Low-level Code Generation

- Assembly code generation
  - Register allocation
  - Instruction selection
- Machine code generation
  - Instruction encoding
  - Linker and loader
  - Relocatable code
    - Defer assignment of locations for static objects (code, variables) to linking phase
      - Static linking
      - Dynamic linking

Machine code generation (contd.)

- Position-independent code (PIC)
  - Can be shared by different processes that map a library to different locations
  - Code does not assume knowledge of memory location of its code or variables
- Symbol tables
  - Often, code that is shipped has all symbols “stripped off”
  - For libraries, need to maintain a minimal amount of symbol info

Register Allocation: Factors

- Special-purpose registers
  - Stack pointer, Base pointer, Instruction pointer, ...
  - Reserved for specific uses across most code
    - Register allocation deals with general-purpose registers
- Application/binary interface requirements
  - Caller- Vs Callee-save registers
    - Caller-save registers need to be explicitly saved by the caller before every procedure call, and restored after
    - Callee-save registers have to be saved before use by every function, and restored if used.
- Some (most) instructions may operate only on register operands

Register Allocation: Simple Strategies

1. Load a register from memory before each operation, store immediately afterwards
   - Too inefficient
2. Avoid load/store's within a basic block
   - Load registers at entry of a BB, and store at its end.
   - Fails to discriminate between loops and other Bbs
   - May require too many registers

- “Global” register allocation
  - Consider uses across Bbs
  - Even more “pressure” on registers ...

Global Register Allocation

- Model cost of instructions
  - Cost of fetching
    - On modern processors, fetching costs can be ignored to a certain extent due to the use of dedicated pipelines for instruction fetching/decoding, plus branch prediction etc.
  - Cost of memory access
    - For loading registers
    - For saving registers
    - For accessing memory (in case of instructions that accept memory operands)
  - Take into account loops
    - e.g., treat the cost of non-loop operations to be zero

Register usage counts

- Use(x) = number of uses of variable x (before reassignment) within a block, plus 2 if x is live at the end of the loop
  - Use registers to hold variables with highest use count
- If there are nested loops, allocate registers for innermost loop, and then allocate remaining registers to outer loops
  - Alternatively, reuse registers used in inner loops in outer loops by saving/restoring registers
  - Avoid unnecessary save/restores by analyzing across BBs to find variables used in inner as well as outer loops.

Working with fixed number of Registers

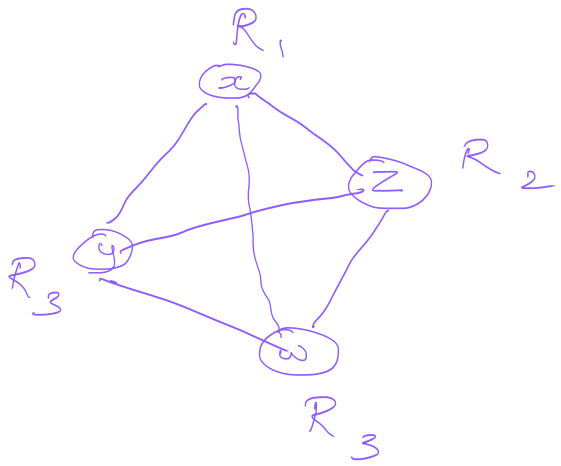
- Can be modeled as a graph-coloring problem
  - Allocate a symbolic register for each variable
  - Construct a register-interference graph (RIG)
    - Edge between two symbolic registers if one is live at the point where the other is assigned
  - You can use N registers if RIG is N-colorable
    - i.e., there is a way to assign N colors to graph nodes such that neighboring nodes have different colors

Graph-coloring (contd.)

