ILOC

Visão geral do capítulo

ILOC é o código assembly para uma máquina abstrata simples. Foi projetado originalmente como uma IR de baixo nível, linear, para uso em um compilador otimizador. Nós o utilizamos no decorrer do livro como uma IR de exemplo, e também como uma linguagem-alvo simplificada nos capítulos que discutem a geração de código. Este apêndice serve como uma referência ao ILOC.

Palavras-chave: Representação intermediária, Código de três endereços, ILOC

A.1 Introdução

ILOC é o código assembly linear para uma máquina RISC abstrata simples. O ILOC usado neste livro é uma versão simplificada da representação intermediária que foi usada no Massively Scalar Compiler Project da Rice University. Por exemplo, o ILOC, conforme apresentado aqui, considera um tipo de dado genérico, um inteiro sem um tamanho específico; no compilador, a IR admitia uma grande variedade de tipos de dados.

A máquina abstrata ILOC tem número ilimitado de registradores. Ela tem operações de três endereços, de registrador-para-registrador, operações load e store, comparações e desvios. Ela admite apenas alguns modos de endereçamento simples – direto, endereço + deslocamento (*offset*), endereço + imediato e imediato. Os operandos de origem são lidos no início do ciclo, quando a operação é emitida. Os operandos de resultado são definidos ao final do ciclo em que a operação termina.

Além do seu conjunto de instruções, os detalhes da máquina não são especificados. A maior parte dos exemplos considera uma máquina simples, com única unidade funcional que executa operações ILOC em sua ordem de aparecimento. Quando outros modelos são usados, discutimos sobre eles explicitamente.

Um programa ILOC consiste em uma lista sequencial de instruções. Cada instrução pode ser precedida por um rótulo (*label*). Um rótulo é apenas uma string de texto; ela é separada da instrução por um sinal de dois pontos. Por convenção, limitamo-nos a rótulos no formato [*a*–*z*] ([*a*–*z*] | [0–9] | –)*. Se alguma instrução precisar de mais de um label, inserimos uma instrução que só contém um `nop` antes dela, e colocamos o label extra no `nop`. Para definir um programa ILOC mais formalmente,

ProgramaIloc → *ListaInstruções*
ListaInstruções → *Instrução*
| `label` : *Instrução*
| *Instrução ListaInstruções*

Cada instrução contém uma ou mais operações. Uma instrução de operação única é escrita em uma linha isolada, enquanto uma instrução com múltiplas operações pode se espalhar por várias linhas. Para agrupar operações em uma única instrução, elas são delimitadas por colchetes e separadas por ponto e vírgula. Mais formalmente,

```
Instrução          → Operação
                   |  [ ListaOperações ]
ListaOperações     → Operação
                   |  Operação ; ListaOperações
```

Uma operação ILOC corresponde a uma instrução em nível de máquina que poderia ser emitida para uma única unidade funcional em um único ciclo. Ela tem um código de operação (*opcode*), uma sequência de operandos de origem separados por vírgulas e uma sequência de operandos de destino também separados por vírgulas. As origens são separadas dos destinos pelo símbolo ⇒, pronunciado como "para".

```
Operação           → OpNormal
                   |  OpFluxoControle
OpNormal           → Opcode ListaOperandos ⇒ ListaOperandos
ListaOperandos     → Operando
                   |  Operando , ListaOperandos
Operando           → register
                   |  num
                   |  label
```

O não terminal *Opcode* pode ser qualquer operação ILOC, exceto `cbr`, `jump` e `jumpI`. Infelizmente, como em uma linguagem assembly real, o relacionamento entre um código de operação e o formato de seus operandos não é sistemático. O modo mais fácil de especificar o formato dos operandos para cada código de operação é em um formato tabular. As tabelas que ocorrem mais adiante neste apêndice mostram o número de operandos e seus tipos para cada código de operação ILOC usado no livro.

Operandos podem ser um de três tipos: `register`, `num` e `label`. O tipo de cada operando é determinado pelo código de operação e pela posição do operando na operação. Nos exemplos, usamos tanto nomes numéricos (r_{10}) quanto simbólicos (r_i) para os registradores. Números são inteiros simples, com sinal, se necessário. Sempre iniciamos um rótulo com um `l` (de *label*) para tornar seu tipo óbvio. Esta é uma convenção, e não uma regra. Os simuladores e ferramentas ILOC devem tratar qualquer sequência no formato descrito acima como um potencial rótulo.

A maioria das operações tem um único operando de destino; algumas das operações `store` têm vários operandos de destino, assim como os desvios. Por exemplo, `storeAI` tem um único operando de origem e dois operandos de destino. A origem precisa ser um registrador, e os destinos, um registrador e uma constante imediata. Assim, a operação ILOC

$$\text{storeAI } r_i \Rightarrow r_j, 4$$

calcula um endereço somando 4 ao conteúdo de r_j e armazena o valor encontrado em r_i no local de memória especificado pelo endereço. Em outras palavras,

$$\text{MEMÓRIA}(r_j + 4) \leftarrow \text{CONTEÚDO}(r_i)$$

As operações de fluxo de controle têm uma sintaxe ligeiramente diferente. Como elas não definem seus destinos, as escrevemos com uma seta simples, →, ao invés de ⇒.

		`cbr`	
OpFluxoControle	→	`register`	`label, label`
|		`jumpI`	`label`
|		`jump`	`register`

A primeira operação, `cbr`, implementa um desvio condicional. As outras duas operações são desvios incondicionais, chamados saltos.

A.2 Convenções de nomeação

O código ILOC nos exemplos de texto utiliza um conjunto simples de convenções de nomeação.
1. Deslocamentos (*offsets*) de memória para variáveis são representados simbolicamente prefixando o nome da variável com o caractere `@`.
2. O usuário pode considerar um estoque ilimitado de registradores. Estes são nomeados com inteiros simples, como em r_{1776}, ou com nomes simbólicos, como em r_i.
3. O registrador r_{arp} é reservado como um ponteiro para o registro de ativação atual. Assim, a operação

carrega o conteúdo da variável `x`, armazenada no deslocamento `@x` a partir de ARP, em r_1.

Comentários ILOC começam com a sequência `//` e continuam até o final de uma linha. Consideramos que estes são retirados pelo scanner; assim, podem ocorrem em qualquer lugar em uma instrução e não são mencionados na gramática.

A.3 Operações individuais

Os exemplos no livro utilizam um conjunto limitado de operações ILOC. As tabelas ao final deste apêndice mostram o conjunto de todas as operações ILOC usadas no livro, exceto pela sintaxe alternativa de desvio, utilizada no Capítulo 7 para discutir o impacto de diferentes formas de construções de desvio.

A.3.1 Aritmética

Para expressar aritmética, ILOC tem operações com três endereços, de registrador-para-registrador.

Opcode	Fontes	Destinos	Significado
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$

Todas essas operações leem seus operandos de origem a partir de registradores ou constantes e escrevem seu resultado de volta para um registrador. Qualquer registrador pode servir como operando de origem ou de destino.

As quatro primeiras são operações padrão de registrador-para-registrador. As seis seguintes especificam um operando imediato. As operações não comutativas, `sub` e `div`,

têm duas formas imediatas para permitir o operando imediato em qualquer lado do operador. As formas imediatas são úteis para expressar os resultados de certas otimizações, escrever exemplos de forma mais concisa e registrar maneiras óbvias de reduzir a demanda por registradores.

Observe que um processador real baseado em ILOC precisaria de mais de um tipo de dado, o que levaria a códigos de operação tipados ou a códigos de operação polimórficos. Preferimos uma família de *opcodes* tipados – um add de inteiro, um add de ponto flutuante e assim por diante. O compilador de pesquisa do qual o ILOC foi originado tem operações aritméticas distintas para inteiro, ponto flutuante de precisão simples, ponto flutuante de precisão dupla, complexo e dados de ponteiro, mas não para dados de caractere.

A.3.2 Deslocamentos (*Shifts*)

ILOC admite um conjunto de operações de deslocamento aritmético – à esquerda e à direita, nas formas de registrador e imediata.

Opcode	Origens	Destinos	Significado
lshift	r_1, r_2	r_3	$r_1 << r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 << c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 >> r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 >> c_2 \Rightarrow r_3$

A.3.3 Operações de memória

Para mover valores entre memória e registradores, ILOC admite um conjunto completo de operações load e store. As operações `load` e `cload` movem itens de dados da memória para registradores.

Opcode	Origens	Destinos	Significado
load	r_1	r_2	MEMÓRIA(r_1) ⇒ r_2
loadAI	r_1, c_2	r_3	MEMÓRIA(r_1 + c_2) ⇒ r_3
loadAO	r_1, r_2	r_3	MEMÓRIA(r_1 + r_2) ⇒ r_3
cload	r_1	r_2	`load` de caractere
cloadAI	r_1, c_2	r_3	`loadAI` de caractere
cloadAO	r_1, r_2	r_3	`loadAO` de caractere

As operações diferem nos modos de endereçamento que admitem. As formas `load` e `cload` consideram que o endereço completo está no único operando de registrador. As formas `loadAI` e `cloadAI` acrescentam um valor imediato ao conteúdo do registrador para formar um endereço imediato antes de realizar o load. Chamamos essas formas de operações de *endereço-imediato*. As formas `loadAO` e `cloadAO` acrescentam o conteúdo de dois registradores para calcular um endereço efetivo antes de realizar o load. Chamamos essas formas de operações de *endereço-deslocamento (offset)*.

Como uma forma final de load, o ILOC admite uma operação de load imediato simples. Ela usa um inteiro do fluxo de instruções e o coloca em um registrador.

Opcode	Origens	Destinos	Significado
loadI	c_1	r_2	c_1 ⇒ r_2

Uma IR completa, tipo ILOC, deve ter um load imediato para cada tipo distinto de valor que admite.

As operações store combinam com as operações load. ILOC admite tanto stores numéricos quanto stores de caractere em sua forma de registrador simples, nas formas endereço-imediato e endereço-deslocamento.

Opcode	Origens	Destinos	Significado
store	r_1	r_2	$r_1 \Rightarrow$ MEMÓRIA(r_2)
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow$ MEMÓRIA($r_2 + c_3$)
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow$ MEMÓRIA($r_2 + r_3$)
cstore	r_1	r_2	store de caractere
cstoreAI	r_1	r_2, c_3	storeAI de caractere
cstoreAO	r_1	r_2, r_3	storeAO de caractere

Não existe uma operação de store imediato.

A.3.4 Operações de cópia registrador-para-registrador

Para mover valores entre registradores sem passar pela memória, o ILOC inclui um conjunto de operações de cópia de registrador-para-registrador.

Opcode	Origens	Destinos	Significado
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ para inteiros
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ para caracteres
c2i	r_1	r_2	converte caractere em inteiro
i2c	r_1	r_2	converte inteiro em caractere

As duas primeiras operações, `i2i` e `c2c`, copiam um valor de um registrador para outro, sem conversão; o primeiro é para uso com valores inteiros; o segundo, para caracteres. As duas últimas operações realizam conversões entre caracteres e inteiros, substituindo um caractere por sua posição ordinal no conjunto de caracteres ASCII e um inteiro pelo caractere ASCII correspondente.

A.4 Operações de fluxo de controle

Em geral, os operadores de comparação ILOC usam dois valores e retornam um valor booleano. Se o relacionamento especificado for válido entre seus operandos, a comparação estabelece o registrador de destino com o valor verdadeiro; caso contrário, o registrador de destino recebe o valor falso.

Opcode	Origens	Destinos	Significado	
cmp_LT	r_1, r_2	r_3	true ⇒ r_3	if $r_1 < r_2$
			false ⇒ r_3	caso contrário
cmp_LE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \leq r_2$
			false ⇒ r_3	caso contrário
cmp_EQ	r_1, r_2	r_3	true ⇒ r_3	if $r_1 = r_2$
			false ⇒ r_3	caso contrário
cmp_GE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \geq r_2$
			false ⇒ r_3	caso contrário
cmp_GT	r_1, r_2	r_3	true ⇒ r_3	if $r_1 > r_2$
			false ⇒ r_3	caso contrário
cmp_NE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \neq r_2$
			false ⇒ r_3	caso contrário
cbr	r_1	l_2, l_3	$l_2 \rightarrow$ PC	if r_1 = true
			$l_3 \rightarrow$ PC	caso contrário

A operação de desvio condicional, `cbr`, usa um booleano como argumento e transfere o controle para um de dois rótulos de destino. O primeiro é selecionado se o booleano for verdadeiro; o segundo, se o booleano for falso. Como os dois destinos de desvio não são "definidos" pela instrução, mudamos a sintaxe ligeiramente. Ao invés de usar a seta ⇒, escrevemos desvios com a seta simples →.

Todos os desvios no ILOC têm dois rótulos. Esta técnica elimina um desvio seguido por um salto e torna o código mais conciso. Também elimina quaisquer caminhos "*fall-through*"; tornando-os explícitos, ela remove qualquer dependência posicional e simplifica a construção do grafo de fluxo de controle.

A.4.1 Sintaxe alternativa de comparação e desvio

Para discutir a forma de código em processadores que usam um código de condição, devemos introduzir uma sintaxe alternativa de comparação e desvio. O esquema de código de condição simplifica a comparação e empurra a complexidade para a operação de desvio condicional.

Opcode	Origens	Destinos	Significado	
comp	r_1, r_2	cc_3	define cc_3	
cbr_LT	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = LT
			$l_3 \rightarrow PC$	caso contrário
cbr_LE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = LE
			$l_3 \rightarrow PC$	caso contrário
cbr_EQ	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = EQ
			$l_3 \rightarrow PC$	caso contrário
cbr_GE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = GE
			$l_3 \rightarrow PC$	caso contrário
cbr_GT	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = GT
			$l_3 \rightarrow PC$	caso contrário
cbr_NE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if cc_3 = NE
			$l_3 \rightarrow PC$	caso contrário

Aqui, o operador de comparação, `comp`, usa dois valores e define o código de condição de modo apropriado. Sempre designamos o destino de `comp` como um registrador de código de condição escrevendo-o como cc_1. O desvio condicional correspondente tem seis variantes, uma para cada resultado de comparação.

A.4.2 Saltos

ILOC inclui duas formas da operação de salto. A forma usada em quase todos os exemplos é um salto imediato que transfere o controle para um rótulo literal. A segunda, uma operação de salto-para-registrador, usa um único operando de registrador. Ela interpreta o conteúdo do registrador como um endereço de runtime e

transfere o controle para esse endereço.

Opcode	Origens	Destinos	Significado
jumpI	—	l_1	$l_1 \rightarrow$ PC
jump	—	r_1	$r_1 \rightarrow$ PC

A forma salto-para-registrador é uma transferência ambígua de fluxo de controle. Quando tiver sido gerada, o compilador pode ser incapaz de deduzir o conjunto correto de rótulos de destino para o salto. Por este motivo, o compilador deve evitar, se possível, usar o salto para registrador.

Às vezes, as voltas necessárias para evitar um salto para registrador são tão complexas que o salto para registrador se torna atraente, apesar de seus problemas. Por exemplo, FORTRAN inclui uma construção que salta para uma variável de rótulo; sua implementação com desvios imediatos exigiria uma lógica semelhante a uma instrução case – uma série de desvios imediatos, junto com o código para fazer a correspondência entre o valor de runtime da variável de rótulo e o conjunto de rótulos possíveis. Em tais circunstâncias, o compilador provavelmente usaria um salto para registrador.

Para reduzir a perda de informações do salto para registrador, ILOC inclui uma pseudo-operação que permite que o compilador registre o conjunto de rótulos possíveis para um salto para registrador. A operação `tbl` tem dois argumentos, um registrador e um *label* imediato.

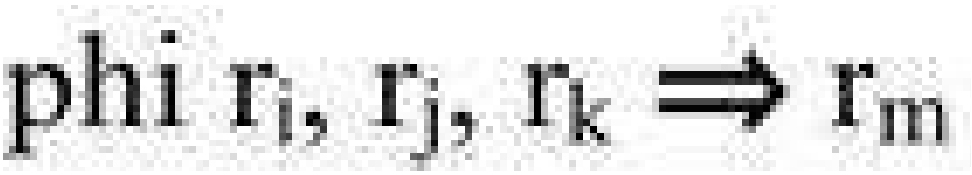
Opcode	Origens	Destinos	Significado
tbl	r_1, l_2	—	r_1 poderia manter l_2

