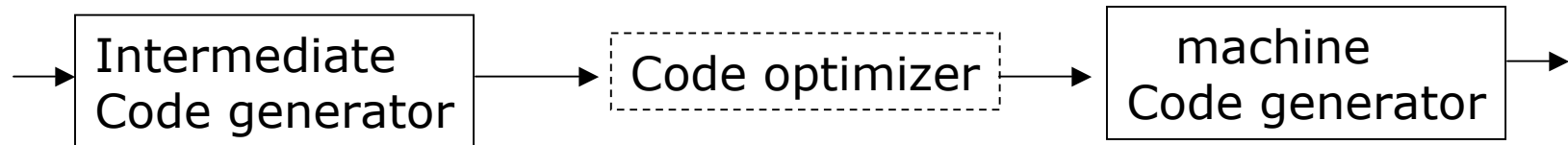


Code Generation



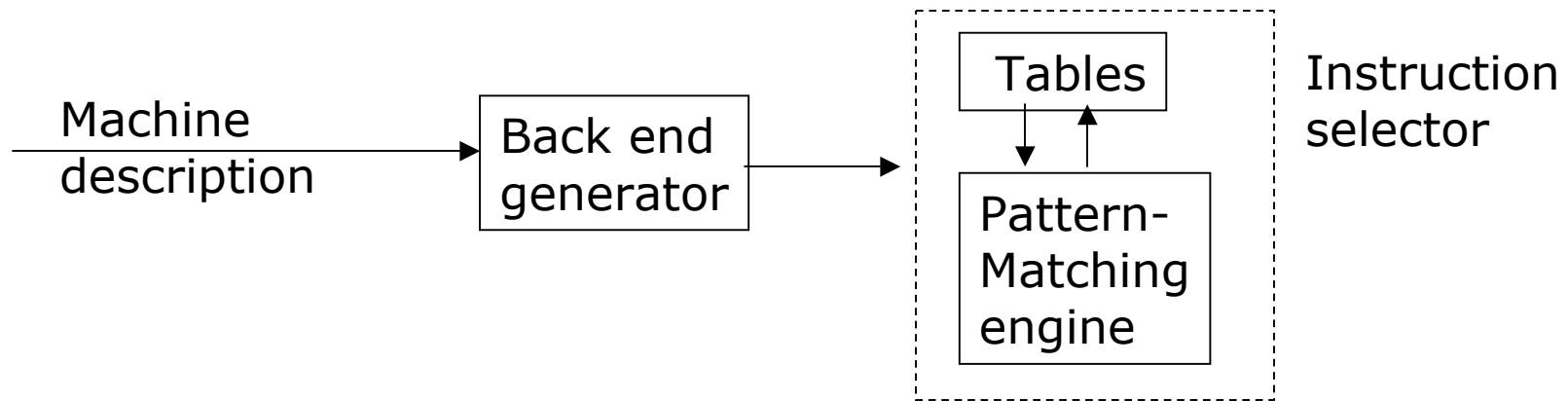
Machine code generation

Machine code generation



- Input: intermediate code + symbol tables
 - In our case, three-address code
 - All variables have values that machines can directly manipulate
 - Assume program is free of errors
 - Type checking has taken place, type conversion done
- Output:
 - Absolute/relocatable machine code or assembly code
 - In our case, use assembly
 - Architecture variations: RISC, CISC, stack-based
- Issues:
 - Memory management, instruction selection and scheduling, register allocation and assignment

Retargetable Back End



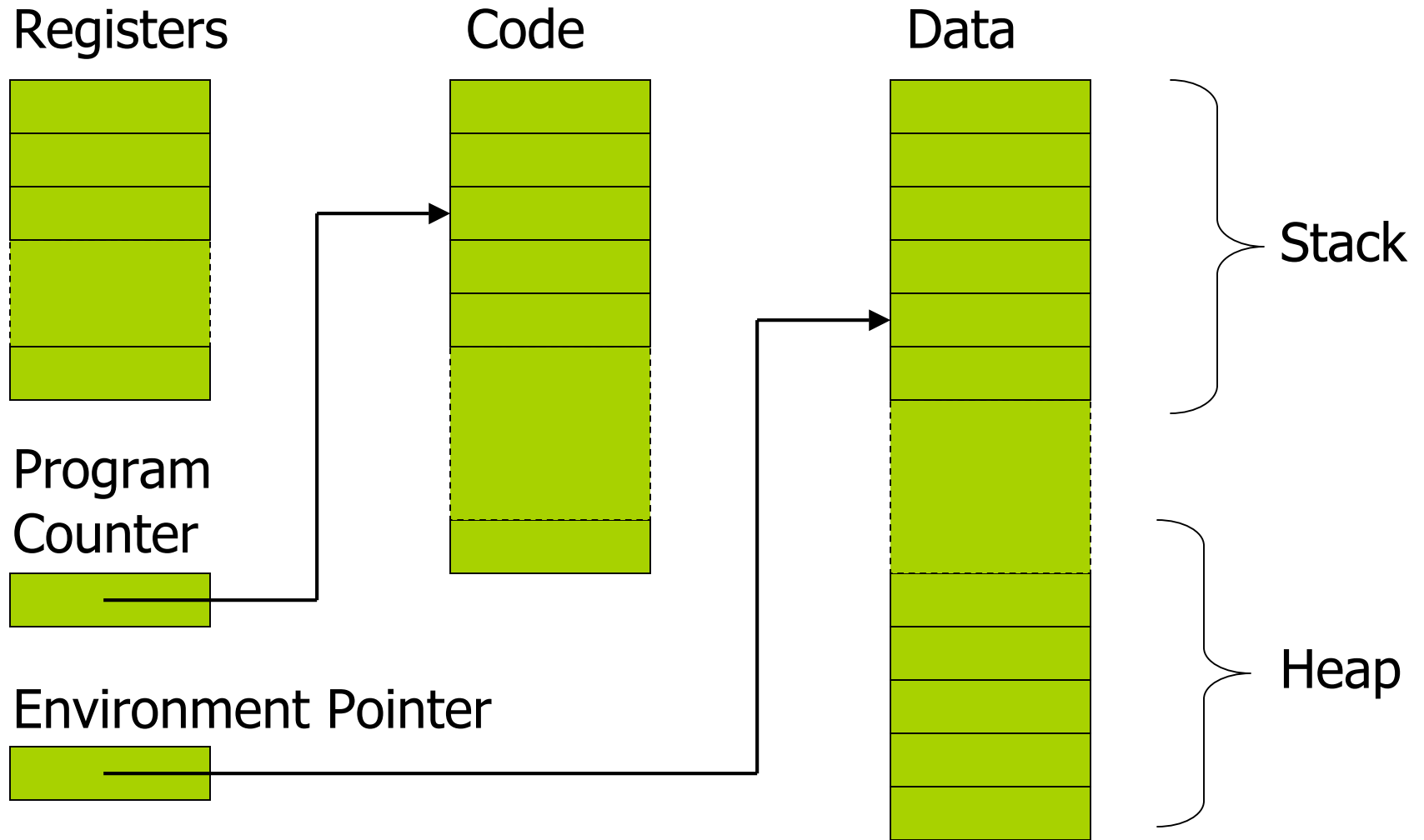
- Build retargetable compilers
 - Isolate machine dependent info
 - Compilers on different machines share a common IR
 - Can have common front and mid ends
 - Table-based back ends share common algorithms
- Table-based instruction selector
 - Create a description of target machine, use back-end generator

