計算機システム

自習 CA0301:プログラミング言語と命令セット

http://arch.naist.jp/Lectures/ARCH/ca0301/ca0301j.pdf

条件コード

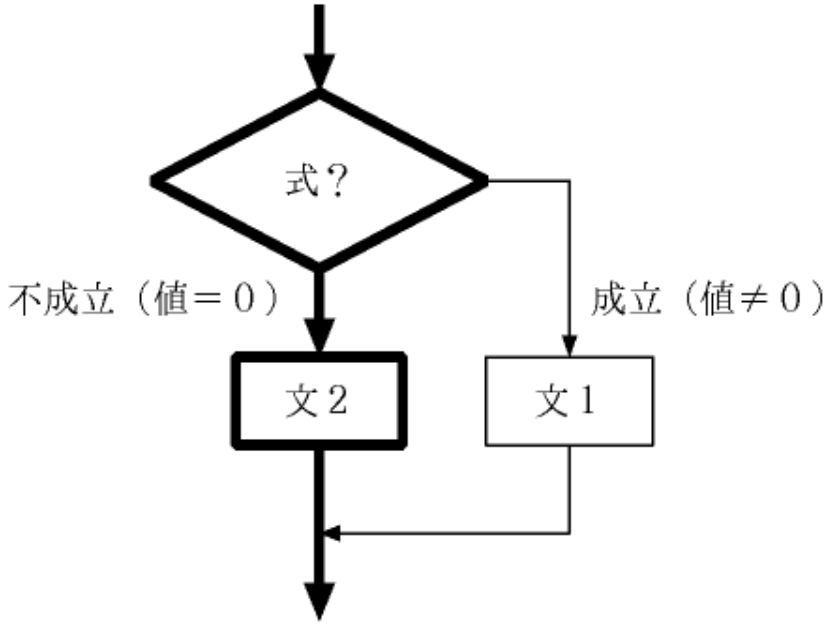
IF文とキャリーフラグの関係を正しく理解していますか？

条件コード

命令セット理解の前に，条件コードの理解が必要

条件別実行

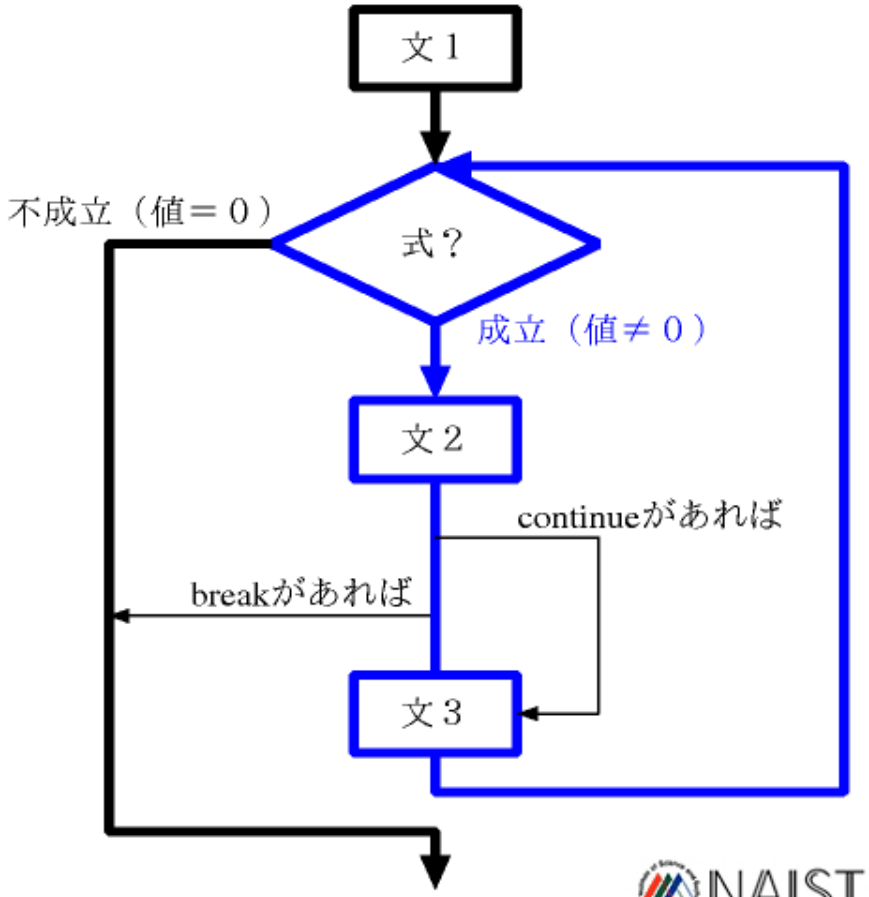
```
if (式) { 文1 }
else    { 文2 }
```



- 条件判定は，変数の型と条件コードに依存
- プログラミング言語における条件記述と，ハードウェアにおける条件コードの関係を正しく理解する

ループ

```
for (文1;式;文3) {
  文2
  continue; /*forの最後に分岐*/
  break;    /*forから脱出*/
}
```



NAIST

条件コード

ハードウェアが生成する固定小数点演算の条件コード

- Negative … 最上位ビットが0なら0，その他は1
- Zero … 全ビットが0なら1，その他は0
- Carry … 最上位ビットからの桁上げ無なら0，その他は1
  - 符号無のオーバフローに等しい
- oVerflow … オーバフロー無なら0，その他は1
  - 符号付のオーバフロー（最上位桁からと-1桁からのCarryが異なる場合）

C言語にて記述可能な固定小数点数の判定との対応

== !=	… 全ビット一致（各ビットXORの全てが0）か不一致
符号無A<B	… A-Bの結果，C==0
符号無 <=	… A-Bの結果，C==0 or Z==1
符号無 >=	… A-Bの結果，C==1
符号無 >	… A-Bの結果，C==1 and Z==0
符号付A<B	… A-Bの結果，N!=V (Bが0の時はN)
符号付 <=	… A-Bの結果，Z==1 or N!=V
符号付 >=	… A-Bの結果，N==V (Bが0の時はNの反転)
符号付 >	… A-Bの結果，Z==0 and N==V

- 通常，上記に対応するインタフェースが条件分岐命令に組み込まれる

条件コード

前章の説明通り，Ｃ言語の枠組ではオーバフローを判定できない

- 条件分岐命令（bicc）にて，iccにVを指定すれば可能

ハードウェアが生成する浮動小数点演算の条件コード

- 条件コード更新の競合を防ぐため，比較命令のみ更新するのが一般的
- = … 比較の結果，等しい
- < … 比較の結果，<
- > … 比較の結果，>
- Unordered … 対象が非数値を含むため比較困難

Ｃ言語にて記述可能な，浮動小数点数の判定

- == !=
- < <= >= >

符号ビットが独立しているので固定小数点数よりは単純
ただし，Unorderedの検出は困難
正規化数，∞，非数値の判定は，ライブラリ isnormal()，isinf()，isnan() により判定可能

命令セット

二つの顔（ハードウェア制御手段とプログラム表現手段）

命令セット

関数機能（引数⇒計算⇒返り値）を表現するには,どのような命令が必要か

- 代表的CISCのIBM360は,高級言語のために10進演算や編集命令を装備
  最近ではライブラリ（ソフトウェア）による処理が一般的

- 現在一般的な形式は，固定小数点数/論理/文字/文字列/浮動小数点数
  可変長の文字列もライブラリ（ソフトウェア）による処理が一般的

- 代わって，同一形式の複数データに対して一度に演算可能な命令（一般にマルチメディア命令と呼ぶ）の装備が一般化している
  小規模なベクトル命令，ロードストアのスループット保証はない
  SPARCのVIS，x86のMMX/SSE/SSE2/SSE3，ARMのVFP

- 演算対象となる２つのソースデータがレジスタにあれば高速
- メモリ上にある場合，メモリ参照結果を演算器に入力するので低速
- 元々の定義では，
  直交性を重視,レジスタ-メモリ間演算も可能としたのがCISC
  単純高速を重視，レジスタ間演算に絞ったのがRISC

命令セット

条件コードをふまえて必要な命令を整理 … 算術演算

固定小数点add,sub,mul,div

- ソース/デスティネーション共に単一ビット長なら，この4個で十分
- レジスタを64bitに拡張し，32bit/64bit演算を混在させる場合

  32bit長命令語では命令追加が困難なので，なるべく演算を共用する

  条件コードを32bit用と64bit用に分け一度に生成すれば命令数増加を抑制できる

  条件コードを1組に抑えると，32bit演算とほぼ同数の64bit演算が必要

32/64bit ± 32/64bit ⇒ 64bit

64bit演算のみ用意，32⇒64bit型変換は32bit右シフトによる符号拡張で対応

32bit * 32bit ⇒ 32bit

NegativeとZeroフラグだけであれば，符号無/符号付の結果は同じ

32bit * 32bit ⇒ 64bit

符号無演算と符号付演算の両方が必要．除算も同様だが，十分なハードウェアを投入しなければ低速なため，除算はライブラリによる処理が妥当という判断もある

命令セット

オプション（制御ビット）

- 条件コード更新の有無（条件コードを破壊しない演算のため）
- 条件コードを含めた演算（多倍長演算のため）

単精度/倍精度浮動小数点演算fadd,fsub,fmul,fdiv,比較,型変換

- 使用頻度が低い拡張倍精度はライブラリによる処理が一般的
- 汎用と浮動小数点レジスタを分ける場合，レジスタ間転送命令が必要
- 2命令による実行より高速であれば，積和（m&a,m&s）演算命令が有効
- 除算程度のサイクルで実行可能なfsqrtがあると効果的
- 1サイクルで実行可能な以下があると効果的

  `fmov` … レジスタ間転送
  `fabs` … 絶対値演算（符号ビットを0にするだけ）
  `fneg` … 符号反転（符号ビットを反転するだけ）

ARMのように，使用頻度によっては，浮動小数点演算は全てライブラリにより処理するという判断もある

命令セット

論理演算 and,andn,or,orn,xor,xnor

- nは第2オペランドをビット反転して演算するオプション

シフト sll,srl,sra

- 32bit/64bitを混在させる場合

  左シフトでは,シフト量を厳密に制限する場合のみ32/64用命令が必要

  右シフトでは，論理/算術共に，32bit/64bit用命令が必要

  `srl-32` … 64bitレジスタの下位32bitのみを右論理シフト，結果の上位32bitは0

  `srl-64` … 64bit全体を右論理シフト

  `sra-32` … 64bitレジスタの下位32bitのみ右算術シフト，結果の上位32bitは符号拡張

  `sra-64` … 64bit全体を右算術シフト

ロード/ストア ldsb,ldsh,ldub,lduh,ld,ldf,ldd,stb,sth,st,stf,std

- データ長＜レジスタ長のロードは,符号無/付に合わせた符号拡張が必要
- 倍精度浮動小数点数には，32bit浮動小数点レジスタの偶奇ペアを使用
- 汎用レジスタを64bit化するなら，浮動小数点レジスタも64bit化が自然

  前章で説明したエンディアンの影響を受けることに注意

  倍精度浮動小数点数の上位/下位32bitの順序がエンディアンに従わない場合もある

命令セット

条件分岐 bicc,bxcc,bfcc

- 各々，前述した32bit/64bit/浮動小数点用分岐条件に対応
  - NZCVなどの条件コードをそのまま使うのではないことに注意

定数セット sethi（SPARC特有）

- 大域変数はコンパイル時にラベル付けされ，リンカによるアドレス解決を経て，32bitの定数アドレスが割り当てられる
  - 32bit定数アドレスを32bit命令語に埋め込むことは困難なので，上位/下位に分けて設定するなどの工夫が必要（sethi命令+ld/st命令）
  - 最寄りのアドレスをベースレジスタにセットしておき，12bit程度の相対距離を加算して参照する方法もある（IBM360やARMの方法）

関数呼び出し/復帰 call,jmpl,save,restore（SPARC特有）

- call … 戻り先アドレスを特定のレジスタに格納して分岐
  - Delay-slotがない場合，戻り先は現PC+4
  - Delay-slotがある場合，戻り先は現PC+8
- jmpl … レジスタの内容を分岐先アドレスとして分岐
  - 前章にて説明した関数からの復帰に使用
    - 繰り返すが，セキュリティホールの主要な入口でもある
  - Call同様の戻り先アドレス格納機能があれば，間接関数callを直接実行

```
int (*func)(); (func)(引数);
```

今日はここまで