Uma operação `tbl` pode ocorrer somente após um `jump`. O compilador interpreta um conjunto de um ou mais `tbl`s como nomeação de todos os *labels* possíveis para o registrador. Assim, a sequência de código a seguir declara que o salto visa um dentre os rótulos `L01`, `L03`, `L05` ou `L08`:

```
jump                                    →ri

tbl ri, L01

tbl ri, L03

tbl ri, L05

tbl ri, L08
```

A.5 Representação da forma SSA

Quando um compilador constrói a forma SSA de um programa a partir de sua versão IR, ele precisa de um modo para representar funções-ϕ. Em ILOC, o modo natural de escrever uma função-ϕ é como uma operação ILOC. Assim, às vezes escrevemos



para a função-ϕ $r_m \leftarrow \phi(r_i, r_j, r_k)$. Devido à natureza da forma SSA, a operação `phi` pode usar um número qualquer de origens. Ela sempre define um único destino.

Resumo dos códigos de operação ILOC

Opcode	Origens	Destinos	Meaning
nop	*nenhum*	*nenhum*	Usado para preencher espaço (*placeholder*)
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$
and	r_1, r_2	r_3	$r_1 \wedge r_2 \Rightarrow r_3$
andI	r_1, c_2	r_3	$r_1 \wedge c_2 \Rightarrow r_3$
or	r_1, r_2	r_3	$r_1 \vee r_2 \Rightarrow r_3$
orI	r_1, c_2	r_3	$r_1 \vee c_2 \Rightarrow r_3$
xor	r_1, r_2	r_3	r_1 *xor* $r_2 \Rightarrow r_3$
xorI	r_1, c_2	r_3	r_1 *xor* $c_2 \Rightarrow r_3$
loadI	c_1	r_2	$c_1 \Rightarrow r_2$
load	r_1	r_2	MEMÓRIA(r_1)$\Rightarrow r_2$
loadAI	r_1, c_2	r_3	MEMÓRIA($r_1 + c_2$)$\Rightarrow r_3$
loadAO	r_1, r_2	r_3	MEMÓRIA($r_1 + r_2$) $\Rightarrow r_3$
cload	r_1	r_2	load de caractere
cloadAI	r_1, c_2	r_3	loadAI de caractere
cloadAO	r_1, r_2	r_3	loadAO de caractere
store	r_1	r_2	$r_1 \Rightarrow$ MEMÓRIA(r_2)
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow$ MEMÓRIA($r_2 + c_3$)
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow$ MEMÓRIA($r_2 + r_3$)
cstore	r_1	r_2	store de caractere
cstoreAI	r_1	r_2, c_3	storeAI de caractere
cstoreAO	r_1	r_2, r_3	storeAO de caractere
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ para inteiros
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ para caracteres
c2i	r_1	r_2	converte caractere em inteiro
i2c	r_1	r_2	converte inteiro em caractere

Operações de fluxo de controle ILOC

Opcode	Origens	Destinos	Significado	
jump	—	r_1	$r_1 \rightarrow$ PC	
jumpI	—	l_1	$l_1 \rightarrow$ PC	
cbr	r_1	l_2, l_3	$l_2 \rightarrow$ PC	if r_1 = true
			$l_3 \rightarrow$ PC	caso contrário
tbl	r_1, l_2	—	r_1 poderia manter l_2	
cmp_LT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 < r_2$
			false $\Rightarrow r_3$	caso contrário
cmp_LE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \leq r_2$
			false $\Rightarrow r_3$	caso contrário
cmp_EQ	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 = r_2$
			false $\Rightarrow r_3$	caso contrário
cmp_GE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \geq r_2$
			false $\Rightarrow r_3$	caso contrário
cmp_GT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 > r_2$
			false $\Rightarrow r_3$	caso contrário
cmp_NE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \neq r_2$
			false $\Rightarrow r_3$	caso contrário
comp	r_1, r_2	cc_3	define cc_3	
cbr_LT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = LT
			$l_3 \rightarrow$ PC	caso contrário
cbr_LE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = LE
			$l_3 \rightarrow$ PC	caso contrário
cbr_EQ	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = EQ
			$l_3 \rightarrow$ PC	caso contrário
cbr_GE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = GE
			$l_3 \rightarrow$ PC	caso contrário
cbr_GT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = GT
			$l_3 \rightarrow$ PC	caso contrário
cbr_NE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if cc_3 = NE
			$l_3 \rightarrow$ PC	caso contrário