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 em questão possui um conjunto de 16 registradores de propósito geral (organizados em um banco de registradores) e apresenta uma arquitetura com 32 bits de largura. A ISA deste processador foi desenvolvida utilizando-se algumas características da arquitetura µRISC e outras da arquitetura 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 possuem uma largura de 32 bits (4 bytes). 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 (4 bytes). O registrador R_7 é utilizado como padrão para armazenar o endereço de retorno nas chamadas de função. Estas chamadas são realizadas através 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

Cada endereço utilizado pelo processador para acessar a memória é feito a nível de palavra, ou seja, cada endereço aponta para um conjunto de 4 bytes. Ao realizar um incremento no registrador PC, o mesmo apontará para uma nova palavra na memória (pulará 4 bytes ao no lugar de apenas 1).

2.1.5. Operações aritméticas inteiras

Para a realização de cálculos aritméticos, os operandos fornecidos através das instruções terão que estar apresentados na forma inteira, ou seja, o processador em questão não fornece suporte ao cálculo envolvendo ponto flutuante.

2.1.6. Dois modos de endereçamento

Para que as instruções possam ser executadas é necessário obter os operandos da mesma e, para isso, é utilizada algum dos dois modos de endereçamento possíveis: (1) via registrador, (2) imediato. O modo de endereçamento via registrador é o que utiliza apenas os registradores do processador para realizar as operações, visto que os operandos estarão previamente armazenados neles. O modo imediato deste processador diz respeito a somente as operações de carregamento de constantes, visto que as demais operações precisam que estas constantes já estejam em registradores. É válido notar também que é possível 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.

2.1.7. *Quatro tipos de instrução*

As instruções da ISA deste processador são divididas em 4 principais grupos: (1) instruções de acesso à memória, (2) instruções logico-aritméticas, (3) instruções de desvios e saltos e (4) instruções de carregamento de constantes. No primeiro grupo estão as instruções utilizadas para ler (load) da memória e gravar em um registrador e escrever (store) na memória apartir de um valor presente em um registrador. No segundo grupo estão as principais instruções de cálculos aritméticos (add, sub, etc) e lógicas (and, or, etc). No terceiro grupo estão presentes as instruções de salto (j) com e sem condições. Já no quarto grupo estão presentes as instruções responsáveis por gravar as constantes em um registrador para posterior uso (lcl, lch, etc).

2.1.8. *Instruções com três operandos*

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

2.1.9. *Flags para determinados estados do processador*

O processador possui a capacidade de indicar diversas flags para determinar algum estado decorrente de uma execução. 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 e aconteceu um "vai-um". Overflow é alerado 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	Operação realizou um "vai-um". Carry é alterado em operações sem sinal.

É válido mencionar também que caso uma flag overflow ou carry seja indicada, as demais flags (neg, negzero, zero e true) não serão verificadas. Esta característica existe devido ao fato de não haver como garantir a integridade do resultado quando a operação gera carry ou overflow. Neste contexto, as demais flags serão definidas como zero lógico.

2.1.10. *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 em questão engloba algumas características que não dizem respeito diretamente a uma funcionalidade do mesmo e sim a 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 o processador em diversas atividades e operações 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 correto que o processador necessita. O montado faz parte do pacote de programas necessários para validar o funcionamento lógico do processador (confiabilidade).

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 (confiabilidade).

2.2.3. *Programas de Teste*

A fim de atestar o funcionamento do montador e do processador nas diversas 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. Estes programas de teste também fazem parte do pacote de programas necessários para validar o funcionamento do processador (confiabilidade).

2.2.4. *Suporte a grande quantidade memória*

Devido as características de endereçamento a nível de palavra e arquitetura com 32 bits de largura, o processador em questão dá suporte a grande quantidade de memória de dados, possibilitanto a ampla utilização do sistema para diversas atividades (escalabilidade). A quantidade máxima de memória suportada pelo processador é calculada através da quantidade de endereços que o o mesmo suporta (2^{32}) multiplicado pela quantidade de memória armazenada em cada endereço (4 bytes). Com isso, chegamos ao valor de 16 Gigabytes de memória.