The Example Target Machine

- N general-purpose registers r_0, r_2, \dots, r_{N-1}
- Three address instructions: $op \text{ source} \Rightarrow \text{destination}$
 - op : LD, ST, ADD, SUB, MUL, BR, BLTZ, HALT, ...
 - source and destination: constant, register, or memory
 - Use bit patterns to distinguish different address modes
 - All computation operators require both operands to be either constants or in registers

ST $r_0 \Rightarrow M$	Store content of register r_0 into memory M
LD $*a(r_0) \Rightarrow r_1$	Load content of memory $a + \text{content}(r_0)$ to r_1
ST $r_1 \Rightarrow *4(r_0)$	Store content of r_1 to memory indirectly addressed by $4 + \text{content}(r_0)$
ST $*r_0 \Rightarrow M$	Store content indirectly addressed by $\text{content}(r_0)$ to M
LD $1 \Rightarrow r_0$	Load constant integer 1 into register r_0

Simplified Machine Model



Translating from three-address code

- ❑ No more support for structured control-flow
 - Function calls => explicit memory management and goto jumps
- ❑ Every three-address instruction is translated into one or more target machine instructions
 - The original evaluation order is maintained
- ❑ Memory management
 - Every variable must have a location to store its value
 - ❑ Register, stack, heap, static storage
 - Memory allocation convention
 - ❑ Scalar/atomic values and addresses => registers, stacks
 - ❑ Arrays => heap
 - ❑ Global variables => static storage

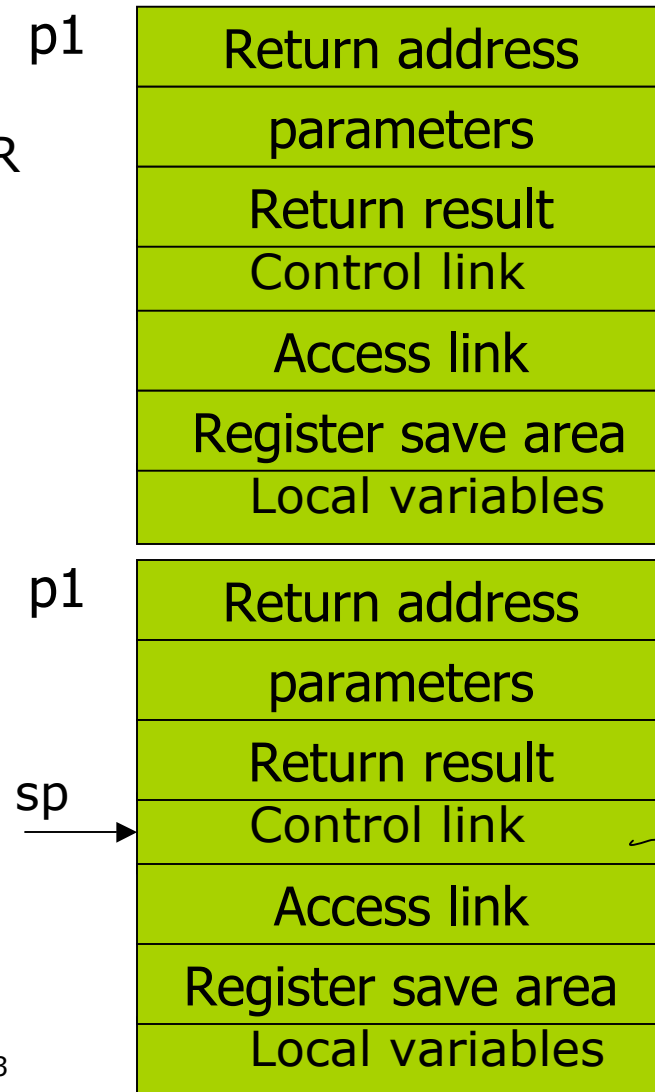
Assigning storage locations

- Compilers must choose storage locations for all values
 - Procedure-local storage
 - Local variables not preserved across procedural calls
 - Procedure-static storage
 - Local variables preserved across procedural calls
 - Global storage --- global variables
 - Run-time heap --- dynamically allocated storage
- Registers---temporary storage for applying operations to values
 - Unambiguous values can be assigned to registers with no backup storage

```
void fee() {  
    int a, *b, c;  
    a = 0;    b = &a;    *b = 1;  
    c = a + *b;  
}
```

