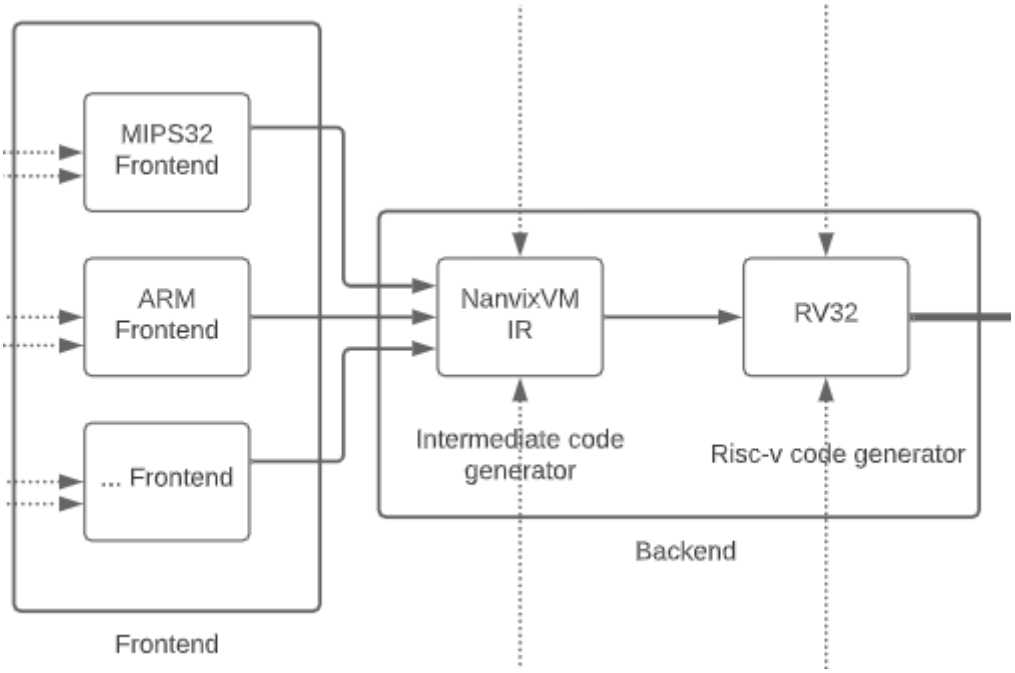
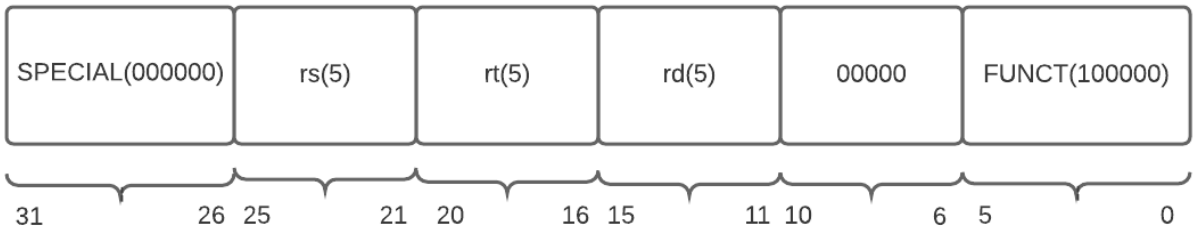
A ideia de uma linguagem intermediária visa simplificar e centralizar o processo de tradução de todos os conjuntos de instrução do front-end para o conjunto de instrução da arquitetura RISC-V. Para isso o objetivo é criar uma linguagem intermediária ou uma expressão intermediária que seja capaz de ser como seu nome diz um intermédio entre o front-end e o back-end.



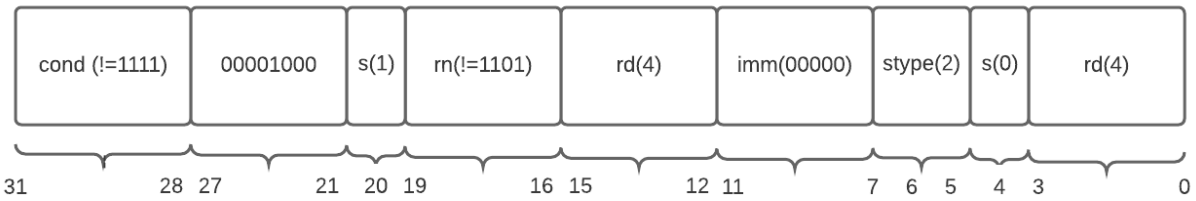
Para isso a linguagem intermediária deve apresentar elementos que estão presentes em instruções do front-end e elementos que estão presentes no back-end.

