Software Básico

Primeiro Trabalho

Prof. Pedro Carlos da Silva Lara
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca

Processador SP1 (Simple Processor 1)

1 Informações Gerais

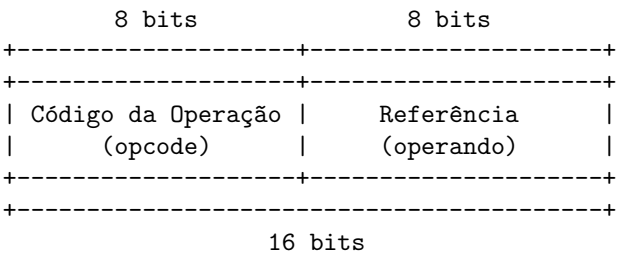
SP1 é um processador hipotético especificado com fim puramente acadêmico. Ele possui três registradores de 8 bits (1 byte) cada um: `acc` (acumulador) e `stat` (*status*) e `pc` (program counter). O ISA do processador SP1 é composto por 20 instruções. O formato das instruções são fixas, neste caso 16 bits (8 bits para o `opcode` e 8 bits para o operando). O registrador `stat` mantém informações de carry, overflow e zero do registrador `acc`. **O objetivo do trabalho é desenvolver um emulador, em linguagem `C`, para o processador SP1**. Os detalhes do SP1 serão vistos nas Seções seguintes.

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) baseada em acumulador. O registrador `pc` (*program counter*) contém a posição de memória (endereço) onde se deve buscar a próxima instrução.

3 Formato das Instruções

O SP1 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 SP1 possui apenas 20 instruções, seria necessário apenas 5 bits para a representação da operação. 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:

```
instrution = program[pc];           // Busca a instrução a ser realizada
opcode = instruction.opcode;
operand = instruction.operand;
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 (000000011_2)	–
sub	4 (00000100_2)	–
mul	5 (00000101_2)	–
div	6 (00000110_2)	–
inc	7 (00000111_2)	–
dec	8 (00001000_2)	–
and	9 (00001001_2)	–
or	10 (00001010_2)	–
not	11 (00001011_2)	–
jmp	12 (00001100_2)	–
jz	13 (00001101_2)	–
jnz	14 (00001110_2)	–
jg	15 (00001111_2)	–
jl	16 (00010000_2)	–
jge	17 (00010001_2)	–
jle	18 (00010010_2)	–
hlt	19 (00010011_2)	fim do programa

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
```

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
```

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 definida por 'addr'
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000010 | 10000111 |
+----------+----------+
   store       $135; mem[135] = acc
```

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]
```

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]
```

Exemplo de instrução:

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

4.5 MUL

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

```
Formato de Código  : mul $addr
Código da Operação : 05
Função             : acc = acc * mem[addr]
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000101 | 00100100 |
+----------+----------+
    mul       $36
```

4.6 DIV

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

```
Formato de Código  : div $addr
Código da Operação : 06
Função             : acc = acc / mem[addr]
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000110 | 00100100 |
+----------+----------+
    div       $36
```

4.7 INC

Incrementa em uma unidade o `acc`.

```
Formato de Código  : inc
Código da Operação : 07
Função             : acc = acc + 1
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00000111 | 00000000 |
+----------+----------+
    inc
```

4.8 DEC

Decrementa em uma unidade o `acc`.

```
Formato de Código   : dec
Código da Opervação : 08
Função              : acc = acc - 1
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00001000 | 00000000 |
+----------+----------+
    dec
```

4.9 AND

Operação de **E-lógico** entre o acumulador e um valor em memória.

```
Formato de Código  : and $addr
Código da Operação : 09
Função             : acc = acc & mem[addr]
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00001001 | 00000100 |
+----------+----------+
    and       $4
```