Function call and return

- At each function call
 - Allocate an new AR on stack
 - Save return address in new AR
 - Set parameter values and return results
 - Go to callee's code
 - Save SP and other regs; set AL if necessary
- At each function return
 - Restore SP and regs
 - Go to return address in callee's AR
 - Pop callee's AR off stack
- Different langauges may implement this differently
 - Conversion necessary when linking code in different lang.



Translating function calls


- Use a register SP to store addr of activation record on top of stack
 - SP,AL and other registers saved/restored by callee
- Use C(Rs) address mode to access parameters and local variables

```
/* code for s */  
Action1  
Param 5  
Call q, 1  
Action2  
Halt  
  
.....  
/* code for q */  
Action3  
return
```

```
LD stackStart =>SP /* initialize stack*/  
.....  
108: ACTION1  
128: Add SP,ssize=>SP /*now call sequence*/  
136: ST 160 =>*SP /*push return addr*/  
144: ST 5 => 2(SP) /* push param1*/  
152: BR 300 /* call q */  
160: SUB SP, ssize =>SP /*restore SP*/  
168: ACTION2  
190: HALT  
..... /* code for q*/  
300: save SP,AL and other regs  
ACTION3  
restore SP,AL and other regs  
400: BR *0(SP) /* return to caller*/
```

Translating variable assignment

- Keep track of locations for variables in symbol table
 - The current value of a variable may reside in a register, a stack memory location, a static memory location, or a set of these
 - Use symbol table to store locations of variables
- Allocation of variables to registers
 - Assume infinite number of pseudo registers
 - Relocate pseudo registers afterwards

$x := y \text{ op } z$

 LD $y' \Rightarrow r1$
 LD $z' \Rightarrow r2$ where x', y', z' are locations of x, y, z
 OP $r1 \ r2 \Rightarrow r3$
 ST $r3 \Rightarrow x'$

statements	Generated code	Register descriptor	Address descriptor
$t := a - b$	LD $a \Rightarrow r0$ LD $b \Rightarrow r1$ SUB $r0, r1 \Rightarrow r0$	$r0$ contains t $r1$ contains b	t in $r0$ b in $r1$
$u := t + c$	LD $c \Rightarrow r2$ ADD $r0, r2 \Rightarrow r0$	$r0$ contains u $r1$ contains b $r2$ contains c	u in $r0$ b in $r1$ c in $r2$


Translating arrays

Translating Array assignments (arrays are allocated in heap)

Statement	i in register ri	i in memory Mi	i in stack
a := b[i]	Mult ri, elsize=>r1 LD b(r1)=>ra	LD Mi => ri Mult Ri,elsize=>r1 LD b(r1) =>ra	LD i(SP) => ri Mult ri,elsize=>r1 LD b(r1) =>ra
a[i] := b	Mult ri, elsize=>r1 ST rb => a(r1)	LD Mi => ri Mult Ri,elsize=>r1 ST rb => a(r1)	LD i(SP) => ri Mult ri,elsize=>r1 ST rb => a(r1)

Translating conditional statements

If $x < y$ goto z  SUB $rx, ry => rt$
BLTZ z

$X := y + z$
if $(x < 0)$ goto L  ADD $ry, rz => rx$
BLTZ L

Condition determined after ADD or SUB

Example

```
foo(int a,int b) {
  int i = 0;
  if (a>-100 && a<100){
    i = 0;
    while (i < 50) {
      a = a + b *2;
    }
    foo(a,b)
  }
}
```

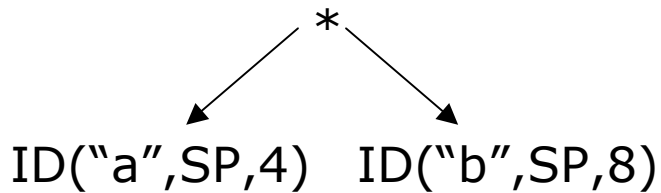
Assumptions:

size of address: 4 bytes
size of int: 2 bytes

```
foo:
  if a>-100 goto L1
  goto done
L1: if a<100 goto L2
  goto done
L2: i := 0
s0: if i < 50 goto s1
  goto s2
s1: t1 := b * 2
  a := a + t1
  goto s0
s2: param a
  param b
  call foo, 2
done: return
```

```
Foo:save SP and regs
  LD a(SP)=>ra
  Sub ra, -100=>ra
  BGTZ L1
  BR done
L1: LD a(SP)=>ra
  Sub ra, 100=>ra
  BLTZ L2
  BR done
L2: LD 0 => ri
  ST ri=>i(SP)
S0: LD i(SP)=>ri
  Sub ri, 50=>ri
  BLTZ S1
  BR S2
S1: LD b(SP)=>rb
  Mul rb, 2 => r1
  LD a(SP)=>ra
  Add ra,r1=> ra
  ST ra=>a(SP)
  BR S0
S2: Add SP, Foosz=>SP
  LD done=>*SP
  ST ra=>4(SP)
  ST rb=>6(SP)
  BR Foo
done: Sub SP,Foosz=>SP
  restore SP and regs
  BR *0(SP)
```

