Primeiro Trabalho de POO
Emulador para o Processador Winter

Prof. Pedro Carlos da Silva Lara

Entrega: 21/10/2014

1 Informações Gerais

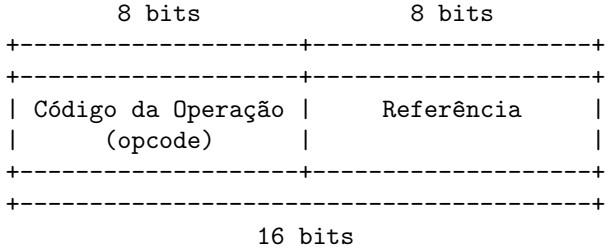
Winter é um processador hipotético especificado com fim puramente acadêmico. Ele possui dois registradores de 8 bits (1 byte): `acc` (acumulador) e `pc` (*program counter*) e 8 instruções. O formato das instruções são fixas, neste caso 16 bits. O Winter também conta com uma *flag* (`zero_flag`), esta flag assume o valor 1 caso o registrador acumulador seja igual a 0 e 1 caso contrário. Todos os detalhes acerca deste processador serão expostos nas próximas seções.

2 Registradores

O registrador acumulador (`acc`) recebe todos os resultados das operações lógicas e aritméticas. Sendo um registrador essencial em arquiteturas de 1 endereço (1 operando). O registrador `pc` (*program counter*) contém a posição de memória (endereço) onde se deve buscar a próxima instrução.

3 Formato das Instruções

O Winter suporta instruções fixas de 16 bits como especificado abaixo:



O campo código da operação ou `opcode` identifica unicamente a instrução que será realizada. Desta forma, cada instrução possui um `opcode` associado. O campo referência identifica o valor ou a localização dos dados que serão utilizados na instrução. Como o Winter possui apenas 8 instruções (veja Tabela 1), são necessários apenas 3 bits para a representação da operação. Neste caso, os 5 bits mais significativos do código de operação são sempre iguais a 0. Para o campo Referência podemos utilizar dados com 8 bits. Assim, podemos referenciar (endereçar) até 2^8 bytes na memória. O trecho de código abaixo mostra como funciona o algoritmo para buscar uma instrução na memória:

```
opcode = prg[pc]; // Busca a operação a ser realizada
pc = pc + 1;      // Incrementa o contador de programa
ref    = prg[pc]; // Busca o dado envolvido na instrução
pc = pc + 1;      // Incrementa o contador de programa
```

4 Instruções

Instrução	opcode	Comentário
load	0 (00000000_2)	memória
load	1 (00000001_2)	valor
store	2 (00000010_2)	–
add	3 (00000011_2)	–
sub	4 (00000100_2)	–
jnz	5 (00000101_2)	–
jz	6 (00000110_2)	–
hlt	7 (00000111_2)	fim do programa

Tabela 1. Instruções do processador Winter e seus respectivos opcodes.

4.1 LOAD

A instrução `load` opera de duas maneiras distintas: carrega um valor **direto** para o acumulador ou carrega um valor em **memória** para o acumulador. O trecho de código abaixo exibe as diferenças.
Exemplo de código:

```
load  10 ; acc = 10. Carrega o valor 10 no acumulador
load $10 ; acc = mem[10]. Carrega o valor que tem endereço 10 no acumulador

Formato de Código  : load $addr
Código da Operação : 00
Função             : Carrega o valor da posição de memória 'addr' em acc
Altera zero_flag   : Sim
```

Exemplo de instrução:

```
   opcode     addr
+----------+----------+
| 00000000 | 00001010 |
+----------+----------+
   load       $10

Formato de Código  : load num
Código da Operação : 01
Função             : Carrega o valor de 'num' em acc
Altera zero_flag   : Sim
```

Exemplo de instrução:

```
   opcode     addr
+----------+----------+
| 00000001 | 00001011 |
+----------+----------+
   load       11
```

4.2 STORE

```
Formato de Código  : store $addr
Código da Operação : 02
Função             : Armazena o valor do acumulador na posição de memória 'addr'
Altera zero_flag   : Não
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000010 | 10000111 |
+----------+----------+
   store       $135
```

