CS502: Compilers & Programming Systems

Low IR and Assembly code generation

Zhiyuan Li
Department of Computer Science
Purdue University, USA

PURDUE
UNIVERSITY

- We first discuss the generation of low-level intermediate code of a program.
  - Inside the compiler such a code has its low-level representation, called *low-level internal representation*, or *low IR*.
- We then discuss the generation of assembly code by walking through the low-level IR

PURDUE
UNIVERSITY

Motivation of machine independent representation

- From the MIPS code example shown previously, we see that the assembly code is heavily dependent on the specific instruction format of a particular processor
- We want to perform optimizations on machine independent representation such that we do not implement a new set of optimizations for each particular processor

Differences between Low IR and Assembly Code

- In low IR
  - Unlimited user-level registers are assumed. Hence,
    - symbolic registers, instead of hardware registers are named as operands.
  - Operators on different data type are overloaded.
    - For example, “+” represents both integer add and float add.
  - Instructions selected for special cases are abstracted away.
    - For example, a simple load instruction represents loading of a value, instead of two instructions (loading the high half and then the low half) as on RISC processors such as MIPS and SPARC.
    - Another example is that we simply write R4 * 2 when it could be done by a potentially less expensive left shift instruction.

Intermediate Code

- A well-known form of intermediate code is called three-address code (3AC)
  - In the "quadruples" form, 3AC for a function is represented by a list quadruples (dest, op1, op2, op), with "op" being the operation
  - An alternative form is a list of "triples" (op1, op2, op) and the index or the pointer to a specific triple implies the location of the intermediate result
- When we plug such locations (indices or pointers) to the operand fields of an operation, we in effect obtain a forest, i.e. a number of operation trees
  - Similar to AST but at lower level

- Usually the internal nodes are intermediate results that are associated with temporary variable names (*temps*) assigned by the compiler
  - These are also known as *symbolic registers*
  - They are eventually allocated to hardware registers
  - If there are not enough hardware registers, such temp variables may need to have their values saved to the activation record of the current function,.
    - This is called *register spills*, to be discussed further later.

- If one exploits *common sub-expression* (*CSE*) s in a *basic block*, i.e. a code block w/o branches in or out, then the expression trees may become directed acyclic graphs (DAGs).
  - The algorithm to identify CSEs uses the *value number:*
    - *The idea is to find isomorphism among sub-trees.*

PURDUE UNIVERSITY

From AST to Low IR

- We shall assume a load-store processor architecture as on RISC processors.
  - As in assembly code, all static memory addresses are symbolic labels.
  - Variables local to a routine are referred to via stack pointer (or frame pointer) plus an offset.

PURDUE
UNIVERSITY

We assume the following 3AC instruction types:

- **ALU operations:** $R_x = R_y$ *op* R_z, $R_x = R_y$ *op integer*, $R_x =$ *integer op integer*, $R_x =$ *integer op* R_z. **(If** *op* **is a logical or comparison operation, then** R_x **becomes 0 if the result is** *false* **and becomes 1 if** *true***.)**
- **Simple register assignments:** $R_x = R_y$, $R_x =$ *integer*, $R_x = Label_y$, **where** $Label_y$ **is a symbolic memory address.**
- **Load operations:** $R_x = (R_y)$, $R_x = (Label_y)$, **where** $Label_y$ **is a symbolic memory address.**
- **Store operation:** $(R_y) = R_x$, $(Label_y) = R_x$,

PURDUE
UNIVERSITY

- **Branch operations:**
  - **if** R_c **goto** $Label_x$ **(if** $R_c \neq 0$ **jump to** $Label_x$**).**
  - **if** R_c **goto** (R_x) **(if** $R_c \neq 0$ **jump to the address stored in** R_x**).**
  - **goto** $Label_x$**, goto** (R_x)**.**

- We assume a few symbolic registers with fixed meanings:
  - %pc – program counter.
  - %sp – stack pointer.
  - %fp – stack frame pointer.
- We assume all labels are inserted as "some ID name :",
  e.g. "L11: R1 = R2 + R3".
- If a label is followed by a directive ".size n", then a memory space of n words is reserved for the static variable named by the label.
  For a scalar, n = 1.

