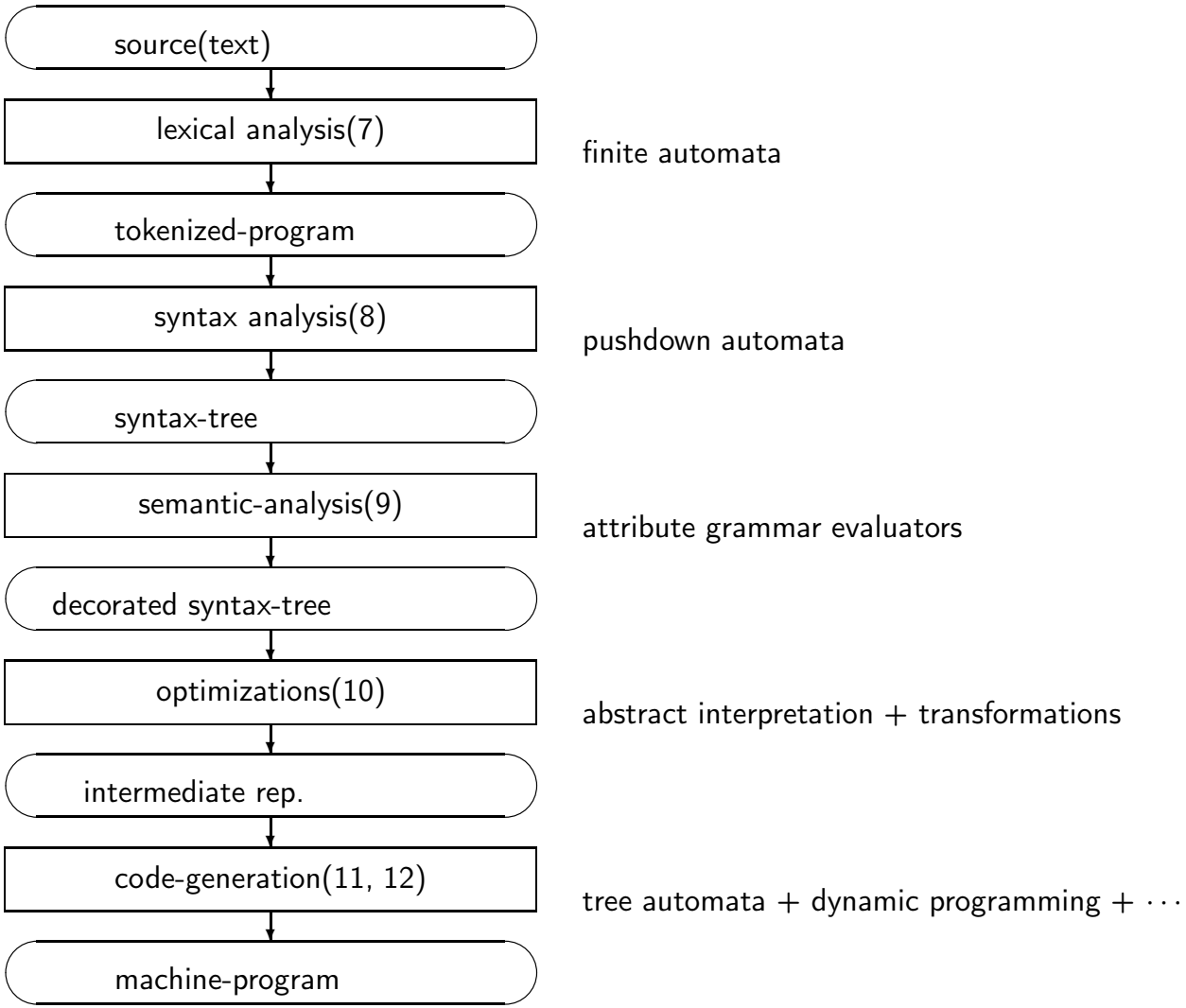
Code Generation

– Wilhelm/Maurer: Compiler Design, Chapter 12 –

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de
and
Mooly Sagiv
Tel Aviv University

11. Januar 2010

"Standard" Structure



Code Generation

Real machines (instead of abstract machines):

- Register machines,
- Limited resources (registers, memory),
- Fixed word size,
- Memory hierarchy,
- Intraprocessor parallelism.