Instruction Selection

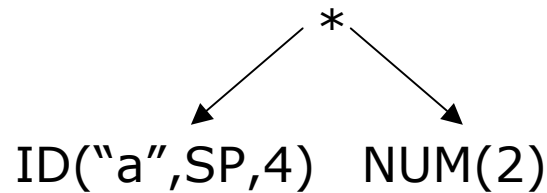


Generated code

```
loadI 4 => r5
loadA0 r5,SP => r6
LoadI 8 => r7
loadA0 r7,SP => r8
Mult r6, r8 => r9
```

Desired code

```
LoadAI SP, 4 => r5
loadAI SP,8 => r6
Mult r5, r6=>r7
```



Generated code

```
loadI 4 => r5
loadA0 r5,SP, => r6
loadI 2 => r7
Mult r6, r7 => r8
```

Desired code

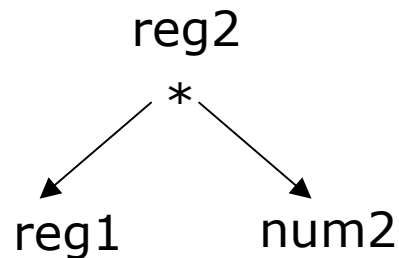
```
LoadAI SP,4 => r5
MultI r5, 2 => r6
```

Based on locations of operands, different instructions may be selected.

Tree-pattern matching

- Define a collection of operation patterns
 - Define a code generation template for each pattern
- Match each AST subtree with an operation pattern
 - Select instructions accordingly

Operation tree:



Prefix notation of operation tree:

```
<-(reg2, *(reg1, num2))
```

Code template:

```
MultI reg1, num2 => reg2
```

```
Example: low-level AST for  $w \leftarrow x - 2 * y$ 
```

Rewrite rules through tree grammar

- Use attributed grammar to define code generation rules
 - Summarize structures of AST through context-free grammar
 - Each production defines a tree pattern in prefix-notation
 - Each production is associated with a cost
 - Each grammar symbol (terminal or non-terminal) has an attribute (location of value)

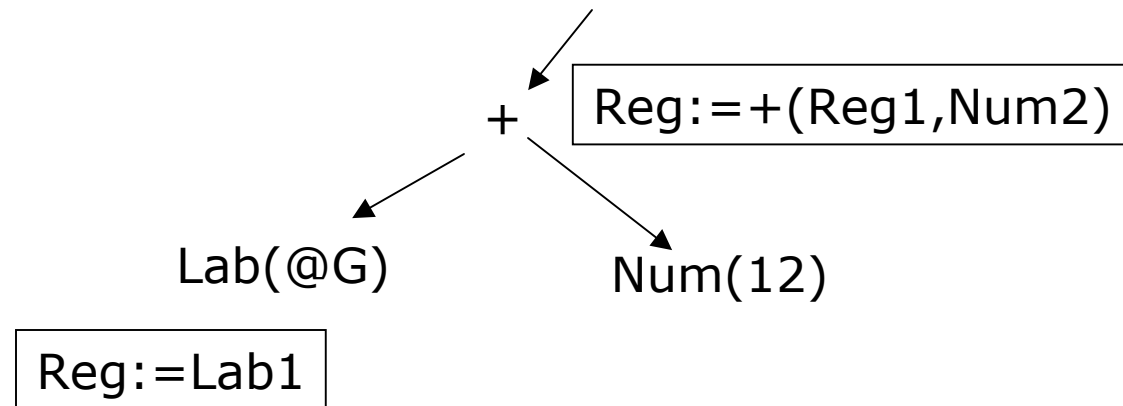
production	cost	Code template
1: Goal := Assign	0	
2: Assign := <- (Reg1, Reg2)	1	move r2 => r1
3: Assign := <- (+ (Reg1, Reg2), Reg3)	1	storeA0 r3 => r1, r2
4: Assign := <- (+ (Reg1, num2), Reg3)	1	storeAI r3 => r1, n2
5: Assign := <- (+ (num1, Reg2), Reg3)	1	storeAI r3 => r2, n1
6: Reg := lab1	1	loadI I1 => rnew
7: Reg := val1	0	
8: Reg := Num1	1	loadI n1 => rnew

Example: applying rewrite rules

production	cost	Code template
9: Reg := M(Reg1)	1	Load r1 => rnew
10: Reg := M(+ (Reg1,Reg2))	1	loadA0 r1, r2 => rnew
11: Reg := M(+ (Reg1,Num2))	1	loadAI r1, n2 => rnew
12: Reg := M(+ (Num1,Reg2))	1	loadAI r2, n1 => rnew
13: Reg := M(+ (Reg1, Lab2))	1	loadAI r1, l2 => rnew
14: Reg := M(+ (Lab1,Reg2))	1	loadAI r2, l1 => rnew
15: Reg := - (Reg1,Reg2)	1	Sub r1 r2 => rnew
16: Reg := - (Reg1, Num2)	1	subI r1, n2 => rnew
17: Reg := +(Reg1, Reg2)	1	add r1, r2=> rnew
18: Reg := + (Reg1, Num2)	1	addI r1, n2 => rnew
19: Reg := + (Num1, Reg2)	1	addI r2, n1 => rnew
20: Reg := + (Reg1, Lab2)	1	addI r1, l2 => rnew
21: Reg := + (Lab1, Reg2)	1	addI r2, l1 => rnew

Tiling the AST

- Given an AST and a collection of operation trees, tiling the AST maps each AST subtree to an operation tree
- A tiling is a collection of <ASTnode, op-tree> pairs, each specifying the implementation for a AST node
 - Storage for result of each AST operation must be consistent across different operation trees



Find a tiling

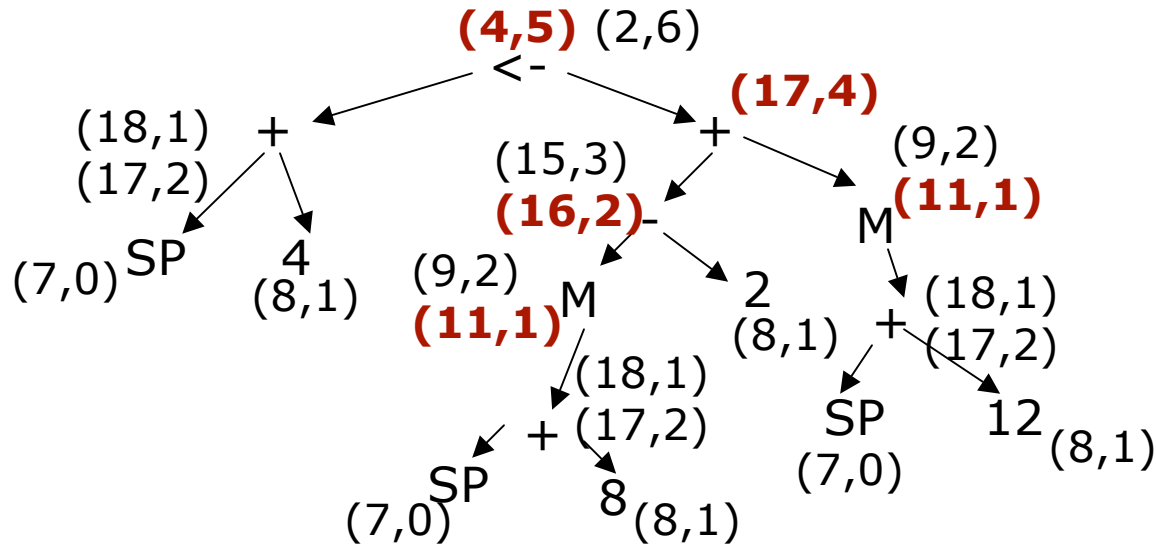
- Bottom-up walk of the AST, for each node n
 - $\text{Label}(n)$ contains the set of all applicable tree patterns

```
Tile(n)
  Label(n) :=  $\emptyset$ 
  if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    for each rule r that matches n's operation
      if left(r)  $\in$  Label(left(n)) and right(r)  $\in$  Label(right(n))
        then Add r to Label(n)
  else if n is a unary node then
    Tile(left(n))
    for each rule r that matches n's operation
      if (left(r)  $\in$  Label(left(n)))
        then Add r to Label(n)
  else /* n is a AST leaf */
    Label(n) := {all rules that match the operation in n}
```

Find the low-cost matches

- Tiling can find all the matches in the pattern set
 - Multiple matches exist because grammar is ambiguous
- To find the one with lowest cost, must keep track of the cost in each matched translation

Example: low-level AST for $w \leftarrow x - 2 + y$



```
loadAI SP,8=>r1
subI r1, 2=> r2
loadAI SP,12=>r3
Add r2, r3 => r4
storeAI r4=>SP, 4
```

Summary of tree matching approach

- ❑ Need to select lowest-cost instructions in bottom-up traversal of AST
 - Need to determine lowest-cost match for each storage class
- ❑ Automatic tools
 - Hand-coding of tree matching
 - Encode the tree-matching problem as a finite automata
 - Use parsing techniques
 - ❑ Need to be extended to handle ambiguity
 - Use string-matching techniques
 - ❑ Linearize the tree into a prefix string
 - ❑ Apply string pattern matching algorithms

Peephole optimization

- Use a simple scheme to match IR to machine code
 - efficiently discover local improvements by examining short sequences of adjacent operations

```
StoreAI r1 => SP, 8  
loadAI SP,8 => r15
```



```
storeAI r1 => SP, 8  
r2r    r1 => r15
```

```
addI r2, 0 => r7  
Mult r4, r7 => r10
```



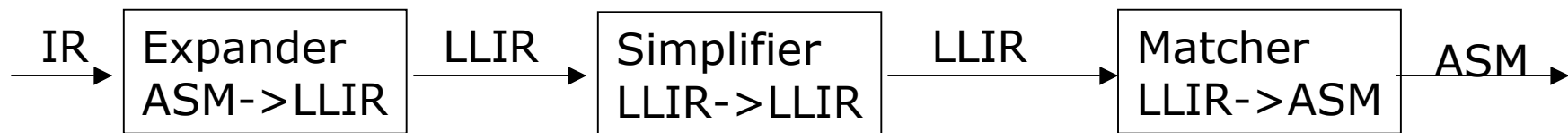
```
Mult r4, r2 => r10
```

```
    jumpI -> L10  
L10: jumpI -> L11
```



```
    jumpI -> L11  
L10: jumpI -> L11
```

Systematic peephole optimization



- Expander
 - Rewrites each assembly instruction to a sequence of low-level IRs that represent all the direct effects of operation
- Simplifier
 - Examine and improve LLIR operations in a small sliding window
 - Forward substitution, algebraic simplification, constant evaluation, eliminating useless effects
- Matcher
 - Match simplified LLIR against pattern library for ASM instructions that best captures the LLIR effects

