4 命令セット (2)

- ♣ 世界で最も有名なアーキテクチャの一つ x86 の命令セット (の複雑さと不規則さ) を垣間みる
- ♣ アドレッシングモード を覚える
- ♣ RISC と CISC という分類を知る

4.1 例 2—x86 の命令セット

- x86 とは, Intel の 8086 に始まり, 80286, 80386, 80486, Pentium, Core, ··· と続くプロセッサの総称 [1]
- 本講義では 80386 以降の 32bit のアーキテクチャを扱う [2]
- 本講義ではアセンブリは gas (Cygwin 等の Unix 系 OS で利用できるアセンブラ) の記法を用いる [3]

4.1.1 x86 のレジスタ

- 主要な整数データは 8 bit (byte), 16 bit (word), 32 bit (long word) の 3 通り
- 符号付き整数は 2の補数 表現
- 外部 (プログラマ) から見えるレジスタ

`eax, ebx, ecx, edx` `esp, ebp, esi, edi`	汎用 レジスタ (32bit)
`eip`	命令ポインタ (32bit) MIPS の プログラムカウンタ に相当
`eflags`	条件コード (32bit) フラグ レジスタとも呼ばれ, 演算結果が 0 か負か オーバフローがあったか等を記録する.
`cs, ss, ds, es, fs, gs`	セグメントレジスタ (16bit) (※)

(※) x86 の主記憶は「セグメント」と呼ばれる単位 (サイズは 2^{16} バイトか 2^{32} バイト) で管理される. セグメントレジスタは, アクセスするセグメントの番号を指定するのに用いる. 本講義では無視してよい.

[1] 演習室の PC も x86 である. というより, 全ての Windows マシンも新しい Mac もプロセッサは x86.
[2] 8086~80286 は 16bit プロセッサだった. Pentium 4 の後期より 64bit 拡張もされているが, 本講義では省略する.
[3] C プログラム `foo.c` を `gcc -S foo.c` でコンパイルすると, アセンブリプログラムが `foo.s` に得られる.

4.1.2 加算命令 (32bit データを対象とするもののみ紹介する)

(1) アセンブリ記法と動作

- 命令名は addl (add long word)

  16bit データに対する加算は `addw` (add word)

  8bit データに対する加算は `addb` (add byte)

- オペランドは 2 つ

  `addl` S,D ⇒ D = $D+S$ を計算する

- オペランドの可能な組合せ

  レジスタや即値だけでなく 主記憶 も指定可能 (しかし, 下記の組合せのみ可能)

S	D	
レジスタ	レジスタ	レジスタ = レジスタ + レジスタ
即値	レジスタ	レジスタ = レジスタ + 即値
主記憶	レジスタ	レジスタ = レジスタ + 主記憶
レジスタ	主記憶	主記憶 = 主記憶 + レジスタ
即値	主記憶	主記憶 = 主記憶 + 即値

- レジスタオペランド

  レジスタ名の前に `%` をつける

  (例) `addl %ebx,%eax` の動作は eax= eax+ebx

- 即値オペランド

  即値の前に `$` をつける

  (例) `addl $85,%eax` の動作は eax= eax+85

- 主記憶オペランド (ややこしい)

  (1) (例) `addl 1048,%eax` の動作は eax = eax + Mem[1048 ,4]

  (2) (例) `addl (%edx),%eax` の動作は eax = eax + Mem[edx ,4]

  (3) (例) `addl 24(%ebp),%eax` の動作は eax = eax + Mem[24+ebp ,4]

  ebp=1024 のとき, アクセスする番地は 1048

  (4) (例) `addl 20(%ebp,%ecx,4),%eax` の動作は eax = eax + Mem[20+ebp+ecx*4 ,4]

  ebp=1024, ecx=5 のとき, アクセスする番地は 20+1024+5*4 = 1064

  (例) `addl (%ebp,%ecx,2),%eax` の動作は eax = eax + Mem[ebp+ecx*2 ,4]

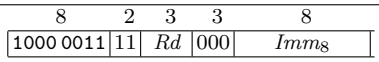
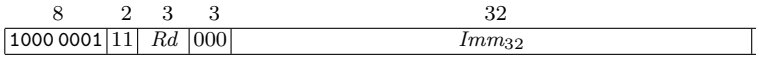
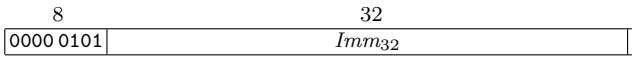
(2) 命令の符号化 (機械語)

☆ 命令フォーマットは, オペランドの組合せにより異なり, 命令長も異なる (2 ~ 11 バイト)