4.10 OR

Operação de **OU-lógico** entre o acumulador e um valor em memória.

```
Formato de Código  : or $addr
Código da Operação : 10
Função             : acc = acc | mem[addr]
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00001010 | 00001100 |
+----------+----------+
    or         $12
```

4.11 NOT

Operação de **negação** de todos os bits do acumulador: executa o complemento a 2 no acumulador.

```
Formato de Código  : not
Código da Operação : 11
Função             : acc = ~acc
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00001011 | 00001100 |
+----------+----------+
    not        $12
```

4.12 JMP

Desvio incondicional para o label.

```
Formato de Código  : jmp #label
Código da Operação : 12
Função             : desvia a execução para
                     a definição de 'label'
```

Exemplo de instrução:

```
   opcode      addr
+----------+----------+
| 00001100 | 00001101 |
+----------+----------+
    jmp        13
```

4.13 JZ

Desvia caso o resgistrador `acc` seja igual a 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for igual a 0.

```
Formato de Código  : jnz #label
Código da Operação : 13
Função             : Se acc == 0, desvia a execução para
                     a definição de 'label'
```

Exemplo de instrução:

```
   opcode      ref.
+----------+----------+
| 00001101 | 01001110 |
+----------+----------+
    jz         78
```

4.14 JNZ

Desvia caso o registrador `acc` seja diferente de 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for diferente de 0.

```
Formato de Código  : jz #label
Código da Operação : 14
Função             : Se acc != 0, desvia a execução para
                     a definição de 'label'
```

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

```
   opcode      ref.
+----------+----------+
| 00001110 | 00000111 |
+----------+----------+
    jz          7
```

4.15 JG

Desvia caso o registrador `acc` seja maior que 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for menor ou igual a 0.

```
Formato de Código  : jg #label
Código da Operação : 15
Função             : Se acc > 0, desvia a execução para
                     a definição de 'label'
```

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

```
   opcode      ref.
+----------+----------+
| 00001111 | 00000111 |
+----------+----------+
    jg          7
```

4.16 JL

Desvia caso o registrador `acc` seja menor que 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for maior ou igual a 0.

```
Formato de Código  : jg #label
Código da Operação : 16
Função             : Se acc < 0, desvia a execução para
                     a definição de 'label'
```

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

```
   opcode      ref.
+----------+----------+
| 00010000 | 00000111 |
+----------+----------+
    jl          7
```

4.17 JGE

Desvia caso o registrador `acc` seja maior ou igual a 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for menor que 0.

```
Formato de Código  : jg #label
Código da Operação : 17
Função             : Se acc >= 0, desvia a execução para
                     a definição de 'label'
```

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

```
   opcode      ref.
+----------+----------+
| 00010001 | 00000111 |
+----------+----------+
    jge          7
```

No exemplo acima, se no momento que foi executada a função `jz` o acumulador for zero *program counter* deverá ser alterado para 7.

4.18 JLE

Desvia caso o registrador `acc` seja menor ou igual a 0. Caso contrário, prossiga no código. Desta forma, o registrador `pc` será modificado se o `acc` for maior que 0.

```
Formato de Código  : jg #label
Código da Operação : 18
Função             : Se acc <= 0, desvia a execução para
                     a definição de 'label'
```

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

```
   opcode      ref.
+----------+----------+
| 00010010 | 00000111 |
+----------+----------+
    jge          7
```

No exemplo acima, se no momento que foi executada a função `jz` o acumulador for zero *program counter* deverá ser alterado para 7.

4.19 HLT

Indica o fim do programa.

```
Formato de Código  : hlt
Código da Operação : 19
Função             : Fim do programa
```

Exemplo de instrução:

```
   opcode
+----------+----------+
| 00010011 | 00000000 |
+----------+----------+
    hlt
```

5 Registrador STAT

```
STAT (8 bits)

+----------+
| xxxxxOCZ |
+----------+

O : Overflow
C : Carry
Z : Zero ACC
x : não utilizado
```

6 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 00010011 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 0x0F 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
```