4.3 ADD

Operação de **adição** entre o acumulador e um valor em memória.

```
Formato de Código  : add $addr
Código da Operação : 03
Função             : acc = acc + mem[addr]
Altera zero_flag   : Sim
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000011 | 00100101 |
+----------+----------+
    add       $37
```

4.4 SUB

Operação de **subtração** entre o acumulador e um valor em memória.

```
Formato de Código  : sub $addr
Código da Operação : 04
Função             : acc = acc - mem[addr]
Altera zero_flag   : Sim
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000100 | 11000010 |
+----------+----------+
    sub        $194
```

4.5 JNZ

Desvia caso a flag `zero_flag` seja igual a 0. Caso contrario, prossiga no código. Desta forma, o registrador `pc` será modificado se a flag `zero_flag` for igual a 0.

```
Formato de Código  : jnz #label
Código da Operação : 05
Função             : Se zero_flag == 0, desvia a execução para
                     a definição de 'label'
Altera zero_flag   : Não
```

Exemplo de instrução:

```
   opcode      ref.
+----------+----------+
| 00000101 | 01001110 |
+----------+----------+
    jnz        $78
```

4.6 JZ

Desvia caso a flag `zero_flag` seja igual a 1. Caso contrario, prossiga no código. Desta forma, o registrador `pc` será modificado se a flag `zero_flag` for igual a 1.

```
Formato de Código  : jz #label
Código da Operação : 06
Função             : Se zero_flag == 1, desvia a execução para
                     a definição de 'label'
Altera zero_flag   : Não
```

O assembler `vas` traduz cada *label* em uma posição do programa. **Exemplo de instrução:**

```
   opcode      ref.
+----------+----------+
| 00000110 | 00000111 |
+----------+----------+
    jz          7
```

No exemplo acima, se no momento que foi interpretada a função `jz` a flag `zero_flag` for 1 o registrador *program counter* deverá ser alterado para 7.

4.7 HLT

Indica o fim de um determinado programa.

```
Formato de Código  : hlt
Código da Operação : 07
Função             : Fim do programa
Altera zero_flag   : Não
```

Exemplo de instrução:

```
   opcode
+----------+----------+
| 00000111 | 00000000 |
+----------+----------+
    hlt
```

5 Exemplos

```
; soma acc = 10 + 20
load   20   ; acc = 20
store  $1   ; mem[1] = 20
load   10   ; acc = 10
add    $1   ; acc = acc + mem[1]
hlt         ; fim do programa

Programa executável:
   load       20     store      $1     load       10      add       $1      hlt
 00000001 00010100 00000010 00000001 00000001  00001010 00000011 00000001 00000111 00000000
|                 |
+-----16 bits-----+
```

O programa executável constitui apenas os valores. Se esse programa fosse visualizado por bytes em representação de base 16 (hexadecimal) teríamos o seguinte programa:

```
0x01 0x14 0x02 0x01 0x01 0x0A 0x03 0x01 0x07 0x00
```

```
; Exemplo de loop
; acc = 10 + 9 + 8 + ... + 1
load 0
store $1 ; $1 = 0

load 1
store $3 ; $3 = 1

load 10  ; acc = 10

#While:
    store $2 ; $2  = acc
    load  $1 ; acc = $1
    add   $2 ; acc = $2 + $1
    store $1 ; $1  = acc
    load  $2 ; acc = $2
    sub   $3 ; acc = acc - 1
    jnz #While
load $1

hlt
```

6 Instruções para o desenvolvimento do trabalho

O objetivo do trabalho é criar um emulador usando os conceitos de orientação a objetos para o processador Winter. O resultado final é um interpretador para os binários gerados pela ferramenta `wass` (disponibilizada no site). O trabalho deverá ser feita usando classes e objetos. É necessário a criação de uma classe denominada `Winter` usada para simular o comportamento do processador Winter. Esta classe deverá possuir um método `run()` que executa o binário. O progama final irá receber pela linha de comando o programa binário e irá executar o programa. Por exemplo, temos o programa em assembly `prog.asm`:
`$ wass prog.asm -o prog.bin` *gerando o programa binário*
`$ wemu prog.bin` *executando o programa*
Ao final da execução o programa deverá imprimir o valor dos registradores `pc, zero_flag e acc`.