- Graph-coloring problem is NP-complete
  - But good heuristics exist:
    - Eliminate all nodes that have less than degree N
      - Eliminating one node will reduce the degree of nodes connected to it
      - Color for the eliminated node can be chosen to be one of those that is not assigned to any of its neighbors
    - If all nodes have degree >= N, pick one to “spill,” i.e., save to memory and restore later
      - Pick registers that have least cost savings
      - Avoid spills in inner loops

```
x = 5;
y = 3;
z = x + y;
w = z - 5;
y = x + z;
z = w + 3;
```

x, z, w



Instruction Selection

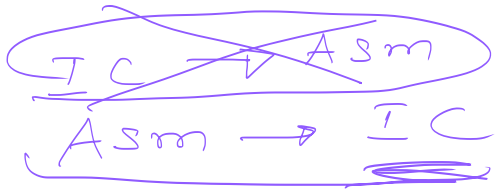
- Instruction selection is a complex task, especially when considering modern processors with a large number of instructions and addressing modes
- Many semantically equivalent instructions sequences may perform the same desired task
  - How to select the “minimal cost” sequence?
- Ideally, one does not have to hand-code a code generator, but have it be generated from specifications!
  - Instruction selection by tree-rewriting
  - Initially, the tree represents generated intermediate code

Target code generation in GCC

- gcc uses machine descriptions to automatically generate code for target machine
  - Enables gcc to support numerous target machines, with greatly reduced programmer effort
- machine descriptions specify:
  - memory addressing (bit, byte, word, big-endian, ...)
  - registers (how many, whether general purpose or not, ...)
  - stack layout
  - parameter passing conventions
  - semantics of instructions
  - ...

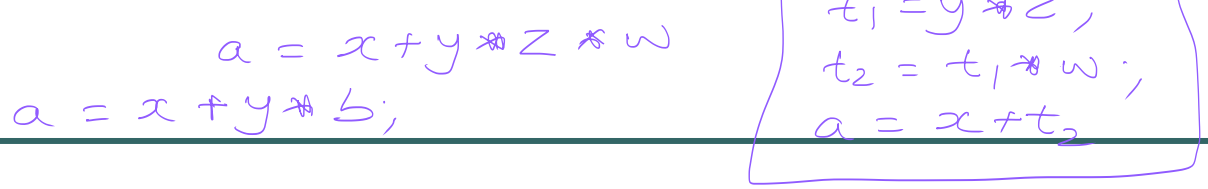
Instruction Specification

- For each instruction in target language, specify:
  - Assembly representation of target machine instructions
    - Instruction parameters include registers and constants
  - Its semantics in the intermediate language
    - Parameterized in terms of registers and constants in the target instruction
    - Specify input operands as well as the location where the result is stored
  - Cost of executing the instruction
  - Additional constraints on applicability of instruction
    - e.g., a certain constant must be at most 8 bits



Code generation by rewriting

- Represent intermediate code generated by the compiler as a tree, and use rewriting using the rules in the instruction specification
- Trees can represent expressions as well as sequence of statements
  - Introduce a sequencing operation to represent sequencing
  - Don't force sequencing of unrelated statements, or else the code generator won't be able to choose evaluation orders that lead to more efficient code.
    - Example: a=b+5; c=d+5; e=a+b
    - More efficient if c=d+5 is moved later, as it would allow a and b to continue to be in registers while evaluating e=a+b



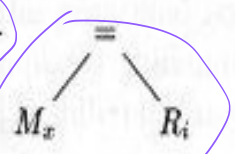
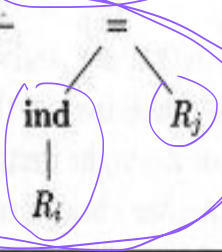
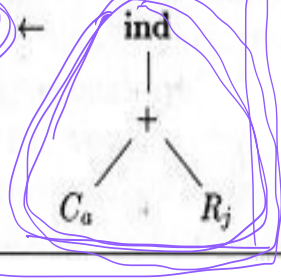
GCC target code generation