非常に複雑で **不規則**

(1) レジスタ = レジスタ + 即値

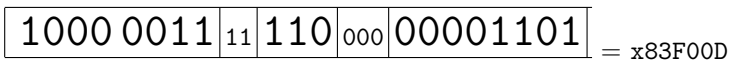
- 即値のビット数は先頭バイトで指定する (先頭=x83 ⇒ 即値 8 bit, 先頭=x81 ⇒ 即値 32 bit)
- 代入先が `eax` の場合のみ別扱い

1) `addl $`Imm_8`,%`Rd	
2) `addl $`Imm_{32}`,%`Rd	
3) `addl $`Imm_{32}`,%eax`	

レジスタ (*Rs*, *Rd*, *Rb*, *Rx*) の符号化

レジスタ	符号	レジスタ	符号
eax	000	esp	100
ecx	001	ebp	101
edx	010	esi	110
ebx	011	edi	111

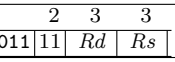
(例) `addl $13,%esi` の機械語は, 即値が 8bit 以内なので 1) のフォーマット



= x83F00D

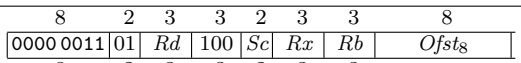
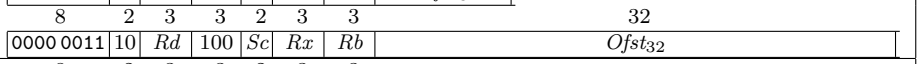
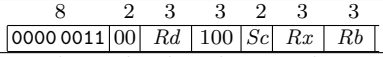
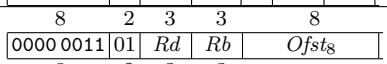
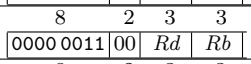
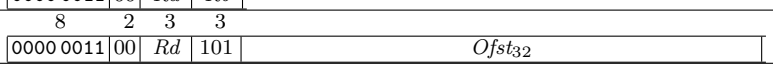
(2) レジスタ = レジスタ + レジスタ

- 最初の 1 バイトが x03 で, 次の 2bit が b 11

`addl %`Rs`,%`Rd	

(3) レジスタ = レジスタ + 主記憶

- 最初の 1 バイトは x03 ((2) と同じ)
- 次の 2 bit は *Ofst* の bit 数を表す (b00⇒ なし, b01⇒ 8 bit, b10⇒ 32 bit)
- その次の 3bit の次の 3bit が 100 なら *Rx*, *Sc* あり; 101 なら *Ofst* のみ (この場合前の 2bit は b00) [4]
- *Sc* の符号化は 1 ⇒`b00`, 2 ⇒`b01`, 4 ⇒`b10`

`addl` $Ofst_8$`(%`Rb`,%`Rx`,`Sc`),%`Rd	
`addl` $Ofst_{32}$`(%`Rb`,%`Rx`,`Sc`),%`Rd	
`addl (%`Rb`,%`Rx`,`Sc`),%`Rd	
`addl` $Ofst_8$`(%`Rb`),%`Rd	
`addl` $Ofst_{32}$`(%`Rb`),%`Rd	
`addl (%`Rb`),%`Rd	
`addl` $Ofst_{32}$`,%`Rd	

(4) 主記憶 = 主記憶 + レジスタ (全部で多分 6 パターン) は省略

(5) 主記憶 = 主記憶 + 即値 (全部で多分 12 パターン) は省略

[4] *Rb* に `esp` (`b100`) や `ebp` (`b101`) が指定できない場合があるのはこのため.

(3) 条件フラグの設定

演算結果に応じて, `eflags` レジスタの下記のビットが 0/1 に設定される

名称	ビット位置	意味
`CF`	carry flag	2^0	符号無し整数のオーバフローが起こっていれば 1, そうでなければ 0
`ZF`	zero flag	2^6	演算結果がゼロなら 1, そうでなければ 0
`SF`	sign flag	2^7	演算結果の符号ビットと同じ値
`OF`	overflow flag	2^{11}	符号付き整数のオーバフローが起こっていれば 1, そうでなければ 0

(例) `eax=7` のときに `addl $-10,%eax` を実行すると, `eax=7+(-10)=-3` となる

キャリーはない → `CF=0`
結果はゼロではない → `ZF=0`
結果が負 → `SF=` 1
オーバフローはない → `OF=0`

`eflags =`

	OF							SF	ZF							CF
	2^{11}							2^7	2^6							2^0
$\cdots$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

☆ このフラグを用いて, 条件付き分岐命令やループ命令を実行する

4.1.3 その他の命令

☆ 命令は非常に多いため, 以下ではよく使われるもののみ記載

- データ転送命令

`movl`	データ転送 (move) 命令. `movl S, D` で S から D に 32bit データを コピー する.
`pushl`	レジスタの値を主記憶中の「スタック領域」に退避 (保存) する.
`popl`	`pushl` の逆. スタック領域からレジスタに値を読み戻す.

