Especificação da Arquitetura

Processador BLA

Efficiency Eletronics S.A.

Build 2.0

History Review

Date	Description	Author(s)
05/12/2015	Definição do propósito do documento e conteúdo das seções (introdução)	Joacy Mesquita
05/12/2015	Definição das características operacionais	Thiago Sampaio Lima
06/12/2015	Definição do conjunto de instruções	Matheus Moura Batista
08/12/2015	Definição dos requisitos não funcionais e flags do processador	Thiago Sampaio Lima
09/12/2015	Definição do formato das instruções	Matheus Moura Batista
09/12/2015	Definição dos acrônimos e abreviações	Todos
10/12/2015	Definição dos componentes do processador e criação do datapath	Joacy Mesquita
14/12/2015	Adição de requisitos não funcionais e descrição do limite de endereços para saltos	Thiago Sampaio Lima
16/12/2015	Adição da descrição do montador e modo de execução do mesmo juntamente com o simulador	Joacy Mesquita
18/12/2015	Revisão geral e correções no documento	Todos

SUMÁRIO

1. Introdução

1.1. Propósito

O propósito deste documento é mostrar uma visão geral sobre o processador BLA, incluindo suas principais características e a descrição dos seus componentes. Este documento também apresenta a arquitetura do conjunto de instruções do processador, incluindo o código da operação e uma breve descrição da mesma.

1.2. Organização do documento

As seções deste documento são organizadas conforme mostrado a seguir.

- Seção 2: Apresenta uma visão geral das principais características do processador, incluindo os principais componentes do mesmo e os requisitos não funcionais.
- Seção 3: Apresenta a arquitetura do conjunto de instruções do processador e suas características.
- Seção 4: Apresenta as características do montador e o como utilizá-lo juntamente com o simulador.

1.3. Acrônimos e Abreviações

Acrônimo	Descrição
RISC	Computador com Conjunto Reduzido de Instruções
MIPS	Microprocessador sem Estágios Interligados de Pipeline
GPR	Registradores de Propósito Geral
ISA	Arquitetura do Conjunto de Instruções
ULA	Unidade Lógica e Aritmética
PC	Contador de Programa
IR	Registrador de Instruções
RA, RB	Registradores Fonte
RC	Registrador Destino
NOP	Instrução Sem Operação

2. Visão Geral

2.1. Principais características

O processador possui um conjunto de 16 registradores de propósito geral e apresenta uma arquitetura de 32 bits. O ISA deste processador foi desenvolvido utilizando-se algumas características das arquiteturas μRISC e MIPS. As principais características operacionais e seus respectivos significados são apresentados a seguir.

2.1.1. Arquitetura com 32 bits de largura

Neste modelo de arquitetura, todos registradores do processador são compostos por 32 bits. Com isso, é possível utilizar valores inteiros sem sinal que variam de 0 a 4294967296 ou com sinal que variam de -2147483648 a +2147483647.

2.1.2. Dezesseis registradores de propósito geral

O processador fornece 16 registradores de propósito geral (R_0 a R_{15}) de 32 bits cada um. O registrador R_15 é utilizado como padrão para armazenar o endereço de retorno nas chamadas de função. Estas chamadas são realizadas por meio da instrução jump and link (jal).

2.1.3. Palavras com tamanho de 4 bytes

O tamanho das palavras de dados são de 4 bytes cada. Esta característica está relacionada principalmente com a largura da arquitetura do processador, permitindo armazenar uma palavra por registrador.

2.1.4. Endereçamento a nível de palavra

O endereçamento utilizado pelo processador para acessar a memória é feito a nível de palavra, ou seja, cada endereço aponta para o início de um conjunto de 4 bytes (32 bits). Dessa forma, ao realizar um incremento no registrador PC, o mesmo pulará 4 bytes e passará a apontar para a proxima palavra da memória.

2.1.5. Dois modos de endereçamento

Para que as instruções possam ser executadas é necessário obter os operandos da mesma. Para tal, é necessário o armazenamento dos valores dos operandos nos registradores do processador. Para armazenar estes valores, pode-se utilizar um dos modos de endereçamento disponíveis: (1) via registrador, (2) imediato e (3) indexado indireto.

O modo de endereçamento via registrador utiliza apenas os registradores do processador para realizar as operações, visto que os operandos estão previamente armazenados neles. O modo imediato diz respeito somente às operações de carga de constantes, visto que as demais operações precisam que estas constantes já estejam em registradores. É válido notar também a possibilidade de utilizar o modo indexado de forma indireta, realizando incremento ou adição em um endereço base disponível e, com isso, obter um novo endereço para o que se deseja acessar na memória.

2.1.6. *Quatro tipos de instrução*

As instruções da ISA deste processador são divididas em 4 principais grupos:

- Instruções de acesso à memória
- Instruções lógico-aritméticas
- Instruções de desvios e transferência de controle
- Instruções de carregamento de constantes

No primeiro grupo estão as instruções utilizadas para ler da memória e gravar em um registrador e escrever na memória apartir de um valor presente em um registrador. No segundo grupo estão as instruções lógicas e as operações aritméticas. No terceiro grupo estão presentes as instruções de transferência de controle com e sem condições. Já no quarto grupo estão presentes as instruções responsáveis pela carga de constantes.

2.1.7. *Instruções com três operandos*

Instruções lógico-aritméticas possuem três operandos, sendo um deles o registrador de destino e os outros dois os registradores fonte.

2.1.8. *Flags para determinados estados do processador*

O processador possui a capacidade de indicar seis flags para determinar o estado decorrente da execução de uma instrução. A tabela a seguir mostra as flags suportadas pelo processador e uma breve descrição das situações em que são ativadas.

Código	Flag	Descrição
0001	.overflow	Operação executada extrapolou o número máximo de bits para representar o resultado. Overflow é alterado em operações com sinal.
0010	.zero	Operação executada apresentou como resultado o valor zero
0011	.neg	Operação executada apresentou como resultado um valor negativo
0100	.negzero	Operação executada apresentou como resultado um valor igual ou menor que zero
0101	.true	Operação executada apresentou como resultado um valor diferente de zero
0110	.carry	Carry é alterado em operações sem sinal.

É válido mencionar também que, caso uma flag overflow ou carry seja acionada, as demais flags (neg, negzero, zero e true) consequentemente serão desativadar. Esta

característica existe devido ao fato de não haver como garantir a integridade do resultado quando a operação geram carry ou overflow. Dessa forma, as demais flags serão definidas como zero lógico.

2.1.9. Dados organizados na forma Big Endian

Ao serem armazenados na memória, os dados são organizados na forma Big Endian, ou seja, os bytes mais significativos ficam nas menores posições de memória.

2.2. Requisitos não funcionais

O projeto do processador engloba características que não dizem respeito diretamente a uma funcionalidade do mesmo e sim à questões de escalabilidade, eficiência ou, até mesmo, plataforma de funcionamento. A seguir são apresentadas todas estas características, conhecidas também como requisitos não funcionais.

2.2.1. Assembler

Para que seja possível utilizar tanto o simulador quanto o processador implementado é necessário a criação de um montador para esta arquitetura. O montador é o responsável por traduzir códigos-fonte em linguagem Assembly para o código de máquina. O Assembler faz parte do pacote de programas necessários para validar o funcionamento lógico do processador.

2.2.2. Simulador

Devido a complexidade do processo de implementação de um processador, é necessário a criação de um simulador em software para validar a especificação e a arquitetura do conjunto de instruções. Essa validação é necessária para que todos os erros graves no projeto sejam descobertos com antecedência e, com isso, não cheguem a prejudicar o processo de implementação. O simulador também faz parte do pacote de programas para validar o funcionamento do processador.

2.2.3. Programas de Teste

A fim de verificar o funcionamento básico do montador e do processador nas situações que poderão ocorrer, é necessário a entrega de três programas em Assembly para de serem traduzidos e executados durante a etapa de testes. Esses programas serão, cada um, um dos algoritmos especificados no problema: (1) Busca binária, (2) Ordenação Bubble Sort e (3) Sequência Fibonacci.

2.2.4. Suporte à 16 Gigabytes de memória

Devido as características de endereçamento a nível de palavra e arquitetura de 32 bits, o processador dá suporte a 16 Gigabytes de memória de dados. A quantidade máxima de memória suportada pelo processador é calculada a partir da quantidade de endereços que o o mesmo suporta (2^{32}) multiplicado pela capacidade de armazenamento de cada endereço de memória (4 bytes).