Architectural Classes: CISC vs. RISC

CISC IBM 360, PDP11, VAX series, INTEL 80x86, Pentium, Motorola 680x0

- A large number of addressing modes
- Computations on stores
- Few registers
- Different instruction lengths
- Different execution times for instructions
- Microprogrammed instruction sets

RISC Alpha, MIPS, PowerPC, SPARC

- One instruction per cycle (with pipeline for load/stores)
- Load/Store architecture – Computations in registers (only)
- Many registers
- Few addressing modes
- Uniform lengths
- Hard-coded instruction sets
- Intra-processor parallelism: Pipeline, multiple units, Very Long Instruction Words (VLIW), Superscalarity, Speculation

Phases in code generation

Code Selection: selecting semantically equivalent sequences of machine instructions for programs,

Register Allocation: exploiting the registers for storing values of variables and temporaries,

Code Scheduling: reordering instruction sequences to exploit intraprocessor parallelism.

Optimal register allocation and instruction scheduling are NP-hard.

Phase Ordering Problem

Partly contradictory optimization goals:

Register allocation: minimize number of registers used $\Longrightarrow$ reuse registers,

Code Scheduling: exploit parallelism $\Longrightarrow$ keep computations independent, no shared registers

Issues:

- Software Complexity
- Result Quality
- Order in Serialization

Challenges in real machines: CISC vs. RISC

CISC IBM 360, PDP11, VAX series, INTEL 80x86, Motorola 680x0

- A large number of addressing modes
- Computations on stores
- Few registers
- Different instruction lengths
- Different execution times for instructions
- Microprogrammed instruction sets

RISC Alpha, MIPS, PowerPC, SPARC

- One instruction per cycle (with pipeline for load/stores)
- Load/Store architecture – Computations in registers (only)
- Many registers
- Few addressing modes
- Uniform lengths
- Hard-coded instruction sets
- Intra-processor parallelism: Pipeline, multiple units, Very Long Instruction Words (VLIW), Superscalarity, Speculation

Example: $x = y + z$

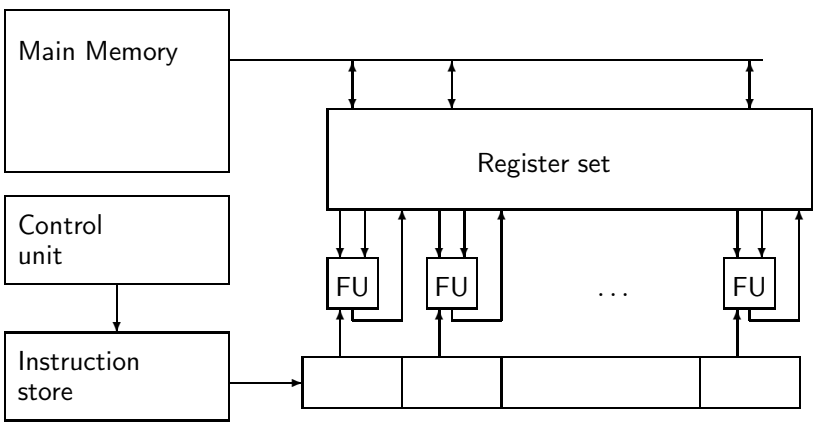
CISC/Vax **addl3** $4(fp), 6(fp), 8(fp)$

RISC

load $r_1, 4(fp)$
load $r_2, 6(fp)$
add r_1, r_2, r_3
store $r_3, 8(fp)$

The VLIW Architecture

- Several functional units,
- One instruction stream,
- Jump priority rule,
- FUs connected to register banks,
- Enough parallelism available?



Instruction Pipeline

Several instructions in different states of execution
Potential structure:

1. instruction fetch and decode,
2. operand fetch,
3. instruction execution,
4. write back of the result into target register.

		cycle						
		1	2	3	4	5	6	7
Pipe-	1	B_1	B_2	B_3	B_4			
line-	2		B_1	B_2	B_3	B_4		
stage	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

Pipeline hazards

- Cache hazards: Instruction or operand not in cache,
- Data hazards: Needed operand not available,
- Structural hazards: Resource conflicts,
- Control hazards: (Conditional) jumps.

Program Representations

- Abstract syntax tree: algebraic transformations, code generation for expression trees,
- Control Flow Graph: Program analysis (intraproc.)
- Call Graph: Program analysis (interproc.)
- Static Single Assignment: optimization, code generation
- Program Dependence Graph: instruction scheduling, parallelization
- Register Interference graph: register allocation

Code Generation: Integrated Methods

- Integration of register allocation with instruction selection,
- Machine with interchangeable machine registers,
- Input: Expression trees
- Simple target machines.
- Two approaches:
  1. Ershov[58], Sethi&Ullman[70]: unique decomposition of expression trees,
  2. Aho&Johnson[76]: dynamic programming for more complex machine models.

Contiguous evaluation

(Sub-)expression can be evaluated into

register: this register is needed to hold the result,

memory cell: no register is needed to hold the result.

Contiguous evaluation of an expression, thus, needs 0 or 1 registers while other (sub-)expressions are evaluated.
Evaluate-into-memory-first strategy: evaluate subtrees into memory first.
Contiguous evaluation + evaluate-into-memory-first define a normal form for code sequences.
Theorem (Aho&Johnson[76]): Any optimal program using no more than r registers can be transformed into an optimal one in normal form using no more than r registers.

Simple machine model, Ershov[58], Sethi&Ullman[70]

- r general purpose nterchangeable registers $R_0, \ldots, R_{r-1}$,
- Two-address instructions

R_i	:=	$M[V]$	Load
$M[V]$	:=	R_i	Store
R_i	:=	R_i *op* $M[V]$	Compute
R_i	:=	R_i *op* R_j	

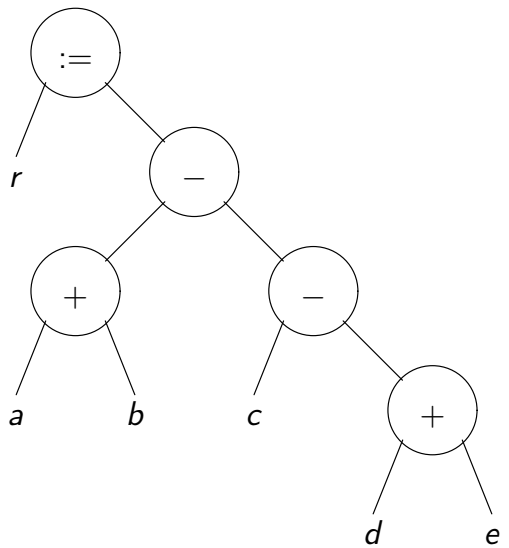
Two phases:

1. Computing register requirements,
2. Generating code, allocating registers and temporaries.

Example Tree

Source $r := (a + b) - (c - (d + e))$

Tree



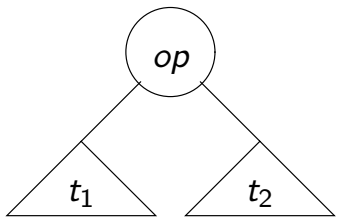
Generated Code

2 Registers R_0 and R_1

Two possible code sequences:

R_0	$:=$	$M[a]$	R_0	$:=$	$M[c]$
R_0	$:=$	$R_0 + M[b]$	R_1	$:=$	$M[d]$
R_1	$:=$	$M[d]$	R_1	$:=$	$R_1 + M[e]$
R_1	$:=$	$R_1 + M[e]$	R_0	$:=$	$R_0 - R_1$
$M[t_1]$	$:=$	R_1	R_1	$:=$	$M[a]$
R_1	$:=$	$M[c]$	R_1	$:=$	$R_1 + M[b]$
R_1	$:=$	$R_1 - M[t_1]$	R_1	$:=$	$R_1 - R_0$
R_0	$:=$	$R_0 - R_1$	$M[f]$	$:=$	R_1
$M[f]$	$:=$	R_0			
stores result for $c - (d + 2)$			evaluates $c - (d + 2)$ first		
in a temporary			(needs 2 registers)		
no register available			saves one instruction		

The Algorithm



Principle: Given tree t for expression e_1 *op* e_2
t_1 needs r_1 registers, t_2 needs r_2 registers,

$r \geq r_1 > r_2$: After evaluation of t_1:
$r_1 - 1$ registers freed, one holds the result,
t_2 gets enough registers to evaluate, hence
t can be evaluated in r_1 registers,

$r_1 = r_2$: t needs $r_1 + 1$ registers to evaluate,

$r_1 > r$ or $r_2 > r$: spill to temporary required.