Peephole optimization example

```

mult 2 y => t1
sub  x t1 => w
    
```

expand

```

r10 := 2
r11 := @G
r12 := 12
r13 := r11 + r12
r14 := M(r13)
r15 := r10 * r14
r16 := -16
r17 := SP + r16
r18 := M(r17)
r19 := M(r18)
r20 := r19 - r15
r21 := 4
r22 := SP + r21
M(r22) := r20
    
```

Optimizations:

```
r1 := n1
```

```
r1 := r2 + n1
```

```
r1 := r2 + n1
```

```
r2 := r3 + r1
```

```
r3 := M(r1)
```

```
M(r1) := r3
```

```
r2 := r3 + n1
```

```
r3 := M(r2 + n1)
```

```
M(r2 + n1) := r3
```

```

r10 := 2
r11 := @G
r14 := M(r11 + 12)
r15 := r10 * r14
r18 := M(SP + -16)
r19 := M(r18)
r20 := r19 - r15
M(SP + 4) := r20
    
```

simplify

cs4713

```

loadI 2 => r10
loadI @G => r11
loadAI r11 12 => r14
Mult r10 r14 => r15
loadAI SP -16 => r18
Load r18 => r19
Sub r19 r15 => r20
storeAI r20 => SP 4
    
```

match

Efficiency of peephole optimization

- Design issues
 - Dead values
 - May intervene with valid simplification
 - Need to be recognized expansion process
 - Control flow operations
 - Complicates simplifier
 - Clear window vs. special-case handling
 - Physical vs. logical windows
 - Adjacent operations may be irrelevant
 - Sliding window includes ops that define or use common values
- RISC vs. CISC architectures
 - RISC architectures makes instruction selection easier
- Additional issues
 - Automatic tools to generate large pattern libraries for different architectures
 - Front ends that generate LLIR make compilers more portable

Register allocation and assignment

- Values in registers are easier and faster to access than memory
 - Reserve a few registers for stack pointers, base registers etc
 - Efficiently utilize the rest of general-purpose registers
- Register allocation
 - At each program point, select a set of values to reside in registers
- Register assignment
 - Pick a specific register for each value, subject to hardware constraints
 - Register classes: not all registers are equal
- Optimal register allocation/assignment in general are NP-complete
 - Register assignment in many cases can be solved in polynomial time

```
.....  
    i := 0  
s0: if i < 50 goto s1  
    goto s2  
s1: t1 := b * 2  
    a := a + t1  
    goto s0  
S2: ...
```

- Variables that can stay in registers
i, a, b, t1
- Need to know how variables will be used after each statement.
- Problem: given a statement I, what statements may follow I in the future?

The register allocation problem

Input program

Assumes infinite #
of registers

**Register
allocator**

Output program

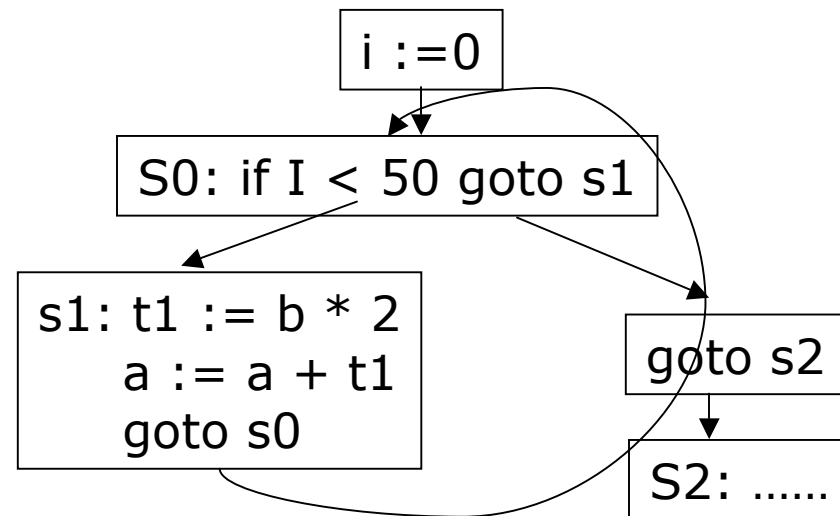
Uses registers on
machine

- ❑ At each point of execution, a program may have arbitrary number of live variables
 - Only a subset may be kept in registers
 - If a value cannot be kept in register, it must be stored in memory and loaded again when next needed → spilling of value to register
- ❑ Goal: make effective use of registers
 - Minimize the number of loads and stores for spilling
- ❑ Register-to-register model
 - Early translation tries to store all values in registers; select values to spill to memory
- ❑ Memory-to-memory model
 - Early translation allocates memory for all user variables; promote values to register
 - ❑ Must decide which values do not require memory storage

Control-flow graph

- Graph representation of program
 - Nodes of graph: basic blocks (straight-line computations)
 - Edges of graph: flows of control
- Useful for collecting information about computation
 - Detect loops, remove redundant computations, ...
 - Find live range of each variable v
 - All statements where v might be used in the future
 - Candidate for register allocation

```
.....  
  i := 0  
s0: if i < 50 goto s1  
    goto s2  
s1: t1 := b * 2  
    a := a + t1  
    goto s0  
S2: ...
```



Identifying basic blocks

- ❑ Input: a sequence of three-address statements
- ❑ Output: a list of basic blocks
- ❑ Method:
 - Determine each statement that starts a new basic block, including
 - ❑ The first statement of the input sequence
 - ❑ Any statement that is the target of a goto statement
 - ❑ Any statement that immediately follows a goto statement
 - Each basic block consists of
 - ❑ A starting statement S_0
 - ❑ All statements following S_0 up to but not including the next starting statement (or the end of input)

