Livrão de OAC

Thiago Tomás de Paula

setembro de 2022

Sumário

1 Introdução

1.1 O que é este arquivo?

1.2 O que é o RARS e o FPGRARS?

É necessário esclarecer alguns termos usados na subseção anterior, começando por Assembly: é qualquer linguagem de programação de baixo nível cujo nível se encontra logo acima da linguagem de máquina propriamente dita. O Assembly RISC-V é um exemplo particular dessa linguagem: a sigla RISC significa Reduced Instruction Set Computer, isto é, é uma linguagem Assembly com um pequeno conjunto de instruções. Mais do que isso, o RISC-V (V porque se trata da quinta geração do RISC) também tem uma arquitetura limpa e regular; por exemplo, todas as instruções e registradores possuem 32 bits (ou 64 bits, no caso do RISC-V 64), e existem apenas um punhado de tipos de instruções.

O RARS - RISC-V Assemble, Runtime, and Simulator - é um simulador baseado em Java de códigos em Assembly RISC-V 32 bits. No curso, usa-se uma versão modificada (customizada) do RARS, que possui capacidades extras como por exemplo converter o arquivo .s do programa em Assembly nos seus .mif's de dados e de instruções. Todavia, o RARS é lento e "cheio de bug" de forma que Leonardo Riether, durante seu curso de OAC e um pouco além dele, criou o FPGRARS, simulador de RISC-V baseado em Rust que busca ser similar ao RARS mas muito mais rápido (o FPG é sigla para Fast, Pretty Good). Na prática, é o FPGRARS que será usado para a grande maioria das execuções dos códigos feitos em OAC, mas o RARS ainda é muito útil devido às suas ferramentas de debug e geração dos arquivos .mif, essenciais para a implementação do trabalho final na FPGA.

1.3 O que é a FPGA?

A FPGA - Field-programmable gate array - é um circuito integrado composto por vários blocos lógicos que podem ser reconfigurados pelo usuário. Em OAC, a FPGA se tornará um processador compatível com RISC-V com uma de três arquiteturas: Uniciclo, Multiciclo e Pipeline [1]. Como o objetivo deste pdf é apresentar lógicas interessantes de resolução de problemas em Assembly RISC-V, iremos, em grande parte, deixar a FPGA em segundo plano, comentado sobre apenas quando oportuno. A primeira dessas oportunidades é comentar o propósito dos códigos `SYSTEMv21.s` e `MACROSv21.s`.

1.4 O que é o SYSTEM/MACROS?

Como comentado, a FPGA é apenas um processador, e carece de um sistema operacional (SO) que sirva de interface entre ela e o usuário. Isto gera um problema quando queremos rodar nela um .s que realize alguma chamada ao sistema, i. e., que apresente `ecall`em algum trecho do código. Essas *syscalls*, detalhadas no Help do RARS, ocorrem através de um mini SO particular ao aplicativo, e que não possui equivalente na placa. O mesmo ocorre para o FPGRARS. Sendo assim, é necessário para execução via FPGA definir cada `ecall`realizada como uma função.

O `SYSTEMv21.s` e o `MACROSv21.s` visam exatamente isso: recriar as `ecall`'s do RARS de maneira compatível à FPGA, sem contudo mudar a execução do código no RARS/FPGRARS. Para tanto, o `MACROS` deve ser incluído no início do `.text` do programa, e o `SYSTEM` ao final.

Listing 1: *Forma geral do uso do `SYSTEMv21.s` e `MACROSv21.s`*

```
        .data
```

[1] Vale notar que as reconfigurações da FPGA em processadores também foram feitas por alunos.

```
# estruturas de dados do aluno

.text
.include "MACROSv21.s"
# codigo em Assembly do aluno
.include "SYSTEMv21.s"
```

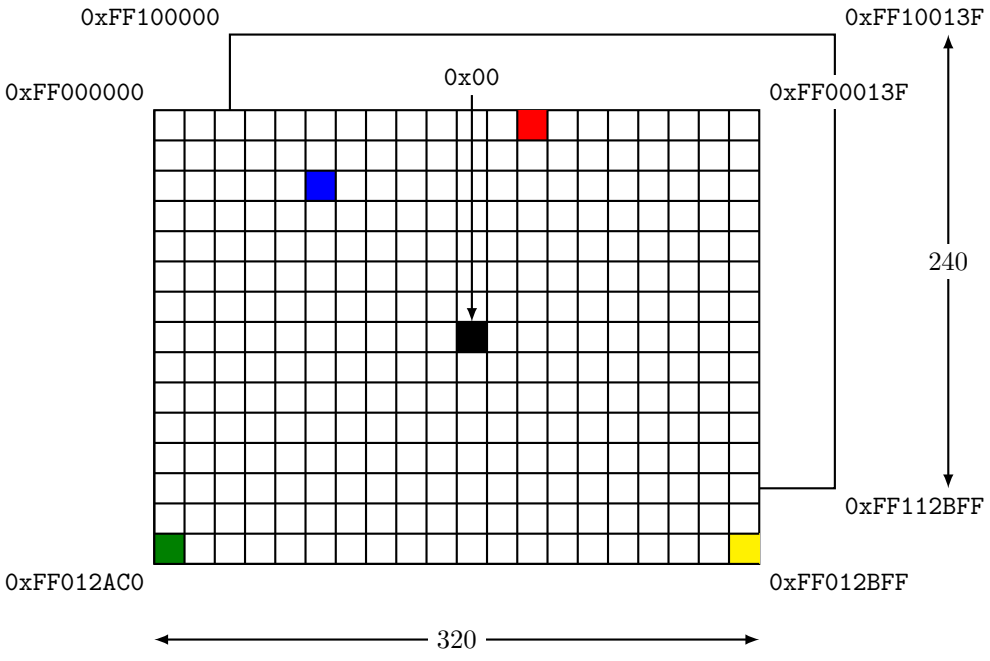
O `SYSTEMv21.s` em particular é um compilado de funções feitas por alunos de semestres passados procurando resolver alguma necessidade do trabalho. Pontualmente, iremos mergulhar a fundo nesse código para tentar explicar o funcionamento das funções mais utilizadas, e evitaremos comentar sobre o `MACROS`: é minha experiência que vale mais a pena extrair as funções necessárias do `SYSTEM` (fazendo as adaptações necessárias) do que importar tudo da dupla.

Por enquanto, deixemos essa conversa de lado e coloquemos a mão na massa, a começar pelo *bitmap*.

2 O Bitmap

É a tela de display do RARS/FPGRARS/FPGA, ou, mais precisamente, a forma como essa tela está arquitetada. É a união de dois grandes vetores de bytes, um com endereço inicial em `0xFF000000` o segundo com endereço inicial em `0xFF100000`. A tela do bitmap tem 320 bytes de largura por 240 bytes de altura (resolução 4:3), de forma que cada vetor citado tem endereço final em `0xFF012BFF` e `0xFF112BFF`, respectivamente. Perceba que $320 \times 240 = 76800 = 12C00_{16}$. Cada valor de byte num vetor codifica uma cor (veremos como em breve), e é conveniente colocar o primeiro byte (pixel) na quina superior esquerda da telinha, e ir formando ordenadamente o vetor até atingir a quina inferior direita, "pulando linha" a cada 320 pixeis. A Figura 1 esclarece o que foi dito até aqui.

Figura 1: *Bitmap do RARS/FPGRARS/FPGA, com alguns piexis coloridos para ilustração. O `0x00` se refere ao valor do byte, e não seu endereço.*



⚠ Escrevo pular linha entre aspas porque, a nível de vetor, bytes em linhas distintas podem ser na verdade adjacentes.

■ **Exemplo 2.1** O código abaixo guarda o byte da cor em `0xFF0096A0` no registrador `t0`, e o de `0xFF1096A0` em `t1`. Executando-o, vê-se que esses valores são ambos 0.

Listing 2: *Verificando bytes de cor*

```
.text
# t0 recebe o valor do byte em 0xFF0096A0
li  t0, 0xFF0096A0
lbu t0, 0(t0)
# t1 recebe o valor do byte em 0xFF1096A0
li  t1, 0xFF1096A0
lbu t1, 0(t1)
# encerramento por ecall
li  a7, 10
ecall
```

■

Exercício 2.1 Encontre os valores dos bytes de cor de todo o bitmap, incluindo ambos vetores. Abra o `Bitmap Display` no `Tools` do RARS e responda: qual a cor desse byte?
■

Em geral, gostaríamos de usar o bitmap para (i) apresentar alguma imagem na tela e (ii) apresentar alguma animação na tela. De fato, é por causa desse segundo desejo que o bitmap é composto por 2 vetores de cores, e não 1 só: pondo cada um num *frame*, animações se tornam melhor realizáveis. Vejamos o que isso quer dizer a seguir.

2.1 Frames

De maneira geral, uma animação é a rápida e organizada apresentação sucessiva de diferentes imagens estáticas, o que gera a percepção de movimento.

No bitmap, existem dois frames: o frame 0, que guarda o vetor de endereços `0xFF0...`, e o frame 1, que guarda o vetor de endereços `0xFF1...`. Para definir qual frame está sendo exibido, o RARS reserva o endereço de memória `0xFF200604`. Salvando 0 (`0x00`) nele, exibe-se o frame 0, e salvando 1, o 1. Inicialmente, o frame 0 é o mostrado.

■ **Exemplo 2.2** O programa abaixo torna o pixel em `0xFF1096A0` num pontinho verde, somente visível após a troca de frame.

```
.text
# colore o pixel em 0xFF1096A0 com a cor 50
li  t1, 0xFF1096A0
li  t0, 50
sb  t0, 0(t1)
# exibe o frame 1
ebreak
li  t0, 0xFF200604
li  t1, 1
sb  t1, 0(t0)
# programa e encerrado por ecall
li  a7, 10
ecall
```

■

Pelo apresentado até aqui, não é difícil imaginar como uma animação seria implementada em Assembly; por exemplo, podemos seguir os passos

1. Exibe-se o frame 0, que possui uma imagem
2. Coloca-se uma imagem no frame 1
3. Exibe-se o frame 1
4. Troca-se a imagem no frame 0
5. Volte ao passo 1

Exercício 2.2 Encontre o endereço do pixel no frame 0 que está "uma linha" acima do pontinho verde do Exemplo 2.2, e coloque nele a cor 255. Feito isso, siga os passos acima para fazer uma animação em loop infinito. *Sugestão:* use a `ecall` de sleep do RARS para que a animação não seja tão frenética. ■

Com isso dito, não entraremos em detalhes na animação por enquanto, mas primeiro procurar obter uma forma de converter arquivos de imagem em bytes de cor do RARS. Para este objetivo médio, será necessário explicar um assunto pendente desde o início desta seção: como os valores dos bytes de cor são feitos.

2.2 Conversão de RGB para byte

No tratamento de imagens, é comum identificar uma cor visível pela sua composição de vermelho, verde e azul, isto é, aferir como aquela cor seria obtida misturando-se apenas as quantidades acertadas de vermelho, verde e azul.

Matematicamente, uma cor pode ser identificada pela sua tripla RGB (Red, Green, Blue) que indicam, em valores inteiros de 0 a 255, a quantidade de vermelho (R), verde (G) ou azul (B) que foi usada na composição. Por exemplo, o vermelho puro teria RGB igual a (255, 0, 0), o preto igual a (0, 0, 0) e o branco, (255, 255, 255).

Note que até aqui estamos falando de 3 bytes distintos, enquanto que a cor no RARS é apenas 1 byte.

A conversão se dá da seguinte maneira: dado três bytes de um RGB, digamos, R, G e B, o byte do bitmap β é dado (em base 2) por

$$\begin{aligned}\beta &= R/\!/32 + (G/\!/32 \ll 3) + (B/\!/64 \ll 6) \\ &= bbgggrrr,\end{aligned} \tag{1}$$

onde $bb = B/\!/64$, $ggg = G/\!/32$, $rrr = R/\!/32$. Os $/\!/$ denotam divisão inteira e $\ll$ shifts lógicos para a esquerda.

■ **Exemplo 2.3** O byte 50 é `00110010` em base 2, de forma que $bb = 00_2 = 0$, $ggg = 110_2 = 6$, e $rrr = 010_2 = 2$. Um possível RGB para essa cor seria $R = 64$, $G = 192$, $B = 0$. ■

Note que a representação da cor no RARS perde muita informação em relação ao fornecido no RGB, especialmente no azul: dos 24 bits iniciais, restam apenas 8. Em todo caso, 256 cores deve ser uma paleta decente para os trabalhos finais.

2.2.1 O byte invisível

Dessas 256 cores, uma é especial: a cor invisível, de valor 199 (`0xC7`). No RARS, quando o `Bitmap Display` padrão detecta esse valor, é mostrado naquele endereço a cor do byte no outro frame. No FPGRARS (e na FPGA), o bitmap detecta esse valor e coloca no lugar o valor que estava lá antes da troca, ou seja, a cor é "ignorada".

■ **Exemplo 2.4** Para ilustrar esses efeitos, vamos revisitar o código no Exemplo 2.2. Dessa vez, vamos colocar o byte em `0xFF0096A0` como verde, o bytem em `0xFF1096A0` como invisível e trocaremos a frame. Note o loop infinito ao final para evitar a finalização do FPGRARS.

Listing 3: *Teste da cor invisível*

```
.text
# colore o pixel em 0xFF0096A0 com a cor 50
li   t1, 0xFF0096A0
li   t0, 50
sb   t0, 0(t1)
# colore o pixel em 0xFF1096A0 com a cor 199
li   t1, 0xFF1096A0
li   t0, 199
sb   t0, 0(t1)
# mostra o frame 1
li   t1, 0xFF200604
li   t0, 1
sb   t0, 0(t1)
# loop eterno
fpg:  j fpg
```

Executando o código no RARS, o ponto verde aparece mesmo com aquela cor pertencendo ao outro frame, enquanto que no FPGRARS a tela permanece escura. ■

Exercício 2.3 Encontre os valores *bb*, *ggg* e *rrr* e o RGB do byte `0xC7` e responda: se não fosse invisível, qual seria a cor desse byte? ■

Avisados sobre o comportamento suspeito da cor invisível, estamos prontos para lidar com imagens no bitmap.

2.3 Imprimindo imagens no bitmap

Essencialmente, imagens no bitmap são apenas organizações de pixeis de cor. O desafio aqui é sistematizar o carregamento desses bytes, o que pode ser feito usando uma estrutura de dados na memória (arquivo `.data`) que guarde

1. a largura da imagem, em bytes;
2. a altura da imagem, em bytes;
3. o vetor de cores daquela imagem.

■ **Exemplo 2.5** O `.data` de um retângulo 4x2 verde (cor 50) tem o seguinte conteúdo.

Listing 4: *Exemplo de `.data` sem cor invisível*

```
quadrado: .word 4, 2     # dimensoes
.byte                    # cores
50, 50, 50, 50,
50, 50, 50, 50
```

Note a ausência do `.text`: este arquivo deve ser escrito/incluído no campo `.data`. ■

Nesta configuração, as dimensões são words uma vez que a largura pode passar de 255 (1 byte cheio), e imagens que a princípio não são retangulares ficam com algum excesso de bytes de cor, que logicamente devem ser colocados como invisíveis.

■ **Exemplo 2.6** O `.data` de um segmento de reta branco com inclinação negativa é apresentado a seguir. Repare no posicionamento dos bytes invisíveis.

Listing 5: *Exemplo de `.data` com cor invisível*

```
reta: .word 2, 2     # dimensoes
.byte                # cores
255, 199,
199, 255
```

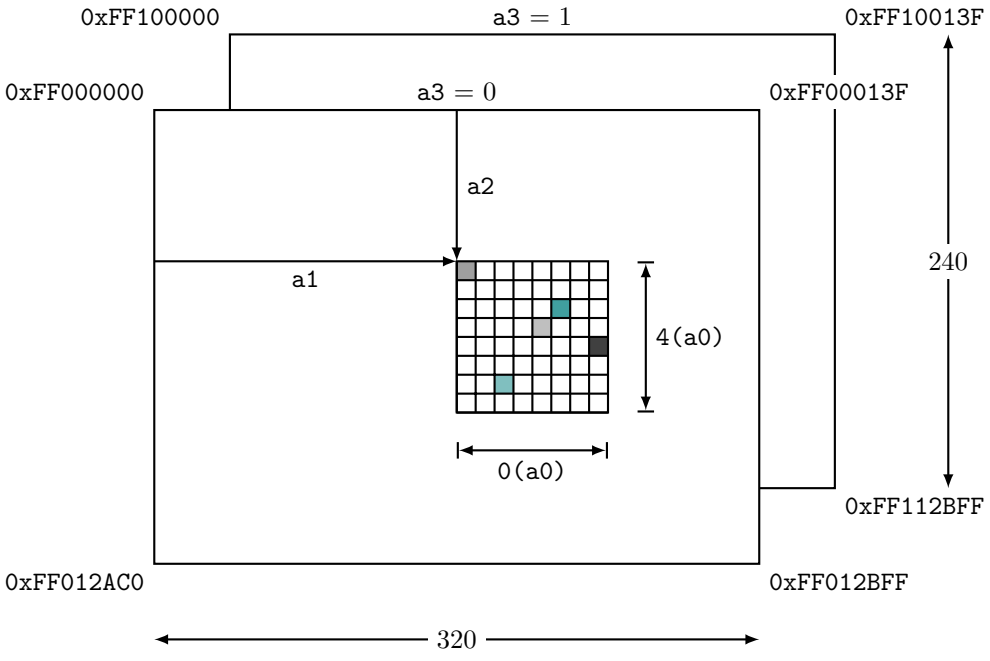
■

Estabelecido como os bytes de cor se organizam, é imediato implementar uma função que os imprimam. Chamemos de `Print` essa função; por desenho, ela deve receber em `a0` a label de um `.data` de imagem e imprimi-lo na posição (x, y) = `(a1, a2)` e frame `a3` do bitmap. O intervalo de valores válidos de cada registrador de argumento é resumido na tabela abaixo. A Figura 2 ilustra os argumentos da `Print`.

Tabela 1: *Argumentos da função `Print`*

Registrador	Função	Valor
`a0`	label/endereço do `.data`	-
`a1`	quantidade em x da posição da impressão	$[0, 320]$
`a2`	quantidade em y da posição da impressão	$[0, 240]$
`a3`	frame da impressão	0 ou 1

Figura 2: *Entendimento visual dos argumentos do `Print`.*



A ideia do `Print` será carregar no bitmap, byte a byte, as cores em `a0`. Para começar, criamos um código que coloque em `t0` o endereço inicial do bitmap de acordo com o frame em `a3`. Em outras palavras, `t0` = `0xFF000000` se `a3` = 0 e `t0` = `0xFF100000` se `a3` = 1.

```
    li    t0, 0xFF0   # carrega 0xFF0 em t0
    add   t0, t0, a3  # adiciona o frame a FF0
    slli  t0, t0, 20  # shift de 20 bits pra esquerda
```

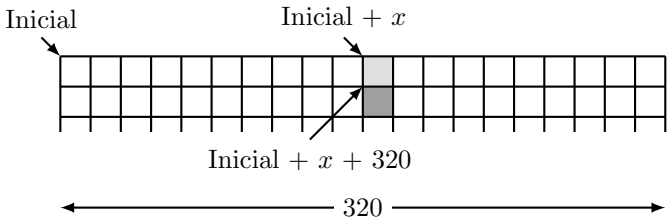
No `add`, `t0` vira `0xFF0` se `a3` $= 0$ e 0xFF1 se `a3` $= 1$. Em seguida, no `slli`, são acrescentados 4 bytes de zeros à direita de `t0`, e daí chegamos ao endereço inicial desejado.

É claro, não é esse endereço que nos importa, mas o (x, y) codificado por `a1` e `a2`. Como o acréscimo de um byte na altura corresponde a um pulo de 320 bytes no vetor de cores, não é difícil ver que um endereço de impressão terá a forma geral Final = Inicial + x + $320y$. No nosso `Print`, fazemos essa conta da seguinte maneira.

```
    add   t0, t0, a1    # adiciona x ao t0
    li    t1, 320       # t1 = 320
    mul   t1, t1, a2    # multiplica y por t1
    add   t0, t0, t1    # coloca o endereco em t0
```

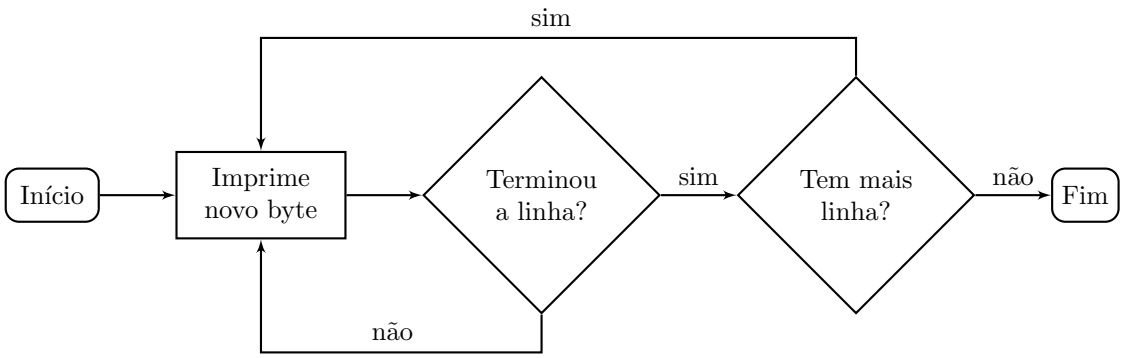
Caso a ideia ainda não tenha ficado claro, leia a Figura 3.

Figura 3: *Ilustração do endereço final quando $y = 1$.*



Doravante realizaremos a impressão das cores de fato, e já terminamos de usar os regs. `a1`, `a2`, e `a3`. A lógica da impressão segue o fluxograma (loop) abaixo.

Figura 4: *Fluxograma da lógica de impressão*



Antes de começar o loop, é necessário inicializar as variáveis de controle. Isto é feito a seguir.

```
    mv   t1, zero    # zera t1
    mv   t2, zero    # zera t2
    mv   t6, a0      # data em t6 para nao mudar a0
    lw   t3, 0(t6)   # carrega a largura em t3
    lw   t4, 4(t6)   # carrega a altura em t4
    addi t6, t6, 8   # primeira cor em t6
```

Mais precisamente, `t6` tem o *endereço* da primeira cor. Finalmente entrando no loop, a primeira coisa a se fazer é carregar o byte em `t6` e colocá-lo no bitmap, i. e., em `t0`. Feito isso, vamos para o endereço da próxima cor, atualizando os endereços de impressão.

```
      PrintLinha:
        lbu   t5, 0(t6)   # carrega em t5 um byte da imagem
        sb    t5, 0(t0)   # imprime no bitmap o byte da imagem
        addi  t0, t0, 1   # incrementa endereco do bitmap
        addi  t6, t6, 1   # incrementa endereco da imagem
        addi  t2, t2, 1   # incrementa contador de coluna
        blt   t2, t3, PrintLinha # cont da coluna < largura ?
```

O branch ao final do listing garante iremos para a próxima linha de cores somente quando a largura da linha sendo analisada for completada. Quando a linha está completa, precisamos ir para a próxima caso a próxima exista, e repetir o processo.

```
        addi  t0, t0, 320         # t0 += largura do bitmap
        sub   t0, t0, t3          # t0 -= largura da imagem
        mv    t2, zero            # zera t2 (cont de coluna)
        addi  t1, t1, 1           # incrementa contador de linha
        bgt   t4, t1, PrintLinha # altura > contador de linha ?
        ret                       # retorna
```

O `bgt` é exatamente a verificação da existência de uma próxima linha, e o `ret` é uma pseudo-instrução que retorna a função procedimento que a chamou. É necessário o `t0 -=` para que a impressão da próxima linha não começe logo abaixo o último byte impresso. Juntando os trechos de códigos feitos até agora, a função `Print` fica completa.

Listing 6: *Função `Print` completa*

```
      Print:
        li    t0, 0xFF0   # carrega 0xFF0 em t0
        add   t0, t0, a3  # adiciona o frame a FF0
        slli  t0, t0, 20  # shift de 20 bits pra esquerda

        add   t0, t0, a1  # adiciona x ao t0
        li    t1, 320     # t1 = 320
        mul   t1, t1, a2  # multiplica y por t1
        add   t0, t0, t1  # coloca o endereco em t0

        mv  t1, zero      # zera t1
        mv  t2, zero      # zera t2
        mv  t6, a0        # data em t6 para nao mudar a0
        lw  t3, 0(t6)     # carrega a largura em t3
        lw  t4, 4(t6)     # carrega a altura em t4
        addi t6, t6, 8    # primeira cor em t6

      PrintLinha:
        lbu   t5, 0(t6)   # carrega em t5 um byte da imagem
        sb    t5, 0(t0)   # imprime no bitmap o byte da imagem
        addi  t0, t0, 1   # incrementa endereco do bitmap
        addi  t6, t6, 1   # incrementa endereco da imagem
        addi  t2, t2, 1   # incrementa contador de coluna
        blt   t2, t3, PrintLinha # cont da coluna < largura ?

        addi  t0, t0, 320 # t0 += largura do bitmap
        sub   t0, t0, t3  # t0 -= largura da imagem
        mv    t2, zero    # zera t2 (cont de coluna)
        addi  t1, t1, 1   # incrementa contador de linha
```

```
        bgt    t4, t1, PrintLinha # altura > contador de linha ?
        ret                  # retorna
```

■ **Exemplo 2.7** Definidas os `.datas quadrado`, `reta` e agora a função `Print`, podemos criar o desenho de uma reta branca atravessando um retângulo verde através do código abaixo.

Listing 7: *Uso do `Print`*

```
        .data
        .include "quadrado.data"
        .include "reta.data"

        .text
        li     a3, 0           # frame fixado em 0
        la     a0, quadrado    # a0 recebe label do quadrado

        li     a1, 160         # x = 160
        li     a2, 120         # y = 120
        jal    Print           # primeiro quadrado eh impresso
        li     a2, 122         # y = 122
        jal    Print           # segundo quadrado eh impresso

        la     a0, reta        # a0 recebe label da reta
        li     a1, 160         # x = 160
        li     a2, 120         # y = 120
        jal    Print           # primeira reta eh impressa
        li     a1, 162         # x = 162
        li     a2, 122         # y = 122
        jal    Print           # segunda reta eh impressa

        fpg: j fpg             # fim do programa

        .include "Print.s"
```

■

Exercício 2.4 Modifique o `Print` de forma que, se o byte de cor na label em `a0` for invisível, a função não imprima nada no bitmap, e continue o processo como normal. ■

⚠ A impressão byte a byte de um `.data` deixa a impressão robusta contra desalinhamento, mas também mais lenta, visto que poderíamos realizar a impressão de 2 em 2 bytes ou até de 4 em 4 bytes por vez.

Exercício 2.5 Modifique o `Print` de forma que sejam impressos 4 bytes de cor por vez. Rode o programa no exemplo acima com a função modificada. Verifique que a execução gera erro no RARS, e resultado inesperado no FPGRARS. ■

Exercício 2.6 Modifique o `Print` de forma que sejam impressos 2 bytes de cor por vez. Rode o programa no exemplo acima com a função modificada, e verifique que a execução não gera erro nem no RARS e nem no FPGRARS. ■

2.3.1 Conversão de imagem para .bmp

Ter de criar seus próprios `.datas` do zero é uma tarefa laboriosa e ingrata, especialmente quando estamos apenas tentando copiar uma imagem já existente. Ao longo do curso, foram criados alguns executáveis que fazem a conversão de um arquivo de imagem ou `.bmp` no `.data` correspondente, exatamente no formato usado aqui. A lista desses aplicativos é dada abaixo.

Em particular, acrescento na lista também o `paint.net`, que converte arquivos de imagem comum (`.png`, `.jpg`, etc.) em `.bmp`, formato de imagem próprio do bitmap. Aos `.bmp`'s e/ou `.data`'s correspondentes daremos o nome de *sprites*.

Tabela 2: *Programas úteis para a criação de* `.datas`

Nome	Conversão	Criadores	SO
`paint.net`	imagem → `.bmp`	Rick Brewster	Windows
`bmp2oac3`	`.bmp` → `.data`	prof. Lamar	Windows

⚠ Após converter uma imagem para `.bmp` no paint.net, save o arquivo com intensidade de 24 bits. Caso contrário, o bmp2oac3 dará um `.data` inesperado.

Quando for necessário alterar o desenho de uma sprite, recomenda-se o software GraphicsGale para a edição.

Para completar a seção, é interessante apresentar uma conversão de imagem e a sua impressão no bitmap. Digamos que a sprite desejada venha da imagem abaixo.

Figura 5: *Nanachi de Made in Abyss – imagem original.*

Ela está dimensionada para ter 320 pixeis de largura por 180 pixeis de altura, de forma a caber no bitmap padrão. Usando o `paint.net` para gerar a cópia em `.bmp`, executar o comando `bmp2oac3.exe Nanachi.bmp` gera o arquivo `Nanachi.data`. Finalmente, o código abaixo tem output mostrado na Figura 6.

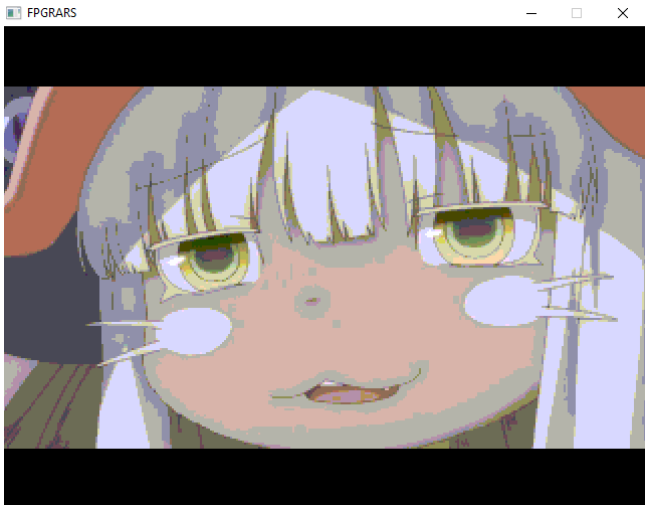
Listing 8: *Impressão de sprite gerada a partir de imagem. Note que tanto* `Nanchi.data` *quanto o* `Print.s` *estão no mesmo diretório deste programa. Em* `Print.s` *está escrito apenas a função* `Print.s` *discutida anteriormente.*

```
.data
  .include "Nanachi.data"
```

```
        .text
          la   a0, Nanachi # label da sprite a ser impressa
          li   a1, 0       # pos em x
          li   a2, 30      # pos em y
          li   a3, 0       # frame
          jal Print        # impressao no bitmap
          fpg: j fpg       # loop eterno para o fpgrars

          .include "Print.s"
```

Figura 6: *Nanachi de Made in Abyss – imagem no bitmap.*



Infelizmente a sprite final fica com qualidade menor que a imagem original. Isso ocorre a princípio, devido ao modo como RGBs se tornam bytes de cor (assunto já detalhado): tudo o mais constante, $R = 100$ e $R = 127$ geram a mesma cor no bitmap.

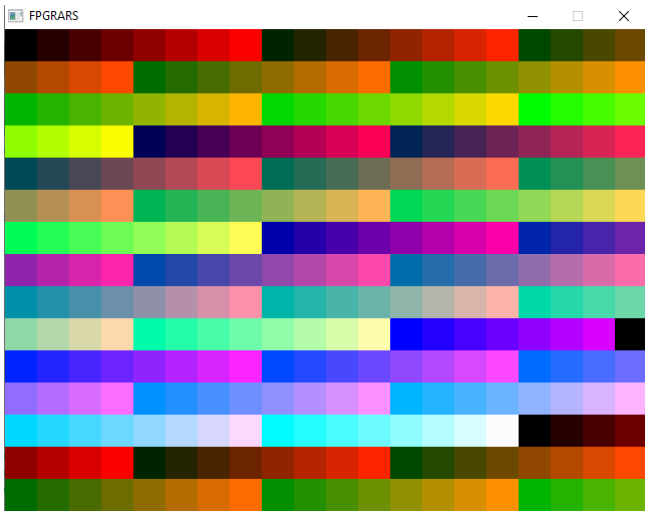
Para contornar essa limitação na representação de cores, evite usar cores muito próximas como branco e azul muito claro, preto e vermelho escuro, etc.

Com os conhecimentos adquiridos até aqui, não é difícil criar um programa que mostre no bitmap as 256 cores usadas pelo RARS. Uma forma de fazer isso é criar uma tile 16x16 cujo `.data` é inteiramente um byte de cor, e variar esse byte até chegar no valor 256.

A Figura 7 é um resultado desse algoritmo. Nela, a cor 0 está na quina superior esquerda, a cor 1 logo à sua direita, e por aí vai, até chegar ao fim do bitmap.

Perceba que a cor 199 é a cor invisível e por isso apresenta a cor inicial da tela, o preto; note também que as cores voltam a se repetir após atingir o 256 (branco).

Figura 7: *Cores do RARS em ordem crescente. O código executado para chegar a esta tela está presente no apêndice.*



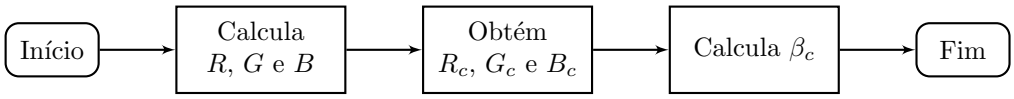
2.4 Curiosidade: como tornar uma cor cinza

Conhecido a fórmula de criação de bytes de cor, Equação 1, é simples arranjar a versão cinza de cada cor. Para tanto, sejam (R, G, B) o RGB original e (R_c, G_c, B_c) o RGB cinza associado. Então

$$R_c = G_c = B_c = \frac{R + G + B}{3}$$

e a obtenção do byte cinza é direta; basta seguir o fluxograma abaixo, por exemplo.

Figura 8: *Fluxograma da lógica de acinzentar uma cor. β_c é o valor (byte) da cor cinza associada a R, G, B. Este é o fluxograma que a função* ***`ByteCinza`*** *segue.*



Na sequência, apresentam-se rotinas que juntas com a `Print` realizam a impressão da versão cinza de uma sprite. Dito isso, é interessante que o leitor tome como exercício de fixação realizar a sua própria função `ByteCinza`.

Listing 9: *`ByteCinza`, função que recebe um byte de cor em `t5` e retorna a versão cinza desse byte também em `t5`.*

```
ByteCinza:
  addi  sp, sp, -20 # expansao da pilha
  sw   ra, 0(sp)     # armazenamento de registradores
  sw   t0, 4(sp)
  sw   t1, 8(sp)
  sw   t2, 12(sp)
  sw   t3, 16(sp)

  li   t3, 0x000000c7
  beq t5, t3, fimByteCinza  # cor invisivel -> nao a acinzentamos
  jal byte2RGB       # t0 <- R, t1 <- G, t2 <- B
```

```
      add t0, t0, t1     # t0 <- R + G
      add t0, t0, t2     # t0 <- R + G + B
      li  t1, 3          # t1 = 3
      divu  t0, t0, t1   # t0 <- media[R,G,B]
      mv  t1, t0         # t1 <- G=R=media
      mv  t2, t0         # t2 <- B=R=media

      jal RGB2byte       # t5 <- versao cinza do byte original

fimByteCinza:
      lw  ra, 0(sp)      # recuperacao de registradores
      lw  t0, 4(sp)
      lw  t1, 8(sp)
      lw  t2, 12(sp)
      lw  t3, 16(sp)
      addi  sp, sp, 20   # contracao da pilha
      ret                # retorno por pseudo
```

⚠ A pseudo instrução `call` é uma versão mais poderosa da `jal`, mas ela **também modifica** `t1`. Por causa disso, da forma como a `ByteCinza` foi feita, não podemos chamar a `RGB2byte` via `call`.

A `byte2RGB` obtém a partir do byte de cor original os valores R, G, B associados, e coloca-os em `t0`, `t1` e `t2`, nessa ordem. Por outro lado, a `RGB2byte` recebe o RGB em `t0`, `t1` e `t2` e coloca o byte de cor associado em `t5`.

Confira essas rotinas a seguir.

Listing 10: *`byte2RGB`. Recebe um byte de cor em `t5` e retorna seu RGB em `t0`, `t1`, `t2`.*

```
    byte2RGB:
      # bits de vermelho  # t5 =  bbgggrrr
      andi  t0, t5, 0x07  # t0 <- 0000 0rrr
      li  t3, 32          # t3 <- 32
      mul t0, t0, t3      # t0 <- rrr*32 = R
      # bits de verde
      andi  t1, t5, 0x38  # t1 <- 00GG G000
      srli  t1, t1, 3     # t1 <- 00000ggg
      #li t3, 32          # t0 <- 32
      mul t1, t1, t3      # t1 <- ggg*32 = G
      # bits de azul
      srli  t2, t5, 6     # t0 <- 000000bb
      li  t3, 64          # t3 <- 64
      mul t2, t2, t3      # t2 <- bb*64 = B

      ret                 # retorno por pseudo
```

Listing 11: *`RGB2byte`. Recebe um RGB em `t0`, `t1`, `t2`, e retorna seu byte de cor em `t5`.*

```
    RGB2byte:
      # bits de vermelho
      li  t3, 32          # t3 <- 32
      divu  t0, t0, t3    # t0 <- R/32 = 0...000 rrr
      # bits de verde
      divu  t1, t1, t3    # t1 <- G/32 = 0...000 ggg
      slli  t1, t1, 3     # t1 <- 0...00 ggg000
      # bits de azul
```

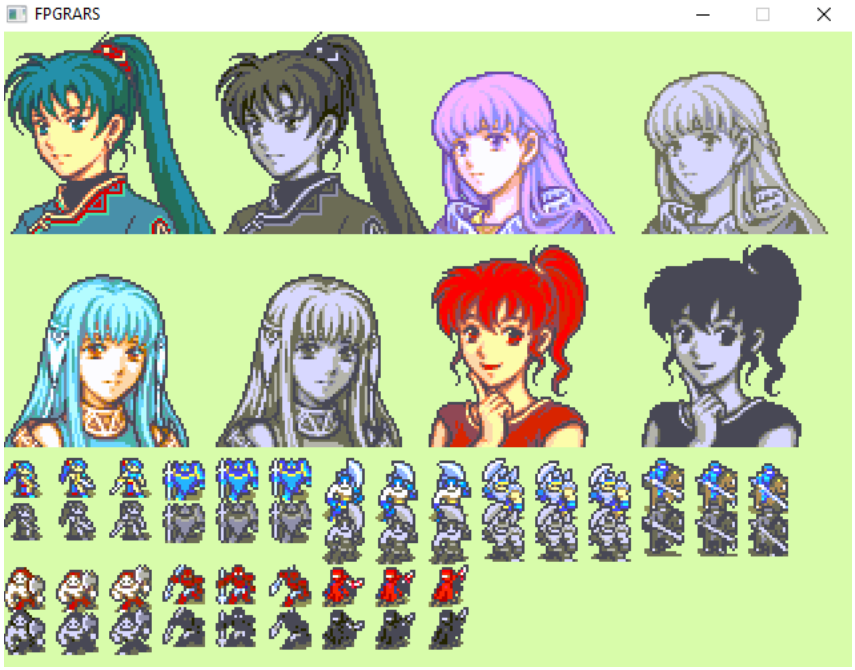
```
    li   t3, 64             # t3 <- 64
    divu   t2, t2, t3       # t2 <- B/64 = 0...000 0bb
    slli   t2, t2, 6        # t2 <- 0...0 bb000000
    # montagem de t5
    add t5, x0, t0          # t5 <- 0...0 00000rrr
    add t5, t5, t1          # t5 <- 0...0 00gggrrr
    add t5, t5, t2          # t5 <- 0...0 bbgggrrr

    ret                     # retorno por pseudo
```

Exercício 2.7 Crie a função `PrintCinza`, que consiste na modificação da `Print` de forma que todo byte de cor (exceto o invisível) obtido do `.data` em `a0` seja impresso cinza no bitmap. ■

A seguir, apresentam-se alguns resultados da `PrintCinza` definida no exercício acima.

Figura 9: *Sprites antes e depois da conversão para cinza.*



2.5 Criando animações no bitmap

Na Seção 2.1 foi discutido uma maneira simples porém rudimentar de criar uma animação usando o bitmap. Agora, apresentaremos duas maneiras mais adequadas para o trabalho final: animação em intervalos regulares com sleep e animação em intervalos distintos sem sleep.

Por "mais adequadas", entende-se funções inseridas num loop, que no caso do trabalho é o loop do jogo. Chamaremos esse loop de `GameLoop`.

2.5.1 Animação em intervalos regulares

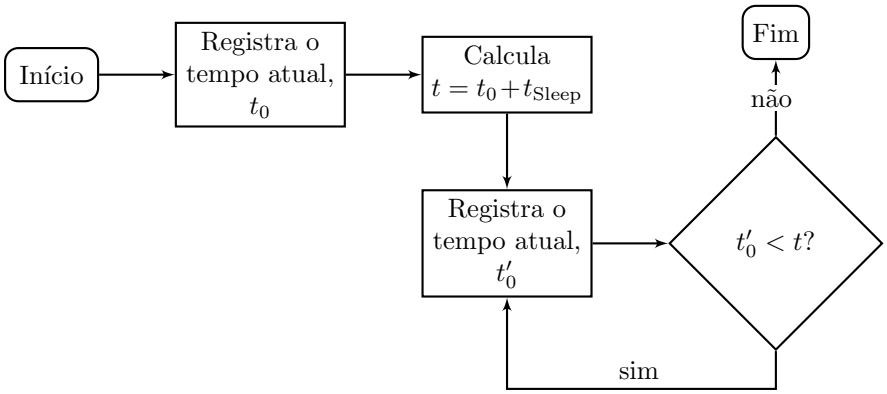
Digamos que desejamos criar a animação *idle* de um personagem de 3 poses diferentes. Por animação em intervalos regulares, entende-se que cada pose durará um certo tempo t em

ms. Neste contexto, é confortável usar a `ecall`de sleep do RARS, que congela a execução do código por `a0` milissegundos.

Infelizmente, chamadas ao sistema não funcionam na FPGA, de forma que precisamos criar nossa própria função `Sleep`, o que é tranquilamente satisfeito usando o registrado de status `time`. Este registrador guarda a quantidade de ms despendida desde o início da execução do programa.

Com isso em mente, é razoável pensar no seguinte fluxograma para a `Sleep`.

Figura 10: *Lógica do `Sleep`*



A partir da lógica exposta acima, ficamos com a seguinte função `Sleep`.

Listing 12: *Rotina de sleep compatível com a FPGA.*

```
Sleep:
  csrr   t0, time             # le o tempo do sistema
  add    t1, t0, a0           # soma com o tempo solicitado
SleepLoop:
  csrr   t0, time             # le o tempo do sistema
  sltu   t2, t0, t1
  bne    t2, zero, SleepLoop # t0<t1 ?
  ret
```

⚠ Economizaríamos um registrador, e duas instruções, se usássemos `bltu` ao invés do `sltu` e `bne`. Infelizmente, por motivo desconhecido, essa versão do `Sleep` não funciona no RARS.

Obtido o sleep, passamos à animação de fato. Lembre-se, queremos criar animação num contexto de loop, isto é, criar uma rotina `Animacao` inserida num loop "eterno" como abaixo.

```
GameLoop:
  # ...
  call Animacao
  # ...
  j GameLoop
```

Uma maneira de implementar a animação se inspira na máquina de estados finita: associamos a cada pose um estado, i. e., um número de 0 a 2, e imprimimos a pose de acordo com o estado. Por estética, é importante usar os frames de maneira inteligente, trocando para o próximo frame somente depois da sprite ser impressa lá. Finalmente, antes da impressão de qualquer pose precisamos antes limpar aquele espaço.

Por exemplo, limpar pode significar imprimir uma tile do mapa sobre o espaço desejado. Resumindo, nosso sistema de animação segue a seguinte ordem:

I. chamada à função: estado atual num frame (digamos, 0) com espaço livre no outro frame;

II. armazenamento na pilha do espaço livre;

III. 1. atualiza o estado;
   2. imprime, no outro frame, a pose de acordo com o estado;
   3. troca o frame sendo exibido;

IV. impressão da sprite armazenada na pilha (etapa II) sobre a pose no frame original (no caso, 0);

V. retorno da função.

O fluxograma e as figuras abaixo resumem e ilustram o discutido até aqui.