PURDUE
UNIVERSITY

Generating Three Address Code

- Three address code (3AC) can be generated directly by semantic actions.
- But It is easier to construct the AST first and then generate 3AC by traversing the AST.
- In this course, we focus on explaining 3AC for different kinds of program constructs
  - and we skip the traversal procedure and the mechanical transformation steps

PURDUE UNIVERSITY

- First consider a basic block, i.e. a sequence of statements containing no branches.

1. a = (X * Y + Z / 100) * 3;
2. b = a * a + 3.14;
3. c = a * a + b * b;
4. b = b + c;

- Assuming X, Y and Z are global variables statically allocated, what should the 3AC be for the above C code segment?
  - Consider two cases: (1) a, b, c, d are also globals; and (2) a, b, c, d are locals which are not static. In the case of (2), we need to know how the locals are allocated in the AR.

Branches

- Next consider IF statements.

```
1.  if ((a < b) || (c < d) && (e < f))
2.  x = 3;
3.  else
4.  x = 2;
5.  y = x * x;
```

- There are two methods to evaluate the IF condition:
  - One is based a full evaluation of the logical condition,
  - the other generates *jump code* to *short-circuit* the IF condition. The latter is more commonly used.
- Nested IF statements be treated just the same way

PURDUE UNIVERSITY

Loops

- Next consider a WHILE loop.

```
1.  i = 100;
2.  while ((i > 0) && (x < 10000)) {
3.  x = x * x;
4.  i = i - 1;
5.  }
```

- Now consider another WHILE loop which contains array references.

```
• i = 100;
• while (i > 0) {
• i = i - 1;
• a[i] = b[i] * 3.14;
• }
```

PURDUE UNIVERSITY

Function Calls

- Lastly, consider a recursive function call.
- int main() {
- int i;
- i = fibo(100);
- }
- int fibo(int n) {
- if (n < 0) return -1;
- if (n == 1) return 1;
- if (n == 0) return 0;
- return fibo(n-1) + fibo(n-2);
- }

PURDUE
UNIVERSITY

Assembly Code Generation

- Assembly code generation concerns
  - (i) instruction selection
  - (ii) register allocation
- We first focus on instruction selection
  - Register allocation will be discussed in later lectures
- During instruction selection, we will assume the use of abundant symbolic registers
  - During register allocation later, if register spills occur, the assembly code will be slightly modified by inserting the “spilling code”
  - Designation of callee-save and caller-save registers will also affect the prologue and epilogue

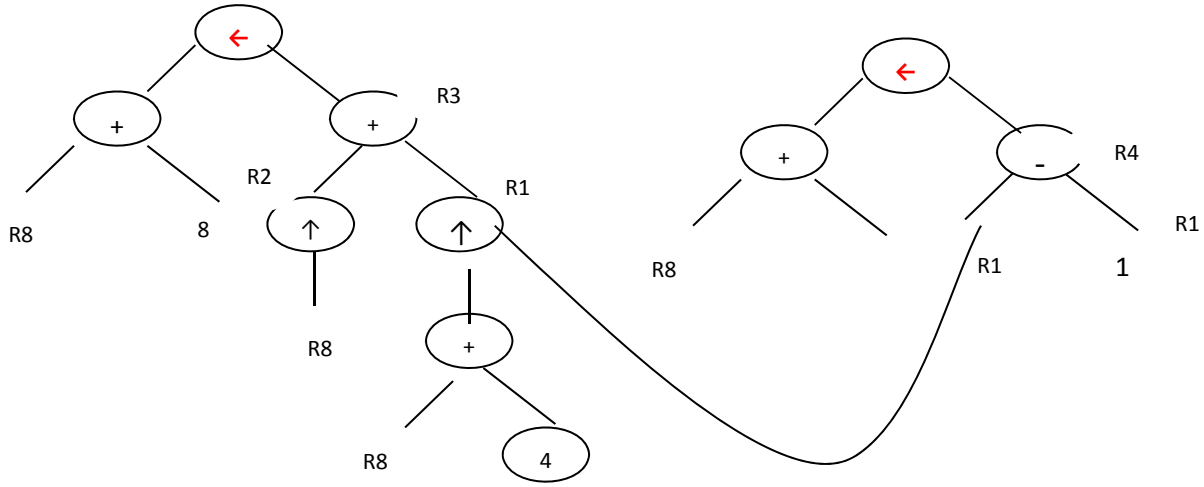
Assumptions in Code Generation

- We assume that the input to the code generator is a *basic block* (single entry and single exit) of 3AC operations, in which no branch operations may occur.
- As stated previously, memory allocation is already performed and made explicit in the 3AC (e.g. the RTL in GCC).
- To facilitate code generation, if the 3AC is a non-tree DAG (directed acyclic graph), then it is first converted to a forest, (just as in the RTS of GCC).