2.2.5. *Suporte à ampliação do ISA*

Como pode ser visto na seção que trata do ISA do processador, cada instrução possue um código. Esse código foi feito em um conjunto de bits com capacidade acima do necessário, ou seja, conforme surgir a necessidade de mais instruções, será consideravelmente fácil adicionar estas funcionalidades ao processador (escalabilidade).

2.2.6. *Plataforma Linux*

Tanto o assembler (montador) quanto o simulador desenvolvidos devem ser capazes de executar corretamente na plataforma Linux (plataforma).

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 dos dados na execução de uma instrução, além do ciclo de execução de uma instrução.

2.3.1. *Program Counter (PC)*

O *Program Counter*, ou PC é o registrador que armazena o endereço da próxima instrução a ser buscada na memória.

2.3.2. *Instruction Register (IR)*

O *Instruction Register*, ou IR é o registrador que armazena a instrução buscada na memória.

2.3.3. *Somador*

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. Neste caso, o banco possui 64 bytes de capacidade (16 GPR de 4 bytes cada). É neste banco que são armazenadas as variáveis temporárias de cada execução de instrução.

2.3.5. *Unidade de Extensão de Sinal*

] O extensor de sinal é o 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. Essa extensão é feita replicando o bit mais significativo do operando a todos os bits que vêm a sua esquerda, até o bit de número 31.

2.3.6. *Unidade Lógico-Aritmética*

A Unidade Lógico-Aritmética é a responsável por realizar todas as operações aritméticas (soma, subtração, etc) e lógicas (and, or, etc).

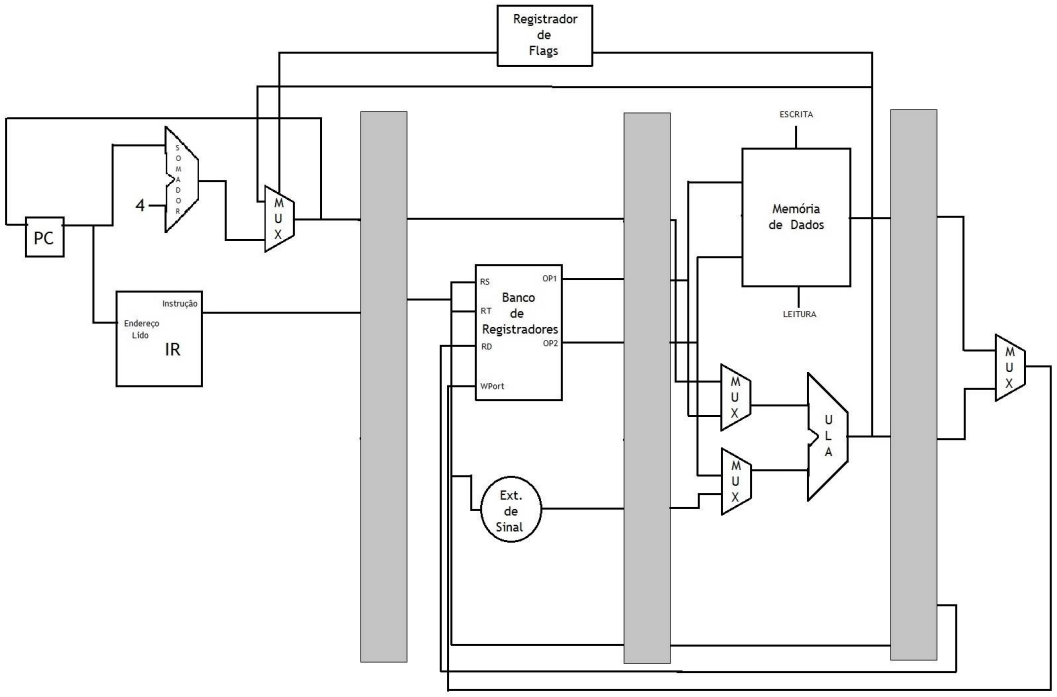

Figura 1: Visão geral dos componentes e datapath do processador

2.3.7. Registrador de Flags

Registrador utilizado para armazenar todas as flags geradas durante as operações do processador. Estas flags são utilizadas, principalmente, nas operações de jump condicionais.