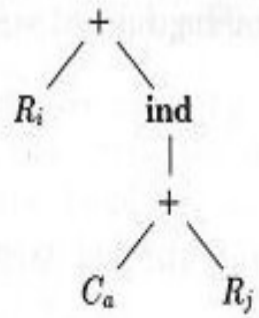
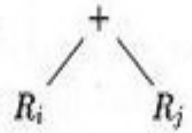
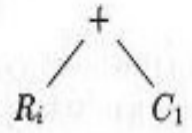
- gcc uses intermediate code called RTL, which uses a LISP-like syntax
  - Actually, gcc uses multiple intermediate languages, with RTL being the lowest level among them
- semantics of each instruction is also specified using RTL:
  - **movl (r3), @8(r4)**

```
(set (mem: SI (plus: SI (reg: SI 4) (const_int 8)))
     (mem: SI (reg: SI 3)))
```

- gcc code generation = selecting a low-cost instruction sequence that has the same semantics as the intermediate code

Instruction Specification

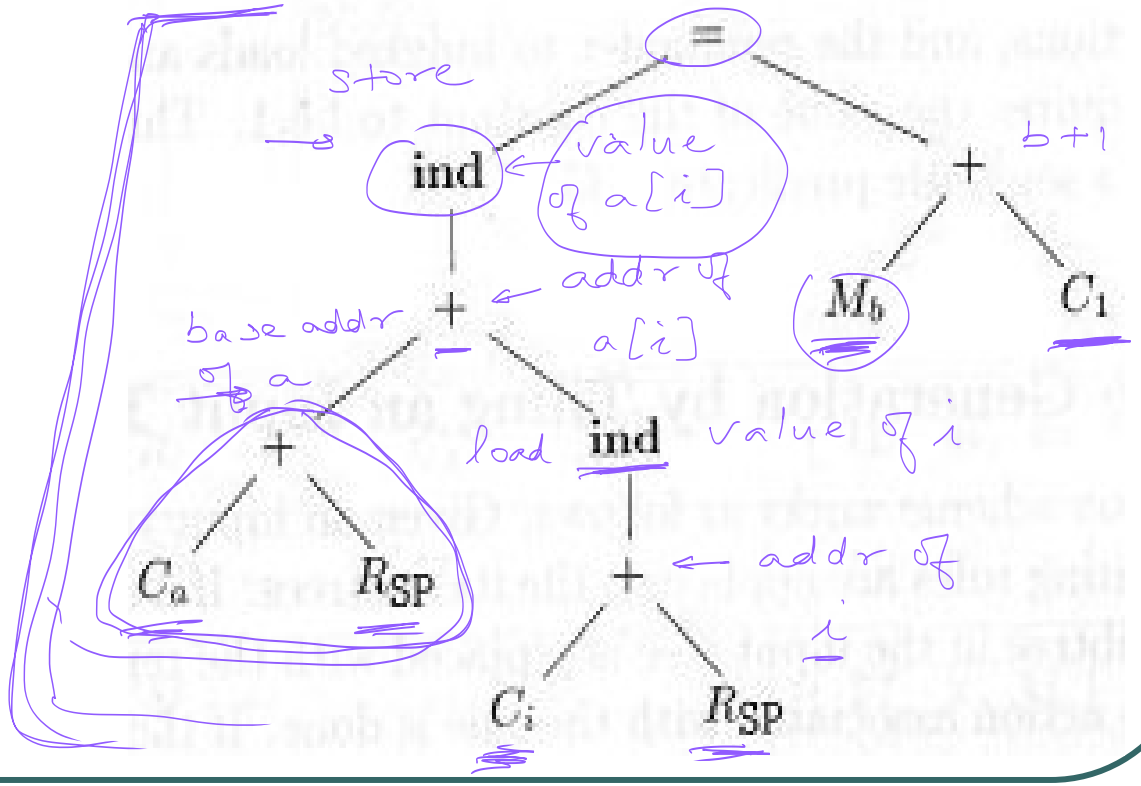
1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ 	{ ST x, Ri }
4)	$M \leftarrow$ 	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ 	{ LD Ri, a(Rj) }