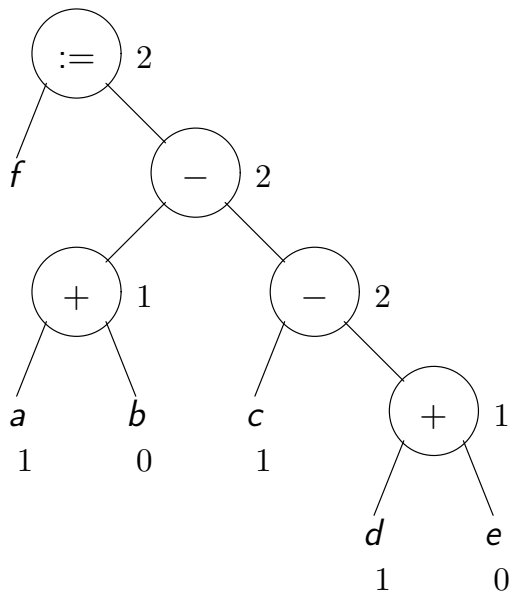
Labeling Phase

- Labels each node with its register needs,
- Bottom-up pass,
- Left leaves labeled with '1' have to be loaded into register,
- Right leaves labeled with '0' are used as operands,
- Inner nodes:

$$regneed(op(t_1, t_2)) = \begin{cases} max(r_1, r_2), & \text{if } r_1 \neq r_2 \\ r_1 + 1, & \text{if } r_1 = r_2 \end{cases}$$

where $r_1 = regneed(t_1)$, $r_2 = regneed(t_2)$

Example



Generation Phase

Principle:

- Generates instruction **Op** for operator *op* in $op(t_1, t_2)$ after generating code for t_1 and t_2.
- Order of t_1 and t_2 depends on their register needs,
- The generated **Op**–instruction finds value of t_1 in register,
- *RSTACK* holds available registers, initially all registers,
  Before processing t: *top(RSTACK)* is determined as result register for t,
  After processing t: all registers available,
  but *top(RSTACK)* is result register for t.
- *TSTACK* holds available temporaries.

Algorithm Gen_Opt_Code

Algorithm	*RSTACK*-Contents result register
var *RSTACK*: **stack of register**;	
var *TSTACK*: **stack of address**;	
proc *Gen_Code* (t : **tree**);	$(R', R'', \ldots)$
var R: **register**, T: **address**;	
case t **of**	
$(leaf\ a, 1)$: (∗left leaf∗)	
$emit(top(RSTACK) := a)$;	result in R'
$op((t_1, r_1), (leaf\ a, 0))$: (∗right leaf∗)	
$Gen\_Code(t_1)$;	
$emit(top(RSTACK) := top(RSTACK)$ **Op** $a)$;	result in R'

$op((t_1, r_1), (t_2, r_2))$:	$(R', R'', \ldots)$
cases	
$r_1 < min(r_2, r)$:	$(R', R'', \ldots)$
begin	
exchange$(RSTACK)$;	$(R'', R', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$R :=$ **pop**$(RSTACK)$;	$(R', \ldots)$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) := top(RSTACK)$ **Op** $R)$;	result in R'
push$(RSTACK, R)$;	$(R'', R', \ldots)$
exchange$(RSTACK)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r_2 \wedge r_2 < r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_1)$;	result in R'
$R :=$ **pop**$(RSTACK)$;	$(R'', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$emit(R := R$ **Op** $top(RSTACK))$;	result in R'
push$(RSTACK, R)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r \wedge r_2 \geq r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_2)$;	result in R'
$T :=$ **pop**$(TSTACK)$;	
$emit(M[T] := top(RSTACK))$;	result in $M[T]$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) := top(RSTACK)$ **Op** $M[T])$;	result in R'
push$(TSTACK, T)$;	
end ;	
endcases	
endcase	
endproc	

Dynamic Programming, Aho&Johnson[76]

- More complex architecture,
  - r general purpose registers $R_0, \ldots, R_{r-1}$,

R_i	$:=$	e	Compute
R_i	$:=$	$M[V]$	Load
$M[V]$	$:=$	R_i	Store

  - Instruction formats:

    e term with register and memory-cell operands,
    costs $c(I)$ associated with each instruction I.
- Goal: Generate a cheapest instruction sequence using no more than r registers.

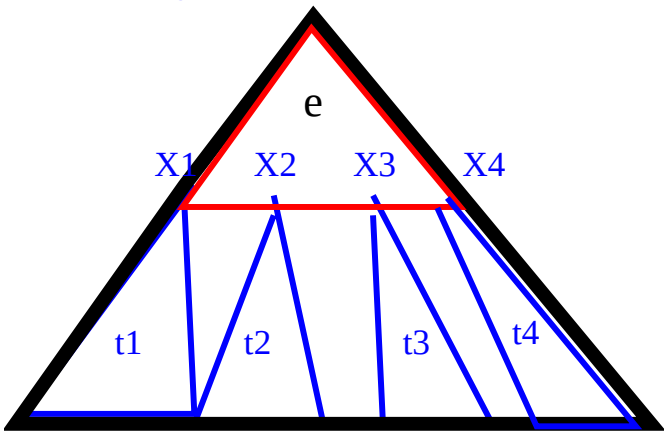
Canonical recursive solution

- Assume e of instruction $R_i := e$ matches tree t, and j registers are available
- some subtrees of t corresp. to memory operands of e – computed into memory first, no registers occupied after that
- let e have k register operands; compute corresponding subtrees $t_1, t_2, \ldots, t_k$ into registers:
- for all permutations $i_1, i_2, \ldots, i_k$ for the evaluation of $t_1, t_2, \ldots, t_k$: generate optimal code for t_{i_1} using no more than j registers, t_{i_2} with no more than $j-1, \ldots,$ t_k with no more than $j-k+1$
  add the minimal costs for computing all subtrees in this way to the costs of e to yield the minimal costs for this combination
- Doing it for all potential combinations recomputes the costs for subtrees $\Longrightarrow$ exponential complexity

Dynamic Programming, the Principle

- Partition the problem into subproblems, here code generation for an expression into code generation for subexpressions,
- combine optimal solutions to the subproblems to optimal solution of the problem.

How to obtain a partition? — by tree parsing: a match of the expression decomposes the tree into
the pattern and a sequence of subtrees, some to be computed into memory, the others into registers.



Optimal Solutions

There may be several optimal solutions for subexpressions using different number of registers!
Therefore, explore all permutations for the subtrees to be computed into registers.

Dynamic Programming

- Convert top-down algorithm into bottom-up algorithm tabulating partial solutions
- Associate cost vector $C[0..r]$ with each node n,
  C[0] cheapest costs for computing t/n into a temporary,
  C[i] cheapest costs computing t/n into a register using i registers.
- Compute cost vector at n minimizing over all "legal" combinations of
  - one applicable instruction,
  - the cost vectors of the nodes "under" non–terminal nodes in the applied rule.
- What is a legal combination for $C[j], j > 0$?
  A combination of generated code for subtrees needing $\leq j$ registers.
- Extract cheapest instruction sequence in a second pass.

Global Register Allocation

So far, register allocation for assignments.
Now, register allocation across whole procedures/programs,
Tasks of the Register Allocator:

1. determine candidates, i.e., variables and intermediate results, called Symbolic Registers, to keep in real registers, and determine their "life spans".
2. assign symbolic registers without "collisions" to real registers using some optimality criterion,
3. modify the code to implement the decisions.

