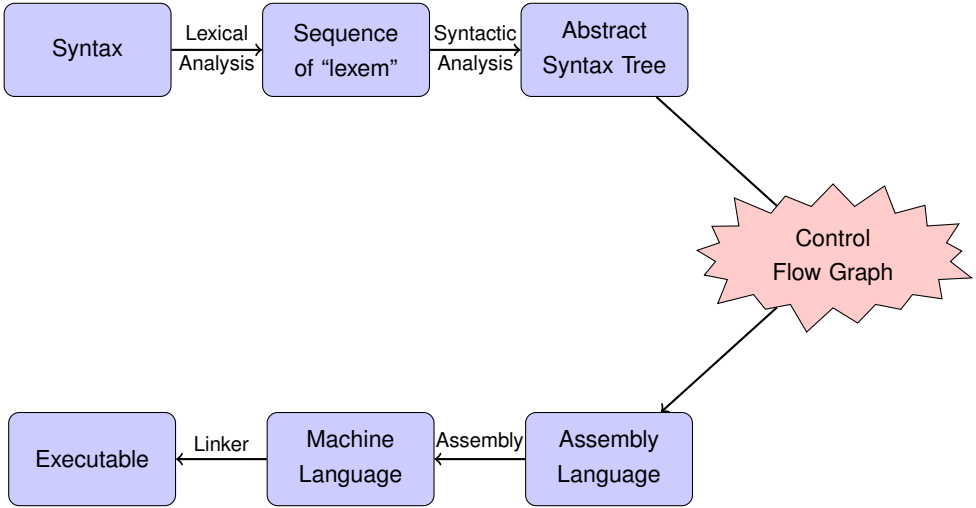
Compilation (#7) : Intermediate Representations: CFG, DAGs, and local optimisations (Instruction Selection and Scheduling)

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Big Picture


Syntax
Lexical Analysis
Sequence of “lexem”
Syntactic Analysis
Abstract Syntax Tree
Control Flow Graph
Executable
Linker
Machine Language
Assembly
Assembly Language

3 address construction “problems”

Temporary reuse ?

```
li temp3, 4
mv temp0, temp3
;; temp3 is never used again
li temp4, 0
mv temp1, temp4
```

`temp3` and `temp4` could be mapped to the same physical location.

```
li temp5, 4
bge ..., foo
;; temp5 not used.
;; Its physical location
;; can be shared.
j end
foo:
;; temp5 used
end
```

A first IR

We thus need a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

- Control-Flow Graph.

1. Control flow Graph
2. Local optimizations

Definitions

Definition (Basic Block)

Basic block: largest (3-address RISCXX*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ *exists if* B_2 *can follow immediately* B_1 *in an execution.*

- two optimisation levels: local (BB) and global (CFG)

An example 1/2

Let us consider the program:

```
int x,y;
if (x<4) y=7; else y=42;
x=10;
```

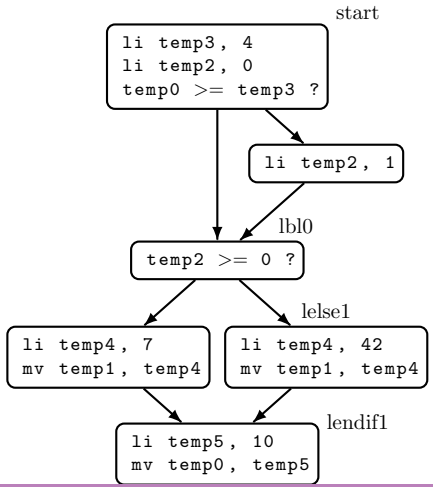
We already generated the (linear code) for a large part of it.