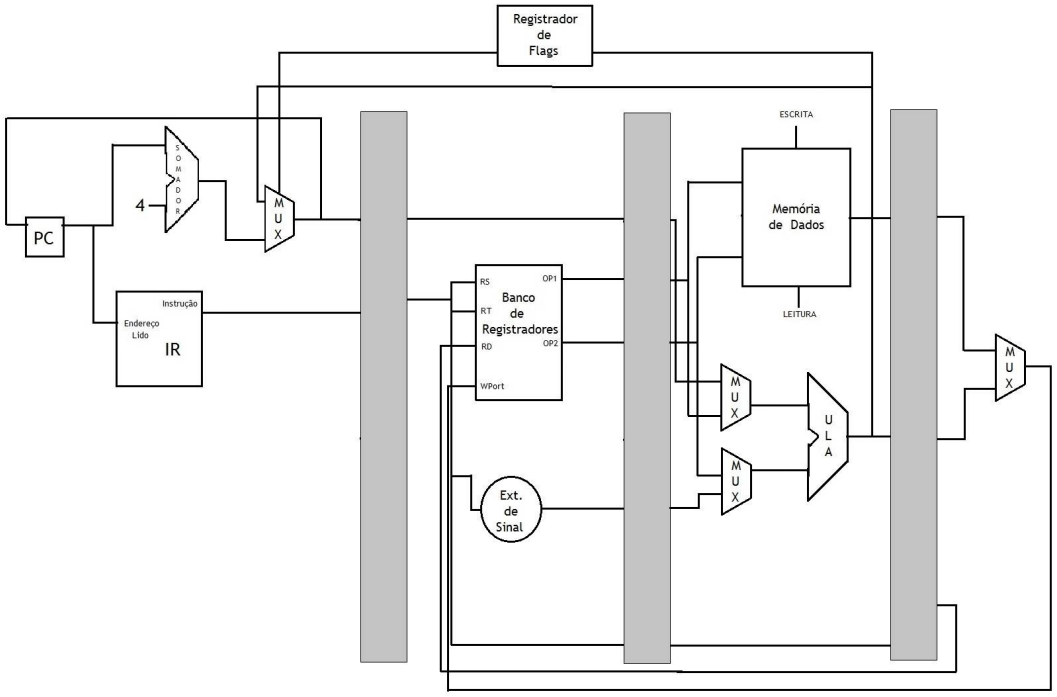

Figura 1: Visão geral dos componentes e datapath do processador

2.2.5. Suporte à ampliação do ISA

O formato e a quantidade de bits disponíveis para determinar o código de cada instrução foram feitos com capacidade acima do necessário, ou seja, conforme surgir a necessidade de mais instruções, será consideravelmente fácil adicionar estas funcionalidades ao processador.

2.3. Componentes internos e datapath

Esta seção tem por finalidade apresentar uma visão geral da arquitetura do processador, abrangendo os compentes principais e o caminho de dados, além do ciclo de execução de uma instrução.

2.3.1. Program Counter (PC)

O *Program Counter* é o registrador que armazena o endereço da próxima instrução a ser lida da memória para que posteriormente possa ser executada.

2.3.2. Instruction Register (IR)

O *Instruction Register* é o registrador que armazena a instrução que está executando no momento. Esta instrução foi lida a partir do endereço que o registrador PC armazenava num momento anterior.

2.3.3. *Somador do PC*

Utilizado para calcular o endereço da próxima instrução. Para tal, o endereço armazenado em PC é adicionado a 4. Levando em consideração a arquitetura de 32 bits do processador e o modo de endereçamento a nível de palavra, o número 4 representa a palavra (4 bytes), fazendo com que o PC seja atualizado com um endereço 4 bytes após o que já possui.

2.3.4. *Banco de Registradores*

O banco de registradores é o conjunto onde os Registradores de Propósito Geral ficam agrupados. O banco possui 16 GPR de 4 bytes cada e é onde são armazenados os operandos e os resultados temporários da execução de uma instrução.

2.3.5. *Unidade de Extensão de Sinal*

] A Unidade de Extensão de Sinal é a responsável por extender operandos com menos de 32 bits. Essa extensão é necessária pois todas as operações deste processador necessitam de valores com 32 bits de largura. Este procedimento é realizado através da replicação do bit mais significativo do operando a todos os bits que vêm a sua esquerda, até o bit 31.