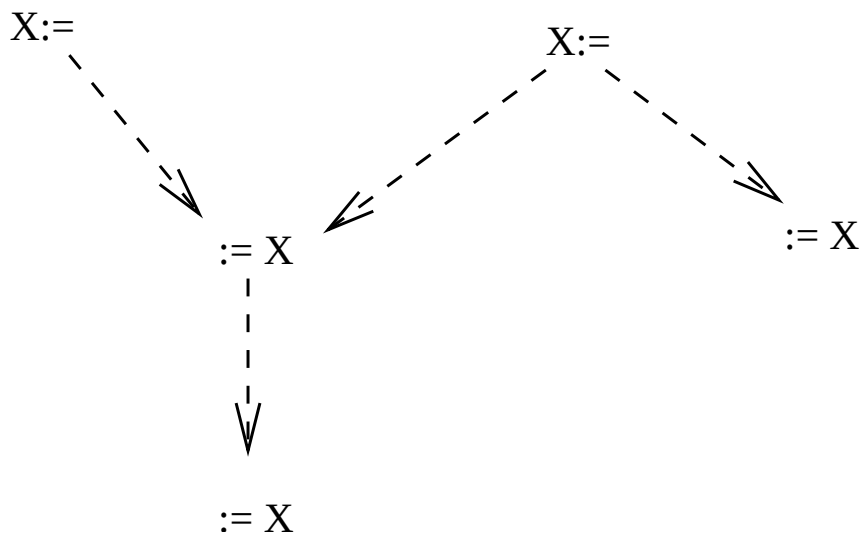
Constraint for assignment:

- Two symbolic registers collide if their contents are "live" at the same time,
- Colliding symbolic registers cannot be allocated to the same real register.

Definitions

- A definition of a symbolic register is the computation of an intermediate result or the modification of a variable,
- A use of a symbolic register is a reading access to the corresponding variable or a use of the intermediate value, Note: uses of symbolic registers in an individual computation step, e.g. execution of an instruction or of an assignment precede definitions of symbolic registers.
- A definition–path of s to program point p is a path from the entry point of the program to p containing a definition of s,
- A use–path from p is a definition-free path starting at p containing a use of s,
- Symbolic register s is live at program point p if exists a definition–path to p and a use–path from p,

- The life span of s is the set of all program points, on which s is live.
  Value of a live symbolic register may still be used.
- Two life spans of symbolic registers collide if one of the registers is set in the life span of the other.



A life span for variable X

Computation of life ranges

Needs du (definition-use) chains.
A du (definition-use) chain connects a definition of a variable to all the associated uses, i.e., uses that a value set at the definition may flow to.
Two du chains are use-connected iff they share a use.
One could say, shared uses were vel-defined[1].
A life range of a variable is the connected component of all use-connected du chains of that variable.

[1] Thanks to Raimund Seidel

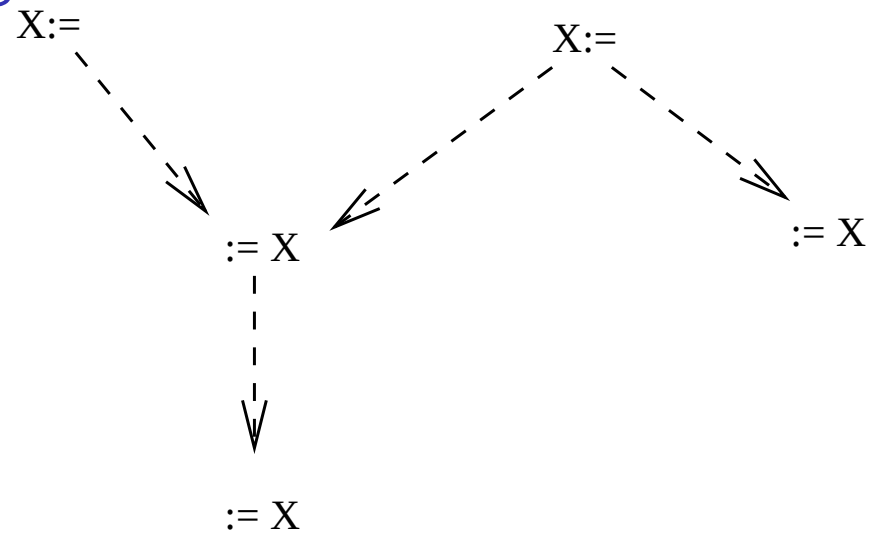
Register Interference Graph

- nodes – life spans,
- edge between colliding life spans.

Allows to view the register-allocation problem as a graph coloring problem.

- k physical registers available,
- Solve k–coloring problem,
- NP–complete for $k > 2$,
- Use heuristics.

Algorithm



Build constructs the register interference graph G,

Reduce initializes an empty stack;
repeatedly removes locally colorable nodes and pushes them onto the stack.
Continue at Assign Colours, if arrived at the empty graph: G is k-colorable
Continue at Spill if locally uncolorable nodes remain in the graph.

Algorithm cont'd

Assign Colours pops nodes from the stack,
reinserts them into the graph,
and assigns a color not assigned to any neighbour.

Spill uses heuristics to select one node (variable) to spill to memory,
inserts a **load** before each use of the variable and a **store** after each definition.
Then continues with Build.

The classical method by Chaitin uses $degree(n) < k$ as local-colorability criterion.
It means, n and its neighbours can be colored with different colors.

Properties

- Assign Colours pops nodes off the stack in reverse order as Reduce pushed them onto the stack.
- The $degree(n) < k$ criterium holding, when n was pushed, guarantees colorability.
- Termination:
  Reduce repeatedly removes nodes from the finite set of nodes; each cycle through Spill reduces the graph by 1 node.

Heuristics for Node Removal

1. degree of the node: high degree causes many deletions of edges,
2. costs of spilling.

Example

Input-program

```
x := 1
y := 2
w := x + y
u := y + 2
z := x * y
x := u + z
print x,z,u
```

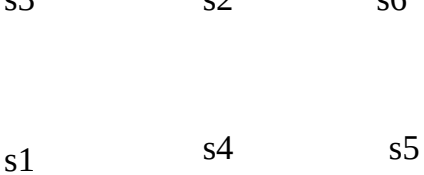
Example

Input-program	Symbolic Reg. Assign.
x := 1	s1 := 1
y := 2	s2 := 2
w := x + y	s3 := s1 + s2
u := y + 2	s4 := s2 + 2
z := x * y	s5 := s1 * s2
x := u + z	s6 := s4 + s5
print x,z,u	print s6,s5,s4

Example

Input-program	Symbolic Reg. Assign.
x := 1	s1 := 1
y := 2	s2 := 2
w := x + y	s3 := s1 + s2
u := y + 2	s4 := s2 + 2
z := x * y	s5 := s1 * s2
x := u + z	s6 := s4 + s5
print x,z,u	print s6,s5,s4

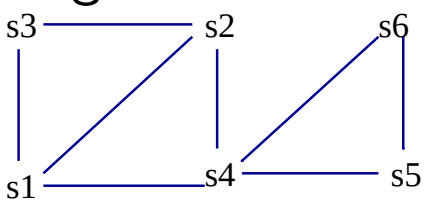
Register interference graph



Example

Input-program	Symbolic Reg. Assign.
x := 1	s1 := 1
y := 2	s2 := 2
w := x + y	s3 := s1 + s2
u := y + 2	s4 := s2 + 2
z := x * y	s5 := s1 * s2
x := u + z	s6 := s4 + s5
print x,z,u	print s6,s5,s4

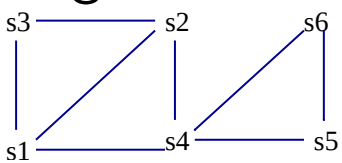
Register interference graph



Example

Input-program	Symbolic Reg. Assign.	After Register Allocation
x := 1	s1 := 1	r1 := 1
y := 2	s2 := 2	r2 := 2
w := x + y	s3 := s1 + s2	r3 := r1 + r2
u := y + 2	s4 := s2 + 2	r3 := r2 + 2
z := x * y	s5 := s1 * s2	r1 := r1 + r2
x := u + z	s6 := s4 + s5	r2 := r3 + r1
print x,z,u	print s6,s5,s4	print r2,r1,r3

Register interference graph



Problems

Architectural irregularities:

- not every physical register can be allocated to every symbolic register,
- some symbolic registers need combinations of physical registers, e.g. pairs of aligned registers.

Dedication: Some registers are dedicated for special purposes, e.g. transfer of arguments.

Extensions

Remember: An edge in the interference graph means:
the connected objects can not be allocated to the same physical register.
Assume, that physical register r can not be allocated to symbolic register s.
Solution: Add nodes for physical registers to the interference graph; connect r with s.
Disadvantage: Graph now describes program-specific constraints ($s1$ and $s2$ live at the same time) and architecture-specific constraints (fixed-point operands should not be allocated to floating-point registers).