Figura 11: *Fluxograma da animação*

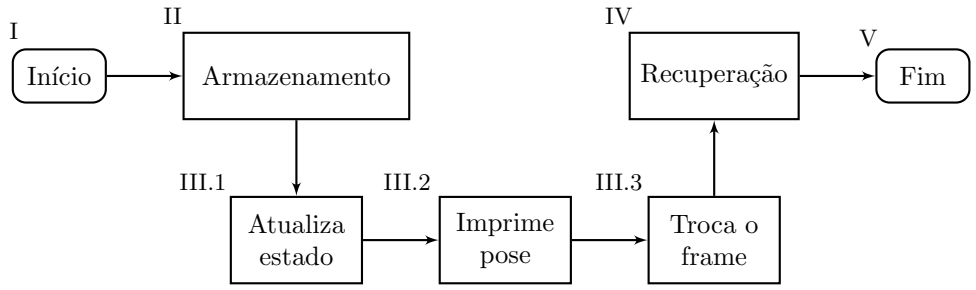
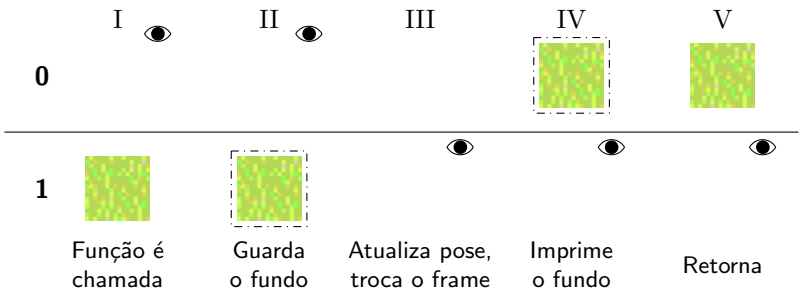


Figura 12: *Ilustração da animação. Os números arábicos indicam o frame, e os olhos indicam qual frame está sendo exibido.*



Tendo em vista o apresentado acima, faz sentido compor o `Animacao`, principalmente, por 3 funções, cujos nomes deixam claro objetivos: `GuardaFundo`, `AtualizaPose` e `ImprimeFundo`.

Para que a animação funcione adequadamente, (i) precisamos saber qual frame está sendo exibido; (ii) criaremos ainda uma label `personagem` que guarda a posição (x, y) do fundo/pose no bitmap e o estado daquele personagem. Com isso dito, definimos com as seguintes tabelas de argumentos.

Tabela 3: *Argumentos da* ***Animação***

Registrador	Conteúdo
`s0`	frame atual
`a0`	label do personagem

Tabela 4: *Argumentos da* ***GuardaFundo***, ***ImprimeFundo***, ***AtualizaPose***

Registrador	Conteúdo
`s0`	frame atual
`a1`	x do personagem
`a2`	y do personagem
`a3`	estado do personagem

Como estamos trabalhando com mudança de poses em intervalos iguais e usando sleep, `AtualizaPose` sempre colocará o estado atual como estado anterior + 1 mod 3 quando for chamada; daí, o estado 0 vai para 1, 1 vai para 2, e 2 vai para 0.

Uma maneira de escrever essa rotina é mostrada abaixo.

Listing 13: *Função* ***AtualizaPose***, *reponsável por implementar toda a etapa III da animação.*

```
AtualizaPose:
  # uso da Print -> ra, a0 e a3 devem ser salvos
  addi   sp, sp, -12
  sw     ra, 0(sp)
  sw     a0, 4(sp)
  sw     a3, 8(sp)

  addi   a3, a3, 1
  li     t0, 3
  remu   a3, a3, t0       # estado atual = estado anterior + 1 (mod 3)
  sw     a3, 8(a0)        # atualiza na label

  # decide a sprite a ser impressa de acordo com o estado
  beqz   a3, Lyn0         # estado = 0?
  li     t0, 1
  beq    a3, t0, Lyn1     # estado = 1?

  la     a0, Lyn3StandBy # estado = 2

fimAtualizaPose:
  xori   s0, s0, 1
    mv     a3, s0         # a3 <- frame oposto ao atual
  call   Print            # impressao da pose no outro frame
  li     t0, 0xFF200604
  sw     s0, 0(t0)        # exibe o outro frame

  lw     ra, 0(sp)
  lw     a0, 4(sp)
  lw     a3, 8(sp)
  addi   sp, sp, 12
  ret                     # retorno
```

```
Lyn0: la  a0, Lyn1StandBy
  j fimAtualizaPose
Lyn1: la  a0, Lyn2StandBy
  j fimAtualizaPose
```

Quanto ao `GuardaFundo` e `ImprimeFundo`, podemos nos basear na `Print`, apenas usando a pilha no lugar do bitmap/`a0`, respectivamente. Outra diferença é que aqui vamos assumir largura e altura da sprite de fundo fixas e iguais a 16, para simplificar o código.

Confira uma possível implementação do `GuardaFundo` abaixo.

Listing 14: *Função `GuardaFundo`. Note como o fundo é armazenado na pilha*

```
GuardaFundo:
    li    t0, 0xFF0   # carrega 0xFF0 em t0
    xori  t1, s0, 1 # t1 <- valor do outro frame
    add   t0, t0, t1  # adiciona o frame a FF0
    slli  t0, t0, 20  # shift de 20 bits pra esquerda

    add   t0, t0, a1  # adiciona x ao t0
    li    t1, 320     # t1 = 320
    mul   t1, t1, a2  # multiplica y por t1
    add   t0, t0, t1  # coloca o endereco em t0

    mv  t1, zero      # zera t1
    mv  t2, zero      # zera t2
    li  t3, 16        # carrega a largura em t3
    li  t4, 16        # carrega a altura em t4

GuardaLinha:
    lbu   t5, 0(t0)   # carrega em t5 um byte da imagem
    addi  sp, sp, -1  # expande a pilha em 1 byte
    sb    t5, 0(sp)   # guarda na pilha um byte da imagem
    addi  t0, t0, 1   # incrementa endereco do bitmap
    addi  t2, t2, 1   # incrementa contador de coluna
    blt   t2, t3, GuardaLinha # cont da coluna < largura ?

    addi  t0, t0, 320 # t0 += 320
    sub   t0, t0, t3  # t0 -= largura da imagem
    mv    t2, zero    # zera t2 (cont de coluna)
    addi  t1, t1, 1   # incrementa contador de linha
    bgt   t4, t1, GuardaLinha # altura > contador de linha ?
    ret               # retorna
```

Dada a implementação acima, se a `ImprimeFundo` simplesmente tirasse os bytes da pilha a partir da posição onde o `GuardaFundo` a deixou, a sprite do fundo sairia "de trás pra frente" no bitmap. Por causa disso, vamos usar o registrador `t6` para ajustar a pilha quando necessário no `ImprimeFundo`, veja o próximo listing (e compare-o com o do `GuardaFundo`).

Listing 15: *Função `ImprimeFundo`.*

```
ImprimeFundo:
  li    t0, 0xFF0     # carrega 0xFF0 em t0
    xori  t1, s0, 1   # t1 <- valor do outro frame
    add   t0, t0, t1  # adiciona o frame a FF0
    slli  t0, t0, 20  # shift de 20 bits pra esquerda

    add   t0, t0, a1  # adiciona x ao t0
    li    t1, 320     # t1 = 320
```

```
    mul   t1, t1, a2  # multiplica y por t1
    add   t0, t0, t1  # coloca o endereco em t0

    mv  t1, zero      # zera t1
    mv  t2, zero      # zera t2
    li  t3, 16        # carrega a largura em t3
    li  t4, 16        # carrega a altura em t4

    mul t6, t3, t4    # t6 <- quantidade de pixels
    add t6, sp, t6    # t6  <- endereco do primeiro byte do fundo

ImprimeLinha:
    addi  t6, t6, -1  # vai para o proximo pixel do fundo
    lbu   t5, 0(t6)   # carrega em t5 um byte da pilha
    sb    t5, 0(t0)   # imprime no bitmap um byte da imagem
    addi  t0, t0, 1   # incrementa endereco do bitmap
    addi  t2, t2, 1   # incrementa contador de coluna
    blt   t2, t3, ImprimeLinha # cont da coluna < largura ?

    addi  t0, t0, 320 # t0 += 320
    sub   t0, t0, t3  # t0 -= largura da imagem
    mv    t2, zero    # zera t2 (cont de coluna)
    addi  t1, t1, 1   # incrementa contador de linha
    bgt   t4, t1, ImprimeLinha # altura > contador de linha ?

    # ajusta a pilha
    mul t6, t3, t4    # t6 <- quantidade de pixels
    add sp, sp, t6    # sp fecha
    ret               # retorna
```

Finalmente, obtemos o seguinte para a `Animacao`,

Listing 16: *Função `Animacao`*

```
Animacao:
    addi  sp, sp, -4
    sw    ra, 0(sp)

    call  GuardaFundo
    lw    a3, 8(a0) # a3 <- estado atual
    call  AtualizaPose
    call  ImprimeFundo

    lw    ra, 0(sp)
    addi  sp, sp, 4
    ret
```

que foi projetado para ser chamado num loop como o abaixo.

Listing 17: *Inicialização e `GameLoop`. `Sleep` é a mesma função do início desta subseção.*

```
.data               # x    y    estado
    personagem: .word 152, 112, 0
    .include "sprites/Lyn_datas.data"
    .include "sprites/TileGrama.data"
.text
    # inicializacao da animacao, s0 e a0
    li  t0, 0xFF200604
    lw  s0, 0(t0)     # s0 = frame sendo mostrado
```

```
    la   a0, TileGrama    # a0 <- sprite do fundo
    li   a1, 152          # x
    li   a2, 112          # y
    mv   a3, zero         # frame 0
    call  Print
    li   a3, 1            # frame 1
    call  Print
    mv   a3, s0           # frame exposto
    la   a0, Lyn1StandBy # a0 <- sprite do personagem
    call  Print

GameLoop:
    la   a0, personagem
    call  Animacao
    li   a0, 150          # intervalo em ms de cada pose
    call  Sleep
    j    GameLoop

.include "Animacao.s"
.include "GuardaFundo.s"
.include "ImprimeFundo.s"
.include "Print.s"
.include "Sleep.s"
```

Executando o programa no RARS e no FPGRARS, vê-se que a animação é executada corretamente, com exceção do já comentado mau comportamento do RARS com a cor invisível. 

2.6 Analisando o system: o printChar

2.7 Resumo da seção

3 O teclado KDMMIO

3.1 Leitura por keypoll

3.2 Leitura por interrupção

3.3 Resumo da seção

4 Como tocar música no RARS/FPGRARS

4.1 Como tocar uma nota: as ecall's de midi

4.2 Tocando música dentro de um loop de nota em nota

4.3 Analisando o system: midiOut

4.4 Resumo da seção

5 Apêndice

5.1 Cores do RARS

```
#
    ################################################################################

# exibicao das cores do RARS pelo metodo de impressao "azulejada"   #
# agosto de 2022 - 2022/1 - OAC           #
#
    ################################################################################


li   s0, 300        # s0 <- 300 = quantidade de tiles que serao
    impressas
li   s1, 256        # s1 <- 256 = quantidade de cores
li   a0, 0x00       # a0 <- byte de cor da tile (inicialmente 0, que eh
    preto)
mv   a1, x0         # a1 <- posicao em x no bitmap da primeira tile
mv   a2, x0         # a2 <- posicao em y no bitmap da primeira tile
mv   a3, x0         # contador de tiles
PrintTileLoop:
  beq a3, s0, PrintTileEnd  # num. de tiles = total ? fim do programa
      : mais uma tile
  call  PrintTile   # uma tile de cor a0 eh impressa em (a1, a2)
  # ^jal ao inves de call economizaria uma instrucao, vide o Execute
  addi  a0, a0, 1   # uma tile foi impressa => mudamos a cor
  rem a0, a0, s1    # a0 = a0 % 256 para evitar a0 > 255
  # ^comente a linha acima para ver que o RARS a executa por debaixo
      dos panos,
  #  isto eh, as cores passam a se repetir
  addi  a1, a1, 16    # deslocamento em x para a proxima tile
  addi  a3, a3, 1   # incrementacao do contador de tiles
  li  t0, 320       # t0 <= largura do bitmap
  beq a1, t0, ProximaLinhaDeTiles # eh hora de pular linha ? pulamos :
       iteramos o loop
  j PrintTileLoop
ProximaLinhaDeTiles:
  mv  a1, x0        # pos. em x reinicia do 0 (seria possivel usar rem)
  addi  a2, a2, 16    # deslocamento em y para a proxima tile
  j PrintTileLoop   # iteracao do loop
PrintTileEnd:
  fpg:  j fpg       # loop eterno para o fpgrars
  li  a7, 10        # fim do programa por pseudoinstrucao
  ecall
#

# para onde foi a cor 199? ;)

#
    ################################################################################

# esta funcao imprime uma "tile" colorida 16x16 no bitmap   #
#
    ################################################################################

# a0 = byte de cor da tile          #
# a1 = posicao em x no bitmap         #
# a2 = posicao em y no bitmap         #
#
```

```
    ###########################################################################


PrintTile:
  # primeiramente, colorimos uma linha horizontal de 16 bits
  li  t0, 16      # t0 <- tamanho da linha
  mv  t1, x0      # t1 <- contador de bytes na linha
  mv  t2, x0      # t2 <- contador de linhas
  li  t3, 0xFF000000    # t3 recebe o endereco inicial do bitmap no
      frame 0 (canto superior esquerdo)
  # ^ate aqui, estamos no canto superior esquerdo (0,0), mas queremos
      ir a (a1, a2)
  mv  t4, a1      # guardamos a1 em t4 por seguranca
  mv  t5, a2      # guardamos a2 em t5 por seguranca
  add t3, t3, t4    # t3 <- 0xFF000000 + a1
  # ^ate aqui, estamos em (a1, 0)
  li  t6, 320     # auxiliar t6=320 para pularmos as linhas
  mul t5, t5, t6    # t5 = 320*a2 => pula a2 linhas no bitmap
  add t3, t3, t5    # finalmente chegamos ao endereco em (a1, a2) :)
PrintLoop:
  beq t1, t0, PulaLinha # len(linha oclorida) = 16 ? proxima linha :
      mais um byte de cor
  sb  a0, 0(t3)     # colore o byte de endereco t3
  addi  t3, t3, 1   # 1 byte foi impresso => vamos ao endereco do
      proximo byte
  addi  t1, t1, 1   # 1 byte foi impresso => contador incrementa
  j PrintLoop   # iteracao do loop
PulaLinha:
  addi  t2, t2, 1   # nova linha <=> contador de linhas aumenta
  beq t2, t0, PrintEnd  # num. de linhas = 16 ? fim da funcao :
      continua o loop
  mv  t1, x0      # contador reinicia
  addi  t3, t3, 320   # nova linha <=> endereco do bitmap incrementa
      na largura do bitmap
  # ^ate aqui, saimos de um ponto na aresta direita e caimos no ponto
      logo abaixo da mesma aresta
  addi  t3, t3, -16   # subtraindo a largura da tile, caimos na aresta
       esquerda
  j PrintLoop   # iteracao do loop
PrintEnd:
  ret       # retorno da funcao por pseudoinstrucao
#
```