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```



Identifying Basic Blocks (from 3 address code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

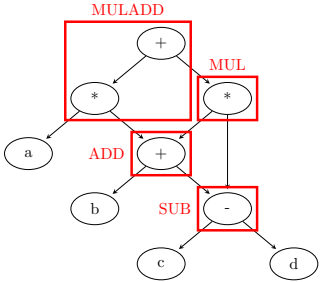
1 Control flow Graph

2 Local optimizations
- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

Big picture (Basic Block Optimisation)

- Front-end $\rightarrow$ a CFG where nodes are basic blocks.
- Basic blocks $\rightarrow$ DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



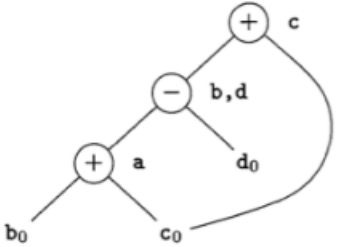
- choose instructions(**selection**) and order them (**scheduling**).

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

An Example of BB DAG construction

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Useful links : https://www.youtube.com/watch?v=PXTKWvyQUwE and

https://www.cse.iitm.ac.in/~krishna/cs3300/pm-lecture3.pdf for other BB optimisations.

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

Instruction Selection, in general

The problem:

- a list of instructions/operations that compute one or more expressions.
- map these operations in “real machine instructions”.
- at minimum cost.

Instruction Selection

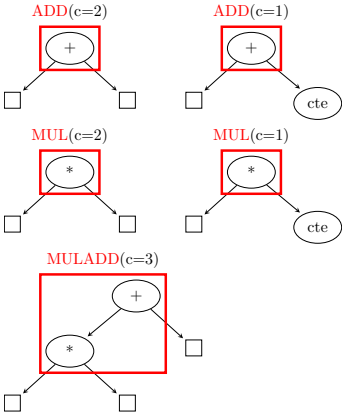
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions:

- cover bigger parts of computation.
- cause few memory accesses.

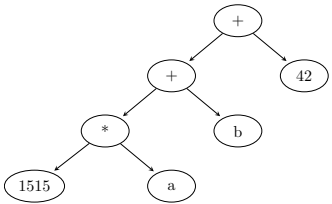
- Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example



(Our RISCXX has no MULADD instruction nor “add with constants”, this is just an example).

What is the optimal instruction selection for:



- Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practice

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programming.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The literature is plethoric on the subject.

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

Instruction Scheduling, in general

The problem:

- change the order of instructions.
- to “optimise’.
- without “cutting dependencies”.

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependencies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$. We can choose $\theta(S_1) = 0, \theta(S_2) = 1$

- How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

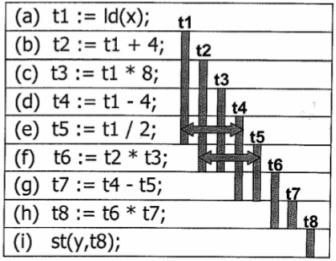
The idea is to exploit the different ressources of the machine at their best:

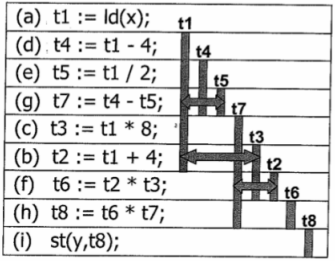
- instruction parallelism: some machines have parallel units (subinstructions of a given instruction).
- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)
- pipeline.
- registers: see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:





In this picture the dates of the instructions are implicit : line 1 is date 1, line 2 is date 2...

- How to find a schedule with less register pressure?

Scheduling wrt register pressure

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

A slight variation of this algorithm can be found on Wikipedia, the leaves values here are chosen equal to 1 since our machine does not have any direct access to constant values.

Sethi-Ullman algorithm on trees

$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

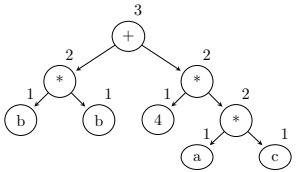
- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non “balanced” subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example

Min number of (additional) registers for $b^2 + 4ac$ with a,b,c already in registers ?



The tree traversal then produces the following code:

	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b, b`				
`mul tmp2, a, c`	■			
`li tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`add tmp5, tmp1, temp4`	■			■

cells in black denote for each instruction the set of entry alive temporaries.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, ...
- Scheduling loops **in advanced compilation courses**

Bilan

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling