Linguagem Assembler para o Z0

A linguagem do Assembly usada pelo Z0 é baseada na sintaxe AT&T (usada na implementação no Assembler GNU as). Esta sintaxe possui diversas diferenças da sintaxe Intel.

Formato das Instruções:
As instruções são construídas por mnemônicos e seus valores, com marcadores, constante e possíveis variáveis. O formato das instruções na sintaxe AT&T segue a forma de:

    mnemônico origem, destino

Registradores
Todos os registradores devem ter como prefixo o sinal de porcentagem '%', por exemplo: %A ou %D.

Valores Literais
Todos os valores literais devem ter como prefixo o sinal de cifrão '$', por exemplo: $55, $376

Endereçamento de Memória
Na sintaxe AT&T a memória é referenciada com parêntese em volta do registrador que armazena o endereço: por exemplo (%A)

Tamanho dos operadores
Algumas instruções podem trabalhar com diferentes tamanhos de dados, assim as instruções podem ter um sufixo, informado o tamanho do dado que vão manipular, sendo b (8 bits), w (16 bits) e l (32 bits), por exemplo: movw $2000, (%A)

Instruções de Transferência de Controle
As instruções de jump, fazem o fluxo do programa mudar de uma posição do programa para outra. Para marcar as posições no programa são usados marcadores (labels) que sempre terminam com dois pontos (:). Por exemplo: loop:

Registradores virtuais:
Os símbolos R0, ..., R15 são automaticamente predefinidos para se referir aos endereços de RAM 0, ..., 15

Ponteiros de I/O :
Os símbolos SCREEN e KBD são automaticamente predefinidos para se referir aos endereços de RAM 16384 e 24576, respectivamente

Ponteiros de controle da VM:
Os símbolos SP, LCL, ARG, THIS, e THAT são automaticamente predefinidos para se referir aos endereços de RAM 0-4, respectivamente

Notações:
im : valor imediato (im* somente os valores 1, 0 e -1)
reg: registrador
mem: memória, ou seja (%A)

Limitações:

- A arquitetura não permite somar o valor da memória apontada por (%A) com o valor de %A, ou de (%A) com (%A), nem %A com %A.
- Não é possível somar (ou subtrair, se é que isso faz sentido) o registrador com o mesmo, por exemplo somar %D com %D.
- Não é possível ler e gravar da memória ao mesmo tempo, por exemplo incw (%A), ou subw (%A),%D,(%A) não funcionando no computador.

Observação: A linguagem Assembly apresentada é especifica para o processador produzido no curso, embora muito similar a outras usadas em produto de mercado, as instruções possuem limitações inerentes a cada hardware.

LEA - Carregamento Efetivo do Endereço (Valor)

```
lea{w}      im[16], reg[16]
```

Descrição: A instrução *lea* armazena o valor passado no registrador especificado (somente o %A na implementação desenvolvida).

Exemplo: Assembly: `leaw $15,%A` => *A = 15*
Linguagem de Máquina: **0**00 0000000 001 111

MOV – Copia Valores

```
mov{w}      reg/mem, reg/mem {, reg/mem, reg/mem}
mov{w}      im*, rem/mem {, reg/mem, reg/mem}
```

Descrição: A instrução *mov,* copia o valor da primeira posição para a segunda posição (terceira e quarta opcional).

Exemplo: Assembly: `movw (%A),%D` => *D = RAM[A]*
Linguagem de Máquina: **1**11 **1110000 010** 000

ADD - Adição de Inteiros

```
add{w}      reg/mem, reg/mem, reg/mem {, reg/mem, reg/mem}
add{w}      reg/mem, im*, reg/mem
```

Descrição: A instrução *add* soma dois valores inteiros e armazena o resultado no terceiro parâmetro (quarto e quinto opcional).

Exemplo: Assembly: `addw (%A), %D, %D` => *D = RAM[A]+D*
Linguagem de Máquina: **1**11 **1000010 010** 000

SUB - Subtração de Inteiros

```
sub{w}          reg/mem, rem/mem, reg/mem {, reg/mem, reg/mem}
sub{w}          rem/mem, im*, reg/mem {, reg/mem, reg/mem}
```

Descrição: A instrução *sub,* subtrai o segundo valor do primeiro valor e armazena o resultado no terceiro parâmetro (quarto e quinto opcional).

Exemplo: Assembly: `subw %D, (%A), %A` => *A = D-RAM[A]*
Linguagem de Máquina: **1**11 **1010011 100** 000

RSUB - Subtração de Inteiros Reversa

```
rsub{w}         reg/mem, rem/mem, reg/mem {, reg/mem, reg/mem}
rsub{w}         im*, rem/mem, reg/mem {, reg/mem, reg/mem}
```

Descrição: A instrução r*sub,* subtrai o primeiro valor do segundo valor e armazena o resultado no terceiro parâmetro (quarto e quinto opcional).