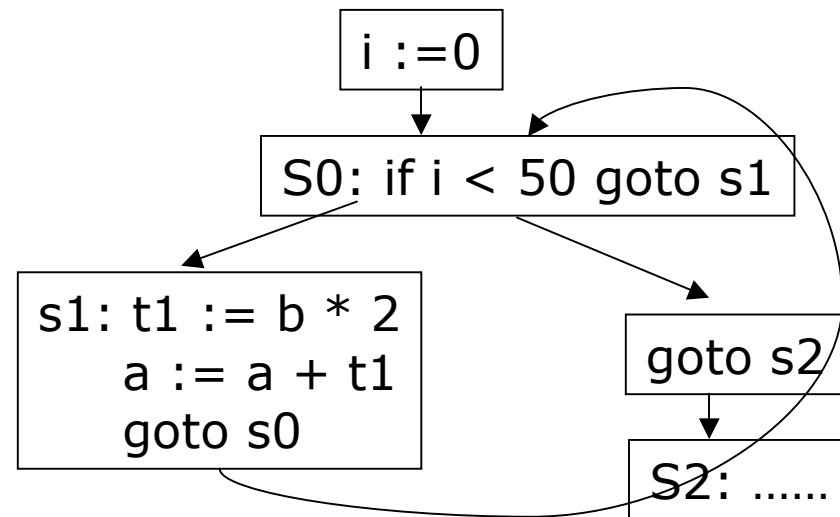
```
.....  
[ i := 0  
[ s0: if i < 50 goto s1  
[   goto s2  
[ s1: t1 := b * 2  
[   a := a + t1  
[   goto s0  
[ S2: ...
```

Starting statements:
i := 0
S0,
goto S2
S1,
S2

Building control-flow graphs

- Identify all the basic blocks
 - Create a flow graph node for each basic block
- For each basic block B1
 - If B1 ends with a jump to a statement that starts basic block B2, create an edge from B1 to B2
 - If B1 does not end with an unconditional jump, create an edge from B1 to the basic block that immediately follows B1 in the original evaluation order

```
.....  
[ i := 0  
[ s0: if i < 50 goto s1  
[   goto s2  
[ s1: t1 := b * 2  
[   a := a + t1  
[   goto s0  
[ S2: ...
```



Register allocation on flow graphs

- Local (single basic block) register allocation
 - At start of basic block
 - Assume every value resides in memory
 - Load values from memory to registers if necessary
 - At end of basic block
 - Store all modified values in registers back to memory
 - Within each basic block,
 - Find live ranges of variables (statements where variables are further used in basic block)
 - Allocate variable live ranges to registers based on use counts

- Global (single procedural) register allocation
 - Allocate registers across basic block boundaries
 - Compute the live range of each variable
 - The duration of code (the collection of basic blocks) that variables are alive (may be used in the future)
 - Use data-flow analysis on control-flow graphs (not covered)
 - Allocate registers to live ranges of variables

Local register allocation

- Allocation model
 - Assumes register-to-register memory model
 - Input program assumes infinite # of registers
 - Assume all registers on target machine are equivalent
- Approaches
 - Top-down: count the number of references to each value
 - the most heavily used values should reside in registers
 - Weakness: dedicate a register to value for entire block
 - Bottom-up: spill the value that is needed the latest
 - For each variable use, compute the distance of its next use
 - process each instruction in evaluation order; when running out of registers, spill the value whose next use is farthest in the future
 - Produces excellent result in many cases
 - Not optimal: not all spilling takes the same number of cycles
 - Clean vs. dirty spill: has the variable been modified?
 - Graph Coloring based allocation

Computing local live variables

- At each program point, a variable is alive if it may be used in the future
 - Only values of live variables need to be kept in registers
- Local live variable analysis (within a basic block)
 - A variable is alive if its value is used within the basic block

Algorithm at each basic block:

- Set every variable in symbol table ``not alive''
- Scan statements in reverse order, at every i : $x := y \text{ op } z$
 - Alive(i) = live variables in symbol table
 - Set x to "not alive" in symbol table
 - Set y and z to "alive" in symbol table

Local live variables

	a, b
(1) t1 := a * a	t1, a, b
(2) t2 := a * b	t1, b, t2
(3) t3 := 2 * t2	t1, t3, b
(4) t4 := t1 + t3	t4, b
(5) t5 := b * b	t4, t5
(6) t6 := t4 + t5	none

Live ranges and interference graph

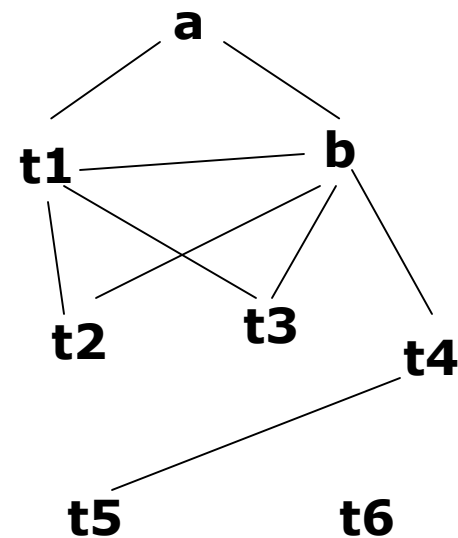
Live variables

	a, b
(1) $t1 := a * a$	t1, a, b
(2) $t2 := a * b$	t1, b, t2
(3) $t3 := 2 * t2$	t1, t3, b
(4) $t4 := t1 + t3$	t4, b
(5) $t5 := b * b$	t4, t5
(6) $t6 := t4 + t5$	none

variable	live range	# of uses
a	(1)-(2)	3
b	(1)-(5)	3
t1	(2)-(4)	2
t2	(3)	1
t3	(4)	1
t4	(5)-(6)	1
t5	(6)	1
t6	none	0