6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

Instruction Selection Example

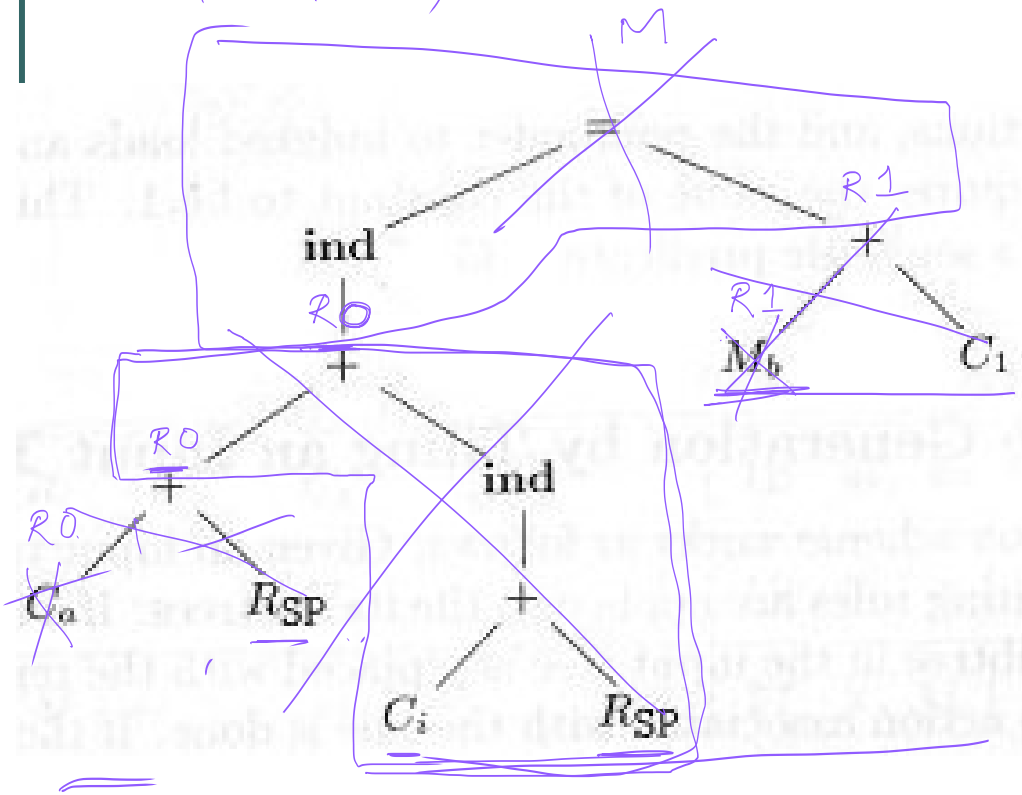
- Intermediate code for a[i] = b+1
- Rewrite tree repeatedly using rules corresponding to instruction specifications until you get to a single node tree.
- Result

```
LD    R0, #a
ADD R0, R0, SP
ADD R0, R0, i[SP]
LD    R1, b
INC  R1
ST    *R0, R1
```



```
LD  R0, #a
ADD R0, R0, RSP
ADD R0, R0, i(RSP)

LD  R1, b
INC R1
ST  *R0, R1
```



1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ = (M_x, R_i)	{ ST x, Ri }
4)	$M \leftarrow$ = (ind(R_i), R_j)	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ ind(+(C_a, R_j))	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ + (R_i, ind(+(C_a, R_j)))	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ + (R_i, R_j)	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ + (R_i, C_1)	{ INC Ri }

Optimal Code Generation

- Some intermediate operations may not have equivalent instructions
  - e.g.,“add R0, R0, M” versus “ld R1, M; add R0, R0, R1”
- Multiple rules may match the same node
  - Cost of evaluation may hinge on which match is chosen
  - Example: “inc R0” versus “add R0, 1”
- The order of rewriting can change the cost
  - Mainly due to selection of registers, and based on which intermediate results remain in registers as opposed to being stored in memory.