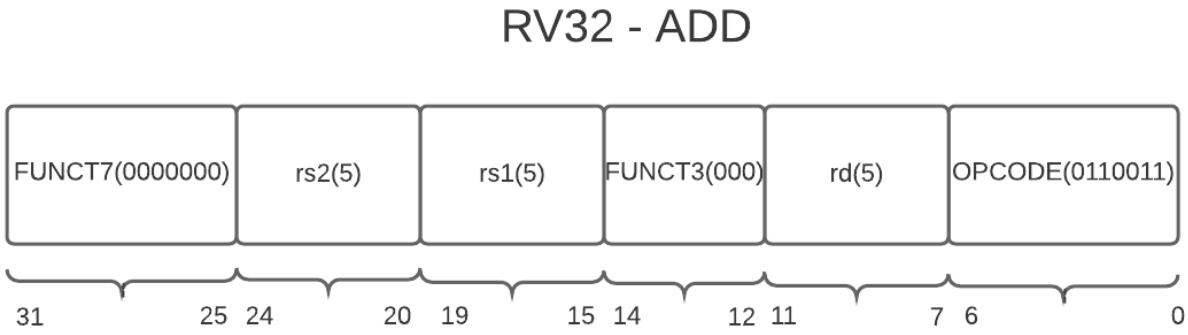
O primeiro passo para esta descoberta de padrões foi colocar os formatos das instruções “lado a lado”. Para o exemplo vamos utilizar a instrução **ADD**.

MIPS32 - ADD



ARM32 - ADD

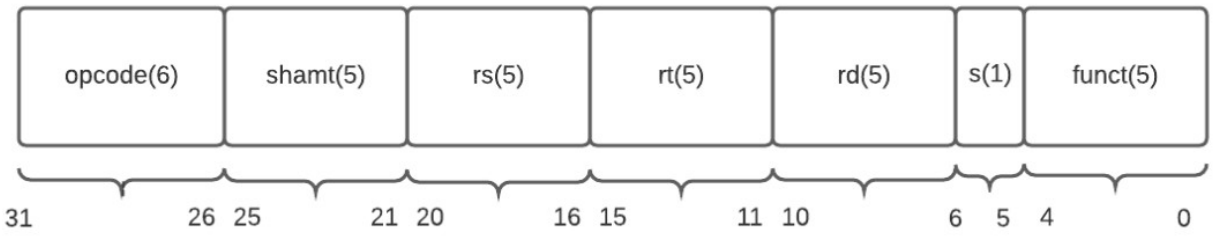




Verificando cada instrução concluí-se que a instrução ADD do MIPS é bem parecida com a instrução ADD do RISC-V. Já a instrução ADD do ARM possui algumas coisas semelhantes e outras diferentes! Com isso, a ideia foi desenvolver uma linguagem intermediária parecida com o MIPS e o RISC-V com a extração de apenas 1 campo utilizado na arquitetura ARM.

A montagem da linguagem intermediária é igual a do MIPS32, porém com duas diferenças:

1. Os campos sofreram uma permutação para facilitar a extração da Gramática Livre de Contexto à partir de uma árvore.
2. O campo S (Signal) foi adicionado para identificar qual instrução é signal e qual é unsignal.



Tendo a linguagem formada, foi montada uma árvore binária de modo que o algorítmo de busca em largura seja capaz de resgatar e montar a linguagem. Vendo a linguagem como um vetor, e sendo **n=7** que é o número de itens que a linguagem possui inserimos na raiz da árvore o elemento que está na posição (n-1)/2 = 3 ou seja na metade do "vetor". O elemento então que ficará na raiz é o **rt(5)**.