- 2 進算術命令

`subl`	減算 (subtract) 命令.
`incl`	increment 命令. 値を 1 増やす.
`decl`	decrement 命令. 値を 1 減らす.
`negl`	negate 命令. 符号反転させる.
`imull`	符号付き 整数の乗算 (signed multiply) 命令.
`mull`	符号なし整数の乗算 (unsigned multiply) 命令.
`idivl`	符号付き整数の除算 (signed divide) 命令. 剰余 も同時に計算する. `idivl S` の動作は `edx = (edx||eax)/`S, `eax = (edx||eax)%`S
`divl`	符号なし整数の除算 (unsigned divide) 命令.
`leal`	実効アドレス (後述) をレジスタに求める (load effective address) 命令.
`cmpl`	比較 (compare) 命令. 2 数を大小等号比較し, 結果を `eflags` にセットする.

- 論理命令

`andl`	ビット毎の論理積
`orl`	ビット毎の論理和
`xorl`	ビット毎の排他的論理和
`notl`	ビット毎の論理否定

- シフト

`shll` `sall`	論理左シフト (shift logical left) 命令. 算術左シフト (shift arithmetic left) 命令. … 2 命令とも動作は全く 同じ
`shrl`	論理右シフト (shift logical right) 命令.
`sarl`	算術右シフト (shift arithmetic right) 命令.

- 制御転送命令

`jmp`	無条件ジャンプ (jump) 命令. 次の命令ではなく, 指定された番地の命令を実行する (`eip` の値を指定された値に設定する).
`j`*cc*	`eflags` 中のフラグの値が指定 条件 を満たせばジャンプし, そうでなければジャンプしない. `cmp` 等 `eflags` を設定する命令に続けて実行する.
`je/jz`	等しい, ゼロのとき (`ZF=1`) ジャンプ
`jne/jnz`	等しくない, ゼロでないとき (`ZF=0`) ジャンプ
`jg/jnle`	大きいとき (`((SF xor OF) or ZF)=0`) ジャンプ
`jge/jnl`	以上のとき (`(SF xor OF)=0`) ジャンプ
`jl/jnge`	小さいとき (`(SF xor OF)=1`) ジャンプ
`jle/jng`	以下のとき (`((SF xor OF) or ZF)=1`) ジャンプ
`jo`	オーバフローのとき (`OF=1`) ジャンプ
`js`	負のとき (`SF=1`) ジャンプ
`call`	サブルーチン (C 言語の 関数) 呼び出し (call procedure) 命令. 戻り番地 (`call` 命令の次の命令の番地) をスタックに保存し, 指定された番地にジャンプする.
`ret`	サブルーチン復帰 (return) 命令. スタックに保存されている戻り番地にジャンプすることにより, サブルーチンの呼び出し元に戻る.
`loop`	ループ命令. `ecx` レジスタの値を -1 し, その値がゼロでなければ指定番地 (ループの先頭) にジャンプし, ゼロになればジャンプしない.

- 基本命令としては, その他に下記の命令群がある

ビット命令とバイト命令	レジスタやメモリーのビット列操作やバイト列操作のための命令群
文字列演算	文字列 (メモリー上のバイト列) のコピーやサーチ等のための命令群
10 進算術命令	10 進表現 (※) のデータに対する四則演算のための命令群
フラグ制御命令	`eflags` レジスタ内のフラグを操作する命令群
セグメントレジスタ命令	セグメントレジスタを操作する命令群

(※) 整数を可変長で 10 進数と対応がとりやすいビット列で表現する方法. 例えば, 1945 を x3$\underline{1}$3$\underline{9}$3$\underline{4}$C$\underline{5}$ と表すゾーン表現や, x0$\underline{1945}$C のように表すパック表現がある.

- さらに, 下記の拡張命令群がある

FPU 命令群	浮動小数点演算を実行する命令群
MMX 命令群	マルチメディア処理や通信処理を高速化するための拡張命令群
SSE 命令群	グラフィックス, 音声・画像処理, 動画像の符号/復号化等の応用に対する性能強化のための拡張命令群
システム命令	プロセッサの機能を制御するための命令 (主に OS が使用)

4.2 アドレッシングモード

命令のオペランドの 指定法 (特に主記憶上にあるデータの 番地 の指定法)

(1) 即値アドレッシング (immediate addressing)

命令中 にオペランドそのものが書かれている

(例) MIPS の `addi $8,$9,123` ($8=$9+123)

(例) x86 の `addl $123,%eax` (eax=eax+123)

(2) 直接アドレッシング (direct addressing) [5]

命令に書かれている番地にそのままアクセスする

(例) x86 の `addl 1048,%eax` (eax=eax+M[1048,4])

(3) 間接アドレッシング (indirect addressing)