Separating Architectural and Program Constraints

Machine description:

- **Regs** register names,
- **Conflict** relation on Regs,
  $(r_1, r_2) \in \mathit{Conflict}$ iff r_1 and r_2 can not be allocated simultaneously.
  Example: registers and register pairs containing them.
- **Class** Subsets of registers
  - required as operands of instructions, or
  - dedicated for special purposes of the run-time system

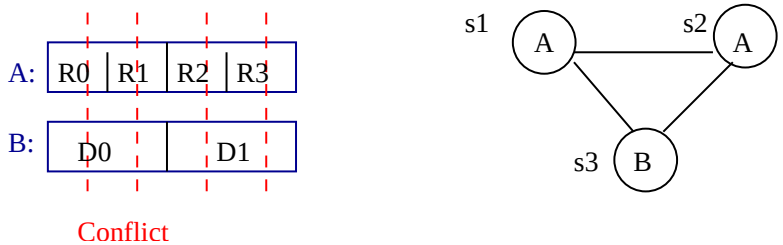
Constraints on allocation (connection between symb. and phys. registers)

- Association of register classes with symbolic registers
- Conjunction of constraints $\Longrightarrow$ intersection of register classes is new register class.

Generalized Interference Graph

extended by assoc. register classes to symbolic registers.
Assignment for $S \subseteq \mathit{SymbRegs}$ is $A : S \mapsto \mathit{Regs}$ such that $A(s) \in \mathit{class}(s)$ for all $s \in S$.
New local colorability criterion:
$s \in S \subseteq \mathit{SymbRegs}$ is locally colorable iff
for all assigments A of the neighbours of s
there exists a register $r \in \mathit{class}(s)$
that does not conflict with the assignment on any neighbour.

Coloring the Generalized Interference Graph



Register classes with conflicts and generalized interference graph.

s1 and s2 are locally colorable, s3 is not.
Old local-colorability criterion is satisfied, *degree* = 2 for all three symb. registers.

Efficient Approximative Test for Local Colorability

Let A, B be two register classes.

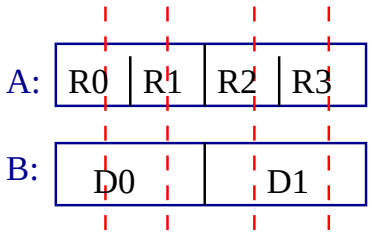
$$maxConflict_A(B) = \max_{a \in A} |\{b \in B | (a, b) \in Conflict\}|$$

maximal numer of registers in B, that a single register in A can conflict with.
Approximative colorability test for s with $class(s) = B$:

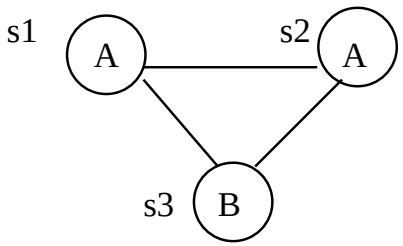
$$\sum_{(s,s') \in E, class(s') = A} maxConflict_A(B) < |B|$$

Precompute $maxConflict_A(B)$ for all A and B,
depends only on the architecture!

Example



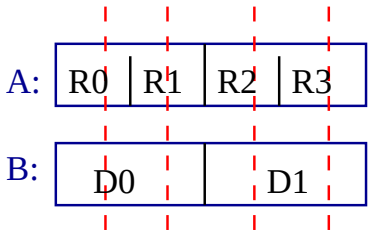
Conflict



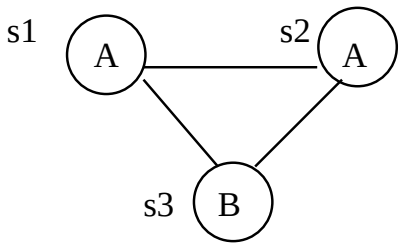
Tabulating $maxConflicts_C(D)$

$C \backslash D$	A	B
A		
B		

Example



Conflict



Tabulating $maxConflicts_A(B)$

$C \backslash D$	A	B
A	1	1
B	2	1