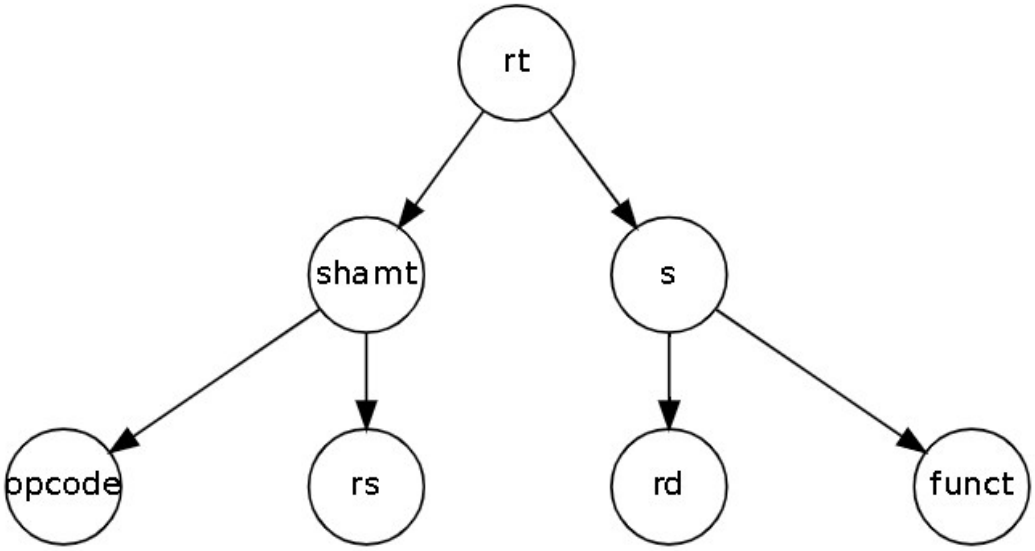
Para inserir o filho á esquerda da raiz é aplicado a mesma lógica, porém com apenas a primeira metade do "vetor". A lógica então foi particionar o vetor em dois sem a posição da metade do vetor original. A primeira partição é um novo vetor **v1** que vai de 0 até 2, e um novo vetor **v2** que vai de 4 até 6. Então é feito o cálculo com n=3: ((n-1)/2)/2 = 1. O filho á esquerda da raiz é **shamt(5)** que está no vetor **v1.**

Para calcular o filho á direita, o mesmo cálculo é feito para um vetor **v2** para n=3: ((n-1)/2)/ 2 = 1. O filho a direita da raiz é **s(1)** que está no vetor **v2.**

Seguindo este raciocínio, o filho á esquerda do **shamt(5)** é o **opcode(6)** e o filho á direita é **rs(5)**. Ambos são folhas.

O filho á esquerda do **s(1)** é o **rd(5)** e o filho á direita é **funct(5).** Ambos são folhas.

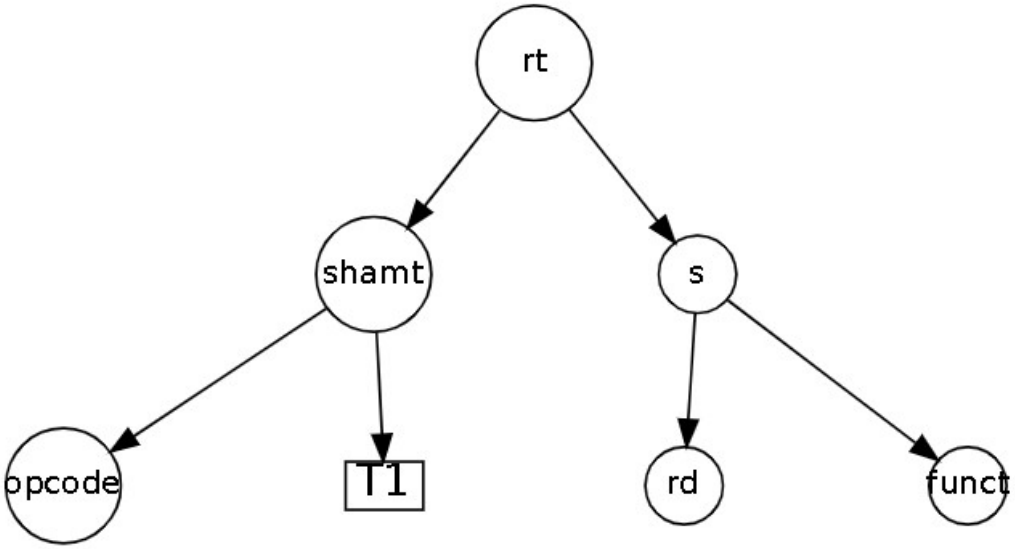
Com isso a árvore está montada.