- The root of each tree can either be
  - a memory store, or
  - a write to a symbolic register that is live at the end of the basic block, i.e. to be used by some operations outside the basic block
- A node in the 3AC may or may not yet have been labeled by a (symbolic or hardware) register to hold its value.
- See an illustration

In the following 3AC, *r8* is assumed to be the stack frame pointer

r2 ← [r8]
r1 ← [r8+4]
r3 ← r2 + r1
[r8+8] ← r3
r4 ← r1 – 1
[r8+4] ← r4



In the graph, a red ← represents a memory store, a ↑ represents a memory load.

Before generating machine code, the arc representing the common sub-expression stored at R1 is deleted, which results in a forest of two trees, both rooted at "←".

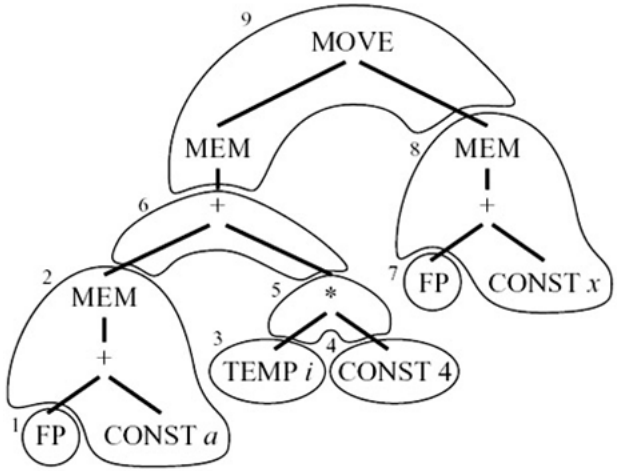
- For modern microprocessors, each node in the 3AC can always find a machine instruction to implement it.
- Often the implementation can have more than one way.
- Moreover, often a single instruction may implement more than one node in the 3AC.
- Which way is “best”, that is an optimization issue.
- The objective function may vary
  - Code size?
  - Speed?

- In general, the optimization problem is NP-hard even if we know exactly the run-time trace of the program execution
- Instead of performing dynamic programming as some literature suggests, current compilers usually traverse the 3AC only a fixed time, e.g. twice.

In what order should we traverse the tree?

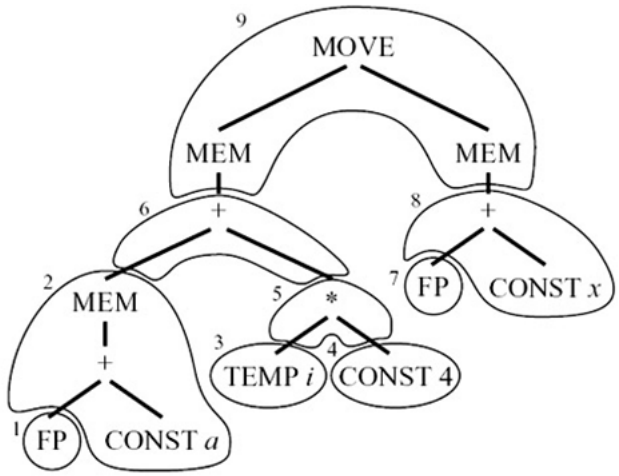
- If bottom-up, do we save r8 + 8 to a register, or we wait till we find out it is actually part of the memory store operation's addressing mode? The latter will results in one fewer instruction and is better.
- However, if we traverse top down, then we don't know what register holds an operand.
- Hence, we take two passes.
  - In the top down pass, the tree is partitioned in non-overlapping tiles, using *maximal munch*.
  - In the bottom up pass, instructions are generated following the proper execution order of the tiles.

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

- The tree walk and instruction pattern matching can be implemented by hand in the compiler.
- However, it is more desirable to automate part of the process, by using some code-generator generator. One of the attempt in this direction is based on an LR parsing approach proposed by *Glanville and Graham* (not discussed further here).