

説明用の簡易モデルに於ける計算の実行

予め

- 主記憶にプログラム・データを
 - PC に実行開始番地 (通常 0 番地) を
- それぞれ書き込んでおいて、

パルスが発生させると動作する。

プログラム・データ: bit 列 (機械語)

説明用の簡易モデルの機械語:

命令の符号化 (encoding)

命令	番号	符号化
stop	1	001
load	2	010
store	3	011
add	4	100
subtract	5	101
jump	6	110
jump flag	7	111

例: $87 + 26$ を計算する

	番地	機械語		アセンブリ言語	
		二進	十六進		
0	00000	01000100	0x44	load	A
1	00001	10000101	0x85	add	B
2	00010	01100110	0x66	store	C
3	00011	00100000	0x20	stop	0
4	00100	01010111	0x57	A 87	
5	00101	00011010	0x1a	B 26	
6	00110	00000000	0x00	C 0	

アセンブリ言語 (assembly language):
機械語と一対一対応する符丁 (ニーモニック) で
プログラムを記述したもの

アセンブラ (assembler):
アセンブリ言語で記述されたプログラムを
機械語に直すプログラム

→ CPU のアーキテクチャによって
それぞれ異なる (互換性がない)

例: A と B とを入れ替える

0		load	A
1		store	C
2		load	B
3		store	A
4		load	C
5		store	B
6		stop	0
7		A	15
8		B	5
9		C	0

例: $C \leftarrow \max\{A, B\}$

0		load	A
1		subtract	B
2		jump flag	M
3		load	A
4		jump	N
5	M	load	B
6	N	store	C
7		stop	0
8	A	5	
9	B	7	
10	C	0	

例: 15×3 を計算する

0	LOOP	load	B
1		subtract	D
2		jump flag	END
3		store	B
4		load	C
5		add	A
6		store	C
7		jump	LOOP
8	END	stop	0
9	A	15	
10	B	3	
11	C	0	
12	D	1	

演習問題:

簡易モデル上でのアセンブリ言語により、
自然数 n に対して、
1 から n までの和を計算する
プログラムを作成せよ。

- 実行前 (入力): ラベル A の場所に
数値 n が書き込まれている
- 実行後 (出力): ラベル C の場所に
計算結果の数値が書き込まれている

→ 出来たら提出

演習問題:

簡易モデル上でのアセンブリ言語により、
自然数 n に対して、
1 から n までの和を計算する
プログラムを作成せよ。

- 実行前 (入力): ラベル A の場所に
数値 n が書き込まれている
- 実行後 (出力): ラベル C の場所に
計算結果の数値が書き込まれている

→ 出来たら提出

プログラム (アルゴリズム) の評価

- 計算時間が速い
(= 計算ステップ数が少ない)
… 時間計算量
- 使用メモリ量が少ない … 空間計算量

最低限どれだけ必要か

→ “問題の難しさ” の評価

… 計算の理論・計算量の理論

プログラム (アルゴリズム) の評価

- 計算時間が速い
(= 計算ステップ数が少ない)
… 時間計算量
- 使用メモリ量が少ない … 空間計算量

最低限どれだけ必要か

→ “問題の難しさ” の評価

… 計算の理論・計算量の理論

プログラム (アルゴリズム) の評価

- 計算時間が速い
(= 計算ステップ数が少ない)
… 時間計算量
- 使用メモリ量が少ない … 空間計算量

最低限どれだけ必要か

→ “問題の難しさ” の評価

… 計算の理論・計算量の理論

演習問題:

簡易モデル上でのアセンブリ言語により、自然数 n に対して、1 から n までの和を計算するプログラムを作成せよ。

- 実行前 (入力): ラベル A の場所に数値 n が書き込まれている
- 実行後 (出力): ラベル C の場所に計算結果の数値が書き込まれている
- メモリ量: 12 **words**
- ステップ数: $7n + 5$ **steps**

まで絞れるようだ。 → 挑戦者求む

但し。初期のプログラム開発では、確かに

- 少しでも使用メモリが少ない
- 少しでも早い(ステップ数が少ない)

プログラムが良いとされたが、

現在は、実際には、

- 移植性(可搬性)・拡張性・部品性が高い
- 保守・管理コストが低い

プログラムが求められている。

- ハードウェアコストの低下
- 人的コストの(相対的)上昇

例: $S \leftarrow \sum_{i=0}^{N-1} A_i$ を計算する

0	LOOP	load	N	12	N	5
1		subtract	D	13	D	1
2		jump flag	END	14	SUM	0
3		store	N	15	A	6
4		load	SUM	16		37
5	ADD	add	A	17		-23
6		store	SUM	18		25
7		load	ADD	19		-3
8		add	D			
9		store	ADD			
10		jump	LOOP			
11	END	stop	0			

例: $S \leftarrow \sum_{i=0}^{N-1} A_i$ を計算する

0	LOOP	load	N	12	N	5
1		subtract	D	13	D	1
2		jump flag	END	14	SUM	0
3		store	N	15	A	6
4		load	SUM	16		37
5	ADD	add	A	17		-23
6		store	SUM	18		25
7		load	ADD	19		-3
8		add	D			
9		store	ADD			
10		jump	LOOP			
11	END	stop	0			

例: $S \leftarrow \sum_{i=0}^{N-1} A_i$ を計算する

0	LOOP	load	N	12	N	5
1		subtract	D	13	D	1
2		jump flag	END	14	SUM	0
3		store	N	15	A	6
4		load	SUM	16		37
5	ADD	add	A	17		-23
6		store	SUM	18		25
7		load	ADD	19		-3
8		add	D			
9		store	ADD			
10		jump	LOOP			
11	END	stop	0			

例: $S \leftarrow \sum_{i=0}^{N-1} A_i$ を計算する

0	LOOP	load	N	12	N	5
1		subtract	D	13	D	1
2		jump flag	END	14	SUM	0
3		store	N	15	A	6
4		load	SUM	16		37
5	ADD	add	A	17		-23
6		store	SUM	18		25
7		load	ADD	19		-3
8		add	D			
9		store	ADD			
10		jump	LOOP			
11	END	stop	0			

0	load	12	01001100	12	5
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	0
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	15	10001111	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00000=**0** **Acc:** -----=**FR:** -

0	load	12	01001100	12	4
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	0
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	15	10001111	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00100= 4 **Acc:** 00000100= 4 **FR:** 0

0	load	12	01001100	12	4
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	6
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	15	10001111	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00111= 7 **Acc:** 00000110= 6 **FR:** 0

0	load	12	01001100	12	4
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	6
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	16	10010000	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 01010 = 10 **Acc:** 10010000 = -112 **FR:** 1

0	load	12	01001100	12	4
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	6
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	16	10010000	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00000 = 0 **Acc:** 10010000 = -112 **FR:** 1

0	load	12	01001100	12	3
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	6
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	16	10010000	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00100= 4 **Acc:** 00000011= 3 **FR:** 0

0	load	12	01001100	12	3
1	subtract	13	10101101	13	1
2	jump flag	11	11101011	14	43
3	store	12	01101100	15	6
4	load	14	01001110	16	37
5	add	16	10010000	17	-23
6	store	14	01101110	18	25
7	load	5	01000101	19	-3
8	add	13	10001101		
9	store	5	01100101		
10	jump	0	11000000		
11	stop	0	00100000		

PC: 00111 = 7 **Acc:** 00101011 = 43 **FR:** 0

こうして、

loop の 2 巡目に

次の場所のデータを読むことに成功した。

以下略。

しかし、この方法では、
プログラム部分を書き換えてしまうため、
プログラミングし難い。
(面倒で、間違い易く、読み難い。)

→ 実機では、
指標レジスタ (index register) を使って
計算が行なえるように実装するのが普通。

しかし、この方法では、
プログラム部分を書き換えてしまうため、
プログラミングし難い。
(面倒で、間違い易く、読み難い。)

→ 実機では、
指標レジスタ (index register) を使って
計算が行なえるように実装するのが普通。

指標レジスタ (index register)

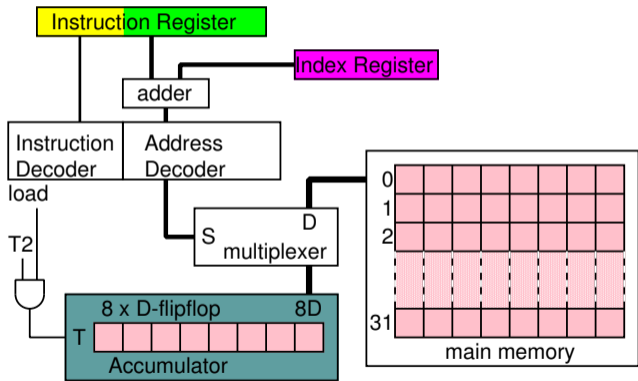
命令の番地部の解読の際に、

ここの値を加算して、

実際に読む番地 (実効アドレス) とする

→ 「配列」の実装

指標レジスタ (index register)



ともかく、これが出来ると、例えば、

N 個のデータを別の場所にコピーする
プログラム

が書ける。

(レポート課題の例)

更に

- データを次々と書き込んで
メモリを一杯にする
- 自分自身の複製を別の場所に作る
- 更にその複製に実行を移して
自分自身の複製を繰り返す

などのプログラムも作れる。

(但し、この簡易モデルでは、
機能限定なので書き難いかも)

更に

- データを次々と書き込んで
メモリを一杯にする
- 自分自身の複製を別の場所に作る
- 更にその複製に実行を移して
自分自身の複製を繰り返す

などのプログラムも作れる。

(但し、この簡易モデルでは、
機能限定なので書き難いかも)

とは言え、

こんな簡易モデルでも結構色々出来るので、

遊んでみて下さい。

計算の理論

命令の実行 (= 「計算」) とは、

レジスタまたは主記憶の
現在の値 (状態) に従って、

その値を変更 (書込) すること

であった。

プログラム内蔵方式 (von Neumann 型) では、プログラム・データを区別なくメモリ上に置いていたが、やはり本質的に違うもの。

- プログラム: 一つの問題では固定
- データ: 可変な入力



どんな (有効な) データ (入力) が来ても、
所定の出力を返すことが要請される。

プログラム内蔵方式 (von Neumann 型) では、プログラム・データを区別なくメモリ上に置いていたが、やはり本質的に違うもの。

- プログラム: 一つの問題では固定
- データ: 可変な入力



どんな (有効な) データ (入力) が来ても、
所定の出力を返すことが要請される。

或る問題の「計算が可能」



その計算を行なうプログラムが存在



計算機の機能 (= 「計算」のモデル)
を決めて議論する

ここでは、代表的な「計算のモデル」を
幾つか紹介する。

或る問題の「計算が可能」



その計算を行なうプログラムが存在



計算機の機能 (= 「計算」のモデル)
を決めて議論する

ここでは、代表的な「計算のモデル」を
幾つか紹介する。

或る問題の「計算が可能」



その計算を行なうプログラムが存在



計算機の機能 (= 「計算」のモデル)
を決めて議論する

ここでは、代表的な「計算のモデル」を
幾つか紹介する。