指定したレジスタ等に 格納 されている番地にアクセスする

(例) x86 の `addl (%ebx),%eax` (eax=eax+Mem[ebx,4]) レジスタ間接アドレッシング

(4) 相対アドレッシング (relative addressing)

指定されたレジスタ等に格納された 基準 番地に, 相対番地を足した番地にアクセスする

(例) MIPS の `lw $8,12($30)` ($8=Mem[$30+12,4])

(例) x86 の `addl -12(%ebp),%eax` (eax=eax+Mem[ebp-12,4])

(例) MIPS の `bne $8,$9,12` (pc=pc+4+12*4)

(5) インデックス修飾 (index modification)

(4) 等の番地に, さらにレジスタ等の値を加算した番地にアクセスする

(例) x86 の `addl -24(%ebp,%ecx,1),%eax` (eax=eax+Mem[ebp-24+ecx,4])

(例) x86 の `addl -24(%ebp,%ecx,4),%eax` (eax=eax+Mem[ebp-24+ecx*4,4]) [6]

【用語】

- ベースレジスタ (base register) [7] … 相対アドレッシングの基準番地を保持するレジスタ
- オフセット (offset) [8] … 基準番地からの変位 (相対番地)
- インデックスレジスタ (index register) [9] … インデックス修飾に用いるレジスタ
- 実効アドレス (effective address) … 実際にアクセスされる番地

(例) x86 の `addl %eax,-24(%ebp,%ecx,4)` において

ベースレジスタは ebp , オフセットは -24 インデックスレジスタは ecx

ebp=1024, ecx=8 のとき, 実効アドレスは ebp-24+ecx*4= 1032

[5] 絶対アドレッシング (absolute addressing) とも言う)

[6] スケール付きインデックス修飾 (scaled index modification) と言う

[7] 純和訳は「基底レジスタ」

[8] ディスプレースメント (displacement) ともいう

[9] 純和訳は「指標レジスタ」

相対アドレスを用いる理由

1. オペランドの ビット長 の制約
   主記憶の番地の範囲 (アドレス空間) が大きくなると, 番地を表すのに必要なビット数も大きくなり, 機械語の中に全てを収容するのが難しくなる. 相対番地の指定ならビット数は少なくてすむ.
2. マルチタスク の実現
   最近のコンピュータでは, 1 つの CPU で複数のプログラムを実行するが, そのためにプログラムは, 主記憶のどの番地にロードしても実行可能 (再配置可能 (relocatable) あるいは position independnt) である必要がある. 相対番地はこれを容易にする.
3. C 言語などのプログラミング言語からのコード生成
   C 言語では, 関数毎に「フレーム」と呼ばれる領域を作り, 関数内で宣言した変数はここに置く. フレームが主記憶のどこに置かれるかは実行時に決まり, その先頭番地がレジスタに格納される. 関数内の変数には, この基準番地からの相対番地でアクセスする.

4.3 命令セットのバリエーション

4.3.1 オペランド数

(a) 3 アドレス方式 … 命令のオペランドが 3 つ (例: MIPS)
(b) 2 アドレス方式 … 命令のオペランドが 2 つ (例: x86)
(c) 1 アドレス方式 … 命令のオペランドが 1 つ
(d) 0 アドレス方式 … 命令のオペランドを持たない

4.3.2 固定長命令と可変長命令

	固定長命令 (fixed length instruction)	可変長命令 (variable length instruction)
機械語長	全ての命令で 同じ	命令毎に 異なる
(例)	MIPS (4 バイト)	x86 (1 ~ 17 バイト)
機械語の解読	単純	複雑
コード密度 (*1)	低 い	高 い (*2)
オペランド数や長さの制約	強 い	弱 い

(*1) 同じ計算をする機械語プログラムがより短いとき, コード密度 (code density) が高いと言う.

(*2) 出現頻度の高い命令の長さを 短 くすることにより実現.

4.3.3 RISC と CISC

- CISC (Complex Instruction Set Computer)

  多種類 の 高機能 な命令を持つコンピュータ

  アドレッシングモード も豊富

  (例) x86 , IBM 360, VAX

  狙い

  - 命令を高機能化 → ハードウェア 実行で高速化 (神話)
  - アセンブリ/機械語プログラムの命令数 削減
    (プログラミングの負担軽減)

- RISC (Reduced Instruction Set Computer)

  最小限 の 単純 な命令セットを持つコンピュータ

  狙い

  - ハードウェアの 単純化 による実行の高速化

- CISC と RISC の比較

  例) x86 (1 命令)

  ```
  addl 124(%ebp,%ecx,4),%edx
  ```

  ⇓

  MIPS (4 命令)

  ```
  sll $10,$8,2
  add $10,$10,$30
  lw $10,124($10)
  add $8,$8,$10
  ```

  ここで, x86 の 1 命令が MIPS の 4 命令より速い

  とは限らない

Nagisa ISHIURA