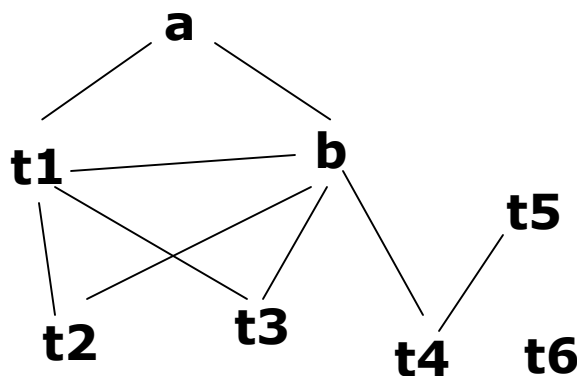
Interference graph:

- **Nodes:** live ranges of variables
- **Put an edge between (n1,n2) if they are overlapping live ranges (values are alive simultaneously)**



Interference graph and register allocation

- ❑ Interference graph
 - Nodes: live ranges of variables
 - An edge between $(n1, n2)$ if they are overlapping live ranges
- ❑ Register allocation
 - Allocating registers to nodes of interference graph
 - If there is an edge between $(n1, n2)$
 - ❑ Must allocate $n1$ and $n2$ to different registers
 - If there is no edge between $(n1, n2)$, they can share a register
- The graph coloring problem
 - Assign colors to nodes of a graph, neighboring nodes must have different colors



a, b, t1 must reside in different registers
b, t1, t3 must reside in different registers
b, t1, t2 must reside in different registers
t2, t3, t4, t5, t6 can share a single register

Need 4 registers to hold all values

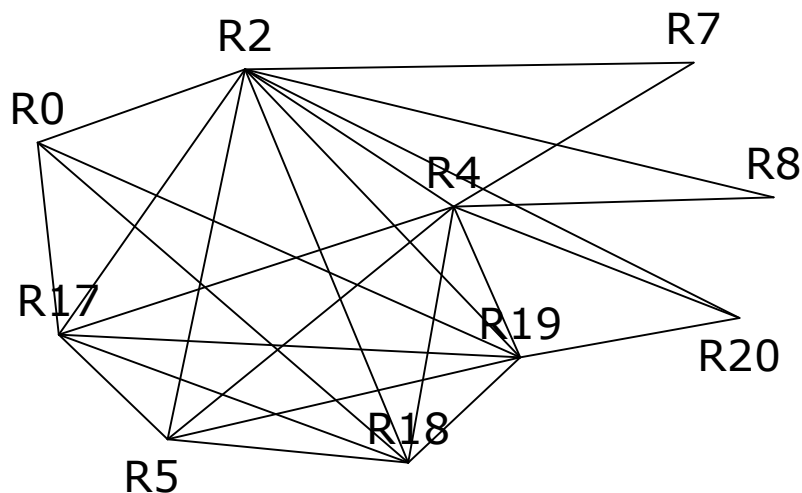
Estimating register spilling cost

$$\text{Cost} = (\text{address calculation} + \text{memory load/store}) * \text{frequency}$$

- When not sufficient registers are available, compilers must choose registers to spill into memory
 - Choose the variables with the lowest spilling cost
- Address calculation --- where to spill
 - Compilers can choose where to spill values
 - E.g. Register-save area of local activation record
- Memory load/store
 - Negative spill costs
 - live ranges that contain a single load /store and no other uses
 - Infinite spill costs
 - live ranges short enough that spilling never helps
 - E.g., a use immediately following a definition
- Global allocation ==> frequency of basic block execution
 - Compilers annotate each block with an execution count
 - E.g., assume each loop executes 10 times, and each unpredictable branch is evaluated 50% of times

Graph-coloring based allocation

- Rank all live ranges
 - Live ranges with high spilling costs are ranked higher
 - Color constrained live ranges first
 - Live ranges with more than k interfering neighbors
 - Unconstrained live ranges can always be colored
- At each step, try to color the current live range R_i with top priority
 - if neighbors of R_i have not taken all the colors
assign an available color (register) to R_i
else /*no color is available for R_i */
invoke spilling or splitting mechanisms



Assume 5 physical registers: P1-P5

Unconstrained nodes:

R0, R7, R8, R20

Ordering of nodes for coloring

R5 → P1; R2 → P2; R4 → P3;

R17 → P4; R18 → P5; R19 → spill

R0 → P1; R7 → P1; R8 → P1;

R20 → P1;

The register allocation problem

- Local register allocation
 - Allocate registers with a single basic block
 - Load all registers from memory at block entry; store all registers to memory at block exit
- Global register allocation
 - Allocating registers across basic block boundaries
 - Apply data-flow analysis on control-flow graph to determine live ranges of variables
 - Build global interference graph and apply graph coloring algorithm
- Register allocation is hard
 - Optimal graph coloring is NP complete
 - Building global interference graph and applying graph coloring algorithms are expensive
 - Not suitable for just-in-time compilers
 - When not enough registers, need to spill values to memory
- Heuristics for register allocation
 - Allocate registers to values that are used more times
 - Avoid register spilling in loops
 - Just in time compilation
 - Aggressively allocate registers in a linear scan of program