2.3.6. *Unidade Lógico-Aritmética*

A Unidade Lógico-Aritmética é a responsável por realizar todas as operações aritméticas e lógicas.

2.3.7. *Registrador de Flags*

Registrador utilizado para armazenar as flags geradas durante a execução das instruções no processador. Estas flags são utilizadas, principalmente, nas operações de transferência de controle condicionais.

2.3.8. *Memória de Dados*

É a memória sistema, necessária para o armazenamento de dados a longo prazo.

2.3.9. *Ciclo de Execução da Instrução*

O ciclo de execução de uma instrução será dividido em quatro passos.

- 1º: Busca da Instrução (*Fetch Instruction*).
- 2º: Decodificação da Instrução.
- 3º: Busca dos Operandos (*Fetch Operands*).
- 4º: Execução da Instrução.

3. Arquitetura do conjunto de instruções (ISA)

O processador BLA foi construido baseado na arquitetura de processadores µRISC. A camada ISA tem 42 instrucões, cada uma composta por uma palavra de 32-bits, e organizadas nos seguintes grupos:

- Logicas e aritiméticas
- Carregamento de constantes
- Transferência de dados
- Desvio/Transferência de controle

Na arquitetura BLA, as operações são realizadas unicamente através de operandos localizados no banco de registradores. Dessa forma, a memória principal é acessada apenas através das instruções de acesso à memória (load e store). Constantes com e sem sinal são suportadadas pela instrução de carregamento de constante (loadlit), que carrega o número com sinal extendido para o registrador especificado.

3.1. Instruções lógicas e aritméticas

As instruções lógico-aritméticas possuem o seguinte formato:

31 30	29 12	11 8	7 4	3 0
0 0	Código	RC	RA	RB

Código	Mnemonico	Operação	Descrição	Flags Atualizadas
00...000000	add c, a, b	c = a + b	Adiciona dois números.	todas
00...000001	addu c, a, b	c = a + b	Adiciona dois números sem sinal.	todas
00...000010	addinc c, a, b	c = a + b + 1	Adiciona dois números e incrementa 1.	todas
00...000011	inca c, a	c = a + 1	Incrementa 1.	todas
00...000100	sub c, a, b	c = a - b	Subtrai dois números.	todas
00...000101	subdec c, a, b	c = a – b - 1	Subtrai dois números e decrementa 1.	todas

Continua na próxima página

Continuação da página anterior

Código	Mnemonico	Operação	Descrição	Flags Atualizadas
00...000110	deca c, a	c = a - 1	Decrementa 1.	todas
00...000111	subu c, a, b	c = a - b	Subtrai dois números sem sinal	todas
00...001000	addincu c, a, b	c = a + b + 1	Adiciona dois números sem sinal e incrementa 1	todas
00...001001	passb c, b	c = b	Transfere a palavra em 'b' para 'c'.	nenhuma
00...001010	passnotb c, b.	c = !b	Nega o valor de 'b' e transfere a palavra para 'c'.	nenhuma
00...001011	div c, a, b	c = a / b	Divide dois números.	negzero, neg, true
00...001100	divu c, a, b	c = a / b	Divide dois números sem sinal.	true, carry
00...001101	asl c, a	c = a ASL 1	Deslocamento aritmético à esquerda.	carry, zero, true
00...001110	asr c, a	c = a ASR 1	Deslocamento aritmético à direita.	carry, zero, true
00...001111	zeros c	c = 0	Reseta o registrador.	negzero, zero
00...010000	ones c	c = 1	Seta o registrador como o número 1.	nenhuma
00...010001	passa c, a	c = a	Transfere a palavra em 'a' para 'c'.	nenhuma

Continua na próxima página

Continuação da página anterior

Código	Mnemonico	Operação	Descrição	Flags Atualizadas
00...010010	passnota c, a	c = !a	Nega a palavra em 'a' e transfere para 'c'.	nenhuma
00...010011	and c, a, b	c = a & b	AND lógico bit-a-bit.	zero, true
00...010100	andnota c, a, b	c = !a & b	AND lógico bit-a-bit com o 'a' negado.	zero, true
00...010101	nand c, a, b	c = !(a & b)	AND lógico bit-a-bit negado.	zero, true
00...010110	or c, a, b	c = a | b	OR lógico bit-a-bit.	zero, true
00...010111	ornota c, a, b	c = a | !b	OR lógico bit-a-bit com o 'a' negado.	zero, true
00...011000	nor c, a, b	c = !(a | b)	OR lógico bit-a-bit negado.	zero, true
00...011001	xor c, a, b	c = a $\oplus$ b	XOR lógico bit-a-bit.	zero, true
00...011010	xornota c, a, b	c = !a $\oplus$ b	XOR lógico bit-a-bit com o 'a' negado.	zero, true
00...011011	xnor c, a, b	c = !(a $\oplus$ b)	XOR lógico bit-a-bit negado.	zero, true
00...011100	lsl c, a	c = a LSL 1	Deslocamento lógico à esquerda.	carry, zero, true
00...011101	lsr c, a	c = a LSR 1	Deslocamento lógico à direita.	carry, zero, true
00...011110	slt c, a, b	if (a < b): c = 1; else: c = 0	Compara dois números.	negzero, zero, true

Continua na próxima página

Continuação da página anterior

Código	Mnemonico	Operação	Descrição	Flags Atualizadas
00...011111	sltu c, a, b	if (a < b): c = 1; else: c = 0	Compara dois números sem sinal.	negzero, zero, true

Por haver 32 instruções lógicas e artméticas, estas precisariam de um código de apenas 5-bits. Apesar disso, nesta arquitetura as instruções lógicas e aritméticas tem um código de 6-bits de largura, porque dessa forma mais 32 instruções são suportadas, tornando a arquitetura muito mais escalável.

Uma das características que diferem as instruções lógicas e aritméticas das outras instruções é a influência dos seus resultados na alteração do valor das *flags*. Todas as instruções lógicas zeram as *flags overflow* e *carry*.

3.2. Instruções de carga de constantes

As instruções de carga de constantes tem o seguinte formato:

31 30	29 20	19 16	15 0
0 1	Código	RC	Constante

Código	Mnemonico	Operação	Descrição
00...000	loadlit c, const	c = const	Carrega a constante com sinal extendido em 'c'.
00...001	lcl c, const	c = const | (c & 0xffff0000)	Carrega a constante nos dois bytes menos significativos de 'c'.
00...010	lch c, const	c = (const « 8) | (c & 0x0000ffff)	Carrega a constante nos dois bytes mais significativos de 'c'.

Na instrução 'loadlit' antes da constante ser carregada, o seu bit de sinal é extendido para os bits mais significativos para completar uma palavra de 32-bits. Por examplo: se a constante é 1001011101100101 ela se tornará 11111111111111111001011101100101. As constantes na arquitetura BLA têm 2 bytes de largura.

3.3. Instruções de acesso à memória

As instruções de acesso à memória seguem o seguinte formato:

31 30	29 8	7 4	3 0
1 0	Código	RC	RA

Código	Mnemônico	Operação	Descrição
00...00	load c, a	c = Mem[a]	Lê uma palávra da memória.
00...01	store c, a	Mem[c] = a	Escreve uma palávra na memória.

3.4. Instruções de transferência de controle

As instruções de transferência de controle seguem o seguinte formato:

31 30	29 20	19 16	15 12	11 0
1 1	Código	Condição	RA	Destino

O campo 'condição' deve ser preenchido com uma das flags mostradas na seção 2.1.8. Quando as áreas 'condição', 'RA' e 'destino' não forem utilizadas, estas devem ser preenchidas com zero lógico.

Código	Mnemonico	Operação	Descrição
00...0000	j Destino	salto incondicional	Salto incondicional para o endereço especificado.
00...0001	jt.Condição Destino	if (condition) jump	Salta apenas se a condição for verdadeira.
00...0010	jf.Condição Destino	if (!condition) jump	Salta apenas se a condição for falsa.
00...0011	jal a	jump and link	Salto incondicional para o endereço em 'a' e salva o endereço da próxima instrução em 'r7'.
00...0100	jr a	jump register	Salto incondicional para o endereço em 'a'.

Nas operações de salto j, jt e jf o endereço do destino é limitado à 12 bits, fazendo com que só seja possível realizar saltos até um certo limite de endereços. Para solucionar este problema basta utilizar a instrução jr (jump register), onde o endereço para o qual se deseja saltar estará pré armazenado em um registrador de 32 bits e, com isso, será possível saltar para qualquer endereço de memória que se deseje.

3.5. Instrução sem operação

A instrução sem operação (`nop`) pode ser utilizada para controlar o fluxo das instruções ou para inserir atrasos (delays) no programa.

3.6. Fim das operações

A instrução de fim de operação (HALT) é implementada a partir de um salto incondicional para o endereço atual, Por exemplo: "`L: j L`".

4. Montador e Simulador

Esta seção visa fornecer as informações necessárias para escrever programas em linguagem de montagem para o processador BLA. Os arquivos executáveis do montador e do simulador serão fornecidos para que seja possível converter o programa *Assembly* para linguagem de máquina e posteriormente executá-lo. Para realização dos testes, os seguintes aplicativos serão utilizados:

- **MONTADOR**: Também conhecido como Assembler, é o responsável por traduzir o arquivo fonte em *Assembly* para um arquivo de saída codificado em binário específico para o processador.
- **SIMULATOR**: Utilizado para simular o funcionamento do processador. A entrada do simulador é o mesmo conjunto de dados que saem do assembler. Sua saida é representada na forma de um arquivo de texto, onde é exibido todo o conteúdo dos registradores e da memória de dados.

A tradução das instruções do arquivo *Assembly* pelo montador consiste na conversão dos mnemônicos, registradores, condições, constantes e destinos em seus equivalentes binários formando uma palavra de 32 bits. O montador e o simulador devem ser compilados e executados utilizando os seguintes comandos, **em plataformas UNIX e Windows**:

Para o montador:

```
$ gcc -o montador montador.c
$ ./montador teste.asm
```

O primeiro comando compila o código do montador utilizando o compilador gcc. O segundo comando executa o montador usando como arquivo de entrada o arquivo *teste.asm*, obtendo como saída um arquivo de nome *result.bin*. Observe que arquivo de entrada para o montador deve ter uma extensão *.asm*.

Para o simulador:

```
$ gcc -o simulator simulator.c
$ ./simulator result.bin
```

O primeiro comando compila o código do simulador e o segundo executa o simulador utilizando como arquivo de entrada o arquivo *result.bin* obtido no montador. O simulador gera como saída o arquivo *result.txt* que irá conter os últimos valores armazenados na memória de dados do processador. O arquivo de saída do montador é representado em binário e é uma tradução feita a partir do arquivo Assembly de entrada. O formato do arquivo de entrada do montador é descrito a seguir.

4.1. Formato dos arquivos de entrada do montador

O arquivo Assembly de instruções para o processador BLA possui um formato simples e deve conter as seguintes características:

- O arquivo fonte deve conter apenas uma instrução por linha;

- O arquivo pode conter linhas em branco ou linhas apenas com comentários;
- Cada instrução pode ser seguida de um trecho de comentário;
- Labels podem ou não estar na mesma linha da instrução;
- As instruções devem, obrigatoriamente, estar após uma diretiva *.pseg*;
- Os dados devem, obrigatoriamente, estar após uma diretiva *.dseg*;
- Tanto as instruções quanto os dados devem, obrigatoriamente, estar entre uma diretiva *.module* e uma diretiva *.end*;
- Cada um dos dados fornecidos devem ser precedidos de uma diretiva *.word*;
- As instruções devem estar codificadas em assembly, de acordo com o nível ISA do BLA.

A seguir é mostrado um código fonte em Assembly totalmente compatível com o montador em questão.

```
.module fibonacci
.pseg
        ;Algoritmo da Sequencia Fibonacci iterativo
MAIN:
        lcl r0, LOWBYTE ARR1
        lch r0, HIGHBYTE ARR1
        load r0, r0 ;carrega numero n para realizar
                    ;o fibonacci
        loadlit r14, 1 ;f(1) = 1
        loadlit r15, 1 ;f(2) = 1
        loadlit r1, 2 ;i = 2
LOOP:
        sub r2, r1, r0 ;verifica se i < n
        jf.neg FIM ;se i for maior ou igual a n,
        add r3, r14, r15 ;termina se i for menor que n,
                         ;adiciona os dois numeros
                         ;correspondentes ao fibonacci
                         ;de i-1 e i-2 e coloca em r3
        passa r14, r15 ;f(i-2) = f(i-1)
        passa r15, r3 ;f(i-1) = soma em r3
        inca r1, r1 ;i++
        j LOOP
FIM:
        j FIM ;ao final do processamento o resultado
              ;estara armazenado no registrador r15
.dseg
ARR1:
        .word 4
```

```
.end
```