Agora a ideia é criar as regras da GLC á partir das folhas. Começando então da folha **rs:**

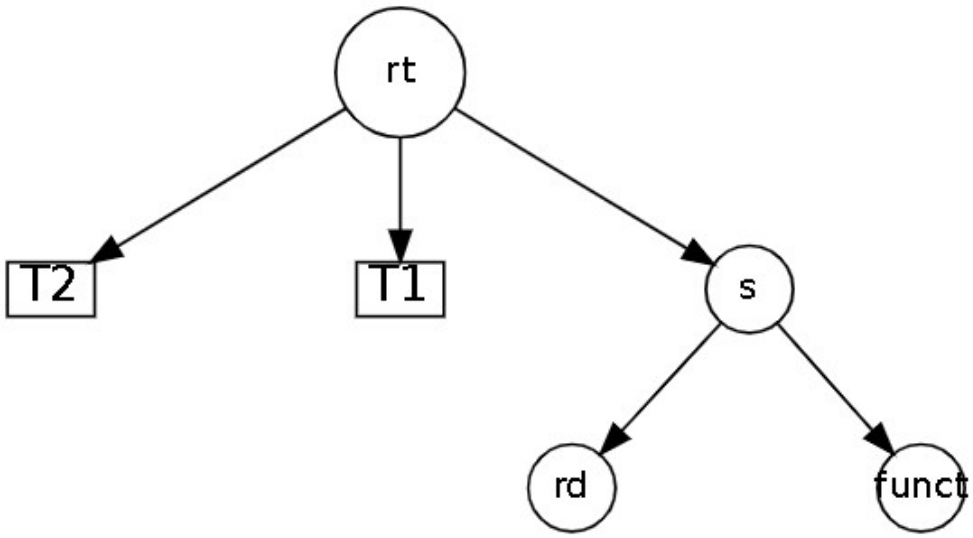
G:

T1 = rs



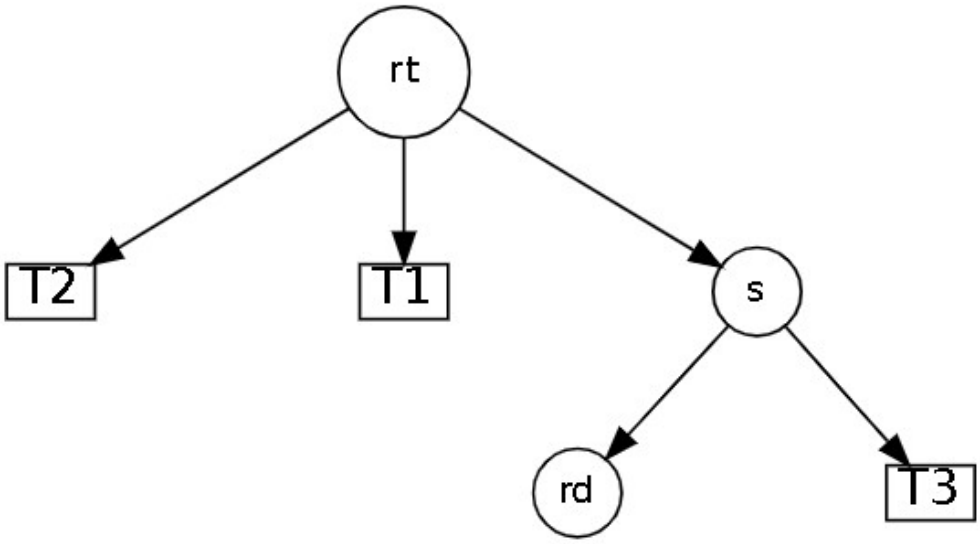
G:

T1 = rs
T2 = opcode shamt



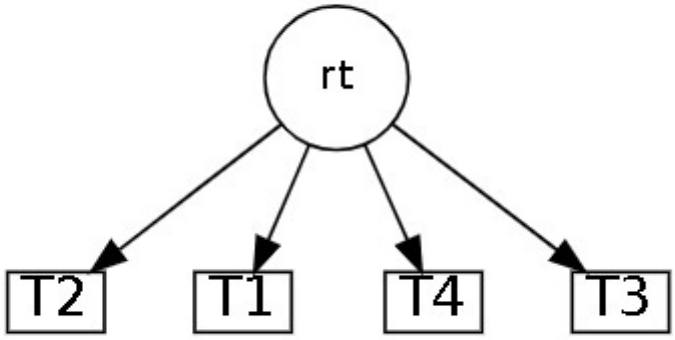
G:

T1 = rs
T2 = opcode shamt
T3 = funct



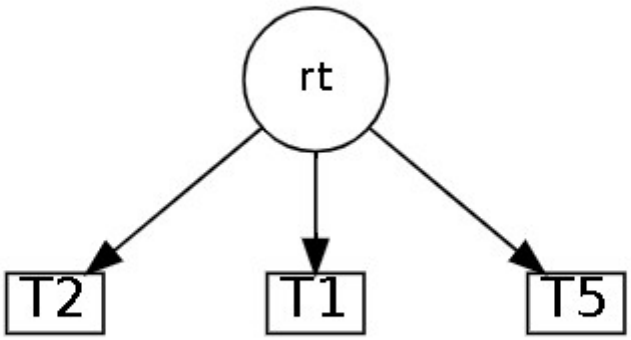
G:

T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s



G:

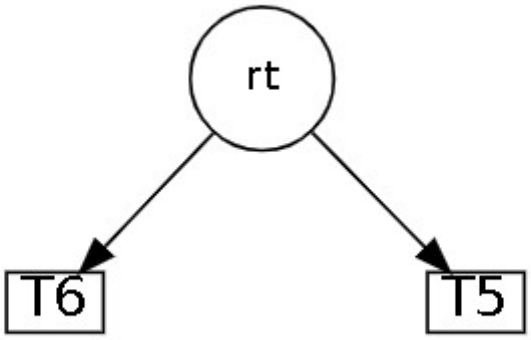
T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = **T4 T3**



G:

T1 = rs

T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = T4 T3
T6 = T2 T1



G:

T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = T4 T3
T6 = T2 T1
T7 = rt **T5**

G:

T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = T4 T3
T6 = T2 T1

T7 = rt **T5**
T8 = T6 T7

Ao final a regra **T8** é renomeada para **S**, pois ela que é a regra inicial para a execução da árvore de parsing.

G:

S = T6 T7
T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = T4 T3
T6 = T2 T1
T7 = rt **T5**

No final, é realizado a ótimização da gramática, que consiste em ir substituindo as regras do tipo **X0** = **X1 X2** para **X0** = **X1**(valor) **X2**(valor) até que o FIRST(**X1**(valor)) e FISRT(**X2**(valor)) seja um **token.**

G:

S = (T2 T1) (rt **T5)**
T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = T4 T3
T6 = T2 T1
T7 = rt **T5**

Na regra **S** foi substituito o **T6** pelo seu valor (**T2 T1**) e **T7** também pelo seu valor (rt **T5**).

Podemos observar que o FISRT(**T7**(rd **T5**)) = rd é um token porém o FISRT(**T6**(**T2 T1**)) = **T2** não é um token. Por isso é realizado novamente o processo para os valores de **T6** que são **T1** e **T2** até que o FIRST(**T1**) e FIRST(**T2**) seja um token.

Apaga-se então a regra **T6** que não é usada em mais nenhum outro lugar da gramática e executamos de novo a lógica.

G:

S = **(**opcode shamt**) (**rs**) (**rt **T5)**
T1 = rs
T2 = opcode shamt
T3 = funct
T4 = rd s
T5 = **T4 T3**
T7 = rt **T5**

Ao final deste passo nota-se que o FIRST(**T1**) e FIRST(**T2**) são tokens. Então a otimização da regra **S** é finalizada excluindo as regras T1 e T2 que não estão sendo mais utilizadas.

G:
S = **(**opcode shamt**) (**rs**) (**rt **T5)**
T3 = funct
T4 = rd s
T5 = **T4 T3**
T7 = rt **T5**

Inicia-se a a otimização da regra **T5** utilizando a mesma lógica.

G:
S = **(**opcode shamt**) (**rs**) (**rt **T5)**
T3 = funct
T4 = rd s
T5 = **(**rd s**) (**funct**)**
T7 = rt **T5**

Após realizado o passo inicial, verifica-se que o FIRST(**T4**) e FIRST(**T3**) são tokens. Então a otimização da regra **T5** é finalizada excluindo as regras **T4** e **T3** que não estão sendo mais utilizadas.

G:

S = **(**opcode shamt**) (**rs**) (**rt **T5)**
T5 = **(**rd s**) (**funct**)**
T7 = rt **T5**

Como não existe mais regras do formato **X0** = **X1 X2** a otimização da gramática é encerrada.

Ao final esta grámatica será implementada para executar o parse e os esquemas de tradução dos conjuntos de instrução do front-end para o Risc-V.

G:

S = **(**opcode shamt**) (**rs**) (**rt **T5)**
T5 = **(**rd s**) (**funct**)**
T7 = rt **T5**