2.3.8. Memória de Dados

É a memória sistema, necessária para o armazenamento de dados a longo prazo. Acessada através das intruções de acesso à memória, load e store.

2.3.9. Ciclo de Execução da Instrução

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

- 1º: Busca da Instrução (*Fetch Instruction*).
- 2º: Decodificação da Instrução.
- 3º: Busca dos Operandos (*Fetch Operands*).
- 4º: Execução da Instrução.
- 5º: Armazenamento dos Resultados.

3. Arquitetura do conjunto de instruções (ISA)

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
- Salto

Na arquitetura BLA, as operações são realizadas unicamente através de operandos localizados em algum registrador do 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 de 16-bits são suportadadas pela instrução de carregamento de constante (loadlit), que carrega o número com sinal extendido para o registrador especificado.

3.1. Arithmetic and logical instructions

As instruções lógico-aritméticas tem o seguinte formato:

31 30	29 18	17 12	11 8	7 4	3 0
0 0	00...0	Código	RC	RA	RB

Código	Mnemonico	Operação	Descrição
000000	add c, a, b	c = a + b	Adiciona dois números.
000001	addu c, a, b	c = a + b	Adiciona dois números sem sinal.
000010	addinc c, a, b	c = a + b + 1	Adiciona dois números e incrementa 1.
000011	inca c, a	c = a + 1	Incrementa 1.
000100	sub c, a, b	c = a - b	Subtrai dois números.
000101	subdec c, a, b	c = a – b - 1	Subtrai dois números e decrementa 1.
000110	deca c, a	c = a - 1	Decrementa 1.
000111	subu c, a, b	c = a - b	Subtrai dois números sem sinal
001000	addincu c, a, b	c = a + b + 1	Adiciona dois números sem sinal e incrementa 1

Continua na próxima página

Continuação da página anterior

Código	Mnemonico	Operação	Descrição
001001	passb c, b	c = b	Transfere a palavra em 'b' para 'c'. Não acende flags.
001010	passnotb c, b.	c = !b	Nega o valor de 'b' e transfere a palavra para 'c'.
001011	div c, a, b	c = a / b	Divide dois números.
001100	divu c, a, b	c = a / b	Divide dois números sem sinal.
001101	asl c, a	c = a ASL 1	Deslocamento aritmético à esquerda.
001110	asr c, a	c = a ASR 1	Deslocamento aritmético à direita.
001111	zeros c	c = 0	Reseta o registrador.
010000	ones c	c = 1	Seta o registrador como o número 1.
010001	passa c, a	c = a	Transfere a palavra em 'a' para 'c'.
010010	passnota c, a	c = !a	Nega a palavra em 'a' e transfere para 'c'.
010011	and c, a, b	c = a & b	AND lógico bit-a-bit.
010100	andnota c, a, b	c = !a & b	AND lógico bit-a-bit com o 'a' negado.
010101	nand c, a, b	c = !(a & b)	AND lógico bit-a-bit ngegado.
010110	or c, a, b	c = a | b	OR lógico bit-a-bit.
010111	ornota c, a, b	c = a | !b	OR lógico bit-a-bit com o 'a' negado.
011000	nor c, a, b	c = !(a | b)	OR lógico bit-a-bit negado.
011001	xor c, a, b	c = a $\oplus$ b	XOR lógico bit-a-bit.
011010	xornota c, a, b	c = !a $\oplus$ b	XOR lógico bit-a-bit com o 'a' negado.
011011	xnor c, a, b	c = !(a $\oplus$ b)	XOR lógico bit-a-bit negado.
011100	lsl c, a	c = a LSL 1	Deslocamento lógico à esquerda.
011101	lsr c, a	c = a LSR 1	Deslocamento lógico à direita.
011110	slt c, a, b	if (a < b): c = 1; else: c = 0	Compara dois números.