Exemplo: Assembly: `rsubw %D, (%A), %A` => *A = RAM[A]-D*
Linguagem de Máquina: **1**11 **1000111 100** 000

INC - Incrementa Inteiro

`inc{w}` `reg/mem`

Descrição: A instrução *inc,* adiciona um ao valor do registrador ou memória.

Exemplo: Assembly: `incw %D` `=> D = D+1`

Linguagem de Máquina: **1**`11` **`0011111 010`** `000`

DEC - Decrementa Inteiro

`dec{w}` `reg/mem`

Descrição: A instrução *dec,* diminui um ao valor do registrador ou memória.

Exemplo: Assembly: `decw %A` `=> A = A-1`

Linguagem de Máquina: **1**`11` **`1110010 100`** `000`

NOT – Negação por Complemento de Um

`not{w}` `reg/mem`

Descrição: A instrução *not,* inverte o valor de cada bit da série, ou seja, se um bit tem valor 0 fica com 1 e vice-versa.

Exemplo: Assembly: `notw %D` `=> D = !D`

Linguagem de Máquina: **1**`11` **`0001101 010`** `000`

NEG – Negação por Complemento de Dois

`not{w}` `reg/mem`

Descrição: A instrução *neg,* faz o valor ficar negativo, ou seja, um valor de x fica -x.

Exemplo: Assembly: `negw %A` `=> A = -A`

Linguagem de Máquina: **1**`11` **`0110011 100`** `000`

AND – Operador E (and)

`and{w}` `reg/mem, rem/mem`

Descrição: A instrução *and* faz o operador lógico E (and).

Exemplo: Assembly: `andw %A, %D, %D => D = A&D`

Linguagem de Máquina: **1**`11` **`0000000 010`** `000`

OR – Operador OU (or)

`or{w}` `reg/mem, rem/mem`

Descrição: A instrução *or* faz o operador lógico Ou (or).

Exemplo: Assembly: `orw (%A), %D, %D => D = RAM[A]|D`

Linguagem de Máquina: **1**`11` **`1010101 010`** `000`

NOP – Não faz nada (Not Operation)

`nop`

Descrição: A instrução *nop* não faz nada, usado para pular um ciclo de execução.

Exemplo: Assembly: `nop`
Linguagem de Máquina: **1**`11` **0101010 000 000**

JMP – Jump

`jmp`

Descrição: A instrução *jmp* faz um salto de execução para o endereço armazenado em %A.

Exemplo: Assembly: `jmp`
Linguagem de Máquina: **1**`11` **0001100 000 111**

JE – Salta Execução se Igual a Zero

`je`

Descrição: A instrução *je* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for igual a zero.

Exemplo: Assembly: `je`
Linguagem de Máquina: **1**`11` **0001100 000 010**

JNE – Salta Execução se Igual a Zero

`jne`

Descrição: A instrução *jne* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for diferente de zero.

Exemplo: Assembly: `jne`
Linguagem de Máquina: **1**`11` **0001100 000 101**

JG – Salta Execução se Maior que Zero

`jg`

Descrição: A instrução *jg* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for maior que zero.

Exemplo: Assembly: `jg`
Linguagem de Máquina: **1**`11` **0001100 000 001**

JGE – Salta Execução se Maior Igual a Zero

`jge`

Descrição: A instrução *jge* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for maior ou igual a zero.

Exemplo: Assembly: `jge`
Linguagem de Máquina: **1**`11` **0001100 000 011**

JL – Salta Execução se Menor que Zero

`jl`

Descrição: A instrução *jl* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for menor que zero.

Exemplo: Assembly: `jl`
Linguagem de Máquina: **1**`11` **0001100 000 100**

JLE – Salta Execução se Menor Igual a Zero

`jle`

Descrição: A instrução *jle* faz um salto de execução para o endereço armazenado em %A, se o valor de D% for menor ou igual a zero.

Exemplo: Assembly: `jle`
Linguagem de Máquina: **1**`11` **0001100 000 110**

Tabela de Instruções:

Instrução Binária tipo A:

0 v_{15} v_{14} v_{13} v_{12} v_{11} v_{10} v_9 v_8 v_7 v_6 v_5 v_4 v_3 v_2 v_1

Instrução Binária tipo C:

1 1 1 a c_1 c_2 c_3 c_4 c_5 c_6 d_1 d_2 d_3 j_1 j_2 j_3

Intruções do tipo C sempre afetam as Flags: zr, ng.

Campo de Calculo:

caso a=0	c_1	c_2	c_3	c_4	c_5	c_6	caso a=1
0	1	0	1	0	1	0	0
1	1	1	1	1	1	1	1
-1	1	1	1	0	1	0	-1
D	0	0	1	1	0	0	D
A	1	1	0	0	0	0	(A)
!D	0	0	1	1	0	1	!D
!A	1	1	0	0	0	1	!(A)
-D	0	0	1	1	1	1	-D
-A	1	1	0	0	1	1	-(A)
D+1	0	1	1	1	1	1	D+1
A+1	1	1	0	1	1	1	(A)+1
D-1	0	0	1	1	1	0	D-1
A-1	1	1	0	0	1	0	(A)-1
D+A	0	0	0	0	1	0	D+(A)
D-A	0	1	0	0	1	1	D-(A)
A-D	0	0	0	1	1	1	(A)-D
D&A	0	0	0	0	0	0	D&(A)
D|A	0	1	0	1	0	1	D|(A)

Campo de Destino:

d_1	d_2	d_3	Destino
0	0	0	Null
0	0	1	(A)
0	1	0	D
1	0	0	A

Campo de Salto (Jump):

j_1	j_2	j_3	Salto
0	0	0	Null
0	0	1	jg
0	1	0	je
0	1	1	jge
1	0	0	jl
1	0	1	jne
1	1	0	jle
1	1	1	jmp

Instruções Regulares:

a	c_1	c_2	c_3	c_4	c_5	c_6	d_1	d_2	d_3	j_1	j_2	j_3	instrução
0	1	0	1	0	1	0	0	0	1	0	0	0	mov $0,(%A)
0	1	0	1	0	1	0	0	1	0	0	0	0	mov $0,%D
0	1	0	1	0	1	0	1	0	0	0	0	0	mov $0,%A
0	1	1	1	1	1	1	0	0	1	0	0	0	mov $1,(%A)
0	1	1	1	1	1	1	0	1	0	0	0	0	mov $1,%D
0	1	1	1	1	1	1	1	0	0	0	0	0	mov $1,%A
0	1	1	1	0	1	0	0	0	1	0	0	0	mov $-1,(%A)
0	1	1	1	0	1	0	0	1	0	0	0	0	mov $-1,%D
0	1	1	1	0	1	0	1	0	0	0	0	0	mov $-1,%A
0	0	0	1	1	0	0	0	0	1	0	0	0	mov %D,(%A)
0	0	0	1	1	0	0	1	0	0	0	0	0	mov %D,%A
0	1	1	0	0	0	0	0	0	1	0	0	0	mov %A,(%A)
0	1	1	0	0	0	0	0	1	0	0	0	0	mov %A,%D
0	0	0	1	1	0	1	0	1	0	0	0	0	not %D
0	1	1	0	0	0	1	1	0	0	0	0	0	not %A
0	0	0	1	1	1	1	0	1	0	0	0	0	neg %D
0	1	1	0	0	1	1	1	0	0	0	0	0	neg %A
0	0	1	1	1	1	1	0	1	0	0	0	0	inc %D
0	1	1	0	1	1	1	1	0	0	0	0	0	inc %A
0	0	0	1	1	1	0	0	1	0	0	0	0	dec %D
0	1	1	0	0	1	0	1	0	0	0	0	0	dec %A
0	0	0	0	0	1	0	1	0	0	0	0	0	add %D,%A,%A
0	0	1	0	0	1	1	1	0	0	0	0	0	sub %D,%A,%A
0	0	0	0	1	1	1	0	1	0	0	0	0	sub %A,%D,%D
0	0	0	0	0	0	0	1	0	0	0	0	0	and %D,%A,%A
0	0	1	0	1	0	1	1	0	0	0	0	0	or %D,%A,%A
1	1	1	0	0	0	0	0	1	0	0	0	0	mov (%A),%D,%D
1	1	1	0	0	0	0	1	0	0	0	0	0	mov (%A),%A,%A
1	0	0	0	0	1	0	0	1	0	0	0	0	add (%A),%D,%D
1	0	1	0	0	1	1	0	1	0	0	0	0	sub (%A),%D,%D
1	0	0	0	1	1	1	0	1	0	0	0	0	sub (%A),%D,%D
1	0	0	0	0	0	0	0	1	0	0	0	0	and (%A),%D,%D
1	0	1	0	1	0	1	0	1	0	0	0	0	or (%A),%D,%D
0	0	0	1	1	0	0	0	0	0	1	1	1	jmp
0	0	0	1	1	0	0	0	0	0	0	1	0	je
0	0	0	1	1	0	0	0	0	0	1	0	1	jne
0	0	0	1	1	0	0	0	0	0	0	0	1	jg
0	0	0	1	1	0	0	0	0	0	0	1	1	jge
0	0	0	1	1	0	0	0	0	0	1	0	0	jl
0	0	0	1	1	0	0	0	0	0	1	1	0	jle
0	1	0	1	0	1	0	0	0	0	0	0	0	nop