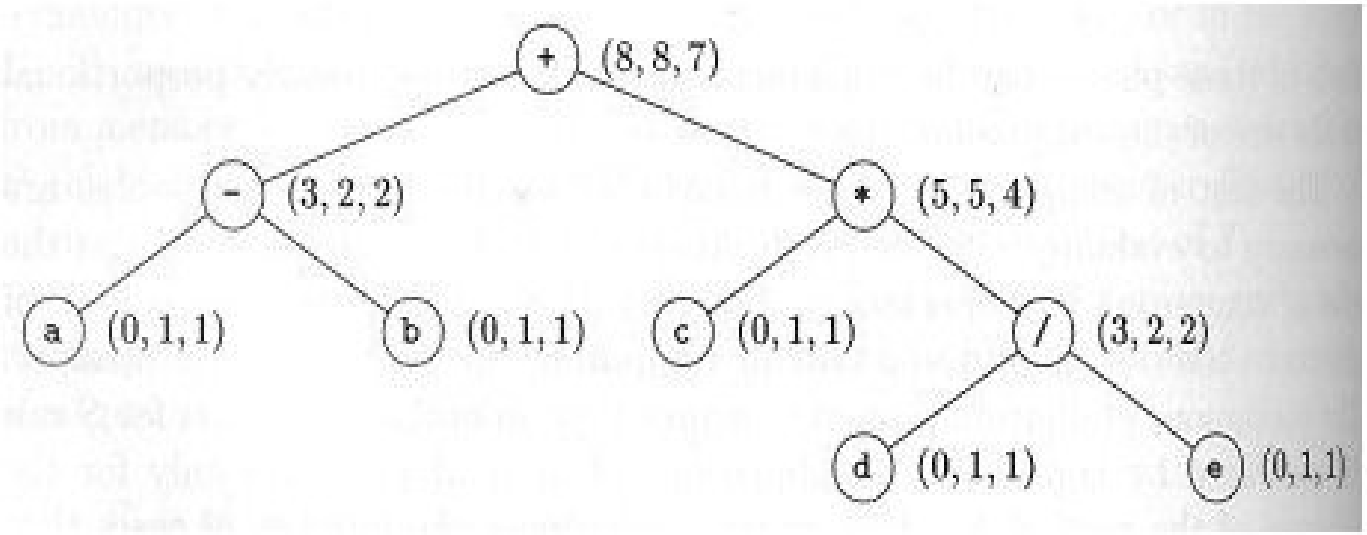
Optimal Code Generation

- But, dynamic programming algorithms for optimal code generation exist under reasonable assumptions
  - Optimal code for E1 op E2 will contain optimal code for evaluating E1 and optimal code for evaluating E2
  - Dynamic programming algorithm tries to construct the optimal code bottom-up: from E1 and E2's optimal codes, build optimal code for E1 op E2
  - Dynamic programming algorithm iterates over
    - number of registers used for operand evaluation
    - order of evaluation of operand (when permissible)

Dynamic Programming Algorithm

- For each node *n* in tree, compute *C*[*n*][*i*] which represents the minimum cost for evaluating the subtree rooted at *n* using at most *i* registers, for 0 <= *i* <= *k* (# of registers in the target architecture)
- The operands for evaluating the operation at *n* may differ, depending on the matching instruction
- While evaluating operands of *n,* we may use:
  - All *i* registers for evaluating each operand, but this requires evaluation results to be stored in memory in order to free up registers for evaluating other operands
  - Use less than *i* registers so that operands can be retained in registers
  - We prefer an order of evaluation that minimizes the number of registers that need to be saved to memory
- For the root node *r*, pick how many registers to use (may be *k*)
- Generate instructions based on the choices at each node that result in the least cost for *C*[*r*][*k*]

Illustration of Dynamic Programming Algorithm




```
LD  Ri, Mj        // Ri = Mj
op  Ri, Ri, Rj    // Ri = Ri op Rj
op  Ri, Ri, Mj    // Ri = Ri op Mj
LD  Ri, Rj        // Ri = Rj
ST  Mi, Rj        // Mi = Rj
```

Target Instructions

```
LD  R0, c          // R0 = c
LD  R1, d          // R1 = d
DIV R1, R1, e      // R1 = R1 / e
MUL R0, R0, R1     // R0 = R0 * R1
LD  R1, a          // R1 = a
SUB R1, R1, b      // R1 = R1 - b
ADD R1, R1, R0     // R1 = R1 + R0
```

Optimal Code