Continua na próxima página

Continuação da página anterior

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

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. Todas as instruções lógicas resetam os *flags overflow* e *carry*.

3.2. Instruções de carregamento de constante

The constant load instructions have the following format:

31 30	29 22	21 20	19 16	15 0
0 1	00...0	Código	RC	Constante

Código	Mnemonico	Operação	Descrição
00	loadlit c, const	c = const	Carrega a constante com sinal extendido em 'c'.
01	lcl c, const	c = const | (c & 0xffff0000)	Carrega a constante nos dois bytes menos significativos de 'c'.
10	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 tem 2 bytes de largura.

3.3. Instruções de transferência de dados

As instruções de transferência de dados seguem o seguinte formato:

31 30	29 10	9 8	7 4	3 0
1 0	00...0	Código	RC	RA

Code	Mnemonic	Operation	Description
0	load c, a	c = Mem[a]	Lê uma palávra da memória.
1	store c, a	Mem[c] = a	Escreve uma palávra na memória.

3.4. Instruções de salto

As instruções de salto seguem o seguinte formato:

31 30	29 24	23 20	19 16	15 12	11 0
1 1	00...0	Código	Condição	RA	Destino

Quando as áreas ‘condição’, ‘RA’ and ‘destino’ não forem utilizadas, estas devem ser preenchidas com zero lógico.

Código	Mnemonico	Operação	Descrição
0000	j Destino	salto incondicional	Salto incondicional para o endereço especificado.
0001	jt.Condição Destino	if (condition) jump	Salta apenas se a condição for verdadeira.
0010	jf.Condição Destino	if (!condition) jump	Salta apenas se a condição for falsa.
0011	jal a	jump and link	Salto incondicional para o endereço em ’a’ e salva o endereço da próxima instrução em ’r7’.
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 a 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 das operações (HALT) deve ser implementada através de um salto incondicional para o endereço atual, Por exemplo: "`L: j L`".

4. Montador

Esta seção visa fornecer as informações necessárias para escrever programas em linguagem de montagem para o processador BLA. O arquivo executável do montador será fornecido para converter o programa em linguagem de montagem para linguagem de máquina. 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 em questão.
- **SIMULATOR**: Utilizado para simular o funcionamento real do processador. A entrada do simulador é o mesmo conjunto de dados que saem do assembler. Sua saida é feita através de um arquivo de texto, onde é mostrado todo o conteúdo dos registradores e da memória de dados.

O montador e o simulador devem ser compilados e executados utilizando os seguintes comandos, **em plataformas UNIX**:

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 nos registradores e na memória de dados do processador. O formato do arquivo de entrada do montador é descrito a seguir.

4.1. Formato dos arquivos de entrada do montaodr

O arquivo assembly de instruções para o processador BLA possui um formato simples e deve possuir 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. Este código fonte realiza o algoritmo da sequência Fibonacci para o número 4.

```
.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
```

4.2. Funcionamento do montador

Como dito na subseção anterior, o montador recebe como entrada um arquivo de extensão .asm, ao receber este arquivo ele executa algums passos até gerar o arquivo binário.

Primeiramente, temos que destacar três variáveis de controle (booleanas), **module**, **pseg**, e **dseg**, que respectivamente indicam o início do programa, o início do bloco de instruções, e o início do bloco de dados.

O arquivo é lido no intuito de encontrar e traduzir as intruções para os 32 bits da palavra. Para isso é necessário encontrar as diretivas e salvá-las em um array, também é preciso retirar os labels, salvando-os em uma tabela hash como chave para a linha onde se encontram, além de descartar os comentários e quebras de linha.

Feito isso, a análise do conteúdo do array de diretivas acontece. Se a diretiva for *module* indica que o programa começou, quando a diretiva for *pseg*, o programa começará a leitura e tradução das instruções.

A função de traduzir é composta por 42 comparações relativas às 42 instruções do conjunto, nessa função os mnemônicos e os registradores são convertidos aos seus equivalentes binários. A tradução é feita linha a linha até o fim do arquivo. Quando executada a tradução de uma linha, a palavra é escrita no arquivo de saída.