0〜M7 および X0〜X15 として割当たっています. また固定小数演算用のレジスタとして R8〜R15 が割り当てられています. この R ではじまるレジスタ名は MC68020 のアセンブラで用いられているデータレジスタの表記方法ですが, amd64 のそれではこのように外部レジスタの割当先として R15 以下の名前を利用しています.

3.2.2 使い方における差異

前述の通り 64bit 幅レジスタ用に特別な名前は存在しないため, MOV 疑似命令のサフィクスを以て操作するビット幅を決定します. その他のサフィクスを用いる命令を含め 64bit 幅用には **Q** (Quad) を付与することでこれを指定します. また 128 bit 幅での操作では **O** (Octo) を付与することとしています. Intel のマニュアルでは 128bit 幅向けには O, DQ など様々なサフィクスが存在しますが 6a ではこれに統一されています. XMM の命令でも同じく L, Q, O などを用いますが一部の場合に PL (Packed Long) を Q, DQ, PI の代わりとして用いることがあります. いずれの場合も, 浮動小数点用のレジスタへのロード/ストア命令にも MOV を利用することができ具体的な実装はアセンブラにて抽象化されます.

amd64 では AX 等のレジスタが 64bit 幅になるため MOVL, MOVW などの狭い MOV 疑似命令を用いた場合に実際に格納される値について注意が必要です. これらのオペレーションではでは指定されたビット幅分のみ下位からコピーされます. それ以外の上位ビットはゼロクリアされます.

型にも気を付ける必要があります. 即値としてのオペランドは符号付き 32bit 整数に限定されており 64bit 幅の操作を行う時のみ符号付き 64bit 整数として扱われます. 例外として MOVQ のオペランドとなる場合のみ即値として 64bit 整数が記述できます.

サブルーチン呼び出しの場合における値の受け渡しについても注意が必要です. i386 (8a) と異なる点として, 第 1 引数が整数値またはポインタの場合はレジスタでの値渡しとなりこれに BP レジスタが使われます. 簡便のため RARG なる名前が用意されておりこれで第 1 引数を参照することができます (リスト 3.6). 逆にサブルーチンを呼ぶ場合には, RARG に値を格納する必要があります.

リスト 3.6: RARG の利用例: insb サブルーチン (src/9/pc/l.s)

```
TEXT inb(SB), 1, $-4
        MOVL    RARG, DX   /* MOVL port+0(FP), DX */
        XORL    AX, AX
        INB
        RET
```

サブルーチンの返値に AX を用いるのは 8a と共通です. リスト 3.6 の例でも INB の結果 が AX に値が格納されるためそのまま RET 命令にて呼び出し元に戻っています. この例とは異なり浮動小数点の返値は X0 にて渡すこととされています. ただしこの例は少なく, リスト 3.7 に挙げる sqrt() のみでしか用いられていません.

リスト 3.7: X0 による返値の例: sqrt() (src/libc/amd64/sqrt.s)

```
sqrt(SB), $0
        MOVSD   a+0(FP), X0
        SQRTSD  X0, X0
        RET
```

なおこの例では SQRTSD つまり XMM での浮動小数点演算命令が明示的に使われています. 歴史的に x86 には x87 なる浮動小数点演算を行うコプロセッサが付属しており, XMM ではなくそちらを明示的に使うことも可能です. たとえば sqrt 相当の演算には XMM 側の SQRTSD に加えて x87 側の AFSQRT なる命令も定義されています. ただしこのようにアセンブラ側で明示的に指定している場合を除いて, C コンパイラ (8c, 6c) やリンカ (8l, 6l) では XMM 命令セットを用いるようになっています.

おわりに

本書では Plan 9 のアセンブラについて解説しました.

もともとの個人的なモチベーションとしては Plan 9 のブートローダや UEFI ローダのコードを読んで, 名著である「Linux のブートプロセスをみる」や「FreeBSD のブートプロセスをみる」をパチって「Plan 9 のブートプロセスをみる」を作ってみたい, あるいは "Linux Inside" (https://www.gitbook.com/book/0xax/linux-insides/details) に倣って"Plan 9 Inside" を作ってみたいという目標がありました. これにあたって Plan 9 の\アセンブラを読まなければならず, お作法についての本が欲しくなったため本書の執筆に至りました.

Plan 9 アセンブラは最近では golang の裏側で使われています. とはいえ元々の OS やその fork たちについては最近は Install Battle を試みる声もあまり聞こえず比較的忘れ去られた感があります. そんな需要のなさそうな OS のさらに局所的なアセンブラ部分のみをターゲットにした薄い本なぞさらに需要がなさそうですが, 本書をお手にとって頂いて何がしか「ふーん」と興味深い点が見つけて頂けたのなら幸いです.

Plan 9 Assembler Handbook
8a & 6a

2016 年 08 月 14 日 第一版発行
著　者　@n_kane
印刷所　Glenda9